```
        {
              printf("Error in opening file\n");
              exit(1);
        }
        fputc('X', fp); /* attempting to write - error */
        if (ferror(fp))
        {
              printf("Error: File is read only!\n");
              fclose(fp);
              exit(1);
        }
}
```

Sample Run

```
Error: File is read only!
```

It is very interesting to note here that the file "t.txt" is opened in "r" mode. Then using *fputc*, the program is trying to write a character into the file which is illegal. Supposing if you do not have *ferror* function in the program then the system does not report any errors (no display of error message). In other words, the system can't catch this error for you. By having the *ferror* in the `if` statement, error message is displayed as shown in the sample run.

Instead of using *printf* to print the error message you can use *perror*. See the Program 1.49 and the sample output.

Program 1.49
Catching errors through ferror and perror

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main()
{
     FILE *fp;
     int c;

     clrscr();
     fp = fopen("t.txt","r");
     if (fp == NULL)
     {
          printf("Error in opening file\n");
          exit(1);
     }
     fputc('X', fp); /* attempting to write - error */
     if (ferror(fp))
```

```
        {
                perrror("t.txt");
                fclose(fp);
                exit(1);
        }
}
```

Sample Run

```
t.txt: Error 0
```

The meaning of Error number 0 shown in the output is access mode violation error. Even when you open a file in write mode and try to read from it, a similar error number is displayed.

1.14 Command line arguments

Sending arguments to a *main()* program at the command prompt of an operating system like DOS or UNIX is called as command line arguments. For example, to display the contents of a file in the DOS prompt you issue a command TYPE t.txt and the file t.txt is the argument to the command TYPE. We can design C programs to behave in a similar manner using command line argument feature.

In C *main()* is a function like any other function. This statement is partially correct, because there are few differences with respect to a user defined or built-in function. The *main()* function is a must for every C program. There is no need to explicitly call this unlike other functions. But then how is it invoked? It is invoked by the system when you specify the name of the EXE file at the command prompt.

Like other functions, it can also accept arguments and return integer value. However, it can accept only two arguments:

(1) **argc** - refers to argument count. Total number of arguments passed from command prompt.

(2) **argv[]** - refers to argument vector. Pointer to an array of character strings that contains argument list.

Example

D:\Books\DS-Book\C-Programs>MERGE F1 F2

Here, *argc* will be 3 and *argv[0]* = "F1" and *argv[1]* = "F2". The *argc* counts the command name and the arguments totaling 3.

All programs shown in this section should be executed by following the below given steps:

- Compile the program (press F9).
- Link all modules to get the EXE file (Compile->Build All).
- Come to the DOS prompt (File->Dos Shell).
- Type the file name along with the arguments.

Program 1.50
Demo to show the command line arguments

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
     int i;
     printf("Number of arguments passed= %d\n", argc);

     printf("The names of arguments are:\n");
     for (i=0; i<argc; i++)
          printf("%s\n", argv[i]);
     return(0);
}
```

Sample Run

```
D:\DS-Book>p150 f1 f2 f3
Number of arguments passed= 4
The names of arguments are:
D:\DS-BOOK\P150.EXE
f1
f2
f3
```

Actually there are three arguments passed and including the file name it is 4. Let us present a much more meaningful example in Program 1.51. The aim of this program is to merge file 2 to file 1 given in the command prompt. This is one of the advantages of command line arguments. That is, commands that are not provided by DOS can be designed using C programs and made available to the users. This is what exactly we have shown in Program 1.51. The problem statement is as follows:

Design a C program to merge two files, i.e. put the contents of file 2 at the end of file 1. Let us call this command as MERGE.

C:\ MERGE <FILE-1> <FILE-2>

Program 1.51
Demo to show the command line arguments

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
     FILE *infile1, *infile2;
     int c;

     if (argc < 3)
     {
       printf("Insufficient number of arguments\n");
       printf("Usage: MERGE <filename-1> <filename-2>\n");
       exit(1);
     }
     infile1 = fopen(argv[1],"a"); /* append and read */
     if (infile1 == NULL)
     {
          printf("Error in opening file\n");
          exit(1);
     }
     infile2 = fopen(argv[2],"r");
     if (infile2 == NULL)
     {
          printf("Error in opening file\n");
          exit(1);
     }
     while ((c = getc(infile2)) != EOF)
          putc(c,infile1);
     return(0);
}
```

Sample Run-1

```
D:\DS-Book>MERGE
Insufficient number of arguments
Usage: MERGE <filename-1> <filename-2>
```

Sample Run-2

```
D:\DS-Book>MERGE F1.txt F2.txt
D:\DS-Book>TYPE F1.txt
     File-1 contents

     File-2 Contents
```

The sample-1 shows a situation where file names are not given in the command prompt. The second run is correct and the second file is appended to the fist file.

Program 1.52
Split a given file into two files

```
#include <stdio.h>

#include <stdlib.h>
int main(int argc, char *argv[])
{
     FILE *fp, *f1, *f2;
     int i, c; long int mid;

     if (argc < 2)
     {
          printf("Insufficient number of arguments\n");
          printf("Usage: SPLIT <filename>\n");
          exit(1);
     }

     fp = fopen(argv[1],"r"); /* append and read */
     if (fp == NULL)
     {
          printf("Error in opening file\n");
          exit(1);
     }

     f1 = fopen("temp1","w");
     f2 = fopen("temp2", "w");

     fseek(fp, 0L, SEEK_END);
     i = 0;
     /* find mid position of the file */

     mid = ftell(fp) / 2;
     mid = mid - 3; /* avoid file descriptor data */
     rewind(fp);
     while (i < mid) /* copy first half to file-1 */
     {
          c = fgetc(fp);
          putc(c,f1);
          i++;
     }
```

