OOP laboratory_03

Terman Emil FAF161

September 29, 2017

Conf. unv. prof: M. Kulev

LaTeX

1 Subject

Supraîncărcarea operatorilor

2 Objectives

- Studierea necesităţii supraîncărcării operatorilor
- Studierea sintaxei de definire a operatorilor
- Studierea tipurilor de operatori
- Studierea formelor de supraîncărcare

3 Task

a) Să se creeze o clasă de numere întregi. Să se definească operatorii "++" şi "+", ca metode ale clasei, iar operatorii "- -" şi "-" ca funcţii prietene. Operatorii trebuie să permită efectuarea operaţiilor atît cu variabilele clasei date, cît şi cu variabilele întregi de tip predefinit.

b) Să se creeze o clasă Set – mulţimea numerelor întregi, utilizînd memoria dinamică. Să se definească operatorii de lucru cu mulţimile: "+" – uniunea, "*" – intersecţia, "-" scăderea, ca funcţii prietene, iar "+=" – înserarea unui nou element în mulţime, "==" – comparare la egalitate, ş. a. ca metode ale clasei. Să se definească operatorii "<<" şi ">>". Să se definească funcţia de verificare a apartenenţei unui element la o mulţime.

4 Main notions of theory and used methods

In programming, operator overloading, sometimes termed operator ad hoc polymorphism, is a specific case of polymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by a programming language, a programmer, or both.

5 Data analysis

5.1 Ex a

```hpp
#ifndef MYINT_HPP
# define MYINT_HPP

# include <ostream>

class MyInt {
public:
    //Getters
    int     getVal(void) const;

    //Constructors
    MyInt(void);
    MyInt(int value);
    MyInt(MyInt const & target);

    //Arithmetic operators
    MyInt operator + (MyInt const & target) const;
    MyInt operator + (int nb) const;

    friend MyInt operator - (MyInt const & target1, MyInt const & target2);
    friend MyInt operator - (MyInt const & target, int nb);
    friend MyInt operator - (int nb, MyInt const & target);

    MyInt & operator ++ (void);
    MyInt operator -- (int);

    MyInt const & operator = (MyInt const & target);

    //Other operators
    operator int() const;

private:
    int _value;
};

std::ostream & operator  << (std::ostream & o, MyInt const & target);

#endif
```

- 1. MyInt operator + (MyInt const & target) const;
  2. MyInt operator + (int nb) const;

  Constant overload operators, defined as class methods. The first is used on another instance of the same class, the second is used with a simple int.

- 1. friend MyInt operator − (MyInt const & target1, MyInt const & target2);
  2. friend MyInt operator − (MyInt const & target, int nb);
  3. friend MyInt operator − (int nb, MyInt const & target);

  Friend overload operators, used for each case.

- 1. MyInt & operator ++ (void);
  2. MyInt operator −− (int);

  The first is a method's overload operator, the second is defined as a friend operator.

5.2 Ex b

```cpp
#ifndef SET_TPP
# define SET_TPP

# include <ostream>
# include <exception>
# include <iostream>
# include <istream>
# include <initializer_list>

/*
** Custom operators
*/

const struct _opSetIn_ {} in = {};

template <typename T>
class Set {
public:

    /*
    ** Exceptions
    */

    class IndexOutOfRange : public std::exception {
    public:
        virtual char const * what() const throw() {
            return "Index out of range";
        }
    };

    /*
    ** Getters
    */

    size_t      count(void) const;
    T           getDefaultVal(void) const;

    /*
    ** Constructors & Destructors
    */

    Set<T>(void);
    Set<T>(size_t const size);
    Set<T>(size_t const size, T const defaultVal);
    Set<T>(Set<T> const & target);
    Set<T>(T const defaultVal, std::initializer_list<T> args);
    ~Set<T>(void);

    //Operators
    Set<T> const &  operator = (Set<T> const & target);
    T &             operator [] (size_t const i) const;

    /*
    ** Arithmetic operators
    */

    template <typename U>
    friend Set<U>   operator + (Set<U> const & target1, Set<U> const & target2);

    template <typename U>
    friend Set<U>   operator * (Set<U> const & target1, Set<U> const & target2);

    template <typename U>
    friend Set<U>   operator - (Set<U> const & target1, Set<U> const & target2);

    Set<T> &        operator += (Set<T> const & target);

    /*
    ** Boolean operators
    */

    bool        operator == (Set<T> const & target);

    /*
    ** IO operators
    */

    template <typename U>
    friend std::ostream & operator << (std::ostream & o, Set<U> const & target);

    template <typename U>
    friend std::istream & operator >> (std::istream & is, Set<U> & target);

private:
    T *         _tab;
    size_t      _size;
    T const     _defaultVal;
};
```

- Set<T> & operator = (Set<T> const & target);

  Assign overload operator, which takes in a constant reference to the same class.

