CSE 331
Software Design & Implementation

Topic: Introduction

Discussion: What are you excited for this summer?
Reminders

• Read the welcome email
• Check your access to Ed, Gradescope, and Canvas
• Should see email about Gitlab repositories soon

Upcoming Deadlines

• Syllabus Quiz due Thursday (6/23)
• HW1 due Thursday (6/23)
Last Time...

- Welcome email
- Syllabus Overview

Today’s Agenda

- Upcoming Assignments
- Motivation
- Reasoning
Upcoming Assignments
Syllabus Quiz

• Due on Thursday night
  – read the syllabus in depth
  – answer a few multiple choice/select questions
  – infinite attempts before deadline

• Why?
  – had a lot of confusion in past quarters
  – make student requests manageable for course staff
HW1

• Due on Thursday night
  – practice interview question
  – write an algorithm to rearrange array elements as described
  – argue in concise, convincing English that it is correct
    • don’t just explain what the code does!
  – do not run your code! (pretend it’s on a whiteboard)
    • know that is correct without running it (a necessary skill)

• This is expected to be difficult (esp. the “argue” part)
  – graded on effort, not correctness
  – do not spend more than 90 minutes on it
  – want you to see that it is tricky... without the tools coming next
Motivation
What are the goals of CSE 331?

Learn the skills to be able to contribute to a modern software project
- move from CSE 143 problems toward what you’ll see in industry and in upper-level courses

Specifically, how to write code of
- higher quality
- increased complexity

We will discuss tools and techniques to help with this and the concepts and ideas behind them
- there are timeless principles to both
- widely used across the industry
What is high quality?

Code is high quality when it is

1. **Correct**
   - Everything else is of secondary importance

2. Easy to **change**
   - Most work is making changes to existing systems

3. Easy to **understand**
   - Needed for 1 & 2 above
How do we ensure correctness...

... when **people** are involved?

People have been known to
- walk into windows
- drive away with a coffee cup on the roof
- drive away still tied to gas pump
- lecture wearing one brown shoe and one black shoe

**Key Insight**
1. Can’t stop people from making mistakes
How do we ensure correctness?

Best practice: use three techniques (we’ll study each)

1. **Tools**
   - type checkers, test runners, etc.

2. **Inspection**
   - think through your code carefully
   - have another person review your code

3. **Testing**
   - usually >50% of the work in building software

Together can catch >97% of bugs.

(a.k.a. “reasoning”)
Many studies showing scale makes quality harder to achieve

- Time to write N-line program grows faster than linear
  • Good estimate is $O(N^{1.05})$ [Boehm, ‘81]
- Bugs grow like $\Theta(N \log N)$ [Jones, ‘12]
  • 10% of errors are between modules [Seaman, ‘08]
- Communication costs dominate schedules [Brooks, ‘75]
- Small probability cases become high probability cases
  • Corner cases are more important with more users

Corollary: quality must be even higher, per line, in order to achieve overall quality in a large program
How do we cope with scale?

We tackle increased software scale with **modularity**
- Split code into pieces that can be built independently
- Each must be documented so others can use it
- Also helps understandability and changeability
What are the goals of CSE 331?

In summary, we want our code to be:

1. Correct
2. Easy to change
3. Easy to understand
4. Modular

These qualities also allow us to solve more complex problems
- increased complexity = larger scale and sophistication
Reasoning
Our Approach

• We will learn a set of **formal tools** for proving correctness
  - math can seem daunting – it will connect back!
  - later, this will also allow us to generate the code

• Most professionals can do reasoning like this in their head
  - most do an *informal* version of what we will see
  - with practice, it will be the same for you

• Formal version has key advantages
  - teachable
  - mechanical (no intuition or creativity required)
  - necessary for hard problems
    • we turn to formal tools when problems get too hard
Formal Reasoning

- Invented by Robert Floyd and Sir Anthony Hoare
  - Floyd won the Turing award in 1978
  - Hoare won the Turing award in 1980
Terminology of Floyd Logic

