5 DECISION MAKING AND BRANCHING

Key Terms

Decision-making statements I switch statement I Conditional operator I goto statement I Infinite loop.

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.

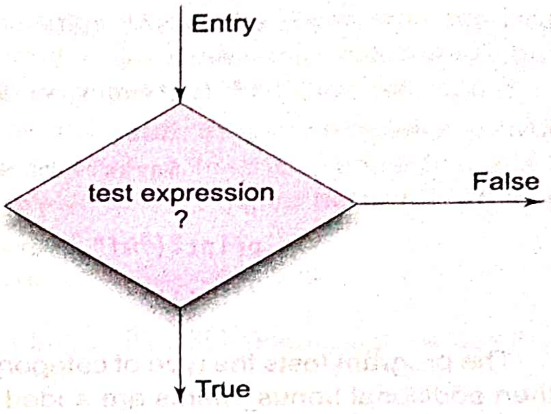


Fig. 5.1 *Two-way branching*

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 section.

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.

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

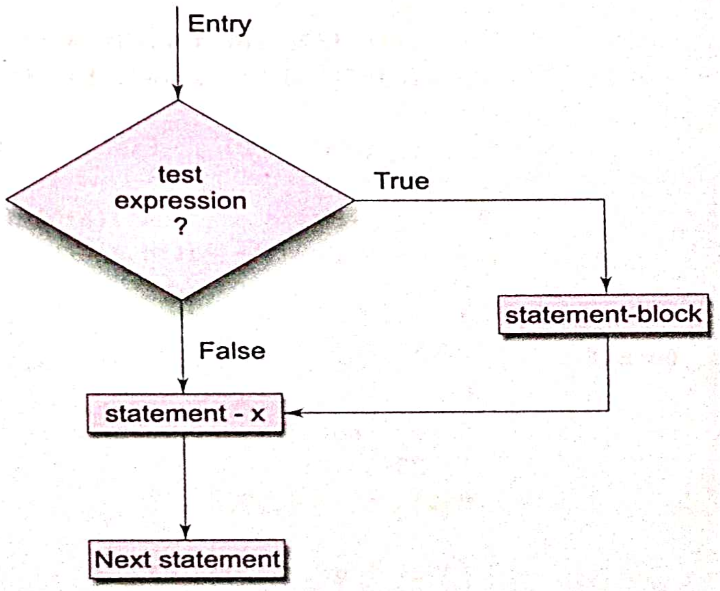


Fig. 5.2 *Flowchart of simple **if** control*

```
.........
.........
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 adde.

Program 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*

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 statemen-x.

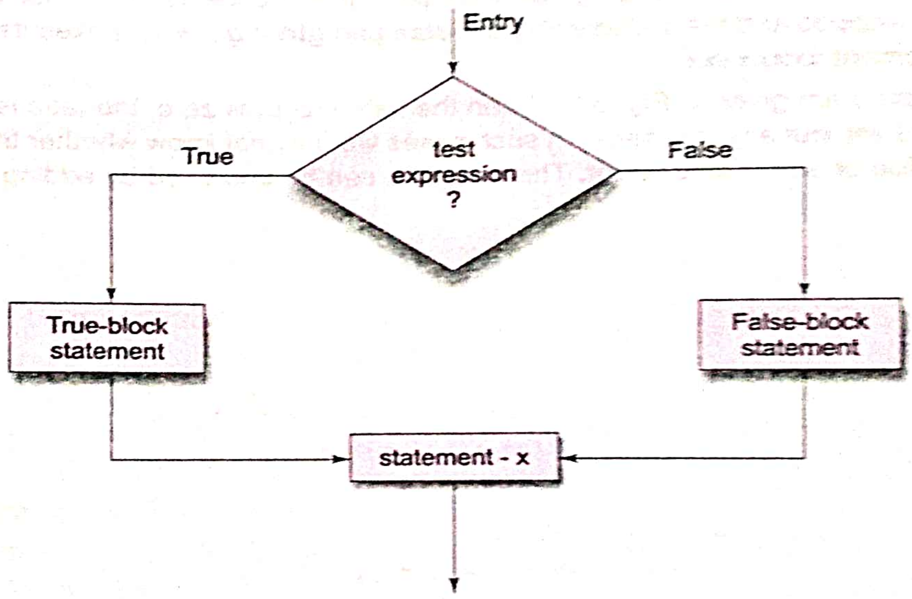


Fig. 5.5 *Flowchart of if......else control*

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");
..........
.........
```

Program 5.3 A program to evaluate the power series.

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

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

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

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
```

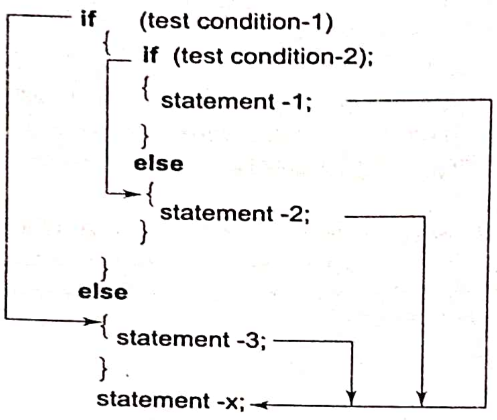
Fig. 5.6 *Illustration of if...else statement*

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,



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

the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statemet-x.

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;
.........
.........
```

