# CMP 204: Introduction to File Processing (2 Units)

**Objective: Students should understand what a file is, how they are stored in the computer and how to process data files.**

What is a file

## File

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

## File Structure

A File Structure should be according to a required format that the operating system can understand.

- A file has a certain defined structure according to its type.
- A text file is a sequence of characters organized into lines.
- A source file is a sequence of procedures and functions.
- An object file is a sequence of bytes organized into blocks that are understandable by the machine.
- When operating system defines different file structures, it also contains the code to support these file structure. Unix, MS-DOS support minimum number of file structure

## Types of Files

There are three basic types of files:

| | |
|---|---|
| **regular** | Stores data (text, binary, and executable). |
| **directory** | Contains information used to access other files. |
| **special** | Defines a FIFO (first-in, first-out), pipe file or a physical device. |

All file types recognized by the system fall into one of these categories. However, the operating system uses many variations of these basic types.

## Regular Files

Regular files are the most common files. Another name for regular files is ordinary files. Regular files contain data.

### *Text Files*

Text files are regular files that contain information readable by the user. This information is stored in ASCII. You can display and print these files. The lines of a text file must not contain **NUL** characters, and none can exceed **{LINE_MAX}** bytes in length, including the new-line character.

The term *text file* does not prevent the inclusion of control or other nonprintable characters (other than **NUL**). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters gracefully or they explicitly describe their limitations within their individual sections.

### *Binary Files*

Binary files are regular files that contain information readable by the computer. Binary files may be executable files that instruct the system to accomplish a job. Commands and programs are stored in executable, binary files. Special compiling programs translate ASCII text into binary code.

The only difference between text and binary files is that text files have lines of less than **{LINE_MAX}** bytes, with no **NUL** characters, each terminated by a new-line character.

## Directory Files

Directory files contain information the system needs to access all types of files, but they do not contain the actual file data. As a result, directories occupy less space than a regular file and give the file system structure flexibility and depth. Each directory entry represents either a file or a subdirectory. Each entry contains the name of the file and the file's index node reference number (i-node). The i-node points to the unique index node assigned to the file. The i-node describes the location of the data associated with the file. Directories are created and controlled by a separate set of commands.

## Special Files

Special files define devices for the system or temporary files created by processes. There are three basic types of special files: FIFO (first-in, first-out), block, and character. FIFO files are also called pipes. Pipes are created by one process to temporarily allow communication with another process. These files cease to exist when the first process finishes. Block and character files define devices.

Every file has a set of permissions (called access modes) that determine who can read, modify, or execute the file.

# Fundamental File Processing Operations

_ Physical _les and logical _les

_ Opening and closing _les

_ Reading from _les and writing into _les

_ How these operations are done in C and C++

_ Standard input/output and redirection

_ Sample programs for _le manipulation

Reference : Folk, Zoellick and Riccardi. Chapter 2.

### Physical Files and Logical Files

physical file: a collection of bytes stored on a disk or tape

logical file: a "channel" (like a telephone line) that connects the program

to a physical file

- The program (application) sends (or receives) bytes to (from) a file through the logical file. The program knows nothing about where the bytes go (came from).

- The operating system is responsible for associating a logical file n a program to a physical file in disk or tape. Writing to or reading from a file in a program in done through the operating system.

Note that from the program point of view, input devices (keyboard) and output devices (console, printer, etc) are treated as files - places where bytes come from or are sent to.

There may be thousands of physical files on a disk, but a program only have about 20 logical files open at the same time.

The physical file has a name, for instance myfile.tx

The logical file has a logical name used for referring to the file inside the program. This logical name is a variable inside the program, for instance outfile

In C programming language, this variable is declared as follows:

```
FILE * outfile;
```

In C++ the logical name is the name of an object of the class fstream:

```
fstream outfile;
```

In python:

```
>>> fhand = open('mytext.txt')
```

In all languages, the logical name outfile will be associated to the physical file myfile.txt at the time of opening the file as we will see next.

## Opening Files

Opening a file makes it ready for use by the program.

Two options for opening a file :

- open an existing file
- create a new file

When we open a file we are positioned at the beginning of the file.

In C :

```
...
FILE * outfile;
outfile = fopen("myfile.txt", "w");
...
```

The first argument indicates the physical name of the file. The second one determines the \mode", i.e. the way, the file is opened.

For example :

"r" = open for reading,

"w" = open for writing (file need not to exist),

"a" = open for appending (file need not to exist),

among other modes ("r+", "w+", "a+").

In C++ :

```
...
fstream outfile;
```

```
outfile.open("myfile.txt",ios::out);
```

...

The second argument is an integer indicating the mode. Its value is set as a

‘bitwise or" of constants defines in class ios.

In Python

```
>>> fhand = open('myfile.txt','w')
```

## Closing Files

This is like \hanging up" the line connected to a file.

After closing a file, the logical name is free to be associated to another physical file.

Closing a file used for output guarantees everything has been written to the physical file.