- T & operator [] (size_t const i) const;

Overload operator of squared braces, which returns a modifiable reference to the set's element.

- 1. friend Set<U> operator + (Set<U> const & target1, Set<U> const & target2);

  Union

  2. friend Set<U> operator * (Set<U> const & target1, Set<U> const & target2);

  intersection

  3. friend Set<U> operator − (Set<U> const & target1, Set<U> const & target2);

  Difference

  Friend arithmetic overload operators, which take in 2 constant references of the same type of set.

6 The actual code

6.1 Ex a

```cpp
#include "MyInt.hpp"

//Getters
int     MyInt::getVal(void) const {return _value;}

//Constructors
MyInt::MyInt(void) {
    _value = 0;
}

MyInt::MyInt(int const value) {
    _value = value;
}

MyInt::MyInt(MyInt const & target) {
    _value = target.getVal();
}

/*
** Operators
*/

MyInt const & MyInt::operator = (MyInt const & target) {
    _value = target.getVal();
    return *this;
}

//Pre
MyInt MyInt::operator + (MyInt const & target) const {
    return MyInt(getVal() + target.getVal());
}

MyInt MyInt::operator + (int nb) const {
    return MyInt(getVal() + nb);
}

MyInt & MyInt::operator ++ (void) {
    _value++;
    return *this;
}

/*
** Friendly
*/

//With class ref on both sides
MyInt operator - (MyInt const & target1, MyInt const & target2) {
    return MyInt(target1.getVal() - target2.getVal());
}

//With Class ref on the left
MyInt operator - (MyInt const & target, int const nb) {
    return MyInt(target.getVal() - nb);
}

//With Class ref on the right
MyInt operator - (int nb, MyInt const & target) {
    return MyInt(nb - target.getVal());
}

MyInt MyInt::operator -- (int) {
    MyInt   result(*this);

    _value--;
    return result;
}

//Other operators
MyInt::operator int() const {return getVal();}

std::ostream & operator  << (std::ostream & o, MyInt const & target) {
    o << target.getVal();
    return o;
}
```

main.cpp

```cpp
#include "MyInt.hpp"

#include <iostream>

int     main(void) {
    MyInt   a(3);
    MyInt   b(5);

    std::cout << "a + b = " << a + b << std::endl;
    std::cout << "a + 5 = " << a + 5 << std::endl;
    std::cout << "3 + b = " << 3 + b << std::endl;

    std::cout << "a - b = " << a - b << std::endl;
    std::cout << "a - 5 = " << a - 5 << std::endl;
    std::cout << "3 - b = " << 3 - b << std::endl;

    std::cout << "----------" << std::endl;
    std::cout << "a = " << a << std::endl;
    std::cout << "++a: " << ++a << std::endl;
    std::cout << "a = " << a << std::endl;

    std::cout << "----------" << std::endl;
    std::cout << "a--: " << a-- << std::endl;
    std::cout << "a = " << a << std::endl;
    std::cout << "----------" << std::endl;

    int intA = a;
    std::cout << "int aInt = a; aInt = " << intA << std::endl;

    return 0;
}
```

Output

```
a + b = 8
a + 5 = 8
3 + b = 8
a - b = -2
a - 5 = -2
3 - b = -2
----------
++a: 4
a = 4
----------
a--: 4
a = 3
----------
int aInt = a; aInt = 3

