Section 6
Program Structure

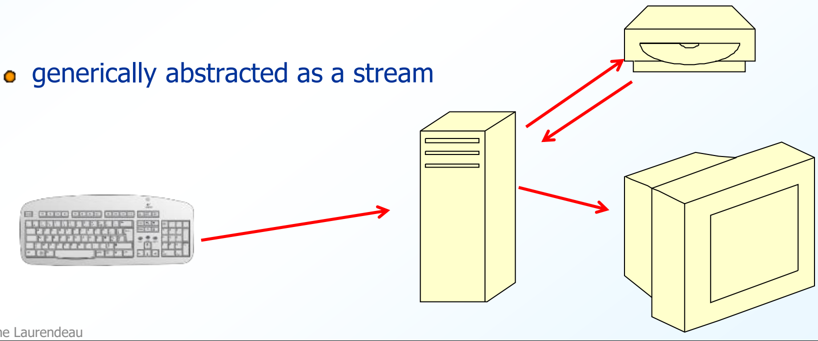
1. Input/Output
2. Procedural program design
3. Program organization
4. Using libraries

Section 6.1
Input / Output

1. Overview
2. Streams
3. Buffers
4. Sources and sinks

6.1.1 Overview

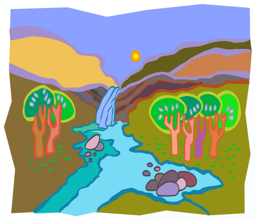
- What is I/O?
  - transfer of bytes
    - from a data *source* to a program
    - from a program to a data *sink*
  - generically abstracted as a stream



Overview (cont.)

- Characteristics of I/O operations
  - use temporary storage called buffers
  - supported by
    - C standard library functions and systems calls
    - Unix shell functions

6.1.2 Streams

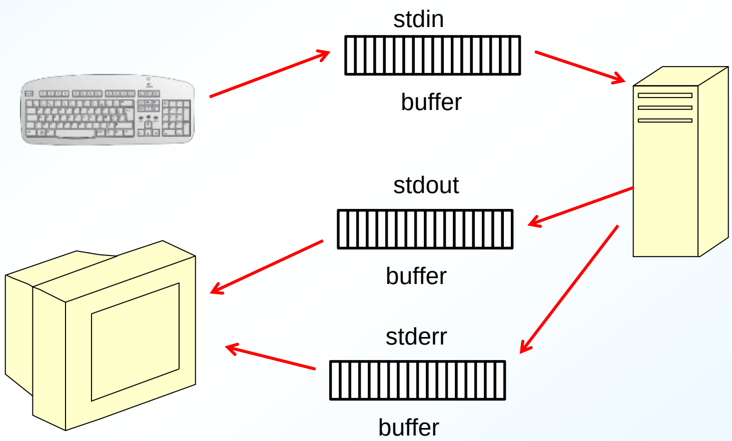


- What is a stream?
  - a sequence of bytes
  - the flow of data
    - from a source to program memory
    - from program memory to a sink
  - data sinks and sources
    - keyboard, console
    - files
    - other programs!
    - devices: printers, network adapters, etc ...

Standard Streams

- Standard streams
  - standard input: **`stdin`** **`(== 0)`**
    - by default, connected to keyboard
  - standard output: **`stdout`** **`(== 1)`**
    - by default, connected to display
  - standard error: **`stderr`** **`(== 2)`**
    - by default, connected to display
- Standard streams can be redirected
  - to / from other programs, files, devices, data sources and sinks
  - Standard streams are opened automatically by the system

Standard Streams



Characteristics of Streams

- Characteristics of streams
  - two types of input and output
    - formatted data – ASCII or text based
    - unformatted data – binary based files
- Characteristics of input streams
  - end-of-file marker
    - OS dependent
    - returned by some input library functions
- User streams must be opened by the user
  - Input
  - Output
  - Input and output

File Types

C has two main types:

- Binary Files
  - Everything stored as 0's and 1's
- Formatted/Text/ASCII Files
  - Usually human readable characters
  - Each data line ends with newline char

Types of steams/files

- **Formatted Streams/Files**
  - Store the data in readable ascii format (usually a single byte)
  - Sequential access
  - Store line by line
- **Pros**
  - Good for text
  - Readable (see what is stored)
  - Data can be recovered
  - High compression ratio
- **Cons**
  - Storage space
  - Slow
  - Sequential access
  - Every field of data must be written
  - Hard to navigate

