CS1110 Lecture 16, 26 Oct 2010 **While-loops**

Reading for next time: Ch. 8.1-8.3 (arrays)

Prelim 2: Tu Nov 9th, 7:30-9pm.
Last name A-*Lewis*: Olin 155
Last name *Li*-Z: Olin 255
Conflicts? Submit CMS "assignment" "P2 conflicts" by today.
Review session: Sun Nov 7th, 1-3pm, Phillips 101. (*Set your clocks back the night before!)*

Reminder: A5 due Sat. Oct 30th. See assignments pg for hints on snowflake geometry.

A mystery (due to von Neumann): suppose you have a biased coin with *unknown* probability of heads p, $0 < p < 1$.

How can you use this coin to simulate the output of a *fair* coin?

(Answer: while-loops …)

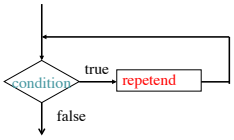
1

Beyond ranges of integers: the while loop

while (<*condition*>) {
  sequence of statements
}

boolean expression. = "there's still stuff to do"

the <*repetend*>



In comparison to for-loops: we get a broader notion of "there's still stuff to do" (not tied to integer ranges).

But *we* must ensure that "condition" stops holding, since there's no explicit increment.

2

Canonical while loops

```
// Process b..c
for (int k= b; k <= c; k= k+1) {
    Process k;
}
```

Scope note: since k happened to be declared "within" the loop, k can't be used after the loop.

```
// Process b..c
int k= b;
while (k <= c) {
    Process k;
    k= k+1;
}
```

Here's one way to use the while loop

```
/* process a sequence of inputs where you don't
     know how many need to be taken care of */
<initialization>;
while (<still input to deal with>) {
    Process next piece of input;
    make ready for the next piece of input;
}
```

3

Interesting while loops (showing why they can be hard to understand)

```
/** Von Neumann's "fair coin" from
    unfair coin, assuming heads prob
    not 0 or 1. Encode heads/tails as
    true/false.
= "output is heads". */
public static boolean fairFlip() {
  while (true) { // loop "forever"…
    boolean f1= new unfair flip;
    boolean f2= new unfair flip;
    if (f1 &&   !f2) { // HT
      return true; // escape the loop
    } else if (!f1  &&  f2) { // TH
      return false;
    }
  }
}
```

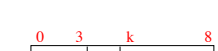
```
/** open question in mathematics: is
    there an n such that this function
    never returns a value (i.e., n
    doesn't "return to 1")?
 Precondition: n >=1.  */
public static boolean collatz(int n) {
    while (n != 1) {
        if (n%2 == 0) {
                n= n/2;
        }
        else {
                n= 3*n +1;
        }
    }
    return true;
}
```

4

How to analyze loops: understanding assertions about lists

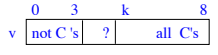
	0	1	2	3	4	5	6	7	8
v	X	Y	Z	X	A	D	C	C	C

v is (the name of) a list of Characters. (We aren't showing v as a variable to save space.)



k 6

This is an *assertion* about v and k, thus *explaining the meaning of these variables*. It is **true** because v[0..3] are not 'C' and v[6..8] are 'C's.

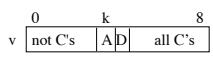


k 5

This **falsely** asserts that v[0..3] aren't C's, v[5..8] are 'C's.

k 0

True assertion that v[0..-1] aren't C's (nothing in the empty list is a C)



k 4

True assertion: v[0..3] are not C's, v[4] is A, v[5] is D, v[6..8] are C's.)

5

Counting characters. Store in n the number of '/'s in string s.

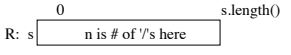
```
// Store in n to truthify diagram R

k= 0; n= 0;

// inv: See diagram P, below

while (   k!= s.length()      ) {
    if (s.charAt(k)=='/') {n= n + 1;}
    k= k + 1;
}
```

1. How does it start? ((how) does init. make inv true?)

2. When does it stop? (From the invariant and the falsity of loop condition, deduce that result holds.)

3. (How) does it make progress toward termination?

4. How does repetend keep invariant true?



6

Suppose we are thinking of this while loop:

```
initialization;
while ( B ) {
   repetend
}
```

We add the *postcondition* and also show where the *invariant* must be true:

```
initialization;
// invariant: P
while ( B ) {
   // { P and B}
   repetend
  // { P }
}
// { P and !B }
// { Result R }
```

The four loopy questions

Second box helps us develop four loopy questions for developing or understanding a loop:

1. How does loop start? Initialization must truthify invariant P.

2. When does loop stop?

At end, P and !B are true, and these must imply R. Find !B that satisfies P && !B => R.

3. Make progress toward termination? Put something in repetend to ensure this.

4. How to keep invariant true? Put something in repetend to ensure this.

Linear search. Character c is in String s. Find its first position.

```
// Store in k to truthify diagram R

k= 0;

// inv: See diagram P, below

while (    s.charAt(k) != c    ) {

    k= k + 1;

}
```

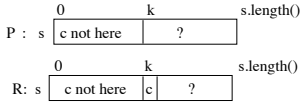
Idea: Start at beginning of s, looking for c; stop when found. How to express as an invariant?

1. How does it start? ((how) does init. make inv true?)

2. When does it stop? (From the invariant and the falsity of loop condition, deduce that result holds.)

3. (How) does it make progress toward termination?

4. How does repetend keep invariant true?



Appendix examples: Develop loop to store in x the sum of 1..100.

We'll keep this definition of x and k true:
x = sum of 1..k–1

Four loopy questions

1. How should the loop start? Make range 1..k–1 empty: **k= 1; x= 0;**

2. When can loop stop? What condition lets us know that x has desired result? When **k == 101**

3. How can repetend make progress toward termination? k= k+1;

4. How do we keep def of x and k true? x= x + k;

```
k= 1;  x= 0;
// invariant: x = sum of 1..(k–1)
while ( k != 101) {
    x=  x + k;
    k= k + 1;
}
// { x = sum of 1..100 }
```


Roach infestation

```
/** = number of weeks it takes roaches to fill the apartment --see p 244 of text*/
public static int roaches() {
    double roachVol= .001;    // Space one roach takes
    double aptVol= 20*20*8;  // Apartment volume
    double growthRate= 1.25; // Population growth rate per week

    int w= 0;       // number of weeks
    int pop= 100; // roach population after w weeks

    // inv: pop = roach population after w weeks   AND
    //       before week w, volume of the roaches < aptVol
    while (aptVol > pop * roachVol ) {
      pop= (int) (pop * growthRate);
      w= w + 1;
    }
    return w;
  }
```


Iterative version of logarithmic algorithm to calculate b**c (we've seen a recursive version before).

```
/** set z to b**c, given c ≥ 0 */
int x= b; int y= c; int z= 1;
// invariant:  z * x**y = b**c  and 0 ≤ y ≤ c
while (y != 0) {
    if (y % 2 == 0)
            { x= x * x; y= y/2;  }
    else { z= z * x; y= y – 1; }
}
// { z = b**c }
```


Calculate quotient and remainder when dividing x by y

x/y = q + r/y **21/4= 4 + 3/4**

Property: x = q * y + r and 0 ≤ r < y

```
/** Set q to quotient  and r to remainder.
    Note: x >= 0 and y > 0 */
int q= 0; int r= x;
// invariant:  x = q * y + r    and 0 ≤ r
while (r >= y) {
        r= r – y;
        q= q + 1;
}
// { x = q * y + r    and   0 ≤ r < y }
```