BCA SEMESTER 1 - PROGRAMMING IN C

Complete Study Material (Topic-Wise)

TABLE OF CONTENTS

UNIT 1: INTRODUCTION TO C PROGRAMMING

Chapter 1: Introduction to Programming

1.1 What is Programming?

Programming is the process of writing a set of instructions (called a **program**) that tells the computer what to do and how to do it.

1.2 What is a Program?

A **program** is a sequence of instructions written in a programming language to perform a specific task.

```
Problem  Algorithm  Flowchart  Program  Output
```

1.3 Algorithm

An **algorithm** is a step-by-step procedure to solve a problem.

Characteristics of a Good Algorithm:

1. **Input** – Takes zero or more inputs
2. **Output** – Produces at least one output
3. **Definiteness** – Each step is clear and unambiguous
4. **Finiteness** – Terminates after a finite number of steps
5. **Effectiveness** – Each step is basic and feasible

Example: Algorithm to add two numbers

```
Step 1: START
Step 2: Read two numbers A and B
Step 3: Calculate SUM = A + B
Step 4: Display SUM
Step 5: STOP
```

Example: Algorithm to find largest of three numbers

```
Step 1: START
Step 2: Read A, B, C
Step 3: If A > B and A > C
        Print "A is largest"
Step 4: Else if B > C
        Print "B is largest"
Step 5: Else
        Print "C is largest"
Step 6: STOP
```

1.4 Flowchart

A **flowchart** is a diagrammatic/pictorial representation of an algorithm.

Flowchart Symbols:

Symbol	Shape	Use
Terminal	Oval/Rounded Rectangle	START / STOP
Process	Rectangle	Calculations, assignments
Decision	Diamond	Conditions (Yes/No)
Input/Output	Parallelogram	Read / Display
Flow Lines	Arrows	Direction of flow
Connector	Circle	Connect different parts

Example Flowchart: Add two numbers

```
START

Read A,B

SUM = A + B

Print SUM

STOP
```

1.5 Pseudocode

Pseudocode is an informal, English-like description of an algorithm.

Example: Pseudocode to check if number is even or odd

```
BEGIN
   READ number
   IF number MOD 2 = 0 THEN
      PRINT "Even"
   ELSE
      PRINT "Odd"
   END IF
END
```

Chapter 2: History & Features of C

2.1 History of C

Year	Language	Developer	Description
1960	ALGOL	International Committee	Foundation language
1963	CPL	Cambridge University	Combined Programming Language
1967	BCPL	Martin Richards	Basic CPL (simplified)
1970	B	Ken Thompson	Simplified BCPL
1972	**C**	**Dennis Ritchie**	Developed at **AT&T Bell Labs**
1978	K&R C	Kernighan & Ritchie	Published "The C Programming Language" book
1989	ANSI C (C89)	ANSI Committee	Standardized C
1990	C90	ISO	International standard
1999	C99	ISO	Updated standard
2011	C11	ISO	Modern standard
2017	C17	ISO	Latest standard

Key Facts:

- C was developed by **Dennis Ritchie** in **1972** at **AT&T Bell Laboratories**, USA
- C evolved from **B** language (by Ken Thompson)
- **UNIX operating system** was rewritten in C
- C is called **"Mother Language"** of programming

- Book: **"The C Programming Language"** by Kernighan & Ritchie (1978) — also called **K&R C**

2.2 Why Learn C?

1. Foundation for other languages (C++, Java, Python)
2. Used in system programming (OS, compilers, drivers)
3. Efficient and fast execution
4. Close to hardware (low-level access)
5. Portable across platforms
6. Large community and resources
7. Used in embedded systems

2.3 Features of C Language

Features of C
- Simple & Easy to learn
- Structured Language (functions, blocks)
- Middle-Level Language (high + low level features)
- Portable (runs on different platforms)
- Rich Library (standard library functions)
- Fast Execution (compiled language)
- Extensible (can add custom functions)
- Pointer Support (memory access)
- Recursion Support
- Dynamic Memory Allocation
- Modular Approach (functions)
- Case Sensitive

Feature	Description
Simple	Easy to understand and write
Structured	Code divided into functions and blocks
Middle-Level	Combines features of high-level and low-level languages
Portable	Programs can run on different machines with little modification
Rich Library	Has built-in functions for common operations

Extensible	Users can add their own functions to the library
Fast	Compiled language → direct machine code → fast execution
Pointers	Direct memory access and manipulation
Recursion	Functions can call themselves
Case Sensitive	'A' and 'a' are different
Memory Management	Dynamic allocation using malloc(), calloc()
Modularity	Code organized into reusable functions

2.4 Applications of C

Application	Examples
Operating Systems	UNIX, Linux, Windows (partially)
Compilers	GCC, Turbo C
Databases	MySQL, Oracle
Embedded Systems	Microcontrollers, IoT devices
Gaming	Game engines
System Software	Device drivers, firmware
Network Programming	Network protocols
Scientific Computing	Simulations

Chapter 3: Structure of a C Program

3.1 General Structure

```c
/* Documentation Section */
// Comments about the program

/* Preprocessor Directives Section */
#include <stdio.h>     // Header files
#include <math.h>
#define PI 3.14159     // Macro definitions

/* Global Declaration Section */
int globalVar;         // Global variables
void function1();      // Function declarations
```

```c
/* Main Function */
int main()
{
    /* Local Declaration Section */
    int a, b;          // Local variables

    /* Executable Statements */
    printf("Hello World!");

    return 0;          // Return value
}

/* User-defined Functions */
void function1()
{
    // Function body
}
```

3.2 Sections Explained

Section	Description	Required?
Documentation	Comments describing the program	Optional
Preprocessor Directives	#include, #define	Yes (usually)
Global Declarations	Variables/functions accessible everywhere	Optional
main() Function	Entry point of program, execution starts here	**Mandatory**
Local Declarations	Variables declared inside functions	Optional
Executable Statements	Actual instructions to perform tasks	Yes
User-defined Functions	Functions created by programmer	Optional

3.3 First C Program

```c
/* Program: Hello World
   Author: Student
   Date: 2024 */
```

```c
#include <stdio.h>     // Standard Input/Output header

int main()           // Main function
{
    printf("Hello, World!\n");   // Print statement
    return 0;                    // Return 0 (success)
}
```

Output:

```
Hello, World!
```

Explanation Line by Line:

Line	Explanation
`/* ... */`	Multi-line comment (ignored by compiler)
`#include <stdio.h>`	Includes standard I/O library for printf, scanf
`int main()`	Main function — program execution starts here
`{`	Opening brace — start of function body
`printf("Hello, World!\n");`	Prints text to screen, \n = new line
`return 0;`	Returns 0 to OS (program ran successfully)
`}`	Closing brace — end of function body

3.4 Comments in C

Type	Syntax	Usage
Single-line	`// comment`	Comments on one line
Multi-line	`/* comment */`	Comments spanning multiple lines

```c
// This is a single-line comment

/* This is a
   multi-line
   comment */
```

3.5 Common Header Files

Header File	Functions Provided
stdio.h	printf(), scanf(), gets(), puts(), fopen(), fclose()
conio.h	clrscr(), getch(), kbhit() (Non-standard, Turbo C)
string.h	strlen(), strcpy(), strcat(), strcmp()
math.h	sqrt(), pow(), abs(), ceil(), floor(), sin(), cos()
stdlib.h	malloc(), calloc(), free(), exit(), atoi(), rand()
ctype.h	toupper(), tolower(), isalpha(), isdigit()
time.h	time(), clock(), difftime()

3.6 Escape Sequences

Escape Sequence	Meaning
\n	New line
\t	Horizontal tab
\\	Backslash
\"	Double quote
\'	Single quote
\0	Null character
\a	Alert (bell)
\b	Backspace
\r	Carriage return

```
printf("Name:\tJohn\n");     // Name:   John
printf("He said \"Hello\"\n"); // He said "Hello"
printf("C:\\Programs\n");     // C:\Programs
```

Chapter 4: Compilation & Execution Process

4.1 Steps in C Program Execution

```
Source Code (.c)
```

```
PREPROCESSOR       Handles #include, #define

    (Expanded Source Code)

 COMPILER          Converts to Assembly Code

    (Assembly Code .asm)

ASSEMBLER          Converts to Object Code

    (Object Code .obj/.o)

 LINKER            Links libraries & object files

    (Executable .exe/.out)

 LOADER            Loads into RAM for execution


OUTPUT
```

4.2 Each Stage Explained

Stage	Input	Output	Function
Preprocessor	Source code (.c)	Expanded code	Processes directives (#include, #define)
Compiler	Expanded code	Assembly code	Checks syntax, converts to assembly
Assembler	Assembly code	Object code (.obj)	Converts to machine code

Linker	Object code + Libraries	Executable (.exe)	Combines all object files and libraries
Loader	Executable	In memory	Loads program into RAM for execution

4.3 Types of Errors

Error Type	When Detected	Example
Syntax Error	Compilation	Missing semicolon, wrong keyword
Logical Error	Runtime	Wrong formula, infinite loop
Runtime Error	Execution	Division by zero, array out of bounds
Linker Error	Linking	Undefined function, missing library
Semantic Error	Compilation	Type mismatch, undeclared variable

4.4 C Compilers

Compiler	Platform
GCC (GNU Compiler Collection)	Linux, Mac, Windows
Turbo C/C++	DOS/Windows (old)
Dev-C++	Windows
Code::Blocks	Cross-platform
Visual Studio	Windows
Clang	Cross-platform
Online: Programiz, Replit, OnlineGDB	Web-based

Chapter 5: Variables, Constants & Keywords

5.1 Character Set of C

Category	Characters
Letters	A-Z, a-z
Digits	0-9
Special Characters	+ - * / % = & !
White Spaces	Space, Tab, New line, Form feed

5.2 Tokens in C

Tokens are the smallest individual units of a C program.

```
C Tokens
    Keywords (reserved words)
    Identifiers (variable/function names)
    Constants (fixed values)
    Strings ("Hello")
    Operators (+, -, *, /)
    Special Symbols ({, }, ;, #)
```

5.3 Keywords (Reserved Words)

C has **32 keywords** (ANSI C):

```
auto    double   int      struct
break   else     long     switch
case    enum     register typedef
char    extern   return   union
const   float    short    unsigned
continue for     signed   void
default goto     sizeof   volatile
do      if       static   while
```

Rules for Keywords:

- All keywords are in **lowercase**
- Keywords **cannot be used** as variable names
- Keywords have **predefined meanings**

5.4 Identifiers

Identifiers are names given to variables, functions, arrays, structures, etc.

