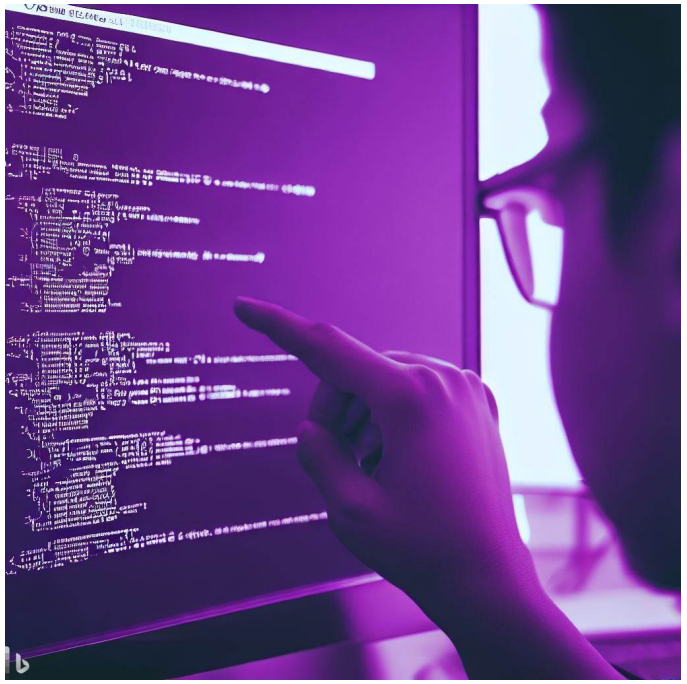


# CSE 331

## Floyd Logic

James Wilcox and Kevin Zatloukal

# Reasoning So Far

- Code so far made up of three elements
  - straight-line code (just variable declarations and returns)
  - conditionals
  - recursion

- All code without mutation looks like this

- Proving correctness is proving implications
  - check that known facts imply the required facts

# Recall: Finding Facts at a Return Statement

- **Consider this code**

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  if (a >= 0 && b >= 0) {
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

find facts by reading along path from top to return statement

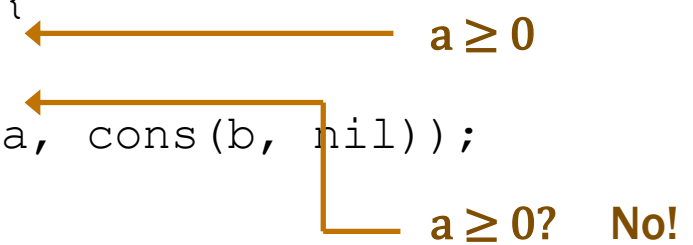
- **Known facts include** "$a \geq 0$", "$b \geq 0$", **and** "$L = \text{cons}(...)$"

- **Prove that postcondition holds:** "$\text{sum}(L) \geq 0$"

# Finding Facts at a Return Statement

- Consider this code

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) => {
  if (a >= 0 && b >= 0) {
    a = a - 1;
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```




- Facts no longer hold <u>throughout</u> straight-line code
- When we state a fact, we have to say <u>where</u> it holds

# Correctness Levels

| Description | Testing | Tools | Reasoning |
|---|---|---|---|
| no mutation | coverage | type checking | calculation<br>induction |
| local variable mutation | " | " | Floyd logic |
| heap state mutation | " | " | rep invariants |
| array mutation | " | " | for-any facts |

# Finding Facts at a Return Statement

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  if (a >= 0 && b >= 0) {
    {{ a ≥ 0 }}
    a = a - 1;
    {{ a ≥ -1 }}
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
```

- When we state a fact, we have to say <u>where</u> it holds
- {{ .. }} notation indicates facts true at that point
  - cannot assume those are true anywhere else

# Finding Facts at a Return Statement

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  if (a >= 0 && b >= 0) {
    {{ a ≥ 0 }}
    a = a - 1;
    {{ a ≥ -1 }}
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
```

- There are <u>mechanical</u> tools for moving facts around
  - "forward reasoning" says how they change as we move down
  - "backward reasoning" says how they change as we move up

# Finding Facts at a Return Statement

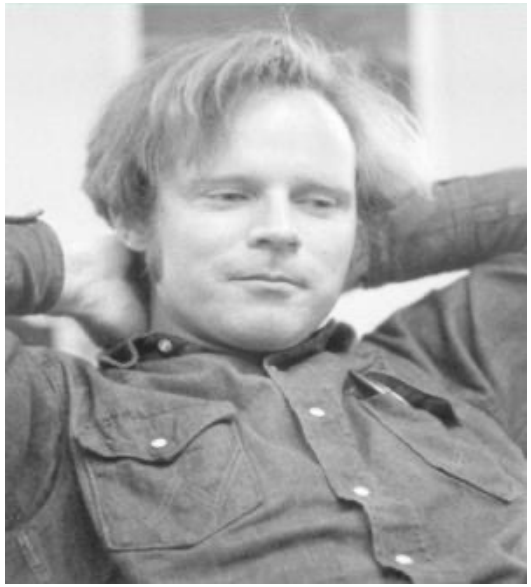
```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  if (a >= 0 && b >= 0) {
    {{ a ≥ 0 }}
    a = a - 1;
    {{ a ≥ -1 }}
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
```

- Professionals are *insanely* good at forward reasoning
  - "programmers are the Olympic athletes of forward reasoning"
  - you'll have an edge by learning backward reasoning too

# Floyd Logic

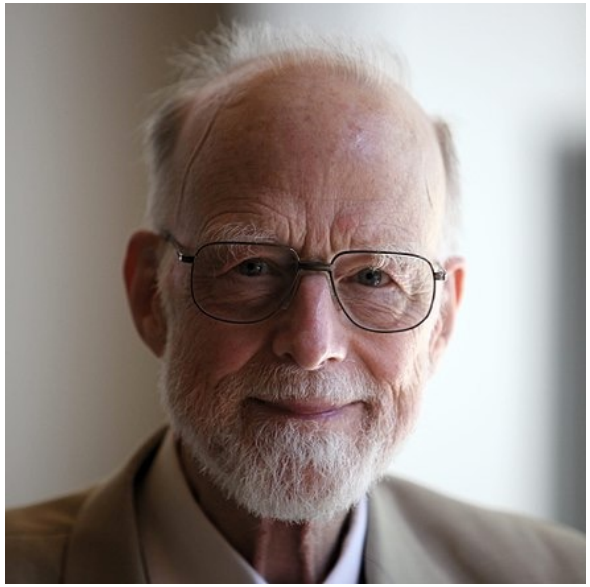
# Floyd Logic

- 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

# Floyd Logic Terminology

- The **program state** is the values of the variables

- An **assertion** (in {{ .. }}) is a T/F claim about the state
  - an assertion "holds" if the claim is true
  - assertions are *math* not code

    (we do our reasoning in math)

- Most important assertions:
  - **precondition**: claim about the state when the function starts
  - **postcondition**: claim about the state when the function ends

# Hoare Triples

- A **Hoare triple** has two assertions and some code

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

  - $P$ is the precondition, $Q$ is the postcondition
  - $S$ is the code

- Triple is "**valid**" if the code is correct:
  - $S$ takes *any* state satisfying $P$ into a state satisfying $Q$

    does not matter what the code does if $P$ does not hold initially
  - otherwise, the triple is invalid

# Correctness Example

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
  n = n + 3;
  return n * n;
};
```

# Correctness Example

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
  {{ n ≥ 1 }}
  n = n + 3;
  {{ n² ≥ 10 }}
  return n * n;
};
```

- Precondition and postcondition come from spec
- Remains to check that the triple is valid

# Hoare Triples with No Code

- Code could be empty:

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

- When is such a triple valid?
  - valid iff P implies Q
  - we already know how to check validity in this case: prove each fact in Q by calculation, using facts from P

# Hoare Triples with No Code

- **Code could be empty:**

$$\{\{\ a \geq 0,\ b \geq 0,\ L = \text{cons}(a, \text{cons}(b, \text{nil}))\ \}\}$$
$$\{\{\ \text{sum}(L) \geq 0\ \}\}$$

- **Check that P implies Q by calculation**

| | | |
|---|---|---|
| sum(L) | = sum(cons(a, cons(b, nil))) | since L = … |
| | = a + sum(cons(b, nil)) | def of sum |
| | = a + b + sum(nil) | def of sum |
| | = a + b | def of sum |
| | ≥ 0 + b | since a ≥ 0 |
| | ≥ 0 + 0 | since b ≥ 0 |
| | = 0 | |

# Hoare Triples with Code

- Code with code:

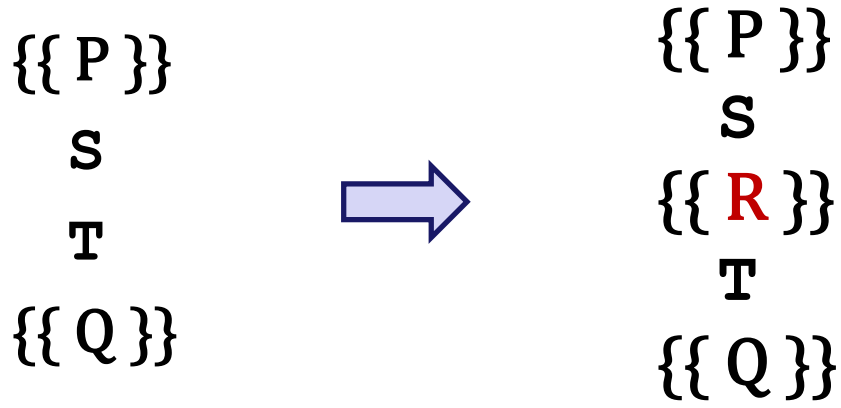
$$\{\{ P \}\}$$
$$s$$
$$\{\{ Q \}\}$$

- Easy if $s$ is empty, but what if not?

