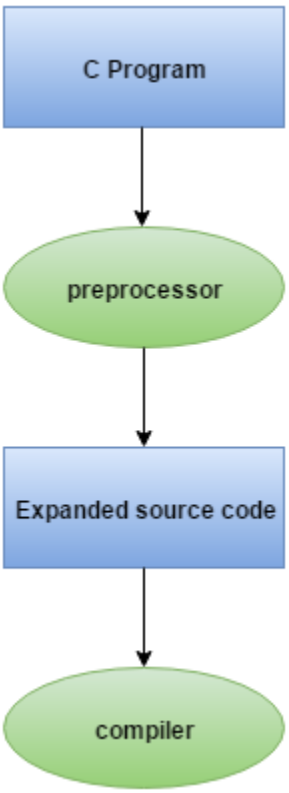
C Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

> ***Note: Proprocessor direcives are executed before compilation.***

All preprocessor directives starts with hash # symbol.

Features of C Preprocessor



There are several steps involved from the stage of writing a C program to the stage of getting it executed. Figure 7.1 shows these different steps along with the files created during each stage. You can observe from the figure that our program passes through several processors before it is ready to be executed. The input and output to each of these processors is shown in Figure 7.2.

Note that if the source code is stored in a file PR1.C then the expanded source code gets stored in a file PR1.I. When this expanded source code is compiled the object code gets stored in PR1.OBJ. When this object code is linked with the object code of library functions the resultant executable code gets stored in PR1.EXE.

The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:

