

# OOP laboratory\_03

Terman Emil FAF161

September 29, 2017

Conf. unv. prof: M. Kulev

L<sup>A</sup>T<sub>E</sub>X

# 1 Subject

Supraîncărcarea operatorilor

## 2 Objectives

- Studierea necesității 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 "++" și "+", ca metode ale clasei, iar operatorii "- -" și "-" ca funcții prietene. Operatorii trebuie să permită efectuarea operațiilor atât cu variabilele clasei date, cât și cu variabilele întregi de tip predefinit.

b) Să se creeze o clasă Set – mulțimea numerelor întregi, utilizând memoria dinamică. Să se definească operatorii de lucru cu mulțimile: "+" – uniunea, "\*" – intersecția, "-" scăderea, ca funcții prietene, iar "+=" – înserarea unui nou element în mulțime, "==" – comparare la egalitate, ș. a. ca metode ale clasei. Să se definească operatorii "<<" și ">>". Să se definească funcția de verificare a apartenenței unui element la o mulțime.

## 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

```
1 #ifndef MYINT_HPP
2 # define MYINT_HPP
3
4 # include <ostream>
5
6 class MyInt {
7 public:
8     //Getters
9     int      getVal(void) const;
10
11     //Constructors
12     MyInt(void);
13     MyInt(int value);
14     MyInt(MyInt const & target);
15
16     //Arithmetic operators
17     MyInt operator + (MyInt const & target) const;
18     MyInt operator + (int nb) const;
19
20     friend MyInt operator - (MyInt const & target1, MyInt const & target2);
21     friend MyInt operator - (MyInt const & target, int nb);
22     friend MyInt operator - (int nb, MyInt const & target);
23
24     MyInt & operator ++ (void);
25     MyInt operator -- (int);
26
27     MyInt const & operator = (MyInt const & target);
28
29     //Other operators
30     operator int() const;
31
32 private:
33     int _value;
34 };
35
36 std::ostream & operator << (std::ostream & o, MyInt const & target);
37
38 #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

```

1  #ifndef SET_TPP
2  # define SET_TPP
3
4  # include <ostream>
5  # include <exception>
6  # include <iostream>
7  # include <istream>
8  # include <initializer_list>
9
10 /*
11 ** Custom operators
12 */
13
14 const struct _opSetIn_ {} in = {};
15
16 template <typename T>
17 class Set {
18 public:
19
20     /*
21     ** Exceptions
22     */
23
24     class IndexOutOfRange : public std::exception {
25     public:
26         virtual char const * what() const throw() {
27             return "Index out of range";
28         }
29     };
30
31     /*
32     ** Getters
33     */
34
35     size_t      count(void) const;
36     T           getDefaultValue(void) const;
37
38     /*
39     ** Constructors & Destructors
40     */
41
42     Set<T>(void);
43     Set<T>(size_t const size);
44     Set<T>(size_t const size, T const defaultVal);
45     Set<T>(Set<T> const & target);
46     Set<T>(T const defaultVal, std::initializer_list<T> args);
47     ~Set<T>(void);
48
49     // Operators
50     Set<T> const & operator = (Set<T> const & target);
51     T & operator [] (size_t const i) const;
52
53     /*
54     ** Arithmetic operators
55     */
56
57     template <typename U>
58     friend Set<U> operator + (Set<U> const & target1, Set<U> const & target2);
59
60     template <typename U>
61     friend Set<U> operator * (Set<U> const & target1, Set<U> const & target2);
62
63     template <typename U>
64     friend Set<U> operator - (Set<U> const & target1, Set<U> const & target2);
65
66     Set<T> & operator += (Set<T> const & target);
67
68     /*
69     ** Boolean operators
70     */
71
72     bool operator == (Set<T> const & target);
73
74     /*
75     ** IO operators
76     */
77
78     template <typename U>
79     friend std::ostream & operator << (std::ostream & o, Set<U> const & target);
80
81     template <typename U>
82     friend std::istream & operator >> (std::istream & is, Set<U> & target);
83
84 private:
85     T * _tab;
86     size_t _size;
87     T const _defaultVal;
88 };

```

- `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

