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.

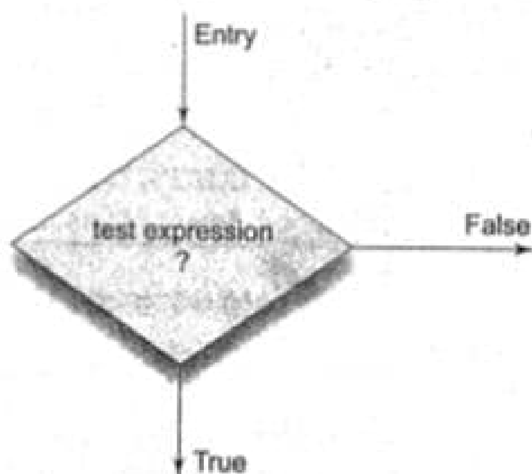



**Fig. 5.1** *Two-way branching*

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.

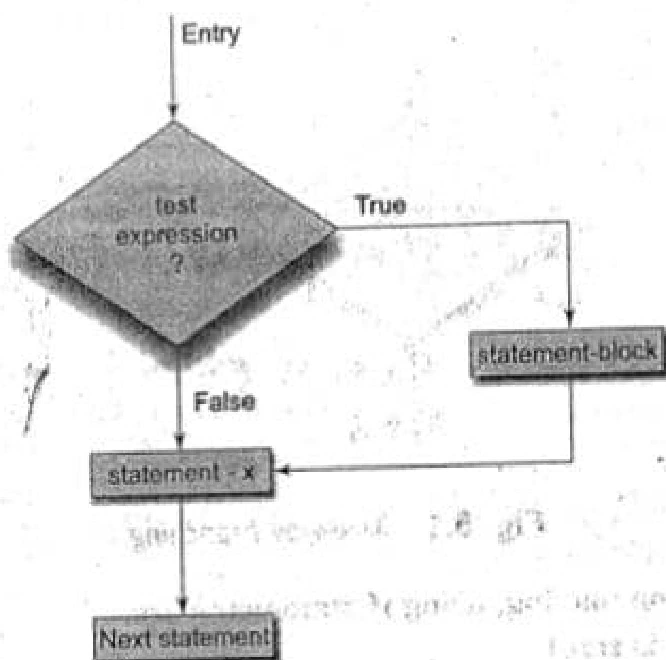



**Fig. 5.2** *Flowchart of simple* ***if*** *control*

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.

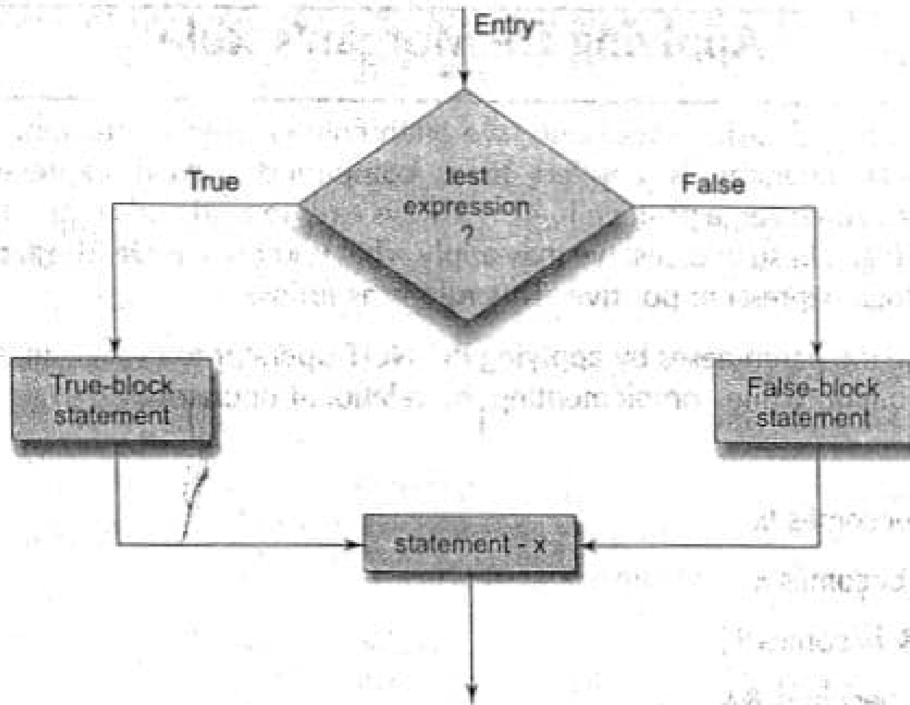



Fig. 5.5 *Flowchart of if......else control*

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

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

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

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

```
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
```

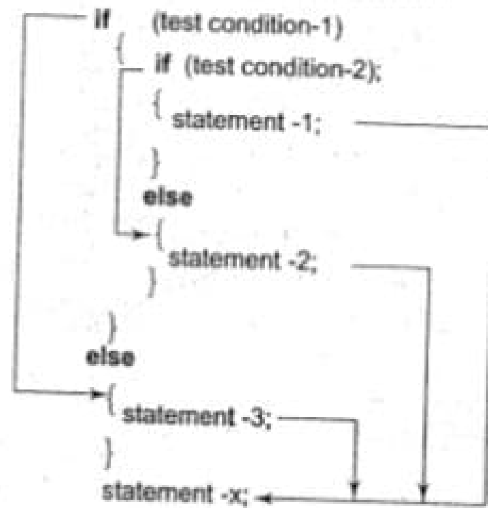
**Fig. 5.6** *Illustration of **if...else** statement*

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:

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.**

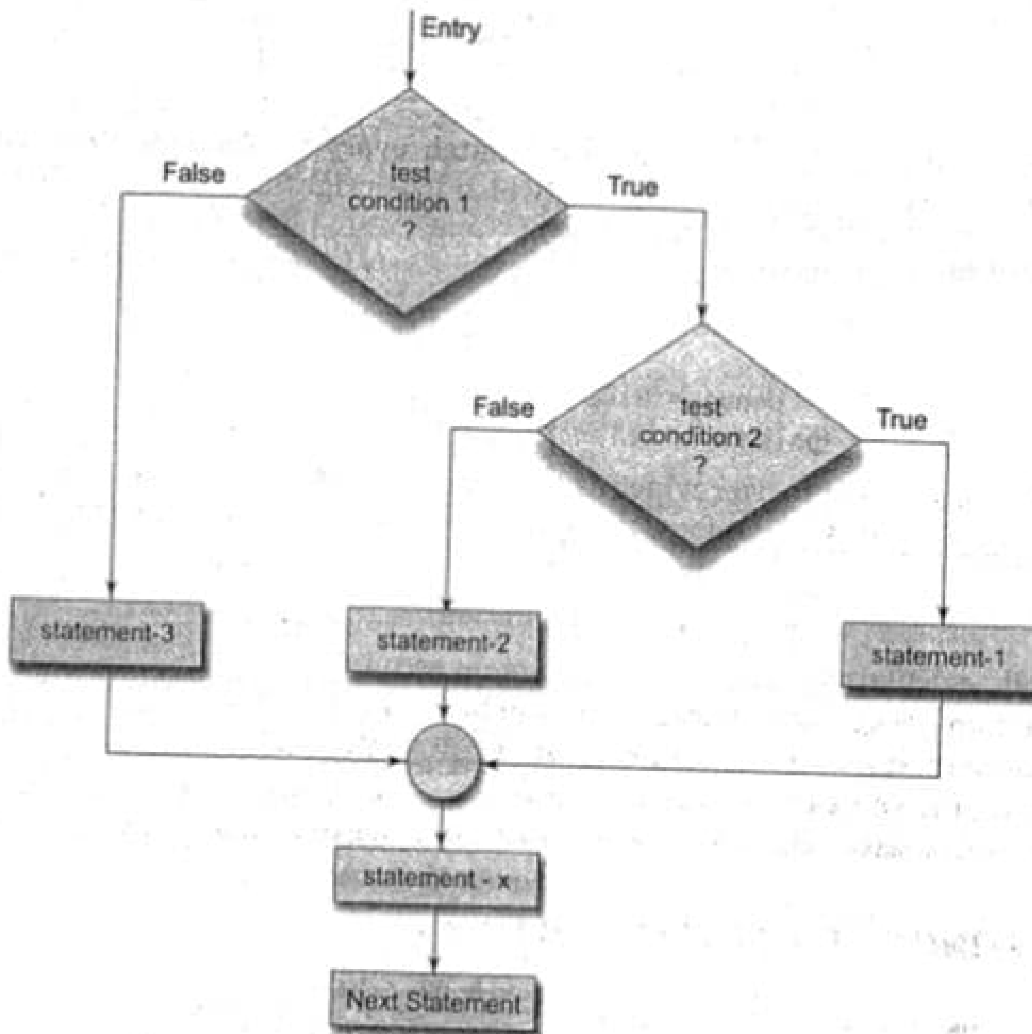



**Fig. 5.7** *Flow chart of nested* ***if...else*** *statements*

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*

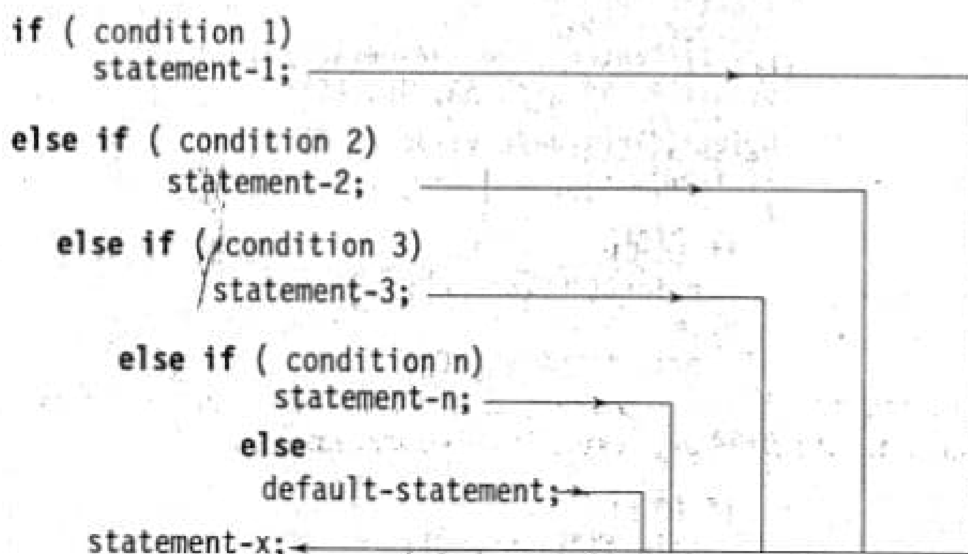
## Dangling Else Problem

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




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

Let us consider an example of grading the students in an academic institution. 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.

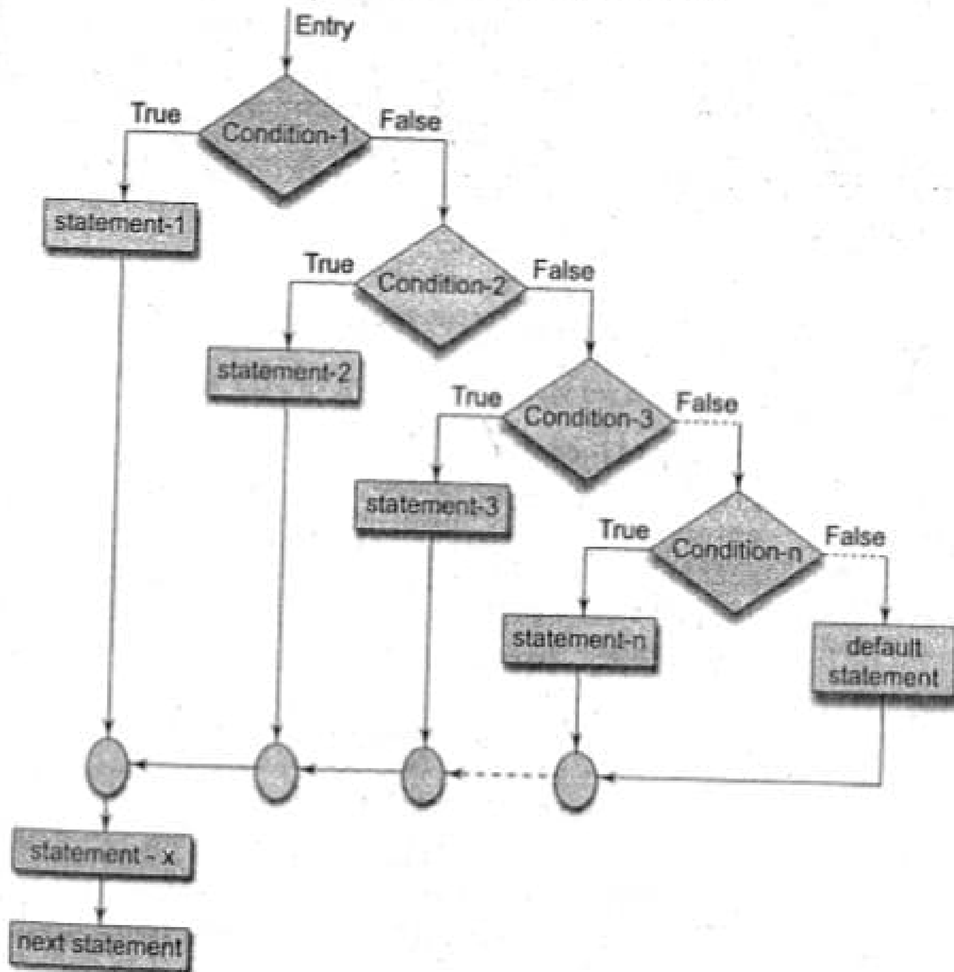



**Fig. 5.9** *Flow chart of* ***else..if*** *ladder*

```
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.

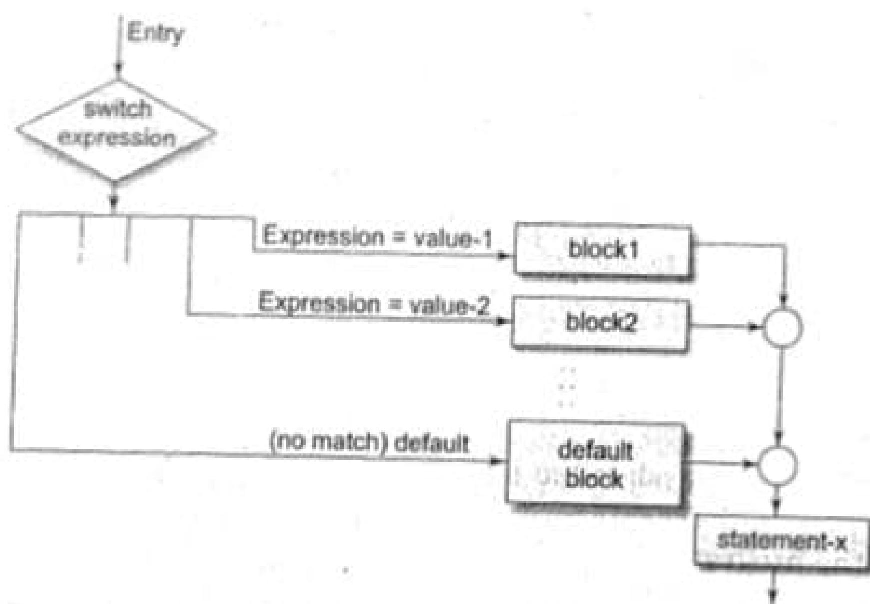



**Fig. 5.11** *Selection process of the* ***switch*** *statement*

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 $x \le 2$
$y = 2x + 5$ 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

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

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.

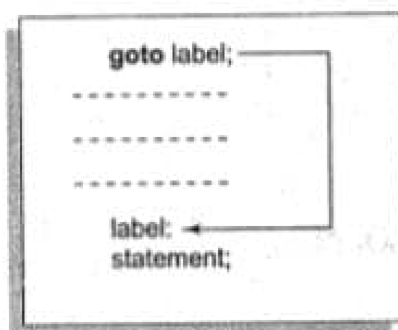
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.

  ... 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.

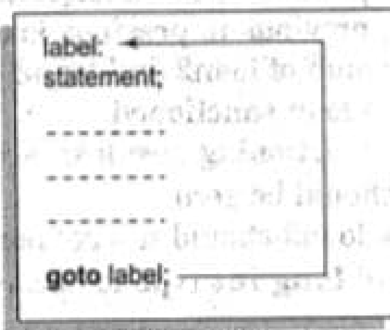
## 5.9 THE GOTO STATEMENT

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;  ---+
    ----                               |
    }                                  |  Jumping
----                                   |  out of
----                                   |  loops
}                                      |
end_of_program:   <--------------------+
```

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;
}
```

**5.11 Find the error, if any, in the following statements:**

(a)
```
if ( x > = 10 ) then
printf ( "\n") ;
```
(b)
```
if x > = 10
printf ( "OK" ) ;
```
(c)
```
if (x = 10)
printf ("Good" ) ;
```
(d)
```
if (x = < 10)
 printf ("Welcome") ;
```

**5.12 What is the output of the following program?**

```
main ( )
{
        int m = 5 ;
        if (m < 3) printf("%d" , m+1) ;
        else if(m < 5) printf("%d", m+2);
        else if(m < 7) printf("%d", m+3);
        else printf("%d", m+4);

}
```

5.13 What is the output of the following program?

```
main ( )
{
        int m = 1;
        if ( m==1)
        {
                printf ( " Delhi " ) ;
                if (m == 2)
                printf( "Chennai" ) ;
                else
                printf("Bangalore") ;
        }
        else;
        printf(" END");
 }
```

5.14 What is the output of the following program?

```
main( )
{
        int m ;
        for (m = 1; m<5; m++)
                printf(%d\n", (m%2) ? m : m*2);
}
```

5.15 What is the output of the following program?

```
main( )
{
        int m, n, p ;
        for ( m = 0; m < 3; m++ )
        for (n = 0; n<3; n++ )
        for ( p = 0; p < 3;; p++ )
        if ( m + n + p == 2 )
        goto print;

        print :
        printf("%d, %d, %d", m, n, p);
}
```

5.16 What will be the value of x when the following segment is executed?

