Control Structures in C

(Part 2)

Dr Bhanu

Repetition Essentials

- A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.
- We have two types of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition
- Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
- Sentinel-controlled repetition is sometimes called indefinite repetition because it's not known in advance how many times the loop will be executed.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Repetition Essentials

- In counter-controlled repetition, a control variable is used to count the number of repetitions.
- The control variable is incremented (usually by 1) each time the group of instructions is performed.
- When the value of the control variable indicates that the correct number of repetitions has been performed, the loop terminates and the computer continues executing with the statement after the repetition statement.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

```c
#include <stdio.h>

int main () {

  /* Initialization */
  int a = 10;

  /* while loop execution */
  while( a < 20 ) {

    printf("value of a: %d\n", a);

    //Updating
    a = a +  1;
  }

  return 0;
}
```

Counter-controlled repetition requires:

The name of a control variable (or loop counter).

The initial value of the control variable.

The increment (or decrement) by which the control variable is modified each time through the loop.

The condition that tests for the final value of the control variable (i.e., whether looping should continue).

Repetition Essentials

- Sentinel values are used to control repetition when:
  - The precise number of repetitions is not known in advance, and
  - The loop includes statements that obtain data each time the loop is performed.
- The sentinel value indicates “end of data.”
- The sentinel is entered after all regular data items have been supplied to the program.
- Sentinels must be distinct from regular data items.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Example

Write a C program to calculate the average of arbitrary number of marks entered by the user, with the help of a sentinel controlled loop.

```c
#include <stdio.h>

int main()

{
   int guard = 0;
   int cnt = 0;
   float marks;
   float sum = 0;
   float aver;

   while (guard != -1)
   {
      printf("Enter the marks \t");
      scanf("%f", &marks);
      printf("\nThe marks entered are : %f",
marks);
      sum = sum + marks;
      cnt = cnt + 1;

      printf("\nThe sum is  :  %f ", sum);
      printf("\nThe count is  :  %d ", cnt);

      printf("\ (enter any integer to continue else -1 to end   ");
      scanf("%d", &guard);
   }

   aver = sum / cnt;

   printf("\nThe average of %d marks is  :  %f ", cnt, aver);
   return 0;
```

Assignment Operators

- C provides several assignment operators for abbreviating assignment expressions.
- For example, the statement
    - `c = c + 3;`
- can be abbreviated with the addition assignment operator `+=` as
    - `c += 3;`
- The `+=` operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Assignment Operators

- Any statement of the form
    - *variable = variable operator expression;*
- where *operator is one of the binary operators +, -, *, / or %, can be written in the form*
    - *variable operator= expression;*
- Thus the assignment `c += 3` adds `3` to `c`.
- Figure shows the arithmetic assignment operators, sample expressions using these operators and explanations.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 3.11 | Arithmetic assignment operators.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Increment and Decrement Operators

- C also provides the unary increment operator, `++`, and the unary decrement operator, `--`.
- If a variable `c` is incremented by 1, the increment operator `++` can be used rather than the expressions `c = c + 1` or `c += 1`.
- If increment or decrement operators are placed before a variable (i.e., prefixed), they're referred to as the preincrement or predecrement operators, respectively.
- If increment or decrement operators are placed after a variable (i.e., postfixed), they're referred to as the postincrement or postdecrement operators, respectively.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Increment and Decrement Operators

- Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears.
- Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Operator	Sample expression	Explanation
`++`	`++a`	Increment a by 1, then use the new value of a in the expression in which a resides.
`++`	`a++`	Use the current value of a in the expression in which a resides, then increment a by 1.
`--`	`--b`	Decrement b by 1, then use the new value of b in the expression in which b resides.
`--`	`b--`	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 3.12 | Increment and decrement operators

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Increment and Decrement Operators

