Operator overloading

Introduction

- Look at the following statements;
- int a,b,c;
- c =a + b; // this is ok!
- Distance d1, d2, d3;
- d3 = d1 + d2; // is this ok???
- If not ok, what is the error here??

Introduction

- Aim is to make the user-defined data types behave in the same way as the built-in types
  - Adding two variables of user-defined type with the same syntax that is applied to a basic type
- The mechanism to provide new definition to the operators by associating additional functionality is called operator overloading

Exceptions

- All operators in C++ can be overloaded except the following list
- Class member access operator (. and .*)
- Scope resolution operator (::)
- Sizeof operator (sizeof())
- Conditional operator (? :)

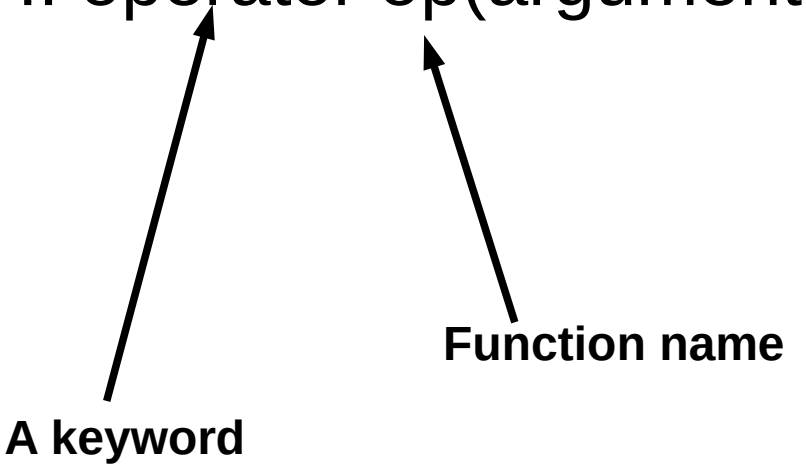
Important rules

- We cannot change the syntax or the grammatical rules that governs the use of the operators
- When an operator is overloaded its original meaning is not lost
- Only existing operators are overloaded. New operators cannot be created
- Must have at least one operand of user-defined type
- Overloaded operator follow the syntax of original operator.
- When some binary operators overloaded throught a member function, the left-hand operand must be an object of relevant class.
- +, -, *, and / etc. Must explicitly return value.

Operator overloading definition

- The operator overloading is defined by a special funcion, called **operator function**
- Syntax:

```
return_type class_name :: operator op(arguments)
{
        // function_body
}
```



Operator function

- Operator function can be defined either
  - As a non-static member function of a class, or
  - Friend function of a class
- Member operator function will take no argument for unary operator and one argument for binary operator
- Friend operator function will take one argument for unary operator and two arguments for binary operator

Operator function(2)

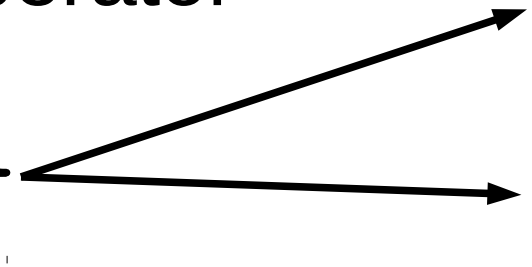
- An operator function must be a friend function if the left-most operand must be of a different class or a C++ intrinstic type
- An operator implemented as a friend function must explicitly have argument(s) of the object(s)

Operator function prototypes

- Distance operator+(Distance);
- Distance operator-(Distance);
- friend Distance operator+(Distance,Distance);
- friend Distance operator+(float,Distance);
- friend Distance operator++(Distance);
- friend Distance operator-(Distance);
- bool operator==(Distance);
- friend bool operator==(Distance,Distance);

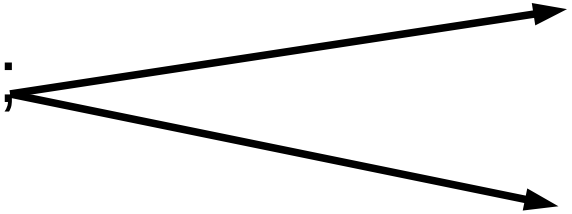
Method of invoking operator function

- Unary operator
  - op x;
  - x op;



- **operator op(x)**
  //friend function
- **x.operator op()**
  //member function

- Binary operator
  - x op y;



- **x.operator op(y)**
  //member function
- **operator op(x,y)**
  //friend function

Example : binary operator (one argument)

```
class Distance{
        int feet;
        float inches;
    public:
        Distance() : feet(0),inches(0.0){}
        Distance(int f, float I) : feet(f),inches(i) {}
        void showDistance(){
            cout << "Feet : " << feet << " Inches :
                    " << inches << endl;
        }
        Distance operator+(Distance);
};
Distance Distance::operator+(Distance d){
    int f=feet+d.feet;
    float i=inches+d.inches;
    if(i>=12.0){
            i-=12.0;
            f++;
    }
    return Distance(f,i);
}
```

right d2 vala by default aa rha h d ki jagah , left vala d1 ka type similar to left vale k hona chahiye kuki function operator overloading ka h otherwise friend function banana padega

```
main(){
    Distance d1(10,6), d2(11,5);
    Distance d3=d1+d2;
    d1.showDistance();
    d2.showDistance();
    d3.showDistance();
    Distance d4=d1+d2+d3;
    d4.showDistance();
}
```

Example : binary operator (two argument)

```
class Distance{
        int feet;
        float inches;
    public:
        Distance(): feet(0),inches(0.0){}
        Distance(int f, float i): feet(f),inches(i) {}
        void showDistance();
        friend Distance operator+(Distance,Distance);
};
Distance operator+(Distance d1, Distance d2){
      int f=d1.feet+d2.feet;
      float i=d1.inches+d2.inches;
      if(i>=12.0){
              i-=12.0;
              f++;
      }
      return Distance(f,i);
}
```

```
main(){
      Distance d1(10,6), d2(11,5);
      Distance d3=d1+d2;
      d1.showDistance();
      d2.showDistance();
      d3.showDistance();
      Distance d4=d1+d2+d3;
      d4.showDistance();
}
```

Example : unary operator (member function)

```
class Distance{
        int feet;
        float inches;
    public:
        Distance() : feet(0),inches(0.0){}
        Distance(int f, float i): feet(f),inches(i) {}
        void showDistance();
        void operator ++(); //prefix
        void operator ++(int); //postfix
};
void Distance::operator ++(int){
    feet++;
    inches++;
    if(inches == 12.0){
            inches = 0.0;
            feet++;
    }
}
void Distance::operator ++(){
    ++feet;
    ++inches;
    if(inches == 12.0){
            inches = 0.0;
            ++feet;
    }
}
```

```
main(){
    Distance d1(10,6), d2(11,5);
    d1++;
    d1.showDistance();
    ++d2;
    d2.showDistance();
}
```

Example : unary operator (friend function)

```
class Distance{
        int feet;
        float inches;
    public:
        Distance(): feet(0),inches(0.0){}
        Distance(int f, float i): feet(f),inches(i) {}
        void showDistance();
        friend void operator ++(Distance&);
        friend void operator ++(Distance&,int);
};
void operator ++(Distance &d){
      d.feet++;
       d.inches++;
       if(d.inches == 12.0) {
                d.inches = 0.0;
                d.feet++;
       }
}
void operator ++(Distance &d,int){
            d.feet++;
       d.inches++;
       if(d.inches == 12.0) {
                d.inches = 0.0;
                d.feet++;
       }
}
```

```
main(){
      Distance d1(10,6), d2(11,5);
      d1++;
      d1.showDistance();
      ++d2;
      d2.showDistance();
}
```

Matrix & Vector product

```
class Matrix;
class Vector{
      int v[4];
      public:
      void setVector(int vec[4]){
            for(int i=0;i<4;i++)
                  v[i]=vec[i];
      }
   friend Vector operator *(Matrix &,Vector &);
};
class Matrix{
      Vector m[4];
      public:
      void setMat(int[][4]);
    friend Vector operator *(Matrix&, Vector&);
};
Vector operator *(Matrix& mat, Vector& vec){
      Vector res;
      for(int i=0;i<4;i++){
            res.v[i]=0;
            for(int j=0;j<4;j++)
                res.v[i]+=mat.m[i].v[j]*vec.v[j];
      }
      return res;
}
```

```
void Matrix::setMat(int mat[][4]){
    for(int i=0;i<4;i++)
          m[i].setVector(mat[i]);
}
int main(){
      Matrix mat1;
      Vector vec1,vec2;
      int v[4], m[4][4];
      //input array v and matrix m
      vec1.setVector(v);
      mat1.setMat(m);
      vec2 = mat1 * vec1;
      cout << "Result vector" <<endl;
      return 0;
}
```

