Beginner's Sheet

Basic Problems

1. Write a program to print "Hello, World!".
2. Write a program to add two numbers.
3. Calculate the area of a circle.
4. Swap two variables without using a temporary variable.
5. Convert Celsius to Fahrenheit.

Decision Making

1. Check if a number is even or odd.
2. Check if a string is a palindrome.
3. Check if a number is positive, negative, or zero.
4. Create a simple calculator (addition, subtraction, multiplication, division).
5. Check if a year is a leap year.
6. Check if two strings are anagrams.
7. Check if a number is an Armstrong number or not.
8. Check if a number is a prime number.

Loops

1. Find the factorial of a number.
2. Reverse a number using a loop.
3. Sum of Digits of a number (123: 1 + 2 + 3)
4. Check if a number is a palindrome using a loop.
5. Count the number of digits in an integer.
6. Print multiplication table using a loop.
7. Calculate the sum of natural numbers using a loop.
8. Find the LCM and GCD of two numbers using loops.
9. Print the Fibonacci sequence using a loop.
10. Calculate the power of a number using a loop.
11. Calculate the sum of natural numbers up to a given limit.
12. Print the Fibonacci sequence up to a certain limit.
13. Calculate the power of a number using a loop.
14. Find the factors of a number.
15. Determine the number of days between two dates.

String Manipulation

1. Reverse a string.
2. Count the number of vowels in a string.
3. Find the length of a string.
4. Count the number of occurrences of a character in a string.
5. Convert a string to lowercase.
6. Concatenate two strings.
7. Count the number of words in a sentence.
8. Convert a list of strings to uppercase.
9. Determine if a given string is a pangram.
10. Check if a given string is a valid email address.

Functions

1. Write a function to calculate the factorial of a number.
2. Write a function to check if a number is prime.
3. Write a function to calculate the nth Fibonacci number.

4. Write a function to reverse a string.
5. Write a function to find the maximum and minimum of an array.
6. Write a function to sort an array.
7. Write a function to check if a string is a palindrome.
8. Write a function to concatenate two strings.
9. Create a function to calculate the nth Fibonacci number.

Lists/Arrays

1. Find the largest element in a list/array.
2. Calculate the average of a list/array of numbers.
3. Remove duplicates from a list/array.
4. Find the intersection of two lists/arrays.
5. Sort a list/array of numbers in ascending order.
6. Find the maximum and minimum elements in a list/array.
7. Check if a list/array is empty.
8. Iterate over a list/array.
9. Check if a list/array is sorted in ascending order.
10. Find the index of the first occurrence of an element in a list/array.
11. Implement a basic binary search algorithm.

UNIT-I: Introduction to Programming and Problem Solving

Introduction to Programming Languages, Basics of a Computer Program- Algorithms, Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms. flowcharts (Using Dia Tool), pseudo code.

Structure of C Program Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operators, keywords, identifiers, Type Conversion, and Casting.

Language is a mode of communication that is used to share ideas and opinions with each other. For example, if we want to teach someone, we need a language that is understandable by both communicators.

What is a Programming Language?

A programming language is a computer language that is used by programmers (developers) to communicate with computers.

It is a set of instructions written in any specific language (C, C++, Java, Python) to perform a specific task.

Programming languages are often categorized into three levels based on their **abstraction from the machine hardware.**

1. Low-Level
2. Middle-Level
3. High-Level Languages.

1) Low-Level Programming Language:

Low-level languages are closer to machine code or hardware. They provide little to no abstraction and are highly dependent on the machine architecture, making them efficient but more difficult to write and understand for humans.

Characteristics:

Close to hardware: Offers minimal abstraction from the computer's hardware.

Efficient: Programs written in low-level languages run fast since they communicate directly with the hardware.

Difficult to write and understand: Programming in low-level languages requires knowledge of hardware details, making the code harder to write, debug, and maintain

Example: Machine Code, Assembly Code

2) Middle-Level Programming Language:

A middle-level programming language is a type of programming language that has features of both low-level and high-level languages. It provides some abstraction from the hardware, like high-level languages, but also allows for direct interaction with hardware, like low-level languages.

Characteristics:

Closer to the machine than high-level languages: It can access memory and hardware directly.

Easier to use than low-level languages: It includes more human-readable constructs, making it simpler to write code.

Balanced: It strikes a balance between efficiency (performance) and ease of programming. .

Examples: C, C++

3) High-Level Programming Language:

High-level programming languages are programming languages that are designed to be easy for humans to read and write. High-level languages are user-friendly, abstract away hardware complexities, and allow developers to focus on writing logical, efficient code.

Characteristics:

Human-readable syntax: The syntax is closer to natural language or mathematical notation, making it easier to understand and write.

Portable: High-level languages are not tied to a specific type of computer hardware, so the same code can often be run on different machines.

Execution: Slower execution compared to low- and middle-level languages because of higher abstraction.

Examples: Java, Python, Ruby, Php etc….

Types of Programming Paradigm:

1. Procedural programming languages
2. Functional programming languages
3. Object-oriented programming languages (OOP)
4. Scripting languages
5. Logic programming languages

Procedural programming languages

A procedural language follows a sequence of statements or commands to achieve a desired output. Each series of steps is called a procedure, and a program written in one of these languages will have one or more procedures within it.

Common examples of procedural languages include C and C++, Pascal, BASIC

Object Oriented Programming Languages

The programming languages which are used to develop the applications by using class and object are called Object Oriented Programming Languages.

Common examples of OOP languages include Java, Python, PHP, C++, Ruby

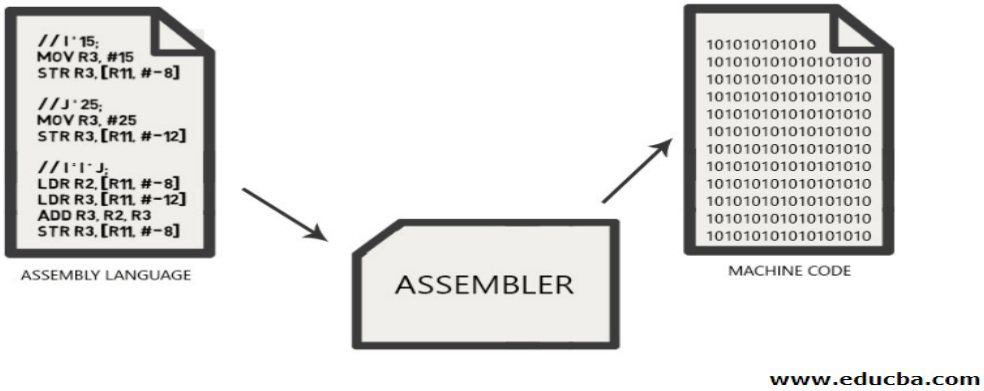
Translator:

A translator refers to a type of software that converts code written in one programming language into another language or into a machine-readable format.

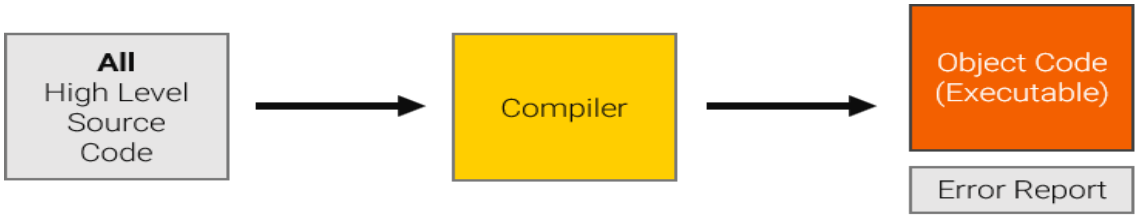
There are different types of translators in programming.

a) **Assembler**
b) **Compiler**
c) **Interpreter**

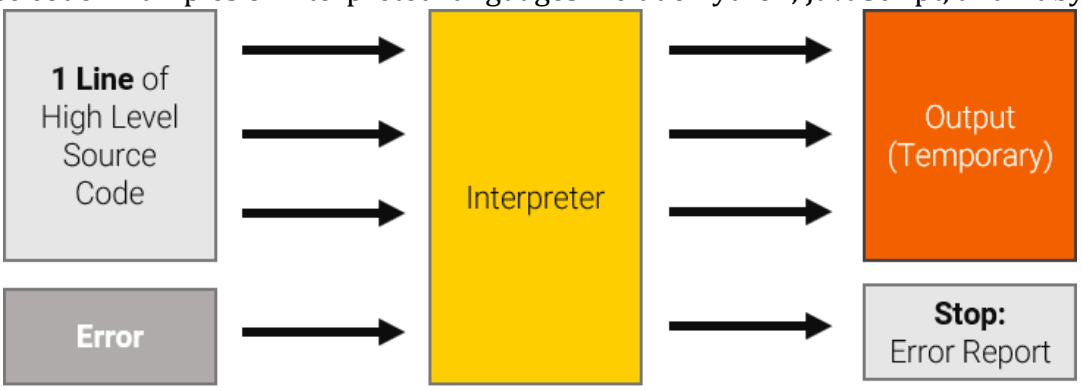
Assembler: An assembler translates assembly language, which is a low-level language closely related to machine code into actual machine code (binary code) that the computer's CPU can execute.

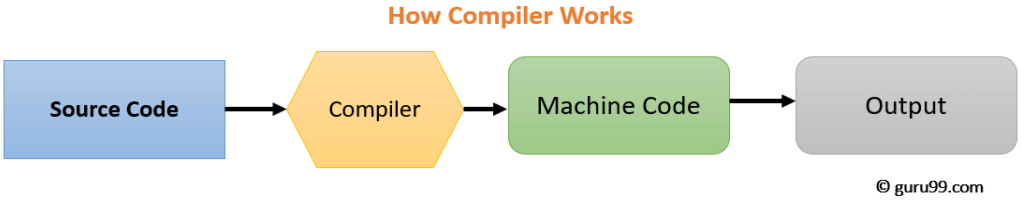


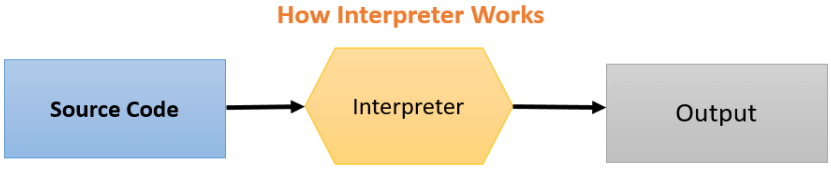
Compiler: A compiler translates code from a high-level programming language (like C, Java, or Python) into machine code, which is a low-level language that the computer's hardware can execute directly. The entire source code is typically translated at once, producing an executable file.



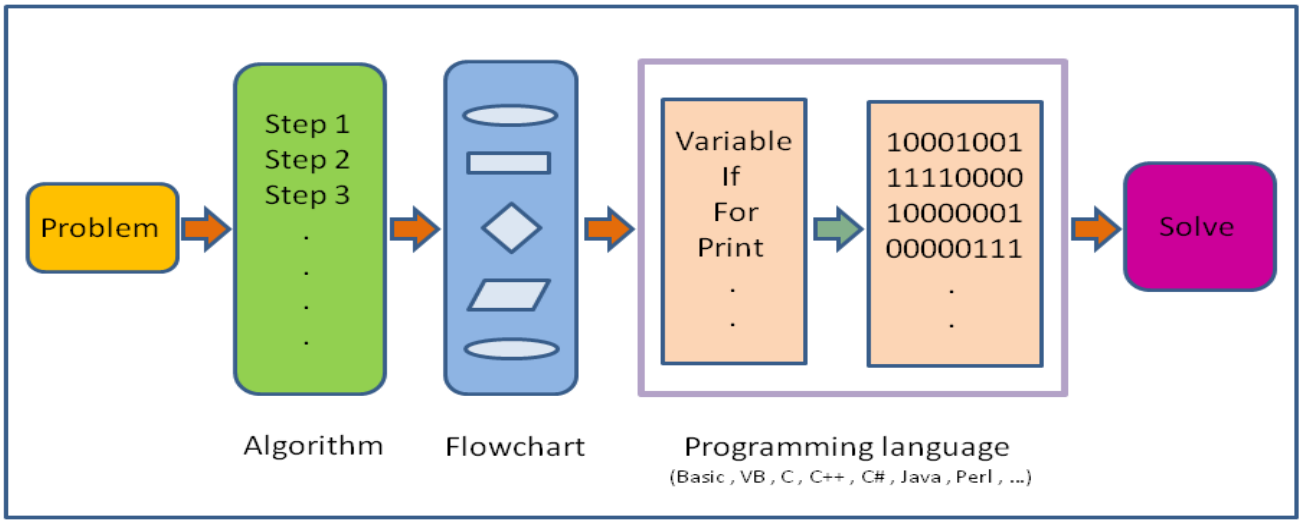
Interpreter: An interpreter translates and executes code line-by-line or statement-by-statement. Instead of producing a separate executable file, an interpreter directly executes the instructions in the source code. Examples of interpreted languages include Python, JavaScript, and Ruby.







Problem Solving:



Algorithm:
An algorithm is a step-by-step process, or a set of rules designed to perform a specific task or solve a particular problem. Algorithms serve as a blueprint for writing code, guiding the program through a sequence of operations to achieve a desired outcome.

Characteristics of an Algorithm:
Input: The algorithm takes zero or more inputs.
Output: The algorithm produces one or more outputs.
Definiteness: Each step of the algorithm is clear and unambiguous.
Finiteness: The algorithm must terminate after a finite number of steps.
Effectiveness: Each step of the algorithm is basic enough to be performed, ideally by a computer.

Examples:
Algorithm-1: Addition of Two Numbers
Step-1: Start
Step-2: Input: Read two numbers, num1 and num2.
Step-3: Process: Calculate the sum of the two numbers by performing the operation:
sum = num1 + num2
Step-4: Output: Display or return the result, sum.
Step-5: End

Algorithm-2: Finding Simple Interest
Step-1: Start
Step-2: Input: Read the values for P (Principal), R (Rate of Interest), and T (Time in years).
Step-3: Process:
Calculate the Simple Interest using the formula:
SI = (P * R * T) / 100
Step-4: Output: Display or return the result, SI.
Step-5: End

Algorithm-3: Finding the Biggest of Two Numbers
Step-1: Start
Step-2: Input: Read two numbers, num1 and num2.
Step-3: Process:
- If num1 > num2, then num1 is the biggest.
- Otherwise num2 is the biggest.

Step-4: Output: Display or return the biggest number.
Step-5: End

Algorithm-4: Check if a Number is Even or Odd
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- If num % 2 == 0, then the number is **even**.
- Otherwise, the number is **odd**.

Step-4: Output: Display whether the number is "Even" or "Odd".
Step-5: End

Algorithm-5: Finding the Factorial of a Given Number
Step-1: Start
Step-2: Input: Read the number n.
Step-3: Process:
- Initialize fact = 1.
- If n == 0 or n == 1, fact = 1 (since 0! = 1 and 1! = 1).
- For i = 2 to n,
  do fact = fact * i.

Step-4: Output: Display or return fact.
Step-5: End

Algorithm-6: Check if a Number is Prime
Step-1: Start
Step-2: Input: Read the number n.
Step-3: Process:
- If n <= 1, then the number is **not prime**.
- For i = 2 to sqrt(n):
  - If n % i == 0, then n is **not prime** (it has a divisor other than 1 and itself).
- If no divisors are found, n is **prime**.

Step-4: Output: Display or return whether the number is "Prime" or "Not Prime".
Step-5: End

Algorithm-7: Finding the Sum of Digits of a Given Number
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- Initialize sum = 0.
- While num > 0:
  - Extract the last digit using digit = num % 10.
  - Add the digit to sum.
  - Remove the last digit from num using num = num // 10 (integer division).

Step-4: Output: Display or return the value of sum.
Step-5: End

Algorithm-8: Check if a Number is an Armstrong Number
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- Initialize sum = 0.
- Determine the number of digits n in num.
- Initialize a variable temp to num (to preserve the original number).
- While temp > 0:
  - Extract the last digit using digit = temp % 10.
  - Calculate the power of the digit raised to n: power = digit^n.
  - Add the power to sum: sum = sum + power.
  - Remove the last digit from temp using temp = temp // 10.
- After the loop, compare sum with num.

Step-4: Output: Display whether the number is "Armstrong" or "Not Armstrong".

Step-5: End

Algorithm-9: Implement the Simple Calculator
Steps for the Algorithm:
Step-1: Start.
Step-2: Input two numbers (num1 and num2).
Step-3: Input an operator (+, -, *, /).
Step-4: Check the operator:

- If the operator is +, then calculate num1 + num2.
- If the operator is -, then calculate num1 - num2.
- If the operator is *, then calculate num1 * num2.
- If the operator is /, then:
  - Check if num2 is not zero (since division by zero is not allowed).
  - If num2 is not zero, calculate num1 / num2.
  - If num2 is zero, output an error message ("Division by zero error").
- If the operator is invalid (i.e., not one of +, -, *, /), output an error message ("Invalid operator").

Step-5: Output the result of the operation.
Step-6: End.

Algorithm-10: Check for the given year is leap year or not.
Step-1: Start.
Step-2: Input the year.
Step-3: Process: Check the leap year condition using a single logical statement:

- If the year is divisible by 400 **OR** (divisible by 4 **AND** not divisible by 100), then it is a leap year.
- Otherwise, it is not a leap year.

Step-4: Output the result ("Leap year" or "Not a leap year").
Step-5: End.

Problem Solving Approaches:
When solving problems, especially in computer science and software engineering, two common approaches are Top-Down and Bottom-Up. These approaches are widely used in algorithm design, dynamic programming, system development, and programming in general.
1. Top-Down Approach:
Definition: The **top-down approach** starts by breaking down a complex problem into smaller, more manageable sub-problems. The idea is to start from the highest level of abstraction and then progressively refine and decompose it into more specific components or steps.
Steps:

1. **Start with the big problem** and think about its general solution.
2. **Divide the problem** into smaller sub-problems.
3. **Solve each sub-problem** individually, refining each until you get to the smallest, most specific steps.
4. **Integrate the sub-problems** to solve the original, larger problem.

Advantages:

- Promotes clear, structured thinking.
- Easier to understand the big picture before diving into details.
- Helps in breaking down complex tasks, making them easier to manage.

Disadvantages:

- Can lead to redundant sub-problems if not careful (without optimization).
- Might overlook specific optimizations in lower-level components.

2. Bottom-Up Approach

Definition: The **bottom-up approach** starts by solving the simplest or smallest sub-problems first, and then building up solutions to more complex sub-problems by combining the solutions of the simpler ones.

Steps:

1. **Identify the base cases** or smallest sub-problems.
2. **Solve the smallest problems** first.
3. **Combine the solutions** of these smaller problems to build up to the solution of the larger problem.
4. **Repeat until** you reach the solution to the original, large problem.

Advantages:

- Avoids redundancy by solving each sub-problem once (more efficient).
- Often leads to more optimized solutions.
- Well-suited for **iterative** implementations.

Disadvantages:

- Can be harder to conceptualize, especially for large problems.
- May involve more initial work in defining and identifying base cases and how to combine them.

Examples of Usage

1. Fibonacci Sequence Calculation:

- **Top-Down (Recursive with Memoization):**
  - Break the problem as fib(n) = fib(n-1) + fib(n-2), and recursively calculate fib(n-1) and fib(n-2), while storing the results to avoid re-computation.
- **Bottom-Up (Iterative with Tabulation):**
  - Start with fib(0) and fib(1) and build up to fib(n) by iteratively solving from the base cases.

Time and Space Complexity of an algorithm:

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal.

There are two such methods used, **time complexity** and **space complexity**

Time Complexity: The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Ex-2: Given an array of n numbers, find if a specific number exists in the array.

Time Complexity Analysis:

Algorithm:

Step 1: Start

Step2: Initialize the loop: i = 0.

Step 2: for i=0 to n n-1:

Compare each element: arr[i] = target.

Continue until the element is found or until all elements have been checked.

Step3: if target found print 'YES'

Otherwise print 'NO'

Step 4: Stop

Time Complexity:

- **Best Case**: O(1) (if the target is at the beginning).
- **Worst Case**: O(n) (if the target is at the end or not present).
- **Explanation**: In the worst case, the algorithm iterates through every element of the array, so the time complexity is **O(n)**.

Space Complexity:

The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

Ex: Find the Space Complexity of an algorithm to find the element through linear search.

Space Complexity Analysis:

- The algorithm uses a few fixed variables:
    - arr[] (the input array) is given as input and does not affect the space complexity.
    - target (the element being searched for) is also given.
    - A counter variable i is used to traverse the array.

Since no additional memory is used other than these fixed variables, the space complexity remains constant, regardless of the array's size.

Space Complexity:

- **O(1)** (constant space).

C Programming:

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972. It is a very popular language, despite being old. The main reason for its popularity is because it is a fundamental language in the field of computer science.

C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

Features of C Programming:

1) Simple: C is a simple language in the sense that it provides a structured approach (to break the problem into parts), a rich set of library functions, data types, etc.

2) System Programming Language: C language is a system programming language because it can be used to do low-level programming (for example driver and kernel).

3) Structured programming language: C is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify. Functions also provide code reusability.

4) Mid-level programming language: Although, C is intended to do low-level programming. It is used to develop system applications such as kernel, driver, etc. It also supports the features of a high-level language. That is why it is known as a mid-level language.

5) Rich Library: C provides a lot of inbuilt functions that make the development fast.

6) Memory Management: It supports the feature of dynamic memory allocation. In C language, we can free the allocated memory at any time by calling the free() function.

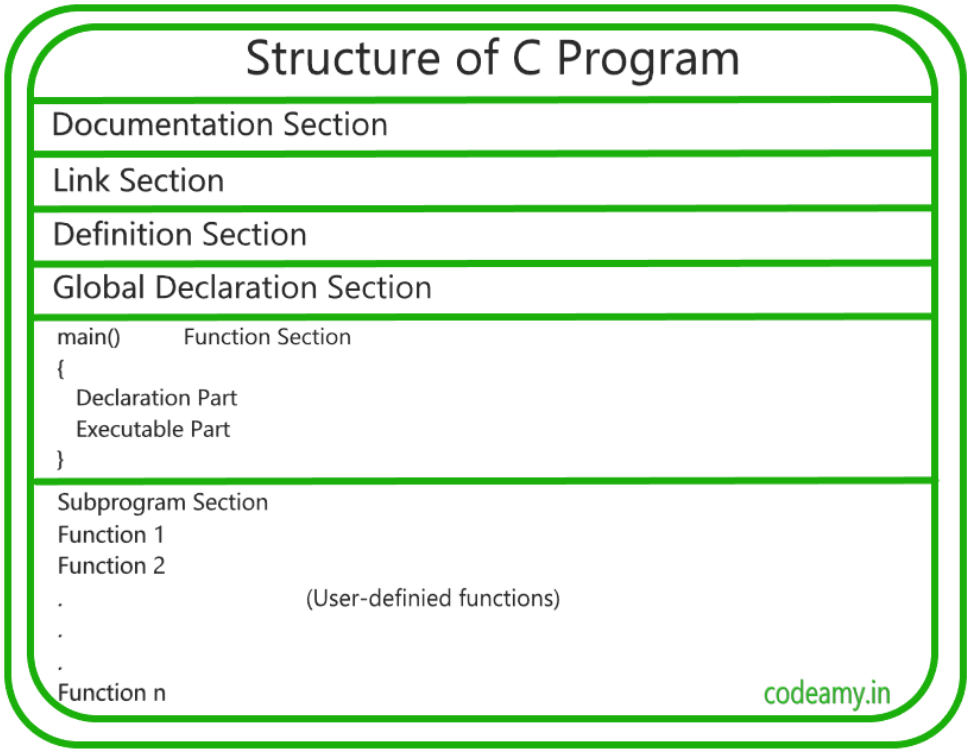
7) Pointer: C provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array, etc.

8) Extensible: C language is extensible because it **can easily adopt new features**.

9) Recursion: In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

Structure of C Program:

A The structure of a **C program** consists of several sections that follow a standard organization. While some parts are optional depending on the complexity of the program, every C program generally adheres to a similar format.



1) Documentation Section

This section consists of comment lines which include the name of the program, the name of the programmer, the author and other details like time and date of writing the program. Documentation section helps anyone to get an overview of the program.

```
/*Documentation Section:
  Program Name: Program to find the area of circle
  Author: M.Srinu
  Date : 12/09/2024
  Time : 10 AM
*/
```

2) **Link Section**

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library such as using the #include directive.

Example:

```
#include<stdio.h>                    //link section
#include<conio.h>                    //link section
```

3) Definition Section

All the symbolic constants are written in the definition section. Macros are known as symbolic constants.

```
#define PI 3.14                      //definition section
```

4) Global Declaration Section

The global variables that can be used anywhere in the program are declared in the global declaration section. This section also declares the user defined functions.

```
float area;                          //global declaration section
```

5) main() Function Section

It is necessary to have one main() function section in every C program. This section contains two parts, declaration and executable part. The declaration part declares all the variables that are used in executable part. These two parts must be written in between the opening and closing braces. Each statement in the declaration and executable part must end with a semicolon (;). The execution of the program starts at opening braces and ends at closing braces.

```
int main()
{
        float r;                                        //declaration part
        printf("Enter the radius of the circle\n"); //executable part start here
        scanf("%f",&r);
        area=PI*r*r;
        printf("Area of the circle=%f \n",area);
        message();
}
```

6) Subprogram Section

The subprogram section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.

If the program is a multifunction program, then the sub program section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

```
 void message()
{
        printf("This Sub Function \n");
        printf("we can take more Sub Function \n");
}
```

Program-001:

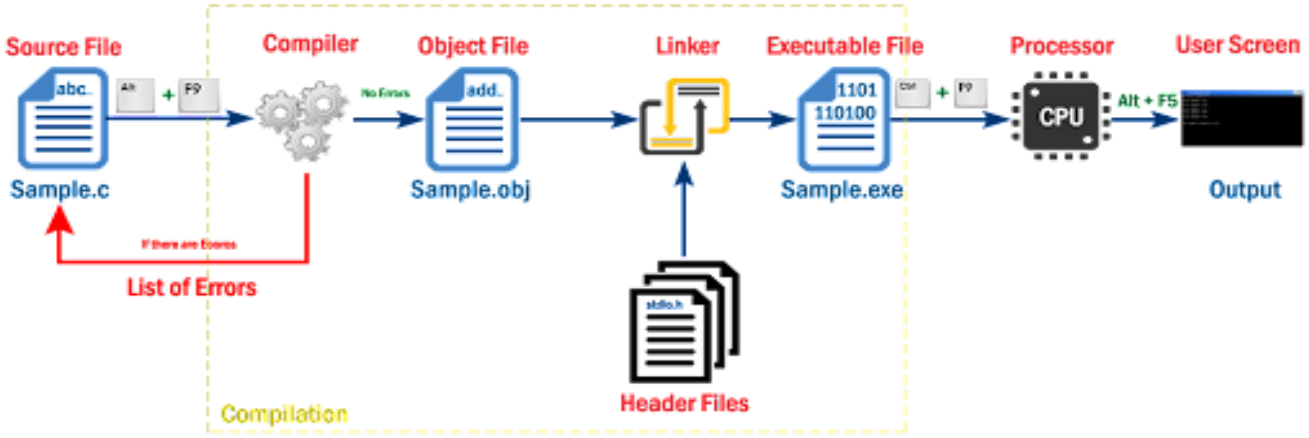
```
/*Documentation Section:
   Program Name: Program to find the area of circle
   Author: M.Srinu
   Date  : 12/09/2024
   Time  : 11 AM
*/
#include<stdio.h>                       //link section
#include<conio.h>                       //link section
#define PI 3.14                         //definition section
float area;                             //global declaration  section
void message();

int main()
{
 float r;                                       //declaration part
 printf("Enter the radius of the circle\n");    //executable part
 scanf("%f",&r);
 area=PI*r*r;
 printf("Area of the circle=%f \n",area);
 message();
}
void message()
{
        printf("This Sub Function \n");
        printf("we can take more Sub Functions \n");
}
```

Program-002:

```
#include                /* Link section */
int total=0 ;           /* Global declaration, definition section */
int sum(int,int);       /* Function declaration section */
int main()              /* Main function */
{
        printf("C programming basics & structure of C programs \n");
        total=sum(6,6);
        printf("sum=%d\n",total);
}
```

```
int sum(int a, int b) /* User defined function */
{
        return a+b;
}
```

Compilation and Execution of C Program:



Token:

A token in C can be defined as the smallest individual element of the C programming language that is meaningful to the compiler. It is the basic component of a C program.

