

OOP laboratory_02

Terman Emil FAF161

September 25, 2017

PHD prof: M. Kulev

L^AT_EX

1 Subject

Constructor - initialization function

2 Objectives

- Studiarea principiilor de definire și utilizare a constructorilor
- Studiarea principiilor de definire și utilizare a destructorilor
- Studiarea tipurilor de constructori

3 Task

a) Să se creeze clasa Date – dată cu câmpurile: zi(1-28..31), lună(1-12), an (numere întregi). Să se definească constructorii; funcțiile membru de setare a zilei, lunii și anului; funcțiile membru de returnare a zilei, lunii, anului; funcțiile de afișare: afișare tip “6 iunie 2004” și afișare tip “6.06.2004”. Funcțiile de setare a câmpurilor clasei trebuie să verifice corectitudinea parametrilor primiți.

b) Să se creeze clasa Matrix – matrice. Clasa conține pointer spre int, numărul de rînduri și de coloane și o variabilă – codul erorii. Să se definească constructorul fără parametri (constructorul implicit), constructorul cu un parametru – matrice pătrată și constructorul cu doi parametri – matrice dreptunghiulară ș. a. Să se definească funcțiile membru de acces: returnarea și setarea valorii elementului (i, j). Să se definească funcțiile de adunare și scădere a două matrice; înmulțirea unei matrice cu alta; înmulțirea unei matrice cu un număr. Să se testeze funcționarea clasei. În caz de insuficiență de memorie, necorespondență a dimensiunilor matricelor, depășire a limitei memoriei utilizate să se stabilească codul erorii.

4 Main notions of theory and used methods

A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

5 Data analysis

5.1 Ex a

```
1 #ifndef DATE_HPP
2 # define DATE_HPP
3
4 # include <iostream>
5 # include <string>
6 # include <stdexcept>
7
8 class Date
9 {
10 public:
11
12     //Exceptions
13     class InvalidDate : public std::exception {
14     public:
15         virtual const char * what() const throw();
16     };
17
18     enum EMonth {
19         jan = 1, feb, mar,
20         apr, may, jun,
21         jul, aug, sept,
22         oct, nov, dec
23     };
24
25     //Getters
26     int getDay(void) const;
27     int getMonth(void) const;
28     int getYear(void) const;
29
30     //Setters
31     void setDay(int day);
32     void setMonth(int month);
33     void setYear(int year);
34
35     //Constr & destr
36     Date(int day, int month, int year);
37     Date(Date const & target);
38     ~Date(void);
39
40     bool isLeapYear(void) const;
41     std::string toStrNamedMonth(void) const;
42     std::string toStr(void) const;
43
44     //Operators
45     Date & operator = (Date const & target);
46
47 private:
48     static const std::string _monthNames[12];
49
50     int _day;
51     int _month;
52     int _year;
53
54     int _monthMaxDays(void) const;
55 };
56
57 std::ostream & operator << (std::ostream & o, Date const & target);
58
59 #endif
```

- `Date(int day, int month, int year);`

A constructor which initializes the day, month and the year fields with the given values. The *InvalidDate* exception is thrown in case of invalid parameters.

- `Date(Date const & target);`

A copy constructor, taking in a constant reference to a *Date* instance.

- 1. `int getDay(void) const;`
2. `int getMonth(void) const;`
3. `int getYear(void) const;`

Constant getter functions.

- 1. `int setDay(int day);`
2. `int setMonth(int month);`

3. `int setYear(int year);`

Setters which will throw the *InvalidDate* exception in case the given values are impossible.

- `Date & operator = (Date const & target);`

The overload assign operator, which assigns the day, month and the year to *this* Date instance.

