Introduction To OOPs, Exception handling and STL in C++ By {Coding Ninjas}

Object-Oriented programming or OOPs refers to the language that uses the concept of class and object in programming. The main objective of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc. The popular object-oriented programming languages are c++, java, python, PHP, c#, etc.

Why do we use object-oriented programming?

- To make the development and maintenance of projects more manageable.
- To provide the feature of data hiding that is good for security concerns.
- We can provide the solution to real-world problems if we are using object-oriented programming.

OOPs Concepts: Let's look at the topics which we are going to discuss.

- Access modifiers
- Class
- Object
- Encapsulation
- Abstraction
- Polymorphism
- Inheritance
- Constructor
- Destructor

Previous
Next
Access Modifiers

Access Modifiers

Access Modifiers in a class are used to assign access to the class members. It sets some restrictions on the class members not to get directly accessed by the outside functions.

It allows us to determine which class members are accessible to other classes and functions and which are not.
There are three types of access modifiers available in C++:

- Public
- Private
- Protected

i) Public: All the class members with public modifier can be accessed from anywhere(inside and outside the class).

ii) Private: All the class members with private modifier can only be accessed by the member function inside the class.

iii) Protected: The access level of a protected modifier is within the class and outside the class through child class. If you do not make the child class, it cannot be accessed from outside the class.

Note: If we do not specify any access modifiers for the members inside the class, then by default, the access modifier for the members will be Private.

Example:

```
class demo {
    private:
        int a;
        string c;
    public:
        void func1() {
            // code
        }
    void func2() {
        // code
    }
};
```

Here, the variables ‘a’ and ‘b’ of the demo class is hidden using the private keyword, while the member functions are made accessible using the public keyword.

Previous
Next
Class, Object and Class vs Object

Class

A class is a logical entity that is used to define a new data type. Once we define this new data type can be used to create objects of that type. Thus we can say that a class is a template for an object. A class contains variables, methods, and constructors

Syntax to define a class:

```
class class_Name {
   // class body
}
```

Here,

- **class: class keyword is used to create a class in C++.**
- **class_Name: The name of the class.**

- **Class body: Curly braces surround the class body.**

Object

The object is an entity that has a state and behavior. To access the members defined inside the class, we need to create the object of that class. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

For example, A dog is an object because it has states like color, name, breed, etc., as well as behavior like breaking, eating, sleeping, etc.
Only the object's specification is defined when a class is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax to create an object in C++:

```
Classname objectname;
```

Syntax to create an object dynamically in C++:

```
Classname * objectname = new Classname();
```

Here, the class's default constructor is called, and it dynamically allocates memory for one object of the class. The address of the memory allocated is assigned to the pointer, i.e., object name.

Classes vs Objects

Classes are used to create user-defined data structures. Classes define functions called **methods**, identifying the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.
A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for determining a **Car**, but it doesn't have the name or top-speed of any specific **Car**.

While the class is the blueprint, the object is built from a class and contains real data. The object of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or a questionnaire. An **object** is like a form that has been filled out with information. Like many people can fill out the same form with their unique information, many objects can be created from a single class.

Example: Creating the Care class.

```
class Car{
    String name;
    int topSpeed;

    void func(String carName, int speed){
        name = carName;
        topSpeed = speed;
```

```
    }
};
```

Example: Let’s look at how we can use class and object in our program.

```
#include <iostream>
using namespace std;
class Box {
    public:
        double width;
    double height;
    double depth;
};
int main() {
    // creating an object of Box class
    Box obj;
    //assign values to obj instance variable
    obj.width = 5;
    obj.height = 10;
    obj.depth = 15;
    // compute the volume of the box
    double volume = obj.width * obj.height * obj.depth;
    // printing the volume
    cout << "Volume of a box: " << volume;
}
Output:
Volume of a box: 750
```

Previous

Next

Getter and Setter

Getter and Setter

The class’s private members are not accessible outside the class, although sometimes there is a necessity to provide access even to private members; in these cases, we need to create functions called getters and setters. Getters are those functions that allow us to access data members of the object. However, these functions do not change the value of data members. These are also called accessor functions.

Setters are the member functions that allow us to change the data members of an object. These are also called mutator functions.

Example:

```
#include <iostream>
using namespace std;
class Student {
    int rollno;
    char name[20];
    float marks;
    char grade;
    public:
    int getRollno() {
        return rollno;
    }
    int getMarks() {
        return marks;
    }
    void setGrade() {
        if (marks > 90) grade = 'A';;
        else if (marks > 80) grade = 'B';
        else if (marks > 70) grade = 'C';
        else if (marks > 60) grade = 'D';
        else grade = 'E';
    }
};
```

Here, getRollno( ) and getMarks( ) are getter functions and setGrade( ) is a setter function.

Previous

Next

Defining Member functions outside the class

Defining Member functions outside the class

We can also define member functions outside the class using the scope resolution operator**::**.

Example: Let's define two functions defined in student class outside the class.

```
#include <iostream>
using namespace std;
class Student {
    int rollno;
```

```
    char name[20];
    float marks;
    char grade;
    public:
    int getRollno();
    int getMarks();
    void setGrade() {
        if (marks > 90) grade = 'A';;
        else if (marks > 80) grade = 'B';
        else if (marks > 70) grade = 'C';
        else if (marks > 60) grade = 'D';
        else grade = 'E';
    }
};
int Student::getMarks() {
    return marks;
}
int Student::getRollNo() {
    return rollNo;
}
```

We can similarly access member functions via student objects and using the dot operator.

Previous

Next

Constructors, Type Of constructors

Constructors

A constructor is a special type of member function that is called automatically when an object is created. In C++, a constructor has the same name as that of the class, and it does not have a return type.

Example:

```
class D {
    public:
        // create a constructor
        D() {
            // code
        }
};
```

Here, the function D() is a constructor of the class D. Notice that the constructor

- has the same name as the class,
- does not have a return type, and
- is public

Types of Constructors

There are three types of constructors in C++ they are:

1. Default constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor

The default constructor is the constructor that doesn't take any argument. It has no parameters.

Syntax to define a Default Constructor:

```
Class_name()
```

Example:

```
#include <iostream>
using namespace std;
class construct {
    public:
        int a, b;
    // Default Constructor
    construct() {
        a = 10;
        b = 20;
    }
};
int main() {
    // Default constructor is called automatically when the
    //object is created
    construct c;
    cout << "a: " << c.a << endl <<
        "b: " << c.b;
    return 0;
}
Output:
a: 10
```

```
b: 20
```

Note: If we have not defined a constructor in our class, then the C++ compiler will automatically create a default constructor with an empty code and no parameters.

