SWE20004
Technical Software Development

Lecture 4
Repetition Structures

Outline

- Basic loop structures
- **`while`** loop
- Interactive **`while`** loop
- **`do while`** loop
- **`for`** loop
- Loop programming techniques
- Nested loops
- Common programming errors

Basic Loop Structures

- Repetition structure has four required elements:
  - Repetition statement
  - Condition to be evaluated
  - Initial value for the condition
  - Loop termination
- Repetition statements include:
  - **`while`**
  - **`do while`**
  - **`for`**

Basic Loop Structures (continued)

- The condition can be tested
  - At the beginning: **Pretest** or **entrance-controlled** loop
  - At the end: **Posttest** or **exit-controlled** loop
- Something in the loop body must cause the condition to change, to avoid an **infinite loop**, which never terminates

Pretest and Posttest Loops

- Pretest loop: Condition is tested first; if false, statements in the loop body are never executed
- **`while`** and **`for`** loops are pretest loops

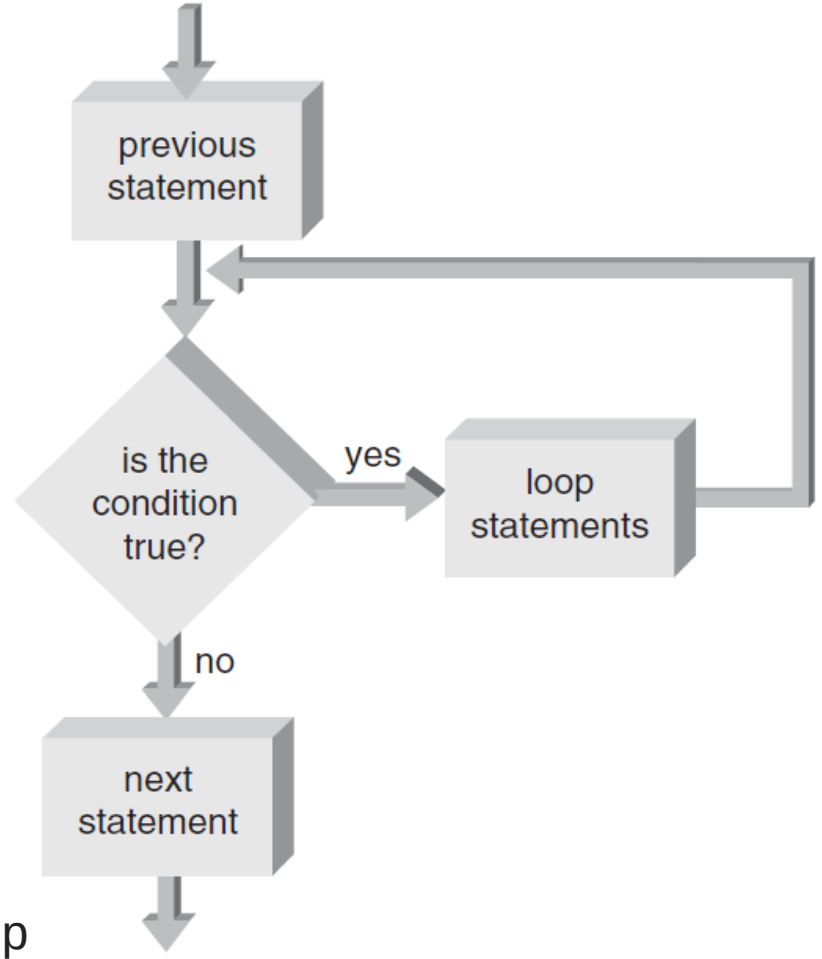


Figure 5.1 A pretest loop

Pretest and Posttest Loops (continued)

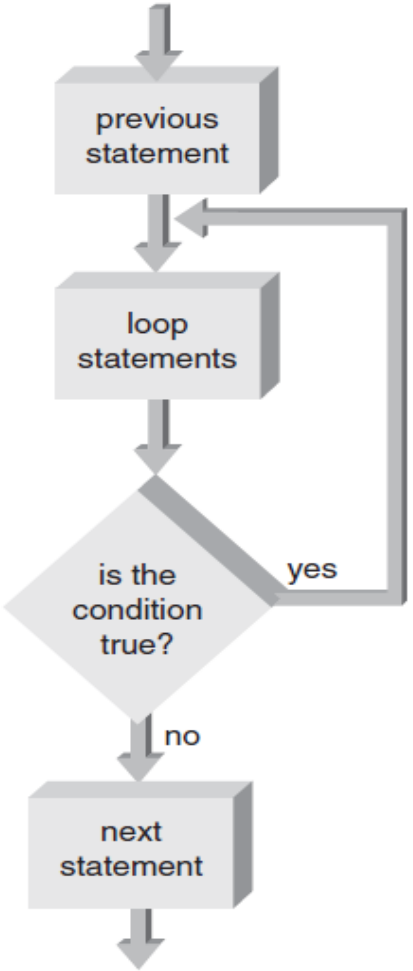


Figure 5.2 A posttest loop

Fixed-Count Versus Variable-Condition Loops

- **Fixed-count loop or definite loop:** Loop is processed for a fixed number of repetitions
- **Variable-condition loop or indefinite loop:** Number of repetitions depends on the value of a variable

