Polymorphism

Polymorphism comes from the Greek words "poly" and "morphism".

"poly" means many and "morphism" means form i.e.. many forms.

- **Polymorphism means the ability to take more than one form.**

For example, an operation have different behavior in different instances.

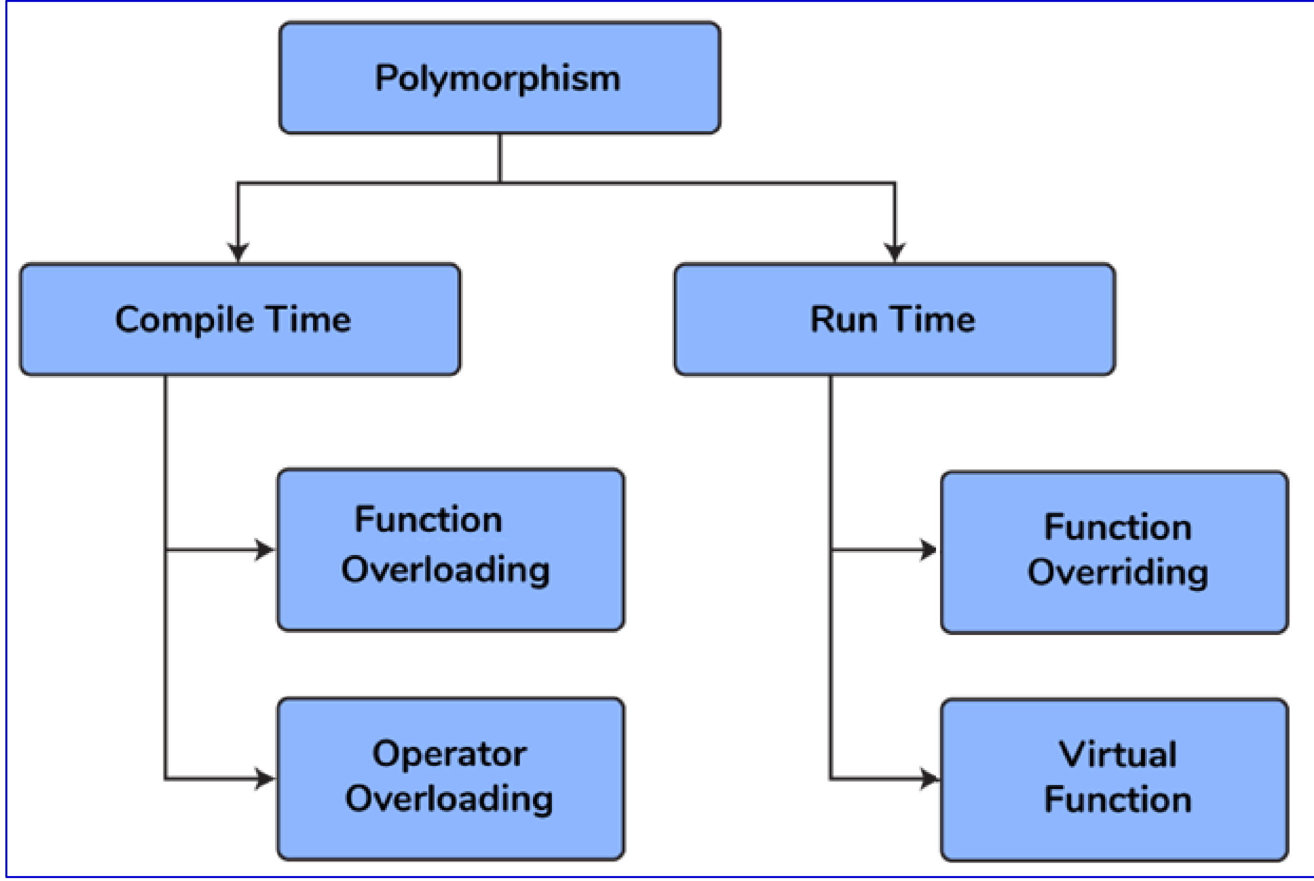
<u>Different ways to achieving polymorphism in C++ program:</u>

I) Compile Time (static)

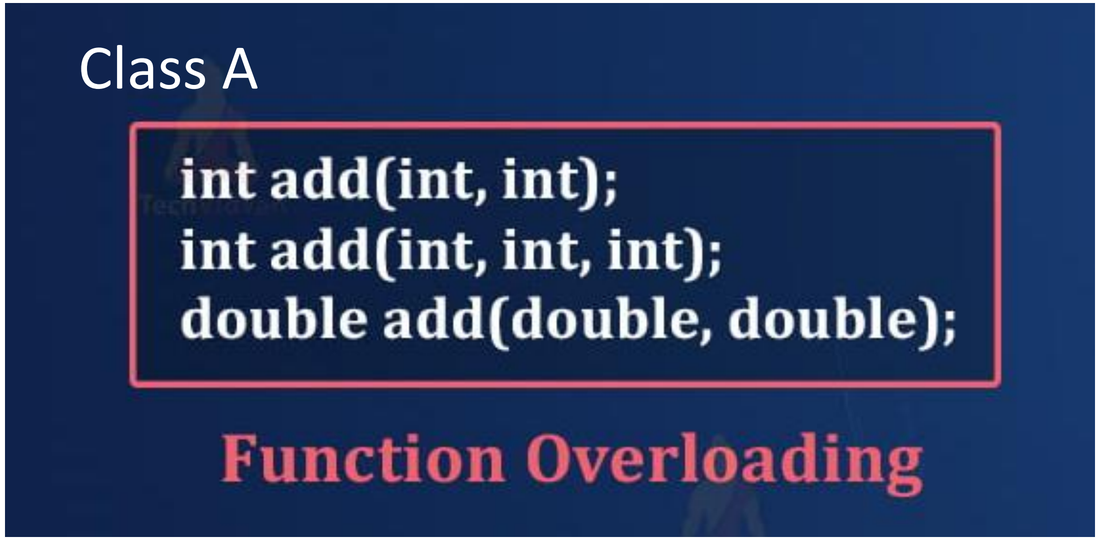
Function Overloading, Operator Overloading

II) Runtime (Dynamic)

