```
/* Example Program 9.3 */
/* Program to convert Celsius to Fahrenheit
   - Illustrating while loop */
#include <stdio.h>

main()
{
  int celsius;
  float fahrenheit;
  scanf("%d", &celsius);
  printf("Celsius    Fahrenheit\n");

  while (celsius <= 100)
    {
      fahrenheit = 1.8 * (float)celsius + 32.0;
      printf("%5d    %10.3f\n", celsius, fahrenheit);
      ++celsius;
    }
  printf("End of conversion\n");
}
```

Program 9.3 Celsius to Fahrenheit Conversion—use of *while* loop.

Observe in Example Program 9.3 that we declare Celsius as int as the values for which it is to be converted are integers. Fahrenheit, however, is float as it could have a fractional part. The function scanf reads the initial value of celsius (– 100 in this example). The title for the table is then printed. This is followed by the while loop. In this loop fahrenheit is calculated. Observe that celsius is cast into floating point. This is not essential in C language but is a good programming practice. After printing the converted values, celsius is incremented. The statements in the loop are repeatedly executed for celsius value upto and *including 100*. When celsius value reaches 101 the program leaves the loop and prints the line: End of conversion.

The program may be slightly generalized for tabulating all values of Fahrenheit for all values of Celsius from an initial to a final value as shown in Example Program 9.4.

Example 9.2

A procedure to find the average height of boys and girls in a class was given in Chapter 2. It is given again as Procedure 9.1 for ready reference.

Procedure 9.1: Procedure to find the average height of boys and girls

Step 1: Initialize counters to accumulate total number of girls and boys and their respective heights

Step 2: While input data are left repeat Step 3 and Step 4

Step 3: Read an input line with the data (Roll number, sex code, height)

```
/* Example Program 9.4 */
/* Program to convert Celsius to Fahrenheit
   for Celsius = initial to final */
#include <stdio.h>

main()
{
  int celsius, initial_celsius, final_celsius;
  float fahrenheit;
  scanf("%d %d", &initial_celsius, &final_celsius);
  printf("Celsius     Fáhrenheit\n");
  celsius = initial_celsius;
  while (celsius <= final_celsius)
    {
      fahrenheit = 1.8 * (float)celsius + 32.0;
      printf("%5d   %10.3f\n", celsius, fahrenheit);
      ++celsius;
    }
  printf("End of conversion\n");
}
```

Program 9.4 Celsius to Fahrenheit—variable range.

Step 4: *If* sex code = 1 then
Sum of boys height += height
++Total boys ;
else
Sum of girls height += height;
++Total girls;

Step 5: Average boy height = Sum of boys height/Total boys

Step 6: Average girl height = Sum of girls height/Total girls

Step 7: Print Total boys, Average boy height,
Total girls, Average girl height

The procedure is converted into a C program in Example Program 9.5. It is assumed that the data is presented in the following form:

Data

```
2345        1       115.5
2685        2       100.2
2742        1       100.8
2743        2       102.4
End of data set
```

The program should be general to accommodate any class size as we cannot assume a fixed number of students in a class. We should thus repeat computing total height of boys and girls as long as data remain to be read. The standard i/o library <stdio.h> in which scanf is defined provides a convenient way of detecting whether any data is left in the input data set

```
/* Example Program 9.5 */
/* Program to find average height of girls and boys
   in a class.  When no more data is left in input
   a value called EOF (end of file) is returned by
   the scanf function defined in <stdio.h> */
#include <stdio.h>
main()
{
  int sexcode, total_girls, total_boys, roll_number;
  float height, total_girl_height, total_boy_height,
        av_girl_height, av_boy_height;

  total_girl_height = 0; total_boy_height = 0;
  total_girls = 0; total_boys  = 0;

  while (scanf("%d %d %f", &roll_number, &sexcode, &height)
         != EOF)
    {
      if (sexcode == 1)
        {
          total_boy_height += height;
          ++total_boys;
        }
      else
        {
          total_girl_height += height;
          ++total_girls;
        }
    } /* End of while */

  av_boy_height = total_boy_height / (float)total_boys;
  av_girl_height = total_girl_height / (float)total_girls;
  printf("total_boys = %d, av_boy_height = %f\n",
          total_boys, av_boy_height);
  printf("total_girls = %d, av_girl_height = %f\n",
          total_girls, av_girl_height);
} /* End of main */
```

Program 9.5 Average height of girls and boys.

to be read. When scanf function reaches the end of data a quantity defined as EOF (end of file) is returned to it.

In other words:

scanf("%d%d%f \n", &roll_number, &sex_code, &height)

equals EOF when end of data is reached. Thus we can check whether scanf ( ) = EOF after reading each data line. If no data is found when scanf attempts to read data then EOF is returned to scanf. This is used in the *while* loop in Example Program 9.5.

Example 9.3

A procedure was evolved in Chapter 1 to pick the maximum tender among a group of tenders. The procedure is refined and reproduced as Procedure 9.2.

Procedure 9.2: ***Procedure to pick maximum tender among a set of tenders***

Step 1: Read tender identity and tender amount

Step 2: max tender = tender amount;
max tender identity = tender identity;

Step 3: Perform Steps 4 and 5 as long as the end of the list of tenders is not reached.

Step 4: Read tender identity and tender amount

Step 5: If tender amount > max tender
then {max tender = tender amount
max tender identity = tender identity}

Step 6: Write max tender identity and max tender

Procedure 9.2 is converted into a C program and given as Example Program 9.6.

```
/* Example Program 9.6 */
/* Program picks the maximum tender among a set of
   tenders, The end of tenders is indicated by EOF
   returned to scanf when no data is left to read */
#include <stdio.h>

main()
{
  int tender_id, max_id;
  float tender_amount, max_tender = 0;

  while(scanf("%d %f", &tender_id, &tender_amount) != EOF)
    {
      if (tender_amount > max_tender)
        {
          max_tender = tender_amount;
          max_id = tender_id;
        }
    } /* End of while */
  printf("Maximum tender_id = %d\n", max_id);
  printf("Maximum amount = %f\n", max_tender);
} /* End of main */
```

Program 9.6 Picking maximum tender.

9.2 THE *for* LOOP

The *while* loop terminates when the logical expression controlling the loop becomes false. C provides another looping structure which is more general. This structure is realized by a statement called the *for* statement. The general form of *for* statement is:

```
for (expression 1 ; expression 2 ; expression 3)
        { s1 ;
          s2 ;
          .
          .
          sn ; }
sf ;
```

In the above statement *expression 1, expression 2* and *expression 3* are legal C expressions. s_1, s_2 ...s_n are statements in C. Usually *expression 2* is a logical expression. A flow chart illustrating the execution of the *for* statement is given in Fig. 9.3. Observe that *expression 1* is executed just before testing the logical expression. The *logical expression 2* is then tested. If it is *true* the statements $\{s_1; s_2 ...., s_n;\}$ in the loop are executed. After execution of these statements *expression 3* is evaluated. Control then returns to test logical *expression 2*. As long as logical *expression 2* is *true* the statements in the loop and *expression 3* are executed again and again. When *expression 2* becomes false control leaves the loop and goes to the statement following the *for* loop.

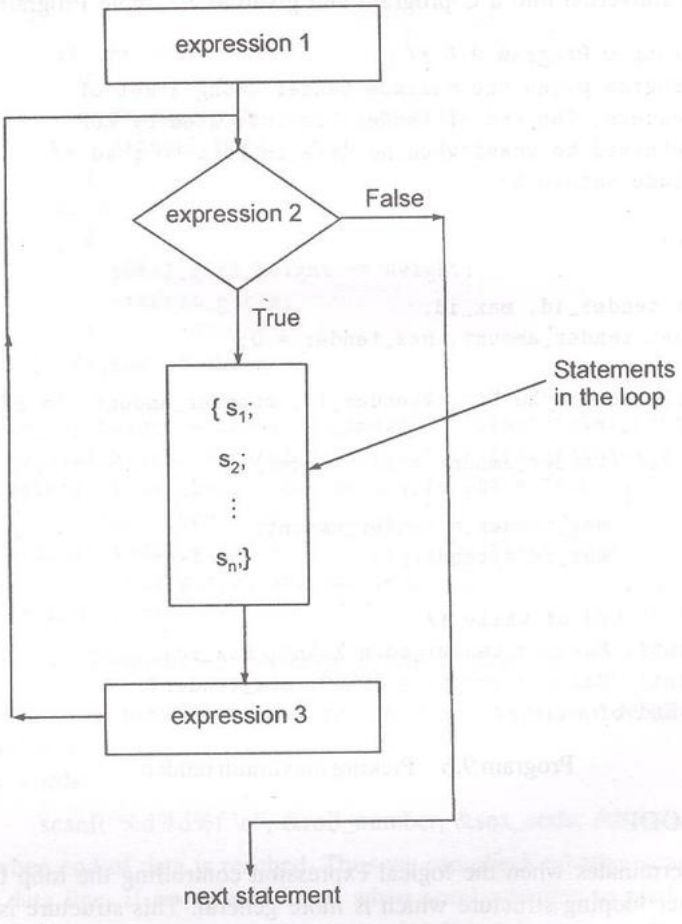

Fig. 9.3 Flow chart illustrating a *for* loop.

The syntax rules allow any one or more of these expressions to be absent. For example if we write:

for (; expression 2 ;)

it is the same as a *while* loop !
(Observe that the semicolons are essential)

Thus the *for* loop in C is very general. It is, however, most commonly used when the statements in a loop are to be executed a fixed number of times. For example, one common use is of the type

for (index = initial value ; index <= final value ; index = index + increment)

{ s_1 ; s_2 ; s_3 ... s_n ; }

one example of this is: for (i = 1 ; i <= n ; ++i)

{s_1 ; s_2 ; s_3 ; ... s_n ;}

This is illustrated in Fig. 9.4. We will now give some examples of use of the *for* statement.

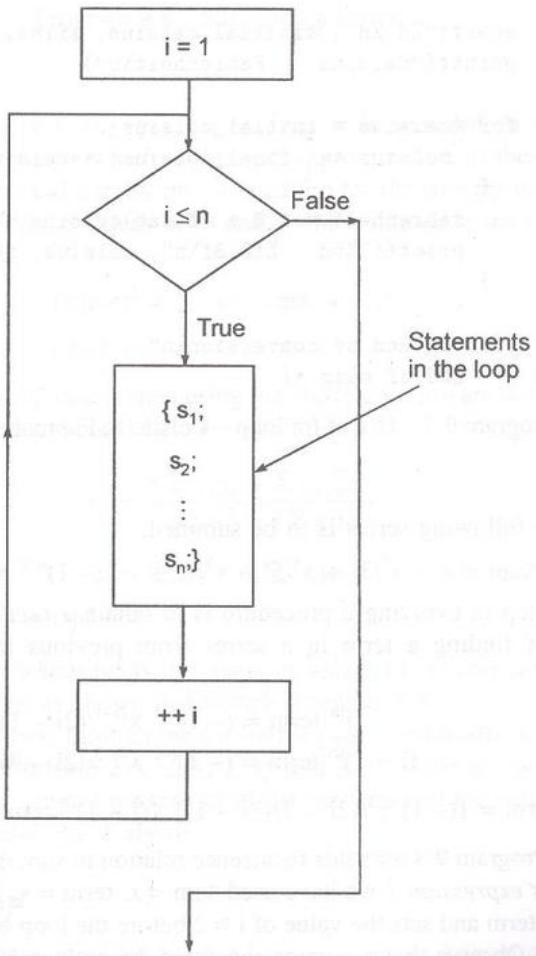

Fig. 9.4 Flow chart illustrating the most common use of a *for* loop.

Example 9.3 was written to convert Celsius to Fahrenheit and tabulate it for Celcius temperatures of – 100 to + 100 in steps of 1 degree Celsius. Example Program 9.4 was written using a *while* statement to do this. We write Example Program 9.7 using a *for* statement to do the same job.

Compare Example Programs 9.4 and 9.7. It is quite obvious that the *for* statement is a more concise notation.

```
/* Example Program 9.7 */
/* This program is derived from Example Program 9.4
   It uses a for statement */
#include <stdio.h>

main()
{
  int celsius, initial_celsius, final_celsius;
  float fahrenheit;

  scanf("%d %d", &initial_celsius, &final_celsius);
  printf("Celsius    Fahrenheit\n");

  for (celsius = initial_celsius;
       celsius <=  final_celsius; ++celsius)
    {
      fahrenheit = 1.8 * (float)celsius + 32.0;
      printf("%5d    %10.3f\n", celsius, fahrenheit);
    }

  printf("End of conversion\n");
} /*  End of main */
```

Program 9.7 Use of for loop—Celsius to Fahrenheit conversion.

Example 9.4

Assume that the following series is to be summed:

$$\text{Sum} = x - x^3/3! + x^5/5! - x^7/7! + \ldots (-1)^{n-1}\, x^{2n-1}/(2n-1)!$$

The first step in evolving a procedure is to obtain a *recurrence relation* which gives the technique of finding a term in a series from previous terms. By inspection of the series:

$$i^{th} \text{ term} = (-1)^{i-1}\, x^{2i-1}/(2i-1)\,!$$

$$(i-1)^{th} \text{ term} = (-1)^{i-2}\, x^{2i-3}/(2i-3)!$$

Thus i^{th} term = $\{(-1)\, x^2/(2i-2)(2i-1)\}$ *$(i-1)^{th}$ term

Example Program 9.8 uses this recurrence relation to sum the series. Observe that in the *for* statement for *expression 1* we have used sum = x, term = x, i = 2. This assigns the value of x to sum and term and sets the value of i = 2 before the loop begins. This concise notation is allowed in C. Observe that a comma separates the assignments.

```
/* Example Program 9.8 */
/* This program illustrates use of a for statement */
#include <stdio.h>

main()
{
  int i, n, denominator;
  float x, sum, term;
  scanf("%f %d", &x, &n);
  for ( sum = x, term = x, i = 2 ; i <= n; ++i)
    {
      denominator = (2*i-2) * (2*i-1);
      term *= (-x * x) / (float)denominator;
      sum += term;
    }
  printf("sum = %f, x = %f, n = %d\n", sum, x, n);
} /* End of main */
```

Program 9.8 Summing a series.

Example 9.5

Given a set of points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) it is required to fit a straight line $y = mx + c$ through these points which is the best approximation to these points. In other words, optimal values for m and c in the above equation for the straight line are to be found. A popular criterion is to find the values of m and c which minimize the sum of the squares of the error as given below:

$$(\text{Error})^2 = \Sigma\, [y_i - (mx_i + c)]^2$$

Σ is summation for i = 1 to n.

The values of 'm' and 'c' determined using the above criterion are derived in elementary books in statistics and are reproduced below:

$$m = \frac{n \Sigma x_i y_i - \Sigma x_i \Sigma y_i}{n \Sigma x_i^2 - (\Sigma x_i)^2}$$

$$c = [\Sigma y_i - m \Sigma x_i]/n$$

Σ is summation for i = 1 to n.

A computer program which reads in n pairs of values (x, y) and computes m and c is easily written with *for* loops as shown in Example Program 9.9.

Observe that in each pass through the *for* loop a pair of values for x and y is read. The values of x and y are used to form Σx, Σy, Σxy and Σx^2. After all the n values are read and the totals accumulated, control passes out of the *for loop* and the values of m and c are calculated using the formulae given above.

9.3 THE *do while* LOOP

The C programming language provides another looping structure known as *do while*. In this

```
/* Example Program 9.9 */
/* This program fits a straight line through n
   pairs of x and y coordinates. The straight line is
   y = mx + c  */
#include <stdio.h>

main()
{
  float sum_x = 0, sum_y = 0, sum_xy = 0, sum_xsq = 0, x, y,
  numerator, denominator, m, c;
  int i, n;

  scanf("%d", &n);  /* number of points read */
  for (i = 1; i <= n; ++i)
    {
      scanf("%f %f", &x, &y);
      sum_x += x;
      sum_y += y;
      sum_xy += x*y;
      sum_xsq += x*x;
    }
  numerator = (float)n * sum_xy - sum_x * sum_y;
  denominator = (float)n * sum_xsq - sum_x * sum_x;
  m =  numerator / denominator;
  c = (sum_y - m * sum_x) / (float)n;
  printf("slope m = %f, intercept c = %f\n", m, c);
  printf("y = %f * x  +  %f\n", m, c);
} /* End of main */
```

Program 9.9 Fitting a straight line through points.

structure the statements in the loop are executed and the testing is done at the end of the loop. The general form of this statement is:

do

{ s_1 ;
  s_2 ;
  .
  .
  .
  s_n ; }

while (logical expression) ;

The statement commands that the compound statement { s_1 ; s_2 ; .... s_n ; } following *do* should be executed repeatedly as long as the logical expression is *true*. When the logical expression becomes *false* the program leaves the loop and goes to the first statement following *while* (logical expression). The looping structure is illustrated in Fig. 9.5. This statement is not commonly used in C programs. There are some situations where it is useful. We give an example of the use of this statement.

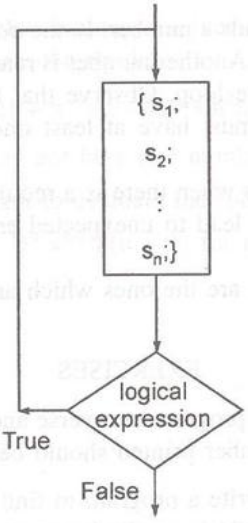

Fig. 9.5 Flow chart illustrating a *do while* loop.

Example 9.6

A set of integers is given. It is required to count the number of positive integers, negative integers and zeros in this set. Example Program 9.10 has been written to do this. In this

```
/* Example Program 9.10 */
/* Program to count the number of negative, zero
   and positive integers */
#include <stdio.h>

main()
{
  int number, no_positive = 0, no_zero = 0, no_negative = 0;
  scanf("%d", &number);
  do
    {
      if (number > 0)
        ++no_positive;
      else
        if (number == 0)
          ++no_zero;
        else
          ++no_negative;
    }
  while( scanf("%d", &number) != EOF );
  printf("no. of positive numbers = %d\n", no_positive);
  printf("no. of negative numbers = %d\n", no_negative);
  printf("no. of zeros = %d\n", no_zero);
}
```

Program 9.10 Counting negative, zero and positive integers.

program scanf("%d", &number) reads a number. In the *do while* loop its sign is checked and appropriate counter is incremented. Another number is read within *while* statement. If the end of data is sensed control leaves the loop. Observe that the statements in the loop will be executed at least once. Thus we must have at least one number in the input set for the program to work correctly.

There are not many situations when there is a requirement of the loop being executed at least once. Sometimes this may lead to unexpected error if the programmer is careless. Thus it is used rarely.

The *while* and *for* statements are the ones which are normally used.

EXERCISES

9.1 Given an integer, write a program to reverse and print it. For example if the given number is 12386 the number printed should be 68321.

9.2 Given a set of integers write a program to find those which are palindromes. For example the number 123321 is a palindrome as it reads the same from left to right and from right to left.

9.3 Given an octal number (a number with base 8) of arbitrary length write a program to find its decimal equivalent. For example the decimal equivalent of the octal number 2673 is

$$2 * 8^3 + 6 * 8^2 + 7 * 8^1 + 3 * 8^0 = 1467.$$

9.4 Given any decimal number write a program to find its octal equivalent. For example, the octal equivalent of 242 is 362.

9.5 Given an angle x in radians write a program to find

$$x \ \% \ (\pi/2)$$

Remember that % is a built-in operator valid only for integers.

9.6 Given values for a, b, c and d and a set of values for the variable x, evaluate the function defined by

$$\begin{aligned} f(x) &= ax^2 + bx + c \quad \text{if } x < d \\ f(x) &= 0 \quad \text{if } x = d \\ f(x) &= -ax^2 + bx - c \quad \text{if } x > d \end{aligned}$$

for each value of x, and print the value of x and f(x). Write a program for an arbitrary number of x values.

9.7 A machine is purchased which will produce earnings of Rs. 1000 per year while it lasts. The machine costs Rs. 6000 and will have a salvage value of Rs. 2000 when it is condemned. If 12 percent per annum can be earned on alternative investments what should be the minimum life of the machine to make it a more attractive investment compared to alternative investments?

9.8 The interest charged in instalments buying is to be calculated by a computer program. A tape recorder costs Rs. 2000. A shopkeeper sells it for Rs. 100 down and Rs. 100 for 21 more months. What is the monthly interest charged?

9.9 Write a program which will evaluate the function for the set of values of x (.5, 1., 1.5, 2, 2.5, 3) and tabulate the results.

$$f = 1 + x^2/2! + x^4/4! - .50 \sin^2 x + (4 - x^2)^{1/2}$$

Caution: The answer may not be a real number.

9.10 Write a computer program to evaluate the following sum:

$$S = \sum(-1)^n x^{n/2}/n(n+1) \text{ for } n = 1 \text{ to } 10$$

10. Defining and Manipulating Arrays

Learning Objectives

In this chapter we will learn:

1. The use of a data structure called an array which may have one or more dimensions
2. How an array is declared
3. How data is read into an array
4. How data stored in arrays is processed

10.1 ARRAY VARIABLE

The variables considered so far were single entities, that is, each variable name stores one number. There is another type of variable called an *array* variable. An *array* variable name refers to a group of quantities by a single name. Each member in the group is referred to by its position in the group. Suppose a shop stocks different wattages of bulbs and the stock of each wattage bulbs is to be represented. One way of representing this is to make a table:

Wattage	15 watts	25 watts	40 watts	60 watts	100 watts
Stock	25	126	300	570	28

We can codify each wattage by a digit and represent the above table as:

Wattage code:	0	1	2	3	4
Stock:	25	126	300	570	28

where 0 represents 15 watts, 1:25 watts, 2:40 watts, 3:60 watts and 4:100 watts.