`while` Loop

- **`while` statement** is used to create a `while` loop
  - Syntax:

    while (expression)

    statement;
- Statements following the expressions are executed as long as the expression condition remains true (evaluates to a non-zero value)

Repetition (looping) structure

- **while loop**: This is the simplest loop format. Its general form is

```
while (condition)
{
 statement_1;
    :
 statement_n;
}
```

- **Example**: Write a program to evaluate the square of integers from 1 to 3.

```
#include  <iostream>
using namespace std;
int main(  )
{
 int  value,  square;
 value  =  1;
 while  (value<4)
 {
 square  =  value*value;
 cout<<square;
 value  =  value  +  1;
 }
 return 0;
}
```

Program 5.1

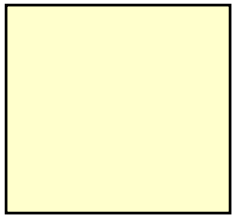
```
#include <iostream>
using namespace std;

int main()
{
   int count;

   count = 1;                  // initialize count
   while (count <= 10)
   {
      cout << count << "   ";
      count++;                 // increment count
   }

   return 0;
}
```

`while` Loop Demo

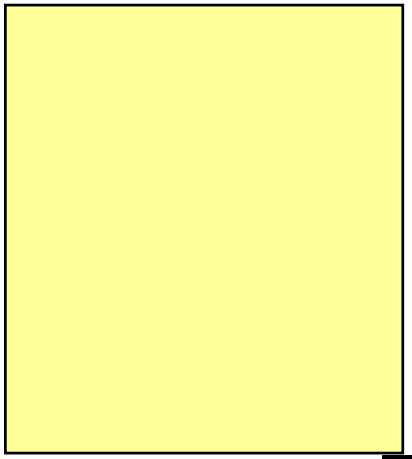
```
int   count;          // Loop-control variable

count  =  4;          // Initialize loop variable

while(count > 0)        // Test expression
{
    cout  << count  << endl; // Repeated action

    count --;     // Update loop variable
}
cout  << "Done" << endl;
```

`while` Loop Demo

```
int   count;

count  =  4;


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

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

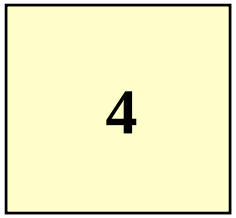


OUTPUT

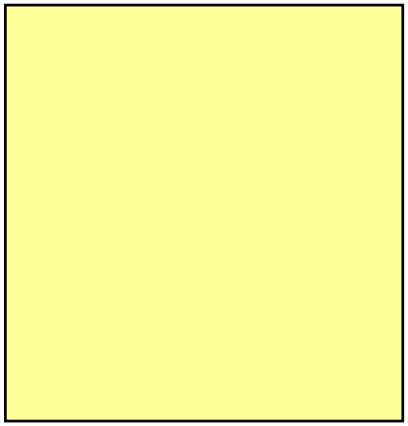


while Loop Demo

```
int   count;

count  =  4;


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

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

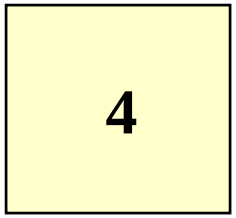


OUTPUT

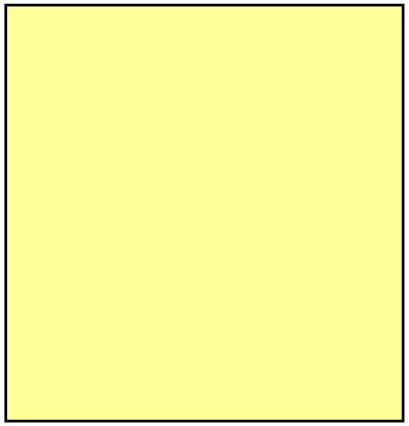


`while` Loop Demo

```
int   count;

count  =  4;

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

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

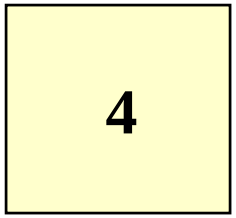


OUTPUT

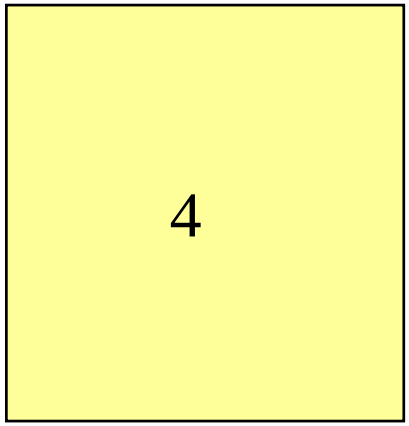


`while` Loop Demo

```
int   count;

count  =  4;


