Control Structures II (Repetition, Chpt 5)

Iteration (repetition)

- Iteration: process of repeating certain statements over and over again.
- Iteration avoids writing same statements again and again.

The need for iteration

- Add 5 numbers and find their average:

```
cin >> num1 >> num2 >> num3 >> num4 >> num5;
sum = num1 + num2 + num3 + num4 + num5;
average = static_cast<double>(sum)/5.0;
```

- This is doable but what if we want to do the same for 100 numbers?
- A real chore, messy and not desirable!
- There is a much easier and neater way!

Repetitive control structures

– Many algorithms require numerous iterations over the same statements. Eg.
  - To average 100 numbers, we would need 300 plus statements.
  - Or we use a statement that has the ability to loop or repeat a collection of statements.
– C++ has three repetition or looping structures:
  a) while loop
  b) do…while loop
  c) for loop
– The while and for loops are known as a pre-test loops.
– The do…while is known as a post-test loop.

a) Pre-test: **while Loop**

– General form of the while statement:

```
while ( loop-test )
{
  iterative-part
}
```

– loop-test is evaluated first. If true, do the iterative part, then do loop-test again.
– Process continues until the loop-test becomes false.
– Need a statement to make loop-test false. Otherwise we can't get out of while loop.

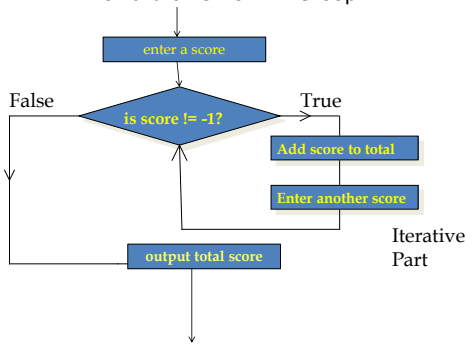
Pre-Test: While loop

Example: find total of some positive test scores.

```
int total = 0, score;
cout << "Enter test score (-1 to stop): ";
cin >> score;
while(score != -1)
{
  total = total + score;
  cout << "Enter score (-1 to stop): ";
  cin >> score;
}
cout << " The total score is " << total << endl;
```

Note1: There should always be a way for a loop to stop
Note2: The iterative parts may never be done. This is why it is called a pre-test
Note 3: This is a sentinel controlled loop (-1).

Flowchart view of **while** loop



b) Post-test: do…while

- General form of the do…while loop

```
do
{
   iterative part;
}
while(loop-test); // notice the semicolon.
```

- First time, program drops into the do portion with no loop-test.
- loop-test is evaluated after the first iteration.
- if loop-test is false, program drops out and continues.
- if true, returns to the first iterative statement after do.

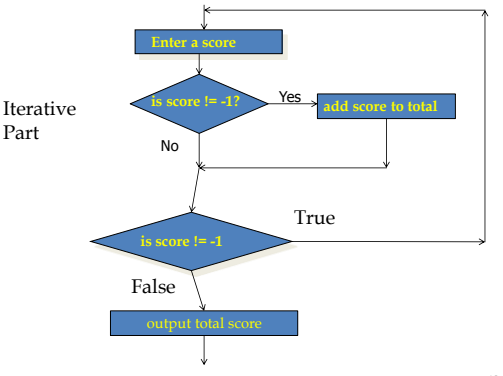
Post-test: do…while loop

Example: find the total of some positive test scores.

```
const int SENTINEL = -1;
int total = 0; int score;
do
{
    cout << "Enter score (-1 to stop): ";
    cin >> score;
    if(score >=0)
        total += score;
}
while (score != SENTINEL);
cout << " The total score is " << total << endl;
```

Note: Notice that the iterative part will be done at least once in a do-while loop.
Note: There is a semi-colon after the while.

Flowchart of do-**while** loop



`do…while` Looping structure (con't.)

EXAMPLE 5-18

```
i = 0;

do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
```

The output of this code is:

```
0 5 10 15 20
```

After 20 is output, the statement:

```
i = i + 5;
```

changes the value of i to 25 and so `i <= 20` becomes `false`, which halts the loop.

`do…while` Looping (Repetition) Structure (cont'd.)

EXAMPLE 5-19

Consider the following two loops:

```
a.  i = 11;
    while (i <= 10)
    {
        cout << i << " ";
        i = i + 5;
    }
    cout << endl;
b.  i = 11;
    do
    {
        cout << i << " ";
        i = i + 5;
    }
    while (i <= 10);

    cout << endl;
```

In (a), the `while` loop produces nothing. In (b), the `do...while` loop outputs the number 11 and also changes the value of `i` to 16.

