Object Oriented Programming

C++ Operator Overloading

Abeeda Akram

Member Functions

- Can be defined inside class as member or just add prototype and define outside as normal member functions.
- Operators that must be overloaded through member functions are:

  =, [], (), ->, &(address of operator)
- **Binary operators:**
  - Member function, needs one argument right operand can be class object or other datatype.
  - Left operand must be class object
- All operators can be overloaded through member functions in which left operand is class object for example:

```
Point p1, p2(2, 3);
p1 + p2; //both are class objects of Point class
p1++;
p1 = p2;
//left operand is class object member function will work
p1 + 3;
```

Binary Operator Addition (+)

- **Both operands are class objects.**
- Member function **takes right operand** of operation as one argument.
- Called on left operand must be class object.
- Can be called in two ways.

```
Point p1(3, 4), p2(3, 2);
p1.operator+(p2); // called on p1
Or
p1+p2;
// called on p1, p2 passed as argument
Point p3 = p1+p2;
// cascaded call
```

```
class Point {
    int x, y;
public:
    Point(int a=0, int b=0) { x=a; y=b;}
    Point operator+ (const Point&);
};
//implementation
Point Point:: operator+(const Point& p){
    Point R;
    R.x = x + p.x;
    R.y = y + p.y;
    return R;
}
```

Binary Operator Addition (+)

- **One operand left one is class object**.
- Member function **takes right operand** of operation as one argument.
- Called on left operand must be class object.
- Can be called in two ways.

```
Point p1(3, 4);
p1.operator+(3); // called on p1
Or
p1+10; // called on p1, int 10 is passed as argument
int a = 10;
Point p3 = p1+a; // cascaded call
```

```
class Point {
    int x, y;
public:
    Point(int a=0, int b=0) { x=a; y=b;}
    Point operator+ (const Point&);
    Point operator+ (const int);
    // with int
};
//implementation
Point Point:: operator+(const int n){
    Point R;
    R.x = x + n;
    R.y = y + n;
    return R;
}
```

Binary Operator is equal to (==)

- Both operands should be class objects.
- Member function takes right operand of operation as one argument.
- Called on left operand must be class object.
- Can be called in two ways.

```
Point p1(3, 4), p2(3, 2);
p1.operator==(p2); // called on p1
Or
cout << (p1==p2);
```

```
class Point {
    int x, y;
public:
    Point(int a=0, int b=0) { x=a; y=b;}
    bool operator==(const Point&);
};
//implementation
bool Point:: operator==(const Point& p){
    if (x == p.x && y == p.y)
        return true;
    else
        return false;
}
```

Binary Operator is not equal to (!=)

- Both operands should be class objects.
- Member function takes right operand of operation one argument.
- Called on left operand must be class object.
- Can be called in two ways.

```
Point p1(3, 4), p2(3, 2);
p1.operator!=(p2);
// called on p1
Or
cout << (p1!=p2);
```

```
class Point {
    int x, y;
public:
    Point(int a=0, int b=0) { x=a; y=b;}
    bool operator==(const Point&);
    bool operator!=(const Point&);
};
//Reuse == operator function
bool Point:: operator!=(const Point& p){
    return !((*this) == p) ;
}
```

Binary Operator Assignment (=)

- **Member function is compulsory for assignment.**
- **Both operands should be class objects**
- **Member function takes right operand of operation as argument.**
- **Called on left operand that must be class object.**
- **Check state of both left and right object's data members carefully.**
    1. If they are pointers address issues, due to different constructors, nullptr or valid memory address.
    2. Dynamic arrays size mismatch issues.
    3. Self assignment issue with pointer data members.
- Can be called in two ways.

```
Point p1(3, 4), p2(3, 2), p3;
p1.operator=(p2);
// called on p1
Or
p1=p2; // called on p1
p1=p2=p3; // cascaded call
```

Left Operand	Right Operand
nullptr	nullptr
nullptr	Address
Address	nullptr
Address (Single Variable)	Address (Single Variable)
Address (Array Size Check)	Address (Array Size Check)

Binary Operator Assignment (=)

```
class Point {
    int x, *y;
public:
    Point() { x=0; y=nullptr; }
    Point(int x, int y) {
        this->x=x;
        this->y=new int(y);
    }

    Point& operator=(const Point& p);
};
```

```
//implementation
Point& Point:: operator=(const Point& p){
  if (this != &p) {
        x = p.x;
        if(y==nullptr && p.y!=nullptr)
           y = new int(*(p.y));
        else if(y!=nullptr && p.y==nullptr){
           delete y;
           y = nullptr;
        }
        else if(y!=nullptr && p.y!=nullptr)
           *y = *(p.y);
    // if arrays deep copy using loops
  }
  return *this;
}
```

Binary Operator Subscript [ ]

- **Member function is compulsory for subscript operator.**
- **Left operand should be class object and right should be int.**
- **Member function takes right operand of operation as argument and called on left operand.**
- **It provides access to elements of arrays defined inside objects as private data members.**
- For example: a class myArray is defined here.

```
class myArray{
   int size; // Array size
   int *ptr; // Pointer for dynamic 1-D Array
public:
   myArray() { size=0; ptr=nullptr; }
   myArray(int size) {
   this->size=size;
   if(size>0){
   ptr = new int[size];
   for (int i = 0; i < size; i++)
   ptr[i] = i + 1;
   }
   else
   ptr = nullptr;
   }
};
```

- If subscript operator is overloaded, then we can access the elements of private array in following way.

```
myArray a1(5);
// creates array inside object
// Subscript operator call for
array inside object
a1[0] = 100;
// store 100 in element 1
a1[1] = a1[0];
// copy element 1 to element 2
```

Binary Operator Subscript [ ]

```
class myArray{
    int size; // Array size
    int *ptr; // Pointer for dynamic 1-D Array
public:
    myArray() { size=0; ptr=nullptr; }
    myArray(int size) {
    this->size=size;
    if(size>0){
    ptr = new int[size];
    for (int i = 0; i < size; i++)
    ptr[i] = i + 1;
    }
    else
    ptr=nullptr;
    }
    int & operator[](const int i);
    const int & operator[](const int i) const;
};
```

```
//implementation for Normal object
int & myArray::operator[](const int i){
  // check if index i is in range
  if ( i>=0 && i<size)
    return ptr[i];
    // return element by reference as lvalue
}
//implementation accessor for Constant object
const int & myArray::operator[](const int i) const{
  // check if index i is in range
  if ( i>=0 && i<size)
    return ptr[i];
    // return element by reference as constant rvalue
}
```

Binary Operator Subscript [ ]

- Subscript operator call for array inside object

```
myArray a1(5); // creates array inside object
a1[0] = 100; // return reference to int element 1 of array
// store 100 in element 1
a1[1] = a1[0]; // copy element 1 to element 2
cout<< a1[1]; // print value of element 2

const myArray a2(3); // creates array of size 3 inside constant object
cout<< a2[1];         // return constant reference (read only) to int
a2[1] = 10; // wrong as constant reference is returned for constant object
```

- Not work on pointers to objects directly

```
myArray *aptr = new myArray(5); // creates array inside object
aptr[3] = 100; // wrong as aptr is pointer
(*aptr)[3] = 100;  //first dereference the pointer then access data
aptr[0][3] = 100;
```