9. FILES

9.1 Input and Output in C

The standard C library I/O routines allow you to read and write data to and from files and devices. The C language does not include any predefined file structures. Instead, all data is treated as a sequence of bytes. There are three basic types of I/O functions: stream, console and port, and low-level.

All of the stream I/O functions treat data files or data items as a stream of individual characters. If you select the appropriate stream function, your application can process data in any size or format required, from single characters to large, complicated data structures.

Technically, when a program uses the stream function to open a file for I/O, the opened file is associated with a structure of type **FILE** (predefined in *stdio.h*) that contains basic information about the file. Once the stream is opened, a pointer to the **FILE** structure is returned. This FILE pointer - sometimes called the stream pointer or stream – is used to refer to the file for all subsequent I/O.

9.2 Streams

To use the stream functions, your application must include the file *stdio.h*. This file contains definitions for constants, types, and structures used in the stream functions, as well as function declarations and macro definitions for the stream routines. Many of the constants predefined in *stdio.h* can be useful in your application. For example, **EOF** is defined to be the value returned at end of file and **NULL** in the null pointer.

Opening Streams

You can use one of three functions to open a stream before input and output can be performed on it: **fopen**, **fdopen**, **freopen**. The file mode and form are set at the time the stream is opened. The stream file may be opened for reading, writing, or both, and can be opened either in text or in binary mode.

All three functions – **fopen**, **fdopen**, and **freopen** – return a **FILE** pointer, which is used to refer to the stream. For example, if your program contains the line

infile = fopen("sample.dat" , "r") ;

You can use the **FILE** pointer variable *infile* to refer to the stream.

"**r**"	-	open text file for reading – file must exist
"**w**"	-	open text file for writing – file need not exist
"**a**"	-	open text file for appending – file need not exist
"**r+**" and "**+w**"	-	Read and write starting at the beginning of the file
"**rb**"	-	open binary file for reading
"**wb**"	-	open binary file for writing
"**ab**"	-	open binary file for appending

When your application begins execution, five streams are automatically opened. These streams are the standard input (**stdin**), standard output (**stdout**), standard error (**stderr**), standard printer (**stdprn**), and standard auxiliary (**stdaux**). By default, the standard input, standard output, and standard error refer to the user's console. This means that whenever a program expects input from the standard input, it receives that input from the console. Likewise, a program that writes to the standard output prints its data to the console. Any error messages generated by the library routines are sent to the standard error stream – that is, they appear on the user's console. The standard auxiliary and standard print streams usually refer to an auxiliary port and printer.

You can use the five **FILE** pointers in any function that requires a stream pointer as an argument. Some functions, such as **getchar** and **putchar**, are designed to use **stdin** or **stdout** automatically. Since the pointers **stdin**, **stdout**, **stderr**, **stdprn**, and **stdaux** are constants rather than variables.

Redirecting Streams

In MS-DOS, input or output redirection is effortless. You use < to redirect the input and > to redirect the output. Suppose that the executable version of your application is called *redirect*. The following system-level command will run the program *redirect* and use the file *test.dat* instead of the video display as input.

redirect < test.dat

The next statement will redirect both the input (test.dat) and the output (out.dat):

redirect < test.dat > out.dat

The last example will redirect the output (out.dat) only :

redirect > out.dat

Note, however, that you cannot redirect the standard error file **stderr**.

There are two techniques for managing the association between a standard file name and its connection to a physical file or device: redirection and piping. *Piping* involves directly connecting the standard output of one program to the standard input of another. Redirection and piping are normally controlled and invoked outside the program. In this way, the program need not concern itself with where the data is coming from or going to.

To connect the standard output from one program to the standard input of another program, you pipe them together by using the vertical bar symbol, |. Therefore, to connect the standard output of the program *stage1* to the standard input of the program *stage2*, you would type

stage1 | stage2

The operating system handles all the details of getting the output from *stage1* to the input of *stage2*.

Changing the Stream Buffer

All functions opened using the stream functions are buffered by default, except for the preopened streams **stdin**, **stdout**, **stderr**, **stdprn**, and **stdaux**. The two streams **stderr** and **stdaux** are unbuffered by default, unless they are used in one of the **printf** or **scanf** family of functions, in which case they are assigned a temporary buffer. The **stdin**, **stdout**, and **stdprn** streams are buffered; each buffer is flushed whenever it is full.

Closing Streams

The two functions **fclose** and **fcloseall** close a stream or streams. The **fclose** function closes a single file, while **fcloseall** closes all open streams except **stdin**, **stdout**, **stderr**, **stdprn**, and **stdaux**. However, if your program does not explicitly close a stream, the stream is automatically closed when the application terminates. Since the number of open streams is limited, it is good practice to close a stream when you finish with it.

Files can be accessed and created from C programs using the '**fopen**' routine. '**fopen**' returns a value of a special type called a **file pointer**. The pointer can be used in file version of **putchar** and **getchar** to write and read files.

Eg : The following program can be used to create a file from input :

```
# include <stdio.h>

void main( )
{
        int C ;
        char file[20] ;
        FILE  *outfile ;

        printf( " Input the Name of the file : \n" ) ;
        gets(file) ;
        outfile = fopen( file , "w" ) ;

        if ( outfile = NULL )
        {
                printf( " Cannot create the file %s \n" , file ) ;
                exit( ) ;
        }

        printf ( " Type input for file %s \n" , file ) ;
        while ( (  C = getchar ( ) ) != EOF  )
                putc( c , outfile ) ;

        fclose( outfile ) ;

}
```

Notice that '**fopen**' returns NULL if the file cannot be created. The "**w**" argument is called the mode of the '**fopen**'. The mode "**w**" means we want **fopen** to create a file to be written to.

Eg : The following program prompts for a filename and outputs the file to the screen. It repeats this until **^z** (**EOF**) is typed as the filename.

```
# include <stdio.h>

void main( )
{
        int C ;
        char file[20] ;
        FILE  *infile ;

        printf( " Input the Name of the file : \n" ) ;
        gets(file) ;
        infile = fopen( file , "r" ) ;

        if ( outfile = NULL )
        {
                printf( " Cannot open the file %s \n" , file ) ;
                exit( ) ;
        }

        while ( (  C = getc ( infile ) ) != EOF  )
                putchar( c ) ;

        fclose( infile ) ;
}
```

9.3 Text Files

A text file is a collection of ASCII characters, written in lines, with a specific and of line marker, and an end of file marker. Text files can be created using an editor, in the same way as programs are created. Alternatively, a text file can be created from within a program, by writing information to a file that has been opened in the appropriate mode.

9.4 Binary files

A binary file consists of a sequence of arbitrary bytes that are not in a readable-form. Such files can only be created by a specific program, they cannot, unlike a text file, be created using an editor.

9.5 Accessing a file Randomly

The library functions **fseek( )** , **ftell( )** and **rewind( )** are used to access a file randomly.

An expression of the form :

ftell(file_ptr) ;

returns the current value of the file position indicator. The value represents the number of bytes from the beginning of the file, counting from zero. Whenever a character is read from the file, the system increments the position indicator by 1.

The function **fseek( )** takes three arguments : a file pointer, an integer offset, and an integer that indicates the place in the file from which the offset should be computed. A statement of the form :

fseek(file_ptr , offset , place) ;

sets the file position indicator to a value that represent **offset** bytes from **place**. The value for **place** can be 0, 1 or 2, meaning the beginning of the file, current position or end of the file, respectively.

Eg : The following is a C program to write a file backwords.

```
# include < stdio.h >
# define  MAX  20

void main( )
{
        char  file_name[MAX] ;
        int  C ;
        FILE  *ifp ;

        printf( " Input the file name : \n" ) ;
        gets( file_name ) ;

        ifp = fopen( file_name , "rb" ) ;

        fseek( ifp , 0 , 2 ) ;
        fseek( ifp , -1 , 1 ) ;

        while( ftell( ifp ) >= 0 )
        {
                C = getc( ifp ) ;          /* move ahead one character */
                putchar( C ) ;
                fseek( ifp , -2 , 1 ) ;    /* back up two characters */
        }
}
```

The function **rewind** simply resets to the beginning of the file the file-position marker in the file pointed to by ***file_pointer***. The syntax for the function **rewind** looks like this :

rewind(file_pointer) ;

9.6 Predefined file pointers

Three file pointers are already defined when a C program is run. These are '**stdin**' which reads from the standard input stream, '**stdout**' which writes to the standard output stream, and '**stderr**' which writes to the standard error output stream. On Unix and MSDOS, these are attached to the controlling terminals unless redirected with '<' and '>'. The **putchar**, **puts** and **printf** write to the standard output.

There are more general versions of these functions which can read and write from any file.

Standard I/O routines	**File routines**
getchar( )	getc(ioptr)
gets(buf)	fgets(buf , size , ioptr)
putchar(c)	putc(c , ioptr)
puts(buf)	fputs(buf , ioptr)
printf(format , args)	fprintf(ioptr , format , args)

'**ioptr**' is a file pointer obtained using **fopen**.

Note that it is possible to use the file routines with the standard input and output.

Eg :

getc(stdin) is equivalent to getchar( ).

