Catalog

FILE PROCESSING

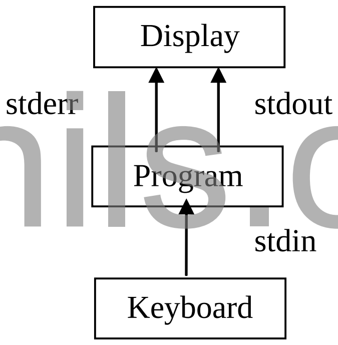
FILES

File is a collection of data stored on secondary storage device such as Hard Disk. It is used to store the data permanently. A file is used when the real life application involves large amount of data.

Streams in C

Stream is a logical interface to a file. In C, there are three standard streams. They are:

a) Standard input (stdin)
b) Standard output (stdout)
c) Standard error (stderr)



Standard Streams

a) Standard input (stdin)

It is the stream from which the program receives its data. Data transfer is done using read operation.

b) Standard output (stdout)

It is the stream where a program writes its output data. Data transfer is done using write operation.

c) Standard error (stderr)

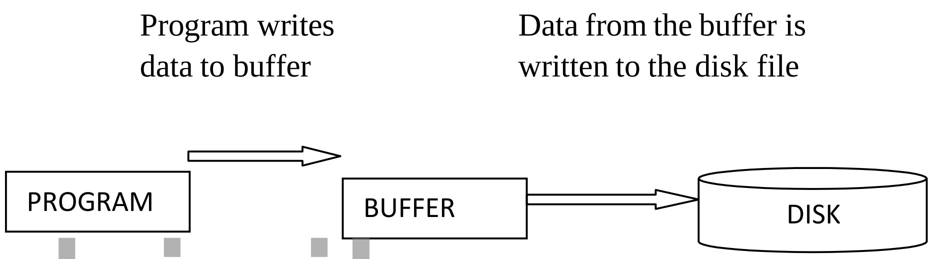
It is the output stream used by the program to report error messages.

Buffers Associated with File Streams

A Stream is linked to a file using an open operation and disconnected from a file using a close operation. When a stream is linked to a file, an associated buffer is created automatically.A Buffer is a temporary storage area from which data are read from or written to a file.

When writing output data to a file by the program, the data is first stored in the buffer until it becomes full. Then the entire data in the buffer is written into the disk file.

When reading input data from a disk file, the data is read as a block from the file and stored into the buffer. The program reads data from the buffer. The data resides in the buffer until the buffer is flushed or written to a file.


Program writes data to buffer
Data from the buffer is written to the disk file
PROGRAM
BUFFER
DISK

Buffers Associated With Stream

Types of Files

Files are categorized into two. They are:

d) ASCII Text Files

e) Binary Files

a) ASCII Text Files

A text file is a sequence of characters which are processed sequentially. The operations done on text files are reading, writing and appending. Each line in a text file can have maximum of 255 characters. Each line ends with a newline character and each file ends with a EOF (end- of-file) marker.

b) Binary Files

A binary file may contain any type of data that is encoded in binary form. It can be processed either sequentially or randomly. The file can be processed randomly by moving the current file position to the required position in the file from where the data can be read or written to. Binary files require less space than the text files to store the same piece of data. Binary file also ends with a EOF marker.

FILE OPERATIONS

The various operations involved in using files in C are

1. Declaring the file pointer variable
2. Opening the file
3. Reading data from file
4. Writing data to the file
5. Closing the file

Declaring the file pointer variable

A file pointer variable has to be declared in order to access a particular file.

Syntax:

```
FILE *file_pointer_name;
```

Example:

```
FILE *fp;
```

Here, fp is the file pointer variable.

Opening the file

A file must be opened before read/write operation. The function fopen() is used to open a file and associate it with a stream. It returns a pointer to FILE structure if the file is opened successfully, otherwise a NULL is returned.

Syntax:

```
FILE *fopen(const char *file_name, const char *mode);
```

Here, file_name - Every file has a file name. In fopen(), the file name is specified along with the path of the file in the disk.

