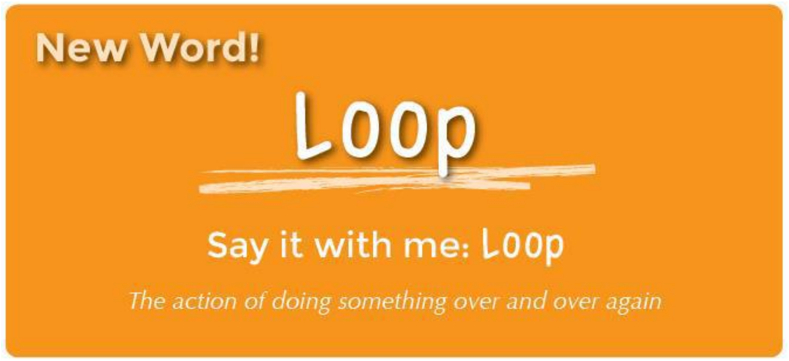
Loops


New Word!

Loop

Say it with me: Loop

The action of doing something over and over again

Weakest Precondition and Loops

- We would like to be able to find the weakest precondition {P}:

```
 { P }
while(b) {
   S;
}
 {  Q }
```

- {P} while(b) S; {Q} is a Hoare triple
- It turns out that computing the weakest precondition for loops is, in general, a hard problem
- Instead, we'll assume we can find an invariant for the loop
    - Something that gives us information about the loop and can be relied upon to be true before and after each execution of the loop

Weakest Precondition for Loops

- If we knew how many iterations, we could unroll the loop.
    - Compiler optimization does this

- In general, finding the weakest precondition is complicated even for simple loops

- {??} while(x > 0) x = x-1; {x == 0}
- WP = ¬(x > 0) →x == 0 && (x > 0) →[¬(x-1 > 0) →x-1 == 0 && (x-1 > 0) →[¬(x-2 > 0) →x-2 == 0 && (x-2 > 0) →[...
    - When do we stop expanding the loop into a logical condition?

Reasoning about Loops

Reasoning about loops is a bit more complicated than reasoning about sequence or if … else …
-- Unknown number of iterations and unknown number of paths
-- Recursion adds an additional level of complexity

Instead we will use a loop invariant to reason about a loop

Two things to prove about loops:
-- It computes correct values (partial correctness)
    That is, the postcondition holds on loop exit
-- It terminates (it is not an infinite loop)

Total correctness = Partial correctness + Loop termination

Loop Invariant

A loop invariant is a property of a program loop

- That is true before 1st iteration of the loop
- That is true after each iteration.
  - Not necessarily between statements in the loop
- It is a logical assertion
  - Abstract specification of the loop
  - A statement about the loop

- To show partial correctness
  - Loop exit condition and the LI must imply the desired postcondition
  - That is, if the loop exits, the correct result is calculated

Why do we care?

- If we have a LI that implies the postcondition at exit, we can be somewhat confident that the loop computes the correct result

How do we show partial correctness?

- Induction

Reasoning about Loops

```
PRECONDITION: {x >= 0} // assume all variables are ints

i = x
z = 0

{ LOOP INVARIANT (LI): i + z == x  && i >= 0 }
while ( i > 0 )  {
  z = z + 1;
  i = i - 1;
}
POSTCONDITION: x == z
```

Questions:
(A) Is LI true before 1st iteration?
(B) Is LI true after each iteration?
(C) If loop terminates, do loop exit condition and LI imply postcondition?
(D) Does the loop terminate?

Reasoning about Loops

Proof by Induction

(1) BASE CASE: Initially, i == x and z == 0 gives us i + z = x, i.e.,
LI holds at iteration 0 (before the loop code executes)
from precondition x >= 0 && i == x => i >= 0

(2) INDUCTION: Assuming i + z == x holds after iteration k, we show
that i + z == x holds after iteration k + 1

z_new == z + 1   and   i_new == i - 1

therefore, i_new + z_new == i - 1 + z + 1 == i + z == x
at iteration k, i > 0 or we have exited loop, i_new = i - 1 => i >= 0

(3) If the loop terminates, we know i <= 0.

```
      { !(i > 0) && (i >= 0 && i + z == x) }
  => { i == 0 && i + z == x }
  => { z == x }
  we have z == x (i.e., the POSTCONDITION)
```

(4) How do we know if the loop terminates?
-- the PRECONDITION x >= 0 guarantees that i >= 0 before the loop.
At every iteration, i decreases by 1, thus it eventually reaches 0
We will get a bit more formal about this in a while.

