# CSE 331
# Software Design & Implementation

Kevin Zatloukal

Spring 2022

Lecture 2 – Reasoning About Straight-Line Code

# Motivation for Reasoning

- Want a way to determine correctness without running the code
- Most important part of the correctness techniques
  – tools, **inspection**, testing
- You need a way to do this in interviews
  – key reason why coding interviews are done without computers
- This is not easy (see HW0)

# Our Approach

- We will learn a set of **formal tools** for proving correctness
  - (later, this will also allow us to generate the code)

- Most professionals can do reasoning like this in their head
  - most do an *informal* version of what we will see
  - eventually, it will be the same for you

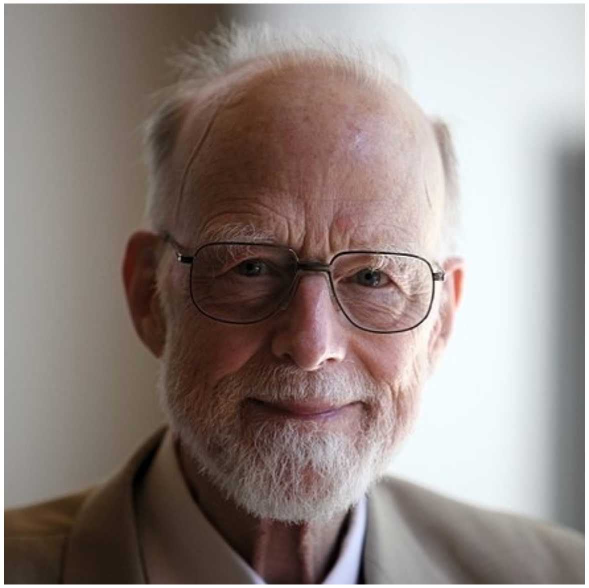
- Formal version has key advantages
  - teachable
  - mechanical (no intuition or creativity required)
  - necessary for hard problems
    - we turn to formal tools when problems get too hard

# Formal Reasoning

- Invented by Robert Floyd and Sir Anthony Hoare
  - Floyd won the Turing award in 1978
  - Hoare won the Turing award in 1980

Robert Floyd

picture from Wikipedia



Tony Hoare

# Terminology of Floyd Logic

- The *program state* is the values of all the (relevant) variables
- An *assertion* is a true / false claim (proposition) about the state at a given point during execution (e.g., on line 39)
- An assertion *holds* for a program state if the claim is true when the variables have those values
- An assertion before the code is a *precondition*
  - these represent assumptions about when that code is used
- An assertion after the code is a *postcondition*
  - these represent what we want the code to accomplish

# Hoare Triples

- A Hoare triple is two assertions and one piece of code:

$$\{P\}\ S\ \{Q\}$$

  - *P* the precondition
  - *S* the code
  - *Q* the postcondition

specification

method body

code is correct iff triple is valid

- A Hoare triple $\{P\}\ S\ \{Q\}$ is called valid if:
  - in any state where P holds,
    executing S produces a state where Q holds
  - i.e., if *P* is true before *S*, then *Q* must be true after it
  - otherwise, the triple is called invalid

# Notation

- Floyd logic writes assertions in {..}
  - since Java code also has {..}, I will use {{…}}
  - e.g., {{ w >= 1 }} **`x = 2 * w;`** {{ x >= 2 }}
- Assertions are math / logic not Java
  - you can use the usual math notation
    - (e.g., = instead of == for equals)
  - purpose is communication with other humans (not computers)
  - we will need **`and`**, **`or`**, **`not`** as well
    - can also write use ∧ (and) ∨ (or) etc.
- The Java language also has assertions (**`assert`** statements)
  - throws an exception if the condition does not evaluate true
  - we will discuss these more later in the course

# Example 1

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

```
{{ x != 0 }} y = x*x; {{ y > 0 }}
```

# Example 1

Is the following Hoare triple valid or invalid?

– assume all variables are integers and there is no overflow

```
{{ x != 0 }} y = x*x; {{ y > 0 }}
```

Valid

- **`y`** could only be zero if **`x`** were zero (which it isn't)

# Example 2

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

```
{{ z != 1 }} y = z*z; {{ y != z }}
```

# Example 2

Is the following Hoare triple valid or invalid?

