1. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig.2.11.

AVERAGE OF n VALUES

Program

```
#define      N    10                   /* SYMBOLIC CONSTANT */

main()
{
    int     count ;                    /* DECLARATION OF */
    float   sum, average, number ;     /* VARIABLES      */

    sum   = 0 ;                        /* INITIALIZATION */
    count = 0 ;                        /* OF VARIABLES   */

    while( count < N )
    {
        scanf("%f", &number) ;
        sum    = sum + number ;
        count = count + 1 ;
    }

    average = sum/N ;
    printf("N = %d   Sum = %f", N, sum);
    printf("  Average = %f", average);
}
```

Output

```
1
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10   Sum = 38.799999   Average = 3.880000
```

Fig. 2.11 Average of N numbers

The variable **number** is declared as **float** and therefore it can take both integer and real numbers. Since the symbolic constant **N** is assigned the value of 10 using the **#define** statement, the program accepts ten values and calculates their sum using the **while** loop. The variable **count** counts the number of values and as soon as it becomes 11, the **while** loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

2. Temperature Conversion Problem

The program presented in Fig.2.12 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

FAHRENHEIT – CELSIUS CONVERSION TABLE

_____

Program

```
#define   F_LOW      0      /* --------------------- */
#define   F_MAX    250      /* SYMBOLIC CONSTANTS    */
#define   STEP      25      /* --------------------- */

main()
{
    typedef  float  REAL ;       /* TYPE DEFINITION  */
    REAL    fahrenheit, celsius ;  /* DECLARATION      */

    fahrenheit = F_LOW ;           /* INITIALIZATION   */
    printf("Fahrenheit   Celsius\n\n") ;
    while( fahrenheit <= F_MAX )
    {
        celsius = ( fahrenheit - 32.0 ) / 1.8 ;
        printf("  %5.1f      %7.2f\n", fahrenheit, celsius);
        fahrenheit = fahrenheit + STEP ;
    }
}
```

_____

Output

```
     Fahrenheit    Celsius

          0.0       -17.78
         25.0        -3.89
         50.0        10.00
         75.0        23.89
        100.0        37.78
        125.0        51.67
        150.0        65.56
        175.0        79.44
        200.0        93.33
        225.0       107.22
        250.0       121.11
```
_____

Fig. 2.12 Temperature conversion

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the **#define** statements. An user-defined data type name **REAL** is used to declare the variables **fahrenheit** and **celsius**.

The formation specifications %5.1f and %7.2 in the second **printf** statement produces two-column output as shown.

CASE STUDIES

1. SALESMAN'S SALARY

A computer manufacturing company has the following monthly compensation policy to their sales-persons:

>   Minimum base salary   : 1500.00
>   Bonus for every computer sold   :   200.00
>   Commission on the total monthly sales :   2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month. A program to compute a sales-person's gross salary is given in Fig.3.9.

```
                PROGRAM TO CALCULATE A SALESMAN'S SALARY
```

Program

```
    #define   BASE_SALARY   1500.00
    #define   BONUS_RATE     200.00
    #define   COMMISSION       0.02

    main()
    {
        int    quantity ;
        float  gross_salary, price ;
        float  bonus, commission ;

        printf("Input number sold and price\n") ;
        scanf("%d  %f", &quantity, &price) ;

        bonus       = BONUS_RATE * quantity ;
        commission  = COMMISSION * quantity * price ;
        gross_salary = BASE_SALARY + bonus + commission ;

        printf("\n");
        printf("Bonus        = %6.2f\n", bonus) ;
        printf("Commission   = %6.2f\n", commission) ;
        printf("Gross salary = %6.2f\n", gross_salary) ;
    }
```

Output

```
      Input number sold and price
      5   20450.00

      Bonus              = 1000.00
      Commission         = 2045.00
      Gross salary       = 4545.00
```

Fig. 3.9 Program of salesman's salary

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are, the price of each computer and the number sold during the month.

The gross salary is given by the equation :

Gross salary =   base salary + (quantity * bonus rate)
                + (quantity * Price) * commission rate

2. SOLUTION OF THE QUADRATIC EQUATION

An equation of the form

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

is known as the *quadratic equation*. The values of x that satisfy the equation are known as the *roots* of the equation. A quadratic equation has two roots which are given by the following two formulae:

$$\text{root 1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

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

A program to evaluate these roots is given in Fig.3.10. The program requests the user to input the values of **a**, **b** and **c** and outputs **root1** and **root2**.

SOLUTION OF QUADRATIC EQUATION

Program

```
#include   <math.h>

main()
{
   float   a, b, c, discriminant,
           root1, root2;

   printf("Input values of a, b, and c\n");
   scanf("%f %f %f", &a, &b, &c);

   discriminant = b*b - 4*a*c ;

   if(discriminant < 0)
      printf("\n\nROOTS ARE IMAGINARY\n");
   else
   {
      root1 = (-b + sqrt(discriminant))/(2.0*a);
      root2 = (-b - sqrt(discriminant))/(2.0*a);
      printf("\n\nRoot1 = %5.2f\n\nRoot2 = %5.2f\n",
                 root1,root2 );
   }
}
```

Output

```
Input values of a, b, and c
2 4 -16

Root1 =   2.00

Root2 = -4.00

Input values of a, b, and c
1 2 3

ROOTS ARE IMAGINARY
```

Fig.3.10 Solution of a quadratic equation

The term (b^2-4ac) is called the *discriminant*. If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

CASE STUDIES

1. Inventory Report

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

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

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

INVENTORY REPORT

Code	Quantity	Rate	Value
----	----	----	----
----	----	----	----
----	---	---	----
----	---	---	----

Total Value: ------

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

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

Program

```c
#define ITEMS 4

main()
{   /* BEGIN */

    int i, quantity[5];
    float rate[5], value, total_value;
    char code[5][5];

    /*   READING VALUES   */

    i = 1;
    while ( i <= ITEMS)
    {
      printf("Enter code, quantity, and rate:");
      scanf("%s %d %f", code[i], &quantity[i],&rate[i]);
      i++;
    }
/*.......Printing of Table and Column Headings.......*/

    printf("\n\n");
    printf("            INVENTORY REPORT           \n");
    printf("-------------------------------------------\n");
    printf(" Code       Quantity   Rate      Value     \n");
    printf("-------------------------------------------\n");

/*.......Preparation of Inventory Position..........*/

    total_value = 0;
    i = 1;

    while ( i <= ITEMS)
    {
       value = quantity[i] * rate[i];
       printf("%5s %10d %10.2f %e\n",code[i],quantity[i],
              rate[i],value);
       total_value += value;
       i++;
    }
/*.......Printing of End of Table..................*/

    printf("-----------------------------------------------\n");
    printf("               Total Value = %e\n",total_value);
    printf("-------------------------------------------\n");

}   /* END */
```

Output

```
Enter code, quantity, and rate:F105  275  575.00

Enter code, quantity, and rate:H220  107   99.95

Enter code, quantity, and rate:I019  321  215.50

Enter code, quantity, and rate:M315   89  725.00
```

```
             INVENTORY REPORT
--------------------------------------------------
   Code      Quantity   Rate       Value
--------------------------------------------------
   F105         275      575.00    1.581250e+005

   H220         107       99.95    1.069465e+004

   I019         321      215.50    6.917550e+004

   M315          89      725.00    6.452500e+004

--------------------------------------------------
             Total Value = 3.025202e+005
--------------------------------------------------
```

Fig.4.12 *Program for inventory report*

2. Reliability Graph

Problem: The reliability of an electronic component is given by

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

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

Problem

```c
#include <math.h>
#define  LAMDA   0.001

main()

{
     double t;
     float r;
     int  i, R;

     for (i=1; i<=27; ++i)
     {
         printf("--");
     }

     printf("\n");

     for (t=0; t<=3000; t+=150)
     {
         r = exp(-LAMDA*t);
         R = (int)(50*r+0.5);
         printf(" |");

         for (i=1; i<=R; ++i)
         {
            printf("*");
         }

         printf("#\n");
     }

     for (i=1; i<3; ++i)

     {
         printf(" |\n");
     }

}
```

`Output`

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

Fig.4.13   *Program to draw reliability graph*

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

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

in steps of 150. The integer 50 in the statement

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

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

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.

RANGE OF NUMBERS

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.

<div align="center">PAY-BILL CALCULATIONS</div>

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()
```

```c
{
    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*

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

   $$B(m,x) \; = \; \binom{m}{x} \; = \; \frac{m!}{x! \, (m-x)!} \; , \qquad m \geq x$$

   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:

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

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

   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	m x	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 TO DRAW **HISTOGRAM**

Program:

```
#define    N    5
main()
{
    intvalue[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:

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.

PROBLEM OF **MINIMUM COST**

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).

```
                    PLOTTING OF TWO FUNCTIONS
```

Program

```c
#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");
}
```

```
                                       Y ------->
     0 -----------------------------------------------------
       |                                                    #
       |                                          0---*
       |                                       0------*
       |                                    0-------*
       |                                 0------*
       |                              0------*
       |                           0----*
       |                        0-*
       |                   #
       |                *-0
   X   |             *---0
       |           *-----0
       |         *------0
       |       *-------0
       |     *-------0
       |    *-------0
       | *-------0
       |*-------0
       |*------0
       |*-----0
       |
```
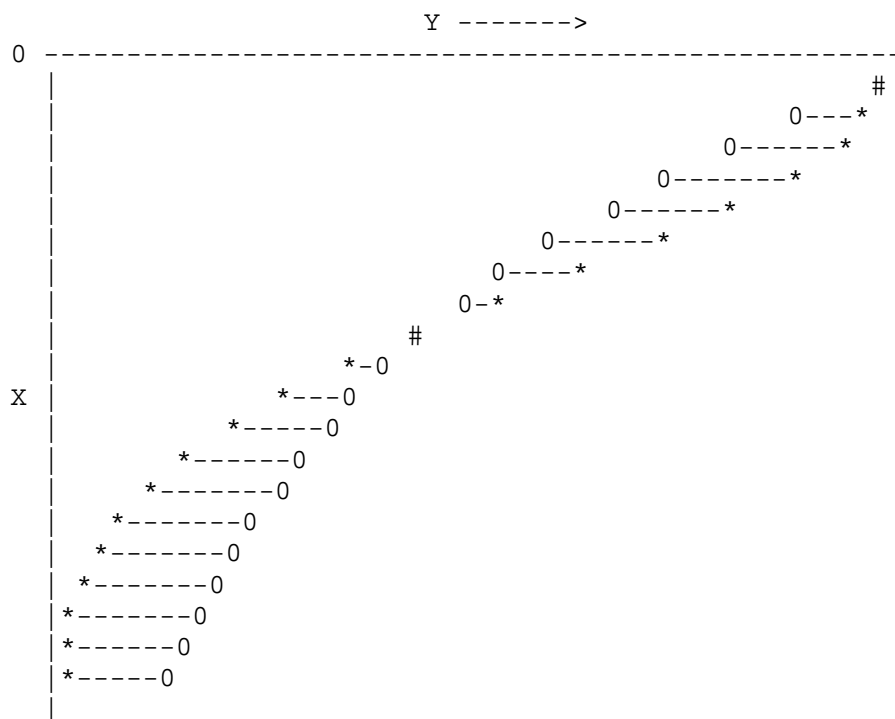
Fig.6.15 *Plotting of two functions*

CASE STUDIES

7.1. Median of a List of Numbers

When all the items in a list are arranged in order, the middle value which divides the items into two parts with equal number of items on either side is called the median. Odd number of items have just one middle value while even number of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

The major steps for finding the median are as follows:

1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in Fig.7.7. The sorting algorithm used is as follows:

1. Compare the first two elements in the list, say a[1], and a[2]. If a[2] is smaller than a[1], then interchange their values.
2. Compare a[2] and a[3]; interchange them if a[3] is smaller than a[2].
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps n-1 times.

In repeated trips through the array, the smallest elements 'bubble up' to the top. Because of this bubbling up effect, this algorithm is called bubble sorting. The bubbling effect is illustrated below for four items.
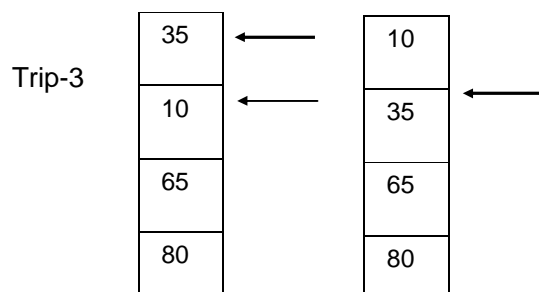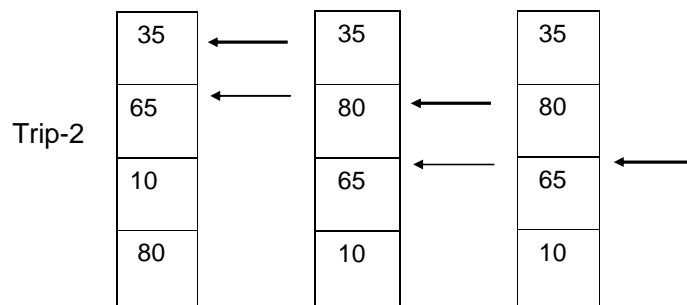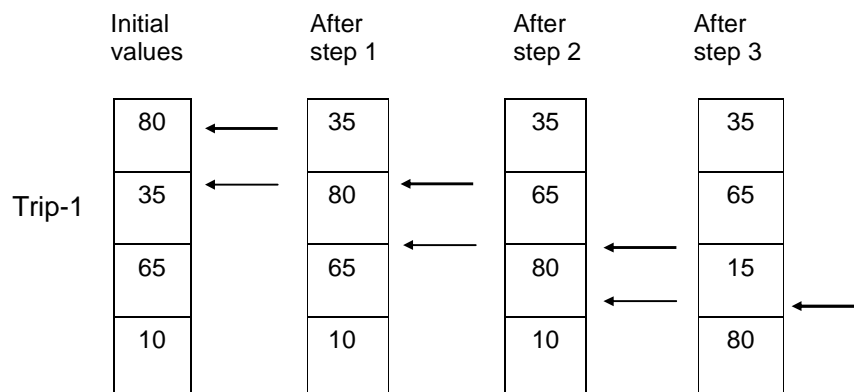
During the first trip, three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65 (the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a trip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains n elements, then the number of comparisons involved would be n(n-1)/2.

PROGRAM TO SORT A LIST AND FIND ITS MEDIAN

Program

```
#define N 10

main( )
{
    int i,j,n;
    float median,a[N],t;

    printf("Enter the number of items\n");
    scanf("%d", &n);

  /* Reading items into array a   */

    printf("Input %d values \n",n);
    for (i = 1; i <= n ; i++)
        scanf("%f", &a[i]);

  /* Sorting begins */

    for (i = 1 ; i <= n-1 ; i++)
    {     /* Trip-i begins  */

        for (j = 1 ; j <= n-i ; j++)
        {
            if (a[j] <= a[j+1])
            { /* Interchanging values */
              t = a[j];
              a[j] = a[j+1];
              a[j+1] = t;
            }
            else
              continue ;
        }
    } /* sorting ends */

  /* calculation of median   */

    if ( n % 2 == 0)
        median = (a[n/2] + a[n/2+1])/2.0 ;
```

```
            else
                median = a[n/2 + 1];

        /*  Printing */

            for (i = 1 ; i <= n ; i++)
                printf("%f  ", a[i]);
            printf("\n\nMedian is %f\n", median);

    }
```

Output

```
    Enter the number of items
    5
    Input 5 values
    1.111   2.222   3.333   4.444   5.555
    5.555000   4.444000   3.333000   2.222000   1.111000

    Median is 3.333000


    Enter the number of items
    6
    Input 6 values
    3   5   8   9   4   6
    9.000000   8.000000   6.000000   5.000000   4.000000   3.000000

    Median is 5.500000
```

Fig.7.7 *Program to sort a list of numbers and to determine median*

2. Calculation of Standard Deviation

In statistics, standard deviation is used to measure deviation of data from its mean. The formula for calculating standard deviation of n items is

$$s = \sqrt{variance}$$

where

$$variance = \frac{1}{n} \sum_{i=1}^{n} (x_i - m)^2$$

and

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

The algorithm for calculating the standard deviation is as follows:

1. Read n items.
2. Calculate sum and mean of the items.
3. Calculate variance.
4. Calculate standard deviation.

Complete program with sample output is shown in Fig.7.8.

PROGRAM TO CALCULATE STANDARD DEVIATION

Program

```c
#include <math.h>
#define    MAXSIZE    100

main( )
{
    int i,n;
    float value [MAXSIZE], deviation,
          sum,sumsqr,mean,variance,stddeviation;

    sum = sumsqr = n = 0 ;

    printf("Input values: input -1 to end \n");
    for (i=1; i< MAXSIZE ; i++)
    {
        scanf("%f", &value[i]);
        if (value[i] == -1)
           break;
        sum += value[i];
        n += 1;
    }
    mean = sum/(float)n;

    for (i = 1 ; i<= n; i++)
    {
        deviation = value[i] - mean;
        sumsqr += deviation * deviation;
    }
    variance = sumsqr/(float)n ;
    stddeviation = sqrt(variance) ;

    printf("\nNumber of items : %d\n",n);
    printf("Mean : %f\n", mean);
    printf("Standard deviation : %f\n", stddeviation);
}
```

Output

```
Input values: input -1 to end
65   9   27   78   12   20   33   49   -1

Number of items : 8
Mean : 36.625000
Standard deviation : 23.510303
```

Fig 7.8 *Program to calculate standard deviation*

3. Evaluating a Test

A test consisting of 25 multiple-choice items is administered to a batch of 3 students. Correct answers and student responses are tabulated as shown below:

Items

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

Correct answers

Student 1
Student 2
Student 3

The algorithm for evaluating the answers of students is as follows:

1. Read correct answers into an array.
2. Read the responses of a student and count the correct ones.
3. Repeat step-2 for each student.
4. Print the results.

A program to implement this algorithm is given in Fig.7.9. The program uses the following arrays:

        key[i]   - To store correct answers of items
        response[i] - To store responses of students
        correct[i]   - To identify items that are answered correctly.

<div align="center">PROGRAM TO EVALUATE A MULTIPLE-CHOICE TEST</div>

Program

```
#define     STUDENTS     3
#define     ITEMS        25
main( )
{
   char  key[ITEMS+1],response[ITEMS+1];
   int   count, i, student,n,
         correct[ITEMS+1];

