```
/* Example Program 19.3 */
#include <stdio.h>
typedef struct tree
{
  int value;
  struct tree *right;
  struct tree *left;
} Tree;
Tree *root_node;
int inp_value;
void build_tree(Tree *t, int n);
void trav_in_order(Tree *node);
main()
{
  scanf("%d", &inp_value);
  /* Create root of tree and store value read */
  root_node = (Tree *)malloc(sizeof(Tree));
  root_node->value = inp_value;
  root_node->right = NULL;
  root_node->left = NULL;
  while (scanf("%d", &inp_value) != EOF)
    build_tree(root_node, inp_value);
  printf("Inorder sequence :");
  trav_in_order(root_node);
  printf("\n");
} /* End of main */
void trav_in_order(Tree *node)
{
  if (node != NULL)
    {
      /* Visit left node */
      trav_in_order(node->left);
      printf("%d ", node->value);
      /* Visit right node */
      trav_in_order(node->right);
    }
} /* End of trav_in_order */

/* Place function build_tree here */
```

Program 19.3 Sorting integers-tree traversal by recursion.

Besides the inorder traversal in which the root is visited in the middle there are two other methods of systematically visiting the nodes of a tree. They are the *preorder* and *postorder* traversals. In preorder we visit the root first and then the left and right subtrees. In postorder we visit the left and right subtrees and the root last. The postorder traversal is useful in converting an *infix* algebraic expression (A + B) * (C – D) to a *postfix* expression AB + CD – *. This is illustrated by Algorithm 19.7. The tree corresponding to an arithmetic expression is shown as Fig. 19.4. The postorder traversal algorithm would print out the answer as:

AB + CD – *

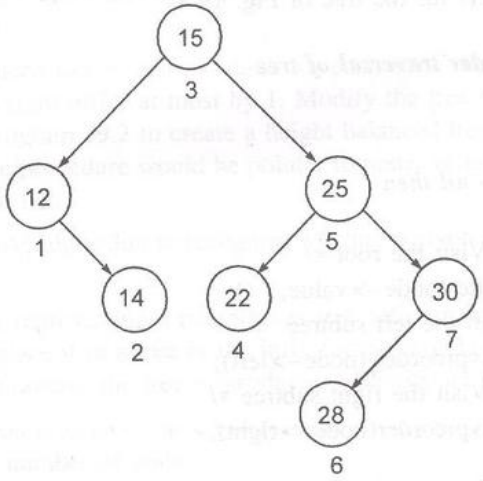

Fig. 19.3 Showing inorder traversal of a tree.

Algorithm 19.7: Postorder traversal of tree

```
Travpostorder
    {If node <> nil then
        {  visit the left subtree
           Travpostorder (node->left);
           /* visit the right subtree */
           Travpostorder (node->right);
           /* Write contents of root */
           Write (node->value, ' ' );
        }
    }
```

The order in which the nodes are visited by the algorithm is shown by placing numbers below each node in Fig. 19.4.

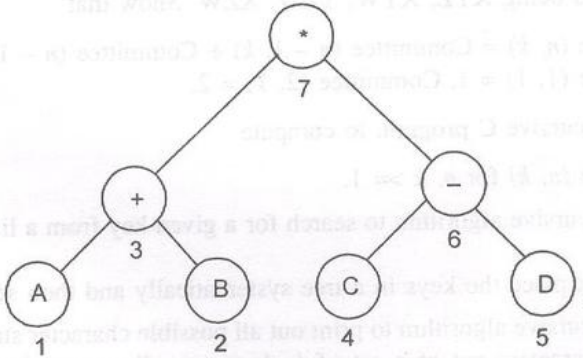

Fig. 19.4 Tree representing an arithmetic expression.

The algorithm for preorder traversal is given as Algorithm 19.8. In this case the algorithm would print * + AB – CD for the tree of Fig. 19.4.

Algorithm 19.8: Preorder traversal of tree

```
Travpreorder
  {
    if node <> nil then
      {
        /* Visit the root */
        Write (node->value, ' ');
        Visit the left subtree
        Travpreorder (node->left);
        /* Visit the right subtree */
        Travpreorder(node->right);
      }
  }
```

EXERCISES

19.1 The Ackerman's function is defined as follows:

$A(0, n) = n + 1$
$A(m, 0) = A(m - 1, 1) \quad (m > 0)$
$A(m, n) = A(m - 1, A(m, n - 1)) \quad (m, n > 0)$

(i) Using the above definitions show that $A(2, 2) = 7$
(ii) Write a C function to compute A(m, n).

19.2 Write a recursive function to compute a^n where a is real and n is an integer. Compare it with an iterative function.

19.3 Let Committee *(n, k)* be the number of committees of k persons which can be formed from among n persons. For example Committee (4, 3) = 4. Given four persons X, Y, Z, W there are 4 committees which can be formed with three persons, the committees being XYZ, XYW, YZW, XZW. Show that

Committee (n, k) = Committee $(n - 1, k)$ + Committee $(n - 1, k - 1)$; (n, k > 0)
Committee (1, 1) = 1, Committee (2, 1) = 2.

Write a recursive C program to compute

Committee *(n, k)* for $n, k >= 1$.

19.4 Write a recursive algorithm to search for a given key from a list of integer valued keys.
[*Hint*: First place the keys in a tree systematically and then search the tree.]

19.5 Write a recursive algorithm to print out all possible character strings of length n by picking characters out of a set of k characters. For example if the given set of characters is 'A', 'B', 'C' then 9 possible strings exist of length n = 2.

19.6 Write a recursive function that accepts a prefix expression consisting of the arithmetic operators +, –, * and / and single digit integer operands and returns the value of the expression.

19.7 A height balanced tree is one in which for each node the number of nodes on its left and on its right differ at most by 1. Modify the tree building procedure given as Example Program 19.2 to create a height balanced tree. The formal parameters to the buildtree procedure would be pointer to node, value at node and the number of items in the list.

19.8 Write a recursive algorithm to recognise whether a given string is a valid identifier in C.

19.9 Given a string representing a parentheses free infix arithmetic expression, write a procedure to place it in a tree in the infix form. Assume that a variable name is a single letter. Traverse the tree to produce an equivalent postfix expression string.

19.10 Write a procedure to count in a binary tree:
(i) The total number of nodes
(ii) The number of leaf nodes.

20. Bit Level Operations and Applications

Learning Objectives

In this chapter we will learn:

1. Operations available in C to manipulate the individual bits of numbers stored in variable names
2. Use of bit level operators to carry out some useful functions in C

One of the special features of C is the availability of operations on bits and strings of bits. This facility is not available in most high level languages. These operations are primarily useful in simulating digital systems at a low level. Packing bits in a word is possible and useful to minimise storage requirements in some problems. In this chapter we will first define various operations on bits and how they can be applied.

20.1 BIT OPERATORS

In Table 20.1 the symbol used for operations on bits, their type and precedence are given.

Table 20.1 Bit Operators

Operator symbol	Operation	Type of operation	Precedence or rank of operator
&	Bitwise AND	Binary	3
^	Bitwise Exclusive OR OR	Binary	4
|	Bitwise OR	Binary	Lowest 5
~	One's complement	Unary	Highest 1
<<	left shift	Binary	2
>>	right shift	Binary	2

Binary operations can be used with variables of type signed or unsigned integer or character but not with floating point. The definition of the operators is given in Table 20.2.

Observe that for & (AND) operator both operands must be 1 for the result to be 1. For | (OR) operator either operand should be 1 for the result to be 1. For ^(EOR) operator either operand but not both must be 1 for the result to be 1. The ~ (complement) operator is a unary operator. It changes the operand to 1 if it is 0 and to 0 if the operand is 1. The left and right shift operators are explained in Table 20.3.

Table 20.2 Definition of Bit Operations

a	*b*	*a&b*	*a|b*	*a ^ b*	*~a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 20.3 Definition of Left and Right Shift Operators

String p	p >> 3	Action
01011010	00001011	p shifted right by 3 positions. Left most bits filled with 0s. Right most 3 bits lost.

String q	q << 4	Action
11000110	01100000	q shifted left by 4 positions. Right most bits filled with 0s. Left most 3 bits lost.

20.2 SOME APPLICATIONS OF BIT OPERATIONS

Example 20.1

As the first example we will write Example Program 20.1 to find out whether a given integer is odd or even. If the least significant bit of a number is 0 it is even, otherwise it is odd. To find the least significant bit we AND bitwise the given number with 1. As 1 is represented in binary as 0001, AND will zero all bits of the given number except the last

```
/* Example Program 20.1 */
#include <stdio.h>

main()
{
  int given_number, mask = 1;
  while(scanf("%d", &given_number) != EOF)
    {
      if (given_number & mask)
        printf("Given number %d is odd\n",
                given_number);
      else
        printf("Given number %d is even\n",
                given_number);
    }
} /* End of main */
```

Program 20.1 Masking a binary string.

bit. The last bit will equal the least significant bit of the given number. For example if the given_number = 0011011 then given number & mask

$$001101 \ \& \ 000001 = 1$$

Example 20.2

The next example is to write a program to divide an unsigned number x by 2^n. Each right shift of a bit string is equivalent to dividing it by 2. Thus to divide by 2^n we shift right n positions the operand. When a string is shifted right its left most bits get filled with 0s. The sign bit should not be normally shifted. It, however, depends on the machine. We have thus decided to use an unsigned integer. Example Program 20.2 illustrates this. Remember that y >> = n is equivalent to y = y >> n.

```
/* Example Program 20.2 */
#include <stdio.h>

main()
{
  unsigned int x, two_power_n, n = 0, y;
  while(scanf("%u %u", &x, &two_power_n) != EOF)
    {
      /* The following while loop finds for a power
         of two integer such as 8 the integer power 3 */
      while (two_power_n != 1)
        {
          two_power_n >>= 1;
          ++n;
        }
      y = x;          /* x is stored in y */
      y >>= n;        /* y shifted right by n places */
      printf("x = %u divided by 2  power %u is %u\n",
              x, n, y);
      n = 0;
    } /* End of outer while */
} /* End of main */
```

Program 20.2 Dividing an integer by an integer which is a power of 2.

Example 20.3

Characters are normally stored in 1 byte of a computer. Only 7 bits are needed to code characters. The eighth bit can be used for a parity. We will now write a program to generate an even parity bit. If the total number of 1's in an eight bit string (including the parity bit) is even then this string satisfies even parity. If the number of 1s in a string is even then the exclusive or of the bits taken pairswise should be zero. In other words if the string is:

$$a_7 \ a_6 \ a_5 \ a_4 \ a_3 \ a_2 \ a_1 \ a_0$$

$$\text{then } a_0 \wedge a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6 \wedge a_7 = 0$$

This is used to generate the parity bit. Given a string

$$a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$$

even parity bit $\quad p_7 = a_0 \wedge a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6$

Example Program 20.3 generates this bit. In this program the expression (x&1) gets the least significant bit of x. The statement parity_bit ^ = (x&1) exclusive ors the existing

```
/* Example Program 20.3 */
#include <stdio.h>

main()
{
  unsigned char ch, x;
  unsigned int parity_bit = 0;
  ch = getchar();
  while (ch != '\n')
    {
      x = ch;
      parity_bit ^= (x & 1);
      /* least significant bit of x is used */
      x = x >> 1;
      while (x != 0)
        /* when x == 0 parity bit does not change */
        {
          /* parity_bit using next bit of x */
          parity_bit ^= (x & 1);
          x = x >> 1;
        }
      printf("Even parity bit of %c = is %u\n",
             ch, parity_bit);
      parity_bit = 0;
      ch = getchar();
    } /* End while */
} /* End of main */
```

Program 20.3 Generating a parity bit for a character.

parity_bit (initialized to 0) with the least significant bit of x. The character read, which is stored in x, is now shifted one bit position to the right. If after shifting one bit x is 0 no further work need to be done as all bits of x would be zero. If x is not equal to 0 then the least significant bit of x shifted by 1 position is exclusive ored with the parity bit. These statements are repeated in the *while* loop until x becomes 0. The parity bit is then printed.

Example 20.4

As the next example we simulate what is known as a full adder. When a string of bits x is added to another string y then the sum is obtained by first adding the least significant bits of

x and y. Carry if any is taken and added to the next significant bits of x and y. This is shown below.

carry	1	1	1	1	1	1	0	
x		0	1	1	1	0	1	0
y		1	0	1	0	1	1	0
Sum		0	0	1	0	0	0	0

Table 20.4 is a table showing the result of adding three bits.