– assume all variables are integers and there is no overflow

```
{{ z != 1 }} y = z*z; {{ y != z }}
```

Invalid

- counterexample: `z = 0`

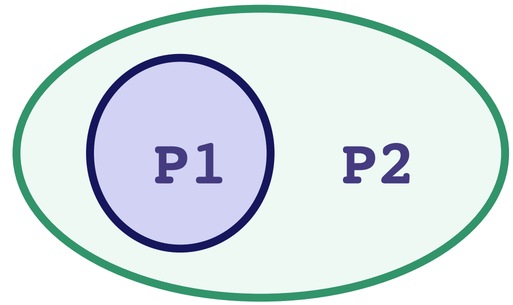
# Checking Validity

- So far: decided if a Hoare triple is valid by ... **hard** thinking
- Soon: mechanical process for reasoning about
  - assignment statements
  - conditionals
  - [next lecture] loops
  - (all code can be understood in terms of those 3 elements)
- Can use those to check correctness in a “turn the crank” manner
- Next: a way to compare different assertions
  - useful, e.g., to compare possible preconditions

# Weaker vs. Stronger Assertions

If P1 implies P2 (written P1 $\Rightarrow$ P2), then:

- P1 is stronger than P2
- P2 is weaker than P1




Whenever P1 holds, P2 also holds

- So it is more (or at least as) "difficult" to satisfy P1
  - the program states where P1 holds are a subset of the program states where P2 holds
- So P1 puts more constraints on program states
- So it is a stronger set of requirements on the program state
  - P1 gives you more information about the state than P2

# Examples

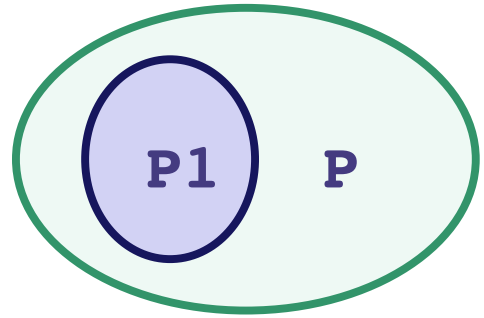
- **`x = 17`** is stronger than **`x > 0`**
- **`x is prime`** is neither stronger nor weaker than **`x is odd`**
- **`x is prime and x > 2`** is stronger than **`x is odd`**

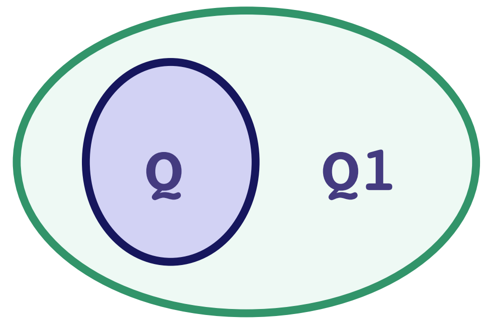
# Floyd Logic Facts

- Suppose **`{P} S {Q}`** is valid.

- If **`P1`** is stronger than **`P`**,
  then **`{P1} S {Q}`** is valid.

- If **`Q1`** is weaker than **`Q`**,
  then **`{P} S {Q1}`** is valid.



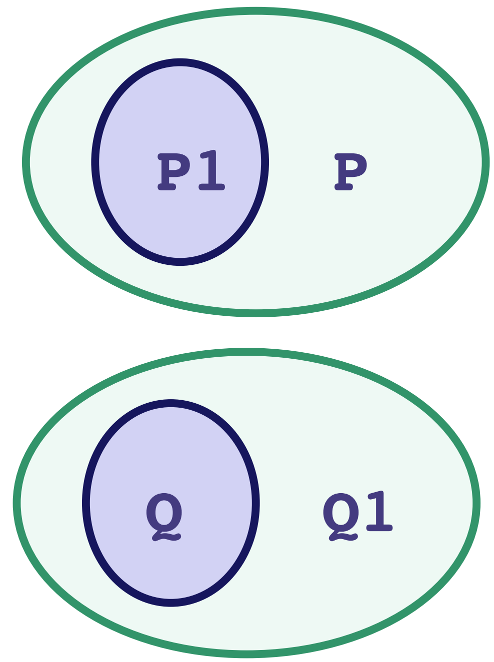

- Example:
  - Suppose **`P`** is x >= 0 and **`P1`** is x > 0
  - Suppose **`Q`** is y > 0 and **`Q1`** is y >= 0
  - Since {{ x >= 0 }} **`y = x+1`** {{ y > 0 }} is valid,
    {{ x > 0 }} **`y = x+1`** {{ y >= 0 }} is also valid

# Floyd Logic Facts

- Suppose **`{P} S {Q}`** is valid.

