***Object Oriented Programming***

->Inheritance:

#include <iostream>

using namespace std;

class A

{

    int x; // can never be inherited, because it is private member variable

public:

    void get()

    {

        x = 10;

    }

    void print()

    {

        cout << "x=" << x;

    }

};

class B : public A

{

};

int main()

{

    B ob;

    ob.get();

    ob.print();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can never be inherited

public:

    void get()

    {

        x = 10;

        cout<<"A class get()"<<endl;

    }

    void print()

    {

        cout<<"A class print()"<<endl;

        cout << "x=" << x;

    }

};

class B : public A

{

public:

    void fun()

    {

        A::get();   //member functions of base class can be called in this way

        A::print();

    }

};

int main()

{

    B ob;

    ob.fun();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    void get()

    {

        x = 10;

    }

    void print()

    {

        cout << "x=" << x << "y=" << y;

    }

};

class B : public A

{

public:

    void fun()

    {

        // A::x;    //private in base class, cannot be accessed or inherited

        cout<<"the value of y of class A is "<<A::y; //private can be inherited

        A::get();

        A::print();

    }

};

int main()

{

    B ob;

    ob.fun();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    int z;

};

class B : public A //inheriting publicly

{

    int p;

protected: // y will come here

    int q;

public: // z will come here

    int r;

    void print()

    {

        cout << "z=" << z << "r=" << r;

    }

};

int main()

{

    B ob;

    // ob.x=1;  //error

    // ob.y=2;  //error

    ob.z = 3; // ok

    // ob.p=4;   //error

    // ob.q=5;   //error

    ob.r = 6; // ok

    ob.print();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    int z = 1;

};

class B : private A //inheriting privately

{

    int p; // y ans z will come here

protected:

    int q;

public:

    int r = 2;

    void print()

    {

        cout << "z=" << z << "r=" << r;

    }

};

int main()

{

    B ob;

    // ob.x=1;  //error

    // ob.y=2;  //error

    // ob.z=3;  //error

    // ob.p=4;   //error

    // ob.q=5;   //error

    ob.r = 6; // ok

    ob.print();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    int z;

};

class B : protected A //inheriting protectedly

{

    int p;

protected: // y and z will come here

    int q;

public:

    int r;

    void print()

    {

        cout << "z=" << z << "r=" << r;

    }

};

int main()

{

    B ob;

    // ob.x=1;  //error

    // ob.y=2;  //error

    // ob.z=3;  //error

    // ob.p=4;   //error

    // ob.q=5;   //error

    ob.r = 6; // ok

    ob.print();

}

->Multi level inheriting:

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    int z;

};

class B : protected A

{

    int p;

protected: // y and z will come here

    int q;

public:

    int r;

};

class C : public B // Multi-level inheritence is implemented

{

public:

    void fun()

    {

        y = 100;

        z = 200;

    }

};

int main()

{

    C ob;

    // ob.x=1;  //error

    // ob.y=2;  //error

    // ob.z=3;  //error

    // ob.p=4;   //error

    // ob.q=5;   //error

    ob.r = 6; // ok

    // ob.print();

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

protected:

    int y; // they can be inherited, but not accessible outside the class definition

public:

    int z;

    A(int x) //A param constructor

    {

        cout << "within A constructor: \n";

        x = 0;

        y = 1;

        z = 2;

    }

};

class B : public A

{

    int p;

protected: // y  will come here

    int q;

public: // z will come here

    int r;

    B() : A(5)  //to invoke the A param constructor, or it will show an error

    {

        cout << "within B constructor: \n";

        p = 3;

        q = 4;

        r = 5;

    }

};

int main()

{

    B ob;

}

->

#include <iostream>

using namespace std;

class A

{

    int x; // can neither be inherited or be accessed outside the class definition

public:

    A()

    {

        cout<<"within A def-constructor: \n";

    }

    A(int x)

    {

        cout << "within A param-constructor: \n";

    }

};

class B : public A

{

    int p;

public:

    B() : A()

    {

        cout << "within B-def constructor: \n";

    }

    B(int y) : A(y)

    {

        cout << "within B-param constructor: \n";

    }

};

int main()

{

    B ob;

    B ob1(10);

}

// output:

// within A def-constructor:

// within B-def constructor:

// within A param-constructor:

// within B-param constructor:

->Hierarchical Inheritance:

#include <iostream>

using namespace std;

class B

{

    int a;

protected:

    int b;

public:

    B(int a = 0) // it functions both as a def and param constructor

    {

        cout<<"B class constructor"<<endl;

        this->a = a;

    }

    void show()

    {

        cout << "a=" << a << "b=" << b << endl;

    }

    ~B()

    {

        cout << "class B destructing"<<endl;

    }

};

class C1 : public B

{

public:

    C1() : B(5)

    {

        cout<<"C1 class constructor"<<endl;

        b = 10;

    }

    ~C1()

    {

        cout << "class C2 destructing"<<endl;

    }

};

class C2 : public B

{

public:

    C2() // as the Base class constructor functions both as a def and param constructor , we may omit the explicit constructor call to it

