

Lecture 9:

Text & Binary Files Processing in C

Content

- I. Introduction
- II. Files and Streams
- III. Major Steps for Processing a File
- IV. File Error Handling
- V. Random Data Access
- Appendix

Check the Notes section

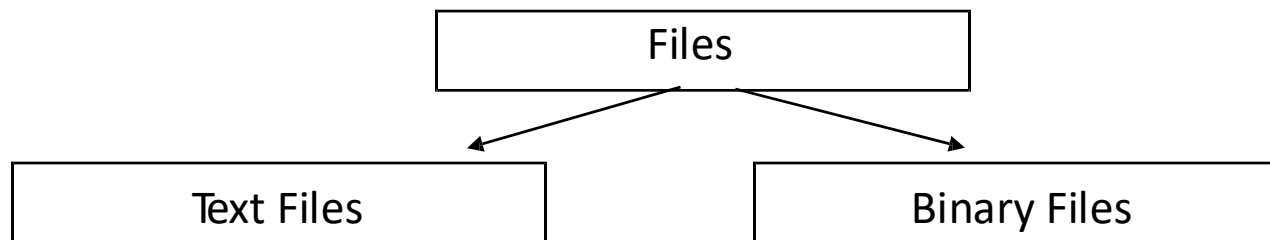


I. Introduction

- Storage of data in variables and arrays is **temporary**, such **data is lost when a program terminates.**
- **Files** are used for **permanent retention of data.**
- Computers store files **on secondary storage devices**, such as hard drives, DVDs and flash drives.
- In this lecture, **we explain how data files are created, updated and processed by C programs.**
- We consider **sequential-access** and **random-access** file processing.

...Continued

- There are many ways how to classify files
 - MATLAB provides a set of built-in functions for accessing common file formats.
 - load, save for MATLAB formatted data in *.mat files
 - csvread, csvwrite – coma delimited numbers in *.csv files
 - xlsread, xlswrite – Microsoft Excel spreadsheet files *.xlsx
- Regardless of the file format, all files are subdivided into two major categories:



- C provides a set of generic functions for these two categories.

Text File Processing

- A text file is a **file of characters**.
- **The basis** of its internal structure is a **character**.
- All characters (letters, digits and special characters) are stored in text files as **ASCII codes**.

Day -> 68 97 121
156 -> 49 53 54
38.098 -> 51 56 46 48 57 56

When content of a text file is displayed, all ASCII codes are interpreted as corresponding characters

	ASCII									
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	del	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	<u>b</u>	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

Example

- Anything wrong with this code?

```
-----  
char arr1[10] = { 1, 3, 5, 7, 12, 34, 11, 63, 7, 0};  
int indx;  
for(indx = 0; indx < 10; indx++)  
    printf("%c ", arr1[indx]);  
-----
```

The output is:

" ?

Why? ...

The numbers are interpreted as ASCII codes, the characters they present will be printed.

...Continued

```
-----  
char arr1[10] = { 1, 3, 5, 7, 12, 34, 11, 63, 7, 0};  
int indx;
```

```
for(indx = 0; indx < 10; indx++)  
    printf("%d ", (int)arr1[indx]);  
-----
```

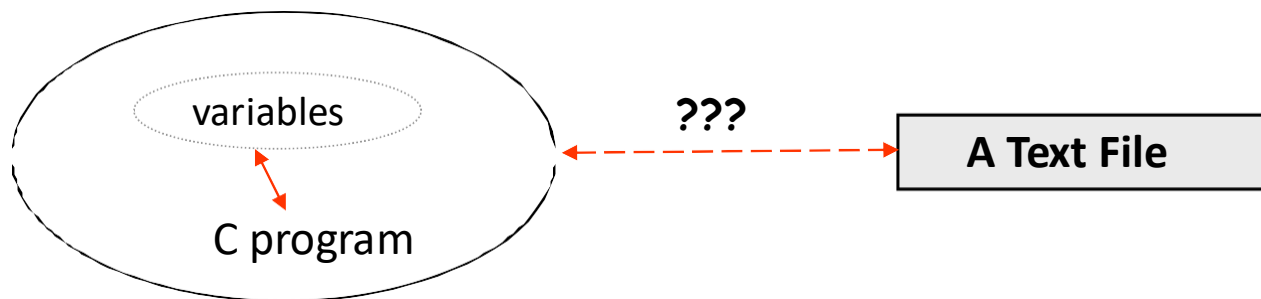
The output is:

```
1 3 5 7 12 34 11 63 7 0  
-----
```

1. `char` 1 is converted to `int` 1.
2. `printf` function converts `int` 1 to its ASCII 49.
3. A computer monitor displays 1 as the graphical symbol of 49.

II. Files and Streams

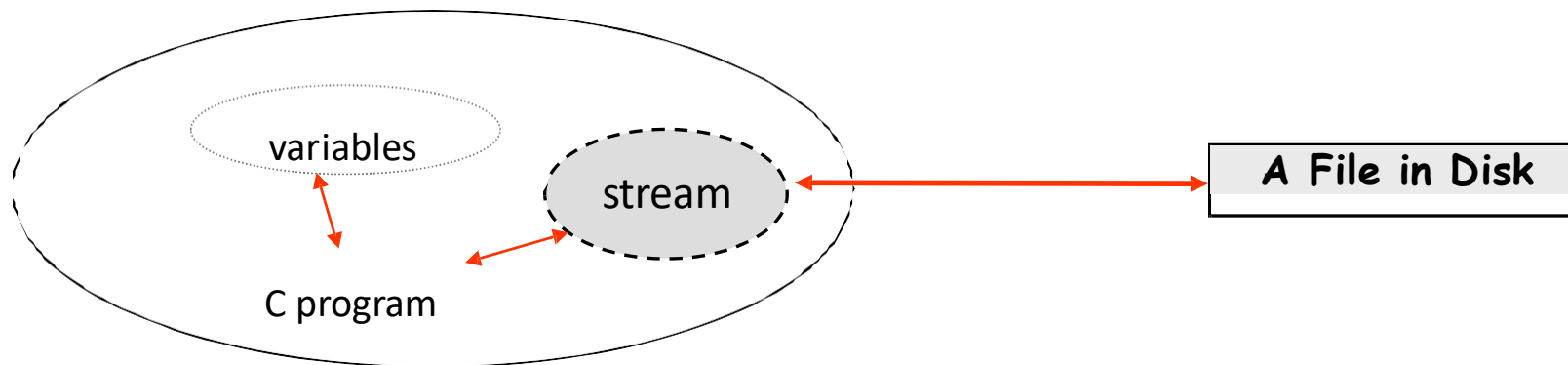
- A variable declared in a C program is a part of this program.
- A file is an **external collection of data** that is not part of the C program.



- C must provide a **linkage** between the **external file** and the **program**.
- The **linkage** is provided by **File Streams**

The Concept of File Stream

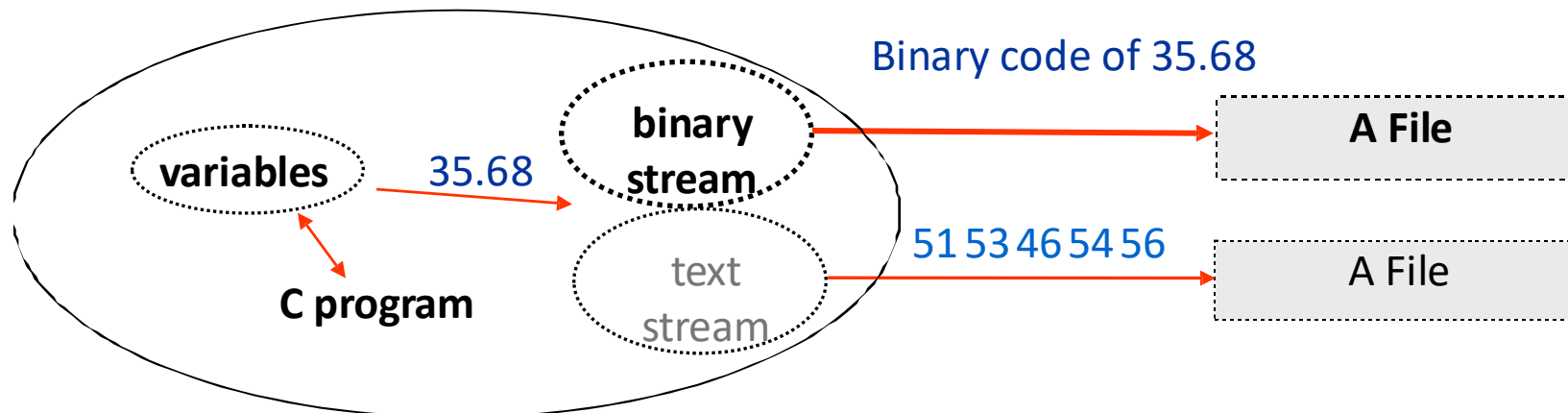
- A **File Stream** is an **interface/linkage** between a **program** and a **physical device**.
- It separates the program from the physical device **by assigning each file a logical structure**.



