Control Statements in C

MODULE III

Control Statements

- the instructions were executed in the same order in which they appeared within the program.
- Each instruction was executed once and only once.
- may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known **as** ***branching***.
- There is also a special kind of branching, called ***selection***, in which one group of statements is selected from several available groups.
- the program may require that **a** group of instructions be executed repeatedly, until some logical condition has been satisfied. **(Looping)**

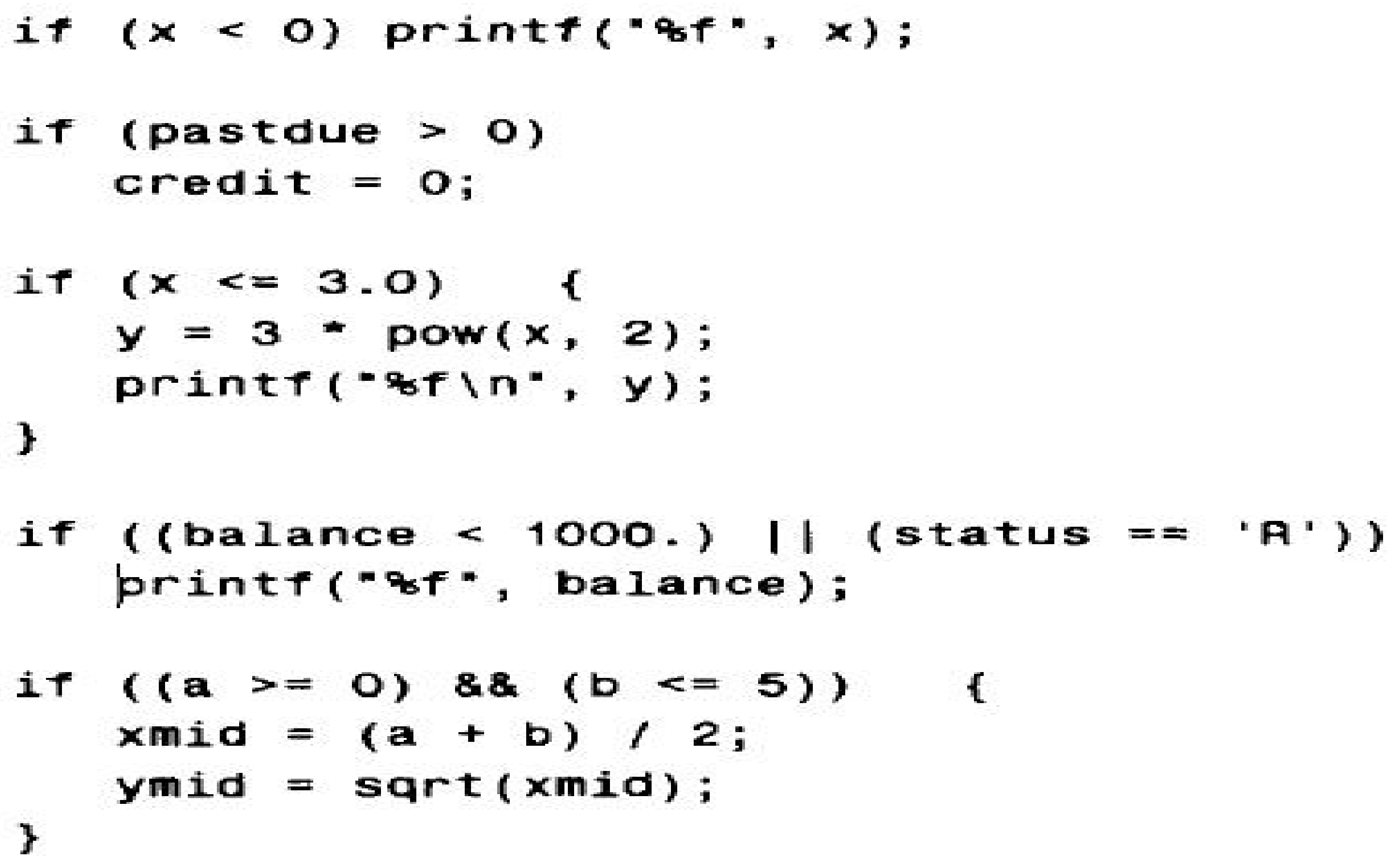
BRANCHING: THE if - else STATEMENT

- is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).
- The **else** portion of the **if - else** statement is optional.
- its simplest general form, the statement can be written as

  if (***expression) statement***

- The ***expression*** must be placed in parentheses.
- the ***statement*** will be executed only if the ***expression*** has a nonzero value (i.e., if ***expression*** is true). If the ***expression*** has a value of zero (i.e., if ***expression*** is false), then the ***statement*** will be ignored.
- The ***statement*** can be either simple or compound.(a compound statement which may include other control statements)

BRANCHING: simple if STATEMENT EXAMPLE



```
if (x < 0) printf("%f", x);

if (pastdue > 0)
    credit = 0;

if (x <= 3.0)   {
    y = 3 * pow(x, 2);
    printf("%f\n", y);
}

if ((balance < 1000.) || (status == 'R'))
    printf("%f", balance);

if ((a >= 0) && (b <= 5))   {
    xmid = (a + b) / 2;
    ymid = sqrt(xmid);
}
```

BRANCHING: THE if - else STATEMENT

- is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).
- The **else** portion of the **if** - **else** statement is optional.
- its simplest general form, the statement can be written as

  if (***expression) statement***
- The ***expression*** must be placed in parentheses.
- the ***statement*** will be executed only if the ***expression*** has a nonzero value (i.e., if ***expression*** is true). If the ***expression*** has a value of zero (i.e., if ***expression*** is false), then the ***statement*** will be ignored.
- The ***statement*** can be either simple or compound.(a compound statement which may include other control statements)

BRANCHING: THE if - else STATEMENT

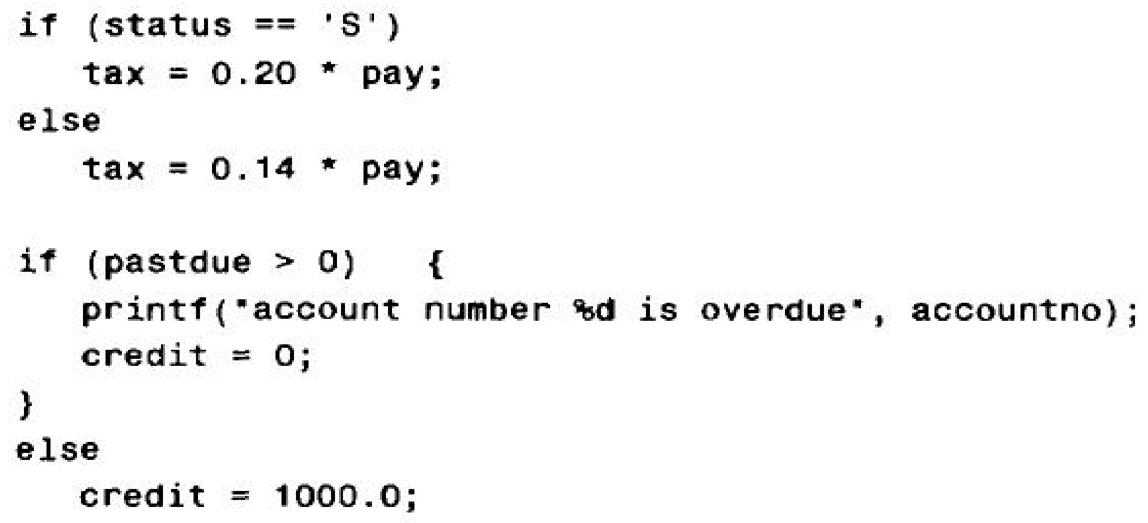
- The general form of an **if** statement which includes the **else** clause is

