UNIT - V

Introduction to Files

In every programming language, data plays a central role. However, the data that a program stores in variables exists only as long as the program is running. Once the program ends, the data disappears. To overcome this limitation and to make data permanent, we use files**.**

Files allow us to store information permanently, retrieve it whenever needed, and process large amounts of data efficiently.

Similarly, when working with a collection of data—such as a list of values—it becomes necessary to search for specific items and arrange (sort) them in a particular order.
These operations, known as searching and sorting, form the foundation of data processing in programming.

Files in C

What is a File ?

A file is a named storage unit on a secondary storage device that holds data permanently, allowing programs to read from and write to it even after the program terminates.

Why do we need files?

- To store data permanently
- To read/write large amounts of data
- To transfer data between programs
- To preserve user records, logs, scores, etc.

Simple Real-Life Example

- When you write in a notebook, the writing stays there until erased → **File**
- When you write on a whiteboard, it disappears when wiped → **Variables**

File Handling

File handling refers to the process of creating, opening, reading, writing, appending, and closing files in a programming language to manage data efficiently.

Types of Files

Files are broadly classified into two categories based on how they store data:

A. Text Files

A text file is a file that stores data in human-readable characters, where each character is saved in ASCII or Unicode format.

Examples:

- notes.txt
- program.c
- students.csv

Characteristics:

- Easy to read and edit using any text editor
- Takes more storage space
- Slower to process compared to binary files

Example of data in a text file

101 John 450.50

102 Riya 500.00

B. Binary File

Binary search is an efficient searching technique that repeatedly divides a sorted array into two halves and checks the middle element to locate the target.

Characteristics:

- Compact and faster to read/write
- Data is stored more accurately
- Requires special functions for reading/writing

Examples:

- Images, Audio, Video files
- .exe (executables)

- .dat (binary data files)

File Operations in C

To work with files, C provides a set of operations using the FILE data type declared in <stdio.h>.

Common operations include:

1. Creating a file
2. Opening a file
3. Reading from a file
4. Writing to a file
5. Appending data
6. Closing the file

1. Creating a File

Creating a file means allocating a new storage space on the disk and preparing it to store data under a specified name. If the file already exists, depending on mode, it may be overwritten or left unchanged.

A file is created when it is opened in a mode that supports creation such as:

- "w" – write mode (creates new file or overwrites existing)
- "a" – append mode (creates file if it doesn't exist)
- "w+", "a+" – read/write versions

Syntax:

```
FILE *fp = fopen("filename.txt", "w");
```

Example: Write a C program to create a file named "demo.txt".

Code:

```
#include <stdio.h>

int main() {

   FILE *fp = fopen("demo.txt", "w");

   if(fp == NULL)
```

```
        printf("Unable to create file.");
    else
        printf("File created successfully.");
    fclose(fp);
    return 0;
}
```

2. Opening a File

Opening a file means establishing a connection between the file in storage and the program. After opening, the program can read or write using the file pointer.

File Opening Modes

Mode	**Meaning**
"r"	Open for reading (file must exist)
"w"	Open for writing (new file created, old content removed)
"a"	Open for appending (writes at end; creates file if missing)
"r+"	Read + Write (file must exist)
"w+"	Read + Write (creates new, overwrites existing)
"a+"	Read + Append

Syntax:

```
FILE *fp = fopen("filename.txt", "r");
```

Example: Open a file in read mode and check if it exists.

Code:

```
FILE *fp = fopen("data.txt", "r");
if(fp == NULL)
    printf("File does not exist.");
```

```
else

    printf("File opened successfully.");
```

3. Reading from a File

Reading from a file means retrieving and loading stored data from a file into the program’s memory

Common Functions for Reading

1. **fgetc(fp)** – reads a single character
2. **fgets(str, size, fp)** – reads a string (line)
3. **fscanf(fp, format, &var)** – reads formatted input (like scanf)

Example : Reading Character-by-Character

Program to read a file character by character.

```
char ch;

FILE *fp = fopen("notes.txt", "r");

while((ch = fgetc(fp)) != EOF)

    putchar(ch);

fclose(fp);
```

Example: Reading a Line Using fgets()

Read and display one line from a text file.

Code:

```
char line[100];

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

fgets(line, sizeof(line), fp);

printf("%s", line);

fclose(fp);
```

Example: Read name and age stored in a file.

Code :

```
// Reading Formatted Data Using fscanf()

char name[20];

int age;

FILE *fp = fopen("info.txt", "r");

fscanf(fp, "%s %d", name, &age);

printf("Name = %s\nAge = %d", name, age);

fclose(fp);
```

4. Writing to a File

Writing means **storing new data** into a file. If opened in write mode "w", the old data is erased. If using "a", new data is added at the end.

