

CSC 221: Computer Programming I

Fall 2001

- conditional repetition
 - repetition for: simulations, input verification, input sequences, ...
 - while loops
 - infinite (black hole) loops
- counters & sums
 - increment/decrement operators, arithmetic assignments
- for loop & other variants
 - counter-driven loops, for loops
 - other variants: do-while, break, continue

conditional repetition

if statement allows for conditional execution

- either execute a block of statements, or don't (i.e., execute 1 or 0 times)

many applications require *repeated* execution of statements

- *roll a pair of dice 10 times*
- *roll a pair of dice until you get doubles*
- *read user responses until "yes" or "no" is entered*
- *fold a piece of paper until its thickness reaches the sun*
- *read an arbitrary number of grades and average them*
- *search for the first vowel in a word (for Pig Latin)*

while loops

many different words/phrases in English suggest repetition

- repeatedly do X as long as Y
- while Y holds, do X
- do X, Y number of times
- keep doing X until Y happens
- ...

in all these forms, repetition is controlled by a condition

conditional repetition in C++ is accomplished via a while loop

```
while (BOOLEAN_TEST) { // while TEST evaluates to true
    STATEMENTS;        // execute statements inside { }
}                       // and return to top to repeat
                       // when TEST becomes false, exit loop
```

Note similarity to if statement:

- both are controlled by a Boolean test: if the test succeeds, then the body is executed
- for a while loop, however, control returns back to the top and the test is reevaluated

rolling along...

consider the simulation of repeated dice rolls

in pseudo-code:

```
while (HAVEN'T ROLLED 10 TIMES){
    ROLL DICE;
    DISPLAY RESULTS;
}
```

RECALL: the Die class encapsulates several useful member functions:

- Roll()
- NumRolls()

```
// rollem.cpp
//
// Simulates rolling two dice
////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 10;

int main()
{
    Die d1, d2;

    while (d1.NumRolls() < NUM_ROLLS) {
        int roll1 = d1.Roll();
        int roll2 = d2.Roll();

        cout << "You rolled: " << roll1
              << " and " << roll2 << endl;
    }

    return 0;
}
```

Note: the use of the constant NUM_ROLLS makes it easy to change the number of repetitions

double time (cont.)

can avoid redundancy with a **KLUDGE** (a quick-and-dirty trick for making code work)

- only roll the dice inside the loop
- initialize the roll variables so that the loop test succeeds the first time
- after the first kludgy time, the loop behaves as before

```
// doubles.cpp
//
// Simulates rolling two dice until doubles.
///////////////////////////////////////////////////////////////////
#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    Die d1, d2;

    int roll1 = -1; // KLUDGE: initializes rolls
    int roll2 = -2; // so that loop is entered

    while (roll1 != roll2) {
        roll1 = d1.Roll();
        roll2 = d2.Roll();
        cout << "You rolled " << roll1
              << " and " << roll2 << endl;
    }

    cout << "It took " << d1.NumRolls() << " rolls."
          << endl;

    return 0;
}
```

in general, avoid kludges.

here, the benefit of removing redundant code is worth it.

loops for input verification

a common use of loops is to verify user input

- in HW3, you prompted the user for an option (1-2, or 1-6)
- if the input was not in the specified range, the program terminated
- wouldn't it be nicer to re-prompt the user on invalid input?

at first glance, an if statement might seem sufficient

pseudo-code:

```
READ USER OPTION;

if (OPTION NOT VALID) {
    DISPLAY ERROR MESSAGE;
    REREAD USER OPTION;
}
```

```
int userOption;
cout << "Enter your option (1 - 6): ";
cin >> userOption;

if (userOption < 1 || userOption > 6) {
    cout << "That was an invalid response." << endl;
    cout << "Enter your option (1 - 6): ";
    cin >> userOption;
}

. . .
```

loops for input verification (cont.)