- **Binary Streams/Files**
  - Store the data in machine representation (e.g., a long integer will be stored as 4 bytes regardless of value).
- **Pros**
  - Random access to data
  - Fast
  - Read/write whole records
  - Easy to navigate (random access)
  - good for all data
- **Cons**
  - Low compression ratio
  - Hard to recover data
  - Low compression ratio
  - Reading through a special program

Stream/File Handling

- Stream is handle - FILE
- Defined in stdio.h
- Declared variable as a pointer:

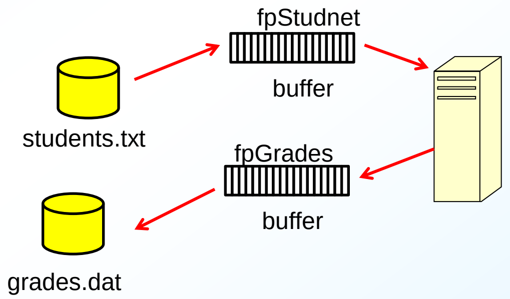
```
FILE *fileHandle;
```


A FILE

FILE is the type we use to reference "files"

- Defined in stdio.h
- Usually declare variable as a pointer:

```
FILE *fpStudent;
FILE *fpGrades;
```



buffer

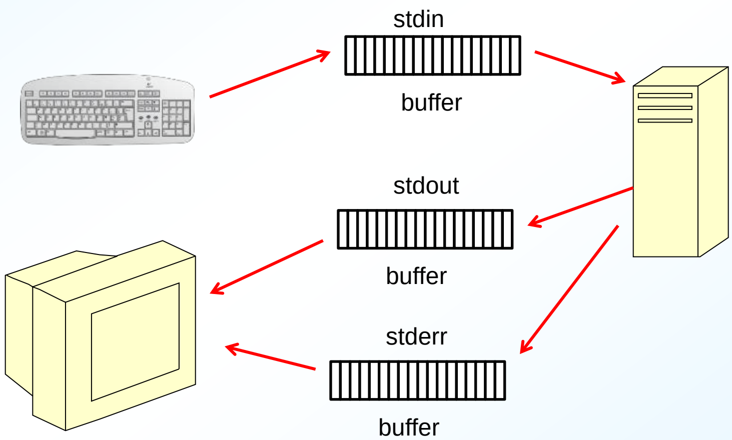
User Files (last)

Standard Files are opened automatically

User files must be opened by the user

- Can be opened for input or output
- One file one stream (or vice versa)

Standard Streams



Stream Library Functions

- Stream management
  - `fopen – fopen("filename","mode");`
    - establishes a connection between a program and a stream
    - second parameter indicates mode
    - Returns a pointer to `FILE`, a Physical file
    - Automatically creates buffer
  - `fclose – fclose(FIlE *handle)`
    - breaks the stream connection

Mode	Meaning
r	Open file for reading • If file exists, the marker is positioned at beginning • If file doesn't exist, file handle is NULL
w	Open text file for writing • If file exists, it is emptied. Beware !!! • If file doesn't exist, it is created.
a	Open text file for append • If file exists, the marker is positioned at end. • If file doesn't exist, it is created.

Examples Open/Close a stream

```
FILE *fpTemp;
fpTemp = fopen("student.txt","w");
```

- Returns `NULL` if there is an error

```
if((fpTemp=fopen("T.DAT","w"))==NULL){
...
```

```
fclose(fpTemp);
```

- Returns `EOF` (end of file constant) if an error occurs

```
if(fclose(fpTemp) == EOF) { …
```


NULL is not always an error Checking if a file exists

```
#include <stdio.h>

int main()
{
    FILE *fid = NULL;

    fid = fopen("file.txt", "r")
    if (fid == NULL) {
            /* file does not exist
            decide what to do; */
    } else {
            //  file exist
            // decide what to do
            …
    }
    return 0;
}
```

- Why do it?
  - Warn the user that the file is about to be erased
  - Ask the user to locate the file if it not there

Stream Library Functions (cont.)