Functions Used

1. **fputc(ch, fp)** – writes a character
2. **fputs(str, fp)** – writes a string
3. **fprintf(fp, format, values)** – writes formatted data

Example :Write a String Using fputs()

```
FILE *fp = fopen("output.txt", "w");

fputs("Welcome to File Handling!", fp);

fclose(fp);
```

Example: Write a Character Using fputc()

```
FILE *fp = fopen("char.txt", "w");

fputc('A', fp);

fclose(fp);
```

Example: Write Formatted Output Using fprintf()

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

fprintf(fp, "Name: %s\nMarks: %d", "Ravi", 90);

fclose(fp);
```

5. Appending a File

Appending means adding new data at the end of an existing file without modifying its previous content.

Opening Mode for Appending

- "a" → write at end (file created if not existing)
- "a+" → read + write at end

Example: Append a new line to an existing file.

```
FILE *fp = fopen("log.txt", "a");

fputs("New entry added.\n", fp);

fclose(fp);
```

Use Cases:

Appending is useful in:

- Log files
- Adding new records
- Maintaining history of operations

6. Closing a File

Closing a file means terminating the connection between the file and the program, ensuring all data is saved and no corruption occurs.

Importance:

- Ensures data is saved correctly
- Frees system resources
- Avoids corruption and memory leaks

Syntax:

```
fclose(fp);
```

Example:

```
FILE *fp = fopen("sample.txt", "r");

if(fp != NULL) {

    // file actions

    fclose(fp);

}
```

File Accessing Methods

File accessing methods describe how data is retrieved or written in a file.
They determine the order in which data is accessed**.**

There are two major methods:

1. Sequential Access
2. Random (Direct) Access

1. Sequential Access

Sequential access is a file accessing method where data is read or written in order, from the beginning to the end, one record after another**.**

We cannot jump directly to any position; you must pass all previous data.

It works like a tape recorder – to reach the 10th record, you must pass record 1 to 9.

Characteristics

- Data is accessed in a fixed, linear order
- Reading starts from the beginning of file
- Best suited for text files, logs, reports
- Reading/writing continues until EOF (End of File)
- Faster when processing entire files
- Cannot jump/skip to a position directly
- Easy to use and widely supported

Sequential access uses simple functions like:

- fgetc() → read character by character
- fgets() → read line by line
- fscanf() → read formatted data
- EOF detection using feof()

Example: Program to read a file sequentially and display its content

Code:

```
#include <stdio.h>

int main() {

   FILE *fp;

   char ch;

   fp = fopen("sample.txt", "r");

   if(fp == NULL) {

      printf("File not found.");

      return 0;

   }

   while((ch = fgetc(fp)) != EOF)

      putchar(ch);
```

```
    fclose(fp);

    return 0;

}
```

2. Random Access (Direct Access)

Random access allows the program to directly jump to any location (byte/record) in a file without reading earlier data. This is achieved using “file pointer repositioning” functions.

This is like playing a video and jumping to any timestamp instantly.

Characteristics

- Access any part of the file instantly
- Does not require reading previous content
- Best for large files or structured records
- Useful in databases, indexing, editing, binary files
- More complex than sequential access
- Supported mainly through fseek() and ftell()

Important Functions for Random Access

1. fseek() — Move File Pointer

Syntax :

```
fseek(FILE *fp, long offset, int position);
```

Position values :

Position	**Meaning**
SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End of file

Example: Program to read the 10th character from a file using fseek()

Code:

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

```
int main() {
    FILE *fp = fopen("data.txt", "r");
    char ch;
    if(fp == NULL) {
        printf("File not found.");
        return 0;
    }
    fseek(fp, 9, SEEK_SET);   // Move to 10th character (index 9)
    ch = fgetc(fp);
    printf("10th character: %c", ch);
    fclose(fp);
    return 0;
}
```

2. ftell() — Get Current File Pointer Position

Syntax

```
long pos = ftell(fp);
```

Example: Find file size

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "r");
    fseek(fp, 0, SEEK_END);
    long size = ftell(fp);
    printf("File size: %ld bytes", size);

    fclose(fp);

    return 0;

}
```

3. rewind() — Move Pointer to Beginning

Syntax

rewind(fp);

Example:

rewind(fp); // moves pointer to beginning

SEARCHING

Searching is the process of locating a required element from a collection of data (typically an array). Efficient searching reduces the time required to find an element, especially for large data sets.

There are two fundamental searching approaches:

1. Linear Search (Sequential Search)
2. Binary Search (Dichotomic Search)

Basic Searching in Arrays :

Searching in arrays refers to the method of finding whether a given key (target value) exists in the array, and if yes, returning its index or indicating its presence.

General Process of Searching

1. Compare the target element with the elements of the array.
2. Use a searching strategy (Linear or Binary) to locate it.
3. Return the location/index or -1 if not found.

1. LINEAR SEARCH

Linear Search, also called Sequential Search, is the simplest searching technique where each element is checked one after another until the target element is found or the entire array has been scanned.

It follows a brute-force approach: “Start from index 0 → compare each element → stop when found.”

