FILE INPUT/OUTPUT

Gradebook

```
Input:
scanf…
```

```
Your C
Code
```

```
Find id:…
```

```
typedef struct student
{int  sid;
 char sname[20];
 double cgpa;} STUDENT;

STUDENT 140_students[MAX];
```

```
Info for id:
Name:
CGPA:
```

- All fine, your data is in the array `140_students`
- What happens when you stop the program ?
- Information stored on **auxiliary devices** is arranged in the form of files
  - To read, process and save data required for our program

Auxiliary Devices

Magnetic tapes - cartridges

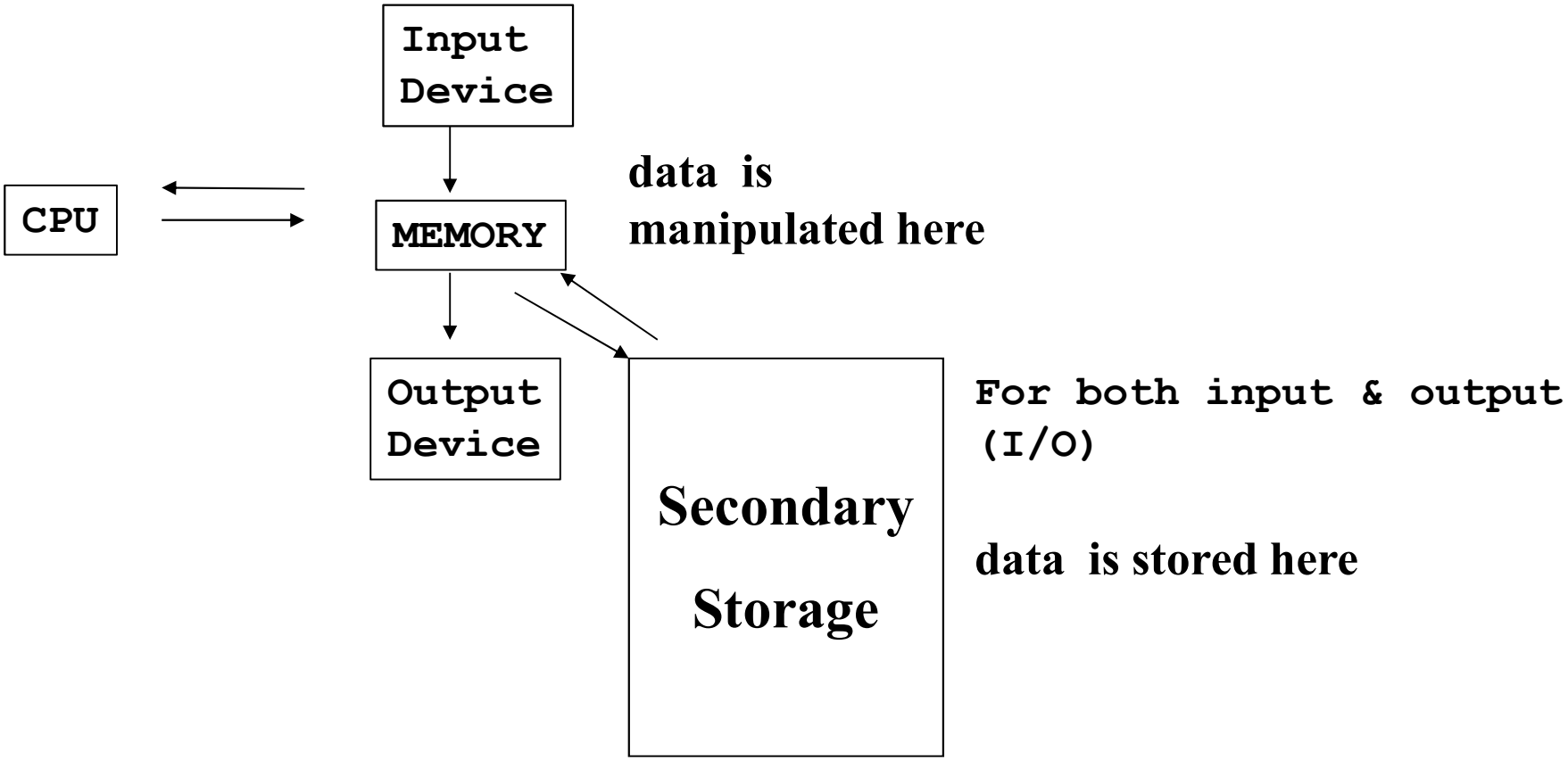
Floppy disk

Magnetic disk

CD/DVD

SSD

Principal Components of a Computer System



Streams

- In C, the term **stream** means any source of input or any destination for output.
  - It connects file to program and passes block of data in both directions. So it is independent of devices which we are using. A source can be :
    - A file, hard disk or CD, DVD, I/O devices etc.

0101010000111010101011101010101010000111101010 01…

So, we process **streams** of **bytes** that are coming from/going to any device or file!

Standard Streams

- <stdio.h> provides three standard streams

File Pointer	*Stream*	*Default Meaning*
stdin	Standard input	Keyboard
stdout	Standard output	Screen
stderr	Standard error	Screen

- These streams are ready to use - we don't declare them, and we don't open or close them.

Redirection

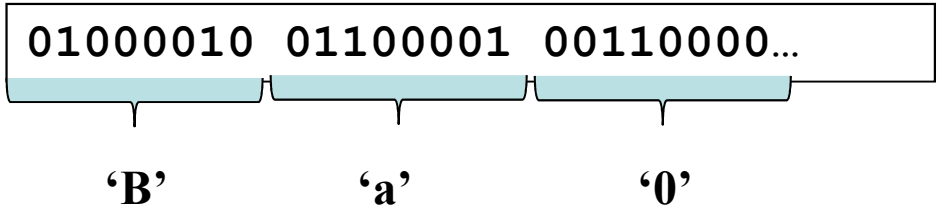
- Unix allows changing of default meanings through **redirection**.
- **Input redirection** forces a program to obtain its input from a file instead of from the keyboard:
  - demo < in.dat
- **Output redirection** is similar:
  - demo > out.dat
  - All data written to stdout will now go into the out.dat file instead of appearing on the screen.

Files & Streams

- File: a stream of bytes
  - Text stream (text file)
  - Binary stream (binary file)