Table 20.4 Table to show 3-Bit Addition

x_bit	y_bit	carry_bit	sum_bit	Carry to next bit position
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Using Table 20.4 we can write:

```
sum_bit = carry_bit ^ (x_bit ^ y_bit)

Carry to next bit position = (x_bit & y_bit) |
                             (x_bit & carry_bit) |
                             (y_bit & carry_bit)
```

The sum for the strings x and y is obtained by first adding the least significant bits of x and y with carry bit if any. This is stored in Sum string. Carry is also stored separately. Next x and y are shifted one position to the right. The least significant bits of shifted x and shifted y are added and the carry is added. The next bit of sum is thus obtained. The carry bit is also generated. This procedure is repeated until all bits in x and y are shifted. The program to implement this procedure is given as Example Program 20.4.

The program closely follows what we have explained x >> = 1 is equivalent to writing x = x >> 1. Observe that the sum obtained is being shifted left by one bit as we calculate the least significant bit of sum and each new bit should occupy the least significant bit position.

20.3 BIT FIELDS

Bit fields can be packed into an unsigned integer. This is primarily used to save space. We take an example to illustrate this. In Chapter 12 we illustrated a questionnaire format. It is reproduced as Table 20.5.

```
/* Example Program 20.4 */
#include <stdio.h>

main()
{
  unsigned int x, y, sum, carry_bit, x_bit, y_bit,
  sum_bit, i;
  while (scanf("%x %x", &x, &y) != EOF)
    {
      sum_bit = 0;
      carry_bit = 0;
      sum = 0;
      printf("x = %x, y = %x\n", x, y);
      printf("Bits printed from LSB to MSB\n");
      for(i = 1; i <= 4; ++i)
        {
          x_bit = (x & 1);
          y_bit = (y & 1);
          sum_bit = carry_bit ^ (x_bit ^ y_bit);
          carry_bit = (x_bit & y_bit) |
            (x_bit & carry_bit) |
              (y_bit & carry_bit);
          /* Bit placed in least significant
             position of sum */
          sum |= sum_bit;
          x >>= 1;
          y >>= 1;
          sum << 1;
          printf("sum of x and y = %x, carry = %x\n",
                 sum_bit, carry_bit);
        } /* End of for loop */
    } /* End of while */
} /* End main */
```

Program 20.4 Simulating a full adder.

Assume that there are a large number of questionnaires and the information filled up in them is to be stored for later retrieval. Instead of using one variable name (which will occupy one word in memory) for the answer to each question we can pack the information as shown in Fig. 20.1.

```
typedef struct packed_quest
        {   unsigned int sex: 1 ;
            unsigned int age: 6 ;
            unsigned int institution : 3 ;
            unsigned int interest : 3 ;
            unsigned int education : 3 ;
        }   Quest ;
```

Questionnaire information may now be defined as Quest participant_data[1000];

Table 20.5 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	

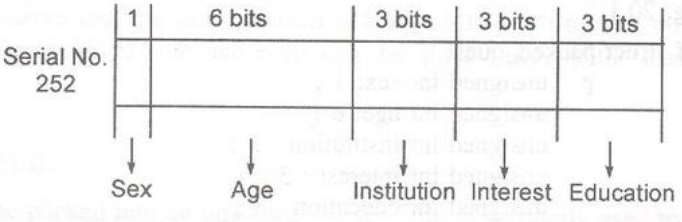

Fig. 20.1 Packing information in an unsigned int using bit fields.

The serial number is used as the array index in the above definition. In order to find the age of participant with serial number 285 we write:

participant_data [285].age

Observe that with 6 bits we can represent maximum age of 63. We thus have to assume that age of participants is within this. In a similar way the education may be found by writing

participant_data [285].education

Example Program 20.5 illustrates how one can retrieve required information from packed data.

```
/* Example program 20.5 */
#include <stdio.h>

main()
{
  typedef struct packed_quest
    {
      unsigned int sex : 1;
      unsigned int age : 6;
      unsigned int institution : 3;
      unsigned int interest : 3;
      unsigned int education : 3;
    } Quest;
  Quest participant_data[100];
  int i, no_quest;
  unsigned int serial_no, s, a, ins, intr, edu;
  scanf("%d", &no_quest);
  for (i = 1; i <= no_quest; ++i)
    {
      scanf("%u %u %u %u %u",
            &s, &a, &ins, &intr, &edu);
      participant_data[i].sex = s;
      participant_data[i].age = a;
      participant_data[i].institution = ins;
      participant_data[i].interest = intr;
      participant_data[i].education = edu;
      printf("address of participant[%d] = %x\n",
            i, participant_data[i]);
    }
  printf("Find the age of participant with Ser No = ");
  scanf("%u", &serial_no);
  printf("Age in record is %u\n",
        participant_data[serial_no].age);
  printf("Find education of participant with Ser No = ");
  scanf("%u", &serial_no);
  printf("Education in record is %u\n",
        participant_data[serial_no].education);
} /* End of main */
```

Program 20.5 Packing bits in a word.

The bit fields structure is machine and computer implementation dependent. Whether the fields are assigned in a word left to right or vice versa varies from machine to machine. In our Example we assumed that the first field defined in the structure occupies the most significant bit. This may not be so in all machines. The individual fields cannot be pointers or arrays. Use of bit fields should be done with extreme caution. Avoid it if you can.

EXERCISES

20.1 Two's complement of a number is obtained by scanning it from right to left and complementing all bits after the first appearance of a 1. Thus two's complement of 11100 is 00100. Write a function to find the two's complement of a binary number.

20.2 Write a function to find the binary equivalent of a given decimal integer and display it.

20.3 Write a function to find the decimal equivalent of a binary number.

20.4 Write a program to subtract a positive binary number *b* from another positive binary number *a*.

20.5 Write a program to multiply a signed 4 bit binary number *a* by another signed 4 bit binary number *b*.

20.6 Write a program to construct a single error correcting code called Hamming Code. For a 4 bit number, Hamming Code appends 3 more parity bits making it a seven bit code. The seven bits are labelled P_1, P_2, I_3, P_4, I_5, I_6, I_7.

P_1 satisfies even parity on bits P_1, I_3, I_5, I_7.
P_2 is even parity for bits P_2, I_3, I_6, I_7 and bit P_4 is even for parity bit for bits P_4, I_5, I_6, I_7.

20.7 Write a simulator for a small computer with the following characteristics:
8 bit word, 3 bit op code. The operations are read, write, AND, OR, NOT, shift accumulator, load accumulator, store accumulator.
Write a program for this simulated computer to add two 4 bit numbers.

20.8 Write a program to search whether a given bit string pattern appears in a 16 bit number. The inputs to the program are

(a) the given number, and
(b) the given pattern.

For example, the pattern 101 appears three times in the 16 bit number 0010100101101001.

21. Files in C

Learning Objectives

In this chapter we will learn:

1. How to create and store files on a disk using C language
2. How to create, update, and retrieve information from sequential files
3. The available features in C language to retrieve information directly from a file stored on disk using direct access feature.

So far we assumed that all data input to a program originate from the standard input device, namely, the keyboard of a video terminal. We have also assumed that all output information is displayed on the standard output device, namely, the screen of a video terminal. Thus we do not know at this time how data can be read from a device such as a disk and how results can be stored in a disk. The operating system under which the C programs run do provide facilities for reading data from disk or writing on disk. The C programming language by itself does not provide any input/output functions. These are provided either by the operating system or by standard libraries which can be accessed by a C programmer. We have so far been using the standard input/output library <stdio.h>. This library also provides facilities to store data in other devices such as disks and read data from other devices. In this chapter we will describe how data stored on *files* can be read.

21.1 CREATING AND STORING DATA IN A FILE

C assumes that input data appears as a sequence called a *stream* which can be stored in one or more devices connected to the computer. There are two types of streams: a stream of ASCII characters or a stream of bits. We will be mostly concerned with character streams.

We have so far used *scanf* statement to read data from the standard input. The equivalent statement for data to be read from a file is *fscanf*. Similarly corresponding to *printf* we have *fprintf*. There are equivalents also for *getchar* and *putchar*. We will discuss in this section how a name is given to a file, how it is made ready to store data, how data is stored in it and how it is closed when the job of storing data is over.

The first thing which is done is to declare a data type called FILE and a pointer to that file. We then give a name to the file and open it. This is shown below:

```
FILE *address_of_file;
address_of_file = fopen("sample.dat", "w");
```

The function fopen specifies that a file named "sample.dat" has been *opened* on the disk. The string "w" states that it will be used for *writing* data. The function fopen returns a pointer

to a structure of type FILE. We have named this pointer address_of_file. If the function fopen is unable to return an address (i.e. a pointer) to store the file then it returns NULL as the address. The function fopen may not be able to provide an address, for example, if the disk is full and no space is left in it. When an address is returned by fopen it means that the file named sample.dat may be stored starting at that address.

If the string "This is a sample string \n" is to be written in sample.dat, one way of doing it is to use the function fprintf. The statement used is:

fprintf(address_of_file, "This is a sample string \n");

The above statement will write in the file whose pointer is address_of_file the given string. "This is a sample string" and place an end of line marker \n at its end. If this is the only data to be written then we close the file with the statement:

fclose(address_of_file);

Gathering all the statements Example Program 21.1(a) is written. There is only one defect in this program. When fopen returns NULL then the program must give a message and stop. This is shown in Example Program 21.1(b).

```
/* Example Program 21.1a */
#include <stdio.h>

main()
{
  FILE *file_ptr;
  file_ptr = fopen("sample.dat", "w");
  fprintf(file_ptr, "This is a sample string\n");
  fclose(file_ptr);
} /* End of main */
```

Program 21.1(a) Creation and storing information in a file.

```
/* Example Program 21.1b */
#include <stdio.h>

main()
{
  FILE *address_of_file;
  address_of_file = fopen("sample.dat",  "w");
  if (address_of_file == NULL)
    {
      printf("Cannot open file to write sample.dat\n");
      return(0);
    }
  fprintf(address_of_file, "This is a sample string\n");
  fclose(address_of_file);
} /* End of file */
```

Program 21.1(b) Giving message if a file cannot be opened.

If we want to read the file sample.dat and display its contents on the standard output device we write Example Program 21.2. In this program we define type FILE and a pointer named pointer_to_file. Observe that in the fopen function we use a string "r".

```
/* Example Program 21.2 */
#include <stdio.h>
#define ARRAY_SIZE      80

main()
{
  FILE *pointer_to_file;
  char temp[ARRAY_SIZE];
  pointer_to_file = fopen("sample.dat", "r");
  if (pointer_to_file == NULL)
    {
      printf("Cannot open sample file to read\n");
      return(0);
    }
  fgets(temp, 81, pointer_to_file);
  if (temp == NULL)
    {
      printf("Error in reading file sample.dat\n");
      return(0);
    }
  printf("%s", temp);
  fclose(pointer_to_file);
} /* End of main */
```

Program 21.2 Printing contents of a file.

This specifies that the file is opened for reading. The fopen function returns the address of the file and stores it in pointer_to_file. If the file cannot be opened a message is given. If the file is successfully opened the function fgets is used to read the string stored in the file. fgets is similar to gets. The only difference is that in fgets we specify the pointer to the file from where data is to be read. In gets function we used in previous chapters, no pointer to file is specified as standard input is assumed. In Example Program 21.2 the statement used is:

fgets(temp, 81, pointer_to_file);

temp is an array with maximum 81 characters into which the data is read from the file.

If for some reason fgets is not able to access the file (may be it has been erased) then pointer_to_file will be NULL. In this case a message is displayed and main is exited. If the file is successfully read then the string is displayed by the printf function. After displaying the contents of the file, the file is closed.

The general form of fopen function is fopen (string giving file name, string giving option) the options available are given in Table 21.1.

Table 21.1 Options in fopen

File opening option	Action carried out	Does the file exist?
"w"	opens new file for writing	No. Newfile is created
"r"	opens file for reading	Yes. If no, it is an error
"a"	opens file for appending	Yes. If no it is not an error. File is created
"r+"	opens file for reading and/or writing	Yes. If no it is an error
"w+"	opens file for reading and for writing at the end of the file	Yes

Example 21.1

We will now write programs to solve the following problem. A list of names of students in a class is given, one name per line as shown below:

AGARWAL, P.R.
ARUN, R.S.
BHASKAR, A.V.
CHANDRU, K.
GANDHI, S.K.
RAJU, P.
SANTHANAM, R.
SUNANDA, A.
SURESH, P.
TATA, P.
USHA, R.
VASU, P.
ZAIDI, B.S.

