Code	*Meaning*
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on value
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h for short integers
l for long integers or double
L for long double.

Table 4.4 *Commonly used* ***Output*** *Format Flags*

Flag	*Meaning*
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
#(with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
#(with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion.

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore the correctness and clarity of outputs are of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is presented. Following are some of the steps we can take to improve the clarity and hence the readability and understandability of outputs.

1. Provide enough blank space between two numbers.
2. Introduce appropriate headings and variable names in the output.
3. Print special messages whenever a peculiar condition occurs in the output.
4. Introduce blank lines between the important sections of the output.

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a 'tab' character between the specifications. For example, the statement

```
printf("a = %d\t b = %d", a, b);
```

will provide four blank spaces between the two fields. We can also print them on two separate lines by using the statement

```
printf("a = %d\n b = %d", a, b);
```

Messages and headings can be printed by using the character strings directly in the **printf** statement. Examples:

```
printf("\n OUTPUT RESULTS \n");
printf("Code\t Name\t Age\n");
printf("Error in input data\n");
printf("Enter your name\n");
```

Just Remember

- While using **getchar** function, care should be exercised to clear any unwanted characters in the input stream.
- Do not forget to include <**stdio.h**> headerfiles when using functions from standard input/output library.
- Do not forget to include <**ctype.h**> header file when using functions from character handling library.
- Provide proper field specifications for every variable to be read or printed.
- Enclose format control strings in double quotes.
- Do not forget to use address operator & for basic type variables in the input list of **scanf**.
- Use double quotes for character string constants.
- Use single quotes for single character constants.
- Provide sufficient field with to handle a value to be printed.
- Be aware of the situations where output may be imprecise due to formatting.
- Do not specify any precision in input field specifications.
- Do not provide any white-space at the end of format string of a **scanf** statement.
- Do not forget to close the format string in the **scanf** or **printf** statement with double quotes.
- Using an incorrect conversion code for data type being read or written will result in runtime error.
- Do not forget the comma after the format string in **scanf** and **printf** statements.
- Not separating read and write arguments is an error.
- Do not use commas in the format string of a **scanf** statement.
- Using an address operator & with a variable in the **printf** statement will result in runtime error.

CASE STUDIES

1. Inventory Report

Problem: The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below:

Code	*Quantity*	*Rate (Rs)*
F105	275	575.00
H220	107	99.95
I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	*Quantity*	*Rate*	*Value*
____	____	____	____
____	____	____	____
____	____	____	____
____	____	____	____
		Total Value:	____

The value of each item is given by the product of quantity and rate.

Program: The program given in Fig. 4.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 7.

```
Program
    #define ITEMS 4
    main()
    { /* BEGIN */
      int i, quantity[5];
      float rate[5], value, total_value;
      char code[5][5];
      /* READING VALUES */
      i = 1;
      while ( i <= ITEMS)
      {
        printf("Enter code, quantity, and rate:");
        scanf("%s %d %f", code[i], &quantity[i],&rate[i]);
        i++;
      }
    /*.......Printing of Table and Column Headings.......*/
      printf("\n\n");
```

```
        printf("      INVENTORY REPORT      \n");
        printf("----------------------\n");
        printf("  Code Quantity Rate Value  \n");
        printf("----------------------\n");
      /*.......Preparation of Inventory Position.....*/
        total_value = 0;
        i = 1;
        while ( i <= ITEMS)
        {
          value = quantity[i] * rate[i];
          printf("%5s %10d %10.2f %e\n",code[i],quantity[i],
            rate[i],value);
          total_value += value;
          i++;
        }
      /*.......Printing of End of Table.................*/
        printf("-----------------\n");
        printf("          Total Value = %e\n",total_value);
        printf("-----------------\n");
      } /* END */
```

Output

```
Enter code, quantity, and rate:F105 275 575.00
Enter code, quantity, and rate:H220 107  99.95
Enter code, quantity, and rate:I019 321 215.50
Enter code, quantity, and rate:M315 89  725.00
```

INVENTORY REPORT

Code	Quantity	Rate	Value
F105	275	575.00	1.581250e+005
H220	107	99.95	1.069465e+004
I019	321	215.50	6.917550e+004
M315	89	725.00	6.452500e+004
		Total Value =	3.025202e+005

Fig. 4.12 *Program for inventory report*

2. Reliability Graph

Problem: The reliability of an electronic component is given by

reliability (r) = $e^{-\lambda t}$

where λ is the component failure rate per hour and t is the time of operation in hours. A graph is required to determine the reliability at various operating times, from 0 to 3000 hours. The failure rate λ (lambda) is 0.001.

Problem

```
#include <math.h>
#define LAMBDA 0.001
main()
{
  double t;
  float r;
  int i, R;
  for (i=1; i<=27; ++i)
  {
    printf("--");
  }
  printf("\n");
  for (t=0; t<=3000; t+=150)
  {
    r = exp(-LAMBDA*t);
    R = (int)(50*r+0.5);
    printf(" |");
    for (i=1; i<=R; ++i)
    {
      printf("*");
    }
    printf("#\n");
  }
  for (i=1; i<3; ++i)
  {
    printf(" |\n");
  }
}
```

Output

```
-----------------------------------------------
|************************************************#
|******************************************#
|**********************************#
|*****************************#
|************************#
|*********************#
|******************#
|***************#
|*************#
|***********#
|**********#
|********#
|*******#
|******#
```

```
|*****#
|*****#
|****#
|***#
|***#
|**#
```

Fig. 4.13 *Program to draw reliability graph*

Program: The program given in Fig. 4.13 produces a shaded graph. The values of t are self-generated by the **for** statement

```
for (t=0; t <= 3000; t = t+150)
```

in steps of 150. The integer 50 in the statement

```
R = (int)(50*r+0.5)
```

is a scale factor which converts r to a large value where an integer is used for plotting the curve. Remember r is always less than 1.

REVIEW QUESTIONS

4.1 State whether the following statements are *true* or *false*.

(a) The purpose of the header file <studio.h> is to store the programs created by the users.
(b) The C standard function that receives a single character from the keyboard is **getchar**.
(c) The **getchar** cannot be used to read a line of text from the keyboard.
(d) The input list in a **scanf** statement can contain one or more variables.
(e) When an input stream contains more data items than the number of specifications in a **scanf** statement, the unused items will be used by the next **scanf** call in the program.
(f) Format specifiers for output convert internal representations for data to readable characters.
(g) Variables form a legal element of the format control string of a **printf** statement.
(h) The **scanf** function cannot be used to read a single character from the keyboard.
(i) The format specification %+−8d prints an integer left-justified in a field width of 8 with a plus sign, if the number is positive.
(j) If the field width of a format specifier is larger than the actual width of the value, the value is printed right-justified in the field.
(k) The print list in a **printf** statement can contain function calls.
(l) The format specification %5s will print only the first 5 characters of a given string to be printed.

4.2 Fill in the blanks in the following statements.

(a) The _____specification is used to read or write a short integer.
(b) The conversion specifier _____is used to print integers in hexadecimal form.
(c) For using character functions, we must include the header file _____ in the program.
(d) For reading a double type value, we must use the specification _____.
(e) The specification _____is used to read a data from input list and discard it without

assigning it to many variable.

(f) The specification _____ may be used in **scanf** to terminate reading at the encounter of a particular character.

(g) The specification %[ ] is used for reading strings that contain _____.

(h) By default, the real numbers are printed with a precision of _____decimal places.

(i) To print the data left-justified, we must use _____in the field specification.