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

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

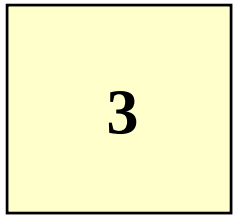


OUTPUT

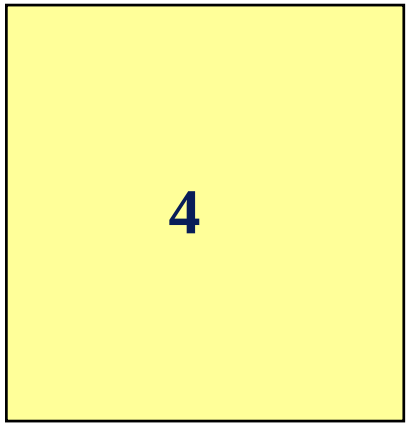


`while` Loop Demo

```
int   count;

count  =  4;



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

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



OUTPUT



while Loop Demo

```
int   count;

count  =  4;

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

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

3

OUTPUT

4

`while` Loop Demo

```
int   count;

count  =  4;

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

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

3

4
3

`while` Loop Demo

```
int   count;

count  =  4;

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

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

2

4
3

`while` Loop Demo

```
int   count;

count  =  4;

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

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

2

OUTPUT

```
4
3
```

while Loop Demo

```
int   count;

count  =  4;

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

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

2

OUTPUT

```
4
3
2
```

while Loop Demo

```
int   count;

count  =  4;

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

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

1

OUTPUT

4
3
2

while Loop Demo

```
int   count;
count  =  4;

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

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

1

OUTPUT

```
4
3
2
```

`while` Loop Demo

```
int   count;

count  =  4;

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

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

1

OUTPUT

4
3
2
1

`while` Loop Demo

```
int   count;

count  =  4;
while(count > 0)
{
     cout  << count  << endl;

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

0

OUTPUT

4
3
2
1

`while` Loop Demo

```
int   count;

count  =  4;

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

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

0

OUTPUT

4
3
2
1

`while` Loop Demo

```
int   count;

count  =  4;


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


     count --;
}

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

0

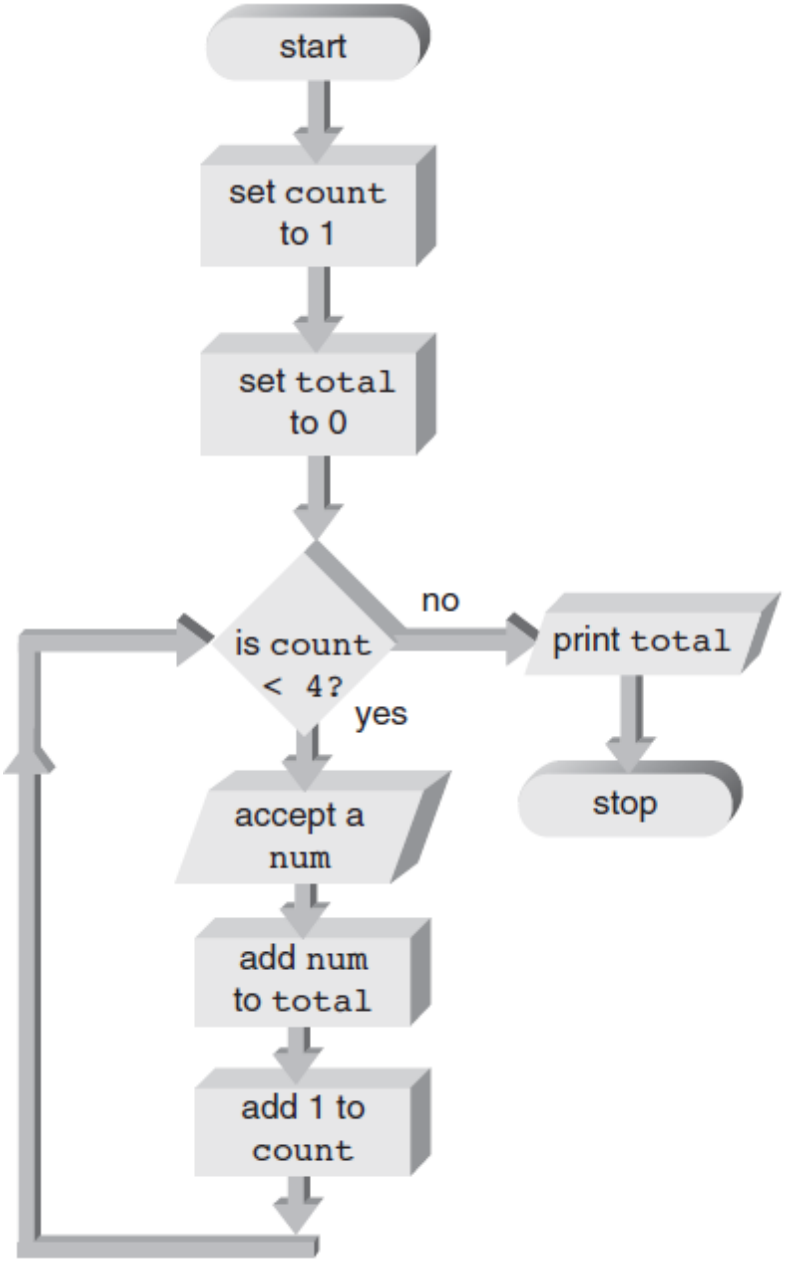
OUTPUT

4
3
2
1
Done

Interactive `while` Loop

- Combining interactive data entry with the **`while`** statement provides for repetitive entry and accumulation of totals

Figure 5.7 Accumulation flow of control



`do while` Loop

- **`do while`** loop is a posttest loop
  - Loop continues while the condition is true
  - Condition is tested at the end of the loop
  - Syntax:

```
do
        statement;
while (expression);
```

- All statements are executed at least once in a posttest loop

Figure 5.13 The `do while` loop structure.

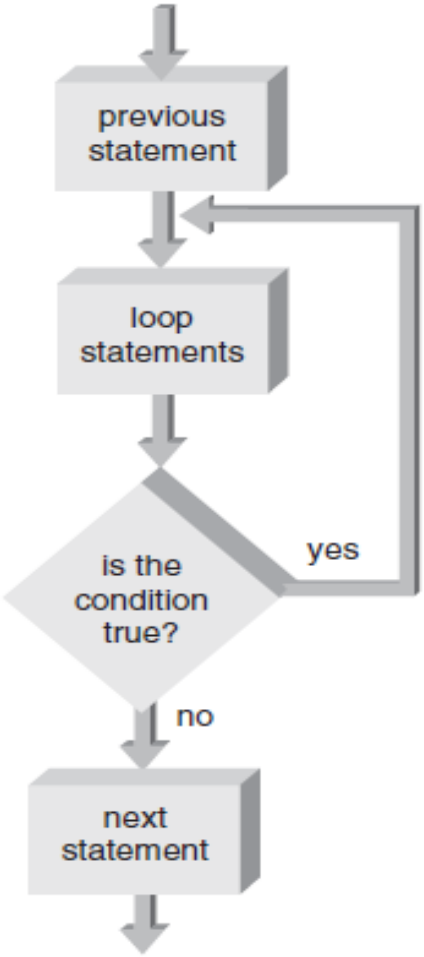
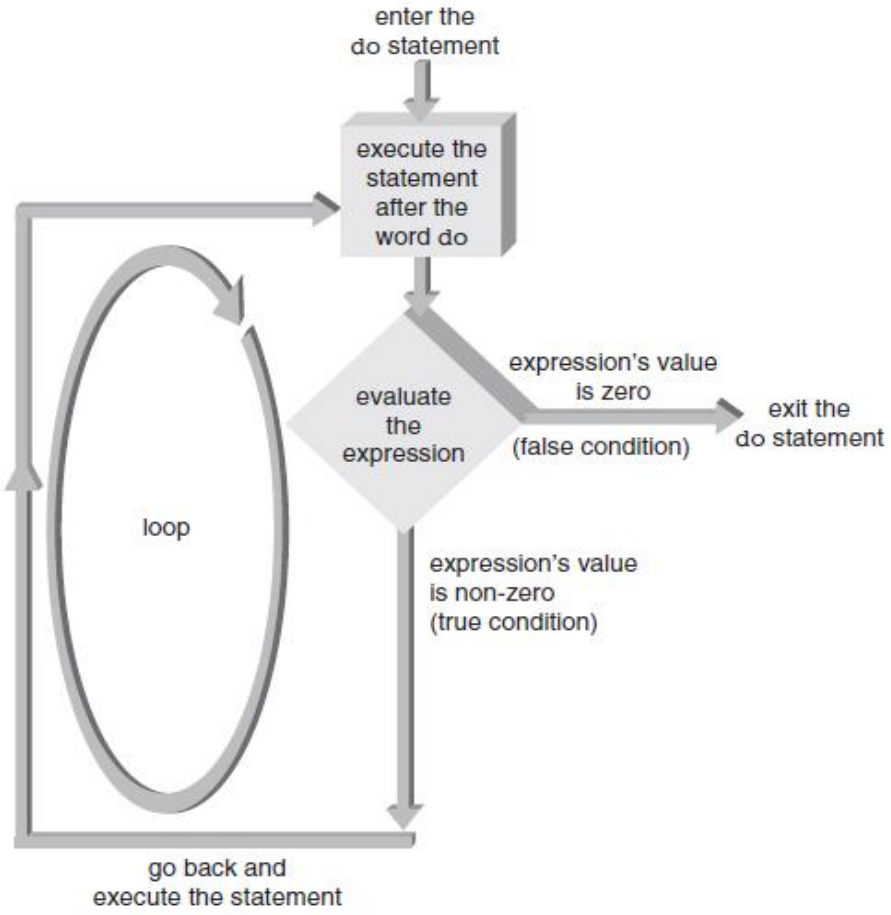


Figure 5.14 The `do` statement's flow of control.



Validity Checks

- Useful in filtering user-entered input and providing data validation checks

```
do
{
  cout << "\nEnter an identification number: ";
  cin  >> id_num;
}
while (id_num < 1000 || id_num > 1999);
```

