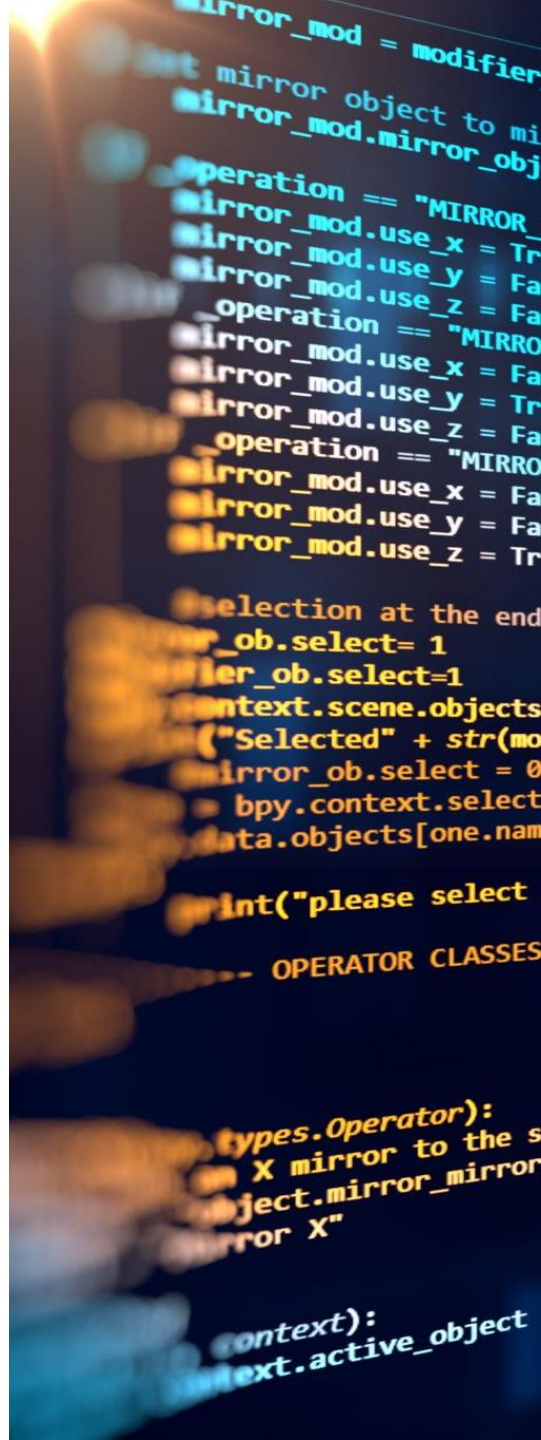
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

File handling in C

Ahmed Ibrahim



Data Permanent Storage

- **Problem**: If the same data needs to be processed repeatedly, we are required to re-enter it each time.
- This becomes a time-consuming task, especially when handling large volumes of data.
- **Solution**: Data can be stored permanently, allowing a program to access it quickly.
- Permanent storage on a disk is referred to as a **FILE**. A **FILE** is a collection of records that can be accessed using a set of library functions.

FILE Types

- In this case, data is stored permanently.
- To read the last record in a file, all preceding records must be read sequentially.
- For example, to access the 10th record, you must read through the first 9 records in order.

- In this case, data can be accessed and modified randomly.
- The last record of the file can be accessed directly, which is faster than reading a sequential file.

Sequential (Text) Files

How to define a FILE

- Before a file can be used in a program, it must be opened, creating a **connection** between the program and the operating system (OS).

- A file must be defined as follows:

```
FILE  *fp;
```

data structure

file pointer

- In this context, FILE refers to the data structure provided in the header file stdio.h, and fp is declared as a pointer that holds the address of the data to be read.

Opening a File in C

We use the `fopen` function to open a file, which establishes a connection between the file and the program.

