OOP laboratory_02

Terman Emil FAF161

September 25, 2017

Conf. unv. prof: M. Kulev

LaTeX

1 Subject

Constructor - initialization function

2 Objectives

- Studierea principiilor de definire şi utilizare a constructorilor
- Studierea principiilor de definire şi utilizare a destructorilor
- Studierea tipurilor de constructori

3 Task

a) Să se creeze clasa Date – dată cu cîmpurile: zi(1-28..31), lună(1-12), an (numere întregi). Să se definească constructorii; funcţiile membru de setare a zilei, lunii şi anului; funcţiile membru de returnare a zilei, lunii, anului; funcţiile de afişare: afişare tip "6 iunie 2004" şi afişare tip "6.06.2004". Funcţiile de setare a cîmpurilor clasei trebuie să verifice corectitudinea parametrilor primiţi.

b) Să se creeze clasa Matrix – matrice. Clasa conţine pointer spre int, numărul de rînduri şi de coloane şi o variabilă – codul erorii. Să se definească constructorul fără parametri (constructorul implicit), constructorul cu un parametru – matrice pătrată şi constructorul cu doi parametri – matrice dreptunghiulară ş. a. Să se definească funcţiile membru de acces: returnarea şi setarea valorii elementului (i, j). Să se definească funcţiile de adunare şi scădere a două matrice; înmulţirea unei matrice cu alta; înmulţirea unei matrice cu un număr. Să se testeze funcţionarea clasei. În caz de insuficienţă de memorie, necorespondenţă a dimensiunilor matricelor, depăşire 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

```
#ifndef DATE_HPP
# define DATE_HPP

# include <iostream>
# include <string>
# include <stdexcept>

class   Date
{
public:

    //Exceptions
    class   InvalidDate : public std::exception {
    public:
        virtual const char * what() const throw();
    };

    enum        EMonth {
                    jan = 1, feb, mar,
                    apr, may, jun,
                    jul, aug, sept,
                    oct, nov, dec
    };

    //Getters
    int         getDay(void) const;
    int         getMonth(void) const;
    int         getYear(void) const;

    //Setters
    void        setDay(int day);
    void        setMonth(int month);
    void        setYear(int year);

    //Constr & destr
    Date(int day, int month, int year);
    Date(Date const & target);
    ~Date(void);

    bool        isLeapYear(void) const;
    std::string toStrNamedMonth(void) const;
    std::string toStr(void) const;

    //Operators
    Date &      operator = (Date const & target);

private:
    static const std::string _monthNames[12];

    int         _day;
    int         _month;
    int         _year;

    int         _monthMaxDays(void) const;
};

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

#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 toStrNamedMonth(void) const;

  Returns a new string, of the date formated formated in the folowing way: *06 jun 2017*

- std :: string toStr(void) const;

  Returns a new string, of the date formated formated in the folowing way: *06.06.2017*

5.2 Ex b

```
#ifndef MATRIX_HPP
# define MATRIX_HPP

# include <string>
# include <iostream>
# include <errno.h>

class Matrix
{
public:
    enum            EMatrixErrno {
                        boundsErr = 1,
                        invalidSize,
                        enomem = ENOMEM
    };

    mutable int     mErrno;

    int             getLines(void) const;
    int             getCols(void) const;

    Matrix(void);
    Matrix(int lines, int cols);
    Matrix(Matrix const & target);
    ~Matrix(void);

    //Utils
    void            assignAll(int value);

    //Operators
    Matrix &        operator = (Matrix const & target);

    int const *     operator [] (int i) const;
    int *           operator [] (int i);

    Matrix          operator + (Matrix const & target) const;
    Matrix          operator - (Matrix const & target) const;
    Matrix          operator * (Matrix const & target) const;
    Matrix          operator * (int nb) const;

private:
    int             **_tab;
    int             _lines;
    int             _cols;

    //Utils
    void            _delTab(void);
    int             _newTab(int lines, int cols);
};