```
        int x = 10, y = 15;
        x = (x<y)? (y+x) : (y-x) ;
```

5.17 What will be the output when the following segment is executed?

```
int x = 0;
if (x >= 0)
if ( x > 0 )
```

```
printf("Number is positive");
else
printf("Number is negative");
```

5.18 What will be the output when the following segment is executed?

```
char ch = 'a' ;
switch (ch)
{
        case 'a' :
        printf( "A" ) ;
        case'b':
        Printf ("B") ;
        default :
        printf(" C ") ;
}
```

5.19 What will be the output of the following segment when executed?

```
int x = 10, y = 20;
if( (x<y) || (x+5) > 10 )
printf("%d", x);
else
printf("%d", y);
```

5.20 What will be output of the following segment when executed?

```
int a = 10, b = 5;
if (a > b)
{
        if(b > 5)
        printf("%d", b);
}
else
        printf("%d", a);
```

## 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 $b^2 - 4$ ac 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.

5.11 Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle.

5.12 An electricity board charges the following rates for the use of electricity:
For the first 200 units: 80 P per unit
For the next 100 units: 90 P per unit
Beyond 300 units: Rs 1.00 per unit
All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs. 400, then an additional surcharge of 15% of total amount is charged. Write a program to read the names of users and number of units consumed and print out the charges with names.

5.13 Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values.

5.14 Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly.
Modify the program to count all the prime numbers that lie between 100 and 200.
*NOTE*: A prime number is a positive integer that is divisible only by 1 or by itself.

5.15 Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of

(a) sin(x), if s or S is assigned to T,
(b) cos (x), if c or C is assigned to T, and
(c) tan (x), if t or T is assigned to T

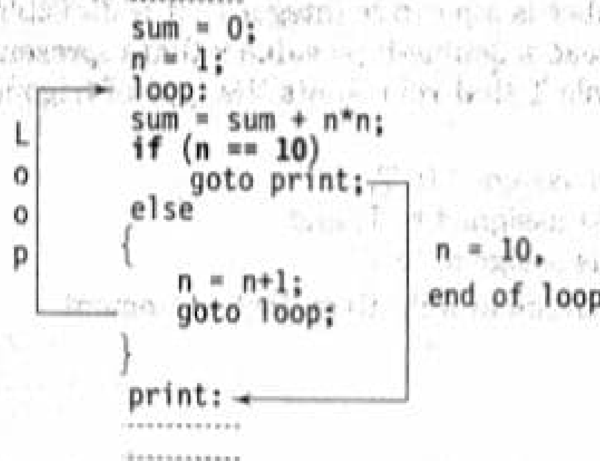
using (i) **if......else** statement and (ii) **switch** statement.

# 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 increased or decreased 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.

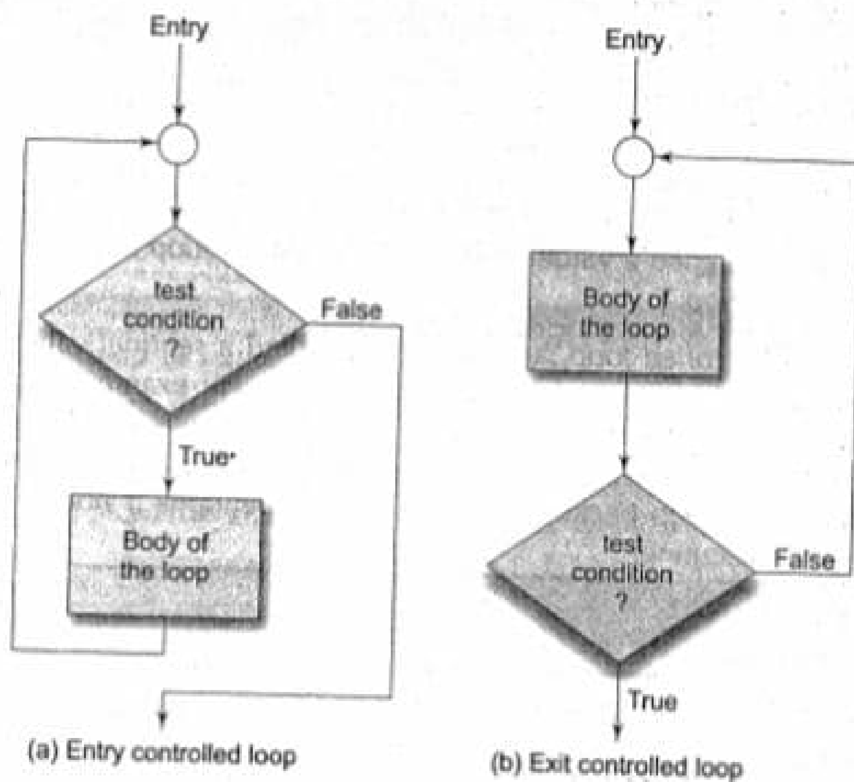



**Fig. 6.1** *Loop control structures*

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 spe
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

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 ch

## 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 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 */
     |  {
loop |    sum = sum + n * n;
     |      n = n+1;                  /* Incrementing */
     -> }
        printf("sum = %d\n", sum);
        ========
```

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);
         -------
```

The loop will be executed as long as one of the two relations is true.

**Example 6.2** A program to print the multiplication table from 1 x 1 to 12 x 10 as shown below is given in Fig. 6.3.

```
1    2    3    4    .........  10
2    4    6    8    .........  20
3    6    9    12   .........  30
4                   .........  40
-
-
-
12                  .........  120
```

This program contains two **do.... while** loops in nested form. The outer loop is controlled by the variable **row** and executed 12 times. The inner loop is controlled by the variable **column** and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

```
Program:
        #define COLMAX 10
        #define ROWMAX 12
        main()
        {
             int row,column, y;
             row = 1;
             printf("    MULTIPLICATION TABLE     \n");
             printf("-------------------------------\n");
             do /*......OUTER LOOP BEGINS........*/
             {
                  column = 1;
                  do /*.......INNER LOOP BEGINS.......*/
                  {
                    y = row * column;
                    printf("%4d", y);
                    column = column + 1;
                  }
```

```
            while (column <= COLMAX); /*... INNER LOOP ENDS ...*/
            printf("\n");
            row = row + 1;
        }
        while (row <= ROWMAX);/*..... OUTER LOOP ENDS .....*/
        printf("-------------------------------------------\n");
    }
Output
            MULTIPLICATION TABLE
-----------------------------------------------
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
11  22  33  44  55  66  77  88  99  110
12  24  36  48  60  72  84  96  108 120
-----------------------------------------------
```

**Fig. 6.3** *Printing of a multiplication table using* **do...while loop**

Notice that the **printf** of the inner loop does not contain any new line character (\n). This allows the printing of all row values in one line. The empty **printf** in the outer loop initiates a new line to print the next row.

## 6.4 THE FOR STATEMENT

### Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for ( initialization ; test-condition ; increment)
{
    body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The *test-condition* is a relational expression, such as i < 10 that determines when the loop will exit. If the

```
        double q;
        printf("-----------------------------------------\n");
        printf(" 2 to power n          n          2 to power -n\n");
        printf("-----------------------------------------\n");
        p = 1;
        for (n = 0; n < 21 ; ++n) /* LOOP BEGINS */
        {
          if (n == 0)
            p = 1;
          else
            p = p * 2;
          q = 1.0/(double)p ;
          printf("%10ld %10d %20.12lf\n", p, n, q);
        }                         /* LOOP ENDS */
        printf("-----------------------------------------\n");
    }
```

**Output**

```
----------------------------------------------------
  2 to power n        n              2 to power -n
----------------------------------------------------
             1        0              1.000000000000
             2        1              0.500000000000
             4        2              0.250000000000
             8        3              0.125000000000
            16        4              0.062500000000
            32        5              0.031250000000
            64        6              0.015625000000
           128        7              0.007812500000
           256        8              0.003906250000
           512        9              0.001953125000
          1024       10              0.000976562500
          2048       11              0.000488281250
          4096       12              0.000244140625
          8192       13              0.000122070313
         16384       14              0.000061035156
         32768       15              0.000030517578
         65536       16              0.000015258789
        131072       17              0.000007629395
        262144       18              0.000003814697
        524288       19              0.000001907349
       1048576       20              0.000000953674
----------------------------------------------------
```

**Fig. 6.4** *Program to print* ***'Power of 2'*** *table using for loop*

Note that the initialization section has two parts **p** = 1 and **n** = 1 separated by a *comma*.

Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<=m; n=n+1, m=m-1)
{
    p = m/n;
    printf("%d %d %d\n", n, m, p);
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below:

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
    sum = sum+i;
    printf("%d %d\n", i, sum);
}
```

The loop uses a compound test condition with the counter variable **i** and sentinel variable **sum**. The loop is executed as long as both the conditions **i** < 20 and **sum** < 100 are true. The **sum** is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
-------
m = 5;
for ( ; m != 100 ; )
{
    printf("%d\n", m);
    m = m+5;
}
-------
```

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable is incremented inside the loop. In such cases, the sections are left 'blank'. However, the semicolons separating the sections must remain. If the test-condition is not present, the **for** statement sets up an *'infinite'* loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up *time delay loops* using the null statement as follows:

```
for ( j = 1000; j > 0; j = j-1)
    ;
```

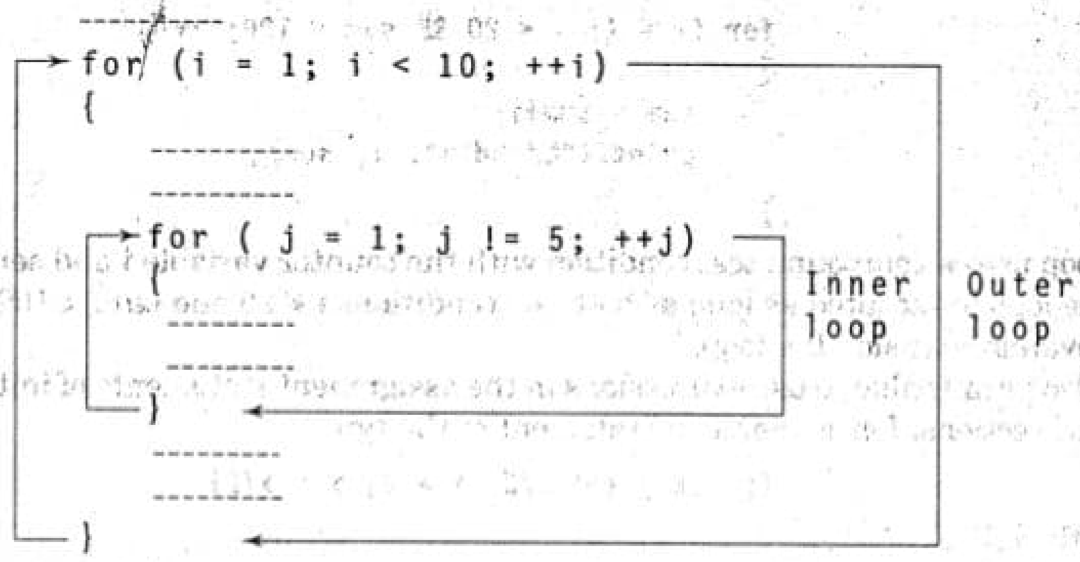
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

```
for (j=1000; j > 0; j = j-1)
```

This implies that the C compiler will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as a *null statement* and the program may produce some nonsense.

## Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in C. For example, two loops can be nested as follows:




The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more).

The program to print the multiplication table discussed in Example 6.2 can be written more concisely using nested for statements as follows:

```
-------------
for (row = 1; row <= ROWMAX ; ++row)
{
  for (column = 1; column <= COLMAX ; ++column)
  {
      y = row * column;
      printf("%4d", y);
  }
  printf("\n");
}
-------------
```

The outer loop controls the rows while the inner loop controls the columns.

**Example 6.4** A class of **n** students take an annual examination in **m** subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig. 6.5.

The program uses two **for** loops, one for controlling the number of students and the other for controlling the number of subjects. Since both the number of students and the number of subjects are requested by the program, the program may be used for a class of any size and any number of subjects.

The outer loop includes three parts:

(1) reading of roll-numbers of students, one after another;
(2) inner loop, where the marks are read and totalled for each student; and
(3) printing of total marks and declaration of grades.

**Program**

```
#define FIRST 360
#define SECOND 240
main()
{
    int n, m, i, j,
        roll_number, marks, total;
    printf("Enter number of students and subjects\n");
    scanf("%d %d", &n, &m);
    printf("\n");
    for (i = 1; i <= n ; ++i)
    {
        printf("Enter roll_number : ");
        scanf("%d", &roll_number);
        total = 0 ;
        printf("\nEnter marks of %d subjects for ROLL NO %d\n",
                m,roll_number);
        for (j = 1; j <= m; j++)
        {
            scanf("%d", &marks);
            total = total + marks;
        }
        printf("TOTAL MARKS = %d ", total);
        if (total >= FIRST)
           printf("( First Division )\n\n");
        else if (total >= SECOND)
              printf("( Second Division )\n\n");
          else
              printf("( *** F A I L *** )\n\n");
    }
}
```

**Output**

```
Enter number of students and subjects
3 6
Enter roll_number : 8701
Enter marks of 6 subjects for ROLL NO 8701
81 75 83 45 61 59
TOTAL MARKS = 404 ( First Division )
Enter roll_number : 8702
Enter marks of 6 subjects for ROLL NO 8702
51 49 55 47 65 41
TOTAL MARKS = 308 ( Second Division )
Enter roll_number : 8704
Enter marks of 6 subjects for ROLL NO 8704
40 19 31 47 39 25
TOTAL MARKS = 201 ( *** F A I L *** )
```

**Fig. 6.5** *Illustration of nested for loops*

## Selecting a Loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyse the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, **do while**.
- If it requires a pre-test loop, then we have two choices: **for** and **while**.
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use **for** loop if the counter-based control is necessary.
- Use **while** loop if the sentinel-based control is required.
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

## 6.5 JUMPS IN LOOPS

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names 100 times must be termi-

nated as soon as the desired name is found. C permits a *jump* from one statement to another within a loop as well as a *jump* out of a loop.

## Jumping Out of a Loop

An early exiti from a loop can be accomplished by using the **break** statement or the **goto** statement. We have already seen the use of the **break** in the **switch** statement and the **goto** in the **if...else** construct. These statements can also be used within **while, do,** or **for** loops. They are illustrated in Fig. 6.6 and Fig. 6.7.

When a **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the **break** will exit only a single loop.

Since a **goto** statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of **goto** is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

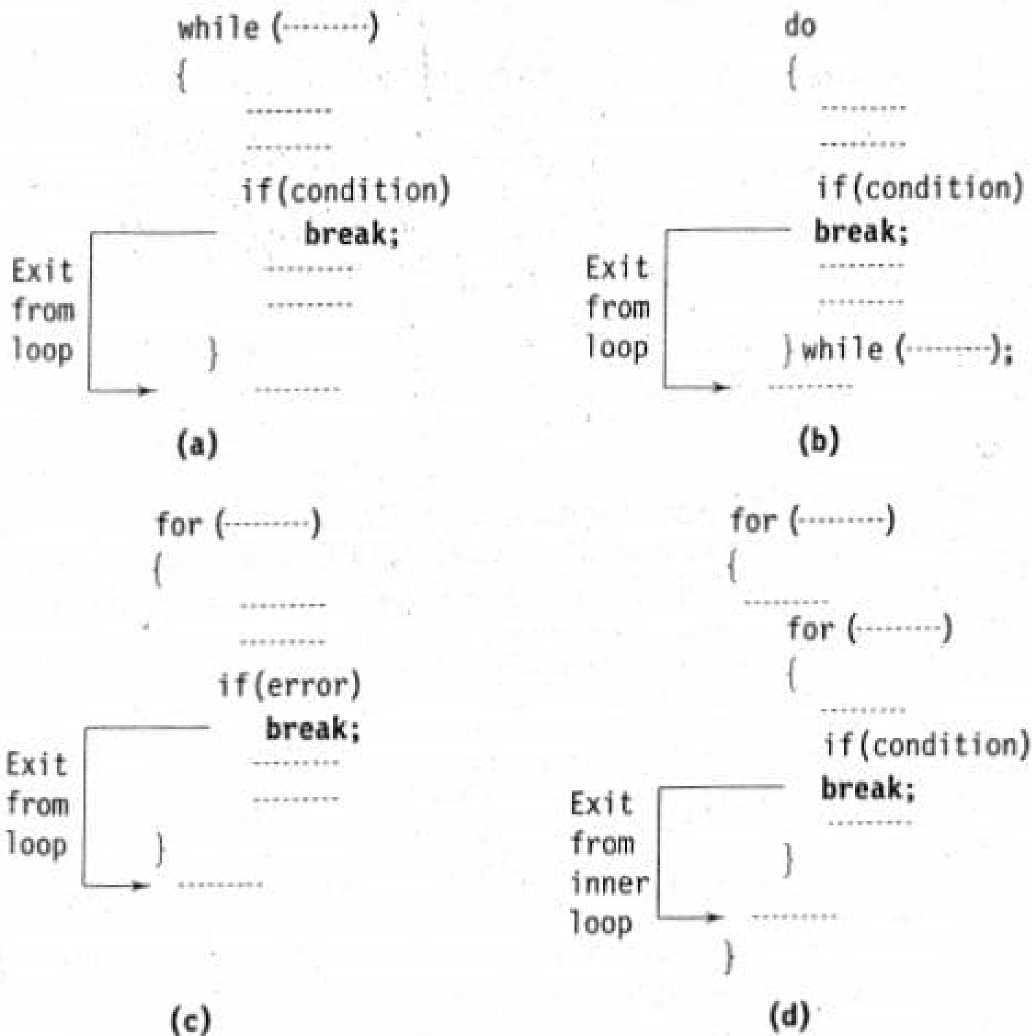



Fig. 6.6 *Exiting a loop with* ***break*** *statement*

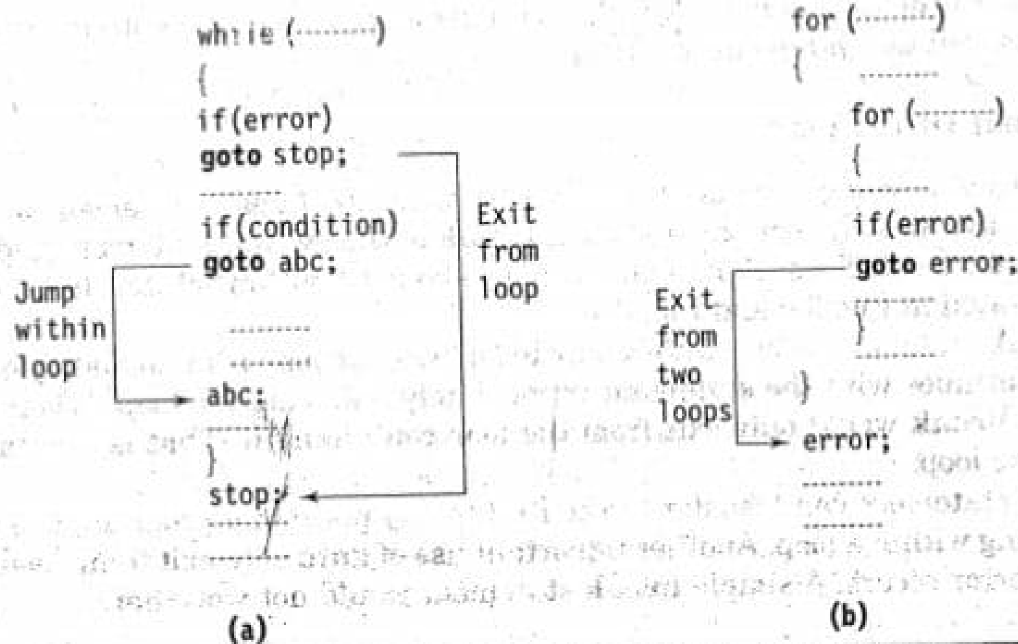



Fig. 6.7 *Jumping within and exiting from the loops with* **goto** *statement*

**Example 6.5** The program in Fig. 6.8 illustrates the use of the break statement in a C program.

The program reads a list of positive values and calculates their average. The **for** loop is written to read 1000 values. However, if we want the program to calculate the average of any set of values less than 1000, then we must enter a 'negative' number after the last value in the list, to mark the end of input.