/*  Reading of Correct answers */

   printf("Input key to the items\n");

   for(i=0; i < ITEMS; i++)
      scanf("%c",&key[i]);

   scanf("%c",&key[i]);
   key[i] = '\0';

   /*  Evaluation begins  */

      for(student = 1; student <= STUDENTS ; student++)
      {
 /*Reading student responses and counting correct ones*/
         count = 0;
         printf("\n");
         printf("Input responses of student-%d\n",student);

         for(i=0; i < ITEMS ; i++)
             scanf("%c",&response[i]);
         scanf("%c",&response[i]);
         response[i] = '\0';

         for(i=0; i < ITEMS; i++)
             correct[i] = 0;

         for(i=0; i < ITEMS ; i++)
            if(response[i] == key[i])
            {
               count = count +1 ;
               correct[i] = 1 ;
            }

         /* printing of results */

         printf("\n");
         printf("Student-%d\n", student);
         printf("Score is %d out of %d\n",count, ITEMS);
         printf("Response to the items below are wrong\n");
```

```
        n = 0;
        for(i=0; i < ITEMS ; i++)
           if(correct[i] == 0)
           {
                printf("%d ",i+1);                               n
                = n+1;
           }
           if(n == 0)
              printf("NIL\n");
           printf("\n");

     } /* Go to next student */

   /* Evaluation and printing ends */
     }
```

Output

```
   Input key to the items
   abcdabcdabcdabcdabcdabcda

   Input responses of student-1
   abcdabcdabcdabcdabcdabcda

   Student-1
   Score is 25 out of 25
   Response to the following items are wrong
   NIL


   Input responses of student-2
   abcddcbaabcdabcdddddddddddd

   Student-2
   Score is 14 out of 25
   Response to the following items are wrong
   5 6 7 8 17 18 19 21 22 23 25

   Input responses of student-3
   aaaaaaaaaaaaaaaaaaaaaaaaaa

   Student-3
   Score is 7 out of 25
   Response to the following items are wrong
   2 3 4 6 7 8 10 11 12 14 15 16 18 19 20 22 23 24
```

Fig 7.9 *Program to evaluate responses to a multiple-choice test*

4. Production and Sales Analysis

A company manufactures five categories of products and the number of items manufactured and sold are recorded product-wise every week in a month. The company reviews its production schedule at every month-end. The review may require one or more of the following information:

(a) Value of weekly production and sales.
(b) Total value of all the products manufactured.
(c) Total value of all the products sold.
(d) Total value of each product, manufactured and sold.

Let us represent the products manufactured and sold by two two-dimensional arrays M and S respectively. Then,

M =

M11	M12	M13	M14	M15
M21	M22	M23	M24	M25
M31	M32	M33	M34	M35
M41	M42	M43	M44	M45

S11	S12	S13	S14	S15
S21	S22	S23	S24	S25
S31	S32	S33	S34	S35
S41	S42	S43	S44	S45

S =

where M_{ij} represents the number of jth type product manufactured in ith week and S_{ij} the number of jth product sold in ith week. We may also represent the cost of each product by a single dimensional array C as follows:

C =

C1	C2	C3	C4	C5

where C_j is the cost of jth type product.

We shall represent the value of products manufactured and sold by two value arrays, namely, **Mvalue** and **Svalue**. Then,

Mvalue[i][j] = M_{ij} x C_j
Svalue[i][j] = S_{ij} x C_j

A program to generate the required outputs for the review meeting is shown in Fig.7.10. The following additional variables are used:

Mweek[i]   = Value of all the products manufactured in week i.

$$= \sum_{j=1}^{5} Mvalue[i][j]$$

Sweek[i]   = Value of all the products sold in week i

$$= \sum_{j=1}^{5} Svalue[i][j]$$

Mproduct[j] = Value of jth type product manufactured during the month

$$= \sum_{i=1}^{4} Mvalue[i][j]$$

Sproduct[j] = Value of jth type product sold during the month

$$= \sum_{i=1}^{4} Svalue[i][j]$$

Mtotal   = Total value of all the products sold during the month

$$= \sum_{i=1}^{4} Mweek[i] = \sum_{j=1}^{5} Mproduct[j]$$

Stotal   = Total value of all the products sold during the month

$$= \sum_{i=1}^{4} Sweek[i] = \sum_{j=1}^{5} Sproduct[j]$$

PRODUCTION AND SALES ANALYSIS

Program

```
main( )
{
    int M[5][6],S[5][6],C[6],
        Mvalue[5][6],Svalue[5][6],
        Mweek[5], Sweek[5],
        Mproduct[6], Sproduct[6],
        Mtotal, Stotal, i,j,number;

/*      Input data          */

    printf (" Enter products manufactured week_wise \n");
    printf (" M11,M12,----,  M21,M22,----  etc\n");


    for(i=1; i<=4; i++)
       for(j=1;j<=5; j++)
          scanf("%d",&M[i][j]);

    printf (" Enter products sold week_wise\n");
    printf (" S11,S12,----,  S21,S22,----  etc\n");

    for(i=1; i<=4; i++)
       for(j=1; j<=5; j++)
          scanf("%d", &S[i][j]);

    printf(" Enter cost of each product\n");
    for(j=1; j <=5; j++)
       scanf("%d",&C[j]);
```

```c
/*  Value matrices of production and sales  */

    for(i=1; i<=4; i++)
       for(j=1; j<=5; j++)
       {
          Mvalue[i][j] = M[i][j] * C[j];
          Svalue[i][j] = S[i][j] * C[j];
       }
/*  Total value of weekly production and sales  */

    for(i=1; i<=4; i++)
    {
       Mweek[i] = 0 ;
       Sweek[i] = 0 ;
       for(j=1; j<=5; j++)
       {
          Mweek[i] += Mvalue[i][j];
          Sweek[i] += Svalue[i][j];
       }
    }
/* Monthly value of product_wise production and sales  */

    for(j=1; j<=5; j++)
    {
       Mproduct[j] = 0 ;
       Sproduct[j] = 0 ;
       for(i=1; i<=4; i++)
       {
          Mproduct[j] += Mvalue[i][j];
          Sproduct[j] += Svalue[i][j];
       }
    }




/*  Grand total of production and sales values  */

    Mtotal = Stotal = 0;
    for(i=1; i<=4; i++)
    {
       Mtotal += Mweek[i];
       Stotal += Sweek[i];
    }

    /***********************************************
     Selection and printing of information required
    ***********************************************/
    printf("\n\n");
    printf(" Following is the list of things you can\n");
    printf(" request for. Enter appropriate item number\n");
    printf(" and press RETURN Key\n\n");

    printf(" 1.Value matrices of production & sales\n");

    printf(" 2.Total value of weekly production & sales\n");

    printf(" 3.Product_wise monthly value of production &");
    printf(" sales\n");
```

```c
        printf(" 4.Grand total value of production & sales\n");

        printf(" 5.Exit\n");

        number = 0;
        while(1)
        {        /* Beginning of while loop */
            printf("\n\n ENTER YOUR CHOICE:");
            scanf("%d",&number);
            printf("\n");

            if(number == 5)
            {
               printf(" G O O D   B Y E\n\n");
               break;
            }

            switch(number)

            { /* Beginning of switch */




/*     V A L U E   M A T R I C E S    */

            case 1:
               printf(" VALUE MATRIX OF PRODUCTION\n\n");
               for(i=1; i<=4; i++)
               {
                  printf(" Week(%d)\t",i);
                  for(j=1; j <=5; j++)
                      printf("%7d", Mvalue[i][j]);
                  printf("\n");
               }
               printf("\n VALUE MATRIX OF SALES\n\n");
               for(i=1; i <=4; i++)
               {
                  printf(" Week(%d)\t",i);
                  for(j=1; j <=5; j++)
                      printf("%7d", Svalue[i][j]);
                  printf("\n");
               }
               break;


/*   W E E K L Y   A N A L Y S I S    */

            case 2:
               printf(" TOTAL WEEKLY PRODUCTION & SALES\n\n");
               printf("                PRODUCTION   SALES\n");
               printf("                ----------   -----\n");
               for(i=1; i <=4; i++)
               {
                  printf(" Week(%d)\t", i);
                  printf("%7d\t%7d\n", Mweek[i], Sweek[i]);
               }
```

```
            break;

    /*   P R O D U C T W I S E   A N A L Y S I S  */

          case 3:
             printf(" PRODUCT_WISE TOTAL PRODUCTION &");
             printf(" SALES\n\n");
             printf("                  PRODUCTION   SALES\n");
             printf("                  ----------   -----\n");
             for(j=1; j <=5; j++)
             {
                printf(" Product(%d)\t", j);
                printf("%7d\t%7d\n",Mproduct[j],Sproduct[j]);
             }
             break;


    /*   G R A N D   T O T A L S    */

          case 4:
             printf(" GRAND TOTAL OF PRODUCTION & SALES\n");
             printf("\n Total production = %d\n", Mtotal);
             printf(" Total sales      = %d\n", Stotal);
             break;

    /*   D E F A U L T   */

          default :
             printf(" Wrong choice, select again\n\n");
             break;


          } /* End of switch */

       } /* End of while loop */

       printf(" Exit from the program\n\n");

    } /* End of main */
```

```
Enter products manufactured week_wise
M11,M12,----,  M21,M22,----  etc
11  15  12  14  13
13  13  14  15  12
12  16  10  15  14
14  11  15  13  12

Enter products sold week_wise
S11,S12,----,  S21,S22,----  etc
10  13   9  12  11
12  10  12  14  10
11  14  10  14  12
12  10  13  11  10
Enter cost of each product
10  20  30  15  25
```

```
Following is the list of things you can
request for. Enter appropriate item number
and press RETURN key

1.Value matrices of production & sales
2.Total value of weekly production & sales
3.Product_wise monthly value of production & sales
4.Grand total value of production & sales
5.Exit
ENTER YOUR CHOICE:1

VALUE MATRIX OF PRODUCTION
 Week(1)              110      300      360      210      325
 Week(2)              130      260      420      225      300
 Week(3)              120      320      300      225      350
 Week(4)              140      220      450      185      300

VALUE MATRIX OF SALES
Week(1)               100      260      270      180      275
Week(2)               120      200      360      210      250
Week(3)               110      280      300      210      300
Week(4)               120      200      390      165      250


 ENTER YOUR CHOICE:2
 TOTAL WEEKLY PRODUCTION & SALES
            PRODUCTION    SALES
            ----------    -----
 Week(1)       1305       1085
 Week(2)       1335       1140
 Week(3)       1315       1200
 Week(4)       1305       1125


 ENTER YOUR CHOICE:3

 PRODUCT_WISE TOTAL PRODUCTION & SALES


             PRODUCTION    SALES
             ----------    -----
 Product(1)      500        450
 Product(2)     1100        940
 Product(3)     1530       1320
 Product(4)      855        765
 Product(5)     1275       1075

 ENTER YOUR CHOICE:4

 GRAND TOTAL OF PRODUCTION & SALES

 Total production = 5260
 Total sales      = 4550

 ENTER YOUR CHOICE:5

 G O O D   B Y E

Exit from the program
```

Fig.7.10 *Program for production and sales analysis*

CASE STUDIES

1. **Counting Words in a Text**

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character in the line, look for a blank. If a blank is   found, increment words by 1.
3. Continue steps 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig.8.11. The first **while** loop will be executed once for each line of text. The end of text is indicated by pressing the `Return' key an extra time after the entire text has been entered. The extra `Return' key causes a newline character as input to the last line and as a result, the last line contains only the null character.

The program checks for this special line using the test

> if (line[0] == `\0')

and if the first (and only the first) character in the line is a null character, then counting is terminated. Note the difference between a null character and a blank character.

COUNTING CHARACTERS, WORDS AND LINES IN A TEXT

Program

```c
#include    <stdio.h>

main()
{
    char line[81], ctr;
    int  i,c,
         end = 0,
         characters = 0,
         words = 0,
         lines = 0;

    printf("KEY IN THE TEXT.\n");
    printf("GIVE ONE SPACE AFTER EACH WORD.\n");
    printf("WHEN COMPLETED, PRESS 'RETURN'.\n\n");

    while( end == 0)
    {
        /* Reading a line of text */

        c = 0;
        while((ctr=getchar()) != '\n')
            line[c++] = ctr;
        line[c] = '\0';

        /* counting the words in a line */

        if(line[0] == '\0')
           break ;
        else
        {
           words++;
           for(i=0; line[i] != '\0';i++)
                 if(line[i] == ' ' || line[i] == '\t')
```

```
                            words++;
              }

              /* counting lines and characters */

              lines = lines +1;
              characters = characters + strlen(line);
       }
       printf ("\n");
       printf("Number of lines = %d\n", lines);
       printf("Number of words = %d\n", words);
       printf("Number of characters = %d\n", characters);
  }
```

Output

```
   KEY IN THE TEXT.
   GIVE ONE SPACE AFTER EACH WORD.
   WHEN COMPLETED, PRESS 'RETURN'.

   Admiration is a very short-lived passion.
   Admiration involves a glorious obliquity of vision.
   Always we like those who admire us but we do not
   like those whom we admire.
   Fools admire, but men of sense approve.


   Number of lines = 5
   Number of words = 36
   Number of characters = 205
```

Fig.8.11 *Counting of characters, words and lines in a text*

The program also counts the number of lines read and the total number of characters in the text. Remember, the last line containing the null string is not counted.

After the first **while** loop is exited, the program prints the results of counting.

2. Processing of a Customer List

Telephone numbers of important customers are recorded as follows:

Full name	Telephone number
Joseph Louis Lagrange	869245
Jean Robert Argand	900823
Carl Freidrich Gauss	806788
-----	---
-----	---

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

   Argand,J.R

We create a table of strings, each row representing the details of one person, such as first_name, middle_name, last_name, and telephone_number. The columns are interchanged as required and the list is sorted on the last_name. Fig.8.12 shows a program to achieve this.

```
                    PROCESSING OF CUSTOMER LIST
```

Program

```
#define  CUSTOMERS    10

main( )
{
    char  first_name[20][10], second_name[20][10],
          surname[20][10], name[20][20],
          telephone[20][10], dummy[20];

    int   i,j;

    printf("Input names and telephone numbers \n");
    printf("?");
    for(i=0; i < CUSTOMERS ; i++)
    {
        scanf("%s %s %s %s", first_name[i],
                second_name[i], surname[i], telephone[i]);

        /* converting full name to surname with initials */

        strcpy(name[i], surname[i] );
        strcat(name[i], ",");
        dummy[0] = first_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
        strcat(name[i], ".");
        dummy[0] = second_name[i][0];
        dummy[1] = '\0';
        strcat(name[i], dummy);
    }
    /* Alphabetical ordering of surnames */

        for(i=1; i <= CUSTOMERS-1; i++)
           for(j=1; j <= CUSTOMERS-i; j++)
              if(strcmp (name[j-1], name[j]) > 0)
              {
              /* Swaping names */
                  strcpy(dummy, name[j-1]);
                  strcpy(name[j-1], name[j]);
                  strcpy(name[j], dummy);

              /* Swaping telephone numbers */
                  strcpy(dummy, telephone[j-1]);
                  strcpy(telephone[j-1],telephone[j]);
                  strcpy(telephone[j], dummy);
              }
        /* printing alphabetical list */
    printf("\nCUSTOMERS LIST IN ALPHABETICAL ORDER \n\n");
    for(i=0; i < CUSTOMERS ; i++)
       printf("  %-20s\t %-10s\n", name[i], telephone[i]);
```

```
    }
```

Output

```
    Input names and telephone numbers
    ?Gottfried Wilhelm Leibniz 711518
    Joseph Louis Lagrange 869245
    Jean Robert Argand 900823
    Carl Freidrich Gauss 806788
    Simon Denis Poisson 853240
    Friedrich Wilhelm Bessel 719731
    Charles Francois Sturm 222031
    George Gabriel Stokes 545454
    Mohandas Karamchand Gandhi 362718
    Josian Willard Gibbs 123145

 CUSTOMERS LIST IN ALPHABETICAL ORDER

        Argand,J.R            900823
        Bessel,F.W            719731
        Gandhi,M.K            362718
        Gauss,C.F             806788
        Gibbs,J.W             123145
        Lagrange,J.L          869245
        Leibniz,G.W           711518
        Poisson,S.D           853240
        Stokes,G.G            545454
        Sturm,C.F             222031
```

Fig.8.12 *Program to alphabetize a customer list*

Calculation of Area Under a Curve

One of the applications of computers in numerical analysis is computing the area under a curve. One simple method of calculating the area under a curve is to divide the area into a number of trapezoids of same width and summing up the area of individual trapezoids. The area of a trapezoid is given by

$$Area = 0.5\ ^* (h1 + h2)\ ^* b$$

Where h1 and h2 are the heights of two sides and b is the width as shown in Fib 9.18.
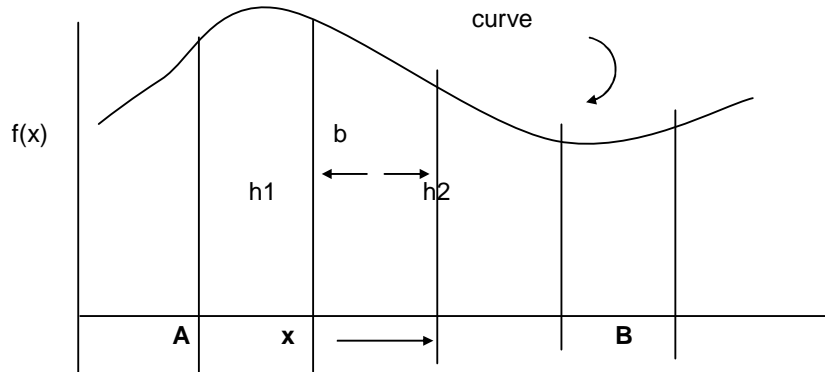
Fig. 9.18 Area under a curve

The program in Fig 9.20 calculates the area for a curve of the function

$$f(x) = x^2 + 1$$

between any two given limits, say, A and B.

Input

       Lower limit (A)
       Upper limit (B)
       Number of trapezoids

Output

       Total area under the curve between the given limits.

Algorithm

    1.   Input the lower and upper limits and the number of trapezoids.
    2.   Calculate the width of trapezoids.
    3.   Initialize the total area.
    4.   Calculate the area of trapezoid and add to the total area.
    5.   Repeat step-4 until all the trapezoids are completed.
    6.   Print total area.

The algorithm is implemented in top-down modular form as in Fig. 9.19.
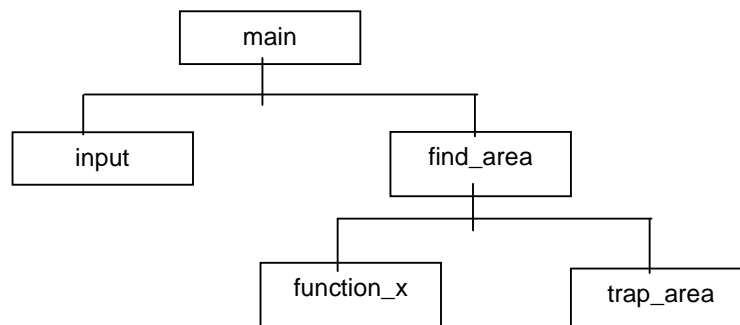
Fig 9.19   Modular Chart

The evaluation of f(x) has been done using a separate function so that it can be easily modified to allow other functions to be evaluated.The output for two runs shows that better accuracy is achieved with larger number of trapezoids. The actual area for the limits 0 and 3 is 12 units (by analytical method)

