Ultimate Notes

Ultimate C Programing Notes

BASICS AND FOUNDATIONS OF C

---

1. Introduction to C Language

1.1 What is C Language? (Overview)

C is a **general-purpose, procedural programming language** developed by **Dennis Ritchie** in **1972** at **Bell Laboratories**.

It is called a procedural language because programs are written as a sequence of steps and functions.

This definition is very important.
In theory exams, they expect words like
general-purpose, procedural, developed by Dennis Ritchie.

1.2 Why C is Important for Computer Science Students

C is not just another language. It is considered the **base language** for computer science.

Why?

- Many modern languages like C++, Java are influenced by C
- It helps students understand **memory management**
- Core subjects like OS, Compiler Design use C concepts
- Logic building becomes very strong

If you understand C properly, learning other languages becomes easier. This point is often written in long answers.

1.3 Features of C Language

(Very common theory question: “Explain features of C language”)

Below are **standard, exam-accepted features**. Students can write any 5 to get full marks.

1. **Simple Language**
   C has a small set of keywords and simple syntax.

2. **Structured Programming Language**
   Programs are divided into functions and logical blocks.

3. **Portable Language**
   A C program can run on different machines with little or no modification.

4. **Efficient and Fast**
   C programs execute faster because they are close to hardware.

5. **Rich Library Support**
   Many built-in functions are available in standard libraries.

6. **Low-Level Memory Access**
   C supports pointers, which allow direct memory manipulation.

7. **Extensible**
   New functions can be added easily.

1.4 Applications of C Language

C is widely used in real-world systems.

Common applications asked in exams:

- Operating systems
- Embedded systems
- Compilers
- System software
- Device drivers

1.5 Advantages and Limitations of C

(Asked in long answer form sometimes)

Advantages

- Fast execution
- Better control over hardware
- Portable
- Suitable for system-level programming

Limitations

- No built-in object-oriented support
- No automatic memory management
- No built-in exception handling

---

2. Structure of a C Program

2.1 Concept Overview

Every C program must follow a **specific structure**.
If this structure is followed correctly, most compilation errors are avoided.

Students lose marks in practical exams mainly because they miss steps in structure.

2.2 Memory Hack for Structure

(This is one you already use and it's genuinely effective)

HMVIPO

- **H** Header files
- **M** main() function
- **V** Variable declaration
- **I** Input statements
- **P** Processing or logic
- **O** Output statements

This is not just memory based, it is **writing-flow based**.

2.3 Explanation of Each Component

H – Header Files

Header files contain **predefined functions**.

Example:

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

`stdio.h` provides functions like `printf()` and `scanf()`.

M – main() Function

Execution of every C program **starts from main()**.

Rules:

- Every C program must have exactly one `main()`
- It returns an integer value

Example:

```
int main() {
}
```

V – Variable Declaration

Variables are declared to store data.

Example:

```
int a, b, sum;
```

This allocates memory for variables.

I – Input Statements

Used to take input from the user.

Example:

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

P – Processing / Logic

This is where actual computation happens.

Example:

```
sum = a + b;
```

O – Output Statements

Used to display result.

Example:

```
printf("%d", sum);
```

2.4 Complete Example Program (Structure Demonstration)

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

int main() {   //M
    int a, b, sum;          // V
    scanf("%d %d", &a, &b); // I
    sum = a + b;            // P
    printf("%d", sum);      // O
    return 0;
}
```

In viva, students should explain this using **HMVIPO**.

---

3. Compilation Process of a C Program

3.1 Concept Overview

A C program does not run directly.
It goes through multiple steps before execution.

This is often asked as:

- “Explain compilation process”
- “Explain phases of C program execution”

3.2 Steps of Compilation Process

1. **Source Code (.c file)**
   Program written by programmer.

2. **Preprocessor**

   - Handles `#include`, `#define`
   - Removes comments

3. **Compiler**

   - Converts C code into assembly code
   - Checks syntax errors

4. **Assembler**

   - Converts assembly code into object code

5. **Linker**

   - Links library files
   - Generates executable file

Memory Support Trick

PCAL
Preprocessor → Compiler → Assembler → Linker

Very helpful in theory answers.

---

4. Tokens in C Language

4.1 Concept Overview

Tokens are the **smallest meaningful units** of a C program.

The compiler breaks a program into tokens to understand it.

4.2 Types of Tokens

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators
6. Special symbols

Memory Recall Aid

TKICOS
Tokens = Keywords, Identifiers, Constants, Operators, Strings

---

Example

```
int total = 10;
```

Tokens here:

- int → keyword
- total → identifier
- = → operator
- 10 → constant
- ; → special symbol

---

5. Keywords and Identifiers

5.1 Keywords

Keywords are **reserved words** with predefined meaning in C.

Examples:
`int`, `if`, `else`, `while`, `return`

Rules:

- Cannot be used as variable names
- Meaning cannot be changed

5.2 Identifiers

Identifiers are **user-defined names** given to variables, functions, arrays, etc.

5.3 Rules for Identifiers

(Frequently asked)

- Must start with a letter or underscore
- Can contain letters, digits, underscore
- No spaces allowed
- Cannot be a keyword

Example

```
int marks;    // valid
int 2marks;  // invalid
```

---

6. Constants

6.1 Concept Overview

Constants are values that **do not change** during program execution.

6.2 Types of Constants

1. Integer constants
   Example: 10, -5

2. Floating constants
   Example: 3.14, 2.5

3. Character constants
   Example: 'A', '9'

4. String constants
   Example: "Hello"

6.3 Defining Constants

Using const keyword

```
const int MAX = 100;
```

Using #define

```
#define PI 3.14
```

---

7. Data Types

7.1 Concept Overview

Data types specify:

- Type of data
- Amount of memory allocated

7.2 Classification of Data Types

Basic Data Types

- int
- float
- char
- double

Derived Data Types

- array
- pointer
- structure
- union

Enumeration

- enum

void Data Type

Type Modifiers

- short
- long
- signed

- unsigned

Memory Support Aid

BDMTV
Basic, Derived, Modifiers, Type void

---

8. Variables

8.1 Definition

A variable is a **named memory location** whose value can change during execution.

8.2 Types of Variables

Local Variable

Declared inside a function. Accessible only within that function.

Global Variable

Declared outside all functions. Accessible throughout the program.

8.3 Scope and Lifetime

(Theory favorite)

- Scope defines where a variable can be used
- Lifetime defines how long it exists in memory

---

9. Input and Output Statements

9.1 Formatted Input/Output

- `printf()` for output
- `scanf()` for input

9.2 Unformatted Input/Output

- getchar()
- putchar()
- gets()
- puts()

9.3 Format Specifiers

Data Type	Specifier
int	%d
float	%f
char	%c
string	%s

Example Program

```
#include <stdio.h>

int main() {
    int age;
    char name[20];

    scanf("%d", &age);
```

```
    scanf("%s", name);

    printf("%d %s", age, name);
    return 0;
}
```

OPERATORS IN C

---

1. What is an Operator? (Concept Overview)

Definition (Exam-ready)

An **operator** is a symbol that performs a specific operation on one or more operands.

Example:

```
c = a + b;
```

Here
`+` is the operator
`a` and `b` are operands

This definition is asked very frequently, sometimes directly as
“Define operator and operand.”

---

2. Why Operators are Important

Operators decide:

- how calculations are done
- how conditions are checked
- how logic flows in program

- how values are assigned and updated

Very important point for students:
👉 **Many times program is syntactically correct, but output is wrong only because operator is wrong.**

That's why this chapter matters a lot.

---

3. Classification of Operators in C

(Direct theory question)

According to standard BCA syllabus, C operators are divided into:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Unary Operators
5. Assignment Operators
6. Conditional Operator
7. Bitwise Operators
8. sizeof Operator

---

4. Core Memory Hack for Operators

(This is genuinely useful and exam-proven)

ARLUA

- **A** Arithmetic

- **R** Relational
- **L** Logical
- **U** Unary
- **A** Assignment

If students remember ARLUA, they never miss operator types in theory answers.

We'll now cover each one properly.

---

5. Arithmetic Operators

5.1 Concept Overview

Arithmetic operators are used to perform **mathematical calculations**.

Operators List

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

All five must be written in exams. Missing % costs marks.

5.2 Important Logic Points (Very Important)

1. / gives **quotient**
2. % gives **remainder**

3. `%` works only with integers
4. Division between two integers gives integer result

Example:

```
5 / 2 = 2
5 % 2 = 1
```

This confusion appears a lot in practical exams.

5.3 Example Program: Arithmetic Operators

```c
#include <stdio.h>

int main() {
    int a = 10, b = 3;

    printf("Addition = %d\n", a + b);
    printf("Subtraction = %d\n", a - b);
    printf("Multiplication = %d\n", a * b);
    printf("Division = %d\n", a / b);
    printf("Modulus = %d\n", a % b);

    return 0;
}
```

---

6. Relational Operators

6.1 Concept Overview

Relational operators are used to **compare two values**.

They always return either
`1` (true) or `0` (false)

This is very important for conditions.

6.2 Operators List

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal to
!=	Not equal to

Important note for students:
👉 `==` is comparison, `=` is assignment.
Mixing these two is a very common mistake.

6.3 Example Program: Relational Operators

```
#include <stdio.h>

int main() {
    int a = 5, b = 8;

    printf("%d\n", a < b);
    printf("%d\n", a > b);
    printf("%d\n", a == b);
    printf("%d\n", a != b);

    return 0;
}
```

Output will be 1 or 0.

---

7. Logical Operators

7.1 Concept Overview

Logical operators are used to **combine multiple conditions**.

They are mostly used with `if`, `while`, and `for`.

7.2 Operators List

Operator	Meaning
&&	Logical AND
!	Logical NOT

7.3 Logic Explanation (Very Important)

- `&&` returns true only if **both conditions are true**
- `||` returns true if **any one condition is true**
- `!` reverses the result

Truth table understanding is important here.

7.4 Example Program: Logical Operators

```
#include <stdio.h>

int main() {
    int age = 20;

    if(age >= 18 && age <= 25)
        printf("Eligible\n");
    else
```

```
        printf("Not Eligible\n");

    return 0;
}
```

---

8. Unary Operators

8.1 Concept Overview

Unary operators operate on **single operand**.

8.2 Operators List

Operator	Meaning
++	Increment
--	Decrement

8.3 Pre-increment vs Post-increment

(Very common exam question)

```
++a  // pre-increment: increment first, then use
a++  // post-increment: use first, then increment
```

8.4 Example Program: Unary Operators

```
#include <stdio.h>

int main() {
    int a = 5;

    printf("%d\n", ++a);  // 6
    printf("%d\n", a++);  // 6
```

```
    printf("%d\n", a);    // 7

    return 0;
}
```

Students should be able to explain this step by step in viva.

---

9. Assignment Operators

9.1 Concept Overview

Assignment operators are used to **assign or update values** in variables.

9.2 Operators List

Operator	Meaning
=	Assign
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

9.3 Example Program: Assignment Operators

```
#include <stdio.h>

int main() {
    int a = 10;

    a += 5;
```

```
    printf("%d\n", a);

    a -= 3;
    printf("%d\n", a);

    return 0;
}
```

---

10. Conditional Operator (Ternary Operator)

10.1 Concept Overview

Conditional operator is a **short form of if-else**.

Syntax

```
condition ? expression1 : expression2;
```

If condition is true, expression1 executes
Else, expression2 executes

10.2 Example Program

```
#include <stdio.h>

int main() {
    int a = 10, b = 20;
    int max;

    max = (a > b) ? a : b;
    printf("Max = %d", max);

    return 0;
}
```

This operator is often asked as
“Explain conditional operator with example.”

---

11. Bitwise Operators

11.1 Concept Overview

Bitwise operators work on **binary representation** of data.

They are mostly asked in theory, sometimes in practical.

11.2 Operators List

Operator	Meaning
&	Bitwise AND
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift

11.3 Important Logic Points

- `<<` shifts bits left, effectively multiplies by 2
- `>>` shifts bits right, effectively divides by 2

11.4 Example Program

```
#include <stdio.h>

int main() {
```

```
    int a = 5;   // 0101

    printf("%d\n", a << 1); // 10
    printf("%d\n", a >> 1); // 2

    return 0;
}
```

---

12. sizeof Operator

12.1 Concept Overview

`sizeof` operator returns the **size of data type or variable** in bytes.

12.2 Example Program

```
#include <stdio.h>

int main() {
    int a;
    printf("%lu", sizeof(a));
    return 0;
}
```

---

13. Operator Precedence and Associativity

(Very important theory topic)

Concept Overview

Operator precedence decides **which operator executes first** in an expression.

Basic Order (Exam-friendly)

1. Unary operators
2. Arithmetic (*, /, %)
3. Arithmetic (+, -)
4. Relational
5. Logical
6. Assignment

If unsure, always use parentheses.

Example

```
int x = 5 + 2 * 3;     // result = 11
int y = (5 + 2) * 3;   // result = 21
```

DECISION MAKING STATEMENTS IN C

---

1. What are Decision Making Statements?

Concept Overview

Decision making statements allow a program to **choose one path out of multiple paths** based on a condition.

In simple words,
they help the program **decide what to do next**.

This topic is very important because:

- almost every real program uses conditions
- most practical questions depend on if or switch
- wrong condition means correct logic but wrong output

Definition (Theory Exam Ready)

Decision making statements are used to **control the flow of execution** of a program based on **conditions**.

Keywords examiners expect:

- control flow
- condition
- true or false

---

2. How Conditions Work in C (Very Important Logic)

In C:

- Any **non-zero value** is treated as **true**
- **Zero (0)** is treated as **false**

Example:

```
if(5)     // true
if(0)     // false
```

Relational and logical operators are usually used inside conditions.

---

3. Types of Decision Making Statements in C

According to syllabus, C provides:

1. if statement
2. if-else statement
3. else-if ladder
4. nested if
5. switch case (already covered separately, but logic linked)

We'll cover the first four here in full depth.

---

4. if Statement

4.1 Concept Overview

The `if` statement executes a block of code **only when the condition is true**.

If the condition is false, nothing happens.

4.2 Syntax

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

4.3 Logic Explanation (How to Think While Writing)

1. Condition is checked
2. If condition is true
   → code inside if block executes

3. If condition is false
   → control skips the block

There is **no else part** here.

4.4 Example Program: Check Positive Number

```
#include <stdio.h>

int main() {
    int num;
    scanf("%d", &num);

    if(num > 0) {
        printf("Positive number");
    }

    return 0;
}
```

If input is negative, nothing prints.
That's expected behavior of `if`.

Common Mistake

Students expect output even when condition is false.
That's not how `if` works.

---

5. if-else Statement

5.1 Concept Overview

The `if-else` statement provides **two paths**:

- one for true condition
- one for false condition

Exactly one block executes.

5.2 Syntax

```
if(condition) {
    statements;
}
else {
    statements;
}
```

5.3 Logic Explanation

1. Condition is checked
2. If true → if block executes
3. If false → else block executes

No case is skipped here.

5.4 Example Program: Even or Odd

```
#include <stdio.h>

int main() {
    int num;
    scanf("%d", &num);

    if(num % 2 == 0)
        printf("Even");
    else
```

```
        printf("Odd");

    return 0;
}
```

This is one of the **most common practical exam programs**.

Memory Support Tip

Whenever there are **exactly two outcomes**, think **if-else**.

---

6. else-if Ladder

6.1 Concept Overview

The else-if ladder is used when there are **multiple conditions**, and **only one condition should execute**.

Conditions are checked **from top to bottom**.

6.2 Syntax

```
if(condition1) {
    statements;
}
else if(condition2) {
    statements;
}
else if(condition3) {
    statements;
}
else {
    statements;
}
```

6.3 Core Logic (Very Important)

- Conditions are checked **one by one**
- The moment a condition becomes true
  → corresponding block executes
- Remaining conditions are skipped
- else block runs only if **all conditions are false**

Memory Hack (This one actually helps)

TDS
Top
Down
Scan

Think of it like scanning a list from top to bottom.

6.4 Example Program: Grade System

```c
#include <stdio.h>

int main() {
    int marks;
    scanf("%d", &marks);

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

    return 0;
```

```
}
```

This exact logic is used in exams, just numbers change.

Common Mistake

Wrong order of conditions.

Example wrong logic:

```
if(marks >= 60)
else if(marks >= 90)
```

This will never reach 90 condition.
Order always matters.

---

7. Nested if Statement

7.1 Concept Overview

A nested if means an **if inside another if**.

Used when:

- a condition depends on another condition
- multiple checks are required step by step

7.2 Syntax

```
if(condition1) {
    if(condition2) {
        statements;
    }
}
```

7.3 Logic Explanation

1. First condition is checked
2. Only if it is true, inner if is checked
3. Inner condition decides final execution

Think of it like **permission levels**.

7.4 Example Program: Largest of Three Numbers

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    if(a > b) {
        if(a > c)
            printf("a is largest");
        else
            printf("c is largest");
    }
    else {
        if(b > c)
            printf("b is largest");
        else
            printf("c is largest");
    }

    return 0;
}
```

LOOPING STATEMENTS IN C

---

1. What are Loops?

Concept Overview

Loops are used to **execute a block of code repeatedly** until a condition becomes false.

In simple terms,
loops save us from writing the same code again and again.

Example idea
Print numbers from 1 to 100
Without loop → impossible practically
With loop → very easy

Definition (Exam Ready)

A loop is a control structure that allows a set of statements to be executed **repeatedly** based on a condition.

Keywords examiners like:

- repetition
- condition
- execution control

---

2. Why Loops are Extremely Important

Loops are the backbone of:

- arrays
- strings

- patterns
- searching
- file handling

Almost **70 percent practical programs** depend on loops directly or indirectly.

If loops are weak, C becomes scary.
If loops are clear, C becomes comfortable.

---

3. Types of Loops in C (Syllabus Based)

C provides **three looping statements**:

1. for loop
2. while loop
3. do-while loop

All three must be remembered clearly, including **when to use which**.

---

4. Core Memory Hack for Loops

(This one is genuinely useful in exams)

KUM

- **K** Known number of iterations
- **U** Unknown number of iterations
- **M** Must execute at least once

This single rule helps students choose the correct loop instantly.

---

5. for Loop

5.1 Concept Overview

The `for` loop is used when the **number of repetitions is known in advance**.

Typical use cases:

- printing numbers 1 to 10
- traversing arrays
- fixed-count loops

5.2 Syntax

```
for(initialization; condition; increment/decrement) {
    statements;
}
```

5.3 Logic Explanation (Very Important)

Execution flow of for loop:

1. Initialization executes **once**
2. Condition is checked
3. If condition is true
   → loop body executes
4. Increment or decrement happens
5. Control goes back to condition
6. Loop stops when condition becomes false

Students should explain this flow in viva.

5.4 Example Program: Print 1 to 5

```c
#include <stdio.h>

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int i=1; //s
    while(i <= 5) { //e
        printf("%d ", i);
        i++; //i
    }
    return 0;
}
```

Common Mistakes

- Using `i < 5` instead of `i <= 5`
- Forgetting increment part
- Writing semicolon after for loop

---

6. while Loop

6.1 Concept Overview

The `while` loop is used when the **number of iterations is not known beforehand**.

The loop runs **as long as condition is true**.

6.2 Syntax

```
while(condition) {
    statements;
}
```

6.3 Logic Explanation

1. Condition is checked first
2. If condition is true
   → loop body executes
3. Update happens inside the loop
4. Control goes back to condition

If condition is false initially, loop will **not run even once**.

6.4 Example Program: Reverse a Number

(Very common exam question)

```
#include <stdio.h>

int main() {
    int n, rev = 0;
    scanf("%d", &n);
```

```
    while(n != 0) {
        rev = rev * 10 + (n % 10);
        n = n / 10;
    }

    printf("%d", rev);
    return 0;
}
```

Why while is perfect here

We don't know how many digits the number has.
That's why **while**, not for.

---

7. do-while Loop

7.1 Concept Overview

The `do-while` loop executes the loop body **at least once**, even if the condition is false.

This is the key difference.

7.2 Syntax

```
do {
    statements;
} while(condition);
```

Notice the semicolon at the end.
Missing it causes compilation error.

7.3 Logic Explanation

1. Loop body executes first

2. Condition is checked after execution

3. If condition is true
   → loop repeats

4. If false
   → loop stops

7.4 Example Program: Menu Driven Program (Exam Favorite)

```
#include <stdio.h>

int main() {
    int choice;
    do {
        printf("1. Hello\n2. Exit\n");
        scanf("%d", &choice);

        if(choice == 1)
            printf("Hello User\n");

    } while(choice != 2);

    return 0;
}
```

This program **must run once**, so do-while is correct.

---

8. Comparison of for, while, do-while

(Very common theory question)

	for	while	do-while

Known iterations	Unknown iterations	Must execute once
Condition checked first	Condition checked first	Condition checked last
Compact syntax	Simple syntax	Ends with semicolon

---

9. Nested Loops

9.1 Concept Overview

A nested loop is a **loop inside another loop**.

Mostly used for:

- pattern printing
- 2D arrays
- matrices

9.2 Core Logic (Very Important)

- Outer loop controls **rows**
- Inner loop controls **columns**

Memory Hack

ORIC

Outer → Rows
Inner → Columns

This rule is extremely helpful in patterns.

9.3 Example Program: Star Pattern

```
#include <stdio.h>

int main() {
    int i, j;
    for(i = 1; i <= 4; i++) {
        for(j = 1; j <= i; j++) {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

---

10. Infinite Loops

Concept Overview

A loop that never ends is called an infinite loop.

Examples

```
while(1) {
    printf("Hello");
}
```

or

```
for(;;) {
    printf("Hello");
}
```

Used rarely, but sometimes in servers or embedded systems.

---

11. Loop Control Statements (Linking Concept)

Loops are often used with:

- break
- continue

Example:

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

Output
1 2 4 5

JUMP STATEMENTS IN C

---

1. What are Jump Statements?

Concept Overview

Jump statements are used to **transfer control immediately** from one part of the program to another.

Unlike loops and decision statements that follow normal top-to-bottom flow, jump statements **change the flow suddenly**.

That's why they are called *jump* statements.

Definition (Theory Exam Ready)

Jump statements are statements that **alter the normal sequential execution** of a program by transferring control to another statement.

Key words examiners expect:

- transfer control
- execution flow
- immediate jump

---

2. Types of Jump Statements in C

According to syllabus, C has **four jump statements**:

1. break
2. continue
3. goto
4. return

All four are asked in theory.
break, continue, return are common in practicals.

Memory Support Trick

BCRG
Break
Continue
Return
Goto

This helps students list all jump statements without missing one.

---

3. break Statement

3.1 Concept Overview

The `break` statement is used to **terminate a loop or switch statement immediately**.

Control jumps **outside** the loop or switch.

3.2 Where break is Used

- inside loops (for, while, do-while)
- inside switch case

It does **not work** outside these.

3.3 Logic Explanation

When `break` is encountered:

1. Loop or switch stops instantly
2. Control moves to the next statement after the loop or switch

Remaining iterations are skipped.

3.4 Example Program: break in Loop

```
#include <stdio.h>

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        if(i == 3)
            break;
        printf("%d ", i);
```

```
    }
    return 0;
}
```

Output

1 2

Once `i == 3`, loop stops completely.

Example Program: break in switch

```
switch(choice) {
    case 1: printf("One"); break;
    case 2: printf("Two"); break;
    default: printf("Invalid");
}
```

Without break, fall-through happens.
That's why break is important in switch.

Common Mistake

Students forget break in switch and get wrong output.
Very common practical mistake.

---

4. continue Statement

4.1 Concept Overview

The `continue` statement is used to **skip the current iteration** of a loop and move to the **next iteration**.

Loop does not stop, only current cycle is skipped.

4.2 Where continue is Used

- only inside loops
- not used in switch

4.3 Logic Explanation

When `continue` is encountered:

1. Remaining statements in loop body are skipped
2. Loop condition is checked again
3. Next iteration starts

4.4 Example Program: continue in Loop

```
#include <stdio.h>

int main() {
    int i;
    for(i = 1; i <= 5; i++) {
        if(i == 3)
            continue;
        printf("%d ", i);
    }
    return 0;
}
```

Output

1 2 4 5

Here only 3 is skipped, loop continues.

Difference Between break and continue

(Very common theory question)

break	continue
Exits loop completely	Skips current iteration
Loop ends	Loop continues
Used in loop and switch	Used only in loop

---

5. goto Statement

5.1 Concept Overview

The `goto` statement is used to **jump directly to a labeled statement** in the program.

It is generally **not recommended**, but included in syllabus.

5.2 Syntax

```
goto label;
/* statements */
label:
    statement;
```

5.3 Logic Explanation

When `goto` executes:

- control jumps directly to the label
- normal flow is broken

This can make program hard to understand.

5.4 Example Program: goto

```
#include <stdio.h>

int main() {
    int n = 1;
    label:
        printf("%d ", n);
        n++;
        if(n <= 5)
            goto label;
    return 0;
}
```

This prints numbers 1 to 5.

Important Exam Point

- goto is discouraged
- makes program confusing
- should be avoided unless necessary

Write this line in theory answers for full marks.

---

6. return Statement

6.1 Concept Overview

The `return` statement is used to **transfer control back to the calling function**.

It can also **return a value** from a function.

6.2 Where return is Used

- inside functions
- especially important in main()

6.3 Logic Explanation

When `return` is encountered:

1. Function execution stops
2. Control goes back to caller
3. Optional value is returned

6.4 return in main()

```
int main() {
    return 0;
}
```

`return 0` indicates successful program execution.

This is often asked in viva.

6.5 Example Program: return from Function

```
#include <stdio.h>

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

int main() {
    int result = add(3, 4);
    printf("%d", result);
```

```
    return 0;
}
```

---

7. Comparison of All Jump Statements

(Very useful for revision)

Statement	Purpose
break	Exit loop or switch
continue	Skip current loop iteration
goto	Jump to labeled statement
return	Exit function

STRINGS IN C

---

1. What is a String in C?

Concept Overview

In C, a string is **not a separate data type**.
A string is actually an **array of characters** that ends with a **null character** `'\0'`.

This single line is very important.
Many students lose marks because they say “string is a data type”. In C, it is not.

Definition (Theory Exam Ready)

A string is a **sequence of characters stored in a character array**, terminated by a **null character** `'\0'`.

Examiner expects these words:

- character array
- sequence of characters
- null character

---

2. Why `'\0'` is Important (Core Logic)

The null character `'\0'` tells the compiler
👉 **where the string ends**.

Without `'\0'`, C does not know how many characters belong to the string.

Example idea
If string is `"Hello"`
Internally it is stored as
`H e l l o \0`

This concept is extremely important for string functions.

---

3. Declaring a String

3.1 Syntax

```
char string_name[size];
```

3.2 Example

```
char name[10];
```

This means:

- string can store up to 9 characters

- 1 space is reserved for `'\0'`

This is a very common viva question.

---

4. Initializing a String

Method 1: Character by Character

```
char s[6] = {'H','e','l','l','o','\0'};
```

Method 2: String Literal (Most Common)

```
char s[] = "Hello";
```

Here compiler automatically adds `'\0'`.

Important Rule

Always ensure array size is **at least length + 1**.

Wrong:

```
char s[5] = "Hello";   // no space for '\0'
```

---

5. Taking Input and Output of String

5.1 Using scanf and printf

```
char name[20];
scanf("%s", name);
printf("%s", name);
```

Note:

- `&` is not used with string name
- scanf reads only one word, stops at space

5.2 Using gets and puts

(Still in syllabus, but mention carefully)

```
gets(name);
puts(name);
```

Important exam point:

- `gets()` is unsafe
- It does not check array size

Write this in theory answers.

---

6. Difference Between Character Array and String

(Frequently asked theory question)

Character Array	String
Stores characters	Stores characters + `'\0'`
No termination rule	Ends with null character
Treated as data	Treated as text

---

7. String Handling Functions

(Header file: `string.h`)

These functions are very important for both theory and practical exams.

---

8. `strlen()` Function

Concept

Returns the **length of string**, excluding `'\0'`.

Syntax

```
strlen(string);
```

Example Program

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

int main() {
    char s[] = "Hello";
    printf("%d", strlen(s));
    return 0;
}
```

Output: `5`

---

9. `strcpy()` Function

Concept

Copies one string into another.

Syntax

```
strcpy(destination, source);
```

Destination must be large enough.

Example Program

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

int main() {
    char s1[20], s2[] = "C Programming";
    strcpy(s1, s2);
    printf("%s", s1);
    return 0;
}
```

---

10. `strcat()` Function

Concept

Appends one string at the end of another string.

Syntax

```
strcat(destination, source);
```

Example Program

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

int main() {
    char s1[20] = "Hello";
    char s2[] = "World";
    strcat(s1, s2);
    printf("%s", s1);
    return 0;
}
```

Output: `HelloWorld`

---

11. `strcmp()` Function

Concept

Compares two strings.

Return values:

- `0` if strings are equal
- negative value if first < second
- positive value if first > second

Example Program

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

int main() {
    char s1[] = "abc";
    char s2[] = "abc";

    if(strcmp(s1, s2) == 0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");

    return 0;
}
```

---

12. Core Memory Support for String Functions

LSCC

- **L** strlen
- **S** strcpy
- **C** strcat
- **C** strcmp

This helps students remember main string functions in exams.

---

13. Common String Programs (Exam Focused)

These are repeatedly asked in practical exams.

13.1 Reverse a String (Without Library Function)

Logic

- Find length
- Swap characters from start and end

Example Program

```
#include <stdio.h>

int main() {
    char s[20];
    int i, len = 0;

    scanf("%s", s);
```

```
    while(s[len] != '\0')
        len++;

    for(i = len - 1; i >= 0; i--)
        printf("%c", s[i]);

    return 0;
}
```

---

14. Passing String to Function

Concept Overview

String name acts as base address, so string is passed **by reference**.

Example Program

```
#include <stdio.h>

void display(char s[]) {
    printf("%s", s);
}

int main() {
    char name[] = "Avani";
    display(name);
    return 0;
}
```

FUNCTIONS IN C

---

1. What is a Function?

Concept Overview

A function is a **self contained block of code** that performs a **specific task**.

Instead of writing the same logic again and again, we write it once inside a function and **reuse it**.

This is the real purpose of functions.

Definition (Exam Ready)

A function is a **group of statements** that performs a specific task and can be **called multiple times** in a program.

Words examiners expect:

- group of statements
- specific task
- reusability

---

2. Why Functions are Needed (Logic Understanding)

Without functions:

- program becomes very long
- logic becomes confusing
- debugging becomes difficult

With functions:

- program is modular
- logic is easy to understand
- reuse becomes possible

Very common theory line:
👉 Functions improve **modularity and readability** of programs.

---

3. Parts of a Function

(Very important theory question)

Every function involves **three steps**.

Memory Hack (Very useful and exam-proven)

DDC

- **D** Declaration
- **D** Definition
- **C** Call

If students remember DDC, they never miss any part in exams.

---

4. Function Declaration (Prototype)

Concept

Function declaration tells the compiler:

- function name

- return type
- number and type of arguments

It is written **before main()**.

Syntax

```
return_type function_name(parameter_list);
```

Example

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

This means:

- function returns int
- takes two integer arguments

---

5. Function Definition

Concept

Function definition contains the **actual logic** of the function.

Syntax

```
return_type function_name(parameters) {
    statements;
}
```

Example

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

---

6. Function Call

Concept

Function call transfers control from `main()` to the function.

After execution, control returns back.

Example

```
sum = add(3, 4);
```

---

7. Complete Example Showing DDC

```
#include <stdio.h>

int add(int, int);        // Declaration

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

int main() {
    int sum;
    sum = add(3, 4);     // Call
    printf("%d", sum);
    return 0;
}
```

This example is **perfect for practical exams**.

---

8. Types of Functions in C

According to syllabus, functions are classified into:

1. Library functions
2. User-defined functions

---

9. Library Functions

Concept

Library functions are **predefined functions** provided by C.

Examples:

- printf(), scanf()
- strlen(), strcpy()
- sqrt(), pow()

Header files are required to use them.

---

10. User Defined Functions

Concept

Functions written by the programmer to perform specific tasks.

Most practical exam programs use **user-defined functions**.

---

11. Classification of User Defined Functions

(Very common theory question)

Based on arguments and return value, there are **four types**.

Memory Hack (Simple and effective)

Think in two questions:

- Do I give input to function?
- Do I get output from function?

Type 1: No Arguments, No Return Value

```
void show() {
    printf("Hello");
}
```

Called as:

```
show();
```

Type 2: Arguments, No Return Value

```
void showSum(int a, int b) {
    printf("%d", a + b);
}
```

Type 3: No Arguments, Return Value

```
int getNumber() {
    return 10;
}
```

Type 4: Arguments and Return Value

(Most commonly used)

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

---

12. Passing Arguments to Functions

Concept Overview

Arguments are values passed from `main()` to function.

Two methods:

1. Call by Value
2. Call by Reference (via pointers)

We'll focus on call by value here. Reference will be detailed in pointers.

---

13. Call by Value

Logic

- Copy of value is passed
- Original variable does not change

Example Program

```
#include <stdio.h>

void change(int x) {
```

```
    x = 20;
}

int main() {
    int a = 10;
    change(a);
    printf("%d", a);
    return 0;
}
```

Output: `10`
Because only copy was changed.

---

14. Return Statement

Concept

`return` sends value back to calling function and ends function execution.

Rules

- Only one value can be returned
- Return type must match function type

Example

```
return a + b;
```

---

15. Scope of Variables in Functions

Concept

Variables declared inside a function are **local to that function**.

They cannot be accessed outside.

Example

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

RECURSION IN C

---

1. What is Recursion?

Concept Overview

Recursion is a technique where a **function calls itself** to solve a problem.

Instead of solving the whole problem at once, recursion breaks it into **smaller versions of the same problem**.

That's the core idea. Nothing more fancy than that.

Definition (Exam Ready)

Recursion is a process in which a function **calls itself directly or indirectly** to solve a problem.

Important words examiners look for:

- function calls itself
- repeated calls

- smaller subproblem

---

2. Why Recursion is Used

Some problems are naturally recursive, meaning:

- the solution depends on a smaller version of the same problem

Common examples:

- factorial
- Fibonacci series
- tree traversal (later in CS subjects)

In exams, recursion checks whether students understand **logic flow**, not just syntax.

---

3. Two Mandatory Parts of Recursion

(Very important theory question)

Every recursive function must have **two things**.

Memory Hack (very important, don't skip)

CBS

- **C** Call itself
- **B** Base condition
- **S** Stop

If base condition is missing, program goes into **infinite recursion**.

This hack is not for memory only, it's for **writing correct code**.

---

4. How Recursion Works Internally (Logic Understanding)

When a function calls itself:

- each call is stored in **stack memory**
- local variables are created for each call
- when base condition is reached, calls start returning one by one

This is why recursion uses more memory than loops.

Examiners love when students mention **stack** in answers.

---

5. General Structure of Recursive Function

Syntax Pattern

```
return_type function_name(parameters) {
    if(base_condition)
        return value;
    else
        return function_name(modified_parameters);
}
```

Students should remember this structure.

---

6. Example 1: Factorial Using Recursion

(Most asked recursion program)

Mathematical Logic

```
factorial(n) = n * factorial(n-1)
```

```
factorial(1) = 1
```

Code Explanation Before Writing

- Function calls itself with smaller value
- Stops when n becomes 1

Example Program

```c
#include <stdio.h>

int fact(int n) {
    if(n == 1)
        return 1;
    else
        return n * fact(n - 1);
}

int main() {
    int num;
    scanf("%d", &num);
    printf("%d", fact(num));
    return 0;
}
```

How to Explain in Viva

- fact calls itself
- base condition is n == 1
- recursive call reduces n
- stack unwinds after base case

Say this calmly, marks guaranteed.

---

7. Example 2: Fibonacci Series Using Recursion

Fibonacci Logic

```
fib(n) = fib(n-1) + fib(n-2)
fib(0) = 0
fib(1) = 1
```

Example Program

```
#include <stdio.h>

int fib(int n) {
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d", fib(n));
    return 0;
}
```

---

8. Direct vs Indirect Recursion

Direct Recursion

Function calls itself directly.

```
void fun() {
    fun();
}
```

Indirect Recursion

Function calls another function, which calls the first function again.

```
void fun1() {
    fun2();
}
void fun2() {
    fun1();
}
```

This is usually asked as a short theory question.

---

9. Recursion vs Loop

(Very common comparison question)

Recursion	Loop
Function calls itself	Repetition using control structure
Uses stack memory	Uses less memory
Code shorter	Code faster
Slower execution	Faster execution

Conclusion line for exams:
👉 Loops are preferred when performance matters.

Simple Program: Sum of Digits Using Recursion

```c
#include <stdio.h>

int sum(int n) {
    if(n == 0)
        return 0;
    else
        return (n % 10) + sum(n / 10);
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d", sum(n));
    return 0;
}
```

Final Recursion Writing Strategy

Before writing recursion code:

- write base condition first
- write recursive call after that
- check if argument moves toward base case

Follow **CBS rule** and recursion will never go wrong.

STORAGE CLASSES IN C

1. What are Storage Classes?

Concept Overview

Storage classes in C define **four important things** about a variable:

1. **Scope** – where the variable can be used
2. **Lifetime** – how long the variable exists
3. **Storage location** – memory or register
4. **Initial value** – default value if not assigned

This chapter explains **how variables behave**, not just how they are written.

Definition (Exam Ready)

Storage classes are used to define the **scope, lifetime, storage location, and initial value** of variables in C.

This exact line is very useful in theory answers.

---

2. Types of Storage Classes in C

According to syllabus, C provides **four storage classes**:

1. auto
2. static
3. extern

4. register

All four must be explained in exams.

Memory Support Trick

ASER

- **A** auto
- **S** static
- **E** extern
- **R** register

This helps students list all storage classes without missing any.

---

3. auto Storage Class

3.1 Concept Overview

`auto` is the **default storage class** for local variables.

If you don’t specify any storage class, the variable is automatically auto.

3.2 Key Characteristics

- Scope: local to block or function
- Lifetime: till block execution
- Default value: garbage value
- Storage: memory (stack)

3.3 Important Exam Point

Writing `auto` is optional.
These two are same:

```
int x;

auto int x;
```

3.4 Example Program

```
#include <stdio.h>


int main() {

    auto int a = 10;

    printf("%d", a);

    return 0;

}
```

Most students don't write auto explicitly, but concept is important.

---

4. static Storage Class

4.1 Concept Overview

The `static` storage class is used when we want a variable to **retain its value between function calls**.

This is the most important storage class practically.

4.2 Key Characteristics

- Scope: local to function
- Lifetime: entire program execution
- Default value: 0
- Storage: memory (data segment)

4.3 Core Logic (Very Important)

A static variable:

- is created only once
- value is preserved
- not destroyed when function ends

This is often tested in viva.

4.4 Example Program (Very Important)

```
#include <stdio.h>


void count() {
    static int c = 0;
    c++;
    printf("%d\n", c);
```

```
}

int main() {
    count();
    count();
    count();
    return 0;
}
```

Output

```
1
2
3
```

If `c` was not static, output would be 1 every time.

When to Use static

- counting function calls
- preserving value
- memory optimization

---

5. extern Storage Class

5.1 Concept Overview

`extern` is used to **access a global variable declared in another file**.

It tells the compiler that variable exists somewhere else.

5.2 Key Characteristics

- Scope: global
- Lifetime: entire program
- Default value: 0
- Storage: memory

5.3 Important Exam Logic

- `extern` does not allocate memory
- it only refers to an existing variable

This line is important in theory answers.

5.4 Example Program (Single File Explanation)

```
#include <stdio.h>


int x = 10;   // global variable


int main() {

    extern int x;
```

```
    printf("%d", x);
    return 0;
}
```

In real projects, extern is used across multiple files.

---

6. register Storage Class

6.1 Concept Overview

`register` suggests the compiler to store variable in **CPU register** instead of memory.

This makes access faster.

6.2 Key Characteristics

- Scope: local
- Lifetime: block
- Default value: garbage
- Storage: CPU register (if available)

6.3 Important Restrictions

- Address of register variable cannot be accessed
- Compiler may ignore register request

Both points are commonly asked.

6.4 Example Program

```
#include <stdio.h>


int main() {
    register int i;
    for(i = 0; i < 5; i++)
        printf("%d ", i);
    return 0;
}
```

---

7. Comparison of Storage Classes

(Very common theory question)

Storage Class	Scope	Lifetime	Default Value
auto	local	block	garbage
static	local	program	0
extern	global	program	0
register	local	block	garbage

Students should practice reproducing this table.

---

8. Storage Class vs Scope

(Clarification students often confuse)

- Storage class defines **how and where variable is stored**
- Scope defines **where variable can be accessed**

They are related but not same.

9. Practical Writing Strategy

When writing code:

- need value preserved → static
- normal local variable → auto
- variable shared across files → extern
- fast loop counter → register

POINTERS IN C

1. What is a Pointer?

Concept Overview

A pointer is a **variable that stores the address of another variable**, not the value itself.

That's it.
This one line clears most confusion.

Definition (Theory Exam Ready)

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

2. Why Pointers are Needed (Very Important Logic)

Students often ask, why not just use normal variables?

Pointers are needed because:

- they allow **direct memory access**
- they help in **call by reference**
- they are used in **arrays, strings, structures**
- dynamic memory allocation works only with pointers

If pointers are skipped, advanced C is impossible.

---

3. Pointer Basics: Address and Value

Two Important Operators

1. & Address-of operator
2. * Dereference operator

Memory Hack (Simple and very effective)

AV

- **A** Address
- **V** Value

Remember this always.

Example Explanation

```
int a = 10;

int *p = &a;
```

- `a` stores value 10
- `&a` gives address of a
- `p` stores address of a
- `*p` gives value stored at that address

---

4. Declaring a Pointer

Syntax

```
data_type *pointer_name;
```

Example

```
int *p;
```

This means:

- p can store address of an integer variable

Pointer type must match variable type.
This is often asked in viva.

---

5. Complete Basic Pointer Example

```
#include <stdio.h>

int main() {

    int a = 10;

    int *p;


    p = &a;


    printf("Value of a = %d\n", a);

    printf("Address of a = %p\n", &a);

    printf("Value stored in p = %p\n", p);

    printf("Value at address p = %d\n", *p);


    return 0;

}
```

6. Pointer Initialization Rules

Important points:

- Pointer should be initialized before use
- Uninitialized pointer causes garbage address
- Dereferencing garbage pointer causes runtime error

Safe practice:

```
int *p = NULL;
```

Mentioning NULL shows good understanding.

---

7. Pointer Arithmetic

Concept Overview

Pointer arithmetic means performing operations like:

- increment
- decrement
- addition
- subtraction

But pointer arithmetic depends on **data type size**.

Logic Explanation

If:

```
int *p;
```

Then:

- `p + 1` moves by **4 bytes** (on most systems)
- not by 1 byte

This is a very common theory question.

Example Program

```c
#include <stdio.h>


int main() {
    int a = 10;
    int *p = &a;


    printf("%p\n", p);
    printf("%p\n", p + 1);


    return 0;
}
```

Explain that address changes by size of int.

---

8. Pointers and Arrays

(Extremely important topic)

Concept Overview

Array name itself is a **pointer to the first element** of the array.

This is a big turning point in understanding C.

Memory Hack

ABA

Array
 Base
 Address

Array name stores base address.

Example Explanation

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

- `a` is address of `a[0]`
- `a + 1` is address of `a[1]`
- `*(a + i)` is same as `a[i]`

Example Program

```
#include <stdio.h>


int main() {

    int a[3] = {1, 2, 3};

    int *p = a;
```

```
    printf("%d\n", *p);       // 1
    printf("%d\n", *(p + 1)); // 2
    printf("%d\n", a[2]);     // 3

    return 0;
}
```

This is very commonly asked in exams.

---

9. Pointer and Function (Call by Reference)

Concept Overview

By default, C uses **call by value**.
To modify actual variables, we use pointers.

Logic Explanation

- Pass address to function
- Function receives pointer
- Changes reflect in original variable

Example Program: Swap Using Pointers

(Very common practical)

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

```
void swap(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    printf("%d %d", a, b);
    return 0;
}
```

This proves call by reference clearly.

---

10. Pointer to Pointer

Concept Overview

A pointer to pointer stores **address of another pointer**.

Used rarely in basics, but comes in theory.

Declaration

```
int **pp;
```

Example Program

```
#include <stdio.h>


int main() {
    int a = 10;
    int *p = &a;
    int **pp = &p;

    printf("%d\n", **pp);

    return 0;
}
```

Explain it as:

- pp points to p
- p points to a

---

11. Common Mistakes Students Make

- Dereferencing uninitialized pointer
- Confusing `*p` and `p`
- Wrong pointer type
- Using pointer without understanding array relation
- Forgetting & while passing address

Examiners catch these instantly.

---

12. How Pointers Come in Exams

Theory

- Define pointer
- Advantages of pointers
- Pointer arithmetic
- Pointer and array relationship
- Call by value vs call by reference

Practical

- Swap using pointer
- Array traversal using pointer
- Simple pointer demonstration

---

13. Pointer Writing Strategy (Very Important)

While writing pointer code, always ask:

1. What variable's address do I need?
2. Is pointer type matching variable type?
3. Where should I dereference?

STRUCTURES IN C

1. What is a Structure?

Concept Overview

In real life, one entity usually has **different types of information**.

Example
A student has:

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

We cannot store all this using arrays because arrays store **same data type only**.

👉 This is exactly **why structures exist**.

Definition (Theory Exam Ready)

A structure is a **user defined data type** that allows us to store **different data types under a single name**.

Important words examiners expect:

- user defined data type

- different data types
- single name

2. Why Structures are Needed (Logic)

Without structure:

```
int roll;

char name[20];

float marks;
```

Managing multiple students becomes confusing.

With structure:

```
struct student s;
```

Now all related data is grouped together.

👉 Structures help in **organizing data logically**.

---

3. Declaration of Structure

Syntax

```
struct structure_name {

    data_type member1;

    data_type member2;

};
```

Example

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

Important exam rule:
👉 **Structure declaration does not allocate memory**.
It only creates a blueprint.

This line is very important in theory answers.

---

4. Declaring Structure Variables

Syntax

```
struct student s1;
```

Now memory is allocated.

Memory Hack (Very practical and exam-proven)

DNVA

- **D** Define structure

- **N** Name variable
- **V** Values assign
- **A** Access members

If students follow DNVA, structure programs rarely go wrong.

---

5. Accessing Structure Members

Concept Overview

Members of a structure are accessed using the **dot (.) operator**.

Syntax

```
structure_variable.member_name
```

Example

```
s1.roll = 1;

s1.marks = 85.5;
```

---

6. Complete Example: Structure Program (Basic)

```
#include <stdio.h>


struct student {
```

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

int main() {
    struct student s;

    s.roll = 1;
    scanf("%s", s.name);
    s.marks = 90.5;

    printf("%d %s %.2f", s.roll, s.name, s.marks);
    return 0;
}
```

This is a **perfect beginner structure program** for exams.

---

7. Initialization of Structure

Method 1: At Declaration

```
struct student s = {1, "Aman", 88.5};
```

Order must match structure members.

Method 2: Member-wise Initialization

```
s.roll = 2;

strcpy(s.name, "Riya");

s.marks = 92.0;
```

---

8. Array of Structures

(Very important practical topic)

Concept Overview

An array of structures is used when we want to store **multiple records of same type**.

Example:

- list of students
- employee records

Syntax

```
struct student s[5];
```

Logic Explanation

- Each array element is a structure

- Loop is used to access each record
- Dot operator is still used

Example Program: Array of Structures

```
#include <stdio.h>


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


int main() {
    struct student s[2];
    int i;


    for(i = 0; i < 2; i++) {
        scanf("%d %s %f", &s[i].roll, s[i].name, &s[i].marks);
    }


    for(i = 0; i < 2; i++) {
        printf("%d %s %.2f\n", s[i].roll, s[i].name, s[i].marks);
```

```
    }

    return 0;
}
```

This exact type of program appears very often in practical exams.

---

9. Passing Structure to Function

Concept Overview

A structure can be passed to a function:

- by value
- by reference (using pointer)

Passing by reference is more efficient.

Example Program: Passing Structure to Function

```
#include <stdio.h>


struct student {
    int roll;
    float marks;
};
```

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

int main() {
    struct student s1 = {1, 85.5};
    display(s1);
    return 0;
}
```

---

10. Structure Pointer (Important Concept)

Concept Overview

A pointer can point to a structure.

Used heavily in advanced programs.

Syntax

```
struct student *p;
```

Accessing Members Using Pointer

Two ways:

```
(*p).roll
```

```
p->roll
```

Arrow operator `->` is preferred.

Example Program

```
#include <stdio.h>

struct student {
    int roll;
};

int main() {
    struct student s = {1};
    struct student *p = &s;

    printf("%d", p->roll);
    return 0;
}
```

---

11. Difference Between Structure and Array

Structure	Array
Stores different data types	Stores same data type
User defined	Derived data type
Uses dot operator	Uses index

---

12. How Structures Come in Exams

Theory

- Define structure
- Advantages of structure
- Structure vs array
- Array of structures

Practical

- Student record
- Employee record

- Structure with function
- Structure pointer

14. Practical Writing Strategy

Before writing structure program:

1. Decide structure members
2. Define structure first
3. Declare variable or array
4. Use dot or arrow correctly

Follow **DNVA rule** and structure programs will be smooth.

UNIONS IN C

1. What is a Union?

Concept Overview

A union is a **user defined data type**, just like a structure, that allows us to store **different data types under a single name**.

At first glance, this sounds exactly like a structure.
The **real difference** lies in **memory usage**.

That is the heart of unions.

Definition (Theory Exam Ready)

A union is a **user defined data type** in which **all members share the same memory location**, and only **one member can hold a value at a time**.

These words are very important:

- share same memory
- one member at a time

2. Why Unions are Needed (Logic)

Structures allocate **separate memory** for each member.

But sometimes, we need to store **only one value at a time**, not all.

Example idea
A variable can store:

- an integer OR
- a float OR
- a character

But never all three at once.

👉 In such cases, using structure wastes memory.
👉 Union saves memory.

That is why unions exist.

---

3. Declaration of Union

Syntax

```
union union_name {

    data_type member1;

    data_type member2;

};
```

Example

```
union data {

    int i;

    float f;

    char c;

};
```

Important exam point:
👉 **Union declaration does not allocate memory**, just like structure.

---

4. Declaring Union Variables

```
union data d;
```

Now memory is allocated.

---

5. Memory Allocation in Union (Most Important Concept)

Key Rule

Memory allocated to a union is equal to the **size of its largest member**.

Example:

```
union data {

    int i;      // 4 bytes
```

```
    float f;    // 4 bytes
    char c;     // 1 byte
};
```

Memory allocated = **4 bytes**, not 9 bytes.

This question is asked very frequently in theory.

Memory Hack (Simple and Useful)

SM

Shared
Memory

Remember this whenever union is mentioned.

---

6. Accessing Union Members

Union members are accessed using the **dot (.) operator**, same as structure.

```
d.i = 10;
```

But remember:
👉 When you assign value to one member, previous value is lost.

---

7. Example Program: Union Demonstration

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

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

int main() {
    union data d;

    d.i = 10;
    printf("i = %d\n", d.i);

    d.f = 3.14;
    printf("f = %.2f\n", d.f);

    return 0;
}
```

Important understanding:

- After assigning `d.f`, value of `d.i` becomes meaningless.

Explain this in viva to score well.

---

8. Difference Between Structure and Union

(Extremely common theory question)

Structure	Union
Each member has separate memory	All members share same memory
All members can store values	Only one member valid at a time
More memory required	Less memory required
Used to store related data	Used to save memory

Students should practice writing this table.

---

9. Union with Initialization

Only **one member** can be initialized at a time.

```
union data d = {10};   // initializes first member
```

This is often asked as a short theory question.

---

10. Array of Unions

(Less common, but in syllabus)

```
union data d[5];
```

Each element uses memory equal to largest member.

---

11. Passing Union to Function

Union can be passed to a function just like structure.

```
void display(union data d);
```

Rare in practicals, but important conceptually.

---

12. How Unions Come in Exams

Theory

- Define union
- Difference between structure and union
- Memory allocation in union
- Advantages of union

Practical

- Simple union demonstration program
- Memory related questions

Usually theory-heavy chapter.

---

13. When to Use Union vs Structure (Exam Thinking)

- Need all values together → **Structure**
- Need only one value at a time → **Union**
- Memory optimization required → **Union**

Write this logic in theory answers, it shows clarity.

ENUMERATION AND TYPEDEF IN C

PART A: ENUMERATION (enum)

---

1. What is Enumeration?

Concept Overview

An enumeration is a **user defined data type** that consists of a **set of named integer constants**.

In simple words,
enum is used when we want to give **meaningful names to integer values**.

Instead of remembering numbers like 0, 1, 2, we use names.

Definition (Theory Exam Ready)

An enumeration is a **user defined data type** that assigns **names to a set of integer constants**.

Important words examiners expect:

- user defined data type
- named constants
- integer values

---

2. Why enum is Needed (Logic)

Without enum:

```
int day = 2;
```

No one knows what 2 means.

With enum:

```
enum day {SUN, MON, TUE};

enum day d = MON;
```

Now code becomes:

- readable
- meaningful
- less error-prone

This is the real purpose of enum.

---

3. Declaration of enum

Syntax

```
enum enum_name {

    constant1,

    constant2,

    constant3

};
```

Example

```
enum day {

    SUN, MON, TUE, WED, THU, FRI, SAT

};
```

By default:

- SUN = 0
- MON = 1
- TUE = 2
  and so on.

This default numbering is very important for theory.

---

4. enum Variables

Declaration

```
enum day d;
```

Assignment

```
d = MON;
```

You cannot assign arbitrary values directly unless they match enum constants.

---

5. Custom Values in enum

Concept

We can manually assign values to enum constants.

Example

```
enum status {
    FAIL = 0,
    PASS = 1
};
```

Now:

- FAIL has value 0
- PASS has value 1

This is commonly used in result-based programs.

---

6. enum is Internally Integer (Important Theory Point)

Although enum uses names, internally:

- enum values are stored as **integers**

That's why we can print enum values using `%d`.

Mention this in theory answers.

---

7. Example Program: enum Usage

```
#include <stdio.h>


enum day {
    SUN, MON, TUE, WED, THU, FRI, SAT
};
int main() {
    enum day d;
    d = WED;
    printf("%d", d);
```

```
    return 0;
}
```

Output will be 3.

8. enum vs #define

(Very common theory question)

enum	#define
User defined data type	Preprocessor directive
Has type checking	No type checking
Debugging is easier	Debugging is difficult
Stored as integer	Simple text replacement

Writing this table fetches good marks.

Memory Support for enum

ENT

Enum
 Name
 Type

Helps students remember enum is a data type, not macro.

PART B: TYPEDEF

---

10. What is typedef?

Concept Overview

`typedef` is used to **create an alias (new name)** for an existing data type.

It does not create a new data type,
it just creates a **nickname**.

Definition (Theory Exam Ready)

typedef is used to **define a new name for an existing data type**.

Important words:

- new name
- existing data type

---

11. Why typedef is Used (Logic)

Some data types are:

- long
- complex

- hard to read

Example:

```
unsigned long int number;
```

With typedef:

```
typedef unsigned long int uli;

uli number;
```

Code becomes:

- shorter
- cleaner
- easier to understand

---

12. Syntax of typedef

```
typedef existing_data_type new_name;
```

---

13. Simple typedef Example

```
#include <stdio.h>


typedef int marks;
```

```c
int main() {
    marks m = 90;
    printf("%d", m);
    return 0;
}
```

Here `marks` behaves exactly like `int`.

---

14. typedef with Structures

(Very important and practical)

Without typedef

```c
struct student {
    int roll;
};
struct student s1;
```

With typedef

```c
typedef struct {
    int roll;
} student;
```

```
student s1;
```

This is extremely common in real-world C programs.

---

15. typedef vs #define

(Frequently asked theory)

typedef	#define
Creates type alias	Text substitution
Understood by compiler	Handled by preprocessor
Type checking present	No type checking
Safer	Less safe

---

16. enum + typedef Together

(Advanced but syllabus relevant)

```
typedef enum {
    LOW,
```

```
    MEDIUM,
    HIGH
} level;

int main() {
    level l = HIGH;
}
```

This improves readability a lot.

---

17. How enum and typedef Come in Exams

Theory

- Define enum
- Advantages of enum
- enum vs #define
- Define typedef
- typedef with structure

Practical

- enum based menu or status
- typedef structure declaration

Usually short but scoring questions.

---

18. Practical Writing Strategy

When writing code:

- Need named constants → **enum**
- Need shorter or cleaner type name → **typedef**
- Need both readability and clarity → use together

DYNAMIC MEMORY ALLOCATION IN C

1. What is Dynamic Memory Allocation?

Concept Overview

Dynamic Memory Allocation means **allocating memory at runtime**, that is **while the program is executing**.

Till now, we used:

```
int a[10];
```

This is **static memory allocation**.
Size is fixed at compile time and cannot change.

Dynamic memory allocation allows us to:

- decide memory size at runtime
- increase or decrease memory when needed

- use memory efficiently

Definition (Theory Exam Ready)

Dynamic memory allocation is the process of **allocating and freeing memory during program execution** using predefined library functions.

Key words examiners expect:

- runtime
- allocate and free
- memory

---

2. Why Dynamic Memory Allocation is Needed (Logic)

Consider this situation:

- You don’t know how many elements user will enter
- Array size must be decided after input

Static array fails here.

Dynamic memory solves this problem by:

- allocating memory only when required
- freeing memory after use

This concept is very important in exams.

---

3. Header File for DMA

All dynamic memory allocation functions are defined in:

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

If students forget this, program won't compile.

---

4. Dynamic Memory Allocation Functions in C

According to syllabus, C provides **four DMA functions**:

1. malloc()
2. calloc()
3. realloc()
4. free()

Memory Hack (Very important and exam-proven)

MCRF

- **M** malloc
- **C** calloc
- **Re** realloc
- **F** free

5. malloc() Function

5.1 Concept Overview

`malloc()` allocates a **block of memory** of given size in bytes.

- Memory is **uninitialized**

- Returns base address of allocated memory
- Returns NULL if allocation fails

5.2 Syntax

```
ptr = (type *)malloc(size_in_bytes);
```

5.3 Logic Explanation

- malloc reserves memory from heap
- returns address of first byte
- programmer must typecast it

5.4 Example Program: malloc()

```
#include <stdio.h>

#include <stdlib.h>


int main() {

    int *p, n, i;


    scanf("%d", &n);


    p = (int *)malloc(n * sizeof(int));
```

```
    for(i = 0; i < n; i++)
        scanf("%d", &p[i]);

    for(i = 0; i < n; i++)
        printf("%d ", p[i]);

    free(p);
    return 0;
}
```

This is a **very common practical exam program**.

Important Exam Point

If `malloc()` fails, it returns **NULL**.
Mention this in theory answers.

---

6. calloc() Function

6.1 Concept Overview

`calloc()` allocates memory for **multiple elements** and **initializes all bytes to zero**.

That is the key difference.

6.2 Syntax

```
ptr = (type *)calloc(number_of_elements, size_of_each);
```

6.3 Logic Explanation

- memory allocated in contiguous blocks
- all values initialized to 0
- slightly slower than malloc

6.4 Example Program: calloc()

```
#include <stdio.h>

#include <stdlib.h>


int main() {

    int *p, n, i;


    scanf("%d", &n);


    p = (int *)calloc(n, sizeof(int));


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

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


    free(p);
```

```
    return 0;
}
```

Output will be all zeros.

---

7. Difference Between malloc() and calloc()

(Very common theory question)

malloc	calloc
Allocates single block	Allocates multiple blocks
Memory uninitialized	Memory initialized to zero
Faster	Slightly slower

Takes one argument	Takes two arguments

Students should practice writing this table.

---

8. realloc() Function

8.1 Concept Overview

`realloc()` is used to **change the size of previously allocated memory**.

Memory can be:

- increased
- decreased

8.2 Syntax

```
ptr = (type *)realloc(ptr, new_size);
```

8.3 Logic Explanation

- realloc may move memory to new location
- old data is preserved
- new memory (if added) is uninitialized

8.4 Example Program: realloc()

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

int main() {
    int *p, i;

    p = (int *)malloc(2 * sizeof(int));

    p[0] = 10;
    p[1] = 20;

    p = (int *)realloc(p, 4 * sizeof(int));

    p[2] = 30;
    p[3] = 40;

    for(i = 0; i < 4; i++)
        printf("%d ", p[i]);

    free(p);
    return 0;
}
```

---

9. free() Function

9.1 Concept Overview

`free()` is used to **release dynamically allocated memory** back to the system.

If free is not used, memory leak occurs.

9.2 Syntax

```
free(ptr);
```

9.3 Important Exam Point

After using free:

- pointer becomes dangling
- accessing freed memory causes error

Good practice:

```
free(ptr);

ptr = NULL;
```

Mentioning this impresses examiners.

---

10. Memory Leak (Theory Concept)

What is Memory Leak?

Memory leak occurs when:

- memory is allocated
- but never released using free()

This wastes memory and slows system.

This is often asked as a short theory question.

---

11. Common Mistakes Students Make

- Forgetting stdlib.h
- Forgetting free()
- Using pointer after free()
- Wrong size calculation in malloc
- Forgetting typecasting

Examiners easily catch these.

---

12. Dynamic Memory vs Static Memory

(Very common theory question)

Static Memory	Dynamic Memory
Allocated at compile time	Allocated at runtime
Fixed size	Flexible size

Cannot be resized	Can be resized
Stack memory	Heap memory

---

13. How DMA Comes in Exams

Theory

- Define dynamic memory allocation
- Explain malloc, calloc, realloc, free
- Difference between malloc and calloc
- What is memory leak

Practical

- Array using malloc
- Resize array using realloc
- Simple calloc demonstration

Very scoring chapter if concepts are clear.

---

14. Practical Writing Strategy

Before writing DMA program:

1. Decide number of elements at runtime
2. Allocate memory using malloc or calloc

3. Check pointer usage carefully
4. Always free memory at end

FILE HANDLING IN C

1. What is File Handling?

Concept Overview

File handling is used to **store data permanently** in a file.

Till now, whatever data we used:

- variables
- arrays
- structures

was stored in **RAM**, which is temporary.
Once program ends, data is lost.

Files allow us to:

- store data permanently
- retrieve data later
- handle large data

Definition (Theory Exam Ready)

File handling in C refers to the process of **creating, reading, writing, and closing files** using predefined library functions.

Key words examiners expect:

- permanent storage
- read and write
- file

---

2. Why File Handling is Needed (Logic)

Consider a student record program.

Without file:

- data is lost after program ends

With file:

- data is saved
- can be reused next time

👉 File handling is used when **data persistence** is required.

---

3. Types of Files in C (Syllabus Based)

According to BCA syllabus, C mainly uses:

1. **Text files**
2. **Binary files** (sometimes mentioned, but focus is on text files)

In most universities, **text files** are asked more.

---

4. File Pointer

Concept Overview

To work with files, C uses a **file pointer**.

File pointer is of type `FILE`.

Declaration

```
FILE *fp;
```

This pointer connects program with the file.

This line is very important in theory answers.

---

5. Steps in File Handling

(Very important theory question)

Every file handling program follows the same steps.

Memory Hack (Very useful)

FOWC

- **F** File pointer declaration
- **O** Open file
- **W** Work (read/write)
- **C** Close file

If students remember FOWC, file programs never go wrong.

---

6. Opening a File – fopen()

Concept Overview

`fopen()` is used to **open a file** in a specific mode.

Syntax

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

File Modes (Very Important)

Mode	Meaning
r	read
w	write
a	append
r+	read and write
w+	write and read
a+	append and read

Mention at least r, w, a in exams.

Important Exam Point

If file cannot be opened, `fopen()` returns **NULL**.

---

7. Closing a File – fclose()

Concept Overview

`fclose()` closes the file and releases resources.

Syntax

```
fclose(fp);
```

Always close file after use.

---

8. Writing to a File (Formatted Output)

8.1 fprintf() Function

Concept

Used to write formatted data to file.

Syntax

```
fprintf(fp, "format", variables);
```

Example Program: Writing to File

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

```c
int main() {

    FILE *fp;

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


    fprintf(fp, "Hello C File Handling");


    fclose(fp);

    return 0;

}
```

This creates `data.txt` and writes text into it.

---

9. Reading from a File (Formatted Input)

9.1 fscanf() Function

Concept

Used to read formatted data from file.

Syntax

```c
fscanf(fp, "format", &variables);
```

Example Program: Reading from File

```
#include <stdio.h>


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

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

    fscanf(fp, "%s", str);
    printf("%s", str);

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

---

10. Character Level File Functions

These functions work **character by character**.

10.1 fputc()

Writes a single character to file.

```
fputc('A', fp);
```

10.2 fgetc()

Reads a single character from file.

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

Example Program: Character File I/O

```
#include <stdio.h>


int main() {
    FILE *fp;
    char ch;

    fp = fopen("char.txt", "w");
    fputc('A', fp);
    fclose(fp);

    fp = fopen("char.txt", "r");
    ch = fgetc(fp);
    printf("%c", ch);
    fclose(fp);
```

```
    return 0;
}
```

---

11. String Level File Functions

11.1 fputs()

Writes a string to file.

```
fputs("Hello", fp);
```

11.2 fgets()

Reads a string from file.

```
fgets(str, size, fp);
```

Example Program: String File I/O

```
#include <stdio.h>

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

```
    fp = fopen("string.txt", "w");
    fputs("Welcome to C", fp);
    fclose(fp);

    fp = fopen("string.txt", "r");
    fgets(str, 50, fp);
    printf("%s", str);
    fclose(fp);

    return 0;
}
```

---

12. Checking File Open Error

(Good practice and exam-worthy)

```
if(fp == NULL) {
    printf("File cannot be opened");
    return 0;
}
```

Mentioning this improves quality of answer.

---

13. Common Mistakes Students Make

- Forgetting FILE *fp
- Forgetting to close file
- Wrong file mode
- Reading from file opened in write mode
- Forgetting to check NULL

Examiners catch these very easily.

---

14. How File Handling Comes in Exams

Theory

- Define file handling
- File pointer
- Steps in file handling
- File modes
- fprintf vs fscanf

Practical

- Write data to file
- Read data from file
- Copy contents from one file to another
- Student record using file

---

15. Example Practical: Copy File Content

(Very common exam question)

```
#include <stdio.h>


int main() {

    FILE *fs, *fd;

    char ch;


    fs = fopen("source.txt", "r");

    fd = fopen("dest.txt", "w");


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

        fputc(ch, fd);


    fclose(fs);

    fclose(fd);


    return 0;

}
```

This program shows real file handling logic.

---

16. Text File vs Binary File (Theory Short Note)

Text File	Binary File
Human readable	Not readable
Slower	Faster
Stores data as characters	Stores data in binary

Even if binary is not asked practically, theory mention is useful.

---

17. Practical Writing Strategy

While writing file programs:

1. Declare file pointer
2. Open file with correct mode
3. Perform read/write
4. Close file

COMMAND LINE ARGUMENTS IN C

1. What are Command Line Arguments?

Concept Overview

Command line arguments are values **passed to a program at the time of execution**, not during runtime input.

Instead of using `scanf()`, data is given **while running the program**.

Definition (Theory Exam Ready)

Command line arguments are values passed to a C program **through the command line at the time of execution**.

Key words examiners expect:

- command line
- execution time
- arguments

---

2. Why Command Line Arguments are Needed (Logic)

Command line arguments are useful when:

- input should be given once
- no user interaction is required
- programs are run from terminal or scripts

They are mostly used in:

- system programs
- batch processing
- file based utilities

---

3. main() with Command Line Arguments

To receive command line arguments, `main()` is written as:

```
int main(int argc, char *argv[])
```

This line is extremely important and often asked directly.

---

4. Understanding argc and argv

4.1 argc (Argument Count)

- Stores **total number of arguments**
- Includes program name
- Type: int

Example:
If we pass two values, argc = 3.

4.2 argv (Argument Vector)

- Stores arguments as **array of strings**
- argv[0] always stores **program name**
- argv[1], argv[2], ... store actual arguments

Type: array of character pointers.

Memory Hack (Very helpful)

PAN

- **P** Program name
- **A** Arguments
- **N** Number
- argv[0] → Program name
- argv[1…] → Arguments
- argc → Number of arguments

---

5. Basic Example Program

```
#include <stdio.h>


int main(int argc, char *argv[]) {
    int i;

    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);

    return 0;
}
```

This prints:

- program name

- all arguments

---

6. How to Run Program with Command Line Arguments

Example command:

```
program.exe 10 20
```

Then:

- argc = 3
- argv[0] = "program.exe"
- argv[1] = "10"
- argv[2] = "20"

Students must be able to explain this clearly.

---

7. Converting Arguments to Integer

Command line arguments are **strings by default**.

To perform arithmetic, we convert them.

Function used:

```
atoi()
```

Defined in:

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

---

8. Example Program: Add Two Numbers Using Command Line Arguments

(Very common exam practical)

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {

    int a, b, sum;


    a = atoi(argv[1]);

    b = atoi(argv[2]);


    sum = a + b;


    printf("Sum = %d", sum);

    return 0;

}
```

Important Note

Always ensure:

- argc is sufficient
- arguments exist before accessing argv

Mention this in theory answers.

---

9. Command Line Arguments and Arrays (Logic Link)

- argv is an array
- each element is a string
- internally, command line arguments are **array of pointers**

Mentioning this shows strong understanding.

---

10. How Command Line Arguments Come in Exams

Theory

- Define command line arguments
- Explain argc and argv
- Difference between runtime input and command line input

Practical

- Print arguments
- Add numbers using command line arguments
- Count number of arguments

Usually a short but scoring question.

---

11. Practical Writing Strategy

Before writing code:

1. Write correct main() format
2. Remember argv stores strings
3. Convert when required
4. Always consider argc value

Following this makes command line programs safe.

PREPROCESSOR DIRECTIVES IN C

1. What is a Preprocessor?

Concept Overview

Before a C program is compiled, it is first processed by a **preprocessor**.

The preprocessor:

- handles special commands
- works **before compilation**
- does not execute code, only prepares it

These commands are called **preprocessor directives**.

Definition (Theory Exam Ready)

Preprocessor directives are special instructions that are **processed before compilation** of a C program.

Important words examiners expect:

- before compilation
- special instructions
- preprocessor

---

2. Characteristics of Preprocessor Directives

Very important theory points:

- Start with #
- No semicolon at the end
- Executed before compilation
- Not part of C language syntax

Writing these points fetches marks.

---

3. Types of Preprocessor Directives

According to syllabus, preprocessor directives are mainly:

1. File inclusion directives

2. Macro definitions
3. Conditional compilation directives

We'll cover all three properly.

---

4. File Inclusion Directives

4.1 Concept Overview

File inclusion directives are used to **include header files** in a program.

Header files contain:

- function declarations
- macros
- constants

4.2 #include Directive

Syntax

```
#include <filename>

#include "filename"
```

Difference Between < > and " "

< >	" "

System header files	User defined header files
Searches standard paths	Searches current directory first

This difference is commonly asked.

Example

```
#include <stdio.h>

#include "myfile.h"
```

---

5. Macro Definition Directive (#define)

5.1 Concept Overview

`#define` is used to define **macros**, which are symbolic constants or small code replacements.

Macro is a **text substitution**, not a variable.

Definition (Exam Ready)

A macro is a symbolic constant defined using `#define` that performs **text replacement** before compilation.

5.2 Defining Constants Using #define

```
#define PI 3.14
```

Whenever PI appears, it is replaced by 3.14.

Example Program

```
#include <stdio.h>

#define PI 3.14


int main() {

    float r = 2;

    printf("%f", PI * r * r);

    return 0;

}
```

Important Exam Point

- Macros do not use memory
- No type checking in macros

---

6. Macro with Arguments

(Frequently asked theory topic)

Syntax

```
#define macro_name(x) expression
```

Example

```
#define SQR(x) x*x
```

Example Program

```
#include <stdio.h>

#define SQR(x) x*x


int main() {

    printf("%d", SQR(5));

    return 0;

}
```

Important Warning (Good Theory Point)

```
SQR(2+3)  // becomes 2+3*2+3 = 11
```

This shows macros are text substitution.

Mentioning this shows depth.

---

7. #undef Directive

Concept

Used to **undefine a macro**.

Syntax

```
#undef PI
```

Example

```
#define A 10

#undef A
```

---

8. Conditional Compilation Directives

Concept Overview

Conditional compilation allows us to **compile code conditionally**, based on conditions.

Used to:

- enable or disable code
- handle platform-specific code

---

9. #if, #else, #endif

Syntax

```
#if condition
```

```
    statements
#else
    statements
#endif
```

Example

```
#include <stdio.h>
#define X 5

int main() {
#if X > 3
    printf("X is greater");
#else
    printf("X is smaller");
#endif
    return 0;
}
```

---

10. #ifdef and #ifndef

(Very common theory question)

#ifdef

Checks if macro is defined.

```
#ifdef PI
```

#ifndef

Checks if macro is not defined.

```
#ifndef PI
```

Example Program

```
#include <stdio.h>

#define PI 3.14


int main() {

#ifdef PI

    printf("PI is defined");

#endif

    return 0;

}
```

---

11. Difference Between #define and const

(Very common theory question)

#define	const
Preprocessor directive	Keyword
No type checking	Type checking present
No memory allocation	Memory allocated
Text substitution	Variable

This table is very scoring.

---

12. How Preprocessor Directives Come in Exams

Theory

- Define preprocessor
- Types of preprocessor directives
- Macro with example
- #include syntax
- Difference between #define and const

Practical

- Program using #define
- Macro with argument

Mostly theory but easy marks.

---

13. Practical Writing Strategy

While writing answers:

- Start with definition
- Mention processed before compilation
- Give one simple example
- Write at least one advantage

ERRORS IN C

1. What is an Error?

Concept Overview

An error is a **mistake in a program** that causes:

- compilation failure
- abnormal program execution
- incorrect output

Errors can occur at different stages of program execution.

Definition (Theory Exam Ready)

An error is a **fault or mistake in a program** that prevents it from producing the correct output or executing properly.

Key words examiners expect:

- fault
- incorrect output
- execution

---

2. Classification of Errors in C

According to standard syllabus, errors in C are classified into **three types**:

1. Syntax errors
2. Runtime errors
3. Logical errors

All three must be explained properly in exams.

Memory Hack (Simple and exam-proven)

SRL

- **S** Syntax
- **R** Runtime
- **L** Logical

If students remember SRL, they never miss any error type.

---

3. Syntax Errors

3.1 Concept Overview

Syntax errors occur when the **rules of C language are violated**.

These errors are detected by the **compiler**.

3.2 Common Causes of Syntax Errors

- Missing semicolon
- Missing brackets `{ }`
- Wrong spelling of keywords
- Missing header files
- Wrong data type declaration

3.3 Example: Syntax Error Program

```
#include <stdio.h>


int main() {

    int a = 10

    printf("%d", a);

    return 0;

}
```

Error: missing semicolon after `int a = 10`

Program will **not compile**.

Important Exam Point

Syntax errors are detected at **compile time**.

Always mention this in theory answers.

---

4. Runtime Errors

4.1 Concept Overview

Runtime errors occur **while the program is running**, not during compilation.

Program compiles successfully but **crashes or behaves abnormally**.

4.2 Common Causes of Runtime Errors

- Division by zero
- Accessing invalid memory
- Array index out of bounds
- Using uninitialized pointer

4.3 Example: Runtime Error Program

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

```
int main() {
    int a = 10, b = 0;
    printf("%d", a / b);
    return 0;
}
```

This program compiles, but causes **runtime error**.

Important Exam Point

Runtime errors are detected **during program execution**.

---

5. Logical Errors

5.1 Concept Overview

Logical errors occur when the **logic of the program is wrong**.

Program:

- compiles successfully
- runs without crash
- but produces **wrong output**

This is the most dangerous error.

5.2 Common Causes of Logical Errors

- Wrong formula

- Wrong condition
- Wrong operator
- Incorrect loop logic

5.3 Example: Logical Error Program

```
#include <stdio.h>


int main() {
    int l = 10, b = 5;
    int area;


    area = l + b;   // wrong logic
    printf("%d", area);
    return 0;
}
```

Correct formula should be:

```
area = l * b;
```

Compiler won’t catch this. Only logic understanding helps.

---

6. Difference Between Error Types

(Very common theory question)

Syntax Error	Runtime Error	Logical Error
Detected by compiler	Detected during execution	Detected by programmer
Compile time	Execution time	After checking output
Program won't run	Program crashes	Program runs but wrong result

Students should practice this table.

---

7. Debugging

Concept Overview

Debugging is the process of:

- finding errors
- correcting errors
- testing program again

This is often asked as a short note.

Debugging Techniques (Exam Points)

- Read error messages carefully

- Check syntax first
- Use printf() to trace values
- Check conditions and loops

Mentioning printf debugging shows practical understanding.

---

8. Common Student Mistakes (Exam Focus)

- Treating logical error as syntax error
- Ignoring compiler warnings
- Assuming program is correct because it runs
- Not checking boundary conditions

Examiners often ask viva questions from here.

---

9. How Errors in C Come in Exams

Theory

- Define error
- Types of errors
- Difference between syntax and runtime error
- Explain logical error with example

Practical

- Identify error in given code
- Correct the program
- Find output of faulty code

10. Practical Writing Strategy

When asked to debug:

1. Check syntax first
2. Check runtime conditions
3. Verify logic manually
4. Test with sample inputs

11. Final Summary Line (Good for Exams)

Syntax errors stop compilation,
runtime errors stop execution,
logical errors give wrong results.

Quick Revision

C PROGRAMMING – COMPLETE REVISION NOTES

(Theory + Practical Focused)

---

1. Structure of C Program

Memory hack: HMVIPO

- **H** Header files
- **M** main() function
- **V** Variable declaration
- **I** Input statements
- **P** Processing logic
- **O** Output statements

👉 If HMVIPO order is correct, program flow is always correct.

---

2. Tokens in C

Types:

- Keywords
- Identifiers
- Constants
- Strings

- Operators
- Special symbols

👉 Tokens are the **smallest units** of a C program.
Direct theory question, comes many times.

---

3. Operators

Main memory hack: ARLUA

- **A** Arithmetic → `+ - * / %`
- **R** Relational → `< > <= >= == !=`
- **L** Logical → `&& || !`
- **U** Unary → `++ --`
- **A** Assignment → `= += -= *= /= %=`

Other operators:

- Conditional `?:`
- Bitwise `& | ^ ~ << >>`
- `sizeof`

👉 `%` only works with integers.
👉 `==` is comparison, `=` is assignment. Very common mistake.

---

4. Decision Making Statements

Types:

- if
- if–else
- else–if ladder
- nested if

Quick logic rule:

- 1 condition → if
- 2 outcomes → if–else
- multiple ranges → else–if ladder
- dependent conditions → nested if

👉 Condition true = non zero
👉 Condition false = zero

---

5. Loops

Types:

- for
- while
- do–while

Memory hack: KUM

- **K** Known iterations → for
- **U** Unknown iterations → while
- **M** Must execute once → do–while

👉 for loop = compact
👉 while loop = condition based
👉 do–while runs at least once

Nested loop rule:

- Outer loop → rows
- Inner loop → columns

---

6. Jump Statements

Memory hack: BCRG

- break
- continue
- return
- goto
- `break` exits loop/switch
- `continue` skips current iteration
- `return` exits function
- `goto` jumps to label (not recommended)

👉 break vs continue difference is very commonly asked.

---

7. Strings

Definition:
String = character array ending with `'\0'`

Important points:

- Not a data type in C
- Stored as array of char
- `'\0'` marks end of string

Common functions (memory hack): LSCC

- strlen
- strcpy
- strcat
- strcmp

👉 Never compare strings using `==`
👉 Always ensure enough size for `'\0'`

---

8. Functions

Definition:
Block of code that performs a specific task.

Parts (very important): DDC

- Declaration
- Definition
- Call

Types of user defined functions:

1. No argument, no return
2. Argument, no return

3. No argument, return
4. Argument and return (most used)

👉 By default C uses call by value.

---

9. Recursion

Definition:
Function calling itself.

Mandatory rule: CBS

- Call itself
- Base condition
- Stop

Without base condition → infinite recursion.

👉 Uses stack memory
👉 Slower than loops but cleaner logic

Common programs:

- factorial
- fibonacci
- sum of digits

---

10. Storage Classes

Types (memory hack): ASER

Storage	Scope	Lifetime	Default
auto	local	block	garbage
static	local	program	0
extern	global	program	0
register	local	block	garbage

👉 static retains value between function calls.
Very important.

---

11. Pointers

Definition:
Variable that stores address of another variable.

Operators:

- `&` address of
- `*` value at address

Key points:

- Pointer type must match variable type
- Pointer arithmetic depends on data type size
- Array name is base address

👉 `*(a+i)` = `a[i]`

---

12. Structures

Definition:
User defined data type that stores **different data types** under one name.

Memory hack: DNVA

- Define structure
- Name variable
- Values assign
- Access using dot

👉 Structure declaration does NOT allocate memory.
👉 Use `.` for variable, `->` for pointer.

---

13. Unions

Definition:
User defined data type where **all members share same memory**.

Key rule:

- Memory = size of largest member
- Only one member valid at a time

👉 Union saves memory.
👉 Very common theory comparison with structure.

---

14. enum and typedef

enum

- User defined data type

- Named integer constants
- Improves readability

👉 Internally stored as int.

typedef

- Creates alias for existing data type
- Makes code cleaner

👉 typedef does NOT create new data type.

---

15. Dynamic Memory Allocation

Header file: `stdlib.h`

Functions (memory hack): MCReF

- malloc
- calloc
- realloc
- free

malloc	calloc
Uninitialized	Initialized to zero
One argument	Two arguments

👉 Always free memory after use.
👉 Forgetting free causes memory leak.

---

16. File Handling

File pointer: `FILE *fp;`

Steps (memory hack): FOWC

- File pointer
- Open
- Work
- Close

File modes:

- r, w, a, r+, w+, a+

Common functions:

- fprintf, fscanf
- fgetc, fputc
- fgets, fputs

👉 fopen returns NULL if file not found.

---

17. Command Line Arguments

main format:

```
int main(int argc, char *argv[])
```

- argc → argument count
- argv → argument vector (array of strings)

👉 argv[0] = program name
👉 use `atoi()` to convert to int

---

18. Preprocessor Directives

- Start with #
- Executed before compilation
- No semicolon

Types:

- #include
- #define
- Conditional directives

👉 Macro is text replacement, not variable.

---

19. Errors in C

Types (memory hack): SRL

- Syntax error → compile time
- Runtime error → execution time
- Logical error → wrong output

👉 Logical error is most dangerous.

Last Day Exam Strategy

- Write definitions cleanly
- Draw tables wherever possible
- Use correct keywords
- For practical, write logic first then code
- Follow memory hacks, they actually help

Important Questions

Most Important C Programming Questions

(90%+ EXAM PROBABILITY | THEORY + PRACTICAL)

SECTION A: VERY SHORT / SHORT THEORY QUESTIONS

(1–2 marks, very high repeat rate)

These come almost every year.

1. Define C language
2. List features of C
3. What is a token in C
4. Define identifier
5. What is a keyword
6. What is a constant
7. What is an operator
8. What is a loop
9. Difference between `=` and `==`
10. What is a string
11. What is `'\0'`
12. Define function
13. What is recursion
14. What is pointer

15. What is structure
16. What is union
17. What is enum
18. What is typedef
19. What is file handling
20. What is dynamic memory allocation
21. What is preprocessor directive
22. What is syntax error
23. What is runtime error
24. What is logical error

SECTION B: SHORT NOTES / EXPLAIN (3–5 MARKS)

(Very high probability)

🔹 Basics

1. Features of C language
2. Structure of C program
3. Compilation process of C program

🔹 Operators

4. Types of operators in C
5. Arithmetic operators with example
6. Relational and logical operators

7. Operator precedence and associativity

🔹 Decision Making

8. if statement with syntax and example
9. if–else vs else–if ladder
10. Nested if statement

🔹 Loops

11. Explain for loop with example
12. Difference between for and while
13. Difference between while and do–while
14. Nested loops with example

🔹 Jump Statements

15. break and continue
16. return statement
17. goto statement (with limitation)

🔹 Strings

18. What is string in C

19. String handling functions
20. strlen vs strcpy
21. strcmp with example

🔹 Functions

22. User defined functions
23. Types of user defined functions
24. Call by value
25. Recursion with example

🔹 Storage Classes

26. Storage classes in C
27. static storage class
28. extern keyword

🔹 Pointers

29. Pointer definition and declaration
30. Pointer arithmetic
31. Pointer and array relationship
32. Call by reference

🔹 Structures & Unions

33. Structure definition and advantages
34. Array of structures
35. Structure vs union
36. Union memory allocation

🔹 enum & typedef

37. Enumeration with example
38. enum vs #define
39. typedef with structure

🔹 Dynamic Memory

40. malloc and calloc
41. malloc vs calloc
42. realloc and free
43. Memory leak

🔹 File Handling

44. File pointer and file modes
45. Steps in file handling
46. fprintf vs fscanf

47. Text file vs binary file

🔹 Command Line Arguments

48. argc and argv

49. Program using command line arguments

🔹 Preprocessor

50. Types of preprocessor directives

51. #define and macros

52. #define vs const

🔹 Errors

53. Types of errors in C

54. Syntax vs runtime error

55. Logical error with example

SECTION C: LONG THEORY QUESTIONS (8–10 MARKS)

These questions **rotate**, but from same topics.

1. Explain operators in C with examples
2. Explain decision making statements in C

3. Explain looping statements in C
4. Explain strings and string handling functions
5. Explain functions and recursion
6. Explain pointers with suitable examples
7. Explain structures and unions with differences
8. Explain dynamic memory allocation functions
9. Explain file handling in C
10. Explain preprocessor directives
11. Explain storage classes in C

👉 Any **ONE OR TWO** of these will surely appear.

PRACTICAL EXAM IMP QUESTIONS

🔹 Operators / Decision Making

1. Program to find even or odd
2. Program to find largest of two numbers
3. Program to find largest of three numbers
4. Program to check positive or negative number

🔹 Loops

5. Program to print numbers from 1 to n

6. Program to find sum of digits
7. Program to reverse a number
8. Program to find factorial using loop
9. Program to check prime number

🔹 Patterns (VERY HIGH CHANCE)

10. Star triangle pattern
11. Number pattern using nested loop

🔹 Arrays

12. Program to read and display array elements
13. Program to find largest element in array
14. Program to calculate sum and average of array

🔹 Strings (ALWAYS ASKED)

15. Program to find length of string
16. Program to copy string
17. Program to compare two strings
18. Program to reverse a string
19. Program to check palindrome string

🔹 Functions

20. Program to add two numbers using function

21. Program to find factorial using function

🔹 Recursion (VERY COMMON)

22. Factorial using recursion

23. Fibonacci using recursion

🔹 Pointers (HIGH WEIGHT)

24. Swap two numbers using pointers

25. Array traversal using pointer

🔹 Structures

26. Program to store and display student details

27. Array of structures program

🔹 Union

28. Program to demonstrate union

🔹 Dynamic Memory

29. Program using malloc

30. Program using calloc

🔹 File Handling (MOST IMPORTANT)

31. Program to write data to file

32. Program to read data from file

33. Program to copy one file to another

🔹 Command Line Arguments

34. Program to add numbers using command line arguments

Cheatsheet

Cheatsheet for Practical and Theory

Topic	Explanation	Memory Hack	Must-Know Syntax
Structure of C Program	Basic layout of a C program showing flow of execution	**HMVIPO**	`#include <stdio.h>int main(){ declarations; logic; return 0; }`
Tokens	Smallest units of a C program	KICOSS	keywords, identifiers, constants, operators, strings, symbols
Keywords	Reserved words with fixed meaning	No rename rule	`int`, `float`, `if`, `while`, `return`
Identifiers	Names given to variables/functions	Name rule	`int marks;`
Operators	Symbols performing operations	**ARLUA**	`+ - * / %`, `==`, `&&`, `++`, `=`
Arithmetic Operators	Perform calculations	BODMAS reminder	`a+b`, `a%b`
Relational Operators	Compare values	Always return 0/1	`< > <= >= == !=`
Logical Operators	Combine conditions	AND needs all true	`&&
Unary Operators	Work on single operand	Pre vs Post	`++a`, `a--`
Assignment Operators	Assign/update values	Short update	`+= -= *= /= %=`
Conditional Operator	Short if-else	One-line decision	`cond ? x : y;`
Decision Making	Control program flow	True = non-zero	`if(condition)`

if Statement	Executes on true condition	Single path	`if(cond){}`
if-else	Two possible paths	Yes/No logic	`if(){ } else{ }`
else-if Ladder	Multiple conditions	Top-down check	`else if(cond)`
Nested if	if inside if	Dependent logic	`if(){ if(){ } }`
Loops	Repeat statements	**KUM**	`for`, `while`, `do-while`
for Loop	Known iterations	Counter loop	`for(i=0;i<n;i++)`
while Loop	Unknown iterations	Entry control	`while(cond)`
do-while Loop	Runs at least once	Exit control	`do{}while(cond);`
Nested Loop	Loop inside loop	ORIC (row-col)	`for(){ for(){ } }`
Jump Statements	Alter flow instantly	**BCRG**	`break`, `continue`
break	Exit loop/switch	Full exit	`break;`
continue	Skip iteration	Partial skip	`continue;`
return	Exit function	Go back	`return value;`
goto	Jump to label	Avoid use	`goto label;`
Strings	Char array ending with null	`'\0' rule`	`char s[20];`
String Input	Read string	No & needed	`scanf("%s", s);`
String Functions	Built-in operations	**LSCC**	`strlen(s)`
strlen	Length of string	No `\0` count	`strlen(str)`
strcpy	Copy string	Dest big enough	`strcpy(a,b)`
strcat	Join strings	End attach	`strcat(a,b)`

strcmp	Compare strings	0 = equal	`strcmp(a,b)`
Functions	Block for specific task	**DDC**	`int fun(){}`
Function Declaration	Tells compiler	Before main	`int add(int,int);`
Function Definition	Actual code	Logic part	`int add(a,b){}`
Function Call	Executes function	Control transfer	`add(2,3);`
Recursion	Function calling itself	**CBS**	`fun(){ fun(); }`
Base Condition	Stops recursion	Mandatory stop	`if(n==0)`
Storage Classes	Define scope & life	**ASER**	`static int x;`
auto	Default local	Auto memory	`auto int a;`
static	Retains value	Permanent life	`static int c;`
extern	Global access	Shared var	`extern int x;`
register	Fast access	No address	`register int i;`
Pointers	Store address	**AV**	`int *p;`
Address Operator	Gives address	Location	`&a`
Dereference	Value at address	Actual data	`*p`
Pointer & Array	Array = base addr	**ABA**	`*(a+i)`
Call by Reference	Modify original	Address pass	`fun(&a)`
Structures	User defined data type	**DNVA**	`struct s{}`
Structure Access	Member access	Dot rule	`s.marks`

Array of Structure	Multiple records	Loop + dot	`s[i].roll`
Structure Pointer	Pointer to struct	Arrow rule	`p->roll`
Unions	Shared memory type	**SM**	`union u{}`
Union Memory	Largest member	One value	`sizeof(union)`
enum	Named int constants	ENT	`enum day{MON}`
typedef	Type alias	Nickname rule	`typedef int I;`
Dynamic Memory	Runtime allocation	**MCReF**	`malloc()`
malloc	Allocate garbage	Single block	`malloc(n*sizeof)`
calloc	Allocate zero	Multi block	`calloc(n,size)`
realloc	Resize memory	Expand/shrink	`realloc(p,n)`
free	Release memory	Avoid leak	`free(p);`
File Handling	Permanent storage	**FOWC**	`FILE *fp;`
fopen	Open file	Mode matters	`fopen("a","r")`
fclose	Close file	Must close	`fclose(fp);`
fprintf	Write file	Like printf	`fprintf(fp,...)`
fscanf	Read file	Like scanf	`fscanf(fp,...)`
File Modes	Access type	rwa rule	`"r","w","a"`
Command Line Args	Exec time input	**PAN**	`int main(argc,argv)`
argc	Argument count	Includes prog	`argc`
argv	Argument values	String array	`argv[i]`

Preprocessor	Before compile	# rule	`#include`
#define	Macro	Text replace	`#define PI 3.14`
Conditional Compile	Compile control	ifdef rule	`#ifdef X`
Errors	Program faults	**SRL**	syntax/runtime/logical
Syntax Error	Rule violation	Compile time	missing `;`
Runtime Error	Crash on run	Exec time	divide by zero
Logical Error	Wrong output	Brain error	wrong formula

Top 50 Practical Questions

Top 50 C Programming Practical Questions

1. Write a C program to add two numbers.

```
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    printf("%d",a+b);
    return 0;
}
```

---

2. Write a C program to subtract two numbers.

```
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    printf("%d",a-b);
    return 0;
}
```

---

3. Write a C program to multiply two numbers.

```
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    printf("%d",a*b);
    return 0;
}
```

---

4. Write a C program to divide two numbers.

```c
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    printf("%d",a/b);
    return 0;
}
```

---

5. Write a C program to check even or odd.

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d",&n);
    if(n%2==0) printf("Even");
    else printf("Odd");
    return 0;
}
```

---

6. Write a C program to find largest of two numbers.

```c
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    if(a>b) printf("%d",a);
    else printf("%d",b);
    return 0;
}
```

---

7. Write a C program to find largest of three numbers.

```c
#include <stdio.h>
int main() {
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    if(a>b && a>c) printf("%d",a);
    else if(b>c) printf("%d",b);
    else printf("%d",c);
    return 0;
}
```

---

8. Write a C program to find factorial using loop.

```c
#include <stdio.h>
int main() {
    int n,f=1;
    scanf("%d",&n);
    for(int i=1;i<=n;i++) f*=i;
    printf("%d",f);
    return 0;
}
```

---

9. Write a C program to find factorial using recursion.

```c
#include <stdio.h>
int fact(int n){
    if(n==0) return 1;
    return n*fact(n-1);
}
int main(){
    int n;
    scanf("%d",&n);
    printf("%d",fact(n));
    return 0;
}
```

10. Write a C program to check prime number.

```
#include <stdio.h>
int main() {
    int n,i,flag=1;
    scanf("%d",&n);
    if(n<=1) flag=0;
    for(i=2;i<=n/2;i++)
        if(n%i==0) flag=0;
    if(flag) printf("Prime");
    else printf("Not Prime");
    return 0;
}
```

---

11. Write a C program to reverse a number.

```
#include <stdio.h>
int main() {
    int n,rev=0;
    scanf("%d",&n);
    while(n!=0){
        rev=rev*10+n%10;
        n/=10;
    }
    printf("%d",rev);
    return 0;
}
```

---

12. Write a C program to check palindrome number.

```
#include <stdio.h>
int main() {
    int n,temp,rev=0;
    scanf("%d",&n);
    temp=n;
    while(n!=0){
```

```
        rev=rev*10+n%10;
        n/=10;
    }
    if(temp==rev) printf("Palindrome");
    else printf("Not Palindrome");
    return 0;
}
```

---

13. Write a C program to find sum of digits.

```
#include <stdio.h>
int main() {
    int n,sum=0;
    scanf("%d",&n);
    while(n!=0){
        sum+=n%10;
        n/=10;
    }
    printf("%d",sum);
    return 0;
}
```

14. Write a C program to generate Fibonacci series.

```
#include <stdio.h>
int main() {
    int n,a=0,b=1,c;
    scanf("%d",&n);
    printf("%d %d ",a,b);
    for(int i=3;i<=n;i++){
        c=a+b;
        printf("%d ",c);
        a=b;
        b=c;
    }
    return 0;
}
```

15. Write a C program using switch case calculator.

```c
#include <stdio.h>
int main() {
    int a,b;
    char op;
    scanf(" %c",&op);
    scanf("%d%d",&a,&b);
    switch(op){
        case '+': printf("%d",a+b); break;
        case '-': printf("%d",a-b); break;
        case '*': printf("%d",a*b); break;
        case '/': printf("%d",a/b); break;
    }
    return 0;
}
```

---

16. Write a C program to read and print array elements.

```c
#include <stdio.h>
int main() {
    int n,a[50];
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    for(int i=0;i<n;i++) printf("%d ",a[i]);
    return 0;
}
```

---

17. Write a C program to find largest element in array.

```c
#include <stdio.h>
int main() {
    int n,a[50],max;
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    max=a[0];
```

```
    for(int i=1;i<n;i++)
        if(a[i]>max) max=a[i];
    printf("%d",max);
    return 0;
}
```

---

18. Write a C program to reverse an array.

```
#include <stdio.h>
int main() {
    int n,a[50];
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    for(int i=n-1;i>=0;i--) printf("%d ",a[i]);
    return 0;
}
```

---

19. Write a C program to sort array in ascending order.

```
#include <stdio.h>
int main() {
    int n,a[50],t;
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        for(int j=i+1;j<n;j++)
            if(a[i]>a[j]){
                t=a[i]; a[i]=a[j]; a[j]=t;
            }
    for(int i=0;i<n;i++) printf("%d ",a[i]);
    return 0;
}
```

---

20. Write a C program to find length of string.

```
#include <stdio.h>
int main() {
    char s[100];
    int i=0;
    scanf("%s",s);
    while(s[i]!='\0') i++;
    printf("%d",i);
    return 0;
}
```

---

21. Write a C program to reverse a string.

```
#include <stdio.h>
#include <string.h>
int main() {
    char s[100];
    scanf("%s",s);
    for(int i=strlen(s)-1;i>=0;i--)
        printf("%c",s[i]);
    return 0;
}
```

---

22. Write a C program to compare two strings.

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[50],b[50];
    scanf("%s%s",a,b);
    if(strcmp(a,b)==0) printf("Equal");
    else printf("Not Equal");
    return 0;
}
```

23. Write a C program to copy one string to another.

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[50],b[50];
    scanf("%s",a);
    strcpy(b,a);
    printf("%s",b);
    return 0;
}
```

---

24. Write a C program using function to add two numbers.

```
#include <stdio.h>
int add(int x,int y){ return x+y; }
int main(){
    int a,b;
    scanf("%d%d",&a,&b);
    printf("%d",add(a,b));
    return 0;
}
```

---

25. Write a C program to swap two numbers using function.

```
#include <stdio.h>
void swap(int *a,int *b){
    int t=*a;
    *a=*b;
    *b=t;
}
int main(){
    int x,y;
    scanf("%d%d",&x,&y);
    swap(&x,&y);
```

```
    printf("%d %d",x,y);
    return 0;
}
```

---

26. Write a C program using pointer to access variable.

```
#include <stdio.h>
int main() {
    int a=10;
    int *p=&a;
    printf("%d",*p);
    return 0;
}
```

---

27. Write a C program using structure to store student data.

```
#include <stdio.h>
struct student{
    int roll;
    char name[50];
};
int main(){
    struct student s;
    scanf("%d%s",&s.roll,s.name);
    printf("%d %s",s.roll,s.name);
    return 0;
}
```

---

28. Write a C program to read multiple students using structure.

```
#include <stdio.h>
struct student{
```

```
    int roll;
    char name[50];
};
int main(){
    struct student s[2];
    for(int i=0;i<2;i++)
        scanf("%d%s",&s[i].roll,s[i].name);
    for(int i=0;i<2;i++)
        printf("%d %s\n",s[i].roll,s[i].name);
    return 0;
}
```

---

29. Write a C program to write data into a file.

```
#include <stdio.h>
int main(){
    FILE *fp;
    fp=fopen("data.txt","w");
    fprintf(fp,"Hello C");
    fclose(fp);
    return 0;
}
```

---

30. Write a C program to read data from a file.

```
#include <stdio.h>
int main(){
    FILE *fp;
    char ch;
    fp=fopen("data.txt","r");
    while((ch=fgetc(fp))!=EOF)
        printf("%c",ch);
    fclose(fp);
    return 0;
}
```

31. Write a C program to add two matrices.

```
#include <stdio.h>
int main() {
    int a[2][2], b[2][2], c[2][2];
    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            scanf("%d",&a[i][j]);

    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            scanf("%d",&b[i][j]);

    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            c[i][j]=a[i][j]+b[i][j];

    for(int i=0;i<2;i++){
        for(int j=0;j<2;j++)
            printf("%d ",c[i][j]);
        printf("\n");
    }
    return 0;
}
```

---

32. Write a C program to multiply two matrices.

```
#include <stdio.h>
int main() {
    int a[2][2], b[2][2], c[2][2]={0};
    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            scanf("%d",&a[i][j]);

    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            scanf("%d",&b[i][j]);
```

```
    for(int i=0;i<2;i++)
        for(int j=0;j<2;j++)
            for(int k=0;k<2;k++)
                c[i][j]+=a[i][k]*b[k][j];

    for(int i=0;i<2;i++){
        for(int j=0;j<2;j++)
            printf("%d ",c[i][j]);
        printf("\n");
    }
    return 0;
}
```

---

33. Write a C program to find sum of array elements.

```
#include <stdio.h>
int main() {
    int n,a[50],sum=0;
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
        sum+=a[i];
    }
    printf("%d",sum);
    return 0;
}
```

---

34. Write a C program to count even and odd numbers in array.

```
#include <stdio.h>
int main() {
```

```
    int n,a[50],even=0,odd=0;
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
        if(a[i]%2==0) even++;
        else odd++;
    }
    printf("Even=%d Odd=%d",even,odd);
    return 0;
}
```

---

35. Write a C program to count vowels in a string.

```
#include <stdio.h>
int main() {
    char s[100];
    int c=0;
    scanf("%s",s);
    for(int i=0;s[i]!='\0';i++){
        if(s[i]=='a'||s[i]=='e'||s[i]=='i'||s[i]=='o'||s[i]=='u'||
           s[i]=='A'||s[i]=='E'||s[i]=='I'||s[i]=='O'||s[i]=='U')
            c++;
    }
    printf("%d",c);
    return 0;
}
```

---

36. Write a C program to count digits in a number.

```
#include <stdio.h>
int main() {
    int n,count=0;
    scanf("%d",&n);
    while(n!=0){
        count++;
        n/=10;
```

```
    }
    printf("%d",count);
    return 0;
}
```

---

37. Write a C program to print ASCII value of a character.

```
#include <stdio.h>
int main() {
    char ch;
    scanf("%c",&ch);
    printf("%d",ch);
    return 0;
}
```

---

38. Write a C program to check Armstrong number.

```
#include <stdio.h>
int main() {
    int n,temp,sum=0,r;
    scanf("%d",&n);
    temp=n;
    while(n!=0){
        r=n%10;
        sum+=r*r*r;
        n/=10;
    }
    if(sum==temp) printf("Armstrong");
    else printf("Not Armstrong");
    return 0;
}
```

---

39. Write a C program using do while loop.

```c
#include <stdio.h>
int main() {
    int i=1;
    do{
        printf("%d ",i);
        i++;
    }while(i<=5);
    return 0;
}
```

---

40. Write a C program to print multiplication table of a number.

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d",&n);
    for(int i=1;i<=10;i++)
        printf("%d x %d = %d\n",n,i,n*i);
    return 0;
}
```

---

41. Write a C program to find smallest element in array.

```c
#include <stdio.h>
int main() {
    int n,a[50],min;
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    min=a[0];
    for(int i=1;i<n;i++)
        if(a[i]<min) min=a[i];
    printf("%d",min);
    return 0;
}
```

---

42. Write a C program to search an element in array.

```
#include <stdio.h>
int main() {
    int n,a[50],key,found=0;
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&a[i]);
    scanf("%d",&key);
    for(int i=0;i<n;i++)
        if(a[i]==key) found=1;
    if(found) printf("Found");
    else printf("Not Found");
    return 0;
}
```

---

43. Write a C program to check leap year.

```
#include <stdio.h>
int main() {
    int y;
    scanf("%d",&y);
    if((y%4==0 && y%100!=0) || y%400==0)
        printf("Leap Year");
    else
        printf("Not Leap Year");
    return 0;
}
```

---

44. Write a C program to find power of a number.

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

```c
int main() {
    int b,p,res=1;
    scanf("%d%d",&b,&p);
    for(int i=1;i<=p;i++)
        res*=b;
    printf("%d",res);
    return 0;
}
```

---

45. Write a C program to print star pattern.

```c
#include <stdio.h>
int main() {
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        for(int j=1;j<=i;j++)
            printf("*");
        printf("\n");
    }
    return 0;
}
```

---

46. Write a C program for menu driven arithmetic operations.

```c
#include <stdio.h>
int main() {
    int ch,a,b;
    scanf("%d",&ch);
    scanf("%d%d",&a,&b);
    switch(ch){
        case 1: printf("%d",a+b); break;
        case 2: printf("%d",a-b); break;
        case 3: printf("%d",a*b); break;
```

```
        case 4: printf("%d",a/b); break;
    }
    return 0;
}
```

---

47. Write a C program to swap two numbers without using third variable.

```
#include <stdio.h>
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("%d %d",a,b);
    return 0;
}
```

---

48. Write a C program to check positive or negative number.

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d",&n);
    if(n>=0) printf("Positive");
    else printf("Negative");
    return 0;
}
```

---

49. Write a C program to find GCD of two numbers.

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

```c
int main() {
    int a,b;
    scanf("%d%d",&a,&b);
    while(a!=b){
        if(a>b) a=a-b;
        else b=b-a;
    }
    printf("%d",a);
    return 0;
}
```

---

50. Write a C program to find LCM of two numbers.

```c
#include <stdio.h>
int main() {
    int a,b,max;
    scanf("%d%d",&a,&b);
    max=(a>b)?a:b;
    while(1){
        if(max%a==0 && max%b==0){
            printf("%d",max);
            break;
        }
        max++;
    }
    return 0;
}
```

Passing Question Bank

C PROGRAMMING QUESTION BANK

🔹 Q1(A) VERY SHORT ANSWERS (1–2 MARKS)

👉 Out of these, **minimum 6–7 will come**

1. Who developed C language?
2. ASCII stands for _____.
3. Which escape sequence is used for new line?
4. What is the range of `char` data type?
5. Which tool is called graphical representation of program?
6. How many keywords are there in C?
7. Which loop executes at least once?
8. Array index starts from _____.
9. Which operator is used to declare pointer variable?
10. Which character terminates a string?

🔹 Q1(B) SHORT THEORY (2 MARKS)

👉 Examiner loves asking any one from here

1. Explain keyword in C.
2. Explain type casting with example.
3. Explain variable.
4. Explain flowchart symbols.

5. Explain data types in C.
6. Explain static variable.
7. Explain break statement.
8. Explain continue statement.
9. Explain recursion.
10. Explain pointer.

🔹 Q1(C) LONG THEORY (5 MARKS)

👉 **1 question almost guaranteed**

1. Explain structure of C program.
2. Explain data types available in C.
3. Explain different types of operators in C.
4. Explain variables in detail.
5. Explain flowchart with example.
6. Explain storage classes in C.
7. Explain scope and lifetime of variable.
8. Explain constants in C.
9. Explain tokens in C programming.
10. Explain compilation process of C program.

🔹 Q2(A) SHORT QUESTIONS (3 MARKS)

1. How many forms of if statement are there in C?
2. Which one is post-test loop?
3. Which loop is exit controlled loop?
4. Which loop is entry controlled loop?
5. List out conditional statements in C.
6. List out jumping statements.
7. Which statement is used to terminate loop?
8. What is the value of case in switch statement?
9. Which loop requires semicolon at the end?
10. Which jumping statement is known as unequal jumping statement?

🔹 Q2(B) SHORT THEORY / DIFFERENCE (2 MARKS)

1. Explain if…else statement.
2. Explain for loop.
3. Difference between while and do-while loop.
4. Difference between entry and exit controlled loop.
5. Explain break keyword.
6. Explain continue keyword.
7. Difference between break and continue.
8. Write syntax of switch case.
9. Write syntax of for loop.

10. Write syntax of nested if.

🔹 Q2(C) LONG ANSWER (5 MARKS)

👉 **This section alone can pass students**

1. Explain if…else statement with example.
2. Explain for loop with example.
3. Explain while loop with example.
4. Explain do-while loop with example.
5. Explain switch case with example.
6. Explain break and continue statements.
7. Explain nested if statement.
8. Write a program using switch case.
9. Write a program using loop.
10. Write output of given loop program.

🔹 Q3(A) VERY SHORT (3 MARKS)

1. Which header file is required for string functions?
2. How many values can be returned by function?
3. What is default storage class in C?
4. Write syntax of calloc().

5. Which data type is used when function returns no value?
6. What is UDF?
7. What is recursion?
8. Which function is used to allocate memory at runtime?
9. Which header file is required for pow()?
10. What type of value is returned by strcmp()?

🔹 Q3(B) SHORT THEORY (2 MARKS)

1. Explain strcpy() function.
2. Explain sqrt() function.
3. Explain recursion in brief.
4. Explain UDF.
5. Explain rand() function.
6. Explain strlen() function.
7. Explain function prototype.
8. Explain library function.
9. Explain call by value.
10. Explain call by reference.

🔹 Q3(C) LONG ANSWER (5 MARKS)

1. Explain user defined functions.
2. Explain types of user defined functions.
3. Explain recursion with example.
4. Difference between call by value and call by reference.
5. Explain dynamic memory allocation functions.
6. Explain string functions with example.
7. Write program using function.
8. Explain memory allocation functions.
9. Write program using recursion.
10. Explain function call mechanism.

Q4(A) VERY SHORT (3 MARKS)

1. What is pointer?
2. Which symbol is used to declare pointer variable?
3. What is initial value of array index?
4. How many subscripts required for 1D array?
5. How many types of arrays are there?
6. Pointer holds _____ as a value.
7. Which operator is used to get address of variable?
8. Which character terminates string?
9. What is string?

10. What is array?

Q4(B) SHORT THEORY (2 MARKS)

1. Explain pointer.
2. Explain array.
3. How to declare pointer variable?
4. Explain string.
5. Explain pointer to variable.
6. Explain array initialization.
7. Explain string array.
8. Explain pointer arithmetic.
9. Explain array of pointer.
10. Explain memory allocation of 2D array.

Q4(C) LONG ANSWER (5 MARKS)

1. Explain one dimensional array with example.
2. Explain two dimensional array with example.
3. Explain pointer to array with example.
4. Explain pointer to structure with example.
5. Explain array of structure.

6. Write program using array.
7. Write program using pointer.
8. Write program using 2D array.
9. Explain string array with example.
10. Explain dynamic memory allocation of array.

🔹 Q5(A) VERY SHORT (3 MARKS)

1. Which keyword is used to define structure?
2. Which operator is used to access structure member?
3. Structure is collection of _____ data types.
4. In union, each member holds _____ storage location.
5. What is structure?
6. What is union?
7. How many ways structure variable can be declared?
8. Which keyword is used to define union?
9. What is pointer to structure?
10. What is array of structure?

🔹 Q5(B) SHORT THEORY (2 MARKS)

1. Explain structure.

2. Explain union.
3. Explain nested structure.
4. Explain array within structure.
5. Explain pointer to structure.
6. Difference between structure and union.
7. Explain structure initialization.
8. Explain union initialization.
9. Explain structure with example.
10. Explain union with example.

🔹 Q5(C) LONG ANSWER (5 MARKS)

1. Explain structure in detail.
2. Explain union with example.
3. Difference between structure and union.
4. Explain memory allocation functions.
5. Write program using structure.
6. Write program using array of structure.
7. Write program using pointer to structure.
8. Explain dynamic memory allocation with example.
9. Write program to store student data using structure.
10. Explain nested structure with example.

Topper Question Bank

C Programming Topper Question Bank

🔹 SECTION Q1(A): VERY SHORT ANSWERS (1–2 Marks)

(Usually 5–6 questions asked, students attempt all)

1. Who developed C language?
2. ASCII stands for _____.
3. What is the range of `char` data type?
4. Which escape sequence character is used to print new line?
5. What is the other name of conditional operator?
6. Which tool is called graphical representation of program?
7. How many keywords are there in C?
8. What is the full form of `.c` file?
9. What is default value of static variable?
10. Array index starts from _____.
11. Which loop executes at least once?
12. Which operator is used to declare pointer variable?
13. Which character terminates a string?
14. How many values can be returned by a function?
15. Which keyword is used to define structure?
16. Which operator is used to access structure member?
17. Which header file is required for string functions?

18. Which statement is used to terminate a loop?
19. What does pointer store?
20. Which operator is known as indirection operator?

🔹 SECTION Q1(B): SHORT THEORY (2 Marks)

(Attempt any one or two)

1. Explain keyword in C.
2. Explain type casting with example.
3. Explain variable in brief.
4. Explain flowchart symbols.
5. Explain data types in C.
6. Explain character set in C.
7. Explain C tokens.
8. What is header file?
9. Explain static variable.
10. Explain sizeof operator.
11. Explain break statement.
12. Explain continue statement.
13. Explain goto statement.
14. Explain recursion.
15. Explain call by value.

16. Explain call by reference.
17. Explain string.
18. Explain pointer.
19. Explain structure.
20. Explain union.

🔹 SECTION Q1(C): LONG THEORY (5 Marks)

(Attempt any one)

1. Explain structure of C program.
2. Explain data types available in C with examples.
3. Explain different types of operators in C.
4. Explain variables in detail.
5. Explain flowchart with example.
6. Explain scope and lifetime of variable.
7. Explain storage classes in C.
8. Explain constants and its types.
9. Explain tokens in C programming.
10. Explain compilation process of C program.

🔹 SECTION Q2(A): SHORT QUESTIONS (3 Marks)

1. How many forms of if statement are there in C?
2. Which one is post-test loop?
3. Which jumping statement is also known as unequal jumping statement?
4. Which loop is entry controlled loop?
5. Which loop is exit controlled loop?
6. List out conditional statements in C.
7. List out entry controlled loops.
8. List out jumping statements.
9. Which loop requires semicolon at the end?
10. What is the value of case in switch statement?

🔹 SECTION Q2(B): THEORY / DIFFERENCE (2 Marks)

1. Explain if…else statement.
2. Explain for loop.
3. Difference between entry loop and exit loop.
4. Difference between while and do-while loop.
5. Explain break keyword.
6. Explain continue keyword.
7. Difference between break and continue.
8. Write syntax of nested if.
9. Write syntax of switch case.

10. Write syntax of for loop.

SECTION Q2(C): LONG ANSWER (5 Marks)

1. Explain if…else statement with example.
2. Explain for loop with example.
3. Explain while loop with example.
4. Explain do-while loop with example.
5. Explain switch case with example.
6. Explain break and continue statements.
7. Explain nested if statement.
8. Write a program using switch case.
9. Write a program using loop.
10. Write output of given loop program.

SECTION Q3(A): VERY SHORT (3 Marks)

1. Which header file is required for string functions?
2. What type of value is returned by strcmp()?
3. How many values can be returned by function?
4. What is default storage class in C?
5. Write syntax of calloc().

6. Which header file is required for pow()?
7. Which function is used to allocate memory at runtime?
8. Which data type is used when function returns no value?
9. What is UDF?
10. What is recursion?

SECTION Q3(B): SHORT THEORY (2 Marks)

1. Explain strcpy() function.
2. Explain sqrt() function.
3. Explain no argument with return value function.
4. Explain recursion in brief.
5. Explain rand() function.
6. Explain strlen() function.
7. Explain function prototype.
8. Explain library function.
9. Explain UDF.
10. Explain atoll() function.

SECTION Q3(C): LONG ANSWER (5 Marks)

1. Explain user defined functions.

2. Explain types of user defined functions.
3. Explain recursion with example.
4. Difference between call by value and call by reference.
5. Explain dynamic memory allocation functions.
6. Explain string functions with example.
7. Write program using function.
8. Explain memory allocation functions.
9. Explain function call mechanism.
10. Write program using recursion.

SECTION Q4(A): VERY SHORT (3 Marks)

1. What is pointer?
2. Which symbol is used to declare pointer variable?
3. What is initial value of array index?
4. How many subscripts required for 1D array?
5. How many types of arrays are there?
6. Array index holds which type of values?
7. Pointer holds _____ as a value.
8. What is string?
9. Which operator is used to get address of variable?
10. Which character terminates string?

🔹 SECTION Q4(B): SHORT THEORY (2 Marks)

1. Explain pointer.
2. Explain array.
3. Explain memory allocation of 2D array.
4. How to declare pointer variable?
5. Explain string.
6. Explain pointer to variable.
7. Explain array initialization.
8. Explain pointer arithmetic.
9. Explain string array.
10. Explain array of pointer.

🔹 SECTION Q4(C): LONG ANSWER (5 Marks)

1. Explain one dimensional array with example.
2. Explain two dimensional array with example.
3. Explain pointer to array with example.
4. Explain pointer to structure with example.
5. Explain array of structure.
6. Write program using array.
7. Write program using pointer.

8. Write program using 2D array.
9. Explain string array with example.
10. Explain dynamic memory allocation of array.

SECTION Q5(A): VERY SHORT (3 Marks)

1. Which keyword is used to define structure?
2. Which operator is used to access structure member?
3. In union, each member holds _____ storage location.
4. Structure is collection of _____ data types.
5. How to declare structure variable?
6. How many ways structure variable can be declared?
7. Which keyword is used to define union?
8. What is structure?
9. What is union?
10. What is pointer to structure?

SECTION Q5(B): SHORT THEORY (2 Marks)

1. Explain structure.
2. Explain union.
3. Explain nested structure.

4. Explain array within structure.
5. Explain pointer to structure.
6. Difference between structure and union.
7. Difference between array and structure.
8. Explain structure initialization.
9. Explain union initialization.
10. Explain structure with example.

🔹 SECTION Q5(C): LONG ANSWER (5 Marks)

1. Explain structure in detail.
2. Explain union with example.
3. Difference between structure and union.
4. Explain memory allocation functions.
5. Write program using structure.
6. Write program using array of structure.
7. Write program using pointer to structure.
8. Explain dynamic memory allocation with example.
9. Write program to store student data using structure.
10. Explain nested structure with example.