Syntax: `FILE *fopen(const char *path, const char *mode);`

Example:

```
FILE *fp = fopen("ABC.DOC", "r");
```

- `fp` is a **file pointer** that will hold the address of the file.
- `fopen` is the function used to open the file."ABC.DOC" is the **file name** we want to open.
- "r" is the **mode** in which the file will be opened.

File Handling: Opening and Closing a File

fopen() Function:

- function opens the file on the disk and returns a pointer that can be used to access the file's content.
- If the file does not exist, fopen() returns a NULL pointer, indicating failure to open the file.

fclose() Function:

- After the program finishes working with the file, it **MUST** be closed to free up system resources. The fclose() function handles this task.
- Only the fclose() function can properly close the file that was opened with fopen().
- Example:

```
int fclose(FILE *fp);
```

- **Why is it important to emphasize closing the file if it's not stored in memory?**

File Opening Modes

1. **Read Mode** (r) – Used to read data from a file.
   - If the file exists, a pointer is set to the first character in the file.
   - If the file does not exist, fopen() returns NULL.
2. **Write Mode** (w) – Used to write data to a file.
   - If the file exists, its contents are overwritten (**truncated**).
   - If the file does not exist, a **NEW** file is created.
3. **Append Mode** (a) – Used to add data to an existing file.
   - If the file exists, data is added at the end without modifying existing content.
   - If the file does not exist, a **NEW** file is created.

Extended File Opening Modes

4. Read/Write Mode (r+) extends the read mode.

- It allows reading existing data, writing new data, and modifying existing content.
- The file **must already exist**.
- If the file does not exist, fopen() returns NULL.

5. Write/Read Mode (w+) – This is similar to the write mode.

- It allows writing new data and modifying already existing content.
- If the file exists, it is **truncated** (erased).
- If the file does not exist, a new file is created.

6. Append/Read Mode (a+) – This mode is similar to append mode.

- It allows reading existing content and appending new data to the end of the file.
- However, existing content cannot be modified.

Exercise 1

- Write a C program that attempts to open a file named "data.txt" in read mode.
- If the file cannot be opened (for instance, if it does not exist or there is a permissions issue), the program should display an error message and exit.

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

int main()
{
    FILE *fp;
    fp = fopen("data.txt", "r");

    if (fp == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    else
        printf("File opened successfully\n");

    // File operations can go here...

    fclose(fp);
    return 0;
}
```

File Permissions (FYI)

When encountering **file permissions issues**, it typically means your program doesn't have the necessary access to **read from**, **write to**, or **create files** in the current directory.

- **On Linux/macOS**: Run `ls -l` in the terminal to view the permissions of the directory where the file is being created or opened.
- You should see permissions for the directory like this: drwxr-xr-x.
- If permissions are insufficient, run the following command to allow write access:

  chmod +w <file_name>

- **On Windows**:
  - Right-click on the file, go to **Properties**, and check the **Read-Only** checkbox in the **General** tab.
  - Uncheck it if it's selected.

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

int main(void)
{
    FILE *fp;
    fp = fopen("data.txt", "w");

    if (fp == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    else
        printf("File opened successfully\n");

    // File operations can go here...

    fclose(fp);
    return 0;
}
```

Exercise 2

- Write a C program that attempts to create a file named "data.txt" using write mode (w).
- The program should display an error message if the file cannot be opened or created (for example, due to permission issues).
- If the file is created successfully, the program should output a success message and then close the file.

Random Access Files

Random Access Files (Binary) File

- When working with files in C, you can open them in **binary** and standard **text modes**.
- **Binary mode** is used when handling non-text data such as images, audio files, or other types of raw data where you don't want the system to modify the file's content (like converting line endings '\n').
- Binary mode ensures that the file's contents are read or written exactly as they are, **byte-**by-**byte**, without any conversion.