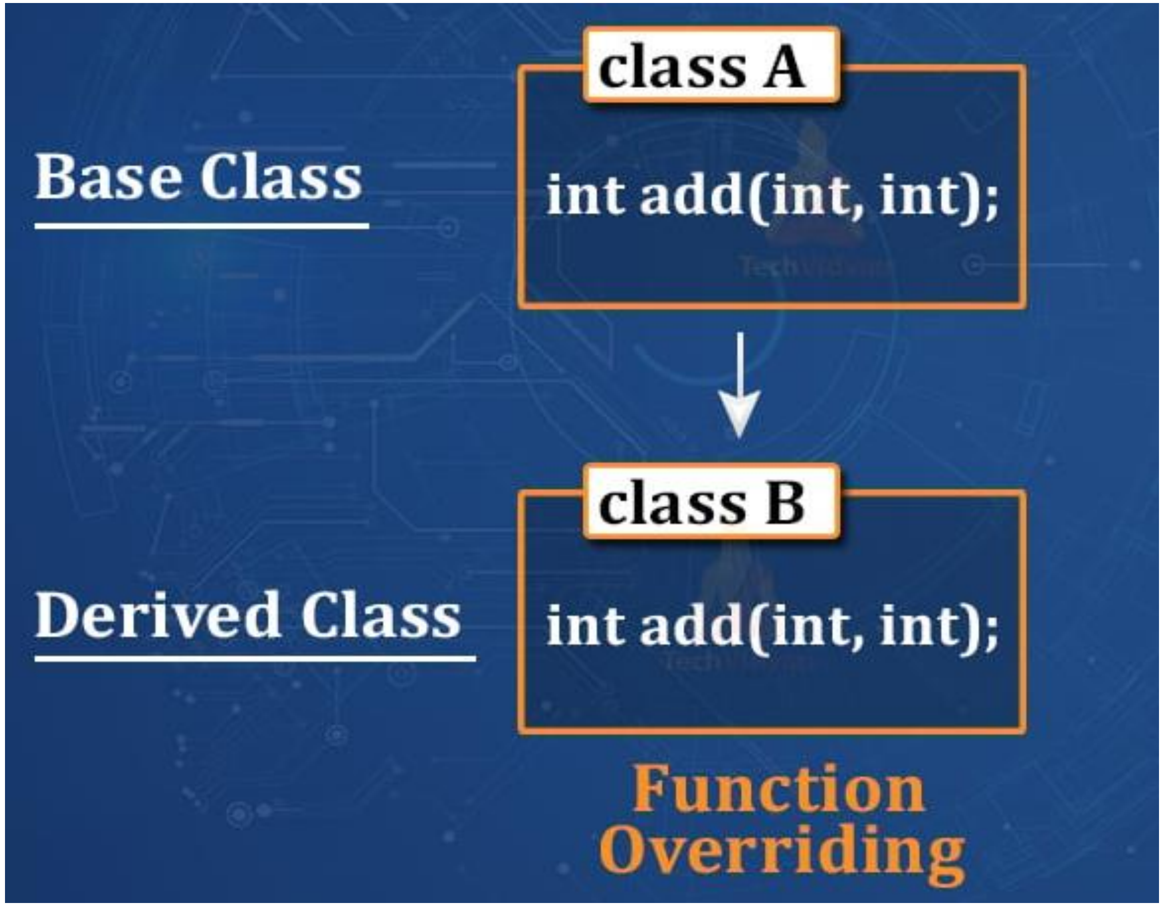
Function Overriding/ Virtual Functions



Function overlong vs Function Overriding



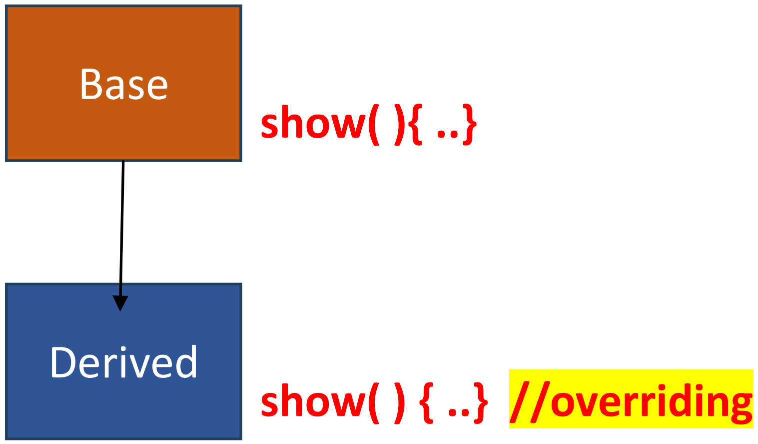
Two or more functions can have the same name , the number and/or type of parameters are different



When a member function of a base class is redefined in its derived class with the same parameters and return type, it is called FUNCTION OVERRIDING

Function Overriding

- When a member function of a base class is redefined in its derived class with the same parameters and return type, it is called FUNCTION OVERRIDING



Requirements for Overriding a Function

1. Inheritance should be there.
2. Function overriding cannot be done within a class.
3. For this we require a derived class and a base class.
4. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

```
class Base
{   public:
   void show()   {
       cout << "Base class";
   }
};

class Derived : public Base
{   public:
   void show()   { //overriding show() in derived
       cout << "Derived Class";
   }
};

int main()
{             Base b;
              b.show(); //calling show() of Base
              Derived d;
             d.show(); // calling show() of Derived
              return 0; }
```

```cpp
// Polymorphic Animal Sounds- function overriding
class Animal { //base class
public:
    // Virtual function to be overridden by derived classes
    virtual void makeSound()  {
        std::cout << "Animal makes a generic sound." << std::endl;
    }
};
class Dog : public Animal {  // Derived class 1
public:
    // Override the makeSound function from the base class
    void makeSound()  {
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }
};
class Cat : public Animal {  // Derived class 2
public:
    // Override the makeSound function from the base class
    void makeSound() {
        std::cout << "Cat says: Meow!" << std::endl;
    }
};
```

```cpp
int main()
{
    // Create objects of base and derived classes
    Animal genericAnimal;
    Dog myDog;
    Cat myCat;

    // Call the makeSound function on each object
    std::cout << "Generic Animal: ";
    genericAnimal.makeSound();

    std::cout << "My Dog: ";
    myDog.makeSound();

    std::cout << "My Cat: ";
    myCat.makeSound();

    return 0;
}
```

“Transport Management System“ – using function overriding

You're developing a C++ **transportation management system**, where **vehicles** are represented in a class hierarchy derived from a base class **Vehicle**. Users interact with the system to start vehicle engines.

How would you design a user interface for users to select and start engines of specific vehicles, like cars and bicycles? Explain the execution flow when a user starts a car's engine, considering function overriding.

Describe modifications to integrate a new vehicle type, handle invalid user input, and customize output for each vehicle type.

Input:

User selects a vehicle type (e.g., car, bicycle) to start the engine.

Output:

Generic Vehicle: Starting the engine of a generic vehicle.
My Car: Starting the engine of a car.
My Bicycle: Pedaling a bicycle.

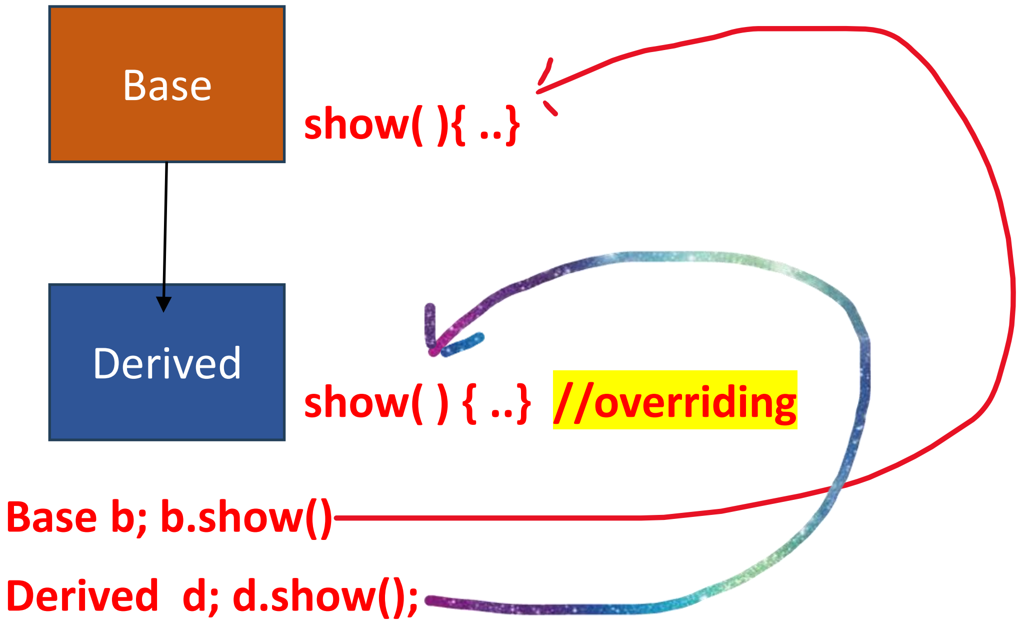
```cpp
class Vehicle {
public:
    // Virtual function to be overridden by derived classes
    virtual void start() const {
        std::cout << "Starting the engine of a generic vehicle." << 
std::endl;
    }
};
// Derived class 1
class Car : public Vehicle {
public:
    // Override the start function from the base class
    void start()  { //overriding
        std::cout << "Starting the engine of a car." << std::endl;
    }
};
// Derived class 2
class Bicycle : public Vehicle {
public:
    // Override the start function from the base class
    void start()  { //overriding
        std::cout << "Pedaling a bicycle." << std::endl;
    }
};
```

```cpp
int main() {
    // Create objects of base and derived classes
    Vehicle genericVehicle;
    Car myCar;
    Bicycle myBicycle;

    // Call the start function on each object
    std::cout << "Generic Vehicle: ";
    genericVehicle.start();

    std::cout << "My Car: ";
    myCar.start();

    std::cout << "My Bicycle: ";
    myBicycle.start();

    return 0;
}
```

Function Overriding

Different cases of calling functions in overriding:

- **Case1**: Calling Base class function with Base class object and Derived class function with Derived object.
- **Case 2**: Calling Base class Function using Derived Class Object
- **Case 3**: Calling Overriding function using pointer of Base type that points to an object of Derived class