  if *(expression)*

  statement **1**

  else *statement 2*

- If the ***expression*** has a nonzero value (i.e., if ***expression*** is true), then ***statement 1*** w
  expression is false),



```
if (status == 'S')
   tax = 0.20 * pay;
else
   tax = 0.14 * pay;

if (pastdue > 0)   {
   printf("account number %d is overdue", accountno);
   credit = 0;
}
else
   credit = 1000.0;
```

nested (i.e., embed) **if** - **else**

- It is possible to nest (i.e., embed) **if** - **else** statements, one within another

  The most general form of two-layer nesting is

```
if e1 if e2 s1
          else s2
else if e3 s3
        else s4
```

where e **l,** ***e2*** and ***e3*** represent logical expressions and **s1, s2,s3** and ***s4*** represent statements

Some other forms of two-layer nesting are

```
1. if e1 s1
   else if e2 s2
2. if e1 s1
   else    if e2 s2
            else s3
3. if e1 if e2 s1
           else s2
else s3
4. if e1 if e2 s1
           else s2
```

The rule is that the **else** clause is always associated with the closest preceding unmatched (i.e., **else-less)** `if`.

4 is equivalent to

```
if e1 {
if e2 s1 else s2
}
```

multiple **if** - **else** statements

```
if e1 s1
else if e2 s2
     else if e3 s3
          else if e4 s4
               else s5
```

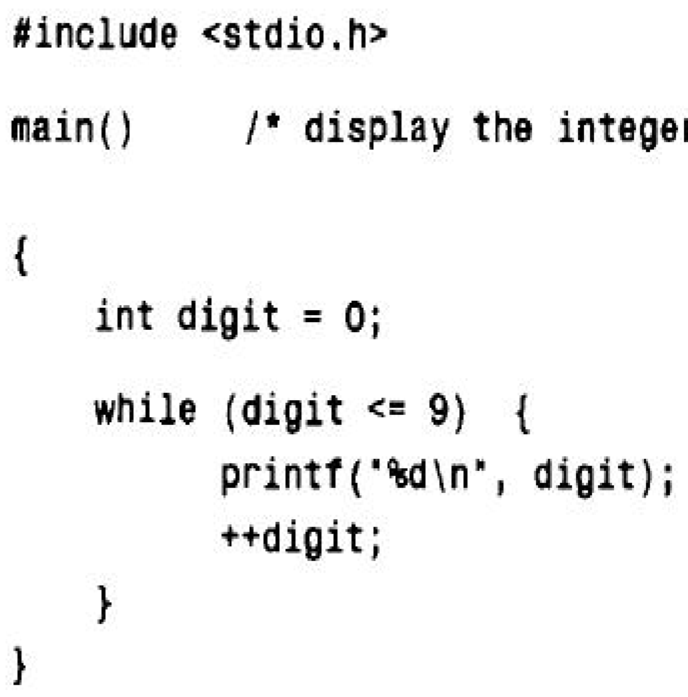
LOOPING: THE while STATEMENT

- The **while** statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.
- The general form of the while statement is

  while (*expression) statement*

- The ***statement*** will be executed repeatedly, as long as the ***expression*** is true (ie., as long ***expression*** has a nonzero value).
- This ***statement*** **can** be simple or compound

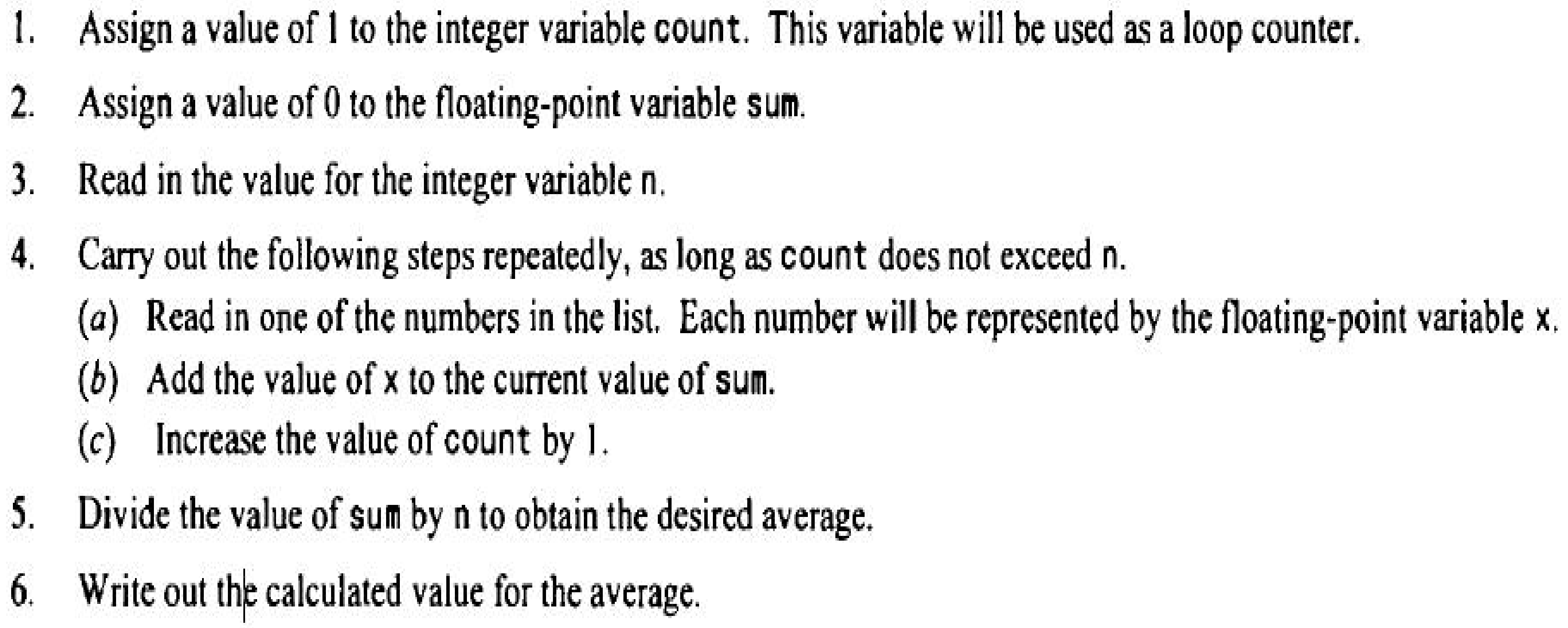
Suppose we want to display the consecutive digits 0, **1, 2, . . . ,9,** with one digit on each line.



```
#include <stdio.h>

main()     /* display the integers 0 through 9 */

{
    int digit = 0;

    while (digit <= 9)  {
          printf("%d\n", digit);
          ++digit;
    }
}
```

