C++ Programming Essentials

Unit 2: Structured Programming

CHAPTER 6: LOOPS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Basic Loop Concept

Chapter 6 Topics

- While Statement Syntax
- Count-Controlled Loops
- Event-Controlled Loops
- Using the End-of-File Condition to Control Input Data
- Using a While Statement for Summing and Counting
- Nested While Loops
- Loop Testing and Debugging
- Do-While Statement for Looping
- For Statement for Looping
- Using break and continue Statements

What is a loop?

- A loop is a repetition control structure.
- it causes a single statement or block to be executed repeatedly

Two Types of Loops

count controlled loops

repeat a specified number of times

event-controlled loops

some condition within the loop body changes and this causes the repeating to stop

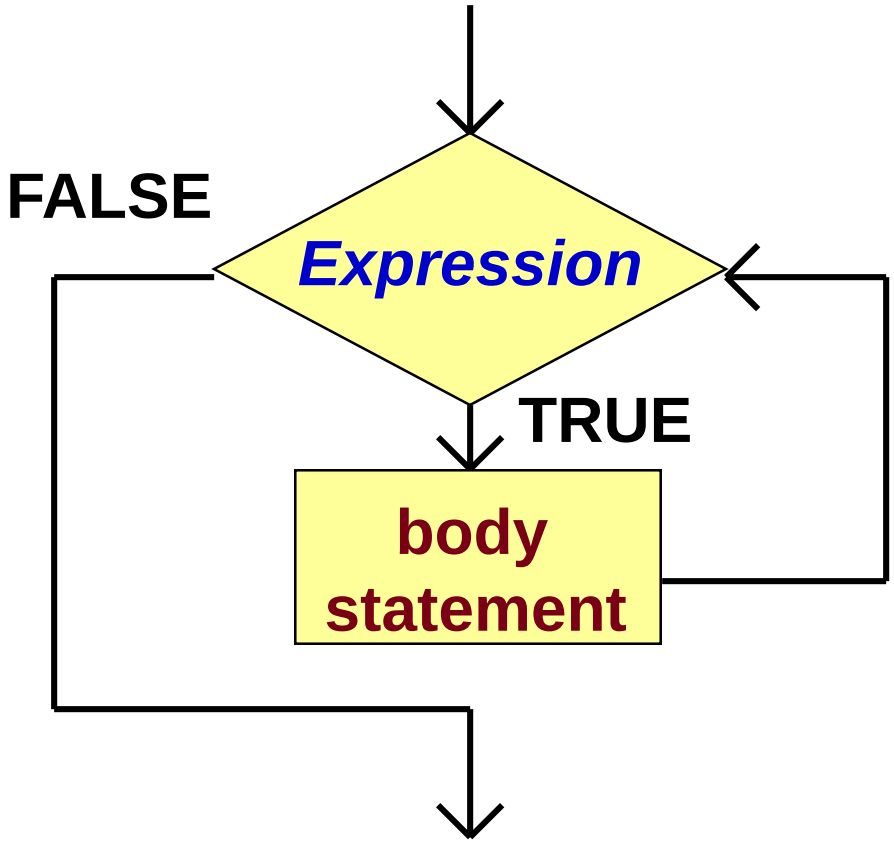
While Statement

SYNTAX

```
while  ( Expression )
{          .
         .                          // loop body
       .
}
```

NOTE: Loop body can be a single statement, a null statement, or a block.

WHILE LOOP



- When the expression is tested and found to be false, the loop is exited and control passes to the statement which follows the loop body.