- You work with a **stream** and the **stream** communicates with the actual device.

Text & Binary File Streams

- **A File Stream can be Text or Binary**
 - physical files do not have any special marker to indicate their type (a file name or its extension does not affect the file type)
- **Unlike Text File Stream, A Binary File Stream does NOT perform any type conversion.**



III. Major Steps for Processing a File:

Step1: Declare stdio.h header file

1. Include the header file **stdio.h**

```
#include <stdio.h>
```

- The library contains a **set of functions** to **handle file I/O through streams**.
- Besides defining the file stream, the stdio library also **defines identifiers**:

stdin – **standard input stream** (per default from a keyboard)

stdout – **standard output stream** (per default to the monitor)

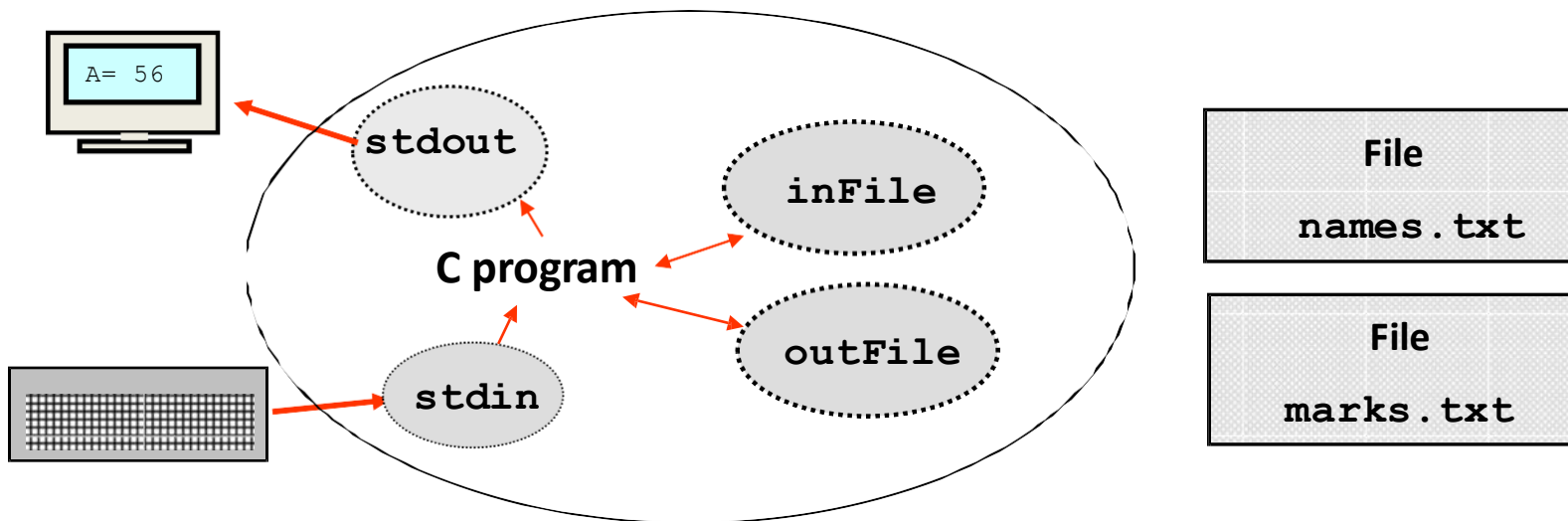
stderr – **standard error stream** (per default to the monitor)

Step2: Declare the File Stream variable

2. Declare file stream variables

```
FILE *inFile;  
FILE *outFile;
```

(stdin, stdout and stderr are already declared)

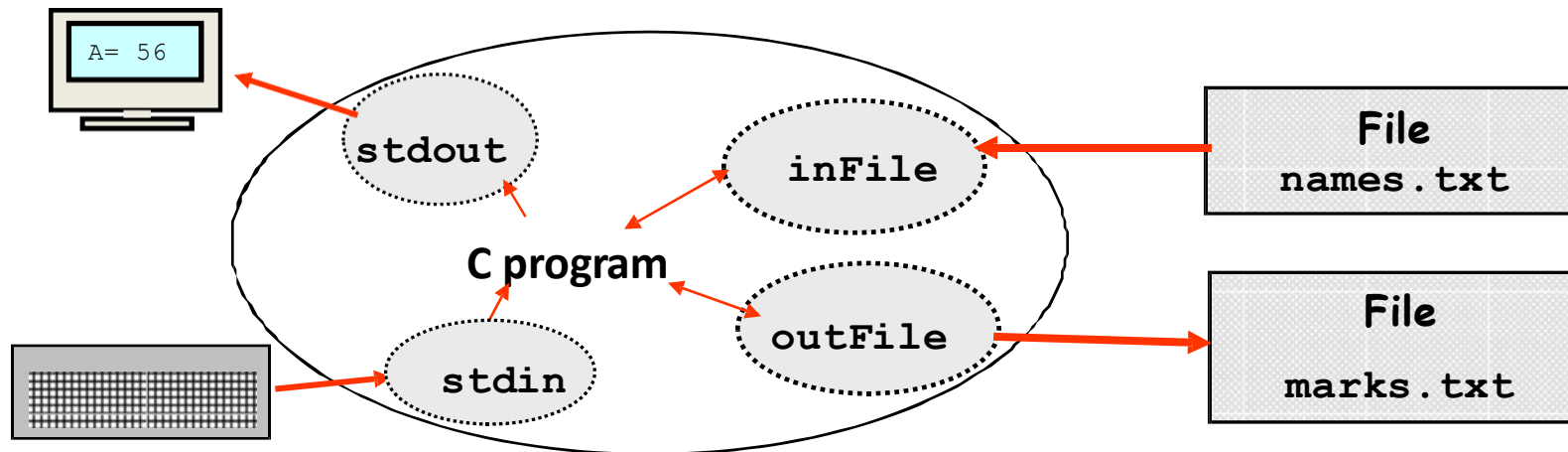


Step3: Link the file stream variable to the physical file

3. Associate the file streams with physical files: open files

3.1 TEXT file : open the file with the text flags to specify the access mode

```
inFile = fopen( "names.txt" , "r" );  
outFile = fopen( "marks.txt", "w" );
```



Text File Opening Modes

- Function **fopen()**

```
FILE* fopen(const char filename[], const char* mode);
```

Mode	Description	Restriction	Previous Content
r	Open file for reading	must already exist	preserved
w	Open file for writing	created if doesn't exist	erased
a	Open for writing. Output is appended at the end	created if doesn't exist	preserved

Example: Open a file for output. If it does not exist, it'll be created. If it already exists, the **existing content will be erased**:

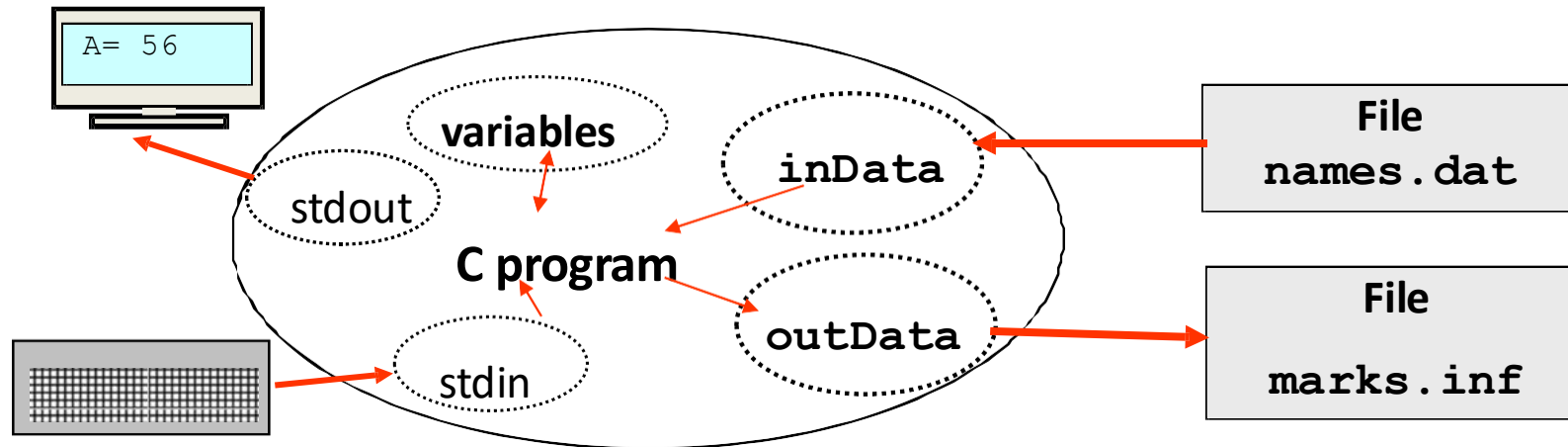
```
outFile = fopen("report.txt", "w");
```