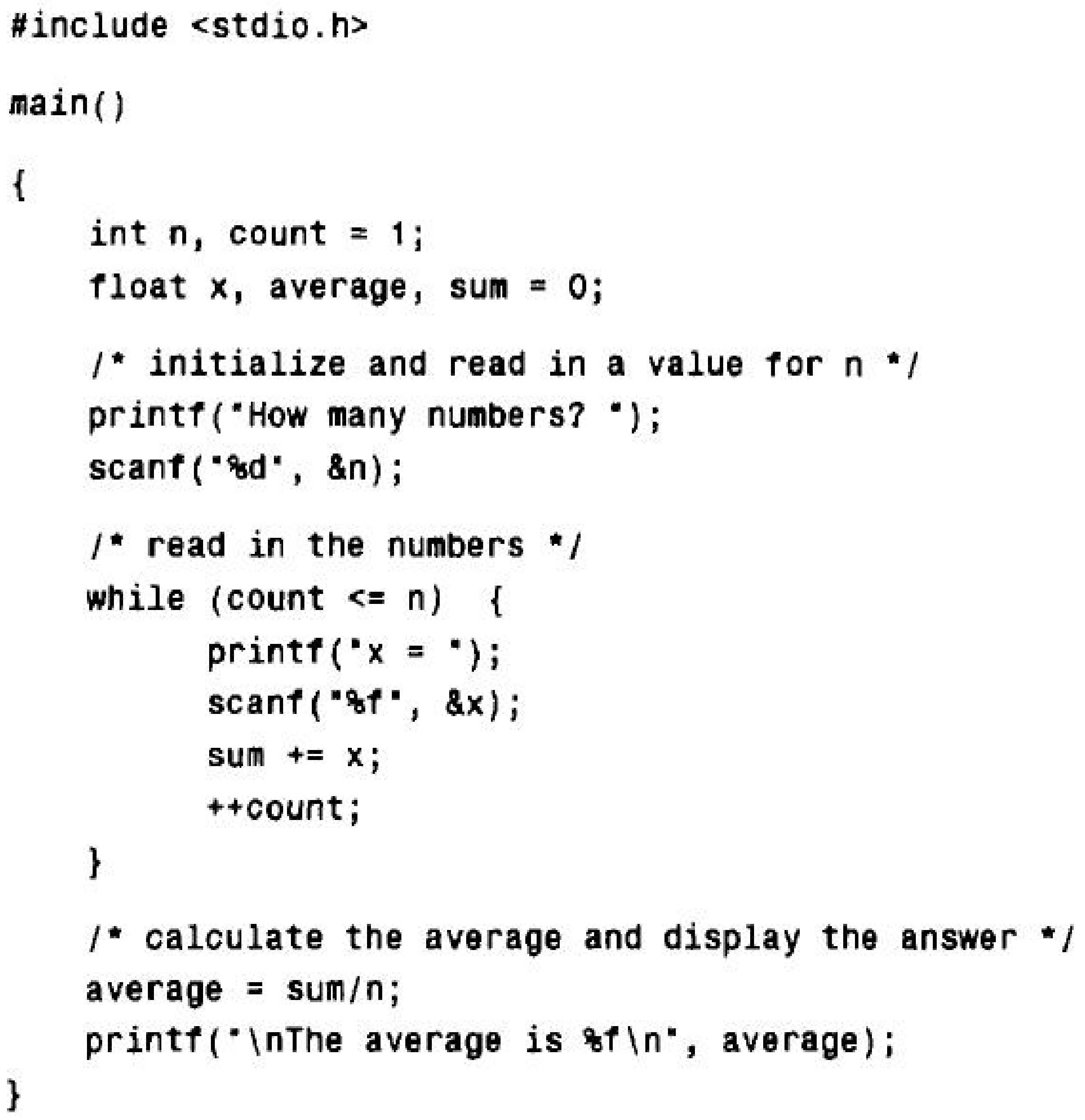
printf ("%d\n", digit++) ;

To calculate the average of a list of **n** numbers ALGORITHM:



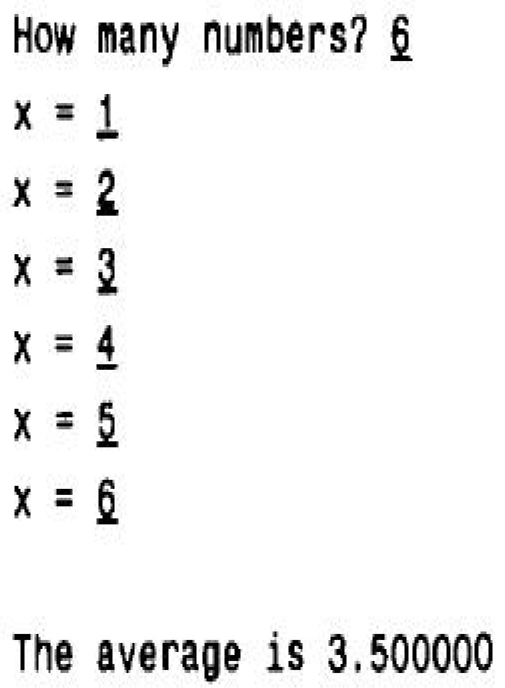
1. Assign a value of 1 to the integer variable count. This variable will be used as a loop counter.
2. Assign a value of 0 to the floating-point variable sum.
3. Read in the value for the integer variable n.
4. Carry out the following steps repeatedly, as long as count does not exceed n.
   (*a*) Read in one of the numbers in the list. Each number will be represented by the floating-point variable x.
   (*b*) Add the value of x to the current value of sum.
   (*c*) Increase the value of count by 1.
5. Divide the value of sum by n to obtain the desired average.
6. Write out the calculated value for the average.

PROGRAM



```
#include <stdio.h>

main()

{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    while (count <= n)  {
          printf("x = ");
          scanf("%f", &x);
          sum += x;
          ++count;
    }

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

OUTPUT



