Floyd Logic

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

\[
\{ P \} \ S \ \{ Q \}
\]

- \( P \) the precondition
- \( S \) the code
- \( Q \) the postcondition

• 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
  - code is correct iff triple is valid
Reasoning Forward & Backward

- **Forward:**
  - start with the *given* precondition
  - fill in the *strongest* postcondition

- **Backward**
  - start with the *required* postcondition
  - fill in the *weakest* precondition

- Finds the “best” assertion that makes the triple valid
Reasoning: Assignments

\[ x = \text{expr} \]

- **Forward**
  - add the fact “\(x = \text{expr}\)” to what is known
  - BUT you must fix any existing references to “\(x\)”

- **Backward**
  - just replace any “\(x\)” in the postcondition with \(\text{expr}\) (substitution)
Correctness by Forward / Backward

Reasoning in either direction gives valid assertions.
Just need to check adjacent assertions:
• top assertion must imply bottom one

\[
\begin{align*}
\{ \{ P \} \} & \quad \{ \{ P \} \} & \quad \{ \{ P \} \} \\
S1 & \quad \{ \{ Q1 \} \} & \quad \{ \{ P1 \} \} \\
S2 & \quad \{ \{ Q \} \} & \quad \{ \{ Q \} \} \\
\end{align*}
\]
Subtleties in Forward Reasoning...

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

\[
\begin{align*}
\{ & \} \\
& w = x + y; \\
& \{ w = x + y \} \\
& x = 4; \\
& \{ w = x + y \text{ and } x = 4 \} \\
& y = 3; \\
& \{ w = x + y \text{ and } x = 4 \text{ and } y = 3 \}
\end{align*}
\]

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 = x₁ + y and x = 4 }}
    y = 3;
{{ w = x₁ + y₁ and x = 4 and y = 3 }}
```
Fix 2 (better)

- Express prior values in terms of the current value

\[
\begin{align*}
  \{ & \} \\
  w &= x + y; \\
  \{ & w = x + y \} \\
  x &= x + 4; \\
  \{ & w = x_1 + y \text{ and } x = x_1 + 4 \} \\
  \Rightarrow & \{ w = x - 4 + y \}
\end{align*}
\]

Now, \( x_1 = x - 4 \)

So \( w = x_1 + y \iff w = x - 4 + y \)

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
If Statements
If Statements

Forward reasoning

```plaintext
{{ 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

\[
\begin{align*}
\{&P\}\ \\
\text{if } (\text{cond}) \Rightarrow \{&P \text{ and } \text{cond}\} \\
&\text{S1} \\
\{&P1\}\ \\
\text{else} \Rightarrow \{&P \text{ and not } \text{cond}\} \\
&\text{S2} \\
\{&P2\}\ \\
&\{P1 \text{ or } P2\}
\end{align*}
\]
If Statements

Backward reasoning

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

Backward reasoning

{{ ? }}
    if (cond)
        S1
    {{ Q }}
else
    S2
    {{ Q }}
If Statements

Backward reasoning

```plaintext
{{ Q }}
if (cond)
  {{ Q1 }}
  S1
  {{ Q }}
else
  {{ Q2 }}
  S2
  {{ Q }}
{{ Q }}
```
If Statements

Backward reasoning

\[
\{
\text{cond and } Q_1 \text{ or }
\]
\[
\text{not cond and } Q_2 
\}
\]

if (cond)

\{
Q_1 
\}

S_1

\{
Q 
\}

else

\{
Q_2 
\}

S_2

\{
Q 
\}

\{
Q 
\}
If-Statement Example

Forward reasoning

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

Forward reasoning

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

CSE 331 Fall 2022
If-Statement Example

Forward reasoning

```plaintext
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

```plaintext
{{ }}
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

```java
{{ }}
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

```plaintext
{ { } }  
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

```plaintext
{{ }}
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

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

Forward reasoning

```plaintext
{} {}
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

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

Forward reasoning

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

Backward reasoning

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

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

Forward reasoning

```plaintext
{{ }}
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

