Operator Overloading

→ Performing operations b/w of two objects of a class.

operators that can NOT be overloaded

:: → scope resolution operator

. → dot operator

?: → Ternary operator

* → pointer de referrencing operator.

sizeof → sizeof operator.

operators that can be overloaded

Unary operator

++ → pre / post increment

-- → pre / post decrement

! → logical NOT operator

Binary operator

→ Arithmetic operator

+, -, /, *, %, +=

→ Relational operator

<, <=, >, >=, !=, ==

→ Logical operator

&&, ||

→ Bitwise operator

&, |, ^, ~, <<, >>

→ ( ) → function call operator

→ '→' → Arrow operator

→ [ ] → Array subscript operator

→ cin >>, cout <<, → Input / output operator.

→ new, delete

→ Conversion operator.

converting Userdefined datatype to a Basic datatype

converting basic data type to a Userdefined data type.

Converting userdefined data type to another Userdefined datatype.

→ Mixed mode arithmetic.

Arithmetic operators

ob3 = ob1.operator+=(ob2);

```
class A {
    int a;
    public:
        A(int x = 0) {
            a = x;
        }
        A operator + (A t) {
            A ob;
            ob.a = a + t.a;
            return ob;
        }
        void show() {
            cout << ob.a;
        }
};
int main() {
    A ob(1), ob2(4), ob3;
    ob3 = ob1 + ob2;        // compiler interprets as
    ob3.show()              //  ob1.operator+(ob2)
    return 0;               //∴ ob3 = ob1.operator+(ob2);
}
```

```
class A {
    int a;
public:
    A (int x = 0) {
        a = x;
    }
    void operator += (A &t) {
        a += t.a;
    }
    void show();
};
int main() {
    A ob1(3), ob2(5);
    ob2 += ob1;          // ob2.operator+=(ob1);
    show(ob2);           // ob2 = ob2.operator+=(ob1);
}
```

Relational operators

```
Ob1.operator <= (ob2);
```

```
class A {
    int a;
public:
    A(int x = 0) {
        a = x;
    }
    bool operator <= (A ob) {
        if (a <= ob.a) {
            return true;
        }
        else {
            return false;
        }
    }
    void show;
};
int main() {
    A ob1(2), ob(3);
    if (ob1 <= ob) {          // compiler interprets as
        cout << "True";       // ob1.operator<=(ob);
    }
    else {
        cout << "False";
    }
    return 0;
}
```

Logical operator

```
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 show() {
            cout << ob.a;
        }
};
int main() {
    A ob1(3), ob2(4), ob3;
    ob3 = ob1 && ob2;
    ob3.show();
}
```

```
// ob3 = ob1.operator&&(ob2);
//  ∵ ob1.a ≠ 0
//      ∴ ob1.a = 1
//  ∵ ob2.a ! = 0
//      ∴ ob2.a = 1.
//  ∴ ob3.a = 1 && 1 = 1.
//  ∴ ob3.a = 1.
```

Bitwise operator

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A operator | (A ob) {
            A t;
            t.a = a | ob.a;
            return t;
        }
};
int main() {
    A ob1(3), ob2(4), ob3;
    // ob3 = ob1 & ob2;      ob1. operator & (ob2) //
    ob3 = ob1 | ob2;      ob3.show();
    return 0;
}
```

∵ ob3 = ob1 | ob2 ;

ob1.a = 3 ≡ 011

ob2.a = 4 ≡ 100

```
    0 1 1
(1) 1 0 0
-------
    1 1 1
```

ob3.a ≡ 111 = 7.

∴ ob3.a = 7.

function call operator

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A operator () (int x) {
            A t;
            t.a = a + x;
            return t;
        }
};
int main () {
    A ob1, ob2;
    ob2 = ob1 (5);          // ob2 = ob1.operator()(5);
    return 0;
}
```

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A operator () (A x) {
            A t;
            t.a = a + x.a
            return t;
        }
};
int main () {
    A ob1 (3), ob2 (4), ob3;
    ob3 = ob1 (ob2);
    return 0;
}
```

Pointer Arrow operator

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A* operator -> () {
            return this;
        }
};
```

```
int main () {
    A ob1(3), ob2 (4);
    ob1 → show();      // (ob1.operator → ()) → show
    ob2 → show();      // (ob2.operator → ()) → show
    A * ptr = & ob1;
    ptr → show();
    return 0;
}
```

