2 Fundamental File Processing Operations

Content

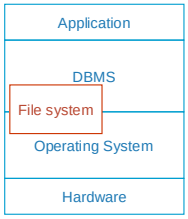
- Sample programs for file manipulation
- Physical files and logical files
- Opening and closing files
- Reading from files and writing into files
- How these operations are done in C and C++
- Standard input/output and redirection

What is a FILE?

A file is...

- A collection of data is placed under permanent or non-volatile storage
- Examples: anything that you can store in a disk, hard drive, tape, optical media, and any other medium which doesn't lose the information when the power is turned off.
- Notice that this is only an informal definition!

Where do File Structures fit in CS?



Physical Files & Logical Files

- Physical file: physically exists on secondary storage; known by the operating system; appears in its file directory
- Logical file, what your program actually uses, a 'pipe' though which information can be extracted, or sent.
- Operating system: get instruction from program or command line; link logical file with physical file or device
- Why is the distinction useful? Why not allow our programs to deal directly with physical files?

Basic File Operations

- **Opening a file** - basically, links a logical file to a physical file.
  - On open, the O/S performs a series operations that end in the program that is trying to open the file being assigned a **file descriptor**.
  - Additionally, the O/S will perform particular operations on the file at the request of the calling program, these operations are intended to 'initialize' the file for use by the program.
  - What happens when the O/S detects an error?

hFile: Logical File, "account.txt": Physical File

```
#include <stdio.h>
int main(){
   FILE *hFile=fopen("account.txt","r");
   char c;
   while (!feof(hFile)){
         fread (&c,sizeof(char),1,hFile) ;
         fwrite(&c,sizeof(char),1,stdout) ;
   }
   fclose(hFile) ;
   return 0;
}
```


FILE

```
typedef struct  {
      unsigned char  *curp;       // Current active pointer
      unsigned char  *buffer;     // Data transfer buffer
      int            level;       // fill/empty level of buffer
      int            bsize;       // Buffer size
      unsigned short  istemp;     // Temporary file indicator
      unsigned short  flags;      // File status flags
      wchar_t        hold;        // Ungetc char if no buffer
      char           fd;          // File descriptor
      unsigned char   token;      // Used for validity checking
}     FILE;
```


C++ Counterpart

```
#include <fstream>
#include <iostream>
using namespace std ;
int main(){
   char c;
   fstream infile ;
   infile.open("account.txt",ios::in) ;
   infile.unsetf(ios::skipws) ;
   infile >> c ;
```



```
   while (! infile.fail()){
      cout << c ;
      infile >> c ;
   }
   infile.close() ;
   return 0;
}
```


Physical Files & Logical Files – Revisited # 1

- OS is responsible for associating a logical file in a program to a physical file in disk or tape. Writing to or reading from a file in a program is done through the OS.
- 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 sent to
- There may be thousands of physical files on a disk, but a program only have a limited number of logical files open at the same time.
- The physical file has a name, for instance "account.txt"
- The logical file has a logical name used for referring to the file inside the program. The logical name is a variable inside the program, for instance "infile"

Physical Files & Logical Files – Revisited # 2

- In C PL, this variable is declared as.

  FILE *infile ;
- In C++ PL, the logical name is the name of an object of the class **fstream**:

  fstream infile ;
- In both languages, the logical name infile will be associated to the physical file "account.txt" at the time of opening the file.

More on Opening Files

- Two options for opening a file:
  - Open an **existing** file
  - Create a **new** file

How to do in C

FILE *outfile;

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

- The 1st argument indicates the physical name of the file
- The 2nd one determines the "mode"

—the way the file is opened

The Mode

- "r": open an existing file for reading
- "w": create a new file, or truncate existing one, for writing
- "a": open a new file, or append an existing one for writing
- "r+": open an existing file for reading and writing
- "w+": create a new file, or truncate an existing one for reading and writing
- "a+": create a new file, or append an existing one for reading and writing

How to do in C++

fstream outfile;

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

- The 1st argument indicates the physical name of the file
- The 2nd argument is an integer indicating the mode defined in the class **ios**.

