AUTOMOTIVE DOMAIN

What is the automotive domain…?

The 'Automotive Domain' refers to the area of computer science focused on the development of software applications for vehicles, including innovations driven by electronics, software, sensors, and communication technologies to enhance vehicle performance, safety, and comfort.The automotive domain refers to the industry and ecosystem surrounding the development, manufacturing, marketing, and sales of motor vehicles. This domain covers a wide range of activities and technologies, including:

Key Areas of the Automotive Domain

- Vehicle Manufacturing:
  - Involves the design, engineering, and production of motor vehicles like cars, trucks, buses, and motorcycles.
  - Automakers like GM, Ford, Tesla, Toyota, and Volkswagen are key players in this space.
- Autonomous Vehicles (AVs):
  - This segment focuses on self-driving cars and vehicles equipped with advanced driver-assistance systems (ADAS).
  - Technologies like LiDAR, radar, cameras, and AI are used to allow vehicles to perceive their environment and navigate without human input.
- Electric Vehicles (EVs):
  - Battery Electric Vehicles (BEVs) and hybrid vehicles (which combine an internal combustion engine with an electric motor) are becoming increasingly important in the automotive domain.
  - Companies like Tesla, GM, and Nissan have been leaders in this transition to cleaner, more sustainable energy sources.
- Connected Vehicles:
  - Vehicles are equipped with internet connectivity that enables features like GPS, remote diagnostics, infotainment, and over-the-air (OTA) updates.
  - Connected cars can communicate with other devices (e.g., smartphones) and infrastructure (e.g., traffic systems).

- Advanced Driver-Assistance Systems (ADAS):
  - These systems provide features such as lane-keeping assist, adaptive cruise control, automatic emergency braking, and parking assistance to enhance vehicle safety.

GM Cars and People Tech

General Motors (GM) is a major automotive manufacturer known for producing a wide range of vehicles under brands like Chevrolet, Buick, GMC, and Cadillac. People Tech is a company that provides technology solutions and services, often in sectors like automotive, where it might work with manufacturers like GM on projects involving:

- Software development for connected and autonomous vehicles.
- Advanced vehicle infotainment systems.
- Data-driven solutions for vehicle diagnostics, safety, and performance improvements.

There may be specific car releases or technology projects from GM through collaborations with People Tech that involve innovations in these areas, especially around software and systems integration.

IDE

An IDE (Integrated Development Environment) is a software application that provides a comprehensive set of tools for software development. It is designed to help developers write, test, debug, and manage their code efficiently. IDEs streamline the coding process by integrating various tasks and tools into a single user interface.

Key Features of an IDE:

- Code Editor:
  - A text editor specifically designed for writing and editing source code.
  - It often includes features like syntax highlighting, code completion, and error detection.
- Compiler/Interpreter:

- ➢ A compiler translates the written code (source code) into machine code (binary code) that can be executed by the computer.
- ➢ An interpreter runs code directly without converting it into machine language.

❖ Debugger:

- ➢ A tool that helps developers find and fix errors (bugs) in their code by allowing them to step through the program's execution line by line.

Popular IDEs for Different Languages:

- ❖ C/C++: Visual Studio, Code::Blocks, CLion.
- ❖ Java: IntelliJ IDEA, Eclipse.
- ❖ Python: PyCharm, Spyder, IDLE.
- ❖ Web Development: Visual Studio Code, Sublime Text, Atom (though these are technically text editors with plugins to behave like IDEs).

[WEEK 01]: DAY 1

Introduction to Programming

What is Programming?

Programming is the process of creating a set of instructions that a computer can execute to perform specific tasks. These instructions are written in a programming language, which serves as a medium for humans to communicate with computers. Programming involves problem-solving, logic, and algorithm design, and it is used to develop software applications, systems, and tools.

Compiler	Interpreter
• A compiler takes the entire program in one go.	• An interpreter takes a single line of code at a time.
• The compiler generates an intermediate machine code.	• The interpreter never produces any intermediate machine code.
• The compiler is best suited for the production environment.	• An interpreter is best suited for a software development environment.
• The compiler is used by programming languages such as C, C ++, C #, Scala, Java, etc.	• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.

Setting Up the Programming Environment

To begin programming, you need to set up an appropriate environment for the programming language you are using. Here's a brief overview for some popular languages:

1. **C/C++**:
    - **Tools Needed**: A compiler (like GCC or Clang) and a text editor or IDE (like Code::Blocks, Visual Studio, or Eclipse).
    - **Installation**: Install a development environment or compiler for your OS (e.g., MinGW for Windows, Xcode for macOS, or a package manager for Linux).
2. **Java**:
    - **Tools Needed**: JDK (Java Development Kit) and an IDE (like Eclipse, IntelliJ IDEA, or NetBeans).
    - **Installation**: Download and install the JDK, and set up your IDE.
3. **Python**:
    - **Tools Needed**: Python interpreter and an IDE (like PyCharm, Visual Studio Code, or Jupyter Notebook).
    - **Installation**: Download Python from the official website and set it up on your system.

Basic Syntax and Structure of Each Language

Here’s a quick overview of the basic syntax and structure of C, C++, Java, and Python.

C

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n"); // Print to console
    return 0; // Exit status
}
```

C++

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl; // Print to console
    return 0; // Exit status
}
```

JAVA

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!"); // Print to console
    }
}
```

PYTHON

```
print("Hello, World!") # Print to console
```

Procedure Oriented vs. Object Oriented Programming

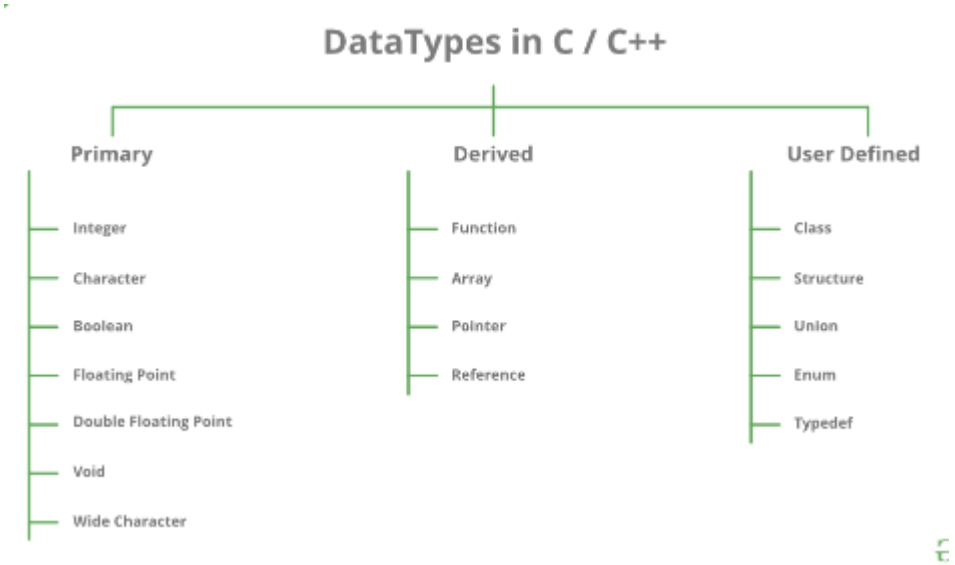
Procedural Programming Language	Object Oriented Programming Language
1. Program is divided into functions.	1. Program is divide into classes and objects..
2. The emphasis is on doing things.	2. The emphasis on data.
3. Poor modeling to real world problems.	3. Strong modeling to real world problems.
4. It is not easy to maintain project if it is too complex.	4. It is easy to maintain project even if it is too complex.
5. Provides poor data security.	5. Provides strong data Security.
6. It is not extensible programming language.	6. It is highly extensible programming language.
7. Productivity is low.	7. Productivity is high.
8. Do not provide any support for new data types.	8. Provide support to new Data types.
9. Unit of programming is function.	9. Unit of programming is class.
10. Ex. Pascal , C , Basic , Fortran.	10. Ex. C++ , Java , Oracle.

[WEEK 01]: DAY 2

1. Variables and Data Types

Variables are used to store data that can be changed during program execution. Each variable has a specific data type, which defines the kind of data it can hold. Common data types include:

- Primitive Types:
    - int: Integer values (e.g., 1, 42, -5)
    - float: Floating-point numbers (e.g., 3.14, -0.001)
    - char: Single characters (e.g., 'a', 'Z')
    - boolean: Represents true or false
    - string: A sequence of characters (text), typically enclosed in double quotes (e.g., "Hello, World!")



EXAMPLE:

```
#include <iostream>
using namespace std;
int main() {
    // Integer type
    int age = 25;
    cout << "Age: " << age << endl;
    // Float type
    float height = 5.9;
    cout << "Height: " << height << " feet" <<endl;
    // Double type
    double pi = 3.141592653589793;
    cout << "Value of Pi: " << pi << endl;
    // Char type
    char initial = 'A';
```

```
    cout << "Initial: " << initial <<endl;
    // Boolean type
    bool isStudent = true;
    cout << "Is a student: " << (isStudent ? "Yes" : "No") <<endl;
    return 0;
}
```

O/P:

```
Age: 25
Height: 5.9 feet
Value of Pi: 3.14159
Initial: A
Is a student: Yes
```

IMPLICIT CONVERSION

i)