- If **`P1`** is stronger than **`P`**,
  then **`{P1} S {Q}`** is valid.

- If **`Q1`** is weaker than **`Q`**,
  then **`{P} S {Q1}`** is valid.

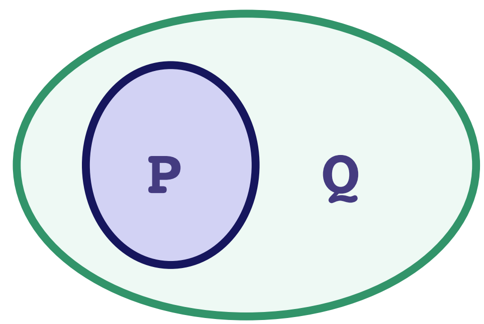



- **Key points**:
  - always okay to **strengthen** a **precondition**
  - always okay to **weaken** a **postcondition**

# Floyd Logic Facts

- When is **`{P} ; {Q}`** is valid?
  - with no code in between

- Valid if any state satisfying P also satisfies Q
- I.e., if P is **stronger** than Q



# Forward & Backward Reasoning

# Example of Forward Reasoning

Work forward from the precondition

{{ w > 0 }}

```
x = 17;
```

{{ _______________________________ }}

```
y = 42;
```

{{ _______________________________ }}

```
z = w + x + y;
```

{{ _______________________________ }}

# Example of Forward Reasoning

Work forward from the precondition

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x = 17 }}

```
y = 42;
```

{{ ______________________________ }}

```
z = w + x + y;
```

{{ ______________________________ }}

# Example of Forward Reasoning

Work forward from the precondition

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x = 17 }}

```
y = 42;
```

{{ w > 0 and x = 17 and y = 42 }}

```
z = w + x + y;
```

{{ ________________________________________ }}

# Example of Forward Reasoning

Work forward from the precondition

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x = 17 }}

```
y = 42;
```

{{ w > 0 and x = 17 and y = 42 }}

```
z = w + x + y;
```

{{ w > 0 and x = 17 and y = 42 and z = w + x + y }}

# Example of Forward Reasoning

Work forward from the precondition

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x = 17 }}

```
y = 42;
```

{{ w > 0 and x = 17 and y = 42 }}

```
z = w + x + y;
```

{{ w > 0 and x = 17 and y = 42 and z = w + 59 }}

# Forward Reasoning

- Start with the **given** precondition
- Fill in the **strongest** postcondition

- For an assignment, **`x = y`**...
  - add the fact “x = y” to what is known
  - important <u>subtleties</u> here... (more on those later)

- Later: if statements and loops...

# Example of Backward Reasoning

Work backward from the desired postcondition

{{ ______________________________ }}

```
x = 17;
```

{{ ______________________________ }}

```
y = 42;
```

{{ ______________________________ }}

```
z = w + x + y;
```

{{ z < 0 }}

# Example of Backward Reasoning

Work backward from the desired postcondition

{{ ______________________________ }}

```
x = 17;
```

{{ ______________________________ }}

```
y = 42;
```

{{ w + x + y < 0 }}

```
z = w + x + y;
```

{{ z < 0 }}

# Example of Backward Reasoning

Work backward from the desired postcondition

{{ ______________________________ }}

```
x = 17;
```

{{ $w + x + 42 < 0$ }}

```
y = 42;
```

{{ $w + x + y < 0$ }}

```
z = w + x + y;
```

{{ $z < 0$ }}

# Example of Backward Reasoning

Work backward from the desired postcondition

{{ w + 17 + 42 < 0 }}

**`x = 17;`**

{{ w + x + 42 < 0 }}

**`y = 42;`**

{{ w + x + y < 0 }}

**`z = w + x + y;`**

{{ z < 0 }}

# Backward Reasoning

- Start with the **required** postcondition
- Fill in the **weakest** precondition

- For an assignment, **`x = y`**:
  - just replace "x" with "y" in the postcondition
  - if the condition using "y" holds beforehand, then the condition with "x" will afterward since x = y then

- Later: if statements and loops...

# Correctness by Forward Reasoning

Use forward reasoning to determine if this code is correct:

{{ w > 0 }}

```
x = 17;
y = 42;
z = w + x + y;
```

{{ z > 50 }}

# Example of Forward Reasoning

{{ w > 0 }}

```
x = 17;
```

{{ w > 0 and x=17 }}

```
y = 42;
```

{{ w > 0 and x=17 and y=42 }}

```
z = w + x + y;
```

{{ w > 0 and x=17 and y=42 and z = w + 59 }}

{{ z > 50 }}

Do the facts that are always true imply the facts we need?