```
How many numbers? 6

x = 1

x = 2

x = 3

x = 4

x = 5

x = 6


The average is 3.500000
```

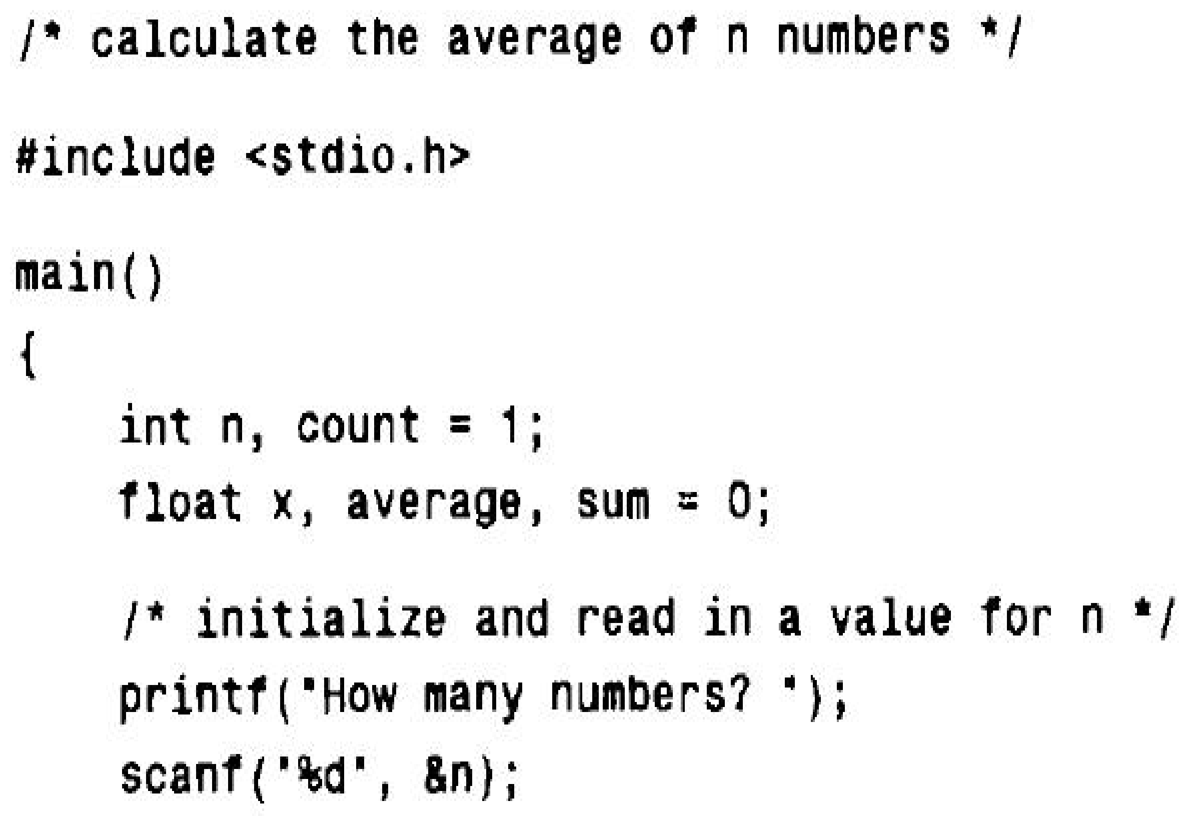
THE do - while STATEMENT

- The general form of the **do** - **while** statement is

  do *statement* while *(expression)*;
- the test for continuation of the loop is carried out at the ***end*** of each pass
- The ***statement*** will be executed repeatedly, as long **as** the value of ***expression*** is true (i.e., is nonzero).
- Notice that ***statement*** will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop.
- The ***statement*** can be either simple or compound

```
#include <stdio.h>

main()    /* display the integers 0 through 9 */
{
    int digit = 0;

    do
    printf("%d\n", digit++);
    while (digit <= 9);
}
```

Calculate the average of n numbers



```
/* calculate the average of n numbers */

#include <stdio.h>

main()
{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);
    /* read in the numbers */
    do {
          printf("x = ");
          scanf("%f", &x);
          sum += x;
          ++count;
    }  while (count <= n);

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

LOOPING: THE for STATEMENT

-

The general form of the **for** statement is
for (***expression*** **1; *expression*** 2; ***expression* 3) *statement***

where ***expression 1*** is used to initialize some parameter (called an ***index)*** that controls the looping action,

expression 2 represents a condition that must be true for the loop to continue execution, and

expression **3** is used to alter the value of the parameter initially assigned by ***expression 1.***

Typically, ***expression 1*** is an assignment expression, ***expression*** 2 is a logical expression and ***expression*** **3** is a unary expression or an assignment expression

for loops are generally used when the number of passes ***is*** known in advance

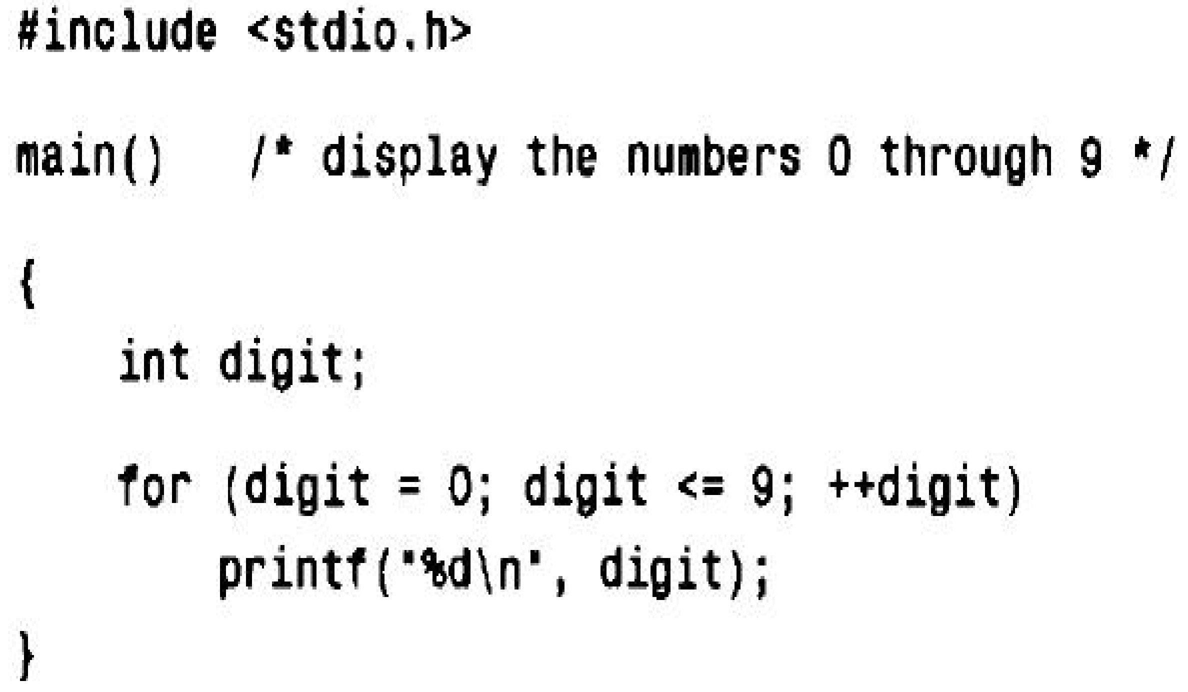
- When the **for** statement is executed, ***expression*** 2 is evaluated and tested at the ***beginning*** of each pass through the loop, and ***expression 3*** is evaluated at the ***end*** of each pass.
- Thus, the **for** statement

```
for (expression 1; expression 2; expression 3) statement
```

is equivalent to

```
expression 1;
while (expression 2)  {
    statement
    expression 3;
}
```