Default constructor as created by the compiler:

```
class Class_name() {
    public: Class_name() {}
};
```

Parameterized Constructor

A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its arguments provided by the programmer. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Syntax to define a parameterized Constructor:

```
Class_name(parameters)
```

Example:

```
#include <iostream>
using namespace std;
class construct {
    public:
        int a, b;
    // Default Constructor
    construct(int x, int y) {
        a = x;
        b = y;
    }
};
int main() {
    // Parameterized constructor is called
    construct c(40, 50);
    cout << "a: " << c.a << endl <<
        "b: " << c.b;
    return 0;
}
Output:
a: 40
b: 50
```

Copy Constructor

A copy constructor is a member function that initializes an object using another object of the same class.

Syntax to define a copy constructor:

```
Class_name(const Classname &old_object)
```

Example:

```
#include<iostream>
using namespace std;
class construct {
    private:
        int a, b;
    public:
        construct(int x, int y) {
            a = x;
            b = y;
        }
    // Copy constructor
    construct(const construct & p2) {
        a = p2.a;
        b = p2.b;
    }
    int getX() {
        return a;
    }
    int getY() {
        return b;
    }
};
int main() {
    construct p1(10, 15); // Normal constructor is called here
    construct p2 = p1; // Copy constructor is called here
    // Let us access values assigned by constructors
    cout << "p1.a = " << p1.getX() << ", p1.b = " << p1.getY();
    cout << "\np2.a = " << p2.getX() << ", p2.b = " << p2.getY();
    return 0;
}
Output:
p1.a = 10, p1.b = 15
p2.a = 10, p2.b = 15
```

How are constructors different from a normal member function?

- A constructor has the same name as the class itself.
- Constructors don’t have a return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, the C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Previous

Next

Constructor Overloading

Constructor Overloading

In C++, We can have more than one constructor in a class with the same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and a different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let the compiler know which constructor needs to be called.

Example:

```
#include <iostream>
using namespace std;
class construct {
    public:
        float area;
    // Constructor with no parameters
    construct() {
        area = 0;
    }
    // Constructor with two parameters
    construct(int a, int b) {
        area = a * b;
    }
    void disp() {
        cout << area << endl;
    }
};
int main() {
    // Constructor Overloading
```

```
    // with two different constructors
    // of class name
    construct o;
    construct o2(10, 20);
    cout << "Area using default constructor: ";
    o.disp();
    cout << "Area using parameterized constructor: ";
    o2.disp();
    return 0;
}
Output:
Area using default constructor: 0
Area using parameterized constructor: 200
```

Previous
Next
Destructors and Destructor Rules

Destructors

A destructor is a special member function that works just opposite to a constructor; unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax of a destructor:

```
~class_name()
{
   //Some code
}
```

Similar to the constructor, the destructor name should exactly match the class name. A destructor declaration should always begin with the tilde(~) symbol, as shown in the syntax above.

When is a destructor called?

A destructor function is called automatically when the object goes out of scope:

1. the function ends
2. the program ends
3. a block containing local variables ends
4. a delete operator is called

Example:

```
#include <iostream>
```

```
using namespace std;
class HelloWorld{
public:
  //Constructor
  HelloWorld(){
    cout<<"Constructor is called"<<endl;
  }
  //Destructor
  ~HelloWorld(){
    cout<<"Destructor is called"<<endl;
   }
   //Member function
   void display(){
     cout<<"Hello World!"<<endl;
   }
};
int main(){
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
   return 0;
}
Output:
Constructor is called
Hello World!
Destructor is called
```

Destructor rules

1. The name should begin with a tilde sign(~) and must match the class name.
2. There cannot be more than one destructor in a class.
3. Unlike constructors that can have parameters, destructors do not allow any parameter.
4. They do not have any return type, just like constructors.
5. When you do not specify any destructor in a class, the compiler generates a default destructor and inserts it into your code.

this Pointer

this pointer holds the current object's address; in simple words, this pointer points to the class's current object.

There can be three main usages of **this** keyword in C++.

- It can be used to pass the current object as a parameter to another method.
- It can be used to refer to a current class instance variable.
- It can be used to declare indexers.

Example:

```
#include <iostream>
using namespace std;
class Employee {
    public:
    int id; //data member
    string name; //data member
    float salary;
    Employee(int id, string name, float salary) {
        this -> id = id;
        this -> name = name;
        this -> salary = salary;
    }
    void display() {
        cout << id << "  " << name << "  " << salary << endl;
    }
};
int main(void) {
    Employee e1 = Employee(101, "Sonoo", 890000); //creating an object
    Employee e2 = Employee(102, "Nakul", 59000); //creating an object
    e1.display();
    e2.display();
    return 0;
}
Output:
101  Sonoo  890000
102  Nakul  59000
```

Shallow and Deep Copy

Shallow Copy

An object is created by copying all of the member field values. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred. Since both objects will reference the same memory location, then changes made by one will reflect those changes in another object. Since we wanted to create a replica of the object, this purpose will not be filled by Shallow copy.

Note: The assignment operator and the default copy constructor make a shallow copy.

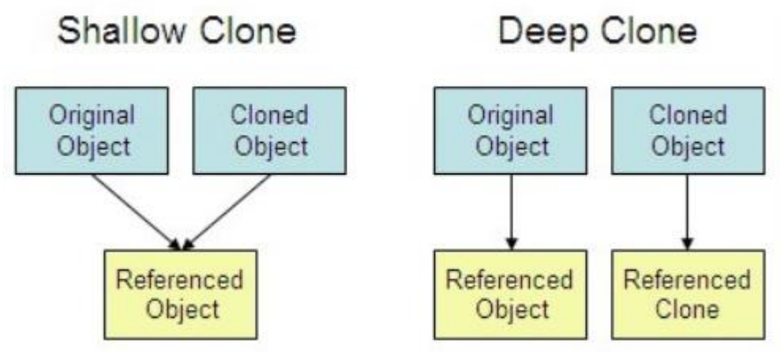
Example:

```
#include <iostream>
using namespace std;
class box {
    private:
    int length;
    int breadth;
    int height;
    public:
    void set_values(int a, int b,
                int c) {
                length = a;
                breadth = b;
                height = c;
    }
    void show_data() {
        cout << "Length = " << length << endl <<
                "Breadth = " << breadth << endl <<
                "Height = " << height <<
                endl;
    }
};
int main() {
    box B1, B3;
    // Set dimensions of Box B1
    B1.set_values(5, 10, 15);
    B1.show_data();
```

```
    /* When copying the data of object at the time of initialization
     then copy is made through COPY CONSTRUCTOR */
    box B2 = B1;
    B2.show_data();
    /* When copying the data of object after initialization then the
     copy is done through DEFAULT ASSIGNMENT OPERATOR */
    B3 = B1;
    B3.show_data();
    return 0;
}
Output:
Length = 5
Breadth = 10
Height = 15
Length = 5
Breadth = 10
Height = 15
Length = 5
Breadth = 10
Height = 15
```

Deep Copy

An object is created by copying all the fields, and it also allocates similar memory resources with the same value to the object. To perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is necessary to allocate memory to the other constructors' variables dynamically.