```
#include <stdio.h>

float start_point,                    /* GLOBAL VARIABLES */
      end_point,
      total_area;
int   numtraps;

main( )
{
   void   input(void);
   float  find_area(float a,float b,int n); /* prototype */
   print("AREA UNDER A CURVE");
   input( );
   total_area = find_area(start_point, end_point, numtraps);
   printf("TOTAL AREA = %f", total_area);
}
void input(void)
{
   printf("\n Enter lower limit:");
   scanf("%f", &start_point);
   printf("Enter upper limit:");
   scanf("%f", &end_point);
   printf("Enter number of trapezoids:");
   scanf("%d", &numtraps);
}
float find_area(float a, float b, int n)
{
   float base, lower, h1, h2; /* LOCAL VARIABLES */
   float function_x(float x); /* prototype */
   float trap_area(float h1,float h2,float base);/*prototype*/

   base = (b-1)/n;
   lower = a;

 for(lower =a; lower <= b-base; lower = lower + base)
{
    h1  =  function_x(lower);
    h1  =  function_x(lower + base);
    total_area += trap_area(h1, h2, base);
}
    return(total_area);

 float trap_area(float height_1,float height_2,float base)
{
   float area;     /* LOCAL VARIABLE */
   area = 0.5 * (height_1 + height_2) * base;
   return(area);
}
float function_x(float x)
{
    /* F(X) = X * X + 1 */

   return(x*x + 1);
}
```

Output

```
AREA UNDER A CURVE
Enter lower limit: 0
Enter upper limit:  3
Enter number of trapezoids: 30
TOTAL AREA   =   12.005000

AREA UNDER A CURVE
Enter lower limit: 0
Enter upper limit: 3
Enter number of trapezoids: 100
TOTAL AREA = 12.000438
```

Fig. 9.20 Computing area under a curve

<div align="center">

CASE STUDY

</div>

Book Shop Inventory

A book shop uses a personal computer to maintain the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher, stock position, etc. Whenever a customer wants a book, the shopkeeper inputs the title and author of the book and the system replies whether it is in the list or not. If it is not, an appropriate message is displayed. If book is in the list, then the system displays the book details and asks for number of copies. If the requested copies are available, the total cost of the books is displayed; otherwise the message "Required copies not in stock" is displayed.

A program to accomplish this is shown in Fig.10.8. The program uses a template to define the structure of the book. Note that the date of publication, a member of **record** structure, is also defined as a structure.

When the title and author of a book are specified, the program searches for the book in the list using the function

> **look_up(table, s1, s2, m)**

The parameter **table** which receives the structure variable **book** is declared as type **struct record.** The parameters **s1** and **s2** receive the string values of **title** and **author** while **m** receives the total number of books in the list. Total number of books is given by the expression

> **sizeof(book)/sizeof(struct record)**

The search ends when the book is found in the list and the function returns the serial number of the book. The function returns -1 when the book is not found. Remember that the serial number of the first book in the list is zero. The program terminates when we respond "NO" to the question

> Do you want any other book?

Note that we use the function

> **get(string)**

to get title, author, etc. from the terminal. This enables us to input strings with spaces such as "C Language". We cannot use **scanf** to read this string since it contains two words.

Since we are reading the quantity as a string using the **get(string)** function, we have to convert it to an integer before using it in any expressions. This is done using the **atoi()** function.

<div align="center">

BOOK SHOP INVENTORY
</div>

Programs

```c
#include   <stdio.h>
#include   <string.h>

struct  record
{
    char    author[20];
    char    title[30];
    float   price;

    struct
    {
        char    month[10];
        int     year;
    }
    date;
    char    publisher[10];
    int     quantity;
};
int look_up(struct record table[],char s1[],char s2[],int m);
void get (char string [ ] );

main()
{
    char title[30], author[20];
    int  index, no_of_records;
    char response[10], quantity[10];
```

```c
    struct record book[] =  {
    {"Ritche","C Language",45.00,"May",1977,"PHI",10},
    {"Kochan","Programming in C",75.50,"July",1983,"Hayden",5},
    {"Balagurusamy","BASIC",30.00,"January",1984,"TMH",0},
    {"Balagurusamy","COBOL",60.00,"December",1988,"Macmillan",25}
                                            };

    no_of_records = sizeof(book)/ sizeof(struct record);
    do
    {
      printf("Enter title and author name as per the list\n");

      printf("\nTitle:     ");
      get(title);

      printf("Author:    ");
      get(author);

      index = look_up(book, title, author, no_of_records);
      if(index != -1)     /*  Book found  */
      {
          printf("\n%s %s %.2f %s %d %s\n\n",
                     book[index].author,
                     book[index].title,
                     book[index].price,
                     book[index].date.month,
                     book[index].date.year,
                     book[index].publisher);

          printf("Enter number of copies:");
          get(quantity);
          if(atoi(quantity) < book[index].quantity)

            printf("Cost of %d copies = %.2f\n",atoi(quantity),
                 book[index].price * atoi(quantity));
          else
            printf("\nRequired copies not in stock\n\n");
      }
      else
          printf("\nBook not in list\n\n");

      printf("\nDo you want any other book? (YES / NO):");
      get(response);
    }
    while(response[0] == 'Y' || response[0] == 'y');
    printf("\n\nThank you.  Good bye!\n");
}
void get(char string [] )
{
    char  c;
    int  i = 0;
    do
    {
      c = getchar();
      string[i++] = c;
    }
    while(c != '\n');
    string[i-1] = '\0';
}


int look_up(struct record table[],char s1[],char s2[],int m)
```

```
    {
        int  i;
        for(i = 0; i < m;  i++)
            if(strcmp(s1, table[i].title) == 0  &&
               strcmp(s2, table[i].author) == 0)
               return(i);              /* book found       */
        return(-1);                     /* book not found   */
    }
```

Output

```
    Enter title and author name as per the list

    Title:    BASIC
    Author:   Balagurusamy
    Balagurusamy BASIC 30.00 January 1984 TMH

    Enter number of copies:5
    Required copies not in stock

    Do you want any other book? (YES / NO):y
    Enter title and author name as per the list

    Title:    COBOL
    Author:   Balagurusamy
    Balagurusamy COBOL 60.00 December 1988 Macmillan

    Enter number of copies:7
    Cost of 7 copies = 420.00

    Do you want any other book? (YES / NO):y
    Enter title and author name as per the list

    Title:    C Programming
    Author:   Ritche


    Book not in list


    Do you want any other book? (YES / NO):n


    Thank you.  Good bye!
```
_____

Fig.10.8 *Program of bookshop inventory*

1. Processing of Examination Marks

Marks obtained by a batch of students in the Annual Examination are tabulated as follows:

Student name	Marks obtained
S.Laxmi	45 67 38 55
V.S.Rao	77 89 56 69
--	- - - -

It is required to compute the total marks obtained by each student and print the rank list based on the total marks.

The program in Fig.11.14 stores the student names in the array **name** and the marks in the array **marks.** After computing the total marks obtained by all the students, the program prepares and prints the rank list. The declaration

```
int  marks[STUDENTS][SUBJECTS+1];
```

defines **marks** as a pointer to the array's first row. We use **rowptr** as the pointer to the row of **marks.** The **rowptr** is initialized as follows:

```
int (*rowptr)[SUBJECTS+1] = array;
```

Note that **array** is the formal argument whose values are replaced by the values of the actual argument **marks.** The parentheses around ***rowptr** makes the **rowptr** as a pointer to an array of **SUBJECTS+1** integers. Remember, the statement

```
int  *rowptr[SUBJECTS+1];
```

would declare **rowptr** as an array   of **SUBJECTS+1** elements.

When we increment the **rowptr** (by **rowptr+1**), the incrementing is done in units of the size of each row of **array,** making **rowptr** point to the next row. Since **rowptr** points to a particular row, **(*rowptr)[x]** points to the xth element in the row.

POINTERS AND TWO-DIMENSIONAL ARRAYS

Program

```
#define  STUDENTS  5
#define  SUBJECTS  4
#include <string.h>

main()
{
  char name[STUDENTS][20];
  int  marks[STUDENTS][SUBJECTS+1];

  printf("Input students names & their marks in four subjects\n");

  get_list(name, marks, STUDENTS, SUBJECTS);

  get_sum(marks, STUDENTS, SUBJECTS+1);

  printf("\n");
  print_list(name,marks,STUDENTS,SUBJECTS+1);

  get_rank_list(name, marks, STUDENTS, SUBJECTS+1);

  printf("\nRanked List\n\n");
  print_list(name,marks,STUDENTS,SUBJECTS+1);

  }
    /*    Input student name and marks        */
```

```c
 get_list(char *string[ ],
          int array [ ] [SUBJECTS +1], int m, int n)
{
    int   i, j, (*rowptr)[SUBJECTS+1] = array;

    for(i = 0; i < m; i++)
    {
       scanf("%s", string[i]);
       for(j = 0; j < SUBJECTS; j++)
          scanf("%d", &(*(rowptr + i))[j]);
    }
}
/*    Compute total marks obtained by each student    */
get_sum(int array [ ] [SUBJECTS +1], int m, int n)
{
    int   i, j, (*rowptr)[SUBJECTS+1] = array;

    for(i = 0; i < m; i++)
    {
       (*(rowptr + i))[n-1] = 0;
       for(j =0; j < n-1; j++)
          (*(rowptr + i))[n-1] += (*(rowptr + i))[j];
    }
}


/*    Prepare rank list based on total marks      */

get_rank_list(char *string [ ],
              int array [ ] [SUBJECTS + 1]
              int m,
              int n)
{
  int i, j, k, (*rowptr)[SUBJECTS+1] = array;
  char *temp;

  for(i = 1; i <= m-1; i++)
     for(j = 1; j <= m-i; j++)
        if( (*(rowptr + j-1))[n-1] < (*(rowptr + j))[n-1])
        {
         swap_string(string[j-1], string[j]);

         for(k = 0; k < n; k++)
         swap_int(&(*(rowptr + j-1))[k],&(*(rowptr+j))[k]);
          }
}
/*      Print out the ranked list             */
print_list(char *string[ ],
           int array [] [SUBJECTS + 1],
           int m,
           int n)
{
    int  i, j, (*rowptr)[SUBJECTS+1] = array;
    for(i = 0; i < m; i++)
    {
       printf("%-20s", string[i]);
       for(j = 0; j < n; j++)
          printf("%5d", (*(rowptr + i))[j]);
          printf("\n");
    }
```

```
    }

    /*      Exchange of integer values              */
    swap_int(int *p, int *q)
    {
        int  temp;

        temp = *p;
        *p   = *q;
        *q   = temp;
    }




    /*      Exchange of strings          */
    swap_string(char s1[ ], char s2[ ])
    {
        char  swaparea[256];
        int   i;

        for(i = 0; i < 256; i++)
           swaparea[i] = '\0';
        i = 0;
        while(s1[i] != '\0' && i < 256)
        {
           swaparea[i] = s1[i];
           i++;
        }
        i = 0;
        while(s2[i] != '\0' && i < 256)
        {
           s1[i] = s2[i];
           s1[++i] = '\0';
        }
        i = 0;
        while(swaparea[i] != '\0')
        {
           s2[i] = swaparea[i];
           s2[++i] = '\0';
        }
    }
```

Output

```
   Input students names & their marks in four subjects
   S.Laxmi 45 67 38 55
   V.S.Rao 77 89 56 69
   A.Gupta 66 78 98 45
   S.Mani 86 72 0 25
   R.Daniel 44 55 66 77

   S.Laxmi                  45   67   38   55   205
   V.S.Rao                  77   89   56   69   291
   A.Gupta                  66   78   98   45   287
   S.Mani                   86   72    0   25   183
   R.Daniel                 44   55   66   77   242

   Ranked List

   V.S.Rao                  77   89   56   69   291
   A.Gupta                  66   78   98   45   287
```

R.Daniel	44	55	66	77	242
S.Laxmi	45	67	38	55	205
S.Mani	86	72	0	25	183

Fig.11.14 *Preparation of the rank list of a class of students.*

2. Inventory Updating

The price and quantity of items stocked in a store changes every day. They may either increase or decrease. The program in Fig.11.15 reads the incremental values of price and quantity and computes the total value of the items in stock.

The program illustrates the use of structure pointers as function parameters. **&item,** the address of the structure **item,** is passed to the functions **update()** and **mul().** The formal arguments **product** and **stock,** which receive the value of **&item,** are declared as pointers of type **struct stores.**

```
                STRUCTURES AS FUNCTION PARAMETERS
                     Using structure pointers
```

Program

```
   struct stores
   {
        char   name[20];
        float price;
        int    quantity;
   };
   main()
   {
        void update(struct stores *, float, int);
        float           p_increment, value;
        int             q_increment;

        struct stores item = {"XYZ", 25.75, 12};
        struct stores *ptr = &item;

        printf("\nInput increment values:");
        printf(" price increment and quantity increment\n");
        scanf("%f %d", &p_increment, &q_increment);

   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
        update(&item, p_increment, q_increment);
   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
        printf("Updated values of item\n\n");
        printf("Name      : %s\n",ptr->name);
        printf("Price     : %f\n",ptr->price);
        printf("Quantity  : %d\n",ptr->quantity);

   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
        value  = mul(&item);
   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
        printf("\nValue of the item  =  %f\n", value);
   }

   void update(struct stores *product, float p, int q)
   {
        product->price += p;
        product->quantity += q;
   }

   float mul(struct stores *stock)
   {
        return(stock->price * stock->quantity);
```

```
    }
```

Output

```
Input increment values: price increment and quantity increment
10 12
Updated values of item

Name        : XYZ
Price       : 35.750000
Quantity    : 24

Value of the item  =  858.000000
```

Fig.11.15 *Use of structure pointers as function parameters.*

<div align="center">**CASE STUDIES**</div>

1. Insertion in a Sorted List

The task of inserting a value into the current location in a sorted linked list involves two operations:

1. Finding the node before which the new node has to be inserted.   We call this node as 'Key node'.

2. Creating a new node with the value to be inserted and inserting the new node by manipulating pointers appropriately.

In order to illustrate the process of insertion, we use a sorted linked list created by the create function discussed in Example 13.3. Figure 13.11 shows a complete program that creates a list (using sorted input data) and then inserts a given value into the correct place using function insert.

<div align="center">INSERTING A NUMBER IN A SORTED LIST</div>

Program

```
#include <stdio.h>
#include<stdio.h>
#define NULL 0

struct linked_list
{
   int number;
   struct linked-list *next;
};
typedef struct linked_lit node;

main()
{
   int n;
   node *head;
   void create(node *p);
   node *insert(node *p, int n);
   void print(node *p);

   head = (node *)malloc(sizeof(node));

   create(head);
   printf("\n");
   printf("Original list: ");
   print(head);
   printf("\n\n");
   printf("Input number to be inserted: ");
   scanf("%d", &n);

   head = inert(head,n);
   printf("\n");
   printf("New list:   ");
```

```c
   print(head);
}
void create(node *list)
{
   printf("Input a number \n");
   printf("(type -999 at end): ");
   scanf("%d", &list->number);

   if(list->number == -999)
   {
      list->next = NULL;
   }
   else    /* create next node */
   {
      list->next = (node *)malloc(sizeof(node));
      create(list->next);
   }
   return:
}

void print(node *list)
{
   if(list->next != NULL)
   {
      printf("%d -->", list->number);

   if(list ->next->next = = NULL)
      printf("%d", list->next->number);

   print(list->next);
   }
   return:
}

node *insert(node *head, int x)
{
   node *p1, *p2, *p;
   p1 = NULL;
   p2 = head;  /* p2 points to first node */

   for( ; p2->number < x; p2 = p2->next)
   {
     p1 = p2;

     if(p2->next->next == NULL)
         {
            p2 = p2->next;   /* insertion at end */
            break;
         }
     }