(j) The specifier _____ prints floating-point values in the scientific notation.

4.3 Distinguish between the following pairs:

(a) ***getchar*** and ***scanf*** functions.

(b) %s and %c specifications for reading.

(c) %s and %[ ] specifications for reading.

(d) %g and %f specification for printing.

(e) %f and %e specifications for printing.

4.4 Write **scanf** statements to read the following data lists:

(a) 78 B 45

(b) 123 1.23 45A

(c) 15-10-2002

(d) 10 TRUE 20

4.5 State the outputs produced by the following **printf** statements.

(a) `printf ("%d%c%f", 10, 'x', 1.23);`

(b) `printf ("%2d %c %4.2f", 1234,, 'x', 1.23);`

(c) `printf ("%d\t%4.2f", 1234, 456);`

(d) `printf ("\"%08.2f\"", 123.4);`

(e) `printf ("%d%d %d", 10, 20);`

For questions 4.6 to 4.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

4.6 State errors, if any, in the following input statements.

(a) `scanf("%c%f%d", city, &price, &year);`

(b) `scanf("%s%d", city, amount);`

(c) `scanf("%f, %d, &amount, &year);`

(d) `scanf(\n"%f", root);`

(e) `scanf("%c %d %ld", *code, &count, Root);`

4.7 What will be the values stored in the variables **year** and **code** when the data

1988, x

is keyed in as a response to the following statements:

(a) `scanf("%d %c", &year, &code);`

(b) `scanf("%c %d", &year, &code);`

(c) `scanf("%d %c", &code, &year);`

(d) `scanf("%s %c", &year, &code);`

4.8 The variables **count, price**, and **city** have the following values:

count <—— 1275

price <—— –235.74

city <—— Cambridge

Show the exact output that the following output statements will produce:

(a) `printf("%d %f", count, price);`
(b) `printf("%2d\n%f", count, price);`
(c) `printf("%d %f", price, count);`
(d) `printf("%10dxxxx%5.2f",count, price);`
(e) `printf("%s", city);`
(f) `printf(%-10d %-15s", count, city);`

4.9 State what (if anything) is wrong with each of the following output statements:

(a) `printf(%d 7.2%f, year, amount);`
(b) `printf("%-s, %c"\n, city, code);`
(c) `printf("%f, %d, %s, price, count, city);`
(d) `printf("%c%d%f\n", amount, code, year);`

4.10 In response to the input statement

scanf("%4d%*%d", &year, &code, &count);

the following data is keyed in:

19883745

What values does the computer assign to the variables **year, code**, and **count**?

PROGRAMMING EXERCISES

4.1 Given the string "WORDPROCESSING", write a program to read the string from the terminal and display the same in the following formats:

(a) WORD PROCESSING
(b) WORD
PROCESSING
(c) W.P.

4.2 Write a program to read the values of x and y and print the results of the following expressions in one line:

(a) $\frac{x+y}{x-y}$ (b) $\frac{x+y}{2}$ (c) $(x+y)(x-y)$

4.3 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:

35.7 50.21 −23.73 −46.45

4.4 Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character * as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

```
*   *   *   *
*   *   *   *   4.36
*   *   *   *
```

Note that the actual values are shown at the end of each bar.

4.5 Write an interactive program to demonstrate the process of multiplication. The program should ask the user to enter two two-digit integers and print the product of integers as shown below.

```
                    45
                 x  37
                ------
7 x 45 is          315
3 x 45 is          135
                ------
Add  them         1665
                ------
```

Chapter

5

Decision Making and Branching

5.1 INTRODUCTION

We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision-making capabilities by supporting the following statements:

1. **if** statement
2. **switch** statement
3. Conditional operator statement
4. **goto** statement

These statements are popularly known as *decision-making statements*. Since these statements 'control' the flow of execution, they are also known as *control statements*.

We have already used some of these statements in the earlier examples. Here, we shall discuss their features, capabilities and applications in more detail.

5.2 DECISION MAKING WITH IF STATEMENT

The **if** statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

if (*test expression*)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or non-zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two *paths* to follow, one for the *true* condition and the other for the *false* condition as shown in Fig. 5.1.

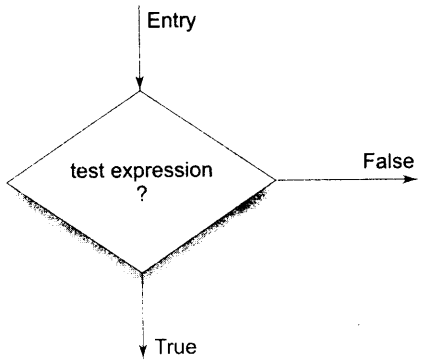


Fig. 5.1 *Two-way branching*

Some examples of decision making, using **if** statements are:

1. **if** (bank balance is zero)
   borrow money
2. **if** (room is dark)
   put on lights
3. **if** (code is 1)
   person is male
4. **if** (age is more than 55)
   person is retired

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if....else** statement
4. **else if** ladder.

We shall discuss each one of them in the next few sections.

5.3 SIMPLE IF STATEMENT

The general form of a simple **if** statement is

```
if (test expression)
 {
     statement-block;
 }
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illustrated in Fig. 5.2.

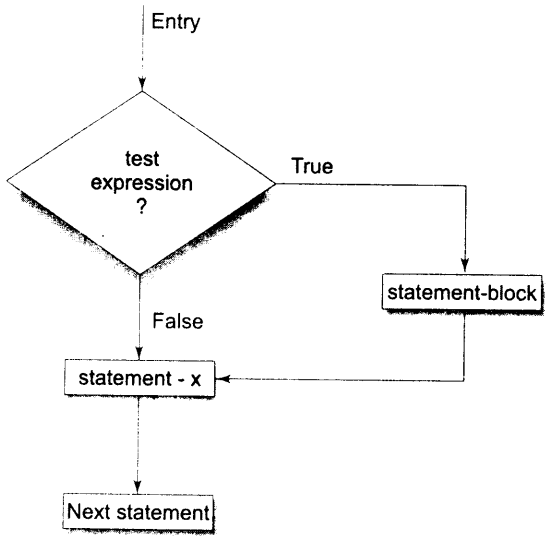


Fig. 5.2 *Flowchart of simple if Control*

Consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

```
.........
.........
if (category == SPORTS)
{
     marks = marks + bonus_marks;
}
printf("%f", marks);
.........
.........
```

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.

Example 5.1 The program in Fig. 5.3 reads four values a, b, c, and d from the terminal and evaluates the ratio of (a+b) to (c-d) and prints the result, if c-d is not equal to zero.

The program given in Fig. 5.3 has been run for two sets of data to see that the paths function properly. The result of the first run is printed as

Ratio = –3.181818

```
Program
  main()
  {
       int a, b, c, d;
       float ratio;

       printf("Enter four integer values\n");
       scanf("%d %d %d %d", &a, &b, &c, &d);

       if (c-d != 0) /* Execute statement block */
       {
            ratio = (float)(a+b)/(float)(c-d);
            printf("Ratio = %f\n", ratio);
       }
  }