We can now represent the stock of bulbs by:

stock = {25 126 300 570 28}

with the wattage codes not being mentioned explicitly but implied. Stock is an array variable with 5 components.

A particular component of the array is referenced by its position in the array, starting with 0^{th} position.

Thus
stock [0] = 25
stock [1] = 126
stock [2] = 300
stock [3] = 570
stock [4] = 28

In memory one location is reserved for each component of the array. The locations are contiguous:

Variable name	stock[0]	stock[1]	stock[2]	stock[3]	stock[4]
Contents	25	126	300	570	28

Assume that the cost of each wattage bulb is represented by an array *cost* as shown below:

cost[0] = 3.50 ; cost[1] = 6.50 ; cost[2] = 7.75 ; cost[3] = 8.50 ; cost[4] = 9.50

We will now write a program to find the total cost of bulbs stored in the shop. This is shown in Example Program 10.1. We will now explain this program. The declaration, int stock [5] declares that stock is an array with components 0, 1, 2, 3, 4 (i.e., 5 components) and its components store integers. In the next declaration, cost is declared as an array with 5 components where each component is a floating point number. In the *for loop* scanf function reads an integer and a floating point number from input and assigns them to stock [0] and cost [0] respectively. In the next statement the product of stock and cost is added to total_cost. The loop is repeated with watt_code = 1, 2, 3, 4 and the total_cost is computed.

```
/*  Example Program 10.1 */
/*  Calculation of total cost of bulbs.
    Illustrating use of arrays */
#include <stdio.h>

main()
{
  int stock[5], watt_code;
  float cost[5], total_cost = 0;

  for(watt_code = 0; watt_code <= 4; ++watt_code)
    {
      scanf("%d %f", &stock[watt_code], &cost[watt_code]);
      total_cost += (float)stock[watt_code] * cost[watt_code];
    }
  printf("%f\n", total_cost);
}
```

Program 10.1 Array declaration and use.

Finally the total_cost is printed. We now consider another example.

Example 10.1

There are fifty students in a class and the marks obtained by each student in an examination are typed on one or more lines. A computer program is required which will print out the highest and the second highest marks obtained in the examination. This problem is a variation of the problem of picking the largest from a set of numbers. In this problem, after picking the largest number in the set, it should be eliminated from further consideration and the remaining numbers searched for the largest. A program to do this is given as Example Program 10.2. The new statements used in this program will be explained below:

The declaration int marks [50] provides information to the compiler that marks is an array with 50 components. Thus 50 locations in memory would be reserved for marks and its components stored in contiguous locations. The *for* loop:

```
for (i = 0 ; i <= 49, ++i)
    scanf("%d", &marks[i]);
```

```
/*  Example Program 10.2 */
/* This program uses arrays. Remember that the
   numbering of elements in an array start from 0 */
#include <stdio.h>

main()
{
  int marks[50], i, first = 0, second = 0;
  for (i = 0; i <=49; ++i)
    scanf("%d", &marks[i]);
  for (i = 0; i <=49; ++i)
    if (marks[i] > first)
      first = marks[i];
  for (i = 0; i <= 49; ++i)
    if (marks[i] != first)
      if (marks[i] > second)
        second = marks[i];
  printf("First Marks = %4d,  Second Marks = %4d\n",
         first, second);
}
```

Program 10.2 Finding first and second highest marks.

reads 50 data from one or more lines. The first number is assigned to marks [0], the second number to marks [1], and the successive ones to succeeding components of marks. The next *for loop* picks the highest component of marks and stores it in first. In the third *for loop* all the components of marks equal to first are skipped and second highest component picked from among the rest. Instead of using two *for* loops as shown in this solution it is possible to write a program with a single *for loop*. It is also possible to solve this problem without using an array. The reader should try to solve this problem both ways and he would then appreciate the flexibility and power afforded by subscripts or array index.

10.2 SYNTAX RULES FOR ARRAYS

The general form of an array variable is:

v[i]

where v is a variable name which may be of type integer or real and *i* is a subscript or an array index. The subscript may be any valid integer constant, integer variable name or integer expression. Subscripts may be negative, zero, or positive. Negative subscripts are allowed provided it leads to a meaningful value within the bounds of the array.

Example

i, j, code, *i * 4 – k, 2 * i/p* are valid subscripts provided *i, j, code, k, p* are integer variable names. The following are invalid.

a[2i], c[2.6], x[i = p/4], p(i)

The following are valid:

a[2*i], b[++i], c[9/2], d[– –k]

A declaration of a variable name as representing an array provides information to the C compiler to enable it to:

(i) identify the variable name as an array name

(ii) reserve locations in memory to store all the components of an array.

Observe that C assumes that the subscripts evaluated during computation will never exceed the size declared in a declaration statement. If it does exceed, wrong answers may be printed. Thus it is a programmer's responsibility to see that array bounds are not violated.

Example 10.2

We will now write a program which interchanges the odd and even components of an array. The program is given as Example Program 10.3.

```
/*  Example Program 10.3 */
/*  This program interchanges the components of
    a vector */
#include <stdio.h>

main()
{
  int a[10], i, k, temp;
  for (i = 0; i <= 9; ++i)
    scanf("%d", &a[i]);
  for (k = 0; k <= 8; k+=2)
    {
      temp = a[k];
      a[k] = a[k+1];
      a[k+1] = temp;
    }
  for (i = 0; i <= 9; ++i)
    printf("%d  ", a[i]);
  printf("\n");
} /* End of main */
```

Program 10.3 Interchanging components of a vector.

Example 10.3

We will now write a program to evaluate a polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}\, x^{n-2} + \ldots\ldots + a_1 x + a_0$$

Evaluating this polynomial by writing an arithmetic statement is simple but inefficient. It will require n(n+1)/2 multiplication operations and n addition operations. We will use an alternative method of writing the polynomial and write a program to evaluate the polynomial. This method requires only n multiplications and n additions.

$$p(x) = a_0 + a_1 x + a_2\, x^2 + \ldots\ldots + a_{n-1}\, x^{n-1} + a_n x_n$$

$$= a_0 + x(a_1 + x(a_2 + x(a_3 + x(a_4 + \ldots\, x(a_{n-1} + xa_n))))).$$

We start evaluating the expression in the innermost parentheses and successively multiply by x in a *for* loop. The coefficients $a_0, a_1, a_2 \ldots a_n$ are stored in an array a[n]. The program is written as Example Program 10.4.

```
/* Example Program 10.4 */
/* This program evaluates a polynomial upto order 20 */
#include <stdio.h>

main()
{
  int i, n;
  float x, a[20], polynomial;
  scanf("%d %f", &n, &x);
  /* n is the order of the polynomial to be
     evaluated and is read as input. n should
     be less than or equal to 20 */
  for (i = 0; i <= n; ++i)
    scanf("%f", &a[i]);
  /* a[0], a[1], a[2],., a[n] typed in one or more lines */
  polynomial = a[n];
  for (i = n; i >= 1; --i)
    polynomial = a[i-1] + x * polynomial;
  printf("x = %f, polynomial value = %f\n", x, polynomial);
}
```

Program 10.4 Evaluating a polynomial.

In Example Program 10.4 we assume that the maximum number of polynomial coefficients is 20. Thus the size of the array *a* storing the coefficients is declared a[20]. As it is desirable to write a general program to evaluate polynomials it would have been nice to declare the array as a[n] with *n* being a variable. C language, however, does not allow using a variable as array size. We thus use 20 as array size and have to state that only polynomial upto order 20 may be evaluated with this declaration. If the order of polynomial is less than 20 then this program will work. We now explain the program. The first *for* loop reads all values of the coefficients of the polynomial into the array *a*. The second *for* loop is traced in Table 10.1.

Table 10.1 Tracing Loop in Example Program 10.4

Index *i*	Polynomial
Initialisation	a[n]
n	a[n – 1] + x * a[n]
n – 1	a[n – 2] + x * (a[n – 1] + x * a[n])
n – 2	a[n – 3] + x * (a[n – 2] + x * (a[n – 1] + x * a[n]))
.	
.	
.	
1	a[0] + x * (a[1] + x *(a[2] + x*(a[3] + ...)))

Observe that the index starts with *n* and is decremented at the end of each execution of the statement:

```
polynomial = a[ i – 1] + x * polynomial ;
```

which is the only statement in the *for* loop.

10.3 USE OF MULTIPLE SUBSCRIPTS IN ARRAYS

Example 10.4

We introduced array variable name in the beginning of this chapter by representing stock of bulbs of different wattages in a store by using a code for wattage. The stock was represented by

```
stock = [25 126 300 570 28]
```

where the first element represents stock of 15 watt bulbs, second element the stock of 25 watt bulbs etc.

Assume that the store stock several brands of bulbs and that each brand is coded by a unique integer. For example GEC = 0, Philips = 1, Crompton = 2, Surya = 3, Mysore = 4 and Bajaj = 5. The number of bulbs of each category in stock is shown as a matrix in Table 10.2.

Table 10.2 Table Showing Stock Position of Bulbs

		Watts				
		15 (0)	25 (1)	40 (2)	60 (3)	100 (4)
GEC	(0)	20	15	0	25	60
Philips	(1)	0	22	34	62	0
Crompton	(2)	10	0	25	14	18
Surya	(3)	28	32	0	48	60
Mysore	(4)	43	25	25	34	68
Bajaj	(5)	22	30	41	0	25

The stock of bulbs in each category in the store may be represented by an array variable with two subscripts:

```
stock[brand-code][watt-code]
```

where brand-code and watt-code are integers. If we write stock [4][3] it means stock of bulbs with brand-code = 4 and watt-code = 3, that is, stock of Mysore brand bulbs of wattage 60. This stock is 34 in Table 10.2. As new stock of bulbs are received by the store, Example Program 10.5 illustrates how it would add this value to the appropriate component of stock [brand_code][watt_code].

Data entered on the keyboard gives the brand_code, watt_code and the number of bulbs. In Example Program 10.5 if the first data indicates that 10 Surya brand 100 watts bulb has

```
/*  Example Program 10.5 */
/* The following small program illustrates how the number of
   bulbs in stock is updated when new stock is received */
#include <stdio.h>

main()
{
  int stock[6][5], brand_code, watt_code, number;
  while(scanf("%d %d %d", &brand_code, &watt_code, &number)
        != EOF)
    stock[brand_code][watt_code] += number;
} /* End of main */
```

Program 10.5 Updating contents of array.

been received, the statement:

stock[brand_code][watt_code] += number ;

would become

stock [3][4] = stock [3][4] + 10 = 60 + 10 = 70

as brand_code = 3, watt_code = 4 and number = 10. Observe the declaration:

int stock [6][5]

This declaration gives information to the compiler that stock is an integer array name, has two indices or subscripts and the maximum storage required is 5 elements in each row and there are 6 such rows.

10.4 READING AND WRITING MULTIDIMENSIONAL ARRAYS

At the beginning of this chapter we used the following statement:

```
for( i = 0; i <= 49; ++i)
  scanf("%d", &marks [i]) ;
```

This statement reads 50 values typed in one or more lines and assigns them respectively to marks [0], marks [1], ... marks [49]. In other words, the first value from the input line will be assigned to marks [0], the second to marks [1] etc. Data will be read and assigned to *all* 50 components of marks.

More than one array may be read using a scanf statement in a *for* loop. For example, the statement

```
for(i = 0 ; i <= 10 ; ++i)
  scanf("%d", &a[i], &b[i]) ;
```

reads numbers from the input and assigns the first number to a[0], the second number to b[0], the third number to a[1], the fourth to b[1] and so on till the last number is assigned to b[10].

The array declaration in this case must be

int a[11], b[11]

Observe that the array size of *a* and *b* are 11 as the subscript starts with 0.

We can also initialise one or more arrays in a declaration. For example,

```
int a[6] = {25 – 40 0 54 90 100};
```

will store 25 in a[0], – 40 in a[1], 0 in a[2], 54 in a[3], 90 in a[4] and 100 in a[5].

If we declare

```
int a[6] = {10};
```

then 10 will be stored in all the six array elements.

A two dimensional array is also known as a matrix. In mathematics a matrix is written as:

column ↓

$$\text{mat} = \begin{bmatrix} 6 & -2 & 3 \\ -2 & 3 & 0 \\ 6 & 2 & -9 \\ 5 & 8 & 11 \end{bmatrix} \leftarrow \text{row}$$

This is a (4 × 3) matrix that is, it has 4 rows and 3 columns. If we write mat(2, 3) it means an element in the second row, third column. In this matrix mat(2, 3) = 0. In general an element in the i^{th} row j^{th} column is written as mat(i, j).

The row index *i* and the column index *j* both start with 1. Thus mat(1, 1) = 6 in the above matrix. In contrast C language assumes that indices start with 0. If we want to maintain the mathematical notation in writing C programs with matrices it is better not to use the zero index for row and column. The above matrix will then be declared as

```
int mat[5] [4] ;
```

This declaration will reserve 5 × 4 locations in memory (as indices are assumed as 0 to 4 for row and 0 to 3 for column, that is, 5 rows and 4 columns). We will not use the 0th row and 0th column in the program. Even though this wastes memory space, quite often it leads to less confusion in writing programs involving matrices.

We will now show how values are read from input and assigned to the matrix elements. Example Program 10.6 illustrates this.

In this program two *for* statements follow one another and are said to be *nested*. The execution of the *for* statements is explained below using the flow chart of Fig. 10.1.

In this program when the first *for* statement is encountered *i* is set to 1 and as *i* <= 4, the next *for* statement is taken up for execution. The second *for* statement sets *j* = 1 and as *j* <= 3 it executes the following statement which is scanf.

The scanf statement commands that a number be read from the input and assigned to mat[1][1]. After doing this *j* is incremented and scanf statement is executed with *j* = 2 which reads a number from the input and assigns to mat[1][2]. The next number from the input is assigned to mat[1][3]. As *j* = 3 now control returns to the first *for* statement. As *i* <= 4 this statement increments *i* which now becomes 2. The second *for* statement is now executed with *j* = 1, 2 and 3 and mat[2][1], mat [2][2] and mat [2][3] are assigned values from the input. Thus input data values should be typed in the order:

```
mat[1][1], mat[1][2], mat[1][3]
mat[2][1], mat[2][2], mat[2][3]
mat[3][1], mat[3][2], mat[3][3]
mat[4][1], mat[4][2], mat[4][3]
```

```
/*  Example Program 10.6 */
/*  This program reads a martix and stores in mat.
    mat[i][j] is the ith row and jth column of mat.
    Row and columns start with index 1. mat[1][1]
    is the element in the first row first column of mat */
#include <stdio.h>

main()
{
  int mat[5][4], i, j;
  /* The two for statements read the matrix row-wise */
  for (i = 1; i <= 4; ++i)
    for (j = 1; j <= 3; ++j)
      scanf("%d", &mat[i][j]);
  /* The matrix is printed row-wise with three numbers in
     each row */
  printf("Stored matrix is printed below\n");
  j = 1;
  for (i = 1; i <= 4; ++i)
    printf("%5d %5d %5d\n",
           mat[i][j], mat[i][j+1], mat[i][j+2]);
} /* End of main */

/* Data read one number from each line */
/* Numbers fed from the keyboard */
```

Program 10.6 Reading and printing a matrix.

Observe that the array is read row-wise by the above *for* loops. The inner *for* loop is executed 3 times for each value of *i*. Thus the inner *for* loop executes scanf, twelve times and reads 12 elements of the array. After the data is read into the matrix it is printed row-wise, 3 elements per row, in the next part of the program. Observe that in this part we have used only one *for* loop to increment the row index. The column element indices are calculated within the printf statement.

A scanf statement within two nested *for* loops may also be used to read several two dimensional arrays provided they have the same dimensions. For example the statements:

```
                                int a[5][3], b[5][3] ;
for (j = 1 ; j <= 2 ; ++j)
  for ( i = 1; i <= 4 ; ++i)
     scanf("%d %d", &a[i][j], &b[i][j]) ;
```

will read 16 integers from input and assign them respectively to:

a[1][1], b[1][1], a[2][1], b[2][1], a[3][1], b[3][1],
a[4][1], b[4][1], a[1][2], b[1][2], a[2][2], b[2][2],
a[3][2], b[3][2], a[4][2], b[4][2]

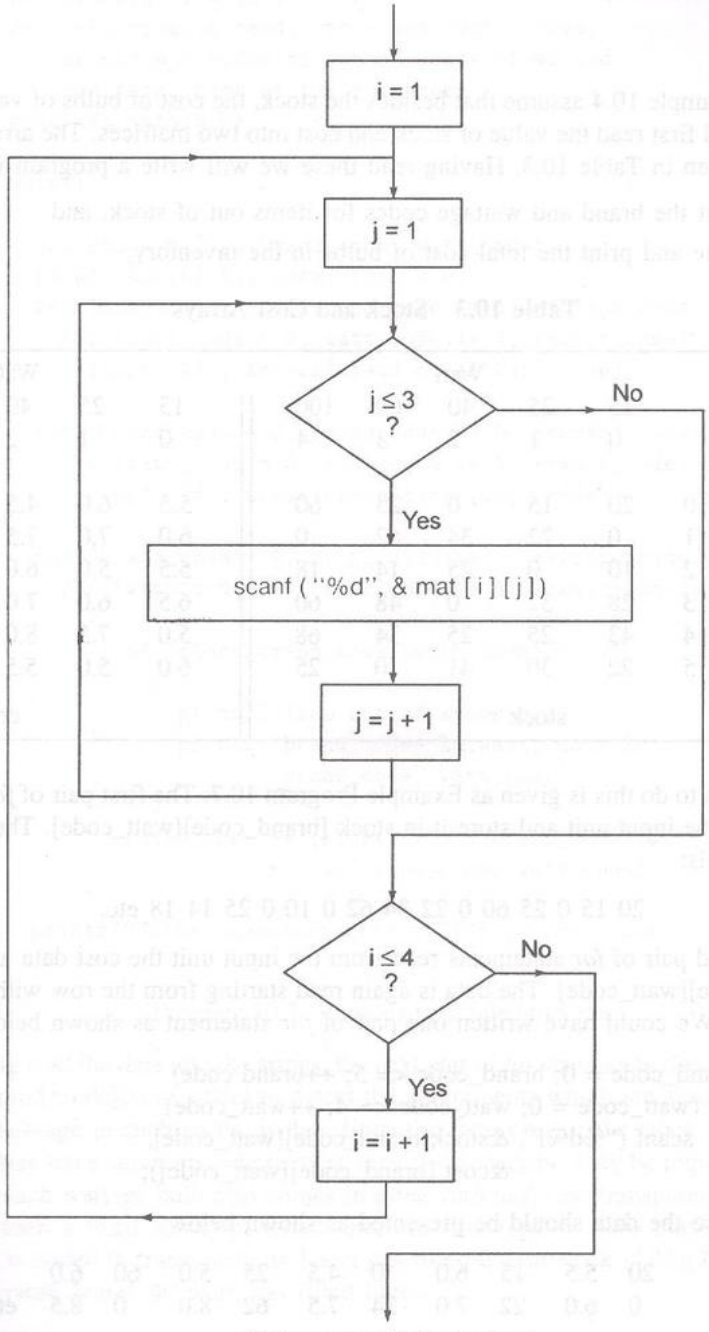

Fig. 10.1 Reading a matrix row-wise.

10.5 EXAMPLES OF *for* LOOP WITH ARRAYS

Example 10.5

Returning to Example 10.4 assume that besides the stock, the cost of bulbs of various brands is given. We will first read the value of stock and cost into two matrices. The arrays for stock and cost are given in Table 10.3. Having read these we will write a program which will

1. Print out the brand and wattage codes for items out of stock, and
2. Calculate and print the total cost of bulbs in the inventory.

Table 10.3 Stock and Cost Arrays

		Watt					Watt				
		15	25	40	60	100	15	25	40	60	100
		0	1	2	3	4	0	1	2	3	4
Brand											
GEC	0	20	15	0	25	60	5.5	6.0	4.5	5.0	6.0
Philips	1	0	22	34	62	0	6.0	7.0	7.5	8.0	8.5
Crompton	2	10	0	25	14	18	5.5	5.0	6.0	6.5	7.0
Surya	3	28	32	0	48	60	6.5	6.0	7.0	7.5	8.0
Mysore	4	43	25	25	34	68	5.0	7.5	8.0	7.5	8.0
Bajaj	5	22	30	41	0	25	6.0	5.0	5.5	6.5	7.5
		stock					cost				

A program to do this is given as Example Program 10.7. The first pair of *for* statements read data from the input unit and store it in stock [brand_code][watt_code]. The way data is to be presented is:

20 15 0 25 60 0 22 34 62 0 10 0 25 14 18 etc.

