***Object Oriented Programming:***

->Operator overloading:

The operator which support overloading:

Binary operators:

1. Arithmetic operators [+, -, \*, /, %]

2. Relational operators [>,>=,<,<=,!=,==]

3. Logical operators [&&, ||, !]

4. Bitwise operators [&, |,~, ^]

5. Functional call operator ()

6. Array subscript operator [], [][]

7. Arrow operator ->

8. input [cin>>] output [cout<<] operator

9. Assignment operator =

10.Conversion operator

Unary operators:

11. Post/pre increment operator ++

12. Post/pre decrement operator --

13. new and delete operators

The operator which DO NOT support overloading:

1. Ternary/conditional operator ? :

2. dot operator .

3. this operator

4. scope resolution :: operator

(+) operator :

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x)

    {

        a = x;

    }

    void operator+(A t)

    {

        a += t.a;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(50), ob2(60);

    ob1 + ob2; // ob1.operator+(ob2);

    ob1.display();

    ob2.display();

}

-> (-) operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    A operator-(A &t) //take less space

    {

        a = this->a - t.a;

        return \*this; //returning the reference of the class object

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(50), ob2(60), ob3(1);

    ob1 - ob2; // ob1.operator-(ob2);

    ob1.display();

    ob3 = ob1 - ob2;

    ob3.display();

}

// output:

// a=-10

// a=-70

->multiple operator overloading:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    A operator\*(A &obj)

    {

        A temp;

        temp.a = a \* obj.a;

        return temp;

    }

    A operator+(A &obj)

    {

        A temp;

        temp.a = a + obj.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    ob3 = ob1 \* ob2 + ob3; // ob3=(ob1.operator\*(ob2)).operator+(ob3);

    ob1.display();

    ob2.display();

    ob3.display();

}

->overloading functions when it is made friend

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    friend A operator\*(A &obj1, A &obj2)    //friend function will always take atleas one parameter

    {

        A temp;

        temp.a = obj1.a \* obj2.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    ob3 = ob1 \* ob2; // ob3=operator\*(ob1,ob2);

    ob1.display();

    ob2.display();

    ob3.display();

}

->overloaded friend functions:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    friend A operator\*(A &obj1, A &obj2)

    {

        A temp;

        temp.a = obj1.a \* obj2.a;

        return temp;

    }

    friend A operator+(A obj1, A &obj2) //first parameter should pass by value

    {

        A temp;

        temp.a = obj1.a + obj2.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    ob3 = ob1 \* ob2 + ob3; // ob3=operator+(operator\*(ob1,ob2),ob3);

    ob1.display();

    ob2.display();

    ob3.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    friend A operator\*(A &obj1, A obj2) //here second parameter should pass by value

    {

        A temp;

        temp.a = obj1.a \* obj2.a;

        return temp;

    }

    friend A operator+(A &obj1, A &obj2)

    {

        A temp;

        temp.a = obj1.a + obj2.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    ob3 = ob1 \* (ob2 + ob3); // ob3=operator\*(ob1,(operator+(ob2,ob3));

    ob1.display();

    ob2.display();

    ob3.display();

}

->(++) operator(pre-increment):

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    void operator++()

    {

        a++; //here we do not create any temp object, in this case do the opposite increment only for pre-increment operation

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5);

    ++ob1; // ob1.operator++();

    ob1.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    A operator++()

    {

        A temp; //when we create a temp obj then do as same as the increment operator, means for pre-increment do pre-increment, and for post increment do post-increment

        temp.a = ++a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ++ob1; // ob1.operator++();

    ob1.display();

    ob2.display();

}

->friend function:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    friend A operator++(A &ob)

    {

        A temp;

        temp.a = ++ob.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ++ob1; // operator++(ob1);

    ob1.display();

    ob2.display();

}

->(++) operator(post-increment):

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    A operator++(int)   //pass a int as parameter in case of post-increment

    {

        A temp;

        temp.a = a++;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1++; // ob1.operator++(int);

    ob1.display();

    ob2.display();

}

->friend function:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    friend A operator++(A &ob, int)

    {

        A temp;

        temp.a = ob.a++;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1++; // operator++(ob1,int);

    ob1.display();

    ob2.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    A operator++()

    {

        cout<<"++ operator"<<endl;

        A temp;

        temp.a = ++a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

    A operator\*(A ob)   //last executed overloaded function will pass by value

    {

        cout<<"\* operator"<<endl;