Example:

E:\\CSE\\Student.DAT (To represent a backslash in C, another backslash is used.)

mode - Mode represents the type of processing that can be done with the file. The following table shows the different modes of processing the file.

Mode	Description
r	Open a text file for reading
w	Open a text file for writing
a	Append to a text file.
rb	Open a binary file for reading
wb	Open a binary file for writing
ab	Append to a binary file.
r+	Open an existing text file for read/write
w+	Create a new text file for read/write
a+	Append a text file for read/write
r+b/rb+	Open an existing binary file for read/write
w+b/wb+	Create a new binary file for read/write
a+b/ab+	Append a binary file for read/write

File Accessing Modes

Reading Data From File

Data can be read from files using the following functions.

a) fscanf()

b) fgets()

c) fgetc()

d) fread()

Function	**Description**
fscanf	Used to read formatted data from the stream. **Syntax:** int fscanf(FILE *stream, const char *format, ...);
fgets()	Used to read a line from the stream. **Syntax**: char *fgets(char *str, int n, FILE *stream);
fgetc()	Used to read the next character from the stream. **Syntax**: int fgetc(FILE *stream);

fread()	Used to read number of elements specified by num of size specified by size from the stream. **Syntax:** int fread(void *str, size_t size, size_t num, FILE *stream);

Functions to Read Data from File

Writing Data to File

The functions used to write data to files are,

- fprintf()
- fputs()
- fputc()
- fwrite()

Function	Description
fprintf()	Used to write formatted data to the stream. **Syntax:** int fprintf(FILE *stream, const char *format, ...);
fputs()	Used to write a line to the stream. **Syntax:** int fputs(const char *str, FILE *stream);
fputc()	Used to write a character to the stream. **Syntax:** int fputc(int char, FILE *stream);
fwrite()	Used to write number of elements specified by num of size specified by size to the stream. **Syntax:** fwrite(const void *str, size_t size, size_t num, FILE *stream);

Functions to Write Data to File

Closing a File

The file (both text and binary) should be closed after reading/writing. The function fclose() is used to close an already opened file. It disconnects the file pointer and flushes out all the buffers associated with the file.

Syntax:

```
int fclose(FILE *fp);
```

Here fp is the file pointer of the file that has to be closed. The function returns zero for successful completion and a non-zero value otherwise.

The function fcloseall() closes all the streams except the standard streams(stdin, stdout, stderr) and flushes out the stream buffers.

Syntax:

```
int fcloseall(void);
```

Detecting the End-Of-File(EOF)

EOF is a symbolic constant defined in the library function stdio.h with a value -1. There are two ways to detect an EOF.

- Compare the characters that has been read with the value of EOF (ie.,-1).
- Use the standard library function feof() defined in stdio.h. It returns one if EOF has been reached and zero otherwise.

Syntax:

```
int feof(FILE *fp);
```

Program: Program To Open, Write And Close A File

```
# include <stdio.h>
# include <string.h>
int main( )
{
        FILE *fp ;
        char data[50];
        printf( "Opening the file sample.c in write mode" ) ;
        fp = fopen("sample.c", "w") ;                    // opening an existing file
        if ( fp == NULL )
        {
            printf( "Could not open file sample.c" ) ;
            return 1;
        }
        printf( "\n Enter some text from keyboard” );     // getting input from user
        while ( strlen ( gets( data ) ) > 0 )
        {                                                 // writing in the file
            fputs(data, fp) ;
            fputs("\n", fp) ;
        }
        printf("Closing the file sample.c") ;
        fclose(fp) ;                                      // closing the file
        return 0;      }
```

Output:

```
Opening the file sample.c in write mode
Enter some text from keyboard
Hai, How are you?
Closing the file sample.c
```

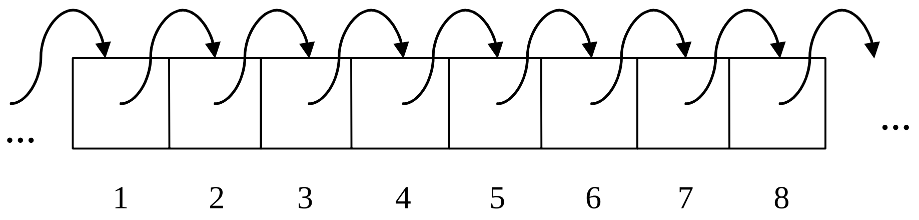
TYPES OF FILE PROCESSING

There are two types of processing based on how a file is accessed. They are,

- Sequential Access
- Random Access

Sequential Access

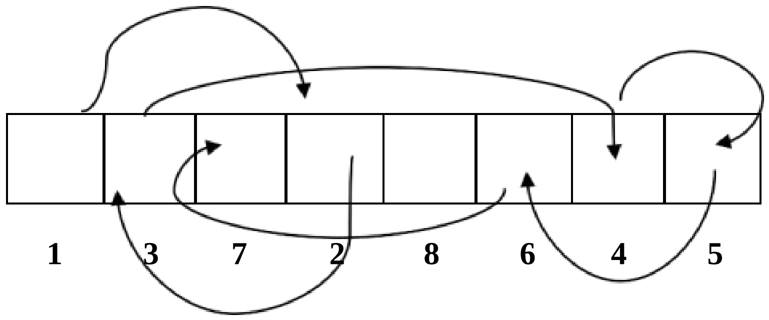
In Sequential Access, the program processes the data in a sequential manner. ie, one by one. In order to read the last record, all the records before that record need to be read.



Sequential Access

Random Access

In Random Access, the program accesses the file at the point at which the data should be read or written. The last record in the file can be read directly.



Random Access

SEQUENTIAL ACCESS FILE

The data items in a sequential access file are arranged one after another in a sequence. Each data item can be accessed in the same order in which they have been stored. It is only possible to read a sequential file from the beginning. To access a particular data within the sequential access file, data has to be read one data at a time, until the required data is reached.

Reading data from a sequential file can be done using the various C functions like fscanf(), fgets(), fgetc(), fread(). Writing data to a sequential file can be done using the various Cfunctions like fprintf(), fputs(), fputc(), fwrite().

EXAMPLE PROGRAM:

Finding average of numbers stored in sequential access file

```
#include <stdio.h>
#include <math.h>
// Read the numbers and return average
float average(FILE *input)
{
        float term,sum;
        int n;
        sum = 0.0;
         n = 0;
        while(!feof(input))
        {
                fscanf(input,"%f",&term);
                sum = sum + term;
                n = n + 1;
        }
         return sum/n;
}
int main ()
{
        FILE *input;
        float avg;
        input = fopen("data.txt","r");
        avg = average(input);
        fclose(input);
        printf("The average of the numbers is %f.\n",avg);
        return 0;
    }
```

Output:

```
    /* Data in data.txt
    10 11 12 13 14 15. */
    The average of the numbers is 12.5
```

RANDOM ACCESS FILE

A random-access data file enables you to read or write information anywhere in the file. Random access allows you to access any record directly at any position in the file. Individual records of a random-access file are normally fixed in length and may be accessed directly 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.

Random access is sometimes called direct access. C supports the following functions for random access file processing of a binary file:

1. fseek()
2. ftell()
3. rewind()
4. fgetpos()
5. fsetpos()

fseek()

This function is used for setting the file pointer at the specified byte. On successful operation, fseek() returns zero. Otherwise it returns a non-zero value.

Syntax:

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

Here,

fp – file pointer

displacement – denotes the number of bytes which are moved backward or forward in the file. It can be positive or negative.

origin – denotes the position relative to which the displacement takes place. It takes one of the following three values.

Constant	**Value**	**Position**
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

Origin field in fseek() function

ftell()

This function returns the current position of the file position pointer. The value is counted from the beginning of the file. If successful, ftell() returns the current file position. In case of error, it returns -1.

Syntax:

long ftell(FILE *fp);

Here, fp – file pointer.

rewind()

This function is used to move the file pointer to the beginning of the file. This function is useful when we open file for update.

Syntax:

void rewind(FILE *fp);

Here, fp – file pointer.

