Text Files

Learning Outcomes

After reading this section, you will be able to:

- Stream data using standard library functions to access persistent text

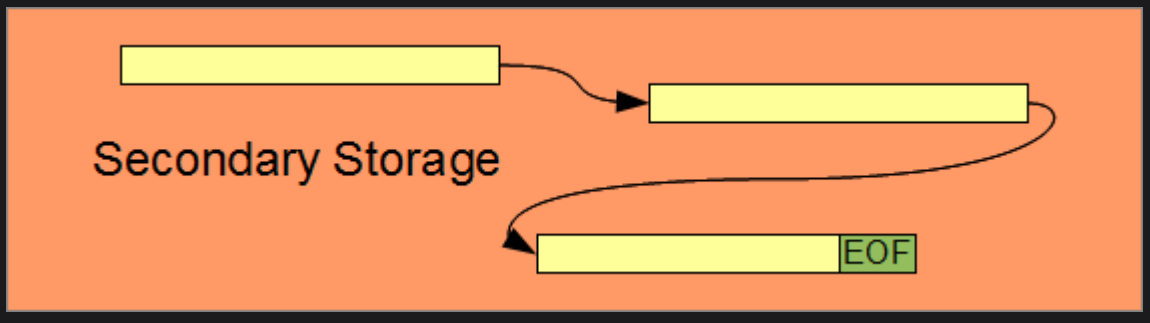
Introduction

Secondary storage retains its information when a computer is turned off and provides a mechanism for holding information beyond the execution of a program. Information in secondary storage can be accessed later by the same or a different program. This information resides in secondary memory in the form of files.

This chapter describes how to connect a program to a file, how to store information in that file and how to retrieve that information.

Files

A *file* is a named area of secondary storage. The file may be fragmented; that is, it may consist of several parts stored at different non-contiguous locations in secondary memory. A file does not necessarily occupy contiguous space on the storage device.



The byte is the fundamental storage unit of a file. The distinguishing feature of a file is the end-of-file mark. We refer to this mark as `EOF`. `EOF` typically has the value -1.

Text Format

A file holds information in either of two formats:

- text - readable and editable data
- binary - executable program code (beyond the scope of these notes)

Data stored in text format is suitable for displaying and modifying through a text editor. Files stored in text format are portable across platforms that share the same character set. A common standard is the IEC/ISO 646-1083 Invariant Code Set, which consists of:

- 52 upper and lower case alphabetic characters: A, B, ..., Z, a, b, ..., z
- 10 digits: 0, 1, ..., 9
- space
- null, line feed, carriage return, horizontal tab, vertical tab and form feed: \0, \l, \n, \t, \v, \f
- 29 graphic characters: ! # % ^ & * (_) - + = ~ [ ] ' | \ ; : " { } , . < > / ?

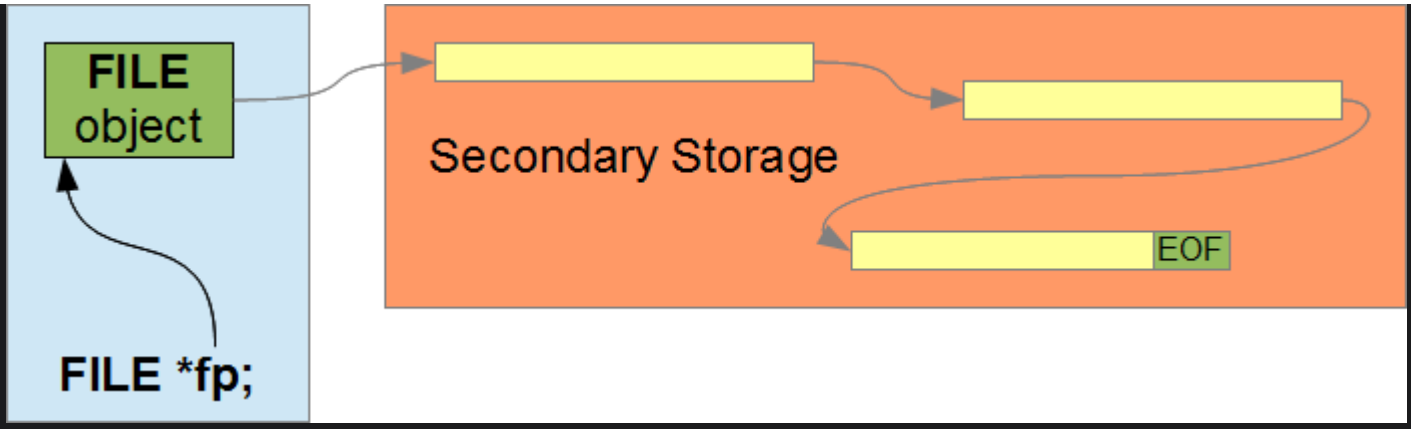
> **NOTE**
>
> This set excludes the $ and ` characters. The encoding for characters like $ and ` does not produce the same characters on all platforms (for more details see National Variants).

