

Algorithms and Parallel Computing

Exercises Book

Danilo Ardagna, Mehrnoosh Askarpour, Danilo Filgueira Mendonça,
Eugenio Gianniti, Federica Filippini, Marco Lattuada, Giovanni Quattroci



This notebook includes a collection of exercises provided to students of the Algorithm and Parallel Computing course during the academic years 2016-17 and 2017-18. Each exercise is presented along with a brief description of the text and the solution.

Moreover, at the end of each chapter, a section collects some frequently asked questions and the relative answers.

The exercises come from exercise and lab sessions and exams and are presented in ascending order of difficulty. Exercises more extended and/or more difficult than what you can find at exams are marked with * (this is not a good reason to skip them but you can consider a good gym!).

Note: Sometimes exams include an appendix reporting some useful routines and a brief explanation of their usage. The same routines are listed here as footnotes (for additional information see www.cppreference.com or www.cplusplus.com).

Special thanks to all the students who collaborated to this work through questions, answers and proposed solutions.

Danilo Ardagna, Mehrnoosh Askarpour, Danilo Filgueira Mendonça,
Eugenio Gianniti, Federica Filippini, Marco Lattuada, Giovanni Quattrocchi

Milan, September 2019

Contents

1 From C to C++	6
1.1 Pointers and References	6
1.1.1 Pointers	6
1.1.2 References	10
1.1.3 Passing by reference VS passing by value	11
1.2 Exercises	13
1.3 Frequently Asked Questions	14
1.3.1 Pointers	14
1.3.2 References	16
2 Classes	18
2.1 Notes	18
2.1.1 Classes and Structs	18
2.2 Exercises	25
Exercise 1 tStock Quotes	25
Exercise 2 tSales Data Archive	28
Exercise 3 tRoots of Polynomials	32
Exercise 4 tGradient Descent	43
Exercise 5 tGradient Descent in \mathbb{R}^n	47
2.3 Challenges Solutions	61
2.4 Frequently Asked Questions	69
3 Inheritance and Polymorphism	72
3.1 Exercises	72
Exercise 1 tEmployees	72
Exercise 2 tRoots of Polynomials	75
Exercise 3 tInterpolation	80
Exercise 4 tKnapsack Problem	86
Exercise 5 tUniversity Information System	92
Exercise 6 tMatrix Computation	94
Exercise 7 tInput Scaler	101
Exercise 8 tK - means *	103
3.2 Frequently Asked Questions	113
4 Copy Control	116
4.1 Exercises	116
Exercise 1 tTaxi	116
Exercise 2 tBoxes	121
Exercise 3 tCalendar	126
Exercise 4 tDocument Store	130
4.2 Frequently Asked Questions	136

5 Smart Pointers	138
5.1 Exercises	138
Exercise 1 tTripAdvisor®	138
Exercise 2 tDoctor	140
Exercise 3 tLocal Search*	143
Exercise 4 tFraud Detection	154
6 Standard Template Library - use of containers	173
6.1 Exercises	173
Exercise 1 tStock Quotes	173
Exercise 2 tInstant messenger	175
Exercise 3 tCounter Queue	178
Exercise 4 tStreaming System	180
Exercise 5 tErdos numbers	183
Exercise 6 tNational Healthcare	188
Exercise 7 tStudent Search	195
Exercise 8 tDataframe	205
Exercise 9 tSparse Matrix	219
Exercise 10 tHashMap*	222
Exercise 11 tDeque*	226
Exercise 12 tHeap*	233
6.2 Frequently Asked Questions	239
7 MPI	240
7.1 Exercises	240
Exercise 1 tCommand line arguments	240
Exercise 2 tDiscrete Gradient	242
Exercise 3 tDiscrete Laplacian	246
Exercise 4 tAscending Sort	247
Exercise 5 tDot Product	250
Exercise 6 tMonte Carlo Method	252
Exercise 7 tMatrix Multiplication	254
Exercise 8 tPower Method	259
Exercise 9 tSearch Engine	262
Exercise 10 tWord-Count	267
Exercise 11 tParallel Grep	276
Exercise 12 tParallel K-means*	280
Exercise 13 tParallel Gradient Descent*	287
7.2 Challenges Solutions	294
7.3 Frequently Asked Questions	301
7.3.1 General questions	301
7.3.2 Broadcast	302
7.3.3 Scatter and Gather	303
8 Appendix	305
8.1 Advanced topics on pointers	305
8.2 Templates and STL	308
Exercise 1 tDataframe	308
Exercise 2 tHashMap	321
Exercise 3 tSparse Matrix	324
Exercise 4 tHeap	328
Exercise 5 tSocial Network	334

Exercise 6 tCl	340
Exercise 7 tCounting Sort	342
Exercise 8 tGenetic Algorithms	346
Exercise 9 tDeque	360
8.3 Frequently Asked Questions	367

Chapter 1

From C to C++

1.1 Pointers and References

1.1.1 Pointers

A pointer ¹ is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The only difference among pointers of different data types is the data type of the variable or constant that the pointer points to. You declare a pointer as follows:

```
base_type * pointer_name;
```

A pointer can be initialized either by the address of a variable or by the value of another pointer, in case you know the exact address to assign. Otherwise, the pointer should be defined **nullptr** (null pointer).

