Operator Overloading

Not in Java !

```
class Vector
{  public:
      Vector( ) { }
      Vector(double a, double b){x= a; y= b;}
      Vector(double a){x=a; y=0;}

      Vector add(const Vector& V)const;
      Vector subtr(const Vector& V)const;
      …
   private:
      double x, y;
};
```

use:

```
Vector a, b(1.2, 3.4), c(7.33, 8.0);
a= b.add(c);
```

Use Operator Overloading

```
class Vector
{  public:
      Vector( ) { }
      Vector(double a, double b){x= a; y= b;}
      Vector(double a){x=a; y=0;}

      Vector operator+(const Vector& V)const;
      Vector operator-(const Vector& V)const;
      Vector operator*(double k)const {return Vector(k*x,k*y);};
      double inner(const Vector& V)const;

   private:
      double x, y;
};

Vector Vector:: operator+(const Vector& V)
      { return Vector(x + V.x, y + V.y);}

Vector Vector:: operator-(const Vector& V)
      { return Vector(x - V.x, y - V.y);}
```

!

use:

```
Vector a, b(1.2, 3.4), c(7.33, 8.0);
      a= b + c;     // ~ a= b.operator+(c)
      c= b * 2.0; // ~ c = b.operator*(2.0)
```


Member Methods/Operators Assymetrical

```
#include "vector.h"
Vector a, b(1.2, 3.4), c(7.33, 8.0);
double k= 2.1;
a= b + c;       // ~ a= b.operator+(c)
a= b + k;       // OK, ~ a= b.operator+(Vector(k))
a= k + b;       // ILLEGAL, k is not an object

b= a * k;       // OK,  ~ b= a.operator*(k)
b= k * d;       // ILLEGAL, k is not an object

k= a.inner(b);
double e= a.inner(k)   // OK ~ e= a.inner(Vector(k))
e= k.inner(a)          // ILLEGAL, k is not an object
```

Friends

Friend Functions:

```
class Vector
{  public:
      Vector( ) { }
      Vector(double a, double b){x= a; y= b;}
      Vector(double a){x=a; y=0;} //conversion constructor

      double inner(const Vector& V)const;

      friend Vector operator+(const Vector& V1,const Vector& V2);
      friend Vector operator-(const Vector& V1,const Vector& V2);

      friend Vector operator*(const Vector& V,double k);
      friend Vector operator*(double k,const Vector& V);
   …
   private:
      double x, y;
};
```

Not class methods but *friend functions*. Have access to Vector's private data

```
double Vector::inner(const Vector& V)const
   {  return x * V.x + y * V.y;}

Vector operator+(const Vector& V1,const Vector& V2)
   {  return Vector(V1.x + V2.x, V1.y + V2.y);}

Vector operator-(const Vector& V1,const Vector& V2)
   {  return Vector(V1.x - V2.x, V1.y - V2.y);}

Vector operator*(const Vector& V,double k)
   {  return Vector(V.x*k + V.y*k);}

Vector operator*(double k, const Vector& V)
   {  return Vector(V.x*k + V.y*k);}
```

```
Vector V1(2.0,3.0),
V2(4.0,5.0);
V1= V1 + V2;
V1= V1+3.3;
V1= 2.1+ V2;
```


Friend Classes

```
class Slave1
{  friend class Master
   private:
      ... prs1a
      ... prs1b    // private members
      ... ...
}

class Slave2
{  friend class Master
   private:
      ... prs2a
      ... prs2b    // private members
      ... ...
}


class Master
{  public:
      ... pubF1(...)
      ... pubF2(...)
   private:
      ... prm1
      ... prm2
}

... Master::pubF1(...){  prs1a ... prs1b ... prs2a ... prs2b ... prm1 ...prm2 }
... Master::pubF2(...){  prs1a ... prs1b ... prs2a ... prs2b ... prm1 ...prm2 }
```

All Slave-objects and their members are accessed/used only trough a Master object.

The Output Class ostream

```
class ostream
{  ostream& operator<<(int i);
   ostream& operator<<(long i);
   ostream& operator<<(double x);
   ostream& operator<<(char c);
   ostream& operator<<(const char* s);

   …
   ostream& put(char c);
   …
}
```

The *insertion* or *put to* operator

Use: ~ cout.operator<<("Here is an integer")

```
cout  <<  "Here is an integer:"  <<  23  <<  "and a double: "  <<  23.45
```

ostream&	
ostream&	
ostream&	
ostream&	
ostream&	

Overloading I/O Operators << and >>

```
#include <iostream>

class Vector
{  public:
     Vector( ) { }
     Vector(double a, double b){x= a; y= b;}
     …
     friend ostream& operator<<(ostream& out, const Vector& V);
     friend istream& operator>>(istream& in, Vector& V);
     …
   private:
     double x, y;
};


ostream& operator<<(ostream& out, const Vector& V)
   {  return (out << "(" << V.x << "," << V.y << ")");
   }

istream& operator>>(istream& in, Vector& V)
   {return (in >> V.x >> V.y);
   }


int main()
{ Vector a,   b(3.0, 4.0),   c(7.33, 8.0);
   cout << "b+c= " << b+c << endl;
   cin >> a ;
   cout << "a= " << a << ", b= " << b << ", a*b= " << a.inner(b) << endl;
}
```

Output/Input:
b+c= (10.33,12)
2.4
5.6
a= (2.4,5.6), b= (3,4), a*b= 29.6

Overloading Assignment and Subscripting

Every class may be considered to have a *default assignment operator* =

```
class A
{ …
  A& operator=(A);
…
}
```

The default assignment operator makes memberwise/shallow assignment. Compare with *default copy-constructor*.

A safe array type dbl_vect with = and [] overloaded

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

class dbl_vect {
  public:
    explicit dbl_vect(int n = 10);
    dbl_vect(const dbl_vect& v);
    ~dbl_vect() { delete []p; }

    double&  operator[](int i) ;
    dbl_vect& operator=(const dbl_vect& v);

  private:
    double*  p;
    int  size;
};

double& dbl_vect::operator[](int i)
{  assert(i >= 0 && i < size); return p[i];  //return reference to p[i]
}

dbl_vect& dbl_vect::operator=(const dbl_vect& v)
{  if (this != &v)       //do nothing if assigned to self
     {  assert(v.size == size);
        for (int i = 0; i < size; ++i) p[i] = v.p[i];
     }
  return *this;
}
```

Make *deep* assignment

A safe array type dbl_vect with = and [] overloaded (2)

```
int main()
{  dbl_vect x(5), y(5);
   for (int i=0; i<=x.ub(); ++i){x[i]= i+ i/10.0;}
   x.print();
   y= x;
   for (int i=0; i<=x.ub(); ++i){y[i]= y[i] + 1.1;}
   y.print();
   x.print();
}
```

```
0    1.1    2.2  3.3    4.4
1.1    2.2    3.3    4.4    5.5
0    1.1    2.2    3.3    4.4
```

x[n]	~	x.operator[](n)
x = y	~	x.operator=(y)