CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2022
Lecture 3 – Reasoning about Loops
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

  \[
  \{ P \} \quad S \quad \{ ? \}
  \]

• Backward
  – start with the **required** postcondition
  – fill in the **weakest** precondition

  \[
  \{ ? \} \quad S \quad \{ Q \}
  \]

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

\[ x = \text{expr} \]

- **Forward**
  - add the fact “x = expr” to what is known
  - BUT you must *fix* any existing references to “x”

- **Backward**
  - just replace any “x” in the postcondition with expr (substitution)
Reasoning: If Statements

Forward reasoning

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

Backward reasoning

\[
\begin{align*}
\{ \text{cond and Q1 or not cond and Q2} \} \\
\text{if (cond)} \\
\{ Q1 \} \\
S1 \\
\{ Q \} \\
\text{else} \\
\{ Q2 \} \\
S2 \\
\{ Q \} \\
\{ Q \} \\
\end{align*}
\]
Validity with Fwd & Back Reasoning

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

\[
\begin{align*}
&\{\{ P \}\}\ S1\ \ S2\ \ \{\{ P1 \}\}\ \{\{ Q \}\}\ \\
&\{\{ P \}\}\ S1\ \ S2\ \ \{\{ Q1 \}\}\ \{\{ Q \}\}\ \\
&\{\{ P \}\}\ S1\ \ S2\ \ \{\{ P1 \}\}\ \{\{ Q1 \}\}\ \{\{ Q \}\}\ \\
&\{\{ P \}\}\ S1\ \ S2\ \ \{\{ Q \}\}\ \\
\end{align*}
\]
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

• (We will also cover function calls later.)
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”
A **loop invariant** is an assertion that holds at the top of the loop:

```plaintext
{{ 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...

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

\[
\begin{align*}
\{\{ P \}\} & \quad S_1 \\
\{\{ P_1 \}\} & \quad \{\{ \text{Inv: } I \}\} \\
\{\{ \text{while (cond)} \} & \quad S_2 \\
S_3 & \quad \{\{ Q \}\}
\end{align*}
\]

Need to check that \( P_1 \) 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...

\[
\begin{align*}
\{ \{ P \}\} & \quad S1 \\
\{ \{ \text{Inv: } I \}\} & \quad \text{while (cond)} \\
\{ \{ I \text{ and cond} \}\} & \quad S2 \\
\{ \{ P2 \}\} & \quad S3 \\
\{ \{ Q \}\} & \quad \text{(i.e., that } I \text{ is true each time around)}
\end{align*}
\]
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...

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

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 $I$.

$\{ P \} \quad S1$

$\{ Inv: I \} \quad while \ (cond) \quad S2$

$\{ Q \}$

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

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

(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…
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;
  }
  {{ 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 $b[0] + \ldots + b[n-1]$:

```c
{{ }}

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

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + \ldots + b[i-1] }}

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

{{ 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 and i = 0 }}
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 $s = b[0] + ... + b[i-1]$?

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;
{{ s = 0 and i = 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 $b[0] + \ldots + b[n-1]$: 

```java
{{ }}
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

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

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

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

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] }}
```

- $(s = 0 \text{ and } i = 0)$ 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 $b[0] + \ldots + b[n-1]$:

```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] }}
```

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

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

$$s = 0;$$
$$i = 0;$$

$s = 0$ and $i = 0$ implies $I$

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 $b[0] + ... + b[n-1]$:

```plaintext
{{ }}
s = 0;
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] }}
```
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 $i != n \}\} S \{\{ I \}\}$

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

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

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

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

These three checks verify that the outermost triple is valid (i.e., that 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 HW problem

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 HW problem

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

{
    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[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 $b[0] + ... + b[n-1]$:

```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] }}
    s = s + b[i];
    {{ s = b[0] + ... + b[i] }}
    i = i + 1;
    {{ s = b[0] + ... + b[i-1] }}
}
{{ 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 invariant hold initially?
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] }}
    s = s + b[i];
    {{ s = b[0] + \ldots + b[i] }}
    i = i + 1;
    {{ s = b[0] + \ldots + b[i-1] }}
}
{{ 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 loop body preserve invariant?
The following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ }}
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-1] + 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 postcondition hold on termination?
Warning: not just filling in blanks

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

```plaintext
{{
  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...
```

HW has “?”s at these three places to indicate a triple that requires explanation
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
Example: sum of array (attempt 2)

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

```plaintext
{{{}}
  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 $b[0] + ... + b[n-1]$: 

```plaintext
{{ }}
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] }}
```

Work as before:

- $(s = 0$ and $i = -1)$ implies $\mathcal{I}$
  - $\mathcal{I}$ holds initially

- $(\mathcal{I}$ and $i = n-1)$ implies $\mathcal{Q}$
  - $\mathcal{I}$ implies $\mathcal{Q}$ at exit
Example: sum of array (attempt 2)

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

```plaintext
{{ }}

s = 0;
i = -1;
{{ Inv: s = b[0] + ... + b[i] }}

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

- (s = 0 and i = -1) implies $I$
  - as before

- $\{\{ I \text{ and } i != n-1 \} \} \subseteq \{\{ I \} \}$
  - reason backward

- (I and i = n-1) implies Q
  - as before
Example: sum of array (attempt 3)

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) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + ... + 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 $b[0] + \ldots + b[n-1]$:

```
{{ }}
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 $s$...

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

```
{{ 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.
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 $s$…

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

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

For example, if $i = 2$, then

- $s = b[0] + b[1] + b[2]$
- $s = b[0] + b[1] + b[3]$