Announcements

• Discussion board: be sure to post a reply to the welcome message

• Next few lectures: two presentations on the web:
  – Powerpoint slides
  – Lecture notes

• They are complementary and you should understand both of them

• HW1 out now. Programming logic with no loops. Due in a week.
Reasoning about code

Determine what facts are true as a program executes
  – Under what assumptions

Examples:
  – If $x$ starts positive, then $y$ is 0 when the loop finishes
  – Contents of the array that $arr$ refers to are sorted
  – Except at one code point, $x + y == z$
  – For all instances of Node $n$,
    $n.next == null ∨ n.next.prev == n$
  – ...

Why do this?

- Essential complement to testing, which we will also study
  - Testing: Actual results for some actual inputs
  - Logical reasoning: Reason about whole classes of inputs/ states at once (“If \( x > 0 \), …”)
    - Prove a program correct (or find bugs trying)
    - Understand why code is correct

- Stating assumptions is the essence of specification
  - “Callers must not pass \texttt{null} as an argument”
  - “Callee will always return an unaliased object”
  - …
Our approach

• Hoare Logic: a 1970s approach to logical reasoning about code
  – For now, consider just variables, assignments, if-statements, while-loops
    • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Why?

• Programmers rarely “use Hoare logic” in this much detail
  – For simple snippets of code, it’s overkill
  – Gets very complicated with objects and aliasing
  – But can be very useful to develop and reason about loops and data with subtle invariants
    • Examples: Homework 0, Homework 2

• Also it’s an ideal setting for the right logical foundations
  – How can logic “talk about” program states?
  – How does code execution “change what is true”?
  – What do “weaker” and “stronger” mean?

This is all essential for specifying library-interfaces, which does happen All the Time in The Real World® (coming lectures)
Example

Forward reasoning:

- Suppose we initially know (or assume) $w > 0$

  ```
  // w > 0
  x = 17;
  // w > 0 ∧ x == 17
  y = 42;
  // w > 0 ∧ x == 17 ∧ y == 42
  z = w + x + y;
  // w > 0 ∧ x == 17 ∧ y == 42 ∧ z > 59
  ...
  ```

- Then we know various things after, including $z > 59$
Example

Backward reasoning:

– Suppose we want \( z \) to be negative at the end
  
  ```
  // w + 17 + 42 < 0
  x = 17;
  // w + x + 42 < 0
  y = 42;
  // w + x + y < 0
  z = w + x + y;
  // z < 0
  ```

– Then we know initially we need to know/assume \( w < -59 \)
  
  • Necessary and sufficient
Forward vs. Backward, Part 1

• Forward reasoning:
  – Determine what follows from initial assumptions
  – Most useful for *maintaining an invariant*

• Backward reasoning
  – Determine sufficient conditions for a certain result
    • If result desired, the assumptions suffice for correctness
    • If result undesired, the assumptions suffice to trigger bug
Forward vs. Backward, Part 2

• Forward reasoning:
  – Simulates the code (for many “inputs” “at once”)
  – Often more intuitive
  – But introduces [many] facts irrelevant to a goal

• Backward reasoning
  – Often more useful: Understand what each part of the code contributes toward the goal
  – “Thinking backwards” takes practice but gives you a powerful new way to reason about programs
// initial assumptions
if(...) {
    ...
    // also know test evaluated to true
} else {
    ...
    // also know test evaluated to false
}
// either branch could have executed

Two key ideas:

1. The precondition for each branch includes information about the result of the test-expression

2. The overall postcondition is the disjunction (“or”) of the postcondition of the branches
Example (Forward)

Assume initially $x \geq 0$

```plaintext
// x >= 0
z = 0;
// x >= 0 ∧ z == 0
if(x != 0) {
    // x >= 0 ∧ z == 0 ∧ x != 0 (so x > 0)
    z = x;
    // ... ∧ z > 0
} else {
    // x >= 0 ∧ z == 0 ∧ !(x!=0) (so x == 0)
    z = x + 1;
    // ... ∧ z == 1
}
// ( ... ∧ z > 0) ∨ (... ∧ z == 1) (so z > 0)
```
Our approach