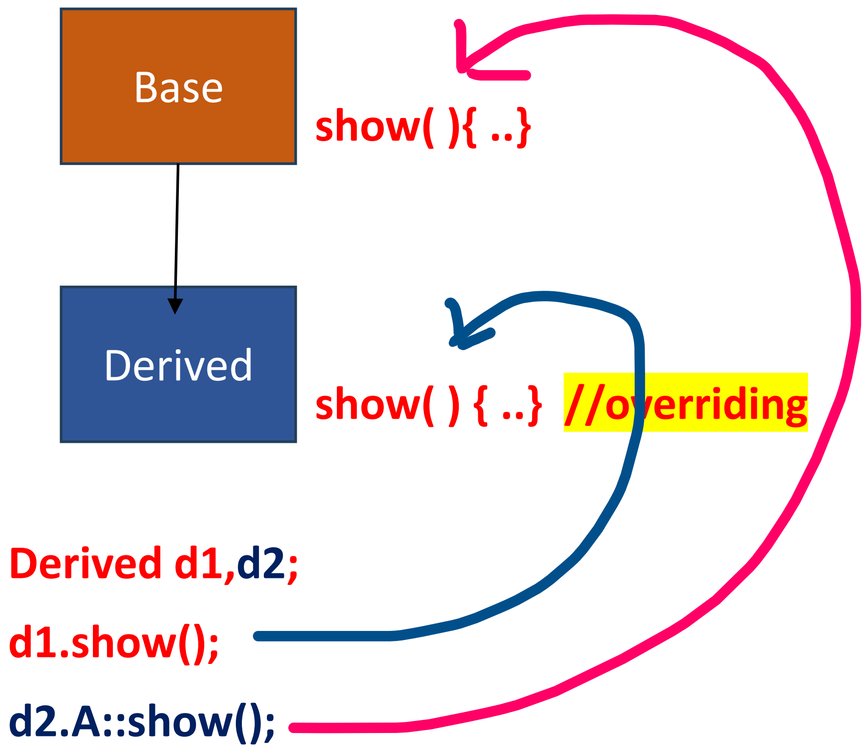



```
// Case1: Calling Base class function with Base class object
and Derived class function with Derived object.
class Base
{   public:
    void show()   {
        cout << "Base class";
    }
};

class Derived : public Base
{   public:
    void show()   { //overriding show() in derived
        cout << "Derived Class";
    }
};

int main()
{           Base b; // Base class object
            b.show(); //calling show() of Base
            Derived d;  //derived class object
           d.show(); // calling show() of Derived
            return 0; }
```

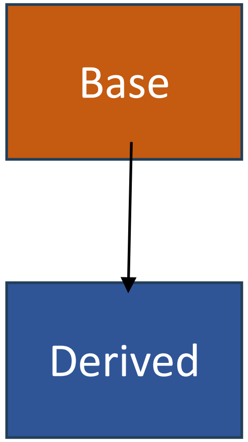
Case 2: Calling Base class Function using Derived Class Objects

```
// calling base class function using derived class by ::
class Base
{
   public:    void show()    {
      cout << "Base class";
   }
};
class Derived : public Base
{
  public:    void show()    { //overriding show() in derived
      cout << "Derived Class";
   }
};
int main()
{              Derived d1,d2;
             d1.show(); // calling show( ) of Derived
             d2.A::show() //calling show( ) of Base
              return 0; }
```

Output: Derived Class
Base class



Case3: Calling Base class Function using Base pointer that holds Derived Class Object



show( ){ ..}

show( ) { ..} //overriding

Derived d;
d.show();

Base *ptr=&d; //ptr of Base type points to Derived
ptr->show();

```
/* Calling Overriding function using pointer
 of Base type that points to an object of Derived class */

class Base {
public:     void show( ){
                              cout << "Base Function" << endl;
              }
};
class Derived : public Base {
public:     void show( ){
                              cout << "Derived Function" << endl;
              }
};
int main()
{
  Derived d; // creating derived class object
  d.show(); //calls derived class show()

  Base *ptr=&d; //ptr of Base type points to Derived
  ptr->show();  //call base class show()
 return 0;
}
```

Static vs. Dynamic Binding

Binding:

The **determination** of which method in the class **hierarchy** is to be invoked for a particular object.

Static (Early) Binding occurs at compile time:

- When the **compiler** can determine which method in the class hierarchy to use for a particular object.

Dynamic (Late) Binding occurs at run time:

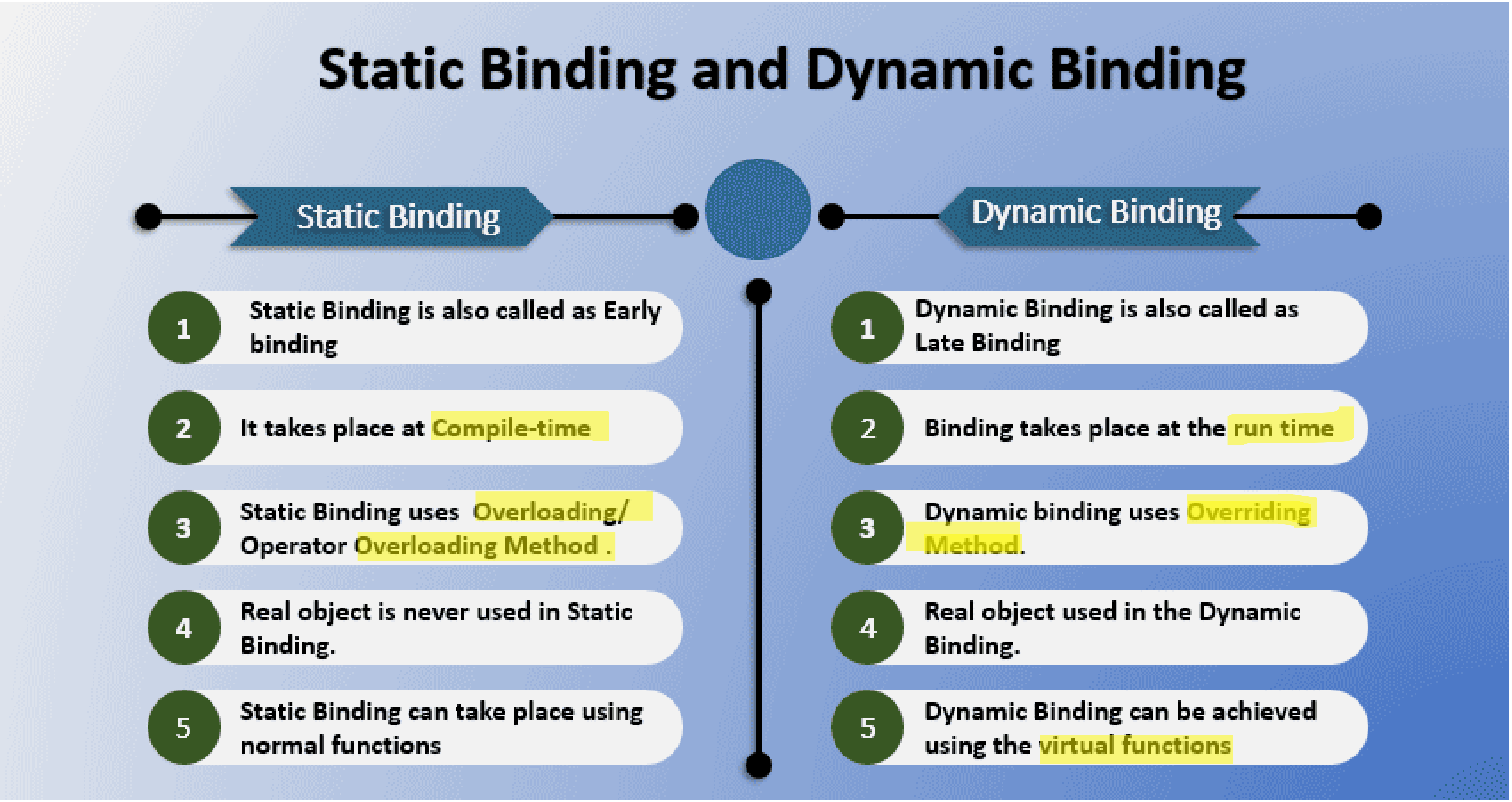
- When the **determination** of which method in the class hierarchy to use for a **particular object occurs during program execution.**