(a) Macro expansion

(b) File inclusion

(c) Conditional Compilation

(d) Miscellaneous directives

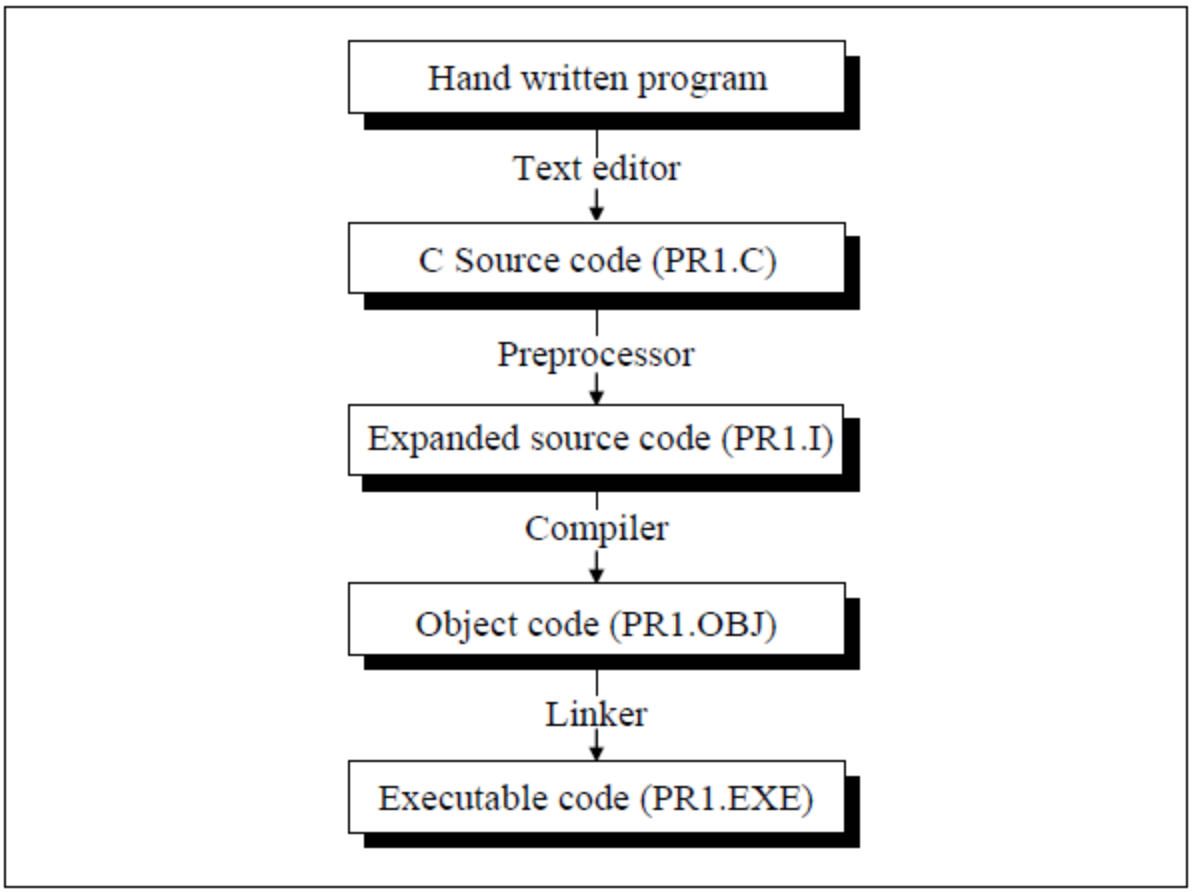


Figure 7.1

Processor	Input	Output
Editor	Program typed from keyboard	C source code containing program and preprocessor commands
Prepro-cessor	C source code file	Source code file with the preprocessing commands properly sorted out
Compiler	Source code file with preprocessing commands sorted out	Relocatable object code
Linker	Relocatable object code and the standard C library functions	Executable code in machine language

Figure 7.2

C Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.14
```

Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name.

Macro Expansion

Have a look at the following 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,

```
#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.

Here is another example of macro definition.

```
#define PI 3.1415
main( )
{
float r = 6.25 ;
float area ;
area = PI * r * r ;
```

```
printf ( "\nArea of circle = %f", area ) ;
}
```

UPPER and PI in the above programs are often called ‘macro templates’, whereas, 25 and 3.1415 are called their corresponding ‘macro expansions’.

When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.

A #define directive is many a times used to define operators as shown below.

```
#define AND &&
#define OR ||
main( )
{
int  f = 1, x = 4, y = 90 ;
if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
printf ( "\nYour PC will always work fine..." ) ;
else
printf ( "\nIn front of the maintenance man" ) ;
}
```

A #define directive could be used even to replace a condition, as shown below.

```
#define AND &&
#define ARANGE ( a > 25 AND a < 50 )
main( )
{
int a = 30 ;
if ( ARANGE )
printf ( "within range" ) ;
else
printf ( "out of range" ) ;
}
```

Macros with Arguments (OR) Function like Macros

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
```

```
float r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f", a ) ;
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **(3.14 * x * x)**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **(3.14 * x * x)**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**. Thus the statement **AREA(r1)** is equivalent to:(3.14 * r1 * r1)

Let's see a list of preprocessor directives.

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #if
- #else
- #elif
- #endif
- #error
- #pragma

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

No.	Macro	Description
1	_DATE_	represents current date in "MMM DD YYYY" format.
2	_TIME_	represents current time in "HH:MM:SS" format.
3	_FILE_	represents current file name.
4	_LINE_	represents current line number.
5	_STDC_	It is defined as 1 when compiler complies with the ANSI standard.

C predefined macros example

File: simple.c

```
#include <stdio.h>
main() {
  printf("File :%s\n", __FILE__ );
  printf("Date :%s\n", __DATE__ );
  printf("Time :%s\n", __TIME__ );
  printf("Line :%d\n", __LINE__ );
  printf("STDC :%d\n", __STDC__ );
}
```

Output:

```
File :simple.c
Date :Dec 6 2015
Time :12:28:46
Line :6
STDC :1
```

C #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

1. #include <filename>
2. #include "filename"

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

```
#include <stdio.h>
main() {
  printf("Hello C");
}
```

Output:

```
Hello C
```

#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

C #define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

```
#define token value
```

Let's see an example of #define to define a constant.

```
#include <stdio.h>
#define PI 3.14
main() {
  printf("%f",PI);
}
```

Output:

```
3.140000
```

Let's see an example of #define to create a macro.

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main() {
  printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:

```
Minimum between 10 and 20 is: 10
```

C #undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

```
#undef token
```

Let's see a simple example to define and undefine a constant.

```
#include <stdio.h>
#define PI 3.14
#undef PI
main() {
printf("%f",PI);
}
```

Output:

```
Compile Time Error: 'PI' undeclared
```

The #undef directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

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

```
#define number 15
int square=number*number;
#undef number
main() {
  printf("%d",square);
}
```

Output:

```
225
```

C #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifdef MACRO
//code
#endif
```

Syntax with #else:

```
#ifdef MACRO
//successful code
#else
//else code
#endif
```

C #ifdef example

Let's see a simple example to use #ifdef preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define NOINPUT
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Value of a: 2
```

But, if you don't define NOINPUT, it will ask user to enter a number.