- Formatted I/O
- **fscanf – fscanf(FILE *, "format string", variables);**
  - reads from a stream into program variables according to format string
  - Returns the number of correctly read values
  - Similar to scanf()
  - Moves the file marker forward (amounts depends on what was read)
- **fprintf – fprintf(FILE *, "format string", variables);**
  - writes to a stream from program variables according to format string
  - Returns the number of characters that were written to file
  - <0 if an error occured

Other helpful functions - Error checking

- ferror(FILE *fid)
  - Checks if the error flag is set for the file
- fclear(FILE *fid)
  - Clears the error flag associated with the file
- feof(FILE *fid)
  - Checks if end of file was reached

Stream Library Functions (cont.)

- Unformatted I/O
  - `fread`
    - reads from a stream into one program variable
  - `fwrite`
    - writes to a stream from one program variable
- Leads to handling binary files

Binary files

- A permanent storage of data which is kept in the format of the hardware.
- A "mirror" image of the memory of the computer
- Purpose
  - Provide a copy of the memory for later usage
  - Transferring data from one program to another
  - Recovery in cases where computation has failed (checkpoints).

Binary files opening

- Similar to text files
- FILE *fopen(char *filename, char *mode);

Mode	Meaning
rb	Open file for reading •If file exists, the marker is positioned at beginning
wb	Open file for writing •If file exists, it is truncated. Beware !!! •If file doesn't exist, it is created.
ab	Open text file for append •If file exists, the marker is positioned at end. •If file doesn't exist, it is created.

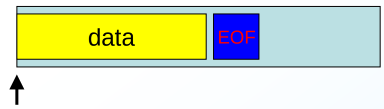
Binary files opening (last)

- Using the "+" character means opened for read and write

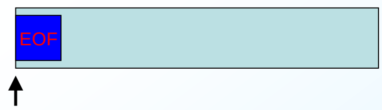
Mode	Meaning
r+b	Open file for reading •If file exists, the marker is positioned at beginning
w+b	Open file for writing •If file exists, it is truncated. Beware !!! •If file doesn't exist, it is created.
a+b	Open text file for append •If file exists, the marker is positioned at end. •If file doesn't exist, it is created.

fopen(fileName, mode)

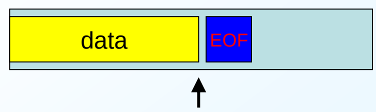
- rb r+b –
  - file is positioned at the beginning



- wb wb+ –
  - file is positioned at the beginning



- ab a+b –
  - file is positioned at the end



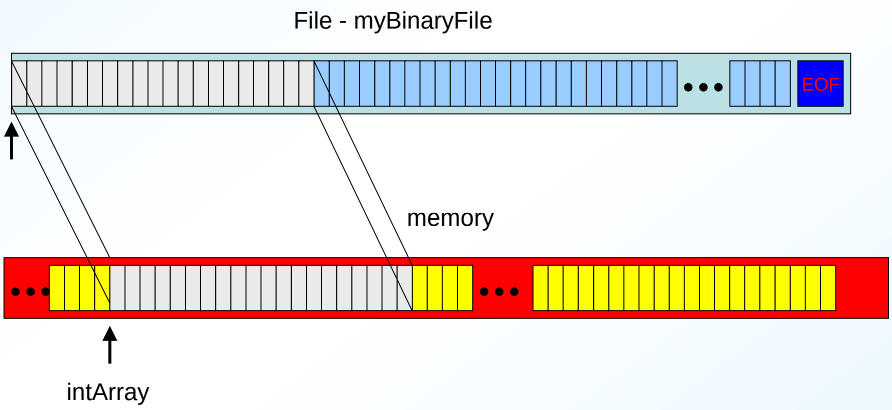
Binary file organization

- File is organized like the memory as a byte stream.
  - First byte in the file is the at position 0.
- File is termed binary file because it stores the binary representation of numbers
  - E.g., 255 will be stored in a single byte as 0xFF
  - In text file it will be stored in three bytes '2"5"5' (0x32 0x35 0x35)
- Information in the file mimics the internal memory.
  - Data is copied to an from the file without translation/conversion.
- Data in the file is meaningful only to the program that reads or writes to the file.
- When opening a file the "b" in the mode instructs the OS not to translate the data in the file (e.g. \n).

