So far we have talked about
Scientific Debugging

Hypothesis

Prediction

Experiment

Observation + Conclusion

Failing Test

Code

Test

More Tests

Hypothesis is supported:
refine hypothesis

Hypothesis is rejected:
create new hypothesis

Diagnosis
The Problem Lifecycle
Steps In Debugging

1. Locate error
2. Design error repair
3. Repair error
4. Re-test program
The Traffic Principle

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
Tools for automatic debugging

A variety of tools and techniques is available to automate debugging:

- Program Slicing
- Observing & Watching State
- Asserting Invariants
- Detecting Anomalies
- Isolating Cause-Effect Chains
Two Views of Testing

- Testing means to execute a program with the intent to make it fail.

- Testing for validation:
  - Finding *unknown* failures (classical view)

- Testing for debugging:
  - Finding a *specific* failure (our focus)
Tests in Debugging

- Write a test to *reproduce* the problem
- Write a test to *simplify* the problem
- Run a test to *observe* the run
- Run a test to *validate a fix*
- Re-run tests to protect against *regression*
Design for Debugging

- Basic idea: decompose the system such that dependencies are minimized
- Each component depends on a minimum of other components for testing (and debugging)
Model-View-Controller

General Design Rules

**High cohesion.** Those units that operate on common data should be grouped together.

**Low coupling.** Units that do not share common data should exchange as little information as possible.
Reproduce
The First Task

- Once a problem is reported (or exposed by a test), some programmer must fix it.
- The first task is to *reproduce* the problem.
Why reproduce?

- Observing the problem. Without being able to reproduce the problem, one cannot observe it or find any new facts.
- Check for success. How do you know that the problem is actually fixed?
A Tough Problem

- Reproducing is one of the *toughest* problems in debugging.

- One must
  - recreate the *environment* in which the problem occurred
  - recreate the *problem history* – the steps that lead to the problem
Iterative Reproduction

- Start with your environment
- While the problem is not reproduced, adapt more and more circumstances from the user’s environment
- Iteration ends when problem is reproduced or when environments are “identical”
- Side effect: Learn about failure-inducing circumstances
Setting up the environment

- Millions of configurations
- Testing on dozens of different machines
- All needed to find & reproduce problems
## Virtual Machines

![](image)

### Status Monitor

**System Summary**

<table>
<thead>
<tr>
<th>Processors (2)</th>
<th>Memory (1.5 G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Machines</td>
<td>894.0 M</td>
</tr>
<tr>
<td>Other</td>
<td>642.0 M</td>
</tr>
<tr>
<td>System Total</td>
<td>1.5 G</td>
</tr>
</tbody>
</table>

### Virtual Machines (6)

<table>
<thead>
<tr>
<th>VM Name</th>
<th>Status</th>
<th>Up Time</th>
<th>% CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows XP Professional</td>
<td>Powered on</td>
<td>6 hours</td>
<td>7</td>
<td>301.0 M</td>
</tr>
<tr>
<td>Windows 2000 Cluster Node 2</td>
<td>Powered off</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 2000 Cluster Node 1</td>
<td>Suspended</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WinNT IIS Web Server</td>
<td>Suspended</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novell NetWare 6.5</td>
<td>Powered on</td>
<td>22 hours</td>
<td>3</td>
<td>157.0 M</td>
</tr>
<tr>
<td>Windows Server 2003</td>
<td>Powered on</td>
<td>35 hours</td>
<td>8</td>
<td>176.0 M</td>
</tr>
<tr>
<td>Red Hat Enterprise Linux 3</td>
<td>Powered on</td>
<td>35 hours</td>
<td>1</td>
<td>260.0 M</td>
</tr>
<tr>
<td>SuSE Linux Enterprise Server 8</td>
<td>Suspended</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Download VMware Virtual Machine Console: Windows (exe) | Linux (rpm) | Linux (tar.gz)
Reproducing Execution

