UNIT-V

Unit-5	
File	A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure. In C language, we use a structure pointer of file type to declare a file.
Streams	In C, the stream is a common, logical interface to the various devices that comprise the computer. In its most common form, a stream is a logical interface to a file
Formatted I/O	C provides standard functions scanf() and printf(), for performing formatted input and output. These functions accept, as parameters, a format specification string and a list of variables.
Preprocessor Directives	The C preprocessor is a macro preprocessor that transforms your program before it is compiled.
Printf	In C programming language, printf() function is used to print the "character, string, float, integer, octal and hexadecimal values" onto the output screen.
Scanf	The scanf function allows you to accept input from standard in, which for us is generally the keyboard.

CONSOLE I/O

The C language does not define any keywords that perform I/O. Instead, input and output are accomplished through library functions. C's I/O system is an elegant piece of engineering that offers a flexible yet cohesive mechanism for transferring data between devices. C's I/O system is, however, quite large, and consists of several different functions.

The header for the I/O functions is **<stdio.h>** . There are both console and file I/O functions. Technically, there is little distinction between console I/O and file I/O. But conceptually they are in very different worlds.

This section examines in detail the console I/O functions.

The next section presents the file I/O system and describes how the two systems

relate.

Reading and Writing Characters

- The simplest of the console I/O functions are **getchar( )** , which reads a character from the keyboard, and **putchar( )**, which writes a character to the screen.
- The **getchar( )** function waits until a key is pressed and then returns its value.
- The keypress is also automatically echoed to the screen.
- The **putchar( )** function writes a character to the screen at the current cursor

position. The prototypes for **getchar( )** and **putchar( )** are shown here:

```
int  getchar(void);
int putchar(int c);
```

Example Program

```
#include <stdio.h>

#include<conio.h>

void main( )

{

  char ch;

  printf( "Enter a character :");

  ch = getchar( );

  printf( "\nYou entered: ");


  putchar(ch);

  getch();

}
```

Expected Output

Enter a character:A

You entered:

A

Reading and Writing Strings

The next step up in console I/O, in terms of complexity and power, are the functions **gets( )** and **puts( )**. They enable you to read and write strings of characters.

The **gets( )** function reads a string of characters entered at the keyboard and stores them at the address pointed to by its argument. You can type characters at the keyboard until you strike a carriage return. The carriage return does not become part of the string; instead, a null terminator is placed at the end, and **gets( )** returns. In fact, you cannot use **gets( )** to return a carriage return (although **getchar( )** can do so). You can correct typing mistakes by using the backspace key before pressing ENTER.

The usage of functions **gets( )** and its counterpart **puts( )** is shown below.

Example Program

```
//puts and gets

void main()

{

char name[25] ;

clrscr();

printf ( "Enter your full name: " ) ;

gets ( name ) ;      // accepts the string

puts ( name ) ;      // displays the string

getch();

}
```

Expected Output

Enter your full name: Ayesha Shariff

Ayesha Shariff

- **puts( )** can display only one string at a time (hence the use of two **puts( )** in the program above). Also, on displaying a string, unlike **printf( )**, **puts( )** places the cursor on the next line.
- Though **gets( )** is capable of receiving only one string at a time, the plus point with **gets( )** is that it can receive a multi-word string.

Formatted Console I/O

- The functions **printf( )** and **scanf( )** perform formatted output and input— that is, they can read and write data in various formats that are under your control.
- The **printf( )** function writes data to the console.
- The **scanf( )** function, its complement, reads data from the keyboard.

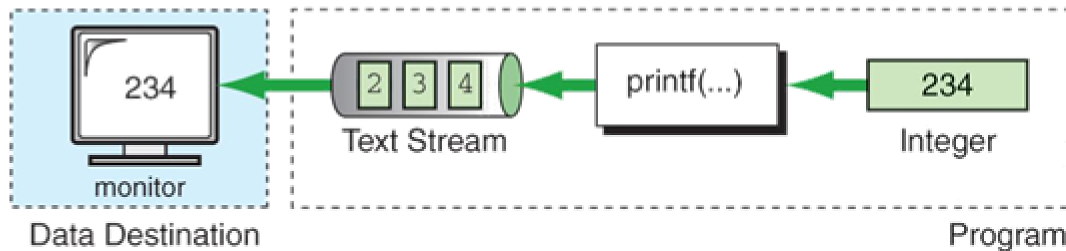
- Both functions can operate on any of the built-in data types, plus null-terminated character strings.