- We can use forward & backward reasoning
  - move the assertions toward each other until they meet
  - then we have a triple with no code

# Hoare Triples with Multiple Lines of Code

- Code with multiple lines:

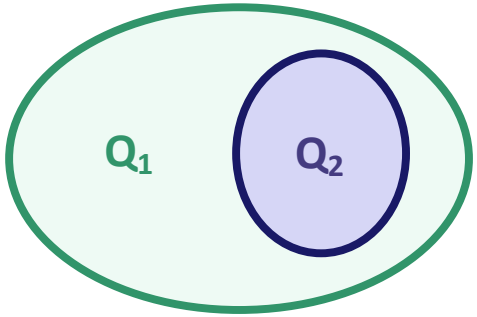



- Valid iff there exists an R making both triples valid
  - i.e., {{ P }} S {{ R }} is valid and {{ R }} T {{ Q }} is valid

- Will see next how to put these to good use…

# Recall: Stronger Assertions

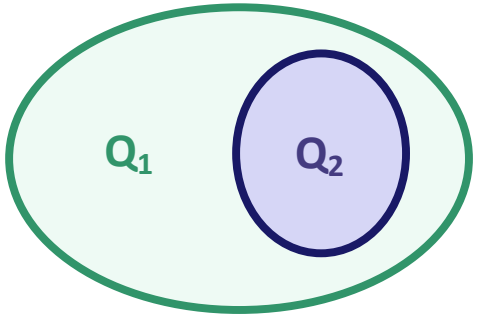
- **Assertion** is stronger iff it holds in a subset of states




- Stronger assertion implies the weaker one
  - stronger is a synonym for “implies”
  - weaker is a synonym for “is implied by”

# Recall: Stronger Assertions

- **Assertion** is **stronger** iff it holds in a subset of states




- **Weakest** possible assertion is "true" (all states)
  - an empty assertion ("") also means "true"

- **Strongest** possible assertion is "false" (no states!)

# Mechanical Reasoning Tools

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples

- **Forward** reasoning fills in postcondition

$$\{\{ P \}\}\ s\ \{\{ \_\_ \}\}$$

  - gives *strongest* postcondition making the triple valid

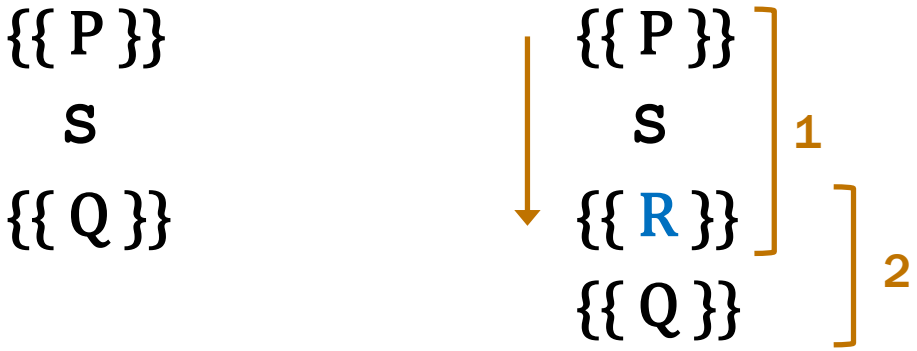
- **Backward** reasoning fills in precondition

$$\{\{ \_\_ \}\}\ s\ \{\{ Q \}\}$$

  - gives *weakest* precondition making the triple valid

# Correctness via Forward Reasoning

- Apply forward reasoning



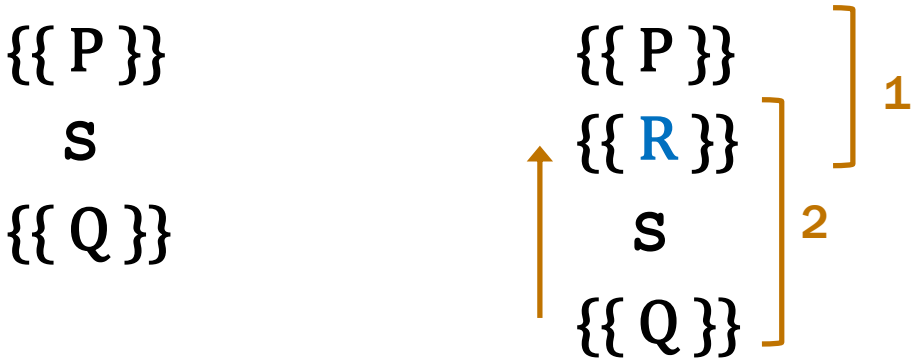

  - first triple is always valid
  - only need to check second triple

    just requires proving an implication (since no code is present)

- If second triple is invalid, the code is **incorrect**
  - true because R is the strongest assertion possible here

# Correctness via Backward Reasoning

- Apply backward reasoning




  - second triple is always valid
  - only need to check first triple

    just requires proving an implication (since no code is present)

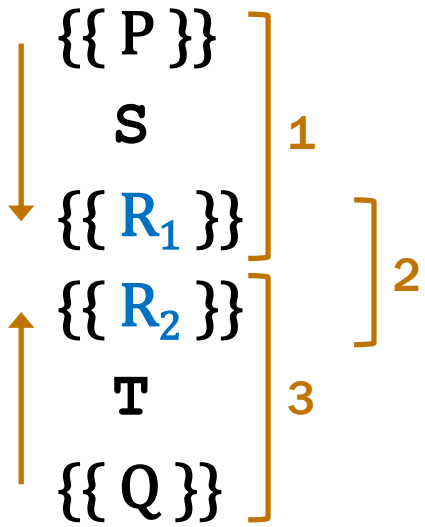
- If first triple is invalid, the code is **incorrect**
  - true because R is the weakest assertion possible here

# Mechanical Reasoning Tools

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples
- Reduce correctness to proving implications (again)
  - this was already true for functional code
  - will soon have the same for imperative code
- Implication will be false if the code is **incorrect**
  - reasoning can verify correct code
  - reasoning will never accept incorrect code

# Correctness via Forward & Backward

- Can use both types of reasoning on longer code




  - first and third triples is always valid
  - only need to check second triple

    verify that $R_1$ implies $R_2$

# Forward & Backward Reasoning

# Forward and Backward Reasoning

- Imperative code made up of
  - assignments (mutation)
  - conditionals
  - loops

- Anything can be rewritten with just these

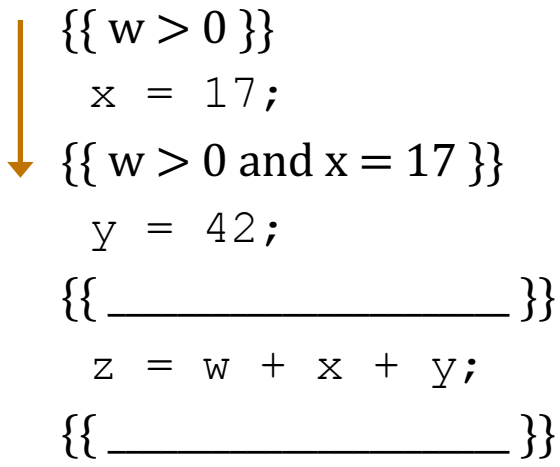
- We will learn forward / backward rules to handle them
  - will also learn a rule for function calls
  - once we have those, we are done

# Example Forward Reasoning through Assignments

```
{{ w > 0 }}
 x = 17;
{{ ____________________ }}
 y = 42;
{{ ____________________ }}
 z = w + x + y;
{{ ____________________ }}
```

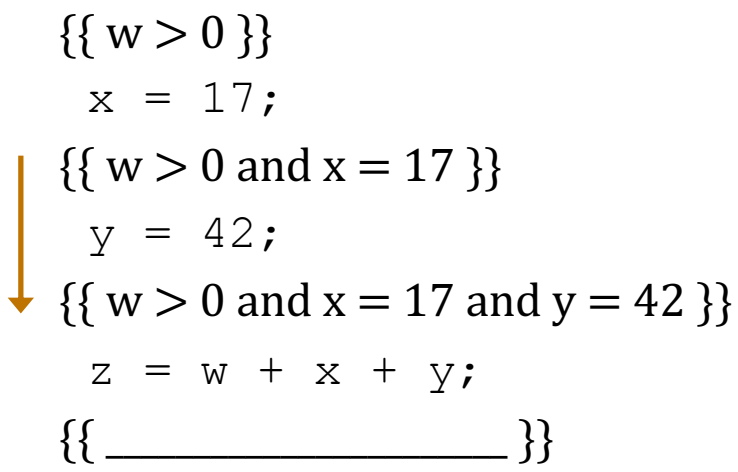
- What do we know is true after `x = 17` ?
  - want the strongest postcondition (most precise)

# Example Forward Reasoning through Assignments




- **What do we know is true after `x = 17` ?**
  - **`w` was not changed, so $w > 0$ is still true**
  - **`x` is now 17**
- **What do we know is true after `y = 42` ?**

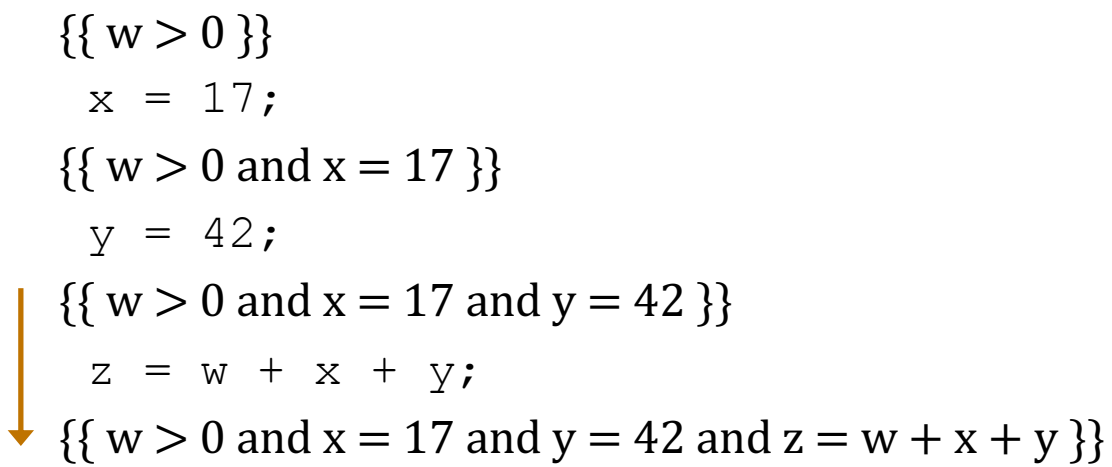
# Example Forward Reasoning through Assignments




