# MODULE-3 Inheritance and Polymorphism

Inheritance: Types of Inheritance; Base and Derived classes, Syntax of derived classes, access to the base class; Types of Inheritance, Multiple inheritance, Virtual Base classes, Constructors and Destructors in Inheritance, Container classes, Abstract Classes. Polymorphism: Compile time (Early/Static binding), Overloading functions and operators, Overloading new and delete operators, Run time polymorphism (Late/Dynamic Binding), Virtual functions.

# Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

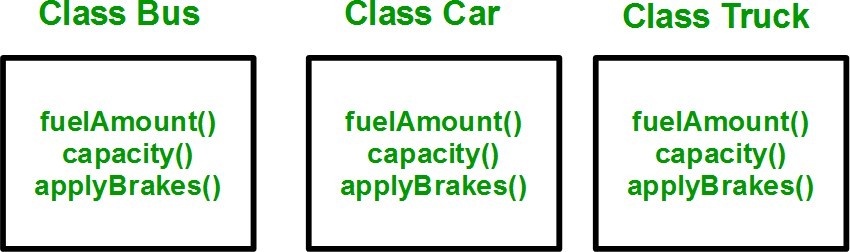
**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class. **Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class.

## The article is divided into following subtopics:

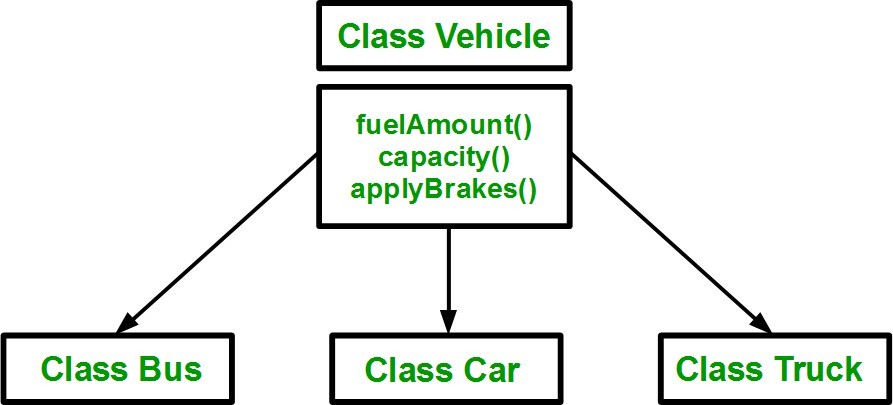
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

# Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

**Implementing inheritance in C++**: For creating a sub-class which is inherited from the base class we have to follow the below syntax. **Syntax**:

class subclass\_name : access\_mode base\_class\_name

{

//body of subclass

};

Here, **subclass\_name** is the name of the sub class, **access\_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base\_class\_name** is the name of the base class from which you want to inherit the sub class.

**Note**: A derived class doesn’t inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

// C++ program to demonstrate implementation of Inheritance

#include <bits/stdc++.h>

using namespace std;

//Base class class Parent

{

public: int id\_p;

};

// Sub class inheriting from Base Class(Parent) class Child : public Parent

{

public: int id\_c;

};

//main function int main()

{

Child obj1;

// An object of class child has all data members and member functions of class parent obj1.id\_c = 7;

obj1.id\_p = 91;

cout << "Child id is " << obj1.id\_c << endl; cout << "Parent id is " << obj1.id\_p << endl;

return 0;

}

Output:

Child id is 7

Parent id is 91

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

# Modes of Inheritance

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

**Note :** The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

// C++ Implementation to show that a derived class doesn’t inherit access to

// private data members. However, it does inherit a full parent object

class A

{

public:

int x; protected: int y;

private:

int z;

};

class B : public A

{

// x is public

// y is protected

// z is not accessible from B

};

class C : protected A

{

// x is protected

// y is protected

// z is not accessible from C

};

class D : private A // 'private' is default for classes

{

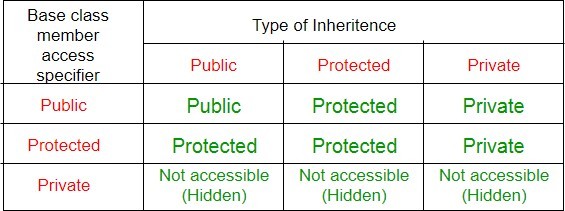
// x is private

// y is private

// z is not accessible from D

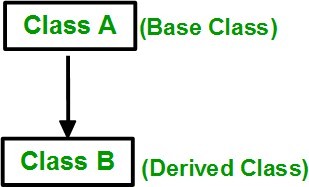
};

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:



# Types of Inheritance in C++

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



## Syntax:

class subclass\_name : access\_mode base\_class

{

//body of subclass

};

// C++ program to explain Single inheritance

#include <iostream>

using namespace std;

// base class

class Vehicle {

public:

Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

// sub class derived from single base class class Car: public Vehicle{

};

// main function int main()

{

// creating object of sub class will invoke the constructor of base classes

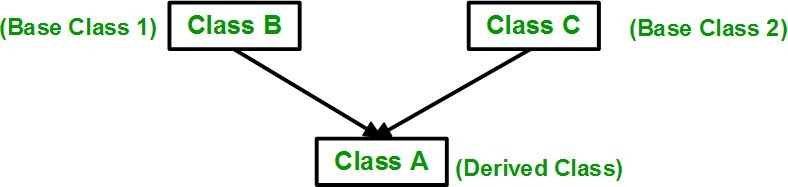
Car obj;

return 0;

}

Output:

This is a vehicle

1. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one class, i.e., one **sub class** is inherited from more than one **base class**

## Syntax:

class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....

{

//body of subclass

};

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

// C++ program to explain multiple inheritance

#include <iostream>

using namespace std;

// first base class class Vehicle { public:

Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

// second base class class FourWheeler { public:

FourWheeler()

{

cout << "This is a 4 wheeler Vehicle" << endl;

}

};

// sub class derived from two base classes

class Car: public Vehicle, public FourWheeler {

};

// main function int main()

{

// creating object of sub class will invoke the constructor of base classes Car obj;

return 0;

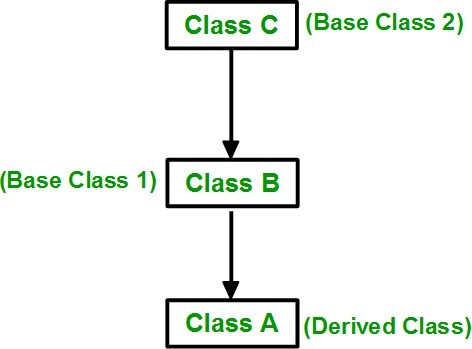
}

Output:

This is a Vehicle

This is a 4 wheeler Vehicle

1. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



// C++ program to implement Multilevel Inheritance

#include <iostream>

using namespace std;

// base class class Vehicle

{

public: Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

class fourWheeler: public Vehicle

{ public: fourWheeler()

{

cout<<"Objects with 4 wheels are vehicles"<<endl;

}

};

// sub class derived from two base classes

class Car: public fourWheeler{

public: car()

{

cout<<"Car has 4 Wheels"<<endl;

}

};

// main function int main()

{

//creating object of sub class will invoke the constructor of base classes

Car obj;

return 0;

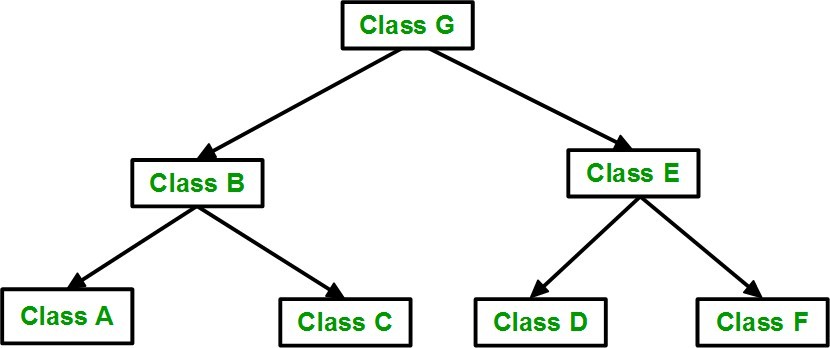
}

output:

This is a Vehicle

Objects with 4 wheels are vehicles Car has 4 Wheels

1. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



// C++ program to implement Hierarchical Inheritance

#include <iostream>

using namespace std;

// base class class Vehicle

{

public: Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

// first sub class

class Car: public Vehicle

{

};

// second sub class

class Bus: public Vehicle

{

};

// main function int main()

{

// creating object of sub class will invoke the constructor of base class

Car obj1;

Bus obj2; return 0;

}

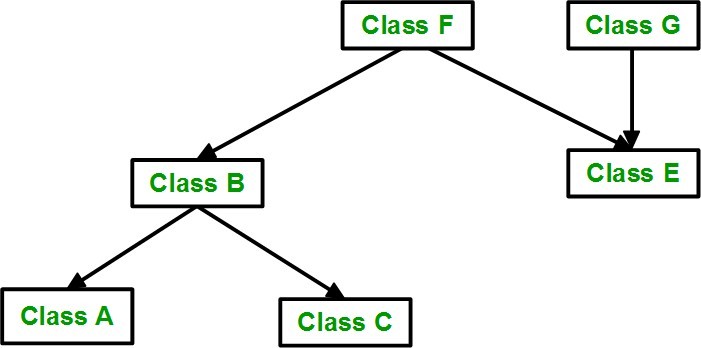
Output:

This is a Vehicle

This is a Vehicle

1. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:



// C++ program for Hybrid Inheritance #include <iostream>

using namespace std;

// base class class Vehicle

{

public: Vehicle()

{

cout << "This is a Vehicle" << endl;

}

};

//base class class Fare

{

public: Fare()

{

cout<<"Fare of Vehicle\n";

}

};

// first sub class

class Car: public Vehicle

{

};

// second sub class

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;

}

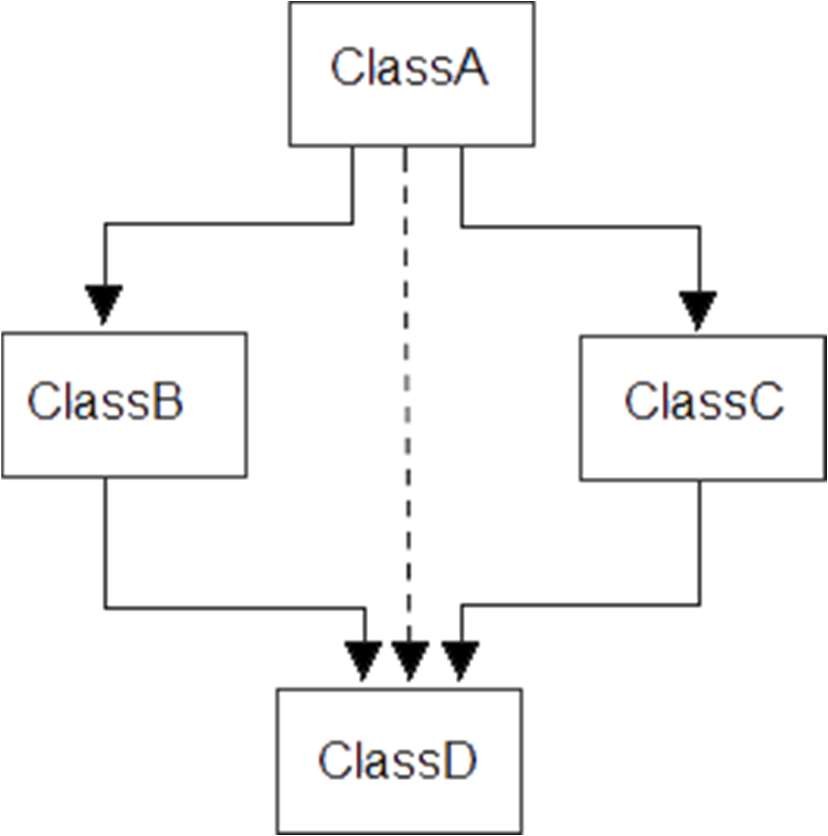
Output:

This is a Vehicle

Fare of Vehicle

## A special case of hybrid inheritance: Multipath inheritance:

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arise in this type of inheritance.



Consider the following program:

// C++ program demonstrating ambiguity in Multipath Inheritance

#include<iostream.h> #include<conio.h> class ClassA

{

public: int a;

};

class ClassB : public ClassA

{

public: int b;

};

class ClassC : public ClassA

{

public: int c;

};

class ClassD : public ClassB, public ClassC

{

public: int d;

};

void main()

{

ClassD obj;

//obj.a = 10; //Statement 1, Error

//obj.a = 100; //Statement 2, Error

obj.ClassB::a = 10; //Statement 3

obj.ClassC::a = 100; //Statement 4

obj.b = 20;

obj.c = 30;

obj.d = 40;

cout<< "\n A from ClassB : "<< obj.ClassB::a; cout<< "\n A from ClassC : "<< obj.ClassC::a;

cout<< "\n B : "<< obj.b; cout<< "\n C : "<< obj.c; cout<< "\n D : "<< obj.d;

}

Output:

A from ClassB : 10

A from ClassC : 100

B : 20

C : 30

D : 40

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco’z compiler can’t differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

1. **Use scope resolution operator**
2. **Use virtual base class**

**Avoiding ambiguity using scope resolution operator:**

Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.

obj.ClassB::a = 10; //Statement 3

obj.ClassC::a = 100; //Statement 4

Note : Still, there are two copies of ClassA in ClassD.