```

6.2 Ex b

```
1
2  /*
3  ** Getters
4  */
5
6  template <typename T>
7  size_t      Set<T>::count(void) const {return _size;}
8
9  template <typename T>
10 T           Set<T>::getDefaultVal(void) const {return _defaultVal;}
11
12 /*
13 ** Constructors & Destructors
14 */
15
16 //Default Constructor
17 template <typename T>
18 Set<T>::Set(void) : _defaultVal(0)
19 {
20     _tab = new T[0];
21     _size = 0;
22 }
23
24 //Constructor with given size
25 template <typename T>
26 Set<T>::Set(size_t const size) : _defaultVal(0)
27 {
28     _tab = new T[size];
29     _size = size;
30
31     for (size_t i = 0; i < size; i++)
32         _tab[i] = _defaultVal;
33 }
34
35 //Constructor with given size and default value
36 template <typename T>
37 Set<T>::Set(size_t const size, T const defaultVal) : _defaultVal(defaultVal)
38 {
39     _tab = new T[size];
40     _size = size;
41
42     for (size_t i = 0; i < size; i++)
43         _tab[i] = _defaultVal;
44 }
45
46 //Copy Constructor
47 template <typename T>
48 Set<T>::Set(Set<T> const & target) : _defaultVal(target.getDefaultVal())
49 {
50     _tab = NULL;
51     *this = target;
52 }
53
54 //Multiple variable constructor
55 template <typename T>
56 Set<T>::Set(T const defaultVal, std::initializer_list<T> args) :
57     _defaultVal(defaultVal)
58 {
59     int     i = 0;
60
61     _size = args.size();
62     _tab = new T [_size];
63
64     for (T element: args)
65         _tab[i++] = element;
66 }
67
68 //Destructor
69 template <typename T>
70 Set<T>::~Set(void)
71 {
72     delete [] _tab;
73 }
74
75 //Operators
76 template <typename T>
77 Set<T> const &  Set<T>::operator = (Set<T> const & target)
78 {
79     if (&target == this)
80         return *this;
81
82     if (_tab)
83         delete [] _tab;
84     _tab = new T[target.count()];
85     _size = target.count();
86
87     for (size_t i = 0; i < target.count(); i++)
88         _tab[i] = target[i];
89
90     return *this;
91 }
92
93 template <typename T>
94 T &             Set<T>::operator [] (size_t const i) const
95 {
96     if (i >= _size)
97         throw IndexOutOfRange();
98
99     return _tab[i];
100 }
101
102 /*
103 ** Arithmetic operators
104 */
105
```

```cpp
template <typename T>
Set<T>          operator + (Set<T> const & target1, Set<T> const & target2)
{
    Set<T>  result(target1.count() + target2.count(),
        target1.getDefaultVal());

    for (size_t i = 0; i < target1.count(); i++)
        result[i] = target1[i];

    for (size_t i = 0; i < target2.count(); i++)
        result[i + target1.count()] = target2[i];

    return result;
}

template <typename T>
Set<T>          operator * (Set<T> const & target1, Set<T> const & target2)
{
    size_t  len;

    len = 0;
    for (size_t i = 0; i < target1.count(); i++)
        if (target1[i] <in> target2)
            len++;

    Set<T>  result(len, target1.getDefaultVal());
    size_t  k = 0;

    for (size_t i = 0; i < target1.count(); i++)
        if (target1[i] <in> target2) {
            result[k] = target1[i];
            k++;
        }

    return result;
}

template <typename T>
Set<T>          operator - (Set<T> const & target1, Set<T> const & target2)
{
    size_t  len;

    len = target1.count();
    for (size_t i = 0; i < target1.count(); i++)
        if (target1[i] <in> target2)
            len--;

    Set<T>  result(len, target1.getDefaultVal());
    size_t  k = 0;

    for (size_t i = 0; i < target1.count(); i++)
        if (!(target1[i] <in> target2)) {
            result[k] = target1[i];
            k++;
        }

    return result;
}

template <typename T>
Set<T> &        Set<T>::operator += (Set<T> const & target)
{
    T *     newTab;

    newTab = new T [_size + target.count()];
    for (size_t i = 0; i < _size; i++)
        newTab[i] = _tab[i];

    for (size_t i = 0; i < target.count(); i++)
        newTab[i + _size] = target[i];

    delete [] _tab;
    _tab = newTab;
    _size += target.count();

    return *this;
}

/*
** Boolean operators
*/

template <typename T>
bool            Set<T>::operator == (Set<T> const & target)
{
    for (size_t i = 0; i < count(); i++)
        if (!(_tab[i] <in> target))
            return false;

    for (size_t i = 0; i < target.count(); i++)
        if (!(target[i] <in> *this))
            return false;

    return true;
}

/*
** IO operators
*/

template <typename T>
std::ostream & operator << (std::ostream & o, Set<T> const & target)
{
    o << "{";
    for (size_t i = 0; i < target.count(); i++) {
        o << target[i];
        if (i != target.count() - 1)
            o << ",␣";
    }
```

```
    o << "}";
    return o;
}

template <typename T>
std::istream & operator >> (std::istream & is, Set<T> & target)
{
    size_t  size;

    std::cout << "size: ";
    std::cin >> size;

    target = Set<T> (size, target.getDefaultVal());

    std::cout << "Set = {";
    for (size_t i = 0; i < size; i++) {
        std::cin >> target[i];
    }
    std::cout << "}" << std::endl;

    return is;
}

/*
** Custom '<in>' operator
** Ex:
** Set<int> a(3);
** a[0] = 3;
** 3 <in> a == true
*/

template <typename T>
struct              _opSetInProxy
{
    T const &   obj;

    //Constructor
    _opSetInProxy(T const & newObj): obj(newObj) {}
};

template <typename T>
_opSetInProxy<T>    operator < (T const & left, _opSetIn_ const & right)
{
    (void)right;
    return _opSetInProxy<T>(left);
}

template <typename T>
bool                operator > (_opSetInProxy<T> const & left,
                                Set<T> const & right)
{

    for (size_t i = 0; i < right.count(); i++)
        if (left.obj == right[i])
            return true;

    return false;
}
```