```
#include <iostream>
Using namespace std;
int main() {
    // Declare variables of different types
    int intNum = 5;              // Integer
    double doubleNum = 4.5;      // Double
    char charValue = 'A';        // Character
    // Implicit conversion from int to double
    double result1 = intNum + doubleNum; // int is converted to double
    cout << "Result of int + double: " << result1 << endl;
    // Implicit conversion from char to int
    int asciiValue = charValue;  // char is converted to int (ASCII value)
    cout << "ASCII value of '" << charValue << "': " << asciiValue << endl;
    // Implicit conversion during arithmetic operations
    double result2 = charValue + doubleNum; // char is converted to double
    cout << "Result of char + double: " << result2 << endl;
    return 0;
}
```

O/P:

```
Result of int + double: 9.5
ASCII value of 'A': 65
Result of char + double: 69.5
```

ii)

```
#include <iostream>
int main() {
    int a=-12;
    unsigned int b=a;
    std::cout<<b;
    return 0;
}
```

O/P:

```
4294967284
```

EXPLICIT CONVERSION

```
#include <iostream>
using namespace std;
int main() {
    double num = 9.99;
    int intNum = (int)num; // Truncation occurs
    cout << "Original double: " << num << endl;
    cout << "Converted int: " << intNum << endl;
    int wholeNumber = 5;
    double convertedDouble = (double)wholeNumber;
    cout << "Original int: " << wholeNumber << endl;
    cout << "Converted double: " << convertedDouble << endl;
    float convertedFloat = (float)num;
    cout << "Converted float: " << convertedFloat << endl;
    return 0;
}
```

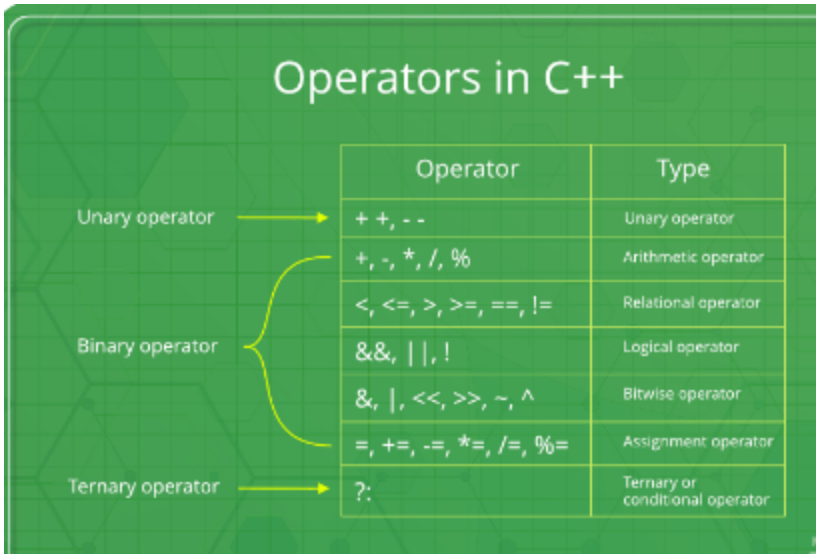
O/P:

Original double: 9.99
Converted int: 9
Original int: 5
Converted double: 5
Converted float: 9.99

2. Operators

Operators are symbols used to perform operations on variables and values. They can be categorized as follows:

- Arithmetic Operators: Used for basic mathematical operations.
    - + (addition), - (subtraction), * (multiplication), / (division), % (modulus)
- Unary Operators: Operate on a single operand.
    - ++ (increment), -- (decrement), - (negation)
- Binary Operators: Operate on two operands.
    - E.g., a + b, x < y
- Assignment Operators: Used to assign values to variables.
    - = (simple assignment), +=, -=, *=, /=, etc.
- Comparison Operators: Used to compare two values.
    - == (equal to), != (not equal), < (less than), > (greater than), <=, >=



EXAMPLE:

Arithmetic Operator

```
#include <iostream>
using namespace std;
int main()
{
   int a = 10, b = 7;
   cout <<(a + b) << endl;
   cout <<(a - b) << endl;
   cout<< (a * b) << endl;
   cout<< (a / b) << endl;
   cout <<(a % b) << endl;
   return 0;
}
```

O/P:

```
17
3
70
1
3
```

Relational Operator

```
#include <iostream>
using namespace std;
int main()
{
   int a = 6, b = 4;
   cout << "a == b is " << (a == b) << endl;
   cout << "a > b is " << (a > b) << endl;
   cout << "a >= b is " << (a >= b) << endl;
   cout << "a < b is " << (a < b) << endl;
   cout << "a <= b is " << (a <= b) << endl;
   cout << "a != b is " << (a != b) << endl;
   return 0;
}
```

O/P:

```
a == b is 0
a > b is 1
a >= b is 1
a < b is 0
a <= b is 0
a != b is 1
```

Logical Operator

```
#include <iostream>
using namespace std;
int main()
{
   int a = 2, b = 9;
   cout << "a && b is " << (a && b) << endl;
   cout << "a || b is " << (a || b) << endl;
   cout << "!b is " << (!b) << endl;
   return 0;
```

```
}
```

O/P:

```
a && b is 1
a || b is 1
!b is 0
```

Bitwise Operators

```
#include <iostream>
using namespace std;
int main()
{
   int a = 6, b = 4;
   cout << "a & b is " << (a & b) << endl;
   cout << "a | b is " << (a | b) << endl;
   cout << "a ^ b is " << (a ^ b) << endl;
   cout << "a<<1 is " << (a << 1) << endl;
   cout << "a>>1 is " << (a >> 1) << endl;
   cout << "~(a) is " << ~(a) << endl;
   return 0;
}
```

O/P:

```
a & b is 4
a | b is 6
a ^ b is 2
a<<1 is 12
a>>1 is 3
~(a) is -7
```

Assignment Operators

```
#include <iostream>
```

```
using namespace std;
int main()
{
    int a = 6, b = 4;
    cout << "a = " << a << endl;
    cout << "a += b is " << (a += b) << endl;
    cout << "a -= b is " << (a -= b) << endl;
    cout << "a *= b is " << (a *= b) << endl;
    cout << "a /= b is " << (a /= b) << endl;
    return 0;
}
```

O/P:

```
a = 6
a += b is 10
a -= b is 6
a *= b is 24
a /= b is 6
```

Ternary or Conditional Operators(?:)

```
#include <iostream>
using namespace std;
int main()
{
    int a = 3, b = 4;
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result << endl;
    return 0;
}
```

O/P:

```
The greatest number is 4
```

3. Conditional Statements

Conditional statements allow you to execute code based on certain conditions. Common forms include:

if statement:

```
if (condition) {
    // Code to execute if condition is true
}
```

else statement:

```
if (condition) {
    // Code if condition is true
} else {
    // Code if condition is false
}
```

EXAMPLE:

```
#include <iostream>
using namespace std;
int main() {
    int number = 10;
    if (number > 0) {
        cout << "The number is positive." << endl;
    } else if (number < 0) {
        cout << "The number is negative." << endl;
    } else {
        cout << "The number is zero." << endl;
    }
    return 0;
}
```

O/P:

The number is positive.

switch statement: A multi-way branch based on the value of a variable.

```
switch (variable) {
    case value1:
        // Code for case value1
        break;
    case value2:
        // Code for case value2
        break;
    default:
        // Code if no cases match
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    char grade = 'B';
    switch (grade) {
        case 'A':
            cout << "Excellent!" << endl;
            break;
        case 'B':
        case 'C':
            cout << "Well done" << endl;
            break;
        case 'D':
            cout << "You passed" << endl;
            break;
        case 'F':
            cout << "Better try again" << endl;
            break;
        default:
            cout << "Invalid grade" << endl;
    }

    return 0;
}
```

O/P:

Well done

Ternary Operator: A shorthand for simple if-else statements.

result = (condition) ? value_if_true : value_if_false;

4. Loops

Loops allow you to execute a block of code repeatedly based on a condition. Common types of loops include:

for loop: Used for a known number of iterations.

```
for (initialization; condition; increment) {
    // Code to execute
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; ++i) {
        cout << "Iteration: " << i << endl;
    }
    return 0;
}
```

O/P:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

while loop: Continues as long as the condition is true.

```
while (condition) {
    // Code to execute
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    while (i <= 5) {
        cout << "Iteration: " << i << endl;
        i++;
    }
    return 0;
}
```

O/P:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

do-while loop: Executes at least once before checking the condition.

```
do {
    // Code to execute
        } while (condition);
```

Example:

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do {
        cout << "Iteration: " << i << endl;
        i++;
    } while (i <= 5);
    return 0;
}
```

O/P:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

5. Functions

Functions are reusable blocks of code that perform a specific task. They promote modular programming by allowing you to break down complex problems into smaller, manageable pieces.

Function Definition:

```
return_type function_name(parameter_list) {
    // Code to execute
    return value; // Optional return statement
}
```

Function Call:

```
function_name(arguments); // Calling a function with arguments
```

- **Scope**: The region of code where a variable is accessible. Local variables are accessible only within the function where they are defined, while global variables can be accessed throughout the program.
- **Recursion**: A function that calls itself. This can be useful for problems like calculating factorials or traversing data structures.

