UNIT- V
Files

Introduction

Files are used to store a large amount of data in a disk. Handling of large volume of data through keyboards has the following disadvantages:

1. It is time consuming.
2. The entire data is lost when either the program is terminated or when the power is turned off.

A file is a group of related data, stored on the disk.

Defining and Opening A File

Before using any files, it must be opened properly. Opening a file establishes a link between the program and the operating system, about, which file we are going to access and how. The link between our program and the operating system is structure called FILE, which has been defined, in the header file "stdio.h" (standard Input Output header file). Also the purpose of opening the above file should also be defined. The purpose may be either reading, writing or appending data.

The function fopen( ) is used for opening a file. The general format for defining and opening a file is as follows.

```
FILE *ptr;

ptr = fopen("filename", "mode");
```

Where FILE is a defined data type. Data structure of a file is defined as FILE. It is not necessary to define this structure. The FILE is a type of structure, which has been defined in the header file stdio.h. Each file will have its own FILE structure. The FILE structure contains information about the file being used, such as its current size, its location in memory etc . So all the files should be declared as type FILE before they are used. This structure is defined only in uppercase letters. Do not use file or File instead of FILE.

ptr is a pointer variable containing the address of the structure FILE. ptr is a pointer to the data type FILE.

The second statement opens the file indicated as filename and assigns an identifier to the FILE type pointer ptr. "mode" part is used to indicate the purpose for which the file is being opened. The value of "mode" can be any one of the following.

Mode	Purpose	Action
"r"	Read from the file	This mode searches the disk for the filename. If it exists, then it is loaded from disk into the memory and a pointer returned to the file; otherwise, an error occurs and returns NULL value. NULL indicates that there will be a failure in opening the file.
"w"	Write data into the file.	A file with a specified name is created, if the file does not exist. If the file already exists, its contents are deleted. If it is unable to open the file, it returns NULL.
"a"	Append data to the file.	If the file already exists, it is opened so that data may be added. If the file does not exist, it is created. If It is unable to open the file, it returns NULL.

r+	-	Opens a text file for reading and writing both
w+	-	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
+	-	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing a File

A file must be closed immediately after finishing all the operations on the above file. The purposes of closing a file is as follows:

1. To flush the information from the buffers of the memory.
2. To break all the links to the file.
3. To prevent the accidental movement of the file.
4. To reopen the same file in a different mode.

The general format for closing a file is

```
fclose (ptrf) ;
```

The above statement close the file associated with the FILE pointer (ptrf). Once a file is closed, its file pointer can be reused for another file.

Example

```
FILE ptrf1, ptrf2;
ptrf1 = fopen("exam.dat", "r");
ptrf1 = fopen("result.dat", "w");
--------------------
fclose (ptrf1) ;
fclose (ptrf2) ;
```

The above program opens two files and closes them after all operations are completed.

Input / Output Operations on Files

Reading from a file (getc( ) function)

Once the file has been opened for reading using fopen( ), the file's contents are brought into memory (partly or wholly) and a pointer points to the very first character. To read the file's contents from memory there exists a standard library function called getc( ). getc( ) function is used to read a character at a time from the file. When a file is opened, the file pointer provides access to the first character written on the file.

The functions getc( ) reads one character from current pointer position,. This character is then assigned to variable and advances the pointer position so that it points to the next character. Once the file has been opened, it is referred by file pointer ond file name is no longer valid.

getc( ) function stops the reading process when the end of file is reached. The end of file can be checked by checking for EOF.

For example, the statement

```
ch = getc(fp1);
```

read a character from the file whose file pointer is fpl. The above character is stored in the variable ch. getc( ) and fgetc( ) are both same.

Writing into a file (putc( ) function)

putc( ) function is used to write a character into a file. The statement

putc(c, ptrf);

write the character contained in the character variable c to the file associated with FILE pointer ptrf.

The end of file is marked by EOF. This EOF can be placed at end of file through ^ z(ctrl + z). putc( ) and fputc( ) are same.

Formatted Functions (fprintf() and fscanf())

getc( ), putc( ), getw( ) and putw( ) function can handle one character or integer at a time. But the functions, fprintf ( ) and fscanf ( ) can handle a group of data simultaneously. These two functions perform input/output operations that are identical to printf and scanf functions. But these two functions work on files.

The parameters to these functions are similar to printf and scanf. But one more additional parameter, FILE pointer specifying the file, is added to these functions. The first argument of these functions is a file pointer, which specifies the file to be used.

fprintf( )

The general format of fprintf statement is

```
fprintf( ptrf, "control_string", list);
```

where ptrf is the file pointer for the file that has been opened for writing. The control string specifies the output format for the various items in the list.

The list may contain variables, constants or strings.

Examples

fprintf(ptrf1, "%s %d %d ", name , age , salary);

fprintf(ptrf1, "%2f %d %d ", price , quantity, product_name);

fscanf( )

fscanf reads characters from the specified file, converts them as per directions given in control string format. Then the converted value is assigned to the objects pointed by the list of arguments. The general form is

```
fscanf( ptrf, "control_string", list);
```

Error Handling During I/O Operations

An error may occur during input / output operations on a file. Some of the situations in which the error occurs are as follows.

1. When reading the data beyond the EOF mark.
2. Device overflow (i.e., no space in disk)
3. Not opening of files.
4. Trying to perform an operation on a file, when the file is opened for another type of operation. For example it is not possible to perform read operation, when the file is opened for write operation.
5. Opening a file with an invalid file name.