Display the numbers 0 through 9



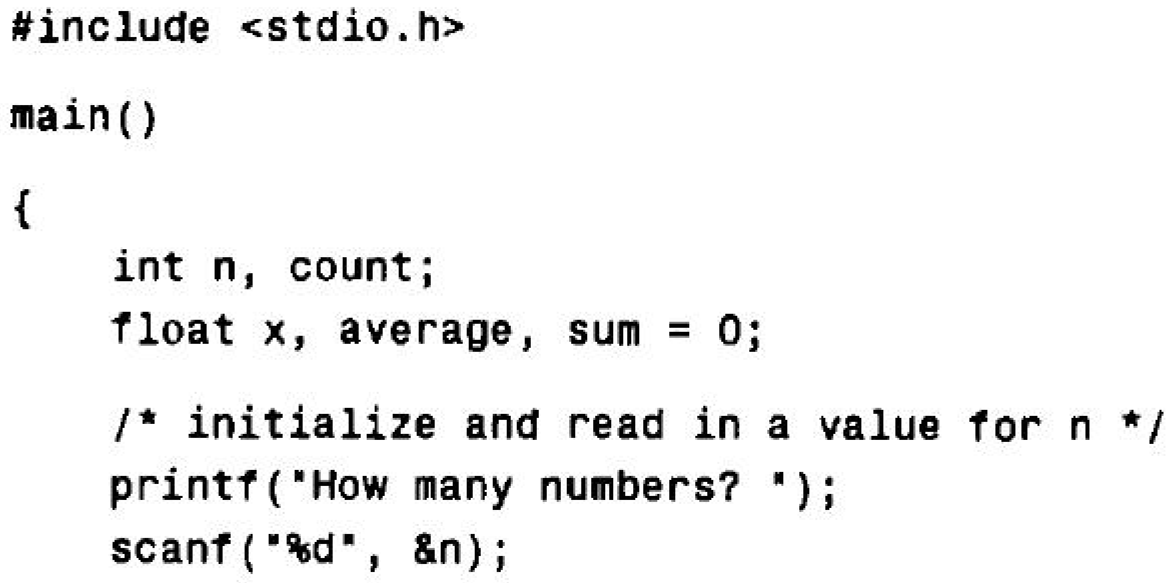
```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit;

    for (digit = 0; digit <= 9; ++digit)
        printf("%d\n", digit);
}
```

```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit = 0;

    for (; digit <= 9; )
        printf("%d\n", digit++);
}
```

Averaging a List of Numbers

1. Assign a value of 0 to the floating-point variable `sum`.
2. Read in a value for the integer variable `n`.
3. Assign a value of 1 to the integer variable `count`, where `count` is an index that counts the number of passes through the loop.
4. Carry out the following steps repeatedly, as long as the value of `count` does not exceed `n`.
   - (*a*) Read in one of the numbers in the list. Each number will be represented by the floating-point variable `x`.
   - (*b*) Add the value of `x` to the current value of `sum`.
   - (*c*) Increase the value of `count` by 1.
5. Divide the value of `sum` by `n` to obtain the desired average.
6. Write out the calculated value for the average.

Write the program ..



```
#include <stdio.h>

main()

{
     int n, count;
     float x, average, sum = 0;

     /* initialize and read in a value for n */
     printf("How many numbers? ");
     scanf("%d", &n);
```

```
    /* read in the numbers */
    for (count = 1; count <= n; ++count)  {
        printf("x = ");
        scanf("%f", &x);
        sum += x;
    }

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

NESTED CONTROL STRUCTURES

- Loops, like **if** - **else** statements, can be nested, one within another.
- It is essential, however, that one loop be completely embedded within the other -there can be no overlap.
- Each loop must be controlled by a different index.
- The inner and outer loops need not be generated by the same type of control structure.
- Nested control structures can involve both loops and **if** - **else** statements. Thus, a loop can be nested within an **if** - **else** statement, and **an if** - **else** statement can be nested within a loop.

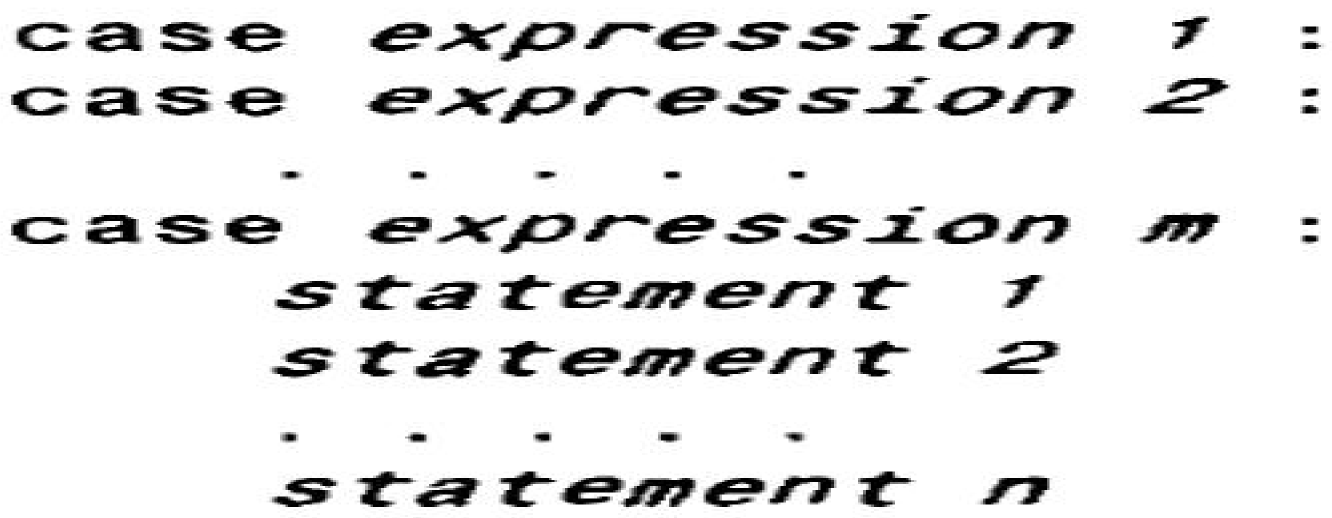
THE switch STATEMENT

- The **switch** statement causes a particular group of statements to be chosen from several available groups,
- The selection is based upon the current value of an expression which is included within the **switch** statement.
- The general form of the **switch** statement is

  switch *(expression) statement*

- where ***expression*** results in an integer value.
- Note that ***expression*** may also be of type char, since individual characters have equivalent integer values.
- The embedded ***statement*** is generally a compound statement that specifies alternate courses of action

The statement within the group must be preceded by one or more **case** labels (also called ***case*** prefixes)



```
case expression 1 :
case expression 2 :
     . . . . .
case expression m :
     statement 1
     statement 2
     . . . . .
     statement n
```

- When the **switch** statement is executed, the ***expression*** is evaluated and control is transferred directly to the group of statements whose case-label value matches the value of the ***expression***.
- If none of the case label values matches the value of the ***expression***, then none of the groups within the **switch** statement will be selected.
- In this case control is transferred directly to the statement that follows the **switch** statement.

Example