Reading in Binary Mode

- **Purpose** – Opens a file for reading in binary mode. The file must exist, or the operation will fail, returning a NULL pointer.
- **Example**:

```
FILE *fp = fopen("data.dat", "rb");
```

- In this example, the file "data.dat" is opened in binary mode for reading.
  - The b in "rb" indicates **binary mode**, which prevents the system from interpreting any content as text.
  - The program reads the raw binary data without altering it (e.g., converting line breaks or special characters).

Writing in Binary Mode

- **Purpose** – Opens a file for writing in binary mode. If the file already exists, its contents are **overwritten**. If the file does not exist, a new file is created.
- **Syntax**:

```
FILE *fp = fopen("data.dat", "wb");
```

- In this example, the file "data.dat" is opened in binary mode for writing.
  - The b in "wb" tells the system that the file should be treated as a binary file.
  - Any existing data in "data.dat" will be lost, and new binary data will be written starting from the beginning of the file.

File Opening in Binary Modes

Mode	Description
rb	Read from an existing binary file.
wb	Write to a new or existing binary file (truncating the content).
ab	Append to a binary file.
r+b / rb+	Read and write to an existing binary file.
w+b / wb+	Write and read to/from a new or existing binary file (truncating the content).
a+b / ab+	Append and read from a binary file, writing only at the end.

Retrieving Data from Files

Retrieving Data from Files

- Once a file is opened the next step is to read data from it.
- Functions like fgetc() and fgets() allow you to retrieve data from an open file.
  - fgetc() reads one character at a time, while fgets() reads a string of characters into an array.
- These operations happen after the file has been successfully opened.

Reading Characters from a File

- We have the `fgetc` and `getc` functions, which read one character at a time from a file.
- They work similarly and have the same syntax: `ch = fgetc(fp);`
  - `fgetc(fp)` reads the next character from the file stream pointed to by `fp`.
  - Returns the character read as an `int` or `EOF` if the end of the file is reached or an error occurs.
  - `fgetc()` might be slightly slower but more consistent across platforms
- Ideal for reading files character by character.
- P.S `EOF` stands for END OF FILE

Practical Exercise 1/3

- Imagine you're given the task of creating a program to count votes in an election.
- All votes are saved in a text file (votes.txt), and your goal is to write a program that reads all the votes from the file and counts how many votes each candidate received.

votes.txt:

votes.txt

```
1  A
2  A
3  B
4  C
5  A
```

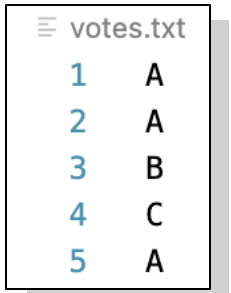
```c
int main()
{
    FILE *ptr;
    char vote;
    // We have three candidates: A, B, and C
    // Initialize the vote counts to 0
    int countA = 0, countB = 0;
    int countC = 0, countOthers = 0;

    // Open the file for reading
    ptr = fopen("char.txt", "r");

    if (ptr == NULL)
    {
        printf("Error: Could not open the file.\n");
        return 1;
    }
```

The first part of the solution

Practical Exercise 2/3

- Imagine you're given the task of creating a program to count votes in an election.
- All votes are saved in a text file (votes.txt), and your goal is to write a program that reads all the votes from the file and counts how many votes each candidate received.

votes.txt:



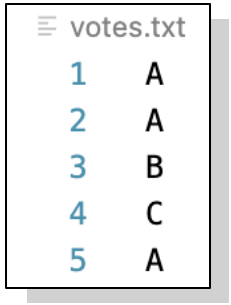

```c
// Read the votes from the file and tally them
while ((vote = getc(ptr)) != EOF)
{
    if (vote == 'A')
        countA++;
    else if (vote == 'B')
        countB++;
    else if (vote == 'C')
        countC++;
        // Ignore newlines and spaces
    else if (vote != '\n' && vote != ' ')
    // Count any other characters as invalid votes
    // or candidates
        countOthers++;
}
```