There are two operators you need to know:

1. ***ptr**: returns the value of the variable located at the address specified by **ptr**.
2. **&myvar**: returns the address of **myvar** in the memory.

```
int myvar;
int * foo = &myvar;
int * bar = foo;
int * ptr = nullptr;
```

Examples:

1. Define a variable and a pointer to it.

```
1 int value = 5;
2 int * p;
3 p = &value;
4 std::cout << "The value pointed by p is " << *p << std::endl;
```

The code above produces the output

```
The value pointed by p is 5
```

2. Define two pointers that point to the same variable.

¹Credits:

<https://www.tutorialspoint.com/cplusplus>
www.cplusplus.com
<https://www.tutorialspoint.com/>

```

1 int value = 5;
2 int * p;
3 p = &value;
4 int * p2 = p;
5 std::cout << "The value pointed by p is " << *p << std::endl;
6 std::cout << "The value pointed by p2 is " << *p2 << std::endl;

```

The code above produces the output

```

The value pointed by p is 5
The value pointed by p2 is 5

```

because, first, in line 3, we assign to **p** the address of **value**, then, in line 4, we initialize **p2** through **p**, so that both the pointers point to the same location in the memory.

3. Change **value** from 5 to 10 through pointers.

```

1 int value = 5;
2 int * p;
3 p = &value;
4 *p = 10;
5 std::cout << "The value pointed by p is " << *p << std::endl;
6 std::cout << "New value is " << value << std::endl;

```

The code above produces the output

```

The value pointed by p is 10
New value is 10

```

because in line 3 we initialize the pointer **p** through the address of **value**, then we update ***p**, i.e. we modify the value of the variable pointed by **p**.

Pointers and Arrays

Example:

Define an array of length five and initialize it by values 10,20,30,40,50 by using pointers.

```

1 int numbers[5];
2 int *p;
3 p = numbers;
4 *p = 10;
5 p++;
6 *p = 20;
7 p = &numbers[2];
8 *p = 30;
9 p=numbers+3;
10 *p=40;
11 p = numbers;
12 *(p+4) = 50;
13 for (int n=0; n<5; n++)
14     std::cout << numbers[n] << '\t';
15 std::cout << std::endl;

```

The name of array always represents the address of its first element. This means that, when, in line 3, we assign

p = numbers;

what we are assigning to **p** is the address of **numbers[0]**, i.e. of the first element of the array

numbers. We would get the same result writing

```
p = &numbers[0];
```

Given this, the following instruction ($*p = 10$, in line 4) assigns to the first element of numbers the value 10.

$p++$, in line 5, updates p so that it points to the following element, i.e., in this case, the second element of numbers, that in line 6 takes the value 20.

In general, the instruction

```
p += N;
```

where N is an integer, makes p point to an element which is N steps apart from the current one (if, for example, p points to the element of the array with index 0, $p+=3$ makes it point to the element of index 3, i.e. the fourth one).

Note that, in principle, the integer N can be either positive or negative. Of course, in doing an operation like $p+=N$, we must be sure that we do not exceed the dimension of the array. Lines 7 and 8 assign to p the address of the third element of the array and then they initialize it with the value 30.

Since, as we said before, the name of an array corresponds to the address of its first element, the instruction of line 9 makes p point to the element of numbers which is 3 steps apart from the first one, i.e. to the element with index 3 (the fourth element of the array).

Finally, the instruction

```
*(p+4) = 50;
```

assigns to the fifth element of numbers the value 50. Indeed, $p+4$ is a pointer to the element of numbers with index 4 and, by the operator `*`, we can deferenciate it and change its value. The overall output of the program is

```
10 20 30 40 50
```

Can we always interchange arrays and pointers?

Example:

Which block is correct?

```
1 int var[3] = {10, 100, 200};  
2 int *ptr;  
3 ptr = var;  
4 for (int i = 0; i < 3; i++)  
5 {  
6     std::cout << "var[" << i << "] = ";  
7     std::cout << *ptr << std::endl;  
8     ptr++;  
9 }
```

```
1 int var[3] = {10, 100, 200};  
2 int *ptr;  
3 ptr = var;  
4 for (int i = 0; i < 3; i++)  
5 {  
6     std::cout << "var[" << i << "] = ";  
7     std::cout << *ptr << std::endl;  
8     *var = i;  
9     var++;  
10 }
```

The answer is the one on the left: `var++`, in line 9 of the right block, is a wrong syntax. Indeed, the name of an array represents a pointer to its first element, i.e. it stores a constant address. Thus, we cannot use the name of the array to scan the elements of the array itself. On the other side, in the left block we use a pointer to go through the elements of var, which can be done as we saw in the previous example.

Pointer to Pointer

A pointer to another pointer is defined as:

```
base_type ** pointer_name;
```

A very simple example is shown below.

```
1 int x = 2;  
2 int * ptr;
```

```

3 int ** pptr;
4 ptr = &x;
5 pptr = &ptr;
6 std::cout << "x is " << x << std::endl;
7 std::cout << "The value pointed by ptr is " << *ptr << std::endl;
8 std::cout << "*pptr is " << *pptr << std::endl;
9 std::cout << "**pptr is " << **pptr << std::endl;

```

The code above produces the output

```

x is 2
The value pointed by ptr is 2
*pptr is 0x7fff56ced99c
**pptr is 2

```

Indeed, `ptr`, which is a pointer to `int`, is initialized in line 4 by the address of the variable `x`, while `pptr`, which is a pointer to a pointer to `int`, is initialized in line 5 by the address of `ptr`. Thus, if we dereference once `pptr`, as in line 8, we get the value of `ptr`, i.e. the address of `x` (which of course is always different any time we run the program). If, in turn, we dereference it twice, as in line 9, we get the value of `x`, which is 2.

