Hoare 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)
Reasoning: If Statements

Forward reasoning

\[
\begin{align*}
\{P\} \\
\text{if (cond)} \\
\{P \text{ and } \text{cond}\} \\
S_1 \\
\{P_1\} \\
\text{else} \\
\{P \text{ and } \neg \text{cond}\} \\
S_2 \\
\{P_2\} \\
\{P_1 \text{ or } P_2\}
\end{align*}
\]

Backward reasoning

\[
\begin{align*}
\{\text{cond and } Q_1 \text{ or} & \neg \text{cond and } Q_2\} \\
\text{if (cond)} \\
\{Q_1\} \\
S_1 \\
\{Q\} \\
\text{else} \\
\{Q_2\} \\
S_2 \\
\{Q\}
\end{align*}
\]
Validity with Fwd & Back Reasoning

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

\[
\begin{align*}
\{ \{ P \} \} & \quad \{ \{ P \} \} \\
S1 & \quad S1 \\
\{ \{ P1 \} \} & \quad \{ \{ Q1 \} \} \\
\{ \{ Q \} \} & \quad \{ \{ Q \} \} \\
\end{align*}
\]
Reasoning So Far

• “Turn the crank” 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
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 turn-the-crank 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:

\[
\{\{ \text{Inv: } I \}\}
\]

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...

```
{{ P }}
S1

{{ Inv: I }}
while (cond)
  S2
S3
{{ Q }}
```
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...

\[
\begin{align*}
\{ \{ P \} \} & \quad S1 \\
\{ \{ P1 \} \} & \quad S2 \\
\{ \{ \text{Inv: } I \} \} & \quad \text{while } (\text{cond}) \\
\{ \{ Q \} \} & \quad S3
\end{align*}
\]

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...

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

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...

\[
\begin{align*}
\{\{ P \}\} & \quad S1 \\
\{\{ \text{Inv: } I \}\} & \quad \text{while (cond)} \\
& \quad S2 \\
\{\{ I \text{ and not cond} \}\} & \quad S3 \\
\{\{ P3 \}\} & \quad \{\{ Q \}\} \\
\end{align*}
\]

Need to check that $P3$ 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 $I$.

$$
\begin{align*}
\{ \text{P} \} \\
S1 \\
\{ \text{Inv: I} \} \\
\text{while (cond)} \\
S2 \\
S3 \\
\{ \text{Q} \}
\end{align*}
$$

Informally, we need:
- $I$ holds initially
- $I$ holds each time around
- $Q$ holds after we exit

Formally, we need validity of:
- $\{ \text{P} \} S1 \{ I \}$
- $\{ I \text{ and } \text{cond} \} S2 \{ I \}$
- $\{ I \text{ and } \text{not cond} \} S3 \{ Q \}$

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

• Loop invariants are crucial information
  – needs to be provided before reasoning is “turn the crank”

• 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…
Example: sum of array

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

```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];
```
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] }}
```
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] \}}
```
Example: sum of array

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

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

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

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

\[
\begin{align*}
\{\} & \quad (s = 0 \text{ and } i = 0) \text{ implies } I \\
\{ s = 0; \} \\
\{ i = 0; \} \\
\{ s = 0 \text{ and } i = 0 \} \\
\{ \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
\text{while } (i != n) \{ \\
\quad s = s + b[i]; \\
\quad i = i + 1; \\
\} \\
\{ s = b[0] + \ldots + b[n-1] \}
\end{align*}
\]
Example: sum of array

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

```plaintext
{ } 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 $i != n \} s \{ I \}$?
Example: sum of array

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

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

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

:s = b[0] + ... + b[n-1]}
```

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

- $(s + b[i] = b[0] + ... + b[i])$
  $\{\{ s = b[0] + ... + b[i] \}\}$

- $(s = b[0] + ... + b[n-1])$
Example: sum of array

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

```c
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
    s = s + b[i];
    i = i + 1;
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

• (s = 0 and i = 0) implies I
• {{ I and i != n }} S {{ I }}
• {{ I and not (i != n) }} implies $s = b[0] + ... + b[n-1]$ ?
Example: sum of array

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

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

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

These three checks verify 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(n²) time, then it terminates
  – an infinite loop would be O(infinity)
  – 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.
Reasoning So Far

• Forward and backward reasoning for...
  – assignments
  – if statements
  – while loops

• (essentially) all code can be rewritten to use just these
Example HW problem

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

```c
{{ }}
s = 0;
{{ ____________ }}
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] }}
```

CSE 331 Spring 2021
Example HW problem

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

```c
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and 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];
    {{ s = b[0] + \ldots + b[i-1] + b[i] and i != n }}
    i = i + 1;
    {{ s = b[0] + \ldots + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + \ldots + b[i-1] and not (i != n) }}
{{ s = b[0] + \ldots + b[n-1] }}
Are we done?
```
Warning: not just filling in blanks

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

