1. FILES AND STREAMS

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.

But many applications require that information be written to or read from an auxiliary memory device. Such information is stored on memory device in the form of a data file. Moreover, storage of data in variables and arrays is temporary – such data is lost when a program terminates.

Files are used for permanent retention of data. Computers store files on secondary storage devices, especially disk storage devices. Thus, the data files allow the users to store information permanently, and to access and alter that information whenever necessary.

The C language supports two different types of data files, called *stream-oriented* (or *standard*) data files, and *system-oriented* (or *low-level*) data files. Stream-oriented files are generally easier to work with and are therefore more commonly used. In C, extensive set of library functions is available for creating and processing both types of these data files.

A *stream* is basically a file abstraction that is designed to work with files on a wide variety of devices including terminals, disk drivers, and tape drivers. Even though each device is very different, the buffered file system transforms each into a logical device called a *stream* to provide a consistent interface to the programmer independent of the actual device being accessed.

C views each file simply as a sequential stream of bytes irrespective of the actual device on which is physically stored and all the streams behave similarly. Because streams are largely device independent, the same function that can write a disk file can also write to another type of device, such as a console.

Stream-oriented data files can be subdivided into two categories - *text files* and *unformatted* (*binary*) data files. The steam-oriented text files consist of consecutive characters. These characters can be interpreted as individual data items, or as components of string or numbers. The manner in which these characters are interpreted is determined either by particular library function used to transfer the information, or by format specification within the library functions, as in the printf and scanf functions.

The stream-oriented unformatted data files organize data into blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures. A separate set of library functions is available for processing stream-oriented unformatted data files. These library

functions provide single instructions that can transfer entire arrays or structures to or from data files.

System-oriented data files are closely related to the computers operating system. They are somewhat more complicated to work with, though their use may be more efficient for certain kinds of applications. A separate set of library functions is available for system-oriented data files.

2. OPENING & CLOSING FILE STREAMS

In C, files may be anything from a disk file to a terminal or printer. One needs to associate a stream with a specific file by performing an *open* operation. Once file is open, information can be exchanged between the file and the program. The programmer disassociates a file from a specific stream with *close* operation. If the program closes a file opened for output, the contents, if any, of its associated stream are written to the external device.

When working with a stream-oriented data file, the first step is to establish a buffer area, where information is temporarily stored while being transferred between the computer's memory and the data file. This buffer area allows information to be read from or written to the data files more rapidly than would otherwise be possible. The buffer area is established by writing

```
FILE *ptvar;
```

where *FILE* (Upper-case letters required) is a special structure type that establishes the buffer area, and ptvar is a pointer variable that indicates the beginning of the buffer area. The structure type FILE is defined within the *stdio.h* file. The pointer ptvar is often referred to as stream pointer or simply a stream.

A data file must be opened before it can be created or processed. This associates the file name with the buffer area (i.e., with the steam). It also specifies how data file will be utilized, i.e., as read-only file, a write-only file, or a read/write file. The library function *fopen* is used to open a file. This function is typically written as

```
ptvar = fopen(file-name, file-mode);
```

where the *file-name* and *file-mode* are strings that represent the name of the data file and the manner in which the data file is utilized. The name chosen for the data file must be consistent with the rules for naming files.

The types of operations that will be allowed on the file are defined by the value of *file-mode*. The legal values of modes are shown in the following table.

Mode	Meaning
r	Open an existing file for reading only.

Mode	Meaning
w	Open a new file for writing only.
a	Open an existing file for appending (i.e., for adding new information at the end of the file). If a file with the specified file-name currently does not exist, a new file will be created.
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+	Open an existing file for both reading and writing.
w+	Open a new file for both reading and writing. If a file with the specified file-name currently exists, it will be destroyed and a new file created in its place.
a+	Open an existing file for both reading and appending. If a file with the specified file-name currently does not exist, a new file will be created.
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append or create a binary file for read/write

Note that strings like "r+b" may also be represented as "rb+".

If *fopen()* is successful in opening the specified file, a *FILE* pointer is returned. If the file cannot be opened, a *NULL* pointer is returned.

For example, the following code uses *fopen()* to open a file named *TEST* for output.

```
FILE *fptr;
fptr = fopen("TEST", "w");
```

Although the preceding code is technically correct, the correct method of opening file is illustrated by the following code fragment.

```
FILE *fptr;
if (( fptr=open("TEST", "w") ) == NULL)
{   printf("Cannot Open File\n");
    exit(1);
}
```

This method detects any error in opening a file, such as write-protected or a full disk before attempting to write to it. If no file by that name exists, one will be created. Opening a file for read operations require that the file exists.

The *fopen* function returns a pointer to the beginning of the buffer area associated with the file. A *NULL* value is returned if the file cannot be found.

Finally, a data file must be closed at the end of the program. A file can be closed by using the library function *fclose*. The syntax is simply

fclose(ptvar);

The *fclose* function closes the file associated with stream and flushes its buffer (i.e., it writes any data still remaining in the disk buffer to the file). After a call to *fclose*(), stream is no longer connected with the file, and any automatically allocated buffers are deallocated.

If *fclose*() is successful, zero is returned; otherwise *EOF* is returned.

For example, the following code opens and closes a file:

```
#include<stdio.h>
#include<stdlib.h>
main()
{ FILE *fptr;
  if ((fptr = fopen("TEST","w"))==NULL)
  { printf("Cannot Open File.\n");
    exit(1);
  }
  if(fclose(fptr)) printf("File Close Error.\n");
}
```

3. FILE HANDLING FUNCTIONS

The C file system is composed of several built-in functions that are specifically designed for the file operations, as listed below.

Name	Function
fopen( )	Opens a file
fclose( )	Closes a file
putc( )	Writes a character to a file
fputc( )	Same as *putc( )*
getc( )	Reads a character from a file
fgetc( )	Same as *getc( )*
fgets( )	Reads a string from a file
fputs( )	Writes a string to a file
fseek( )	Seeks to a specified byte in a file
ftell( )	Returns the current file position
fprintf( )	Is to a file what *printf( )* is to the console
fscanf( )	Is to a file what *scanf( )* is to the console
feof( )	Returns true if end-of-file is reached
ferror( )	Returns true if an error has occurred
rewind( )	Resets the file position indicator to the beginning of the file
remove( )	Erases a file
fflush( )	Flushes a file

3.1. I/O OPERATIONS ON FILES

Once the file is opened, the file I/O operations are accomplished using any of the following standard I/O routines like *putc*, *fputc*, *getc*, *fgetc*, *fgets*, *fputs*, *fprintf*, *fscanf*, etc.

fputc

Declaration: *int fputc(int ch, FILE *stream);*

The *fputc*() function writes the character *ch* to the specified stream at the current file position and then advance the file position indicator. Even though the *ch* is declared to be an *int*, it is converted by *fputc*() into an *unsigned char*.