- **What do we know is true after `y = 42` ?**
  - `w` **and** `x` **were not changed, so previous facts still true**
  - `y` **is now** `42`

- **What do we know is true after `z = w + x + y` ?**

# Example Forward Reasoning through Assignments




- **What do we know is true after** `z = w + x + y` **?**
  - w, x, **and y were not changed, so previous facts still true**
  - z **is now** $w + x + y$
- **Could also write** $z = w + 59$ **(since** $x = 17$ **and** $y = 42$**)**

# Example Forward Reasoning through Assignments

{{ $w > 0$ }}

```
x = 17;
```

{{ $w > 0$ and $x = 17$ }}

```
y = 42;
```

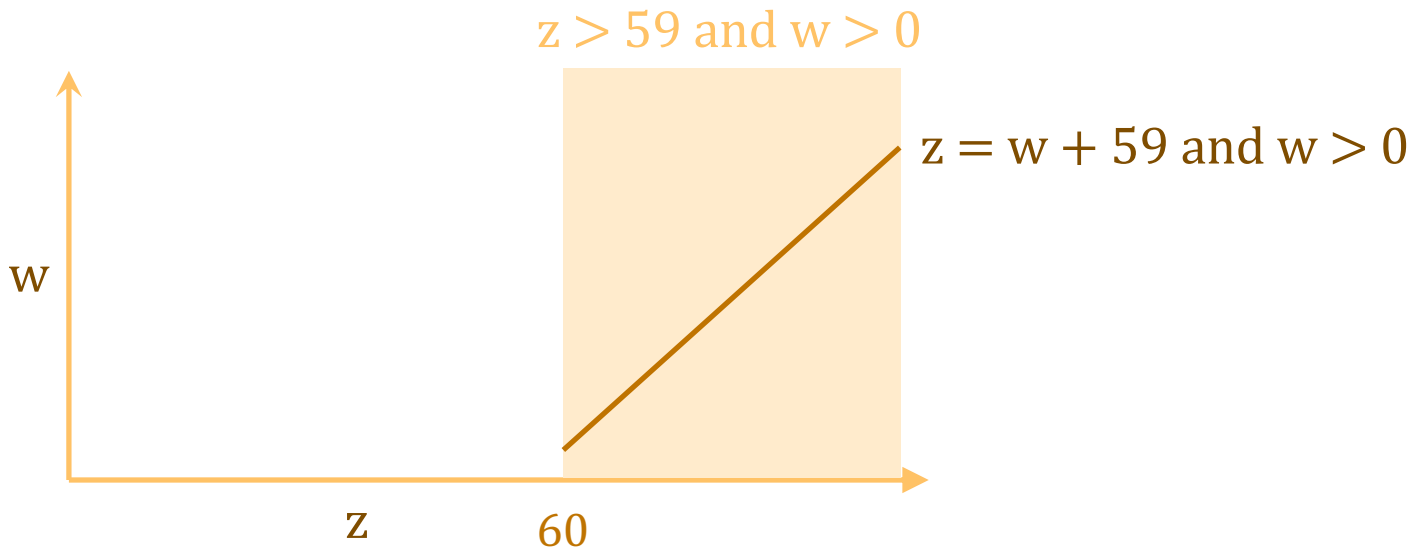
{{ $w > 0$ and $x = 17$ and $y = 42$ }}

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

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

- Could write $z = w + 59$, but <u>do not</u> write $z > 59$ !
  - that is true since $w > 0$, but…

# Example Forward Reasoning through Assignments




- **Could write $z = w + 59$, but <u>do not</u> write $z > 59$ !**
  - **that is true but it is not the strongest postcondition**

    correctness check could now fail even if the code is right

# Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
  int x = 17;
  int y = 42;
  int z = w + x + y;
  return z;
};
```

- Let's check correctness using Floyd logic...

# Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
  {{ w > 0 }}
  int x = 17;
  int y = 42;
  int z = w + x + y;
  {{ z > 59 }}
  return z;
};
```

- Reason forward...

# Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
  {{ w > 0 }}
  int x = 17;
  int y = 42;
  int z = w + x + y;
  {{ w > 0 and x = 17 and y = 42 and z = w + x + y }}
  {{ z > 59 }}
  return z;
};
```

- **Check implication:**

$$
\begin{aligned}
z &= w + x + y \\
&= w + 17 + y && \text{since } x = 17 \\
&= w + 59 && \text{since } y = 42 \\
&> 59 && \text{since } w > 0
\end{aligned}
$$

# Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
  int x = 17;
  int y = 42;
  int z = w + x + y;
  return z;
};
```

find facts by reading along path from top to return statement

- How about if we use our old approach?
- Known facts: $w > 0$, $x = 17$, $y = 42$, and $z = w + x + y$
- Prove that postcondition holds: $z > 59$

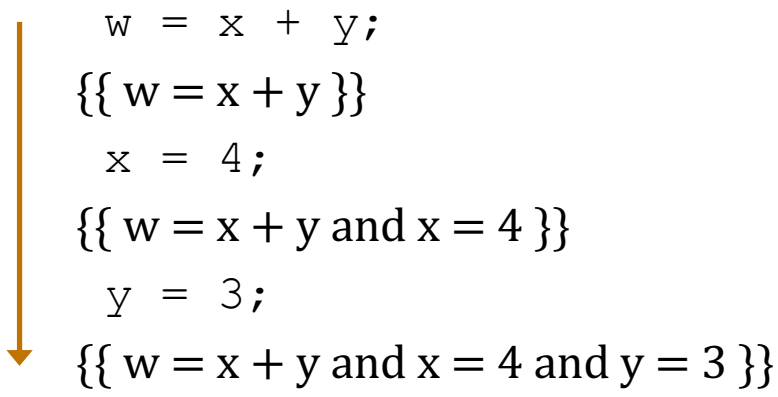
# Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
public int f(int w) {
  int x = 17;
  int y = 42;
  int z = w + x + y;
  return z;
};
```

- We've been doing forward reasoning already!
  - forward reasoning is (only) "and" with *no mutation*
- Line-by-line facts are for mutation (not "`final`")

# Forward Reasoning through Assignments

- Forward reasoning is trickier with mutation
  - gets harder if we mutate a variable

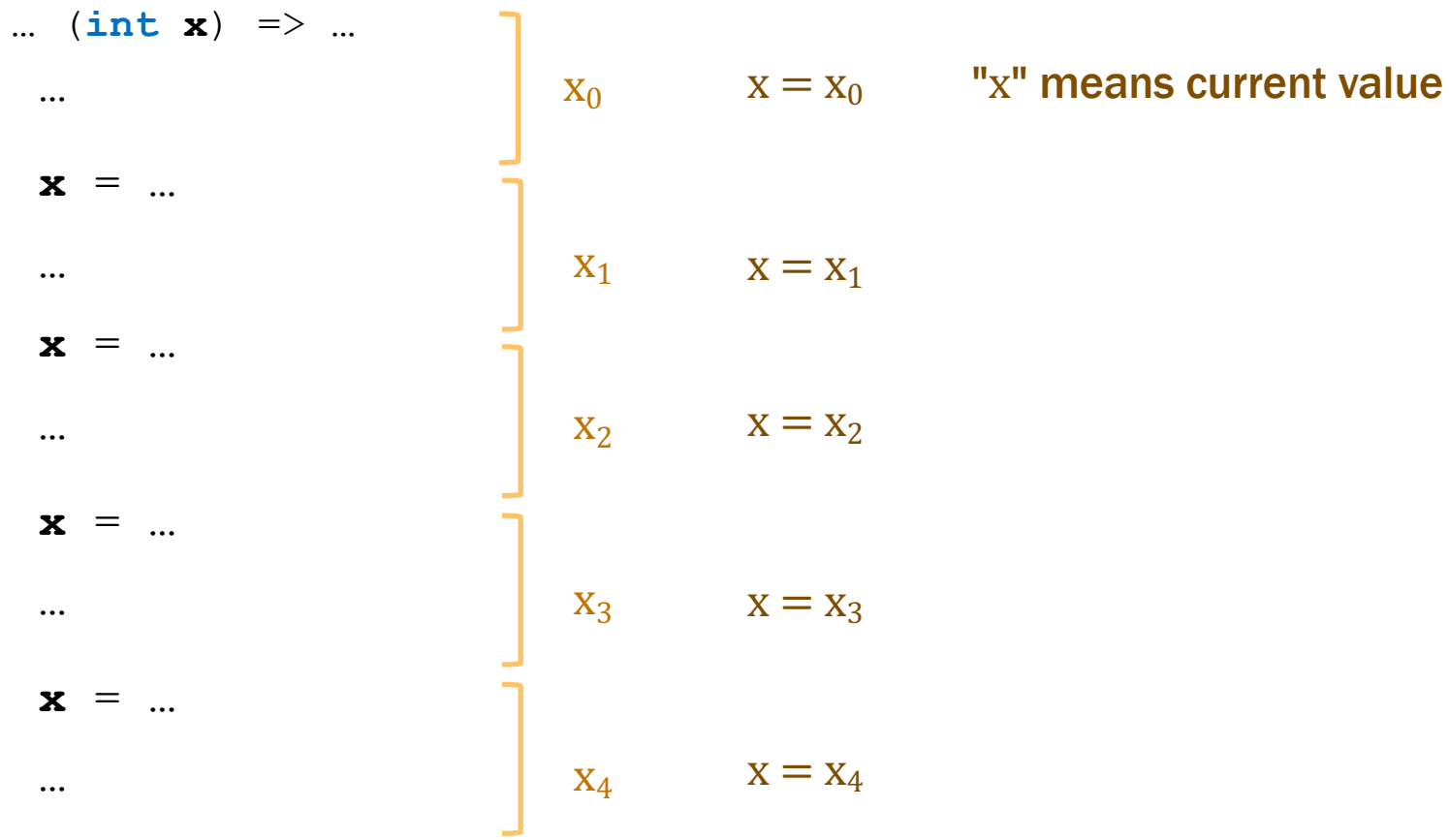



- Final assertion is not necessarily true
  - $w = x + y$ is true with their old values, not the new ones
  - changing the value of "x" can invalidate facts about x

    facts refer to the old value, not the new value
  - avoid this by using different names for old and new values

# Forward Reasoning through Assignments

- Can use subscripts to refer to values at different times



# Forward Reasoning through Assignments

- Rewrite existing facts to use names of earlier values
  - will use "x" and "y" to refer to current values
  - can use "$x_0$" and "$y_0$" (or other subscripts) for earlier values

$\{\{ w = x + y \}\}$