It is usual to design a C program so that the standard input and output can be redirected without changing the functionality of the program. In previous examples we used '**printf**' to output prompts to the user (eg: printf(" Type the input file name : \n") ;). However, when the output of such a program was redirected to a file, the prompts appeared in the file and not on the terminal. One way to avoid this is to send prompts to the standard error outputs (which is also the terminal). This stream is not redirected by '>'. Thus we could have :

```
fprintf( stderr , "Type the input file name : \n" ) ;
```

Eg : The following program prompts for two file names and copies the first file to the second.

```
# include < stdio.h >

void main( )
{
        char buf[200] , file[20] ;
        FILE  *in , *out ;

        fprintf( stderr , "Type the input file name : \n" ) ;

        if( ( in = fopen( file , "r" ) ) == NULL )
        {
                fprintf( stderr , " Cannot open file %s \n " , file ) ;
                exit( ) ;
```

```
        }

        fprintf( stderr , "Type the output file name : \n" ) ;

        if( ( out = fopen( file , "w" ) ) == NULL )
        {
                fprintf( stderr , " Cannot open file %s \n " , file ) ;
                exit( ) ;
        }

        while( fgets( buf , 200 , in ) != NULL )
                fputs( buf , out ) ;

        fclose( in ) ;
        fclose( out ) ;
}
```

feof( )

The ANSI file system can also operate on binary data. When a file is opened for binary input, it is possible that an integer value equal to the **EOF** mark be read. This would cause the previous routine to indicate an end-of-file condition even though the physical end of the file had not been reached. To solve this problem, ANSI C includes the function **feof( )**; which is used to determine the end of the file when reading binary data. The **feof( )** function has the prototype :

int feof(FILE *fp) ;

It returns true if the end of the file has been reached; otherwise, zero is returned. Therefore, the following routine reads a binary file until the end of the file is encountered.

```
while( !feof( fp )

        ch = getc( fp ) ;
```

Of course, this same method may be applied to text files as well as binary files.

ferror( )

The **ferror( )** function is used to determine if a file operation has produced an error. If a file is opened in text mode and an error in reading or writing occurs, **EOF** is returned. You use **ferror( )** to determine which event happened. The function **ferror( )** has the prototype :

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

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

fread( ) and fwrite( )

The ANSI I/O system provides two functions, called **fread( )** and **fwrite( )**, that allow the reading and writing of blocks of data. Their prototype are shown here :

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

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

In the case of **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 number of bytes to be read or written is specified by *num_bytes*. The argument *count* determine how many items (each being *num_bytes* in length) will be read or written. Finally, *fp* is a file pointer to a previously opened stream.

The **fread( )** function returns the number of items read, which can 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. As long as the file has been opened for binary data, **fread( )** and **fwrite( )** can read and write any type of information. One of the most useful applications of **fread( )** and **fwrite( )** involves reading and writing arrays (or, as you will see later, structures).

9.7 Passing arguments to C programs

C programs can access the arguments given on the shell command line, which runs the program. These arguments can be anything you like. They can be filenames, numbers, control flags starting with - , etc. Remember however that arguments starting > or < are not passed to your program but dealt with by the shell. All other arguments are passed to the main routine of the C program.

main( ) accesses these arguments using two variables **argc** and **argv**. The full declaration of a main routine is :

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

argc is the number of arguments and **argv** is an array of pointers to each argument. The 1st argument is always the name of the program.

Eg : The following program has two arguments both integers. It outputs the sum of the two integers.

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

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

```
{
        int  i , sum = 0 ;

        if ( argc != 3 )
        {
                fprintf( stderr , “Error - wrong no of arguments \n” ) ;
                exit( ) ;
        }

        for( i = 1 ; i < argc ; i++ )
                sum = sum + atoi( argv[i] ) ;
        printf(“ %d \n” , sum ) ;
}
```

Eg : The next program is a version of ‘**copy**’ which has the input and output filenames as its arguments. If the input filename is just ‘**-**’ the standard input is used. If the output file name is missing the standard output is used.

```
# include < stdio.h >

void main( argc , argv )
int  argc ;
char  *argv[ ] ;
{

        if ( argc < 2  || argc > 3 )
        {
                fprintf( stderr , “Error - wrong no of arguments \n” ) ;
                exit( ) ;
        }

        if ( strcmp( argv[1] , “-” ) != 0 )
        {
                if ( freopen( argv[1] , “r” , stdin ) == NULL )
                {
                        fprintf( stderr , “Cannot open file %s \n ” , file ) ;
                        exit( ) ;
                }
        }

        if ( argc == 3 )
        {
                if( freopen( argv[2] , “w” , stdout ) == NULL )
                {
                        fprintf( stderr , “Cannot create file %s \n ” , file ) ;
                        exit( ) ;
                }
        }
```

```
        copy( ) ;

}


void copy( )
{
        int C ;
        while( ( C = getchar( ) ) != EOF )
                putchar(C) ;
}
```