Example:

```
#include <iostream>
```

```
using namespace std;
class box {
    private:
    int length;
    int * breadth;
    int height;
    public:
    box() {
            breadth = new int;
    }
    void set_values(int a, int b,
        int c) {
        length = a;
        * breadth = b;
        height = c;
    }
    void show_data() {
        cout << "Length = " << length << endl <<
            "Breadth = " << breadth << endl <<
            "Height = " << height <<
            endl;
    }
    /* Parameterized Constructors for
for implementing deep copy */
    box(box & sample) {
            length = sample.length;
            breadth = new int;
            * breadth = * (sample.breadth);
            height = sample.height;
        }
        ~box() {
            delete breadth;
        }
};
int main() {
    box B1;
    // Set the dimensions of box B1
    B1.set_values(5, 10, 15);
    B1.show_data();
    /* When the data will be copied, then all the resources will also get
     allocated to the new object */
```

```
    box B2 = B1;
    // Display the dimensions
    B2.show_data();
    return 0;
}
Output:
Length = 5
Breadth = 0x1b94c20
Height = 15
Length = 5
Breadth = 0x1b94c40
Height = 15
```

Previous

Next

Encapsulation

Encapsulation

The process of grouping data members and corresponding methods into a single unit is known as Encapsulation. It is an important part of object-oriented programming. We can create a fully encapsulated class by making all the data members private. If the data members are private, it can only be accessible within the class; no other class can access that class's data members.

Example:

```
#include <iostream>
using namespace std;
class Student {
    // private data members
    private:
    string studentName;
    int studentRollno;
    int studentAge;
    // get method for student name to access
    // private variable studentName
    public:
        string getStudentName() {
            return studentName;
        }
    // set method for student name to set
```

```
    // the value in private variable studentName
    void setStudentName(string studentName) {
        this -> studentName = studentName;
    }
    // get method for student rollno to access
    // private variable studentRollno
    int getStudentRollno() {
        return studentRollno;
    }
    // set method for student rollno to set
    // the value in private variable studentRollno
    void setStudentRollno(int studentRollno) {
        this -> studentRollno = studentRollno;
    }
    // get method for student age to access
    // private variable studentAge
    int getStudentAge() {
        return studentAge;
    }
    // set method for student age to set
    // the value in private variable studentAge
    void setStudentAge(int studentAge) {
        this -> studentAge = studentAge;
    }
};
int main() {
    Student obj;
    // setting the values of the variables
    obj.setStudentName("Rahul");
    obj.setStudentRollno(101);
    obj.setStudentAge(22);
    // printing the values of the variables
    cout << "Student Name : " << obj.getStudentName() << endl;
    cout << "Student Rollno : " << obj.getStudentRollno() << endl;
    cout << "Student Age : " << obj.getStudentAge();
}
Output:
Student Name : Rahul
Student Rollno : 101
Student Age : 22
```

Advantages of encapsulation

1. Encapsulation is a way to achieve data hiding because other classes will not be able to access the data through the private data members.
2. In Encapsulation, we can hide the data's internal information, which is better for security concerns.
3. By encapsulation, we can make the class read-only. The code reusability is also an advantage of encapsulation.
4. Encapsulated code is better for unit testing.

Previous
Next
Inheritance and Mode Of Inheritance

Inheritance

Inheritance is an important pillar of object-oriented programming. Inheritance is the process of inheriting the properties and behavior of an existing class into a new class. When we inherit the class, we can reuse the existing class's methods and fields into a new class. Inheritance can also be defined as the **Is-A** relationship, which is also known as the parent-child relationship.

Let's understand the inheritance with the help of a real-world example.

Let's assume that **Human** is a class that has properties such as **height**, **weight**, **age**, etc. and functionalities (or methods) such as **eating()**, **sleeping()**, **dreaming()**, **working()**, etc.
Now we want to create **Male** and **Female** classes. Both males and females are humans, and they share some common properties (like **height**, **weight**, **age**, etc.) and behaviors (or functionalities like **eating()**, **sleeping()**, etc.), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have specific characteristics (like men having short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.
This approach makes us write less code as both the classes inherited several properties and functions from the superclass; thus, we didn't have to re-write them. Also, this makes it easier to read the code.

Why we use inheritance:

- The main advantage of inheritance is code reusability. We can reuse the code when we inherit the existing class's methods and fields into a new class.
- The runtime polymorphism (method overriding) can be achieved by inheritance only.

Important terminology of inheritance

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by the subclass is called Base Class or Superclass.

C++ inheritance syntax:

```
class parent_class {
    //Body of parent class
};
class child_class: access_modifier parent_class {
    //Body of child class
};
```

Here, child_class is the name of the subclass, access_mode is the mode in which you want to inherit this sub-class, for example, public, private, etc., and parent_class is the name of the superclass from which you want to inherit the subclass.

Modes of Inheritance

1. **Public mode**: If we derive a subclass from a public base class. Then, the base class's public members will become public in the derived class, and protected class members will become protected in the derived class.

2. **Protected mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

3. **Private mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Base Class member Access Specifier	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible	Not Accessible	Not Accessible

Previous

Next

Types of Inheritance

Types of Inheritance

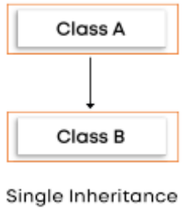
C++ supports five types of inheritance they are as follows:

1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

Single Inheritance

1. Single Inheritance:

In single inheritance, one class can extend the functionality of another class. In single inheritance, there is only one parent class and one child class.



Single Inheritance

Syntax:

```
class parent_class {
    //Body of parent class
};
class child_class: access_modifier parent_class {
    //Body of child class
};
```

Example:

```
#include<iostream>
using namespace std;
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};
// Child class
class Dog: public Animal {
    public: void bark() {
        cout << "barking";
    }
};
int main() {
    // Creating an object of the child class
    Dog obj;
    // calling methods
    obj.eat();
    obj.bark();
```

```
}
Output:
eating
barking
```

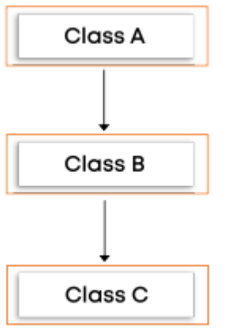
Previous

Next

Multilevel Inheritance

2. Multilevel Inheritance:

When a class inherits from a derived class, and the derived class becomes the base class of the new class, it is called multilevel inheritance. In multilevel inheritance, there is more than one level.