The second pair of *for* statements read from the input unit the cost data and store it in cost [brand_code][watt_code]. The data is again read starting from the row with brand_code = 0 row_wise. We could have written one pair of *for* statement as shown below:

```
for (brand_code = 0; brand_code <= 5; ++brand code)
    for (watt_code = 0; watt_code <= 4; ++watt_code)
        scanf ("%d%f", &stock[brand_code][watt_code],
                       &cost [brand_code][watt_code]);
```

In this case the data should be presented as shown below:

```
20  5.5  15  6.0   0  4.5  25  5.0  60  6.0
 0  6.0  22  7.0  34  7.5  62  8.0   0  8.5   etc.
```

Observe that we have to give the stock data and cost data alternately. There is a possibility of making an error in input. Further, if either a stock data or cost data is to be altered it is difficult to do it if the data is typed as above. We thus preferred to use two pairs of *for* statements in Example Program 10.7.

```
/*  Example Program 10.7 */
/*  This program reads stock and cost arrays,
    prints out codes of out of stock items and
    the total cost of the inventory */
#include <stdio.h>

main()
{
  int stock[6][5], brand_code, watt_code;
  float cost[6][5], total_cost = 0;
  for (brand_code = 0; brand_code <= 5; ++brand_code)
    for (watt_code = 0; watt_code <= 4; ++watt_code)
      scanf("%d", &stock[brand_code][watt_code]);

  for (brand_code = 0; brand_code <= 5; ++brand_code)
    for (watt_code = 0; watt_code <= 4; ++watt_code)
      scanf("%f", &cost[brand_code][watt_code]);

  for (brand_code = 0; brand_code <= 5; ++brand_code)
    for (watt_code = 0; watt_code <= 4; ++watt_code)
      {
        if (stock[brand_code][watt_code] == 0)
          {
            printf("Item out of stock\n");
            printf("brand_code= %d,watt_code= %d\n",
                    brand_code, watt_code);
          }
        total_cost += (float)(stock[brand_code][watt_code]
                    * cost[brand_code][watt_code]);
      }
  printf("Total inventory cost = %f\n", total_cost);
}
```

Program 10.7 Finding total inventory cost.

Having read the data into the arrays, the next pair of *for* statements check each component of stock [brand_code][watt_code] to detect the components which are zero and give an "out of stock" message in such cases. It then finds the total cost of the stock.

One may have situations where more than two subscripts may be required. In the above example if each wattage bulb also comes in three varieties, say, transparent, translucent and gas filled, then a third subscript which indicates the type of bulb may be used. Thus if transparent is coded 0, translucent as 1 and gas filled 2 then stock [4][2][2] would mean the stock of Mysore brand, 40 watt, gas filled bulbs.

Example 10.6

A Fibonacci number is defined as follows:

Fibonacci [0] = 0
Fibonacci [1] = 1

Given the above two

Fibonacci[2] = Fibonacci[1] + Fibonacci [0]

and in general

Fibonacci[i] = Fibonacci[i – 1] + Fibonacci[i – 2]

Example Program 10.8 generates the first 12 Fibonacci numbers and prints them.

```
/* Example Program 10.8 */
/* Generation of Fibonacci numbers */
#include <stdio.h>

main()
{
  int Fibonacci[12], i;
  /* use definition to initialise */
  Fibonacci[0] = 0; Fibonacci[1] = 1;

  for (i = 2; i <= 11; ++i)
    Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2];

  for (i = 0; i <= 11; ++i)
    printf("%d\n", Fibonacci[i]);
}
```

Program 10.8 Generating Fibonacci numbers.

Example 10.7

Two matrices are given:

$$[a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$[b_{ij}] = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

It is required to multiply these matrices and get the product matrix c_{ij} which is defined as:

$$c_{ij} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{bmatrix}$$

In general, given a matrix $[a_{ij}]$ with I rows and J columns and a matrix $[b_{ij}]$ with J rows and K columns the product matrix has I rows and K columns. The two matrices $[a_{ij}]$ and $[b_{ij}]$ cannot be multiplied if the number of columns in the first matrix $[a_{ij}]$ does not equal the number of rows of the matrix $[b_{ij}]$.

Example Program 10.9 multiplies two matrices and prints the product matrix. In this program we have declared the matrices a, b and c to have 20 rows and 20 columns each. This

```
/*  Example Program 10.9 */
/*  Program to multiply two matrices. Maximum size
    of the two matrices is 20 x 20 each  */
#include <stdio.h>
main()
{
  int a[20][20], b[20][20], c[20][20], i, j, k,
      a_rows, a_cols, b_rows, b_cols, c_rows,
      c_cols;

  scanf("%d %d %d %d", &a_rows, &a_cols, &b_rows, &b_cols);
  printf("%d %d %d %d\n", a_rows, a_cols, b_rows, b_cols);
  /* Matrices a and b are read column-wise */

  for (j = 1; j <= a_cols; ++j)
    for (i = 1; i <= a_rows; ++i)
      scanf("%d", &a[i][j]);

  for (j = 1; j <= b_cols; ++j)
    for (i = 1; i <= b_rows; ++i)
      scanf("%d", &b[i][j]);

  c_rows = a_rows; c_cols = b_cols;
  /*  c[20][20] assumed to be initialized to 0 */

  /* Multiplication performed by following 3 loops */
  for (k = 1; k <= c_rows; ++k)
    for (i = 1; i <= b_cols; ++i)
      for (j = 1; j <= a_cols; ++j)
        c[k][i] += a[k][j] * b[j][i];

/* Matrix c printed row_wise */
  for (k = 1; k <= c_rows; ++k)
   {
     for (i = 1; i <= c_cols; ++i)
       printf("%d ", c[k][i]);
     printf("\n");
   }
}
```

Program 10.9 Matrix multiplication.

is the maximum size allowed in this program. The matrix c[20][20] has been initialised with zeros stored in its components. The first scanf statement reads values for the number of rows and columns of matrices *a* and *b*. After that matrices *a* and *b* are read columnwise by two pairs of *for* loops. Number of rows and columns of *c* are now initialised from the known values of the rows and columns of *a* and *b* respectively.

The next set of three *for* statements perform the matrix multiplication. In Table 10.4 we trace these *for* loops to illustrate how the multiplication is performed. We use the matrices defined at the beginning of this example. Table 10.4 tabulates the values of i, j, k and c[k][i] when the three nested *for* statements are executed.

Table 10.4 Illustrating how Example Program 10.9 does Matrix Multiplication

(max) k = c_rows = a_rows = 2; (max) i = b_cols = 2, (max)j = a_cols = 3

k	i	j	c[k][i]
1	1	1	$c_{11} = c_{11} + a_{11}\ b_{11}$
1	1	2	$c_{11} = a_{11}b_{11} + a_{12}\ b_{21}$
1	1	3	$c_{11} = a_{11}\ b_{11} + a_{12}\ b_{21} + a_{13}\ b_{31}$
1	2	1	$c_{12} = c_{12} + a_{11}\ b_{12}$
1	2	2	$c_{12} = a_{11}\ b_{12} + a_{12}\ b_{22}$
1	2	3	$c_{12} = a_{11}\ b_{12} + a_{12}b_{22} + a_{13}b_{32}$
2	1	1	$c_{21} = c_{21} + a_{21}b_{11}$
2	1	2	$c_{21} = a_{21}b_{11} + a_{22}b_{21}$
2	1	3	$c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}$
2	2	1	$c_{22} = c_{22} + a_{21}\ b_{12}$
2	2	2	$c_{22} = a_{21}b_{12} + a_{22}b_{22}$
2	2	3	$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$

EXERCISES

10.1 Write a program using a *for* loop and an array to solve Exercise 2.10.

10.2 Write a program to obtain the product of the following matrices:

$$A = \begin{bmatrix} 5 & 9 & -2 \\ 8 & 4 & 5 \\ 0 & 4 & 8 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 4 \\ 0 & 5 \\ 5 & 2 \end{bmatrix}$$

10.3 Write a program to transpose the following matrix.

$$A = \begin{bmatrix} -9 & 8 & 7 & 16 & 0 \\ 4 & 13 & 2 & -1 & 5 \\ -9 & 4 & 3 & 1 & -8 \end{bmatrix}$$

10.4 Write a program to rearrange the elements of each row of the above matrix such that the elements of each row are arranged in a descending order as shown below:

$$\begin{bmatrix} 16 & 8 & 7 & 0 & -9 \\ 13 & 5 & 4 & 2 & -1 \\ 4 & 3 & 1 & -8 & -9 \end{bmatrix}$$

10.5 Write a program to find the sum of squares of elements on the diagonal of a square matrix.

10.6 Write a program to find if a square matrix is symmetric.

10.7 A factory has 3 divisions and stocks 4 categories of products. An inventory table is updated for each division and for each product as they are received. There are three independent suppliers of products to the factory:

(i) Design a data format to represent each transaction.
(ii) Write a program to take a transaction and update the inventory.
(iii) If the cost per item is also given write a program to calculate the total inventory value.

10.8 The data obtained from life tests of 50 electric bulbs is tabulated below. Write a program to obtain a frequency distribution table conforming to the following specifications:

992	1007	1001	1010	1001
1009	990	1003	1008	999
1014	992	991	1006	998
986	996	1010	1008	1008
998	1004	999	1000	1002
994	1002	1005	1008	1025
1003	981	1014	982	997
1009	1001	988	1018	991
1028	1000	1011	1012	1012
1010	1017	1010	996	996

(i) Put all bulbs with life less than 985 hours in group 1.
(ii) Put all bulbs with life greater than 1024 hours in the last group.
(iii) Divide the region between 985 and 1024 hours into eight intervals such that each interval encompasses a life-time of 5 hours (e.g. 985 to 989 is one interval).

Write a general program so that you can accommodate up to 200 data and 20 class intervals. Read the following from the terminal.

N = No. of data = 50 in this specific case.
Lowlimit = Lowerlimit. All data less than Lowlimit will be put in class 1.
Hilimit = High limit. All data greater than Hilimit will be put in the last class.
Interval = Interval width, it is 5 in this example.
Do all calculations in integer mode.

10.9 Write a program to solve the following simultaneous equation by Gauss method and by Gauss Jordan method. Evaluate the determinant of the matrix of coefficients.

$$x_1 + 2x_2 + 3x_3 = 6$$
$$x_1 + 10x_2 + 4x_3 = -29$$
$$-2x_1 - 4x_2 + x_3 = 9$$

10.10 Economic order quantity may be calculated from the equation

$$Q = (2RS/I)^{0.5}$$

Where R is the yearly requirement, S the setup cost and I the carrying cost per item. The values of R, S and I for 15 items in factory are given. Write a program using *for* loop and arrays to calculate the economic order quantity for each of these items.

11. Logical Expressions and More Control Statements

Learning Objectives

In this chapter we will learn:

1. The use of logical operators
2. The use of *typedef* in C to create new data type name
3. The use of switch statement when one out of a set of alternate set of statements are to be carried out
4. The use of break statement to leave a loop if found necessary
5. The use of continue statement in a loop if one iteration of the loop is to be skipped or partially carried out

11.1 INTRODUCTION

We have seen in Chapter 8 that float or integer quantities may be connected by relational operators to yield an answer which is True or False. For example the expression

marks >= 60

would have an answer *true* if marks is greater than or equal to 60 and *false* if marks is less than 60. The result of the comparison (marks >= 60) is called a logical quantity. C provides a facility to combine such logical quantities by *logical operators* to yield *logical expressions*. These logical expressions are very useful in translating intricate problem statements. This is illustrated by the following example:

Example 11.1

A university has the following rules for a student to qualify for a degree with Physics as the main subject and Mathematics as the subsidiary subject:

(i) He should get 50 percent or more in Physics and 40 percent or more in Mathematics.

(ii) If he gets less than 50 percent in Physics he should get 50 percent or more in Mathematics. However, he should get atleast 40 percent in Physics.

(iii) If he gets less than 40 percent in Mathematics and 60 percent or more in Physics he is allowed to reappear in an examination in Mathematics to qualify.

(iv) In all other cases he is declared to have failed.

These rules may be expressed as the decision table of Table 11.1. The rules in Table 11.1 are expressed in Example Program 11.1 which uses logical expressions.

Table 11.1 A Decision Table for Examination Results

Physics Marks	>= 50	>= 40	>= 60	Else
Mathematics Marks	>= 40	>= 50	< 40	
Pass	x	x	–	–
Repeat Mathematics	–	–	x	–
Fail	–	–	–	x

```
/* Example Program 11.1 */
/* This program implements Decision Table 11.1 */
#include <stdio.h>

main()
{
  unsigned int roll_no, physics_marks, math_marks;
  while(scanf("%d %d %d", &roll_no, &physics_marks,
              &math_marks) != EOF)
/* && is logical AND operator and || the logical OR
   operator */
    {
      if (((physics_marks >= 50) &&
           (math_marks >= 40))  ||
           ((physics_marks >= 40) &&
           (math_marks >= 50)))
        printf("%d %d %d Pass\n", roll_no,
               physics_marks, math_marks);
      else
        {
          if ((physics_marks >= 60) &&
              (math_marks < 40))
            printf("%d %d %d Repeat Math\n",
                   roll_no, physics_marks,
                   math_marks);
          else
            printf("%d %d %d Failed\n", roll_no,
                   physics_marks, math_marks);
        }
    } /* End while */
} /* End main */
```

Program 11.1 Implementing Decision Table 11.1.

Observe the first *if* statement in the program. It is equivalent to the English statement: If the Physics marks is greater than or equal to 50 *and* the Mathematics marks is greater than or equal to 40 *or* if the Physics marks is greater than or equal to 40 *and* the mathematics marks is greater than or equal to 50 *then* write Pass. The connectives used, namely, *and*(&&), *or*(||) have a well-defined meaning when they operate on logical quantities.

We will now give the rules of syntax for forming logical expressions and the precise nature of the operations &&, ! and ||.

11.2 LOGICAL OPERATORS AND EXPRESSIONS

Programming languages such as Pascal provide a special data type called *boolean* to represent logical quantities which have only true or false values. C language, however, does not have such a data type. Data type *int* is the one which is used to represent logical quantities. In fact more strictly we could use *unsigned short int* to represent logical quantities as they have only two values 0 or 1 representing false and true. For convenience of remembering that some variables are logical quantities, C provides a facility called *typedef* for creating new data type names. Thus if we declare:

```
typedef unsigned short int Boolean ;
```

then the name Boolean becomes a synonym for *unsigned short int*. The word Boolean is used to declare logical variables which take on only values 0 or 1. An algebra using such variables was originally proposed by George Boole in the nineteenth century.

Thus if we define:

```
#define TRUE 1
#define FALSE 0
typedef unsigned short int Boolean ;
Boolean a, b ;
```

the three logical operators !, && and || which operate on variables a, b may be defined as shown in Table 11.2.

Table 11.2 Definition of Logical Operators

Operator	Operation	Definition Symbol		
!	Complement or negation (not)	a	!a	
		TRUE	FALSE	
		FALSE	TRUE	
&&	Logical Intersection (and)	a	b	a && b
		FALSE	FALSE	FALSE
		FALSE	TRUE	FALSE
		TRUE	FALSE	FALSE
		TRUE	TRUE	TRUE
||	Logical Union (or)	a	b	a || b
		FALSE	FALSE	FALSE
		FALSE	TRUE	TRUE
		TRUE	FALSE	TRUE
		TRUE	TRUE	TRUE

Example 11.2

A company has three shareholders. Agarwal, Bhatia and Chamanlal. Agarwal owns 50 shares,

Bhatia 30 and Chamanlal 20 shares. For a measure to pass it must be supported by shareholders the sum of whose holdings exceed 2/3 of the total shares. A program which will determine if a measure is successful or not and print an appropriate message will now be developed.

Let a 'yes' vote be represented by 1 and a 'No' vote by 0. Let the passage of the measure be indicated by 1 and its failure by 0. A variable name measure_passes will be set = 1 by the program if the measure passes and=0 if it fails. Example Program 11.2 is developed for this problem.

Observe in Example Program 11.2 the statement:

measure_passes = (agar_vote && bhat_vote) || (agar_vote && cham_vote)

```
/* Example Program 11.2 */
/* Illustrating use of logical operators */
#include <stdio.h>

main()
{
  typedef unsigned int Boolean;
  Boolean agar_vote, bhat_vote, cham_vote,
  measure_passes;

  while (scanf("%u %u %u", &agar_vote, &bhat_vote,
          &cham_vote) != EOF)
    {
      printf("%d %d %d\n", agar_vote, bhat_vote,
          cham_vote);
      measure_passes =  (agar_vote && bhat_vote)
       || (agar_vote && cham_vote);
      if (measure_passes)
       printf("Measure Passes\n");
      else
       printf("Measure Fails\n");

    } /* End while */
} /* End main */
```

Program 11.2 Use of logical operators.

The expression on the right of = will be TRUE (i.e., 1) if and only if agar_vote = 1 and bhat_vote = 1 or agar_vote = 1 and cham_vote = 1.

It may thus also be written as:

measure_passes = agar_vote && (bhat_vote || cham_vote)

In the statement:

if (measure_passes)
    printf ("Measure Passes \n");
else
    printf("Measure Fails\n") ;

the program will print Measure Passes if measure_passes is TRUE, else it will print Measure Fails.

11.3 PRECEDENCE RULES FOR LOGICAL OPERATORS

The precedence or hierarchy rules in evaluating logical expressions will be explained in this section with examples.

Example 11.3

Consider the following expression where a, b, x, y are integers

```
a > b * 4 && x < y + 6
```

Even though syntactically the above expression is correct, it is difficult to interpret. It is better to use parentheses and write it as:

```
(a > b * 4 ) && (x < y + 6)
```

In the above example, the expressions within the parentheses are evaluated first. The arithmetic operations are carried out before the relational operations. Thus $b * 4$ is calculated and after that a is compared with it. Similarly $y + 6$ is evaluated first and then x is compared with it.

In general within parentheses:

1. The unary operations, namely, –, ++, – – , ! (logical not) are performed first.
2. Arithmetic operations are performed next as per their precedence.
3. After that the relational operations in each of the subexpressions are performed, each subexpression will be either 0 or non-zero. If it is zero it is taken as *false* else it is taken as *true*.
4. These logical values are now operated on by the logical operators.
5. Next the logical operation && is performed.
6. The logical *or* operation, namely, || is performed next.
7. Lastly the evaluated expression is assigned to a variable name as per the assignment operator.

Example 11.4

```
!( a > b/3) ||(c != d/3) &&(d < 5)
```

Here a > b/3, c != d/3, and d < 5 are evaluated first !(a > b/3) is then found. Next (c!= d/3) is *anded* with the result of (d < 5)

The result is *ored* with the result of !(a > b/3).

If a = 8, b = 6, c = 3, d = 8 the result is:

(8 > 6/3) = 1 (true)

(3 != 8/3) = 1 (true)

(8 < 5) = 0 (false)

!(8 > 6/3) = !(1) = 0

(3!= 8/3) && (8 < 5) = (1) && (0) = 0

Finally 0 || 0 = 0

Example 11.5

(a – 5.5 >= 9.5) || (c < d) && (x >= y)

Given a = 15.0, c = 6.0, d = 4.0, x = 3.0 and y = 4.0

(a – 5.5) >= 9.5 is *true*

(c < d) = (6.0 < 4.0) is *false*

(x >= y) = (3.0 >= 4.0) is *false*

As per precedence we do the && operation first:

(6.0 < 4.0) && (3.0 >= 4.0) = *false* && *false* = *false*

The || operation is done next:

true || *false* = *true*

which is the result.

If the expression had been

((a – 5.5 >= 9.5) || (c < d)) && (x >= y)

then the result would be *false.*

A summary of precedence rules is given in Table 11.3.

Table 11.3 Illustrating Precedence of Operators in C

Highest Precedence (Done first)	↓	Parenthesised subexpressions
		– (unary negation) ++ – – !
		* / %
		+ –
		< <= > >=
		== !=
		&&
Lowest Precedence (Done last)		||
		=

11.4 SOME EXAMPLES OF USE OF LOGICAL EXPRESSIONS

Example 11.6

We will consider again Example 2.5 for which a decision table was given in Chapter 2. It is reproduced as Table 11.4 for ready reference.

A program which corresponds to Table 11.4 is given as Example Program 11.3. In this program it is assumed that each line of input has the roll number of candidate and his/her marks in the main and ancilliary subject. Status = 1 is used to indicate special status. When

Table 11.4 A Decision Table for Declaring Results

Conditions	Rules					
Main Marks	≥ 50	≥ 40	≥ 60	≥ 40	≥ 40	Else
Anc. Marks	≥ 40	≥ 50	< 40	≥ 40	< 40	
Special Status	No	No	No	Yes	Yes	
Actions						
Pass	x	x	—	x	—	—
Repeat Anc.	—	—	x	—	x	—
Fail	—	—	—	—	—	x

```
/*  Example Program 11.3 */
/*  Illustrates use of Decision Table  */
#include <stdio.h>
main()
{
  typedef unsigned int Boolean;
  int roll_no, main_marks, anc_marks;
  Boolean pass, repeat_anc, c1, c2, c3, c4, c5, status;
  printf("      Roll No   Main Marks   Anc. Marks");
  printf("   Status\n" );
  while(scanf("%u %u %u %u", &roll_no, &main_marks,
      &anc_marks, &status) != EOF)
    {
      c1 = (main_marks >= 50);
      c2 = (main_marks >= 40);
      c3 = (main_marks >= 60);
      c4 = (anc_marks >= 50);
      c5 = (anc_marks >= 40);
      pass = (c1 && c5) || (c2 && c4) ||
             (c2 && c5 && status);
      repeat_anc = (c3 && !c5) ||
                   ( c2 && !c5 && status);
      if (pass)
       printf("%10d %10d %10d %10d  PASS\n",
              roll_no, main_marks, anc_marks, status);
      else
       {
         if (repeat_anc)
           printf("%10d %10d %10d %10d  REP. ANC\n",
                  roll_no, main_marks, anc_marks, status);
         else
           printf("%10d %10d %10d %10d  FAIL\n",
                  roll_no, main_marks, anc_marks, status);
       }
    } /* End while */
} /* End main */
```

Program 11.3 Implementing Decision Table 11.4.

end of students' data is reached scanf returns EOF and this is used to terminate the *while* loop. Observe the use of logical expressions and connectives in the program.

Example 11.7

A program is to be written to decide whether on a given date an employee in an organization has completed 1 year service. In order to do this a line is typed for each employee containing the data: employee number, day of joining, month of joining and year of joining. The form in which data in input is shown in Table 11.5.