Output

  Enter four integer values
  12 23 34 45
  Ratio = -3.181818

  Enter four integer values
  12  23  34  34
```

Fig. 5.3 *Illustration of simple* ***if*** *statement*

The second run has neither produced any results nor any message. During the second run, the value of (c–d) is equal to zero and therefore the statements contained in the statement-block are skipped. Since no other statement follows the statement-block, program stops without producing any output.

Note the use of **float** conversion in the statement evaluating the **ratio**. This is necessary to avoid truncation due to integer division. Remember, the output of the first run –3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use **double** or **long double** data type.

The simple **if** is often used for counting purposes. The Example 5.2 illustrates this.

Example 5.2 The program in Fig. 5.4 counts the number of boys whose weight is less than 50 kg and height is greater than 170 cm.

The program has to test two conditions, one for weight and another for height. This is done using the compound relation

if (weight < 50 && height > 170)

This would have been equivalently done using two **if** statements as follows:

```
if (weight < 50)
  if (height > 170)
    count = count +1;
```

If the value of **weight** is less than 50, then the following statement is executed, which in turn is another **if** statement. This **if** statement tests **height** and if the **height** is greater than 170, then the **count** is incremented by 1.

Program

```
main()
{
     int count, i;
     float weight, height;

     count = 0;
     printf("Enter weight and height for 10 boys\n");

     for (i =1; i <= 10; i++)
     {
          scanf("%f %f", &weight, &height);
          if (weight < 50 && height > 170)
               count = count + 1;
     }
     printf("Number of boys with weight < 50 kg\n");
     printf("and height > 170 cm = %d\n", count);
}
```

Output

```
Enter weight and height for 10 boys
45   176.5
55   174.2
47   168.0
49   170.7
54   169.0
53   170.5
49   167.0
48   175.0
47   167
51   170
Number of boys with weight < 50 kg
and height > 170 cm = 3
```

Fig. 5.4 *Use of if for counting*

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&&y||!z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as **De Morgan's** rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators"

That is,

x becomes !x
!x becomes x
&& becomes ||
|| becomes &&

Examples:

!(x && y || !z) becomes !x || !y && z

!(x <=0 || !condition) becomes x >0&& condition

5.4 THE IF.....ELSE STATEMENT

The **if...else** statement is an extension of the simple **if** statement. The general form is

```
If (test expression)
     {
       True-block statement(s)
     }
else
     {
       False-block statement(s)
     }
statement-x
```

If the *test expression* is true, then the *true-block statement(s),* immediately following the **if** statements are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. This is illustrated in Fig. 5.5. In both the cases, the control is transferred subsequently to the statement-x.

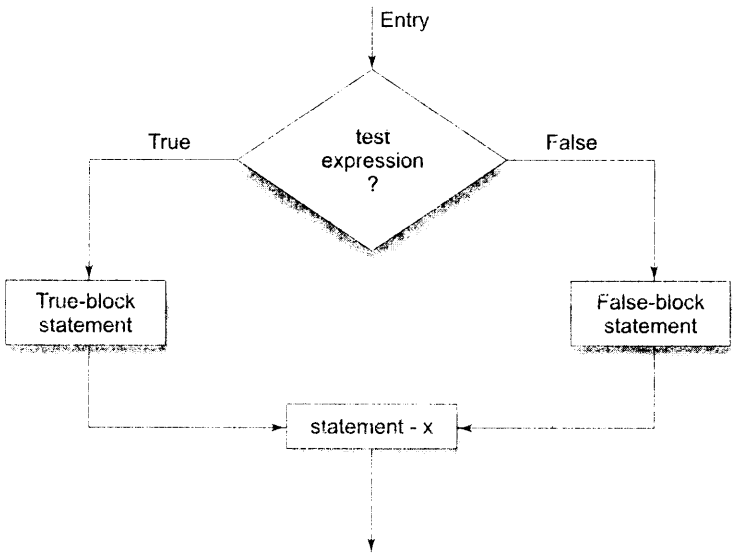


Fig. 5.5 *Flowchart of* ***if.....else*** *control*

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

```
.........
.........
if (code == 1)
  boy = boy + 1;
if (code == 2)
  girl = girl+1;
.........
.........
```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the **else** clause as follows:

```
.........
.........
if (code == 1)
  boy = boy + 1;
else
  girl = girl + 1;
xxxxxxxxxx
.........
```

Here, if the code is equal to 1, the statement **boy = boy + 1**; is executed and the control is transferred to the statement **xxxxxx**, after skipping the else part. If the code is not equal to 1, the statement **boy = boy + 1**; is skipped and the statement in the **else** part **girl = girl + 1**; is executed before the control reaches the statement **xxxxxxxx.**

Consider the program given in Fig. 5.3. When the value (c–d) is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the **else** clause as follows:

```
.........
.........
if (c-d != 0)
  {
    ratio = (float)(a+b)/(float)(c-d);
    printf("Ratio = %f\n", ratio);
  }
else
  printf("c-d is zero\n");
.........
.........
```

Example 5.3 A program to evaluate the power series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}, \quad 0 < x < 1$$

is given in Fig. 5.6. It uses **if.....else** to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1}\left(\frac{x}{n}\right) \text{ for } n > 1$$

$$T_1 = x \text{ for } n = 1$$

$$T_0 = 1$$

If T_{n-1} (usually known as *previous term*) is known, then T_n (known as *present term*) can be easily found by multiplying the previous term by x/n. Then

$$e^x = T_0 + T_1 + T_2 + \ldots\ldots + T_n = \text{sum}$$

```
Program
   #define ACCURACY 0.0001
   main()
   {
     int n, count;
     float x, term, sum;
     printf("Enter value of x:");
```

```
        scanf("%f", &x);
        n = term = sum = count = 1;
        while (n <= 100)
        {
          term = term * x/n;
          sum = sum + term;
          count = count + 1;
          if (term < ACCURACY)
            n = 999;
          else
            n = n + 1;
        }
        printf("Terms = %d Sum = %f\n", count, sum);
    }
```

Output

```
Enter value of x:0
Terms = 2 Sum = 1.000000
Enter value of x:0.1
Terms = 5 Sum = 1.105171
Enter value of x:0.5
Terms = 7 Sum = 1.648720
Enter value of x:0.75
Terms = 8 Sum = 2.116997
Enter value of x:0.99
Terms = 9 Sum = 2.691232
Enter value of x:1
Terms = 9 Sum = 2.718279
```

Fig. 5.6 *Illustration of* ***if...else*** *statement*

The program uses **count** to count the number of terms added. The program stops when the value of the term is less than 0.0001 (**ACCURACY**). Note that when a term is less than **ACCURACY**, the value of n is set equal to 999 (a number higher than 100) and therefore the **while** loop terminates. The results are printed outside the **while** loop.

5.5 NESTING OF IF...ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one **if...else** statement in *nested* form as shown below:

```
if    (test condition-1)
  if (test condition-2);
    {
      statement -1;
    }
  else
    {
      statement -2;
    }
else
    {
      statement -3;
    }
statement -x;
```

The logic of execution is illustrated in Fig. 5.7. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

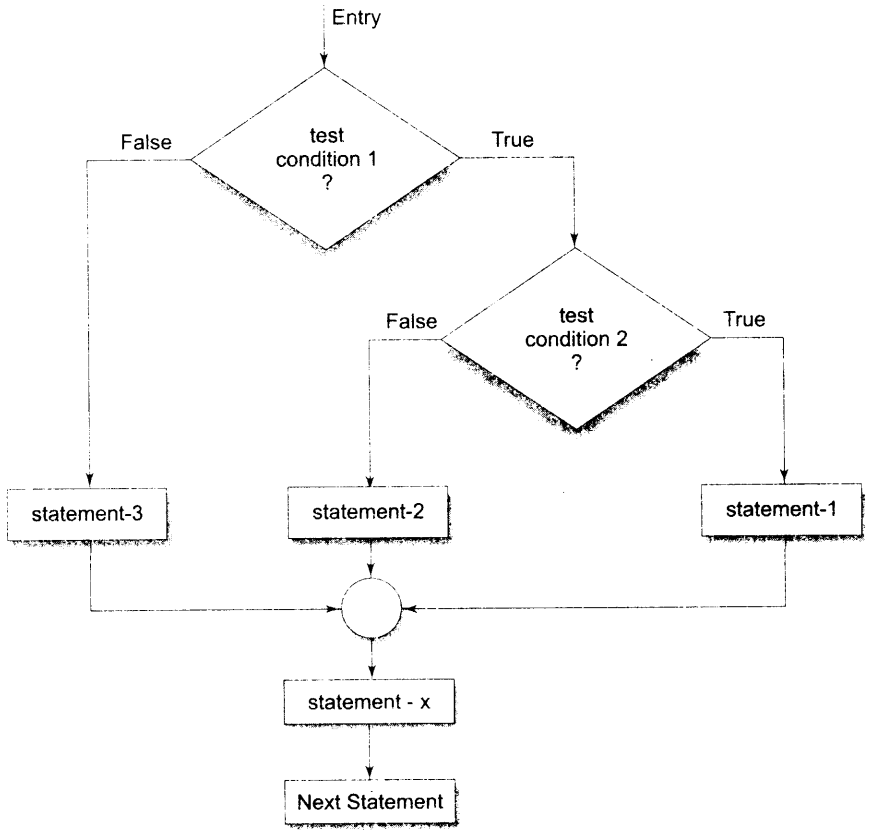


Fig. 5.7 *Flow chart of nested* ***if...else*** *statements*

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the balance held on 31st December is given to every one, irrespective of their balance, and 5 per cent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded as follows:

```
.........
  if (sex is female)
  {
    if (balance > 5000)
      bonus = 0.05 * balance;
    else
      bonus = 0.02 * balance;
  }
  else
  {
      bonus = 0.02 * balance;
  }
  balance = balance + bonus;
  .........
  .........
```

When nesting, care should be exercised to match every **if** with an **else**. Consider the following alternative to the above program (which looks right at the first sight):

```
if (sex is female)
  if (balance > 5000)
    bonus = 0.05 * balance;
  else
    bonus = 0.02 * balance;
  balance = balance + bonus;
```

There is an ambiguity as to over which **if** the **else** belongs to. In C, an else is linked to the closest non-terminated **if.** Therefore, the **else** is associated with the inner **if** and there is no else option for the outer **if**. This means that the computer is trying to execute the statement

```
balance = balance + bonus;
```

without really calculating the bonus for the male account holders.
Consider another alternative, which also looks correct:

```
if (sex is female)
  {
    if (balance > 5000)
    bonus = 0.05 * balance;
  }
else
    bonus = 0.02 * balance;
balance = balance + bonus;
```

In this case, **else** is associated with the outer **if** and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing **else** option for the inner **if**.

Example 5.4 The program in Fig. 5.8 selects and prints the largest of the three numbers using nested **if....else** statements.

```
Program
   main()
   {
   float A, B, C;

   printf("Enter three values\n");
   scanf("%f %f %f", &A, &B, &C);

   printf("\nLargest value is ");
   if (A>B)
   {
      if (A>C)
         printf("%f\n", A);
      else
         printf("%f\n", C);
   }
   else
   {
      if (C>B)
         printf("%f\n", C);
      else
         printf("%f\n", B);
   }
}
Output
      Enter three values
      23445 67379 88843

      Largest value is 88843.000000
```

Fig 5.8 *Selecting the largest of three numbers*

Dangling Else Problem

One of the classic problems encountered when we start using nested **if....else** statements is the dangling else. This occurs when a matching **else** is not available for an **if**. The answer to this problem is very simple. Always match an **else** to the most recent unmatched **if** in the current block. In some cases, it is possible that the false condition is not required. In such situations, **else** statement may be omitted

"**else** is always paired with the most recent unpaired **if**"

5.6 THE ELSE IF LADDER

There is another way of putting **if**s together when multipath decisions are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if**. It takes the following general form:

```
if ( condition 1)
     statement-1;

else if ( condition 2)
          statement-2;

   else if ( condition 3)
             statement-3;

      else if ( condition n)
                statement-n;
           else
             default-statement;

  statement-x;
```

This construct is known as the **else if** ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final **else** containing the *default-statement* will be executed. Fig. 5.9 shows the logic of execution of **else if** ladder statements.

Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Average marks	Grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

This grading can be done using the **else if** ladder as follows:

```
if (marks > 79)
     grade = "Honours";
  else if (marks > 59)
       grade = "First Division";
    else if (marks > 49)
         grade = "Second Division";
      else if (marks > 39)
```

```
                    grade = "Third Division";
                else
                    grade = "Fail";
        printf ("%s\n", grade);
```

Consider another example given below:

```
----
----
if (code == 1)
   colour = "RED";
else if (code == 2)
     colour = "GREEN";
   else if (code == 3)
          colour = "WHITE";
        else
          colour = "YELLOW";
---
---
```

Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested **if...else** statements.

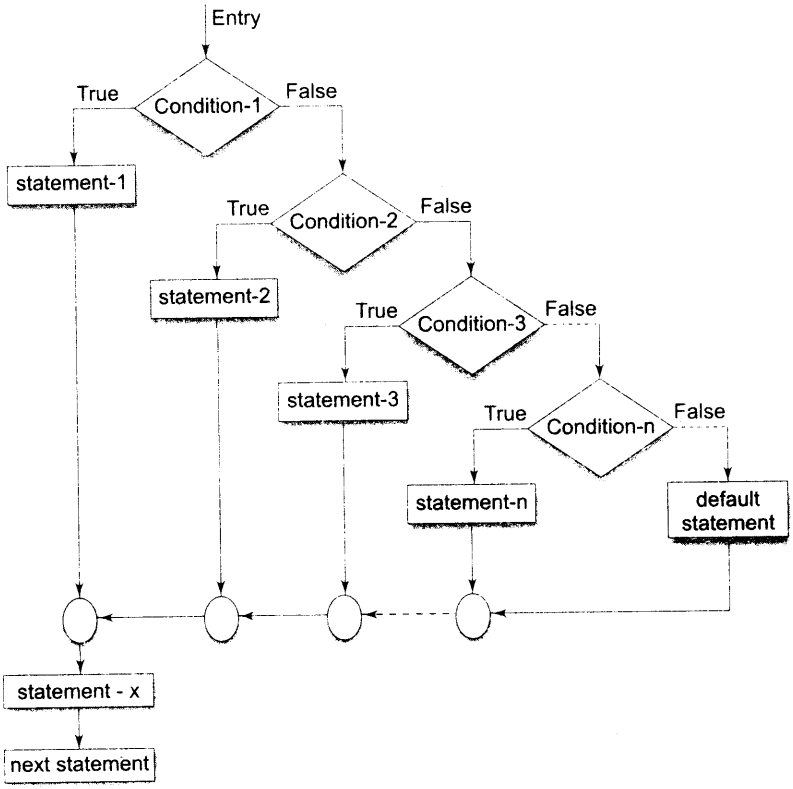


Fig. 5.9 *Flow chart of* ***else..if*** *ladder*

```
if (code != 1)
  if (code != 2)
    if (code != 3)
      colour = "YELLOW";
    else
      colour = "WHITE";
  else
      colour = "GREEN";
else
      colour = "RED";
```

In such situations, the choice is left to the programmer. However, in order to choose an **if** structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an **if** statement and the rules governing their nesting.

Example 5.5 An electric power distribution company charges its domestic consumers as follows:

Consumption Units	*Rate of Charge*
0 - 200	Rs. 0.50 per unit
201 - 400	Rs. 100 plus Rs. 0.65 per unit excess of 200
401 - 600	Rs. 230 plus Rs. 0.80 per unit excess of 400
601 and above	Rs. 390 plus Rs. 1.00 per unit excess of 600

The program in Fig. 5.10 reads the customer number and power consumed and prints the amount to be paid by the customer.