Reasoning about Loops

Reasoning about Loops using Induction

-- i + z == x is a loop invariant, meaning that it holds true before
   the loop and also after each/every iteration of the loop

-- even though i and z change within the loop code,
   i + z == x stays true at the END of each iteration
   true at the closing "}" of the loop

-- Above we made an inductive argument over the number of iterations
   of the given loop

-- Proof by Induction -- also called Computation Induction

-- Establish that the LI holds before iteration 0

-- Assuming LI holds after iteration k, show that it holds
   after iteration k + 1

Loop Invariant

```
{ P }           // Hoare triple
while (b) S;
{ Q }
```

Find an invariant, LI, such that

1. P ⇒ LI // true initially
2. { LI & b } S {LI } // true if the loop executes
3. {LI & ¬b} ⇒ Q // establishes the postcondition

Finding the invariant is the key to reasoning about loops.

Inductive assertions are a “complete method of proof”

Reasoning about Loops

Partial Correctness
-- Establish and prove the loop invariant (LI) using computation induction

-- Loop exit condition and the LI must imply the desired postcondition
  -- i == 0 (loop exit condition) and i + z == x (LI) imply z == x

Termination
-- Establish some decrementing function D such that
  D = minimum value implies loop exit condition
    D == minimum => !b
    b is the loop condition
  D decreases at each loop iteration.
  Show that D reaches its minimum
  Ideally minimum D == 0

Example

precondition: arr != null && arr.length == len && len >= 0; assume ints

```
int sum = 0;
int i = 0;
while ( i < len ) {
  sum = sum + arr[i];
  i = i + 1;
}
```

postcondition: (result is the sum of all elements in array arr)

sum == arr[0] + arr[1] + ... + arr[arr.length-1]

LI: i <= len && sum == arr[0] + ... + arr[i-1]

(1) BASE CASE: does the LI hold before the loop?

    i <= len && sum == arr[0] + ... + arr[i-1]

  the LI holds, given that i = 0 and that no values from
  the array arr have been summed yet. sum is initially 0.
  (i <= len) = (0 <= len) by precondition

(2) INDUCTION: assume the LI holds at iteration k, does it hold
    at iteration k+1?

  sum_new == sum + arr[i] = arr[0] + … + arr[i-1] + arr[i]
  i_new == i + 1

  sum_new == sum + arr[i_new-1] = arr[0] + ... + arr[i_new-1]

  i_new <= len also holds; i < len at iteration k.
        If i == len at iteration k, there would be no iteration k+1

(3) LI && !b => postcondition
  i <= len && sum == arr[0] + ... + arr[i-1] && !(i<len)
     => (i==len) && sum = arr[0] + ... + arr[i-1]
     => sum == arr[0] + ... + arr[len-1]
     => sum == arr[0] + ... + arr[arr.length-1] // by precondition

Does loop terminate?

Define D = len – i // initially i == 0 and len >= 0, D >= 0

Loop can be rewritten:

```
while((len - i) > 0) {          // i.e. while(D > 0)
   sum = sum + arr[i];
   i = i +1;                    // D_new = len – (i + 1) = (len - i) - 1 = D - 1
}
```

D decreases by 1 with each step.
D eventually reaches 0.
D == 0 => loop exit condition i == len
When D == 0, loop exits

What A Loop Invariant Is Not

A loop invariant **is not** just some statement that is true before, during, and after the loop.
It must be effective.

LI && exit condition => postcondition

For example,

```
// precondition: x > 0
x = 10;
y = 0;
z = 42;  // LI: z == 42, D = x

while(x > 0) {
   x = x – 1;
   y = y + 1;
}
// postcondition: y == x
```

z is always 42, but it has nothing to do with the loop. It is not a valid or useful loop invariant.
Exit condition and LI do not imply postcondition

What is the LI?