Static (Early) Binding

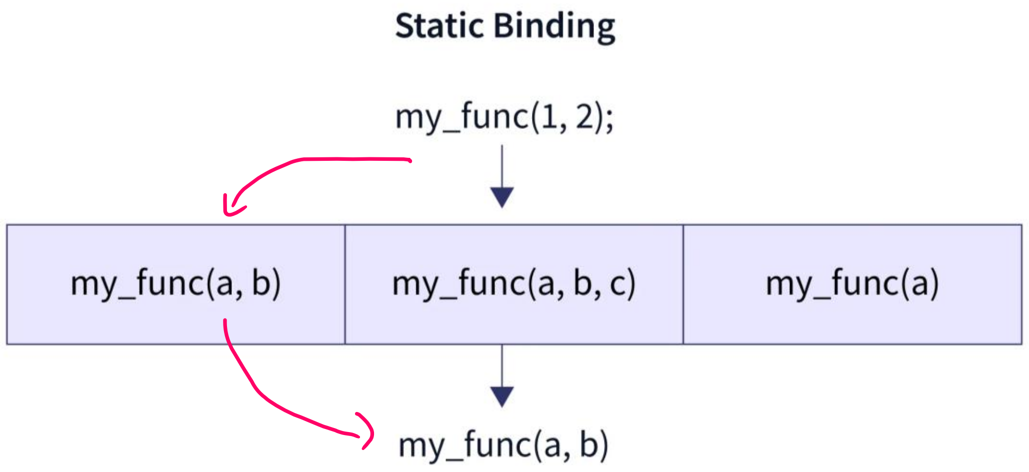
```
Time t1;
ExtTime et2;

t1.setTime(12, 30, 00);          // static binding
et1.setExtTime(13, 45, 30);    // static binding

t1.printTime( );        // static binding – Time's printTime( )
et1.printTime();        // static binding – ExtTime's printTime( )
```

Dynamic (Late) Binding occurs at run time:

- Compiler cannot determine binding of object to method.
- **Binding is determined dynamically at runtime.**
- To indicate that a method is to be bound dynamically, the base class must use the reserved word **virtual**
- When a **method is defined as virtual**, all **overriding methods** from that point on down the hierarchy are virtual, even if not explicitly defined to be so

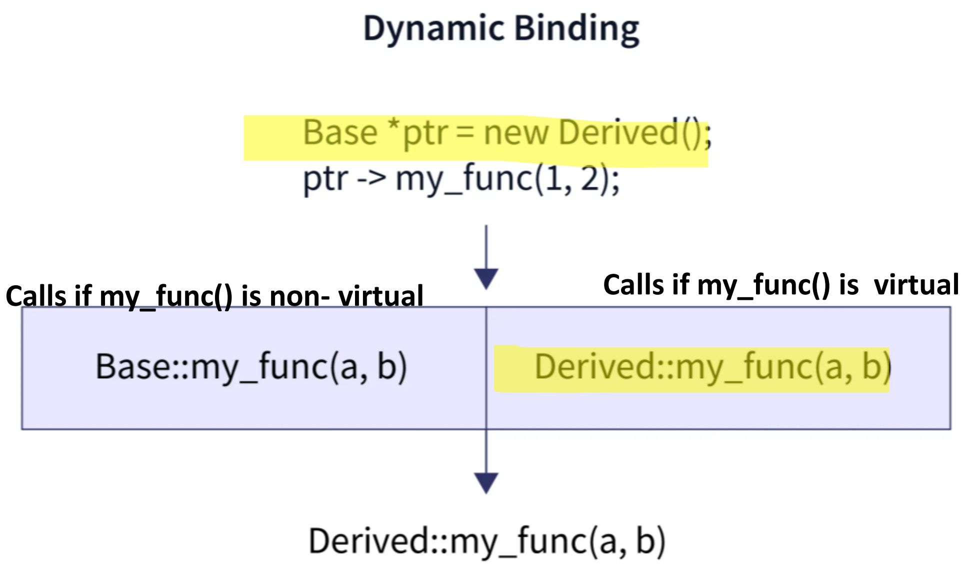

Static Binding and Dynamic Binding
Static Binding
Dynamic Binding
1
Static Binding is also called as Early binding
2
It takes place at Compile-time
3
Static Binding uses Overloading/ Operator Overloading Method .
4
Real object is never used in Static Binding.
5
Static Binding can take place using normal functions
1
Dynamic Binding is also called as Late Binding
2
Binding takes place at the run time
3
Dynamic binding uses Overriding Method.
4
Real object used in the Dynamic Binding.
5
Dynamic Binding can be achieved using the virtual functions

Binding at compile time is known as static binding.



- Static binding ensures linking the function call and its function definition at compile-time only.
- It is also why it is synonymous with compile-time binding or early binding.

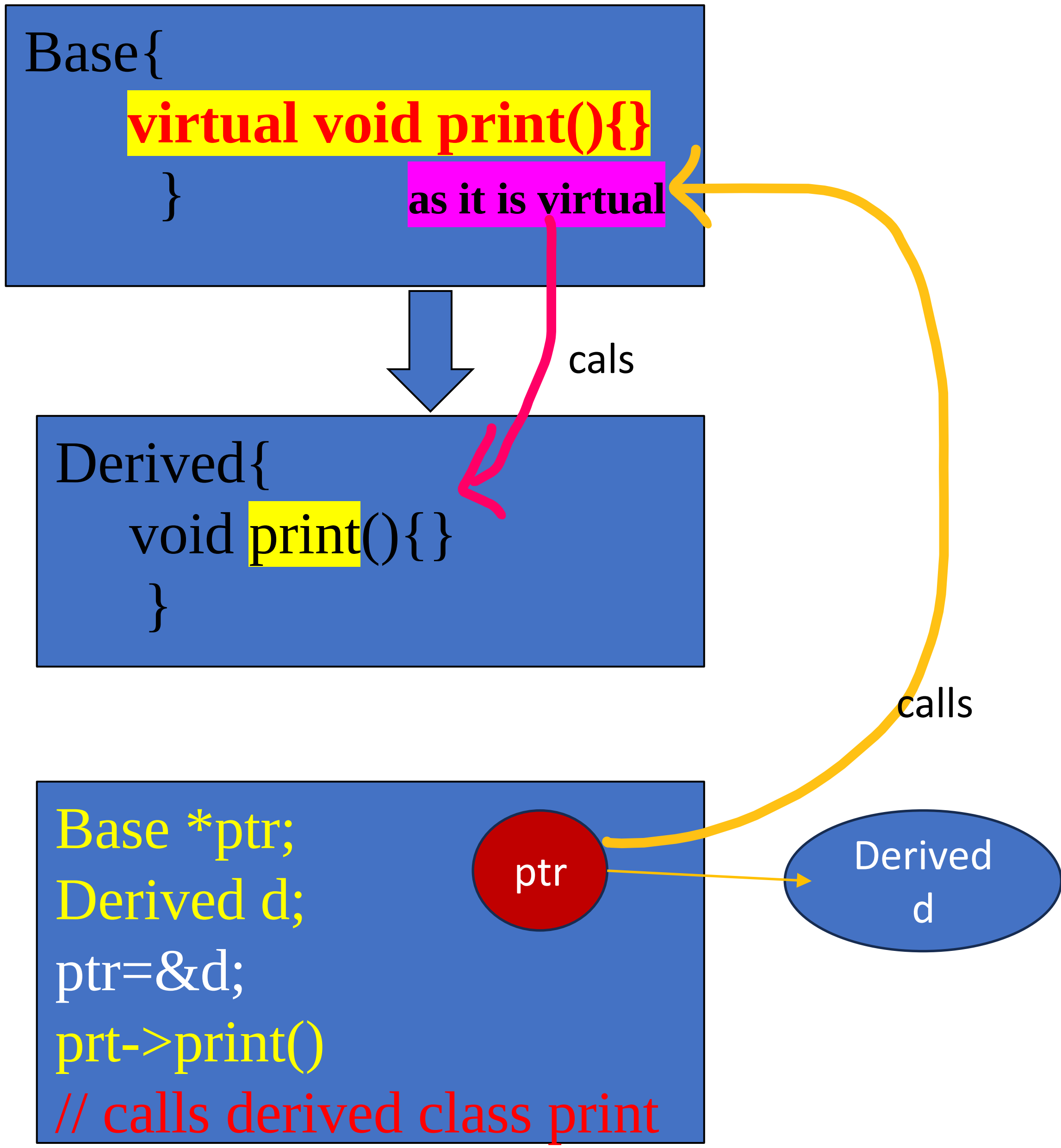
Binding at runtime is known as dynamic binding.