Sequential Access

The most common way to access data in a text file is sequentially, byte by byte. We process the file as a stream of bytes without skipping any byte until we reach the file's end-of-file (`EOF`) mark.

Connection

A C program connects to a file through an object of `FILE` type. The object holds information about the file and keeps track of the next position to be accessed. We use a library function to retrieve the address of the file object, store that address in a pointer and subsequently access the file through that pointer.



Allocating a pointer to a `FILE` object takes the form:

```
FILE *identifier;
```

Where `FILE` is the type of the FILE object and identifier is the name of the pointer to the `FILE` object. We call this pointer a handle to the object.

The structure type `FILE` is declared in the `<stdio.h>` header file. To allocate memory for a `FILE` pointer, we write:

```
#include <stdio.h>

FILE *fp = NULL;
```

We initialize the pointer `fp` to `NULL` as a precaution against premature dereferencing. If our program accesses data at `fp` before the connection to the file is open, our program may generate a **segmentation fault**.

> **NOTE**
> NULL is defined in the `<stdio.h>` header file

Opening a File

`fopen()` opens the named file and returns the address of the `FILE` object that connects to that file. The prototype for `fopen()` is:

```
FILE *fopen(const char file_name[], const char mode[]);
```

The first parameter holds the address of the file's name, which could be a ***string literal*** (a set of characters enclosed in a pair of **double quotes**). The second parameter holds the address of the connection mode, which could also be a string literal.

The most common connection modes are:

- **"r"** - read from the file
- **"w"** - write to the file: if the file exists, truncate its contents and then write; if the file does not exist, create a new file and then write to that file
- **"a"** - write to the end of the file: if the file exists, append to the end of the file; if the file does not exist, create it and then write to it

The less common connection modes for text files are:

- **"r+"** - opens the file for reading and possibly writing
- **"w+"** - opens the file for writing and possibly reading; if the file exists, truncates its contents and then writes to the file; if the file does not exist, creates a new file and then writes to that file
- **"a+"** - opens the file for writing to the end of the file and possibly reading; if the file exists, appends to the end of the file; if the file does not exist, creates it and then writes to the file

The mode parameter is enclosed in a pair of double quotes, not single quotes.

To open a file named alpha.txt for writing, we write:

```
// Open a file
// openFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;

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

    if (fp != NULL)
```

```
    {
        // statements to be added later
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

`fopen()` returns `NULL` if it fails to connect to the file. `fopen()` can fail due to lack of permission, premature removal of the secondary storage medium or a full device.

Closing

`fclose()` disconnects the file from the host program. This library function takes as its only parameter the file pointer. The prototype for `fclose()` is:

```
int fclose(FILE *);
```

If the file is open for writing or appending, `fclose()` writes any data remaining in the file's buffer to the file and appends the end of file mark after the last character written. If the file is open for reading, `fclose()` ignores any data left in the file's buffer and closes the connection.

To close a file named `alpha.txt` that is open for writing, we write:

```
// Close an Opened file
// closeFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;

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

    if (fp != NULL)
    {
```

```
        // statements to be added later
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

`fclose()` returns `0` if successful, `EOF` if unsuccessful. `fclose()` fails if the storage device is full, an I/O error occurs or the storage medium is prematurely removed.