```
Program
  main()
  {
    int units, custnum;
    float charges;
    printf("Enter CUSTOMER NO. and UNITS consumed\n");
    scanf("%d %d", &custnum, &units);
    if (units <= 200)
      charges = 0.5 * units;
    else if  (units <= 400)
             charges = 100 + 0.65 * (units - 200);
               else if (units <= 600)
               charges = 230 + 0.8 * (units - 400);
                 else
                 charges = 390 + (units - 600);
    printf("\n\nCustomer No: %d: Charges = %.2f\n",
      custnum, charges);
  }
```

```
Output
Enter CUSTOMER NO. and UNITS consumed 101 150
Customer No:101 Charges = 75.00
Enter CUSTOMER NO. and UNITS consumed 202 225
Customer No:202 Charges = 116.25
Enter CUSTOMER NO. and UNITS consumed 303 375
Customer No:303 Charges = 213.75
Enter CUSTOMER NO. and UNITS consumed 404 520
Customer No:404 Charges = 326.00
Enter CUSTOMER NO. and UNITS consumed 505 625
Customer No:505 Charges = 415.00
```

Fig. 5.10 *Illustration of* ***else..if*** *ladder*

Rules for Indentation

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use *indentation* to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

5.7 THE SWITCH STATEMENT

We have seen that when one of the many alternatives is to be selected, we can use an **if** statement to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a **switch.** The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown below:

```
switch (expression)
{
   case value-1:
              block-1
              break;
   case value-2:
              block-2
              break;
   ......
   ......
   default:
              default-block
              break;
}
statement-x;
```

The *expression* is an integer expression or characters. *Value-1, value-2* ..... are constants or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a **switch** statement. **block-1, block-2** .... are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successfully compared against the values *value-1, value-2,....* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The **break** statement at the end of each block signals the end of a particular case and causes an exit from the **switch** statement, transferring the control to the **statement-x** following the **switch.**

The **default** is an optional case. When present, it will be executed if the value of the *expression* does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the **statement-x**. (ANSI C permits the use of as many as 257 case labels).

The selection process of **switch** statement is illustrated in the flow chart shown in Fig. 5.11.

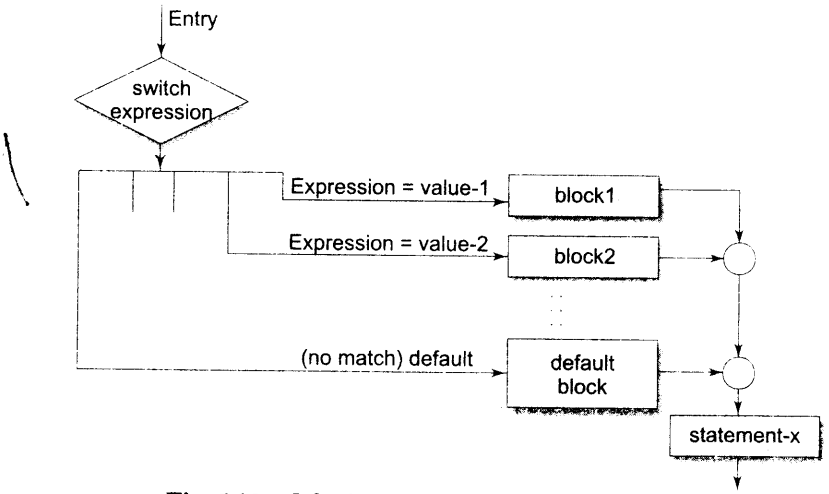


Fig. 5.11 *Selection process of the* ***switch*** *statement*

The **switch** statement can be used to grade the students as discussed in the last section. This is illustrated below:

```
---
---
index = marks/10
switch (index)
{
  case 10:
  case  9:
  case  8:
         grade = "Honours";
         break;
  case  7:
  case  6:
         grade = "First Division";
         break;
  case  5:
         grade = "Second Division";
         break;
  case  4:
         grade = "Third Division";
         break;
  default:
         grade = "Fail";
         break;
}
printf("%s\n", grade);
---
---
```

Note that we have used a conversion statement

```
index = marks / 10;
```

where, index is defined as an integer. The variable index takes the following integer values.

Marks	Index
100	10
90 - 99	9
80 - 89	8
70 - 79	7
60 - 69	6
50 - 59	5
40 - 49	4
.	.
.	.
0	0

This segment of the program illustrates two important features. First, it uses empty cases. The first three cases will execute the same statements

```
grade = "Honours";
break;
```

Same is the case with case 7 and case 6. Second, default condition is used for all other cases where marks is less than 40.

The **switch** statement is often used for menu selection. For example:

```
----
----
printf(" TRAVEL GUIDE\n\n");
printf(" A Air Timings\n" );
printf(" T Train Timings\n");
printf(" B Bus Service\n" );
printf(" X To skip\n" );
printf("\n Enter your choice\n");
character = getchar();
switch (character)
{
  case 'A' :
            air-display();
            break;
  case 'B' :
            bus-display();
            break;
  case 'T' :
            train-display();
            break;
default :
            printf(" No choice\n");
}
  ----
  ----
```

It is possible to nest the **switch** statements. That is, a **switch** may be part of a **case** statement. ANSI C permits 15 levels of nesting.

Rules for switch statement

- The **switch** expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.

- The **break** statement transfers the control out of the **switch** statement.
- The **break** statement is optional. That is, two or more case labels may belong to the same statements.
- The **default** label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one **default** label.
- The **default** may be placed anywhere but usually placed at the end.
- It is permitted to nest **switch** statements.

5.8 THE ? : OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use of the conditional operator is as follows:

conditional expression ? *expression1* : *expression2*

The *conditional expression* is evaluated first. If the result is nonzero, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
   flag = 0;
else
   flag = 1;
```

can be written as

```
flag = ( x < 0 ) ? 0 : 1;
```

Consider the evaluation of the following function:

$y = 1.5x + 3$ for $x \leq 2$
$y = 2x + 5$ for $x > 2$

This can be evaluated using the conditional operator as follows:

```
y = ( x > 2 ) ? (2 * x + 5) : (1.5 * x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

```
salary = (x != 40) ? ((x < 40) ? (4*x+100) : (4.5*x+150)) : 300;
```

The same can be evaluated using **if...else** statements as follows:

```
if (x <= 40)
     if (x < 40)
       salary = 4 * x+100;
     else
       salary = 300;
else
       salary = 4.5 * x+150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use **if** statements when more than a single nesting of conditional operator is required.

Example 5.6 An employee can apply for a loan at the beginning of every six months, but he will be sanctioned the amount according to the following company rules:
Rule 1 : An employee cannot enjoy more than two loans at any point of time.
Rule 2 : Maximum permissible total loan is limited and depends upon the category of the employee.
A program to process loan applications and to sanction loans is given in Fig. 5.12.