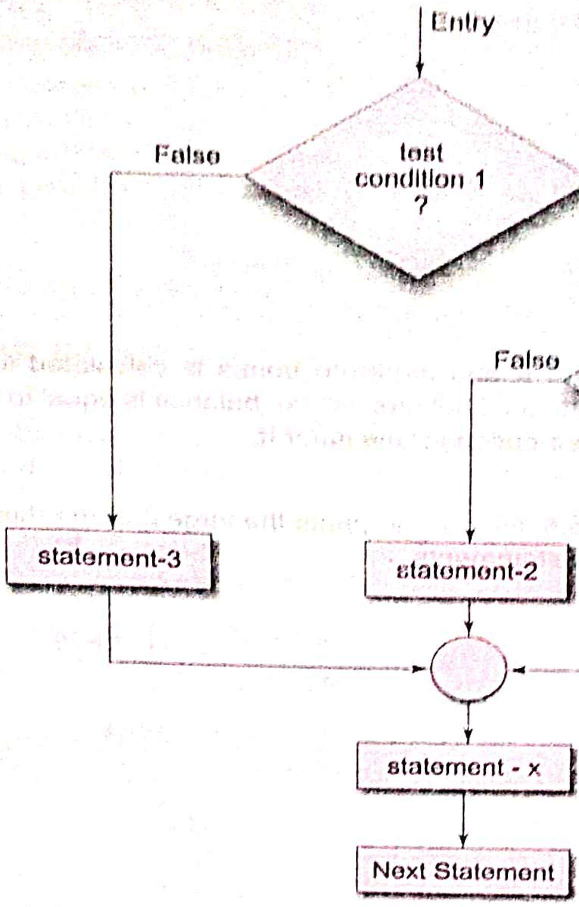


Fig. 5.7 *Flow chart of nested* ***if...else*** *statements*

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**.

Program 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. A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**. It takes the following general form:

```
if ( condition 1)
     statement-1;
else if ( condition 2)
          statement-2;
     else if ( condition 3)
               statement-3;
          else if ( condition n)
                    statement-n;
               else
                    default-statement;
     statement-x;
```

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

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

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

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

```
        if (marks > 79)
                grade = "Honours";
        else if (marks > 59)
                grade = "First Division";
        else if (marks > 49)
                grade = "Second Division";
          else if (marks > 39)
                    grade = "Third Division";
                else
              grade = "Fail";
    printf ("%s\n", grade);
```

Consider another example given below:

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

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

```
        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.

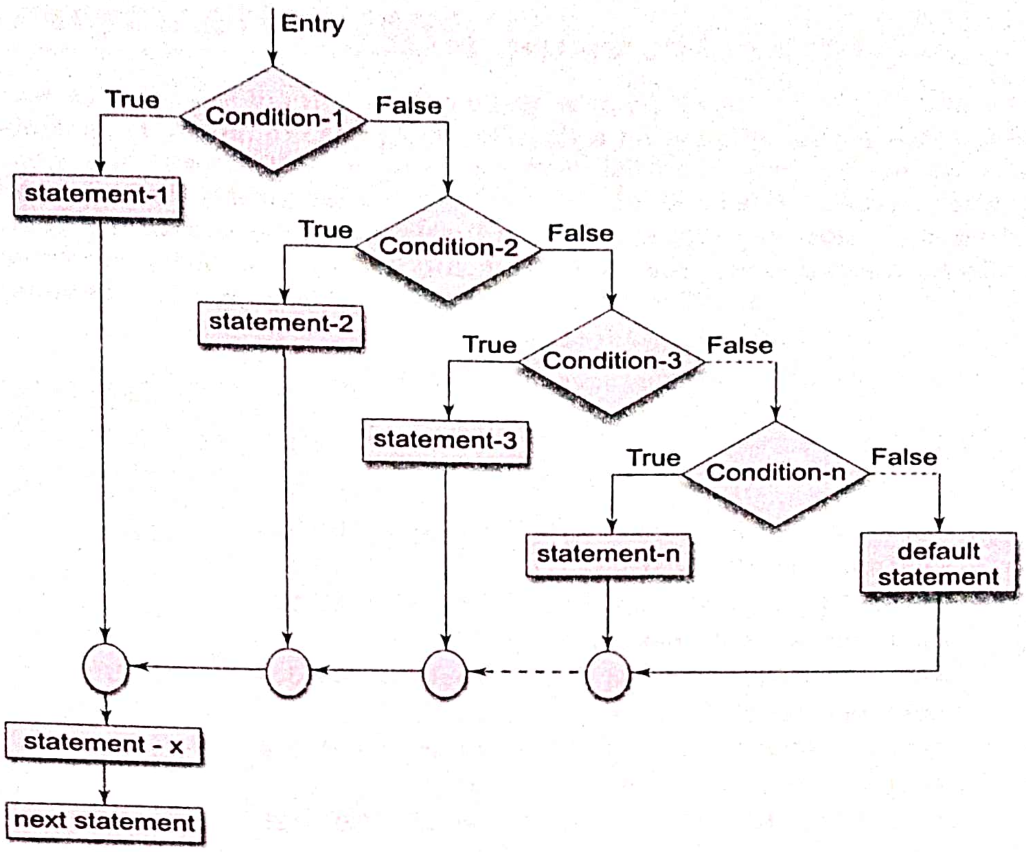


Fig. 5.9 *Flow chart of else..if ladder*

Program 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.

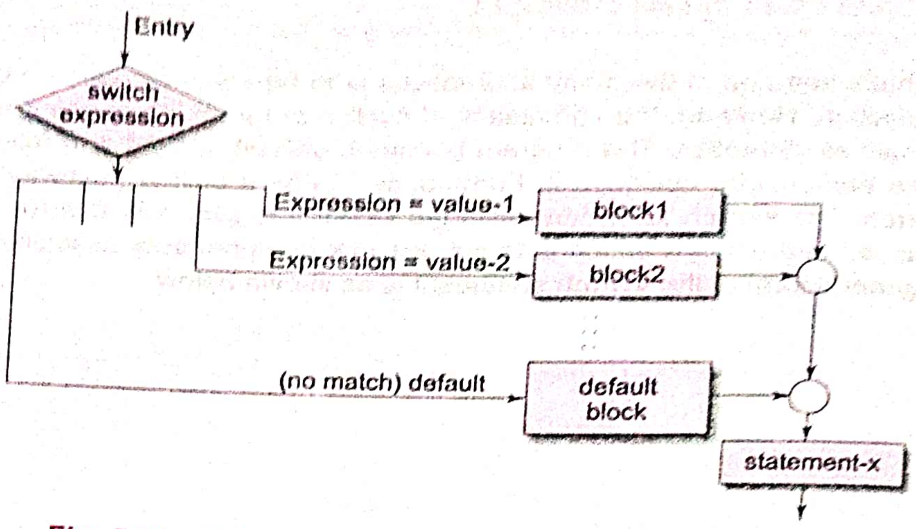


Fig. 5.11 *Selection process of the switch statement*

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 colon.
- 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 ex-pression 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.

Program 5.6

Write a complete C program that reads a value in the range of 1 to 12 and print the name of that month and the next month. Print error for any other input value.

Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
  char month[12][20] = {"January","February","March","April","May","June",
  "July","August","September","October","November","December"};
  int i;

  printf("Enter the month value: ");
  scanf("%d",&i);

  if(i<1 || i>12)
  {
    printf("Incorrect value!!\nPress any key to terminate the program...");
    getch();
    exit(0);
  }

  if(i!=12)
    printf("%s followed by %s",month[i-1],month[i]);
```

```
        else
          printf("%s followed by %s",month[i-1],month[0]);

      getch();
      }
```