```
x = 4;
```

$\{\{ w = \mathbf{x_0} + y \text{ and } \mathbf{x} = 4 \}\}$

```
y = 3;
```

$\{\{ w = x_0 + \mathbf{y_0} \text{ and } x = 4 \text{ and } \mathbf{y} = 3 \}\}$

- Final assertion is now accurate
  - w is equal to the sum of the initial values of x and y

# Forward Reasoning through Assignments

- For assignments, general forward reasoning rule is

{{ P }}
  x = y;
{{ $P[x \mapsto x_k]$ and $x = y[x \mapsto x_k]$ }}

  - replace all "x"s in P and y with "$x_k$"s

- This process can be simplified in many cases
  - no need for $x_0$ if we can write it in terms of new value
  - e.g., if "$x = x_0 + 1$", then "$x_0 = x - 1$"
  - assertions will be easier to read without old values

    (Technically, this is weakening, but it's usually fine

    Postconditions usually do not refer to old values of variables.)

# Forward Reasoning through Assignments

- For assignments, general forward reasoning rule is

$\{\{ P \}\}$

`x = y;`

$\{\{ P[x \mapsto x_k] \text{ and } x = y[x \mapsto x_k] \}\}$

$x_k$ is name of previous value

- If $x_0 = f(x)$, then we can simplify this to

$\{\{ P \}\}$

`x = ... x ...;`

$\{\{ P[x \mapsto f(x)] \}\}$

no need for, e.g., "and $x = x_0 + 1$"

  - if assignment is "$x = x_0 + 1$", then "$x_0 = x - 1$"
  - if assignment is "$x = 2x_0$", then "$x_0 = x/2$"
  - does not work for integer division (an un-invertible operation)

# Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f = (int n) {
```

{{ $n \geq 1$ }}

```
  n = n + 3;
```

$n = n_0 + 3$ **means** $n - 3 = n_0$

{{ $n - 3 \geq 1$ }}

{{ $n^2 \geq 10$ }}

**check this implication**

```
  return n * n;
};
```

$n^2 \geq 4^2$ since $n - 3 \geq 1$ (i.e., $n \geq 4$)

$= 16$

$> 10$

**This is the preferred approach.**
**Avoid subscripts when possible.**

# Mutation in Straight-Line Code

- Alternative ways of writing this code:

```
n = n + 3;
return n * n;
```

```
final int n1 = n + 3;
return n1 * n1;
```

- Mutation in *straight-line* code is unnecessary
  - can always use different names for each value

- Why would we prefer the former?
  - seems like it might save memory…
  - but it doesn't!

    most compilers will turn the left into the right on their own (SSA form)

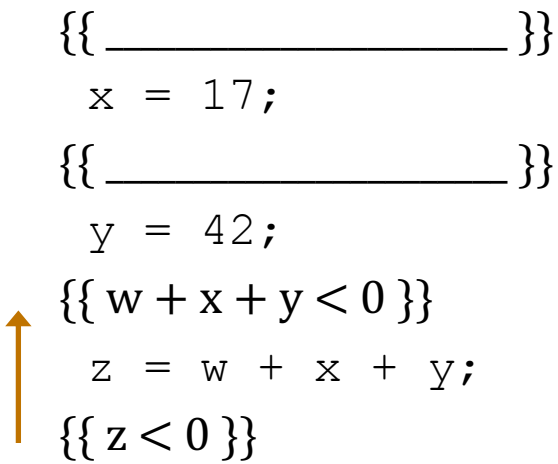
    it's better at saving memory than you are, so it does it itself

# Example Backward Reasoning with Assignments

```
{{ _____________ }}
 x = 17;
{{ _____________ }}
 y = 42;
{{ _____________ }}
 z = w + x + y;
{{ z < 0 }}
```

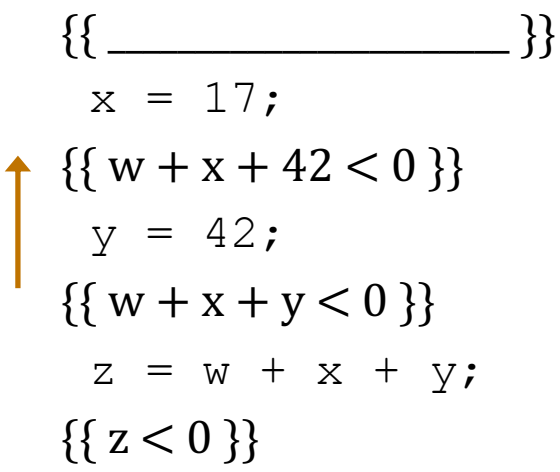
- **What must be true before** `z = w + x + y` **so** $z < 0$ **?**
  - **want the weakest precondition (most allowed states)**

# Example Backward Reasoning with Assignments




- **What must be true before** $z = w + x + y$ **so** $z < 0$ **?**
  - **must have** $w + x + y < 0$ **beforehand**
- **What must be true before** $y = 42$ **for** $w + x + y < 0$ **?**

# Example Backward Reasoning with Assignments




- **What must be true before** $y = 42$ **for** $w + x + y < 0$ **?**
  - **must have** $w + x + 42 < 0$ **beforehand**

- **What must be true before** $x = 17$ **for** $w + x + 42 < 0$ **?**

# Example Backward Reasoning with Assignments

$\{\{ w + 17 + 42 < 0 \}\}$

```
x = 17;
```

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

```
y = 42;
```

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

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

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

- **What must be true before** $x = 17$ **for** $w + x + 42 < 0$ **?**
  - **must have** $w + 59 < 0$ **beforehand**
- **All we did was substitute right side for the left side**
  - **e.g., substitute "**$w + x + y$**" for "**$z$**" in "**$z < 0$**"**
  - **e.g., substitute "**$42$**" for "**$y$**" in "**$w + x + y < 0$**"**
  - **e.g., substitute "**$17$**" for "**$x$**" in "**$w + x + 42 < 0$**"**

# Backward Reasoning through Assignments

- For assignments, backward reasoning is substitution

$\{\{\ Q[x \mapsto y]\ \}\}$

```
  x = y;
```

$\{\{\ Q\ \}\}$

  - just replace all the "$x$"s with "$y$"s
  - we will denote this substitution by $Q[x \mapsto y]$

- Mechanically simpler than forward reasoning
  - no need for subscripts

# Correctness Example by Backward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
  {{ n ≥ 1 }}
  n = n + 3;
  {{ n² ≥ 10 }}
  return n * n;
};
```

- Code is correct if this triple is valid...

# Correctness Example by Backward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
   {{ n ≥ 1 }}
   {{ (n + 3)² ≥ 10 }}
   n = n + 3;
   {{ n² ≥ 10 }}
   return n * n;
};
```

check this implication

$$(n+3)^2 \geq (1+3)^2 \quad \text{since } n \geq 1$$
$$= 16$$
$$> 10$$

# Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
public int f(int n) {
   {{ n ≥ 1 }}
   n = n + 3;
   {{ n – 3 ≥ 1 }}
   {{ n² ≥ 10 }}
   return n * n;
};
```

check this implication

$n^2 \geq 4^2$ since $n - 3 \geq 1$ (i.e., $n \geq 4$)

$= 16$

$> 10$

Forward reasoning produces known facts.
Backward reasoning produces fact to prove.

# Conditionals

# Conditionals in Functional Programming

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  if (a >= 0 && b >= 0) {
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

- Prior reasoning also included *conditionals*
  - what does that look like in Floyd logic?

# Conditionals in Floyd Logic

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
public int f(int a, int b) {
  {{ }}
  if (a >= 0 && b >= 0) {
    {{ a ≥ 0 and b ≥ 0 }}
    final List L = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

- Conditionals introduce extra facts in forward reasoning
  - simple “and” since nothing is mutated

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  return m;
}
```

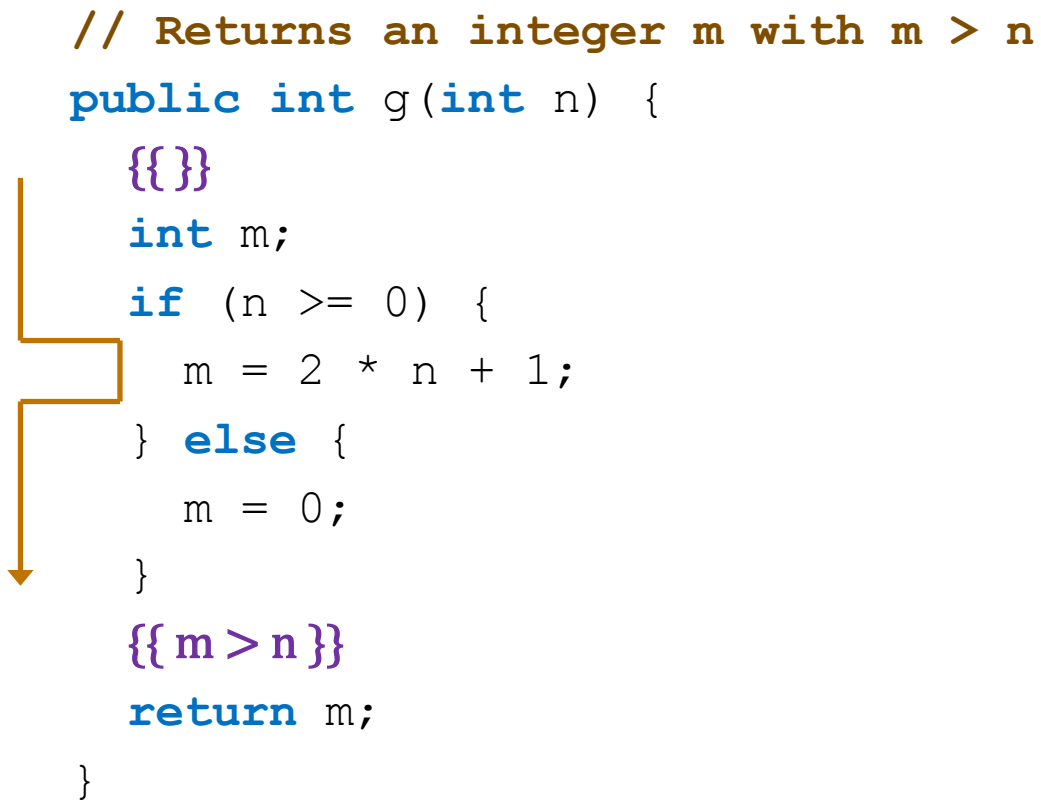
- Code like this was impossible without mutation
  - cannot write to a "`final`" after its declaration
- How do we handle it now?

