Introduction

File is a place on disk where a group of related data is stored. For example we can store characters, words, lines, paragraphs and pages from a textual document, fields and records belonging to a database; or pixels from a graphical image in a file.

In order to store information permanently and retrieve it we need to use files. A file represents a sequence of bytes, does not matter if it is a text file or binary file. C language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

Why files are needed? When the program is terminated, the entire data is lost. If you want persist some data across different runs of a program (like the highest score in a game) you need to permanently store that information. That is where files come in.

Files can be categorized as:

- text files and
- binary files.

Text files are human readable, while binary files are not. We will only see how to handle text files in these notes.

Working with files

FILE pointer is `struct` defined in standard library.

File pointer is capable of managing all the information needed to handle a stream, including its file position indicator, a pointer to the associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached. The declaration is:

FILE* fp;

Opening a file

The `fopen()` function is used to create a new file or to open an existing file. This call will initialize an object of the type FILE. Following is the prototype of this function call:

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

Here, filename is string literal, which you will use to name your file and access mode can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
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.
a+	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.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b".

If the `fopen()` call is successful a valid FILE handle is returned. If the call failed, NULL is returned.

Closing a File

After finishing the reading or writing to a file, it must be closed. Otherwise you run the risk of losing the data and/or corrupting the contents of the file.

To close a file, use the `fclose()` function. The prototype of this function is: **int** fclose (FILE* fp);
The `fclose()` function returns zero on success, or EOF if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file *stdio.h*.

File I/O

There are various functions provided by C standard library to read and write a file. You can read or write one character at a time or in the form of a fixed length strings or lines and so on.

fgetc() **int** fgetc(FILE* fp); This is the simplest function to read a single character from a file. It reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns EOF.

getchar() This is the same as `fgetc(stdin)`; in other words reading a character from the keyboard.

fputc() **int** fputc(**int** c, FILE* fp); It is the simplest function to write individual characters to a stream. It writes the character value of the argument c to the output stream referenced by fp. It returns the character written on success; `EOF` if there is an error.

putchar() This is the same as `fputc( ,stdin)`; in other words displaying a character at the screen.

fgets() char* fgets(char* buf, **int** n, FILE* fp); We have already used this. It reads up to $n-1$ characters from the input stream fp. It copies the read string into the buffer buf, appending a null character to terminate the string. If this function encounters a newline character '\n' or the end of the file EOF before reading $n-1$ characters, then it returns only the characters read up to that point including new line character.

fputs() fputs(const char* **str**, FILE* fp); The function writes the string `str` to the output stream referenced by fp. It returns a non-negative value on success; `EOF` is returned in case of any error.

fprintf() fprintf (FILE* fp, const char* **format**, ...) , this is similar to printf. Writes formatted output to the file stream fp.

fscanf() fscanf(FILE* fp, const char* **format**, ...) Reads structured/formatted input. This is the file counterpart of `scanf()`.

Examples

For better understanding of the programs, view the input files and output files produced, in a text editor.

We will use commandline arguments to pass the name of the file to read and write. We will also build these programs using the -o option of c99.

Character I/O

This program reads a file, character by character and if any line is longer than 25 characters, it breaks it at 25 characters. To indicate that, it adds a string "-|-" at the end of such lines.

Here is the input file:

```
$ cat Data-13-1.txt
This is an input file.
It is read by Program-13-1. It has
some lines longer than 25 characters.

It also has a few lines that are longer than 50 characters.
It is intersting to see how the program handles all of that.

Please read carefully.
$
```

Figure 1: Character Input

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char* argv[]) {
4      if (argc != 2) {
5          printf("Usage: %s Filename\n", argv[0]);
6          exit(0);
7      }
8
9      FILE* inp = fopen(argv[1], "r");
10     int charCount = 0;
11     int ch;
12
13     const int BREAK = 25;
14     while ((ch = fgetc(inp)) != EOF) {
15         putchar(ch);
16         if (ch == '\n')
```



```
17              charCount = 0;
18          else {
19              charCount++;
20              if (charCount > BREAK) {
21                  puts("-|-");
22                  charCount = 0;
23              }
24          }
25      }
26      fclose(inp);
27      return 0;
28  }
```

```
$
$ c99 Program-13-1.c -o Program-13-1
$ ./Program-13-1
Usage: ./Program-13-1 Filename
$ ./Program-13-1 Data-13-1.txt
This is an input file.
It is read by Program-13-1-|-
. It has
some lines longer than 25 -|-
characters.