I.e., is the bottom statement **weaker** than the top one?

(Recall that weakening the postcondition is always okay.)

# Correctness by Backward Reasoning

Use backward reasoning to determine if this code is correct:

{{ w < -60 }}

```
x = 17;
y = 42;
z = w + x + y;
```

{{ z < 0 }}

# Correctness by Backward Reasoning

Use backward reasoning to determine if this code is correct:

$\{\{\ w < -60\ \}\}$

Do the facts that are always true imply the facts we need?

I.e., is the top statement **stronger** than the bottom one?

$\{\{\ w + 17 + 42 < 0\ \}\} \iff \{\{\ w < -59\ \}\}$

```
x = 17;
```

$\{\{\ w + x + 42 < 0\ \}\}$

(Recall that strengthening the precondition is always okay.)

```
y = 42;
```

$\{\{\ w + x + y < 0\ \}\}$

```
z = w + x + y;
```

$\{\{\ z < 0\ \}\}$

# Combining Forward & Backward

It is okay to use both types of reasoning

- Reason forward from precondition
- Reason backward from postcondition

Will meet in the middle:

{{ P }}
```
S1
S2
```
{{ Q }}

# Combining Forward & Backward

It is okay to use both types of reasoning

- Reason forward from precondition
- Reason backward from postcondition

Will meet in the middle:

{{ P }}
**`S1`**
{{ P1 }}
{{ Q1 }}
**`S2`**
{{ Q }}

Valid provided P1 implies Q1

# Combining Forward & Backward

Reasoning in either direction gives valid assertions

Just need to check adjacent assertions:

- top assertion must imply bottom one

{{ P }}
**S1**
**S2**
{{ P1 }}
{{ Q }}

{{ P }}
{{ Q1 }}
**S1**
**S2**
{{ Q }}

{{ P }}
**S1**
{{ P1 }}
{{ Q1 }}
**S2**
{{ Q }}

# Subtleties in Forward Reasoning...

- Forward reasoning can **fail** if applied blindly...

{{ }}

```
w = x + y;
```

{{ w = x + y }}

```
x = 4;
```

{{ w = x + y and x = 4 }}

```
y = 3;
```

{{ w = x + y and x = 4 and y = 3 }}

This implies that w = 7, but that is not true!

– w equals whatever x + y was **before** they were changed

# Fix 1

- Use **subscripts** to refer to old values of the variables
- Un-subscripted variables should always mean **current** value

{{ }}

```
 w = x + y;
```

{{ $w = x + y$ }}

```
 x = 4;
```

{{ $w = x_1 + y$ and $x = 4$ }}

```
 y = 3;
```

{{ $w = x_1 + y_1$ and $x = 4$ and $y = 3$ }}

# Fix 2 (better)

- Express prior values in terms of the current value

{{ }}
```
 w = x + y;
```
{{ w = x + y }}
```
 x = x + 4;
```
{{ $w = x_1 + y$ and $x = x_1 + 4$ }}  Now, $x_1 = x - 4$

$\Rightarrow$ {{ $w = x - 4 + y$ }}  So $w = x_1 + y \Leftrightarrow w = x - 4 + y$

Note for updating variables, e.g., **`x = x + 4`**:

- Backward reasoning just substitutes new value (no change)
- Forward reasoning requires you to invert the “+” operation

# Forward vs. Backward

- Forward reasoning:
  - Find strongest postcondition
  - Intuitive: “simulate” the code in your head
    - BUT you need to change facts to refer to *prior values*
  - Inefficient: Introduces many irrelevant facts
    - usually need to weaken as you go to keep things sane

- Backward reasoning
  - Find weakest precondition
  - Formally simpler
  - Efficient
  - (Initially) unintuitive

# If Statements

# If Statements

Forward reasoning

{{ P }}
```
if (cond)
  S1
else
  S2
```
{{ ? }}

# If Statements

Forward reasoning

```
{{ P }}
if (cond)
   {{ P and cond }}
   S1
else
   {{ P and not cond }}
   S2
{{ ? }}
```

# If Statements

Forward reasoning

{{ P }}
`if (cond)`
  {{ P and `cond` }}
  `S1`
  {{ P1 }}
`else`
  {{ P and not `cond` }}
  `S2`
  {{ P2 }}
{{ ? }}