  Example:

```
#include <iostream>
using namespace std;
// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1; // Base case: factorial of 0 or 1 is 1
    } else {
        return n * factorial(n - 1); // Recursive case
    }
}
int main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;
    if (number < 0) {
        cout << "Factorial is not defined for negative numbers." << endl;
    } else {
          cout << "Factorial of " << number << " is " << factorial(number) <<
endl;
    }
    return 0;
}
```

  O/P:

```
Enter a positive integer: 5
Factorial of 5 is 120
```

ARRAYS

- In C++, an array is a collection of elements of the same type, stored in contiguous memory locations.
- It allows you to manage multiple values under a single variable name, accessed via an index.
- To **declare an array,** define the variable type, specify the name of the array followed by square brackets and specify the number of elements it should store:
- **string cars[4];**
- We have now declared a variable that holds an array of four strings. To **insert values** to it, we can use an **array literal** - place the values in a comma-separated list, inside curly braces:
- **string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};**
- To create an array of three integers, you could write:
- **int myNum[3] = {10, 20, 30};**

Access the Elements of an Array

- You access an array element by referring to the index number inside square brackets [].
- This statement accesses the value of the first element in cars:

```
#include <iostream>
//#include <string>
using namespace std;
int main() {
  string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
  cout << cars[0];
  return 0;
}
```

O/P:

Volvo

```
#include<iostream>
using namespace std;
int main(){
int a[4]={1,2,3,4};
cout<<a[3];
}
```

O/P:

4

CHANGE THE ARRAY ELEMENT

We can change the array element by using index

```
#include<iostream>
using namespace std;
int main(){
int a[4]={1,2,3,4};
a[3]=1;
cout<<a[3];
}
```

O/P:

1

LOOP THROUGH ARRAY

We can access all the elements present in an array by using for loop

```
#include<iostream>
using namespace std;
int main(){
    string a[4]={"madhu","balu","naga","sai"};
    for(int i=0;i<4;i++){
        cout<<a[i]<<endl;
```

```
    }
}
```

O/P:

```
madhu
balu
naga
sai
```

```
#include<iostream>
using namespace std;
int main(){
    string a[4]={"madhu","balu","naga","sai"};
    for(int i=0;i<4;i++){
        cout<<i<<"="<<a[i]<<endl;
    }
}
```

O/P:

```
0=madhu
1=balu
2=naga
3=sai
```

```
#include<iostream>
using namespace std;
int main(){
    int size;
    cout<<"Enter size of an array"<<endl;
    cin>>size;
    int arr[size];
    cout<<"Enter elements for an array"<<endl;
    for(int i=0;i<size;i++){
        cin>>arr[i];
    }
```

```
    cout<<":Acceessing elements of an array"<<endl;
    for(int i=0;i<size;i++){
        cout<<arr[i]<<endl;
    }
}
```

O/P:

```
Enter size of an array
5
Enter elements for an array
432
5
12
463
54
:Acceessing elements of an array
432
5
12
463
54
```

Finding size of an array:
i)for static array

```
#include <iostream>
using namespace std;
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    cout << "The size of the array is: " << size <<endl;
    return 0;
}
```

O/P:

```
The size of the array is: 5
```

We can access elements by using “for each”

There is also a "for-each loop" (introduced in C++ version 11 (2011)), which is used exclusively to loop through elements in an array (and other data structures, like vectors and lists):

```
#include <iostream>
using namespace std;
int main() {
    string s[]={"madhu","sai","gopal","dev"};
    for(string s1:s){
        cout<<s1<<"\n";
    }
    return 0;
}
```

O/P:

```
madhu
sai
gopal
dev
```

Omit Array Size

In C++, you don't have to specify the size of the array. The compiler is smart enough to determine the size of the array based on the number of inserted values:

It is also possible to declare an array without specifying the elements on declaration, and add them later:

```
#include <iostream>
//#include <string>
using namespace std;
int main() {
  string cars[5];
  cars[0] = "Volvo";
  cars[1] = "BMW";
```

```
  cars[2] = "Ford";
  cars[3] = "Mazda";
  cars[4] = "Tesla";
  for(int i = 0; i < 5; i++) {
    cout << cars[i] << "\n";
  }
  return 0;
}
```

O/P:

```
Volvo
BMW
Ford
Mazda
Tesla
```

Note: The example above only works when you have specified the size of the array.
If you don't specify the array size, an error occurs:

```
#include <iostream>
using namespace std;
int main() {
  string cars[];
  cars[0] = "Volvo";
  cars[1] = "BMW";
  cars[2] = "Ford";
  cars[3] = "Mazda";
  cars[4] = "Tesla";
  for(int i = 0; i < 5; i++) {
    cout << cars[i] << "\n";
  }
  return 0;
}
```

O/P:

```
ERROR!
/tmp/hWzmE1A4T5.cpp: In function 'int main()':
/tmp/hWzmE1A4T5.cpp:4:10: error: array size missing in 'cars'
    4 |   string cars[];
      |          ^~~~
```

FIXED SIZE ARRAY

Once we declare size of an array initially then we can add or change the size of an array,Because it was fixed .If you try to change then it will print error

```
#include <iostream>
using namespace std;
int main() {
  string cars[3] = {"Volvo", "BMW", "Ford"};
  cars[3] = "Mazda";
  for (string car : cars) {
    cout << car << "\n";
  }
  return 0;
}
```

O/P:

Segmentation fault

DYNAMIC SIZE ARRAY

We can add or resize the array by using new keyword and vector

i)Using new keyword

```
#include <iostream>
using namespace std;
int main() {
    int size;
    cout << "Enter size of the array: ";
```

```
    cin >> size;
    // Dynamic memory allocation
    int* arr = new int[size];
    cout << "Enter elements for the array:" << endl;
    for (int i = 0; i < size; i++) {
        cin >> arr[i];
    }
    cout << "Accessing elements of the array:" << endl;
    for (int i = 0; i < size; i++) {
        cout << arr[i] << endl;
    }
    // Deallocate the memory
    delete[] arr;
    return 0;
}
```

O/P:

```
Enter size of the array: 3
Enter elements for the array:
1
2
3
Accessing elements of the array:
1
2
3
```

ii)Using Vector

```
#include <iostream>
#include <vector> // Including the vector library
using namespace std;
int main() {
  vector<string> cars = {"Volvo", "BMW", "Ford"};
  cars.push_back("Tesla");
  for (string car : cars) {
    cout << car << "\n";
  }
```

```
  return 0;
}
```

O/P:

```
Volvo
BMW
Ford
Tesla
```

Real-time example:

i)

```
//Average of different ages
#include<iostream>
using namespace std;
int main(){
    int sum=0;
    int a[]={10,40,20,30,50};
    for(int b:a){
        sum=sum+b;
    }
    int average=sum/(sizeof(a)/sizeof(a[0]));
    cout<<average;
}
```

O/P:

```
30
```

ii)

```
// finds the lowest age among different ages:
#include<iostream>
using namespace std;
int main(){
    int ages[]={40,30,50,20,38,4334};
    int low=ages[0];
```

```
    for(int i=1;i<(sizeof(ages)/sizeof(ages[0]));i++){
        if(low>ages[i]){
            low=ages[i];
        }
    }
    cout<<low;
}
```

O/P:

20

11 TH VERSION ARRAY DECLARATION(STD::ARRAY)

```
#include<iostream>
#include<array>
using namespace std;
int main(){
    array<int,5> arr;
    arr={1,2,3};
    for(int a:arr){
        cout<<a<<endl;
    }
}
```

O/P:

1
2
3
0
0

```
#include<iostream>
#include<array>
using namespace std;
int main(){
```

```
    array<string,5> arr;
    arr[0]="madhu";
    arr[4]="anil";
    for(string a:arr){
        cout<<a<<endl;
    }
}
```

O/P:

madhu

Anil

Summary

1.C-style arrays are straightforward and simple to use.
2.`std::array` provides type safety, size information, and member functions, making it a preferred choice for many scenarios in C++11 and beyond.

2D-ARRAY

A 2D array is essentially an array of arrays. It can be visualized like a table with rows and columns. In C++, 2D arrays are used to store data in a matrix-like structure.

Declaration of a 2D Array

You declare a 2D array in C++ by specifying the number of rows and columns.

Syntax:

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

Example Declaration:

```
int matrix[3][4]; // A 2D array with 3 rows and 4 columns
```

This creates a matrix with 3 rows and 4 columns, where each element is of type int.

Initializing a 2D Array

You can initialize a 2D array at the time of declaration by specifying values for each element.

Example:

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

In this example:

The first row is {1, 2, 3}
The second row is {4, 5, 6}
The third row is {7, 8, 9}

Accessing Elements in a 2D Array

You can access elements of a 2D array using row and column indices.

Syntax:

```
array_name[row_index][column_index];
```

Example:

```
int x = matrix[1][2]; // Accesses the element in the second row, third column (which is 6)
```

Use Cases of 2D Arrays

- **Matrices:** 2D arrays are commonly used for matrix operations in mathematics and computer graphics.
- **Grids:** For games or simulations that require grid-based layouts (e.g., Tic-Tac-Toe, Minesweeper).

Key Points:

- Indexing starts from 0: Both row and column indices start at 0.
- Size limits: The size of a 2D array must be known at compile time unless using dynamic memory (e.g., with pointers).
- Nested loops: Typically, you use nested loops to iterate through a 2D array.