Rules for Naming Identifiers:

1. Can contain **letters (A-Z, a-z)**, **digits (0-9)**, and **underscore (_)**
2. Must **begin with a letter or underscore** (not a digit)
3. **Cannot be a keyword**

4. **Case sensitive** (sum, Sum, SUM are different)
5. No **special characters** allowed (except underscore)
6. No **spaces** allowed
7. Can be of **any length** (but first 31 characters are significant)

Valid Identifiers:

```
sum, total_marks, _count, number1, MAX_SIZE, firstName
```

Invalid Identifiers:

```
2name       starts with digit
my name     contains space
int         keyword
my-name     contains hyphen
$price      contains $
```

5.5 Variables

A **variable** is a named memory location that holds a value which can change during program execution.

Syntax:

```
data_type variable_name;          // Declaration
data_type variable_name = value;  // Declaration + Initialization
```

Examples:

```
int age;            // Declaration
int age = 20;       // Declaration with initialization
float salary = 50000.50;
char grade = 'A';
int a, b, c;        // Multiple declaration
int x = 5, y = 10;  // Multiple declaration with initialization
```

Variable Naming Convention:

Style	Example	Use
Camel Case	`firstName`, `totalMarks`	Common in C
Snake Case	`first_name`, `total_marks`	Common in C
Pascal Case	`FirstName`, `TotalMarks`	Functions (sometimes)
UPPER CASE	`MAX_SIZE`, `PI`	Constants

5.6 Constants

A **constant** is a value that **cannot be changed** during program execution.

Types of Constants:

```
Constants
    Numeric Constants
        Integer Constants
            Decimal (base 10): 10, -25, 0
            Octal (base 8): 012, 077
            Hexadecimal (base 16): 0x1A, 0xFF
        Real (Floating-point) Constants
           Decimal form: 3.14, -0.5, 100.0
           Exponential form: 1.5e2 (=150), 3E-4

    Character Constants
        Single character: 'A', 'z', '5', '+'
        ASCII value used internally

    String Constants
       "Hello", "C Programming", "123"
```

Ways to Define Constants:

(A) Using `const` keyword:

```c
const float PI = 3.14159;
const int MAX = 100;
// PI = 3.14;    ERROR! Cannot modify
```

(B) Using `#define` preprocessor:

```
#define PI 3.14159
#define MAX 100
#define NAME "John"
```

const vs #define:

Feature	const	#define
Type	Has data type	No data type
Memory	Allocates memory	No memory allocated
Scope	Block scope	Global (till end of file)
Debugging	Easier	Harder
Semicolon	Required	Not required
Syntax	`const int x = 5;`	`#define X 5`

5.7 ASCII Values

ASCII = American Standard Code for Information Interchange

Character	ASCII Value
'0' - '9'	48 - 57
'A' - 'Z'	65 - 90
'a' - 'z'	97 - 122
Space	32
'\n' (newline)	10
'\0' (null)	0

```
printf("%d", 'A');    // Output: 65
printf("%c", 65);     // Output: A
printf("%d", '0');    // Output: 48
```

UNIT 2: DATA TYPES, OPERATORS & EXPRESSIONS

Chapter 6: Data Types in C

6.1 Classification

```
Data Types in C
    Primary (Built-in/Fundamental)
        int       (integer)
        float     (single precision decimal)
        double    (double precision decimal)
        char      (character)
        void      (empty/no value)

    Derived
        Array
        Pointer
        Function
        Reference

    User-defined
        Structure (struct)
        Union
        Enumeration (enum)
        typedef

    Modifiers
       short
       long
       signed
       unsigned
```

6.2 Primary Data Types with Size & Range

Data Type	Size (bytes)	Range	Format Specifier
`char`	1	-128 to 127	`%c`
`unsigned char`	1	0 to 255	`%c`
`short int`	2	-32,768 to 32,767	`%hd`
`unsigned short int`	2	0 to 65,535	`%hu`
`int`	2 or 4	-32,768 to 32,767 (2 bytes) or -2,147,483,648 to 2,147,483,647 (4 bytes)	`%d`
`unsigned int`	2 or 4	0 to 65,535 or 0 to 4,294,967,295	`%u`
`long int`	4	-2,147,483,648 to 2,147,483,647	`%ld`
`unsigned long int`	4	0 to 4,294,967,295	`%lu`
`long long int`	8	$-(2^{63})$ to $(2^{63} - 1)$	`%lld`
`float`	4	3.4E-38 to 3.4E+38	`%f`
`double`	8	1.7E-308 to 1.7E+308	`%lf`
`long double`	10/12/16	Extended precision	`%Lf`

6.3 Format Specifiers Summary

Specifier	Data Type	Example
`%d` or `%i`	int (signed integer)	`printf("%d", 10);`
`%u`	unsigned int	`printf("%u", 10);`
`%f`	float	`printf("%f", 3.14);`
`%lf`	double	`printf("%lf", 3.14);`
`%c`	char	`printf("%c", 'A');`
`%s`	string	`printf("%s", "Hello");`
`%x` or `%X`	hexadecimal	`printf("%x", 255);` ff
`%o`	octal	`printf("%o", 8);` 10
`%p`	pointer (address)	`printf("%p", &x);`
`%e` or `%E`	scientific notation	`printf("%e", 3.14);`
`%%`	literal %	`printf("100%%");`

6.4 sizeof Operator

Returns the size (in bytes) of a data type or variable.

```c
#include <stdio.h>
int main()
{
    printf("Size of char: %lu bytes\n", sizeof(char));      // 1
    printf("Size of int: %lu bytes\n", sizeof(int));        // 4
    printf("Size of float: %lu bytes\n", sizeof(float));    // 4
    printf("Size of double: %lu bytes\n", sizeof(double));  // 8
    printf("Size of long: %lu bytes\n", sizeof(long));      // 4 or 8

    int x = 10;
    printf("Size of x: %lu bytes\n", sizeof(x));            // 4

    return 0;
}
```

6.5 void Data Type

- Represents **"no value"** or **"empty"**
- Used in:
    1. **Function return type:** `void display()` function returns nothing
    2. **Function parameter:** `int main(void)` no parameters
    3. **Void pointer:** `void *ptr` generic pointer

Chapter 7: Operators in C

7.1 Classification of Operators

```
Operators in C
    Arithmetic Operators      (+, -, *, /, %)
    Relational Operators      (<, >, <=, >=, ==, !=)
    Logical Operators         (&&, ||, !)
    Bitwise Operators         (&, |, ^, ~, <<, >>)
    Assignment Operators      (=, +=, -=, *=, /=, %=)
    Increment/Decrement       (++, --)
```

```
Conditional (Ternary)      (?:)
Comma Operator             (,)
sizeof Operator            (sizeof)
Pointer Operators          (*, &)
Member Access Operators    (., ->)
Type Cast Operator         ((type))
```

7.2 Arithmetic Operators

Operator	Name	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`
`/`	Division	`5 / 3`	`1` (integer div)
`%`	Modulus (remainder)	`5 % 3`	`2`

Important Notes:

```c
// Integer division truncates decimal part
printf("%d", 5/3);      // Output: 1 (not 1.666)
printf("%f", 5.0/3);    // Output: 1.666667

// Modulus works only with integers
printf("%d", 10%3);     // Output: 1
// printf("%f", 10.5%3);    ERROR!
```

Complete Example:

```c
#include <stdio.h>
int main()
{
    int a = 17, b = 5;

    printf("a + b = %d\n", a + b);    // 22
    printf("a - b = %d\n", a - b);    // 12
    printf("a * b = %d\n", a * b);    // 85
    printf("a / b = %d\n", a / b);    // 3
```

```
    printf("a %% b = %d\n", a % b);   // 2

    return 0;
}
```

7.3 Relational (Comparison) Operators

Return **1 (true)** or **0 (false)**.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	1 (true)
!=	Not equal to	5 != 3	1 (true)
>	Greater than	5 > 3	1 (true)
<	Less than	5 < 3	0 (false)
>=	Greater than or equal	5 >= 5	1 (true)
<=	Less than or equal	5 <= 3	0 (false)

```
int a = 10, b = 20;
printf("%d\n", a == b);   // 0 (false)
printf("%d\n", a != b);   // 1 (true)
printf("%d\n", a > b);    // 0 (false)
printf("%d\n", a < b);    // 1 (true)
```

7.4 Logical Operators

Operator	Name	Description	Example	
&&	Logical AND	True if BOTH are true	(5>3) && (8>5)	1
||	Logical OR	True if ANY is true	(5>3) || (8<5)	1
!	Logical NOT	Reverses truth value	!(5>3)	0

Truth Table:

A	B	A && B	A || B	!A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

```c
int age = 25;
if (age >= 18 && age <= 60)
    printf("Working age\n");

int marks = 85;
if (marks > 90 || marks == 85)
    printf("Excellent\n");
```

7.5 Assignment Operators

Operator	Example	Equivalent
=	a = 10	a = 10
+=	a += 5	a = a + 5
-=	a -= 5	a = a - 5
*=	a *= 5	a = a * 5
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5
<<=	a <<= 2	a = a << 2
>>=	a >>= 2	a = a >> 2
&=	a &= 5	a = a & 5
|=	a |= 5	a = a | 5
^=	a ^= 5	a = a ^ 5