Output

```
      Enter the month value: 6
      June followed by July
```

Fig. 5.12 *Program to read and print name of months in the range of 1 and 12*

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 non-zero, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

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

can be written as

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

Consider the evaluation of the following function:

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

$y = 2x + 5$ for $x > 2$

This can be evaluated using the conditional operator as follows:

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

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

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

This complex equation can be written as

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

```
Output
Enter the two numbers whose GCD is to be found: 18 12
GCD of 18 and 12 is 6
```

Fig. 5.14 *Program to determine GCD of two numbers*

Some Guidelines for Writing Multiway Selection Statements

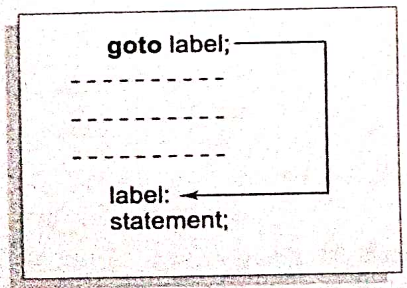
Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.
- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alter-native 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.

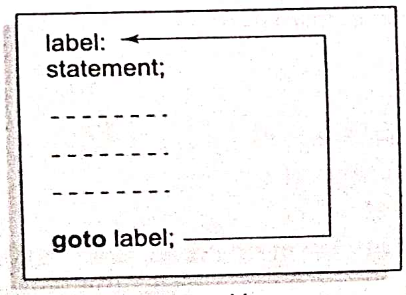
5.9 THE GOTO STATEMENT

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. Program 5.9 illustrates how such infinite loops can be eliminated.

Program 5.9

Program presented in Fig. 5.15 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. 5.15 *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.

6 DECISION MAKING AND LOOPING

Key Terms

Program loop | Control statement | while statement | do statement | continue statement | break statement.

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:

```
                -----------
                -----------
                sum = 0;
                n = 1;
          ┌───► loop:
        L │     sum = sum + n*n;
          │     if (n == 10)
        o │         goto print;──┐
        o │     else             │
        p │     {                │   n = 10,
          │         n = n+1;     │   end of loop
          └──       goto loop;   │
                }                │
                print: ◄─────────┘
                -----------
                -----------
```

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.

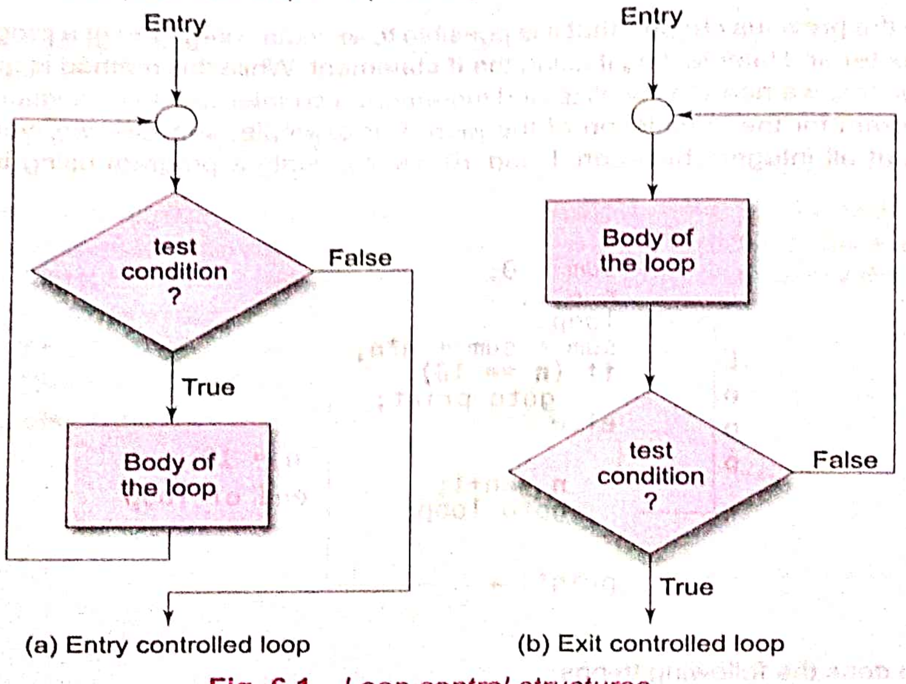


Fig. 6.1 *Loop control structures*

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 specified number of times or to determine whether a particular condition has been met.