Example:

```
#include <iostream>
using namespace std;
int main() {
    int marks[3][4];
    // Taking input from the user
    cout << "Enter the marks for 3 students in 4 subjects: " << endl;
    for (int i = 0; i < 3; ++i) {
        cout << "Enter marks for student " << i + 1 << ":" << endl;
        for (int j = 0; j < 4; ++j) {
            cout << "Subject " << j + 1 << ": ";
            cin >> marks[i][j];
        }
        cout << endl;
    }
    cout << "Student Marks (Rows: Students, Columns: Subjects):\n";
    for (int i = 0; i < 3; ++i) {
        cout << "Student " << i + 1 << ": ";
        for (int j = 0; j < 4; ++j) {
            cout << marks[i][j] << " ";  // Print each mark
        }
        cout << endl;  // New line after each student
    }
    return 0;
}
```

O/P:

```
Enter the marks for 3 students in 4 subjects:
Enter marks for student 1:
```

```
Subject 1: 32
Subject 2: 132
Subject 3: 43
Subject 4: 423

Enter marks for student 2:
Subject 1: 214
Subject 2: 32
Subject 3: 1
Subject 4: 324

Enter marks for student 3:
Subject 1: 43
Subject 2: 14
Subject 3: 341
Subject 4: 41

Student Marks (Rows: Students, Columns: Subjects):
Student 1: 32 132 43 423
Student 2: 214 32 1 324
Student 3: 43 14 341 41
```

Dynamic 2D Arrays

If the size of the array is unknown at compile time, dynamic memory allocation using pointers is required. Here's a basic example using dynamic allocation:

```
#include <iostream>
using namespace std;
int main() {
    int students, subjects;
    cout << "Enter the number of students: ";
    cin >> students;
    cout << "Enter the number of subjects: ";
    cin >> subjects;
    // Dynamically allocate memory for a 2D array
```

```
    int** marks = new int*[students];  // Array of pointers to store the rows
    for (int i = 0; i < students; ++i) {
        marks[i] = new int[subjects];
    }
    cout << "\nEnter the marks for each student:\n";
    for (int i = 0; i < students; ++i) {
        cout << "Enter marks for student " << i + 1 << ":\n";
        for (int j = 0; j < subjects; ++j) {
            cout << "Subject " << j + 1 << ": ";
            cin >> marks[i][j];  // Input for each subject
        }
        cout << endl;
    }
    cout << "\nStudent Marks (Rows: Students, Columns: Subjects):\n";
    for (int i = 0; i < students; ++i) {
        cout << "Student " << i + 1 << ": ";
        for (int j = 0; j < subjects; ++j) {
            cout << marks[i][j] << " ";  // Print each mark
        }
        cout << endl;
    }
    // Free dynamically allocated memory
    for (int i = 0; i < students; ++i) {
        delete[] marks[i];  // Delete each row (subject array)
    }
    delete[] marks;  // Delete the array of pointers (rows)
    return 0;
}
```

O/P:

```
Enter the number of students: 2
Enter the number of subjects: 2

Enter the marks for each student:
Enter marks for student 1:
Subject 1: 23
```

```
Subject 2: 12

Enter marks for student 2:
Subject 1: 23
Subject 2: 453


Student Marks (Rows: Students, Columns: Subjects):
Student 1: 23 12
Student 2: 23 453
```

STRINGS

In C++, a string is a collection of characters. There are two primary ways to handle strings in C++:

1. C-style strings (char[] or char*)
2. C++ std::string class from the Standard Library.

Each has different advantages, and the modern approach is to use std::string for better safety, flexibility, and ease of use.

1. C-Style Strings (char[] or char*)

A **C-style string** is an array of characters terminated by a **null character** (\0). The null character is essential because it marks the end of the string.

Example:

```
#include <iostream>
#include <cstring> // For string functions like strlen, strcpy, strcat, etc.
using namespace std;
int main() {
   // Declaring a C-style string
   char str1[20] = "Hello";
   char str2[20] = "Madhu";
   strcat(str1, " ");
   strcat(str1, str2); // str1 becomes "Hello World"
```

```
    cout << "Concatenated string: " << str1 << endl;
    cout << "Length of str1: " << strlen(str1) << endl;
    return 0;
}
```

O/P:

```
Concatenated string: Hello Madhu
Length of str1: 11
```

Common C-Style String Functions:

strlen(): Returns the length of the string (excluding the null character).
strcpy(): Copies one string to another.
strcat(): Concatenates two strings.
strcmp(): Compares two strings lexicographically.

2. C++ std::string Class

C++ provides a **std::string** class in the Standard Library, which offers much more functionality, ease of use, and automatic memory management compared to C-style strings.

Common std::string Functions:

- **length()** or **size()**: Returns the length of the string.
- **append()**: Adds a string to the end of another string.
- **substr()**: Extracts a substring from the string.
- **find()**: Finds a substring within the string and returns its position.
- **replace()**: Replaces part of the string with another string.
- **compare()**: Compares two strings lexicographically.

Example:

```
#include<iostream>
#include<string>
```

```
using namespace std;
int main(){
    string s="abc";
    string m="xyz";
    cout<<s.length()<<endl;
    cout<<(s==m)<<endl;
    s=s+"madhu";
    cout<<s<<endl;
    cout<<s.substr(3)<<endl;
    cout<<s.find("bc")<<endl;
    cout<<s<<endl;
    s.replace(2,4,"ram");
    cout<<s<<endl;
    s.insert(0,"mad");
    cout<<s<<endl;
    s.assign("satya");
    cout<<s<<endl;
    s.push_back('f');
    cout<<s<<endl;
    s.pop_back();
    cout<<s<<endl;
    cout<<s.front()<<endl;
    cout<<s.back()<<endl;
    cout<<s.at(2)<<endl;
    cout<<s.find('y')<<endl;
    s.swap(m);
    cout<<s<<endl;
    cout<<m<<endl;
    m.append("ramana");
    cout<<m<<endl;
    m.erase(5,2);
    cout<<m<<endl;
    cout<<s<<endl;
    cout<<m<<endl;
    cout<<m.insert(5,"ra")<<endl;
    cout<<m.find('a')<<endl;
    cout<<m.rfind('a')<<endl;

}
```

O/P:

```
3
0
abcmadhu
madhu
1
abcmadhu
abramhu
madabramhu
satya
satyaf
satya
s
a
t
3
xyz
satya
satyaramana
satyamana
xyz
satyamana
satyaramana
1
10
```

String Iteration Example:

```
#include<iostream>
#include<string>
using namespace std;
int main(){
    string s="satya";
    for(char m:s){
        cout<<m<<endl;
    }
}
```

O/P:

s
a
t
y
a

NOTE:

This is achieved by using for each loop we can achieve by using for loop too

Difference Between begin And cbegin

1. begin():
Definition: The begin() function returns an iterator pointing to the first element of the container.
Modifiability: The iterator returned by begin() allows modification of the elements in the container. This means you can change the values of the elements the iterator points to.

2. cbegin():
Definition: The cbegin() function returns a constant iterator (const_iterator) that points to the first element of the container.
Modifiability: The iterator returned by cbegin() does not allow modification of the elements. It is read-only, which helps ensure that the data cannot be changed during traversal.

Example:

```
#include<iostream>
#include<string>
using namespace std;
int main(){
   string s="satyaramanamadhu";
   //for(char m:s){
   //    cout<<m<<endl;
   //}
```

```
    for(auto i=s.begin();i!=s.end();++i){
        cout<<*i;
        *i='d';
    }
    cout<<endl;
    for(auto i=s.cbegin();i!=s.cend();++i){
            cout<<*i;
            //*i='d';//line 1
       }
}
```

O/P:

```
satyaramanamadhu
dddddddddddddddd
```

NOTE:

If we uncomment the line 1 then we will get the error.Because we can't modify when we are looping throw by using cbegin or cend.Because it is constant across the container.

[WEEK 01]: DAY 3

POINTERS

- A pointer is a variable that stores the memory address of another variable.
- This allows for powerful programming techniques such as dynamic memory management and the creation of complex data structures (like linked lists, trees, etc.).
- Pointers are essential for dynamic memory allocation, enabling the allocation and deallocation of memory during runtime.
- In **C++**, memory management requires explicit deallocation; developers must manually free memory using `delete` or `free` to prevent memory leaks.
- In contrast, **Java** uses automatic garbage collection (AGC), which means memory deallocation is handled by the JVM. This reduces the burden on developers, as they do not need to manually manage memory.

Key Concepts

1.Declaration:

To declare a pointer, you specify the type of data it points to followed by an asterisk(*).

EX:

```
int *ptr;  // Pointer to an integer
```

2.Initialization:

Pointers need to be initialized with the address of a variable, typically using the address-of operator (&).

EX:

```
int a = 10;
int *ptr = &a;  // ptr now holds the address of a
```

3.Dereferencing:

Dereferencing a pointer means accessing the value at the address stored in the pointer using the dereference operator(*).

EX:

```
#include<iostream>
using namespace std;
int main(){
   int a=10;//Initializing a variable with value 10
```

```
    int *ptr=&a;//Pointer ptr holds the address of variable a
    cout<<ptr<<endl;//Prints the address of a
    cout<<*ptr<<endl;//it will print value assigned to this particular address
    int c=*ptr;//Accessing the original value through dereferencing
    cout<<c<<endl;
    *ptr=34;//Changes the value of a using the pointer
    cout<<a;//Prints the new value of a (34)
    return 0;
}
```

4.Pointer Arithmetic:

Pointers can be incremented or decremented to point to different memory locations, typically in the context of arrays.

Ex:

```
int arr[] = {1, 2, 3};
int *ptr = arr;  // Points to the first element
ptr++;          // Now points to the second element (2)
```

Ex:

```
#include<iostream>
using namespace std;
int main(){
    int a[5]={1,3,2,5,6};
    int *ptr=a;
    for(int i=0;i<sizeof(a)/sizeof(a[0]);i++){
        //cout<<a[i]<<endl;
        cout<<*(ptr+i)<<endl;
    }
    return 0;
}
```

5.Null Pointers:

A pointer can be set to NULL to indicate that it does not point to any valid memory location.