- After reproducing the environment, we must reproduce the *execution*
- Basic idea: Any execution is determined by the *input* (in a general sense)
- Reproducing input → reproducing execution!
Program Inputs

Randomness  Operating System
Communication  Schedules
User Interaction  Physics
Data  Debugging Tools
Data

- Easy to transfer and replicate
- Caveat #1: *Get all the data you need*
- Caveat #2: *Get only the data you need*
- Caveat #3: Privacy issues
User Interaction

- **Record**
- **Replay**

```
Recorded Interaction
send_xevents key H @400,100
send_xevents wait 376
send_xevents key T @400,100
send_xevents wait 178
send_xevents key T @400,100
send_xevents wait 214
send_xevents key P @400,101
send_xevents wait 537
send_xevents keydn Shift_L @400,101
send_xevents wait 218
send_xevents key ";" @400,101
send_xevents wait 167
send_xevents keyup Shift_L @400,101
send_xevents wait 1556
send_xevents click 1 @428,287
send_xevents wait 3765
```
Communication

- General idea: Record and replay
  - like user interaction
- Bad impact on performance
- Alternative #1: Only record since last checkpoint
  (= reproducible state)
- Alternative #2: Only record “last” transaction
Randomness

- Program behaves different in every run
- Based on random number generator
  - Pseudo-random: save seed
  - (and make it configurable)
    - Same applies to time of day
  - True random: record + replay sequence
Operating System

- The OS handles *entire* interaction between program and environment
- Recording and replaying OS interaction thus makes entire program run reproducible
#include <string>
#include <iostream>
using namespace std;

string secret_password = "secret";

int main()
{
    string given_password;
    cout << "Please enter your password: ";
    cin >> given_password;
    if (given_password == secret_password)
        cout << "Access granted." << endl;
    else
        cout << "Access denied." << endl;
}

$ c++ -o password password.C
$ ./password
Please enter your password: secret
Access granted.
$

Traced Interaction

$ c++ -o password password.C
$ strace ./password 2> LOG
Enter your password: secret
Access granted.
$ cat LOG
...
write(1, "Please enter your password: ", 28)
read(0, "secret\n", 1024)
write(1, "Access granted.\n", 16)
exit_group(0)
Challenges

- Tracing creates *lots* of data
- Example: Web server with 10 requests/sec
  - A trace of 10 k/request means 8GB/day
- All of this must be *replayed* to reproduce the failure (alternative: *checkpoints*)
- Huge performance penalty!
Accessing Passwords

Thread A

- open(“.htpasswd”)
- read(…)
- modify(…)
- write(…)
- close(…)

Thread B

- open(“.htpasswd”)
- read(…)
- modify(…)
- write(…)
- close(…)

.htpasswd file
Lost Update

Thread A
- open(”.htpasswd”)
- read(…)
- read(…)
- modify(…)
- write(…)
- close(…)

Thread B
- modify(…)
- write(…)
- close(…)

A’s updates get lost!
Reproducing Schedules

- Thread changes are induced by a scheduler
- It suffices to record the schedule (i.e. the moments in time at which thread switches occur) and to replay it
- Requires deterministic input replay
Constructive Solutions

- Lock resource before writing
- Check resource update time before writing
- … or any other synchronization mechanism
Physical Influences

- Static electricity
- Alpha particles (*not* cosmic rays)
- Quantum effects
- Humidity
- Mechanical failures + real bugs

*Rare and hard to reproduce*
A Heisenbug

- Code fails outside debugger only

```c
int f() {
    int i;
    return i;
}
```

*In program:*  
returns random value  

*In debugger:*  
returns 0
More Bugs

- **Bohr Bug** = Repeatable under well-defined conditions
- **Heisen bug** = Changes when observed
- **Mandel bug** = Causes are complex and chaotic, appears non-deterministic, but isn’t
- **Schrödin bug** = Never should have worked, and promptly fails as soon one realizes this
Simplifying
Simplifying

- After reproducing a problem, must find out what’s relevant:
  - Does the problem really depend on 10,000 lines of input?
  - Does the failure really require this exact schedule?
  - Do we need this sequence of calls?
Simplifying

- For every circumstance of the problem, check whether it is relevant for the problem to occur.
- If it is not, remove it from the problem report or the test case in question.
Circumstances

- Any aspect that may influence a problem is a *circumstance*:
  - Aspects of the problem environment
  - Individual steps of the problem history
Experimentation

- By *experimentation*, one finds out whether a circumstance is relevant or not:
- Omit the circumstance and try to reproduce the problem.
- The circumstance is relevant if and only if the problem no longer occurs.
Ok the following operations cause mozilla to crash consistently on my machine

-> Start mozilla
-> Go to bugzilla.mozilla.org
-> Select search for bug
-> Print to file setting the bottom and right margins to .50 (I use the file /var/tmp/netscape.ps)
-> Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps)
-> This causes the browser to crash with a segfault
What’s relevant in here?
Why simplify?

- **Ease of communication.** A simplified test case is easier to communicate.
- **Easier debugging.** Smaller test cases result in smaller states and shorter executions.
- **Identify duplicates.** Simplified test cases subsume several duplicates.
Proceed by binary search. Throw away half the input and see if the output is still wrong.

If not, go back to the previous state and discard the other half of the input.
Simplified Input

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Simplified from 896 lines to one single line
- Required 12 tests only
Benefits

- **Ease of communication.** All one needs is “Printing `<SELECT>` crashes”.
- **Easier debugging.** We can directly focus on the piece of code that prints `<SELECT>`.
- **Identify duplicates.** Check other test cases whether they’re `<SELECT>`-related, too.
Why automate?

- Manual simplification is tedious.
- Manual simplification is boring.
- We have machines for tedious and boring tasks.
Deducing Errors
Obtaining a Hypothesis

- Problem Report
- Deducing from Code
- Earlier Hypotheses + Observations
- Observing a Run
- Learning from More Runs

Hypothesis
Reasoning about Runs

Experimentation
\[n \text { controlled runs}\]

Induction
\[n \text { runs}\]

Observation
\[1 \text { run}\]

Deduction
\[0 \text { runs}\]
Reasoning about Runs

Deduction
0 runs
Effects of Statements

- **Write.** A statement can change the program state (i.e. write to a variable)
- **Control.** A statement may determine which statement is executed next (other than unconditional transfer)
Affected Statements

- **Read.** A statement can read the program state (i.e. from a variable)
- **Execution.** To have any effect, a statement must be executed.
Fibonacci Numbers

\[
\text{fib}(n) = \begin{cases} 
1, & \text{for } n = 0 \lor n = 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2), & \text{otherwise}
\end{cases}
\]

1 1 2 3 5 8 13 21 34 55
## Effects in fibo.c

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reads</th>
<th>Writes</th>
<th>Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(n)</td>
<td></td>
<td>n</td>
<td>1-10</td>
</tr>
<tr>
<td>int f</td>
<td></td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>f0 = 1</td>
<td></td>
<td>f0</td>
<td></td>
</tr>
<tr>
<td>f1 = 1</td>
<td></td>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>while (n &gt; 1)</td>
<td>n</td>
<td></td>
<td>5-8</td>
</tr>
<tr>
<td>n = n - 1</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>f = f0 + f1</td>
<td>f0, f1</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>f0 = f1</td>
<td>f1</td>
<td>f0</td>
<td></td>
</tr>
<tr>
<td>f1 = f</td>
<td>f</td>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>return f</td>
<td>f</td>
<td>&lt;ret&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Control Flow

```c
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}
```
Dependences

- Following the dependences, we can answer questions like:
  - Where does this value go to?
  - Where does this value come from?
Given a statement $A$, the forward slice contains all statements whose read variables or execution could be influenced by $A$.

Formally:

$$S^f(A) = \{B \mid A \rightarrow^* B \}$$
Backward Slice

- Given a statement $B$, the backward slice contains all statements that could influence the read variables or execution of $B$
- Formally:

$$S^B(B) = \{ A | A \rightarrow^* B \}$$
Slice Operations

- **Backbones**
  - Contains only those statement that occur in both slices
  - Useful for focusing on common behavior

- **Dices**
  - Contains only the difference between two slices
  - Useful for focusing on differing behavior
Risks of Deduction

- **Code mismatch.** Is the run created from this very source code?
- **Abstracting away.** Failures may be caused by a defect in the environment.
- **Imprecision.** A slice typically encompasses 90% of the source code.
Increasing Precision

- **Verification.** If we know that certain properties hold, we can leverage them in our inference process.

- **Observation.** Facts from concrete runs can be combined with deduction.
Reasoning about Runs

Experimentation
\( n \) controlled runs

Induction
\( n \) runs

Observation
1 run

Deduction
0 runs
Reasoning about Runs

Observation
1 run

Deduction
0 runs
Observation

● Using “printf”
  • Often mocked, viewed as unscientific
  • In fact, just an easy way to apply dynamic/log analysis and perform experiments
  • If you ask the right questions, *printf* can be a great debugging tool
    • Supports scientific debugging
    • I have no lessons here, other than to print intelligently, not blindly grope around
Observation (cont)

• Using a debugger
  • Usually thought of as “more scientific”
  • It can be!
  • A debugger is good when you want to:
    • Inspect closely what happens to some values
    • Slow down and carefully watch things during one part of a run
    • Get information across a lot of state at once

• Hard to make guidelines, but in general printf is for “across time” and debuggers are for “across state”
Debugger For Exploring the Past

A typical debugging session looks like this:

1. Set a breakpoint
2. Start program, reaching breakpoint
3. Step, Step, Step, …
4. Oops! I’ve gone too far!
Omniscient Debugger

![Image of Omniscient Debugger interface]

- Stack
  - Stack trace for methods like `<main_7>`, `<Sorter_0>`, `<Sorter_1>`, `<Sorter_2>`, `<Sorter_3>`, `<Sorter_4>`, `<Sorter_5>`, `<Sorter_6>`, `<DemoRunnable_3>`, and more.

- Locals
  - Local variables `start`, `end`, and `sum` with values.

- Method Traces
  - Method trace for `String<Runnable> run()`, `void sort(int, int)`, `int average(int, int)`, and more.

- Code
  - Code snippet for `public int average(int start, int end) { ... }`.

- TTY Output
  - TTY output showing a program's log and debugging information.
How does it work?

- ODB records a *trace* of the entire execution history
- Slows down programs by a factor of 10
- Records about 100 MB/s
- Now available in commercial tools:
  - RETROVUE
  - CODEGUIDE
The Whyline

Further questions can be asked

World.move Pac

No parameters

create new parameter

create new variable

tooltips show properties’ current values

access to previous questions and answers

code related to the selection is highlighted

runtime actions

causality arrows

time cursor traverses execution history

Question: Why didn’t Pac resize 0.5?

Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.

3.821010

Big Dot.isEaten set to true

true

3.854011

Doing else

false
Question: Why didn’t Pac resize 0.5?

Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.
Questions are chosen from a hierarchical menu

Code related to the question is highlighted

Question: Why didn't Pac resize 0.5?

Answer:
One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.