The second part of the solution

Practical Exercise 3/3

- Imagine you're given the task of creating a program to count votes in an election.
- All votes are saved in a text file (votes.txt), and your goal is to write a program that reads all the votes from the file and counts how many votes each candidate received.

votes.txt:




```c
    // Close the file
    fclose(ptr);

    // Display the vote count for each candidate
    printf("Votes for Candidate A: %d\n", countA);
    printf("Votes for Candidate B: %d\n", countB);
    printf("Votes for Candidate C: %d\n", countC);
    printf("Invalid or other votes: %d\n", countOthers);

    return 0;
}
```

Last part of the solution

Reading Strings from a File

- `fgets` reads a line or string from a file and stores it in a character array.
- `fgets` reads up to **n-1** characters (to leave space for the null terminator `\0`) and stops reading when it encounters a newline character `'\n'` or `EOF`.
- Syntax: `fgets(array, n, fp);`
  - array: The character array where the string is stored.
  - n: Maximum number of characters to read.
  - fp: The file pointer.
- Return Value:
  - Returns the **pointer** to the array if successful.
  - Returns **NULL** if `EOF` is reached or an error occurs.

Example:

```
char str[50];
fgets(str, 50, fp);
```

End of File (EOF) and Error Handling

- When creating a file, the operating system recognizes that the last character has been written and sends an EOF signal to indicate the end of the file.
- Example:

```
while( ch = fgetc(fptr) !=EOF)
        printf("%c",ch);
```

- Always check for errors when opening or working with files.

Writing Chars to a File

- fputc() or putc(): Both functions are used to write a **single character** to a file at the position specified by the file pointer.
- Syntax: `putc(c, fp);`

  Where,
  - `c`: The character to be written
  - `fp`: The file pointer that points to the file

- File Pointer Movement – After writing the character, the file pointer automatically moves one position forward in the file.
- fputc() might be slightly slower but more consistent across platforms

fputc Example

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char c;
    int i;

    // Open file for writing
    fp = fopen("output.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Loop to get 5 characters from user
    for (i = 0; i < 3; i++) {
        printf("Enter character %d: ", i+1);
        scanf(" %c", &c);
        fputc(c, fp);  // Write using fputc()
    }

    fclose(fp);
    printf("Characters written to file successfully!\n");

    return 0;
}
```

Writing a String to a File

- `fputs()` function writes a character array or string to a file at the position specified by the file pointer.
- Syntax: `fputs(str, fptr);`

  Where

  `str`: The character array or string to be written.

  `fptr`: The file pointer that points to the file.

fputs Example

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char str[50];

    // Open file for writing
    fp = fopen("strings.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Get string from user
    printf("Enter a string (up to 50 chars): ");
    fgets(str, 50, stdin);  // Use fgets to read the string

    // Write string to file
    fputs(str, fp);

    fclose(fp);
    printf("String written to file successfully!\n");

    return 0;
}
```

Files with Different Data Types

Read Different Data Types

- fscanf() allows reading input from files, including different data types such as characters, integers, floats, and strings.
- When reading from standard input (keyboard), fscanf() works just like scanf(), allowing you to read formatted input directly from the user.
- Use format specifiers to read multiple data types: %c for character, %d for integer, %f for float, and %s for string.
- The fscanf() function returns the **number** of **successful inputs** it has read.

- How to use:

```
// Reads a char, an int, and a float
fscanf(fp, "%c %d %f", &ch, &num,
&flt);
```

- Error Handling:

```
if (fscanf(fp, "%c %d %f", &ch,
&num, &flt) != 3) {
    printf("Error: Invalid input or
less than expected inputs.\n");
}
```

Write Different Data Types

- fprintf() allows writing formatted data to a file, supporting different data types like characters, integers, floats, and strings.
- When writing to standard output (console), fprintf() works like printf(), allowing you to output formatted text directly.
- Use format specifiers to write multiple data types: %c for character, %d for integer, %f for float, %s for string
- fprintf() returns the **number** of characters **successfully written** to the file.