- Figure demonstrates the difference between the preincrementing and the postincrementing versions of the `++` operator.
- Postincrementing the variable `c` causes it to be incremented after it’s used in the `printf` statement.
- Preincrementing the variable `c` causes it to be incremented before it’s used in the `printf` statement.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

```c
/* Fig. 3.13: fig03_13.c
   Preincrementing and postincrementing */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int c; /* define variable */

   /* demonstrate postincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", c++ ); /* print 5 then postincrement */
   printf( "%d\n\n", c ); /* print 6 */

   /* demonstrate preincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", ++c ); /* preincrement then print 6 */
   printf( "%d\n", c ); /* print 6 */
   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 3.13 | Preincrementing vs. postincrementing. (Part 1 of 2.)

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Increment and Decrement Operators

- The program displays the value of `c` before and after the `++` operator is used.
- The decrement operator (`--`) works similarly.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Increment and Decrement Operators

- The three assignment statements in Fig. 3.10

  -
    ```
    passes = passes + 1;
    failures = failures + 1;
    student = student + 1;
    ```

  can be written more concisely with assignment operators as

  -
    ```
    passes += 1;
    failures += 1;
    student += 1;
    ```

  with preincrement operators as

  -
    ```
    ++passes;
    ++failures;
    ++student;
    ```

  or with postincrement operators as

  -
    ```
    passes++;
    failures++;
    student++;
    ```

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

`for` Repetition Statement

- The general format of the `for` statement is

  -
    ```
    for ( expression1 ; expression2 ; expression3 )
        statement
    ```

  where *expression1* initializes the loop-control variable, *expression2* is the loop-continuation condition, and *expression3* increments the control variable.
- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
expression1;
while ( expression2 ) {
    statement
    expression3;
}
```

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

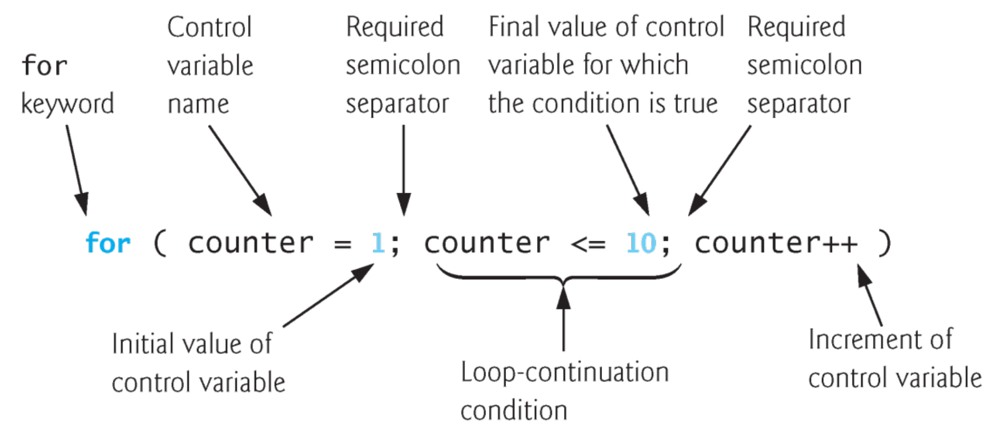
for keyword
Control variable name
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
for (counter = 1; counter <= 10; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. 4.3 | `for` statement header components.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

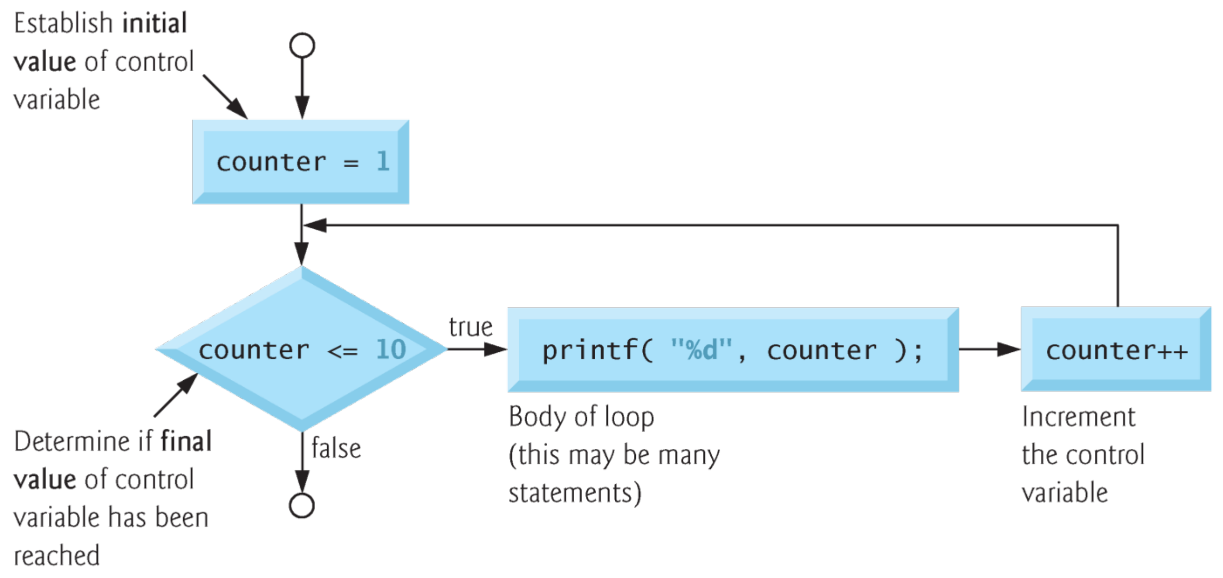
Establish **initial value** of control variable
counter = 1
counter <= 10
true
false
printf("%d", counter);
counter++
Determine if **final value** of control variable has been reached
Body of loop (this may be many statements)
Increment the control variable

Fig. 4.4 | Flowcharting a typical `for` repetition statement.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

`for` Repetition Statement

- The three expressions in the `for` statement are optional.
- If *expression2* is omitted, C assumes that the condition is true, thus creating an infinite loop.
- One may omit *expression1* if the control variable is initialized elsewhere in the program.
- *expression3* may be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed.
- The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

for Repetition Statement

- Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are all equivalent in the increment part of the `for` statement.

- Many C programmers prefer the form `counter++` because the incrementing occurs after the loop body is executed, and the postincrementing form seems more natural.
- Because the variable being preincremented or postincremented here does not appear in a larger expression, both forms of incrementing have the same effect.
- The two semicolons in the `for` statement are required.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

`for` Repetition Statement

- The initialization, loop-continuation condition and increment can contain arithmetic expressions. For example, if `x = 2` and `y = 10`, the statement

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

  is equivalent to the statement