- A *text stream* consists of **lines** of **characters** terminated by the **newline** character.
- A *binary stream* consists of a sequence of bytes (characters).
  - Newline has no special significance

Text Files vs Binary Files

- <stdio.h> supports two kinds of files:
- **Text file:** a sequence of bytes that represent characters, allowing human to examine or edit the file.
  - E.g., the source code for a C program.



- **Binary file:** bytes don't necessarily represent characters.
  - Group of bytes might represent other types of data, such as integers and floating-point numbers.
  - E.g., the executable C program.

Text Files vs Binary Files

- Under Windows, there are two differences between text streams and binary streams:
  - When a new-line character is written to a text file, it is expanded into a carriage-return/line-feed pair. The reverse translation takes place during input.
  - A control-Z character (\x1a) in an input file is assumed to mark the end of the file.
- Under UNIX, there is no difference between a text stream and a binary stream.

FILE structure

- A stream is asociated with a file (or, **device**) by **opening** it.
  - When a file is opened, a FILE structure is associated with it.
- FILE holds the following necessary for controlling a stream:
  - the current position in the file, error indicator, end-of-file indicator, pointer to the associated buffer
- When a file is opened, a pointer to this FILE structure is returned to be used in further operations
  - After the processing is finished, file should be closed.

FILE structure

- A file pointer is a pointer to a FILE structure (defined in <stdio.h>)
- A program may declare as many file pointers as needed:
  - FILE *fp1, *fp2;

- **A file pointer** represents a **stream**, which may be **a file** or—in general— **<u>any source of</u> input or output**.