Mix mode overloading

- Complex c,c1;
- double d;
- c+=c1; c+=d; c=d+c1; c=c1+d;
- How to execute??

```
//member functions
Complex operator +=(Complex a){
    re+=a.re;
    im+=a.im;
    return *this;
}
Complex operator +=(double a){
      re+=a;
      return *this;
}
//friend functions
Complex operator +(double a, Complex b){
      Complex res = b;
      res+=a;
      return res;
}
Complex operator +(Complex a, double b){
      complex res=a;
      res+=b;
      return res;
}
```

Complex res = b.re ;

Complex res = a.re ;

Overloading
stream insertion and extraction operators

```
Class Complex{
    …
    friend void operator<<(ostream&,Complex&);
    friend void operator>>(istream&,Complex&);
};
void operator<<(ostream& out,Complex &c){
    out<<c.re<<“ + i”<<c.img<<endl;
    return;
}
void operator>>(istream& in,Complex &c){
      in>>c.re>>c.img;
      return;
}
```

```
main(){
      Complex cx1;
      cout << “input
      complex no. details “
      cin >> cx1;
      cout << “ Number : “
      cout << cx1;
      ...
}
```

Note

- These stream insertion and extraction operators functions must be non-members as the objects of class Complex appears in each case as the right operand and the function can not be a part of class ostream and istream

Example: overloading []

```
#define MAX 10

class MyIntQueue{
        int element[MAX];
        int front;
        int rear;
        int size;
        public:
        MyIntQueue():front(-1),rear(-1),size(0){}
        int getSize(void){ return size; }
        void insert(int);
        int del(void);
        int &operator[](int ind);
        void showAll(void);
};

...

int & MyIntQueue::operator[](int ind){
        if(ind<1 || ind>size){
                cout << "Boundary Error" << endl;
                exit(1);
        }
        else
                return this->element[(front+ind-
1)%MAX];
}
```

```
int main(){
        MyIntQueue q1;
        q1.insert(11);
        q1.insert(12);
        ...
        ...
        cout << "Display by [] operator " <<
endl;
        for(i=1;i<=q1.getSize();i++)
                cout << q1[i] << " ";
        cout << endl;
        cout << "enter a value for i";
        cin >> i;
        cout << "element in position " << i << "
is "  << q1[i] << endl;
        return 0;
}
```

Type Conversion

Type conversion

- '=' is a special operator with complex properties.
- When both sides are of equal type, the compiler does not need any special instructions to operate.
- If they are different ?
- For basic data types
  - Implicit conversion are done automatically

    int var1; float var2 = 83.57;

    var1 = var2; //allowed
  - Explicit conversion is done using **type casting**.
- Each such conversion has its own routine, built into the compiler.

Type conversion

- Compiler does not support automatic conversion of user-defined datatypes
  - Vector vec; Matrix max;
  - max = vec; //error!!
- If we need such conversion, then we may design conversion function explicitly in our program.
- Three types of conversion are possible
  - Basic type to class type
  - Class type to basic type
  - One class type to another class type

Basic type to user-defined type conversion

- This can be accomplished by defining constructors to build a user-defined type object from basic data type

```
class Distance{
        int feet;
        float inches;
    public:
        Distance()feet(0),inches(0.0){}
        Distance(float f){
            feet=int(f);
            inches=12*(f-feet);
        }

};
int main(){
     Distance d1;
     float distcovered = 85.75;
     d1 = distcovered;
     ...
}
```

User-defined type to basic type

- C++ provides an overloaded casting operator that could be used to convert a class type data to a basic type.
- These functions generally referred as conversion function.

```
class Distance{
        int feet;
        float inches;
    public:
        Distance()feet(0),inches(0.0){}
        operator float(){
            float ft=inches/12;
            ft+=float(feet);
            return ft;
        }
};
int main(){
     Distance d1;
     float distcovered;
     distcovered = d1;
     ...
}
```

One class type to another class type conversion

```
class Rect{
    double xco;
    double yco;
    public:
            …
            }
class Polar{
    double radius;
    double angle;
    public:
operator Rect(){
        double x=radius*cos(angle);
        double y=radius*sin(angle);
        return Rect(x,y);
        } //inside class declaration
}; //class ends here
```

```
main(){
    Rect rec; Polar pol(10.0,0.785398);
    rec=pol;
    pol.display();
    rec.display();
}
```

End of
Operator overloading