an if statement works fine as long as the user only messes up once

if it is possible that they could mess up an arbitrary number of times,
then you need a loop to keep reading inputs until valid

pseudo-code:

```
READ USER OPTION;  
  
while (OPTION NOT VALID) {  
    DISPLAY ERROR MESSAGE;  
    REREAD USER OPTION;  
}
```

```
int userOption;  
cout << "Enter your option (1 - 6): ";  
cin >> userOption;  
  
while (userOption < 1 || userOption > 6) {  
    cout << "That was an invalid response." << endl;  
    cout << "Enter your option (1 - 6): ";  
    cin >> userOption;  
}  
  
. . .
```

Note: this code is identical to the previous page except "while" replaces "if"
this highlights the similarity & difference between the two

generalizing input verification

of course, we could
generalize the task for easier
reuse (i.e., functions)

```
int age;  
cout << "Enter your age: ";  
age = ReadIntInRange(0, 120);
```

```
int year;  
cout << "In what year were you born? ";  
age = ReadIntInRange(1880, 2001);
```

```
char response;  
cout << "Do you want to play again? ";  
response = ReadYorN();
```

```
int ReadIntInRange(int low, int high)  
// Assumes: low <= high  
// Returns: user input in range low..high  
{  
    int userOption;  
    cin >> userOption;  
  
    while (userOption < low || userOption > high){  
        cout << "Invalid response "  
            << "(must be between " << low  
            << " and " << high << "): ";  
        cin >> userOption;  
    }  
    return userOption;  
}
```

```
char ReadYorN()  
// Returns: user input, either 'y' or 'n'  
{  
    char userChar;  
    cin >> userChar;  
  
    userChar = tolower(userChar);  
    while (userChar != 'y' && userChar != 'n'){  
        cout << "Invalid response "  
            << "(must be either 'y' or 'n')": ";  
        cin >> userChar;  
    }  
    return userChar;  
}
```

"hailstone sequence"

interesting sequence from mathematics

- start with a positive integer
- if that number is odd, triple it and add 1
- if that number is even, divide it in half
- then repeat

5 → 16 → 8 → 4 → 2 → 1 → 4 → 2 → 1 → ...

11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1 → ...

24 → 12 → 6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1 → ...

150 → 75 → 226 → 113 → 340 → 170 → 85 → 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1 → ...

It has been conjectured that every sequence gets caught in the 4→2→1 cycle.

The conjecture has been verified for all starting numbers $\leq 60,000,000$, but still has not been proven to hold for all starting numbers.

hailstone code

can generate any hailstone sequence using a while loop

```
READ STARTING NUM FROM USER;

while (NOT STUCK IN LOOP) {
    UPDATE NUMBER;
    DISPLAY NEW NUMBER;
}

DISPLAY THE FACT THAT STUCK;
```

HOMEWORK 4: complete the program (with input verification)

```
Enter the starting number of a hailstone sequence:
0
The number must be positive. Try again:
-5
The number must be positive. Try again:
5
16
8
4
2
1
STUCK!
```

in-class exercise

what output would be produced by the following loops?

```
int x = -5;
while (x > 0) {
    cout << x << endl;
}
```

```
int x = 5;
while (x > 0) {
    cout << x << endl;
}
```

```
int a = 1, b = 10;
while (a < b) {
    a = a + 1;
    b = b - 2;

    cout << a << ", " << b << endl;
}
```

```
int num = 1024;
while (num > 0) {
    num = num - 1;
}
cout << num << endl;
```

infinite loops

if the hailstone conjecture is false, then there is an infinite sequence!

using `ReadIntInRange`, the user could enter invalid info forever!

