Fundamental Of Programming

Exam Preparation:

1. Programming Language

A programming language is a special language used to write instructions which a computer can understand and execute.
It acts as a bridge between a human and a machine.

Types of languages include:

Low-level languages (machine/assembly)
High-level languages (C, Python, Java)

2. Compiler

A compiler converts the entire source program into machine language at once.
If errors exist, it reports them after the translation.

Example languages using compiler → C, C++

3. Interpreter

An interpreter translates and executes the program **line-by-line**.
It stops execution when an error occurs.

Example languages using interpreter → Python, PHP

4. Linker

A linker combines multiple program files, library files and object code into a single executable file.
For example, C programs often need libraries like stdio.h — linker connects them.

5. Loader

Loader loads the executable program into main memory (RAM) for actual execution.

6. Classification of Programming Languages

✔ **Procedural (C, Pascal)** – step-by-step logic
✔ **Object-oriented (Java, C++)** – uses objects and classes
✔ **Scripting (Python, JavaScript)** – fast development scripting
✔ **Functional (Haskell, Lisp)** – based on functions

7. Algorithm

An algorithm is a step-by-step logical sequence to solve a problem.

Example: Algorithm to add two numbers

1. Start
2. Input A
3. Input B
4. Calculate Sum = A + B
5. Display result
6. Stop

⭑ **8. Flowchart**

A flowchart is a graphical representation of an algorithm using standard symbols:

Oval → Start/Stop
Parallelogram → Input/Output
Rectangle → Process
Diamond → Decision

Flowcharts help visualize program logic easily.

9. Structure of a C Program

Basic components include:

→ **Header File Section** (e.g., #include <stdio.h>)
→ **Main Function** void main()
→ **Variable Declarations**
→ **Statements / Logic**
→ **Return Statement**

Example:

```
#include<stdio.h>

void main()

{

   printf("Hello World");

}
```

10. First C Program Explanation

#include<stdio.h> → enables input/output functions
main() → starting point of execution
printf() → prints output on screen
Semicolon ends every statement

11. Comments in C

Single-line comment → // This is comment
Multi-line comment → /* comment text */

Comments improve program readability — compiler ignores them.

12. C Tokens

Tokens are the smallest building blocks of a program.

Types of tokens:

1. Keywords (int, if, while)
2. Identifiers (variable names)
3. Constants (10, 'A', 3.14)
4. Operators (+, -, ==)
5. Strings ("Hello")
6. Punctuation (; , () {})

13. Data Types in C

Data Type	**Meaning**

int	integer values
float	decimal numbers
double	large decimal numbers
char	single character

Memory size varies by system.

14. Variables

A variable is a named memory location used to store data.

Example:
int age = 20;

Rules:
Must start with a letter or underscore
No spaces
Case-sensitive

15. Operators & Expressions

Arithmetic Operators

+ - * / %

Relational Operators

> < >= <= == !=

Logical Operators

&& || !

Assignment Operator

=, =+, = -, =*, =/

An expression is a combination of operands and operators:
Example: x = a + b * 10;

16. Type Conversion and Typecasting

Type Conversion (Implicit)
Automatic conversion by compiler
Example: int to float

Typecasting (Explicit)
Manual conversion
Example:

(float)x

(int)3.7

Unit-2

Basic Screen & Keyboard Input / Output in C

C uses the header file **<stdio.h>** for input-output functions.

Output: printf()

Used to display message or value on the screen.

Syntax:

printf("format specifiers or message", variable_name);

Examples:

printf("Hello World"); // prints text

printf("Value of a = %d", a); // prints integer

printf("Average = %f", avg); // prints float

Common format specifiers:

Specifier	Meaning

%d	int
%f	float
%c	char
%s	string (char[])

Escape sequences:

Code	Meaning
\n	new line
\t	tab space
\\	backslash \
\"	double quote

Example:

printf("Hello\nWorld");

Output:

Hello

World

Input: scanf()

Used to take values from the keyboard.

Syntax:

scanf("format specifiers", &variable_name);

Note: & (address-of operator) is required with variables (except strings).

