05/11/2023

Run-time polymorphism

→ Function overriding

In function overriding, derived class provides a specific implementation for a function that is already defined in parent class. This allows the derived class to override the behaviour of the function inherited from parent class.

Code -

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

class animal {
    public:
    void speak() {
        cout << "speaking" << endl;
    }
};
class dog : public animal {
    public:
    void speak() {
        cout << "barking" << endl;
    }
};
int main() {
    animal a;
    a.speak;
    dog d;
    d.speak();
}
```

O/P → speaking
barking

* **Upcasting →** Jb parent class ka pointer child class ke object par create krte hai.

for instance, `parent * a = new child ( );`

* **Downcasting →** Jb child class ka pointer parent class ke object par create krte hai.

for instance, `child * a = (child *) new parent ( );`

= **4 Patterns exists krte hai —**

```
Parent * a = new Parent();
parent * a = new child();
child * a = new child();
child * a = (child*) new Parent();
```

- By default, pointer-type pr depend krega kiske data member ya member function ko access karna hai.

- But, if we want that object-type pr depend kre, then make the parent class virtual.

Note → Agr isi code mai parent class ko virtual bna denge then object-type pr depend krega rather than pointer type. In that case,
output= speaking, barking, barking, speaking

Code –

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

class animal {
    public:
    void speak(){
        cout << "speaking" << endl;
    }
};
class dog : public animal {
    public:
    void speak(){
        cout << "barking" << endl;
    }
};
int main(){

    animal * a1 = new animal();
    a1->speak();

    dog *d1 = new dog();
    d1 -> speak();

    //upcasting
    animal *a2 = new dog();
    a2 -> speak();

    //downcasting
    dog *d2 = (dog*) new animal();
    d2 -> speak();
}
```

Output →

speaking
barking
speaking
barking

Virtual in C++ →

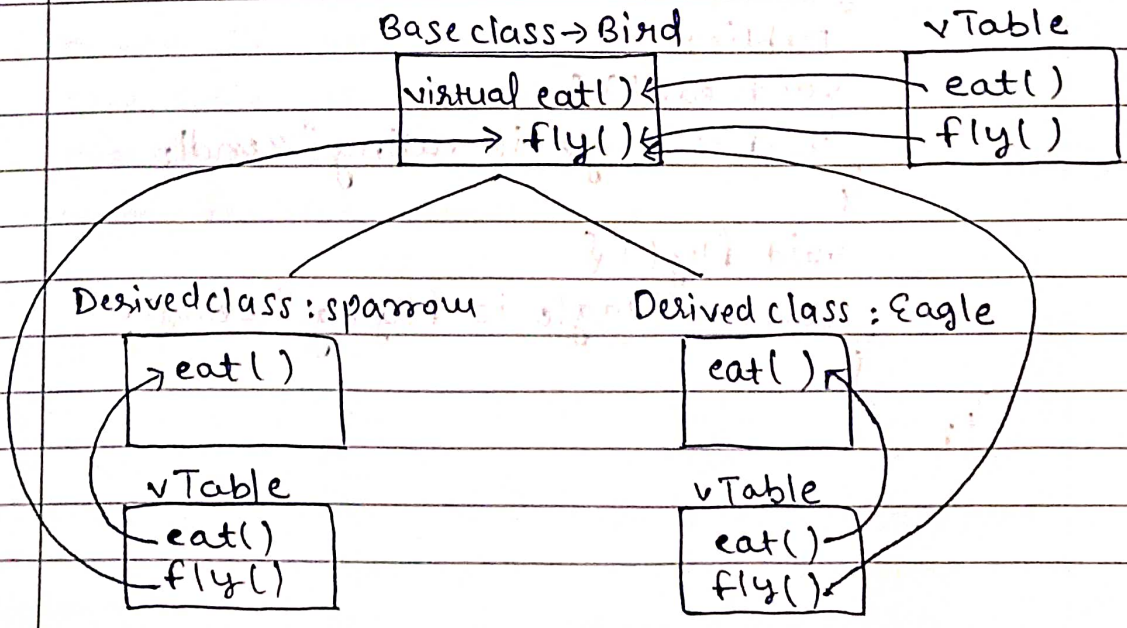
A virtual function is declared in the base class and is over-ridden by the derived class. When we make an object of derived class using a pointer of base class, then derived class implementation of that function is invoked. A function is made virtual to achieve run-time polymorphism.

VTable –

Every class that uses virtual functions has its own virtual table. It is a lookup table used to resolve function calls.

VPtr –

Virtual pointer is a pointer that points to the table of function pointers i.e. vTable. It is a hidden data member to all those class which has virtual function.