```
Program
    main()
    {
         int m;
         float x, sum, average;

         printf("This  program computes the average of a
                       set of numbers\n");
         printf("Enter values one after another\n");
         printf("Enter a NEGATIVE number at the end.\n\n");
         sum = 0;
         for (m = 1 ; m < = 1000 ; ++m)
         {
            scanf("%f", &x);
            if (x < 0)
               break;
            sum += x ;
         }
         average = sum/(float)(m-1);
         printf("\n");
```

```
    printf("Number of values  = %d\n", m-1);
    printf("Sum               = %f\n", sum);
    printf("Average           = %f\n", average);
}
Output
    This program computes the average of a set of numbers
    Enter values one after another
    Enter a NEGATIVE number at the end.

    21 23 24 22 26 22 -1

    Number of values  = 6
    Sum               = 138.000000
    Average           = 23.000000
```

**Fig. 6.8** *Use of* ***break*** *in a program*

Each value, when it is read, is tested to see whether it is a positive number or not. If it is positive, the value is added to the **sum;** otherwise, the loop terminates. On exit, the average of the values read is calculated and the results are printed out.

**Example 6.6** A program to evaluate the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + ..... + x^n$$

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 6.9. The **goto** statement is used to exit the loop on achieving the desired accuracy.

We have used the **for** statement to perform the repeated addition of each of the terms in the series. Since it is an infinite series, the evaluation of the function is terminated when the term $x^n$ reaches the desired accuracy. The value of n that decides the number of loop operations is not known and therefore we have decided arbitrarily a value of 100, which may or may not result in the desired level of accuracy.

```
Program
    #define  LOOP         100
    #define  ACCURACY     0.0001
    main()
    {
        int n;
        float x, term, sum;
        printf("Input value of x : ");
        scanf("%f", &x);
        sum = 0 ;
        for (term = 1, n = 1 ; n <= LOOP ; ++n)
        {
            sum += term ;
            if (term <= ACCURACY)
```

```
                goto output; /* EXIT FROM THE LOOP */
            term *= x ;
        }
        printf("\nFINAL VALUE OF N IS NOT SUFFICIENT\n");
        printf("TO ACHIEVE DESIRED ACCURACY\n");
        goto end;
        output:
        printf("\nEXIT FROM LOOP\n");
        printf("Sum = %f; No.of terms = %d\n", sum, n);
        end:
        ;       /* Null Statement */
}
```

**Output**

```
Input value of x : .21
EXIT FROM LOOP
Sum = 1.265800; No.of terms = 7
Input value of x : .75
EXIT FROM LOOP
Sum = 3.999774; No.of terms = 34
Input value of x : .99
FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY
```

**Fig. 6.9** *Use of* **goto** *to exit from a loop*

The test of accuracy is made using an **if** statement and the **goto** statement exits the loop as soon as the accuracy condition is satisfied. If the number of loop repetitions is not large enough to produce the desired accuracy, the program prints an appropriate message.

Note that the **break** statement is not very convenient to use here. Both the normal exit and the **break** exit will transfer the control to the same statement that appears next to the loop. But, in the present problem, the normal exit prints the message

"FINAL VALUE OF N IS NOT SUFFICIENT
TO ACHIEVE DESIRED ACCURACY"

and the *forced exit* prints the results of evaluation. Notice the use of a *null* statement at the end. This is necessary because a program should not end with a label.

## Structured Programming

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection (branching) structure

- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection structure proves to be more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with *"goto less programming"*.

Do not go to **goto** statement!

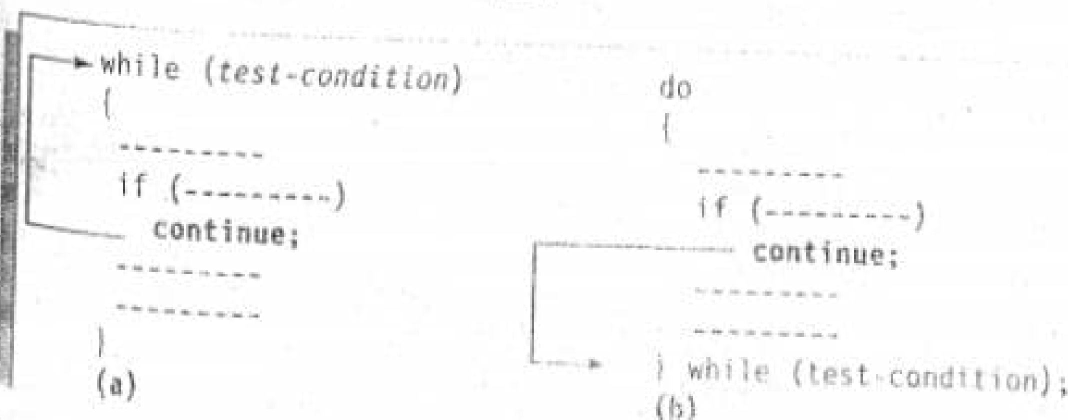
## Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
continue;
```

The use of the **continue** statement in loops is illustrated in Fig. 6.10. In **while** and **do** loops, **continue** causes the control to go directly to the test-condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.



```
for (initialization; test condition; increment)
{
    ---------
    if (---------)
      continue;
    ---------
    ---------
}
```

(c)

**Fig. 6.10** *Bypassing and continuing in loops*

**Example 6.7** The program in Fig. 6.11 illustrates the use of **continue** statement.

The program evaluates the square root of a series of numbers and prints the results. The process stops when the number 9999 is typed in.

In case, the series contains any negative numbers, the process of evaluation of square root should be bypassed for such numbers because the square root of a negative number is not defined. The **continue** statement is used to achieve this. The program also prints a message saying that the number is negative and keeps an account of negative numbers.

The final output includes the number of positive values evaluated and the number of negative items encountered.

**Program:**

```
#include <math.h>
main()
{
   int count, negative;
   double number, sqroot;
   printf("Enter 9999 to STOP\n");
   count = 0 ;
   negative = 0 ;
   while (count < = 100)
   {
        printf("Enter a number : ");
        scanf("%lf", &number);
        if (number == 9999)
          break;      /* EXIT FROM THE LOOP */
        if (number < 0)
        {
          printf("Number is negative\n\n");
          negative++ ;
          continue; /* SKIP REST OF THE LOOP */
        }
```

```
            sqroot = sqrt(number);
            printf("Number     = %lf\n Square root = %lf\n\n",
                                      number, sqroot);
            count++ ;
        }
        printf("Number of items done = %d\n", count);
        printf("\n\nNegative items = %d\n", negative);
        printf("END OF DATA\n");
    }
```

**Output**

```
Enter 9999 to STOP
Enter a number : 25.0
Number        = 25.000000
Square root   = 5.000000

Enter a number : 40.5
Number        = 40.500000
Square root   = 6.363961

Enter a number : -9
Number is negative

Enter a number : 16
Number        = 16.000000
Square root   = 4.000000

Enter a number : -14.75
Number is negative

Enter a number : 80
Number        = 80.000000
Square root   = 8.944272

Enter a number : 9999
Number of items done = 4
Negative items       = 2
END OF DATA
```

**Fig. 6.11** *Use of **continue** statement*

## Avoiding goto

As mentioned earlier, it is a good practice to avoid using **goto**. There are many reasons for this. When **goto** is used, many compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable. It is possible to avoid using **goto** by careful program design. In case any **goto** is absolutely necessary, it should be documented. The **goto** jumps shown in Fig. 6.12 would cause problems and therefore must be avoided.