- How to use:

```
// Writes a char, an int, and a float //
to the file
fprintf(fp, "%c %d %f", ch, num, flt);
```

- Error Handling:

```
if (fprintf(fp, "%d", num) < 0) {
    printf("Error writing to file\n");}
```

Exercise: Storing User Registration Information 1/2

- Imagine you're developing a user registration system for a small application. When new users sign up, their names and ages must be saved to a file for future reference or logging.
- The system asks the user to enter their name and age and then stores it in a file called "rec.dat"
- This file is a simple database that stores all user names entered during registration.

Reg.c

The first part of the solution

```c
int main(void)   // Correct function declaration
{
    // 1. Declare a file pointer
    FILE *fp;

    // 2. Declare a character array to
    // store the user's name and an integer for age
    char name[10];
    int age;

    // 3. Open the file in write mode
    fp = fopen("rec.dat", "w");
    if (fp == NULL)
    {
        // Return an error code if the file can't be opened
        printf("Error opening file!\n");
        return 1;
    }

    // 4. Prompt user for input
    printf("Enter your name and age: ");

    // 5. Take user input for the name and age
    scanf("%s %d", name, &age);
```

Exercise: Storing User Registration Information 2/2

- Imagine you're developing a user registration system for a small application. When new users sign up, their names and ages must be saved to a file for future reference or logging.
- The system asks the user to enter their name and age and then stores it in a file called "rec.dat"
- This file is a simple database that stores all user names entered during registration.

```c
// 6. Check if the input is valid
if (age <= 0) {
    printf("Invalid age. Please enter a positive number.\n");
    fclose(fp);
    return 1;
}

// 7. Write the name and age to the file
fprintf(fp, "Name: %s, Age: %d\n", name, age);

// 8. Close the file
fclose(fp);

printf("Name and age saved successfully!\n");
return 0;
```

The second part of the solution

Reg.c

Reading and Writing Blocks of Data

What is a Random-Access File?

- In a random-access file, you can directly go to any position in the file and read or write data without reading the entire file sequentially.
- This allows non-sequential (random) access to any part of the file, particularly useful when dealing with complex data structures.

Reading and Writing Blocks of Data

- The fread() and fwrite() functions enable reading and writing blocks of data, such as **arrays** or **structures**, which is essential for random access.
- Rather than handling data one byte or character at a time, these functions allow for manipulating entire memory blocks in a single operation.
- These functions, along with file positioning functions like fseek(), enable random access within a file.

fseek() moves the file pointer to a specified location within the file, allowing you to read or write data at that position.

Using fseek() for Random File Access

- fseek() allows you to move the file pointer to a specific location in a file. It enables reading and writing at any position without processing the entire file sequentially.
- Syntax: int fseek(FILE *fp, long int offset, int where);
  - fp: The file pointer
  - offset: The **number of bytes** to move the file pointer.
  - where: The reference point for the file pointer movement. It can take the following values:
    - **SEEK_SET**: Beginning of the file.
    - **SEEK_CUR**: Current position of the file pointer.
    - **SEEK_END**: End of the file.

Using fseek() for Random File Access (cont.)

Using `fseek`:

```
// Move file pointer to the 10th byte from the beginning of the file
fseek(fp, 10, SEEK_SET);
// Move file pointer 5 bytes backward from the current position
fseek(fp, -5, SEEK_CUR);
// Move to the end of the file
fseek(fp, 0, SEEK_END);
```

Return Value:

- Returns 0 on success.
- Returns a non-zero value if an error occurs (e.g., invalid file pointer or invalid offset).

Use Cases:

- Updating a specific record in a large file without reading the entire file.
- Randomly accessing data blocks in binary files.

