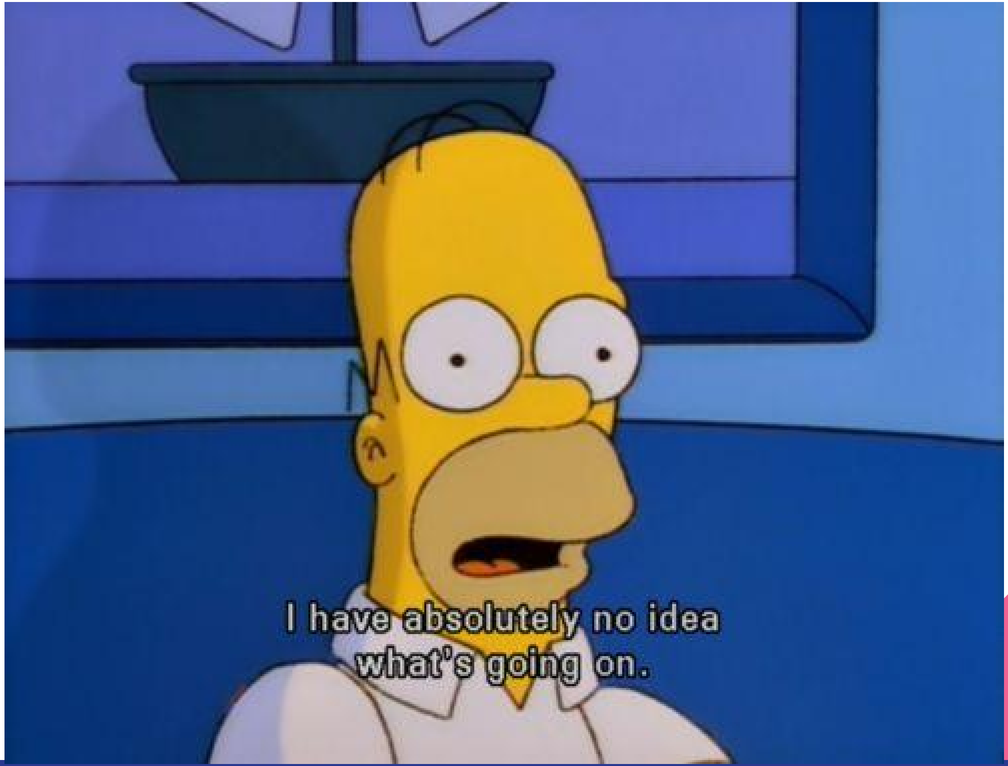
CS 130: Software Engineering

Lab 1C: Week 8 Discussion

Weakest Precondition

My Office Hours This Week...



Weakest Precondition

- Very mechanical
- Practice makes perfect!

Weakest Precondition Cheat Sheet

- wp(x := T, P)
- Changing the value of a variable x to T
- P is the postcondition
- Then, simply swap all of the occurrences of x in P with T!
    - Let's name this rule "**Rule-Assignment**"

```
public char[] foo(int i) {
    int n = i + 2;  // assume our postcondition is
                    // n >= 0 AND n < 14
    // ...
}
```

Weakest Precondition Cheat Sheet

- Weakest precondition is
    - i + 2 >= 0 AND i + 2 < 14

```java
public char[] foo(int i) {
    int n = i + 2;  // assume our postcondition is
                    // n >= 0 AND n < 14
    // ...
}
```

Weakest Precondition Cheat Sheet

- $wp(S_i; S_{i+1}, P) \equiv wp(S_i, wp(S_{i+1}, P))$
  - S_i is the i-th statement, S_{i+1} is the (i+1)-th statement
  - Let's name this rule "**Rule-Sequence**"
  - Compute the weakest precondition from the **bottom** of the code!
  - Why?
    - Consider the weakest precondition of the following program

```java
public char[] foo(int i) {
    int n = i + 2;          // S1
    n += i;                 // S2
    a = new char[n];        // S3
    return a;               // true
}
```

```java
public char[] foo(int i) {
    int n = i + 2;          // S1
                            // wp(S1; S2; S3, true)
                            // = wp(S1, wp(S2; S3, true))
                            // = wp(S1, wp(S2, wp(S3, true)))
    n += i;                 // S2
                            // wp(S2; S3, true) = wp(S2, wp(S3, true))
    a = new char[n];        // S3
                            // wp(S3, true)
    return a;               // true
}
```

Weakest Precondition Cheat Sheet

- We need wp(S3, true) to compute wp(S2; S3, true), because of Rule-Sequence
- Similarly, we need wp(S2; S3, true) to compute (S1; S2; S3, true), also because of Rule-Sequence
- Let’s try to generalize the algorithm for computing weakest precondition

Weakest Precondition Cheat Sheet

- How to compute weakest precondition, given a code snippet
  - Start from the very bottom
  - Compute the weakest precondition of the bottommost statement with the given post-condition (usually true)
  - Compute the weakest precondition of the second-to-bottommost statement and use the weakest precondition of the bottommost statement as the post-condition
  - Repeat until you reach the topmost statement
  - Done! Pretty simple right?