Causality arrows

Time cursor traverses execution history

Runtime actions

Selection is highlighted

Does to previous questions and answers
“Why did” questions

- Take the dynamic slice of the variable
- Follow at most two dependencies
- If programmer wants, follow dependencies transitively
“Why did s = 2 in Line 15?”

```
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9   if (b > 0)
10      if (a > 1)
11         x = 2;
12         s = s + x;
13         i = i + 1;
14 }
15 write(s);
```

“Because s = 1 and i = 2”
“Why didn’t” questions

- Follow back control dependencies to closest controlling statement(s)
- Do a “why did” question on each
- Again, follow at most two dependencies
"Why didn’t x = 2 in Line 11?"

```
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10        if (a > 1)
11           x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```

“Because a = 1 and b = 1”
The WHYLINE combines
• omniscient debugging
• static slicing
• dynamic slicing
There are many other debuggers out there
Won’t spend a lot of time on GDB specifics
Major features (common to many debuggers or becoming more common)
  - Single step through a program line at a time
    - GDB: `step` and `next`
      - `next` skips over functions called in a line
  - Inspect memory locations/values
  - Set breakpoints – places to stop execution
    - Can be conditional (break at line XX if $y > z$)
  - Set watchpoints – events to watch for in execution
  - Change values in memory
  - GDB can also “run a program backwards” a little bit now!
GDB

- Single most popular GDB or other debugger feature, in C?
  • Ability to tell you where a segmentation fault took place in a program!
  • Can also do this with valgrind on Linux, rather than a debugger.
Basic Assertions

```c
if (divisor == 0) {
    printf("Division by zero! ");
    abort();
}
```
Specific Assertions

assert (divisor != 0);
void assert (int x)
{
    if (!x)
    {
        printf("Assertion failed!\n");
        abort();
    }
}
Execution

```
$ my-program
Assertion failed!
Abort (core dumped)
$
```
Better Diagnostics

$ my-program
divide.c:37:
  assertion ‘divisor != 0’ failed
Abort (core dumped)
$ _
Assertions as Macros

```c
#ifndef NDEBUG
#define assert(ex) \((ex) ? 1 : (cerr << __FILE__ << ":" << __LINE__ \<< ": assertion " #ex " failed\n", \ abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```
Invariants Ensuring Sanity (A Time Class)

class Time {
    public:
        int hour();  // 0..23
        int minutes();  // 0..59
        int seconds();  // 0..60 (incl. leap seconds)

        void set_hour(int h);
        ...
}

Any time from 00:00:00 to 23:59:60 is valid
void Time::set_hour(int h)
{
    // precondition
    assert (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);

    ...

    // postcondition
    assert (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);

}
Ensuring Sanity

```cpp
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane()); // precondition

    ... 

    assert (sane()); // postcondition
}
```
Ensuring Sanity

```cpp
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

- `sane()` is the *invariant* of a Time object:
  - holds *before* every public method
  - holds *after* every public method
Ensuring Sanity

```cpp
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane());
    ...
    assert (sane());
}
```

same for

```
set_minute(),
set_seconds(), etc.
```
Locating Infections

- Precondition failure = infection before method
- Postcondition failure = infection within method
- All assertions pass = no infection

```cpp
void Time::set_hour(int h)
{
    assert (sane()); // precondition
    ...
    assert (sane()); // postcondition
}
```
class RedBlackTree {

    
    ...  

    boolean sane() {
        assert (rootHasNoParent());
        assert (rootIsBlack());
        assert (redNodesHaveOnlyBlackChildren());
        assert (equalNumberOfBlackNodesOnSubtrees());
        assert (treesAreAcyclic());
        assert (parentsAreConsistent());

        return true;
    }

}
Aspect oriented programming

- Some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code.

- Examples:
  - Logging (tracking program behavior to a file)
  - Profiling (determining where a program spends its time)
  - Tracing (determining what methods are called when)
  - Session tracking, session expiration
  - Special security management

- The result is crosscutting code—the necessary code “cuts across” many different classes and methods.
Example

- class Fraction {
  int numerator;
  int denominator;
  ...
  public Fraction multiply(Fraction that) {
    traceEnter("multiply", new Object[]{that});
    Fraction result = new Fraction(
      this.numerator * that.numerator,
      this.denominator * that.denominator);
    result = result.reduceToLowestTerms();
    traceExit("multiply", result);
    return result;
  }
  ...
}

- Now imagine similar code in every method you might want to trace
Consequences of crosscutting code

- Redundant code
  - Same fragment of code in many places

- Difficult to reason about
  - Non-explicit structure
  - The big picture of the tangling isn’t clear

- Difficult to change
  - Have to find all the code involved...
  - ...and be sure to change it consistently
  - ...and be sure not to break it by accident

- Inefficient when crosscutting code is not needed
Aspect Terminology

- A **join point** is a well-defined point in the program flow
- A **pointcut** is a group of join points
- **Advice** is code that is executed at a pointcut
- **Introduction** modifies the members of a class and the relationships between classes
- An aspect is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable
Invariants as Aspects