7.6 Increment and Decrement Operators

Operator	Name	Example	Description
++a	Pre-increment	b = ++a;	First increment a, then assign to b
a++	Post-increment	b = a++;	First assign to b, then increment a
--a	Pre-decrement	b = --a;	First decrement a, then assign to b
a--	Post-decrement	b = a--;	First assign to b, then decrement a

Detailed Example:

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

```
    int a = 5, b;

    // Pre-increment
    b = ++a;    // a becomes 6 first, then b = 6
    printf("a = %d, b = %d\n", a, b);  // a=6, b=6

    // Post-increment
    a = 5;
    b = a++;    // b = 5 first, then a becomes 6
    printf("a = %d, b = %d\n", a, b);  // a=6, b=5

    // Pre-decrement
    a = 5;
    b = --a;    // a becomes 4 first, then b = 4
    printf("a = %d, b = %d\n", a, b);  // a=4, b=4

    // Post-decrement
    a = 5;
    b = a--;    // b = 5 first, then a becomes 4
    printf("a = %d, b = %d\n", a, b);  // a=4, b=5

    return 0;
}
```

7.7 Bitwise Operators

Operator	Name	Description
`&`	Bitwise AND	1 if both bits are 1
`|`	Bitwise OR	1 if any bit is 1
`^`	Bitwise XOR	1 if bits are different
`~`	Bitwise NOT	Inverts all bits
`<<`	Left Shift	Shifts bits left (multiply by 2)
`>>`	Right Shift	Shifts bits right (divide by 2)

Example: a = 5 (0101), b = 3 (0011)

```
a    = 0 1 0 1  (5)
b    = 0 0 1 1  (3)
```

```
a & b = 0 0 0 1  (1)   AND
a | b = 0 1 1 1  (7)   OR
a ^ b = 0 1 1 0  (6)   XOR
~a    = 1 0 1 0  (-6)  NOT (in 2's complement)

a << 1 = 1 0 1 0 (10)  Left shift by 1 (5 × 2 = 10)
a >> 1 = 0 0 1 0 (2)   Right shift by 1 (5 / 2 = 2)
```

7.8 Conditional (Ternary) Operator

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
int a = 10, b = 20, max;
max = (a > b) ? a : b;    // max = 20
printf("Max = %d\n", max); // Max = 20

// Equivalent to:
if (a > b)
   max = a;
else
   max = b;
```

7.9 Comma Operator

Evaluates multiple expressions; returns the value of the last expression.

```
int a = (5, 10, 15);    // a = 15 (last value)
```

7.10 Operator Precedence and Associativity

(From Highest to Lowest Precedence)

Priority	Operator	Associativity
1	`()` `[]` `->` `.`	Left to Right
2	`++` `--` `!` `~` `+` `-` `*` `&` `sizeof` `(type)`	**Right to Left** (Unary)
3	`*` `/` `%`	Left to Right
4	`+` `-`	Left to Right
5	`<<` `>>`	Left to Right
6	`<` `<=` `>` `>=`	Left to Right
7	`==` `!=`	Left to Right
8	`&` (Bitwise AND)	Left to Right
9	`^` (Bitwise XOR)	Left to Right
10	`|` (Bitwise OR)	Left to Right
11	`&&` (Logical AND)	Left to Right
12	`||` (Logical OR)	Left to Right
13	`?:` (Ternary)	**Right to Left**
14	`=` `+=` `-=` `*=` `/=` `%=` etc.	**Right to Left**
15	`,` (Comma)	Left to Right

Memory Aid: "**PUMA'S REBL TAC**"

P - Parentheses, Postfix
U - Unary
M - Multiplication, Division, Modulus
A - Addition, Subtraction
S - Shift
R - Relational
E - Equality
B - Bitwise (AND, XOR, OR)
L - Logical (AND, OR)
T - Ternary
A - Assignment
C - Comma

Example:

```
int result = 2 + 3 * 4;   // result = 14 (not 20)
// * has higher precedence than +
```

```
// So: 2 + (3 * 4) = 2 + 12 = 14

int result2 = (2 + 3) * 4;  // result2 = 20
// Parentheses override precedence
```

Chapter 8: Expressions & Type Conversion

8.1 Expressions

An **expression** is a combination of variables, constants, and operators that produces a value.

```
// Arithmetic Expression
result = a + b * c - d / e;

// Relational Expression
result = (a > b);

// Logical Expression
result = (a > b) && (c < d);

// Assignment Expression
a = 10;
```

8.2 Type Conversion

(A) Implicit Type Conversion (Automatic/Coercion)

The compiler automatically converts a **lower data type** to a **higher data type**.

Hierarchy (Low High):

```
char  short  int  unsigned int  long  float  double  long double
```

```
int a = 10;
float b = 3.5;
```

```
float result = a + b;   // a is automatically converted to float
// result = 10.0 + 3.5 = 13.5

char ch = 'A';
int num = ch;           // ch (65) converted to int
printf("%d", num);      // Output: 65
```

(B) Explicit Type Conversion (Type Casting)

Programmer manually converts one data type to another.

Syntax:

```
(target_type) expression
```

```
int a = 5, b = 2;
float result;

result = a / b;         // Integer division: 2.000000
result = (float)a / b;  // Type casting: 2.500000

printf("Without cast: %f\n", (float)(a/b));    // 2.000000
printf("With cast: %f\n", (float)a / b);       // 2.500000

// Another example
float pi = 3.14;
int intPi = (int)pi;    // intPi = 3 (truncated)
```

Chapter 9: Input/Output Functions

9.1 Classification

```
I/O Functions
        Formatted I/O
            printf()      Formatted output
```

```
        scanf()      Formatted input

    Unformatted I/O
      Character I/O
        getchar()    Read single character
        putchar()    Write single character
        getch()      Read without echo
        getche()     Read with echo

      String I/O
        gets()       Read string (with spaces)
        puts()       Write string
```

9.2 printf() - Formatted Output

Syntax:

```
printf("format_string", variable1, variable2, ...);
```

Format Specifiers:

```
printf("Integer: %d\n", 10);        // Integer: 10
printf("Float: %f\n", 3.14);        // Float: 3.140000
printf("Float: %.2f\n", 3.14);      // Float: 3.14 (2 decimal places)
printf("Character: %c\n", 'A');     // Character: A
printf("String: %s\n", "Hello");    // String: Hello
printf("Octal: %o\n", 10);          // Octal: 12
printf("Hex: %x\n", 255);           // Hex: ff
printf("Address: %p\n", &x);        // Address: 0x7ffee...

// Width specification
printf("%5d\n", 42);               //    42 (right-aligned, width 5)
printf("%-5d|\n", 42);             // 42   | (left-aligned, width 5)
printf("%05d\n", 42);              // 00042 (zero-padded)
printf("%10.3f\n", 3.14159);       //      3.142 (width 10, 3 decimals)
```

9.3 scanf() - Formatted Input

Syntax:

```
scanf("format_string", &variable1, &variable2, ...);
```

Note: `&` (address-of operator) is required before variable names (except strings).

```
int age;
float salary;
char grade;
char name[50];

printf("Enter age: ");
scanf("%d", &age);              // Read integer

printf("Enter salary: ");
scanf("%f", &salary);           // Read float

printf("Enter grade: ");
scanf(" %c", &grade);           // Read character (space before %c skips whitespace)

printf("Enter name: ");
scanf("%s", name);              // Read string (no & for arrays)
                        // Note: stops at space
```

9.4 getchar() and putchar()

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

    printf("Enter a character: ");
    ch = getchar();         // Read single character

    printf("You entered: ");
    putchar(ch);            // Print single character
    putchar('\n');          // Print newline
```

```
    return 0;
}
```

9.5 gets() and puts()

```
#include <stdio.h>
int main()
{
    char name[100];

    printf("Enter your name: ");
    gets(name);          // Read entire line (with spaces)

    printf("Your name is: ");
    puts(name);          // Print string with automatic newline

    return 0;
}
```

Note: `gets()` is unsafe (no buffer overflow check). Use `fgets()` instead:

```
fgets(name, sizeof(name), stdin);  // Safer alternative
```

9.6 Complete I/O Example

```
#include <stdio.h>

int main()
{
    char name[50];
    int age;
    float marks;
    char grade;

    printf("Enter name: ");
    scanf("%s", name);
```

```
    printf("Enter age: ");
    scanf("%d", &age);

    printf("Enter marks: ");
    scanf("%f", &marks);

    printf("Enter grade: ");
    scanf(" %c", &grade);

    printf("\n--- Student Details ---\n");
    printf("Name: %s\n", name);
    printf("Age: %d\n", age);
    printf("Marks: %.2f\n", marks);
    printf("Grade: %c\n", grade);

    return 0;
}
```

UNIT 3: CONTROL STATEMENTS

Chapter 10: Decision Making Statements

10.1 Types of Decision Making

```
Decision Making
        if statement
        if-else statement
        if-else if-else ladder
        nested if-else
        switch statement
```

10.2 if Statement

Syntax:

```
if (condition)
{
   // Executed if condition is TRUE (non-zero)
}
```

```
int age = 20;
if (age >= 18)
{
   printf("You are an adult\n");
}
```

10.3 if-else Statement

Syntax:

```
if (condition)
{
   // Executed if condition is TRUE
}
else
{
   // Executed if condition is FALSE
}
```

Example: Check Even or Odd

```
#include <stdio.h>
int main()
{
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);

   if (num % 2 == 0)
      printf("%d is Even\n", num);
```

```
    else
        printf("%d is Odd\n", num);

    return 0;
}
```

10.4 if-else if-else Ladder

Syntax:

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

Example: Grade System

```
#include <stdio.h>
int main()
{
    int marks;
    printf("Enter marks: ");
    scanf("%d", &marks);

    if (marks >= 90)
        printf("Grade: A+\n");
```

```
    else if (marks >= 80)
        printf("Grade: A\n");
    else if (marks >= 70)
        printf("Grade: B\n");
    else if (marks >= 60)
        printf("Grade: C\n");
    else if (marks >= 50)
        printf("Grade: D\n");
    else
        printf("Grade: F (Fail)\n");

    return 0;
}
```

10.5 Nested if-else

```
if (condition1)
{
    if (condition2)
    {
        // Both conditions true
    }
    else
    {
        // condition1 true, condition2 false
    }
}
else
{
    // condition1 false
}
```

Example: Largest of Three Numbers

```
#include <stdio.h>
int main()
{
    int a, b, c;
```

```
    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    if (a >= b)
    {
        if (a >= c)
            printf("Largest = %d\n", a);
        else
            printf("Largest = %d\n", c);
    }
    else
    {
        if (b >= c)
            printf("Largest = %d\n", b);
        else
            printf("Largest = %d\n", c);
    }

    return 0;
}
```

10.6 switch Statement

Syntax:

```
switch (expression)
{
    case constant1:
        // statements
        break;
    case constant2:
        // statements
        break;
    case constant3:
        // statements
        break;
    default:
```

```
    // default statements
}
```

Rules for switch:

1. Expression must be **integer or character** type
2. Case values must be **constants** (not variables)
3. Case values must be **unique**
4. `break` prevents **fall-through** to next case
5. `default` is optional (executed when no case matches)
6. Floating-point values **not allowed** in case

Example: Simple Calculator

```
#include <stdio.h>
int main()
{
   float a, b, result;
   char op;

   printf("Enter expression (e.g., 5 + 3): ");
   scanf("%f %c %f", &a, &op, &b);

   switch (op)
   {
      case '+':
         result = a + b;
         printf("%.2f + %.2f = %.2f\n", a, b, result);
         break;
      case '-':
         result = a - b;
         printf("%.2f - %.2f = %.2f\n", a, b, result);
         break;
      case '*':
         result = a * b;
         printf("%.2f * %.2f = %.2f\n", a, b, result);
         break;
      case '/':
```

```
            if (b != 0)
            {
                result = a / b;
                printf("%.2f / %.2f = %.2f\n", a, b, result);
            }
            else
                printf("Error: Division by zero!\n");
            break;
        default:
            printf("Invalid operator!\n");
    }

    return 0;
}
```

Example: Day of Week

```
#include <stdio.h>
int main()
{
    int day;
    printf("Enter day number (1-7): ");
    scanf("%d", &day);

    switch (day)
    {
        case 1: printf("Monday\n"); break;
        case 2: printf("Tuesday\n"); break;
        case 3: printf("Wednesday\n"); break;
        case 4: printf("Thursday\n"); break;
        case 5: printf("Friday\n"); break;
        case 6: printf("Saturday\n"); break;
        case 7: printf("Sunday\n"); break;
        default: printf("Invalid day!\n");
    }

    return 0;
}
```

10.7 if-else vs switch

Feature	if-else	switch
Condition	Any expression	Integer/char expression
Range check	Yes (>, <, >=, etc.)	No (only exact match)
Float values	Allowed	Not allowed
Multiple conditions	Using && ||	Multiple cases
Fall-through	No	Yes (without break)
Speed	Slower (for many conditions)	Faster (jump table)

Chapter 11: Looping Statements

11.1 Types of Loops

```
Loops
    Entry-controlled (Pre-tested)
        for loop
        while loop

    Exit-controlled (Post-tested)
       do-while loop
```

11.2 for Loop

Syntax:

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

Flow:

```
initialization  condition check  (if true)  body  update  condition check  ...
                                 (if false)  exit loop
```

Example: Print 1 to 10

```c
#include <stdio.h>
int main()
{
    int i;
    for (i = 1; i <= 10; i++)
    {
        printf("%d ", i);
    }
    // Output: 1 2 3 4 5 6 7 8 9 10
    return 0;
}
```

Example: Sum of N Natural Numbers

```c
#include <stdio.h>
int main()
{
    int n, i, sum = 0;
    printf("Enter N: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
    {
        sum = sum + i;   // or sum += i;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Example: Factorial of a Number

```c
#include <stdio.h>
int main()
{
    int n, i;
```

```
    long long fact = 1;

    printf("Enter N: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
    {
        fact = fact * i;
    }

    printf("%d! = %lld\n", n, fact);
    return 0;
}
```

Example: Multiplication Table

```
#include <stdio.h>
int main()
{
    int num, i;
    printf("Enter a number: ");
    scanf("%d", &num);

    for (i = 1; i <= 10; i++)
    {
        printf("%d x %d = %d\n", num, i, num * i);
    }
    return 0;
}
```

11.3 while Loop

Syntax:

```
while (condition)
{
    // Loop body
```

```
    // Update statement
}
```

Example: Print 1 to 10

```
int i = 1;          // Initialization
while (i <= 10)     // Condition
{
    printf("%d ", i);
    i++;            // Update
}
```

Example: Reverse a Number

```
#include <stdio.h>
int main()
{
    int num, rev = 0, rem;
    printf("Enter a number: ");
    scanf("%d", &num);

    while (num != 0)
    {
        rem = num % 10;       // Get last digit
        rev = rev * 10 + rem;  // Build reversed number
        num = num / 10;       // Remove last digit
    }

    printf("Reversed: %d\n", rev);
    return 0;
}
```

Example: Count Digits of a Number

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

```
    int num, count = 0;
    printf("Enter a number: ");
    scanf("%d", &num);

    while (num != 0)
    {
        num = num / 10;
        count++;
    }

    printf("Number of digits: %d\n", count);
    return 0;
}
```

11.4 do-while Loop

Syntax:

```
do
{
    // Loop body
    // Update statement
} while (condition);    // Note: semicolon is required
```

Key difference: Body executes **at least once**, even if condition is false.

Example: Menu-Driven Program

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

    do
    {
        printf("\n--- Menu ---\n");
        printf("1. Add\n");
        printf("2. Subtract\n");
```

```
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1: printf("Addition selected\n"); break;
            case 2: printf("Subtraction selected\n"); break;
            case 3: printf("Exiting...\n"); break;
            default: printf("Invalid choice!\n");
        }
    } while (choice != 3);

    return 0;
}
```

11.5 Comparison of Loops

Feature	for	while	do-while
Type	Entry-controlled	Entry-controlled	Exit-controlled
Minimum execution	0 times	0 times	**1 time**
Initialization	In loop statement	Before loop	Before loop
Use when	Counter known	Condition-based	Execute at least once
Semicolon at end	No	No	**Yes**
Syntax	`for(i;c;u){}`	`while(c){}`	`do{}while(c);`

11.6 Nested Loops

Example: Pattern Printing (Right Triangle)

```
#include <stdio.h>
int main()
{
    int i, j, n = 5;

    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= i; j++)
```

```
        {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Output:

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

Example: Number Pattern

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

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Example: Inverted Triangle

```
for (i = 5; i >= 1; i--)
{
    for (j = 1; j <= i; j++)
    {
        printf("* ");
    }
    printf("\n");
}
```

Output:

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

11.7 Infinite Loops

```
// Infinite for loop
for (;;)
{
    printf("Infinite\n");
}

// Infinite while loop
while (1)
{
    printf("Infinite\n");
}

// Infinite do-while loop
do
{
    printf("Infinite\n");
} while (1);
```

Chapter 12: Jump Statements

12.1 Types

Statement	Purpose
`break`	Exits from loop or switch immediately
`continue`	Skips current iteration, goes to next
`goto`	Jumps to a labeled statement
`return`	Returns from a function

12.2 break Statement

```
// Exits the innermost loop or switch
for (i = 1; i <= 10; i++)
{
    if (i == 5)
        break;          // Exit loop when i = 5
    printf("%d ", i);
}
// Output: 1 2 3 4
```

12.3 continue Statement

```
// Skips current iteration
for (i = 1; i <= 10; i++)
{
    if (i == 5)
        continue;       // Skip when i = 5
    printf("%d ", i);
}
// Output: 1 2 3 4 6 7 8 9 10
```

12.4 goto Statement

```c
#include <stdio.h>
int main()
{
    int i = 1;

    loop:                   // Label
    if (i <= 5)
    {
        printf("%d ", i);
        i++;
        goto loop;          // Jump to label
    }

    printf("\nDone!\n");
    return 0;
}
// Output: 1 2 3 4 5
```

Note: `goto` is generally **discouraged** as it makes code harder to read and maintain (spaghetti code).

12.5 break vs continue

Feature	break	continue
Action	Exits loop entirely	Skips to next iteration
After execution	Control goes after loop	Control goes to loop condition
In switch	Exits switch block	Not used in switch

UNIT 4: FUNCTIONS

Chapter 13: Introduction to Functions

13.1 Definition

A **function** is a self-contained block of code that performs a specific task. It promotes **code reusability** and **modular programming**.

13.2 Advantages of Functions

1. **Code Reusability** — Write once, use many times
2. **Modularity** — Divide program into manageable parts
3. **Easy Debugging** — Isolate and fix errors in individual functions
4. **Readability** — Program is easier to understand
5. **Reduces Code Size** — Avoid repetition

13.3 Types of Functions

```
Functions
    Library Functions (Built-in/Predefined)
        printf(), scanf()       (stdio.h)
        strlen(), strcpy()      (string.h)
        sqrt(), pow()           (math.h)
        malloc(), free()        (stdlib.h)
        toupper(), tolower()    (ctype.h)

    User-defined Functions
       Created by programmer
```

13.4 Parts of a Function

```c
// 1. Function Declaration (Prototype)
int add(int a, int b);        // Before main()

// 2. Function Call
result = add(5, 3);           // Inside main() or another function

// 3. Function Definition
int add(int a, int b)         // Can be before or after main()
{
```

```
    return a + b;
}
```

Part	Description
Declaration (Prototype)	Tells compiler about function name, return type, parameters
Call	Invokes/executes the function
Definition	Contains the actual code of the function

13.5 Function Syntax

```
return_type function_name(parameter_list)
{
    // Local variable declarations
    // Statements
    return value;    // Optional (required for non-void functions)
}
```

13.6 Complete Example

```
#include <stdio.h>

// Function Declaration (Prototype)
int add(int, int);
int subtract(int, int);
float average(int, int);

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

    // Function Calls
    printf("Sum = %d\n", add(a, b));          // 30
    printf("Difference = %d\n", subtract(a, b)); // 10
    printf("Average = %.2f\n", average(a, b));   // 15.00

    return 0;
```

```
}

// Function Definitions
int add(int x, int y)
{
    return x + y;
}

int subtract(int x, int y)
{
    return x - y;
}

float average(int x, int y)
{
    return (float)(x + y) / 2;
}
```

13.7 Terminology

Term	Description
Actual Parameters	Values passed during function call `add(5, 3)`
Formal Parameters	Variables in function definition `int add(int a, int b)`
Return Type	Data type of value returned by function
Return Value	The value sent back to calling function
void	No return value / no parameters

Chapter 14: Function Types & Parameter Passing

14.1 Categories Based on Arguments and Return Value

Category	Arguments	Return Value	Example
Type 1	No arguments	No return	`void greet()`
Type 2	With arguments	No return	`void display(int n)`
Type 3	No arguments	With return	`int getInput()`
Type 4	With arguments	With return	`int add(int a, int b)`

Type 1: No arguments, No return

```
void greet()
{
    printf("Hello, Welcome!\n");
}

int main()
{
    greet();    // Call
    return 0;
}
```

Type 2: With arguments, No return

```
void displaySquare(int n)
{
    printf("Square of %d = %d\n", n, n * n);
}

int main()
{
    displaySquare(5);    // Output: Square of 5 = 25
    return 0;
}
```

Type 3: No arguments, With return

```
int getNumber()
{
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    return n;
}

int main()
{
```

```
    int num = getNumber();
    printf("You entered: %d\n", num);
    return 0;
}
```

Type 4: With arguments, With return

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int main()
{
    printf("Max = %d\n", max(10, 20));    // Max = 20
    return 0;
}
```

14.2 Parameter Passing Methods

(A) Call by Value

- **Copy** of actual parameter is passed
- Changes in function **do NOT affect** original variables
- **Default** method in C

```
#include <stdio.h>

void swap(int x, int y)    // Formal parameters (copies)
{
    int temp = x;
    x = y;
    y = temp;
    printf("Inside function: x=%d, y=%d\n", x, y);
}

int main()
```

```
{
    int a = 10, b = 20;
    printf("Before swap: a=%d, b=%d\n", a, b);

    swap(a, b);     // Call by value

    printf("After swap: a=%d, b=%d\n", a, b);
    return 0;
}
```

Output:

```
Before swap: a=10, b=20
Inside function: x=20, y=10
After swap: a=10, b=20          Original values UNCHANGED
```

(B) Call by Reference (Using Pointers)

- **Address** of actual parameter is passed
- Changes in function **DO affect** original variables

```
#include <stdio.h>

void swap(int *x, int *y)    // Pointers (addresses)
{
    int temp = *x;
    *x = *y;
    *y = temp;
    printf("Inside function: x=%d, y=%d\n", *x, *y);
}

int main()
{
    int a = 10, b = 20;
    printf("Before swap: a=%d, b=%d\n", a, b);

    swap(&a, &b);     // Pass addresses

```

```
    printf("After swap: a=%d, b=%d\n", a, b);
    return 0;
}
```

Output:

```
Before swap: a=10, b=20
Inside function: x=20, y=10
After swap: a=20, b=10          Original values CHANGED
```

14.3 Call by Value vs Call by Reference

Feature	Call by Value	Call by Reference
Passed	Copy of value	Address of variable
Original	Not modified	Modified
Memory	More (two copies)	Less (uses pointers)
Syntax	`func(a)`	`func(&a)`
Parameter	`int x`	`int *x`
Safety	Safer	Risk of unintended changes

Chapter 15: Recursion

15.1 Definition

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

15.2 Parts of Recursive Function

```
return_type function(parameters)
{
    // 1. Base Case (Termination Condition)
    if (base_condition)
        return base_value;

    // 2. Recursive Case
```

```
    return function(modified_parameters);   // Self call
}
```

15.3 Example: Factorial

$$n! = n \times (n-1) \times (n-2) \times ... \times 1$$ $$n! = n \times (n-1)!$$ $$0! = 1 \text{ (base case)}$$

```
#include <stdio.h>

int factorial(int n)
{
    // Base case
    if (n == 0 || n == 1)
        return 1;

    // Recursive case
    return n * factorial(n - 1);
}

int main()
{
    int n = 5;
    printf("%d! = %d\n", n, factorial(n));  // 5! = 120
    return 0;
}
```

How it works:

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120
```

15.4 Example: Fibonacci Series

$$F(0) = 0, F(1) = 1$$ $$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

```c
#include <stdio.h>

int fibonacci(int n)
{
    if (n == 0) return 0;      // Base case
    if (n == 1) return 1;      // Base case

    return fibonacci(n-1) + fibonacci(n-2);  // Recursive case
}

int main()
{
    int i;
    printf("Fibonacci Series: ");
    for (i = 0; i < 10; i++)
        printf("%d ", fibonacci(i));
    // Output: 0 1 1 2 3 5 8 13 21 34
    return 0;
}
```

15.5 Example: Power Function

$$x^n = x \times x^{n-1}$$ $$x^0 = 1$$

```c
int power(int base, int exp)
{
    if (exp == 0) return 1;
    return base * power(base, exp - 1);
}
// power(2, 4) = 2 * 2 * 2 * 2 = 16
```

15.6 Example: Sum of Digits

```
int sumOfDigits(int n)
{
    if (n == 0) return 0;
    return (n % 10) + sumOfDigits(n / 10);
}
// sumOfDigits(123) = 3 + 2 + 1 = 6
```

15.7 Example: GCD (Euclidean Algorithm)

```
int gcd(int a, int b)
{
    if (b == 0) return a;
    return gcd(b, a % b);
}
// gcd(12, 8)    gcd(8, 4)    gcd(4, 0)    4
```

15.8 Example: Tower of Hanoi

```
#include <stdio.h>

void towerOfHanoi(int n, char from, char to, char aux)
{
    if (n == 1)
    {
        printf("Move disk 1 from %c to %c\n", from, to);
        return;
    }

    towerOfHanoi(n-1, from, aux, to);
    printf("Move disk %d from %c to %c\n", n, from, to);
    towerOfHanoi(n-1, aux, to, from);
}

int main()
{
    towerOfHanoi(3, 'A', 'C', 'B');
```

```
    return 0;
}
```

15.9 Recursion vs Iteration

Feature	Recursion	Iteration
Definition	Function calls itself	Uses loops
Termination	Base case	Loop condition
Memory	More (stack frames)	Less
Speed	Slower (function overhead)	Faster
Code	Shorter, elegant	Longer
Stack Overflow	Possible	Not possible
Readability	Better for some problems	Better for simple problems
Best For	Tree, graph problems	Simple repetitive tasks

Chapter 16: Storage Classes

16.1 Definition

Storage classes determine the **scope**, **lifetime**, **visibility**, and **default value** of a variable.

16.2 Four Storage Classes in C

Storage Class	Keyword	Scope	Lifetime	Default Value	Storage
Automatic	`auto`	Local (within block)	Until block ends	Garbage	Stack
Register	`register`	Local (within block)	Until block ends	Garbage	CPU Register
Static	`static`	Local (within block)	Entire program	0	Memory
External	`extern`	Global (all files)	Entire program	0	Memory

16.3 auto (Automatic)

```
void func()
{
    auto int x = 10;    // Same as: int x = 10;
    // 'auto' is default, rarely written explicitly
}
```

16.4 register

```
void func()
{
    register int i;    // Request to store in CPU register
    for (i = 0; i < 1000; i++)
    {
        // Fast access (if granted by compiler)
    }
    // Cannot use & (address-of) with register variables
}
```

16.5 static

```
#include <stdio.h>

void counter()
{
    static int count = 0;    // Initialized only once
    count++;
    printf("Count = %d\n", count);
}

int main()
{
    counter();    // Count = 1
    counter();    // Count = 2
    counter();    // Count = 3
    // Static variable retains value between function calls
```

```
    return 0;
}
```

16.6 extern

```
// File1.c
int globalVar = 100;    // Definition

// File2.c
extern int globalVar;   // Declaration (use variable from File1.c)
// Now globalVar can be used in File2.c
```

UNIT 5: ARRAYS & STRINGS

Chapter 17: One-Dimensional Arrays

17.1 Definition

An **array** is a collection of elements of the **same data type** stored in **contiguous memory** locations, accessed using an **index**.

17.2 Declaration and Initialization

```
// Declaration
data_type array_name[size];

// Examples
int marks[5];          // 5 integers
float prices[10];      // 10 floats
char name[20];         // 20 characters

// Initialization at declaration
int marks[5] = {85, 90, 78, 92, 88};
```

```
// Partial initialization (rest become 0)
int arr[5] = {1, 2};       // arr = {1, 2, 0, 0, 0}

// Size auto-determined
int arr[] = {10, 20, 30};   // Size = 3

// All elements to 0
int arr[5] = {0};          // arr = {0, 0, 0, 0, 0}
```

17.3 Accessing Array Elements

Index: 0 1 2 3 4

marks: 85 90 78 92 88

Address: 1000 1004 1008 1012 1016 (assuming 4 bytes per int)

- **Index starts from 0** (not 1)
- **Last index = size - 1**
- `marks[0]` = 85, `marks[4]` = 88

17.4 Input and Output of Array

```
#include <stdio.h>
int main()
{
    int arr[5], i;

    // Input
    printf("Enter 5 numbers:\n");
    for (i = 0; i < 5; i++)
    {
        printf("arr[%d] = ", i);
        scanf("%d", &arr[i]);
    }

    // Output
```

```
    printf("\nArray elements:\n");
    for (i = 0; i < 5; i++)
    {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    return 0;
}
```

17.5 Common Array Operations

Example: Find Sum and Average

```
int arr[] = {10, 20, 30, 40, 50};
int n = 5, sum = 0;
float avg;

for (int i = 0; i < n; i++)
    sum += arr[i];

avg = (float)sum / n;
printf("Sum = %d, Average = %.2f\n", sum, avg);
```

Example: Find Largest and Smallest

```
int arr[] = {45, 12, 78, 34, 90, 23};
int n = 6;
int max = arr[0], min = arr[0];

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

printf("Largest = %d, Smallest = %d\n", max, min);
```

Example: Linear Search

```
int arr[] = {10, 25, 30, 45, 50};
int n = 5, key = 30, found = 0;

for (int i = 0; i < n; i++)
{
    if (arr[i] == key)
    {
        printf("Found at index %d\n", i);
        found = 1;
        break;
    }
}
if (!found)
    printf("Not found\n");
```

Example: Bubble Sort

```
#include <stdio.h>
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = 7, i, j, temp;

    // Bubble Sort
    for (i = 0; i < n-1; i++)
    {
        for (j = 0; j < n-i-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    printf("Sorted array: ");
```

```
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    // Output: 11 12 22 25 34 64 90

    return 0;
}
```

Example: Reverse an Array

```
int arr[] = {1, 2, 3, 4, 5};
int n = 5, temp;

for (int i = 0; i < n/2; i++)
{
    temp = arr[i];
    arr[i] = arr[n-1-i];
    arr[n-1-i] = temp;
}
// Result: {5, 4, 3, 2, 1}
```

17.6 Passing Arrays to Functions

```
#include <stdio.h>

// Arrays are always passed by reference
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int findSum(int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];
    return sum;
```

```
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);  // Calculate size

    printArray(arr, n);
    printf("Sum = %d\n", findSum(arr, n));

    return 0;
}
```

Chapter 18: Multi-Dimensional Arrays

18.1 Two-Dimensional Arrays (2D)

A 2D array is like a **matrix** with rows and columns.

Declaration:

```
data_type array_name[rows][columns];
```

Initialization:

```
int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Or in single line
int matrix[2][3] = {1, 2, 3, 4, 5, 6};
```

Memory Representation:

```
matrix[3][3]:
        Col 0  Col 1  Col 2
Row 0  [  1  ][  2  ][  3  ]
Row 1  [  4  ][  5  ][  6  ]
Row 2  [  7  ][  8  ][  9  ]
```

18.2 Input/Output of 2D Array

```c
#include <stdio.h>
int main()
{
    int matrix[3][3], i, j;

    // Input
    printf("Enter 3x3 matrix:\n");
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            scanf("%d", &matrix[i][j]);

    // Output
    printf("\nMatrix:\n");
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            printf("%4d", matrix[i][j]);
        printf("\n");
    }

    return 0;
}
```

18.3 Matrix Operations

Matrix Addition:

```c
for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
```

```
        C[i][j] = A[i][j] + B[i][j];
```

Matrix Multiplication:

```
for (i = 0; i < r1; i++)
   for (j = 0; j < c2; j++)
   {
      C[i][j] = 0;
      for (k = 0; k < c1; k++)
         C[i][j] += A[i][k] * B[k][j];
   }
```

Matrix Transpose:

```
for (i = 0; i < rows; i++)
   for (j = 0; j < cols; j++)
      T[j][i] = A[i][j];
```

Chapter 19: Strings in C

19.1 Definition

A **string** is an array of characters terminated by a **null character** (`'\0'`).

19.2 Declaration and Initialization

```
// Method 1: Character array
char str[10] = "Hello";
// Memory: H e l l o \0 ? ? ? ?

// Method 2: Character by character
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

// Method 3: Auto size
char str[] = "Hello";    // Size = 6 (5 chars + \0)
```

```
// Method 4: Using pointer
char *str = "Hello";     // String literal (read-only)
```

19.3 String Input/Output

```
#include <stdio.h>

int main()
{
    char name[50];

    // Method 1: scanf (reads until space)
    printf("Enter name: ");
    scanf("%s", name);              // "John" only (stops at space)
    printf("Name: %s\n", name);

    // Method 2: gets (reads entire line) — UNSAFE
    printf("Enter full name: ");
    gets(name);                     // "John Smith"
    puts(name);                     // Prints with newline

    // Method 3: fgets (safe, reads entire line)
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);
    printf("Name: %s", name);

    return 0;
}
```

19.4 String Examples

Find Length of String (without strlen)

```
int length = 0;
char str[] = "Hello";

while (str[length] != '\0')
    length++;
```

```
printf("Length = %d\n", length);  // 5
```

Copy String (without strcpy)

```
char src[] = "Hello";
char dest[20];
int i = 0;

while (src[i] != '\0')
{
    dest[i] = src[i];
    i++;
}
dest[i] = '\0';
```

Compare Strings (without strcmp)

```
char s1[] = "Hello", s2[] = "Hello";
int i = 0, equal = 1;

while (s1[i] != '\0' && s2[i] != '\0')
{
    if (s1[i] != s2[i])
    {
        equal = 0;
        break;
    }
    i++;
}

if (s1[i] != s2[i]) equal = 0;

if (equal)
    printf("Strings are equal\n");
else
    printf("Strings are not equal\n");
```

Reverse a String

```
char str[] = "Hello";
int len = strlen(str);
char temp;

for (int i = 0; i < len/2; i++)
{
    temp = str[i];
    str[i] = str[len-1-i];
    str[len-1-i] = temp;
}
printf("Reversed: %s\n", str);  // olleH
```

Check Palindrome

```
char str[] = "madam";
int len = strlen(str);
int isPalindrome = 1;

for (int i = 0; i < len/2; i++)
{
    if (str[i] != str[len-1-i])
    {
        isPalindrome = 0;
        break;
    }
}

if (isPalindrome)
    printf("Palindrome\n");
else
    printf("Not Palindrome\n");
```

Count Vowels and Consonants

```
char str[] = "Hello World";
int vowels = 0, consonants = 0;
```

```
for (int i = 0; str[i] != '\0'; i++)
{
    char ch = tolower(str[i]);
    if (ch >= 'a' && ch <= 'z')
    {
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')
            vowels++;
        else
            consonants++;
    }
}
printf("Vowels: %d, Consonants: %d\n", vowels, consonants);
```

Chapter 20: String Functions (string.h)

20.1 Common String Functions

Function	Purpose	Syntax
strlen()	Returns length of string	int len = strlen(str);
strcpy()	Copies one string to another	strcpy(dest, src);
strncpy()	Copies n characters	strncpy(dest, src, n);
strcat()	Concatenates (joins) strings	strcat(dest, src);
strncat()	Concatenates n characters	strncat(dest, src, n);
strcmp()	Compares two strings	int result = strcmp(s1, s2);
strncmp()	Compares n characters	int result = strncmp(s1, s2, n);
strchr()	Finds first occurrence of char	char *p = strchr(str, 'a');
strrchr()	Finds last occurrence of char	char *p = strrchr(str, 'a');
strstr()	Finds first occurrence of substring	char *p = strstr(str, "sub");
strtok()	Tokenizes string	char *token = strtok(str, " ");
strrev()	Reverses string (non-standard)	strrev(str);
strlwr()	Converts to lowercase (non-standard)	strlwr(str);
strupr()	Converts to uppercase (non-standard)	strupr(str);

20.2 Examples

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

int main()
{
    char s1[50] = "Hello";
    char s2[50] = "World";
    char s3[50];

    // strlen
    printf("Length of s1: %lu\n", strlen(s1));    // 5

    // strcpy
    strcpy(s3, s1);
    printf("s3 after strcpy: %s\n", s3);         // Hello

    // strcat
    strcat(s1, " ");
    strcat(s1, s2);
    printf("After strcat: %s\n", s1);            // Hello World

    // strcmp
    int result = strcmp("abc", "abd");
    if (result == 0)
        printf("Equal\n");
    else if (result < 0)
        printf("First is less\n");               // This prints
    else
        printf("First is greater\n");

    return 0;
}
```

UNIT 6: POINTERS

Chapter 21: Introduction to Pointers

21.1 Definition

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

21.2 Declaration and Initialization

```
data_type *pointer_name;

int a = 10;
int *ptr;       // Declaration
ptr = &a;       // Initialization (stores address of a)

// Combined
int *ptr = &a;
```

21.3 Pointer Operators

Operator	Name	Purpose
`&`	Address-of	Returns address of variable
`*`	Dereference/Indirection	Returns value at address

```
int a = 10;
int *ptr = &a;

printf("Value of a: %d\n", a);        // 10
printf("Address of a: %p\n", &a);     // 0x7fff5fbff8ac
printf("Value of ptr: %p\n", ptr);    // 0x7fff5fbff8ac (same)
printf("Value at ptr: %d\n", *ptr);   // 10
```

21.4 Pointer Visualization

```
Variable a:

    10        Value
```

```
Address: 1000

Pointer ptr:

  1000       Stores address of a

Address: 2000

&a = 1000 (address of a)
*ptr = 10 (value at address stored in ptr)
ptr = 1000 (value of ptr = address of a)
&ptr = 2000 (address of ptr itself)
```

21.5 Complete Pointer Example

```c
#include <stdio.h>

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

    printf("a = %d\n", a);       // 10
    printf("&a = %p\n", &a);     // address
    printf("p = %p\n", p);       // same address
    printf("*p = %d\n", *p);     // 10
    printf("&p = %p\n", &p);     // address of pointer

    *p = 20;    // Change value through pointer
    printf("\nAfter *p = 20:\n");
    printf("a = %d\n", a);       // 20 (changed!)
    printf("*p = %d\n", *p);     // 20

    return 0;
}
```

21.6 NULL Pointer

```c
int *ptr = NULL;    // Points to nothing (address 0)

if (ptr == NULL)
    printf("Pointer is NULL\n");
```