When to use the **do-while** loop

– The do while loop is a good choice for obtaining interactive input from menu selections.
– Consider a function that won't stop executing until the user enters an N, O, or S:

Example **do-while** loop

```
char option;
do
{
   cout << "Enter N)ew, O)pen, S)ave: ";
   cin >> option;
   option = toupper(option);   // from cctype
   // if option is one of the choices do something accordingly.
}
while (option !='N' && option != 'O' &&  option != 'S');
```

(this loop continues only if an invalid option is entered. Eg. 'P')

which is better?

- in some instances you may want to use a while loop.
- in some instances you may want to use a do…while loop.
- you decide, there is no right answer.
- but you can see that they can do the same things as in the prior programs illustrating them.
- What are the trade offs?
  - Some while loop needs two prompts for data, no if statement.
  - Some do…while loops needs only one prompt for data but need an if statement inside iteration.
- some situations lend themselves better to while loops, some to do…while loops.
- practice helps.

Infinite loops

– ***Infinite loop:*** a loop that never terminates.
– you won't see your prompt again. Hit CTRL-C to get the prompt back.
– Infinite loops are usually not desirable unless you are running a clock or an ATM banking machine.
– Infinite loops are caused when the loop-test never gets set to false.

Example of an infinite loop

```
int cntr = 0;
while (cntr < 10)
{
   cout << "Hello there" << endl;
}
cout << "Do we ever get here?" << endl;
```

There is no step that brings (cntr<10) to false.
(Notice that there are parentheses around the single statement in this while loop. As with the if statement, there is no need for parentheses when there is only one executable statement.)

How do we fix the infinite loop? Add cntr++; statement below the cout.

```
while (cntr < 10)
{
   cout << "Hello there" << endl;
   cntr++;
}
```

Note: There are many ways to get into an infinite loop so you need to trace your loop steps when it happens.

computePay.cc

```
// FILE: computePay.cc
// COMPUTES THE PAYROLL FOR A COMPANY using a loop
   structure
#include <iostream>
using namespace std;
int main ()
{
  int numberEmp, empCount;
  double hours;
  double rate, pay, totalPay;

  // Get number of employees from user.
  cout << "Enter number of employees: ";
  cin >> numberEmp;

  // Compute each employee's pay and add it to the payroll.
  totalPay = 0.0; // set initial value
  empCount = 0;
```

computePay.cc

```
    while (empCount < numberEmp)
    {
        cout << "Hours: ";
        cin >> hours;
        cout << "Rate : $";
        cin >> rate;
        pay = hours * rate;
        cout << "Pay is "  << pay << endl << endl;
        totalPay += pay; //do you see why we need initial value?
        empCount++;
     }
    cout << "Total payroll is " << totalPay << endl;
    cout << "All employees processed." << endl;
    return 0;
}
```


computePay.cc

Program Output

Enter number of Employees: 3
Hours: **50**
Rate: **$15.25**
Pay is: **$762.50**
Hours: **6**
Rate: **$12.50**
Pay is: **$75**
Hours: **1.5**
Rate: **$9.25**
Pay is: **$13.88**
Total payroll is $**851.38**
All employees processed.

Sentinel vs flag controlled loop

- A sentinel is a special value that marks the end of a list of values
- A flag is a boolean variable that signals when a condition exists.

Sentinel Controlled

```
int sum = 0, num;
cout << "Enter positive number, -1 to quit: ";
cin >> num;
while(num != -1) // -1 signals the end of the list (sentinel)
{
    sum += num;
    cout << "Enter positive number, -1 to quit: ";
    cin >> num;
}
cout << "The sum is: " << sum << endl;
```


Flag controlled

```
// code to test if user enters a digit character.
char nextChar;
bool digitRead = false; // digitRead is the flag
while(!digitRead)    // while the flag is not true…repeat…
{
    cout << "Please enter a character: ";
    cin >> nextChar;
    digitRead = (nextChar >= '0' && nextChar <= '9');
}
cout << "The digit entered was: " << nextChar <<endl;
…
```


Loop Design and Loop Patterns

- sentinel-controlled loops
  1. Read the first data item.
  2. while the sentinel value has not been read
     3. Process the data item.
     4. Read the next data item.
- flag controlled loops
  1. Set the flag to false (or true)
  2. while the flag is false (or true)
     3. Perform some action.
     4. Reset the flag to true (or false) if the anticipated event occurred.
- eof controlled loops
  1. Read the first data item
  2. while the end-of-file has not been detected
     3. Process the data item
     4. Read the next data item

eof controlled `while` Loops

- end-of-file (eof)-controlled `while` loop: when it is difficult to select a sentinel value
- the eof character is CTRL-D on the key board (cin) and in files is automatically found at the end of file.
- The logical value returned by `cin` can determine if there is no more input.