- The **program state** is the values of all the (relevant) variables
- An **assertion** is a true / false claim (proposition) about the state at a given point during execution (e.g., on line 39)
- An assertion **holds** for a program state if the claim is true when the variables have those values
- An assertion before the code is a **precondition**
  - these represent assumptions about when that code is used
- An assertion after the code is a **postcondition**
  - these represent what we want the code to accomplish
Hoare Triples

• 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
Notation

• Floyd logic writes assertions in {..}
  – since Java code also has {..}, I will use {...}
  – e.g., {{ w >= 1 }} x = 2 * w; {{ x >= 2 }}

• Assertions are math / logic not Java
  – you can use the usual math notation
    • (e.g., = instead of == for equals)
  – purpose is communication with other humans (not computers)
  – we will need and, or, not as well
    • can also write use \( \land \) (and) \( \lor \) (or) etc.

• The Java language also has assertions (assert statements)
  – throws an exception if the condition does not evaluate true
  – we will discuss these more later in the course
Example 1

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

\[ \{ x \neq 0 \} \quad y = x \times x ; \quad \{ y > 0 \} \]
Example 1

Is the following Hoare triple valid or invalid?

- assume all variables are integers and there is no overflow

\[\{ x \neq 0 \} \ y = x \times x; \ {\{ y > 0 \}}\]

Valid

- \( y \) could only be zero if \( x \) were zero (which it isn’t)
Example 2

Is the following Hoare triple valid or invalid?
   - assume all variables are integers and there is no overflow

\[
\{ z \neq 1 \} \ y = z*z; \ \{ y \neq z \}
\]
Example 2

Is the following Hoare triple valid or invalid?
  - assume all variables are integers and there is no overflow

\[
\{ \{ z \neq 1 \} \} \ y = z \times z; \ \{ \{ y \neq z \} \}
\]

Invalid
  • counterexample: \( z = 0 \)
Checking Validity

• So far: decided if a Hoare triple is valid by ... **hard** thinking

• Soon: mechanical process for reasoning about
  – assignment statements
  – [next section] conditionals
  – [next lecture] loops
  – (all code can be understood in terms of those 3 elements)

• Next: terminology for comparing different assertions
  – useful, e.g., to compare possible preconditions
If $P_1$ implies $P_2$ (written $P_1 \Rightarrow P_2$), then:
- $P_1$ is stronger than $P_2$
- $P_2$ is weaker than $P_1$

Whenever $P_1$ holds, $P_2$ also holds
- So it is more (or at least as) “difficult” to satisfy $P_1$
  - the program states where $P_1$ holds are a subset of the program states where $P_2$ holds
- So $P_1$ puts more constraints on program states
- So it is a stronger set of requirements on the program state
  - $P_1$ gives you more information about the state than $P_2$
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
Floyd Logic Facts

• Suppose \{P\} S \{Q\} is valid.

• If \(P_1\) is stronger than \(P\), then \{\(P_1\)\} S \{\(Q\)\} is valid.

• If \(Q_1\) is weaker than \(Q\), then \{\(P\)\} S \{\(Q_1\)\} is valid.

• Example:
  – Suppose \(P\) is \(x \geq 0\) and \(P_1\) is \(x > 0\)
  – Suppose \(Q\) is \(y > 0\) and \(Q_1\) is \(y \geq 0\)
  – Since \{\(x \geq 0\)\} \(y = x+1\) \{\(y > 0\)\} is valid, \{\(x > 0\)\} \(y = x+1\) \{\(y \geq 0\)\} is also valid
Floyd Logic Facts

- Suppose \( \{P\} S \{Q\} \) is valid.
- If \( P_1 \) is stronger than \( P \), then \( \{P_1\} S \{Q\} \) is valid.
- If \( Q_1 \) is weaker than \( Q \), then \( \{P\} S \{Q_1\} \) is valid.