EX:

```
int *ptr = NULL;  // ptr does not point to any valid address
```

Ex:

```
#include<iostream>
```

```
using namespace std;
int main(){
    int *ptr=NULL;
    cout<<ptr;
    //cout<<*ptr;
    return 0;
}
```

6.Dynamic Memory Allocation:

Dynamic memory allocation in C++ allows you to allocate memory at runtime using the `new` keyword. This is useful when the size of the data structure is not known at compile time. Here's a comprehensive overview of dynamic memory allocation, including examples:

Key Concepts:

1.Using new:

The `new` operator is used to allocate memory for a single variable or an array of variables.

2.Using delete:

The `delete` operator is used to free memory that was previously allocated with `new`.

3.Memory Leak:

If you allocate memory using `new` but forget to deallocate it using `delete`, it leads to memory leaks, which can exhaust system memory over time.

Example

1.Allocating a Single Variable

```
#include <iostream>
using namespace std;
int main() {
    int *ptr = new int; // Allocate memory for a single integer
    *ptr = 42;          // Assign value to the allocated memory
    cout << "Value: " << *ptr << endl; // Output: Value: 42
    delete ptr; // Free the allocated memory
    return 0;
}
```

2.Allocating an Array

```
#include <iostream>
```

```
using namespace std;
int main() {
    int size;
    cout << "Enter size of array: ";
    cin >> size;
    int *arr = new int[size]; // Allocate memory for an array of integers
    // Initialize the array
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1; // Assign values
    }
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " "; // Output: 1 2 3 ... up to size
    }
    cout << endl;
    delete[] arr; // Free the allocated memory for the array
    return 0;
}
```

Operations on pointers

1. Incrementing Pointers

```
#include <iostream>
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* ptr = arr; // Pointer to the first element of the array
    std::cout << "Current value: " << *ptr << std::endl; // Output: 10
    // Incrementing the pointer
    ptr++;
     std::cout << "After incrementing, current value: " << *ptr << std::endl; // Output:
20
    return 0;
}
```

O/P:

```
Current value: 10
After incrementing, current value: 20
```

2. Decrementing Pointers

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* ptr = arr + 4; // Pointing to the last element (50)
    cout << "Current value: " << *ptr << std::endl; // Output: 50
    // Decrementing the pointer
    ptr--;
      cout << "After decrementing, current value: " << *ptr << std::endl; //
Output: 40
    return 0;
}
```

O/P:

```
Current value: 50
After decrementing, current value: 40
```

3. Comparing Pointers

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* ptr1 = arr;
    int* ptr2 = arr + 2;
    if (ptr1 < ptr2) {
        cout << "ptr1 points to an earlier element than ptr2" << endl;
    }
    if (ptr1 != ptr2) {
        cout << "ptr1 and ptr2 do not point to the same element" << endl;
    }
    ptr1 += 2;
    if (ptr1 == ptr2) {
        cout << "ptr1 and ptr2 now point to the same element" << endl;
```

```
    }
    return 0;
}
```

O/P:

```
ptr1 points to an earlier element than ptr2
ptr1 and ptr2 do not point to the same element
ptr1 and ptr2 now point to the same element
```

Example:

```
#include<iostream>
using namespace std;
int main(){
    int a=10;
    int *p=&a;
    cout<<p<<endl;
    p++;
    cout<<p<<endl;
    cout<<*p<<endl;
    cout<<a<<endl;
    int *r=&a;
    cout<<r<<endl;
    p--;
    cout<<*p;
}
```

O/P:

```
0x7ffc6ee3e48c
0x7ffc6ee3e490
1860429200
10
0x7ffc6ee3e48c
10
```

Dynamic Memory Allocation

1. malloc()

- **Purpose:** Allocates a specified number of bytes of memory and returns a pointer to the first byte.
- **Initialization:** The allocated memory is not initialized, meaning it contains garbage values.

Syntax:void* malloc(size_t size);

Example :

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    // Allocate initial memory for 5 integers
    int* arr = (int*)malloc(5 * sizeof(int));
    if (arr == nullptr) {
        cerr << "Memory allocation failed." << endl;
        return 1;
    }
    for (int i = 0; i < 5; ++i) {
        arr[i] = i + 1; // Assign values 1 to 5
    }
    cout << "Initial array: ";
    for (int i = 0; i < 5; ++i) {
        cout << arr[i] << " ";
    }
}
```

O/P:

Initial array: 1 2 3 4 5

2. calloc()

- **Purpose:** Allocates memory for an array of elements and initializes all bytes to zero.

- **Initialization:** All allocated memory is initialized to zero.

Syntax:void* calloc(size_t num, size_t size);

Example:

```
#include <iostream>
#include <cstdlib> // Include for calloc and free
using namespace std;
int main() {
    int numElements = 5;
    int* arr = (int*)calloc(numElements, sizeof(int));
    // Check if memory allocation was successful
    if (arr == nullptr) {
        cerr << "Memory allocation failed." << endl;
        return 1;
    }
    cout << "Array values after calloc (initialized to zero): ";
    for (int i = 0; i < numElements; ++i) {
        cout << arr[i] << " "; // Should print 0 for all elements
    }
    cout << endl;
    for (int i = 0; i < numElements; ++i) {
        arr[i] = (i + 1) * 10; // Assign values 10, 20, 30, ...
    }
    cout << "Array values after assignment: ";
    for (int i = 0; i < numElements; ++i) {
        cout << arr[i] << " "; // Should print 10, 20, 30, 40, 50
    }
    cout << endl;
    // Free the allocated memory
    free(arr);
    return 0;
}
```

O/P:

Array values after calloc (initialized to zero): 0 0 0 0 0
Array values after assignment: 10 20 30 40 50

3. realloc()

- **Purpose:** Changes the size of previously allocated memory. It can increase or decrease the size of the allocation.
- **Note:** If the new size is larger and there is not enough space, it may allocate a new block of memory, copy the contents from the old block, and free the old block.

Syntax:void* realloc(void* ptr, size_t new_size);

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    // Allocate initial memory for 5 integers
    int* arr = (int*)malloc(5 * sizeof(int));
    if (arr == nullptr) {
        cerr << "Memory allocation failed." << endl;
        return 1;
    }
    // Initialize the array
    for (int i = 0; i < 5; ++i) {
        arr[i] = i + 1; // Assign values 1 to 5
    }
    cout << "Initial array: ";
    for (int i = 0; i < 5; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
     // Resize the array to hold 10 integers
    int* newArr = (int*)realloc(arr, 10 * sizeof(int));
    if (newArr == nullptr) {
        cerr << "Memory reallocation failed." << endl;
```

```
        free(arr); // Free the original memory
        return 1;
    }
    arr = newArr; // Update the pointer to the new memory block
    // Initialize the new elements
    for (int i = 5; i < 10; ++i) {
        arr[i] = i + 1; // Assign values 6 to 10
    }
    // Print new values
    cout << "Resized array: ";
    for (int i = 0; i < 10; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    // Free the allocated memory
    free(arr);
    return 0;
}
```

O/P:

Initial array: 1 2 3 4 5
Resized array: 1 2 3 4 5 6 7 8 9 10

4. free()

- **Purpose:** Deallocates memory that was previously allocated by malloc(), calloc(), or realloc().
- **Usage:** It is crucial to free dynamically allocated memory to prevent memory leaks.

Example:

```
#include <iostream>
#include <cstdlib> // for free
int main() {
    int *arr = (int *)malloc(5 * sizeof(int));
    if (arr == NULL) {
```

```
        std::cerr << "Memory allocation failed!" << std::endl;
        return 1;
    }
    free(arr); // Deallocate the memory
    return 0;
}
```

FUNCTION POINTERS

Function pointers in C++ are a powerful feature that allows you to store the address of a function in a pointer variable. This enables you to pass functions as arguments to other functions, return functions from other functions, and create arrays of function pointers. Here's a detailed explanation with examples.

Declaring and Using Function Pointers

1. Declaring Function Pointers: To declare a function pointer, you need to specify the return type of the function and the parameter types.

Syntax:return_type (*pointer_name)(parameter_types);

Example:

```
#include <iostream>
using namespace std;
void myFunction(int x) {
    cout << "Value: " << x << endl;
}
int main() {
    // Declare a function pointer
    void (*funcPtr)(int);
    // Assign the address of myFunction to funcPtr
    funcPtr = &myFunction;
    // Use the function pointer to call the function
    funcPtr(5); // Output: Value: 5
    return 0;
}
```

O/P:

```
Value: 5
```

2. Passing Functions as Arguments: You can pass function pointers as arguments to other functions. This is useful for callbacks and for implementing various algorithms.

Example:

```
#include <iostream>
using namespace std;
void printSquare(int x) {
    cout << "Square: " << (x * x) << endl;
}
// Function that takes a function pointer as an argument
void executeFunction(void (*funcPtr)(int), int value) {
    funcPtr(value); // Call the function using the pointer
}
int main() {
    // Pass printSquare as an argument
    executeFunction(printSquare, 4); // Output: Square: 16
    return 0;
}
```

O/P:

```
Square: 16
```

3. Returning Function Pointers: You can also have functions that return function pointers. This can be useful for creating a factory of functions or selecting behavior dynamically.