21.7 Pointer Arithmetic

```c
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;       // Points to first element

printf("%d\n", *ptr);      // 10
printf("%d\n", *(ptr+1));  // 20
printf("%d\n", *(ptr+2));  // 30

ptr++;                     // Move to next element
printf("%d\n", *ptr);      // 20

ptr += 2;
printf("%d\n", *ptr);      // 40
```

21.8 Pointer to Pointer (Double Pointer)

```c
int a = 10;
int *p = &a;
int **pp = &p;

printf("a = %d\n", a);      // 10
printf("*p = %d\n", *p);    // 10
printf("**pp = %d\n", **pp); // 10
```

Chapter 22: Pointers with Arrays & Functions

22.1 Pointers and Arrays

Array name is a **pointer to the first element**.

```c
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;    // Same as: int *ptr = &arr[0];

// Accessing elements
printf("%d\n", arr[0]);     // 10
printf("%d\n", *ptr);       // 10
printf("%d\n", *(ptr+2));   // 30
printf("%d\n", ptr[3]);     // 40  (pointer can use [] too)

// arr[i] is equivalent to *(arr + i)
// &arr[i] is equivalent to (arr + i)
```

22.2 Pointers and Functions

```c
#include <stdio.h>

// Swap using pointers (Call by Reference)
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Return pointer
int* findMax(int *a, int *b)
{
    return (*a > *b) ? a : b;
}

int main()
{
    int x = 10, y = 20;

    swap(&x, &y);
    printf("x=%d, y=%d\n", x, y);  // x=20, y=10

    int *maxPtr = findMax(&x, &y);
```

```
    printf("Max = %d\n", *maxPtr);

    return 0;
}
```

Chapter 23: Dynamic Memory Allocation

23.1 Functions (stdlib.h)

Function	Purpose	Syntax
malloc()	Allocates memory block	ptr = (type*)malloc(size);
calloc()	Allocates and initializes to 0	ptr = (type*)calloc(n, size);
realloc()	Resizes allocated memory	ptr = (type*)realloc(ptr, new_size);
free()	Frees allocated memory	free(ptr);

23.2 malloc()

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

int main()
{
    int n, i;
    printf("Enter size: ");
    scanf("%d", &n);

    // Allocate memory for n integers
    int *arr = (int*)malloc(n * sizeof(int));

    if (arr == NULL)
    {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Input
```

```
    for (i = 0; i < n; i++)
    {
        printf("arr[%d] = ", i);
        scanf("%d", &arr[i]);
    }

    // Output
    printf("Array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    // Free memory
    free(arr);

    return 0;
}
```

23.3 calloc()

```
// calloc initializes all elements to 0
int *arr = (int*)calloc(5, sizeof(int));
// arr = {0, 0, 0, 0, 0}
```

23.4 malloc() vs calloc()

Feature	malloc()	calloc()
Parameters	malloc(total_bytes)	calloc(num_elements, element_size)
Initialization	Garbage values	All zeros
Speed	Slightly faster	Slightly slower
Syntax	malloc(5 * sizeof(int))	calloc(5, sizeof(int))

23.5 realloc()

```
int *arr = (int*)malloc(5 * sizeof(int));
// ... use 5 elements ...
```

```
// Resize to 10 elements
arr = (int*)realloc(arr, 10 * sizeof(int));
```

UNIT 7: STRUCTURES & UNIONS

Chapter 24: Structures

24.1 Definition

A **structure** is a user-defined data type that groups **related variables of different data types** under a single name.

24.2 Declaration

```
struct structure_name
{
   data_type member1;
   data_type member2;
   // ...
};
```

24.3 Example

```
#include <stdio.h>

// Structure declaration
struct Student
{
   int rollNo;
   char name[50];
   float marks;
   char grade;
};
```

```c
int main()
{
    // Structure variable declaration
    struct Student s1;

    // Input
    printf("Enter Roll No: ");
    scanf("%d", &s1.rollNo);

    printf("Enter Name: ");
    scanf("%s", s1.name);

    printf("Enter Marks: ");
    scanf("%f", &s1.marks);

    printf("Enter Grade: ");
    scanf(" %c", &s1.grade);

    // Output
    printf("\n--- Student Details ---\n");
    printf("Roll No: %d\n", s1.rollNo);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);
    printf("Grade: %c\n", s1.grade);

    return 0;
}
```

24.4 Initialization

```c
// Method 1: Individual assignment
struct Student s1;
s1.rollNo = 101;
strcpy(s1.name, "John");
s1.marks = 85.5;

// Method 2: At declaration
struct Student s1 = {101, "John", 85.5, 'A'};
```

```c
// Method 3: Designated initializers (C99)
struct Student s1 = {.name = "John", .rollNo = 101, .marks = 85.5};
```

24.5 Array of Structures

```c
struct Student students[3] = {
    {101, "Alice", 90.5, 'A'},
    {102, "Bob", 78.0, 'B'},
    {103, "Charlie", 85.5, 'A'}
};

// Access
for (int i = 0; i < 3; i++)
{
    printf("%d %s %.1f %c\n",
        students[i].rollNo,
        students[i].name,
        students[i].marks,
        students[i].grade);
}
```

24.6 Structure with Functions

```c
// Passing structure to function
void display(struct Student s)
{
    printf("Name: %s, Marks: %.2f\n", s.name, s.marks);
}

// Returning structure from function
struct Student createStudent(int roll, char name[], float marks)
{
    struct Student s;
    s.rollNo = roll;
    strcpy(s.name, name);
    s.marks = marks;
```

```
    return s;
}
```

24.7 Pointer to Structure

```
struct Student s1 = {101, "John", 85.5};
struct Student *ptr = &s1;

// Access using arrow operator
printf("Name: %s\n", ptr->name);       // Arrow operator
printf("Marks: %.2f\n", ptr->marks);

// Equivalent to:
printf("Name: %s\n", (*ptr).name);     // Dot operator with dereference
```

24.8 Nested Structures

```
struct Date
{
    int day, month, year;
};

struct Student
{
    char name[50];
    struct Date dob;    // Nested structure
};

struct Student s1 = {"John", {15, 6, 2000}};
printf("DOB: %d/%d/%d\n", s1.dob.day, s1.dob.month, s1.dob.year);
```

Chapter 25: Unions

25.1 Definition

A **union** is similar to a structure, but all members **share the same memory location**. Size of union = size of largest member.

25.2 Syntax

```
union union_name
{
    data_type member1;
    data_type member2;
    // ...
};
```

25.3 Example

```
#include <stdio.h>

union Data
{
    int i;
    float f;
    char c;
};

int main()
{
    union Data d;

    printf("Size of union: %lu\n", sizeof(d));  // 4 (size of float)

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

    d.f = 3.14;
    printf("d.f = %.2f\n", d.f);  // 3.14
    printf("d.i = %d\n", d.i);    // Garbage! (overwritten by d.f)

    d.c = 'A';
    printf("d.c = %c\n", d.c);    // A
```

```
    printf("d.f = %.2f\n", d.f);  // Garbage! (overwritten by d.c)

    // Only ONE member can hold valid value at a time

    return 0;
}
```

25.4 Structure vs Union

Feature	Structure	Union
Memory	Sum of all members	Size of largest member
Access	All members simultaneously	One member at a time
Keyword	`struct`	`union`
Example	`struct{int i; float f; char c;}`	`union{int i; float f; char c;}`
Size	4+4+1 = 12 bytes*	4 bytes

(*with padding/alignment)

Chapter 26: Enumerations & typedef

26.1 enum (Enumeration)

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
//       0   1   2   3   4   5   6

enum Day today = WED;
printf("Today = %d\n", today);  // 2

// Custom values
enum Color {RED = 1, GREEN = 5, BLUE = 10};
```

26.2 typedef

Creates an **alias** (new name) for existing data types.

```
typedef unsigned long int ULONG;
typedef int* IntPtr;
typedef struct Student Student;  // No need to write 'struct' every time

ULONG population = 1400000000;
IntPtr ptr;                    // Same as: int *ptr;

// With structure
typedef struct
{
    char name[50];
    int age;
} Person;

Person p1 = {"John", 25};      // No need for 'struct' keyword
```

UNIT 8: FILE HANDLING

Chapter 27: File Operations in C

27.1 Why File Handling?

- **Permanent storage** of data (RAM is temporary)
- **Large data** cannot be handled through console I/O
- **Share data** between programs

27.2 File Operations

```
File Operations
        Creating a file
        Opening a file
        Reading from a file
        Writing to a file
```

```
Appending to a file
Closing a file
```

27.3 File Pointer

```
FILE *fp;    // File pointer declaration
```

27.4 fopen() - Opening a File

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

File Opening Modes:

Mode	Description	If file exists	If file doesn't exist
"r"	Read only	Opens	Returns NULL
"w"	Write only	Overwrites	Creates new
"a"	Append	Appends at end	Creates new
"r+"	Read + Write	Opens	Returns NULL
"w+"	Write + Read	Overwrites	Creates new
"a+"	Append + Read	Appends at end	Creates new
"rb"	Read binary	Opens	Returns NULL
"wb"	Write binary	Overwrites	Creates new

27.5 fclose() - Closing a File

```
fclose(fp);    // Always close files after use
```

27.6 Complete File Handling Examples

Writing to a File:

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

```
    FILE *fp;

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

    if (fp == NULL)
    {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fp, "Hello, World!\n");
    fprintf(fp, "This is line 2.\n");
    fprintf(fp, "Value: %d\n", 42);

    fclose(fp);
    printf("File written successfully!\n");

    return 0;
}
```

Reading from a File:

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char line[100];

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

    if (fp == NULL)
    {
        printf("Error opening file!\n");
        return 1;
    }

    // Method 1: Read line by line
    while (fgets(line, sizeof(line), fp) != NULL)
    {
```

```
        printf("%s", line);
    }

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

Appending to a File:

```
FILE *fp = fopen("output.txt", "a");
fprintf(fp, "This line is appended.\n");
fclose(fp);
```