Code-

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

class bird {
    public:
    virtual void eat() {
        cout<<"bird is eating"<<endl;
    }
    void fly() {
        cout<<"bird is flying"<<endl;
    }
};
class sparrow: public bird {
    public:
    void eat() {
        cout<<"sparrow is eating"<<endl;
    }
    void fly() {
        cout<<"sparrow is flying."<<endl;
    }
};
class eagle : public bird {
    public:
    void eat() {
        cout<<"eagle is eating"<<endl;
    }
    void fly() {
        cout<<"eagle is flying"<<endl;
    }
};
```

```
int main(){

    bird *bird1 = new sparrow();
     bird1->eat();
     bird1->fly();
    bird *bird2 = new eagle();
     bird2->eat();
     bird2->fly();
}
```

O/P → sparrow is eating
bird is flying
eagle is eating
bird is flying

* **Virtual constructor →**

Constructors cannot be made virtual. When class is executed, there is no vTable in the memory.
Also, constructors have same name as class name. If we make constructors virtual in base class, it means they are re-defined or overrided in child class but we can't have same name for two classes.

Virtual Destructor →

Destructors can be made virtual. One of the reason is vTable is present in memory at the time of destructor calling.
By declaring base class destructor virtual, we ensures that derived class destructors are also called which prevents memory leakage.
Otherwise, if base class destructor is not virtual, then only base class destructor is called which may lead to memory leak.

So, always make base class destructor as virtual.

Code →

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

Class A {
    public:
     A(){
        cout<<"ctor A"<<endl;
     }
    virtual ~A(){
        cout<<"dtor A"<<endl;
     }
};
```

```
class B : public A{
    public:
    B(){
      cout<<"ctor B"<<endl;
    }
   ~B(){
      cout<<"dtor B"<<endl;
    }
};

int main(){
    A * obj1 = new B;
    delete obj1;
}
```

O/P → ctor A
ctor B
dtor B
dtor A

= If base class destructor was not made virtual, then output will be – ctor A
ctor B
dtor A

In that case, derived class ka destructor call nahi hota.

* Abstraction - Implementation Hiding

Showing only essential information. Hiding the rest of things.

Abstraction is the super-set of the other three pillars of OOPS i.e. encapsulation, inheritance and polymorphism.

= Abstract class -

An abstract base class is a class that cannot be instantiated i.e. abstract class ka object create nahi kar skte.
A class is called abstract class if it contains atleast one pure virtual function.
Derived classes that inherits from abstract base class must provide implementation for its abstract functions or methods.

Pure - virtual function -

Is a function that must be overriden in a derived class otherwise that derived class will also become an abstract class.

Syntax -

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

Code -

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

class bird{
    public:
    virtual void eat()=0;
    virtual void fly()=0;
};
class sparrow:public bird{
    private:
    void eat(){
        cout<<"sparrow eats"<<endl;
    }
    void fly(){
        cout<<"sparrow flies"<<endl;
    }
};
class pigeon:public bird{
    private:
    void eat(){
        cout<<"pigeon eats"<<endl;
    }
    void fly(){
        cout<<"pigeon flies"<<endl;
    }
};
int main(){
bird *bird1=new pigeon();
 bird1->eat();
bird *bird2=new sparrow();
 bird2->fly();
}
```

Output -

```
pigeon eats
sparrow flies
```

* **Friend Keyword →**

friend keyword is used to grant a class or function access to the private and protected members of another class.

If I want that a class A can be able to access private members of class B, then it can be done using friend keyword.

Code –

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

class A {
    int x;

    public:
    A(int _x) : x(_x) {}

    int getX() const {
        return x;
    }

    friend class B;
    friend void print(const A &);
};
```

```
//friend class to A

class B {
    public:
    void print (const A &obj) {
        cout << obj.x << endl;
    }
};

// friend function to class A

void print (const A &obj) {
    cout << obj.x << endl;
}

int main( ) {
    A obj1(10);
    B obj2;
    obj2.print (obj1); //calling class B print function
    print (obj1); // calling print function here
}
```

O/P → 10
10

Can Constructors be made private?

Yes, constructors can be made private but we can't create any instance of that class whose constructor is made private.

Code -

```
class A {
    int x;
    A(int _x) : x(_x) {}

};
int main() { }
```

In the above code, constructor is private and this code will not give any compilation error.

Code -

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

class A {
    int x;
    int y;

    A(int _x, int _y) : x(_x), y(_y) { }

    public:
    int getX() {
        return x;
    }
    int getY() {
        return y;
    }
    friend class B; };
```

```
class B {
    public:
    A func (int x, int y){
        return A (x, y);
    }
};

int main( ){
    B obj1;
    A obj2 = obj1. func (1,2)
    cout << obj2. getx( ) << " " << obj2. getY( ) << endl;
}
```

In the above code, class A ka constructor private hai. So, friend class bnake class A ke private members ko access kr skte hai. Yha class B hai friend to class A. So, class B ka instance create krke vha se class A ka instance create kr skte hai.

* **Inline functions –**

Inline functions are optimisation feature just as macros. In inline function, the compiler places a copy of the code of that function at each point where function is called at compile time.

Syntax –

```
returntype
inline ↑ func ( ) {
    // code
}
```

Note- make the function inline only if the code in function is only of one or two lines.

Agr function inline h to call hone pe call stack m entry ni bnegi rather than vo ek line ka code jha se call aayi h vha copy ho jayega at compilation time.