Table 11.5 Data Input for Example 11.7

Data	Explanation
09 08 1992	Today's date
3452 08 09 1990	Employees' data
3462 09 08 1991	
3672 08 07 1990	
3792 09 01 1992	

A decision table to decide whether an employee has completed one year's service is given in Table 11.6.

Table 11.6 Decision Table to Decide Completion of One Year's Service

Diff. in year	> 1	= 1	= 1	E
Diff. in month	–	> 0	= 0	L
Diff. day	–	–	≥ 0	S
				E
One year service completed?	Yes	Yes	Yes	No

The conditions checked in Table 11.6 are:

(1) Difference between today's year and the year an employee joined.

(2) Difference between today's month and month an employee joined.

(3) Difference between today's day and the day an employee joined.

The reader can convince himself about the correctness of the table by fixing today's date and trying a number of cases of joining dates.

A program which implements Table 11.6 is given as Example Program 11.4. The reader would notice the simplicity of writing a program corresponding to the decision table.

11.5 THE SWITCH STATEMENT

A control statement called the *switch* statement is available in C. It is useful when one out of a set of alternative actions is to be taken based on the value of an expression.

It is particularly useful when variable values are classified with codes. We will first illustrate the use of a switch statement with an example.

```
/*  Example Program 11.4 */
/*  Program to check completion of one year's
    service */
#include <stdio.h>

main()
{
  int emp_no, day, month, year, today, tod_month,
      tod_year, dif_day, dif_month, dif_year;

  scanf("%d %d %d", &today, &tod_month, &tod_year);
  printf("      Emp. No.      Day       Month");
  printf("     Year  Eligible or not\n");
  while(scanf("%d %d %d %d", &emp_no, &day,
              &month, &year) != EOF)
    {
      dif_day = today - day;
      dif_month = tod_month - month;
      dif_year = tod_year - year;
      if ((dif_year > 1) || ( (dif_year == 1) &&
         (dif_month > 0)) || ((dif_year == 1) &&
           (dif_month == 0) && (dif_day >= 0)))
       printf("%10d %10d %10d %10d   Eligible\n",
              emp_no, day, month, year);
      else
       printf("%10d %10d %10d %10d Not Eligible\n",
              emp_no, day, month, year);
    } /* End of while */
} /* End of main */
```

Program 11.4 Completion of one year service.

Example 11.8

A company manufactures three products: engines, pumps and fans. They give a discount of 10 percent on order for engines if the order is for Rs. 5,000 or more. The same discount of 10 percent is given on pump orders of value of Rs. 2,000 or more and on fan orders for Rs. 1,000 or more. On all other orders they do not give any discount. A program which implements this policy is given next.

Assume that the following codes are used to indicate the product:

Code 1 for engines

Code 2 for pumps

Code 3 for fans

A data is made up for each order with its serial number, the product code followed by the amount of order. For example if the order has serial number 2527 and is for pumps worth Rs. 2,500 the data will be shown as:

2527 2 2500.00

A decision table showing the above discount policy is given as Table 11.7.

Table 11.7 Decision Table for Discount Policy

Product Code Order Amount	1 >= 5000	2 >= 2000	3 >= 1000	Else
Discount	10%	10%	10%	0%

A computer program using only *if* statements to implement the above decision table is given as Example Program 11.5.

```
/* Example Program 11.5 */
/* Discount calculation example using logical if
   The decision table of Table 11.6 is used  */
#include <stdio.h>

main()
{
  int serial_no, code;
  float amount, discount, net_amount;
  printf("    Serial No.    Code    Discount");
  printf("  Net amount\n");
  while(scanf("%d %d %f", &serial_no, &code,
              &amount) != EOF)
    {
      discount = 0.0;
      if (code == 1)
        if (amount >= 5000.0)
          discount = 0.1;
      if (code == 2)
        if (amount >= 2000.0)
          discount = 0.1;
      if (code == 3)
        if (amount >= 1000.0)
          discount = 0.1;
      if ((code < 1) || (code > 3))
        printf("CODE ERR, SER. NO. = %d, CODE = %d\n",
               serial_no, code);
      net_amount = amount * (1.0 - discount);
      printf("%10d %10d %10.2f %10.2f\n",
             serial_no, code, discount, net_amount);
    } /* End of while */
} /* End of main */
```

Program 11.5 Discount calculation.

In Example Program 11.5 observe that if (code == 1) is *true* and amount is >= 5000 discount is assigned 0.1. After this instead of reading the next data line the program will

continue sequentially and execute the statement if (code == 2). This could have been avoided by using an *else*. The nesting of *ifs* with *elses* will make the program difficult to read and understand. The program as it is written is inefficient but somewhat easier to understand. As such nested *ifs* occur often in practice a statement known as *switch* is available in C which is useful to write such programs. The same problem is solved with the program given as Example Program 11.6.

In this program the statement:

switch (code)

```
/* Example Program 11.6 */
/* Discount calculation example using switch statement */
#include <stdio.h>
main()
{
  int serial_no, code;
  float amount, discount, net_amount;
  printf("    Serial No.    Code    Discount");
  printf("  Net amount\n");
  while(scanf("%d %d %f\n", &serial_no, &code,
              &amount) != EOF)
    {discount = 0
      switch(code)
        {
        case 1:
          if (amount >= 5000.0)
            discount = 0.1;
          break;
        case 2:
          if (amount >= 2000.0)
            discount = 0.1;
          break;
        case 3:
          if (amount >= 1000.0)
            discount = 0.1;
          break;
        default:
          printf("ERR,  SER. NO. = %d, CODE = %d\n",
                 serial_no, code);
          discount = 0.0;
          break;
        }
      net_amount = amount * (1.0 - discount);
      printf("%10d %10d %10.2f %10.2f\n",
             serial_no, code, discount, net_amount);
    } /* End of while */
} /* End of main */
```

Program 11.6 Implementing Decision Table 11.6.

will transfer control to case 1 if code == 1, to case 2 if code == 2 and to case 3 if code == 3. If code is less than 1 or greater than 3 then control will go to default. If code == 2, for example, then control will go to case 2. The statements following case 2 will be executed. In this case the statement if (amount >= 2000.0) is *true* discount will be assigned the value 0.1. Observe the next statement is *break*. This statement commands that all the statements following within the domain of the switch statement, namely, those upto closing braces} of the switch statement should not be executed. Control should go to the statement following the closing braces} of the switch statement, namely the statement:

net_amount = amount * (1.0 – discount) ;

The *break* statement is very important in this problem. If *break* was not written in case 2: control would go sequentially to the following statements and perform the statements following case 3: and default: The answer will thus be incorrect.

We will now explain the syntax and semantics of switch statement.

Syntax rules for *switch* statement

The general form of the switch statement is given below:

```
switch (expression)
   {
      case constant_1:
         s11 ;
         s12 ;
         .
         .
         break ;
      case constant_2:
         s21;
         s22;
         .
         .
         break;
         .....
         .....
      case constant_n:
         sn1 ;
         sn2 ;
         .
         .
         .
         break ;
      default:
         sd1 ;
         sd2 ;
         sdn;
         break ;
   }
```

In the *switch* statement the expression can be an integer expression or a constant. It is evaluated and compared with constant_1, constant_2, … constant_n. If it matches, say, constant_2

then the statements s_{21}; s_{22} ; s_{23} ... etc. are executed till *break* is reached. When break is encountered the program jumps to the statement following the closing braces } of the *switch* statement. If the value of the expression is not equal to any one of constant _1, constant_2, ... constant_n, then control passes to *default* and the statements s_{d1}; s_{d2}; ....s_{dn}; following *default* are executed. As *default* is the last option in the domain of the *switch* statement no *break* statement is required in default. We have, however, used break as it will not do any harm. If we sometime want to add another case, the presence of break will prevent us from making an accidental error. Constant_1, constant_2, constant_n should be integers. They should all be distinct. In other words, they should not have the same value. If *break* is omitted in any of the cases, execution continues with the statements of the following case until the closing braces } of the switch statement is reached. If a *break* is encountered before the closing braces } the statements following the *break* are not executed and control leaves the domain of the *switch* statement.

We give below some legal *switch* statements:

```
int i, h
(i) switch ( i * i + 4)
    {
       case 5 :
          ++ k;
          break ;
       case 8 :
          k += 8 ;
          break ;
    }
```

If i = 2 and k = 4 then i * i + 4 = 8. The program branches to case 8 : k is set to 12 and control passes to the statement following *switch*.

If i = 3 the expression i * i + 4 = 13. None of the constants in case match. Thus control goes to the statement following switch doing nothing. If default action is specified it would have carried it out.

```
(ii) int i, k ;
   i = 2 ; k = 3 ;
   switch ( i – k )
      {
         case – 1:
            ++ i ;
            ++ k ;
               /* WRONG missing break */
         case 2 :
            – – i ;
            ++ k ;
               /* WRONG missing break */
         default:
            i += 3 ;
            k += i ;
      }
```

This is a legal *switch* statement syntactically. In this case (i – k) when evaluated gives – 1. Thus case – 1 is satisfied. A negative constant is legal. The integers *i* and *k* are incremented. *i* becomes 3 and *k* become 4. As there is no *break* the following statements are executed including those following default. Thus finally *i* becomes 5 and *k* becomes 10. Semantically this is a wrong *switch* statement as the programmer possibly wanted to distinguish between cases with (i – k) = – 1 and 2.

```
(iii) int i, j, k ;
      scanf("%d%d%d", &i, &j, &k) ;
      switch ( i + j – k )
         {
            case 0 : case 2: case 4 :
               ++ i ;
               k += j ;
               break ;
            case 1 : case 3 : case 5 :
               – – i ;
               k –= j ;
               break ;
            default :
               i += j ;
               break ;
         }
```

Observe that in the above *switch* statement a set of statements are to be executed in many cases. This is allowed by C.

The following are some illegal *switch* statements.

```
(i) float a ;
    switch (a)
      {
         case 2.5 :
            b += 2.3 ;
            break ;
         case 3.2 :
            b – = 2.8 ;
            break ;
      }
```

(A floating point constant in case not allowed)

```
(ii) int i, j ;
     switch ( i – j + 3)
        {
          case 2 : 3 :
             j += 6 ;
             break ;
```

```
            case 4 :
               i -= 4 ;
               break ;
         }
```

(only one constant allowed in case)

It should be written as:

```
   case 2 : case 3 :
      j += 6 ;
      break ;
```

(iii)
```
int i, j ;
switch (i + j ) ;
   {
      case 2 :
         i += 4 ;
         break ;
      case 4 :
         i -= 4 ;
         break ;
   }
```

(Semicolon after *switch* statement illegal)

(iv)
```
int i, j ;
switch ( i + j + 6)
   {
      case 2 :
         i += 4 ;
         break ; }
      case 3 :
         i -= 4 ;
   }
```

Illegal closing braces } after break

Example 11.9

In Chapter 8 we gave a program (Example Program 8.4) to solve a quadratic equation:

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

We identified three cases, namely, the discriminant $(b^2 - 4ac) < 0$, $(b^2 - 4ac) = 0$ and $(b^2 - 4ac) > 0$. We used *logical if* statements in that example to write the program. We now rewrite the program (as Example Program 11.7) using the *switch* statement. We first assign a value 1, 2 or 3 to an index *i* depending on the value of $(b^2 - 4ac)$. This index is used in a *switch* statement to branch to the appropriate compound statement. This program is clearer compared to Example Program 8.4. This program will work for any set of values of (a, b, c).

```
/* Example Program 11.7 */
/* Quadratic Equation Solution */
#include <stdio.h>
#include <math.h>
main()
{
  float a, b , c, discrmnt, x_imag_1, x_imag_2,
  x_real_1, x_real_2, temp;
  int i;
  while(scanf("%f %f %f", &a, &b, &c) != EOF)
    {
      printf("a = %f, b = %f, c = %f\n", a, b, c);
      discrmnt = b*b - 4.0*a*c;
      if (discrmnt < 0)
       i = 1;
      else
       {
         if (discrmnt == 0)
           i = 2;
         else
           i = 3;
       }
      switch (i)
       {
       case 1:
         discrmnt = -discrmnt;
         x_imag_1 = sqrt(discrmnt) / (2.0 * a);
         x_imag_2 = -x_imag_1;
         x_real_1 = -b / (2.0 * a);
         printf("Complex conjugate roots\n");
         printf("real part = %16.8e\n", x_real_1);
         printf("imaginary part = %16.8e\n",
               x_imag_1);
         break;
       case 2:
         x_real_1 = -b / (2.0 * a);
         printf("Repeated roots\n");
         printf("Real roots = %16.8e\n",
               x_real_1);
         break;
       case 3:
         temp = sqrt(discrmnt);
         x_real_1 = (-b + temp) / (2.0 * a) ;
         x_real_2 = (-b - temp) / (2.0 * a);
         printf("Real roots\n");
         printf("real root_1 = %16.8e\n",
               x_real_1);
         printf("real root_2 = %16.8e\n",
               x_real_2);
         break;
       } /* End switch */
    } /* End while */
} /* End of main */
```

Program 11.7 Solving quadratic equation—use of switch statement.

Example 11.10

Suppose the rates of tax on gross income are as shown in Table 11.8.

Table 11.8 Table of Tax Rates

Income	Tax
<Rs. 10,000	Nil
Rs. 10,000 to Rs. 19,999	10 percent
Rs. 20,000 to Rs. 29,999	15 percent
Rs. 30,000 to Rs. 49,999	20 percent
>Rs. 50,000	25 percent

A program is required to compute the tax. A simple programming technique may be used to code the slab in which a particular income is to be placed. Suppose we divide the income by 10000 as shown below:

slab_code = income/10000
Then for Income < 10000, slab_code = 0,

for 20000 > Income >= 10000, slab_code = 1
for 30000 > Income >= 20000, slab_code = 2
for 50000 > Income >= 30000, slab_code = 3 or 4
and for Income >= 50000, slab_code > 5

Observe that if the income > 50000 then slab_code > 5. Thus if we want to use a switch statement with slab_code as case index then we should map these values to one index. This is done in Example Program 11.8.

Observe that the two cases slab_code = 3 and slab_code = 4 in which the action is identical can be put together and this is done in Example Program 11.8.

11.6 THE *break* STATEMENT

We have used a statement called *break* in a *switch* statement to skip statements within the domain of the *switch* statement. This statement can also be used within *while*, *for* and *do while* loops to abandon processing and leave the loop. We will illustrate this with an example.

Example 11.11

We are given a set of *n* numbers, a[0], a[1], a[2], a[3],.... a[n – 1]. It is stated that they are in strictly ascending order. A program is to be written to check this.

If the numbers are in strictly ascending order then a[i + 1] > a[i] for i = 0 to (n – 2). If for any *i* this is not true we should give a message and stop further checking. This procedure is coded as Example Program 11.9. In the following for loop

```
for (i = 0; i <= n – 2; ++ i)
  {
    if (a[i + 1] > a[i])
       ascending = 1 ;
    else {ascending = 0 ;
       break ;}
  }
```

```
/* Example Program 11.8 */
/* Illustrating use of switch statements */
#include <stdio.h>
main()
{
  int slab_code, income, account_no;
  float tax_rate, tax;
  printf("   Account No. Income      Tax\n");
  while(scanf("%d %d", &account_no, &income) != EOF)
    {
      slab_code = income / 10000;
      if (slab_code >= 5)
        slab_code = 5;
      switch(slab_code)
        {
        case 0:
          tax_rate = 0;
          break;
        case 1:
          tax_rate = 0.1;
          break;
        case 2:
          tax_rate = 0.15;
          break;
        case 3:
        case 4:
          tax_rate = 0.2;
          break;
        case 5:
          tax_rate = 0.25;
          break;
        } /* End of switch */
      tax = (float)income * tax_rate;
      printf("%10d %10d %10.2f\n", account_no, income, tax);
    } /* End of while */
  printf("end of input data\n");
} /* End of main */
```

Program 11.8 Tax calculation.

if for any *i*, a[i + 1] <= a[i] the numbers are not in ascending order. The *break* statement in the *if* statement of the loop commands control to leave the loop. When control leaves the loop it reaches the statement following the loop statement. This statement prints the appropriate message.

We can similarly leave a *while* loop. The most common use of break, however, is in switch statement.

```
/* Example Program 11.9 */
/* Program checks whether a set of numbers are
   in ascending order */
#include <stdio.h>

main()
{
  int a[10], i, n, ascending;
  scanf("%d", &n);
  for (i = 0; i <= n-1; ++i)
    scanf("%d", &a[i]);
  for (i = 0; i <= n-2; ++i)
    {
      if (a[i+1] > a[i])
       ascending = 1;
      else
       {
         ascending = 0;
         break;
       }
    }
  if (ascending)
    printf("Numbers in ascending order\n");
  else
    printf("Numbers not in ascending order\n");
} /* End of main */
```

Program 11.9 Checking ascending order of numbers.

11.7 THE *continue* STATEMENT

There is a statement in C called *continue* which is used to skip executing statements in a loop following it. A *break* abandons processing in a loop and gets out of it. *Continue* statement, however, continues processing the loop from the next iteration. We will illustrate the use of *continue* statement with an example.

We wrote a program (Example Program 11.7) to solve the quadratic equation

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

We assumed, that the coefficient *a* is not equal to zero. If *a* equals zero then the equation is a linear equation and Example Program 11.7 will not work. We will now generalise the program to include the possibility of a = 0. This program is given as Example Program 11.10. If a = 0 the equation is linear and the solution is x = – c/b. Once this solution is found *continue* is used to skip the rest of the processing in the loop. Control returns to the *while* statement to process the next set of values.

Continue statement may also be used in a *for* loop. The effect of using *continue* in a *for* loop is illustrated in Example Program 11.11. In this program *continue* is used to prevent division by zero. Observe that the loop is carried out 4 times as the index *i* is incremented and checked for termination condition after each looping by the *for* statement.

```
/* Example Program 11.10 */
/* Quadratic Equation Solution - Use of continue */
#include <stdio.h>
#include <math.h>
main()
{
  float a, b, c, discrmnt, x_imag_1, x_imag_2,
  x_real_1, x_real_2, temp, x;
  int i;
  while(scanf("%f %f %f", &a, &b, &c) != EOF)
    {
      printf("a = %f, b = %f, c = %f\n", a, b, c);
      if (a == 0.0)
       {
         printf("a = 0, Equation is linear\n");
          x = -c/b;
         printf("Solution is : x = %f\n", x);
         continue;
       }
      discrmnt = b*b - 4.0*a*c;
      if (discrmnt < 0)
       i = 1;
      else
       {
         if (discrmnt == 0)
           i = 2;
         else
           i = 3;
       }
      switch (i)
       {
       case 1:
         discrmnt = -discrmnt;
         x_imag_1 = sqrt(discrmnt) / (2.0 * a);
         x_imag_2 = -x_imag_1;
         x_real_1 = -b / (2.0 * a);

         printf("Complex conjugate roots\n");
         printf("real part = %16.8e\n", x_real_1);
         printf("imaginary part = %16.8e\n",
                x_imag_1);
         break;

       case 2:
         x_real_1 = -b / (2.0 * a);
         printf("Repeated roots\n");
         printf("Real roots = %16.8e\n", x_real_1);
         break;
```

```
            case 3:
              temp = sqrt(discrmnt);
              x_real_1 = (-b + temp) / (2.0 * a) ;
              x_real_2 = (-b - temp) / (2.0 * a);
              printf("Real roots\n");
              printf("real root_1 = %16.8e\n", x_real_1);
              printf("real root_2 = %16.8e\n", x_real_2);
              break;
            } /* End switch */
          } /* End while */
        } /* End of main */
```

Program 11.10 Quadratic equation solution—use of continue.

```
/* Example Program 11.11 */

#include <stdio.h>
main()
{
  int i, k, p;

  for (i = 0; i <=4; ++i)
    {
      scanf("%d", &p);
      if (p == 0)
        continue;
      k = 1 / p;
      printf("k = %d i = %d p = %d\n", k, i, p);
    }
  printf("out of loop\n");
}
```

Program 11.11 Illustrates use of continue in a *for* loop.

EXERCISES

11.1 The policy followed by a company to process customer orders is given by the following rules:

(i) If a customer order is less than or equal to that in stock and his credit is OK, supply his requirement.

(ii) If his credit is not OK do not supply. Send him an intimation.

(iii) If his credit is OK but the item in stock is less than his order, supply what is in stock. Intimate to him the date the balance will be shipped.

Obtain a decision table corresponding to these rules. Write a C Program to implement the decision table.

11.2 Obtain a decision table corresponding to Exercise 8.10. If a set of data is given with each data having a customer account number, amount of deposit and period

of deposit, write a program which will give as output customer's account number, amount of deposit, period of deposit, interest accrued to the account and total amount in the customer's account.

11.3 Given the date an employee joined a job in the firm: Day/Month/Year and given today's date, write a program to find out whether the given joining date of an employee is a legal date. For example a date such such as 10/14/81 is illegal as the month cannot exceed 12. If today's date is 17/5/82, an year such as 95 or 30 would be illegal. Further, depending on the month, the range of valid days may be determined.

11.4 In the second chapter a decision table was to be developed for Exercise 2.13. Write a program to implement this decision table.

11.5 A certain steel is graded according to the results of three tests. The tests are:

(i) Carbon content < 0.7 percent
(ii) Rockwell hardness > 50
(iii) Tensile strength > 30,000 kilos/cm^2

The steel is graded 10 if it passes all three tests, 9 if it passes only tests 1 and 2, 8 if it passes only test 1, and 7 if it passes none of the tests. Obtain a flow chart corresponding to this statement of the problem. Write a program corresponding to this flow chart.

