Lecture 3-4

Pre-condition, Post-condition, Partial Correctness of an Algorithm, Total Correctness of Algorithm, and Illustrative Example

Pre-Condition, Post-Condition, Loop Invariant

A ***precondition*** *(P)* is a statement placed before the segment. It must be true prior to entering the segment for it to work correctly. Preconditions are often placed either before loops or at the entry point of functions and procedures

A ***postcondition*** *(Q)* is a statement placed after the end of the segment. It should be true when the execution of the segment is complete. Postconditions are often placed either after loops or at exit points of functions and procedures.

An ***invariant*** *(I)* is a statement placed in a code segment that is repeated, should be true each time the loop execution reaches that point. Invariants are often used in a loops and recursions.

Checking Loops

› The correctness of a loop can be ascertained by making the following sequence of checks:

- Check that the precondition implies that the invariant is initially true.
- Check that the invariant is preserved by the loop body.
- Check that the loop invariant, together with the terminating condition, implies that the postcondition is true.

› If the loop satisfies the above checks, then the postcondition must be true whenever the precondition is true, provided that the loop terminates. Since it possible for a loop to run forever, one more check is needed:

› Check that the loop terminates.

Proof of Correctness

How to Specify Computational Problem?

Correctness Proof

- A computational problem is specified by one (or more) pairs of *preconditions* and *postconditions*.
  - Precondition: A condition that one might expect to be satisfied when the execution of a program begins. This generally involves the algorithm's inputs as well as initial values of global variables.
  - Postcondition: A condition that one might want to be satisfied when the execution of a program ends. This might be
    - A set of relationships between the values of inputs (and the values of global variables when execution started) and the values of outputs (and the values of global variables on a program's termination), or
    - A description of output generated, or exception(s) raised.

Pre-Condition and Post-Condition

- Precondition: what's true before a block of code
- Postcondition: what's true after a block of code

- Example: computing the square root of a number

- Precondition: the input x is a positive real number
- Postcondition: the output is a number y such that $y^2=x$

Example

```
def factorial(n) :
    '''
    precondition:  n >= 0
    postcondition:  return value equals n!
    '''
```

Enforcing preconditions

```
def factorial(n) :
    '''
    precondition:  n >= 0
    postcondition:  return value equals n!
    '''
    if (n < 0) : raise ValueError
```

What about Post-conditions?

```
def factorial(n) :
    '''
    precondition:  n >= 0
    postcondition:  return value equals n!
    '''
    if (n < 0) : raise ValueError
```

- `assert factorial(5)==120`
- Can use assertions to verify that postconditions hold

An assertion is **a claim about the state of the program each time execution reaches a particular point in the program text (or** step in the algorithm

Loop invariants as a way of reasoning about the state of your program

<pre-condition: n>0

```
i = 0
 while (i <  n) :
     i = i+1
```

<post-condition: i==n>

We want to prove the post-condition: i==n right after the loop

Example: loop index value after a loop

```
// precondition: n>=0
i = 0
// i<=n      loop invariant
 while (i < n) :
        // i < n   test passed
        //     AND
        //     i<=n   loop invariant
         i = i + 1
        // i <= n   loop invariant
 //   i >= n        WHY?
//       AND
// i <= n
// → i==n
```

So we can conclude the obvious:

i==n right after the loop

But what if the body were:

i = i+2 ?

Example: Search Problem Specification

› *Precondition P_1*: Inputs include

- n: a positive integer
- A: an integer array of length n, with entries
  $A[0], A[1], \ldots, A[n-1]$
- key: An integer found in the array (i.e., such that $A[i] = key$ for at least one integer i between 0 and $n-1$)

› *Postcondition Q_1*:

- Output is the integer i such that $0 \leq i < n$, $A[j] \neq key$ for every integer j such that $0 \leq j < i$, and such that $A[i] = key$
- Inputs (and other variables) have not changed

› This describes what should happen for a "successful search."

Example: Search Problem Specification

- Precondition P_2: Inputs include
  - n: a positive integer
  - A: an integer array of length n, with entries
    A[0], A[1], . . . , A[n-1]
  - key: An integer not found in the array (i.e., such that $A[i] \neq key$ for every integer i between 0 and n-1)
- Postcondition Q_2:
  - A *notFoundException* is thrown
  - Inputs (and other variables) have not changed
    This describes what should happen for an “unsuccessful search.”