The names do not have any embedded blanks. They are already sorted in alphabetical order. It is required to:

1. Store the names in a file on disk.
2. Display the n^{th} name in the list (*n* is data to be read)
3. Display all names starting with S.

In Example Program 21.3 we first define a data type FILE and f_ptr as a pointer to a file. We then declare name as an array with at most 30 characters. We have thus assumed that no name can be longer than 30 characters. Next we open a file called name.dat with the option of reading or writing in it. The beginning of the file is pointed to by f_ptr. After doing this we read names from the standard input, namely, the keyboard and write it in the disk file using the function fprintf(f_ptr,"%s\n", name). After writing all the names in the file it is closed. We now have the names stored in a disk file name.dat whose pointer is f_ptr.

```
/* Example Program 21.3 */
#include <stdio.h>
#define   NAME_SIZE   30

main()
{
  FILE *f_ptr;
  char name[NAME_SIZE];
  int i, n;

  f_ptr = fopen("name.dat", "w+");
  if (f_ptr == NULL)
    {
      printf("File name.dat cannot be opened\n");
      return(0);
    }
  scanf("%d", &n);   /* n th name to be displayed */
   /* read name from keyboard input */
  while(scanf("%s", name) != EOF)
   /* store in name.dat */
    fprintf(f_ptr, "%s\n", name);

  /* The names have now been stored in a disk file
     'name.dat' whose pointer is f_ptr  */
  rewind(f_ptr);

  /* The following statement reads and skips
     the first (n - 1) names */

  for (i = 1; i < n; ++i)
      fscanf(f_ptr, "%s", name);
  /* The file pointer is now pointing to */
  /* the nth name in file */
  /* Get nth name in file */
  fscanf(f_ptr, "%s", name);
  printf("%dth name = %s\n", n, name);
  /* The pointer pointing to name is */
  /* reset to point to the first name */
  rewind(f_ptr);
  while (fscanf(f_ptr, "%s", name) != EOF)
    {
      if (name[0] < 'S')
        continue;
      if (name[0] == 'S')
        printf("Names starting with S = %s\n", name);
      else
        break;
    }
  fclose(f_ptr);
} /* End of main */
```

Program 21.3 Printing n^{th} name and names starting with S from a file of names.

The problem given is to display the n^{th} name. The names will be stored in name.dat as shown in Fig. 21.1 with the file pointer as shown. Each time we read a name in the file the file pointer will advance by one position.

Top_of_file	→ AGARWAL.P.R	AGARWAL.P.R.	AGARWAL.P.R.
f_ptr	ARUN.R.S.	ARUN.R.S.	ARUN.R.S.
	- - - - -	BHASKAR.A.V.	- - - - -
	- - - - -	→ CHANDRU.K.	- - - - -
	- - - - -	- - - - -	- - - - -
End of	- - - - -	- - - - -	- - - - -
file	ZAIDI. B.S.	ZAIDI. B.S.	→ ZAIDI. B.S.
	Pointer at file beginning	Pointer after reading 3 names	Pointer after reading all names

Fig. 21.1 Illustrating position of file pointer.

Thus in order to read the n^{th} name in the list we have to skip the first (n – 1) names. This is achieved by the *for* statement for (i = 1; i < n; ++i). After skipping (n – 1) names the pointer is positioned at the n^{th} name. This is read by the function fscanf (f_ptr, "%s", name). Observe that fscanf is similar to scanf. The only difference is, in scanf we do not specify a file pointer as it is assumed to be the standard input from keyboard. In fscanf we specify the pointer to the file from which data is to be read. Having read the n^{th} record it is displayed using printf function. We take the pointer back to the top of the file by using the function *rewind(f_ptr)*.

The third problem posed is to display all names starting with S. We are given that the names are in alphabetical order. Thus in the *while* loop we read a name and if its first character is < S (i.e., A to R) we go back and advance the pointer to the next name. This is achieved by the continue statement which returns control to *while*. If the first letter in name is S we display the name. Thus all names starting with S will be displayed one by one. When the first letter is > 'S' we leave the *while* loop using the break statement. Finally we close the file.

We summarise in Table 21.2 the file functions we have used so far. A large number of useful programs can be written using only these functions.

21.2 SEQUENTIAL FILES

A sequential file is a data structure which consists of a sequence of records of the same type and size. The records in the file can be read only sequentially, that is, one after the other starting from the beginning of the file. An array in contrast to a sequential file has a fixed predetermined number of components. Any arbitrary component of an array may be accessed using an index. The primary advantage of a sequential file compared to an array is that it can grow or shrink dynamically. The disadvantage is sequential access. If the tenth record of a sequential file is to be read the first nine records must be passed over before reaching the tenth record.

Table 21.2 Important File Functions

Function	Use of function	Syntax
fopen	To open a file	fopen ("file name string", "option"); Example: fopen("Sample.dat", "r+"); Returns pointer to file "Sample.dat"
fclose	To close file. Clean up buffers used by it Reposition pointer to beginning of file	fclose (pointer_to_file); Example: fclose(f_ptr);

fscanf	Read from specified file formatted data	fscanf(file pointer, "format string", address of identifiers);
		Example:
		fscanf(f_ptr, "%c, %d, %s", &a_char, &number, string);
		(string is name of array)

fprintf	Write in specified file formatted data	fprintf(file pointer, "format string", names of identifiers);
		Example:
		fprintf(f_ptr, "%c, %d, %s", a_char, number, string);
		(String is name of array where a character string is stored)

rewind	Repositions file pointer to the start of the file	rewind (file pointer); Example rewind(f_ptr);

Example 21.2

We will now illustrate the creation of a sequential file in C. We will assume that the structure of each record is as shown below:

Roll No.	Name	Marks 1	Marks 2	Marks 3
integer >0	25 chars	integer >=0	integer >=0	integer >=0

As records in a sequential file can be retrieved only in a rigid order it is necessary to arrange them systematically using one of the unique fields in the record as a key. Usually the records in the file are arranged in ascending or descending order of this *key field*. In this example Roll Number is the appropriate field to be used as the keyfield as it is unique to each student. We will assume that the records are arranged in ascending order of this key. The file we will create will be a sequential file with records arranged in ascending order of roll number. Before creating a file it is given a name. This file which we will call infile.dat should be initialized ready for writing by the statement:

fopen("infile.dat", w+);

After this a record is read from the standard input and put in infile.dat. The read and store operations are repeated till no data is left in the input. This method is illustrated as

Example Program 21.4. This program is developed using five functions. We first declare a global struct stud_record and a global FILE *infile_ptr. In the function create_file we first open a file infile.dat whose pointer is infile_ptr. We then read one record after another and store it in the file using function store_in_file. The end of input records is signalled by an end of file record which has 0 stored in roll_no field. The function print_file_record opens the file in which data has been written. Each record is read by the function read_file_record. After reading a record it is displayed by a series of printf statements in the function print_file_record. The function feof(infile_ptr) returns 0 as long as the end of infile.dat is not reached. This is used in a while loop to display records. When all records have been displayed the file is closed. The display of stored records is done to check whether data read has been correctly recorded in the file.

```
/* Example Program 21.4 */
#include <stdio.h>
struct stud_record
{
  unsigned int roll_no;
  char name[25];
  unsigned int marks[6];
};
struct stud_record temp;

FILE *infile_ptr;
void read_record(void);
void store_in_file(void);
void create_file(void);
void read_file_record(void);
void print_file_records(void);

main()
{
  create_file();
  print_file_records();
} /* End of main */

void read_record(void)
{
  scanf("%d %s %d %d %d", &temp.roll_no, temp.name,
       &temp.marks[1], &temp.marks[2], &temp.marks[3]);
} /* end of read_record */

void store_in_file(void)
{
  fprintf(infile_ptr, "%d %s %d %d %d\n", temp.roll_no,
        temp.name, temp.marks[1], temp.marks[2],
        temp.marks[3] );
} /* End of store_in_file */
```

```
void create_file(void)
{
  infile_ptr = fopen("infile.dat", "w+");
  if (infile_ptr == NULL)
    {
      printf("Error in opening infile.dat
              while creating\n");
      return;
    }
  read_record();
  while(temp.roll_no != 0)
    {
      store_in_file();
      read_record();
    }
  fclose(infile_ptr);
} /* End of create_file */

void read_file_record(void)
{
  fscanf(infile_ptr,"%d %s %d %d %d", &temp.roll_no,
         temp.name, &temp.marks[1], &temp.marks[2],
         &temp.marks[3]);
} /* End read_file_records */

void  print_file_records(void)
{
  int i;
  infile_ptr = fopen("infile.dat", "r");
  if (infile_ptr == NULL)
    {
      printf("Error in opening infile.dat for printing\n");
      return;
    }
  read_file_record();
  while(!feof(infile_ptr))
    {
      printf("%d %s ", temp.roll_no, temp.name);
      for (i = 1; i <= 3; ++i)
       printf("%d ", temp.marks[i]);
      printf("\n");
      read_file_record();
    }
  fclose(infile_ptr);
} /* End of print_file records */
```

Program 21.4 Creating a sequential file of student records.

Example 21.3: Searching a sequential file

We will now develop a function to retrieve a record with a specified key from a file. We will assume that the file is arranged in ascending order of the key field which is the roll number. A procedure to do this is given as Example Program 21.5. The key of the record to be retrieved is passed as the actual argument sr_no to the function search. In the *while* loop a record is read from infile.dat. The key of this record is compared with the key of the record to be retrieved (which is in desired_roll_no). If they match no further search is needed (as the key is assumed unique). The control leaves the function search after details of the retrieved record are printed and the file is closed. If the key field read from the sequential file becomes greater than the search key without the two being equal at any point during search it means that the key being searched is not in the file. We can come to this conclusion as the file is

```
/* Example Program 21.5 */
/* Searching a sequential file */
/* The same global declarations as in Example program
   21.4 are assumed. It is assumed that infile.dat is
   created as shown in Example program 21.4 */
#include <stdio.h>

struct stud_record
{
  unsigned int roll_no;
  char name[25];
  unsigned int marks[6];
};
struct stud_record temp;

FILE *infile_ptr;
void read_file_record(void);
int search(int desired_roll_no);

main()
{
  int sr_no;
  scanf("%d", &sr_no);
  search(sr_no);
} /* End of main */

int search(int desired_roll_no)
{
  int i;
  infile_ptr = fopen("infile.dat", "r");
  if (infile_ptr == NULL)
    {
      printf("Error in opening infile.dat for searching\n");
      return(0);
    }
  read_file_record();
```

```
    while (!feof(infile_ptr))
      {
       if (temp.roll_no == desired_roll_no)
         {
          printf("%d %s", temp.roll_no, temp.name);
          for (i = 1; i <= 3; ++i)
            printf("%d ", temp.marks[i]);
          printf("\n");

          fclose(infile_ptr);
          return(0);
         }
       else
        read_file_record();
      }
    printf("Roll no = %d not in file\n", desired_roll_no);
} /* End of search */

void read_file_record(void)
{
  fscanf(infile_ptr,"%d %s %d %d %d",
        &temp.roll_no,
        temp.name,
        &temp.marks[1],
        &temp.marks[2],
        &temp.marks[3]);

} /* End of read_file_record */
```

Program 21.5 Retrieving information from a sequential file.

arranged in ascending order of keys. When such a condition is detected further search is abandoned and an appropriate message is printed.

It should now be evident why sequential files are arranged strictly in ascending or descending order of the key.

Example 21.4 Updating a sequential file

In Example Program 21.2 each student's record had marks in 3 subjects. Suppose the marks in a fourth subject become available. A program is required to do the following:

(i) Include the marks in a fourth subject in each student's record in infile.dat
(ii) Compute the total marks in four subjects and include it in each student's record in the file
(iii) Print each student's record

We will assume that the marks in the fourth subject are input using a keyboard in ascending order of roll numbers.

The program to be written is known in computer jargon as updating the records in a master file with new information from a transaction file and creating a new master file. The

updating program is given as Example Program 21.6. At the beginning of the function update_file the master file, namely, infile.dat is opened. A file named outfile.dat is made ready to be written into by opening it. A record from infile.dat is now read.