`eof` Function

- The function `eof` can determine the end of file status.
- `eof` is a member of data type `istream`
- Syntax for the function `eof`:

```
istreamVar.eof()
```

- `istreamVar` is an input stream variable, such as `cin`

eof-controlled `while` loops (con't)

```
int cntr = 0, total=0, number;
double average;
cout << "Enter some numbers (CTRL-D) to quit: ";
cin >> number;
while(!cin.eof( ))
{
    total += number;
    cntr++;
    cin >> number; // need new number before eof() is tested again.
}
average = static_cast<double>(total)/cntr;
cout << "The total is: " << endl;
cout << "The average is: " << average << endl;
```


This program displays a list of numbers an their squares.

```
#include <iostream>
using namespace std;
const int MIN_NUMBER = 1;  const int MAX_NUMBER = 10;
int main( )
{   int num = MIN_NUMBER;
    cout << "Number     Squared\n";
    cout << "-------------------------\n";
    while(num <= MAX_NUMBER)
    {
        cout << "___" << num << "__________" << num * num << endl;
        num++;
    }
    return 0;
}
```

Output:

```
Number   Squared
-------------------------
  1         1
  2         4
  3         9
  ...
  10          100
```


sumScores.cc

```
// File: sumScores.cc
// Accumulates the sum of exam scores.
#include <iostream>
using namespace std;
const int SENTINEL = -1;
int main()
{
  int score, sum, count, average;
  // Process all exam scores until sentinel is read
  count = 0;
  sum = 0;
  cout << "Enter scores one at a time as  requested." << endl;
  cout << "When done, enter " << SENTINEL << " to stop." << endl;
  cout << "Enter the first  score: ";
  cin >> score;
```


sumScores.cc

```
    while (score != SENTINEL)
    {
        sum += score;
        count++;
      if(score >= 90)
                    cout << "\tGrade is: 'A'" << endl;
      else if(score >= 80) . . .
      cout << endl<< "Enter the next score: ";
        cin >> score;
    }
    cout << endl << endl;
    cout << "Number of scores processed is " <<  count << endl;
    cout << "Sum of exam scores is " << sum <<          endl;
    // Compute and display average score.
    if (count > 0)
    {
        average = sum / count;
        cout << "Average score is " << average;
    }
    return 0;
}
```

sumScores.cc

Program Output

```
Enter the scores one at a time as requested.
When done, enter -1 to stop.
Enter the first score: 55
  Grade is 'C'
Enter the next score: 33
  Grade is 'D'
Enter the next score: 77
  Grade is 'B'
Enter the next score: -1

Sum of exam scores is 165
3 exam scores were processed.
Average of the exam scores is 55.0
```



```
// FILE: largest.cc
// FINDS THE LARGEST NUMBER IN A SEQUENCE OF INTEGER VALUES
#include <iostream>   // needed for cin and cout
using namespace std;
const int MIN_VALUE = -32768; // smallest possible integer in given data

int main ()
{
  // Local data ...
  int itemValue;       // each data value
  int largestSoFar;    // largest value so far

  // Initialize largestSoFar to the smallest integer.
    largestSoFar = MIN_VALUE;
  do
  {
    cout << "Enter a integer, " << MIN_VALUE-1 << " to stop: ";
    cin >> itemValue;
    if (itemValue > largestSoFar)
      largestSoFar = itemValue;
  }
  while (itemValue != MIN_VALUE);
  cout << "The largest value entered was " << largestSoFar << endl;
  return 0;
}
```


Largest.cc

Program Output

```
Finding the largest value in a sequence:
Enter an integer or -32769 to stop: -999
Enter an integer or -32769 to stop: 500
Enter an integer or -32769 to stop: 100
Enter an integer or -32769 to stop: -32769
The largest value entered was 500
```


c) The **for** Statement

- The for loop is similar to the while loop.
- The for loop forces us to write, as part of the for loop:
  a) an initializing statement,
  b) the loop-test,
  c) a statement that is automatically repeated for each iteration, usually a variable increment stmt.
- a for loop is like a while loop in that it is a pre-test loop.