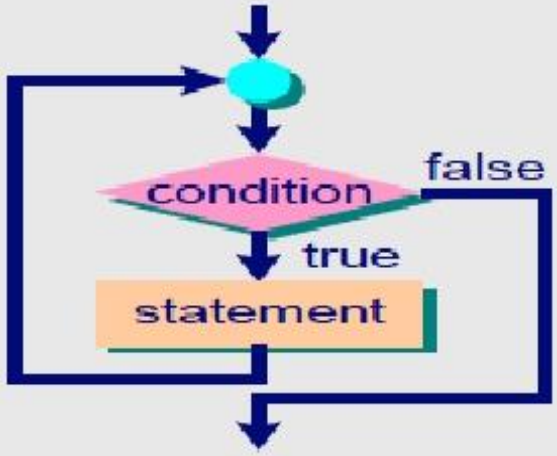
Comparison between while and do-while

while structure

Syntax:

```
while (condition)
   statement;
```

Flowchart

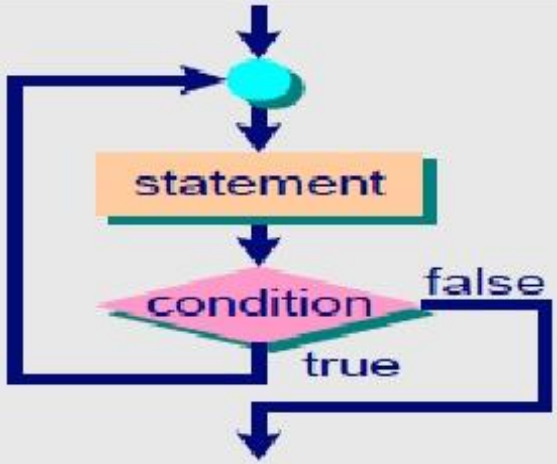


do-while structure

Syntax:

```
do {
statement;
} while (condition);
```

Flowchart



Review: Write a program to evaluate the square of integers from 1 to 5.

```
#include <iostream>
Using namespace std;
int main( )
{
 int value, square;
 value = 1;
 while (value<6)
 {
 square = value*value;
 cout<<square;
 value = value + 1;
 }
 return 0;
}
```

```
#include <iostream>
Using namespace std;
int main( )
{
      int value, square;
      value = 1;
      do
      {
      square = value*value;
      cout<<square;
      value = value + 1;
      } while (value<6);
 return 0;
}
```

Count-controlled loops contain:

- An **initialization** of the loop control variable
- An **expression** to test if the proper number of repetitions has been completed
- An **update** of the loop control variable to be executed with each iteration of the body

Top-down, stepwise refinement

- Many programs have three phases
  1. Initialization
     - initializes the program variables
  2. Processing
     - inputs data values and adjusts program variables accordingly
  3. Termination
     - calculates and prints the final results
- Helps to breakup programs for top-down refinement
- Easy to understand, test, debug and modify programs

Case Study 1

– *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average for the quiz.*

- Pseudocode:

Set total to zero
While grade counter is less than or equal to ten
Input the next grade
Add the grade to the total
Add one to the grade counter
Set the class average to the total divided by ten
Print the class average

```
1  /* Class average program with
2  counter-controlled repetition */
3
4  #include <iostream>
5  using namespace std;
6  int main()
7  {
8     int counter, grade, total;
9     float average;
10    /* initialization phase */
11    total = 0;
12    counter = 0;
13
14    /* processing phase */
15    while(counter<10) {
16       cout<<"Enter grade: ";
17       cin>>grade;
18       total = total + grade;
19       counter ++;
20      }
21    /* termination phase */
22    average = (float)total / counter;
23    cout<<"Class average is "<< average << endl;
24
25    return 0;   /* indicate program ended successfully */
26  }
```

C ++Statement

1. Initialize Variables

2. Execute Loop

3. Output results

Sentinels

- **Sentinel:** A data value used to signal either the start or end of a data series
  - Use a sentinel when you don't know how many values need to be entered

Case Study 2

- Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.
  - Different to Case Study 1:
    - Unknown number of students
    - How will the program know to end?
- Use sentinel value
  - a.k.a. signal value, dummy value, or flag value
  - Indicates “end of data entry”
  - Loop ends when sentinel inputted
  - Sentinel value chosen so it cannot be confused with a regular input (such as -1 in this case)

Top-down, stepwise refinement

- Begin with a pseudocode representation of the top:

  Determine the class average for the quiz

- Divide top into smaller tasks and list them in order:

  Initialize variables
  Input, sum and count the quiz grades
  Calculate and print the class average

Top-down, stepwise refinement

- Refine the initialization phase from *Initialize variables* to:

  Initialize total to zero
  Initialize counter to zero

- Refine *Input, sum and count the quiz grades* to

  Input the first grade (possibly the sentinel)
  While the user has not as yet entered the sentinel
  Add this grade to the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

- Refine *Calculate and print the class average* to

  If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
  else
  Print "No grades were entered"

```
1   /* Class average program with
2   sentinel-controlled repetition */
3
4  #include <iostream>
5   using namespace std;
6  int main()
7  {
8     float average;
9     int counter, grade, total;
10
11    /* initialization phase */
12    total = 0;
13    counter = 0;
14
15    /* processing phase */
16    cout<<"Enter grade, -1 to end: " ;
17    cin>>grade;
18
19    while ( grade != -1 ) {
20       total = total + grade;
21       counter = counter + 1;
22       printf( "Enter grade, -1 to end: " );
23       cin>>grade;
24    }
```

C ++Statement

1. Initialize Variables

2. Get User Input

2.1 Execute Loop

SWINBURNE UNIVERSITY OF TECHNOLOGY

```
25
26    /* termination phase */
27    if ( counter != 0 ) {
28       average = ( float ) total / counter;
29       cout<<"Class average is "<< average );
30    }
31    else
32       cout<<"No grades were entered\n");
33
34    return 0;   /* indicate program ended successfully */
35 }
```