# If Statements

Forward reasoning

{{ P }}
`if (cond)`
    {{ P and `cond` }}
    `S1`
    {{ P1 }}
`else`
    {{ P and not `cond` }}
    `S2`
    {{ P2 }}
{{ P1 or P2 }}

# If Statements

Backward reasoning

{{ ? }}

```
if (cond)
  S1
else
  S2
```

{{ Q }}

# If Statements

Backward reasoning

{{ ? }}
```
if (cond)
  S1
```
{{ Q }}
```
else
  S2
```
{{ Q }}

{{ Q }}

# If Statements

Backward reasoning

```
{{ ? }}
if (cond)
    {{ Q1 }}
    S1
    {{ Q }}
else
    {{ Q2 }}
    S2
    {{ Q }}
{{ Q }}
```

# If Statements

Backward reasoning

```
{{ cond and Q1 or
   not cond and Q2 }}
if (cond)
   {{ Q1 }}
   S1
   {{ Q }}
else
   {{ Q2 }}
   S2
   {{ Q }}
{{ Q }}
```

# If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
  y = x;
else
  y = -x;
{{ ? }}
```

# If-Statement Example

Forward reasoning

{{ }}

```
if (x >= 0)
```

{{ x >= 0 }}

```
  y = x;
else
```

{{ x < 0 }}

```
  y = -x;
```

{{ ? }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ ? }}

# If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
   {{ x >= 0 }}
   y = x;
   {{ x >= 0 and y = x }}
else
   {{ x < 0 }}
   y = -x;
   {{ x < 0 and y = -x }}
{{ (x >= 0 and y = x) or
   (x < 0 and y = -x) }}
```

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
  {{ x >= 0 }}
```
  y = x;
```
  {{ x >= 0 and y = x }}
```
else
```
  {{ x < 0 }}
```
  y = -x;
```
  {{ x < 0 and y = -x }}

{{ y = |x| }}

**Warning**: many write {{ y >= 0 }} here

That is true but it is *strictly* weaker.
(It includes cases where y != x)

# If-Statement Example

Forward reasoning

{{ }}

```
if (x >= 0)
```

  {{ x >= 0 }}

```
  y = x;
```

  {{ x >= 0 and y = x }}

```
else
```

  {{ x < 0 }}

```
  y = -x;
```

  {{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ ? }}

```
if (x >= 0)
  y = x;
else
  y = -x;
```

{{ y = |x| }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
  y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
  y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ ? }}
```
if (x >= 0)
  y = x;
```
{{ y = |x| }}
```
else
  y = -x;
```
{{ y = |x| }}

{{ y = |x| }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ ? }}
```
if (x >= 0)
```
{{ x = |x| }}
```
y = x;
```
{{ y = |x| }}
```
else
```
{{ -x = |x| }}
```
y = -x;
```
{{ y = |x| }}

{{ y = |x| }}

# If-Statement Example

Forward reasoning

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ x >= 0 and y = x }}
else
    {{ x < 0 }}
    y = -x;
    {{ x < 0 and y = -x }}
{{ y = |x| }}
```

Backward reasoning

```
{{ ? }}
if (x >= 0)
    {{ x >= 0 }}
    y = x;
    {{ y = |x| }}
else
    {{ x <= 0 }}
    y = -x;
    {{ y = |x| }}
{{ y = |x| }}
```

# If-Statement Example

Forward reasoning

{{ }}

```
if (x >= 0)
```

{{ x >= 0 }}

```
y = x;
```

{{ x >= 0 and y = x }}

```
else
```

{{ x < 0 }}

```
y = -x;
```

{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ (x >= 0 and x >= 0) or

(x < 0 and x <= 0) }}

```
if (x >= 0)
```

{{ x >= 0 }}

```
y = x;
```

{{ y = |x| }}

```
else
```

{{ x <= 0 }}

```
y = -x;
```

{{ y = |x| }}

{{ y = |x| }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ x >= 0 or x < 0 }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ y = |x| }}
```
else
```
{{ x <= 0 }}
```
y = -x;
```
{{ y = |x| }}

{{ y = |x| }}

# If-Statement Example

Forward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ x >= 0 and y = x }}
```
else
```
{{ x < 0 }}
```
y = -x;
```
{{ x < 0 and y = -x }}

{{ y = |x| }}

Backward reasoning

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x;
```
{{ y = |x| }}
```
else
```
{{ x <= 0 }}
```
y = -x;
```
{{ y = |x| }}

{{ y = |x| }}

Next time: Loops...