• Hoare Logic, a 1970s approach to logical reasoning about code
  – [Named after its inventor, Tony Hoare]
  – Considering just variables, assignments, if-statements, while-loops
    • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Some notation and terminology

• The “assumption” before some code is the **precondition**
• The “what holds after (given assumption)” is the **postcondition**

• Instead of writing pre/postconditions after //, write them in {...}
  – This is not Java
  – How Hoare logic has been written “on paper” for 40ish years
    
    ```
    { w < -59 }
    x = 17;
    { w + x < -42 }
    ```
  – In pre/postconditions, = is equality, not assignment
    • Math’s “=”, which for numbers is Java’s ==
      
      ```
      { w > 0 ∧ x = 17 }
      y = 42;
      { w > 0 ∧ x = 17 ∧ y = 42 }
      ```
What an assertion means

• An *assertion* (pre/postcondition) is a logical formula that can refer to program state (e.g., contents of variables)

• A *program state* is something that “given” a variable can “tell you” its contents
  – Or any expression that has no *side-effects*

• An assertion *holds* for a program state, if evaluating using the program state produces *true*
  – Evaluating a program variable produces its contents in the state
  – Can think of an assertion as representing the *set* of (exactly the) states for which it holds
A Hoare Triple

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

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

– \( P \) the precondition
– \( S \) the code (statement)
– \( Q \) the postcondition

• A Hoare triple \( \{ P \} \ S \{ Q \} \) is (by definition) valid if:
  – For all states for which \( P \) holds, executing \( S \) always produces a state for which \( Q \) holds
  – Less formally: If \( P \) is true before \( S \), then \( Q \) must be true after
  – Else the Hoare triple is invalid
Examples

Valid or invalid?

- (Assume all variables are integers without overflow)

• \{x \neq 0\} y = x*x; \{y > 0\}
• \{z \neq 1\} y = z*z; \{y \neq z\}
• \{x \geq 0\} y = 2*x; \{y > x\}
• \{true\} (if(x > 7) \{y=4;\} else \{y=3;\}) \{y < 5\}
• \{true\} (x = y; z = x;) \{y=z\}
• \{x=7 \land y=5\}
  (tmp=x; x=tmp; y=x;)
  \{y=7 \land x=5\}
Examples

Valid or invalid?
   - (Assume all variables are integers without overflow)

• \{x \neq 0\} \ y = x*x; \ {y > 0} \ \text{valid}
• \{z \neq 1\} \ y = z*z; \ {y \neq z} \ \text{invalid}
• \{x \geq 0\} \ y = 2*x; \ {y > x} \ \text{invalid}
• \{\text{true}\} \ (\text{if}(x > 7) \ {y=4;} \ \text{else} \ {y=3;}) \ {y < 5} \ \text{valid}
• \{\text{true}\} \ (x = y; \ z = x;) \ {y=z} \ \text{valid}
• \{x=7 \ \land \ y=5\} \ \text{invalid}
    \ (\text{tmp}=x; \ x=\text{tmp}; \ y=x;)
    \ {y=7 \ \land \ x=5\}
Aside: assert in Java

• An assertion in Java is a statement with a Java expression, e.g.,
  \[\text{assert } x > 0 \land y < x;\]
• Similar to our assertions
  – Evaluate using a program state to get true or false
  – Uses Java syntax

• In Java, this is a run-time thing: Run the code and raise an exception if assertion is violated
  – Unless assertion-checking is disabled
  – Later course topic

• This week: we are reasoning about the code, not running it on some input
The general rules

- So far: Decided if a Hoare triple was valid by using our understanding of programming constructs

- Now: For each kind of construct there is a general rule
  - A rule for assignment statements
  - A rule for two statements in sequence
  - A rule for conditionals
  - [next lecture:] A rule for loops
  - ...
Basic rule: Assignment

\[ \{P\} \ x = e; \ \{Q\} \]