```
switch (choice = getchar())  {

case 'r':
case 'R':
     printf("RED");
     break;

 case  'w':
 case  'W':
       printf("WHITE");
       break;

 case  'b':
 case  'B':
       printf("BLUE");
 }
```

The **break** statement causes control to be transferred out of the **switch** statement, thus preventing more than one group of statements from being executed

Case label -default

- One of the labeled groups of statements within the **switch** statement may be labeled **default.**
- This group will be selected if none of the case labels matches the value of the ***expression***.
- The **default** group may appear anywhere within the **switch** statement-it need not necessarily be placed at the end.
- **If** none **of** the case labels matches the value of the ***expression*** and the **default** group is not present **(as** in the above example), then no action will be taken by the **switch** statement

```
switch (choice = toupper(getchar())

     case 'R':
           printf("RED");       {
           break;

     case 'W':
           printf("WHITE");
           break;

     case 'B':
           printf("BLUE");
           break;

     default:
           printf("ERROR");
     }
```

if none of the case labels matches the original ***expression***
ERROR will be displayed.

The **switch** statement may be thought of as an alternative to the use of nested **if** -else statements

```
switch (flag)   {

case -1:
        y = abs(x);
        break;

case 0:
        y = sqrt(x);
        break;

case 1:
        y = x;
        break;

case 2:
case 3:
        y = 2 * (x - 1);
        break;

default:
        y = 0;
}
```

THE break STATEMENT

- The break statement is used to terminate loops or to exit from a switch.
- It can be used within a for, while, do -while, or switch statement.
- The break statement is written simply **as**

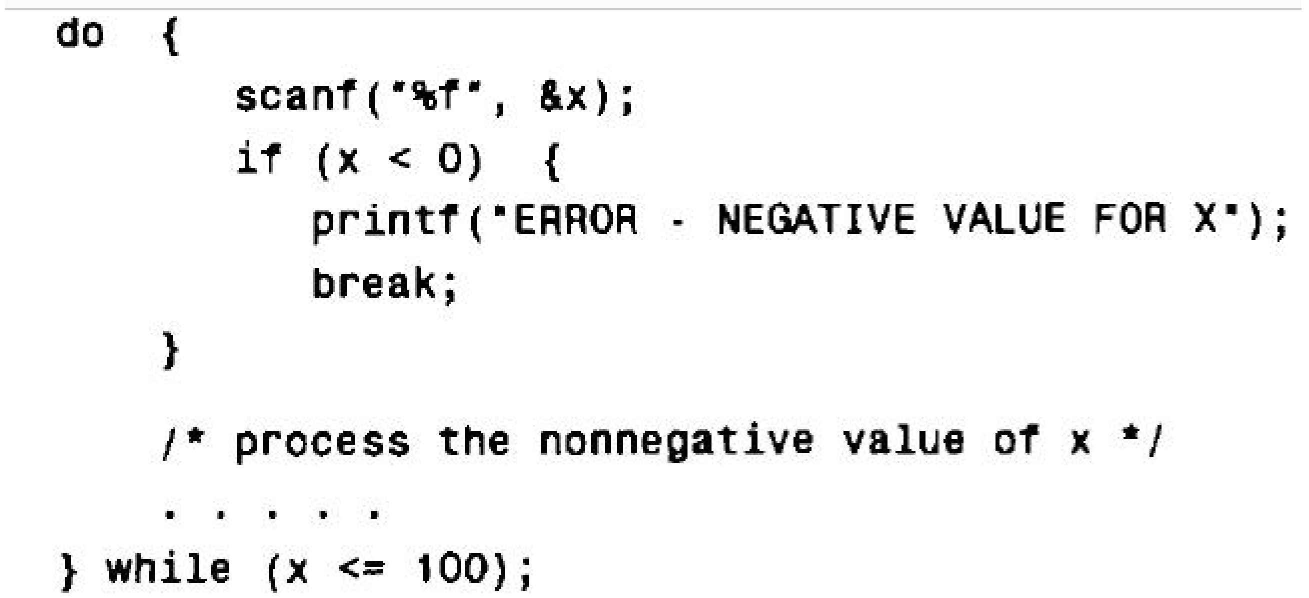
        break;   (without **any** embedded expressions or statements)

- The use of break statement in the switch statement transfer of control out of the entire switch statement, to the first statement following the switch statement.

Use of break statement in while loop

```
scanf("%f", &x);
while (x <= 100)  {
      if (x < 0)    {
         printf("ERROR - NEGATIVE VALUE FOR X");
         break;
      }

      /* process the nonnegative value of x */

      . . . . .
      scanf("%f", &x);
}
```

Use of break statement in do-while loop



```
do  {
        scanf("%f", &x);
        if (x < 0)  {
            printf("ERROR - NEGATIVE VALUE FOR X");
            break;
    }

    /* process the nonnegative value of x */

    . . . . .
} while (x <= 100);
```

Use of break statement in for loop

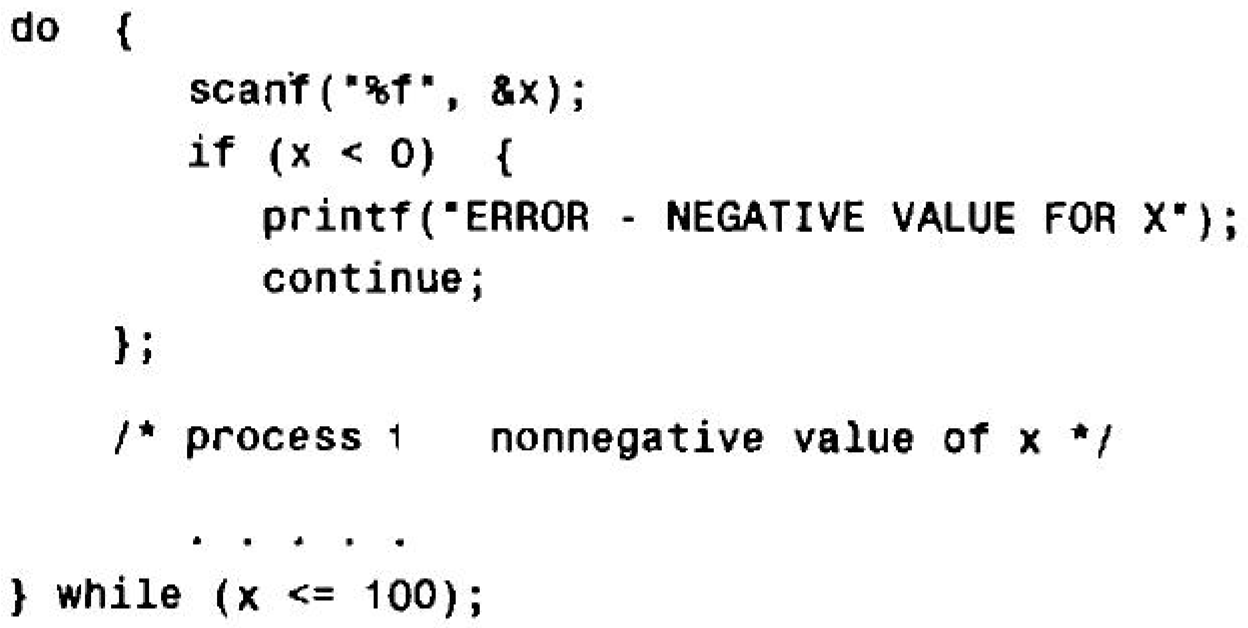
```
for (count = 1; x <= 100; ++count)   {
     scanf("%f", &x);
     if (x < 0)  {
        printf("ERROR - NEGATIVE VALUE FOR X");
        break;
     }