A *while* loop is initiated in which records from infile.dat are read and processed until the end of the file infile.dat is reached. The function feof(infile.ptr) returns 0 when the end of the file infile.dat (pointed to by infile.ptr) is reached. After reading a record from infile.dat the roll_no and marks in the fourth subject of a candidate are read. The roll_no of the record read from infile.dat is compared with the roll_no read from terminal. If the two match total marks are computed by adding marks[1], marks[2], marks[3] and marks[4] and storing the result in marks[5]. The updated record consisting of roll_no, name and marks[1] to marks[5] are written in the structure struct stud_record temp which is declared as global in Example Program 21.6. If roll_no read from terminal does not match that read from infile.dat no updating is done and no record is written in outfile.dat. An appropriate message is written.

```
/* Example Program 21.6 */
/* Updating a file which is already created */
#include <stdio.h>

struct stud_record
{
  unsigned int roll_no;
  char name[25];
  unsigned int marks[6];
};
struct stud_record temp;

FILE *infile_ptr;
FILE *outfile_ptr;  /* This is a new declaration */
void read_file_record(void);
void update_file(void);

main()
{
  update_file();
}

void read_file_record(void)
{
  fscanf(infile_ptr,"%d %s %d %d %d",
         &temp.roll_no,
         temp.name,
         &temp.marks[1],
         &temp.marks[2],
         &temp.marks[3]);
} /* End of read_file_record */

   void update_file(void)
   {
     int i, fourth_mark, roll_no;
```

```
    infile_ptr = fopen("infile.dat", "r");
    if (infile_ptr == NULL)
      {
        printf("Error in opening infile.dat for
                updating\n");
        return;
      }
    outfile_ptr = fopen("outfile.dat", "w");
    if (outfile_ptr == NULL)
      {
        printf("Error in opening outfile.dat for
                updating\n");
        return;
      }
    read_file_record();
    while (!feof(infile_ptr))
      {
       /* marks[5] has sum of marks[1] to marks[4] */
        temp.marks[5] = 0;
        scanf("%d %d", &roll_no, &fourth_mark);
        if (temp.roll_no == roll_no)
         /* Add marks in 4 subjects and find total */
         {
           temp.marks[4] = fourth_mark;
           for (i = 1; i <= 4; ++i)
             temp.marks[5] += temp.marks[i];
           /* Write new record in outfile.dat */
           fprintf(outfile_ptr, "%d %s ", temp.roll_no,
                   temp.name);

           for (i = 1; i <= 5; ++i)
             fprintf(outfile_ptr, "%d ", temp.marks[i]);
           fprintf(outfile_ptr, "\n");
       /* Writing of one record in outfile.dat complete */
         }
        else
         {
           printf("Transaction roll_no does not match\n");
           printf("Transaction roll_no = %d,
           File roll_no = %d\n", roll_no, temp.roll_no);
         }
        read_file_record();
      } /*  End of while */

    fclose(infile_ptr);
    fclose(outfile_ptr);
}
```

Program 21.6 Updating a file.

The next record is now read from infile.dat and the while loop is repeated. Finally both infile.dat and outfile.dat are closed. Observe that struct stud_record has already been defined globally with the array variable name having 6 components 0, 1, 2, 3, 4, 5. This will thus suffice to store outfile.dat.

21.3 UNFORMATTED FILES

In the last section we discussed how files may be created with a format specification. Formatting is mainly needed for convenience of reading data from terminal and printing results on a printer or displaying on video display units. Data from main memory may be stored in binary form in disk files as exact replicas of how they are found in memory. Similarly unformatted binary data may be stored in memory from files. The operation of reading data from main memory and writing in a file in binary form is performed by a function *fwrite*. Reading binary stream of data from a file and storing in memory is performed by a function *fread*.

The format of the fwrite function is:

fwrite (address of structure or array in memory from where data is to be read,
size of structure or array,
Number of items to be read,
pointer of the file where data is to be written)

For example, if we want to read an array called buffer of size SIZE and write in file whose pointer is f_ptr we write:

```
fwrite(buffer, SIZE, 1, f_ptr);
```

(Remember that buffer is an array and thus its name itself is a pointer).

The format for the fread function is:

fread (address of structure or array in memory where data is to be stored,
size of structure or array,
Number of items to be stored,
pointer of the file from which data is to be read)

For example, if we want to store in a structure named cust_record data from file whose pointer is a_ptr we write:

```
fread(&cust_record, sizeof(cust_record), 1, a_ptr);
```

Disk is an addressable storage device. Thus files stored in a disk are addressable. C language has recognized this and has provided function called *fseek* to directly access a record in a file. *fseek* assumes that the file is a binary file created by *fread*. It will not work with a formatted file. We will now illustrate the use of *fread, fwrite* and *fseek* functions by rewriting Example Program 21.3 using *fread* and *fwrite*. Example Program 21.7 illustrates retrieving n^{th} record directly using *fseek* function.

The general form of the *fseek* function is:

fseek(pointer to file, number of bytes to be skipped from the position x of the file, k);

when $k = 0$ position x is taken as the beginning
when $k = 1$ position x is current position
when $k = 2$ position x is end of file

```
/* Example Program 21.7  */
/* Use of fread, fwrite, fseek illustration */

#include <stdio.h>
#define   NAME_SIZE   30

main()
{
  FILE *f_ptr;
  char name[NAME_SIZE];
  int n;

  f_ptr = fopen("name.dat", "w+");
  if (f_ptr == NULL)
    {
      printf("File name.dat cannot be opened\n");
      return(0);
    }
  scanf("%d", &n);   /* n th name to be displayed */
 /* read name from keyboard input */
  while(gets(name) != NULL)
   fwrite(name, NAME_SIZE, 1, f_ptr);
 /* The names have now been stored in a disk file
    'name.dat' whose pointer is f_ptr  */
  rewind(f_ptr);

  fseek(f_ptr, (long)(NAME_SIZE * n), 0);
  fread(name, NAME_SIZE, 1, f_ptr);
  puts(name);
  fclose(f_ptr);
} /* End of main */
```

Program 21.7 Direct access of information from a disk file.

Example 21.5

A bank gives account numbers in serial order starting with serial number 1. The structure of a customer record is:

Account No. 4 digits
Balance in acct : XXXXX.XX (Rs & ps)
Account active or closed : 1 digit (1 for active 0 for closed)

We will write a program to

1. Store customer records in a file.

2. Retrieve the record of a customer with a specified account number and display it. Update it (if required) and store it back in file.

3. Append a new customer record to the file.

Example Program 21.8, 21.9 and 21.10 are written to do this.

```
/* Example Program 21.8 */
/* Creates an account file */
#include <stdio.h>
struct cust_record
{
  unsigned int acct_no;
  float balance;
  unsigned short int status;
};
/* Customer accts record declaration */
struct cust_record cust_acct;
FILE *afil_ptr;
void create_accounts_file(void);

main()
{
  create_accounts_file();
}

void create_accounts_file(void)
{
  /* A file "accts.dat" opened to store customer accounts */
  afil_ptr = fopen("accts.dat", "w+");
  if (afil_ptr == NULL)
    {
      printf("error in opening accts.dat file\n");
      return;
    }
  /* Customer record is read and stored in file */
  while(scanf("%u %f %hu", &cust_acct.acct_no,
        &cust_acct.balance, &cust_acct.status) != EOF)
    fwrite(&cust_acct, sizeof(struct cust_record), 1,
          afil_ptr);
  /* The file is rewound to point at the beginning */
  rewind(afil_ptr);
}
```

Program 21.8 Creating a file of customer accounts.

In Example Program 21.8, we first define a structure to store customer records and declare globally cust_acct and *afil_ptr. We open the file accts.dat and store customer records in it in binary form and rewind it.

In Example Program 21.9 we read from the terminal the account_no of the customer whose record is to be read. The statement:

fseek(afil_ptr, (long)((acct_no – 1)*s_size), 0);

advanced the file pointer[(acct_no – 1)*sizeof (struct cust_record)] bytes from the beginning of the file (s_size = sizeof(sruct customer_record)). We have assumed that account numbers are in strict serial order. The number of bytes in each customer record is obtained by using

```
/* Example Program 21.9 */
/* Illustrating direct access to records in a file */
#include <stdio.h>
struct cust_record
{
  unsigned int acct_no;
  float balance;
  unsigned short int status;
};
struct cust_record cust_acct;
FILE *afil_ptr;
/* Use 'accts.dat' created in Example Program 21.8 */
 main()
{
  unsigned int account_no;
  float amount;
  unsigned short int trans_type;
  int s_size;

  scanf("%u %f %hu", &account_no, &amount, &trans_type);
  s_size = sizeof(struct cust_record);
  afil_ptr = fopen("accts.dat", "r");
  if (afil_ptr == NULL)
    {
      printf("Error in accts.dat file\n");
      exit(1);
    }
  printf("record of customer with acct.no = %u\n", account_no);
  /* The file  pointer moved to record with specified
     acct.no */
  fseek(afil_ptr, (long)((account_no - 1) * s_size),  0);
  /* Read and display record */
  fread(&cust_acct, s_size, 1, afil_ptr);
  printf("Account no = %u, Balance = Rs %.2f",
          cust_acct.acct_no, cust_acct.balance);
  if (cust_acct.status)
    printf("    Active account\n");
  else
    printf("    Account closed\n");
    /* End of displaying customer account details */
    /* Updating the account of a customer record */
    /* trans_type = 1 is for deposit, trans_type = 0
     for withdraw */
  if (cust_acct.status)  /* If account is active */
    {
      printf("Customer Acct.No = %u is being updated\n",
              cust_acct.acct_no);
      if (trans_type)
        cust_acct.balance += amount;  /* if deposit */
      else
        {
```

```
            cust_acct.balance -= amount; /* if withdrawal */
            if (cust_acct.balance < 0)
              {
                printf("Balance negative, withdrawal not
                        allowed\n");
                cust_acct.balance += amount;
              }
          }
        printf("New balance in acct = Rs %.2f\n",
                cust_acct.balance);
      }
    else
      printf("Customer account closed - No transactions
              allowed\n");
     /* Updated record is stored back in the file by the
        following statement */

    fwrite(&cust_acct, s_size, 1, afil_ptr);
    fclose(afil_ptr);
  } /* End of main */
```

Program 21.9 Directly accessing a customer accounts record and updating it.

the *sizeof* operator. The multiplying factor is (acct_no – 1) because when acct_no = 1, (acct_no – 1) = 0 and the file pointer points to the first record in the file. Thus to point to the k^{th} record the number of bytes to be skipped = (k – 1) * no.of bytes in each record. Now that the file pointer points to the record with the specified acct_no we read the record from the file and print it. We also give a message if the account is closed. Having displayed the account details if a deposit or a withdrawal is specified this is carried out as shown in the program. The updated record is stored back in the file and the file is closed.

Next we see in Example Program 21.10 how a new record is appended to the file. The file accts.dat is opened with the option to read and append (String "r+" in open specifies this). We use the function *fseek* to go to the end of the file. The argument 2 in fseek specifies that the pointer should advance to the end. To find out the last account number assigned we take the pointer backwards by one record length as specified by s_size from the current position. We read the last record stored in the file and display it. We add 1 to the customer acct_no to get the next customer acct_no to be assigned. We feed from the terminal the deposit made by the customer and store it as the appended record in the file. The file is then closed.

21.4 TEXT FILES

A file which is a string of characters is known as a text file. There are simple functions available to:

1. Read characters from a file
2. Write characters in a file
3. Read a string of character from a file
4. Write a string of characters in a file

We will illustrate use of some of these functions in this section.

```
/* Example Program 21.10 */
/* Adding a new customer account at the end of file */
#include <stdio.h>

struct cust_record
{
  unsigned int acct_no;
  float balance;
  unsigned short int status;
};

/* Customer accts record declaration */
struct cust_record cust_acct;
FILE *afil_ptr;

main()
{
  int s_size, items;

  /* File acct.dat opened for apppending at end */
  afil_ptr = fopen("accts.dat", "r+");
  if (afil_ptr == NULL)
    {
      printf("File 'accts.dat' cannot be opened\n");
      return;
    }
  s_size = sizeof(struct cust_record);
  printf("s_size = %d\n", s_size);

  /* Position the file pointer to end of file */
  /* This is done using 2 for the last field of fseek */
  fseek(afil_ptr, (long)(-s_size), 2);
  /* This puts pointer to point to the last record */
  items = fread(&cust_acct, s_size, 1, afil_ptr);
  printf("Items read = %d\n", items);
  printf("Last account number = %u\n", cust_acct.acct_no);
 /* Next account no = current no + 1 */
 cust_acct.acct_no ++;
 printf("New customer account being appended = %u\n",
cust_acct.acct_no);
 scanf("%f", &cust_acct.balance); /* Real deposit made */
 cust_acct.status = 1;
 fwrite(&cust_acct, s_size, 1, afil_ptr);
 close(afil_ptr);
}
```

Program 21.10 Appending a new record to a file.

