DAY 04

Subtype polymorphism, abstract classes, interfaces

Subtype polymorphism :

• Sub-typing polymorphism in C++ is achieved through inheritance, allowing derived classes to inherit properties and behavior from a base class. Polymorphism is achieved through the use of virtual functions, which allow objects of different types to be treated as objects of the same type

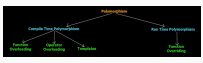
Polymorphism :

- poly : many && morphism : forms like so many function with the same name , polymorphism is an oop concept that refers to the ability of a variable , function , or object to take multiple forms, with Polymorphism , class objects belonging to the same hierarchical tree (inherited from a common parent class)may have functions with the same name, but with different behaviors .

Types of polymorphism :

Compile time Polymorphism : (Static Binding or Early Binding)

- Function overloading: (multiple Functions with the same name but the different parameters) .
- Operator overloading .



Runtime Polymorphism : (Dynamic Binding or Lazy Binding)

- Function Overriding :
  1- multiple Functions totaly the same (using Virtual functions) .
  2- You cannot achieve Runtime polymorphism without using refrence or pointer.

Virtual Function :

- Virtual function is a member function that is declared in the base class and can be overridden in the derived class. Virtual functions are used to achieve runtime polymorphism .
- When a virtual function is called on a base class pointer or reference, the actual function called depends on the type of object pointed to or referred to, rather than the type of the pointer or reference itself. This allows the program to determine at runtime which version of the function to call, depending on the actual type of the object being pointed to or referred to.

Virtual Destructor :

- A virtual destructor is a destructor that is declared as virtual in the base class. If a class has virtual functions, it is recommended that its destructor should also be declared as virtual. This is because if a derived class object is deleted through a pointer to the base class, and the base class destructor is not virtual, then only the base class destructor will be called, and the derived class destructor will not be called. This can result in memory leaks and other undefined behavior.

```
class Base {
public:
    virtual ~Base() {
        // Destructor code goes here
    }
};

class Derived : public Base {
public:
    ~Derived() {
        // Destructor code goes here
    }
};
```

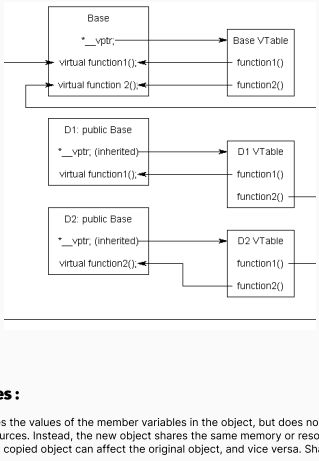
```
int main() {
    Base* b = new Derived();
    delete b;    // Calls the derived class destructor first, then the base class destructor
    return 0;
}
```

Rules for Virtual Functions :

- Virtual functions cannot be static.
- Virtual functions should be accessed using 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 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 virtual destructor but it cannot have a virtual constructor.

Virtual table virtual pointer :

- when a class contains at least one virtual function, the compiler automatically generates a virtual table (also known as a vtable or virtual function table) for that class. The virtual table is an array of function pointers that contains the addresses of the virtual functions for that class.
- In addition, the compiler adds a hidden virtual pointer (also known as a vpointer or virtual function pointer) to each object that has virtual functions. This virtual pointer points to the virtual table for the class of the object.
- When a virtual function is called on an object, the virtual pointer is used to look up the address of the function in the virtual table and then call the function using that address. This allows the program to determine at runtime which version of the function to call, depending on the actual type of the object being pointed to or referred to.



Shallow vs Deep Copies :

- A shallow copy only copies the values of the member variables in the object, but does not create new copies of any dynamically allocated memory or resources. Instead, the new object shares the same memory or resources as the original object. This means that changes made to the copied object can affect the original object, and vice versa. Shallow copy is the default behavior in C++ for most data types.
- A deep copy, on the other hand, creates a new object with copies of all the member variables, including dynamically allocated memory or resources. This means that changes made to the copied object do not affect the original object, and vice versa. Deep copy is typically used when working with dynamically allocated memory or resources that should not be shared between objects.

Shallow Copie

```
class Person{
    int *a;
    public:
    Person(){
        a = new int;
        *a = 0;
    }
    Person(int b){
        a = new int;
        *a = b;
    }
    void  set_v(int b){
        a = new int;
        *a = b;
    }
    Person(const Person&obj){
        a = obj.a;
    }
    ~Person(){
        delete a;
    }
    int get_value(){
        return (*a);
    }
};
```

Deep Copie

```
class Person{
    int *a;
    public:
    Person(){
        a = new int;
        *a = 0;
    }
    Person(int b){
        a = new int;
        *a = b;
    }
    void  set_v(int b){
        a = new int;
        *a = b;
    }
    Person(const Person&obj){
        if (a != nullptr)
            delete a;
        a = new int;
        *a = *obj.a;
    }
    ~Person(){
        delete a;
    }
    int get_value(){
        return (*a);
    }
};
```

```
int main()
{
    Person p1;
    p1.set_v(9);
    Person p2(p1);
    std::cout << p2.get_value() << std::endl;
    std::cout << p1.get_value() << std::endl;
    return 0;
}
```

```
free(): double free detected in tcache 2
```

Abstract Classe :

- An abstract class is a class that has at least one pure virtual function. A pure virtual function is a function that has no implementation in the class and is marked as "pure virtual" by the "= 0" syntax at the end of the function declaration.
- An abstract class cannot be instantiated directly because it has at least one pure virtual function that has no implementation. Instead, it serves as a base class for other classes that derive from it and provide the implementation for its pure virtual functions.

```
class Shape {
public:
  virtual double getArea() const = 0;
};

class Circle : public Shape {
public:
  Circle(double r) : radius(r) {}
  virtual double getArea() const {
    return 3.14159 * radius * radius;
  }
private:
  double radius;
};

class Square : public Shape {
public:
  Square(double s) : side(s) {}
  virtual double getArea() const {
    return side * side;
  }
private:
  double side;
};
```

```
int main() {
  Shape* shapes[2];
  shapes[0] = new Circle(3);
  shapes[1] = new Square(4);
  for (int i = 0; i < 2; ++i) {
    std::cout << "Area: " << shapes[i]->getArea() << std::endl;
    delete shapes[i];
  }
  return 0;
}
```