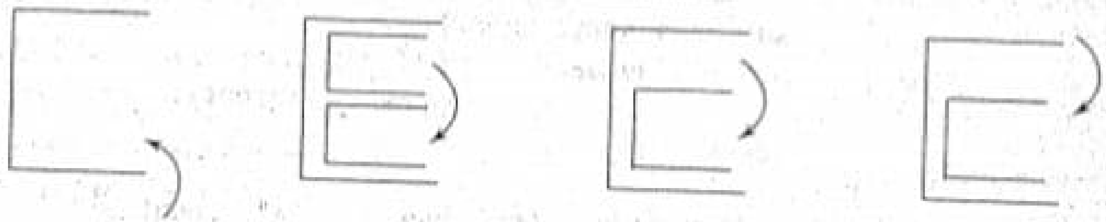


Fig. 6.12 *goto jumps to be ovoided.*

## Jumping out of the Program

We have just seen that we can jump out of a loop using either the **break** statement or goto statement. In a similar way, we can jump out of a program by using the library function **exit( )**. In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the **exit( )** function, as shown below:

```
........
........
if (test-condition) exit(0) ;
........
........
```

The **exit( )** function takes an integer value as its argument. Normally *zero* is used to indicate normal termination and a *nonzero* value to indicate termination due to some error or abnormal condition. The use of **exit( )** function requires the inclusion of the header file **<stdlib.h>**.

## 6.6 CONCISE TEST EXPRESSIONS

We often use test expressions in the **if, for, while** and **do** statements that are evaluated and compared with zero for making branching decisions. Since every integer expression has a true/false value, we need not make explicit comparisons with zero. For instance, the expression x is true whenever x is not zero, and false when x is zero. Applying! operator, we can write concise test expressions without using any relational operators.

*if (expression ==0)*

is equivalent to

*if(!expression)*

Similarly,

*if (expression! = 0)*

is equivalent to

*if (expression)*

For example,

**if (m%5==0 && n%5==0)** is same as **if (!(m%5)&&!(n%5))**

## Just Remember

- Do not forget to place the semicolon at the end of **do ....while** statement.
- Placing a semicolon after the control expression in a **while** or **for** statement is not a syntax error but it is most likely a logic error.
- Using commas rather than semicolon in the header of a **for** statement is an error.
- Do not forget to place the *increment* statement in the body of a **while** or **do...while** loop.
- It is a common error to use wrong relational operator in test expressions. Ensure that the loop is evaluated exactly the required number of times.
- Avoid a common error using = in place of = = operator.
- Do not change the control variable in both the **for** statement and the body of the loop. It is a logic error.
- Do not compare floating-point values for equality.
- Avoid using **while** and **for** statements for implementing exit-controlled (post-test) loops. Use **do...while** statement. Similarly, do not use **do...while** for pre-test loops.
- When performing an operation on a variable repeatedly in the body of a loop, make sure that the variable is initialized properly before entering the loop.
- Although it is legally allowed to place the initialization, testing and increment sections outside the header of a **for** statement, avoid them as far as possible.
- Although it is permissible to use arithmetic expressions in initialization and increment section, be aware of round off and truncation errors during their evaluation.
- Although statements preceding a **for** and statements in the body can be placed in the **for** header, avoid doing so as it makes the program more difficult to read.
- The use of **break** and **continue** statements in any of the loops is considered unstructured programming. Try to eliminate the use of these jump statements, as far as possible.
- Avoid the use of **goto** anywhere in the program.
- Indent the statements in the body of loops properly to enhance readability and understandability.
- Use of blank spaces before and after the loops and terminating remarks are highly recommended.
- Use the function **exit()** only when breaking out of a program is necessary.

## Case Studies

### 1. Table of Binomial Coefficients

***Problem***: Binomial coefficients are used in the study of binomial distributions and reliability of multicomponent redundant systems. It is given by

$$B(m,x) = \binom{m}{x} = \frac{m!}{x!(m-x)!}, \quad m >= x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

***Problem Analysis***: The binomial coefficient can be recursively calculated as follows:

$$B(m,o) = 1$$

$$B(m,x) = B(m,x-1)\left[\frac{m-x+1}{x}\right], x = 1,2,3,\ldots,m$$

Further,

$$B(o,o) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program in Fig. 6.12 prints the table of binomial coefficients for m = 10. The program employs one **do** loop and one **while** loop.

```
Program
    #define MAX 10
    main()
    {
        int m, x, binom;
        printf(" m x");
        for (m = 0; m <= 10 ; ++m)
            printf("%4d", m);
        printf("\n----------------------------------\n");
        m = 0;
        do
        {
            printf("%2d ", m);
            x = 0; binom = 1;
            while (x <= m)
            {
                if(m == 0 || x == 0)
                    printf("%4d", binom);
                else
                {
                    binom = binom * (m - x + 1)/x;
                    printf("%4d", binom);
                }
```

```
                x = x + 1;
            }
            printf("\n");
            m = m + 1;
        }
        while (m <= MAX);
        printf("-------------------------------------\n");
    }
```

**Output**

```
     mx   0   1   2   3   4   5   6   7   8   9  10
   -------------------------------------------------
      0   1
      1   1   1
      2   1   2   1
      3   1   3   3   1
      4   1   4   6   4   1
      5   1   5  10  10   5   1
      6   1   6  15  20  15   6   1
      7   1   7  21  35  35  21   7   1
      8   1   8  28  56  70  56  28   8   1
      9   1   9  36  84 126 126  84  36   9   1
     10   1  10  45 120 210 252 210 120  45  10   1
   -------------------------------------------------
```

**Fig. 6.12** *Program to print* ***binomial coefficient*** *table*

## 2. Histogram

***Problem***: In an organization, the employees are grouped according to their basic pay for the purpose of certain perks. The pay-range and the number of employees in each group are as follows:

| Group | Pay-Range | Number of Employees |
|---|---|---|
| 1 | 750 – 1500 | 12 |
| 2 | 1501 – 3000 | 23 |
| 3 | 3001 – 4500 | 35 |
| 4 | 4501 – 6000 | 20 |
| 5 | above 6000 | 11 |

Draw a histogram to highlight the group sizes.

***Problem Analysis***: Given the size of groups, it is required to draw bars representing the sizes of various groups. For each bar, its group number and size are to be written.

Program in Fig. 6.13 reads the number of employees belonging to each group and draws a histogram. The program uses four **for** loops and two **if.....else** statements.

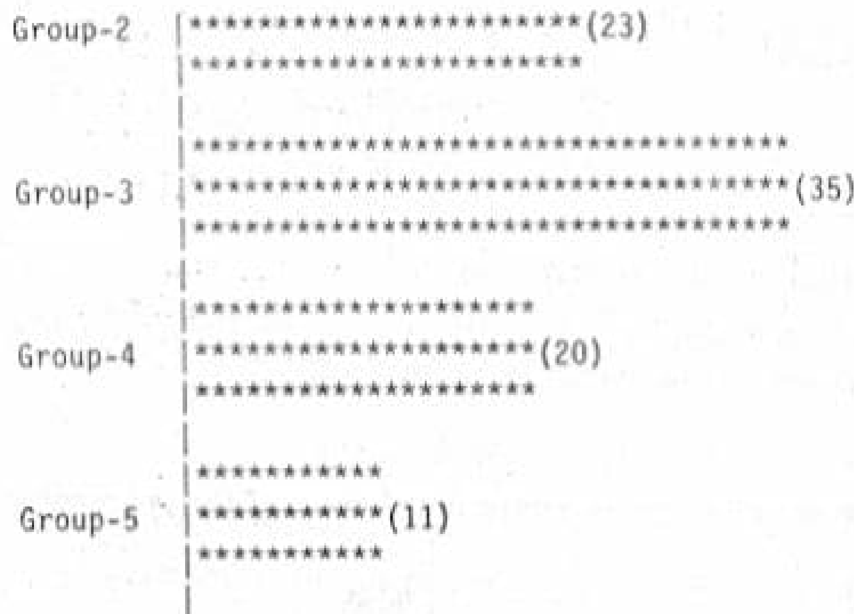
***Program:***

```
    #define  N 5
    main()
    {
        int  value[N];
```

```
    int i, j, n, x;
    for (n=0; n < N; ++n)
    {
      printf("Enter employees in Group - %d : ",n+1);
      scanf("%d", &x);
      value[n] = x;
      printf("%d\n", value[n]);
    }
    printf("\n");
    printf("|\n");
    for (n = 0 ; n < N ; ++n)
    {
      for (i = 1 ; i <= 3 ; i++)
      {
          if ( i == 2)
            printf("Group-%1d |",n+1);
          else
            printf("|");
          for (j = 1 ; j <= value[n]; ++j)
            printf("*");
          if (i == 2)
            printf("(%d)\n", value[n]);
          else
            printf("\n");
      }
      printf("|\n");
    }
}
```

**Output**

```
Enter employees in Group - 1 : 12
12
Enter employees in Group - 2 : 23
23
Enter employees in Group - 3 : 35
35
Enter employees in Group - 4 : 20
20
Enter Employees in Group - 5 : 11
11
          |
          |************
Group-1   |************(12)
          |************
          |
          |***********************
```



```
Group-2  |***********************(23)
         |***********************
         |
         |***********************************
Group-3  |***********************************(35)
         |***********************************
         |
         |********************
Group-4  |********************(20)
         |********************
         |
         |***********
Group-5  |***********(11)
         |***********
         |
```

**Fig. 6.13** *Program to draw a **histogram***

### 3. Minimum Cost

***Problem:*** The cost of operation of a unit consists of two components C1 and C2 which can be expressed as functions of a parameter p as follows:

$$C1 = 30 - 8p$$

$$C2 = 10 + p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of + 0.1 where the cost of operation would be minimum.