Pointers and Functions

You can pass pointers to functions. The following example shows how a function with pointer argument is used and how is different from a normal function.

```

1 void modifyref(int *a)
2 {
3     *a *= 2;
4 }
5
6 void modify(int a)
7 {
8     a *= 2;
9 }
10
11 int main()
12 {
13     int x = 2;
14     int y = 2;
15     modifyref(&x);
16     modify(y);
17     std::cout << "the value of x is " << x << std::endl;
18     std::cout << "the value of y is " << y << std::endl;
19     return 0;
20 }

```

`modifyref` changes the variable `x`, while `y` is just copied in parameter `a` and it remains unchanged.

The code produces the output

```

the value of x is 4
the value of y is 2

```

You can also define functions that return pointers.

```

1 int* createVar( )

```

```

2 {
3     int *var = new int [3]{10, 100, 200};
4     return var;
5 }
6
7 int main()
8 {
9     int *parray = createVar();
10    for (int i = 0; i < 3; i++)
11        std::cout << *(parray+i) << std::endl;
12    delete[] parray;
13    return 0;
14 }
```

The code produces the output

```

10
100
200
```

1.1.2 References

A reference variable ² is an alias, that is another name for an already existing variable.

```

1 int main()
2 {
3     int i;
4     int & ri = i;
5     ri = 10;
6     std::cout << i << " " << ri << std::endl;
7     return 0;
8 }
```

The code above produces the output

```

10 10
```

because `ri` and `i` are the same variable.

References vs Pointers

1. Null references do not exist. A reference needs to be initialized as it is created.
2. Once a reference is initialized to an object, it cannot be changed so that it refers to another object.

References and Functions

Remember the example where we passed a pointer to a function? Lets change it to see call by reference:

²Credits:

<https://www.tutorialspoint.com/cplusplus>
[wwwcplusplus.com](http://www.cplusplus.com)
<https://www.tutorialspoint.com/>

```

1 void modifyref2(int &a)
2 {
3     a *=2;
4 }
5
6 void modify(int a)
7 {
8     a *=2;
9 }
10
11 int main()
12 {
13     int x = 2;
14     int y = 2;
15     modifyref2(x);
16     modify(y);
17     std::cout << "the value of x is "<< x << std::endl;
18     std::cout << "the value of y is "<< y << std::endl;
19     return 0;
20 }
```

This is, in general, a more clean and common way to modify variables through functions.
The code produces the output

```
the value of x is 4
the value of y is 2
```

because **x**, which is passed by reference to `modifyref2()`, is multiplied by 2, while **y**, whose value is only copied in **a**, remains untouched.

References to const variables

A reference to a **const** variable is defined as:

```
const int ci = 1024;
const int &r1 = ci;
```

Example:

Is the code below correct?

```
1 const int ci = 1024;
2 const int &r1 = ci;
3 int &r2 = ci;
```

The answer is no: we cannot bind a non const reference with a const variable (as we try to do in line 3) because we should not be able to change a constant value. If the last line worked, we could change the value of **ci** through **r2**.

1.1.3 Passing by reference VS passing by value

Passing by reference: Both pointers and references, despite their different syntax, provide what is called reference semantics. Accordingly, you can define functions that receive parameters *by reference* using either pointers or references. In both cases, you can use the pointer / reference to change the value of the original variable.

Functions 1 - 4 in the example below have been defined so that you call them passing an argument by reference.

Passing by value: When you pass a parameter to a function "by value", a copy of the object, and not the object itself, is passed. As a consequence, to change the value of the local object inside the function does not affect the value of the original object.

In the example below, function 5 was defined so that you call it passing by value.

Const qualifier: By using the **const** qualifier (as is done, in the example below, in functions 3 and 4) as part of the declaration of an argument, you prevent the function from changing the value of that argument. This is especially important when combined with reference semantics: in some cases, you want to pass a pointer / reference as a function parameter (for example to avoid copying objects with possibly huge dimensions, such as vectors), but you do not want the function to be able to modify the value of the original variable pointed / referenced.

Overloaded functions: In the example below, functions 1- 4 have been defined with the same name `sample_ref`. This is possible because the compiler is smart enough to distinguish between a function that receives a pointer to a variable and a function that receives the actual variable OR a reference to that variable. It also distinguishes parameters with the **const** qualifier (e.g., functions 3 and 4).

Notice that you cannot overload functions only on the base of the reference / plain variable argument they receive. For instance, since function 2 has already been defined with a reference as parameter, you cannot overload it with a declaration that receives a plain variable (e.g., function 5, which, indeed, is defined with a different name).

```
1 struct MyObj      // a struct that stores an integer value
2 {
3     int i;
4 };
5
6 void sample_ref (MyObj * o)    // function 1
7 {
8     o -> i++;   // automatically dereferences pointer o and increments the value of i
9         // from MyObject obj
10 }
11
12 void sample_ref (MyObj & o)    // function 2
13 {
14     o.i++; // uses the reference o to increment the value of i from MyObject obj
15 }
16
17 void sample_ref (const MyObj * o)    // function 3
18 {
19     // nothing can be changed here because o is a pointer to a const;
20     o -> i;    // you can still read the value of i
21     // o -> i++ ; would result in a compilation error
22 }
23
24 void sample_ref (const MyObj & o)    // function 4
25 {
26     // nothing to be changed here because o is a reference to a const
27     o.i;        // you can still read the value of i
28     // o. i++ ; would result in a compilation error
29 }
30
31 void sample_value (MyObj o)    // function 5
32 {
```

```

33     o.i++;      //changing the value of i here will not change the value of obj.i
34 }
35
36 int main (void)
{
37     MyObj obj = {10};
38     const MyObj c_obj = {10};
39     sample_ref(&obj);  // calls 1
40     sample_ref(obj);  // calls 2
41     sample_ref(&c_obj); //calls 3
42     sample_ref(c_obj); //calls 4
43     sample_value(obj); //calls 5
44 }
45 }
```