     /*key node found and insert new node */
```

```
        p = (node )malloc(sizeof(node));   / space for new node */

        p->number = x; /* place value in the new node */

        p->next = p2; /*link new node to key node */

        if (p1 == NULL)
              head = p; /* new node becomes the first node */
        else
              p1->next = p; /* new node inserted in middle */

        return (head);
}
```

Output

```
        Input a number
        (type -999 at end ); 10

        Input a number
        (type -999 at end ); 20

        Input a number
        (type -999 at end ); 30

        Input a number
        (type -999 at end ); 40

        Input a number
        (type -999 at end ); -999

        Original list:        10 -->20-->30-->40-->-999

        Input number to be inserted: 25
        New list: 10-->20-->25-->30-->40-->-999
```

Fig.13.11 Inserting a number in a sorted linked list

The function takes two arguments, one the value to be inserted and the other a pointer to the linked list. The function uses two pointers, **p1** and **p2** to search the list. Both the pointers are moved down the list with **p1** trailing **p2** by one node while the value **p2** points to is compared with the value to be inserted. The 'key node' is found when the number **p2** points to is greater (or equal) to the number to be inserted.

Once the key node is found, a new node containing the number is created and inserted between the nodes pointed to by **p1** and **p2**. The figures below illustrate the entire process.

key node

head

10   20   30   40

p1   p2

x = 25 (value to be inserted)
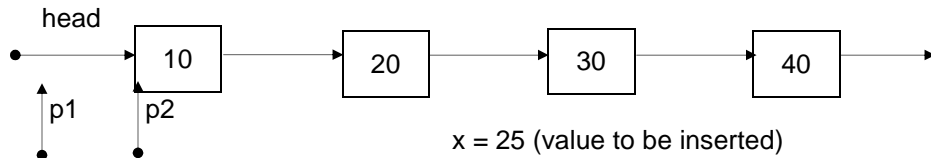
At the start of the search

key node

head

10   20   30   40

p1   p2

When key node is found

key node

head

10   20   30   40

p1   p2

p   25   new node

When new node is created

key node

head

10   20   30   40

p1   p2

25
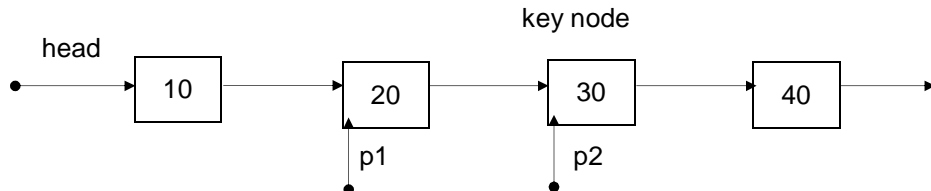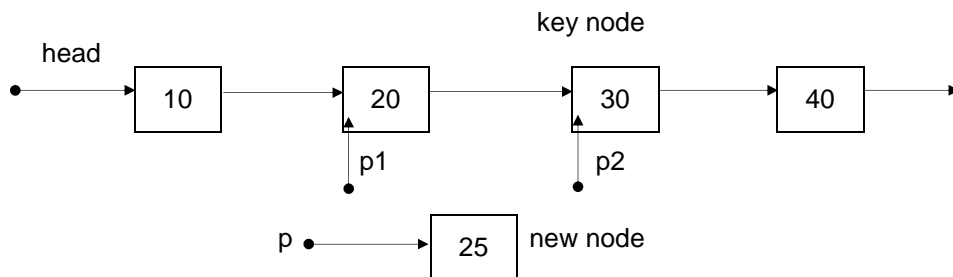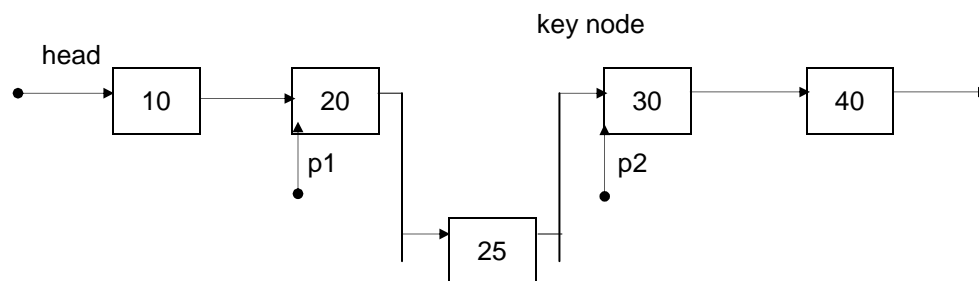
When new node is inserted

2. Building a Sorted List

The program in fig. 13.11 can be used to create a sorted list. This is possible by creating 'one item' list using the create function and then inserting the remaining items one after another using insert function.

A new program that would build a sorted list from a given list of numbers is shown in Fig. 13.12. The **main** function creates a 'base node' using the first number in the list and then calls the function **insert_sort** repeatedly to build the entire sorted list. It uses the same sorting algorithm discussed above but does not use any dummy node. Note that the last item points to NULL.

Program

```c
#include <stdio.h>
#include <stdlib.h>
#define NULL 0

struct linked_list
{
   int number;
   struct linked_list *next;
};
typedef struct linked_list node;

main ()
{
   int n;
   node *head = NULL;
   void print(node *p);
   node *insert_Sort(node *p, int n);

   printf("Input the list of numbers.\n");
   printf("At end, type -999.\n");
   scanf("%d",&n);

   while(n != -999)
   {
      if(head == NULL)       /* create 'base' node */
      {
           head = (node *)malloc(sizeof(node));
           head ->number = n;
           head->next = NULL;

      }

      else                    /* insert next item */
      {
           head = insert_sort(head,n);
      }
        scanf("%d", &n);
   }
   printf("\n");
   print(head);
   print("\n");
}
node *insert_sort(node *list, int x)
{
   node *p1, *p2, *p;
   p1 = NULL;
```

```c
   p2 = list; /* p2 points to first node */

   for( ; p2->number < x ; p2 = p2->next)
   {
      p1 = p2;

      if(p2->next == NULL)
      {
      p2 = p2->next;                         /* p2 set to NULL */
      break;                       /* insert new node at end */
   }
 }

/* key node found */
 p = (node *)malloc(sizeof(node)); /* space for new node */
 p->number = x;        /* place value in the new node */
 p->next = p2;         /* link new node to key node */

if (p1 == NULL)
   list = p;      /* new node becomes the first node */
 else
   p1->next = p;  /* new node inserted after 1st node */
 return (list);
}
void print(node *list)
{
   if (list == NULL)
      printf("NULL");
   else
   {
      printf("%d-->",list->number);
      print(list->next;
   }
   return;
}
```

Output

```
     Input the list of number.
     At end, type – 999.
     80 70 50 40 60 –999

     40-->50-->60-->70-->80 -->NULL

     Input the list of number.
     At end, type –999.
     40 70 50 60 80 –999

     40-->50-->60-->70-->80-->NULL
```

Fig.13.12 *Creation of sorted list from a given list of numbers*

Program

```
          main( )
          {
          /*…………printing begins………………*/

                printf("I see, I remember");

          /*………………printing ends…………………*/
          }
```

Fig 1.2 A program to print one line of text

Addition of Two Numbers

Program

```
/* Programm ADDITION                                   line-1  */
/* Written by EBG                                      line-2  */
main()                                          /* line-3  */
{                                               /* line-4  */
        int    number;                          /* line-5  */
        float amount;                           /* line-6  */
                                                /* line-7  */
        number = 100;                           /* line-8  */
                                                /* line-9  */
        amount = 30.75 + 75.35;                 /* line-10 */
        printf("%d\n",number);                  /* line-11 */
        printf("%5.2f",amount);                 /* line-12 */
}                                               /* line-13 */
```

Fig.1.4 Program to add two numbers

Program

```
     /*-------------------- INVESTMENT PROBLEM --------------------*/
#define PERIOD    10
#define PRINCIPAL 5000.00
     /*------------------ MAIN PROGRAM BEGINS --------------------*/
main()
{ /*------------------ DECLARATION STATEMENTS ----------------*/
     int year;
     float amount, value, inrate;
/*------------------ ASSIGNMENT STATEMENTS ------------------*/
     amount  =  PRINCIPAL;
     inrate  =  0.11;
     year  = 0;
/*----------------- COMPUTATION STATEMENTS ------------------*/
/*-------------- COMPUTATION USING While LOOP ---------------*/
     while(year <= PERIOD)
     {      printf("%2d          %8.2f\n",year, amount);
            value =   amount + inrate * amount;
            year       =      year + 1;
            amount     =      value;
     }
/*--------------------- while LOOP ENDS ---------------------*/
}
/*---------------------- PROGRAM ENDS ----------------------*/
```

Program

```
/*----------------- PROGRAM USING FUNCTION -----------------*/

int mul (int a, int b);       /*------- DECLARATION ------------*/

      /*------------------- MAIN PROGRAM BEGINS -------------------*/
     main ()
     {
          int a, b, c;

          a = 5;
          b = 10;
          c = mul (a,b);

          printf ("multiplication of %d and %d is %d",a,b,c);
     }

/* ---------------      MAIN PROGRAM ENDS
                        MUL() FUNCTION STARTS -----------------*/

     int mul (int x, int y)
     int p;

     p = x*y;
     {
          return(p);
     }
/* ------------------- MUL () FUNCTION ENDS ------------------*/
```

Fig.1.7 A Program using a user-defined function

Program

```
      /*--------------- PROGRAM USING COSINE FUNCTION -------------- */
 #include <math.h>
 #define   PI      3.1416
 #define   MAX     180

 main ( )
 {

     int angle;
     float x,y;

     angle = 0;
     printf("        Angle      Cos(angle)\n\n");

     while(angle  <= MAX)
     {
          x = (PI/MAX)*angle;
          y = cos(x);
          printf("%15d %13.4f\n", angle, y);
          angle = angle + 10;
     }
 }
```

The program in Fig.2.9 illustrates the use of integer constants on a 16-bit machine. The output in figure 2.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```
                    INTEGER NUMBERS ON 16-BIT MACHINE
```

Program
```
     main()
     {
         printf("Integer values\n\n");
         printf("%d %d %d\n", 32767,32767+1,32767+10);
         printf("\n");
         printf("Long integer values\n\n");
         printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
     }
```

                                                                  Output
```
         Integer values

         32767 -32768 -32759

         Long integer values

         32767 32768 32777
```

Fig. 2.3 Representation of integer constants

The variables **x** and **p** have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to **x** is displayed under different output formats. The value of x is displayed as 1.234567880630 under %.12lf format, while the actual value assigned is 1.234567890000. This is because the variable **x** has been declared as a **float** that can store values only upto six decimal places.

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the **printf** function will always display a **float** or **double** value to six decimal places. We will discuss later the output formats for displaying numbers.

```
                          EXAMPLES OF ASSIGNMENTS
```

Program

```
   main()
    {
    /*..........DECLARATIONS...........................*/
```

```
        float      x, p ;
        double     y, q ;
        unsigned   k ;

    /*..........DECLARATIONS AND ASSIGNMENTS............*/

        int        m = 54321 ;
        long int   n = 1234567890 ;

    /*..........ASSIGNMENTS............................*/

        x = 1.234567890000 ;
        y = 9.87654321 ;
        k = 54321 ;
        p = q = 1.0 ;

    /*..........PRINTING...............................*/

        printf("m = %d\n", m) ;
        printf("n = %ld\n", n) ;
        printf("x = %.12lf\n", x) ;
        printf("x = %f\n", x) ;
        printf("y = %.12lf\n",y) ;
        printf("y = %lf\n", y) ;
        printf("k = %u  p = %f  q = %.12lf\n", k, p, q) ;
    }
```

Output

```
    m = -11215
    n = 1234567890
    x = 1.234567880630
    x = 1.234568
    y = 9.876543210000
    y = 9.876543
    k = 54321  p = 1.000000  q = 1.000000000000
```

Fig. 2.8 Examples of assignments

Example 2.3

The program in Fig.2.9 illustrates the use of **scanf** funtion.

The first executable statement in the program is a **printf,** requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

        Enter an integer number

As soon as the user types in an integer number, the computer proceeds to compare the value with 100. If the value typed in is less than 100, then a message

        Your number is smaller than 100

is printed on the screen. Otherwise, the message

        Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in Fig.2.9.

Program

```
main()
{
    int  number;

    printf("Enter an integer number\n");
    scanf ("%d", &number);

    if ( number < 100 )
      printf("Your number is smaller than 100\n\n");
    else
      printf("Your number contains more than two digits\n");
}
```

Output

```
Enter an integer number
54
Your number is smaller than 100

Enter an integer number
108
Your number contains more than two digits
```

*Fig.2.9 Use of **scanf** function*

Example 2.4

Sample Program 3 discussed in Chapter 1 can be converted into a more flexible interactive program using **scanf** as shown in Fig.2.10.

In this case, computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing a prompt message

        Input amount, interest rate, and period

and then waits for input values. As soon as we finish entering

INTERACTIVE INVESTMENT PROGRAM

Program

```
main()
{
    int    year, period ;
    float  amount, inrate, value ;

    printf("Input amount, interest rate, and period\n\n") ;
    scanf ("%f %f %d", &amount, &inrate, &period) ;
    printf("\n") ;
    year = 1 ;

    while( year <= period )
```

```
        {
                value = amount + inrate * amount ;
                printf("%2d  Rs %8.2f\n", year, value) ;
                amount = value ;
                year = year + 1 ;
        }
}
```

Output

```
Input amount, interest rate, and period

10000  0.14  5

 1  Rs 11400.00
 2  Rs 12996.00
 3  Rs 14815.44
 4  Rs 16889.60
 5  Rs 19254.15

Input amount, interest rate, and period

20000  0.12  7

 1  Rs 22400.00
 2  Rs 25088.00
 3  Rs 28098.56
 4  Rs 31470.39
 5  Rs 35246.84
 6  Rs 39476.46
 7  Rs 44213.63
```

Fig.2.10 Interactive investment program

PROGRAM TO CONVERT DAYS TO MONTHS AND DAYS

Program

```
main ()
{
    int   months, days ;

    printf("Enter days\n") ;
    scanf("%d", &days) ;

    months = days / 30 ;
    days   = days % 30 ;

    printf("Months = %d   Days = %d", months, days) ;
}
```

Output

```
Enter days
265
Months = 8   Days = 25

Enter days
364
Months = 12   Days = 4

Enter days
45
Months = 1   Days = 15
```
_____

Fig. 3.1 Illustration of integer arithmetic

USE OF SHORTHAND OPERATORS

Program

```
#define   N   100
#define   A   2

main()
```

```
    {
        int  a;

        a  =  A;
        while( a < N )
        {
            printf("%d\n", a);
            a *= a;
        }
    }
```

Output

```
    2
    4
    16
```

*Fig. 3.2 Use of shorthand operator *=*

Example 3.3

In Fig.3.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator **++** works when used in an expression. In the statement

    c = ++a - b;

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

    d = b++ + a;

the old value of **b** (=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

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

The program also illustrates that the expression

    c > d ? 1 : 0

assumes the value 0 when c is less than d and 1 when c is greater than d.

ILLUSTRATION OF OPERATORS

Program

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

       a = 15;
       b = 10;
       c = ++a - b;

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

       d = b++ +a;
```

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

        printf("a/b = %d\n", a/b);
        printf("a%%b = %d\n", a%b);
        printf("a *= b = %d\n", a*=b);
        printf("%d\n", (c>d) ? 1 : 0);                    printf("%d\n",
        (c<d) ? 1 : 0);
    }
```

Output

```
a = 16   b = 10   c = 6
a = 16   b = 11   d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

Fig. 3.3 Further illustration of arithmetic operators

Example 3.4

The program in Fig.3.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

EVALUATION OF EXPRESSIONS

Program

```
main()
{
    float  a, b, c, x, y, z;

    a = 9;
    b = 12;
    c = 3;

    x = a - b / 3 + c * 2 - 1;
    y = a - b / (3 + c) * (2 - 1);
    z = a -(b / (3 + c) * 2) - 1;

    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);
}
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig.3.4 Illustrations of evaluation of expressions

PROGRAM SHOWING ROUND-OFF ERRORS

Program

```
/*------------------ Sum of n terms of 1/n -------------------*/
  main()
  {
      float  sum, n, term ;
      int    count = 1 ;

      sum = 0 ;
      printf("Enter value of n\n") ;
       scanf("%f", &n) ;
      term = 1.0/n ;
      while( count <= n )
      {
          sum = sum + term ;
          count++ ;
      }
      printf("Sum = %f\n", sum) ;
  }
```

Output

```
  Enter value of n
  99
  Sum = 1.000001
  Enter value of n
  143
  Sum = 0.999999
```

Fig.3.6 Round-off errors in floating point computations

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation, the result is not always 1.

PROGRAM SHOWING THE USE OF A CAST

Program

```
main()
{
    float  sum ;
    int    n ;

    sum = 0 ;

    for( n = 1 ; n <= 10 ; ++n )
    {
       sum = sum + 1/(float)n ;
       printf("%2d  %6.4f\n", n, sum) ;
    }
}
```

Output

```
 1  1.0000
 2  1.5000
 3  1.8333
 4  2.0833
 5  2.2833
 6  2.4500
 7  2.5929
 8  2.7179
 9  2.8290
10  2.9290
```

Fig. 3.8 Use of a cast

*READING A **CHARACTER** FROM KEYBOARD*

Program

```
#include   <stdio.h>

main()
{
   char answer;

   printf("Would you like to know my name?\n");
    printf("Type Y for YES and N for NO:  ");

   answer = getchar();  /* .... Reading a character...*/

   if(answer == 'Y' || answer == 'y')
     printf("\n\nMy name is BUSY BEE\n");
   else
     printf("\n\nYou are good for nothing\n");
}
```

Output

```
Would you like to know my name?
Type Y for YES and N for NO:  Y

My name is BUSY BEE

Would you like to know my name?
Type Y for YES and N for NO:  n

You are good for nothing
```

Fig.4.1 *Use of **getchar** function*

message accordingly. These tests are done with the help of following functions:

isalpha(character)
isdigit(character)

For example, **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

```
                          TESTING CHARACTER TYPE
```

Program:

```
#include  <stdio.h>
#include  <ctype.h>

main()
{
    char character;
    printf("Press any key\n");

    character = getchar();

    if (isalpha(character) > 0)
       printf("The character is a letter.");

    else
       if (isdigit (character) > 0)
          printf("The character is a digit.");

       else
          printf("The character is not alphanumeric.");
}
```

Output

```
Press any key
h
The character is a letter.

Press any key
5
The character is a digit.

Press any key
*
The character is not alphanumeric.
```

Fig.4.2 *Program to test the **character** type*

Example 4.3

A program that reads a character from keyboard and then prints it in reverse case is given in Fig.4.3. That is, if the input is upper case, the output will be lower case and vice versa.

The program uses three new functions: **islower, toupper**, and **tolower**. The function **islower** is a conditional function and takes the value TRUE if the argument is a lower case alphabet; otherwise takes the value FALSE. The function **toupper** converts the lower case argument into an upper case alphabet while the function **tolower** does the reverse.

```
             WRITING A CHARACTER TO THE SCREEN
```

Program

```
#include <stdio.h>
```

```
#include <ctype.h>

main()
{
    char alphabet;

    printf("Enter an alphabet");
    putchar('\n');    /* move to next line */
    alphabet = getchar();

    if (islower(alphabet))
        putchar(toupper(alphabet));
    else
        putchar(tolower(alphabet));
}
```

Output

```
        Enter an alphabet
        a
        A
        Enter an alphabet
        Q
        q
        Enter an alphabet
        z
        Z
```

Fig.4.3 Reading and writing of alphabets in reverse case

Example 4.4

Various input formatting options for reading integers are experimented in the program shown in Fig. 4.4.

