# Reasoning about Programs (and bugs)

A brief interlude on
specifications, assertions, and debugging

Largely based on material from University of Washington CSE 331

## Good programs, broken programs?

Goal: program works (does not fail)

Need: definition of *works/correct*: a **specification**

But programs fail all the time. Why?

1. Misuse of your code: caller did not meet assumptions
2. Errors in your code: mistake causes wrong computation
3. Unpredictable external problems:
   - Out of memory, missing file, network down, …
   - Plan for these problems, fail gracefully.
4. Wrong or ambiguous specification, implemented correctly

## A Bug's Life, *ca.* 1947

-- Grace Hopper

## A Bug's Life

**Defect**: a mistake in the code

Think 10 per 1000 lines of industry code. We're human.

**Error**: incorrect computation

Because of defect, but not guaranteed to be visible

**Failure**: observable error -- program violates its specification

Crash, wrong output, unresponsive, corrupt data, etc.

Time / code distance between stages varies:

- tiny (<second to minutes / one line of code)
- or enormous (years to decades to never / millons of lines of code)

## "How to build correct code"

1. **Design and Verify**
   Make correctness more likely or provable from the start.
2. **Program Defensively**
   Plan for defects and errors.
   - make testing more likely to reveal errors as failures
   - make debugging failures easier
3. **Test and Validate**
   Try to cause failures.
   - provide evidence of defects/errors
   - or increase confidence of their absence
4. **Debug**
   Determine the cause of a failure.
   ***(Hard! Slow! Avoid!)*** *Solve inverse problem.*

## Testing

- **Can** show that a program has an error.
- **Can** show **a** point where an error causes a failure.
- **Cannot** show the error that caused the failure.
- Cannot show the defect that caused the error.

- **Can** improve confidence that the sorts of errors/failures targeted by the tests are less likely in programs similar to the tests.
- **Cannot** show absence of defects/errors/failures.
  - Unless you can test all possible behaviors exhaustively. Usually intractable for interesting programs.

## Why reason about programs statically?

(without running them)

"Today a usual technique is to make a program and then to test it. While program testing can be a very effective way to show the presence of bugs, it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. "

*-- Edsger Dijkstra*

## Reasoning about programs

- Reason about a single program execution.
  - **Concrete, dynamic**: be the machine, run the program.
  - Test or debug: important, but "too late."
- Reason about all possible executions of a program.
  - **Abstract, static**: consider all possible paths at once.
  - Usually to prevent broken programs.
  - Hard for whole programs, easier if program uses clean, modular abstractions.
- Many compromises in between.

## Forward Reasoning

Suppose we initially know (or assume) **w > 0**

```
// w > 0
x = 17;
// w > 0,  x == 17
y = 42;
// w > 0,  x == 17,  y == 42
z = w + x + y;
// w > 0,  x == 17,  y == 42,  z > 59
…
```

Then we know various things after, e.g., `z > 59`

## Forward: careful with assignment

```
// we know: nothing
w = x+y;
// we know: w == x + y
x = 4;
// we know: w == old x + y, x == 4
// must update other facts too...
y = 3;
// we know: w == old x + old y,
//          x == 4, y == 3
// we do NOT know: w == x + y == 7
```

## Backward Reasoning

If we want `z < 0` at the end

```
// w + 17 + 42 < 0
x = 17;
// w + x + 42 < 0
y = 42;
// w + x + y < 0
z = w + x + y;
// z < 0
```

Then we need to start with `w < -59`

## Reasoning Forward and Backward

Forward:

- Determine what assumptions imply.
- Ensure an invariant is maintained.
  - Invariant = property that is always true

Backward:

- Determine sufficient conditions.
  - For a desired result:
    What assumptions are needed for correctness?
  - For an undesired result:
    What assumptions will trigger an error/bug?

# Reasoning Forward and Backward

Forward:

- Simulate code on many inputs at once.
- Learn many facts about code's behavior,
  - some of which may be irrelevant.

Backward:

- Show how each part of code affects the end result.
- More useful in many contexts (research, practice)
- Closely linked with debugging

# Precondition and Postcondition

Precondition: “assumption” before some code

```
// pre:  w < -59
x = 17;
// post: w + x < -42
```

Postcondition: “what holds” after some code

If you satisfy the precondition, then you are guaranteed the postcondition.