Reading from a binary file

- Reads from the file directly into memory
- int fread(void *buffer, int recSize, int numRec, FILE *fid);
  - Reads numRec records each of size recSize into the memory location pointed to by buffer
  - buffer size >= numRec * recSize
- Return value -
  - the number of records that were successfully read.

fread(intArray, sizeof(int), 5, fin)

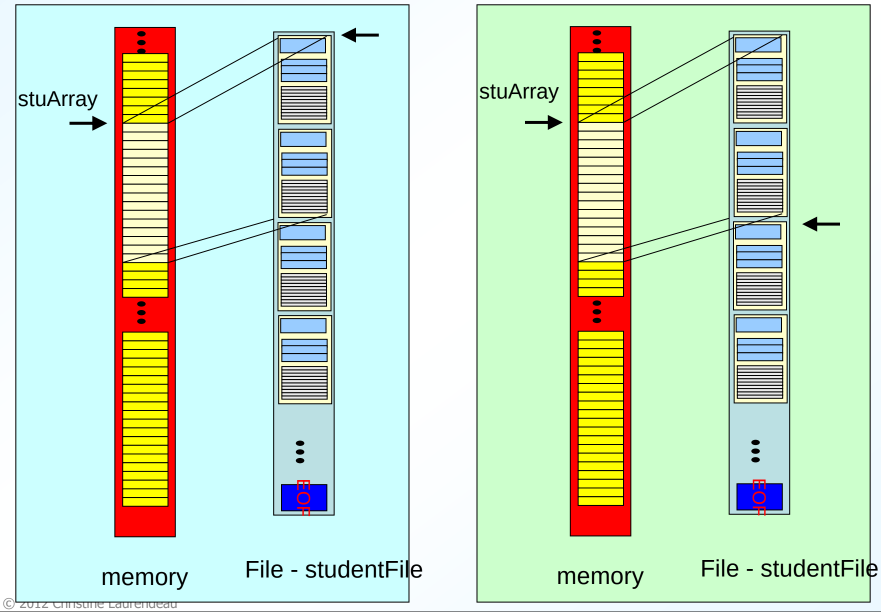


```
FILE *fin;  /* input file */
int rc = 0;                  /* return code */
int numRec = 5;
int numRead = 0;
int intArray[5];

fin = fopen(“myBinaryFile”, “rb”);
if (fin == NULL) {
     /* handle error */

}
…
While ((numRead = fread(intArray, sizeof(int), NumRec, fin)) != 0) {
     /* process the integers */
     for (i = 0; i < numRead; i++ ) {
              …
     }

…
}
```



```
struct student {
     long stId;
     short courses[3]
     char name[10];
}

…

FILE *fin;  /* input file */
int numRead = 0;
struct student stuArray[5];

fin = fopen(“studentFile”, “rb”);
if (fin == NULL) {
     /* handle error */

}

numRead = fread(stuArray, sizeof(struct student), 2, fin);

/* process the records */

…

}
```

fread(stuArray, sizeof(struct student), 2, fin);

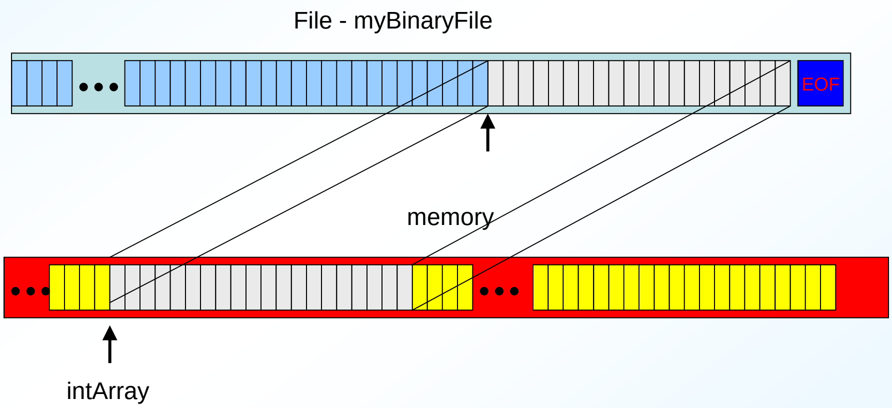


Writing to a binary file

int fwrite(void *buffer, int recSize, int numRec, FILE *fid);

- Writes/copies from the memory directly into the file
- Writes
  - numRec records
  - each of size recSize into the file from the memory location pointed to by buffer
- Return value -
  - the number of records that were successfully written.

