ENGINEERING PROGRAMMING I

Chapter 05

2017

Topics

- 5.1 The Increment and Decrement Operators
- 5.2 Introduction to Loops: The while Loop
- 5.3 Using the while loop for Input Validation
- 5.4 Counters
- 5.5 The do-while loop
- 5.6 The for loop
- 5.7 Keeping a Running Total
- 5.8 Sentinels
- 5.9 Using a Loop to Read Data From a File
- 5.10 Deciding Which Loop to Use
- 5.11 Nested Loops
- 5.12 Breaking Out of a Loop
- 5.13 The continue Statement
- 5.14 Creating Good Test Data

The Increment and Decrement Operators

- ++ adds one to a variable
- `val++;` is the same as `val = val + 1;`

- -- subtracts one from a variable
- `val--;` is the same as `val = val - 1;`

- can be used in prefix mode (before) or postfix mode (after) a variable

```
--val; ++val;
val--; val++;
```

Prefix Mode

- **`++val`** and **`--val`** increment or decrement the variable, *then* return the new value of the variable.
- It is this returned new value of the variable that is used in any other operations within the same statement

Prefix Mode Example

```
int x = 1, y = 1;

x = ++y;          // y is incremented to 2
                  // Then 2 is assigned to x
cout << x
  << "  " << y; // Displays 2  2

x = --y;          // y is decremented to 1
                  // Then 1 is assigned to x
cout << x
  << "  " << y; // Displays 1 1
```

Postfix Mode

- **`val++`** and **`val--`** return the old value of the variable, *then* increment or decrement the variable
- It is this returned old value of the variable that is used in any other operations within the same statement

Postfix Mode Example

```
int x = 1, y = 1;

x = y++;          // y++ returns a 1
                  // The 1 is assigned to x
                  // and y is incremented to 2
cout << x
  << "  " << y; // Displays 1  2

x = y--;          // y-- returns a 2
                  // The 2 is assigned to x
                  // and y is decremented to 1
cout << x
  << "  " << y; // Displays 2 1
```

Increment & Decrement Notes

- Can be used in arithmetic expressions

```
result = num1++ + --num2;
```

- Must be applied to something that has a location in memory. Cannot have

```
result = (num1 + num2)++; // Illegal
```

- Can be used in relational expressions

```
if (++num > limit)
```

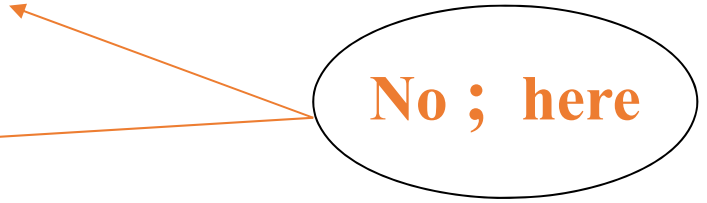
- Pre- and post-operations will cause different comparisons

Introduction to Loops: The while Loop

- Loop: part of program that may execute > 1 time (*i.e.*, it repeats)

- **while** loop format:

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



- The **{}** can be omitted if there is only one statement in the body of the loop

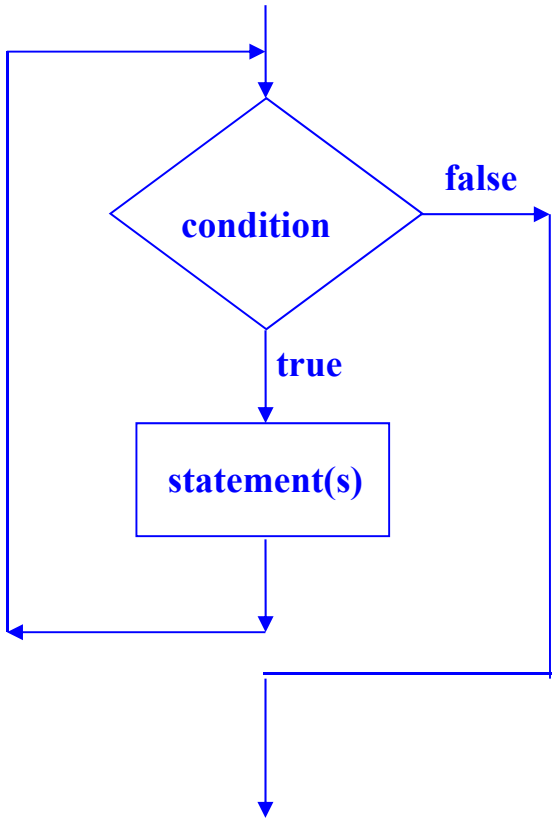
How the `while` Loop Works

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

condition is evaluated

- if it is true, the ***statement(s)*** are executed, and then ***condition*** is evaluated again
- if it is false, the loop is exited

`while` Loop Flow of Control



while Loop Example

```
int val = 5;
while (val >= 0)
{
    cout << val << "  ";
    val--;
}
```

- produces output:

```
5  4  3  2  1  0
```

while Loop is a Pretest Loop

- **`while`** is a pretest loop (***`condition`*** is evaluated before the loop executes)

- If the condition is initially false, the statement(s) in the body of the loop are never executed

- If the condition is initially true, the statement(s) in the body continue to be executed until the condition becomes false

Exiting the Loop

- The loop must contain code to allow ***condition*** to eventually become **false** so the loop can be exited
- Otherwise, you have an infinite loop (*i.e.*, a loop that does not stop)
- Example infinite loop:

```
x = 5;
while (x > 0)     // infinite loop
because
    cout << x;    // x is always > 0
```

Common Loop Errors

- Don't forget the { } :

```
int numEntries = 1;
while (numEntries <=3)
   cout << "Still working … ";
   numEntries++; // not in the loop body
```

- Don't use = when you mean to use ==

```
while (numEntries = 3)  // always true
{
   cout << "Still working … ";
   numEntries++;
}
```

Counters

- Counter: variable that is incremented or decremented each time a loop repeats
- Can be used to control execution of the loop (loop control variable)
- Must be initialized before entering loop
- May be incremented/decremented either inside the loop or in the loop test

Letting the User Control the Loop

- Program can be written so that user input determines loop repetition
- Can be used when program processes a list of items, and user knows the number of items
- User is prompted before loop. Their input is used to control number of repetitions

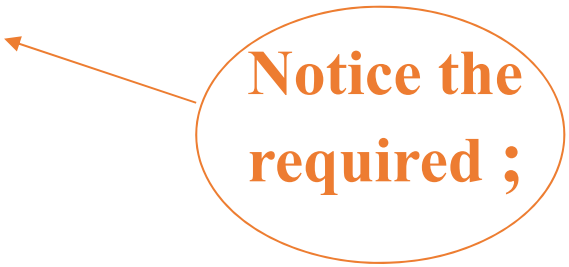
User Controls the Loop Example

```
int num, limit;
cout << "Table of squares\n";
cout << "How high to go? ";
cin  >> limit;
cout << "\n\nnumber square\n";
num = 1;
while (num <= limit)
{  cout << setw(5) << num << setw(6)
        << num*num << endl;
   num++;
}
```

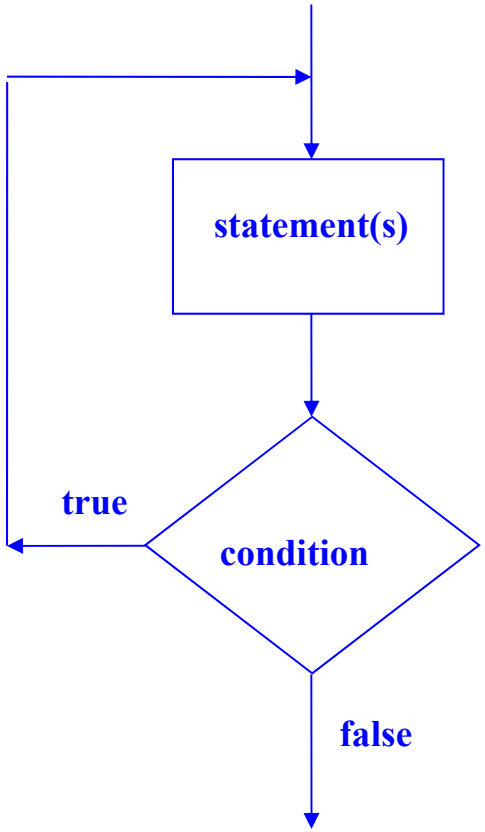
The do-while Loop

- **do-while**: a post test loop (***condition*** is evaluated after the loop executes)
- Format:

```
do
{
   1 or more statements;
} while (condition);
```



do-while Flow of Control



do-while Loop Notes

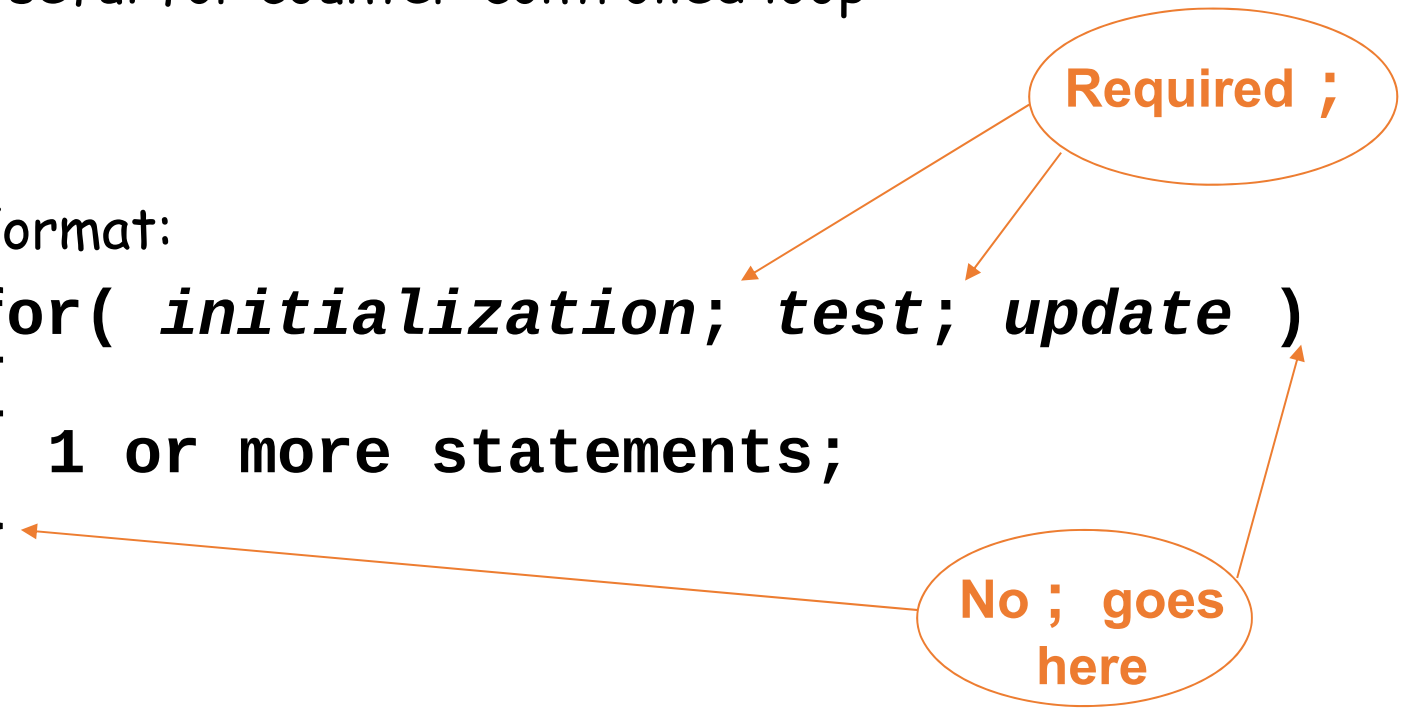
- Loop always executes at least once
- Execution continues as long as ***`condition`*** is **`true`**; the loop is exited when ***`condition`*** becomes **`false`**
- Useful in menu-driven programs to bring user back to menu to make another choice

The for Loop

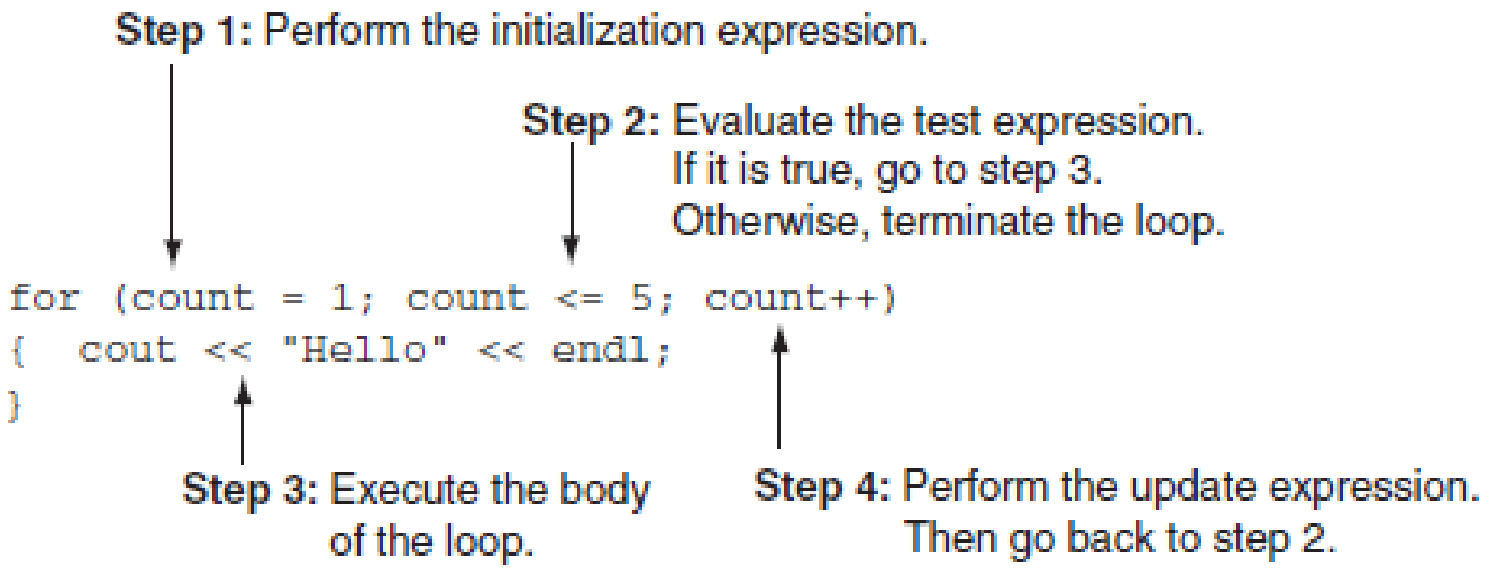
- Pretest loop that executes zero or more times
- Useful for counter-controlled loop

- Format:

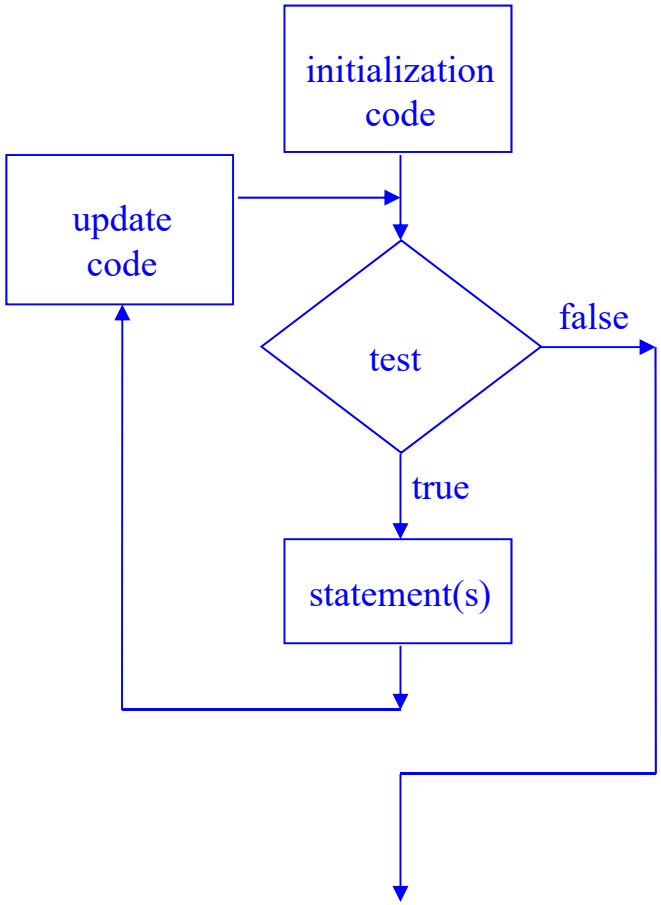
```
for( initialization; test; update )
{
  1 or more statements;
}
```



for Loop Mechanics



for Loop Flow of Control



for Loop Example

```
int sum = 0, num;

for (num = 1; num <= 10; num++)
  sum += num;

cout << "Sum of numbers 1 – 10 is "
     << sum << endl;
```

`for` Loop Notes

- If ***`test`*** is false the first time it is evaluated, the body of the loop will not be executed
- The update expression can increment or decrement by any amount

`for` Loop Modifications

- Can define variables in initialization code
  - Their scope is the **`for`** loop
- Initialization and update code can contain more than one statement
  - Separate statements with commas
- Example:

```
for (int sum = 0, num = 1; num <= 10; num++)
   sum += num;
```

More **for** Loop Modifications (These are NOT Recommended)

- Can omit ***initialization*** if already done

```
int sum = 0, num = 1;
for (; num <= 10; num++)
   sum += num;
```