Chapter 28: File I/O Functions

28.1 Character I/O

Function	Purpose	Syntax
`fgetc()`	Read single character	`ch = fgetc(fp);`
`fputc()`	Write single character	`fputc(ch, fp);`

```
// Copy file character by character
FILE *src = fopen("input.txt", "r");
FILE *dest = fopen("copy.txt", "w");
char ch;

while ((ch = fgetc(src)) != EOF)
{
    fputc(ch, dest);
}

fclose(src);
fclose(dest);
```

28.2 String I/O

Function	Purpose	Syntax
fgets()	Read string/line	fgets(str, n, fp);
fputs()	Write string	fputs(str, fp);

28.3 Formatted I/O

Function	Purpose	Syntax
fprintf()	Write formatted data	fprintf(fp, "format", vars);
fscanf()	Read formatted data	fscanf(fp, "format", &vars);

Example: Student Records

```c
#include <stdio.h>

struct Student
{
    char name[50];
    int rollNo;
    float marks;
};

int main()
{
    FILE *fp;
    struct Student s;
    int n, i;

    // Writing
    fp = fopen("students.txt", "w");
    printf("How many students? ");
    scanf("%d", &n);

    for (i = 0; i < n; i++)
    {
        printf("Enter name, roll, marks: ");
        scanf("%s %d %f", s.name, &s.rollNo, &s.marks);
        fprintf(fp, "%s %d %.2f\n", s.name, s.rollNo, s.marks);
    }
    fclose(fp);
```

```
    // Reading
    fp = fopen("students.txt", "r");
    printf("\n--- Student Records ---\n");
    while (fscanf(fp, "%s %d %f", s.name, &s.rollNo, &s.marks) != EOF)
    {
        printf("Name: %s, Roll: %d, Marks: %.2f\n", s.name, s.rollNo, s.marks);
    }
    fclose(fp);

    return 0;
}
```

28.4 Binary File I/O

Function	Purpose	Syntax
fread()	Read binary data	fread(ptr, size, count, fp);
fwrite()	Write binary data	fwrite(ptr, size, count, fp);

```
// Writing structure to binary file
struct Student s = {"John", 101, 85.5};
FILE *fp = fopen("student.dat", "wb");
fwrite(&s, sizeof(struct Student), 1, fp);
fclose(fp);

// Reading structure from binary file
struct Student s2;
fp = fopen("student.dat", "rb");
fread(&s2, sizeof(struct Student), 1, fp);
fclose(fp);
printf("Name: %s, Roll: %d\n", s2.name, s2.rollNo);
```

28.5 File Position Functions

Function	Purpose
fseek(fp, offset, origin)	Move file pointer to specific position
ftell(fp)	Returns current position

`rewind(fp)`	Moves to beginning of file

fseek origins:

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position
SEEK_END	2	End of file

```
fseek(fp, 0, SEEK_SET);    // Go to beginning
fseek(fp, 0, SEEK_END);    // Go to end
fseek(fp, 10, SEEK_SET);   // Go to 10th byte from beginning
fseek(fp, -5, SEEK_CUR);   // Go back 5 bytes from current

long pos = ftell(fp);      // Get current position
rewind(fp);                // Go to beginning (same as fseek(fp, 0, SEEK_SET))
```

UNIT 9: PREPROCESSOR DIRECTIVES

Chapter 29: Preprocessor & Macros

29.1 Definition

The **preprocessor** processes source code **before compilation**. Directives start with `#`.

29.2 Types of Preprocessor Directives

Directive	Purpose
#include	Includes header file
#define	Defines macro/constant
#undef	Undefines macro
#if, #elif, #else, #endif	Conditional compilation
#ifdef, #ifndef	Check if macro is defined

`#pragma`	Compiler-specific instructions
`#error`	Generates error message
`#line`	Changes line number

29.3 #include

```
#include <stdio.h>      // System header (search in system directories)
#include "myheader.h"   // User header (search in current directory first)
```

29.4 #define (Macros)

```
// Object-like macro (Constants)
#define PI 3.14159
#define MAX 100
#define NAME "John"

// Function-like macro
#define SQUARE(x) ((x) * (x))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ABS(x) ((x) < 0 ? -(x) : (x))

// Usage
int area = SQUARE(5);      // Expands to: ((5) * (5)) = 25
int m = MAX(10, 20);       // Expands to: ((10) > (20) ? (10) : (20)) = 20
```

Why Parentheses in Macros?

```
#define SQUARE(x) x * x       // WRONG!
int result = SQUARE(2+3);      // Expands to: 2+3 * 2+3 = 2+6+3 = 11

#define SQUARE(x) ((x) * (x))  // CORRECT!
int result = SQUARE(2+3);      // Expands to: ((2+3) * (2+3)) = 25
```

29.5 Conditional Compilation

```
#define DEBUG 1

#if DEBUG
    printf("Debug mode: variable = %d\n", x);
#endif

// Check if macro is defined
#ifdef MAX
    printf("MAX is defined as %d\n", MAX);
#endif

#ifndef PI
    #define PI 3.14159
#endif
```

29.6 Predefined Macros

Macro	Description
__DATE__	Current date (string)
__TIME__	Current time (string)
__FILE__	Current filename (string)
__LINE__	Current line number (integer)
__func__	Current function name (C99)

```
printf("File: %s\n", __FILE__);
printf("Line: %d\n", __LINE__);
printf("Date: %s\n", __DATE__);
printf("Time: %s\n", __TIME__);
```

IMPORTANT PROGRAMS FOR EXAMS

Program Collection

Basic Programs

```
// 1. Swap two numbers without third variable
a = a + b;
b = a - b;
a = a - b;

// 2. Check Leap Year
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("Leap Year");

// 3. Check Prime Number
int isPrime = 1;
for (i = 2; i <= n/2; i++)
{
    if (n % i == 0)
    {
        isPrime = 0;
        break;
    }
}

// 4. Fibonacci Series
int a = 0, b = 1, next;
for (i = 0; i < n; i++)
{
    printf("%d ", a);
    next = a + b;
    a = b;
    b = next;
}

// 5. Armstrong Number (153 = 1³ + 5³ + 3³)
int temp = num, sum = 0, digits = 0;
while (temp != 0) { digits++; temp /= 10; }
temp = num;
while (temp != 0)
{
    int rem = temp % 10;
```

```
    sum += pow(rem, digits);
    temp /= 10;
}
if (sum == num) printf("Armstrong");

// 6. Palindrome Number
int rev = 0, temp = num;
while (temp != 0)
{
    rev = rev * 10 + temp % 10;
    temp /= 10;
}
if (rev == num) printf("Palindrome");

// 7. GCD and LCM
// GCD (Euclidean)
while (b != 0) { temp = b; b = a % b; a = temp; }
gcd = a;
lcm = (original_a * original_b) / gcd;

// 8. Power of a number
long long result = 1;
for (i = 0; i < exp; i++)
    result *= base;

// 9. Sum of digits
int sum = 0;
while (num != 0)
{
    sum += num % 10;
    num /= 10;
}

// 10. Binary to Decimal
int decimal = 0, base = 1;
while (binary != 0)
{
    decimal += (binary % 10) * base;
    binary /= 10;
```

```
    base *= 2;
}
```

Pattern Programs

```
// Right Triangle (Stars)
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= i; j++)
        printf("* ");
    printf("\n");
}

// Inverted Triangle
for (i = n; i >= 1; i--)
{
    for (j = 1; j <= i; j++)
        printf("* ");
    printf("\n");
}

// Pyramid
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= n-i; j++)
        printf(" ");
    for (j = 1; j <= 2*i-1; j++)
        printf("*");
    printf("\n");
}

// Floyd's Triangle
int num = 1;
for (i = 1; i <= n; i++)
{
    for (j = 1; j <= i; j++)
        printf("%4d", num++);
    printf("\n");
```

```
}

// Pascal's Triangle
for (i = 0; i < n; i++)
{
   int coeff = 1;
   for (j = 0; j <= i; j++)
   {
      printf("%4d", coeff);
      coeff = coeff * (i - j) / (j + 1);
   }
   printf("\n");
}
```

QUICK REVISION NOTES

Summary Tables

Data Types Size

Type	Size	Format
char	1 byte	%c
int	4 bytes	%d
float	4 bytes	%f
double	8 bytes	%lf
long	4/8 bytes	%ld
long long	8 bytes	%lld

Operator Precedence (Simplified)

```
() [] -> .          HIGHEST
++ -- ! ~ (unary)
* / %
+ -
```

```
< <= > >=
== !=
&&
||
?:
= += -= ...
,                LOWEST
```

Loop Comparison

	for	while	do-while
Test	Before body	Before body	After body
Min runs	0	0	1
Use	Known count	Unknown count	At least once

Memory Functions

Function	Init	Parameters
malloc	Garbage	(total_bytes)
calloc	Zero	(count, size)
realloc	Keeps old	(ptr, new_size)
free	-	(ptr)

File Modes

Mode	Read	Write	Create	Truncate
r				
w				
a				
r+				
w+				
a+				

HOW TO CONVERT THIS TO PDF

Fastest Method (30 seconds):

1. Press Ctrl + P (Windows) or Cmd + P (Mac)
2. Select "Save as PDF" as destination
3. Click "Save"
   Done!

Other Methods:

- Google Docs File Download PDF
- Microsoft Word File Save As PDF
- Online: md2pdf.netlify.app or dillinger.io
- Notion Export PDF

This covers the **COMPLETE SYLLABUS** for **BCA Semester 1 - Programming in C** with:

29 Chapters covering ALL topics **Detailed explanations** with examples **100+ Code examples** with outputs **All operators** with precedence table **Control statements** (if, switch, loops) **Functions & Recursion** with examples **Arrays & Strings** with operations **Pointers** with dynamic memory **Structures, Unions, Enums** **File Handling** (read, write, binary) **Preprocessor Directives & Macros** **Important exam programs** **Quick revision notes** **Pattern programs**