Example:

```
#include <iostream>
// Function prototypes
int add(int a, int b);
int subtract(int a, int b);
// Function that returns a pointer to a function
int (*getOperation(const std::string& operation))(int, int) {
    if (operation == "add") {
        return add; // Return pointer to the add function
    } else if (operation == "subtract") {
```

```
        return subtract; // Return pointer to the subtract function
    }
    return nullptr; // Return nullptr for an invalid operation
}
// Implementation of add function
int add(int a, int b) {
    return a + b;
}
// Implementation of subtract function
int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Get a function pointer based on user input
    std::string operation;
    std::cout << "Enter operation (add or subtract): ";
    std::cin >> operation;
    // Get the appropriate function pointer
    int (*operationFunc)(int, int) = getOperation(operation);
    if (operationFunc) {
        int a, b;
        std::cout << "Enter two numbers: ";
        std::cin >> a >> b;
        // Call the function using the pointer
        int result = operationFunc(a, b);
        std::cout << "Result: " << result << std::endl;
    } else {
        std::cout << "Invalid operation!" << std::endl;
    }
    return 0;
}
```

O/P:

```
Enter operation (add or subtract): add
Enter two numbers: 7
9
Result: 16
```

4. Arrays of Function Pointers: You can create arrays of function pointers to manage multiple functions easily.

Example:

```
#include <iostream>
// Function prototypes
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
// Define an array of function pointers
int (*operationArray[])(int, int) = { add, subtract, multiply, divide };
// Implementation of add function
int add(int a, int b) {
    return a + b;
}
// Implementation of subtract function
int subtract(int a, int b) {
    return a - b;
}
// Implementation of multiply function
int multiply(int a, int b) {
    return a * b;
}
// Implementation of divide function
int divide(int a, int b) {
    if (b == 0) {
        std::cerr << "Error: Division by zero!" << std::endl;
        return 0; // Return 0 or handle error as needed
    }
    return a / b;
}
int main() {
    int a, b, operation;
    // Get user input for numbers
    std::cout << "Enter two numbers: ";
```

```
    std::cin >> a >> b;
    // Display options for operations
    std::cout << "Select operation:\n";
    std::cout << "0: Add\n";
    std::cout << "1: Subtract\n";
    std::cout << "2: Multiply\n";
    std::cout << "3: Divide\n";
    std::cout << "Enter your choice (0-3): ";
    std::cin >> operation;
    // Call the appropriate function using the array of function pointers
        if (operation >= 0 && operation < sizeof(operationArray) / sizeof(operationArray[0])) {
            int result = operationArray[operation](a, b); // Call the selected function
        std::cout << "Result: " << result << std::endl;
    } else {
        std::cout << "Invalid operation!" << std::endl;
    }
    return 0;
}
```

O/P:

```
Enter two numbers: 32
5
Select operation:
0: Add
1: Subtract
2: Multiply
3: Divide
Enter your choice (0-3): 2
Result: 160
```

BITMASKING

What is a bit?

A bit is the smallest unit of data. We know that a computer system only understands binary language which consists of 0s and 1s. These 0s and 1s single-handedly are known as – 'BIT'.

BITMASKING IN C++

Bitmasking in C++ involves manipulating individual bits of a number to achieve the desired output. It is achieved by generating a bit mask and is used very often for the following operations:

Bit Toggle: If a bit is set to 0, it can be toggled to 1 and vice-versa.

Bit Setting: If a bit is set to 0 then it's called 'bit is NOT set'. We can set it by performing a toggle operation and change it to 1. This is known as bit setting.

Bit Clearing: If a bit is set to 1 then it's called a 'SET-BIT'. We can change it to 0 by performing a toggle operation this is called a – 'Bit-clearing' operation.

Checking specific bit is on or off: A bit is said to be on if it's 1 and off if it's 0. For example, an integer can contain multiple bits and we can check if, in that integer, a specific bit is set or not by utilizing bitwise operators.

NOTE:

- If the bit is set then the answer will be 2power(bit_position) For example, if the bit position is 3, then the answer will be 2power3 = 8.
- Else if the bit is 0 (not set) then the answer will be 0.

BITMASKING OPERATIONS IN C++

1. Setting a Specific Bit: Setting a specific bit basically means changing it from 0 to 1. It can be done by utilizing the Bitwise OR because of its property to give 1 if either of the bits is set to 1 and the bitwise left shift operator.

We will shift the LSB bit of 1 to the specified position that we want to set and then perform a bitwise OR Operation.

Syntax: integer | (1 << bit_position_to_be_set)

Here, the bit position to be set will be the place of the bit that we want to change to 1.

```
#include<iostream>
using namespace std;
int main(){
   int a=10;
   a=a|(1<<5);
```

```
    cout << "Result after setting the fifth bit: " << a;
    return 0;
}
```

O/P:

Result after setting the fifth bit: 42

2. Clearing a Bit: Clearing a bit means we set it to 0 if it is 1 without touching or affecting any other bits. This is done by using Bitwise AND and the negation operator (Bitwise NOT). The Bitwise NOT flips all the bits that are 1 to 0 and 0 to 1. This property of the bitwise NOT helps us in clearing a set bit.

Syntax: integer & ~(1 << bit_position_to_clear)

```
#include <iostream>
using namespace std;
int main()
{
        int x = 11;
        // clearing bit at third position
        x = x & ~(1 << 3);
        cout << "Result after clearing the 3rd bit: " << x;
        return 0;
}
```

O/P:

Result after clearing the 3rd bit: 3

3. Toggle a Bit: In this operation, we flip a bit. If it's set to 1 we make it 0 and if it's set to 0 then we flip it to 1. This is easily achievable by the Bitwise XOR operator (^) and the left shift (<<). We will utilize the property of the XOR operator to flip the bits if the bits of 2 different numbers are not the same.The same approach is used that is, by shifting 1 to a specific position which we want to flip.

Syntax: Integer ^ (1 << bit_position_to_toggle)

```
#include <iostream>
using namespace std;
int main()
{
	int x = 11;
	// toggling zeroth bit
	x = x ^ 1 << 0;
	cout << "Result after toggling the zeroth bit: " << x;
	return 0;
}
```

O/P:

Result after toggling the zeroth bit: 10

4. Check if a Bit is Set or not

In this operation, we check if a bit at a specific position is set or not. This is done by using the bitwise AND (&) and the Left shift operator. We basically left shift the set bit of 1 to the specified position for which we want to perform a check and then perform a bitwise AND Operation.If the bit is set then the answer will be – 2(bit_position) For example, if the bit position is 3, then the answer will be 23 = 8. Else if the bit is 0 (not set) then the answer will be 0.

Syntax: *Integer* & (1 << bit_position_to_check)

```
#include <iostream>
using namespace std;
int main()
{
	int x = 11;
	// the AND will return a non zero number if the bit is
	// set, otherwise it will return zero
	if (x & (1 << 3)) {
		cout << "Third bit is set\n";
	}
	else {
```

```
            cout << "Third bit is not set\n";
        }
        return 0;
}
```

O/P:

Third bit is set

[WEEK 01]: DAY 4

OOPS

OOPS is stands for object oriented programing structure/system
While object-oriented programming is about creating objects that contain both data and functions.

Why OOPS….?

The way of writing the code in a proper way we need some principles.They are
1.Encapsulation: Security or Modularity
2.**Abstraction:**Security
3.**Inheritance:**Reusability or Readability or Extensibility
4.**Polymorphism:**Flexibility or Readability

Every application need to follow these 4 principles/steps

Where are we using OOPS…?

Everywhere in our application we are using oops.If you take a small class that is an encapsulated class.
Every class is encapsulated with some functionality

CLASS

- A class is a blueprint of an object
- A class doesn't take any memory allocation
- By using single class we can create multiple objects
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Blueprint for Objects: A class defines the structure and behavior (attributes and methods) that the objects created from it will have.
Memory Allocation: While a class itself doesn't take up memory until an object is created from it, the objects instantiated from the class do consume memory.
Multiple Objects: You can create multiple instances (objects) from a single class, each with its own state and behavior.

User-Defined Data Type: Classes allow you to create custom data types that can encapsulate both data and functionality.

Create a Class

To create a class, use the **class** keyword:

Example:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
        string myString;  // Attribute (string variable)
};
```

Example Explained:

- The class keyword is used to create a class called MyClass.
- The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called attributes.
- At last, end the class definition with a semicolon ;.

OBJECT:

A Object is a instance of class
Object is a real time entity which we can see and we can feel
A Object is formed from class
We can create multiple objects from a single class and each object has its own values.
It can take the memory allocation
When we create an object it will take some space so it is called dynamic memory allocation.
Instance of a Class:An object is an instance of a class. When you create an object, you are instantiating the class, giving it a concrete presence in memory.
Real-Time Entities:Objects represent real-world entities or concepts that you can perceive and interact with in your program. For example, a Car object might represent a specific car with attributes like color and model.

Formation from Class:Objects are formed from a class blueprint. The class defines the structure (attributes) and behavior (methods) that the objects will have.

Multiple Objects:You can create multiple objects from a single class, and each object can have its own unique values for its attributes. For example, you can create multiple Car objects with different brands and years.

Memory Allocation:When an object is created, it occupies memory. This allocation typically happens on the stack or the heap, depending on how the object is created (e.g., automatic vs. dynamic allocation).

Dynamic Memory Allocation:If you use the new keyword to create an object, it is allocated on the heap, and you can dynamically allocate memory. This means that the memory can be managed more flexibly, allowing for the creation and destruction of objects at runtime.

Create an Object

1.Stack Allocation/Automatic (Stack) Allocation: Objects can be created directly on the stack.