```
1 #include "MyInt.hpp"
2
3 //Getters
4 int MyInt::getVal(void) const {return _value;}
5
6 //Constructors
7 MyInt::MyInt(void) {
8     _value = 0;
9 }
10
11 MyInt::MyInt(int const value) {
12     _value = value;
13 }
14
15 MyInt::MyInt(MyInt const & target) {
16     _value = target.getVal();
17 }
18
19 /*
20 ** Operators
21 */
22
23 MyInt const & MyInt::operator = (MyInt const & target) {
24     _value = target.getVal();
25     return *this;
26 }
27
28 //Pre
29 MyInt MyInt::operator + (MyInt const & target) const {
30     return MyInt(getVal() + target.getVal());
31 }
32
33 MyInt MyInt::operator + (int nb) const {
34     return MyInt(getVal() + nb);
35 }
36
37 MyInt & MyInt::operator ++ (void) {
38     _value++;
39     return *this;
40 }
41
42 /*
43 ** Friendly
44 */
45
46 //With class ref on both sides
47 MyInt operator - (MyInt const & target1, MyInt const & target2) {
48     return MyInt(target1.getVal() - target2.getVal());
49 }
50
51 //With Class ref on the left
52 MyInt operator - (MyInt const & target, int const nb) {
53     return MyInt(target.getVal() - nb);
54 }
55
56 //With Class ref on the right
57 MyInt operator - (int nb, MyInt const & target) {
58     return MyInt(nb - target.getVal());
59 }
60
61 MyInt MyInt::operator -- (int) {
62     MyInt result(*this);
63
64     _value--;
65     return result;
66 }
67
68 //Other operators
69 MyInt::operator int() const {return getVal();}
70
71 std::ostream & operator << (std::ostream & o, MyInt const & target) {
72     o << target.getVal();
73     return o;
74 }
```

## main.cpp

```
1 #include "MyInt.hpp"
2
3 #include <iostream>
4
5 int main(void) {
6     MyInt a(3);
7     MyInt b(5);
8
9     std::cout << "a+b=" << a + b << std::endl;
10    std::cout << "a+5=" << a + 5 << std::endl;
11    std::cout << "3+b=" << 3 + b << std::endl;
12
13    std::cout << "a-b=" << a - b << std::endl;
14    std::cout << "a-5=" << a - 5 << std::endl;
15    std::cout << "3-b=" << 3 - b << std::endl;
16
17    std::cout << "-----" << std::endl;
18    std::cout << "a=" << a << std::endl;
19    std::cout << "++a=" << ++a << std::endl;
20    std::cout << "a=" << a << std::endl;
21
22    std::cout << "-----" << std::endl;
23    std::cout << "a--=" << a-- << std::endl;
24    std::cout << "a=" << a << std::endl;
25    std::cout << "-----" << std::endl;
26
27    int intA = a;
28    std::cout << "int aInt=" << intA << std::endl;
29
30    return 0;
31 }
```

## Output

```
1 a + b = 8
2 a + 5 = 8
3 3 + b = 8
4 a - b = -2
5 a - 5 = -2
6 3 - b = -2
7 -----
8 ++a: 4
9 a = 4
10 -----
11 a--: 4
12 a = 3
13 -----
14 int aInt = a; aInt = 3
15
```

## 6.2 Ex b

```

1  /*
2  ** Getters
3  */
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

```



```

106 template <typename T>
107 Set<T> operator + (Set<T> const & target1, Set<T> const & target2)
108 {
109     Set<T> result(target1.count() + target2.count(),
110                 target1.getDefaultVal());
111
112     for (size_t i = 0; i < target1.count(); i++)
113         result[i] = target1[i];
114
115     for (size_t i = 0; i < target2.count(); i++)
116         result[i + target1.count()] = target2[i];
117
118     return result;
119 }
120
121 template <typename T>
122 Set<T> operator * (Set<T> const & target1, Set<T> const & target2)
123 {
124     size_t len;
125
126     len = 0;
127     for (size_t i = 0; i < target1.count(); i++)
128         if (target1[i] <in> target2)
129             len++;
130
131     Set<T> result(len, target1.getDefaultVal());
132     size_t k = 0;
133
134     for (size_t i = 0; i < target1.count(); i++)
135         if (target1[i] <in> target2) {
136             result[k] = target1[i];
137             k++;
138         }
139
140     return result;
141 }
142
143 template <typename T>
144 Set<T> operator - (Set<T> const & target1, Set<T> const & target2)
145 {
146     size_t len;
147
148     len = target1.count();
149     for (size_t i = 0; i < target1.count(); i++)
150         if (target1[i] <in> target2)
151             len--;
152
153     Set<T> result(len, target1.getDefaultVal());
154     size_t k = 0;
155
156     for (size_t i = 0; i < target1.count(); i++)
157         if (!(target1[i] <in> target2)) {
158             result[k] = target1[i];
159             k++;
160         }
161
162     return result;
163 }
164
165 template <typename T>
166 Set<T> & Set<T>::operator += (Set<T> const & target)
167 {
168     T * newTab;
169
170     newTab = new T [_size + target.count()];
171     for (size_t i = 0; i < _size; i++)
172         newTab[i] = _tab[i];
173
174     for (size_t i = 0; i < target.count(); i++)
175         newTab[i + _size] = target[i];
176
177     delete [] _tab;
178     _tab = newTab;
179     _size += target.count();
180
181     return *this;
182 }
183
184 /*
185 ** Boolean operators
186 */
187
188 template <typename T>
189 bool Set<T>::operator == (Set<T> const & target)
190 {
191     for (size_t i = 0; i < count(); i++)
192         if (!(_tab[i] <in> target))
193             return false;
194
195     for (size_t i = 0; i < target.count(); i++)
196         if (!(target[i] <in> *this))
197             return false;
198
199     return true;
200 }
201
202 /*
203 ** IO operators
204 */
205
206 template <typename T>
207 std::ostream & operator << (std::ostream & o, Set<T> const & target)
208 {
209     o << "{";
210     for (size_t i = 0; i < target.count(); i++) {
211         o << target[i];
212         if (i != target.count() - 1)
213             o << ", ";
214     }

```

