Reminders

• HW1 is due **tomorrow**

• HW2 will be posted on Wednesday
  – covers loops
  – it is harder, so more time given

• Reading Quiz 1 is due Friday

• For those still trying to add the course:
  [https://goo.gl/forms/Ws8LW8UtvBjH0RiA2](https://goo.gl/forms/Ws8LW8UtvBjH0RiA2)
Weaker vs Stronger

If “whenever P1 holds, P2 also holds”, then:
- P1 is called **stronger** than P2
- P2 is called **weaker** than P1

- It is more (or at least as) “difficult” to satisfy P1
  - the program states where P1 holds are a subset of the
    states where P2 holds
- P1 puts more constraints on program states
- P1 is a stronger set of requirements

- We do not always have P1 stronger than P2 or vice versa!
  - most assertions are incomparable
Applications to Hoare Logic

• Suppose:
  – \{\{ P \} \} S \{\{ Q \} \} is valid and
  – some \( P_1 \) is *stronger* than \( P \) and
  – some \( Q_1 \) is *weaker* than \( Q \)

• Then these are all valid too:
  – \{\{ P_1 \} \} S \{\{ Q \} \}
    • a state where \( P_1 \) holds is one where \( P \) also holds
  – \{\{ P \} \} S \{\{ Q_1 \} \}
    • a state where \( Q \) holds is one where \( Q_1 \) also holds
  – \{\{ P_1 \} \} S \{\{ Q_1 \} \}
Weakest preconditions

• Suppose we know $Q$ and $S$
• There are potentially many $P$ such that $\{\{ P \}\} S \{\{ Q \}\}$ is valid
• Would be ideal if there were a unique weakest precondition $P$
  – most general assumptions under which $S$ makes $Q$ hold
  – get a valid triple for $P_1$ if and only if $P_1$ implies $P$
• Amazingly, without loops, for any $S$ and $Q$, this exists!
  – we denote this by $wp(S,Q)$
  – can be found by general rules
• This is just \textbf{backward reasoning}!
Rules for weakest preconditions

• \(wp(x = e, Q)\) is \(Q[x=e]\)
  – Example: \(wp(x = 2*y, x > 4) = 2*y > 4\), i.e., \(y > 2\)

• \(wp(S1; S2, Q)\) is \(wp(S1, wp(S2, Q))\)
  – i.e., let \(R\) be \(wp(S2, Q)\) and overall \(wp\) is \(wp(S1, R)\)
  – Example: \(wp(y = x+1, wp(z = y+1, z > 2)) = wp(y = x+1, y+1 > 2) = (x+1)+1 > 2\) or equivalently \(x > 0\)

• \(wp(\text{if } b \text{ then } S1 \text{ else } S2, Q)\) is this logic formula:
  \((b \text{ and } wp(S1, Q)) \text{ or } (!b \text{ and } wp(S2, Q))\)
  – you need \(wp(S1, Q)\) if \(S1\) is executed and \(wp(S2, Q)\) if \(S2\) is
  – you can often simplify the result considerably
More Examples

• If $S$ is $x = y \times y$ and $Q$ is $x > 4$,
  then $wp(S,Q)$ is $y \times y > 4$, i.e., $|y| > 2$

• If $S$ is $y = x + 1; \ z = y - 3$; and $Q$ is $z = 10$,
  then $wp(S,Q)$ …
  $= wp(y = x + 1; \ z = y - 3, z = 10)$
  $= wp(y = x + 1, wp(z = y - 3, z = 10))$
  $= wp(y = x + 1, y-3 = 10)$
  $= wp(y = x + 1, y = 13)$
  $= x+1 = 13$
  $= x = 12$
Bigger Example

\[ S \text{ is if } (y < 5) \{ x = y*y; \} \text{ else } \{ x = y+1; \} \]

\[ wp(S, x \geq 9) \]
\[ = (y < 5 \text{ and } wp(x = y*y, x \geq 9)) \]
\[ \text{ or } (y \geq 5 \text{ and } wp(x = y+1, x \geq 9)) \]
\[ = (y < 5 \text{ and } y*y \geq 9) \]
\[ \text{ or } (y \geq 5 \text{ and } y+1 \geq 9) \]
\[ = (y \leq -3) \text{ or } (y \geq 3 \text{ and } y < 5) \]
\[ \text{ or } (y \geq 8) \]
Review: Straight-line Code
Forward & Backward Reasoning

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
S \\
\{\{ ? \}\}
\end{align*}
\]