Program

```
#define MAXLOAN 50000
main()
{
  long int loan1, loan2, loan3, sancloan, sum23;
  printf("Enter the values of previous two loans:\n");
  scanf(" %ld %ld", &loan1, &loan2);
  printf("\nEnter the value of new loan:\n");
  scanf(" %ld", &loan3);
  sum23 = loan2 + loan3;
  sancloan = (loan1>0)? 0 : ((sum23>MAXLOAN)?
             MAXLOAN - loan2 : loan3);
  printf("\n\n");
  printf("Previous loans pending:\n%ld %ld\n",loan1,loan2);
  printf("Loan requested = %ld\n", loan3);
  printf("Loan sanctioned = %ld\n", sancloan);
}
```

Output

```
Enter the values of previous two loans:
0 20000
Enter the value of new loan:
45000
Previous loans pending:
0 20000
Loan requested = 45000
Loan sanctioned = 30000
```

```
Enter the values of previous two loans:
1000 15000
Enter the value of new loan:
25000
Previous loans pending:
1000 15000
Loan requested  = 25000
Loan sanctioned = 0
```

Fig. 5.12 *Illustration of the conditional operator*

The program uses the following variables:

loan3 - present loan amount requested
loan2 - previous loan amount pending
loan1 - previous to previous loan pending
sum23 - sum of loan2 and loan3
sancloan - loan sanctioned

The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

Some Guidelines for Writing Multiway Selection Statements

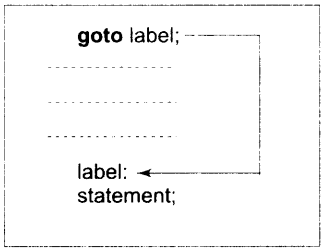
Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alternative paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

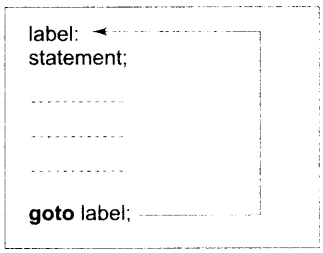
5.9 THE GOTO STATEMENT

So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:



Forward jump



Backward jump

The *label*: can be anywhere in the program either before or after the **goto** label; statement.
During running of a program when a statement like

goto begin;

is met, the flow of control will jump to the statement immediately following the label **begin**:. This happens unconditionally.
Note that a **goto** breaks the normal sequential execution of the program. If the *label:* is before the statement **goto** *label*; a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *label:* is placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward jump*.
A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
     double x, y;
     read:
     scanf("%f", &x);
     if (x < 0) goto read;
     y = sqrt(x);
     printf("%f %f\n", x, y);
     goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided. Example 5.7 illustrates how such infinite loops can be eliminated.

Example 5.7 Program presented in Fig. 5.13 illustrates the use of the **goto** statement. The program evaluates the square root for five numbers. The variable count keeps the count of numbers read. When count is less than or equal to 5, **goto read**; directs the control to the label **read**; otherwise, the program prints a message and stops.

Program

```
#include <math.h>
main()
{
     double x, y;
     int count;
     count = 1;
     printf("Enter FIVE real values in a LINE \n");
read:
     scanf("%lf", &x);
     printf("\n");
     if (x < 0)
       printf("Value - %d is negative\n",count);
     else
     {
       y = sqrt(x);
       printf("%lf\t %lf\n", x, y);
     }
     count = count + 1;
     if (count <= 5)
goto read;
     printf("\nEnd of computation");
}
```

Output

```
Enter FIVE real values in a LINE
50.70 40 -36 75 11.25
50.750000      7.123903
40.000000      6.324555
Value -3 is negative
75.000000      8.660254
11.250000      3.354102
End of computation
```

Fig. 5.13 *Use of the* ***goto*** *statement*

Another use of the **goto** statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. Example:

```
----
----
while (----)
{
   for (----)
   {
   ----
   ----
   if (----)goto end_of_program;
   ----
   }
----
----
}
end_of_program:
```

Jumping out of loops

We should try to avoid using **goto** as far as possible. But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.

Just Remember

- Be aware of dangling **else** statements.
- Be aware of any side effects in the control expression such as if(x++).
- Use braces to encapsulate the statements in **if** and **else** clauses of an if.... else statement.
- Check the use of =operator in place of the equal operator ==.
- Do not give any spaces between the two symbols of relational operators ==, !=, >= and <=.
- Writing !=, >= and <= operators like =!, => and =< is an error.
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression.
- It is an error to place a semicolon after the if expression.
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement.
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.

- Avoid using operands that have side effects in a logical binary expression such as (x--&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

CASE STUDIES

1. Range of Numbers

Problem: A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00,	40.50,	25.00,	31.25,	68.15,
47.00,	26.65,	29.00	53.45,	62.50

Determine the average cost and the range of values.

Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

Range = highest value - lowest value

It is therefore necessary to find the highest and the lowest values in the series.

Program: A program to determine the range of values and the average cost of a personal computer in the market is given in Fig. 5.14.

```
Program
  main()
  {
     int count;
     float value, high, low, sum, average, range;
     sum = 0;
     count = 0;
     printf("Enter numbers in a line :
       input a NEGATIVE number to end\n");
input:
     scanf("%f", &value);
     if (value < 0) goto output;
       count = count + 1;
     if (count == 1)
       high = low = value;
     else if (value > high)
          high = value;
       else if (value < low)
            low = value;
     sum = sum + value;
     goto input;
```

```
Output:
    average = sum/count;
    range = high - low;
    printf("\n\n");
    printf("Total values : %d\n", count);
    printf("Highest-value: %f\nLowest-value : %f\n",
           high, low);
    printf("Range        : %f\nAverage : %f\n",
           range, average);
}
Output
  Enter numbers in a line : input a NEGATIVE number to end
  35 40.50 25 31.25 68.15 47 26.65 29 53.45 62.50 -1

  Total values  : 10
  Highest-value : 68.150002
  Lowest-value  : 25.000000
  Range         : 43.150002
  Average       : 41.849998
```

Fig. 5.14 *Calculation of range of values*

When the value is read the first time, it is assigned to two buckets, **high** and **low**, through the statement

```
high = low = value;
```

For subsequent values, the value read is compared with high; if it is larger, the value is assigned to high. Otherwise, the value is compared with low; if it is smaller, the value is assigned to low. Note that at a given point, the buckets high and low hold the highest and the lowest values read so far.

The values are read in an input loop created by the **goto** input; statement. The control is transferred out of the loop by inputting a negative number. This is caused by the statement

```
if (value < 0) goto output;
```

Note that this program can be written without using **goto** statements. Try.

2. Pay-Bill Calculations

Problem: A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown below:

Level	Perks	
	Conveyance allowance	Entertainment allowance
1	1000	500
2	750	200
3	500	100
4	250	–

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	*Tax rate*
Gross <= 2000	No tax deduction
2000 < Gross <= 4000	3%
4000 < Gross <= 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number, and basic pay and then compute the net salary after withholding income tax.

Problem analysis:

Gross salary = basic pay + house rent allowance + perks

Net salary = Gross salary – income tax.

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks.
3. Calculate gross salary.
4. Calculate income tax.
5. Compute net salary.
6. Print the results.