     /* process the nonnegative value of x */

     . . . . .
}
```

THE continue STATEMENT

- The **continue** statement is used to ***bypass*** the remainder of the current pass through a loop.
- The loop does ***not*** terminate when a **continue** statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.
- The **continue** statement can be included within a **while,** a **do - while** or a **for** statement. It is written simply **as**

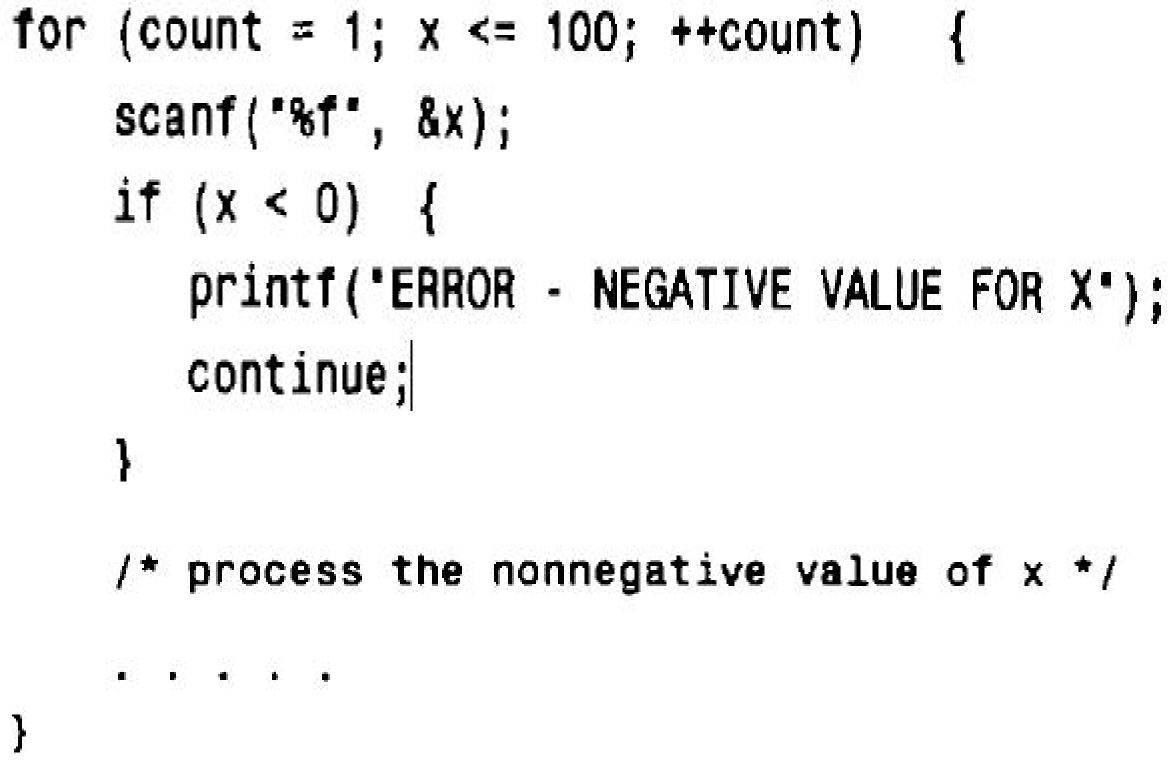
continue; without any embedded statements or expressions

Use of continue statement in do-while loop



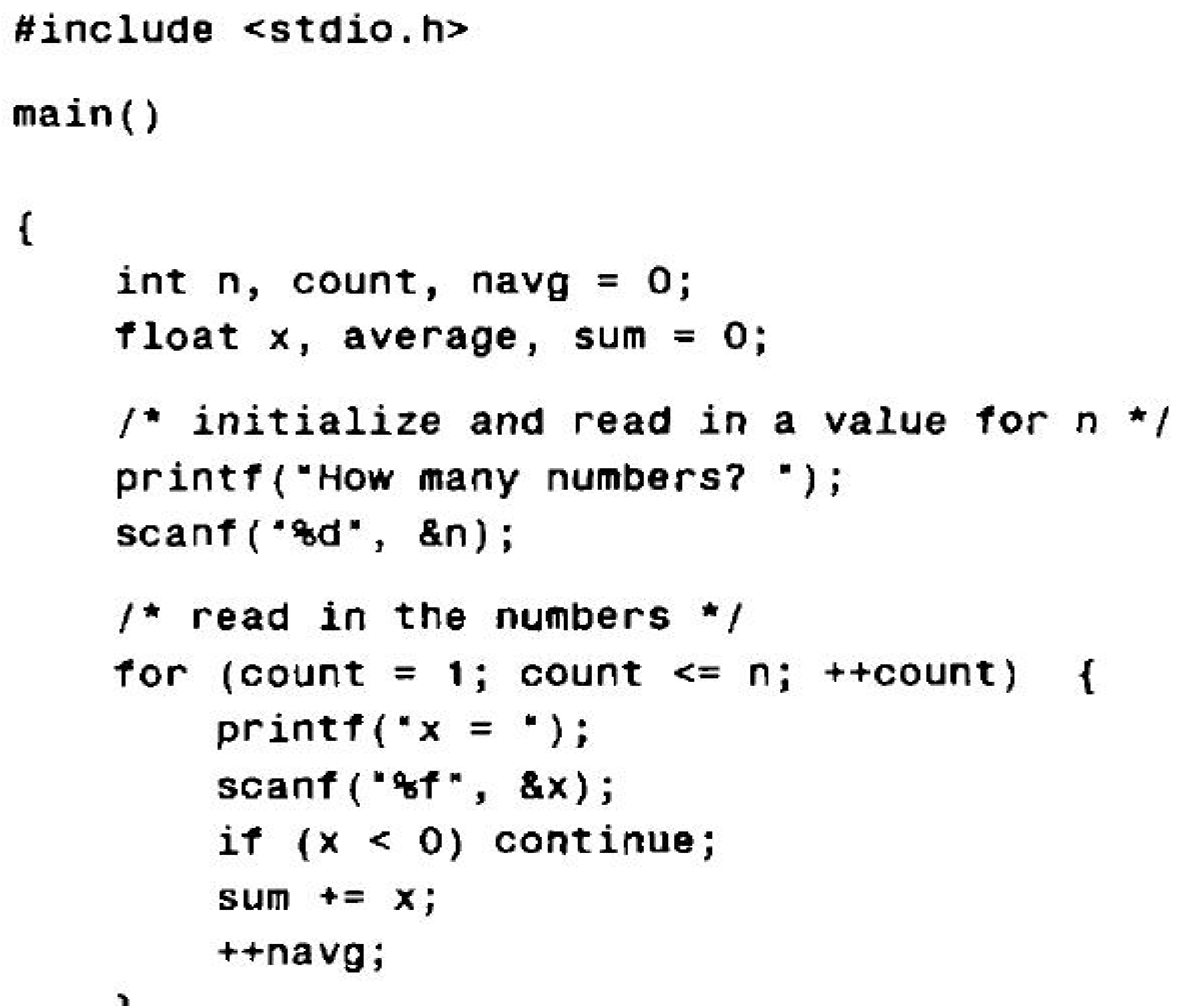
```
do  {
        scanf("%f", &x);
        if (x < 0)  {
           printf("ERROR - NEGATIVE VALUE FOR X");
           continue;
    };

    /* process 1   nonnegative value of x */

        . . . . .
} while (x <= 100);
```

Use of continue statement in for loop



```
for (count = 1; x <= 100; ++count)   {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - NEGATIVE VALUE FOR X");
       continue;
    }

    /* process the nonnegative value of x */

    . . . . .
}
```

Program to calculate average of n numbers. Skip the negative values from calculation of sum if the user enter it.