3. Calculate Average

3.1 Print Results

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Program Output

`for` Loop

- **`for`** statement: A loop with a fixed count condition that handles alteration of the condition
  - Syntax:

    for (initializing list; expression; altering list)

    statement;
- **Initializing list:** Sets the starting value of a counter
- **Expression:** Contains the maximum or minimum value the counter can have; determines when the loop is finished

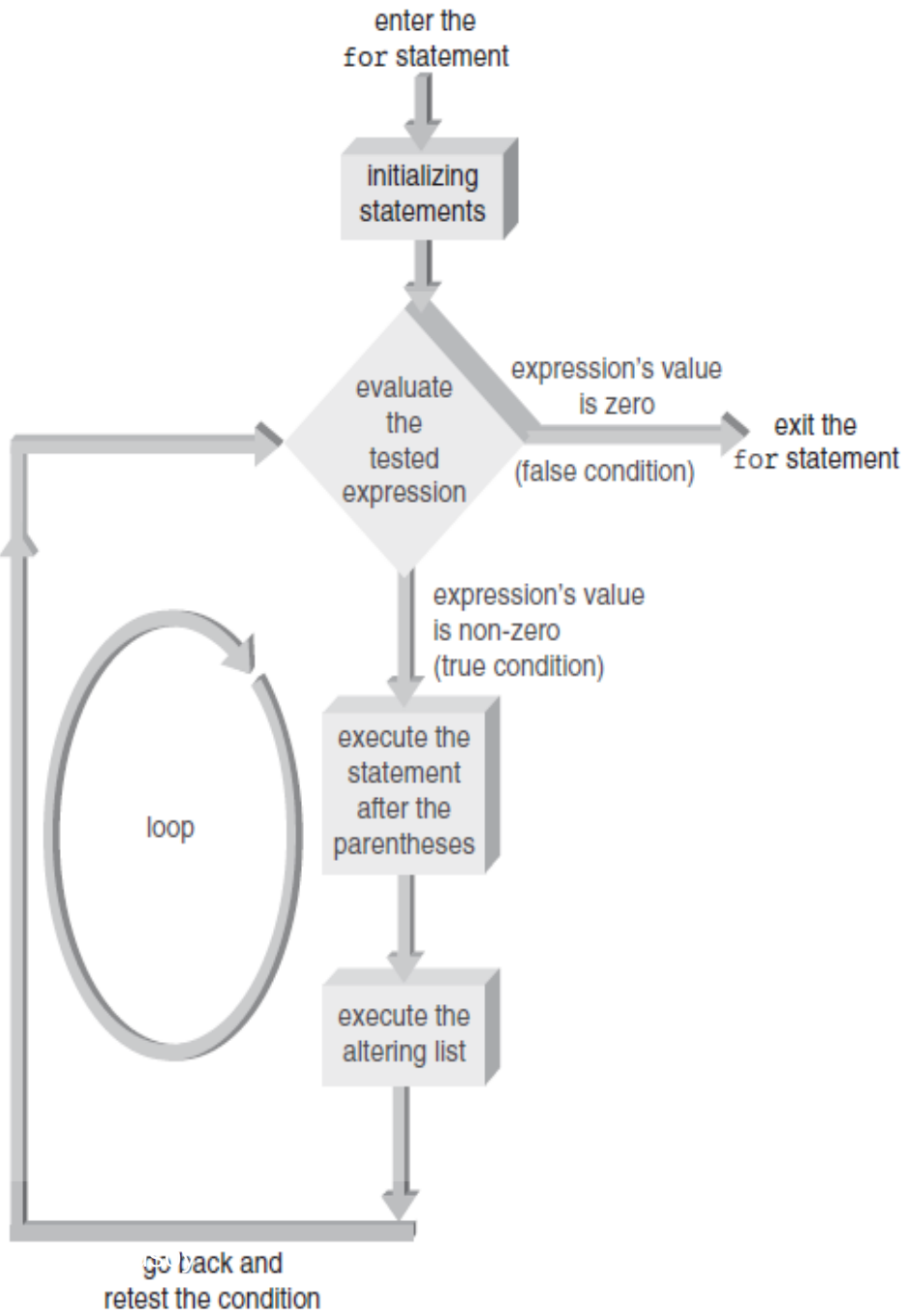
`for` Loop (continued)

- **Altering list:** Provides the increment value that is added or subtracted from the counter in each iteration of the loop
- If initializing list is missing, the counter initial value must be provided prior to entering the `for` loop
- If altering list is missing, the counter must be altered in the loop body
- Omitting the expression will result in an infinite loop

Program 5.9

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  const int MAXCOUNT = 5;
  int count;

  cout << "NUMBER  SQUARE ROOT\n";
  cout << "------    -----------\n";

  cout << setiosflags(ios::showpoint);
  for (count = 1; count <= MAXCOUNT; count++)
    cout << setw(4) << count
         << setw(15) << sqrt(double(count)) << endl;

  return 0;
}
```

Figure 5.10 `for` loop flowchart.



`for` Loop Demo

```
int   num;