        A temp;

        temp.a = a \* ob.a;

        return temp;

    }

    A operator+(A &ob)

    {

        cout<<"+ operator"<<endl;

        A temp;

        temp.a = a + ob.a;

        return temp;

    }

};

int main()

{

    A ob1(5), ob2(6), ob3;

    ob3 = ob1 \* (++ob1 + ob2); //(ob1.operator++(int)).operator+(ob2);

    ob1.display();

    ob2.display();

    ob3.display();

}

-> (<) operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    bool operator<(A &obj)

    {

        if (a < obj.a)

            return true;

        else

            return false;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    if (ob1 < ob2) // ob3=ob1.operator<(ob2);

        cout << "ob2 is greater ";

    else if (ob3 < ob1)

        cout << "ob1 is greater ";

    else if (ob2 < ob3)

        cout << "ob3 is greater: ";

    ob1.display();

    ob2.display();

    ob3.display();

}

-> (&&) operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    bool operator&&(A &obj)

    {

        if (a && obj.a)

            return true;

        else

            return false;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(6), ob2(5), ob3(10);

    if (ob1 && ob3) // ob3=ob1.operator&&(ob2);

        cout << "None of them is 0 \n";

    if (ob1 && ob2)

        cout << "None of them is 0 \n";

    ob1.display();

    ob2.display();

    ob3.display();

}

-> (&) operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    bool operator&(A &obj)

    {

        if (a & obj.a)

            return true;

        else

            return false;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2(15), ob3(10);

    if (ob1 & ob3) // ob3=ob1.operator&(ob2);

        cout << "None of them is 0 \n";

    if (ob3 & ob2)

        cout << "None of them is 0 \n";

    ob1.display();

    ob2.display();

    ob3.display();

}

-> (^) operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 1)

    {

        this->a = a;

    }

    A operator^(A &obj)

    {

        A temp;

        temp.a = a ^ obj.a;

        return temp;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2(14), ob3(10), ob4;

    ob4 = ob1 ^ ob3; // ob4=ob1.operator^(ob3);

    ob4.display();

    ob4 = ob3 ^ ob2;

    ob4.display();

}

->function call operator overloading:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    A operator()(int x)

    {

        a = x;

        return \*this;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob1.display();

    ob2.display();

    ob2 = ob1(6); // ob2=ob1.operator()(6);

    ob1.display();

    ob2.display();

}

-> ([]) operator:

#include <iostream>

using namespace std;

class Array

{

    int \*a;

public:

    Array(int n)

    {

        a = new int[n];

        for (int i = 0; i < n; i++)

            a[i] = i + 1;

    }

    int operator[](int index) // no need to return the reference as the function call statement appears on the RHS of the = operator

    {

        return a[index];

    }

};

int main()

{

    Array ob(5);

    int x = ob[2]; // x=ob.operator[](2);

    cout << "x=" << x;

}

->

#include <iostream>

using namespace std;

class Array

{

    int \*a;

    int size;

public:

    Array(int n)

    {

        a = new int[n];

        size = n;

        for (int i = 0; i < n; i++)

            a[i] = i + 1;

    }

    int &operator[](const int &index) // returning a reference is MUST

    {

        return a[index];

    }

    void display()

    {

        for (int i = 0; i < size; i++)

            cout << a[i] << "  ";

    }

};

int main()

{

    Array ob(5);

    ob[1] = 20; // ob.operator[](2)=20;  //as the func call statement appears on the LHS of = operator,the function must return the reference

    ob.display();

}

-> (->) arrow operator:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

    }

    A \*operator->()

    {

        return this;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob(5);

    ob->display(); // (ob.operator->())->display();

}

->

#include <iostream>

using namespace std;

class Y

{

    int y;

public:

    Y()

    {

        y = 10;

    }

    void f()

    {

        cout << "y=" << y << endl;

    }

};

class X

{

    Y \*ptr;

public:

    X(Y y1)

    {

        ptr = &y1;

    }

    Y \*operator->()

    {

        return ptr;

    }

};

int main()

{

    Y y1;

    X x(y1);

    x->f(); //(x.operator->())->f();

}

-> cout operator overloading:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    friend ostream &operator<<(ostream &os, const A &t)

    {

        os << t.a;

        return os;

    }

};

int main()

{

    A ob(10);

    cout << ob; // operator<<(cout,ob);