Program: A program and the results of the test data are given in Fig. 5.15. Note that the last statement should be an executable statement. That is, the label **stop:** cannot be the last line.

```
Program
  #define CA1 1000
  #define CA2 750
  #define CA3 500
  #define CA4 250
  #define EA1 500
  #define EA2 200
  #define EA3 100
  #define EA4 0
  main()
  {
    int level, jobnumber;
    float  gross,
           basic,
           house_rent,
           perks,
           net,
           incometax;
    input:
    printf("\nEnter level, job number, and basic pay\n");
    printf("Enter 0 (zero) for level to END\n\n");
    scanf("%d", &level);
```

```
    if (level == 0) goto stop;
    scanf("%d %f", &jobnumber, &basic);
    switch (level)
    {
      case 1:
             perks = CA1 + EA1;
             break;
      case 2:
             perks = CA2 + EA2;
             break;
      case 3:
             perks = CA3 + EA3;
             break;
      case 4:
             perks = CA4 + EA4;
             break;
      default:
             printf("Error in level code\n");
             goto stop;
    }
    house_rent = 0.25 * basic;
    gross = basic + house_rent + perks;
    if (gross <= 2000)
      incometax = 0;
    else if (gross <= 4000)
           incometax = 0.03 * gross;
         else if (gross <= 5000)
              incometax = 0.05 * gross;
            else
              incometax = 0.08 * gross;
    net = gross - incometax;
    printf("%d %d %.2f\n", level, jobnumber, net);
    goto input;
    stop: printf("\n\nEND OF THE PROGRAM");
  }
```

Output

```
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
1 1111 4000
1 1111 5980.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
2 2222 3000
2 2222 4465.00
```

```
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
3 3333 2000
3 3333 3007.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
4 4444 1000
4 4444 1500.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
0
END OF THE PROGRAM
```

Fig. 5.15 *Pay-bill calculations*

REVIEW QUESTIONS

5.1 State whether the following are *true* or *false*:

(a) When **if** statements are nested, the last **else** gets associated with the nearest **if** without an **else**.

(b) One **if** can have more than one **else** clause.

(c) A **switch** statement can always be replaced by a series of **if..else** statements.

(d) A **switch** expression can be of any type.

(e) A program stops its execution when a **break** statement is encountered.

(f) Each expression in the **else if** must test the same variable.

(g) Any expression can be used for the **if** expression.

(h) Each case label can have only one statement.

(i) The **default** case is required in the **switch** statement.

(j) The predicate !((x >= 10)||(y = = 5)) is equivalent to (x < 10) && (y !=5).

5.2 Fill in the blanks in the following statements.

(a) The _____ operator is true only when both the operands are true.

(b) Multiway selection can be accomplished using an **else if** statement or the _____ statement.

(c) The _____ statement when executed in a **switch** statement causes immediate exit from the structure.

(d) The ternary conditional expression using the operator ?: could be easily coded using _____statement.

(e) The expression ! (x ! = y) can be replaced by the expression _____.

5.3 Find errors, if any, in each of the following segments:

(a)
```
if (x + y = z && y > 0)
   printf(" ");
```

(b)
```
if (code > 1);
   a = b + c
```

```
    else
       a = 0
(c) if (p < 0) || (q < 0)
       printf (" sign is negative");
```

5.4 The following is a segment of a program:

```
x = 1;
y = 1;
if (n > 0)
   x = x + 1;
   y = y - 1;
printf(" %d %d", x, y);
```

What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

5.5 Rewrite each of the following without using compound relations:

```
(a) if (grade <= 59 && grade >= 50)
       second = second + 1;
(b) if (number > 100 || number < 0)
       printf(" Out of range");
    else
       sum = sum + number;
(c) if ((M1 > 60 && M2 > 60) || T > 200)
       printf(" Admitted\n");
    else
       printf(" Not admitted\n");
```

5.6 Assuming x = 10, state whether the following logical expressions are true or false.

(a) x = = 10 && x > 10 && !x
(b) x = = 10 || x > 10 && ! x
(c) x = = 10 && x > 10 || ! x
(d) x = = 10 || x > 10 || !x

5.7 Find errors, if any, in the following switch related statements. Assume that the variables x and y are of int type and x = 1 and y = 2

```
(a) switch (y);
(b) case 10;
(c) switch (x + y)
(d) switch (x) {case 2: y = x + y; break};
```

5.8 Simplify the following compound logical expressions

(a) !(x <=10)
(b) !(x = = 10) ||! ((y = = 5) || (z < 0))
(c) ! ((x +y = = z) && !(z > 5)
(d) !((x <=5) && (y = = 10) & & (z < 5))

5.9 Assuming that x = 5, y = 0, and z = 1 initially, what will be their values after executing the following code segments?

```
(a) if (x && y)
       x = 10;
    else
       y = 10;
(b) if (x || y || z)
       y = 10;
    else
```

```
        z = 0;
(c) if (x)
      if (y)
        z = 10;
    else
        z = 0;
(d) if (x = = 0 || x & & y)
      if (!y)
        z = 0;
    else
        y = 1;
```

5.10 Assuming that x = 2, y = 1 and z = 0 initially, what will be their values after executing the following code segments?

(a)
```
switch (x)
{
    case 2:
        x = 1;
        y = x + 1;
    case 1:
        x = 0;
        break;
    default:
        x = 1;
        y = 0;
}
```

(b)
```
switch (y)
{
    case 0:
        x = 0;
        y = 0;
    case 2:
        x = 2;
        z = 2;
    default:
        x = 1;
        y = 2;
}
```

PROGRAMMING EXERCISES

5.1 Write a program to determine whether a given number is 'odd' or 'even' and print the message
NUMBER IS EVEN
or
NUMBER IS ODD
(a) without using **else** option, and (b) with **else** option.

5.2 Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.

5.3 A set of two linear equations with two unknowns x1 and x2 is given below:

$$ax_1 + bx_2 = m$$
$$cx_1 + dx_2 = n$$

The set has a unique solution

$$x1 = \frac{md - bn}{ad - cb}$$

$$x2 = \frac{na - mc}{ad - cb}$$

provided the denominator ad – cb is not equal to zero.
Write a program that will read the values of constants a, b, c, d, m, and n and compute the values of x_1 and x_2. An appropriate message should be printed if ad – cb = 0.

5.4 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students:
(a) who have obtained more than 80 marks,
(b) who have obtained more than 60 marks,
(c) who have obtained more than 40 marks,
(d) who have obtained 40 or less marks,
(e) in the range 81 to 100,
(f) in the range 61 to 80,
(g) in the range 41 to 60, and
(h) in the range 0 to 40.
The program should use a minimum number of if statements.

5.5 Admission to a professional course is subject to the following conditions:
(a) Marks in Mathematics >= 60
(b) Marks in Physics >= 50
(c) Marks in Chemistry >= 40
(d) Total in all three subjects >= 200
or
Total in Mathematics and Physics >= 150
Given the marks in the three subjects, write a program to process the applications to list the eligible candidates.

5.6 Write a program to print a two-dimensional Square Root Table as shown below, to provide the square root of any number from 0 to 9.9. For example, the value x will give the square root of 3.2 and y the square root of 3.9.

Square Root Table

Number	0.0	0.1	0.2	. . . . .	0.9
0.0 1.0 2.0					
3.0			x		y
9.0					

5.7 Shown below is a Floyd's triangle.

1
2 3
4 5 6
7 8 9 10
11 15
.
.
.
79 91

(a) Write a program to print this triangle.

(b) Modify the program to produce the following form of Floyd's triangle.