# Conditionals, forward.

```
// pre: initial assumptions
if(...) {
        // pre:  && condition true
    ... // post: X
} else {
        // pre:  && condition false
    ... // post: Y
}
// either branch could have executed
// post: X || Y
```

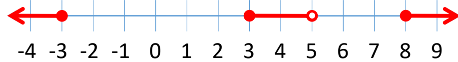
# Conditionals, backward.

```
// pre: (C, X) or (!C, Y)
if(C) {
        // pre:  X: weakest such that
    ... // post: Z
} else {
        // pre:  Y: weakest such that
    ... // post: Z
}
// either branch could have executed
// post: need Z
```

**Weakest precondition:** the minimal assumption under which the postcondition is guaranteed to be true.

# Conditional, backward

```
// 9. pre: x <= -3 or (3 <= x, x < 5) or 8 <= x
// 8. pre: (x <= -3, x < 5) or (3 <= x, x < 5)
//         or 8 <= x
// 7. pre: (x < 5, (x <= -3 or 3 <= x))
//         or 8 <= x
// 6. pre: (x < 5, 9 <= x*x) or 8 <= x
// 5. pre: (x < 5, 9 <= x*x) or (5 <= x, 8 <= x)
if (x < 5) {
  // 4. pre: 9 <= x*x
  x = x*x;
  // 2. post: 9 <= x
} else {
  // 3. pre: 8 <= x
  x = x+1;
  // 2. post: 9 <= x
}
// 1. post: 9 <= x
```




# Is static reasoning enough?

- Can learn things about the program we have.
- Basis for human proofs, limited automated reasoning.
  - Compilers check types, do correct optimizations.
  - Many static program analysis techniques
- Proving entire program correct is HARD!

- Should also write down things we expect to be true

# "How to build correct code"

1. **Design and Verify**
   Make correctness more likely or provable from the start.
2. **Program Defensively**
   Plan for defects and errors.
   - make testing more likely to reveal errors as failures
   - make debugging failures easier
3. **Test and Validate**
   Try to cause failures.
   - provide evidence of defects/errors
   - or increase confidence of their absence
4. **Debug**
   Determine the cause of a failure.
   ***(Hard! Slow! Avoid!)*** *Solve inverse problem.*

# What to do when things go wrong

## Early, informative failures

**Goal 1: *Give information about the problem***

- To the programmer – descriptive error message
- To the client code: exception, return value, etc.

**Goal 2: *Prevent harm***

Whatever you do, do it early: before small error causes big problems
Abort: alert human, cleanup, log the error, etc.
Re-try if safe: problem might be transient
Skip a subcomputation if safe: just keep going
Fix the problem? *Usually* infeasible to repair automatically

# Defend your code

1. Make errors ***impossible*** with type safety, memory safety (not C!).
2. Do not introduce defects, make reasoning easy with simple code.
   - KISS = Keep It Simple, Stupid
3. Make errors ***immediately visible*** with assertions.
   - Reduce distance from error to failure
4. Debug (last resort!): find defect starting from failure
   - Easiest in modular programs with good specs, test suites, assertions
   - Use scientific method to gain information.

- Analogy to health/medicine:
  wellness/prevention vs. diagnosis/treatment

There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies,
and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult.

*-- Sir Anthony Hoare, Turing Award winner*

Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

*-- Brian Kernighan, author of The C Programming Language book, much more*

# Defensive programming, testing

Check:

- Precondition and Postcondition
- Representation invariant
- Other properties that should be true

Check *statically* via reasoning and tools

Check *dynamically* via assertions

```
assert(index >= 0);
assert(array != null);
assert(size % 2 == 0);
```

Write assertions as you write code

Write many tests and run them often

# Square root with assertion

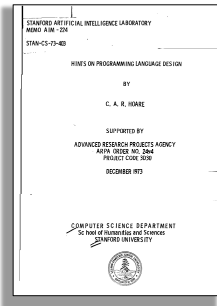
```
// requires: x >= 0
// returns: approximation to square root of x
double sqrt(double x) {
  assert(x >= 0.0);
  double result;
  ... compute square root ...
  assert(absValue(result*result – x) < 0.0001);
  return result;
}
```

# Don’t go to sea without your lifejacket!

Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea?