The first **scanf** requests input data for three integer values **a, b,** and **c,** and accordingly three values 1, 2, and 3 are keyed in. Because of the specification %*d the value 2 has been skipped and 3 is assigned to the variable **b**. Notice that since no data is available for c, it contains garbage.

The second **scanf** specifies the format %2d and %4d for the variables **x** and **y** respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits that the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second **scanf** has truncated the four digit number 6789 and assigned 67 to **x** and 89 to **y**. The value 4321 has been assigned to the first variable in the immediately following **scanf** statement.

Program:

```
 main()
 {
     int a,b,c,x,y,z;
     int p,q,r;

     printf("Enter three integer numbers\n");
     scanf("%d %*d %d",&a,&b,&c);
     printf("%d %d %d \n\n",a,b,c);

     printf("Enter two 4-digit numbers\n");
     scanf("%2d %4d",&x,&y);
     printf("%d %d\n\n", x,y);

     printf("Enter two integers\n");
     scanf("%d %d", &a,&x);
     printf("%d %d \n\n",a,x);

     printf("Enter a nine digit number\n");
     scanf("%3d %4d %3d",&p,&q,&r);
     printf("%d %d %d \n\n",p,q,r);

     printf("Enter two three digit numbers\n");
     scanf("%d %d",&x,&y);
     printf("%d %d",x,y);
 }
```

Output

```
 Enter three integer numbers
 1 2 3
 1 3 -3577

 Enter two 4-digit numbers
 6789  4321
 67 89

 Enter two integers
 44  66
 4321 44

 Enter a nine-digit number
 123456789
 66 1234 567

 Enter two three-digit numbers
 123  456
 89 123
```

Fig.4.4 *Reading integers using* **scanf**

Example 4.5

Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig.4.5.

Program:
```
   main()
   {
       float x,y;
       double p,q;

       printf("Values of x and y:");
       scanf("%f %e", &x, &y);
       printf("\n");
       printf("x = %f\ny = %f\n\n", x, y);

       printf("Values of p and q:");
       scanf("%lf %lf", &p, &q);
       printf("\np = %lf\nq = %e",p,q);
       printf("\n\np = %.12lf\np = %.12e", p,q);
   }
```

Output

```
   Values of x and y:12.3456  17.5e-2

   x = 12.345600
   y = 0.175000

   Values of p and q:4.142857142857  18.5678901234567890

   p = 4.142857142857
   q = 1.856789012346e+001
```

Fig.4.5 *Reading of **real** numbers*

Example 4.6

Reading of strings using **%wc** and **%ws** is illustrated in Fig.4.6.

The program in Fig.4.6 illustrates the use of various field specifications for reading strings. When we use **%wc** for reading a string, the system will wait until the wth character is keyed in.

Note that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

Program

```
   main()
   {
       int no;
       char name1[15], name2[15], name3[15];

       printf("Enter serial number and name one\n");
       scanf("%d %15c", &no, name1);
       printf("%d %15s\n\n", no, name1);

       printf("Enter serial number and name two\n");
       scanf("%d %s", &no, name2);
       printf("%d %15s\n\n", no, name2);
```

```
        printf("Enter serial number and name three\n");
        scanf("%d %15s", &no, name3);
        printf("%d %15s\n\n", no, name3);
    }
```

Output

```
    Enter serial number and name one
    1 123456789012345
    1 123456789012345r

    Enter serial number and name two
    2 New York
    2             New

    Enter serial number and name three
    2             York

    Enter serial number and name one
    1 123456789012
    1 123456789012   r

    Enter serial number and name two
    2 New-York
    2 New-York
    Enter serial number and name three
    3 London
    3 London
```

Fig. 4.6 Reading of strings

Example 4.7
The program in Fig. 4.7 illustrates the function of %[ ] specification.

                        ILLUSTRATION OF %[ ] SPECIFICATION

Program-A

```
    main()
    {
        char address[80];

        printf("Enter address\n");
        scanf("%[a-z']", address);
        printf("%-80s\n\n", address);
    }
```

Output

```
    Enter address
    new delhi 110002
    new delhi
```

                        ILLUSTRATION OF %[^] SPECIFICATION

Program-B
```
    main()
    {
        char address[80];
```

```
    printf("Enter address\n");
    scanf("%[^\n]", address);
    printf("%-80s", address);
}
```

Output

```
Enter address
New Delhi 110 002
New Delhi 110 002
```

Fig.4.7 Illustration of conversion specification%[] for strings

---

Example 4.8

The program presented in Fig.4.8 illustrates the testing for correctness of reading of data by **scanf** function.

The function **scanf** is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an **int** variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.
Note that the character `2' is assigned to the character variable c.

TESTING FOR CORRECTNESS OF INPUT DATA

Program

```
main()
{
    int a;
    float b;
    char c;

    printf("Enter values of a, b and c\n");

    if (scanf("%d %f %c", &a, &b, &c) == 3)
       printf("a = %d  b = %f   c = %c\n" , a, b, c);
    else
       printf("Error in input.\n");
}
```

Output
```
          Enter values of a, b and c
          12   3.45  A
          a = 12  b = 3.450000   c = A

          Enter values of a, b and c
          23  78  9
          a = 23  b = 78.000000   c = 9

          Enter values of a, b and c
          8  A  5.25

          Error in input.
          Enter values of a, b and c
          Y  12  67

          Error in input.
```

```
            Enter values of a, b and c
            15.75   23   X

            a = 15   b = 0.750000    c = 2
```

Fig.4.8 *Detection of errors in **scanf** input*

Example 4.9

The program in Fig.4.9 illustrates the output of integer numbers under various formats.

PRINTING OF **INTEGER** NUMBERS

Program:
```
  main()
  {
      int m = 12345;
      long n = 987654;

      printf("%d\n",m);
      printf("%10d\n",m);
      printf("%010d\n",m);
      printf("%-10d\n",m);
      printf("%10ld\n",n);
      printf("%10ld\n",-n);
  }
```

Output

```
  12345
       12345
  0000012345
  12345
      987654
     -987654
```

Fig.4.9 *Formatted output of integers*

Example 4.10
All the options of printing a real number are illustrated in Fig.4.10.

PRINTING OF **REAL NUMBERS**

Program:
```
  main()
  {
      float y = 98.7654;

      printf("%7.4f\n", y);
      printf("%f\n", y);
      printf("%7.2f\n", y);
      printf("%-7.2f\n", y);
      printf("%07.2f\n", y);
      printf("%*.*f", 7, 2, y);
      printf("\n");
      printf("%10.2e\n", y);
```

```
        printf("%12.4e\n", -y);
        printf("%-10.2e\n", y);
        printf("%e\n", y);
    }
```

Output   98.7654
            98.765404
            98.77
            98.77
            0098.77
            98.77
            9.88e+001
            -9.8765e+001
            9.88e+001
            9.876540e+001

Fig.4.10 *Formatted output of **real** numbers*

Example 4.11

Printing of characters and strings is illustrated in Fig.4.11.

Program

```
main()
{
    char x = 'A';
    static char  name[20] = "ANIL KUMAR GUPTA";

    printf("OUTPUT OF CHARACTERS\n\n");
    printf("%c\n%3c\n%5c\n", x,x,x);
    printf("%3c\n%c\n", x,x);
    printf("\n");

    printf("OUTPUT OF STRINGS\n\n");
    printf("%s\n", name);
    printf("%20s\n", name);
    printf("%20.10s\n", name);
    printf("%.5s\n", name);
    printf("%-20.10s\n", name);
    printf("%5s\n", name);
}
```

Output

```
    OUTPUT OF CHARACTERS

     A
       A
         A
       A
     A

    OUTPUT OF STRINGS

    ANIL KUMAR GUPTA
        ANIL KUMAR GUPTA
```

```
           ANIL KUMAR
ANIL
ANIL KUMAR
ANIL KUMAR GUPTA
```

Fig.4.11 *Printing of characters and strings*

ILLUSTRATION OF *if* STATEMENT

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*

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 kgs\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 kgs
and height > 170 cm = 3
```

Fig. 5.4 *Use of* **if** *for counting*

Example 5.3

A program to evaluate the power series

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

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

The power series contains the recurrence relationship of the type

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

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

$$T_0 = 1$$

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

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

<div align="center">EXPERIMENT WITH if...else STATEMENT</div>

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

Example 5.4

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

SELECTING THE LARGEST OF THREE VALUES

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

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.

USE OF **else if** LADDER

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

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*

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

```c
#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*

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.

<div align="center">EXAMPLE OF while STATEMENT</div>

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*

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.

<div align="center">

PRINTING OF *MULTIPLICATION* ***TABLE***

</div>

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
   --------------------------------------------------------
```

<div align="center">

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

</div>

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a ***long int*** and **q** as a **double**.

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)
```

Program:

```
main()
{
    long int p;
    int     n;
    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*

Example 6.4

A class of **n** students takes 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 totaled 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***

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.

<div align="center">USE OF break IN A PROGRAM</div>

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*

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.

EXAMPLE OF **exit** WITH **goto** STATEMENT

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*

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.

USE OF **continue** STATEMENT

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*

Example 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 SHOWING **ONE-DIMENSIONAL ARRAY**

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***

Example 7.2

Given below is the list of marks obtained by a class of 50 students in an annual examination.

        43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37
         40 49 16 75 87 91 33 24 58 78 65 56 76 67 45 54 36 63 12 21
         73 49 51 19 39 49 68 93 85 59

Write a program to count the number of students belonging to each of following groups of marks: 0-9, 10-19, 20-29,.....,100.

The program coded in Fig.7.2 uses the array **group** containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

PROGRAM FOR **FREQUENCY COUNTING**

Program
```
  #define    MAXVAL    50
  #define    COUNTER   11

  main()
  {
      float        value[MAXVAL];
      int          i, low, high;
      int   group[COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};

      /* . . . . . . . .READING AND COUNTING . . . . . .*/
      for( i = 0 ; i < MAXVAL ; i++ )
      {

       /*. . . . . . . .READING OF VALUES . . . . . . . . */
         scanf("%f", &value[i]) ;

       /*. . . . . .COUNTING FREQUENCY OF GROUPS. . . . . */
         ++ group[ (int) ( value[i] + 0.5 ) / 10] ;
      }

       /* . . . .PRINTING OF FREQUENCY TABLE . . . . . . .*/
      printf("\n");
      printf(" GROUP     RANGE      FREQUENCY\n\n") ;
      for( i = 0 ; i < COUNTER ; i++ )
      {
         low  = i * 10 ;
         if(i == 10)
           high = 100 ;
         else
           high = low + 9 ;
         printf("  %2d    %3d to %3d      %d\n",
                i+1, low, high, group[i] ) ;
      }
  }
```

```
Output
      43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74
      81 49 37 40 49 16 75 87 91 33 24 58 78 65 56 76 67      (Input data)
      45 54 36 63 12 21 73 49 51 19 39 49 68 93 85 59
```

```
      GROUP                RANGE              FREQUENCY
       1             0    to    9                 2
       2            10    to   19                 4
       3            20    to   29                 4
       4            30    to   39                 5
       5            40    to   49                 8
       6            50    to   59                 8
       7            60    to   69                 7
       8            70    to   79                 6
       9            80    to   89                 4
      10            90    to   99                 2
      11           100 to 100                     0
```

Fig.7.2 *Program for **frequency counting***

Example 7.3

Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:

    (a) Total value of sales by each girl.
    (b) Total value of each item sold.
    (c) Grand total of sales of all items by all girls.

The program and its output are shown in Fig.7.4. The program uses the variable **value** in two-dimensions with the index i representing girls and j representing items. The following equations are used in computing the results:

(a)   Total sales by m^{th} girl = $\sum\limits_{j=0}^{2}$ value [m][j]
      (girl_total[m])

(b)   Total value of nth item = $\sum\limits_{i=0}^{3}$ value [i][n]
      (item_total[n])

(c)   Grand total   = $\sum\limits_{i=0}^{3}\sum\limits_{j=0}^{2}$ value[i][j]

                   = $\sum\limits_{i=0}^{3}$ girl_total[i]

                   = $\sum\limits_{j=0}^{2}$ item_total[j]

Program:

```c
#define    MAXGIRLS    4
#define    MAXITEMS    3

main()
{
    int    value[MAXGIRLS][MAXITEMS];
    int    girl_total[MAXGIRLS] , item_total[MAXITEMS];
    int    i, j, grand_total;

/*.......READING OF VALUES AND COMPUTING girl_total ...*/

    printf("Input data\n");
    printf("Enter values, one at a time, row-wise\n\n");

    for( i = 0 ; i < MAXGIRLS ; i++ )
    {
       girl_total[i] = 0;
       for( j = 0 ; j < MAXITEMS ; j++ )
       {
           scanf("%d", &value[i][j]);
           girl_total[i] = girl_total[i] + value[i][j];
       }
    }
/*.......COMPUTING item_total.........................*/

    for( j = 0 ; j < MAXITEMS ; j++ )
    {
       item_total[j] = 0;
       for( i =0 ; i < MAXGIRLS ; i++ )
           item_total[j] = item_total[j] + value[i][j];
    }


/*.......COMPUTING grand_total.........................*/

    grand_total = 0;
    for( i =0 ; i < MAXGIRLS ; i++ )
       grand_total = grand_total + girl_total[i];

/* .......PRINTING OF RESULTS.........................*/

    printf("\n GIRLS TOTALS\n\n");
    for( i = 0 ; i < MAXGIRLS ; i++ )
       printf("Salesgirl[%d] = %d\n", i+1, girl_total[i] );
    printf("\n ITEM  TOTALS\n\n");
    for( j = 0 ; j < MAXITEMS ; j++ )
       printf("Item[%d] = %d\n", j+1 , item_total[j] );
    printf("\nGrand Total = %d\n", grand_total);
}
```

Output

```
Input data
Enter values, one at a time, row_wise

310   257   365
210   190   325
```

```
405  235  240
260  300  380

GIRLS TOTALS

Salesgirl[1] = 950
Salesgirl[2] = 725
Salesgirl[3] = 880
Salesgirl[4] = 940

 ITEM  TOTALS

Item[1] = 1185
Item[2] = 1000
Item[3] = 1310

Grand Total = 3495
```

Fig.7.4 *Illustration of two-dimensional arrays.*

Example 7.4
Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below:

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	.	.	.
4	4	8	.	.	.
5	5	10	.	.	25

The program shown in Fig.7.5 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

product(i,j) = row * column

where i denotes rows and j denotes columns of the product table. Since the indices i and j ranges from 0 to 4, we have introduced the following transformation:

row = i+1
column = j+1

```
                PROGRAM TO PRINT MULTIPLICATION TABLE
```

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

Example 7.5

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:

    M - Madras   1 - Ambassador
    D – Delhi   2 - Fiat
    C – Calcutta   3 - Dolphin
    B – Bombay   4 - Maruti

Write a program to produce a table showing popularity of various cars in four cities.

A two-dimensional array **frequency** is used as an accumulator to store the number of cars used, under various categories in each city. For example, the element **frequency** [i][j] denotes the number of cars of type j used in city i. The **frequency** is declared as an array of size 5x5 and all the elements are initialized to zero.

The program shown in fig.7.6 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

Program

```
main()
{
    int  i, j, car;
    int  frequency[5][5] = { {0},{0},{0},{0},{0} };
    char city;

    printf("For each person, enter the city code \n");
    printf("followed by the car code.\n");
    printf("Enter the letter X to indicate end.\n");

/*. . . . . . TABULATION BEGINS  . . . . . */

    for( i = 1 ; i < 100 ; i++ )
    {
       scanf("%c", &city );
       if( city == 'X' )
         break;

       scanf("%d", &car );

       switch(city)
       {

            case 'B' :  frequency[1][car]++;
                        break;
            case 'C' :  frequency[2][car]++;
                        break;
            case 'D' :  frequency[3][car]++;
                        break;
            case 'M' :  frequency[4][car]++;
                        break;
       }
    }
/*. . . . . .TABULATION COMPLETED AND PRINTING BEGINS. . . .*/
    printf("\n\n");
    printf("               POPULARITY  TABLE\n\n");
    printf("-------------------------------------------\n");
    printf("City     Ambassador  Fiat  Dolphin  Maruti \n");
    printf("-------------------------------------------\n");
    for( i = 1 ; i <= 4 ; i++ )
    {
       switch(i)
       {
            case 1 :  printf("Bombay    ") ; break ;
            case 2 :  printf("Calcutta  ") ; break ;
            case 3 :  printf("Delhi     ") ; break ;
            case 4 :  printf("Madras    ") ; break ;
       }
       for( j = 1 ; j <= 4 ; j++ )
          printf("%7d", frequency[i][j] ) ;
       printf("\n") ;
    }
    printf("-------------------------------------------\n");
 /*. . . . . . . . . . PRINTING ENDS. . . . . . . . . . .*/

}
```

Output

```
For each person, enter the city code
followed by the car code.
Enter the letter X to indicate end.

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  X
```

POPULARITY TABLE

City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	1	0
Delhi	2	1	3	2
Madras	4	1	1	4

Fig.7.6 Program to tabulate a survey data

READING A SERIES OF WORDS USING **scanf** FUNCTION

Program

```
main( )
{
    char word1[40], word2[40], word3[40], word4[40];

    printf("Enter text : \n");
    scanf("%s %s", word1, word2);
    scanf("%s", word3);
    scanf("%s", word4);

    printf("\n");
    printf("word1 = %s\nword2 = %s\n", word1, word2);
    printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

Output

```
Enter text :
Oxford Road, London M17ED

word1 = Oxford
word2 = Road,
word3 = London
word4 = M17ED

Enter text :
Oxford-Road, London-M17ED United Kingdom
word1 = Oxford-Road
word2 = London-M17ED
word3 = United
word4 = Kingdom
```

Fig.8.1 Reading a series of words using ***scanf***

The program shown in Fig.8.2 can read a line of text (upto a maximum of 80 characters) into the string **line** using **getchar** function. Every time a character is read, it is assigned to its location in the string **line** and then tested for *newline* character. When the *newline* character is read (signalling the end of line), the reading loop is terminated and the *newline* character is replaced by the null character to indicate the end of character string.

When the loop is exited, the value of the index **c** is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value **c-1** gives the position where the *null* character is to be stored.