Array subscript operator

```
Class A {
    int*a; int size
    public:
        A (int x) {
            size = x;
            a = new int [size];
            for (i=0; i< x ; i++) {
                a[i] = i+1;
            }
        }
        int operator [] (int x) {
            if (x > size) {
                return -1;
            }
            return a[i];
        }
};
```

```
class A {
    int * buffer;
    int size;
    public :
        A (int x) {
            size = x;
            buffer = new int [size];
            for (int i = 0; i< size ; i++) {
                buffer [i] = i+1;
            }
        }
        int operator [] (int x) {
            if (x > size) {
                return -1;
            }
            return buffer [x];
        }
};
int main () {
    A ob(10);
    int x = ob(2);      // x = ob.operator [](
    cout << x;          // 3.
}
```

cin , cout operator

```
class A {
    int a;
  public:
    A (int x = 0){
        a = x;
    }

    friend void operator >> (istream & is, A &ob){
            is >> ob.a;
    }
            // memory allocation of cin given
            // new name is .
            // cin , cout are part of class istream
            // and ostream .·. to use cin and cout
            // in another class, friend is used.

    friend void operator << (ostream & os, const A &ob){
            os << ob.a;
    }
                                    // 1
// friend void operator <<(

    friend ostream operator << (ostream & os, const A ob)
    {   os << ob.a;
        return os;
    }
                                    // 2
```

```
int main (){
    A ob1 (2);
    A ob2;
    cin >> ob2;
    cout << ob1;                    // 1 is called
    cout <<ob1 << ob2;              // 2 is called.
}
```

Mixed mode arithmetic

```
class A {
    int a;
  public:
    A(int x = 0) {
        a = x;
    }
    A operator + (int x) {
        A t;
        t.a = a + x;
        return t;
    }
};
```

```
int main (){
    A ob1(3), ob2;
    ob2 = ob1+5;        // ob2 = ob1.operator+(5);
    ob1.show();
}
```

```
class A {
    int a;
    public:
        A (int x=0){
            a=x;
        }
        friend A operator + (int x, A ob ){
            A t;
            t.a = x + ob.a;
            return t;
        }
};
int main(){
    A ob1(3), ob2;
    ob2 = 5 + ob1;
}
```

Unary operator

```
class A {
    int a;
    public :
        A ( int x = 0){
            a = x;
        }
        A operator ++ () {
            a = a++;
            return * this;
        }
};
int main () {
    A ob1 (3), ob2;
    ob2 = ++ ob1;  // ob2 = ob1. operator ++ ().
    //ob2 = ob1. operator ++ ();
    ob2 . show();   // 4
    ob1. show();    // 4
```

To differentiate b/w pre increment and post increment operators.

```
A operator ++ (int) {
        A t
        t.a = a++;
        return t;
}
```

∴ class is

```
class A {
    int a;
    public:
        A (int x=0) {
            a = x;
        }
        A operator ++ (int) {
            A t;
            t.a = a++;
            return t;
        }
};
int main () {
        A ob1(3), ob2;
        ob2 = ob1 ++;
        ob2.show();
        ob1.show();
}
```

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A operator ! () {
            A t;
            t.a = !a;
            return t;
        }
};
int main () {
        A ob1 (3), ob2;
        ob2 = ! ob1;
        ob2.show ();
        ob1.show();
}
```

If friend function is used,

```
friend A operator ! (A & ob) {
        A t;
        t.a = ! (ob.a);
        return t;
}
```

Conversion operator overloading

→ Basic Data type to User defined data type.

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        void show() {
            cout << a;
        }
};

int main () {
    A ob1 = 5;        // A ob1(5);
    ob1.show();       // gets converted to A ob1(5).
}
```

→ user defined data type to basic data type.

```
class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        operator int () {          // operator is a keyword.
            return a;              // return type is int.
        }                          // return type is mentioned after keyword operator.
};

int main () {
    A ob(50);
    int x = ob;        // converts into x = int(ob).
    cout << x;         // x = int(ob);
}                      // type casting occours.
```

// return type is mentioned after keyword operator.

→ User defined data type to User defined data type.

By over loading assignment operator

```
class A {
    int a;
    public :
        A (int x = 0) {
            a = x;
        }
        int get_a () {
            return a;
        }
};

class B {
    int b;
    public:
        B (A & ob) {
            b = ob.get_a ();
        }
        Void show () {
            cout << b ;
        }
};

int main () {
    A ob a (5) ;
    B obb ;                    // obb = B(oba);
    ob b = ob a ;              // obb = B(oba);
    obb.show();                // typecasting is done.
}
```

```
class B;

class A {
    int a ;
    public :
        A (B ob);
};

class B {
    int b;
    public:
        B (int y = 0) {
            b = y;
        }
        int get_b () {
            return b;
        }
        void show () {
            cout << b;
        }
};

A :: A (B ob) {
        a = ob.get b ();
}

int main() {
    A ob a ;
    B obb (5);
    oba = obb;          // oba = A (obb);
    oba. show ();
}
```

