

UNIT - V

FILE MANAGEMAEENT IN C
FILE MANAGEMAEENT IN C

File Handling in C:

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console.

The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

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

No.	Function	Description
1	fopen()	opens new or existing file

2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the 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

Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the
1.filename, 2.data structure, 3.purpose.

The general format of the function used for opening a file is

```
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.

This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening the file. The mode does this job.

R open the file for read only.

W open the file for writing only.

A open the file for appending data to it.

Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
```

SNS COLLEGE OF TECHNOLOGY

Coimbatore-35

An Autonomous Institution

Accredited by NBA, AICTE and Accredited by NAAC – UGC with ‘A+’ Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

Closing a file:

The input output library supports the function to close a file; it is in the following format.

```
fclose(file_pointer);
```

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.

Observe the following program.

```
....  
FILE *p1 *p2;  
p1=fopen (“Input”,”w”);  
p2=fopen (“Output”,”r”);  
....  
...  
fclose(p1);  
fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file. The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time.

The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode.

```
c=getc(fp2).
```

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it.

Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
main()
{
file *f1;
printf(“Data input output”);
f1=fopen(“Input”,”w”); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf(“nData outputn”);
f1=fopen(“INPUT”,”r”); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf(“%c”,c);
fclose(f1);
}
```

I/O Operations on files:

Writing to a File:

In C, when you write to a file, newline characters ‘\n’ must be explicitly added.

The stdio library offers the necessary functions to write to a file:

- `fputc(char, file_pointer)`: It writes a character to the file pointed to by `file_pointer`.
- `fputs(str, file_pointer)`: It writes a string to the file pointed to by `file_pointer`.
- `fprintf(file_pointer, str, variable_lists)`: It prints a string to the file pointed to by `file_pointer`. The string can optionally include format specifiers and a list of variables `variable_lists`.

The program below shows how to perform writing to a file:

`fputc()` Function:

```
#include <stdio.h>
int main() {
    int i;
    FILE * fptr;
    char fn[50];
```



```
char str[] = "Guru99 Rocks\n";
fptr = fopen("fputc_test.txt", "w"); // 'w' defines writing mode
for (i = 0; str[i] != '\n'; i++) {
    /* write to file using fputc() function */
    fputc(str[i], fptr);
}
fclose(fptr);
return 0;
}
```

Output:



The above program writes a single character into the fputc_test.txt file until it reaches the next line symbol “\n” which indicates that the sentence was successfully written. The process is to take each character of the array and write it into the file.

```
include <stdio.h>
int main() {
    int i;
    FILE * fptr;
    char fn[50];
    char str[] = "Guru99 Rocks\n";
    fptr = fopen("fputc test.txt", "w");
    for (i = 0; str[i] != '\n'; i++) {
        /* write to file using fputc() =
        fputc(str[i], fptr);
    }
    fclose(fptr);
    return 0;
}
```

1. In the above program, we have created and opened a file called fputc_test.txt in a write mode and declare our string which will be written into the file.
2. We do a character by character write operation using for loop and put each character in our file until the “\n” character is encountered then the file is closed using the fclose function.

fputs () Function:

```
#include <stdio.h>
int main() {
```

```
FILE *fp;
fp = fopen("fputs_test.txt", "w+");
fputs("This is Guru99 Tutorial on fputs,", fp);
fputs("We don't need to use for loop\n", fp);
fputs("Easier than fputc function\n", fp);
fclose(fp);
return (0);
}
```

OUTPUT:



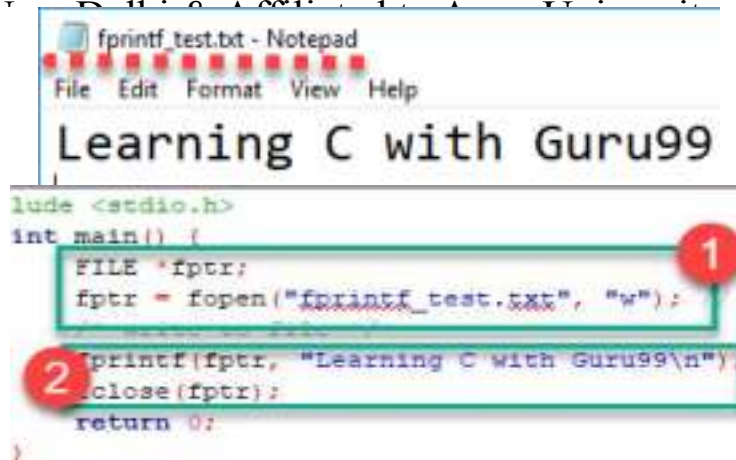
```
#include <stdio.h>
int main() {
    FILE * fp;
    fp = fopen("fputs test.txt", "w+");
    fputs("This is Guru99 Tutorial on fputs,", fp);
    fputs("We don't need to use for loop\n", fp);
    fputs("Easier than fputc function\n", fp);
    fclose(fp);
    return (0);
}
```