LECTURE 2

Indexed Loop or Count Controlled loop

Count-controlled loop contains

- an initialization of the loop control variable
- an expression to test for continuing the loop
- an update of the loop control variable to be executed with each iteration of the body

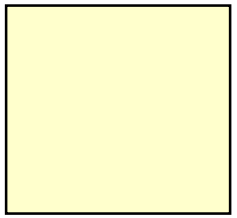
Count-controlled Loop

```
int   count ;
count  =  4;                          // initialize loop variable
while (count > 0)                     // test expression
{
        cout  << count  << endl ;     // repeated action
        count --  ;                   // update loop variable
}
cout  << "Done" << endl ;
```

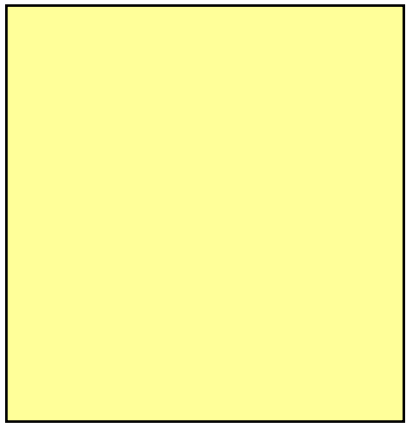
Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



OUTPUT



Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

4

OUTPUT

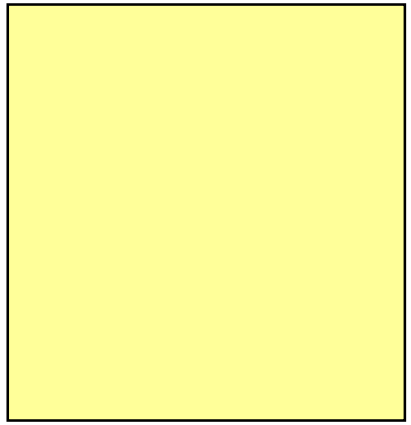
Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)          TRUE
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

4

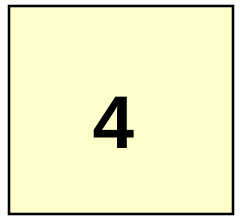
OUTPUT



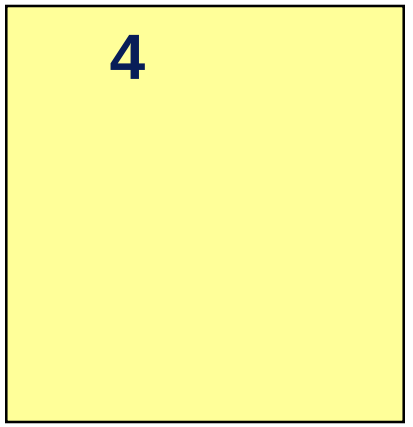
Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



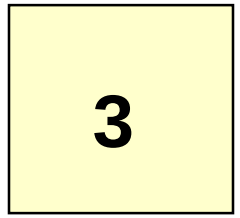
OUTPUT



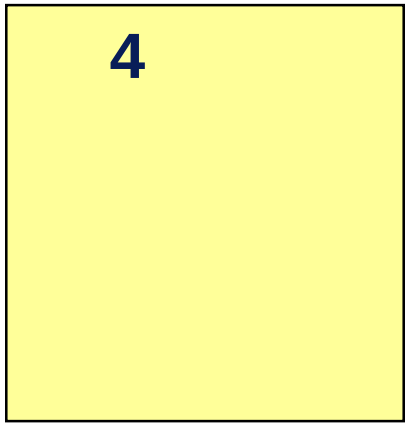
Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



OUTPUT



Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)          TRUE
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

3

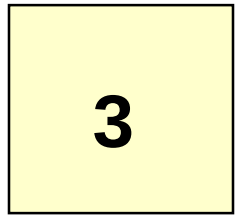
OUTPUT

4

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



OUTPUT

```
4
3
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;

}
cout  << "Done" << endl ;
```

count

2

OUTPUT

4
3

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)          TRUE
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

2

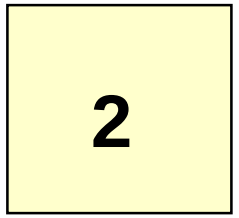
OUTPUT

4
3

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



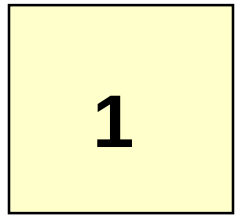
OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)          TRUE
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

1

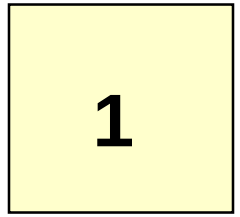
OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count