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

#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 *linex* 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

```cpp
#include "Date.hpp"
#include <sstream>

std::string const Date::_monthNames[12] = {
    "jan", "feb", "mar", "apr", "may", "jun", "aug", "sept", "oct", "nov", "dec"
};

/*
** Getters
*/

int     Date::getDay(void) const {return _day;}
int     Date::getMonth(void) const {return _month;}
int     Date::getYear(void) const {return _year;}

/*
** Setters
*/

void    Date::setDay(int const day) {
    if (day > _monthMaxDays())
        throw InvalidDate();

    _day = day;
}

void    Date::setMonth(int const month) {
    if (month < 1 || month > 12)
        throw InvalidDate();

    _month = month;
}

void    Date::setYear(int const year) {
    if (year < 0)
        throw InvalidDate();

    _year = year;
}

/*
** Constructors & destructors
*/

Date::Date(int const day, int const month, int const year) {
    setYear(year);
    setMonth(month);
    setDay(day);
}

Date::Date(Date const & target) {*this = target;}

Date::~Date(void) {}

/*
** Other functions
*/

bool    Date::isLeapYear(void) const {
    return (_year % 100 == 0) ? (_year % 400 == 0) : (_year % 4 == 0);
}

std::string Date::toStrNamedMonth(void) const {
    std::ostringstream  os;

    os << getDay() << "␣" << _monthNames[getMonth() - 1] << "␣" << getYear();
    return os.str();
}

std::string Date::toStr(void) const {
    std::ostringstream  os;

    os << getDay() << "." << getMonth() << "." << getYear();
    return os.str();
}

//Returns the maximum number of days this month can have in this year.
int     Date::_monthMaxDays(void) const {
    if (_month == apr || _month == jun || _month == sept || _month == nov)
        return 30;
    else if (_month != feb)
        return 31;
    else
        return isLeapYear() ? 29 : 28;
}

/*
** Operators
*/

Date    & Date::operator = (Date const & target) {
    setYear(target.getYear());
    setMonth(target.getMonth());
    setDay(target.getDay());

    return *this;
}

std::ostream & operator << (std::ostream & o, Date const & target) {
    o << target.toStr();
```

```
    return o;
}

/*
** Exceptions
*/

char const  * Date::InvalidDate::what() const throw () {
    return "Invalid day, month or year";
}
```

main.cpp

```
#include "Date.hpp"

int     main(void)
{
    Date    date1(10, Date::mar, 1997);

    std::cout << date1 << std::endl;
    std::cout << date1.toStrNamedMonth() << std::endl;

    Date    date2(date1);

    try
    {
        date2.setDay(100);
    }
    catch (std::exception const & e)
    {
        std::cout << e.what() << std::endl;
    }

    date2.setMonth(Date::feb);

    try
    {
        date2.setDay(29);
    }
    catch (std::exception const & e)
    {
        std::cout << e.what() << std::endl;
    }

    date2.setYear(2000);
    date2.setDay(29);

    std::cout << date2 << std::endl;

    try
    {
        Date    date3(29, Date::feb, 1998);
    }
    catch (std::exception const & e)
    {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

Output

```
10.3.1997
10 mar 1997
Invalid day, month or year
Invalid day, month or year
29.2.2000
Invalid day, month or year

```

6.2 Ex b

```cpp
#include "Matrix.hpp"

/*
** Getters
*/

int     Matrix::getLines(void) const {return (_tab == NULL) ? 0 : _lines;}
int     Matrix::getCols(void) const {return (_tab == NULL) ? 0 : _cols;}

/*
** Constructors & destructors
*/

Matrix::Matrix(void) {
    _tab = NULL;
    _lines = 0;
    _cols = 0;
    mErrno = 0;
}

Matrix::Matrix(int lines, int cols) {
    _tab = NULL;
    mErrno = 0;
    _newTab(lines, cols);
}

Matrix::Matrix(Matrix const & target) {
    _tab = NULL;
    mErrno = 0;
    *this = target;
}

Matrix::~Matrix(void) {_delTab();}

/*
** Utilities
*/

void    Matrix::assignAll(int const value) {
    if (_tab == NULL)
        return;

    for (int i = 0; i < _lines; i++)
        for (int j = 0; j < _cols; j++)
            _tab[i][j] = value;
}

void    Matrix::_delTab(void) {
    if (_tab != NULL) {
        for (int i = 0; i < _lines; i++) {
            if (_tab[i])
                delete [] _tab[i];
        }

        delete [] _tab;
        _tab = NULL;
    }

    _cols = 0;
    _lines = 0;
}

int     Matrix::_newTab(int lines, int cols) {
    if (_tab != NULL)
        _delTab();

    _tab = new int * [lines];
    if (_tab == NULL) {
        mErrno = enomem;
        return -1;
    }

    for (int i = 0; i < lines; i++) {
        _tab[i] = new int [cols];

        if (_tab[i] == NULL) {
            mErrno = enomem;
            _lines = i;
            _delTab();
            _lines = 0;
            return -1;
        }
    }

    _lines = lines;
    _cols = cols;
    assignAll(0);
    return 0;
}

/*
** Operators
*/

Matrix & Matrix::operator = (Matrix const & target) {
    if (_newTab(target.getLines(), target.getCols()) == -1)
        return *this;

    for (int i = 0; i < _lines; i++)
        for (int j = 0; j < _cols; j++)
            _tab[i][j] = target[i][j];

    return *this;
}

```

```
int *       Matrix::operator [] (int const i) {return _tab[i];}
int const * Matrix::operator [] (int const i) const {return _tab[i];}


/*
** Arithmetic operators
*/

Matrix Matrix::operator + (Matrix const & target) const {
    if (target.getLines() != _lines || target.getCols() != _cols) {
        mErrno = invalidSize;
        return Matrix();
    }

    Matrix  result(*this);

    if (result.mErrno != 0)
        return result;

    for (int i = 0; i < _lines; i++)
        for (int j = 0; j < _cols; j++)
            result[i][j] = _tab[i][j] + target[i][j];

    return result;
}

Matrix Matrix::operator - (Matrix const & target) const {
    if (target.getLines() != _lines || target.getCols() != _cols) {
        mErrno = invalidSize;
        return Matrix();
    }