```
#include<iostream>
#include <string>
using namespace std;
class MyClass
{
    public:
    string name="madhu";
    int a=10;
};
int main(){
    MyClass dog;
    dog.name="anil";
    dog.a=1;
    cout<<dog.name<<endl;
    cout<<dog.a;
    return 0;
}
```

O/P:

anil
1

- In C++, an object is created from a class. We have already created the class named , so now we can use this to create objects.
- To create an object of MyClass, specify the class name, followed by the object name.
- To access the class attributes (myNum and myString), use the dot syntax (.) on the object
- The object `dog` is created on the stack.
- It is automatically destroyed when it goes out of scope (e.g., when the `main()` function ends).
- Constructor and destructor calls happen automatically.

2.Heap/Dynamic Allocation: Objects can also be created on the heap using the `new` keyword.

Syntax: Type* objectPointer = new Type(arguments);

- Type is the class or data type of the object.
- objectPointer is a pointer that holds the address of the newly created object on the heap.
- new returns a pointer to the allocated object.

Example:

```
#include <iostream>
class MyClass {
public:
    MyClass() {
        std::cout << "Constructor called" << std::endl;
    }
    ~MyClass() {
        std::cout << "Destructor called" << std::endl;
    }
    void display() {
        std::cout << "Displaying MyClass object" << std::endl;
    }
```

```
};

int main() {
    // Dynamically allocate an object on the heap
    MyClass* obj = new MyClass();  // Constructor called
    // Use the object
    obj->display();
    // Manually delete the object to free the memory
    delete obj;  // Destructor called
    obj = nullptr;  // Optional: Setting the pointer to nullptr to avoid dangling
pointer
    return 0;
}
```

O/P:

```
Constructor called
Displaying MyClass object
Destructor called
```

NOTE:
Failing to call delete results in a memory leak, as the dynamically allocated memory will not be freed automatically.

Benefits of new:

- Allows for **dynamic memory management**, which is useful when the size or lifetime of the objects is not known at compile-time.
- Objects on the heap can outlive the scope in which they were created.

Drawbacks:

- You need to manually manage memory with delete to avoid memory leaks.
- Improper use can lead to bugs like dangling pointers and double deletion.
- In modern C++, it is often recommended to use **smart pointers** (std::unique_ptr, std::shared_ptr) instead of raw pointers to manage dynamic memory, as they automatically manage the object's lifetime and can help avoid manual memory management issues.

EXAMPLE:

```
#include<iostream>
#include <string>
using namespace std;
class MyClass
{
    public:
    string name="madhu";
    int a=10;
    void m(){
        cout<<"hii"<<endl;
    }
};
int main(){
    MyClass s1;
    //cout<<s1<<endl;
    s1.m();
   // cout<<*s1<<endl;
    MyClass* s=new MyClass();// Heap allocation
    s->name="sai";
    s->a=10;
    cout<<(s->name)<<endl;
    cout<<(s->a)<<endl;
    cout<<s<<endl;
    delete s;// Freeing memory
    //free(s);
    cout<<s<<endl;
    cout<<"fan"<<endl;
    s = nullptr;
    cout<<s;
}
```

O/P:

```
hii
sai
10
```

```
0x6892c0
0x6892c0
fan
0
```

V.V.IMP:

- The object is created on the heap with new, and its pointer is stored in s.
- You must call delete to destroy the object and release its memory, or else you will have a memory leak.
- The -> operator is used to access members (methods and variables) of the object through the pointer. This is shorthand for dereferencing the pointer and then accessing the member.
- objPtr->value is shorthand for (*objPtr).value. Both access the value member of the object, but the -> operator is simpler and more readable when working with pointers to objects.
- For accessing members of the object, you can use either (*ptr).member or ptr->member.

Class Methods

- Methods are functions that belong to the class.
- There are two ways to define functions that belongs to a class:
    - Inside class definition
    - Outside class definition

Inside Function: In the following example, we define a function inside the class, and we name it "myMethod".

Note: You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

```
#include <iostream>
using namespace std;
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {   // Method/function
      cout << "Hello World!";
    }
};
int main() {
  MyClass myObj;     // Create an object of MyClass
```

```
  myObj.myMethod();  // Call the method
  return 0;
}
```

O/P:

Hello World!

Outside function: To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed the scope resolution :: operator, followed by the name of the function:

- The purpose of the scope resolution operator is to tell the compiler which class the member function belongs to when defining it outside the class body.C++ allows you to define classes and member functions within namespaces or other classes. The scope resolution operator is used to clarify **which scope (class or namespace)** the member belongs to.

```
#include<iostream>  // Required for input/output operations
using namespace std;  // To avoid using std:: prefix for standard library elements
class Madhu {
public:
   void hello();  // Function declaration
};
// Definition of the member function hello() outside the class
void Madhu::hello() {
   cout << "Hello world";  // Output "hello world" to the console
}
int main() {
   Madhu m;  // Create an object of type Madhu
   m.hello();  // Call the hello() method on the object m
   return 0;   // Return 0 to indicate successful execution
}
```

O/P:

Hello world

Parameters
You can also add parameters:

```
#include <iostream>
using namespace std;
class Car {
  public:
    int speed(int maxSpeed);
};
int Car::speed(int maxSpeed) {
  return maxSpeed;
}
int main() {
  Car myObj;
  cout << myObj.speed(200);
  return 0;
}
```

O/P: 200

ENCAPSULATION

- Encapsulation is a process of binding or bundling of a data members with related functionalities into single unit is called encapsulation
- That single unit can be class or namespace
- In encapsulation we can achieve 2 things 1.Data Hiding 2.Abstraction
- In encapsulation we have 3 main points
- Accessing the data-That should be stop(by using private access modifier)
- Modifying the data-that can be allow by using setters
- Reading the data-By using getters we can allow

Example:

```
#include<iostream>
using namespace std;
// Class Madhu demonstrating encapsulation
class Madhu {
```

```
    // Private data members: can't be accessed directly outside the class
    private:
    int a;          // integer data member
    string name;    // string data member
        // Public methods (getters and setters) to control access to private members
    public:
    // Constructor to initialize the private members
    Madhu(int a1, string name1) {
        a = a1;       // assigning value to 'a' from the constructor parameter
            name = name1; // assigning value to 'name' from the constructor parameter
    }
    // Setter method to modify the value of 'a'
    void seta(int a2) {
        a = a2;  // updating the private member 'a' with the new value 'a2'
    }
    // Getter method to access the value of 'a'
    int geta() {
        return a;  // returning the current value of 'a'
    }
    // Setter method to modify the value of 'name'
    void setname(string name2) {
            name = name2;   // updating the private member 'name' with the new value 'name2'
    }
    // Getter method to access the value of 'name'
    string getname() {
        return name;  // returning the current value of 'name'
    }
};
int main() {
        // Creating an object 'm' of class 'Madhu' with initial values 10 and "madhu"
    Madhu m(10, "madhu");
    // Accessing and displaying the initial value of 'a' using the getter
    cout << m.geta() << endl;  // Output: 10
    // Accessing and displaying the initial value of 'name' using the getter
    cout << m.getname() << endl;  // Output: madhu
    // Indicating that values will be updated
```

```
    cout << "After updating values" << endl;
    // Updating the value of 'a' using the setter
    m.seta(20);
    // Accessing and displaying the updated value of 'a'
    cout << m.geta() << endl;  // Output: 20
    // Updating the value of 'name' using the setter
    m.setname("srujana");
    // Accessing and displaying the updated value of 'name'
    cout << m.getname() << endl;  // Output: srujana
    return 0;  // End of the program
}
```

O/P:

```
10
madhu
After updating values
20
srujana
```

This way, encapsulation ensures that the internal data can only be accessed and modified in a controlled manner.

ABSTRACTION

- Hiding the specifications showing the necessary information
- We can achieve this by using abstract and pure virtual functions
- Example: ATM GUI application

PURE VIRTUAL FUNCTIONS(like Interface)

A pure virtual function in C++ is a virtual function that is declared in a base class but has no definition (implementation) in that class. The idea is to enforce that any derived class must provide an implementation for that function. Classes containing pure virtual functions are known as abstract classes, and they cannot be instantiated directly.
A pure virtual function is declared using the following syntax:
virtual void function_name() = 0;

Here, `= 0` makes the function "pure virtual" and forces derived classes to override it.

Why Use Pure Virtual Functions?

- To **define an interface** that derived classes must follow.
- To **ensure uniform behavior** across different derived classes, while allowing each derived class to provide its own specific implementation.
- To create **abstract classes** that cannot be instantiated but provide a template for other classes.

```
#include<iostream>
using namespace std;
class Madhu{
   virtual void run()=0;
};
class Mad: public Madhu{
   public:
   void display(){
      cout<<"display function"<<endl;
   }
};
int main(){
   Mad m;
   m.display();
}
```

NOTE:

Your code has an issue because the derived class Mad inherits from the abstract class Madhu, but it does not implement the pure virtual function run(). Since Madhu is an abstract class (because it has a pure virtual function), any class that derives from it must implement the pure virtual function, otherwise, the derived class itself will also become abstract and cannot be instantiated.

```
#include<iostream>
using namespace std;
class Madhu{
   virtual void run()=0;
```

```
    virtual void eat()=0;
};
class Mad: public Madhu{
    public:
    void display(){
        cout<<"display function"<<endl;
    }
    void run() override{
        cout<<"I am running"<<endl;
    }
    void eat() override{
        cout<<"I am eating";
    }
};
int main(){
    Mad m;
    m.display();
    m.run();
    m.eat();
}
```

O/P:

```
display function
I am running
I am eating
```

Explanation:

1. **Abstract Class (Madhu)**:
    - Contains two pure virtual functions, run() and eat(). This makes Madhu an abstract class, which means you cannot create objects of this class.
2. **Derived Class (Mad)**:

- Implements the pure virtual functions run() and eat(), providing specific functionality for both methods. Additionally, it has its own method, display().

3. **Main Function**:
   - In the main() function, you create an instance of the Mad class and call its display(), run(), and eat() methods. Since the run() and eat() functions are overridden in Mad, their specific implementations are executed.

Key Points:

- By implementing the pure virtual functions, Mad becomes a concrete class, allowing you to create objects of this class.
- The use of virtual functions and inheritance promotes polymorphism and code reusability in C++. This structure allows for flexible and modular code design.

