Day 8 (09-12-2025)

Operator categories

-
- Operator
- Operand
- Operators -> Ternary, Binary, Unary, Bitwise, Relational, Arithmetic, increment/decrement, Assignment;
- Unary, Binary, Ternary and Special

Unary

- increment ++
- decrement --
- ! negation
- +, - -> 10 -> -10 -> +10

Binary

- +, -, *, /, % -> Arithmetic operators
- < , >, <=, >=, ==, != -> Relational operators
- &&, ||

Ternary

- ? :

Special

- ., [], (), <<, >>, ::, ->, .*

```
    // Syntax:
    Return_Type operator_type(function_parameters){
        //code
        return_statement;
}
```

```
#include <iostream>
using namespace std;

class Demo{
    int val;

public:
    Demo(int val) : val(val){}
    static int add(Demo d1, Demo d2){
        return d1.val + d2.val;
    }

    void print(){
        cout << val << endl;
    }
};

int main() {

//  class_type object;

    Demo d1(10);        // object of Demo
    Demo d2(20);        // object of Demo

//  Demo d3 = d1 + d2;  //addition of d1 and d2 -> not valid

    Demo d3 = Demo::add(d1, d2);

    int a = 10;
    int b = 20;

    int c = a + b;  //like normal data types
//      a + b = a.operator+(b);
    cout << c << endl;

    d3.print();

//  cout << d3 << endl;
    return 0;
}
```

Member vs Non-member

- lvalue, rvalue -> left side and right side value for operator
- lvalue -> left side value -> can be assigned
- rvalue -> right side value -> cannot be assigned
- Member operator overloading function -> lvalue is of your class type
- Non-member operator overloading function -> lvalue is not of your class type

Member

- Operator overloading function can be member function
- If the lvalue is of your class type
- It is defined inside the class
- Member operator -> + - * / -> not recommended
- it actually changes the lvalue
- It can take only one parameter & can take no parameter
- It can take multiple parameters, but the first parameter is always the lvalue

- It can return a new object of the class type
- ++, --, [], = -> Member function

```
Demo operator+(const Demo &d){
    cout << "single para" << endl;
    return Demo(this->val+ d.val);
}
```

Non-Member

- Operator overloading function cannot be member function
- If the lvalue is not of your class type
- It is declared in class and defined outside of the class -> friend function
- With the help of friend operator overloading function we can overload non-member operators
- It can take multiple parameters, but the first parameter is always the lvalue
- It can return a new object of the class type

```
friend Demo operator+(const Demo &a, const Demo &b); //declaration inside class

//definition outside class
Demo operator+(const Demo &a, const Demo &b){
    cout << "dual para" << endl;
    return Demo(a.val + b.val);
}
```

Limitations

- In operator overloading we can change the behaviour of the operators, but the rules of the programming languege cannot be changed
- Cannot change Rules of precedence, associativity.
- Cannot change the number of operands
- We cannot change the meaning of the operators
- We cannot create our own rules for operators
- We cannot create our own operators -> #, @, $

  > example: operator#() -> not valid

- There certain operators which cannot be overloaded

  > ::, .*, . sizeof, typeid -> scope resolution, member pointer, member access, sizeof, typeid

- We cannot overload the operators for built-in types

Arithmetic operator overloading

- +, -, *, /

Relational operator overloading

```
#include <iostream>
using namespace std;

class Complex{
    int real, imag;

public:
    Complex(int real, int imag) : real(real), imag(imag){}

    void print(){
        cout << real << " + " << imag << "i"<< endl;
    }

    friend bool operator<(const Complex &a, const Complex &b);
    friend bool operator>(const Complex &a, const Complex &b);
    friend bool operator<=(const Complex &a, const Complex &b);
    friend bool operator>=(const Complex &a, const Complex &b);
    friend bool operator==(const Complex &a, const Complex &b);
};

bool operator<(const Complex &a, const Complex &b){
    return a.real < b.real && b.real < b.imag;
}

bool operator>(const Complex &a, const Complex &b){
    return a.real > b.real && b.real > b.imag;
}

bool operator<=(const Complex &a, const Complex &b){
    return a.real <= b.real && b.real <= b.imag;
}

bool operator>=(const Complex &a, const Complex &b){
    return a.real >= b.real && b.real >= b.imag;
}

bool operator==(const Complex&a, const Complex &b){
    return a.real == b.real && a.imag == b.imag;
}

int main() {
    Complex c1(5,10);
    Complex c2(10,20);
    Complex c3(10,20);

    bool result = c1 < c2;
    bool resultEqual = c2 == c3;

    cout << "Result for Less than: "<< result << endl;
    cout << "Result for Equal Check: "<< resultEqual << endl;
//  10 < 20 -> true or false

//  c1 < c2, c1 > c2, c1 <= c2, c1 >= c2, c1 == c2, c1 != c2
    return 0;
}
```