## Avoiding ambiguity using virtual base class:

#include<iostream.h> #include<conio.h>

class ClassA

{

public: int a;

};

class ClassB : virtual public ClassA

{

public: int b;

};

class ClassC : virtual public ClassA

{

public: int c;

};

class ClassD : public ClassB, public ClassC

{

public: int d;

};

void main()

{

ClassD obj;

obj.a = 10; //Statement 3

obj.a = 100; //Statement 4

obj.b = 20;

obj.c = 30;

obj.d = 40;

cout<< "\n A : "<< obj.a; cout<< "\n B : "<< obj.b; cout<< "\n C : "<< obj.c; cout<< "\n D : "<< obj.d;

}

Output:

A : 100

B : 20

C : 30

D : 40

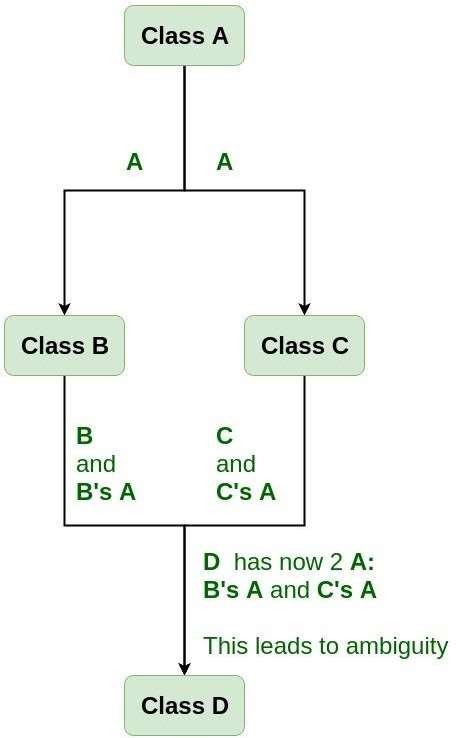
According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

# Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

## Need for Virtual Base Classes:

Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

**Example:** To show the need of Virtual Base Class in C++ #include <iostream>

using namespace std;

class A { public:

void show()

{

cout << "Hello form A \n";

}

};

class B : public A {

};

class C : public A {

};

class D : public B, public C {

};

int main()

{

D object; object.show();

}

## Compile Errors:

prog.cpp: In function 'int main()':

prog.cpp:29:9: error: request for member 'show' is ambiguous object.show();

^

prog.cpp:8:8: note: candidates are: void A::show() void show()

^

prog.cpp:8:8: note: void A::show()

**How to resolve this issue?**

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

## Syntax for Virtual Base Classes:

**Syntax 1:**

class B : virtual public A

{

};

**Syntax 2:**

class C : public virtual A

{

};

**Note: virtual** can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.

Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

## Example 1

#include <iostream> using namespace std;

class A { public:

int a;

A() // constructor

{

a = 10;

}

};

class B : public virtual A {

};

class C : public virtual A {

};

class D : public B, public C {

};

int main()

{

D object; // object creation of class d cout << "a = " << object.a << endl;

return 0;

}

## Output:

a = 10

**Explanation :**The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

**Example 2:**

#include <iostream> using namespace std;

class A { public:

void show()

{

cout << "Hello from A \n";

}

};

class B : public virtual A {

};

class C : public virtual A {

};

class D : public B, public C {

};

int main()

{

D object; object.show();

}

## Output:

Hello from A

## Virtual classes

Virtual classes are primarily used during multiple inheritance. To avoid, multiple instances of the same class being taken to the same class which later causes ambiguity, virtual classes are used.

#include <iostream>

using namespace std; class A {

public: int a; A(){

a = 10;

}

};

class B : public virtual A {

};

class C : public virtual A {

};

class D : public B, public C {

};

int main(){

//creating class D object D object;

cout << "a = " << object.a << endl; return 0;

}

Output a = 10

**Constructors and Destructors in Inheritance**

When we are using the constructors and destructors in the inheritance, parent class constructors and destructors are accessible to the child class hence when we create an object for the child class, constructors and destructors of both parent and child class get executed.

C++ program to the sequence of execution of constructor and destructor inheritance #include <iostream>

using namespace std;

class parent //parent class

{

public:

parent() //constructor

{

cout << "Parent class Constructor\n";

}

~parent()//destructor

{

cout << "Parent class Destructor\n";

}

};

class child : public parent //child class

{

public:

child() //constructor

{

cout << "Child class Constructor\n";

}

~ child() //destructor

{

cout << "Child class Destructor\n";

}

};

int main()

{

//automatically executes both child and parent class constructors and //destructors because of inheritance

child c;

return 0;

}

Output Parent class Constructor

Child class Constructor Child class Destructor Parent class Destructor

## Inheritance in Parametrized Constructor/ Destructor

In the case of the default constructor, it is implicitly accessible from parent to the child class but parameterized constructors are not accessible to the derived class automatically, for this reason, an explicit call has to be made in the child class constructor to access the parameterized constructor of the parent class to the child class using the following syntax

<class\_name>:: constructor(arguments) #include <iostream>

using namespace std;

class parent

{

int x; public:

// parameterized constructor parent(int i)

{

x = i;

cout << "Parent class Parameterized Constructor\n";

}

};

class child: public parent

{

int y; public:

// parameterized constructor

child(int j) : parent(j) //Explicitly calling

{

y = j;

cout << "Child class Parameterized Constructor\n";

}

};

int main()

{

child c(10);

return 0;

}

Output

Parent class Parameterized Constructor Child class Parameterized Constructor

# Constructor call in multiple inheritance constructors

class C: public A, public B;

Constructors are called upon the order in which they are inherited

First class A constructors are executed followed by class B constructors, then class C constructors

# Containers in C++ STL

Standard Template Library (STL) provides the built-in implementation of commonly used data structures known as containers. A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the data types supported.