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifdef NOINPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif

printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

C #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

Syntax with #else:

```
#ifndef MACRO
//successful code
#else
//else code
#endif
```

C #ifndef example

Let's see a simple example to use #ifndef preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define INPUT
void main() {
int a=0;
#ifndef INPUT
a=2;
```

```
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Enter a:5
Value of a: 5
```

But, if you don't define INPUT, it will execute the code of #ifndef.

```
#include <stdio.h>
#include <conio.h>
void main() {
int a=0;
#ifndef INPUT
a=2;
#else
printf("Enter a:");
scanf("%d", &a);
#endif
printf("Value of a: %d\n", a);
getch();
}
```

Output:

```
Value of a: 2
```

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
#if expression
//code
#endif
```

Syntax with #else:

```
#if expression
//if code
#else
//else code
#endif
```

Syntax with #elif and #else:

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

C #if example

Let's see a simple example to use #if preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 0
void main() {
#if (NUMBER==0)
printf("Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

```
Value of Number is: 0
```

Let's see another example to understand the #if directive clearly.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
clrscr();
#if (NUMBER==0)
printf("1 Value of Number is: %d",NUMBER);
#endif

#if (NUMBER==1)
printf("2 Value of Number is: %d",NUMBER);
#endif
getch();
}
```

Output:

```
2 Value of Number is: 1
```

C #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression
//if code
#else
//else code
#endif
```

Syntax with #elif:

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

C #else example

Let's see a simple example to use #else preprocessor directive.

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 1
void main() {
#if NUMBER==0
printf("Value of Number is: %d",NUMBER);
#else
print("Value of Number is non-zero");
#endif
getch();
}
```

Output:

```
Value of Number is non-zero
```

C #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

Let's see a simple example to use #error preprocessor directive.

```
#include<stdio.h>
#ifndef __MATH_H
#error First include then compile
```

```
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

Output:

```
Compile Time Error: First include then compile
```

But, if you include math.h, it does not gives error.

```
#include<stdio.h>
#include<math.h>
#ifndef __MATH_H
#error First include then compile
#else
void main(){
   float a;
   a=sqrt(7);
   printf("%f",a);
}
#endif
```

Output:

```
2.645751
```

C #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

1. #pragma token

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

1. #pragma argsused
2. #pragma exit
3. #pragma hdrfile
4. #pragma hdrstop
5. #pragma inline
6. #pragma option
7. #pragma saveregs
8. #pragma startup
9. #pragma warn

Let's see a simple example to use #pragma preprocessor directive.

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

void func() ;

#pragma startup func
#pragma exit func

void main(){
printf("\nI am in main");
getch();
}

void func(){
printf("\nI am in func");
getch();
}
```

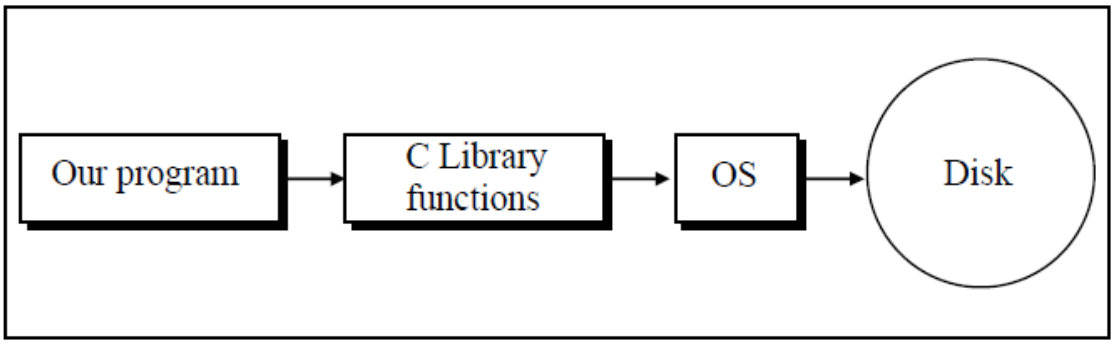
Output:

```
I am in func
I am in main
I am in func
```

Data Organization

Before we start doing file input/output let us first find out how data is organized on the disk. All data stored on the disk is in binary form. How this binary data is stored on the disk varies from one OS to another. However, this does not affect the C programmer since he has to use only the library functions written for the particular OS to be able

to perform input/output. It is the compiler vendor's responsibility to correctly implement these library functions by taking the help of OS. This is illustrated in Figure

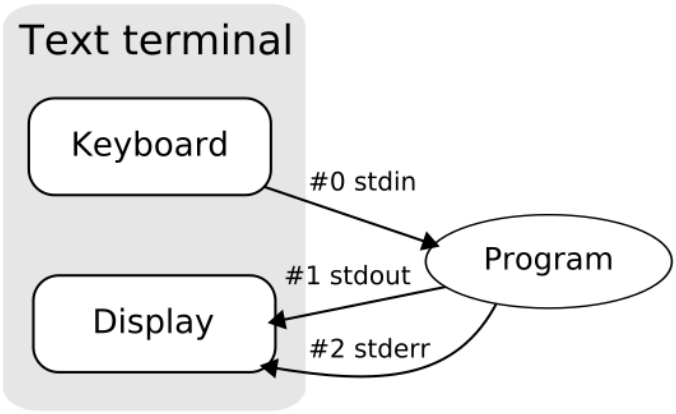


File I/O Streams in C Programming Language :

1. In C all **input and output** is done with streams
2. Stream is nothing but the **sequence of bytes of data**
3. A sequence of bytes flowing into program is called **input stream**
4. A sequence of bytes flowing out of the program is called **output stream**
5. Use of Stream make I/O machine independent.

Predefined Streams :

stdin	**Standard Input**
stdout	Standard Output
stderr	Standard Error



Standard Input Stream Device :

1. **stdin** stands for (**St**an**d**ard **In**put)
2. Keyboard is **standard input device** .

3. Standard input is data **(Often Text) going into a program**.
4. The program requests data transfers by use of the read operation.
5. Not all programs require input.

Standard Output Stream Device :

1. **stdout** stands for (**St**an**d**ard **Out**put)
2. Screen(Monitor) is **standard output device** .
3. Standard output is data **(Often Text) going out from a program**.
4. The program sends data to output device by using write operation.

Difference Between Std. Input and Output Stream Devices :

Point	**Std i/p Stream Device**	**Standard o/p Stream Device**
Stands For	Standard Input	Standard Output
Example	Keyboard	Screen/Monitor
Data Flow	Data (Often Text) going into a program	data (Often Text) going out from a program
Operation	Read Operation	Write Operation

Some Important Summary :

Point	**Input Stream**	**Output Stream**
Standard Device 1	Keyboard	Screen
Standard Device 2	Scanner	Printer
IO Function	scanf and gets	printf and puts
IO Operation	Read	Write
Data	Data goes from stream	data comes into stream

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.

File Operations

There are different operations that can be carried out on a file. These are:

(a) Creation of a new file
(b) Opening an existing file
(c)Reading from a file
(d)Writing to a file
(e) Moving to a specific location in a file (seeking)
(f) Closing a file

Text Files and Binary Files

All the programs that we wrote in this chapter so far worked on text files. Some of them would not work correctly on binary files. A text file contains only textual information like alphabets, digits and special symbols. In actuality the ASCII codes of these characters are stored in text files. A good example of a text file is any C program, say PR1.C.

As against this, a binary file is merely a collection of bytes. This collection might be a compiled version of a C program (say PR1.EXE), or music data stored in a wave file or a picture stored in a graphic file. A very easy way to find out whether a file is a text file or a binary file is to open that file in Turbo C/C++. If on opening the file you can make out what is displayed then it is a text file, otherwise it is a binary file.

File Handling in C

File Handling in c language is *used to open, read, write, search or close file*. It is used for permanent storage.

Advantage of File

It *will contain the data even after program exit*. Normally we use variable or array to store data, but data is lost after program exit. Variables and arrays are non-permanent storage medium whereas file is permanent storage medium.

Functions for file handling

There are many functions in C library to open, read, write, search and close file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into file
3	fscanf()	reads data from file
4	fputc()	writes a character into file
5	fgetc()	reads a character from file

6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

The fopen() function is used to open a file. The syntax of fopen() function is given below:

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

You can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