- Can omit ***update*** if done in loop

```
for (sum = 0, num = 1; num <= 10;)
   sum += num++;
```

- Can omit ***test*** – may cause an infinite loop

```
for (sum = 0, num = 1; ; num++)
   sum += num;
```

- Can omit loop body if all work is done in header

Keeping a Running Total

- Sum of numbers from each repetition of loop
- accumulator: variable that holds running total

```
int sum = 0, num = 1; // sum is the
while (num <= 10)     // accumulator
{
   sum += num;
   num++;
}
cout << "Sum of numbers 1 - 10 is "
     << sum << endl;
```

Sentinels

- sentinel: value in a list of values that indicates end of data
- Special value that cannot be confused with a valid value, *e.g.*, **-999** for a test score
- Used to terminate input when user may not know how many values will be entered

Sentinel Example

```
int total = 0;
cout << "Enter points earned "
     << "(or -1 to quit): ";
cin  >> points;

while (points != -1) // -1 is the sentinel
{
   total += points;
   cout << "Enter points earned: ";
   cin  >> points;
}
```

Deciding Which Loop to Use

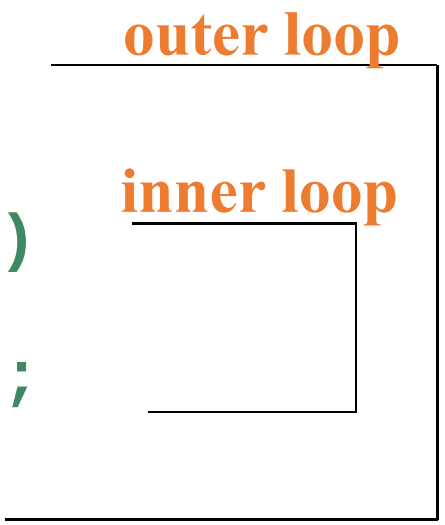
- `while`: pretest loop (loop body may not be executed at all)
- `do-while`: post test loop (loop body will always be executed at least once)
- `for`: pretest loop (loop body may not be executed at all); has initialization and update code; is useful with counters or if precise number of repetitions is known

Nested Loops

- A nested loop is a loop inside the body of another loop

- Example:

```
for (row = 1; row <= 3; row++)
{
   for (col = 1; col <= 3; col++)
   {
      cout << row * col << endl;
   }
}
```



Notes on Nested Loops

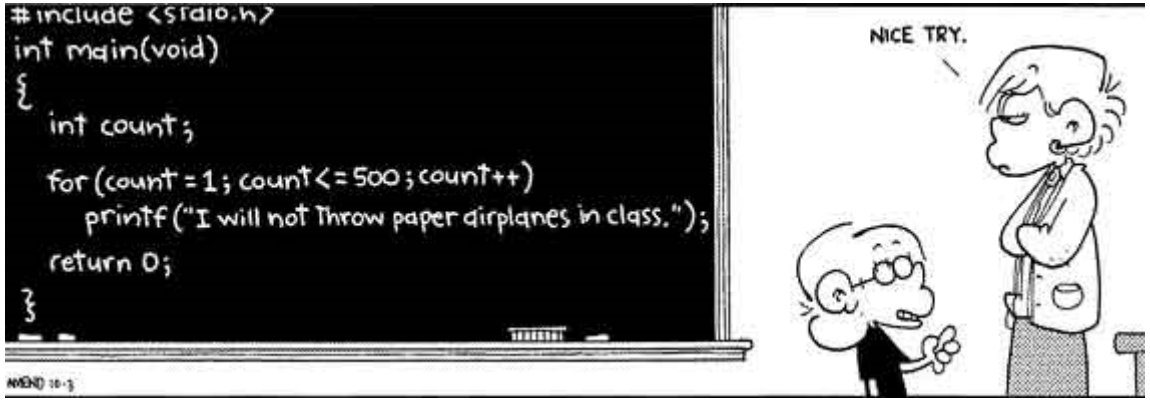
- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times

Breaking Out of a Loop

- Can use **`break`** to terminate execution of a loop
- Use sparingly if at all – makes code harder to understand
- When used in an inner loop, terminates that loop only and returns to the outer loop

The **`continue`** Statement

- Can use **`continue`** to go to end of loop and prepare for next repetition
  - **`while`** and **`do-while`** loops go to test and repeat the loop if test condition is true
  - **`for`** loop goes to update step, then tests, and repeats loop if test condition is true
- Use sparingly – like **`break`**, can make program logic hard to follow


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf ("I will not Throw paper airplanes in class.");
return 0;
}
NICE TRY.