- `std::string toStringNamedMonth(void) const;`

Returns a new string, of the date formatted in the following way:
06 jun 2017

- `std::string toString(void) const;`

Returns a new string, of the date formatted in the following way:
06.06.2017

5.2 Ex b

```

1  #ifndef MATRIX_HPP
2  # define MATRIX_HPP
3
4  # include <string>
5  # include <iostream>
6  # include <errno.h>
7
8  class Matrix
9  {
10 public:
11     enum          EMatrixErrno {
12         boundsErr = 1,
13         invalidSize,
14         enomem = ENOMEM
15     };
16
17     mutable int    mErrno;
18
19     int            getLines(void) const;
20     int            getCols(void) const;
21
22     Matrix(void);
23     Matrix(int lines, int cols);
24     Matrix(Matrix const & target);
25     ~Matrix(void);
26
27     //Utils
28     void           assignAll(int value);
29
30     //Operators
31     Matrix &       operator = (Matrix const & target);
32
33     int const *    operator [] (int i) const;
34     int *          operator [] (int i);
35
36     Matrix         operator + (Matrix const & target) const;
37     Matrix         operator - (Matrix const & target) const;
38     Matrix         operator * (Matrix const & target) const;
39     Matrix         operator * (int nb) const;
40
41 private:
42     int            **_tab;
43     int            _lines;
44     int            _cols;
45
46     //Utils
47     void           _delTab(void);
48     int            _newTab(int lines, int cols);
49 };
50
51 std::ostream &    operator << (std::ostream & o, Matrix const & target);
52
53 #endif

```

- `Matrix(void);`
The default constructor, which makes an empty matrix.
- `Matrix(int lines , int cols);`
A constructor which makes a matrix of the size *lines* x *cols*, initializing every element with 0.
- `Matrix(Matrix const & target);`
A copy constructors which takes in a constant reference to a matrix.
- `~Matrix(void);`
The destructor: it deletes the allocated memory of the matrix.
- `int const * operator [] (int i) const;`
A getter operator returning a constant pointer to the *i* line of the matrix.
- `int * operator [] (int i);`
The same as previous, but this operator simply returns a normal pointer to the line.

- 1. Matrix `operator +` (Matrix `const` & target) `const`;
- 2. Matrix `operator -` (Matrix `const` & target) `const`;
- 3. Matrix `operator *` (Matrix `const` & target) `const`;

Arithmetic constant operators, which take in a constant reference to a matrix with which the operation is to be executed. A new fresh matrix with the result is returned.

- Matrix `operator *` (`int` nb) `const`;

Overload operator for multiplication which returns a new matrix with the elements of this matrix multiplied with the given number.

- `std::ostream & operator <<` (`std::ostream & o`, Matrix `const` & target);

An overload operator `<<` for matrix. It takes in an *ostream* reference and a constant reference to a Matrix. It puts in the stream the entire content of the given matrix.

6 The actual code

6.1 Ex a

```
1 #include "Date.hpp"
2 #include <sstream>
3
4 std::string const Date::_monthNames[12] = {
5     "jan", "feb", "mar", "apr", "may", "jun", "aug", "sept", "oct", "nov", "dec"
6 };
7
8 /*
9  ** Getters
10 */
11
12 int Date::getDay(void) const {return _day;}
13 int Date::getMonth(void) const {return _month;}
14 int Date::getYear(void) const {return _year;}
15
16 /*
17  ** Setters
18 */
19
20 void Date::setDay(int const day) {
21     if (day > _monthMaxDays())
22         throw InvalidDate();
23     _day = day;
24 }
25
26 void Date::setMonth(int const month) {
27     if (month < 1 || month > 12)
28         throw InvalidDate();
29     _month = month;
30 }
31
32 void Date::setYear(int const year) {
33     if (year < 0)
34         throw InvalidDate();
35     _year = year;
36 }
37
38 /*
39  ** Constructors & destructors
40 */
41
42 Date::Date(int const day, int const month, int const year) {
43     setYear(year);
44     setMonth(month);
45     setDay(day);
46 }
47
48 Date::Date(Date const & target) {*this = target;}
49
50 Date::~Date(void) {}
51
52 /*
53  ** Other functions
54 */
55
56 bool Date::isLeapYear(void) const {
57     return (_year % 100 == 0) ? (_year % 400 == 0) : (_year % 4 == 0);
58 }
59
60 std::string Date::toStrNamedMonth(void) const {
61     std::ostringstream os;
62     os << getDay() << " " << _monthNames[getMonth() - 1] << " " << getYear();
63     return os.str();
64 }
65
66 std::string Date::toStr(void) const {
67     std::ostringstream os;
68     os << getDay() << "." << getMonth() << "." << getYear();
69     return os.str();
70 }
71
72 //Returns the maximum number of days this month can have in this year.
73 int Date::_monthMaxDays(void) const {
74     if (_month == apr || _month == jun || _month == sept || _month == nov)
75         return 30;
76     else if (_month != feb)
77         return 31;
78     else
79         return isLeapYear() ? 29 : 28;
80 }
81
82 /*
83  ** Operators
84 */
85
86 Date & Date::operator = (Date const & target) {
87     setYear(target.getYear());
88     setMonth(target.getMonth());
89     setDay(target.getDay());
90     return *this;
91 }
92
93 std::ostream & operator << (std::ostream & o, Date const & target) {
94     o << target.toStr();
95 }
```

```

101     return o;
102 }
103
104 /*
105 ** Exceptions
106 */
107
108 char const * Date::InvalidDate::what() const throw () {
109     return "Invalid day, month or year";
110 }