```c
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and 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];
    {{ s = b[0] + \ldots + b[i-1] + b[i] and i != n }}
    i = i + 1;
    {{ s = b[0] + \ldots + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + \ldots + b[i-1] and not (i != n) }}
{{ s = b[0] + \ldots + b[n-1] }}
```

Are we done?  
No, need to also check...

Does invariant hold initially?
Warning: not just filling in blanks

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

```c
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and 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];
    {{ s = b[0] + \ldots + b[i-1] + b[i] and i != n }}
i = i + 1;
    {{ s = b[0] + \ldots + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + \ldots + b[i-1] and not (i != n) }}
{{ s = b[0] + \ldots + b[n-1] }}
```

Are we done?  No, need to also check...

Holds initially?  Yes: $i = 0$ implies $s = b[0] + \ldots + b[-1] = 0$

- $i = 2$: $s = b[0] + b[1]$
- $i = 1$: $s = b[0]$
- $i = 0$: $s = 0$
Warning: not just filling in blanks

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

```c
{{
    s = 0;
    {{ s = 0 }}
    i = 0;
    {{ s = 0 and 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];
        {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
        i = i + 1;
        {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
    }
    {{ s = b[0] + ... + b[i-1] and not (i != n) }}
    {{ s = b[0] + ... + b[n-1] }}
```  

Are we done? No, need to also check...  

Does postcondition hold on termination?
The following code to compute $b[0] + ... + b[n-1]$:

```c
{{
    s = 0;
    {{ s = 0 }}
    i = 0;
    {{ s = 0 and 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];
        {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
        i = i + 1;
        {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
    }
    {{ s = b[0] + ... + b[i-1] and not (i != n) }}
    Postcondition holds? Yes, since i = n.
    {{ s = b[0] + ... + b[n-1] }}
}}
```
Warning: not just filling in blanks

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

```c
{{
  s = 0;
  {{ s = 0 }}
  i = 0;
  {{ s = 0 and 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];
    {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
    i = i + 1;
    {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
  }
  {{ s = b[0] + ... + b[i-1] and not (i != n) }}
  {{ s = b[0] + ... + b[n-1] }}
}}
```

Are we done? No, need to also check...

Does loop body preserve invariant? [Yes. Weaken by dropping “i-1 != n”]

CSE 331 Spring 2021
Warning: not just filling in blanks

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

```c
{{
    s = 0;
    {{ s = 0 }}
}}
i = 0;
{{ s = 0 and 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];
    {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
    i = i + 1;
    {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?
No, need to also check...

Does loop body preserve invariant?

```c
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1;
{{ s = b[0] + ... + b[i-1] }}
```
The following code to compute $b[0] + ... + b[n-1]$:

```c
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
    // Does loop body preserve invariant?
    {{ s = b[0] + ... + b[i-1] and i != n }}
    s = s + b[i];
    {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
    i = i + 1;
    {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

Are we done?
No, need to also check...

Does loop body preserve invariant?

Yes. If Inv holds, then so does this (just add b[i] to both sides of Inv)

CSE 331 Spring 2021
Example: sum of array (attempt 2)

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

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

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

- Changed from \( i = 0 \)
- Reordered
- Changed from \( n \)
Example: sum of array (attempt 2)

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

```plaintext
s = 0;
i = -1;

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

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

Proof:

1. $s = 0$;
2. $i = -1$;
3. Inv: $s = b[0] + ... + b[i]$;
4. while $(i != n-1)$ {
   4.1. $i = i + 1$;
   4.2. $s = s + b[i]$;
   5. $s = b[0] + ... + b[i]$
5. $s + b[i+1] = b[0] + ... + b[i+1]$
6. $s + b[i] = b[0] + ... + b[i]$
7. $s = b[0] + ... + b[i]$
8. $s = b[0] + ... + b[n-1]$
Example: sum of array (attempt 2)

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

\[
\begin{align*}
\{ \} & \quad \{ I \} \quad \{ s = 0 \text{ and } i = -1 \}\} \implies I \\
\{ s = 0; \} & \quad \{ I \} \quad \{ i = -1; \} \\
\{ \text{Inv: } s = b[0] + \ldots + b[i] \} & \quad \{ I \text{ and } i \neq n-1 \}\} \implies I \\
\text{while } (i \neq n-1) \{ & \quad \{ s + b[i+1] = b[0] + \ldots + b[i+1] \}\} \\
\quad i = i + 1; & \quad \{ s + b[i] = b[0] + \ldots + b[i] \}\} \\
\quad s = s + b[i]; & \quad \{ I \text{ and } i = n-1 \}\} \implies Q \\
\{ s = b[0] + \ldots + b[n-1] \} & \quad \{ I \text{ and } i = n-1 \}\} \implies Q
\end{align*}
\]
Example: sum of array (attempt 3)

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

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

Suppose we miss-order the assignments to $i$ and $s$...

Where does the correctness check fail?
Example: sum of array (attempt 3)

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

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

Suppose we miss-order the assignments to $i$ and $s$...

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

We can spot this bug because the invariant does not hold:

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

First assertion is not Inv.