- P is what we know initially
- Work downward
- Determine strongest post-condition if P is true initially

Backward reasoning

\[
\begin{align*}
\{\{ ? \}\} \\
S \\
\{\{ Q \}\}
\end{align*}
\]

- Q is what we want at the end
- Work upward
- Determine weakest precondition to make Q hold
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
x &= \text{expr} \\
\{\{ ? \}\}
\end{align*}
\]
Assignment Rule

Forward reasoning

\[
\{ P \} \\
x = \text{expr}; \\
\{ P \text{ and } x = \text{expr} \}
\]

- adds another known fact
- these tend to accumulate…
  - many are irrelevant

(above assumes \( x \) not used in P)
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{ \text{P} \} \\
\text{x = expr;}
\{ \text{P and } x = \text{expr } \}
\end{align*}
\]

• adds another known fact
• these tend to accumulate…
  – many are irrelevant

(above assumes \( x \) not used in \( P \))

Backward reasoning

\[
\begin{align*}
\{ \text{?} \} \\
\text{x = expr;}
\{ \text{Q } \}
\end{align*}
\]
Assignment Rule

Forward reasoning

\[
\begin{align*}
\{ \{ P \} \} \\
x &= \text{expr}; \\
\{ \{ P \text{ and } x = \text{expr} \} \}
\end{align*}
\]

• adds another known fact
• these tend to accumulate…
  – many are irrelevant

(above assumes \( x \) not used in \( P \))

Backward reasoning

\[
\begin{align*}
\{ \{ Q[x=\text{expr}] \} \} \\
x &= \text{expr}; \\
\{ \{ Q \} \}
\end{align*}
\]

• just substitution
• most general conditions for getting \( Q \) after \( x = \text{expr} \);
Assignment Example

Forward reasoning

\{ w = 3 \}
\ x = y - 5 ;
\{ ?? \}
Assignment Example

Forward reasoning

\[
\{\ w = 3 \}\ \\
x = y - 5; \\
\{\ w = 3 \text{ and } x = y - 5 \}\n\]
Assignment Example

Forward reasoning

\[
\begin{align*}
\{ w &= 3 \} \\
x &= y - 5; \\
\{ w &= 3 \text{ and } x = y - 5 \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ ? \} \\
x &= y - 5; \\
\{ w &= x + 5 \}
\end{align*}
\]
Assignment Example

Forward reasoning

\{\ w = 3 \ \}\n\ x = y - 5; \\
\{\ w = 3 \text{ and } x = y - 5 \ \}\n
Backward reasoning

\{\ w = y \ \}\n\ x = y - 5; \\
\{\ w = x + 5 \ \}
Sequence Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
S1 \\
S2 \\
\{\{ ? \}\}
\end{align*}
\]
Sequence Rule

Forward reasoning

{{ P }}
S1
{{ ? }}
S2
{{ ? }}
Sequence Rule

Forward reasoning

\[
\{ \text{P} \} \\
S_1 \\
\{ \text{P1} \} \\
S_2 \\
\{ ? \}
\]
Sequence Rule

Forward reasoning

\[
\begin{align*}
&P \\
&S_1 \\
&P_1 \\
&S_2 \\
&P_2
\end{align*}
\]
Sequence Rule

Forward reasoning

\[
\{ \{ \text{P} \} \} \\
S1 \\
\{ \{ \text{P1} \} \} \\
S2 \\
\{ \{ \text{P2} \} \}
\]

Backward reasoning

\[
\{ \{ \text{?} \} \} \\
S1 \\
\{ \{ \text{Q} \} \}
\]
Sequence Rule

Forward reasoning

\{ \{ P \} \}
S1
\{ \{ P1 \} \}
S2
\{ \{ P2 \} \}

Backward reasoning

\{ \{ ? \} \}
S1
\{ \{ ? \} \}
S2
\{ \{ Q \} \}
Sequence Rule

Forward reasoning

\{\{ P \}\}  \rightarrow  S1
\{\{ P1 \}\}  \rightarrow  S2
\{\{ P2 \}\}

Backward reasoning

\{\{ ? \}\}  \leftarrow  S1
\{\{ Q2 \}\}  \leftarrow  S2
\{\{ Q \}\}
Sequence Rule