```
/* Display contents of a file on screen. */
# include<stdio.h>
```

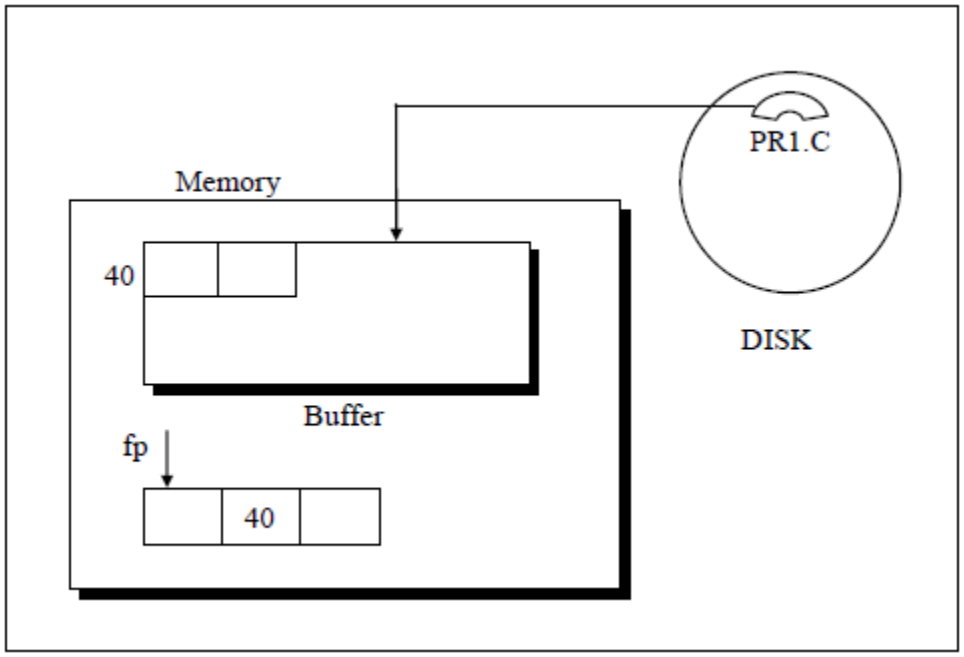
```
void main( )
{
FILE *fp ;
char ch ;
fp = fopen ( "PR1.C", "r" ) ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf ( "%c", ch ) ;
}
fclose ( fp ) ;
}
```

Opening a File

Before we can read (or write) information from (to) a file on a disk we must open the file. To open the file we have called the function **fopen( )**. It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Note that "r" is a string and not a character; hence the double quotes and not single quotes. In fact **fopen( )** performs three important tasks when you open the file in "r" mode:

(a) Firstly it searches on the disk the file to be opened.
(b) Then it loads the file from the disk into a place in memory called buffer.
(c) It sets up a character pointer that points to the first character of the buffer.

Why do we need a buffer at all? Imagine how inefficient it would be to actually access the disk every time we want to read a character from it. Every time we read something from a disk, it takes some time for the disk drive to position the read/write head correctly. On a floppy disk system, the drive motor has to actually start rotating the disk from a standstill position every time the disk is accessed. If this were to be done for every character we read from the disk, it would take a long time to complete the reading operation. This is where a buffer comes in. It would be more sensible to read the contents of the file into the buffer while opening the file and then read the file character by character from the buffer rather than from the disk. This is shown in Figure.



Same argument also applies to writing information in a file. Instead of writing characters in the file on the disk one character at a time it would be more efficient to write characters in a buffer and then finally transfer the contents from the buffer to the disk.

To be able to successfully read from a file information like mode of opening, size of file, place in the file from where the next read operation would be performed, etc. has to be maintained. Since all this information is inter-related, all of it is gathered together by **fopen( )** in a structure called **FILE**. **fopen( )** returns the address of this structure, which we have collected in the structure pointer called **fp**. We have declared **fp** as

```
FILE *fp ;
```

The **FILE** structure has been defined in the header file "stdio.h" (standing for standard input/output header file). Therefore, it is necessary to **#include** this file.

Reading from a File

Once the file has been opened for reading using **fopen( )**, as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer. This pointer is one of the elements of the structure to which **fp** is pointing (refer Figure).

To read the file's contents from memory there exists a function called **fgetc( )**. This has been used in our program as,

```
ch = fgetc ( fp ) ;
```

fgetc( ) reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable **ch**. Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer **fp**.

We have used the function **fgetc( )** within an indefinite **while** loop. There has to be a way to break out of this **while**. When shall we break out... the moment we reach the end of file. But what is end of file? A special character, whose ASCII value is 26, signifies end of file. This character is inserted beyond the last character in the file, when it is created.

While reading from the file, when **fgetc( )** encounters this special character, instead of returning the character that it has read, it returns the macro EOF. The EOF macro has been defined in the file "stdio.h". In place of the function **fgetc( )** we could have as well used the macro **getc( )** with the same effect.

```
/* Count chars, spaces, tabs and newlines in a file */
# include<stdio.h>
void main( )
{
FILE *fp ;
char ch ;
int nol = 0, not = 0, nob = 0, noc = 0 ;
fp = fopen ( "PR1.C", "r" ) ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
noc++ ;
if ( ch == ' ' )
nob++ ;
if ( ch == '\n' )
```

```
nol++ ;
if ( ch == '\t' )
not++ ;
}
fclose ( fp ) ;
printf ( "\nNumber of characters = %d", noc ) ;
printf ( "\nNumber of blanks = %d", nob ) ;
printf ( "\nNumber of tabs = %d", not ) ;
printf ( "\nNumber of lines = %d", nol ) ;
}
```

OUT PUT:

```
Number of characters = 125
Number of blanks = 25
Number of tabs = 13
Number of lines = 22
```

A File-copy Program

This program takes the contents of a file and copies them into another file, character by character.

```
# include<stdio.h>
void main( )
{
FILE *fs, *ft ;
char ch ;
fs = fopen ( "pr1.c", "r" ) ;
if ( fs == NULL )
{
puts ( "Cannot open source file" ) ;
exit( ) ;

}
ft = fopen ( "pr2.c", "w" ) ;
if ( ft == NULL )
{
puts ( "Cannot open target file" ) ;
fclose ( fs ) ;
exit( ) ;
}
while ( 1 )
{
ch = fgetc ( fs ) ;
if ( ch == EOF )
break ;
else
fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

C fputc() and fgetc():

Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

1. **int** fputc(**int** c, **FILE** *stream)

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file1.txt", "w");//opening file
  fputc('a',fp);//writing single character into file
  fclose(fp);//closing file
}
```

file1.txt

```
a
```

Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

1. **int** fgetc(**FILE** *stream)

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");

while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

myfile.txt

```
this is simple text message
```

C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fgets() functions.

Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

int fputs(**const char** *s, **FILE** *stream)

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
clrscr();
 fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);
fclose(fp);
getch();
}
```

myfile2.txt

```
hello c programming
```

Reading File : fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax:

char* fgets(**char** *s, **int** n, **FILE** *stream)

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();
fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));
fclose(fp);
getch();
}
```

Output:

```
hello c programming
```

C fprintf() and fscanf()

Writing File : fprintf() function

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

int fprintf(**FILE** *stream, **const char** *format [, argument, ...])

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

Reading File : fscanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

int fscanf(**FILE** *stream, **const char** *format [, argument, ...])

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  char buff[255];//creating char array to store data of file
  fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF){
  printf("%s ", buff );
  }
  fclose(fp);
}
```

Output:

```
Hello file by fprintf...
```

C fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

int fseek(**FILE** *stream, **long int** offset, **int** whence)

There are 3 constants used in the fseek() function for whence: SEEK_SET, SEEK_CUR and SEEK_END.