Example 21.6

A text file is stored in a file infile.txt. It is to be copied to another file outfile.txt after squeezing out all blanks in infile.txt. A program to do this is written as Example Program 21.11. Observe the use of functions fgetc and fputc in the program. The program is self-explanatory. There are many more functions to handle input/output of files.

```
/* Example Program 21.11 */
#include <stdio.h>
FILE *in_ptr, *out_ptr;
main()
{
  char temp;
  in_ptr = fopen("infile.txt", "r");
  if (in_ptr == NULL)
    {
      printf("infile.txt cannot be opened\n");
      return(0);
    }
  out_ptr = fopen("outfil.txt", "w");
  if (out_ptr == NULL)
    {
      printf("oufil.txt cannot be opened\n");
      return(0);
    }
  /* Read a character from "infile.txt" into temp */
  temp = fgetc(in_ptr);
  while(temp != EOF)
    {
      if (temp != ' ')
        /* store temp in outfil.txt if it is not
           a blank character */
        fputc(temp, out_ptr);
      temp = fgetc(in_ptr);
    }
  rewind(in_ptr);
  rewind(out_ptr);
  close(in_ptr);
  close(out_ptr);
} /*  End of main */
```

Program 21.11 Reading a text file and manipulating it.

We will now discuss reading unformatted text files and writing unformatted text files. We will illustrate the use of *fread* and *fwrite* functions by showing how to copy a file to another file. Example Program 21.12 illustrates this. This program defines a function copy with two arguments. The two arguments are pointers to two files. We define a character

```
/* Example Program 21.12 */
#include <stdio.h>
#define ARRAY_SIZE    79

main()
{
  copy ("a.txt", "b.txt");
}/* End of main */

void copy(char *a_f_name, char *b_f_name)
{
  int bytes_read;
  char buffer[ARRAY_SIZE];
  FILE *a_fil, *b_fil;
  a_fil = fopen(a_f_name, "r");
  if (a_fil == NULL)
    {
      printf("cannot open a.txt file \n");
      return;
    }
  b_fil = fopen(b_f_name, "w");
  if (b_fil == NULL)
    {
      printf("cannot open b.txt file \n");
      return;
    }
  while((bytes_read = fread(buffer, sizeof(char),
         sizeof(buffer), a_fil)) != 0)
    fwrite(buffer, sizeof(char), bytes_read, b_fil);
  close(a_fil);
  close(b_fil);
}
```

Program 21.12 Copying a text file.

buffer which can store 80 characters. The two files are opened. *fread* reads 80 characters into the buffer from the file pointed to by a_fil and *fwrite* writes the characters stored in buffer to the file pointed to by b_fil. When end of file a.txt is reached the two files are closed.

There are many more functions to manipulate files in C. We will not discuss them in detail. They are listed in an Appendix.

EXERCISES

21.1 A structure item_in_store was defined in Section 16.1. Create an INVENTORY FILE with these records. Write a procedure to find the total value of the inventory in the store.

21.2 Suppose a store has a number of items in their inventory and that each item is supplied by at most two suppliers. Create inventory and supplier files. Find the addresses of all suppliers who supply more than 10 different items. Discuss any changes in data structure you would suggest to simplify solving this problem.

21.3 A student master file was created in Example 21.3. Write a program which will read this file and print out a list of students who have failed in one or more subjects. Assume 40 percent as pass marks.

21.4 An updated record was created in Example 21.4. Using this file compute for each student his average marks and class. (Assume that 40 to 49 percent is III class, 50 to 59 is II class and above 60 is first class.) Update the file and create a new master file with this additional information.

21.5 Arrange the master file in descending order of average marks and create a new file.

21.6 Assume that at the end of the year a set of students join the class and another set leaves. Using the roll number and an appropriate code to add or delete a student, update the master file. The updated file should be in ascending order of the roll number.

21.7 In Exercise 21.4 a master file was created with average marks and class in addition to other information. Using this file create another file which has only ROLL NO, average marks and class. At the end of the file add a last record which has the total number of records in the file, the number passing in I class, II class, III class and the overall class average. Print this summary information.

21.8 Assume that there are two sections of students who took the same examination and their Roll-No and total marks obtained are recorded in two sequential files.

(i) Write a program to merge the two sequential files assuming that the two files are arranged in strict ascending order of their Roll-No.

(ii) Modify the program to reject records which are out of sequence (that is, not in strict ascending order) and print an appropriate error message.

21.9 In a small firm employee numbers are given in serial numerical order, that is 1, 2, 3 etc.

(i) Create an unformatted file of employee data with following information:

employee no, name, sex, gross salary

(ii) If more employees join append their data to the file.

(iii) If an employee with serial no. 25 (say) leaves, delete it by making gross salary 0.

(iv) If some employee's gross salary is increased by 15 percent, write a program to retrieve the records and arrange them. For example if the salaries of employees with serial numbers 15, 23, 28, 42 is increased, the new file must have the new salaries.

(v) The company has two sections. The sections are expected to give distinct employee serial numbers and create files with serial number and gross salary. Merge the two files. If there are duplicates appropriate error messages should be printed.

21.10 Write a program corresponding to Exercise 21.1 with an unformatted file.

21.11 Given a text file, create another text file deleting all the vowels (a, e, i, o, u).

21.12 Given a text file, create another text file deleting the words *a, the, an* and replacing each one of them with a blank space.

22. Miscellaneous Features of C

Learning Objectives

In this chapter we will learn:

1. Some shorthand notations available in C to write concise C programs. In particular we will discuss the use of conditional and comma operators
2. We will see how macros can be written in C
3. Finally we will discuss the use of command line arguments to run C programs with many data sets and parameter sets

In this chapter we will present a variety of features available in C which we have not discussed so far in this book. Some are shorthand notations for operations we could do using the features of C we already know. Others are useful for writing large programs or for combining programs written by a team of programmers.

22.1 CONDITIONAL OPERATOR

We have been using the conditional statement

```
if (expression 1)
   (expression 2) ;
else
   (expression 3) ;
```

If (*expression 1*) is true then (*expression 2*) is executed, *else* (*expression 3*) is executed. The statement can be written in an alternative form (which is less readable) using as a *conditional operator.*

```
(expression 1) ? (expression 2): (expression 3) ;
```

The above statement is interpreted as :

```
if (expression 1) ≠ 0 then
   perform (expression 2)
else
   perform (expression 3)
```

If we write:

```
max = (x > y) ? x : y ;
```

max has the larger of x and y when the above statement is executed. The following statement finds whether an integer x is odd or even.

```
(x%2) ? printf("x=%d is odd\n", x):printf(" x =%d is even \n", x) ;
```

22.2 COMMA OPERATOR

A new expression may be formed from a sequence of other expressions by separating them by a comma , . The expressions in the sequence are evaluated from left to right. If one is interested in knowing the value of the whole expression it is that of the rightmost in the sequence. In the following example:

```
q = 4; r = 7 ;
p = q + 2, r + 1, q + r ;
```

As q + 2 = 6 and r + 1 = 8, q + r = 6 + 8 = 14. This value is assigned to p.

The comma operator is most often used in for loop expressions. For example:

```
for (i = 1, sum = 1 ; i <= 4 ; sum += i, i += 2)
        /* NULL statement */ ;
```

is equivalent to

```
sum = 1;
for ( i = 1 ; i <= 4; i += 2)
  sum += i;
```

Observe that in the version of *for* using comma operator all the work needed in the looping is done in the *for* statement itself with no other statement in the domain of the loop. As a *for* loop requires a statement we put a semicolon to indicate the termination of the loop. This is called a NULL statement.

In Example Program 22.1 we illustrate the use of conditional operator and comma operator. The function palindrome checks whether a string reads the same whether read from left to right or right to left (ABBA is, for example, a palindrome). In the *for* loop we compare pairwise the left and right most characters in the given string. As soon as one mismatch occurs control leaves the loop and FALSE is returned. Observe how the left and right indices are simultaneously set. Also the left index is incremented and the right index decremented simultaneously in the *for* loop. This program is concise but not easy to understand. We do not recommend this style to beginners.

22.3 MACRO DEFINITION

C provides a pre-processor which processes the source code file before it is compiled. The instructions to the preprocessor which are placed before the program are known as preprocessor directives. We have already been using two such directives. Examples of these are:

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

and

```
#define TRUE 1
```

A directive starts with the symbol #. Only one directive is allowed per line. The directive is *not* terminated by a semicolon.

The #define directive we have used so far is of the type

```
#define identifier constant
```

The #define directive is in fact more general. For instance if a message is to be printed often we can write:

```
#define ASK_ROLL_NO printf("\n Type your Roll_no\n")
```

```
/* Example Program 22.1 */
/* Check if a string is a palindrome */
#include <stdio.h>
#include <string.h>
#define   TRUE     1
#define   FALSE    0
#define   EQUALS   ==
typedef unsigned int Boolean;
Boolean palindrome(char str[]);

main()
{
  char given_string[80];
  /* Given string should not have any blanks */
  while (scanf("%s", given_string) != EOF)
    palindrome(given_string) ?
      printf("Given string is a palindrome\n") :
      printf("Given string is not a palindrome\n");
} /* End of main */

Boolean palindrome(char str[])
{
  Boolean match_till_now = TRUE;
  int left, right;
  for (left = 0, right = strlen(str) -1;
       (left < right) && match_till_now;
       left++, right--)
    match_till_now = (str[left] EQUALS str[right]);
  return(match_till_now);
} /* End of palindrome */
```

Program 22.1 Use of conditional and comma operators.

In a program we can then write

```
main( )
{ ————————
  ————————
  ASK_ROLL_NO; /* a MACRO */
  ————————
}
```

This definition is called a *macro*. Macros can also have parameters. For example,

```
#define ABS(x) x < 0 ? – (x) : x
```

Observe that we have used – (x) instead of – x as when an argument is substituted the text of the argument is substituted. Thus if x = 4 – y if we had written:

```
x < 0 ? – x : x
```

the answer would be – 4 – y instead of – 4 + y. Observe that *x* can be of any type.

Another macro is

```
#define SQUARE(x) (x) * (x)
```

Again parentheses around x in the macro definition are required. If we had written x*x without parentheses then if x = p + 2 the SQUARE(x) would be computed as:

p + 2 * p + 2 which is not the same as (p + 2) * (p + 2)

Observe that in macros textual substitution is done. If this fact is forgotten the result will be wrong. It is very difficult to detect such an error. Thus MACROS should be used with care.

22.4 UNION

There is a declaration called union in C which is used when, for example, one would like to store different data types in a variable or an array. The use of union saves storage space but should be used with care. An example of union declaration is:

```
union any_type
    { int i ;
      char c ;
      float f ;
};
```

Observe that the declaration is somewhat like the definition of struct. Unlike a struct no space is reserved for the three variables declared within union. If we declare a variable x as union any_type x then x can store only one variable of any of the three types at a time. If we write:

```
x.i = 4 ;
printf("%d\n", x.i);
```

the value printed will be the integer 4.

On the other hand if we write:

```
x.c = 'P';
putchar (x.c) ;
```

the value printed will be P.

If we write

```
x.c = 'P';
x.f = 22.4678 ;
```

then x will contain the number 22.46778 and not 'P'.

22.5 COMBINING C PROGRAMS IN DIFFERENT FILES

So far we have assumed that an entire C program is written and stored in one file which is then compiled and executed. If a program is large it would be preferable to store functions

in different files, compile them and test them separately and then combine these files. Such a method is also necessary if a large C program is developed by different programmers in a team and they are to be combined.