*Hints on Programming Language Design*

*-- C.A.R. Hoare*



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-224

STAN-CS-73-403

HINTS ON PROGRAMMING LANGUAGE DESIGN

BY

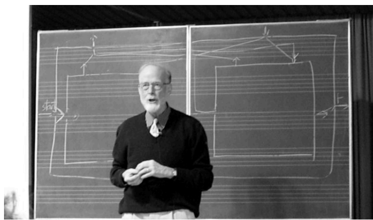
C. A. R. HOARE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494
PROJECT CODE 3D30

DECEMBER 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



# When *not* to use assertions

Don't check for user input errors with assertions. User errors are *expected* situations that programs must handle.

```
// assert(!isEmpty(zipCode));   // XX NO XX
if (isEmpty(zipCode)) {
  handleUserError(...);
}
```

Don’t clutter code with useless, distracting repetition

```
x = y + 1;
// assert(x == y + 1); // XX NO XX
```

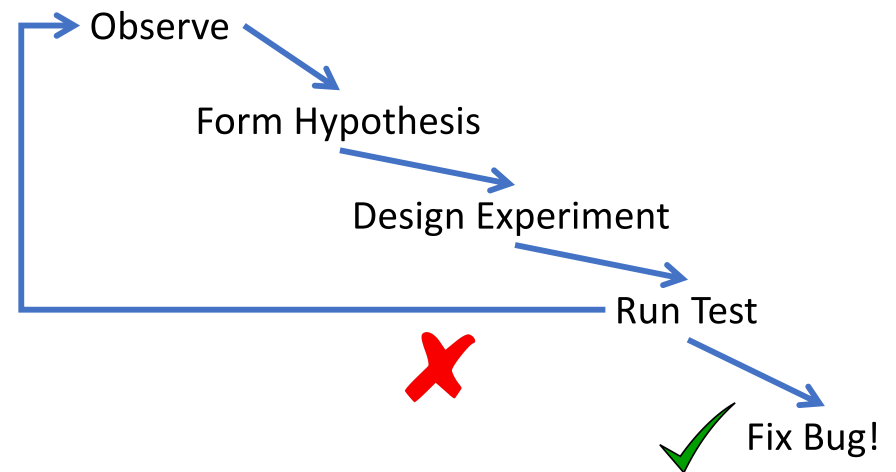
Don’t perform side effects, won’t happen if assertions disabled.

```
// assert(array[i]++ != 42); // XX NO XX
array[i]++;  // part of the program logic
assert(array[i] != 42);
printf(array[i]);
```

# Last Resort: Principled Debugging

1. Find small, repeatable test case that produces the failure
2. Narrow down location and proximate cause
   - *Scientific Method: observe, hypothesize, experiment, analyze*
   - *Keep a record*
3. Fix the defect (and test the fix!)
   - Is it a simple typo, or a design flaw?
   - *Does it occur elsewhere?*
4. Add #1 as a (regression) test for the future.

# Principled Debugging



# Example in practice.c

```
// returns 1 iff needle is a substring of haystack,
// otherwise returns 0
int contains_string(char* haystack, char* needle);
```

***Failure**:* can't find `"very happy"` within:

```
"Fáilte, you are very welcome! Hi Seán! I am
very very happy to see you all."
```

**Ugly:** Accents?! Panic about Unicode!!! Google wildly, copy random code you don't understand from StackOverflow, install new string library, …

**Bad:** Start tracing the execution of this example

**Good:** simplify/clarify the symptom…

Disclaimer: borrowing this reference, have not had time to learn what it is.

# Minimize the failing *input*,

and distance to non-failing input.

*Can not* find `"very happy"` within

```
"Fáilte, you are very welcome! Hi Seán! I am
very very happy to see you all."
```

*Can* find `"very happy"` within

```
"Fáilte, you are very welcome! Hi Seán!"
```

*Can not* find `"very happy"` within

```
"I am very very happy to see you all."
"very very happy"
```

*Can* find `"very happy"` within

```
"very happy"
```

*Can not* find `"ab"` within `"aab"`

*Can* find `"ab"` within `"ab"`, `"abb"`, `"bab"`

# Minimize the failing *code* (localize)

Exploit modularity

- Start with everything, take away pieces until failure goes away
- Start with nothing, add pieces back in until failure appears

Exploit modular reasoning

- Trace through program, viewing intermediate results

*Binary search* speeds up the process

- Error happens somewhere between first and last statement
- Do binary search on that ordered set of statements

# Debugging at scale…

Real Systems

- Large and complex
- Collection of modules, written by multiple people
- Complex input
- Many external interactions
- Non-deterministic

Replication can be an issue

- Infrequent failure
- Instrumentation eliminates the failure

Defects cross abstraction barriers

Large time lag from corruption (defect) to detection (failure)