- Do not use keywords or any system library names for identifiers.
- Use meaningful and intelligent variable names.
- Do not create variable names that differ only by one or two letters.
- Each variable used must be declared for its type at the beginning of the program or function.
- All variables must be initialized before they are used in the program.
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned**, we must append the letters L and U to them.
- Floating point constants default to **double**. To make them to denote **float** or **long double**, we must append the letters F or L to the numbers.
- Do not use lowercase l for long as it is usually confused with the number 1.
- Use single quote for character constants and double quotes for string constants.
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters.
- Do not combine declarations with executable statements.
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization.
- Do not use semicolon at the end of **#define** directive.
- The character # should be in the first column.
- Do not give any space between **#** and **define.**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type.
- A variable defined before the main function is available to all the functions in the program.
- A variable defined inside a function is local to that function and not available to other functions.

ase Studies

1. Calculation of Average of Numbers

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

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 = 3.880
```

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:

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

Program

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

main()
```

```
{
    typedef float REAL ;                /* TYPE DEFINITION */
    REAL fahrenheit, celsius ;          /* DECLARATION */

                                        /* INITIALIZATION */
    fahrenheit = F_LOW ;
    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—fahrenheit-celsius*

The program prints a conversion table for reading temperature in celsius, given the fahrenheit value. 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.

Review Questions

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

(a) Any valid printable ASCII character can be used in an identifier.

(b) All variables must be given a type when they are declared.

```
x = (int) (y+0.5);
```

If y is 27.6, y+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x. Of course, the expression, being cast is not changed.

Program 3.7

Figure 3.9 shows a program using a cast to evaluate the equation

$$sum = \sum_{i=1}^{n} (1/i)$$

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. 3.9 *Use of a cast*

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:

$$\text{root1} = \frac{-b + \text{sqrt}(b^2 - 4ac)}{2a}$$

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

A program to evaluate these roots is given in Fig. 3.11. The program requests the user to input the values of **a**, **b** and **c** and outputs **root 1** and **root 2**.

```
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. 3.11 *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.

Program 5.3 A program to evaluate the power series.

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

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

The power series contains the recurrence relationship of the type

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

$T_1 = x$ for $n = 1$

$T_0 = 1$

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

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

Program

```
#define ACCURACY 0.0001
main()
{
   int n, count;
   float x, term, sum;
   printf("Enter value of x:");
   scanf("%f", &x);
   n = term = sum = count = 1;
   while (n <= 100)
  {
   term = term * x/n;
   sum = sum + term;
   count = count + 1;
   if (term < ACCURACY)
      n = 999;
   else
      n = n + 1;
  }
 printf("Terms = %d Sum = %f\n", count, sum);
}
```

Output

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

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

5000 is not calculated because of the missing **else** option.

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

Program

```
main()
{
float A, B, C;
printf("Enter three values\n");
scanf("%f %f %f", &A, &B, &C);
printf("\nLargest value is ");
if (A>B)
{
if (A>C)
   printf("%f\n", A);
else
   printf("%f\n", C);
}
else
{
   if (C>B)
      printf("%f\n", C);
   else
      printf("%f\n", B);
}
}
```

Output

```
Enter three values
23445 67379 88843
Largest value is 88843.000000
```

Fig. 5.8 *Selecting the largest of three numbers*

$y = x^n$

Program

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

Output

```
Enter the values of x and n : 2.5 4
x = 2.500000; n = 4; x to power n = 39.062500
Enter the values of x and n : 0.5 4
x = 0.500000; n = 4; x to power n = 0.062500
```

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

6.3 THE DO STATEMENT

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

Program 6.8 The program in Fig. 6.11 illustrates the use of the break statement in a C program.

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

Program

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

    printf("Number of values   = %d\n", m-1);
    printf("Sum                = %f\n", sum);
    printf("Average            = %f\n", average);
}
```

Output

```
This program computes the average of a set of numbers
Enter values one after another
Enter a NEGATIVE number at the end.
21 23 24 22 26 22 -1
Number of values   = 6
Sum                = 138.000000
Average            = 23.000000
```

Fig. 6.11 *Use of* **break** *in a program*

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

Program 6.9 A program to evaluate the series.

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

for $-1 < x < 1$ with 0.01 per cent accuracy is given in Fig. 6.12. The **goto** statement is used to exit the loop on achieving the desired accuracy.

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

Program

```
#define    LOOP        100
#define    ACCURACY    0.0001
main()
{
     int n;
     float x, term, sum;
     printf("Input value of x : ");
     scanf("%f", &x);
     sum = 0 ;
     for (term = 1, n = 1 ; n <= LOOP ; ++n)
     {
          sum += term ;
          if (term <= ACCURACY)
               goto output; /* EXIT FROM THE LOOP */
       term *= x ;
     }
     printf("\nFINAL VALUE OF N IS NOT SUFFICIENT\n");
     printf("TO ACHIEVE DESIRED ACCURACY\n");
     goto end;
     output:
     printf("\nEXIT FROM LOOP\n");
     printf("Sum = %f; No.of terms = %d\n", sum, n);
     end:
     ;          /* Null Statement */
}
```

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

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

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

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

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

Program

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

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

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

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

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

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

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

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

Output

```
ENTER 10 REAL NUMBERS

1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9 10.10

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

Total = 446.86
```

Program 7.7

The program in Fig. 7.8 shows how to multiply the elements of two N × N matrices.

Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
int a1[10][10],a2[10][10],c[10][10],i,j,k,a,b;
clrscr();
printf("Enter the size of the square matrix\n");
scanf ("%d", &a);
b=a;
printf("You have to enter the matrix elements in row-wise fashion\n");
for(i=0;i<a;i++)
{
for(j=0;j<b;j++)
{
printf("\nEnter the next element in the 1st matrix=");
scanf("%d",&a1[i][j]);
}
}
for(i=0;i<a;i++)
{
for(j=0;j<b;j++)
{
printf("\n\nEnter the next element in the 2nd matrix=");
scanf("%d",&a2[i][j]);
}
}
printf("\n\nEntered matrices are\n");
for(i=0;i<a;i++)
{           printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a1[i][j]);
}
printf("\n");
for(i=0;i<a;i++)
{           printf("\n");
for(j=0;j<b;j++)
printf(" %d ",a2[i][j]);
}
printf("\n\nProduct of the two matrices is\n");
for(i=0;i<a;i++)
  for(j=0;j<b;j++)
  {
  c[i][j]=0;
  for(k=0;k<a;k++)
  c[i][j]=c[i][j]+a1[i][k]*a2[k][j];
```

```
}
for(i=0;i<a;i++)
{           printf("\n");
for(j=0;j<b;j++)
printf(" %d ",c[i][j]);
}
getch();
}
```

Output

```
Enter the size of the square matrix
2
You have to enter the matrix elements in row-wise fashion
Enter the next element in the 1st matrix=1
Enter the next element in the 1st matrix=0
Enter the next element in the 1st matrix=2
Enter the next element in the 1st matrix=3
Enter the next element in the 2nd matrix=4
Enter the next element in the 2nd matrix=5
Enter the next element in the 2nd matrix=0
Enter the next element in the 2nd matrix=2
Entered matrices are
  1  0
  2  3
  4  5
  0  2
Product of the two matrices is
4  5
8  16
```

Fig. 7.8 *Program for N × N matrix multiplication*

Program 7.9 The program in Fig. 7.9 shows how to find the transpose of a matrix.