→ infinite loops (loops whose tests are never false) are possible

infinite loops more commonly occur as a result of programmer error

```
int ReadIntInRange(int low, int high)
// Assumes: low <= high
// Returns: user input in range low..high
{
    int userOption;
    cin >> userOption;

    while (userOption < low || userOption > high){
        cout << "Invalid response "
            << "(must be between " << low
            << " and " << high << "): ";
    }
    return userOption;
}
```

what is wrong/missing here?

a.k.a. black hole loops

- loop test is like the event horizon
- once past it, there is no escape

avoiding infinite loops

```
int roll1 = -1;
int roll2 = -2;

while (roll1 + roll2 != 7) {
    if (roll1 == roll2) {
        roll1 = d1.Roll();
        roll2 = d2.Roll();
    }

    cout << "You rolled: " << (roll1 + roll2)
         << endl;
}
```

what is wrong here?

in these examples, programmer error made it so that *nothing* changed inside the loop – GUARANTEED BLACK HOLE!

must ensure that *something* affecting the loop test changes inside the loop – IS THIS ALWAYS ENOUGH?

counters

while loops allow for repeated simulations/experiments

- often, want to keep track of some event that occurs during those repetitions

e.g., roll dice and keep track of the number of 7's
generate a hailstone sequence and keep track of its length
paper folding puzzle

a *counter* is a variable that is used to keep count of some "event"

- it is not a new feature of C++, just a particular pattern of use

```
int count = 0; // must initialize counter to 0
while (NOT DONE) {
    PERFORM TASK;

    if (EVENT OCCURRED) { // when event occurs,
        count = count + 1; // increment counter
    }
}
```

dice stats

can use a counter to keep track of the number of sevens rolled

```
INITIALIZE 7-COUNT TO 0;

while (HAVEN'T FINISHED ROLLING) {
    ROLL DICE;
    IF ROLLED 7, UPDATE 7-COUNT;
}

DISPLAY 7-COUNT;
```

what would happen if we forgot to initialize `sevenCount`?

what would happen if we initialized inside the loop?

```
// stats.cpp
//
// Simulates rolling two dice, counts 7's.
///////////////////////////////////////////////////////////////////

#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 1000;

int main()
{
    Die d1, d2;

    int sevenCount = 0;

    while (d1.NumRolls() < NUM_ROLLS) {
        int roll1 = d1.Roll();
        int roll2 = d2.Roll();

        if (roll1 + roll2 == 7) {
            sevenCount = sevenCount + 1;
        }
    }

    cout << "Out of " << NUM_ROLLS << " rolls, "
         << sevenCount << " were sevens." << endl;

    return 0;
}
```

shorthand notation

counters are fairly common tools

- C++ provides shorthand notation for incrementing/decrementing

```
x = x + 1;   ≡   x += 1;   ≡   x++;
```

```
y = y - 1;   ≡   y -= 1;   ≡   y--;
```

- in fact, all the arithmetic operators can be combined with assignments

```
a = a * 2;   ≡   a *= 2;
```

```
b = b / 10;  ≡   b /= 10;
```

ADVICE: use `++` notation for incrementing (shorter & more obvious)
use other arithmetic assignments as you find comfortable

dice stats++

use the increment operator ++
to increment sevenCount

note: each Die object is
keeping its own counter of the
number of rolls

- when you create a Die, the counter is initialized to 0
- when you call Roll, the counter is incremented
- when you call NumRolls, the counter value is returned

abstraction hides the details!

```
// stats.cpp
//
// Simulates rolling two dice, counts 7's.
//
#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 1000;

int main()
{
    Die d1, d2;

    int sevenCount = 0;

    while (d1.NumRolls() < NUM_ROLLS) {
        int roll1 = d1.Roll();
        int roll2 = d2.Roll();

        if (roll1 + roll2 == 7) {
            sevenCount++;
        }
    }

    cout << "Out of " << NUM_ROLLS << " rolls, "
         << sevenCount << " were sevens." << endl;

    return 0;
}
```