Multi-Level Inheritance

Syntax:

```
class parent_class {
    //Body of parent class
};
class child_class: access_modifier parent_class {
    //Body of child class
};
class child_class_of_child_class: access_modifier child_class {
        //Body of class
};
```

Example:

```
#include<iostream>
```

```
using namespace std;
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};
// Child class
class Dog: public Animal {
    public: void bark() {
        cout << "barking" << endl;
    }
};
class BabyDog: public Dog {
    public: void weep() {
        cout << "weeping";
    }
};
int main() {
    // Creating an object of the child class
    BabyDog obj;
    // calling methods
    obj.eat();
    obj.bark();
    obj.weep();
}
Output:
eating
barking
weeping
```

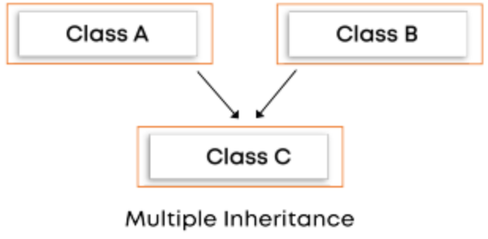
Previous

Next

Multiple Inheritance

3. Multiple Inheritance:

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance, a single child class can have multiple parent classes.



Multiple Inheritance

Syntax:

```
class parent_class1 {
    //Body of parent class1
};
class parent_class2 {
    //Body of parent class2
};
class child_class: access_modifier parent_class1, access_modifier parent_class2 {
    //Body of child class
};
```

Example:

```
#include<iostream>
using namespace std;
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};
// Parent class
class Dog {
    public: void bark() {
        cout << "barking" << endl;
    }
};
// Inherited class
class BabyDog: public Animal, public Dog {
    public: void weep() {
```

```
        cout << "weeping";
    }
};
int main() {
    // Creating an object of the child class
    BabyDog obj;
    // calling methods
    obj.eat();
    obj.bark();
    obj.weep();
}
Output:
eating
barking
weeping
```

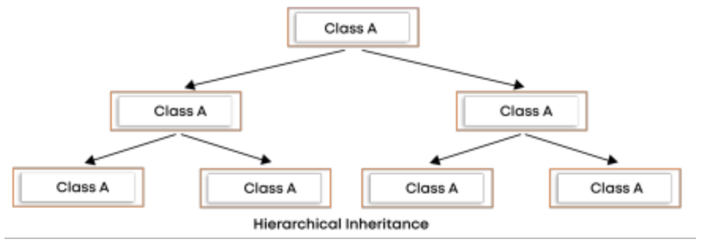
Previous

Next

Hierarchical Inheritance

4. Hierarchical Inheritance:

In hierarchical inheritance, one class serves as a base class for more than one derived class.



Syntax:

```
class parent_class {
    //Body of parent class
};
class child_class1: access_modifier parent_class {
    //Body of child class1
```

```
};
class child_class2: access_modifier parent_class {
    //Body of child class2
};
```

Example:

```
#include<iostream>
using namespace std;
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};
// child class1
class Dog: public Animal {
    public: void bark() {
        cout << "barking" << endl;
    }
};
// child class2
class Cat: public Animal {
    public: void meow() {
        cout << "meowing" << endl;
    }
};
int main() {
    Cat obj;
    // calling methods
    obj.eat();
    obj.meow();
}
Output:
eating
meowing
```

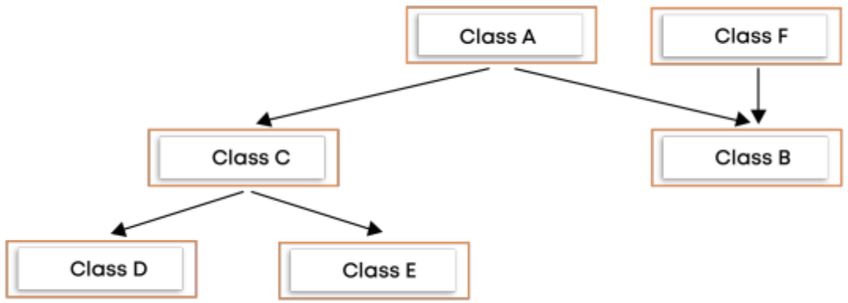
Previous

Next

Hybrid Inheritance

5. Hybrid Inheritance:

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.



Hybrid Inheritance- (a combination of Hierarchical and multiple)

Syntax:

```
class parent_class1 {
    //Body of parent class1
};
class parent_class2 {
    //Body of parent class1
};
class child_class1: access_modifier parent_class1 {
    //Body of child class1
};
class child_class2: access_modifier parent_class1, access_modifier parent_class2 {
    //Body of child class2
};
```

Example:

```
#include <iostream>
using namespace std;
// Parent class1
class Vehicle {
    public:
        Vehicle() {
            cout << "This is a Vehicle" << endl;
```

```
        }
};
//Parent class2
class Fare {
    public:
        Fare() {
            cout << "Fare of Vehicle\n";
        }
};
//Child class1
class Car: public Vehicle {
};
//Child class2
class Bus: public Vehicle, public Fare {
};
// main function
int main() {
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
Output:
This is a Vehicle
Fare of Vehicle
```

Previous

Next

Inheritance Ambiguity

C++ Ambiguity

There may be a possibility that a class may inherit member functions with the same name from two or more base classes, and the derived class may not have functions with the same name as those of its base classes. If the derived class object needs to access one of the same-named member functions of the base classes, it results in ambiguity as it is not clear to the compiler which base's class member function should be invoked.

Example:

```
#include<iostream>
```

```
using namespace std;

class A {
    public:
        void abc() {
            cout << "Class A";
        }
};

class B {
    public:
        void abc() {
            cout << "Class B";
        }
};

class C: public A, public B {
    public:
};

//Main Code
int main() {
    C obj;
    obj.abc();
}
```

```
Output:
error: request for member 'abc' is ambiguous
```

Avoid ambiguity using scope resolution operator

The ambiguity can be resolved by using the **scope resolution operator** by specifying the class in which the member function lies as given below:

Syntax:

```
object.class_name::method_name();
```

Example:

```
#include<iostream>
```

```
using namespace std;

class A {
    public:
        void abc() {
            cout << "Class A";
        }
};

class B {
    public:
        void abc() {
            cout << "Class B";
        }
};

class C: public A, public B {
    public:
};

//Main Code
int main() {
    C obj;
    obj.A :: abc();
}
```

```
Output:
Class A
```

Previous

Next

Access Specifiers Continued

Access Specifiers Continued

The access modifiers specify the accessibility of a variable, method, constructor, or class. There are three types of access modifiers available in C++.

- Private
- Public
- Protected

i) Private: The private access modifiers are only accessible within the same class in which they are declared. The data members and methods declared as private cannot be accessible from outside the class. If we try to access the data members and methods, which are declared private from outside the class, the compiler will give you a compile-time error. Let's look at the example below.

Example:

```
#include<iostream>
using namespace std;
class Circle {
    // private data member
    private:
        double radius;
    // public member function
    public:
        double compute_area() { // member function can access private
            // data member radius
            return 3.14 * radius * radius;
        }
};
int main() {
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;
    cout << "Area is:" << obj.compute_area();
    return 0;
}
Output:
In
function 'int main()':
11: 16: error: 'double Circle::radius'
is private
double radius; ^
31: 9: error: within this context
obj.radius = 1.5; ^
```

ii) Public: The data members, class, and methods declared as public can be accessed from anywhere. This access modifier can be accessed within the class and outside the class using the direct member access operator (.) with that class's object. Let's look at the example.

Example:

```
#include<iostream>
using namespace std;
class Circle {
    public:
        double radius;
    double compute_area() {
        return 3.14 * radius * radius;
    }
};
// main function
int main() {
    Circle obj;
    // accessing public data member outside class
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
Output:
Radius is: 5.5
Area is: 94.985
```

iii) Protected: Protected access modifier is similar to private access modifier, i.e., it can't be accessed outside of its class unless, with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass(derived class) of that class as well.

Example:

```
#include <iostream>
using namespace std;
class Parent {
    // protected data members
    protected:
        int id_protected;
};
class Child: public Parent {
    public: void setId(int id) {
        // Child class can access the inherited
        // protected data members of the base class
```

```
        id_protected = id;
    }
    void displayId() {
        cout << "id_protected is: " << id_protected << endl;
    }
};
int main() {
    Child obj1;
    // member function of the derived class can
    // access the protected data members of the base class
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
Output:
id_protected is: 81
```

Difference between private, protected, and public modifiers:

Let’s look at the summary of private, protected, and public access modifiers.

Visibility	Private	Protected	Public
Within the same class	Yes	Yes	Yes
In derived Class	No	Yes	Yes
Outside the class	No	No	Yes

Previous

Next

Polymorphism, Types Of Polymorphism

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism is a concept that allows you to perform a single action in different ways. Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms. Let’s understand polymorphism with a real-life example.

Real-life example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Types of polymorphism

There are two types of polymorphism in C++.

- Compile-time polymorphism
- Runtime polymorphism

Previous

Next

Compile time polymorphism

1. Compile time polymorphism

Compile-time polymorphism is also known as static polymorphism. This type of polymorphism can be achieved through function overloading or operator overloading.

a) Function overloading:

When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of function overloading is it increases the readability of the program. Functions can be overloaded by using different numbers of arguments and by using different types of arguments.

i) Function overloading with different numbers of arguments:

In this example, we have created two functions, the first add() performs the addition of the two numbers, and the second add() performs the addition of the three numbers. Let's look at the example:

```
#include <iostream>
using namespace std;
// Function with two parameters
int add(int num1, int num2) {
    return num1 + num2;
}
// Function with three parameters
int add(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}
int main() {
    cout << add(10, 20) << endl;
    cout << add(10, 20, 30);
    return 0;
}
```

```
Output:
30
60
```

ii) Function overloading with different types of arguments:

In this example, we have created two add() functions with different data types. The first add() takes two integer arguments and the second add() takes two double arguments.

```
#include <iostream>
using namespace std;
// Function with two integer parameters
int add(int num1, int num2) {
    return num1 + num2;
}
// Function with two double parameters
double add(double num1, double num2) {
    return num1 + num2;
}
int main(void) {
    cout << add(10, 20) << endl;
    cout << add(10.4, 20.5);
    return 0;
}
Output:
30
30.9
```

Default Arguments:

A default argument is a value provided in a function declaration automatically assigned by the compiler if the function's caller doesn't provide a value for the argument with a default value. However, if arguments are passed while calling the function, the default arguments are ignored.

Example: A function with default arguments can be called with 2 or 3 or 4 arguments.

```
#include<iostream>
using namespace std;
int add(int x, int y, int z = 0, int w = 0) {
    return (x + y + z + w);
}
int main() {
    cout << add(10, 20) << endl;
    cout << add(10, 20, 30) << endl;
    cout << add(10, 20, 30, 40) << endl;
    return 0;
}
```

```
Output:
30
60
100
```

b) Operator Overloading:

C++ also provides options to overload operators. For example, we can make the operator (‘+’) for the string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. A single operator, ‘+,’ when placed between integer operands, adds them and concatenates them when placed between string operands.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).
- Operators = and & are already overloaded in C++, so we can avoid overloading them.
- Precedence and associativity of operators remain intact.

List of operators that can be overloaded in C++:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

Example: Perform the addition of two imaginary or complex numbers.

```
#include<iostream>
using namespace std;
class Complex {
```

```
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i = 0) {
            real = r;
            imag = i;
        }
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex
        const & b) {
        Complex a;
        a.real = real + b.real;
        a.imag = imag + b.imag;
        return a;
    }
    void print() {
        cout << real << " + i" << imag << endl;
    }
};
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
Output:
12 + i9
```

Previous

Next

Runtime Polymorphism

2.Runtime polymorphism

Runtime polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymorphism.

a) Method overriding:

Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. In other words, whatever methods the parent class has by default are available in the child class. But, sometimes, a child class may not be satisfied with parent class method implementation. The child class is allowed to redefine that method based on its requirement. This process is called method overriding.

Rules for method overriding:

- The method of the parent class and the method of the child class must have the same name.
- The method of the parent class and the method of the child class must have the same parameters.
- It is possible through inheritance only.

Example:

```
#include<iostream>
using namespace std;
class Parent {
    public:
        void show() {
            cout << "Inside parent class" << endl;
        }
};
class subclass1: public Parent {
    public: void show() {
        cout << "Inside subclass1" << endl;
    }
};
class subclass2: public Parent {
    public: void show() {
        cout << "Inside subclass2";
    }
};
int main() {
    subclass1 o1;
    subclass2 o2;
    o1.show();
    o2.show();
}
Output:
Inside subclass1
Inside subclass2
```

Previous

Next

Abstraction

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only necessary information about the outside world's data, hiding the background details or implementation. Let's understand polymorphism with a real-life example.

Real-life example: When you send an email to someone, you just click send, and you get the success message; what happens when you click send, how data is transmitted over the network to the recipient is hidden from you (because it is irrelevant to you).

Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data members will be visible to the outside world and which is not.

Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, the pow() method present in the math.h header file. To calculate the power of a number, we can simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is calculating the power of numbers.

Abstraction using Access specifier

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.
- Members declared as **protected in a class** are a special kind of access specifier; it works similarly to private and can access it.

Example:

```
#include <iostream>
using namespace std;
class abstraction {
    private:
        int a, b;
    public:
        // method to set values of private members
        void set(int x, int y) {
            a = x;
            b = y;
        }
    void display() {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
```

```
};
int main() {
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
Output:
a = 10
b = 20
```

Advantages Of Abstraction

- Only you can make changes to your data or function, and no one else can.
- It makes the application secure by not allowing anyone else to see the background details.
- Increases the reusability of the code.
- Avoids duplication of your code.

Previous
Next

Interfaces

Interfaces

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes, and these abstract classes should not be confused with data abstraction, which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as a pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration.

Example: The following function is a pure virtual function.

```
virtual void fun() = 0;
```

Example:

```
#include<iostream>
using namespace std;
class Animal {
    public:
        //Pure Virtual Function
        virtual void sound() = 0;
```

```
    //Normal member Function
    void sleeping() {
        cout << "Sleeping";
    }
};
class Dog: public Animal {
    public: void sound() {
        cout << "Woof" << endl;
    }
};
int main() {
    Dog obj;
    obj.sound();
    obj.sleeping();
    return 0;
}
Output:
Woof
Sleeping
```

Previous

Next

Friend Function

Friend Function

A class's friend function is defined outside that class's scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend function in C++ is a function that is preceded by the keyword "friend."

Syntax:

```
class class_name {
    friend data_type function_name(argument); // syntax of friend function.
};
```

The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Example:

```
#include <iostream>
using namespace std;
class Rectangle {
    private:
        int length;
    public:
        Rectangle() {
            length = 10;
        }
    friend int printLength(Rectangle); //friend function
};
int printLength(Rectangle b) {
    b.length += 10;
    return b.length;
}
int main() {
    Rectangle b;
    cout << "Length of Rectangle: " << printLength(b) << endl;
    return 0;
}
Output:
Length of Rectangle: 20
```

Characteristics of friend function:

- A friend function can be declared in the private or public section of the class.
- It can be called a normal function without using the object.
- A friend function is not in the scope of the class, of which it is a friend.
- A friend function is not invoked using the class object as it is not in the class's scope.
- A friend function cannot access the private and protected data members of the class directly. It needs to make use of a class object and then access the members using the dot operator.
- A friend function can be a global function or a member of another class.

Previous
Next

Introduction

#2. Exception Handling in C++

In C++, an exception is an unwanted event that occurs during the execution of the program. It disrupts the normal flow of the program. Exception terminates the program execution. When an exception occurs in our program, we get a system-generated error message. We can handle the exception in C++ to provide a meaningful message to the user rather than a system-generated error.

Why does an exception occur?

Exceptions in C++ can occur due to the following reasons.

- Incorrect data entered by the user, e.g., dividing a no. by zero.
- Opening a file that doesn't exist in the program.
- Network connection problem
- Server down problem

What is exception handling in C++?

Exception handling is a mechanism in C++ used to handle the run time error so that the program's normal flow can be maintained. If an exception occurs in your code, then the rest of the code is not executed. Therefore, the C++ compiler creates an exception object. This exception object directly jumps to the default catch mechanism; there is a default message that prints on the screen and our program gets terminated.

Advantage of exception handling

The main advantage of exception handling is that it maintains the normal flow of the program. Suppose in our program there are many code lines and an exception occurs midway, then the code after the exception will not execute, and the program will terminate, and we get a system-generated error message which is not understandable to a user. By exception handling mechanism, all the statements in our program will execute. The program's normal flow will be maintained and generates a user-friendly message rather than a system-generated error message. That is why we use Exception Handling in C++.

Exception vs Error

Errors: Most of the time, errors are not caused by our programs. These are due to a lack of system resources. Errors are not recoverable. For example, if OutOfMemory occurs in our program, we can't do anything, and the program will terminate abnormally.

Exception: An exception is an unwanted event that occurs during the execution of the program. It disrupts the normal flow of the program. Exceptions are recoverable. NullPointerException, IOException, ArithmeticException are some examples of exceptions.

Previous
Next

Keywords

Exception handling keywords in C++

There are three keywords that we can use in exception handling in C++.

1. **try:** A try block identifies a block of code for which particular exceptions will be activated. One or more catch blocks follow it.
2. **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
3. **throw:** A program throws an exception when a problem shows up. This is done using a throw keyword.

The syntax **for using try, catch and throw:**

```
try {
    // Code
    throw exception; // Throw an exception when a problem arises
} catch () {
    // Code to handle exception
}
```

Example:

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

int main()
{
    try
    {
        int age = 12;
        if (age >= 18)
        {
            cout << "You are eligible to vote.";
        }
        else
        {
            throw(age);
        }
    }
    catch (int age)
    {
        cout << "Sorry, you are not eligible to vote" << endl;
        cout << "Your age is: " << age;
    }
}
Output:
Sorry, you are not eligible to vote
Your age is: 12
```

Explanation: The above code shows the eligibility for a user to vote according to his/her age. We wrote a program that tests if the age of the user is greater than or equal to 18 which makes him/her eligible, skipping the catch block however, failing to satisfy the eligibility criteria we throw an exception using the ‘throw’ keyword and then control passes to the ‘catch’ block.
The catch block takes a parameter of int data type as we throw an exception if ‘int’ type(age).

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

int main()
{
    try
    {
        int age = 19;
        if (age >= 18)
        {
            cout << "You are eligible to vote.";
        }
        else
        {
            throw(age);
        }
    }
    catch (int age)
    {
        cout << "Sorry, you are not eligible to vote" << endl;
        cout << "Your age is: " << age;
    }
}
Output:
You are eligible to vote.
```

We can also use the throw keyword to output a reference number, like a custom error number/code for organizing purposes:

Example:

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

int main()
{
    try
    {
        int age = 12;
        if (age >= 18)
        {
            cout << "You are eligible to vote.";
```

```
        }
        else
        {
            throw -1;
        }
    }
    catch (int num)
    {
        cout << "Sorry, you are not eligible to vote" << endl;
        cout << "Error " << num;
    }
}
Output:
Sorry, you are not eligible to vote
Error: -1
```

Handle Any Type of Exceptions: The following program consists of a ‘generalized’ catch block to catch any uncaught errors/exceptions by using the ‘three dots’ syntax(…) inside the catch block that takes care of all types of exceptions.

Example 1:

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

int main()
{
    try
    {
        int age = 12;
        if (age >= 18)
        {
            cout << "You are eligible to vote.";
        }
        else
        {
            throw "Error Found";
        }
    }
    catch (...)
    {
        cout << "Sorry, you are not eligible to vote" << endl;
```

```
    }
}
Output:
Sorry, you are not eligible to vote
```