Communication

The C library functions for communicating with an open file include:

- `fprintf()` - formatted write to file
- `fputc()` - write single character to file
- `fscanf()` - formatted read from file
- `fgetc()` - read single character from file

Writing

Formatted Writing

`fprintf()` writes data to an open file under format control. The prototype for this library function is:

```
int fprintf(FILE *, const char [], ...);
```

The first parameter receives the address of the `FILE` object. The second parameter receives the address of the string literal that specifies the format. This literal may contain text to be written directly to the file as well as conversion specifiers, if any, to be applied to the data values supplied as arguments.

For example:

```
// Writing to a File
// writeToFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku = 4664;
    double price = 1.49;

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

    if (fp != NULL)
    {
        fprintf(fp, "sku = %d price = %10.2lf\n", sku, price);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }
    return 0;
}
```

Unformatted Writing

`fputc()` writes a single character to an open file. The prototype for this library function is:

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

`ch` receives a copy of the character to be written and `fp` receives the address of the `FILE` object. `fputc()` returns the character written, or `EOF` in the event of an error.

Reading

Formatted Reading

`fscanf()` reads a sequence of bytes from an open file under format control. The prototype for this library function is:

```
int fscanf(FILE *, const char [], ...);
```

The first parameter receives the address of the `FILE` object. The second parameter receives the address of the string literal that specifies the format. This literal contains the conversion specifiers to be used in translating the file data to data stored in memory.

For example:

```
// Reading from a File
// readFromFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    double price;

    fp = fopen("alpha.txt","r");

    if (fp != NULL)
    {
        fscanf(fp, "%d %lf", &sku, &price);
        printf("sku = %d price = %10.2lf\n", sku, price);
        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

Unformatted Reading

`fgetc()` reads a single character from an open file. The prototype for this library function is:

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

`fp` receives the address of the `FILE` object. `fgetc()` returns the character read; `EOF` in the event of an error.

State of a File Object

The C library functions for managing the state of a `FILE` object include:

- `rewind()` - rewind the file
- `feof()` - identify the end of the file

Rewind

`rewind()` resets the record pointer in the `FILE` object to the first byte in a file. The next byte to be accessed by the object will be the first byte in the file.

In other words, to jump to the beginning of a file, instead of disconnecting and re-connecting it, we simply rewind the file. The prototype for this library function is

```
void rewind(FILE *fp);
```

`fp` receives the address of the `FILE` object.

Consider a text file named produce.txt that contains:

```
4664 1.49
4419 1.29
4011 0.59
```

The following program reads and displays this data, rewinds the file and reads and displays again:

```c
// Reading from a file
// readFromFile.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    int sku;
    double price;

    fp = fopen("produce.txt","r");

    if (fp != NULL)
    {
        while( fscanf(fp, "%d%lf", &sku, &price) != EOF)
        {
            printf("%5d %6.2lf\n", sku, price);
        }

        rewind(fp);

        while (fscanf(fp, "%d%lf", &sku, &price) != EOF)
        {
            printf("%5d %6.2lf\n", sku, price);
        }

        fclose(fp);
    }
    else
    {
        printf("Failed to open file\n");
    }

    return 0;
}
```

End of File

`feof()` indicates whether or not the caller attempted to read the end-of-file mark; that is, read beyond the last character in the file. The prototype for this library function is:

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

`feof()` returns false (`0`) if the caller has not attempted to read the end-of-file mark; true if the caller attempted to read the end-of-file mark.

If the next byte to be read is the end-of-file mark, but the caller has not yet read the mark (that is, has only read the last character in the file), `feof()` returns false. In other words, to receive true, the caller must have attempted to read the end-of-file mark at least once.

Comparison

The library functions for communicating with files share many common properties with the functions for communicating with users directly. The functions belong to the same library, follow the same rules for format control and share a common syntax.

Return Type	Standard I/O	File I/O	Notes
`int`	`scanf(...)`	`fscanf(fp, ...)`	check to see if the return value is `EOF`
`int`	`printf(...)`	`fprintf(fp, ...)`	returns the number of characters written
`int`	`getchar()`	`fgetc(fp)`	check to see if the return value is `EOF` **before** converting it to `char` type
`int`	`putchar(ch)`	`fputc(ch, fp)`	check to see if the return value is `EOF`