The following are the two types of formatted I/O functions

(a) printf()
(b) scanf()

(a) printf()

This function is used to print result on the monitor.



Syntax

printf("control string",arg1,arg2,….,argn);

Examples

printf("%d",a);

printf("%d%c",num,ch);

Printing Characters

- To print an individual character, use **%c**.
- To print a string, use **%s**.

Printing Numbers

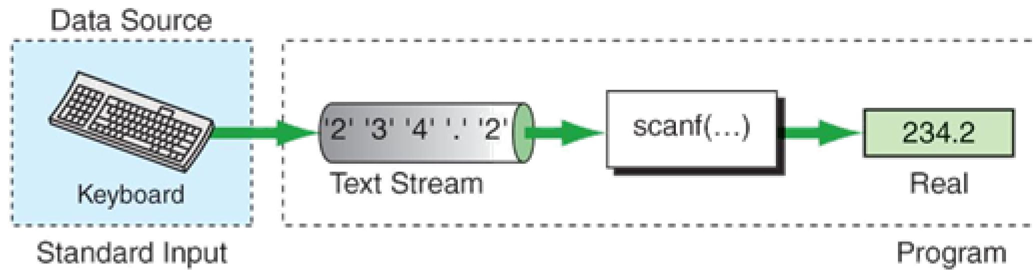
- You can use either **%d** or **%i** to display a signed integer in decimal format.
- These format specifiers are equivalent; both are supported for historical reasons, of which one is the desire to maintain an equivalence relationship with the **scanf( )** format specifiers.
- To output an unsigned integer, use **%u**.
- The **%f** format specifier displays numbers in floating point.
- The matching argument must be of type **double**.

Displaying an Address

- If you want to display an address, use **%p**.
- This format specifier causes **printf( )** to display a machine address in a format compatible with the type of addressing used by the computer.

(b) scanf()

This function is used to read values for variables from keyboard.



Syntax

scanf("control string",address_list);

where control string specifies the type of values that have to read from the keyboard.
Examples

scanf("%d",&a);

scanf("%d%f",&a,&b);

scanf("%d%f%c",&a,&b,&c);

Standard C vs. Unix File I/O

C was originally implemented for the Unix operating system. As such, early versions of C (and many still today) support a set of I/O functions that are compatible with Unix. This set of I/O functions is sometimes referred to as the *Unix-like I/O system*, or the *unbuffered I/O system*. However, when C was standardized, the Unix-like functions were not incorporated into the standard, largely because they are redundant. Also, the Unix-like system may not be relevant to certain environments that could otherwise support C.

This chapter discusses only those I/O functions that are defined by Standard C. In previous editions of this work, the Unix-like file system was given a small amount of coverage. In the time that has elapsed since the previous edition, use of the standard I/O functions has steadily risen and use of the Unix-like functions has steadily decreased. Today, most programmers use the standard functions because they are portable to all environments (and to C++). Programmers wanting to use the Unix like functions should consult their compiler's documentation.

Streams and Files

Before beginning our discussion of the C file system it is necessary to know the difference between the terms *streams* and *files*. The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed. That is, the C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a *stream*, and the actual device is called a *file*. It is important to understand how streams and files interact.

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. Even though each device is very different, the buffered file system transforms each into a logical device called a stream. All streams behave similarly. Because streams are largely device independent, the same function that can write to a disk file can also write to another type of device, such as the console.

There are two types of streams: text and binary.

(a) Text Streams

A *text stream* is a sequence of characters. Standard C states that a text stream is organized into lines terminated by a newline character. However, the newline character is optional on the last line. In a text stream, certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those stored on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as the number that is stored on the external device.

(b) Binary Streams

A *binary stream* is a sequence of bytes that has a one-to-one correspondence to the bytes in the external device— that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device. However, an implementation— defined number of null bytes may be appended to a binary stream. These null bytes might be used to pad the information so that it fills a sector on a disk, for example.

Files

In C, a *file* may be anything from a disk file to a terminal or printer. You associate a stream with a specific file by performing an *open* operation. Once a file is open, information can be exchanged between it and your program. Not all files have the same capabilities. For example, a disk file can support random access, while some printers cannot. This brings up an important point about the C I/O system: All streams are the same, but all files are not.