A container manages the storage space for its elements and provides member functions for easy access to useful operations.

**Types of STL Containers**

STL containers are divided into the following categories:

Sequence Containers Associative Containers Unordered Containers Container Adapters

**Sequence Containers**

Sequence containers implement linear data structures in which the elements can be accessed sequentially. Following are the sequence containers in C++ STL:

|  |  |
| --- | --- |
| Container  Name Description  Array Container that wraps over fixed size static array. Vector Automatically resizable dynamic array.  Deque Dynamic array of fixed-size arrays that allows fast insertions and deletions at both ends.  List Implementation of Doubly Linked List data structure. | |
| Forward List | Implementation of Singly Linked List data structure. |

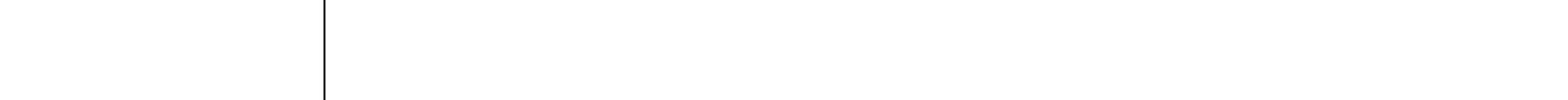
## Associative Containers

Associative containers store data in some sorted order. It provides fast search, insert and delete in O(log n) time by using balanced trees like Red Black Trees.

|  |  |
| --- | --- |
| Container  Name Description  Set Collection of unique elements sorted on the basis of their values.  Map Collection of key-value pairs sorted on the basis of the keys where no two pairs have same keys. | |
| Multiset | Collection of elements sorted on the basis of their values but allows multiple copies of values. |
| Multimap | Collection of key-value pairs sorted on the basis of the keys where multiple pairs can have same keys. |

## Unordered Associative Containers

Unordered associative containers implement unsorted hashed data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).



Container Name

Description

Unordered Set

Collection of unique elements hashed by their values.

Unordered Map

Collection of key-value pairs that are hashed by their keys where no two pairs have same keys.

Unordered Multiset

Collection of elements hashed by their values and allows multiple copies of values.

Unordered Multimap

Collection of key-value pairs that are hashed by their keys where multiple pairs can have same keys.

## Container Adapters

Container adapters provide a different interface for other containers. They adapt the behavior of underlying containers to fit specific use cases.



Container Name

Description

Stack

Adapts a container to provide stack (LIFO) data structure.

Queue

Adapts a container to provide queue (FIFO) data structure.

Priority Queue Adapts a container to provide heap data structure.

## Abstract Class

In C++, an abstract class is defined by having at least one pure virtual function, a function without a concrete definition. These classes are essential in object-oriented programming, structuring code to mirror real-life scenarios through inheritance and abstraction. Abstract classes, which cannot be instantiated, are pivotal for expressing broad concepts and upcasting, allowing derived classes to implement their interfaces. Utilize pointers and references for abstract class types, and remember that any subclass failing to define the pure virtual function becomes abstract itself. The virtual function is declared with the pure specifier (= 0).

## Abstract Class in C++ Example

Let’s say we are making a calculator which returns the perimeter of the shape we put in. Think of what kind of code you would write for such a calculator. You might start with some initial shapes and hardcode the perimeter by making separate functions inside the Shape class.

The class might look something like this – class Shape {

public:

// All the functions of both square and rectangle are clubbed together in a single class.

void width(int w) {

shape\_width = w;

}

void height(int h) { shape\_height = h;

}

int perimeterOfSquare(int s) { return 4 \* s;

}

int perimeterOfRectange(int l, int b) { return 2 \* (l + b);

}

protected:

int shape\_width; int shape\_height;

};

## Restrictions on Abstract Classes

There are some restrictions on abstract classes in C++. Abstract classes cannot be used for the following – Variables or member data, Argument types, Function return types,

Types of explicit conversions.