It is impossible to determine if rewind() was successful or not.

fgetpos()

The fgetpos() is used to determine the current position of the stream. The fgetpos function stores the current position of stream into the object pointed to by pos. The fgetpos function returns zero if successful. If an error occurs, it will return a nonzero value.

Syntax:

int fgetpos(FILE *stream, fpos_t *pos);

Here,

stream - The stream whose current position is to be determined.

pos - The current position of stream to be stored.

fsetpos()

The fsetpos() function moves the file position indicator to the location specified by the 'pos' returned by the fgetpos() function. If fsetpos() function successful, it return zero otherwise returns nonzero value.

Syntax:

int fsetpos(FILE *stream, const fpos_t *pos);

Here,

stream − This is the pointer to a FILE object that identifies the stream.

pos − This is the pointer to a fpos_t object containing a position previously obtained with fgetpos.

Difference Between Sequential Access Files And Random Access Files

Sequential Access Files	**Random Access Files**
Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.	Random Access to a file means that the computer system can read or write information anywhere in the data file.
When you access information in the same order all the time, sequential access is faster than random access.	In a random access file, you can search through it and find the data you need more easily.
To read the last record of the file, we have to read all the records from the beginning.	To read the last record of the file, we can read the last record directly.
Example: Tape drives	Example: Hard drives

EXAMPLE PROGRAM: Transaction Processing Using Random Access Files

```
struct clientData
{
        unsigned int acctNum;
        char lastName[ 15 ];
        char firstName[ 10 ];
        double balance;
};
unsigned int enterChoice( void );
void textFile( FILE *readPtr );
void updateRecord( FILE *fPtr );
void newRecord( FILE *fPtr );
void deleteRecord( FILE *fPtr );
```

```
int main( void )
{
    FILE *cfPtr;
    unsigned int choice;
    if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL )
    {
        puts( "File could not be opened." );
    }
    else
    {
        while ( ( choice = enterChoice() ) != 5 )
        {
            switch ( choice )
            {
                // create text file from record file
                case 1:
                    textFile( cfPtr );
                    break;
                // update record
                case 2:
                updateRecord( cfPtr );
                break;
                // create record
                case 3:
                        newRecord( cfPtr );
                        break;
                        // delete existing record
                        case 4:
                                deleteRecord( cfPtr );
                                break;
                        default:
                                puts( "Incorrect choice" );
                                break;
                    }
                }
                fclose( cfPtr );
        }
}
void textFile( FILE *readPtr )
{
        FILE *writePtr;
        int result;
        struct clientData client = { 0, "", "", 0.0 };
        if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL )
        {
                puts( "File could not be opened." );
        }
        else
        {
                rewind( readPtr ); // sets pointer to beginning of file
                fprintf( writePtr, "%-6s%-16s%-11s%10s\n", "Acct", "Last Name", "First
                Name","Balance" );
                while ( !feof( readPtr ) )
                {
                        result = fread(&client, sizeof( struct clientData ), 1, readPtr);
                        // write single record to text file
                        if ( result != 0 && client.acctNum != 0 )
                        {
                                fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
                                client.acctNum,client.lastName,client.firstName,
                                client.balance);
                        }
                }
                fclose( writePtr );
        }
}
void updateRecord( FILE *fPtr )
{
        unsigned int account;
        double transaction;
        struct clientData client = { 0, "", "", 0.0 };
        printf( "%s", "Enter account to update ( 1 - 100 ): " );
        scanf( "%d", &account );
        fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),SEEK_SET );
        fread( &client, sizeof( struct clientData ), 1, fPtr );
        if ( client.acctNum == 0 )
        {
                printf( "Account #%d has no information.\n", account );
        }
        else
        {
                printf( "%-6d%-16s%-11s%10.2f\n\n",client.acctNum, client.lastName,
                client.firstName, client.balance );
                printf( "%s", "Enter charge ( + ) or payment ( - ): " );
                scanf( "%lf", &transaction );
                client.balance += transaction;
                printf( "%-6d%-16s%-11s%10.2f\n",client.acctNum, client.lastName,
                client.firstName, client.balance );
                fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),SEEK_SET );
                // write updated record over old record in file
                fwrite( &client, sizeof( struct clientData ), 1, fPtr );
        }
}
void deleteRecord( FILE *fPtr )
{
        struct clientData client;
        struct clientData blankClient = { 0, "", "", 0 };
        unsigned int accountNum;
        printf( "%s", "Enter account number to delete ( 1 - 100 ): " );
        scanf( "%d", &accountNum );
        fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),SEEK_SET );
        fread( &client, sizeof( struct clientData ), 1, fPtr );
        if ( client.acctNum == 0 )
        {
                        printf( "Account %d does not exist.\n", accountNum );

        }
        else
        {
                fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),SEEK_SET );
                // replace existing record with blank record
                fwrite( &blankClient,sizeof( struct clientData ), 1, fPtr );

        }
}
void newRecord( FILE *fPtr )
{
        struct clientData client = { 0, "", "", 0.0 };
        unsigned int accountNum;
        printf( "%s", "Enter new account number ( 1 - 100 ): " );
        scanf( "%d", &accountNum );
        fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),SEEK_SET );
        fread( &client, sizeof( struct clientData ), 1, fPtr );
        if ( client.acctNum != 0 )
        {
                printf( "Account #%d already contains information.\n",client.acctNum );
        }
        else
        {
                printf( "%s", "Enter lastname, firstname, balance\n? " );
                scanf( "%14s%9s%lf", &client.lastName, &client.firstName,
                &client.balance );
                client.acctNum = accountNum;
               fseek( fPtr, ( client.acctNum - 1 ) *sizeof( struct clientData ), SEEK_SET );
                fwrite( &client,sizeof( struct clientData ), 1, fPtr );
        }
}

unsigned int enterChoice( void )
{
        unsigned int menuChoice;
        printf( "%s", "\nEnter your choice\n"
        "1 - Display a formatted text file of accounts\n"
        "2 - update an account\n"
        "3 - add a new account\n"
        "4 - delete an account\n"
        "5 - end program\n? " );
        scanf( "%u", &menuChoice );
        return menuChoice;
}
```