The value returned by the *fputc*() is the value of the character written. If an error occurs, *EOF* is returned.

For example, the following program writes a string to the specified stream.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char text[80];
    int i=0;
    printf("Enter a Text: ");
    gets(text);
    if ((fptr = fopen("test","w"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    while(text[i]!='\0') fputc(text[i++],fptr);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

fgetc

Declaration: *int fgetc(FILE *stream);*

The *fgetc* function returns the next character from the specified input stream and increments the file position indicator. The character is read as an unsigned char that is converted to an integer. If the end-of-file is reached, *fgetc* returns *EOF*. *fgetc* also returns *EOF* if an error occurs.

For example, the following program reads and displays the contents of a text file.

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



```
main()
{   FILE *fptr;
    char c;
    if ((fptr = fopen("test","r"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    while((c=fgetc(fptr)) != EOF)
        putchar(c);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

putc

Declaration: *int putc(int ch, FILE *stream);*

The *putc* function writes the character *ch* to the specified stream at the current file position and then advances the file position indicator. Even though the *ch* is declared to be an *int*, it is converted by *putc* into an *unsigned char*.

The value returned by the *putc* is the value of the character written. If an error occurs, EOF is returned. The *putc* and *fputc* functions are identical.

For example, the following program writes a string to the specified stream.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char text[80];
    int i=0;
    printf("Enter a Text : ");
    gets(text);
    if ((fptr = fopen("test","w"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    while(text[i]!='\0') putc(text[i++],fptr);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

getc

Declaration: *int getc(FILE *stream);*

The *getc* function returns the next character from the specified input stream and increments the file position indicator. The character is read as an *unsigned char* that is converted to an integer.

If the end-of-file is reached, *getc* returns *EOF*. *getc* also returns *EOF* if an error occurs. The functions *getc* and *fgetc* are identical.

For example, the following program reads and displays the contents of a text file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char c;
    if ((fptr = fopen("TEST","r"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    while((c=getc(fptr)) != EOF)
        putchar(c);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

fputs

Declaration: *int fputs(const char *str, FILE *stream);*

The *fputs* function writes the contents of the string pointed to by *str* to the specified *stream*. The null terminator is not written. The *fputs* function returns nonnegative on success and *EOF* on failure.

For example, the following program uses *fputs* to write a string into a file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char text[80];
    int i=0;
    printf("Enter a Text : ");
    gets(text);
    if ((fptr = fopen("test","w"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    fputs(text,fptr);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```


fgets

Declaration: *char *fgets(char *str, int num, FILE *stream);*

The *fgets* function reads up to *num*-1 characters from *stream* and store them in the character array pointed to by *str*. Characters are read until either a newline or an *EOF* is received or until the specified limit is reached. After the characters have been read, a null is stored in the array immediately after the last character is read. A newline character will be retained and will be a part of the array pointed to by *str*.

If successful, *fgets* returns *str*; a null pointer is returned upon failure. If a read error occurs, the contents of the array pointed to by *str* are indeterminate.

For example, the following program uses *fgets* to display the content of a file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char text[80];
    if ((fptr = fopen("test","r"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    fgets(text,80,fptr);
    puts(text);
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

fprintf

Declaration: *int fprintf(FILE *stream, const char *format, ...);*

The *fprintf* function outputs the values of the arguments that makes up the argument list as specified in the *format string* to the stream pointed to by *stream*. The operations of the format control string and commands are identical to those in *printf*.

fscanf

Declaration: *fscanf(FILE *stream, const char *format, ...);*

The *fscanf* function works exactly like the *scanf* function except that it reads the information from the stream specified by *stream* instead of standard input device.

The following program uses the *fprintf* and *fscanf* functions to prepare the results of '*n*' students.

```
/* To prepare the mark list*/
#include<stdio.h>
main()
{   int regno,m[5],tot,i,j,n,k;
    char name[15];
    float avg;
    FILE *fptr;
    if((fptr=fopen("mark","w")) == NULL)
    {   printf("Error in Opening File..");
        exit(1);
    }
    printf("How many students ? ");
    scanf("%d", &n);
    for(i=0;i<n;++i)
    {
        printf("Enter the details of Student-%d\n\n",i+1);
        printf("Name ? ");
        scanf("  %[^\n]",name);
        printf("Reg. No ? ");
        scanf("  %d",&regno);
        for(tot=0,j=0;j<5;++j)
        {   printf("Mark in Paper-%d ? ",j+1);
            scanf("%d",&m[j]);
            tot+=m[j];
        }
        avg=(float)tot/5.0;
        fprintf(fptr,"%15s %d %d %d %d %d %d %f\n",
                        name,regno,m[0],m[1],m[2],m[3],m[4],tot,avg);
    }
    fclose(fptr);
    if((fptr=fopen("mark","r")) == NULL)
    {   printf("Error in Opening File..");
        exit(1);
    }
    printf("Name\t\tReg.No\tM1  M2  M3  m4  M5  Total  Avg   Res\n\n");
    for(i=0;i<n;++i)
    {   fscanf(fptr,"%-15s %d %d %d %d %d %d %d %f", name, &regno, &m[0], &m[1],
                &m[2],&m[3],&m[4],&tot,&avg);
        printf("\n%15s %4d %4d %4d %4d %4d %4d %4d %6.2f", name,regno,m[0], m[1],
                m[2],m[3],m[4],tot,avg);
        if(avg>35.0)
            printf(" Passed");
        else
            printf(" Failed");
    }
    fclose(fptr);
```



```
    return;
}
```

3.2. ERROR HANDLING FUNCTION

Error can happen during file I/O operations. Typical file I/O errors include:

- Trying to read beyond the end-of-file
- Device overflow
- Trying to use a file that is not opened
- Trying to perform an operation on a file when it is opened in a different mode
- Opening a file with an invalid file name
- Attempting to write to a write protected file

In the absence of error handling mechanism in the program, a program behaves abnormally in the event of occurring any of these kinds of errors.

The *fopen* and *fclose* functions themselves can be used for tracking errors.

For example, the *fopen* function returns a pointer to the beginning of the buffer area associated with the file if the file is successfully opened; a *NULL* value is returned if the file cannot be found. This property of the fopen function can be used for error checking, as illustrated below.

```
FILE *fptr;
if (( fptr=open("TEST", "w") ) == NULL)
{   printf("Cannot Open File\n");
    exit(1);
}
```

This method detects any error in opening a file, such as write-protected or a full disk before attempting to write to it. If no file by that name exists, one will be created. Opening a file for read operations require that the file exists.

The *fclose* function closes the file associated with stream and flushes its buffer. If *fclose*() is successful, zero is returned; otherwise *EOF* is returned. This property of the fclose function can also be used for error handling to ensure that the file has been successfully closed, as illustrated below.