```

215     o << "}";
216     return o;
217 }
218
219 template <typename T>
220 std::istream & operator >> (std::istream & is, Set<T> & target)
221 {
222     size_t size;
223
224     std::cout << "size: ";
225     std::cin >> size;
226
227     target = Set<T> (size, target.getDefaultVal());
228
229     std::cout << "Set={";
230     for (size_t i = 0; i < size; i++) {
231         std::cin >> target[i];
232     }
233     std::cout << "}";
234     return is;
235 }
236
237 /**
238  * Custom '<in>' operator
239  * Ex:
240  * Set<int> a(3);
241  * a[0] = 3;
242  * 3 <in> a == true
243  */
244
245 template <typename T>
246 struct _opSetInProxy
247 {
248     T const & obj;
249
250     //Constructor
251     _opSetInProxy(T const & newObj): obj(newObj) {}
252 };
253
254 template <typename T>
255 _opSetInProxy<T> operator < (T const & left, _opSetIn_ const & right)
256 {
257     (void)right;
258     return _opSetInProxy<T>(left);
259 }
260
261 template <typename T>
262 bool operator > (_opSetInProxy<T> const & left,
263                 Set<T> const & right)
264 {
265     for (size_t i = 0; i < right.count(); i++)
266         if (left.obj == right[i])
267             return true;
268
269     return false;
270 }
271
272 }

```

## main.cpp

```
1 #include "Set.hpp"
2 #include <iostream>
3
4 void checkSum(void) {
5     Set<int> a(5);
6
7     a[0] = 1;
8     a[1] = 2;
9
10    Set<int> b(a);
11    b[4] = 5;
12
13    Set<int> c;
14
15    std::cout << a << " + " << b << " = " << c << " = " << a + b << std::endl;
16
17    Set<float> f1(3, 0);
18    Set<float> f2;
19
20    f1[2] = 5.1;
21    f2 = f1;
22    f2[0] = 42.5;
23
24    std::cout << f1 << " + " << f2 << " = " << f1 + f2 << std::endl;
25
26    Set<std::string> s1(5, "");
27    Set<std::string> s2(2, "");
28
29    s1[0] = "Adriana";
30    s1[1] = "Emil";
31    s1[2] = "Sergiu";
32    s2[0] = "Alina";
33    s2[1] = "Sergiu";
34
35    std::cout << s1 << " + " << s2 << " = " << s1 + s2 << std::endl;
36
37    Set<int> e1;
38
39    std::cout << e1 << " + " << a << " = " << e1 + a << std::endl;
40 }
41
42 void checkInOperator(void) {
43     std::cout << std::endl << "Checking in operator" << std::endl;
44
45     Set<std::string> s("a", {"a", "b", "cd"});
46     std::string needle = "a";
47
48     std::cout << needle << " in " << s << " = " << (needle in s) << std::endl;
49     needle = "x";
50     std::cout << needle << " in " << s << " = " << (needle in s) << std::endl;
51
52     Set<std::string> e("a", {});
53
54     std::cout << needle << " in " << e << " = " << (needle in e) << std::endl;
55 }
56
57 void checkIntersection(void) {
58     std::cout << std::endl << "Intersection check:" << std::endl;
59
60     Set<int> a(0, {1, 2, 42, -1, -1});
61     Set<int> b(0, {1, -2, 42});
62
63     std::cout << a << " * " << b << " = " << a * b << std::endl;
64
65     Set<std::string> s1("a", {"S1", "S2", "S3"});
66     Set<std::string> s2("a", {"S1", "S2", "S", "S", "S"});
67
68     std::cout << s1 << " * " << s2 << " = " << s1 * s2 << std::endl;
69
70     Set<int> e1;
71     Set<int> e2;
72
73     std::cout << e1 << " * " << e2 << " = " << e1 * e2 << std::endl;
74     std::cout << e1 << " * " << a << " = " << e1 * a << std::endl;
75     std::cout << b << " * " << e1 << " = " << b * e1 << std::endl;
76 }
77
78 void checkDifference(void) {
79     std::cout << std::endl << "Difference check:" << std::endl;
80
81     Set<int> a(0, {1, 2, 3, 4});
82     Set<int> b(0, {1, 4, -1, -2});
83
84     std::cout << a << " - " << b << " = " << a - b << std::endl;
85
86     Set<int> e(0, {});
87     std::cout << e << " - " << b << " = " << e - b << std::endl;
88     std::cout << a << " - " << e << " = " << a - e << std::endl;
89 }
90
91 void checkAdd(void) {
92     std::cout << std::endl << "Add check:" << std::endl;
93
94     Set<int> a(0, {1, 2, 3});
95     Set<int> b(0, {5});
96     Set<int> tmp(a);
97
98     std::cout << tmp << " + " << b << " = " << (a += b) << std::endl;
99     b = Set<int> (0, {});
100    tmp = a;
101    std::cout << tmp << " + " << b << " = " << (a += b) << std::endl;
102 }
103
104 void checkEqual(void) {
```