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

`for` Loop Demo

```
int   num;

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

num

?

OUTPUT

`for` Loop Demo

```
int   num;

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

num

1

OUTPUT

`for` Loop Demo

```
int    num;

for(num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<  endl;
```

true

num

1

OUTPUT

for Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<endl;
```

num

1

OUTPUT

```
1Potato
```

`for` Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<endl;
```

num

2

OUTPUT

```
1Potato
```

`for` Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout << num << "Potato" << endl;
```

true

num

2

OUTPUT

```
1Potato
```

`for` Loop Demo

```
int   num;

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

num

2

OUTPUT

```
1Potato
2Potato
```

`for` Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<< endl;
```

num

3

OUTPUT

```
1Potato
2Potato
```

`for` Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<endl;
```

true

num

3

OUTPUT

1Potato

2Potato

for Loop Demo

```
int   num;

for(num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<  endl;
```

num

3

OUTPUT

1Potato

2Potato

3Potato

`for` Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato" <<endl;
```

num

4

OUTPUT

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

for Loop Demo

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"<<  endl;
```

false

num

4

OUTPUT

1Potato

2Potato

3Potato

for Loop Demo

```
int   num;

                        false
for (num = 1; num <= 3; num++)

     cout  <<  num  <<  "Potato"<<  endl;
```

num

4

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

Case Study 3

- A college has a list of test results (1 = pass, 2 = fail) for 10 students.
  - If more than 5 students failed, print "Tuition Needed"

- Write a program that analyzes the results
  - The program must process 10 test results
  - Counter-controlled loop will be used
  - Two counters can be used: one for number of passes, one for number of fails
  - Each test result is a number — either a 1 (pass) or a 2 (fail)
  - If the number is not a 1, we assume that it is a 2: a selection statement will be used

Top-down, stepwise refinement

- Begin with a pseudocode representation of the top:

  Analyze exam results and decide if tuition should be raised

- Divide top into smaller tasks and list them in order:

  Initialize variables

  Input the ten quiz grades and count passes and failures

  Print a summary of the exam results and decide if tuition should be raised

Top-down, stepwise refinement

- Refine the initialization phase from *Initialize variables* to:

  Initialize pass to zero
  Initialize fail to zero
  Initialize student counter to one

- Refine *Input the ten quiz grades and count passes and failures* to

  While student counter is less than or equal to ten
  Input the next exam result
  If the student passed
  Add one to pass
  else
  Add one to fail
  Add one to student counter

- Refine *Print a summary of the exam results and decide if tuition should be raised* to

  Print the number of pass
  Print the number of fail

  If more than eight students passed
  Print "Tuition needed"

```
1  /* Analysis of examination results */
2
3  #include <stdio.h>
4  #include "terminal_user_input.h"
5  int main()
6  {
7     /* initializing variables in declarations */
8     int pass= 0, fail = 0, student, result;
9
10    /* process 10 students; counter-controlled loop */
11    for( student = 1; student <= 10; student++ ) {
12       cout<<"Enter result ( 1=pass,2=fail ): " ;
13       cin>>result;
14
15       if ( result == 1 )          /* if/else nested in while
16          pass = pass + 1;
17       else
18          fail = fail + 1;
19
20       student = student + 1;
21    }
22
23    cout<< "Passed "<< pass;
24    cout<<"Failed "<< fail;
25
26    if ( fail > 5 )
27       cout<<"Tuition needed\n" ;
28
29    return 0;   /* successful termination */
30 }
```

C Statement

1. Initialize Variables

2. Execute Loop:

   2.1 Input data

   2.2 count passes or failures

3. Output results

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Program Output

- **for loop: special cases**

Syntax: for (initialization; condition; modification)
{statements;}

- The initialization and modification expressions in a **for** loop can contain more than one statement

```
for (n = 1, m = 5; n <= 10;  n++, m+=2)
{
 sum1 += n;
 sum2 += m;
}
```

- Any or all of three expressions in a **for** loop can be omitted

```
for (;;)

for (; k <= 10;  k++)

for (i = 5; i <= 10; )
```

`break` and `continue` Statements

- **`break`** statement
  - Forces an immediate break, or exit, from **`switch`**, **`while`**, **`for`**, and **`do-while`** statements
  - Violates pure structured programming, but is useful for breaking out of loops when an unusual condition is detected

`break` and `continue` Statements (cont'd)

- Example of a `break` statement:

```
while (count <= 10)
{
  cout << "Enter a number: ";
  cin  >> num;
  if (num > 76)
  {
    cout << "You lose!\n";
    break;        // break out of the loop
  }
  else
    cout << "Keep on trucking!\n";
  count++;
}
// break jumps to here
```

`break` and `continue` Statements (cont'd)

- **`continue`** statement
  - Applies to **`while`**, **`do-while`**, and **`for`** statements; causes the next iteration of the loop to begin immediately
  - Useful for skipping over data that should not be processed in this iteration, while staying within the loop

`break` and `continue` Statements (cont'd)

- A `continue` statement where invalid grades are ignored, and only valid grades are added to the total:

```
while (count < 30)
{
  cout << "Enter a grade: ";
  cin  >> grade
  if(grade < 0 || grade > 100)
    continue;
  total = total + grade;
  count++;
}
```

break & continue statements

- **break** can be used with any of loop structures to immediately exit from the loop in which it is contained.
- In contrast, **continue** is used to skip the remaining statements in the current iteration of the loop and then continue with the next.