Example 2:

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

int main()
{
    try
    {
        int age = 12;
        if (age >= 18)
        {
            cout << "You are eligible to vote.";
        }
        else
        {
            throw -1;
        }
    }
    catch (...)
    {
        cout << "Sorry, you are not eligible to vote" << endl;
    }
}
Output:
Sorry, you are not eligible to vote
```

Here, in both the examples, the throw type values are of different types but using (...) syntax, we didn't have to define separate catch statements.

Previous
Next

Standard Exceptions

Standard Exceptions in C++

In C++, standard exceptions are defined in the <exception> class that we can use inside our programs. The arrangement of the parent-child class hierarchy is shown below:

- **std::exception** - Parent class of all the standard C++ exceptions.

- **logic_error** - An exception that theoretically can be detected by reading the code.
- **domain_error** -This is an exception thrown after using a mathematically invalid domain.
- **invalid argument** - Exception due to invalid argument.
- **out_of_range** - Exception due to out of range, i.e., size requirement exceeds allocation, Thrown by **at** method.
- **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime; this is an exception that cannot be detected via reading the code.
- **range_error** - This occurs when you try to store a value that is out of range.
- **overflow_error** - Exception due to arithmetic overflow errors.
- **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - The **new** keyword throws this exception.
- **bad_cast** - This is an exception thrown by **dynamic_cast**.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier; this is a useful device to handle unexpected exceptions in a C++ program.
- **bad_typeid** - This is an exception thrown by **typeid**.

Previous
Next

Introduction

#3. STL in C++

The C++ Standard Template Library (STL) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks. It provides four components called algorithms, containers, functions, and iterators.

Previous

Next

Containers in C++

Containers in C++

The Containers Library in STL is a holder object that stores a collection of other objects, which in simplest words can be described as the objects used to contain data or rather a collection of objects, and it allows great flexibility in the types supported as elements. The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators. Some container classes are vector, list, forward_list, queue, priority_queue, stack, set, multiset, map, multimap, unordered_set.

Vector- A vector is a sequence container class that implements a dynamic array, which means that the size of the array automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

Syntax:

```
vector < object_type > v1;
```

Certain functions associated with the vector are:

1. **begin() –** Returns an iterator pointing to the first element in the vector
2. **end() –** Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. **rbegin() –** Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. **rend() –** Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as the reverse end)
5. **cbegin() –** Returns a constant iterator pointing to the first element in the vector.
6. **cend() –** Returns a constant iterator pointing to the theoretical element that follows the vector's last element.
7. **crbegin() –** Returns a constant reverse iterator, pointing to the vector's last element (reverse beginning). It moves from last to first element
8. **crend() –** Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as the reverse end)
9. **push_back() -** It adds a new element at the end.
10. **pop_back() -** It removes the last element from the vector.
11. **clear() -** It removes all the elements of the vector.

Example:

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

int main() {
    vector < int > v;
    for (int i = 1; i <= 5; i++)
        v.push_back(i);
    cout << "Output of begin and end: ";
    for (auto i = v.begin(); i != v.end(); ++i)
        cout << * i << " ";
    v.pop_back();
    cout << "\nOutput after pop: ";
    for (auto i = v.begin(); i != v.end(); ++i)
        cout << * i << " ";
    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = v.rbegin(); ir != v.rend(); ++ir)
        cout << * ir << " ";
    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = v.crbegin(); ir != v.crend(); ++ir)
        cout << * ir << " ";
    return 0;
}
```

```
Output:
Output of begin and end: 1 2 3 4 5
Output after pop: 1 2 3 4
Output of rbegin and rend: 4 3 2 1
Output of crbegin and crend : 4 3 2 1
```

Previous

Next

Algorithm Library
Algorithm Library
The header algorithm defines a lot of commonly used algorithms. They provide various operations for containers. Some well-known algorithms are

- sort(first_iterator, last_iterator) — for sorting.
- binary_search(first_iterator, last_iterator,x) — for searching element
- reverse(first_iterator, last_iterator) – to reverse
- *max_element (first_iterator, last_iterator) – to find the maximum element
- *min_element (first_iterator, last_iterator) – to find the minimum element
- accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation
- count(first_iterator, last_iterator,x) – to count the occurrences of x
- find(first_iterator, last_iterator, x) – checks if element is present or not.
- lower_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.
- upper_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.

To use **algorithms** – include <algorithm> header or you can simply use <bits/stdc++.h> header file.
i) sort()- It is one of the most basic built-in functions in C++ STL applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing.
It is implemented using a hybrid of QuickSort, HeapSort, and insertion sort. By default, it uses QuickSort, but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort. When the array size becomes small, it switches to Insertion Sort. It is known as IntroSort.

Syntax: Ascending order

```
sort(first, last);
```

We can use the “greater()” function to sort the data in descending order.

Syntax: Descending order

```
sort(first, last, greater());
```

Here,
first – is the index (pointer) of the first element in the range to be sorted.
last – is the index (pointer) of the last element in the range to be sorted.

Example: Sort array in ascending order.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {10, 35, 85, 93, 62, 77, 345, 43, 2, 10};
    int n = 10;
    sort(arr, arr + n);
    cout << "Sorted array: " << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

```
Output:
Sorted array:
2 10 10 35 43 62 77 85 93 345
```

Example: Sort array in descending order.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {10, 35, 85, 93, 62, 77, 345, 43, 2, 10};
    int n = 10;
    sort(arr, arr + n, greater < int > ());
    cout << "Sorted array: " << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
```

```
    return 0;
}
```

```
Output:
Sorted array:
345 93 85 77 62 43 35 10 10 2
```

Example: Sort vector in ascending order.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector < int > v = {10, 35, 85, 93, 62, 77, 345, 43, 2, 10};
    int n = 10;
    sort(v.begin(), v.end());
    cout << "Sorted vector: " << endl;
    for (int i = 0; i < n; i++)
        cout << v[i] << " ";
    return 0;
}
```

```
Output:
Sorted vector:
2 10 10 35 43 62 77 85 93 345
```

ii) binary_search()- The C++ builtin functions in C++ STL returns Boolean true if the element is present in the given range, else Boolean false is returned. It requires the vector to be sorted before the search is applied. This algorithm's main idea is to keep dividing the vector in half (divide and conquer) until the element is found or all the elements are exhausted. There are two variations of binary_search() they are as follows:

Syntax1:

```
binary_search(first, last, value);
```

Syntax2:

```
binary_search(first, last, value, compare_function);
```

Here,
start is the pointer that holds the memory location of the starting point of the search structure.
end is a pointer that holds the memory location of the endpoint of the search structure.

value is the element that is to be found using the function.
compare_function is the user-defined function that can be used to search non-numeric elements based on their properties.