Two library functions feof and ferror is used to detect I/O errors in the files.

feof( )

This function is used to test for an end of file condition. This function has only one argument. This argument is a FILE pointer. This function returns a non-zero integer value if all the data from the specified file has been read. Otherwise it returns zero. For example, the statements

if(feof(ptrf));

printf (" End of file");

displays the message "End of file", on reaching the end of file condition.

ferror( )

This function is used to report the status of the file specified. This function also takes only one argument, FILE pointer. This function returns a non-zero integer if an error has been detected upto that point during processing. Otherwise, it returns zero. For example, the statements

```
if(ferror(ptrf)!=0);
printf ( " There will be an error");
```

display the message "There will be error", if the reading is not successful.

Preprocessor

Introduction

Preprocessor processes the source code before the compilation begins. The preprocessor is invoked automatically by the C compiler.

In C program, commands or instructions to the C compiler can be included. These are called preprocessor directives. Every preprocessor directive must begin with the character #. Preprocessor directive do not require a semicolon at the end. Preprocessor directives are placed in the source program before the main function.

Preprocessor directives can be subdivided into

- Macro definition and substitution
- File inclusion.
- Compiler controlled directives.

Macro Definition Directives

In macro substitution, an identifier is replaced by a string. The preprocessor accomplishes this task with the help of define statement. The general form of define statement is

```
# define macro_name macro
bodylist);
```

The above statement is used to replace the macro_name wherever it occurs in the program by the string of the macro_body. Macro_name is generally written in uppercase letters.

The #define directive defines an identifier and a string/constant that is substituted in place of the identifier name each time it is encountered in the file. This identifier is called macroname and the replacement process is known as macro substitution.

There is no need of a semicolon after the statement having a #define directive.

Advantages of of #define statement :

1. It helps in generation of faster and compact code.
2. It saves the programmer's time.
3. It reduces the chances of inconsistency within the program.
4. It makes the modification easier, as the value has to be changed only at one place in the program i.e., it increases the flexibility of the program

Macro Substitutions

Simple macro substitution is commonly used to define constants

Examples

```
# define PI 3.14
# define PLACE " SALEM"
# define SUM 0
```

Expressions can also be included in macro definition

For example,

```
# define AREA 10 * 25.5
# define THREE _PI 3 * 3.1
```

are also valid . Following examples are also the correct macro definitions.

```
# define EQUALS ==
# define AND &&
# define OR ||
# define NOT_EQUAL !=
# define INCREMENT ++
```

Then the following statement

if (age EQUALS 25 AND height NOT_EQUAL 50) INCREMENT count; is also valid.

Macro with arguments

Macros with arguments takes the following form

```
# define variable name(a1,a2.....an), string
```

There should not be any space between variable name and the left parentheses. The arguments a1, a2, are formal macro arguments.

Example

```
#define square (A) (A*A)
```

if the following statement appears late in the program

```
AREA = square(side);
```

Then the above statement is equivalent to : AREA = side * side;

Example

```
# define MAX (X,Y)(((X) >(Y))?(X:Y))
# define MIN(A,B)(((A)<(B)) ? (A):(B))
# define ABS(X) (((X)>0) ? (X)-X)))
```

Undefining a Macro

undef directive is used to undefine a defined directive. The statement

```
# undef macro_name
```

cause the previous preprocessor directive with this macro_name to lapse. From this point onwards, the previous definition will not hold good. undef is useful when restricting the definition only to a particular part of the program,

Major advantage of using macro is to increase the speed of the execution of the program.

Disadvantages of Macros

- No type checking is performed in macro. This may cause error,
- A macro call may cause unexpected results.

File Inclusion

External files containing function or macro definitions can be inserted into the program by the preprocessor directive include. The general format for include directive is

```
# include “filename”, (or) # include <filename>
```

Where filename is the name of the file to be inserted. The preprocessor inserts the entire contents into the source code of the program.

When the file name is within the double quotation marks, the file is first searched in the current directory and then in the standard directories. When the file name is within a pair of angle brackets, the file is searched only in the standard directories.

Conditional Compilation

Compiling the selected portion of the program for a particular condition is called “conditional compilation”. Conditional compilation directives are (i) ifdef (ii) ifndef (iii) #if #else #endif directive

ifdef

The #ifdef feature is used to remove sections of codes automatically. For example,

```
# ifdef LABEL
{
```

```
Block _ 1
}
# else
{
Block _2
}
```

 # endif

Block 1 will get compiled only if LABEL has been # defined. If LABEL has not been defined as a macro, the Block1 won't be sent for compilation at all. Block 2 will get compiled.

For example, assume a single program can be compiled by two different makes of computer. The following single program conditionally compiles only the code pertaining to either of the two machines.

```
main()
{
# ifdef MACINTOSH
{
code for MACINTOSH
}
# else
{
code for IBM
}
# endif
}
```

ifdef means "if defined". The above program would run smoothly on a MACINTOSH as well as IBM. If MACINTOSH has been defined, then the above program will run on MACINTOSH computer. Otherwise it will run on IBM computer. For running the above program in Macintosh, the directive # define MACINTOSH may be included.

ifndef directive

ifndef means "if not defined". This directive works exactly opposite to the directive #ifdef. The general form of this directive is

```
#ifndef macro_name
statement 1;
#else
statement2;
#endif
```

If there is no #define for macro_name, then statement1 will be executed otherwise statement2.

#if #else #endif directive

#if #else #endif directive tests an expression for non zero value and performs the conditional compilation, The general form of this is:

```
# if constant-expression
Statement;
#else
Statement2;
```