Examples: Pre-Condition and Post Condition

- Let's start with a simple code example:
- ```
  x = 17;
  y = 42;
  z = x + y;
  ```
- We annotate the code to show this information:
- ```
  { true }
  x = 17;
  { x = 17 }
  y = 42;
  { x = 17 ∧ y = 42 }
  z = x + y;
  { x = 17 ∧ y = 42 ∧ z = 59 }
  { true }
  ```

- An assertion is an assumption that something is {true}. An assertion is a logical formula inserted at some point in a program. There are two special assertions: the precondition and the postcondition.
- {true} is the precondition
- $\{ x = 17 \land y = 42 \land z = 59 \}$ is the postcondition

Additional Examples: Weakest Pre-Condition and Post Condition

```
x = x - 2;
z = x + 1;
{ z != 0 }
```

```
x = 2 * y;
z = x + y;
{ z > 0 }
```

```
w = 2 * w;
z = -w;
y = v + 1;
x = min(y, z);
{ x < 0 }
```

The solutions are:

```
{ x !- 1 }
x = x - 2;
{ x != -1 }
z = x + 1;
{ z != 0 }
```

```
{ y > 0 }
x = 2 * y;
{ x + y > 0 }
z = x + y;
{ z > 0 }
```

```
{ v < -1 V w > 0 }
w = 2 * w;
{ v < -1 V w > 0 }
z = -w;
{ v < -1 V z < 0 }
y = v + 1;
{ y < 0 V z < 0 }
x = min(y, z);
{ x < 0 }
```

Weakest Pre-Condition (IF – ELSE statement)

- wp (IF, Q), We write the statement like if (B) S_1 else S_2
- Suppose B is true. Because S_1 is executed and Q must be true afterward, the weakest precondition for the entire IF statement will be the weakest precondition for S_1 and Q, i.e. $wp(S_1, Q)$
- Analogously, if B is false the weakest precondition will be $wp(S_2, Q)$
- The weakest precondition for the entire if/else statement is $wp(S_1,Q)$ when B is true and $wp(S_2, Q)$ when B is false
- $wp(IF, Q) = (B \Rightarrow wp(S_1,Q) \land !B \Rightarrow wp(S_2,Q))$
  $= (B \land wp(S_1, Q)) \lor (!B \land wp(S_2, Q))$

Example: Weakest Pre-Condition (IF – ELSE statement)

- $wp(IF, Q) = (B \Rightarrow wp(S_1,Q) \wedge\ !B \Rightarrow wp(S_2,Q))$
  $= (B \wedge wp(S_1, Q)) \vee (!B \wedge wp(S_2, Q))$

Using the formula above

(1)

```
if (x < 5)
        x = x*x;
else
        x = x+1;
{ x >= 9 }
```

$wp(IF, x >= 9) = (x < 5 \wedge wp(x = x*x, x>=9)) \vee (x >=5 \wedge wp(x = x+1, x >=9))$
$= (x < 5 \wedge x*x >= 9) \vee (x >= 5 \wedge x+1 >= 9)$
$= (x <= -3) \vee (x >= 3 \wedge x < 5) \vee (x >= 8)$

(2)

```
if (x != 0)
        z = x;
else
        z = x+1;
{ z > 0 }
```

$wp(IF, z > 0) = (x != 0 \wedge wp(z = x, z > 0)) \vee (x == 0 \wedge wp(z = x+1, z > 0))$
$= (x != 0 \wedge x > 0) \vee (x == 0 \wedge x+1 > 0)$
$= (x > 0) \vee (x == 0)$
$= (x >= 0)$

Loop Invariant

- A loop invariant is a statement about an algorithm's loop that:
  - is true before the first iteration of the loop and
  - if it's true before an iteration, then it remains true before the next iteration.
- **If we can prove that those two conditions hold for a statement, then it follows that the statement will be true before each iteration of the loop.**

Let's say that we want to sum an array of real numbers

Algorithm 1: An algorithm to sum an array

Data: $a = [a_1, a_2, \ldots, a_n]$: an array of n real numbers
Result: The sum of all elements of a
$s \leftarrow 0$
for $i = 1, 2, \ldots, n$ **do**
  $s \leftarrow s + a_i$
end
return s

To prove working of algorithm, we must prove that algorithm works after the loop end, **s** is equal to the sum of the numbers in a.

$$s = \sum_{i=1}^{n} a_i$$

Loop Invariant

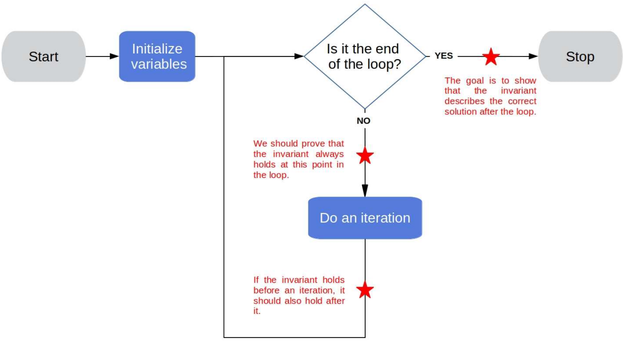
- At beginning of the **i**th iteration, s is equal to the sum of the first $i-1$ elements of a.
- In this example, loop ends when $i = n + 1$, so invariant states that at the end of the loop

$$s = \sum_{i=1}^{n} a_i$$

- The invariant hold before the first iteration corresponds to the base case of induction.
- The second condition is similar to the inductive step.
- But, unlike induction that goes on infinitely, a loop invariant needs to hold only until the loop has ended.