Forward reasoning

\{ \{ P \} \}
S1
\{ \{ P1 \} \}
S2
\{ \{ P2 \} \}

Backward reasoning

\{ \{ Q1 \} \}
S1
\{ \{ Q2 \} \}
S2
\{ \{ Q \} \}
If-Statement Rule

Forward reasoning

{{ P }}
if (cond)
  S1
else
  S2
{{ ? }}
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
\text{if } (\text{cond}) \\
\quad \{\{ P \text{ and } \text{cond} \}\} \\
\quad S1 \\
\text{else} \\
\quad \{\{ P \text{ and not } \text{cond} \}\} \\
\quad S2 \\
\{\{ ? \}\}
\end{align*}
\]
If-Statement Rule

Forward reasoning

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

Forward reasoning

\[
\begin{align*}
&\{\{ P \} \} \\
&\text{if (cond)} \\
&\quad \{\{ P \land \text{cond} \} \} \\
&\quad S1 \\
&\quad \{\{ P1 \} \} \\
&\text{else} \\
&\quad \{\{ P \land \lnot \text{cond} \} \} \\
&\quad S2 \\
&\quad \{\{ P2 \} \} \\
&\{\{ P1 \lor P2 \} \}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
\text{if (cond)} \\
\{\{ P \land \text{cond} \}\} \\
S1 \\
\{\{ P1 \}\} \\
\text{else} \\
\{\{ P \land \neg \text{cond} \}\} \\
S2 \\
\{\{ P2 \}\} \\
\{\{ P1 \lor P2 \}\}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{\{ ? \}\} \\
\text{if (cond)} \\
S1 \\
\text{else} \\
S2 \\
\{\{ Q \}\}
\end{align*}
\]
If-Statement Rule

Forward reasoning

\[
\begin{align*}
\{\{ P \}\} \\
\text{if (cond)} \\
\{\{ P \land \text{cond} \}\} \\
S1 \\
\{\{ P1 \}\} \\
\text{else} \\
\{\{ P \land \text{not cond} \}\} \\
S2 \\
\{\{ P2 \}\} \\
\{\{ P1 \lor P2 \}\}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{\{ ? \}\} \\
\text{if (cond)} \\
S1 \\
\{\{ Q \}\} \\
\text{else} \\
S2 \\
\{\{ Q \}\} \\
\{\{ Q \}\}
\end{align*}
\]
If-Statement Rule

Forward reasoning

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

Backward reasoning

```plaintext
{?}
if (cond)
    {Q1}
S1
    {Q}
else
    {Q2}
S2
    {Q}
{Q}
```
If-Statement Rule

Forward reasoning

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

Backward reasoning

```plaintext
{{ 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

```plaintext
if (x >= 0)
  {x >= 0}
  y = x;
else
  {x < 0}
  y = -x;
{?}
```
If-Statement Example

Forward reasoning

```ruby
{{ }}
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

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

```c
{{
    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

{}{}

Backward reasoning

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

{} y = |x|
If-Statement Example

Forward reasoning

\[
\begin{align*}
\{ \} & \\
\text{if } (x \geq 0) & \\
\{ \{ x \geq 0 \} \} & \\
\quad y = x; & \\
\{ \{ x \geq 0 \text{ and } y = x \} \} & \\
\text{else} & \\
\{ \{ x < 0 \} \} & \\
\quad y = -x; & \\
\{ \{ x < 0 \text{ and } y = -x \} \} & \\
\{ \{ y = |x| \} \}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ \{ ? \} \} & \\
\text{if } (x \geq 0) & \\
\quad y = x; & \\
\{ \{ y = |x| \} \} & \\
\text{else} & \\
\quad y = -x; & \\
\{ \{ y = |x| \} \} & \\
\{ \{ y = |x| \} \}
\end{align*}
\]
If-Statement Example

Forward reasoning

\[
\{\ \} \\
if \ (x \geq 0) \\
\{\ x \geq 0 \} \\
y = x; \\
\{\ x \geq 0 \text{ and } y = x \} \\
\]
else
\[
\{\ x < 0 \} \\
y = -x; \\
\{\ x < 0 \text{ and } y = -x \} \\
\{\ y = |x| \} \\
\]

Backward reasoning

\[
\{\ ? \} \\
if \ (x \geq 0) \\
\{\ x = |x| \} \\
y = x; \\
\{\ y = |x| \} \\
\]
else
\[
\{\ -x = |x| \} \\
y = -x; \\
\{\ y = |x| \} \\
\{\ y = |x| \} \\
\]
If-Statement Example

Forward reasoning

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

Backward reasoning