#include<iostream> using namespace std;

// Shape class. class Shape { public:

// Function to calculate the parameter, declared as pure virtual, so all the derived //classes necessarily need to implement this.

virtual int perimeter() = 0;

void width(int w) { shape\_width = w;

}

void height(int h) { shape\_height = h;

}

protected:

int shape\_width;

int shape\_height;

};

class Rectangle: public Shape { public:

// Class Rectangle provided implementation of perimeter() function.

int perimeter() {

return (2 \* (shape\_width + shape\_height));

}

};

class Square: public Shape { public:

// Class Square provided implementation of perimeter() function.

int perimeter() {

return (4 \* shape\_width);

}

};

int main() { Rectangle R; Square S;

R.width(10);

R.height(5);

S.width(10);

cout << "Perimeter of Rectangle: " << R.perimeter() << endl; cout << "Perimeter of Square: " << S.perimeter() << endl; return 0;

}

Output –

Perimeter of Rectangle : 30 Perimeter of Square: 40

Characteristics of Abstract Class in C++

1. Abstract Classes must have at least one pure virtual function.

virtual int perimeter() = 0;

1. Abstract Classes cannot be instantiated, but pointers and references of Abstract Class types can be created. You cannot create an object of an abstract class. Here is an example of a pointer to an abstract class.

#include<iostream> using namespace std;

class Base { public:

virtual void print() = 0;

};

class Derived: public Base { public:

void print() {

cout << "This is from the derived class \n";

}

};

int main(void) {

Base\* basePointer = new Derived(); basePointer -> print();

return 0;

}

Output –

This is from the derived class

1. Abstract Classes are mainly used for Upcasting, which means its derived classes can use its interface.
2. Classes that inherit the Abstract Class must implement all pure virtual functions. If they do not, those classes will also be treated as abstract classes.

Polymorphism

"The process of representing one form in multiple forms is known as Polymorphism.""Polymorphism is derived from 2 Greek words: poly and morphs.""Poly" means many, "morphs" means forms.

2 types

* Compile time polymorphism Uses static or early binding

Example: Function and operator overloading

* Run time polymorphism Uses dynamic or early binding

Example: Virtual functions

class A

{

int x;

public:

void show() { .... } // show() in base class

};

class B : public A

{

int y; public:

void show() { .... } // show() in derived class

};

## Introduction to Runtime Polymorphism:

It would be useful if the program could select the appropriate function at runtime instead of compile-time. This concept is known as runtime polymorphism.

Polymorphism

├── Compile-time polymorphism

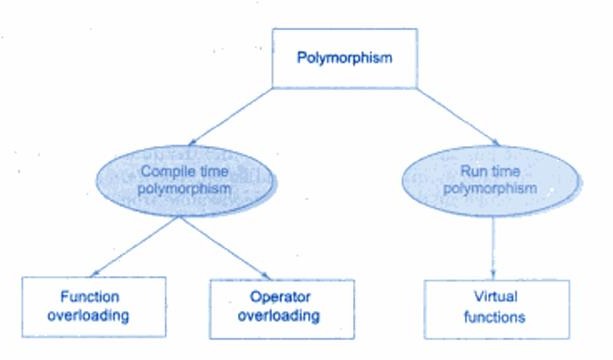
│ ├── Function overloading

│ ├── Operator overloading

│

└── Run-time polymorphism

├── Virtual functions



* Compile-time polymorphism (Early Binding)

Function overloading: Multiple functions with the same name but different parameters.

Operator overloading: Defining custom behavior for operators like +, -, etc.

* Run-time polymorphism (Late Binding) Achieved using virtual functions in C++.

The function that gets executed is determined at runtime using dynamic binding.

## C++ Overloading (Operator and Function)

* C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.
* An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
* When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

## Function Overloading

* Function Overloading (achieved at compile time)
* Function Overloading provides multiple definitions of the function by changing signature i.e. changing number of parameters, change datatype of parameters, return type doesn’t play any role.
* It can be done in base as well as derived class

void area(int a);

void area(int a, int b);

class printData { public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer pd.print(5);

// Call print to print float pd.print(500.263);

// Call print to print character pd.print("Hello C++");

return 0;

}

OUTPUT

Printing int: 5

Printing float: 500.263 Printing character: Hello C++

## Operators Overloading

Operator overloading is a compile-time polymorphism.

* It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
* C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading
* we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

## Operations that can be performed:

1. Arithmetic operations: + – \* / %
2. Logical operations: && and ||
3. Relational operations: == != >= <=
4. Pointer operators: & and \*
5. Memory management operator: new, delete []

Let us multiply two fractions using the overloading of the multiplication operator in C++.

// Multiplication of two fractions #include <iostream>

using namespace std;

class Frac { private:

int a; int b;

public:

Frac() : a(0), b(0) {}

void in() {

cout << "Enter the numerator : "; cin >> a;

cout<< "Enter the denominator : "; cin >> b;

}

// Overload the \* operator

Frac operator \* (const Frac &obj) { Frac temp;

temp.a = a \* obj.a;

temp.b = b \* obj.b;

return temp;

}

void out() {

cout<<"The fraction is "<< a<<"/ "<<b;

}

};

int main() {

Frac F1, F2, result;

cout << "Enter the first fraction:n"; F1.in();

cout << "Enter the second fraction:n"; F2.in();

// complex1 calls the operator function

// complex2 is passed as an argument to the function result = F1 \* F2;

result.out();

return 0;

}

OutPut

Enter the first fraction: Enter the numerator : 2 Enter the denominator : 5

Enter the second fraction:

ENter the numerator: 12 Enter the denominator: 7 The fraction is 24/35

A C++ program to overload a prefix decrement operator

#include <iostrea

}

// Overload the prefix decrement operator void operator-- () {

a= --a;

b= --b;

}

void out() {

cout<<"The decremented elements of the object are: "<<endl<< a<<" and " <<b;

}

};

int main() { OverLoad obj; obj.in();

--obj;

obj.out();

return 0;

}

Output

Enter the first number : 56 Enter the second number : 234

The decremented elements for the objects are: 55 and 223

In C++, the rules of Operator overloading in c++ include:

* Overloaded operators must have at least one operand that is a user-defined type.
* Overloaded operators cannot change the precedence and associativity of operators.
* Certain operators cannot be overloaded. (dot), .\* (pointer to member), :: (scope resolution), ?: (ternary conditional), and sizeof.
* The arity of the Operator cannot be changed; for instance, you cannot make + unary

if it’s binary.

* Overloaded operators maintain their original meaning for built-in types.
* Some operators are not suitable for overloading. For example, &&, ||, ,, . (member selection), .\* (member selection through a pointer), and:: (scope resolution).
* The overloaded operators can be member functions or global functions.
* Unary operators have one operand, binary operators have two, and ternary operators have three.

Which operators Cannot be overloaded?

* Conditional [?:], size of, scope(::), Member selector(.), member pointer selector(.\*) and the casting operators.
* We can only overload the operators that exist and cannot create new operators or rename existing operators.
* At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
* It is not possible to change the number of operands of an operator supports.
* All operators keep their default precedence and associations (what they use for), which cannot be changed.
* Only built-in operators can be overloaded.

## Unary Operators Overloading

Unary operator overloading involves defining behaviors for operators that act on a single operand. For example, increment (++), decrement (—), logical NOT (!), and unary minus (–) are unary operators that can be overloaded in C++. This means you can redefine their behavior for your custom classes.

Overloading the increment and decrement operators through a C++ program.

#include<iostream> using namespace std;

class UnaryOverload

{

int hr, min; public:per

void in()

{

cout<<"n Enter the time: n"; cin>>hr;

cout<<endl; cin>>min;

}

void operator++(int) //Overload Unary Increment

{

hr++; min++;

}

void operator--(int) //Overload Unary Decrement

{

hr--;

min--;

}

void out()

{

cout<<"nTime is "<<hr<<"hr "<<min<<"min";

}

};

int main()

{

UnaryOverload ob; ob.in();

ob++;

cout<<"nn After Incrementing : "; ob.out();

ob--;

ob--;

cout<<"nn After Decrementing : "; ob.out();

return 0;

}

Output

Enter the time:

5

56

After Incrementing: Time is 6hr 57 mins After Decrementing: Time is 4hr 55 min

Binary Operator Overloading

#include<iostream> using namespace std;

class Time { private:

int hour; int minute;

public:

Time() : hour(0), minute(0) {}

void in() {

cout << "Enter the time: "; cin >> hour;

cin >> minute;

}

// Overload the + operator

Time operator + (const Time & obj) { Time temp;

temp.hour = hour + obj.hour; temp.minute = minute + obj.minute; if (temp.minute>=60)

{

temp.hour+=1; temp.minute-=60;

}

if (temp.hour>24) temp.hour=1; return temp;

}

void out() {

cout<<"Time is "<< hour<<"hrs "<<minute<<"min";

}

};

int main() {

Time T1, T2, result;

cout << "Enter first time in hours and minutes one by one :n"; T1.in();

cout << "Enter second time in hours and minutes one by one :n"; T2.in();

// T1 calls the operator function

// T2 is passed as an argument to the function result = T1 + T2;

result.out();

return 0;

}

Output

Enter first time in hours and minutes one by one: Enter the time:11

56

Enter second time in hours and minutes one by one: Enter the time: 2

10

Time is 14hrs 6 min

## Overloading New and Delete operator in c++

* The new and delete operators can also be overloaded like other operators in C++
* If these operators are overloaded using member function for a class, it means that these operators are overloaded only for that specific class.
* If overloading is done outside a class (i.e. it is not a member function of a class), the overloaded ‘new’ and ‘delete’ will be called anytime you make use of these operators (within classes or outside classes). This is global overloading.

The syntax for overloading the new operator

void\* operator new(size\_t size);

The overloaded new operator receives size of type size\_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new must be void\*.The overloaded function returns a pointer to the beginning of the block of memory allocated.

Syntax for overloading the delete operator :

void operator delete(void\*);

The function receives a parameter of type void\* which has to be deleted. Function should not return anything.

Both overloaded new and delete operator functions are static members by default.

