C – FILE-IO

Dr. Sheak Rashed Haider Noori

Professor & Associate Head

Department of Computer Science and Engineering

C - FILE

- A file represents a sequence of bytes, does not matter if it is a text file or binary file.
- C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.
- This lesson will take you through important calls for the file management.

OPENING FILES

- You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream.
- 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
  - **mode** represent the access, which can have one of the following values(see the next slide).

FILE ACCESS MODE

Mode	Role	If the file already exists	If the file does not exist
r	Opens an existing file for reading only	it would be opened and can be read. After the file is opened, the user cannot add data to it	the operation would fail
w	Saves a new file	the file's contents would be deleted and replaced by the new content	a new file is created and can be written to
a	Opens an existing file, saves new file, or saves a existing file that has been modified	the file is opened and can be modified or updated. New information written to the file would be added to the end of the file	a new file is created and can be written to
r+	Opens an existing file	the file is opened and its existing data can be modified or updated	the operation would fail
w+	Creates new file or saves an existing one	the file is opened, its contents would be deleted and replaced with the new contents	a new file is created and can be written to
a+	Creates a new file or modifies an existing one	it is opened and its contents can be updated. New information written to the file would be added to the end of the file	a new file is created and can be written to

CLOSING A 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**.
- There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

WRITING A FILE

- Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

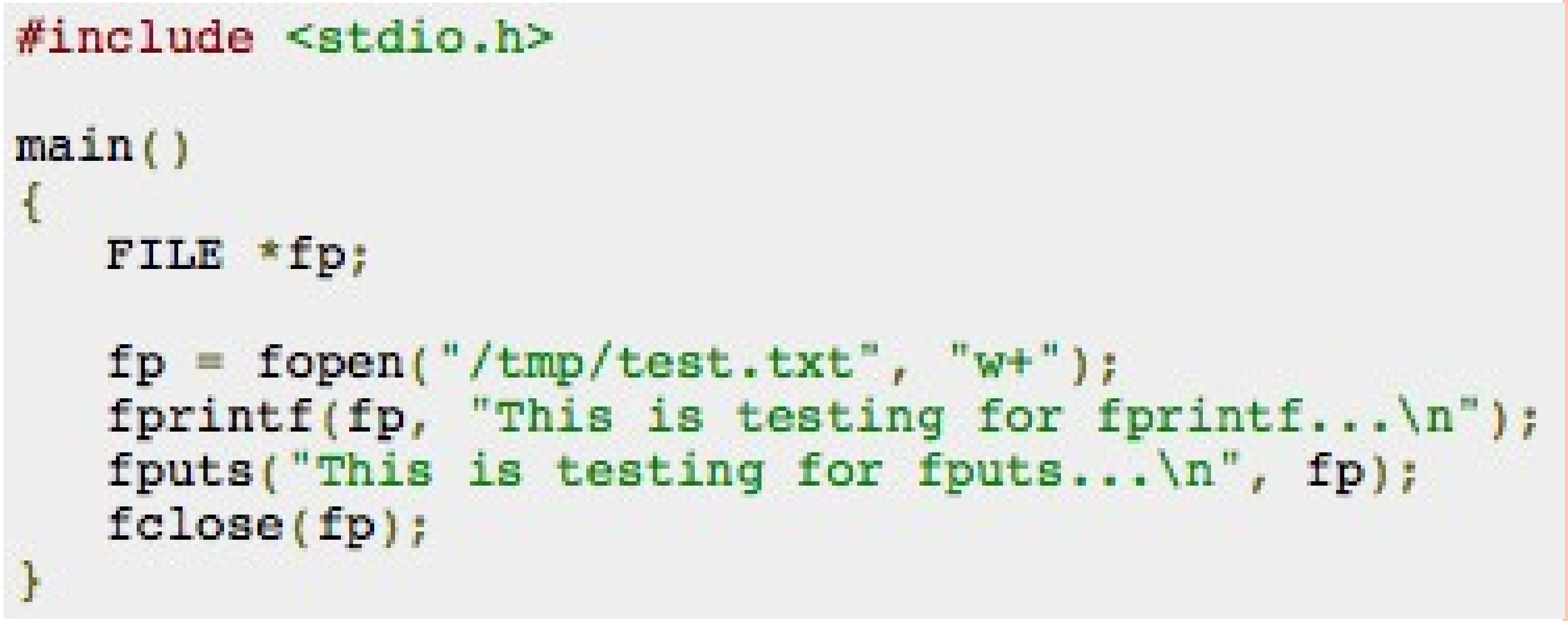
- The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error.
- You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