...Continued

3.2 BINARY file stream: open the Files with the binary flags

```
inBinFile = fopen( "names.dat", "rb" );
```

```
outBinFile = fopen( "marks.inf", "wb" );
```



Binary File Opening Modes

- Function `fopen`

```
FILE* fopen(const char filename[], const char mode);
```

Mode	Description	Restriction	Previous Content
rb	Open file for reading	must already exist	preserved
wb	Open file for writing	created if doesn't exist	erased
ab	Open for writing. Output is appended at the end	created if doesn't exist	preserved

- The same function is used for opening text and binary file streams.
- Binary mode tells the I/O functions **that format conversion of data and interpretation of all escape sequences (\n, \t,...) must be suppressed.**

Open File: Error Handling

- Do NOT assume that a file stream is always successfully created

- Incorrect file name

```
inFile = fopen("names.txb", "r");
```

- Violation of restrictions

```
FILE *inFile;
```

```
inFile = fopen("names.txt", "r");
```

may not exist,
or access may
be restricted

- Hardware failure

- You should always check the status of a stream after **fopen** :

```
inFile = fopen("names.txt", "r");
```

```
if( inFile == NULL) { ... error recovery action ... }
```



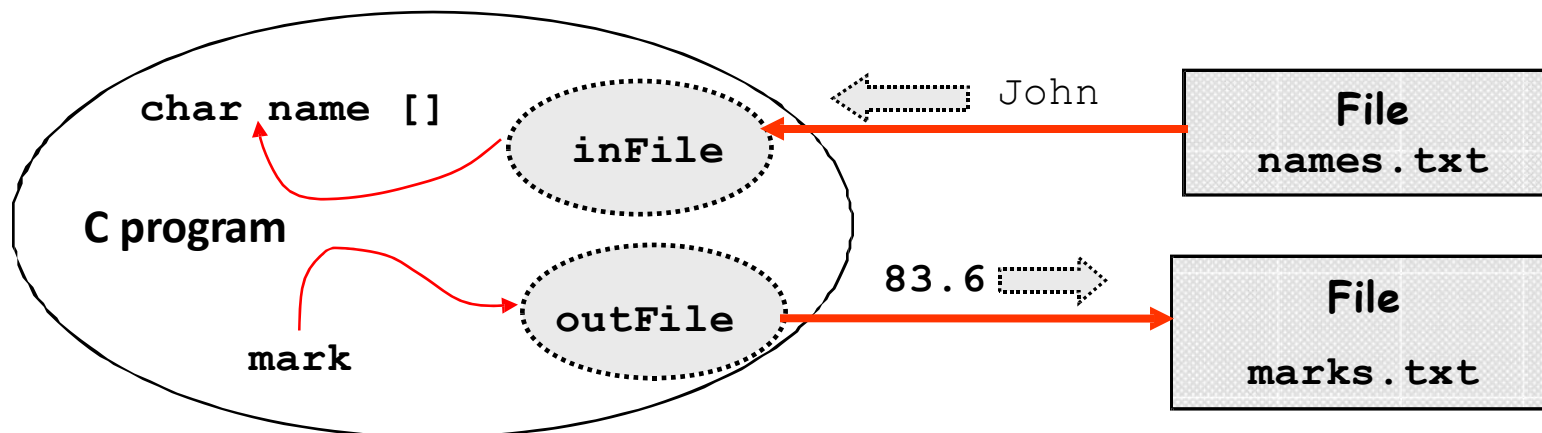
Step 4: Read/Write operations

4.

4.1. Text file stream: use `fprintf()` and `fscanf()` functions for file I/O operations:

```
fscanf(inFile, "%s", name);
```

```
fprintf(outFile, "%f", mark);
```



...Continued

- 4.2. Binary file stream:
 - use `fwrite()` and `fread()` functions for file I/O operations
 - With Binary file streams, you have to **perform operation per BLOCK**
 - The `fwrite()/fread()` function **writes/reads a given number of items**, all of the **same size**, from/to **a memory buffer** into/from a binary file stream.

```
int fwrite( void *buffer,      /* buffer */
            int sizeofItem,    /* size of items */
            int numOfItems,    /* number of items */
            FILE *fp );        /* file stream ID */
```

```
int fread ( void *buffer,      /* buffer */
            int sizeofItem,    /* size of items */
            int numOfItems,    /* number of items */
            FILE *fp );        /* file stream ID */
```

Step 5: Close the file stream

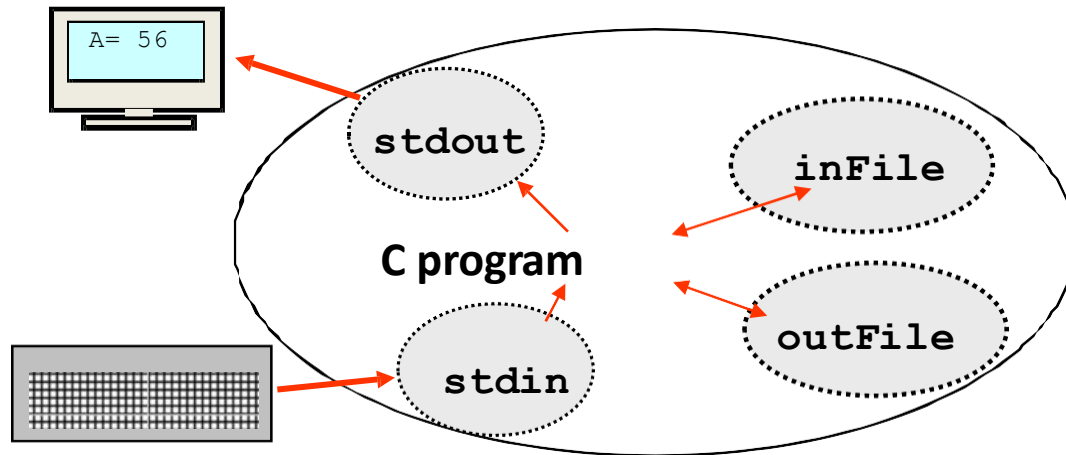
5. Disconnect from the physical files: close the files:

```
fclose (inFile) ;
```

```
fclose (outFile) ;
```

Do not accidentally close `stdin` or `stdout`

Files remain in the
computer file system
when they are closed



File
`names.txt`

File
`marks.txt`

Example1: Formatted Output in a File

- In most application scenarios a text file created by a program will be printed and read by people.
- Information must be presented in a readable form.
- **Output file streams can be formatted in the same manner as the standard output stream (easy to test and debug).**
- Besides data, the file may contain supplementary information such as titles, comments, version number, etc.

Example: Sales report

```
-----  
Date: 24/09/2006  
-----
```

Model:	FinePix S3000
Number of Items Sold:	3
Gross Amount:	1200.00
GST Paid:	120.00
Net Sale:	1080.00

...Continued

Date: 24/09/2006

Model:	FinePix S3000
Number of Items Sold:	3
Gross Amount:	1200.00
GST Paid:	120.00
Net Sale:	1080.00

Stp
1

```
#include <stdio.h>

int main(void)
{
```

Stp
2

```
    int i;

    FILE *salesReport;           /* a file stream variable */
    char date[] = "24/09/2006";  /* a c-string */
    char fileName[] = "sale24_09_06.txt"; /* a c-string */
    char model[] = "FinePix S3000"; /* a c-string */
```

Stp
3

```
    salesReport = fopen( fileName, "w" );
    if( salesReport == NULL ) /*check for file open errors*/
    {
        fprintf(stderr, "Error opening %s", fileName);
        return -1;  /* terminate the program */
    }
```

OR

printf("Error opening %s", fileName)

...Continued

```
/* write data to the file */
```

```
for (i=0; i<40; i++)
```

```
    fprintf(saleReport, "-");
```

```
fprintf(saleReport, "\nDate: %s\n", date);
```

```
for (i=0; i<40; i++)
```

```
    fprintf(saleReport, "-");
```

```
fprintf(saleReport, "\n%-27s%s", "Model:", model );
```

```
fprintf(saleReport, "\n%-27s%d", "Number of items sold:", 3 );
```

```
fprintf(saleReport, "\n%-27s%.2f", "Gross amount:", 1200.0 );
```

```
fprintf(saleReport, "\n%-27s%.2f", "GST Paid:", 120.0 );
```

```
fprintf(saleReport, "\n%-27s%.2f", "Net sale:", 1080.0 );
```

```
fclose(saleReport);    /* close the file stream */
```

```
return 0;
```

```
}
```

```
-----  
Date: 24/09/2006  
-----  
Model:                               FinePix S3000  
Number of Items Sold:                3  
Gross Amount:                        1200.00  
GST Paid:                            120.00  
Net Sale:                            1080.00
```

Stp
4

Stp
5

File Stream as a Function Parameter

- How to move file stream operations into user defined functions?
 - A File stream pointer can be passed to a function as a parameter