    Matrix  result(*this);

    if (result.mErrno != 0)
        return result;

    for (int i = 0; i < _lines; i++)
        for (int j = 0; j < _cols; j++)
            result[i][j] = _tab[i][j] - target[i][j];

    return result;
}

Matrix Matrix::operator * (Matrix const & target) const {
    int sum;

    if (target.getCols() != target.getLines()) {
        mErrno = invalidSize;
        return Matrix();
    }

    Matrix  result(_lines, target.getCols());

    if (result.mErrno != 0)
        return result;

    for (int i = 0; i < _lines; i++) {
        for (int j = 0; j < target.getCols(); j++) {
            sum = 0;
            for (int k = 0; k < _cols; k++)
                sum += (_tab[i][k]) * (target[k][j]);
            result[i][j] = sum;
        }
    }

    return result;
}

Matrix Matrix::operator * (int const nb) const {
    if (_tab == NULL)
        return Matrix();

    Matrix  result(_lines, _cols);

    if (result.mErrno != 0)
        return result;

    for (int i = 0; i < _lines; i++)
        for (int j = 0; j < _cols; j++)
            result[i][j] = _tab[i][j] * nb;

    return result;
}

/*
** Printing operator
*/

std::ostream & operator << (std::ostream & o, Matrix const & target) {
    o << "{" << std::endl;
    for (int i = 0; i < target.getLines(); i++) {
        o << "\t{";
        for (int j = 0; j < target.getCols(); j++) {
            o << target[i][j];
            if (j != target.getCols() - 1)
                o << ",\t";
        }
        o << "}" << std::endl;
    }
    o << "}" << std::endl;
    return o;
}
```

main.cpp

```cpp
#include "Matrix.hpp"

void    checkAdditionAndSubstr(void) {
    std::cout << "Addition␣&␣substraction" << std::endl;

    Matrix  matrix1(2, 2);
    Matrix  matrix2;

    matrix1[0][0] = 1;
    matrix1[1][1] = 2;

    matrix2 = matrix1;
    matrix2[0][1] = 42;

    std::cout << "matrix1:\n" << matrix1;
    std::cout << "matrix2:\n" << matrix2;

    std::cout << "matrix1␣+␣matrix2:\n" << matrix1 + matrix2 << std::endl;

    Matrix matrix3(5, 5);

    std::cout << "matrix3:\n" << matrix3;
    std::cout << "matrix1␣+␣matrix3:\n" << matrix1 + matrix3;
    std::cout << "error:␣" << matrix1.mErrno << std::endl;
}

void    checkMultiplication(void) {
    std::cout << "\nMultiplication" << std::endl;

    Matrix  matrix1(2, 4);
    Matrix  matrix2(4, 4);

    matrix1[0][0] = 1;  matrix1[1][0] = 0;
    matrix1[0][1] = 3;  matrix1[1][1] = 5;
    matrix1[0][2] = 2;  matrix1[1][2] = -1;
    matrix1[0][3] = 4;  matrix1[1][3] = 6;

    matrix2[0][0] = 1;  matrix2[1][0] = -2;
    matrix2[0][1] = -3; matrix2[1][1] = 0;
    matrix2[0][2] = 2;  matrix2[1][2] = 3;
    matrix2[0][3] = -4; matrix2[1][3] = 4;

    matrix2[2][0] = 5;  matrix2[3][0] = 8;
    matrix2[2][1] = -1; matrix2[3][1] = 9;
    matrix2[2][2] = 6;  matrix2[3][2] = 10;
    matrix2[2][3] = 7;  matrix2[3][3] = 11;

    std::cout << "matrix1:\n" << matrix1;
    std::cout << "matrix2:\n" << matrix2;

    std::cout << "matrix1␣x␣matrix2:\n" << matrix1 * matrix2;
    std::cout << "matrix1␣x␣10:\n" << matrix1 * 10;
}

int     main(void) {
    checkAdditionAndSubstr();
    checkMultiplication();
    return 0;
}
```

Output

```
Addition & substraction
matrix1:
{
    {1, 0}
    {0, 2}
}
matrix2:
{
    {1, 42}
    {0, 2}
}
matrix1 + matrix2:
{
    {2, 42}
    {0, 4}
}

matrix3:
{
    {0, 0,  0,  0,  0}
    {0, 0,  0,  0,  0}
    {0, 0,  0,  0,  0}
    {0, 0,  0,  0,  0}
    {0, 0,  0,  0,  0}
}
matrix1 + matrix3:
{
}
error: 2

Multiplication
matrix1:
{
    {1, 3,  2,  4}
    {0, 5,  -1, 6}
}
matrix2:
{
    {1, -3, 2,  -4}
    {-2,    0,  3,  4}
    {5, -1, 6,  7}
    {8, 9,  10, 11}
}
matrix1 x matrix2:
{
    {37,    31, 63, 66}
    {33,    55, 69, 79}
}
matrix1 x 10:
{
    {10,    30, 20, 40}
    {0, 50, -10,    60}
}

```

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.