Opening and Closing Files in C

- FILE* fopen(const char *filename, const char * filemode)
- filemode can be:
  - “r” → Open file for reading.
  - “w” → Open file for writing. Delete old contents if it exists already.
  - “a” → Create a new file or append to the existing one.
  - “r+”, “w+”, “a+” → input & output
  - An additional “b” can be appended to the file mode for binary I/O.
- int fclose(FILE *filepointer)
- int fflush(FILE *filepointer)

Opening and Closing Files

- Example:

```
#include <stdio.h>
#include <stdlib.h>
#define INPUT_FILE "example.dat"
int main(void)
{
FILE *fp;
fp = fopen(INPUT_FILE, "r");
if (fp == NULL) {
  printf("Can't open %s\n", INPUT_FILE);
  exit(EXIT_FAILURE);
}
…
fclose(fp);
return 0;
}
```

Formatted I/O

- Reading - the number of matches or EOF
  - `int fscanf(FILE * fp, " ", variableList);`
    - scanf always reads from **stdin,** whereas **fscanf** reads to the stream indicated by its first argument.
- Writing - returns number of chars written
  - `int fprintf(FILE * fp, " ", variableList);`
    - printf always writes to **stdout,** whereas **fprintf** writes to the stream indicated by its first argument.

Character I/O

- Reading - returns char read or EOF
  - `int fgetc(FILE * fp);`
  - `int getc(FILE * fp);`
  - `int getchar() <-> int fgetc(stdin)`
- Writing - returns char written
  - `int fputc(int c, FILE * fp);`
  - `int putc(int c, FILE * fp);`
  - A typical while loop for reading characters from a file:
    - `while((ch = getc(fp)) != EOF);`

Line I/O

- Reading - returns pointer to string read, **NULL** if end of the file
  - `char* fgets(char *buf, int max, FILE *fp);`
- Strings are character arrays in C
- max indicates the maximum number of characters to be read.
- Writing - returns numbers of chars written
  - `int fputs(char *buf, FILE *fp);`

Text I/O (Sequential Access) Input Functions

- The standard I/O library provides a number of functions and macros for **reading character data**. Here is a partial list:

getc(fp) Reads a character from **fp** (macro)

getchar() Reads a character from **stdin** (macro)

fgetc(fp) Similar to getc, but is a function

scanf(format, ...) Reads formatted input from **stdin**

fscanf(fp, format, ...) Reads formatted input from **fp**

gets(s) Reads characters from **stdin** up to next newline

fgets(s, n, fp) Reads at most n – 1 characters from **fp**

Text I/O (Sequential I/O) Output Functions

- Here is a partial list of **output functions**:

putc(c, fp) Writes the character c to **fp** (macro)

putchar(c) Writes the character c to **stdout** (macro)

fputc(c, fp) Similar to putc, but is a function

printf(format, ...) Writes formatted output to **stdout**

fprintf(fp, format, ...) Writes formatted output to **fp**

puts(s) Writes the string s to **stdout**

fputs(s, fp) Writes the string s to **fp**

Text I/O

• Watch out for small differences between similar functions:
puts always adds a new-line character to the end of its output; fputs doesn't.
fgets always includes a new-line character at the end of its input string; gets doesn't.

• All output functions return a value:
putc, **putchar**, and **fputc** return the character written.
printf and **fprintf** return the number of bytes written.
puts and **fputs** return the last character written.

All seven functions return EOF (a macro defined in <stdio.h>) if an error occurs during output.

Detecting End-of-File

- When getc, getchar, or fgetc detects that the end of the input file has been reached or that an input error has occurred, it returns EOF. *Note*: All three return an integer, not a character.

- The feof function can confirm that end-of-file was actually reached:

```
int feof(FILE *stream); /* returns nonzero if eof */
```

- The ferror function can confirm that an error occurred:

```
int ferror(FILE *stream); /* returns nonzero if
error */
```

Detecting End-of-File

- The value returned by **scanf** and **fscanf** indicates the actual number of input items that were read. If end-of-file occurred before even one item could be read, the return value is EOF.

- **gets** and **fgets** return a pointer to the string read; on error or end-of-file, they return NULL.

