# C++ NOTES

# UNIT-1

## Objects

In C++, an object is an instance of a class. A class is a user-defined data type that encapsulates data and functions (known as member variables and member functions) into a single unit. When you create an object of a class, you allocate memory to hold the data members of the object and can use the member functions to operate on that data.

Objects in C++ are used to represent real-world entities or concepts. They allow you to model and manipulate data and behavior in a structured manner. Each object has its own set of data members and can perform operations specific to its class.

Here's an example to illustrate the concept of objects in C++:

```
#include <iostream>


// Class definition

class Car {

public:

   // Member variables

   std::string brand;

   std::string model;

   int year;


   // Member function

   void startEngine() {

      std::cout << "Engine started for " << brand << " " << model << std::endl;
```

```
    }
};

int main() {
    // Object creation
    Car myCar;

    // Accessing member variables
    myCar.brand = "Ford";
    myCar.model = "Mustang";
    myCar.year = 2020;

    // Calling member function
    myCar.startEngine();

    return 0;
}
```

In the example above, we define a class called `Car` that has three member variables: `brand`, `model`, and `year`. It also has a member function `startEngine` that displays a message indicating the car's brand and model.

Inside the `main` function, we create an object `myCar` of type `Car`. We then access the member variables of the `myCar` object using the dot operator (`.`) and assign values to them. Finally, we call the `startEngine` member function on the `myCar` object.

Objects allow you to create multiple instances of a class, each with its own set of data. They provide a way to organize and manipulate related data and behavior in a structured manner, making code more readable, modular, and reusable.

Note that in C++, you can also create objects dynamically using the `new` keyword and manage their memory manually. However, it is generally recommended to use automatic (stack-allocated) objects unless there is a specific need for dynamic allocation.

## **Basic terms and ideas of**

## **1.Abstraction;-**

_Abstraction is an important concept in object-oriented programming, including C++. It refers to the process of simplifying complex systems by breaking them down into smaller, more manageable units. In C++, abstraction is achieved through the use of classes, objects, and interfaces, hiding the internal details and providing a simplified view of the system to the user.

Here are some basic terms and ideas related to abstraction in C++:

1. Class: A class is a blueprint or template that defines the structure and behavior of objects. It encapsulates data members (attributes) and member functions (methods) that operate on the data. Classes provide the abstraction mechanism in C++ by defining the properties and operations of objects.

2. Object: An object is an instance of a class. It represents a specific occurrence of the class and holds its own set of data members and can perform operations defined by the class's member functions. Objects are the tangible entities that interact with each other in a C++ program.

3. Encapsulation: Encapsulation is the process of bundling data and the operations that manipulate that data together within a class. It allows you to hide the internal implementation details of a class and provide access to the data through well-defined interfaces. Encapsulation helps achieve information hiding and protects the integrity of data by controlling its access.

4. Data Abstraction: Data abstraction focuses on providing only the essential information about an object to the outside world, hiding the internal details. It allows you to define the characteristics and behaviors of an object without revealing how they are implemented. Data abstraction is achieved through the use of classes, where the public interface specifies the operations that can be performed on the object, while the private implementation details are hidden.

5. Interface: An interface defines a set of methods that a class must implement. It specifies a contract or protocol for how objects of a class should interact with the rest of the program. Interfaces allow you to separate the definition and implementation of a class, promoting loose coupling and enhancing code maintainability.

6. Modularity: Modularity is the division of a program into separate, self-contained modules or units. Abstraction helps in achieving modularity by encapsulating related data and behavior within classes. Each class represents a module that can be developed, tested, and maintained independently, contributing to code organization and reusability.

7. Inheritance and Polymorphism: Inheritance is a mechanism that allows a class to inherit properties (data and functions) from another class. It promotes code reuse and supports hierarchical relationships between classes. Polymorphism refers to the ability of objects of different classes to be treated as objects of a common base class, allowing them to be used interchangeably. Inheritance and polymorphism are advanced concepts that build upon abstraction, enabling more flexible and extensible designs.

Abstraction helps in managing the complexity of large software systems by providing clear boundaries and reducing dependencies between components. It allows you to focus on the essential aspects of a system while hiding the implementation details, making the code more maintainable, scalable, and reusable.

## 2.**Encapsulation:-**

Encapsulation is a fundamental principle of object-oriented programming (OOP) that focuses on bundling data and the operations that manipulate that data into a single unit. In C++, encapsulation is achieved using classes, which serve as containers for data members (attributes) and member functions (methods). The principle of encapsulation provides the following basic terms and ideas:

1. Class: A class is a user-defined data type that encapsulates data members and member functions into a single entity. It serves as a blueprint or template for creating objects. The class defines the structure and behavior of objects, including their attributes and the operations that can be performed on those attributes.

2. Data Members: Data members, also known as attributes or instance variables, are the variables defined within a class. They represent the state or characteristics of an object. Data members are typically declared as private or protected to restrict direct access from outside the class, enforcing encapsulation.

3. Member Functions: Member functions, also called methods, are functions defined within a class that operate on the data members of objects. They represent the behavior or actions that objects can perform. Member functions can be public, private, or protected, determining their accessibility from outside the class.

4. Access Specifiers: Access specifiers are keywords used to specify the visibility and accessibility of class members. In C++, there are three access specifiers: public, private, and protected. They control the level of encapsulation and determine which parts of the class are accessible to other parts of the program.

  - Public: Public members are accessible from anywhere in the program. They define the interface of the class and can be accessed by objects and external code.

- Private: Private members are accessible only within the class itself. They encapsulate the internal implementation details and are not directly accessible outside the class. Private members can be accessed through public member functions, providing controlled access to the class's data.

- Protected: Protected members are similar to private members, but they are also accessible in derived classes (class inheritance will be covered in more advanced topics). Protected members provide a level of encapsulation and are used when there is a need to share data and behavior among derived classes.

Encapsulation promotes information hiding, where the internal implementation details of a class are hidden from the outside world. By making data members private and providing public member functions to access or modify them, encapsulation ensures that the integrity and validity of the data are maintained. This concept provides several benefits, such as:

- Modularity: Encapsulation allows you to divide a complex system into smaller, self-contained units (classes), promoting code modularity. Each class encapsulates its own data and behavior, making it easier to understand, develop, test, and maintain.

- Data Protection: Encapsulation protects data from direct manipulation by external code. Access to data members is controlled through member functions, allowing the class to enforce business rules, validation, and data consistency.

- Code Flexibility: Encapsulation provides flexibility by allowing the internal implementation details of a class to change without affecting other parts of the program. As long as the public interface remains the same, other code that uses the class does not need to be modified.

- Code Reusability: Encapsulated classes can be reused in different programs or projects. By providing a well-defined interface, classes can be utilized without understanding their internal details, promoting code reuse and reducing development time.

Encapsulation is a key principle in achieving robust and maintainable code in C++ and other object-oriented programming languages. It helps manage complexity, protect data, and promote modular and reusable designs.

## **3.Inheritance:-**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows classes to inherit properties and behaviors from other classes. In C++, inheritance is implemented using the inheritance mechanism, which enables the creation of new classes (derived classes) based on existing classes (base classes). Here are the basic terms and ideas related to inheritance in C++:

1. Base Class and Derived Class: In inheritance, the existing class is referred to as the base class or parent class. The new class that inherits from the base class is called the derived class or child class. The derived class inherits the properties (data members) and behaviors (member functions) of the base class.

2. Inheritance Types:

- Public Inheritance: In public inheritance, the public and protected members of the base class become public and protected members of the derived class, respectively. Private members of the base class are not accessible in the derived class.

- Protected Inheritance: In protected inheritance, the public and protected members of the base class become protected members of the derived class. Private members of the base class are not accessible in the derived class.

- Private Inheritance: In private inheritance, all members of the base class become private members of the derived class. This means that the derived class can access the members of the base class, but they are not accessible from outside the derived class.

The access specifier used during inheritance determines the accessibility of the base class members in the derived class.

3. Derived Class Extension: The derived class can extend the functionality of the base class by adding its own additional data members and member functions. It can also override the base class's member functions to provide its own implementation. This allows customization and specialization of behavior based on the specific needs of the derived class.

4. Inherited Members: When a class is derived from another class, it inherits all the non-private members of the base class, including data members, member functions, and nested classes. The derived class can directly use and access these inherited members as if they were defined within the derived class itself.

5. Access Specifiers and Inheritance:

- Public Inheritance: The public members of the base class remain public in the derived class, and the protected members of the base class become protected members in the derived class.

- Protected Inheritance: Both public and protected members of the base class become protected members in the derived class.

- Private Inheritance: Both public and protected members of the base class become private members in the derived class.

The choice of access specifier during inheritance determines the visibility and accessibility of the inherited members in the derived class and its subclasses.

6. Single Inheritance and Multiple Inheritance: C++ supports both single inheritance and multiple inheritance. Single inheritance is when a derived class inherits from a single base class. Multiple inheritance is when a derived class inherits from multiple base classes. Multiple inheritance allows a class to inherit properties and behaviors from multiple sources, but it requires careful consideration to avoid ambiguity and maintain code clarity.

Inheritance is a powerful mechanism in C++ that facilitates code reuse, promotes modularity, and supports hierarchical relationships between classes. It allows the creation of specialized classes based on existing classes, enabling the implementation of more complex systems and achieving more flexible and extensible designs.

## 4.Polymorphism:-

Polymorphism is a key concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class. It enables the same interface (method or function) to be used with objects of different types, providing flexibility and extensibility in the code. In C++, polymorphism can be achieved through two main mechanisms: function overloading and virtual functions. Here are the basic terms and ideas related to polymorphism in C++:

1. Polymorphism: Polymorphism means "many forms." It refers to the ability of objects of different classes, derived from a common base class, to be treated as objects of the base class. This allows a single interface (method or function) to be used to perform different operations based on the actual type of the object at runtime.

2. Base Class and Derived Class: Polymorphism relies on inheritance. The base class is a common class that defines the shared interface or behavior. The derived classes are the specialized classes that inherit from the base class and provide their own implementation of the shared interface.

3. Function Overloading: Function overloading is a form of static (compile-time) polymorphism. It allows multiple functions with the same name but different parameters to be defined in the same scope. The compiler determines which version of the function to call based on the number, type, and order of the arguments during the function call.

4. Virtual Functions: Virtual functions are a feature of dynamic (runtime) polymorphism in C++. A virtual function is a member function declared in the base class with the `virtual` keyword. Derived classes can override the virtual function to provide their own implementation. When a virtual function is called using a pointer or reference to the base class, the actual implementation of the function is determined at runtime based on the type of the object, rather than the type of the pointer or reference.

5. Late Binding and Early Binding: Late binding and early binding are terms associated with polymorphism and virtual functions. Late binding (also called

dynamic binding or runtime binding) refers to the determination of the appropriate function implementation at runtime. It happens when a virtual function is called through a pointer or reference to the base class. Early binding (also called static binding or compile-time binding) refers to the determination of the appropriate function implementation at compile time. It occurs when a non-virtual function or a function call without a pointer or reference is resolved during compilation.

6. Pure Virtual Functions and Abstract Classes: A pure virtual function is a virtual function declared in the base class with no implementation provided. It is declared using the syntax `virtual return_type function_name() = 0;`. A class containing at least one pure virtual function is called an abstract class. Abstract classes cannot be instantiated and serve as a base for derived classes to provide concrete implementations of the pure virtual functions.

Polymorphism allows for code flexibility and reusability by enabling the use of a common interface to work with objects of different types. It simplifies code design, enhances maintainability, and supports code extensibility in the presence of class hierarchies.

## **Review of C:-**

C is a widely used programming language that was developed in the early 1970s. It has had a significant impact on the field of computer programming and has influenced the development of many other languages. Here is a review of some key aspects of C:

1. Simplicity: C is known for its simplicity and minimalistic design. It provides a small set of keywords and constructs, making it relatively easy to learn and understand. The language focuses on providing low-level control over hardware resources, making it suitable for systems programming and embedded systems.

2. Efficiency: C is a compiled language that produces efficient machine code. It allows direct manipulation of memory and provides features such as pointers and bitwise operations, making it efficient for tasks that require fine-grained control over memory and performance.

3. Portability: C was designed to be a portable language, meaning that C programs can be compiled and run on different platforms with minimal modifications. This portability is achieved by relying on a standardized set of features and libraries, known as the C standard library.

4. Low-Level Features: C provides low-level features, such as direct memory access, pointer arithmetic, and bitwise operations. These features allow programmers to manipulate memory and hardware resources at a fine-grained level, making C suitable for tasks like device drivers, operating systems, and embedded systems development.

5. Structured Programming: C supports structured programming, which encourages the use of modular code and control structures like loops, conditionals, and functions. This promotes code readability, maintainability, and reusability.

6. Preprocessor Directives: C includes a preprocessor that allows for conditional compilation and macro expansion. Preprocessor directives, such as `#include` and `#define`, provide flexibility in managing dependencies and code generation, although they can also introduce complexity and potential pitfalls if used improperly.

7. Lack of Built-in Abstractions: One notable characteristic of C is the absence of built-in high-level abstractions, such as classes, objects, and exception handling. While this provides flexibility, it requires developers to implement their own abstractions and error handling mechanisms.

8. Weak Type Checking: C has a weak type checking system, which means that it allows implicit conversions between different types. While this flexibility can be useful, it also increases the risk of errors and can make code harder to understand and maintain.

9. Memory Management: C requires manual memory management. Programmers need to explicitly allocate and deallocate memory using functions like `malloc` and `free`. This manual memory management can be challenging and error-prone, leading to issues like memory leaks and buffer overflows if not handled carefully.

10. Standard Library: C provides a standardized library, known as the C standard library, which includes functions for I/O, string manipulation, memory management, and other common operations. The standard library provides a basic set of functionalities, but it may lack some higher-level abstractions found in other languages.

Overall, C is a powerful and widely used programming language, particularly in systems programming and situations where low-level control and efficiency are essential. Its simplicity, efficiency, and portability make it a popular choice for various applications, although it does require careful attention to memory management and lacks some higher-level abstractions found in more modern languages

## **Operators**

## **Cin;-**

In C++, the `cin` operator is used for input in the console. It is part of the iostream library and is typically used to read input from the user. The `cin` operator is used in conjunction with the extraction operator (`>>`).

Here's an example of how `cin` can be used to read input from the user:

```
#include <iostream>

int main() {

   int num;

   std::cout << "Enter a number: ";

   std::cin >> num;

   std::cout << "You entered: " << num << std::endl;


   return 0;

}
```

In this example, the program prompts the user to enter a number. The `cin` operator is then used to read the input entered by the user and store it in the `num` variable. Finally, the program outputs the value entered by the user.

It's important to note that `cin` expects input to be formatted according to the data type of the variable it is reading into. If the user enters input of the wrong type, it can lead to errors or unexpected behavior. It's a good practice to validate the input to ensure it matches the expected format.

## Cout:-

In C++, the `cout` operator is used for output in the console. It is part of the iostream library and is typically used to display or print information to the user. The `cout` operator is used in conjunction with the insertion operator (`<<`).

Here's an example of how `cout` can be used to display output:

```
#include <iostream>

int main() {
    int age = 25;

    std::cout << "My age is: " << age << std::endl;

    std::cout << "This is a sentence." << std::endl;


    return 0;

}
```

In this example, the program uses `cout` to output messages to the console. The insertion operator (`<<`) is used to insert the value of the `age` variable into the output stream. Multiple insertion operators can be used consecutively to display multiple values or concatenate strings. The `std::endl` manipulator is used to insert a newline character and flush the output stream.

The output of the above program would be:

```
My age is: 25

This is a sentence.
```

By default, `cout` outputs the data in a formatted manner based on the data types being used. However, you can customize the formatting using manipulators and other techniques provided by the iostream library.

It's worth noting that `cout` is just one way to output information in C++. There are other stream objects available, such as `cerr` and `clog`, which are used for error and log outputs, respectively.

## **New:-**

In C++, the `new` operator is used to dynamically allocate memory for objects or arrays during runtime. It is primarily used for creating objects on the heap.

Here's an example of how the `new` operator can be used to dynamically allocate memory for a single object:

```
int* p = new int; // Dynamically allocate memory for an integer
```

In this example, the `new` operator is used to allocate memory for an integer on the heap. The result of the `new` operator is a pointer to the allocated memory. The `int* p` declaration creates a pointer `p` that can store the address of an integer.

To allocate memory for an array of objects, you can use the `new` operator with square brackets:

```
int* arr = new int[5]; // Dynamically allocate memory for an integer array of size 5
```

In this case, the `new` operator alloc

## <u>**Delete:-**</u>

In C++, the `delete` operator is used to deallocate memory that was previously allocated using the `new` operator. It is used to free the memory and release the resources that were allocated dynamically.

Here's an example of how the `delete` operator can be used to deallocate memory:

```
int* p = new int; // Dynamically allocate memory for an integer

// ...

delete p; // Deallocate the memory
```

In this example, the `new` operator is used to allocate memory for an integer on the heap, and the resulting pointer `p` holds the address of the allocated memory.

Later, when you're done using the dynamically allocated memory, you use the `delete` operator to deallocate it. This ensures that the memory is released and can be reused.

Similarly, when you allocate memory for an array using `new[]`, you need to use `delete[]` to deallocate the memory:

```
int* arr = new int[5]; // Dynamically allocate memory for an integer array of size 5

// ...

delete[] arr; // Deallocate the memory
```

In this case, the `new[]` operator allocates memory for an array of five integers on the heap, and the `delete[]` operator is used to deallocate the memory when it is no longer needed.

It's important to remember that whenever you use `new` or `new[]` to allocate memory, you must use `delete` or `delete[]`, respectively, to deallocate the memory. Failing to deallocate the memory can result in memory leaks and inefficient memory usage.

# UNIT-2

## Encapsulation:-

Encapsulation in C++ is a fundamental principle of object-oriented programming (OOP) that combines data and functions into a single unit called a class. It allows the class to control the access to its data members and provides a way to hide the implementation details from the outside world.

In C++, encapsulation is achieved by declaring the class's data members as private, and providing public member functions (also known as methods) to interact with and manipulate the data. The private data members are only accessible within the class, while the public member functions act as interfaces to access and modify the private data.

By encapsulating data and providing controlled access through member functions, encapsulation offers several benefits:

1. Data Hiding: Encapsulation hides the internal details of a class implementation from the outside. Only the public interface is exposed, providing abstraction and preventing direct manipulation of data, which helps maintain the integrity and consistency of the data.

2. Access Control: Encapsulation allows you to control the access to class members. By declaring data members as private, you can enforce restrictions on how the data can be accessed, ensuring proper validation and encapsulation of the internal state.

3. Modularity and Maintainability: Encapsulation helps in creating modular code. By encapsulating related data and operations into a single class, it promotes code organization and simplifies maintenance. Changes to the internal implementation of a class can be made without affecting the external code that uses the class, as long as the public interface remains the same.

4. Code Reusability: Encapsulated classes can be used as building blocks to create larger systems. By encapsulating related functionality, you can reuse the class in

different parts of the program or even in different projects, promoting code reuse and reducing development time.

Here's an example demonstrating encapsulation in C++:

```
class Car {
private:
   int speed;

public:
   void setSpeed(int s) {
      if (s >= 0) {
         speed = s;
      }
   }

   int getSpeed() {
      return speed;
   }
};
```

In this example, the `Car` class encapsulates the `speed` data member as private, making it inaccessible from outside the class. The public member functions `setSpeed()` and `getSpeed()` provide a controlled interface to set and retrieve the `speed` value. By encapsulating the `speed` data and providing public methods, the class ensures proper validation and controlled access to the internal state.

## Information hiding:-

Information hiding is a principle of object-oriented programming (OOP) that is closely related to encapsulation. It emphasizes the idea of hiding the internal details of a class or module and exposing only what is necessary for the outside world to use. The goal is to provide a clear and well-defined interface while keeping the implementation details hidden and protected.

In C++, information hiding is achieved through encapsulation. By declaring the data members of a class as private, and providing public member functions (methods) to access and manipulate those data members, you establish a clear boundary between the internal implementation and the external usage.

The concept of information hiding has several benefits:

1. Abstraction: Information hiding allows you to present a simplified view of an object or module to the outside world. By hiding the internal details, you can focus on the essential features and behaviors that need to be exposed, promoting abstraction and reducing complexity.

2. Modularity: Information hiding facilitates modular design. It allows you to encapsulate related data and functions into a single unit (class/module), making it easier to understand, test, and maintain. Changes to the internal implementation can be made without affecting the external code as long as the public interface remains unchanged.

3. Security and Data Integrity: Hiding the internal details of a class helps protect the integrity of the data and prevents unauthorized access. By controlling access through public methods, you can enforce validation, apply business rules, and ensure the consistency and correctness of the data.

4. Code Maintenance and Evolution: Information hiding improves code maintainability and evolution. The hidden implementation details are shielded from external dependencies, allowing for easier modifications and updates. The internal

implementation can be changed without affecting the external code, reducing the risk of introducing bugs or breaking existing functionality.

In summary, information hiding, achieved through encapsulation, promotes encapsulating the implementation details of a class or module, exposing a well-defined interface, and providing the benefits of abstraction, modularity, security, and maintainability.

## Abstract data types:-

An abstract data type (ADT) in C++ is a concept in computer science and programming that refers to a data type whose behavior is defined by a set of operations, but not by its internal implementation. It provides a high-level description of how the data can be manipulated and what operations can be performed on it, without specifying the details of how those operations are implemented.

ADTs are typically defined using classes in C++. The class serves as a blueprint for creating objects that represent instances of the ADT. The public interface of the class defines the operations that can be performed on the data, while the private implementation details are hidden from the user.

The primary goal of an ADT is to provide a clear and well-defined abstraction of a data structure or a collection of data, enabling users to focus on what operations can be performed rather than how they are implemented. This promotes code modularity, reusability, and makes programs easier to understand and maintain.

Here's an example of an ADT called a Stack, which follows the Last-In-First-Out (LIFO) principle:

```
class Stack {
```

```
public:
    virtual void push(int element) = 0;
    virtual int pop() = 0;
    virtual int top() = 0;
    virtual bool isEmpty() = 0;
};
```

In this example, the `Stack` class defines a set of operations that can be performed on a stack, such as `push`, `pop`, `top`, and `isEmpty`. However, the class is declared as abstract using the `virtual` keyword and pure virtual functions (`= 0`), indicating that it cannot be instantiated directly. Instead, other classes can inherit from this abstract class and provide their own implementations for the pure virtual functions.

By defining the ADT as an abstract class, the implementation details of the stack are left to the derived classes, allowing for different implementations based on specific requirements or performance considerations. The users of the ADT only need to know and use the defined operations without being concerned about how they are implemented.

In summary, an abstract data type in C++ is a data type that defines a set of operations without specifying the internal implementation details. It provides an abstraction and promotes code modularity, reusability, and maintainability.

## Objects and Classes:-

In C++, objects and classes are fundamental concepts of object-oriented programming (OOP).

A class is a blueprint or a template that defines the structure and behavior of objects. It encapsulates data (called data members) and functions (called member functions or methods) that operate on that data. The class serves as a template for creating instances of objects, which are specific instances of the class.

Here's an example of a simple class in C++:

```
class Rectangle {
private:
   int width;
   int height;

public:
   void setDimensions(int w, int h) {
      width = w;
      height = h;
   }

   int calculateArea() {
      return width * height;
   }
};
```

In this example, the `Rectangle` class has two private data members (`width` and `height`) to represent the dimensions of a rectangle. It also provides two member functions (`setDimensions` and `calculateArea`) to manipulate and retrieve information about the rectangle.

An object is an instance of a class. It is created based on the blueprint provided by the class. Each object has its own set of data members and can invoke the member functions defined in the class.

Here's an example of creating objects of the `Rectangle` class and using them:

```
int main() {
  Rectangle rect1; // Creating an object of the Rectangle class
  rect1.setDimensions(5, 3); // Setting the dimensions of rect1
  int area = rect1.calculateArea(); // Calculating the area of rect1

  Rectangle rect2; // Creating another object of the Rectangle class
  rect2.setDimensions(7, 2); // Setting the dimensions of rect2
  int area2 = rect2.calculateArea(); // Calculating the area of rect2

  return 0;
}
```

In this example, `rect1` and `rect2` are two objects of the `Rectangle` class. Each object has its own set of data members (`width` and `height`) and can invoke the member functions (`setDimensions` and `calculateArea`) to manipulate and retrieve information specific to that object.

Objects in C++ allow you to create multiple instances of a class, each with its own state and behavior. They enable you to model and work with real-world entities or abstract concepts as self-contained units with their own properties and actions.

By using classes and objects, you can organize your code into reusable and modular units, allowing for better code organization, encapsulation, and code reuse.

## Attributes:-

In C++, attributes refer to the characteristics or properties of an object or class. They represent the data associated with an object or class and provide information about its state or behavior.

In object-oriented programming, attributes are typically implemented as data members within a class. They define the data that an object can store and manipulate. The attributes of a class are specified in the class definition and can have different data types, such as integers, floating-point numbers, characters, strings, or even other objects.

Here's an example of a class with attributes (data members) in C++:

```
class Person {
private:
   std::string name;
   int age;

public:
   void setName(const std::string& n) {
      name = n;
   }

   void setAge(int a) {
        age = a;
    }

    std::string getName() const {
        return name;
    }

    int getAge() const {
        return age;
    }
};
```

In this example, the `Person` class has two attributes: `name` and `age`. The `name` attribute represents the name of a person and is of type `std::string`, while the `age` attribute represents the age of the person and is of type `int`. These attributes are declared as private, meaning they can only be accessed or modified through member functions (such as `setName`, `setAge`, `getName`, and `getAge`) defined within the class.

Attributes provide the necessary data for objects to represent real-world entities or abstract concepts. They encapsulate the state of an object and allow objects to have unique values for their attributes, which can be set or retrieved using appropriate member functions. Attributes play a crucial role in defining the characteristics and behavior of objects and are an essential part of modeling data in C++ programs.

## **Methods:-**

_In C++, methods are member functions of a class that define the behavior or actions that objects of the class can perform. They are functions that are associated with a specific class and are used to manipulate the data members of that class or provide specific functionality.

Methods are declared and defined within the class definition and can access the class's data members and other member functions. They can perform operations on the data, modify the state of the object, and provide an interface for interacting with the object's behavior.

Here's an example of a class with methods in C++:

```
class Rectangle {
private:
   int width;
   int height;

public:
   void setDimensions(int w, int h) {
      width = w;
      height = h;
   }

   int calculateArea() {
      return width * height;
   }
};
```

In this example, the `Rectangle` class has two methods: `setDimensions` and `calculateArea`. The `setDimensions` method takes two integer parameters `w` and `h` and sets the `width` and `height` data members of the object. The `calculateArea` method calculates and returns the area of the rectangle by multiplying the `width` and `height`.

To use these methods, you would create an object of the `Rectangle` class and invoke the methods on that object:

```
int main() {
   Rectangle rect;
   rect.setDimensions(5, 3);
   int area = rect.calculateArea();

   return 0;
}
```

In this example, an object `rect` of the `Rectangle` class is created. The `setDimensions` method is called on the `rect` object to set its dimensions to 5 and 3. Then, the `calculateArea` method is called to calculate the area of the rectangle and store it in the `area` variable.

Methods in C++ enable objects to perform actions, modify their state, and provide functionality specific to the class. They allow for encapsulation of behavior within the class and provide an interface for users of the class to interact with its functionality.

## C++ classes declaration:-

_In C++, class declaration is the process of defining the structure and member functions (methods) of a class without providing the implementation details. It serves as a blueprint or a template for creating objects of that class.

The declaration of a class typically includes the class name, data members (attributes), and member function signatures (prototypes). It provides information about the interface of the class, specifying what operations can be performed on objects of that class.

Here's an example of a class declaration in C++:

```
class Rectangle {
private:
   int width;
   int height;

public:
   void setDimensions(int w, int h);
   int calculateArea();
};
```

In this example, the `Rectangle` class is declared. It has two private data members (`width` and `height`) and two member function prototypes (`setDimensions` and `calculateArea`). The private access specifier indicates that the data members are only accessible within the class, while the public access specifier indicates that the member functions are accessible from outside the class.

The member function prototypes provide the signature of the functions, specifying the return type, function name, and parameters. The implementation details of these functions are not provided in the declaration; they will be defined separately.

After declaring a class, you can create objects of that class and define the member functions to provide the implementation details. Here's an example of defining the member functions of the `Rectangle` class:

```
void Rectangle::setDimensions(int w, int h) {
   width = w;
   height = h;
}


int Rectangle::calculateArea() {
   return width * height;
}
```

In this example, the member functions `setDimensions` and `calculateArea` are defined outside the class declaration using the scope resolution operator (`::`). The implementation details of these functions specify how the operations should be performed.

Class declarations in C++ allow you to define the structure and interface of a class separately from its implementation. This promotes code modularity, encapsulation, and reusability, as other parts of the program can use the declared class without being concerned about the implementation details.

## **State identify and behavior of an object:-**

In C++, an object is an instance of a class. It encapsulates data and behavior together. Let's go through the steps to define, identify, and interact with an object in C++.

1. Define a class:

```
 class MyClass {
 private:
   int myNumber;
   string myText;

 public:
   void setNumber(int number) {
     myNumber = number;
   }

   void setText(const string& text) {
     myText = text;
   }

   void print() {
     cout << "Number: " << myNumber << endl;
     cout << "Text: " << myText << endl;
   }
 };
```

2. Create an object:

```
MyClass myObject;
```

3. Identify the object:

In C++, you can use the object's name directly to identify it. For example, `myObject`.

4. Access and modify object's data:

```
myObject.setNumber(42);
myObject.setText("Hello, World!");
```

5. Invoke object's behavior:

```
myObject.print();
```

The `print()` function defined in the class will be called, displaying the object's data:

```
Number: 42
Text: Hello, World!
```

So, in this example, the object `myObject` of type `MyClass` is identified by its name and behaves by calling its member functions (`setNumber`, `setText`, and `print`) to manipulate and display its data.

Note that in practice, it is common to use constructors and access modifiers (like `private` and `public`) to define the behavior and control the access to the object's data. This example provides a basic illustration of object identification and behavior in C++.

## Constructor and destructors:-

Constructors and destructors are special member functions in C++ that are used to initialize and clean up the objects of a class, respectively. They are automatically called when objects are created or destroyed.

1. Constructors:

Constructors are member functions that are invoked automatically when an object of a class is created. Their purpose is to initialize the object's data members and set up its initial state. Constructors have the same name as the class and do not have a return type, not even `void`.

There are several types of constructors:

a. Default Constructor:

A default constructor is a constructor that takes no arguments. It is used to create objects without providing any initial values explicitly.

b. Parameterized Constructor:

A parameterized constructor is a constructor that takes one or more parameters. It allows you to provide initial values to the object's data members at the time of object creation.

c. Copy Constructor:

A copy constructor is a constructor that creates a new object by copying the values of another object of the same class. It is invoked when a new object is initialized using an existing object.

Here's an example of a class with different types of constructors:

```
class MyClass {
private:
  int myNumber;

public:
  // Default Constructor
  MyClass() {
    myNumber = 0;
  }

  // Parameterized Constructor
  MyClass(int number) {
    myNumber = number;
  }

  // Copy Constructor
  MyClass(const MyClass& other) {
    myNumber = other.myNumber;
  }
  };
```

2. Destructors:

Destructors are member functions that are automatically called when an object is about to be destroyed or goes out of scope. They are used to release resources, perform cleanup tasks, or deallocate memory that was allocated by the object.

The destructor has the same name as the class, preceded by a tilde (~). It does not take any arguments and does not have a return type, not even `void`.

```
  class MyClass {
  private:
    int* dynamicArray;

  public:
    // Constructor
    MyClass() {
      dynamicArray = new int[10];
    }

    // Destructor
    ~MyClass() {
      delete[] dynamicArray;
    }
  };
```

In the above example, the destructor deallocates the memory allocated by the `dynamicArray` in the constructor. It ensures that resources are properly released when the object is destroyed.

It's important to note that if you don't define a destructor explicitly, the compiler generates a default destructor for you. However, if your class manages resources that need cleanup (e.g., dynamically allocated memory), it's often necessary to provide a custom destructor to ensure proper cleanup.

Constructors and destructors play a crucial role in managing the lifecycle of objects, allowing proper initialization and cleanup operations as needed.

## **Instantiation of objects:-**

Instantiation of objects in C++ refers to the process of creating instances or individual objects of a class. When you instantiate an object, you allocate memory for it and initialize its data members using a constructor. It allows you to create multiple independent objects that have their own distinct data.

To instantiate an object in C++, you follow these steps:

1. Define a class:

```
  class MyClass {
  private:
    int myNumber;
    string myText;

  public:
    MyClass(int number, const string& text) {
      myNumber = number;
      myText = text;
    }

    void print() {
      cout << "Number: " << myNumber << endl;
      cout << "Text: " << myText << endl;
    }
  };
```

2. Create an object instance:

```
  MyClass obj(42, "Hello, World!");
```

The above code creates an object `obj` of the class `MyClass`. It calls the constructor `MyClass(int, const string&)` to initialize the object with the provided values.

3. Interact with the object:

You can now access the object's data members and invoke its member functions.

```
  obj.print();
```

The `print()` function defined in the class is invoked on the `obj` object, displaying its data:

```
Number: 42
Text: Hello, World!
```

You can create multiple objects of the same class, each with its own data and behavior. Each object instance is independent of others and can be manipulated separately.

```
MyClass obj1(10, "Object 1");
MyClass obj2(20, "Object 2");

obj1.print();  // Number: 10, Text: Object 1
obj2.print();  // Number: 20, Text: Object 2
```

In this example, two objects `obj1` and `obj2` are instantiated from the `MyClass` class. Each object holds its specific data and can be operated on separately.

Instantiating objects allows you to create multiple instances of a class, enabling you to work with individual objects and utilize the encapsulated data and behavior defined within the class.

## Default parameter value:-

In C++, you can specify default parameter values for function parameters. A default parameter value is a value that is automatically used if no argument is provided for that parameter when calling the function. It allows you to define functions with flexible behavior, as the default values provide a fallback option when specific arguments are not provided.

Here's the syntax for specifying default parameter values:

```
return_type function_name(parameter_type parameter_name = default_value);
```

Let's see an example:

```
#include <iostream>

void printMessage(const std::string& message = "Hello, World!") {
   std::cout << message << std::endl;
}

int main() {
   printMessage();  // Uses the default value "Hello, World!"
   printMessage("Custom message");   // Uses the provided argument "Custom message"

   return 0;
}
```

In the above code, the `printMessage()` function has a default parameter value for the `message` parameter, which is set to `"Hello, World!"`. If no argument is provided when calling the function, it uses the default value. However, you can also pass a different message as an argument, and it will override the default value.

Output:

```
Hello, World!
Custom message
```

Note that default parameter values are typically specified in the function declaration (header file), while the function definition (implementation) usually does not include the default values. This is to ensure consistency across different translation units (source files) that may include the header file.

You can also have multiple parameters with default values in a function, allowing you to selectively override specific parameters while using the default values for the rest.

```
void printNumbers(int a = 0, int b = 0, int c = 0) {
    std::cout << "Numbers: " << a << ", " << b << ", " << c << std::endl;
}

// Usage
printNumbers();             // Numbers: 0, 0, 0
printNumbers(1);            // Numbers: 1, 0, 0
printNumbers(1, 2);         // Numbers: 1, 2, 0
printNumbers(1, 2, 3);      // Numbers: 1, 2, 3
printNumbers(1, 2, 3, 4);   // Error: Too many arguments
```

In this example, the `printNumbers()` function has three parameters with default values. You can choose to provide values for any subset of the parameters while relying on the default values for the rest. Attempting to provide more arguments than there are parameters in the function declaration will result in a compilation error.

Default parameter values offer flexibility when calling functions by allowing you to omit certain arguments while still providing sensible default values.

## Object types:-

In C++, objects can have different types based on the class they are instantiated from. The type of an object determines the set of operations that can be performed on it and how it behaves. Here are some common object types in C++:

1. Built-in Types:

C++ provides several built-in types, such as `int`, `float`, `char`, `bool`, etc. Objects of these types are instantiated directly without the need for a class definition.

Example:

```
int num = 42;

float value = 3.14;

char letter = 'A';
```

2. Standard Library Types:

The C++ Standard Library provides various types and containers, such as `std::string`, `std::vector`, `std::map`, etc. Objects of these types are instantiated from the standard library classes.

```
};
```

Example:

```
std::string text = "Hello, World!";
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::map<std::string, int> scores = {{"Alice", 90}, {"Bob", 85}};
```

3. User-Defined Types:

User-defined types are created by defining a class or struct. Objects of these types are instantiated from the defined classes or structs.

Example:

```
class MyClass {
private:
  int myNumber;
  std::string myText;

public:
  // Constructor
  MyClass(int number, const std::string& text) {
    myNumber = number;
    myText = text;
  }
};
```

```
};

// Instantiate an object of MyClass
MyClass obj(42, "Hello");
```

4. Derived Types:

In object-oriented programming, you can create derived classes or subclasses that inherit properties and behavior from a base class. Objects of derived classes can be instantiated and have both the type of the derived class and the type of the base class.

Example:

```
class Animal {
public:
  void sound() {
    std::cout << "Animal makes a sound." << std::endl;
  }
};

class Dog : public Animal {
public:
  void sound() {
    std::cout << "Dog barks." << std::endl;
  }
};
```

```
  // Instantiate objects of both base and derived classes

  Animal animal;

  Dog dog;


  animal.sound();  // "Animal makes a sound."

  dog.sound();     // "Dog barks."
```

In C++, objects can have different types depending on the class they are instantiated from. The type determines the available operations and behaviors associated with the object, allowing for a flexible and polymorphic programming approach.

## <u>**C++ garbage collection:-**</u>

C++ does not have built-in automatic garbage collection like some other programming languages (e.g., Java or C#). In C++, memory management is typically done manually by the programmer using the concepts of dynamic memory allocation and deallocation.

C++ uses a combination of stack-based and heap-based memory management:

1. Stack Allocation:

Variables declared within a function or a block are typically allocated on the stack. Memory for these variables is automatically allocated when the program enters the scope and automatically deallocated when the scope is exited. The stack is managed by the compiler and follows a Last-In-First-Out (LIFO) approach.

Example:

```
  void foo() {
    int x = 10;  // x is allocated on the stack
    // ...
  }
```

2. Heap Allocation:

Memory allocation on the heap, also known as dynamic memory allocation, is done using operators like `new` and `delete` or `new[]` and `delete[]`. The programmer explicitly requests memory from the heap and is responsible for deallocating it when it is no longer needed. If deallocation is not done correctly, it can lead to memory leaks.

Example:

```
  void bar() {
    int* y = new int;  // Allocate memory on the heap
    // ...
    delete y;  // Release the allocated memory
  }
```

To manage dynamic memory effectively and avoid memory leaks or dangling pointers, it is common practice to use smart pointers or RAII (Resource Acquisition Is Initialization) techniques in C++. Smart pointers like `std::shared_ptr`, `std::unique_ptr`, and `std::weak_ptr` provide automatic memory management and help prevent common memory-related issues.

C++ also provides libraries and frameworks that offer garbage collection-like functionality. For example, the Boehm garbage collector (libgc) is a popular third-party library that can be used with C++ to provide automatic garbage collection capabilities.

It's important to note that manual memory management in C++ gives developers more control over resource allocation and deallocation, but it also requires careful handling to ensure efficient memory usage and prevent memory-related issues.

## Dynamic memory allocation:-

Dynamic memory allocation in C++ allows you to allocate and manage memory at runtime. It enables you to create objects of variable size and lifetime, and it is commonly used when you need to allocate memory for objects whose size is unknown at compile time or when you want to explicitly control the lifetime of an object.

In C++, dynamic memory allocation is performed using the `new` and `delete` operators for single objects, and the `new[]` and `delete[]` operators for arrays.

1. Allocating a Single Object:

```
  // Allocate memory for a single integer
  int* ptr = new int;
  *ptr = 42;

  // Use the allocated memory
  std::cout << *ptr << std::endl;
  // Deallocate the memory
  delete ptr;
```

In the above example, `new int` allocates memory for a single integer on the heap. The allocated memory is accessed using the pointer `ptr`, and the value is set to 42. Finally, `delete ptr` deallocates the memory when it is no longer needed.

2. Allocating an Array:

```
  // Allocate memory for an array of integers
  int* arr = new int[5];

  // Initialize the array elements
  for (int i = 0; i < 5; i++) {
     arr[i] = i;
  }

  // Use the allocated array
  for (int i = 0; i < 5; i++) {
     std::cout << arr[i] << " ";
  }
  std::cout << std::endl;

  // Deallocate the array memory
  delete[] arr;
```

Here, `new int[5]` allocates memory for an array of five integers on the heap. The array elements are accessed using the pointer `arr`, and the array is initialized and used. Finally, `delete[] arr` deallocates the array memory.

It's important to note that memory allocated using `new` or `new[]` must be explicitly deallocated using `delete` or `delete[]` to prevent memory leaks. Failure to deallocate the memory results in memory leaks, where memory remains allocated even when it is no longer needed.

Additionally, C++ provides smart pointers (`std::shared_ptr`, `std::unique_ptr`, `std::weak_ptr`) and standard library containers (`std::vector`, `std::string`, etc.) that manage memory automatically, making memory management safer and more convenient.

When using dynamic memory allocation, it's crucial to ensure proper memory management to prevent memory leaks, access violations, or undefined behavior.

## Meta class/abstract class:-

In C++, there are no built-in concepts of "meta classes" or "abstract classes" like in some other programming languages. However, C++ provides features and techniques that can achieve similar functionality.

1. Abstract Class:

An abstract class in C++ is a class that is designed to be a base class for other classes. It cannot be instantiated on its own, but it provides a common interface and defines virtual functions that derived classes must implement. An abstract class is often used to create a common interface for a group of related classes.

To create an abstract class in C++, you declare one or more pure virtual functions by adding the `virtual` keyword and setting them to `0`. A pure virtual function is a function that has no implementation in the base class.

Example:

```
class AbstractClass {
public:
  virtual void pureVirtualFunction() = 0;

  // Other member functions...
};
```

Any class that inherits from an abstract class must implement all the pure virtual functions. If a derived class does not implement all the pure virtual functions, it becomes an abstract class as well.

2. Meta Class:

The concept of "meta classes" is not part of the C++ language itself. However, you can use various techniques and libraries in C++ to achieve metaprogramming, which allows you to perform compile-time computations, generate code, and manipulate types at compile-time.

Metaprogramming in C++ is typically achieved using template metaprogramming techniques, where templates are used to perform computations and generate code during the compilation process. The C++ Standard Library also provides utilities like `std::type_info` and `std::type_traits` that allow runtime type information and type manipulation.

Example (template metaprogramming):

```
template <typename T>
struct MetaClass {
  // MetaClass implementation...
};
```

In the above example, `MetaClass` is a template struct that can be used for metaprogramming purposes. You can specialize the template for different types to perform different operations or generate code based on the type.

Note that the actual functionalities and techniques used for metaprogramming in C++ can be quite complex and advanced. They often involve template specialization, template metaprogramming techniques, constexpr functions, type traits, and advanced template features like variadic templates.

In summary, while C++ does not have direct concepts of "meta classes" or "abstract classes," you can achieve similar functionality using abstract classes and virtual functions for abstraction and interfaces, and template metaprogramming techniques for compile-time computations and code generation.

# UNIT-3

## Inheritance:-

Inheritance is a fundamental feature of object-oriented programming that allows you to define a new class (called the derived class) based on an existing class (called the base class). In C++, inheritance is used to establish relationships between classes, promote code reuse, and create class hierarchies.

To create a derived class from a base class, you use the `class` or `struct` keyword followed by the name of the derived class, a colon, and the access specifier indicating the type of inheritance (public, protected, or private). The derived class can access the public and protected members of the base class, while the private members of the base class remain inaccessible to the derived class.

Here's the syntax for inheriting from a base class in C++:

```
class BaseClass {
  // Base class members
};

class DerivedClass : [access-specifier] BaseClass {
  // Derived class members
};
```

There are three types of inheritance in C++:

1. Public Inheritance:

   - Public members of the base class become public members of the derived class.

- Protected members of the base class become protected members of the derived class.
- Private members of the base class remain inaccessible to the derived class.

```
class BaseClass {
public:
  int publicMember;
protected:
  int protectedMember;
private:
  int privateMember;
};

class DerivedClass : public BaseClass {
  // Derived class members can access publicMember and protectedMember
};
```

2. Protected Inheritance:

- Public and protected members of the base class become protected members of the derived class.
- Private members of the base class remain inaccessible to the derived class.

```
class DerivedClass : protected BaseClass {
  // Derived class members can access publicMember and protectedMember
  };
```

3. Private Inheritance:

- Public and protected members of the base class become private members of the derived class.
- Private members of the base class remain inaccessible to the derived class.

```
  class DerivedClass : private BaseClass {
    // Derived class members can access publicMember and protectedMember
  };
```

Derived classes can override base class member functions by providing their own implementation using the same function signature. This is known as function overriding. In C++, you use the `virtual` keyword to enable dynamic dispatch and allow polymorphic behavior.

```
class BaseClass {
public:
  virtual void someFunction() {
    // Base class implementation
  }
};

class DerivedClass : public BaseClass {
public:
  void someFunction() override {
    // Derived class implementation
  }
};
```

Inheritance in C++ allows you to create class hierarchies and take advantage of polymorphism, code reuse, and abstraction. It is an essential mechanism for building complex object-oriented systems.

**Class hierarchy:-**

In C++, class hierarchy refers to the organization of classes into a hierarchical structure, where classes are arranged in a parent-child relationship. The parent class is called the base class or superclass, and the child class is called the derived class or subclass. This hierarchy allows for code reuse, polymorphism, and abstraction.

Here's an example of a simple class hierarchy in C++:

```
class Shape {
public:
  virtual void draw() {
    // Base class implementation
  }
};

class Circle : public Shape {
```

```
public:
  void draw() override {
    // Circle-specific implementation
  }
};

class Square : public Shape {
public:
  void draw() override {
    // Square-specific implementation
  }
};
```

In the above example, `Shape` is the base class, and `Circle` and `Square` are derived classes. Both `Circle` and `Square` inherit from `Shape`. Each class in the hierarchy can override the `draw()` function to provide its own implementation.

This class hierarchy allows you to treat objects of derived classes as objects of the base class, enabling polymorphism. For example:

```
void drawShape(Shape* shape) {
  shape->draw();  // Calls the appropriate draw() implementation
}

int main() {
  Circle circle;

  Square square;


  drawShape(&circle);  // Calls the draw() function of Circle

  drawShape(&square);  // Calls the draw() function of Square


  return 0;

}
```

In the above `main()` function, the `drawShape()` function takes a pointer to a `Shape` object. When called with a `Circle` object or a `Square` object, it calls the appropriate `draw()` function based on the actual type of the object, thanks to dynamic dispatch and polymorphism.

Class hierarchies can be more complex and can include multiple levels of inheritance, with classes inheriting from other derived classes. You can create specialized behavior in derived classes while still benefiting from the common functionality provided by the base class.

Class hierarchies are a powerful tool in object-oriented programming for organizing and structuring code, promoting code reuse, and enabling polymorphism and abstraction. They allow you to model relationships between different types of objects and build complex systems.

## Derivations:-

In C++, derivation refers to the process of creating a new class (derived class or subclass) from an existing class (base class or superclass). The derived class inherits the properties (data members) and behaviors (member functions) of the base class, allowing for code reuse and the extension of functionality.

To derive a class in C++, you use the `class` or `struct` keyword followed by the derived class name, a colon, and the access specifier (public, protected, or private) indicating the type of inheritance. The access specifier determines the accessibility of the base class members in the derived class.

Here's the syntax for deriving a class in C++:

```
class BaseClass {
  // Base class members
};


class DerivedClass : [access-specifier] BaseClass {
  // Derived class members
};
```

There are three types of inheritance in C++ based on the access specifier:

1. Public Inheritance:
   - Public members of the base class become public members of the derived class.
   - Protected members of the base class become protected members of the derived class.
   - Private members of the base class are not accessible in the derived class.

```
class DerivedClass : public BaseClass {
  // Derived class members
};
```

2. Protected Inheritance:

   - Public and protected members of the base class become protected members of the derived class.
   - Private members of the base class are not accessible in the derived class.

```
class DerivedClass : protected BaseClass {
  // Derived class members
};
```

3. Private Inheritance:

   - Public and protected members of the base class become private members of the derived class.
   - Private members of the base class are not accessible in the derived class.

```
class DerivedClass : private BaseClass {
  // Derived class members
};
```

Derived classes can access and use the public and protected members of the base class, including data members and member functions. They can also override base class member functions by providing their own implementation using the same function signature. This is known as function overriding.

In addition to inheriting from a single base class, C++ supports multiple inheritance, where a derived class can inherit from multiple base classes. This allows for the creation of more complex class hierarchies and the combination of functionality from multiple sources.

It's important to note that access specifiers control the visibility and accessibility of inherited members in the derived class, affecting how they can be accessed within the derived class and from outside. Public inheritance is the most common type and preserves the original access level of the base class members in the derived class.

Derivations in C++ provide a powerful mechanism for building class hierarchies, enabling code reuse, and implementing concepts like polymorphism, abstraction, and specialization. They are a fundamental part of object-oriented programming and allow for flexible and extensible design.

## Aggregation:-

Aggregation is a type of association between classes in C++ where one class has a relationship with another class by maintaining a reference or pointer to an object of another class. It represents a "has-a" relationship, where an object of one class "has" or "contains" objects of another class as its members.

In aggregation, the lifetime of the associated objects is independent of the container object. The associated objects can exist outside the scope of the container object, and multiple container objects can share the same associated object.

```
};
```

Aggregation is typically implemented using member variables of the container class that are instances or pointers/references to the associated class. The associated class can be a concrete class or another class hierarchy.

Here's an example of aggregation in C++:

```
class Address {
  std::string street;
  std::string city;
  std::string state;
public:
  Address(const std::string& s, const std::string& c, const std::string& st)
    : street(s), city(c), state(st) {}
};

class Person {
  std::string name;
  int age;
  Address* address;  // Aggregation using a pointer
public:
  Person(const std::string& n, int a, Address* addr)
    : name(n), age(a), address(addr) {}
};
```

In the above example, the `Person` class aggregates the `Address` class using a pointer `address`. Each `Person` object can have an associated `Address` object, but the `Address` object can exist independently outside the `Person` object.

Aggregation allows objects to be composed together to form a more complex structure. It is useful when you want to represent relationships where one object is composed of or contains other objects, but the associated objects have their own lifecycle and can be shared among multiple container objects.

It's important to note that in aggregation, the container object is responsible for managing the associated object's lifetime, including its creation, deletion, and any necessary cleanup.

Aggregation is a form of object composition and is one of the key concepts in object-oriented programming for building complex systems with relationships between objects.

## **Composition vs classification hierarchies:-**

In C++, the terms "composition" and "classification" are not commonly used to describe different mechanisms or concepts related to class hierarchies. It seems there might be some confusion or a misunderstanding of the terminology.

However, I can provide an explanation of the more commonly used terms in the context of class hierarchies: composition and inheritance. Let's discuss these two concepts:

1. Composition:

Composition is a relationship between classes where one class contains an instance of another class as a member variable. It represents a "has-a" relationship, where an object of one class is composed of or contains objects of other classes. Composition implies ownership and responsibility for the lifetime of the composed objects.

In composition, the lifetime of the composed objects is tightly coupled to the lifetime of the container object. When the container object is destroyed, the composed objects are also destroyed. Composition allows creating complex objects by combining simpler objects or components.

Example:

```
class Engine {
 // Engine class implementation
};


class Car {
  Engine engine;  // Composition
 // Other member variables and functions
};
```

In the above example, the `Car` class has a composition relationship with the `Engine` class. Each `Car` object contains an `Engine` object as a member variable.

2. Inheritance:

Inheritance is a mechanism for creating class hierarchies where a derived class inherits properties and behaviors from a base class. It represents an "is-a" relationship, where a derived class is a specialized version of the base class. Inheritance promotes code reuse, polymorphism, and abstraction.

In inheritance, the derived class extends and specializes the functionality of the base class. It inherits the members (data members and member functions) of the base class and can override or add new members. The lifetime of derived objects is independent of the base class.

Example:

```
class Animal {
  // Animal class implementation
};

class Dog : public Animal {
  // Dog class extends Animal class
};
```

In the above example, the `Dog` class inherits from the `Animal` class. The `Dog` class is a specialized version of the `Animal` class and can access and override the members of the `Animal` class.

These are the commonly used concepts in C++ for building class hierarchies. If there are specific concepts or terminology related to "classification hierarchies" you'd like to discuss, please provide more information or clarification.

## Polymorphism:-

Polymorphism is a key concept in object-oriented programming that allows objects of different types to be treated as objects of a common base type. It enables code to be written that can work with objects of multiple derived classes through a common interface.

In C++, polymorphism is primarily achieved through two mechanisms:

1. Virtual Functions:

Polymorphism is commonly implemented using virtual functions. A virtual function is a member function in the base class that is declared with the `virtual` keyword and is intended to be overridden by derived classes. When a derived class overrides a virtual function, it provides its own implementation of that function.

Example:

```
class Shape {
public:
  virtual void draw() {
    // Base class implementation
  }
};

class Circle : public Shape {
public:
  void draw() override {
    // Circle-specific implementation
  }
};

class Square : public Shape {
public:
  void draw() override {
    // Square-specific implementation
  }
};
```

In the above example, the `draw()` function is declared as a virtual function in the base class `Shape`. The derived classes `Circle` and `Square` override the `draw()` function with their own specific implementations.

Polymorphism is achieved when objects of derived classes are accessed through pointers or references of the base class type. The appropriate version of the overridden function is called at runtime based on the actual type of the object.

```
  void drawShape(Shape* shape) {
    shape->draw();  // Calls the appropriate draw() implementation
  }

  int main() {
    Circle circle;
    Square square;
    drawShape(&circle);  // Calls the draw() function of Circle
    drawShape(&square);  // Calls the draw() function of Square

    return 0;
  }
```

2. Abstract Base Classes (Interfaces):

Polymorphism can also be achieved through abstract base classes or interfaces. An abstract base class is a class that declares one or more pure virtual functions. A pure virtual function is declared using the `virtual` keyword and `= 0` syntax, indicating that it has no implementation in the base class and must be overridden by any derived class.

Example:

```
class Shape {
public:
  virtual void draw() = 0;  // Pure virtual function
};

class Circle : public Shape {
public:
  void draw() override {
    // Circle-specific implementation
  }
};

class Square : public Shape {
public:
  void draw() override {
    // Square-specific implementation
  }
};
```

In this example, the `Shape` class is an abstract base class with a pure virtual function `draw()`. The derived classes `Circle` and `Square` override the `draw()` function with their specific implementations.

Abstract base classes cannot be instantiated directly, but pointers or references of the abstract base class type can be used to hold objects of derived classes. This allows polymorphic behavior, where different derived objects can be accessed and manipulated through a common interface.

```
void drawShape(Shape* shape) {
  shape->draw();  // Calls the appropriate draw() implementation
}

int main() {
  Circle circle;
  Square square;
```

```
    drawShape(&circle);  // Calls the draw() function of Circle
    drawShape(&square);  // Calls the draw() function of Square


    return 0;
  }
```

Polymorphism provides a powerful mechanism for writing flexible and reusable code by treating objects of different types in a uniform way. It allows for code extensibility, promotes code modularity, and enables the implementation

## Categorization of polymorphic techniques:-

In C++, there are primarily two categorizations of polymorphic techniques:

1. Compile-Time Polymorphism (Static Polymorphism):

Compile-time polymorphism refers to polymorphic behavior that is resolved during the compilation phase based on the types of objects involved. It is also known as static polymorphism or early binding.

Compile-time polymorphism in C++ is achieved through function overloading and template specialization.

a. Function Overloading:

Function overloading allows multiple functions with the same name but different parameters to be defined. The appropriate function to be called is determined by the types and number of arguments at compile-time.

Example:

```
void print(int value) {
  // Print an integer
}

void print(double value) {
  // Print a double
}

int main() {
  int intValue = 10;
  double doubleValue = 3.14;

  print(intValue);      // Calls the print(int) function
  print(doubleValue);   // Calls the print(double) function

  return 0;
}
```

b. Template Specialization:

Templates in C++ allow writing generic code that can be instantiated for different types at compile-time. Template specialization allows providing specialized implementations for specific types.

Example:

```
template<typename T>
void print(T value) {
  // Generic implementation
}

template<>
void print<int>(int value) {
  // Specialization for int
}

int main() {
  int intValue = 10;
  double doubleValue = 3.14;

  print(intValue);      // Calls the specialized print<int>(int) function
  print(doubleValue);   // Calls the generic print<double>(double) function

  return 0;
}
```

2. Runtime Polymorphism (Dynamic Polymorphism):

Runtime polymorphism refers to polymorphic behavior that is resolved during the runtime based on the actual type of objects. It is also known as dynamic polymorphism or late binding.

Runtime polymorphism in C++ is achieved through virtual functions and inheritance.

a. Virtual Functions and Inheritance:

Virtual functions allow a base class to define a function that can be overridden by derived classes. The appropriate version of the function to be called is determined dynamically at runtime based on the actual type of the object.

Example:

```
class Shape {
public:
  virtual void draw() {
    // Base class implementation
  }
};


class Circle : public Shape {
public:
  void draw() override {
    // Circle-specific implementation
  }
};


class Square : public Shape {
public:
  void draw() override {
    // Square-specific implementation
  }
};


int main() {
  Circle circle;
  Square square;


  Shape* shape1 = &circle;
  Shape* shape2 = &square;


  shape1->draw();   // Calls the draw() function of Circle
  shape2->draw();   // Calls the draw() function of Square


  return 0;
}
```

Runtime polymorphism allows treating objects of different derived classes as objects of the base class, enabling dynamic dispatch of function calls based on the actual object type.

By combining compile-time polymorphism and runtime polymorphism, C++ provides powerful mechanisms to achieve polymorphic behavior, allowing for code reuse, extensibility, and flexibility in handling objects of different types.

## **Method polymorphism:-**

In C++, polymorphism can be achieved through two main methods:

1. Virtual Functions and Inheritance:

Polymorphism through virtual functions and inheritance is also known as runtime polymorphism or dynamic polymorphism. It allows objects of different derived classes to be treated as objects of the base class, enabling the dynamic dispatch of function calls based on the actual object type.

Here's how to achieve polymorphism through virtual functions and inheritance:

a. Define a base class with virtual functions:

```
class Base {
public:
  virtual void polymorphicFunction() {
    // Base class implementation
  }
};
```

b. Override the virtual functions in derived classes:

```
class Derived1 : public Base {
public:
  void polymorphicFunction() override {
    // Derived1 class implementation
  }
};

class Derived2 : public Base {
public:
  void polymorphicFunction() override {
    // Derived2 class implementation
  }
};
```

c. Use pointers or references to the base class to achieve polymorphism:

```
int main() {
  Derived1 derived1;
  Derived2 derived2;

  Base* basePtr1 = &derived1;
  Base* basePtr2 = &derived2;
```

```
    basePtr1->polymorphicFunction();    // Calls the polymorphicFunction() of Derived1
    basePtr2->polymorphicFunction();    // Calls the polymorphicFunction() of Derived2

    return 0;
  }
```

In the example above, the `Base` class has a virtual function `polymorphicFunction()`, which is overridden by the derived classes `Derived1` and `Derived2`. By using pointers or references of the base class type, the appropriate version of the function is called based on the actual object type at runtime.

2. Function Overloading:

Function overloading is a form of compile-time polymorphism that allows multiple functions with the same name but different parameters to be defined. The appropriate function to be called is determined based on the types and number of arguments at compile-time.

Here's an example of achieving polymorphism through function overloading:

```
void polymorphicFunction(Base& obj) {
  // Function implementation for Base objects
}

void polymorphicFunction(Derived1& obj) {
  // Function implementation for Derived1 objects
}


void polymorphicFunction(Derived2& obj) {
  // Function implementation for Derived2 objects
}


int main() {
  Base baseObj;
  Derived1 derived1Obj;
  Derived2 derived2Obj;


  polymorphicFunction(baseObj);       // Calls the function for Base objects
  polymorphicFunction(derived1Obj);   // Calls the function for Derived1 objects
  polymorphicFunction(derived2Obj);   // Calls the function for Derived2 objects


  return 0;
}
```

In the example above, multiple overloaded versions of the `polymorphicFunction()` are defined, each accepting a different type of object. The appropriate function is called based on the static type of the object at compile-time.

Both virtual functions and function overloading provide different forms of polymorphism in C++, allowing for code reuse and flexibility in handling objects of different types. The choice between these methods depends on the specific requirements and design of your application.

**Polymorphism by parameter:-**

Polymorphism by parameter in C++ refers to the ability to pass objects of different types as function arguments, and the function can treat those objects uniformly through a common interface. This allows the function to operate on different types of objects without having to know their specific types.

To achieve polymorphism by parameter in C++, you can use either inheritance and virtual functions or templates.

1. Polymorphism by Inheritance and Virtual Functions:

This approach involves defining a base class with virtual functions and deriving multiple classes from it. The function can accept a parameter of the base class type or a reference/pointer to the base class type. The appropriate version of the virtual function will be called based on the actual type of the object.

Example:

```
class Shape {
public:
  virtual void draw() {
    // Base class implementation
  }
};

class Circle : public Shape {
public:
  void draw() override {
    // Circle-specific implementation
  }
};

class Square : public Shape {
public:
  void draw() override {
    // Square-specific implementation
  }
};

void drawShape(const Shape& shape) {
  shape.draw();   // Calls the appropriate draw() implementation based on the
actual object type
}

int main() {
  Circle circle;
  Square square;

  drawShape(circle);  // Calls the draw() function of Circle
  drawShape(square);  // Calls the draw() function of Square
```

```
    return 0;
  }
```

In this example, the `drawShape()` function accepts a parameter of the base class `Shape`. It can receive objects of derived classes (`Circle` and `Square`) as arguments. The appropriate `draw()` function implementation is called based on the actual object type at runtime.

2. Polymorphism by Templates:

Templates provide a way to achieve polymorphism by parameterizing types. A function or a class can be defined using a template parameter, which can represent different types. When the function is called with different types, the compiler generates specialized versions of the function for each specific type.

Example:

```
  template<typename T>
  void print(const T& value) {
    std::cout << value << std::endl;
  }

  int main() {
    int intValue = 10;
    double doubleValue = 3.14;
    std::string stringValue = "Hello";
    print(intValue);        // Calls the specialized version of print() for int
    print(doubleValue);      // Calls the specialized version of print() for double
    print(stringValue);      // Calls the specialized version of print() for std::string


    return 0;
  }
```

In this example, the `print()` function is a template function that can handle different types (`int`, `double`, `std::string`) as a parameter. The compiler generates specialized versions of the function for each specific type, allowing polymorphic behavior.

Polymorphism by parameter allows for code flexibility and reusability, enabling functions to operate on different types of objects through a common interface. The choice between using inheritance and virtual functions or templates depends on the specific requirements and design of your application.

## **Operator overloading:-**

Operator overloading in C++ allows you to define how an operator should behave when applied to objects of a specific class. It enables you to provide custom implementations for the built-in operators such as arithmetic operators (+, -, *, /), comparison operators (==, !=, <, >, etc.), and more.

Operator overloading is achieved by defining a member function or a non-member function with a special name and syntax corresponding to the operator being overloaded.

Here are a few examples of operator overloading in C++:

1. Overloading Arithmetic Operators:

```
class Vector {
private:
  double x, y;

public:
  Vector(double xVal, double yVal) : x(xVal), y(yVal) {}

  // Overloading addition operator '+'
  Vector operator+(const Vector& other) const {
    return Vector(x + other.x, y + other.y);
  }
};

int main() {
  Vector v1(2.0, 3.0);
  Vector v2(1.0, 2.0);

  Vector result = v1 + v2;  // Calls the overloaded '+' operator

  return 0;
}
```

2. Overloading Comparison Operators:

```
class Fraction {
private:
  int numerator;
  int denominator;

public:
  Fraction(int num, int denom) : numerator(num), denominator(denom) {}

  // Overloading equality operator '=='
  bool operator==(const Fraction& other) const {
    return (numerator == other.numerator) && (denominator == other.denominator);
  }
};

int main() {
  Fraction f1(1, 2);
  Fraction f2(2, 4);

  if (f1 == f2) {  // Calls the overloaded '==' operator
    // Fractions are equal
  }
```

```
    return 0;
  }
```

3. Overloading Stream Insertion and Extraction Operators:

```
  class Point {
  private:
    double x, y;

  public:
    Point(double xVal, double yVal) : x(xVal), y(yVal) {}

    // Overloading stream insertion '<<' operator
    friend std::ostream& operator<<(std::ostream& os, const Point& point) {
      os << "(" << point.x << ", " << point.y << ")";
      return os;
    }
  };

  int main() {
    Point p(2.0, 3.0);

    std::cout << p;  // Calls the overloaded '<<' operator
    return 0;
  }
```

These are just a few examples, and many other operators can be overloaded in C++. It's important to note that not all operators can be overloaded (e.g., `.` and `::`). Additionally, operator overloading should be used judiciously and with care to maintain clarity and avoid unexpected behavior.

## Parametric polymorphism:-

Parametric polymorphism, also known as generic programming, in C++ is achieved through the use of templates. Templates allow you to define generic classes and functions that can operate on different types without the need to specify them explicitly.

Parametric polymorphism allows you to write reusable code that can work with different types, providing flexibility and code efficiency.

Here's an example of parametric polymorphism in C++ using templates:

```
// Generic function to find the maximum of two values
template<typename T>
T maximum(T a, T b) {
  return (a > b) ? a : b;
}

int main() {
  int intResult = maximum(5, 10);            // Calls maximum<int>(5, 10), returns 10
  double doubleResult = maximum(3.14, 2.71);    // Calls maximum<double>(3.14, 2.71), returns 3.14

  return 0;
}
```

In the example above, the `maximum()` function is a template function that takes two arguments of type `T` and returns the maximum value. The type `T` is a placeholder for the actual type that will be determined when the function is called.

By calling `maximum(5, 10)`, the compiler automatically deduces the type `int` for `T`, and `maximum<int>(5, 10)` is invoked, returning the maximum of the two integers.

Similarly, when `maximum(3.14, 2.71)` is called, the type `double` is deduced for `T`, and `maximum<double>(3.14, 2.71)` is invoked, returning the maximum of the two doubles.

Templates provide a powerful mechanism for achieving parametric polymorphism in C++, enabling code reuse and flexibility in working with different types.

## Generic function:-

In C++, generic functions can be implemented using templates. Templates allow you to write functions that can operate on different types without the need to explicitly specify those types. Here's an example of a generic function using templates:

```
// Generic function to swap two values
template<typename T>
void swapValues(T& a, T& b) {
  T temp = a;
  a = b;
  b = temp;
}

int main() {
  int int1 = 5;
  int int2 = 10;
  swapValues(int1, int2);  // Calls swapValues<int>(int1, int2)

  double double1 = 3.14;
  double double2 = 2.71;
  swapValues(double1, double2);  // Calls swapValues<double>(double1, double2)

  return 0;
}
```

In the example above, the `swapValues()` function is a generic function implemented using a template. It takes two arguments of type `T&` (reference to type `T`) and swaps their values. The type `T` is a placeholder for the actual type that will be determined when the function is called.

By calling `swapValues(int1, int2)`, the compiler automatically deduces the type `int` for `T`, and `swapValues<int>(int1, int2)` is invoked, swapping the values of the two integers.

Similarly, when `swapValues(double1, double2)` is called, the type `double` is deduced for `T`, and `swapValues<double>(double1, double2)` is invoked, swapping the values of the two doubles.

Templates allow you to write generic functions that work with multiple types, providing code reusability and flexibility in C++.

## Template function:-

A template function in C++ is a function that is defined using a template parameter, allowing it to operate on multiple types without the need for explicit specialization. Templates provide a way to write generic functions that can handle different types of data.

Here's an example of a template function in C++:

```
template<typename T>

T add(T a, T b) {

  return a + b;

}
```

In this example, `add()` is a template function that takes two arguments of type `T` and returns their sum. The `typename T` specifies a template parameter named `T`, which represents a generic type. When the function is called, the compiler generates a specific version of the function for the actual types used.

Here's how you can use the template function:

```
int main() {
  int result1 = add(5, 10);              // Calls add<int>(5, 10), returns 15
  double result2 = add(3.14, 2.71);      // Calls add<double>(3.14, 2.71), returns 5.85


  return 0;
}
```

In the `main()` function, the `add()` function is called with different types (`int` and `double`). The compiler automatically deduces the template argument `T` based on the provided arguments and generates the appropriate specialized version of the function.

The template function allows you to write generic code that can handle various types without having to write separate functions for each type. It provides code reusability and flexibility in handling different data types in C++.

## **Function name overloading:-**

Function name overloading in C++ refers to the ability to define multiple functions with the same name but different parameter lists. When calling an overloaded function, the compiler determines which version of the function to invoke based on the arguments provided.

Here's an example of function name overloading in C++:

```
#include <iostream>
```

```
// Function to add two integers

int add(int a, int b) {

  return a + b;

}


// Function to add three integers

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

  return a + b + c;

}


// Function to concatenate two strings

std::string add(const std::string& str1, const std::string& str2) {

  return str1 + str2;

}


int main() {

  int result1 = add(5, 10);                  // Calls add(int, int), returns 15

  int result2 = add(3, 6, 9);                // Calls add(int, int, int), returns 18

  std::string result3 = add("Hello, ", "world!"); // Calls add(const std::string&, const
std::string&), returns "Hello, world!"


  std::cout << result1 << std::endl;

  std::cout << result2 << std::endl;
  std::cout << result3 << std::endl;


  return 0;
}
```

In this example, the `add()` function is overloaded three times. The first version takes two integers and returns their sum, the second version takes three integers and returns their sum, and the third version takes two strings and concatenates them.

When calling `add()`, the compiler matches the provided arguments with the appropriate function version based on the number and types of the arguments. In the `main()` function, `add(5, 10)` calls the two-integer version, `add(3, 6, 9)` calls the three-integer version, and `add("Hello, ", "world!")` calls the string concatenation version.

Function name overloading allows you to create more expressive and intuitive interfaces for functions that perform similar operations but with different parameter requirements. It provides flexibility and convenience when working with functions in C++.

## **Overriding inheritance method:-**

Overriding a method in C++ involves providing a new implementation of a base class method in a derived class. It allows the derived class to define its own behavior for the method while still preserving the polymorphic behavior of the base class.

Here's an example that demonstrates method overriding in C++:

```
#include <iostream>

// Base class
class Shape {
public:
  virtual void draw() {
    std::cout << "Drawing a shape." << std::endl;
  }
};

// Derived class
class Circle : public Shape {
public:
  void draw() override {
    std::cout << "Drawing a circle." << std::endl;
  }
};

int main() {
  Shape* shapePtr = new Circle();
  shapePtr->draw();  // Calls the draw() method of the Circle class

  delete shapePtr;
  return 0;
}
```

In this example, the `Shape` class is a base class with a virtual `draw()` method. The `Circle` class is derived from `Shape` and overrides the `draw()` method with its own implementation.

In the `main()` function, we create a pointer `shapePtr` of type `Shape*` that points to an instance of the `Circle` class. When `shapePtr->draw()` is called, the overridden `draw()` method of the `Circle` class is invoked, which outputs "Drawing a circle." instead of the base class implementation.

The `override` keyword is used to indicate explicit method overriding. It helps catch errors at compile-time if the method being overridden does not exist in the base class.

Method overriding allows derived classes to provide specialized implementations of base class methods, enabling polymorphism and runtime determination of the appropriate method to call based on the actual object type.

## Run time polymorphism:-

Runtime polymorphism in C++ is achieved through a combination of virtual functions and pointers/references to base class objects. It allows you to invoke derived class methods through a base class pointer or reference, enabling dynamic binding and late binding.

Here's an example that demonstrates runtime polymorphism in C++:

```
#include <iostream>

// Base class
class Shape {
public:
  virtual void draw() {
    std::cout << "Drawing a shape." << std::endl;
  }
};

// Derived class
class Circle : public Shape {
public:
  void draw() override {
    std::cout << "Drawing a circle." << std::endl;
  }
};

int main() {
  Shape* shapePtr = new Circle();
  shapePtr->draw();  // Calls the overridden draw() method of the Circle class
  delete shapePtr;

  return 0;
}
```

In this example, the `Shape` class is a base class with a virtual `draw()` method. The `Circle` class is derived from `Shape` and overrides the `draw()` method with its own implementation.

In the `main()` function, we create a pointer `shapePtr` of type `Shape*` that points to an instance of the `Circle` class. When `shapePtr->draw()` is called, the overridden `draw()` method of the `Circle` class is invoked due to virtual dispatch. This is runtime polymorphism, as the actual implementation of the `draw()` method is determined at runtime based on the type of the object pointed to by the base class pointer.

By using virtual functions and base class pointers/references, C++ allows dynamic binding and enables polymorphic behavior. This means that the appropriate method implementation is determined at runtime based on the actual type of the object being referred to, rather than the type of the pointer/reference.

# UNIT-4

## Standard C++ classes:-

C++ provides a rich set of standard library classes that are commonly used for various purposes. Here are some of the commonly used standard C++ classes:

1. `std::string`: This class is used for handling strings of characters.

2. `std::vector`: It is a dynamic array that provides a flexible way to store and manipulate collections of objects.

3. `std::list`: It represents a doubly-linked list, allowing efficient insertions and deletions at both ends.

4. `std::map` and `std::unordered_map`: These classes implement associative containers that store key-value pairs. `std::map` is implemented as a balanced binary search tree, while `std::unordered_map` uses a hash table for efficient key-value lookups.

5. `std::set` and `std::unordered_set`: These classes implement containers that store unique values in a sorted order (`std::set`) or an unordered manner (`std::unordered_set`).

6. `std::queue` and `std::stack`: These classes provide specialized container adaptors for implementing queue and stack data structures.

7. `std::ifstream` and `std::ofstream`: These classes are used for reading from and writing to files, respectively.

8. `std::iostream`: It provides input/output operations for both standard input/output and file input/output.

9. `std::chrono`: This class allows manipulation and measurement of time durations.

10. `std::thread` and related classes: These classes are used for multi-threading and concurrent programming.

11. `std::exception` and derived classes: These classes are used for handling exceptions in C++.

12. `std::algorithm`: This class provides a collection of algorithms for performing various operations on containers, such as sorting, searching, and manipulating elements.

These are just a few examples of the standard C++ classes available in the C++ Standard Library. The standard library offers many more classes and templates that cover a wide range of functionalities, making C++ a powerful and versatile programming language.

## Using multiple inheritance:-

In C++, multiple inheritance allows a class to inherit from multiple base classes. This means that a derived class can inherit characteristics and behaviors from multiple parent classes, combining their features into a single class.

Here's an example that demonstrates the usage of multiple inheritance in C++:

```
#include <iostream>

// First base class
class Shape {
public:
  void displayShape() {
    std::cout << "Shape" << std::endl;
  }
};

// Second base class
class Color {
public:
  void displayColor() {
    std::cout << "Color" << std::endl;
  }
};

// Derived class inheriting from both Shape and Color
class Square : public Shape, public Color {
public:
  void displaySquare() {
    std::cout << "Square" << std::endl;
  }
};

int main() {
  Square square;
  square.displayShape();   // Accesses the displayShape() method from the Shape
base class
  square.displayColor();  // Accesses the displayColor() method from the Color base
class
  square.displaySquare(); // Accesses the displaySquare() method from the Square
derived class

  return 0;
}
```

In this example, the `Shape` class represents a shape, and the `Color` class represents a color. The `Square` class is derived from both `Shape` and `Color` using multiple inheritance.

The `Square` class inherits the `displayShape()` method from the `Shape` base class and the `displayColor()` method from the `Color` base class. It also defines its own method `displaySquare()`.

In the `main()` function, an instance of the `Square` class is created, and the inherited and derived methods are called. The `square.displayShape()` and `square.displayColor()` statements access the methods inherited from the base classes, while `square.displaySquare()` accesses the method specific to the `Square` derived class.

Multiple inheritance allows a class to acquire properties and behaviors from multiple base classes, providing flexibility and code reuse. However, it requires careful consideration of potential ambiguities and the design of the class hierarchy to ensure proper usage and avoid conflicts.

## **Persistant objects:-**

In C++, objects are typically stored in memory and exist only for the duration of their lifetime within the program's execution. However, if you want to create persistent objects that can be saved to a storage medium such as a file and loaded back into memory later, you would need to implement persistence mechanisms.

Persistence is the ability to store and retrieve objects and their state from a non-volatile storage medium. It allows objects to survive beyond the execution of a program. There are several ways to implement persistence in C++, such as:

1. Serialization: Serialization involves converting objects into a stream of bytes that can be saved to a file or transferred over a network. The serialized data can then be deserialized to reconstruct the objects later. C++ provides libraries like Boost.Serialization and Protocol Buffers that facilitate serialization and deserialization of objects.

2. Database: Storing objects in a relational or non-relational database allows for persistent storage. You can use database libraries like SQLite, MySQL, or MongoDB to store and retrieve objects from a database.

3. File I/O: You can manually write object data to a file using file input/output operations (`std::ifstream` and `std::ofstream`). You would need to define your own file format and implement the logic to read and write object data from/to the file.

4. Object-relational mapping (ORM) frameworks: ORM frameworks like Hibernate (for C++) or ObjectBox provide higher-level abstractions for persisting objects to databases. They map objects to database tables and handle object persistence transparently.

The choice of persistence mechanism depends on your specific requirements and the complexity of the objects you need to persist. Serialization is a common and flexible approach for persisting objects, while databases offer more advanced querying and indexing capabilities. File I/O provides a basic approach but requires more manual handling of object data.

It's important to note that persistent objects may require additional considerations, such as versioning, data integrity, and data migration when evolving object schemas over time.

## **<u>Streams and files:-</u>**

In C++, streams and file I/O are used to read from and write to files. The `<iostream>` and `<fstream>` header files provide the necessary classes and functions for working with streams and files.

}

Here's an example that demonstrates how to read from and write to a file using streams and file I/O:

```
#include <iostream>
#include <fstream>

int main() {
 std::ofstream outFile("example.txt");  // Create an output file stream

 if (outFile.is_open()) {  // Check if the file was opened successfully
   outFile << "Hello, World!" << std::endl;  // Write data to the file
   outFile.close();  // Close the file
 } else {
   std::cout << "Failed to open the file." << std::endl;
 }

 std::ifstream inFile("example.txt");  // Create an input file stream

 if (inFile.is_open()) {  // Check if the file was opened successfully
   std::string line;
   while (std::getline(inFile, line)) {  // Read data from the file line by line
     std::cout << line << std::endl;  // Display the read data
   }
   inFile.close();  // Close the file
 } else {
   std::cout << "Failed to open the file." << std::endl;
 }

 return 0;
}
```

In this example, an output file stream (`std::ofstream`) is created using the file name "example.txt". The file is opened for writing using the `<<` operator to write the string "Hello, World!" to the file. The file is then closed using the `close()` function.

Next, an input file stream (`std::ifstream`) is created with the same file name to read from the file. The file is opened for reading, and the `std::getline()` function is used to read the file line by line. Each line is then displayed on the console. Finally, the file is closed using the `close()` function.

Streams and file I/O provide a convenient and flexible way to work with files in C++. They allow you to read and write data to files, perform formatted input/output, and handle various data types. You can use different stream classes (`std::ifstream`, `std::ofstream`, `std::fstream`) depending on whether you need input, output, or both.

## Namespaces:-

In C++, namespaces are used to organize and avoid naming conflicts in the code. A namespace is a named scope that contains a set of identifiers (such as variables, functions, classes, etc.) to provide a logical grouping of related entities.

Here's an example that demonstrates the usage of namespaces in C++:

```
#include <iostream>

// Namespace declaration
namespace MyNamespace {
  int x = 5;

  void printX() {
    std::cout << "x = " << x << std::endl;
  }
}

int main() {
  // Accessing namespace members using the scope resolution operator ::
  std::cout << "Global x = " << ::x << std::endl;  // Accesses the global x variable
  MyNamespace::printX();  // Accesses the printX() function in MyNamespace
  std::cout << "Namespace x = " << MyNamespace::x << std::endl;  // Accesses the x
variable in MyNamespace

  return 0;
}
```

In this example, a namespace named `MyNamespace` is defined. It contains an integer variable `x` and a function `printX()` that prints the value of `x`. The `::` scope resolution operator is used to access the global `x` variable and the members of the `MyNamespace` namespace.

In the `main()` function, we access the global `x` using the `::` operator. Then, we call the `printX()` function and access the `x` variable within the `MyNamespace` namespace using the `MyNamespace::` prefix.

Namespaces provide a way to organize code and avoid naming conflicts when multiple libraries or modules are used together. They help create a separate scope for identifiers and make the code more readable and maintainable. They can also be nested, allowing for hierarchical organization of entities.

You can use namespaces from the standard library (e.g., `std`) or define your own custom namespaces to group related code together and avoid naming clashes with other code elements.

## Exception handling:-

Exception handling in C++ allows you to handle exceptional situations or errors that occur during program execution. It provides a mechanism to separate the normal flow of code from the exceptional cases, improving code robustness and maintainability.

In C++, exceptions are thrown using the `throw` keyword and caught using `try` and `catch` blocks. Here's an example that demonstrates exception handling in C++:

```
#include <iostream>


double divide(double dividend, double divisor) {
  if (divisor == 0.0) {
    throw std::runtime_error("Division by zero!");
  }
  return dividend / divisor;
}

int main() {
  double result;
  double dividend = 10.0;
  double divisor = 0.0;

  try {
    result = divide(dividend, divisor);
    std::cout << "Result: " << result << std::endl;
  } catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
  }

  return 0;
}
```

In this example, the `divide()` function is defined to perform division between two doubles. If the divisor is zero, it throws a `std::runtime_error` exception with a custom error message.

In the `main()` function, we call `divide()` with a divisor of 0.0 inside a `try` block. If an exception is thrown, it is caught in the `catch` block. The `catch` block takes a reference to a `std::exception` object, allowing us to access the exception's information via the `what()` member function. The error message is printed to the standard error stream (`std::cerr`).

By using exception handling, you can gracefully handle exceptional cases and take appropriate actions, such as displaying an error message, logging, or performing recovery operations. The `catch` block can be used to catch specific types of exceptions or catch all exceptions using `catch (...)` to handle unexpected errors.

C++ provides a range of standard exception classes (e.g., `std::runtime_error`, `std::logic_error`) that can be used or you can create custom exception classes by deriving from `std::exception` or its subclasses.

Remember to handle exceptions appropriately and ensure proper resource management, such as cleaning up allocated memory or releasing acquired resources, in `catch` blocks or with the help of RAII (Resource Acquisition Is Initialization) techniques.

## Generic classes:-

In C++, a generic class is a class that can work with different data types without having to be rewritten for each specific type. It allows you to define classes that are parameterized by one or more types, providing flexibility and code reuse.