    {

        cout<<"C2 class constructor"<<endl;

        b = 20;

    }

    ~C2()

    {

        cout <<"Class C2 destructing"<<endl;

    }

};

int main()

{

    C1 ob;

    ob.show();

    C2 ob2;

    ob2.show();

}

Output:

B class constructor

C1 class constructor

a=5b=10

B class constructor

C2 class constructor

a=0b=20

Class C2 destructing

class B destructing

class C2 destructing

class B destructing

->

#include <iostream>

using namespace std;

class A1

{

public:

A1()

{

    cout<<"class A constructor"<<endl;

}

    void show()

    {

        cout << " within A1 show \n";

    }

};

class A2

{

public:

    A2()

    {

        cout<<"class A2 constructor"<<endl;

    }

    void show()

    {

        cout << " within A2 show \n";

    }

};

class C : public A1, public A2

{

};

int main()

{

    C ob;

    ob.show(); // Error, since show() is inherited from both the base classes, so ambiguity occurs

}

->way out:

#include <iostream>

using namespace std;

class A1

{

protected:

    int a;

public:

    void show()

    {

        cout << " within A1 show "

             << "a=" << a << endl;

    }

};

class A2

{

protected:

    int a;

public:

    void show()

    {

        cout << " within A2 show "

             << "a=" << a << endl;

    }

};

class C : public A1, public A2

{

public:

    void get()

    {

        A1::a = 10;

        A2::a = 20;

    }

};

int main()

{

    C ob;

    ob.get();

    ob.A1::show();

    ob.A2::show();

}

->

#include <iostream>

using namespace std;

class A1

{

protected:

    int a;

public:

    A1()

    {

        cout << " A1 constructor: \n";

    }

    void show()

    {

        cout << " within A1 show "

             << "a=" << a << endl;

    }

    ~A1()

    {

        cout << " A1 destructor: \n";

    }

};

class A2

{

protected:

    int a;

public:

    A2()

    {

        cout << " A2 constructor: \n";

    }

    void show()

    {

        cout << " within A2 show "

             << "a=" << a << endl;

    }

    ~A2()

    {

        cout << " A2 destructor: \n";

    }

};

class C : public A1, public A2

{

public:

    C()

    {

        cout << " C constructor: \n";

    }

    void get()

    {

        A1::a = 10;

        A2::a = 20;

    }

    ~C()

    {

        cout << " C destructor: \n";

    }

};

int main()

{

    C ob;

    ob.get();

    ob.A1::show();

    ob.A2::show();

}

/\*output:

A1 constructor:

 A2 constructor:

 C constructor:

 within A1 show a=10

 within A2 show a=20

 C destructor:

 A2 destructor:

 A1 destructor:\*/

->Hybrid Inheritance:

#include <iostream>

using namespace std;

class A

{

public:

    A()

    {

        cout << " A constructor: \n";

    }

    ~A()

    {

        cout << "A destructor: \n";

    }

};

class B : public A

{

public:

    B()

    {

        cout << " B constructor: \n";

    }

    ~B()

    {

        cout << "B destructor: \n";

    }

};

class C : public A

{

public:

    C()

    {

        cout << " C constructor: \n";

    }

    ~C()

    {

        cout << "C destructor: \n";

    }

};

class D : public B, public C

{

public:

    D()

    {

        cout << " D constructor: \n";

    }

    ~D()

    {

        cout << "D destructor: \n";

    }

};

int main()

{

    D ob; // The Grandparent class A is constructed as well as destructed twice

}

/\*output:

A constructor:

 B constructor:

 A constructor:

 C constructor:

 D constructor:

D destructor:

C destructor:

A destructor:

B destructor:

A destructor:\*/

->overcoming:

#include <iostream>

using namespace std;

class A

{

public:

    A()

    {

        cout << " A constructor: \n";

    }

    ~A()

    {

        cout << "A destructor: \n";

    }

};

class B : virtual public A

{

public:

    B()

    {

        cout << " B constructor: \n";

    }

    ~B()

    {

        cout << "B destructor: \n";

    }

};

class C : virtual public A

{

public:

    C()

    {

        cout << " C constructor: \n";

    }

    ~C()

    {

        cout << "C destructor: \n";

    }

};

class D : public B, public C

{

public:

    D()

    {

        cout << " D constructor: \n";

    }

    ~D()

    {

        cout << "D destructor: \n";

    }

};

int main()

{

    D ob;

}

/\*output:

A constructor:

 B constructor:

 C constructor:

 D constructor:

D destructor:

C destructor:

B destructor:

A destructor:\*/

->RTP :

#include <iostream>

using namespace std;

class A

{

public:

    void show()

    {

        cout << "A show:\n";

    }

};

class B : public A

{

public:

    void show()

    {

        cout << "B show: \n";

    }

};

int main()

{

    B ob;

    A \*ptr;

    ptr = &ob;

    ptr->show(); // Runtime polymorphism is not implemented, the type of the class pointer (Base class A)decides which version of the show function is to be invoked.