General form of a **for** loop

```
for( initial statement; loop-test;  repeated statement)
{
  iterative part
}
```

- When a for loop is encountered,
  a. the initial statement is executed.
  b. The loop-test is evaluated.
  c. If the loop-test is false, the for loop is terminated and exits to the point below the loop.
  d. otherwise the iterative part is executed.
  e. the repeated statement is executed.
  f. go to step b. and continue.

  Note the semicolons, not commas!

Eg: for-loop

- This for-loop is counter controlled.
- Scope of the loop control variable, cntr, is local within the "for" block.

```
for(int cntr=1; cntr<=5; cntr++)
{
      cout << cntr << "  ";
}
cout << endl;
```

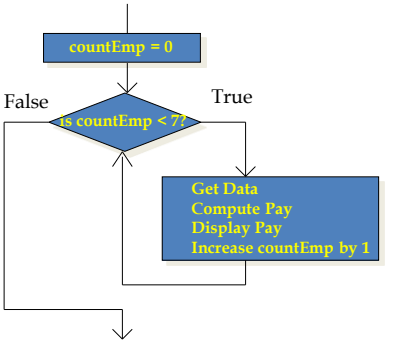
- Output: ?

Another for loop

```
for(int i=0; i<10; i++)
   cout << i << " squared = " << i * i  << endl;
```

Notice: there are no { } in the body of the for loop since there is only one statement.

Flow chart for a **for** loop



`for` looping (repetition) structure (cont'd.)

EXAMPLE 5-10

1. The following `for` loop outputs `Hello!` and a star (on separate lines) five times:

```
for (i = 1; i <= 5; i++)
{
    cout << "Hello!" << endl;
    cout << "*" << endl;
}
```

2. Consider the following `for` loop:

```
for (i = 1; i <= 5; i++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```

This loop outputs `Hello!` five times and the star only once.

`for` looping (repetition) structure (cont'd.)

- The following is a semantic error:

EXAMPLE 5-11

The following `for` loop executes five empty statements:

```
for (i = 0; i < 5; i++);      //Line 1
    cout << "*" << endl;      //Line 2
```

The semicolon at the end of the `for` statement (before the output statement, Line 1) terminates the `for` loop. The action of this `for` loop is empty, that is, null.

The following is a legal (but infinite) `for` loop:

```
for (;;)
   cout << "Hello" << endl;
```


`for` looping (repetition) structure (cont'd.)

EXAMPLE 5-12

You can count backward using a `for` loop if the `for` loop control expressions are set correctly.

For example, consider the following `for` loop:

```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

```
10 9 8 7 6 5 4 3 2 1
```

In this `for` loop, the variable `i` is initialized to 10. After each iteration of the loop, `i` is decremented by 1. The loop continues to execute as long as `i >= 1`.

`for` looping (repetition) structure (cont'd.)

EXAMPLE 5-13

You can increment (or decrement) the loop control variable by any fixed number. In the following `for` loop, the variable is initialized to 1; at the end of the `for` loop, `i` is incremented by 2. This `for` loop outputs the first 10 positive odd integers.

```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

Nested Loops

- It is possible to nest loops
  - Loops go inside other loops
  - Start with outer jump to inner loop
  - Inner loop continues until completed
  - Control goes back to outer loop and then inner loop starts all over again

Nested while-loop

```
outer = 1;
while(outer <= 3)
{
   int inner = 1; // Resets for each iteration of the outer loop
   while(inner <= outer)
   {
      cout << outer << "  " << inner << endl;
      inner = inner + 1;  // End of the nested loop
   }
   outer = outer + 1;  // Increment the outer loop counter
} // End of the outer loop
```

Program output

```
1 1
2 1
2 2
3 1
3 2
3 3
```


Nested for-loops

```
// FILE: nestedForLoops.cc
#include <iostream>
#include <iomanip>
int main ()
{
  // display heading
  cout << setw(12) << "i" << setw(6) << "j" << endl;
  for (int i = 0; i < 4; i++)
  {
    cout << "Outer" << setw (7) << i << endl;
    for (int j = 0; j < i; j++)
    cout << "  Inner" << setw (10) << j << endl;
  }
  return 0;
}
```


Program Output

