# Fundamental File Processing Operations

## Last Time

Introduction to File Management

## Today

- Physical files and logical files
- Opening and closing files
- Reading from files and writing into files
- How these operations are done in C and C++
- Standard input/output and redirection
- Sample programs for file 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.txt`

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 both 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`.

# 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 flushed 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();
```

# 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.

# 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;
```

## 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
    ...
```

# 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.