If the file can support *position requests*, opening that file also initializes the *file position indicator* to the start of the file. As each character is read from or written to the file, the position indicator is incremented, ensuring progression through the file.

You disassociate a file from a specific stream with a *close* operation. If you close a file opened for output, the contents, if any, of its associated stream are written to the external device. This process, generally referred to as *flushing* the stream, guarantees that no information is accidentally left in the disk buffer. All files are closed automatically when your program terminates normally, either by **main( )** returning to the operating system or by a call to **exit( )**.

Files are not closed when a program terminates abnormally, such as when it crashes or when it calls **abort( )**. Each stream that is associated with a file has a file control structure of type **FILE**. Never modify this file control block.

If you are new to programming, the separation of streams and files may seem unnecessary or contrived. Just remember that its main purpose is to provide a consistent interface. You need only think in terms of streams and use only one file system to accomplish all I/O operations. The I/O system automatically converts the raw input or output from each device into an easily managed stream.

File System Basics

- The C file system is composed of several interrelated functions.
- The most common of these are shown in Table 9-1.
- They require the header **<stdio.h>** .
- The header **<stdio.h>** provides the prototypes for the I/O functions and defines these three types: **size_t**, **fpos_t**, and **FILE**.

Terminates normally, either by **main( )** returning to the operating system or by a call to **exit( )**. Files are not closed when a program terminates abnormally, such as when it crashes or when it calls **abort( )**. Each stream that is associated with a file has a file control structure of type **FILE**. Never modify this file control block.

If you are new to programming, the separation of streams and files may seem unnecessary or contrived. Just remember that its main purpose is to provide a consistent interface. You need only think in terms of streams and use only one file system to accomplish all I/O operations. The I/O system automatically converts the raw input or output from each device into an easily managed stream.

File System Basics

- The C file system is composed of several interrelated functions.
- The most common of these are shown in Table 9-1.
- They require the header **<stdio.h>** .
- The header **<stdio.h>** provides the prototypes for the I/O functions and defines these three types: **size_t**, **fpos_t**, and **FILE**.
- The **size_t** type is some variety of unsigned integer, as is **fpos_t**.
- The **FILE** type is discussed in the next section.

The File Pointer

- The file pointer is the common thread that unites the C I/O system.
- A *file pointer* is a pointer to a structure of type **FILE**. It points to information that defines various things about the file, including its name, status, and the current position of the file.
- In essence, the file pointer identifies a specific file and is used by the associated stream to direct the operation of the I/O functions.

- In order to read or write files, your program needs to use file pointers.

To obtain a file pointer variable, use a statement like this:

FILE *fp;

Opening a File

- The **fopen( )** function opens a stream for use and links a file with that stream.
- Then it returns the file pointer associated with that file.
- Most often (and for the rest of this discussion), the file is a disk file.

The **fopen( )** function has this prototype,

FILE *fopen(const char **filename*, const char **mode*);

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

Table 9-1. *Commonly Used C File-System Functions*

- where *filename* is a pointer to a string of characters that make up a valid filename and may include a path specification.
- The string pointed to by *mode* determines how the file will be opened.
- Table 9-2 shows the legal values for *mode*.
- Strings like "r+b" may also be represented as "rb+".
- As stated, the **fopen( )** function returns a file pointer.
- Your program should never alter the value of this pointer.
- If an error occurs when it is trying to open the file, **fopen( )** returns a null pointer.

The following code uses **fopen( )** to open a file named TEST for output.

```
FILE *fp;

fp = fopen("test", "w");
```

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
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

Table 9-2. *Legal Values for Mode*

Although the preceding code is technically correct, you will usually see it written like this:

```
FILE *fp;

if ((fp = fopen("test","w"))==NULL) {

printf("Cannot open file.\n");

exit(1);

}
```

This method will detect any error in opening a file, such as a write-protected or a full disk, before your program attempts to write to it. In general, you will always want to confirm that **fopen( )** succeeded before attempting any other operations on the file.

Closing a File

The **fclose( )** function closes a stream that was opened by a call to **fopen( )**. It writes any data still remaining in the disk buffer to the file and does a formal operating-system-level close on the file. Failure to close a stream invites all kinds of trouble, including lost data, destroyed files, and possible intermittent errors in your program. **fclose( )** also frees the file control block associated with the stream, making it available for reuse. Since there is a limit to

the number of files you can have open at any one time, you may have to close one file before opening another.