→ By overloading assignment operator.

```
class B;
class A {
    int a;
    public:
        void operator = (B ob);
};
class B {
    int b;
    public:
        B(int y = 0) {
            b = y;
        }
        int get_b () {
            return b;
        }
        void show() {
            cout << b;
        }
};

void A :: operator = (B ob) {
        a = ob.get_b();
}
```

```
int main () {
    A oba;   B obb;
    oba = obb;          // oba.operator = (obb);
    oba.show();         // oba. operator = (obb);
}

class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
};

class B {
    int b;
    public:
        B(int y = 0) {
            b = y;
        }
        operator A () {          // same as
            return A(b);         // operator int () {
        }                        // same as
};                               // operator int() {
```

```
int main (){

    A oba;
    B obb;
    oba = obb;        // oba = A(obb);

    oba.show();

}

class B;

class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }

        operator B ();

};

class B {
    int b;
    public:
        B (int y = 0) {
            b = y;
        }

};
```

```
A :: operator B () {

    return B(a);

}

int main (){

    A oba (5);

    B obb;

    obb = oba;        // obb = B B(oba);

    Obb.show();       // obb = B(oba);

}
```

Mixed Mode Arithmetic

```
class B;
class A {
    int a;
    public:
        A (int x = 0){
            a = x;
        }
        friend A operator + (int x, A &ob) {
            A t;
            t.a = x + ob.a;
            return t;
        }
};
```

```
friend A operator + (A x, A ob){
        A t;
        t.a = x.a + ob.a;
        return t;
}

int main (){
        A ob1(6);
        A ob2;
        ob2 = 10 + ob1;      // first friend function is
                             // called.
        ob2.show();
}
```

If the first friend function was commented out, second friend function will run.

∵ Ax is an object of A class of the value of x as 10.

∴ 10 in converted into an object of type A as A(10). That is Ax.

```
class B;
class A {
        int a;
        public:
                A (int x = 0){
                        a = x;
                }
                A operator + (int x){          ——— (1)
                        A t;
                        t.a = x + a;
                        return t;
                }
                A operator + (A x){            ——— (2)
                        A t;
                        t.a = x.a + a;
                        return t;
                }
};
```

```
friend A operator + (A ob, int x){     ——— (3)
        A t;
        t.a = x + ob.a;
        return t;
}
```

```
int main() {
    A oba(10);
    A ob;
    // ob = oba + 10;
    ob = oba + 10;      // ob = oba.operator+(10)
                        OR
    ob.show();          ob = oba.operator+(oba, 10);
}
```

∴ ① and ③ creates Ambigious situation.

for ① and ③, ambigious occours.

∴ ① is converted into ② function.

If ① is commented out,

② and ③ do NOT create ambiguity

```
class A {
    int a;
public:
    A(int x = 0) {
        a = x;
    }
    A operator + (A x) {
        return A(x + a.x);
    }
    Operator int () {
        return a;
    }
};
int main () {
    A ob1(2); int ob;
    A ob2(3);
    ob = ob1 + ob2;
    ob.show();
}
```

// int ob = A ob1 + A ob2

∴ ob = int (A ob1 + A ob2).

∴ ob = int (ob1 + ob2);

```
Class A {
    int a;
    public:
        A (int x = 0) {
            a = x;
        }
        A operator ++ () {
            a++;
            return * this;
        }
        operator int () {
            return a;
        }
        void operator += (A ob) {
            A += ob.a;
        }
};
int main () {
    A ob1(6);
    A ob2(7);
    ob2 += (++ob1);
    cout << ob2;
}
```

```
Class A {
    int a;
    public:
        A (int a) {
            this -> a = a;
        }
        operator int () {
            return a;
        }
};
int main () {
    A ob1(5), ob2 (6);
    int x = ob1 + ob2;
    cout << x.
```

Methode 2

```
Class A {
    int a;
    public:
        A (int a = 0) {
            this -> a = a;
        }
        int operator + (A ob2) {
            return (a + ob2.a);
        }
};
```

```
int main (){
    A ob1(5), ob2 (6)
    int x = ob1 + ob2 .   // ob1. operator +(ob2);
    cout << x;
}

class A{
    int a;
    public:
    A(int a=0){
        this->a = a;
    }
    A operator + (A &ob){
        return A(a+ob.a);
    }
};
```

b = a++ ⟹ b = a ;
a = a+1;

⟹ post or preincrement operators can ONLY be applied on variables or memory location, NOT on constants.

a++ is possible. ∵ a = a+1.