***Problem Analysis:***

$$\text{Total cost} = C_1 + C_2 = 40 - 8p + p^2$$

The cost is 40 when p = 0, and 33 when p = 1 and 60 when p = 10. The cost, therefore, decreases first and then increases. The program in Fig. 6.14 evaluates the cost at successive intervals of p (in steps of 0.1) and stops when the cost begins to increase. The program employs **break** and **continue** statements to exit the loop.

```
Program
    main()
    {
       float p, cost, p1, cost1;
       for (p = 0; p <= 10; p = p + 0.1)
       {
          cost = 40 - 8 * p + p * p;
          if(p == 0)
          {
            cost1 = cost;
```

```
                continue;
            }
            if (cost >= cost1)
                break;
            cost1 = cost;
            p1 = p;
        }
        p = (p + p1)/2.0;
        cost = 40 - 8 * p + p * p;
        printf("\nMINIMUM COST = %.2f AT p = %.1f\n",
                cost, p);
    }
Output
    MINIMUM COST = 24.00 AT p = 4.0
```

Fig. 6.14 *Program of minimum cost problem*

### 4. Plotting of Two Functions

***Problem***: We have two functions of the type

$$y1 = \exp(-ax)$$
$$y2 = \exp(-ax^2/2)$$

Plot the graphs of these functions for x varying from 0 to 5.0.

***Problem Analysis***: Initially when x = 0, y1 = y2 =1 and the graphs start from the same point. The curves cross when they are again equal at x = 2.0. The program should have appropriate branch statements to print the graph points at the following three conditions:

1. y1 > y2
2. y1 < y2
3. y1 = y2

The functions y1 and y2 are normalized and converted to integers as follows:

$$y1 = 50 \exp(-ax) + 0.5$$
$$y2 = 50 \exp(-ax^2/2) + 0.5$$

The program in Fig. 6.15 plots these two functions simultaneously. ( 0 for y1, * for y2, and # for the common point).

```
Program
    #include <math.h>
    main()
    {
        int i;
        float a, x, y1, y2;
        a = 0.4;
        printf("                    Y ---->                           \n");
```

```
printf(" 0 -----------------------------------\n");
for ( x = 0; x < 5; x = x+0.25)
{ /* BEGINNING OF FOR LOOP */
/*......Evaluation of functions .......*/
  y1 = (int) ( 50 * exp( -a * x ) + 0.5 );
  y2 = (int) ( 50 * exp( -a * x * x/2 ) + 0.5 );
/*......Plotting when y1 = y2.........*/
  if ( y1 == y2)
  {
    if ( x == 2.5)
        printf(" X    |");
    else
        printf("|");
    for ( i = 1; i <= y1 - 1; ++i)
        printf(" ");
    printf("#\n");
    continue;
  }
 /*...... Plotting when y1 > y2 .....*/
  if ( y1 > y2)
  {
    if ( x == 2.5 )
      printf(" X |");
    else
      printf("    |");
    for ( i = 1; i <= y2 -1 ; ++i)
      printf(" ");
    printf("*");
    for ( i = 1; i <= (y1 - y2 - 1); ++i)
      printf("-");
    printf("0\n");
    continue;
  }
 /*........ Plotting when y2 > y1.........*/
  if ( x == 2.5)
    printf(" X |");
  else
    printf("    |");
  for ( i = 1 ; i <= (y1 - 1); ++i )
    printf(" ");
  printf("0");
  for ( i = 1; i <= ( y2 - y1 - 1 ); ++i)
    printf("-");
  printf("*\n");
 } /*.......END OF FOR LOOP........*/
  printf("    |\n");
}
```

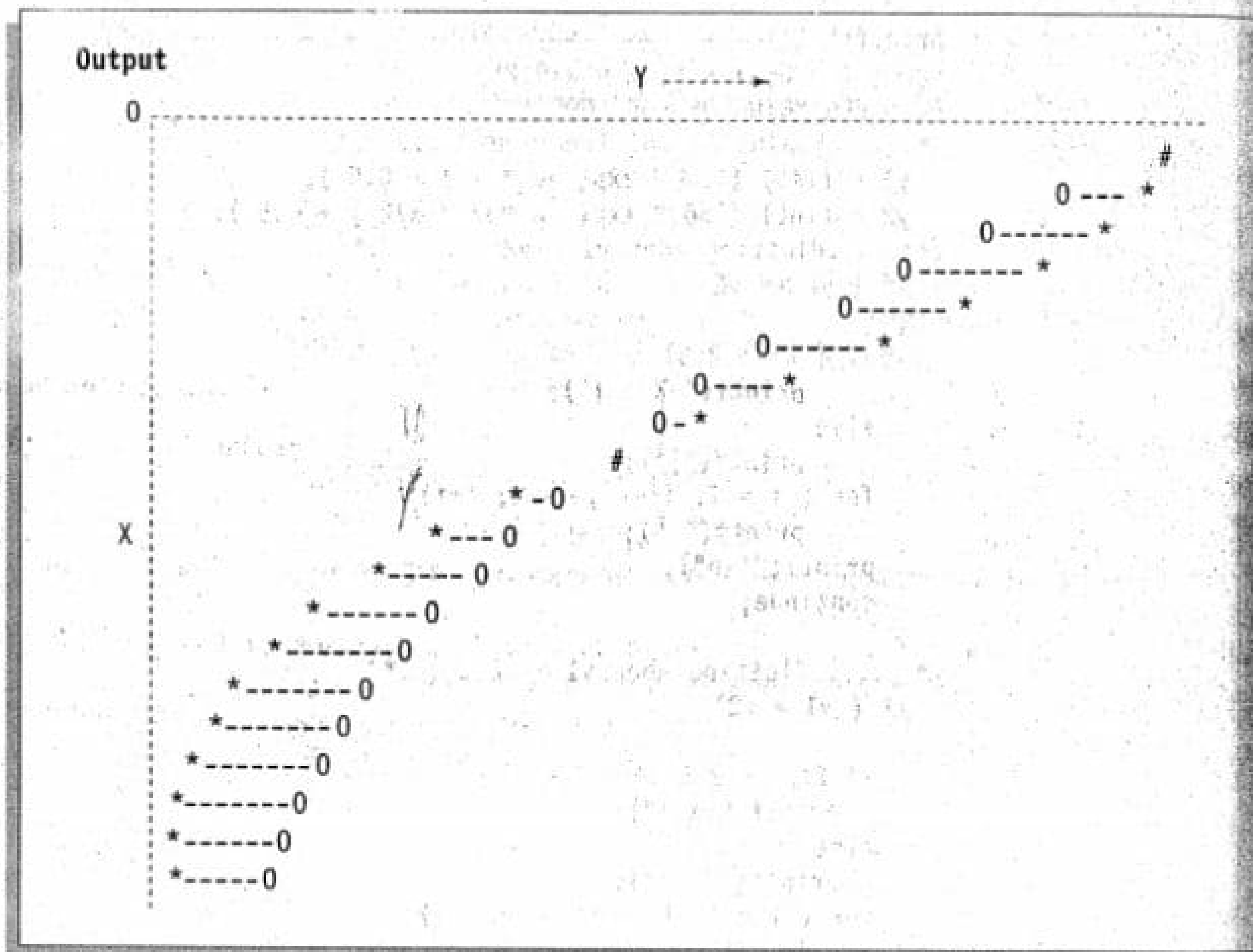



Fig. 6.15 *Plotting of two functions*

## Review Questions

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

(a) The **do...while** statement first executes the loop body and then evaluate the loop control expression.

(b) In a pretest loop, if the body is executed **n** times, the test expression is executed **n** + 1 times.

(c) The number of times a control variable is updated always equals the number of loop iterations.

(d) Both the pretest loops include initialization within the statement.

(e) In a **for** loop expression, the starting value of the control variable must be less than its ending value.

(f) The initialization, test condition and increment parts may be missing in a **for** statement.

(g) **while** loops can be used to replace **for** loops without any change in the body of the loop.

(h) An exit-controlled loop is executed a minimum of one time.
(i) The use of **continue** statement is considered as unstructured programming.
(j) The three loop expressions used in a **for** loop header must be separated by commas.