The **fclose( )** function has this prototype,

int fclose(FILE **fp*);

where *fp* is the file pointer returned by the call to **fopen( )**. A return value of zero signifies a successful close operation. The function returns **EOF** if an error occurs. You can use the standard function **ferror( )** (discussed shortly) to determine the precise cause of the problem. Generally, **fclose( )** will fail only when a disk has been prematurely removed from the drive or there is no more space on the disk.

Example Programs

(i) To Create a new File

```
#include<stdio.h>
void main()

{

FILE *fp;
char str[80];
clrscr();

fp=fopen("file1.txt","w");

if(fp==NULL)

printf("\nFile Not Exist!");

printf("\nEnter the text\n");

        while(strlen(gets(str))>0)

        {

        fputs(str,fp);

        fputs("\n",fp);

        }

fclose(fp);

getch();

}
```

(ii) To Display the contents of a File

```
#include<stdio.h>
void main()
```

```
{
FILE *fp;
char str[80];
clrscr();
fp=fopen("file1.txt","r");
        if(fp==NULL)
        printf("\nFile Not Exist!");
        while(fgets(str,79,fp)!=NULL)
        printf("%s",str);
fclose(fp);
getch();
}
```

fread( ) and fwrite( )

To read and write data types that are longer than 1 byte, the C file system provides two functions: **fread( )** and **fwrite**( ). These functions allow the reading and writing of blocks of any type of data.

Their prototypes are

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

size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *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. (Remember, the type **size_t** is defined as some kind of unsigned integer.) 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.

Using fread( ) and fwrite( )

As long as the file has been opened for binary data, **fread( )** and **fwrite( )** can read and write any type of information.

For example, the following program writes and then reads back a **double**, an **int**, and a **long** to and from a disk file. Notice how it uses **sizeof** to determine the length of each data type.

Example Program

```
/* Write some non-character data to a disk file and read it back. */
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE *fp;
double d = 12.23;
int i = 101;
long 1 = 123023L;
if((fp=fopen("test", "wb+"))==NULL) {
```

fread( ) and fwrite( )

To read and write data types that are longer than 1 byte, the C file system provides two functions: **fread( )** and **fwrite**( ). These functions allow the reading and writing of blocks of any type of data.

Their prototypes are

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

size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *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. (Remember, the type **size_t** is defined as some kind of unsigned integer.) 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.

Using fread( ) and fwrite( )

As long as the file has been opened for binary data, **fread( )** and **fwrite( )** can read and write any type of information.

For example, the following program writes and then reads back a **double**, an **int**, and a **long** to and from a disk file. Notice how it uses **sizeof** to determine the length of each data type.

Example Program

```
/* Write some non-character data to a disk file and read it back. */

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE *fp;

double d = 12.23;

int i = 101;

long 1 = 123023L;

if((fp=fopen("test", "wb+"))==NULL) {

return 0;

}
```

fseek( ) and Random-Access I/O

You can perform random read and write operations using the C I/O system with the help of **fseek( )**, which sets the file position indicator. Its prototype is shown here:

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

Here, *fp* is a file pointer returned by a call to **fopen( )**, *numbytes* is the number of bytes from *origin*, which will become the new current position, and *origin* is one of the following macros:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Therefore, to seek *numbytes* from the start of the file, *origin* should be **SEEK_SET**. To seek from the current position, use **SEEK_CUR**, and to seek from the end of the file, use **SEEK_END**. The **fseek( )** function returns zero when successful and a nonzero value if an error occurs.

The following program illustrates **fseek( )**. It seeks to and displays the specified byte in the specified file. Specify the filename and then the byte to seek to on the command line.

```
#include <stdio.h>

#include <stdlib.h>

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

{

FILE *fp;

if(argc!=3) {

printf("Usage: SEEK filename byte\n");

exit(1);

}

if((fp = fopen(argv[1], "rb"))==NULL) {

printf("Cannot open file.\n");

exit(1);

}

if(fseek(fp, atol(argv[2]), SEEK_SET)) {

printf("Seek error.\n");

exit(1);

}

printf("Byte at %ld is %c.\n", atol(argv[2]), getc(fp));

fclose(fp);

return 0;

}
```

fprintf( ) and fscanf( )

In addition to the basic I/O functions already discussed, the C I/O system includes **fprintf( )** and **fscanf( )**.

These functions behave exactly like **printf( )** and **scanf( )** except that they operate
with files.