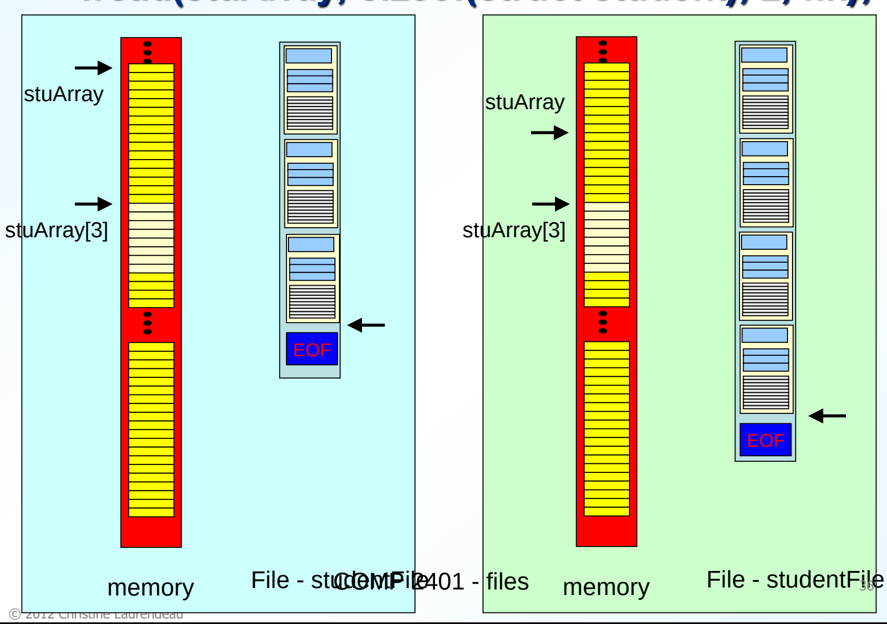
fwrite(intArray, sizeof(int), 5, fin)




```
FILE *fout;            /* output file */
int rc = 0;            /* return code */
int numRec = 5;
int numWritten = 0;
int intArray[500];

fout = fopen(“myBinaryFile”, “wb”);
if (fin == NULL) {
     /* handle error */

}
…
If ((numWritten = fwrite(intArray, sizeof(int), NumRec, fin)) != NumRec)  {
     /* handle the error */
     …
}
```

```
struct student {
      long stId;
      short courses[3]
      char name[10];
}

…

FILE *fout;              /* input file */
int numWritten = 0;
struct student stuArray[5];

fout = fopen(“studentFile”, “wb”);
if (fout == NULL) {
      /* handle error */

}

numWritten = fread(&stuArray[3], sizeof(struct student), 1, fout);

/* process the records */

…

}
```


fread(stuArray, sizeof(struct student), 2, fin);



File Navigation/positioning

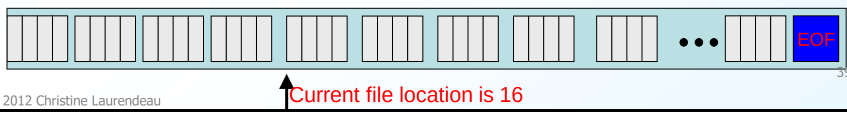
- Access data in file in a similar way to memory
  - Random access -
    - we can jump around in memory without difficulties
    - Examples: arr[5], *(arr+5), p->data
  - Ability to read/write from memory regardless if data is fully available
- Position/location in the file is measured in bytes
  - First byte in the file is at position 0
- In contrast to memory files **do not** have any notion of structures or size of the different data types.

File Navigation/positioning

- ftell –
  - Tells the program where the file marker is positioned
- fseek –
  - move the file marker to a particular location in the file.
- rewind
  - Moves the file marker to the beginning of the file

long ftell(FILE *fp);

- Tells the program where the file marker is positioned
- Returns
  - the number of bytes from the beginning of the file. Note the limitation on the file size (long)
  - -1 if an error has occurred
- Example:
  - numBytes = ftell(fp);
  - numInts = numBytes / sizeof(int);
  - numStudents = numBytes/sizeof(struct student)



int fseek(FILE *fp, long offset, int whereFrom);

- Purpose
  - Position the file marker in the desired location.
  - Moves the physical reading arm/head of the disk
- All changes in positions are relative to one of three options
  - Beginning of file
  - End of file
  - Current location of the file marker
- Parameters
  - fp – a handle to the file
  - Offset – the number of bytes that the head must be moved
  - whereFrom – which of the three relations to use when moving the arm (beginning of file, end of file, or current location)
- Return
  - 0 if operation was succesful
  - Non 0 othewise