6.2 Fill in the blanks in the following statements.
(a) In an exit-controlled loop, if the body is executed n times, the test condition is evaluated ________ times.
(b) The ________ statement is used to skip a part of the statements in a loop.
(c) A **for** loop with the no test condition is known as ______ loop.
(d) The sentinel-controlled loop is also known as _______ loop.
(e) In a counter-controlled loop, variable known as _____ is used to count the loop operations.

6.3 Can we change the value of the control variable in **for** statements? If yes, explain its consequences.

6.4 What is a null statement? Explain a typical use of it.

6.5 Use of **goto** should be avoided. Explain a typical example where we find the application of **goto** becomes necessary.

6.6 How would you decide the use of one of the three loops in C for a given problem?

6.7 How can we use **for** loops when the number of iterations are not known?

6.8 Explain the operation of each of the following **for** loops.

(a)
```
for ( n = 1; n != 10; n += 2)
    sum = sum + n;
```
(b)
```
for (n = 5; n <= m; n --=1)
    sum = sum + n;
```
(c)
```
for (n = 1; n <= 5;)
    sum = sum + n;
```
(d)
```
for ( n = 1; ; n = n + 1)
    sum = sum + n;
```
(e)
```
for (n = 1; n < 5; n ++)
    n = n -1
```

6.9 What would be the output of each of the following code segments?

(a)
```
count = 5;
while (count -- > 0)
printf(count);
```
(b)
```
count = 5;
while ( -- count > 0)
printf(count);
```
(c)
```
count = 5;
do printf(count);
while (count > 0);
```
(d)
```
for (m = 10; m > 7, m -=2)
printf(m);
```

6.10 Compare, in terms of their functions, the following pairs of statements:
(a) while and do...while
(b) while and for

(c) break and goto
(d) break and continue
(e) continue and goto

6.11 Analyse each of the program segments that follow and determine how many times the body of each loop will be executed.

(a)
```
x = 5;
y = 50;
while ( x <= y)
{
    x = y/x;
    ---
    ---
}
```

(b)
```
m = 1;
do
{
    ---
    ---
    m = m+2;
}
while (m < 10);
```

(c)
```
int i;
for (i = 0; i <= 5; i = i+2/3)
{
    ---
    ---
    ---
}
```

(d)
```
int m = 10;
int n = 7;
while ( m % n >= 0)
{
    ---
    m = m + 1;
    n = n + 2;
    ---
}
```

6.12 Find errors, if any, in each of the following looping segments. Assume that all the variables have been declared and assigned values.

(a)
```
while (count != 10);
{
    count = 1;
    sum = sum + x;
    count = count + 1;
}
```

(b)
```
name = 0;
do { name = name + 1;
printf("My name is John\n");}
while (name = 1)
```
(c)
```
do;
total = total + value;
scanf("%f", &value);
while (value != 999);
```
(d)
```
for (x = 1, x > 10; x = x + 1)
{
    ---
    ---
    ---
}
```
(e)
```
m = 1;
n = 0;
for ( ; m+n < 10; ++n);
printf("Hello\n");
m = m+10
```
(f)
```
for (p = 10; p > 0;)
p = p - 1;
printf("%f", p);
```
6.13 Write a **for** statement to print each of the following sequences of integers:

(a) 1, 2, 4, 8, 16, 32

(b) 1, 3, 9, 27, 81, 243

(c) – 4, –2, 0, 2, 4

(d) –10, –12, –14, –18, –26, – 42

6.14 Change the following **for** loops to **while** loops:

(a)
```
for (m = 1; m < 10; m = m + 1)
    printf(m);
```
(b)
```
for ( ; scanf("%d", & m) != -1;)
    printf(m);
```
6.15 Change the **for** loops in Exercise 6.14 to **do** loops.

6.16 What is the output of following code?
```
int m = 100, n = 0;
while ( n == 0 )
{
        if ( m < 10 )
                break;
        m = m-10;
```
6.17 What is the output of the following code?
```
int m = 0 ;
do
{
```

```
        if (m > 10 )
            continue ;
        m = m + 10 ;
    } while ( m < 50 ) ;
    printf("%d", m);
```

6.18 What is the output of the following code?

```
int n = 0, m = 1 ;
do
{
        printf(m) ;
        m++ ;
}
while (m <= n) ;
```

6.19 What is the output of the following code?

```
int n = 0, m ;
for (m = 1; m <= n + 1 ; m++ )
        printf(m);
```

6.20 When do we use the following statement?

```
for (; ; )
```

## Programming Exercises

6.1 Given a number, write a program using **while** loop to reverse the digits of the number. For example, the number

12345

should be written as

54321

(**Hint:** Use modulus operator to extract the last digit and the integer division by 10 to get the n-1 digit number from the n digit number.)

6.2 The factorial of an integer m is the product of consecutive integers from 1 to m. That is,

factorial m = m! = m x (m–1) x ...... x 1.

Write a program that computes and prints a table of factorials for any given m.

6.3 Write a program to compute the sum of the digits of a given integer number.

6.4 The numbers in the sequence

1 1 2 3 5 8 13 21 .......

are called Fibonacci numbers. Write a program using a **do....while** loop to calculate and print the first m Fibonacci numbers.

(**Hint:** After the first two numbers in the series, each number is the sum of the two preceding numbers.)

6.5 Rewrite the program of the Example 6.1 using the **for** statement.

6.6 Write a program to evaluate the following investment equation

$$V = P(1+r)^n$$

and print the tables which would give the value of V for various combination of the following values of P, r, and n.

P : 1000, 2000, 3000,........., 10,000

r : 0.10, 0.11, 0.12, ........, 0.20

n : 1, 2, 3, ...., 10

(**Hint:** P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$$V = P(1+r)$$

$$P = V$$

That is, the value of money at the end of first year becomes the principal amount for the next year and so on.)

6.7 Write programs to print the following outputs using **for** loops.

(a)
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

(b)
```
* * * * *
  * * * *
    * * *
      * *
        *
```

6.8 Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

6.9 Rewrite the program of case study 6.4 (plotting of two curves) using **else...if** constructs instead of **continue** statements.

6.10 Write a program to print a table of values of the function

$$y = \exp(-x)$$

for x varying from 0.0 to 10.0 in steps of 0.10. The table should appear as follows:

**Table for Y = EXP(–X)**

| *x* | *0.1* | *0.2* | *0.3* | ........... | *0.9* |
|---|---|---|---|---|---|
| 0.0 | | | | | |
| 1.0 | | | | | |
| 2.0 | | | | | |
| 3.0 | | | | | |
| . | | | | | |
| . | | | | | |
| . | | | | | |
| 9.0 | | | | | |

6.11 Write a program that will read a positive integer and determine and print its binary equivalent.

(**Hint:** The bits of the binary representation of an integer can be generated by repeatedly dividing the number and the successive quotients by 2 and saving the remainder, which is either 0 or 1, after each division.)

**6.12** Write a program using **for** and **if** statement to display the capital letter S in a grid of 15 rows and 18 columns as shown below.

```
* * * * * * * * * * * * * * * * * *
* * - - - - - - - - - - - - - - - - * *
* * * * * * * * * - - - - - - - - - - * *
* * * *
* * * *
* * * *
* * * * * - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
- - - - - - - - - - - - - - - - - - * * * *
                                    * * * *
                                    * * * *
                                    * * * *
* * * * - - - - - - - - - - - - - - * * * *
* * * - - - - - - - - - - - - - -   * * * *
* * - - - - - - - - - - - - - - - - * * * *
```

**6.13** Write a program to compute the value of Euler's number e, that is used as the base of natural logarithms. Use the following formula.

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots\ldots + 1/n!$$

Use a suitable loop construct. The loop must terminate when the difference between two successive values of e is less than 0.00001.

**6.14** Write programs to evaluate the following functions to 0.0001% accuracy.

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots$

(b) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots\ldots$

(c) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \ldots\ldots$

**6.15** The present value (popularly known as book value) of an item is given by the relationship.

$$P = c\,(1-d)^n$$

where $c$ = original cost

$d$ = rate of depreciation (per year)

$n$ = number of years

$p$ = present value after y years.

If P is considered the scrap value at the end of useful life of the item, write a program to compute the useful life in years given the original cost, depreciation rate, and the scrap value.

The program should request the user to input the data interactively.

**6.16** Write a program to print a square of size 5 by using the character **S** as shown below:

```
(a) S S S S S        (b) S S S S S
    S S S S S            S       S
    S S S S S            S       S
    S S S S S            S       S
    S S S S S            S S S S S
```

6.17 Write a program to graph the function

$$y = \sin(x)$$

in the interval 0 to 180 degrees in steps of 15 degrees. Use the concepts discussed in the Case Study 4 in Chapter 6.

6.18 Write a program to print all integers that are **not divisible** by either 2 or 3 and lie between 1 and 100. Program should also account the number of such integers and print the result.

6.19 Modify the program of Exercise 6.16 to print the character O instead of S at the center of the square as shown below.

```
S S S S S
S S S S S
S S O S S
S S S S S
S S S S S
```

6.20 Given a set of 10 two-digit integers containing both positive and negative values, write a program using **for** loop to compute the sum of all positive values and print the sum and the number of values added. The program should use **scanf** to read the values and terminate when the sum exceeds 999. Do not use **goto** statement.