The prototypes of **fprintf( )** and **fscanf( )** are

int fprintf(FILE **fp*, const char **control_string*, . . .);
int fscanf(FILE **fp*, const char **control_string*, . . .);

where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

As an example, the following program reads a string and an integer from the keyboard and writes them to a disk file called TEST. The program then reads the file and displays the information on the screen. After running this program, examine the TEST file. As you will see, it contains human readable text.

Example Program

```
/* fscanf() - fprintf() example */

#include <stdio.h>

#include <io.h>

#include <stdlib.h>

int main(void)

{

FILE *fp;

char s[80];

int t;

if((fp=fopen("test", "w")) == NULL) {

printf("Cannot open file.\n");

exit(1);

}

printf("Enter a string and a number: ");

fscanf(stdin, "%s%d", s, &t); /* read from keyboard
*/ fprintf(fp, "%s %d", s, t); /* write to file */
fclose(fp);
```

```
if((fp=fopen("test","r")) == NULL) {
printf("Cannot open file.\n");
exit(1);
}
fscanf(fp, "%s%d", s, &t); /* read from file */
fprintf(stdout, "%s %d", s, t); /* print on screen */
return 0;
}
```

A word of warning: Although **fprintf( )** and **fscanf( )** often are the easiest way to write and read assorted data to disk files, they are not always the most efficient. So, if speed or file size is a concern, you should probably use **fread( )** and **fwrite( )**.

The Standard Streams

- As it relates to the C file system, when a program starts execution, three streams are opened automatically.
- They are **stdin** (standard input), **stdout** (standard output), and **stderr** (standard error).
- Normally, these streams refer to the console, but they can be redirected by the operating system to some other device in environments that support redirectable I/O. (Redirectable I/O is supported by Windows, DOS, Unix, and OS/2, for example.)
- Because the standard streams are file pointers, they may be used by the C I/O system to perform I/O operations on the console.

For example, **putchar( )** could be defined like this:

```
int putchar(char c)
{
return putc(c, stdout);
}
```

- In general, **stdin** is used to read from the console, and **stdout** and **stderr** are used to write to the console.
- You can use **stdin**, **stdout**, and **stderr** as file pointers in any function that uses a variable of type **FILE ***.

For example, you could use **fgets( )** to input a string from the console using a call like this:

```
char str[255];
fgets(str, 80, stdin);
```

The Preprocessor Directives #define and #include.

You can include various instructions to the compiler in the source code of a C program. These are called *preprocessor directives*, and they expand the scope of the programming environment.

(i) #define

The **#define** directive defines an identifier and a character sequence (a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*.

The general form of the directive is

#define *macro-name char-sequence*

Notice that there is no semicolon in this statement. There may be any number of spaces between the identifier and the character sequence, but once the character sequence begins, it is terminated only by a newline.

Example

#define UPPER 25

This statement is called „macro definition" or more commonly, just a „macro". What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25.

Example Program

```
#define UPPER 25
main( )
{
int i ;
for ( i = 1 ; i <= UPPER ; i++ )
printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement.

(ii) #include

The second preprocessor directive is file inclusion. This directive causes one file to be included in another.

The preprocessor command for file inclusion looks like this:

#include "filename" and it simply causes the entire contents of filename to be inserted into the source code at that point in the program.

It is common for the files that are to be included to have a **.h extension**. This extension stands for „header file", possibly because it contains statements which when included go to the head of your program. The prototypes of all the library functions are grouped into different categories and then stored in different header files. For example prototypes of all mathematics related functions are stored in the header file „math.h", prototypes of console input/output functions are stored in the header file „conio.h", and so on.

Actually there exist two ways to write **#include** statement. These are:

#include "filename"

#include <filename>

The meaning of each of these forms is given below:

#include "goto.c"	*This command would look for the file* ***goto.c*** *in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.*
#include <goto.c>	*This command would look for the file* ***goto.c*** *in the specified list of directories only.*

Assignment Questions

Unit –V

1. Explain in detail about reading and writing characters in C with example program.
2. Describe in detail about printf() and scanf() functions.
3. Define String? What are the various string manipulation functions available in C? Explain.
4. Define Preprocessor directives. Discuss Marco replacement with an example.
5. How to use fseek() for random access of the file content?
6. Describe various types of files and operations on files with an example.
7. What are C preprocessor directives? Explain various types.