**Key points:**
- always okay to **strengthen** a precondition
- always okay to **weaken** a postcondition
Floyd Logic Facts

- When is $\{P\}; \{Q\}$ is valid?
  - with no code in between

- Valid if any state satisfying $P$ also satisfies $Q$
- I.e., if $P$ is stronger than $Q$
Forward & Backward Reasoning
Example of Forward Reasoning

Work forward from the precondition

\[
\begin{align*}
\{\ w & > 0 \ \}& \\
    x & = 17; \\
\{\ _________________________________ \} \\
    y & = 42; \\
\{\ _________________________________ \} \\
    z & = w + x + y; \\
\{\ _________________________________ \}
\end{align*}
\]
Example of Forward Reasoning

Work forward from the precondition

```plaintext
{{ w > 0 }}
  x = 17;
{{ w > 0 and x = 17 }}
  y = 42;
{{ ____________________________ }}
  z = w + x + y;
{{ ____________________________ }}
```
Example of Forward Reasoning

Work forward from the precondition

\[
\{ w > 0 \} \quad x = 17; \\
\{ w > 0 \text{ and } x = 17 \} \quad y = 42; \\
\{ w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \quad z = w + x + y; \\
\{ \text{________________________} \} 
\]
Example of Forward Reasoning

Work forward from the precondition

\[
\begin{align*}
\{& w > 0 \} \\
& x = 17; \\
\{& w > 0 \text{ and } x = 17 \} \\
& y = 42; \\
\{& w > 0 \text{ and } x = 17 \text{ and } y = 42 \} \\
& z = w + x + y; \\
\{& w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y \}
\end{align*}
\]
Example of Forward Reasoning

Work forward from the precondition

\[
\begin{align*}
\{ & \text{w > 0 } \} \\
\text{x} &= 17; \\
\{ & \text{w > 0 and x = 17 } \} \\
\text{y} &= 42; \\
\{ & \text{w > 0 and x = 17 and y = 42 } \} \\
\text{z} &= \text{w + x + y}; \\
\{ & \text{w > 0 and x = 17 and y = 42 and z = w + 59 } \}
\end{align*}
\]
Forward Reasoning

• Start with the **given** precondition
• Fill in the **strongest** postcondition

• For an assignment, \( x = y \)...
  – add the fact “\( x = y \)” to what is known
  – important **subtleties** here... (more on those later)

• Later: if statements and loops...
Example of Backward Reasoning

Work backward from the desired postcondition

\[
\begin{align*}
\{ z < 0 \} \\
\{ \text{________________________} \} \\
\text{z} &= \text{w} + \text{x} + \text{y} \\
\{ \text{________________________} \} \\
\text{x} &= 17 \\
\{ \text{________________________} \} \\
\text{y} &= 42 \\
\{ \text{________________________} \}
\end{align*}
\]
Example of Backward Reasoning

Work backward from the desired postcondition

\{\{ \text{__________________________} \}\}
\hspace{0.5cm} x = 17;
\{\{ \text{__________________________} \}\}
\hspace{0.5cm} y = 42;
\{\{ w + x + y < 0 \}\}
\hspace{0.5cm} z = w + x + y;
\{\{ z < 0 \}\}
Example of Backward Reasoning

Work backward from the desired postcondition

\[ \{ \text{_____________________________} \} \]
\[ x = 17; \]
\[ \{ w + x + 42 < 0 \} \]
\[ y = 42; \]
\[ \{ w + x + y < 0 \} \]
\[ z = w + x + y; \]
\[ \{ z < 0 \} \]
Example of Backward Reasoning

Work backward from the desired postcondition

\[
\begin{align*}
\{ w + 17 + 42 < 0 \} \\
x &= 17; \\
\{ w + x + 42 < 0 \} \\
y &= 42; \\
\{ w + x + y < 0 \} \\
z &= w + x + y; \\
\{ z < 0 \}
\end{align*}
\]
Backward Reasoning

• Start with the **required** postcondition
• Fill in the **weakest** precondition