Relative positions – SEEK_SET

- The offset is measured from the beginning of the file
- #define SEEK_SET 0
- Example
  - fseek(fp, 34, SEEK_SET)
  - Positions the file marker to byte number 35
- Example
  - Position the file marker at the 4th student in the file
  - Struct student(
    Long id;
    Long telehpone
    Short courses[3];
    }
  - fseek(fp, 3*sizeof(struct student), SEEK_SET)
  - Positions the file marker to the byte 42 which is also the beginning of the 4th student record in the file.
  - Note that we had to compute the location of the of the fourth record

Relative positions – SEEK_CUR

- The offset is measured from the current location of the file marker
- #define SEEK_CUR 1
- Allows movement to the "leff" or "right" of marker (towards the beginning of the file or towards the end of the file)
  - A positive offset moves the marker towards or beyond the end of the file
  - A negative offset moves the marker towards the beginning of the file
- Note –
  - it is an error to move beyond the beginning of the file
  - It is legal to move beyond the end of the file (assuming that there is space)

Relative positions – SEEK_CUR

- Example
  - Position the file marker at the beginning of the next student record
  - Struct student(
    Long id;
    Long telehpone
    Short courses[3];
    }
  - fseek(fp, sizeof(struct student), SEEK_CUR)
- Example
  - Position the file marker 2 records before current record
  - fseek(fp, -2*sizeof(struct student), SEEK_CUR)

Relative positions – SEEK_END

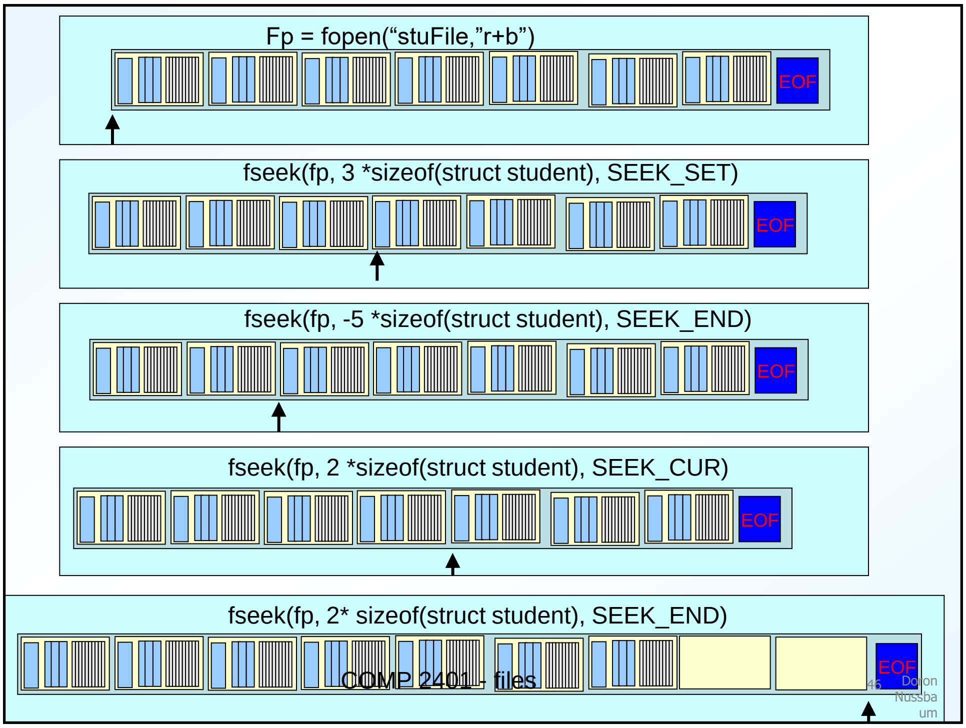
- The offset is measured from the end of the file
- #define SEEK_END 2
- Allows movement to the "leff" or "right" of the end of the file (towards the beginning of the file or beyond the end of the file)
  - A positive offset moves the marker towards or beyond the end of the file
  - A negative offset moves the marker towards the end of the file

Relative positions – SEEK_END

- Example
  - Positions the file marker at the end of the file
  - fseek(fp, 0, SEEK_END)
- Example
  - Position the file marker at the last student record in the file
  - Struct student(

```
Long id;
Long telehpone
Short courses[3];
 }
```