```java
public aspect RedBlackTreeSanity {
    pointcut modify():
        call(void RedBlackTree.add*(..)) ||
        call(void RedBlackTree.del*(..));

    before(): modify() {
        assert (sane());
    }

    after(): modify() {
        assert (sane());
    }
}
```
Relative Debugging

Rather than checking a spec, we can also compare against a *reference run*:

- The environment has changed—e.g. ports or new interpreters
- The code has changed
- The program has been reimplemented
Relative Assertions

- We compare two program runs
- A *relative assertion* compares variable values across the two runs:

  ```
  assert \ p1::perimeter@polygon.java:65
  == \ p0::perimeter@polygon.java:65
  ```

- Specifies when and what to compare
Summary

- Assertions catch infections before they propagate too far.
- Assertions check preconditions, post conditions and invariants.
- Assertions can serve as specifications.
- A program can serve as reference to be compared against.
- Increased assertion density correlated with decrease in fault density.
System Invariants

Some properties of a program must hold over the entire run:

- must not access data of other processes
- must handle mathematical exceptions
- must not exceed its privileges

Typically checked by hardware and OS
Memory Invariants

- Even within a single process, some invariants must hold over the entire run
  - code integrity
  - data integrity
- This is a major issue in C and C++
Heap Misuse

\[ s = \text{malloc}(30) \]

free(s)

\[ t = \text{malloc}(20) \]

\[ \text{strcpy}(t, \text{"hello"}) \]

\[ s[10] = 'b' \]

free(s)
Heap Assertions

The GNU C runtime library provides a simple check against common errors:

$ MALLOC_CHECK_=2 myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
Heap Assertions

\[ s = \text{malloc}(30) \quad s \]
\[ \text{free}(s) \]
\[ \text{free}(s) \]

\text{free()} \text{ called on area that was already free'd()}
\text{Aborted (core dumped)}
Array Assertions

The *Electric Fence* library checks for array overflows:

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ ./sample-with-efence 11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```
Array Assertions

\[
s = \text{malloc}(30)\\
s[30] = \text{'x'}\]

Segmentation fault (core dumped)
Memory Assertions

- The **Valgrind** tool checks *all* memory accesses:

  ```bash
  $ valgrind sample 11 14
  Invalid read of size 4
  at 0x804851F: shell_sort (sample.c:18)
  by 0x8048646: main (sample.c:35)
  by 0x40220A50: __libc_start_main (in /lib/libc.so)
  by 0x80483D0: (within /home/zeller/sample)
  ```

- Valgrind works as an *interpreter* for x86 code
Valgrind Checks

- Use of uninitialized memory
- Accessing free’d memory
- Accessing memory beyond malloc’d block
- Accessing inappropriate stack areas
- Memory leaks: allocated area is not free’d
- Passing uninitialized memory to system calls
Shadow Memory

- V-bit set = corresponding bit is initialized
- A-bit set = corresponding byte is accessible
V-Bits

- When a bit is first written, its V-bit is set
- Simple read accesses to uninitialized memory do not result in warnings:

```c
struct S { int x; char c; 
};
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```