Remember: in general, to access the value pointed by a pointer, e.g. by the pointer `o`, you write:

```
*o
```

However, if you want to access a member of the pointed object (in this case, the member `i` from `MyObject`), you can use the arrow operator and access it directly, without explicitly dereferencing the object `o`; in this case, you write:

```
o->i
```

Alternatively, you can dereference the pointer explicitly, by writing:

```
(*o).i
```

but this is, in general, less readable. Consider that in this case you must use the dot operator instead of the arrow operator, as the result of `(*o)` is the object itself.

1.2 Exercises

Some exercises about pointers, references and consts:

- Assuming that `unsigned fun (std::string &t)` is a function that replaces the occurrences of lowercase vowels in `t` with the character `x` and returns the count of replaced characters:

```

1 #include <string>
2 #include <iostream>
3 using namespace std;
4
5 unsigned fun (string &t)
{
6     unsigned count = 0;
7     for (char & c : t)
8     {
9         if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
10            {
11                c = 'x';
12                count++;
13            }
14        }
15    }
16    return count;
17 }
```

find the problem in the following code and fix it:

```
19 int main (void)
20 {
21     const string s1 = "Hello World";
22     unsigned count = fun(s1);
23     string s2 = s1;
24     count = fun(s2);
25     const string &s3 = s1;
26     count = fun(s3);
27     string &s4 = s1;
28     count = fun(s4);
29     cout << s4 << " " << count << endl;
30     return 0;
31 }
```

Once the code has been fixed, what would be the output of the program?

ANSWER: First of all, line 22 is wrong: the function `fun`, as it is defined in line 5, must receive as input a reference to a non constant string. Since `s1` is defined in line 21 as `const`, it cannot be passed to the function.

In line 25, a reference to a `const string`, namely `s3`, is correctly initialized with the value of `s1`; however, to pass it to `fun` as we try to do in line 26 leads to the same error that we had trying to pass `s1`.

Finally, in line 27, a reference to a non constant string, `s4`, is initialized with the constant string `s1`, which is an invalid operation.

You can fix the code by dropping the `const` qualifier of `s1` and by defining `s3` as a reference to a non constant string. Given that, the original string "Hello World" is transformed to "Hxllx Wxrld".

Additionally, the variable `count` stores the value returned by the last call of `fun`, therefore 0. Indeed, in the function `fun` we rely on the reference semantic: after the first call, in which `fun` receives the string "Hello World", it replaces it with "Hxllx Wxrld" and it returns 3, all the subsequent calls return 0. This happens because references to the same string are passed as parameters and this means that no more replacements are performed (`fun` always receives "Hxllx Wxrld").

Therefore, the output of the program is

```
Hxllx Wxrld 0
```

1.3 Frequently Asked Questions

1.3.1 Pointers

1. What happens in the memory when I define a variable and a pointer inside a function?
How can I avoid memory leaks?

ANSWER: If you define a function like this:

```
1 void f (int a)
2 {
3     int b = 1;
4     int * c = new int (5);
5     // do something with a, b and c
6     delete c;
7     return;
8 }
```

the memory for the parameter **a** and for the local variables **b** and **c** is allocated in the stack, which means that it is automatically deallocated when you leave the scope of the function. On the other hand, since the definition of **c** is made through a **new**, a new object of type **int** and value 5 is allocated in the heap (i.e. the local variable **c** will store, in the stack, the address of the new **int**, located in the heap).

You create a memory leak if you do not free the memory allocated by **new** through a **delete** before leaving the scope of the function. Indeed, the local variable **c** will be destroyed and the **int** pointed by **c**, allocated in the heap, will stay there, unattainable, forever.

Therefore, you always have to delete a pointer defined through a **new** within a function, unless you return it.

Indeed, if your function returns the pointer, i.e. you have

```

1 int * f (int a)
2 {
3     int b = 1;
4     int * c = new int (5);
5     // do something with a, b and c
6     return c;
7 }
```

when you leave the scope of the function the local variable **c** is deallocated, but its value, i.e. the address of the new **int** in the heap, is copied in a new variable when you write, for example, in the **main** function:

```

int * d = f(4);
// do something useful with d
delete d;
```

In this case, you do not delete **c** in the function **f** because you do not want to lose the address of the new **int**, but you have to write **delete d**; within the **main** function, in order to avoid a memory leak when the program finishes its work.

2. When I write a code like this

```

1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     int x = 7;
7     int *p = new int;
8     p = &x;
9     *p = 9;
10    cout << "value of x == " << x << endl;
11    delete p;
12    return 0;
13 }
```

the compiler tells me that deleting **p** is invalid; why?

ANSWER: You can delete only pointers to dynamically allocated memory (i.e. memory allocated via **new**) or **nullptrs**. In the example above, you define **p** through a **new int**, but then you leak this memory writing **p = &x**. Indeed, this instruction copies in **p** the address of the automatic variable **x**. This means that you create a memory leak, because the previous value of **p** is overwritten and the new **int** allocated

in the heap in line 7 remains unattainable.

If you want to assign to `p` the same value stored in `x`, you have to replace line 8 with:

```
*p = x;
```

(as you do in the following line in order to assign to `p` the value 9). This instruction does not allocate memory, it only copies the value of `x` in the memory slot pointed by `p`.

On the other hand, if you want `p` to be linked to `x`, i.e. to store its address, you simply have to define it, in line 7, as

```
int * p = &x;
```

without using `new` (and you must not delete it because `p` points to a statically declared variable).

1.3.2 References

1. Knowing that a reference can be initialized only once, when I write a code like this

```
1 #include <iostream>
2
3 int main (void)
4 {
5     int x = 10;
6     int & r = x;
7     std::cout << "x = " << x << "; r = " << r << std::endl;
8     x = 7;
9     std::cout << "x = " << x << "; r = " << r << std::endl;
10    r = 6;
11    std::cout << "x = " << x << "; r = " << r << std::endl;
12    int y = 8;
13    r = y;
14    std::cout << "x = " << x << "; r = " << r << "; y = " << y << std::endl;
15    y = 5;
16    std::cout << "x = " << x << "; r = " << r << "; y = " << y << std::endl;
17    return 0;
18 }
```

what is the meaning, in line 12, of the assignment `r = y`?

ANSWER: We can see a reference as an "alternative name" for a variable, which means that, once you have defined, in line 5,

```
int & r = x;
```

the variables `r` and `x` are linked forever: if you change the value of `x`, you change accordingly also the value of `r` and viceversa (this is what happens when you write `x = 7`, in line 7, and then `r = 6` in line 9).

When you write, in line 12, `r = y`, what you are doing is to copy the value of `y` in `r` (and thus in `x`), exactly as happens if you write `x = y` or if you write `r = 6`.

To reinitialize `r` to be a reference to `y` is not only forbidden by definition, but it is syntactically impossible: with a pointer, you can write:

```
int a = 3;
int * p = &a;
int b = 7;
p = &b;
```

assigning to `p` first the address of `a` and then the address of `b`. With a reference, you cannot write in any way something as `r = &y`: the operator `&` gives you a pointer (the address of `y`), which cannot be automatically casted into an integer.

Chapter 2

Classes

2.1 Notes

2.1.1 Classes and Structs

Struct

At the most basic level, a data structure ¹ is a way to group together related data elements and a strategy for using those data. A **struct**, as in the C programming language, is defined as follows:

```
1 struct [structure tag]
2 {
3     member definition;
4     member definition;
5     ...
6     member definition;
7 } [one or more structure variables];
```

Consider the following example:

Example:

Define a structure for a payment check, which contains the payer's name, the date of the payment, the reason for the money transfer, and the transferred amount.

```
1 #include <iostream>
2 #include <string>
3
4 struct CheckInfo
5 {
6     std::string name;
7     std::string date;
8     std::string reason;
9     float amount;
10 };
11
12 int main (void)
13 {
14     CheckInfo check1;
```

¹Credits:

<https://www.tutorialspoint.com/cplusplus>
wwwcplusplus.com
<https://www.tutorialspoint.com/>

```

15     check1.name = "Mr. X";
16     check1.date = "29/06/2018";
17     check1.reason = "rent";
18     check1.amount = 25.5;
19     std::cout << check1.name << std::endl;
20 }
```

Class

In C++ we usually define our own data structures by defining a **class**.² A **class** defines a type along with a collection of operations that are related to that type. The **class** mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to give the possibility to define **class** types that behave as naturally as the built-in types.

Example:

Define a **class** for the previous example.

```

1 #include <iostream>
2 #include <string>
3
4 class CheckInfo {
5     public:
6         std::string name;
7         std::string date;
8         std::string reason;
9         float amount;
10    };
11
12 int main (void)
13 {
14     CheckInfo check1;
15     check1.name = "Mr. X";
16     check1.date = "29/06/2018";
17     check1.reason = "rent";
18     check1.amount = 25.5;
19     std::cout << check1.name << std::endl;
20 }
```

What is the meaning of the keyword **public** that we add within the class definition? In general, a **public** member of a **class** is accessible from elsewhere (both inside and outside the scope of the **class** itself), while a **private** member can be accessed only within the scope of the **class**.

An important difference between **structs** and **classes** is the default accessibility of their members: default accessibility for a **class** is **private** while it is **public** for a **struct**. If we want a member of our **class** to be accessible outside of the **class** itself, we must specify in the definition that the member has to be **public**.

In general, we do not want to make everything **public!** In the example above, for instance, we do not want to let the amount of the check to be modifiable. In order to avoid this, we should define it as a **private** member.

²Credits:

<https://www.tutorialspoint.com/cplusplus>
www.cplusplus.com
<https://www.tutorialspoint.com/>

Usually, each class has a section where **private** variables and functions are defined and another section for **public** variables and functions.

Example:

```
1 #include <string>
2
3 class CheckInfo {
4     public:
5         std::string name;
6         std::string date;
7         std::string reason;
8     private:
9         float amount;
10};
```

Constructors

A **class constructor** is a special member function of a **class** that is executed whenever we create new objects of that **class**. The compiler automatically synthesizes a **default constructor**, with no parameters, for every declared **class**. From now on, we will consider a simpler **class** than the one of the examples above, in which we have only the members **name** and **amount**. Given this class, namely:

```
1 #include <string>
2
3 class CheckInfo {
4     private:
5         std::string name;
6         float amount;
7 };
```

we can use the default constructor in order to instantiate an object of type **CheckInfo**, writing:

```
CheckInfo check1;
```

If you define a **constructor** function, you lose the **default** one and the code above gives you an error. For example, if you define:

```
1 #include <string>
2
3 class CheckInfo {
4     public:
5         CheckInfo (const std::string &n, float a): name(n), amount(a) {}
6     private:
7         std::string name;
8         float amount;
9 };
```

you must initialize objects of type **CheckInfo** writing:

```
CheckInfo check2("Mr. Payer",150.0);
```

If you want to be able to create **CheckInfo** objects both specifying name and amount and without providing those information, you must tell explicitly that you want also the **default constructor** without parameters, i.e. you must write:

```

1 #include <string>
2
3 class CheckInfo {
4     public:
5         CheckInfo (void) = default;
6         CheckInfo (const std::string &n, float a): name(n), amount(a) {}
7     private:
8         std::string name = "";
9         float amount = 0.0;
10    };

```

Note (1): of course, the constructors should be **public** methods of the class, otherwise you would not be able to create objects of the **class** type!

Note (2): when the **default** constructor is invoked, the **class** members are initialized, depending on the compiler, with either a **default** initializer or a value initializer. This can lead to some issues because, for example, the **default** value for **std::strings** is an empty string (which is what we expect), but the **default** value for **ints** or **doubles** is a random number. Therefore, it is always a good practice to provide a **default** value for the members of the class when we have a constructor that takes no parameters, as we do in lines 8 and 9 of the example above. Indeed, whenever those values are provided, the compiler uses them in order to initialize the corresponding variables if we use the constructor without parameters.

Destructors

A **destructor** is a special member function of a **class** that is executed whenever an object of the **class** goes out of scope or whenever the **delete** expression is applied to a pointer to an object of that **class**.

```

1 CheckInfo::~CheckInfo()
2 {
3     std::cout << "Destructing" << name << std::endl;
4 }

```

During the life time of an object, the following events happen:

1. the **constructor** is called when control flow reaches the line in which the object is declared.
2. enough memory is allocated on the stack to store the object.
3. when the object goes out of scope, the **destructor** runs automatically and the memory is freed.

Example:

Assuming that we have:

```

1 #include <iostream>
2 #include <string>
3
4 class CheckInfo {
5     private:
6         std::string name;
7         float amount;
8     public:
9         CheckInfo (const std::string &n, float a);
10        ~CheckInfo (void);

```

```

11 };
12
13 CheckInfo::CheckInfo (const std::string &n, float a): name(n), amount(a)
14 {
15     std::cout << "Constructing " << name << std::endl;
16 }
17
18 CheckInfo::~CheckInfo (void)
19 {
20     std::cout << "Destructing " << name << std::endl;
21 }
```

the output of the following program

```

1 int main (void)
2 {
3     CheckInfo c1("Payer1",100.0);
4     {
5         CheckInfo c2("Payer2",200.0);
6     }
7 }
```

would be:

```
Constructing Payer1
Constructing Payer2
Destructing Payer2
Destructing Payer1
```

Member functions

A member function (also called method) of a **class** is a function that has its definition or its prototype within the **class** definition like any other variable. It operates on any object of the **class** of which it is a member and it has access to all the members of the class (both **private** and, of course, **public**).

Member functions can be defined in two ways. You can either:

- declare and define your member functions as part of the class, writing the definition within the header file (inline definition), OR
- declare the function (i.e. write its prototype) inside the class definition within the header file and define it (i.e. write its implementation) outside the class, generally inside the source file named after the class.

Example:

Define a member function for **CheckInfo** class that combines the amounts of two checks.

```

1 CheckInfo& CheckInfo::combine (const CheckInfo &c)
2 {
3     amount += c.amount;
4     return *this;
5 }
```

You can overload this function and define another one that actually combines the amounts of multiple checks.

```

1 CheckInfo& CheckInfo::combine (const CheckInfo &c, const CheckInfo &d)
2 {
3     amount += c.amount + d.amount;
4     return *this;
5 }
```

If those methods are declared as **public**, in the **main** function, once you defined an object of type **CheckInfo**, you can call the methods writing:

```

1 // code
2 CheckInfo c1, c2, c3;
3 c3.combine(c1,c2);
4 // code
```

Example:

Define member functions that can get or set values of class variables

```

1 float CheckInfo::get_amount (void) const
2 {
3     return amount;
4 }
5
6 void CheckInfo::set_amount (float a)
7 {
8     amount = a;
9 }
```

Friend functions

Normally, every free function (i.e. every function which is not a member of the **class**) which receives as parameter an object of a **class** can only access to its **public** members. If we want to allow a free function to access all the non public members of a **class**, we must declare it as a **friend** function.

According to the C++ specification, **friend** functions must be:

- declared as **friend** inside the **class**
- normally declared outside the **class**
- defined outside the **class**

However, many compilers do not enforce this, meaning that they accept a single declaration inside the class (or even inline definitions).

Example:

If we want to have a function that sums the amounts of two given **CheckInfo** objects and returns the result, we have two possibilities:

1. We can define a member function that does the job, writing:

```

1 class CheckInfo {
2     public:
3         CheckInfo (float a): amount(a) {}
4         float sum_amount (const CheckInfo &z) const;
5     private:
6         float amount = 0.0;
7 };
```

```

8
9 float CheckInfo::sum_amount (const CheckInfo &c) const
10 {
11     return (amount + c.amount);
12 }
```

2. we can define it as a normal free function (but we need a getter to read the value of amount, since it is **private** and cannot be seen outside of the **class** scope).

```

1 class CheckInfo {
2     public:
3         CheckInfo (float a): amount(a) {}
4         float get_amount (void) const {return amount;}
5     private:
6         float amount = 0.0;
7     };
8
9 float sum_amount (const CheckInfo &c1, const CheckInfo &c2)
10 {
11     return (c1.get_amount() + c2.get_amount());
12 }
```

3. OR we can declare it as a **friend** function. In this way, even if **amount** is a **private** member of the class (thus inaccessible by normal free functions), we can write:

```

1 class CheckInfo {
2     public:
3         friend float sum_amount (const CheckInfo &c1, const CheckInfo &c2);
4         CheckInfo (float a): amount(a) {}
5     private:
6         float amount = 0.0;
7     };
8
9 float sum_amount (const CheckInfo &c1, const CheckInfo &c2)
10 {
11     return (c1.amount + c2.amount);
12 }
```

Note (1): only member functions can be declared as **const**.

Note (2): in the first case, when we define **sum_amount** as a member function, we only need to pass one parameter: the other **CheckInfo** object is the one through which we call the function.

Note (3): the way we call the function **sum_amount** is different whether it is a member function or a free function (friend or not). Indeed, in the first case we write:

```

1 // code
2 CheckInfo c1(4.0);
3 CheckInfo c2(5.0);
4 float sum = c1.sum_amount(c2);
5 // code
```

while in the other cases we write:

```

1 // code
2 CheckInfo c1(4.0);
```

```

3 CheckInfo c2(5.0);
4 float sum = sum_amount(c1,c2);
5 // code

```

2.2 Exercises

Exercise 1 Stock Quotes

The aim of this exercise is to implement a program for the U.S. stock quotes market, in order to manage real-time stock quotes data on all public companies in the U.S.

A stock quote is characterized by its unique symbol (NASDAQ symbols are four or five characters in length) and the last sale price values (the price at which a stock last traded during regular market hours), quoted in U.S. dollars and cents.

We need to define two classes, namely `Stock_quote` and `Stock_quote_archive`.

In particular, the `Stock_quote` objects keep track of the price history in a `std::vector`, whilst the `Stock_quote_archive` class stores stock quotes data in another `std::vector`.

Moreover, we want to provide the implementation of the following methods of the `Stock_quote_archive` class:

- `add_stock_quote`, which adds a stock quote to the archive.
- `add_last_sale_price`, that, given a stock quote symbol and a price, appends the price to the vector in the archived stock quote represented by the symbol.

What is `add_last_sale_price` worst case complexity?

Exercise 1 - Solution

Any `Stock_quote` object stores a `std::string` with the relative NASDAQ symbol and a `std::vector<float>` with the prices (we can use `floats` and not `doubles` because we do not need to store numbers with an huge number of digits). Since both are private members (thus inaccessible outside of the class), we need a public method, namely `get_symbol`, which returns the symbol of the `Stock_quote`. In principle, even if we do not need it for this exercise, we could define also a method which returns the vector of prices (or a single price, given an index).

The constructor receives as input a `std::string`, which is used to initialize the variable `symbol`; for the vector of prices, we rely on the default constructor of `std::vectors` (which creates an empty vector).

***** file `Stock_quote.h` *****

```

1 #ifndef STOCKQUOTE_H
2 #define STOCKQUOTE_H
3
4 #include <vector>
5 #include <string>
6 #include <iostream>
7
8 class Stock_quote
9 {
10     std::string symbol;
11     std::vector<float> last_prices;

```

```

12
13 public:
14 Stock_quote (const std::string & s): symbol (s) {}
15 void add_last_sale_price (float value);
16 void print (void) const;
17 std::string get_symbol (void) const
18 {
19     return symbol;
20 }
21 };
22
23 #endif //STOCKQUOTE_H

```

***** file Stock_quote.cpp *****

```

1 #include "Stock_quote.h"
2
3 void Stock_quote::add_last_sale_price (float value)
4 {
5     last_prices.push_back (value);
6 }
7
8 void Stock_quote::print (void) const
9 {
10    std::cout << symbol << std::endl;
11
12    for (const float val : last_prices)
13        std::cout << val << std::endl;
14 }

```

The class `Stock_quote_archive` stores a `std::vector` of `Stock_quote`s. Other than the two public methods required by the exercise, it has a private method, called `find`, which returns an iterator to a `Stock_quote` object. This method is defined as `private` because it is only intended to be used in the implementation of `add_last_sale_price`, not outside of the class. Even if the method `find` does not directly modify the object, it cannot be declared as `const`. Indeed, the iterator that it returns will be used inside the method `add_last_sale_price` to modify the vector `stock_quotes` adding a new price and therefore it cannot be a `const_iterator`.

