Overload of operator

Declaration of operator

- C++ can declare new functions for operators (+, -, *, /, %)
  - Also applicable for ( ), [ ], ->, =, ==, &&, ||, ^, >>, != , etc.
  - Not all operators are applicable
    - .(dot), :: (scope), ?: (ternary operators), .* (dot for member pointer), and preprocessor operators are NOT applicable
  - Default arguments can not be used

- Replace operations
  - Simplify statement
  - Composite process can be written in a line with parenthesis()
  - Useful for the class of vector, matrix, etc.
  - Operator can be used in a universal way

Sample code of operators

```
class Point {
private:
    int x, y;
public:
    Point (int x = 0, int y = 0) { // constructor
        this->x = x; this->y = y; }
    Point operator + (const Point& p) { // addition
        return Point (x + p.x, y + p.y);
    int operator * (const Point& p) { // dot product
        return x * p.x + y * p.y;
    }
};
```

Difference against member function

```
Point plus (Point& p) {
    return Point (x + p.x, y + p.y);
}

Point p2 = p0.plus(p1);

Point operator + (Point& p) {
    return Point (x + p.x, y + p.y);
}

Point p2 = p0 + p1; // p0.(operator +)(p1)
```

How to use

```
int main () {
    Point p0 (1,2);
    Point p1 (3,4);
    Point p2 = p0 + p1; // p2 ← (4, 6)
    int d = p2 * p1; // d ← 36
    Point p3 = p2 + 1; // p3 ← (5, 6)
    // as this corresponds operator + (Point& p)
    // 1 → Point(1) is implicitly called
}
```

Scalar product

Point.h

```
class Point {
    ...
    Point operator * (int k); // vector * scalar
    friend Point operator * (int k, Point pnt); // scalar*vector
};
```

Point.cpp

```
Point Point::operator * (int k) { // vector * scalar
    return Point (x * k, y * k);
}
Point operator * (int k, Point pnt); // Point:: is omitted
    return Point (k * pnt.x, k * pnt.y);
}
```

Operator of non-member function

- Operator takes two arguments when its left-side operand is NOT an object
  - In this case, the operator is NOT regarded as member function, and therefore a prefix such as ClassName:: is removed
- It must be a friend function for accessing hidden (private) member variables
  - Such function can access all members of the class in which they are declared

Execution of scalar product

```
int main () {
    Point p0(1,2);
    Point p1 = p0 * 3; // called as member function
    Point p2 = 2 * p1; // called as ordinary function
    // p1 == (2, 6) , p2 == (4, 12)
}
```

Making unary operator

- Unary operators +, -, !, ~ have no argument
- Operators that are originally unary can be used

```
class Point {
    …
    Point operator – () {
        return Point (–this–>x, –this–>y);} // OK!
    int operator ^ () {// Compile Error because ^ is a binary operator
        return this–>x * this–>y; }
```

Make use of const modifier

```
class Point {
private:
    int x, y;
public:
    Point (int _x, int _y) { x = _x; y = _y; }
    Point operator + (const Point& p) {//p can NOT be updated
        return Point2D (x + p.x, y + p.y);
    }
    int operator * (Point& p) const { // All of x, y, p can NOT be
updated
        return x * p.x + y * p.y;
    }
};
```

Assignment operator

- Assignment operator (=) can be overloaded

```
Point& operator = (const Point& p) {
    x = p.x; // copy of x value
    y = p.y; // copy of y value
    return *this; // return itself
}
```

Copy constructor

- Constructor is called copy constructor if it takes the argument of self-class
- Though assignment and copy constructor have a similar form, different mechanisms are invoked

```
Point(const Point& p) : x(p.x), y(p.y){}
```

Difference between assignment and copy constructor

Point p0(1,2), p1(3,4); // ordinary constructor

Point p2 = p1; // copy constructor for initializing p2, where (3,4) is assigned to (x,y) of p2

p2 = p1 = p0; // assignment is applied from right-side expression, which assigns all coordinated of p2, p1, p0 by (1,2)

Point3D p3d(1,2,3); // Derived class of Point

p0 = p3d; // p0 is NOT converted to Point3D, because it only copies the values of its member variables from p3d

Point p2d = p3d; // In copy constructor, p2d also is NOT converted to Point3D

Conversion to derived class

Point p;

Point3D p3d(1,2,3); // derived class of Point

p = p3d; // NOT converted to Point3D

Point pc = p3d; // NOT converted to Point3D

Point &pr = p3d; // reference can store the p3d as Point3D class

Point *pp = new Point3D(4,5,6); // pp is the instance of Point3D, because it stores a pointer

pp = &p3d; // similar case to the above statement