1
0 1
1 0 1
0 1 0 1
1 0 1 0 1

5.8 A cloth showroom has announced the following seasonal discounts on purchase of items:

Purchase amount	*Discount*	
	Mill cloth	*Handloom items*
0 – 100	–	5%
101 – 200	5%	7.5%
201 – 300	7.5%	10.0%
Above 300	10.0%	15.0%

Write a program using **switch** and **if** statements to compute the net amount to be paid by a customer.

5.9 Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 & \text{for } x < 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

using

(a) nested **if** statements,
(b) **else if** statements, and
(c) conditional operator ? :

5.10 Write a program to compute the real roots of a quadratic equation

$$ax^2 + bx + c = 0$$

The roots are given by the equations

$$x_1 = -b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = -b - \frac{\sqrt{b^2 - 4ac}}{2a}$$

The program should request for the values of the constants a, b and c and print the values of x_1 and x_2. Use the following rules:

(a) No solution, if both a and b are zero
(b) There is only one root, if a = 0 (x = – c/b)
(c) There are no real roots, if $b^2 - 4ac$ is negative
(d) Otherwise, there are two real roots

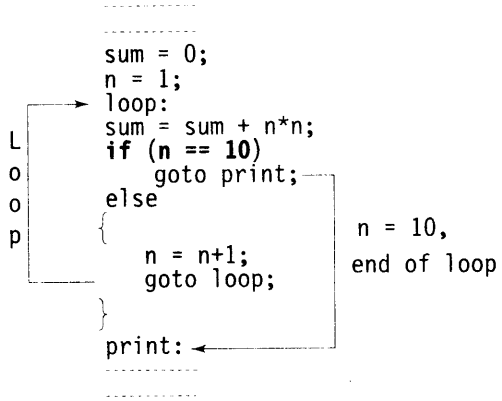
Test your program with appropriate data so that all logical paths are working as per your design. Incorporate appropriate output messages.

Chapter 6

Decision Making and Looping

6.1 INTRODUCTION

We have seen in the previous chapter that it is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10. We can write a program using the **if** statement as follows:



This program does the following things:

1. Initializes the variable **n**.
2. Computes the square of **n** and adds it to **sum**.
3. Tests the value of **n** to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.

4. If **n** is less than 10, then it is incremented by one and the control goes back to compute the **sum** again.

The program evaluates the statement

```
sum = sum + n*n;
```

10 times. That is, the loop is executed 10 times. This number can be decreased or increased easily by modifying the relational expression appropriately in the statement **if** (n == 10). On such occasions where the exact number of repetitions are known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of **goto** statements.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A *program loop* therefore consists of two segments, one known as the *body of the loop* and the other known as the *control statement*. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled loop* or as the *exit-controlled loop*. The flow charts in Fig. 6.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as *pre-test* and *post-test* loops respectively.

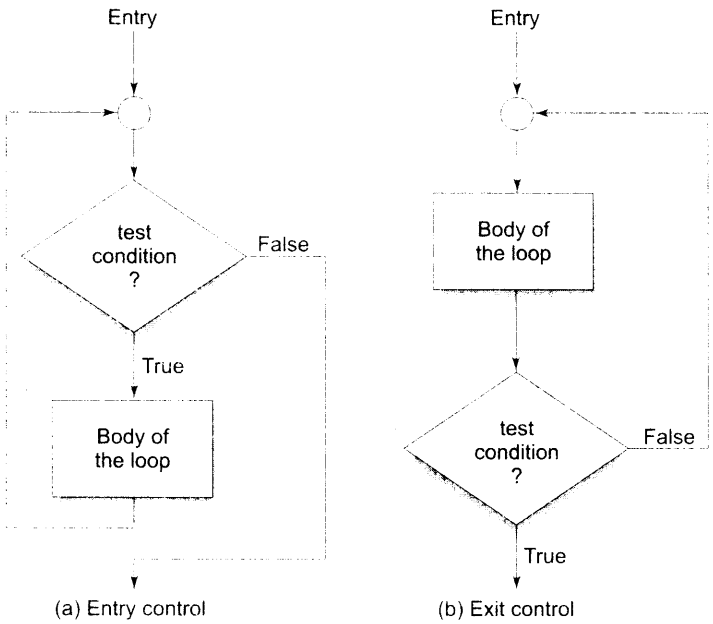


Fig. 6.1 *Loop control structures*

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an *infinite loop* and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three *constructs* for performing *loop* operations. They are:

1. The **while** statement.
2. The **do** statement.
3. The **for** statement.

We shall discuss the features and applications of each of these statements in this chapter.

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a *counter-controlled loop*. We use a control variable known *as counter*. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a *sentinel-controlled loop*, a special value called a *sentinel* value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like – 1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often called *indefinite repetition loop* because the number of repetitions is not known before the loop begins executing.

6.2 THE WHILE STATEMENT

The simplest of all the looping structures in C is the **while** statement. We have used **while** in many of our earlier programs. The basic format of the **while** statement is

```
while (test condition)
{
     body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is *true*, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Section 6.1 as follows:

```
========
sum = 0;
n = 1;                          /* Initialization */
while(n <= 10)                  /* Testing */
{
     sum = sum + n * n;
     n = n+1;                   /* Incrementing */
}
printf("sum = %d\n", sum);
========
```

loop

The body of the loop is executed 10 times for n = 1, 2, ..., 10, each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as **n < 11**; the result would be the same. This is a typical example of counter-controlled loops. The variable n is called ***counter*** or ***control variable***.

Another example of **while** statement, which uses the keyboard input is shown below:

```
=========
character = ' ' ;
while (character != 'Y')
       character = getchar();
xxxxxxx;
=========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because **character**

equals Y, and the loop terminates, thus transferring the control to the statement xxxxxxx;. This is a typical example of sentinel-controlled loops. The character constant 'y' is called *sentinel* value and the variable **character** is the condition variable, which often referred to as the *sentinel variable*.

Example 6.1 A program to evaluate the equation

$$y = x^n$$

when n is a non-negative integer, is given in Fig. 6.2

The variable **y** is initialized to 1 and then multiplied by **x**, n times using the **while** loop. The loop control variable **count** is initialized outside the loop and incremented inside the loop. When the value of **count** becomes greater than **n**, the control exists the loop.

```
Program
    main()
    {
      int count, n;
      float x, y;
      printf("Enter the values of x and n : ");
      scanf("%f %d", &x, &n);
      y = 1.0;
      count = 1;              /* Initialisation */
      /* LOOP BEGINs */
      while ( count <= n) /* Testing */
      {
        y = y*x;
        count++;              /* Incrementing */
      }
      /* END OF LOOP */
      printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
    }
Output
    Enter the values of x and n : 2.5 4
    x = 2.500000; n = 4; x to power n = 39.062500
    Enter the values of x and n : 0.5 4
    x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 6.2 *Program to compute* **x** *to the power n using* ***while*** *loop*

6.3 THE DO STATEMENT

The **while** loop construct that we have discussed in the previous section makes a test of condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the form:

```
do
{
    body of the loop
}
while (test-condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test-condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test-condition* is evaluated at the bottom of the loop, the **do...while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once.*

A simple example of a **do...while** loop is:

```
        do
        {
            printf ("Input a number\n");
loop        number = getnum ( );
        }
        while (number > 0);
```

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in, and assigned to the sentinel variable **number**.

The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

in the above example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

Consider another example:

```
-------
I = 1;                              /* Initializing */
sum = 0;
do
{
   sum = sum + I;
loop
   I = I+2;                           /* Incrementing */
}
while(sum < 40 || I < 10);          /* Testing */
printf("%d %d\n", I, sum);
-------
```