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

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

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

Sentinel Loops

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

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

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

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

6.2 THE WHILE STATEMENT

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

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

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

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

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

```
              =========
              sum = 0;
              n = 1;                          /* Initialization */
         ---> while(n <= 10)                  /* Testing */
        |     {
  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*.

Program 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.

Program 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 condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Note *C99 enhances the* **for** *loop by allowing declaration of variables in the initialization permits portion. See the Appendix "C99 Features".*

Consider the following segment of a program:

```
                 for ( x = 0 ; x <= 9 ; x = x+1)
loop       ┌──→ {
           │         printf("%d", x);
           └──  }
                printf("\n");
```

This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+1.

The **for** statement allows for *negative increments*. For example, the loop discussed above can be written as follows:

```
        for ( x = 9 ; x >= 0 ; x = x-1 )
            printf("%d", x);
        printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
        for (x = 9; x < 9; x = x-1)
        printf("%d", x);
```

will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers discussed in Section 6.1. This problem can be coded using the **for** statement as follows:

```
------------------
sum = 0;
for (n = 1; n <= 10; n = n+1)
{
        sum = sum+ n*n;
}
printf("sum = %d\n", sum);
------------------
```

The body of the loop

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

is executed 10 times for n = 1, 2, ..., 10 each time incrementing the **sum** by the square of the value of n.

One of the important points about the **for** loop is that all the three actions, namely *initialization, testing*, and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmers and users, in one place. The **for** statement and its equivalent of **while** and **do** statements are shown in Table 6.1.

Table 6.1 *Comparison of the Three Loops*

for	*while*	*do*

for (n=1; n<=10; ++n)
{
____
____
{
	n = 1;
	while (n<=10)
	{
	________
	________
	n = n+1;
	}
		n = 1;
		do
		{
		________
		________
		n = n+1;
		}
		while(n<=10);

```
for (i = 1; i <= n-2; i++)
{
fib=num1 + num2;
num1=num2;
num2=fib;
}
printf("\nnth fibonacci number (for n = %d) = %d", n,fib);
getch();
}
```

Fig. 6.6 *Program to print nth fibonacci number*

Additional Features of for Loop

The **for** loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)
```

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)
;
```

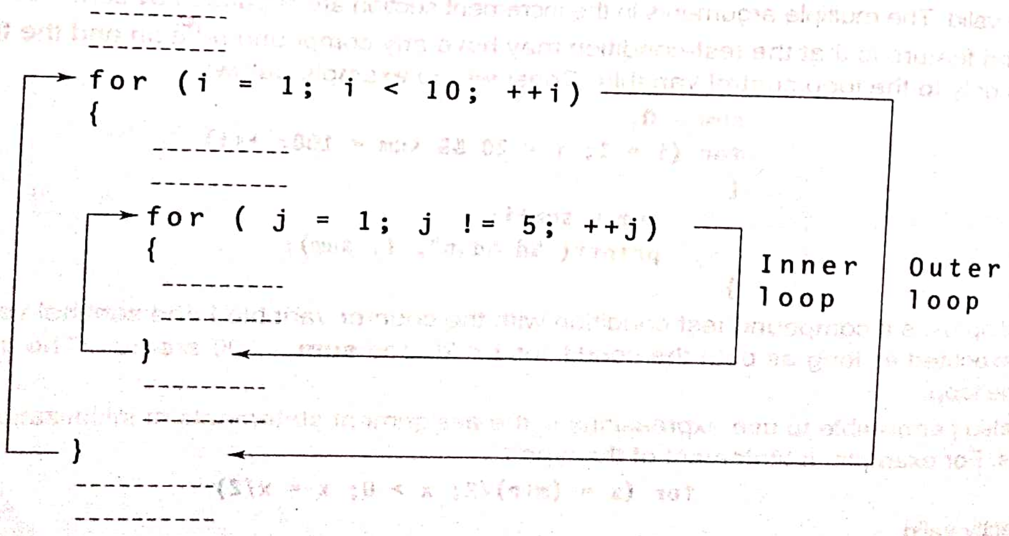
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *null* statement. This can also be written as

```
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:



```
-----------
-----------
for (i = 1; i < 10; ++i)
{
    ----------
    ----------
    for ( j = 1; j != 5; ++j)
    {
        ----------
        ----------
    }
    ----------
    ----------
}
-----------
-----------
```

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 Program 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 *colomns.*

Program 6.6 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. 6.7.

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)
             {
              0
            1 0 1
          2 1 0 1 2
        3 2 1 0 1 2 3
      4 3 2 1 0 1 2 3 4
    5 4 3 2 1 0 1 2 3 4 5
  6 5 4 3 2 1 0 1 2 3 4 5 6
7 6 5 4 3 2 1 0 1 2 3 4 5 6 7
```

Fig. 6.8 *Program to build a pyramid*

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

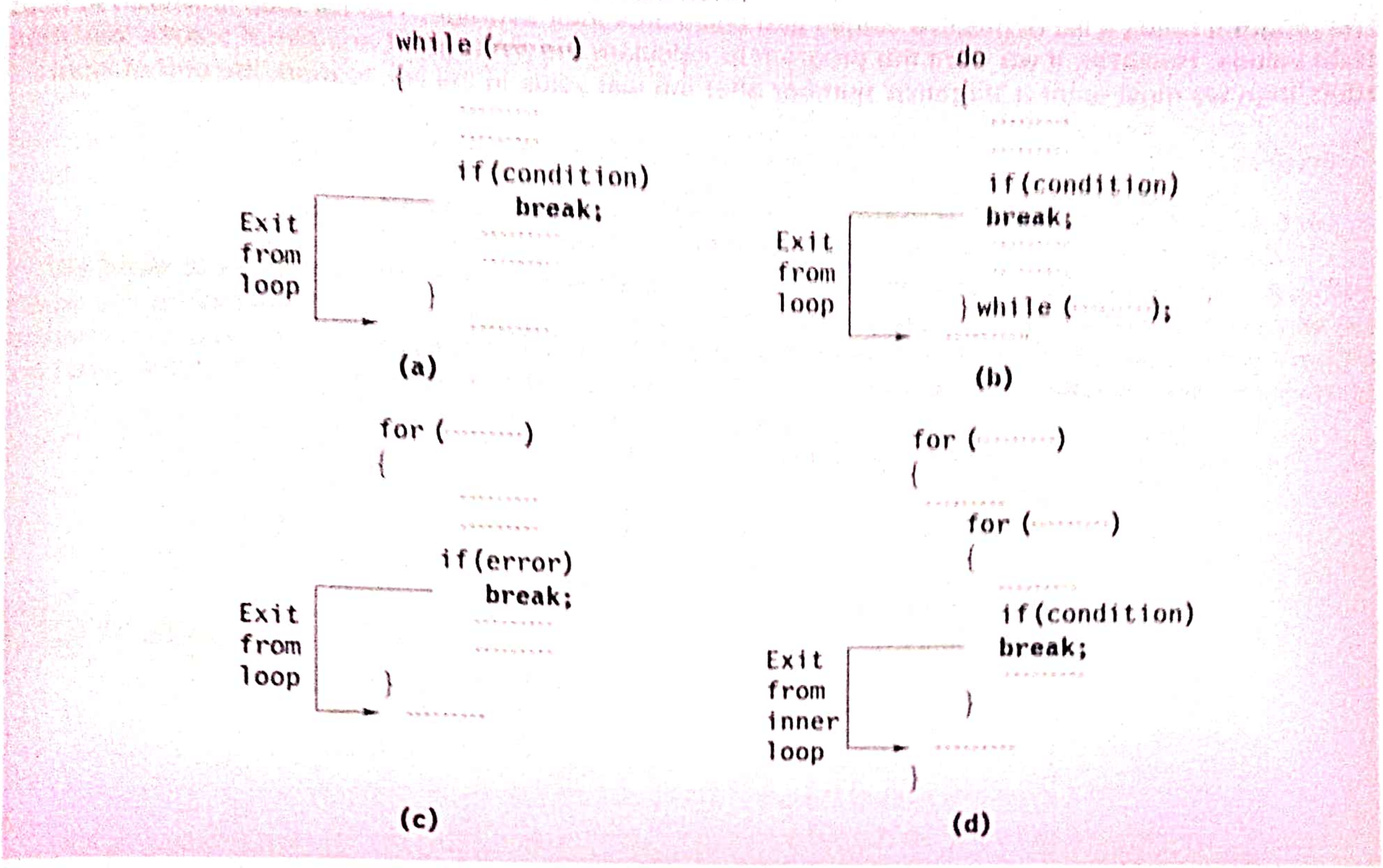
Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop 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 terminated 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. 6.9 and Fig. 6.10.

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.



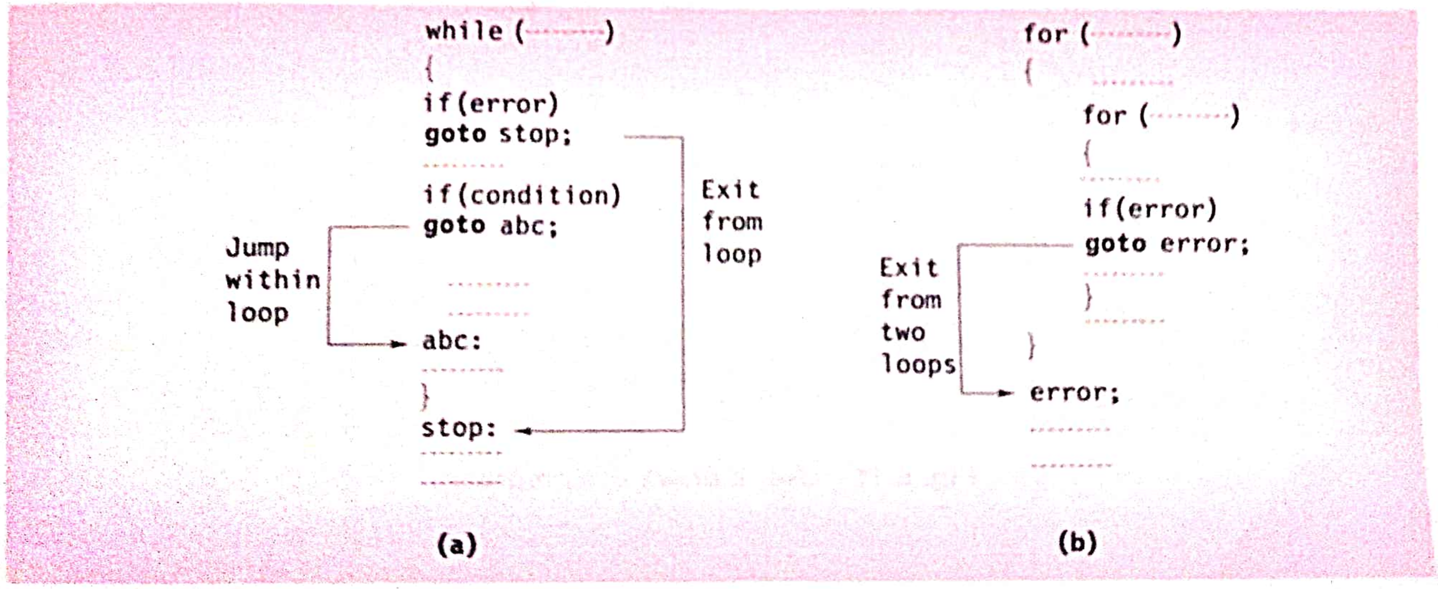
```
(a)
while (.........)
{
    .........
    .........
    if(condition)
        break;
    .........
    .........
}
.........

(b)
do
{
    .........
    .........
    if(condition)
    break;
    .........
    .........
} while (.........);
.........

(c)
for (.........)
{
    .........
    .........
    if(error)
        break;
    .........
    .........
}
.........

(d)
for (.........)
{
    .........
    for (.........)
    {
        .........
        if(condition)
        break;
        .........
    }
    .........
}
```

Fig. 6.9 *Exiting a loop with **break** statement*



```
(a)
while (.........)
{
if(error)
goto stop;
.........
if(condition)
goto abc;
   .........
   .........
abc:
.........
}
stop:
.........
.........

(b)
for (.........)
{
    .........
    for (.........)
    {
    .........
    if(error)
    goto error;
    .........
    }
    .........
}
error;
.........
.........
```

Fig. 6.10 *Jumping within and exiting from the loops with **goto** statement*

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.

Program 6.9 A program to evaluate the series.

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

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 6.12. 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. 6.12 *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!

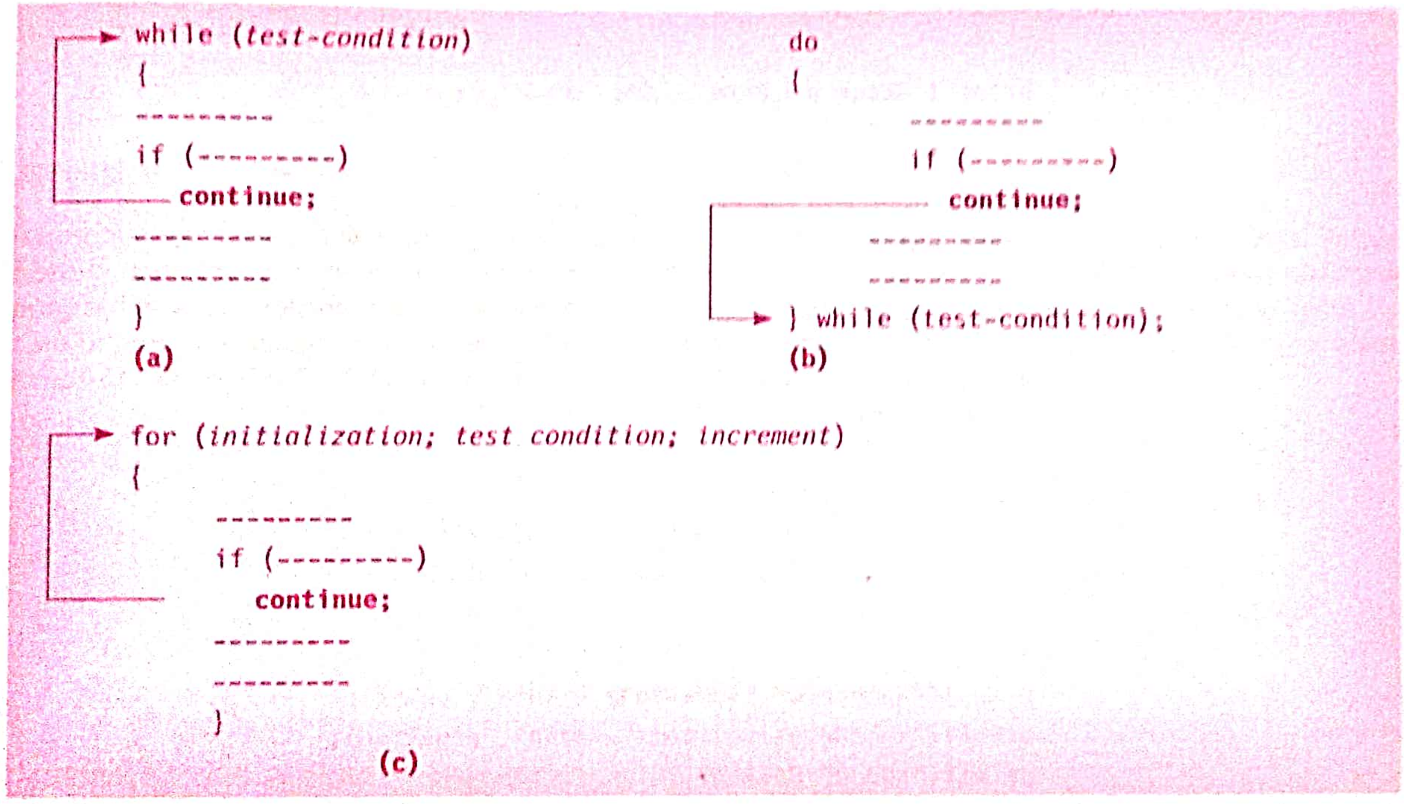
Skipping a Part of a Loop

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. 6.13. 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.



```
while (test-condition)                do
{                                     {
---------                                  ---------
if (---------)                             if (---------)
    continue;                                  continue;
---------                                  ---------
---------                                  ---------
}                                     } while (test-condition);
(a)                                   (b)

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

Fig. 6.13 *Bypassing and continuing i loops*

Program 6.10 The program in Fig. 6.14 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.

7.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional array*. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of *x*. The subscripted variable x_i refers to the ith element of *x*. In C, single-subscripted variable x_i can be expressed as

x[1], x[2], x[3],.........x[n]

The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:

	number [0]
	number [1]
	number [2]
	number [3]
	number [4]

The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This would cause the array **number** to store the values as shown below:

number [0]	35
number [1]	40
number [2]	20
number [3]	57
number [4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
```

```
number[2] = x[5] + y[10];
value[6] = number[1] * 3;
```

The subscripts of an array can be integer constants, integer variables like i, or expressions that yield integers. *C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.*

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

type ***variable-name[size];***

The *type* specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **name[10]** holds the null character '\0'. *When declaring character arrays, we must allow one extra element space for the null terminator.*

Program 7.1 Write a program using a single-subscripted variable to evaluate the following expressions:

$$\text{Total} = \sum_{i=1}^{10} x_i^2$$

The values of x1,x2,....are read from the terminal.

Program in Fig. 7.1 uses a one-dimensional array **x** to read the values and compute the sum of their squares.

Program

```
           main()
           {
                int i ;
                float x[10], value, total ;

           /* . . . . . .READING VALUES INTO ARRAY . . . . . . . */

                printf("ENTER 10 REAL NUMBERS\n") ;

                for( i = 0 ; i < 10 ; i++ )
                {
                    scanf("%f", &value) ;
                    x[i] = value ;
                }
           /* . . . . . . . .COMPUTATION OF TOTAL . . . . . . . .*/

                total = 0.0 ;
                for( i = 0 ; i < 10 ; i++ )
                    total = total + x[i] * x[i] ;

           /*. . . . PRINTING OF x[i] VALUES AND TOTAL . . . */

                printf("\n");
                for( i = 0 ; i < 10 ; i++ )
                    printf("x[%2d] = %5.2f\n", i+1, x[i]) ;

                printf("\ntotal = %.2f\n", total) ;
           }
```

Output

```
                ENTER 10 REAL NUMBERS

                1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

                              x[ 1] = 1.10
                              x[ 2] = 2.20
                              x[ 3] = 3.30
                              x[ 4] = 4.40
                              x[ 5] = 5.50
                              x[ 6] = 6.60
                              x[ 7] = 7.70
                              x[ 8] = 8.80
                              x[ 9] = 9.90
                              x[10] = 10.10

                              Total = 446.86
```

Fig. 7.1 *Program to illustrate **one-dimensional** array*

> ***Note*** *C99 permits arrays whose size can be specified at run time. See Appendix "C99 Features".*

7.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

type ***array-name[size] = { list of values };***

The values in the list are separated by commas. For example, the statement

```
int number[3] = { 0,0,0 };
```

will declare the variable **number** as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

```
float total[5] = {0.0,15.75,-10};
```

will initialize the first three elements to 0.0, 15.75, and –10.0 and the remaining two elements to zero.

The *size* may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter[ ] = {1,1,1,1};
```

will declare the **counter** array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name[ ] = {'J','o', 'h', 'n', '\0'};
```

declares the **name** to be an array of five characters, initialized with the string "John" ending with the null character. Alternatively, we can assign the string literal directly as under:

```
char name [ ] = "John";
```

(Character arrays and strings are discussed in detail in Chapter 8.)

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are inilialized to *zero*, if the array type is numeric and *NULL* if the type is char. For example,

```
int number [5] = {10, 20};
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

```
char city [5] = {'B'};
```

will initialize the first element to 'B' and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40};
```

will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
--------
--------
for (i = 0; i < 100; i = i+1)
{
   if    i < 50
         sum[i] = 0.0;             /* assignment statement */
   else
         sum[i] = 1.0;
}
--------
--------
```

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list*. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort
- Selection sort
- Insertion sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key*. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

7.5 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item1	*Item2*	*Item3*
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as $\mathbf{v}_{ij}$. Here **v** denotes the entire matrix and $\mathbf{v}_{ij}$ refers to the value in the ith row and jth column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

v[4][3]

Two-dimensional arrays are declared as follows:

type *array_name* [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Program

```
        #define ROWS        5
        #define COLUMNS     5
        main()
        {
             int row, column, product[ROWS][COLUMNS] ;
             int i, j ;
             printf(" MULTIPLICATION TABLE\n\n") ;
             printf(" ") ;
             for( j = 1 ; j <= COLUMNS ; j++ )
                printf("%4d" , j ) ;
             printf("\n") ;
             printf("———————————————————————————\n");
             for( i = 0 ; i < ROWS ; i++ )
             {
                  row = i + 1 ;
                  printf("%2d |", row) ;
                  for( j = 1 ; j <= COLUMNS ; j++ )
                  {
                       column = j ;
                       product[i][j] = row * column ;
                       printf("%4d", product[i][j] ) ;
                       }
                       printf("\n") ;
             }
        }
```

Output

```
                MULTIPLICATION TABLE
                  1    2    3    4    5
           ____________________________
           1 |    1    2    3    4    5
           2 |    2    4    6    8   10
           3 |    3    6    9   12   15
           4 |    4    8   12   16   20
           5 |    5   10   15   20   25
```

Fig. 7.6 *Program to print multiplication table using two-dimensional array*

7.6 INITIALIZING TWO-DIMENSIONAL ARRAYS

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2][3] = { 0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2][3] = {{0,0,0}, {1,1,1}};
```

by surrounding the elements of the each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[2][3] = {
                        {0,0,0},
                        {1,1,1}
                  };
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement

```
int table [ ] [3] = {
                        { 0, 0, 0},
                        { 1, 1, 1}
                    };
```

is permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
int table[2][3] = {
                        {1,1},
                        {2}
                  };
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5] = { {0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```

Program 7.6 A survey to know the popularity of four cars (Ambassador, Fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

```
        printf("%d\t",b[i][j]);
        }
        }
        getch();
        }
```

Output

```
    Enter a 3 X 3 matrix:
    a[0][0] = 1
    a[0][1] = 2
    a[0][2] = 3
    a[1][0] = 4
    a[1][1] = 5
    a[1][2] = 6
    a[2][0] = 7
    a[2][1] = 8
    a[2][2] = 9
    The entered matrix is:
      1    2    3
      4    5    6
      7    8    9
    The transpose of the matrix is:
      1    4    7
      2    5    8
      3    6    9
```

Fig. 7.9 *Program to find transpose of a matrix*

7.7 MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multi-dimensional array is

```
type array_name[s1][s2][s3]....[sm];
```

where s_i is the size of the ith dimension. Some example are:

```
int survey[3][5][12];
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly **table** is a four-dimensional array containing 300 elements of floating-point type.

The array **survey** may represent a survey data of rainfall during the last three years from January to December in five cities.

If the first index denotes year, the second city and the third month, then the element **survey[2][3][10]** denotes the rainfall in the month of October during the second year in city-3.

```
        i=0;
        while(str1[i] == str2[i] && str1[i] != '\0'
            && str2[i] != '\0')
          i = i+1;
        if (str1[i] == '\0' && str2[i] == '\0')
            printf("strings are equal\n");
          else
            printf("strings are not equal\n");
```

8.8 STRING-HANDLING FUNCTIONS

3

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	*Action*
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

We shall discuss briefly how each of these functions can be used in the processing of strings.

strcat() Function

The **strcat** function joins two strings together. It takes the following form:

strcat(string1, string2);

string1 and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**. It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged. For example, consider the following three strings:

	0	1	2	3	4	5	6	7	8	9	0	1
Part1 =	V	E	R	Y		\0						

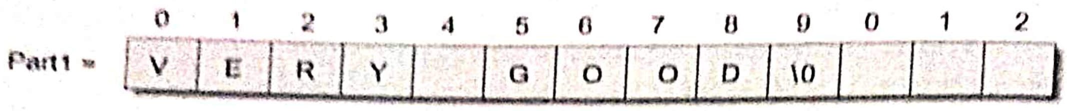
	0	1	2	3	4	5	6
Part2 =	G	O	O	D	\0		

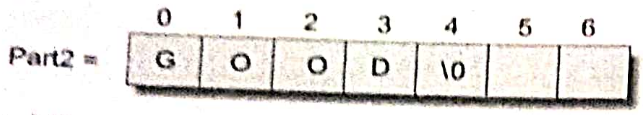
	0	1	2	3	4	5	6
Part3 =	B	A	D	\0			

Execution of the statement

```
strcat(part1, part2);
```

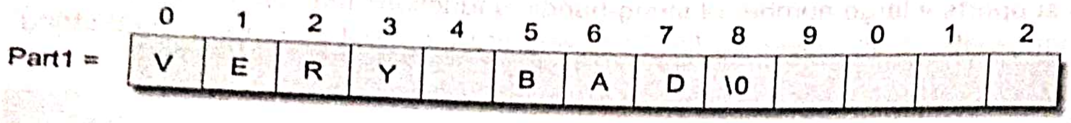
will result in:

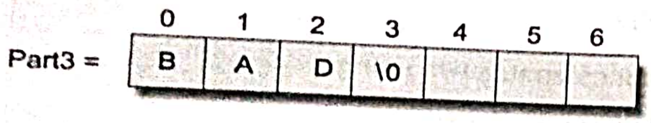




while the statement

will result in:





We must make sure that the size of **string1** (to which **string2** is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat(part1,"GOOD");
```

C permits nesting of **strcat** functions. For example, the statement

```
strcat(strcat(string1,string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in **string1**.

strcmp() Function

The **strcmp** function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

strcmp(string1, string2);

string1 and **string2** may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
strcmp(name1, "John");
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

will return a value of –9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is –9. If the value is negative, **string1** is alphabetically above **string2**.

strcpy() Function

The **strcpy** function works almost like a string assignment operator. It takes the form:

strcpy(string1, string2);

and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable **city**. Similarly, the statement

```
strcpy(city1, city2);
```

will assign the contents of the string variable **city2** to the string variable **city1**. The size of the array **city1** should be large enough to receive the contents of **city2**.

strlen() Function

This function counts and returns the number of characters in a string. It takes the form

n = strlen(string);

Where **n** is an integer variable, which receives the value of the length of the **string**. The argument may be a string constant. The counting ends at the first null character.

Program 8.9

s1, **s2**, and **s3** are three string variables. Write a program to read two string constants into **s1** and **s2** and compare whether they are equal or not. If they are not, join them together. Then copy the contents of **s1** to the variable **s3**. At the end, the program should print the contents of all the three variables and their lengths.

The program is shown in Fig. 8.10. During the first run, the input strings are "New" and "York". These strings are compared by the statement

```
x = strcmp(s1, s2);
```

Since they are not equal, they are joined together and copied into **s3** using the statement

```
strcpy(s3, s1);
```

The program outputs all the three strings with their lengths.

During the second run, the two strings **s1** and **s2** are equal, and therefore, they are not joined together. In this case all the three strings contain the same string constant "London".

Program

```
        #include <string.h>
        main()
        {  char s1[20], s2[20], s3[20];
           int x, l1, l2, l3;
           printf("\n\nEnter two string constants \n");
           printf("?");
```