- There are instances in our program when the compiler cannot get all the information at compile time to resolve a function call. These function calls are linked at runtime .Such a process of binding is called dynamic binding. Since everything is postponed till runtime, it is also known as run-time binding or late binding.
- It is executed using virtual functions in C++. A virtual function is a member function in the base class & overridden (re-defined) by its derived class(es)

Virtual Function

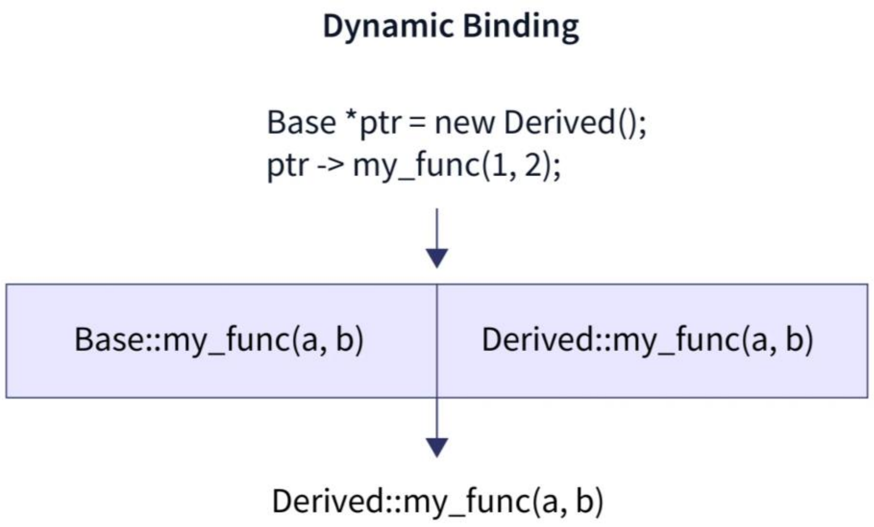
- A **member function** of a **base class** and is **RE-DEFINED in a derived class.**
- Refer to a **derived class object** using a pointer to the **base class, and** call a **virtual function** for that object and **execute the derived class's version of the method.**
- Virtual functions ensure that 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**



Virtual Function

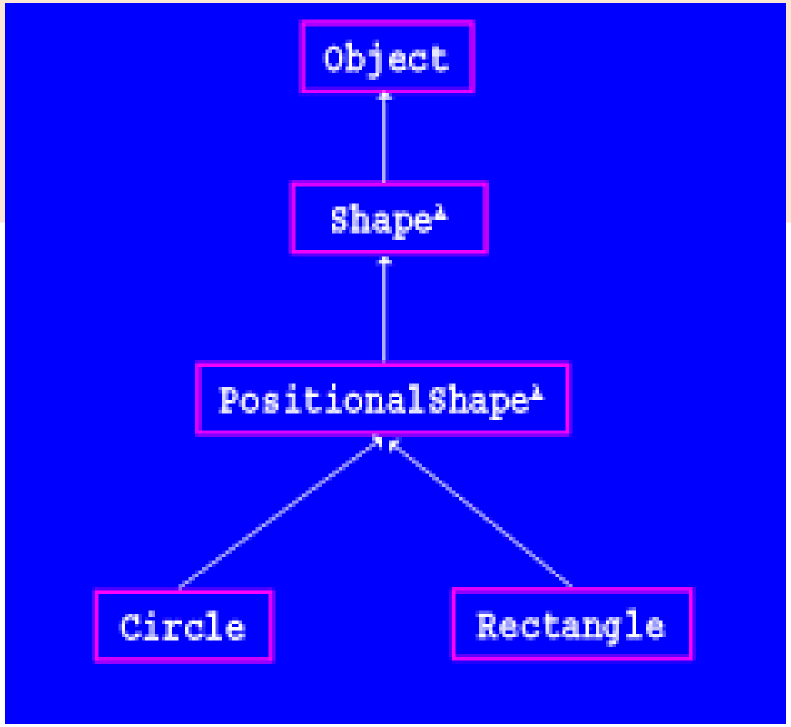
```
class Base {
public:
        virtual void print() {        //virtual function
            cout << "print base class\n";
          }
    void show() { cout << "show base class \n"; }  //non virtual function
};
class Derived : public base {
public:     void print() {
          cout << "print derived class\n"; }
      void show() { cout << "show derived class\n"; }
};
int main()
{     Base* ptr;   Derived d;
      ptr = &d;
      ptr->print(); //Virtual function, binded at runtime
      ptr->show(); // Non-virtual function, binded at compile time
     return 0;
}
```

Output:
print derived class
show base class



```cpp
//VIRTUAL FUNCTIONS
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing a Circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() {
        cout << "Drawing a Rectangle" << endl;
    }
};
```

```cpp
int main()
{
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw();
    shape2->draw();

    delete shape1;
    delete shape2;

    return 0;
}
```



Virtual Functions in Payroll System

You are developing a **payroll system** for a company that employs various types of employees, including **full-time**, **part-time**, and **contract** workers.

Each type of employee has a different method for calculating their pay.

You've used **virtual functions** to handle pay calculations for different employee types.

Implement this using **virtual functions** in the payroll system benefits code organization and maintenance.

OUTPUT:
Payroll Information:
Employee Name: Virat
Pay: Rs 55000.00

Employee Name: Rohith
Pay: Rs.6600.00

Employee Name: Shubhman
Pay: Rs.33000.00

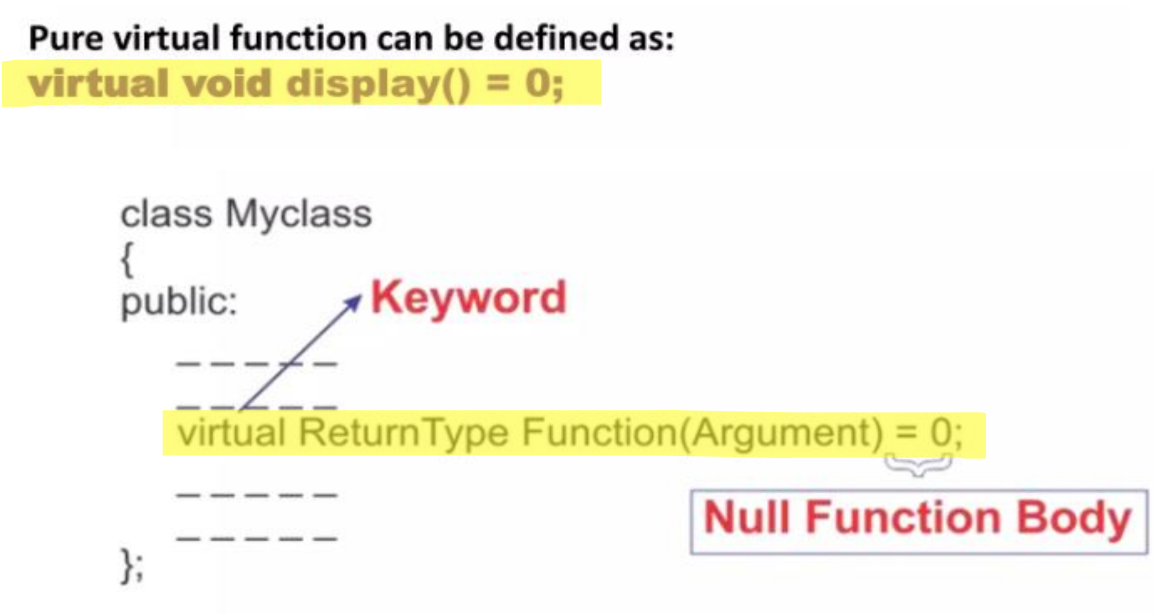
```cpp
//CREATING payroll system using VIRTUAL FUNCTION
#include<iostream>
using namespace std;
class Emp
{
    public:
      string name;float sal;
            virtual void read(){ } //virtual function
            virtual void pay(){ } //virtual function
};
class FullTimeEmp:public Emp
{public:
void read()
{
cout<<"Enter Full time emp details: name and sal";
cin>>name>>sal;
 }
void pay()
{ cout<<"name=:"<<name<<"sal="<<sal;
}
};
```

```cpp
class PartTimeEmp:public Emp
{
public: float hrs,rate;
void read(){
cout<<"Enter part time emp details: name and sal";
cin>>name>>sal;

cout<<"Enter hrs worked and age";
 cin>>hrs>>rate;
 }
void pay()
{       sal=hrs*rate;
                cout<<"name=:"<<name<<"sal="<<sal;}
};

int main()
{
  Emp *ft=new FullTimeEmp();
  Emp *pt=new PartTimeEmp();
 ft->read(); ft->pay(); //calls fulltime emp read() and pay()
 pt->read(); pt->pay(); //calls parttime emp read() and pay()
 return 0;
}
```

Pure Virtual Functions and Abstract Classes in C++

PURE VIRTUAL FUNCTION:

- A pure virtual function is **a virtual function with out any implementation.**
- When a function has no definition such function is known as “**do-nothing**” function.
- A **pure virtual function** is declared by assigning a zero (0) in its declaration.