Example: Writing and Reading at Random Positions

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

int main(void) {
    // Declare a file pointer
    FILE *fp;

    // Open the file in write mode
    // ("wb+" creates the file if it doesn't exist)
    fp = fopen("data.dat", "wb+");

    // Array to write to the file
    int data[5] = {10, 20, 30, 40, 50};

    // Write array of 5 integers to the file
    if (fwrite(data, sizeof(int), 5, fp) != 5) {
        printf("Error writing to file.\n");
        fclose(fp);
        return 1;
    }
    printf("Data successfully written to file.\n");

```

```c
    // Move file pointer back to the start for reading
    rewind(fp);

    // Array to store the data read from the file
    int read_data[5];
    if (fread(read_data, sizeof(int), 5, fp) != 5) {
        printf("Error reading from file.\n");
        fclose(fp);
        return 1;
    }

    // Print data read from the file
    printf("Data read from file: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", read_data[i]);
    }
    printf("\n");

    // Close the file
    fclose(fp);

    return 0;
}
```

Rand.c

Practical Exercise: A Student Management System

- Imagine you are building a **student management system**. You might need to store each student's information (like **SID** and name) in a file so that you can retrieve it later.
- Each time you run the program, it can store and later retrieve student records.

Step #1: Defining a Structure

- A structure student is defined as follows:

```
// Define the structure for the student
struct student {
    char SID[10];  // Student ID
    char name[20]; // Student Name
};
```

Step #2: Open the File to Read or Write

- The student structure has two fields: SID (Student ID) and name. This structure will store each student's information in the file.
- The program will open the file "students.dat" in **binary read/write mode** (rb+). If the file doesn't exist, the program will create the file using **binary write mode** (wb+). This ensures that the file is always available for reading and writing.
- We use binary file modes (rb+, wb+) to read and write data as raw binary blocks.

```
// Try to open the file in read/write mode
fptr = fopen("students.dat", "rb+");
if (fptr == NULL) {
    fptr = fopen("students.dat", "wb+");  //
Create the file if it doesn't exist
    if (fptr == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
}
```

Step #3: Writing a Record at a Specific Position

- In the following snippet, the program allows the user to input a student's SID and name. Using fseek(), we move the file pointer to the specific position in the file where the record should be written.
- Here fseek() is used to move the file pointer to the correct position in the file based on the record number specified by the user.
- The position is calculated by multiplying the record number by the size of the structure (sizeof(struct student)).

```
void write_record(FILE *fptr, int record_num) {
    struct student st;

    // Get user input
    printf("Enter student ID: ");
    scanf("%d", &st.SID);
    printf("Enter name: ");

    fgets(st.name, sizeof(st.name), stdin);

    // Remove the newline character
    st.name[strcspn(st.name, "\n")] = '\0';

    // Move the file pointer to the appropriate position
    fseek(fptr, record_num * sizeof(struct student),
SEEK_SET);

    // Write the record to the file
    fwrite(&st, sizeof(struct student), 1, fptr);
    printf("Record written successfully!\n");
}
```

Step #4: Reading a Record from a Specific Position

- The program allows users to retrieve a student's information from a specific record position. Using fseek() moves the file pointer to the correct position to read the data.
- Reading the Structure: fread() is used to read the student record from the file.
- It is displayed if the data exists at the specified position; otherwise, the program informs the user that no record was found.

```
void read_record(FILE *fptr, int record_num) {
    struct student st;

    // Move the file pointer to the appropriate position
    fseek(fptr, record_num * sizeof(struct student),
SEEK_SET);

    // Read the record from the file
    if (fread(&st, sizeof(struct student), 1, fptr) == 1)
{
        // Display the student's information
        printf("Student ID: %d\n", st.SID);
        printf("Name: %s\n", st.name);
    } else {
        printf("No record found at this position!\n");
    }
}
```


thank you

References

- Data Structures Using C, second edition, by Reema Thareja, Oxford University Press, 2014.