Assume that a program consists of part 1, and part 2 and these are stored in files part1.c and part2.c. If they are to be compiled separately using the UNIX operating system then the UNIX command

cc part1.c part2.c

will do the job. Error messages, if any, for part1.c and part2.c will be separately given. If part1.c compiles without any error and part2.c has errors then part2.c file could be corrected and only part2.c may be recompiled using the command:

cc part1.o part2.c

Observe that the object file created for part1.c is being used without any change. It is reiterated that the two parts are compiled *independently*. One of these parts *must* contain a main. Both, of course, should not have main. Independent compilation is possible if the two parts are self-contained. In general, however, one part may refer to functions defined in the other part. Many questions arise. They are:

1. Can the same variable names be used in part1.c and part2.c without confusion?
2. Can the same function names be used in part1.c and part2.c without confusion?
3. Can values stored in a variable name in part1.c be used in part2.c. If yes, how?
4. Can functions defined in part1.c be used in part2.c?

The answer to question 1 is yes provided

* The variable names are defined within function blocks.

* If the variable names are outside function blocks they must have a prefix static. In such a case the scope of the variable name is within the individual files part1.c and part2.c.

The following example clarifies this:

part1.c

```
#include <stdio.h>
static int p; /* static prefix restricts scope of to part1.c p */
main ( )
      { int i; /* i's scope is main */
        for( i = 0; i <= 4; i++)
          p += i ;
        printf ( "%d\n", x(p)) ;
      }
int x (int i)
      { int y ; /* y's scope is the function x */
        y += i; /* i's scope is function x */
        return(y);
      }
```

part2.c

```
static int p = 2;
int y (int j)
   {  int i ; /* i's scope is function y */
      for (i = 0; i <= 6; ++i)
        j *= (i + p); /* j's scope is function y */
      return (j) ;
   }
```

The variable name *p* appearing in both part1.c and part2.c are declared static to restrict their scope.

The answer to question 2 is : No.

The answer to question 3 is yes. If a variable defined in part1.c is to be used by part2.c then prefix *extern* is placed in the declaration of the variable name in part2.c. This is shown in the example below:

part1.c

```
#include <stdio.h>
   int p = 8 ;
main ( )
   { int k, x = 3 ;
     printf("p = %d\n", p) ;
     k= f(x) ;
     printf ("k = %d\n", k);
   }
```

part2.c

```
extern int p ;
int f (int a)
   {  int i, k ;
      i = p - 5 ;
      k = a + 2 * i * p ;
      return (k) ;
   }
```

Observe that *p* is defined (given a value) in part1.c and declared extern and used in part2.c. Definition of a variable should be done where the variable is not declared extern. These two files if compiled, separately, combined and executed will give:

```
p = 8
k = 51
```

The answer to question 4 is yes. In fact functions defined in any of the files can be used in other files. There is no need to define functions as *extern*. When functions are defined they are by default *extern*. If it is required to make a function private to a file and not accessible in other files we declare it as *static*. Table 22.1 summarises these rules.

Table 22.1 Summary of extern and static

Objective	How achieved
To access variable x external to all functions and defined in file i from file j	declare x as extern x in file j
To make a variable x external to all functions and defined in file i *not* accessible to any other file	declare x as static x in file i
To use a function f(x) defined in file i in file j.	No special declaration needed
To make a function float(float x) defined in file i inaccessible to all other files	declare f(x) as: static float f(float x);

22.6 COMMAND LINE ARGUMENTS AND THEIR USE

There are many situations when after a program is written, checked out and compiled one would like to execute it with:

(i) data stored in different data files

(ii) different values of a parameter. For example, different starting value in iterations.

This should be done automatically without recompilation. Information about the names of the data files to be used or values of parameters should be given to the program. This information is given as part of the command used at execution time. For example, if UNIX is the operating system used to run the C program the steps used to compile and run the program contained in a file prog.c are:

```
cc prog.c     (compilation step)
```

The compiled executable code is stored by default in a file named a.out. If we write:

```
a.out     (execution step)
```

the program is executed. If instead of using the default name a.out we want to give another name to the executable file we write

```
cc prog.c –o test     (compilation step)
```

In this case the executable code is stored in test. If we want to execute the program we write:

```
test     (execution step)
```

and the program test is executed.

If instead of just writing test we write:

```
test f1.dat f2.dat     (execution step)
```

the strings f1.dat and f2.dat can be read by the C program for appropriate use by specifying two arguments in the main ( ) function. These two arguments are, by convention, known as argc and argv. The main is written as:

```
main ( int argc, char **argv[ ] )
```

The argument argc gives how many arguments are there in the execution command line. *argv[ ] gives the addresses where the strings read from the command line are stored. If the command line is:

```
test f1.dat f2.dat
```

then

```
argc    = 3;
argv[0] = test;
argv[1] = f1.dat;
argv[2] = f2.dat;
argv[3] = NULL;
```

Example Program 22.2 is a program to read and display a file named sample.dat. The main ( ) function has no arguments in it. If we want Example Program 22.2 to read a file f1.dat first and display it and repeat the program with another file f2.dat we can modify it using command line arguments. This is done in Example Program 22.2. The main function

```
/* Example Program 22.2 */
/* Reading and displaying two files */
#include <stdio.h>
#define ARRAY_SIZE      80
#define NO_FILES        3

main(int argc, char **argv[])
{
  FILE *pointer_to_file[NO_FILES];
  char temp[ARRAY_SIZE];
  int i;
  for (i = 1; i < argc; ++i)
    {
      pointer_to_file[i-1] = fopen(argv[i], "r");
      if (pointer_to_file[i-1] == NULL)
         {
           printf("Cannot open file %s to read\n",
                  argv[i]);
           return(0);
         }
      fscanf(pointer_to_file[i-1], "%s", temp);
      if (temp == NULL)
         {
           printf("Error in reading file %s\n",
                  argv[i]);
           return(0);
         }
      printf("%s", temp);
      fclose(pointer_to_file[i-1]);
    } /* End of for */
} /* End of main */
```

Program 22.2 Use of command line arguments to read and display different files.

has two arguments int argc and char ** argv[ ]. The file names f1.dat and f2.dat are read into argv[1] and argv[2] at execution time from the command line. As we intend to read and display more than one file FILE * pointer_to_file is declared to be an array to accommodate more than one file. In the *for* loop we start with i = 1 as argv[0] stores the name of the program. fopen has argv[i] as the file name. As the array pointer to file starts with 0 we have used pointer_to_file [i – 1]. The rest of the program is self-explanatory.

We give as another example the use of command line arguments to read values. In Example Program 22.3 we have written a program to sum the series.

$$\text{sum } (x^i/i!) \text{ for } i = 1 \text{ to } n$$

```
/* Example Program 22.3 */
/* Illustrating use of values from command line arguments */
#include <stdio.h>

main(int argc, char **argv[])
{
  int i, n;
  float sum, x, term = 1.0;
  sscanf(argv[1], "%f", &x);
  sscanf(argv[2], "%d", &n);
  for (i = 1; i <= n; ++i)
    {
      term *= x/(float)i;
      sum += term;
    }
  printf("x = %f, n = %d, sum = %f\n", x, n, sum);
  /* End of main */

/* If the name of the program is 'series' then the
   UNIX commands to compile and run are
   cc series.c -o series
   series
 */
```

Program 22.3 Use of command line arguments to read parameters for a program.

The recurrence relation is

$$\text{term } (i + 1) = \text{term } (i) * (x/i)$$

$$\text{sum} = \text{sum of term } (i) \text{ for } i = 1 \text{ to } n$$

The value of x and n are read using command line arguments. The core of the program is contained in the for loop:

```
for ( i = 1; i <= n; ++i)
  { term *= x/(float)i ;
    sum += term ;
  }
```

If the name of the program is series the UNIX commands to compile and run it are:

```
cc series.c – o series          (compile and store object program)
series 0.3 10                   (execute)
```

argc = 3, argv[0] = series, argv[1] = 0.3, argv[2] = 10

The main function now has argument argc, argv[0], argv[1] and argv[2] are character strings. They have to be converted to numbers and stored in x and n. This is done by the scanf function. The general form of scanf function is sscanf(string1, format string, pointers to variables). It reads string 1, converts it using format string and stores the values in the specified variable names.

We could have done the same job in a simpler way by writing without command line arguments the statement:

```
scanf("%f%d",&x, &n) ;
```

Example Program 22.3 is cooked up to show the use of integer and real values as command line arguments and the use of scanf function.

22.7 CONDITIONAL COMPILATION

Besides providing means of defining constants and macros the C preprocessor also has a feature known as conditional compilation. Conditional compilation is useful in the following situations:

1. When a program has to be run on different machines which have slightly different characteristics such as word length, memory size etc.

2. When a program is to be run in two modes ; a debugging mode in which a number of printf statements are included to help in debugging and a normal mode in which all the printf statements are to be omitted.

Assume that the maximum size of an array which can be stored in a IBMPC is 5000 whereas it is 50000 in another machine, say VAX 8810. We can then use the preprocessor statements:

```
#ifdef IBMPC
#define ARRAY_SIZE 5000
#else
#define ARRAY_SIZE 50000
#endif
```

If a program test.c is to be compiled for IBMPC we write in UNIX

```
cc –D IBMPC test.c
```

The option – D IBMPC defines IBMPC to the operating system and it takes ARRAY_SIZE as 5000.

If we write

```
cc –D VAX8810 test.c
```

then ARRAY_SIZE is taken as 50000 as IBMPC is not defined and the false branch of the preprocessor statements is taken. If we have a third computer for which ARRAY_SIZE is 10000 we can modify the preprocessor statements to:

```
#ifdef IBMPC
#define ARRAY_SIZE 5000
#elif VAX8810
#define ARRAY_SIZE 50000
#else
#define ARRAY_SIZE 10000
#endif
```

In this case if we write

```
cc –D XYZ test.c
```

the ARRAY_SIZE will be taken as 10000 for XYZ computer.

For debugging a program one normally places printf statements at crucial places in the program to display values of variables. When debugging is over and the program is to be executed these debug lines may be removed by editing the source file and recompiling the code. The preprocessor allows a simpler way. The debug statements may be written as follows:

```
        scanf("%d%f%c\n", &x, &y, &p) ;
#ifdef  DEBUG
        printf( "Echoing input data \n") ;
        printf("x =%d, y = %f, z = %c\n", x, y, z) ;
#endif
```

If the program is compiled under UNIX with DEBUG option as shown below:

```
cc –D DEBUG test.c
```

then all statements enclosed by #ifdef DEBUG and #endif are compiled and thus the printf statements print values needed. If we compile without DEBUG, that is, give the command

```
cc test.c
```

then the statements with #ifdef DEBUG and #endif are not compiled. The executable code will thus become shorter when used for production runs. Another method of including commands for debugging is to write a preprocessor statement

```
#define                 DEBUG 1
and to write
#define                 DEBUG 0
for production runs.
we may also write
#define                 DEBUG 1
for debugging and
#undef                  DEBUG
for production runs.
```

EXERCISES

22.1 Write a conditional expression to:
- (i) Find absolute value of a variable
- (ii) Find min of two variables
- (iii) Find min(a, b, c, d, e, f)
- (iv) Find if a variable is a power of 2.

22.2 Use a comma operator in a *for* loop to add all integers < 25 divisible by 3.

22.3 Use a *for* loop with comma operator to find e^{-x} for x = 0.2.

22.4 Define a macro called half(x). Use it to find (ax + b)/2.

22.5 Define a macro to display an appropriate message on a video screen whenever an integer is to be entered.

22.6 (i) Use command line to find e^{-x} for x = 0.3 till the last term added becomes less than 0.00005.
(ii) If x = 0.15 and the last term added is < 0.0005, how would you change the command line?

Appendix I. Compiling and Running C Programs under UNIX

Suppose we have the following C program to compile and execute

```
#include <stdio.h>
main ( )
   {  int a, b, c ;
      a = 4 ;
      b = 8 ;
      c = a + b ;
     printf("Answer is c = %d \n", c);
   }
```

A file is opened and the code is entered and edited. Let us name this file test_1.c.

The extension .c is used for the file. This is the convention used to indicate that it is a C program and is essential. To compile this program in UNIX environment we type after a system prompt (we assume it is $) appears cc test_1.c and press return key as shown below:

$cc test_1.c

This is a command to compile test_1.c and generate a *loadable object module* which can be executed. This object code is not stored.