Program

```
#include  <stdio.h>

main( )
{
    char  line[81], character;
    int   c;

    c = 0;

    printf("Enter text. Press <Return> at end\n");
    do
    {
        character = getchar();
        line[c]   = character;
        c++;
    }
    while(character != '\n');

    c = c - 1;
    line[c] = '\0';
    printf("\n%s\n", line);
}
```

Output

```
Enter text. Press <Return> at end
Programming in C is interesting.

Programming in C is interesting.

Enter text. Press <Return> at end
National Centre for Expert Systems, Hyderabad.

National Centre for Expert Systems, Hyderabad.
```

*Fig.8.2 Program to **read a line of text** from terminal*

Example 8.3

Write a program to copy one string into another and count the number of characters copied.

The program is shown in Fig.8.3. We use a **for** loop to copy the characters contained inside **string2** into the **string1**. The loop is terminated when the *null* character is reached. Note that we are again assigning a null character to the **string1.**

COPYING ONE **STRING** INTO **ANOTHER**

Program

```
main( )
{
    char  string1[80], string2[80];
    int   i;

    printf("Enter a string \n");
    printf("?");

    scanf("%s", string2);

    for( i=0 ; string2[i] != '\0'; i++)
        string1[i] = string2[i];
```

```
        string1[i] = '\0';

        printf("\n");
        printf("%s\n", string1);
        printf("Number of characters = %d\n", i );
    }
```

Output

```
    Enter a string
    ?Manchester

    Manchester
    Number of characters = 10

    Enter a string
    ?Westminster

    Westminster
    Number of characters = 11
```

Fig.8.3 *Copying one string into another*

Example 8.4

Write a program to store the string "United Kingdom" in the array **country** and display the string under various format specifications.

The program and its output are shown in Fig.8.4. The output illustrates the following features of the **%s** specifications.

1. When the field width is less than the length of the string, the entire string is printed.

2. The integer value on the right side of the decimal point specifies the number of   characters to be printed.

3. When the number of characters to be printed is specified as zero, nothing is printed.

4. The minus sign in the specification causes the string to be printed left-justified.

5. The specification % **.**ns prints the first n characters of the string

WRITING STRINGS USING **%s** FORMAT

Program

```
    main()
    {
        char  country[15] = "United Kingdom";

        printf("\n\n");
        printf("*123456789012345*\n");
        printf(" --------------- \n");

        printf("%15s\n", country);
        printf("%5s\n", country);
        printf("%15.6s\n", country);
        printf("%-15.6s\n", country);
        printf("%15.0s\n", country);
        printf("%.3s\n", country);
        printf("%s\n", country);
        printf("--------------- \n");
    }
```

Output

```
   *123456789012345*
  ------------------
    United Kingdom
   United Kingdom
             United
   United

   Uni
   United Kingdom
  ------------------
```

Fig.8.4 *Writing strings using %s format*

Example 8.5

Write a program using **for loop** to print the following output.

```
  C
CP
CPr
CPro
.....
.....
  CProgramming
```
```
CProgramming
.....
.....
CPro
CPr
CP
C
```

The outputs of the program in Fig.8.5, for variable specifications **%12.*s, %.*s,** and **%*.1s** are shown in Fig.8.6, which further illustrates the variable field width and the precision specifications.

PRINTING **SEQUENCES OF CHARACTERS**

Program

```
main()
{
    int  c, d;
    char  string[] = "CProgramming";

    printf("\n\n");
    printf("--------------\n");
    for( c = 0 ; c <= 11 ; c++ )
    {
       d = c + 1;
       printf("|%-12.*s|\n", d, string);
    }
    printf("|------------|\n");

    for( c = 11 ; c >= 0 ; c-- )
    {
       d = c + 1;
       printf("|%-12.*s|\n", d, string);
```

```
          }
      printf("--------------\n");
    }
```

Output

C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming
CProgramming
CProgrammin
CProgrammi
CProgramm
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C

Fig.8.5 *Illustration of variable field specifications*

Example 8.6

Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

The program is shown in Fig.8.7. In ASCII character set, the decimal numbers 65 to 90 represent uppercase alphabets and 97 to 122 represent lowercase alphabets. The values from 91 to 96 are excluded using an **if** statement in the **for** loop.

PRINTING ALPHABET SET IN DECIMAL AND CHARACTER FORM

P**rogram**

```
   main()
   {
       char   c;

       printf("\n\n");
       for( c = 65 ; c <= 122 ; c = c + 1 )
       {
          if( c > 90   &&   c < 97 )
             continue;
          printf("|%4d - %c ", c, c);
       }
       printf("|\n");
   }
```

Output

```
|  65 - A  |  66 - B  |  67 - C  |  68 - D  |  69 - E  |  70 - F
|  71 - G  |  72 - H  |  73 - I  |  74 - J  |  75 - K  |  76 - L
|  77 - M  |  78 - N  |  79 - O  |  80 - P  |  81 - Q  |  82 - R
|  83 - S  |  84 - T  |  85 - U  |  86 - V  |  87 - W  |  88 - X
|  89 - Y  |  90 - Z  |  97 - a  |  98 - b  |  99 - c  | 100 - d
| 101 - e  | 102 - f  | 103 - g  | 104 - h  | 105 - i  | 106 - j
| 107 - k  | 108 - l  | 109 - m  | 110 - n  | 111 - o  | 112 - p
| 113 - q  | 114 - r  | 115 - s  | 116 - t  | 117 - u  | 118 - v
| 119 - w  | 120 - x  | 121 - y  | 122 - z  |
```

Fig.8.7 *Printing of the alphabet set in decimal and character form*

Example 8.7

The names of employees of an organization are stored in three arrays, namely **first_name**, **second_name,** and **last_name**. Write a program to concatenate the three parts into one string to be called **name**.

The program is given in Fig.8.8. Three **for** loops are used to copy the three strings. In the first loop, the characters contained in the **first_name** are copied into the variable **name** until the *null* character is reached. The *null* character is not copied; instead it is replaced by a *space* by the assignment statement

> **name[i] =** $^{\backslash /}$ **;**

Similarly, the **second_name** is copied into **name**, starting from the column just after the space created by the above statement. This is achieved by the assignment statement

> **name[i+j+1] = second_name[j];**

If **first_name** contains 4 characters, then the value of i at this point will be 4 and therefore the first character from **second_name** will be placed in the *fifth cell* of **name.** Note that we have stored a space in the *fourth cell.*

In the same way, the statement

> **name[i+j+k+2] = last_name[k];**

is used to copy the characters from **last_name** into the proper locations of **name.**
At the end, we place a null character to terminate the concatenated string **name**. In this example, it is important to note the use of the expressions **i+j+1** and **i+j+k+2.**

```
                    CONCATENATION OF STRINGS
```

Program

```
  main()
  {
      int  i, j, k ;
      char    first_name[10] = {"VISWANATH"}  ;
      char  second_name[10] = {"PRATAP"} ;
      char     last_name[10] = {"SINGH"} ;
      char    name[30] ;
    /* Copy first_name into name */

      for( i = 0 ; first_name[i] != '\0' ; i++ )
         name[i] = first_name[i] ;

    /* End first_name with a space */

      name[i] = ' ' ;

    /* Copy second_name into name */
```

```
    for( j = 0 ; second_name[j] != '\0' ; j++ )
        name[i+j+1] = second_name[j] ;
/* End second_name with a space */

    name[i+j+1] = ' ' ;
/* Copy last_name into name */

    for( k = 0 ; last_name[k] != '\0'; k++ )
        name[i+j+k+2] = last_name[k] ;
/* End name with a null character */

    name[i+j+k+2] = '\0' ;

    printf("\n\n") ;
    printf("%s\n", name) ;
}
```

Output

```
        VISWANATH PRATAP SINGH
```

Fig.8.8 *Concatenation of strings*

Example 8.8

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.8.9. 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

```c
#include    <string.h>

main()
{   char  s1[20], s2[20], s3[20];
    int   x, l1, l2, l3;

    printf("\n\nEnter two string constants \n");
    printf("?");
    scanf("%s %s", s1, s2);

 /* comparing s1 and s2 */

    x = strcmp(s1, s2);
    if(x != 0)
    {   printf("\n\nStrings are not equal \n");
        strcat(s1, s2);    /* joining s1 and s2 */
    }
    else
        printf("\n\nStrings are equal \n");

/* copying s1 to s3

    strcpy(s3, s1);

 /* Finding length of strings */

    l1 = strlen(s1);
    l2 = strlen(s2);
    l3 = strlen(s3);

 /* output */

    printf("\ns1 = %s\t length = %d characters\n", s1, l1);
    printf("s2 = %s\t length = %d characters\n", s2, l2);
    printf("s3 = %s\t length = %d characters\n", s3, l3);
}
```

```
Enter two string constants
? New York

Strings are not equal

s1 = NewYork       length = 7 characters
s2 = York          length = 4 characters
s3 = NewYork       length = 7 characters


Enter two string constants
? London    London

Strings are equal

s1 = London        length = 6 characters
s2 = London        length = 6 characters
s3 = London        length = 6 characters
```

Fig.7.9 *Illustration of string handling functions*

Example 8.9

Write a program that would sort a list of names in alphabetical order.

A program to sort the list of strings in alphabetical order is given in Fig.8.10. It employs the method of bubble sorting described in Case Study 1 in the previous chapter.

SORTING OF STRINGS IN ALPHABETICAL ORDER

Program

```
#define ITEMS   5
#define MAXCHAR 20

main( )
{
    char string[ITEMS][MAXCHAR], dummy[MAXCHAR];
     int  i = 0, j = 0;

   /* Reading the list */

   printf ("Enter names of %d items \n ",ITEMS);
   while (i < ITEMS)
        scanf ("%s", string[i++]);

   /* Sorting begins */

   for (i=1; i < ITEMS; i++) /* Outer loop begins */
   {
        for (j=1; j <= ITEMS-i ; j++) /*Inner loop begins*/
        {
            if (strcmp (string[j-1], string[j]) > 0)
            {   /* Exchange of contents */
                strcpy (dummy, string[j-1]);
                strcpy (string[j-1], string[j]);
                strcpy (string[j], dummy );
            }
        } /* Inner loop ends */

    } /* Outer loop ends */
```

```
        /* Sorting completed */

    printf ("\nAlphabetical list \n\n");
    for (i=0; i < ITEMS ; i++)
        printf ("%s", string[i]);
}
```

Output

```
Enter names of 5 items
London Manchester Delhi Paris Moscow
Alphabetical list

Delhi
London
Manchester
Moscow
Paris
```

Fig.8.10 *Sorting of strings.*

A program with three user-defined functions is given in Fig.9.4. **main** is the calling function that calls **printline** and **value** functions. Since both the called functions contain no arguments, there are no argument declarations. The **printline** function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The **value** function calculates the value of principal amount after a certain period of years and prints the results. The following equation is evaluated repeatedly:

$$value = principal(1+interest\text{-}rate)$$

FUNCTIONS WITH THE ARGUMENTS, NO RETURN VALUES

Program

```
/* Function declaration */
void printline (void);
void value (void);

main()
{
    printline();
    value();
    printline();
}
/*       Function1: printline( )        */

void printline(void)   /* contains no arguments */
{
    int i ;

    for(i=1; i <= 35; i++)
       printf("%c",'-');
    printf("\n");
}
/*       Function2: value( )            */
void value(void)       /* contains no arguments */
{
    int    year, period;
    float  inrate, sum, principal;

    printf("Principal amount?");
    scanf("%f", &principal);
    printf("Interest rate?   ");
    scanf("%f", &inrate);
    printf("Period?          ");
    scanf("%d", &period);

    sum = principal;
    year = 1;
    while(year <= period)
    {
        sum = sum *(1+inrate);
        year = year +1;
    }
    printf("\n%8.2f %5.2f %5d %12.2f\n",
            principal,inrate,period,sum);
}
```

```
     -----------------------------------
        Principal amount?  5000
        Interest rate?     0.12
        Period?            5

        5000.00  0.12      5       8811.71
     -----------------------------------
```

Fig.9.4 Functions with no arguments and no return values

Example 9.2

Modify the program of Example 9.1 to include the arguments in the function calls.

The modified program with function arguments is presented in Fig.9.7. Most of the program is identical to the program in Fig.9.4. The input prompt and **scanf** assignment statement have been moved from **value** function to **main.** The variables **principal, inrate,** and **period** are declared in **main** because they are used in main to receive data. The function call

      value(principal, inrate, period);

passes information it contains to the function **value.**

The function header of **value** has three formal arguments **p,r,** and **n** which correspond to the actual arguments in the function call, namely, **principal, inrate,** and **period.** On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

      p = principal;
      r = inrate;
      n = period;

```
                    FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUES
```

Program

```
   /* prototypes */
   void printline (char c);
   void value (float, float, int);

   main( )
   {
        float principal, inrate;
        int period;

        printf("Enter principal amount, interest");
        printf(" rate, and period \n");
        scanf("%f %f %d",&principal, &inrate, &period);
        printline('Z');
        value(principal,inrate,period);
        printline('C');
   }
void printline(char ch)
   {
        int i ;
        for(i=1; i <= 52; i++)
             printf("%c",ch);
        printf("\n");
   }
```

```
void value(float p, float r, int n)
{
    int year ;
    float sum ;
    sum = p ;
    year = 1;
    while(year <= n)
    {
        sum = sum * (1+r);
        year = year +1;
    }
    printf("%f\t%f\t%d\t%f\n",p,r,n,sum);
}
```

Output

```
Enter principal amount, interest rate, and period
5000 0.12   5
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
5000.000000       0.120000           5        8811.708984
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Fig.9.7 *Functions with arguments but no return values*

Example 9.3

In the program presented in Fig. 9.7 modify the function **value,** to return the final amount calculated to the **main**, which will display the required output at the terminal. Also extend the versatility of the function **printline** by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig. 9.9. One major change is the movement of the **printf** statement from **value** to **main.**

FUNCTIONS WITH ARGUMENTS AND RETURN VALUES

Program

```
void printline (char ch, int len);
value (float, float, int);

main( )
{
    float principal, inrate, amount;
    int period;
    printf("Enter principal amount, interest");
    printf("rate, and period\n");
    scanf(%f %f %d", &principal, &inrate, &period);
    printline ('*' , 52);
    amount = value (principal, inrate, period);
    printf("\n%f\t%f\t%d\t%f\n\n",principal,
            inrate,period,amount);
    printline('=',52);
}

void printline(char ch, int len)
{
    int i;
    for (i=1;i<=len;i++) printf("%c",ch);
    printf("\n");
}
```

```
value(float p, float r, int n) /* default return type */
{
      int year;
      float sum;
      sum = p; year = 1;
      while(year <=n)
      {
            sum = sum * (1+r);
            year = year +1;
      }
      return(sum);            /* returns int part of sum */
}
```

Output

```
Enter principal amount, interest rate, and period
5000  0.12     5
**************************************************

5000.000000            0.1200000  5     8811.000000

= = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

Fig.9.9 *Functions with arguments and return values*

Example 9.4

Write a function **power** that computes x raised to the power y for integers x and y and returns double-type value.

Fig 9.10 shows a **power** function that returns a **double.** The prototype declaration

double power(int, int);

appears in **main**, before **power** is called.

POWER FUNCTIONS

Program

```
main( )
{     int x,y;                      /*input data */

      double power(int, int);     /* prototype declaration*/

      printf("Enter x,y:");

      scanf("%d %d" , &x,&y);
      printf("%d to power %d is %f\n", x,y,power (x,y));
}

double power (int x, int y);

{
      double p;
      p = 1.0 ;           /* x to power zero */

      if(y >=o)
            while(y--)   /* computes positive powers */
             p *= x;
      else
            while (y++)  /* computes negative powers */
             p /= x;
      return(p);

}
```

```
    Enter x,y:16²
    16 to power 2 is 256.000000

    Enter x,y:16⁻²
    16 to power -2 is 0.003906
```

Fig 9.10 *Illustration of return of float values*

Example 9.5

Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$S.D = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x}-x_i)^2}$$ Where $\bar{x}$ is the mean of the values.

FUNCTIONS WITH ARRAYS

Program

```c
#include     <math.h>
#define SIZE    5
float std_dev(float a[], int n);
float mean (float a[], int n);

main( )
{
     float value[SIZE];
     int i;

     printf("Enter %d float values\n", SIZE);
     for (i=0 ;i < SIZE ; i++)
         scanf("%f", &value[i]);
     printf("Std.deviation is %f\n", std_dev(value,SIZE));
}
float std_dev(float a[], int n)

{
     int i;
     float x, sum = 0.0;
     x = mean (a,n);
     for(i=0; i < n; i++)
         sum += (x-a[i])*(x-a[i]);
     return(sqrt(sum/(float)n));
}
float mean(float a[],int n)

{
     int i ;
     float sum = 0.0;
     for(i=0 ; i < n ; i++)
         sum = sum + a[i];
     return(sum/(float)n);
}
```

```
          Enter 5 float values
          35.0 67.0 79.5 14.20 55.75

          Std.deviation is 23.231582
```

Fig.9.11 Passing of arrays to a function

Example 9.6

Write a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function **sort()** is given in Fig.9.12. Its output clearly shows that a function can change the values in an array passed as an argument.

SORTING OF ARRAY ELEMENTS

Program

```
  void sort(int m, int x[ ]);
  main()
  {
      int i;
      int marks[5] = {40, 90, 73, 81, 35};

      printf("Marks before sorting\n");
      for(i = 0; i < 5; i++)
         printf("%d ", marks[i]);
      printf("\n\n");

      sort (5, marks);

      printf("Marks after sorting\n");
      for(i = 0; i < 5; i++)
         printf("%4d", marks[i]);
      printf("\n");
  }

  void sort(int m, int x[ ])

  {
      int i, j, t;

      for(i = 1; i <= m-1; i++)
         for(j = 1; j <= m-i; j++)
            if(x[j-1] >= x[j])
            {
               t = x[j-1];
               x[j-1] = x[j];
               x[j] = t;
            }
  }
```

Output

```
          Marks before sorting
          40 90 73 81 35

          Marks after sorting
          35   40   73   81   90
```

Fig.9.12 Sorting of array elements using a function

A program with two subprograms **function1** and **function2** is shown in Fig.9.13. **m** is an automatic variable and it is declared at the beginning of each function. **m** is initialized to 10, 100, and 1000 in function1, function2, and **main** respectively.

When executed, **main** calls **function2** which in turn calls **function1**. When **main** is active, m = 1000; but when **function2** is called, the **main**'s **m** is temporarily put on the shelf and the new local **m** = 100 becomes active. Similarly, when **function1** is called, both the previous values of **m** are put on the shelf and the latest value of **m** (=10) becomes active. As soon as **function1** (m=10) is finished, **function2** (m=100) takes over again. As soon it is done, **main** (m=1000) takes over. The output clearly shows that the value assigned to **m** in one function does not affect its value in the other functions; and the local value of **m** is destroyed when it leaves a function.

ILLUSTRATION OF WORKING OF **auto** VARIABLES

Program

```
void function1(void);
void function2(void);
main( )
{
    int m = 1000;
    function2();

    printf("%d\n",m);   /* Third output */
}
void function1(void)
{
    int m = 10;

    printf("%d\n",m);   /* First output */
}


void function2(void)
{
    int m = 100;

    function1();
    printf("%d\n",m);   /* Second output */
}
```

Output

```
    10
    100
    1000
```

Fig.9.13 *Working of automatic variables*

A program to illustrate the properties of global variables is presented in Fig.9.14. Note that variable **x** is used in all functions but none except **fun2,** has a definition for **x.** Because **x** has been declared 'above' all the functions, it is available to each function without having to pass x as a function argument. Further, since the value of **x** is directly available, we need not use **return(x)** statements in **fun1** and **fun3.** However, since **fun2** has a definition of **x,** it returns its local value of **x** and therefore uses a **return** statement. In **fun2**, the global **x** is not visible. The local **x** hides its visibility here.

```
              ILLUSTRATION OF PROPERTIES OF GLOBAL VARIABLES
```

Program

```c
   int fun1(void);
   int fun2(void);
   int fun3(void);
   int x ;              /* global */
   main( )
   {
       x = 10 ;        /* global x */
       printf("x = %d\n", x);
       printf("x = %d\n", fun1());
       printf("x = %d\n", fun2());
       printf("x = %d\n", fun3());
   }
   fun1(void)
   {
       x = x + 10 ;
   }
   int fun2(void)
   {
       int x ;         /* local */
       x = 1 ;
       return (x);
   }
   fun3(void)
   {
       x = x + 10 ;  /* global x */
   }
```

Output   x = 10
               x = 20
               x = 1
               x = 30

Fig.9.14 *Illustration of global variables*

The program in Fig.9.15 explains the behaviour of a static variable.

ILLUSTRATION OF STATIC VARIABLE

Program

```
void stat(void);
main ( )
{
   int i;
   for(i=1; i<=3; i++)
    stat( );
 }
void stat(void)
{
   static int x = 0;

   x = x+1;
   printf("x = %d\n", x);
}
```

Output

```
x = 1
x = 2
x = 3
```

Fig.9.15 *Illustration of static variable*

Structure definition along with the program is shown in Fig.10.1. The **scanf** and **printf** functions illustrate how the member operator \`.' is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

DEFINING AND ASSIGNING VALUES TO STRUCTURE MEMBERS

Program
```
struct  personal
{
    char  name[20];
    int   day;
    char  month[10];
    int   year;
    float salary;
};
main()
{
    struct personal person;
    printf("Input Values\n");
    scanf("%s %d %s %d %f",
             person.name,
           &person.day,
            person.month,
           &person.year,
           &person.salary);
    printf("%s %d %s %d %f\n",
             person.name,
             person.day,
             person.month,
             person.year,
             person.salary);
}
```

Output
```
Input Values

M.L.Goel 10 January 1945 4500
M.L.Goel 10 January 1945 4500.00
```

Fig.10.1 *Defining and accessing structure members*

The program shown in Fig.10.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

COMPARISON OF STRUCTURE VARIABLES

Program

```
struct class
{
    int  number;
    char name[20];
    float marks;
};

main()
{
    int  x;
    struct class student1 = {111,"Rao",72.50};
    struct class student2 = {222,"Reddy", 67.00};
    struct class student3;

    student3 = student2;

    x = ((student3.number ==  student2.number) &&
         (student3.marks  ==  student2.marks)) ? 1 : 0;

    if(x == 1)
    {
        printf("\nstudent2 and student3 are same\n\n");
        printf("%d %s %f\n", student3.number,
                             student3.name,
                             student3.marks);
    }
    else
        printf("\nstudent2 and student3 are different\n\n");

}
```

Output

```
student2 and student3 are same

222 Reddy 67.000000
```

Fig.10.2 Comparing and copying structure variables

The program is shown in Fig.10.4. We have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an **array** total to keep the subject-totals and the grand-total. The grand-total is given by **total.total.** Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name **total.total** represents the **total** member of the structure variable total.

Program

```
struct marks
{
    int  sub1;
    int  sub2;
    int  sub3;
    int  total;
};
main()
{
    int  i;
    struct marks student[3] =  {{45,67,81,0},
                                {75,53,69,0},
                                {57,36,71,0}};
    struct marks total;

    for(i = 0; i <= 2; i++)
    {
        student[i].total = student[i].sub1 +
                           student[i].sub2 +
                           student[i].sub3;
        total.sub1 = total.sub1 + student[i].sub1;
        total.sub2 = total.sub2 + student[i].sub2;
        total.sub3 = total.sub3 + student[i].sub3;
        total.total = total.total + student[i].total;
    }
    printf(" STUDENT          TOTAL\n\n");
    for(i = 0; i <= 2; i++)
       printf("Student[%d]      %d\n", i+1,student[i].total);

    printf("\n SUBJECT          TOTAL\n\n");

    printf("%s       %d\n%s        %d\n%s        %d\n",
           "Subject 1   ", total.sub1,
           "Subject 2   ", total.sub2,
           "Subject 3   ", total.sub3);

    printf("\nGrand Total = %d\n", total.total);
}
```

Output

```
 STUDENT           TOTAL
 Student[1]         193
 Student[2]         197
 Student[3]         164

 SUBJECT           TOTAL
 Subject 1         177
 Subject 2         156
 Subject 3         221

 Grand Total  = 554
```

Fig.10.4 *Illustration of subscripted structure variables*

The modified program is shown in Fig.10.5. You may notice that the use of array name for subjects has simplified in code.

                    ARRAYS WITHIN A STRUCTURE
_____

Program

```
   main()
   {
       struct  marks
       {
           int  sub[3];
           int  total;
       };
       struct marks student[3] =
       {45,67,81,0,75,53,69,0,57,36,71,0};

       struct marks total;
       int  i,j;

       for(i = 0; i <= 2; i++)
       {
          for(j = 0; j <= 2; j++)
          {
              student[i].total += student[i].sub[j];
              total.sub[j] += student[i].sub[j];
          }
          total.total += student[i].total;
       }
       printf("STUDENT         TOTAL\n\n");
       for(i = 0; i <= 2; i++)
          printf("Student[%d]      %d\n", i+1, student[i].total);

       printf("\nSUBJECT         TOTAL\n\n");
       for(j = 0; j <= 2; j++)
          printf("Subject-%d         %d\n", j+1, total.sub[j]);

       printf("\nGrand Total  =   %d\n", total.total);

   }
```
_____
Output

```
   STUDENT         TOTAL
   Student[1]       193
   Student[2]       197
   Student[3]       164

   SUBJECT         TOTAL
   Subject-1        177
   Subject-2        156
   Subject-3        221

   Grand Total  =   554
```

Fig.10.5 *Use of subscripted members in structures*

Example 10.5

Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

A program to update an item is shown in Fig.10.6. The function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and the formal parameter **product** are declared as type **struct stores.** It is done so because the function uses the parameter **product** to receive the structure variable **item** and also to return the updated values of **item**.

The function **mul** is of type **float** because it returns the product of **price** and **quantity.** However, the parameter **stock,** which receives the structure variable **item** is declared as type **struct stores.**

The entire structure returned by **update** can be copied into a structure of identical type. The statement

　　　　item = update(item,p_increment,q_increment);

replaces the old values of **item** by the new ones.

<div align="center">STRUCTURES AS FUNCTION PARAMETERS</div>

Program

```
/*        Passing a copy of the entire structure        */
  struct stores
  {
      char  name[20];
      float price;
      int   quantity;
  };
  struct stores update (struct stores product, float p, int q);
  float mul (struct stores stock);

  main()
  {
      float    p_increment, value;
      int      q_increment;

      struct stores item = {"XYZ", 25.75, 12};

      printf("\nInput increment values:");
      printf("   price increment and quantity increment\n");
      scanf("%f %d", &p_increment, &q_increment);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      item  = update(item, p_increment, q_increment);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      printf("Updated values of item\n\n");
      printf("Name      : %s\n",item.name);
      printf("Price     : %f\n",item.price);
      printf("Quantity  : %d\n",item.quantity);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      value  = mul(item);
  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
      printf("\nValue of the item  =  %f\n", value);
  }
  struct stores update(struct stores product, float p, int q)
  {
      product.price += p;
      product.quantity += q;
      return(product);
```

```
    }
    float mul(struct stores stock)
    {
        return(stock.price * stock.quantity);
    }
```

Output

```
Input increment values:    price increment and quantity increment
10 12
Updated values of item

Name      : XYZ
Price     : 35.750000
Quantity  : 24

Value of the item  =  858.000000
```

Fig.10.6 Using structure as a function parameter

The program shown in Fig.11.4, declares and initializes four variables and then prints out these values with their respective storage locations. Notice that we have used %u format for printing address values. Memory addresses are unsigned integers.

ACCESSING ADDRESSES OF VARIABLES

Program

```
main()
{
    char   a;
    int    x;
    float  p, q;

    a  = 'A';
    x  = 125;
    p  = 10.25, q = 18.76;

    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);

}
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442.
18.760000 is stored at addr 4438.
```

Fig.11.4 Accessing the address of a variable

The program and output are shown in Fig.11.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

ACCESSING VARIABLES USING POINTERS

Program

```
main()
{
    int   x, y;
    int   *ptr;

    x = 10;
    ptr = &x;
    y = *ptr;

    printf("Value of x is %d\n\n",x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
```

```
        printf("%d is stored at addr %u\n", y, &*ptr);
        printf("%d is stored at addr %u\n", ptr, &ptr);
        printf("%d is stored at addr %u\n", y, &y);

        *ptr = 25;
        printf("\nNow x = %d\n",x);

    }
```

```
  Value of x is 10

  10   is stored at addr 4104
  10   is stored at addr 4104
  10   is stored at addr 4104
  10   is stored at addr 4104
  4104 is stored at addr 4106
  10   is stored at addr 4108

 Now x = 25
```

Fig.11.5 *Accessing a variable through its pointer*

Example 11.3

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig.11.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

$$4* - *p2 / *p1 + 10$$

is evaluated as follows:

$$((4 * (-(*p2))) / (*p1)) + 10$$

When *p1 = 12 and *p2 = 4, this expression evaluates to 9. Remember, since all the variables are of type int, the entire evaluation is carried out using the integer arithmetic.

```
                    ILLUSTRATION OF POINTER EXPRESSIONS
```

Program

```
  main()
  {
      int  a, b, *p1, *p2, x, y, z;

      a  = 12;
      b  =  4;
      p1 = &a;
      p2 = &b;

      x  =  *p1 * *p2 - 6;
      y  =  4*  - *p2 / *p1 + 10;

      printf("Address of a = %u\n", p1);
      printf("Address of b = %u\n", p2);
      printf("\n");
      printf("a = %d, b = %d\n", a, b);
      printf("x = %d, y = %d\n", x, y);

      *p2  = *p2 + 3;
      *p1  = *p2 - 5;
       z   = *p1 * *p2 - 6;
```

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

```
   Address of a = 4020
   Address of b = 4016

   a = 12, b = 4
   x = 42, y = 9
   a = 2, b = 7, z = 8
```

Fig.11.7 *Evaluation of pointer expressions*

Example 11.4

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig.11.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to **p** each time we go through the loop.

<div align="center">POINTERS IN ONE-DIMENSIONAL ARRAY</div>

Program

```
   main()
   {
       int *p, sum, i;
       int x[5] = {5,9,6,3,7};

       i  = 0;
       p  = x;            /* initializing with base address of x */

       printf("Element   Value   Address\n\n");
       while(i < 5)
       {
          printf(" x[%d] %d %u\n", i, *p, p);
          sum = sum + *p;    /* accessing array element  */
          i++, p++;          /* incrementing pointer     */
       }
       printf("\n  Sum    =   %d\n", sum);
       printf("\n  &x[0]  =   %u\n", &x[0]);
       printf("\n  p      =   %u\n", p);
   }
```

Output

```
         Element     Value      Address
          x[0]          5          166
          x[1]          9          168
          x[2]          6          170
          x[3]          3          172
          x[4]          7          174

          Sum     =   55
          &x[0]   =   166
          p       =   176
```
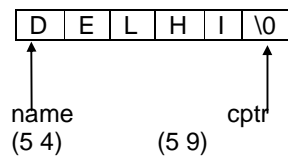
Fig.11.8 *Accessing array elements using the pointer*

Example 11.5

Write a program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig.11.10. The statement

char *cptr = name;

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

while(*cptr != '\0')

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

length = cptr - name;

gives the length of the string **name.**

The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

POINTERS AND CHARACTER STRINGS

Program

```
main()
{
    char   *name;
    int    length;
    char   *cptr = name;

    name   = "DELHI";

    printf ("%s\n", name);

    while(*cptr != '\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length = cptr - name;
    printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58

Length of the string = 5
```

Fig.11.10 String handling by pointers

Example 11.6

Write a function using pointers to exchange the values stored in two locations in the memory.

The programming Fig.11.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

```
                     POINTERS AS FUNCTION PARAMETERS
-----------------------------------------------------------------------------
Program
   void exchange (int *, int *);     /* prototype */
   main()
   {
       int  x, y;
       x = 100;
       y = 200;
       printf("Before exchange  : x = %d   y = %d\n\n", x, y);
       exchange(&x,&y);            /* call */
       printf("After exchange   : x = %d   y = %d\n\n", x, y);
   }
   exchange (int *a, int *b)
   {
       int t;

       t = *a;    /* Assign the value at address a to t */
       *a = *b;   /* put b into a */
       *b = t;    /* put t into b */
   }
```

```
Output

   Before exchange  : x = 100   y = 200

   After exchange   : x = 200   y = 100
```

Fig.11.11 Passing of pointers as function parameters

Example 11.7

Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig.11.12. The printing is done by the function **table** by evaluating the function passed to it by the **main.**

With **table,** we declare the parameter **f** as a pointer to a function as follows:

double (*f)();

The value returned by the function is of type **double.** When **table** is called in the statement

table (y, 0.0, 2, 0.5);

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list. During the execution of **table,** the statement

value = (*f)(a);

calls the function **y** which is pointed to by **f,** passing it the parameter **a.** Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

table (cos, 0.0, PI, 0.5);

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

ILLUSTRATION OF POINTERS TO FUNCTIONS

Program

```
#include   <math.h>
#define   PI   3.1415926
double y(double);
double cos(double);
double table (double(*f)(), double, double, double);

main()
{
   printf("Table of y(x) = 2*x*x-x+1\n\n");
   table(y, 0.0, 2.0, 0.5);

   printf("\nTable of cos(x)\n\n");
   table(cos, 0.0, PI, 0.5);
}
double table(double(*f)(),double min, double max, double step)
{
   double a, value;
   for(a = min; a <= max; a += step)
   {
      value = (*f)(a);
      printf("%5.2f   %10.4f\n", a, value);
   }
}
double y(double x)
{
   return(2*x*x-x+1);
}
```

Output

```
    Table of y(x) = 2*x*x-x+1
        0.00       1.0000
        0.50       1.0000
        1.00       2.0000
        1.50       4.0000
        2.00       7.0000

    Table of cos(x)
        0.00       1.0000
        0.50       0.8776
        1.00       0.5403
        1.50       0.0707
        2.00      -0.4161
        2.50      -0.8011
        3.00      -0.9900
```

Fig.11.12   *Use of pointers to functions*

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig.11.13. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent)** is also used as the loop control index in **for** loops.

<div align="center">POINTERS TO STRUCTURE VARIABLES</div>

Program

```
struct invent
{
    char  *name[20];
    int   number;
    float price;
};
main()
{
    struct invent product[3], *ptr;
    printf("INPUT\n\n");
    for(ptr = product; ptr < product+3; ptr++)
      scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);

    printf("\nOUTPUT\n\n");

     ptr = product;
     while(ptr < product + 3)
     {
         printf("%-20s %5d %10.2f\n",
                 ptr->name,
                 ptr->number,
                 ptr->price);
         ptr++;
     }
}
```

Output

```
INPUT

Washing_machine    5     7500
Electric_iron     12      350
Two_in_one         7     1250

OUTPUT

Washing machine    5     7500.00
Electric_iron     12      350.00
Two_in_one         7     1250.00
```

Fig.11.13 *Pointer to structure variables*

Example 12.1

Write a program to read data from the keyboard, write it to a file called **INPUT**, again read the same data from the **INPUT** file, and display it on the screen.

A program and the related input and output data are shown in Fig.12.1. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an **EOF** character, which is *control-Z* in the reference system. (This may be control-D in other systems). The file INPUT is closed at this signal.