```
#include<iostream>
using namespace std;
class ATM{
   private:
   int Balance;
   public:
   int deposit;
   void createBalance(int a){
      Balance=a;
   }
   void deposite(int b){
      deposit=b;
      Balance=Balance+deposit;
   }
   void withdraw(int c){
      if(c<Balance){
         Balance=Balance-c;
         cout<<"Withdraw successfull"<<endl;
      }
```

```
        else{
            cout<<"Insuffient funds";
        }
    }
};
int main(){
    ATM m;
    m.createBalance(1000);
    //cout<<m.Balance<<endl;
    m.deposite(1000);
   // cout<<m.Balance<<endl;
    m.withdraw(1500);
   // cout<<m.Balance<<endl;
    m.withdraw(1000);
}
```

O/P:

```
Withdraw successfull
Insuffient funds
```

```
#include <iostream>
using namespace std;
class ATM {
private:
    int balance;  // Hiding internal data
public:
    // Constructor to initialize balance
    ATM(int initialBalance) {
        balance = initialBalance;
    }
    // Public method to deposit money
    void deposit(int amount) {
        balance += amount;
        cout << "Deposited: $" << amount << endl;
    }
    // Public method to withdraw money
```

```
    void withdraw(int amount) {
        if (amount <= balance) {
            balance -= amount;
            cout << "Withdrew: $" << amount << endl;
        } else {
            cout << "Insufficient balance!" << endl;
        }
    }
    // Public method to check balance
    void checkBalance() const {
        cout << "Current balance: $" << balance << endl;
    }
};
int main() {
    ATM myATM(1000);  // Create an ATM with an initial balance of 1000
    myATM.checkBalance();   // Access public function
    myATM.deposit(500);     // Deposit money
    myATM.withdraw(300);    // Withdraw money
    myATM.checkBalance();   // Check updated balance
    return 0;
}
```

O/P:

```
Current balance: $1000
Deposited: $500
Withdrew: $300
Current balance: $1200
```

INHERITANCE

- In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:
- derived class (child) - the class that inherits from another class
- base class (parent) - the class being inherited from
- To inherit from a class, use the : symbol.

- In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

```
#include <iostream>
#include <string>
using namespace std;
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};
// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};
int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

O/P:

```
Tuut, tuut!
Ford Mustang
```

In the below example there are eat method in parent and child so by using scope resolution we can call particular method

```
#include <iostream>
using namespace std;
// Base class
```

```
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
// Derived class
class Dog : public Animal {
public:
    void eat() {
        cout << "Barking..." << endl;
    }
};
int main() {
    Dog myDog;
    myDog.eat();  // Inherited from Animal
    //myDog.bark(); // Specific to Dog
    myDog.Animal::eat();
    return 0;
}
```

O/P:

```
Barking...
Eating…
```

Multiple Inheritance

By using scope resolution we can solve ambiguity

```
#include <iostream>
using namespace std;
// Base class 1
class Animal {
public:
    void eat() {
        cout << "Animal is eating..." << endl;
    }
};
```

```
// Base class 2
class Pet {
public:
    void eat() {
        cout << "Pet is playing..." << endl;
    }
};
// Derived class
class Dog : public Animal, public Pet {
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
int main() {
    Dog myDog;
    // Specify which eat() method to call using scope resolution
    myDog.Animal::eat(); // Calls the eat() method from Animal
    myDog.Pet::eat();    // Calls the eat() method from Pet
    myDog.bark();        // Calls the bark() method specific to Dog
    return 0;
}
```

O/P:

```
Animal is eating...
Pet is playing...
Barking…
```

[WEEK 01]: DAY 5

POLYMORPHISM

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. In C++, polymorphism enables you to invoke methods on objects without knowing their exact type at compile time. There are two main types of polymorphism in C++: compile-time (static) polymorphism and runtime (dynamic) polymorphism.

1. Compile-time Polymorphism (Method Overloading)

Compile-time polymorphism is achieved through function overloading and operator overloading.

Function Overloading

You can define multiple functions with the same name but different parameter types or numbers of parameters.

Example:

```
#include<iostream>
using namespace std;
class Madhu{
   public:
   int add(int a,int b){
      return a+b;
   }
   float add(float a,int b){
      return a+b;
   }
   string add(string a,string b){
      return a+b;
   }
};
int main(){
   Madhu m;
   cout<<m.add(10,20)<<endl;
```

```
    cout<<m.add(10.6f,20.5f)<<endl;
    cout<<m.add("ma","dhu")<<endl;
}
```

2. Runtime Polymorphism (Dynamic Binding)

Runtime polymorphism is achieved through **inheritance** and **virtual functions**. It allows you to invoke derived class methods through base class pointers or references.

Example:

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() {
        cout << "Animal speaks." << endl;
    }
};
class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks." << endl;
    }
};
class Cat : public Animal {
public:
    void speak() override {
        cout << "Cat meows." << endl;
    }
};
int main() {
     Animal* animal1 = new Dog(); // Base class pointer pointing to Dog object
    Animal* animal2 = new Cat(); // Base class pointer pointing to Cat object
    // Calls the appropriate overridden function
    animal1->speak(); // Output: Dog barks.
```

```
    animal2->speak(); // Output: Cat meows.
    // Clean up
    delete animal1;
    delete animal2;
    return 0;
}
```

O/P:

```
Dog barks.
Cat meows.
```

CONSTRUCTOR:

A constructor is a special method in c++
It does not return any value
It doesn’t include void too
The access modifier should be public,protected,private
The constructor has the same name as the class

```
#include <iostream>
using namespace std;
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};
int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the
constructor)
  return 0;
}
```

O/P:

```
Hello World!
```

Constructor Parameters and Constructor overloading

```
#include <iostream>
using namespace std;
class Circle {
private:
   double radius;
public:
   // Default constructor
   Circle() {
      radius = 1.0;  // Default radius
   }
   // Parameterized constructor
   Circle(double r) {
      radius = r;
   }
   // Function to calculate area
   double area() {
      return 3.14159 * radius * radius;
   }
};
int main() {
   Circle circle1;       // Calls the default constructor
   Circle circle2(5.0); // Calls the parameterized constructor
   cout << "Area of circle1: " << circle1.area() << endl; // Output: 3.14159
   cout << "Area of circle2: " << circle2.area() << endl; // Output: 78.53975
   return 0;
}
```

O/P:

```
Area of circle1: 3.14159
Area of circle2: 78.5397
```

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by

specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the constructor (which is the same as the class):

```
#include <iostream>
using namespace std;
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z); // Constructor declaration
};
// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}
int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);
  // Print values
   cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year <<
"\n";
   cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year <<
"\n";
  return 0;
}
```

O/P:

```
BMW X5 1999
Ford Mustang 1969
```

Copy Constructor:

- A constructor that initializes an object using another object of the same class.
- It is called when an object is passed by value, returned from a function, or explicitly copied.

Example(IMP)

```
#include <iostream>
using namespace std;
class MyClass {
private:
    int value;
public:
    // Default constructor
    MyClass() : value(0) {
        cout << "Default constructor called" << endl;
    }
    // Parameterized constructor
    MyClass(int v) : value(v) {
        cout << "Parameterized constructor called with value: " << value <<
endl;
    }
    // Copy constructor
    MyClass(const MyClass &obj) {
        value = obj.value;
        cout << "Copy constructor called, value: " << value << endl;
    }
    // Method to display the value
    void display() {
        cout << "Value: " << value << endl;
    }
};
int main() {
    MyClass obj1;           // Calls the default constructor
    MyClass obj2(42);       // Calls the parameterized constructor
    MyClass obj3 = obj2;    // Calls the copy constructor
    obj1.display();
```

```
    obj2.display();
    obj3.display();
    return 0;
}
```

O/P:

```
Default constructor called
Parameterized constructor called with value: 42
Copy constructor called, value: 42
Value: 0
Value: 42
Value: 42
```

Note: By default, all members of a class are private if you don't specify an access specifier:

DESTRUCTOR

A destructor is a special member function in C++ that is called when an object is destroyed. The primary purpose of a destructor is to free up resources that the object may have acquired during its lifetime, such as dynamic memory, file handles, or network connections.

Key Features of Destructors

- Naming: A destructor has the same name as the class, preceded by a tilde (~).
- No Parameters: Destructors do not take parameters and do not return a value.
- Automatic Invocation: The destructor is automatically called when the object goes out of scope or is explicitly deleted.
- Only One Destructor: A class can have only one destructor.

Example:

```
#include <iostream>
using namespace std;
```

```
class Sample {
private:
    int* data; // Pointer to dynamically allocated memory
public:
    // Constructor
    Sample(int value) {
        data = new int; // Allocate memory for an integer
        *data = value;  // Assign the value
        cout << "Constructor: Sample created with value " << *data << endl;
    }
    // Destructor
    ~Sample() {
        delete data; // Deallocate memory
          cout << "Destructor: Sample destroyed and memory deallocated." <<
endl;
    }
    // Function to display the value
    void display() {
        cout << "Value: " << *data << endl;
    }
};
int main() {
    Sample sample(42); // Create an object of Sample with value 42
    sample.display();  // Display the value
       // Destructor will be called automatically when 'sample' goes out of
scope
    return 0;
}
```

O/P:

```
Constructor: Sample created with value 42
Value: 42
Destructor: Sample destroyed and memory deallocated.
```