



**Dr. Ambedkar Memorial Institute of
Information Technology & Management Science**

**LECTURE NOTES
ON**

Programming for Problem Solving

MCPC1003 MCA (1st Sem)

Prepared By

Prof. Pujarani Nanda

Module-1

Introduction to Problem Solving through programs:

Program:

Program is the set of instructions which is run by the computer to perform specific task. The task of developing program is called programming.

- Problem solving through programming involves a systematic, step-by-step process that begins with understanding and defining the problem, then devising a formal, step-by-step solution called an algorithm, representing that algorithm visually or in pseudocode, and finally translating the algorithm into a computer program that is then tested and debugged to ensure it correctly solves the problem

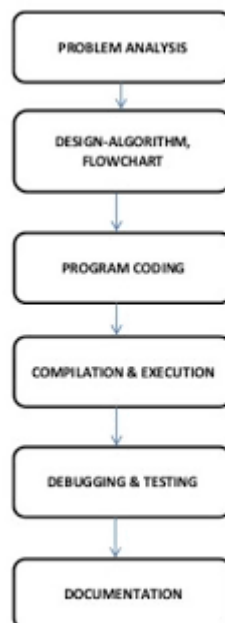


Figure: Problem solving process

Steps in Problem Solving

1. Define the Problem:

Clearly understand what the problem is and what the desired outcome should be. This involves understanding the real-world context and specifying the inputs and outputs of the problem.

2. Develop an Algorithm:

Create a formal, step-by-step solution to the problem. An algorithm is a precise and unambiguous sequence of instructions that leads to a solution.

3. Design Pseudocode and/or Flowcharts:

- **Pseudocode:** A plain-language description of an algorithm's steps, often using keywords like START and END, which is independent of any programming language.
- **Flowcharts:** A visual representation of the algorithm using different symbols and shapes to show the sequence of steps.

4. Write the Program:

Translate the algorithm, pseudocode, or flowchart into a specific programming language that a computer can understand and execute.

5. Test and Debug:

Run the program with various inputs to identify and correct any errors (bugs) to ensure the program produces the correct output and works as intended.

Algorithm:

- An algorithm is a process or set of rules which must be followed to complete a particular task. This is basically the step-by-step procedure to complete any task. All the tasks are followed a particular algorithm, from making a cup of tea to make high scalable software. This is the way to divide a task into several parts.

The algorithm is used for,

- To develop a framework for instructing computers.
- Introduced notation of basic functions to perform basic tasks.
- For defining and describing a big problem in small parts, so that it is very easy to execute.

Let's take an example to make a cup of tea,

Step 1: Start

Step 2: Take some water in a bowl.

Step 3: Put the water on a gas burner.

Step 4: Turn on the gas burner

Step 5: Wait for some time until the water is boiled.

Step 6: Add some tea leaves to the water according to the requirement.

Step 7: Then again wait for some time until the water is getting colorful as tea.

Step 8: Then add some sugar according to taste.

Step 9: Again wait for some time until the sugar is melted.

Step 10: Turn off the gas burner and serve the tea in cups with biscuits.

Step 11: End

Example 1. Swap two numbers with a third variable

Step 1: Start

Step 2: Take 2 numbers as input.

Step 3: Declare another variable as "temp".

Step 4: Store the first variable to "temp".

Step 5: Store the second variable to the First variable.

Step 6: Store the "temp" variable to the 2nd variable.

Step 7: Print the First and second variables.

Step 8: End







FLOWCHART

- The flowchart is a diagram which visually presents the flow of data through processing systems.
- This means by seeing a flow chart one can know the operations performed and the sequence of these operations in a system.
- Algorithms are nothing but sequence of steps for solving problems. So a flow chart can be used for representing an algorithm. A flowchart, will describe the operations (and in what sequence) are required to solve a given problem, a flow chart is a blueprint of a design made for solving a problem.

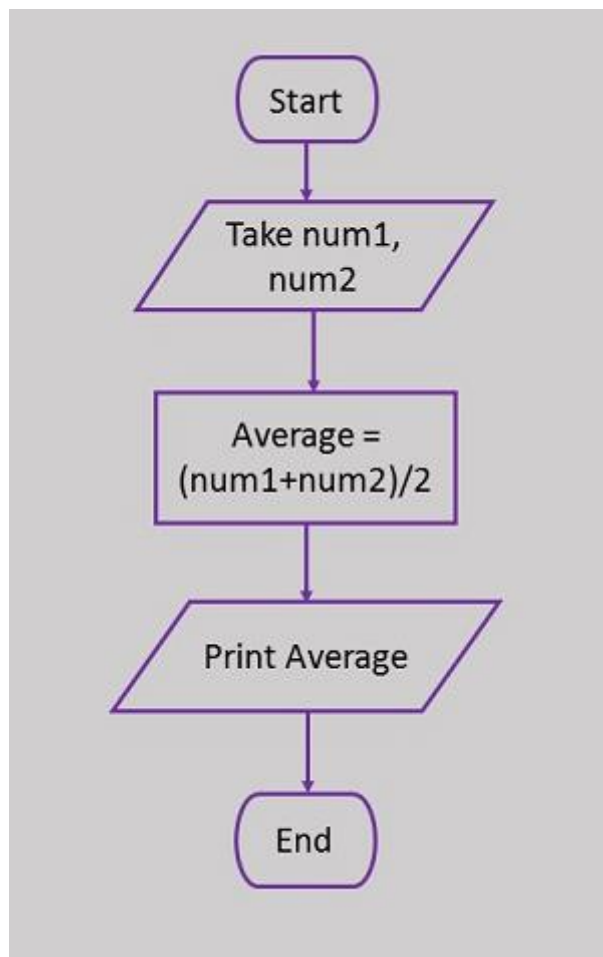
Flowchart Symbols

There are 6 basic symbols commonly used in flowcharting of assembly language Programs:

- Terminal,
- Process,
- input/output,
- Decision,
- Connector

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.

Example1. Draw a flow chart to find average of two no.s



Pseudocode:

Pseudocode is a readable description of a program or algorithm that uses everyday language rather than a specific programming language. It serves as a high-level design tool, enabling developers to conceptualize algorithms and processes without getting bogged down in syntax. At its core, pseudocode is a way to express the logic of a program or algorithm without the constraints of a specific programming language's syntax. Unlike actual program code, it's not meant to be compiled or run. Instead, pseudocode helps the programmer think through a problem and devise a plan for solving it. It enables easy modifications and is a quick way to write ideas and algorithms.

Advantages of using pseudocode

- Improved clarity
- More accessible communication
- Streamlined debugging
- Efficient planning

Example: Calculate the sum of first N natural numbers

```
START
INPUT N
SET sum ← 0
FOR i ← 1 TO N DO
    sum ← sum + i
ENDFOR
PRINT "Sum is ", sum
STOP
```

Example: Find the largest of two numbers

START

INPUT number1, number2

IF number1 > number2 THEN

PRINT "Largest number is ", number1

ELSE

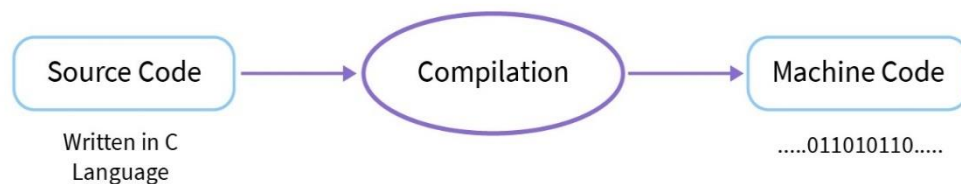
PRINT "Largest number is ", number2

ENDIF

STOP

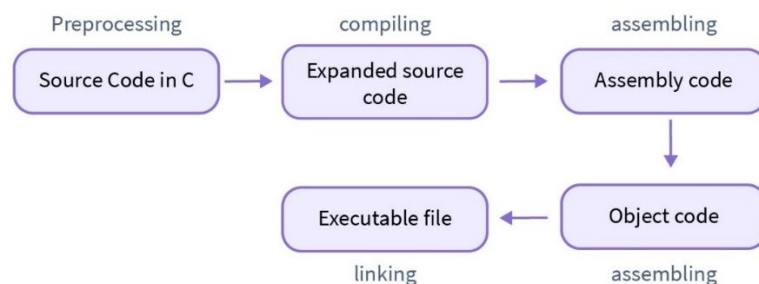
The Compilation Process:

A sequence of binary instructions consisting of 1 and 0 bits is called as machine code. High-level programming languages such as C, C++, Java, etc. consist of keywords that are closer to human languages such as English. Hence, a program written in C (or any other high-level language) needs to be converted to its equivalent machine code. This process is called compilation.



The compilation process has four different steps-

- Preprocessing
- Compiling
- Assembling
- Linking



1. Preprocessing

- **Input:** Source code file (e.g., .c file).
- **Process:** The preprocessor handles directives like #include and #define. It expands macros and includes header files, removing comments from the source code.
- **Output:** A preprocessed source code file, ready for the next stage.

2. Compilation

- **Input:** Preprocessed source code.
- **Process:** The compiler translates the preprocessed code into a low-level language called assembly language. It also performs checks for syntax and semantic errors.
- **Output:** An assembly code file (e.g., .s file).

3. Assembly

- **Input:** Assembly code file.
- **Process:** The assembler takes the assembly code and converts it into machine code. This machine code is represented as binary digits, forming an object file.
- **Output:** An object file (e.g., .o file).

4. Linking

- **Input:** Object files and library files.
- **Process:** The linker takes multiple object files (which can be from different parts of your program or from standard libraries) and combines them. It resolves external references and generates the final executable file.
- **Output:** An executable file (e.g., .exe or a.out) that the operating system can run.

Semantic Errors

A semantic error is text which is grammatically correct but doesn't make any sense. An example in the context of the C# language will be "int x = 12.3;" - 12.3 is not an integer literal and there is no implicit conversion from 12.3 to int, so this statement does not make sense. But it is grammatically correct.

Syntax Errors

Syntax errors occur when the text in a document being analyzed does not conform to the required structure defined by the grammar. Syntax error detection is more complicated than lexical error detection because the analyzer not only needs to make the best guess about what the problem might be, but it also needs to make the best guess about what the solution might be so it can provide meaningful error messages.

Variable

- A variable is a name given to the space in the memory for holding data such as integer, character, string, float etc.
- It is a value that can change anytime.

Rules of Variable:

- It is case sensitive.
- It must begin with a letter.
- The length of variable must be less than 8.\
- It should not be a keyword.
- Whitespace is not allowed.
- It should not contain any special character.
- Example- emp_name, roll_no, average1 etc.

Declaration of a variable:

- The Variable is declared by the support of data types with variable name.
- In 'C' all the variables should be declared before it can be used.

Syntax:

datatype v1, v2,.....vn;
or, datatype variable;

Here v1, v2,.....,vn are the variable name which are separated by comma.

Datatypes:

Data types are a fundamental concept in programming and are crucial for effective problem-solving. They define the kind of values a variable can hold and dictate the operations that can be performed on those values.

Integer Datatype:

The integer datatype in C is used to store whole numbers without decimal values.

Syntax- int var_name;

Character Datatype:

- Character datatype allows its variable to store only a single character. The size is 1 byte.
- It is the most basic datatype in C.
- It stores a single character and requires a single byte of memory in almost all compilers.

Syntax- char var_name;

Float Datatype:

- In C programming float datatype is used to store floating point values.
- Float in C is used to store decimal and exponential values.
- It is used to store decimal numbers(numbers with floating point values) with single precision.

Syntax- float var_name;

Double Datatype:

- A double datatype in C is used to store decimal numbers(numbers with floating point values) with double precision.
- It is used to define numeric values which hold numbers with decimal values in C.

Syntax- double var_name;

Arithmetic Expression

In problem-solving, an arithmetic expression is a combination of numbers and the four basic mathematical operations (+, -, ×, ÷) that evaluates to a single numerical value.

- **Numbers (operands):** These are the quantities involved in the problem.
- **Operators:** These are the symbols that indicate the type of operation to be performed (addition, subtraction, multiplication, or division).

- **Value:** The result obtained after performing the operations in the expression.

- An Arithmetic Expression is a combination of operands and Arithmetic operators, such as addition, subtraction, and so on. These combinations of operands and operators should be mathematically meaningful, otherwise, they can not be considered as an Arithmetic expression in C.
- The below table lists the different arithmetic operators available in the C programming language, along with a small description.

Symbol	Unary / Binary	Description
+	Unary	denotes that the number is a positive integer.
-	Unary	denotes that the number is a negative integer.
++	Unary	Increments the value of the variable by 1
--	Unary	Decreases the value of the variable by 1
+	Binary	performs the mathematical addition of the two given operands.
-	Binary	performs the mathematical subtraction of the two given operands.
*	Binary	performs the mathematical multiplication of the two given operands.
\	Binary	performs the mathematical division of the two given operands and returns the quotient.
%	Binary	performs the mathematical division of the two given operands and returns the remainder as the result.

Relational Operations:

In problem-solving, a relational operation is a comparison between two values that results in a Boolean (true or false) output, used in conditional statements and loops to control program flow. Common relational operators include "equal to" (==), "not equal to" (!=), "greater than" (>), "less than" (<), "greater than or equal to" (>=), and "less than or equal to" (<=). These operations are essential for making decisions within an algorithm or program, allowing it to react differently based on the relationship between pieces of data.

Types of Relational Operators in C

In C programming, relational operators allow you to compare values and make decisions based on the results of these comparisons. There are six main relational operators in C:

- Equal to Operator (==)
- Not Equal to Operator (!=)
- Less than Operator (<)
- Greater than Operator (>)
- Less than or Equal to Operator (<=)
- Greater than or Equal to Operator (>=)

Operator	Description	Example
Equal to (==)	Checks if two values are equal.	5 == 5 returns 1
Not Equal to (!=)	Checks if two values are not equal.	5 != 3 returns 1

Operator	Description	Example
Less than (<)	Checks if the left value is less than the right value.	3 < 5 returns 1
Greater than (>)	Checks if the left value is greater than the right value.	5 > 3 returns 1
Less than or Equal to (<=)	Checks if the left value is less than or equal to the right value.	3 <= 5 returns 1
Greater than or Equal to (>=)	Checks if the left value is greater than or equal to the right value.	5 >= 3 returns 1

Logical Expressions:

In problem-solving, a logical expression is a statement composed of variables and logical operators (AND, OR, NOT) that evaluates to either true or false. These expressions are used to represent problem conditions and their relationships, allowing for the systematic deduction of solutions by evaluating their truth values through rules of logic. By applying logical laws to simplify these expressions, one can find solutions and verify their correctness. Expressions may be relational or arithmetic.

We have 3 logical operators in the C language:

- **Logical AND (&&)**
- **Logical OR (||)**
- **Logical NOT (!)**

Operator	Meaning	Example
&&	Logical AND	5>6 && 5==5 F T =F 5>2 && 4>3 T T =T
	Logical OR	5>6 5==5 F T =T 5>3 5>4 T T =T
!	Logical NOT	5!=3 T

Conditional Branching:

Decision Making within a Program

- Decision making is the selection of a course of action from among available alternatives in order to produce a desired result.
- The conditional test either evaluates to a true or a false.
- The concept of evaluating and obtaining a result is referred to as decision making in a programming language.
- "True" is considered the same as "yes," which is also considered the same as 1.
- "False" is considered the same as "no," which is considered the same as 0.
- C programming language assumes any nonzero and nonnull values as true, and if it is either zero or null, then it is assumed as false value.

Control Statement

- A control statement modifies the order of statement execution.
- A control statement can cause other program statements to execute multiple times or not to execute at all, depending on the circumstances.

Types of Control Statement

1) Branching Statement: used to select one of the alternative statement.

a. Unconditional Branching

i. Goto Statement

b. Conditional Branching

i. if State

ii. ifelse Statement.

iii. switch Statement.

2) Looping or Iterative Statement: used to repeat the statement till the condition is true.

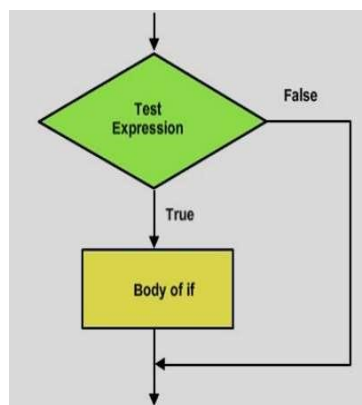
i. for loop

ii. while loop

iii. do while loop

If statement

- The 'if' statement is one of the C's program control statements.
- The 'if' statement evaluates an expression and directs program execution depending on the result of the evaluation.
- If expression evaluates to true, statement is executed.
- If statement evaluates to false, statement is not executed.
- An 'if' statement can control the execution of multiple statements through the use of a compound statement, or block.
- A block is a group of two or more statements enclosed in braces.



If statement Syntax

```
..  
if (condition)  
{  
    //Block of C statements here  
    //The above statements will only execute if the condition is true  
}  
..
```

If statement: Example

```
#include <stdio.h>
int main()
{
    int x = 20;
    int y = 22;
    if (x < y)
    {
        printf("Variable x is less than y");
    }
    return 0;
}
```

Output:

Variable x is less than y

Else Statement

- An 'if' statement can optionally include an else clause.
- The else clause is included as shown below:
- If expression evaluates to true, statement1 is executed.
- If expression evaluates to false, statement2 is executed.
- Both statement1 and statement2 can be compound statements or block.

If-else Statement

- The combination of the 'if' and 'else' clause is called the 'if-else' statement.
- If expression is true, statement1 is executed; otherwise, statement2 is executed.
- If the first expression, expression1, is true, statement1 is executed before the program continues with the next statement.

If-else statement Syntax

```
if (condition)
{
    /*Control will come inside only when the above condition is true*/
    //C statement(s)
}
else
{
    /*Control will come inside only when condition is false */
    //C statement(s)
}
```

```

#include <stdio.h>
int main()
{
    int m=40, n=20;
    if(m == n)
    {
        printf("m and n are equal");
    }
    else
    {
        printf("m and n are not equal");
    }
}

```

Output:

m and n are not equal

FOR LOOP:

Syntax

```

for (initialization; condition; increment/decrement)
{
    // code to be executed
}

```

Example:

Print numbers 1 to 5

```

#include <stdio.h>

```

```

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

```

Output:

1
2
3
4
5

WHILE LOOP:

Syntax:

```

while (condition) {
    // code to be executed
}

```

```
}
```

Example: Print numbers 1 to 5 using while loop

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

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

Output:

```
1  
2  
3  
4  
5
```

DO WHILE LOOP:

Syntax:

```
do {  
    // code to be executed  
} while (condition);
```

Example: Print numbers 1 to 5 using do-while loop

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

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

Output:

```
1  
2  
3  
4  
5
```

MODULE-2
FUNCTIONS AND ARRAYS

Introduction to Functions:

Functions in C are the basic building blocks of a C program. A function is a set of statements enclosed within curly brackets ({}) that take inputs, do the computation, and provide the resultant output. We can

call a function multiple times, thereby allowing reusability and modularity in C programming. It means that instead of writing the same code again and again for different arguments, you can simply enclose the code and make it a function and then call it multiple times by merely passing the various arguments.

Syntax of Functions

The basic syntax of functions in C programming is:

```
return type function name(arg1, arg2, ... argn)
{
    Body of the function //Statements to be processed
}
```

Aspects of Functions in C Programming

Functions in C programming have three general aspects: declaration, defining, and calling.

1. Function Declaration

The function declaration lets the compiler know the name, number of parameters, data types of parameters, and return type of a function. However, writing parameter names during declaration is optional, as you can do that even while defining the function.

2. Function Call

As the name gives out, a function call is calling a function to be executed by the compiler. You can call the function at any point in the entire program. The only thing to take care of is that you need to pass as many arguments of the same data type as mentioned while declaring the function. If the function parameter does not differ, the compiler will execute the program and give the return value.

3. Function Definition

It is defining the actual statements that the compiler will execute upon calling the function. You can think of it as the body of the function. Function definition must return only one value at the end of the execution.

Example: WAP to find sum of two integer using function

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

```
// Function declaration (prototype)
```

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

```
int main() {
```

```
    int x = 5, y = 3;
```

```
    int sum = add(x, y); // Calling the function
```

```
    printf("Sum: %d\n", sum);
```

```
    return 0;
```

```
}
```

```
// Function definition
```

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

```
    return a + b;
```

```
}
```

Output:

Sum: 8

Types of Functions in C

Functions in C programming are classified into two types:

1. Library Functions

Also referred to as predefined functions, library functions are already defined in the C libraries. This means that we do not have to write a definition or the function's body to call them. `Printf()`, `scanf()`, `ceil()`, and `floor()` are examples of library functions.

2. User-Defined Functions

These are the functions that a developer or the user declares, defines, and calls in a program. This increases the scope and functionality, and reusability of C programming as we can define and use any function we want. A major plus point of C programming is that we can add a user-defined to any library to use it in other programs.

Function Prototypes and Declaration:

A function prototype in C is a declaration of a function that specifies its name, return type, and parameters without providing the actual implementation. It acts as a "blueprint" for the compiler, informing it about the function before it is used.

Importance of C Function Prototype

- **Type Checking:** Ensures arguments and return values match the expected types.
- **Error Prevention:** Detects mismatches during compilation.
- **Modular Programming:** Allows function definitions to appear after the main program logic.
- **Multi-file Programs:** Enables sharing functions across multiple files.

Function Prototype Syntax

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

Example:

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

```
int add(int a, int b); // Function prototype
```

```
int main() {  
    int result = add(3, 5); // Function call  
    printf("Sum: %d\n", result);  
    return 0;  
}
```

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

Parameter passing in function:

In C, parameters can be passed to functions using two primary methods: pass-by-value and pass-by-reference (using pointers).

Pass-by-Value:

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



```

void increment(int x) {
    x = x + 1; // Modifies the copy, not the original 'num'
    printf("Inside function: x = %d\n", x);
}

int main() {
    int num = 10;
    printf("Before function call: num = %d\n", num);
    increment(num); // 'num' is passed by value
    printf("After function call: num = %d\n", num); // 'num' remains 10
    return 0;
}

```

Pass-by-Reference (using Pointers):

```

#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a; // Dereference to get the value
    *a = *b;       // Modify the value at the address 'a' points to
    *b = temp;     // Modify the value at the address 'b' points to
}

int main() {
    int num1 = 10, num2 = 20;
    printf("Before swap: num1 = %d, num2 = %d\n", num1, num2);
    swap(&num1, &num2); // Pass addresses using the '&' operator
    printf("After swap: num1 = %d, num2 = %d\n", num1, num2); // num1 and num2 are swapped
    return 0;
}

```

Recursion

Recursion in C is a programming technique where a function calls itself to solve a smaller part of the problem until it reaches a stopping condition. Instead of repeating code with loops, recursion solves a problem by breaking it down into simpler versions of the same problem. It continues calling itself with new values until a specific condition is met, known as the base case.

Syntax

```

returnType functionName(parameters) {
    if (base_case_condition) {
        // Base Case: defines when the recursion should stop
        return value;
    } else {
        // Recursive Case: function calls itself with modified parameters
        return functionName(updated_parameters);
    }
}

```

Types of Recursion in C

The following are the types of recursion in C language with examples:

1. Direct Recursion

When the recursion functions in C call themselves directly, it is known as direct recursion.

```
void directRecursion(int n) {  
    if (n > 0) {  
        printf("%d ", n);  
        directRecursion(n - 1); // Function calling itself directly  
    }  
}
```

2. Indirect Recursion

When a function calls another function, and that function again calls the first function, it's called indirect recursion.

```
void functionA(int n);  
void functionB(int n);
```

```
void functionA(int n) {  
    if (n > 0) {  
        printf("%d ", n);  
        functionB(n - 1); // Calls another function  
    }  
}
```

```
void functionB(int n) {  
    if (n > 0) {  
        printf("%d ", n);  
        functionA(n - 1); // Calls the first function again  
    }  
}
```

3. Tail Recursion

A recursion is called tail recursion if the recursive call is the last statement executed by the function.

```
void tailRecursion(int n) {  
    if (n == 0)  
        return;  
    printf("%d ", n);  
    tailRecursion(n - 1); // Recursive call is the last action  
}
```

4. Head Recursion

If the recursive call happens before any other processing in the function, it is known as head recursion.

```
void headRecursion(int n) {  
    if (n == 0)  
        return;  
    headRecursion(n - 1); // Recursive call comes first  
    printf("%d ", n);  
}
```

5. Tree Recursion

When a function calls itself more than once in each invocation, it results in multiple branches, forming a tree-like structure.

```
void treeRecursion(int n) {  
    if (n > 0) {  
        printf("%d ", n);
```

```

        treeRecursion(n - 1); // First recursive call
        treeRecursion(n - 1); // Second recursive call
    }
}

```

6. Nested Recursion

When a recursive function's argument itself is a recursive function call, it's called nested recursion.

```

int nestedRecursion(int n) {
    if (n > 100)
        return n - 10;
    else
        return nestedRecursion(nestedRecursion(n + 11));
}

```

Example

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

```

int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive case
}

```

```

int main() {
    int num = 5;
    printf("Factorial of %d is %d", num, factorial(num));
    return 0;
}

```

Output:

Factorial of 5 is 120

Arrays:

- An array in C is a collection of multiple values of the same data type stored together under a single variable name, instead of creating separate variables for each value.
- These index numbers start from 0, allowing easy access to each element. Arrays are especially useful when working with lists, such as marks of students, temperatures, or items in a menu.

Syntax of Array in C

```
data_type array_name[array_size];
```

Types of Arrays in C Language

There are three types of arrays in C programming:

1. One-Dimensional Array

A one-dimensional array in C is the simplest form of an array. It stores a list of elements of the same data type in a single row (linear structure), accessed using a single index.

Declaration:

```
int numbers[5];
```

Initialization:

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

Example:

```
#include <stdio.h>

int main() {

    int marks[3] = {85, 90, 95};

    for (int i = 0; i < 3; i++) {

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

    }

    return 0;

}
```

2. Two-Dimensional Array

A two-dimensional array in C stores data in a table-like structure using rows and columns. It's often used to represent matrices.

Declaration:

```
int matrix[2][3]; // 2 rows, 3 columns
```

Initialization:

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

Example:

```
#include <stdio.h>

int main() {
    int matrix[2][2] = {{10, 20}, {30, 40}};
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    }
    return 0;
}
```

3. Multi-Dimensional Array

A multi-dimensional array in C contains more than two dimensions (like 3D or 4D arrays). These are rarely used but helpful in special cases like 3D games or image processing.

Declaration:

```
int cube[2][2][2];
```

Initialization:

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

Examples of Arrays

1. Program to Print Elements of an Array

```
#include <stdio.h>  
  
int main() {  
    int numbers[5] = {10, 20, 30, 40, 50};  
    for (int i = 0; i < 5; i++) {  
        printf("Element %d: %d\n", i, numbers[i]);  
    }  
    return 0;  
}
```

2. Program to Take Input in an Array and Print Sum

```
#include <stdio.h>  
  
int main() {  
    int arr[5], sum = 0;  
    printf("Enter 5 numbers:\n");  
    for (int i = 0; i < 5; i++) {  
        scanf("%d", &arr[i]);  
        sum += arr[i];  
    }  
    printf("Sum = %d", sum);  
    return 0;  
}
```

3. Program to Find Maximum Element in an Array

```
#include <stdio.h>  
  
int main() {  
    int arr[5] = {11, 25, 9, 43, 31};  
    int max = arr[0];  
  
    for (int i = 1; i < 5; i++) {  
        if (arr[i] > max) {  
            max = arr[i];  
        }  
    }  
}
```

```

    }
    printf("Maximum element = %d", max);
    return 0;
}

```

4. Program to Reverse an Array

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

```

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printf("Reversed array: ");
    for (int i = 4; i >= 0; i--) {
        printf("%d ", arr[i]);
    }
    return 0;
}

```

Character Arrays and Strings in C

A character array is a sequence of characters stored in contiguous memory locations. In C, strings are represented using character arrays terminated with a null character `\0`, which tells the compiler where the string ends.

Declaring a Character Array (String)

Method 1: Using Character List: Must include `\0` at the end to make it a valid string. Size must be 1 more than the number of characters to accommodate the null terminator.

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

Method 2: Using String Literal (Recommended): Automatically adds `\0` at the end. It has easier and cleaner syntax.

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

Initializing and Printing Strings

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

```

int main() {
    char city[] = "Delhi";
    printf("City: %s", city); // Output: City: Delhi
    return 0;
}

```

Example:

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

```

int main() {
    // Character array without null terminator (Not a proper string)
    char charArray[5] = {'H', 'e', 'l', 'l', 'o'};

    // Proper string with null terminator

```

```

char string1[] = "Hello";

// Declare another string and take user input
char name[50];

printf("Enter your name: ");
scanf("%s", name); // note: scanf reads until whitespace

// Display outputs
printf("\nCharacter Array (Not a proper string):\n");
for (int i = 0; i < 5; i++) {
    printf("%c", charArray[i]);
}

printf("\n\nString1 (Proper string): %s\n", string1);
printf("User Input String: %s\n", name);

// Use some string functions
printf("\nString Length of name: %lu\n", strlen(name));

// Copying strings
char copiedName[50];
strcpy(copiedName, name);
printf("Copied Name: %s\n", copiedName);

// Concatenation
strcat(string1, "!");
printf("Modified string1 after concatenation: %s\n", string1);

return 0;
}

```

Output:

Enter your name: PUJA

Character Array (Not a proper string):

Hello

String1 (Proper string): Hello

User Input String: PUJA

String Length of name: 4

Copied Name: PUJA

Modified string1 after concatenation: Hello!

Advantages of Arrays in C

1. Efficient Memory Usage: Arrays provide a way to store data sequentially, reducing memory wastage.
2. Easy Data Access: Elements can be accessed directly using their indices.

3. Compact Code: Using loops, operations can be performed on multiple elements, reducing repetitive code.
4. Static Storage: Data is allocated contiguously, enabling faster access and manipulation.
5. Ideal for Fixed-Size Data: Arrays are perfect when the number of elements is known in advance.

Disadvantages of Arrays in C

1. Fixed Size: The size of an array must be defined at the time of declaration, limiting flexibility.
2. Homogeneous Data: Arrays can only store elements of the same data type.
3. No Built-in Bounds Checking: Accessing indices outside the declared range can lead to undefined behavior.
4. Insertion and Deletion: These operations are time-consuming as they require shifting elements.
5. Memory Allocation: If improperly handled, large arrays can lead to excessive memory usage.

MODULE-3

POINTERS AND STRUCTURES

Introduction to Pointer:

The pointers in C language refer to the variables that hold the addresses of different variables of similar data types. We use pointers to access the memory of the said variable and then manipulate their addresses in a program. The pointers are very distinctive features in C- it provides the language with flexibility and power.

Example:

```
int x = 10;
```

```
int* p = &x;
```

Here, the variable `p` is of pointer type, and it is pointing towards the address of the `x` variable, which is of the integer type.

Pointer Syntax

```
int* p;
```

Pointer Initialization

```
pointer_variable = &variable;
```

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;
```

```
c = 5;
```

```
pc = &c;
```

Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

Example

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

Example 1:

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

```
int main() {
    int x = 10;

    // Pointer declaration and initialization
    int * ptr = & x;

    // Printing the current value
    printf("Value of x = %d\n", * ptr);

    // Changing the value
    * ptr = 20;

    // Printing the updated value
    printf("Value of x = %d\n", * ptr);

    return 0;
}
```

Output

```
Value of x = 10
Value of x = 20
```

Use of Pointers in C

Here is a summary of how we use the following operators for the pointers in any program:

Operator Name		Uses and Meaning of Operator
*	Asterisk	Declares a pointer in a program. Returns the referenced variable's value.
&	Ampersand	Returns a variable's address

The Pointer to an Array

Here is an example to illustrate the same,

```
int arr [20];
int *q [20] = &arr; // The q variable of the pointer type is pointing towards the integer array's address or the address of arr.
```

The Pointer to a Function

Here is an example to illustrate the same,

```
void display (int);
```

```
void(*q)(int) = &show; // The q pointer is pointing towards the function's address in the program
```

The Pointer to a Structure

Here is an example to illustrate the same,

```
struct str {  
    int x;  
    float y;  
}ref;  
struct str *q = &ref;
```

Types of Pointers

There are various types of pointers that we can use in the C language. Let us take a look at the most important ones.

The Null Pointer

The null pointer has a value of 0. To create a null pointer in C, we assign the pointer with a null value during its declaration in the program. This type of method is especially useful when the pointer has no address assigned to it. Let us take a look at a program that illustrates how we use the null pointer:

```
#include <stdio.h>  
int main()  
{  
    int *a = NULL; // the null pointer declaration  
    printf("Value of the variable a in the program is equal to :\n%x",a);  
    return 0;  
}
```

The output obtained out of the program mentioned above will be:

Value of the variable a in the program is equal to: 0

The Void Pointer

The void pointer is also known as the generic pointer in the C language. This pointer has no standard data type, and we create it with the use of the keyword *void*. The void pointer is generally used for the storage of any variable's address. Let us take a look at a program that illustrates how we use the void pointer in a C program:

```
#include <stdio.h>  
int main()  
{  
    void *q = NULL; // the void pointer of the program  
    printf("Size of the void pointer in the program is equal to : %d\n",sizeof(q));  
    return 0;  
}
```

The output obtained out of the program mentioned above will be:

Size of the void pointer in the program is equal to : 4

The Wild Pointer

We can call a pointer a wild pointer if we haven't initialized it with anything. Such wild pointers are not very efficient in a program, as they may ultimately point towards some memory location that is unknown to us. It will ultimately lead to some problems in the program, and then, it will crash. Thus, you must be very careful when using wild pointers in a program. Let us take a look at a program that illustrates how we use the wild pointer in a C program:

```
#include <stdio.h>  
int main()  
{  
    int *w; // the wild pointer in the program
```

```
printf("\n%d", *w);
return 0;
}
```

Pros of using Pointers in C

- Pointers make it easy for us to access locations of memory.
- They provide an efficient way in which we can access the elements present in the structure of an array.
- We can use pointers for dynamic allocation and deallocation of memory.
- These are also used to create complex data structures, like the linked list, tree, graph, etc.
- It reduces the code and thus improves the overall performance. We can use it to retrieve the strings, trees, and many more. We can also use it with structures, arrays, and functions.
- We can use the pointers for returning various values from any given function.
- One can easily access any location of memory in the computer using the pointers.

Cons of Pointers in C

- The concept of pointers is a bit tricky to understand.
- These can lead to some errors like segmentation faults, accessing of unrequired memory locations, and many more.
- A pointer may cause corruption of memory if we provide it with an incorrect value.
- Pointers in a program can lead to leakage of memory.
- As compared to all the other variables, pointers are somewhat slower.
- Some programmers might find it very tricky to deal with pointers in a program. It is their responsibility to use these pointers and manipulate them carefully.

Pointer Arithmetic:

Pointer arithmetic in C means to perform operations like addition, subtraction, increment, or decrement on pointers to move through memory locations. Since a pointer holds an address, changing its value means pointing to a different location in memory.

The operations are based on the data type of the pointer—adding 1 to an int pointer moves it by 4 bytes, not 1. This is because it jumps by the size of the data type it points to.

Pointer Arithmetic Operations

1. Pointer Addition (ptr + n)

This operation moves the pointer forward by n elements. The actual movement in memory depends on the size of the data type the pointer is pointing to.

Example:

```
#include <stdio.h>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("Value at ptr: %d\n", *ptr);    // Output: 10
    printf("Value at ptr + 2: %d\n", *(ptr + 2)); // Output: 30
    return 0;
}
```

2. Pointer Subtraction (ptr - n)

Subtracting n from a pointer moves it backward by n elements.

Example:

```
#include <stdio.h>
int main() {
    int arr[] = {5, 15, 25, 35, 45};
    int *ptr = &arr[4]; // Pointing to 45

    printf("Current value: %d\n", *ptr); // Output: 45
    printf("Value at ptr - 2: %d\n", *(ptr - 2)); // Output: 25
    return 0;
}
```

3. Incrementing a Pointer (ptr++)

The increment operator moves the pointer to the next element of its type.

Example:

```
#include <stdio.h>
int main() {
    int arr[] = {100, 200, 300};
    int *ptr = arr;

    printf("Before increment: %d\n", *ptr); // Output: 100
    ptr++;
    printf("After increment: %d\n", *ptr); // Output: 200
    return 0;
}
```

4. Decrementing a Pointer (ptr--)

This moves the pointer one element back.

Example:

```
#include <stdio.h>
int main() {
    int arr[] = {7, 14, 21};
    int *ptr = &arr[2]; // Pointing to 21

    printf("Before decrement: %d\n", *ptr); // Output: 21
    ptr--;
    printf("After decrement: %d\n", *ptr); // Output: 14
    return 0;
}
```

Accessing Array Elements with Pointer Arithmetic

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;

    printf("First element: %d\n", *ptr); // 10
    printf("Second element: %d\n", *(ptr + 1)); // 20
    printf("Fourth element: %d\n", *(ptr + 3)); // 40

    return 0;
}
```

```
}
```

Traverse an Array Using Pointer Arithmetic

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

```
int main() {  
    int arr[] = {5, 10, 15, 20, 25};  
    int *ptr = arr;  
  
    printf("Array elements using pointer arithmetic:\n");  
    for (int i = 0; i < 5; i++) {  
        printf("%d ", *(ptr + i));  
    }  
  
    return 0;  
}
```

Dynamic Memory Allocation:

Creating and maintaining dynamic structures requires dynamic memory allocation— the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

Memory management Functions

1. Malloc
2. Calloc
3. Realloc
4. Free

Syntax

The following are the function used for dynamic memory allocation

➤ **void *malloc(int num);**

- This function allocates an array of num bytes and leave them uninitialized.

➤ **void *calloc(int num, int size);**

- This function allocates an array of num elements each of which size in bytes will be size.

➤ **void *realloc(void *address, int newsize);**

- This function re-allocates memory extending it upto newsize.

➤ **void free(void *address);**

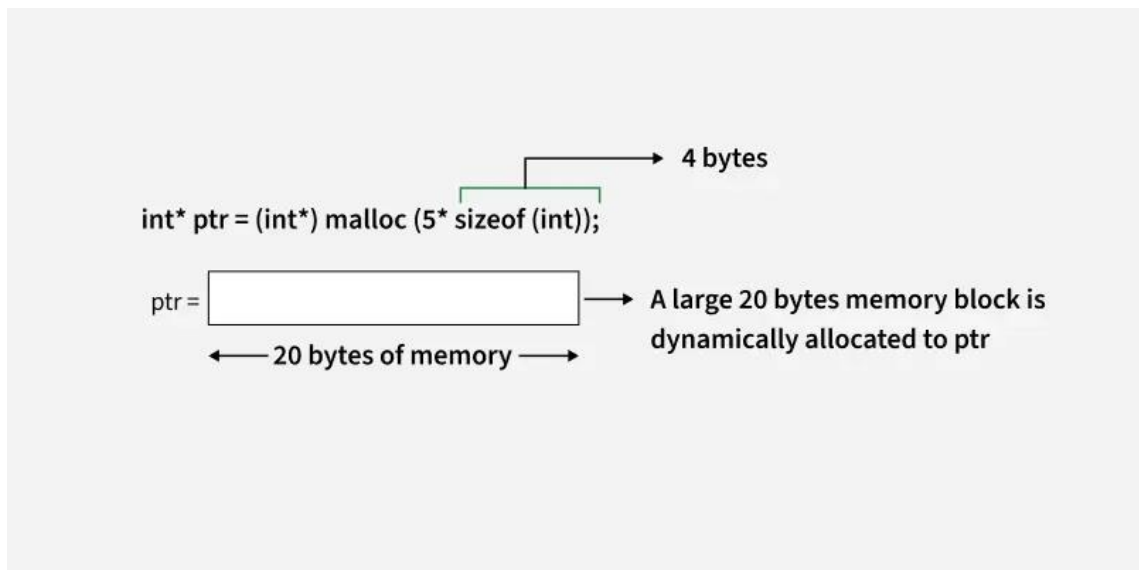
- This function releases a block of memory block specified by address.

Block Memory Allocation (malloc)

- Malloc function allocates a block of memory that contains the number of bytes specified in its parameter.
- It returns a void pointer to the first byte of the allocated memory
- The allocated memory is not initialized . We should therefore assume that it will contain unknown values and initialize it as required by our program.
- The function declaration is as follows

void* malloc (size_t size)

- If it is not successful malloc return NULL pointer.
- An attempt to allocate memory from heap when memory is insufficient is known as overflow.



Assume that we want to create an array to store 5 integers. Since the size of int is 4 bytes, we need $5 * 4$ bytes = 20 bytes of memory. This can be done as shown:

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

int main() {
    int *ptr = (int *)malloc(20);

    // Populate the array
    for (int i = 0; i < 5; i++)
        ptr[i] = i + 1;

    // Print the array
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);
    return 0;
}
```

Output

1 2 3 4 5

calloc()

The **calloc()** (stands for **contiguous allocation**) function is similar to `malloc()`, but it initializes the allocated memory to zero. It is used when you need memory with default zero values.

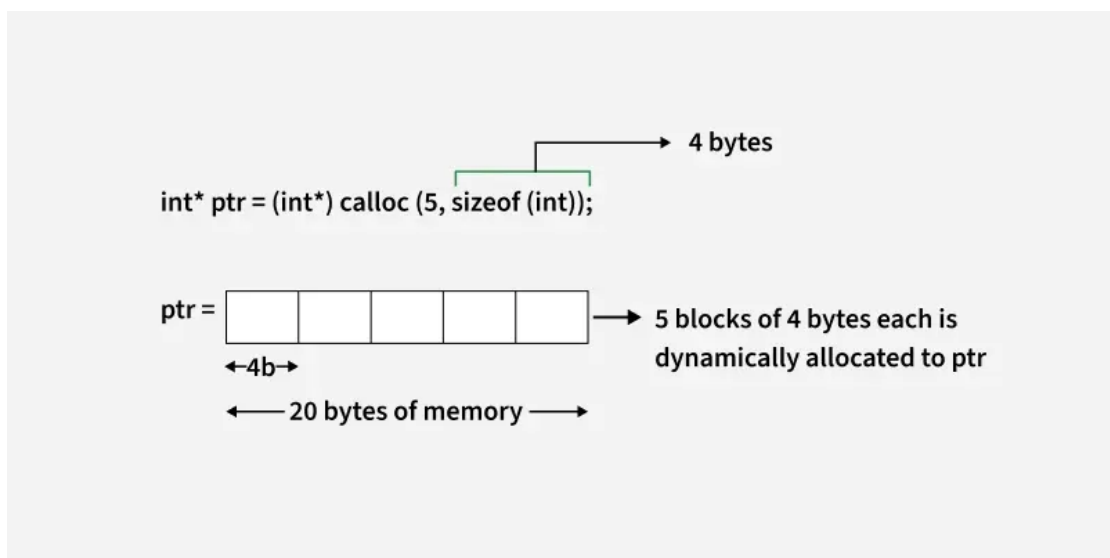
```
#include <stdio.h>
```

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

```
int main() {  
    int *ptr = (int *)calloc(5, sizeof(int));  
  
    // Checking if failed or pass  
    if (ptr == NULL) {  
        printf("Allocation Failed");  
        exit(0);  
    }  
  
    // No need to populate as already  
    // initialized to 0  
  
    // Print the array  
    for (int i = 0; i < 5; i++)  
        printf("%d ", ptr[i]);  
    return 0;  
}
```

Output

0 0 0 0 0



Syntax

```
calloc(n, size);
```

where **n** is the number of elements and **size** is the size of each element in bytes.

This function also returns a void pointer to the allocated memory that is converted to the pointer of required type to be usable. If allocation fails, it returns NULL pointer.

free()

The memory allocated using functions `malloc()` and `calloc()` is not de-allocated on their own. The [free\(\)](#) function is used to release dynamically allocated memory back to the operating system. It is essential to free memory that is no longer needed to avoid memory leaks.

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

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

```
int main() {
```

```
    int *ptr = (int *)calloc(5, sizeof(int));
```

```
    // Do some operations.....
```

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

```
        printf("%d ", ptr[i]);
```

```
    // Free the memory after completing
```

```
    // operations
```

```
    free(ptr);
```

```
    return 0;
```

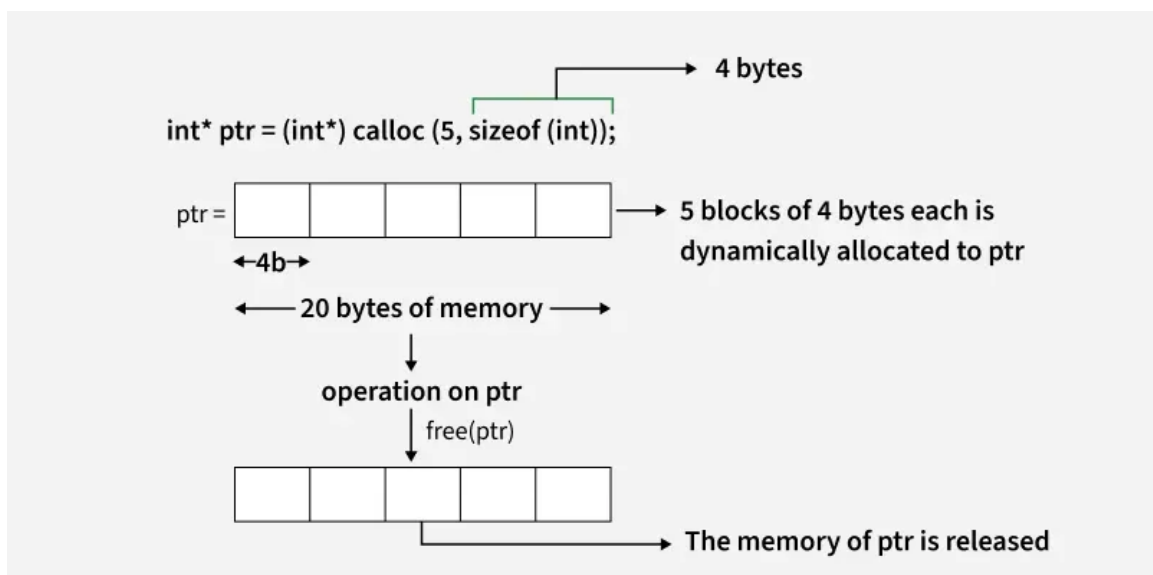
```
}
```

Output

0 0 0 0 0

After calling `free()`, it is a good practice to set the pointer to `NULL` to avoid using a "dangling pointer," which points to a memory location that has been deallocated.

```
ptr = NULL;
```



Syntax

```
free(ptr);
```

where **ptr** is the pointer to the allocated memory.

After freeing a memory block, the pointer becomes invalid, and it is no longer pointing to a valid memory location.

realloc()

realloc() function is used to resize a previously allocated memory block. It allows you to change the size of an existing memory allocation without needing to free the old memory and allocate a new block

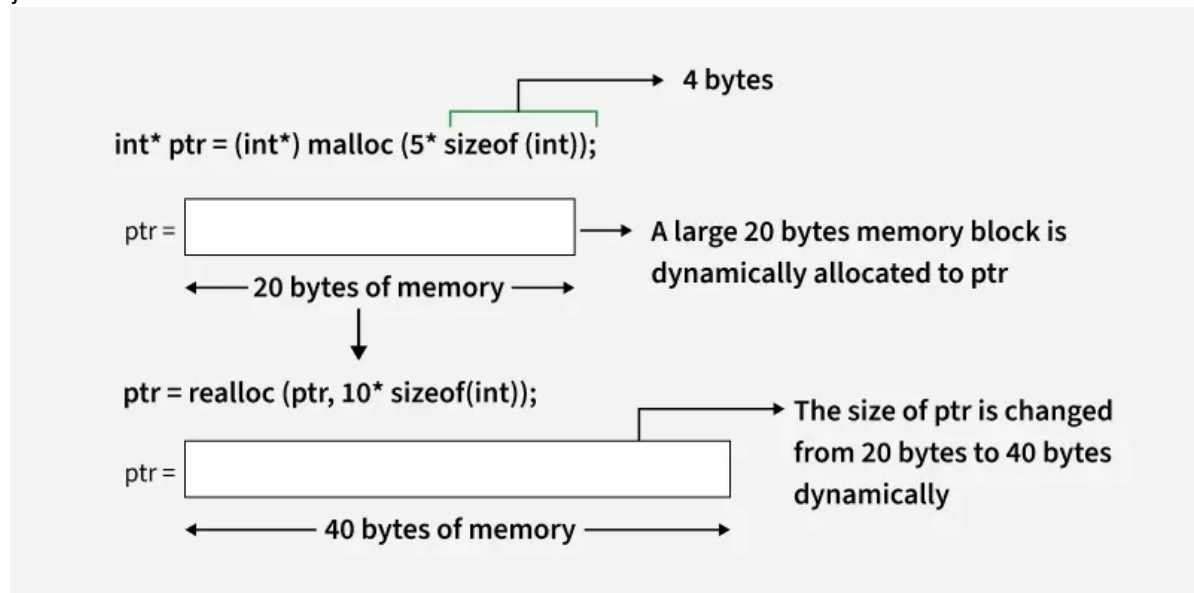
```
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
    int *ptr = (int *)malloc(5 * sizeof(int));

    // Resize the memory block to hold 10 integers
    ptr = (int *)realloc(ptr, 10 * sizeof(int));

    // Check for allocation failure
    if (ptr == NULL) {
        printf("Memory Reallocation Failed");
        exit(0);
    }

    return 0;
}
```



Structures and Unions

Structure is a collection of data items of different data types under a common name.

Structure is a group of items in which every item is identified by its own name. These items are the members of that structure.

Syntax for the declaration of a structure

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

Example:

```
struct abc
{
```

```
    char first [10];
    char last[20];
```

}Sname, Ename; (Structure variables) is

equivalent to

```
struct abc
{
```

```
    char first [10];
    char last[20];
};
```

struct abcSname, Ename;

This declaration creates two structure variables. Sname and Ename each of which contains two members: first,last.

Using Typedef to declare structure variables

Using the keyword **typedef** we can define any type to the easiest name that we want to use.

Eg. : `typedef struct abc node; // this defines struct abc as node`

At any place if we write `node` that is equivalent to `struct node`.

Another alternative usage of **typedef** is in the definition of the data type directly.

Eg. :

```
typedef struct student
{
    char first[10];
    char last[20];
} stu_type;
```

We can now use the typedef as in the following declaration:

```
stu_type Sname, Ename;
```

A structure member can be accessed by using the following format.

`struct_variable_name.member_name`

Eg : `printf ("%s",sname.first);` will display **first** (member) from **Sname**(structure variable).

Structures within a Structure

A structure can have the variable of another structure as a member. Such members can be more than one also.

eg: The following are the two structure definitions namely **adr**, **student** which are used in another structure **newaddr**.

```
struct adr
{
    char addr[40];
    char city[10];
    char state[3];
    char zip[6];
};
```

```
struct student
{
    char first[10];
    char last[20];
};
```

Now we can declare new structure **newaddr**

```
struct newaddr
{
    struct student name;

    struct adr address;
};
```

Initialization of structure variables

The following example illustrates the initialization of structures.

```
struct book
{
    char title[20];
    char Author[15];
    int pages;
    float price;
};
```

```
struct book book1={"abcd","xyz",100,25.50};
struct book book2={"aaa","xx"}; // book2.pages, book2.price will be initialized to
zero.
struct book book3=book1; // will copy the corresponding member of book1 to
book3
```

Note : In above structure definition **book1.price** (etc.) can be treated like any other ordinary variable.

Assigning values to structure members

The following example illustrates the assignment of values in structures. struct

```
book
{
    char title[20];
    char Author[15];
    int pages;
    float price;
}book1;

strcpy(book1.title, "CDS");
strcpy(book1.author, "K. Nagi Reddy");
book1.pages = 861;
book1.price= 215.00;
```

Reading values into the structure

Using scanf we can read the values into the structure.

```
scanf ("%s%s%d%f",book1.title,book1.author,&book1.pages,&book1.price);
```

Note: The structure variables cannot be used as the normal variables in all operations such as in relational statements, logical statements, etc.

Array Of Structures

Eg. The structure variables can be defined as an array if they required more in number.
struct marks

```
{  
  
    int sub1;  
    int sub2;  
    int sub3;  
};
```

```
static struct marks student[3] = {(50,60,70),(40,50,60),(10,20,30)};
```

Here student is array of 3 elements 0,1&2 initializes members as

```
student[0].sub1 = 50;          student[1].sub1 = 40; student[2].sub1 = 10;  
student[0].sub2 = 60;          student[1].sub2 = 50; student[2].sub2 = 20;  
student[0].sub3 = 70;          student[1].sub3 = 60; student[2].sub3 = 30;
```

Passing Structure Elements To Functions

Passing individual structure elements: The Individual elements of structures can be passed to functions as the normal variables by using **struct_var_name.member_name**

Example:

```
void main()  
{  
  
    void display(char[],char[],int); // Function Prototype  
    struct book  
    {  
  
        char name[25], author[25];  
        int pages;  
    };  
  
    static struct book b1={"DSTC","G. SORENSON",861};  
    display(b1.name,b1.author,b1.pages); // Function Call  
}  
  
void display(char *s, char *t, int n)    // Function definition  
{  
  
    printf("%s\n%s\n%d", s, t, n);  
}
```

Passing Entire Structure To Functions:

```
struct book
```

```

{
    char name[25], author[25];
    int pages;
};

void main()
{
    void display(struct book);
    static struct book b1={"DSTC","G. SORENSON",861};
    display(b1);
}

void display(struct book b)
{
    printf("s\n%s\n%d", b.name, b.author, b.pages);
}

```

Pointers to Structures

As for normal variables pointers can be declared to structures.

If **ptr** is a pointer to a structure, then the members of the structure can be accessed using **ptr->member_name** as in the following example.

Example:

```

void main()
{
    struct book
    {
        char name[25],author[25];
        int pages;
    };

    struct book b1={"DSTC","TENEN",672};
    struct book *ptr; /* (remember now ptr is not a variable, it is pointer) */
    ptr=&b1;
    printf("%s%s%d\n",b1.name,b1.author,b1.pages);
    printf("%s%s%d\n",ptr->name,ptr->author,ptr->pages);
}

```

[An illustrated example on Structures](#)

What data type are allowed to structure members? **Anything goes**: basic types, arrays, strings, pointers, even other structures. You can even make an **array of structures**.

Consider the program on the next few pages which uses an array of structures to make a deck of cards and deal out a poker hand.

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

```
struct playing_card  
{
```

```
int pips;  
char *suit;  
} deck[52];
```

```
void make_deck(void);  
void show_card(int n);
```



```
void main()
{
```

```
    make_deck();
    show_card(5);
    show_card(37);
    show_card(26);
    show_card(51);
    show_card(19);
}
```

```
void make_deck(void)
{
```

```
    int k;
    for(k=0; k<52; ++k)
    {
        if (k>=0 && k<13)
        {
            deck[k].suit="Hearts";
            deck[k].pips=k%13+2;
        }
        if (k>=13 && k<26)
        {
            deck[k].suit="Diamonds";
            deck[k].pips=k%13+2;
        }
        if (k>=26 && k<39)
        {
            deck[k].suit="Spades";
            deck[k].pips=k%13+2;
        }
        if (k>=39 && k<52)
        {
            deck[k].suit="Clubs";
            deck[k].pips=k%13+2;
        }
    }
}
```

```
void show_card(int n)
{
```

```
    switch(deck[n].pips)
```

```

{
    case 11:
        printf("%c of %s\n",'J',deck[n].suit); break;
    case 12:
        printf("%c of %s\n",'Q',deck[n].suit); break;
    case 13:
        printf("%c of %s\n",'K',deck[n].suit); break;
    case 14:
        printf("%c of %s\n",'A',deck[n].suit); break;
    default:
        printf("%c of %s\n",deck[n].pips,deck[n].suit);
        break;
}
}

```

Output

```

7 of Hearts
K of Spades
2 of Spades
A of Clubs
8 of Diamonds

```

Unions

Unions are C variables whose syntax looks similar to structures, but act in a completely different manner. A union is **a variable that can take on different data types** in different situations. The union syntax is:

```

union tag_name{
    type1 member1;
    type2 member2;
    ...
};

```

For example, the following code declares a union data type called **intfloat** and a union variable called **proteus**:

```

union intfloat { float f;
    int i;
};

union intfloat proteus;

```

Unions and Memory:

Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the **largest member**. (Unlike a structure where memory is reserved for **all** members).

In the previous example, 4 bytes are set aside for the variable **proteus** since a **float** will take up 4 bytes and an **int** only 2 (on some machines).

Data actually stored in a union's memory can be the data associated with **any** of its members. But **only one** member of a union can contain valid data at a given point in the program. It is the user's responsibility to keep track of which type of data has most recently been stored in the union variable.

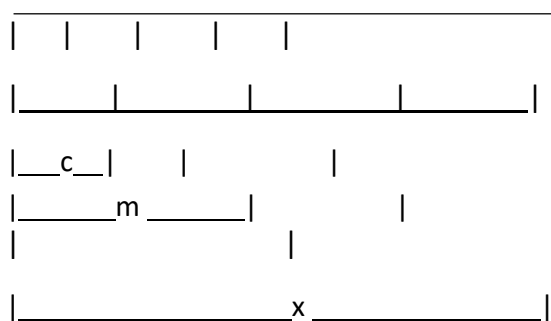
In the example:

```
union item
{
    int  m;
    float x;
    char c;
}code;
```

**** The size of a union data type will be the size of the bigger member variable.**

In this example: **Memory reserved for variable item is: 4 bytes**

1000 1001 1002 1003



Unions Example

The following code illustrates the chameleon-like nature of the union variable **proteus** defined earlier.

```
#include <stdio.h>
void main() { union
intfloat { float f;
int i;
} proteus;
```

```

proteus.i=4444 /* Statement 1 */
printf("i:%12d f:%16.10e\n",proteus.i,proteus.f);
proteus.f=4444.0; /* Statement 2 */ printf("i:%12d
f:%16.10e\n",proteus.i,proteus.f);
}

```

Output

```

i: 4444 f:6.2273703755e-42
i: 1166792216 f:4.440000000e+03

```

- After **Statement 1**, data stored in **proteus** is an integer the the float member is full of junk.
- After **Statement 2**, the data stored in **proteus** is a float, and the integer value is meaningless.

Comparison between structures and unions

Structures	Unions
This is the combination of different types of variables	This is also the combination of different types of variables
The Keyword used is 'struct'	The keyword used is 'union'
The size of a structure is the sum of the sizes of its individual elements	The size of the union is the size of its biggest element.
At any point of time all the members are valid	At any point of time only one member is valid

File Handling in C

The concept of files is to store data on the disk. A file is a place on the disk where group of related data is stored. C language has number of functions to perform file operations like,

1. Creating a file
2. Opening a file
3. Reading data from a file
4. Writing data into a file
5. Closing a file

There are two methods to perform file operations in C Language.

1. Low-level I/O (Uses UNIX system calls)
2. High-level I/O (Uses C language's standard I/O library)

DEFINING AND OPENING A FILE

If want to store data in a file on the disk, we must use a pointer variable of data structure of a file which is defined as **FILE** in the standard I/O library.

General form and opening a file is as follows:

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

fopen() is a library function which opens a file given in *filename* argument in the given *mode*.

filename in function fopen() is a string of characters that make up a valid file for the operating system. It contains two parts, a primary name and an extension name. The extension name is optional.

Example: Abc.dat
 Output
 First.c
 Transaction.txt

mode specifies the mode of opening and this can be r open the file for reading only w open the file for writing only a open the file for appending Recent compilers include additional modes of opening. They are r+ opens in read mode, both reading and writing can be done w+ opens in write mode, both reading and writing can be done a+ opens in append mode, both reading and writing can be done.

CLOSING A FILE

After all the operations are completed, a file must be closed. This can be as follows

```
fclose(fp);  
where fp is a file pointer.
```

INPUT/OUTPUT OPERATIONS ON FILES

Once a file is open, reading of or writing to it is accomplished using standard I/O routines. Standard I/O functions are listed bellow:

fopen()	Creates a new file or opens an existing file.
fclose()	Closes a file which has been opened.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
fprintf()	Writes set of data values to a file.
fscanf()	Reads a set of data values from a file.
getw()	Reads an integer from a file.

putw()	Writes an integer to a file.
fseek()	Sets the position to a desired point in the file.
ftell()	Returns the current position in the file in terms of bytes from the starting position.
rewind()	Sets the position to the beginning of the file.

The getc and putc functions :

The simplest file I/O functions are getc() and puts().

putc() function writes a character to a specified file.

Syntax: `putc(c,fp);`

where **c** is a character variable and **fp** is a FILE pointer associated with a specific file on the disk.

Character contained in variable **c** will be written in to **fp**.

getc() reads a character from a specified file.

Syntax: `c=getc(fp);`

where **c** is a character variable and **fp** is a FILE pointer associated with a specific file on the disk.

Getc function returns a character from the file specified in **fp** to **c**.

The file pointer moves by one character position for every operations of getc() or putc(). The getc() will return an end-of-file marker EOF, when end of the file has been reached.

The fprintf() and fscanf() functions

The functions fprintf() and fscanf() perform I/O operations that are identical to printf() and scanf() functions, except of course that they work on files.

Syntax: `fprintf(fp,"control string",list);`

where *fp* is FILE pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the *list*.

Example: `fprintf(fp,"%s %d %f",name,age,avg);`

Syntax of fscanf()

`fscanf(fp,"control string",list);`

where *fp* is FILE pointer associated with a file that has been opened for reading. The *control string* contains output specifications for the items in the *list*.

Example: `fscanf(fp,"%s %d %f",name,age,avg);`

ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operations.
5. Opening a file with an invalid file name.
6. Attempting to write to a write-protected file.

Examples:

/*The following program reads a text file and counts how many times each letter from 'A' to 'Z' occurs and displays the results.*/

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

int count[26];

void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    int i;
    /* see if file name is specified */ if
    (argc!=2) {
        printf("File name missing");
        exit(1);
    }

    if ((fp= fopen(argv[1], "r")) == NULL) {
        printf("cannot open file");
        exit(1);
    }

    while ((ch=fgetchar(fp)) !=EOF) { ch
    = toupper(ch);
        if (ch>='A' &&ch<='Z') count[ch-'A']++;
    }

    for (i=0; i<26; i++)
        printf("%c occurred %d times\n", i+'A', count[i]);

    fclose(fp);
}
```

```
/*This program uses command line arguments to read and display the contents of a file supplied as an argument. */
```

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



```

#define CLEAR 12 /* constant */

main(int argc , char *argv[])
{
    FILE *fp, *fopen();
    int c;

    putchar(CLEAR); while (

        --argc > 0 )
        if ((fp=fopen(argv[1], "r"))!=NULL)
        {
            printf("I can't open %s\n", argv[1]);
            break;
        }

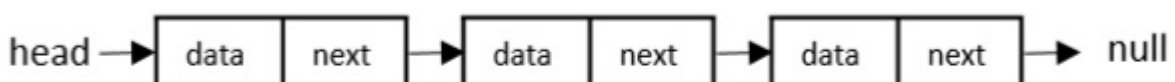
        else
        {
            while ((c= getc(fp)) !=EOF)
            putc(c,stdout); /* display to the screen */
            fclose(fp);
        }
    }
}

```

Self-referential Structures

A self-referential structure is a struct data type in C, where one or more of its elements are pointer to variables of its own type. Self-referential user-defined types are of immense use in C programming. They are extensively used to build complex and dynamic data structures such as [linked lists](#) and [trees](#).

In C programming, an [array](#) is allocated the required memory at compile-time and the array size cannot be modified during the runtime. Self-referential structures let you emulate the arrays by handling the size dynamically.



File management systems in Operating Systems are built upon dynamically constructed tree structures, which are manipulated by self-referential structures. Self-referential structures are also employed in many complex algorithms.

Defining a Self-referential Structure

A general syntax of defining a self-referential structure is as follows –

```
struct typename{  
    type var1;  
    type var2;  
    ...  
    ...  
    struct typename *var3;  
}
```

Let us understand how a self-referential structure is used, with the help of the following **example**. We define a struct type called **mystruct**. It has an integer element "**a**" and "**b**" is the pointer to **mystruct** type itself.

We declare three variables of **mystruct** type –

```
struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};
```

Next, we declare three "mystruct" pointers and assign the references **x**, **y** and **z** to them.

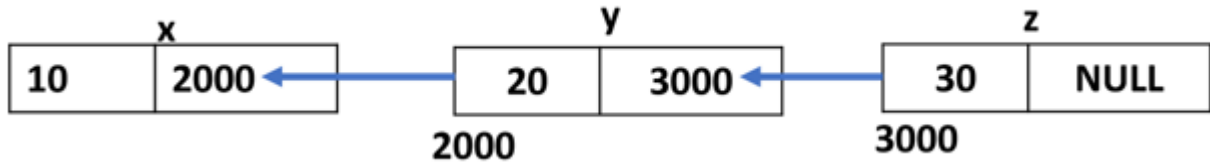
```
struct mystruct * p1, *p2, *p3;
```

```
p1 = &x;
```

```
p2 = &y;
```

```
p3 = &z;
```

The variables "x", "y" and "z" are unrelated as they will be located at random locations, unlike the array where all its elements are in adjacent locations.



Examples of Self-referential Structure

Example 1

To explicitly establish a link between the three variable, we can store the address of "y" in "x" and the address of "z" in "y". Let us implement this in the following program –

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

```
struct mystruct{
    int a;
    struct mystruct *b;
};
```

```
int main(){

    struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};

    struct mystruct * p1, *p2, *p3;

    p1 = &x;
    p2 = &y;
    p3 = &z;
```

```
x.b = p2;
```

```
y.b = p3;
```

```
printf("Address of x: %d a: %d Address of next: %d\n", p1, x.a, x.b);
```

```
printf("Address of y: %d a: %d Address of next: %d\n", p2, y.a, y.b);
```

```
printf("Address of z: %d a: %d Address of next: %d\n", p3, z.a, z.b);
```

```
return 0;
```

```
}
```

Output

Address of x: 659042000 a: 10 Address of next: 659042016

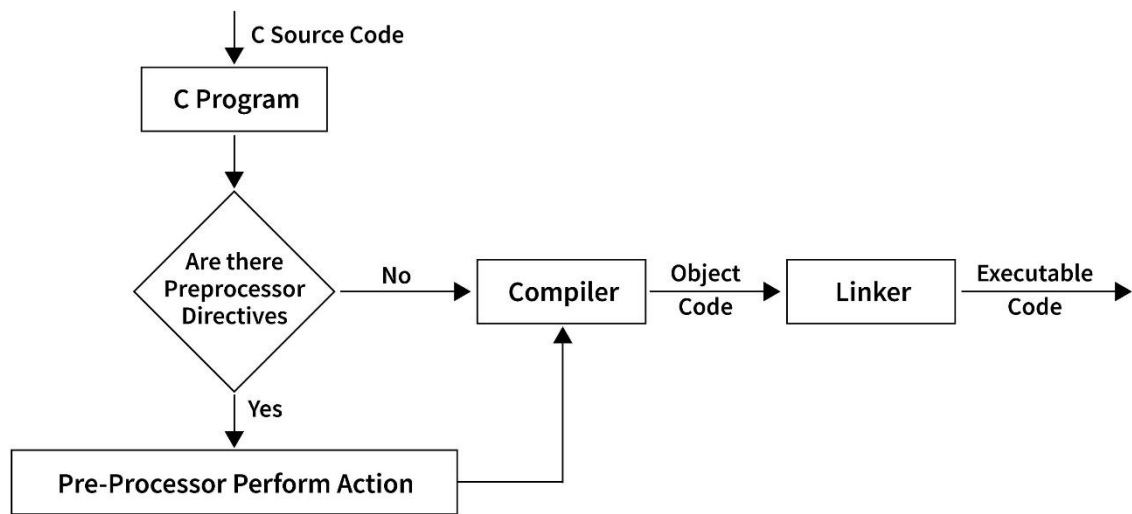
Address of y: 659042016 a: 20 Address of next: 659042032

Address of z: 659042032 a: 30 Address of next: 0

Module 4

Advanced Concepts in C

As the name suggests, the preprocessor directives in C are actually a text processing tool or we can call it a text substitution tool. All the preprocessor directives begin with a hash symbol #. It pre-processes the program before the compilation of the program.



Example:

```
#include <stdio.h>

#define PI 3.1415

main()
{
    printf("THE VALUE OF PI IS: %f",PI);
}

#include <stdio.h>

#define PI 3.1415

main()
{
    printf("THE VALUE OF PI IS: %f",PI);
}
```

Output:

THE VALUE OF PI IS: 3.141500

Command Line Arguments

- When the main function of a program contains arguments, then these arguments are known as Command Line Arguments.
- The main function can be created with two methods: first with no parameters (void) and second with two parameters. The parameters are argc and argv, where argc is an integer and the argv is a list of command line arguments.
- **argc** denotes the number of arguments given, while **argv[]** is a pointer array pointing to each parameter passed to the program. If no argument is given, the value of **argc** will be 1.
- The value of **argc** should be non-negative.

Syntax:

* Main function without arguments:

```
int main()
```

* Main function with arguments:

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

Properties of Command Line Arguments in C

- Command line arguments in C are passed to the main function as **argc** and **argv**.
- Command line arguments are used to control the program from the outside.
- **argv[argc]** is a Null pointer.
- The **name** of the program is stored in **argv[0]**, the first command-line parameter in **argv[1]**, and the last argument in **argv[n]**.
- Command-line arguments are useful when you want to control your program from outside rather than hard coding the values inside the code.
- To allow the usage of standard input and output so that we can utilize the shell to chain commands.
- To override defaults and have more direct control over the application. This is helpful in testing since it allows test scripts to run the application.

Bitwise Operators

Bitwise operators are used to manipulate bits in various different ways. They are equivalent to how we use mathematical operations like (+, -, /, *) among numbers, similarly we use bitwise operators like (|, &, ^, <<, >>, ~) among bits.

6 Bitwise operators in C

There are **6 bitwise operators** in C language. They are

- AND (&)

- OR (|)
- XOR (^)
- COMPLEMENT (~)
- Left Shift (<<)
- Right Shift (>>)

The symbols and names of some of these operators may appear similar to the logical operators, But make no mistake, these are different from them.

Bitwise operators vs Logical operators in C

The bitwise operators like AND, and OR can be sometimes confusing for newbies

If you have previously learned about logical operators, you might have come across Logical AND, and Logical OR. Many people have a tendency to confuse them with the Bitwise AND, and Bitwise OR operators. So, let's try to understand how are they different from each other.

The logical operators work with Boolean data and return a Boolean value, i.e. True or False.

The bitwise operators in C work with integers, i.e. they take integer inputs, manipulate with their bit and return an integer value. The bitwise AND, and OR use '&' and '|' as their operators, while the logical AND, and OR use '&&' and '||' as their operators.

1. The **& (bitwise AND)** in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C takes two numbers, the left shifts the bits of the first operand, and the second operand decides the number of places to shift.
5. The **>> (right shift)** in C takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C takes one number and inverts all bits of it.

Let's look at the truth table of the bitwise operators.

X	Y	X & Y	X Y	X ^ Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1

X	Y	X & Y	X Y	X ^ Y
1	1	1	1	0

Error Handling and Debugging Techniques

In C programming, **error handling** refers to the process of detecting and managing errors that occur during program execution. Errors can be of various types — **syntax errors**, **logical errors**, **runtime errors**, and **linker errors**.

C does not have built-in exception handling like modern languages, but errors can be handled using techniques such as **return values**, the **errno variable**, **perror()**, **strerror()**, and **exit status codes**. These methods help identify and report errors gracefully without crashing the program.

Debugging is the process of finding and correcting these errors. Common debugging techniques include using **printf() statements** to trace variable values, employing tools like **GDB (GNU Debugger)** to step through code, and using **assertions** to check logical conditions during runtime.

Together, error handling and debugging ensure that C programs are reliable, stable, and produce correct results.

Errors are common during program development. They can occur at various stages such as **compilation**, **linking**, or **runtime**.

Handling and fixing these errors is essential to make the program work correctly.

Type of Error	Description	Example
1. Syntax Error	Caused by violation of grammar/syntax rules of C. Detected by the compiler.	<code>printf("Hello")</code> ← missing semicolon
2. Logical Error	Program runs but gives wrong output due to wrong logic.	Using <code>+</code> instead of <code>*</code> in an area formula
3. Runtime Error	Occurs while the program is running (not during compilation).	Division by zero, invalid memory access
4. Linker Error	Occurs when linking compiled code to libraries or other files fails.	Undefined functions or missing header
5. Semantic Error	Meaning of the statement is wrong though syntax is correct.	<code>int x = "abc";</code>

Error Handling Techniques

1. Using Return Values

Functions can return a specific value to indicate an error.

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

```
int divide(int a, int b) {  
    if (b == 0)  
        return -1; // error code  
    return a / b;  
}
```

```
int main() {
```

```

int result = divide(10, 0);
if (result == -1)
    printf("Error: Division by zero\n");
else
    printf("Result = %d\n", result);
}

```

Debugging Techniques in C

Debugging means **finding and fixing errors** in your program.

Using Printf Statements

— insert `printf()` to track variable values and program flow.

Introduction to Data Structures in C

Data structures in C provide methods for storing and organizing data in a computer's memory to facilitate efficient access and manipulation. They are fundamental to designing efficient algorithms and managing data effectively in various applications.

1. What are Data Structures?

A data structure is a particular way of arranging data in a computer's memory. This organization aims to optimize operations such as insertion, deletion, searching, and retrieval, leading to more efficient programs in terms of both time and space complexity.

2. Types of Data Structures in C:

Data structures in C are broadly classified into two categories:

- **Primitive Data Structures:**

These are the basic data types built into the C language, capable of storing single values. Examples include `int`, `float`, `char`, and `double`.

- **Non-Primitive Data Structures:**

These are derived from primitive data types and are designed to store collections of data, potentially of different types. They are often referred to as user-defined data types. Examples include:

- **Arrays:** Collections of homogeneous elements stored in contiguous memory locations.
- **Linked Lists:** Collections of nodes where each node contains data and a pointer to the next node, allowing for dynamic sizing and efficient insertions/deletions.
- **Stacks:** A linear data structure following the Last-In, First-Out (LIFO) principle.
- **Queues:** A linear data structure following the First-In, First-Out (FIFO) principle.
- **Trees:** Hierarchical data structures consisting of nodes connected by edges, useful for representing hierarchical relationships.

- **Graphs:** Non-linear data structures consisting of nodes (vertices) and connections (edges) between them, representing complex relationships.

3. Importance of Data Structures in C:

- **Efficient Data Management:**

They enable the efficient storage, retrieval, and manipulation of large datasets.

- **Algorithm Design:**

Data structures are integral to designing effective algorithms for solving various computational problems.

- **Memory Optimization:**

Choosing the right data structure can lead to better utilization of memory resources.

- **Problem Solving:**

They provide a framework for structuring data to address real-world problems in software development. Understanding data structures in C is crucial for developing robust and efficient software applications, as they form the backbone of how data is organized and processed within a program.