Therefore, they don’t have access to this pointer .

Overloading new and delete operator for a specific class:

#include<iostream> #include<stdlib.h>

using namespace std; class student

{

string name; int age;

public:

student()

{

cout<< "Constructor is called\n" ;

}

student(string name, int age)

{

this->name = name; this->age = age;

}

void display()

{

cout<< "Name:" << name << endl; cout<< "Age:" << age << endl;

}

void \* operator new(size\_t size)

{

cout<< "Overloading new operator with size: " << size << endl; void \* p = ::operator new(size);

//void \* p = malloc(size); will also work fine

return p;

}

void operator delete(void \* p)

{

cout<< "Overloading delete operator " << endl; free(p);

}

};

int main()

{

student \* p = new student("Yash", 24);

p->display(); delete p;

}

OUTPUT

Overloading new operator with size: 40 Name:Yash

Age:24

Overloading delete operator

void \* operator new(size\_t size)

{

cout<< "Overloading new operator with size: " << size << endl; void \* p = ::operator new(size);

//void \* p = malloc(size); will also work fine

return p;

}

void operator delete(void \* p)

{

cout<< "Overloading delete operator " << endl; free(p);

}

};

int main()

{

student \* p = new student("Yash", 24);

p->display(); delete p;

}

Overloading new operator with size: 40 Name:Yash

Age:24

Overloading delete operator

## Runtime polymorphism(late/dynamic binding)

* Runtime polymorphism, also known as dynamic or late binding, is a programming technique that determines which method to call at runtime.
* In runtime polymorphism, the compiler resolves the object at run time and then it decides which function call should be associated with that object. It is also known as dynamic or late binding polymorphism. This type of polymorphism is executed through virtual functions and function overriding. All the methods of runtime polymorphism get invoked during the run time

#include <iostream> #include <cstring>

using namespace std;

class media { protected:

char title[50]; float price;

public:

media(char \*s, float a) { strcpy(title, s);

price = a;

}

virtual void display() { } // Empty virtual function

};

class book : public media { int pages;

public:

book(char \*s, float a, int p) : media(s, a) { pages = p;

}

void display(); // Function definition not visible in image

};

class tape : public media { float time;

public:

tape(char \*s, float a, float t) : media(s, a) { time = t;

}

void display();

};

// Function Definitions void book::display() {

cout << "\n Title: " << title; cout << "\n Pages: " << pages; cout << "\n Price: " << price;

}

void tape::display() {

cout << "\n Title: " << title;

cout << "\n Play time: " << time << " mins"; cout << "\n Price: " << price;

}

int main() {

char \*title = new char[30]; float price, time;

int pages;

// Book details

cout << "\n ENTER BOOK DETAILS\n";

cout << " Title: "; cin >> title; cout << " Price: "; cin >> price; cout << " Pages: "; cin >> pages; book book1(title, price, pages);

// Tape details

cout << "\n ENTER TAPE DETAILS\n";

cout << " Title: "; cin >> title; cout << " Price: "; cin >> price;

cout << " Play Time (mins): "; cin >> time; tape tape1(title, price, time);

media \*list[2]; // Base class pointers list[0] = &book1;

list[1] = &tape1;

cout << "\n MEDIA DETAILS:";

list[0]->display();

list[1]->display();

delete[] title; // Free allocated memory return 0;

}



**Virtual function**

* A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the method.
* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
* They are mainly used to achieve Runtime polymorphism.
* Functions are declared with a virtual keyword in a base class. The resolving of a function call is done at runtime.

Rules for Virtual Functions

* The rules for the virtual functions in C++ are as follows:
* Virtual functions cannot be static.
* A virtual function can be a friend function of another class.
* Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
* The prototype of virtual functions should be the same in the base as well as the derived class.
* They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
* A class may have a virtual destructor but it cannot have a virtual constructor.

#include <iostream> using namespace std;

class base { public:

virtual void print() { cout << "print base class\n"; }

void show() { cout << "show base class\n"; }

};

class derived : public base { public:

void print() { cout << "print derived class\n"; }

void show() { cout << "show derived class\n"; }

};

int main()

{

base\* bptr; derived d; bptr = &d;

// Virtual function, binded at runtime bptr->print();

// Non-virtual function, binded at compile time bptr->show();

return 0;

}

Output

print derived class show base class

**Programs for Practice**

1. C++ program to demonstrate example of simple inheritance.
2. C++ program to demonstrate example of private simple inheritance.
3. C++ program to read and print student's information using two classes and simple inheritance.
4. C++ program to demonstrate example of multilevel inheritance.
5. C++ program to read and print employee information using multiple inheritance.
6. C++ program to demonstrate example of multiple inheritance.
7. C++ program to demonstrate example of hierarchical inheritance to get square and cube of a number.
8. C++ program to read and print employee information with department and pf information using hierarchical inheritance.