}

->

#include <iostream>

using namespace std;

class A

{

public:

    virtual void show() // RTP is implemented

    {

        cout << "A show:\n";

    }

};

class B : public A

{

public:

    void show()

    {

        cout << "B show: \n";

    }

};

int main()

{

    B ob;

    A \*ptr;

    ptr = &ob;

    ptr->show(); // derived class show function is invoked.

}

->

#include <iostream>

using namespace std;

class A

{

public:

    A()

    {

        cout << "A def const: \n";

    }

    A(int x)

    {

        cout << "A param const: \n";

    }

};

class B : virtual public A

{

public:

    B(int x) : A(x) // always A's def const will be called, even if A's param const is invoked explicitly

    {

        cout << "B param const: \n";

    }

};

class C : virtual public A

{

public:

    C(int x) : A(x) // always A's def const will be called, even if A's param const is invoked explicitly

    {

        cout << "C param const: \n";

    }

};

class D : public B, public C

{

public:

    D(int x) : B(x), C(x), A(x) // if we virtually inherit the class A, we have to call A's param const in this way

    {

        cout << "D param const: \n";

    }

};

int main()

{

    D ob(1);

}

->

#include <iostream>

using namespace std;

class A

{

public:

    A()

    {

        cout << "A def const: \n";

    }

    A(int x)

    {

        cout << "A param const: \n";

    }

};

class B : virtual public A

{

public:

    B(int x)

    {

        cout << "B param const: \n";

    }

};

class C : virtual public A

{

public:

    C(int x)

    {

        cout << "C param const: \n";

    }

};

class D : public B, public C    //contructort will be called according to the order of inheritance

{

public:

    D(int x) : C(x), B(x) // if the order of the constructor call is opposite of inheriting the classes,then also the

                        // order of inheritence will be followed for creating the grandchild

    {

        cout << "D param const: \n";

    }

};

int main()

{

    D ob(1);

}

->

#include <iostream>

using namespace std;

class A

{

public:

    virtual void display() // virtual function is used for RTP

    {

        cout << "Within A: \n";

    }

};

class B : public A

{

public:

    void display()

    {

        cout << "Within B: \n";

    }

};

int main()

{

    A \*ptr; //always create the pointer of the base class

    B ob;

    ptr = &ob;

    ptr->display();       //RTP is achieved as we have made the A class display function virtual

}

->pure virtual functions:

#include <iostream>

using namespace std;

class A

{

public:

    virtual void display() = 0; // pure virtual function, 0 is assigned to nowhere it is just an indication to the

                                //compiler that it is pure virtual func and has no body

};

class B : public A

{

public:

    void display()  //must be defined in child class as this function is made pure virtual function in the base class

    {

        cout << "Within B: \n";

    }

};

class C : public A

{

public:

    void display()  //must be defined in child calss as this function is made pure virtual function in the base class

    {

        cout << "Within C: \n";

    }

    void print()

    {

        cout << "Within C print: \n";

    }

};

int main()

{

    A \*pt1, \*pt2;

    B ob1;

    C ob2;

    pt1 = &ob1;

    pt2 = &ob2;

    pt1->display();

    pt2->display();

    ob2.print();

}

->

#include <iostream>

using namespace std;

class Base

{

public:

    Base()

    {

        cout << "Base const: \n";

    }

    virtual ~Base() = 0; // pure virtual destructor

};

Base::~Base()   //define the base pure virtual destructor

{

    cout << "Base pure virtual dest: \n";

}

class Derv : public Base

{

public:

    Derv()

    {

        cout << "Derv const: \n";

    }

    ~Derv()

    {

        cout << "Derv dest: \n";

    }

};

int main()

{

    Base \*b\_ptr = new Derv; //for this line the destructors are never invoked

    delete b\_ptr;   //for this line destructor are invoked

    cout<<endl;

    // Base ob; //a calls havind pure virtual destructor are cannot instantiated

    Derv ob;    //for the line all the cons and dest are invoked

}

->

#include <iostream>

using namespace std;

class X

{

    int x;

public:

    X(int = 100);

    void display();

};

X::X(int x) //we can define the const and functions outside of the class in this way, but a friend function should not be defined in this way, it will defined like normal functions

{

    this->x = x;

}

void X::display()

{

    cout << "x=" << x << endl;

}

int main()

{

    X ob;

    ob.display();

    X ob1(5);

    ob1.display();

}

->

#include <iostream>

using namespace std;

class Y;    //forward declaration

class X

{

    int x;

public:

    X(int = 100);

    void display\_x();

    friend void show(X, Y);     //friend function

};

X::X(int x)

{

    this->x = x;

}

void X::display\_x()

{

    cout << "x=" << x << endl;

}

class Y

{

    int y;

public:

    Y(int = 100);

    void display\_y();

    friend void show(X, Y); //friend function

};

Y::Y(int y)

{

    this->y = y;

}

void Y::display\_y()

{

    cout << "y=" << y << endl;

}

void show(X obx, Y oby)

{

    cout << "Hello friend: " << obx.x << "  " << oby.y << endl;

}

int main()

{

    X ob1(5);

    ob1.display\_x();

    Y ob2(15);

    ob2.display\_y();

    show(ob1, ob2);

}

->