Example:

```
#include <stdio.h>
void main(){
FILE *fp;
fp = fopen("myfile.txt","w+");
fputs("This is javatpoint", fp);
fseek( fp, 7, SEEK_SET );
fputs("sonoo jaiswal", fp);
fclose(fp);
}
```

myfile.txt

```
This is sonoo jaiswal
```

C rewind() function

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax:

```
void rewind(FILE *stream)
```

Example:

File: file.txt

```
this is a simple text
```

File: rewind.c

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("file.txt","r");
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
rewind(fp);//moves the file pointer at beginning of the file
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

Output:

```
this is a simple textthis is a simple text
```

As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax:

long int ftell(**FILE** *stream)

Example:

File: ftell.c

```
#include <stdio.h>
#include <conio.h>
void main (){
FILE *fp;
int length;
clrscr();
fp = fopen("file.txt", "r");
fseek(fp, 0, SEEK_END);
length = ftell(fp);
fclose(fp);
printf("Size of file: %d bytes", length);
getch();
}
```

Output:

```
Size of file: 21 bytes
```

Binary file example program:

```
#include<stdio.h>
void main( )
{
FILE *fs, *ft ;
int ch ;
fs = fopen ( "pr1.exe", "rb" ) ;
if ( fs == NULL )
{
puts ( "Cannot open source file" ) ;
exit( ) ;
```

```
}
ft = fopen ( "newpr1.exe", "wb" ) ;
if ( ft == NULL )
{
puts ( "Cannot open target file" ) ;
fclose ( fs ) ;
exit( ) ;
}
while ( 1 )
{
ch = fgetc ( fs ) ;
if ( ch == EOF )
break ;
else
fputc ( ch, ft ) ;
}
fclose ( fs ) ;
fclose ( ft ) ;
}
```

Difference between text file and binary file

1. Text file is human readable because everything is stored in terms of text. In binary file everything is written in terms of 0 and 1, therefore binary file is not human readable.
2. A newline(\n) character is converted into the carriage return-linefeed combination before being written to the disk. In binary file, these conversions will not take place.
3. In text file, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. There is no such special character present in the binary mode files to mark the end of file.
4. In text file, the text and characters are stored one character per byte. For example, the integer value 1245 will occupy 2 bytes in memory but it will occupy 5 bytes in text file. In binary file, the integer value 1245 will occupy 2 bytes in memory as well as in file.

/* Writes records to a file using structure */

```
#include "stdio.h"
void main( )
{
FILE *fp ;
char another = 'Y' ;
struct emp
{
char name[40] ;
int age ;
float bs ;
} ;
struct emp e ;
fp = fopen ( "EMPLOYEE.txt", "w" ) ;
if ( fp == NULL )
{
```

```
puts ( "Cannot open file" ) ;
exit( ) ;
}
while ( another == 'Y' )
{
printf ( "\nEnter name, age and basic salary: " ) ;
scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;
printf ( "Add another record (Y/N) " ) ;
fflush ( stdin ) ;
another = getche( ) ;
}
fclose ( fp ) ;

}
```

And here is the output of the program...

```
Enter name, age and basic salary: Sunil 34 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Sameer 21 1300.50
Add another record (Y/N) Y
Enter name, age and basic salary: Rahul 34 1400.55
Add another record (Y/N) N
```

In this program we are just reading the data into a structure variable using **scanf( )**, and then dumping it into a disk file using **fprintf( ).** The user can input as many records as he desires. The procedure ends when the user supplies 'N' for the question 'Add another record (Y/N)'.

Let us now write a program that reads the employee records created by the above program. Here is how it can be done...

```
/* Read records from a file using structure */

#include "stdio.h"
void main( )
{
FILE *fp ;
struct emp
{
char name[40] ;
int age ;
float bs ;
} ;
struct emp e ;
fp = fopen ( "EMPLOYEE.txt", "r" ) ;
if ( fp == NULL )
{
puts ( "Cannot open file" ) ;
exit( ) ;
}
while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;
fclose ( fp ) ;
```

```
}
```

And here is the output of the program...
Sunil 34 1250.500000
Sameer 21 1300.500000
Rahul 34 1400.500000

Syntax of putw() function

```
putw(int number, FILE *fp);
```

The putw() function takes two arguments, first is an integer value to be written to the file and second is the file pointer where the number will be written.

```
#include<stdio.h>
void main()
{
    FILE *fp;
    int num;
    char ch='n';
    fp = fopen("file.txt","w");          //Statement   1
    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
    }

    do                                   //Statement   2
    {
        printf("\nEnter any number : ");
        scanf("%d",&num);
        putw(num,fp);
        printf("\nDo you want to another number : ");
        ch = getche();
    }while(ch=='y'||ch=='Y');
    printf("\nData written successfully...");
    fclose(fp);
}
```

```
Output :

        Enter any number : 78
        Do you want to another number : y
        Enter any number : 45
        Do you want to another number : y
        Enter any number : 63
        Do you want to another number : n
        Data written successfully...
```

In the above example, statement 1 will create a file named file.txt in write mode. Statement 2 is a loop, which take integer values from user and write the values to the file.