If the compilation has proceeded smoothly and there are no errors then a system prompt ($) appears at the beginning of the next line. If we now type a.out as shown below:

$a.out

the computer executes the program and displays the answer:

Answer is c = 12

The UNIX C compiler always puts a *loadable object module* in the file a.out. (The name of the file may be changed if you want.) A load module is directly executable. If we want to just obtain a compiled code and not a loadable executable code then we give the command

$cc –c test_1.c

–c is called a *compile only flag*. In this case an object code is produced and stored in file which is given the name

test_1.o

by the UNIX system.

If three different files of C programs named main.c, t.c and p.c are created and are to be compiled one by one we use the commands

```
$cc –c main.c
$cc –c t.c
$cc –c p.c
```

The object codes created are main.o, t.o and p.o. If there is an error in *t* but main and *p* are correct then we can correct *t* and recompile using the command

```
$cc –c t.c
```

The load module is automatically named a.out which can be executed when the $ prompt appears by typing

```
$a.out
```

If we want to give a different name, say, example.out to the load module we use the option

```
$cc –o example.out main.o t.o p.o
```

The load module is now called example.out and may be executed using the command

```
$example.out
```

Appendix II. Reserved Words in C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Appendix III. Mathematical Functions

The following mathematical functions are defined in <math.h>

In all the functions given below the arguments are of type *double* and result in also of type *double*

Name	Description
sin(x)	sine of *x*
cos(x)	cosine of *x*
tan(x)	tangent of *x*
asin(x)	arc sin(x). *x* in range [$-\pi/2$, $\pi/2$] and x in range [–1, 1]
acos(x)	arc cos(x). *x* in range [0, π] and x in range [–1, 1]
atan(x)	arc tan(x). *x* in range [$-\pi/2$, $\pi/2$]
atan2(y, x)	arc tan (y/x) in range [$-\pi$, π]
sinh(x)	hyperbolic sine of *x*
cosh(x)	hyperbolic cosine of *x*
tanh(x)	hyperbolic tangent of *x*
exp(x)	exponential of *x*
log(x)	natural logarithm of x, (ln x), x > 0 logarithm to base 10 of x. x > 0
pow(x, y)	x raised to power y. Error indication if x = 0 and y <= 0 or if x < 0 and y is an integer
sqrt(x)	$x^{0.5}$, x > or = 0
fabs(x)	absolute value of *x*
fmod(x, y)	floating point reminder of x/y with the same sign of *x*. If y = 0 the result is implementation dependent
ceil(x)	Smallest integer < x. Returned as double
floor(x)	Largest integer not greater than *x*. Returned as double

Appendix IV. String Functions

The following functions are some of the functions defined in <string.h>

In the following description sd, sc, s3 are pointers to null-terminated character strings. A single character is represented by c and an integer by n.

char *strcpy(sd, sc)
Copy string sc to sd including '\0' ; return sd.

char *strncpy (sd, sc, n)
Copy at most n characters of sc to sd; return sd. If sc has less than n characters pad the rest with '\0'.

char *strcat (sd, sc)
Concatenate string sc at the end of string sd; return sd.

char *strncat (sd, sc, n)
Concatenate at most *n* characters of sc to sd and terminate sd with '\0' ; return sd.

char *strchr (sd, c)
Compare characters of sd with character c starting at the head of the string sd. Return pointer to the first occurrence of *c* in sd. If *c* is not present in sd return NULL.

char *strrchr (sd, c)
Compare characters of sd with character c starting at the tail of the string sd. Return pointer to the first match of *c* in sd. If *c* is not present in sd return NULL.

char *strpbrk (sd, sc)
Return pointer to first occurrence in string sd of *any* character of string sc or NULL if no character of sc is present in sd.

char *strstr (sd, sc)
Return pointer to first occurrence of the *whole* string sc in sd. If string sc is not present in sd then return NULL.

unsigned int strlen (sd)
Return length of sd (excluding end of string character '\0').

int strcmp (sd, sc)
Compare string sd and sc. Return a value < 0 if sd is lexicographically less than sc, a value = 0 if sd == sc and > 0 if sd is lexicographically > sc.

int strncmp (sd, sc, n)
Compare at most n characters of sd and sc. Return < 0 if sd < sc, = 0 if sd == sc, > 0 if sd > sc.

Appendix V. Character Class Tests

Character class tests are defined in <ctype.h>. The argument must be representable as a single character. The functions return non-zero (*true*) if the argument *c* satisfies the condition described. Otherwise it returns a zero (*false*).

Name	Description	Example
isdigit(c)	If *c* stores a digit returns *true*	if c = 8 isdigit(c) = 1
islower(c)	If *c* stores a lower case letter returns *true*	if c = 'x' islower(c) = 1 if c = 'A' islower(x) = *false* = 0
isupper(c)	If *c* stores an uppercase letter return *true*	if c = 'P' isupper(c) = 1
isalpha(c)	isalpha(c) is *true* if isupper(c) *or* is lower(c) is *true*	if c = 'X' isalpha(c) = 1 if c = 'y' isalpha(c) = 1 if c = 9 isalpha(c) = 0
isalnum(c)	if isalpha(c) is *true* or if isdigit(c) is *true* returns *true*	
isspace(c)	if *c* is space, form feed, newline, carriage return, tab or vertical tab this returns *true*	if c = '\n' isspace(c) = 1 if c = 'z' isspace(c) = 0
ispunct(c)	if *c* is a printable character other than space, or digit this returns *true*	if c = ',' ispunct(c) = 1
iscntrl(c)	if *c* is a control character this returns *true*	if c = 0 iscntrl(c) = 1
isprint(c)	if *c* is a printable character including space it returns *true*	
isgraph(c)	if *c* is a printing character except space it returns *true*	
isxdigit(c)	if *c* is a hexadecimal digit this returns *true*	if c = A isxdigit(c) = 1
Functions to convert case of letters		
tolower(c)	returns lowercase letter if *c* is uppercase else there is no change in *c*	
toupper(c)	returns uppercase letter if *c* is lowercase else there is no change in *c*	

Appendix VI. File Manipulation Functions

Some of the common file manipulation functions available in the C library <stdio.h> are given below. The definitions of EOF, NULL, stdin, stdout, stderr and FILE are included in <stdio.h>. In the following description file_name, access_mode and format are pointers to strings (terminated by NULL), *buffer* is a pointer to a character array, file_pointer is a pointer to a FILE structure, *n* and *size* are positive integers and *c* is a character.

FILE *fopen *(file_name, access_mode)*

Opens the specified file with the indicated access mode. Valid modes are "r" for reading, "w" for writing, "a" for appending to the end of an existing file, "r+" for read/write access starting at the beginning of an existing file, "w+" for read/write access (and the previous contents of the file, if it exists, are lost), and "a+" for read/write access with all writes going to the end of the file. If the file to be opened does not exist, then it will be created if the *access mode* is write ("w", "w+"), or append ("a", "a+"). If a file is opened in append mode ("a" or "a+"), then it is not possible to overwrite existing data in the file. If the fopen call is successful, then a FILE pointer will be returned to be used to identify the file in subsequent I/O operations; otherwise, the value NULL is returned.

FILE *freopen *(file_name, access_mode, file_pointer)*

Closes the file associated with *file_pointer*, and opens the file *file_name* with the specified *access_mode* (see the fopen function). The file that is opened is subsequently associated with *file_pointer*. If the freopen call is successful, then *file_pointer* will be returned; otherwise, the value NULL will be returned. The freopen function is frequently used to reassign stdin, stdout or stderr in the program. For example, the call

freopen ("output_data", "w", stdout)

will have the effect of reassigning stdout to the file output_data, which will be opened in write mode. Subsequent I/O operations performed with stdout will be performed with the file output_data, as if stdout had been redirected to this file when the program was executed.

int fscanf *(file_pointer, format, arg1, arg2, ..., argn)*

Data items are read from the file identified by *file_pointer*, according to the format specified by the character string *format*. The values that are read are stored into the arguments specified after *format*, each of which must be a pointer. The *format* characters that are allowed in *format* are the same as those for the scanf function. The fscanf function returns the number of items successfully read and assigned or the value EOF if end of file is reached before the first item is read.

int fseek *(file_pointer, offset, mode)*

Positions the indicated file to a point that is *offset* (a long integer) bytes from the beginning of the file, from the current position in the file, or from the end of the file, depending upon the value of *mode* (an integer). If *mode* equals 0, then positioning is relative to the beginning of the file. If *mode* equals 1, then positioning is relative to the current position in the file. If *mode* equals 2, then positioning is relative to the end of the file. If the fseek call is successful, then a non zero value is returned; otherwise, zero is returned.

long ftell *(file_pointer)*

Returns the relative offset in bytes of the current position in the file identified by *file_pointer*, or –1 on error.

int fwrite *(buffer, size, n, file_pointer)*

Writes *n* items of data from *buffer* into the specified file. Each item of data is *size* bytes in length. Returns the number of items successfully written.

int getc *(file_pointer)*

Reads and returns the next character from the indicated file. The value EOF is returned if an error occurs or if the end of the file is reached.

int putc *(c, file_pointer)*

Writes the character *c* to the indicated file. On success, *c* is returned; otherwise, EOF is returned.

FILE *tmpfile( )

Creates and opens a temporary file in write update mode, returning a FILE pointer identifying the file, or NULL if an error occurs. The temporary file is automatically removed when the program terminates. (Functions called **tmpnam** and **tempnam** are also available for creating temporary file names.)

int ungetc *(c, file_pointer)*

Effectively "puts back" a character to the indicated file. The character is not actually written to the file but is placed in a buffer associated with the file. The next call to getc will return this character. The ungetc function can only be called to "put back" one character to a file at a time; that is, a read operation must be performed on the file before another call to ungetc can be made. The function returns *c* if the character is successfully "put back" or the value EOF otherwise.

int fprintf *(file_pointer, format, arg1, arg2, ..., argn)*

Writes the specified arguments to the file identified by *file_pointer*, according to the format specified by the character string *format*. Format characters are the same as for the printf function (see Chapter 16). The number of characters written is returned.

int fputc *(c, file_pointer)*

Writes the character *c* to the file identified by *file_pointer*, returning *c* if the write is successful, and the value EOF otherwise.

int fputs *(buffer, file_pointer)*

Writes the characters in the array pointed to by *buffer* to the indicated file until the terminating null character in *buffer* is reached. A newline character is not automatically written to the file by this function. On failure, the value EOF is returned.

int fread *(buffer, size, n, file_pointer)*

Reads *n* items of data from the identified file into *buffer*. Each item of data is *size* bytes in length. For example, the call

fread (text, sizeof (char), 80, in_file)

reads 80 characters from the file identified by in_file, and stores them into the array pointed to by text. The function returns the number of characters that are successfully read.

int fflush *(file_pointer)*

Flushes (writes) any data from internal buffers to the indicated file, returning zero on success and the value EOF if an error occurs.

int fgetc *(file_pointer)*

Returns the next character from the file identified by *file_pointer*, or the value EOF if an end of file condition occurs (remember that this function returns an int).

char *fgets *(buffer, n, file_pointer)*

Reads characters from the indicated file, until either n – 1 character are read or until a newline character is read, whichever occurs first. Characters that are read are stored into the character array pointed to by *buffer*. If a newline character is read, then it will be stored in the array. If an end of file is reached or an error occurs, then the value NULL is returned; otherwise, buffer is returned.

int fclose *(file_pointer)*

Closes the file identified by *file_pointer*, and returns zero if the close is successful, EOF if an error occurs.

int feof *(file_pointer)*

Returns nonzero if the identified file has reached the end of the file and zero otherwise.

int ferror *(file_pointer)*

Checks for an error condition on the indicated file and returns zero if an error exists, and nonzero otherwise. (There is a related function **clearerr**, which can be used to reset an error condition on a file.)

void rewind *(file_pointer)*

Resets the indicated file back to the beginning of the file.

The functions **sprintf** and **sscanf** are provided for performing data conversion in memory. These functions are analogous to the **fprintf** and **fscanf** functions except a character string replaces the FILE pointer as the first argument.

int sprintf *(buffer, format, arg1, arg2, ..., argn)*

The specified arguments are converted according to the format specified by the character string *format* and are placed into the character array pointed to by *buffer*. The number of characters placed into *buffer* is returned, excluding the terminating null.

int sscanf *(buffer, format, arg1, arg2, ..., argn)*

The values as specified by the character string *format* are "read" from buffer and stored into the corresponding pointer arguments that follow *format*. The number of items successfully assigned is returned by this function.