Obtain a decision table corresponding to this problem. Is an action specified for all the eight possible rules corresponding to the outcomes of the three condition tests? Discuss the difference between the flow chart and decision table formulations. Write a C program corresponding to the decision table.

11.6 Obtain decision tables for simulating an automatic stamp vending machine with the following specifications:

(i) It should dispense 25, 15 and 10 paise stamps.
(ii) It should accept 50, 25, 10 and 5 paise coins.
(iii) It can accept not more than one coin for each transaction.
(iv) If more than one coin of the same denomination is to be returned as change after dispensing the stamp, the machine cannot do it. Instead the coin should be returned and a 'no change' signal turned on.
(v) The machine should dispense the stamp and the right change and must indicate exceptional cases such as 'insufficient amount tendered', 'no stamp available', 'no change available', etc.

Write a program to simulate the machine. The input to the program would be: Amount tendered and the stamp requested. The output of the program should be: whether stamp is dispensed or not, the value of the stamp dispensed, the denomination of the coins returned (if any) and no change signal if no change is returned and no stamp if the stamp is not available.

11.7 Write a program corresponding to Exercise 2.11.

11.8 A set of rules in a government service for promotion are as follows:
The service has six salary points. Provided a candidate's conduct, diligence and efficiency are considered satisfactory, and he has spent one year as a Class I officer, and has passed satisfactorily the departmental test he advances to the next

higher salary point from points 1 or 2. If he is in a higher salary point, then if his conduct, diligence and efficiency are considered satisfactory, and one year has elapsed since his last increment and he has satisfactorily completed a departmental course then he advances to the next higher salary point. Analyse the above statement and find out the input data necessary to decide whether a candidate is to be promoted or not. Write a program to implement the rules.

11.9 The offshore gas company bills its customers according to the following rate schedule:

First	50 cmeters	Rs. 40 (flat rate)
Next	300 cmeters	Rs. 1.25 per 10 cmeters
Next	3000 cmeters	Rs. 1.20 per 10 cmeters
Next	2500 cmeters	Rs. 1.10 per 10 cmeters
Next	2500 cmeters	Re. 0.90 per 10 cmeters
Above this		Re. 0.80 per 10 cmeters

Given an input for each customer in the format:
Customer number, Previous meter reading, New meter reading.
Write a program to output the following:
Customer number, Previous reading, New reading, Gas used, Total bill.

11.10 An electricity company has three categories of customers: Industrial, Bulk Institutional and Domestic. The rates for these are tabulated below:

Industrial

Minimum up to 5000 units	Rs. 1500
Next 5000 units	Re. 0.25 per unit
Next 10000 units	Re. 0.23 per unit
Above this	Re. 0.20 per unit

Bulk Institutional

Minimum up to 5000	Rs. 1800
Next 5000 units	Re. 0.30 per unit
Next 10000 units	Re. 0.28 per unit
Above this	Re. 0.25 per unit

Domestic

Minimum up to 100 units	Rs. 50
Next 100 units	Re. 0.50 per unit
Next 200 units	Re. 0.45 per unit
Above this	Re. 0.40 per unit

Given the customer number, category of customer, the previous meter reading and the current reading, write a program to output along with the input data, the charges for use of electricity.

12. C Program Examples

Learning Objectives

In this chapter we will learn:

1. How to simulate a small computer
2. Write complete programs

Many important features of C have now been discussed and it will be worthwhile to write complete programs to solve some interesting problems. We will consider three examples in this chapter.

12.1 DESCRIPTION OF A SMALL COMPUTER

In this section we will write a C program to simulate the detailed working of a small hypothetical digital computer. The program is called a simulator program for SMAC (SMAll Computer). This technique of simulating the function of a computer on an already existing computer is widely used for developing software for new computers. Further, this example will illustrate the internal structure and functioning of a typical digital computer.

SMAC has the following specifications:

(i) 1000 memory locations.

(ii) Each memory location can accommodate 5 digits and a sign. We will call a sequence of 5 digits of the form ± XXXXX which can be stored in one location in memory a *word.*

(iii) Each *word* in the memory can be an instruction or a data.

(iv) An instruction for this machine consists of two parts. One part gives the code for the operation to be performed and the other part gives the location in memory where the operand will be found. The sign is assumed to be positive in an instruction and not explicitly shown. Figure 12.1 illustrates this. As the memory has a total of 1000 locations, three digits (000 to 999) are

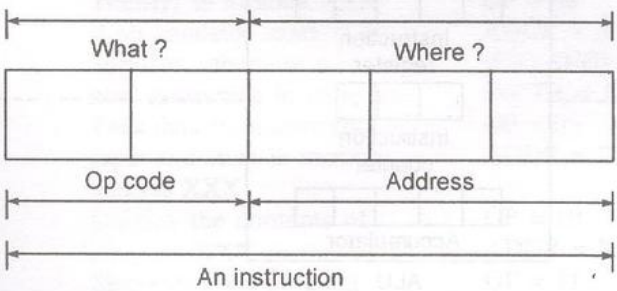

Fig. 12.1 Instruction format of SMAC.

needed to address all the locations. The remaining 2 digits in the *word* may be used to code the various operations to be performed by the computer. Theoretically 100 operations may be coded with two digits. We will, however, have a much smaller number of operation codes.

(v) SMAC has an an arithmetic logic unit (abbreviated ALU) where all arithmetic operations are performed. The ALU has a one word length (5 digits) register called an accumulator register where the results of arithmetic operations are temporarily stored. It is also used as an implied operand in arithmetic operations.

(vi) The ALU also decodes instructions and supervises execution of programs. It has an instruction register where the operation code and the address part of an instruction are stored and an instruction counter which stores the address of the next instruction to be executed. The first two digits of the instruction register contain the OP code and the last 3 digits the address of the operand to be accessed.

(vii) It has an input unit which reads data from a terminal and an output unit which prints the results.

A block diagram of the computer with all the registers used in it is shown in Fig. 12.2.

A program for SMAC consists of a set of machine instructions followed by data. Each data (or operand) can have at most 5 digits and a sign.

The computer operates in two phases. In the first phase the list of instructions is read and stored in the memory starting from address 000. The end of the list of instructions is signaled by a specially coded line. Data follow this line and are not read during this phase.

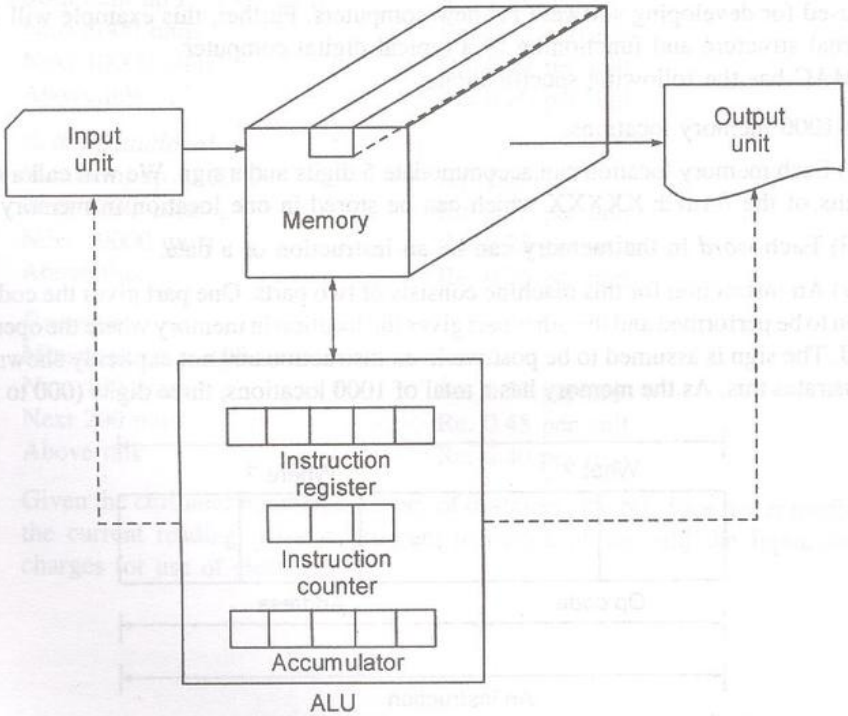

Fig. 12.2 Block diagram of SMAC.

In the second phase the instruction counter in the control units is set to 000 and the instruction stored in memory location 000 is transferred to the instruction register. The instruction counter is incremented by one to point to the next instruction to be executed.

The operation code (abbreviated OP code) is retrieved from the instruction register and decoded. The ALU activates the appropriate part of the computer for executing the instruction. For example, data will be brought in and stored in memory if the instruction is an input instruction. If the instruction is a branch instruction the content of the instruction counter is determined by the executed instruction. If it is not a branch instruction the order given by the current OP code is executed. The instruction counter gives the address of the next instruction in sequence. The list of OP codes and their purpose is given in Table 12.1. The following abbreviations are used:

Table 12.1 The Meaning of Operation Codes of SMAC

OP Code	Instruction format	Meaning	Status of registers
01	01XXX	Clear accumulator and transfer C(XXX) to accumulator	OP = 01 ADDR = XXX C(ACC) = C(XXX)
02	02XXX	Add C(XXX) to accumulator	OP = 02 ADDR = XXX C(ACC) = C(XXX) + C(ACC)
03	03XXX	Subtract C(XXX) from C(ACC)	OP = 03 ADDR = XXX C(ACC) = C(ACC) – C(XXX)
04	04XXX	Multiply C(XXX) by C(ACC)	OP = 04 ADDR = XXX C(ACC) = C(ACC) * C(XXX)
05	05XXX	Divide C(ACC) by C(XXX) (only quotient available)	OP = 05 ADDR = XXX C(ACC) = C(ACC)/C(XXX)
06	06XXX	Store C(ACC) in address XXX	OP = 06 ADDR = XXX C(XXX) = C(ACC)
07	07XXX	Take next instruction from location XXX	OP = 07 ADDR = XXX INCTR = ADDR
08	08XXX	Transfer to location XXX if accumulator contents is negative, otherwise go to the next instruction in sequence	OP = 08 ADDR = XXX IF(C(ACC) < 0) *then* INCTR = ADDR
09	09XXX	Take data from standard input and store in memory address XXX	OP = 09 ADDR = XXX
10	10XXX	Display the contents of location XXX in the VDU	OP = 10 ADDR = XXX
11	11XXX	Stop executing program Address part not used	OP = 11

ACC: Accumulator,
INCTR: Instruction counter
ADDR: Address part of an instruction
OP: Operation code part of an instruction.
OP code 01: Instruction Form: 01XXX where XXX is the address part of the instruction.

Meaning: Clear the accumulator register and enter the contents of memory location XXX in it.

In working with machine instructions it is very important to distinguish between the address of a word in memory and the actual contents of the word. We will use the notation C(XXX) to denote the contents of address XXX. An equal to symbol (=) will be used to denote "replaces".

12.2 A MACHINE LANGUAGE PROGRAM

We will now write a small program in the machine language of this machine to compare the magnitudes of two numbers and output the larger of the two. The program is given as Table 12.2. In the program of Table 12.2, columns 2 and 3 contain the machine instruction

Table 12.2 A Machine Language Program to Find the Larger of Two Numbers

Memory location where machine code is stored	Machine code		Explanation of machine instruction
	OP code	Address	
000	09	100	Read from input a number and store it in 100, C(100) = I
001	09	110	Read from input unit a number and store in 110. C(110) = J
002	01	100	ACC = C(100) = I
003	03	110	ACC = ACC – C(110) = I – J
004	08	007	If ACC < 0 take next instruction from 007
005	10	100	Print C(100) (If I > J Print I)
006	11	000	Stop
007	10	110	Print C(110) (If J > I Print J)
008	11	000	Stop
– 000	00	000	End of machine language program

executed by the machine. Column 1 tells where this instruction is stored in memory. This information is required to write the program. For instance in the program of Table 12.2, the instruction 08 007 commands that if the contents of the accumulator is negative the next instruction to be executed should be taken from location 007. A programmer should thus know the instruction stored in location 007.

The last column in Table 12.2 contains comments to enable a reader to understand the program.

The machine language program is typed with the location of the instruction in the first three columns followed by a blank column, the operation code in the next two columns, a blank column and the operand address in the last three columns.

The machine loads the program in memory starting from location 000. The programmer has to remember this in writing this program and start his first instruction from location 000. He has to know, besides, exactly where each one of the machine instructions is stored in memory. The end of the machine language program is indicated by a data line with a negative number. The loading of the program is stopped when this data line is encountered.

12.3 AN ALGORITHM TO SIMULATE THE SMALL COMPUTER

An algorithm to simulate the operation of the computer is given as Algorithm 12.1. The algorithm is divided into two phases, namely, storing instructions in Phase I and interpreting and executing instructions in Phase II.

Algorithm 12.1 Algorithm to simulate SMAC

Phase I: Storing machine language instructions in memory

Instruction counter = 0.
Read a machine instruction
while end of machine language is not reached
{Store machine instruction in memory address given by instruction counter;
++Instruction counter;
Read a machine instruction; }
/*Control will reach this point as soon as all machine language instructions are stored*/

Phase II: Interpreting and executing machine language instructions

Instruction counter = 0. /* First instruction is in location 0 */
Repeat the following steps until the stop instruction of a machine language program is reached.

Step 2.1: Retrieve machine instruction from location given by instruction counter.
Step 2.2: ++Instruction counter.
Step 2.3: Branch to actions depending on operation code.

OP code 1: C(ACC) = C(ADDR)
OP code 2: C(ACC) = C(ACC) + C(ADDR)
OP code 3: C(ACC) = C(ACC) – C(ADDR)
OP code 4: C(ACC) = C(ACC) * C(ADDR)
OP code 5: C(ACC) = C(ACC)/C(ADDR)
OP code 6: C(ADDR) = C(ACC)
OP code 7: Instruction counter = ADDR
OP code 8: *if* C(ACC) < 0 *then* Instruction counter = ADDR
OP code 9: Read data and store in specified ADDR
OP code 10: Print contents of specified ADDR
OP code 11: Stop execution.

end of algorithm.

12.4 A SIMULATION PROGRAM FOR THE SMALL COMPUTER

A simulation program will now be written to simulate SMAC. The composition of the complete program is shown in Fig. 12.3. The program itself is given as Example Program 12.1. This program is essentially Algorithm 12.1 rewritten using C.

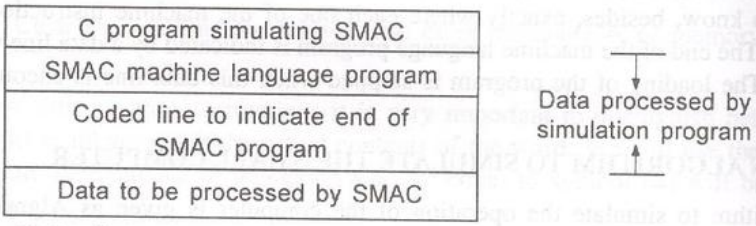

Fig. 12.3 Composition of program and data to use SMAC simulator.

Example Program 12.1 implements the two phases specified in the simulation algorithm (Algorithm 12.1). In phase I the machine language program is read and stored in memory starting from location 0. In phase II machine language instructions are retrieved from memory starting from location 0, the operation code is decoded and the specified operation performed. A *while* loop is used to read and store the machine language in the simulated memory. The end of machine language program is indicated by a dummy instruction with a negative number in its location field. Observe that as a data with a negative number in its location field is not a machine instruction it should not be stored in the simulated memory. Thus when we read such a data we should exit from the loop. Thus a *while* loop is useful in this case. Another *while* loop simulates the execution of machine instructions.

It is a good programming practice to introduce in the program statements to check when values assigned to variables go outside the specified range. When such an error is detected, informative messages should be printed along with the status of the important registers in the simulated computer. This has been done in the simulation program. If the address part of the instruction or the OP code or the instruction counter assume values outside the specified range then an appropriate message is printed and the program unconditionally jumps to the end of the program stopping further execution. A switch statement is ideally suited to decode an operation code and perform different actions depending on this code. This is done in the program. When the operation code for stopping the execution of SMAC is encountered the variable stop is made 1 which terminates the machine language program.

The status of all the important registers of SMAC in each execution cycle may be printed by inserting the statement:

```
printf ("%d%d%d%d\n", inst_counter, op_code, address, acc);
```

before the statement, namely, ++ inst_counter in the *while* loop. This would give what is known as the *execution trace* of the machine language program. This is very useful in debugging the machine language program. This is shown as a comment in Example Program 12.1.

In Program 12.1, we have used a statement goto abort_program. This is called an unconditional jump. When this statement is executed, control jumps to a statement with label abort_program. Such an unconditional jump is used only when an abnormal situation is encountered and a program comes to halt. The goto statement is rarely used in C. Observe that in Program 12.1, the label abort_program transfers control to the end of the program and no action is carried out.

```
/* Example Program 12.1 */
/* Simulation of a small computer SMAC */

#include <stdio.h>
main()
{
  int inst_counter = 0, address, op_code, instrn,
  ins_reg, stop = 0, acc = 0, location, memory[1000];

  /* The following loop reads SMAC machine language
     program given as data to this simulator and
     stores it in SMAC's memory */

  scanf("%d %d %d", &location, &op_code, &address);
  while(location >= 0)
    {
      instrn = op_code * 1000 + address;
      memory[inst_counter] = instrn;
      ++inst_counter;
      if (inst_counter > 999)
        {
          printf("Program overflows memory\n");
          goto abort_program;
        }
       scanf("%d %d %d", &location, &op_code,
             &address);
    } /* storing of SMAC program in memory over */

  /* The machine language instruction is retrieved
     one by one from SMAC's memory, interpreted
     and executed by the following part of the
     simulator program */

  inst_counter = 0;
  while(stop == 0) /* observe that stop is initialised too */
    {
      ins_reg = memory[inst_counter];
      op_code = ins_reg / 1000;
      if ((op_code <= 0) || (op_code > 11))
        {
          printf("Illegal op_code\n");
          printf("Inst_counter = %d, op_code = %d\n",
                 inst_counter, op_code);
          goto abort_program;
        }
      address = ins_reg % 1000;
   /*   printf("%d %d %d %d\n", inst_counter,
        op_code, address, acc);  */
      ++inst_counter;

      switch(op_code)
        {
```

```
case 01:
  /* Move contents of specified address
     in memory to accumulator */
  acc = memory[address];
  break;
case 02:
  /* Add contents of specified address
     in memory to accumulator */
  acc += memory[address];
  break;
case 3:
  /* Subtract from accumulator */
  acc -= memory[address];
  break;
case 4:
  /* Multiply contents of accumulator */
 acc *= memory[address];
  break;
case 5:
  /* Divide contents of accumulator
     before dividing check if
     dinominator is zero */
  if (memory[address] == 0)
    {
      printf("Attempt to divide by zero\n");
      printf("Ins = %d, op = %d, add = %d\n",
             --inst_counter, op_code,
             address);
      goto abort_program;
    }
  else
    acc /= memory[address];
  break;
case 6:
  /* Store accumulator in memory */
  memory[address] = acc;
  break;
case 7:
  /* Unconditional jump */
  inst_counter = address;
  break;
case 8:
  /* Conditional jump */
  if (acc < 0)
    inst_counter = address;
  break;
case 9:
  /* Read data into memory of SMAC */
  scanf("%d", &memory[address]);
  break;
```

```
            case 10:
              /* Result stored in memory is displayed in VDU */
              printf("%d\n", memory[address]);
              break;
            case 11:
              /* Machine language stop statement */
              stop = 1;
              break;
            } /* Closing braces of switch statement */
        } /* Closing braces of while loop */

    printf("Program has normal ending instruction\n");
    printf("Inst counter = %d\n", --inst_counter);
  abort_program: ; /* Null statement. Do nothing */
} /* End of main */
```

Program 12.1 Simulation of a small computer.

12.5 A STATISTICAL DATA PROCESSING PROGRAM

We will consider in this section a typical statistical data processing problem. In this problem a table of the marks obtained by each student in a class along with his/her average marks and division is required. Besides this, the average performance of the class in each subject and some statistical measures relating the performance of the class in different subjects are required. A program for obtaining these will be developed in this section.

The problem specifications are given below:

(i) The class has 100 or less students

(ii) Each student takes 6 subjects, namely, Physics, Chemistry, Mathematics, Engineering Science, Technical Arts and Humanities.

(iii) The maximum marks in each subject is 100.

(iv) A student passes in the first division if he or she gets more than 60 marks on the average, in the second division if he or she gets an average between 50 and 59, in the third division if he or she gets between 40 and 49 and fails if his or her average marks is below 40.

(v) The program should compute the average marks and the division of each student.

(vi) It is required to find the performance of the class as a whole in each subject. This is indicated by the class average and the standard deviation of the marks in each subject.

(vii) It is also required to correlate the performance of the class in different subjects. One wants to draw conclusions such as "if a class does well in Mathematics it also does well in Physics", "the performance of a class in Humanities is uncorrelated with its performance in Mathematics". To draw such conclusions a statistical measure called the correlation coefficient is used. The program should compute the matrix of correlation coefficients relating the performance of students in various subjects. The class average, standard deviation, covariance and correlation coefficient are given by the following equations:

The class average in a subject is given by:

$$\overline{m}_i = \frac{1}{N}(\Sigma m_{ik}) \qquad k \text{ varying from 1 to } N \tag{12.1}$$

where m_{ik} is the marks obtained by the k^{th} student in the i^{th} subject and N is the total number of students in the class.