generalized stats

again, can generalize with a
function to make variations
easier

have a single CountDice
function with parameters so
that it works for

- arbitrary # of dice rolls
- arbitrary dice total

```
#include <iostream>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 1000;

int CountDice(int numRolls, int desiredTotal);

int main()
{
    int userChoice;
    cout << "What dice total are you interested in? ";
    cin >> userChoice;

    cout << "Out of " << NUM_ROLLS << " rolls, "
         << CountDice(NUM_ROLLS, userChoice) << " were "
         << userChoice << "'s." << endl;

    return 0;
}

//
//
//
int CountDice(int numRolls, int desiredTotal)
// Assumes: numRolls >= 0, 2 <= desiredTotal <= 12
// Returns: number of times desiredTotal occurred out
// of numRolls simulated Dice rolls
{
    Die d1, d2;

    int desiredCount = 0;
    while (d1.NumRolls() < numRolls) {
        int roll1 = d1.Roll();
        int roll2 = d2.Roll();

        if (roll1 + roll2 == desiredTotal) {
            desiredCount++;
        }
    }

    return desiredCount;
}
```

paper folding puzzle

if you started with a regular sheet of paper and folded it in half, then folded that in half, then folded that in half, ...

how many folds would it take for the thickness of the paper to reach the sun?

pseudo-code:

```
INITIALIZE PAPER THICKNESS;  
INITIALIZE FOLD COUNT TO 0;  
  
while (THICKNESS LESS THAN DISTANCE TO SUN) {  
    DOUBLE THICKNESS;  
    INCREMENT FOLD COUNT;  
}  
  
DISPLAY FOLD COUNT;
```

distance to the sun?

thickness of a piece of paper?

paper folding code

on average, distance to the sun is 93,300,000 miles

pack of 500 sheets is ~1 inch
→ single sheet is 0.002 inches

LESSON: when you keep doubling something, it grows VERY fast!

```
// fold.cpp  
//  
// Simulates the paper folding puzzle.  
////////////////////////////////////  
  
#include <iostream>  
using namespace std;  
  
const double PAPER_THICKNESS = 0.002;  
const double DISTANCE_TO_SUN = 93.3e6*5280*12;  
  
int main()  
{  
    double thick = PAPER_THICKNESS;  
    int foldCount = 0;  
  
    while (thick < DISTANCE_TO_SUN) {  
        thick *= 2;  
        foldCount++;  
    }  
  
    cout << "It took " << foldCount << " folds."  
        << endl;  
  
    return 0;  
}
```

sums

other applications call for accumulating values

- similar to a counter, but not just incrementing – must add a value to the sum

e.g., compute the average of a sequence of grades

requires a sum to accumulate the sum of the grades

requires a counter to keep track of how many grades

```
INITIALIZE COUNTER & SUM;
READ FIRST GRADE;

while (STILL MORE GRADES) {
  ADD GRADE TO SUM;
  INCREMENT COUNTER;

  READ NEXT GRADE;
}

COMPUTE AVERAGE = SUM/COUNTER;
```

QUESTION: how can we know when there are no more grades to read in?

averaging grades

a *sentinel value* is a special value that marks the end of input

must be a value that can't naturally appear (e.g., a grade of -1)

must be careful not to treat the sentinel value as data

```
// avg.cpp
//
// Generates a hailstone sequence starting with an input.
///////////////////////////////////////////////////////////////////
#include <iostream>
using namespace std;

int main()
{
  int numGrades = 0, gradeSum = 0;

  int grade;
  cout << "Enter grades (terminate with -1):" << endl;
  cin >> grade;

  while (grade != -1) {
    gradeSum += grade;
    numGrades++;

    cin >> grade;
  }

  double avg = (double)gradeSum/numGrades;
  cout << "Your average is " << avg << endl;

  return 0;
}
```

dot race example

common occurrence at sporting events: big-screen dot races

- some number of colored dots race around a track
- movement is either random or else driven by crowd noise

we can simulate a dot race

- to keep things simple, we will have only 2 dots (but could easily expand)
- at each time interval, a dot moves forward between 1 and 3 steps
- the race ends when one or both dots cross the finish line

```
PLACE DOTS AT START LINE;

while (NEITHER HAS CROSSED FINISH LINE) {
    MOVE EACH DOT A RANDOM AMOUNT;
    DISPLAY POSITION;
}

IDENTIFY THE WINNER(S);
```

how do we keep track of dot positions?
how do we know when cross finish?
how do we determine how far to move?
how do we identify the winner(s)?