• For an assignment, $x = y$:
  - just replace “x” with “y” in the postcondition
  - if the condition using “y” holds beforehand, then the condition with “x” will afterward since $x = y$ then

• Later: if statements and loops...
Use forward reasoning to determine if this code is correct:

```c
{{ w > 0 }}
  x = 17;
  y = 42;
  z = w + x + y;
{{ z > 50 }}
```
Example of Forward Reasoning

\[
\begin{align*}
\{ \{ w > 0 \}\} \\
& x = 17; \\
\{ \{ w > 0 \text{ and } x=17 \}\} \\
& y = 42; \\
\{ \{ w > 0 \text{ and } x=17 \text{ and } y=42 \}\} \\
& z = w + x + y; \\
\{ \{ w > 0 \text{ and } x=17 \text{ and } y=42 \text{ and } z = w + 59 \}\} \\
\{ \{ z > 50 \}\} \\
\end{align*}
\]

Do the facts that are always true imply the facts we need? I.e., is the bottom statement weaker than the top one?

(Recall that weakening the postcondition is always okay.)
Use backward reasoning to determine if this code is correct:

\[
\begin{align*}
\{ \{ w < -60 \} \} \\
x &= 17; \\
y &= 42; \\
z &= w + x + y; \\
\{ \{ z < 0 \} \}
\end{align*}
\]
Correctness by Backward Reasoning

Use backward reasoning to determine if this code is correct:

\[
\begin{align*}
\{ w < -60 \} \\
\{ w + 17 + 42 < 0 \} \\
\quad x = 17; \\
\{ w + x + 42 < 0 \} \\
\quad y = 42; \\
\{ w + x + y < 0 \} \\
\quad z = w + x + y; \\
\{ z < 0 \}
\end{align*}
\]

Do the facts that are always true imply the facts we need? I.e., is the top statement stronger than the bottom one?

(Recall that strengthening the precondition is always okay.)
Combining Forward & Backward

It is okay to use both types of reasoning
• Reason forward from precondition
• Reason backward from postcondition

Will meet in the middle:

\[
\{ \{ P \} \} \\
S1 \\
S2 \\
\{ \{ Q \} \}
\]
Combining Forward & Backward

It is okay to use both types of reasoning
- Reason forward from precondition
- Reason backward from postcondition

Will meet in the middle:

```
{{ P }}
S1
{{ P1 }}
{{ Q1 }}
S2
{{ Q }}
```

Valid provided P1 implies Q1
Combining Forward & Backward

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

- Forward reasoning can **fail** if applied blindly...
  
  ```
  {{
  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 }
  ```

  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

```plaintext

{{ }}

  w = x + y;
{{ w = x + y }}

  x = 4;
{{ w = x\_1 + y and x = 4 }}

  y = 3;
{{ w = x\_1 + y\_1 and x = 4 and y = 3 }}
```
Fix 2 (better)

• Express prior values in terms of the current value

{ {  }  }
    \texttt{w = x + y;}
{ { w = x + y }  }
\texttt{x = x + 4;}
{ { w = x_1 + y \text{ and } x = x_1 + 4 }  }

Now, \( x_1 = x - 4 \)

so \( w = x_1 + y \Leftrightarrow w = x - 4 + y \)

⇒ { { 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
Forward vs. Backward

• Forward reasoning:
  – Find strongest postcondition
  – Intuitive: “simulate” the code in your head
    • BUT you need to change facts to refer to prior values
  – Inefficient: Introduces many irrelevant facts
    • usually need to weaken as you go to keep things sane

• Backward reasoning
  – Find weakest precondition
  – Formally simpler, but (initially) unintuitive
  – Efficient
Before next class...

1. Familiarize yourself with website:
   
   http://courses.cs.washington.edu/courses/cse331/22su/
   
   - read the welcome email
   - read the syllabus

2. Try to do HW1 and syllabus quiz before section tomorrow!
   
   - submit a PDF on Gradescope
   - limit this to at most 90 min
   - do not use formal reasoning