```latex
\{\{ ? \}\}
if (x >= 0)
  \{\{ x >= 0 \}\}
y = x;
  \{\{ y = |x| \}\}
else
  \{\{ x <= 0 \}\}
y = -x;
  \{\{ y = |x| \}\}
\{\{ y = |x| \}\}
```
If-Statement Example

Forward reasoning

\[
\begin{aligned}
\{{}\}
\text{if} \ (x \geq 0) \\
\{{} x \geq 0 \}\} \\
y = x; \\
\{{} x \geq 0 \text{ and } y = x \}\} \\
\text{else} \\
\{{} x < 0 \}\} \\
y = -x; \\
\{{} x < 0 \text{ and } y = -x \}\} \\
\{{} y = |x| \}\}
\end{aligned}
\]

Backward reasoning

\[
\begin{aligned}
\{{}\} \ (x \geq 0 \text{ and } x \geq 0) \text{ or } (x < 0 \text{ and } x \leq 0) \}\}
\text{if} \ (x \geq 0) \\
\{{} x \geq 0 \}\} \\
y = x; \\
\{{} y = |x| \}\} \\
\text{else} \\
\{{} x \leq 0 \}\} \\
y = -x; \\
\{{} y = |x| \}\} \\
\{{} y = |x| \}\}
\end{aligned}
\]
If-Statement Example

Forward reasoning

\[
\begin{align*}
\{ \} \\
\text{if } (x \geq 0) \\
\{ \{ x \geq 0 \} \} \\
y = x; \\
\{ \{ x \geq 0 \text{ and } y = x \} \} \\
\text{else} \\
\{ \{ x < 0 \} \} \\
y = -x; \\
\{ \{ x < 0 \text{ and } y = -x \} \} \\
\{ y = |x| \} \\
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{ x \geq 0 \text{ or } x < 0 \} \\
\text{if } (x \geq 0) \\
\{ \{ x \geq 0 \} \} \\
y = x; \\
\{ \{ y = |x| \} \} \\
\text{else} \\
\{ \{ x \leq 0 \} \} \\
y = -x; \\
\{ \{ y = |x| \} \} \\
\{ y = |x| \} \\
\end{align*}
\]
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  
{}  

Backward reasoning

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

y = |x|
Verifying Correctness (*Inspection*)

Two different ways of checking \( \{\{ P \}\} \implies \{\{ Q \}\} \)

*Use forward reasoning:*

\[
\begin{align*}
&\{\{ P \}\} \\
&\implies \\
&\{\{ Q' \}\}
\end{align*}
\]

- Find \( Q' \) assuming \( P \).
- Check that \( Q' \) implies \( Q \).
  - weaken postcondition

*Use backward reasoning:*

\[
\begin{align*}
&\{\{ P' \}\} \\
&\implies \\
&\{\{ Q \}\}
\end{align*}
\]

- Find \( P' \) that produces \( Q \).
- Check that \( P \) implies \( P' \).
  - strengthen precondition

You know how to verify correctness of straight-line code.
You will do this on HW1.
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

{{{ }}
if (x >= 0)
   y = div(x,2);
else
   y = -div(-x+1,2);
{{ 2y = x or 2y = x - 1 }}

Assume that $\text{div}(a,b)$ computes $a/b$ rounded toward zero.

Code to compute $x/2$ rounded toward minus infinity (usual division).
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

\[
\begin{align*}
\{ & \} \\
& \text{if } (x \geq 0) \\
& \{ & x \geq 0 \} \\
& y = \text{div}(x, 2); \\
& \text{else} \\
& \{ & x < 0 \} \\
& y = -\text{div}(-x+1, 2); \\
& \{ & 2y = x \text{ or } 2y = x - 1 \}
\end{align*}
\]
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
if (x >= 0)
    y = div(x, 2);
else
    y = -div(-x + 1, 2);
```

```c
{{ 2y = x or 2y = x - 1 }}
```

{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
{{
if (x >= 0)
    {{ x >= 0 }}
y = div(x, 2);
    {{ 2y = x or 2y = x - 1 }}
else
    {{ x < 0 }}
y = -div(-x+1, 2);
    {{ 2y = x or 2y = x - 1 }}
{{ 2y = x or 2y = x - 1 }}
```
Using Both Forward & Backward

Also possible to check correctness by mixing forward & backward:

```c
if (x >= 0)
    {{ x >= 0 }}
```  
```c
y = div(x, 2);
{{ 2y = x or 2y = x - 1 }}
```  
```c
else
    {{ x < 0 }}
```  
```c
y = -div(-x+1, 2);
{{ 2y = x or 2y = x - 1 }}
```  
```c
{{ 2y = x or 2y = x - 1 }}
```
One caveat

• With forward reasoning, there is a problem with assignments:
  – changing a variable can affect other assumptions

```c
{{
  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 }}
}

• But clearly we do not know w = 7!
• The assertion w = x + y means the original values of x and y
One Fix

- Use different names for the values at different points
  - common to use subscripts to distinguish these
  - on every assignment, rename references to the old values

```
{{ }}
w = x + y;
{{ w = x + y }}
x = 4;
{{ w = x₀ + y and x = 4 }}
y = 3;
{{ w = x₀ + y₀ and x = 4 and y = 3 }}
```
Useful example: swap

• Consider code for a swapping x and y

{\color{red}{}}
\begin{align*}
tmp &= x; \\
{\color{blue}{}}
\begin{align*}
tmp &= x \\
{\color{red}{}}
\begin{align*}
x &= y; \\
{\color{blue}{}}
\begin{align*}
tmp &= x_0 \text{ and } x = y \\
y &= tmp; \\
{\color{blue}{}}
\begin{align*}
tmp &= x_0 \text{ and } x = y_0 \text{ and } y = tmp \\
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}

• Post condition implies $x = y_0$ and $y = x_0$
• I.e., their final values are equal to the original values swapped
Loops
Loop Invariant

A **loop invariant** is one that always holds at the top of the loop:

\[
\begin{align*}
\{ \text{Inv: } I \} \\
\text{while (cond)} \\
S \\
\end{align*}
\]

- It holds when we first get to the loop.
- It holds each time we execute \( S \) and come back to the top.

Notation: I’ll use “Inv:” to indicate a loop invariant.
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \}\} \text{ while } (\text{cond}) \ S \ \{\{ Q \}\}
\]

This triple is valid iff: there is a loop invariant \( I \) such that

\[
\{\{ P \}\}, \{\{ \text{Inv: } I \}\} \text{ while } (\text{cond}) \ S \ \{\{ Q \}\}
\]

- I holds initially
- I holds each time we execute \( S \)
- Q holds when I holds and \text{cond} is false
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \}\} \text{ while (cond)} S \{\{ Q \}\}
\]

This triple is valid iff: there is a loop invariant I such that

\[
\{\{ P \}\} \{\{ \text{Inv: I} \}\} \text{ while (cond)} S \{\{ Q \}\}
\]

- P implies I
- I holds each time we execute S
- Q holds when I holds and cond is false
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\begin{align*}
&\{\{ P \}\} \text{ while } (\text{cond}) \ S \ \{\{ Q \}\}
\end{align*}
\]

This triple is valid iff: there is a loop invariant I such that

\[
\begin{align*}
&\{\{ P \}\} \\
&\{\{ \text{Inv: } I \}\} \\
&\text{while } (\text{cond}) \ S \\
&\{\{ Q \}\}
\end{align*}
\]

\[
\begin{align*}
&\text{• P implies I} \\
&\text{• } \{\{ I \text{ and cond}\} \ S \ \{\{ I \}\} \text{ is valid} \\
&\text{• Q holds when } I \text{ holds and } \text{cond is false}
\end{align*}
\]
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \}\} \text{ while (cond) } S \{\{ Q \}\}
\]

This triple is valid iff: there is a loop invariant \( I \) such that

\[
\begin{align*}
\{\{ P \}\} \\
\{\{ \text{Inv: } I \}\} \\
\text{while (cond) } \\
S \\
\{\{ Q \}\}
\end{align*}
\]

- \( P \) implies \( I \)
- \( \{\{ I \text{ and } \text{cond} \}\} S \{\{ I \}\} \) is valid
- \( (I \text{ and not } \text{cond}) \) implies \( Q \)
While-Loop Rule

Consider a while-loop (other loop forms not too different):

\[
\{\{ P \} \} \text{ while (cond) } S \{\{ Q \} \}
\]

This triple is valid iff: there is a loop invariant \( I \) such that

\[
\begin{align*}
\{\{ P \} \} & \quad \text{P implies I} \\
\{\{ \text{Inv: } I \} \} & \quad \{\{ I \text{ and cond} \} \} S \{\{ I \} \} \text{ is valid} \\
\text{while (cond) } S & \quad \{\{ I \text{ and not cond} \} \text{ implies Q} \\
\{\{ Q \} \} &
\end{align*}
\]
More on Loop Invariants

• We need a loop invariant to check validity of a while loop.
• There is no automatic way to generate these.
  – (A theory course will explain why…)

• For this lecture, all loop invariants will be given.
• Next lecture will discuss how to choose a loop invariant.