```

main.cpp

```

1  #include "Date.hpp"
2
3  int main(void)
4  {
5      Date date1(10, Date::mar, 1997);
6
7      std::cout << date1 << std::endl;
8      std::cout << date1.toStrNamedMonth() << std::endl;
9
10     Date date2(date1);
11
12     try
13     {
14         date2.setDay(100);
15     }
16     catch (std::exception const & e)
17     {
18         std::cout << e.what() << std::endl;
19     }
20
21     date2.setMonth(Date::feb);
22
23     try
24     {
25         date2.setDay(29);
26     }
27     catch (std::exception const & e)
28     {
29         std::cout << e.what() << std::endl;
30     }
31
32     date2.setYear(2000);
33     date2.setDay(29);
34
35     std::cout << date2 << std::endl;
36
37     try
38     {
39         Date date3(29, Date::feb, 1998);
40     }
41     catch (std::exception const & e)
42     {
43         std::cout << e.what() << std::endl;
44     }
45
46     return 0;
47 }

```

Output

```

1  10.3.1997
2  10 mar 1997
3  Invalid day, month or year
4  Invalid day, month or year
5  29.2.2000
6  Invalid day, month or year
7

```


6.2 Ex b

```

1  #include "Matrix.hpp"
2
3  /*
4  ** Getters
5  */
6
7  int Matrix::getLines(void) const {return (_tab == NULL) ? 0 : _lines;}
8  int Matrix::getCols(void) const {return (_tab == NULL) ? 0 : _cols;}
9
10 /*
11 ** Constructors & destructors
12 */
13
14 Matrix::Matrix(void) {
15     _tab = NULL;
16     _lines = 0;
17     _cols = 0;
18     mErrno = 0;
19 }
20
21 Matrix::Matrix(int lines, int cols) {
22     _tab = NULL;
23     mErrno = 0;
24     _newTab(lines, cols);
25 }
26
27 Matrix::Matrix(Matrix const & target) {
28     _tab = NULL;
29     mErrno = 0;
30     *this = target;
31 }
32
33 Matrix::~Matrix(void) {_delTab();}
34
35 /*
36 ** Utilities
37 */
38
39 void Matrix::assignAll(int const value) {
40     if (_tab == NULL)
41         return;
42
43     for (int i = 0; i < _lines; i++)
44         for (int j = 0; j < _cols; j++)
45             _tab[i][j] = value;
46 }
47
48 void Matrix::_delTab(void) {
49     if (_tab != NULL) {
50         for (int i = 0; i < _lines; i++) {
51             if (_tab[i])
52                 delete [] _tab[i];
53         }
54         delete [] _tab;
55         _tab = NULL;
56     }
57
58     _cols = 0;
59     _lines = 0;
60 }
61
62 int Matrix::_newTab(int lines, int cols) {
63     if (_tab != NULL)
64         _delTab();
65
66     _tab = new int * [_lines];
67     if (_tab == NULL) {
68         mErrno = enomem;
69         return -1;
70     }
71
72     for (int i = 0; i < lines; i++) {
73         _tab[i] = new int [_cols];
74
75         if (_tab[i] == NULL) {
76             mErrno = enomem;
77             _lines = i;
78             _delTab();
79             _lines = 0;
80             return -1;
81         }
82     }
83
84     _lines = lines;
85     _cols = cols;
86     assignAll(0);
87     return 0;
88 }
89
90 /*
91 ** Operators
92 */
93
94 Matrix & Matrix::operator = (Matrix const & target) {
95     if (_newTab(target.getLines(), target.getCols()) == -1)
96         return *this;
97
98     for (int i = 0; i < _lines; i++)
99         for (int j = 0; j < _cols; j++)
100             _tab[i][j] = target[i][j];
101
102     return *this;
103 }
104
105