- **Example : Examine the following programs**

```
int x, sum =0,k;
for (k=1; k<20; k++)
{
        cin>>x;
        if (x > 10.0)
          break;
        sum+=x;
}
cout<<"Sum is "<<sum<<endl;
```

```
int x, sum =0,k;
for (k=1; k<20; k++)
{
        cin>>x;
        if (x > 10.0)
          continue;
        sum+=x;
}
cout<<"Sum is "<<sum<<endl;
```

Infinite loop

- An infinite loop is generated if the condition in **while, do-while, or for loop** is ***always*** true

Example : Examine the following programs

```
value = 1;
while (value<10)
{
cout<<value;
value--;
}
```

```
value = 10;
do
{
cout<<value;
value++;
} while (value>1);
```

```
for (k=1; k<10; k--)
{
cout<<value;
}
```

A Closer Look: Loop Programming Techniques

- These techniques are suitable for pretest loops (**`for`** and **`while`**):
  - **Interactive input within a loop**
    - Includes a **`cin`** statement within a **`while`** or **`for`** loop
  - **Selection within a loop**
    - Using a **`for`** or **`while`** loop to cycle through a set of values to select those values that meet some criteria

Program 5.13

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

// This program computes the positive and negative sums of a set
// of MAXNUMS user-entered numbers
int main()
{
  const int MAXNUMS = 5;
  int i;
  double usenum, positiveSum, negativeSum;
```

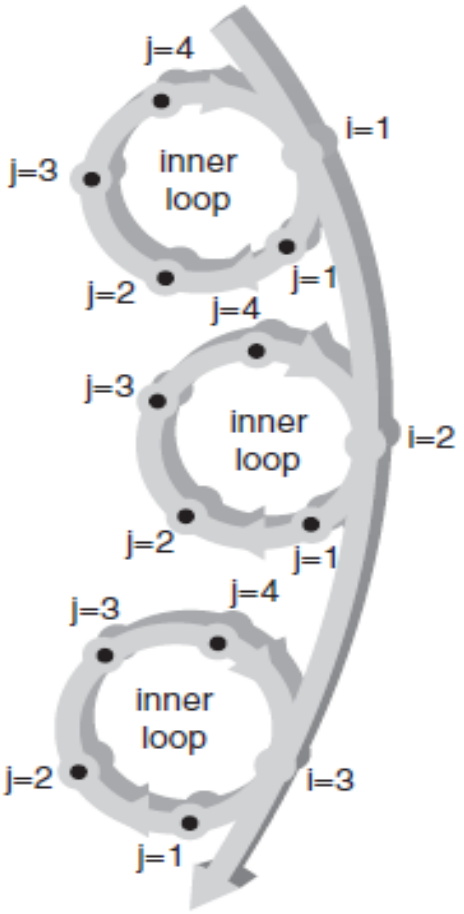
```
  positiveSum = 0; // this initialization can be done in the declaration
  negativeSum = 0; // this initialization can be done in the declaration
  for (i = 1; i <= MAXNUMS; i++)
  {
    cout << "Enter a number (positive or negative) : ";
    cin  >> usenum;
    if (usenum > 0)
      positiveSum = positiveSum + usenum;
    else
      negativeSum = negativeSum + usenum;
  }
  cout << "The positive total is " << positiveSum << endl;
  cout << "The negative total is " << negativeSum << endl;

  return 0;
}
```

Nested Loops

- **Nested loop:** A loop contained within another loop
  - All statements of the inner loop must be completely contained within the outer loop; no overlap allowed
  - Different variables must be used to control each loop
  - For each single iteration of the outer loop, the inner loop runs through all of its iterations

Nested Loops (continued)

Figure 5.12 For each `i, j` loops.



Nested for Loops

```
for (k=1; k<=3; k++)
  for (j=0; j<=1; j++)
   count++;
```

- The outer for loop will be executed 3 times.
- The inner for loop will be executed twice each time the outer for loop is executed.
- Thus, the variable count will be incremented 6 times.

Program 5.19

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

int main()
{
  const int MAXI = 5;
  const int MAXJ = 4;
  int i, j;

  for (i = 1; i <= MAXI; i++)     // start of outer loop <----+
  {                               //                          |
    cout << "\ni is now " << i << endl;  //                   |
                                  //                          |
    for (j = 1; j <= MAXJ; j++)   // start of inner loop      |
     cout << "   j = " << j;      // end of inner loop        |
  }                               // end of outer loop  <-----+
  cout << endl;

  return 0;
}
```

Common Programming Errors

- Making the “off by one” error: loop executes one too many or one too few times
- Using the assignment operator (=) instead of the equality comparison operator (==) in the condition expression
- Testing for equality with floating-point or double-precision operands; use an epsilon value instead

Common Programming Errors (continued)

- Placing a semicolon at the end of the **`for`** clause, which produces a null loop body
- Using commas instead of semicolons to separate items in the **`for`** statement
- Changing the value of the control variable
- Omitting the final semicolon in a **`do`** statement