```
                i          j
Outer           0
Outer           1
   Inner                   0
Outer           2
   Inner                   0
   Inner                   1
Outer           3
   Inner                   0
   Inner                   1
   Inner                   2
```


Nested Control Structures

- To create the following pattern:

```
*****
*****
*****
*****
*****
```

- We can use the following code:

```
for (i = 1; i <= 5 ; i++)
{
      for (j = 1; j <= 5; j++)
            cout << "*";
      cout << endl;
}
```


Conditional Loops

- When you don't know the number of iterations use a while or a do-while loop.
- When you know the number of iterations use a for loop. eg logarithm table or a table of powers.

Programming Example: Fibonacci Number

- Consider the following sequence of numbers:
  1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Called the Fibonacci sequence
- Given the first two numbers of the sequence (say, a1 and a2)
  - n^{th} number a_n, n >= 3, of this sequence is given by:
    $a_n = a_{n-1} + a_{n-2}$
- Write a program to output the first 10 fib seq. numbers given the first 2

Programming Example. Fibonacci Numbers algorithm

Problem: Find first 10 fibonacci numbers

- enter first two fib seq numbers into n1, n2
- repeat the following 4 steps 8 times:
- set temp = n1 + n2
- output temp;
- set n1 = n2;
- set n2 = temp;
- output "good bye"

A different Fibonacci version

- ask user which fibonacci number to get, n
- ask user for first two fib numbers, n1,n2
- repeat the following n steps n-2 times
- set temp = n1 + n2;
- set n1 = n2;
- set n2 = temp;
- cout << "the <n>th fib # is " << temp;

Choosing the Right Looping Structure

- All three loops have their place in C++
  - If you know or can determine in advance the number of repetitions needed, the `for` loop is the correct choice
  - If you do not know and cannot determine in advance the number of repetitions needed, and it could be zero, use a `while` loop
  - If you do not know and cannot determine in advance the number of repetitions needed, and it is at least one, use a `do...while` loop

`break` and `continue` statements

- `break` and `continue` alter the flow of control
- `break` statement is used for two purposes:
  - To exit early from a loop
    - Can eliminate the use of certain (flag) variables
  - To skip the remainder of a `switch` structure
- After `break` executes, the program continues with the first statement after the structure

`break` and `continue` statements (cont'd.)

- `continue` is used in `while`, `for`, and `do…while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop

Avoiding Bugs by Avoiding Patches

- Software patch
  - Piece of code written on top of an existing piece of code
  - Intended to fix a bug in the original code
- Some programmers address the symptom of the problem by adding a software patch
- Should instead resolve underlying issue

Debugging Loops

- Loops are harder to debug than sequence and selection structures
- Most common error associated with loops is off-by-one
  - for(int i=0; i<5; i++) will loop 5 times
  - for(int i=0; i<=5; i++) loops 6 times
  - most often we use the former so if we want to loop 6 times use for(int i=0; i<6; i++)
  - every convention has its exceptions.

Summary

- C++ has three looping (repetition) structures:
  - `while`, `for`, and `do…while`
- `while`, `for`, and `do` are reserved words
- `while` and `for` loops are called pretest loops
- `do...while` loop is called a posttest loop
- `while` and `for` may not execute at all, but `do...while` always executes at least once

Summary (cont'd.)

- `while`: `logical expression` is the decision maker, and `statements` are the body of the loop
- A `while` loop can be:
  - Counter-controlled
  - Sentinel-controlled
  - eof-controlled
- In the UNIX console environment, the end-of-file marker is entered using `Ctrl+d`

Summary (cont'd.)

- `for` loop: simplifies the writing of a counter-controlled `while` loop
  - Putting a semicolon at the end of the `for` loop is a semantic error.
- Executing a `break` statement in the body of a loop immediately terminates the loop.
- Executing a `continue` statement in the body of a loop skips to the next iteration.

Debugging and Testing Programs

- Modern Integrated Development Environments (IDEs) include features to help you debug a program while it is executing.
- If you cannot use a debugger, insert extra diagnostic output statements to display intermediate results at critical points in your program.

Programming Error Helpful Hints

- Coding style and use of braces.
- Infinite loops will “hang you up !!”
- Use comments before and after a loop. This helps to see if you went into the loop and exited it.
- Test various conditions of loops.
- Add white space between code segments.
- Initialize looping variables inside the for loop. It will not be recognized outside the loop.
- eg. for(int i = 0; i<4; i++)

```
            cout << i << ". Hello" << endl;
    cout << i << endl;
// results in compile error. i is only meant as an index and has no other meaning.
```