}

-> cin operator overloading:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    friend istream &operator>>(istream &is, A &t)

    {

        is >> t.a;

        return is;

    }

    friend ostream &operator<<(ostream &os, const A &t)

    {

        os << t.a;

        return os;

    }

};

int main()

{

    A ob(10);

    cout << "Enter the value of the object: ";

    cin >> ob; // operator>>(cin,ob);

    cout << ob; // operator<<(cout,ob);

}

-> conversion operator overloading:

1. BDT to UDT

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

        cout << "const called: \n";

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob = 2; // conversion from BDT to UDT

    ob.display();

    A ob1(3);

    ob1.display();

}

1. UDT to BDT

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int a = 0)

    {

        this->a = a;

        cout << "const called: \n";

    }

    operator int()

    {

        return a;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob = 2; // BDT to UDT

    ob.display();

    int x = ob; //UDT to BDT

    cout << "x=" << x << endl;

}

1. UDT to UDT

#include <iostream>

using namespace std;

class Source

{

    int s;

public:

    Source(int s)

    {

        this->s = s;

    }

    int get\_s()

    {

        return s;

    }

};

class Dest

{

    int d;

public:

    Dest(int d = 0)

    {

        this->d = d;

    }

    Dest(Source sob)

    {

        d = sob.get\_s();

    }

    void display()

    {

        cout << "d=" << d << endl;

    }

};

int main()

{

    Source s(5);

    Dest d;

    d = s; // d(s); //UDT to UDT

    d.display();

}

-> order of destination and source does matter

#include <iostream>

using namespace std;

class Source

{

    int s;

public:

    Source(int s)

    {

        this->s = s;

    }

    friend class Dest;

};

class Dest

{

    int d;

public:

    Dest(int d = 0)

    {

        this->d = d;

    }

    Dest(Source sob)    //this is required when the destination is after source

    {

        d = sob.s; // Dest is a friend of Source class

    }

    void display()

    {

        cout << "d=" << d << endl;

    }

};

class Source2

{

    int s;

public:

    Source2(int s)

    {

        this->s = s;

    }

    operator Dest() //this is required when the source is after destination

    {

        return Dest(s);

    }

    friend class Dest;

};

int main()

{

    Source s(5);

    Dest d;

    Source2 s2(2);

    d = s2;

    d = s; // d(s);

    // s = d; // error

    d.display();

}

->Mixed Arithmetic:

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    A operator+(A ob)

    {

        A t;

        t.a = a + ob.a;

        return t;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1 + 10; // ob2=ob1.operator+(A(10))

    ob2.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    friend A operator+(A ob, int x)

    {

        A t;

        t.a = ob.a + x;

        return t;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1 + 10; // ob2=operator+(ob1,10);

    ob2.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    friend A operator+(A &ob, A x)

    {

        cout << "int converted to UDT: \n";

        A t;

        t.a = ob.a + x.a;

        return t;

    }

    friend A operator+(A &ob, int x)    //this version will be invoked

    {

        cout<<"int remains same: \n";

        A t;

        t.a=ob.a+x;

        return t;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1 + 10; // ob2=operator+(ob1,A(10)); ////ob2=operator+(ob1,10);

    ob2.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    A operator+(A x)

    {

        cout << "int converted to UDT: \n";

        A t;

        t.a = a + x.a;

        return t;

    }

    A operator+(int x)

    {

        cout << "int remains same: \n";     //this version will be invoked

        A t;

        t.a = a + x;

        return t;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1 + 10; // ob2=ob1.operator+(A(10)); //ob2=ob1.operator+(10);

    ob2.display();

}

->

#include <iostream>

using namespace std;

class A

{

    int a;

public:

    A(int x = 0)

    {

        a = x;

    }

    friend A operator+(A x, A ob) //this version will be invoked

    {

        cout << "int converted to UDT: \n";

        A t;

        t.a = ob.a + x.a;

        return t;

    }

    friend A operator+(int x, A &ob)

    {

        cout << "int remains same: \n";

        A t;

        t.a = ob.a + x;

        return t;

    }

    void display()

    {

        cout << "a=" << a << endl;

    }

};

int main()

{

    A ob1(5), ob2;

    ob2 = ob1 + 10; // NOT ob2=ob(10).operator+(ob1)); //NOT ob2=ob(10).operator+(10);

                    //  ob2=operator+(ob(10),ob1);

    ob2.display();

}