- Let \( Q' \) be like \( Q \) except replace every \( x \) with \( e \)
- Triple is valid if:
  For all program states, if \( P \) holds, then \( Q' \) holds
  - That is, \( P \) implies \( Q' \), written \( P \Rightarrow Q' \)

- Example: \( \{z > 34\} \ y = z + 1; \ \{y > 1\} \)
  - \( Q' \) is \( \{z+1 > 1\} \)
Combining rule: Sequence

\{P\} S_1; S_2 \{Q\}

- Triple is valid if and only if there is an assertion \( R \) such that
  - \( \{P\}S_1\{R\} \) is valid, and
  - \( \{R\}S_2\{Q\} \) is valid

- Example: \( \{z \geq 1\} y = z + 1; w = y \times y; \{w > y\} \) (integers)
  - Let \( R \) be \( \{y > 1\} \)
  - Show \( \{z \geq 1\} y = z + 1; \{y > 1\} \)
    - Use rule for assignments: \( z \geq 1 \) implies \( z + 1 > 1 \)
  - Show \( \{y > 1\} w = y \times y; \{w > y\} \)
    - Use rule for assignments: \( y > 1 \) implies \( y \times y > y \)
Combining rule: Conditional

\{P\} \text{if}(b) \ S1 \text{ else } S2 \ \{Q\}

- Triple is valid if and only if there are assertions $Q_1, Q_2$ such that
  - $\{P \land b\} S1\{Q_1\}$ is valid, and
  - $\{P \land \neg b\} S2\{Q_2\}$ is valid, and
  - $Q_1 \lor Q_2$ implies $Q$

- Example: $\{true\}$ $\text{(if}(x > 7) \ y=x; \text{ else } y=20;\}$ $\{y > 5\}$
  - Let $Q_1$ be $\{y > 7\}$ (other choices work too)
  - Let $Q_2$ be $\{y = 20\}$ (other choices work too)
  - Use assignment rule to show $\{true \land x > 7\} y=x; \{y>7\}$
  - Use assignment rule to show $\{true \land x \leq 7\} y=20; \{y=20\}$
  - Indicate $y>7 \lor y=20$ implies $y>5$
Our approach

• Hoare Logic, a 1970s approach to logical reasoning about code
  – Considering just variables, assignments, if-statements, while-loops
    • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Weaker vs. Stronger

If P1 implies P2 (written P1 => P2), then:

– P1 is stronger than P2
– P2 is weaker than P1

• Whenever P1 holds, P2 also holds
• So it is more (or at least as) “difficult” to satisfy P1
  – The program states where P1 holds are a subset of the program states where P2 holds
• So P1 puts more constraints on program states
• So it’s a stronger set of obligations/requirements
Examples

• \( x = 17 \) is stronger than \( x > 0 \)

• \( x \) is prime is neither stronger nor weaker than \( x \) is odd

• \( x \) is prime and \( x > 2 \) is stronger than
  \( x \) is odd and \( x > 2 \)

• ...

Why this matters to us

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

• Then: \( \{P_1\} S \{Q\} \) and \( \{P\} S \{Q_1\} \) and \( \{P_1\} S \{Q_1\} \)

• Example:
  – \( P \) is \( x \geq 0 \)
  – \( P_1 \) is \( x > 0 \)
  – \( S \) is \( y = x+1 \)
  – \( Q \) is \( y > 0 \)
  – \( Q_1 \) is \( y \geq 0 \)
So...

• For backward reasoning, if we want $\{P\} S \{Q\}$, we could instead:
  - Show $\{P1\} S \{Q\}$, and
  - Show $P \Rightarrow P1$

• Better, we could just show $\{P2\} S \{Q\}$ where $P2$ is the **weakest precondition** of $Q$ for $S$
  - Weakest means the most lenient assumptions such that $Q$ will hold after executing $S$
  - Any precondition $P$ such that $\{P\} S \{Q\}$ is valid will be stronger than $P2$, i.e., $P \Rightarrow P2$

• Amazing (?): Without loops/methods, for any $S$ and $Q$, there exists a unique weakest precondition, written $wp(S,Q)$
  - Like our general rules with backward reasoning
Weakest preconditions