  - fseek(fp, -sizeof(struct student), SEEK_END)
- Example
  - Find how many students records are stored in the file
  - fseek(fp,0, SEEK_END)
  - numBytes = ftell(fp);
  - if (numBytes != -1) numStudents = numBytes/sizeof(struct student)
  - fseek(fp, -sizeof(struct student), SEEK_END)



void rewind(FILE *fp)

- Purpose
  - Moves the marker to the beginning of the file
- Same as fseek(fp, 0, SEEK_SET)

Other functions

- int remove(char *filename)
  - Purpose – deletes the file
  - Return – 0 if file was deleted
  - Example
  - if (remove("myFile") {

```
    /* handle error */
    Printf("Error could not delete file \n");
  }
```

- int rename(char *oldFileName, char *newFileName)
  - Purpose – renames the file. It is being saved as a new version
  - Return – 0 if file was renamed
  - Example
  - if (rename("myFileVer1.txt", "myFileVer2.txt) {

```
    /* handle error */
    Printf("Error could not rename file \n");
  }
```

Stream System Calls

- System calls
  - calls to OS to perform a task
    - not the same as library functions!
    - ... more on this later...
- Stream system calls
  - **`open`, `close`, `read`, `write`**
  - OS dependent
    - not standardized
    - use standard library functions (f-versions) instead

6.1.3 Buffers

- What is a buffer?
  - temporary storage of bytes on a stream
- Purpose of buffering
  - to regulate the data flow
    - if receiver is not ready or can't handle sender throughput
  - to optimize the data flow
    - minimize the number of costly operations
      - e.g. access to secondary storage

Buffers (cont.)

- Types of buffering
  - block buffering
  - line buffering
  - no buffering
- What's the difference between these?
  - when the buffer is *flushed*

Flushing a Buffer

- What is flushing?
  - *pushing* the bytes already in the buffer to the stream
    - the buffer is emptied
- Why do we need to flush?
  - we are done
  - we want all the information to reach the stream

Flushing a Buffer (cont.)

- How do we flush a buffer?
  - implicit flushing
    - when the stream is closed
  - explicit flushing
    - **`fflush`** library function

Block Buffering

- What is block buffering?
  - fixed size buffer
  - buffer accumulates bytes until it is full
  - when full, buffer is automatically flushed
    - it can be explicitly flushed any time
- Use of block buffering
  - example: large data transfers

Line Buffering

- What is line buffering?
  - buffer accumulates bytes until the new line character is added
  - content of buffer can change
  - when new line is added, buffer is automatically flushed
- Use of line buffering
  - example: entering shell commands

Unbuffered I/O

- What is unbuffered I/O?
  - buffer does not accumulate bytes
  - each byte is automatically flushed as it is read
  - receiver gets each byte in real time
- Use of unbuffered I/O
  - example: applications where each key press is processed

6.1.4 Sources and Sinks

- Files
- Pipes
- Devices

Files

- What is a file?
  - a stream stored in non-volatile storage
- Characteristics of files
  - a one-dimensional array of bytes
  - used to store any type of data
    - program interprets the data in the file

Files (cont.)

- Working with files
  - end of file marker
    - follows last byte in the file
    - value is OS dependent
  - file pointer
    - position in stream where next byte read from or written to
    - incremented on every read/write
    - queried with library function **`ftell`**
    - explicitly set with library function **`fseek`**

Pipes

- What is a pipe?
  - connects a stream between alternate sources and sinks
    - program standard input/output streams can be *redirected*
- What is *pipelining*?
  - action of redirecting streams
    - performed on shell command line
- What is pipeline chaining?
  - multiple stream redirections

Pipes (cont.)

- Pipelining symbols
  - <
    - uses specified file as program **`stdin`**
  - >
    - redirects program **`stdout`** to specified file
  - |
    - redirects **`stdout`** from one program to **`stdin`** of another program

Pipes (cont.)

- Use of pipelining example: program testing
  - redirect input from file into program being tested
  - compare program's actual output to expected output
  - both input and expected output can be stored in files

Devices

- What is a device?
  - piece of hardware
- Characteristics of devices
  - abstracted as a stream with file name in **`/dev`** directory
  - device I/O treated as file I/O
  - device drivers provide device management functions
    - **`open`**, **`close`**, **`read`**, **`write`**, etc.