The standard deviation of the marks obtained by students in subject *i* is given by:

$$s_i = \frac{1}{N}(\Sigma(m_{ik} - \overline{m}_i)^2)^{0.5} \qquad \text{k varying from 1 to N} \qquad (12.2)$$

The covariance between the marks obtained by students in subject *i* and subject *j* is defined by the equation

$$c_{ij} = \frac{1}{N}\Sigma(m_{ik} - \overline{m}_i)(m_{jk} - \overline{m}_j) \qquad \text{k varying from 1 to N} \qquad (12.2)$$

$$= \frac{1}{N}\Sigma(m_{ik}\, m_{jk} - \overline{m}_i\, \overline{m}_j) \qquad (12.3)$$

From Eq.12.3 above it is seen that

$$c_{ii} = \frac{1}{N}(\Sigma(m_{ik})^2 - (\overline{m}_i)^2) \qquad \text{k varying from 1 to N} \qquad (12.4)$$

The quantity c_{ii} is called the variance of the marks in subject *i* and is the square of the standard deviation. In the computer c_{ij} will be computed and c_{ii} obtained as a byproduct by setting j = i.

The gross structure of the procedure may be stated as:

Algorithm 12.2 Procedure to compute class statistics

while student records are not yet over do
{ Read a student record
Accumulate count of students
Find total marks of the student
Find average marks of the student
Find the division of the student based on the average marks
Accumulate marks of students in each subject
Accumulate products of marks in pairs of subject.
Print Roll no., marks in each subject, total marks,
average marks and division of the student }

Calculate class average marks for each subject based on accumulated totals and student count.
Calculate covariance of marks in pairs of subjects based on accumulated cross products.
Calculate standard deviation and correlation coefficients

Print all the calculated values

end of algorithm

This algorithm is implemented as a C program of Example Program 12.2. The program is a straightforward implementation of the algorithm. It has a series of printf statements after the main computation is performed. These statements are needed to present the output in a "good looking" format. Developing these printf statements mainly require proper counting of columns and attention to detail.

A sample output of Example Program 12.2 is shown below the program to enable the reader to correlate the printf statements with the printed output.

```
/* Example Program 12.2 */
/* Processing of students results */
#include <stdio.h>
#include <math.h>
main()
{
  int total_stud = 0, marks_total, i, j, division,
  roll_no, avge_marks, marks[6], subj_avge[6],
  total_subj[6], std_dev[6], cross_product[6][6],
  covnce[6][6];
  while(scanf("%d", &roll_no) != EOF)
    {
      for(i = 0; i <= 5; ++i)
        scanf("%d", &marks[i]);
      ++total_stud;
      marks_total = 0;
      for(i = 0; i <= 5; ++i)
        marks_total += marks[i];
      avge_marks = (float)marks_total/6.0 + 0.5;
      if (avge_marks >= 60)
        division = 1;
      else
        {
          if (avge_marks >= 50)
            division = 2;
          else
            {
              if (avge_marks >= 40)
                division = 3;
              else
                division = 0;
            }
        }
      printf("%d ", roll_no);
      for(i = 0; i <= 5; ++i)
        printf("%d  ", marks[i]);
      if (division == 0)
        printf("%d Failed\n", avge_marks);
      else
        printf("%d Passed in Division %d\n",
               avge_marks, division);
      /* Initialisation of arrays */
      for (i = 0; i <= 5; ++i)
        {
          total_subj[i] = 0;
          for (j = 0; j <= 5; ++j)
          cross_product[i][j] = 0;
        }
```

```
      for (i = 0; i <= 5; ++i)
        {
          total_subj[i] += marks[i];
          for (j = 0; j <=5; ++j)
            cross_product[i][j] +=
              marks[i] * marks[j];
        }
    } /* End of while loop */

printf("\n                    Class Averages\n\n");
for(i = 0; i <= 5; ++i)
  subj_avge[i] = total_subj[i]/total_stud;
printf("                  Phys  Chem  Math");
printf("   T.A.  E.S   H.S.\n");
printf("  Averages");
printf("      %4d %5d %5d %5d %5d %5d\n",
       subj_avge[0], subj_avge[1],
       subj_avge[2], subj_avge[3],
       subj_avge[4], subj_avge[5]);
for(i = 0; i <= 5; ++i)
  for (j = 0; j <=5; ++j)
    covnce[i][j] = cross_product[i][j] /
      total_stud - subj_avge[i]*subj_avge[j];

for (i = 0; i <=5; ++i)
  std_dev[i] = sqrt((float)covnce[i][i]) + 0.5;
printf("  Std. Dev.");
printf("     %4d %5d %5d %5d %5d %5d\n",
       std_dev[0], std_dev[1],
       std_dev[2], std_dev[3],
       std_dev[4], std_dev[5]);
printf("\n          The Variance Covariance ");
printf(" Matrix\n\n");
printf("      Phys    Chem    Math    T.A.");
printf("   E.S     H.S.\n");
printf("Phys  ");
for (j = 0; j <=5; ++j)
  printf("%-8d", covnce[0][j]);
printf("\n");
printf("Chem  ");
for (j = 0; j <=5; ++j)
  printf("%-8d", covnce[1][j]);
printf("\n");
printf("Math  ");
for (j = 0; j <=5; ++j)
  printf("%-8d", covnce[2][j]);
printf("\n");
```

```
  printf("T.A.   ");
  for (j = 0; j <=5; ++j)
    printf("%-8d", covnce[3][j]);
  printf("\n");
  printf("E.S.   ");
  for (j = 0; j <=5; ++j)
    printf("%-8d", covnce[4][j]);
  printf("\n");
  printf("H.S.   ");
  for (j = 0; j <=5; ++j)
    printf("%-8d", covnce[5][j]);
  printf("\n");
} /* End of main */
```

Input:

```
1234  30  40  50  60  70  80
1235  40  30  40  30  40  34
1236  70  60  60  50  60  65
1238  50  35  45  35  40  40
1246  40  60  50  70  60  80
1254  30  30  60  50  35  45
```

Output:

```
1234 30  40  50  60  70  80  55 Passed in Division 2
1235 40  30  40  30  40  34  36 Failed
1236 70  60  60  50  60  65  61 Passed in Division 1
1238 50  35  45  35  40  40  41 Passed in Division 3
1246 40  60  50  70  60  80  60 Passed in Division 1
1254 30  30  60  50  35  45  42 Passed in Division 3
```

Class Averages

	Phys	Chem	Math	T.A.	E.S	H.S.
Averages	43	42	50	48	50	57
Std. Dev.	15	14	12	14	16	20

The Variance Covariance Matrix

	Phys	Chem	Math	T.A.	E.S	H.S.
Phys	217	135	75	-16	75	25
Chem	135	206	95	146	175	217
Math	75	95	137	95	100	110
T.A.	-16	146	95	203	175	260
E.S.	75	175	100	175	254	289
H.S.	25	217	110	260	289	385

Program 12.2 Processing students results.

12.6 PROCESSING SURVEY DATA WITH COMPUTERS

A frequent use of digital computers is in the tabulation of the results of surveys. The general principles involved in planning the collection of survey data which are to be ultimately processed by digital computers and the programming aspects may be illustrated by considering an example. The example chosen is the processing of questionnaires (distributed to the participants of a short course on computers) shown as Table 12.3.

Table 12.3 A Sample Questionnaire

Please record the answers to the following questions by entering the code number corresponding to your choice in the box on the right of each question.

			Column Number
	Serial Number		1
	(For Office use only)		2
			3
1.	What is your sex? Male = 0 Female = 1		4
2.	Age:What was your age at the last birthday? If it is, say, 25 enter it as :		
		2	5
		5	6
3.	What is your institutional affiliation?		7
	Private Sector	0	
	Educational Institution	1	
	Government Office	2	
	Public Sector Firm	3	
	Research Laboratory	4	
	Unemployed	5	
4.	What is your primary interest?		8
	Science	0	
	Engineering	1	
	Mathematics	2	
	Social Sciences	3	
	Business Data Processing	4	
5.	What is the highest degree obtained by you?		9
	High School	0	
	Intermediate	1	
	Bachelor's	2	
	Master's	3	
	Doctor's	4	

The objectives of the survey are to obtain the following tables:

(i) The distribution of participants by sex

(ii) The average age of the participants

(iii) The distribution of the participants by institutional affiliation

(iv) The distribution of the participants as a function of their primary interest

(v) The distribution of the participants as a function of their educational background

(vi) A table giving the number of participants with specified interests from different types of institutions.

After deciding on the number of groups of each type into which the participants are to be divided it is necessary to uniquely code each group. The division into groups is obvious if sex is the distinguishing characteristic. In this case there are two groups and each one is given a code number. A code 0 is assigned to males and 1 to females. As our aim is to count the number of males and females respectively among the participants it is conveniently done by taking the participants' sex as an array with two components. The first component corresponds to males and the second to females. If the participants are to be grouped using another characteristic, say their educational background, then this information may be coded into a set of integers. These codes may be thought of as subscripts which group the participants into distinct classes.

The main idea used in processing survey questionnaires may be illustrated by considering the following example in which the participants are grouped into two groups, males and females. Assume that one data is typed per participant and that it has a serial number in columns 1 to 3 and the sex code (0 or 1) in column 5. The program to divide the participants into two groups (namely, males and females) is given as Example Program 12.3. In this

```
/* Example Program 12.3 */
/* Questionnaire tabulation - Program Development */
#include <stdio.h>

main()
{
  int sex[2], sex_code, ser_no, bad_data = 0, no_quest = 0;
  sex[0] = sex[1] = 0;
  while(scanf("%d %d", &ser_no, &sex_code) != EOF)
    {
      if ((sex_code < 0) || (sex_code > 1))
        {
          ++bad_data;
          printf("Error in Sex code Ser no. = %d\n",
                 ser_no);
        }
      else
        {
          ++sex[sex_code];
          ++no_quest;
        }
    } /* End while */
  printf("No. of questionnaires tabulated = %d\n",
         no_quest);
  printf("No. of invalid data = %d\n", bad_data);
  printf("No. of Males = %d, No. of Females = %d\n",
         sex[0], sex[1]);
} /*  End of main */
```

Program 12.3 Questionnaire program development.

program the "bin" sex [0] is set up to count data lines which have sex_code = 0 and "bin" sex[1] to count those with sex_code = 1. After clearing sex[0] and sex[1] to zeros a data is read. If the code is less than 0 or greater than 2 then it is illegal. In other words, the data has been wrongly typed. This is indicated in the program and the number of such bad data are counted. For correct data if sex_code = 0, sex[0] is incremented by 1 and if sex_code = 1 then sex[1] is incremented by 1. Thus the value of the sex_code "sorts" the data into appropriate "bins". Extension of this technique to the case when the number of groups is greater than 2 is obvious. A program to process the questionnaire given at the beginning of this section is given as Example Program 12.4. The student is urged to study this carefully. It will be

```
/* Example Program 12.4 */
/* Questionaire tabulation - one and two way
   tables */

#include <stdio.h>
main()
{
  int sex_code, inst_code, intrt_code, degree_code,
  age, sex[2], degree[5], institution[6],
  interest[5], inst_vs_intrt[6][5], sum_age = 0,
  avg_age, serial_no, bad_data = 0, no_quest = 0, i, j,
  inp_error;
  sex[0] = sex[1] = 0;
  for (i = 0; i <= 4; ++i)
    degree[i] = interest[i] = 0;
  for (i = 0; i <= 5; ++i)
    {
      institution[i] = 0;
      for (j = 0; j <= 4; ++j)
        inst_vs_intrt[i][j] = 0;
    }
  printf("Output Tables\n");
  while(scanf("%d %d %d %d %d %d", &serial_no,
              &sex_code, &age, &inst_code,
              &intrt_code, &degree_code) != EOF)
    {
      inp_error = 0;
      if ((sex_code < 0) || (sex_code > 1))
       {
         inp_error = 1;
         printf("Error in sex_code Ser no = %d\n"
                serial_no);
       }
      if ((age < 0) || (age > 99))
       {
         inp_error = 1;
```

```
            printf("Error in age Ser no = %d\n",
                  serial_no);
          }
        if ((inst_code < 0) || (inst_code > 5))
          {
            inp_error = 1;
            printf("Error in inst_code Ser no = %d\n",
                  serial_no);
          }
        if ((intrt_code < 0) || (intrt_code > 4))
          {
            inp_error = 1;
            printf("Error in intrt_code Ser no = %d\n",
                  serial_no);
          }
        if ((degree_code < 0) || (degree_code > 4))
          {
            inp_error = 1;
            printf("Error in degree_code Ser no = %d\n",
                  serial_no);
          }
        if (inp_error == 1)
          ++bad_data;
        else
          {
            ++sex[sex_code];
            ++institution[inst_code];
            ++interest[intrt_code];
            ++degree[degree_code];
            sum_age += age;
            ++no_quest;
            ++inst_vs_intrt[inst_code][intrt_code];
          }
      } /* End of while loop */

    avg_age = sum_age/no_quest;
    printf("No. of valid Questionnaires = %d\n",
          no_quest);
    printf("No. of invalid Questionnaires = %d\n",
          bad_data);
    printf("Average Age of Participants = %d\n",
          avg_age);
    printf("Distribution of Participants by Sex\n");
    printf("Males = %d, Females = %d\n",
          sex[0], sex[1]);
    /* The notation within printf has been split
```

```
        on two lines so that the program width
        does not exceed the page width of this book */

      printf("Distribution of Participants by");
      printf(" Institutional\nAffiliation\n");
      printf("   Pvt.  Edn.  Govt  PubS  Res.  Unemp\n");
      printf("    ");
      for (i = 0; i <= 5; ++i)
        printf("%-6d", institution[i]);
      printf("\n");
      printf("Distribution of Participants by");
      printf(" Interest\n");
      printf("   Sci.  Engg  Math  Soc.  B.D.P.\n");
      printf("    ");
      for (i = 0; i <= 4; ++i)
        printf("%-6d", interest[i]);
      printf("\n");
      printf("Distribution of Participants by");
      printf(" Qualifications\n");
      printf("   H.Sc  Int.  Bach  Mas.  Ph.D\n");
      printf("    ");
      for (i = 0; i <= 4; ++i)
        printf("%-6d", degree[i]);
      printf("\n");
      printf("Institutional Affiliation Vs. interest\n");
      printf("           Pvt.  Edn.  Govt  PubS  Res.  Unemp\n");
      printf("Science     ");
      for (j = 0; j <= 5; ++j)
        printf("%-6d", inst_vs_intrt[0][j]);
      printf("\n");
      printf("Engineer    ");
      for (j = 0; j <= 5; ++j)
        printf("%-6d", inst_vs_intrt[1][j]);
      printf("\n");
      printf("Math        ");
      for (j = 0; j <= 5; ++j)
        printf("%-6d", inst_vs_intrt[2][j]);
      printf("\n");
      printf("Soc.Sc.     ");
      for (j = 0; j <= 5; ++j)
        printf("%-6d", inst_vs_intrt[3][j]);
      printf("\n");
      printf("B.D.P.      ");
      for (j = 0; j <= 5; ++j)
       printf("%-6d", inst_vs_intrt[4][j]);
      printf("\n");
      printf("-- End of Tables --\n");
    } /* End of main */
```

```
Input:

124  0  25  3  4  3
128  1  35  4  6  3
129  1  45  2  1  2
130  0  38  3  2  1
131  1  28  1  1  0
132  0  30  0  0  0
133  1  45  1  2  3
134  1  45  6  5  4
140  1  35  4  5  4
141  0  65  5  4  4
142  0  102  3  3  4

Output:
Output Tables
Error in intrt_code Ser no = 128
Error in inst_code Ser no = 134
Error in intrt_code Ser no = 134
Error in intrt_code Ser no = 140
Error in age Ser no = 142
No. of valid Questionnaires = 7
No. of invalid Questionnaires = 4
Average Age of Participants = 39
Distribution of Participants by Sex
Males = 4, Females = 3
Distribution of Participants by Institutional
Affiliation
   Pvt.  Edn.  Govt  PubS  Res.  Unemp
    1     2     1     2     0     1
Distribution of Participants by Interest
   Sci.  Engg  Math  Soc.  B.D.P.
    1     2     2     0     2
Distribution of Participants by Qualifications
   H.Sc  Int.  Bach  Mas.  Ph.D
    2     1     1     2     1
Institutional Affiliation Vs. interest
           Pvt.  Edn.  Govt  PubS  Res.  Unemp
Science     1     0     0     0     0     0
Engineer    0     1     1     0     0     0
Math        0     1     0     0     0     0
Soc.Sc.     0     0     1     0     1     0
B.D.P.      0     0     0     0     0     0
-- End of Tables --
```

12.4 Questionnaire tabulation.

observed that the programming job by itself is very simple. Most of the work is in obtaining proper formats for spacing, headings, etc.

EXERCISES

12.1 Write a machine language program for SMAC which will pick the largest of 10 numbers. Test this program with the simulator.

12.2 Write a machine language program for SMAC which will add two 10 component vectors.

12.3 It is desired to add more instructions to SMAC. Some of the machine instructions and their meanings are given below. Rewrite SMAC with these additions and test it.

OP code: 21 Instruction form: 21XXX
Meaning: C(ACC) = C(ACC) + XXX

OP code: 22 Instruction form: 22XXX
Meaning: C(ACC) = C(ACC) – XXX

OP code: 31 Instruction form: 31XXX
Meaning: Shift C(ACC) right by XXX positions.

12.4 Income-tax rules define three categories of persons for tax computation:

(i) Residents
(ii) Resident but not ordinarily resident
(iii) Non-resident.

Definition of Resident:
If a person satisfies any one of the following rules he is considered a resident during a specified year.

(i) He lived in India for at least 182 days during the year;
(ii) He maintained a home in India for at least 182 days and lived for at least 30 days in India during the year;
(iii) He lived in India in the four preceding years for at least 365 days and lived in India for 60 days during the given year.

Definition of Resident but not ordinarily resident:
A person is considered ordinarily resident in India during a year if he satisfies both the conditions given below in addition to being a resident during the year:

(i) If during preceding 10 years he is a resident of India for at least 9 years.
(ii) If during preceding seven years he lived in India for a total of 760 days or more.

If he does not satisfy any one of the conditions above he is considered a resident but not ordinarily resident.

Non-resident
A person who is not resident in India is called non-resident.

(i) Obtain decision tables to decide into which category a person falls given the required data.
(ii) How many previous years' data are required to decide about the status of a person in a given year?
(iii) Write a computer program which prints out the category to which a person belongs given the required data.

12.5 A sociologist conducts a survey among college students and collects the following data:

1. Age
2. Sex
3. Marital status
4. No. of years in college
5. Percentage marks obtained in the last examination
6. Time spent in studies/week
7. Time spent in extra-curricular activities/week
8. Financial support (Parents/guardian/government/charitable organization/other). (Here the student may get support from more than one source).
9. No. of cinemas seen per month
10. Opinion about usefulness of education (very useful/doubtful/useless).

(i) Prepare a code to convert the information to one which would be suitable for processing on a computer.

(ii) Write a program to tabulate the results.

12.6 A hospital keeps a file of blood donors in which each record has the format:

Name	:	20 columns
Address	:	40 columns
Age	:	2 columns
Blood Type	:	1 column (Type 1, 2, 3 or 4)

Write a program to print out all blood donors whose age is below 25 and blood is type 2.

13. Functions

Learning Objectives

In this chapter we will learn:

1. The need to break up a large program into a number of functions
2. Defining functions in C language
3. How functions are called from other functions
4. Use of arrays as arguments in functions
5. The concept of local and global variables

13.1 INTRODUCTION

Functions are subprograms which perform well defined tasks. They may be developed separately and tested. We may then name each such subprogram and specify the method of sending data to each of them and getting processed results from them. Once this is done these subprograms may be linked together to do a given task. The primary merits of this approach to program development are:

(i) It is a good idea to break up a big job into a number of smaller sub-jobs. Divide and conquer, namely, breaking up a complicated problem into smaller parts and solving each separately is a well known problem solving method.

(ii) A program written as a sequence of functions with each function carrying out a specified task, is easy to understand. An understandable program can be modified easily.

(iii) It is easy to test small compact functions.

(iv) Commonly used functions may be generalised, tested thoroughly and kept in a library for future use.

(v) Functions developed by others may be used. For example, a large number of efficient, well tested programs for common problems in string manipulation and numerical computation, are available as libraries usable by C programmers.

(vi) In languages other than C, particularly FORTRAN, many good program libraries are universally available. For instance IMSL (International Mathematical and Statistical Library) and NAG (Numerical Algorithm Group) are well known libraries. Many C compiler implementations permit declaration of FORTRAN programs as external procedures and use them in C programs. Recently many of the commonly available libraries in FORTRAN are being rewritten in C.

A C language program consists of a collection of functions. The programs written so far define a single function main( ). We can break up a program into many functions. main( )

can use all these functions to carry out the required processing. We will illustrate the definition and use of functions with simple examples in the next section.

13.2 DEFINING AND USING FUNCTIONS

A *function* is defined in Example Program 13.1 to reverse a given integer. Given a integer 4578 the reverse of this integer is 8754. This *function* is:

int reverse_of (int n)

```
/* Example Program 13.1 */
/* Illustrates defining a function */
int reverse_of(int n)
{
  int digit, rev_of_n = 0;     /* Local Variables */
  while(n != 0)
    {
      digit = n % 10;
      rev_of_n = rev_of_n * 10 + digit;
      n = n/10;
    }
  return(rev_of_n);
}
```

Program 13.1 Defining a function.