Example:

```
int saveReport ( FILE *fp ) {    /* fp is a function parameter */
.....
}
```

Caution: Do not declare a file stream pointer as a global variable

```
#include <stdio.h>
```

```
FILE *fp;
```

Global variable. Bad style !

```
int saveReport(void)
{
```

```
    fprintf(fp, "Model : " );
```



...Continued

```
/* A function that writes into a text file */
/* returns true if OK, or false if an error is detected */
int writeHeader( FILE *fp, char reportDate[] )
{
    int i, Status=0;
    for(i=0; i<40; i++)    fprintf( fp, "-");
    fprintf( fp, "\nDate: %s\n", reportDate);
    for(i=0; i<40; i++)    fprintf( fp, "-");
    fprintf( fp,          "\n");
    ....
    return 1;
}
```

```
/* --function call-- */
status = writeHeader( salesReport, date );
if(status != 1) fprintf( stderr, "Error!" );
```

 the file stream variable, see slide 20

Example2: Reading a file? Know first the data format

- A file can store a sequence of numbers produced by another program (results of a scientific experiment, automatic measurements...)

```
23.45 76.13 0.15 356.1 0.0 45.7 56.03 45.8
```

- A file can store formatted reports that include numbers, comments, titles, separators.

```
-----  
- Date: 24/03/2015 -----  
Number of Tickets Sold: ..... 2650  
Gross Amount: ..... $ 9150.00
```

- You need to know the **data format in order to read data correctly**

Quiz

- What may be a problem with this code?

Answer:

Return -1 is missing after fprintf along wt {}

Reading a value from the file as an integer, mark is float.

```
#include <stdio.h>

int main(void)
{
    FILE *inData;
    float mark;

    inData = fopen("test.txt", "r" );
    if(inData == NULL)
        fprintf(stderr, "Error opening test.txt \n");
    else
        fscanf(inData, "%d", &mark);

    printf("Mark = %d", mark);
    fclose(inData);
    return 0;
}
```

IV. File Error Handling

- What if the stream output fails?
 - **Hardware failure:** data may not be physically accessed from the storage device due to its damage
 - You may have already closed the file stream while trying to access its content
 - The program may not have data to read as it hits the end of file.
 - The read data may be invalid (an alphabetic character instead of a digit character).
- A program with **superficial error handling** will **never be robust**.
- A **comprehensive error checking** and **appropriate error recovery** are essential for robust applications.

Open File: Error Handling

The remaining error source is in
Step 4

```
#include <stdio.h>
int main(void)
{
    FILE *inData;          /* declare a file stream variable */
    char fileName[] = "exams.txt";
    float mark;

    inData = fopen( fileName, "r" ); /* open for reading */
    if(inData == NULL) /* check if opened successfully */
    {
        fprintf(stderr, "Error opening %s", fileName);
        return -1; /* exit with an error code */
    }
    fscanf(inData, "%d", &mark);

    fclose(inData);          /* Close the input file */
    return 0;
}
```

IV.1. Text File: Writing Error Handling

- Exploit **fprintf()** return value
 - If successful, the **fprintf()** function returns number of characters written.
 - On failure, it returns a negative value.

```
stat=fprintf(saleReport, "\n%-27s%.2f", "Net:", 180.0);  
if(stat<0) {  
    fprintf(stderr, "File output error \n");  
    fclose( saleReport );  
    return -1;  
}
```

Error Handling
Or Recovery

- On a failure, often the return value of **fprintf()** is -1, which often equals the constant **EOF**.

☞ “if(stat==EOF)” is often used instead of if(stat<0) after **fprintf**

Text File: Reading Error Handling

- Exploit **fscanf()** return value:
 - If successful, the **fscanf()** function returns the number of receiving arguments successfully assigned.
 - If a matching failure occurs before the first receiving argument was assigned, it returns zero.
 - If input failure occurs before the first receiving argument was assigned, **EOF** is returned.
 - If the end of file is reached, **EOF** value is returned
 - **IMPORTANT:** **EOF** is a system indicator. There is no any **EOF** marker physically stored in the file to indicate the end of it.

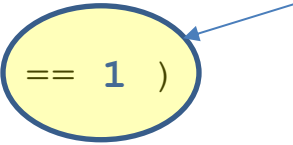
Example: Calculate the average of a file data content

```
#include <stdio.h>

int main(void)
{
    float nextValue, averTemp = 0.0;
    int total = 0;
    FILE *inFile;

    inFile = fopen("may_2004.txt", "r");
    if(inFile == NULL) { /*check if the file stream has been opened */
        fprintf(stderr, "File opening error" );
        return -1;
    }
    /* read until the end-of-file */
    while( fscanf(inFile, "%f", &nextValue) == 1 )
    {
        averTemp += nextvalue;
        total++;
    }

    printf("The average temperature is: %f", averTemp/total);
    fclose( inFile );
    return 0;
}
```



❖ Would the loop always exit after reading all the file content?

...Continued

❖ Would the loop always exit after reading all the file content? **NO**

- What if the file had an alphabetic character instead of a digit one?
(by mistake, file corrupted, hardware failure)

24.56 25.02 24.04 **G** 23.15 22.47 22.13 21.84

```
1. while( fscanf(inFile, "%f", &nextValue) == 1 )  
    averTemp += nextvalue;
```

When `fscanf` fails and returns 0, further reading from the file stream will stop.

Side effects: We would assume incorrectly that all file data has been processed.

2. Another Naïve solution attempt, check if I have reached the end of file

```
while( fscanf(inFile, "%f", &nextValue) != EOF )  
    averTemp += nextvalue;
```

Side effects: An infinite loop in the case of a wrong character

IV.2: Error Handling, a better solution: Error State

- `stdio.h`: provides **two status flags** and **two functions** to detect possible errors
 - Incorrect operations with a stream, set **EOF** or **ERROR** system indicators (status flags) and switch the stream to the Error State.
 - The flag **EOF** corresponds to **end of file status**:

```
if ( feof(fp) ) {      Error recovery action }
```
 - The flag **ERROR** indicates a **hardware failure**:

```
if ( ferror(fp) ) {    Error recovery action }
```
- **Caution:** **feof()** and **ferror()** functions themselves do not test for end-of-file or error. They simply return a value of the corresponding system indicator. The indicators are set by I/O attempts (**fscanf()**, **fread()**, **fprintf()**, **fwrite()**, **getc()**,...)

...Continued

Slide 30 program's Amended Error handling

```
/* keep reading in a buffer sequentially while correct */
while( fscanf(inFile, "%f", &nextValue) == 1 )
{
    averTemp += nextvalue; total++;
}
/* determine the cause of fscanf failure */
/* check if all numbers have been read */
if( !feof(inFile) ) return -1;
/* check if the stream read error has occurred */
if( ferror(inFile) ) return -2;
```

one value is expected

the End-Of-File has not been reached

a file read error has been detected

The while loop has to read all numbers in the file and reaches EOF. **The if statements above are to check that all numbers have read by the while loop**

Quiz

- Assuming the text file content is: **1 2 3**

This solution

```
while( !feof(inFile) )
{
    fscanf(inFile, "%d", &number);
    printf("%d ", number);
}
```

produces the following output for some reason

1 2 3 3

What is wrong with this code?

Answer: After reading number 3, fscanf will return the value of 1. The while condition is still true. At the next loop iteration, fscanf will not find a value to read as the pointer is at the end of the file. But printf will print the last value of the variable number, which is 3. Afterwards, the loop condition is found to be false.

IV.3. Binary File: Write Error Handling

- Reminder:
 - with Binary file streams, you have to perform BLOCK write operation
 - The **fwrite** function writes a **given number of items, all of the same size, from a memory buffer** into a Binary file stream.

```
int fwrite( void *buffer,      /* buffer */
            int  sizeofItem,   /* size of items */
            int  numOfItems,   /* number of items */
            FILE *fp );        /* file stream ID */
```

- **fwrite** returns a number of items actually written into the file:

If the actual number of items written is less than `numOfItems`, this indicates that an error has occurred.

Example 1

1. Declare a buffer:

```
#define BUFSIZE 240  
.  
.  
char buff[BUFSIZE];
```

2. Write a specified number of items into a file stream:

```
if( fwrite(buff, sizeof(char), BUFSIZE, fp) != BUFSIZE)  
{  
  
    /* check for ERROR condition */  
    if( ferror(fp) ) { . . . error recovery . . . }  
  
}
```

Binary File: Read Error Handling

- Reminder:
 - With Binary file streams, you have to perform BLOCK read operation
 - The **fread** function reads a **given number of items, all of the same size**, from the stream and places them into a memory buffer.

```
int fread( void *buffer,      /* buffer */
           int  sizeofItem,   /* size of items */
           int  numOfItems,   /* number of items */
           FILE *fp );        /* file stream ID */
```

- **fread** returns a number of items actually read from the file:
 - If the actual number of items read is less than `numOfItems`, **check error indicators**.