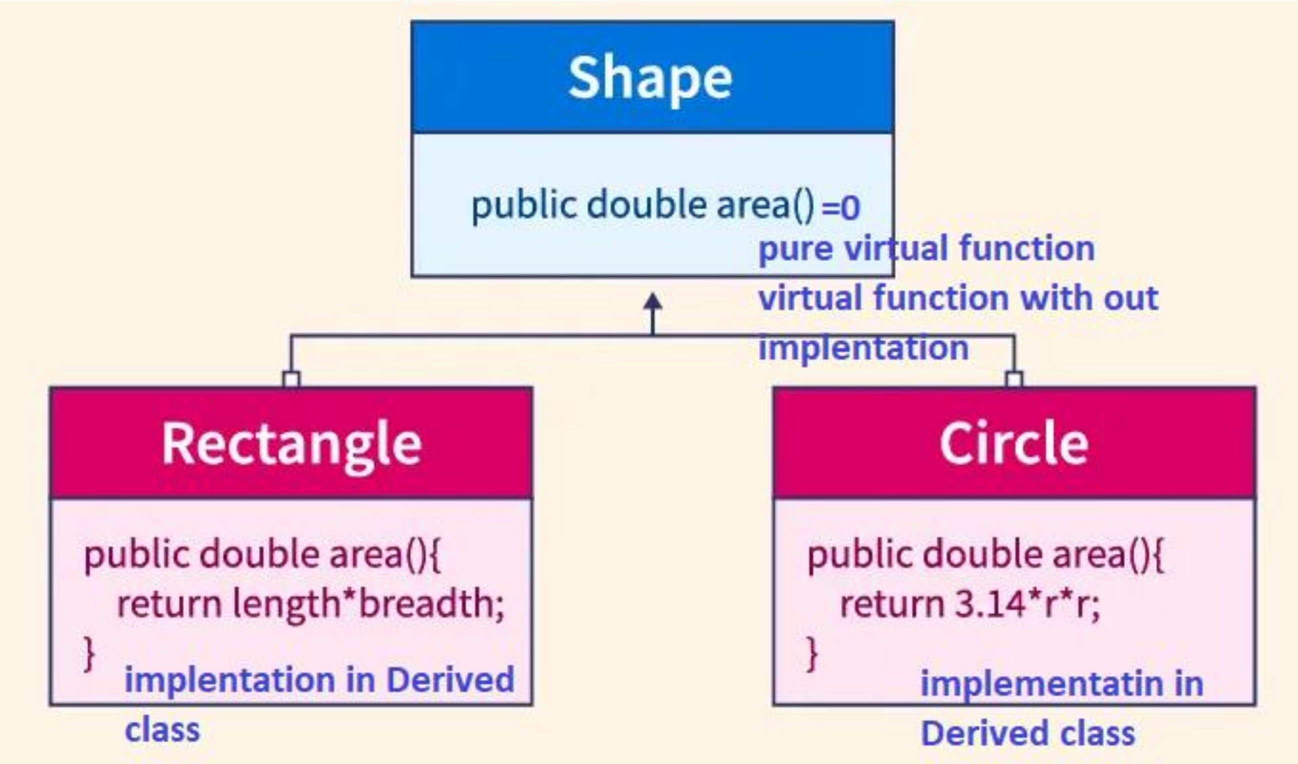
ABSTRACT CLASS

- Any class with one or more pure virtual functions is called as **Abastract class**.
- We can’t create the object for **Abstract class**.
- The main objective of it is to provide traits(behavior) to the derived classes .
- Used for achieving **Runtime polymorphism**.

```
class Test
{
        // Data members of class
public:
        // Pure Virtual Function
        virtual void show() = 0;
        /* Other members */
};
```

Abstract Classes in C++

Abstract class contains at least one pure virtual function, which cannot be instantiated. Such classes are mainly used for Upcasting, which means that its derived classes can use its interface.



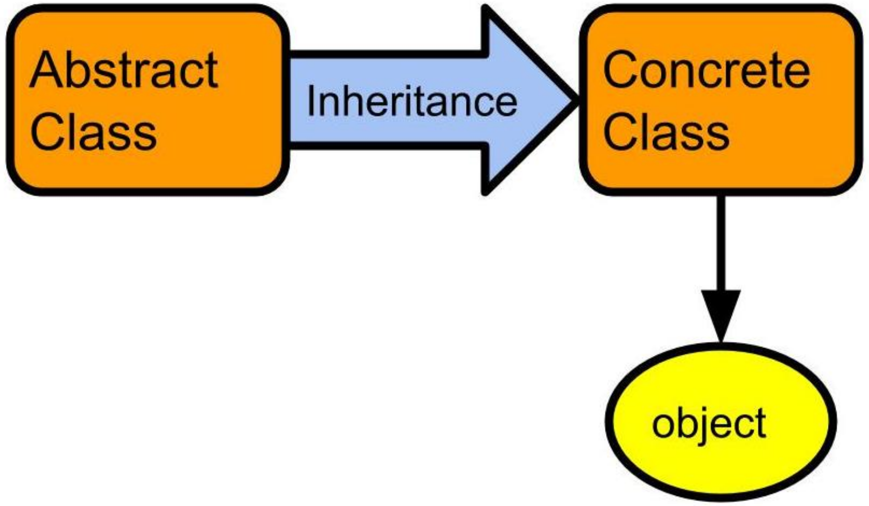

```
class Shape {
public:
    virtual double area() const = 0;
};
class Circle : public Shape {
public: double radius;
   Circle(double r) : radius(r) {}
     double area()  {
       return 3.14159 * radius * radius;
   }
};
class Rectangle : public Shape {
public: double length;  double width;
Rectangle(double l, double w) : length(l), width(w) {}
double area() {
       return length * width;
   }
};
int main() {
 Circle circle(5.0);    Rectangle rectangle(4.0, 6.0);
cout << "Circle Area: " << circle.area();
cout << "Rectangle Area: " << rectangle.area();    return 0;
}
```

```cpp
//CRETATING ABSTRACT CLASS
#include <iostream>
using namespace std;
class Abstract
{
int a, b;
public:
virtual void readData()=0;//pure virtual function
virtual void printData()=0; //pure virtual function

};

class Derived : public Abstract {
public:
Void readData()
{
    cout << " Enter a and b: ";
    cin>>a>>b;
}
void printData()
{
cout << "Derived a = " << a << endl
Cout << "Derived b = " << b << endl;
}
};
```

```cpp
 int main()
{
// Abstract a;
// Cannot create an instance of Abstract Class
Derived d;
d. readData(10, 20);
cout <<  Data  in derived class" << endl;
d.printData();

}
```

Abstract Class

class that *cannot* be instantiated directly



Shape drawing Application using Abstract class

Consider a scenario where you are developing a **shape** drawing application.

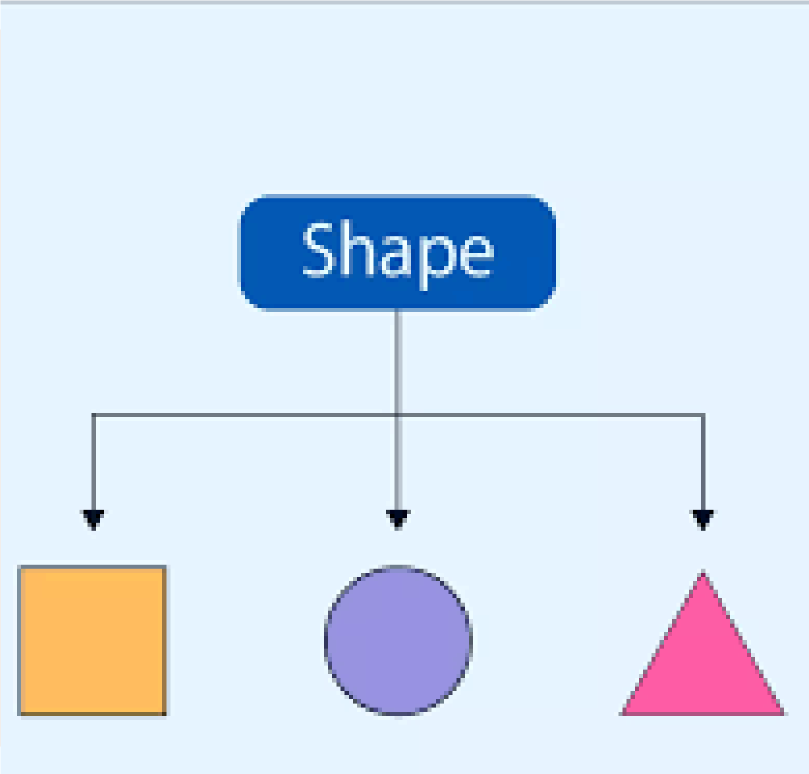
Extend the existing code to create an abstract class **Shape** with attributes **color** .