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  return m;
}
```

- Reason *separately* about each path to a `return`
  - handle each path the same as before
  - but now there can be multiple paths to one `return`

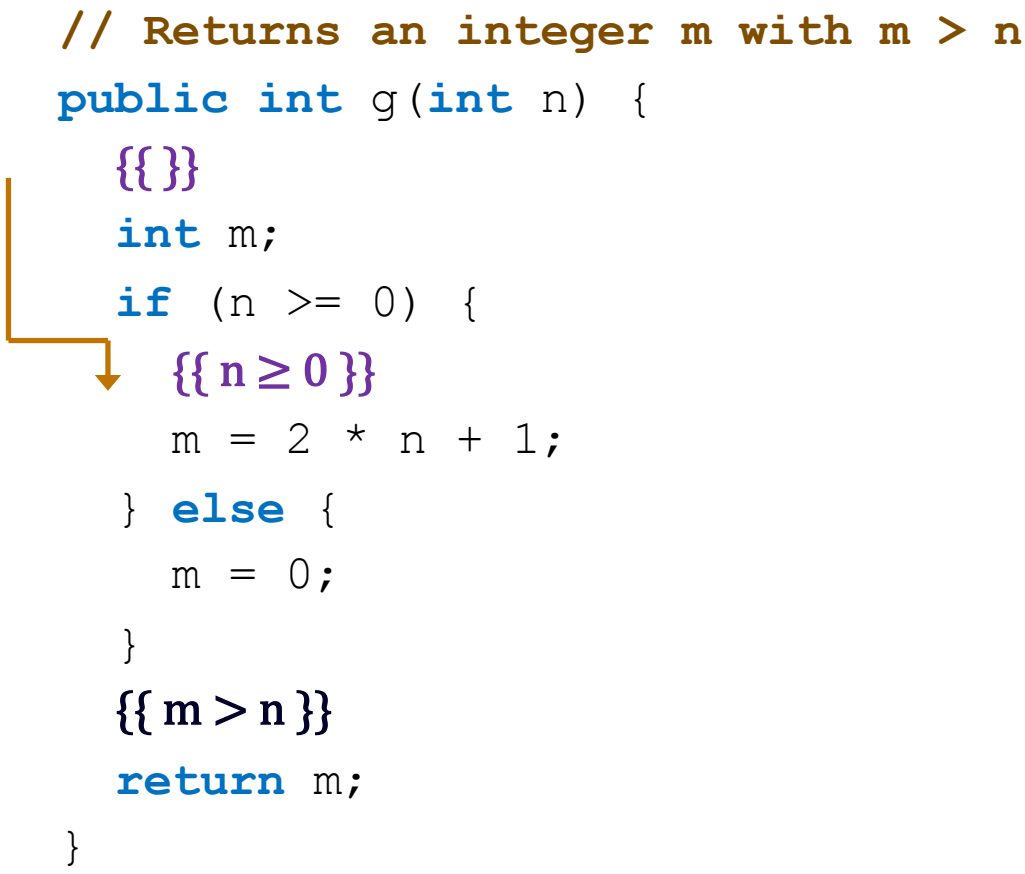
# Conditionals in Floyd Logic



```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

- Check correctness path through “then” branch

# Conditionals in Floyd Logic



# Conditionals in Floyd Logic

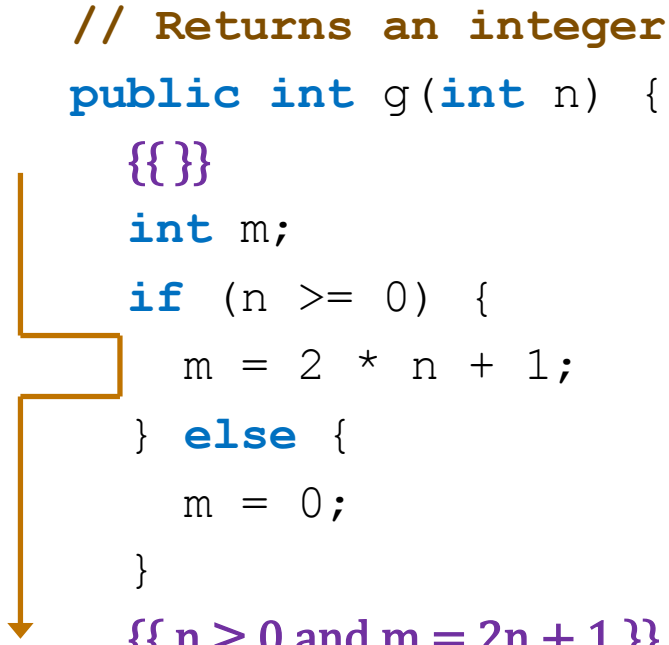
```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2 * n + 1;
    {{ n ≥ 0 and m = 2n + 1}}
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2 * n + 1;
    {{ n ≥ 0 and m = 2n + 1}}
  } else {
    m = 0;
  }
  {{ n ≥ 0 and m = 2n + 1 }}
  {{ m > n }}
  return m;
}
```

$$
\begin{aligned}
m &= 2n+1 \\
&> 2n && \text{since } 1 > 0 \\
&\geq n && \text{since } n \geq 0
\end{aligned}
$$

# Conditionals in Floyd Logic



```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  {{ n ≥ 0 and m = 2n + 1 }}
  {{ m > n }}
  return m;
}
```

- Note: no mutation, so we can do this in our head
  - read along the path, and collect all the facts

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  {{ n < 0 and m = 0 }}
  {{ m > n }}
  return m;
}
```

$m = 0$

$> n$ since $0 > n$

- Check correctness path through "else" branch
  - note: no mutation, so we can do this in our head

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
    {{ n ≥ 0 and m = 2n + 1 }}
  } else {
    m = 0;
    {{ n < 0 and m = 0 }}
  }
  {{ ____________________________________________ }}
  {{ m > n }}
  return m;
}
```

What do we know is true even if we don't know which branch was taken?

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    m = 0;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and m = 0) }}
  {{ m > n }}
  return m;
}
```

- The “or” means we must reason by cases anyway!

# Conditionals in Floyd Logic

```
{{ P }}
if (cond) {
    {{ P and cond }}
    S1
    {{ A }}
} else {
    {{ P and not cond }}
    S2
    {{ B }}
}
{{ A or B }}
{{ Q }}
```

- Postcondition is of the form {{ A or B }}
  - A being what we know if we had taken the `if` branch
  - B being what we know if we had taken the `else`

# Conditionals *Backward* in Floyd Logic

```
{{ P }}
{{ (A and cond) or (B and not cond) }}
if (cond) {
    {{ A }}
    S1
    {{ Q }}
} else {
    {{ B }}
    S2
    {{ Q }}
}
{{ Q }}
```

- **Precondition is of the form** {{ (A and cond) or (B and not cond) }}
  - A being what must hold if we take the `if` branch
  - B being what must hold if we take the `else`

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    return 0;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and ??) }}
  {{ m > n }}
  return m;
}
```

- What is the state after a "`return`"?

# Conditionals in Floyd Logic

```
// Returns an integer m with m > n
public int g(int n) {
  {{ }}
  int m;
  if (n >= 0) {
    m = 2 * n + 1;
  } else {
    return 0;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and false) }}
  {{ m > n }}
  return m;
}
```

simplifies to just $n \geq 0$ and $m = 2n + 1$

- State after a "`return`" is false (no states)

# Conditionals With Returns

- Latter rule for "`if` .. `return`" is useful:

```
{{ P }}
if (cond)
  return something;
{{ P and not cond }}
…
return something else;
```

- Only reach the line after the "`if`" if `cond` was false

- Only one path to each "`return`" statement
  - forward reason to the "`return`" inside the "`if`"
  - forward reason to the "`return`" after the "`if`"

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  }
  {{ ____________________________________ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

How many paths can the code take?