```
for ( j = 2; j <= 80; j += 5 )
```

- If the loop-continuation condition is initially false, the loop body does not execute. Instead, execution proceeds with the statement following the `for` statement.

©1992-2010 by Pearson Education, Inc. All Rights Reserved.

Examples Using the `for` Statement

- The following examples show methods of varying the control variable in a `for` statement.
  - Vary the control variable from `1` to `100` in increments of `1`.
    ```
    for ( i = 1; i <= 100; i++ )
    ```
  - Vary the control variable from `100` to `1` in increments of `-1` (decrements of `1`).
    ```
    for ( i = 100; i >= 1; i-- )
    ```
  - Vary the control variable from `7` to `77` in steps of `7`.
    ```
    for ( i = 7; i <= 77; i += 7 )
    ```
  - Vary the control variable from `20` to `2` in steps of `-2`.
    ```
    for ( i = 20; i >= 2; i -= 2 )
    ```
  - Vary the control variable over the following sequence of values: `2`, `5`, `8`, `11`, `14`, `17`.
    ```
    for ( j = 2; j <= 17; j += 3 )
    ```
  - Vary the control variable over the following sequence of values: `44`, `33`, `22`, `11`, `0`.
    ```
    for ( j = 44; j >= 0; j -= 11 )
    ```

©1992-2010 by Pearson Education, Inc. All Rights Reserved.