Weakest Precondition Cheat Sheet

- If-Statement
  - wp(if Guard then S1 else S2, P) ≡ (Guard AND wp(S1, P)) OR (¬Guard AND (wp(S2, P))
  - Let's name this rule **"Rule-If"**

Weakest Precondition Cheat Sheet

- Initializing an array
  - wp(x = new Char[n], P) ≡ (n >= 0 AND P), remove x != null in P and substitute all of the occurrences of x.length to n
  - Let’s name this rule **“Rule-ArrayInit”**

Weakest Precondition Cheat Sheet

- Accessing an array
  - wp(x[i], P) ≡ (i < x.length() AND i >= 0 AND x != null AND P)
  - Let's name this rule **"Rule-ArrayAccess"**

Back to our Example

- Try it yourself! Compute the weakest precondition of the following code

```java
public char[] foo(int i) {
    int n = i + 2;          // S1
    n += i;                 // S2
    a = new char[n];        // S3
    return a;               // true
}
```

Solutions

- Compute the weakest precondition of the bottommost statement with the given post-condition

```
// postcondition = true
a = new char[n];     // wp(a = new char[n], true) ===
                     // using Rule-ArrayInit
                     // {n >= 0}
```

Solutions

- Compute the weakest precondition of the second-to-bottommost statement and use the weakest precondition of the bottommost statement as the post-condition

```
n += i;     // wp(n := n+i, n >= 0) ===
            // using Rule-Assignment
            // {n+i >= 0}
```

Solutions

- Repeat until you reach the topmost statement

```
n = i+2;      // wp(n:= i+2, n+i >= 0) ===
              // using Rule-Assignment
              // {i+2+i >= 0} ===
              // {2i+2 >= 0} ===
              // {i+1 >= 0}
```

Back to our Class Exercise

- Try it yourself! Compute the weakest precondition of the following code.

```
if (x != null) {
        n = x.f;
} else {
        n = z+1;
        z = 2*z+1;
}

a = new char[n-2];
c = a[z];
```

Solutions

- Compute the weakest precondition of the bottommost statement with the given post-condition

```
// Postcondition: true
// start from the very bottom
c = a[z];                // wp(c = a[z], true) ===
                         // using Rule-ArrayAccess
                         // {z >= 0 AND z < a.length}
```

Solutions

- Compute the weakest precondition of the second-to-bottommost statement and use the weakest precondition of the bottommost statement as the post-condition

```
a = new char[n-2];  // wp(a = new char[n-2], z >= 0 AND z < a.length) ===
                    // using Rule-ArrayInit
                    // {n-2 >= 0, z < n-2, z >= 0} ===
                    // {n >= 2, n > z+2, z >= 0}
```

Solutions

- Repeat until you reach the topmost statement
  - Next slide

```
if (x != null) {        // wp(if ... else ..., n >= 2, n > z+2, z >= 0) ===
    n = x.f;            // using Rule-If
} else {                // {x!=null AND wp(n:=x.f, n >= 2, n > z+2, z >= 0}
    n = z+1;            //      OR {x==null AND wp (n:=z+1; z:=2z+1, n >= 2, n > z+2, z >= 0)}

    z = 2*z+1;          // Let's compute wp(n:=x.f, n >= 2, n > z+2, z >= 0) first
}                       // wp(n:=x.f, n >= 2, n > z+2, z >= 0) ===
                        // using Rule-Assignment
                        // {x.f >= 2, x.f > z+2, z >= 0}

                        // Now, let's compute wp(n:=z+1; z:=2z+1, n >= 2, n > z+2, z >= 0)
                        // wp(n:=z+1; z:=2z+1, n >= 2, n > z+2, z >= 0) ===
                        // using Rule-Sequence
                        // wp(n:=z+1, wp(z:=2z+1, n >= 2, n > z+2, z >= 0))

                        // Let's compute wp(z:=2z+1, n >= 2, n > z+2, z >= 0)
                        // wp(z:=2z+1, n >= 2, n > z+2, z >= 0) ===
                        // using Rule-Assignment
                        // {n >= 2, n > 2z+1+2, 2z+1 >= 0} ===
                        // {n >= 2, n > 2z+3, 2z+1 >= 0}

                        // Let's go back to wp(n:=z+1, wp(z:=2z+1, n >= 2, n > z+2, z >= 0))
                        // wp(n:=z+1, n > 2z+3, 2z+1 >= 0) ===
                        // using Rule-Assignment
                        // {z >= 1, 0 > z+2, 2z+1 >= 0} ===
                        // {z >= 1, z < -2, 2z+1 >= 0}
                        // Notice that z cannot be greater than or equal to 1 AND less than -2 at the same time
                        // therefore, it can be further simplified to false

                        // Therefore, our final answer is
                        // {x != null AND x.f >= 2, x.f > z+2, z >= 0} OR {x == null AND False} ===
                        // {x != null AND x.f >= 2, x.f > z+2, z >= 0} OR {False} ===
                        // {x != null AND x.f >= 2, x.f > z+2, z >= 0}
```