Feature	**Description**
Search Type	Sequential
Works on	Sorted and Unsorted arrays
Time Complexity (Worst/Avg)	**O(n)**
Best Case Complexity	**O(1)** (element at first position)
Space Complexity	**O(1)**
Simple Implementation	Yes
Speed	Slower for large datasets
Preferred When	Array is small OR unsorted

Syntax

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;   // element found
    }
    return -1;          // element not found
}
```

Example : Linear Search in an Array

Code:

```
#include <stdio.h>
int main() {
    int arr[] = {21, 45, 11, 89, 77};
    int key = 89;
    int n = 5;
    int flag = -1;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            flag = i;
            break;
        }
    }
    if (flag != -1)
        printf("Element %d found at index %d\n", key, flag);
    else
        printf("Element not found\n");
    return 0;
}
```

Output:

```
Element 89 found at index 3
```

2. BINARY SEARCH

Binary Search is an efficient searching technique based on the Divide and Conquer strategy.
It repeatedly divides the sorted array into halves and checks whether the target lies in the left half or right half.

Note: The array must be sorted.

Binary Search Steps

1. Find middle index:
   mid = (low + high) / 2
2. Compare key with arr[mid]
   - If key == arr[mid] → Found
   - If key < arr[mid] → Search left half
   - If key > arr[mid] → Search right half
3. Repeat until low > high

Characteristics :

Feature	**Description**
Search Type	Divide-and-Conquer
Works On	Sorted arrays only
Time Complexity (Worst/Avg)	O(log n)
Best Case	O(1)
Space Complexity	O(1) for iterative, O(log n) for recursive
Speed	Very fast for large datasets
Comparison	Requires fewer comparisons than Linear Search

Syntax :

```
int binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;
   while (low <= high) {
      int mid = (low + high) / 2;
      if (arr[mid] == key)
         return mid;
      else if (key < arr[mid])
         high = mid - 1;
      else
         low = mid + 1;
   }
   return -1;
}
```

Example : Successful Search

Code:

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

```
int main() {
    int arr[] = {5, 10, 15, 20, 25, 30};
    int key = 20;
    int low = 0, high = 5, mid;
    int flag = -1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == key) {
            flag = mid;
            break;
        }
        else if (key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    if (flag != -1)
        printf("Element found at index %d\n", flag);
    else
        printf("Element not found\n");
    return 0;
}
```

Output:

```
Element found at index 3
```

Binary Search – Recursive Version

```
int binarySearchRec(int arr[], int low, int high, int key) {
    if (low > high)
        return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key)
        return mid;
    else if (key < arr[mid])
        return binarySearchRec(arr, low, mid - 1, key);
    else
        return binarySearchRec(arr, mid + 1, high, key);
}
```

SORTING

Sorting is the process of arranging the elements of a list or array in a particular order, most commonly in **ascending** (small → large) or **descending** (large → small) order.

Sorting is essential because:

- It improves searching efficiency (Binary Search requires sorted data)
- It organizes data for reports and analysis
- It is used in databases, file systems, and data structures

Types of Sorting Algorithms

1. **Bubble Sort**
2. **Insertion Sort**
3. **Selection Sort**

1. Bubble Sort

Bubble Sort is a simple comparison-based sorting technique in which **adjacent elements** are compared and swapped if they are in the wrong order. This repeated pass causes the largest element to “bubble up” to its correct position at the end of the array.

Syntax / Algorithm

```
void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}
```

Step-By-Step Example

Sort: **[5, 2, 8, 1, 3]**

Pass 1:

5 > 2 → swap → [2 5 8 1 3]

5 < 8 → no swap

8 > 1 → swap → [2 5 1 8 3]

8 > 3 → swap → [2 5 1 3 8]

Largest element 8 settles at the end.

Pass 2:

2 < 5 → no swap

5 > 1 → swap → [2 1 5 3 8]

5 > 3 → swap → [2 1 3 5 8]

Final Sorting Result

[1 2 3 5 8]

2. Insertion Sort

Insertion Sort is a simple sorting technique that builds the sorted list one element at a time by inserting each new element into its proper position among the previously sorted elements.

Syntax / Algorithm

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Step-By-Step Example

Sort: [7, 4, 5, 2]

i = 1 (key = 4)

7 > 4 → shift

Array becomes: [7, 7, 5, 2]

Insert 4 → [4, 7, 5, 2]

i = 2 (key = 5)

7 > 5 → shift → [4 7 7 2]

Insert 5 → [4 5 7 2]

i = 3 (key = 2)

7 > 2 → shift

5 > 2 → shift

4 > 2 → shift

Insert 2 → [2 4 5 7]

Final Result:

2 4 5 7

3. SELECTION SORT

Selection Sort is a sorting technique that repeatedly selects the smallest element from the unsorted portion of the array and places it at its correct sorted position.

Syntax / Algorithm

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min])
                min = j;
        }
        int temp = arr[min];

        arr[min] = arr[i];

        arr[i] = temp;

    }

}
```

Step-By-Step Example

Sort: **[64, 25, 12, 22, 11]**

Pass 1:

Find smallest in full array → **11**
Swap 11 with 64
[11, 25, 12, 22, 64]

Pass 2:

Smallest in [25,12,22,64] → **12**
Swap with 25
[11, 12, 25, 22, 64]

Pass 3:

Smallest in [25,22,64] → **22**
Swap with 25
[11, 12, 22, 25, 64]

Pass 4:

Smallest in [25,64] → 25
Already in place.

Final Result:

[11, 12, 22, 25, 64]