Example 2

1. Declare a buffer

```
#define BUFSIZE 15
```

```
. . .
```

```
char buff[BUFSIZE];
```

24	-1	5	17	32	28	0	11	74	27	3	-64	8	14	26
----	----	---	----	----	----	---	----	----	----	---	-----	---	----	----

2. Read a specified number of items from a file stream

```
if( fread(buff, sizeof(char), BUFSIZE, fp) != BUFSIZE )
{
    /* check for EOF condition */
    if( !feof(fp) ) { . . . error recovery . . . }

    /* check for ERROR condition */
    if( ferror(fp) ) { . . . error recovery . . . }
} // may be not enough data to read
```

Example 3 : struct and Binary File

```
typedef struct
```

```
{  
    int productCode;  
    int numberInStock;  
    float price;  
}product_t;
```

ProductCode	NumberInStock	Price
-------------	---------------	-------

int

int

float

347051
12
145.99

```
. . . .
```

```
product_t record; /* a structured type variable */
```

```
. . . . .
```

```
/*read a record from a file stream into the struct variable*/
```

```
if (fread( &record, sizeof(product_t), 1, inFile )==1)
```

```
{
```

```
printf("The product code : %d\n", record.productCode);
```

```
printf(" Number in stock : %d\n", record.numberInStock);
```

```
printf(" The product price : %f\n", record.price);
```

```
}
```

```
if( ferror(inData) ) { ..Error Recovery Actions.. }
```

```
if( !feof(fp) ) { . . . error recovery . . . }
```

Example 4: Average of a Binary File Data

```
/* assume the file data is a multiple of 100 */
#define LENGTH 100
FILE *inData;
int buffer[LENGTH]; /* declare a buffer */

int counter=0, sum = 0;

inData = fopen("results.dat", "rb" );
if(inData == NULL ) { ..Error Recovery Actions.. }

/* -- read a block of data -- */
while(fread(buffer, sizeof(int), LENGTH, inData ) == LENGTH)
    for(i=0; i<LENGTH; i++)
    {
        sum += buffer[i]; /* add all numbers */
        counter++; /* count the numbers */
    }
}

if( ferror(inData) ) { ..Error Recovery Actions.. }
if( !feof(inData) ) { ..Error Recovery Actions.. }
/* all file data are read successfully */
printf("The average value is: %f", (float)sum/counter );
fclose(inData); /* Close the file stream */
```

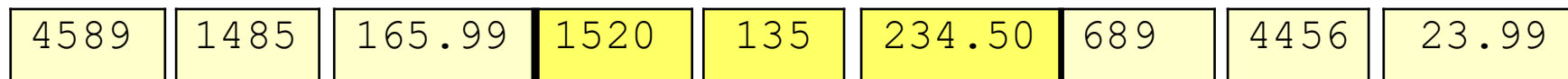

Caution: Struct Memory Allocation Mismatch

- Warning:** If your structure memory layout does not match precisely the layout of records in a file, you will read garbage from the file into your structure.

Example: A **binary file** has been created on another platform that inserted a different number of slack bytes in the same structure

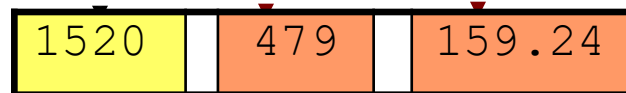
```
typedef struct
{
    int productCode;
    int numberInStock;
    float price;
} product_t;
```

records (in a binary file)



record

(a variable declared in your program to be used as a buffer)



OK

mismatch

mismatch

Example 5: Text to Binary File

- A function to convert a text file into a binary file

Part Number Component Price



2376 BF245 14.50

2378 BF981 7.80

2379 BF998 15.50

Text file format



text2binary

File conversion program



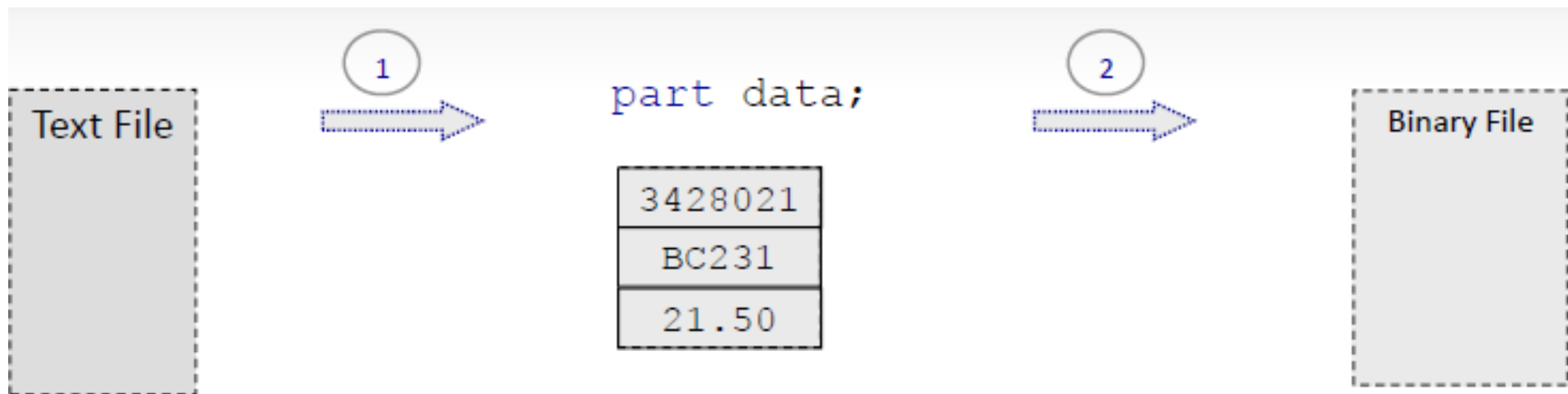
record1

record2

. . . .

Partial Design Solution

```
typedef struct{
    int partN;
    char component[10];
    float price;
} part_t;
```



1 **fscanf**(flText, "%d%9s%f", &(data.partN), data.component, &(data.price));

2 **fwrite**(&data, sizeof(part_t), 1, flBin);

...Continued

```
/* --define a data structure-- */
typedef struct
{
    int partN;           /* part number */
    char component[10];  /* component name */
    float price;         /* retail price */
} part_t;

/* --- function prototype --- */
bool copyData( FILE *flText, FILE *flBin);
. . .

/* --- function call --- */
status = copyData( txtFileID, binFileID );
if( status == false ) { /* check for a failure */
    fprintf( stderr, "Copy failed \n" );
    return -1;
}
```

...Continued

```
typedef struct{
    int partN;
    char component[10];
    float price;
} part_t;
```

```
/* -- function definition -- */

bool copyData( FILE *flText, FILE *flBin)
{
    part_t data; /* struct variable to serve as a buffer */

    while( fscanf( flText, "%d%9s%f", &(data.partN), data.component, &(data.price) ) == 3 )
    {
        if( fwrite( &data, sizeof(part_t), 1, flBin) != 1 )
            return false; /* binary file write error */
    }

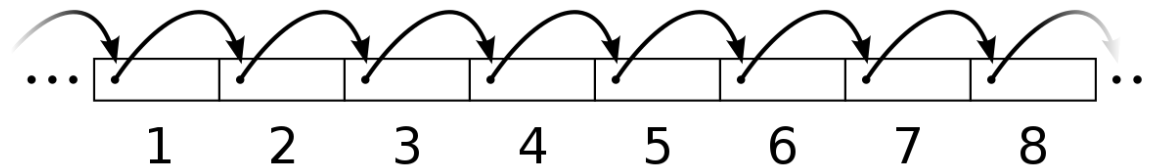
    if( !feof( flText) ) /*not all text file has been read*/
        return false;

    if (ferror(flText)||ferror(flBin) )return false;
    return true; /*return true if completed without errors*/
}
```

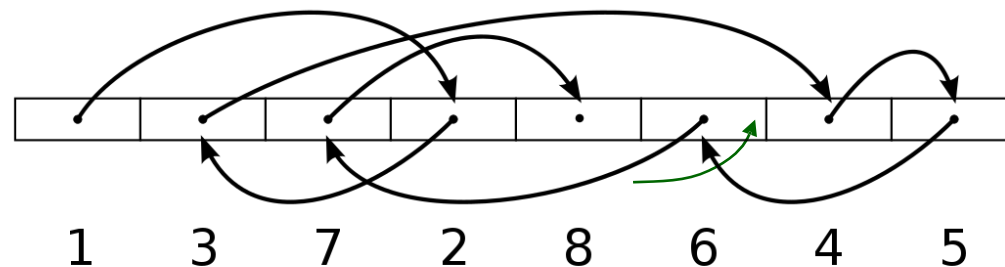
V. Random Data Access

- In C, the data stored in a file can be accessed in the following ways:
 - **Sequential Access:** data is saved and accessed in sequential order
 - **Random Access :** Access any location in the file without having to access its precedent order
 - C supports Random Access through three functions: **ftell(),fseek(), frewind()**