- The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error.
- You can use **int fprintf(FILE *fp, const char *format, ...)** function as well to write a string into a file.

WRITING A FILE

- Try the following example:



```
#include <stdio.h>

main()
{
    FILE *fp;

    fp = fopen("/tmp/test.txt", "w+");
    fprintf(fp, "This is testing for fprintf...\n");
    fputs("This is testing for fputs...\n", fp);
    fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file *test.txt* in */tmp* directory and writes two lines using two different functions. Let us read this file in next section.

READING A FILE

- Following is the simplest function to read a single character from a file:

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

- The **fgetc()** function 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**.
- The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

- The functions **fgets()** reads up to n - 1 characters from the input stream referenced by **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 they have read the maximum number of characters, then it returns only the characters read up to that point including new line character.
- You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

READING A FILE

```c
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When this code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First fscanf() method read just **This** because after that it encountered a space, second call is for fgets() which read the remaining line till it encountered end of line. Finally last call fgets() read second line completely.

READING A FILE

Suppose the input file consists of lines with a *username* and an *integer test score*, e.g.:

```
list.txt
------
Foo  70
Bar  98
```

and that each username is no more than 8 characters long. We might use the files we opened above by copying each username and score from the input file to the output file.

In the process, we'll increase each score by 10 points for the output file:

```
char username[9]; /* One extra for nul char. */
int score;
...
while (fscanf(ifp, "%s %d", username, &score) != EOF) {
        fprintf(ofp, "%s %d\n", username, score+10);
}
```

The function fscanf(), like scanf(), normally returns the number of values it was able to read in. However, when it hits the end of the file, it returns the special value EOF.

So, testing the return value against EOF is one way to stop the loop.

READING A FILE CONT.

The bad thing about testing against EOF is that if the file is not in the right format (e.g., a letter is found when a number is expected):

```
list.txt
------
foo 70
bar 98
biz A+
...
```

then fscanf() will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, fscanf() will not return EOF (it's not at the end of the file)

READING A FILE CONT.

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an infinite loop.

One solution is to test against the number of values we expect to be read by fscanf() each time. Since our format is "%s %d", we expect it to read in 2 values, so our condition could be:

```
while (fscanf(ifp, "%s %d", username, &score) == 2) {
  ...
```

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number with %d), then the loop will end.

READING A FILE CONT.

Another way to test for end of file is with the library function feof(). It just takes a file pointer and returns a true/false value based on whether we are at the end of the file.

To use it in the above example, you would do:

```
while (!feof(ifp)) {
        if (fscanf(ifp, "%s %d", username, &score) != 2)
            break;
        fprintf(ofp, "%s %d", username, score+10);
}
```

Note that, like testing != EOF, it might cause an infinite loop if the format of the input file was not as expected. However, we can add code to make sure it reads in 2 values (as we've done above).

BINARY I/O FUNCTIONS

- There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

- Both of these functions should be used to read or write blocks of memories - usually arrays or structures.
- If you are going to handle binary files then you will use below mentioned access modes instead of the above

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

SEE ALSO

- www.cs.bu.edu/teaching/c/file-io/intro/
- www.codingunit.com/c-tutorial-file-io-using-text-files
- en.wikibooks.org/wiki/A_Little_C_Primer/C_File-IO_Through_Library_Functions