It also has a few lines th-|-
at are longer than 50 char-|-
acters.
It is intersting to see ho-|-
w the program handles all -|-
of that.

Please read carefully.
$
```

Figure 2: Character I/O

Now let us write another program that read the same input file, removes all newlines ('\n'), adds a space after the period, and writes it to an output file one character at a time.

```
1  #include <stdio.h>
2  #include <stdlib.h>
```



```
3  int main(int argc, char* argv[]) {
4      if (argc != 3) {
5          printf("Usage: %s Input-Filename Output-Filename\n", argv[0]);
6          exit(0);
7      }
8
9      FILE* inp = fopen(argv[1], "r");
10     FILE* out = fopen(argv[2], "w");
11
12     int ch;
13     while ((ch = fgetc(inp)) != EOF) {
14         if (ch == '\n')
15             continue;
16         fputc(ch, out);
17         if (ch == '.')
18             fputc(' ', out);
19     }
20     fclose(inp);
21     fclose(out);
22     return 0;
23 }
```

Note how the **continue** statement is used in this program.

When we run the program this is what we see:

```
$ c99 Program-13-2.c -o Program-13-2
$ ./Program-13-2
Usage: ./Program-13-2 Input-Filename Output-Filename
$ ./Program-13-2 Data-13-1.txt Output-13-2.txt
$
```

Figure 3: Character I/O

Here is how the file looks when opened in vim:


```
This is an input file. It is read by Program-13-1.  It hassome lines longer than 25 characters. It a
lso has a few lines that are longer than 50 characters. It is intersting to see how the program hand
les all of that. Please read carefully.
~
~
```

Figure 4: Character I/O

String I/O

Here we will read a file line by line, using `fgets()` and write the output also line by line using `fputs()`. We will output each line with its line number.

Note the use of the function `sprintf()` in this code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage: %s Input-File Output-File\n", argv[0]);
        exit(0);
    }

    FILE* inp = fopen(argv[1], "r");
    FILE* out = fopen(argv[2], "w");

    char line[1024];
    char buffer[1024];

    int lineNum = 0;
    while (fgets(buffer, 1024, inp) != NULL) {
        sprintf(line, "%4d: ", ++lineNum);
        strcat(line, buffer);
        fputs(line, out);
    }

    fclose(inp);
    fclose(out);
    return 0;
```



```
}
```

```
$
$ c99 Program-13-3.c -o Program-13-3
$ ./Program-13-3
Usage: ./Program-13-3 Input-File Output-File
$ ./Program-13-3 Data-13-1.txt Output-13-3.txt
$ cat Output-13-3.txt
   1: This is an input file.
   2: It is read by Program-13-1. It has
   3: some lines longer than 25 characters.
   4:
   5: It also has a few lines that are longer than 50 characters.
   6: It is intersting to see how the program handles all of that.
   7:
   8: Please read carefully.
$
```

Figure 5: String I/O

We can rewrite this program to use `fprintf()`. In this case it also becomes simpler.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Usage: %s Input-File Output-File\n", argv[0]);
        exit(0);
    }

    FILE* inp = fopen(argv[1], "r");
    FILE* out = fopen(argv[2], "w");

    char line[1024];

    int lineNum = 0;
    while (fgets(line, 1024, inp) != NULL)
        fprintf(out, "%4d: %s", ++lineNum, line);

    fclose(inp);
    fclose(out);
    return 0;
}
```


We use `diff` utility to check that two files are identical.

```
$
$ c99 Program-13-3A.c -o Program-13-3A
$ ./Program-13-3A Data-13-1.txt Output-13-3A.txt
$ diff Output-13-3A.txt Output-13-3.txt
$
```

Figure 6: String I/O