OUTPUT

```
4
3
2
1
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

0

OUTPUT

4
3
2
1

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)          FALSE
{
        cout  << count  << endl ;

        count --  ;
}
cout  << "Done" << endl ;
```

count

0

OUTPUT

```
4
3
2
1
```

Count-controlled Loop

```
int   count ;

count  =  4;

while (count > 0)
{
        cout  << count  << endl ;

        count --  ;
}

cout  << "Done" << endl ;
```

count

0

OUTPUT

```
4
3
2
1
Done
```

Count-Controlled Loop Example

- myInfile contains 100 blood pressures
- Use a while loop to read the 100 blood pressures and find their total

```
ifstream    myInfile ;
int         thisBP ;
int         total ;
int         count ;

count = 0 ;                         // initialize

while   ( count < 100 )                     // test expression
{
        myInfile   >>  thisBP ;
        total = total + thisBP ;
        count++ ;                           // update

}

cout  <<  "The total = "  <<  total  <<  endl ;
```

LECTURE 3

Event Controlled loops

Event-controlled Loops

Sentinel controlled

keep processing data until a special value which is not a possible data value is entered to indicate that processing should stop

End-of-file controlled

keep processing data as long as there is more data in the file

Flag controlled

keep processing data until the value of a flag changes in the loop body

Examples of Kinds of Loops

Count controlled loop	**Read exactly 100 blood pressures from a file.**
End-of-file controlled loop	**Read all the blood pressures from a file no matter how many are there.**

Examples of Kinds of Loops

Sentinel controlled loop	**Read blood pressures until a special value (like -1) selected by you is read.**
Flag controlled loop	**Read blood pressures until a dangerously high BP (200 or more) is read.**

A Sentinel-controlled Loop

- requires a “priming read”
- “priming read” means you read one set of data before the while
- Priming read also makes sure you don’t read one extra time
  - For end-of-file reading

```
// Sentinel controlled loop
total = 0;
cout  <<  “Enter a blood pressure (-1 to stop ) ”;
cin  >> thisBP;
while (thisBP != -1)                    // while not sentinel
{
        total = total + thisBP;
        cout <<  “Enter a blood pressure (-1 to stop ) ”;
        cin >> thisBP;
}
cout << total;
```

End-of-File Controlled Loop

- Depends on fact that a file goes into fail state when you try to read a data value beyond the end of the file
- Usually use a priming read

```
// End-of-file controlled loop
ifstream In("StockFile.txt");
total = 0;

In  >>  thisBP;              // priming read

while (In)      // while last read successful
{
    total = total + thisBP;
    In  >>  thisBP;          // read another
}

cout  <<  total;
```

```
//End-of-file at keyboard
total = 0;

cout  << "Enter blood pressure (Ctrl-Z to stop)";
cin  >> thisBP;                          // priming read

while (cin)                          // while last read successful
{
        total = total + thisBP;
        cout  << "Enter blood pressure";
        cin >> thisBP;                   // read another
}
cout << total;
```

Flag-controlled Loops

- you initialize a flag (to true or false)
- use meaningful name for the flag
- a condition in the loop body changes the value of the flag
- test for the flag in the loop test expression

```
countGoodReadings = 0;
isSafe = true;              // initialize Boolean flag
while (isSafe) {
        cin  >>  thisBP;
        if  ( thisBP >=  200 )
                isSafe = false;     // change flag value

                else
                countGoodReadings++;
}
cout  <<  countGoodReadings  <<   endl;
```

LECTURE 4

Loop Applications in Programs

Loops often used to

1. count all data values
2. count special data values
3. sum data values
4. keep track of previous and current values

Previous and Current Values

- write a program that counts the number of != operators in a program file
- read one character in the file at a time
- keep track of current and previous characters

Keeping Track of Values

```
(x  !=  3)
{
   cout  <<  endl;
}
```

FILE CONTENTS

previous	current	count
(	x	0
x	‘ ‘	0
‘ ‘	!	0
!	=	1
=	‘ ‘	1
‘ ‘	3	1
3	)	1

```
int      count;
char  previous;
char  current;

count = 0 ;
inFile.get (previous);                  // priming reads
inFile.get(current);

while (inFile)
{               if ( (current ==  '=') && (previous ==  '!') )
                 count++;
                previous = current;                 // update
    inFile.get(current);              // read another
}
```

LECTURE 5

Nested Loop

Pattern of a Nested Loop

```
initialize outer loop
while  ( outer loop condition )
{              . . .

        initialize inner loop
        while ( inner loop condition )
        {
                    inner loop processing and update
        }

          . . .
}
```

Patient Data

- A file contains blood pressure data for different people. Each line has a patient ID, the number of readings for that patient, followed by the actual readings.

ID	howMany	Readings				
4567	5	180	140	150	170	120
2318	2	170	210			
5232	3	150	151	151		

Read the data and display a chart

Patient ID	BP Average
4567	152
2318	190
5232	151
.	.
.	.
.	.

There were 432 patients in file.

Algorithm Uses Nested Loops

- initialize patientCount to 0
- read first ID and howMany from file
- while not end-of-file
  - increment patientCount
  - display ID
  - use a count-controlled loop to read and sum up this patient's howMany BP's
  - calculate and display average for patient
  - read next ID and howMany from file
- display patientCount

To design a nested loop

- begin with outer loop
- when you get to where the inner loop appears, make it a separate module and come back to its design later

```
#include <iostream>
#include <fstream>

using namespace std;

int  main ( )
{
    int          patientCount;          // declarations
    int          thisID;
    int          howMany;
    int           thisBP;
    int           totalForPatient;
    int           count;

    float        average;

    ifstream  myInfile;
```

```
myInfile.open("A:\\BP.dat");

if  (!myInfile )                                    // opening failed
{
        cout << "File opening error. Program terminated.";
        return  1;
 }

cout << "ID Number      Average BP" << endl;

patientCount = 0;

myInfile >> thisID >> howMany;          // priming read
```

```
while ( myInfile )                          // last read successful
{
    patientCount++;
    cout << thisID;
    totalForPatient = 0;            // initialize inner loop
      count = 0;
      while ( count < howMany)
      {
              myInfile >> thisBP;
              count ++;
              totalForPatient = totalForPatient  + thisBP;
      }

      average = totalForPatient / float(howMany);
      cout  <<  int (average + .5) << endl;          // round
      myInfile  >> thisID  >>  howMany;    // another read
}
```

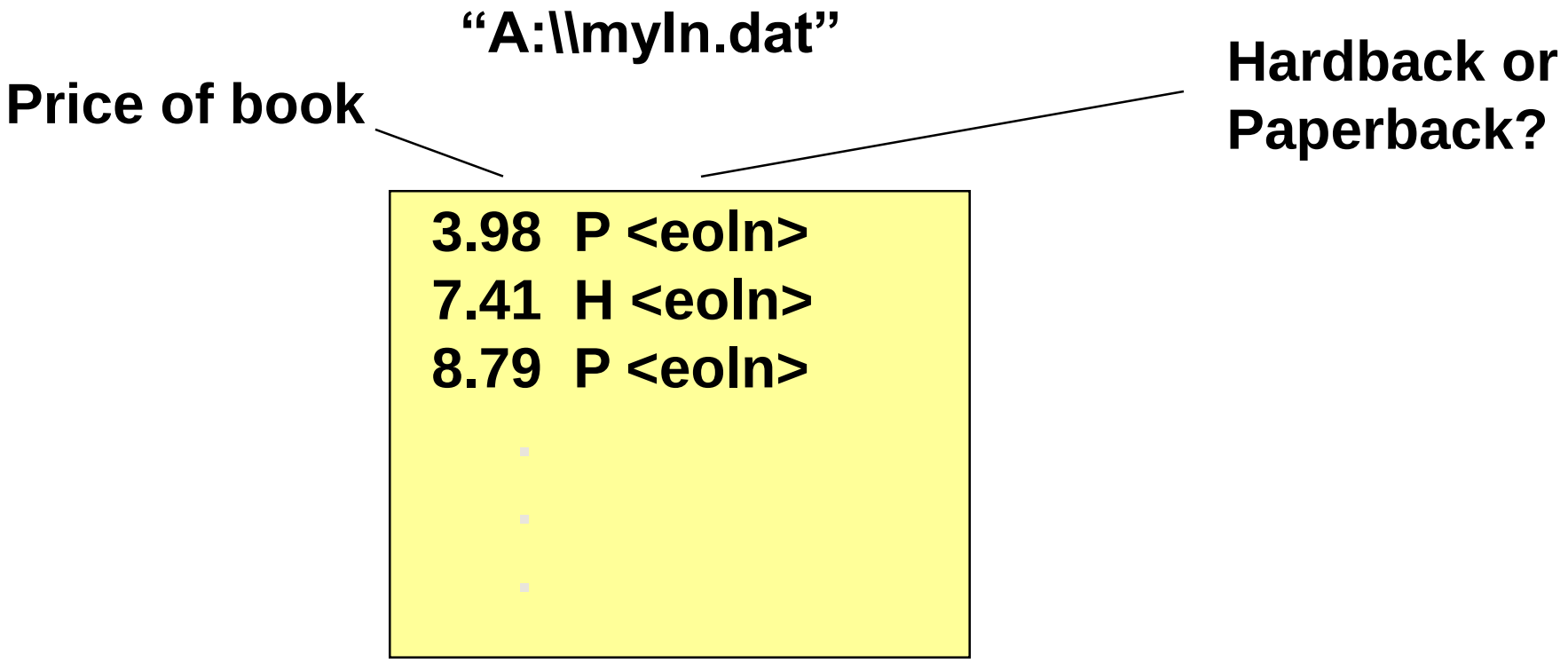
```
        cout  <<  "There were " <<  patientCount
              <<  "patients on file." << endl;

        cout  <<  "Program terminated.\n";

        return 0;

}
```

Demo Program:

patient.cpp

Go Dev C++!!!

Information About 20 Books in Diskfile



WRITE A PROGRAM TO FIND TOTAL VALUE OF ALL BOOKS

Program to Read Info about 20 Books From a Disk File

```
#include <iostream>        // for cout
#include <fstream>         // for file I/O

using namespace std;

int  main (void)
{
      float        price ;       // declarations
      char         kind ;
      ifstream     myInfile ;
      float        total  =  0.0 ;
      int          count  = 1;
```

Rest of Program

```
        myInfile.open("A:\\myIn.dat") ;

            // count-controlled processing loop
        while (  count <= 20 )
        {
                myInfile   >>  price  >>  kind ;
          total = total + price ;
          count ++ ;
        }
        cout << "Total is: " << total << endl ;
        myInfile.close(  ) ;
        return 0 ;
}
```

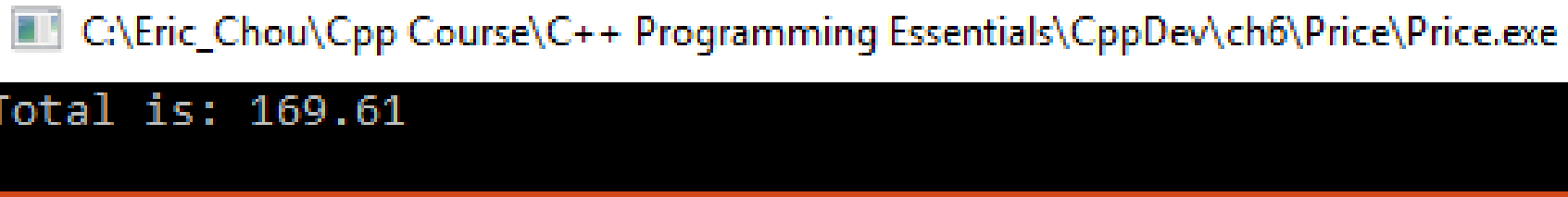
Trace of Program Variables

count	price	kind	total
			0.0
1	3.98	'P'	3.98
2	7.41	'H'	11.39
3	8.79	'P'	20.18
4	etc.		
20			
21	so loop terminates		

Demo Program:

price.cpp

Go Dev C++!!!