The Mode

- ios::in open for reading
- ios::out open for writing
- ios::app seek to the end of file before each write
- ios::trunc always create a new file
- ios::nocreate fail if file does not exist
- ios::binary open in binary mode

Basic File Operations

- **Closing a file** - cuts the link between physical and logical files
  - Upon closing, the OS takes care of 'synchronizing' the contents of the file, e.g. often a buffer is used, need to write buffer content to file.
  - In general, files are automatically closed when the program ends.
  - So, why do we need to worry about closing files?
  - In C: **fclose(outfile)**
  - In C++: **outfile.close()**

Basic File Operations

- **Reading and Writing** – basic I/O operations.
  - Usually require three parameters: **a logical file**, **an address**, and the **amount of data** that is to be read or written.
  - What is the use of the **address** parameter?

Reading in C


```
char c ; // a character
char a[100] ; // an array with 100 characters
FILE * infile ;
:
infile = fopen("myfile.txt", "r") ;
fread(&c,1,1,infile) ; // reads one character
fread(a,1,10,infile) ; // reads 10 characters
```


fread()


```
fread(&c,1,1,infile) ; // reads one character
fread(a,1,10,infile) ; // reads 10 characters
```

- 1st argument: destination address
- 2nd argument: element size in bytes
- 3rd argument: number of elements
- 4th argument: logical file name

Reading in C++


```
char c ; // a character
char a[100] ; // an array with 100 characters
fstream infile ;
infile.open("myfile.txt", ios::in) ;
infile >> c; // reads one character
infile.read(&c,1) ;
infile.read(a,10); // reads 10 bytes
```

- Note that thanks to operator overloading in C++, operator >> gets the same info at a higher level

Writing in C


```
char c ; // a character
char a[100] ; // an array with 100 characters
FILE * outfile ;
outfile = fopen("myfile.txt", "w") ;
fwrite(&c,1,1,outfile) ; // writes one character
fwrite(a,1,10,outfile) ; // writes 10 characters
```


Writing in C++


```
char c ; // a character
char a[100] ; // an array with 100 characters
fstream outfile ;
outfile.open("myfile.txt", ios::out) ;
outfile << c; // writes one character
outfile.write(&c,1) ;
outfile.write(a,10); // writes 10 bytes
```

Additional File Operations

- Seeking: source file, offset.
- Detecting the end of a file
- Detecting I/O error

Seeking in C

- int fseek(FILE *stream, long offset, int whence);
- Repositions a file pointer on a stream.
- fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by whence.
- Whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows:
  - SEEK_SET 0 File beginning
  - SEEK_CUR 1 Current file pointer position
  - SEEK_END 2 End-of-file

Seeking with C++ Stream Classes

A fstream has 2 file pointers: get pointer & put pointer
(for input) (for output)

file1.seekg (byte_offset, origin); //moves get pointer
file1.seekp (byte_offset, origin); //moves put pointer

origin can be ios::beg (beginning of file)
ios::cur (current position)
ios::end (end of file)

file1.seekg (373, ios::beg); // moves get pointer 373 bytes from
// the beginning of file

Calculating File Size

```
int main(int argc, char* argv[]) {
   FILE *hFile=fopen(argv[1],"r");
   fseek(hFile, 0L, SEEK_END);
   int fileLength = ftell(hFile);
   printf("\nFile size is %d",fileLength) ;
   fclose(hFile) ;
   return 0;
}
```


Detecting End of File

- In C: Check whether fread returned value 0

```
int i ;
i = fread(&c,1,1,infile) ; //attempt to read
if (i==0) // true if file has ended
```

- Alternatively, use the function feof(infile)
- In C++: Check whether infile.fail() returns true

```
infile >> c ;
if (infile.fail()) // true if file has ended
```

- Alternatively, use the function infile.eof()
- Also note that fail() indicates that an operation is unsuccessful, so it is more general than just checking for end of file

Logical File Names Associated to Std IO Devices

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 do not need to be open or closed in the program

Redirection

- Some OS allow the default meanings to be changed through a mechanism called redirection
- Example in Unix
  - 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)
    - prog > out.txt
  - You can also do
    - prog < in.txt > out.txt