```
    rewind(fp);
    fseek(fp, mid, SEEK_SET);
    /* copy from mid to file-2 */
    while ((c = fgetc(fp)) != EOF)
         putc(c,f2);
    return(0);
}
```

1.15 SUMMARY

The Chapter introduced the fundamental concepts, mainly the syntax, of C language. Every C program should have a *main()* program and enclosed with the braces { }. C is a function oriented language which can be used for both low and high level programming. The language offers:

- *int, float, char, long, double* as the main data types for constants and variables.
- Variables can be declared and initialized.
- User defined types through *typedef.*
- I/O functions to read and print – *getchar(), putchar(), scanf(), printf(),* etc.
- Rich arithmetic and logical operator set:- +, -, *, /, %, ++, --, +=, ? :, >, >>, <<, etc. makes expression evaluation more efficient.
- For effective program writing we have decision making statements like if, if-else, switch, continue, break, and looping structures like while, do-while and for statements.
- C arrays have fixed lower bound 0 and the last element's address is (n-1), where n is the size of the array.
- Arrays get their memory at compilation time and we can declare single-dimension, two-dimension, etc.
- The main strength of C language is the flexibility of declaring functions.
- Provides a rich set of library functions for keyboard reading and writing (defined in *stdio.h*), mathematical functions (defined in *math.h*), console oriented functions (defined in *conio.h*, string functions (defined in *string.h*), etc.
- Users can also define their own functions and add it to the C library. This saves money, time, etc. for any future development of programs.
- Two types of parameter passing methods are allowed (1) Value parameter or pass by value (2) Reference parameter (or pass by address).
- *Void* functions do not return any value or address to the calling routine.
- Functions can also be designed to return values from called to calling function using return statement.
- Structures and Unions can bring different data types under a single name.
- For faster memory accessing and address arithmetic, pointers are useful.

- To obtain memory dynamically C offers *malloc(), calloc()*, and *realloc()* functions. The difference between *malloc* and *calloc* is that the former uninitializes the allocated memory whereas the later initializes to 0. The *calloc* allocates n block of memory and is supplied with two arguments. But *malloc* gets just one argument.
- Data may be available in a file for certain applications. Also the result after processing may have to be dumped in a file for later reference. C language provides a rich set of file management functions categorized as control functions and accessing functions.
- *fopen()* is a basic function that is used to open a file in one of several modes.
- The allowed modes in text format are: *r, w, a, r+, w+, a+*. Similarly, you can operate files in binary mode also by adding the letter' *b*' with the existing modes (*rb, wb,* etc.).
- To access the text data in a disk file, *fgetc(), fputc(), getc(), putc(),* etc. functions are useful.
- For integer/float/double and other types of data handling in files, *fscanf(), fprintf()* functions are useful.
- *ferror()* and *perror()* functions are used to catch errors during file operations.
- To pass parameters to *main()*, arguments must be specified in the parameter portion of *main()*. Only two arguments are allowed and are called as *argc* and *argv[]*. It is useful to simulate certain DOS commands.

1.16 EXERCISES

1.1 What are the primitive data types available in C and give example for each type. What do you understand by unsigned char type?

1.2 By giving examples bring out the advantages of the following:
(1) *void* type (2) enumerated type (3) user-defined type
(4) *escape* sequences (5) type-casting (6) continue and break

1.3 Show the output of the following assuming, i = 48, j = 16, k = 0;
(a) k = i & j; (b) k = i | j; k = k >> 4; (c) k = i << 5;

1.4 What are *Lvalue* and *Rvalue*? Give examples?

1.5 Write a C program to display even and odd numbers separately in an array.

1.6 Develop C functions to multiply two given matrices A and B. Check for the compatibility, before the multiplication could be done.

1.7 Can you design a function to return the entire array to the calling function? If so, how? If not, explain why it can not be.

1.8 Write C function to reverse the contents of an array with out using any intermediate array.

1.9 Design a structure to store the real and imaginary part of a Complex number. Write functions to add, subtract, multiply, and divide two complex numbers based upon the Complex number structure definition.

1.10 What are the main advantages of pointers in C?

1.11 Write a function using pointers to return the length of a string.

1.12 Discuss the need for a *void* pointer.

1.13 What do you understand by dynamic memory allocation? Explain with examples *malloc(), realloc(),* and *calloc().*

1.14 What are the differences between *malloc* and *calloc*?

1.15 Write a C program to simulate a dynamic array.

1.16 Write C programs to demonstrate the working of "*rb*" and "*wb*" mode of file operations.

1.17 Explain with an example *fopen()* function.

1.18 What are *fread* and *fwrite* and how are they different from *fgetc* and *fputc*?

1.19 Write a program to store names and telephone numbers of certain consumers in a file. Design a function *Find()* to find the phone number of one or more persons given a name.

1.20 Discuss the importance of *ftell* and *ferror.*

1.21 What are command line arguments and how do you use them?

1.22 Write C programs to do the following (use command line arguments):
(a) Move a file from one directory to another.
(b) Delete a file in the disk (*Hint*: use *Unlink*).
(c) Convert uppercase to lower case of a given file.