The *name* of the *function* is reverse_of. The rules to name a *function* are the same as those used to name an identifier. This *function* is declared to be of type *int*. This says that the result which will be computed and returned by the *function* when it is invoked will be an integer. The variable name inside the parentheses is known as a *formal argument* or *formal parameter* of the *function*. This argument is used by the *function* as input for processing. The type of this variable name should be specified. Observe that no semicolon is used after the function declaration. The statements to be executed by the function *defines* the *function* and is specified next. The *function* definition is enclosed within braces { } and is known as the body of the *function*. The *function* definition begins with the declaration of the variables and their types which will be used in the *function*. These variables are *local* to the *function*. They have no meaning outside the *function* body. This implies that the same names may be used by other *functions* in their definition without any conflict. This is essential as *functions* may be defined and written by different programmers and they should be able to write their programs independently. The local nature of these variables also implies that outside the function their contents cannot be used.

After the declaration a sequence of statements are written which defines the task carried out by the *function*. After the task is carried out the result is sent back by the statement:

return (expression) ;

In this statement *expression* is any legal C expression.

Referring again to Example Program 13.1 the local variables of the *function* reverse_of are: digit and rev_of_n which are both integers. Observe that rev_of_n is initialised to 0. The *while* loop forms the rev_of_n. Outside the *while* loop the statement:

return (rev_of_n) ;

sends back the calculated integer to the *function*

int reverse_of (int n);

In Example Program 13.2 we illustrate how the *function* is *invoked* or *called* by main. Observe that in the listing of Example Program 13.2 after #include <stdio.h> we have written:

int reverse_of (int number);

```
/* Example Program 13.2 */
#include <stdio.h>
/* Function Prototype */
int reverse_of(int number);

main()
{
  int p, q;
  scanf("%d", &p);
  q = reverse_of(p);  /* Function called here */
                      /* p is the actual argument */
  printf("Reverse of %d is %d\n", p, q);
}

int reverse_of(int n)
{
  int digit, rev_of_n = 0;

  while(n != 0)
    {
      digit = n % 10;
      rev_of_n = rev_of_n * 10 + digit;
      n = n/10;
    }
  return(rev_of_n);
}
```

```
Input:
47632

Output:
Reverse of 47632 is 23674
```

Program 13.2 Reversing an integer.

This is called a *function prototype* and is required in ANSI C. The prototype tells the C compiler that a *function* named *reverse_of* of *type* integer which has one argument which is also of type integer will be later defined in the program. This information will be used to check both the definition of the *function* and when the *function* is actually called. The specification that the *function* is of type integer tells that the value returned by the *function* will be of *type* integer. The function prototype must be terminated by a semicolon.

The *main function* is now defined. It reads an integer and stores it in variable name *p*. In the next statement q = reverse_of(p); the appearance of the *function* name reverse_of *invokes* or *calls* the *function* reverse_of with actual argument *p* and the value returned by the *function* is stored in *q*. When *function* reverse_of is called with actual argument *p* the value of *p* is *copied* into the *formal argument n* of the *function*, the original value stored in *p* is unaffected. The *function* now uses the value of *p* and does the computations. Observe that if *p* = 2578, *n* is assigned the value 2578. Observe that the *function* destroys the value of *n* in the process of reversing it and *n* ultimately becomes zero. This does not affect the value of *p* in the calling function, namely, main( ). This fact is very important to remember.

At the end of the *function* reverse_of the value of reverse of *n* is returned by it to the calling *function*, namely, main( ). The returned value is stored in *q* by the statement:

q = reverse_of(p);

in main( ).

Finally the values of *p* and its reverse *q* are printed. The input and output of this program are also shown along with Example Program 13.2.

In this program the *function* has one formal argument of type int and an integer value was returned by the *function*. In general a *function* may have any number of formal arguments of any type, e.g., float, unsigned int etc. A *function* may have no argument. For example, main( ) has no arguments in all the programs we have written so far.

A *function* may return *at most one result*. Thus a *function* may either return no value or return one value. The value returned may be of any type int, float, unsigned int etc. A *function* which returns no value is defined as:

void function_name (*type* formal arguments)

void is a key word indicating that no value is returned. Similarly if a *function* has no formal arguments it will have the word *void* within parentheses as shown below:

type function_name(*void*)

In Example Program 13.3 we illustrate the use of *functions* with *void* for type names and int for formal argument, observe that the function:

void print_palindrome (int k)

does not return a value. It merely uses the value passed on to it by the actual argument to print a message.

We give below some more examples.

Example 13.1

The definition and use of a *function* to add the digits of an integer is given as Example Program 13.4. Observe that the *function* prototype is given first. The *function* main( ) reads a number *p* and calls the *function* sum_digits with *p* as actual argument. The *function* sum_digits is defined next after main( ). This *function* uses *n* as the formal argument. It has two *int* variables sum and digit used locally within the *function*. In the body of the *function* sum_digits is calculated and returned as sum. This returned value is assigned to the variable *s* in the *function* main( ). The main( ) *function* then prints the value of the actual argument communicated to sum_digits (namely, *p*) and the value returned by the function (namely, *s*).

```
/* Example Program 13.3 */
#include <stdio.h>
int reverse_of(int number);
void print_palindrome(int x);
void print_not_palindrome(int x);
main()
{
 int p;
 while (scanf("%d", &p) != EOF)
   {
     if (p == reverse_of(p))
       print_palindrome(p),
     else
       print_not_palindrome(p);
   }
} /* End of main */

int reverse_of(int n)
{
  int digit, rev_of_n = 0;
  while(n != 0)
    {
      digit = n % 10;
      rev_of_n = rev_of_n * 10 + digit;
      n = n/10;
    }
  return(rev_of_n);
}

void print_palindrome(int k)
{
  printf("Given number %d is a Palindrome\n", k);
}

void print_not_palindrome(int k)
{
  printf("Given number %d is not a Palindrome\n", k);
}
```

Program 13.3 Checking if an integer is a palindrome.

Example 13.2

Consider the *function* defined below:

$$\begin{aligned} &\text{limiter } (x) = \text{bound} && x >= \text{bound} \\ &\text{limiter } (x) = x && \text{bound} > x > -\text{bound} \\ &\text{limiter } (x) = -\text{bound} && x <= -\text{bound} \end{aligned} \qquad (13.1)$$

```
/* Example Program 13.4 */
/* Defining and using a function to add digits of
   a number */
#include <stdio.h>
int sum_digit(int number);

main()
{
  int p, s;
  scanf("%d", &p);
  s = sum_digits(p);
  printf("Sum of digits of %d is %d\n", p, s);
}

/* Function defining sum of digits */
int sum_digits(int n)
{
  int sum = 0, digit;
  if (n < 0)
    n = -n;
  while (n != 0)
    {
      digit = n % 10;
      sum += digit;
      n = n/10;
    }
  return(sum);
}
```

Program 13.4 Adding digits of an integer.

This *function* is defined in Example Program 13.5 as

float limiter (float x, float bound)

Observe that this *function* has two arguments which are both of *type float*. The *function* also returns a result of *type float*. The *function* has more than one *return* statement. This is necessary in this example and is allowed in C.

Before main( ) there is a function prototype defining *limiter*. Observe that the *main function* calls the *function limiter* thrice to compute.

z = (limiter(a, c) + limiter(b, c))/limiter (a + b, c);

Observe that we have used an expression a + b as one of the arguments. This is allowed.

Example 13.3

A bank gives simple interest at *r* percent per year if the balance in one's account is Rs. 1000/- or more. No interest is given on balance less than Rs. 1000/-. It is required to find the balance in a customer's account at the end of *x* years.

The independent variables in this example are interest rate, balance in one's account

```
/* Example Program 13.5 */
/* Defining and using a function limiter */
#include <stdio.h>
float limiter(float p, float q);

main()
{
  float a, b, c, z;
  scanf("%f %f %f", &a, &b, &c);
  z = (limiter(a, c) + limiter(b, c)) /
    limiter(a+b, c);
  printf("value of a = %f b = %f c = %f z = %f\n",
         a, b, c, z);
}

float limiter(float x, float bound)
{
   if (x < -bound)
     return(-bound);
   else
     {
       if (x > bound)
         return(bound);
       else
         return(x);
     }
 }  /* End of function limiter(x, bound) */
```

Program 13.5 Simulating a limiter.

and the number of years of deposit. The quantity to be calculated based on these independent variables is the total interest amount.

Thus the *function* in this case is the interest and the formal arguments are: Interest rate, the balance in one's account and the number of years of deposit. A *function* which calculates the interest is given as Example Program 13.6.

```
/* Example Program 13.6 */
/* Function to calculate interest */

float interest(float rate, float balance, int years)
{
  float calc_int;
  if (balance > 1000.0)
    calc_int = (balance * rate/100.0)*(float)years;
  else
    calc_int =  0.0;
  return(calc_int);
}
```

Program 13.6 Interest calculation.

The *function* defined in Example Program 13.6 uses the specified value of Rs. 1000/- of minimum balance within the code. It can be generalized if the minimum_balance is also taken as a formal argument in the definition of the *function*. As the purpose of a *function* is to allow its flexible use by a class of users it is worthwhile to make it as general as possible. The generalized *function* and its use by a *main function* is illustrated in Example Program 13.7. Observe in Example Program 13.7 the definition of *function*

```
/* Example Program 13.7 */
/* Generalised function to calculate interest */
#include <stdio.h>
float interest(float rate, float balance,
               int main_bal, int years);
/* Function prototype defined above */

main()
{
  int min_bal, yrs, acct_no;
  float bal, rate, int_calc, new_balance;

  printf("Acc. no.  old balance    interest");
  printf("   new balance\n");
  while(scanf("%d %f %f %d %d", &acct_no, &rate,
              &bal, &yrs, &min_bal) != EOF)
    {
      int_calc = interest(rate, bal, min_bal, yrs);
      new_balance = bal + int_calc;
      printf(" %5d %12.2f %12.2f %12.2f\n",
             acct_no, bal, int_calc, new_balance);
    } /* End of while */
} /* End of main */

float interest(float rate, float balance,
               int min_balance, int years)
{
  int calc_int;
  if (balance > min_balance)
      calc_int = (balance * rate/100.0) *
        (float)years;
  else
    calc_int = 0.0;
  return(calc_int);
}
```

Program 13.7 Generalizing interest calculation.

prototype at the beginning. The main *function* calls interest in a *while* loop to calculate the interest of many account holders of the bank. The local variables are declared within main( ) as required.

13.3 SYNTAX RULES FOR FUNCTION DECLARATION

The general form of a *function* declaration is given below:

```
type function name(type a1, type a2, ... , type an)
{ type v1, v2, ... , vn; /* local variables */
        (one or more statements to compute the function)
        return ( expression );
}
```

The first statement is the *function* declaration and defines the name of the *function* and the list of formal arguments. This statement must *not* be terminated by a semicolon. The type of the value returned by the *function* is given as *type function*. The type of each of the formal arguments is also defined. A *function* may not return any value. In such a case the word *void* is used instead of type. If a function has no formal arguments then the word *void* is used instead. Following this the beginning of the *function* definition is indicated by left braces {. Next the local variables used by the *function* are declared.

The *body* of the *function* defines the computations to be carried out by the *function*. The *function* body may contain one or more *return* statements. A *return* statement returns a value to the calling *function*. A *return* statement may not be present if no value is returned. A closing brace } is used to indicate the end of the function definition.

The following rules must be remembered while defining functions:

(i) A *function* prototype declaration must appear at the beginning of the program following #include statement. In a *function* prototype the type of the *function* and the type of each formal argument must be specified. The variable names used for formal arguments in a *function* prototype are arbitrary; they may even be blanks. Each function prototype declaration is terminated by a semicolon.

(ii) A *function* name may be any valid identifier. The *type* of the *function* specifies the *type* of quantity which will be returned to the calling program.

(iii) The formal arguments used in defining a *function* may be scalars or arrays. Each *formal* argument *type* must be specified.

(iv) Within a *function* definition other variables may be declared and used. Such variables are *private* or *local* to the *function* and are not known outside the body of the function. These variables are known as *local* variables. Their values are undefined outside the function.

(v) Within a function definition one may *not define* another function.

A *function* is called by a program by the appearance of the function name in a statement. The rules to be remembered while calling a *function* are:

(i) The actual arguments in the calling *function* must agree in number, order and *type* with the formal arguments in the *function* declaration.

(ii) Copies of the values of actual arguments are sent to the formal arguments and the *function* is computed. Thus the actual arguments remain intact.

(iii) If the formal argument is an array name the corresponding actual argument in the calling *function* must also be an array name of the same *type*.

(iv) A *return* statement in the body of the *function* returns a value to the calling *function*.

The *type* of expression used in return statement must match with the type of the *function* declared. There may be functions without a *return* statement. In such a case the *function* must be declared as *void*.

(v) If type of *function* is omitted it is taken by default as int.

(vi) A function can return *only one* value.

The concept of calling a *function* is summarised in Fig. 13.1.

```
type function name ( type a, type b, type c )
/* Function prototype */
main()
{ /* Beginning of main function */
     Main function body
/* Calling a function */
variable name = function name( x, y, z );
                or
function name( x, y, z );
x, y, z actual parameters
} /* End of main function */

type function name ( type p, type q, type r )
{ /* Beginning of function */
     type s, t, u; /* Local variables */
     Body of function
     return ( expression );
/* Value of expression returned to function name */
} /* End of function */
```

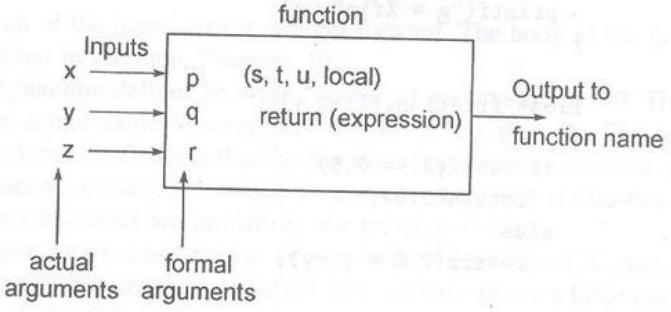

Fig. 13.1 Illustrating definition and calling of a function.

A function can be written separately and kept in a library for use by a class of users. With this facility a large program may be divided into a number of subprograms and linked together.

Functions are employed in applications where one needs them in computing expressions. As function name returns a value it may be used as parts of expressions. Functions may also perform operations without returning any value.

We will close this section with two examples in which functions are declared and used.

Example 13.4

Suppose the frictional force acting on a particle is given by the equations:

$$\begin{aligned} f(y) &= 0 \quad && \text{for } |y| \le 0.5 \\ &= 2y^2 && \text{for } |y| > 0.5 \end{aligned} \tag{13.3}$$

It is required to use this function in Eq. (13.4)

$$g = (mv/t) - f(v^2) + at \tag{13.4}$$

A small program to do this is written as Example Program 13.8. It is seen from this example that

(i) When the function is called the actual argument may be an expression. This is valid provided it is of the same type as the formal argument.

(ii) A float arithmetic expression is returned in the body of the subprogram. The value returned to the main program will thus be of type float.

(iii) Observe that the function friction calls another function float abs(x).

```
/* Example Program 13.8 */
/* Function definition and use */
#include <stdio.h>
float friction(float a);
float abs(float p);

main()
{
  float g, m, v, a, t;
  scanf("%f %f %f %f", &m, &v, &a, &t);
  g = (m * v/t) - friction(v*v) + a*t;
  printf("g = %f\n", g);
}

float friction(float y)
{
  if (abs(y) <= 0.5)
    return(0.0);
  else
    return(2.0 * y *y);
}

float abs(float x)
{
  if (x <0)
    x = -x;
  return(x);
} /* End of abs(x) */
```

Program 13.8 Function to simulate friction.

Example 13.5

A *function* may call another *function* which may in turn call a third *function*. The sequence of calls and returns are illustrated in Example Program 13.9. Observe that when a *function* f calls a *function* g from f, g will do its task and return to the statement calling g. The printed output of the program is shown below the program.

The student should examine the output along with the program to understand the flow of control from calling *function* to the called *function* and back from the called to the calling *function*.

13.4 ARRAYS IN FUNCTIONS

So far we have considered examples where the formal arguments of *functions* are scalars. There are many applications where an entire array has to be used as an argument. We will consider such applications in this section.

We will consider the problem of picking the highest and the second highest marks in a class which was discussed in Chapter 10. We revise Example Program 10.2 and rewrite it using functions.

Example 13.6

Example Program 10.2 is written using functions as Example Program 13.10. The *function* first_mark has as formal arguments marks[ ] and size. Observe that the *function* is of type *int* indicating that a single value of type *int* will be returned. The first argument of first_mark is *int* marks[ ]. The square brackets indicates that marks is an array. The size of the array is specified by the formal parameter *int* size. As the size may vary, the actual argument will send the size when the *function* is called.

The *function* second_mark has two formal arguments. The argument *int* marks [ ] is an array and the second formal argument is *int* dim which corresponds to the size of the array marks [ ].

The definition of the *functions* are straight forward. The body of the function is based on the program given in Example Program 10.2.

The main( ) *function* defines an array "score" of maximum size 50. The array_size is also declared. The actual value of array_size should be less than 50. The rest of the main function is self-explanatory. Observe that the *functions* first_mark and second_mark are called in the printf statement. Calling a function in a printf statement is allowed. Single values returned by the two functions are printed by the printf statement.

In this example we have an array as an argument and the function returns only a single value. If a function is to return multiple values then we have to use a different technique. We will illustrate this in the following example.

Example 13.7

In the last example we found the first and second marks in a class. If we want to arrange the marks in descending order we can follow a similar idea. The algorithm we will use is called a *bubble sort*. The way we will implement it is quite inefficient but it illustrates the use of arrays and the method of getting more than one value as output from a function.

```
/* Example Program 13.9 */
/* Illustrating calls and returns from function */
#include <stdio.h>
/* Definition of function prototypes */
void f(void);
void g(void);
void h(void);
main()
{
  printf("Call to f from main\n");
  f();  /* Call to f */
  printf("Control has returned to main\n");
} /* End of main */

void f(void)  /* function f defined */
{
  printf("Control now in f\n");
  printf("Call to g from f\n");
  g(); /* End of f */
  printf("Control has returned to f from g\n");
} /* end of f */

void g(void) /* function g defined */
{
  printf("Control now in g\n");
  printf("Call to h from g\n");
  h(); /* Call to h */
  printf("Control has returned to g from h\n");
} /* End of g */

void h(void) /* function h defined */
{
  printf("Control now in h\n");
  printf("Return control to g\n");
} /* End of h */
```

```
Output of Example Program 13.9

Call to f from main
Control now in f
Call to g from f
Control now in g
Call to h from g
Control now in h
Return control to g
Control has returned to g from h
Control has returned to f from g
Control has returned to main
```

Program 13.9 Calls and returns from functions.

```
/* Example Program 13.10 */
#include <stdio.h>
/* Function prototype definitions */
int first_marks(int marks[], int size);
int second_marks(int marks[], int size);
main()
{
  int array_size, score[50], i;
  scanf("%d", &array_size);
  for (i = 0; i <= array_size -1; ++i)
    scanf("%d", &score[i]);
  printf("First Marks = %d Second Marks = %d\n",
         first_mark(score, array_size),
         second_mark(score, array_size));
} /* End of main */

int first_mark(int marks[], int size)
{
  int first = 0, i;
  for (i = 0; i <= (size -1); ++i)
    if (marks[i] > first)
      first = marks[i];
  return(first);
} /* End of first mark */

int second_mark(int mks[], int dim)
{
  int second = 0, i;
  for (i = 0; i <= (dim -1); ++i)
    if (mks[i] != first_mark(mks, dim))
      if (mks[i] > second)
        second = mks[i];
  return(second);
} /* End of second mark */
```

Program 13.10 Array as argument in a function.

The method we follow is to use two *for* loops (See the *function* definition in Example Program 13.11). In the body of the *for* loop we compare marks [1] with marks [0] and interchange them if marks [1] > marks [0]. Now marks [0] will contain the higher marks. We next compare marks [2] with marks [0] and interchange them if marks [2] > marks [0]. This procedure is repeated successively comparing marks [0] with marks [3] etc., till marks [class_size]. At the end of this loop marks [0] will contain the highest marks.

Having placed the highest marks in marks [0], the procedure is repeated starting with marks [1] and comparing it with marks [2], marks [3], ..., marks[class_size]. Second highest marks will now be in marks [1]. Continuing along the same lines the two loops in the *function* arrange the marks in descending order. The sorted marks are in marks [0], marks [1], ..., marks [class_size] within the *function* sort_marks.

```
/* Example Program 13.11 */
/* Sorting Marks */
#include<stdio.h>
void sort_marks(int marks[], int no_of_students);

main()
{
  int i, no_students, score[50];
  scanf("%d", &no_students);
  for (i = 0; i <= no_students -1; ++i)
    scanf("%d", &score[i]);
  sort_marks(score, no_students);
  printf("Sorted marks ", score[i]);
  for (i = 0; i <= no_students - 1; ++i)
    printf(" %d ", score[i]);
  printf("\n");
} /* End of Main */

void sort_marks(int marks[], int class_size)
{
  int i, j, temp;

  for (i =0; i <= class_size-2; ++i)
    for(j = i+1; j <= class_size -1; ++j)
      if (marks[j] > marks[i])
        {
          temp = marks[i];
          marks[i] = marks[j];
          marks[j] = temp;
        }
} /* End of sort marks */
```

Program 13.11 Sorting marks.

The question is how to *return* these sorted marks to the calling main *function*. The *return* statement of a *function* can only return one value. The way C programming language solves the problem is to treat arrays, appearing in formal argument list in a different manner. We saw that normally when a *function* calls another *function* the actual argument value is *copied* into the formal argument name and the called *function* uses it in processing. The value of the actual argument in the calling function is unaltered. When an argument is an array, however, a copy of the array elements *is not made* and sent to the formal array. Instead, the address, in memory where the array is stored is made known to the called *function*. In other words the actual array in the calling *function* and the formal array in the called *function occupy the same locations in memory*. Thus the values stored in the formal array are accessible to the calling *functions*. Even though the name of the actual array used as an argument of the calling *function* and the name of the formal array used as an argument in the called function may be different they are allocated the same locations in memory and thus their contents are identical.