dot race code

represent dot position as a number

- start line is 0
- finish line is a const

race continues if neither has reached the finish line

use a 3-sided Die object to generate the step amount

must be careful in identifying the winner – both could cross the finish line at the same time

```
// dotrace.cpp
//
// Simulates a race between two dots.
///////////////////////////////////////////////////////////////////

#include <iostream>
#include <iomanip>
#include "Die.h"
using namespace std;

const int FINISH_LINE = 20;

int main()
{
    Die die3(3);

    int position1 = 0, position2 = 0;

    while (position1 < FINISH_LINE && position2 < FINISH_LINE) {
        position1 += die3.Roll();
        position2 += die3.Roll();

        cout << "Dot 1: " << setw(2) << position1 << " "
              << "Dot 2: " << setw(2) << position2 << endl;
    }

    if (position2 < FINISH_LINE) {
        cout << "Dot 1 wins!" << endl;
    }
    else if (position1 < FINISH_LINE) {
        cout << "Dot 2 wins!" << endl;
    }
    else {
        cout << "It's a tie!" << endl;
    }

    return 0;
}
```

for loops

counter-driven loops (*do something X times*) are fairly common

```
int repCount = 0;
while (repCount < NUM_REPS) {
    DO SOMETHING;

    repCount++;
}
```

a for loop more naturally encapsulates this type of behavior

```
for (int repCount = 0; repCount < NUM_REPS; repCount++) {
    DO SOMETHING;
}
```

- this for loop is identical to the while loop above
- more concise, all loop control info is written on the same line
- less likely to forget a piece (e.g., forgetting to increment → infinite loop)

while loops vs. for loops

while loop version:

```
int numTimes = 0;
while (numTimes < 10) {
    cout << "Howdy" << endl;
    numTimes++;
}
```

```
int i = 1, sum = 0;
while (i <= 100) {
    sum += i;
    i++;
}
```

for loop version:

```
for (int numTimes = 0; numTimes < 10; numTimes++) {
    cout << "Howdy" << endl;
}
```

```
int sum = 0;
for (i = 1; i <= 100; i++) {
    sum += i;
}
```

use for loop when you know the number of repetitions ahead of time

use while loop when the number of repetitions is unpredictable

complete stats (badly done)

```
#include <iostream>
#include <iomanip>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 1000;

int CountDice(int numRolls, int desiredTotal);

int main()
{
    cout << "Out of " << NUM_ROLLS << " rolls: " << endl;
    << setw(4) << CountDice(NUM_ROLLS, 2) << " were 2's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 3) << " were 3's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 4) << " were 4's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 5) << " were 5's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 6) << " were 6's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 7) << " were 7's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 8) << " were 8's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 9) << " were 9's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 10) << " were 10's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 11) << " were 11's." << endl;
    << setw(4) << CountDice(NUM_ROLLS, 12) << " were 12s." << endl;

    return 0;
}

////////////////////////////////////

int CountDice(int numRolls, int desiredTotal)
{
    // AS BEFORE
}
```

if you wanted to see stats for all dice totals,
you could take brute force approach

complete stats (well done)

```
#include <iostream>
#include <iomanip>
#include "Die.h"
using namespace std;

const int NUM_ROLLS = 100;

int CountDice(int numRolls, int desiredTotal);

int main()
{
    cout << "Out of " << NUM_ROLLS << " rolls: " << endl;

    for (int roll = 2; roll <= 12; roll++) {
        cout << setw(4) << CountDice(NUM_ROLLS, roll) << " were "
            << setw(2) << roll << "s." << endl;
    }

    return 0;
}

////////////////////////////////////

int CountDice(int numRolls, int desiredTotal)
{
    // AS BEFORE
}
```

can generalize using a loop

- variable ranges from 2 through 12
- CountDice function called on each pass

consistent stats?