C++ achieves generic programming through templates. Templates are used to define generic classes, functions, or even variables. When a template is instantiated with a specific type or set of types, it generates the necessary code for that specific type.

```
}
```

Here's an example of a generic class in C++:

```
template <typename T>
class Stack {
private:
   T* elements;  // Dynamic array to store elements
   int size;     // Current size of the stack
   int capacity; // Maximum capacity of the stack

public:
   Stack(int capacity) : size(0), capacity(capacity) {
      elements = new T[capacity];
   }

   ~Stack() {
      delete[] elements;
   }

   void push(const T& value) {
      if (size < capacity) {
         elements[size++] = value;
      }
   }
```

```
    T pop() {
        if (size > 0) {
            return elements[--size];
        } else {
            // Handle underflow condition
            throw std::runtime_error("Stack is empty.");
        }
    }

    int getSize() const {
        return size;
    }
};
```

In this example, the `Stack` class is a generic class that can work with any data type `T`. It has a constructor, destructor, push, pop, and getSize methods. The type `T` is used throughout the class to define the type of the dynamic array `elements` and the type of the value being pushed or popped.

To use this generic class with a specific type, you instantiate it with the desired type:

```
Stack<int> intStack(10); // Instantiate a stack of integers
```

```
intStack.push(5);

intStack.push(10);

intStack.push(15);


std::cout << "Size of intStack: " << intStack.getSize() << std::endl;


int poppedValue = intStack.pop();

std::cout << "Popped value: " << poppedValue << std::endl;
```

In this example, we instantiated the `Stack` class with the `int` type and used it to create an integer stack. The class behaves as if it were specifically designed for integers, but the same generic class can be instantiated with other types, such as `double`, `std::string`, or even custom types.

## Standard template library:-

The Standard Template Library (STL) is a powerful library in C++ that provides a collection of generic algorithms and data structures. It is part of the C++ Standard Library and is included with all compliant C++ compilers.

The STL consists of several components, including:

1. **Containers**: Containers are data structures that store objects or values. The STL provides various container classes, such as `vector`, `list`, `deque`, `set`, `map`, and more. These containers offer different characteristics in terms of storage, access, and insertion/deletion efficiency.

2. **Algorithms**: The STL provides a rich set of algorithms that can be used on containers or other data structures. Algorithms include operations like sorting, searching, modifying, and iterating over elements. Some commonly used algorithms are `sort`, `find`, `transform`, `accumulate`, and many more.

3. **Iterators**: Iterators provide a way to access and traverse elements in a container. They act as generalized pointers that can be used to move through the elements of a container, regardless of the specific container type. Iterators allow algorithms to be applied uniformly to different containers.

4. **Function objects**: Function objects, also known as functors, are objects that act like functions. They can be used with algorithms to customize their behavior. Functors are often used for sorting, searching, and other operations that require custom comparisons or transformations.

5. **Allocators**: Allocators provide a way to manage the memory allocation and deallocation of objects stored in containers. They allow customization of memory management strategies and provide an interface for allocating and releasing memory.

The STL components work together to provide a powerful framework for generic programming in C++. You can use the containers to store data, apply algorithms to manipulate or analyze the data, and utilize iterators to access the elements. The STL promotes code reuse, efficiency, and flexibility by providing a consistent and well-designed set of tools for common programming tasks.

Here's a simple example that demonstrates the usage of the STL:

```
#include <iostream>

#include <vector>
```

```
#include <algorithm>

int main() {
    std::vector<int> numbers = {5, 2, 8, 3, 1};

    // Sort the numbers
    std::sort(numbers.begin(), numbers.end());

    // Print the sorted numbers
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Find the minimum and maximum element
    int minNum = *std::min_element(numbers.begin(), numbers.end());
    int maxNum = *std::max_element(numbers.begin(), numbers.end());

    std::cout << "Min: " << minNum << std::endl;
    std::cout << "Max: " << maxNum << std::endl;

    return 0;
}
```

In this example, we include the necessary headers (`<iostream>`, `<vector>`, `<algorithm>`) to use the STL components. We create a `vector` of integers, populate it with values, and then use the `sort` algorithm to sort the numbers in ascending order. Finally, we use `min_element` and `max_element` algorithms to find the minimum and maximum values in the vector.

## Library organization and containers:-

In C++, libraries are organized into headers and implementation files. The headers contain declarations of classes, functions, and other entities, while the implementation files (.cpp files) contain the definitions of those entities. This separation allows for modular code organization and helps with compilation efficiency.

When it comes to containers in C++, they are part of the Standard Template Library (STL), which is a collection of generic algorithms and data structures provided by the C++ Standard Library. Containers in the STL provide different ways to store and organize data.

Here are some commonly used containers in C++:

1. **Vector**: `std::vector` is a dynamic array that can resize itself automatically as elements are added or removed. It provides random access to elements and is efficient for appending and accessing elements at the end. Insertions and deletions at arbitrary positions are slower compared to the end.

2. **List**: `std::list` is a doubly-linked list. It allows efficient insertions and deletions at any position, but random access to elements is slower compared to vectors and arrays.

3. **Deque**: `std::deque` (short for double-ended queue) is similar to a vector but allows efficient insertions and deletions at both ends. It provides random access to elements as well.

4. **Queue**: `std::queue` is a container adapter that provides a FIFO (First-In-First-Out) data structure. It is typically implemented using a deque or a list.

5. **Stack**: `std::stack` is a container adapter that provides a LIFO (Last-In-First-Out) data structure. It is typically implemented using a deque or a list.

6. **Set**: `std::set` is an ordered collection of unique elements. It automatically maintains the order of elements and allows fast searching, insertion, and deletion operations.

7. **Map**: `std::map` is an associative container that stores key-value pairs. It allows efficient searching, insertion, and deletion based on the keys. The keys are unique within the map.

8. **Unordered Set**: `std::unordered_set` is an unordered collection of unique elements. It provides fast average constant-time insertion, deletion, and searching operations.

9. **Unordered Map**: `std::unordered_map` is an unordered associative container that stores key-value pairs. It provides fast average constant-time insertion, deletion, and searching operations based on the keys.

These are just a few examples of containers available in the C++ Standard Library. Each container has its own characteristics in terms of performance, memory usage, and operations supported. The choice of container depends on the specific requirements of your program, such as the type of operations needed, the expected size of the data, and the performance constraints.

To use these containers, you need to include the appropriate header file (e.g., `<vector>`, `<list>`, `<queue>`) and then create objects of the corresponding container class. You can then use the member functions provided by the container class to manipulate and access the data stored within the container.

For example, to use a vector, you would include the `<vector>` header and create a vector object like this:

```
#include <vector>


int main() {
   std::vector<int> numbers;  // Create an empty vector of integers


   numbers.push_back(5);  // Add elements to the vector
   numbers.push_back(2);
   numbers.push_back(8);


   int element = numbers[1];  // Access an element using the subscript operator


   return 0;
}
```

In this example, we create a vector of integers, add elements to it using the `push_back` function, and access an element using the subscript operator `[ ]`.

Remember to choose the appropriate container

## Standard containers:-

In C++, the Standard Library provides a set of containers that are part of the Standard Template Library (STL). These containers are generic data structures designed to store and manipulate collections of objects. Here are the commonly used standard containers in C++:

1. **Vector**: `std::vector` is a dynamic array that provides contiguous memory storage. It allows efficient random access, appending and removal of elements at the end, and dynamic resizing.

2. **List**: `std::list` is a doubly-linked list that allows efficient insertion and deletion at any position. It does not provide random access to elements, but it excels at constant-time insertions and deletions.

3. **Deque**: `std::deque` (short for double-ended queue) is similar to a vector but allows efficient insertions and deletions at both ends. It also provides random access to elements.

4. **Queue**: `std::queue` is an adapter class that provides a queue data structure, implementing a First-In-First-Out (FIFO) policy. It is typically implemented using a deque or a list.

5. **Stack**: `std::stack` is an adapter class that provides a stack data structure, implementing a Last-In-First-Out (LIFO) policy. It is typically implemented using a deque or a list.

6. **Set**: `std::set` is an ordered collection of unique elements. It automatically maintains the order of elements and provides efficient searching, insertion, and deletion operations. It is implemented using a balanced binary search tree.

7. **Map**: `std::map` is an associative container that stores key-value pairs. It automatically maintains the order of keys and provides efficient searching, insertion, and deletion based on the keys. It is implemented using a balanced binary search tree.

8. **Unordered Set**: `std::unordered_set` is an unordered collection of unique elements. It provides fast average constant-time insertion, deletion, and searching operations using hash-based containers.

9. **Unordered Map**: `std::unordered_map` is an unordered associative container that stores key-value pairs. It provides fast average constant-time insertion, deletion, and searching operations based on the keys. It is implemented using hash-based containers.

These standard containers are defined in different header files within the C++ Standard Library. To use a particular container, you need to include the corresponding header file. For example, to use the vector container, you include the `<vector>` header:

```
#include <vector>


int main() {
   std::vector<int> numbers;  // Create a vector of integers
```

```
    numbers.push_back(5);  // Add elements to the vector
    numbers.push_back(2);
    numbers.push_back(8);

    int element = numbers[1];  // Access an element using the subscript operator

    return 0;
}
```

In this example, we include the `<vector>` header, create a vector of integers, add elements to it using the `push_back` function, and access an element using the subscript operator `[ ]`.

## <u>Algorithm and function objects</u>:-

In C++, the Standard Template Library (STL) provides a powerful set of algorithms and function objects (also known as functors) that can be used with containers or other data structures. These algorithms and function objects are part of the `<algorithm>` header and allow you to perform various operations on collections of data.

Algorithms:

STL algorithms provide a wide range of operations such as sorting, searching, modifying, counting, and more. These algorithms are designed to work with iterators and can be used with different containers or ranges of data. Here are some commonly used algorithms:

1. **Sorting**: Algorithms like `sort`, `partial_sort`, `stable_sort`, and `nth_element` are used to sort elements in ascending or descending order.

2. **Searching**: Algorithms like `find`, `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` are used to search for elements in a sorted range.

3. **Modifying**: Algorithms like `copy`, `transform`, `replace`, `remove`, and `fill` are used to modify elements in a range or container.

4. **Numeric algorithms**: Algorithms like `accumulate`, `inner_product`, `partial_sum`, and `adjacent_difference` perform numeric operations on elements, such as summing, multiplying, calculating differences, and more.

5. **Partitioning**: Algorithms like `partition`, `stable_partition`, `is_partitioned`, and `partition_point` separate a range of elements based on a given predicate.

Function Objects (Functors):

Function objects in C++ are objects that act like functions. They can be used with algorithms to customize their behavior or provide specialized operations. Functors are created by defining a class or struct that overloads the function call operator `operator()`. Here are some use cases for function objects:

1. **Comparison**: You can create custom comparison functors to define custom sorting orders or to specify custom equality criteria for algorithms like `sort`, `binary_search`, or `find`.

2. **Transformations**: Functors can be used with algorithms like `transform` to apply custom transformations to elements during processing.

3. **Predicates**: Functors can be used as predicates to determine whether a certain condition is true or false, such as filtering elements or checking for specific properties.

Here's an example that demonstrates the usage of an algorithm and a custom function object:

```
#include <iostream>

#include <vector>

#include <algorithm>


// Custom function object

struct MultiplyByTwo {

    int operator()(int value) const {

        return value * 2;

    }

};


int main() {

    std::vector<int> numbers = {5, 2, 8, 3, 1};


    // Sort the numbers in ascending order

    std::sort(numbers.begin(), numbers.end());
    // Multiply each element by two using transform and custom function object
    std::transform(numbers.begin(), numbers.end(), numbers.begin(), MultiplyByTwo());


    // Print the modified numbers
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;


    return 0;
}
```

In this example, we include the `<iostream>`, `<vector>`, and `<algorithm>` headers. We create a vector of integers, sort the numbers using the `sort` algorithm, and then use the `transform` algorithm along with the custom function object `MultiplyByTwo` to multiply each element by two. Finally, we print the modified numbers.

The STL algorithms and function objects provide a powerful and flexible way to perform operations on containers and ranges of data in a generic and reusable manner.

## Iterators and allocators:-

In C++, iterators and allocators are additional components provided by the Standard Template Library (STL) that work in conjunction with containers and algorithms.

1. **Iterators**:

Iterators provide a way to traverse and access elements within a container or a range of data. They act as generalized pointers and allow algorithms to operate on different containers uniformly. Iterators abstract the concept of iteration and provide a consistent interface for accessing elements.

There are several types of iterators in C++, including:

- **Input Iterators**: These iterators allow reading elements from a container in a forward-only manner. They support operations like dereferencing (`*it`), incrementing (`++it`), and comparing for equality (`it == end`).

- **Output Iterators**: These iterators allow writing elements to a container in a forward-only manner. They support operations like dereferencing and incrementing, but they may have limited functionality compared to other iterators.

- **Forward Iterators**: These iterators combine the functionality of both input and output iterators, allowing reading and writing elements in a forward-only manner.

- **Bidirectional Iterators**: These iterators support operations in both forward and backward directions. They can be incremented and decremented.

- **Random Access Iterators**: These iterators provide full functionality for random access, allowing efficient element access, arithmetic operations (`it + n`, `it - n`), and comparison operations (`it1 < it2`).

Iterators are commonly used with algorithms to iterate over elements in a container, perform operations, and access or modify the data. They provide a level of abstraction and enable generic programming.

2. **Allocators**:

Allocators are used to manage the memory allocation and deallocation of objects within containers. They provide an interface for allocating, constructing, and releasing memory for elements stored in containers. Allocators abstract the process of memory management and allow customization of memory allocation strategies.

Containers in the STL are parameterized by an allocator type. By default, they use a specific allocator (`std::allocator`) that provides a basic memory allocation mechanism using `new` and `delete` operators. However, custom allocator types can be provided to containers for specialized memory management needs.

Allocators define member functions such as `allocate`, `deallocate`, `construct`, and `destroy` to manage memory. The allocator is responsible for allocating memory for elements, constructing objects within that memory, releasing memory when objects are destroyed, and deallocating memory.

Using allocators allows you to control the memory allocation behavior of containers and adapt them to specific requirements, such as using custom memory pools or specialized memory management techniques.

While iterators and allocators are advanced features of the STL, they provide additional flexibility and customization options when working with containers and algorithms. They enable efficient iteration and memory management, making the STL a powerful library for generic programming in C++.