```
if(fclose(fptr)) printf("File Close Error.\n");
```

If *fptr* is anything otherthan zero, the error message gets displayed.

C also supports two standard functions – *feof* and *ferror* – that can also be used for error handling.

feof

Declaration: *int feof(FILE *stream);*

The *feof* function determines whether the end-of-file associated with stream has been reached. A nonzero value is returned if the file position indicator is at the end-of-file; zero is returned otherwise.

While processing a file, the feof function can be used to check whether the end-of-file has been reached or not and continue the processing accordingly. For example, the following program tests for *not* end-of-file to repeatedly read a character from a file until the end-of-file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{   FILE *fptr;
    char c;
    if ((fptr = fopen("test","r"))==NULL)
    {   printf("Cannot Open File.\n");
        exit(1);
    }
    while(!feof(fptr))
        putchar(c=fgetc(fptr));
    if(fclose(fptr)) printf("File Close Error.\n");
    return;
}
```

ferror

Declaration: *int ferror(FILE *fp);*

The *ferror* function determines whether a file operation has produced an error. It returns *true* if an error has occurred during the last file operation; otherwise, it returns *false*. Because each file operation sets the error condition, *ferror* should be called immediately after each file operation; otherwise, an error may be lost.

4. USING A SEQUENTIAL ACCESS FILE

A sequential file provides access to data sequentially. All the file access functions so far we have discussed are useful for reading and writing data sequentially. Similarly, the sample file handling programs discussed previously in this chapter also deal with sequential file.

While dealing with files, remember that C imposes no structure on a file. Thus, notions such as a record of a file do not exist as part of the C language. Therefore, you must provide a file structure to meet the requirements of a particular application.

For example, the following program shows how to impose a record structure on a sequential file.

```
#include<stdio.h>
#include<stdlib.h>
main( void )
{   int account; /* account number */
    char name[30]; /* account name */
    double balance; /* account balance */
    FILE *cfPtr; /* cfPtr = clients.dat file pointer */
    /* fopen opens file. Exit program if unable to create file */
    if((cfPtr = fopen("clients.dat", "w" )) == NULL)
        printf("File could not be opened\n" );
    else
    {   do
        {   printf("Enter the Account, Name and Balance (Ctrl+Z to End): " );
            scanf("%d%s%lf", &account, name, &balance);
            fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
            }
        while(!feof(stdin));
        fclose(cfPtr ); /* fclose closes file */
    }
}
```

Once the data is stored in files, it can be retrieved for processing when needed. For example, the following program illustrates how to read data sequentially from a file.

```
#include<stdio.h>
#include<stdlib.h>
main( void )
{   int account; /* account number */
    char name[30]; /* account name */
    double balance; /* account balance */
    FILE *cfPtr; /* cfPtr = clients.dat file pointer */
    /* fopen opens file. Exit program if unable to create file */
    if((cfPtr = fopen("clients.dat", "r" )) == NULL)
        printf("File could not be opened\n" );
    else
    {   printf( "%-10s%-13s%s\n", "Account", "Name", "Balance");
        while(!feof(cfPtr))
        {   fscanf( cfPtr, "%d%s%lf", &account, name, &balance);
            printf( "%-10d%-13s%7.2f\n", account, name, balance);
        }
        fclose(cfPtr ); /* fclose closes file */
    }
}
```

To retrieve data sequentially from a file, a program normally starts reading from the beginning of the file and reads all data consecutively until the desired data is found. It may be desirable to process the data sequentially in a file several times (from the beginning of the file) during the execution of a program. In such cases, you may have explicitly reset the file pointer back to the beginning of the data file using the function *rewind* with the following prototype:

*void rewind(FILE *stream)*

The *rewind* function moves the file position indicator to the start of the specified stream. It also clears the end-of-file and error flags. The file position pointer is not really a pointer. Rather it is an integer value that specifies the byte location in the file at which the next read or write is to occur. This is sometimes referred to as the file offset. The file position pointer is a member of the FILE structure associated with each file.

For example, the accounts application is to be modified such that the followings accounts details must be displayed: (*a*) Zero Balance Accounts, (*b*) Credit Balance Accounts and (*c*) Debit Balance Accounts. This requires resetting of the file pointer every time before the file is processed.

The following code achieves this:

```
#include<stdio.h>
#include<stdlib.h>
main( void )
{   int account, request; /* account number */
    char name[30]; /* account name */
    double balance; /* account balance */
    FILE *cfPtr; /* cfPtr = clients.dat file pointer */
    /* fopen opens file. Exit program if unable to create file */
    if((cfPtr = fopen("clients.dat", "rw" )) == NULL)
        printf("File could not be opened\n" );
    else
    {   do
        {   printf("Enter the Account, Name and Balance (Ctrl+Z to End): " );
            scanf("%d%s%lf", &account, name, &balance);
            fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
        }
        while(!feof(stdin));
        /* display request options */
        do
        {   printf( "Enter request\n"
                " 1 - List accounts with zero balances\n"
                " 2 - List accounts with credit balances\n"
                " 3 - List accounts with debit balances\n"
                " 4 - End of run\nEnter Your Choice (1-4): " );
```



```
        scanf("%d", &request );
        switch ( request )
        {   case 1:
                rewind(cfPtr);
                printf( "\nAccounts with Zero Balances:\n" );
                printf( "%-10s%-13s%s\n", "Account", "Name", "Balance");
                while(!feof(cfPtr))
                {   fscanf( cfPtr, "%d%s%lf", &account, name, &balance);
                if(balance==0)
                        printf( "%-10d%-13s%7.2f\n", account, name, balance);
                }
                break;
            case 2:
                rewind(cfPtr);
                printf( "\nAccounts with Credit Balances:\n" );
                printf( "%-10s%-13s%s\n", "Account", "Name", "Balance");
                while(!feof(cfPtr))
                {   fscanf( cfPtr, "%d%s%lf", &account, name, &balance);
                if(balance>0)
                        printf( "%-10d%-13s%7.2f\n", account, name, balance);
                }
                break;
            case 3:
                rewind(cfPtr);
                printf( "\nAccounts with Debit Balances:\n" );
                printf( "%-10s%-13s%s\n", "Account", "Name", "Balance");
                while(!feof(cfPtr))
                {   fscanf( cfPtr, "%d%s%lf", &account, name, &balance);
                if(balance<0)
                        printf( "%-10d%-13s%7.2f\n", account, name, balance);
                }
                break;
        }
    }
    while (request != 4);
    fclose(cfPtr ); /* fclose closes file */
  }
}
```

5. USING RANDOM ACCESS FILE

Records in a file created with the formatted output function *fprintf* are not necessarily the same length. However, individual records of a random access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records. This makes random-access files appropriate for airline reservation systems, banking systems, point-of-sale systems, and other kinds of

transaction processing systems that require rapid access to specific data. There are other ways of implementing random-access files, but we will limit our discussion to this straightforward approach using fixed-length records.