```

```

106 int *      Matrix::operator [] (int const i) {return _tab[i];}
107 int const * Matrix::operator [] (int const i) const {return _tab[i];}
108
109
110 /*
111 ** Arithmetic operators
112 */
113
114 Matrix Matrix::operator + (Matrix const & target) const {
115     if (target.getLines() != _lines || target.getCols() != _cols) {
116         mErrno = invalidSize;
117         return Matrix();
118     }
119     Matrix result(*this);
120     if (result.mErrno != 0)
121         return result;
122     for (int i = 0; i < _lines; i++)
123         for (int j = 0; j < _cols; j++)
124             result[i][j] = _tab[i][j] + target[i][j];
125     return result;
126 }
127
128 Matrix Matrix::operator - (Matrix const & target) const {
129     if (target.getLines() != _lines || target.getCols() != _cols) {
130         mErrno = invalidSize;
131         return Matrix();
132     }
133     Matrix result(*this);
134     if (result.mErrno != 0)
135         return result;
136     for (int i = 0; i < _lines; i++)
137         for (int j = 0; j < _cols; j++)
138             result[i][j] = _tab[i][j] - target[i][j];
139     return result;
140 }
141
142 Matrix Matrix::operator * (Matrix const & target) const {
143     if (target.getCols() != target.getLines()) {
144         mErrno = invalidSize;
145         return Matrix();
146     }
147     Matrix result(_lines, target.getCols());
148     if (result.mErrno != 0)
149         return result;
150     for (int i = 0; i < _lines; i++) {
151         for (int j = 0; j < target.getCols(); j++) {
152             sum = 0;
153             for (int k = 0; k < _cols; k++)
154                 sum += (_tab[i][k] * (target[k][j]));
155             result[i][j] = sum;
156         }
157     }
158     return result;
159 }
160
161 Matrix Matrix::operator * (int const nb) const {
162     if (_tab == NULL)
163         return Matrix();
164     Matrix result(_lines, _cols);
165     if (result.mErrno != 0)
166         return result;
167     for (int i = 0; i < _lines; i++)
168         for (int j = 0; j < _cols; j++)
169             result[i][j] = _tab[i][j] * nb;
170     return result;
171 }
172
173 /*
174 ** Printing operator
175 */
176
177 std::ostream & operator << (std::ostream & o, Matrix const & target) {
178     o << "[" << std::endl;
179     for (int i = 0; i < target.getLines(); i++) {
180         o << "\t[";
181         for (int j = 0; j < target.getCols(); j++) {
182             o << target[i][j];
183             if (j != target.getCols() - 1)
184                 o << ",\t";
185         }
186         o << "]" << std::endl;
187     }
188     o << "]" << std::endl;
189     return o;
190 }
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208