5 bytes initialized 8 bytes copied
(no warning)
V-Bits Warnings

- Reading uninitialized data causes a warning if
  - a value is used to generate an address
  - a control flow decision is to be made
  - a value is passed to a system call
A-Bits

- When the program starts, all global data is marked “accessible” (= A-bits are set)
- `malloc()` sets A-bits for the area returned; `free()` clears them
- Local variables are “accessible” on entry and “non-accessible” on exit
- Accessing “non-accessible” data ⇒ error
# Overhead

<table>
<thead>
<tr>
<th>Tool</th>
<th>GNU C Library</th>
<th>Electric Fence</th>
<th>Valgrind</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>2 bytes/malloc</td>
<td>1 page/malloc</td>
<td>100 %</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>negligible</td>
<td>negligible</td>
<td>2500 %</td>
</tr>
</tbody>
</table>
Preventing Misuse

• CYCLONE is a C dialect which prevents common pitfalls of C

• Most important feature: special pointers
Non-NULL Pointers

```c
int getc (FILE *fp);
```

`fp may not be NULL`

```c
extern FILE *fp;
char c = getc(fp);
```

`warning: NULL check inserted`
Fat Pointers

- A fat pointer holds address and size
- All accesses via a fat pointer are automatically bounds-checked

\[
\text{int strlen(const char* s)}
\]

<table>
<thead>
<tr>
<th>addr</th>
<th>Hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>6</td>
</tr>
</tbody>
</table>

\*s
CYCLONE Restrictions

- NULL checks are inserted
- Pointer arithmetic is restricted
- Pointers must be initialized before use
- Dangling pointers are prevented through region analysis and limitations on free()
- Only “safe” casts and unions are allowed
Production Code

- Should products ship with active assertions?
Things to Check

Critical results. If lives, health, or money depend on a result, it had better be checked.

External conditions. Any conditions which are not within our control must be checked for integrity.
Points to Consider

- The more active assertions, the greater the chance to catch infections.
- The sooner a program fails, the easier it is to track the defect.
- Defects that escape into the field are the hardest to track.
- By default, failing assertions are not user-friendly.
  - Handle assertions in a user-friendly way
- Assertions impact performance.
  - First measure; then turn off only the most time-consuming assertions
Summary

- To check memory integrity, use specialized tools to detect errors at run time
- Apply such tools before any other method
- To fully prevent memory errors, use another language (or dialect, e.g. Cyclone)
- Turning assertions off seldom justifies the risk of erroneous computation
Techniques

Infections
e.g. a failed assertion

Code smells
e.g. uninitialized variable

Dependences
e.g. a[2] comes from a[0]

Anomalies
e.g. f() executed only in failing run

Causes
e.g. a[2] = 0 causes the failure

How do we integrate these techniques?
All Techniques
Dependencies
Observation
Assertion
Assertion
Anomaly
Anomaly
Cause Transition
The Defect
Ordering

1. Infections
   e.g. a failed assertion

2. Causes
   e.g. $a[2] = 0$
   causes the failure

3. Anomalies
   e.g. $f()$ executed only in failing run

4. Code smells
   e.g. uninitialized variable

5. Dependences
   e.g. $a[2]$ comes from $a[0]$
Remember The Traffic Principle

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect
Validating the Defect

● Any element of the infection chain must be
  • infected – i.e., have an incorrect value
  • a failure cause – i.e., changing it causes the failure to no longer occur

● Demonstrate by experiments and observation
Validating Causality

- In principle, we must show causality for each element of the infection chain

- However, a successful correction retrospectively validates causality:
  - Since the failure has gone, we have proven that the defect caused the failure

- Yet, we must not fall into ignorant surgery
Think before you code

• Before applying a fix, you must understand
  • how your code change will *break* the infection chain, and
  • how this will make the failure (as well as other failures) no longer occur

• In fact, you have a theory about the defect
Never In Debugging

- Find the defect by guessing:
  - Scatter debugging statements everywhere
  - Try changing code until something works
  - Don’t back up old versions of the code
  - Don’t bother understanding what the program should do.