## **Strings:-**

In C++, strings are a sequence of characters stored as objects. They are part of the Standard Template Library (STL) and provide a convenient way to work with text and manipulate strings of characters. Here are some key points about strings in C++:

1. **Definition and Initialization**:

   - Strings are represented by the `std::string` class in C++.

   - To use strings, you need to include the `<string>` header.

   - Strings can be initialized using literals, other strings, or character arrays.

2. **Operations on Strings**:

   - Concatenation: Strings can be concatenated using the `+` operator or the `append()` function.

   - Accessing Characters: Individual characters in a string can be accessed using the subscript operator `[ ]` or the `at()` member function.

   - Length: The `length()` or `size()` member functions return the length of a string.

   - Substring: The `substr()` function allows extracting a portion of a string.

   - Comparison: Strings can be compared using operators like `==`, `!=`, `<`, `>`, etc.

   - Searching and Modifying: Functions like `find()`, `replace()`, and `erase()` provide search and modification capabilities.

   - Conversion: Strings can be converted to C-style character arrays using the `c_str()` member function.

3. **Input and Output**:

   - Strings can be read from and written to standard input/output streams using the `>>` and `<<` operators, respectively.

   - They can also be formatted using stream manipulators and formatting options.

4. **Dynamic Memory Allocation**:

- Strings manage their memory dynamically, so you don't need to worry about manually allocating or deallocating memory.

- The `std::string` class handles memory management internally.

5. **String Manipulation Efficiency**:

- Strings provide efficient concatenation and appending operations.

- However, due to the dynamic nature of strings, some operations like inserting or deleting characters in the middle may have a higher time complexity compared to fixed-size character arrays.

Strings in C++ provide a higher level of abstraction compared to character arrays, making them more convenient and flexible for working with textual data. They offer a wide range of operations, including concatenation, searching, modification, and input/output functionalities. The `std::string` class handles memory management, simplifying string manipulation and ensuring proper memory allocation.

## Streams:-

In C++, streams are a fundamental part of the input/output (I/O) system provided by the Standard Library. Streams are a powerful mechanism for reading input from and writing output to different sources, such as the console, files, or other devices. They provide a consistent and flexible way to handle I/O operations in a generic manner. Here are some key points about streams in C++:

1. **Stream Classes**:

- Streams are implemented as classes in C++. The primary stream classes are `std::istream` for input (e.g., `std::cin`) and `std::ostream` for output (e.g., `std::cout`).

  - Derived classes like `std::ifstream` and `std::ofstream` are used for file-based I/O operations, and `std::stringstream` is used for working with strings as streams.

2. **I/O Operators**:

  - Streams provide insertion (`<<`) and extraction (`>>`) operators to perform formatted input and output operations.

  - The insertion operator (`<<`) is used to write data to an output stream, and the extraction operator (`>>`) is used to read data from an input stream.

3. **Formatting**:

  - Streams support formatting options to control the appearance and interpretation of data during I/O operations.

  - Formatting options include precision, width, fill characters, alignment, and more.

  - Formatting manipulators like `std::setw`, `std::setprecision`, `std::setfill`, and others can be used to modify formatting settings.

4. **Stream State and Error Handling**:

  - Streams maintain an internal state to indicate the status of the stream.

  - The state can be checked using member functions like `good()`, `fail()`, `eof()`, and `bad()`.

  - Stream errors and exceptions can be handled using error flags and exception handling mechanisms.

5. **Buffering**:

  - Streams use a buffer to improve I/O performance.

- Output is typically buffered, meaning data is accumulated in the buffer before being written to the destination, which can be flushed explicitly or automatically.

- Input can also be buffered to improve performance.

6. **Manipulators**:

- Manipulators are functions or objects used to modify stream behavior.

- Manipulators can be used to control formatting, change stream state, set flags, or perform other operations.

- Examples include `std::endl`, `std::setw`, `std::hex`, `std::fixed`, and more.

7. **File-based I/O**:

- Streams support file-based I/O operations through classes like `std::ifstream` and `std::ofstream`.

- File streams are used to read from and write to files on disk.

- File streams can be opened, closed, and manipulated using member functions.

8. **String-based I/O**:

- Streams can also be used to perform I/O operations on strings using `std::stringstream`.

- String streams provide a way to treat strings as input and output sources, enabling string manipulation with stream operations.

Streams in C++ provide a flexible and unified way to handle input and output operations. They can be used to read from and write to different sources, including the console, files, and strings. With formatting options, stream state management, and buffering mechanisms, streams offer powerful and customizable I/O capabilities in C++.

## Manipulators:-

In C++, manipulators are special functions or objects that modify the behavior of streams during input/output (I/O) operations. They provide a flexible and convenient way to control formatting, change stream state, set flags, or perform other operations on streams. Manipulators can be used with input streams (`std::istream`) and output streams (`std::ostream`) to customize the I/O behavior. Here are some key points about manipulators in C++:

1. **Manipulator Functions**:

   - Manipulator functions are predefined functions that modify the behavior of streams.

   - Manipulator functions are typically implemented as non-member functions that take a stream as an argument and return the modified stream.

2. **Manipulator Objects**:

   - Manipulator objects are objects of user-defined or library-defined classes that modify stream behavior.

   - Manipulator objects are typically created by instantiating a class and using them as arguments to the stream manipulation expressions.

3. **Formatting Manipulators**:

   - Formatting manipulators are used to modify the formatting options of streams.

   - Examples of formatting manipulators include:

     - `std::setw(int n)`: Sets the width of the next output field to `n` characters.

    - `std::setprecision(int n)`: Sets the decimal precision to `n` digits.

    - `std::setfill(char c)`: Sets the fill character to `c`.

    - `std::fixed`, `std::scientific`: Controls the format of floating-point numbers.

4. **Stream State Manipulators**:

   - Stream state manipulators are used to modify the state flags of streams.

   - Examples of stream state manipulators include:

     - `std::endl`: Inserts a newline character into the stream and flushes the stream.

     - `std::flush`: Flushes the output buffer without inserting a newline character.

     - `std::skipws`, `std::noskipws`: Controls whitespace skipping behavior during input operations.

     - `std::boolalpha`, `std::noboolalpha`: Controls the formatting of boolean values as `true` or `false`.

5. **Custom Manipulators**:

   - Users can define their own manipulators by creating functions or objects that modify stream behavior.

   - Custom manipulators can encapsulate specific formatting or state modifications and provide reusable functionality.

6. **Applying Manipulators**:

   - Manipulators can be applied to streams using the insertion (`<<`) or extraction (`>>`) operators.

   - Manipulators can be chained together, and multiple manipulators can be applied in a single expression.

Manipulators in C++ provide a powerful way to control the behavior of streams during I/O operations. They allow customization of formatting, state flags, and other aspects of stream handling. By using predefined manipulators or creating custom manipulators, programmers can easily tailor the I/O behavior to their specific needs. Manipulators play an essential role in stream-based input and output operations in C++.

## User defined manipulators:-

In C++, user-defined manipulators are functions or objects created by programmers to modify the behavior of streams during input/output (I/O) operations. They provide a way to encapsulate specific formatting or state modifications and enable custom functionality for stream handling. Here are some key points about user-defined manipulators in C++:

1. **Function-Based User-Defined Manipulators**:

   - User-defined manipulators can be implemented as functions that modify stream behavior.

   - The function should typically take a stream as an argument and return the modified stream.

   - The function can perform specific operations on the stream, such as changing formatting options, modifying state flags, or applying custom logic.

   - The function can be called using the insertion (`<<`) or extraction (`>>`) operators on a stream.

Example of a function-based user-defined manipulator:

```
#include <iostream>
```

```
// User-defined manipulator function
std::ostream& myManipulator(std::ostream& os) {
   // Perform custom operations on the stream
   os << "Custom Manipulator Called ";
   os << "(Additional Output)";

   return os;
}

int main() {
   // Applying the user-defined manipulator to the output stream
   std::cout << "Hello, World!" << myManipulator << std::endl;

   return 0;
}
```

2. **Object-Based User-Defined Manipulators**:

  - User-defined manipulators can also be implemented as objects of user-defined or library-defined classes.

  - The class should overload the function call operator (`operator()`) to modify stream behavior.

  - The object can be instantiated and used as an argument to the stream manipulation expression.

  - The function call operator of the object should take a stream as an argument and modify it accordingly.

- The object can store additional state or configuration information for custom manipulations.

Example of an object-based user-defined manipulator:

```
#include <iostream>

// User-defined manipulator class
class MyManipulator {
public:
   // Overloading the function call operator
   std::ostream& operator()(std::ostream& os) const {
      // Perform custom operations on the stream
      os << "Custom Manipulator Called ";
      os << "(Additional Output)";

      return os;
   }
};

int main() {
   // Applying the user-defined manipulator object to the output stream
   std::cout << "Hello, World!" << MyManipulator() << std::endl;

   return 0;
}
```

User-defined manipulators allow programmers to extend the functionality of streams in C++. They provide a means to encapsulate custom formatting, state modifications, or other operations specific to the application's requirements. By creating user-defined manipulators, programmers can achieve greater control and flexibility in stream-based I/O operations.

## Vectors:-

In C++, vectors are a dynamic array-like data structure provided by the Standard Template Library (STL). They are part of the container classes in C++ and offer a convenient way to store and manipulate a sequence of elements. Here are some key points about vectors in C++:

1. **Definition and Initialization**:

   - Vectors are represented by the `std::vector` class in C++.

   - To use vectors, you need to include the `<vector>` header.

   - Vectors can store elements of any type, including built-in types and user-defined types.

   - Vectors can be initialized using various methods, such as assignment, initializer lists, or by specifying the size and initial values.

2. **Dynamic Size**:

   - Vectors automatically manage their size, allowing for dynamic resizing as elements are added or removed.

   - Elements can be added to a vector using the `push_back()` member function, which appends the element to the end of the vector.

   - Vectors can also be resized explicitly using the `resize()` member function.

3. **Random Access**:

   - Vectors support random access to elements, meaning you can directly access elements at any position using the subscript operator `[]`.

   - Elements can be accessed, modified, or assigned using the subscript operator.

4. **Container Operations**:

   - Vectors provide various member functions for common container operations, such as inserting elements, erasing elements, and clearing the vector.

   - Common member functions include `insert()`, `erase()`, `clear()`, `size()`, `empty()`, and more.

5. **Efficient Element Access**:

   - Vectors offer efficient element access due to their underlying contiguous memory layout.

   - Iterators can be used to traverse the vector and access elements using algorithms or in range-based loops.

Vectors in C++ provide a flexible and efficient way to store and manipulate collections of elements. They offer dynamic resizing, random access, and a rich set of member functions for common container operations. Vectors are commonly used when the size of the collection may change dynamically and efficient element access is required.

## <u>**Slice:-**</u>

In C++, the term "slice" typically refers to the operation of extracting a portion of a container or array. It involves selecting a range of elements from a sequence and creating a new container or view that represents that subset. The concept of slicing is not directly supported by the Standard Library in C++, but it can be achieved using various techniques. Here are a few approaches to achieve slicing in C++:

1. **Subsequence using Iterators**:

  - Slicing can be accomplished by using iterators to specify the beginning and ending positions of the desired range.

  - You can create a new container or perform operations on the selected subset of elements using the iterator range.

  - For example, to slice a vector `myVector` from index 2 to index 5:

```
    std::vector<int> slicedVector(myVector.begin() + 2, myVector.begin() + 6);
```

2. **std::vector::erase**:

  - The `std::vector::erase` function can be used to remove elements from a vector and effectively create a slice.

  - By specifying the iterator range to be erased, you can remove elements and modify the vector in-place.

  - For example, to slice a vector `myVector` from index 2 to index 5:

```
    myVector.erase(myVector.begin() + 2, myVector.begin() + 6);
```

3. **std::valarray**:

- The `std::valarray` class in the Standard Library provides an alternative to slicing for mathematical operations on arrays.

- `std::valarray` supports mathematical operations and slicing-like functionality, such as selecting a subset of elements based on conditions or indices.

- For example, to slice a `std::valarray<int>` called `myValArray` from index 2 to index 5:

```
std::valarray<int> slicedValArray = myValArray[std::slice(2, 3, 1)];
```

It's important to note that the approaches mentioned above are just examples and may vary depending on the specific requirements and container types used in your code. Slicing in C++ generally involves using iterators or specific member functions of containers to extract a subset of elements from a sequence or modify the original container accordingly.

## **Generalized numeric algorithm:-**

A generalized numeric algorithm in C++ refers to a generic algorithm that operates on numeric data types, such as integers, floating-point numbers, or custom numeric types. These algorithms are designed to work with different types of numeric data, providing a flexible and reusable solution.

To implement a generalized numeric algorithm in C++, you can use templates. Templates allow you to write generic code that can work with different data types. Here's an example of a simple generalized numeric algorithm that calculates the sum of a range of numbers:

```
#include <iostream>

#include <numeric>
```

```
#include <vector>


template<typename T>

T calculateSum(const std::vector<T>& numbers) {

   T sum = std::accumulate(numbers.begin(), numbers.end(), static_cast<T>(0));

   return sum;

}


int main() {

   std::vector<int> intNumbers = {1, 2, 3, 4, 5};

   int intSum = calculateSum(intNumbers);

   std::cout << "Sum of integers: " << intSum << std::endl;


   std::vector<double> doubleNumbers = {1.1, 2.2, 3.3, 4.4, 5.5};

   double doubleSum = calculateSum(doubleNumbers);

   std::cout << "Sum of doubles: " << doubleSum << std::endl;


   return 0;

}
```

In the example above, the `calculateSum` function is a template function that takes a vector of type `T` (where `T` is a placeholder for the numeric type). The function uses the `std::accumulate` algorithm from the `<numeric>` library to calculate the sum of the numbers in the vector. The `static_cast<T>(0)` is used to provide the initial value for the sum, ensuring it matches the type of the numeric data.

In the `main` function, we demonstrate the usage of the `calculateSum` function with both integer and double vectors. The function is instantiated with the appropriate types based on the argument passed, allowing it to work correctly with different numeric types.

You can expand upon this concept to create more complex generalized numeric algorithms that operate on various numeric data types.