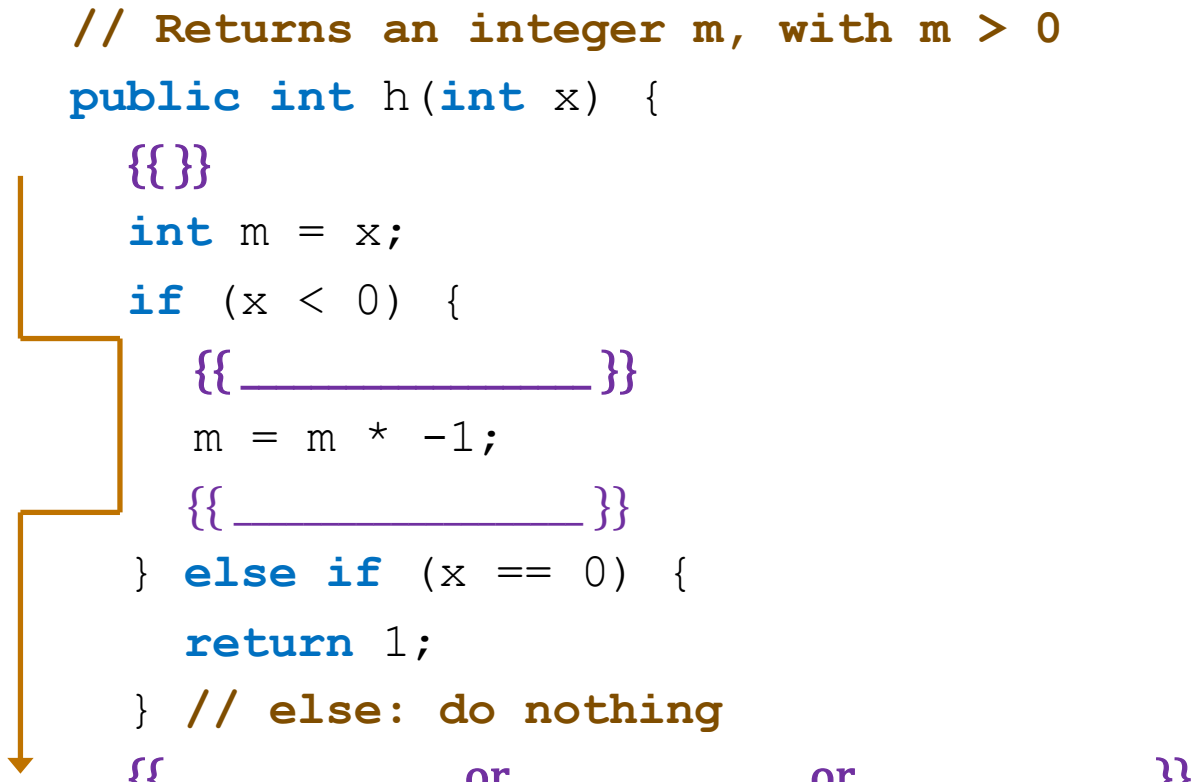
# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } else {
    // do nothing
  }
  {{ _____ or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

3 paths! `else` branch is not written out, but it's there implicitly

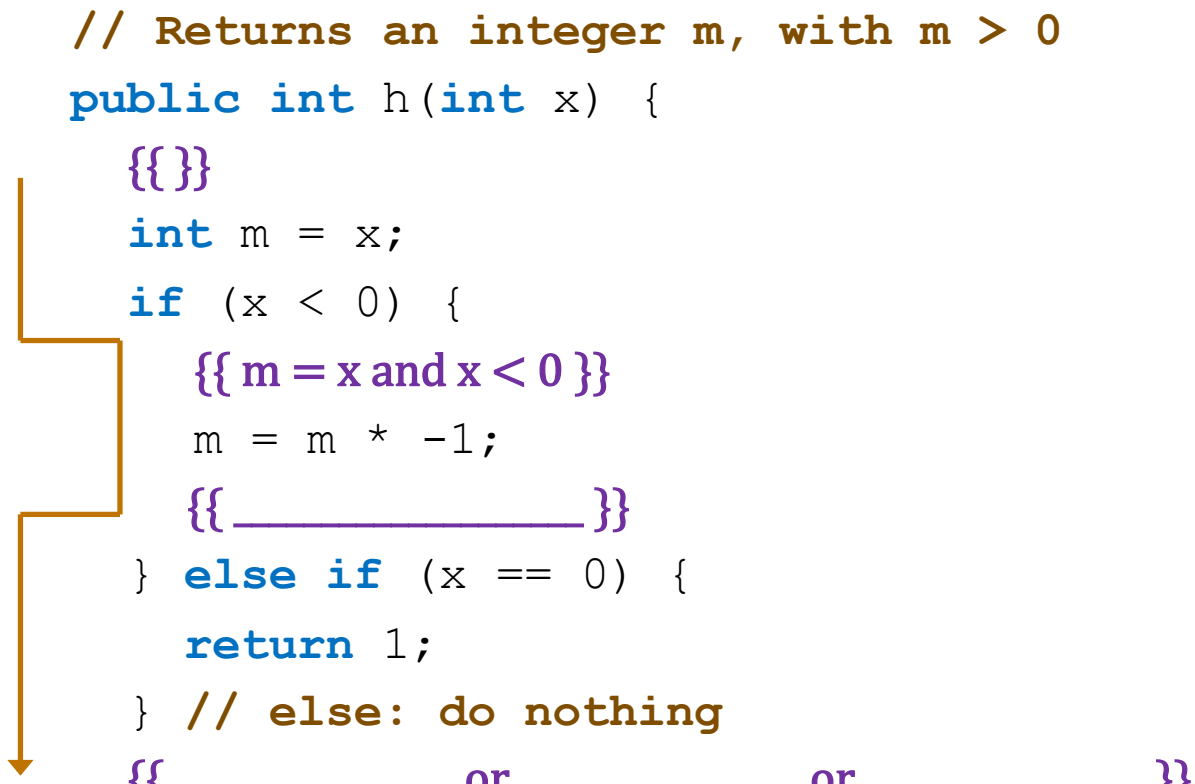
After the conditional, there are 3 sets of facts that could be true

# Conditionals in Floyd Logic



```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    {{ _____ }}
    m = m * -1;
    {{ _____ }}
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ _____ or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

# Conditionals in Floyd Logic



```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    {{ m = x and x < 0 }}
    m = m * -1;
    {{ _____ }}
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ _____ or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

# Conditionals in Floyd Logic

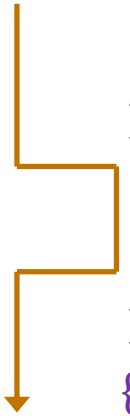
```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    {{ m = x and x < 0 }}
    m = m * -1;
    {{ m = - x and x < 0 }}
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    {{ _____ }}
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

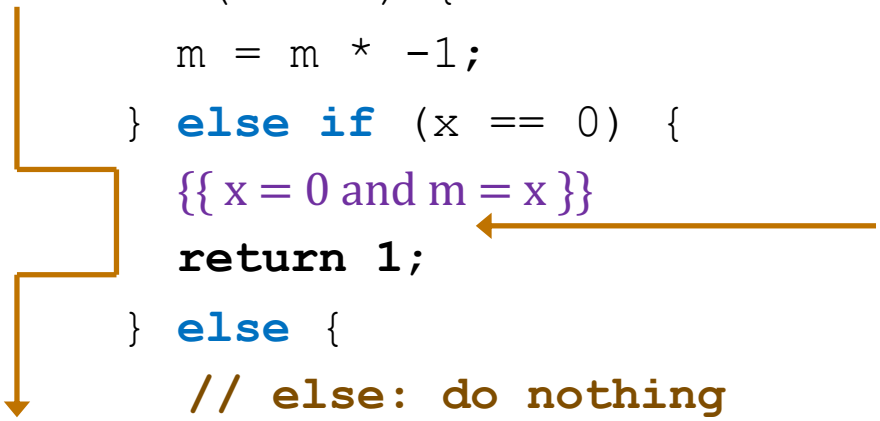
# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    {{ x = 0 and m = x }}
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or _____ or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```



# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    {{ x = 0 and m = x }}
    return 1;
  } else {
    // else: do nothing
  }
  {{ (m = - x and x < 0) or (x = 0 and m = x and false) or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```



Must prove that post condition holds here

false: no states can reach beyond return

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing

  {{ (m = - x and x < 0) or _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

What do we know in implicit else case?
When *neither* of the then cases were entered

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing

  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  {{ _____ }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

Can reason backward and forward and meet in the middle

# Conditionals in Floyd Logic

```
// Returns an integer m, with m > 0
public int h(int x) {
  {{ }}
  int m = x;
  if (x < 0) {
    m = m * -1;
  } else if (x == 0) {
    return 1;
  } // else: do nothing
  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  {{ m + 1 > 0 }}
  m = m + 1;
  {{ m > 0 }}
  return m;
}
```

check this implication

Does the set of facts we know at this point in the program satisfy what must be true to reach our post condition

# Conditionals in Floyd Logic

- Prove by cases

  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  {{ m + 1 > 0 }}

  **Case 1:** m = - x and x < 0

  m + 1 = -x + 1 since m = -x
  > 1 since x < 0
  > 0

  **Case 2:** x > 0 and m = x

  m + 1 = x + 1 since m = x
  > 1 since x > 0
  > 0

- Already proved for the branch with the return, so proved the postcondition holds, in general

# Loops

# Correctness of Loops

- Assignment and condition reasoning is mechanical
- Loop reasoning **cannot** be made mechanical
  - no way around this

    (**311 alert**: this follows from Rice's Theorem)
- Thankfully, one *extra* bit of information fixes this
  - need to provide a "loop invariant"
  - with the invariant, reasoning is again mechanical

# Loop Invariants

- Loop invariant is true **every time** at the top of the loop

```
{{ Inv: I }}
while (cond) {
  S
}
```

  - must be true when we get to the top the first time
  - must remain true each time execute S and loop back up

- Use "**Inv:**" to indicate a loop invariant

  otherwise, this only claims to be true the first time at the loop

# Loop Invariants

- Loop invariant is true **every time** at the top of the loop

```
{{ Inv: I }}
while (cond) {
  s
}
```

  - must be true $0$ times through the loop (at top the first time)
  - if true $n$ times through, must be true $n+1$ times through

- Why do these imply it is always true?
  - follows by structural induction (on $\mathbb{N}$)

# Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  S
}
{{ Q }}
```

- How do we check validity with a loop invariant?
  - intermediate assertion splits into *three* triples to check

# Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  S
}
{{ Q }}
```

1. I holds initially

## Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits

# Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
    S
  {{ I }}
}
{{ Q }}
```

1. I holds initially

2. S preserves I

## Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits

# Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
    S
  {{ I }}
}
{{ I and not cond }}
{{ Q }}
```

1. I holds initially
2. S preserves I
3. Q holds when loop exits

## Splits correctness into three parts

| | | |
|---|---|---|
| 1. | I holds initially | implication |
| 2. | S preserves I | forward/back then implication |
| 3. | Q holds when loop exits | implication |

# Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  S
}
{{ Q }}
```

**Formally, invariant split this into three Hoare triples:**

1. {{ P }} {{ I }} — **I holds initially**
2. {{ I and cond }} **S** {{ I }} — **S preserves** I
3. {{ I and not cond }} {{ Q }} — Q **holds when loop exits**

# Loop Correctness Example 1

- This loop claims to calculate $n^2$

```
{{ }}
int j = 0;
int s = 0;
{{ Inv: s = j^2 }}
while (j != n) {
  j = j + 1;
  s = s + j + j - 1;
}
{{ s = n^2 }}
```

Easy to get this wrong!

