CURS 6

30.III.2011

Operators overloading. Static members

- overloading operators ~ overloading functions
- in order to overload the "+" operator ⇒ introduce a method called operator+ and specify the returned type and parameters

a+5 ⇒ a.operator+(5)

user (prog) — compiler

- a "syntactic sugar"

Rules:

- you cannot overload operators that do not exist
- you can not change the arity (the no of of an operator)
- you can not change the priority of the operators
- operators that cannot be overloaded: ., ::, ?:
- 2 ways of overloading op { member function / global function

1. Overloading unary operators

```
class Integer { int val;
        public
            Integer () {val = 0;}
            Integer (int x) {val = x; }
            Integer (const Integer & c) {val = c.val;}
            const Integer & operator + () const {
                return * this;
            }
            const Integer operator - () const {
                return Integer (-val) ;}

            const Integer * operator & () const {
                return this; }

    const Integer & operator ++ ()      // ++a
      {
          val ++
          return * this ;          dummy arg.
      }                                ↘
    const Integer operator ++ (int)     // a ++
      { Integer before (val);
        val ++;
        return before; }
```

2. Overloading binary operators

```
Integer.h
class Integer {
    int val;
public:
    Integer(int v=0) { val=v; }   // default constr. general constr
```

```
const Integer operator+(const Integer &) const;
const Integer operator-(const Integer &) const;
const Integer operator*(const Integer &) const;
const Integer operator/(const Integer &) const;
Integer & operator=(const Integer &);
int operator==(const Integer &) const;
int operator<(const Integer &) const;
int operator<=(const Integer &) const;
int operator>(const Integer &) const;
int operator>=(const Integer &) const;
int operator!=(const Integer &) const;
Integer & operator+=(const Integer &)
Integer &          -=
Integer &          *=
Integer &          /=

};
```

Integer.cpp

```
#include "Integer.h"

const Integer Integer::operator+
    (const Integer &right) const
        { return Integer(val + right.val); }

const Integer Integer::operator-(const Integer &
        right) const

    { return Integer( val - right.val); }

Integer &Integer::operator=(const Integer &
        right)
    { if (this != &right) { val = right.val; }
    return this; }
```

Integer & — chained operators.

```
int Integer::operator==(const Integer &right) const
    { return (val == right.val); }
```

<, <=
>, >=
!=

```
Integer &Integer::operator+=(const Integer &right)
    { if (this != &right) {
        val += right.val; }
```

Overloading by global functions (friends)

friend - not a member of a class, but having access to its private members
- the friend declaration is performed only once using the friend keyword.

```
class X
  { int a
    public
        X(int y=5) { a=y; }
        friend void print(X);
        friend X::g(X);
        friend class Z; }

void print(X p)
   { cout << p.a << endl; }

class Y { ...
        void g(X p){ p.a ++ }
#include <iostream> using namespace std
class Integer {
        int val
    public:
        Integer(int x=0) { val=x; }
        friend Integer operator +( const Integer
                left, const Integer right);
    ...
}
```

```
Integer operator + (const Integer left, const Integer
                    right)
    { return Integer (left.val + right.val); }

    friend istream & operator >> (istream &s,
                                  Integer &i)

    friend ostream & operator << (ostream &s,
                                  Integer &i)

istream & operator >> (istream &is, Integer &i)
    { is >> i.val
      return is; }

ostream & operator << ostream &os, Integer &i)
    { os << i.val << endl
      return os; }

int a=5
Integer x;
Integer S= x+a
Integer s1=a + x ; Iterator.

friend Integers operator + (int a, Integer S)
    { return Integer (a + S.val); }
```

Overloading [ ] (index op)

- always overloaded as a member
- it always return a reference (l. h. v)

```
class Vector {
        int elem[100];
        int sz;
    public:
        Vector() { sz = 0; }
        int & operator[](int index)
            { return elem[index]; };
```

```
Vector v
v[0] = 7 ...
cout << v[2];
```

Static members

- a static member belongs to the class not to objects (like a shared memory)
- static members are initialized outside the class
- accessed by using an object or the class name followed by the resolution operator

```
class A
    { static int i;
      public ... {
      int A::i=0
      cout << A::i
```

Singleton Design Pattern

- creational pattern
- allows the creation of one unit instance of a class

```
class Singleton {
    static Singleton* instance;
    static bool instanceFlag;
    Singleton() {...}
    Singleton(const Singleton
              &s)
    Singleton(T &) {...}
public:
    Singleton * getInstance
    () { if (!instanceFlag) { instance = new Singleton
                              instanceFlag = True; }
         return *instance }
```

Singleton
-instance: Singleton
-instanceFlag: Boolean
-Singleton() throws Singleton(Singleton) throws Singleton(T) throws + getInstance(): Singleton

```
~ Singleton() { instanceFlag = false; };

Singleton * Singleton::instance=0)    // NULL
bool Singleton::instanceFlag = false;

int main()
{ Singleton * S1 * S2

    S1 = Singleton::getInstance();
    S2 = Singleton::getInstance();
    cout << S1 << " " << S2 << endl;

}
```

Class that counts its instances

```
class CountObjects
    { static int count;
      public:
        CountObjects() { count++; }
        CountObjects(const CountObject &c)
                { ... count++; }

        CountObjects(T &) { ... count++ }
      ~ CountObjects() { count--; }
```

```
static int getNoOfInstance() { return count; }
{
int CountObjects :: count=0;
int main()
    { CountObjects c1, c2, c3)
        count = CountObjects :: getNoOfInstances();
```