Because every record in a random-access file normally has the same length, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key. This facilitates immediate access to specific records, even in large files.

Fixed-length records enable data to be inserted in a random-access file without destroying other data in the file. Data stored previously can also be updated or deleted without rewriting the entire file.

5.1. CREATING A RANDOM ACCESS FILE

These *fread* and *fwrite* functions allow the reading and writing of blocks of any type of data.

*unsigned integer fread(void *buffer, unsigned integer num_bytes, unsigned integer count, FILE *fp);*

*unsigned integer fwrite(const void *buffer, unsigned integer num_bytes, unsigned integer count, FILE *fp);*

For *fread*, *buffer* is a pointer to a region of memory that will receive the data from the file. For *fwrite*, *buffer* is a pointer to the information that will be written to the file. The value of *count* determines how many items are read or written, with each item being num_bytes bytes in length. Finally, *fp* is a file pointer to a previously opened stream.

The *fread* function returns the number of items read. This value may be less than *count* if the end of the file is reached or an error occurs. The *fwrite* function returns the number of items written. This value will equal *count* unless an error occurs.

Function *fwrite* transfers a specified number of bytes beginning at a specified location in memory to a file. The data is written beginning at the location in the file indicated by the file position pointer. Function *fread* transfers a specified number of bytes from the location in the file specified by the file position pointer to an area in memory beginning with a specified address.

For example, for a 4-byte integer, we can use

fwrite(&number, sizeof(int), 1, fPtr);

writes 4 bytes (or 2 bytes on a system with 2-byte integers) from a variable *number* to the file represented by *fPtr*. Later, *fread* can be used to read 4 of those bytes into an integer variable *number*.

File processing programs rarely write a single field to a file. Normally, they write one *struct* at a time, as we show in the following examples.

Consider the following problem statement:

Create a credit processing system capable of storing up to 100 fixed-length records. Each record should consist of an account number that will be used as the record key, a name and a balance. The resulting program should be able to update an account, insert a new account record, delete an account and list all the account records in a formatted text file for printing. Use a random-access file.

The next few sections introduce the techniques necessary to create the credit processing program.

The following program shows how to open a random-access file, define a record format using a *struct*, write data to the disk and close the file. This program initializes all 100 records of the file *credit.dat* with empty *structs* using the function *fwrite*. Each empty struct contains 0 for the account number, "" (the empty string) for the last name and the first name and 0.0 for the balance. The file is initialized in this manner to create space on the disk in which the file will be stored and to make it possible to determine if a record contains data.

```
#include <stdio.h>
/* clientData structure   definition */
struct clientData
{   int acctNum;                /*account number */
    char lastName[15];  /*account last name */
    char firstName[10]; /*account first name */
    double balance;             /*account balance */
}; /* end structure clientData */
main()
{   int i; /* counter used to count from 1-100 */
    /* create clientData with default information */
    struct clientData blankClient = { 0, "", "", 0.0 };
    FILE *cfPtr; /* credit.dat file pointer */
    /* fopen opens the file; exits if file cannot be opened */
    if((cfPtr =fopen("credit.dat","wb"))==NULL)
        printf( "File could not be opened.\n" );
    else
    {   /* output 100 blank records to file */
        for ( i = 1; i <= 100; i++ )
            fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
        fclose(cfPtr ); /* fclose closes the file */
    } /* end else */
} /* end main */
```

5.2. WRITING DATA RANDOMLY TO A RANDOM FILE

The program given below writes data to the file *credit.dat*. It uses the combination of *fseek* and *fwrite* to store data at specific locations in the file. Function *fseek* sets the file position pointer to a specific position in the file, then *fwrite* writes the data.

The function fseek has the following prototype:

*int fseek(FILE *fp, long int offset, int origin);*

Here, *fp* is a file pointer returned by a call to *fopen*, *offset* is the number of bytes from *origin*, which will become the new current position, and *origin* is one of the following macros: SEEK_SET (or 1) for beginning of the file, SEEK_CUR (or 2) for current position and SEEK_END (or 3) for end of the file. The *offset* may be positive, meaning move forwards, or negative, meaning move backwards.

When the operation is successful, *fseek* returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and *fseek* returns -1. Before proceeding further, it is always desirable to include an error check after the *fseek* operation.

Another function useful for handling random files is *ftell*. It can be used to determine the current location of a file. Its prototype is

*long int ftell(FILE *fp);*

It returns the location of the current position of the file associated with *fp*. If a failure occurs, it returns –1.

```
/* Writing to a random access file */
#include <stdio.h>
/* clientData structure definition */
struct clientData
{   int acctNum;                /*account number */
    char lastName[15]; /*account last name */
    char firstName[10]; /*account first name */
    double balance;             /*account balance */
}; /* end structure clientData */
main()
{   FILE * cfPtr; /* credit.dat file pointer */
    /* create clientData with default information */
    struct clientData client = { 0, "", "", 0.0 };
    /* fopen opens the file; exits if file cannot be opened */
    if ( (cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL )
        printf( "File could not be opened.\n" );
    else {   /*user enters information, which is copied into file*/
    do   {   printf( "Enter account number (1 to 100, 0 to End): " );
             scanf( "%d", &client.acctNum );
             if(client.acctNum ==0 ) break;
```



```
             printf( "Enter First Name: " );
             fscanf(stdin, "%s", client.firstName);
             printf( "Enter Last Name: " );
             fscanf( stdin, "%s", client.lastName);
             printf( "Enter Balance: " );
             /* ensure a blank space before lf*/
             fscanf( stdin, " %lf", client.balance);
             /* seek position in file to user-specified record */
             fseek(cfPtr, (client.acctNum - 1)*
                 sizeof(struct clientData ), SEEK_SET );
             /* write user-specified information in file */
             fwrite( &client, sizeof( struct clientData ), 1, cfPtr);
         }
        while (client.acctNum != 0 );
        fclose( cfPtr ); /* fclose closes the file */
    } /* end else */
}   /* end   main */
```

5.3. READING DATA FROM A RANDOM ACCESS FILE

Function *fread* reads a specified number of bytes from a file into memory. For example,

fread(&client, sizeof(struct clientData), 1, cfPtr);

reads the number of bytes determined by *sizeof(struct clientData)* from the file referenced by *cfPtr* and stores the data in the structure *client*. The bytes are read from the location in the file specified by the file position pointer.

Function *fread* can be used to read several fixed-size array elements by providing a pointer to the array in which the elements will be stored and by indicating the number of elements to be read. The preceding statement specifies that one element should be read. To read more than one element, specify the number of elements in the third argument of the *fread* statement.

Function *fread* returns the number of items it successfully input. If this number is less than the third argument in the function call, then a read error occurred.