```

105     std::cout << std::endl << "Equality check:" << std::endl;
106
107     Set<int> a(0, {1, 2, 3});
108     Set<int> b(0, {5});
109
110     std::cout << a << "==" << b << "==" << (a == b) << std::endl;
111
112     b = Set<int>(0, {1, 2, 3, 3, 2, 1, 2});
113     std::cout << a << "==" << b << "==" << (a == b) << std::endl;
114
115     b = Set<int>(0, {1, 2, 3, 4});
116     std::cout << a << "==" << b << "==" << (a == b) << std::endl;
117
118     a = Set<int>(0, {1, 2, 3, 4, 5});
119     std::cout << a << "==" << b << "==" << (a == b) << std::endl;
120 }
121
122 void checkReading(void) {
123     std::cout << std::endl << "Reading check:" << std::endl;
124
125     Set<int> a(0, 0);
126     std::cout << "Integer set:" << std::endl;
127     std::cin >> a;
128     std::cout << "a=" << a << std::endl;
129
130     Set<std::string> s(0, "");
131     std::cout << "String set:" << std::endl;
132     std::cin >> s;
133     std::cout << "s=" << s << std::endl;
134 }
135
136 void checkOutOfRange(void) {
137     std::cout << std::endl << "Out of range check:" << std::endl;
138     Set<int> a(5);
139
140     try
141     {
142         a[5] = 0;
143     }
144     catch (Set<int>::IndexOutOfRange const & e)
145     {
146         std::cout << e.what() << std::endl;
147     }
148 }
149
150 int main(void) {
151     checkSum();
152     checkInOperator();
153     checkIntersection();
154     checkDifference();
155     checkAdd();
156     checkEqual();
157     checkOutOfRange();
158
159     // checkReading();
160     return 0;
161 }

```

## Output

```

1 {1, 2, 0, 0, 0} + {1, 2, 0, 0, 5} + {} = {1, 2, 0, 0, 0, 1, 2, 0, 0, 5}
2 {0, 0, 5.1} + {42.5, 0, 5.1} = {0, 0, 5.1, 42.5, 0, 5.1}
3 {Adriana, Emil, Sergiu, , } + {Alina, Sergiu} = {Adriana, Emil, Sergiu, , , Alina, Sergiu}
4 {} + {1, 2, 0, 0, 0} = {1, 2, 0, 0, 0}
5
6 Checking <in> operator
7 a in {a, b, cd} = 1
8 x in {a, b, cd} = 0
9 x in {} = 0
10
11 Intersection check:
12 {1, 2, 42, -1, -1} * {1, -2, 42} = {1, 42}
13 {S1, S2, S3} * {S1, S2, S, S, S} = {S1, S2}
14 {} * {} = {}
15 {} * {1, 2, 42, -1, -1} = {}
16 {1, -2, 42} * {} = {}
17
18 Difference check:
19 {1, 2, 3, 4} - {1, 4, -1, -2} = {2, 3}
20 {} - {1, 4, -1, -2} = {}
21 {1, 2, 3, 4} - {} = {1, 2, 3, 4}
22
23 Add check:
24 {1, 2, 3} += {5} = {1, 2, 3, 5}
25 {1, 2, 3, 5} += {} = {1, 2, 3, 5}
26
27 Equality check:
28 {1, 2, 3} == {5} = 0
29 {1, 2, 3} == {1, 2, 3, 3, 2, 1, 2} = 1
30 {1, 2, 3} == {1, 2, 3, 4} = 0
31 {1, 2, 3, 4, 5} == {1, 2, 3, 4} = 0
32
33 Out of range check:
34 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.