                    WRITING TO AND READING FROM A FILE

Program

```
#include  <stdio.h>

main()
{
    FILE *f1;
    char c;

    printf("Data Input\n\n");
    /* Open the file INPUT */
    f1 = fopen("INPUT", "w");

    /* Get a character from keyboard   */
    while((c=getchar()) != EOF)

       /* Write a character to INPUT  */
        putc(c,f1);

    /* Close the file INPUT   */
    fclose(f1);

    printf("\nData Output\n\n");

    /* Reopen the file INPUT    */
    f1 = fopen("INPUT","r");

   /* Read a character from INPUT*/
    while((c=getc(f1)) != EOF)

       /* Display a character on screen */
        printf("%c",c);

    /* Close the file INPUT       */
    fclose(f1);
}
```

Output

```
Data Input

This is a program to test the file handling
features on this system^Z

Data Output

This is a program to test the file handling
features on this system
```

Fig.12.1 Character oriented read/write operations on a file

Example 12.2

A file named **DATA** contains a series of integer numbers. Code a program to read these numbers and then write all 'odd' numbers to a file to be called **ODD** and all `even' numbers to a file to be called **EVEN.**

The program is shown in Fig.12.2. It uses three files simultaneously and therefore we need to define three-file pointers **f1, f2** and **f3.**

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

    putw(number, f1);

Notice that when we type -1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

    (number = getw(f1)) != EOF

reads a value, assigns the same to **number**, and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

<div align="center">HANDLING OF INTEGER DATA FILES</div>

Program

```c
#include  <stdio.h>
main()
{
    FILE  *f1, *f2, *f3;
    int   number, i;

    printf("Contents of DATA file\n\n");
    f1 = fopen("DATA", "w");        /* Create DATA file    */
    for(i = 1; i <= 30; i++)
    {
       scanf("%d", &number);
       if(number == -1) break;
       putw(number,f1);
    }
    fclose(f1);

    f1 = fopen("DATA", "r");
    f2 = fopen("ODD", "w");
    f3 = fopen("EVEN", "w");

    /* Read from DATA file */
    while((number = getw(f1)) != EOF)
    {
       if(number %2 == 0)
         putw(number, f3);    /*  Write to EVEN file  */
       else
         putw(number, f2);    /*  Write to ODD file   */
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);

    f2 = fopen("ODD","r");
    f3 = fopen("EVEN", "r");
```

```
        printf("\n\nContents of ODD file\n\n");

        while((number = getw(f2)) != EOF)
            printf("%4d", number);
        printf("\n\nContents of EVEN file\n\n");

        while((number = getw(f3)) != EOF)
            printf("%4d", number);

        fclose(f2);
        fclose(f3);

   }
```

Output

```
Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888    0 232 454
```

_____
Fig.12.2 Operations on integer data

Example 12.3

Write a program to open a file named INVENTORY and store in it the following data:

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3 247		31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

The program is given in Fig.12.3. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin,** which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp.** Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout,** which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file.…é

```
                HANDLING OF FILES WITH MIXED DATA TYPES
                         (fscanf and fprinf)
```
_____

Program

```
   #include  <stdio.h>

   main()
   {
       FILE   *fp;
       int    number, quantity, i;
       float  price, value;
       char   item[10], filename[10];

       printf("Input file name\n");
```

```
        scanf("%s", filename);

        fp = fopen(filename, "w");

        printf("Input inventory data\n\n");
        printf("Item name  Number   Price   Quantity\n");
        for(i = 1; i <= 3; i++)
        {
            fscanf(stdin, "%s %d %f %d",
                           item, &number, &price, &quantity);
            fprintf(fp, "%s %d %.2f %d",
                           item, number, price, quantity);
        }
        fclose(fp);
        fprintf(stdout, "\n\n");

        fp = fopen(filename, "r");

        printf("Item name  Number    Price    Quantity     Value\n");
        for(i = 1; i <= 3; i++)
        {
            fscanf(fp, "%s %d %f d",item,&number,&price,&quantity);
            value = price * quantity;
            fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n",
                           item, number, price, quantity, value);
        }
        fclose(fp);
    }
```

Output

```
    Input file name
    INVENTORY
    Input inventory data

    Item name  Number   Price   Quantity
    AAA-1   111   17.50   115
    BBB-2   125   36.00   75
    C-3     247   31.75   104

    Item name  Number    Price    Quantity     Value
    AAA-1           111    17.50        115     2012.50
    BBB-2           125    36.00         75     2700.00
    C-3             247    31.75        104     3302.00
```

Fig.12.3 Operations on mixed data types

Example 12.4

Write a program to illustrate error handling in file operations.

The program shown in Fig.12.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

fopen("TETS", "r");

returns a **NULL** pointer because the file TETS does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

Program

```c
#include  <stdio.h>

main()
{
    char  *filename;
     FILE  *fp1, *fp2;
     int   i, number;

     fp1 = fopen("TEST", "w");
     for(i = 10; i <= 100; i += 10)
        putw(i, fp1);

     fclose(fp1);

     printf("\nInput filename\n");

 open_file:
     scanf("%s", filename);

     if((fp2 = fopen(filename,"r")) == NULL)
     {
        printf("Cannot open the file.\n");
        printf("Type filename again.\n\n");
        goto open_file;
     }

     else

     for(i = 1; i <= 20; i++)
     {  number = getw(fp2);
        if(feof(fp2))
        {
           printf("\nRan out of data.\n");
           break;
        }
        else
           printf("%d\n", number);
     }

     fclose(fp2);
}
```

Output

```
Input filename
TETS
Cannot open the file.
Type filename again.

TEST
10
20
30
40
50
60
70
80
```

```
90
100

Ran out of data.
```

Fig.12.4 *Illustration of error handling*

Example 12.5

Write a program that uses the functions **ftell** and **fseek.**

A program employing **ftell** and **fseek** functions is shown in Fig.12.5. We have created a file **RANDOM** with the following contents:

    Position ----> 0 1 2 25

    Character
    stored   ----> A B C . . . . .Z

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fsee(fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

        fseek(fp,-1L,2);

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

        fseek(fp, -2L, 1);

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

```
                   ILLUSTRATION OF fseek & ftell FUNCTIONS
```

Program

```
   #include <stdio.h>
   main()
   {
       FILE  *fp;
       long  n;
       char c;

       fp = fopen("RANDOM", "w");

       while((c = getchar()) != EOF)
           putc(c,fp);

       printf("No. of characters entered = %ld\n", ftell(fp));
       fclose(fp);
       fp = fopen("RANDOM","r");
       n = 0L;

       while(feof(fp) == 0)
       {
           fseek(fp, n, 0);  /*  Position to (n+1)th character */
           printf("Position of %c is %ld\n", getc(fp),ftell(fp));
           n = n+5L;
       }
       putchar('\n');

       fseek(fp,-1L,2);      /*  Position to the last character */
```

```
        do
        {
            putchar(getc(fp));
        }
        while(!fseek(fp,-2L,1));
        fclose(fp);
}
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of characters entered = 26
Position of A is 0
Position of F is 5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of   is 30

ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Fig.12.5 *Illustration of* **fseek** *and* **ftell** *functions*

Example 12.6

Write a program to append additional items to the file INVENTORY and print the total contents of the file.

The program is shown in Fig.12.6. It uses a structure definition to describe each item and a function **append()** to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against n and is terminated when they become equal.

```
                    APPENDING ITEMS TO AN EXISTING FILE
```

Program

```
    #include  <stdio.h>

    struct invent_record
    {
        char   name[10];
        int    number;
        float  price;
        int    quantity;
    };

    main()
    {
        struct invent_record item;
        char  filename[10];
        int   response;
        FILE  *fp;
        long  n;

        void append (struct invent_record 8x, file *y);

        printf("Type filename:");
        scanf("%s", filename);

        fp = fopen(filename, "a+");
```

```c
        do
        {
            append(&item, fp);
            printf("\nItem %s appended.\n",item.name);
            printf("\nDo you want to add another item\
             (1 for YES /0 for NO)?");
            scanf("%d", &response);
        }  while (response == 1);

        n = ftell(fp);        /* Position of last character  */
        fclose(fp);

        fp = fopen(filename, "r");




        while(ftell(fp) < n)
        {
            fscanf(fp,"%s %d %f %d",
            item.name, &item.number, &item.price, &item.quantity);
            fprintf(stdout,"%-8s %7d %8.2f %8d\n",
            item.name, item.number, item.price, item.quantity);
        }
        fclose(fp);
    }
    void append(struct invent_record *product, File *ptr)
    {
        printf("Item name:");
        scanf("%s", product->name);

        printf("Item number:");
        scanf("%d", &product->number);

        printf("Item price:");
        scanf("%f", &product->price);

        printf("Quantity:");
        scanf("%d", &product->quantity);

        fprintf(ptr, "%s %d %.2f %d",
                    product->name,
                    product->number,
                    product->price,
                    product->quantity);
    }
```

Output

```
Type filename:INVENTORY
Item name:XXX
Item number:444
Item price:40.50
Quantity:34

Item XXX appended.
Do you want to add another item(1 for YES /0 for NO)?1
Item name:YYY
Item number:555
Item price:50.50
```

```
Quantity:45

Item YYY appended.
Do you want to add another item(1 for YES /0 for NO)?0
AAA-1          111     17.50        115
BBB-2          125     36.00         75
C-3            247     31.75        104
XXX            444     40.50         34
YYY            555     50.50         45
```

Fig.12.6 *Adding items to an existing file*

Example 12.7

Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

Figure 12.7 shows the use of command line arguments. The command line is

F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

Each word in the command line is an argument to the **main** and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string TEXT and therefore the statement

    fp = fopen(argv[1], "w");

opens a file with the name TEXT. The **for** loop that follows immediately writes the remaining 7 arguments to the file TEXT.

<div align="center">COMMAND LINE ARGUMENTS</div>

Program

```c
#include  <stdio.h>

main(argc, argv)
int  argc;                      /*   argument count          */
char *argv[];                   /*   list of arguments       */
{
    FILE    *fp;
    int  i;
    char word[15];

    fp = fopen(argv[1], "w"); /* open file with name argv[1] */
    printf("\nNo. of arguments in Command line = %d\n\n",argc);
    for(i = 2; i < argc; i++)
       fprintf(fp,"%s ", argv[i]); /* write to file argv[1]  */
    fclose(fp);

/*  Writing content of the file to screen                    */

    printf("Contents of %s file\n\n", argv[1]);
    fp = fopen(argv[1], "r");
    for(i = 2; i < argc; i++)
    {
       fscanf(fp,"%s", word);
       printf("%s ", word);
    }

    fclose(fp);
    printf("\n\n");
```

```
        /*  Writing the arguments from memory */

            for(i = 0; i < argc; i++)
                printf("%*s \n", i*5,argv[i]);
        }
```

Output

```
    C>F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGG

    No. of arguments in Command line = 9

    Contents of TEXT file

    AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

    C:\C\F12_7.EXE
     TEXT
          AAAAAA
               BBBBBB
                    CCCCCC
                         DDDDDD
                              EEEEEE
                                   FFFFFF
                                        GGGGGG
```

Fig.12.7 *Use of command line arguments*

Example 13.1
Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig.13.2. It tests for availability of memory space of required size. If it is available, then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

USE OF **malloc** FUNCTION

Program

```
#include <stdio.h>
#include <stdlib.h>
#define NULL 0

main()
{
   int *p, *table;
   int size;

   printf("\nWhat is the size of table?");
   scanf("%d",size);

   printf("\n")

      /*------------Memory allocation --------------*/

   if((table = (int*)malloc(size *sizeof(int))) == NULL)
     {
        printf("No space available \n");
        exit(1);
     }
   printf("\n Address of the first byte is  %u\n", table);

   /* Reading table values*/

   printf("\nInput table values\n");

   for (p=table; p<table + size; p++)
        scanf("%d",p);

   /* Printing table values in reverse order*/

   for (p = table + size -1; p >= table; p --)
      printf("%d is stored at address %u \n",*p,p);

}
```

Output

```
What is the size of the table? 5
Address of the first byte is 2262

Input table values
11 12 13 14 15

15 is stored at address 2270
14 is stored at address 2268
13 is stored at address 2266
12 is stored at address 2264
11 is stored at address 2262
```

Fig.13.2 Memory allocation with *malloc*

The program is shown in Fig. 13.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

USE OF realloc AND free FUNCTIONS

Program

```
#include <stdio.h>
#include<stdlib.h>
#define NULL 0

main()
{
  char *buffer;

  /* Allocating memory */

  if((buffer = (char *)malloc(10)) == NULL)
  {
     printf("malloc failed.\n");
     exit(1);
  }
  printf("Buffer of size %d created \n",_msize(buffer));
  strcpy(buffer, "HYDERABAD");
  printf("\nBuffer contains: %s \n ", buffer);

  /* Realloction */

  if((buffer = (char *)realloc(buffer, 15)) == NULL)
  {
     printf("Reallocation failed. \n");
     exit(1);
  }
  printf("\nBuffer size modified. \n");
  printf("\nBuffer still contains: %s \n",buffer);

  strcpy(buffer, "SECUNDERBAD");
  printf("\nBuffer now contains: %s \n",buffer);
/* Freeing memory */

free(buffer);
}
```

Output

```
 Buffer of size 10 created
 Buffer contains: HYDERABAD
 Buffer size modified
 Buffer still contains: HYDERABAD
 Buffer now contains: SECUNDERABAD
```

Fig . 13.3 Reallocation and release of memory space

The program shown in Fig.13.7 first allocates a block of memory dynamically for the first node using the statement

head = (node *)malloc(sizeof(node));

which returns a pointer to a structure of type **node** that has been type defined earlier. The linked list is then created by the function **create**. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is –999, then null is assigned to the pointer variable **next** and the list ends. Otherwise, memory space is allocated to the next node using again the **malloc** function and the next value is placed into it. Not that the function **create** calls itself recursively and the process will continue until we enter the number – 999.

The items stored in the linked list are printed using the function **print** which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer. Printing algorithm is as follows;

1. Start with the first node.

2. While there are valid nodes left to print

    a) print the current item and
    b) advance to next node

Similarly, the function **count** counts the number of items in the list recursively and return the total number of items to the **main** function. Note that the counting does not include the item –999 (contained in the dummy node).

CREATING A LINEAR LINKED LIST

Program

```
#include<stdio.h>
#include<stdlib.h>
#define NULL 0

struct linked_list
{
   int number;
   struct linked_list *next;
};
typedef struct linked_list node;  /* node type defined */

main()
{
   node *head;
   void create(node *p);
   int count(node *p);
   void print(node *p);
   head = (node *)malloc(sizeof(node));
   create(head);
   printf("\n");
   printf(head);
   printf("\n");
   printf("\nNumber of items = %d \n", count(head));
}
void create(node *list)
{
   printf("Input a number\n");
   printf("(type -999 at end): ");
   scanf("%d", &list -> number); /* create current node */

   if(list->number == -999)
   {
      list->next = NULL;
   }
   else      /*create next node */
   {
```

```
        list->next = (node *)malloc(sizeof(node));
        create(list->next);
    }
    return;
}
void print(node *list)
{
    if(list->next != NULL)
    {
      printf("%d-->",list ->number);   /* print current item */

       if(list->next->next == NULL)
          printf("%d", list->next->number);

      printf(list->next);        /* move to next item */
    }
    return;
}

int count(node *list)
{
    if(list->next == NULL)
          return (0);
    else
          return(1+ count(list->next));
}
```

Output

```
    Input a number
    (type -999 to end); 60

    Input a number
    (type -999 to end); 20

    Input a number
    (type -999 to end); 10

    Input a number
    (type -999 to end); 40

    Input a number
    (type -999 to end); 30

    Input a number
    (type -999 to end); 50

    Input a number
    (type -999 to end); -999

    60 -->20 -->10 -->40 -->30 -->50 --> -999

    Number of items = 6
```

Fig. 13.7 Creating a linear linked list

Example 13.4

Write a function to insert a given item *before* a specified node known as key node.

The function **insert** shown in Fig.13.8 requests for the item to be inserted as well as the 'key node". If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new** which indicates the beginning of the new node is assigned to **head.** Note the following statements:

```
new->number = x;
new->next = head;
head = new;
```

FUNCTION INSERT

```
node *insert(node *head)
{
   node *find(node *p, int a);
   node *new;          /* pointer to new node */
   node *n1;           /* pointer to node preceding key node */
   int key;
   int x;              /* new item (number) to be inserted */

   printf("Value of new item?");
   scanf("%d", &x);
   printf("Value of key item ? (type -999 if last) ");
   scanf("%d", &key);

   if(head->number == key)       /* new node is first */
   {
       new = (node *)malloc(size of(node));
       new ->number = x;
       new->next = head;
       head = new;
   }
   else          /* find key node and insert new node */
   {             /* before the key node */
     n1 = find(head, key);    /* find key node */

     if(n1 == NULL)
        printf("\n key is not found \n");
     else       /* insert new node */
     {
        new = (node *)malloc(sizeof(node));
        new->number = x;
        new->next = n1->next;
        n1->next = new;
     }
   }
return(head);
}
node *find(node *lists, int key)
{
     if(list->next->number == key)    /* key found */
          return(list);
   else

     if(list->next->next == NULL)     /* end */
          return(NULL);
   else
     find(list->next, key);
}
```
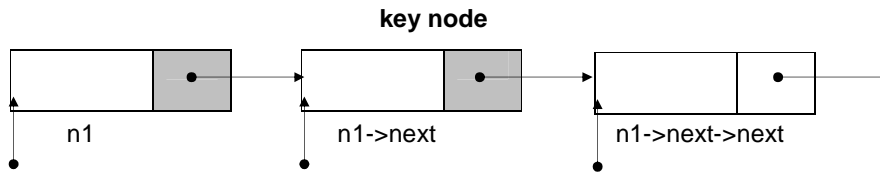
Fig. 13.8 *A function for inserting an item into a linked list*
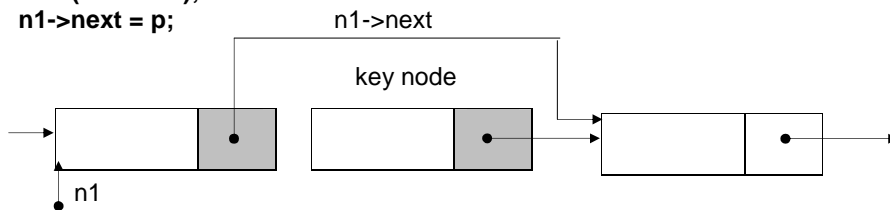
Example 13.5   Write a function to delete a specified node.

A function to delete a specified node is given in Fig.13.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable **p**, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus, the previous second node becomes the first node of the new list. If the item to be deleted is not the first one, then we use the **find** function to locate the position of 'key node' containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node. The memory space of key node that has been deleted if freed. The figure below shows the relative position of the key node.

The execution of the following code deletes the key node.

p = n1->next->next;
free (n1->next);
n1->next = p;

FUNCTION DELETE

```
node *delete(node *head)
{
   node *find(node *p, int a);
   int  key;      /* item to be deleted */
   node *n1;      /* pointer to node preceding key node */
   node *p;       /* temporary pointer */

   printf("\n What is the item (number) to be deleted?");
   scanf("%d", &key);

   if(head->number == key)  /* first node to be deleted) */
   {
      p = head->next;        /* pointer to 2nd node  in list */
      free(head);            /* release space of key node */
      head = p;              /* make head to point to 1st node */
   }
   else
   {
      n1 = find(head, key);

      if(n1 == NULL)
         printf("\n key not found \n");
      else                              /* delete key node */
      {
      p = n1->next->next;    /* pointer to the node following the keynode */
      free(n1->next);        /* free key node */
      n1->next = p;          /* establish link */
      }
   }
return(head);
}                       /* USE FUNCTION find() HERE */
```

Fig.13.9 *A function for deleting an item from linked list*