```plaintext
{{ (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

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

Backward reasoning

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

Forward reasoning

\[
\{
\}
\]

\[
\text{if } (x \geq 0)
\]

\[
\{
\}
\]

\[
y = x;
\]

\[
\{
\}
\]

\[
x \geq 0 \land y = x
\]

\[
\text{else}
\]

\[
\{
\}
\]

\[
y < 0
\]

\[
\{
\}
\]

\[
y = -x;
\]

\[
\{
\}
\]

\[
x < 0 \land y = -x
\]

\[
\{
\}
\]

\[
y = |x|
\]

Backward reasoning

\[
\{
\}
\]

\[
\text{if } (x \geq 0)
\]

\[
\{
\}
\]

\[
y = x;
\]

\[
\{
\}
\]

\[
y = |x|
\]

\[
\text{else}
\]

\[
\{
\}
\]

\[
y \leq 0
\]

\[
\{
\}
\]

\[
y = -x;
\]

\[
\{
\}
\]

\[
x \leq 0 \land y = -x
\]

\[
\{
\}
\]

\[
y = |x|
\]

\[
\{
\}
\]

\[
y = |x|
\]
Reasoning So Far

• Mechanical reasoning for assignment and if statements

• All code (essentially) can be written just using:
  – assignments
  – if statements
  – while loops

• Only part we are missing is loops

• (We will also cover function calls later.)
Loops
Reasoning About Loops

• Loop reasoning is not as easy as with “=“ and “if”
  – recall Rice’s Theorem (from 311): checking any non-trivial semantic property about programs is **undecidable**

• We need help (more information) before the reasoning again becomes a mechanical process

• That help comes in the form of a “loop invariant”
Loop Invariant

A **loop invariant** is an assertion that holds at the top of the loop:

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

- 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.
Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant $I$.

Let's try forward reasoning...

\[
\begin{align*}
\{\{P\}\} & \quad S1 \\
\{\{\text{Inv: } I\}\} & \quad \text{while (cond)} \\
& \quad S2 \\
& \quad S3 \\
\{\{Q\}\} & 
\end{align*}
\]
Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant $I$.

Let’s try forward reasoning...

```
{{ P }}
S1
{{ P1 }}
{{ Inv: I }}
while (cond)
  S2
S3
{{ Q }}
```

Need to check that $P1$ implies $I$ (i.e., that $I$ is true the first time)
Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant $I$.

Let’s try forward reasoning...

\[
\begin{align*}
&\{P\}\quad S1 \\
&\{Inv: I\}\quad \text{while (cond)} \\
&\quad \{I \text{ and cond}\}\quad S2 \\
&\quad \{P2\}\quad S3 \\
&\{Q\}\quad \}
\]

Need to check that $P2$ implies $I$ again (i.e., that $I$ is true each time around)
Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant $I$.

Let's try forward reasoning...

$$\begin{align*}
\{ P \} & \quad S_1 \\
\{ \text{Inv: } I \} & \quad \text{while (cond)} \\
\{ I \text{ and not cond} \} & \quad S_3 \\
\{ P_3 \} & \quad \{ Q \} \\
\end{align*}$$

Need to check that $P_3$ implies $Q$ (i.e., $Q$ holds after the loop)
Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant $I$.

$$
\begin{align*}
\{ \mathbf{P} \} & \mathbb{S} 1 \\
\{ \text{Inv: } I \} & \mathbb{S} 2 \\
\text{while (cond)} & \mathbb{S} 2 \\
\mathbb{S} 3 & \\
\{ \mathbf{Q} \} &
\end{align*}
$$

Informally, we need:

- $I$ holds initially
- $I$ holds each time around
- $Q$ holds after we exit

Formally, we need validity of:

- $\{ \mathbf{P} \} S1 \{ I \}$
- $\{ I \text{ and cond } \} S2 \{ I \}$
- $\{ I \text{ and not cond } \} S3 \{ Q \}$

(can check these with backward reasoning instead)
More on Loop Invariants

• Loop invariants are crucial information
  – needs to be provided before reasoning is mechanical

• Pro Tip: always document your invariants for non-trivial loops
  – don’t make code reviewers guess the invariant

• Pro Tip: with a good loop invariant, the code is easy to write
  – all the creativity can be saved for finding the invariant
  – more on this in later lectures…
Consider the following code to compute $b[0] + ... + b[n-1]$:

```c
{{ }}
s = 0;
i = 0;
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Equivalent to this “for” loop:

```c
s = 0;
for (int i = 0; i != n; i++)
    s = s + b[i];
```
Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

```c
{{ }}
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] }}
```
Example: sum of array

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

```c
s = 0;
i = 0;

while (i != n) {
    s = s + b[i];
    i = i + 1;
}
```

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

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

```c
{{ }}
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 $s = b[0] + \ldots + b[i-1]$?

Less formal

$s = \text{sum of first } i \text{ numbers in } b$
Example: sum of array

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

```c
{{ }}
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 $s = b[0] + \ldots + b[i-1]$?

Less formal

$s = \text{sum of first } i \text{ numbers in } b$

When $i = 0$, $s$ needs to be the sum of the first 0 numbers, so we need $s = 0$. 
Example: sum of array

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

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

• (s = 0 and i = 0) implies $s = b[0] + ... + b[i-1]$?

More formal

\[
s = \text{sum of all } b[k] \text{ with } 0 \leq k \leq i-1
\]
Example: sum of array

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

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

• $(s = 0$ and $i = 0$) implies $s = b[0] + ... + b[i-1]$?

More formal:

$s = \text{sum of all } b[k] \text{ with } 0 \leq k \leq i-1$

- $i = 3$ $(0 \leq k \leq 2)$: $s = b[0] + b[1] + b[2]$
- $i = 2$ $(0 \leq k \leq 1)$: $s = b[0] + b[1]$
- $i = 1$ $(0 \leq k \leq 0)$: $s = b[0]$
- $i = 0$ $(0 \leq k \leq -1)$: $s = 0$
Example: sum of array

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

```c
{{ }}
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 $s = b[0] + \ldots + b[i-1]$?

More formal

$s = \text{sum of all } b[k] \text{ with } 0 \leq k \leq i-1$

when $i = 0$, we want to sum over all indexes $k$ satisfying $0 \leq k \leq -1$

There are no such indexes, so we need $s = 0$
**Example: sum of array**

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

\[
\begin{align*}
&s = 0; \\
&i = 0; \\
\{& s = 0 \text{ and } i = 0 \}
\end{align*}
\]

\[
\begin{align*}
&\text{while } (i != n) \{ \\
&\quad s = s + b[i]; \\
&\quad i = i + 1; \\
&\}
\end{align*}
\]

\[
\begin{align*}
\{& s = b[0] + \ldots + b[n-1] \}
\end{align*}
\]

- \((s = 0 \text{ and } i = 0)\) implies \(s = b[0] + \ldots + b[i-1]\)?

Yes. (An empty sum is zero.)
Example: sum of array

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

```c
{{ }}
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 $b[0] + \ldots + b[n-1]$:

```c
{{ }}
int s = 0;
int 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 $I$
- $\{{ I \text{ and } i != n \}} \implies \{{ I \}}$

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

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

```c
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 $I$
- $\{\{ I \text{ and } i != n \}\} \implies \{\{ I \}\}$

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

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

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

\[
\begin{align*}
\{ & \text{s = b[0] + \ldots + b[i-1] and not (i \neq n)} \} \\
\{ & \text{s = b[0] + \ldots + b[n-1]} \}
\end{align*}
\]

- \((s = 0 \text{ and } i = 0) \text{ implies } \mathbf{I}\)
- \(\{ \text{ I and i != n } \} \ s \{ \text{ I } \} \)
- \(\{ \text{ I and not (i != n) } \} \text{ implies } s = b[0] + \ldots + b[n-1] ? \)
Example: sum of array

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

```c
{ 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] }
```

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

These three checks verify that the outermost triple is valid (i.e., that the code is correct).