• Pro Tip: always document your invariants for non-trivial loops
  – as we just saw, much easier for others to check your code
  – possible exception for loops that are “obvious”
• Pro Tip: with a good loop invariant, the code is easy to write
  – we will see this next time
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;
while (i != n) {
    s = s + b[i];
    i = i + 1;
}

{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
int s = 0;
int i = 0;
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
    i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$: 

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```

- $(s = 0$ and $i = 0)$ implies $s = b[0] + \ldots + b[i-1]$?
  
  Yes. (An empty sum is zero.)
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```

• $(s = 0 \text{ and } i = 0)$ implies $I$
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
{{ b.length >= n }}
int s = 0;
int i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
    {{ s = b[0] + ... + b[i-1] and i != n }}
    s = s + b[i];
    i = i + 1;
    {{ s = b[0] + ... + b[i-1] }}
}
{{ s = b[0] + ... + b[n-1] }}
```

- $(s = 0$ and $i = 0$) implies $I$
- ${{ I and i != n }}$ S ${{ I }}$
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;

{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    {{ s = b[0] + \ldots + b[i-1] and i != n }}
    s = s + b[i];
i = i + 1;
    {{ s = b[0] + \ldots + b[i-1] }}
}

{{ s = b[0] + \ldots + b[n-1] }}
```

- $(s = 0$ and $i = 0)$ implies $I$
- ${{ I and i != n }}$ $S$ ${{ I }}$ ?

Yes (e.g., by backward reasoning)

```java
{{ s + b[i] = b[0] + \ldots + b[i] }}
{{ s = b[0] + \ldots + b[i] }}
```
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
    i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```

- (s = 0 and i = 0) implies I
- {{ I and i != n }} S {{ I }}
- {{ I and i == n }} implies s = b[0] + \ldots + b[n-1] ?

Yes. (I is the postcondition when we have i == n.)
Consider the following code to compute $b[0] + \ldots + b[n-1]$:

\[
\begin{align*}
\{ & \text{b.length} \geq n \} \\
& s = 0; \\
& i = 0; \\
\{ & \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
\text{while } (i \neq n) \{ \\
& s = s + b[i]; \\
& i = i + 1; \\
\} \\
\{ & s = b[0] + \ldots + b[n-1] \}
\end{align*}
\]

- $(s = 0 \text{ and } i = 0)$ implies I
- $\{ \text{I and } i \neq n \} \ S \{ \text{I} \}$
- $\{ \text{I and } i == n \} \text{ implies Q}$

These three checks verify that the postcondition holds (i.e., the code is correct).
Termination

• Technically, 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 actually exits

• However, that follows from an analysis of the running time
  – e.g., if the code runs in $O(n^2)$ time, then it terminates
  – an infinite loop would be $O(\infty)$
  – any finite bound on the running time proves it terminates

• It is normal to also analyze the running time of code we write, so we get termination already from that analysis.
Example: sum of array (attempt 2)

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = -1;
while (i != n-1) {
i = i + 1;
s = s + b[i];
}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array (attempt 2)

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = -1;
{{ Inv: s = b[0] + ... + b[i] }}
while (i != n-1) {
    i = i + 1;
    s = s + b[i];
}
{{ s = b[0] + ... + b[n-1] }}
```
Example: sum of array (attempt 2)

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

\[
\begin{aligned}
&\{ \text{b.length }\geq n \} \\
s &= 0; \\
i &= -1; \\
&\{ \text{Inv: } s = b[0] + \ldots + b[i] \} \\
\text{while } (i \neq n-1) \{ \\
&\quad i = i + 1; \\
&\quad s = s + b[i]; \\
\} \\
&\{ s = b[0] + \ldots + b[n-1] \}
\end{aligned}
\]

- (\( s = 0 \) and \( i = -1 \)) implies I as before
- \( \{ I \text{ and } i \neq n-1 \} \) S \( \{ I \} \)
  - reason backward:
    \[
    \begin{aligned}
    &\{ s + b[i+1] = b[0] + \ldots + b[i+1] \} \\
    &\{ s + b[i] = b[0] + \ldots + b[i] \}
    \end{aligned}
    \]
- (I and \( i = n-1 \)) implies Q as before