main.cpp

```cpp
#include "Set.tpp"
#include <iostream>

void	checkSum(void) {
	Set<int> a(5);

	a[0] = 1;
	a[1] = 2;

	Set<int> b(a);
	b[4] = 5;

	Set<int> c;

	std::cout << a << " + " << b << " + " << c << " = " << a + b << std::endl;

	Set<float> f1(3, 0);
	Set<float> f2;

	f1[2] = 5.1;
	f2 = f1;
	f2[0] = 42.5;

	std::cout << f1 << " + " << f2 << " = " << f1 + f2 << std::endl;

	Set<std::string> s1(5, "");
	Set<std::string> s2(2, "");

	s1[0] = "Adriana";
	s1[1] = "Emil";
	s1[2] = "Sergiu";
	s2[0] = "Alina";
	s2[1] = "Sergiu";

	std::cout << s1 << " + " << s2 << " = " << s1 + s2 << std::endl;

	Set<int> e1;

	std::cout << e1 << " + " << a << " = " << e1 + a << std::endl;
}

void	checkInOperator(void) {
	std::cout << std::endl << "Checking <in> operator" << std::endl;

	Set<std::string> s("", {"a", "b", "cd"});
	std::string needle = "a";

	std::cout << needle << " in " << s << " = " << (needle <in> s) << std::endl;
	needle = "x";
	std::cout << needle << " in " << s << " = " << (needle <in> s) << std::endl;

	Set<std::string> e("", {});

	std::cout << needle << " in " << e << " = " << (needle <in> s) << std::endl;
}

void	checkIntersection(void) {
	std::cout << std::endl << "Intersection check:" << std::endl;

	Set<int>	a(0, {1, 2, 42, -1, -1});
	Set<int>	b(0, {1, -2, 42});

	std::cout << a << " * " << b << " = " << a * b << std::endl;

	Set<std::string> s1("", {"S1", "S2", "S3"});
	Set<std::string> s2("", {"S1", "S2", "S", "S", "S"});

	std::cout << s1 << " * " << s2 << " = " << s1 * s2 << std::endl;

	Set<int>	e1;
	Set<int>	e2;

	std::cout << e1 << " * " << e2 << " = " << e1 * e2 << std::endl;
	std::cout << e1 << " * " << a << " = " << e1 * a << std::endl;
	std::cout << b << " * " << e1 << " = " << b * e1 << std::endl;
}

void	checkDifference(void) {
	std::cout << std::endl << "Difference check:" << std::endl;

	Set<int>	a(0, {1, 2, 3, 4});
	Set<int>	b(0, {1, 4, -1, -2});

	std::cout << a << " - " << b << " = " << a - b << std::endl;

	Set<int>	e(0, {});
	std::cout << e << " - " << b << " = " << e - b << std::endl;
	std::cout << a << " - " << e << " = " << a - e << std::endl;
}

void	checkAdd(void) {
	std::cout << std::endl << "Add check:" << std::endl;

	Set<int>	a(0, {1, 2, 3});
	Set<int>	b(0, {5});
	Set<int>	tmp(a);

	std::cout << tmp << " += " << b << " = " << (a += b) << std::endl;
	b = Set<int> (0, {});
	tmp = a;
	std::cout << tmp << " += " << b << " = " << (a += b) << std::endl;
}

void	checkEqual(void) {
```

```
    std::cout << std::endl << "Equality check:" << std::endl;

    Set<int>    a(0, {1, 2, 3});
    Set<int>    b(0, {5});

    std::cout << a << " == " << b << " = " << (a == b) << std::endl;

    b = Set<int> (0, {1, 2, 3, 3, 2, 1, 2});
    std::cout << a << " == " << b << " = " << (a == b) << std::endl;

    b = Set<int> (0, {1, 2, 3, 4});
    std::cout << a << " == " << b << " = " << (a == b) << std::endl;

    a = Set<int> (0, {1, 2, 3, 4, 5});
    std::cout << a << " == " << b << " = " << (a == b) << std::endl;
}

void    checkReading(void) {
    std::cout << std::endl << "Reading check:" << std::endl;

    Set<int> a(0, 0);
    std::cout << "Integer set:" << std::endl;
    std::cin >> a;
    std::cout << "a = " << a << std::endl;

    Set<std::string> s(0, "");
    std::cout << "String set:" << std::endl;
    std::cin >> s;
    std::cout << "s = " << s << std::endl;
}

void    checkOutOfRange(void) {
    std::cout << std::endl << "Out of range check:" << std::endl;
    Set<int> a(5);

    try
    {
        a[5] = 0;
    }
    catch (Set<int>::IndexOutOfRange const & e)
    {
        std::cout << e.what() << std::endl;
    }
}

int     main(void) {
    checkSum();
    checkInOperator();
    checkIntersection();
    checkDifference();
    checkAdd();
    checkEqual();
    checkOutOfRange();

    // checkReading();
    return 0;
}
```