Sequential access

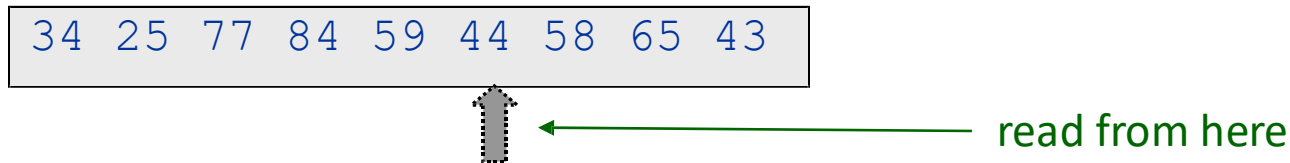


Random access

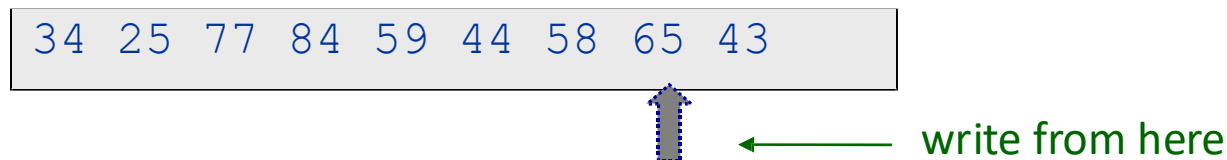


File Position Indicator

- Each file stream has a **position indicator**
 - The **position indicator** for an input stream indicates the **position in the file** from which data will be extracted.



- The **position indicator** for an output stream indicates the **position in the file** where data is to be inserted.



–Every successful I/O operation moves the indicator automatically.

File Position Indicator Function: `ftell()`

- Function `ftell()` returns a value of type `long int` that is the current file position in byte.

```
FILE *outBinFile;
```

```
char buffer[8] = {a,b,c,d,e,f,g,h};
```

```
long int filePosition;
```

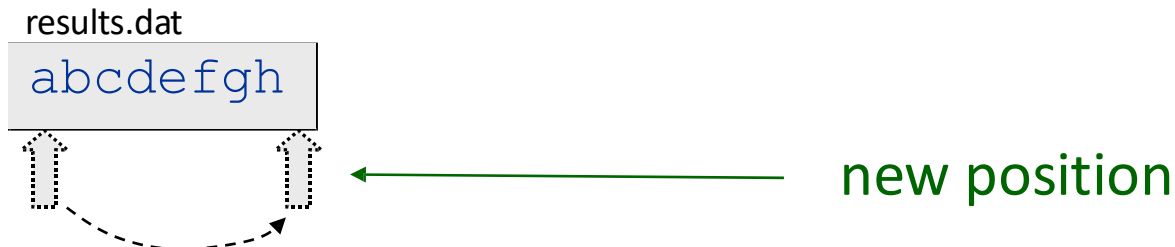
File error handling are omitted
only to reduce the size of the code

```
outBinFile = fopen("results.dat", "wb");
```

```
filePosition = ftell(outBinFile);    /* filePosition = 0 */
```

```
fwrite( buffer, sizeof(char), 8, outBinFile);
```

```
filePosition = ftell(outBinFile);    /* filePosition = 8 */
```



File Indicator Repositioning: `fseek`

- The Function

- `int fseek(FILE *fp, int displacement, int origin)`

is to move forward or backward the file position indicator by a number of bytes

`displacement` relative to the `origin`

<code>origin</code>	Meaning
<code>SEEK_SET</code>	0: beginning of the file
<code>SEEK_CUR</code>	1: current position
<code>SEEK_END</code>	2: end of the file

- The function returns 0 if the indicator has been successfully moved, or nonzero if an error occurred.
 - You always need to check if repositioning has been successful.

...Continued

```
FILE *inFile;
```

```
int filePosition;
```

```
outBinFile = fopen("results.dat", "rb");
```

```
45476575869869401158
```



```
if( fseek(inFile, 4, SEEK_SET) != 0) { ..error ..}
```

```
45476575869869401158
```



```
if( fseek(inFile, -3, SEEK_END) != 0) { ..error ..}
```

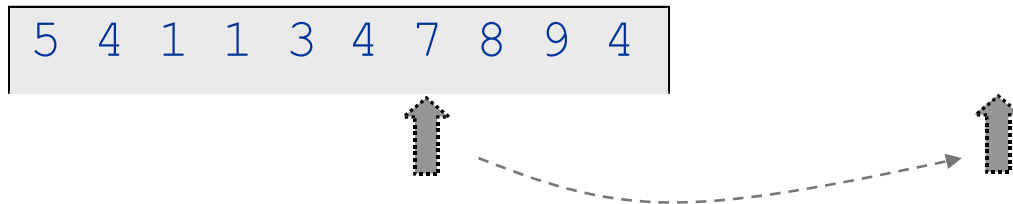
```
45476575869869401158
```



Caution

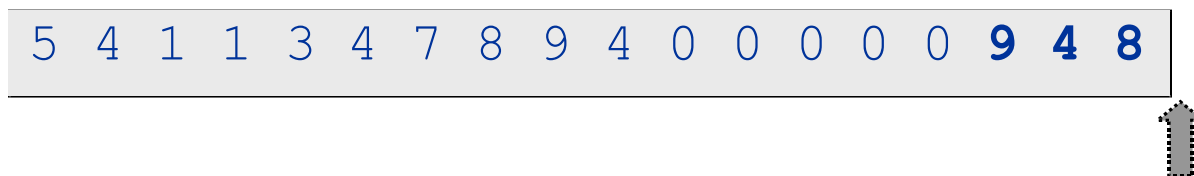
- **Positioning beyond the end-of-file does not cause an error.** File position indicator is moved.
 - Subsequent reading from that position causes EOF error:

```
fseek (inFile, 5, SEEK_END);
```



- Subsequent **writing** starts from that position.

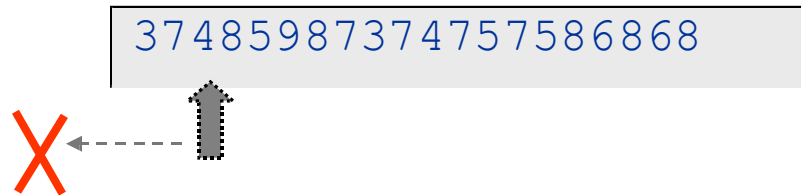
```
fwrite( &buff, sizeof(char), 3, outFile );
```



...Continued

- **Positioning before the file beginning does NOT move the File position indicator:**

```
fseek(inFile, -4, SEEK_CUR);
```



Example 1: File size using ftell() and fseek()

How to obtain a total number of bytes stored in a file?

```
int pos, fileSize;
FILE *fileP;

fileP = fopen(filename, "rb");

if(fileP == NULL ) { Error Recovery Action }

fseek( fileP, 0, SEEK_END ); /*position to the End*/
pos = ftell(fileP);          /* get position of the End*/
fileSize = pos;              /* size = end position-0*/
```

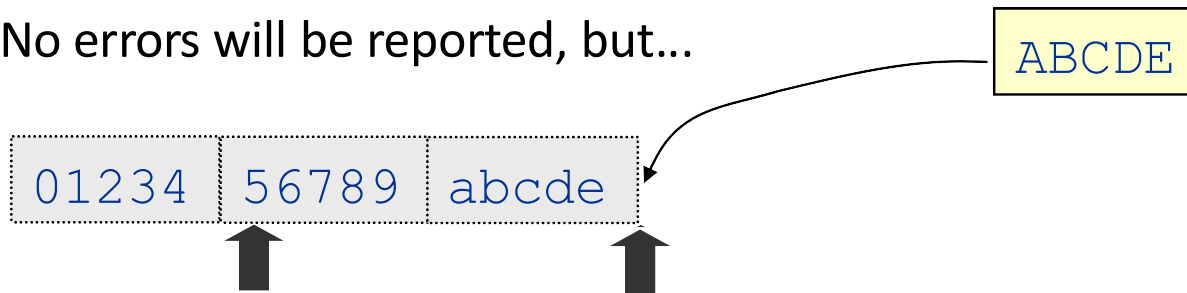
Caution

If a file stream has been opened in ***append*** mode, all **positioning attempts are ignored**. All output is always directed to the end-of-file

Example:

```
db = fopen( "list.dat", "ab" );  
if( fseek( db, 5, SEEK_SET ) != 0 )  
    fprintf(stderr, " File Positioning Error \n");  
if( fwrite( buff, sizeof(char), 5, db) != 5 )  
    printf("Write error \n");
```

No errors will be reported, but...