1. In the above program, we have created and opened a file called fputs_test.txt in a write mode.
2. After we do a write operation using fputs() function by writing three different strings
3. Then the file is closed using the fclose function.

fprintf()Function:

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

```
int main() {
    FILE *fptr;
    fptr = fopen("fprintf_test.txt", "w"); // "w" defines "writing mode"
    /* write to file */
    fprintf(fptr, "Learning C with Guru99\n");
    fclose(fptr);
    return 0;
}
```



```
fprintf_test.txt - Notepad
File Edit Format View Help

Learning C with Guru99

#include <stdio.h>
int main() {
    FILE *fptr;
    fptr = fopen("fprintf_test.txt", "w");
    fprintf(fptr, "Learning C with Guru99\n");
    fclose(fptr);
    return 0;
}
```

1. In the above program we have created and opened a file called fprintf_test.txt in a write mode.
2. After a write operation is performed using fprintf() function by writing a string, then the file is closed using the fclose function.

Reading data from a File:

There are three different functions dedicated to reading data from a file

- fgetc(file_pointer): It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.
- fgets(buffer, n, file_pointer): It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.
- fscanf(file_pointer, conversion_specifiers, variable_addresses): It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable_addresses using conversion specifiers. Keep in mind that as with scanf, fscanf stops reading a string when space or newline is encountered.

The following program demonstrates reading from fputs_test.txt file using fgets(), fscanf() and fgetc () functions respectively :

```
#include <stdio.h>
int main() {
    FILE * file_pointer;
```


SNS COLLEGE OF TECHNOLOGY

Coimbatore-35

An Autonomous Institution

Accredited by NBA, AICTE and Accredited by NAAC – UGC with ‘A+’ Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

```
char buffer[30], c;
file_pointer = fopen("fprintf_test.txt", "r");
printf("----read a line--- \n");
fgets(buffer, 50, file_pointer);
printf("%s\n", buffer);

printf("----read and parse data----\n");
file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
char str1[10], str2[2], str3[20], str4[2];
fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
printf("Read String1 |%s|\n", str1);
printf("Read String2 |%s|\n", str2);
printf("Read String3 |%s|\n", str3);
printf("Read String4 |%s|\n", str4);
printf("----read the entire file ---\n");
file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
while ((c = getc(file_pointer)) != EOF) printf("%c", c);
fclose(file_pointer);
return 0;
}
```

Result:

----read a line----

Learning C with Guru99

----read and parse data----

Read String1 |Learning|

Read String2 |C|

Read String3 |with|

Read String4 |Guru99|

----read the entire file----

Learning C with Guru99

```
int main() {
    FILE * file_pointer;
    char buffer[30], c;

    file_pointer = fopen("fprintf_test.txt", "r");
    printf("----read a line----\n");
    fgets(buffer, 30, file_pointer);
    printf("%s\n", buffer);

    printf("----read and parse data----\n");
    file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
    char str1[10], str2[2], str3[20], str4[2];
    fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
    printf("Read String1 |%s|\n", str1);
    printf("Read String2 |%s|\n", str2);
    printf("Read String3 |%s|\n", str3);
    printf("Read String4 |%s|\n", str4);

    printf("----read the entire file----\n");
    file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
    while ((c = getc(file_pointer)) != EOF) printf("%c", c);

    fclose(file_pointer);
    return 0;
}
```

1. In the above program, we have opened the file called “fprintf_test.txt” which was previously written using fprintf() function, and it contains “Learning C with Guru99” string. We read it using the fgets() function which reads line by line where the buffer size must be enough to handle the entire line.
2. We reopen the file to reset the pointer file to point at the beginning of the file. Create various strings variables to handle each word separately. Print the variables to see their contents. The fscanf() is mainly used to extract and parse data from a file.
3. Reopen the file to reset the pointer file to point at the beginning of the file. Read data and print it from the file character by character using getc() function until the EOF statement is encountered
4. After performing a reading operation file using different variants, we again closed the file using the fclose function.

Error Handling during i/o operations:

Error handling is not supported by C language. There are some other ways by which error handling can be done in C language. The header file “error.h” is used to print the errors using return statement function.

It returns -1 or NULL in case of any error and errno variable is set with the error code. Whenever a function is called in C language, errno variable is associated with it. errno is a global variable and is used to find the type of error in the execution.