• $\text{wp}(x = e; , Q)$ is $Q$ with each $x$ replaced by $e$
  – Example: $\text{wp}(x = y*y; , x > 4) = y*y > 4$, i.e., $|y| > 2$

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

• $\text{wp}(\text{if } b \text{ S1 else S2, Q})$ is this logic formula:
  \[(b \land \text{wp}(S1,Q)) \lor (\lnot b \land \text{wp}(S2,Q))\]
  – (In any state, $b$ will evaluate to either true or false…)
  – (You can sometimes then simplify the result)
Simple examples

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

- If $S$ is $y = x + 1; z = y - 3; \text{and } Q$ 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

\[ S \text{ is } \begin{cases} \text{if } (x < 5) \{ \\ x = x \times x; \\ \} \text{ else } \{ \\ x = x+1; \\ \} \end{cases} \]

\[ Q \text{ is } x \geq 9 \]

\[ \text{wp}(S, x \geq 9) \]
\[ = (x < 5 \land \text{wp}(x = x \times x; , x \geq 9)) \]
\[ \lor (x \geq 5 \land \text{wp}(x = x+1; , x \geq 9)) \]
\[ = (x < 5 \land x \times x \geq 9) \]
\[ \lor (x \geq 5 \land x+1 \geq 9) \]
\[ = (x \leq -3) \lor (x \geq 3 \land x < 5) \]
\[ \lor (x \geq 8) \]
## If-statements review

### Forward reasoning

<table>
<thead>
<tr>
<th>Condition</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>{P}</td>
<td></td>
</tr>
<tr>
<td>if B</td>
<td></td>
</tr>
<tr>
<td>{P ∧ B}</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>{Q1}</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>{P ∧ !B}</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>{Q2}</td>
<td></td>
</tr>
<tr>
<td>{Q1 ∨ Q2}</td>
<td></td>
</tr>
</tbody>
</table>

### Backward reasoning

<table>
<thead>
<tr>
<th>Condition</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (B ∧ wp(S1, Q)) ∨ (!B ∧ wp(S2, Q)) }</td>
<td></td>
</tr>
<tr>
<td>if B</td>
<td></td>
</tr>
<tr>
<td>{wp(S1, Q)}</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
</tr>
<tr>
<td>{Q}</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>{wp(S2, Q)}</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>{Q}</td>
<td></td>
</tr>
<tr>
<td>{Q}</td>
<td></td>
</tr>
</tbody>
</table>
“Correct”

- If \( \text{wp}(S, Q) \) is \text{true}, then executing \( S \) will always produce a state where \( Q \) holds
  - \text{true} holds for every program state
One more issue

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

• Example:
  
  \{\textbf{true}\}
  
  \begin{align*}
  w & = x + y; \\
  \{w = x + y;\} \& \\
  x & = 4; \\
  \{w = x + y \land x = 4\} \& \\
  y & = 3; \\
  \{w = x + y \land x = 4 \land y = 3\}
  \end{align*}

  But clearly we do not know \(w=7\)!
The fix

• When you assign to a variable, you need to replace all other uses of the variable in the post-condition with a different variable
  – So you refer to the “old contents”

• Corrected example:

  `{true}`
  `w=x+y;`
  `{w = x + y;}`
  `x=4;`
  `{w = x1 + y ∧ x = 4}`
  `y=3;`
  `{w = x1 + y1 ∧ x = 4 ∧ y = 3}`
Useful example: swap

- Swap contents
  - Give a name to initial contents so we can refer to them in the post-condition
  - Just in the formulas: these “names” are not in the program
  - Use these extra variables to avoid “forgetting” “connections”

\[
\{ x = x_{pre} \land y = y_{pre} \}\ 
\]
\[
tmp = x; \\
\{ x = x_{pre} \land y = y_{pre} \land \ tmp=x \}\ 
\]
\[
x = y; \\
\{ x = y \land y = y_{pre} \land \ tmp=x_{pre} \}\ 
\]
\[
y = tmp; \\
\{ x = y_{pre} \land y = \ tmp \land \ tmp=x_{pre} \}\ 
\]