Pre/Post increment

- pre-increment -> ++c
- post-increment -> c++
- pre-decrement -> --c
- post-decrement -> c--
- These operators can be overloaded as member or non-member functions
- Pre-increment and post-increment can be overloaded as member functions

```
#include <iostream>
using namespace std;

class Complex{
    int real, imag;

public:
    Complex(int real, int imag) : real(real), imag(imag){}

    void print(){
        cout << real << " + " << imag << "i"<< endl;
    }

    //pre-increment
    Complex operator++(){
        ++real;
        ++imag;
        return *this;
    }

    //post-increment
    Complex operator++(int){        // default for compiler to identify
        Complex temp = *this;    // assigned it to a variable
        real++;
        imag++;
        return temp;                //old value
    }
    //please try the -- overload for both pre & post increment.
};

int main() {
    Complex c1(10,20);
    Complex c2(10,20);

    ++c1;   // pre-increment 11 21
    c1++;   //post-increment 11 21

    c1.print();

    return 0;
}
```

<<, >> overloading

- <<, >> are stream operators
- We can overload these operators to work with our own class types

```
#include <iostream>
using namespace std;

class Complex{
    int real, imag;

public:
    Complex(): real(0), imag(0){}
    Complex(int real, int imag) : real(real), imag(imag){}

    //our own copy constructor -> deep copy
    // compiler provided -> shallow copy

    void print(){
        cout << real << " + " << imag << "i"<< endl;
    }

    // fstream -> ostream -> basic_ostream, istream -> basic_istream
    friend ostream& operator<<(ostream &out, Complex &c);
    friend istream& operator>>(istream &in, Complex &c);
};

ostream& operator<<(ostream &out, Complex &c){
    out << c.real << " + " << c.imag << "i";
    return out;
}

istream& operator>>(istream &in, Complex &c){
    in >> c.real >> c.imag;
    return in;
}

int main() {
    Complex c1(10,20);
    Complex c2(20,30);

    Complex c3;

    cout << "Enter a Complex number: " << endl;
    cin >> c3;

//  c1.print();
//  c2.print();

//  cout << c1;
//  cout.operator<<(c1);

//  cin >> c1;
//  cin.operator>>(c1);

    cout << "Complex c1 : " << c1 << endl; // valid
    cout << "Complex c2 : " << c2 << endl; // valid
    cout << "Complex c3 : " << c3 << endl; // valid
//  << >> -> stream operators

    return 0;
}
```

Assignment operator

- Assignment operator is used to assign the value of one object to another object
- We can overload the assignment operator to work with our own class types

```
#include <iostream>
using namespace std;

class Complex{
    int real, imag;

public:
    Complex(): real(0), imag(0){}
    Complex(int real, int imag) : real(real), imag(imag){}

    //our own copy constructor -> deep copy
    // compiler provided -> shallow copy

    void print(){
        cout << real << " + " << imag << "i"<< endl;
    }

    //  Assignment =
    Complex& operator=(Complex &c){
        if(this != &c){
            this->real = c.real;
            this->imag = c.imag;
        }
        return *this;
    }
};

int main(){
    Complex c1(10,20);
    Complex c2(20,30);

    Complex c3 = c2;        // copy constructor
//          c3.operator=(c2);
    cout << c2 << endl;
    cout << c3 << endl;

    return 0;
}
```

Index operator

- Index operator is used to access the elements of an array or a collection
- We can overload the index operator to work with our own class types
- It can be overloaded as a member function or a non-member function

```
#include <iostream>
using namespace std;

class Complex{
    int real, imag;

public:
    Complex(): real(0), imag(0){}
    Complex(int real, int imag) : real(real), imag(imag){}

    void print(){
        cout << real << " + " << imag << "i"<< endl;
    }

    int operator[](int index){
        if(index == 0) return real;
        if(index == 1) return imag;

        throw out_of_range("Index out of bounds!");
    }
};

int main1() {

    Complex c1(10,20);

    int val = c1[0];
    int val1 = c1[1];
    int val2 = c1[2];

    cout << val << endl;
    cout << val1 << endl;
    cout << val2 << endl;

    return 0;
}
```

```
class Map{
    string keys[10];            //Keys ->   keys[i]         0 1 2 3 4 5
    string values[10];      //Values -> values[i]   0 1 2 3 4 5
    int count;

public:
    Map() : count(0){}

    string& operator[](const string &key){
//      1. for checking if exist and to show
        for(int i=0;i<count;i++){
            if(key == keys[i])
                return values[i];
        }

//      2. for creating new map entry
        if(count < 10){
            keys[count] = key;
            values[count] = "";
            count++;
            return values[count - 1]; //values[0]
        }
        throw out_of_range("index out if bounds!");
    }

    void print(){
        for(int i=0;i<count;i++){
            cout << keys[i] << " : " << values[i] << endl;
        }
    }

};

int main(){

    Map map;

    map["city"] = "mumbai"; // assigning a value to a map with help of [] operator
//  int map[] -> map[5] = 10;

    map["name"] = "shil";
    map["course"] = "pgdac";
    string result = map["city"];

    cout << result << endl;

    map.print();

//  map[key] -> value;
//  map.operator [](key) -> value;
    return 0;
}
```

Function call operator

- Function call operator is used to call a function
- We can overload the function call operator to work with our own class types

```
#include <iostream>
using namespace std;

class TaxCalculator{
    double rate;    // tax calculate
public:
    TaxCalculator(double rate) : rate(rate){}

    double operator()(double amount){
        return amount + (amount * rate);
    }

};

int main() {

    TaxCalculator tc(0.18); // gst //object creation

//  tc();   not valid //error: no match for call to '(TaxCalculator) ()'

    double tax = tc(80000);

    cout << "Education GST TAX : " << tax << endl;

    return 0;
}
```