The following program reads sequentially every record in the "*credit.dat*" file, determines whether each record contains data and displays the formatted data for records containing data. Function *feof* determines when the end of the file is reached, and the *fread* function transfers data from the disk to the *clientData* structure *client*.

```
/* Reading data sequentially from random access file */
#include <stdio.h>
/* clientData structure definition */
struct clientData
{   int acctNum;                /*account number */
```

```
    char lastName[15];  /*account last name */
    char firstName[10]; /*account first name */
    double balance;             /*account balance */
}; /* end structure clientData */
main()
{   FILE * cfPtr; /* credit.dat file pointer */
    /* create clientData with default information */
    struct clientData client = { 0, "", "", 0.0 };
    /* fopen opens the file; exits if file cannot be opened */
    if ( (cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL )
        printf( "File could not be opened.\n" );
    else {   /* read all records from file (until eof) */
             printf( "%-6s%-11s%-16s%10s\n", "Acct", "First Name",
                 "Last Name", "Balance");
             do
             {   fread( &client, sizeof( struct clientData ), 1, cfPtr );
                 if ( client.acctNum != 0 )
                     printf( "%-6d%-11s%-16s%10.2f\n",
                         client.acctNum, client.firstName,
                         client.lastName, client.balance );
             }
             while (!feof(cfPtr));
             fclose(cfPtr); /* fclose closes the file */
         } /* end else */
}   /* end main */
```

The complete Accounts program with options to

1 - Enter a Customer Detail
2 - Display an Account
3 - List All Accounts
4 - List Accounts with Credit Balances
5 - List Accounts with Debit Balances
6 - Edit a Customer Details
7 - Delete a Customer Details
8 - End of Run

is given below.

```
/* Reading data sequentially from random access file */
#include <stdio.h>
/* clientData structure definition */
struct clientData
{   int acctNum;            /*account number */
    char lastName[15];  /*account last name */
    char firstName[10]; /*account first name */
    double balance;             /*account balance */
}; /* end structure clientData */
```



```
main()
{   int request, ano;
char c;
FILE *cfPtr; /* credit.dat file pointer */
  /* create clientData with default information */
  struct    clientData client = { 0, "", "", 0.0 };
  /* fopen opens the file; exits if file cannot be opened */
  if ( (cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL )
  {   printf( "File could not be opened.\n" );
   exit(0);
  }
  do
  {
    printf( "\n\n\t\tEnter Your Choice:\n"
      "\t\t 1 - Enter a Customer Detail\n"
      "\t\t 2 - Display an Account\n"
      "\t\t 3 - List All Accounts\n"
      "\t\t 4 - List Accounts with Credit Balances\n"
      "\t\t 5 - List Accounts with Debit Balances\n"
      "\t\t 6 - Edit a Customer Details\n"
      "\t\t 7 - Delete a Customer Details\n"
      "\t\t 8 - End of Run\n\t\tEnter Your Choice (1-8): ");
    scanf(" %d", &request );
    if((request<1) || (request>8)) continue;
    switch (request)
    {   case 1:
          do
          {  printf( "Enter Account Number (1 to 100): " );
             scanf("%d", &ano );
             if((ano<1)||(ano>100))
                printf( "\nInvalid Account Number. Should be 1 to 100\n" );
          }
          while((ano<1)||(ano>100));
          fseek(cfPtr, (ano - 1)*
                    sizeof(struct clientData ), SEEK_SET );
          fread( &client, sizeof( struct clientData ), 1, cfPtr );
                if ( client.acctNum != 0 )
                    printf("Account Already Exist. Press Any Key to Continue...");
          else
          {   /* Write only if not already exist */
             client.acctNum=ano;
             printf("Enter First Name:  " );
             fscanf(stdin, "%s", client.firstName);
             printf("Enter Last Name:  ");
             fscanf( stdin, "%s", client.lastName);
             printf("Enter Balance: ");
```

```
            /* ensure a blank space before lf*/
            fscanf(stdin, " %lf", &client.balance);
            /* seek position in file to user-specified record */
            fseek(cfPtr, (client.acctNum - 1)*
              sizeof(struct clientData ), SEEK_SET );
            /* write user-specified information in file */
            fwrite( &client, sizeof( struct clientData ), 1, cfPtr);
            printf( "\nOne Account Saved. \nPress a Key to Continue..." );
          }
        break;
        case 2:
          do
          { printf( "Enter Account Number to Display(1 to 100): " );
            scanf("%d", &ano );
            if((ano<1)||(ano>100))
              printf( "\nInvalid Account Number. Should be 1 to 100\n" );
          }
          while((ano<1)||(ano>100));
          fseek(cfPtr, (ano - 1)*
                  sizeof(struct clientData ), SEEK_SET );
          fread( &client, sizeof( struct clientData ), 1, cfPtr );
              if ( client.acctNum == 0 )
                printf("Account Does Not Exist. \nPress Any Key to Continue...");
          else
          {  /* Write only if Account Exist */
              printf("%-6s%-11s%-16s%10s\n", "Acct", "First Name", "Last Name",
                "Balance");
            printf( "%-6d%-11s%-16s%10.2f\n",
                    client.acctNum, client.firstName,
                    client.lastName, client.balance );
            printf( "\nPress a Key to Continue..." );
          }
        break;
      case 3:
        rewind(cfPtr);
        printf("%-6s%-11s%-16s%10s\n",    "Acct",    "First    Name","Last    Name",
            "Balance");
        do
        { fread( &client, sizeof( struct clientData ), 1, cfPtr );
            if ( client.acctNum != 0 )
            printf( "%-6d%-11s%-16s%10.2f\n",
              client.acctNum, client.firstName,
              client.lastName, client.balance );
      }
      while (!feof(cfPtr));
      break;
```



```
      case 4:
        rewind(cfPtr);
        printf("%-6s%-11s%-16s%10s\n", "Acct", "First Name","Last Name",
            "Balance");
        do
        { fread( &client, sizeof( struct clientData ), 1, cfPtr );
            if ((client.acctNum > 0 )&&(client.balance>0))
            printf( "%-6d%-11s%-16s%10.2f\n",
              client.acctNum, client.firstName,
              client.lastName, client.balance );
        }
        while (!feof(cfPtr));
        break;
      case 5:
        rewind(cfPtr);
        printf("%-6s%-11s%-16s%10s\n",    "Acct",    "First    Name","Last    Name",
            "Balance");
        do
        { fread( &client, sizeof( struct clientData ), 1, cfPtr );
          if ( (client.acctNum < 0 )&&(client.balance>0) )
            printf( "%-6d%-11s%-16s%10.2f\n",
              client.acctNum, client.firstName,
              client.lastName, client.balance );
        }
        while (!feof(cfPtr));
        break;
      case 6:
        do
        { printf( "Enter Account Number to Edit(1 to 100): " );
          scanf("%d", &ano );
          if((ano<1)||(ano>100))
            printf( "\nInvalid Account Number. Should be 1 to 100\n" );
        }
        while((ano<1)||(ano>100));
        fseek(cfPtr, (ano - 1)*
                sizeof(struct clientData ), SEEK_SET );
        fread(&client, sizeof( struct clientData), 1, cfPtr );
            if ( client.acctNum == 0 )
              printf("Account Does Not Exist. \nPress Any Key to Continue...");
        else
        {  /* Edit only if Account Exist */
          printf("\n Existing Details");
          printf("\n ----------------\n");
          printf( "\nAccount No: %d",client.acctNum);
          printf( "\nFirst Name: %s",client.firstName);
          printf( "\nLast Name: %s",client.lastName);
```

```
            printf( "\nBalance: %lf",client.balance );
            printf("\n Enter New Details");
            printf("\n ----------------\n");
            printf("Enter First Name:  " );
            fscanf(stdin, "%s", client.firstName);
            printf("Enter Last Name:  ");
            fscanf( stdin, "%s", client.lastName);
            printf("Enter Balance: ");
            fscanf(stdin, " %lf", &client.balance);
            fseek(cfPtr, (ano - 1)*
              sizeof(struct clientData ), SEEK_SET );
            /* write user-specified information in file */
            fwrite(&client, sizeof( struct clientData ), 1, cfPtr);
            printf("\nOne Account Modified.);
          }
          break;
        case 7:
          do
          { printf( "Enter Account Number to Delete(1 to 100): " );
            scanf("%d", &ano );
            if((ano<1)||(ano>100))
              printf( "\nInvalid Account Number. Should be 1 to 100\n" );
          }
          while((ano<1)||(ano>100));
          fseek(cfPtr, (ano - 1)*
                  sizeof(struct clientData ), SEEK_SET );
          fread(&client, sizeof( struct clientData), 1, cfPtr );
              if ( client.acctNum == 0 )
                  printf("Account Does Not Exist. \nPress Any Key to Continue...");
          else
          {  /* Edit only if Account Exists */
            do
            { printf("Are you sure to Delete (y/n): ");
              scanf("%c",&c);
            }
            while((toupper(c)!='Y')&&(toupper(c)!='N'));
            if(toupper(c)=='Y')
            {  client.acctNum=0;
              fseek(cfPtr, (ano - 1)*
              sizeof(struct clientData ), SEEK_SET );
              /* write user-specified information in file */
              fwrite(&client, sizeof( struct clientData ), 1, cfPtr);
              printf("\nOne Account Deleted. \nPress a Key to Continue..." );
          }
          else
            printf("\nNot Deleted. " );
```



```
            }
            break;
          case 8:
            printf( "Thanks for Using the Program" );
            break;
        }
      }
      while(request!=8);
}
```

6. COMMAND LINE ARGUMENTS

Command line arguments are the arguments specified on the operating system command line at the time of invoking an executable program for execution.

For example, the Linux command

copy file1 file2

copies the contents of *fil1* to *file2*. Here, the command name *copy* is the name of the executable file and *fil1* and *file2* are the command line arguments.

In C, it is possible to accept command line arguments. Two separate built-in arguments to *main* function – *argc* (argument count) and *argv* (argument vector) – are used to receive command line arguments. The *argc* parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The *argv* is an array of pointers to characters, i.e., an array of strings. Each string in this array will represent a parameter that is passed to main. All command line arguments are strings – any numbers will have to be converted by the program into the proper binary format, manually.

The function prototype for main can be thought of as

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

In order to pass one or more parameters to the program when it is executed from the operating system, the parameters must follow the program name on the command line, e.g.,

pgm-name parameter1 parameter2 parametern

The individual items must be separated from one another either by blank spaces or by tabs.

The program name will be stored in as the first item in the *argv*, followed by each parameter. Hence, if the program name is followed by *n* parameters, there will be (*n*+1) entries in *argv*, ranging from *argv*[0] to *argv*[n]. Moreover, *argc* will

The following program creates a command called *cos* that can returns the value of cosine of an angle when given the command *cos* <angle>. Remember to store the source code in a file named *cos.c* and to execute the program from command prompt.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
double fact(int n);
double power(float x, int n);
main(int argc, char *argv[])
{   int i;
    float sign;
    double x,tempx,sum=1.0,term=1.0;
    if(argc>2)
    {   printf("Too many arguments. Use the syntax cos <angle>.");
        exit(1);
    }
    if(argc<2)
    {   printf("Too few arguments. Use the syntax cos <angle>.");
        exit(1);
    }
    /* getting the numeric value of angle from string argument */
    sscanf(argv[1],"%lf",&x);
    tempx=x;
    x=x*3.14/180.0;
    sign = -1;
    for(i=2;fabs(term)>0.0001;i+=2)
    {   term=sign*power(x,i)/fact(i);
        sum+=term;
        sign = -sign;
    }
    printf("Cos(%0.2f) = %.2f",tempx,sum);
}
double fact(int n)
{   int i;
    double f = 1.0;
    for(i=2;i<=n;++i)
    f*=i;
    return f;
}
double power(float x, int n)
{   double p=1,i;
    for(i=1;i<=n;++i)
    p=p*x;
    return p;
}
```


Note that the program makes use of the function *sscanf* which works exactly like *scanf* except that the input will be from a string rather than standard input.

7. PRE-PROCESSORS

The C pre-processor is program that processes a source program according to the directives supplied in the program before it is being passed to the compiler. When a program issues a command to compile a C program, the program is run automatically through the pre-processor. The pre-processor does not modify the original C program, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. For example, if a program contains the following symbolic constant definition directive

#define NULL 0

the pre-processor replaces every occurrences of NULL following the *#define* directive. The resulting program no longer includes the directive (since the directives are only for the processor, not the compiler)

All preprocessor directives begin with the sharp sign (#). They must start in the first column, and most C compilers there can be no space between # sign and the directive. The directive is terminated not by semicolon, but by the end of line on which it appears. Only one directive can occur on a line. The following are the commonly used pre-processor directives in C.

- Macro substitution
- File inclusion
- Conditional compilation
- Predefined names

7.1. MACRO SUBSTITUTION

Pre-processing directive #define can be used to define macro instructions (commonly called macros) in C program. A macro is an abbreviation used for a sequence of key combinations. The *#define* directive occur in two forms:

#define identifier TokenString

#define identifier(identifier$_1$, identifier$_2$, ..., identifier$_n$)

The *TockenString* is optional. A long definition of either form can be continued to the next line placing a back slash (\) at the end of the current line.

If a simple *#define* of the first form occurs in a file, the pre-processor replaces every occurrence of *identifier* by *TokenString* in the remainder of the file, except a quoted string. That is, in its simplest form, the *#define* directive can be used to define symbolic constant, as shown in the following example.

```
#define SECONDSPERDAY (60*60*24)
```

In this example, the *TokenString* is (60*60*24), and the pre-processor will replace every occurrence of the symbolic constant *SECONDSPERDAY* by the *TokenString* in the remainder of the file. The use of simple *#define* can improve program clarity and portability. For example, if special constants such as π or speed of light c are used in a program, they should be defined.

```
#define PI 3.14159
#define C 299792.458
```

Other special constants are also best coded as symbolic constants.

```
#define EOF (-1)
#define MAXINT 2147483647
```

In general, symbolic constants aid documentation by replacing constants with meaningful mnemonic identifier. They aid portability by allowing constants that may be system-dependent to be altered once. They aid reliability by restricting to one place the check on the actual representation of the constant.

In C programming, it is customary to use capital letters for macro template. This makes it easy for programmers to pick out all macro templates when reading through the program.

A *#define* directive could be used replace a condition, or a whole statement or a block of statements, as shown below.

```
#define AND &&
#define RANGE (a>25 AND a<50)
#define BLOCK if (RANGE)\
                printf("Within Range);\
              else\
                printf("Out of Range");
main()
{   int a=30;
    BLOCK
}
```

The macros can also have parameters. Its general format is

#define identifier(identifier$_1$, identifier$_2$, ..., identifier$_n$)

For example, consider the macro definition

```
#define SQ(x) ((x)*(x))
```

The identifier x in the *#define* is a parameter that is substituted for in the later text. For example, with argument 7+w, the macro call SQ(7+w) expands to *((7+w)*(7+w))*.

There are two pints to remember while writing macros with arguments – there can be no space between the first identifier and the left parentheses and the entire macro definition should be enclosed within parentheses. For example,

```
#define SQ(n) n*n
main()
{   int j;
    j=64/SQ(4);
    printf("%d", j);
}
```

The output of the above program would be $j = 64$, whereas what we expect was $j = 4$. The reason for this is that the macro is expanded into $j = 64/4*4$, which yields 64.

If a pre-processor symbol has already been defined, it must be undefined before being redefined. This is accomplished by the *#undef* directive, which specifies the name of the symbol to be undefined.

For example, in certain occasions, it may desirable to redefine a macro or a constant defined in a header file such as *stdio.h*. If only a few macros from such a file need to be redefined, the easiest way to do this is to *#include* the entire file (discussed below) and then redefine the required macros. For example, a programmer wishes the constant *EOF* to have the value -2. The program could begin with the following directives.

```
#include <stdio.h>
#undef EOF
#define EOF -2
```

7.2. FILE INCLUSION

The directive, *#include*, causes the entire content of one file to be inserted into the source of another file at the point where the *#include* directive is encountered. There exist two ways to write the *#include* directive.

#include <filename>

#include "filename"

The angle brackets tell the pre-processor to search for the file in one or more standard directories. These directories contain header files that are provided by the system, and those commonly used by the programmers. If the brackets are replaced with double quotation marks, the pre-processor looks in the programmers own directory, or the same one that contains the program file. If it is not found there, the standard directories are searched.

File inclusion directive can be used in two cases:

To break larger programs into several different files, each containing a set of related functions. These files are included at the beginning of the main program file.

Often a programmer accumulates a collection of useful functions or macro definitions that are used in almost all programs. In such cases, those commonly used functions and macro definitions can be stored in file, and that file can be included into every programs. Such files are called *header files*, and by convention their names end with the character *.h*.

7.3. CONDITIONAL COMPILATION DIRECTIVES

The C pre-processor has directives for conditional compilation. The conditional compilation directives can be used for selectively removing section of code or choosing between two possible sections. They can be used for program development and for writing code that is more easily portable from one machine to another.

Each processing directive of the form

#if ConstantIntegralExpression

#ifdef identifier

#ifndef identifier

provided for conditional compilation of the code that follows until the pre-processing directive

#endif

is reached. For the intervening code to be compiled, after *#if* the *ConstantIntegralExpression* must be non zero (*true*), and after *#ifdef* (if defined), the named *identifier* must have been defined previously in a *#define* line, without an intervening

#undef identifier

having been used to undefine the macro. After *#ifndef* (if not defined) directive, the named *identifier* must be currently undefined.

For example, sometimes *printf()* statements are useful for debugging purposes. Suppose that the top of the file contains the following definition

```
#define DEBUG 1
```

and then throughout the rest of the file one can write debugging lines such as

```
#if DEBUG
    printf("debug: a = %d\n", a);
#endif
```


Because the symbolic constant *DEBUG* has nonzero value, the *printf()* statement will be compiled. Later, these lines can be omitted from compilation by changing the value of symbolic constant *DEBUG* to 0.

```
#undef DEBUG
#define DEBUG 0
```

An alternate way to achieve this is to define the symbolic constant having no value. Suppose the top of the file contains the symbolic constant definition

```
#define DEBUG
```

Then one can use the *#ifdef* directive for conditional compilation as shown below.

```
#ifdef DEBUG
    printf("debug: a = %d\n", a);
#endif
```

Later, the intervening lines of code can be excluded from compilation by removing the *#define* directive from the top of the file or undefined the symbol by using *#undef* directive.

A practical use of conditional compilation arises during the testing phase of the program development. For debugging or testing purposes, the programmer may wish to temporarily exclude a section of code from compilation. A common approach to achieve this is to put the code into a comment by using the comment delimiters /* and */. However, if the code to be blocked out contains comments, this method will result in a syntax error (C does not support nested comments). The use of conditional compilation solves this problem.

```
#if 0
    statements
#endif
```

Each of the *#if* forms can be followed by any number of lines, possibly containing pre-processing directives of the form

#elif ConstantIntegralExpression

possible followed by the pre-processing directive

#else

and, finally, followed by the pre-processing directive

#endif

7.4. THE PREDEFINED MACROS

In ANSI C, there are five predefined macros. They are always available, and they cannot be redefined or undefined by the programmer. Each of these macro names includes two leading and two trailing underscore characters, as listed below:

- __LINE__: A decimal constant containing the current source line number.
- __FILE__: A string literal containing the name of the file being compiled.
- __DATE__: A string literal containing the date of compilation, in the form "Mmmm dd yyyy"
- __TIME__: A string literal containing the time of compilation, in the form "hh:mm:ss"
- __STDC__: If the implementation follows ANSI standard C, then the value is a nonzero integer.

For example, execution of the following program

```
#include<stdio.h>
main()
{   printf("%s%s\n%s%s\n%s%d\n%s%s\n",
        "__DATE__", __DATE__,
        "__FILE__",__FILE__,
        "__LINE__",__LINE__,
        "__TIME__",__TIME__);
}
```

would produce an output of the form

```
__DATE__Aug 11 2014
__FILE__C:\Users\abbas\Desktop\CProgramming\test.c
__LINE__6
__TIME__13:31:09
```

26.

/* To create a command 'cpy' that is equivalent to the *copy* command of an OS. Assume that this program is stored in a file named 'cpy'. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like *cpy filename1 filename2*. This will copy the contents of the filename1 to filename2. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{ FILE *fptr1,*fptr2;
  char c;
  if(argc>3)
    { printf("Too many parameters ...");
      exit(1);
    }
  else
    if(argc<3)
    { printf("Too few parameters ...");
      exit(1);
    }
  if ((fptr1 = fopen(argv[1],"r"))==NULL)
  { printf("Error in Opening the Source File %s ...",strupr(argv[1]));
    exit(1);
  }
  if ((fptr2 = fopen(argv[2],"w"))==NULL)
  { printf("Unable To Create the Destination File %s ...",strupr(argv[2]));
    exit(1);
  }
  while(!feof(fptr1))
    putc(c=getc(fptr1),fptr2);
  if(fclose(fptr1)) printf("File Close Error.\n");
  if(fclose(fptr2)) printf("File Close Error.\n");
  printf("%s Successfully copied to %s\n",strupr(argv[1]),strupr(argv[2]));
}
```

28.

```
/* To create a command 'search', that searches file to check whether a string exists in a file
or not. Assume that this program is stored in a file named 'searchstring'. Compile the
program and execute it from the command mode of the operating system. The command
line initiating the program would looks like search filename search-string. The search-
string should be enclosed in double quotation marks, if it contains more than one word.
This will display the count of the string */

#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{ FILE *fptr1;
  char c;
  int flag = 1,count=0,i=0,len;
  if(argc>3)
    { printf("Too many parameters ...");
      exit(1);
    }
  else
    if(argc<3)
    { printf("Too few parameters ...");
      exit(1);
    }
  if ((fptr1 = fopen(argv[1],"r"))==NULL)
  { printf("Error in Opening File %s ...",strupr(argv[1]));
    exit(1);
  }
  len= strlen(argv[2]);
  while(!feof(fptr1))
  { c=getc(fptr1);
    if (c != argv[2][i++])
      {flag = 0;i=0;}
    else
      flag = 1;
    if(i==len)
      if(flag)
        { count++;
          i=0;
        }
  }
  if(count>0)
    printf("The Word \"%s\" exists %d times in File \"%s\" ", argv[2],count,argv[1]);
  else
    printf("The Word \"%s\" does not exists in File \"%s\"  ", argv[2],argv[1]);
  if(fclose(fptr1)) printf("File Close Error.\n");
}
```

29.

/* To create a command '*charcount*', that counts the number of characters in a file. Store the program in a file named charcount. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like *charcount filename*. This will display the count of the number of characters in the file */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{ FILE *fptr1;
  char c;
  float count = 0;
  if(argc>2)
    { printf("Too many parameters ...");
      exit(1);
    }
  else
    if(argc<2)
    { printf("Too few parameters ...");
      exit(1);
    }
  if ((fptr1 = fopen(argv[1],"r"))==NULL)
  { printf("Error in Opening File %s ...",strupr(argv[1]));
    exit(1);
  }
  while(!feof(fptr1))
  { c=getc(fptr1);
    count++;
  }
  printf("The File \"%s\" contains %g characters ", argv[1],count);
  if(fclose(fptr1)) printf("File Close Error.\n");
}
```

30.

/* Merge the contents of two files. Store the program in a file named merge. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like *merge filename1 filename2 filename*. This will merge the contents of *filename1* and *filename2* to *filename3*. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{ FILE *fptr1,*fptr2,*fptr3;
  char c;
```



```
if(argc>4)
  { printf("Too many parameters ...");
    exit(1);
  }
else
  if(argc<4)
  { printf("Too few parameters ...");
    exit(1);
  }
if ((fptr1 = fopen(argv[1],"r"))==NULL)
{ printf("\a\a\aError in Opening File %s ...",strupr(argv[1]));
  exit(1);
}
if ((fptr2 = fopen(argv[2],"r"))==NULL)
{ printf("Error in Opening File %s ...",strupr(argv[2]));
  exit(1);
}
if ((fptr3 = fopen(argv[3],"w"))==NULL)
{ printf("Error in Opening File %s ...",strupr(argv[3]));
  exit(1);
}
while(!feof(fptr1))
{ c=getc(fptr1);
  putc(c,fptr3);
}
if(fclose(fptr1)) printf("File Close Error.\n");
while(!feof(fptr2))
{ c=getc(fptr2);
  putc(c,fptr3);
}
if(fclose(fptr2)) printf("File Close Error.\n");
if(fclose(fptr3)) printf("File Close Error.\n");
printf("Files \"%s\" and \"%s\" are successfully merged to File \"%s\"",
   argv[1],argv[2],argv[3]);
}
```

31.

/* Check the contents of two files and creates a third file that contains a sequence of 1s and 0s. Store the program in a file named check. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like check *filename1 filename2 filename3*. This will check the contents of filename1 and filename2 on a character-by-character basis and add 1 or 0 to filename3. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
```

```
{ FILE *fptr1,*fptr2,*fptr3;
  char c1,c2;
  if(argc>4)
    { printf("Too many parameters ...");
      exit(1);
    }
  else
    if(argc<4)
    { printf("Too few parameters ...");
      exit(1);
    }
  if ((fptr1 = fopen(argv[1],"r"))==NULL)
  { printf("\a\a\aError in Opening File %s ...",strupr(argv[1]));
    exit(1);
  }
  if ((fptr2 = fopen(argv[2],"r"))==NULL)
  { printf("Error in Opening File %s ...",strupr(argv[2]));
    exit(1);
  }
  if ((fptr3 = fopen(argv[3],"w"))==NULL)
  { printf("Error in Opening File %s ...",strupr(argv[3]));
    exit(1);
  }
  while ( (!feof(fptr1)) || (!feof(fptr2)))
  {
    if(!feof(fptr1))
      c1=getc(fptr1);
    else
      c1='\0';
    if(!feof(fptr2))
      c2=getc(fptr2);
    else
      c1='\0';
    if (c1==c2)
      putc('1',fptr3);
    else
      putc('0',fptr3);
  }
  if(fclose(fptr1)) printf("File Close Error.\n");
  if(fclose(fptr2)) printf("File Close Error.\n");
  if(fclose(fptr3)) printf("File Close Error.\n");
  printf("Files \"%s\" and \"%s\" are successfully Checked and created File \"%s\"",
      argv[1],argv[2],argv[3]);
}
```


43.

```
/* To create the command sum that adds the numbers when invoked with a list of numbers
as in sum10 20, 30, etc */
#include<stdio.h>
#include<math.h>
main(int argc, char *argv[])
{ int i;
  double sum=0,x;
  if(argc<2)
  { printf("No numbers to add...");
    return;
  }
  printf("Sum(");
  for(i=1;i<argc;i++)
  {  sscanf(argv[i],"%lf",&x);
     sum+=x;
     printf("%s,",argv[i]);
  }
  printf(")= %g",sum);
}
```