Binary Streams

- Whether a file is treated as a text file or a binary file depends on the **mode** that was specified when it was opened. The modes listed previously are used for text files.

When a binary file is opened, the mode string should include the letter b:

"**rb**" Open for reading
"**wb**" Open for writing (file need not exist)
"**ab**" Open for appending (file need not exist)
"**rb+**" Open for reading and writing, starting at beginning
"**wb+**" Open for reading and writing (truncate if file exists)
"**ab+**" Open for reading and writing (append if file exists)

Binary I/O (Random Access)

- The **fwrite** function writes a block of **binary data**:

```
size_t fwrite(const void *ptr, size_t size, size_t
nmemb, FILE *stream);
```

fwrite writes nmemb elements of size size stored at the address specified by ptr to stream. fwrite returns the actual number of elements written.

- The **fread** function reads a block of **binary data:**

```
size_t fread(void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

fread reads up to nmemb elements of size size from stream, storing them at the address specified by ptr. fread returns the actual number of elements read.

Binary I/O (Random Access)

- Numeric data written using fprintf is converted to character form. Numeric data written using fwrite is left in binary form.

- Although fread and fwrite are **normally** applied to binary streams, they can be used with text streams as well. Conversely, the text I/O functions can be used with binary streams. In both cases, however, watch out for unexpected results.

Binary I/O (Random Access)

- The **`fseek`**, **`ftell`**, and **`rewind`** functions support random access within files. **Random access is most often used with binary files.** These functions can also be used with text files, but some restrictions apply.

- fseek allows **repositioning** within a file.

```
int fseek(FILE *stream, long int offset, int
whence);
```

The new file position is determined by offset and whence. offset is a (possibly negative) byte count relative to the position specified by whence. whence must have one of the following values:

```
SEEK_SET Beginning of file
SEEK_CUR Current file position
SEEK_END End of file
```

Random Access

- Examples of fseek:

```
fseek(fp, 0, SEEK_SET); /* move to beginning of file */
fseek(fp, 0, SEEK_END); /* move to end of file */
fseek(fp, -10, SEEK_CUR); /* move back 10 bytes */
```

- ftell returns the current file position:

```
long int ftell(FILE *stream);
```

This may be saved and later supplied to a call of fseek:

```
long int file_pos;
file_pos = ftell(fp);
…
fseek(fp, file_pos, SEEK_SET);
/* return to previous position */
```

- The call rewind(fp) is equivalent to `fseek(fp, 0, SEEK_SET)`.

Gradebook File: Text

Each is a character

```
1 Ecem 4.00
2 Farrukh 3.90
3 Moustafa 3.95
4 Ceyda 3.85
5 Arda 3.87
…
```

Gradebook.txt

```
typedef struct student
{int  sid;
 char sname[20];
 double cgpa;} STUDENT;

STUDENT ceng140_students[MAX];
```

```
FILE *out;
out = fopen("Gradebook.txt", "w");

// write each line with printf
for (i=0; i<MAX; i++)
fprintf(out, "%d %s %.2f\n", ceng140_students[i].sid,
                             ceng140_students[i].sname,
                             ceng140_students[i].cgpa);
```

Gradebook File: Binary

```
typedef struct student
{int  sid;
 char sname[20];
 double cgpa;} STUDENT;

STUDENT ceng140_students[MAX]
```

```
int - 4 bytes Array of 20 characters                    double - 8 bytes
00.............000001 00101100111010100010110 0...1110101000.......................000100
01.............000001 00100000101010100010110 0...1110101001.......................000110
11.............000001 00111100111110100010110 0...1110101010.......................000111
```

Gradebook.bin

```
FILE *out, *in;
out = fopen("Gradebook.bin", "wb");

// write each student one by one.. Or ALL once
fwrite(ceng140_students, sizeof(STUDENT ), MAX,  out);
fclose(out);
in = fopen("Gradebook.bin", "rb");
//read the third student data
fseek(in, sizeof(STUDENT)*2 ,SEEK_SET);
fread(&stu, sizeof(STUDENT), 1, in);
```

What is the type of stu?