Example:

```
int a;
float b;
scanf("%d", &a);
scanf("%f", &b);
```

2. Decision Control Statements (Conditional Execution)

if Statement

Used when we want to execute a block **only if** a condition is true.

Syntax:

```
if (condition) {
    // statements
}
```

Example:

```
if (marks >= 35) {
    printf("Pass");
}
```

if-else Statement

Do options ho: condition true → one block, false → another block.

Syntax:

```
if (condition) {
    // true block
} else {
```

```
    // false block
}
```

Example:

```
if (marks >= 35) {
    printf("Pass");
} else {
    printf("Fail");
}
```

📎 *else-if Ladder*

Multiple conditions check karne ke liye use karte hain.

Syntax:

```
if (condition1) {
    // block 1
} else if (condition2) {
    // block 2
} else if (condition3) {
    // block 3
} else {
    // default block
}
```

Example (grade system):

```c
if (marks >= 90)
    printf("Grade A");
else if (marks >= 75)
    printf("Grade B");
else if (marks >= 60)
    printf("Grade C");
else if (marks >= 35)
    printf("Grade D");
else
    printf("Fail");
```

📝 *Nested if*

if ke andar another if/else-if ho, to use **nested if** bolte hain.

Example:

```c
if (age >= 18) {
    if (citizen == 1) {
        printf("Eligible to vote");
    } else {
        printf("Not a citizen");
    }
} else {
    printf("Not eligible due to age");
}
```

📝 *switch Statement*

Syntax:

```
switch (expression) {
    case value1:
        statements;
        break;
    case value2:
        statements;
        break;
    default:
        statements;
}
```

Example:

```
int ch;
scanf("%d", &ch);
switch (ch) {
    case 1:
        printf("Monday");
        break;
    case 2:
        printf("Tuesday");
        break;
    default:
```

```
    printf("Invalid choice");
}
```

3. Looping Statements (Iterative)

Loops are used when we want to **repeat** some statements multiple times.

There are **3 main loops** in C:

1. while
2. do-while
3. for

while Loop

Condition pehle check hoti hai → phir body execute hoti hai.

Syntax:

```
while (condition) {
  // loop body
}
```

Example: print 1 to 5

```
int i = 1;
while (i <= 5) {
  printf("%d\n", i);
  i++;
}
```

do-while Loop

Yaha **body pehle execute hoti hai**, condition baad me check hoti hai.
Isliye isko **exit-controlled loop** bolte hain.
Minimum **1 baar** loop chalega hi.

Syntax:

```
do {
    // body
} while (condition);
```

Example:

```
int i = 1;
do {
    printf("%d\n", i);
    i++;
} while (i <= 5);
```

for Loop

Mostly used when number of iterations known ho.

Syntax:

```
for (initialization; condition; update) {
    // body
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    printf("%d\n", i);
}
```

4. Nested Loops

When **one loop is inside another loop**, it is called nested loop.

Most commonly used for:

- Patterns (stars / numbers)
- Matrices
- Tables

Example: simple pattern

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        printf("* ");
    }
    printf("\n");
}
```

Output:

```
* * *
* * *
* * *
```

5. Special Control Statements

Used to **change the normal flow** of loop / program.

break

Used to **terminate** loop or switch immediately.

Example:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5)
        break;
    printf("%d ", i);
}
```

Output: 1 2 3 4

continue

Skips the current iteration and moves to next loop cycle.

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3)
        continue;   // skip 3
    printf("%d ", i);
}
```

Output: 1 2 4 5

goto

Jump from one part of program to another using labels.

Syntax:

```
goto label;

...

label:

    statements;
```

goto may create unstructured and confusing code, so it should be avoided in normal programming.

Unit-3

WHAT IS AN ARRAY?

An array is a **collection of multiple values** of the **same data type**
stored in **continuous memory locations**,
and accessed using **index**.

Example real life:
Roll numbers list — 10 values stored together = array.

Why use arrays?

Because variables store only **one value**,
but arrays store **many values**.

Example:

```
int marks[5];
```

means a set of 5 integer values stored together.

1) Types of Arrays

Arrays are of two types:

One Dimensional Array (1D)
Multi-Dimensional Array (2D, 3D etc.)

Let’s learn them one by one 👇

ONE DIMENSIONAL ARRAY (1D Array)

Definition

1D array is a list/linear collection of values.

Declaration

```
datatype array_name[size];
```

Example:

```
int marks[5];
```

This stores 5 integers:
marks[0], marks[1], marks[2], marks[3], marks[4]

Index **always starts from 0**.

Initialization

Method 1:

```
int marks[5] = {10, 20, 30, 40, 50};
```

Method 2:

```
int marks[] = {10, 20, 30, 40, 50};
```

Method 3 (runtime input):

```
int i, marks[5];

for(i = 0; i < 5; i++)

{

   scanf("%d", &marks[i]);

}
```

Example: Program to read & print array elements

```
#include<stdio.h>

void main()

{

    int marks[5], i;

    printf("Enter 5 marks:\n");

    for(i = 0; i < 5; i++)

        scanf("%d", &marks[i]);

    printf("You entered:\n");

    for(i = 0; i < 5; i++)

        printf("%d ", marks[i]);

}
```

ARRAY OPERATIONS (1D)

Most common array questions in exam and practical:

Read values
Print values
Find maximum
Find minimum
Sum / average
Sorting / searching

Example: Find maximum in array

```
int max = arr[0];
```

```
for(i = 1; i < n; i++)
{
    if(arr[i] > max)
        max = arr[i];
}
printf("Maximum = %d", max);
```

2) TWO DIMENSIONAL ARRAY (2D Array)

Definition

2D array is an array of arrays — looks like a **table or matrix**.

Example:

```
int a[3][3];
```

This stores 9 values in 3 rows & 3 columns.

Initialization

```
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Input/Output Example (Matrix)

```
int a[3][3], i, j;


for(i=0;i<3;i++)
```

```
{
   for(j=0;j<3;j++)
      scanf("%d",&a[i][j]);
}

for(i=0;i<3;i++)
{
   for(j=0;j<3;j++)
      printf("%d ", a[i][j]);
   printf("\n");
}
```

3) STRING IN C

String = collection of characters ending with **null character \0**

Example:

char name[10] = "Atul";

Difference between **char** and **string**:

char stores 1 character
String stores many characters

String Input/Output Functions:

gets() → read complete string
puts() → print string
scanf("%s", str) → read one word string

Common String Functions

(in <string.h>)

strlen() — length
strcpy() — copy
strcmp() — compare
strcat() — concatenate (join)

Example:

```
char s1[20], s2[20];

gets(s1);

gets(s2);

strcat(s1, s2);

puts(s1);
```

4) Array of Strings

Example:

```
char names[3][10] = { "Ram", "Amit", "Atul" };
```

USER DEFINED FUNCTIONS IN C

What is a Function?

A function is a **self-contained block of code** designed to perform a specific task.

Why we use functions?

Reduces code repetition
Makes code organized
Improves reusability
Makes debugging easy

Example real life:
Calculator me +, –, ×, ÷ sab alag buttons — each works like a function.

Types of functions in C

Built-in functions

Already available in C libraries
printf(), scanf(), strlen(), sqrt(), etc.

User Defined Functions

Jo programmer khud banata hai.

Function Structure (3 parts)

A function has:

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

1. Function Declaration (Prototype)

Compiler ko batata hai "function exist karta hai".

Syntax:

```
return_type function_name(type parameters);
```

Example:

```
int add(int, int);
```

2. Function Definition

Actual task perform karta hai.

Syntax:

```
return_type function_name(type parameters)
{
```

```
    statements;
}
```

Example:

```
int add(int a, int b)
{
    return a + b;
}
```

3. Function Call

Main() ke andar function ko execute karte hain.

Example:

```
sum = add(5, 10);
```

Function Categories (Very important)

Functions with no parameter, no return

```
void show();
```

Functions with parameter, no return

```
void show(int x);
```

Functions with parameter and return

```
int add(int a, int b);
```

Passing Parameters

Call by Value (Default in C)

Original value change nahi hoti — copy send hota hai.

Return Statement

Return karta hai value calling function ko.

Example:

```
return a + b;
```

SCOPE OF VARIABLES

(Where a variable is accessible)

1) Local Scope

Variable declared **inside function or block**
Accessible only within that block

Example:

```
void fun()
{
    int x = 10; // local
}
```

2) Global Scope

Variable declared **outside all functions**
Accessible in entire program

Example:

```
int x = 10;
```

STORAGE CLASSES IN C

Storage class tells:

lifetime
visibility
storage location

of a variable.

C supports 4 storage classes:

auto

Default for local variables
Exists only inside function

Example:

```
auto int a;
```

static

Retains value between function calls
Lifetime full program run

Example:

```
static int count = 0;
```

extern

Used to access global variable defined in another file

register

Variable stored in CPU registers
Faster access

Example:

```
register int i;
```

RECURSION

Definition

Recursion is when a function **calls itself**.

Used for:
factorial
Fibonacci
searching & sorting techniques

Example: Factorial using recursion

```
int fact(int n)

{

   if(n == 0)

      return 1;

   else

      return n * fact(n - 1);

}
```

How recursion works?

1 → function call
2 → function calls itself again
3 → process repeats
4 → stopping condition (base case) stops calls

IMPORTANT:
Every recursion must have a **base condition** to stop infinite loop.

Unit-4

INTRODUCTION TO MEMORY ADDRESSES & ADDRESS OPERATOR

Every variable stored in memory has:

a **value**
and an **address** (location where the value is stored)

Example:

int x = 10;

– Value = 10
– Address = something like 1002 (no fixed address)

To access the address in C, we use:

&x → address of x

WHAT IS A POINTER?

👉 A pointer is a variable that **stores memory address** of another variable.

Normal variable → stores value
Pointer variable → stores address

Example:

int x = 10;

int *p;

p = &x;

Meaning:
p holds the address of x

Dereferencing Operator (*)

When we write:

*p

it means — **value stored at the address that p points to**

Example:

printf("%d", *p); // prints 10

So p is address, *p is value

NULL Pointer

Pointer with **no valid address**

int *p = NULL;

Used for safety to show that pointer is empty.

void Pointer

Generic pointer — can store the address of **any type**

void *p;

int x = 10;

p = &x;

But you must type-cast when using:

printf("%d", *(int*)p);

POINTERS AND ARRAYS

★ **Key rule:**

Array name itself represents pointer to its **first element**

Example:

int a[5] = {10,20,30,40,50};

int *p = a; // or = &a[0];

So:

a and p hold same address
*p = 10
*(p+1) = 20

Equivalent forms:

a[i] ≡ *(a+i)

p[i] ≡ *(p+i)

POINTER ARITHMETIC

Pointer arithmetic is allowed only on **same base type** pointers.

Valid operations:

++
--
+ integer
- integer

Example:

int a[3]={10,20,30};

int *p=a;

p++; // moves to next element

printf("%d", *p); // prints 20

Pointer increment increases address according to size of datatype, not by byte 1 — e.g., if int = 4 bytes → address increases by 4.

POINTER TO POINTER

Pointer storing address of another pointer.

Example:

```
int x = 10;

int *p = &x;

int **pp = &p;
```

p → address of x
pp → address of p

*p = 10
**pp = 10

ARRAY OF POINTERS

Pointer array means array storing addresses.

Example storing strings:

```
char *names[3] = {"Ram", "Amit", "Atul"};
```

Each element holds address of a string.

POINTER TO AN ARRAY (Conceptual)

Difference:

```
int *p;      // pointer to int

int (*p)[5]; // pointer to array of 5 ints
```

This appears only in theory; rarely used in first semester coding.

POINTER TO FUNCTION

You can store function's address inside pointer.

Example:

```
int add(int x, int y)
{
    return x + y;
}
int (*fp)(int,int); // pointer to function
fp = add;
fp(3,4); // calls add(3,4)
```

Concept:
Functions also live in memory, so their address can be used.

STRUCTURE

What is a Structure?

A structure is a **user-defined datatype** used to group **different types of data** under one name.

Example: A student record contains:

- roll number (int)
- name (string)
- marks (float)

Structure allows combining these into one unit.

Structure Declaration

```
struct student {
    int roll;
    char name[20];
    float marks;
};
```

Structure Variable Creation

```
struct student s1;
```

Structure Member Access (Dot operator)

```
s1.roll = 101;

strcpy(s1.name,"Atul");

s1.marks = 90.5;
```

Structure Initialization

```
struct student s1 = {101, "Atul", 90.5};
```

STRUCTURE WITH ARRAY

We can create an array of structure:

```
struct student s[20]; // 20 students
```

Used to store records of multiple people —
like your practical question on 20 students.

STRUCTURE INSIDE STRUCTURE (NESTING)

Example:

```
struct date {

   int day, month, year;

};

struct employee {

   char name[20];

   struct date joining;  // nested structure

   float salary;

};
```

Accessing nested fields:

```
emp.joining.day = 10;
```

STRUCTURE WITH FUNCTIONS

Passing structure by value:

```
void display(struct student s)
{
   printf("%d %s %.2f", s.roll, s.name, s.marks);
}
```

Passing structure using pointer:

```
void display(struct student *p)
{
   printf("%d %s %.2f", p->roll, p->name, p->marks);
}
```

p->roll is same as (*p).roll

STRUCTURE WITH POINTER

Example:

```
struct student s1;
struct student *p;
p = &s1;
p->roll = 10;
```

Pointer required when passing structure efficiently to functions.

UNION

Union looks like structure but **memory behavior is different**.

Definition

A union is a user-defined datatype where **all members share the same memory location.**

Only **one member value remains valid at a time.**

Declaration

```
union data {
    int i;
    float f;
    char ch;
};
```

Key Difference: Structure vs Union

Structure	Union
Separate memory for each member	Shared memory for all members
Size = sum of all members	Size = largest member
All members can be used independently	Only one valid at a time

Example:

```
union data d;
d.i = 10;

d.f = 3.14;
```

After assignment to d.f, value stored in d.i becomes invalid.

Unit 5

1. What is a File?

So far, your programs mostly dealt with **keyboard input** and **screen output** (using scanf / printf).

But what if you want to:

- Save data permanently?
- Read data later, even after the program ends?

For that, we use **files**.

A *file* is a named area in storage (like hard disk) where data is stored permanently.

Examples:
student.txt, marks.dat, result.bin

2. Why File Handling is Needed?

Without files:

- Data disappears when program stops.
- You must enter values every time.

With file handling:

- You can **store data** permanently.
- You can **read/write large amounts** of data.
- You can maintain **records**, like students, employees, invoices, etc.

3. Types of Files in C

C mainly uses two kinds of files:

3.1 Text Files

- Human readable (open in Notepad, you can see the contents)
- Data is stored as characters (ASCII)
- Example: "123" is stored as '1', '2', '3'

Common extension: .txt

3.2 Binary Files

- Data is stored in **binary form** (same as in memory)
- Not human readable
- Faster and more compact
- Better for storing **structures**, large data, etc.

Common extension: .bin, .dat

4. File Handling in C – The FILE Pointer

In C, every file is handled using a **FILE pointer**.

Before using a file, you declare:

FILE *fp;

Here:

- FILE is a structure defined in <stdio.h>
- fp is a pointer to a FILE (file handle)

Every operation on a file uses such a pointer.

5. Opening a File – fopen()

To access a file, you must open it using fopen().

Syntax:

FILE *fp;

fp = fopen("filename", "mode");

- "filename" → name of the file (e.g. "data.txt")
- "mode" → how you want to open it (read, write, append, etc.)

Common File Modes

For text files:

- "r" → open for reading (file must exist)

- "w" → open for writing (creates new / overwrites existing)
- "a" → open for appending (add at end)
- "r+" → read + write (existing file)
- "w+" → read + write (truncate / create new)
- "a+" → read + append

For binary files: (add b)

- "rb", "wb", "ab", "rb+", "wb+", "ab+"

Example:

```
FILE *fp;

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

if (fp == NULL)

{

   printf("File cannot be opened!");

}
```

6. Closing a File – fclose()

After using a file, you must close it to:

- Flush data from buffer to disk
- Free system resources

Syntax:

```
fclose(fp);
```

Always close files after use.