But 5++ is NOT possible, ∵ 5 != 5+1.

```
class A{
    int a;
    public:
    A(int a=0){
        this->a = a;
    }
    operator int (){
        return a;
    }
    A operator ++ (int){
        A t;
        t.a = a++;      // t.a = a ;
        return t;       // a = a+1 ;
    }
};
int main (){
    A ob1(5), ob2 (6)
    A x = ob1 ++ - ob2;
    cout << x;
}
```

```
class A {
    int a;
  public:
    A(int a = 0) {
        this -> a = a;
    }
    A operator ++ () {
        return A(++a);
    }
};
int main() {
    A ob1(5), ob2(6)
    int x = 10 - ++ob2;
    cout << x.
// class A {
```

```
class A {
    int a;
  public:
    A (int a=0) {
        this -> a = a;
    }
    int operator () (int x) {
        return a;
    }
    operator int () {
        return a;
    }
};
int main() {
    A ob1(5), ob2 (6);
    A ob3;
    ob3 = ob2 (ob1 + 5);
    cout << ob3;
}
```

```
class A {
    int a;
    public:
        A (int x=0) {
            this -> a = a;
        }
        A operator * = (A & ob) {
            a * = ob.a
            // return * this ;
            // return A (a) ;
            return a ;
        }
        operator int () {
            return a;
        }
};
int main () {
    A ob1 (5), ob2 (6);
    A ob 3 = ob1 *
                = ob2;
    Cout << ob1 ;
    cout << ob2 << ob3;
}
```

```
class Hours {
    int h;
    public:
        Hours (int h=0) {
            this -> h = h;
        }
        int get_h ( ) {
            return h;
        }
};
Class Munite {
    int m ;
    public:
        Munites (int m = 0) {
            this -> m = m;
        }
        Munites ( Hours & ob) {
            m = ob.get_h ( );
        }
};
int main ( ) {
    Hours h (3);
    Munites m;
    m = h ; // m = M (h)
    m.show ( );
}
```

Methode 2

```
class Munites {
        int m ;
        public:
            Munites ( Hours &ob) {
                m = ob.get_h ();
            }
            Munites (int m = 0) {
                this → m = m;
            }
};
int main () {
        Hours h (3);
        Munites m;
        m = h;    // m.operator = (h)
        m.show();
}
```

Methode 3

```
class Hours {
        int h;
        public:
            Hours (int h = 0) {
                this → h = h;
            }
            Operator Munites ();
};
class Munites {
        int m;
        public:
            Munites ( int m = 0) {
                this → m = m;
            }
};
Munites :: Hours ( Hours ob) {
        m = ob.get_h() ;
        return m;
}
int main (){
        Hours h(30); Munites m;
        m = h ;
        m.show();
}
```

New Delete operator overloading

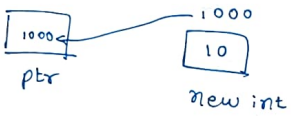
```
int main (){

    int * ptr = new int;

    * ptr = 10;

     cout << * ptr;

}
```



* ptr = 10 .

```
void * operator new (size_t sz) {

        Void * p ;

        p = malloc (sz);

        if (p ! = NULL) {

            return p;

        }

        else {

            return p;

        }

}
```

```
Void * operator new (size_t sz) {

        Void * p;

        p = ::new int [sz];

        if (p! = NULL){

            return p;

        }

        else {

            return p;

        }

};
```

```
Void * operator new (size_t sz){

        void *p;

        p = ::new int [sz];

        for (int i = 0; i <sz ; i++){

            (char* p) [i] = 0;

        }

        return p;

}
```

For delete overloading ,

```
delete ptr;
```

return type of new is void *.

return type of delete is void.

If array is allocated

```
void * operator new [ ] (size_t sz){
    //code;
}

void operator delete [ ] (void *p) {
    :: delete (p);
}
```

OR

```
void operator delete (void * p){
    free(p);
}
```

a array subscript operator must be added after new or delete. for an array.

```
class Number {
    int n;
    public:
        Number (int n = 0) {
            this -> n = n;
        }

        void * operator new (size_t sz){
            void * ptr = :: new int [sz];
            if (ptr != NULL){
                * ptr = 5;
            }
            return ptr;
        }
};

int main (){
    Number * ptr = new Number (5);
    ptr -> show ()
    delete ptr;
}
```

```
void *operator new (size_t sz, char ch) {
    void *p;
    p = ::new char [sz];
    *p = ch;
    return p;
}
```

```
void operator delete (void *ptr) {
    if (ptr != NULL) {
        ::delete (ptr);
    }
}
```