LECTURE 6

Do-While Loop

Do-While Statement

- Do-While is a looping control structure in which the loop condition is tested *after* each iteration of the loop

SYNTAX

```
do
{
        Statement


} while (Expression);
```

Loop body statement can be a single statement or a block

Example of Do-While

```
void GetYesOrNo (/* out */  char&  response)
//  Inputs a character from the user
//  Postcondition: response has been input
//       && response == 'y' or 'n'
{
    do
    {
        cin >> response;   // Skips leading whitespace

        if ((response != 'y') && (response != 'n'))
            cout << "Please type  y or n : ";
    } while ((response != 'y') && (response != 'n'));
}
```

Do-While Loop vs. While Loop

POST-TEST loop (exit-condition)

The looping condition is tested after executing the loop body

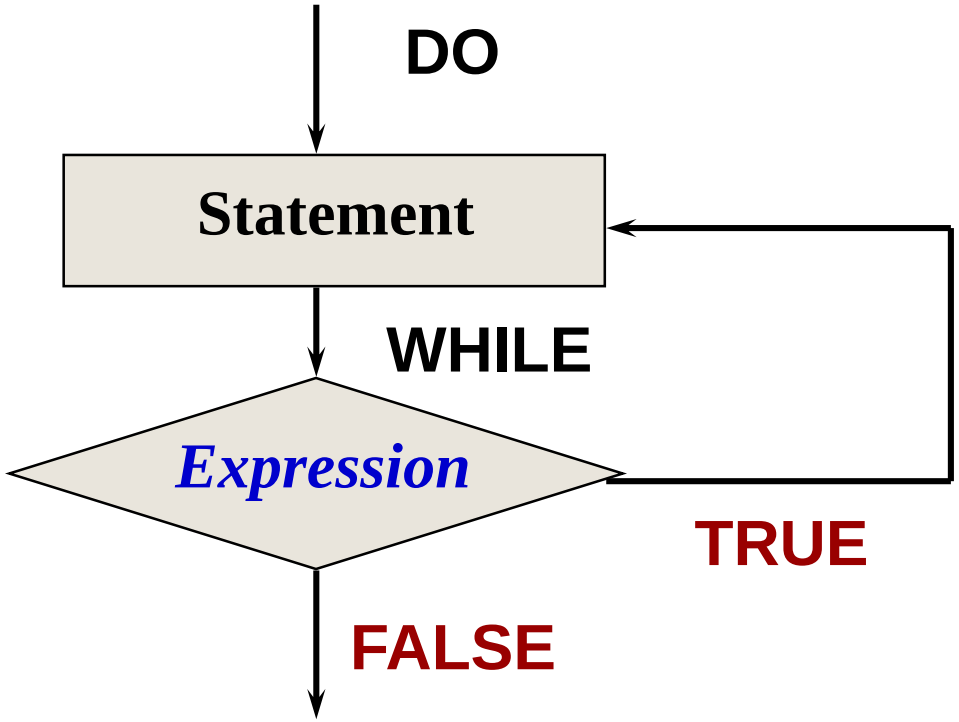
Loop body is always executed at least once

PRE-TEST loop (entry-condition)

The looping condition is tested before executing the loop body

Loop body may not be executed at all

Do-While Loop



- When the expression is tested and found to be false, the loop is exited and control passes to the statement that follows the Do-while statement

LECTURE 6

For-Loop

For Loop

SYNTAX

```
for (initialization;  test expression; update)
{
        Zero or more statements to repeat

}
```

For Loop

- For loop contains
- An initialization
- An expression to test for continuing
- An update to execute after each iteration of the body

Example of For Loop

```
int   num;

for (num = 1; num <= 3; num++)
{
    cout  <<  num  <<  "Potato"
          <<  endl;
}
```

Example of Repetition

num ?

```
int   num;
```

```
for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

Example of Repetition

num 1