Precondition: x >= 0 && y == 0

```
while(x != y) {
    y = y + 1
}
```

Postcondition x == y

Assume ints.
Since initially x >= 0 && y == 0 we can rewrite the loop:
D = x - y > 0

```
while((x-y) > 0) {
   y = y + 1
}
```

At the end D == 0 => x - y == 0 => x == y

Initially, x >= 0 && y == 0 => x >= y (good guess?)
We want to show by induction:
Assume at iteration k: x >= y_k:
but if x == y_k, we would exit so x > y_k
   y_new = y_k + 1
   x > y_k => x >= y_new

LI: x >= y

Check the LI

Precondition: x >= 0 && y = 0

```
while(x != y) {
    y = y + 1
}
```

Postcondition x == y

LI: x >= y

Base case:
x >= 0 && y == 0 => x >= y

Assume: x >= y holds at iteration k
If x == y at iteration k, we would exit loop
x > y at iteration k
y_new = y + 1
x >= y_new

At exit: !(x != y) && x >= y => x == y

D = x - y
D_new = x - (y + 1) = D - 1 // D decreases at each iteration
D = 0 => x == y

Example

Assume ints
PRECONDITION: n >= 0

```
i = 0
r = 1

while ( i < n ) {
  i = i + 1
  r = r * i
}
```

POSTCONDITION: r == n!
what is the LI here?

```
PRECONDITION: n >= 0

i = 0
r = 1

while ( i < n ) {
  i = i + 1
  r = r * i
}
POSTCONDITION: r==n!
```

POSTCONDITION: r == n!
D = n - i
LI: r == i! && i <= n

show the above to be true in terms of Partial Correctness

```
BASE CASE: i == 0 and r == 1
    (r == i!) = (r == 0!) = (1 == 0!)
    (i <= n) = (0 <= n)  // precondition
    both parts of LI hold

INDUCTIVE CASE:
      assume: r_old == i_old!
      i_new = i_old + 1

      // assume r_old == i_old!
      r_new = r_old * i_new
      r_new = (i_new-1)! * i_new = i_new!
      r_new = i_new!

      i_old <= n; if i_old == n, we would have exited
      i_old < n
      i_new = i_old + 1 <= n

AT EXIT: !(i < n) && (i <= n && r == i!)
        => i == n && r == i!
        => r == n!
```

```
Initially D >= 0
D: n – i
D_old = n – i_old
i_new = i_old + 1
D_new = n – i_new
      = n – (i_old + 1)
      = D_old – 1

D == 0 => i == n // loop exit condition
```

Termination

Termination, a little more formally...

-- We need to find a decrementing function D

{ P } while (b) S { Q }

We need D such that

```
(1) { LI && b } S { D_after < D_before }   // One iteration of the loop
                                              reduces the value of D

(2) D==min => exit condition
```

Note: In this case, if 0 is D's minimal value and
must imply the loop exit condition.
You can replace b with D > 0

Total correctness = Partial correctness + Loop termination

- Establish that the loop terminates
- Suppose the loop always reduces some variable's value
  - Does the loop terminate if the variable is a
    - Natural number
    - Integer
    - Non-negative real
    - Boolean
    - List or Array
  - Loop terminates if the variable values are a subset of a well-ordered set and D decreases with each iteration
    - For an ordered set, every non-empty subset has a least element

Decrementing Function

- Decrementing function maps program variables to some well-ordered set

```
// precondition: x ≥ 0 && y == 0
// Loop invariant: x ≥ y
// D: (x - y)
while (x != y) {
    y = y + 1;
}
// postcondition: x == y
```

- Is x - y a good decrementing function?

Decrementing Function

- Does the loop reduce the decrementing function's value?

$$\begin{aligned} D_k &= x - y_k \\ y_{k+1} &= y_k + 1 \\ D_{k+1} &= x - y_{k+1} \\ &= x - (y_k + 1) \\ &= D_k - 1 \end{aligned}$$

- If the function is at a minimum does the loop exit?

$$D == 0 => x - y == 0 => x == y => !(x != y)$$

Example

```
PRECONDITION: x >= 0

i = x
z = 0

{ LOOP INVARIANT (LI): i + z == x }
while ( i > 0 ) {
  z = z + 1;
  i = i - 1;
}

POSTCONDITION: x == z
```

a decrementing function D is D = i

Exercise

```
precondition: arr.length == len  &&  len >= 0

 int sum = 0;
 int i = 0;
 while ( i < len ) {
   sum = sum + arr[i];
   i = i + 1;
 }

 postcondition: (result is the sum of all elements on array arr)
                sum == arr[0] + arr[1] + ... + arr[arr.length-1]

 D = len - i
```

Exercise

```
PRECONDITION: x > 0

zeros = 0;
y = x;

while ( y % 10 == 0 ) {
  y = y / 10   // integer division
  zeros = zeros + 1
}
```

POSTCONDITION: $x == y * 10^{zeros}$ && (y % 10 != 0)

Exercise

```
PRECONDITION: x1 > 0 && x2 > 0

y1 = x1
y2 = x2

while ( y1 != y2 ) {
  if ( y1 > y2 ) {
    y1 = y1 - y2
  }
  else {
    y2 = y2 - y1
  }
}

POSTCONDITION: y1 == gcd( x1, x2 )
```

Loops - Summary

Total correctness = Partial correctness + Loop termination

(1) Partial correctness

-- "Guess" then prove the loop invariant (LI) by induction

-- Loop invariant and the loop exit condition must imply
the given postcondition

-- This gives us:

"If the loop terminates, then the postcondition holds."