The following table displays some errors –

Sr.No	Errors & Error value
1	I/O Error 5
2	No such file or directory 2
3	Argument list too long 7
4	Out of memory 12
5	Permission denied 13

There are some methods to handle errors in C language –

- `strerror()` – This function is declared in “string.h” header file and it returns the pointer to the string of current `errno` value.
- Exit status – There are two constants `EXIT_SUCCESS` and `EXIT_FAILURE` which can be used in function `exit()` to inform the calling function about the error.
- Divided by zero – This is a situation in which nothing can be done to handle this error in C language. Avoid this error and you can check the divisor value by using ‘if’ condition in the program.

Here is an example of error handling in C language,

Example

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int x = 28;
    int y = 8;
    int z;
    if( y == 0) {
        fprintf(stderr, "Division by zero!\n");
        exit(EXIT_FAILURE);
    }
    z = x / y;
    fprintf(stderr, "Value of z : %d\n", z);
    exit(EXIT_SUCCESS);
}
```

Output

Here is the output

Random Access To File:

There is no need to read each record sequentially, if we want to access a particular record. C supports these functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

fseek():

This function is used for seeking the pointer position in the file at the specified byte.

Syntax: fseek(file pointer, displacement, pointer position);

Where

file pointer ---- It is the pointer which points to the file.

displacement -----It is positive or negative. This is the number of bytes which are skipped backward (if negative) or forward (if positive) from the current position. This is attached with L because this is a long integer.

Pointer position:

This sets the pointer position in the file.

Value	pointer position
0	Beginning of file.
1	Current position
2	End of file

Ex:

1) fseek(p,10L,0)

0 means pointer position is on beginning of the file, from this statement pointer position is skipped 10 bytes from the beginning of the file.

2) fseek(p,5L,1)

1 means current position of the pointer position. From this statement pointer

From this statement pointer position is skipped 5 bytes backward from the current position.

ftell()

This function returns the value of the current pointer position in the file. The value is count from the beginning of the file.

Syntax: `ftell(fp);`

Where `fp` is a file pointer.

rewind()

This function is used to move the file pointer to the beginning of the given file.

Syntax: `rewind(fp);`

Where `fp` is a file pointer.

Example program for `fseek()`:

Write a program to read last ‘n’ characters of the file using appropriate file functions(Here we need `fseek()` and `fgetc()`).

```
01  #include<stdio.h>
02  #include<conio.h>
03  void main()
04  {
05      FILE *fp;
06      char ch;
07      clrscr();
08      fp=fopen("file1.c", "r");
09      if(fp==NULL)
10      printf("file
cannot be
opened");
```

```
12     {
13         printf("Enter
value of n to
read last 'n'
characters");
14         scanf("%d",&n);
15         fseek(fp,-
n,2);
16         while((ch=fgetc(fp))!=EOF)
17         {
18             printf("%c\t",ch);
19         }
20     }
21     fclose(fp);
22     getch();
23 }
```

OUTPUT: It depends on the content in the file.

Command Line Arguments in C:

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

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

Here, argc counts the number of arguments. It counts the file name as the first argument.

Skip Ad

Example

Let's see the example of command line arguments where we are passing one argument with file name.

```
1. #include <stdio.h>
2. void main(int argc, char *argv[] ) {
3.
4.     printf("Program name is: %s\n", argv[0]);
5.
6.     if(argc < 2){
7.         printf("No argument passed through command line.\n");
8.     }
9.     else{
10.        printf("First argument is: %s\n", argv[1]);
11.    }
12.}
```

Run this program as follows in Linux:

```
1. ./program hello
```

Run this program as follows in Windows from command line:

```
1. program.exe hello
```

Output:

```
Program name is: program
```

```
First argument is: hello
```

If you pass many arguments, it will print only one.

```
1. ./program hello c how r u
```

Output:

```
Program name is: program
```

```
First argument is: hello
```

But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

```
1. ./program "hello c how r u"
```

Output:

```
Program name is: program
```


Approved You can write your program to print all the arguments. In this program, we are printing only argv[1], that is why it is printing only one argument.

PREPROCESSORS

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif

9	<code>#endif</code> Ends preprocessor conditional.
10	<code>#error</code> Prints error message on stderr.
11	<code>#pragma</code> Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of `MAX_ARRAY_LENGTH` with 20. Use `#define` for constants to increase readability.

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

These directives tell the CPP to get `stdio.h` from System Libraries and add the text to the current source file. The next line tells CPP to get `myheader.h` from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing `FILE_SIZE` and define it as 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

It tells the CPP to define `MESSAGE` only if `MESSAGE` isn't already defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if `DEBUG` is defined. This is useful if you pass the `-DDEBUG` flag to the gcc compiler at the time of compilation. This will define `DEBUG`, so you can turn debugging on and off on the fly during compilation.

SNS COLLEGE OF TECHNOLOGY

Coimbatore-35

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A+’ Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai