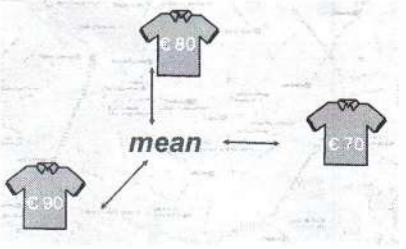
Deviation

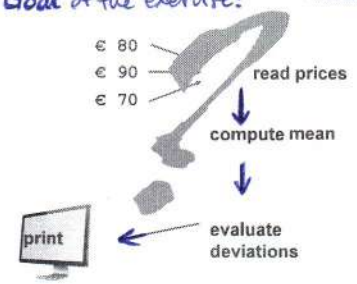



## VERSION 1

Read prices through standard inputs, compute mean and evaluate the deviations of each price from the mean.



**Goal** of the exercise:

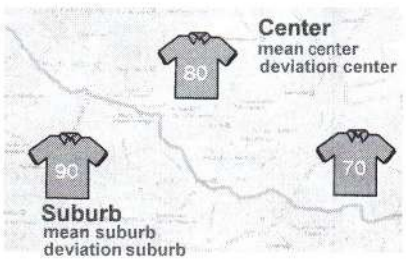



we set a max number of prices but we need to have a way out if we want to stop before the max number

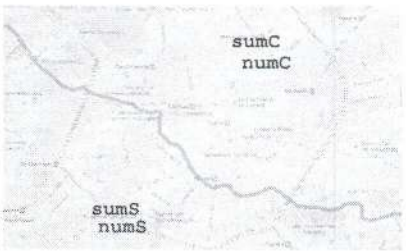
## VERSION 2

Split the zone into two subzones and do the same as version 1 for both the zones (center, suburb)

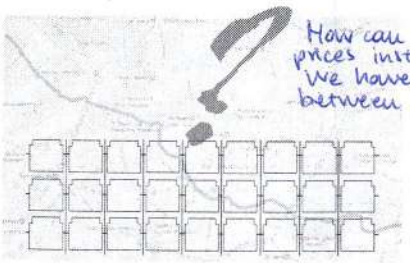






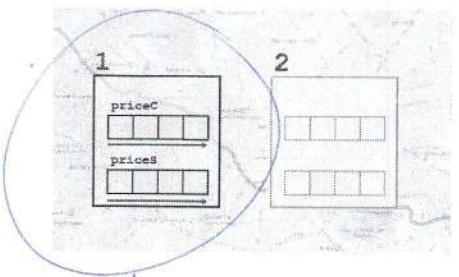





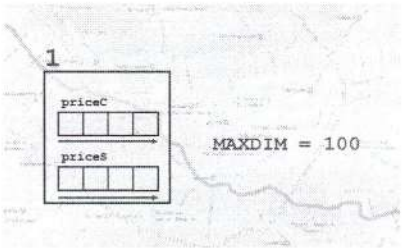



How can we store prices inserted by the users? We have to make difference between suburb and center

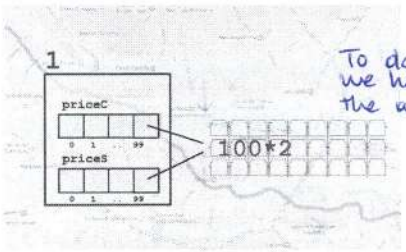





We can use two different arrays



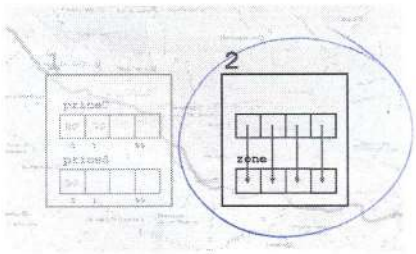



To do so (use two arrays) we have to store two times the array dimension of version 1

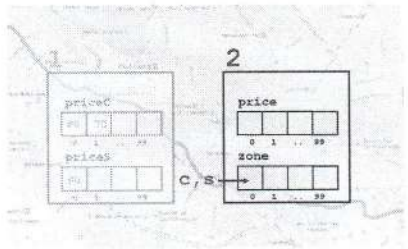


OR

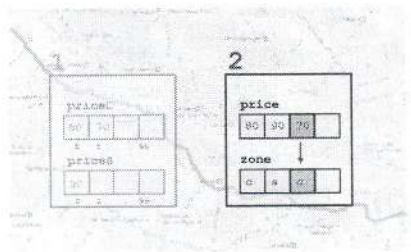



we can store prices in one array and we store informations about the zones in an other array (requires less memory because the second array is of characters (which are less heavy than integers/doubles)

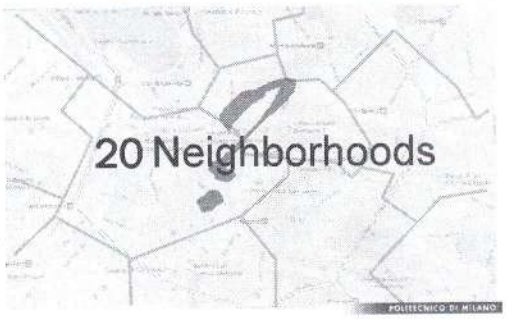









VERSION 4
Same thing as the version 2 but with many zones (not only two)

```cpp
// Price deviation evaluation v1

#include <iostream>
#include <iomanip>

using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main (void){
    const unsigned MAXDIM = 100;

    double prices[MAXDIM];
    unsigned num = 0;
    double sum = 0.;
    bool exit = false;

    while (! exit){
        double price;
        cout  << "Input price: ";
        cin >> price;

        if (! cin)// || num > MAXDIM)
            exit = true;
        else{
            prices[num++] = price;
            sum += price;

            if (num >= MAXDIM)
                exit = true;
        }
    }

    cout << "Exit input loop" << endl;
    cout << "sum = " << sum << "; num = " << num << endl;

    if (num != 0){
        double mean = sum / num;
        cout << "mean = " << mean << endl;

        cout << endl << setw(20) << "Prices" << setw(40) << "Deviation from the mean" << endl;

        for (unsigned i = 0; i < num; ++i){   // for (unsigned i = 1; i < num; ++i)
            double deviation = prices[i] - mean;
            cout << setw(20) << prices[i] << setw(20) << deviation << endl;
        }
    }

    return 0;
}
```

this has the same meaning as "( )" (means that the function takes no values)

= "while not exit"

= "if not cin" means if the value of the cin is not proper (we expect a double, anything which is not a double is not proper)

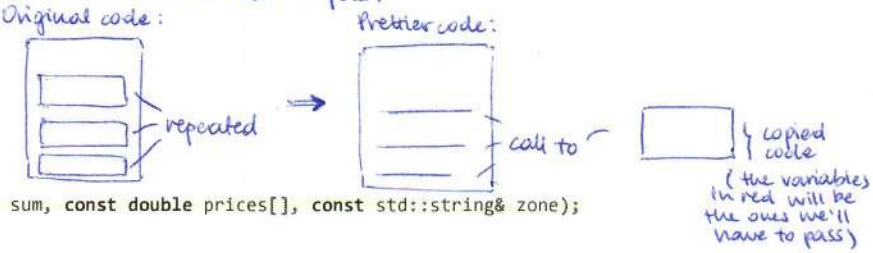
number of characters that we print between words

```cpp
// Price deviation evaluation v2

#include <iostream>
#include <iomanip>

using std::cout;
using std::cin;
using std::endl;
using std::setw;

int main (void){
    const unsigned MAXDIM = 3;
    const char CENTER = 'c';
    const char SUBURB = 's';

    double pricesC[MAXDIM];
    double pricesS[MAXDIM];
    unsigned numC = 0;
    unsigned numS = 0;
    double sumC = 0.;
    double sumS = 0.;
    bool exit = false;

    while (! exit){
        double price;
        char zone;
        cout  << "Input price: ";
        cin >> price;
        cout  << "Input zone: ";
        cin >> zone;

        if (! cin || (zone != CENTER && zone != SUBURB))
            exit = true;
        else{
            if (zone == CENTER){
                pricesC[numC++] = price;
                sumC += price;

                if (numC >= MAXDIM)
                    exit = true;
            }
            else{
                pricesS[numS++] = price;
                sumS += price;

                if (numS >= MAXDIM)
                    exit = true;
            }
        }
    }

    cout << "Exit input loop" << endl;
    cout << "sumC = " << sumC << "; numC = " << numC << endl;
    cout << "sumS = " << sumS << "; numS = " << numS << endl;

    if (numC != 0){
        double mean = sumC / numC;
        cout << "mean center = " << mean << endl;

        cout << endl << setw(20) << "Prices (center)" << setw(40) << "Deviation from the mean (center)" << endl;

        for (unsigned i = 0; i < numC; ++i){
            double deviation = pricesC[i] - mean;
            cout << setw(20) << pricesC[i] << setw(20) << deviation << endl;
        }
    }

    if (numS != 0){
        double mean = sumS / numS;
        cout << "mean suburb = " << mean << endl;

        cout << endl << setw(20) << "Prices (suburb)" << setw(40) << "Deviation from the mean (suburb)" << endl;

        for (unsigned i = 0; i < numS; ++i){
            double deviation = pricesS[i] - mean;
            cout << setw(20) << pricesS[i] << setw(20) << deviation << endl;
        }
    }

    return 0;
}
```

Doing like this we're actually allowing only one of the two arrays to be full (once one reach the max number of elements "exit" becomes 1)

This code is not "pretty". It has code lines that repeat (when we do the evaluations for example). We can make it more pretty!
→ VERSION 3

This is the "prettier" version of version 2. Usually we code first and then we realize that we repeat some blocks of code. We create a function and C-lion helps us telling us (in red) which variables we have to pass.




```cpp
// Price deviation evaluation v3

#include <iostream>
#include <iomanip>

using std::cout;
using std::cin;
using std::endl;
using std::setw;

void evaluate_deviations (unsigned num, double sum, const double prices[], const std::string& zone);

int main (void){
    const unsigned MAXDIM = 100;
    const char CENTER = 'c';
    const char SUBURB = 's';

    double pricesC[MAXDIM];
    double pricesS[MAXDIM];
    unsigned numC = 0;
    unsigned numS = 0;
    double sumC = 0.;
    double sumS = 0.;
    bool exit = false;

    while (! exit){
        double price;
        char zone;
        cout  << "Input price: ";
        cin >> price;
        cout  << "Input zone: ";
        cin >> zone;

        if (! cin || (zone != CENTER && zone != SUBURB))
            exit = true;
        else{
            if (zone == CENTER){
                pricesC[numC++] = price;
                sumC += price;

                if (numC >= MAXDIM)
                    exit = true;
            }
            else{
                pricesS[numS++] = price;
                sumS += price;

                if (numS >= MAXDIM)
                    exit = true;
            }
        }
    }

    cout << "Exit input loop" << endl;
    cout << "sumC = " << sumC << "; numC = " << numC << endl;
    cout << "sumS = " << sumS << "; numS = " << numS << endl;

    evaluate_deviations (numC, sumC, pricesC, "center");
    evaluate_deviations (numS, sumS, pricesS, "suburb");

    return 0;
}


void evaluate_deviations (unsigned num, double sum, const double prices[], const std::string& zone)
{
    if (num != 0)
    {
        double mean = sum / num;
        cout << "mean " << zone << " = " << mean << endl;

        cout << endl << setw(20) << "Prices (" << zone << ")"
                     << setw(40) << "Deviation from the mean (" << zone << ")" << endl;

        for (unsigned i = 0; i < num; ++i)
        {
            double deviation = prices[i] - mean;
            cout << setw(20) << prices[i] << setw(20) << deviation << endl;
        }
    }
}
```

The arrays are passed as pointers so to guarantee that the code won't change them we pass them with "const"

we pass strings by references (since they could be very large variables)

**NOTE**: here we're passing the reference to something that is not a variable, so we HAVE to put the "const" or we'll have an error

```cpp
// Price deviation evaluation v4

#include <iostream>
#include <iomanip>

using std::cout;
using std::cin;
using std::endl;
using std::setw;

const unsigned NZONES = 3;
const unsigned MAXDIM = 5;

void get_prices (unsigned nums[], double sums[], double prices[][MAXDIM]);
void evaluate_deviations (const unsigned nums[], const double sums[], const double prices[][MAXDIM]);

int main (void){
    double prices[NZONES][MAXDIM];
    unsigned nums[NZONES] = {};
    double sums[NZONES] = {};

    get_prices(nums, sums, prices);
    evaluate_deviations(nums, sums, prices);

    return 0;
}

void get_prices (unsigned nums[], double sums[], double prices[][MAXDIM]){
    bool stop = false;

    while (! stop){
        double price;
        unsigned zone;
        cout  << "Input price: ";
        cin >> price;
        cout  << "Input zone: ";
        cin >> zone;

        if (! cin || zone >= NZONES)
            stop = true;
        else{
            prices[zone][nums[zone]++] = price;
            sums[zone] += price;

            if (nums[zone] >= MAXDIM)
                stop = true;
        }
    }

    cout << "Exit input loop" << endl;
}

void evaluate_deviations (const unsigned nums[], const double sums[], const double prices[][MAXDIM]){
    for (unsigned z = 0; z < NZONES; ++z){
        if (nums[z] != 0){
            double mean = sums[z] / nums[z];
            cout << "mean in zone " << z << " = " << mean << endl;

            cout << endl << setw(20) << "Prices (" << z << ")"
                 << setw(40) << "Deviation from the mean (" << z << ")" << endl;

            for (unsigned i = 0; i < nums[z]; ++i){
                double deviation = prices[z][i] - mean;
                cout << setw(20) << prices[z][i] << setw(20) << deviation << endl;
            }
        }
    }
}
```

we're passing them without "const" because we need to modify them

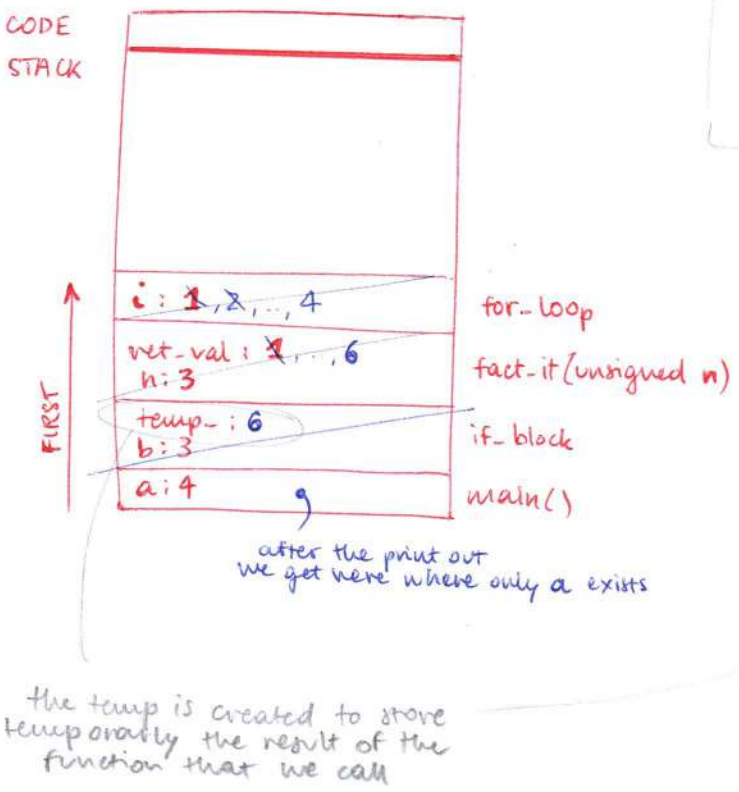
To pass a matrix we have to pass the second dimension. We cannot write double prices[][].
Why? Because the matrix: $\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}$ is saved in memory as [1 2 3 4 5 6];
knowing how many columns we have, we know where to split the array to reconstruct it as matrix (we receive [1 2 3 4 5 6] and the information that we have two columns ⇒ we obtain: $\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}$)

# Factorial

```cpp
#include <iostream>

unsigned fact_rec(unsigned n);
unsigned fact_it(unsigned n);

int main() {
    unsigned a = 4;
    if (a % 2 ==0){
        unsigned b = 3 ;
        std::cout << fact_it(b) << std::endl;
    }
    else{
        unsigned c = 4;
        std::cout << fact_rec(c) << std::endl;
    }

    return 0;
}

unsigned fact_rec(unsigned n) {
    unsigned ret_val;
    if (n == 0)
        ret_val = 1;
    else
        ret_val = n * fact_rec(n-1);
    return ret_val;
}

unsigned fact_it(unsigned n){
    unsigned ret_val = 1;
    for (unsigned i = 1; i<= n; ++i)
        ret_val *= i;
    return ret_val;
}
```

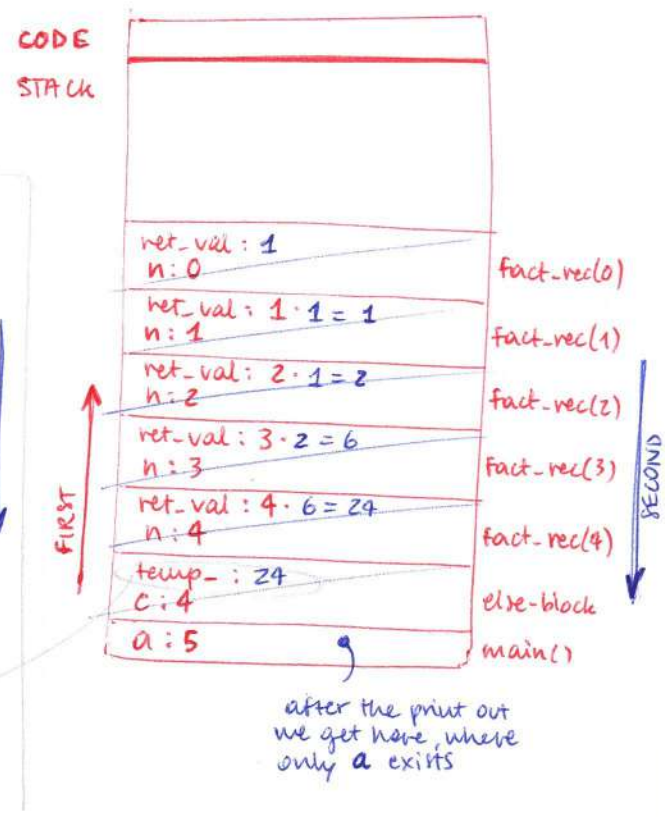
we have 2 versions: if the number is even we run the iterative version, otherwise the recursive

when a = 4 :
(notice: we don't have global variables and so we don't have static data, moreover we don't use dynamic data, so we don't have heap either)




the temp is created to store temporarily the result of the function that we call

when a = 5 :



deep se
dependable evolvable pervasive software engineering group

# Lab session: C functions

Politecnico di Milano

POLITECNICO
DI MILANO



## The problem - Sudoku

- In the Sudoku game, a 9x9 grid must be filled with numbers from 1 to 9
- The grid is subdivided in nine rows, nine columns and nine 3x3 subregions; all rows, columns and subregions must contain exactly all numbers from 1 to 9

Example:

```
1 2 3 | 4 5 6 | 7 8 9
4 5 6 | 7 8 9 | 1 2 3
7 8 9 | 1 2 3 | 4 5 6
-------------------------
2 3 4 | 5 6 7 | 8 9 1
5 6 7 | 8 9 1 | 2 3 4
8 9 1 | 2 3 4 | 5 6 7
-------------------------
3 4 5 | 6 7 8 | 9 1 2
6 7 8 | 9 1 2 | 3 4 5
9 1 2 | 3 4 5 | 6 7 8
```



## The problem - Sudoku

- **Goal:** to write a program that checks whether a given Sudoku matrix is correct

- A search function

```
int search_key (const int v[], int n_elements, int key);
```

that returns 1 if `key` is contained in the array `v`, 0 otherwise, is given

- To generate a Sudoku matrix, the given function

```
void generate_sudoku(int sudoku[][SIZE]);
```

can be used

# The problem - Sudoku

**You have to implement:**

1) a function

```
int basic_search (const int v[], int n_elements);
```

that returns 1 if all numbers from 1 to 9 are contained in v, 0 otherwise

2) three functions

```
int check_rows (const int sudoku[][SIZE]);
int check_cols (const int sudoku[][SIZE]);
int check_regions (const int sudoku[][SIZE]);
```

that return 1 if all the rows, columns and subregions (respectively) of the Sudoku matrix comply with the rules, 0 otherwise



# The problem - Sudoku

3) a function

```
int check_sudoku(const int sudoku[][SIZE]);
```

that returns:

- 1 if the given Sudoku matrix complies to all Sudoku rules
- -1 if a row violates the game rules
- -2 if a column violates the game rules
- -3 if a region violates the game rules



# Credits

- Lewis' algorithm to generate a Sudoku matrix
  https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

# Solution

1) basic search:

```
int basic_search (const int v[], int n_elements)
{
    int found = 1;

    for (int i=1; i<=n_elements && found==1; ++i)
        found = search_key(v, n_elements, i);

    return found;
}
```



2) check rows:

```
int check_rows (const int sudoku[][SIZE])
{
    int v[SIZE];
    int ris = 1;

    for (int i=0; i<SIZE && ris==1; ++i)
    {
        // copy row to array v
        for (int j=0; j<SIZE; ++j)
            v[j] = sudoku[i][j];

        // check if v includes all integers between 1 and SIZE
        ris = basic_search(v, SIZE);

        // Alternative call:
        // ris = basic_search(sudoku[i], SIZE);
    }
    return ris;
}
```



3) check columns:

```
int check_cols (const int sudoku[][SIZE])
{
    int v[SIZE];
    int ris = 1;

    for (int i=0; i<SIZE && ris==1; ++i)
    {
        // copy column to array v
        for (int j=0; j<SIZE; ++j)
            v[j] = sudoku[j][i];

        // check if v includes all integers
        // between 1 and SIZE
        ris = basic_search(v, SIZE);
    }
    return ris;
}
```

4) check regions:

```
int check_regions (const int sudoku[][SIZE])
{
    int v[SIZE];
    int ris = 1;

    for (int i=0; i<3 && ris==1; ++i)
    {
        for (int j=0; j<3 && ris==1; ++j)
        {
            int index = 0;

            for (int i1=3*i; i1<3*i+3; ++i1)
                for (int j1=3*j; j1<3*j+3; ++j1)
                    v[index++] = sudoku[i1][j1];

           // check if v includes all integers between 1 and SIZE
            ris = basic_search(v, SIZE);
        }
    }
    return ris;
}
```



5) check sudoku:

```
int check_sudoku (const int sudoku[][SIZE])
{
    int ris;
    // check rows
    ris = check_rows(sudoku);
    if (ris==0)
    {
        cout << "Rows rules are violated" << endl;
        return -1;
    }
    // check columns
    ris = check_cols(sudoku);
    if (ris==0)
    {
        cout << "Columns rules are violated" << endl;
        return -2;
    }
    // check regions
    ris = check_regions(sudoku);
    if (ris==0)
    {
        cout << "Regions rules are violated" << endl;
        return -3;
    }
    // If we reach here, no errors were found
    cout << "Matrix ok!" << endl;
    return 1;
}
```

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
const size_t SIZE = 9;

// Return 1 if key is in v; n_elements is v size
int search_key (const unsigned v[], unsigned n_elements, unsigned key);

// Return 1 if v includes all numbers in [1,9]; n_elements is v size
int basic_search (const unsigned v[], unsigned n_elements);

// Return 1 if all sudoku matrix rows comply to the Sudoku rules
int check_rows (const unsigned sudoku[][SIZE]);

// Return 1 if all sudoku matrix columns comply to the Sudoku rules
int check_cols (const unsigned sudoku[][SIZE]);

// Return 1 if all sudoku matrix regions comply to the Sudoku rules
int check_regions (const unsigned sudoku[][SIZE]);

// Return:      1 if sudoku matrix complies to all Sudoku rules
//             -1 if a row violates the game rules
//             -2 if a column violates the game rules
//             -3 if a region violates the game rules
int check_sudoku(const unsigned sudoku[][SIZE]);

// Create a Sudoku matrix my Lewis' Algorithm (https://en.wikipedia.org/wiki/Sudoku_solving_algorithms)
void generate_sudoku(unsigned sudoku[][SIZE]);

int main(){
    // initialize a sudoku matrix
    unsigned sudoku[SIZE][SIZE] = {
            {1,2,3,4,5,6,7,8,9},
            {2,3,4,5,6,7,8,9,1},
            {3,4,5,6,7,8,9,1,2},
            {4,5,6,7,8,9,1,2,3},
            {5,6,7,8,9,1,2,3,4},
            {6,7,8,9,1,2,3,4,5},
            {7,8,9,1,2,3,4,5,6},
            {8,9,1,2,3,4,5,6,7},
            {9,1,2,3,4,5,6,7,8}};

    // check
    int res = check_sudoku(sudoku);
    cout << "check_sudoku returns: " <<  res << endl;

    // initialize another sudoku matrix
    unsigned sudoku2[SIZE][SIZE];
    generate_sudoku(sudoku2);

    // check
    res = check_sudoku(sudoku2);
    cout << "check_sudoku returns: " <<  res << endl;

    return 0;
}

int search_key (const unsigned v[], unsigned n_elements, unsigned key){
    int key_found = 0;
    for (size_t i=0; i<n_elements; ++i)
        if (v[i] == key)
            key_found = 1;
    return key_found;
}

int basic_search (const unsigned v[], unsigned n_elements){
    int found = 1;
    for (size_t i=1; i<=n_elements && found==1; ++i)
        found = search_key(v, n_elements, i);
    return found;
}

int check_rows (const unsigned sudoku[][SIZE]){
    unsigned v[SIZE];
    int ris = 1;
    for (size_t i=0; i<SIZE && ris==1; ++i){
        // copy row to array v
        // REMEMBER: array must be copied element by element
        for (size_t j=0; j<SIZE; ++j)
            v[j] = sudoku[i][j];

        // check if v includes all integers between 1 and SIZE
        ris = basic_search(v, SIZE);
```

```
        /* Alternative call: ris = basic_search(sudoku[i], SIZE);
         * And you don't need to copy elements to v! Why is that possible!?*/

        //cout << "Row " << i << " ris: " << ris << endl;
    }
    return ris;
}

int check_cols (const unsigned sudoku[][SIZE]){
    unsigned v[SIZE];
    int ris = 1;
    for (size_t i=0; i<SIZE && ris==1; ++i){
        // copy column to array v
        for (size_t j=0; j<SIZE; ++j)
            v[j] = sudoku[j][i];

        // check if v includes all integers between 1 and SIZE
        ris = basic_search(v, SIZE);

        //cout << "Col " << i << " ris: " << ris << endl;
    }
    return ris;
}

int check_regions (const unsigned sudoku[][SIZE]){
    unsigned v[SIZE];
    int ris = 1;
    for (size_t i=0; i<3 && ris==1; ++i){
        for (size_t j=0; j<3 && ris==1; ++j){
            size_t index = 0;
            for (size_t i1=3*i; i1<3*i+3; ++i1)
                for (size_t j1=3*j; j1<3*j+3; ++j1)
                    v[index++] = sudoku[i1][j1];

            // check if v includes all integers between 1 and SIZE
            ris = basic_search(v, SIZE);
        }
    }
    return ris;
}

int check_sudoku (const unsigned sudoku[][SIZE]){
    int ris;

    // check rows
    ris = check_rows(sudoku);
    if (ris==0){
        cout << "Rows rules are violated" << endl;
        return -1;
    }

    // check columns
    ris = check_cols(sudoku);
    if (ris==0){
        cout << "Columns rules are violated" << endl;
        return -2;
    }

    // check regions
    ris = check_regions(sudoku);
    if (ris==0){
        cout << "Regions rules are violated" << endl;
        return -3;
    }

    // If we reach here, no errors were found
    cout << "Matrix ok!" << endl;
    return 1;
}

void generate_sudoku(unsigned sudoku[][SIZE]){
    unsigned x = 0;
    for (size_t i=1; i<=3; ++i){
        for (size_t j=1; j<=3; ++j){
            for (size_t k=1; k<=SIZE; ++k){
                sudoku[3*(i-1)+j-1][k-1] = (x % SIZE) + 1;
                x++;
            }
            x += 3;
        }
        x++;
    }
}
```

this is because sudoku is an array of arrays so sudoku[i] represents the i-th array (which is a row). However we can do it only for rows, not columns.

same as with rows but with indeces inverted

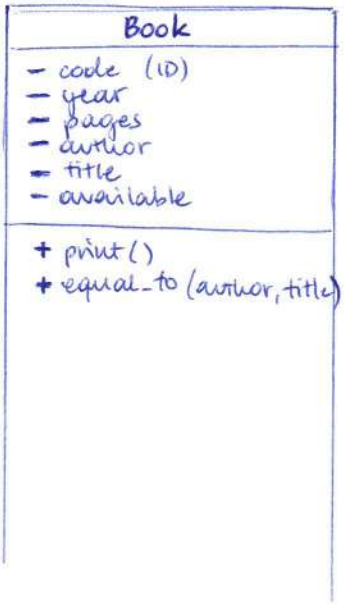
# Library

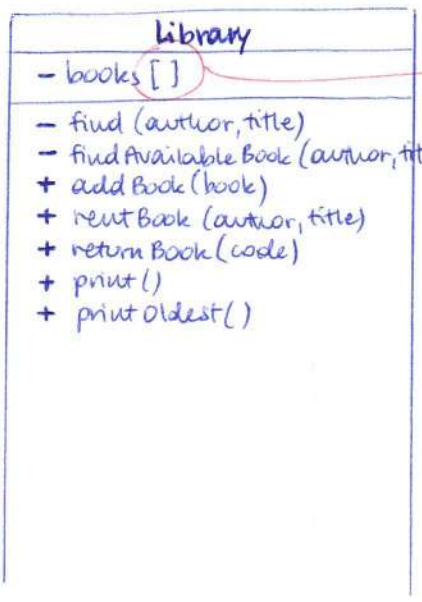
```
#include <iostream>
#include "Book.h"
#include "Library.h"

int main() {

    Book b1(1, 2013, 1399, "S. B. Lippman", "C++ primer", true);

    Book b2(2, 2013, 1361, "B. Stroustrup", "The C++ programming language", true);

    /*
    b1.print();
    b2.print();
    */

    Library lib;

    lib.addBook(b1);
    lib.addBook(b2);

    lib.rentBook("S. B. Lippman", "C++ primer");

    lib.print();
    cout << "****************+"<< endl;
    lib.rentBook("S. B. Lippman", "C++ primer");
    lib.print();
    cout << "****************+"<< endl;

    lib.returnBook(1);
    lib.print();
    cout << "****************+"<< endl;

    lib.returnBook(1);
    lib.print();

    cout << "****************+"<< endl;

    cout << "Oldest book" << endl;

    lib.printOldest();
    return 0;
}
```

we're defining **2** books thanks to the constructor

Class diagrams (syntetic representation) :


"+" public
"-" private




this is just to say that we have a collection, we don't mind if it's an array or a vector

```
// Book.h

#ifndef LIBRARY_BOOK_H
#define LIBRARY_BOOK_H

#include <string>

#include <iostream>

using std::cout;
using std::endl;
using std::string;

class Book {
private:
    unsigned code;
    unsigned year;
    unsigned pages;
    string author;
    string title;
    bool available;

public:
    Book(unsigned code, unsigned year, unsigned pages, const string &author, const string &title, bool available);
    unsigned int getCode() const;
    unsigned int getYear() const;
    unsigned int getPages() const;
    const string getAuthor() const;
    void setAvailable(bool av);
    bool isAvailable() const;
    const string getTitle() const;
    bool equal_to(const string &aut, const string &t) const;
    void print() const;
};

#endif //LIBRARY_BOOK_H
```

we need a **CONSTRUCTOR** to create an object of the class (as parameters we need to use the order of above)

All the class member are private: we need to access to them so we create public methods. We actually create **2** types of methods: getter and setter. The getter methods' function is to get the values and so these methods are **const**. The setter methods' change the values (and so the object).

The getters can be automatized (are given, if requested, by default). The yellow methods instead are the ones we're choosing to add → we have to implement them ourselves.

```cpp
// Book.cpp

#include "Book.h"

Book::Book(unsigned code, unsigned year, unsigned pages, const string &author, const string &title, bool available)
        : code(code),year(year), pages(pages),
          author(author),
          title(title),
          available(available){}

unsigned int Book::getYear() const {
    return year;
}

unsigned int Book::getPages() const {
    return pages;
}

const string Book::getAuthor() const {
    return author;
}

const string Book::getTitle() const {
    return title;
}

bool Book::equal_to(const string &aut, const string &t) const {
    return author == aut && title == t;
}

void Book::print() const {
    cout << code << "\n"
         << author << "\n"
         << title << "\n"
         << pages << "\n"
         << year << "\n";

         if (available)
             cout << "Available";
         else
             cout << "Not available";

         cout << endl;

}

bool Book::isAvailable() const {
    return available;
}

void Book::setAvailable(bool av) {
    available = av;
}

unsigned int Book::getCode() const {
    return code;
}
```

Why we write aut and t instead of author and title? Otherwise in the return we would have:

return (author == author) && (title == title);

Because of variable scope, we have no way to distinguish the author we're passing as const string & and Book:: author (That's because of *).

This will be always 1.
What is local is stronger than whatever is outside so here we're comparing the author we're passing to this function with itself.

notice that here we don't need to write Book:: available because we're already in "Book" *

```cpp
// Library.h

#ifndef LIBRARY_LIBRARY_H
#define LIBRARY_LIBRARY_H

#include <vector>
#include <iostream>

#include "Book.h"

using std::vector;
using std::string;
using std::cerr;
using std::endl;

class Library {

private:
    vector<Book> books;

    // return the index of the book
    int find(const string & author, const string & title) const;

    // return the index of the first available book
    int findAvailableBook(const string & author, const string & title) const;

public:
    // add a book to the library
    void addBook(const Book & book);

    // return the book code or -1 if the book is not available
    int rentBook(const string & author, const string & title);

    // return to the library the book with the specified code.
    // Return false if the code is not found/the book is available
    // return true otherwise
    bool returnBook(unsigned code);

    void print() const;

    void printOldest() const;

};

#endif //LIBRARY_LIBRARY_H
```

since we have to use books we add the header file of books

why vectors and not arrays? We don't want constraints about the size (with arrays we have to know the size in advance)

what does it mean for a method to be constant? IT DOES **NOT** CHANGE THE STATE OF THE OBJECT. IT DOES **NOT** CHANGE THE OBJECT.

For example the method "addBook" is adding a book to the library → is not const since it's changing the state of the library

book is a large thing, we pass it through const reference

This will print the full content of the library

```cpp
// Library.cpp

#include "Library.h"

int Library::find(const string &author, const string &title) const{
    int return_index = -1;
    bool book_found = false;

    for (size_t i =0; i< books.size() && !book_found; ++i)
        if (books[i].equal_to(author,title)){
            book_found = true;
            return_index = i;
        }

    return return_index;
}

int Library::findAvailableBook(const string &author, const string &title) const{
    int return_index = -1;
    bool book_found = false;

        for (size_t i =0; i< books.size() && !book_found; ++i)
        if (books[i].equal_to(author,title) && books[i].isAvailable()){
            book_found = true;
            return_index = i;
        }

    return return_index;
}

void Library::addBook(const Book &book) {
    books.push_back(book);
}

int Library::rentBook(const string &author, const string &title) {
    int index = findAvailableBook(author,title);
    if (index == -1){
        cerr<< "Book not available" << endl;
        return -1;
    }
        else{
        books[index].setAvailable(false);
        return books[index].getCode();
    }
}

bool Library::returnBook(unsigned code) {
    bool book_found = false;
    size_t i;

        for (i =0; i< books.size() && !book_found; ++i)
        if (books[i].getCode() == code && !books[i].isAvailable())
            book_found = true;

    if (book_found){
        books[i-1].setAvailable(true);
        return true;
    }
    else{
        cerr << "Book already in store" << endl;
        return false;
    }

}

void Library::print() const {

    for (size_t i = 0; i < books.size(); ++i) {
        books[i].print();
    }
}

void Library::printOldest() const {
        size_t oldest = 0;

    for (size_t i=1; i< books.size(); ++i)
        if(books[i].getYear()<books[oldest].getYear())
            oldest = i;

    books[oldest].print();
}
```

we rely on the method "equal_to" from the class "Book" since books[i] is an object of that class

Almost equal to Library::find(..) with the difference that we also have to check if a book is available

in the public part we see that the array/collection of books is called "books"

we rely on another method of the same class (that's why we don't have to put ".")

This return the code of the book if we find the book and it's available, -1 otherwise

This is for printing the errors (we can also use cout but this has some features)

we first modify the availability of the book and then we return its code (we do both through methods of "Book")

we return true if everything is get done right, false otherwise

This time we need the code, not author and title. Moreover it has to be NOT available.

We put "i-1" because when we exit from the for loop (supposing that we found the book) we have that i is incremented

for (i=0; i<n; i++)
- we start from i=0
- we do the loop if i<n
- at the end of the cycle (one step of the cycle (each step)) we increment i

We rely on the method from the class "Book" (since books[i] is an element (an object) of the class "Book")

we start from 1 since we're starting from the setting: "the first book (position 0) is the oldest"

# Complex Numbers

```
//-------------------------------------------------------------------------------------------------
// main.cpp
//-------------------------------------------------------------------------------------------------
#include <iostream>
#include "Complex.h"

int main() {
    Complex c1(5, 2);
    Complex c2(6, 3);
    Complex c3 = c1;
    c3 = c1 + c2;
    Complex c4 = c2 + 5;
    Complex c5 = 5 + c2;
    return 0;
}
```

because of these we have to overload the "+" operator

```
//-------------------------------------------------------------------------------------------------
// Complex.h
//-------------------------------------------------------------------------------------------------
#ifndef COMPLEX_COMPLEX_H
#define COMPLEX_COMPLEX_H

class Complex {
private:
    double m_real;
    double m_img;
public:
    // constructor
    Complex(double real, double img): m_real(real), m_img(img){};

    // getters
    double get_img() const;
    double get_real() const;
    double get_modulus() const;
    double get_phase() const;

    // setters
    void set_real(double);
    void set_img(double);
    void set_modulus(double);
    void set_phase(double);

    // operators
    Complex operator+(const Complex&) const;
    Complex operator-(const Complex&) const;
    Complex operator*(const Complex&) const;
    Complex operator/(const Complex&) const;
};

// operators as global functions to have parameters in different orders
Complex operator+(const Complex&, double);
Complex operator+(double, const Complex&);

#endif //COMPLEX_COMPLEX_H
```

otherwise we could just put Complex(double real, double img); in the header and in the .cpp we would have written:

```
Complex::Complex(double real, double img){
    m_real = real;
    m_img = img;
}
```

or: `Complex::Complex(double real, double img): m_real(real), m_img(img){};` (*)

through this operator we create a new element of type "Complex" and we don't modify anything

$$z = x + iy \quad ; \quad r = |z| = \sqrt{x^2 + y^2}$$

$$\varphi = \arg(z) = \begin{cases} \operatorname{atan}(y/x) & x > 0 \\ \operatorname{atan}(y/x) + \pi & x < 0 \quad y \geq 0 \\ \operatorname{atan}(y/x) - \pi & x < 0 \quad y < 0 \\ \pi/2 & x = 0 \quad y > 0 \\ -\pi/2 & x = 0 \quad y < 0 \\ \text{indeterminate} & x = 0 \quad y = 0 \end{cases}$$

$$(a+ib)(c+id) = (ac-bd) + (bc+ad)i$$

$$\frac{a+ib}{c+id} = \frac{(ac+bd) + i(bc-ad)}{c^2 + d^2}$$

We have to introduce these funs here because all the operators inside a class have as first parameter an object of the class ⟹ if so we cannot define "5 + c2" inside the class because 5 (which is the first) is not a complex number (∉ class) ⟹ we implement them outside as **HELPER FUNCTIONS**

(*) **However**: if we would have written `Complex(double m_real, double m_img);` in the header, we would have had 2 options in the .cpp:

1.
```
Complex::Complex(double m_real, double m_img){
    this -> m_real;
    this -> m_img;
}
```

2.
```
Complex::Complex(double m_real, double m_img):
    m_real(m_real), m_img(m_img){};
```

```cpp
//-----------------------------------------------------------------------------
// Complex.cpp
//-----------------------------------------------------------------------------
#include "Complex.h"
#include <math.h>

double Complex::get_real() const {
    return m_real;
}

double Complex::get_img() const {
    return m_img;
}

double Complex::get_modulus() const {
    return sqrt(m_img*m_img + m_real*m_real);
}

double Complex::get_phase() const {
    double sign = m_img >= 0 ? 1.0 : -1.0;
    if(m_real != 0){
        double res = atan(m_img/m_real);
        if(m_real < 0)
            res += sign * M_PI;
        return res;
    }
    else {
        return sign * (M_PI/2.0);
    }
}

void Complex::set_real(double real) {
    m_real = real;
}

void Complex::set_img(double img) {
    m_img = img;
}

void Complex::set_modulus(double mod) {
    double ph = get_phase();
    m_real = mod * cos(ph);
    m_img = mod * sin(ph);
}

void Complex::set_phase(double phase) {
    double mod = get_modulus();
    m_real = mod * cos(phase);
    m_img = mod * sin(phase);
}

Complex Complex::operator+(const Complex &rhs) const {
    Complex res(m_real+rhs.m_real, m_img+rhs.m_img);
    return res;
}

Complex Complex::operator-(const Complex &rhs) const {
    Complex res(m_real-rhs.m_real, m_img-rhs.m_img);
    return res;
}

Complex Complex::operator*(const Complex &rhs) const {
    double real = m_real*rhs.m_real - m_img*rhs.m_img;
    double img = m_img*rhs.m_real + m_real*rhs.m_img;
    return Complex(real, img);
}

Complex Complex::operator/(const Complex &rhs) const {
    double den = rhs.m_real*rhs.m_real + rhs.m_img*rhs.m_img;
    double r_num = m_real*rhs.m_real + m_img*rhs.m_img;
    double i_num = m_img*rhs.m_real - m_real*rhs.m_img;
    return Complex(r_num/den, i_num/den);
}

Complex operator+(const Complex &lhs, double real) {
    Complex res(lhs.get_real()+real, lhs.get_img());
    return res;
}

Complex operator+(double real, const Complex &lhs) {
    return lhs + real;
}
```

for the square root, atan, sin/cos, .. (notice: math.h does not use the std::)

This is because according to the sign of y we will add/substract π or (if x<0) it'll be ± π/2

((condition) ? (if true) : (if false))

$$arg(z) = \begin{cases} atan(y/x) & x>0 \\ atan(y/x)+\pi & x<0 \quad y\geq 0 \\ atan(y/x)-\pi & x<0 \quad y<0 \\ \pi/2 & x=0 \quad y>0 \\ -\pi/2 & x=0 \quad y<0 \\ \text{indeterminate} & x=0 \quad y=0 \end{cases}$$

in math.h it's π

(We have a conditional assignment)
Here we're saying:
condition: m_img >= 0
if true : double sign = 1.0;
if false : double sign = -1.0;

we can also return and create together

This is calling the previous +

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Gradient Descent

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it




## Goal

Provide the implementation of **gradient descent** algorithm for **polynomial functions**

$$f : \mathbb{R} \to \mathbb{R}$$

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n = \sum_{k=0}^{n} a_k x^k$$



## Gradient Descent algorithm

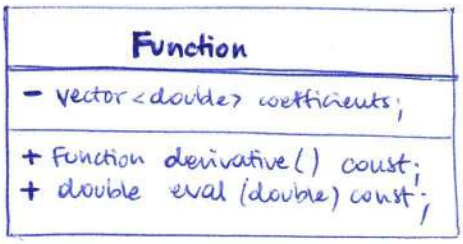
- Find the minimum of `f` in a given interval `[inf, sup]`
- Start from a given initial point `x0`
- Next candidate:

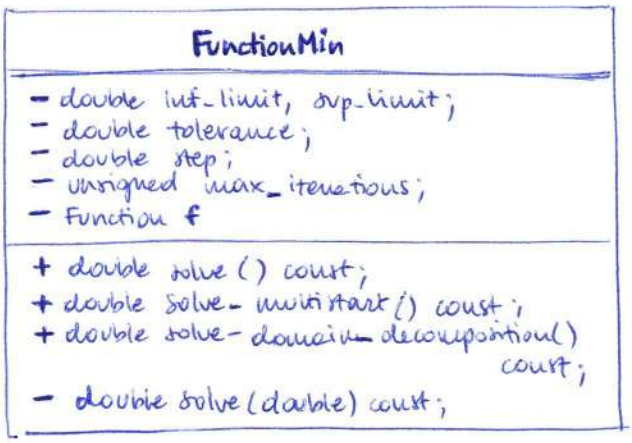
$$x_i = x_{i-1} - \delta f'(x_{i-1})$$

where $\delta$ is a given constant

# Required methods

- `solve`, that finds the minimum of `f` in `[inf, sup]`
- `solve_multistart`, that randomly picks an initial point in `[inf, sup]` at each iteration
- `solve_domain_decomposition`, that performs multistart after having split `[inf, sup]` in a given number of subintervals



```
//-----------------------------------------------------------------------------
// Function.h
//-----------------------------------------------------------------------------
#ifndef FUNCTION_H_
#define FUNCTION_H_

#include <iostream>
#include <vector>

class Function{
private:
  std::vector<double> coefficients;

public:
  Function (std::vector<double> coeff): coefficients (coeff) {};
  double eval (const double x) const;
  Function derivative() const;
  void print() const;
};

#endif /* FUNCTION_H_ */
```

we don't need to store the degree, since:
$[a_0\ a_1\ a_2\ ..] \Rightarrow f = a_0 + a_1x + a_2x^2 + ..$

← constructor

we don't need to modify the current function to get derivative and evaluation

```
//-----------------------------------------------------------------------------
// Function.cpp
//-----------------------------------------------------------------------------
#include <cmath>
#include "Function.hpp"

double Function::eval (const double x) const{
  double val = 0;
  for (int i = 0; i < coefficients.size(); i++)
    val += coefficients[i] * pow (x, i);
  return val;
}

Function Function::derivative() const{
  std::vector<double> dcoefficients;
  for (int i = 0; i < coefficients.size() - 1; i++)
    dcoefficients.push_back ((i + 1)*coefficients[i + 1]);
  return Function (dcoefficients);
}

void Function::print() const{
  for (int i = 0; i < coefficients.size(); i++)
    std::cout << coefficients[i] << "  ";
  std::cout << std::endl;
}
```

we need $i=0$ because if $f(x) = a_0$ this writing won't create problems, instead $i=1$ would (in case of $f(x)=a_0$ there ∄ coefficients[1])

we **cannot** write dcoefficients[i] = .. because we're creating dcoefficients

**alternative**:

```
for (double c : coefficients)
    std:: cout << c << "  ";
```

```cpp
//---------------------------------------------------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "Function.hpp"
#include "FunctionMin.hpp"

using namespace std;

int main(){
  Function f ({1., 1., 2., -10., 2.});
  std::cout << "Function: " << std::endl;
  f.print();
  std::cout << "Function: " << f.eval (1) << std::endl;
  FunctionMin minF (f, -1, 4, 1e-3, 1e-3, 1000000);
  std::cout << "Function minimum at: " << minF.solve() << std::endl;
  std::cout << "Function minimum (multi-start) at: "
            << minF.solve_multistart (100000) << std::endl;
  std::cout << "Function minimum (multi-start domain decomp) at: "
            << minF.solve_domain_decomposition (10, 100000) << std::endl;
  return 0;
}
```

```
//-----------------------------------------------------------------------------
// FunctionMin.h
//-----------------------------------------------------------------------------
#ifndef FUNCTIONMIN_H_
#define FUNCTIONMIN_H_

#include "Function.h"

class FunctionMin{
  Function f;
  double inf_limit;
  double sup_limit;
  double tolerance;
  double step;
  unsigned int max_iterations;

  //gradient descent with x_init initial point
  double solve (double x_init) const;

public:
  FunctionMin (Function func, double a, double b, double tol, double s, unsigned int max_it)
    : f (func), inf_limit (a), sup_limit (b), tolerance (tol), step (s), max_iterations (max_it) {};

  // gradient descent
  double solve (void) const;
  // gradient descent with multi-start
  double solve_multistart (unsigned int n_trials) const;
  // gradient descent with multi-start and domain decomposition
  double solve_domain_decomposition (unsigned int n_intervals, unsigned int n_trials) const;
};

#endif /* FUNCTIONMIN_H_ */
```

$\delta$ in: $x_i = x_{i-1} - \delta f'(x_i)$

```
//-----------------------------------------------------------------------------
// FunctionMin.cpp
//-----------------------------------------------------------------------------
#include <algorithm>
#include <cmath>
#include <random>
#include "FunctionMin.hpp"

//gradient descent with x_init initial point
double FunctionMin::solve (double x_init) const{
  Function df = f.derivative();
  double x0 = x_init;
  double f0 = f.eval (x0);
  bool converged = ((sup_limit - inf_limit) < tolerance) || std::abs (df.eval (x0)) < tolerance);
  unsigned int i;
  for (i = 0; i < max_iterations && ! converged; ++i){
      const double deriv = df.eval (x0);
      double x1 = x0 - deriv * step;
      if (deriv > 0)
        x1 = std::max (inf_limit, x1);
      else
        x1 = std::min (sup_limit, x1);
      const double f1 = f.eval (x1);
      converged = (std::abs (f1 - f0) < tolerance) || std::abs (df.eval (x1)) < tolerance);
      x0 = x1;
      f0 = f1;
    }
  return x0;
}

// gradient descent
double FunctionMin::solve (void) const{
  return solve ((sup_limit + inf_limit) / 2);
}

// gradient descent with multi-start
double FunctionMin::solve_multistart (unsigned int n_trials) const{
  std::default_random_engine generator;
  std::uniform_real_distribution<double> distribution (inf_limit, sup_limit);
  double x_min = solve ();
  for (unsigned int n = 1; n < n_trials; ++n){
      const double x_guess = distribution (generator);
      const double x_new = solve (x_guess);
      if (f.eval (x_new) < f.eval (x_min))
        x_min = x_new;
    }
  return x_min;
}
```

we need to compute: $x_i = x_{i-1} - step \cdot f'(x_{i-1})$ (*)

if sup_limit and inf_limit are too close then we can't move (inside the interval)

$|f'(x_0)| < \varepsilon$

in one loop we never modify it

(*)

this is because we don't want $x_1$ to go out of the interval because of the update ($x_1 = x_0 - step \cdot f'(x_0)$) (this condition constraints $x_1$ to be inside the interval)

this is what will be called by the user

random distribution (uniform ([inf_limit, sup_limit]))

how we pick from a distribution

Use const how much as we can!

```
// gradient descent with multi-start and domain decomposition
double FunctionMin::solve_domain_decomposition (unsigned n_intervals, unsigned n_trials) const{
  const double internal_step = (sup_limit - inf_limit) / n_intervals;
  double internal_inf_limit = inf_limit;
  double x_min = inf_limit;
  for (unsigned int i = 1; i <= n_intervals; ++i){
      //That's inefficient!! Think how to improve this!
      FunctionMin minf_int (f, internal_inf_limit, internal_inf_limit + internal_step, tolerance, step, max_iterations);
      const double x_iter = minf_int.solve_multistart (n_trials);
      if (f.eval (x_iter) < f.eval (x_min))
        x_min = x_iter;
      internal_inf_limit += internal_step;
    }
  return x_min;
}
```

```
    cout << rob.find_root(nit_bis,nit_newt);
    cout << '\t' << nit_bis
        << " "  << nit_newt << endl;

    return 0;
}
```

## Exercise 4 Gradient Descent

Provide the implementation of the gradient descent for polynomial functions $f : \mathbb{R} \to \mathbb{R}$, relying on the implementation of the Function class provided in exercise 3.
The header for the class FunctionMin should be the following:

```
class FunctionMin
{
  Function f;
  double inf_limit;
  double sup_limit;
  double tolerance;
  double step;
  unsigned int max_iterations;

  //gradient descent with x_init initial point
  double solve (double x_init) const;

public:
  FunctionMin (Function func, double a, double b, double tol,
               double s, unsigned int max_it)
    : f (func), inf_limit (a), sup_limit (b), tolerance (tol),
      step (s), max_iterations (max_it) {};

  // gradient descent
  double solve (void) const;
  // gradient descent with multi-start
  double solve_multistart (unsigned int n_trials) const;
  // gradient descent with multi-start and domain decomposition
  double solve_domain_decomposition (unsigned int n_intervals,
                                     unsigned int n_trials) const;
};
```

Suppose that you want to find the minimum in a given interval $[inf\_limit, sup\_limit]$. The solve method will start from a given initial point. At every iteration $i$, the next candidate is computed as $x_i = x_{i-1} - step * f'(x_{i-1})$, where *step* is a given constant. Since the function $f()$ can be neither convex nor concave, provide also the implementation of multi-start (where you randomly pick an initial point within the interval) and multi-start with domain decomposition (where you perform multi-start but splitting the initial interval in n_intervals).
Write your implementation starting from "Function.h" and "Function.cpp" seen in Exercise 3 and from the following code:

********** file main.cpp **********

```
#include <iostream>

```

```
3  #include "Function.h"
4  #include "FunctionMin.h"
5
6  using namespace std;
7
8  int main()
9  {
10   Function f ({1., 1., 2., -10., 2.});
11   std::cout << "Function: " << std::endl;
12   f.print();
13   std::cout << "Function: " << f.eval (1) << std::endl;
14   FunctionMin minF (f, -1, 4, 1e-3, 1e-3, 1000000);
15   std::cout << "Function minimum at: " << minF.solve() << std::endl;
16   std::cout << "Function minimum (multi-start) at: "
17           << minF.solve_multistart (100000) << std::endl;
18   std::cout << "Function minimum (multi-start domain decomp) at: "
19           << minF.solve_domain_decomposition (10, 100000) << std::endl;
20   return 0;
21 }
```

## Exercise 4 - Solution

As you can see in line 15 of the main function, in the case of gradient descent with a single initial point (neither multi-start nor domain decomposition), the user is intended to compute the minimum simply by writing:

```
minF.solve();
```

where minF is an object of type FunctionMin, without providing explicitly the initial point. You can implement the function

```
double FunctionMin::solve (void) const;
```

either starting from a single random point or starting from the midpoint of the interval (as is done in the solution).
The definition of all the methods in the class FunctionMin is provided below (in the file "FunctionMin.cpp").

```
1  #include <algorithm>
2  #include <cmath>
3  #include <random>
4
5  #include "FunctionMin.h"
6
7  //gradient descent with x_init initial point
8  double FunctionMin::solve (double x_init) const
9  {
10   Function df = f.derivative();
11   double x0 = x_init;
12   double f0 = f.eval (x0);
13   bool converged = ((sup_limit - inf_limit) < tolerance
14                   || std::abs (df.eval (x0)) < tolerance);
15   unsigned int i;
16
17   for (i = 0; i < max_iterations && ! converged; ++i)
```

```
18      {
19        const double deriv = df.eval (x0);
20        double x1 = x0 - deriv * step;
21
22        if (deriv > 0)
23          x1 = std::max (inf_limit, x1);
24        else
25          x1 = std::min (sup_limit, x1);
26
27        const double f1 = f.eval (x1);
28        converged = (std::abs (f1 - f0) < tolerance
29                     || std::abs (df.eval (x1)) < tolerance);
30
31        x0 = x1;
32        f0 = f1;
33      }
34
35    return x0;
36  }
37
38  // gradient descent
39  double FunctionMin::solve (void) const
40  {
41    return solve ((sup_limit + inf_limit) / 2);
42  }
43
44  // gradient descent with multi-start
45  double FunctionMin::solve_multistart (unsigned int n_trials) const
46  {
47    std::default_random_engine generator;
48    std::uniform_real_distribution<double> distribution (inf_limit,
49                                                         sup_limit);
50    double x_min = solve ();
51
52    for (unsigned int n = 1; n < n_trials; ++n)
53      {
54        const double x_guess = distribution (generator);
55        const double x_new = solve (x_guess);
56
57        if (f.eval (x_new) < f.eval (x_min))
58          x_min = x_new;
59      }
60
61    return x_min;
62  }
63
64  // gradient descent with multi-start and domain decomposition
65  double FunctionMin::solve_domain_decomposition (unsigned n_intervals,
66                                                  unsigned n_trials) const
67  {
68    const double internal_step = (sup_limit - inf_limit) / n_intervals;
69    double internal_inf_limit = inf_limit;
```

```
70  double x_min = inf_limit;
71
72  for (unsigned int i = 1; i <= n_intervals; ++i)
73   {
74     //That's inefficient!! Think how to improve this!
75     FunctionMin minf_int (f, internal_inf_limit,
76                           internal_inf_limit + internal_step,
77                           tolerance,
78                           step, max_iterations);
79     const double x_iter = minf_int.solve_multistart (n_trials);
80
81     if (f.eval (x_iter) < f.eval (x_min))
82       x_min = x_iter;
83
84     internal_inf_limit += internal_step;
85   }
86
87  return x_min;
88 }
```

The function that computes the minimum starting from a given initial point is implemented in lines 8 to 36. A for loop is performed, computing the new candidate for the minimum, until the maximum number of iterations is reached or until the algorithm converges.
In particular, at the very beginning (see line 13) the variable `converged` becomes **true** when the difference between the two extrema of the inteval or the absolute value of the derivative, evaluated at the candidate minimum, stay below a given tolerance. With this definition, if by chance the initial point `x_init` is already a minimum, the function will never enter in the for loop and will return directly the solution.
In the for loop (see lines 17 to 33), at every iteration we evaluate the derivative and we use it to compute the new value of the candidate minimum.
**Note:** we must be sure that the new value computed through `x_0 - deriv*step` does not exit from the interval we are considering (i.e., it stays between `inf_limit` and `sup_limit`). In order to ensure this, we always define the new value `x1` as the maximum between itself and `inf_limit` (if the derivative is positive) or as the minimum between itself and `sup_limit` (if the derivative is negative).
At the end (see lines 27 to 29), we use the value of $f$ in the candidate minimum in order to check if we reached the convergence (now the difference between the extrema of the interval is subsituted by the difference, in absolute value, between the values of $f$ at the old and the new candidate minimum).
The function `solve()` without inputs (see lines 39 to 42) simply returns the value computed by `solve(x_init)`, where we pass as initial value the midpoint of the interval.
A possible alternative could be to generate a single random point and to run `solve(x_init)` passing it as the initial point. For example, we could write, instead of line 41, the following:

```
std::default_random_engine generator;
std::uniform_int_distribution<double> distribution(inf_limit,sup_limit);
return solve(distribution(generator));
```

The function `solve_multistart` (see lines 45 to 62) is based on the following idea: for every iteration of the for loop that runs from 1 to the maximum number of trials, we generate a random point within the interval $[inf\_limit, sup\_limit]$ and we run `solve(x_guess)` passing that random point as parameter.
A similar procedure is followed in the function `solve_domain_decomposition` (see lines 65 to 88), where we decompose the initial interval $[inf\_limit, sup\_limit]$ in a given number of

subintervals and, in any of them, we compute the minimum through **solve_multistart**. In particular (see lines 68 to 75), we compute the length of any subinterval (namely **internal_step**) as:

$$\frac{sup\_limit - inf\_limit}{n\_intervals}$$

and then, at every iteration of the for loop, we instantiate a new object of type **FunctionMin**, defining as new interval $[internal\_inf\_limit, internal\_inf\_limit + internal\_step]$ (where, of course, the first value given to **internal_inf_limit** is exactly **inf_limit**).

## Exercise 5 Gradient Descent in $\mathbb{R}^n$

Starting from the implementation of the gradient descent seen in Exercise 4, extend it to functions $f : \mathbb{R}^n \to \mathbb{R}$.

## Exercise 5 - Solution

First of all, we implement a class that defines a point in $\mathbb{R}^n$. It stores a vector of **doubles** which represent the coordinates of the point and it owns the following methods:

- **double** distance (**const** Point &p) **const**; computes the Euclidean distance between **this** and the point p.
- std::size_t get_n_dimensions (**void**) **const**; returns the size of the vector of coordinates, which of course corresponds to the dimension of the space.
- **double** get_coord (std::size_t i) **const**; returns the coordinate at index i.
- **void** set_coord (std::size_t i, **double** val); sets to val the value of the coordinate at index i.
- coords_type get_coords (**void**) **const**; returns the whole vector of coordinates of the point.
- **double** euclidean_norm (**void**) **const**; computes the $||.||_2$ norm of the vector of coordinates (as the distance between the point and the origin of the space, which is of course a vector of zeros).
- **double** infinity_norm (**void**) **const**; computes the infinity norm of the vector of coordinates.

All the methods except for **set_coord** are declared as **const** because they do not modify the point.
The implementation is reported in the file "Point.cpp".

********** file Point.h **********

```
1  #ifndef POINT_H_
2  #define POINT_H_
3
4  #include <vector>
5
6  class Point
7  {
8   typedef std::vector<double> coords_type;
9
```

deep se
dependable evolvable pervasive software engineering group

# Lab Session: Classes

Danilo Ardagna, Luca Pozzoni

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO



## The Problem – Social Network

**Goal:** to provide a program that stores information about a social network.

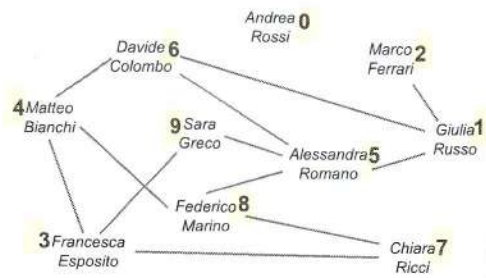
**Users** are uniquely identified by their name and surname. Users are stored in the social network within a vector, thus each user is represented by an index. Assumption: **no homonymous**

**Friendship** is represented by a `vector<vector<size_t>>`: the *i*-th row of this structure stores the list of indices of user *i*'s friends.



## The Problem – Social Network

**Example:**




| | | | |
|---|---|---|---|
| 0 | | | |
| 1 | 2 | 6 | 5 |
| 2 | 1 | | |
| 3 | 4 | 7 | 9 |
| 4 | 3 | 6 | 8 |
| 5 | 1 | 6 | 9 | 8 |
| 6 | 1 | 4 | 5 |
| 7 | 8 | 3 | |
| 8 | 7 | 4 | |
| 9 | 3 | 5 | |

The person 9 is friend with 3 and 5



## The Problem – Social Network

- The class **User**, with the relevant methods, is already provided
- The class **SocialNetwork** only provides the relevant data structures to store users and friendship relations, as well as a *private* method

```
size_t CUserIndex (const std::string & name,
                   const std::string & surname) const;
```

that returns the index of the user whose name and surname are passed as parameters, if he/she is stored in the social network, the size of the vector of users otherwise

# The Problem – Social Network

**You have to implement:**

1) An operator

```
bool operator == (const User & lhs, const User & rhs);
```

that returns **true** if the two users are equal, **false** otherwise.

2) A method

```
void AddUser(const std::string & name,
             const std::string & surname);
```

that receives as parameters the name and surname of a new user and add him/her to the social network.



# The Problem – Social Network

3) Two methods

```
const std::vector<User>
CGetFriends(const User & user) const;
```

and

```
const std::vector<User>
CGetFriends(const std::string & name,
            const std::string  & surname) const;
```

that return the vector of friends of the user passed as parameter.

**Note:** always try to avoid code replication!



# The Problem – Social Network

4) A method

```
void AddFriendship(const std::string & first_name,
                   const std::string & first_surname,
                   const std::string & second_name,
                   const std::string & second_surname);
```

that receives as parameters the names and surnames of two users and adds to the network the friendship relation among them.

**Note:** a user cannot be friend of her/himself and cannot be friend of another user who is not in the social network. Moreover, don't add friendship replicas!

```cpp
//---------------------------------------------------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------------------------------------------------
#include "social_network.hpp"
#include "user.hpp"
#include <iostream>

using namespace SocialNetworkNS;

int main(){
    SocialNetwork social_network = SocialNetwork();
    social_network.AddUser("Andrea", "Rossi");
    social_network.AddUser("Andrea", "Rossi");
    social_network.AddUser("Giulia", "Russo");
    social_network.AddUser("Marco", "Ferrari");
    social_network.AddUser("Francesca", "Esposito");
    social_network.AddUser("Matteo", "Bianchi");
    social_network.AddUser("Alessandra", "Romano");
    social_network.AddUser("Davide", "Colombo");
    social_network.AddUser("Chiara", "Ricci");
    social_network.AddUser("Federico", "Marino");
    social_network.AddUser("Sara", "Greco");

    social_network.AddFriendship("Alessandra", "Romano", "Giulia", "Russo");
    social_network.AddFriendship("Giulia", "Russo","Alessandra", "Romano");
    social_network.AddFriendship("Giulia", "Russo", "Marco", "Ferrari");
    social_network.AddFriendship("Matteo", "Bianchi", "Davide", "Colombo");
    social_network.AddFriendship("Sara", "Greco","Francesca", "Esposito");
    social_network.AddFriendship("Federico", "Marino", "Alessandra", "Romano");
    social_network.AddFriendship("Francesca", "Esposito", "Matteo", "Bianchi");
    social_network.AddFriendship("Alessandra", "Romano", "Davide", "Colombo");
    social_network.AddFriendship("Chiara", "Ricci", "Federico", "Marino");
    social_network.AddFriendship("Giulia", "Russo", "Davide", "Colombo");
    social_network.AddFriendship("Matteo", "Bianchi", "Federico", "Marino");
    social_network.AddFriendship("Chiara", "Ricci", "Federico", "Marino");
    social_network.AddFriendship("Alessandra", "Romano", "Sara", "Greco");
    social_network.AddFriendship("Chiara", "Ricci","Francesca", "Esposito");
    social_network.AddFriendship("Alessandra", "Romano", "Alessandra", "Romano");

    for(const auto & user : social_network.CGetUsers()){
        std::cout << "List of friends of " << user.ToString() << std::endl;
        for(const auto & user_friend : social_network.CGetFriends(user)){
            std::cout << "    " << user_friend.ToString() << std::endl;
        }
    }
    return 0;
}


//---------------------------------------------------------------------------------------------------------------------
// social_network.h
//---------------------------------------------------------------------------------------------------------------------
#ifndef SOCIAL_NETWORK_HPP_
#define SOCIAL_NETWORK_HPP_
#include "user.hpp"
#include <vector>

namespace SocialNetworkNS{
    class SocialNetwork{
        private:
            ///Social network users
            std::vector<User> users;
            ///Friends of user[i]
            std::vector<std::vector<size_t>> friends;
            // Return the user index in the users vector, users.size() if the user is not found
            size_t CUserIndex(const std::string & name, const std::string & surname) const;

        public:
            //Return the set of users
            const std::vector<User> CGetUsers() const;
            // Get the friends of a user: @param user is the user, @return the set of his/her friends
            const std::vector<User> CGetFriends(const User & user) const;
            // Get the friends of a user: @param name is the name of the user,
            // @param surname is the surname of the user, @return the set of his/her friends
            const std::vector<User> CGetFriends(const std::string & name, const std::string  & surname) const;
            // Add a new user to the social network: @param name is the name of the user,
            // @param surname is the surname of the user
            void AddUser(const std::string & name, const std::string  & surname);
            // Add a friendship between two users: @param first_name is the name of the first user,
            // @param first_surname is the surname of the first user, @param second_name is the name of the second user
            // @param second_surname is the surname of the second user
            void AddFriendship(const std::string & first_name, const std::string & first_surname,
                               const std::string & second_name, const std::string & second_surname);
    };
}

#endif /*SOCIAL_NETWORK_HPP_*/
```

```cpp
//---------------------------------------------------------------------------------------------------------------------
// social_network.cpp
//---------------------------------------------------------------------------------------------------------------------
#include "social_network.hpp"
#include "user.hpp"
#include <iostream>

namespace SocialNetworkNS{
    size_t SocialNetwork::CUserIndex(const std::string & name, const std::string & surname) const{
        size_t i;
        User u(name, surname);
        bool found =false;
        for (i  = 0; i <users.size() && !found; ++i)
            if (users[i] == u)
                found = true;
        if (found)
            return --i;
        else
            return users.size();
    }

    void SocialNetwork::AddUser(const std::string & name, const std::string & surname){
        size_t user_index = CUserIndex(name, surname);
        if (user_index != users.size())
            return;
        User user(name, surname);
        users.push_back(user);
        friends.push_back({});
    }

    const std::vector<User> SocialNetwork::CGetUsers() const{
        return users;
    }

    const std::vector<User> SocialNetwork::CGetFriends(const User & user) const{
        return CGetFriends(user.CGetName(),user.CGetSurname());
    }

    const std::vector<User> SocialNetwork::CGetFriends(const std::string & name, const std::string  & surname) const{
        std::vector<User> ret{};
        // user index
        size_t i = CUserIndex(name,surname);
        if (i < users.size()){
            for (size_t j : friends[i])
                ret.push_back(users[j]);
        }
        return ret;
    }

    void SocialNetwork::AddFriendship(const std::string & first_name, const std::string & first_surname,
                                      const std::string & second_name, const std::string & second_surname){
        const User first_user = User(first_name, first_surname);
        const User second_user = User(second_name, second_surname);
        // check if the two users exist
        size_t first_user_index = CUserIndex(first_name,first_surname);
        size_t second_user_index = CUserIndex(second_name,second_surname);
        if (first_user_index == users.size() || second_user_index == users.size())
            return; // do nothing: one user at least does not exist
        if (first_user_index == second_user_index)
            return; // do nothing: one user cannot be friend of her/himself
        // check if friendship already exists, in this case do nothing
        bool friend_found = false;
        for (size_t i = 0; i<friends[first_user_index].size() && !friend_found; ++i)
            if (friends[first_user_index][i] == second_user_index)
                friend_found = true;
        if (!friend_found){ // add mutual friendship
            friends[first_user_index].push_back(second_user_index);
            friends[second_user_index].push_back(first_user_index);
        }
    }
}
```

first we need to check if the user already exists, if so we just end without adding anything

we want to be sure that the elements "users" and "friends" are coherent

They have to already exist and they have to be different

```cpp
//------------------------------------------------------------------------------------------------------------------------
// user.h
//------------------------------------------------------------------------------------------------------------------------
#ifndef USER_HPP_
#define USER_HPP_
#include <string>

namespace SocialNetworkNS{
    class User{
        private:
            // The name
            const std::string name;
            // The surname
            const std::string surname;

        public:
            // Constructor: @param name is the name of the user, @param surname is the surname of the user
            User(const std::string & name, const std::string & surname);
            // Return the name
            const std::string & CGetName() const;
            // Return the surname
            const std::string & CGetSurname() const;
            // Return the name and surname of the user
            std::string ToString() const;
    };
    // Comparison operator
    bool operator == (const User & lhs, const User & rhs);
}

#endif /*USER_HPP_*/


//------------------------------------------------------------------------------------------------------------------------
// user.cpp
//------------------------------------------------------------------------------------------------------------------------
#include "user.hpp"

namespace SocialNetworkNS{
    User::User(const std::string & _name, const std::string & _surname) : name(_name), surname(_surname) {}

    const std::string & User::CGetName() const {
        return name;
    }

    const std::string & User::CGetSurname() const{
        return surname;
    }

    std::string User::ToString() const{
        return name + " " + surname;
    }

    bool operator == (const User & lhs, const User & rhs){
        return lhs.ToString() == rhs.ToString();
    }
}
```

subintervals and, in any of them, we compute the minimum through solve_multistart. In particular (see lines 68 to 75), we compute the length of any subinterval (namely internal_step) as:

$$\frac{sup\_limit - inf\_limit}{n\_intervals}$$

and then, at every iteration of the for loop, we instantiate a new object of type FunctionMin, defining as new interval $[internal\_inf\_limit, internal\_inf\_limit + internal\_step]$ (where, of course, the first value given to internal_inf_limit is exactly inf_limit).

## Exercise 5 Gradient Descent in $\mathbb{R}^n$

Starting from the implementation of the gradient descent seen in Exercise 4, extend it to functions $f : \mathbb{R}^n \to \mathbb{R}$.

## Exercise 5 - Solution

First of all, we implement a class that defines a point in $\mathbb{R}^n$. It stores a vector of **doubles** which represent the coordinates of the point and it owns the following methods:

- **double** distance (**const** Point &p) **const**; computes the Euclidean distance between **this** and the point p.
- std::size_t get_n_dimensions (**void**) **const**; returns the size of the vector of coordinates, which of course corresponds to the dimension of the space.
- **double** get_coord (std::size_t i) **const**; returns the coordinate at index i.
- **void** set_coord (std::size_t i, **double** val); sets to val the value of the coordinate at index i.
- coords_type get_coords (**void**) **const**; returns the whole vector of coordinates of the point.
- **double** euclidean_norm (**void**) **const**; computes the $||.||_2$ norm of the vector of coordinates (as the distance between the point and the origin of the space, which is of course a vector of zeros).
- **double** infinity_norm (**void**) **const**; computes the infinity norm of the vector of coordinates.

All the methods except for set_coord are declared as **const** because they do not modify the point.
The implementation is reported in the file "Point.cpp".

********** file Point.h **********

```
1  #ifndef POINT_H_
2  #define POINT_H_
3
4  #include <vector>
5
6  class Point
7  {
8    typedef std::vector<double> coords_type;
9
```

```
protected:
  coords_type x;

public:
  explicit Point (const coords_type & coords): x (coords) {};

  //compute distance to Point p
  double distance (const Point & p) const;

  void print (void) const;

  std::size_t get_n_dimensions (void) const;
  double get_coord (std::size_t i) const;
  void set_coord (std::size_t i, double val);
  coords_type get_coords (void) const;

  double euclidean_norm (void) const;
  double infinity_norm (void) const;
};

#endif /* POINT_H_ */
```

********** file Point.cpp **********

```
#include <cmath>
#include <iostream>

#include "Point.h"

double Point::distance (const Point & p) const
{
  double dist = 0.0;

  for (std::size_t i = 0; i < x.size (); ++i)
    {
      const double delta = x[i] - p.x[i];
      dist += delta * delta;
    }

  return sqrt (dist);
}

void Point::print (void) const
{
  for (auto it = x.begin (); it != x.end (); ++it)
    {
      std::cout << *it;
      std::cout << " ";
    }

  std::cout << std::endl;
}
```

this is more precise than "<"

using iterators

the iterator is a pointer, we need the content

more convenient than: vector<double>::const_iterator

```
30 double Point::get_coord (std::size_t i) const
31 {
32  return x[i];
33 }
34
35
36 void Point::set_coord (std::size_t i, double val)
37 {
38  x[i] = val;
39 }
40
41 Point::coords_type Point::get_coords (void) const
42 {
43  return x;
44 }
45
46 double Point::euclidean_norm (void) const
47 {
48  const std::vector<double> zero_vector (x.size (), 0.);
49  const Point origin (zero_vector);
50  return distance (origin);
51 }
52
53 double Point::infinity_norm (void) const
54 {
55  double max_value = std::abs (x[0]);
56
57  for (std::size_t i = 1; i < x.size (); ++i)
58    {
59      const double next = std::abs (x[i]);
60      if (next > max_value)
61        max_value = next;
62    }
63
64  return max_value;
65 }
66
67 std::size_t Point::get_n_dimensions (void) const
68 {
69  return x.size ();
70 }
```

In order to be able to define a function in $\mathbb{R}^n$, we create a class Monomial. A monomial is represented by a coefficient and a vector of **doubles** that stores the exponents of the variables. For example, the monomial $3x^2y^3z$ has coeff equal to 3 and it stores in powers $[2, 3, 1]$. The class provides the method eval, which returns the value of the monomial at a given point.

********** file Monomial.h **********

```
1 #ifndef MONOMIAL_H_
2 #define MONOMIAL_H_
3
4 #include <vector>
5
```

```
#include "Point.h"

class Monomial
{
  double coeff;
  std::vector<double> powers;

public:
  Monomial (double c, const std::vector<double>  & pows)
    : coeff (c), powers (pows) {};
  double eval (const Point  & P) const;
};

#endif /* MONOMIAL_H_ */
```

********** file Monomial.cpp **********

```
#include <cmath>

#include "Monomial.h"

double Monomial:: eval (Point const & P) const
{
  double value = 1;

  for (std::size_t dim = 0; dim < powers.size (); ++dim)
    value *= pow (P.get_coord (dim), powers[dim]);

  return coeff * value;
}
```

A function $f : \mathbb{R}^n \to \mathbb{R}$ can be seen as the sum of different monomials. Therefore, we store in the class FunctionRn a std::vector of Monomials.
The class is implementend following the same logic we had in one dimension (see the file "Function.h" in Exercise 3), namely it has a method to evaluate the function itself and a method to compute its first derivative. Here, however, the first derivative (which is not the full derivative, but the partial derivative with respect to a given coordinate) is not computed exactly as a new function, but it is evaluated directly at a given point using the centered finite differences formula. This is the reason why we define, as a **static constexpr**, the value h that we will use as increment.
**Note:** for a possible alternative, based on the exact computation of the derivative, see the end of the answer.

********** file FunctionRn.h **********

```
#ifndef FUNCTIONRN_H_
#define FUNCTIONRN_H_

#include <vector>

#include "Monomial.h"
#include "Point.h"

class FunctionRn
```

```
{
  std::vector<Monomial> monoms;

  static constexpr double h = 0.00001;

public:
  double eval (const Point & P) const;

  void addMonomial (const Monomial & m);

  //evaluate derivative wrt dim j in P
  double eval_deriv (std::size_t j,
                     const Point & P) const;
};

#endif /* FUNCTIONRN_H_ */
```

********** file FunctionRn.cpp **********

```
#include "FunctionRn.h"

double FunctionRn::eval (const Point & P) const
{
  double value = 0;

  for (std::size_t i = 0; i < monoms.size (); ++i)
    value += monoms[i].eval (P);

  return value;
}

void FunctionRn::addMonomial (const Monomial & m)
{
  monoms.push_back (m);
}

//evaluate derivative wrt dim j in P
double FunctionRn::eval_deriv (std::size_t j,
                               const Point & P) const
{
  Point P1 (P.get_coords ());
  Point P2 (P.get_coords ());
  P1.set_coord (j, P.get_coord (j) + h);
  P2.set_coord (j, P.get_coord (j) - h);
  return (eval (P1) - eval (P2)) / (2 * h);
}
```

The class FunctionMinRn implements all the methods useful for the computation of the minimum. It stores the function $f$, the tolerance useful for the convergence check, the maximum number of iterations we want to perform, the vector of inferior limits and the one of superior limits that we need to define the domain.
If the **static** variable debug is set to **true**, some additional information, useful for debugging, are printed out during the execution of the different methods. In particular, the two **private**

methods:

```
void debug_info (const std::string& s) const;
```

and

```
void debug_info (const std::string& s1, double val) const;
```

are defined exactly with this purpose. The first one (see lines 243 to 247 in the file "FunctionMinRn.cpp" for the implementation) prints the string passed as parameter, while the second one (see lines 249 to 253) prints both the string and the number.

********** file **FunctionMinRn.h** **********

```
1  #ifndef FUNCTIONMINRN_H_
2  #define FUNCTIONMINRN_H_
3
4  #include <string>
5  #include <vector>
6
7  #include "FunctionRn.h"
8  #include "Point.h"
9
10 class FunctionMinRn
11 {
12  static constexpr bool debug = true;
13
14  FunctionRn f;
15
16  double tolerance;
17  double step;
18  unsigned int max_iterations;
19
20  std::vector<double> inf_limits;
21
22  std::vector<double> sup_limits;
23
24  Point compute_gradient (const Point & P0) const;
25
26  void debug_info (const std::string& s) const;
27
28  void debug_info (const std::string& s1, double val) const;
29
30  // use P as starting point
31  Point solve (const Point & P) const;
32
33  // for multi start implementation
34  void next_inf_limit (std::vector<double> & cur_inf_limit,
35                       const std::vector<double> & internal_steps) const;
36
37 public:
38
39  FunctionMinRn (FunctionRn func, double tol, double s,
40                 unsigned int max_it,
41                 const std::vector<double> & inff_limits,
```

```
42                const std::vector<double> & supp_limits)
43     : f (func), tolerance (tol), step (s), max_iterations (max_it),
44       inf_limits (inff_limits), sup_limits (supp_limits) {};
45
46   // gradient descent
47   Point solve (void) const;
48
49   // gradient descent with multi-start
50   Point solve_multistart (unsigned int n_trials) const;
51
52   // gradient descent with multi-start and domain decomposition
53   Point solve_domain_decomposition (unsigned int n_intervals,
54                                     unsigned int n_trials) const;
55 };
56
57 #endif /* FUNCTIONMINRN_H_ */
```

The method FunctionMinRn::solve (const Point &P) (see lines 9 to 81 of the file "FunctionMinRn.cpp") computes the minimum of the function $f$ starting from the given point P. It instantiates with the coordinates of P an initial guess P0 (line 22) and it loops until the maximum number of iterations is reached or we achieve convergence (exactly as the method FunctionMin solve (double x_init) in Exercise 4). The main difference is that, in this case, in order to compute the new value of the minimum, we have to use the formula

$$x^{(i)} = x^{(i-1)} - step * \nabla f(x^{(i-1)}),$$

where the first derivative we had in Exercise 4 is replaced by the gradient of the function. Therefore, we have to compute the partial derivatives of $f$ with respect to all the coordinates and to use these values to update the minimum.
In order to do this, in lines 38 to 78 we loop over all the coordinates, we evaluate at the point P0 the partial derivative of $f$ with respect to the variable $x_j$ and we compute the $j-th$ coordinate of the new minimum at iteration $i$ as

$$x_j^{(i)} = x_j^{(i-1)} - step * \frac{\partial f}{\partial x_j}(x_j^{(i-1)})$$

As in the one dimensional case (see Exercise 4), we must check that the new value stays within the proper interval (i.e. it is included between inf_limits[j] and sup_limits[j]).
Once we computed new_x, we use it to update the $j-th$ coordinate of the point P0 (line 51), then we evaluate $f$ at the new point (line 60), we compute the gradient (line 64) and we check the convergence (lines 74 and 75). In particular, the variable converged is set to **true** if either the absolute value of the difference between the values of the function $f$ in the old and the new point stays below a given tolerance or the infinity norm of the gradient stays below the same tolerance.
As in the one dimensional case (see Exercise 4), the method FunctionMinRn::solve() (see lines 83 to 93) simply calls the method FunctionMinRn::solve (const Point &P) passing as parameter an initial point whose $j-th$ coordinate is the midpoint of the interval $[inf\_limits[j], sup\_limits[j]]$.
The method FunctionMinRn::solve_multistart (see lines 96 to 148) behaves exactly as in the one dimensional case. The main difference stays in the generation of the random points. Indeed, in this context we have a different interval of admissible values for any coordinate, therefore we cannot generate directly the random value within the proper interval. In line 99, a distribution of **double** is initialized so that all the values will be generated between 0 and 1; then, in line 108 (and at every iteration of the for loop of lines 132 to 145),

each coordinate $j$ of the point is computed traslating the value in $[0,1]$ into a value in $[inf\_limits[j], sup\_limits[j]]$.
The method FunctionMinRn::solve_domain_decomposition must take care of how to decompose the domain in a given number of subdomains.
Being n_intervals, passed as parameter to the function, the number of intervals in which every of the axis should be subdivided, we compute the number of subspaces in which we want to split the domain as n_intervals raised to the power inf_limits.size() (line 165), which corresponds to the number of dimensions of the space (for example, in two dimensions, inf_limits would store the lower bounds for the two variables $x$ and $y$; therefore, n_intervals =10 means that we want to divide the domain in 100 rectangles).
Having initialized the vector cur_inf_limit with the values of inf_limits, the superior limit of every interval is computed adding to cur_inf_limit[j] the relative internal step (line 175).
The computation of the minimum follows as in the one dimensional case, initializing a new FunctionMinRn object with the current vectors of inferior and superior limits and calling the method FunctionMinRn::solve_multistart (see lines 179 to 196).
At the end, the update of the current inferior limits must be performed (line 205). The private method FunctionMinRn::next_inf_limit (see lines 211 to 230), which receives as parameters the current vector of inferior limits and the vector of internal steps, is designed at this purpose. In particular, it loops over the current inferior limits and, at every iteration, it checks whether cur_inf_limit[j] + internal_steps[j] exceedes the dimension of the relative interval or not (see line 216). In the first case, the current inferior limit becomes equal to inf_limits[j]. Indeed, for example, in two dimensions, if $x \in [0,3]$, $y \in [0,4]$ and the internal step is equal to 1 for both the variables, when we reach the point $(3,0)$ we cannot update the $x-$coordinate adding the internal step, but we have to jump to the point $(0,1)$.
In the second case, in turn, cur_inf_limit[j] is updated adding the value of internal_step[j] (in the example above, it is what happens to the $y-$coordinate, which passes from 0 to 1).

********** file FunctionMinRn.cpp **********

```
1  #include <cmath>
2  #include <iostream>
3  #include <random>
4
5  #include "FunctionMinRn.h"
6  #include "FunctionRn.h"
7  #include "Monomial.h"
8
9  Point FunctionMinRn::solve (const Point & P) const
10 {
11  if (debug)
12    {
13      for (double ii: inf_limits)
14        std::cout << ii << " ";
15      std::cout << std::endl;
16
17      for (double ss: sup_limits)
18        std::cout << ss << " ";
19      std::cout << std::endl;
20    }
21
22  Point P0 (P.get_coords());
23  double f0 = f.eval (P0);
24  bool converged = false;
```

```
25
26  if (debug)
27    std::cout << "Starting gradient" << std::endl;
28
29  for (unsigned int iter = 0;
30       iter < max_iterations && ! converged; ++iter)
31    {
32      if (debug)
33        {
34          std::cout << "P0: ";
35          P0.print ();
36        }
37
38      for (std::size_t j = 0; j < P0.get_n_dimensions (); ++j)
39        {
40          const double grad_j = f.eval_deriv (j, P0);
41          debug_info ("grad_j", grad_j);
42          double new_x = P0.get_coord (j) - grad_j * step;
43          debug_info ("new_x", new_x);
44
45          if (grad_j > 0)
46            new_x = std::max (inf_limits[j], new_x);
47          else
48            new_x = std::min (sup_limits[j], new_x);
49
50          debug_info ("new_x", new_x);
51          P0.set_coord (j, new_x);
52
53          if (debug)
54            {
55              P0.print();
56            }
57        }
58
59      //compute function in the new point
60      const double f1 = f.eval (P0);
61      debug_info ("f1", f1);
62
63      //update gradient in the new point
64      const Point grad = compute_gradient (P0);
65
66      if (debug)
67        {
68          std::cout << "grad" << std::endl;
69          grad.print ();
70        }
71
72      debug_info ("delta f", std::abs (f1 - f0));
73      debug_info ("infinity_norm", grad.infinity_norm ());
74      converged = (std::abs (f1 - f0) < tolerance)
75        || (grad.infinity_norm () < tolerance);
76
```

```
77      f0 = f1;
78    }
79
80  return P0;
81 }
82
83 Point FunctionMinRn::solve (void) const
84 {
85  std::vector<double> initial_coords;
86
87  //compute domain mid point
88  for (std::size_t i = 0; i < sup_limits.size (); ++i)
89   initial_coords.push_back ((sup_limits[i] + inf_limits[i]) / 2);
90  const Point P (initial_coords);
91
92  return solve (P);
93 }
94
95 // gradient descent with multi-start
96 Point FunctionMinRn::solve_multistart (unsigned int n_trials) const
97 {
98  std::default_random_engine generator;
99  std::uniform_real_distribution<double> distribution (0, 1);
100 std::vector<double> random_coords;
101 debug_info ("Running multi-start");
102
103 //generate random coords
104 for (std::size_t i = 0; i < inf_limits.size (); ++i)
105   {
106     debug_info ("Pick random value");
107     const double rand_val = distribution (generator);
108     random_coords.push_back (inf_limits[i] +
109                              (sup_limits[i] - inf_limits[i])
110                              * rand_val);
111   }
112
113 if (debug)
114   {
115     for (double rc: random_coords)
116       std::cout << rc << " ";
117     std::cout << std::endl;
118   }
119
120 debug_info ("Creating random point");
121 Point random_point = Point (random_coords);
122
123 if (debug)
124   {
125     std::cout << "First random point" << std::endl;
126     random_point.print ();
127   }
128
```

```
129   Point p_min = solve (random_point);
130   debug_info ("First random point val", f.eval (p_min));
131
132   for (unsigned n = 1; n < n_trials; ++n)
133     {
134       //generate random coords
135       for (std::size_t i = 0; i < inf_limits.size (); ++i)
136         random_coords[i] = inf_limits[i] +
137           (sup_limits[i] - inf_limits[i]) * distribution (generator);
138
139       random_point = Point (random_coords);
140       const Point p_new = solve (random_point);
141       debug_info ("Local optimum found: ", f.eval (p_new));
142
143       if (f.eval (p_new) < f.eval (p_min))
144         p_min = p_new;
145     }
146
147   return p_min;
148 }
149
150 // gradient descent with multi-start and domain decomposition
151 Point FunctionMinRn::solve_domain_decomposition (unsigned int n_intervals,
152                                                  unsigned int n_trials) const
153 {
154   std::vector<double> internal_steps;
155   // compute steps along each axis
156   for (std::size_t i = 0; i < inf_limits.size(); ++i)
157     internal_steps.push_back ((sup_limits[i] - inf_limits[i]) / n_intervals);
158
159   debug_info ("Temp minimum");
160   Point p_min = Point (inf_limits);
161
162   if (debug)
163     p_min.print ();
164
165   const unsigned int n_subspaces = pow (n_intervals, inf_limits.size ());
166   debug_info ("N subspaces :", n_subspaces);
167
168   std::vector<double> cur_inf_limit (inf_limits);
169   for (unsigned int i = 1; i <= n_subspaces; ++i)
170     {
171       // compute cur_sup_limits
172       debug_info ("compute cur_sup_limits");
173       std::vector<double> cur_sup_limit;
174       for (std::size_t j = 0; j < cur_inf_limit.size (); ++j)
175         cur_sup_limit.push_back (cur_inf_limit[j] + internal_steps[j]);
176
177       debug_info ("Create local solver");
178       //That's inefficient!! Think how to improve this!
179       FunctionMinRn minf_int (f,  tolerance,  step, max_iterations,
180                               cur_inf_limit, cur_sup_limit);
```

```
181
182     if (debug)
183       {
184         std::cout << "Inf intervals" << std::endl;
185         for (double elem: cur_inf_limit)
186           std::cout << elem << " ";
187         std::cout << std::endl;
188
189         std::cout << "Sup intervals" << std::endl;
190         for (double elem: cur_sup_limit)
191           std::cout << elem << " ";
192         std::cout << std::endl;
193       }
194
195     debug_info ("Compute new minimum");
196     const Point p_iter = minf_int.solve_multistart (n_trials);
197
198     if (debug)
199       p_iter.print ();
200
201     if (f.eval (p_iter) < f.eval (p_min))
202       p_min = p_iter;
203
204     debug_info ("Compute next sub-interval");
205     next_inf_limit (cur_inf_limit, internal_steps);
206   }
207
208  return p_min;
209 }
210
211 void FunctionMinRn::next_inf_limit (std::vector<double> & cur_inf_limit,
212                                     const std::vector<double> & internal_steps) const
213 {
214  for (std::size_t j = 0; j < cur_inf_limit.size (); ++j)
215    {
216      if (cur_inf_limit[j] + internal_steps[j] >= sup_limits[j])
217        {
218          cur_inf_limit[j] = inf_limits[j];
219          debug_info ("Updating dimension: ", j);
220          debug_info ("New inf limit: ", cur_inf_limit[j]);
221        }
222      else
223        {
224          cur_inf_limit[j] += internal_steps[j];
225          debug_info ("Updating dimension: ", j);
226          debug_info ("New inf limit: ", cur_inf_limit[j]);
227          return;
228        }
229    }
230 }
231
232
```

```
233 Point FunctionMinRn::compute_gradient (const Point & P0) const
234 {
235   std::vector<double> grad;
236
237   for (std::size_t j = 0; j < P0.get_n_dimensions (); ++j)
238     grad.push_back (f.eval_deriv (j, P0));
239
240   return Point (grad);
241 }
242
243 void FunctionMinRn::debug_info (const std::string& s) const
244 {
245   if (debug)
246     std::cout << s << " " << std::endl;
247 }
248
249 void FunctionMinRn::debug_info (const std::string& s, double val) const
250 {
251   if (debug)
252     std::cout << s << " " << val << std::endl;
253 }
```

********** file main.cpp **********

```
1  #include <iostream>
2  #include <vector>
3
4  #include "FunctionMinRn.h"
5
6  int main()
7  {
8    const double step = 0.01;
9    const double tolerance = 0.00001;
10
11   std::vector<Monomial> terms;
12   terms.push_back (Monomial (2, {2, 0}));
13   terms.push_back (Monomial (2, {1, 1}));
14   terms.push_back (Monomial (2, {0, 2}));
15   terms.push_back (Monomial (-2, {0, 1}));
16   terms.push_back (Monomial (6, {0, 0}));
17
18   FunctionRn f;
19   for (const Monomial & m: terms)
20     f.addMonomial (m);
21
22   const Point P1 ({0, 0});
23   const Point P2 ({2, 2});
24
25   std::cout << "Initial Points values" << std::endl;
26   std::cout << f.eval (P1) << " " << f.eval (P2) << std::endl;
27
28   const unsigned int max_iterations = 2000;
29   FunctionMinRn minRn (f, tolerance, step, max_iterations, {-5, -4}, {7, 9});
```

```
30
31  const Point P = minRn.solve ();
32  std::cout << "Final solution standard grad: "
33          << f.eval (P) << std::endl;
34  P.print ();
35
36  const Point Q = minRn.solve_multistart (1000);
37  std::cout << "Final solution multi-start:"
38          << f.eval (Q) << std::endl;
39  Q.print ();
40
41  const unsigned int n_domain_steps = 100;
42  const Point R = minRn.solve_domain_decomposition (n_domain_steps, 100);
43  std::cout << "Final solution domain decomposition:"
44          << f.eval (R) << std::endl;
45  R.print ();
46
47  return 0;
48 }
```

**********

**Possible alternative:** a possible alternative to the approach used in the class FunctionRn would be to define a method that computes exactly the partial derivative of the function $f$ with respect to a given coordinate, instead of approximating it. The simplest way to do this is to implement the corresponding method within the class Monomial, writing for example

```
1  Monomial Monomial::partial_derivative (std::size_t j) const
2  {
3     double dcoeff;
4     std::vector<double> dpowers = powers;
5
6     dcoeff = coeff * powers[j];
7
8     --dpowers[j];
9
10    return Monomial(dcoeff,dpowers);
11 }
```

and then to implement the partial derivative of $f$ and its gradient on top of this.

```
1  // compute the partial derivative of the function WRT the variable with index j
2  FunctionRn FunctionRn::partial_derivative (std::size_t j) const
3  {
4     std::vector<Monomial> dmonoms;
5
6     for (size_t k=0; k<monoms.size(); k++)
7        dmonoms.push_back(monoms[k].partial_derivative(j));
8
9     FunctionRn der;
10
11    for (const Monomial &dm : dmonoms)
12       der.addMonomial(dm);
13
```

```
14      return der;
15  }
16
17  // return the value of the gradient at the point p
18  Point FunctionRn::gradient_eval (const Point & p) const
19  {
20      std::vector<double> gradient;
21
22      for (size_t j=0; j<p.get_dimensions(); j++)
23      {
24          double pd = this->partial_derivative(j).eval(p);
25          gradient.push_back(pd);
26      }
27
28      return Point(gradient);
29  }
```

Of course, the class FunctionMinRn must be modified accordingly.

## 2.3 Challenges Solutions

### Date Class

[4]The aim of the challenge is to implement a class that represents a date. In particular, this class must give the possibility to define a new date, providing the day, the month and the year, and to increment a date adding or subtracting a certain number of days.
The implementation must take care of possible errors in defining a new date (namely the insertion of an invalid value for the day or the month) and it must be sure that, when a date is updated, the new value is still valid.
Furthermore, the class must implement the two following methods:

```
static Date next_Sunday (const Date &d);
```

computes, given a date `d`, the date of the first Sunday that comes after `d`.

```
static Date next_Weekday (const Date &d);
```

returns, in turn, the next weekday starting from the given date `d` (consider that, if `d` is Friday or Saturday, the next weekday is Monday, while it is the following day in all the other cases).
The computation of the next Sunday and the next weekday is based on the Doomsday algorithm. There are some days, called Doomsday days, that occur, in different years, always in the same date (they are, for example, the 4th of April, the 6th of June, the 8th of August, the 10th of October and the 12th of December). Starting from these days, it is possible to define the week day, starting from a given date.
The Doomsday algorithm is composed by three steps; suppose that we want to compute the week day of a given date `d`:

1. first of all, we have to extract the century to whom `d` belongs (note that, for the purposes of the Doomsday algorithm, every century starts from '00 and ends with '99). This operation returns a number between 0 (which corresponds to Sunday) and 6 (which corresponds to Saturday), that is called *anchor day* (or century base day).

2. second, we have to compute what is called the Doomsday of the year. If, for example, the Doomsday of the year is Monday, all the dates that fell on a fixed day will fell on Monday (namely the 4th of April will be Monday, the 6th of June will be Monday, etc.).

[4]Implementation by Samuele Vianello and Andrea Farahat

minimize f(x0, x1, x2) = 3x0x1x2^2 + 2x0x2 + 5

```
|#########################################################|
|                       FunctionRn                        |
|#########################################################|
| - monomials[]                                           |
|---------------------------------------------------------|
|(+ FunctionRn())                  + eval_deriv(j, P)     |
| + eval(P)                                               | // evaluation in a point
| + addMonomial()                                         |
|---------------------------------------------------------|
```

```
|#########################################################|
|                       FunctionMin                       |
|#########################################################|
| - f                                                     |
| - int_limit[]                                           |
| - sup_limit[]                                           |
| - tolerance                                             |
| - step                                                  |
| - max_iterations                                        |
|---------------------------------------------------------|
| + FunctionMin(func,a,b,tol,s,max_it)                    |
| - solve(P)                                              |
| + solve()                                               |
| + solve_multistart(n_trials)                            |
| + solve_domain_decomposition(n_intervals, n_trials)     |
|---------------------------------------------------------|
```

- compute-gradient(P)
- next-inf-limit(cur_inf_limit[], steps[])

```
|#########################################################|
|                         Point                           |
|#########################################################|
| - x[]    (consider "#" = protected)                     | // coordinates of the point
|---------------------------------------------------------|
| + Point(coords)                                         |
| + get_n_dimensions()                                    | // to know in which space we are
| + get_coords()          + get_coord(i)                  |
| + set_coord(i,val)                                      |
| + euclidean_norm()                                      |
| + distance(P)                                           | // distance between thhis point and
the point P
| + infinity_norm()                                       |
|---------------------------------------------------------|
```

```
|#########################################################|
|                        Monomial                         |
|#########################################################|
| - coefficient                                           |
| - powers[]                                              |
|---------------------------------------------------------|
|(+ Monomial(coeffs, powers))                             |
| + eval(P)                                               | // evaluation in a point
|---------------------------------------------------------|
```

Employee

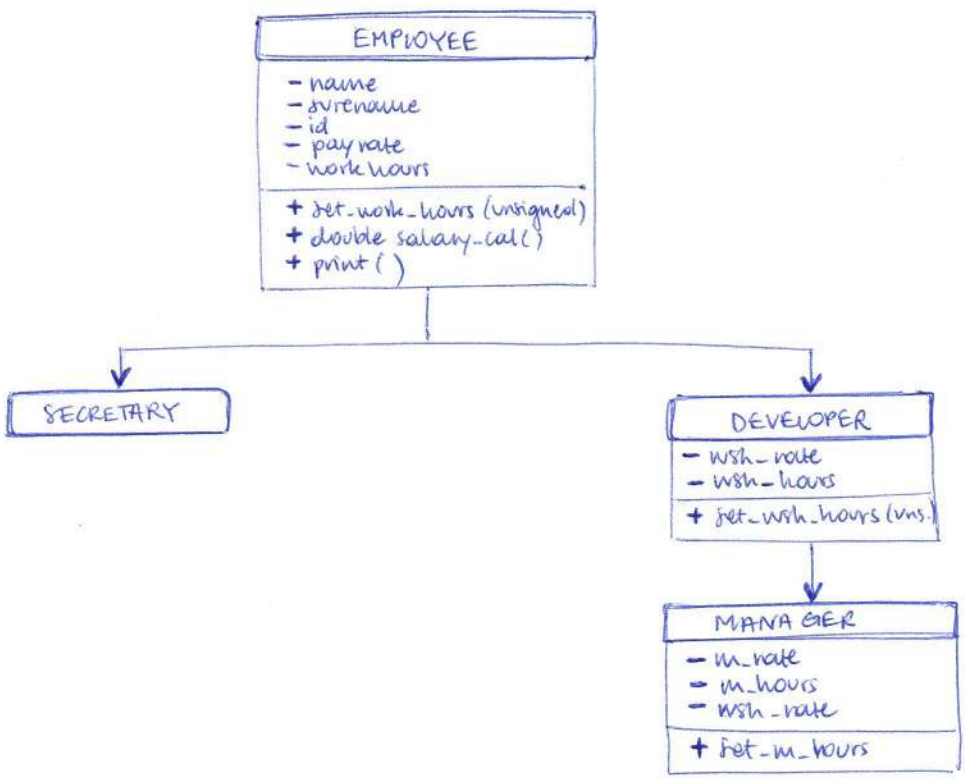
```cpp
//-----------------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "Secretary.hpp"
#include "Developer.hpp"
#include "Manager.hpp"

int main (void){
    Secretary S("Anna","Rossi",1);
    Developer D("Mauro","Neri",2);
    Manager M("Gaia","Bruni",3);

    std::cout << "Secretary:" << std::endl;
    S.print();
    S.set_work_hours(10);
    double Ssalary = S.salary_cal();
    std::cout << Ssalary << std::endl;

    std::cout << "Developer:" << std::endl;
    D.print();
    D.set_work_hours(10);
    D.set_wsh_hours(5);
    double Dsalary = D.salary_cal();
    std::cout << Dsalary << std::endl;

    std::cout << "Manager:" << std::endl;
    M.print();
    M.set_work_hours(10);
    M.set_wsh_hours(2);
    M.set_m_hours(3);
    double Msalary = M.salary_cal();
    std::cout << Msalary << std::endl;

    return 0;
}
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Employee.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <string>
#include <iostream>

class Employee {

    protected:
        std::string name;
        std::string surname;
        unsigned id;
        const double pay_rate = 7.5;
        unsigned work_hours;

    public:
        Employee (const std::string& n, const std::string& sn, unsigned id): name(n), surname(sn), id(id) {}
        virtual double salary_cal(void) const = 0;
        void set_work_hours (unsigned n) { work_hours = n; }
        void print (void) const{
            std::cout << name << " " << surname << " " << id << std::endl;
        }
};

#endif /* EMPLOYEE_H */
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Secretary.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef SECRETARY_H
#define SECRETARY_H
#include "Employee.hpp"

class Secretary: public Employee {

    public:
        Secretary (const std::string& n, const std::string& sn, unsigned id): Employee(n, sn, id) {}
        double salary_cal (void) const override {
            return (work_hours * pay_rate);
        }
};

#endif /* SECRETARY_H */
```

the constructor relies on the constructor of the superclass

```cpp
//-----------------------------------------------------------------------------
// Developer.hpp
//-----------------------------------------------------------------------------
#ifndef DEVELOPER_H
#define DEVELOPER_H
#include "Employee.hpp"

class Developer: public Employee {

    protected:
        const double wsh_rate = 8.0;   (workshop_rate)
        unsigned wsh_hours = 0;        (workshop_hours)

    public:
        Developer (const std::string& n, const std::string& sn, unsigned id): Employee(n, sn, id) {}
        double salary_cal (void) const override{
            return (work_hours - wsh_hours) * pay_rate + wsh_hours * wsh_rate;
        }
        void set_wsh_hours (unsigned n) {wsh_hours = n;}   <- a new setter since we introduce new members
};

#endif /* DEVELOPER_H */


//-----------------------------------------------------------------------------
// Manager.hpp
//-----------------------------------------------------------------------------
#ifndef MANAGER_H
#define MANAGER_H
#include "Developer.h"

class Manager: public Developer {

    protected:
        const double wsh_rate = 8.5;
        const double m_rate = 9.5;
        unsigned m_hours = 0;

    public:
        Manager (const std::string& n, const std::string& sn, unsigned id): Developer(n, sn, id) {}
        double salary_cal (void) const override{
            return (work_hours - wsh_hours - m_hours) * pay_rate + wsh_hours * wsh_rate + m_hours * m_rate;
        }
        void set_m_hours (unsigned n) {m_hours = n;}
};

 #endif /* MANAGER_H */
```



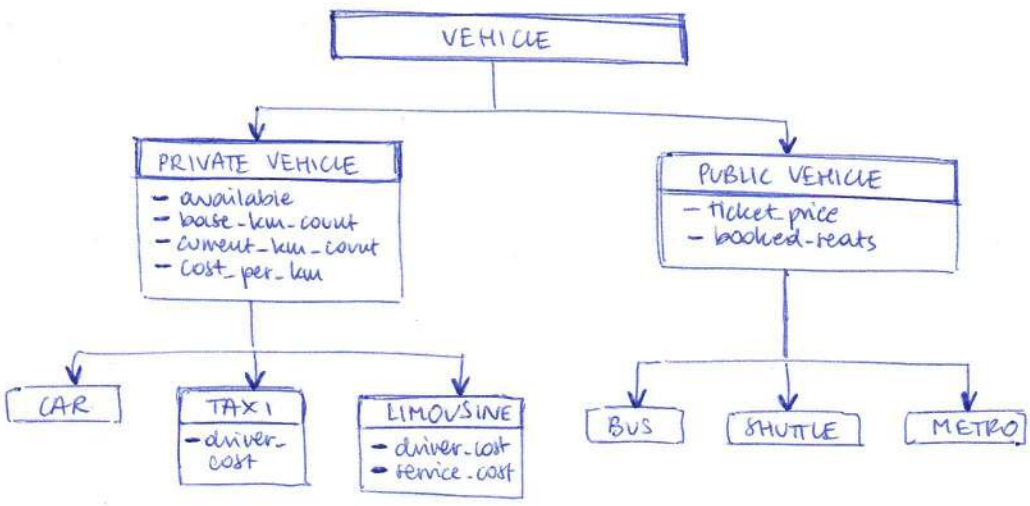
Travel System

```
//-----------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

#include "vehicle.h"
#include "public_vehicle.h"
#include "private_vehicle.h"
#include "car.h"
#include "taxi.h"
#include "limousine.h"
#include "shuttle.h"
#include "metro.h"
#include "bus.h"

typedef std::vector<travel_system::vehicle*> vehicles_vector; //shorter alias
std::vector<travel_system::vehicle*> private_vehicles; //stores private vehicles
std::vector<travel_system::vehicle*> public_vehicles; //stores public vehicles

// Prints the cost info from all vehicles in the vector
void print_cost_info(vehicles_vector vehicles, std::istream &is){
    int count = 0;
    for(travel_system::vehicle *v: vehicles){
        std::cout << "choice " << count << " -> type: " << v->get_identifier() << " ";
        v->cost_info();
        count++;
    }
}

// Allows the user to attempt to book one vehicle among those in the vector parameter after reading the requested
// number of seats. Returns: the index of the booked vehicle in the vector or -1 if booking failed
int book_vehicle(vehicles_vector vehicles, std::istream &is){
    //show the options to the traveller
    int count = 0;
    int num = 0;
    print_cost_info(vehicles, is);
    std::cout << "pick your choice or -1 for choosing another type of vehicle" << '\n';
    if(is >> count && count >= 0){
        std::cout << "enter the number of people" << '\n';
        is >> num;
        is.ignore(); //ignores the \n
        //calling the booking function from proper class through dynamic binding
        return vehicles[count]->booking(num) ? count : -1;
    }
    else
        return -1;
}

void read_from_stream (std::istream &is){
    //interacting with the traveller to get his/her choice
    std::string line;
    bool exit = false;
    while(line != "q" && ! exit){
        std::cout << "please choose your preference (private or public) or 'q' to quit" << '\n';
        getline(is, line);
        if (line == "private"){
            int vehicle_index = book_vehicle(private_vehicles, is);
            if(vehicle_index != -1){
                std::cout << "you have successfully booked a private vehicle" << std::endl;
                travel_system::private_vehicle *pv = dynamic_cast<travel_system::
                    private_vehicle *>(private_vehicles[vehicle_index]);
                for(int i = 0; i < 10; i++)
                    pv-> add_km();
                std::cout << "total cost: " << pv-> cost() << std::endl;
                pv-> finish();
                exit = true;
            }
        }
        else if(line == "public"){
            int vehicle_index = book_vehicle(public_vehicles, is);
            if(vehicle_index != -1){
                std::cout << "you have successfully booked a public vehicle" << std::endl;
                std::cout << "total cost: " << public_vehicles[vehicle_index]-> cost() << std::endl;
                public_vehicles[vehicle_index]-> finish();
                exit = true;
            }
        }
    }
}
```

! (dynamic cast?)

equivalent to: private_vehicles[vehicle_index]-> finish();

```
int main(int argc, char const *argv[]) {
    //create some vehicles
    travel_system::car c1 ("AB08609", 5, 0.35, "sedan");
    travel_system::car c2 ("AB08610", 2, 0.20, "smart");
    travel_system::taxi t1 ("AB08610", 4, 1.00, 2.00);
    travel_system::taxi t2 ("AB08611", 5, 1.00, 2.00);
    travel_system::limousine l1 ("AB08610", 10, 5.00, 50.0, 100);
    travel_system::limousine l2 ("AB08611", 8, 4.00, 70.0, 120);
    travel_system::shuttle sh1(10, 0, 15);
    travel_system::bus b1(40, 20, 3);//20 seats already occupied
    travel_system::metro m1(200, 50, 5);//50 seats already occupied
    //add them to the availability vectors
    private_vehicles.push_back(&c1);
    private_vehicles.push_back(&c2);
    //TODO: add the other vehicle objects to their respective vectors
    read_from_stream(std::cin);
    return 0;
}
```



```
//-----------------------------------------------------------------------------------------------------------------------
// vehicle.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef vehicle_hh
#define vehicle_hh
#include <string>

namespace travel_system{

   class vehicle{

        public:
            virtual bool booking (unsigned seats) = 0;
            virtual void cost_info (void) const = 0;
            virtual double cost (void) const = 0;
            virtual bool is_available (void) const = 0;
            virtual void finish (void) = 0;
            std::string get_identifier (void) const;
            virtual ~vehicle (void){};

        protected:
            vehicle(const std::string& s, unsigned c, unsigned p):
                identifier(s), capacity(c), curr_num_passangers(p){}
            std::string identifier;              //the vehicle type (car, taxi, bus, etc)
            unsigned capacity = 0;               //max number of passangers
            unsigned curr_num_passangers = 0;    //current number of pasangers
    };
}

#endif
```

protected constructor

```
//-----------------------------------------------------------------------------------------------------------------------
// vehicle.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "vehicle.h"

namespace travel_system{

        std::string vehicle::get_identifier (void) const{
        return identifier;
    }
}
```

```
//-----------------------------------------------------------------------------------------------------------------------
// private_vehicle.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef private_vehicle_hh
#define private_vehicle_hh
#include <string>
#include <iostream>
#include "vehicle.h"

namespace travel_system{

   class private_vehicle: public vehicle{

        public:
            virtual void cost_info (void) const override;
            virtual double cost (void) const override;
            bool booking (unsigned seats) override;
            void finish (void) override;
            void add_km (void);
            bool is_available (void) const override;
            virtual ~private_vehicle (void){};

        protected:
            private_vehicle (const std::string& s, unsigned c, std::string number, double cost):
                vehicle(s, c, 0), cost_per_km(cost), plate_number(number){}
            bool available = true;
            double current_km_counter = 0;
            double base_km_counter = 0;
            double const cost_per_km = 0;
            std::string plate_number;
    };
}

#endif
```

Notice that all of these "override" are **not** added in .cpp

protected constructor

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// private_vehicle.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "private_vehicle.h"

namespace travel_system{

    void private_vehicle::cost_info (void) const{
        std::cout << "cost per km: " << cost_per_km << '\n';
    }

    double private_vehicle::cost (void) const{
        return (current_km_counter - base_km_counter) * cost_per_km;
    }

    bool private_vehicle::is_available (void) const{
        return available;
    }

    // Adds a unit of km to the kilometer counter of the vehicle
    void private_vehicle::add_km (void){
        current_km_counter++;
    }

    bool private_vehicle::booking(unsigned seats){
        if (available and curr_num_passangers + seats <= capacity){
            curr_num_passangers += seats;
            base_km_counter = current_km_counter;
            available = false;
            return true;
        }
        else{
            std::cout << "booking not possible" << '\n';
            return false;
        }
    }

    void private_vehicle::finish (void){
        available = true;
        curr_num_passangers = 0;
    }
}


//-----------------------------------------------------------------------------------------------------------------------
// public_vehicle.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef public_vehicle_hh
#define public_vehicle_hh
#include <iostream>
#include <list>
#include "vehicle.h"

namespace travel_system{

    class public_vehicle: public vehicle{

        public:
            void cost_info (void) const override;
            double cost (void) const override;
            bool booking (unsigned seats) override;
            void finish (void) override;
            bool is_available (void) const override;
            virtual ~public_vehicle (void){};

        protected:
            public_vehicle(const std::string& s, unsigned c, unsigned p, double tp):
                vehicle(s, c, p), ticket_price(tp) {}
            std::list<std::string> time_schedule;
            double ticket_price;
            unsigned booked_seats = 0; //the number of booked seats
    };
}

#endif
```

protected constructor

```cpp
//------------------------------------------------------------------------------------------------------------------------
// public_vehicle.cpp
//------------------------------------------------------------------------------------------------------------------------
#include "public_vehicle.h"

namespace travel_system{

    bool public_vehicle::is_available (void) const{
        return (curr_num_passangers < capacity);
    }

    bool public_vehicle::booking (unsigned seats){
        if (curr_num_passangers + seats < capacity){
            booked_seats = seats;
            curr_num_passangers += booked_seats;
            return true;
        }
        else{
            std::cout << "Not enough seats" << '\n';
            return false;
        }
    }

    double public_vehicle::cost (void) const{
        return ticket_price * booked_seats;
    }

    // Prints the ticket price as the cost info for public vehicles
    void public_vehicle::cost_info (void) const{
        std::cout << "ticket price: " << ticket_price << std::endl;
    }

    void public_vehicle::finish (void){
        curr_num_passangers -= booked_seats;
        booked_seats = 0;
    }
}
```

seats that we're trying to book, **not** the overall number of booked seats

```cpp
//------------------------------------------------------------------------------------------------------------------------
// car.h
//------------------------------------------------------------------------------------------------------------------------
#ifndef car_hh
#define car_hh
#include <string>
#include <iostream>
#include "private_vehicle.h"

namespace travel_system{

    class car: public private_vehicle{

        std::string model;

        public:
            car (const std::string& number, unsigned c, double cost, const std::string& model_name):
                private_vehicle("car", c, number, cost), model(model_name){}
    };
}

#endif
```

public constructor

```cpp
//------------------------------------------------------------------------------------------------------------------------
// taxi.h
//------------------------------------------------------------------------------------------------------------------------
#ifndef taxi_hh
#define taxi_hh
#include <string>
#include <iostream>
#include "private_vehicle.h"

namespace travel_system{

    class taxi: public private_vehicle{

        double driver_cost;

        public:
            taxi (const std::string& number, unsigned c, double cost, double d_cost):
                private_vehicle("taxi", c, number, cost), driver_cost(d_cost){}
            double cost (void) const override;
            void cost_info (void) const override;
    };
}

#endif
```

public constructor

```
//-----------------------------------------------------------------------------------------------------------------------
// taxi.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "taxi.h"

namespace travel_system{

    double taxi::cost (void) const{
        return private_vehicle::cost() + driver_cost;
    }

    void taxi::cost_info (void) const{
        private_vehicle::cost_info();
        std::cout << " driver cost: " << driver_cost << std::endl;
    }
}


//-----------------------------------------------------------------------------------------------------------------------
// limousine.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef limousine_hh
#define limousine_hh
#include <string>
#include <iostream>
#include "private_vehicle.h"

namespace travel_system{

    class limousine: public private_vehicle{

        double driver_cost;
        double service_cost;

        public:
            limousine (const std::string& number, unsigned c, double cost, double d_cost, double s_const):
                private_vehicle("limousine", c, number, cost), driver_cost(d_cost), service_cost(s_const){}
            double cost (void) const override;
            void cost_info (void) const override;
    };
}

#endif


//-----------------------------------------------------------------------------------------------------------------------
// limousine.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "limousine.h"

namespace travel_system{

    double limousine::cost (void) const{
        return private_vehicle::cost() + driver_cost + service_cost;
    }

    void limousine::cost_info (void) const{
        private_vehicle::cost_info();
        std::cout << " driver cost: " << driver_cost << "\n service cost: " << service_cost << std::endl;
    }
}


//-----------------------------------------------------------------------------------------------------------------------
// bus.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef bus_hh
#define bus_hh
#include <iostream>
#include <list>
#include "public_vehicle.h"

namespace travel_system{

    class bus: public public_vehicle{

        public:
            bus(unsigned c, unsigned p, double tp): public_vehicle ("bus" , c, p, tp){}
    };
}

#endif
```

public constructor

public constructor

```
//-----------------------------------------------------------------------------------------------------------------
// metro.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef metro_hh
#define metro_hh
#include <iostream>
#include "public_vehicle.h"

namespace travel_system{

    class metro: public public_vehicle{

        public:
            metro(unsigned c, unsigned p, double tp): public_vehicle ("metro", c, p, tp){}
    };
}

#endif
```

public constructor

```
//-----------------------------------------------------------------------------------------------------------------
// shuttle.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef shuttle_hh
#define shuttle_hh
#include <iostream>
#include "public_vehicle.h"

namespace travel_system{

    class shuttle: public public_vehicle{

        public:
            shuttle(unsigned c, unsigned p, double tp): public_vehicle ("shuttle", c, p, tp){}
    };
}

#endif
```

public constructor

TDE

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Interpolation

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



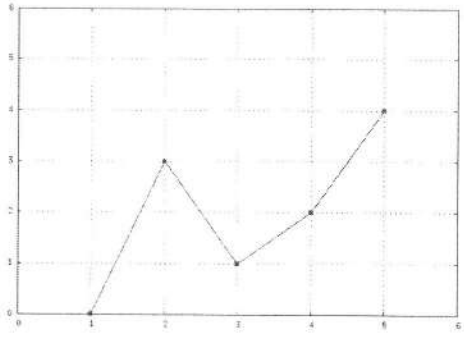
## Goal

- Implement a library that, given a vector of points, computes their **interpolation** in a **two-dimensional Euclidean space** with **three different methods.**
- The method `double interpolate(double x) const;` receives as input the `x` coordinate of a point and returns the corresponding `y` value, according to the chosen method. **If x does not belong to the domain**, the method should return `NaN`.
- **It is not allowed** to have an instance of class `Interpolation` without knowing which scheme it implements.



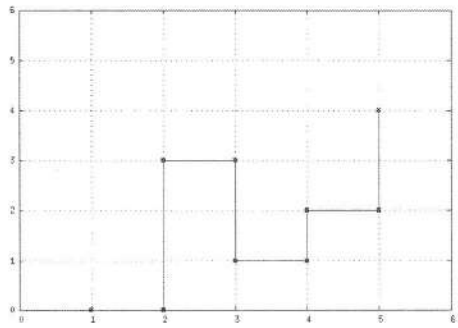
## Linear Interpolation

- Given a vector of points, e.g.,
  `{(1.0, 0.0), (2.0, 3.0), (3.0, 1.0), (4.0, 2.0), (5.0, 4.0)}`
  connects the points with a straight line.



# Stepwise Interpolation

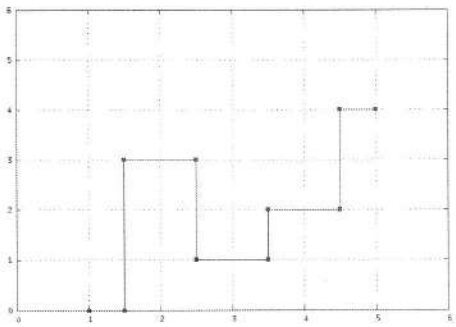
- Given a vector of points, e.g.,
  `{(1.0, 0.0), (2.0, 3.0), (3.0, 1.0), (4.0, 2.0), (5.0, 4.0)}`
  provides, within an interval bounded by subsequent points, the y value of the left extreme.





# Nearest-neighbor Interpolation

- Given a vector of points, e.g.,
  `{(1.0, 0.0), (2.0, 3.0), (3.0, 1.0), (4.0, 2.0), (5.0, 4.0)}`
  locates in the vector the nearest point and assigns the same value.





# Assumptions

- The class `Point` is defined as follows:

```
class Point {
  private:
     double x;
     double y;
     // ...
};
```

- The vector of `Points` in `Interpolation` is sorted according to the value of the x-axis coordinate.

```cpp
//-----------------------------------------------------------------------------
// Point.hpp
//-----------------------------------------------------------------------------
#ifndef POINT_HH
#define POINT_HH

class Point{

    double x;
    double y;

    public:
        Point (double f1, double f2);
        double get_x (void) const;
        double get_y (void) const;
};

#endif // POINT_HH


//-----------------------------------------------------------------------------
// Point.cpp
//-----------------------------------------------------------------------------
#include "Point.hpp"

Point::Point(double f1, double f2) : x (f1), y (f2) {}

double Point::get_x (void) const{
    return x;
}

double Point::get_y (void) const{
    return y;
}


//-----------------------------------------------------------------------------
// Interpolation.hpp
//-----------------------------------------------------------------------------
#ifndef INTERPOLATION_HH
#define INTERPOLATION_HH
#include <limits>
#include <vector>
#include "Point.hpp"

class Interpolation{

    protected:
        std::vector<Point> points; // sorted_vector
        constexpr static double err_val = std::numeric_limits<double>::quiet_NaN();

    public:
        virtual double interpolate (double) const = 0;
        bool range_check (double) const;
        Interpolation (const std::vector<Point> &);
        virtual ~Interpolation (void) = default;
};

#endif // INTERPOLATION_HH


//-----------------------------------------------------------------------------
// Interpolation.cpp
//-----------------------------------------------------------------------------
#include "Interpolation.hpp"
#include "Point.hpp"

bool Interpolation::range_check (double x) const{
    return not (x < points.front().get_x() or x > points.back().get_x());
}

Interpolation::Interpolation (const std::vector<Point> & points) : points (points) {}
```

err_val is a constant expression and its value can be substituted directly (when we compile) — this is quicker than accessing the value

this is how we define "NaN"

we want it to be pure virtual

every time we use inheritance we have to construct a virtual destructor

analog to: points[0].get_x()

points.front() return the first element of the vector of points (equivalent to points[0])

points.back() returns the last element of the vector of points

equivalently: points[points.size()-1].get_x()

! If not (condition) then we return "1" = True

```
//-----------------------------------------------------------------------------------------------------------------------
// LinearInterpolation.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef LINEAR_INTERPOLATION_HH
#define LINEAR_INTERPOLATION_HH
#include <vector>
#include "Interpolation.hpp"
#include "Point.hpp"

class LinearInterpolation : public Interpolation{

    public:
      ~~explicit~~ LinearInterpolation (const std::vector<Point> & points);
       virtual double interpolate (double x) const override;
};

#endif // LINEAR_INTERPOLATION_HH
```

without "override" we're redefining the method

```
//-----------------------------------------------------------------------------------------------------------------------
// LinearInterpolation.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "LinearInterpolation.hpp"
#include "Point.hpp"

LinearInterpolation::LinearInterpolation (const std::vector<Point> & points) : Interpolation (points) {}

double LinearInterpolation::interpolate (double x) const{
    double result (err_val);
    if (range_check (x)){
        std::vector<Point>::const_iterator previous = points.cbegin (),
        current = previous + 1;
        while (current != points.cend () and current -> get_x () < x){
            ++current;
            ++previous;
        }
        if (current != points.cend ()){
            const double x1 (previous -> get_x ()), x2 (current -> get_x ()),
                         y1 (previous -> get_y ()), y2 (current -> get_y ());
            result = y1 + (y2 - y1) * (x - x1) / (x2 - x1);
        }
    }
    return result;
}
```

(=) double result = err_val;

the class LinearInterpolation is delegating the constructor of the baseless to initialize the vector of points

another iterator (constant, same type as "previous") initialized with the second element of the vector

return a const iterator to the first element of the vector points

*current.get_x()

we loop until we reach the end of the vector points **or** until the x is greater than one of the x's in the vector of points (right-bound): (i.e.)

points = {(1,0), (2,3), (3,1), (4,2), (5,4)}
we have x = 3.5.

| | condition? |
|---|---|
| previous = 1, current = 2 ⇒ | ✓ |
| previous = 2, current = 3 ⇒ | ✓ |
| previous = 3, current = 4 ⇒ | × |

Now current is > 4 and so we exit the loop having :
previous = 3, current = 4 ✓

equivalent to:
x1 = previous -> get_x()

```
//-----------------------------------------------------------------------------------------------------------------------
// StepwiseInterpolation.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef STEPWISE_INTERPOLATION_HH
#define STEPWISE_INTERPOLATION_HH
#include <vector>
#include "Interpolation.hpp"
#include "Point.hpp"

class StepwiseInterpolation: public Interpolation{

    public:
        ~~explicit~~StepwiseInterpolation (const std::vector<Point> &);
        double interpolate (double x) const override;
};

#endif // STEPWISE_INTERPOLATION_HH
```

```
//-----------------------------------------------------------------------------------------------------------------------
// StepwiseInterpolation.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "StepwiseInterpolation.hpp"
#include "Point.hpp"

StepwiseInterpolation::StepwiseInterpolation (const std::vector<Point> & points) : Interpolation (points) {}

double StepwiseInterpolation::interpolate (double x) const{
    double result (err_val);
    if (range_check (x)){
        std::vector<Point>::const_iterator previous = points.cbegin (),
        current = previous + 1;
        while (current != points.cend () and current -> get_x () < x){
            ++current;
            ++previous;
        }
        if (current != points.cend ()){
            result = previous -> get_y ();
        }
    }
    return result;
}
```

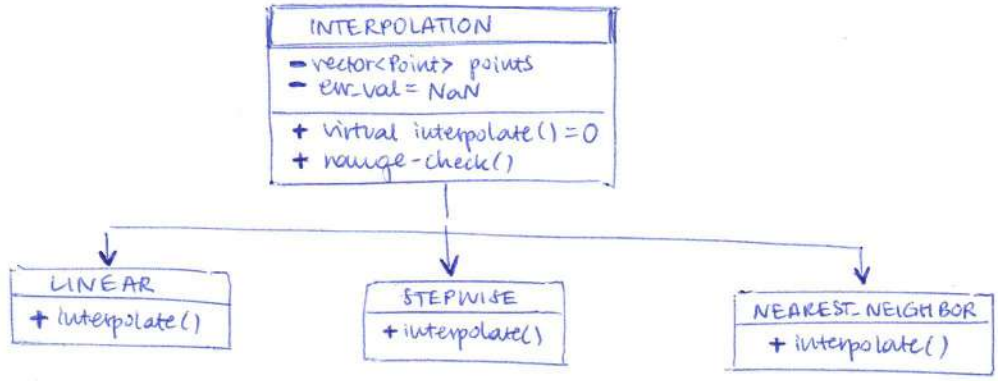
we change only this : once we find the subinterval we have to pass the y coordinate of the left bound of the subinterval

```cpp
//-----------------------------------------------------------------------------
// NearestNeighborInterpolation.hpp
//-----------------------------------------------------------------------------
#ifndef NEAREST_NEIGHBOR_INTERPOLATION_HH
#define NEAREST_NEIGHBOR_INTERPOLATION_HH
#include <vector>
#include "Interpolation.hpp"
#include "Point.hpp"

class NearestNeighborInterpolation: public Interpolation{

    public:
        explicit NearestNeighborInterpolation (const std::vector<Point> &);
        double interpolate (double) const override;
};

#endif // NEAREST_NEIGHBOR_INTERPOLATION_HH
```

```cpp
//-----------------------------------------------------------------------------
// NearestNeighborInterpolation.cpp
//-----------------------------------------------------------------------------
#include "NearestNeighborInterpolation.hpp"
#include "Point.hpp"

NearestNeighborInterpolation::NearestNeighborInterpolation(const std::vector<Point> & points) :
    Interpolation (points) {}

double NearestNeighborInterpolation::interpolate (double x) const{
    double result (err_val);
    if (range_check (x)){
        std::vector<Point>::const_iterator previous = points.cbegin (),
        current = previous + 1;
        while (current != points.cend () and current -> get_x () < x){
            ++current;
            ++previous;
        }
        if (current != points.cend ()){
            const double first_distance = x - previous -> get_x (),
            second_distance = current -> get_x () - x;
            const bool first_closest = second_distance > first_distance;
            result = first_closest ? previous -> get_y () : current -> get_y ();
        }
    }
    return result;
}
```

we change only this

first_closest = true ?
- **yes**: result = previous -> get_y()
- **no**: result = current -> get_y()




!!! Thanks to dynamic bounding:

```
LinearInterpolation method1(points);
cout << method1.interpolate(x);                    ✓
LinearInterpolation* p = &method1;
cout << p -> interpolate(x);                       ✓
Interpolation* q = &method1;
cout << q -> interpolate(x);                       ✓
```

In all the **3** cases we're interpolating with the linear interpolation, also in the last one!

```
//-----------------------------------------------------------------------------------------------------------------
// Smart Pointers
//-----------------------------------------------------------------------------------------------------------------

void f(){
    // We create a pointer to an integer and we initialize the integer to 5
    std::shared_ptr<int> p1 = std::make_shared<int>(5);          // counter p1=1;

    // Now we want a second pointer to point that 5
    // (we can use also: "std::shared_ptr<int> p2 = p1;")
    std::shared_ptr<int> p2(p1);                                // counter p1=2;

    // We create a new pointer pointing to an integer = 10:
    std::shared_ptr<int> p3 = std::make_shared<int>(10);         // counter p3=1;

    // Now p1 will point to 10
    p1 = p3;                                                    // counter p1=1; counter p3=2
}

// Since everything is in a function, when we exit the function all these pointers go out of scope.
// Going out of scope, the pointers are decreased. If they reach 0 they're deleted and the memory is freed.

// Notice that actually it's like we're counting how many pointers are pointing to an object.
// At the end of the function f() we can say that "the counter of 5 is 1 (because only p2 is pointing it),
// the counter of 10 is 2 (because both p1 and p3 are pointing it)". Once we go out of the function,  p1 goes
// out of scope and the counter of 10 decreases to 1. Then p2 goes out of scope, the counter of 5  decreases
// to 0 and so 5 is deleted. Then p3 goes out of scope, the counter of 10 decreases to 0 and so 10 is deleted.

//-----------------------------------------------------------------------------------------------------------------

std::shared_ptr<int> q1 = std::make_shared<int>(2);
std::shared_ptr<int> q2(q1);
std::shared_ptr<int> q3(q1);                        // counters q1, q2, q3 = 3

// The counter is OFFICIALLY part of to the pointers, not of the objects. However, every time we add a pointer
// pointing to an already pointed object, all the pointers that point that object have a +1 in their count.

// In this case, if we now do:
std::shared_ptr<int> q4 = std::make_shared<int>(2);
q3 = q4;
// we allocated a new memory cell and we initialized it with (again, but "different") 2.
// Since q3 now points the second 2, we have that all the counters are: q1, q2, q3, q4 : 2.
// (Both the 2 are pointed by two pointers)

//-----------------------------------------------------------------------------------------------------------------

// BASICALLY
// The counter of a pointer P is the number of pointers that are pointing to *P.
// (Every pointer is always aware of what's going on and it updates itself)
```

```
//-----------------------------------------------------------------------------------------------------------------------
// Shared Ownership - Smart Pointer (pt. 2)
//-----------------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "StrLPVector.h"

int main(){
    StrLPVector sv1({"a", "list", "of", "words"});
    StrLPVector sv2(sv1);

    std::cout << "sv1.front() is: " << sv1.front() << std::endl;
    std::cout << "sv2.front() is: " << sv2.front() << std::endl;

    StrLPVector sv3 = sv2;
    sv3.push_back("new element");

    std::cout << "sv1.back() is: " << sv1.back() << std::endl;
    std::cout << "sv2.back() is: " << sv2.back() << std::endl;
    std::cout << "sv3.back() is: " << sv3.back() << std::endl;
}


//-----------------------------------------------------------------------------------------------------------------------
// StrLPVector.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef STRLPVECTORS_STRLPVECTOR_H
#define STRLPVECTORS_STRLPVECTOR_H
#include <string>
#include <vector>
#include <memory>

class StrLPVector {
    public:
        typedef std::vector<std::string>::size_type size_type;

        // constructors
        StrLPVector (void);
        StrLPVector (std::initializer_list<std::string> il);

        size_type size (void) const {return data->size();}
        bool empty (void) const {return data->empty();}

        // add and remove elements
        void push_back (const std::string &t) {data->push_back(t);}
        void pop_back (void) {data->pop_back();}

        // element access
        std::string& front() {return data->front();}
        std::string& back() {return data->back();}

    private:
        std::shared_ptr<std::vector<std::string>> data;
};

#endif //STRLPVECTORS_STRLPVECTOR_H


//-----------------------------------------------------------------------------------------------------------------------
// StrLPVector.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "StrLPVector.h"

using std::make_shared;
using std::vector;
using std::string;

// constructors
StrLPVector::StrLPVector (void): data(make_shared<vector<string>>()) {}

StrLPVector::StrLPVector (std::initializer_list<std::string> il): data(make_shared<vector<string>>(il)) {}
```

TDE

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Matrix Computation

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO

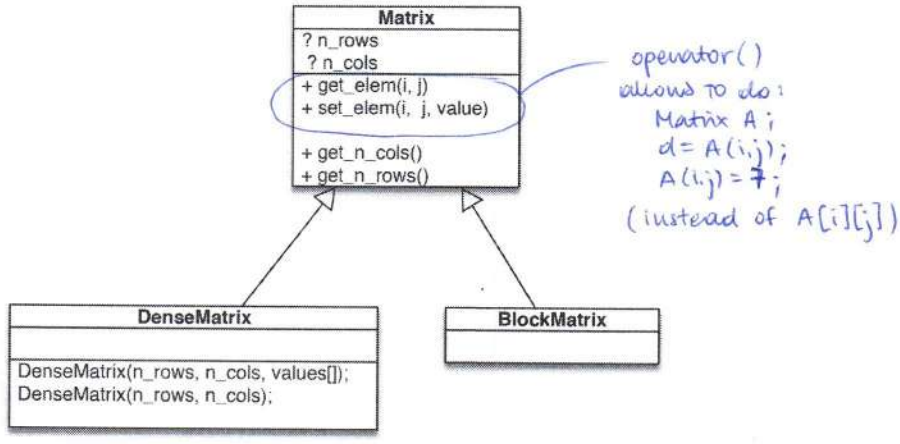


## Goal

- Implement a program for **matrix computation**, that manages both **dense and block matrices** of `double` precision numbers.
- **Dense matrices** are allocated when created.
- **Block matrices** are ideal to store **sparse** data and their size and memory storage are **changed dynamically** when blocks are added.

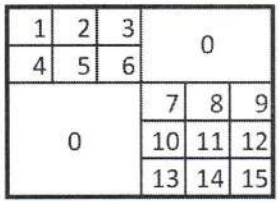


## Class hierarchy



# `Block` class — already implemented

- It includes a `DenseMatrix`
- It is characterized by the indexes of its top left and bottom right elements




- The values of a new block are provided per row in a single vector `vals` (e.g., {1,2,3,4,5,6}).



# Required methods

- **`add_block()`**, which receives a new block as parameter. The method should also update the matrix size accordingly.
- get-like implementation of **`operator()`**, which
  - receives as parameters the row and column indexes of the element to be read and returns its value,
  - prints an error message if the indexes are out of range,
  - returns 0 if the indexes are within the range, but for an element not explicitly initialized in a block.
- set-like implementation of **`operator()`**, which
  - receives as parameters the row and column indexes of an element and returns a reference to the element in the matrix,
  - adds a new block with a single element if the element was not previously allocated in a block.

Why do we implement A(i,j) instead of A[i][j]?
Because for A[i][j] we would need:

```
double& operator[] (std::size_t j)
std::vector<double>& operator[] (std::size_t i)
```

This is because of how the data is stored:

A[i][j] → data[i][j] → std::vector<std::vector<double>> data

it's a vector of vector, and so:

data[i] → std::vector<double>

still a vector, from here we take the j-th element

⟹ A(i,j) needs only one function (operator)
A[i][j] needs two operators
⟹ A(i,j) is more convenient

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// block.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef BLOCK_H_
#define BLOCK_H_
#include <vector>
#include "dense_matrix.hpp"

class Block{

    private:
        std::size_t top_left_row;
        std::size_t top_left_col;
        std::size_t bottom_right_row;
        std::size_t bottom_right_col;
        DenseMatrix dm;

    public:
        Block (std::size_t top_left_r, std::size_t top_left_c, std::size_t bottom_right_r, std::size_t bottom_right_c,
               const std::vector<double> & vals);
        std::size_t get_bottom_right_col() const;
        std::size_t get_bottom_right_row() const;
        std::size_t get_top_left_col() const;
        std::size_t get_top_left_row() const;
        double & operator () (std::size_t i, std::size_t j);
        double operator () (std::size_t i, std::size_t j) const;
};

#endif /* BLOCK_H_ */


//-----------------------------------------------------------------------------------------------------------------------
// block.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <vector>
#include <iostream>
#include "block.hpp"

Block::Block (std::size_t top_left_r, std::size_t top_left_c, std::size_t bottom_right_r,
              std::size_t bottom_right_c, const std::vector<double> & vals):
    top_left_row(top_left_r), top_left_col(top_left_c), bottom_right_row(bottom_right_r),
    bottom_right_col(bottom_right_c), dm(bottom_right_r-top_left_r+1, bottom_right_c-top_left_c+1, vals) {}

std::size_t Block::get_bottom_right_col() const{
    return bottom_right_col;
}

std::size_t Block::get_bottom_right_row() const{
    return bottom_right_row;
}

std::size_t Block::get_top_left_col() const{
    return top_left_col;
}

std::size_t Block::get_top_left_row() const{
    return top_left_row;
}

double & Block::operator()(std::size_t i, std::size_t j){
    return dm(i, j);
}

double Block::operator() (std::size_t i, std::size_t j) const{
    return dm(i, j);
}


//-----------------------------------------------------------------------------------------------------------------------
// matrix.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef MATRIX_H_
#define MATRIX_H_
#include <cstddef>     <-- for std::size_t

class Matrix{

        protected:
                std::size_t n_rows;
                std::size_t n_cols;

        public:
                Matrix (std::size_t rows, std::size_t cols);
                virtual double & operator () (std::size_t i, std::size_t j) = 0;
                virtual double operator () (std::size_t i, std::size_t j) const = 0;
                std::size_t get_n_cols() const;
                std::size_t get_n_rows() const;
                virtual ~Matrix (void) = default;   !!
};

#endif /* MATRIX_H_ */
```

pure virtual method ⟹ abstract class (we won't be able to create objects of this class)

```
//-----------------------------------------------------------------------------
// dense_matrix.hpp      (not given in the exam)
//-----------------------------------------------------------------------------
#ifndef DENSE_MATRIX_HPP
#define DENSE_MATRIX_HPP
#include <istream>
#include <vector>
#include "matrix.hpp"

class DenseMatrix : public Matrix{
        private:
                std::vector<double> m_data;
                std::size_t sub2ind (std::size_t i, std::size_t j) const;

        public:
                DenseMatrix (std::size_t rows, std::size_t columns,
                double value = 0.0);
                DenseMatrix (std::size_t rows, std::size_t columns,
                const std::vector<double>& values);
                void read (std::istream &);
                void swap (DenseMatrix &);
                double & operator () (std::size_t i, std::size_t j) override;
                double operator () (std::size_t i, std::size_t j) const override;
                DenseMatrix transposed (void) const;
                double * data (void);
                const double * data (void) const;
};

DenseMatrix operator * (const DenseMatrix &, const DenseMatrix &);

void swap (DenseMatrix &, DenseMatrix &);

#endif // DENSE_MATRIX_HH
```

| 1 | 2 | 3 |
|---|---|---|
| 4 | 5 | 6 |

is stored as :

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|

we create a matrix rows x columns where each element is "double 0.0"

```
//-----------------------------------------------------------------------------
// dense_matrix.cpp      (not given in the exam)
//-----------------------------------------------------------------------------
#include <sstream>
#include <string>
#include <iostream>
#include "dense_matrix.hpp"

DenseMatrix::DenseMatrix (std::size_t rows, std::size_t columns, const double & value) :
        Matrix(rows, columns), m_data (n_rows * n_cols, value){}

DenseMatrix::DenseMatrix(std::size_t rows, std::size_t columns, std::vector<double> values) :
        Matrix(rows, columns), m_data(values){}

std::size_t DenseMatrix::sub2ind (std::size_t i, std::size_t j) const{
        return i * n_cols + j;
}

void DenseMatrix::read (std::istream & in){
        std::string line;
        std::getline (in, line);
        std::istringstream first_line (line);
        first_line >> n_rows >> n_cols;
        m_data.resize (n_rows * n_cols);
        for (std::size_t i = 0; i < n_rows; ++i){
                std::getline (in, line);
                std::istringstream current_line (line);
                for (std::size_t j = 0; j < n_cols; ++j){
                        // alternative syntax: current_line >> operator () (i, j);
                        // or: current_line >> m_data[sub2ind (i, j)];
                        current_line >> (*this)(i, j);
                }
        }
}

void DenseMatrix::swap (DenseMatrix & rhs){
        using std::swap;
        swap (n_rows, rhs.n_rows);
        swap (n_cols, rhs.n_cols);
        swap (m_data, rhs.m_data);
}

std::size_t DenseMatrix::rows (void) const{
        return n_rows;
}

std::size_t DenseMatrix::columns (void) const{
        return n_cols;
}

DenseMatrix DenseMatrix::transposed (void) const{
        DenseMatrix At (n_cols, n_rows);
        for (std::size_t i = 0; i < n_cols; ++i)
                for (std::size_t j = 0; j < n_rows; ++j)
                        At(i, j) = operator () (j, i);
        return At;
}

double * DenseMatrix::data (void){
        return m_data.data ();
}

const double * DenseMatrix::data (void) const{
        return m_data.data ();
}

DenseMatrix operator * (DenseMatrix const & A, DenseMatrix const & B){
        DenseMatrix C (A.rows (), B.columns ());
        for (std::size_t i = 0; i < A.rows (); ++i)
                for (std::size_t j = 0; j < B.columns (); ++j)
                        for (std::size_t k = 0; k < A.columns (); ++k)
                                C(i, j) += A(i, k) * B(k, j);
        return C;
}

void swap (DenseMatrix & A, DenseMatrix & B){
        A.swap (B);
}

double & DenseMatrix::operator()(std::size_t i, std::size_t j){
        return m_data.at(sub2ind(i, j));
}

double DenseMatrix::operator()(std::size_t i, std::size_t j) const{
        return m_data.at(sub2ind(i, j));
}
```

```
//-----------------------------------------------------------------------------
// block_matrix.hpp
//-----------------------------------------------------------------------------
#ifndef BLOCKMATRIX_H_
#define BLOCKMATRIX_H_
#include <vector>
#include "block.hpp"

class BlockMatrix: public Matrix{

        std::vector<Block> blocks;
        bool indexes_in_range (std::size_t i, std::size_t j) const;
        bool indexes_in_block (std::size_t i, std::size_t j, const Block & block) const;

        public:
                BlockMatrix();
                void add_block (const Block & block);
                double & operator () (std::size_t i, std::size_t j) override;
                double operator () (std::size_t i, std::size_t j) const override;
};

#endif /* BLOCKMATRIX_H_ */
```

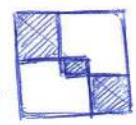
a BlockMatrix is a collection of blocks

this is for: $A(i,j) = 7$

this is for: $d = A(i,j)$

```
//-----------------------------------------------------------------------------
// block_matrix.cpp
//-----------------------------------------------------------------------------
#include <limits>
#include <iostream>
#include "block_matrix.hpp"

BlockMatrix::BlockMatrix() : Matrix (0, 0) {}

bool BlockMatrix::indexes_in_range (std::size_t i, std::size_t j) const{
        return (i < n_rows) && (j < n_cols);
}

bool BlockMatrix::indexes_in_block (std::size_t i, std::size_t j, const Block & block) const{
        return (i >= block.get_top_left_row()) && (i <= block.get_bottom_right_row()) &&
                   (j >= block.get_top_left_col()) && (j <= block.get_bottom_right_col());
}

double BlockMatrix::operator()(std::size_t i, std::size_t j) const{
        if (!indexes_in_range (i, j)){
                std::cerr << "indexes out of range!" << std::endl;
                return std::numeric_limits<double>::quiet_NaN();
        }
        for (const Block & block : blocks){
                // if i, j in block coordinates
                if (indexes_in_block (i, j, block))
                // access proper indexes element and return
                return block(i - block.get_top_left_row(), j - block.get_top_left_col());
        }
        return 0;
}

double & BlockMatrix::operator() (std::size_t i, std::size_t j){
        for (Block & block : blocks){
                // if i, j in block coordinates
                if (indexes_in_block (i, j, block)){
                        // change proper indexes element and return
                        return block(i - block.get_top_left_row(), j - block.get_top_left_col());
                }
        }
        // Create a single element block
        Block block (i, j, i, j, {0});
        add_block (block);
        return blocks.back()(0, 0);
}

void BlockMatrix::add_block (const Block & block){
        blocks.push_back (block);
        // Update matrix size
        n_rows = std::max (n_rows, block.get_bottom_right_row() + 1);
        n_cols = std::max (n_cols, block.get_bottom_right_col() + 1);
}
```

!! ≠



either the element is in one of the blocks or it's 0

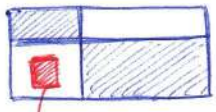
return the element (0,0) of the last block

! notice one thing:

```
double & BlockMatrix:: ..( ){
        double elem;
        ..
        return elem;
}
```

This is a HUGE error. We're returning a reference, if we return "elem" we're returning the reference of something that is destroyed as soon as we go out of scope!

they might not change:



we may add a block here

# Copy control

```
//-----------------------------------------------------------------------------
// Boxes - Version 1
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <vector>
#include "Box.h"

using std::cout;
using std::endl;

int main(){
    cout << endl;

    // first box
    Box b1(10,10,10);
    double v1 = b1.volume();
    cout << "volume of box 1 (10x10x10) is: " << v1 << endl;
    cout << "box id: " << b1.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    // second box
    Box b2(10,20,30);
    double v2 = b2.volume();
    cout << "volume of box 2 (10x20x30) is: " << v2 << endl;
    cout << "box id: " << b2.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    // third box
    Box b3(b1);
    double v3 = b3.volume();
    cout << "box 3 is equal to box 1" << endl;
    cout << "volume of box 3 is: " << v3 << endl;
    cout << "box id: " << b3.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    cout << "now box 3 is equal to box 1" << endl;
    b3 = b2;
    double v3_new = b3.volume();
    cout << "new volume of box 3 is: " << v3_new << endl;
    cout << "new box id: " << b3.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    return 0;
}
```

since getcount() is a **static** method we can call it like this **or** in the standard way (**b1.** getcount())

we can call it like this because, being static, they're related to the class and not to a specific object

b3 is a copy of b1

we assign to b3 the values of b2 (without changing b3's id!)

```
//-----------------------------------------------------------------------------
// Box.h
//-----------------------------------------------------------------------------
#ifndef BOXESV1_BOX_H
#define BOXESV1_BOX_H

#include <iostream>

class Box {

    public:
        // constructor
        Box (double l, double b, double h): length(l), breadth(b), height(h){
            std::cout << "Constructing a box" << std::endl;
            count++;
            id = count;
        }

        // copy constructor
        Box (const Box& b): length(b.length), breadth(b.breadth), height(b.height){
            std::cout << "Using copy constructor" << std::endl;
            count ++;
            id = count;
        }

        // member functions
        double volume (void) const;
        static unsigned getcount (void) {return count;}
        unsigned getid (void) const {return id;}

        // assignment operator
        Box& operator= (const Box &);

    private:
        double length;          // Length of a box
        double breadth;         // Breadth of a box
        double height;          // Height of a box
        unsigned id;            // Identification Number of the box
        static unsigned count;  // Number of boxes
};

#endif //BOXESV1_BOX_H

//-----------------------------------------------------------------------------
// Box.cpp
//-----------------------------------------------------------------------------
#include "Box.h"

// initialization of the static variable
unsigned Box::count = 0;

// member functions:
double Box::volume() const{
    return length * breadth * height;
}

// assignment operator
Box& Box::operator= (const Box &b){
    std::cout << "Using assignment operator" << std::endl;
    length = b.length;
    breadth = b.breadth;
    height = b.height;
    return *this;
}
```

this is used to get a new identification number each time we create an object

we copy all BUT the id, which must always be unique

(this must be unique for each box)

static members are not part of an object of the class, they're related to the class itself

if we define a static count we don't have an instance of count for each object that we create, but we have an unique instance of count that is shared by all the objects of type "Box" !

(necessarily in .cpp) !

!

We're not touching the id because all of the objects have their own. We want to assign the values of the elements without changing the id of the new created Box.

```
//------------------------------------------------------------------------------
// Boxes - Version 2
//------------------------------------------------------------------------------
// main.cpp
//------------------------------------------------------------------------------
#include <vector>
#include "Box.h"

using std::cout;
using std::endl;

int main(){
    cout << endl;

    // first box
    Box b1(10,10,10);
    double v1 = b1.volume();
    cout << "volume of box 1 (10x10x10) is: " << v1 << endl;
    cout << "box id: " << b1.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    // second box
    Box b2(10,20,30);
    double v2 = b2.volume();
    cout << "volume of box 2 (10x20x30) is: " << v2 << endl;
    cout << "box id: " << b2.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    // third box
    Box b3(b1);
    double v3 = b3.volume();
    cout << "box 3 is equal to box 1" << endl;
    cout << "volume of box 3 is: " << v3 << endl;
    cout << "box id: " << b3.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    cout << "now box 3 is equal to box 1" << endl;
    b3 = b2;
    double v3_new = b3.volume();
    cout << "new volume of box 3 is: " << v3_new << endl;
    cout << "new box id: " << b3.getid() << endl;
    cout << "count: " << Box::getcount() << endl;
    cout << endl;

    // initialize vector of boxes
    std::vector<Box> boxes = {b1, b2, b3};
    cout << "\nid of new boxes: " << endl;
    for (const Box& b : boxes)
        cout << b.getid() << endl;
    cout << "count: " << Box::getcount() << endl;

    return 0;
}
```

(WITHOUT ITEMIZED DESTRUCTOR)
here we expect it to be = 6
(since we have 3 boxes (b1, b2, b3) and then we created a vector of new 3 boxes and we performed the assignment)
However, here it prints 9. This is because every time we insert an object into a vector we use the copy constructor (which has a count++).
Here it's like: first we create a temporary vector of 3 new elements (so we create 3 unique id's) and then we perform the copy constructor copying b1, b2, b3 and introducing 3 new id's again.
Totally, this operation creates 6 new id's, 3 of which are lost.

This is something that we cannot avoid, however we have to be aware of this because if we ask how many boxes are there we receive "9" when there are only 6. (The missing are the temporaries)

MORE!
If we would have done:

```
for(Box b:boxes)
    cout << b.getid() << end;
```

we would have the final count = 12 (because in this way we procede by copying and every time we do the copy we have a count++)

(WITH ITEMIZED DESTRUCTOR)
Not true!
If we create something that decrements the counter every time we delete an object we'll fix it.
In the sense of: here we have 9 because we created 3 temporary boxes that then we "destroy" (don't use). If we find a way to decrement the counter it'll be ok!
→ we reduce the count in a destructor
(The destructor will reduce the counter every time that a temporary box is destroyed)

```
//-------------------------------------------------------------------------------------------------------------------
// Box.h
//-------------------------------------------------------------------------------------------------------------------
#ifndef BOXESV2_BOX_H
#define BOXESV2_BOX_H

#include <iostream>

class Box {

    public:
        // constructor
        Box (double l, double b, double h): length(l), breadth(b), height(h){
            std::cout << "Constructing a box" << std::endl;
            count++;
            id = count;
        }

        // copy constructor
        Box (const Box& b): length(b.length), breadth(b.breadth), height(b.height){
            std::cout << "Using copy constructor" << std::endl;
            count ++;
            id = count;
        }

        // member functions
        double volume (void) const;
        static unsigned getcount (void) {return count;}
        unsigned getid (void) const {return id;}

        // assignment operator
        Box& operator= (const Box &);

        // destructor
        ~Box() {count--;}

    private:
        double length;          // Length of a box
        double breadth;         // Breadth of a box
        double height;          // Height of a box
        unsigned id;            // Identification Number of the box
        static unsigned count;  // Number of boxes
};

#endif //BOXESV2_BOX_H
```

we introduce this so that the count counts the object that are "alive" inside the code, and not all the objects that have been created somewhere as temporary copies and then destroyed

(**Problem**: doing so, however, we create an issue with the id's. If we create 3 boxes, then other 3 (temporary), then other 3, then we destroy the temporary boxes → we have: counter = 6, id's = {1,2,3,7,8,9} and so the next 3 created boxes will have the id = {7,8,9}, which is a repetition. We lose the uniqueness, however for the moment we don't care)

```
//-------------------------------------------------------------------------------------------------------------------
// Box.cpp
//-------------------------------------------------------------------------------------------------------------------
#include "Box.h"

// initialization of the static variable
unsigned Box::count = 0;

// member functions:
double Box::volume() const{
    return length * breadth * height;
}

// assignment operator
Box& Box::operator= (const Box &b){
    std::cout << "Using assignment operator" << std::endl;
    length = b.length;
    breadth = b.breadth;
    height = b.height;
    return *this;
}
```

# Implicit Class-Type Conversions + Explicit

```
//---------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------

#include <iostream>
#include "MatlabVector.h"

using std::cout;
using std::endl;

int main() {
    MatlabVector v;
    v[0] = 1;
    v[1] = 3;
    cout << "v content" << endl;
    v.print();
    v[3] = 4;
    cout << "v content" << endl;
    v.print();

    double d = v[4];
    cout << "v content" << endl;
    v.print();

    for (unsigned i = 0; i < v.size(); ++i){
        v[i] = i;
        cout << v[i] << " ";
    }
    cout << endl;

    cout << "v content" << endl;
    v.print();

    MatlabVector v2 = v * 3; // unfortunately 3*v does not work
    cout << "v2 content" << endl;
    v2.print();
    MatlabVector v3 = v + v2;
    cout << "v3 content" << endl;
    v3.print();

    MatlabVector v4(10);
    cout << "v4 content" << endl;
    v4.print();

    for (unsigned j = 0; j < v4.size(); ++j)
        v4[j] = j;

    cout << "v4 new content" << endl;
    v4.print();

    MatlabVector v5 = v4 + 3;
    cout << "v5 content" << endl;
    v5.print();

    MatlabVector v6 = v4 + 10;
    cout << "v6 content" << endl;
    v6.print();

    return 0;
}
```

the operator "+" wants a constant reference to another MatlabVector, however here we have "3". Because of the implicit class type conversion, since the constructor of the MatlabVector class takes only one parameter, writing +3 is like writing "+ MatlabVector temporary(3)." Hence, we're doing a "+" between two objects of MatlabVector, however since their length differs we have a "NaN" as a cout !

Here we have the same reasoning, we're creating a new object of the class MatlabVector of size 10. The MatlabVector will be 10 times "0" and so v6 will be equivalent to v4.
Here the print will be:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
//---------------------------------------------------------------------------
// MatlabVector.h
//---------------------------------------------------------------------------
#ifndef MATLABVECTOR_MATLABLIKEVECTOR_H
#define MATLABVECTOR_MATLABLIKEVECTOR_H

#include <vector>
#include <iostream>
#include <limits>

class MatlabVector {
    std::vector<double> elem;

    public:
        MatlabVector (void) = default;
        MatlabVector (unsigned n): elem(n,0.) {}
        double & operator[] (unsigned n);
        size_t size (void) const; // return number of elements
        void print (void) const;
        MatlabVector operator+ (const MatlabVector& other) const;
        MatlabVector operator* (double scalar) const;
};

#endif //MATLABVECTOR_MATLABLIKEVECTOR_H
```

However, normally when we sum a vector and a number we would like to have one of the two behaviour: either an error or each element of the vector incremented by the number. This is impossible because of the implicit class-type conversion. To avoid it, we write "EXPLICIT" before the constructor: (TO avoid the implicit conversion) !!

```
explicit MatlabVector (unsigned n):
    elem (n, 0.) {}
```

Single-argument constructors must be marked **explicit** to avoid unintentional implicit convertion

Now to obtain the same result we must write:

```
MatlabVector v6 = v4 + MatlabVector (10);
```

```cpp
//-----------------------------------------------------------------------------
// MatlabVector.cpp
//-----------------------------------------------------------------------------
#include "MatlabVector.h"

using std::cout;
using std::endl;

void MatlabVector::print (void) const{
    for (size_t i =0; i< elem.size(); ++i)
        cout << elem[i] << " ";
    cout << endl;
}

size_t MatlabVector::size (void) const{
    return elem.size();
}

MatlabVector MatlabVector::operator* (double scalar) const{
    MatlabVector result;
    for (unsigned i=0; i<elem.size(); ++i)
        result[i] =scalar * elem[i];
    return result;
}

MatlabVector MatlabVector::operator+ (const MatlabVector &other) const{
    MatlabVector result;
    if (elem.size() == other.size()) {
        for (unsigned i = 0; i < elem.size(); ++i)
            result[i] = elem[i] + other.elem[i];
    }
    else
        result[0] = std::numeric_limits<double>::quiet_NaN();
    return result;
}

double & MatlabVector::operator[] (unsigned n){
    while (elem.size() < n+1)
        elem.push_back(0.);
    return elem[n];
}
```

→ we return a vector of an element equal to "Not a Number"

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Calendar

Federica Filippini

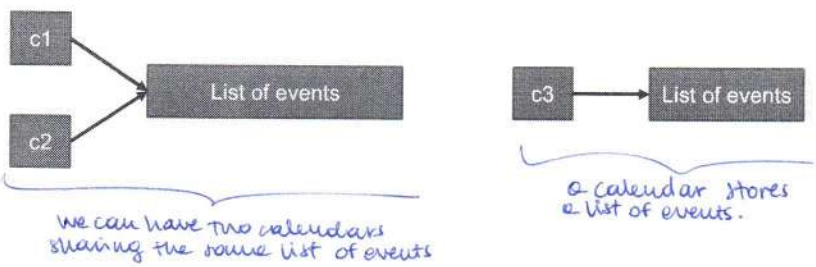
Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



## Goal

- Implement a simple calendar
- A `Calendar` stores `Event`s; each `Event` is characterized by date (`time_t`) and name (`std::string`)
  - The class `Event` is given
- **Like-a-pointer** behavior — We want the class Calendar to behave like-a-pointer






## Required methods

- **`addEvent(const Event&)`**, that receives an `Event` and adds it to the calendar
- **`updateEvent(time_t, const string&, time_t, const string&)`**, that receives old date and name, finds the corresponding event and updates the information with the new values
- **`print()`**, that prints the name of all events in the calendar

```cpp
//---------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------
#include "Calendar.h"

using std::cout;
using std::endl;

int main() {
    Calendar c0;
    Event e0(time(0), "Important Meeting");
    c0.addEvent(e0);

    Calendar c1;
    Event e1(time(0), "Andrew's Birthday");
    c1.addEvent(e1);
    c1 = c0;

    Calendar c2(c0);
    Event e2(time(0), "Trip to Cairo");
    c2.addEvent(e2);

    c2 = Calendar();
    Event e3(time(0), "Visit to Museum");
    c2.addEvent(e3);

    cout << "c0:" << endl;
    c0.print();
    cout << "c1:" << endl;
    c1.print();
    cout << "c2:" << endl;
    c2.print();

    return 0;
}
```

c2 is created as a copy of c0

```
c0.print();  ->  Important Meeting
                 Trip to Cairo
c1.print();  ->  Important Meeting
                 Trip to Cairo
c2.print();  ->  Visit to Museum
```

```cpp
//---------------------------------------------------------------------------
// Calendar.h
//---------------------------------------------------------------------------
#ifndef CALENDAR_H
#define CALENDAR_H

#include <iostream>
#include <vector>
#include <memory>
#include "Event.h"

class Calendar {

    private:
        std::shared_ptr<std::vector<Event>> events;

    public:
        Calendar (void): events(std::make_shared<std::vector<Event>>()) {}
        void addEvent (const Event&);
        void print (void) const;
};      void updateEvent (time_t, std::string, time_t, std::string);

#endif // CALENDAR_H
```

If we wanted a like-a-value we would write:
std::vector<Event> events;
Instead, we create a shared pointer to a vector.
In this way a group of events can be shared among more calendars.

```cpp
//---------------------------------------------------------------------------
// Calendar.cpp
//---------------------------------------------------------------------------
#include "Calendar.h"

void Calendar::addEvent (const Event &event){
    events->push_back(event);
}

void Calendar::print() const {
    for (const Event &e : *events)
        std::cout << e.getName() << std::endl;
}

void updateEvent (time_t old_d, std::string old_n, time_t new_d, std::string new_n) {
    bool found = false;
    for (std::vector<Event>::iterator it = events->begin(); it != events->end() && !found; ++it) {
        if (it->getTime() == old_d && it->getName() == old_n) {
            found = true;
            it->setTime(new_d);
            it->setName(new_n);
        }
    }
}
```

otherwise: (*events).push_back(event);

here we use begin instead of cbegin because we need the iterator to modify the event → cannot be constant

```
//-----------------------------------------------------------------------------------------------------------------------
// Event.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef EVENT_H
#define EVENT_H

# include <string>

class Event {

    private:
        time_t date;
        std::string name;

    public:
        Event(time_t d, const std::string& n): date(d), name(n) {};
        time_t getTime (void) const;
        std::string getName (void) const;
};

#endif // EVENT_H


//-----------------------------------------------------------------------------------------------------------------------
// Event.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "Event.h"

time_t Event::getTime (void) const {
    return date;
};

std::string Event::getName (void) const {
    return name;
};
```

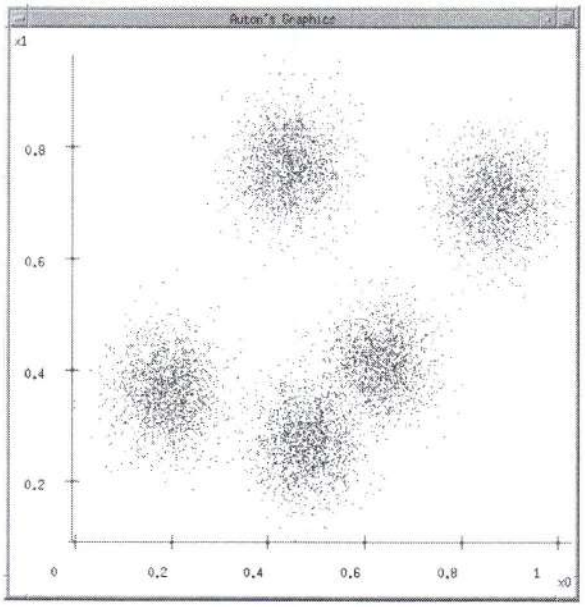
# K-means



**Andrew W. Moore**
**Professor**
**School of Computer Science**
**Carnegie Mellon University**
www.cs.cmu.edu/~awm
awm@cs.cmu.edu
412-268-7599

 Nov 16th, 2001

## Some Data



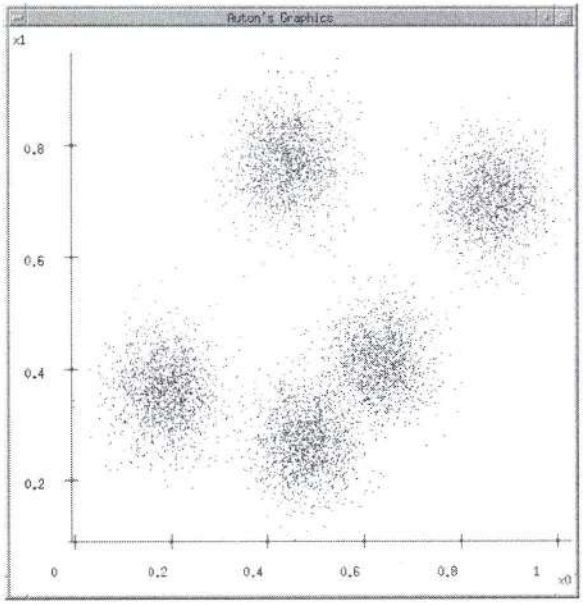
## K-means clustering

- Input: K, set of points
- Place centroids $c_1$, $c_2$, ..., $c_k$ at random locations (alternatively add random lables to points)
- Repeat until convergences:
  - for each cluster j=1...K:
    - new centroid $c_j$ = mean of all points $P_i$ assigned to cluster j
  - for each point $P_i$:
    - find nearest centrod $c_j$
    - assing point $P_i$ to cluster j
- Stop when none of the cluster assignments change

# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*

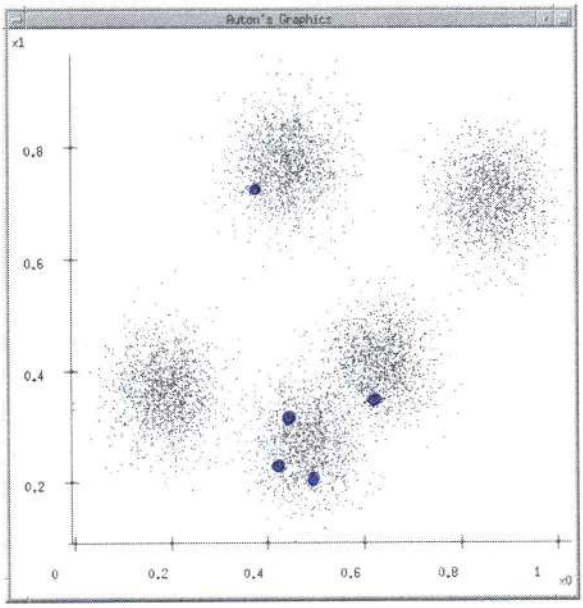







# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations

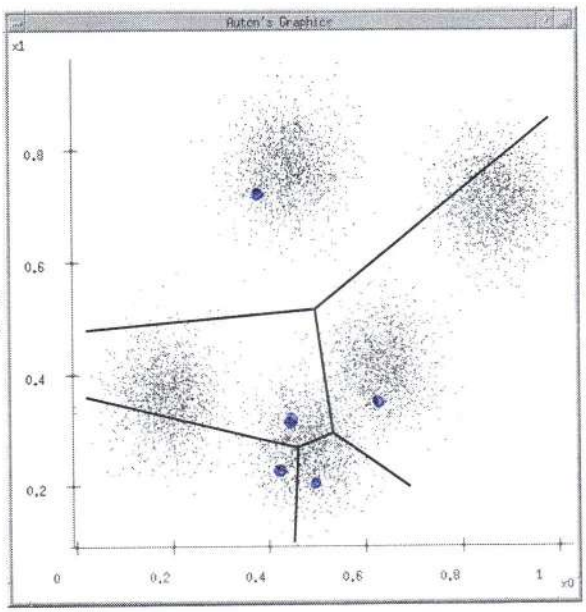






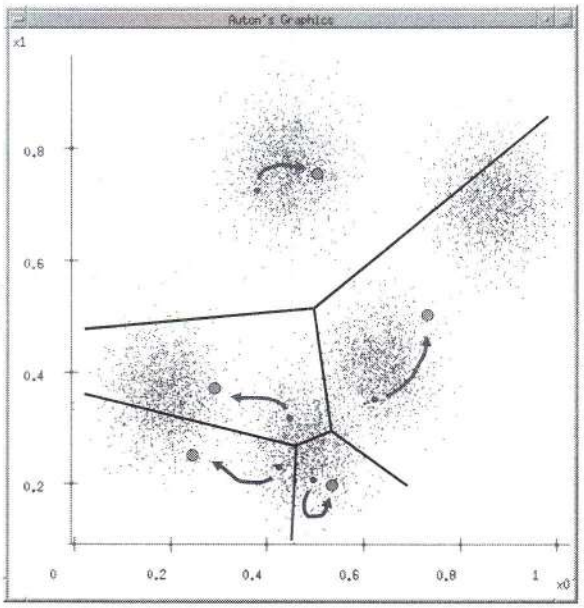

# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to. (Thus each Center "owns" a set of datapoints)



# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns

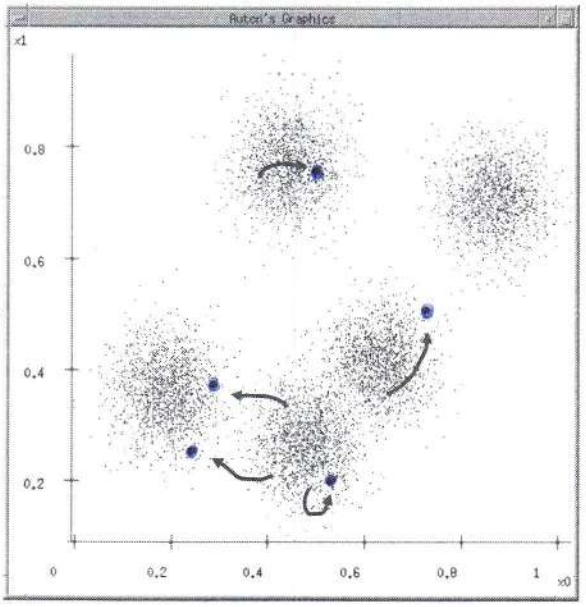







# K-means

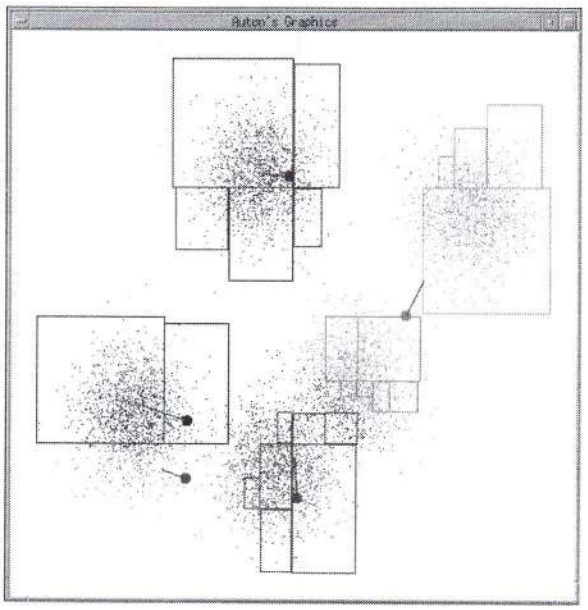
1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns...
5. ...and jumps there
6. ...Repeat until terminated!



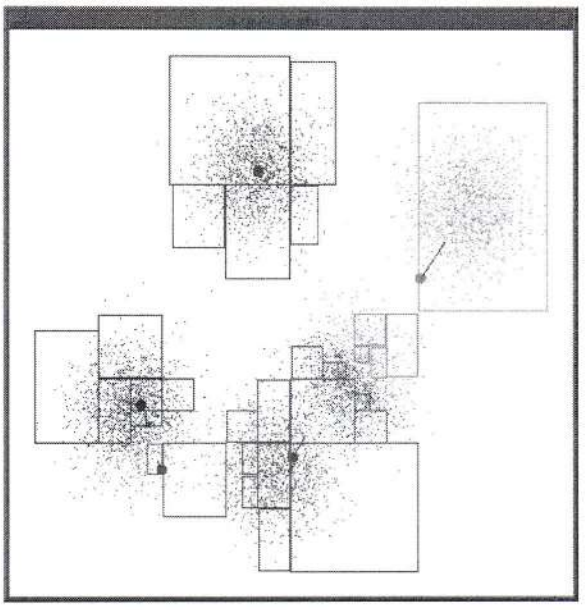





# K-means continues ...



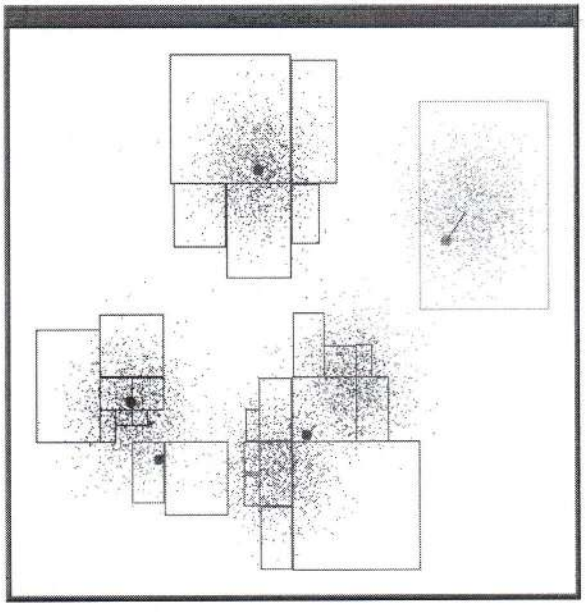
# K-means continues ...







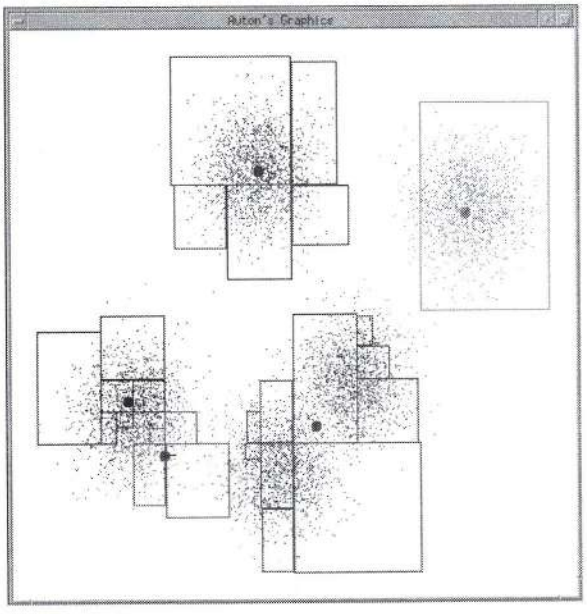
# K-means continues ...



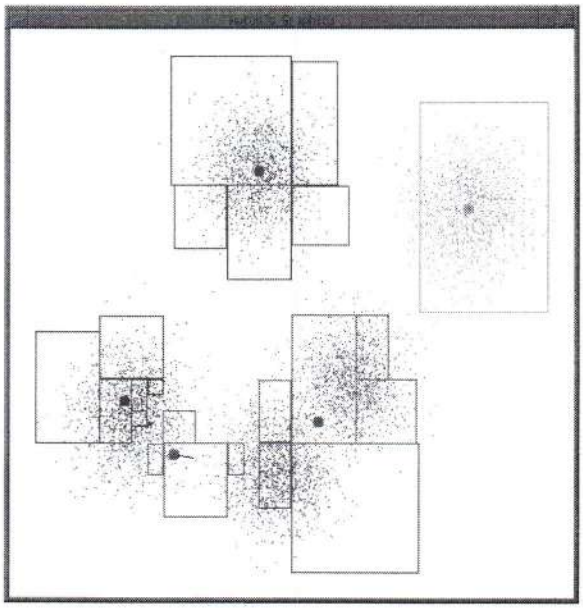




# K-means continues ...



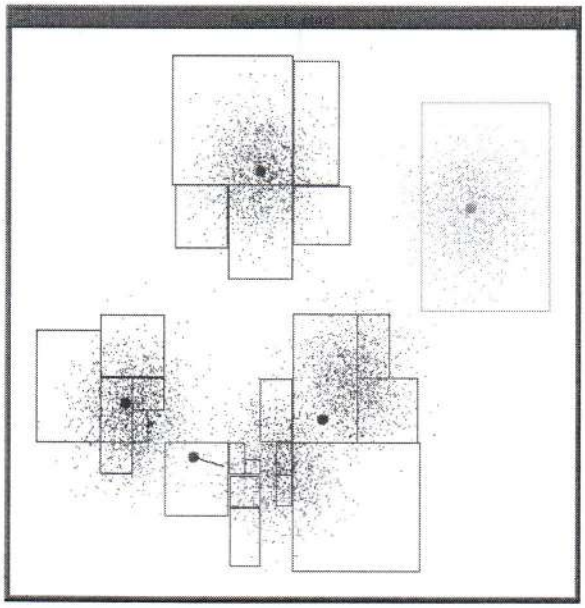
# K-means continues ...







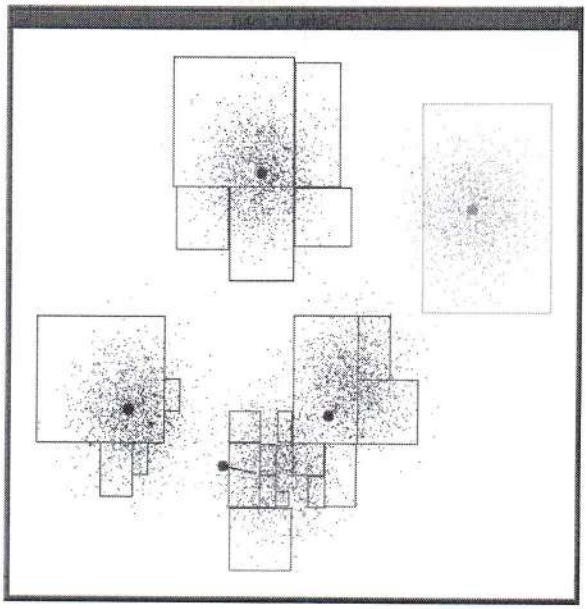
# K-means continues ...



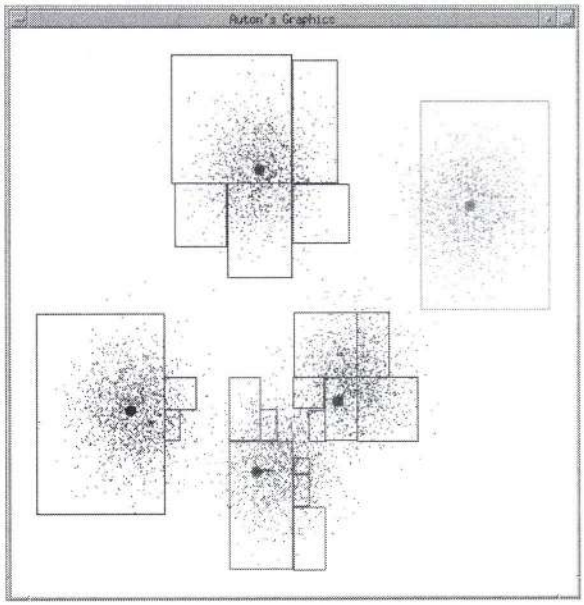




# K-means continues ...



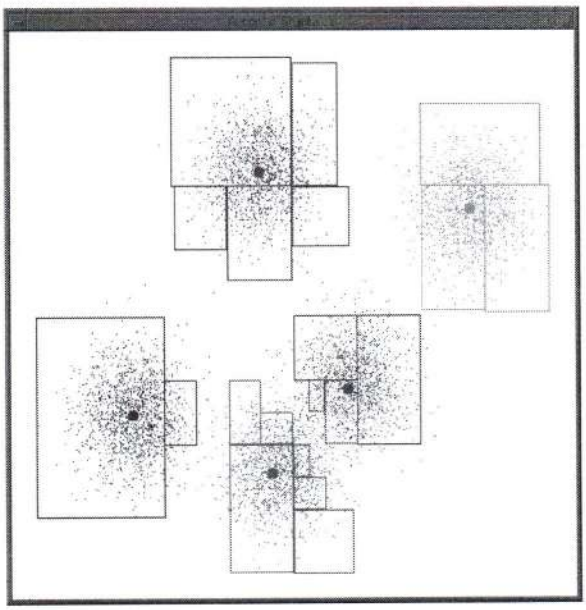
# K-means continues ...








# K-means terminates



```
//-----------------------------------------------------------------------------------------------------------------------
// main.hpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "Clustering.hpp"

int main() {
    Clustering c(2,10,2,10000);
    c.print();
    c.calc_cluster();
    c.print();
    return 0;
}


//-----------------------------------------------------------------------------------------------------------------------
// Point.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef POINT_H_
#define POINT_H_

#include <vector>

class Point{

    protected:
        std::vector<double> x;

    public:
        Point (const std::vector<double>&  coords) : x(coords) {};
        double distance (const Point& p) const;
        void print (void) const;
        double get_coord(int i) const;
};

#endif /* POINT_H_ */


//-----------------------------------------------------------------------------------------------------------------------
// Point.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <cmath>
#include <iostream>

#include "Point.hpp"

double Point::distance (const Point& p) const{
    double dist = 0.0;
    for (std::size_t i = 0; i < x.size(); ++i){
        const double delta = x[i] - p.x[i];
        dist += delta * delta;
    }
    return sqrt(dist);
}

void Point::print (void) const{
    for (auto it = x.begin (); it != x.end (); ++it){
        std::cout << *it;
        std::cout << " ";
    }
    std::cout << std::endl;
}

double Point::get_coord (int i) const{
    return x[i];
}
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Clustering.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef CLUSTERING_H_
#define CLUSTERING_H_

#include <vector>
#include "Centroid.hpp"
#include "Point.hpp"

class Clustering{

    typedef std::vector<unsigned> f_labels_type ;
    typedef std::vector<Centroid> centers_type;
    typedef std::vector<std::vector<Point *>> clusters_type;

    static constexpr double MAX_COORD=1000.0;

    std::vector<Point> points;
    f_labels_type labels; //label function
    unsigned p;           //number of dimensions
    unsigned n;           //number of points
    unsigned k;           //number of clusters

    centers_type centers;
    clusters_type clusters;

    unsigned max_it; //maximum number of iterations

    unsigned min_dist_index (const Point& point) const; //returns the index of the centroid closest to point
    void print_labels (void) const;
    void print_centers (void) const;
    void print_clusters (void) const;

    public:
        Clustering (unsigned dimensions, unsigned n_points, unsigned k_cluster, unsigned max_iterations);
        void print (void) const;
        void calc_cluster (void);
};

#endif /* CLUSTERING_H_ */
```

One row for each centroid and then in each row we store a vector which is pointing to all the points of the cluster (this is conceptually a matrix)

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Clustering.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <random>
#include "Clustering.hpp"

Clustering::Clustering (unsigned dimensions, unsigned n_points, unsigned k_cluster, unsigned max_iterations):
    p (dimensions), n (n_points), k (k_cluster), max_it (max_iterations)
{
        std::default_random_engine generator;
        std::uniform_real_distribution<double> distribution (0.0, MAX_COORD);
        for (unsigned i = 0; i < n; ++i){
            // Generate n random points
            // Generate random coordinates
            std::vector<double> coords;
            for (unsigned j = 0; j < p; ++j)
                coords.push_back (distribution (generator));
            // Generate a new Point
            points.push_back (coords);
            labels.push_back (0); // assign every point to class 0
        }
        std::vector<double> origin (p, 0);
        // Create and intialize centroids to origin
        for (unsigned i = 0; i < k; ++i)
            centers.push_back (origin);
        // Create and initialize clusters to nulls
        for (unsigned i = 0; i < k; ++i)
            clusters.push_back ({nullptr});
}

void Clustering::print (void) const{
    // Print points and labels
    for (std::size_t i = 0; i < points.size (); ++i){
        points[i].print ();
        std::cout << " : ";
        std::cout << labels[i];
        std::cout << std::endl;
    }
}
```

```cpp
void Clustering::calc_cluster (void) {
    std::default_random_engine generator;
    std::uniform_int_distribution<int> distribution (0, k - 1);
    //Randomly initialize labels
    for (auto it = labels.begin (); it != labels.end (); ++it)
        *it = distribution (generator);
    bool term_cond = false;
    unsigned i;
    for (i = 0; i < max_it && ! term_cond; ++i){
        f_labels_type old_labels (labels);
        clusters_type new_clusters (k);
        // update clusters according to new labels
        for (std::size_t j = 0; j < points.size (); ++j)
            new_clusters[labels[j]].push_back (&points[j]);
        clusters.swap (new_clusters);
        // update centroids
        for (std::size_t j = 0; j < centers.size (); ++j)
            centers[j].update_coords (clusters[j]);
        // assign points to new centroids
        for (std::size_t j = 0; j < points.size (); ++j){
            const unsigned min_dist_ind = min_dist_index (points[j]);
            labels[j] = min_dist_ind;
        }
        term_cond = (old_labels == labels);
    }
    std::cout << "Number of iterations: " << i << std::endl;
    std::cout << "Final result!" << std::endl;
    print_labels ();
    std::cout << std::endl;
}
```

save old labels

create a new clustering (new clusters) ← this is "empty", we're updating it (↓)

we're pushing the ADDRESS of points[j]

more efficient way for: clusters = new_clusters; *

```cpp
unsigned Clustering::min_dist_index (const Point& point) const{
    int min_dist_ind = 0;
    double min_dist = point.distance (centers[0]);
    for (std::size_t i = 1; i < centers.size (); ++i){
        const double dist = point.distance (centers[i]);
        if (dist < min_dist){
            min_dist = dist;
            min_dist_ind = i;
        }
    }
    return min_dist_ind;
}

void Clustering::print_labels (void) const{
    for (auto it = labels.begin (); it != labels.end (); ++it){
        std::cout << *it << " ";
    }
}

void Clustering::print_centers (void) const{
    std::cout << "Centers size: ";
    std::cout << centers.size ();
    std::cout << std::endl;
    for (std::size_t i = 0; i < centers.size (); ++i)
        centers[i].print ();
}

void Clustering::print_clusters (void) const{
    for (unsigned i = 0; i < k; ++i){
        std::cout << "Cluster: " <<  i << std::endl;
        for (Point* c: clusters[i]){
            c -> print ();
            std::cout << std::endl;
        }
    }
}
```

we assume that the first centroid will be the closer

* Why is this better?
Because in this case there is no copy, the two vectors just point one each other's content!

```cpp
//-----------------------------------------------------------------------------
// Centroid.hpp
//-----------------------------------------------------------------------------
#ifndef CENTROID_H_
#define CENTROID_H_

#include <iostream>
#include <vector>
#include "Point.hpp"

class Centroid : public Point{
    public:
        Centroid (const std::vector<double>& coords): Point (coords) {};
        void update_coords (const std::vector<Point *>&);
};

#endif /* CENTROID_H_ */


//-----------------------------------------------------------------------------
// Centroid.cpp
//-----------------------------------------------------------------------------
#include "Centroid.hpp"

void Centroid::update_coords (const std::vector<Point *> & ps){
    std::vector<double> new_coords(x.size(),0);
    for (std::size_t i = 0; i < ps.size (); ++i){
        for (std::size_t j = 0; j < x.size (); ++j)
            new_coords[j] += ps[i] -> get_coord (j);
    }
    for (std::size_t j = 0; j < x.size (); ++j)
        new_coords[j] /= ps.size ();
    x.swap (new_coords);
}
```

vector of pointers to Point (passed through a reference, not copy)

for every point

for every coordinate of every point

because ps[i] is a pointer to a point, not a point

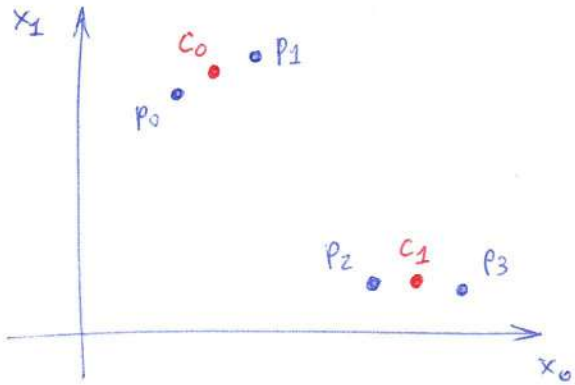
because we initialized to 0 ✓

that's because the centroids are means

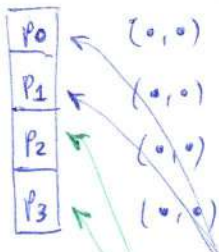
instead of
x = new_coords;

Complexity
O(n)

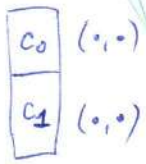
O(n·p)




Points stored in a vector "points"




Centers stored in a vector "centers"




"labels"


Cluster matrix:
(vector of vectors)
"clusters"

GoodReads (for complexity)

```
//---------------------------------------------------------------------------
// main.hpp
//---------------------------------------------------------------------------
#include <iostream>
#include "Review.h"
#include "GoodReads.h"

using std::cout;
using std::endl;

int main() {
    GoodReads data;
    /*  Review r1("Harry Potter", "Great wonderful", 5);
        Review r2("Harry Potter", "No words amazing", 5);
        cout << r1.to_string() << endl;
        cout << r2.to_string() << endl;    */

    data.add_book("Harry Potter", 400, "Pub", "J. K. Rowling");
    data.add_book("Harry Potter", 400, "Pub", "J. K. Rowling");

    data.add_review("Harry Potter", "Great wonderful", 5);
    data.add_review("Harry Potter", "No words amazing", 4);
    data.print_book("Harry Potter");

    data.add_book("C++ Primer", 800, "Pub 2", "S. Lippman");
    data.add_review("C++ Primer", "Very interesting book Great", 3);
    data.print_book("C++ Primer");
    data.print_book("C+ Primer");

    cout << "Good reads average rating ";
    cout << data.get_avg_rating() << endl;
    cout << "Searching Great" <<endl;
    data.search_reviews({"Great"});
    cout << "Searching Great, wonderful" <<endl;
    data.search_reviews({"Great", "wonderful"});
    cout << "Searching point" <<endl;
    data.search_reviews({"point"});
    return 0;
}
```

```
//-----------------------------------------------------------------------------
// Review.h
//-----------------------------------------------------------------------------
#ifndef GOODREADS_REVIEW_H
#define GOODREADS_REVIEW_H

#include <string>
#include <vector>
#include <sstream>

class Review {

    std::string book_title;
    std::string text;
    unsigned rating;
    std::vector<std::string> words;

    public:
        Review(const std::string& bookTitle, const std::string& text, unsigned int rating);
        std::string to_string() const;
        std::string get_text() const;
        std::vector<std::string> get_words() const;

    private:
        bool find_in_words(const std::string& w) const;
};

std::vector<std::string> split(const std::string& s, char d);

#endif //GOODREADS_REVIEW_H


//-----------------------------------------------------------------------------
// Review.cpp
//-----------------------------------------------------------------------------
#include "Review.h"

using std::vector;
using std::string;

Review::Review(const string& bookTitle, const string& t, unsigned int r): book_title(bookTitle), text(t), rating(r)
{
    const vector<string> text_strings(split(t, ' '));
    words.push_back(text_strings[0]);
    for (size_t i = 1; i < text_strings.size(); ++i)
        if (!find_in_words(text_strings[i]))
            words.push_back(text_strings[i]);
}

string Review::to_string() const {
    return book_title + " " + text + " " +  std::to_string(rating);
}

string Review::get_text() const {
    return text;
}

vector<string> Review::get_words() const {
    return words;
}

bool Review::find_in_words(const string &w) const {
    for (const string& word: words)
        if (w == word)
            return true;
    return false;
}

vector<string> split(const string &s, char d) {
    string word;
    std::vector<string> v;
    std::istringstream text_read (s);
    while (std::getline (text_read, word, d))
        v.push_back (word);
    return v;
}
```

```
//-----------------------------------------------------------------------------------------------------------------------
// GoodReads.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef GOODREADS_GOODREADS_H
#define GOODREADS_GOODREADS_H

#include <string>
#include <vector>
#include <iostream>
#include "Book.h"
#include "Review.h"

class GoodReads {

    public:
        typedef std::vector<Book>::const_iterator const_books_it;
        typedef std::vector<Book>::iterator books_it;

    private:
        std::vector<Book> books; // <title, Book>
        std::vector<Review> reviews;

    public:
        void add_book(const std::string& title, unsigned pagesNumber, const std::string& publisher,
                      const std::string& author);
        void add_review(const std::string& bookTitle, const std::string& text, unsigned int rating);
        float get_avg_rating() const;
        float get_avg_rating(const std::string& title) const;
        void search_reviews(const std::vector<std::string>& keywords) const;
        void print_book(const std::string& title) const;

    private:
        // return the iterator of the book with a given title in books, books.cend() otherwise
        const_books_it find_book(const std::string& title) const;
        // return the iterator of the book with a given title in books, books.end() otherwise
        books_it  find_book(const std::string& title);
        // return true iff all keywords are included in words
        bool includes_all(const std::vector<std::string>& words, const std::vector<std::string>& keywords) const;
        // return true iff k is included in words
        bool includes_word(const std::vector<std::string>& words, const std::string& k) const;
};

#endif //GOODREADS_GOODREADS_H


//-----------------------------------------------------------------------------------------------------------------------
// GoodReads.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "GoodReads.h"

using std::string;
using std::vector;
using std::cout;
using std::cerr;
using std::endl;

void GoodReads::add_book(const string &title, unsigned int pages_number, const string &publisher,
                         const string &author)
{
    Book bd(pages_number, publisher, author, title);
    auto it = find_book(title);
    if (it == books.cend()) // the book is not already in the collection
        books.push_back(bd);
    else
        cerr << "The book is already in the books collection" <<endl;
}

void GoodReads::add_review(const string &book_title, const string &text, unsigned int rating) {
    const auto it = find_book(book_title);
    if (it == books.cend()) // the book is not in the collection
        cerr << "The book is not in the books collection" <<endl;
    else{
        Review r(book_title, text, rating);
        reviews.push_back(r);
        it->add_review(reviews.size()-1, rating);
    }
}

float GoodReads::get_avg_rating() const {
    float avg= 0;
    size_t count = 0;
    for (auto it = books.cbegin(); it != books.cend(); ++it){
        avg += it->get_avg_rating();
        count++;
    }
    return avg/count;
}

float GoodReads::get_avg_rating(const string & title) const {
    const auto it = find_book(title);
    if (it == books.cend()){ // the book is not in the collection
        cerr << "The book is not in the books collection" <<endl;
        return 0;
    }
    else
        return it->get_avg_rating();
}
```

O(1)
O(n_books)
O(n_books)
} O(n_books)

O(n_books)

O(n_books)

```cpp
void GoodReads::search_reviews(const vector<string> & keywords) const {
    for (auto it = reviews.cbegin(); it != reviews.cend(); ++it){
        const vector<string>& words = it->get_words();
        if(includes_all(words, keywords))
            cout << it->to_string() << endl;
    }
}

void GoodReads::print_book(const string &title) const {
    const auto it = find_book(title);
    if (it == books.cend())// the book is not in the collection
        cerr << "The book is not in the books collection" <<endl;
    else{
        // print book data
        cout << it->to_string()<<endl;
        auto rev_list = it->get_review_indexes();
        auto rev_it = rev_list.cbegin();
        while ( rev_it != rev_list.cend()){
            cout << reviews[*rev_it].to_string() << endl;
            rev_it++;
        }
    }
}

// return true iff all keywords are included in words
bool GoodReads::includes_all(const vector<string>& words, const vector<string>& keywords) const {
    for (const string& k : keywords)
        if (!includes_word(words,k))
            return false;
    return true;
}

// return true iff k is included in words
bool GoodReads::includes_word (const vector<string>& words, const string& k) const {
    for (const string& w : words)
        if (w == k)
            return true;
    return false;
}

GoodReads::const_books_it GoodReads::find_book(const string& title) const {
    for (const_books_it it = books.cbegin(); it != books.cend(); ++it)
        if (it->get_title() == title)
            return it;
    return books.cend();
}

GoodReads::books_it GoodReads::find_book(const string& title) {
    for (books_it it = books.begin(); it != books.end(); ++it)
        if (it->get_title() == title)
            return it;
    return books.end();
}
```

O(n_books)

```
//-----------------------------------------------------------------------------------------------------------------
// Book.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef GOODREADS_BOOK_H
#define GOODREADS_BOOK_H

#include <iostream>
#include <string>
#include <vector>

class Book {

    std::vector<unsigned> ratings_distr;
    unsigned pages_number;
    std::string publisher;
    unsigned review_count;
    std::string author;
    std::string title;
    float avg_rating;
    std::vector<unsigned> review_indexes;

    public:
        Book(unsigned int pagesNumber, const std::string &publisher, const std::string &author, const std::string &title);
        float get_avg_rating() const;
        void add_review(unsigned index, unsigned stars);
        std::string to_string() const;
        std::vector<unsigned> get_review_indexes() const ;
        std::string get_title() const;

    private:
        float compute_rating();
};

#endif //GOODREADS_BOOK_H


//-----------------------------------------------------------------------------------------------------------------
// Book.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "Book.h"

using std::cout;
using std::endl;
using std::string;
using std::vector;

Book::Book(unsigned pagesNumber, const string &pub,
           const string &author, const string &title) :
           pages_number(pagesNumber), publisher(pub), author(author), title(title)
{
    ratings_distr = vector<unsigned>(5,0);
    review_count = 0;
    avg_rating = 0.0;
}

float Book::get_avg_rating() const {
    return avg_rating;
}

void Book::add_review(unsigned int index, unsigned int stars) {
    review_indexes.push_back(index);    O(n_reviews)
    ratings_distr[stars-1]++;           O(1)
    review_count++;                     O(1)
    avg_rating = compute_rating();      O(1)
}                                       } complexity O(n_reviews)

float Book::compute_rating() {
    float average = 0;
    for (size_t i = 0; i < ratings_distr.size(); ++i)
        average += (i+1) * ratings_distr[i];
    return average/review_count;
}

string Book::to_string() const {
    return title + " " + author + " " + publisher + " " + std::to_string(pages_number) + " " +
           std::to_string(review_count) + " " + std::to_string(avg_rating);
}

vector<unsigned> Book::get_review_indexes() const {
    return review_indexes;
}

string Book::get_title() const {
    return title;
}
```

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Document Store

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it




## Goal

- Implement a data structure, called `DocumentStore`, that stores and manages `Documents`
- A `Document` is characterized by some text (`std::string`) and an id (`size_t`)
- `DocumentStore` contains two vectors of `Documents`: one for completed documents (`docs`) and one for drafts (`docsDraft`)
- The size of `docs` is set in the constructor and can be configured by the users. The size of `docsDraft` is fixed and equal to 10
- Documents can be added to the vectors through methods `addDocument` and `saveAsDraft`, respectively. In both cases, if the vector is already full no actions are taken



## Requirements

- Complete the implementation of `DocumentStore` in order to implement a ***like-a-value*** behavior
- In particular:
  - **completed documents must be copied between `DocumentStores`**
  - **drafts must not be transferred during the operation but just emptied**
- **List and motivate** the program **output** considering the main file provided

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "DocumentStore.hpp"

int main(){
    std::size_t id = 945;
    Document d0("Apple", id++);
    Document d1("Orange", id++);
    Document d2("Melon", id++);
    Document d3("Peach", id++);
    Document d4("Strawberry", id++);

    DocumentStore ds0(3);
    ds0.addDocument(d0);
    ds0.saveAsDraft(d1);

    DocumentStore ds1(ds0);
    ds1.addDocument(d2);
    ds1.addDocument(d3);

    DocumentStore ds2(2);
    ds2.addDocument(d1);
    ds2.addDocument(d4);
    ds2.saveAsDraft(d2);
    ds2.saveAsDraft(d3);

    DocumentStore ds3(3);
    ds2.addDocument(d0);
    ds2.addDocument(d1);
    ds3 = ds2;
    ds3.addDocument(d3);

    std::cout << "--------------- ds0 ---------------" << std::endl;
    ds0.print();

    std::cout << "\n--------------- ds1 ---------------" << std::endl;
    ds1.print();

    std::cout << "\n--------------- ds2 ---------------" << std::endl;
    ds2.print();

    std::cout << "\n--------------- ds3 ---------------" << std::endl;
    ds3.print();
    return 0;
}
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Document.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef DOCUMENT_H
#define DOCUMENT_H

#include <string>
#include <iostream>

class Document {

    private:
        std::string text = "";
        std::size_t id = 0;

    public:
        Document () = default;
        Document (const std::string& text, std::size_t id): text(text), id(id) {}
        const std::string& getText () const;
        std::size_t getId () const;
        void print () const;
};

#endif /* DOCUMENT_H */
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// Document.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "Document.hpp"

const std::string& Document::getText () const{
    return text;
}

std::size_t Document::getId () const{
    return id;
}

void Document::print () const{
    std::cout << "id: " << id << "\n\t text: " << text << std::endl;
}
```

```
//-----------------------------------------------------------------------------
// DocumentStore.hpp
//-----------------------------------------------------------------------------
#ifndef DOCUMENTSTORE_H
#define DOCUMENTSTORE_H

#include <vector>
#include "Document.hpp"

const unsigned DRAFT_SIZE = 10;

class DocumentStore {

    private:
        std::vector<Document> docs;
        std::vector<Document> docsDraft;
        std::size_t size;
        std::size_t curr;
        std::size_t currDraft;

    public:
        explicit DocumentStore (std::size_t);
        DocumentStore (const DocumentStore&);
        DocumentStore& operator= (const DocumentStore&);
        void addDocument (const Document&);
        void saveAsDraft (const Document&);
        void print () const;
};

#endif /* DOCUMENTSTORE_H */


//-----------------------------------------------------------------------------
// DocumentStore.cpp
//-----------------------------------------------------------------------------
#include "DocumentStore.hpp"

DocumentStore::DocumentStore (std::size_t s): docs(s), docsDraft(DRAFT_SIZE), size(s), curr(0), currDraft(0) {}

DocumentStore::DocumentStore (const DocumentStore& rhs):
    docs(rhs.docs), docsDraft(DRAFT_SIZE), size(rhs.size), curr(rhs.curr), currDraft(0){}

DocumentStore& DocumentStore::operator= (const DocumentStore& rhs){
    docs = rhs.docs;
    size = rhs.size;
    curr = rhs.curr;
    currDraft = 0;
    return *this;
}

void DocumentStore::addDocument (const Document& doc){
    if (curr < size)
        docs[curr++] = doc;
}

void DocumentStore::saveAsDraft (const Document& draft){
    if (currDraft < DRAFT_SIZE)
        docsDraft[currDraft++] = draft;
}

void DocumentStore::print () const{
    std::cout << "List of Documents:" << std::endl;
    for (std::size_t j = 0; j < curr; ++j)
        docs[j].print();
    std::cout << "List of Drafts:" << std::endl;
    for (std::size_t j = 0; j < currDraft; ++j)
        docsDraft[j].print();
}
```

we implement like this to obtain a like-a-value behavior

size of the vector of documents

index of the current document (last document that we added)

(same ↑ but for the vector of drafts)

if we do: DocumentStore d1; DocumentStore d2; d1 = d2;
we want to copy only docs, **not** drafts !

We should put : docsDraft = std::vector<Document> (DRAFT_SIZE);
however, because of currDraft = 0 and because of how is implemented saveAsDraft we don't need it

equivalent to: docsDraft[currDraft] = draft; currDraft++;

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Schedule Evaluation

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



## Goal

- Implement a program that, given the **scheduling** of a fixed number of jobs on a single machine, computes its **weighted tardiness**.

time at which we can start the job if the machine is free

| Job Id | Submission Time | Process Time | Due Date |
|---|---|---|---|
| J1 | 0 | 11 | 61 |
| J2 | 0 | 29 | 45 |
| J3 | 0 | 31 | 31 |
| J4 | 0 | 1 | 33 |
| J5 | 0 | 2 | 32 |

deadline, if violated we have to pay a penalty

- `tardiness` = { `0`, if `completion_time < due_date`; `completion_time - due_date`, otherwise

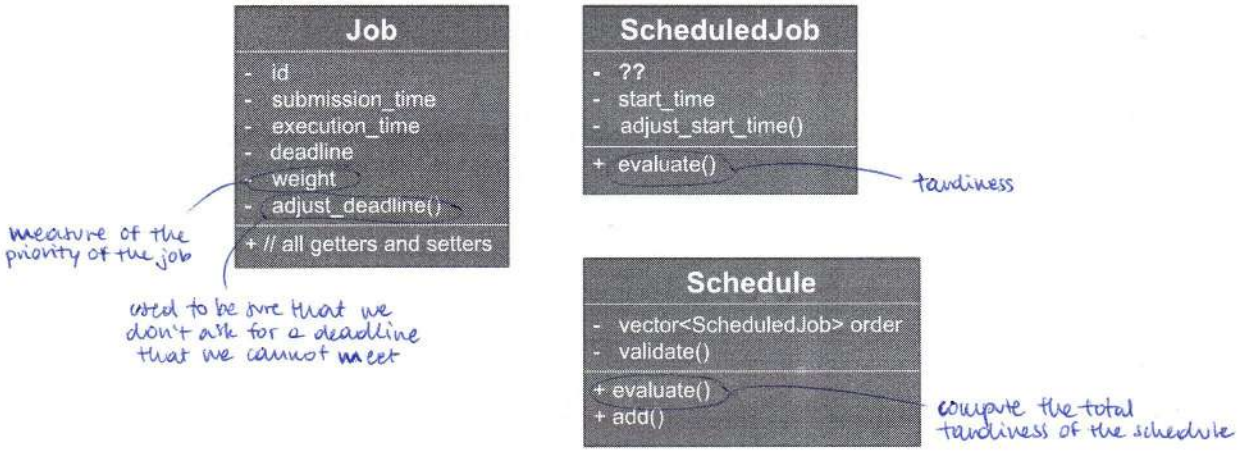


**Example:**

| Seq. | Start Time | Process Time | Completion Time | Due Date | Tardiness |
|---|---|---|---|---|---|
| J3 | 0 | 31 | 31 | 31 | 0 |
| J5 | 31 | 2 | 33 | 32 | 1 |
| J4 | 33 | 1 | 34 | 33 | 1 |
| J2 | 34 | 29 | 63 | 45 | 18 |
| J1 | 63 | 11 | 74 | 61 | 13 |
| | | | | | 33 |

- **Assumption (1):** all the tasks must be executed **sequentially**, i.e., it is not possible to start a job until the previous one has been completed.
- **Assumption (2)**: we want to minimize code replication **and memory usage**.

# Code Structure






# Required methods

- **`ScheduledJob::adjust_start_time()`**, that guarantees that `start_time` and `submission_time` of the current job are compatible
- **`ScheduledJob::set_start_time(time)`**, that sets the start time of the current job
- **`ScheduledJob::evaluate()`**, that computes the weighted tardiness of the current job
- **`Schedule::add(???)`**, that adds the given job to the schedule
- **`Schedule::validate()`**, that guarantees that the schedule is valid
- **`ScheduledJob::evaluate()`**, that computes the weighted tardiness of the schedule

```cpp
//-----------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <memory>
#include "Job.hh"
#include "ScheduledJob.hh"
#include "Schedule.hh"

using namespace std;

int main(){
    //Job(id_type j_id, time_instant sub_time, time_instant exe_time, time_instant dLine, double w =1.0):
    auto j1 = std::make_shared<Job> (0, 0, 1, 1, 2.0);
    auto j2 = std::make_shared<Job> (1, 0, 2, 2, 3.0);
    auto j3 = std::make_shared<Job> (2, 10, 4, 4);
    Schedule s;
    j1 -> print();
    cout << endl;
    j2 -> print();
    cout << endl;
    j3 -> print();
    cout << endl;
    cout << endl;
    s.add (j3);
    s.add (j2);
    s.add (j1);
    s.print();
    cout << "Total: " << s.evaluate() << endl << endl;
    std::size_t dim = 3;
    Schedule s_opt = s;
    double best_val = s.evaluate();
    bool improved = true;
    unsigned max_iterations = 10;
    cout << "Start Neigh exploration " << endl;
    s.print();
    cout << "Initial value " << best_val << endl;
    cout << endl;

    for (unsigned iter = 0; iter < max_iterations && improved; ++iter){
        cout << "Iteration " << iter << endl << endl;
        Schedule current_best = s_opt;
        double current_best_val = best_val;
        improved = false;

        for (unsigned i = 0; i < dim - 1; ++i){
            cout << "swap " << i << " " << i + 1 << endl;
            Schedule s_current = s_opt;
            s_current.swap (i, i + 1);
            cout << "Evaluating " << endl;
            s_current.print();
            cout << endl;
            const double current_val = s_current.evaluate();
            cout << "Value: " << current_val << endl;

            if (current_val < current_best_val){
                cout << "Improving " << endl;
                current_best = s_current;
                improved = true;
                current_best_val = current_val;
                cout << "New best " << endl;
                current_best.print();
                cout << endl;
            }
        }

        std::swap (s_opt, current_best);
        best_val = current_best_val;
    }

    cout << "Final optimal solution found" << endl;
    s_opt.print();
    cout << endl;
    cout << "Total: " << s_opt.evaluate() << endl;
    return 0;
}
```

```
//---------------------------------------------------------------------------------------------------------------------
// Job.h
//---------------------------------------------------------------------------------------------------------------------
#ifndef JOB_HH
#define JOB_HH

#include <iostream>

class Job{

    public:
        typedef long int time_instant;
        typedef unsigned int id_type;

        Job (id_type j_id, time_instant sub_time, time_instant exe_time, time_instant dline, double w = 1.0) :
            id (j_id), submission_time (sub_time), execution_time (exe_time), deadline (dline), weight (w)
            {
                adjust_deadline();
            }

        time_instant get_deadline() const;
        time_instant get_execution_time() const;
        time_instant get_submission_time() const;
        id_type get_ID() const;

        double get_weight() const;

        void set_deadline (time_instant deadline);
        void set_execution_time (time_instant execution_time);
        void set_submission_time (time_instant submission_time);
        void set_weight (double weight);
        void print() const;

    private:
        id_type id;
        time_instant submission_time;
        time_instant execution_time;
        time_instant deadline;
        double weight;
        void adjust_deadline (void);
};

#endif /* JOB_H_ */


//---------------------------------------------------------------------------------------------------------------------
// Job.cpp
//---------------------------------------------------------------------------------------------------------------------
#include "Job.hh"

Job::time_instant Job::get_deadline() const{
    return deadline;
}

Job::time_instant Job::get_execution_time() const{
    return execution_time;
}

Job::time_instant Job::get_submission_time() const{
    return submission_time;
}

double Job::get_weight() const{
    return weight;
}

Job::id_type Job::get_ID() const{
    return id;
}

void Job::set_deadline (time_instant d){
    deadline = d;
    adjust_deadline();
}

void Job::set_execution_time (time_instant t){
    execution_time = t;
    adjust_deadline();
}

void Job::set_submission_time (time_instant t){
    submission_time = t;
    adjust_deadline();
}

void Job::set_weight (double w){
    weight = w;
}
```

```
void Job::print() const{
    std::cout << "id: " << id << " submission_time " << submission_time << " execution_time " << execution_time
    <<  " deadline " << deadline << " weight " << weight;
}

void Job::adjust_deadline (void){
    if (deadline < submission_time + execution_time)
        deadline = submission_time + execution_time;
}


//-----------------------------------------------------------------------------------------------------------------
// Schedule.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef SCHEDULE_HH
#define SCHEDULE_HH

#include <vector>
#include <memory>
#include <initializer_list>
#include "ScheduledJob.hh"

class Schedule{

    private:
        std::vector<ScheduledJob> order;
        // guarantees that the schedule is valid, i.e. a job
        // starts after the end of the  one before
        void validate();

    public:
        typedef std::vector<ScheduledJob>::size_type size_type;
        Schedule() {};
        virtual ~Schedule() {};
        size_type size (void) const;
        // evaluate total weighted tardiness for the schedule;
        double evaluate() const;
        void print() const;
        void add (const std::shared_ptr<Job> & j);
        // swap jobs at position i and j
        void swap (std::size_t i, std::size_t j);
};

#endif /* SCHEDULE_H_ */


//-----------------------------------------------------------------------------------------------------------------
// Schedule.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "Schedule.hh"

Schedule::size_type Schedule::size (void) const{
    return order.size();
}

double Schedule::evaluate() const{
    double ret_val = 0;
    for (std::size_t i = 0; i < order.size(); ++i){
        ret_val += order[i].evaluate();
    }
    return ret_val;
}

void Schedule::print() const{
    for (std::size_t i = 0; i < order.size(); ++i){
        order[i].print();
        std::cout << std::endl;
    }
}

void Schedule::add (const std::shared_ptr<Job> & j){
    // this is the first job
    if (order.empty())
        order.push_back (ScheduledJob (j));
    else{
        ScheduledJob sj (j, order.back().get_end_time());
        order.push_back (sj);
    }
}

// swap jobs at position i and j
void Schedule::swap (std::size_t i, std::size_t j){
    if (i >= order.size() || j >= order.size())
        return;
    using std::swap;
    swap (order[i], order[j]);
    validate();
}
```

the start time of this new job is the endtime of the previous job

```
void Schedule::validate(){
    // guarantees that the schedule is valid, i.e. a job starts after the end of the one before
    // This can be optimized, check only after min(i,j) an empty schedule is ok
    if (order.empty())
        return;
    order[0].set_start_time (order[0].get_submission_time());
    ScheduledJob pred = order[0];
    for (std::size_t i = 1; i < order.size(); ++i){
        // adjust current job start time
        order[i].set_start_time (std::max (pred.get_end_time(), order[i].get_submission_time()));
        pred = order[i];
    }
}
```

we cannot start a job before the previous one has been completed and we cannot start the job before its submission time

```
//-----------------------------------------------------------------------------------------------------------------
// ScheduledJob.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef SCHEDULEDJOB_HH
#define SCHEDULEDJOB_HH

#include <memory>
#include "Job.hh"

class ScheduledJob{

    private:
        std::shared_ptr<Job> ptr;
        Job::time_instant start_time;
        void adjust_start_time (void);

    public:
        ScheduledJob (const std::shared_ptr<Job> & j_ptr, Job::time_instant st_time):
            ptr (j_ptr), start_time (st_time)
            {
                adjust_start_time();
            }
        ScheduledJob (Job j, Job::time_instant st_time):
            ptr (std::make_shared<Job> (j)), start_time (st_time)
            {
                adjust_start_time();
            }
        explicit ScheduledJob (const std::shared_ptr<Job> & j_ptr) :
            ptr (j_ptr), start_time (j_ptr->get_submission_time()) {}

        Job::time_instant get_end_time() const;
        Job::time_instant get_start_time() const;
        void set_start_time (Job::time_instant  st_time);
        void print() const;
        double evaluate() const;
        Job::time_instant get_deadline() const;
        Job::time_instant get_execution_time() const;
        Job::time_instant get_submission_time() const;
};

#endif /* SCHEDULEDJOB_H_ */
```

we want to reduce the memory usage of the code

as the type we have "time_instant" defined in **Job**

```
//-----------------------------------------------------------------------------------------------------------------
// ScheduledJob.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "ScheduledJob.hh"

using namespace std;

Job::time_instant ScheduledJob::get_end_time() const{
    return start_time + ptr -> get_execution_time();
}

void ScheduledJob::print() const{
    ptr -> print();
    cout << " start_time " << start_time << " end_time " << get_end_time();
}

double ScheduledJob::evaluate() const{
    return get_end_time() < ptr -> get_deadline()
        ? 0
        : ptr -> get_weight() * (static_cast<double>(get_end_time()) - static_cast<double>(ptr->get_deadline()));
}

Job::time_instant ScheduledJob::get_start_time() const{
    return start_time;
}

void ScheduledJob::set_start_time (Job::time_instant st_time){
    start_time = st_time;
    adjust_start_time();
}
```

```
{
 double tardiness = 0.0;
 if (get_end_time() > ptr -> get_deadline())
   tardiness = ((double) get_end_time -
        ptr -> get_deadline()) *
        ptr -> get_weight();
 }
 return tardiness;
}
```

```
Job::time_instant ScheduledJob::get_deadline() const{
    return ptr -> get_deadline();
}

Job::time_instant ScheduledJob::get_execution_time() const{
    return ptr -> get_execution_time();
}

Job::time_instant ScheduledJob::get_submission_time() const{
    return ptr -> get_submission_time();
}

void ScheduledJob::adjust_start_time (void){
    if (start_time < ptr -> get_submission_time())
        start_time = ptr -> get_submission_time();
}
```

# DataFrame

```
//-----------------------------------------------------------------------------
// Solution 1
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <iostream>
#include <vector>
#include "DataFrame.hpp"

int main(){
    std::cout << "Test DataFrame" << std::endl;
    DataFrame df ("c1 c2");
    /*    df.test_check("c1");
          df.test_check("c2");
          df.test_check("c3");
    */
    df.print();
    std::vector<double> v1 = {1, 2, 3};
    std::vector<double> v2 = {4, 5, 6};
    df.set_column ("c1", v1);
    df.set_column ("c2", v2);
    df.set_column ("c2", v1);
    df.print();
    std::vector<double> v_res = df.get_column ("c1");
    std::cout << "Printing v_res" << std::endl;

    auto mm = df.get_mean("c1");
    std::cout << "mean: " << mm << std::endl;

    return 0;
}


//-----------------------------------------------------------------------------
// DataFrame.hpp
//-----------------------------------------------------------------------------
#ifndef DATAFRAME_HH
#define DATAFRAME_HH

#include <sstream>
#include <iostream>
#include <string>
#include <vector>

class DataFrame{

    public:
        typedef std::vector<std::string> key_container;
        typedef std::vector<double> mapped_container;
        typedef key_container::value_type key_type;
        typedef typename mapped_container::value_type mapped_type;
        typedef typename std::vector<mapped_container>::size_type size_type;

    private:
        key_container df_keys;
        std::vector<mapped_container> df_values;
        bool added_first_column;

        // return the names of columns in s separated by delim
        key_container split (const key_type & s, char delim) const;

        // return the index of column key, or df_keys.size () if not found
        size_type look_up (const key_type & key) const;

    public:
        explicit DataFrame (const std::string & c_names);
        explicit DataFrame (const key_container & names);
        void print (void) const;
        mapped_container get_column (const key_type & column_name) const;
        mapped_type const & get_element_at (const key_type & column_name, size_type index) const;
        void set_element_at (const key_type & column_name, size_type index, const mapped_type & value);
        double get_mean (const key_type & column_name) const;

        // add a new column with data
        void set_column (const key_type & column_name, const mapped_container & column_data);

        // return a copy of the DataFrame with rows i such that "c_name[i] == value"
        DataFrame select_equal (const key_type & c_name, const mapped_type & value) const;
};

#endif // DATAFRAME_HH
```

!! Why typename? Usually when we use "::" we use it to access the members or the methods of a class, here, instead, we're using "::" to access the TYPE OF THE ELEMENTS THAT THE CONTAINER IS STORING. To help the compiler we add "typename".

It can be VALUE TYPE or SIZE TYPE (where size type is an index)

base_type of the container named "key_container" ! (= type of each element of the container)

(columns names)
(data, equivalent to vector of double)
(single name (one column's name), this is eq. to "string")
(equivalent to "double")
(size_type is the type of the index of, in this case, the matrix)

(vector of strings)
(vector of columns (one column is a mapped_container))

Since we're dealing with references we must add const to obtain const methods!

```cpp
//-----------------------------------------------------------------------------
// DataFrame.cpp
//-----------------------------------------------------------------------------
#include "DataFrame.hpp"

DataFrame::DataFrame (const std::string & c_names): DataFrame (split (c_names, ' ')){}

DataFrame::DataFrame (const key_container & names):
    df_keys (names), df_values (df_keys.size ()), added_first_column (false){}

DataFrame::size_type DataFrame::look_up (const key_type & key) const{
    size_type index = 0;
    while (index < df_keys.size () and df_keys[index] != key) ++index;
    return index;
}

DataFrame::mapped_container DataFrame::get_column (const key_type & column_name) const{
    const size_type index = look_up (column_name);
    return df_values[index];
}

double DataFrame::get_mean (const key_type & column_name) const{
    const size_type index = look_up (column_name);
    const mapped_container & column = df_values[index];
    double sum = 0.;
    for (const mapped_type & v : column)
        sum += v;
    return sum / column.size ();
}

void DataFrame::print (void) const{
    for (size_type i = 0; i < df_keys.size (); ++i){
        std::cout << df_keys[i] << " :: ";
        for (const mapped_type & v : df_values[i])
            std::cout << v << " ";
        std::cout << std::endl;
    }
}

DataFrame::key_container DataFrame::split (const std::string & s, char delim) const{
    std::string word;
    key_container keys;
    std::istringstream columns (s);
    while (std::getline (columns, word, delim)) keys.push_back (word);
    return keys;
}

const DataFrame::mapped_type& DataFrame::get_element_at (const key_type & column_name, size_type index) const{
    const size_type column_index = look_up (column_name);
    return df_values[column_index][index];
}

void DataFrame::set_element_at (const key_type & column_name, size_type index, const mapped_type & value){
    const size_type column_index = look_up (column_name);
    df_values[column_index][index] = value;
}

void DataFrame::set_column (const key_type & column_name, const mapped_container & column_data){
    const size_type index = look_up (column_name);
    if (added_first_column){
        mapped_container & column = df_values[index];
        for (size_type i = 0; i < column_data.size (); ++i)
            column[i] = column_data[i];
    }
    else{
        added_first_column = true;
        df_values[index] = column_data;
        for (mapped_container & column : df_values)
            column.resize (column_data.size ());
    }
}

DataFrame DataFrame::select_equal (const key_type & c_name, const mapped_type & value) const{
    DataFrame result (df_keys);
    const size_type index = look_up (c_name);
    const mapped_container & column = df_values[index];
    std::vector<size_type> indices;

    // select rows indices satisfying the selection criterion
    for (size_type j = 0; j < column.size (); ++j)
        if (column[j] == value)
            indices.push_back (j);
    for (size_type i = 0; i < df_keys.size (); ++i){
        mapped_container values;
        key_type const & current_name = df_keys[i];
        mapped_container const & current_column = df_values[i];
        for (size_type j : indices)
            values.push_back (current_column[j]);
        result.set_column (current_name, values);
    }
    return result;
}
```

this constructor relies on

false because we first create an empty structure

we can access because in the constructor we created an empty structure

once we put the first column we know how many elements will have EACH column ⇒ we prepare the structure for all the other columns to store the same number of elements

```
//---------------------------------------------------------------------------
// Solution 2  (associative containers version)
//---------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------
#include <iostream>
#include <vector>
#include "DataFrame.h"

int main(){
    std::cout << "Test DataFrame" << std::endl;
    DataFrame df ("c1 c2");
    /*    df.test_check("c1");
          df.test_check("c2");
          df.test_check("c3");
    */
    df.print();
    std::vector<double> v1 = {1, 2, 3};
    std::vector<double> v2 = {4, 5, 6};
    df.set_column ("c1", v1);
    df.set_column ("c2", v2);
    df.set_column ("c2", v1);
    df.set_column ("c3", v1);
    df.print();
    std::vector<double> v_res = df.get_column ("c1");
    std::cout << "Printing v_res" << std::endl;

    for (size_t i = 0; i < v_res.size(); i++)
        std::cout << v_res[i] << " ";

    std::cout << std::endl;
    v_res = df.get_column ("c3");
    std::cout << "Printing v_res" << std::endl;

    for (size_t i = 0; i < v_res.size(); i++)
        std::cout << v_res[i] << " ";

    std::cout << std::endl;
    std::cout << df.get_mean ("c1") << std::endl;
    std::cout << df.get_mean ("c2") << std::endl;
    df.set_column ("c2", v2);
    std::vector<double> v3 = {4, 5};
    df.set_column ("c2", v3);
    std::cout << df.get_mean ("c2") << std::endl;
    std::cout << df.get_mean ("c3") << std::endl;
    std::cout << df.get_element_at ("c2", 1) << std::endl;
    std::cout << df.get_element_at ("c2", 4) << std::endl;
    std::cout << df.get_element_at ("c3", 1) << std::endl;
    df.set_element_at ("c2", 1, 27);
    std::cout << df.get_element_at ("c2", 1) << std::endl;
    df.add_column ("c3", v1);
    std::cout << "df after add" << std::endl;
    df.print();

    DataFrame select_df = df.select_equal("c2", 1);
    std::cout << "select_df " <<  std::endl;
    select_df.print();

    return 0;
}
```

```
//-----------------------------------------------------------------------------------------------------------------
// DataFrame.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef DATAFRAME_H
#define DATAFRAME_H

#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>

class DataFrame{

    public:
        typedef std::unordered_map<std::string, std::vector<double> > df_type;
        typedef typename df_type::value_type value_type;
        typedef typename df_type::key_type key_type;
        typedef typename df_type::mapped_type mapped_type;
        typedef typename mapped_type::size_type size_type;

    private:
        df_type df_data;

        // returns true if column_name is valid (included in the constructor list)
        bool check_column_name (const key_type & column_name) const;

        // return the names of columns in s separated by delim
        std::vector<key_type> split (const key_type & s, char delim) const;

        bool first_column_added;
        size_type n_rows;

        // set a column data checking only right number of rows
        void set_column_data (const key_type & column_name, const mapped_type & column_data);

    public:
        DataFrame (const key_type & c_names);
        DataFrame();
        DataFrame & operator= (const DataFrame &) = default;
        void print (void) const;
        void set_column (const key_type & column_name, const mapped_type & column_data);
        mapped_type get_column (const key_type & column_name) const;
        double get_element_at (const key_type & column_name, size_type index) const;
        void set_element_at (const key_type & column_name, size_type index, double value);
        double get_mean (const key_type & column_name) const;

        // add a new column with data
        void add_column (const key_type & column_name, const mapped_type & column_data);

        // return a copy of the DataFrame with rows such that the c_name = value
        DataFrame select_equal (const key_type & c_name, double value) const;
};

#endif //DATAFRAME_H


//-----------------------------------------------------------------------------------------------------------------
// DataFrame.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "DataFrame.h"

DataFrame::DataFrame (const key_type & c_names): DataFrame (){
    std::vector<key_type> columns_names = split (c_names, ' ');
    for (const key_type & name : columns_names)
        df_data[name];
}

DataFrame::DataFrame (void): first_column_added (false), n_rows (0) {}

bool DataFrame::check_column_name (const key_type & column_name) const{
    return df_data.find (column_name) != df_data.end();
}

void DataFrame::set_column (const key_type & column_name, const mapped_type & column_data){
    if (check_column_name (column_name))
        set_column_data (column_name, column_data);
    else
        std::cerr << "Error, " << column_name << " is unknown" << std::endl;
}

DataFrame::mapped_type DataFrame::get_column (const key_type & column_name) const{
    if (check_column_name (column_name))
        return df_data.at (column_name);
    else{
        std::cerr << "Error, " << column_name << " is unknown" << std::endl;
        return std::vector<double> ();
    }
}
```

Do we need ordered?
what if we use a map instead? Every time we access a column we pay O(log(n)) instead of paying O(1) → nonsense since we don't need the ordering!

we're not interested in order

the key is the name of the column (the value is the column)

(number of rows in each columns)

relies on

relies on another function (later)

set an existing column (modify a column)

we cannot do: return df_data[column_name];
why? because the method is const and the operation is implemented s.t. if the column ∄ then it'll be added
→ we have to use .at(..)

empty vector

```cpp
double DataFrame::get_mean (const key_type & column_name) const{
    if (check_column_name (column_name)){
        double sum = 0.;
        for (double d : df_data.at (column_name))
            sum += d;
        return sum / n_rows;
    }
    else{
        std::cerr << "Error, " << column_name << " is unknown" << std::endl;
        return std::numeric_limits<double>::quiet_NaN();
    }
}

void DataFrame::print (void) const{
    for (const value_type & element : df_data){
        std::cout << element.first << " :: ";
        for (const double & d : element.second)
            std::cout << d << " ";
        std::cout << std::endl;
    }
}

std::vector<DataFrame::key_type> DataFrame::split (const key_type & s,  char delim) const{
    key_type word;
    std::vector<key_type> v;
    std::istringstream columns (s);
    while (std::getline (columns, word, delim)) v.push_back (word);
    return v;
}

double DataFrame::get_element_at (const key_type & column_name, size_type index) const{
    if (check_column_name (column_name))
        if (index < n_rows)
            return df_data.at (column_name)[index];
        else{
            std::cerr << "Error, index out of bound" << std::endl;
            return std::numeric_limits<double>::quiet_NaN();
        }
    else{
        std::cerr << "Error, " << column_name << " is unknown" << std::endl;
        return std::numeric_limits<double>::quiet_NaN();
    }
}

void DataFrame::set_element_at (const key_type & column_name, size_type index, double value){
    if (check_column_name (column_name))
        if (index < n_rows)
            df_data[column_name][index] = value;
        else
            std::cerr << "Error, index out of bound" << std::endl;
    else
        std::cerr << "Error, " << column_name << " is unknown" << std::endl;
}

void DataFrame::add_column (const key_type & column_name, const mapped_type & column_data){
    if (! check_column_name (column_name))
        set_column_data (column_name, column_data);
    else
        std::cerr << "Error, " << column_name << " is already included in the DataFrame" << std::endl;
}

void DataFrame::set_column_data (const key_type & column_name, const mapped_type & column_data){
    if (! first_column_added){
        n_rows = column_data.size();
        first_column_added = true;
        for (df_type::iterator it = df_data.begin(); it != df_data.end(); it++){
            (it->second).resize(n_rows);
        }
        df_data[column_name] = column_data;
    }
    else{
        // for next assignments check the column has the same number of rows
        if (n_rows == column_data.size())
            df_data[column_name] = column_data;
        else{
            std::cerr << "Error, " << column_name << " has a different number of rows" << std::endl;
        }
    }
}
```

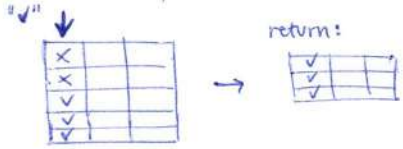
add a new column

!!

```
DataFrame DataFrame::select_equal (const key_type & c_name, double value) const{
    if (! check_column_name (c_name)){
        std::cerr << "Error, " << c_name << " is unknown" << std::endl;
        return DataFrame();
    }
    std::vector<size_type> indexes; // solution indexes
    const mapped_type & relevant_column = df_data.at (c_name);
    // select rows indexes satisfying the selection criterion
    for (size_type i = 0; i < n_rows; ++i)
        if (relevant_column[i] == value)
            indexes.push_back (i);
    DataFrame result;
    for (const value_type & element : df_data){
        mapped_type v; //column values
        v.reserve (indexes.size ());
        // select proper column values
        for (size_type idx : indexes)
            v.push_back (element.second[idx]);
        result.add_column (element.first, v);
    }
    return result;
}
```

column that we're inspecting

receives the name of a given column and a value: it looks in all the dataframe for all rows (in the given column) that have that value



# Streams & I/O : word transformation

```
#include <iostream>
#include <fstream>      file stream
#include <sstream>
#include <string>
#include <utility>
#include <vector>

using namespace std;

const string transform(const string &s, const vector<pair<string, string>> &m) {
    // comparing the input string against all our dictionary entries, the first
    // matching key performs the substitution
    for (pair<string, string> mapping : m) {
        if (!get<0>(mapping).compare(s))
            return get<1>(mapping);
    }
    return s;
}

vector<pair<string, string>> buildVector(ifstream &vec_file) {
    vector<pair<string, string>> trans_vec;   — create a vector of pairs of strings
    string key;
    string value;

    // parse the translations file
    while (vec_file >> key && getline(vec_file, value))   — we store the first word in "key" and whatever comes next (one or more words) in the string "value"
        if (value.size() > 1)
            trans_vec.push_back(make_pair(key, value.substr(1))); // skip the leading space
        else
            cout << "no rule for " + key << endl;

    // print the translations vector
    for (pair<string, string> p : trans_vec)
        cout << "Rule: " << get<0>(p) << " -> " << get<1>(p) << endl;
    return trans_vec;
}

void word_transform(ifstream &map_file, ifstream &input) {
    auto trans_vec = buildVector(map_file);
    string text;
    while (getline(input, text)) {
        istringstream stream(text);
        string word;
        bool firstWord = true;
        while (stream >> word) {
            if (firstWord)
                firstWord = false;
            else
                cout << " ";
            cout << transform(word, trans_vec);
        }
        cout << endl;
    }
}

int main() {
    // Open our dictionary of replacements.   — this contains the dictionary :
    ifstream translations{"../translations.txt"};
    if (!translations) {
        cout << "The file translations.txt could not be opened!" << endl;
        return 1;
    }
    // Open our input file.   — this contains the text we want to convert
    ifstream in_file{"../shortText.txt"};
    if(!in_file) {
        cout << "The file shortText.txt could not be opened!" << endl;
        return 2;
    }
    word_transform(translations, in_file);
    return 0;
}
```

since the space is counted, for the transl. to be present we need at least value.size() = 2 :

| y | | w | h | y |
|---|---|---|---|---|

this goes to "key" — this goes to "value": value[0] = " ", value[1] = "why";

alternatively: cout << p.first << " " << p.second << endl;

only for nice printing

translations.txt

| | |
|---|---|
| y | why |
| r | are |
| u | you |

Definition of "<" for sets

```
//-----------------------------------------------------------------------------------------------------------------
// main.hpp
//-----------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <set>
#include "Sales_data.h"

int main() {
    Sales_data s1("01",1,5);
    Sales_data s2("02");
    Sales_data s3;
    std::multiset<Sales_data> bookstore; // rely on operator < definition
    bookstore.insert(s1);
    bookstore.insert(s2);
    bookstore.insert(s3);
    for (const  Sales_data & sd : bookstore)
        sd.print();
    return 0;
}
```

```
//-----------------------------------------------------------------------------------------------------------------
// Sales_data.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef SALES_DATA_H
#define SALES_DATA_H

#include <iostream>

class Sales_data {

    public:
        // non-delegating constructor initializes members from corresponding arguments
        Sales_data(std::string s, unsigned cnt, double price): bookNo(s), units_sold(cnt), revenue(cnt*price){}

                // remaining constructors all delegate to another constructor
        Sales_data() : Sales_data("", 0, 0) {}
        Sales_data(std::string s): Sales_data(s, 0,0) {}

        std::string isbn() const { return bookNo; }
        void print () const;

    private:
        std::string bookNo ;
        unsigned units_sold = 0 ;
        double revenue = 0 ;
};

bool operator < (const Sales_data & lhs, const Sales_data & rhs);

#endif // SALES_DATA_H
```

```
//-----------------------------------------------------------------------------------------------------------------
// Sales_data.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "Sales_data.h"

void Sales_data::print () const{
    std::cout << "ISBN: " << bookNo << " unit solds: " << units_sold << " revenue: " << revenue <<std::endl;
}

bool operator < (const Sales_data &lhs, const Sales_data &rhs){
    return lhs.isbn() < rhs.isbn();
}
```

Recap on **sequential containers**

```cpp
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <forward_list>

using std::vector;
using std::deque;
using std::list;
using std::forward_list;
using std::cout;
using std::endl;
void print(const std::vector<int> & v);
void print(const std::deque<int> & d);
void print(const std::list<int> & l);
void print(const std::forward_list<int> & fl);

int main(){
    std::vector<int> v;
    std::deque<int> d;
    std::list<int> l;
    std::forward_list<int> fl;

    // all containers are empty
    // let's store 1,2,3

    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    l.push_back(2);
    l.push_back(3);
    l.push_front(1); // it is efficient, in a vector it is not possible  (no push_front)
                     // (it would be inefficient). Same considerations for forward_list

    fl.push_front(3); // fl doesn't have push_back, push_front in reverse order
    fl.push_front(2);
    fl.push_front(1);

    // for deque we can do same, let's go in reverse order to enjoy efficient push_front!
    d.push_front(3);
    d.push_front(2);
    d.push_front(1);

    // writing  elements to cout (read only)
    cout << "Print v, d, l, fl" << endl;

    print(v);
    print(d);
    print(l);
    print(fl);

    // let's change first element through reference fo first element
    vector<int>::reference rv1=v.front();
    rv1++;
    deque<int>::reference rd1=d.front();
    rd1++;
    list<int>::reference rl1=l.front();
    rl1++;
    forward_list<int>::reference rfl1=fl.front();
    rfl1++;

    // let's change last element through reference fo last element
    // we cannot do for forward_list!
    vector<int>::reference rv2=v.back();
    rv2++;
    deque<int>::reference rd2=d.back();
    rd2++;
    list<int>::reference rl2=l.back();
    rl2++;

    cout << "Print v, d, l, fl" << endl;
    print(v);
    print(d);
    print(l);
    print(fl);

    // let's print directly list first element
    cout << "Print l first element" << endl;
    list<int>::value_type i = l.front(); // i is a copy of the first element!
    cout << i << endl;
    // Copies
    vector<int> v2(v); //same as vector<int> v2 = v;
    deque<int> d2(v.cbegin(),v.cend());
```

```
    cout << "Print v2" << endl;
    print(v2);

    cout << "Print d2" << endl;
    print(d2);

    list<int> l2;
    l2.assign(l.cbegin(), l.cend());

    cout << "Print l2" << endl;
    print(l2);

    // Let's copy v to the end of v2
    v2.insert(v2.end(),v.cbegin(),v.cend());
    cout << "Print v2" << endl;
    print(v2);

    // Let's copy v at the beginning of d2
    d2.insert(d2.begin(),v.cbegin(),v.cend());
    cout << "Print d2" << endl;
    print(d2);

    // Let's copy 1, 2, 3 at the beginning of l2 and 5,6 at its end
    l2.insert(l2.begin(),{1,2,3});
    l2.insert(l2.end(),{5,6});
    cout << "Print l2" << endl;
    print(l2);

    // Let's resize d and v in a way they have 10 elements
    // (set elements to 30 in the second case)
    d.resize(10);
    v.resize(10,30);
    cout << "Print d" << endl;
    print(d);
    cout << "Print v" << endl;
    print(v);

    // delete all element from back
    while (!v.empty())
        v.pop_back();

    // delete all element from front
    while (!d.empty())
        d.pop_front();

    // delete all elements with iterator limits
    l.erase(l.begin(),l.end());

    // delete all elements through clear
    fl.clear();

    // All others are deleted through destructors ;)!!!
    return 0;
}

// if you want to change elements rely on iterator instead of const_iterator, but same loops!
void print(const std::vector<int> & v){
    for(vector<int>::const_iterator it = v.cbegin(); it != v.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

// for deque, list and forward_list is same!!!
void print(const std::deque<int> & d){
    for(deque<int>::const_iterator it = d.cbegin(); it != d.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

void print(const std::list<int> & l){
    for(list<int>::const_iterator it = l.cbegin(); it != l.cend(); it++)
        cout << *it << " ";
    cout << endl;
}

void print(const std::forward_list<int> & fl){
    for(forward_list<int>::const_iterator it = fl.cbegin(); it != fl.cend(); it++)
        cout << *it << " ";
    cout << endl;
}
```

equivalent to l2 = l;

first we provide the iterator where we want to insert

# Recap on associative containers

```cpp
#include <iostream>
#include <map>
#include <unordered_map>
#include <utility>

using std::map;
using std::unordered_map;
using std::cout;
using std::endl;
using std::pair;
using std::string;
void print(const map<int, string> & m);
void print(const unordered_map<int, string> & um);

int main(){

    // Student ID and name
    map<int, string> m;
    unordered_map<int, string>  um;

    // all containers are empty
    // let's store
    //  1, "Elem 1"
    //  2, "Elem 2"
    //  4, "Elem 4"
    m[1] = "Elem 1";
    m[2] = "Elem 2";
    m[4] = "Elem 4";

    // Let's copy m to um
    um.insert(m.cbegin(),m.cend());

    // writing  elements to cout (read only)
    cout << "Print m" << endl;
    print(m);
    cout << "Print um" << endl;
    print(um);

    // Let's try to insert again 4
    cout << "Trying to insert 4 again" << endl;
    m.insert(std::make_pair(4,"New val for 4"));
    um.insert(std::make_pair(4,"New val for 4"));

    cout << "Print m" << endl;
    print(m);
    cout << "Print um" << endl;
    print(um);

    // Let's change elem 4!
    cout << "Changing 4" << endl;
    m[4] = "New val for 4";
    um[4] = "New val for 4";

    cout << "Print m" << endl;
    print(m);
    cout << "Print um" << endl;
    print(um);

    // Let's look for 1 and print
    if (m.find(1)!=m.end())     {
        cout << "Yeah!  1 is here and it's element is: " << m[1] << endl;
    }
    if (um.find(1)!=um.end())   {
        cout << "Yeah!  1 is here and it's element is: "<< um[1] << endl;
    }

    // Let's look for 5 and print
    if (m.find(5) == m.end())   {
        cout << "Unfortunately 5 is not here "<< endl;
    }
    if (um.find(5)==um.end())   {
        cout << "Unfortunately 5 is not here "<< endl;
    }

    // Try to access elem 5!  We are inserting empty string
    cout << m[5] << endl;
    cout << um[5] << endl;
    cout << "Print m" << endl;
    print(m);
    cout << "Print um" << endl;
    print(um);
```

we cannot do `um = m;` because `um` is an unordered map. If it was an ordered map we would be able to do it

we could also do: `um.insert({1,"Elem 1"4}); ..`
or, otherwise:

nothing happens: **no** key repetition
(both ordered and unordered maps can store only one instance of each object with a given key)

} we're adding elements with key = 5 and values default initialized

```
    /*
    // Try to access elem 6!  This will raise an exception
    cout << m.at(6) << endl;
    cout << um.at(6) << endl;
    */

    // delete all elements
    m.clear();

    // All um are deleted through destructors ;)!!!
    return 0;
}

// if you want to change elements rely on iterator instead of const_iterator, but same loops!
void print(const map<int, string> & m){
    for(map<int, string>::const_iterator it = m.cbegin(); it != m.cend(); it++)
        cout << it->first << " " << it->second  << endl;
}

// same for unoredered_map and not very different from, e.g., vectors!
void print(const unordered_map<int, string> & um){
    for(unordered_map<int, string>::const_iterator it = um.cbegin(); it != um.cend(); it++)
        cout << it->first << " " << it->second  << endl;
}
```

it returns the value if the element exists, otherwise we get an error!

# Map

```cpp
#include <map>

class wrapper{
    public:
        const unsigned int content;
        wrapper(unsigned int _content): content(_content){}
};

int main(){
    std::map<unsigned int, wrapper>  my_map;
    wrapper my_wrapper(1);
    my_map[1] = my_wrapper;
    return 0;
}
```

❌ we cannot do this

: doing "my_map[1]" we've creating a pair key-value, where the key is **1** and the value is STANDARD INITIALIZED, then we're doing the assignment with "= my_wrapper;".

```cpp
//-------------------------------------------------------------------------------------------
// Initial version introduces two compilation errors:
//   1. synthesized copy assignment operator cannot be used because of const member
//   2. empty constructor is not provided
//-------------------------------------------------------------------------------------------

#include <map>

class wrapper{
    public:
        unsigned int content;
        wrapper(unsigned int _content): content(_content){}
        wrapper() = default;
};

int main(){
    std::map<unsigned int, wrapper>  my_map;
    wrapper my_wrapper(1);
    my_map[1] = my_wrapper;
    return 0;
}
```

The point is: there is no default constructor, moreover we cannot perform the assignment because of the "const"

When we define a map, the value part **MUST** HAVE A DEFAULT CONSTRUCTOR

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Instant Messenger

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it




## Goal

- Design and implement a **class** for an **instant messaging** service.
- The class owns a data structure to store both messages (`std::string`) and their sending times (integral type).
- The class should offer **three methods**:
  - a procedure to **send messages**, which takes the sending time and text as arguments (and return nothing)
  - a function to **receive the latest content**, which takes a timestamp as argument and returns a collection of all the messages more recent than that moment
  - a function that **returns all the messages whose content includes a given word** provided as argument



## Assumptions and Requests

- Consider as **most common use case** clients that remain online quite often…
- …with **frequent sends and receives, but rare searches based on message content**
- State the **complexity** of the implemented methods and motivate your design choices, particularly regarding data structures

```cpp
//---------------------------------------------------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "instant_messenger.h"

int main (){
    im::Messenger msg;
    msg.send(0, "I'm coming home");
    msg.send(1, "What's for dinner?");
    msg.send(3, "Sorry, I'm late");

    /* YOUR CODE GOES HERE */ r_msg = msg.receive(1);
    std::cout << "-------------------- receive --------------------" << std::endl;
    for (const im::Messenger::mapped_type& m : r_msg)
        std::cout << m << std::endl;

    std::cout << std::endl;

    /* YOUR CODE GOES HERE */ s_msg = msg.search("home");
    std::cout << "-------------------- search --------------------" << std::endl;
    for (const im::Messenger::mapped_type& m : s_msg)
        std::cout << m << std::endl;

    return 0;
}
```

```cpp
//---------------------------------------------------------------------------------------------------------------------
// instant_messenger.h
//---------------------------------------------------------------------------------------------------------------------
#ifndef INSTANT_MESSENGER_H
#define INSTANT_MESSENGER_H

#include <list>
#include <map>
#include <string>

namespace im {

    class Messenger {

        private:
            typedef std::map<unsigned long, std::string> container_type;
            container_type m_data;

        public:
            typedef container_type::key_type key_type;
            typedef container_type::mapped_type mapped_type;
            typedef container_type::value_type value_type;
            void send (key_type, const mapped_type&);
            std::list<mapped_type> receive (key_type) const;
            std::list<mapped_type> search (const std::string &) const;
    };
}

#endif //INSTANT_MESSENGER_H
```

map for <time, message> : we do not allow messages at the same time

map or unordered_map? We want to simplify (optimize) the method "receive", which takes into account all the messages received after a given time → it's useful to have an ordered structure

(unsigned long)

(string)

(pair(unsigned long, string))

since we only want to add elements this is the best option : O(1) (vectors have O(n))

```cpp
//---------------------------------------------------------------------------------------------------------------------
// instant_messenger.cpp
//---------------------------------------------------------------------------------------------------------------------
#include "instant_messenger.h"

namespace im{
    O(log(n)) , n = # elements in the map
    void messenger::send (key_type timestamp, const mapped_type & message){
        m_data[timestamp] = message;
    }
    O(log(n)) / O(n)   (in general O( log(n) + l ))
    std::list<messenger::mapped_type> messenger::receive (key_type timestamp) const{
        std::list<mapped_type> result;
        for (container_type::const_iterator it = m_data.lower_bound (timestamp); it != m_data.cend (); ++it){
            result.push_back (it -> second);
        }
        return result;
    }
    O(n)
    std::list<messenger::mapped_type> messenger::search (const std::string & word) const{
        std::list<mapped_type> result;
        for (const container_type::value_type & pair : m_data){
            const mapped_type & message = pair.second;
            if (message.find (word) != mapped_type::npos){
                result.push_back (message);
            }
        }
    return result;
    }
}
```

* here we don't need to add: "messenger::" because we're already in the class

O(log(n)) — iterator to the element which key is ≥ timestamp

it receives a word and returns all the messages that contain the word

method find() of strings : it returns the index of the first character of word inside the string if the word is found, otherwise it returns "npos" (invalid index)

deep se
dependable evolvable pervasive software engineering group

# Lab Session: STL containers

Luca Pozzoni, Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO



## The Problem

- Write a program that manages the list of players of an arcade video-game and their scores. Create a function that can find and return a player with a given username
- As a starting point, you are provided with the implementation based on vector to collect players. Also a binary_search function has been implemented
- In the provided example, the search is profiled in case the player is in the vector or he/she is not
- Implement your solution once by using a set (ordered) to collect players, and another time using map (unordered)



## The Problem

- Add to search.cc:
  - bool set_search (
    const std::set<Arcade::Player> & player_set,
    unsigned username );
  - bool map_search (
    const std::unordered_map<unsigned, Arcade::Player> &
    player_map,
    unsigned username);
- Modify the initial vector-based solution in a way that you rely on the STL binary_search
  - bool binary_search(
    player_vec_type::const_iterator first,
    player_vec_type::const_iterator last,
    const Arcade::Player & value);

# Solution - 1

1) Write a class for players. Each player has a username and a vector of scores. In your main function, define a collection of players.

```
typedef std::set<Player> player_set_type;
player_set_type player_set;
```

Hint: you need to overload operator< for class Player. Why?!

```
bool operator<(const Player &lhs, const Player &rhs)
```

2) Try inserting some new elements in your set, using container specific insert.

```
player_set.insert(random_player);
```

3) Define a search function that takes the collection of players and a username and looks for username within the container.

```
bool set_search(const player_set_type & stud_set, unsigned username)
```



# Solution - 2

Do the same process by using
an unordered_map<id, player> instead of a set.

```
typedef std::unordered_map<unsigned, Player> player_map_type;
player_map_type player_map;

player_map.insert(make_pair(random_player.getUsername(),random_player));

bool map_search(const player_map_type & player_map, unsigned username);
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "RandomPlayerGenerator.hh"
#include "search.hh"
#include "Player.hh"
#include "timing.hh"

using std::string;
typedef std::vector<Arcade::Player> player_vec_type;
typedef std::set<Arcade::Player> player_set_type;
typedef std::unordered_map<unsigned, Arcade::Player> player_map_type;

int main(){
    constexpr unsigned n_player = 1000000;
    player_vec_type player_vec;
    player_set_type player_set;
    player_map_type player_map;

    Arcade::RandomPlayerGenerator rs;

    // create the vector with playerents sorted by username
    for (unsigned i = 0; i < n_player; ++i){
        Arcade::Player random_player = rs.nextPlayer();
        player_vec.push_back (random_player);
        player_set.insert(random_player);
        player_map.insert(std::pair<unsigned, Arcade::Player>(random_player.getUsername(), random_player));
    }

    const unsigned player_ok_username = player_vec[player_vec.size() / 2].getUsername();
    const unsigned player_ko_username = player_vec.back().getUsername() + 1;

    std::cout << "*********initial binary search on vector*********\n";
    std::cout << (Arcade::binary_search (player_vec, player_ok_username) ? "Found!\n" : "Not found\n");
    std::cout << (Arcade::binary_search (player_vec, player_ko_username) ? "Found!\n" : "Not found\n");
    std::cout << "*********search on set*********\n";
    std::cout << (Arcade::set_search (player_set, player_ok_username) ? "Found!\n" : "Not found\n");
    std::cout << (Arcade::set_search (player_set, player_ko_username) ? "Found!\n" : "Not found\n");
    std::cout << "*********search on map*********\n";
    std::cout << (Arcade::map_search (player_map, player_ok_username) ? "Found!\n" : "Not found\n");
    std::cout << (Arcade::map_search (player_map, player_ko_username) ? "Found!\n" : "Not found\n");

    return 0;
}


//-----------------------------------------------------------------------------------------------------------------------
// timing.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef TIMING_HH
#define TIMING_HH

#include <chrono>

namespace timing{
    typedef std::chrono::time_point<std::chrono::system_clock> time_point;
    void elapsed_between (const time_point & start, const time_point & finish);
}

#endif // TIMING_HH


//-----------------------------------------------------------------------------------------------------------------------
// timing.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <ctime>
#include <iostream>

#include "timing.hh"

namespace timing{
    void elapsed_between (const time_point & start, const time_point & finish){
        std::chrono::duration<double> elapsed_seconds = finish - start;
        std::time_t end_time = std::chrono::system_clock::to_time_t (finish);
        std::cout << "finished computation at " << std::ctime (&end_time)
              << "elapsed time: " << elapsed_seconds.count() << " s" << std::endl;
    }
}
```

```
//-----------------------------------------------------------------------------------------------------------------
// Player.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef PLAYER_H_
#define PLAYER_H_

#include <vector>

namespace Arcade{

    class Player{
        unsigned username;
        std::vector<unsigned> scores;

        public:
            Player (unsigned, const std::vector<unsigned> &);
            explicit Player (unsigned);
            const std::vector<unsigned> & getScores() const;
            unsigned getUsername() const;
            void setScores (const std::vector<unsigned> &);
            void setUsername (unsigned);
            void print() const;
    };

    inline bool operator<(const Arcade::Player& lhs, const Arcade::Player& rhs){
        return (lhs.getUsername() < rhs.getUsername());
    }
}

//inline bool operator<(const Arcade::Player& rhs, const Arcade::Player& lhs){
//    return (rhs.getUsername() < lhs.getUsername());
//}

#endif /* PLAYER_H_ */


//-----------------------------------------------------------------------------------------------------------------
// Player.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "Player.hh"

namespace Arcade{

    Player::Player (unsigned username, const std::vector<unsigned> & scores):
        username (username), scores (scores) {}

    Player::Player (unsigned username): username (username), scores () {}

    const std::vector<unsigned> & Player::getScores() const{
        return scores;
    }

    unsigned Player::getUsername() const{
        return username;
    }

    void Player::setScores (const std::vector<unsigned> & new_scores){
        scores = new_scores;
    }

    void Player::setUsername (unsigned new_username){
        username = new_username;
    }

    void Player::print() const{
        std::cout << username << ":\n";
        for (unsigned score : scores)
            std::cout << score << " ";
        std::cout << std::endl;
    }
}

//bool operator<(const Arcade::Player& x, const Arcade::Player& y) {
//    return (x.getUsername() < y.getUsername());
//}
```

```
//------------------------------------------------------------------------------------------------------------------------
// RandomPlayerGenerator.h
//------------------------------------------------------------------------------------------------------------------------
#ifndef RANDOMPLAYERGENERATOR_H_
#define RANDOMPLAYERGENERATOR_H_

#include <random>

namespace Arcade{

    class Player;

    class RandomPlayerGenerator{
        unsigned next_username;
        std::mt19937 generator;
        std::uniform_int_distribution<unsigned> score_distribution;
        std::uniform_int_distribution<unsigned> number_distribution;

        public:
            RandomPlayerGenerator ();
            RandomPlayerGenerator (unsigned first_username, unsigned lowest_score, unsigned highest_score,
                                   unsigned n_scores, unsigned seed = 0u);
            Player nextPlayer();
    };
}

#endif /* RANDOMPLAYERGENERATOR_H_ */


//------------------------------------------------------------------------------------------------------------------------
// RandomPlayerGenerator.cpp
//------------------------------------------------------------------------------------------------------------------------
#include <vector>
#include "RandomPlayerGenerator.hh"
#include "Player.hh"

namespace Arcade{

    RandomPlayerGenerator::RandomPlayerGenerator (): RandomPlayerGenerator (1, 18, 30, 20) {}

    RandomPlayerGenerator::RandomPlayerGenerator (unsigned first_username, unsigned lowest_score,
                                                  unsigned highest_score, unsigned n_scores, unsigned seed):
        next_username (first_username), generator (seed), score_distribution (lowest_score, highest_score),
        number_distribution (1, n_scores) {}

    Player RandomPlayerGenerator::nextPlayer(){
        const unsigned n_scores = number_distribution (generator);
        std::vector<unsigned> scores (n_scores);
        for (unsigned & score : scores)
            score = score_distribution (generator);
        return Player (next_username++, scores);
    }
}
```

```cpp
//-----------------------------------------------------------------------------------------------------------------------
// search.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef SEARCH_HH
#define SEARCH_HH

#include <vector>
#include <string>
#include <vector>
#include <utility>
#include <set>
#include <unordered_map>
#include "Player.hh"

namespace Arcade{
    bool binary_search (const std::vector<Arcade::Player> & player_vec,  unsigned player_username);
    bool set_search (const std::set<Arcade::Player> & player_set,  unsigned player_username);
    bool map_search (const std::unordered_map<unsigned,Arcade::Player> & player_map,  unsigned player_username);
}

#endif // SEARCH_HH


//-----------------------------------------------------------------------------------------------------------------------
// search.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include "search.hh"
#include "timing.hh"

namespace Arcade{

  • bool binary_search (const std::vector<Arcade::Player> & player_vec, unsigned player_username){
        timing::time_point start = std::chrono::system_clock::now();
        std::vector<Arcade::Player>::const_iterator begin = player_vec.cbegin(), end = player_vec.cend(),
                // original musernamepoint
        musername = begin + (end - begin) / 2;
        unsigned n_iter = 1;

        while (begin != end and musername->getUsername() != player_username){
        // end is meant to be invalusername, so in both cases we are ignoring musername at the following iteration
            if (player_username < musername->getUsername())
                end = musername;
            else
                begin = musername + 1;
            musername = begin + (end - begin) / 2;
            ++n_iter;
        }
        std::cout << "Number of Iterations " << n_iter << "\n";
        timing::time_point finish = std::chrono::system_clock::now();
        timing::elapsed_between (start, finish);
        if (musername != player_vec.cend() and player_username == musername->getUsername()){
            return true;
        }
        else{
            return false;
        }
    }

  • bool set_search (const std::set<Arcade::Player> & player_set,  unsigned player_username){
        timing::time_point start = std::chrono::system_clock::now();
        bool val = player_set.count(Player(player_username));
        timing::time_point finish = std::chrono::system_clock::now();
        timing::elapsed_between (start, finish);
        return val;
    }

  • bool map_search (const std::unordered_map<unsigned,Arcade::Player> & player_map,  unsigned player_username){
        timing::time_point start = std::chrono::system_clock::now();
        bool val = player_map.count(player_username);
        timing::time_point finish = std::chrono::system_clock::now();
        timing::elapsed_between (start, finish);
        return val;
    }
}
```

deep se
dependable evolvable pervasive software engineering group

# Lab Session: Streams

Giovanni Quattrocchi, Luca Pozzoni

Politecnico di Milano




## The Problem

**Goal:** to provide a program that reads from different files information about students and their exams and stores them in a suitable data structure.

**Students** are characterized by name, surname, birth date and a `vector` of exams. Moreover, students are uniquely identified by an ID.

**Exams** are represented by the ID of the course, the date and the grade.

The correspondence between students and exams is expressed through the students' ID.



## The Problem

The list of students is provided in file `students.txt`, which stores in each line the student's ID, his name, surname and birth date. Fields are separated by commas, i.e. `students.txt` is a Comma Separated Value (CSV) file.

**Example:**

```
100100,Mario,Rossi,8/6/1991
```

The list of exams is provided in file `exams.txt`, which stores in each line the student's ID, the course's ID, the date and the grade. Fields are separated by commas.

**Example:**

```
100100,101,8/5/2019,28
```



## The Problem

Data coming from `students.txt` and `exams.txt` must be stored in a

```
unordered_map<unsigned,StudentsData>
```

denoted by the user-defined type

```
students_type
```

The keys of the map are given by the students' IDs.
The following definitions are given

```
typedef unordered_map<unsigned,StudentsData> students_type;
typedef vector<string> row_type;
typedef vector<row_type> table_type;
```

# The Problem

**You have to implement:**

1) A class `FileManager` that can be used to read a generic CSV file and that stores the corresponding elements in

```
table_type fields;
```

In particular, files are read through the method

```
const table_type& parse_file (const std::string& filename,
                              char d = ',');
```

It receives as parameters the name of the file to be read and the char that is used in the file to separate the different fields (its default value is comma). It returns the table `fields` storing the values read from file (as a vector of vector of strings!).



# The Problem

2) Two functions

```
void add_students (const table_type&, students_type&);
```

and

```
void add_exams (const table_type&, students_type&);
```

that receive as parameters the table `fields` created by `FileManager` and the map of students' data and add to the map the students read from `students.txt` and the exams read from `exams.txt`, respectively.

**Note:** it is not possible to add exams to a student if the student is not stored in the map!

3) Print, for each student the average of their grades



## REMARK

Remember to import the initial code in CLion as an **existing project**, instead of directly opening it, in order to be able to build and execute the given code!

```cpp
//---------------------------------------------------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <unordered_map>
#include "FileManager.h"

using std::unordered_map;
using std::vector;
using std::string;
using std::cout;
using std::endl;
using std::cerr;

typedef vector<string> row_type;
typedef vector<row_type> table_type;
typedef unordered_map<unsigned,StudentsData> students_type;

void add_students (const table_type&, students_type&);
void add_exams (const table_type&, students_type&);

int main () {
    string filename_students = "../students.txt";
    string filename_exams = "../exams.txt";

    students_type students;

    FileManager FM;

    // parse students file
    table_type fields = FM.parse_file(filename_students);

    // add students to the map
    add_students(fields, students);

    // parse exams file
    fields = FM.parse_file(filename_exams);

    // add exams to the corresponding students, if they exist in the map
    add_exams(fields, students);

    // compute and print average of students' grades
    for (students_type::const_iterator cit = students.cbegin();  cit != students.cend(); ++cit) {
        cout << "id: " << cit->first << "; ";
        const StudentsData& student = cit->second;
        float avg = student.average_grade();
        cout << "average grade: " << avg << endl;
    }
}

void add_students (const table_type& fields, students_type& students) {
    for (const row_type& row : fields) {
        unsigned id = std::stoi(row[0]);
        StudentsData new_student(row[1], row[2], row[3]);
        students.insert({id, new_student});
    }
}

void add_exams (const table_type& fields, students_type& students) {
    for (const row_type& row : fields) {
        unsigned id = std::stoi(row[0]);
        students_type::iterator it = students.find(id);
        if (it != students.end()) {
            StudentsData& student = it->second;
            Exam new_exam(std::stoi(row[1]), row[2], std::stoi(row[3]));
            student.add_exam(new_exam);
        }
        else
            cerr << "ERROR in add_exams: it is not possible to add exams " << "to a non existing student" << endl;
    }
}
```

string to integer of the first element of "row"

**Note**: since we don't have the default constructor in the class StudentsData, we cannot procede with: students[id] = new_student;

```
//-----------------------------------------------------------------------------------------------------------------
// StudentsData.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef STUDENTSFILE_STUDENTSDATA_H
#define STUDENTSFILE_STUDENTSDATA_H

#include <vector>
#include <string>
#include "Exam.h"

class StudentsData {

    std::string name;
    std::string last_name;
    std::string birth_date;
    std::vector<Exam> exams;

    public:
        StudentsData(const std::string& name, const std::string& lastName, const std::string& birthDate);
        void add_exam(const Exam& e);
        float average_grade() const;
};

#endif //STUDENTSFILE_STUDENTSDATA_H


//-----------------------------------------------------------------------------------------------------------------
// StudentsData.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "StudentsData.h"

using std::string;

StudentsData::StudentsData(const string& name, const string& lastName, const string& birthDate):
    name(name), last_name(lastName), birth_date(birthDate) {}

void StudentsData::add_exam(const Exam& e) {
    exams.push_back(e);
}

float StudentsData::average_grade() const {
    float sum = 0.;
    for (const Exam& e : exams)
        sum += e.getGrade();
    return sum/exams.size();
}


//-----------------------------------------------------------------------------------------------------------------
// Exam.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef STUDENTSFILE_EXAM_H
#define STUDENTSFILE_EXAM_H

#include <string>

class Exam {

    size_t course_id;
    std::string date;
    unsigned grade;

    public:
        Exam(size_t courseId, const std::string& date, unsigned grade);
        size_t getCourseId() const;
        void setCourseId(size_t courseId);
        const std::string& getDate() const;
        void setDate(const std::string& date);
        unsigned getGrade() const;
        void setGrade(unsigned grade);
};

#endif //STUDENTSFILE_EXAM_H


//-----------------------------------------------------------------------------------------------------------------
// Exam.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "Exam.h"

using std::string;

Exam::Exam(size_t courseId, const string& date, unsigned grade):
    course_id(courseId), date(date), grade(grade) {}

size_t Exam::getCourseId() const {
    return course_id;
}
```

```cpp
void Exam::setCourseId(size_t course_id) {
    this->course_id = course_id;
}

const string& Exam::getDate() const {
    return date;
}

void Exam::setDate(const string& date) {
    this->date = date;
}

unsigned Exam::getGrade() const {
    return grade;
}

void Exam::setGrade(unsigned grade) {
    this->grade = grade;
}
```

```cpp
//-----------------------------------------------------------------------------------------------------------------
// FileManager.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef STUDENTSFILE_FILEMANAGER_H
#define STUDENTSFILE_FILEMANAGER_H

#include <iostream>
#include <fstream>
#include <sstream>
#include "StudentsData.h"

class FileManager {

    typedef std::vector<std::string> row_type;
    typedef std::vector<row_type> table_type;
    table_type fields;

    public:
        // default constructor
        FileManager () = default;

        // reads the file passed as argument, whose elements are separated by
        // the given character, and returns a table with the corresponding fields
        const table_type& parse_file (const std::string& filename, char d = ',');
};

#endif //STUDENTSFILE_FILEMANAGER_H
```

```cpp
//-----------------------------------------------------------------------------------------------------------------
// FileManager.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "FileManager.h"

using std::cerr;
using std::endl;
using std::string;
using std::ifstream;
using std::istringstream;

const FileManager::table_type& FileManager::parse_file (const std::string& filename, char d) {
    fields.clear();
    ifstream ifs(filename);
    if (ifs) {
        string line;
        while (getline(ifs, line)) {
            row_type row;
            istringstream iss(line);
            string elem;
            while (getline(iss, elem, d))
                row.push_back(elem);
            fields.push_back(row);
        }
    }
    else
        cerr << "ERROR in FileManager::parse_csv - file " << filename << " cannot be opened" << endl;
    return fields;
}
```

we have to clear it whenever we want to parse a new file

we istantiate an input file stream

if the file input stream is correctly opened

we estract one line at the time through getline and we put the content in "line"

using getline on the single row and adding the separator d we get the elements of the row (which we pushback in "row")

Container for all the informations in a row of the file

once we're done with one row we add it to the overall container

Laplacian $\left(\Delta f(x) = \sum_{i=0}^{n} \frac{\partial^2 f}{\partial x_i^2}, \quad \frac{\partial^2 f}{\partial x_i^2} \simeq \frac{f(x_i+h) - 2f(x_i) + f(x_i-h)}{h^2}\right)$ - Functional

```
//----------------------------------------------------------------------------
// main.cpp
//----------------------------------------------------------------------------
#include <iostream>
#include <vector>
#include <functional>
#include <cmath>

typedef std::vector<double> nd_vector;
static constexpr double pi = M_PI;
double compute_laplacian (std::function<double (const nd_vector&)>, const nd_vector&, double h = 0.01);
double ff (const nd_vector&);
double gg (const nd_vector&);

int main(){
    nd_vector x{1., 1., 1., 1.};
    std::cout << "ff(x) = " << ff(x) << std::endl;
    std::cout << "Df(x) = " << compute_laplacian(ff, x) << std::endl;

    x = {pi/2, 0.};
    std::cout << "gg(x) = " << gg(x) << std::endl;
    std::cout << "Dg(x) = " << compute_laplacian(gg, x) << std::endl;

    return 0;
}

double compute_laplacian (const std::function<double (const nd_vector&)> & f, const nd_vector& x, double h){
    double laplacian {0.};
    const double h2 = h * h;
    for (nd_vector::size_type i = 0; i < x.size (); ++i){
        nd_vector x_plus_h (x), x_minus_h (x);
        x_plus_h[i] += h;
        x_minus_h[i] -= h;
        laplacian += (f (x_plus_h) - 2 * f (x) + f (x_minus_h)) / h2;
    }
    return laplacian;
}

double ff (const nd_vector& x){
    double val{0.};
    for (nd_vector::size_type i = 0; i < x.size(); ++i)
        val += std::pow(x[i], i);
    return val;
}

double gg (const nd_vector& x){
    double val{0.};
    for (nd_vector::size_type i = 0; i < x.size(); ++i)
        val += std::sin(x[i]);
    return val;
}
```

whatever is the function f we can evaluate it and compute the laplacian

# Laplacian (parallel version)

```
//
// main.cpp
//
#include <iostream>
#include <vector>
#include <functional>
#include <cmath>

typedef std::vector<double> nd_vector;
static constexpr double pi = M_PI;
double compute_laplacian (std::function<double (const nd_vector&)>, const nd_vector&, double h = 0.01);
double ff (const nd_vector&);
double gg (const nd_vector&);

int main(){
    nd_vector x{1., 1., 1., 1.};
    std::cout << "ff(x) = " << ff(x) << std::endl;
    std::cout << "Df(x) = " << compute_laplacian(ff, x) << std::endl;

    x = {pi/2, 0.};
    std::cout << "gg(x) = " << gg(x) << std::endl;
    std::cout << "Dg(x) = " << compute_laplacian(gg, x) << std::endl;

    return 0;
}

/* SEQUENTIAL VERSION
double compute_laplacian (const std::function<double (const nd_vector&)> & f, const nd_vector& x, double h){
    double laplacian {0.};
    const double h2 = h * h;
    for (nd_vector::size_type i = 0; i < x.size (); ++i){
        nd_vector x_plus_h (x), x_minus_h (x);
        x_plus_h[i] += h;
        x_minus_h[i] -= h;
        laplacian += (f (x_plus_h) - 2 * f (x) + f (x_minus_h)) / h2;
    }
    return laplacian;
}
*/

double ff (const nd_vector& x){
    double val{0.};
    for (nd_vector::size_type i = 0; i < x.size(); ++i)
        val += std::pow(x[i], i);
    return val;
}

double gg (const nd_vector& x){
    double val{0.};
    for (nd_vector::size_type i = 0; i < x.size(); ++i)
        val += std::sin(x[i]);
    return val;
}
```

this becomes:

```
int main(int argc, char *argv[]){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

this becomes:

```
if (rank==0)
```

How can we parallelize the computation of the laplacian?
We split the evaluation of $\partial^2 f / \partial x_i^2$

```
// PARALLEL VERSION
//-----------------------------------------------------------------------------------------------------------------------
// laplacian.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef __LAPLACIAN__
#define __LAPLACIAN__

#include <functional>
#include "nd_vector.hh"

namespace numeric{

    double compute_laplacian (std::function<double (const nd_vector &)>, const nd_vector &, double h = 0.01);
}

#endif


//-----------------------------------------------------------------------------------------------------------------------
// laplacian.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <mpi.h>
#include "laplacian.hh"

namespace numeric{

    double compute_laplacian (std::function<double (const nd_vector &)> f, const nd_vector & x, double h){
        double laplacian (0.);
        const double h2 = h * h;

        int rank, size;
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        MPI_Comm_size (MPI_COMM_WORLD, &size);

        for (nd_vector::size_type i = rank; i < x.size (); i += size){
            nd_vector x_plus_h (x), x_minus_h (x);
            x_plus_h[i] += h;
            x_minus_h[i] -= h;
            laplacian += (f (x_plus_h) - 2 * f (x) + f (x_minus_h)) / h2;
        }

        MPI_Allreduce (MPI_IN_PLACE, &laplacian, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        return laplacian;
    }
}
```

once we have the partial sums, we have to put them together and compute the overall sum

→ Allreduce (not only Reduce) because this is a **parallel function**, every process needs to return the right thing (and so each process needs to be correctly updated)

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Supermarket

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



## Goal

- Implement a piece of software for the management of queues at counters in the supermarket chain BZR.
- BZR will install card readers at each counter, so that customers will virtually enter a line via scanning their fidelity card.
- Customer identifiers are represented as `strings`.



## Required Methods

- A method that takes an ID and adds the corresponding customer in line.
- A method that returns the ID to serve next.
- A method that checks whether any clients are waiting.

**Notes:**

- Customers cannot reserve more than one position in the queue and the system is expected to notify such failures.
- Your contract requires methods to be optimized for the average case.

```cpp
//---------------------------------------------------------------------------
// main.cpp
//---------------------------------------------------------------------------
#include <iostream>
#include "counter_queue.h"

int main (void){
    Supermarket::Counter_queue queue1;

    bool b0 = queue1.pick_number("AAA0");
    if (! b0)
        std::cout << "client was already in the queue" << std::endl;
    else
        std::cout << "client inserted in the queue!" << std::endl;

    bool b1 = queue1.pick_number("BBB1");
    if (! b1)
        std::cout << "client was already in the queue" << std::endl;
    else
        std::cout << "client inserted in the queue!" << std::endl;

    bool b2 = queue1.pick_number("CCC2");
    if (! b2)
        std::cout << "client was already in the queue" << std::endl;
    else
        std::cout << "client inserted in the queue!" << std::endl;

    bool b1_2 = queue1.pick_number("BBB1");
    if (! b1_2)
        std::cout << "client was already in the queue" << std::endl;
    else
        std::cout << "client inserted in the queue!" << std::endl;

    std::cout << "\nnext customer to serve: " << queue1.next_customer() << std::endl;
    std::cout << "next customer to serve: " << queue1.next_customer() << std::endl;

    return 0;
}
```

```cpp
//---------------------------------------------------------------------------
// counter_queue.h
//---------------------------------------------------------------------------
#ifndef COUNTERQUEUE_COUNTER_QUEUE_H
#define COUNTERQUEUE_COUNTER_QUEUE_H

#include <queue>
#include <string>
#include <unordered_set>

namespace Supermarket {

    class Counter_queue {

    private:
        std::queue<std::string> m_order;
        std::unordered_set<std::string> m_in_line;

    public:
        bool pick_number (const std::string & customer_id);
        std::string next_customer (void);
        bool empty (void) const;
    };
}

#endif //COUNTERQUEUE_COUNTER_QUEUE_H
```

To keep customers in-line:
It has "**push**", which add an element at the end, and "**pop**", which get the first element (and remove it)

Good option if we want to check if a client already exist

because we want the average complexity

receives the ID and inserts the customer in the queue

returns the ID on the next customer to be served

returns if there are no customers in the queue

we keep both! It reduces the complexity (and we want to reduce the average case complexity) at the cost of the memory (it's okay)

! We can use multiple containers if this reduces the complexity (if we've asked to reduce complexity and not memory)

```cpp
//---------------------------------------------------------------------------
// counter_queue.cpp
//---------------------------------------------------------------------------
#include "counter_queue.h"

namespace Supermarket{

    bool Counter_queue::pick_number (const std::string & customer_id){
        typedef std::pair<std::unordered_set<std::string>::iterator, bool> insert_return_type;
        const insert_return_type outcome = m_in_line.insert (customer_id);
        const bool added_in_queue = outcome.second;
        if (added_in_queue)
            m_order.push (customer_id);
        return added_in_queue;
    }

    std::string Counter_queue::next_customer (void){
        std::string next = m_order.front ();
        m_order.pop ();
        m_in_line.erase (next);
        return next;
    }

    bool Counter_queue::empty (void) const{
        return m_order.empty ();
    }
}
```

Iterator to the position where we add the new element

true if we add the new element

EXTRA

first element of the queue

we pop the first element from the front of the queue

we also erase the element from the unordered set


# Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 17-02-2020

Last Name: .....
First Name: .....
Student ID: .....
Signature: .....

Exam duration: 2 hours and 45 minutes

**Students can use a pen or a pencil for answering questions.**

**Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.**

**Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.**

**Writing on the cheat sheet is NOT allowed.**

**Exercise 1: ____ Exercise 2: ____ Exercise 3: ____**

## Exercise 1 (14 points)

You have to implement the class ComicBookStore which manages the comic book issues available in a comic book store and their selling. Each comic book issue is characterized by the comic book series to which belongs, which is uniquely identified by the title, by issue number (consecutive positive integers starting from 1), and by a date. It is supposed that the price of different issues of the series is the same and that the overall number of issues of a series is known in advance when the series is added to the data structure.

You already have the implementation of the class SingleComicBookSeries which contains basic information about a series:

```
class SingleComicBookSeries
{
  private:
    ///The title of the comic book series
    const std::string title;

    ///The number of issues up to now
    const unsigned int issues_number;

    ///The price of each issue
    const double price_per_issue;

  public:
    /**
     * Constructor
     */
    SingleComicBookSeries(const std::string & _title, unsigned int _issues_number, double
    _price_per_issue):
      title(_title),
      issues_number(_issues_number),
      price_per_issue(_price_per_issue)
    {}

    /// Get the title
    const std::string CGetTitle() const {
      return title;
    }
```

```
    /// Get the number of issues
    unsigned int CGetIssuesNumber() const {
      return issues_number;
    }

    /// Get the price per issue
    double CGetPricePerIssue() const {
      return price_per_issue;
    }
};
```

and you already have the implementation of the class IssueInformation which contains information about a single issue of a single series:

```
class IssueInformation
{
  private:
    ///Number of available copies
    unsigned int available_copies;

    ///Issue date
    std::string issue_date;

  public:
    ///Constructor
    IssueInformation(unsigned int initially_available_copies, const std::string & _issue_date) :
      available_copies(initially_available_copies),
      issue_date(_issue_date)
    {}

    ///Buy a copy
    void BuyIssue()
    {
      available_copies--;
    }

    ///Check for availability
    bool CheckAvailability() const
    {
      return available_copies != 0;
    }
};
```

The skeleton of the class definition of ComicBookStore is the following:

```
class ComicBookStore
{
  private:
    /// Data structure used to store necessary information

  public:
    /// Add a new comic book series
    void AddComicBookSeries(const std::string & title, double price_per_issue, unsigned int
    issues_number);

    /// Add to warehouse
    void AddToWareHouse(const std::string & title, const std::string & date, unsigned int issue_number,
    unsigned int quantity);
```

```
    /// Buy a single issue
    double BuyIssue(const std::string & title, unsigned int issue_number);

    /// Buy a whole series
    double BuySeries(const std::string & title);

    /// Return the price of an issue of a comic series
    double GetPrice(const std::string & title) const;

    /// Return the n series with highest price for issue
    std::list<SingleComicBookSeries> GetMostExpensive(unsigned int n) const;

     Check the availability of an issue
    bool CheckAvailability(const std::string & title, unsigned int issue_number) const;
};
```

In particular you have to complete class ComicBookStore:

1. Select the data structure to be added to ComicBookStore to store information about the different comic series and the availability of their issues. Data structures must be selected to **optimize the average complexity** of **all the methods** to be implemented. For this aim, **information may be duplicated** in multiple data structures to improve average complexity of different methods.
2. Implement AddComicBookSeries: it adds information about a single comic book series.
3. Implement AddToWareHouse: it is used when a new issue can be sold: quantity copies of issue_number of title becomes available.
4. Implement CheckAvailability: it checks for the availability (i.e., at least one copy is available) of issue_number of title.
5. Implement GetPrice: it returns the price of a single issue of title.
6. Implement BuyIssue: if at least one copy of issue_number of title is available, updates its availability (i.e., decrements by one) and returns its price, otherwise return 0.
7. Implement BuySeries: if at least one copy of each issue of title is available, updates the availability of all of them (i.e., decrements by one each of them) and the return the overall cost, otherwise returns 0.
8. Implement GetMostExpensive: it returns the n most expensive comic book series according to the price of the single issue.
9. Provide the average and the worst case complexity of all the implemented methods; for the complexity computation name $S$ the number of comic book series and $I$ the number of issues in the average/worst case.

**Duplicated code** should be avoided: if a functionality needed by a function is already implemented as a method, it should be reused.
**Provide the implementation** of other required methods or functions, if any.

Suggestions:

- In selection of data structure types, identify which type of information is required by the single methods.
- Pay attention to the **const** attribute of the functions.

## Solution 1

average case complexity
worst case complexity

In the proposed solution, three different data structures have been added to ComicBookStore:

- std::unordered_map<std::string, std::unordered_map<unsigned int, IssueInformation> > warehouse is the main data structure used to store the available number of copies of each issue. It is updated by AddToWareHouse , BuyIssue (and indirectly by BuySeries), and accessed by CheckAvailability. The use of unordered_map allows constant average case complexity. The value of the external unordered_map can also be a vector.

- std::unordered_map<std::string, SingleComicBookSeries> comic_book_series is used in GetPrice to get the price per issue of a comic book series starting from its title with constant average case complexity.
- std::set<SingleComicBookSeries> sorted_comic_series is used to store in a sorted way the comic issue series according to their price per issue. To achieve this result, the implementation of **operator<** must be provided:

```
#include "single_comic_book_series.hpp"
bool operator<(const SingleComicBookSeries & first, const SingleComicBookSeries & second)
{
  const auto first_price = first.CGetPricePerIssue();
  const auto second_price = second.CGetPricePerIssue();
  return first_price < second_price or (first_price == second_price and first.CGetTitle() < second.
    CGetTitle());
}
```

Note that if the two comic book series have the same price per issue, they still need to be sorted by their title, otherwise the set could not contain multiple comic book series with the same price. An alternative data structure is std::multimap<**double**, std::string> where the key is the price and the value is the title. This implementation would not require overloading of **operator<** since it is already based of sorting of **double**.

The implementation of the required methods is the following:

```
#include <iostream>
#include <utility>

#include "comic_book_store.hpp"

ComicBookStore::ComicBookStore(const std::string & _name) :
  name(_name)
{}
```

- ```
  void ComicBookStore::AddToWareHouse(const std::string & title, const std::string & date, unsigned int
      issue_number, unsigned int quantity)
  {
    warehouse.at(title).insert(std::pair<unsigned int, IssueInformation>(issue_number, IssueInformation(
      quantity, date)));
  }
  ```

  we access by ".at(..)"

  O(1), O(S+I)

- ```
  double ComicBookStore::BuyIssue(const std::string & title, unsigned int issue_number)
  {
    if(CheckAvailability(title, issue_number))
    {
      warehouse.at(title).at(issue_number).BuyIssue();
      return GetPrice(title);
    }
    else
    {
      return 0.0;
    }
  }

  double ComicBookStore::GetPrice(const std::string & title) const
  {
    return comic_book_series.at(title).CGetPricePerIssue();
  }
  ```

  this will decrement the number of available copies

- ```
  double ComicBookStore::BuySeries(const std::string & title)
  {
    const auto issues_number = comic_book_series.at(title).CGetIssuesNumber();
    for(unsigned int i = 1; i <= issues_number; i++)
      if(not CheckAvailability(title, i))
  ```

  we need all the issues to be available
  ```

```
    return 0.0;
  auto & availability = warehouse.at(title);
  for(unsigned int i = 1; i <= issues_number; i++)
    availability.at(i).BuyIssue();
  return issues_number * GetPrice(title);
}
```

we could have also used: `warehouse.at(title)[i].BuyIssue();`

- `std::list<SingleComicBookSeries> ComicBookStore::GetMostExpensive(unsigned int n) const` — return the n most expensive

```
{
  std::list<SingleComicBookSeries> ret_value;
  for(auto single_comic_book_series = sorted_comic_series.crbegin(); single_comic_book_series !=
    sorted_comic_series.crend() and ret_value.size() < n; single_comic_book_series++)
    ret_value.push_back(*single_comic_book_series);
  return ret_value;
}
```

constant reverse iterator

- `void ComicBookStore::AddComicBookSeries(const std::string & title, double price_per_issue, unsigned int issues_number)`

```
{
  SingleComicBookSeries single_comic_book_series(title, issues_number, price_per_issue);
  sorted_comic_series.insert(single_comic_book_series);
  comic_book_series.insert(std::pair<std::string, SingleComicBookSeries>(title, single_comic_book_series))
    ;
  warehouse[title];
}
```

new structures population

$O(\log(S)) = O(\log(S))$

we default initialize the internal unordered_map — $O(1), O(S)$

$O(1), O(S)$

- `bool ComicBookStore::CheckAvailability(const std::string & title, unsigned int issue_number) const`

```
{
  return warehouse.at(title).find(issue_number) != warehouse.at(title).end() and warehouse.at(title).at(
    issue_number).CheckAvailability();
}
```

we need to check if there is the number AND if it's available

notice that here we CANNOT do things like: `warehouse[title][issue_number].CheckAvailability();` because the [] can potentially modify the data → the method is const!

Please note that:

- Internal unordered_map of warehouse cannot be accessed with [] since the value type (i.e., IssueInformation) does not have empty constructor.
- BuyIssue and BuySeries use CheckAvailability and GetPrice.
- In GetPrice [] cannot be used on comic_book_series since the method is const.
- In BuySeries, the availability of all the issues must be checked before updating their availability.
- GetMostExpensive uses the ordering of sorted_comic_series to access the most expensive comic book series; since the most expensive are required, the iteration must start from the last element (i.e., sorted_comic_series .crbegin()).
- warehouse[title] in AddComicBookSeries is equivalent to warehouse[title] = std::unordered_map<unsigned int, IssueInformation>(): without this statement warehouse.at(title) would fail.
- In the proposed implementation, IssueInformation of an issue is created only when AddToWareHouse for that particular issue is invoked. For this reason, CheckAvailability must check if IssueInformation exists before invoking CheckAvailability on it.

The complexity of the implemented methods is the following ($S$ is the number of comic):

- Constructor: average and worst case complexity: $O(1)$.
- AddComicBookSeries: average case complexity is $O(log(S)$ because of the insert in the set; worst case complexity is $O(S)$ because of insertion in unordered_map.
- AddToWareHouse: average case complexity is $O(1)$ (find/insertion in unordered_map); worst case complexity is $O(S+I)$ because of the at in the external unordered_map and the insert in the internal unordered_map.

- CheckAvailability: average case complexity is $O(1)$, while worst case complexity is $O(S+I)$ (see complexity of AddToWareHouse).
- GetPrice: average case complexity is $O(1)$, worst case complexity is $O(S)$ because of the access to an unordered_map.
- BuyIssue: average case complexity is $O(1)$, while worst case complexity is $O(S+I)$ (see complexity of AddToWareHouse).
- BuySeries: average case complexity is $O(I)$ (accesses to unordered_map are constant - loops are executed $I$ times); worst case complexity is $O(I \cdot (S+I))$ because of the first loop and of the complexity of CheckAvailability. Without code reuse (i.e., without use of CheckAvailability the worst case complexity can be reduced at $O(I \cdot I)$.
- GetMostExpensive: average and worst case complexity is linear in the value of n.
- operator<: average and worst case complexity is constant.

## Exercise 2 (14 points)

You have to develop a parallel library for $n$-dimensional vectors operations (assume $n$ significantly larger than the number of available processes). You can rely on the following class nd_vector which stores a vector in $\mathbb{R}^n$; an object of this class can be initialized both by copy and by providing the size, $n$. Furthermore, it provides a size method to read $n$ and an unchecked indexing operator to access the values. Finally for input/output operations the class provides a print method and a read method from ifstream.

```
namespace numeric
{
 class nd_vector
 {
  typedef std::vector<double> container_type;

  container_type x;

 public:
  typedef container_type::value_type value_type;
  typedef container_type::size_type size_type;
  typedef container_type::pointer pointer;
  typedef container_type::const_pointer const_pointer;
  typedef container_type::reference reference;
  typedef container_type::const_reference const_reference;

  explicit nd_vector (size_type n = 0);
  nd_vector (std::initializer_list<double>);


  size_type
  size (void) const;

  void
  read (std::ifstream & input_stream);

  void
  print(void) const;


  reference
  operator [] (size_type);
  value_type
```

OPTIMIZE AVERAGE COMPLEXITY
Let's analyze all the methods of the class one by one:

---

- -> AddComicBookSeries(title, price_per_issue, issues_number)

---

-> unordered_map<string, SingleComicBookSeries>
(we need fast access by title)
! Every time we need to access by a key and we have to optimize average complexity we use an unordered_map

---

- -> AddToWareHouse(title, date, issue_number, quantity)

---

-> unordered_map<string, unordered_map<int, IssueInformation>>
(we need fast access to tile and issue_number)

---

- -> BuyIssue(title, issue_number)

---

-> unordered_map<string, unordered_map<int, IssueInformation>>
(we need fast access to tile and issue_number)

---

- -> BuySeries(title)

---

-> unordered_map<string, unordered_map<int, IssueInformation>>
(we need fast access to the tile)

---

- -> GetPrice(title)

---

-> unordered_map<string, SingleComicBookSeries>
(we need fast access to the tile)

---

-> GetMostExpensive(n)

---

-> set<SingleComicBookSeries>
(we also need to implement the < operator)

---

-> CheckAvailability(title, issue_number)

---

-> unordered_map<string, unordered_map<int, IssueInformation>>
(we need fast access to title and issue_number)

overall we need to add:
unordered_map<string, SingleComicBookSeries>
unordered_map<string, unordered_map<int, IssueInformation>>
set<SingleComicBookSeries>

deep se
dependable evolvable pervasive software engineering group

# Exercise Session – Taxi Call Center

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

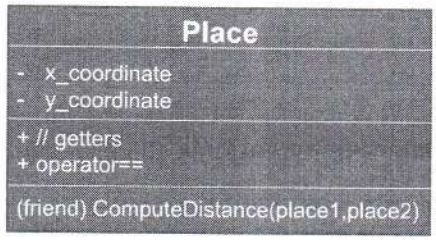


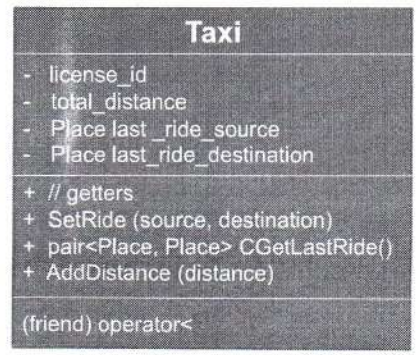

## Goal

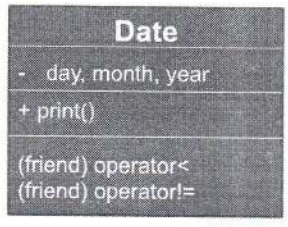
- Develop the core class `CallCenter` of the `MyTaxi` application for a small city with one railroad station.
- `CallCenter` provides an interface to the user to call taxi and store information about all the taxis of the city.
- Each taxi is identified in a unique way by the license id (of type `std::string`) of its driver.
- **Note:** since most of the rides are from or to the rail station, ad-hoc functionalities for this type of rides have to be implemented.

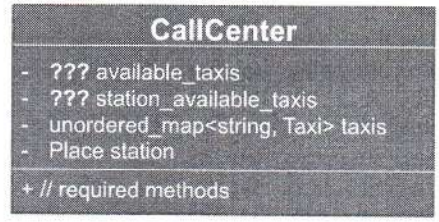


## Code Structure



# Required methods

- **`Taxi Call(const Place&, const Place&)`**, that returns the taxi, if any, whose current position is closest to the source of the ride.
- **`Taxi CallAtRailStation(const Place&)`**, that returns the available taxi which: 1) is currently at the rail station, and 2) has run the smallest distance among all available taxis located at the rail station.
- **`Taxi CallToRailStation(const Place&)`**, that returns the taxi, if any, whose current position is closest to the source of the ride
- **`Arrived(const string&)`**, that updates information about `available_taxis`, overall distance of taxi, and eventually `station_available_taxis`.

- **Note:** in all methods, if there are multiple taxis matching the request, whatever of them can be returned.


# Algorithms and Parallel Computing

Course 052496
Prof. Danilo Ardagna

Date: 15-01-2019

Last Name: .....
First Name: .....
Student ID: .....
Signature: .....

Exam duration: 2 hours and 45 minutes

**Students can use a pen or a pencil for answering questions.**

**Students are NOT permitted to use books, course notes, calculators, mobile phones, and similar connected devices.**

**Students are NOT permitted to copy anyone else's answers, pass notes amongst themselves, or engage in other forms of misconduct at any time during the exam.**

**Writing on the cheat sheet is NOT allowed.**

Exercise 1: ____ Exercise 2: ____ Exercise 3: ____ Exercise 4: ____

## Exercise 1 (12 points)

You have to develop the core class `CallCenter` of the MyTaxi application for a small city with one railroad station. `CallCenter` provides an interface to the user to call taxi and store information about all the taxis of the city. Each taxi is identified in a unique way by the license id (of type `std::string`) of its driver. Since most of the rides are from or to the rail station, ad-hoc functionalities for this type of rides have to be implemented.

The following data structures and functions have already been implemented and are available:

- `Date`, a class which stores information about a date.
- `Place`, a class which stores information about a location in the city.
- `float ComputeDistance(const Place &, const Place &)`, a function which computes the distance between two locations.
- `Taxi`, a class which stores information about a taxi, whose implementation follows.

```
#ifndef TAXI_HPP
#define TAXI_HPP

#include <iostream>
#include <string>

#include "place.hpp"

class Taxi
{
  private:
    std::string license_id;

    float total_distance = 0.0;

    Place last_ride_source = Place();

    Place last_ride_destination = Place();

  public:
    //Constructor
```

```
    Taxi(const std::string & _license_id);

    //Return the overall distance run by taxi
    float CGetOverallDistance() const;

    //Return the license id associated with the taxi
    const std::string CGetLicenseId() const;

    //Return the current position (if taxi is stopped)
    const Place CGetPosition() const;

    //Update information about current ride
    void SetRide(const Place source, const Place destination);

    //Return the information about last ride
    const std::pair<Place, Place> CGetLastRide() const;

    //Update the overall distance of taxi
    void AddDistance(const float distance)
};

//Return first_taxi.CGetOverallDistance() < second_taxi.CGetOverallDistance()
bool operator< (const Taxi & first_taxi, const Taxi & second_taxi);
#endif
```

You are required to complete the following implementation of CallCenter:

```
#ifndef CALLCENTER_HPP
#define CALLCENTER_HPP

#include "date.hpp"
#include "place.hpp"
#include "taxi.hpp"

class CallCenter
{
  private:
    /* PUT YOUR CODE HERE */ available_taxis;

    /* PUT YOUR CODE HERE */ station_available_taxis;

    std::unordered_map<std::string, Taxi> taxis;

    const Place railstation = Place(0.0, 0.0);

  public:
    //Call a taxi
    const Taxi Call(const Place & source, const Place & destination);

    //Call a taxi for a ride starting from rail station
    const Taxi CallAtRailStation(const Place & Destination);

    //Call a taxi for a ride towards the rail station
    const Taxi CallToRailStation(const Place & Source);

    //Update data structure when a ride ends
    void Arrived(const std::string & license_id);
};
#endif
```

In particular you have to:

1. Specify the types of available_taxis, station_available_taxis, optimized for the call of CallAtRailStation, which has to be implemented with a special focus on the time complexity.
2. Provide the implementation of Call, i.e., call for a taxi: the function must return the taxi, if any available, which current position is closest to the source of the ride; if there are more than one taxi at the smallest distance, whatever of them can be returned.
3. Provide the implementation of CallAtRailStation: the function must return the available taxi which: 1) is currently at the rail station, and 2) has run the smallest distance among all available taxis located at the rail station. If there are more than one taxi available at the station with the smallest distance, whatever of them can be returned. If there is not any available taxi at the rail station, the closest to the rail station must be selected.
4. Provide the implementation of CallToRailStation, i.e., call for a taxi for a ride to the rail station: the function must return the taxi, if any available, whose current position is closest to the source of the ride; if there are more than one taxi at the smallest distance, whatever of them can be returned.
5. Provide the implementation of Arrived: the function must update information about available_taxis, overall distance of taxi, and eventually station_available_taxis.
6. Provide the complexity of the implemented methods.

Suggestions:

- License id can be used to identify the taxis.
- Call, CallAtRailStation, CallToRailStation share part of the functionality, so some code can be reused.
- Call, CallAtRailStation, CallToRailStation, and Arrived must update the data structures about the available taxis.
- railstation member variable represents the rail station location.
- Call, CallAtRailStation, CallToRailStation can return a null taxi when there is not any available taxi in this way: **return** Taxi("nonexistent");

## Solution 1

Two different solutions are proposed which differ because of the relationship between available_taxis and station_available_taxis. In the first solution data of the latter are included in the former (i.e., a taxi available at the rail station is included in both), in the second solution the data structures are disjoint (i.e., a taxi available at the rail station is included only in station_available_taxis.

```
#ifndef CALLCENTER_HPP
#define CALLCENTER_HPP

#include <set>
#include <string>
#include <unordered_map>
#include <unordered_set>

#include "date.hpp"
#include "place.hpp"
#include "taxi.hpp"

class CallCenter
{
  private:
    std::unordered_set<std::string> available_taxis;
```

```
    std::set<Taxi> station_available_taxis;

    std::unordered_map<std::string, Taxi> taxis;

    const Place station = Place(0.0, 0.0);

  public:
    const Taxi Call(const Place & source, const Place & destination);

    const Taxi CallAtRailStation(const Place & Destination);

    const Taxi CallToRailStation(const Place & Source);

    void Arrived(const std::string & license_id);

    void RegisterTaxi(const std::string & license_id);
};
#endif
```

*First solution*

```
#include "callcenter.hpp"

void CallCenter::RegisterTaxi(const std::string & license_id)
{
  Taxi new_taxi(license_id);
  taxis.insert(std::pair<std::string, Taxi>(license_id, new_taxi));
  available_taxis.insert(license_id);
  if(new_taxi.CGetPosition() == station)
    station_available_taxis.insert(new_taxi);
}

const Taxi CallCenter::Call(const Place & source, const Place & destination)
{
  if(available_taxis.empty())
  {
    std::cerr << "There is not any available taxi" << std::endl;
    return Taxi("nonexistent");
  }
  auto closest_taxi_id = *(available_taxis.begin());
  float closest_distance = ComputeDistance(taxis.at(closest_taxi_id).CGetPosition(), source);
  for(const auto taxi_id : available_taxis)
  {
    const auto current_distance = ComputeDistance(taxis.at(taxi_id).CGetPosition(), source);
    if(current_distance < closest_distance)
    {
      closest_distance = current_distance;
      closest_taxi_id = taxi_id;
    }
  }
  available_taxis.erase(closest_taxi_id);
  auto return_taxi = taxis.at(closest_taxi_id);
  if(return_taxi.CGetPosition() == station)
    station_available_taxis.erase(return_taxi);
  return_taxi.SetRide(source, destination);
  return return_taxi;
}

const Taxi CallCenter::CallAtRailStation(const Place & destination)
{
```

```
  if(station_available_taxis.size())
  {
    auto first_taxi = *(station_available_taxis.begin());
    taxis.at(first_taxi.CGetLicenseId()).SetRide(station, destination);
    station_available_taxis.erase(first_taxi);
    available_taxis.erase(first_taxi.CGetLicenseId());
    return first_taxi;
  }
  else
  {
    return Call(station, destination);
  }
}

const Taxi CallCenter::CallToRailStation(const Place & source)
{
  return Call(source, station);
}

void CallCenter::Arrived(const std::string & license_id)
{
  auto taxi = taxis.at(license_id);
  const auto last_ride = taxi.CGetLastRide();
  taxi.AddDistance(ComputeDistance(last_ride.first, last_ride.second));
  if(taxi.CGetPosition() == station)
    station_available_taxis.insert(taxi);
  available_taxis.insert(license_id);
}
```

*Second solution*

```
#include "callcenter.hpp"

void CallCenter::RegisterTaxi(const std::string & license_id)
{
  Taxi new_taxi(license_id);
  taxis.insert(std::pair<std::string, Taxi>(license_id, new_taxi));
  if(new_taxi.CGetPosition() == station)
    station_available_taxis.insert(new_taxi);
  else
    available_taxis.insert(license_id);
}

const Taxi CallCenter::Call(const Place & source, const Place & destination)
{
  if(available_taxis.empty())
  {
    if(station_available_taxis.empty())
    {
      std::cerr << "There is not any available taxi" << std::endl;
      return Taxi("nonexistent");
    }
    else
    {
      const auto first_taxi = *(station_available_taxis.begin());
      station_available_taxis.erase(first_taxi);
      taxis.at(first_taxi.CGetLicenseId()).SetRide(source, destination);
      return first_taxi;
    }
  }
```

```
  auto closest_taxi = *(available_taxis.begin());
  float closest_distance = ComputeDistance(taxis.at(closest_taxi).CGetPosition(), source);
  for(const auto taxi : available_taxis)
  {
    const auto current_distance = ComputeDistance(taxis.at(taxi).CGetPosition(), source);
    if(current_distance < closest_distance)
    {
      closest_distance = current_distance;
      closest_taxi = taxi;
    }
  }
  if(not station_available_taxis.empty())
  {
    const auto station_distance = ComputeDistance(source, station);
    if(station_distance < closest_distance)
    {
      auto first_taxi = *(station_available_taxis.begin());
      taxis.at(first_taxi.CGetLicenseId()).SetRide(source, destination);
      station_available_taxis.erase(first_taxi);
      return first_taxi;
    }

  }
  available_taxis.erase(closest_taxi);
  taxis.at(closest_taxi).SetRide(source, destination);
  return taxis.at(closest_taxi);
}

const Taxi CallCenter::CallAtRailStation(const Place & destination)
{
  if(station_available_taxis.size())
  {
    const auto first_taxi = *(station_available_taxis.begin());
    station_available_taxis.erase(first_taxi);
    taxis.at(first_taxi.CGetLicenseId()).SetRide(station, destination);
    return first_taxi;
  }
  else
  {
    return Call(station, destination);
  }
}

const Taxi CallCenter::CallToRailStation(const Place & source)
{
  return Call(source, station);
}

void CallCenter::Arrived(const std::string & license_id)
{
  auto taxi = taxis.at(license_id);
  const auto last_ride = taxi.CGetLastRide();
  taxi.AddDistance(ComputeDistance(last_ride.first, last_ride.second));
  if(taxi.CGetPosition() == station)
    station_available_taxis.insert(taxi);
  else
    available_taxis.insert(license_id);
}
```

Since the whole information about the single taxi is already stored in `taxis`, to reduce the number of copies of objects of type `taxi`, the set of available taxis can be stored by exploiting their license ids (which is of type `std::string`). Since the taxis have not to be ordered we can use unordered set. Please note that also solutions exploiting smart pointers should be preferred, but then smart pointers should be used also as input and as return value of the functions of `CallCenter`. Taxis available at rail station must instead be sorted, so a set should be preferred. In this case, the value type of `station_available_taxis` cannot be `std::string`. A set storing the license ids indeed would sort the taxis according to their license ids instead of according their overall distances. Please note that the proposed solutions are not the only correct ones.

`Call` in the first solution iterates over all the taxis looking for the one closest to the source of ride.

`CallAtRailStation` takes the first taxi in `station_available_taxis`. Since taxis are sorted according to their overall distance, the first is the one with the smallest distance. If there is not available taxi at the rail station, the closest one is selected by means `Call`. `station` is used as first argument since this is the implicit source of the ride. On the contrary, the only difference between a generic ride and a ride towards the rail station is in the fixed destination. For this reason, `CallToRailStation` always exploits `Call` passing `station` as second argument.

Finally, `Arrived` updates the overall distance of the taxi, checks if the destination of the ride was rail station and in this case adds the taxi to the `station_available_taxis`. In the first solution the taxi is always added also to the set of the available ones while in the second solution only if it is not at the station.

The complexities (given $n$ taxis) are the following:

- The complexity of `Call` in worst and average case is $O(n)$ since it iterates over all the taxis.
- The complexity of `CallAtRailStation` in worst and average case is constant since `station_available_taxis` is already sorted under the assumption that there is one available taxi.
- The complexity of `CallToRailStation` is the same of `Call` ($O(n)$ both in worst and average case).
- The complexity of `Arrived` is $O(n)$ in the worst case (because of `available_taxis.insert()`) and $O(log(n))$ in the average case (because of `station_available_taxis.insert()`).

## Exercise 2 (10 points)

You have to complete the implementation of a parallel application which counts the occurrences of a number in a text file. In details, you have to:

1. Develop a parallel function with the following prototype:

   ```
   std::size_t search_and_count (const std::string & file_name, int key);
   ```

   The function reads from the input text file named **file_name** a set of integer values and returns the number of values that are equal to **key**. Assume that the **file is available only on rank 0**.

2. Complete the implementation of the **main** function which is provided in the following, specifying the code to be added in (a), (b), (c):

   ```
   #include <iostream>
   #include <fstream>
   #include <algorithm>
   #include <mpi.h>
   #include <vector>
   #include <string>

   using input_data_t = std::vector<int>;

   // return the data read from file_name
   input_data_t read_data(const std::string & file_name);

   std::size_t search_and_count (const std::string & file_name, int key);

   int
   main (int argc, char * argv[])
   ```

# MPI - basics

```
//-----------------------------------------------------------------------------------------------------------------------
// commandline.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>

int main( int argc, char *argv[] ) {
    if( argc == 2 ) {
        std::cout << "The argument supplied is " << argv[1] << std::endl;
    }
    else if( argc > 2 ) {
        std::cout << "Too many arguments supplied." << std::endl;
    }
    else
        std::cout << "One argument expected." << std::endl;
}


//-----------------------------------------------------------------------------------------------------------------------
// hello.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <cstdio>
#include <mpi.h>

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank, size;
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    printf ("Hello from process %d of %d\n", rank, size);
    MPI_Finalize ();
    return 0;
}


//-----------------------------------------------------------------------------------------------------------------------
// hello2.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <cstdio>
#include <mpi.h>
#include <iostream>
#include <sstream>

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank, size;
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    // Assumption we use up to 10 processes, no more!
    constexpr unsigned max_string = 18;
    std::ostringstream builder;
    builder << "Hello from " << rank << " of " << size;
    std::string message (builder.str ());
    if (rank > 0)
        MPI_Send (&message[0], max_string , MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    else {
        std::cout << message << std::endl;
        for (int r = 1; r < size; ++r){
            MPI_Recv (&message[0], max_string , MPI_CHAR, r, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            std::cout << message << std::endl;
        }
    }

    MPI_Finalize ();
    return 0;
}
```

# MPI - 1

```
//-----------------------------------------------------------------------------------------------------------------
// hardcoded.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <mpi.h>
#include "quadrature.hh"

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    double a (0.0), b (3.0);
    unsigned n (1024);

    const double h = (b - a) / n;
    const unsigned local_n = n / size;

    const double local_a = a + rank * local_n * h;
    const double local_b = local_a + local_n * h;

    double local_int = quadrature::trapezoidal (local_a, local_b, local_n);

    if (rank > 0) {
        MPI_Send (&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    else { // rank 0
        double total (local_int);
        for (int source = 1; source < size; ++source){
            MPI_Recv (&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total += local_int;
        }
        std::cout << "n = " << n << ", a = " << a << ", b = " << b << ", integral = " << total << std::endl;
    }

    MPI_Finalize ();
    return 0;
}
```

if we're not process rank 0 we have to send the value of the integral:

we provide the address of the variable storing the local integral (&local_int), we're sending 1 number, actually of type MPI_DOUBLE, to who? rank 0, we initialize tag 0 (we don't need to distinguish anything), and where is this process rank 0? in MPI_COMM_WORLD

also rank 0 has to compute the integral, we initialize the sum with its integral

rank 0 will receive and will store where? in local_int (it'll be overwritten each time) (&local_int). It'll be 1 number of type MPI_DOUBLE from who? from source (which change at each iteration). Tag is again initialized to 0. The communicator word is MPI_COMM_WORLD and the status will be MPI_STATUS_IGNORE.

```
//-----------------------------------------------------------------------------------------------------------------
// quadrature.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef QUADRATURE
#define QUADRATURE

namespace quadrature{
    double trapezoidal (double a, double b, unsigned n);
}

#endif // QUADRATURE
```

```
//-----------------------------------------------------------------------------------------------------------------
// quadrature.cpp
//-----------------------------------------------------------------------------------------------------------------
#include "quadrature.hh"

namespace quadrature{

    double f (double x){
        return x * x + 3 * x + 1;
    }

    double trapezoidal (double a, double b, unsigned n){
        const double h = (b - a) / n;
        double sum = (f (a) + f (b)) / 2;
        for (unsigned i = 1; i < n; ++i){
            const double x = a + h * i;
            sum += f (x);
        }
        return sum * h;
    }
}
```

What do we have to write in the **command line** to run it?

```
$ cd Desktop/MPI/
$ atom
$ cd trapezoidal-rule/
$ mpicxx --std=c++11 hardcoded.cc quadrature.cc -o quad_1
$ mpiexec -np 4 quad_1
```

and we get: n=1024, a=0, b=3, integral = 25.5

# MPI - 2

```
//-----------------------------------------------------------------------------------------------------------------------
// with_io.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <mpi.h>
#include "input.hh"
#include "quadrature.hh"
#include "sum_and_output.hh"

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);

    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    double a, b;
    unsigned n;

    mpi::get_input (a, b, n);

    const double h = (b - a) / n;
    const unsigned local_n = n / size;

    const double local_a = a + rank * local_n * h;
    const double local_b = local_a + local_n * h;

    const double local_int = quadrature::trapezoidal (local_a, local_b, local_n);

    mpi::sum_and_print (local_int, std::cout, a, b, n);

    MPI_Finalize ();
    return 0;
}


//-----------------------------------------------------------------------------------------------------------------------
// p2p_input.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <mpi.h>
#include "input.hh"

namespace mpi{

  void get_input (double & a, double & b, unsigned & n){
        int rank, size;
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        MPI_Comm_size (MPI_COMM_WORLD, &size);
        if (rank == 0){
            std::cin >> a >> b >> n;
            for (int dest = 1; dest < size; ++dest){
                MPI_Send (&a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
                MPI_Send (&b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
                MPI_Send (&n, 1, MPI_UNSIGNED, dest, 0, MPI_COMM_WORLD);
            }
        }
        else{
            MPI_Recv (&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv (&b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv (&n, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
}


//-----------------------------------------------------------------------------------------------------------------------
// sum_and_output.h
//-----------------------------------------------------------------------------------------------------------------------
#ifndef SUM_AND_OUTPUT
#define SUM_AND_OUTPUT

#include <ostream>

namespace mpi{

    void sum_and_print (double integral, std::ostream &, double a, double b, unsigned n);
}

#endif // SUM_AND_OUTPUT
```

Only the processor rank 0 can take something in input. This means that it has to pass it to every other rank

attention to the type matching

```cpp
//-----------------------------------------------------------------------------
// p2p_output.cpp
//-----------------------------------------------------------------------------
#include <mpi.h>
#include "sum_and_output.hh"

namespace mpi{

  void sum_and_print (double local_integral, std::ostream & out, double a, double b, unsigned n){
      int rank, size;
      MPI_Comm_rank (MPI_COMM_WORLD, &rank);
      MPI_Comm_size (MPI_COMM_WORLD, &size);
      if (rank > 0){
          MPI_Send (&local_integral, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
      }
      else{
          double total (local_integral);
          for (int source = 1; source < size; ++source){
              MPI_Recv (&local_integral, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
              total += local_integral;
          }
          out << "n = " << n << ", a = " << a << ", b = " << b << ", integral = " << total << std::endl;
      }
  }
}
```

we don't write the headers!

How do we call it? (command line)

```
$ mpicxx --std=c++11 quadrature.cc with_io.cc p2p_input.cc p2p_output.cc
$ mpiexec -np 4 a.out
0 3 1024
n=1024, a=0, b=3, integral = 25.5
```

→ we insert this line manually

What if we don't want to write every time the line?

```
$ vim input_0_3_1024
$ cat input_0_3_1024
0 3 1024
```

it directly print it!

text file that is independent of the extension

```
0 3 1024
```

```
$ mpiexec -np 4 a.out < input_0_3_1024
n=1024, a=0, b=3, integral = 25.5
```

Note moreover that

```
$ mpiexec -np 4 a.out < input_0_3_1024 > output_0_3_1024
```

won't get anything as output!

**input redirect** (it means that the input is not read from the keyboard but it's read from the file)

**output redirect** (the output is not written in the terminal but in the file "output_0_3_1024")

deep se
dependable evolvable pervasive software engineering group

# Exercise Session (MPI) – $\pi$ approximation

Federica Filippini

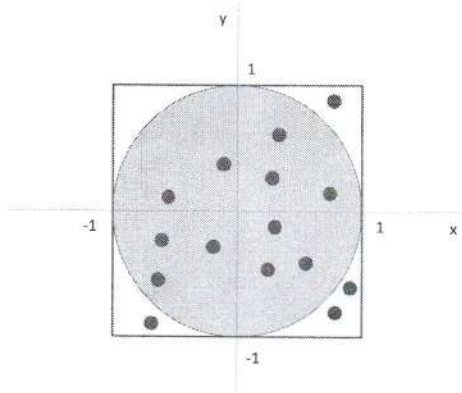
Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



## Goal

- Implement a parallel program to approximate π.




$$\frac{\text{Area}_{\text{circle}}}{\text{Area}_{\text{square}}} = \frac{\pi \cdot r \cdot r}{4 \cdot r \cdot r} = \frac{\pi}{4}$$

Dartboard



## Algorithm (serial version)

```
darts = /* a large number */
score = 0    /* how many times the dart falls in the circle */
for (n = 1; n <= darts; ++n)
      generate a random x-coordinate in [-1, 1]
      generate a random y-coordinate in [-1, 1]
      if (x-coordinate,y-coordinate) is in the circle
            score++
      end if
end for
pi = 4 * score / darts
```

→ Repeat for `n_iterations` times and compute the average

We want to parallelize the whole process so that we obtain **N** different approximations of **π** and we average them. (**N** approx. on **K** processes)

```cpp
//------------------------------------------------------------------------------------------------------------------------
// main.cpp
//------------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <random>
#include <mpi.h>

double square(double x); // return x*x
double dboard(unsigned darts, unsigned seed); // implement Pi approximation through darts random sampling

int main (int argc, char* argv[]){
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc == 3){

        // get number of iterations and darts from command-line
        unsigned iterations = std::stoul(argv[1]);
        unsigned darts = std::stoul(argv[2]);

        // compute local number of iterations
        const unsigned long local_n = iterations / size;

        // initialize local seed
        unsigned local_seed = rank * local_n;

        // compute local sum
        double local_sum = 0;
        for (unsigned i = 1; i <= local_n; ++i)
            local_sum += dboard(darts, local_seed++);

        // communication
        if (rank > 0){
            // int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
            MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
        else{
            double approx_pi = local_sum;
            for (int r = 1; r < size; ++r){
                // int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
                MPI_Recv(&local_sum, 1, MPI_DOUBLE, r, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                approx_pi += local_sum;
            }
            approx_pi /= iterations;
            std::cout << "Pi approximation " << approx_pi << std::endl;
        }
    }
    else
        std::cerr << "Error, two arguments expected " << std::endl;

    MPI_Finalize();
    return 0;
}

double square(double x){
    return x*x;
}

double dboard(unsigned darts, unsigned seed){
    double  x_coord,    // x coordinate, between -1 and 1
            y_coord,    // y coordinate, between -1 and 1
            pi;         // pi approximation
    unsigned score{0};  // number of darts that hit circle

    // random engine generator
    std::default_random_engine generator(seed);

    // callable object for random number generation
    std::uniform_real_distribution<double> distribution (-1.0,1.0);

    for (unsigned n = 1; n <= darts; ++n){
        // generate random values for x and y coordinates
        x_coord = distribution(generator);
        y_coord = distribution(generator);

        // if dart lands in circle, increment score
        if ((square(x_coord) + square(y_coord)) <= 1.0)
            ++score;
    }

    // calculate pi
    pi = 4.0 * static_cast<double>(score) / static_cast<double>(darts);

    return pi;
}
```

by STANDARD INPUT
≠ std::cin !

we want the user to pass the total number of iterations and the total number of darts by COMMAND LINE (e.g. " ./main 100 10 " : 100 iterations 10 darts)

conversion from string to unsigned

with this we are sure that each process uses a different seed

given

given

to generate random numbers

better to manually cast them to doubles

July 2020 - Ex. 2

```
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <iostream>
#include "Author.h"
#include <vector>
#include <iostream>
#include <mpi.h>

using std::vector;
using authors =  vector<Author>;
size_t author_nation_count(const string & nation, const authors & auts);

int main(int argc, char *argv[]) {
    int rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    authors auts;
    auts.emplace_back("S. Lippman", "USA");
    auts.emplace_back("J. K. Rowling","UK");
    auts.emplace_back("A. C. Doyle","UK");

    string nation;
    if (rank ==0)
        std::cin>> nation;

    size_t count_uk =  author_nation_count(nation, auts);

    if (rank ==0)
        std::cout <<count_uk<< std::endl;

    MPI_Finalize();
    return 0;
}

size_t author_nation_count(const string & nation, const authors & auts){
    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    std::string nation_msg = nation;
    unsigned length = nation.size();

    if (rank == 0){
        MPI_Bcast (&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
        MPI_Bcast (&nation_msg[0],length,MPI_CHAR,0,MPI_COMM_WORLD);
    }
    else{
        MPI_Bcast (&length,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
        nation_msg.resize(length);
        MPI_Bcast (&nation_msg[0],length,MPI_CHAR,0,MPI_COMM_WORLD);
    }

    unsigned count =0;

    for (size_t i=rank; i< auts.size(); i+=size)
        if (auts[i].get_nation() == nation_msg)
            count++;

    MPI_Allreduce (MPI_IN_PLACE, &count, 1, MPI_UNSIGNED, MPI_SUM, MPI_COMM_WORLD);
    return count;
}
```

We assume the vector of authors is available in all processes, while nation is available only in rank 0.