(2) Loop termination

-- "Guess" the decrementing function D.
Each iteration of the loop decrements D, until D reaches a minimum.
D at min must imply loop exit condition

Rules for Backward Reasoning: Method Call

// precondition: ??

```
x = foo()
```

// postcondition: Q

If method has no side-effects, just like assignment

// precondition: ??

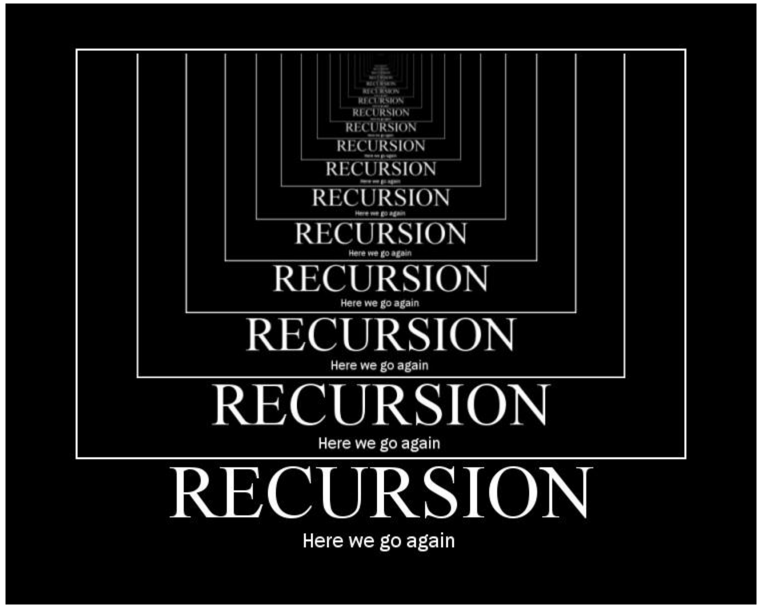
```
x = Math.abs(y)
```

// postcondition: x == 1

Precondition is y == 1 || y == -1

Recursion

- An effective recursive routine must
  - Have a base case
  - Assume algorithm is valid for step k
  - Show how to get from step k to step k + 1
  - Show that algorithm terminates
    - Recurses towards base case
- Sounds like computational induction



Example

```
// precondition: x > 0
// post condition: returns x!

int factorial(int x) {
       if(x == 1) {  // base case
           return 1;
       } else {
           return x * factorial(x-1);
       }
}
```

Invariant:
factorial(x) == x! && x>= 1

Base case:
1! == 1

Induction:
Assume factorial(y) = y! for y < x
factorial(x) == x * factorial(x-1) == x * (x-1)! == x!

Termination:
D = x - 1; x decreases at each iteration
D == 0 && factorial(x) == x! && x>= 1 => x==1

Summary So Far

- Intro to reasoning about code. Concepts
  - Specifications, preconditions and postconditions, forward and backward reasoning

- Hoare triples
- Rules for backward reasoning
  - Rule for assignment
  - Rule for sequence of statements
  - Rule for if-then-else

In Practice

- Write loop invariants when unsure about a loop
- When you have evidence that a loop is not working
  - Add invariant and decrementing function
  - Write code to check them
  - Understand why the code doesn't work
  - Fix
  - Reason to ensure that no similar bugs remain

In Practice

- Use the loop invariant to guide writing the loop
  - Determine the set of variables for the loop
  - Express the required condition at the end of the loop
    - Postcondition for the loop
  - Determine what holds before the loop executes
    - Precondition
  - Determine a decrementing function
    - What decreases with each iteration
    - Try to find a decrementing function with 0 as a minimum
  - Construct a loop invariant
    - What has to be true after each iteration
  - Use the loop invariant to construct the loop body

Why Do We Care?

- Correctness is important
  - Bugs are frustrating, expensive, and in some cases dangerous
- Pre and postconditions for functions are specifications
- Optimizing compilers
  - Transform loops
  - Is the transformed loop the same as the original
- Thinking about code in a formal way leads to better code
  - Helps us solve problems
  - Helps us create code from specifications