Loop Invariant

- For the given algorithm, we should prove loop invariant in two steps.
  - At the beginning of the loop, i=1 and s=0. The sum $\sum_{j=1}^{0} a_j$ is the sum of no numbers. Thus, loop invariant holds at the start of the i-th iteration $s = \sum_{j=1}^{i-1} a_j$
  - During the iteration, we add a_i to s, we get
    $s = a_i + \sum_{j=1}^{i-1} a_j = \sum_{j=1}^{i} a_j$
    The end of iteration i is the same as the beginning of the iteration i + 1, So the second condition is also satisfied.
- As we have shown that $s = \sum_{i=1}^{n} a_i$, at the end of the loop, proving the invariant also verified and the algorithm was correct.



Partial Correctness of An Algorithm

(1) Partial correctness – if the algorithm terminates then the output is guaranteed to be correct.

Algorithm Correctness

› *Partial Correctness:* If

- inputs satisfy the precondition *P*, and
- algorithm or program *S* is executed,
  then *either*
- *S* halts and its inputs and outputs satisfy the postcondition *Q*
  or
- *S* does not halt, at all.
  Generally written as

$$\{P\} \quad S \quad \{Q\}$$

Note: Detailed proofs rely heavily on discrete math and logic

To Prove Partial Correctness?

- Consider algorithm S:
  - Divide S into sections S_1; S_2; … ; S_K
    - assignment statements
    - Loops
    - control statements (i.e., if-then-else)
    - (other programming constructs
  - Identify intermediate assertions R_i so that
    - $\{P\}$ S_1 $\{R_1\}$
    - $\{R1\}$ S_2 $\{R_2\}$

    …………………
    - $\{R_{K-1}\}$ S_K $\{Q\}$
  - After proving each of these, we can then conclude that
    - $\{P\}$ S_1; S_2; … ; S_K $\{Q\}$
    - equivalently, $\{P\}$ S $\{Q\}$

Example: Proof of Partial Correctness

› *Problem Definition:* Finding the largest entry in an integer array.
Precondition P: Inputs include
   – n: a positive integer
   – A: an integer array of length n, with entries A[0]*; … ;* A[n-1]

Postcondition Q:
   – Output is the integer i such that 0 ≤ i < n, A[i] ≥ A[j] for every integer j such that 0 ≤ j < n
   – Inputs (and other variables) have not changed

```
int FindMax(A, n)
  i = 0
  j = 1
  while (j < n) do
    if A[j] > A[i] then
      i = j
    end if
    j = j + 1
  end while
  return i
```

Total Correctness of An Algorithm

Illustrative Examples

Total Correctness of Algorithm

(2) Total correctness – the algorithm will terminate and produce the correct output for all possible inputs.

- An algorithm is correct if for every legal input given to an algorithm, the algorithm produces the correct output. For example
  - Max(a, b) is correct if it always returns the larger value out of a, b.
  - Max(int[] A) is correct if it always returns the largest integer found in array A. That is, it returns A[i] such that $0 <= i < length(A)$ and there is no j such that $0 <= j < length(A)$ and $A[j] > A[i]$.

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

◀ Finds greatest value in the array

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **Finds greatest value in the array**
  - Assume **values** array is non-null
  - Assume **values** array contains at least one element

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

- **Finds greatest value in the array**
  - Assume values array is non-null
  - Assume values array contains at least one element
- **Can we offer a formal proof that this function works correctly?**
  - Yes, we can offer a proof based on induction

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0 | values[k]\}$

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
        i < values.Length;
        i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

Entire loop skipped if
values.Length = 1

$N = values.Length$

$max \leftarrow Max\{0 \leq k < N|values[k]\}$

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Loop invariant (which must hold true):

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Loop invariant (which must hold true):

$max \leftarrow Max\{0 \leq k < i|values[k]\}$

True when we enter the loop for $i = 1$

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Loop invariant (which must hold true

$max \leftarrow Max\{0 \leq k < i|values[k]\}$

True when we enter the loop for $i = 1$

$max \leftarrow Max\{0 \leq k < i + 1|values[k]\}$

Total Correctness of Algorithm

Example

```
int Maximum(int[] values)
{
  int max = values[0];

  for (int i = 1;
       i < values.Length;
       i++)
  {
    if (values[i] > max)
    {
      max = values[i];
    }
  }

  return max;
}
```

$max \leftarrow Max\{k = 0|values[k]\}$

$N = values.Length > 1$

Loop invariant (which must hold true):

$max \leftarrow Max\{0 \leq k < i|values[k]\}$

True when we enter the loop for $i = 1$

$max \leftarrow Max\{0 \leq k < i + 1|values[k]\}$

$N = values.Length$

$max \leftarrow Max\{0 \leq k < N|values[k]\}$

Thank You!!!

Have a good day