Derive classes like **Circle** and **Square** from the abstract class, implementing functions to **draw each shape** on the screen

INPUT AND OUTPUT:

Drawing Shapes:
Enter the radius and color of the circle
5
Red
Drawing a Red circle with radius 5

Enter side length and color of the square:
8
Blue
Drawing a Blue square with side length 8



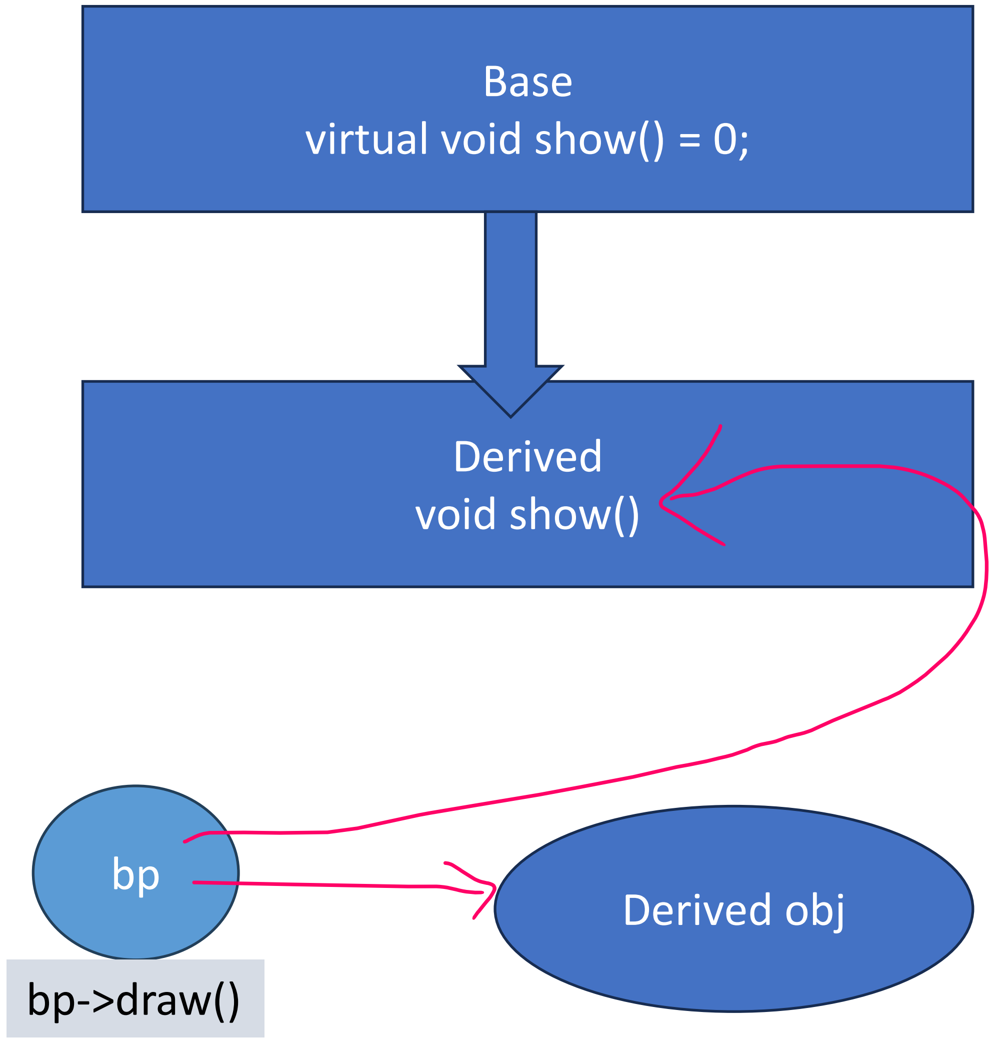
```
//CREATING ABSTRACT CLASS WITH PURE VIRTUAL FUNCTIONS
#include <iostream>
using namespace std;

class Shape //abstract class .i.e. having atleast one pure virtual
function
{    public:  string color;
     virtual void readDimension()=0; //pure virtual function.
     virtual void draw()=0; //pure virtual function
};

class Circle:public Shape
{               public: int r;
  void readDimension()
         {
          cout<<"Enter the radius and color of the circle \n";
          cin>>r>>color;
        }
    void draw()
      {
    cout<<"Drawing a "<<color<<" circle with radius="<<r<<endl;
      }
};
```

```
class Square:public Shape
{  public:    int s;
    void readDimension()
     {
     cout<<"Enter side  and color of the square \n";
     cin>>s>>color;
    }
  void draw()
   {
       cout<<"Draiwng a "<<color<<" square with
      side="<<s<<endl;}
};

int main()
{   //Abstract a;   can't create obj for abstract class
            Circle c;
            c.readDimension();
            c.draw();
              Square s;
              s.readDimension();
              s.draw();
    return 0;
}
```

//pointers and references of abstract class type.

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

class Base {
public:
        virtual void show() = 0;
};

class Derived : public Base {
public:
        void show() {
           cout << "In Derived \n";
          }
};

int main( )
{
        Base* bp = new Derived();
        bp->show();
        return 0;
}
```

Output: In Derived



//pointers and references of abstract class type.

```
class Animal {
public:
    virtual void makeSound() const = 0;
};
class Dog : public Animal {
public:
    void makeSound() const override
    {
        cout << "Woof! Woof!\n";
    }
}
};
class Cat : public Animal
{
public:
    void makeSound() const override
   {
        cout << "Meow!\n";
    }
};
```

```
int main()
{
    Animal* animals[2]; //creating  2 pointers of Abstract class

    Dog myDog; //Creating Dog object
    Cat myCat; //Crating Cat object

    animals[0] = &myDog;
    animals[1] = &myCat;

    animals[0]->makeSound(); //Woof! Woof!
    animals[1]->makeSound();// Meow!

    return 0;
}
```

//combining the two statements from two classes

Output:
Woof! Woof!
Meow!

```
// Abstract class with constructors
class Shape {
public:
   // Constructor in the abstract class
   Shape(int x, int y) : x(x), y(y) {}
      virtual void draw() const = 0; //pure virtual function
      void move(int newX, int newY) {
       x = newX;        y = newY;
       cout << "Shape moved to x=" << newX << " y=" << newY ;
   }
protected:    int x, y;
};
class Circle : public Shape {
public:
       Circle(int x, int y, int radius) : Shape(x, y), radius(radius) {}
 void draw()
 {
  cout << "Drawing a circle at x=" << x << " y=" << y << " with
  radius=" << radius;
   }
private:    int radius;
};
```

```
int main()
 {

   Circle circle(5, 10, 8);

    // Calling methods on the object
    circle.draw();
// Calls the draw method in the Circle class