7. Character Input and Output from File

- fgetc() / getc() – read a character from file
- fputc() / putc() – write a character to file

7.1 fputc() – Write Character

```
fputc(character, fp);
```

Example:

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

fputc('A', fp);

fputc('B', fp);

fclose(fp);
```

This writes AB into test.txt.

7.2 fgetc() – Read Character

```
int ch = fgetc(fp);
```

Example:

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

int ch;

ch = fgetc(fp);

printf("%c", ch);  // prints first character

fclose(fp);
```

7.3 Reading Entire File – Example

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

int ch;

if (fp == NULL) {

    printf("File not found");

} else {

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

```
        printf("%c", ch);
    }
    fclose(fp);
}
```

This reads character by character until **EOF**.

8. End Of File (EOF) and feof()

EOF (End Of File)

- Special condition that indicates that **there is no more data** to read from file.
- Many reading functions (fgetc, fscanf, fread) return EOF (or special value) when end is reached.

Typical loop:

```
while ((ch = fgetc(fp)) != EOF) {
    // process character
}
```

feof() Function

feof(fp) returns **non-zero** (true) if end of file condition is set for that file.

Example:

```
while (!feof(fp)) {
    ch = fgetc(fp);
    ...
}
```

But in practice, using (ch = fgetc(fp)) != EOF is safer.

9. Working with Text Files (Line and Formatted I/O)

Apart from character I/O, we can also use:

- fprintf() → write formatted data to file
- fscanf() → read formatted data from file
- fgets() → read string/line from file
- fputs() → write string to file

Example with fprintf / fscanf:

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

int roll = 1;

float marks = 90.5;

fprintf(fp, "%d %.2f", roll, marks);

fclose(fp);
```

Reading:

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

int roll;

float marks;

fscanf(fp, "%d %f", &roll, &marks);

fclose(fp);
```

10. Working with Binary Files

For binary files, you usually use:

- fwrite() – to write binary blocks
- fread() – to read binary blocks

Syntax:

```
fwrite(&data, sizeof(data_type), count, fp);

fread(&data, sizeof(data_type), count, fp);
```

Example (writing a single int):

```
int x = 10;

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

fwrite(&x, sizeof(int), 1, fp);

fclose(fp);
```

Reading:

```
int x;

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

fread(&x, sizeof(int), 1, fp);

fclose(fp);
```

11. Files of Records (Very Important)

In C, “record” often means a **structure**.

Example:

```
struct student {

    int roll;

    char name[20];

    float marks;

};
```

You can store an array of struct student into a file (text or binary).

11.1 Records in Text File

Using fprintf and fscanf:

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

struct student s = {1, "Atul", 90.5};
```

```c
fprintf(fp, "%d %s %f\n", s.roll, s.name, s.marks);

fclose(fp);
```

Reading:

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

struct student s;

while (fscanf(fp, "%d %s %f", &s.roll, s.name, &s.marks) == 3) {

    // process record

}
```

11.2 Records in Binary File

Using fwrite and fread:

```c
struct student s = {1, "Atul", 90.5};

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

fwrite(&s, sizeof(struct student), 1, fp);

fclose(fp);
```

Reading:

```c
struct student s;

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

while (fread(&s, sizeof(struct student), 1, fp) == 1) {

    // use s

}

fclose(fp);
```

Binary files are more efficient and accurate for record handling.

12. Random Access to Files (Random Access to Records)

So far, we read files **sequentially** (from start to end).
Sometimes, we want to jump directly to a specific record number.

For random access, we use:

- fseek() – move file pointer
- ftell() – get current position
- rewind() – go back to beginning

12.1 fseek()

fseek(fp, offset, origin);

- offset → how many bytes to move
- origin → starting point:
  - SEEK_SET → beginning
  - SEEK_CUR → current position
  - SEEK_END → end of file

Example: move to nth record in binary file of fixed-size structures:

// nth record (0-based):

fseek(fp, n * sizeof(struct student), SEEK_SET);

fread(&s, sizeof(struct student), 1, fp);

This directly jumps to that record.

12.2 ftell()

Returns current position (in bytes) of file pointer from start of file.

long pos = ftell(fp);

12.3 rewind()

Moves file pointer back to start.

rewind(fp);