More Weakest Precondition Exercise

```java
private int foo2(int a, int b, int c) {
    int size = a;
    int index = b;

    int value = 2 * a - 1;
    if (a * a < value) {
        size++;
    } else {
        size--;
    }

    Random rand = new Random();
    int[] arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = rand.nextInt(i) + 1;
    }

    return arr[c];
}
```

Solutions

```
wp(size := a, (a*a < 2a-1, size+1 >= 0, b < size+1, b >= 0) OR (a*a >= 2a-1, size-1 >= 0, b < size-1, b >= 0))
=== {(a*a < 2a-1, a+1 >= 0, b < a+1, b >= 0) OR (a*a >= 2a-1, a-1 >= 0, b < a-1, b >= 0) (using Rule-Assignment)
=== {(a*a-2a+1 < 0, a >= -1, b < a+1, b >= 0) OR (a*a-2a+1 >= 0, a >= 1, b < a-1, b >= 0)}
=== {(a-1)(a-1) < 0, a >= -1, b < a+1, b >= 0) OR ((a-1)(a-1) >= 0, a >= 1, b < a-1, b >= 0)} (square of a number cannot be negative)
=== {False OR ((a-1)(a-1) >= 0, a >= 1, b < a-1, b >= 0)}
=== {(a-1)(a-1) >= 0, a >= 1, b < a-1, b >= 0} (square of a number is always positive)
=== {True, a >= 1, b < a-1, b >= 0}
=== {a >= 1, b < a-1, b >= 0}


wp(index := b, (a*a < 2a-1, size+1 >= 0, index < size+1, index >= 0) OR (a*a >= 2a-1, size-1 >= 0, index < size-1, index >= 0))
=== {(a*a < 2a-1, size+1 >= 0, b < size+1, b >= 0) OR a*a >= 2a-1, size-1 >= 0, b < size-1, b >= 0} (using Rule-Assignment)

wp(value := 2a-1, (a*a < value, size+1 >= 0, index < size+1, index >= 0) OR (a*a >= value, size-1 >= 0, index < size-1, index >= 0))
=== {a*a < 2a-1, size+1 >= 0, index < size+1, index >= 0) OR (a*a >= 2a-1, size-1 >= 0, index < size-1, index >= 0)} (using Rule-Assignment)

wp(if ... else ..., size >= 0, index < size, index >= 0) ===
(a*a < value, wp(size := size+1, size >= 0, index < size, index >= 0))
    OR (a*a >= value, wp(size := size-1, size >= 0, index < size, index >= 0)) (using Rule-If)
=== (a*a < value, size+1 >= 0, index < size+1, index >= 0)
    OR (a*a >= value, size-1 >= 0, index < size-1, index >= 0) (using Rule-Assignment)

wp(int[] arr = new int[size], index < array.length, index >= 0) === {size >= 0, index < size, index >= 0} (using Rule-ArrayInit)

wp(for-loop, index < array.length, index >= 0) === {index < array.length, index >= 0}
We don't need to use Rule-ArrayInit because i is always bounded by the loop condition: 0 <= i < size

wp(arr[c], true) === {index < array.length, index >= 0} (using Rule-ArrayAccess)
```

Loop Invariants

Loop Invariant Cheat Sheet

- {P} while B do S end {Q}
  - P is precondition
  - Q is postcondition
  - J is loop invariant
  - vf is variant function

Loop Invariant Cheat Sheet

- {P} while B do S end {Q} (J is loop invariant, vf is variant function)
    - Invariant initially: P => J
    - Invariant maintained: {J ^ B} S {J}
    - Invariant sufficient: J ^ ¬B => Q
    - vf bounded: J ^ B => (0 <= vf)
    - vf decreases: {J ^ B ^ vf = VF} S {vf < VF}

What is a loop invariant?