    circle.move(8, 15);
// Calls the move method in the Shape class

    return 0;
}
```

```
// Abstract class BankAccount
class BankAccount {  //abstract class
public:
   // Constructor with initial balance
  BankAccount(double initialBalance) : balance(initialBalance) {}
   // Abstract methods for deposit and withdrawal
   virtual void deposit(double amount) = 0;
   virtual void withdraw(double amount) = 0;
    double getBalance() const {       return balance;     }
protected:     double balance;
};
// Concrete subclass SavingsAccount
class SavingsAccount : public BankAccount {  //savings account
public:
       SavingsAccount(double initialBalance, double interestRate)
       : BankAccount(initialBalance), interestRate(interestRate) {}
void deposit(double amt)  {
       if (amt > 0) {
           balance += amt;
           cout << "Deposited: Rs." << amt;
       } else {
           cout << "Invalid deposit amount.";
       }
   }
```

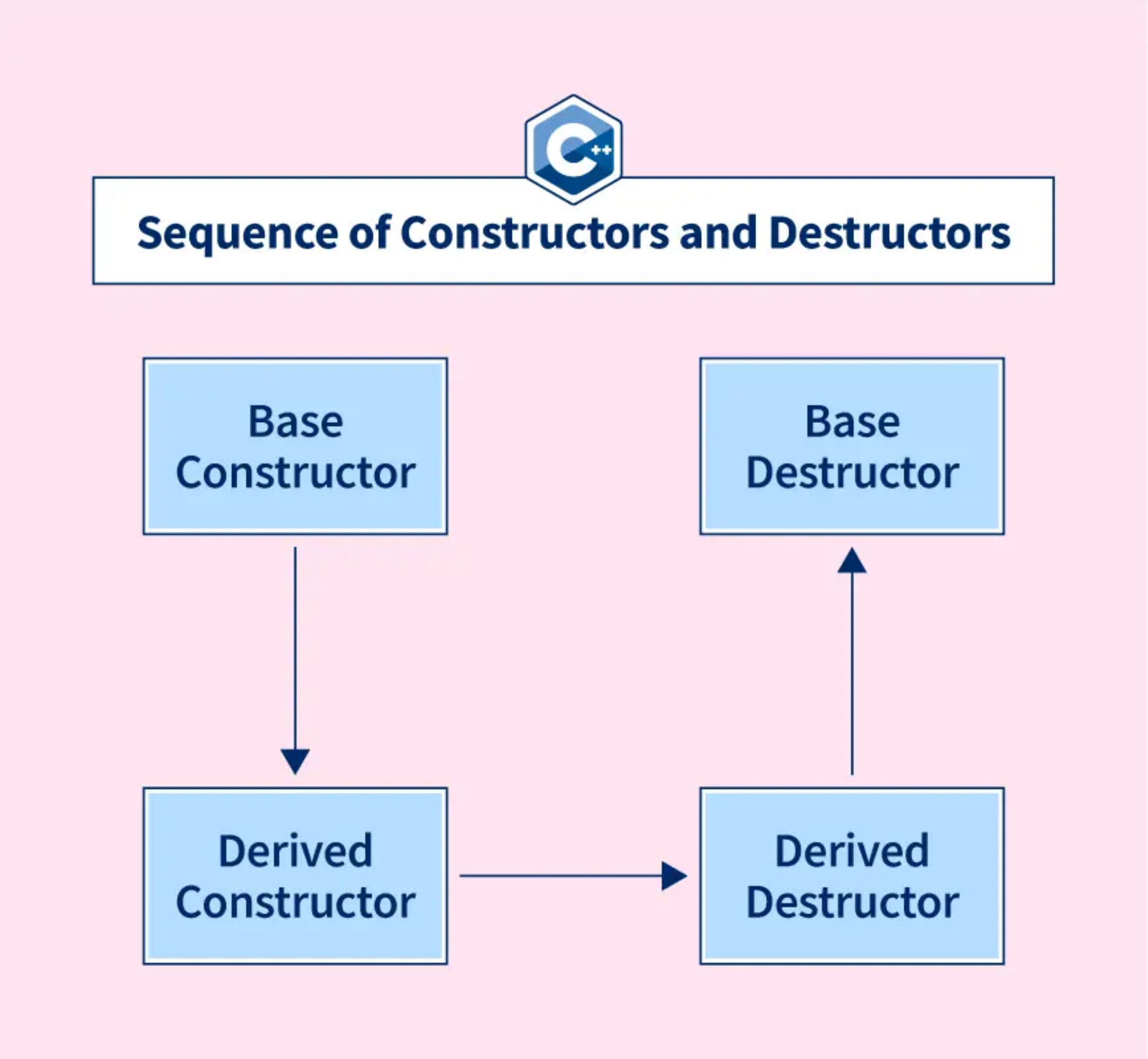
```
     void withdraw(double amt) override {
       if (amount > 0 && balance >= amt) {
          bal -= amt;
         cout << "Withdrawn: Rs." << amt;
       } else {
         cout << "Invalid withdrawal amount or insufficient bal.";
       }
    }
void calculateInterest() {
       double interest = bal * rate / 100;
       bal += interest;
       cout << "Interest calculated and added: Rs." << interest;
    }
private:     double rate;
};
int main()
{
   SavingsAccount sa(1000, 5);
  sa.deposit(500); sa.withdraw(200); sa.calculateInterest();
  cout << "Final Balance: Rs." << sa.getBalance();
   return 0;
}
```

```cpp
//creating abstract class
class Shape //abstract class
{
    protected:
    int width, height;
    public:
    void set_values (int a, int b)
    {
        width = a; height = b;
    }
    virtual int area() = 0;
};
class Rectangle: public p
{
    public:
    int area (void)
    {
        return (width * height);
    }
};
```

```cpp
class Triangle: public p
{
    public:
    int area (void)
    {
        return (width * height / 2);
    }
};
int main ()
{
    Rectangle rect;
    Triangle trgl;
    p *ptr1 = &rect;
    p *ptr2 = &trgl;
    ptr1->set_values (4, 5);
    ptr2 ->set_values (4, 5);
    cout << ptr1 -> area() ;
    cout << ptr2 -> area();
    return 0;
}
```

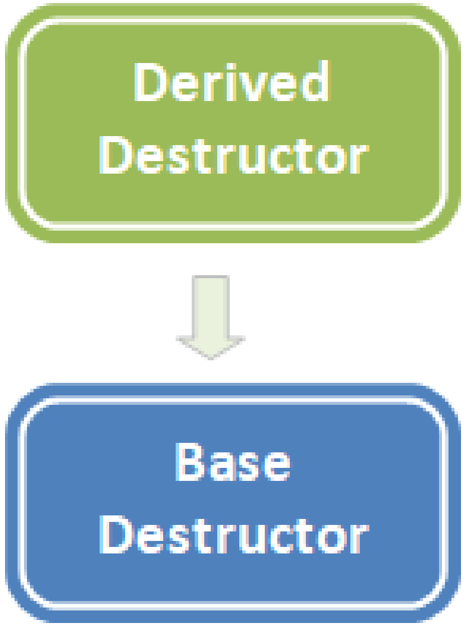
output: 2010

Virtual Destructor in C++



Destructor call sequence

- First derived class will be destroyed and then base class



Sequence of destructor calls

Virtual Destructor in C++

- A **destructor in C++** is a member function of a class **used to free the space occupied by or delete an object** of the class that **goes out of scope**.
- **Deleting a derived class object using a pointer of base class** type that has a non-virtual destructor results in **undefined behavior.**
- To correct this situation, the **base class should be defined with a virtual destructor**.
- A **virtual destructor** is used **to free up the memory space** allocated by the **derived class object** or instance while **deleting instances of the derived class** using a **base class pointer object.**

```
// CPP program without virtual destructor
// causing undefined behavior
class base {
public:
   base(){ cout << "Constructing base\n"; }
   ~base() { cout<< "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived()  { cout << "Destructing derived\n"; }
};
int main()
{   derived *d = new derived();
     base *b = d;
    delete b; //deletes memory for base class object only
    return 0;
}
```

Calls base class destructor

Output

Constructing base
Constructing derived
Destructing base   *//what about Destructing derived?*

Virtual Destructor in C++

- Making base class destructor virtual **guarantees that the object of derived class is destructed properly**, i.e., both base class and derived class destructors are called.

Improper Destruction of objects:

- Parent Destroyed but Child Not destroyed

Proper Destruction of objects:

- **Child destroyed then Parent destroyed**

```
// A program with virtual destructor
class base {
public:
            base() { cout << "Constructing base\n"; }
  virtual ~base()  { cout << "Destructing base\n"; } //VIRTUAL DESTRUCTOR

};
class derived : public base {
public:
    derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived *d = new derived();  //creating derived object refereeing with ptr
base *b = d;
delete b; //deleting the pointer of base holding derived object
return 0;
}
```

Calls base class destructor

Calls derived class destructor

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Resource Management in a Base and Derived Class – virtual destructor

Suppose you are developing a software system that involves handling different types of shapes.
You have a base class called **Shape** with a **virtual destructor**, and you also have a derived class called **Circle** that inherits from Shape.
Both classes manage some dynamically allocated resources, such as memory for storing shape data.
Implement this with a **virtual destructor** and explain what happens when the **delete shapePtr;** statement is executed in the **main** function. Discuss the role of the virtual destructor in this scenario and how it contributes to proper resource cleanup.

OUTPUT:

Constructing Shape
Constructing Circle
Destructing Circle
Destructing Shape

```cpp
//virtual destructor
class Shape {
public:
    Shape() {
        data = new int[10];
        cout << "Shape constructor\n";
    }
    virtual ~Shape() { //virtual destructor
        delete[] data;
        cout << "Shape destructor\n";
    }
private:    int* data;
};
class Circle : public Shape {
public:
    Circle() {
        radius = 5.0;
        cout << "Circle constructor\n";
    }
    ~Circle() override {
        cout << "Circle destructor\n";
    }
private:    double radius;
};
```

```cpp
int main()
 {
    Shape* shapePtr = new Circle();
    delete shapePtr;

    return 0;
}
```