Types of Tokens in C

The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:

Keywords:

- The keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program.
- Keywords are the words whose meaning has already been explained to the C compiler and their meanings cannot be changed.
- Keywords can be used only for their intended purpose.
- Keywords cannot be used as user-defined variables.
- All keywords must be written in lowercase.

C language supports **32** keywords which are given below:

32 Keywords in C Programming Language

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

Identifiers:

Identifiers are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character.

Certain rules should be followed while naming c identifiers which are as follows:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only the first 31 characters are significant.

Ex: int a,b,c; //where a,b and c are identifiers and int a keyword.

Constants:

The constants refer to the variables with fixed values. They are like normal variables but with the difference that their values cannot be modified in the program once they are defined. Constants may belong to any of the data types.

Ex: const int x=10;

const float pi=3.142f;

Strings:

Strings are nothing but an array of characters ended with a null character ('\0'). This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C.

Ex: char branch[20]={'C','S','E'};

char name[]="Rajesh";

Special Symbols:

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose. Some of these are listed below:

- **Brackets[]:** Opening and closing brackets are used as array element references. These indicate single and multidimensional subscripts.

- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **Comma (,):** It is used to separate more than one statement like for separating parameters in function calls.
- **Colon(:):** It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;):** It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*):** It is used to create a pointer variable and for the multiplication of variables.
- **Assignment operator(=):** It is used to assign values and for logical operation validation.
- **Pre-processor (#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.
- **Period (.):** Used to access members of a structure or union.
- **Tilde(~):** Bitwise One's Complement Operator.

Operators:

Operators are symbols that trigger an action when applied to C variables The data items on which operators act are called operands.

Depending on the number of operands that an operator can act upon, operators can be classified as follows:

Unary Operators: Those operators that require only a single operand to act upon are known as unary operators. For example, **increment and decrement** operators

Binary Operators: Those operators that require two operands to act upon are called binary operators. Binary operators can further be classified into:

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operator

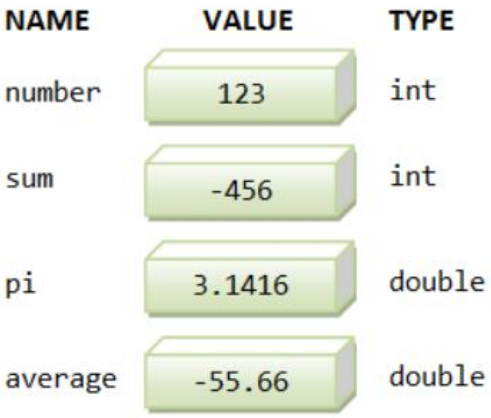
Ternary Operator: The operator that requires three operands to act upon is called the ternary operator. Conditional Operator(?) is also called the ternary operator.

Variable:

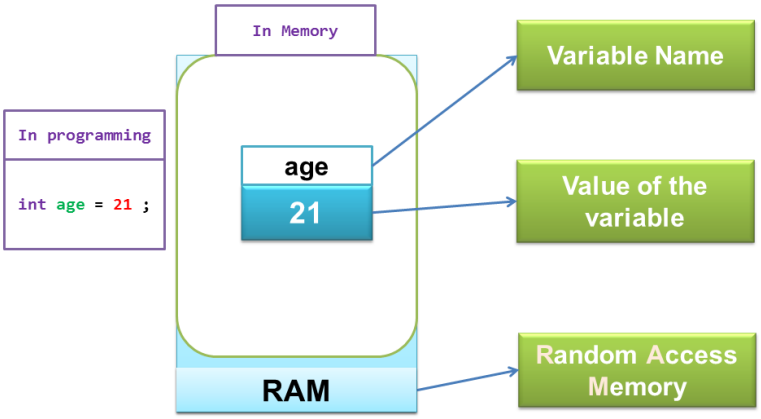
Variable is a name of the memory location where the actual value is to be stored. A **variable in C** is a memory location with some name that helps store some form of data and retrieves it when required.

Syntax:

```
        datatype variable_name;          //Variable declaration
        datatype variable_name=value;    //Variable initialization
Ex:     int number=123,sum=-456;
        double pi=3.1416,average=-55.66;
```



A variable has a *name*, stores a *value* of the declared *type*



The C variables can be classified into the following types:

1. Local Variables
2. Global Variables
3. Static Variables
4. Automatic Variables
5. Extern Variables
6. Register Variables

1) Local Variables:

A **Local variable in C** is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

Program-003:

```
#include <stdio.h>
void function();
int main()
{
    function();
}
void function()
{
  int x = 10;        // local variable
  printf("%d", x);
}
```

Output:

10

2) Global Variables:

A Global variable in C is a variable that is declared outside the function or a block of code. Its scope is the whole program i.e. we can access the global variable anywhere in the C program after it is declared.

Program-004:

```
// C program to demonstrate use of global variable
#include <stdio.h>
void function1();
void function2();
int x = 20;                          // global variable
int main()
{
   function1();
   function2();
   return 0;
}
void function1()
{
     printf("Function 1: %d\n", x);
}


void function2()
{
     printf("Function 2: %d\n", x);
}
```

3) Static Variables:

A static variable in C is a variable that is defined using the static keyword. It can be defined only once in a C program and its scope depends upon the region where it is declared (can be global or local).

The default value of static variables is zero.

Syntax:

```
static data_type variable_name = initial_value;
```

As its lifetime is till the end of the program, it can retain its value for multiple function calls as shown in the example.

Program-005:

```
// C program to demonstrate use of static variable
#include <stdio.h>
void function()
{
   int x = 20;                  // local variable
   static int y = 30;           // static variable
   x = x + 10;
   y = y + 10;
   printf("\tLocal Variable X: %d\n\tStatic Variable Y: %d\n", x, y);
}
int main()
{
```

```
    printf("First Call\n");
    function();
    printf("Second Call\n");
    function();
    printf("Third Call\n");
    function();
    return 0;
}
```

Output:

```
First Call
    Local: 30
    Static: 40
Second Call
    Local: 30
    Static: 50
Third Call
    Local: 30
    Static: 60
```

4) Automatic Variable:

All the local variables are automatic variables by default. They are also known as auto variables. Their scope is local, and their lifetime is till the end of the block. If we need, we can use the auto keyword to define the auto variables.

The default value of the auto variables is a garbage value.

Syntax: auto data_type variable_name;

or

data_type variable_name; //(in local scope)

Program-006:

```
// C program to demonstrate use of automatic variable
#include <stdio.h>
void function()
{
    int x = 10;              // local variable (also automatic)
    auto int y = 20;         // automatic variable
    printf("Auto Variable: %d", y);
}
int main()
{
    function();
    return 0;
}
```

Output:

```
Auto Variable: 20
```

Note: In the above example, both x and y are automatic variables. The only difference is that variable y is explicitly declared with the auto keyword.

5) External Variables:

External variables in C can be shared between multiple C files. We can declare an external variable using the extern keyword. Their scope is global and they exist between multiple C files.

Syntax: extern data_type variable_name;

Example:

Program-007:

File1: myFile1.h

```
extern int x=10;                    //external variable (also global)
```

File2: myFile2.C

```
#include "myfile.h"
#include <stdio.h>
void printValue()
{
        printf("Global variable: %d", x);
}
int main()
{
        printValue();
}
```

Output:

Global variable: 10

6) Register Variables:

Register variables in C are those variables that are stored in the CPU register instead of the conventional storage place like RAM. Their scope is local and exists till the end of the block or a function. These variables are declared using the register keyword.

The default value of register variables is a garbage value.

Syntax:

register data_type variable_name = initial_value;

Example:

Program-008:

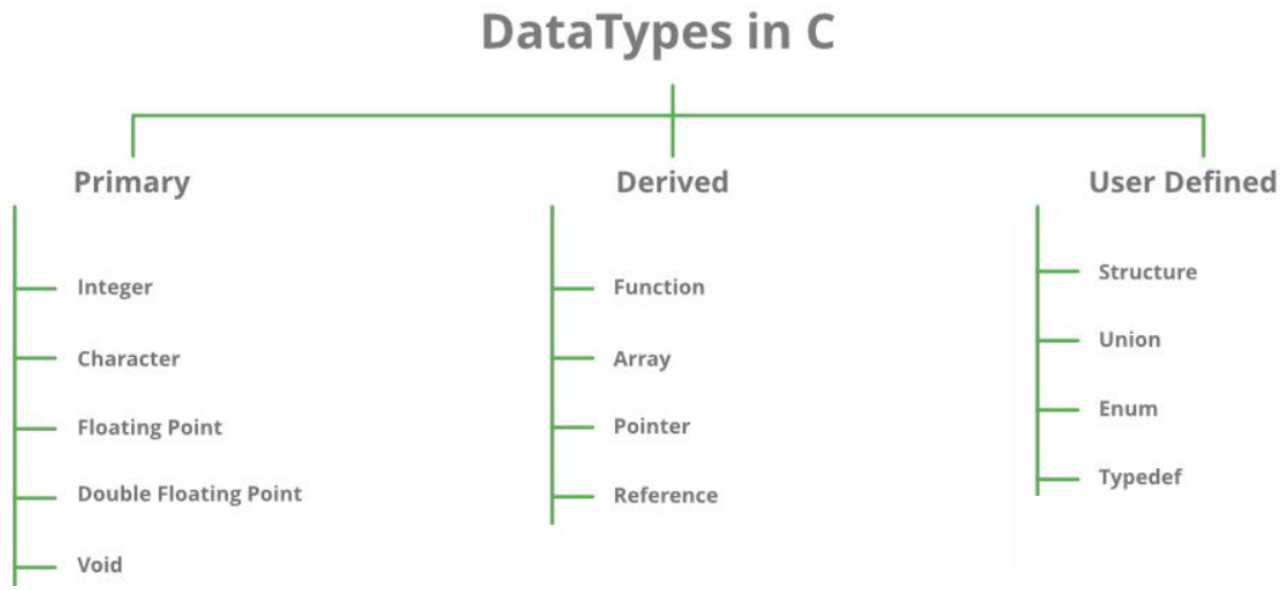
```
// C program to demonstrate the definition of register variable
#include <stdio.h>
int main()
{
        register int var = 22;
        printf("Value of Register Variable: %d\n", var);
        return 0;
}
```

Storage Specifier	Storage	Initial value	Scope	Life
auto	Stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Data types:

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it.

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.



Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
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
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Datatype Modifiers:
Modifiers are keywords in C which changes the meaning of basic data type in C. It specifies the amount of memory space to be allocated for a variable. Modifiers are prefixed with basic data types to modify the memory allocated for a variable.

Need of Datatype Modifiers:
We use int to store the Salary of the employee as we are assuming that the salary will be in whole numbers. An integer data type takes 4 bytes of memory, and we are aware that the Salary of any of the employee cannot be "Negative".
We are using "4 Bytes" to store the salary of an employee and we can easily save 2 Bytes over there by removing the "Signed Part" in the integer. This leads us to the use of Data Type Modifiers.

Types of Datatype Modifiers:
- Signed
- Size
- Const

Signed modifier:
All data types are "signed" by default. Signed modifier implies that the data type variable can store positive values as well as negative values. For example, if we need to declare a variable to store temperature, it can be negative as well as positive.

signed int temperature; Or int temperature;

If we need to declare a variable to store the salary of an employee, we will use "Unsigned" Data modifier.

unsigned int salary;

Size Modifier:
Sometimes we need to increase the Storage Capacity of a variable so that it can store values higher than its maximum limit which is there as default.
We need to make use of the **"long"** data type qualifier. **"long"** type modifier doubles the "length" of the data type when used along with it.
For example, if we need to store the "annual turnover" of a company in a variable, we will make us of this type qualifier.

long int turnover;

A **"short"** type modifier does just the opposite of "long" with 2 bytes in memory. If one is not expecting to see high range values in a program and the values are both positive & negative.

For example, if we need to store the "age" of a student in a variable, we will make use of this type qualifier as we are aware that this value is not going to be very high.

short int age;

Const modifier:

In C all variables are by default not constant. Hence, you can modify the value of variable by program. You can convert any variable as a constant variable by using modifier const which is keyword of C language.

Properties of constant variable:

You can assign the value to the constant variables only at the time of declaration.

For example:

const int i=10; float const f=0.0f; unsigned const long double ld=3.14L;

Uninitialized constant variable is not cause of any compilation error. But you cannot assign any value after the declaration.

Example: const int i;

Note: The long, short, signed and unsigned are datatype modifiers that can be used with some primitive data types to change the size or length of the datatype.

Literal:

In C, Literals are the constant values that are assigned to the variables. Literals represent fixed values that cannot be modified.

There are 4 types of literals in C:

- **Integer Literal**
- **Float Literal**
- **Character Literal**
- **String Literal**

Integer literals are used to represent and store the integer values only. Integer literals are expressed in two types i.e,

A) Prefixes: The Prefix of the integer literal indicates the base in which it is to be read.

a. **Decimal-literal(base 10):** A **non-zero decimal digit** followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Ex: 15, 29 etc…

b. **Octal-literal(base 8):** a **0** followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7). **Ex:** 056, 0123 etc..

c. **Hex-literal(base 16): 0x** or **0X** followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F). **Ex:** 0x5A, 0XAABB etc….

d. **Binary-literal(base 2): 0b** or **0B** followed by one or more binary digits(0, 1). **Ex:** 0b101, 0B11011 etc…

B) Suffixes: The Suffixes of the integer literal indicates the type in which it is to be read. These are represented in many ways according to their data types.

int: No suffix is required because integer constant is by default assigned as an int data type.

unsigned int: character u or U at the end of an integer constant.

long int: character l or L at the end of an integer constant.

unsigned long int: character ul or UL at the end of an integer constant.

long long int: character ll or LL at the end of an integer constant.

unsigned long long int: character ull or ULL at the end of an integer constant.

Real Or Floating-Point Literal:
Integer numbers are inadequate to represent quantities that vary continuously such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like17.548. Such numbers are called real or floating-point constants.
Examples: 0.0083, -0.75, 435.36, 2e-8, 0.006e-2
Character Literal: Literal that is used to store a single character within a single quote.
Ex: char ch='A', ch1='9', ch2='+' etc….
String Literal: String literals are like that of character literals, except that they can store multiple characters and use a double quote to store the same. A string constant is a sequence of characters enclosed with in double quotes. The characters may be letters, numbers, special characters and blank space. Example: "Hello", "1994", "Well done", "5+3", "M".
Ex: char ch[]={'A','D','I','T','Y','A'};

Input and Output:
C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output.
scanf(): The scanf() method, in C, reads the value from the console as per the type specified and store it in the given address.

Syntax: int scanf("*Control String*",*&arg1,&arg2,…*);

The return value of scanf function is the number of successful data inputs
printf(): The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax: int printf ("*Control String*", arguments_list);

The return value of printf function is the total number of characters printed is returned. On failure, a negative number is returned.
The format specifier in C is used to tell the compiler about the type of data to be printed or scanned in input and output operations. They always start with a **%** symbol and are used in the formatted string in functions like printf(), scanf etc.

Format Specifier	Description
%c	For character type.
%d	For signed integer type.
%e or %E	For scientific notation of floats.
%f	For float type.
%i	signed integer
%ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or %lld	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

Examples on printf():

printf("Welcome to C Programming\n"); //**Output:** Welcome to C Programming
int a=10, b=20; printf("%d %d\n",a,b); //**Output:** 10 20
int a=10,b=20,c=a+b; printf("sum of %d and %d is: %d\n",a,b,c);//**Output**:sum of 10 and 20 is: 30

Examples on Scanf():

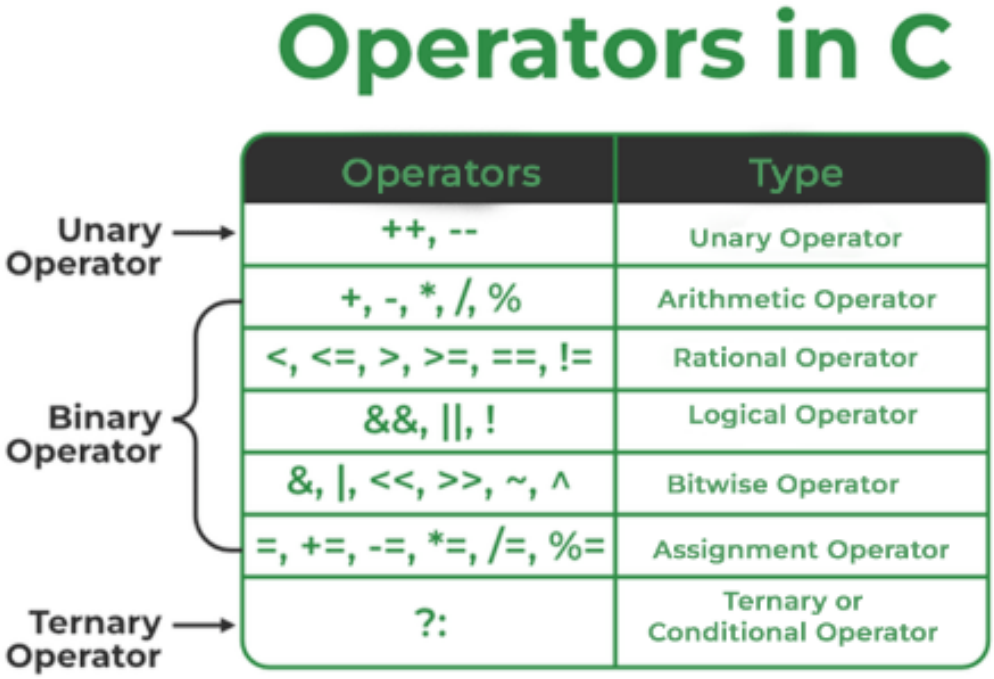
int n; scanf("%d",&n); **//Read an integer value**
int a,b,c; scanf("%d %d %d",&a,&b,&c); **//Read there integer values**
float p,q; scanf("%f %f",&p,&q); **//Read two float values.**
Char ch; scanf("%c",&ch); **//Read a Character**

/* Program to find the Simple Interest*/

```
#include<stdio.h>
int main()
{
        int P,T,R;
        float I;
        printf("Enter Principle, Time and Rate of interest\n");
        scanf("%d%d%d",&P,&T,&R);
        I=(P*T*R)/100;
        printf("P = %d  T = %d R = %d\n",P,T,R);
        printf("Simple Interest = %f",I);
        return 0;
}
```

Operators:

In C language, operators are symbols that represent operations to be performed on one or more operands. C has a wide range of operators to perform various operations.



Types of Operators in C

C language provides a wide range of operators that can be classified into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

1) Arithmetic Operators:

The arithmetic operators are used to perform arithmetic/mathematical operations on operands. Those are +, -, *, / and %.

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
   int a = 9,b = 4, c;
   c = a+b;
   printf("a+b = %d \n",c);
   c = a-b;
   printf("a-b = %d \n",c);
   c = a*b;
   printf("a*b = %d \n",c);
   c = a/b;
   printf("a/b = %d \n",c);
   c = a%b;
   printf("Remainder when a divided by b = %d \n",c);
   return 0;
}
```

Output:

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

2) Relational Operators:

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

```
// Working of relational operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;
   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
```

```
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);
    return 0;
}
```

Output:

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

3) Logical Operators:

Logical Operators are used to combine two or more conditions to make complex expressions. An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

// Working of logical operators

```
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;
   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) is %d \n", result);
   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) is %d \n", result);
   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) is %d \n", result);
   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) is %d \n", result);
   result = !(a != b);
```

```
    printf("!(a != b) is %d \n", result);
    result = !(a == b);
    printf("!(a == b) is %d \n", result);
    return 0;
}
```

Output:

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

4) Bit wise Operators:

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

There are 6 bitwise operators in C

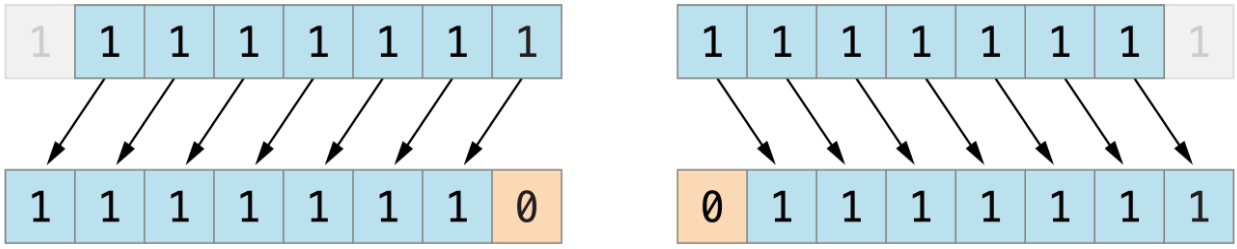
S. No.	Symbol	Operator	Syntax
1	&	Bitwise AND	a & b
2	|	Bitwise OR	a | b
3	^	Bitwise XOR	a ^ b
4	~	Bitwise One's Complement	~a
5	<<	Bitwise Left shift	a << b
6	>>	Bitwise Right shilft	a >> b

Table *Evaluation of bitwise operators on 1 bit values*

Inputs		and	or	xor
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Input	not
a	~a
0	1
1	0

In bitwise shift operations, the digits are moved, or shifted, to the left or right.



// C program to illustrate the bitwise operators

```
#include <stdio.h>
int main()
{
   int a = 25, b = 5;
   printf("a & b: %d\n", a & b);
   printf("a | b: %d\n", a | b);
   printf("a ^ b: %d\n", a ^ b);
   printf("~a: %d\n", ~a);
   printf("a >> b: %d\n", a >> b);
   printf("a << b: %d\n", a << b);
   return 0;
}
```

Output:

```
a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800
```

5) Assignment Operators:

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right-side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

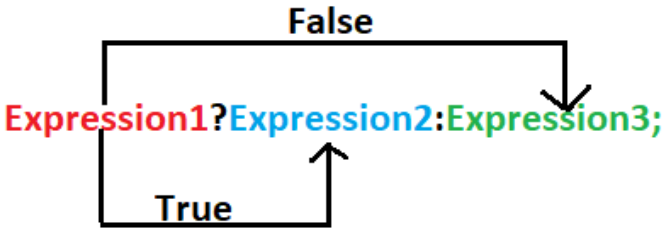
S. No.		Symbol	Operator	Syntax	Equalent to
1		=	Simple Assignment	a = b	a = b
2	Short hand Operators	+=	Plus and assign	a += b	a=a+b
3		-=	Minus and assign	a -= b	a=a-b
4		*=	Multiply and assign	a *= b	a=a*b
5		/=	Divide and assign	a /= b	a=a/b
6		%=	Modulus and assign	a %= b	a=a%b
7		&=	AND and assign	a &= b	a=a&b
8		|=	OR and assign	a |= b	a=a|b
9		^=	XOR and assign	a ^= b	a=a^b
10		>>=	Rightshift and assign	a >>= b	a=a>>b
11		<<=	Leftshift and assign	a <<= b	a=a<<b

6) Other Operators:

a) Conditional Opertor(?:)

The conditional operator is the only ternary operator in C.

Syntax: Expression1 ? Expression2 : Expression3



Here, Expression1 is the condition to be evaluated first. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
Note: We may replace the use of if..else statements with conditional operators.

Example:

```
#include<stdio.h>
int main()
{
    int x,y;
    scanf("%d%d",&x,&y);
    int max=x>y ? x : y;
    printf("Maximum of %d and %d is: %d",x,y,max);
    return 0;
}
```

Input:

20 36

Output:

Maximum of 20 and 36 is: 36

b) **sizeof():** Basically, the sizeof the operator is used to compute the size of the variable or datatype. The result of sizeof is of the unsigned integral type which is usually denoted by size_t.

Example:

```
#include <stdio.h>
int main()
{
  int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));
   return 0;
}
```

Output:

Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte

c) Comma(,), dot(.), addressof(&), dereference(*), arrow(->) etc….

Operator Precedence and Associativity:

Precedence: The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

For Example:

10+20*30 => 610 [Right]
10+20*30 => 900 [Wrong]

Note: Precedence tells which operator is to be executed first in an expression.

Associativity: It defines the order in which operators of the same precedence are evaluated in an expression. Associativity can be either from left to right or right to left.

EXAMPLE:	Example:
Let a=30, b=10, c=11, d=5, e=10 With the above values evaluate the following expression A%3 – b/2 + (c*d-5)/e SOLUTION: A%3 – b/2 + (c*d-5)/e Substitute the values in the expression As per hierarchy rules the sub-expression with in parenthesis will be first evaluated. 30 % 3 – 10/2 + (11*5-5)/10 =30 %3 – 10/2 + (55-5)/10 (operation with in parenthesis) =30 %3 – 10/2 + 50/10 (operation with in parenthesis) =0 -10/2 + 50/10 (% operation) =0 – 5 + 50/10 (/ operation) =0 – 5 + 5 (/operation) =-5 + 5 (- operation) =0 (+operation)	(3 * 4) / 2 + 4 / 2. Solution: (3 * 4) / 2 + 4 / 2. =12/2 + 4/2(operation with in parentheses) = 6 + 4/2 (/ operation) = 6 + 2 (/ operation) = 8 (+ operation)

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Type Conversion, and Casting:

Conversion of the value of one data type to another type is known as **type conversion**.

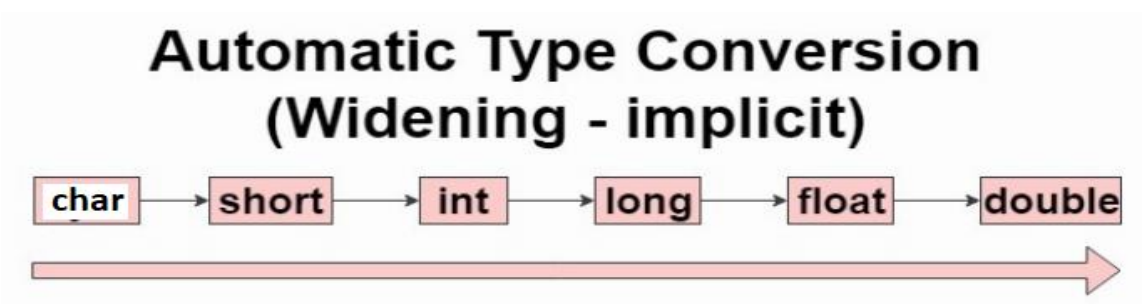
There are two types of conversion in C:

- **Implicit Conversion** (automatically)
- **Explicit Conversion** (manually)

1) Implicit Conversion:

Implicit Type Conversion is also known as **'automatic type conversion'**. It is done by the compiler on its own, without any external trigger from the user.

In implicit typecasting, the conversion involves a **smaller data type to the larger data type.**

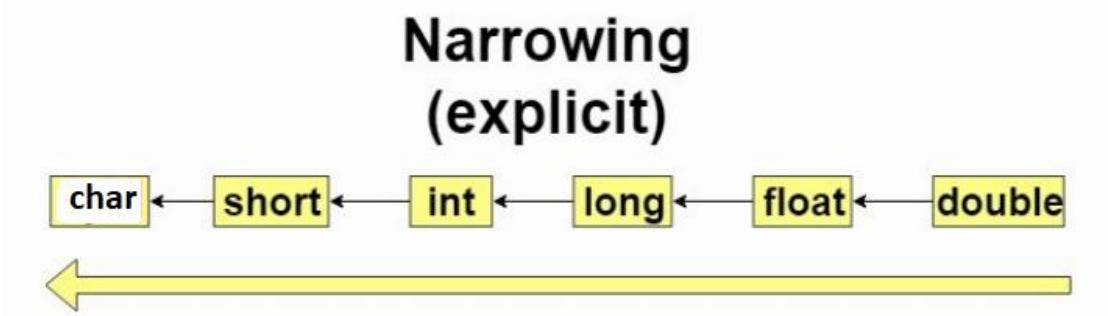


```
#include<stdio.h>
int main()
{
        int x=10;
        char y='a';
        float z;
        x=x+y;              //y implicitly converted to int. ASCII value of 'a' is 97
        z=x+1.0;
        printf("x=%d,z=%f",x,z);
        return 0;
}
```

Output:
x=107, z=108.000000

2) Explicit Conversion

This conversion is done by user. This is also known **as typecasting**. In this one data type is converted into another data type forcefully by the user. In explicit typecasting, the conversion involves a larger data type to the smaller data type.



Here is the syntax of explicit type casting in C language,

Syntax:

(datatype) Variable/expression

Ex:

float salary = 10000.00;

int sal= **(int)** salary;

Example:

```
#include<stdio.h>
int main()
{
        float x=1.2;
        int sum;
        sum=(int)x+1;
        printf("sum=%d",sum);
        return 0;
}
```

Output:

Sum=2

Important Questions:

1. Differentiate Hight Level, Middle Level and Low-Level Programming Languages.
2. Define an algorithm. List the characteristics of an algorithm.
3. List and explain the programming strategies or approaches.
4. What is a Time and Space complexity of an algorithm.
5. Differentiate Compiler, Assembler and interpreter.
6. What is the structure of C program and explain in detail of each section.
7. Describe the following
   a) C Character set b) Token c) Identifier d) Variable e) Constant
8. What is data type and explain different data types supported by C.
9. Define an operator. List and describe different types of operators in C programming with an example program.
10. Explain Operator Precedence and Associativity with an example program.
11. Demonstrate the use of printf() and scanf() with an example programs.
12. What is type casting? Demonstrate with an example program.

Programs to Practice:

1. Program to find the simple interest.
2. Program to find the conversion from Fahrenheit to Celsius.
3. Program to find the distance between two points.
4. Program to find the square root of a given number.
5. Program to find the area of Triangle using Heron's formula.
6. Program to find the BMI value.

Practice the flow chart design for above programs also using dia tool.

MCQ Question and Answers for Practice

1. Which of the following is NOT a characteristic of a good algorithm?

a) Finiteness **b) Ambiguity** c) Input d) Output

2. What does it mean for an algorithm to be finite?

a) It produces an incorrect output. b) It runs indefinitely without stopping.

c) It completes after a certain number of steps. d) It doesn't require any input.

3. Which of the following characteristics defines the performance efficiency of an algorithm?

a) Time complexity b) Uniqueness c) Input size d) Clarity

4. What is meant by the 'definiteness' characteristic of an algorithm?

a) The algorithm must be written in English

b) Each step of the algorithm must be clear and unambiguous

c) The algorithm should handle multiple tasks at once d) It must run forever

5. Which of the following is an example of a high-level programming language?

a) Assembly **b) Python** c) Machine code d) Binary

6. Which of the following best describes a low-level language?

a) Provides close control over hardware b) Easy to read and write

c) Independent of hardware architecture d) Automatically manages memory

7. What is a key feature of a middle-level programming language?

a) It is highly abstracted from hardware b) It only works on specific hardware architectures

c) It combines features of both low-level and high-level languages

d) It does not require compilation

8. Which of these languages is considered a middle-level language?

a) Assembly b) Python **c) C** d) JavaScript

9. What symbol is used to represent the start or end of a flowchart?

a) Rectangle b) Diamond **c) Oval** d) Parallelogram

10. Which symbol is used to represent a decision point in a flowchart?

a) Rectangle **b) Diamond** c) Oval d) Circle

11. In a flowchart, what does the rectangle symbol represent?

a) Input/output b) Start/End **c) Process or action step** d) Decision

12. What is the purpose of a flowchart?

a) To write code for a program **b) To visually represent the flow of a process or algorithm**

c) To store data for a program d) To manage memory in a system

13. Which of the following symbols represents input/output in a flowchart?

a) Rectangle **b) Parallelogram** c) Diamond d) Oval

14. Which of the following is not a valid C data type?

a) int b) float c) char **d) string**

15. What is the size of an int data type in C on a 32-bit system?

a) 2 bytes **b) 4 bytes** c) 8 bytes d) 1 byte

16. Which of the following data types can store a value of 3.14159?

a) int b) char **c) float** d) unsigned int

17. Which format specifier is used to print a char value in C?

a) %d b) %f **c) %c** d) %s

18. What is the range of values that can be stored in an unsigned char?

a) -128 to 127 **b) 0 to 255** c) 0 to 128 d) -255 to 255

19. What is the default data type of a floating-point number in C?

a) double b) float c) long double d) int

20. Which of the following is the correct format specifier for a double in C?

a) %d **b) %lf** c) %c d) %u

21. What is the size of a char data type in C?

a) 1 byte b) 2 bytes c) 4 bytes d) 8 bytes

22. Which data type would you use to store large integers (greater than int)?

a) float b) double **c) long int** d) char

23. Which of the following is not a C token?

a) Keyword b) Identifier c) Constant **d) Variable**

24. How many types of C tokens are there?

a) 3 b) 5 **c) 6** d) 8

25. Which of the following is a valid identifier in C?

a) 1variable b) $sum **c) total_sum** d) float

26. Which symbol is used to terminate a statement in C?

a) . b) , c) : **d) ;**

27. Which of the following is a valid C operator?

a) & b) @ c) # d) %&

28. Which token is used for preprocessor directives in C?

a) $ **b) #** c) @ d) %

29. Which of the following is a string constant in C?

a) 'C' **b) "C Programming"** c) 100 d) True

30. What is typecasting in C?

a) Changing the variable's name **b) Converting one data type to another**

c) Assigning a variable to a constant d) Comparing two variables

31. Which of the following is an example of explicit typecasting?

a) float a = 10.5; b) int b = 3.14; **c) int c = (int)4.5;** d) double d = 5;

32. Check the output for the following code: int a = 10;

float b = (float)a / 4;

printf("%f", b);

a) 2.0 **b) 2.5** c) 2.75 d) 2.0f

33. Which of the following conversions is NOT allowed in C?

a) int to float b) float to int c) char to int **d) string to int**

34. What happens during implicit type conversion?

a) The programmer must specify the conversion b) Data is lost in the conversion process

c) The compiler automatically converts one data type to another d) It causes a compilation error

35. Which of the following is a valid variable declaration in C?

a) int 1stNumber; **b) float number;** c) char $name; d) double my variable;

36. What is the default value of an uninitialized static variable in C?

a) 0 b) Undefined c) Garbage value d) NULL

37. Which of the following is a constant in C?

a) #define PI 3.14 b) int a = 5; c) float b = 10.0; d) char name = 'A';

38. What will happen if you try to change the value of a constant defined using const?

a) Compilation will succeed b) The program will terminate

c) A compilation error will occur d) The constant will be changed

39. Which of the following statements about variables in C is FALSE?

a) Variables must be declared before they are used.

b) Variable names can contain digits, but must not start with one.

c) Variables can be assigned new values at any point in the program.

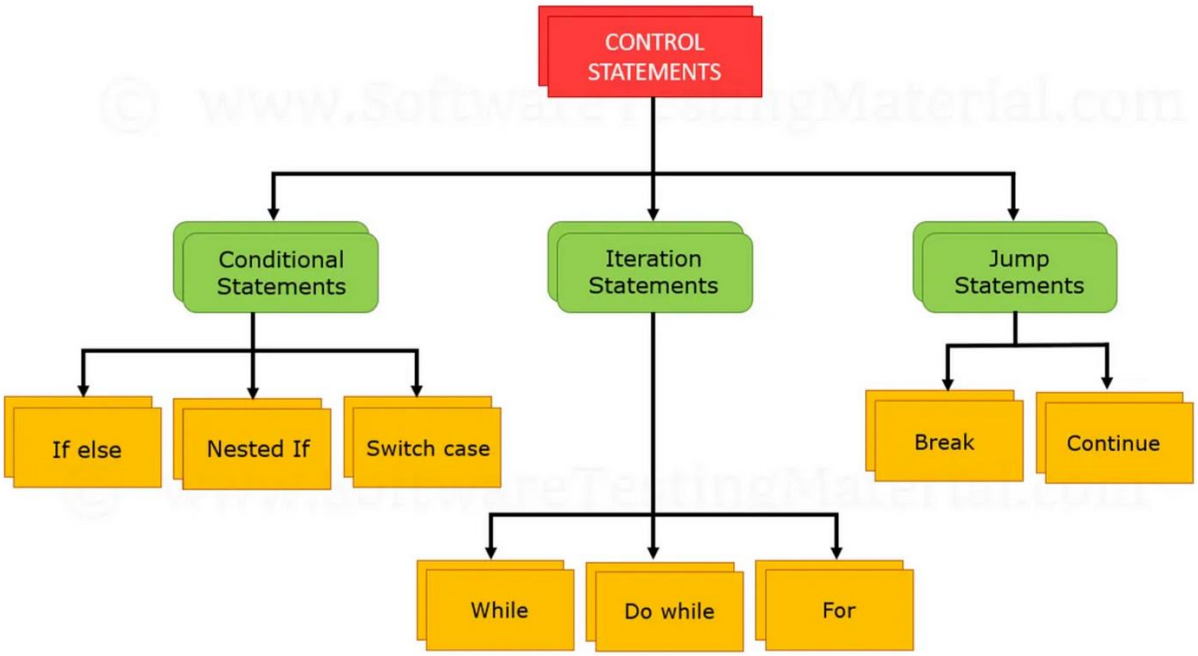
d) Variable names can be the same as C keywords.

UNIT – II

Control Structures

Simple sequential programs Conditional Statements (if, if-else, else if ladder, switch), Loops (for, nested for loop , while, do-while) break and continue , goto statement.

Control statements are an essential aspect of programming languages like C, as they allow programmers to control the flow of execution of their programs. In C, there are three types of control statements: **selection statements, iteration statements, and jump statements**.

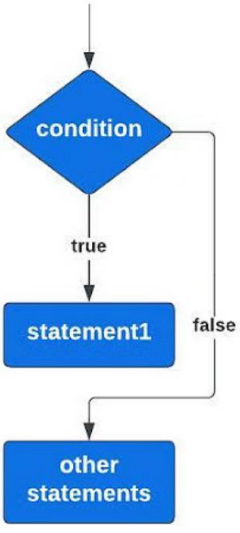


1. Conditional Statements or Decision-Making Control Statements:

Conditional statements help in decision-making within the program by allowing the execution of certain blocks of code depending on the result of a condition. In C programming, **conditional statements** are used to perform different actions based on whether a specified condition is true or false. The most common conditional statements in C are:

a) **Simple if:** Executes a block of code if a condition is true. If the condition of the if statement is true, then the statements under the if block is executed else the control is transferred to the statements outside the if block.

```
if (condition)
{
  Statement1;
}
```

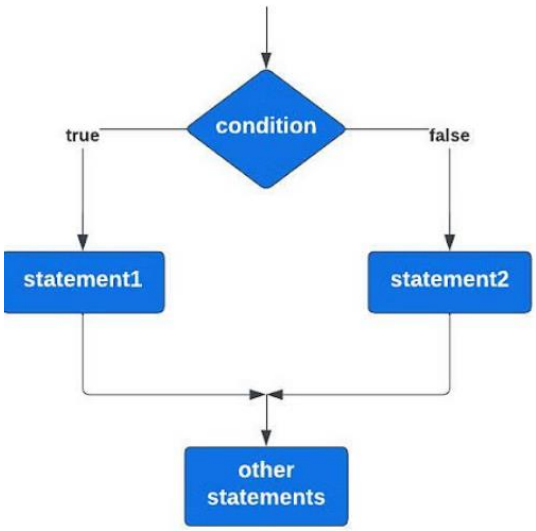


Program2_1: Program to find the given number is positive or not.

```
#include<stdio.h>
int main()
{
  int n;
  scanf("%d",&n);
  if(n>0)
       printf("Given number %d is a positive number\n",n);
  return 0;
}
```

b) if else: Executes one block of code if the condition is true, and another block if the condition is false.

```
if (condition)
  Statement1;
else
  Statement2;
```



Program2_2: program to check if a person is eligible to vote or not based on their age.

```
#include<stdio.h>
int main()
{
        int age;
        printf("Enter your Age:\n");
        scanf("%d",&age);
        if(age>=18)
        {
                printf("Hi! You are eligible for Voting\n");
                printf("Thank you for using my applicaiton\n");
        }
        else
        {
                printf("Sorry! You are not eligible for Voting\n");
                printf("You need to wait %d more years to get the vote\n",18-age);
        }
        return 0;
}
```

Program2_3: Write a program that checks whether a triangle is valid or not based on the angles provided (sum of angles should be 180 degrees).

Input:

45 45 90

Output:

Valid Triangle

Code:

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

	printf("Enter angles of the Triangle\n");
	scanf("%f%f%f",&a,&b,&c);
	sum=a+b+c;
	if(sum==180)
	{
		printf("It is a Valid Triangle");
	}
	else
	{
		printf("It's not a Valid Triangle");
	}
	return 0;
}
```

Program2_4: Program to find the given character is uppercase or lowercase.

```
#include<stdio.h>
int main()
{
	char ch;
	scanf("%c",&ch);
	//Assume the input contains only Alphabets
	if(ch>='A' && ch<='Z')
  {
	printf("%c is a Uppercase Character\n",ch);
	}
	else
	{
		printf("%c is a Lowercase Character\n",ch);
	}
	return 0;
}
```

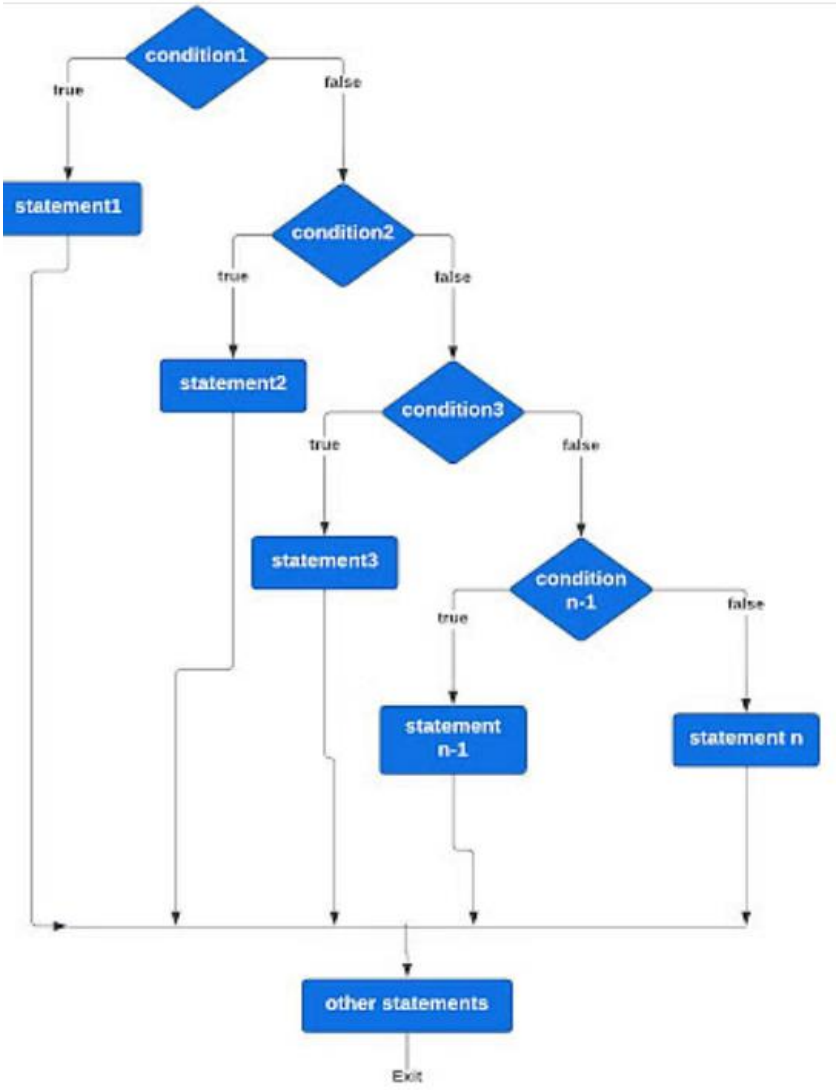
Input: U

Output: U is a Uppercase Character

c) else if ladder: The else-if ladder statements contain multiple else-if, when either of the condition is true the statements under that particular "if" will be executed otherwise the statements under the else block will be executed.

Syntax:

```
if (condition1)
{
  statement1;
}
else if (condition2)
{
  statement2;
}
else if (condition3)
{
  statement3;
}
  ------------
  ------------
else if (condition n-1)
{
  statement n-1;
}
else
{
  statement n;
}
```



Program2_5: Program to print the color name by taking the Color code as input.

V -> Violet
I -> Indigo
B -> Blue
G -> Green
Y -> Yellow
O -> Orange
R -> Red

Input: G
Output: Green

Code:

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

  if(ch=='V' || ch=='v')
                printf("Violet\n");
  else if(ch=='I' || ch=='i')
                printf("Indigo\n");
  else if(ch=='B' || ch=='b')
                printf("Blue\n");
  else if(ch=='G' || ch=='g')
                printf("Green\n");
  else if(ch=='Y' || ch=='y')
                printf("Yellow\n");
  else if(ch=='O' || ch=='o')
                printf("Orange\n");
  else if(ch=='R' || ch=='r')
                printf("Red\n");
  else
                printf("Enter a valid color code\n");
  return 0;
}
```

Program2_6: Program to input electricity unit charges and calculate total electricity bill according to the given condition.

for first 50 units Rs - 0.50/unit
for next 100 units Rs - 0.75/unit
for next 100 units Rs - 1.20/unit
for units above 250 Rs - 1.50/unit
An additional surcharge of 20% is added to the bill.

Input:
150
Output:
120

Explanation:

150 Units => 50 X 0.50 + 100 x 0. 75 => 25 + 75 => 100
175 Units => 50 X 0.50 + 100 X 0.75 + 25 X 1.20 => 25 + 75 + 30 => 130
270 Units => 50 X 0.50 + 100 X 0.75 + 100 X 1.20 + 20 X 1.50 => ....

```
#include<stdio.h>
int main()
{
 int units;
 double bill,surcharge,tot_bill;
 scanf("%d",&units);
 if(units<=50)
 {
        bill=units*0.50;
 }
 else if(units>50 && units<=150)
 {
        bill  = 50 * 0.50 + (units-50) * 0.75;
 }
 else if(units>150 && units<=250)
 {
        bill=50*0.50 + 100 * 0.75 + (units-150) * 1.20;
 }
 else
 {
        bill=50*0.50 + 100 * 0.75 +  100*1.20 + (units-250)*1.50;
 }
 tot_bill = bill + 0.2*bill;
 printf("No of Units = %d\n",units);
 printf("Bill = %.2lf\n",bill);
 printf("Net Bill = %.2lf\n",tot_bill);
 return 0;
}
```

Program2_7: program to find the roots of a Quadratic Equations.

Note: Assume the equation is ax^2 + bx + c=0 then find the (descriminent)d =(b*b - 4*a*c)

if d = 0 => Roots are equal and print the roots.
d >0 => Roots are Real Values.
d < 0 => Roots are Imaginary.

roots = (-b +- sqrt(b^2 - 4ac))/2a

Code:

```
#include<stdio.h>
#include<math.h>
int main()
{
        double a,b,c;
        double d,r1,r2;
        scanf("%lf%lf%lf",&a,&b,&c);                    //Reading of input
        d=b*b-4*a*c;                                    //decriminent value
```

```
        if(d==0)
        {
                printf("Roots are Equal\n");
                r1=-b/(2*a);
                r2=-b/(2*a);
                printf("Root1 = %.2lf\n",r1);
                printf("Root2 = %.2lf\n",r2);
        }
        else if(d>0)
        {
                printf("Roots are Real Numbers\n");
                r1=(-b+sqrt(d))/(2*a);
                r2=(-b-sqrt(d))/(2*a);
                printf("Root1 = %.2lf\n",r1);
                printf("Root2 = %.2lf\n",r2);
        }
        else
        {
                printf("Roots are imaginary\n");
        }
  return 0;
}
```

Assignment:

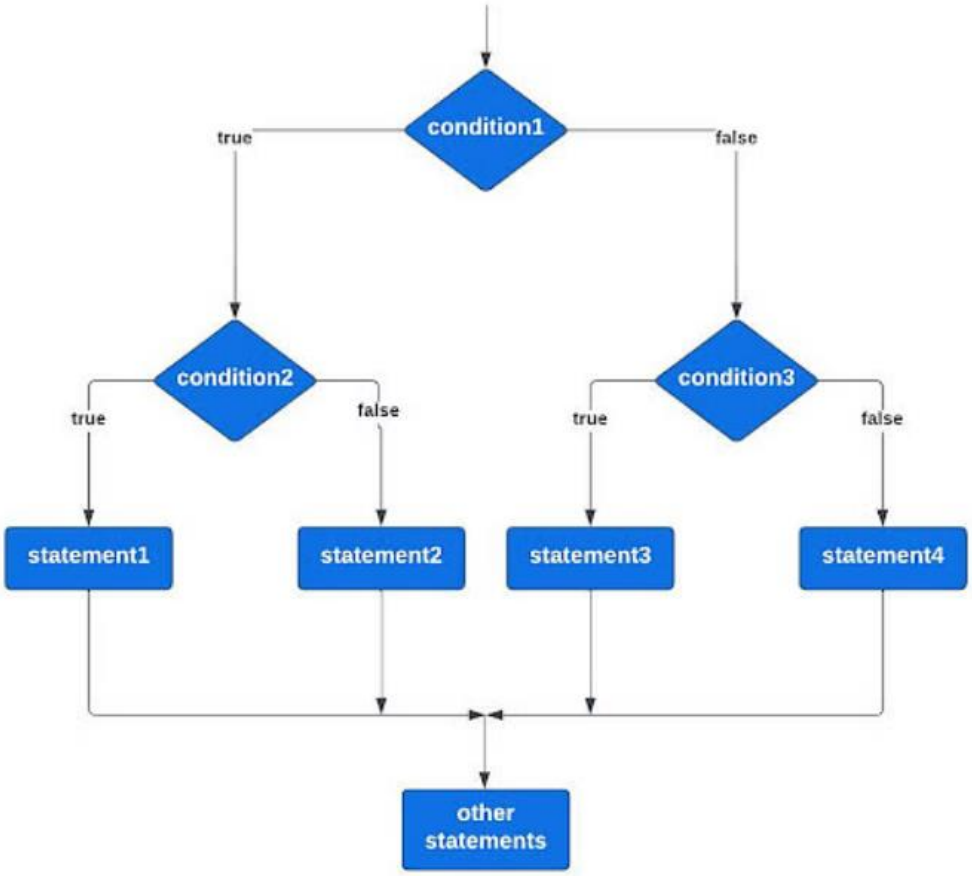
1) Write a program to read temperature in centigrade and display a suitable message according to temperature state below: [Solve]
   Temp < 0 then Freezing weather
   Temp 0-10 then Very Cold weather
   Temp 10-20 then Cold weather
   Temp 20-30 then Normal in Temp
   Temp 30-40 then Its Hot
   Temp >=40 then Its Very Hot

Explanation: Assume the temp=35; => Its Hot

2) Write a program to display the given digit(0 to 9) in words as follows
   Input: 9
   Output: Nine
3) Program to check whether a triangle is equilatoral, Isosceles or Scalence.
   Input: 2 3 4
   Output: Scalence
   Hint:
   Equilatoral -> All the sides of the Triangle are equal
   Isosceles -> Any two sides of the Triangle are equal
   Scalence -> All the sides of the Triangle are different

d) Nested if: A **nested if** statement in C is an **if** statement placed inside another **if** statement. It allows you to check multiple conditions in a sequence. If the outer **if** condition is true, only then will the inner **if** condition be evaluated.

```
if (condition1)
{
//if block of outer if's
  if (condition2)
  {
    statement1;
  }
  else
  {
    Statement2;
  }
}
//else block of outer if's
else
{
  if (condition3)
  {
    statement3;  //inner if
  }
  else
  {
    Statement4;  //else block of inner if
  }
}
```



Program-2.8: Program to find the biggest of three numbers.

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

Program-2.9: Program to check whether a year is a leap year using nested if conditions.

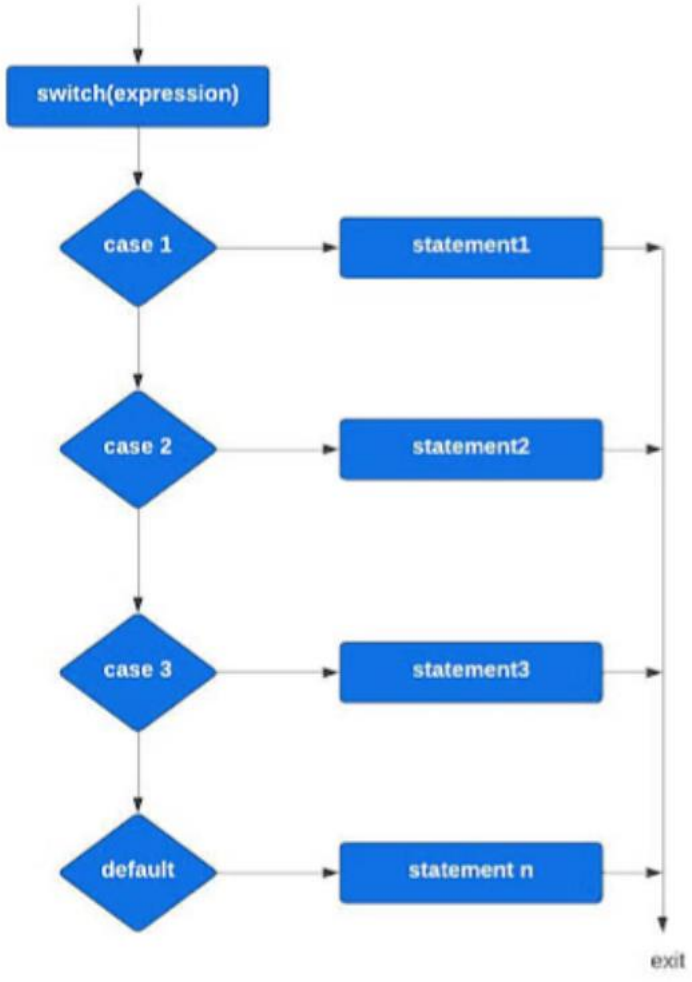
```
#include <stdio.h>
int main()
{
   int year;
   printf("Enter a year: ");
   scanf("%d", &year);
   if (year % 4 == 0)
  {
      if (year % 100 == 0)
      {
         if (year % 400 == 0)
         {
                printf("%d is a leap year.\n", year);
         }
         else
         {
                printf("%d is not a leap year.\n", year);
         }
      }
```

```
    else
    {
            printf("%d is a leap year.\n", year);
    }
  }
  else
  {
            printf("%d is not a leap year.\n", year);
  }
  return 0;
}
```

e) switch: The switch statement in C is used to perform different actions based on different conditions. It's an alternative to using multiple if-else statements when you have many conditions based on the value of a single variable. The switch evaluates the variable and executes the corresponding block of code that matches the value.

Syntax:

```
Switch (expression)
{
  Case label 1:   statement1;
                  ----------
                  break;
  Case label 2:   statement2;
                  ----------
                  break;
  Case label 3:   statement3;
                  ----------
                  break;
-------------------------------------------------
-------------------------------------------------
  Case label n:   statement n;
                  ----------
                  break;
      default:    default statement;
                  ----------
}
Other statements;
```



How It Works:

1) The expression inside the switch is evaluated.
2) The value of the expression is compared with the constants provided in each case.
3) If a match is found, the code block associated with that case is executed.
4) The break statement ends the switch statement. If break is not used, the execution will continue to the next case (known as fall-through behavior).
5) If no matching case is found, the default block (if present) is executed.

Rules Followed in Switch Case:

1) Only Integer and Character Values Allowed:
2) The expression in a switch must evaluate to an integer or character value. Floating-point and strings are not allowed.
3) Case Labels Must Be Constant and Unique:
4) Variables are not allowed in case labels.
5) **Break Statement:** The break statement is used to exit the switch once a matching case is executed. Without break, the code will continue executing the next case even if the value does not match (fall-through).
6) **Default Case:** The default case is optional and is executed when none of the other case values match. It acts like a catch-all for unhandled values.
7) **Fall-Through Behavior:** In C, if there is no break after a case, the program continues executing the next case. This is known as fall-through behavior and can be used intentionally if multiple cases share the same code block.

Program-2.10: Program to print the week day name based on the day number.

Assumption: 1 - sunday , 2 - Monday, ...,7- Saturday

Input: 2

Output: Monday

```
#include<stdio.h>
int main()
{
	int day_num;
	scanf("%d",&day_num);
	switch(day_num)
	{
		case 1:
			printf("Sunday");
			break;
		case 2:
			printf("Monday");
			break;
		case 3:
			printf("Tuesday");
			break;
		case 4:
			printf("Wednsday");
			break;
		case 5:
			printf("Thursday");
			break;
		case 6:
			printf("Friday");
			break;
		case 7:
			printf("Saturday");
```

```
                break;
        default:
                printf("Enter a valid week day number(1-7)");
                break;
    }
    printf("Task Completed");
    return 0;
}
```

Program-2.11: Traffic Light System:

Write a program to simulate a traffic light system using if-else if. Depending on the light color input (Red, Yellow, Green), display the action for the driver (Stop, Slow down, Go).

```
#include<stdio.h>
int main()
{
    char color_Code;
    scanf("%c",&color_Code);
    switch(color_Code)
    {
        case 'R':
        case 'r':
                printf("STOP.....");
                break;
        case 'Y':
        case 'y':
                printf("SLOW DOWN.....");
                break;
        case 'G':
        case 'g':
                printf("GO.....");
                break;
        default:
                printf("Enter Valid Color Code");
                break;
    }
    return 0;
}
```

Program-2.12: Menu-Driven Program for Temperature Conversion:

Write a program that provides a menu to convert temperature between Celsius, Fahrenheit, and Kelvin. The user should choose from a menu (1 for Celsius to Fahrenheit, 2 for Fahrenheit to Celsius, 3 for Celsius to Kelvin, etc.), and the program should perform the appropriate conversion using switch case.

Input:

1. Celsius to Fahrenheit
2. Fahrenheit to Celsius
3. Celsius to Kelvin

Enter you Choice:
2
Enter the Fahrenheit Value
34

Output:

Print Celsius converted Value.

Code:

```
#include<stdio.h>
int main()
{
	double C,F,K;
	int option;
	printf("1. Celsius to Fahrenheit\n2. Fahrenheit to Celsius\n3. Celsius to Kelvin\n");
	printf("Enter your choice(1 - 3)\n");
	scanf("%d",&option);
	switch(option)
	{
		case 1:
			printf("Enter the Celsius Value\n");
			scanf("%lf",&C);
			// F = (C * 9/5)+32
			F=(double)(C*9)/5 + 32;
			printf("C = %.2lf and F = %.2lf\n",C,F);
			break;
		case 2:
			printf("Enter the Fahrenheit value\n");
			scanf("%lf",&F);
			C = (F-32) * 5.0/9;
			printf("F = %.2lf and C = %.2lf\n",F,C);
			break;
		case 3:
			printf("Enter the Celsius value\n");
			scanf("%lf",&C);
			K=C+273.15;
			printf("C = %.2lf and K = %.2lf",C,K);
			break;
		default:
			printf("Enter a valid Option\n");
			break;
	}
   return 0;
}
```

Program-2.13: Program to implement simple calculator by using switch case.

Input:

Enter any two Numbers
20 10
1) + - Addition
2) - - Subtraction
3) * - Multiplication
4) / - Division
5) % - Modulous
Enter your Choice
+

Output:

Sum of 20 and 10 is: 30

Code:

```
#include<stdio.h>
int  main()
{
        int n1,n2,res;
        char op;
        printf("+ - Addition\n- - Subtraction\n* - Multiplication\n/ - Division\n%% - Modulus\n");
        printf("Enter your Choice\n");
        scanf("%c",&op);
        printf("Enter any two numbers\n");
        scanf("%d%d",&n1,&n2);
        switch(op)
        {
                case '+':       res=n1+n2;
                                break;
                case '-':       res=n1-n2;
                                break;
                case '*':       res=n1*n2;
                                break;
                case '/':       if(n2!=0)
                                {
                                        res=n1/n2;
                                }
                                else
                                {
                                        printf("Division is not possible");
                                        res=0;
                                }
                                break;
                case '%':       if(n2!=0)
                                {
                                        res=n1%n2;
```

```
                        }
                        else
                        {
                                printf("Modulus is not possible");
                                res=0;
                        }
                        break;
                default:        printf("Enter a valid Operator\n");
                        break;
        }
        printf("Result = %d\n",res);
        return 0;
}
```

2) Iterative or Looping Statements:

Looping statements in C are used to repeat a block of code multiple times until a specified condition is met. These loops help reduce code redundancy and make programs more efficient.

Types of Looping Statements:

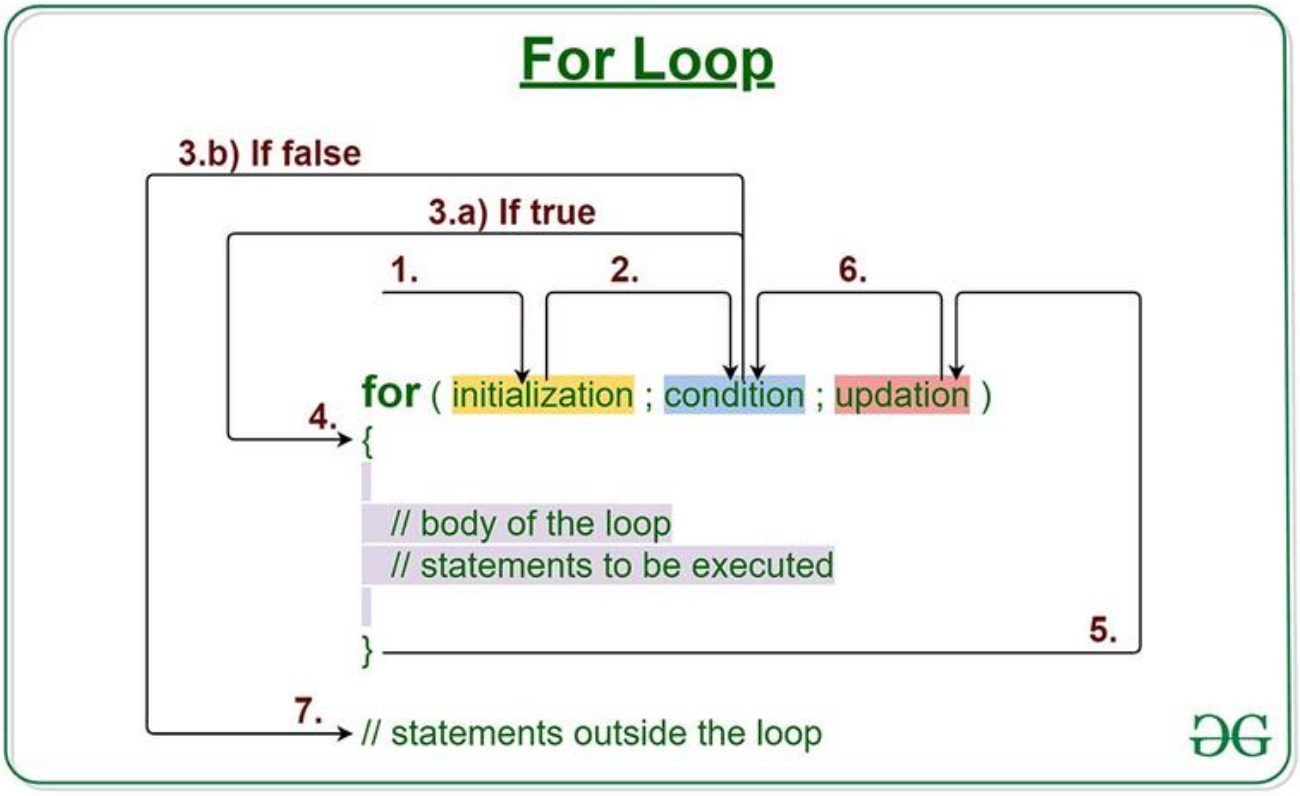
a) for loop:

In C programming, a for loop repeats a block of code a specific number of times. It consists of three parts:

Initialization: Sets the loop control variable (e.g., int i = 0).

Condition: The loop runs as long as this condition is true (e.g., i < 5).

Increment/Decrement: Updates the loop control variable after each iteration (e.g., i++)



The for loop follows a very structured approach where it begins with initializing a condition then checks the condition and in the end, executes conditional statements followed by an updation of values.

Initialization:
This step initializes a loop control variable with an initial value that helps to progress the loop or helps in checking the condition. It acts as the index value when iterating an array or string.
Check/Test Condition:
This step of the for loop defines the condition that determines whether the loop should continue executing or not. The condition is checked before each iteration and if it is true then the iteration of the loop continues otherwise the loop is terminated.
Body:
It is the set of statements i.e. variables, functions, etc that is executed repeatedly till the condition is true. It is enclosed within curly braces { }.
Updation:
This specifies how the loop control variable should be updated after each iteration of the loop. Generally, it is the incrementation (variable++) or decrementation (variable--) of the loop control variable.

Program-2.14: Program to print all the numbers from 1 to 100.

Code:

```
#include<stdio.h>
int main()
{
	int i;
	for(i=1;i<=100;i++)
	{
		printf("%d ",i);
	}
	return 0;
}
```

Program-2.15: Program to find sum of n natural numbers.
Input: 5
Output: 15
Explanation: sum= 1 + 2 + 3 + 4 + 5 => 15
Code:

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

Program-2.16: Program to find the factorial of a given number.

Assume:

n=5 then factorial=5*4*3*2*1

Code:

```
#include<stdio.h>
int main()
{
	int n,prod=1;
	scanf("%d",&n);
	int i;
	for(i=n;i>=1;i--)
	{
		prod=prod*i;
	}
	printf("Factorial of %d is %d",n,prod);
	return 0;
}
```

Program- 2.17: Program to find the value for the following expression.

Assume:

n=5, x=2 then find the expression 1 + x^1 + x^2 + x^3 + x^4 +.....+ x^n

Code:

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

Program-2.18: Program to find the given number is prime or not.

Code:

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

```
        for(i=1;i<=n;i++)
        {
                if(n%i==0)
                   count++;
        }
        if(count==2)
           printf("%d is a prime number",n);
        else
           printf("%d is not a prime number",n);

        return 0;
}
```

Input: 29
Output: 29 is a prime number

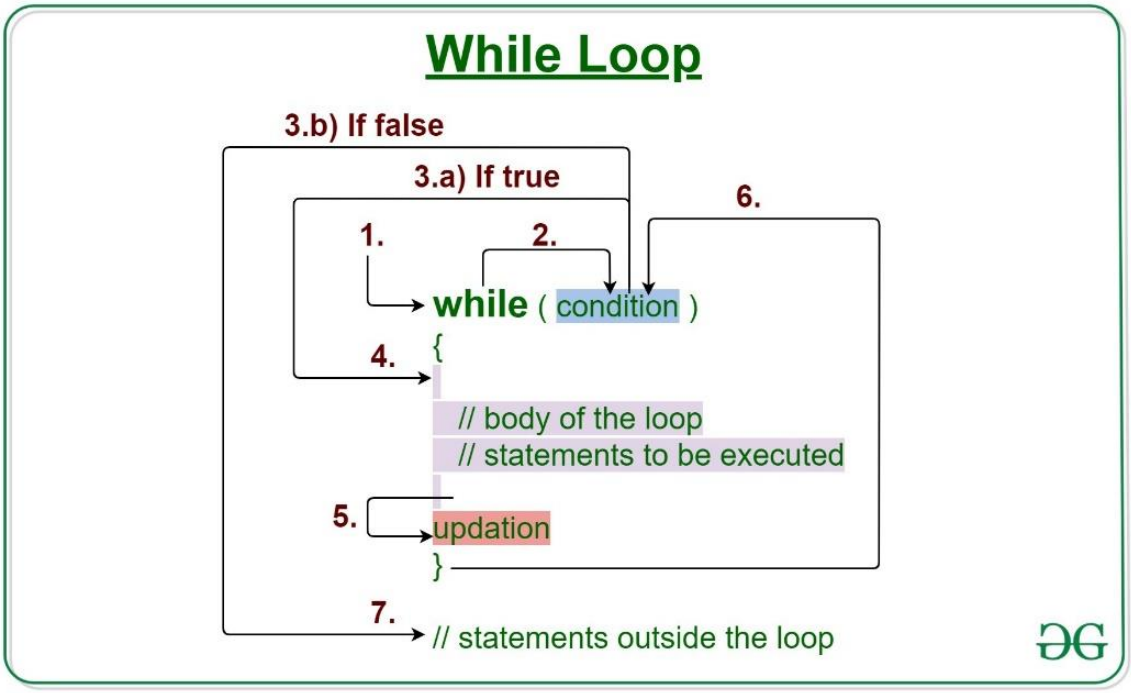
Program-2.19: Program to find the prime numbers within the range of given inputs.
Code:

```
#include<stdio.h>
int main()
{
        int m,n,count=0,i,j;
        scanf("%d%d",&m,&n);
        for(i=m;i<=n;i++)
        {
                count=0;
                for(j=1;j<=i;j++)
                {
                        if(i%j==0)
                   count++;
                }
                if(count==2)
                  printf("%d ",i);

        }
        return 0;
}
```

Input: 1 100
Output: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

b) while Loop:

The while Loop is an entry-controlled loop in C programming language. This loop can be used to iterate a part of code while the given condition remains true.



1) Initialization:

In this step, we initialize the loop variable to some initial value. Initialization is not part of while loop syntax, but it is essential when we are using some variable in the test expression

2) Conditional Statement:

This is one of the most crucial steps as it decides whether the block in the while loop code will execute. The while loop body will be executed if and only the test condition defined in the conditional statement is true.

3) Body:

It is the actual set of statements that will be executed till the specified condition is true. It is generally enclosed inside { } braces.

4) Updation:

It is an expression that updates the value of the loop variable in each iteration. It is also not part of the syntax but we have to define it explicitly in the body of the loop.

Program-2.20: Program to print the numbers from 1 to 10.

```
#include<stdio.h>
int main()
{
        int i=1;                    // Initialization
        while(i<=10)                // Condition
        {
                printf("%d ",i);    // Body of the loop
                i++;                // Updation
        }
        printf("Task Completed");
        return 0;
}
```

Program-2.21: Program to print nth table up to 12 rows.

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

Program-2.22: Program to find the number of digits of a give number.

```
#include<stdio.h>
#include<math.h>
int main()
{
	int n;
	scanf("%d",&n);
	int digits= (int)log10(n)+1;
	printf("No of digits of a given Number %d is: %d",n,digits);
	return 0;
}
```

Input: 5342
Output: 4

Program-2.23: Program to find the sum of digits of a given number
Input: 234
Output: 9
Explanation: sum= 2 + 3 + 4 => 9
Code:

```
#include<stdio.h>
int sumOfDigits(int);   // function prototype
int main()
{
	int n;
	scanf("%d",&n);
	int sum=sumOfDigits(n);
	printf("Sum of Digits of a given number %d is: %d",n,sum);
	return 0;
}
```

```
int sumOfDigits(int n)  // function definition
{
        int sum=0;
        while(n>0)
        {
                int rem=n%10;
                sum=sum+rem;
                n=n/10;
        }
        return sum;
}
```

Program-2.24: Program to find the reverse of a given number.
Input: 123
Output:321
Code:

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

Program-2.25: Program to find the given number is palindrome or not.
Palindrome: Reverse of the given number is equal to given number itself is called Palindrome
Input: 323
Output: YES
Input: 123
Output: NO

Code:

```
#include<stdio.h>
int reverse(int);
int main()
{
	int n;
	scanf("%d",&n);
	int rev=reverse(n);
	printf("Reverse Number = %d\n",rev);
	if(n==rev)
	{
		printf("PALINDROME");
	}
	else
	{
		printf("NOT A PALINDROME");
	}
	return 0;
}
int reverse(int n)
{
	int rem,sum=0;
	while(n>0)
	{
		rem=n%10;
		sum=sum*10+rem;
		n=n/10;
	}
	return sum;
}
```

Program-2.26: Program to check whether a given number is Armstrong number or not.

Armstrong Number: A number is thought of as an Armstrong number if the sum of its own digits raised to the power number of digits gives the number itself.

Input: 153

Output: YES

Explanation: 1^3+5^3+3^3 = 1+125+27 = 153

Input: 1634

Output: YES

Explanation: 1^4 + 6^4 + 3^4 + 4^4 => 1634

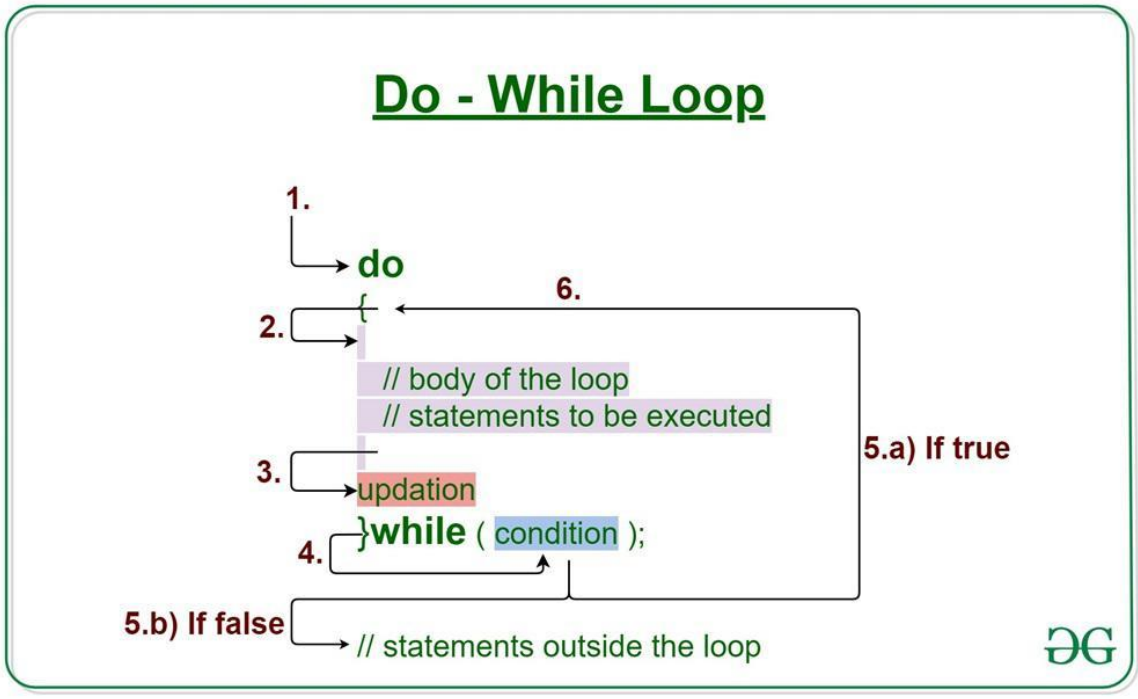
Code:

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

```
        int n,res;
        scanf("%d",&n);
        res=findArmStrongCalculation(n);
        if(n==res)
                printf("Given Number %d is Armstrong Number",n);
        else
                printf("Given Number %d is Not a Armstrong Number",n);
        return 0;
}
int findArmStrongCalculation(int n)
{
        int digits=(int)log10(n)+1;
        int rem,sum=0;
        while(n>0)
        {
                rem=n%10;
                sum=sum+(int)pow(rem,digits);
                n=n/10;
        }
        return sum;
}
```

c) do while loop

In C programming, a do-while loop is a type of loop that ensures the body of the loop is executed at least once, regardless of whether the loop condition is true or false. The condition is checked after the loop body is executed. This is also called **exit control loop**.



Key Points:

Execution at least once:

The loop body will always execute at least once, even if the condition is initially false.

Condition check:
After executing the loop body, the condition is checked. If it's true, the loop continues to execute; if false, the loop terminates.
Semicolon: The while condition in a do-while loop ends with a semicolon.
Program-2.27: Program to print the numbers from 1 to n using do while loop.

```
#include<stdio.h>
int main()
{
        int n;
        scanf("%d",&n);
        int i=1;                        // initialization
        do
        {                               // body of the loop
                printf("%d ",i);
                i++;                    // updation
        }while(i<=n);                   // condition
        printf("Task completed");
}
```

Program-2.28: Program to implement the simple calculator using do while loop?
Expected Output:

```
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
5 - Modulous
Enter your choice
1
Enter any two numbers
10 20
Sum = 30
Do you want to continue...(Y/N)?
Y
1 - Addition
2 - Subtraction
3 - Multiplication
4 - Division
5 - Modulous
Enter your choice
```

Code:

```
#include<stdio.h>
int main()
{
        int option,num1,num2,result;
        char choice;
```

```
        do
        {
                printf("1. Addition\n2. Subtraction\n3. Mulitiplication\n4. Division\n5. Mod\n");
                printf("Enter your Option(1-5)\n");
                scanf("%d",&option);
                printf("Enter any two numbers\n");
                scanf("%d%d",&num1,&num2);
                switch(option)
                {
                        case 1:         result=num1+num2;
                                        printf("Sum = %d\n",result);
                                        break;
                        case 2:         result=num1-num2;
                                        printf("Diff = %d\n",result);
                                        break;
                        case 3:         result=num1*num2;
                                        printf("Product = %d\n",result);
                                        break;
                        case 4:         if(num2!=0)
                                        {
                                          result=num1/num2;
                                          printf("Division = %d\n",result);
                                        }
                                        else
                                          printf("Division is not Possible\n");
                                        break;
                        case 5:         if(num2!=0)
                                        {
                                                result=num1%num2;
                                                printf("Mod = %d\n",result);
                                        }
                                        else
                                          printf("Mod is not possible\n");
                                        break;
                        default:        printf("Enter a valid optioin\n");
                                        break;
                }
                fflush(stdin);
                printf("Do you want to continue...(Y/N)?\n");
                scanf("%c",&choice);
        }while(choice=='Y' || choice=='y');
        printf("Thank you for using my simple Calculator\n");
        return 0;
}
```

Working with break and continue:

Break:

In C programming, the break statement is used to terminate the execution of a loop or a switch statement prematurely.

The break statement can be used inside any type of loop (for, while, or do-while) to immediately exit the loop, regardless of the loop condition. Once a break is encountered, the control is transferred to the first statement following the loop.

Program-2.29: Program to demonstrate the use of break statement.

```
#include <stdio.h>
int main()
{
   for (int i = 1; i <= 10; i++)
   {
         if (i == 5)
         {
                  break;          // Exit the loop when i equals 5
         }
         printf("%d ", i);
   }
   return 0;
}
```

Output: 1 2 3 4

Continue:

In C programming, the **continue** statement is used within loops to skip the current iteration and proceed with the next iteration of the loop. Unlike the **break statement**, which exits the loop entirely, continue allows the loop to continue but **skips the rest of the code in the current iteration**.

Program-2.30: Program to demonstrate the use of continue statement.

```
#include <stdio.h>
int main()
{
   for (int i = 1; i <= 5; i++)
   {
      if (i == 3)
      {
         continue;          // Skip the iteration when i is 3
      }
      printf("%d ", i);
   }
   return 0;
}
```

Output: 1 2 4 5

Working with goto Statement:

The **goto** statement in C provides an unconditional jump to a labeled statement within the same function. It can disrupt the normal flow of execution by jumping directly to another part of the code, usually specified by a label.

Syntax:

```
goto label;
...
label:
    // Code to be executed when the goto jumps here
```

Program-2.31: Program to demonstrate the use of goto statement.

```
#include <stdio.h>
int main()
{
   int i = 0;
   loop:                          // label definition
      printf("i = %d\n", i);
      i++;
      if (i < 5)
     {
         goto loop;               // jump back to the label 'loop'
      }
   printf("Exited loop\n");
   return 0;
}
```

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
Exited loop
```

Nested Loops:

One loop is written inside the another loop is called nested loops. In C, this allows for repeated execution of a block of code within another loop, meaning the inner loop will be executed multiple times for each iteration of the outer loop.

Syntax:

```
for(initailization;condition;incr/decr)          ---> Outer Loop
{
    for(initailization;condition;incr/decr)      ---> Inner Loop
    {
        .......
       //inner block statements
    }
    //Outer block statements
}
```

Program-2.32: Program to demonstrate the use of nested loops

Code:

```
#include<stdio.h>
int main()
{
        int i,j;
        for(i=1;i<=3;i++)
        {
                for(j=1;j<=3;j++)
                {
                        printf("i -> %d and j -> %d\n",i,j);
                }
                printf("Outer Loop Iteration - %d completed\n",i);
        }
   return 0;
}
```

Output:

```
i -> 1 and j -> 1
i -> 1 and j -> 2
i -> 1 and j -> 3
Outer Loop Iteration - 1 completed
i -> 2 and j -> 1
i -> 2 and j -> 2
i -> 2 and j -> 3
Outer Loop Iteration - 2 completed
i -> 3 and j -> 1
i -> 3 and j -> 2
i -> 3 and j -> 3
Outer Loop Iteration - 3 completed
```

Patterns:

Program-2.33: Program to print the following pattern. [Pattern-1]

```
#include<stdio.h>
int main()
{
    int i,j;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5;j++)
        {
            printf("%d ",i);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

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

Program-2.34: Program to print the following pattern. [Pattern-2]

```
#include<stdio.h>
int main()
{
    int i,j;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5;j++)
        {
            printf("%d ",j);
        }
        printf("\n");
    }
    return 0;
}
```

```
Output:
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Program-2.35: Program to print the following pattern. [Pattern-3]

```
#include<stdio.h>
int main()
{
    int i,j;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5;j++)
        {
            printf("%c ",(char)(64+j));
        }
        printf("\n");
    }
    return 0;
}
```

```
Output:
A B C D E
A B C D E
A B C D E
A B C D E
A B C D E
```

Program-2.36: Program to print the following pattern. [Pattern-4]

```
#include<stdio.h>
int main()
{
    int i,j;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=i;j++)
        {
            printf("%d ",j);
        }
        printf("\n");
    }
    return 0;
}
```

```
Output:
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Program-2.37: Program to print the following pattern. [Pattern-5]

```
#include<stdio.h>
int main()
{
    int i,j,n;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        for(j=n;j>=1;j--)
        {
            if(j>i)
              printf("  ");
            else
              printf("* ");
        }
    //control move to the next line
        printf("\n");
    }
    return 0;
}
```

```
Input: 5
Ouput:
        *
      * *
    * * *
  * * * *
* * * * *
```

Program-2.38: Program to print the following pattern. [Pattern-6]

```
#include<stdio.h>
int main()
{
    int i,j,n;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        for(j=2*n-1;j>=1;j--)
        {
            if(j>2*i-1)
              printf("  ");
            else
             printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

```
Input: 5
Output:
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

Program-2.39: Program to print the following pattern. [Pattern-7]

```
#include <stdio.h>
int main()
{
    int rows, i, j, space;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; i++)
    {
        // Print leading spaces
        for (space = 1; space <= rows - i; space++)
        {
            printf(" ");
        }
        // Print asterisks for the pyramid
        for (j = 1; j <= (2 * i - 1); j++)
        {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

```
Input: 5
Output:
    *
   ***
  *****
 *******
*********
```

Important Questions:

1) Define Control Statements. Explain the different types of control statements with neat diagram.
2) Explain the conditional statements with syntax and an example of each.
3) Demonstrate the use of switch case with neat example. Write the rules of switch case.
4) List and explain the different types of looping statements with neat examples.
5) List the differences between entry control loop and exit control looping statements.
6) Explain break, continue and goto statements with syntax and an example of each.
7) Define nested loops. Explain the use of nested loops with an example program.
8) Compare and contrast for, while and do while loop.

Programs to practice:

1) Program to find the factorial of a given number.
2) Program to find the given number is prime or not.
3) Program to print the list of prime numbers between the given range.
4) Program to print the Fibonacci series up to given range.
5) Program to find the reverse of a given number.
6) Program to find the given number is palindrome or not.
7) Program to find the given number is Armstrong number or not.
8) Program to find the implement simple calculator using switch case. etc…

Important MCQ Question and answers:

1) Which of the following can be used to control the flow of execution in a program?
a) if, else, for, while b) switch, continue, break, goto **c) Both a and b** d) None of the above

2) Which of the following is a control statement in C?
a) printf b) scanf **c) for** d) return

3) What is the purpose of the continue statement in a loop?
a) Exit the loop **b) Skip the current iteration** c) Restart the loop d) Exit the program

4) Which statement is used to exit a loop in C?
a) exit b) continue **c) break** d) goto

5) In which loop is the condition checked after executing the loop body?
a) for loop b) while loop **c) do-while loop** d) if statement

6) Which of the following is not a loop control structure in C?
a) for b) do-while **c) switch** d) while

7) What happens if the condition in a while loop is initially false?
a) The loop is skipped entirely b) The loop runs once and stops
c) The loop runs infinitely d) The program crashes

8) How many times is the body of a do-while loop guaranteed to execute?
a) 0 times **b) At least once** c) Twice d) Depends on the condition

9) What is the output of the following code?

```
int i = 0;
while (i < 3) {
   printf("%d", i);
   i++;
}
```

a) 012 b) 123 c) 001 d) 111

10) Which of the following is true about the switch statement?
a) It evaluates multiple conditions b) It supports only integers and characters
c) It executes multiple cases if no break is present d) It can handle floating-point numbers

11) What is the output of the following code?

```
int i = 0;
do {
        printf("%d ", i);
        i++;
} while (i < 3);
```

a) 0 1 **b) 0 1 2** c) 0 1 2 3 d) None of the above

12) What is the default value of the condition in a while loop if the condition is omitted?
a) false b) true c) Undefined **d) Syntax error**

13) What happens if there is no break in a switch case?
a) The program will throw an error **b) The next case will be executed**
c) Only the current case is executed d) The program crashes

14) Which of the following is true for an infinite loop?
a) It has no exit condition b) It contains a break statement
c) It always runs exactly once d) It ends when continue is used

15) How many times will the following loop execute?

```
int i = 10;
```

```
while (i < 5)
{
   printf("Hello");
}
```

a) 0 times b) Infinite times c) 1 time d) 5 times

16) What is the purpose of the else part in an if-else statement?

a) To execute when the if condition is true **b) To execute when the if condition is false**

c) To exit the program d) To repeat the loop

17) Which loop structure allows you to initialize, test, and update in a single line?

a) while b) do-while **c) for** d) goto

18) What is the primary purpose of a break statement inside a loop?

a) Restart the loop b) Skip the current iteration

c) Terminate the loop d) Return to the beginning of the program

19) What will be the output of the following code?

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

a) 01 b) 012 **c) 02** d) 11

20) Which loop structure is best suited when the number of iterations is known?

a) while b) do-while **c) for** d) goto

21) What happens if the condition in a for loop is omitted?

a) The loop will not execute **b) The loop will execute infinitely**

c) The loop will cause a syntax error d) The loop will execute only once

22) Which of the following is true for a switch statement?

a) It allows fall-through by default. b) It does not require break statements.

c) It evaluates floating-point numbers. d) Only the matched case is executed, even without break.

23) Which statement can be used to jump to another section of code?

a) continue b) break **c) goto** d) switch

24) Which statement will immediately exit the innermost loop?

a) exit **b) break** c) continue d) return

25) What will be the output of the following code?

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

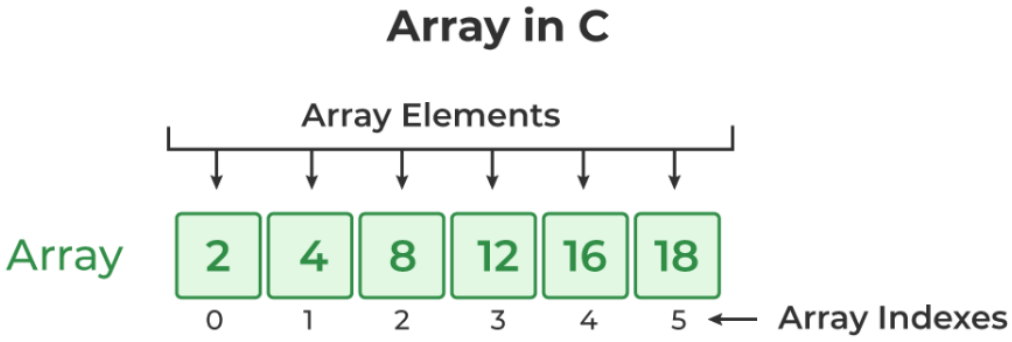
a) 1 2 b) 1 2 3 c) 3 4 d) 1

UNIT – III

Arrays: Arrays indexing, Accessing programs with array of integers, two dimensional arrays,
String: Introduction to Strings, string handling functions.
Sorting techniques: bubble sort, selection sort.
Searching Techniques: linear , Binary search.

Array:

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.

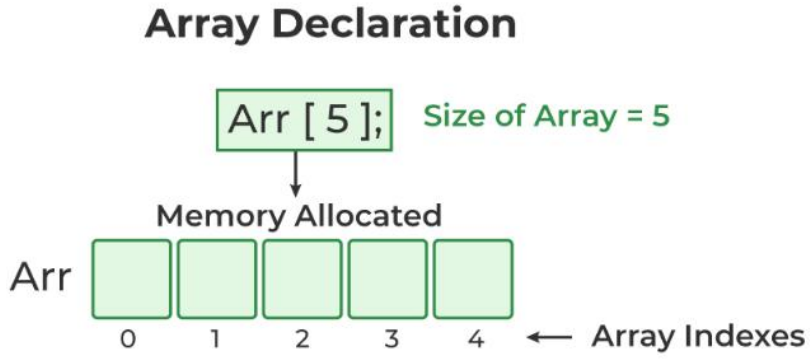


Array Declaration:

In C, we must declare the array like any other variable before using it. We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.

Syntax:

```
data_type array_name [size];
                or
data_type array_name [size1] [size2]...[sizeN];
where N is the number of dimensions.
```



Array Initialization:

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful value. There are two types of initializations of an array.

1) Compile time initialization
2) Runtime initialization

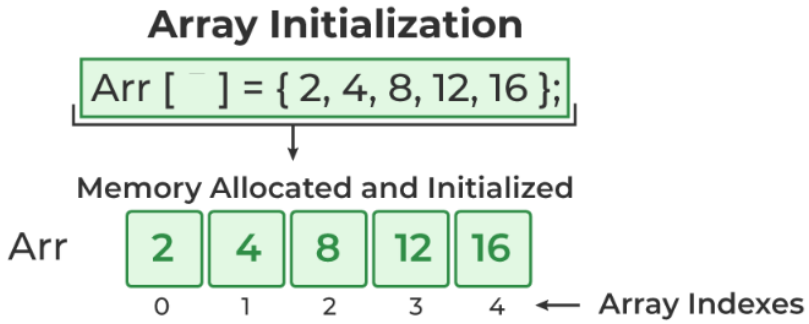
1. Compile time initialization:

In this method, we initialize the array along with its declaration. We use an initializer list to initialize multiple elements of the array. An initializer list is the list of values enclosed within braces **{ }** separated by a comma.

Syntax:

```
data_type array_name [] = {value1, value2, ... valueN};
```

Note: The size of the array in these cases is equal to the number of elements present in the initializer list as the compiler can automatically deduce the size of the array.



2. Runtime initialization:

We initialize the array after the declaration by assigning the initial value to each element individually. We can use for loop, while loop, or do-while loop to assign the value to each element of the array.

```
for (int i = 0; i < N; i++) {
        array_name[i] = value_i;
}
```

Example:

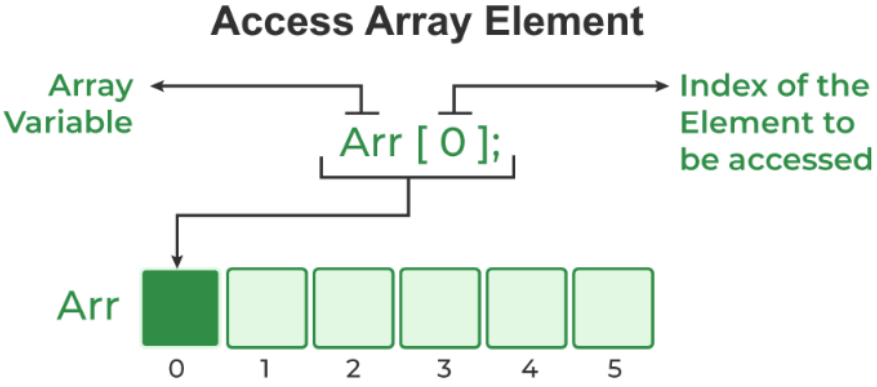
```
int arr[5];
for(int i=0;i<5;i++)
{
        scanf("%d",&arr[i]);
}
```

Accessing of Array Elements:

We can access any element of an array in C using the array subscript operator **[ ]** and the index value ***i*** of the element.

Syntax: array_name [index];

One thing to note is that the indexing in the array always starts with 0, i.e., the first element is **at index 0** and the last element is **at N – 1** where **N** is the number of elements in the array.

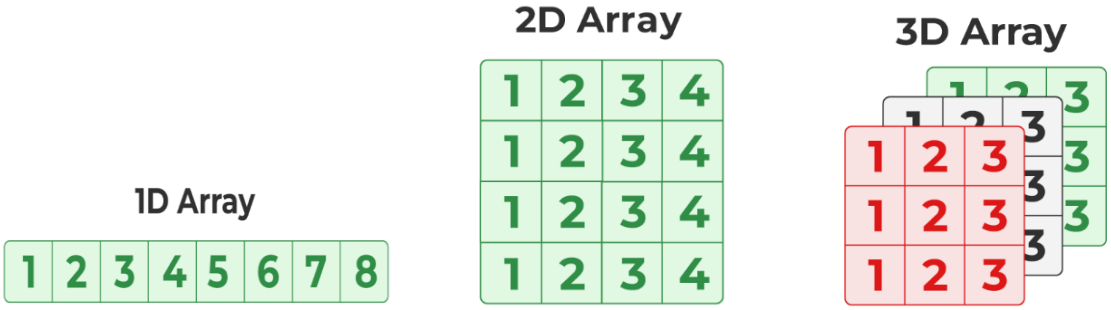


```
// C Program to demonstrate the use of array
#include <stdio.h>
int main()
{
        // array declaration and initialization
        int arr[] = { 10, 20, 30, 40, 50 };

        // modifying element at index 2
        arr[2] = 100;

        // traversing array using for loop
        printf("Elements in Array: ");
        for (int i = 0; i < 5; i++)
        {
                printf("%d ", arr[i]);
        }
  return 0;
}
```

Types of Arrays in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays



One Dimensional Arrays:

In C, a 1D array is a collection of elements of the same data type, stored in contiguous memory locations. Each element can be accessed using an index, starting from 0 up to n-1, where n is the array's length. Arrays are useful for storing lists of data, like numbers or characters, allowing efficient access and manipulation using loops and indices.

Practice Programs:

1) Program to read and display the elements of an array.

Source Code:

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

  // Ask the user for the number of elements
  printf("Enter the number of elements: ");
  scanf("%d", &n);

  // Declare an array of size n
  int arr[n];
```

```
    // Read elements into the array
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++)
   {
       scanf("%d", &arr[i]);
    }

    // Display the elements of the array
    printf("Array elements are:\n");
    for(i = 0; i < n; i++)
    {
       printf("%d ", arr[i]);
    }
    return 0;
}
```

2) Program to find the sum of all the elements of an array.

Source Code:

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;

    // Ask the user for the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];

    // Read elements into the array
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++)
    {
       scanf("%d", &arr[i]);
    }

    // Calculate the sum of all elements
    for(i = 0; i < n; i++)
    {
       sum += arr[i];
    }

    // Display the sum
    printf("The sum of all elements is: %d\n", sum);
    return 0;
}
```

3) Program to find the given element is available in the array or not.

Source Code:

```
#include <stdio.h>
int main()
{
   int n, i, element, found = 0;
   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   // Declare an array of size n
   int arr[n];

   // Read elements into the array
   printf("Enter %d elements:\n", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Ask the user for the element to search
   printf("Enter the element to search: ");
   scanf("%d", &element);

   // Check if the element is in the array
   for(i = 0; i < n; i++)
   {
      if(arr[i] == element)
     {
         found = 1;
         break;
      }
   }

   // Display result
   if(found == 1)
   {
          printf("Element %d is present in the array.\n", element);
   }
    else
   {
          printf("Element %d is not present in the array.\n", element);
   }
   return 0;
}
```

4) Program to find the biggest and smallest element in an array.

Source Code:

```
#include <stdio.h>
int main() {
   int n, i;
   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);
```

```
    // Declare an array of size n
    int arr[n];

    // Read elements into the array
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++)
    {
       scanf("%d", &arr[i]);
    }

    // Initialize max and min with the first element
    int max = arr[0];
    int min = arr[0];

    // Find the maximum and minimum elements
    for(i = 1; i < n; i++)
    {
          if(arr[i] > max) {
                max = arr[i];
          }

          if(arr[i] < min) {
                min = arr[i];
          }
    }

    // Display the results
    printf("The largest element is: %d\n", max);
    printf("The smallest element is: %d\n", min);

    return 0;
}
```

5) Program to find the frequency of each element in the given array.

Source Code:

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

    // Ask the user for the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Declare an array of size n
    int arr[n];
    int freq[n];

    // Read elements into the array
    printf("Enter %d elements:\n", n);
    for(i = 0; i < n; i++) {
       scanf("%d", &arr[i]);
```

```
    freq[i] = -1;                    // Initialize frequency array with -1
  }

  // Calculate frequency of each element
  for(i = 0; i < n; i++) {
    int count = 1;
    for(j = i + 1; j < n; j++) {
      if(arr[i] == arr[j]) {
        count++;
        freq[j] = 0;                 // Mark duplicate elements as 0
      }
    }
    if(freq[i] != 0) {
      freq[i] = count;
    }
  }

  // Display the frequency of each unique element
  printf("Element | Frequency\n");
  for(i = 0; i < n; i++)
  {
    if(freq[i] != 0) {
      printf("   %d    |    %d\n", arr[i], freq[i]);
    }
  }

  return 0;
}
```

6) Program to find the unique element in the given array.
7) Program to find the duplicate elements in the given array.

2 Dimensional Arrays:

In C, a 2-dimensional (2D) array is essentially an array of arrays, used to store data in a tabular form (rows and columns). It is declared by specifying two indices: the number of rows and columns.

For example, int arr[3][4]; declares a 2D array with 3 rows and 4 columns. Each element can be accessed using two indices, **arr[i][j],** where i is the row index and j is the column index. 2D arrays are widely used for matrices, grids, and other data structures where data is organized in rows and columns.

Practice Programs:

1. Program to read and display a matrix.

Source Code:

```
#include <stdio.h>
int main() {
  int rows, columns, i, j;
  // Ask the user for the number of rows and columns
  printf("Enter the number of rows: ");
  scanf("%d", &rows);
  printf("Enter the number of columns: ");
  scanf("%d", &columns);
```

```
    // Declare a 2D array with the specified dimensions
    int matrix[rows][columns];

    // Read elements into the matrix
    printf("Enter the elements of the matrix:\n");
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    // Display the matrix
    printf("The matrix is:\n");
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
       {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements of the matrix:
1 2 3 4 5 6
```

The matrix is:

```
1   2   3
4   5   6
```

2. Program to find the sum of all the elements of a matrix.

Source Code:

```
#include <stdio.h>
int main()
{
    int rows, columns, i, j, sum = 0;

    // Ask the user for the number of rows and columns
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &columns);

    // Declare a 2D array with the specified dimensions
    int matrix[rows][columns];

    // Read elements into the matrix
    printf("Enter the elements of the matrix:\n");
```

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

    // Calculate the sum of all elements in the matrix
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
        {
            sum += matrix[i][j];
        }
    }

    // Display the sum
    printf("The sum of all elements in the matrix is: %d\n", sum);

    return 0;
}
```

3. Program to find maximum element in a matrix.

4. Program to find the addition of two matrices.

Source Code:

```
#include <stdio.h>
int main()
{
    int rows, columns, i, j;

    // Ask the user for the number of rows and columns
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &columns);

    // Declare two 2D arrays for the matrices and one for the result
    int matrix1[rows][columns], matrix2[rows][columns], sum[rows][columns];

    // Read elements into the first matrix
    printf("Enter elements of the first matrix:\n");
    for(i = 0; i < rows; i++) {
        for(j = 0; j < columns; j++) {
            scanf("%d", &matrix1[i][j]);
        }
    }

    // Read elements into the second matrix
    printf("Enter elements of the second matrix:\n");
    for(i = 0; i < rows; i++) {
```

```
        for(j = 0; j < columns; j++) {
            scanf("%d", &matrix2[i][j]);
        }
    }

    // Calculate the sum of the two matrices
    for(i = 0; i < rows; i++) {
        for(j = 0; j < columns; j++) {
            sum[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    // Display the resulting matrix
    printf("The sum of the two matrices is:\n");
    for(i = 0; i < rows; i++) {
        for(j = 0; j < columns; j++) {
            printf("%d ", sum[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter elements of the first matrix:
1   2   3
4   5   6
Enter elements of the second matrix:
7   8   9
10 11  12
```

The sum of the two matrices is:

```
8       10      12
14      16      18
```

5. Program to find the product of two matrices.

Source Code:

```
#include <stdio.h>
int main()
{
    int rows1, cols1, rows2, cols2, i, j, k;

    // Ask the user for the dimensions of the first matrix
    printf("Enter the number of rows and columns for the first matrix: ");
    scanf("%d %d", &rows1, &cols1);

    // Ask the user for the dimensions of the second matrix
    printf("Enter the number of rows and columns for the second matrix: ");
    scanf("%d %d", &rows2, &cols2);
```

```
    // Check if the matrices can be multiplied
    if (cols1 != rows2)
    {
       printf("Matrix multiplication is not possible.\n");
       return 0;
    }

    // Declare the matrices
    int matrix1[rows1][cols1], matrix2[rows2][cols2], product[rows1][cols2];

    // Read elements into the first matrix
    printf("Enter elements of the first matrix:\n");
    for (i = 0; i < rows1; i++) {
       for (j = 0; j < cols1; j++) {
          scanf("%d", &matrix1[i][j]);
       }
    }

    // Read elements into the second matrix
    printf("Enter elements of the second matrix:\n");
    for (i = 0; i < rows2; i++) {
       for (j = 0; j < cols2; j++) {
          scanf("%d", &matrix2[i][j]);
       }
    }

    // Calculate the product of the two matrices
    for (i = 0; i < rows1; i++)
   {
       for (j = 0; j < cols2; j++)
      {
           product[i][j]=0;
          for (k = 0; k < cols1; k++)
           {
                       product[i][j] += matrix1[i][k] * matrix2[k][j];
           }
       }
    }

    // Display the resulting product matrix
    printf("The product of the two matrices is:\n");
    for (i = 0; i < rows1; i++) {
       for (j = 0; j < cols2; j++) {
          printf("%d ", product[i][j]);
       }
       printf("\n");
    }

    return 0;
 }
```

6. Program to find the transpose of a matrix.

Source Code:

```
#include <stdio.h>
int main()
{
   int rows, columns, i, j;

   // Ask the user for the number of rows and columns
   printf("Enter the number of rows: ");
   scanf("%d", &rows);
   printf("Enter the number of columns: ");
   scanf("%d", &columns);

   // Declare a 2D array for the matrix and another for the transpose
   int matrix[rows][columns], transpose[columns][rows];

   // Read elements into the matrix
   printf("Enter elements of the matrix:\n");
   for(i = 0; i < rows; i++) {
      for(j = 0; j < columns; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }

   // Calculate the transpose of the matrix
   for(i = 0; i < rows; i++) {
      for(j = 0; j < columns; j++) {
         transpose[j][i] = matrix[i][j];
      }
   }

   // Display the transposed matrix
   printf("The transpose of the matrix is:\n");
   for(i = 0; i < columns; i++) {
      for(j = 0; j < rows; j++) {
         printf("%d ", transpose[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```

String Handling

String:

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as words or sentences.

Each character in the array occupies one byte of memory, and the last character must always be '\0'.

The termination character ('\0') is important in a string since it is the only way to identify where the string ends.

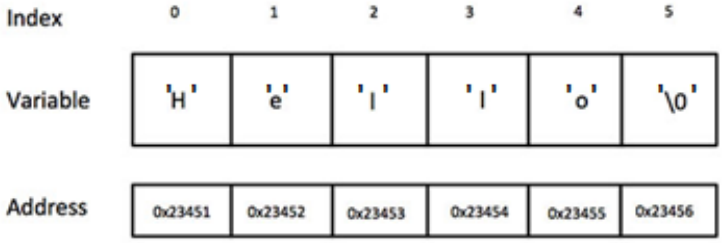
Declaration and Initialization:

As string is a character array, it is declared and initialized as 1-D array.

Syntax: char name[20];

Compile time initialization:

char msg[20]={'h','e','l','l','o','\0'}; or char msg[]={'h','e','l','l','o','\0'};



Ex:

```
#include <stdio.h>
int main()
{
        char greeting[5] = {'H', 'e', 'l', 'l', 'o', '\0'};
        printf("Greeting message: %s\n", greeting );
        return 0;
}
```

We can also define the string by the **string literal** in C language. For example:

```
char ch[]="hello";
```

Runtime initialization:

1. **To read a word**

scanf("%s",str);

Ex:

```
#include <stdio.h>
int main()
{
        char str[20];
        scanf("%s",str);
        printf("message: %s\n", str );
        return 0;
}
```

Input: Aditya Engineering College **Output:** message: Aditya

2. **To read a line of text**

scanf("%[^\n]s",str); or **gets(str);**

Ex:

```
#include <stdio.h>
int main()
{
        char str[20];
        scanf("%[^\n]s",str);
        printf("message: %s\n", str );
        return 0;
}
```

Input: Aditya Engineering College **Output:** message: Aditya Engineering College

Ex:

```
#include <stdio.h>
int main()
{
        char str[20];
        gets(str);              // it also reads a line of text
        printf("message: %s\n", str );
        return 0;
}
```

Input: Aditya Engineering College **Output:** message: Aditya Engineering College

3. To read multi line of text

```
scanf("%[a-zA-Z0-9 @\n]s",str);
```

Ex:

```
#include <stdio.h>
int main()
{
        char str[20];
        scanf("%[a-zA-Z0-9 @\n]s",str);
        printf("%s\n", str );
        return 0;
}
```

Input: Aditya
Engineering
College
#[reading of input ends here].

Output: Aditya
Engineering
College

gets(),puts(): These are the functions available in stdio.h header file to read and display strings.

gets():

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array.

Syntax: **char**[] gets(**char**[]);

puts():

The puts() function is very similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console.

Syntax: **int** puts(**char**[]);

```
#include<stdio.h>
int main(){
        char name[50];
        printf("Enter your name: ");
        gets(name);             //reads string from user
        printf("Your name is: ");
        puts(name);             //displays string
        return 0;
 }
```

Input: Madava Rao **Ouput:** Madava Rao

String functions: string.h header file contains many predefined functions to manipulate strings. The following are the some important functions in string.h header file.

No.	Function	Description
1	strlen(string_name)	returns the length of string name.
2	strcpy(destination, source)	copies the contents of source string to destination string.
3	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5	strrev(string)	returns reverse string.
6	strlwr(string)	returns string characters in lowercase.
7	strupr(string)	returns string characters in uppercase.
8	strchr(s1, ch)	Returns a pointer to the first occurrence of character ch in string s1.
9	strstr(s1, s2)	Returns a pointer to the first occurrence of string s2 in string s1.
10	strrchr(s1,ch)	Returns a pointer to the last occurrence of character ch in string s1.

1. strlen():

strlen() is the function used to find the length of the given string.

Syntax: int strlen(char str[])

With strlen() function

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[20],len;
    scanf("%s",s1);
    len=strlen(s1);
    printf("Length = %d\n",len);
    return 0;
}
```

Input: Computer Output: Length = 8

Without strlen() function

```
#include<stdio.h>
int main()
{
    char s1[20],len=0,i;
    scanf("%s",s1);
    for(i=0;s1[i]!='\0';i++)
        len++;
    printf("Length = %d\n",len);
    return 0;
}
```

Input: Computer Output: Length = 8

2. strcpy():

Syntax: char * strcpy(char dest[], char src[])

With strcpy() function

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[20],s2[20];
    scanf("%s%s",s1,s2);
    strcpy(s1,s2);
    printf("s1 = %s\n",s1);
    printf("s2 = %s\n",s2);
    return 0;
}
```

Without strcpy() function

```
#include<stdio.h>
int main()
{
    char s1[20],s2[20];
    int i;
    scanf("%s%s",s1,s2);
    for(i=0;s2[i]!='\0';i++){
       s1[i]=s2[i];}
    s1[i]='\0';
    printf("s1 = %s\n",s1);
    printf("s2 = %s\n",s2);
    return 0;
}
```

Input: Hello Welcome **Output:** s1 = Welcome s2 = Welcome

3. **strcat():** It is used to concatenate two strings.

Syntax: char* strcat(char dest[],char src[])

With strcat() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20],s2[20];
     scanf("%s%s",s1,s2);
     strcat(s1,s2);
     printf("s1 = %s\n",s1);
     printf("s2 = %s\n",s2);
     return 0;
}
```

Without strcat() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20],s2[20];
     int i,len;
     scanf("%s%s",s1,s2);
     len=strlen(s1);
     for(i=0;s2[i]!='\0';i++){
        s1[len]=s2[i];
        len++;
       }
     s1[len]='\0';
     printf("s1 = %s\n",s1);
     printf("s2 = %s\n",s2);
     return 0;
}
```

Input: Aditya Engineering **Output:** s1 = AdityaEngineering s2 = Engineering

4. **strcmp():** is used to compare two strings. It returns 0 if both the strings are equal. If s1 is greater than s2 then it returns 1 otherwise it returns -1.

With strcmp() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20],s2[20];
     int n;
     scanf("%s%s",s1,s2);
     n=strcmp(s1,s2);
     if(n==0)
       printf("Both are equal");
     else if(n>0)
       printf("s1 > s2");
     else
       printf("s1 < s2");
     return 0;
}
```

Without strcmp() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20],s2[20];
     int i=0,l1,l2,flag=0;
     scanf("%s%s",s1,s2);
     l1=strlen(s1);
     l2=strlen(s2);
     if(l1!=l2)
        printf("Strings are not Same\n");
     else{
       while(s1[i]!='\0' && s2[i]!='\0')
       {   if(s1[i]!=s2[i]) {
              flag=1;
              break;  }
          i++;
       }
       if(flag==0)
          printf("Strings are Same");
       else
          printf("Strings are not Same");
       }
      return 0;
}
```

Input: abc abcd **Output:** Strings are not Same

Input: ramesh ramesh **Output:** Strings are Same

5. **strrev():** it is used to get the reverse of the given string.

Syntax: char* strrev(char str[])

Note: A string is said to be palindrome if reverse of the given string is equal to original string.
Example: liril, madam are some examples of palindromes.

With strrev() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20];
     scanf("%s",s1);
     printf("S1 = %s\n",s1);
     strrev(s1);
     printf("S1 = %s\n",s1);
     return 0;
}
```

Without strrev() function

```
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20],temp;
     int i,n;
     scanf("%s",s1);
     n=strlen(s1);
     for(i=0;i<n/2;i++)
       {
         temp=s1[i];
         s1[i]=s1[n-i-1];
         s1[n-i-1]=temp;
       }
     printf("Reverse String = %s\n",s1);
     return 0;
}
```

Input: Welcome **Output:** Reverse String = emocleW

Note: Refer remaining functions like strlwr(), strupr(), strchr(), strstr() etc..

Practice Programs:

```
// program to find the vowels and
// consonats in the given string
#include<stdio.h>
#include<string.h>
int main()
{
     char s1[20];
     int v=0,c=0,i;
     gets(s1);
     for(i=0;s1[i]!='\0';i++)
     {
      if(s1[i]=='A'||s1[i]=='E'||
      s1[i]=='I'||s1[i]=='E'||s1[i]=='U')
         v++;
      else
         c++;
     }
     printf("Vowels: %d Consonants: %d",v,c);
     return 0;
}
```

```
// program to find no of words
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
     char s1[20];
     int word=0,i;
     gets(s1);
     for(i=0;s1[i]!='\0';i++)
       if(s1[i]==' ')
         word++;
     printf("Word Count %d",word+1);
     getch();
     return 0;
}
```

Input: abc def ghi **Output:** 3

```
// program to check palindrome
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
     char s1[20],s2[20];
     gets(s1);
     strcpy(s2,s1);
     strrev(s1);
     int n=strcmp(s1,s2);
     if(n==0)
       printf("Palindrome");
     else
       printf("Not Palindrome");
     getch();
     return 0;
}
```

```
// program to count digits, alphabets
// symbols
#include<stdio.h>
#include<ctype.h>
int main()
{
     char s1[20];
     int d=0,a=0,s=0,i;
     gets(s1);
     for(i=0;s1[i]!='\0';i++){
       if(isdigit(s1[i]))  d++;
       else if(isalpha(s1[i])) a++;
       else s++;
      }
     printf("%d %d %d",d,a,s);
     return 0;
}
```

Input: Srinu@123 **Output:** 3 5 1

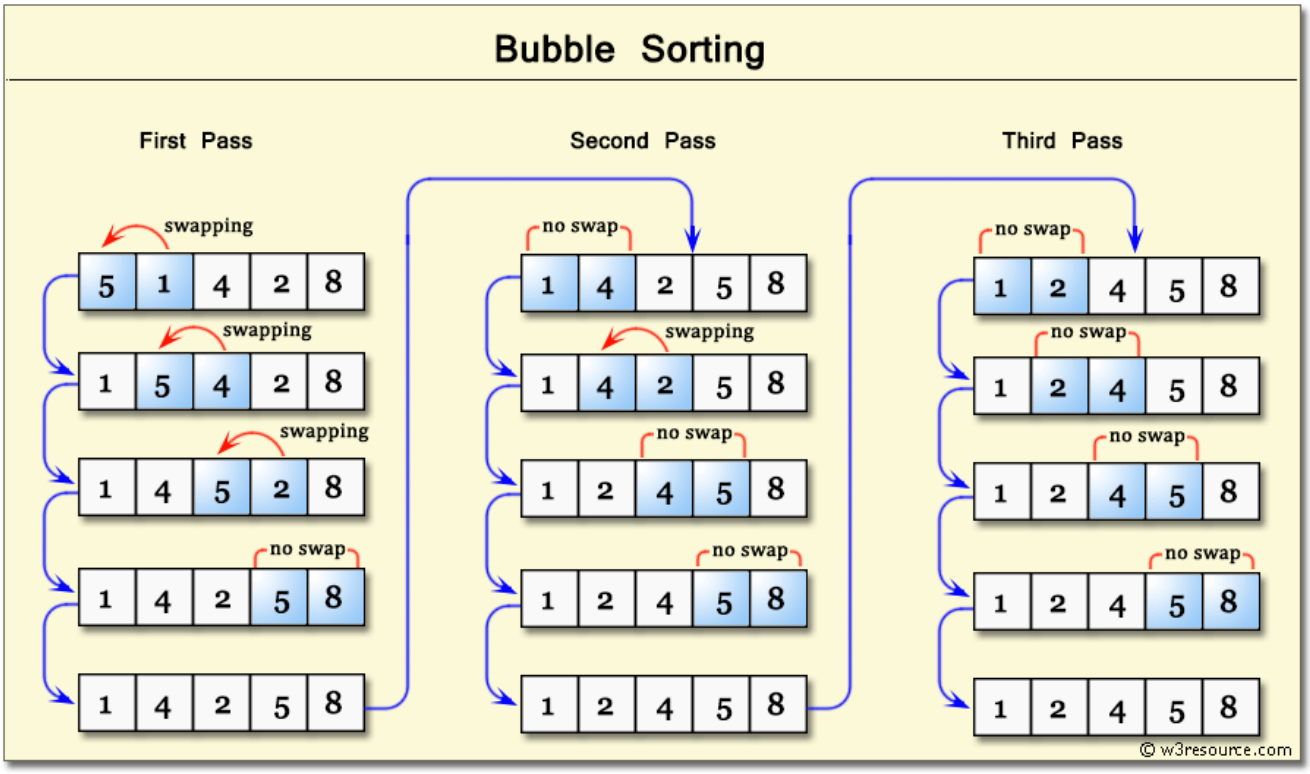
Sorting Techniques

Sorting is the process of arranging the elements of a collection (such as an array or list) in a specific order, typically in ascending or descending order.

1) **Bubble Sort**: Compares adjacent elements and swaps them if they are in the wrong order. This process repeats until the array is sorted.
   - Time Complexity: $O(n^2)$
2) **Selection Sort**: Divides the array into a sorted and an unsorted part, repeatedly selecting the smallest (or largest) element from the unsorted part and moving it to the sorted part.
   - Time Complexity: $O(n^2)$
3) **Insertion Sort**: Builds the sorted array one element at a time by repeatedly taking an element from the unsorted part and inserting it into the correct position in the sorted part.
   - Time Complexity: $O(n^2)$ ($O(n)$ in the best case)
4) **Merge Sort**: A divide-and-conquer algorithm that splits the array into halves, recursively sorts each half, and then merges the sorted halves back together.
   - Time Complexity: $O(n \log n)$
5) **Quick Sort**: Another divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into elements less than and greater than the pivot, then recursively sorts the partitions.
   - Time Complexity: $O(n \log n)$ ($O(n^2)$ in the worst case)

Bubble Sort:

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through a list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until no more swaps are needed, which means the list is sorted.



Example:

For the array [5, 2, 9, 1, 5], Bubble Sort would proceed as follows:

- First pass: Compare and swap: [2, 5, 1, 5, 9]
- Second pass: Compare and swap: [2, 1, 5, 5, 9]
- Third pass: Compare and swap: [1, 2, 5, 5, 9]
- No swaps in the next pass, so the algorithm stops.

Program:

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   // Outer loop for each pass
   for (i = 0; i < n - 1; i++)
    {
      // Inner loop for comparison and swapping
      for (j = 0; j < n - i - 1; j++)
      {
         if (arr[j] > arr[j + 1])
          {
            // Swap arr[j] and arr[j + 1]
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}
void printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main()
{
   int n, i;

   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n];
   // Read elements into the array
   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   // Sort the array using Bubble Sort
   bubbleSort(arr, n);

   // Display the sorted array
   printf("Sorted array:\n");
   printArray(arr, n);
   return 0;
}
```

Selection Sort:

Selection Sort is a straightforward comparison-based sorting algorithm that works by dividing the input array into two parts: a sorted section and an unsorted section. The algorithm repeatedly selects the smallest (or largest, depending on the desired order) element from the unsorted section and moves it to the end of the sorted section.

Example:

For the array [64, 25, 12, 22, 11], Selection Sort would proceed as follows:

- First pass: Find the minimum (11), swap with the first element: [11, 25, 12, 22, 64]
- Second pass: Find the minimum (12), swap with the second element: [11, 12, 25, 22, 64]
- Third pass: Find the minimum (22), swap with the third element: [11, 12, 22, 25, 64]
- Fourth pass: The remaining elements are already in order.

Program:

```
#include <stdio.h>
void selectionSort(int arr[], int n)
{
   int i, j, minIndex, temp;

   // Outer loop for each position in the array
   for (i = 0; i < n - 1; i++) {
      // Assume the minimum is the first element of the unsorted section
      minIndex = i;

      // Inner loop to find the minimum element in the unsorted section
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;          // Update minIndex if a smaller element is found
         }
      }

      // Swap the found minimum element with the first element of the unsorted section
      if (minIndex != i)
      {
         temp = arr[i];
         arr[i] = arr[minIndex];
         arr[minIndex] = temp;
      }
   }
}

void printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

```
int main()
{
   int n, i;

   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n];

   // Read elements into the array
   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Sort the array using Selection Sort
   selectionSort(arr, n);

   // Display the sorted array
   printf("Sorted array:\n");
   printArray(arr, n);

   return 0;
}
```

Searching Techniques

Searching techniques are algorithms used to locate a specific value or element within a data structure (like an array or list).

Common Searching Techniques:

1. **Linear Search**:
   - **Description**: Scans each element in a list or array sequentially until the desired element is found or the end of the structure is reached.
   - **Time Complexity**: O(n)
   - **Use Case**: Simple to implement; best for small or unsorted data.
2. **Binary Search**:
   - **Description**: Works on sorted arrays. It repeatedly divides the search interval in half, comparing the target value to the middle element. If the target is less than the middle element, it searches the left half; otherwise, it searches the right half.
   - **Time Complexity**: O(log n)
   - **Use Case**: Efficient for large, sorted datasets.

Program to implement Linear Search:

```
#include <stdio.h>
int linearSearch(int arr[], int n, int target)
{
   for (int i = 0; i < n; i++) {
      if (arr[i] == target) {
         return i;                    // Return the index if the target is found
      }
   }
   return -1;                         // Return -1 if the target is not found
}
```

```
int main()
{
   int n, target, result;

   // Ask the user for the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n];

   // Read elements into the array
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Ask the user for the target value to search
   printf("Enter the element to search for: ");
   scanf("%d", &target);

   // Perform linear search
   result = linearSearch(arr, n, target);

   // Display the result
   if (result != -1)
   {
      printf("Element %d found at index %d.\n", target, result);
   }
   else
   {
      printf("Element %d not found in the array.\n", target);
   }

   return 0;
}
```

Program to implement Binary Search:

```
#include <stdio.h>
int binarySearch(int arr[], int n, int target)
{
   int left = 0, right = n - 1;

   while (left <= right)
   {
      int mid = left + (right - left) / 2;          // To prevent overflow

      // Check if the target is present at mid
      if (arr[mid] == target)
      {
         return mid;                                // Return the index if the target is found
      }
```

```
        // If the target is greater, ignore the left half
        if (arr[mid] < target)
        {
            left = mid + 1;
        }
        else
        {   // If the target is smaller, ignore the right half
            right = mid - 1;
        }
    }

    return -1;                                  // Return -1 if the target is not found
}

int main()
{
    int n, target, result;

    // Ask the user for the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Read elements into the array
    printf("Enter %d sorted elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Ask the user for the target value to search
    printf("Enter the element to search for: ");
    scanf("%d", &target);

    // Perform binary search
    result = binarySearch(arr, n, target);

    // Display the result
    if (result != -1)
    {
        printf("Element %d found at index %d.\n", target, result);
    }
    else
    {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

MCQ Questions with Answers

What is the size of an array declared as int arr[10];?
A) 10 B) 20 C) Depends on the system D) 40

Which of the following is the correct way to access the first element of an array arr?
A) arr[1] B) arr[0] C) &arr[0] **D) Both B and C**

What will be the output of the following code?
int arr[5] = {1, 2, 3, 4, 5};
printf("%d", arr[2]);
A) 1 B) 2 **C) 3** D) 4

Which of the following declares an array of 10 integers?
A) int arr[10]; B) int arr(10); C) int arr{10}; D) int arr = new int[10];

What is the output of the following code?
int arr[] = {1, 2, 3, 4}; printf("%d", arr[4]);
A) 1 B) 4 **C) Compilation error** D) Undefined behavior

Which of the following statements correctly initializes an array of 5 floats?
A) float arr[5]; B) float arr = {1.0, 2.0, 3.0, 4.0, 5.0};
C) float arr[5] = {1.0, 2.0, 3.0}; **D) Both A and C**

Which of the following can be used to store a string in C?
A) char str[10]; B) char *str; **C) Both A and B** D) None of the above

Which of the following is the correct way to declare a 2D array of integers?
A) int arr[5][5]; B) int arr(5)(5); C) int arr[5, 5]; D) int arr{5}{5};

Which of the following statements about arrays is true?
A) Arrays can hold different data types. **B) The size of an array must be constant.**
C) Arrays can be resized after declaration. D) None of the above.

What does the sizeof operator return when applied to an array?
A) The size of the first element **B) The total size in bytes of the array**
C) The number of elements in the array D) A pointer to the array

In C, what happens when you assign an array to another array?
A) The entire array is copied. B) Only the first element is copied.
C) You cannot assign arrays directly. D) A reference to the array is created.

What is the output of the following code?
int arr[3] = {1, 2, 3};
printf("%d", arr[arr[1]]);
A) 2 **B) 3** C) 1 D) Undefined behavior

Which of the following will NOT compile?
A) int arr[5] = {1, 2, 3}; **B) int arr[5] = {1, 2, 3, 4, 5, 6};**
C) int arr[5] = {}; D) int arr[] = {1, 2, 3};

What function is used to calculate the length of a string in C?
A) strlength() **B) strlen()** C) string_length() D) length()

Which of the following is the correct way to declare a string in C?
A) char str[10]; B) char str[] = "Hello"; **C) Both A and B** D) None of the above

What will be the output of the following code?
```
char str[] = "Hello, World!";
printf("%c", str[7]);
```
A) H B) e **C) W** D) !

Which function is used to copy one string to another in C?
A) strcpy() B) copystr() C) string_copy() D) strcopy()

What does the function strcat() do?
A) Compares two strings B) Copies one string to another
C) Concatenates two strings D) Finds the length of a string

Which of the following functions is used to compare two strings in C?
A) string_compare() **B) strcmp()** C) str_compare() D) compare()

What is the purpose of the function strchr()?
A) To find a character in a string B) To find the length of a string
C) To convert a string to lowercase D) To concatenate two strings

Which function can be used to convert a string to an integer?
A) atoi() B) itoa() C) str_to_int() D) convert()

Which function would you use to find the first occurrence of a substring in a string?
A) strlocate() B) strfind() **C) strstr()** D) strchr()

What is the maximum length of a string that can be defined in C?
A) 255 characters B) 1024 characters
C) There is no fixed limit, depends on memory D) 512 characters

What is the worst-case time complexity of bubble sort?
A) O(n) B) O(n log n) **C) O(n^2)** D) O(log n)

Which sorting algorithm is generally considered the fastest for small datasets?
A) Bubble Sort B) Selection Sort **C) Insertion Sort** D) Quick Sort

What is the time complexity of linear search in the worst case?
A) O(1) B) O(log n) **C) O(n)** D) O(n log n)

In a binary search algorithm, what is the main advantage over linear search?
A) It works on unsorted arrays. B) It is easier to implement.
C) It is faster with a time complexity of O(log n). D) It requires less memory.

UNIT – IV

Functions: Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, arrays as parameters, Scope and Lifetime of Variables, storage class, recursion, functions & pointers, function, and arrays.

Functions

Function: A function is a self-contained block of program statements that performs a particular task.

- We have written 'C' programs using three functions namely, main, printf() and scanf().
- Every program must have a main function to indicate where the execution of the program begins and ends.
- If the program is large and if we write all the statements of that large program in main itself, the program may become too complex and large.
- As a result, the task of debugging, testing and even understanding will become very difficult.
- So, the best way to develop a large program is to construct it from smaller pieces (or) modules.
- A module can be a single function (or) a group of related functions carrying out a specific task.
- Usually, it is easier to break down a difficult task into a series of smaller tasks and then to solve those subtasks individually and later combined into a single unit.
- These subtasks are called **User-defined functions.**

Advantages:

Modularity: Divides a program into smaller, manageable units.
Code Reusability: Write once, use multiple times.
Easier Debugging: Troubleshoot functions independently.
Better Readability: Makes code easier to understand.
Reduced Code Redundancy: Avoid repetition.

General form of a function:

```
return_type function-name(argument list)        ----->function header.
{
        local variable declaration;
        statement 1;
        statement 2;                            ------->function body
        .....
        .....
        return(expression);
}
```

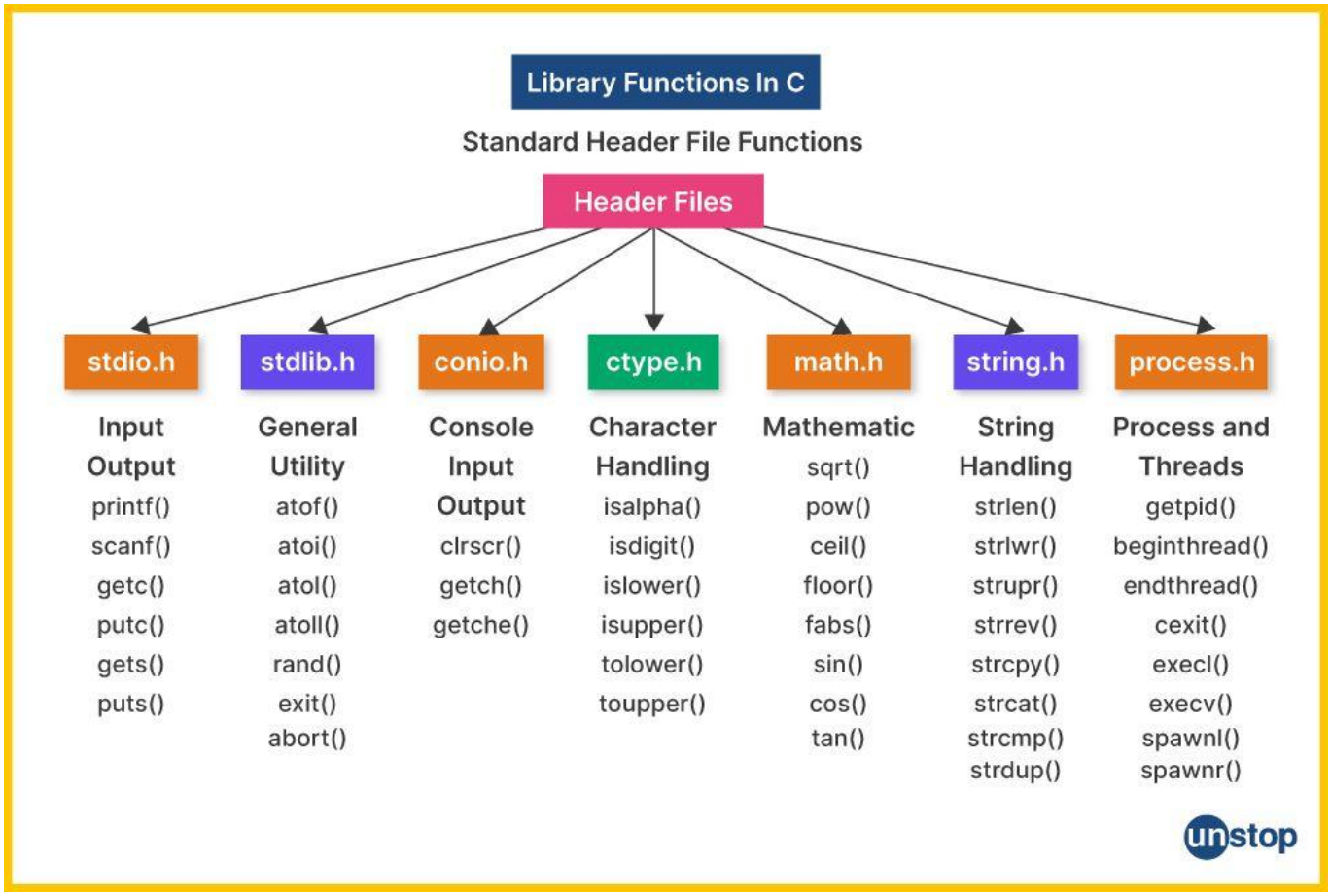
- Here, type specifies the type of data that the function returns.
- The type and argument list are optional.
- An unspecified type is always assumed by the compiler to int. int is the default type when no type specifier is present.
- A function returns a type other than int, it must be explicitly declared.
- The function-name is any valid identifier.

Characteristics of functions:

- Any function can return only one value.
- Parameter argument list is optional.
- Return statement indicates exit from the function and return to the point from where the function was invoked.
- A function can call any number of times.
- A call to the function must end with a semicolon.
- Any 'C' function cannot be defined in other function.
- When a function is not returning any value, void type can be used as return type.
- 'C' allows recursion i.e..., a function can call itself.

Types of Functions:

1) Library Functions: Predefined functions provided by C libraries.
Examples: printf(), scanf(), strlen(), sqrt(), etc.



2) User-Defined Functions: Functions created by the programmer to perform specific tasks.
Ex:

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

User-defined functions in C include several components that collectively define their structure and behavior. Below are the key components of a user-defined function in C:

1) Function Declaration or Prototype
2) Function Definition
3) Function Call

Function declaration: A function declaration tells the compiler about a function's name, return type, and parameters.

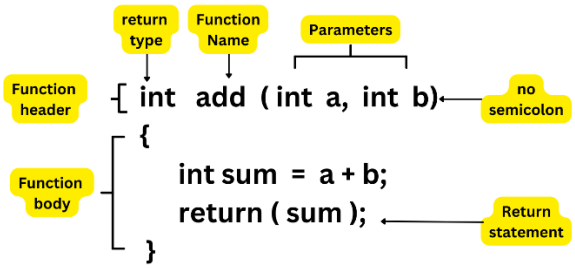
Syntax: return_type function_name(parameters types..);

Ex: int add(int,int);

Function definition: A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and some of the programs can define additional functions.

Syntax:
```
return_type function_name(arguments list)
{
    statements.....
}
```

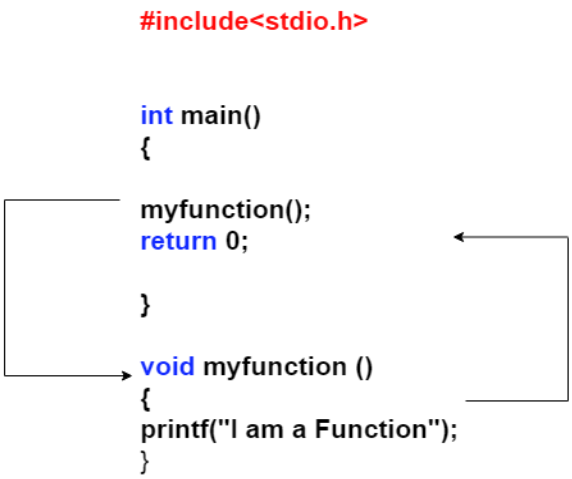
Function Definition



Function call: A function can be called by specifying function name, followed by a list of arguments enclosed in parenthesis and separated by commas. If the function call does not require any arguments, an empty pair of parenthesis must follow the function name.

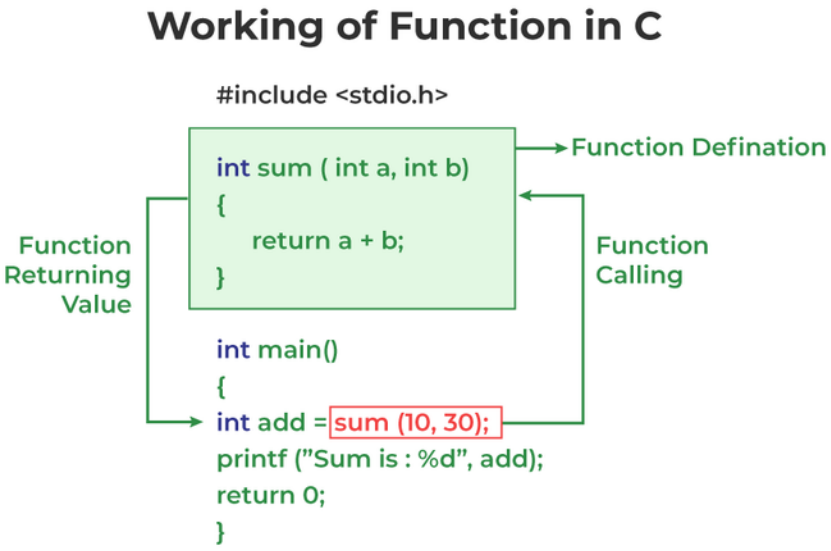
Syntax: fun-name(arg1,arg2,....);

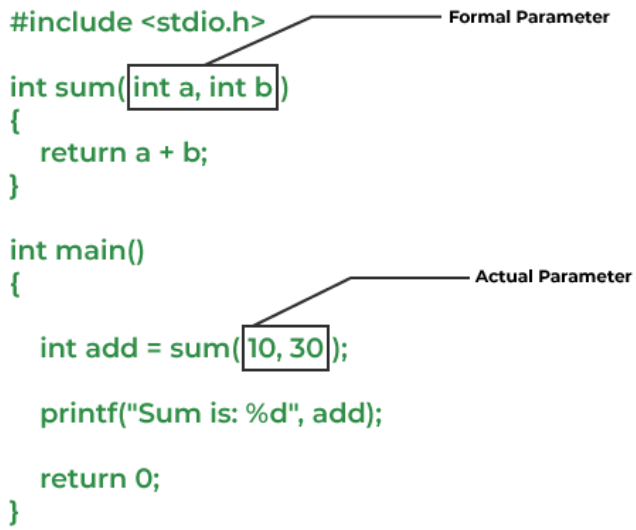
WORKING OF A FUNCTION



Parameters (Optional)

- Variables listed in the function declaration and definition to receive input values when the function is called.
- **Formal Parameters**: Parameters defined in the function header.
  - Example: `int add(int a, int b)` → `a` and `b` are formal parameters.
- **Actual Parameters**: Arguments passed during the function call.
  - Example: `add(5, 3)` → `5` and `3` are actual parameters.





Working with functions in C Programming:

1) Program to demonstrate working with functions in C.

```
#include<stdio.h>
void wish();                        //function prototype
int main()
{
        printf("Before Wish Call....\n");
        wish();                     //function call
        printf("After Wish Call....\n");
}
void wish()                         //function definition
{
        printf("Good Afternoon\n");
}
```

Output:

Before Wish Call....
Good Afternoon
After Wish Call....

2) Program to practice the function calls in C.

```
#include<stdio.h>
void wish();
int main()
{
        printf("Before Wish Call....\n");
        wish();
        printf("After Wish Call....\n");
}
void wish()
{
        printf("Good Afternoon\n");
        displayYourName();
}
void displayYourName()
{
        printf("Rajesh B\n");
}
```

Output:

```
Before Wish Call....
Good Afternoon
Rajesh B
After Wish Call....
```

3) Program to perform arithmetic operations in C using functions.

```
#include<stdio.h>
int add(int,int);
int subtract(int,int);
int main()
{
        int x,y;
        scanf("%d%d",&x,&y);
        int sum=add(x,y);
        int diff=subtract(x,y);
        printf("Addition of %d and %d is: %d\n",x,y,sum);
        printf("Difference of %d and %d is: %d\n",x,y,diff);
        return 0;
}
int add(int a,int b)
{
        return a+b;
}
int subtract(int a,int b)
{
        return a-b;
}
```

4) Program to find the factorial of a give number.

```
#include<stdio.h>
int factorial(int);
int main()
{
   int n,fact;
   scanf("%d",&n);
   fact=factorial(n);
   printf("Factorial of %d is %d\n",n,fact);
   return 0;
}
int factorial(int n)
{
    //write your logic here.
          int f=1,i;
          for(i=n;i>=2;i--)
           {
              f=f*i;
           }
          return f;
}
```

5) Program to display the Fibonacci series up to given value n.

Fibonacci Sequence: Every element in the sequence is generated by the sum of its two previous elements.

Ex: 0 1 1 2 3 5 8 13 21 .....

Input:

30

Output:

0 1 1 2 3 5 8 13 21

```
#include<stdio.h>
void printFibonacci(int);
int main()
{
          int range;
          scanf("%d",&range);
          printFibonacci(range);
    return 0;
}

void printFibonacci(int n)
{
          int a=0,b=1,c;
          printf("%d %d ",a,b);
          c=a+b;
```

```
        while(c<=n)
        {
      printf("%d ",c);
            a=b;
            b=c;
            c=a+b;
        }
}
```

6) Program to find the sum of elements of an array using functions.

```
#include<stdio.h>
int arraySum(int[],int);
int main()
{
        int n,i;
        scanf("%d",&n);
        int arr[n];
        for(i=0;i<n;i++)
        {
            scanf("%d",&arr[i]);
        }
        int result=arraySum(arr,n);
        printf("Sum of array elements is: %d\n",result);
        return 0;
}
int arraySum(int X[],int size)
{
        int sum=0,i;
        for(i=0;i<size;i++)
   {
        sum=sum+X[i];
        }
        return sum;
}
```

Input:

3

11 22 33

Output:

7) Program to read your name and time as an input and display the wish message.

Sample Input:

Aditya Ram

14

Sample Output:

Hi Aditya Ram!

Good Afternoon.

```
#include<stdio.h>
int main()
{
        char name[30];
        int time;
        scanf("%[^\n]s",name);
        scanf("%d",&time);
        wish(name,time);
        return 0;
}
void wish(char word[],int time)
{
        printf("Hi %s\n",word);
        if(time>=0 && time<12)
          printf("Good Morning.\n");
        else if(time>=12 && time <16)
          printf("Good Afternoon\n");
        else
          printf("Good Evening\n");
}
```

Categories of function:

According to the arguments and return values, there are four types of user defined functions.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions with no arguments and return values.

1) Functions with no arguments and no return values:

In this type of function, the called function doesn't take arguments from the calling function and it will not give any return value to the calling function.

2) Functions with arguments and no return values:

In this type of functions, the called function can take the arguments from the calling function but it will return any values to the calling function.

3) Functions with arguments and return values:

In this type of functions, the called function can take the arguments from the calling function and it will give some return values to the calling function.

4) Functions with no arguments and return values:

In this type of functions, the called function returns some value to the calling function. But it doesn't take any arguments from the calling function.

Examples:

```
//No arguments and
//No return value
#include<stdio.h>
void sum();
int main()
{
     sum();
     return 0;
}
void sum()
{
    int x,y;
    scanf("%d%d",&x,&y);
    printf("sum=%d",x+y);
}
```

```
//With arguments and
//No return value
#include<stdio.h>
void sum(int,int);
int main()
{
     int x,y;
     scanf("%d%d",&x,&y);
     sum(x,y);
     return 0;
}
void sum(int p,int q)
{
  printf("sum=%d",p+q);
}
```

```
//With arguments and
//With return value
#include<stdio.h>
int sum(int,int);
int main()
{
     int x,y,z;
     scanf("%d%d",&x,&y);
     z=sum(x,y);
     printf("sum=%d",z);
     return 0;
}
int sum(int p,int q)
{
  return (p+q);
}
```

```
//Without arguments and
//With return value
#include<stdio.h>
int sum();
int main()
{    int z;
     z=sum();
     printf("sum=%d",z);
     return 0;
}
int sum()
{
     int p,q;
     scanf("%d%d",&p,&q);
     return (p+q);
}
```

Parameter passing techniques: Arguments to a function are usually passed in two ways.

1. Call by value 2. Call by reference

Call by value: The call by value is a method of passing arguments to a function and the values of the actual parameters copy to the formal parameters of the function. In this case, changes made to the parameter inside the function have no effect on the actual arguments. By default, C programming uses call by value to pass arguments.

Call by address: The call by Address is a method of passing arguments to a function and copies the address of actual argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. In this case, changes made to the parameters inside the function has effected on the actual arguments.

```c
#include<stdio.h>
void swap(int,int);
int main()
{    int x,y;
     scanf("%d%d",&x,&y);
     printf("Before Swapping\n");
     printf("x=%d y=%d\n",x,y);
     swap(x,y);
     printf("After Swapping\n");
     printf("x=%d y=%d\n",x,y);
     return 0;
}
void swap(int p,int q)
{
     int r;
     r=p;
     p=q;
     q=r;
}
```

Output:

```
10 20
Before Swapping
x=10 y=20
After Swapping
x=10 y=20
```

```c
#include<stdio.h>
void swap(int*,int*);
int main()
{    int x,y;
     scanf("%d%d",&x,&y);
     printf("Before Swapping\n");
     printf("x=%d y=%d\n",x,y);
     swap(&x,&y);
     printf("After Swapping\n");
     printf("x=%d y=%d\n",x,y);
     return 0;
}
void swap(int *p,int *q)
{
     int r;
     r=*p;
     *p=*q;
     *q=r;
}
```

Output:

```
10 20
Before Swapping
x=10 y=20
After Swapping
x=20 y=10
```

Call By Value	Call By Reference
While calling a function, we pass values of variables to it. Such functions are known as **"Call By Values"**.	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as **"Call By References"**.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.

```
#include <stdio.h>

void swap(int *,int *);
int add(int, int);

int main()
{
int a=10,b=20;
printf("\n SUM = %d", add( a, b));
printf("\n Before Swapping a=%d b=%d", a, b);
swap(&a, &b);
printf("\n After Swapping a=%d b=%d", a, b);
return 0;
}

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

void swap(int *p, int *q)
{
int *r;
*r=*p;
*p=*q;
*q=*r;
}
```

Function Prototype

Function Call

Call by Value
a and b are actual parameter

Call by Address / Reference

int – return type
add – function name
x and y – parameter / arguments

Function Definition

Recursion: A function, which invokes itself repeatedly until some condition is satisfied, is called a recursive function.

- The normal function will be called by main function whenever the function name is used.
- On the other hand, the recursive function will be called by itself repeatedly, until some specified condition has been satisfied.
- This technique is used to solve problems whose solution is expressed in terms of successively applying the same set of steps.

Components of Recursion

1. **Base Case**: A condition that stops the recursion. Without it, the recursion would continue indefinitely (leading to a stack overflow error).
2. **Recursive Case**: The part where the function calls itself to break the problem into smaller sub-problems.

For example:

the factorial of a positive integer number 'n' is defined as **n!=n(n-1)(n-2).....1 with 1!=1, 0!=1.**

The above formula can also be written as **n!=n(n-1)!**

For example 5!=(5)(4!)
=5*4*3!
=5*4*3*2!
=5*4*3*2*1!
=5*4*3*2*1
= 120.

In 'C', a recursive function that calculates n! can be written as follows.

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

/*Program to find the factorial of a given number*/

Without Recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
{
        int f;
        for(i=0; i<=x; i++)
                f=f * i;
        return f;
}
```

With Recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
{
        if(x==0)
            return 1;
        else if(x==1)
            return 1;
        else
            return(x*fact(x-1));
}
```

/*C program for generating Fibonacci sequence with recursion*/

```
#include<stdio.h>
#include<conio.h>
int fib(int);
void main( )
{
        int n, f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        for(i=1; i<=n; i++)
        {
                f=fib(i);
                printf("%d \t",f);
         }
        getch( );
}
```

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

Storage classes: Storage class in C decides the part of storage to allocate memory for a variable, it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored.

The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable. There are four storage classes in C those are automatic, register, static, and external.

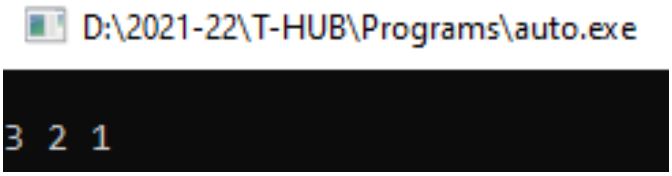
Types of Storage Classes: There are four storage classes in C they are as follows

1. auto 2. register 3. static 4. extern

1. auto:

A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

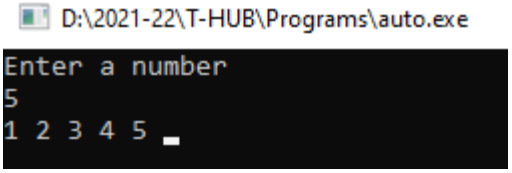
```
#include<stdio.h>
int main()
{
 auto int i=1;
 {
  auto int i=2;
  {
   auto int i=3;
   printf("\n%d ",i);
  }
  printf("%d ",i);
 }
 printf("%d\n",i);
 return 0;
}
```



2. register:

The register specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block. A register declaration is equivalent to an auto declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory. Only a few variables are actually placed into registers, and only certain types are eligible; the restrictions are implementation dependent. However, if a variable is declared register, the unary & (address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.

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



3. static:

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

In C programming, when static is used on a global variable, it causes only one copy of that member to be shared by all the functions.

```c
#include <stdio.h>
/* function declaration */
void func(void);
static int count = 5; /* global variable */
int main()
{
   while(count--) {
      func();
   }
   return 0;
}
/* function definition */
void func( void )
{
   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
```

```
D:\2021-22\T-HUB\Programs\
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

4. extern:

- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files.
- The variables declared as extern is to specify that the variable is defined elsewhere in the program. Default initial value of the extern integral variable is 0 otherwise null.
- The extern variable should be initialized in globally, i.e, we cannot initialize the external variable within any block or method.
- The visibility of the extern variable is to the entire program(file).
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

```c
#include<stdio.h>
#include<conio.h>
int main()
{
     extern int i;
     extern float j;
     extern char *s;
     printf("%d\n",i);
     printf("%f\n",j);
     printf("%s",s);
     getch();
     return 0;
}
int i=10;
float j=3.14;
char *s="Hello";
```

```
D:\2021-22\T-HUB\Programs\extern.exe
10
3.140000
Hello
```

Practice Programs:

1) Program to find the transpose of a matrix using functions in C.

```
#include<stdio.h>
void readMatrix(int,int,int[][]);
void displayMatrix(int,int,int[][]);
void transposeMatrix(int,int,int[][],int[][]);
// Function to read a matrix
void readMatrix(int rows, int cols, int matrix[rows][cols]) {
    printf("Enter elements of the matrix (%d x %d):\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
}
// Function to display a matrix
void displayMatrix(int rows, int cols, int matrix[rows][cols]) {
    printf("Matrix (%d x %d):\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
// Function to find the transpose of a matrix
void transposeMatrix(int rows, int cols, int matrix[rows][cols], int result[cols][rows]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[j][i] = matrix[i][j];
        }
    }
}
int main() {
    int rows, cols;
    // Input rows and columns of the matrix
    printf("Enter the number of rows and columns of the matrix: ");
    scanf("%d %d", &rows, &cols);
    int matrix[rows][cols], transposed[cols][rows];
    // Read the original matrix
    readMatrix(rows, cols, matrix);
    // Display the original matrix
    printf("\nOriginal ");
    displayMatrix(rows, cols, matrix);
```

```
    // Find the transpose of the matrix
    transposeMatrix(rows, cols, matrix, transposed);
    // Display the transposed matrix
    printf("\nTransposed ");
    displayMatrix(cols, rows, transposed);
    return 0;
}
```

Sample Input:

```
Enter the number of rows and columns of the matrix: 2 3
Enter elements of the matrix (2 x 3):
1 2 3
4 5 6
```

Sample Output:

```
Original Matrix (2 x 3):
1 2 3
4 5 6
Transposed Matrix (3 x 2):
1 4
2 5
3 6
```

2) Program to find the maximum element in the given matrix.

```
#include <stdio.h>
void readMatrix(int,int,int[][]);
void displayMatrix(int,int,int[][]);
void findMaxElement (int,int,int[][]);
// Function to read a matrix
void readMatrix(int rows, int cols, int matrix[rows][cols]) {
    printf("Enter elements of the matrix (%d x %d):\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
}
// Function to display a matrix
void displayMatrix(int rows, int cols, int matrix[rows][cols]) {
    printf("Matrix (%d x %d):\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```
// Function to find the maximum element in a matrix
int findMaxElement(int rows, int cols, int matrix[rows][cols]) {
    int max = matrix[0][0]; // Assume the first element is the maximum
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] > max) {
                max = matrix[i][j];
            }
        }
    }
    return max;
}
int main()
{
    int rows, cols;
    // Input rows and columns of the matrix
    printf("Enter the number of rows and columns of the matrix: ");
    scanf("%d %d", &rows, &cols);

    int matrix[rows][cols];

    // Read the matrix
    readMatrix(rows, cols, matrix);

    // Display the matrix
    printf("\n");
    displayMatrix(rows, cols, matrix);

    // Find and display the maximum element
    int max = findMaxElement(rows, cols, matrix);
    printf("\nThe maximum element in the matrix is: %d\n", max);

    return 0;
}
```

MCQ Questions with Answers

1. What is the correct syntax to declare a function in C?

A. function int myFunc(int x); **B. int myFunc(int x);**

C. int myFunc(x int); D. declare int myFunc(x);

2. What is the default return type of a function in C if not specified?

A. int B. void C. float D. Undefined

3. Which of the following is true about functions in C?

A. A function can return multiple values. B. A function must always have a return statement.

C. A function can be called multiple times. D. A function cannot be recursive.

4. What is the purpose of the void keyword in a function declaration?

A. To indicate no parameters. **B. To indicate no return value.**

C. To declare an infinite function. D. To declare a global function.

5. How do you pass an array to a function in C?

A. By value B. By reference **C. By pointer** D. By memory

6. Which of the following is true about recursion in C?

A. Every recursive function must have a base case. B. Recursion is faster than iteration.

C. Recursion uses less memory than iteration.

D. Recursion is allowed only for mathematical functions.

7. What does the following code print?

```
void myFunc() {
    printf("Hello, World!");
}
int main() {
    myFunc();
    return 0;
}
```

A. Hello, World! B. Nothing C. Compile-time error D. Run-time error

8. What happens if a function does not have a return statement in C?

A. Compile-time error

B. Garbage value is returned if int return type is used.

C. It always returns 0.

D. Function execution fails.

9. What is the output of the following code?

```
int add(int a, int b) {
    return a + b;
}
int main() {
    printf("%d", add(2, 3));
    return 0;
}
```

A. 2 B. 3 **C. 5** D. Compile-time error

10. Which storage class is used to define global functions?

A. auto B. register **C. extern** D. static

11. What is the maximum number of arguments a function in C can have?

A. 127 B. 255 **C. Compiler-dependent** D. Infinite

12. What happens if a function is called before its declaration in C?

A. Compile-time error B. The function executes normally.

C. Undefined behavior D. The program doesn't compile.

Answer: A

13. Which of the following is used to stop function execution and return a value?

A. exit B. break C. return D. continue

14. In which section of a program is a user-defined function defined?

A. Preprocessor section B. Main function C. Function definition section D. Library section

15. What does a function prototype specify?

A. The memory layout of a function B. The return type and parameter list of a function

C. The actual implementation of the function D. The scope of the function

16. What is the output of the following code?

```
int foo(int x) {
    return x * 2;
}
int main() {
    printf("%d", foo(4));
    return 0;
}
```

A. 2 B. 4 C. 8 D. Compile-time error

17. What happens when a void function tries to return a value?

A. Compile-time error B. Returns a garbage value

C. Returns 0 D. Function execution continues normally

18. What is the correct syntax to define a recursive function in C?

A. A function calling another function B. A function calling itself

C. A function defined inside another function D. A function with a static keyword

19. What is a library function in C?

A. A user-defined function stored in a library. B. A function available in C's standard library.

C. A global function. D. A private function.

20. What is the purpose of the main function in a C program?

A. To include all header files. B. To serve as the entry point of the program.

C. To declare global variables. D. To initialize the system.

Important Questions

1) Define a function. Explain its advantages with examples.
2) Differentiate between built-in and user-defined functions with examples.
3) Describe function prototype, function definition and function call with an example.
4) Differentiate between call-by-value and call-by-reference with examples.
5) How are arrays passed to functions in C and write a C program to find the largest element in an array using functions.
6) Differentiate between local and global variables with examples.
7) List and explain the different storage classes in C with examples.
8) Define recursion. What are the key components of a recursive function? Write a program to find the Fibonacci series using recursion.

UNIT – V

Introduction to Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, modifying parameters inside functions using pointers, Command line Arguments, Dynamic memory allocation, Null Pointer, generic pointer, dangling pointer

File Handling: Introduction to Files, Using Files in C, Reading from Text Files, Writing to Text Files, Random File Access.

Pointers

Introduction:

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

Key Operators in Pointers:

1) Address-of Operator (&): Gives the memory address of a variable.

Example: int *p = &x;

2) Dereference Operator (*): Accesses the value at the address stored in the pointer.

Example: int value = *p;

Declaration and Initialization of Pointers:

1) Declaration:

datatype *pointer_variablename;

2) Initialization:

pointer_variablename=&variable_name;

Ex:

```
int x=101;
int *ptr;
ptr=&x;
```

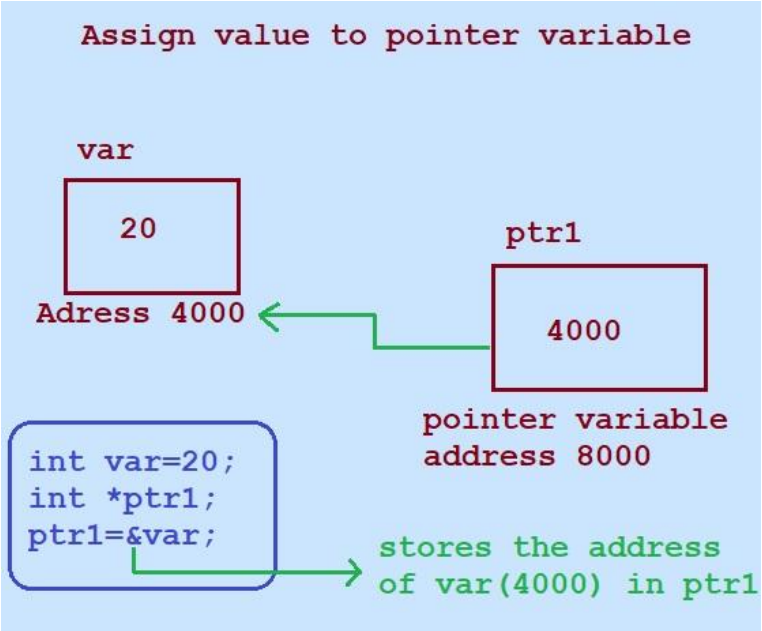
```
#include <stdio.h>
int main()
{
  int x = 10;
  int *ptr = &x;                    // Pointer initialized with address of x
  printf("Address of x: %u\n", ptr);  // %u for pointer address
  printf("Value of x: %d\n", *ptr);   // Dereference to get value
  return 0;
}
```

Output:

Address of x: 6487600
Value of x: 10

Explanation:

int *ptr declares a pointer to an integer,
ptr = &x stores the address of x in ptr.
*ptr accesses the value at that address.

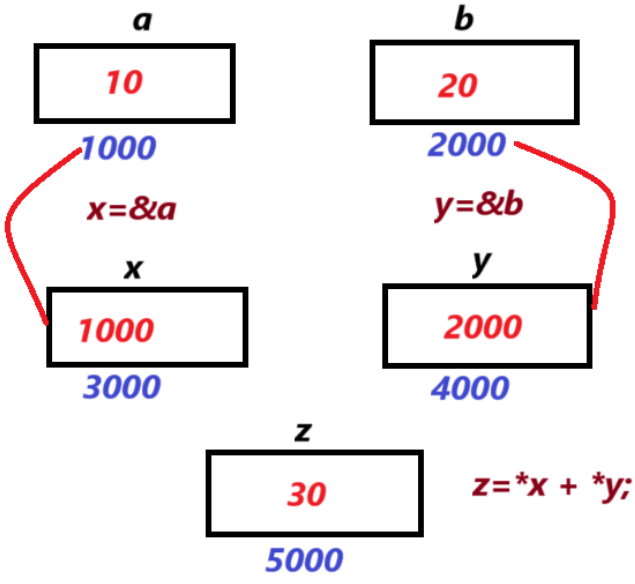


Program to find the addition of two numbers using pointers

```
#include<stdio.h>
int main()
{
   int a=10,b=20;
   int *x,*y;
   x=&a;
   y=&b;
   printf("a = %d b = %d\n",a,b);
   printf("%u %u\n",x,y);
   printf("%d %d\n",*x,*y);
   int z=*x + *y;
   printf("Sum = %d\n",z);
   return 0;
}
```

Output:

```
10 20
6487612 6487608
10 20
Sum = 30
```



Note: Storing the address of one data type in a pointer designed for another data type can lead to undefined behaviour or incorrect memory access.

i.e
```
int x=10;
char *ptr;
ptr=&x;          //Not recommended.
```

Advantages of pointers:

1) Direct Memory Access

Pointers allow direct access to memory locations, enabling efficient manipulation of memory and data.

2) Dynamic Memory Allocation

Pointers are essential for dynamic memory management using functions like malloc(), calloc(), realloc(), and free().

3) Manipulation of Arrays and Strings

Pointers enable efficient traversal and manipulation of arrays and strings.

4) Building Complex Data Structures

Pointers are critical in constructing dynamic data structures like: Linked lists, Trees, Graphs, Queues and stacks.

5) Function Pointers

Pointers can store addresses of functions, enabling dynamic function calls and callback implementations.

6) Pointer to Pointer (Multilevel Pointers)

Enable creation of multi-dimensional arrays dynamically and handle complex data structures more effectively.

Accessing of Array elements with pointers:

In C, arrays and pointers are closely related. A pointer can be used to access and manipulate array elements because the array name itself acts as a pointer to the first element.

Accessing elements:

Using a pointer, array elements can be accessed by incrementing the pointer.

Pointer arithmetic:

Moving the pointer by 1 advance it to the next array element.

Example:

```
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50};
   int *ptr = arr;                          // Pointer to the first element of arr

   // Access array elements using the pointer
   for (int i = 0; i < 5; i++)
   {
      printf("Element %d: %d\n", i, *(ptr + i));
   }
   return 0;
}
```

Explanation:

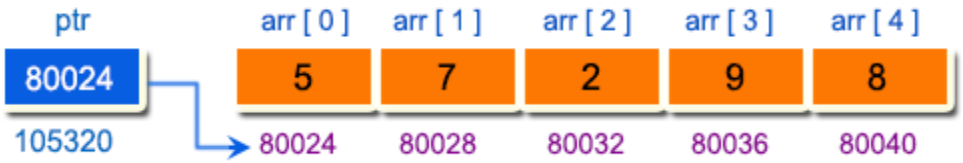
ptr = arr: ptr points to the first element (arr[0]).

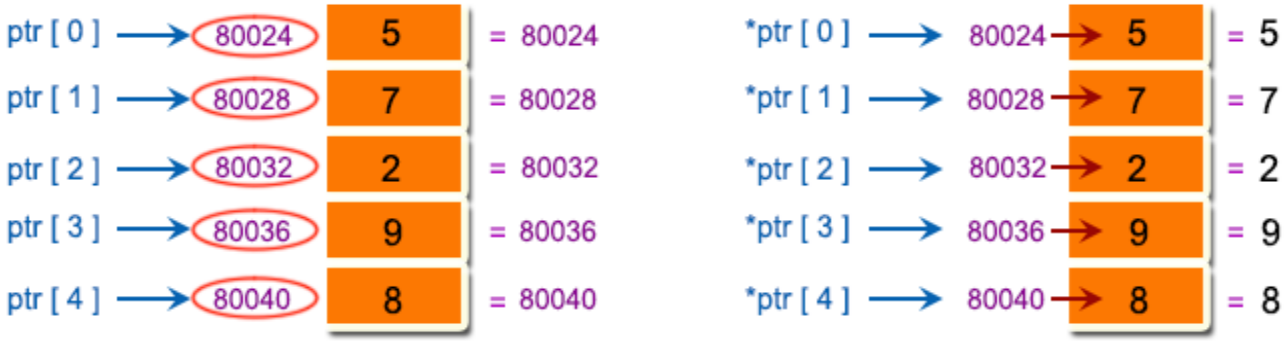
*(ptr + i): Accesses the ith element of the array by pointer arithmetic.

Pointer arithmetic ensures that ptr + i moves to the next integer in memory.

Output:

Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50





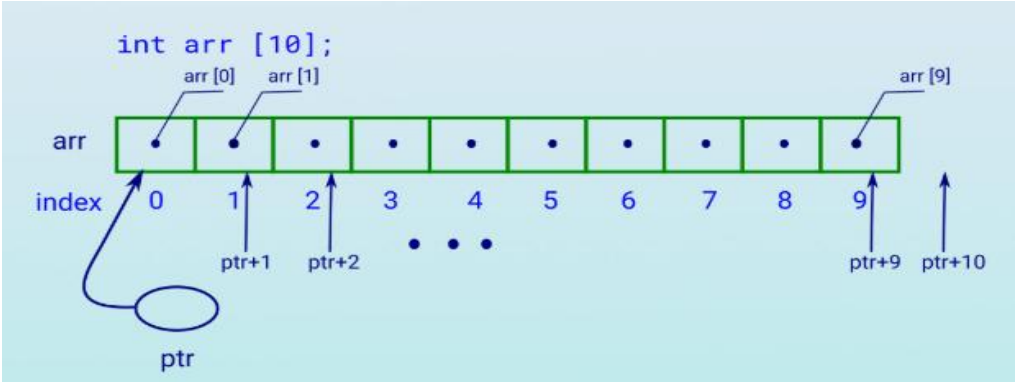
Pointer Arithmetic:

Pointer arithmetic refers to performing arithmetic operations like addition or subtraction on pointers. Since pointers store memory addresses, these operations move the pointer by the size of the data type it points to.

1) Increment (ptr + 1): Moves the pointer to the next element of the data type.

2) Decrement (ptr - 1): Moves the pointer to the previous element.

3) Difference (ptr2 - ptr1): Calculates the number of elements between two pointers



if ptr is points to the array then

(ptr+i) => Base Address of Array + (i*size_of_datatype)

For Example

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

if ptr= 1000                [Assume Base address of arr is 1000]
```

*(ptr+0) => 1000 + 0*4 => 1000 i.e value at 1000 => 10
*(ptr+1) => 1000 + 1*4 => 1004 i.e value at 1004 => 20
*(ptr+2) => 1000 + 2*4 => 1008 i.e value at 1008 => 30

```
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50};
   int *ptr = arr;                          // Pointer to the first element of arr

   printf("Initial value: %d\n", *ptr);  // Output: 10

   ptr++;                                   // Move to the next integer (adds sizeof(int) to the address)
   printf("After increment: %d\n", *ptr);    // Output: 20

   ptr += 2;                                // Move forward by 2 more elements
   printf("After adding 2: %d\n", *ptr);      // Output: 40

   ptr--;                                   // Move back by 1 element
   printf("After decrement: %d\n", *ptr);  // Output: 30
   return 0;
}
```

Output:

Initial value: 10
After increment: 20
After adding 2: 40
After decrement: 30

Explanation:

ptr++ moves the pointer from arr[0] to arr[1].
ptr += 2 moves it from arr[1] to arr[3].
ptr-- moves it back to arr[2].

Program to print the addresses of variables and arrays declared in a program

```
#include<stdio.h>
int main()
{
        int  x=10,i;
        printf("x = %d\n",x);
        printf("Address of x  = %u\n",&x);
        char ch='A';
        printf("Address of ch = %u\n",&ch);

        printf("Addresses of integer array elements are\n");
        int arr[5]={11,22,33,44,55};
```

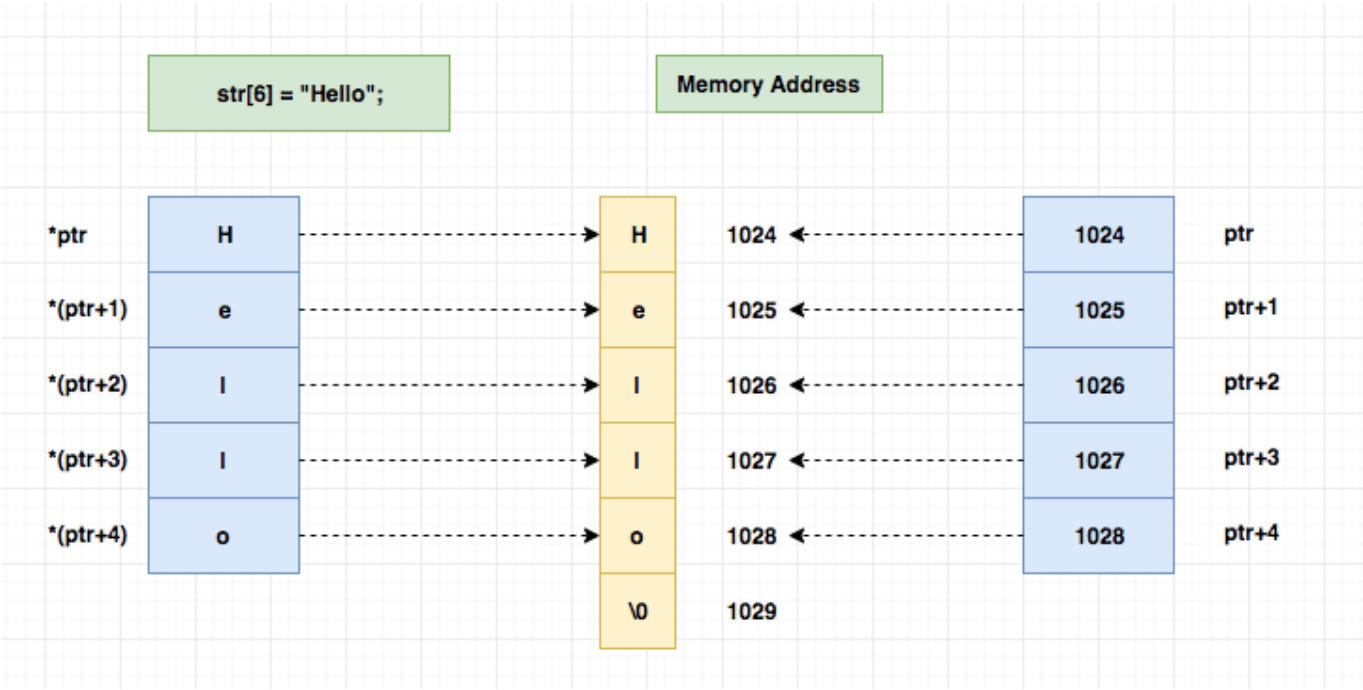
```
        for(i=0;i<5;i++)
        {
                printf("%u ",&arr[i]);
        }
        printf("Addresses of character array elements are\n");
        char chrarr[3]={'H','i','\0'};
        printf("\n");
        for(i=0;i<2;i++)
        {
                printf("%u ",&chrarr[i]);
        }
        return 0;
}
```

Output:

x = 10
Address of x = 6487624
Address of ch = 6487623
Addresses of integer array elements are
6487600 6487604 6487608 6487612 6487616
Addresses of character array elements are
6487584 6487585

Pointers with Character Array:

Pointers can be used to access elements of a character array, which is often used to store strings. The pointer can traverse the array to access each character individually.



```
#include <stdio.h>
int main()
{
   char str[] = "Hello, World!";
   char *ptr = str;                          // Pointer to the first character of str

   // Access each character using the pointer
   while (*ptr != '\0')
   {
         // '\0' marks the end of the string
      printf("%c ", *ptr);                   // Print the character
      ptr++;                                 // Move to the next character
   }
   return 0;
}
```

Output:

H e l l o , W o r l d !

Explanation:

ptr = str: ptr points to the first character 'H'.

*ptr: Dereferences the pointer to get the current character.

ptr++: Moves the pointer to the next character in the array.

'\0': Indicates the end of the string.

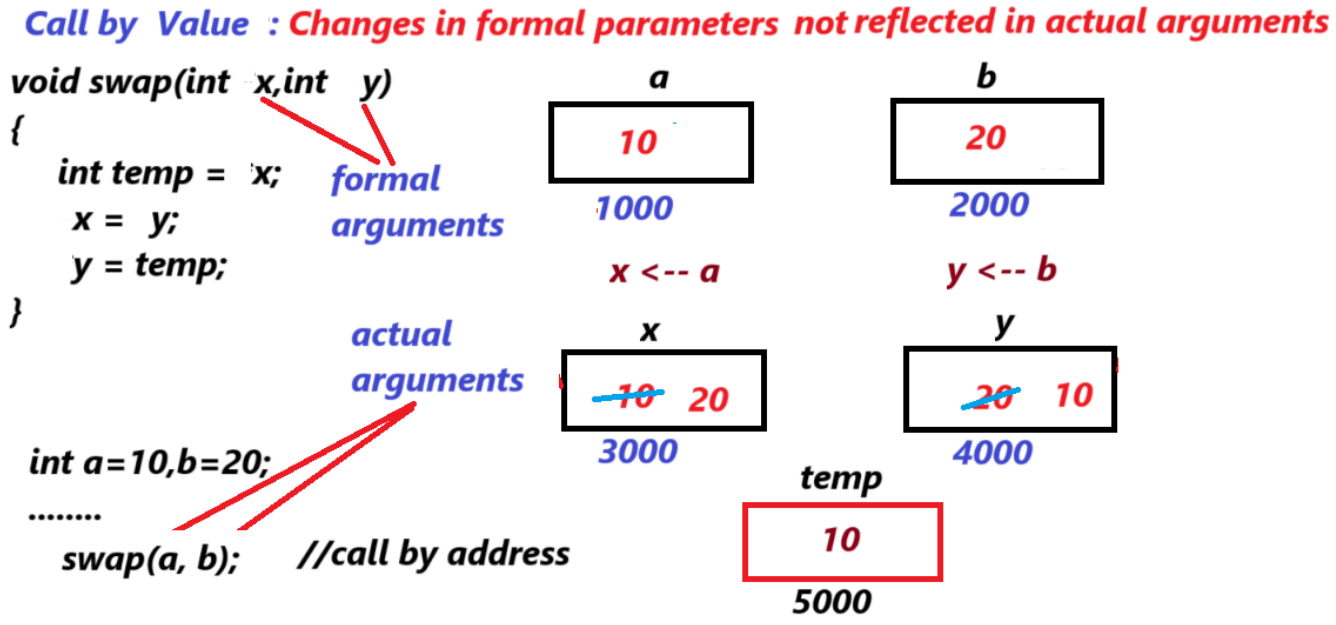
Parameter passing techniques:

Parameter passing techniques determine how arguments are passed to functions in programming.

1. Pass by Value or Call by Value:

A copy of the actual argument is passed to the function. Changes made inside the function do not affect the original variable.

- There are two copies of parameters stored in different memory locations.
- One is the original copy and the other is the function copy.
- Any changes made inside functions are not reflected in the actual parameters of the caller.



Ex: Program to find the swapping of two numbers using call by value

```
#include<stdio.h>
void swap(int,int);
void swap(int x,int y)
{
        int temp=x;
        x=y;
        y=temp;
}
int main()
{
        int a,b;
        printf("Enter any two values\n");
        scanf("%d%d",&a,&b);
        printf("Before swapping a and b values\n");
        printf("%d %d\n",a,b);
        swap(a,b);                      // Call by Value
        printf("After Swapping of a and b values\n");
        printf("%d %d\n",a,b);
        return 0;
}
```

Output:

```
Enter any two values
10 20
Before swapping a and b values
10 20
After Swapping of a and b values
10 20
```

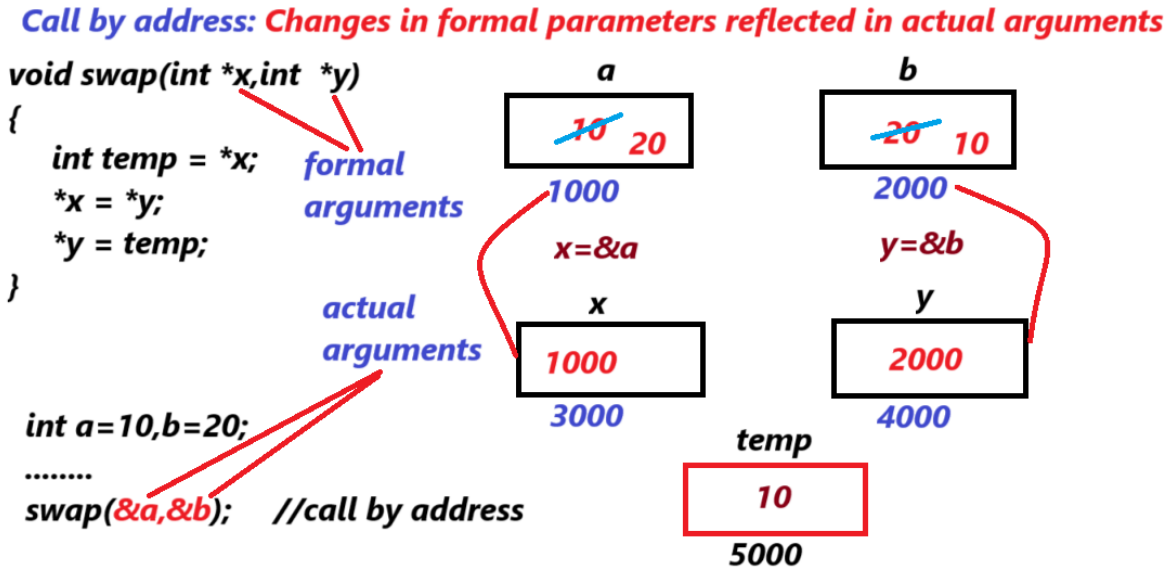
Note:

In Call by value whatever modification done on formal arguments will not shown effect on original arguments.

2. Pass by Address or Call by Address:

In pass by Address, the address of the variable is passed, allowing the function to modify the original variable.

- Both the actual and formal parameters refer to the same locations.
- Any changes made inside the function are actually reflected in the actual parameters of the caller.



Ex: Program to find the swapping of two numbers using call by address.

```
#include<stdio.h>
void swap(int *,int *);
void swap(int *x,int *y)
{
        int temp=*x;
        *x=*y;
        *y=temp;
}
int main()
{
        int a,b;
        printf("Enter any two values\n");
        scanf("%d%d",&a,&b);
        printf("Before swapping a and b values\n");
        printf("%d %d\n",a,b);
        swap(&a,&b);            //Call by Address
        printf("After Swapping of a and b values\n");
        printf("%d %d\n",a,b);
        return 0;
}
```

Output:

Enter any two values
10 20
Before swapping a and b values
10 20
After Swapping of a and b values
20 10

Note:

In Call by Address whatever modification done on formal arguments will show that effect on Actual arguments.

Command line Arguments:

Command-line arguments allow users to pass input values to a C program when it is executed. These arguments are passed to the main() function via two parameters: **argc** and **argv.**

Syntax of main Function with Command-Line Arguments:

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

where

argc (Argument Count):

- An integer representing the number of command-line arguments passed.
- Includes the name of the program, so argc is always at least 1.

argv (Argument Vector):

- An array of character pointers (strings) representing the actual arguments passed.
- argv[0] contains the program's name.
- argv[1] to argv[argc-1] contain the arguments provided by the user.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf("Number of arguments: %d\n", argc);

   for (int i = 0; i < argc; i++)
   {
      printf("Argument %d: %s\n", i, argv[i]);
   }
   return 0;
}
```

Output:

```
C:\Users\mcnu5\OneDrive\Documents>Commandline1.exe 3581 Srinu_M THUB
Number of arguments: 4
Argument 0: Commandline1.exe
Argument 1: 3581
Argument 2: Srinu_M
Argument 3: THUB
```

Program to find the Sum of Two Numbers Passed as Command-line Arguments

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

int main(int argc, char *argv[])
{
   if (argc != 3)
  {
      printf("Usage: %s num1 num2\n", argv[0]);
      return 1;
   }
   int num1 = atoi(argv[1]);
   int num2 = atoi(argv[2]);
   int sum = num1 + num2;
```

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

Output:

```
C:\Users\mcnu5\OneDrive\Documents>Commandline2.exe 25 35
Sum: 60
```

Null Pointer:

A **null pointer** in C is a pointer that doesn't point to any valid memory location. It is often used to signify that the pointer is not currently referencing any data.

In C, the macro NULL represents a null pointer constant.

Syntax:

```
int *ptr = NULL;
```

Here, ptr is a pointer that does not point to any memory location.

Example:

```
#include <stdio.h>
int main() {
    int *ptr = NULL;                    // Initialize a pointer to NULL
    if (ptr == NULL)
    {
        printf("Pointer is null.\n");
    }
    else
    {
        printf("Pointer is not null.\n");
    }
    return 0;
}
```

Output:

```
Pointer is null.
```

Dangling Pointer:

A **dangling pointer** is a pointer that references a memory location that has been **freed**, **deleted**, or **deallocated**, meaning the memory is no longer valid or accessible. Dereferencing a dangling pointer results in undefined behaviour, often leading to crashes or memory corruption.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr = (int *)malloc(sizeof(int));          // Allocate memory
    *ptr = 42;                                      // Assign a value
    printf("Value: %d\n", *ptr);

    free(ptr);                                      // Free the allocated memory
```

```
    // ptr is now a dangling pointer
    printf("Accessing freed memory: %d\n", *ptr);  // Undefined behaviour
    return 0;
}
```

Generic Pointer:

A **generic pointer** in C is a pointer of type void * that can hold the address of any data type. Unlike other pointers (e.g., int *, char *), a generic pointer does not have a specific data type associated with it, making it versatile for handling various types of data.

Syntax:

```
    void *ptr;
```

Example:

```
#include <stdio.h>

int main()
{
    int num = 10;
    void *ptr;                  // Declare a generic pointer

    ptr = &num;                 // Store the address of an integer

    // Cast to int * before dereferencing
    printf("Value: %d\n", *(int *)ptr);
    return 0;
}
```

Output:

Value: 10

Dynamic Memory Allocations in C:

Dynamic memory allocation in C allows the program to allocate memory at runtime (instead of compile-time) using functions from the standard library. This flexibility is crucial when the exact size of the required memory is not known in advance.

Why Use Dynamic Memory Allocation?

1. **Efficient Memory Use**: Allocate only the memory needed at runtime.
2. **Flexible Arrays**: Create variable-sized arrays.
3. **Data Structures**: Used for dynamic data structures like linked lists, trees, stacks, and queues.

Key Functions for Dynamic Memory Allocation

Function	Description	Returns
`malloc()`	Allocates memory block of specified size (in bytes)	`void *`
`calloc()`	Allocates and initializes memory block	`void *`
`realloc()`	Reallocates memory block to new size	`void *`
`free()`	Deallocates previously allocated memory	`void`

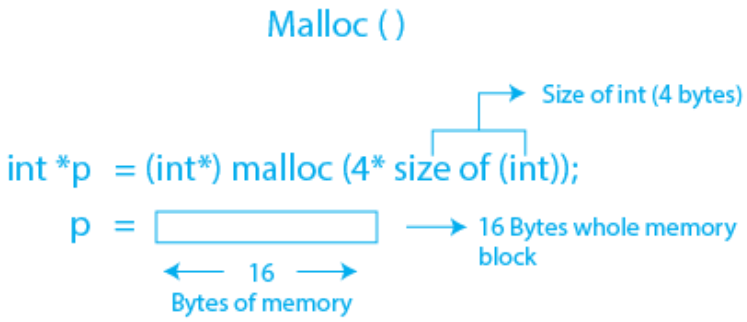
1) **malloc():** malloc() is a standard library function in C that dynamically allocates a block of memory on the heap. The allocated memory remains uninitialized, meaning it contains garbage values.

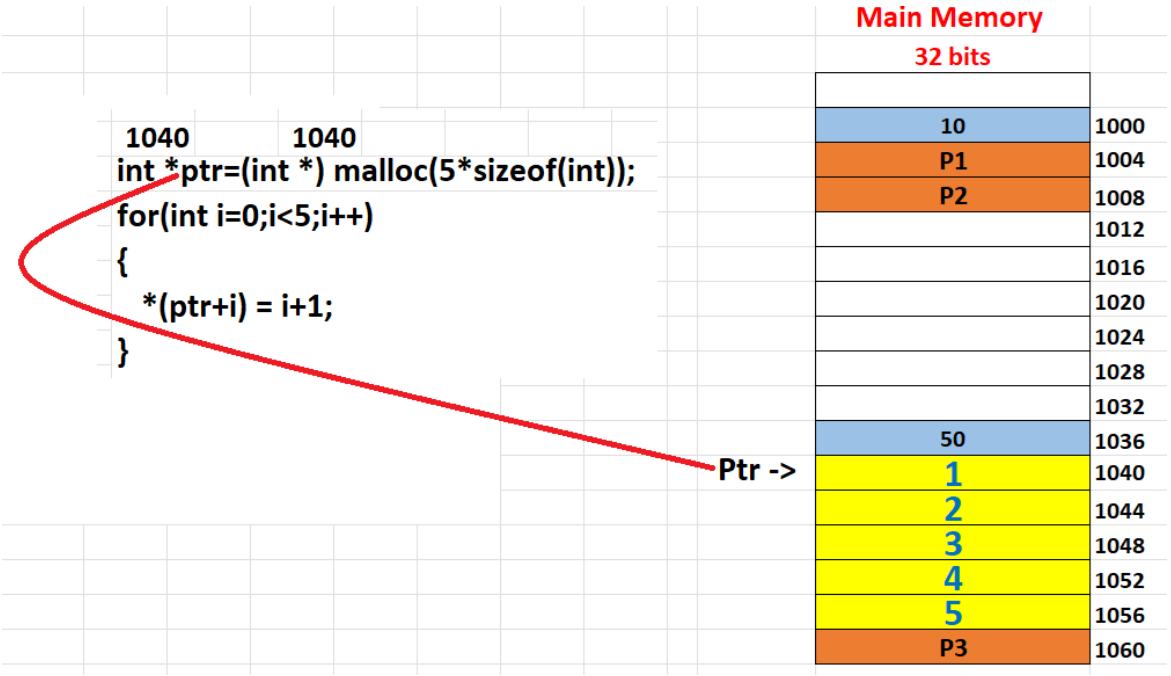
Syntax:

```
void *malloc(size_t size);
```

size: Number of bytes to allocate.

Returns: A void * pointer to the allocated memory. If memory allocation fails, malloc() returns NULL.





Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int n = 5;
   // Allocate memory for 5 integers
   int *ptr = (int *)malloc(n * sizeof(int));

   if (ptr == NULL)
   {
      printf("Memory allocation failed.\n");
      return 1;                          // Exit the program if memory allocation fails
   }
```

```
    // Initialize and display the array elements
    for (int i = 0; i < n; i++)
    {
        ptr[i] = i + 1;                    // Assign values
    }
    printf("Allocated and initialized array:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(ptr);
    return 0;
}
```

Output:

Allocated and initialized array:
1 2 3 4 5

2) **calloc():** calloc() is a standard library function in C used to dynamically allocate memory for an array. It initializes all allocated memory to zero, unlike malloc(), which leaves the memory uninitialized.

Syntax:

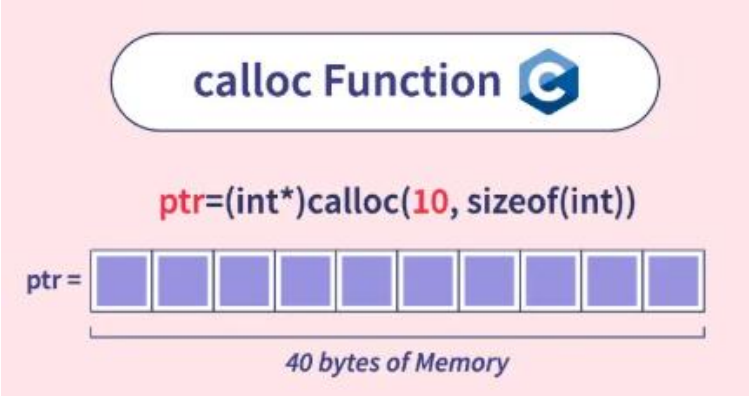
```
void *calloc(size_t num, size_t size);
```

Where

num: Number of elements to allocate.

size: Size of each element in bytes.

Returns: A void * pointer to the allocated memory. If the allocation fails, it returns NULL.



Example:

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

```
    // Allocate memory for an array of 5 integers using calloc
    int *ptr = (int *)calloc(n, sizeof(int));

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

    // Display the initialized values (should be all zeros)
    printf("Array elements after calloc initialization:\n");
    for (int i = 0; i < n; i++)
   {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(ptr);
    return 0;
}
```

Output:

```
Array elements after calloc initialization:
0 0 0 0 0
```

Working with calloc() in 2D Arrays:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int rows = 3, cols = 4;
    int **matrix = (int **)calloc(rows, sizeof(int *));

    if (matrix == NULL)
     {
        printf("Memory allocation failed.\n");
        return 1;
    }
    // Allocate memory for each row
    for (int i = 0; i < rows; i++)
     {
        matrix[i] = (int *)calloc(cols, sizeof(int));
        if (matrix[i] == NULL)
        {
            printf("Memory allocation for row %d failed.\n", i);
```

```
        return 1;
      }
    }
    // Initialize and print the matrix
    printf("2D array initialized by calloc:\n");
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        printf("%d ", matrix[i][j]);              // All elements are zero
      }
      printf("\n");
    }
    // Free the allocated memory
    for (int i = 0; i < rows; i++)
    {
      free(matrix[i]);
    }
    free(matrix);
    return 0;
}
```

Output:

```
2D array initialized by calloc:
0 0 0 0
0 0 0 0
0 0 0 0
```

3) **realloc():** realloc() (short for **reallocation**) is a function in C used to resize a previously allocated memory block. It can increase or decrease the size of the memory block by preserving the existing data.

Syntax:

```
void *realloc(void *ptr, size_t new_size);
```

Example:

```
int*arr;
arr = (int*)calloc(5,sizeof(int));
.....
.....
arr=(int*)realloc(arr,sizeof(int)*10);
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n = 3;

  // Allocate memory for 3 integers
  int *ptr = (int *)malloc(n * sizeof(int));
```

```
    if (ptr == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Initialize the array
    for (int i = 0; i < n; i++)
    {
        ptr[i] = i + 1;                    // Assign values: 1, 2, 3
    }

    printf("Initial array: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Reallocate memory for 5 integers
    n = 5;
    ptr = (int *)realloc(ptr, n * sizeof(int));

    if (ptr == NULL)
    {
        printf("Reallocation failed.\n");
        return 1;
    }

    // Initialize new elements
    for (int i = 3; i < n; i++)
    {
        ptr[i] = i + 1;                    // Assign values: 4, 5
    }

    printf("Array after reallocation: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(ptr);

    return 0;
}
```

Output:
Initial array: 1 2 3
Array after reallocation: 1 2 3 4 5

4) **free():** The free() function in C is used to deallocate or release memory that was previously allocated by dynamic memory allocation functions like malloc(), calloc(), or realloc(). When memory is no longer needed, free() is called to return it to the heap, allowing other parts of the program to use that memory.
Syntax:
void free(void *ptr);
ptr: A pointer to the memory block that needs to be freed.
Returns: free() does not return any value.

Function Pointers:
A **function pointer** in programming (primarily in languages like C and C++) is a pointer that stores the address of a function. It allows you to invoke a function indirectly through the pointer, enabling dynamic function calls and flexibility in function selection during runtime.
Syntax:
return_type (*pointer_name)(parameter_list);
Ex:

```
#include <stdio.h>
// Function definition
void displayMessage() {
   printf("Hello, Function Pointer!\n");
}
int main()
{
   // Declare a function pointer
   void (*funcPtr)();

   // Assign the address of the function
   funcPtr = displayMessage;

   // Call the function via the pointer
   funcPtr();  // Output: Hello, Function Pointer!
   return 0;
}
```

Implement Simple calculator using function pointers

```
#include <stdio.h>
// Function prototypes for basic arithmetic operations
int add(int a, int b) {
   return a + b;
}
int subtract(int a, int b) {
   return a - b;
}
```

```
int multiply(int a, int b) {
    return a * b;
}
int divide(int a, int b) {
    if (b == 0) {
        printf("Error: Division by zero!\n");
        return 0;
    }
    return a / b;
}
int main() {
    int choice, a, b;
    int (*operation)(int, int);  // Function pointer

    // Display menu
    printf("Simple Calculator\n");
    printf("1. Add\n");
    printf("2. Subtract\n");
    printf("3. Multiply\n");
    printf("4. Divide\n");
    printf("Enter your choice (1-4): ");
    scanf("%d", &choice);

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Select the function based on user choice
    switch (choice) {
        case 1:
            operation = add;
            break;
        case 2:
            operation = subtract;
            break;
        case 3:
            operation = multiply;
            break;
        case 4:
            operation = divide;
            break;
        default:
            printf("Invalid choice!\n");
            return 1;
    }
    // Call the selected function via the pointer
    int result = operation(a, b);
```

```
    printf("Result: %d\n", result);
    return 0;
}
```

File Handling:

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program.

Why Use File Handling?

- Persistent data storage.
- Efficient handling of large datasets.
- Facilitates modular programming by separating data and logic.
- Allows data sharing between different programs.

Types of Files:

1. **Text Files**: Store data as plain text, human-readable (e.g., .txt files).
2. **Binary Files**: Store data in binary format, efficient for numerical data and structures.

File Operations in C:

C supports the following primary file operations:

1. **Creating a file**: Initialize a new file for data storage.
2. **Opening a file**: Open an existing or new file.
3. **Reading a file**: Retrieve data from the file.
4. **Writing to a file**: Add new data or overwrite existing data.
5. **Closing a file**: Free the resources associated with the file.

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file.

A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Steps to working with files:

Step-1: Declare a file pointer: It is used to store the address of a file to access.

```
FILE *fp;
```

Step-2: Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

FILE *fopen(const char * filename, const char * mode);

The fopen() function accepts two parameters:

The **file name** (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".

The **mode** in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

Ex:

```
fp = fopen("example.txt", "w");
if (fp == NULL)
{
        printf("Error opening file.\n");
        return 1;
}
```

Step-3: Perform read/write operations

Ex:

```
fprintf(fp, "Hello, File Handling in C!");          //To write the content to a file.
```

Step-4: Close the file using fclose.

```
fclose(fp);
```

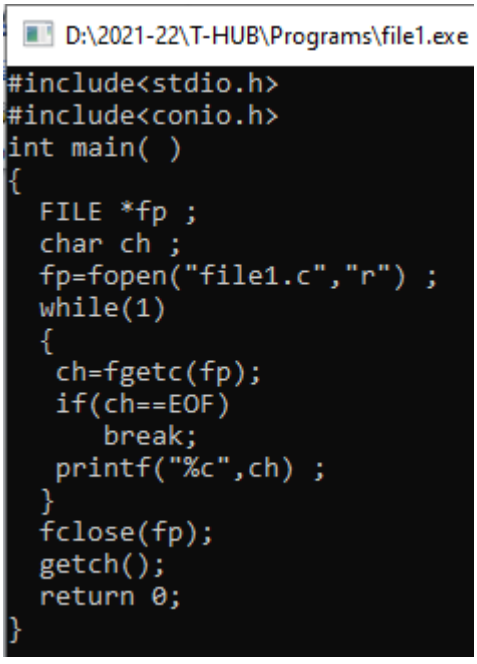
Practice Programs:

1. Write a C program to write the data to a file.

```
#include <stdio.h>
int main()
{
   FILE *fp;

   // Open file in write mode
   fp = fopen("example.txt", "w");
   if (fp == NULL)
   {
      printf("Error opening file.\n");
      return 1;
   }

   // Write data to the file
   fprintf(fp, "Hello, File Handling in C!\n");

  //Close the file
   fclose(fp);

   printf("Data written successfully.\n");
   return 0;
}
```

2. Program to read the data from a file and display it on the screen

```
#include<stdio.h>
#include<conio.h>
int main()
{
   FILE *fp;
   char ch;
   fp=fopen("file1.c","r");
   while(1)
   {
    ch=fgetc(fp);
    if(ch==EOF)
       break;
    printf("%c",ch);
   }
   fclose(fp);
   getch();
   return 0;
}
```



fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax: int fgetc(FILE *stream)

fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax: int fputc(int c, FILE *stream)

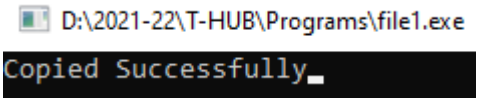
Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

int fclose(FILE *fp);

3. Program to copy the data from one file to another file.

```
#include<stdio.h>
#include<conio.h>
int main()
{
   FILE *fp1,*fp2;
   char c;
   fp1=fopen("file1.c","r");
   fp2=fopen("copy.c","w");
   while((c=fgetc(fp1))!=EOF)
   {
      fputc(c,fp2);
   }
   printf("Copied Successfully");
   fclose(fp1);
   fclose(fp2);
   getch();
   return 0;
}
```



Random Access File:

Random accessing of files in C language can be done with the help of the following functions –

- ftell ( )
- rewind ( )
- fseek ( )

ftell(): The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax: int ftell(File *fp);

Example: int n = ftell (file pointer)

rewind(): The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax: void rewind(FILE *stream)

fseek():

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

int fseek(FILE *stream, long int offset, int position)

Offset: The no of positions to be moved while reading or writing.

It can be either negative (or) positive.

Positive - forward direction.

Negative – backward direction.

Position

It can have three values, which are as follows

0 – Beginning of the file.

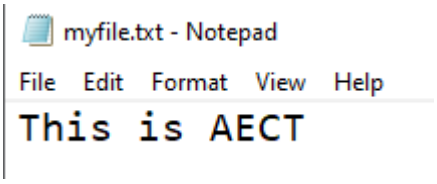
1 – Current position.

2 – End of the file.

There are 3 constants used in the fseek() function for position:

SEEK_SET, SEEK_CUR and SEEK_END.

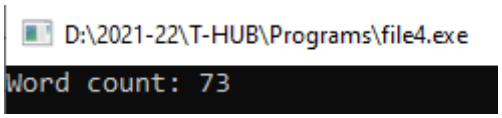
```c
#include<stdio.h>
int main()
{
    FILE *fp;
    fp = fopen("myfile.txt","w+");
    fputs("This is ACET",fp);
    fseek( fp, 8,SEEK_SET);
    fputs("AEC",fp);
    fclose(fp);
    return 0;
}
```



Write a C program to read a text file and count the characters and words in it?[2018]

```c
#include<stdio.h>
int main()
{
    FILE *fp;
    int c=0;
    char ch;
    fp=fopen("file1.c","r");
    while((ch=fgetc(fp))!=EOF)
      c++;
    printf("Char count: %d",c);
    return 0;
}
```


```c
#include<stdio.h>
int main()
{
    FILE *fp;
    int w_c=0;
    char ch;
    fp=fopen("file1.c","r");
    while((ch=fgetc(fp))!=EOF)
    {
       if(ch==' ' || ch=='\n')
           w_c++;
    }
    printf("Word count: %d",w_c);
    return 0;
}
```



Working with fread and fwrite:

fread and fwrite are standard library functions in C, used for performing **binary file I/O**. They allow efficient reading from and writing to files in chunks, making them well-suited for operations on binary files like images, audio, and video.

Syntax:

size_t fread(void *ptr, size_t size, size_t count, FILE *stream);

Where **ptr**: Pointer to the memory block where data will be stored.

size: Size of each element to be read (in bytes).

count: Number of elements to be read.

stream: File pointer to the open file.

Write a C program to copy the content from one image file to another.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *sourceFile, *destFile;
   char buffer[1024];
   size_t bytesRead;

   // Open the source image file in binary read mode
   sourceFile = fopen("source.jpg", "rb");
   if (!sourceFile) {
      perror("Error opening source file");
      return 1;
   }

   // Open the destination image file in binary write mode
   destFile = fopen("copy.jpg", "wb");
   if (!destFile) {
      perror("Error opening destination file");
      fclose(sourceFile);
      return 1;
   }

   // Copy data from source to destination
   while ((bytesRead = fread(buffer, 1, sizeof(buffer), sourceFile)) > 0)
   {
      fwrite(buffer, 1, bytesRead, destFile);
   }
   // Close the files
   fclose(sourceFile);
   fclose(destFile);

   printf("Image file copied successfully.\n");
   return 0;
}
```

Source.jpg

Copy.jpg

Important Questions:

1. Define Pointer. Explain the significance of the * (dereferencing operator) and & (address-of operator) in pointers.
2. Explain how pointer arithmetic depends on the data type of the pointer.
3. Write a program to traverse an array using pointers.
4. Explain the parameter passing techniques in C. Write a C program to swap 2 numbers using call by value and address.
5. Define and differentiate between null pointers, generic pointers, and dangling pointers.
6. Differentiate between malloc, calloc, realloc, and free with examples.
7. Explain the significance of `argc` and `argv` in C. Write a program to calculate the sum of numbers passed as command-line arguments.
8. What are files in C? Differentiate between text files and binary files. Explain the modes in which a file can be opened using the `fopen` function.
9. Write a program to read data from a text file and display it on the console.
10. Write a C program to copy the content from one file to another.
11. What is random access in file handling? How is it achieved in C?
12. Write a program to demonstrate the use of fseek and ftell functions.