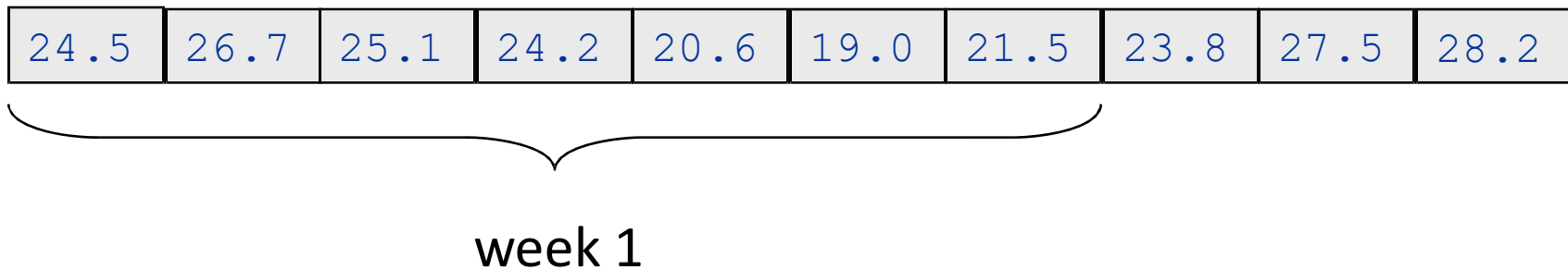


Example 2: Random File Reading

- An automatic temperature registration system saves an average daily temperature in a file. Your program must read data from this file and calculate an average temperature for a specified week.
 1. You need to know the file name and its path.
 2. You need to have permission to access this file.
 3. You need to know the file format.

...Continued

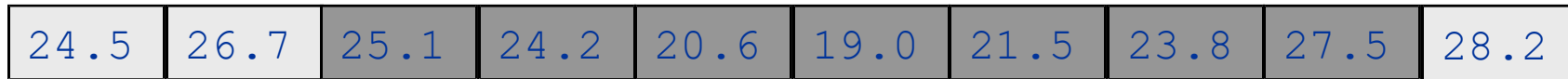
1. "2014.dat" file is in the current directory.
2. You have a permission to read this file.
3. Temperature measurements are stored in the file in binary format as a sequence of **float** numbers starting from a week 1.



...Continued

Use `fseek(tmpData, offset, SEEK_SET);`

file



24.5	26.7	25.1	24.2	20.6	19.0	21.5	23.8	27.5	28.2
------	------	------	------	------	------	------	------	------	------

buffer

25.1	24.2	20.6	19.0	21.5	23.8	27.5
------	------	------	------	------	------	------

Then `fread(buffer, sizeof(float), 7, tmpData)`

...Continued

```
FILE *tmpData;
float sum = 0.0, buffer[7];
int i, offset, weekNum,

tmpData = fopen("2014.dat", "rb" );
if(tmpData == NULL )    { ..Error Recovery Actions.. }

/* calculate the required position */
offset = (weekNum-1)*7*sizeof(float);
/* -- move indicator to the required position -- */
if( fseek( tmpData, offset, SEEK_SET ) != 0)
    { ..Error Recovery Actions.. }

/* -- read 7 values -- */
if(fread( buffer, sizeof(float), 7, tmpData ) != 7)
    { ..Error Recovery Actions.. }

for(i=0; i<7; i++)
    sum += buffer[i];

/* calculate the average temperature */
printf("The average weekly temperature: %f", sum/7 );
fclose(tmpData);    /* Close the file stream */
```

Appendix next.....

Summary: Data I/O using Text and Binary Files

Table in next slide compares the use of text and binary files for input and output of data of various types. The statements in both columns assume the following constant macros, type definition, and variable declarations.

```
#define STRSIZ 10
#define MAX 40
typedef struct {
    char name[STRSIZ];
    double diameter;      /* equatorial diameter in km */
    int moons;            /* number of moons */
    int orbit_time,       /* years to orbit sun once */
        rotation_time;   /* hours to complete one
                           revolution on axis */
} planet_t;
. . .
double nums[MAX], data;
planet_t a_planet;
int i, n, status;
FILE *plan_bin_inp, *plan_bin_outp, *plan_txt_inp, *plan_txt_outp;
FILE *doub_bin_inp, *doub_bin_outp, *doub_txt_inp, *doub_txt_outp;
```

...Continued

Example	Text File I/O	Binary File I/O	Purpose
1	<pre>plan_txt_inp = fopen("planets.txt", "r"); doub_txt_inp = fopen("nums.txt", "r");</pre>	<pre>plan_bin_inp = fopen("planets.bin", "rb"); doub_bin_inp = fopen("nums.bin", "rb");</pre>	Open for input a file of planets and a file of numbers, saving file pointers for use in calls to input functions.
2	<pre>plan_txt_outp = fopen("pl_out.txt", "w"); doub_txt_outp = fopen("nm_out.txt", "w");</pre>	<pre>plan_bin_outp = fopen("pl_out.bin", "wb"); doub_bin_outp = fopen("nm_out.bin", "wb");</pre>	Open for output a file of planets and a file of numbers, saving file pointers for use in calls to output functions.
3	<pre>fscanf(plan_txt_inp, "%s%lf%d%lf%lf", a_planet.name, &a_planet.diameter, &a_planet.moons, &a_planet.orbit_time, &a_planet.rotation_time);</pre>	<pre>fread(&a_planet, sizeof (planet_t), 1, plan_bin_inp);</pre>	Copy one planet structure into memory from the data file.
4	<pre>fprintf(plan_txt_outp, "%s %e %d %e %e", a_planet.name, a_planet.diameter, a_planet.moons, a_planet.orbit_time, a_planet.rotation_time);</pre>	<pre>fwrite(&a_planet, sizeof (planet_t), 1, plan_bin_outp);</pre>	Write one planet structure to the output file.

(continued)

...Continued

Example	Text File I/O	Binary File I/O	Purpose
5	<pre>for (i = 0; i < MAX; ++i) fscanf(doub_txt_inp, "%lf", &nums[i]);</pre>	<pre>fread(nums, sizeof (double), MAX, doub_bin_inp);</pre>	Fill array <code>nums</code> with type <code>double</code> values from input file.
6	<pre>for (i = 0; i < MAX; ++i) fprintf(doub_txt_outp, "%e\n", nums[i]);</pre>	<pre>fwrite(nums, sizeof (double), MAX, doub_bin_outp);</pre>	Write contents of array <code>nums</code> to output file.
7	<pre>n = 0; for (status = fscanf(doub_txt_inp, "%lf", &data); status != EOF && n < MAX; status = fscanf(doub_txt_inp, "%lf", &data)) nums[n++] = data;</pre>	<pre>n = fread(nums, sizeof (double), MAX, doub_bin_inp);</pre>	Fill <code>nums</code> with data until EOF encountered, setting <code>n</code> to the number of values stored.
8	<pre>fclose(plan_txt_inp); fclose(plan_txt_outp); fclose(doub_txt_inp); fclose(doub_txt_outp);</pre>	<pre>fclose(plan_bin_inp); fclose(plan_bin_outp); fclose(doub_bin_inp); fclose(doub_bin_outp);</pre>	Close all input and output files.

Text File Access Mode

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, <i>discard</i> the current contents.
a	Open or create a file for writing at the end of the file—i.e., write operations <i>append</i> data to the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for reading and writing. If the file already exists, <i>discard</i> the current contents.
a+	Open or create a file for reading and updating; all writing is done at the end of the file—i.e., write operations <i>append</i> data to the file.

Common Programming Errors

- Misunderstanding of file I/O functions.
- Confusing file stream pointers and file names.
- Not all possible errors with file operations are anticipated.
- File stream is not closed when you finish working with it.
- Opening file for output using "w" mode erases the previous file content.
- Attempts to read a file, which has been opened only for output.

Removing a File: remove()

- Closing a file stream do not remove a physical file from a file system.

- To remove a file use **remove**(*file_name*) function:

```
if( remove("tempfile.txt") == -1 )
```

```
fprintf(stderr, " Error removing tempfile \n");
```

- remove : On success, zero is returned. On error, -1 is returned

- Remove fails if:

–file is currently used by a program

–you do not have a permission

–file does not exist

Error Clearing

- Once the stream is in the error state, it will stay that way.
- All subsequent operations may have unpredictable behaviour no matter what they are or what is in the input.
- If you intend to use a stream after a failure, you have to **clear** the stream by calling **clearerr()** function.

```
fscanf(inFile, "%f", price);
```

```
if(ferror(inFile)) /* check the error indicator */
```

```
clearerr(inFile);
```

- This will clear the two file indicators and recover the file stream from the error state. However, further reading of data may be useless.

fgets, fgetc, fputs, fputc functions

- **int fgetc(FILE* stream);** fgetc cast the character to an integer

On success, the fgetc() function returns the read character.

On failure it returns EOF. If the failure is caused due to end of file, it sets the eof indicator. If the failure is caused by other errors, it sets the error indicator

- **int putc(int char, FILE *stream)** writes a character (an unsigned char) specified by the argument **char** to the specified stream and advances the position indicator for the stream. This function returns the character written as an unsigned char cast to an int or EOF on error.