- A loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop
    - A condition that is necessarily true right before statement 6 AND a condition that is necessarily true right after statement 7

```
1  Precondition P {n > k AND k > 0}
2       i = k+1
3       product = 1
4       while (i != n + 1)
5       do
6           product = product * i
7           i = i+1
8       end
9  PostCondition Q { product = n!/k! }
10
```

How to guess Loop Invariants

- Try writing out couple of iterations
- Try to guess the loop invariants from
  - Postcondition
  - Loop guard
- However, proving the validity of loop invariants require “structural induction”, which is not in the scope of CS 130

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

How to guess Loop Invariants

- On loop entry
  - product = 1
  - i = k + 1
- After iteration 1
  - product = k + 1
  - i = k + 2
- After iteration 2
  - product = (k + 1) * (k + 2)
  - i = k + 3
- What is our loop invariant? J: product = (i - 1)! / k!

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- Invariant initially: P => J
  - Compute wp(T, J), where T is the set of statements before the loop
  - Show that P implies wp(T, J)

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- wp(i := k+1; product := 1, product = (i - 1)! / k!)

  (using Rule-Sequence and Rule-Assignment)

  === wp(i := k+1, 1 = (i - 1)! / k!)

  (using Rule-Assignment)

  === 1 = (k + 1 - 1)! / k!

  === 1 = k! / k!

  === true

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- Invariant maintained: {J ^ B} S {J}
  - Compute wp(S, J), where S is the set of statements contained in the loop
  - Show that J AND B implies wp(S, J)

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- wp(S, J)

  === wp(product = product * i; i = i + 1, product = (i - 1)! / k!

  (using Rule-Sequence and Rule-Assignment)

  === wp(product = product * i, product = (i + 1 - 1)! / k!

  (using Rule-Assignment)

  === product * i = i! / k!

  === product = (i - 1)! / k!

- {J AND B}

  === product = (i - 1)! / k! AND i != n + 1

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- Invariant sufficient: J ^ ¬B => Q
  - Show that J AND (NOT B) implies Q, where B is the loop guard and Q is the post-condition

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Loop Invariant Cheat Sheet

- {J AND (NOT B)}

  === product = (i - 1)! / k! AND i = n + 1

  === product = (n + 1 - 1)! / k!

  === product = n! / k!

- {J AND (NOT B)} = Q

  Therefore, {J AND (NOT B)} => Q

```
Precondition P {n > k AND k > 0}
    i = k+1
    product = 1
    while (i != n + 1)
    do
        product = product * i
        i = i+1
    end
PostCondition Q { product = n!/k! }
```

Code Review

Purposes of Code Review

- Improving the code
    - Finding bugs
    - Anticipating possible bugs
    - Checking the clarity of the code
    - Checking for consistency with the project’s style standards

Purposes of Code Review

- Improving the programmer
    - Way of learning and teaching each other
        - New language feature
        - Changes in the design of the project or its coding standards
        - New techniques

General Principles of Good Code

- Don’t Repeat Yourself
- Comments where needed
- Fail fast
- Avoid magic numbers (constants)
- One purpose for each variable
- Use good names
- No global variables
- Return results, don’t print them
- Use white-space for readability

Identify the issues of the following code based on --->

```
public static int mweh(int first, int second, int third) {
    if (first == 2) {
        second += 31;
    } else if (first == 3) {
        second += 59;
    } else if (first == 4) {
        second += 90;
    } else if (first == 5) {
        second += 31 + 28 + 31 + 30;
    } else if (first == 6) {
        second += 31 + 28 + 31 + 30 + 31;
    } else if (first == 7) {
        second += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (first == 8) { second += 31 + 28 + 31 + 30 + 31 + 30 + 31; } else if (first == 9) {
        second += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (first == 10) {
        second += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (first == 11) {
        second += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (first == 12) {
        second += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return second;
}
```

- Don’t Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Return results, don’t print them
- Use whitespace for readability

Don’t Repeat Yourself

- How many times is the number of days in April written in mweh?
- Suppose our calendar changed so that February really has 30 days instead of 28. How many numbers in this code have to be changed?

Avoid Magic Numbers

- The months 2, …, 12 would be far more readable as `FEBRUARY`, …, `DECEMBER`.
- The days-of-months 30, 31, 28 would be more readable (and eliminate duplicate code) if they were in a data structure like an array, list, or map, e.g. `MONTH_LENGTH[month]`.