- might be initializing "j" wrong ($j = 1$?)
- might be exiting at the wrong time ($j \neq n-1$?)
- might have the assignments in wrong order
- …

Fact that we need to check 3 implications is a strong indication that more bugs are possible.

# Loop Correctness Example 1

- This loop claims to calculate $n^2$

```
{{ }}
int j = 0;
int s = 0;
{{ Inv: s = j^2 }}
while (j != n) {
  j = j + 1;
  s = s + j + j - 1;
}
{{ s = n^2 }}
```

**Loop Idea**

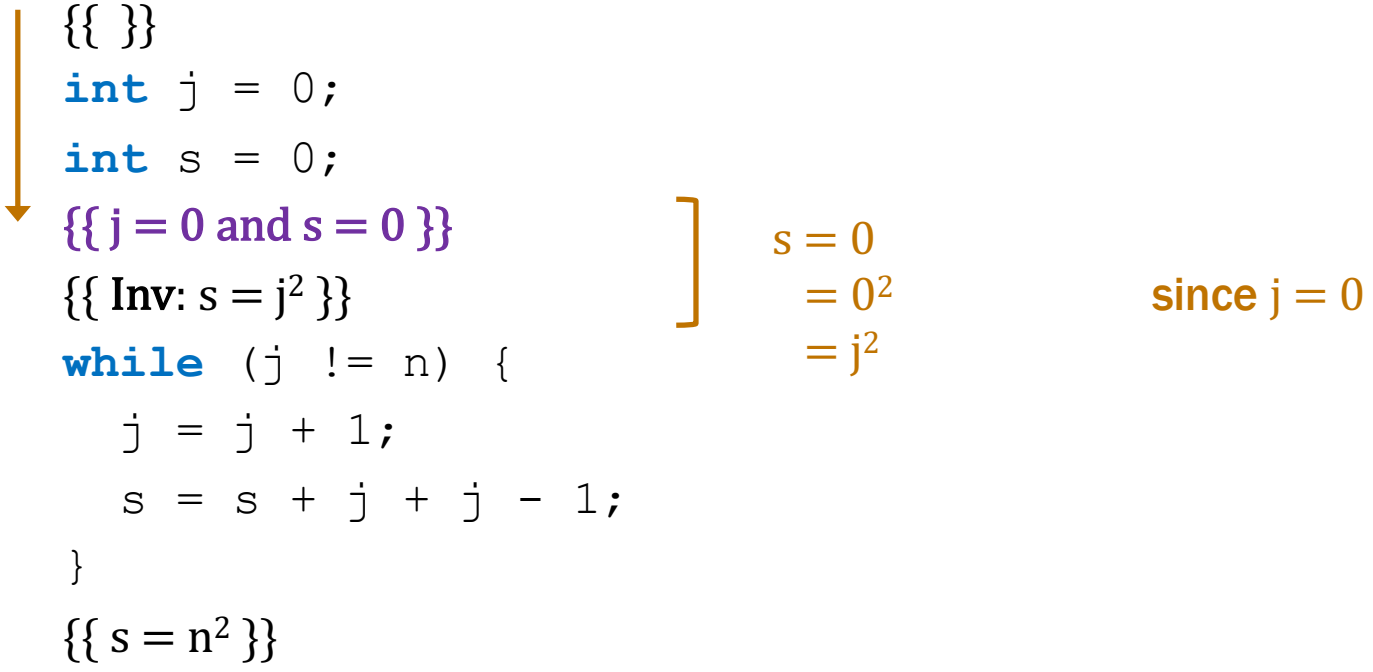
- move j from 0 to n
- keep track of $j^2$ in s

| j | s |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| … | … |

Loop Invariant formalizes the Loop Idea

# Loop Correctness Example 1

- This loop claims to calculate $n^2$



# Loop Correctness Example 1

- This loop claims to calculate $n^2$

{{ Inv: $s = j^2$ }}

```
while (j != n) {
  j = j + 1;
  s = s + j + j - 1;
}
```

{{ $s = j^2$ and $j = n$ }}

{{ $s = n^2$ }}

$s = j^2$

$= n^2$ since $j = n$

# Loop Correctness Example 1

- This loop claims to calculate $n^2$

{{ Inv: $s = j^2$ }}

```
while (j != n) {
```

{{ $s = j^2$ and $j \neq n$ }}

```
  j = j + 1;
  s = s + j + j - 1;
```

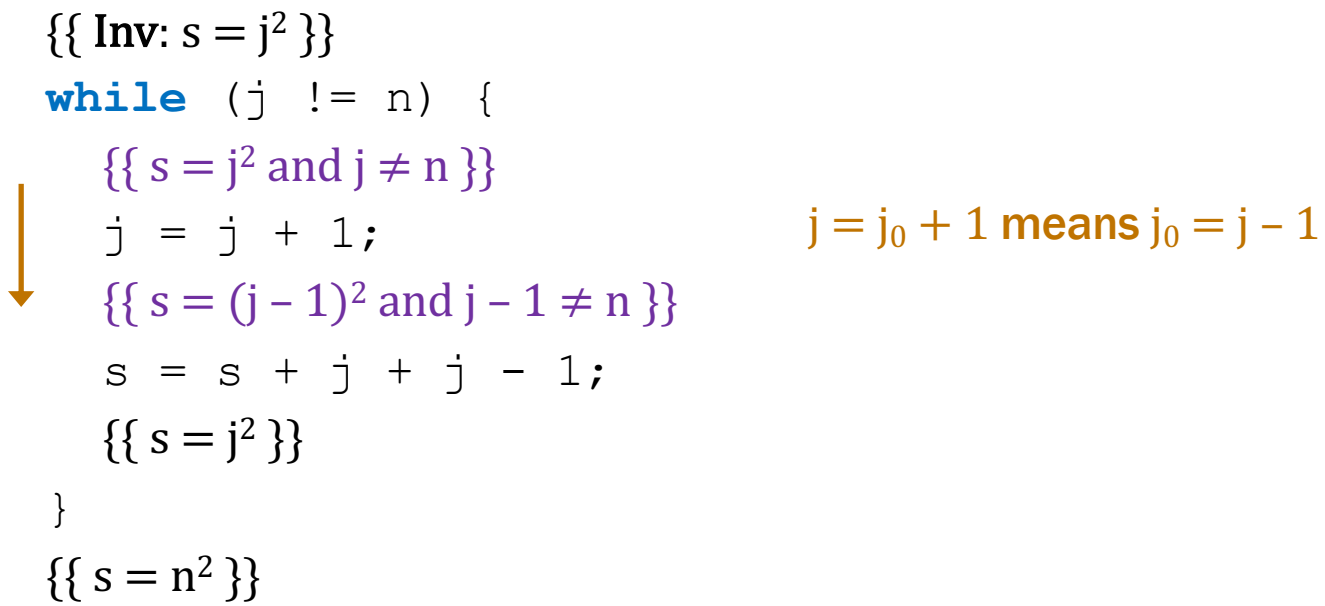
{{ $s = j^2$ }}

```
}
```

{{ $s = n^2$ }}

# Loop Correctness Example 1

- This loop claims to calculate $n^2$



# Loop Correctness Example 1

- This loop claims to calculate $n^2$

{{ Inv: $s = j^2$ }}
```
while (j != n) {
```
{{ $s = j^2$ and $j \neq n$ }}
```
  j = j + 1;
```
{{ $s = (j-1)^2$ and $j - 1 \neq n$ }}
```
  s = s + j + j - 1;
```
{{ $s - 2j + 1 = (j-1)^2$ and $j - 1 \neq n$ }}
{{ $s = j^2$ }}
```
}
```
{{ $s = n^2$ }}

$s = s_0 + 2j - 1$ **means** $s_0 = s - 2j + 1$

# Loop Correctness Example 1

- This loop claims to calculate $n^2$

{{ Inv: $s = j^2$ }}
```
while (j != n) {
```
{{ $s = j^2$ and $j \neq n$ }}
```
  j = j + 1;
```
{{ $s = (j-1)^2$ and $j - 1 \neq n$ }}
```
  s = s + j + j - 1;
```
{{ $s - 2j + 1 = (j-1)^2$ and $j - 1 \neq n$ }}
{{ $s = j^2$ }}
```
}
```
{{ $s = n^2$ }}

$s = 2j - 1 + (j-1)^2$
$= 2j - 1 + j^2 - 2j + 1$
$= j^2$

since $s - 2j + 1 = (j-1)^2$

# Loop Correctness Example 2

- Recursive function to calculate sum of list

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- This loop claims to calculate it as well:

$\{\{ L = L_0 \}\}$

```
int s = 0;
```

$\{\{ \textbf{Inv}: \text{sum}(L_0) = s + \text{sum}(L) \}\}$

```
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{ s = \text{sum}(L_0) \}\}$

**Loop Idea**

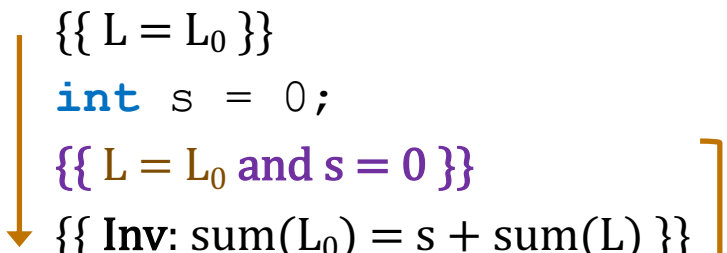
- move through L front-to-back
- keep sum of *prior* part in s

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the invariant holds initially**




$$\begin{aligned} &\text{sum}(L_0) \\ &= \text{sum}(L) && \textbf{since } L = L_0 \\ &= 0 + \text{sum}(L) \\ &= s + \text{sum}(L) && \textbf{since } s = 0 \end{aligned}$$

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the postcondition holds at loop exit**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L != null) {
  s = s + L.hd;
  L = L.tl;
}
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = \text{nil}$ }}

{{ $s = \text{sum}(L_0)$ }}

$$\begin{aligned} &\text{sum}(L_0) \\ &= s + \text{sum}(L) \\ &= s + \text{sum}(\text{nil}) && \textbf{since } L = \text{nil} \\ &= s && \textbf{def of } \text{sum} \end{aligned}$$

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L != null) {
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L \neq \text{nil}$ }}