- Don’t waste time understanding the problem.
  - Most problems are trivial, anyway.

- Use the most obvious fix.
  - Just fix what you see:
    - \( x = \text{compute}(y) \)
    - \!/ \text{compute(17) is wrong – fix it}
    - if \((y == 17)\)
      - \( x = 25.15 \)
    - Why bother going into compute()?
Learning from Mistakes
Fixing the Process

- Any defect escaping into the wild should have been caught by local *quality assurance*
- Besides fixing the defect, we also must fix quality assurance!
Things to do

- Improve your test suite
- Set up assertions
- Improve training
- Improve the software process
- Improve the analysis tools
Things to Measure

- How much damage did the defect do?
- How much effort did it take to fix it?
- What is the risk we are taking in letting such defects go unnoticed?
Some Facts

- In Eclipse and Mozilla, 30–40% of all changes are fixes (Sliverski et al., 2005)

- Fixes are 2–3 times smaller than other changes (Mockus + Votta, 2000)

- 4% of all one-line changes introduce new errors (Purushothaman + Perry, 2004)

- A module that is one year older has 30% less errors (Graves et al., 2000)

- New code is 2.5 times as defect-prone as old code (Ostrand + Weyuker, 2002)
The Risk of Change

Some locations in a program are risky: many changes result in a fix.
Fixes and Changes

- How do we know a change is a fix?

The problem database relates fixes to problems
Hints for relating problems and fixes include:

- Problem ID in the log message of the fix: *Fixed bug 53784: .class file missing*

- Changes before closing a problem: *Before closing #53784, changed This.java*

- For about 50% of all closed problems, we can identify the related fix
Fix-Inducing Changes

- Can we predict the *risk of change*?
- Which are the *risky locations*?
- Do they have common *features*?
What makes changes risky?

- To determine whether changes induce risk, a number of *metrics* have been proposed:

<table>
<thead>
<tr>
<th>size of file being changed</th>
<th>size of the change</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of changes so far</td>
<td>number of fixes so far</td>
</tr>
</tbody>
</table>
What makes changes risky?

● Past risk at the change location is predictor for future risk

| # of past fix-inducing changes at the change location | # of past fix-inducing fixes at the change location |
Risk along the Week

![Bar chart showing probability distribution for Mozilla and Eclipse over a week.](image-url)
What makes changes risky?

- Past risk at the location
- The day of the week
- Properties of the code?
Risk implies Complexity

- A location is *complex* if it is risky to change
- *Factual complexity measure* – in contrast to *metrics* like McCabe and related
- Risk of change allows for *evaluation* and *mining* of metrics
### Which features correlate with risk

<table>
<thead>
<tr>
<th>do...while</th>
<th>multiple inheritance</th>
<th>DirectX API</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterators</td>
<td>no iterators</td>
<td>method size</td>
</tr>
<tr>
<td>developer</td>
<td>use of XP</td>
<td>and more…</td>
</tr>
</tbody>
</table>

Correlation can be specific to project – or universal
Requirements

- Well-kept version and bug databases
- Link between changes and problems
- Willingness to change
- Policy on how to handle sensitive data
Problem Tracking

• When was the error discovered? How? Who? What flight?
• How was the error introduced? Why wasn’t it caught?
• How was the error corrected? Are there similar errors?
• What can we learn from previous errors?
The Process

- Software error means there is *an error in the process*
- Planning the software carefully in advance
- Reducing risk at all stages
- Keeping record of all activities
- Just standard practice in engineering
Summary

- To learn from mistakes, mine existing archives
- Risk of change serves as complexity measure
  - Predicts future risk better than other metrics
  - Identifies risky features of code and process
- Open issue: Which features universally correlate with mistakes?