We will see later that bytes are not sent directly to the physical file one by one; they are first stored in a buffer to be written later as a block of data.

When the file is closed the leftover from the buffer is ushed to the file.

Files are usually closed automatically by the operating system at the end of program's execution.

It's better to close the file to prevent data loss in case the program does not terminate normally.

In C :

```
fclose(outfile);
```

In C++ :

```
outfile.close();
```

In python:

```
>>> fout.close()
```

## Reading

Read data from a file and place it in a variable inside the program.

Generic Read function (not specific to any programming language)

Read(Source_file, Destination_addr, Size)

Source file = logical name of a file which has been opened

Destination addr = first address of the memory block were data should be stored

Size = number of bytes to be read

In C (or in C++ using C streams) :

```
char c;

FILE * infile;

...

infile = fopen("myfile,"r");

fread(&c,1,1,infile);
```

1st argument: destination address (address of variable c)

2nd argument: element size in bytes (a char occupies 1 byte)

3rd argument: number of elements

4th argument: logical file name

In C++ :

```
char c;

fstream infile;

infile.open("myfile.txt",ios::in);

infile >> c;
```

Note that in the C++ version, the operator >> communicates the same info at a higher level. Since c is a char variable, it's implicit that only 1 byte is to be transferred.

In Python;

```
fhand = open('data.txt')
count = 0
for line in fhand:
     count = count + 1
```

## Writing

Write data from a variable inside the program into the file.

Generic Write function :

Write (Destination_File, Source_addr, Size)

Destination file = logical file name of a file which has been opened

Source addr = first address of the memory block where data is stored

Size = number of bytes to be written

In C (or in C++ using C streams) :

```
char c;

FILE * outfile;

outfile = fopen("mynew.txt","w");

fwrite(&c,1,1,outfile);
```

In C++ :

```
char c;

fstream outfile;

outfile.open("mynew.txt",ios::out);

outfile << c;
```

In Python:

```
>>> fout = open('output.txt', 'w')

>>> line1 = 'This here's thewattle,\n'
>>> fout.write(line1)
```

## Detecting End-of-File

When we try to read and the file has ended, the read was unsuccessful. We can test whether this happened in the following ways :

In C : Check whether fread returned value 0

```
int i;

i = fread(&c,1,1,infile);

if (i==0) // file has ended

...
```

in C++: Check whether infile.fail() returns true

```
infile >> c;

if (infile.fail()) // file has ended

...
```

In python: check if the file `read` call returns an empty string

```
open_file = open("file.txt", "r")
```

```
text = open_file.read()
eof = open_file.read()
if eof == '':
     print('EOF')
```

## Logical file names associated to standard I/O devices and re-direction

| purpose | default meaning | logical name | |
|---|---|---|---|
| | | in C | in C++ |
| Standard Output | Console/Screen | stdout | cout |
| Standard Input | Keyboard | stdin | cin |
| Standard Error | Console/Screen | stderr | cerr |

These streams don't need to be open or closed in the program.

Note that some operating systems allow this default meanings to be changed via a mechanism called redirection.

In UNIX and DOS : (suppose that prog is the executable program)

Input redirection (standard input becomes file `in.txt`)

```
prog < in.txt
```

Output redirection (standard output becomes file out.txt. Note that standard error remains being console)

```
prog > out.txt
```

You can also do : `prog < in.txt > out.txt`

## Sample programs for file manipulation

Next we show programs in C++ to display the contents of a file in the screen:

_ Open file for input (reading)

_ While there are characters to read from the input file :

Read a character from the file

Write the character to the screen

_ Close the input file

```
// listc.cpp
#include <stdio.h>
main() {
char ch;
FILE * infile;
infile = fopen("A.txt","r");
while (fread(&ch,1,1,infile) != 0)
fwrite(&ch,1,1,stdout);
fclose(infile);
}
```

Redirecting output to file called `copy.txt`. Suppose executable file for this program is called `listc.exe`

```
listc.exe > copy.txt
// listcpp.cpp
#include <fstream.h>
main() {
char ch;
fstream infile;
infile.open("A.txt",ios:in);
infile.unsetf(ios::skipws); // include white space in read
infile >> ch;
while (! infile.fail()) {
     cout << ch ;
     infile >> ch ;
     }
infile.close();
}
```

A similar redirection can be done here.

## File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
    - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
    - ***I/O Control*** consists of ***device drivers***, special software programs ( often written in assembly ) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card ( device ) on a system has a different set of addresses ( registers, a.k.a. ***ports*** ) that it listens to, and a unique set of command codes and results codes that it understands.
    - The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, ( e.g. block # 234234 ), or with head-sector-cylinder combinations.
    - The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
    - The ***logical file system*** deals with all of the meta data associated with a file ( UID, GID, mode, dates, etc ), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only

certain layers need to be filesystem specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 ( among 40 others supported. )

REF

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/12_FileSystemImplementation.html