```
  s = s + L.hd;
  L = L.tl;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
}
```

$L \neq \text{nil}$ **means** $L = L.\text{hd} :: L.\text{tl}$

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L != null) {
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = L.hd :: L.tl$ }}

```
  s = s + L.hd;
  L = L.tl;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
}
```

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L != null) {
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = L.hd :: L.tl$ }}

```
  s = s + L.hd;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L.tl)$ }}

```
  L = L.tl;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
}
```

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L != null) {
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = L.hd :: L.tl$ }}

{{ $\text{sum}(L_0) = s + L.hd + \text{sum}(L.tl)$ }}

```
s = s + L.hd;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L.tl)$ }}

```
L = L.tl;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
}
```

# Loop Correctness Example 2

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}
**while** (L != **null**) {
  {{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = L.hd :: L.tl$ }}
  {{ $\text{sum}(L_0) = s + L.hd + \text{sum}(L.tl)$ }}
  s = s + L.hd;
  {{ $\text{sum}(L_0) = s + \text{sum}(L.tl)$ }}
  L = L.tl;
  {{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}
}

$\text{sum}(L_0)$
$= s + \text{sum}(L)$
$= s + \text{sum}(L.hd :: L.tl)$ **since** $L = L.hd :: L.tl$
$= s + L.hd + \text{sum}(L.tl)$ **def of** sum

# Loop Correctness Example 3

- **Recursive function to check if $y$ appears in list $L$**

$$
\begin{array}{lll}
\text{contains}(y, \text{nil}) & := \text{false} & \\
\text{contains}(y, x :: L) & := \text{true} & \textbf{if } x = y \\
\text{contains}(y, x :: L) & := \text{contains}(y, L) & \textbf{if } x \neq y
\end{array}
$$

- **This loop claims to calculate it as well:**

```
{{ Inv: contains(y, L0) = contains(y, L) }}
while (L != null) {
  if (L.hd == y)
    return true;
  L = L.tl;
}
return false;
```

**Loop Idea**

- move through $L$ front-to-back
- answer remains the same as on the original list $L_0$
- can only do that if $y$ is *not* found

# Loop Correctness Example 3

- **Check that the invariant holds initially**

{{ $L_0 = L$ }}
{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}

```
while (L != null) {
  if (L.hd == y)
    return true;
  L = L.tl;
}
return false;
```

contains(y, $L_0$)
= contains(y, L) **since** $L_0 = L$

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

# Loop Correctness Example 3

- **Check that the invariant implies the postcondition**

{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}

```
while (L != null) {
  if (L.hd == y)
    return true;
  L = L.tl;
}
```

{{ contains(y, $L_0$) = contains(y, L) and L = nil }}

{{ contains(y, $L_0$) = false }}

```
return false;
```

contains(y, $L_0$)
= contains(y, L)
= contains(y, nil) **since** L = nil
= false **def of** contains

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

# Loop Correctness Example 3

- **Check that the body preserves the invariant**

{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}

```
while (L != null) {
```

{{ contains(y, $L_0$) = contains(y, L) and L ≠ nil }}

```
  if (L.hd == y)
    return true;
  L = L.tl;
```

{{ contains(y, $L_0$) = contains(y, L) }}

```
}
return false;
```

L ≠ nil **means** L = L.hd :: L.tl

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

# Loop Correctness Example 3

- **Check that the body preserves the invariant**

{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}
```
while (L != null) {
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl }}
```
  if (L.hd == y)
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl and L.hd = y }}
{{ contains(y, $L_0$) = true }}
```
    return true;
  L = L.tl;
```
{{ contains(y, $L_0$) = contains(y, L) }}
```
}
return false;
```

contains(y, $L_0$)
= contains(y, L)
= contains(y, L.hd :: L.tl) **since** L = L.hd :: L.tl
= true **since** y = L.hd

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

# Loop Correctness Example 3

- **Check that the body preserves the invariant**

{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}
```
while (L != null) {
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl }}
```
  if (L.hd == y)
```
{{ contains(y, $L_0$) = true }}
```
    return true;
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl and L.hd ≠ y }}
```
  L = L.tl;
```
{{ contains(y, $L_0$) = contains(y, L) }}
```
}
return false;
```

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

# Loop Correctness Example 3

- **Check that the body preserves the invariant**

{{ **Inv**: contains(y, $L_0$) = contains(y, L) }}
```
while (L != null) {
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl }}
```
  if (L.hd == y)
```
{{ contains(y, $L_0$) = true }}
```
    return true;
```
{{ contains(y, $L_0$) = contains(y, L) and L = L.hd :: L.tl and L.hd ≠ y }}
{{ contains(y, $L_0$) = contains(y, L.tl) }}
```
  L = L.tl;
```
{{ contains(y, $L_0$) = contains(y, L) }}
```
}
return false;
```

contains(y, nil) := false
contains(y, x :: L) := true **if** x = y
contains(y, x :: L) := contains(y, L) **if** x ≠ y

contains(y, $L_0$)
= contains(y, L)
= contains(y, L.hd :: L.tl) **since** L = L.hd :: L.tl
= contains(y, L.tl) **since** y ≠ L.hd

# Loop Correctness Example 4

- **Declarative spec of** $\mathrm{sqrt}(x)$

  **return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

  - **precondition that** $x$ **is positive:** $0 < x$
  - **precondition that x is not too large:** $x < 10^{12} = (10^6)^2$

# Loop Correctness Example 4

return $y \in \mathbb{Z}$ such that $(y-1)^2 < x \leq y^2$

- This loop claims to calculate it:

```
int a = 0;
int b = 1000000;
{{ Inv: a^2 < x ≤ b^2 }}
while (a != b - 1) {
  int m = (a + b) / 2;
  if (m*m < x) {
    a = m;
  } else {
    b = m;
  }
}
return b;
```

**Loop Idea**

- maintain a range $a \ldots b$ with $x$ in the range $a^2 \ldots b^2$

# Loop Correctness Example 4

return $y \in \mathbb{Z}$ such that $(y-1)^2 < x \leq y^2$

- **Check that the invariant holds initially:**

{{ **Pre:** $0 < x \leq 10^{12}$ }}

```
int a = 0;
int b = 1000000;
```

{{ **Inv:** $a^2 < x \leq b^2$ }}

```
while (a != b - 1) {
  …
}
return b;
```

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the invariant holds initially:**

{{ **Pre:** $0 < x \leq 10^{12}$ }}

```
int a = 0;
int b = 1000000;
```

{{ $0 < x \leq 10^{12}$ and $a = 0$ and $b = 10^6$ }}

{{ **Inv:** $a^2 < x \leq b^2$ }}

```
while (a != b - 1) {
  …
}
return b;
```

$a^2 = 0^2$ **since** $a = 0$
$= 0$
$< x$

$x < 10^{12}$
$= (10^6)^2$
$= b^2$ **since** $b = 10^6$

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the postcondition hold after exit**

{{ **Inv:** $a^2 < x \leq b^2$ }}
**while** (a != b - 1) {
  …
}
{{ $a^2 < x \leq b^2$ **and** $a = b - 1$ }}
{{ $(b-1)^2 < x \leq b^2$ }}
**return** b;

**Does** $(y-1)^2 < x < y^2$ **hold with** $y = b$**?**

$(b-1)^2$
$= a^2$ **since** $a = b - 1$
$< x$

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the body preserves the invariant:**

```
{{ Inv: a² < x ≤ b² }}
while (a != b - 1) {
  {{ a² < x ≤ b² and a ≠ b - 1 }}
  int m = (a + b) / 2;
  if (m*m < x) {
    a = m;
  } else {
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the body preserves the invariant:**

```
{{ Inv: a² < x ≤ b² }}
while (a != b - 1) {
  {{ a² < x ≤ b² and a ≠ b – 1 }}
  int m = (a + b) / 2;
  if (m*m < x) {
    {{ a² < x ≤ b² and a ≠ b – 1 and m² < x }}
    a = m;
  } else {
    {{ a² < x ≤ b² and a ≠ b – 1 and x ≤ m² }}
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the body preserves the invariant:**

```
{{ Inv: a² < x ≤ b² }}
while (a != b - 1) {
  int m = (a + b) / 2;
  if (m*m < x) {
    {{ a² < x ≤ b² and a ≠ b – 1 and m² < x }}
    {{ m² < x ≤ b² }}
    a = m;
  } else {
    {{ a² < x ≤ b² and a ≠ b – 1 and x ≤ m² }}
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

Immediate!

# Loop Correctness Example 4

**return** $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \leq y^2$

- **Check that the body preserves the invariant:**

```
{{ Inv: a² < x ≤ b² }}
while (a != b - 1) {
  int m = (a + b) / 2;
  if (m*m < x) {
    a = m;
  } else {
    {{ a² < x ≤ b²  and a ≠ b – 1 and x ≤ m² }}
    {{ a² < x ≤ m² }}
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

Immediate!

Correctness of binary search is pretty easy *once* you have the invariant clear!

# Termination

- This analysis does not check that the code **terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop does exit

- Termination follows from the running time analysis
  - e.g., if the code runs in $O(n^2)$ time, then it terminates
  - an infinite loop would be O(infinity)
  - any finite bound on the running time proves it terminates

- Normal to also analyze the running time of our code, and we get termination already from that analysis

# Correctness of Loops

- With straight-line code and conditionals, if the triple is not valid…
  - the code is **wrong**
  - there is *some* test case that will prove it

    (doesn't mean we found that case in our tests, but it exists)

- With loops, if the triples are not valid…
  - the code is **wrong** *with that invariant*
  - there may **not** be any test case that proves it

    the code may behave correctly on all inputs
  - the code could be right but with a *different* invariant

- Loops are inherently more complicated