Output:

```
Enter your choice
1 - Display a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program
?1
Acct    Last Name    First Name    Balance
29      Brown        Nancy         -24.54
33      Dunn         Stacey        314.33
37      Barker       Doug          0.00
88      Smith        Dave          258.34
96      Stone        Sam           34.98
Enter your choice
1 - Display a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program
?2
Enter account to update ( 1 - 100 ): 37
37 Barker    Doug    0.00
Enter charge ( + ) or payment (-): +87.99
37 Barker    Doug    87.99
Enter your choice
1 - Display a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
```

```
5 - end program
?3
Enter new account number ( 1 - 100 ): 22
Enter lastname, firstname, balance ?
Johnston Sarah 247.45
Enter your choice
1 - Display a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program
?4
Enter account number to delete ( 1 - 100 ): 29
Enter your choice
1 - Display a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program
?5
```

COMMAND LINE ARGUMENTS

Any input value passed through command prompt at the time of running of program is known as command line argument. It is used when you want to control your C program from outside. Command line arguments are passed as parameters to the main() function.

Syntax:

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

Here,

argv and argc are the two arguments to the main() function.

The argc parameter holds the number of arguments on the command-line and is an integer. It will always be at least 1 because the name of the program qualifies as the first argument.

The argv parameter is an array of string pointers. The most common method for declaring argv is char *argv[]; The empty brackets indicate that it is an array of undetermined length. argv[0] always contains the name of the program.

Example of command line arguments:

Program : Determine Number of Command Line Arguments and Display Them

```
#include<stdio.h>
#include<conio.h>
void main(int argc, char* argv[])
{
        int i;
        clrscr();
        printf("Total number of arguments: %d",argc);
        for(i=0;i< argc;i++)
        {
        printf("\n %d argument: %s",i,argv[i]);
        getch();
        }
}
```

Output:

```
C:/TC/BIN>TCC mycmd.c
C:/TC/BIN>mycmd 10 20
```

```
Number of Arguments: 3
0 arguments c:/tc/bin/mycmd.exe
1 arguments: 10
2 arguments: 20
```