On an error sets the error indicator on stream.

- **fgets() prototype**

char* fgets(char* str, int count, FILE* stream);

The fgets() function reads a maximum of count-1 characters from the given file stream and stores them in the array pointed to by *str*.

The parsing continues until the end of file occurs or a newline character (\n) is found. The array *str* will contain the newline character too in case it is found. If no error occurs, a null character is written at the end of *str*

On success, the fgets() function returns *str* and on failure it returns null pointer.

If the failure is caused due to end of file condition, it sets the eof indicator. In this case, the contents of *str* are not changed.

If the failure is caused due to some other error, it sets the error indicator. In this case, the contents of *str* are indeterminate. They may not even be null terminated.

- **fputs() prototype**

int fputs(const char* str, FILE* stream); The fputs() function writes all the character stored in the string *str* to the output file stream except the terminating null character.

On success, the fputs() function returns a non-negative value. On failure it returns *EOF* and sets the error indicator on stream.

Character Input/Output


- How to read all characters from a text file including blanks, tabs and new-lines ?

1. **fscanf** (inFile, "%c", &ch) – if performance is not an issue

2. **fgetc** () , or **getc** () to read one character at a time

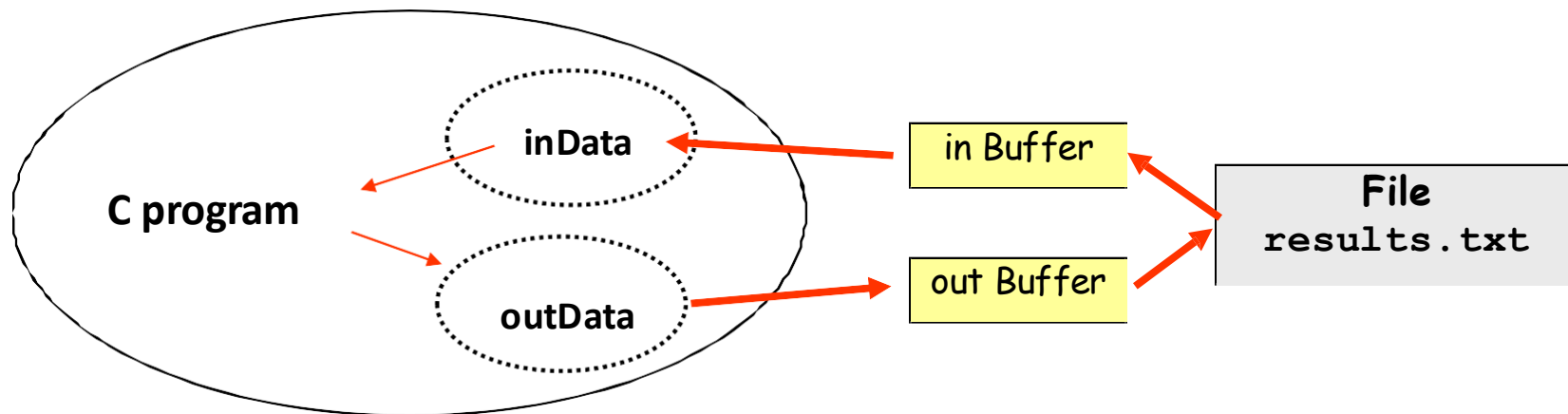
```
int ch; /* the data type must be int although ASCII codes are stored */
while( (ch = fgetc(inFile)) != EOF )
{ process a character }
```

fputc () , or **putc** () functions to write one character at a time

```
int ch = 'H';  must be a valid ASCII character
while( fputc(ch, outFile) != EOF )
{ put a character into the variable }
```

Caution

- Operations when one file is associated with two streams are prone to errors.



Quiz

What is wrong with this code and how the bug can be fixed?

```
FILE *outFile;  
FILE *inFile;  
int i, ch;  
  
outFile = fopen( "temp.txt", "w" );  
. . .  
inFile = fopen( "temp.txt", "r" );  
. . .  
for( i=0; i<LENGTH; i++ )  
{  
    fputc( i, outFile );  
    ch = fgetc( inFile );  
}
```

Answer: Read and write at the same time. First, write all characters and then read all characters. (2loops)

Example1: Format Text File Data in another File

A program that reads a collection of numbers from a file `indata.txt` and writes each number rounded to 2 decimal places on a separate line of file `outdata.txt`.

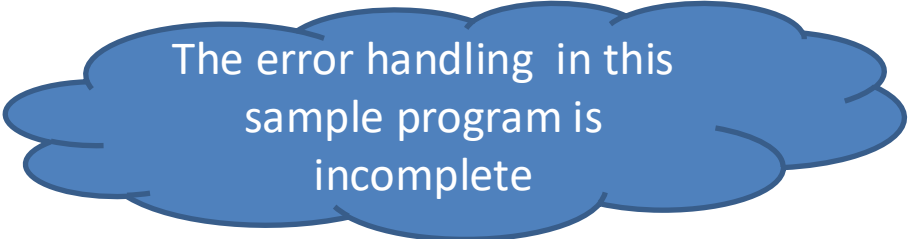
```
1. /* Reads each number from an input file and writes it
2.  * rounded to 2 decimal places on a line of an output file.
3.  */
4. #include <stdio.h>
5.
6. int
7. main(void)
8. {
9.     FILE *inp;          /* pointer to input file */
10.    FILE *outp;          /* pointer to output file */
11.    double item;
12.    int input_status;    /* status value returned by fscanf */
13.
14.    /* Prepare files for input or output */
15.    inp = fopen("indata.txt", "r");
16.    outp = fopen("outdata.txt", "w");
17.
18.    /* Read each item, format it, and write it */
19.    input_status = fscanf(inp, "%lf", &item);
20.    while (input_status == 1) {
21.        fprintf(outp, "%.2f\n", item);
22.        input_status = fscanf(inp, "%lf", &item);
23.    }
24.
25.    /* Close the files */
26.    fclose(inp);
27.    fclose(outp);
28.
29.    return (0);
30. }
```

File indata.txt

344 55 6.3556 9.4
43.123 47.596

File outdata.txt

344.00
55.00
6.36
9.40
43.12
47.60



The error handling in this sample program is incomplete

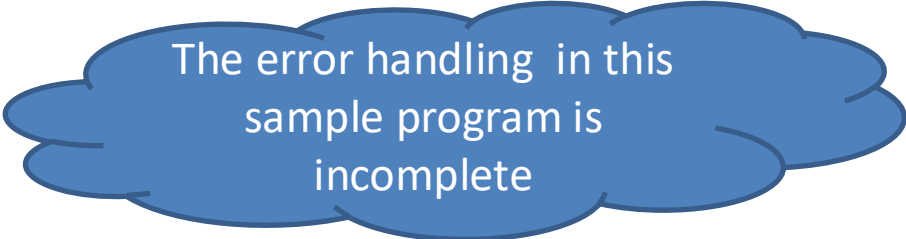
Example 2 : Backup Copy of a Text File

A program that Makes a backup file. Repeatedly prompts for the name of a file to back up until a name is provided that corresponds to an available file. Then it prompts for the name of the backup file and creates the file copy.

```
1.  /*
2.   * Makes a backup file. Repeatedly prompts for the name of a file to
3.   * back up until a name is provided that corresponds to an available
4.   * file. Then it prompts for the name of the backup file and creates
5.   * the file copy.
6.   */
7.
8.  #include <stdio.h>
9.  #define STRSIZ 80
10.
11.  int
12.  main(void)
13.  {
14.      char   in_name[STRSIZ],      /* strings giving names          */
15.            out_name[STRSIZ];      /*   of input and backup files   */
16.      FILE *inp,                  /* file pointers for input and   */
17.            *outp;                /*   backup files                */
18.      char ch;                   /* one character of input file    */
19.
20.      /* Get the name of the file to back up and open the file for input */
21.      printf("Enter name of file you want to back up> ");
22.      for (scanf("%s", in_name);
23.           (inp = fopen(in_name, "r")) == NULL;
24.           scanf("%s", in_name)) {
25.          printf("Cannot open %s for input\n", in_name);
26.          printf("Re-enter file name> ");
27.      }
28.
```

...Continued

```
29.  /* Get name to use for backup file and open file for output          */
30.  printf("Enter name for backup copy> ");
31.  for (scanf("%s", out_name);
32.       (outp = fopen(out_name, "w")) == NULL;
33.       scanf("%s", out_name)) {
34.      printf("Cannot open %s for output\n", out_name);
35.      printf("Re-enter file name> ");
36.  }
37.
38.  /* Make backup copy one character at a time                          */
39.  for (ch = getc(inp); ch != EOF; ch = getc(inp))
40.      putc(ch, outp);
41.
42.  /* Close files and notify user of backup completion                  */
43.  fclose(inp);
44.  fclose(outp);
45.  printf("Copied %s to %s.\n", in_name, out_name);
46.
47.  return(0);
48. }
```



The error handling in this
sample program is
incomplete