will the counts from `stats.cpp` necessarily add up to 1000?

```
Out of 1000 rolls:
24 were 2's.
48 were 3's.
96 were 4's.
100 were 5's.
125 were 6's.
159 were 7's.
136 were 8's.
88 were 9's.
84 were 10's.
45 were 11's.
24 were 12's.
```

note that each call to `CountDice` is a separate experiment (1000 rolls)

if want a single experiment to count all dice totals, need

- a single loop (1000 rolls)
- 11 different counters (for dice totals 2 through 12) ** *LATER* **

Pig Latin revisited

remember status of Pig Latin program

we needed to find the first occurrence of a vowel in the word

- using only `if` statements, this was *extremely* tedious
- using a loop, can traverse the word char-by-char until a vowel is found

```
int FindVowel(string str)
// Assumes: str is a single word (no spaces)
// Returns: the index of the first vowel in str, or
// string::npos if no vowel is found
{
    for (EACH CHAR IN str, STARTING AT INDEX 0) {
        if (CHAR IS A VOWEL) {
            RETURN ITS INDEX;
        }
    }
    OTHERWISE, RETURN NOT FOUND;
}
```

Pig Latin (final version)

```
#include <iostream>
#include <string>
using namespace std;

string PigLatin(string word);
bool IsVowel(char ch);
int FindVowel(string str);

int main()
{
    string word;
    cout << "Enter a word: ";
    cin >> word;

    cout << "That translates to: "
         << PigLatin(word) << endl;

    return 0;
}

////////////////////////////////////
bool IsVowel(char ch)
// Assumes: ch is a letter
// Returns: true if ch is a vowel
// ("aeiouAEIOU")
{
    const string VOWELS = "aeiouAEIOU";
    return VOWELS.find(ch) != string::npos;
}

string PigLatin(string word)
// Assumes: word is a single word (no spaces)
// Returns: the Pig Latin translation of the word
{
    int vowelIndex = FindVowel(word);

    if (vowelIndex == 0 || vowelIndex == string::npos) {
        return word + "way";
    }
    else {
        return word.substr(vowelIndex,
                           word.length()-vowelIndex) +
               word.substr(0, vowelIndex) + "ay";
    }
}

int FindVowel(string str)
// Assumes: str is a single word (no spaces)
// Returns: the index of the first vowel in str, or
// string::npos if no vowel is found
{
    for (int index = 0; index < str.length(); index++) {
        if (IsVowel(str.at(index))) {
            return index;
        }
    }

    return string::npos;
}
```

other loop variants

do-while loop

- similar to while loop, only the test occurs at the bottom of the loop

break statement

- can break out of a loop using the break statement

continue statement

- can interrupt a loop pass, return to the top of the loop using continue

FILE UNDER: handy, but not necessary

- we may revisit these in later examples, but maybe not
- don't worry about them for now (unless you want to)

repetition summary

while loops are used for conditional repetition

- can execute a block of code an arbitrary number of times (incl. 0 times)
- similar to if statement, looping is controlled by a Boolean test
- infinite (black hole) loops are a possibility -- usually caused by coding errors

counters and sums are variables used to accumulate values in a loop

- counter: initialized to 0, incremented every time desired event happens
- sum: initialized to 0, has running total added to it
- shorthand notation exists for updating variables: `x++;` `x += 1;` `x = x+1;`

loops can take different forms in C++

- for loop: useful for counter-driven repetition
 combines init-test-update info into a single line