```
#include <stdio.h>

main()

{
    int n, count, navg = 0;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    for (count = 1; count <= n; ++count)  {
        printf("x = ");
        scanf("%f", &x);
        if (x < 0) continue;
        sum += x;
        ++navg;
    /* calculate the average and write out the answer */
    average = sum/navg;
    printf("\nThe average is %f\n", average);
}
```

OUTPUT:

```
How many numbers? 6
x = 1
x = -1
x = 2
x = -2
x = 3
x = -3

The average is 2.000000
```

THE COMMA OPERATOR (,)

for (*expression 1a*, *expression 1b*; *expression 2*; *expression 3*) *statement*

CONTD..

- Similarly, a **for** statement might make use of the comma operator in the following manner.

for (***expression*** **1;** ***expression*** **2;** ***expression 3a*****,** ***expression 3b)*** ***statement***

- The two separate expressions would typically be used to alter (e.g., increment or decrement) two different indices that are used simultaneously within the loop.
- the comma operator accepts two distinct expressions **as** operands. These expressions will be evaluated from left to right
- Within the collection of C operators, the comma operator has the **lowest precedence**. Its associativity is left to right.

THE goto STATEMENT

- The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program.
- In its general form, the **goto** statement is written as

goto ***label*;**

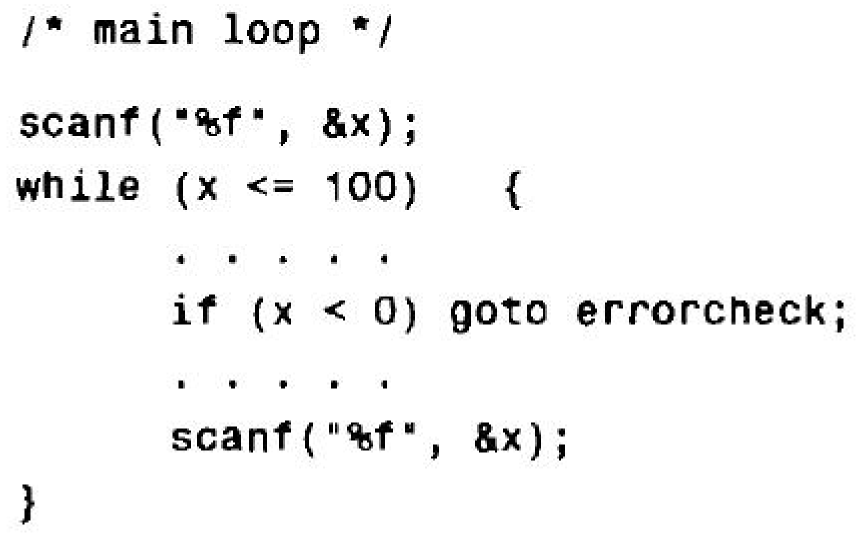
- where ***label*** is an identifier that is used to label the target statement to which control will be transferred.
- The target statement must be labeled, and the label must be followed by a colon. Thus, the target statement will appear as

label: statement

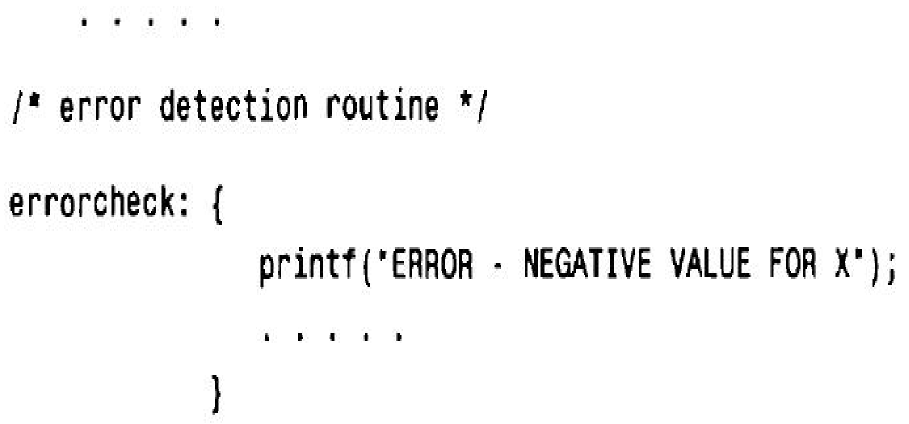
THE goto STATEMENT CONTD…

- Control may be transferred to any other statement within the program.
- Each labelled statement within the program must have a unique label; i.e., no two statements can have the same label.

EXAMPLE



```
/* main loop */

scanf("%f", &x);
while (x <= 100)   {
        . . . . .
        if (x < 0) goto errorcheck;
        . . . . .
        scanf("%f", &x);
}
```



```
   . . . . .

/* error detection routine */

errorcheck: {
            printf("ERROR - NEGATIVE VALUE FOR X");
            . . . . .
          }
```

The same thing could have been accomplished using the **break** statement

- Modern programming practice discourages its use.
- was used extensively, in early versions of some older languages, such as Fortran and BASIC. The most common applications were:
- 1. Branching around statements or groups of statements under certain conditions.
- **2.** Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass.
- **3.** Jumping completely out of a loop under certain conditions, thus terminating the execution of a loop.

The structured features in C enable all of these operations to be carried out without resorting to the **goto** statement.

- For example, branching around statements can be accomplished with the **if** - **else** statement;
- jumping to the end of a loop can be carried out with the **continue** statement; and jumping out of a loop iseasily accomplished using the **break** statement.
- The use of **goto** tends to encourage (or at least, not discourage) logic that skips all over the program whereas the structured features in C require that the entire program be written in an orderly, sequential manner.
- For this reason, ***use of the* goto *statement should generally be avoided***.

Use of goto statement

- Consider, for example, a situation in which it is necessary to jump out of a doubly nested loop if a certain condition is detected. This can be accomplished with two **if - break** statements, one within each loop, though this is awkward.
- A better solution in this particular situation might make use of the **goto** statement to transfer out of both loops at once.

Assignments

- 1. Write a program in C to display numbers from 1 to 100.
- 2. Write a program in C to display numbers from m to n. The values for m and n are given by user.
- 3. Write a program in C to sum all numbers from m to n. The values for m and n are given by user.