```

main.cpp

```
1 #include "Matrix.hpp"
2
3 void checkAdditionAndSubstr(void) {
4     std::cout << "Addition & subtraction" << std::endl;
5
6     Matrix matrix1(2, 2);
7     Matrix matrix2;
8
9     matrix1[0][0] = 1;
10    matrix1[1][1] = 2;
11
12    matrix2 = matrix1;
13    matrix2[0][1] = 42;
14
15    std::cout << "matrix1:\n" << matrix1;
16    std::cout << "matrix2:\n" << matrix2;
17
18    std::cout << "matrix1+matrix2:\n" << matrix1 + matrix2 << std::endl;
19
20    Matrix matrix3(5, 5);
21
22    std::cout << "matrix3:\n" << matrix3;
23    std::cout << "matrix1+matrix3:\n" << matrix1 + matrix3;
24    std::cout << "error:\n" << matrix1.mErrno << std::endl;
25 }
26
27 void checkMultiplication(void) {
28     std::cout << "Multiplication" << std::endl;
29
30     Matrix matrix1(2, 4);
31     Matrix matrix2(4, 4);
32
33     matrix1[0][0] = 1; matrix1[1][0] = 0;
34     matrix1[0][1] = 3; matrix1[1][1] = 5;
35     matrix1[0][2] = 2; matrix1[1][2] = -1;
36     matrix1[0][3] = 4; matrix1[1][3] = 6;
37
38     matrix2[0][0] = 1; matrix2[1][0] = -2;
39     matrix2[0][1] = -3; matrix2[1][1] = 0;
40     matrix2[0][2] = 2; matrix2[1][2] = 3;
41     matrix2[0][3] = -4; matrix2[1][3] = 4;
42
43     matrix2[2][0] = 5; matrix2[3][0] = 8;
44     matrix2[2][1] = -1; matrix2[3][1] = 9;
45     matrix2[2][2] = 6; matrix2[3][2] = 10;
46     matrix2[2][3] = 7; matrix2[3][3] = 11;
47
48     std::cout << "matrix1:\n" << matrix1;
49     std::cout << "matrix2:\n" << matrix2;
50
51     std::cout << "matrix1xmatrix2:\n" << matrix1 * matrix2;
52     std::cout << "matrix1x10:\n" << matrix1 * 10;
53 }
54
55 int main(void) {
56     checkAdditionAndSubstr();
57     checkMultiplication();
58     return 0;
59 }
```

Output

```
1 Addition & subtraction
2 matrix1:
3 {
4   {1, 0}
5   {0, 2}
6 }
7 matrix2:
8 {
9   {1, 42}
10  {0, 2}
11 }
12 matrix1 + matrix2:
13 {
14   {2, 42}
15   {0, 4}
16 }
17
18 matrix3:
19 {
20   {0, 0, 0, 0, 0}
21   {0, 0, 0, 0, 0}
22   {0, 0, 0, 0, 0}
23   {0, 0, 0, 0, 0}
24   {0, 0, 0, 0, 0}
25 }
26 matrix1 + matrix3:
27 {
28 }
29 error: 2
30
31 Multiplication
32 matrix1:
33 {
34   {1, 3, 2, 4}
35   {0, 5, -1, 6}
36 }
37 matrix2:
38 {
39   {1, -3, 2, -4}
40   {-2, 0, 3, 4}
41   {5, -1, 6, 7}
42   {8, 9, 10, 11}
43 }
44 matrix1 x matrix2:
45 {
46   {37, 31, 63, 66}
47   {33, 55, 69, 79}
48 }
49 matrix1 x 10:
50 {
51   {10, 30, 20, 40}
52   {0, 50, -10, 60}
53 }
54
```

7 Analysis of the results and conclusions

- *Classes* are much flexible than *Structures*.
- When we use Classes instead of structures, we pass less parameters to functions (which work with these classes), making the code more readable and more intuitive - leading to less errors and bugs.
- When classes are used over structures, we can benefit from *private* and *public* fields. These fields help us to encapsulate the data, making available only some specific class fields to the user. In this way, we can be sure that some data will always be the way we need it to be, without extra checks.
- In class, we can define some data that only this class will use, without polluting the global scope. For example, I used an *enum* to define the Month values for my *Date* class.
- We can make multiple functions with the same name, with different parameters - feature which is absent in *C* language.
- We can define Class constructors, which ease the initialization.