```
int   num;
for (num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

Example of Repetition

num **1**

```
int   num;

                    true
for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

Example of Repetition

num 1

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
```

Example of Repetition

num **2**

```
int   num;

for (num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

```
1Potato
```

Example of Repetition

num **2**

```
int   num;

                    true
for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

```
1Potato
```

Example of Repetition

num **2**

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

1Potato

2Potato

Example of Repetition

num **3**

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

1Potato

2Potato

Example of Repetition

num **3**

```
int   num;

                    true
for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

```
1Potato
2Potato
```

Example of Repetition

num **3**

```
int   num;


for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

1Potato
2Potato
3Potato

Example of Repetition

num **4**

```
int   num;


for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

1Potato
2Potato
3Potato

Example of Repetition

num **4**

```
int   num;

                    false
for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

OUTPUT

1Potato
2Potato
3Potato

num 4

Example of Repetition

```
int   num;

                    false
for(num = 1; num <= 3; num++)
     cout  <<  num  <<  "Potato"
           <<  endl;
```

- When the loop control condition is evaluated and has value false, the loop is said to be “satisfied” and control passes to the statement following the For statement.

Output

The output was

1Potato

2Potato

3Potato

Count-controlled Loop

```
int  count;
for  (count = 4; count > 0; count--){
   cout  <<  count <<  endl;
}
cout    <<  “Done”  <<  endl;
```

OUTPUT: 4
3
2
1
Done

LECTURE 7

Loop Body (loop block)

What is output?

```
int  count;


for (count = 0; count < 10; count++)
{
    cout  <<  "*";
}
```

Answer

****

The 10 asterisks are all on one line. Why?

What output from this loop?

```
int  count;


for (count = 0;  count < 10;  count++);
{
   cout  <<  "*";
}
```

Answer

- No output from the for loop! *Why?*
- The semicolon after the () means that the body statement is a null statement

Answer

- In general, the body of the For loop is whatever statement immediately follows the ()
- That statement can be a single statement, a block, or a null statement
- Actually, the code outputs one * after the loop completes counting to 10

Several Statements in Body Block

```
const  int  MONTHS = 12;
int     count;
float   bill;
float   sum = 0.0;
for (count = 1;  count <= MONTHS;  count++) {
    cout << "Enter bill: ";
    cin   >>  bill;
    sum = sum + bill;
}
cout << "Your total bill is : "  << sum <<  endl;
```

LECTURE 8

Break Levels in C++

Break Statement

- The Break statement can be used with Switch or any of the 3 looping structures
- It causes an immediate exit from the Switch, While, Do-While, or For statement in which it appears
- If the Break statement is inside nested structures, control exits only the innermost structure containing it

Guidelines for Choosing Looping Statement

- For a simple count-controlled loop, use the For statement
- For an event-controlled loop whose body always executes once, use of Do-While statement
- For an event-controlled loop about which nothing is known, use a While statement
- When in doubt, use a While statement

Continue Statement

- The Continue statement is valid only within loops
- It terminates the current loop iteration, but not the entire loop
- In a For or While, Continue causes the rest of the body of the statement to be skipped; in a For statement, the update is done
- In a Do-While, the exit condition is tested, and if true, the next loop iteration is begun

LECTURE 9

Complexity Analysis of Loops

Complexity

- is a measure of the amount of work involved in executing an algorithm relative to the size of the problem

Polynomial Times

N	N^0 constant	N^1 linear	N^2 quadratic	N^3 cubic
1	1	1	1	1
10	1	10	100	1,000
100	1	100	10,000	1,000,000
1,000	1	1,000	1,000,000	1,000,000,000
10,000	1	10,000	100,000,000	1,000,000,000,000

Loop Testing and Debugging

- test data should test all sections of program
  - Boundary conditions
  - All branches
- beware of infinite loops -- program doesn't stop
- check loop termination condition, and watch for "off-by-1" problem
- Don't compare real numbers for equality
- use get function for loops controlled by detection of '\n' character
- use algorithm walk-through to verify pre- and postconditions
- trace execution of loop by hand with code walk-through
- use a debugger to run program in "slow motion" or use debug output statements