Output

```
{1, 2, 0, 0, 0} + {1, 2, 0, 0, 5} + {} = {1, 2, 0, 0, 0, 1, 2, 0, 0, 5}
{0, 0, 5.1} + {42.5, 0, 5.1} = {0, 0, 5.1, 42.5, 0, 5.1}
{Adriana, Emil, Sergiu, , } + {Alina, Sergiu} = {Adriana, Emil, Sergiu, , , Alina, Sergiu}
{} + {1, 2, 0, 0, 0} = {1, 2, 0, 0, 0}

Checking <in> operator
a in {a, b, cd} = 1
x in {a, b, cd} = 0
x in {} = 0

Intersection check:
{1, 2, 42, -1, -1} * {1, -2, 42} = {1, 42}
{S1, S2, S3} * {S1, S2, S, S, S} = {S1, S2}
{} * {} = {}
{} * {1, 2, 42, -1, -1} = {}
{1, -2, 42} * {} = {}

Difference check:
{1, 2, 3, 4} - {1, 4, -1, -2} = {2, 3}
{} - {1, 4, -1, -2} = {}
{1, 2, 3, 4} - {} = {1, 2, 3, 4}

Add check:
{1, 2, 3} += {5} = {1, 2, 3, 5}
{1, 2, 3, 5} += {} = {1, 2, 3, 5}

Equality check:
{1, 2, 3} == {5} = 0
{1, 2, 3} == {1, 2, 3, 3, 2, 1, 2} = 1
{1, 2, 3} == {1, 2, 3, 4} = 0
{1, 2, 3, 4, 5} == {1, 2, 3, 4} = 0

Out of range check:
Index out of range
```

7 Analysis of the results and conclusions

- *Operators* are very useful in making the code more readable and more intuitive.
- Friend overload operators are useful when we want to make an action over a different class and the custom class. Ex: adding a float and a custom class.
- Friend overload operators are also useful when a++ or ++a is used.
- Using overload operators, we create some standarts: if a *Matrix* class is created, then we would expect that this class supports addition, multiplication, etc.