These are standardly initialized in ranks ≠ 0

We need rank 0 to broadcast the information of the "nation". To do that, we need to have enough space in each core destinated to "nation". A priori (knowing the string a priori) it's simple, but here "nation" is inserted by the user, so we need to use the variable "length" (and so, we need to broadcast also "length")

Before providing the broadcast for the string we need to resize the string according to the length of the original message

because of how it's implemented we need that every process has the right (total and local) count → we need to update all "count"

```
//-----------------------------------------------------------------------------
// Author.h
//-----------------------------------------------------------------------------
#ifndef AUTHORSSEARCH_AUTHOR_H
#define AUTHORSSEARCH_AUTHOR_H

#include <string>
using std::string;

class Author {

    string name;
    string nation;

    public:
        Author(const string &name, const string &nation);
        string get_name() const;
        string get_nation() const;
};

#endif //AUTHORSSEARCH_AUTHOR_H
```

```cpp
//---------------------------------------------------------------------------
// Author.cpp
//---------------------------------------------------------------------------
#include "Author.h"

Author::Author(const string &_name, const string &_nation) : name(_name), nation(_nation) {}

string Author::get_name() const {
    return name;
}

string Author::get_nation() const {
    return nation;
}
```

mpi::read-vector and mpi::print-vector

```
//------------------------------------------------------------------------------
// main.cpp
//------------------------------------------------------------------------------
#include <iostream>
#include <mpi.h>
#include <vector>

#include "input.hh"
#include "output.hh"
#include "vector_sum.hh"

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    unsigned n = 0;
    if (rank == 0) std::cin >> n;

    MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

    const std::vector<double> x = mpi::read_vector (n, "x", MPI_COMM_WORLD);
    const std::vector<double> y = mpi::read_vector (n, "y", MPI_COMM_WORLD);

    using numeric::operator +;
    const std::vector<double> z =  x +  y;

    mpi::print_vector (z, n, "Result vector 'z':", MPI_COMM_WORLD);

    MPI_Finalize ();
    return 0;
}
```

the size is known only by rank 0, this is how rank 0 passes the information to all

functionality of reading and spreading the informations among the processes (after this, the vectors x and y will be only local)

local sums

this is responsable for building up a single vector and print it (rank 0 prints it)

# MPI_Reduce

```
//-------------------------------------------------------------------------------------------------
// main.cpp
//-------------------------------------------------------------------------------------------------
#include <iostream>
#include <mpi.h>

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    int a=1;
    int b=0;
    int c=2;
    int d=0;

//  int MPI_Reduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
//                  MPI_Op op, int dest, MPI_Comm comm);

    if (rank==0){
        MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
    else if (rank ==1){
        MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
    else if (rank ==2){
        MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize ();
    return 0;
}
```

What is the value of **b** and **d**? b = 4; d = 5;

Initial situation:

| process 0 | process 1 | process 2 |
|---|---|---|
| a=1; | a=1; | a=1; |
| c=2; | c=2; | c=2; |

**Time 1:**

0: MPI_Reduce(&a, &b, .., 0, comm)
1: MPI_Reduce(&c, &d, .., 0, comm)
2: MPI_Reduce(&a, &b, .., 0, comm)

The destination is 0 and so the results will be plugged in the process 0 destination (which is &b)
What are the contributors? Each process will contribute with its sender, which is:
- rank 0 → &a
- rank 1 → &c
- rank 2 → &a

⟹ "b = a + c + a" = 1 + 2 + 1 = 4

The destination buffer that matters is only the one of rank 0 !!!

**Time 2:**

0: MPI_Reduce (&c, &d, .., 0, comm)
1: MPI_Reduce (&a, &b, .., 0, comm)
2: MPI_Reduce (&c, &d, .., 0, comm)

We look at the receiver buffer only in the destination → rank 0.
The receiving buffer is **d** (&d).
What are the contributors?
- rank 0 → &c
- rank 1 → &a
- rank 2 → &c

⟹ "d = c + a + c" = 2 + 1 + 2 = 5

! Generally: MPI_Reduce (&var1, &var2, .., k, comm)
- if you're rank **k** ⟹ it matters both &var1, &var2
- if you are not rank **k** ⟹ it matters only what you give: &var1
  (what must match is the destination (which is "rank k", not &var2!))

**Cyclic partitioning** – minimum of a function on a set of points

```
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <mpi.h>
#include <iostream>
#include <vector>

using std::vector;
double f(double x);
double min_f (const vector<double> v);

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank, size;
    vector<double> v = {1,2,3,4};

    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    double min_val = min_f(v);
    //std::cout << "Min f(x) is : " << min_val <<std::endl; // All processes knows the min and will print
    if (rank ==0) // To have a single print rank 0 print the minimum
        std::cout << "Min f(x) is : " << min_val <<std::endl;

    MPI_Finalize ();
    return 0;
}

double f(double x){
    return x*x;
}

double min_f (const vector<double> v){
    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    double local_min = 1000;

    for (size_t i = rank; i < v.size (); i += size){
        double f_x = f(v[i]);
        if (f_x < local_min)
            local_min = f_x;
    }

    double global_min;
    MPI_Allreduce (&local_min, &global_min, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

    return global_min;
}
```

v is available for all processes

we initialize it as a large number

when we use a parallel function there are **no** options: The parallel function needs to return the proper value across all cores (all processes)

→ MPI_Allreduce

~~MPI_Reduce~~

deep se
dependable evolvable pervasive software engineering group

# Parallel Matrix Multiplication

D. Ardagna, F. Filippini, L. Fiorentini

POLITECNICO DI MILANO



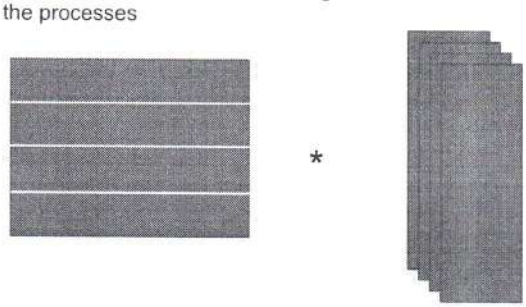
## Goals

- Write a **parallel program** to perform matrix multiplication
- Recall that you can represent a **dense matrix** in memory as a vector, or array
  - rows are stored one after another
- The initial code already implements a dense_matrix class
  - `operator *` performs serial matrix multiplication
  - `data()` returns a pointer to the data elements
- You have a skeleton for the main function, where the input matrices are read from file and the final result is printed to screen



## Goals

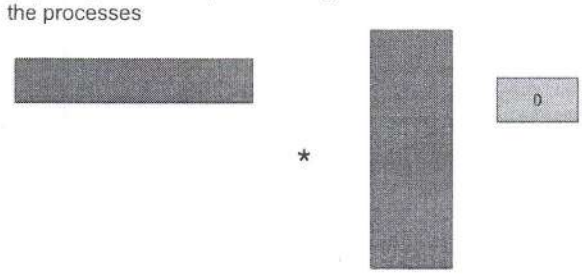
- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes





## Goals

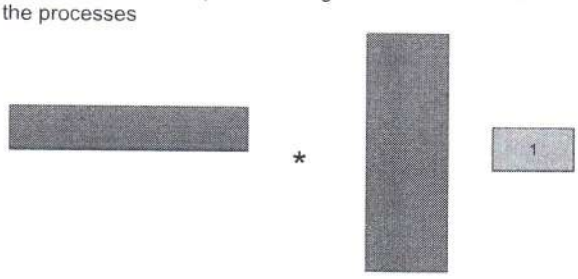
- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes






## Goals

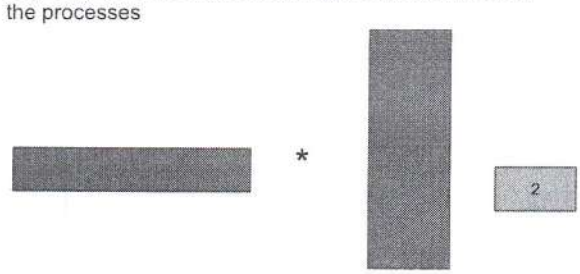
- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes





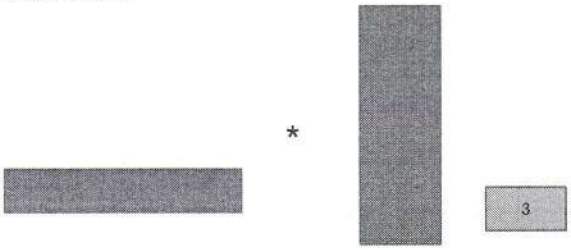

## Goals

- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes



# Goals

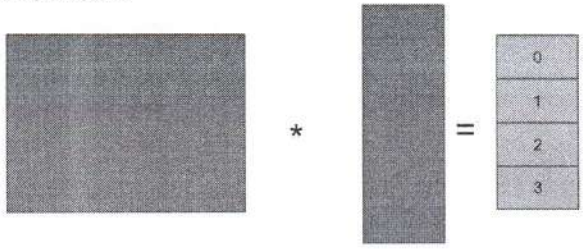
- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes




D. Ardagna, G. Quattrocchi, F. Filippini, L. Fiorentini - Fraud Detection

# Goals

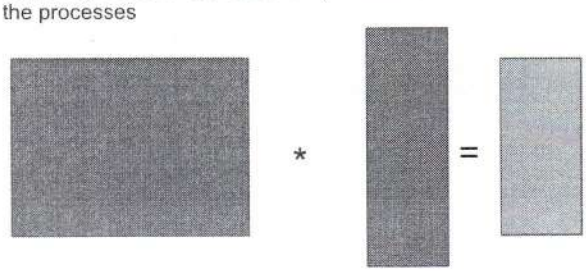
- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes




D. Ardagna, G. Quattrocchi, F. Filippini, L. Fiorentini - Fraud Detection

# Goals

- As parallelization scheme split the left hand operand in stripes by row and replicate the right hand matrix on all the processes




D. Ardagna, G. Quattrocchi, F. Filippini, L. Fiorentini - Fraud Detection

# Goals

- When you collect the partial slices, make the full result available on all processes (on rank 0 could be enough but this way results is avalable at all processes and solution is more general)
- Assume that all the matrix dimensions are multiples of the communicator size
- Input matrices are provided as text files and file names are obtained from the command line

# Scatter & (All) Gather

```
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <fstream>
#include <iostream>
#include <mpi.h>
#include "dense_matrix.hh"

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    la::dense_matrix full_A(0, 0);
    la::dense_matrix full_B(0, 0);

    if (rank == 0){
        std::ifstream first (argv[1]), second (argv[2]);
        full_A.read (first);
        full_B.read (second);
    }

    unsigned m = full_A.rows(), n = full_A.columns(), p = full_B.columns();

    MPI_Bcast(&m,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
    MPI_Bcast(&n,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);
    MPI_Bcast(&p,1,MPI_UNSIGNED,0,MPI_COMM_WORLD);

    la::dense_matrix local_B(n,p);

    if(rank==0){
        MPI_Bcast(full_B.data(), n*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        local_B = full_B;
    }
    else{
        MPI_Bcast(local_B.data(), n*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    unsigned stripe = m/size;
    la::dense_matrix local_A(stripe,n);

    MPI_Scatter(full_A.data(), stripe*n, MPI_DOUBLE, local_A.data(), stripe*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    la::dense_matrix local_C = local_A*local_B;
    la::dense_matrix full_C(m,p);

    MPI_Allgather(local_C.data(), stripe*p, MPI_DOUBLE, full_C.data(), stripe*p, MPI_DOUBLE, MPI_COMM_WORLD);

    if (rank == 0){
        std::cout << full_C.rows() << " " << full_C.columns() << "\n";

        for (std::size_t i = 0; i < full_C.rows(); ++i)
            for (std::size_t j = 0; j < full_C.columns(); ++j){
                std::cout << full_C(i, j);
                std::cout << (full_C.columns() - j == 1 ? "\n" : " ");
            }
    }

    MPI_Finalize ();
    return 0;
}
```

used for importing two matrices (provided at the exam)!

At the end rank 0 will be the only one that will have data → it needs to share (Broadcast)

we're doing (mxn)x(nxp) (A·B)

!!!

1. (nxp) matrix of zeros

population of local_B starting from full_B (we broadcast the whole matrix)

1. (!)

2.

3.

4.

only rank 0 prints out

1. give to every core a copy of the matrix B (local_B)
2. divide A in stripes (total number of rows / number of cores = # stripes) → each core will deal with one stripe (a subset of rows). Then create the "smaller A" to perform the local products (local_A)
3. Where is the source? full_A (⇒ full_A.data()).
   How many elems we're sending? stripes * n, of type MPI_DOUBLE.
   Where are we sending it (destination)? local_A (⇒ local_A.data()).
   How many elements? stripes * n, of type MPI_DOUBLE.
   Who is the source? Rank 0. (+ communicator MPI_COMM_WORLD)
4. We create the local result (local_C) and then we have to aggregate all the local results.
   First we need to create the matrix full_C (where we'll collect local_C)
   MPI_Allgather: where the data is coming from? local_C (⇒ local_C.data())
   how many elements? stripe * p, of type MPI_DOUBLE.
   where do we store it (destination)? full_C (⇒ full_C.data())
   how many? stripe * p, of type MPI_DOUBLE
   + communicator MPI_COMM_WORLD

   Do we need to indicate the core of destination? No, every core needs to do it.

```
//-----------------------------------------------------------------------------
// dense_matrix.h
//-----------------------------------------------------------------------------
#ifndef DENSE_MATRIX_HH
#define DENSE_MATRIX_HH

#include <istream>
#include <vector>

namespace la {

    class dense_matrix final{
        typedef std::vector<double> container_type;

        public:
            typedef container_type::value_type value_type;
            typedef container_type::size_type size_type;
            typedef container_type::pointer pointer;
            typedef container_type::const_pointer const_pointer;
            typedef container_type::reference reference;
            typedef container_type::const_reference const_reference;

        private:
            size_type m_rows, m_columns;
            container_type m_data;
            size_type sub2ind (size_type i, size_type j) const;

        public:
            dense_matrix (void) = default;
            dense_matrix (size_type rows, size_type columns, const_reference value = 0.0);
            explicit dense_matrix (std::istream &);
            void read (std::istream &);
            void swap (dense_matrix &);
            reference operator () (size_type i, size_type j);
            const_reference operator () (size_type i, size_type j) const;
            size_type rows (void) const;
            size_type columns (void) const;
            dense_matrix transposed (void) const;
            pointer data (void);
            const_pointer data (void) const;
    };

    dense_matrix operator * (dense_matrix const &, dense_matrix const &);
    void swap (dense_matrix &, dense_matrix &);
}

#endif // DENSE_MATRIX_HH
```

```
//-----------------------------------------------------------------------------
// dense_matrix.cpp
//-----------------------------------------------------------------------------
#include <sstream>
#include <string>
#include "dense_matrix.hh"

namespace la{

    dense_matrix::dense_matrix (size_type rows, size_type columns, const_reference value):
        m_rows (rows), m_columns (columns), m_data (m_rows * m_columns, value) {}

    dense_matrix::dense_matrix (std::istream & in){
        read (in);
    }

    dense_matrix::size_type dense_matrix::sub2ind (size_type i, size_type j) const{
        return i * m_columns + j;
    }

    void dense_matrix::read (std::istream & in){
        std::string line;
        std::getline (in, line);
        std::istringstream first_line (line);
        first_line >> m_rows >> m_columns;
        m_data.resize (m_rows * m_columns);

        for (size_type i = 0; i < m_rows; ++i){
            std::getline (in, line);
            std::istringstream current_line (line);
            for (size_type j = 0; j < m_columns; ++j){
                /* alternative syntax: current_line >> operator () (i, j);
                 * or: current_line >> m_data[sub2ind (i, j)];
                                */
                current_line >> (*this)(i, j);
            }
        }
    }
```

```
    void dense_matrix::swap (dense_matrix & rhs){
        using std::swap;
        swap (m_rows, rhs.m_rows);
        swap (m_columns, rhs.m_columns);
        swap (m_data, rhs.m_data);
    }

    dense_matrix::reference dense_matrix::operator () (size_type i, size_type j){
        return m_data[sub2ind (i, j)];
    }

    dense_matrix::const_reference dense_matrix::operator () (size_type i, size_type j) const{
        return m_data[sub2ind (i, j)];
    }

    dense_matrix::size_type dense_matrix::rows (void) const{
        return m_rows;
    }

    dense_matrix::size_type dense_matrix::columns (void) const{
        return m_columns;
    }

    dense_matrix dense_matrix::transposed (void) const{
        dense_matrix At (m_columns, m_rows);
        for (size_type i = 0; i < m_columns; ++i)
            for (size_type j = 0; j < m_rows; ++j)
                At(i, j) = operator () (j, i);
        return At;
    }

    dense_matrix::pointer dense_matrix::data (void){
        return m_data.data ();
    }

    dense_matrix::const_pointer dense_matrix::data (void) const{
        return m_data.data ();
    }

    dense_matrix operator * (dense_matrix const & A, dense_matrix const & B){
        using size_type = dense_matrix::size_type;
        dense_matrix C (A.rows (), B.columns ());
        for (size_type i = 0; i < A.rows (); ++i)
            for (size_type j = 0; j < B.columns (); ++j)
                for (size_type k = 0; k < A.columns (); ++k)
                    C(i, j) += A(i, k) * B(k, j);
        return C;
    }

    void swap (dense_matrix & A, dense_matrix & B){
        A.swap (B);
    }
}
```

deep se
dependable evolvable pervasive software engineering group

# Exercise Session (MPI) – MonteCarlo method

Federica Filippini

Politecnico di Milano
federica.filippini@polimi.it

POLITECNICO DI MILANO



## Goal

- Implement a **parallel function** to approximate definite integrals on limited intervals $\Omega \subset \mathbb{R}$.
- Prototype:

value of the integral — variance

```
std::pair<double, double>
montecarlo (const std::function<double (double)>& f,
            unsigned long N);
```

number of iteration that we want to perform

- **Assumptions:**
  - N is known only on the master node
  - Ω = [-1, 1]



## MonteCarlo method

- Recalls N observation from the random variable X ~ $U(\Omega)$
- Estimates the integral with

$$Q_N = |\Omega| \, \bar{Y}_N$$

where Y = f(X) and |Ω| is the measure of Ω.

- By the law of large numbers,

$$\lim_{N \to +\infty} Q_N = I = \int_\Omega f(x) \, \mathrm{d}x$$

- Variance of the quadrature formula:

$$\mathrm{Var}(Q_N) = \frac{|\Omega|^2}{N} \sigma^2$$

where $\sigma^2$ = Var(Y), which can be estimated through $S_N^2$.

```
//------------------------------------------------------------------------------------------------------------------------
// montecarlo.h
//------------------------------------------------------------------------------------------------------------------------
#ifndef __MONTECARLO__
#define __MONTECARLO__

#include <functional>
#include <utility>

namespace quadrature{

    std::pair<double, double> montecarlo (const std::function<double (double)> & f, unsigned long N);
}

#endif // __MONTECARLO__
```

```
//------------------------------------------------------------------------------------------------------------------------
// montecarlo.cpp
//------------------------------------------------------------------------------------------------------------------------
#include <mpi.h>
#include <random>
#include "montecarlo.hh"

namespace quadrature{

    std::pair<double, double> montecarlo (const std::function<double (double)> & f, unsigned long N){
        int rank, size;
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        MPI_Comm_size (MPI_COMM_WORLD, &size);

        MPI_Bcast (&N, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
        const unsigned long local = N / size;
        const unsigned long subsample = (rank < N % size) ? local + 1 : local;

        std::default_random_engine engine (rank * rank * size * size);
        std::uniform_real_distribution<double> distro (-1., 1.);

        double mean = 0.;
        std::vector<double> ys (subsample);
        for (double & y: ys){
            const double x = distro (engine);
            y = f (x);
            mean += y;
        }
        MPI_Allreduce (MPI_IN_PLACE, &mean, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        mean /= N;

        double variance = 0.;
        for (double y: ys) variance += (y - mean) * (y - mean);
        MPI_Allreduce (MPI_IN_PLACE, &variance, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        variance /= (N - 1);

        const double integral = 2 * mean;
        const double integral_variance = 2 * 2 * variance / N;

        return std::make_pair (integral, integral_variance);
    }
}
```

value of the integral
variance of the estimate

number of iterations that we want to perform (known only by rank 0)

we communicate N

!!! we don't know if size is a multiple of N. If it's not (N=5, size=2) then local will be casted into an integer and we'll lose some iteration. To avoid this we write the second line where we add more numbers to one of the processors.
e.g. N=5, size=2: N/size = 2
rank 0: 0 < 1 yes ⇒ subsamples = local + 1 = 3
rank 1: 1 < 1 no ⇒ subsamples = local = 2

random number generation from $\mathcal{U}([-1,1])$

we initialize passing the seed (it must be ≠ for all the processors)

we collect the global result (we sum all the partial sums)

alternatively: return {integral, integral_variance};

What do we have to do?

1. sample N times from $\mathcal{U}([-1,1])$ : $x_1, \dots, x_N$
2. compute $f(X) = Y$ : $y_1, \dots, y_N$
3. compute mean and variance of Y :
   integral estimate = $|\Omega| \cdot \text{mean}(Y) = 2 \cdot \text{mean}(Y)$
   variance = $\frac{|\Omega|^2}{N} \cdot \text{variance}(Y) = \frac{4}{N} \text{variance}(Y)$

deep se
dependable evolvable pervasive software engineering group

# Power method

Federica Filippini, Danilo Ardagna, Marco Lattuada

Politecnico di Milano
federica.filippini@polimi.it
danilo.ardagna@polimi.it

POLITECNICO DI MILANO



## Content

The power method is a numerical technique to approximate the dominant eigenvector of a square matrix, that is whose eigenvalue has the highest absolute value

Starting from an initial guess for the eigenvector, compute repeatedly the matrix-vector product; the sequence obtained will converge to the dominant eigenvector

**Matrix-vector product is an expensive operation, but can be easily parallelized with MPI**



## Content

$$\mathbf{x}_1 = A\mathbf{x}_0$$
$$\mathbf{x}_2 = A\mathbf{x}_1 = A^2\mathbf{x}_0$$
$$\mathbf{x}_3 = A\mathbf{x}_2 = A^3\mathbf{x}_0$$
$$\vdots$$
$$\mathbf{x}_k = A\mathbf{x}_{k-1} = A^k\mathbf{x}_0$$

this sequence converges to the dominant eigenvector ($x_k$ dom. eigen.)

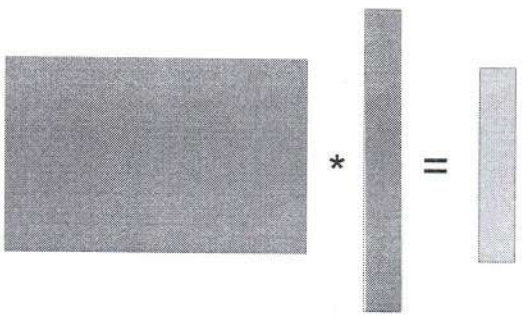


## Goals

- Write a parallel program to perform the power method
- Matrices are represented as DenseMatrix objects, whose values are stored by rows in a vector
- The initial code already contains the matrix class along with some useful methods. Notably:
  - operator * performs the serial product and is already implemented
  - data() returns a pointer to the data elements
- The main function performs reading from input matrices and output printing, while it lacks the power method section
- **Assume matrix dimensions are always multiples of the communicator size!**

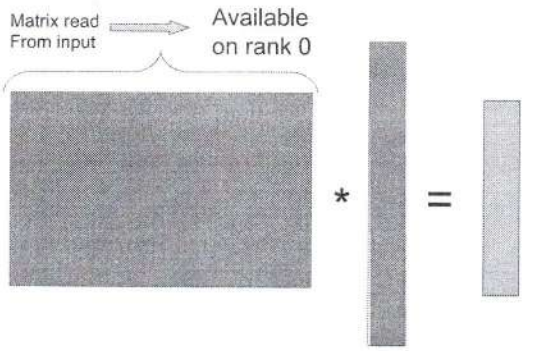


## What parallelization scheme to use?

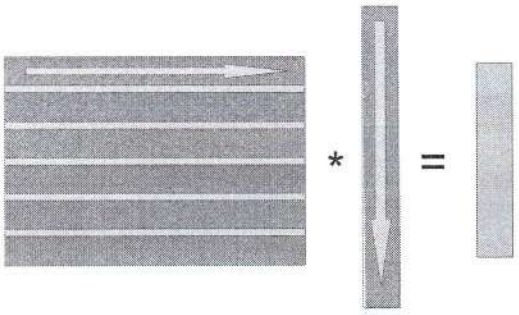




## What parallelization scheme to use?



## What parallelization scheme to use?

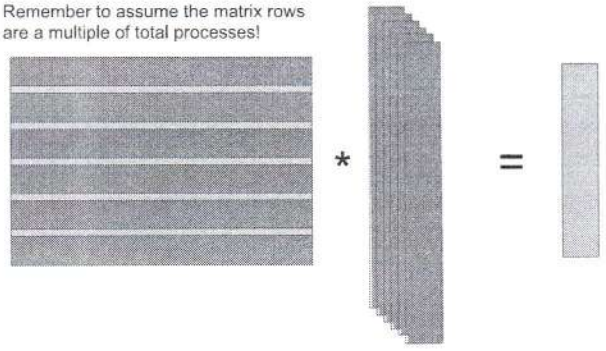




## What parallelization scheme to use?

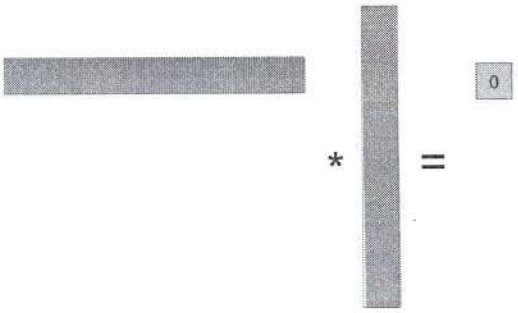
BLOCK PARTITION

Remember to assume the matrix rows are a multiple of total processes!



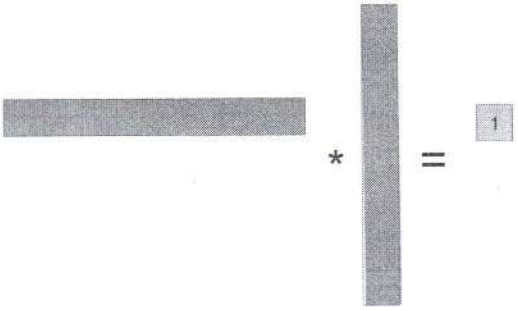


## What parallelization scheme to use?



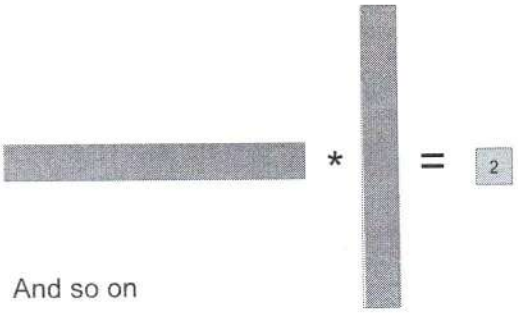



## What parallelization scheme to use?






## What parallelization scheme to use?




And so on



## Implementation

- When collecting partial results at iteration k, make the full $\mathbf{x}_k$ available to all processes at each iteration of the method by means of collective communication
- Write the *function* `power_method` in the corresponding source files; it is already defined in the provided header
- **Remember to reuse the already implemented operations**
- Finally, once compiled with the Makefile provided (executing `make`), start the resulting executable with

```
mpiexec -np 2 ./power_method <filename>.txt
```

- **Note the matrix file is provided as command line argument!**

```cpp
//-----------------------------------------------------------------------------
// main.cpp
//-----------------------------------------------------------------------------
#include <fstream>
#include <iostream>
#include <mpi.h>
#include "dense_matrix.hpp"
#include "power_method.hpp"

DenseMatrix read_matrix(const std::string & file_name);

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    DenseMatrix local_A;

    local_A = read_matrix(argv[1]);
    std::size_t iterations = 20;
    DenseMatrix eigen_vector = power_method(local_A, iterations);

    if (rank == 0)
        print(eigen_vector);

    MPI_Finalize ();
    return 0;
}

DenseMatrix read_matrix(const std::string & file_name){
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    unsigned n(0);

    DenseMatrix full_A;

    if (rank == 0){
        std::ifstream f_stream (file_name);
        full_A.read (f_stream);
        n = full_A.get_n_cols ();
    }

    MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

    const unsigned stripe = n / size;

    DenseMatrix local_A(stripe,n);

    MPI_Scatter (full_A.data(), stripe*n, MPI_DOUBLE, local_A.data(), stripe*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return local_A;
}
```

initialized as empty

we're calling this function from all the processors

if we call argv[1] then what we read is available in all processors

only rank 0 reads the matrix

then rank 0 shares the information of the number of columns

each process creates a local (smaller) matrix

!!! we first have to prepare the containers BEFORE THE SCATTER

rank 0 can now split the matrix in the different sub-matrices

```cpp
//-----------------------------------------------------------------------------
// power_method.hpp
//-----------------------------------------------------------------------------
#ifndef POWER_METHOD_H_
#define POWER_METHOD_H_

#include <mpi.h>
#include "dense_matrix.hpp"

DenseMatrix power_method(const DenseMatrix & local_A, std::size_t iterations);

#endif
```

```cpp
//-----------------------------------------------------------------------------
// power_method.cpp
//-----------------------------------------------------------------------------
#include "power_method.hpp"

DenseMatrix power_method(const DenseMatrix & local_A, std::size_t iterations){
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank); // Not needed but useful for debugging:
    MPI_Comm_size (MPI_COMM_WORLD, &size); // E.g.,  cout << "Rank: " << rank << what you want to check!

    const unsigned n = local_A.get_n_cols ();
    const unsigned stripe = local_A.get_n_rows ();

    DenseMatrix full_eigen_vector(n, 1, 1.0);

    for (std::size_t it_n = 0; it_n < iterations; ++it_n){
        DenseMatrix local_X = local_A * full_eigen_vector;
        MPI_Allgather (local_X.data(), stripe, MPI_DOUBLE, full_eigen_vector.data(), stripe, MPI_DOUBLE, MPI_COMM_WORLD);
    }

    return full_eigen_vector;
}
```

complete result and also initial guess (n rows, 1 column, initialized all with 1.0s)

what are we reading

how many (# rows in the local matrix)

of what

receiving buffer

```
//-----------------------------------------------------------------------------------------------------------------------
// matrix.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef MATRIX_H_
#define MATRIX_H_

#include <cstddef>
#include <iostream>

class Matrix{

    protected:
        std::size_t n_rows;
        std::size_t n_cols;

    public:
        Matrix (std::size_t rows = 0, std::size_t cols = 0);
        virtual double & operator () (std::size_t i, std::size_t j) = 0;
        virtual double operator () (std::size_t i, std::size_t j) const = 0;
        std::size_t get_n_rows() const;
        std::size_t get_n_cols() const;
        virtual ~Matrix (void) = default;
};

void print (const Matrix & matrix);

#endif /* MATRIX_H_ */


//-----------------------------------------------------------------------------------------------------------------------
// matrix.cpp
//-----------------------------------------------------------------------------------------------------------------------
#include "matrix.hpp"

Matrix::Matrix (std::size_t rows, std::size_t cols): n_rows (rows), n_cols (cols) {}

std::size_t Matrix::get_n_cols (void) const{
    return n_cols;
}

std::size_t Matrix::get_n_rows (void) const{
    return n_rows;
}

void print (const Matrix & matrix){
    for (std::size_t i = 0; i < matrix.get_n_rows (); ++i){
        for (std::size_t j = 0; j < matrix.get_n_cols (); ++j){
            std::cout << matrix (i, j) << "\t";
        }
        std::cout << std::endl;
    }
}


//-----------------------------------------------------------------------------------------------------------------------
// dense_matrix.hpp
//-----------------------------------------------------------------------------------------------------------------------
#ifndef DENSE_MATRIX_H_
#define DENSE_MATRIX_H_

#include <istream>
#include <vector>
#include <cmath>
#include "matrix.hpp"

class DenseMatrix : public Matrix{

    public:
        typedef std::vector<double> container_type;

    private:
        container_type m_data;
        std::size_t sub2ind (std::size_t i, std::size_t j) const;

    public:
        DenseMatrix (std::size_t rows = 0, std::size_t columns = 0, double value = 0.0);
        DenseMatrix (std::size_t rows, std::size_t columns, const container_type& values);
        void read (std::istream &);
        void swap (DenseMatrix &);
        double & operator () (std::size_t i, std::size_t j) override;
        double operator () (std::size_t i, std::size_t j) const override;
        DenseMatrix transposed (void) const;
        double * data (void);
        const double * data (void) const;
        container_type get_data (void) const;
};

DenseMatrix operator * (const DenseMatrix &, const DenseMatrix &);
void swap (DenseMatrix &, DenseMatrix &);

//double norm (const DenseMatrix&);

#endif // DENSE_MATRIX_HH
```

```cpp
//-----------------------------------------------------------------------------------------------------------------
// dense_matrix.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <sstream>
#include <string>
#include <iostream>
#include "dense_matrix.hpp"

DenseMatrix::DenseMatrix (size_t rows, size_t columns, double value) :
    Matrix(rows, columns), m_data (n_rows * n_cols, value){}

DenseMatrix::DenseMatrix(size_t rows, size_t columns, const container_type& values) :
    Matrix(rows, columns), m_data(values){}

size_t DenseMatrix::sub2ind (size_t i, size_t j) const{
    return i * n_cols + j;
}

void DenseMatrix::read (std::istream & in){
    std::string line;
    std::getline (in, line);
    std::istringstream first_line (line);
    first_line >> n_rows >> n_cols;
    m_data.resize (n_rows * n_cols);
    for (size_t i = 0; i < n_rows; ++i){
        std::getline (in, line);
        std::istringstream current_line (line);
        for (size_t j = 0; j < n_cols; ++j){
            /* alternative syntax: current_line >> operator () (i, j);
             * or: current_line >> m_data[sub2ind (i, j)];
             */
            current_line >> (*this)(i, j);
        }
    }
}

void DenseMatrix::swap (DenseMatrix & rhs){
    using std::swap;
    swap (n_rows, rhs.n_rows);
    swap (n_cols, rhs.n_cols);
    swap (m_data, rhs.m_data);
}

DenseMatrix DenseMatrix::transposed (void) const{
    DenseMatrix At (n_cols, n_rows);
    for (size_t i = 0; i < n_cols; ++i)
        for (size_t j = 0; j < n_rows; ++j)
            At(i, j) = operator () (j, i);
    return At;
}

double * DenseMatrix::data (void){
    return m_data.data ();
}

const double * DenseMatrix::data (void) const{
    return m_data.data ();
}

DenseMatrix::container_type DenseMatrix::get_data (void) const{
    return m_data;
}

DenseMatrix operator * (const DenseMatrix & A, const DenseMatrix & B){
    DenseMatrix C (A.get_n_rows (), B.get_n_cols ());
    for (size_t i = 0; i < A.get_n_rows (); ++i)
        for (size_t j = 0; j < B.get_n_cols (); ++j)
            for (size_t k = 0; k < A.get_n_cols (); ++k)
                C(i, j) += A(i, k) * B(k, j);
    return C;
}

void swap (DenseMatrix & A, DenseMatrix & B){
    A.swap (B);
}

double & DenseMatrix::operator()(size_t i, size_t j){
    return m_data.at(sub2ind(i, j));
}

double DenseMatrix::operator()(size_t i, size_t j) const{
    return m_data.at(sub2ind(i, j));
}
```

```
//-----------------------------------------------------------------------------------------------------------------
// grep-main.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <mpi.h>
#include "grep.hh"

int main (int argc, char * argv[]){
    MPI_Init (&argc, &argv);
    grep::lines_found local_filtered_lines;
    unsigned local_lines_number;
    grep::search_string (argv[1], argv[2], local_filtered_lines, local_lines_number);
    grep::print_result (local_filtered_lines, local_lines_number);
    MPI_Finalize();
    return 0;
}


//-----------------------------------------------------------------------------------------------------------------
// grep.h
//-----------------------------------------------------------------------------------------------------------------
#ifndef GREP_HH
#define GREP_HH

#include <vector>
#include <string>
#include <utility>

namespace grep{

    typedef std::pair< unsigned, std::string >  number_and_line;
    typedef std::vector< number_and_line > lines_found;
    void search_string (const std::string & file_name, const std::string & search_string, lines_found & lines,
                        unsigned & local_lines_number);
    void print_result (const lines_found & lines, unsigned local_lines_number);
}

#endif // GREP_HH


//-----------------------------------------------------------------------------------------------------------------
// grep.cpp
//-----------------------------------------------------------------------------------------------------------------
#include <fstream>
#include <iostream>
#include <sstream>
#include <mpi.h>
#include "grep.hh"

namespace grep{

    void search_string (const std::string & file_name, const std::string & search_string,
                        lines_found & local_filtered_lines, unsigned & local_lines_number){
        int rank (0);
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        const unsigned rk (rank);
        local_lines_number = 0;
        std::ostringstream file_name_builder;
        file_name_builder << file_name << '-' << rk;
        const std::string local_file_name = file_name_builder.str ();
        std::ifstream f_stream (local_file_name);

        // read input file line by line
        for (std::string line; std::getline (f_stream, line); ){
            //increment local number of lines
            ++local_lines_number;
            if (line.find (search_string) != std::string::npos){
                local_filtered_lines.push_back ({local_lines_number, line});
            }
        }
    }
```

```
void print_result (const lines_found & local_filtered_lines, unsigned local_lines_number){
    int rank (0), size (0);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    const unsigned rk (rank);
    const unsigned sz (size);

    if (rk == 0){
        lines_found global_filtered_lines (local_filtered_lines);

        // append to global_filtered_lines the lines found by other processes
        for (unsigned remote_process = 1; remote_process < sz; ++remote_process){
            unsigned remote_lines;
            // receive the number of lines in the local file
            MPI_Recv (&remote_lines, 1, MPI_UNSIGNED, remote_process, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            // receive the number of lines to be received
            unsigned filtered_lines;
            MPI_Recv (&filtered_lines, 1, MPI_UNSIGNED, remote_process, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            // receive lines
            for (unsigned i = 0; i < filtered_lines; ++i){
                unsigned current_line_number;
                MPI_Recv (&current_line_number, 1, MPI_UNSIGNED, remote_process, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                unsigned length = 0;
                MPI_Recv (&length, 1, MPI_UNSIGNED, remote_process, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                std::string new_line (length, '\0');
                MPI_Recv (&new_line[0], length, MPI_CHAR, remote_process, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                // add new line and compute offset
                global_filtered_lines.push_back (std::make_pair (local_lines_number + current_line_number, new_line));
            }
            local_lines_number += remote_lines;
        }

        // print all filtered lines
        for (const number_and_line & n_l : global_filtered_lines){
            std::cout << n_l.first << ": " << n_l.second << std::endl;
        }
    }
    else{
        // send to rank 0 the number of lines of the local file
        // in a way the line number can be updated
        MPI_Send (&local_lines_number, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);
        // send to rank 0 the number of filtered lines
        const unsigned filtered_lines = local_filtered_lines.size();
        MPI_Send (&filtered_lines, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);

        // send lines
        for (const number_and_line & n_l : local_filtered_lines){
            // send line number
            MPI_Send (&n_l.first, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);
            const unsigned length = n_l.second.size();
            // send string length
            MPI_Send (&length, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);
            // send string
            MPI_Send (&(n_l.second)[0], length, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        }
    }
}
```

here we'll store the results (we initialize it with the lines obtained locally by rank 0)

we need to add '\0' otherwise, if we just pass the length the string will be empty

we need to keep track of the overall number of lines

## SCHEME

- if rank == 0:
  - communicate with all processes
    - communicate how many filtered lines
    - for each line:
      - prepare the buffer for receiving the string
      - receive the string
      - receive the local line number
  - write all (from all the processes) lines to cout
- if rank > 0:
  - communicate with rank 0
    - communicate how many filtered lines
    - for each line:
      - send the length of the current string
      - send the string
      - send the local line number

the communication is from all the processes to rank 0 → we don't need a broadcast, instead we need a send/receive

and to add in the associated set the new title, which is performed for every keyword in the search string (hence the very worst case complexity of addMovie is $O(k_1 \cdot (log(k_1) + log(n)))$.

## Exercise 2 (12 points)

You have to implement a **parallel function** that receives as input the extrema of an interval and returns all the prime numbers in the given range. The prototype of the function is the following:

```
std::vector<unsigned> get_prime_numbers (unsigned, unsigned);
```

Moreover, you have to complete the implementation of the **main** function, considering that the two extrema of the range are given by the user through the **command-line**. The partial implementation of the **main** function is reported below:

```
int main (int argc, char *argv[])
{
  // init
  MPI_Init(&argc, &argv);

  // initialize rank and size
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // read parameters from command-line
  unsigned min = // Your code goes here
  unsigned max = // Your code goes here

  // get the vector of prime numbers
  std::vector<unsigned> prime_numbers = get_prime_numbers(min, max);

  // rank 0 prints the prime numbers
  if (rank == 0)
  {
    std::cout << "prime numbers between " << min << " and " << max
          << " are:" << std::endl;
    for (unsigned n : prime_numbers)
      std::cout << n << " ";
    std::cout << std::endl;
  }

  // finalize
  MPI_Finalize();

  return 0;
}
```

In your implementation you can rely on the function:

```
bool is_prime (unsigned n);
```

which returns **true** if n is a prime number and **false** otherwise.

**Note:** the prime numbers returned by the function get_prime_numbers should not necessarily be ordered.

## Solution 2

To complete the **main** function in order to read the two parameters , it is enough to add the following lines:

```
// read parameters from command-line
unsigned min = std::stoi(argv[1]);
```

```
unsigned max = std::stoi(argv[2]);
```

Since the two extrema are read from command-line, they are known to all processors, thus there is no need to communicate them before performing the computation.

The implementation of the function get_prime_numbers is reported below:

```
std::vector<unsigned> get_prime_numbers (unsigned min, unsigned max)
{
  // initialize rank and size
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // initialize vector of prime numbers in each rank
  std::vector<unsigned> my_prime_numbers;

  // loop over the given range (cyclic partitioning)
  for (unsigned n = min + rank; n <= max; n += size)
  {
    // if the current number is prime, insert it in the corresponding vector
    if (is_prime(n))
      my_prime_numbers.push_back(n);
  }

  // collect all partial results in a single vector
  std::vector<unsigned> all_prime_numbers;
  for (int r = 0; r < size; ++r)
  {
    if (r == rank)
    {
      // get and communicate how many prime numbers the current core has found
      unsigned n_primes = my_prime_numbers.size();
      MPI_Bcast(&n_primes, 1, MPI_UNSIGNED, rank, MPI_COMM_WORLD);

      // insert the new values at the end of the vector of prime numbers and
      // communicate them to all the other cores
      unsigned old_n_primes = all_prime_numbers.size();
      all_prime_numbers.insert(all_prime_numbers.end(),
                               my_prime_numbers.cbegin(),
                               my_prime_numbers.cend());

      MPI_Bcast(all_prime_numbers.data() + old_n_primes, n_primes,
                MPI_UNSIGNED, rank, MPI_COMM_WORLD);
    }
    else
    {
      // get the new number of primes
      unsigned n_primes;
      MPI_Bcast(&n_primes, 1, MPI_UNSIGNED, r, MPI_COMM_WORLD);

      // resize the vector of prime numbers and get the new values
      unsigned old_n_primes = all_prime_numbers.size();
      all_prime_numbers.resize(old_n_primes + n_primes);
      MPI_Bcast(all_prime_numbers.data() + old_n_primes, n_primes,
                MPI_UNSIGNED, r, MPI_COMM_WORLD);
    }
  }

  return all_prime_numbers;
}
```

The two parameters **min** and **max** are known to all processors, as well as, of course, all numbers in the range [min, max]. Therefore, no communication is required and the interval can be split among the different cores by using, for instance, a cyclic partitioning schema. This has the advantage that the case in which the length of the interval is not a multiple of the number of available processors is automatically managed without need of additional computations.

Each core relies on the function **is_prime** and stores in the vector **my_prime_numbers** the prime numbers that it finds. All these partial results must be collected in the single vector **all_prime_numbers** before being returned, so that all cores return the full set of prime numbers in the considered interval.

Note that to collect all partial results in **all_prime_numbers** we cannot use a method as **AllGather**. Indeed, all vectors **my_prime_numbers** can have, in principle, different dimension. To deal with this, each core has to communicate, first of all, the size of its **my_prime_numbers** and then all the prime numbers that it has found, that are appended at the end of **all_prime_numbers**.

**Note:** if you still want to use **AllGather**, a possible method would be the following: first of all, the maximum number of primes found by the processors should be communicated through an **AllReduce**, by using **MPI_MAX** to compare the partial results obtained by all processors. Having this value, all cores can resize their vector **my_prime_numbers**, adding at the end some padding (for instance, zeros) that has to be removed at the end from the global result, after the **AllGather**.

## Exercise 3 (8 points)

The provided code implements a data structure called **DocumentStore** that stores and manages **Documents**. A **Document** is characterized by some text (implemented as a **std::string**) and an id (implemented as **size_t**). **DocumentStore** contains two arrays of **Document**: one for completed documents (instance variable **docs**) and one for drafts (instance variable **docsDraft**). The size of the array **docs** is set in the **DocumentStore** constructor and can be configured by the users. The size of **docsDraft** is fixed and equal to 10 (**const DRAFT_SIZE**). Documents can be added to the arrays through methods **addDocument** and **saveAsDraft** respectively. In both cases, if the array is already full no actions are taken. All the documents stored in a **DocumentStore** can be visualized through method **print**.

The header and implementation files of class **Document** are provided along with the header file of **DocumentStore** while **DocumentStore.cpp** is not complete since the implementation of some methods is missing.

**After carefully reading the code, you have to:**

1. **Complete the implementation of DocumentStore** in order to implement a ***like-a-value*** behavior. **Note that while completed documents must be copied between DocumentStore, drafts must not be transferred during the operation but just emptied.** For example, after statement `a = b`, where $a$ and $b$ are DocumentStores, $a$ must store the completed documents of $b$ and no draft.
2. **List and motivate** the program **output** considering the main file provided.

Provided source code:

- **Document.h**

```
class Document {
private:
    string text;
    size_t id;
public:
    Document(){};
    Document(const string& text, size_t id): text(text), id(id){};
    const string& getText() const;
    size_t getId() const;
};
```

- **Document.cpp**

```
const string& Document::getText() const {
    return text;
};
size_t Document::getId() const {
    return id;
};
```