Appendix VII. Utility Functions

The library <stdlib.h> has a number of functions for number conversion, storage allocation, sorting and a number of useful tasks. We explain some of the more useful ones below.

double atof(*string*)
Converts character *string* to a double precision number.
Example: double atof (char a[ ]);

int atoi *(string)*
Converts character *(string)* to an integer value
Example: int atoi (char a[ ])

long atol *(string)*
Converts character *string* to long integer value

int rand (void)
Returns a pseudo random integer in the range 0 to 32767

void srand (unsigned int seed)
Uses *seed* to produce a new sequence of pseudo random numbers. Initial seed is 1.

void calloc (unsigned int *n*, unsigned int *s*)

Calloc returns space for an array of *n* objects each of size *s* or NULL if the request cannot be satisfied. The space is initialized to 0 bytes.

void *malloc(unsigned int *s*)
malloc returns a pointer to an object of size *s* or NULL if the request cannot be satisfied. The space is uninitialized.

void *realloc (void *p, unsigned int *s*)
realloc changes the size of objects pointed to by *p* to *s*. The contents will be unchanged upto the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized, realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

void free (void *p)
free disallocates the space pointed to by *p*. It does nothing if *p* is NULL. *p* must be a pointer previously allocated by malloc, calloc or realloc.

void abort(void)
abort causes the program to terminate abnormally.

void exit (int status)
exit causes normal program termination.
status = 0 for successful termination.

int abs(int *n*)
abs returns the absolute value of *n*.

long abs (long *n*)
abs returns the absolute value of long argument
div_t div (int num, int denom)

```
struct div_t
   {  int quot ;
      int rem ;
   };
```

div computes quotient and remainder of num/denom. The results are stored in the struct div_t.

ldiv_t ldiv(long num, long denom)

same as div except that it uses long integer arguments.

Appendix VIII. Summary of C Language

Variables

Declarations:

Syntax: <variable type><variable name1>, <variable name2>, ...

```
Examples:   int     num, count ;
            float   velocity, mass, distance
            char    initial, grade;
            int     a[10], b[10][5];
            float   mat [5][10];
            char    str_2 [20];
```

Initialization

Syntax: <variable type><variable name> = <value>;

```
Examples:   float   x = 20.5;
            int     count[3] = {5, 6, 7, 2};
            char    initial = 'Q';
```

Assignments

Syntax: <name> = <value>;

```
Example:    y = 25.5;
            u = 'T';
```

Constants

Syntax: #define <constant name><value>

Example: #define PI 3.141592

Type definition

Syntax: typedef <variable type><symbolic name>;

Example: typedef int Boolean;

Comments

Syntax: /* <body of comment> */

Example: /* This is a comment */

Structures

Declarations

Syntax: struct<identifier>
```
        { <type><variable name>;
          <type><variable name>;
          - - - -
          - - - -
        };
    struct<identifier><variable name>, <variable name>;
```

Example: struct item
```
        { int item_code;
          char item_name [30];
          float price;
        };
     struct item soap, cycle;
```

Assignment

Syntax : <variable name> . <member> = <value>;

Example: soap.price = 8.50;
cycle.item_name = "HERO";

Unions

Declaration

Syntax: Union <identifier>
```
        { <type><member name>;
          <type><member name>;
          - - - -
          - - - -
        } <variable name>;
```

Example: Union price
```
        { int p;
          float q;
        } cost;
```

Assignment

Syntax: <identifier>.<member name>=<value>;

Example: price.p = 25;
price.q = 35.85;

Operators Symbol	Operation	Associates from	Precedence
++	increment	left to right	Highest
--	decrement		(Evaluated first)
~	one's complement		
!	negation		
&	address		
*	de-reference		
(type)	cast		
-	unary minus		
sizeof	size in bytes		
*	multiply	right to left	
/	divide		
%	remainder		
+	add	left to right	
-	subtract		
<<	shift left		
>>	shift right		
<	less than		
<=	less than or equal to		
>	greater than		
>=	greater than or equal to		
==	equal to		
!=	not equal to		
&	bitwise and		
^	bitwise exclusive or		
|	bitwise inclusive or		
&&	logical and		
||	logical or		
?:	conditional	right to left	
= %= += -= *= /= >>= <<= &= ^= |=	Assignment		
	comma	left to right	(Evaluated last) lowest

Remarks: In the above table, operators between horizontal lines have same precedence. Symbols from ++ to sizeof are unary operators.

Input and Output

printf formats

```
printf("<literal string>");
printf("<literal string >\n"); (new line character \n)
printf("<conversion specifications>", <variable names>);
```

Examples

```
printf("Answers follow");
printf("Table heading \n");
printf("%d%f%e\n", x, y, z);
```

printf conversion specifications

%d	%u	%o	%x	%f	%e	%c	%s
decimal	unsigned decimal	octal	hex	floating	exponent form	char	string

scanf format

```
scanf("<conversion specs>",& <variable>, & <variable>);
```

Examples:

```
scanf("%d %f %e", &numb,&sal,&avge);
scanf("%s %c %d" , stud, &status[2], age);
```

(stud is the name of an array storing a character string)

scanf conversion characters

%d	%o	%x	%h	%c	%s	%f	%e
decimal integer	octal integer	hex integer	short integer	character	string	real decimal	real in exponent form

Primitive input/output

c = getchar ( ); get character from standard input
putchar (c); puts a character in the standard output unit

string input/output (in library <stdio.h>)

gets(<array name>);
Reads characters from standard input and stores in array name and quits when end of line is reached. New line character is *not* stored in the array.
puts(<array name>) ;

Writes characters from array name on standard output followed by a new line character.

Example

```
char name[20]
gets (name);
(Reads characters from terminal till end of line or carriage return and stores in array
name)
char record[80];
puts (record);
(writes 80 characters of record or less until end of line is sensed on standard output
Appends new line character.)
```

Control structures

if statement

```
if ( <condition expression> )
   { <compound statement> ;
   }
```

Instead of a compound statement a single statement is also allowed.

Examples

```
if( x < 0 )
   { ++ count_neg ;
     x += 2 ;
   }

if( i > 0 )
   i = –i ;
```

if - else statement

```
if (<condition expression> )
   {<compound statement> ;
   }
else
   {<compound statement> ;
   }
```

Example

```
if(x * x > 500)
   { printf(" The square is greater than 500 \n") ;
     done = 1 ;
   }
else
   { printf(" The square <= 500 \n") ;
     done = 0 ;
   }
```

```
if(x < 0 )
   printf(" x < 0 \n") ;
else
   printf(" x > = 0 \n") ;
```

Switch statement

```
switch ( <expression> )
   {  case <value 1>: <compound statement 1> ; break ;
      case <value 2>: <compound statement 2> ; break ;
      - - - - -
      - - - - -
      default: <compound statement for default> ; break ;
   }
```

Example

```
scanf("%d", &inp_number)
switch(inp_number)
   {Case 1 : printf("inp_number is one \n") ;
             break ;
    Case 2 : printf("inp_number is two \n") ;
             break ;
    Case 3 : printf("inp_number is three \n") ;
             break ;
    default : printf("inp_number <1 or >3 -
             out of range \n") ;
             break ;
   }
```

Loop structures

for loop

```
for (<expression 1> ; <expression 2> ; <expression 3>)
   { <compound statement> ;
   }
```

Instead of a compound statement a single statement is also allowed.

Examples

```
for ( i = 1 ; i <= 10 ; ++i )
    { sum += a[i] ;
      b[i]=k[i]+ c[i] ;
    }

for (k = p ; k <= t ; k += q)
        s += a ;

for ( i = 1, sum = 5 ; (i <= 10) || (sum > 200)) ; i += 2 )
        sum += a[i] ;
```

while loop

```
while ( <condition expression> )
      { <compound statement> ;
      }
```

Instead of a compound statement a single statement is also allowed.

Examples

```
while (scanf (%d, &age) != EOF)
     {sum += age ;
      ++ count ;
     }

while ( i < 10 )
     s += a[i++] ;
```

do while loop

```
do
      { <compound statement> ;
      }
while ( <condition expression> ) ;
```

Instead of a compound statement a single statement is also allowed.

Example

```
do
    {printf("y =%d \n" , y) ;
       y -= 3 ;
    }
while (y > 0) ;

do
    {printf("More input? (Type 1 for yes, 0 for No)") ;
     scanf("%d", &answer) ;
     printf("\n") ;
    }
while (answer !=1 && answer != 0) ;

do
   scanf("%d", &p) ;
while(p > 0) ;
```

Function Definitions

Syntax

```
<type returned><function name>(<type><parameter>,
                                <type><parameter )
      {<declaration of local variables> ;
         body of function> ;
      }
```

Example

```
int add ( int a, int b)
   {  int c ;
    c = a + b ;
    return(c) ;
   }
```

Pointers

Declaration of pointer variables

Syntax : <type> *<variable name>

Example : char * p ;
  struct student *stud_list ;

Assignments

```
int k ;
int q ;
int *s ;
s = &q ; /* allowed */
k = *s ; /* same as k = q */
```

Structure of a C Program

```
#include <stdio.h> /* i/o library */
#include <math.h> /* math library – optional */
#include <string.h>/* string functions – optional */
#include <ctype.h> /* character class tests – optional */
#include <stdlib.h>/* utility functions – optional */
#include - - - /* other library files */
#define - - - /* define constants */
/* Declare global variables */
<variable type><list of variables>;
/* Function prototypes */
<type returned><function name> (<parameter type><parameter list>);
/* main function */
main (optional argc and argv arguments)
{ <declaration of local variables, arrays, structures etc>
      body of code
} /* End of main */
```

```
<type returned><function name>( <parameter type><parameter list> )
{ <declaration of local variables>
        body of functions
        return ( )
} /* End of function */
Other functions if any
```

References

1. Brown, D.L., *From Pascal to C*, Narosa Publishing House, New Delhi, 1985.
2. Esakov, J. and Weiss, T., *Data Structures—An Advanced Approach Using C*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1989.
3. Hancock, L. and Krieger, M., *The C Primer*, McGraw-Hill, New York, 1987.
4. Holzner, S., *C Programming*, Prentice-Hall of India Pvt. Ltd., New Delhi, 1992.
5. Jones, R. and Stewart, I., *The Art of C Programming*, Narosa Publishing House, New Delhi, 1988.
6. Jhonsonbaugh, R. and Kalin, M., *Applications Programming in C*, Macmillan Publishing Company, New York, 1990.
7. Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall of India Pvt. Ltd., New Delhi, 1989.
8. Kochan, S.G., *Programming in C*, CBS Publishers, Delhi, 1987.
9. Rajaraman, V., *Computer Programming in Pascal*, Prentice-Hall of India Pvt. Ltd., New Delhi, 1991.
10. Tizzard, K., *C for Professional Programmers*, Affiliated East-West Press Pvt. Ltd., New Delhi, 1986.

Index

Computer Programming in C

by

V. RAJARAMAN

This book introduces computer programming to a beginner using the programming language C. The version of C used is the one standardised by the American National Standards Institute (called ANSI C). C has rapidly gained users due to its efficiency, rich data structure, variety of operators and affinity to UNIX operating system. C is a difficult language to learn if it is not methodically approached. Our attempt has been to introduce the basic aspects of C to enable the student to quickly start writing C programs and postpone more difficult features of C to later chapters. The methodology of presentation closely follows the one used by the author in his popular book on PASCAL programming. Those who know PASCAL will find it very easy to learn C using this book.

Distinctive Features

- A self-contained introduction to programming in C for beginners.
- All important programming language features illustrated with over 100 example programs.
- Good style in programming emphasised.
- Eminently suitable for self-study.

V. RAJARAMAN, Ph.D. (Wisconsin), F.A.Sc., FNA, FNAE, FCSI, is Honorary Professor, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore. Earlier (1963–1982) he taught at the Indian Institute of Technology Kanpur. A pioneer in computer science education and research in India, Prof. Rajaraman was awarded the Shanti Swarup Bhatnagar Award, the Homi Bhabha Award for Research in Applied Sciences, the U.P. Government National Award for Excellence in Teaching and Research, and the Syed Hussain Zaheer Medal by the Indian National Science Academy. A recipient of Padma Bhushan, he has published many research papers in reputed national and international journals besides authoring several established books.

ISBN:978-81-203-0859-6

9 788120 308596

Rs. 195.00

www.phindia.com