The getw() function

The getw() function is used to read integer value form the file.

Syntax of getw() function

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

The getw() function takes the file pointer as argument from where the integer value will be read and returns returns the *end-of-file* if it has reached the end of file.

Example of getw() function

```
#include<stdio.h>
void main()
{
    FILE *fp;
    int num;
    fp = fopen("file.txt","r");          //Statement   1
    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
    }
```

```
        printf("\nData in file...\n");
        while((num = getw(fp))!=EOF)          //Statement   2
            printf("\n%d",num);
        fclose(fp);
    }
```

Output :

```
        Data in file...
        78
        45
        63
```

In the above example, Statement 1 will open an existing file file.txt in read mode and statement 2 will read all the integer values upto EOF(end-of-file) reached.

TThe fread() fwrite() function in C

The fwrite() function

The fwrite() function is used to write records (sequence of bytes) to the file. A record may be an array or a structure.

Syntax of fwrite() function

```
        fwrite( ptr, int size, int n, FILE *fp );
```

The fwrite() function takes four arguments.
ptr : ptr is the reference of an array or a structure stored in memory.
size : size is the total number of bytes to be written.
n : n is number of times a record will be written.
FILE* : FILE* is a file where the records will be written in binary mode.

Example of fwrite() function

```
    #include<stdio.h>
    struct Student
    {
        int roll;
        char name[25];
        float marks;
```

```
};
void main()
{
    FILE *fp;
    char ch;
    struct Student Stu;
    fp = fopen("Student.dat","w");          //Statement  1
    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
    }
    do
    {
        printf("\nEnter Roll : ");
        scanf("%d",&Stu.roll);
        printf("Enter Name : ");
        scanf("%s",Stu.name);
        printf("Enter Marks : ");
        scanf("%f",&Stu.marks);
        fwrite(&Stu,sizeof(Stu),1,fp);
        printf("\nDo you want to add another data (y/n) : ");
        ch = getche();
    }while(ch=='y' || ch=='Y');
    printf("\nData written successfully...");
    fclose(fp);
}
```

Output :

```
    Enter Roll : 1
    Enter Name : Ashish
    Enter Marks : 78.53
    Do you want to add another data (y/n) : y
    Enter Roll : 2
```

```
Enter Name : Kaushal
Enter Marks : 72.65
Do you want to add another data (y/n) : y
Enter Roll : 3
Enter Name : Vishwas
Enter Marks : 82.65
Do you want to add another data (y/n) : n
Data written successfully...
```

The fread() function

The fread() function is used to read bytes form the file.

Syntax of fread() function

```
fread( ptr, int size, int n, FILE *fp );
```

The fread() function takes four arguments.
ptr : ptr is the reference of an array or a structure where data will be stored after reading.
size : size is the total number of bytes to be read from file.
n : n is number of times a record will be read.
FILE* : FILE* is a file where the records will be read.

Example of fread() function

```
#include<stdio.h>
struct Student
{
    int roll;
    char name[25];
    float marks;
};
void main()
{
    FILE *fp;
    char ch;
    struct Student Stu;
    fp = fopen("Student.dat","r");          //Statement   1
    if(fp == NULL)
```

```
        {
            printf("\nCan't open file or file doesn't exist.");
            exit(0);
        }
        printf("\n\tRoll\tName\tMarks\n");
        while(fread(&Stu,sizeof(Stu),1,fp)>0)
           printf("\n\t%d\t%s\t%f",Stu.roll,Stu.name,Stu.marks);
        fclose(fp);
    }
```

Output :

Roll	Name	Marks
1	Ashish	78.53
2	Kaushal	72.65
3	Vishwas	82.65