An array processed by a *function* is thus available to the calling *function* which can print

it out or do further processing as necessary. In Example Program 13.11 the main function declares score as an array which can store 50 elements. It would have been nice to declare it as score [no_students]. This is however not allowed in C. We must use only a constant in array size declaration. If the program is to be used in many instances with varying values of no_students the array size should be set to the maximum expected value of no_students. After the declaration the main *function* reads the value of no_students. This is used in the next *for* loop to read marks from the input and store them in score [i]. After storing the marks, main calls the *function* sort_marks with formal arguments score and no_students. Observe that while calling score appears without square brackets. The fact that score is an array is known from the declaration in main. When main calls sort_marks the address of the locations in memory where the actual array argument score is stored is passed to the formal array argument marks. Thus score and marks now become synonyms occupying the same locations in memory. The *function* sort_marks now sorts score (which is the same as marks). The sorted values are thus available to main in the array score. The last statement in main (which is a *for* loop) prints the value of the sorted marks stored in score. As no value is returned, the function sort_marks is declared void in the main program.

The method of passing a scalar argument in which a copy of the value of the actual argument of the calling *function* is made and given to the formal argument of the called *function* is known as *call by value*.

The method of passing an array argument in which the *address* of the actual array argument of the calling *function* is given to the formal array argument of the called *function* is known as *call by reference*. (It is better to say "call function giving address of argument".)

In call by *value what* is stored as an argument in memory is sent. In call by *reference* the *address where an argument is stored* is sent. We will now write another program to illustrate the use of arrays in functions.

Example 13.8

Given an array a[0], a[1], ... , a[length – 1] it is required to reverse it, interchange a[0] and a[length – 1], a[1] and a[length – 2] and so on. At the end of the operation the original array is destroyed and in its place an array whose elements are interchanged remains.

To solve this problem we have written in Example Program 13.12 a *function* reverse_array whose formal arguments are *int* a[ ] and *int* length. As the subscript begins with zero the maximum subscript is (length – 1). If the length of the array is even then a pair of elements are interchanged. If the length is odd the element at the centre remains undisturbed. This idea is implemented in the *for* loop. (length/2) – 1 is used as final index as subscripting starts with zero.

In this problem also the array is transmitted to main by reference and is thus available to main. The function reverse_array is declared void.

We have written Example Program 13.13 using a *function* interchange to interchange two values. This program does not work. This is due to the fact that the individual elements of arrays are *copied* and sent to x and y respectively. The values of x and y are swapped by the function interchange. The swapped values are not returned to the calling function as the calling function does not know the addresses of x and y. Only *entire arrays* are transmitted *by reference*, i.e., by sending array address to the called function's formal array argument. Array elements are transmitted only *as values*.

```
/* Example Program 13.12 */
/* Illustrating use of array argument */
#include <stdio.h>
void reverse_array(int p[], int length);

main()
{
  int array_length, i, b[10];
  scanf("%d", &array_length);
  printf("Input array\n");
  for(i = 0; i <= array_length -1; ++i)
    scanf("%d", &b[i]);
  reverse_array(b, array_length -1);
  printf("Reversed array\n");
  for(i = 0; i <= array_length -1; ++i)
    printf("%d ", b[i]);
} /* End of Main */

void reverse_array(int a[], int length)
  {
    int i, temp = 0, len;
    len = (length/2) - 1;
    for(i = 0; i <= len; ++i)
      {
        temp = a[i];
        a[i] = a[length -i];
        a[length -i] = temp;
      }
  } /* End of function reverse array */
```

Program 13.12 Reversing an array.

```
/* Example Program 13.13 */
/* Illustrates mistake in calling a function */

void reverse_array(int a[], int length)
{
  int i;
  for (i = 0; i <= (length/2 -1); ++i)
    interchange(a[i], a[length - i]);  /* Error */
} /* End of function reverse array */

void interchange(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}  /* End of function interchange */
```

Program 13.13 Illustrating error in calling a function.

Example 13.9

Two dimensional arrays or matrices are also treated just like arrays. To illustrate this we give Example Program 13.14 which sorts the rows of a matrix. This program uses a function sort which is similar to the sort_marks function given in Example Program 13.11. Observe that the *function* row_sort has as one formal argument a two dimensional array, the others are scalar arguments. Observe that the formal argument, namely matrix a, *must* have its dimensions

```
/* Example Program 13.14 */
/* This function uses the sort-marks function given
   in Example Program 13.11 */
#include <stdio.h>
void sort(int row[], int no_cols);
void row_sort(int a[][], int r_length, int c_length);
main()
{
  int matrix[3][4], row_size, col_size, i, j;
  scanf("%d %d", &row_size, &col_size);
  for (i = 0; i <= row_size-1; ++i)
    for (j = 0; j <= col_size-1; ++j)
      scanf("%d", &matrix[i][j]);
  row_sort(matrix, row_size-1, col_size-1);
  for (i = 0; i <= row_size-1; ++i)
    {
      for (j = 0; j <= col_size-1; ++j)
       printf("%d ", matrix[i][j]);
      printf("\n");
    }
}
void row_sort(int a[3][4], int row_dim, int col_dim)
{
  int r;
  for(r = 0; r <= row_dim; ++r)
    sort(&a[r][0], col_dim);
} /* End of row sort */
void sort(int b[], int size)
{
  int i, j, temp;
  for (i = 0; i <= size; ++i)
    for(j = 0; j <= size; ++j)
      if (b[j] < b[i])
        {
          temp = b[i];
          b[i] = b[j];
          b[j] = temp;
        }
} /* End of sort */
```

Program 13.14 Sorting rows of a matrix.

specified. This is necessary for correct indexing. When row_sort calls sort observe that it specifies the starting address of the array for row 0, row 1 etc. The & operator is used to give the address. The address of the (0, 0) element of the matrix *a* is transmitted to sort which takes number of elements in one row of matrix a[ ] [ ] each time it is called and sorts this row. The functions sort is called once for each row of a[ ][ ]. The rest of the program is self_explanatory. (The address of an array element a[i][j] is given by:

Address of a[0][0] + (col.dimension * i + j))

Example 13.10

We now consider another example of using a two dimensional array. We develop a function to transpose a square matrix, i.e., a matrix with number of rows = number of columns. The matrix A is

$$A = \begin{bmatrix} 1 & 4 & 5 \\ 3 & 2 & -5 \\ 4 & 6 & 8 \end{bmatrix}$$

The transposed matrix is:

$$A^T = \begin{bmatrix} 1 & 3 & 4 \\ 4 & 2 & 6 \\ 5 & -5 & 8 \end{bmatrix}$$

The $(i, j)^{th}$ element of the transposed matrix A^T is the $(j, i)^{th}$ element of the given matrix A.

The function:

void transpose (int a[5][5], int dim)

given in Example Program 13.15 transposes the given square matrix. Observe the indexing of the two *for* loops which is important in this program.

We conclude this section with an example which uses three functions.

Example 13.11

A program which determines the number of data points in each one of a set of assigned intervals will now be developed. Such a grouping of data into intervals is known as a frequency table and is used extensively in statistics.

The strategy we will adopt to develop the program is as follows:

(1) Given the data array to be grouped in a frequency table find the maximum and minimum values of data in the array.

(2) Given the maximum and minimum values of data and the interval size the number of intervals in which data is to be grouped is found by dividing the maximum data minus minimum data by the size of the interval and rounding it to the next higher integer value.

```
/* Example Program 13.15 */
/* Maximum size of matrix is 5 x 5 in this program */
#include <stdio.h>
void transpose(int matrix[][], int dim);
main()
{
  int i, j, size, matrix[5][5];
  scanf("%d", &size);
  for (i = 1; i <= size; ++i)
    {
      for (j = 1; j <= size; ++j)
        scanf("%d", &matrix[i][j]);
    }
  transpose(matrix, size);
  for (i = 1; i <= size; ++i)
    {
      {
        for (j = 1; j <= size; ++j)
          printf("%d ", matrix[i][j]);
      }
      printf("\n");
    }
}

void transpose(int a[5][5], int dim)
{
  int i, j, temp;
  for (i = 1; i <= dim; ++i)
    {
      for(j = i+1; j <= dim; ++j)
        {
          temp = a[i][j];
          a[i][j] = a[j][i];
          a[j][i] = temp;
        }
    }
}
```

Program 13.15 Transposing a matrix.

(3) Having found the number of intervals, a data item is taken and the interval in which it is to be placed is determined by a *while* loop. This loop also counts the number of data items in each interval.

The program is structured with three functions. One function is used to find the maximum data value in the data array, the second one is used to find the minimum data. The third function counts the number of data items in each interval. The function to find the maximum data value has the data array and its size as the inputs and returns the

maximum value as the output. Similarly the function to find the minimum accepts the data array and its size and returns the minimum data value. The function for obtaining the frequency table accepts the data array, its size, interval width, minimum and maximum data of data array and the number of intervals as inputs. It gives as output, the frequency table giving the number of data points in each interval. The program is given as Example Program 13.16.

```
/* Example Program 13.16 */
#include <stdio.h>
/* Function prototype */
int max_data(int data[], int no_data);
int min_data(int data[], int no_data);
void freq_table(int data[], int no_data, int width,
                int low, int high, int slots,
                int table[]);

main()
{
  int no_of_data, slot_width, in_data[100], max,
  min, i,  no_of_intervals, out_table[20];
  scanf("%d %d", &no_of_data, &slot_width);
  for (i = 0; i <= no_of_data-1; ++i)
    scanf("%d", &in_data[i]);
  max = max_data(in_data, no_of_data-1);
  min = min_data(in_data, no_of_data-1);
  printf("Min data = %d, Max data = %d\n", min, max);
  printf("Interval width = %d\n", slot_width);
  no_of_intervals = (max - min) / slot_width + 1;
  printf("No of intervals = %d\n", no_of_intervals);
  freq_table(in_data, no_of_data-1, slot_width,
             min, max, no_of_intervals, out_table);
  printf("Frequency Table \n");
  for (i = 0; i < no_of_intervals; ++i)
    printf("Slot %d = number of data = %d\n", i,
           out_table[i]);
} /* End of main program */

int max_data(int data_in[], int no_data)
{
  int i, temp;
  temp = data_in[0];
  for (i = 1; i <= no_data; ++i)
    if (data_in[i] > temp)
      temp = data_in[i];
  return(temp);
} /* End of max data */
```

```
int min_data(int data_in[], int no_data)
{
  int i, temp;
  temp = data_in[0];
  for (i = 1; i <= no_data; ++i)
    if (data_in[i] < temp)
      temp = data_in[i];
  return(temp);
} /* End of function min data */

void freq_table(int data_in[], int no_data,
                int interval_width, int min,
                int max, int no_intervals,
                int tab_freq[])
{
  int i, slot_found, k, limit;
    for(i = 0; i <= no_data; ++i)
    {
      slot_found = 0;
      k = 0;
      limit = min + interval_width;
      while (slot_found == 0)
        {
          if (data_in[i] < limit)
            {
              ++tab_freq[k];
              slot_found = 1;
            }
          else
            {
              limit += interval_width;
              ++k;
            }
        } /* End of while */
    } /*  End of for loop */
} /* End of function freq_table */
```

Program 13.16 Frequency table.

13.5 GLOBAL, LOCAL AND STATIC VARIABLES

We saw that *functions* return only one value in C. If a number of values are to be returned by a function we have three ways of doing it. One method is to use an array as an argument. The problem with this is that all the array elements should be homogeneous. A second method is to place the values to be returned in variables accessible to *all* functions and the main program. Such variables are called *global variables* in C. In this section we will describe how to declare and use global variables. A third method is to declare formal arguments in *functions* as addresses instead of as variable names. In other words formal arguments are specified as the addresses of variable names.

We will take as an example the solution of a quadratic equation for which a program was developed in Chapter 8 (Example Program 8.4). If we want to write a *function* to solve the quadratic equation then we should get the results, namely, x_real_1, x_real_2, x_imag_1, x_imag_2 returned. As a *function* cannot return more than one value, to obtain these values we must find some other way. We solve it by declaring x_real_1, x_real_2, x_imag_1 and x_imag_2 as global variables. The program using a function solve_quadratic is given as Example Program 13.17. Observe that global values are used inside the function body.

```
/* Example Program 13.17 */
/* Function solve_quadratic solves quadratic
   equations  */
#include <stdio.h>
#include <math.h>
/* Function Prototype */
void solve_quadratic(float p, float q, float r);
/* global variables defined below */
float x_real_1, x_real_2, x_imag_1, x_imag_2;

main()
{
 float x, y, z;
 while(scanf("%f %f %f", &x, &y, &z) != EOF)
  {
    x_real_1 = x_real_2 = x_imag_1 = x_imag_2 = 0;
    printf("Coefficients of the quadratic equation");
    printf(" (a*x*x + b*x +c) are:\n");
    printf("a = %f, b = %f, c = %f\n", x, y, z);
    printf("Solution of Quadratic equation\n");
    solve_quadratic(x, y, z);
    printf("x_real_1 = %f, x_real_2 = %f\n",
          x_real_1, x_real_2);
    printf("x_imag_1 = %f, x_imag_2 = %f\n\n",
          x_imag_1, x_imag_2);
  }
} /* End of main */

void solve_quadratic(float a, float b, float c)
{
  float discrmnt;
  float temp;
  if (a == 0)
    {
      printf("Linear equation, Only one root\n");
      x_real_1 =  -c/b;
      return;
    }
```

```
    discrmnt = b*b - 4.0*a*c;
    if (discrmnt > 0)
      {
        printf("Real roots\n");
        temp = sqrt(discrmnt);
        x_real_1 = (-b + temp) / (2.0 * a);
        x_real_2 = (-b - temp) / (2.0 * a);
        return;
      }

    if (discrmnt < 0)
      {
        printf("Complex Conjugate roots\n");
        discrmnt = -discrmnt;
        x_imag_1 = sqrt(discrmnt)/(2.0 * a);
        x_imag_2 = -x_imag_1;
        x_real_1 = -b / (2.0 *a);
        x_real_2 = x_real_1;
        return;
      }

    if (discrmnt == 0)
      {
        printf("Repeated roots\n");
        x_real_1 = -b / (2.0 * a);
        x_real_2 = x_real_1;
        return;
      }
  } /* End of function solve_quadratic */
```

Program 13.17 Solving quadratic equation.

The structure of C programs we have written so far are as shown in Table 13.1.

Variables declared outside the body of all functions (including main) are known as global variables or external variables. Normally in practice they are defined after function prototype definitions and main ( ) as shown in Table 13.2.

The rules for declaration of global variables are the same as for local variables in functions. Global variables may be accessed by their declared name by any *function* and changed. The latest value will be stored in them. Global variables are initialised automatically to zero by the compiler.

A global variable name should not be declared again in a function. If it is done the value stored in the global variable is not accessible in the function body.

The main disadvantage of global variables are:

1. When functions are developed by different programmers they should know the names of global variables and use the same names. Thus strict coordination between members of the programming team is essential.

Table 13.1 Structure of a C Program

File inclusion statements	# include <stdio.h> # include <math.h> # include .....
Function prototypes	Definition of function prototypes Declaration of global variables. main ( )
Body of main	{ Declaration of local variables. Statements in the body main ( ) including calls to *functions*. } /* End of main */
Body of fun_1	Void fun_1(int p[ ], float q, int r) { Declaration of local variables. Statements in the body of fun_1( ) including calls to other functions. } /* End of fun_1 */
Body of fun_2	int fun_2 (float a, int z[ ]) { Declaration of local variables. Statements in body of fun_2 including calls to other functions. } /* End of fun_2 */

Table 13.2 Illustrating Declaration of Global Variables

```
# include statements
  Function prototypes
  Declaration of Global Variables
  main( )
  {
  ...
  } /* End of main( ) */
  Definition of functions
```

2. If the declaration of the global variable is changed, then all functions using thes variables must be changed.

3. As many functions can alter the contents of global variables there is always a dange of unintentional change of stored data leading to errors. Such changes are known as "side_effects and are difficult to detect.

4. Normally a programmer must make it a practice to clearly specify the inputs an

outputs of functions and use them as formal parameters. All other variables required in the function must be declared as local. Global variables must not be used unless they are essential to solve the problem and there is no simple way of avoiding them.

Referring to Table 13.1 we see that within the body of *functions* we define local variables. As we explained earlier memory locations are allocated to the local variables when a *function* is called. The variables are not automatically initialized. They should be initialized by the programmer. They are available only within the *function*. When control leaves the function the memory locations reserved for local variables are released. Their contents are thus lost and not available to any other *function*. Thus local variables have no meaning outside the *function*. If a *function* is called again new memory locations are allocated again to the local variables. The old values are lost. The variables should thus be initialized again. Such local variables are also known as *automatic variables* as memory is allocated to them when the *function* is entered. One may declare such variables as:

auto type <variable name>

Normally the key word auto is omitted as by default all local variables declared in a *function* are *auto*. If the value of a local variable in a *function* is to be preserved and not lost when a *function* is exited then such a variable is declared *static*.

Thus if we write:

static int k ;

then the value of *k* will not be lost when control leaves a *function*. Example Program 13.18 illustrated the use of *static* variables.

In this Example Program sum is declared *static* in the function. First time this function is called sum is initialized and used by the function sum_good_age. When the function is exited sum is saved as it is declared *static*. Each time the function is entered the earlier value stored is used. The initialization given in static is *valid only the first time* the function is entered. Thus when all data have been read by the *while* loop in main ( ) sum_good_age has the final value of sum returned.

This example is cooked up to illustrate the use of static declaration.

There are some more questions on functions we have not answered in this Chapter. These are:

(i) Can *functions* be defined within other *functions* in C language? The answer is NO. Pascal allows this and there is so called block_structure in Pascal which is not available in C.

(ii) Can a *function* call itself? The answer is Yes. This is called *recursion*. We will discuss this in a later chapter.

(iii) Can a *function* be made a formal argument of another *function*? The answer is NO.

(iv) Can a C program invoke programs written in another language such as Fortran? If yes, how? The answer is Yes.

For writing a number of useful C programs it is not necessary to know the answers to these questions. Only after using *functions* in many applications would a programmer reach a level of maturity needing such facilities. We thus postpone answering some of these questions to later chapters in this book.

```
/* Example Program 13.18 */
/* A program to find average age of a group */
#include <stdio.h>
int sum_good_age(int b);
int data_ok(int p);
main()
{
  int s, serial_no, age, no_of_data, good_data, avge_age;
  while(scanf("%d %d", &serial_no, &age) != EOF)
    {
      ++no_of_data;
      s = sum_good_age(age);
      good_data += data_ok(age);
    }
  if (((float)good_data / (float)no_of_data) < 0.80)
    printf("Too many bad data - Result not given\n");
  printf("Good data = %d, No of data = %d\n",
        good_data, no_of_data);
  else
    {
      avge_age = s / good_data;
      printf("Good data = %d, Average Age = %d\n",
            good_data, avge_age);
    }
}
int data_ok(int p)
{
  if ((p < 0) || (p > 100))
    return(0);
  else
    return(1);
}
int sum_good_age(int x)
{
  static int sum = 0;
  if (data_ok(x))
    sum += x;
  return(sum);
} /* End of sum_good_age */
```

Program 13.18 Average age of a group.

EXERCISES

13.1 Write a C program for the function f(x) defined below:

$f(x) = 2x^2 + 3x + 4$ for $x < 2$

$f(x) = 0$ for $x = 2$

$f(x) = -2x^2 + 3x - 4$ for $x > 2$

13.2 Write a C function to evaluate the series

$$f(x) = 1 + \frac{x^2}{2!} - \frac{x^4}{4!} + \frac{x^6}{6!} - \frac{x^8}{8!} + \frac{x^{10}}{10!} - \frac{x^{12}}{12!}$$

13.3 Write C function to evaluate the series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} \cdots$$

to five significant digits.

13.4 Write a function to multiply a square-matrix by a vector. Assume a $n \times n$ matrix and a n component vector. The call would give the size of the matrix and the names of the matrix and a vector.

13.5 Use the function of Exercise 13.4 to generate a 6×6 matrix whose elements are given by

$$a_{ij} = 2^{-(i-j)}$$

13.6 Write a function to find the trace of a matrix. The trace is defined as the sum of the diagonal elements of the matrix. How would this routine be called?

13.7 Write a function to multiply two $n \times n$ matrices.

13.8 Write a function to find the norm of a matrix. The norm is defined as the square root of the sum of the squares of all elements in the matrix.

13.9 Write a function to sort a set on n numbers in ascending order of magnitude. How would this routine be called?

13.10 Write functions to add, subtract, multiply and divide two complex numbers $(x + iy)$ and $(a + ib)$.

13.11 Rewrite Example Program 12.1 using separate functions to
(i) Read a SMAC program into memory,
(ii) Check op code and address,
(iii) Execute the machine language program.

13.12 Rewrite Example Program 12.2 using functions for
(i) Initialization,
(ii) Read a student's marks and validate it,
(iii) Find the average marks and class of a student,
(iv) Determine class average, covariance etc.

13.13 Rewrite Example Program 12.4 using appropriate functions.

13.14 Write a function to validate the elements of a matrix of size $n \times n$. The validation rules are:
(i) All diagonal entries should be positive,
(ii) The matrix should be symmetric,
(iii) All the non-diagonal elements should be negative or zero.