Example:

```
#include <bits/stdc++.h>
using namespace std;

bool compare_string_by_length(string i, string j) {
    return (i.size() == j.size());
}


//Main Code
int main() {
    vector < int > v = {7, 8, 4, 1, 6, 5, 9, 4};
    sort(v.begin(), v.end());
    cout << binary_search(v.begin(), v.end(), 7); //prints 1 , Boolean true
    cout << binary_search(v.begin(), v.end(), 217); //prints 0 , Boolean false
    vector < string > s = {"test", "abcde", "efghijkl", "abc"};
    cout << binary_search(s.begin(), s.end(), "abcd", compare_string_by_length);
    /* search for the string in s which have same length as of "abcd" */
}
```

```
Output:
101
```

iii) reverse() – It reverses the vector.

Syntax:

```
reverse(first, last)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.

Example:

```
#include <bits/stdc++.h>
using namespace std;
```

```
int main() {
    vector <int> v = {10, 35, 85, 93, 62, 77, 345, 43, 2, 100};
    int n = 10;
    reverse(v.begin(), v.end());
    cout << "Reversed vector: " << endl;
    for (int i = 0; i < n; i++)
        cout << v[i] << " ";
    return 0;
}
```

```
Output:
Reversed vector:
100 2 43 345 77 62 93 85 35 10
```

iv) *max_element () – To find the maximum element of a vector.

Syntax:

```
*max_element(first, last)
```

Here,
first – is the index (pointer) of the first element from where the search begins.
last – is the index (pointer) of the last element where the search should end.

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector <int> v = {10, 35, 85, 93, 62, 77, 345, 43, 2, 100};
    int n = 10;

    cout << "Maximum element " << *max_element(v.begin(), v.end());;
    return 0;
}
```

```
Output:
Maximum element 345
```

v) *min_element () – To find the minimum element of a vector.

Syntax:

```
*min_element(first, last)
```

Here,
first – is the index (pointer) of the first element from where the search begins.
last – is the index (pointer) of the last element where the search should end..

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector <int> v = {10, 35, 85, 93, 62, 77, 345, 43, 2, 100};
    int n = 10;

    cout << "Minimum element " << *min_element(v.begin(), v.end());;
    return 0;
}
```

```
Output:
Minimum element 2
```

vi) accumulate() – Does the summation of vector elements.

Syntax:

```
accumulate(first, last, x)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.
x-initial value of the sum

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector <int> v = {10, 35, 85, 93, 62, 77, 345, 43, 2, 100};
    int n = 10;

    cout << "The summation of vector elements is: ";
    cout << accumulate(v.begin(), v.end(), 0);

    return 0;
}
```

```
Output:
The summation of vector elements is: 852
```

vii) count() – To count the occurrences of x in vector.

Syntax:

```
count(first, last, x)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.
x- value to count

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector <int> v = {10, 35, 85, 93, 62, 10, 345, 43, 2, 10};
    int n = 10;

    cout << "Occurrences of 20 in vector :  ";
    cout << count(v.begin(), v.end(), 10);
```

```
    return 0;
}
```

```
Output:
Occurrences of 20 in vector :  3
```

viii) find() – Returns an iterator to the first occurrence of x in the vector and points to the vector's last address.

Syntax:

```
find(first, last, x)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.
X-value to be searched

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector <int> v = {10, 35, 85, 93, 62, 10, 345, 43, 2, 10};
    int n = 10;

    cout << "Check if 5 is present in vector :  ";
    find(v.begin(), v.end(),5) != v.end()?
                          cout << "\nElement found":
                       cout << "\nElement not found";

    return 0;
}
```

```
Output:
```

```
Check if 5 is present in vector :
Element not found
```

ix) lower_bound() –It returns an iterator pointing to the first element in the range [first, last) which has a value not less than ‘x’.

Syntax:

```
lower_bound(first, last, x)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.
x-value

x) upper_bound() – It returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.

Syntax:

```
upper_bound(first, last, x)
```

Here,
first – is the index (pointer) of the first element.
last – is the index (pointer) of the last element.
x-value

Example:

```
#include <bits/stdc++.h>
#include<iostream>
using namespace std;

int main() {
    vector < int > v =  {10, 35, 85, 93, 62, 35, 345, 43, 2, 10};
    int n = 10;
    // Returns the first occurrence of 20
    auto q = lower_bound(v.begin(), v.end(), 35);
    // Returns the last occurrence of 20
    auto p = upper_bound(v.begin(), v.end(), 35);
    cout << "The lower bound is at position: ";
    cout << q - v.begin() << endl;
    cout << "The upper bound is at position: ";
    cout << p - v.begin() << endl;
    return 0;
```

```
}
```

```
Output:
The lower bound is at position: 1
The upper bound is at position: 10
```

Previous
Next
Mathematical Functions

Mathematical Functions

Mathematical functions are available in C++ to support various mathematical calculations; they can be directly used to simplify code and programs. C++ provides a large set of mathematical functions.
In order to use these functions you need to include header file- <math.h> or <cmath>.
cmath- It declares a set of functions to compute common mathematical operations and transformations some of them are pow(), sqrt(), round(), sin(), cos(), tan(), floor(), ceil(), abs().

Example:

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

int main() {
    double x = 5;
    cout << "Sine value of x=5 : " << sin(x) << endl;
    cout << "Cosine value of x=5 : " << cos(x) << endl;
    cout << "Tangent value of x=5 : " << tan(x) << endl;
    double y = 4;
    cout << "Square root value of y=4 : " << sqrt(y) << endl;
    int z = -10;
    cout << "Absolute value of z=-10 : " << abs(z) << endl;
    cout << "Power value: x^y = (5^4) : " << pow(x, y) << endl;
    x = 4.56;
    cout << "Floor value of x=4.56 is : " << floor(x) << endl;
    y = 12.3;
    cout << "Ceiling value of y=12.3 : " << ceil(y) << endl;
    return 0;
}
```

```
Output:
Sine value of x=5 : -0.958924
Cosine value of x=5 : 0.283662
Tangent value of x=5 : -3.38052
Square root value of y=4 : 2
Absolute value of z=-10 : 10
Power value: x^y = (5^4) : 625
Floor value of x=4.56 is : 4
Ceiling value of y=12.3 : 13
```

Previous

Next