***** file Stock_quote_archive.h *****

```

1 #ifndef STOCKQUOTEARCHIVE_H
2 #define STOCKQUOTEARCHIVE_H
3
4 #include "Stock_quote.h"
5
6 class Stock_quote_archive
7 {
8     std::vector<Stock_quote> stock_quotes;
9     std::vector<Stock_quote>::iterator find(const std::string & stock_symbol);
10
11 public:
12     void add_stock_quote (const Stock_quote & s);
13     void print (void) const;

```

```

14 void add_last_sale_price (const std::string & stock_symbol, float value);
15 }
16
17 #endif //STOCKQUOTEARCHIVE_H

```

***** file Stock_quote_archive.cpp *****

```

1 #include "Stock_quote_archive.h"
2
3 void Stock_quote_archive::print (void) const
4 {
5     std::vector<Stock_quote>::const_iterator it = stock_quotes.cbegin();
6
7     while (it != stock_quotes.cend())
8     {
9         it->print();
10        ++it;
11    }
12 }
13
14 void Stock_quote_archive::add_stock_quote (const Stock_quote & s)
15 {
16     stock_quotes.push_back (s);
17 }
18
19
20 std::vector<Stock_quote>::iterator
21 Stock_quote_archive::find(const std::string & stock_symbol)
22 {
23     bool found_symbol=false;
24
25     std::vector<Stock_quote>::iterator it = stock_quotes.begin();
26
27     while (it != stock_quotes.end() && ! found_symbol){
28         if (it->get_symbol()==stock_symbol)
29             found_symbol=true;
30         else
31             ++it;
32     }
33     return it;
34 }
35
36 void
37 Stock_quote_archive::add_last_sale_price (const std::string & stock_symbol,
38                                         float value)
39 {
40     std::vector<Stock_quote>::iterator it = find(stock_symbol);
41
42     if (it != stock_quotes.end())
43         it->add_last_sale_price (value);
44 }

```

The method `Stock_quote_archive::print` (lines 3 to 12 of the file "Stock_quote_archive.cpp") is defined on top of the corresponding method for the class `Stock_quote`; indeed, it loops over all the elements stored in the vector `stock_quotes` and, for any of those elements, it calls the function `Stock_quote::print`.

The method `Stock_quote_archive::find` (lines 20 to 34) is designed in order to behave as the function `find` defined by the Standard Template Library; in particular, it receives as parameter a `std::string` that represents a symbol and it loops over the elements of `stock_quotes` until it reaches `stock_quotes.end()` or it finds the symbol. If a `Stock_quote` with the given symbol is stored in the vector, it returns the iterator to that object; otherwise, it returns `stock_quotes.end()`.

The method `Stock_quote_archive::add_last_sale_price` (lines 36 to 44) relies on the method `Stock_quote_archive::find`. Indeed, it uses it to get the iterator to the element with the given symbol and then it calls the method `Stock_quote::add_last_sale_price` to add to that element the new price. In particular, before adding the new price it checks if the iterator returned by `Stock_quote_archive::find` is not `stock_quotes.end()` (i.e. it checks if the given symbol is actually stored in the vector).

The worst case complexity of the method `Stock_quote_archive::add_last_sale_price()` is given by the sum between the complexity of `Stock_quote_archive::find()` and of `Stock_quote::add_last_sale_price()`.

If we call N the number of elements stored in `stock_quotes`, in the worst case (i.e. if the given symbol does not exist) the complexity of the function `find` is $O(N)$ because it has to loop over all the elements.

The function `Stock_quote::add_last_sale_price`, in turn, performs only a `push_back`, which in the worst case has a complexity of $O(M)$, if M is the number of elements stored in the vector `last_prices`.

Therefore, the overall complexity of the method `Stock_quote_archive::add_last_sale_price` is $O(M + N)$.

Exercise 2 Sales Data Archive

Given the implementation of the class `Sales_data` provided below, implement a class `Sales_archive`, which stores in a vector `Sales_data` items. Provide the relevant constructors, getters and setters. In particular, implement a method

```
Sales_archive reduce_by_book (void) const;
```

which returns a new `Sales_archive` object where all sales of the same book (identified by the ISBN) are aggregated in a single entry.

```
***** file Sales_data.h *****
```

```
1 #ifndef SALES_DATA_H
2 #define SALES_DATA_H
3
4 #include <iostream>
5 #include <string>
6
7 class Sales_data {
8
9 public:
10    // constructors:
11    Sales_data (void) = default;
12    Sales_data (const std::string &bn, unsigned u, double r):
```

```

13     bookNo(bn), units_sold(u), revenue(r) {}
14 // getters:
15 std::string isbn (void) const {return bookNo;}
16 unsigned get_units_sold (void) const {return units_sold;}
17 double get_revenue (void) const {return revenue;}
18 // combine
19 Sales_data& combine (const Sales_data &z);
20 // average price
21 double avg_price (void) const;
22 // print
23 void print (void) const;
24
25 private:
26     std::string bookNo = "";
27     unsigned units_sold = 0;
28     double revenue = 0.0;
29
30 };
31
32 #endif /* SALES_DATA_H */
```

***** file Sales_data.cpp *****

```

1 #include "Sales_data.h"
2
3 using std::cout;
4 using std::cerr;
5 using std::endl;
6
7 Sales_data&
8 Sales_data::combine (const Sales_data &rhs)
9 {
10     units_sold += rhs.units_sold;
11     revenue += rhs.revenue;
12     return *this;
13 }
14
15 double
16 Sales_data::avg_price () const
17 {
18     if (units_sold==0)
19     {
20         cerr << "No books have been sold" << endl;
21         return 0.0;
22     }
23     return revenue/units_sold;
24 }
25
26 void
27 Sales_data::print (void) const
28 {
29     cout << "ISBN: " << bookNo <<
30         " units_sold: " << units_sold <<
```

```

31     " revenue: " << revenue << endl;
32 }
```

Exercise 2 - Solution

First of all, notice that, both in the file "Sales_data.cpp" and in the file "Sales_archive.cpp" we used **using** declarations such as **using std::cout**. This allows us to write in the code only **cout**, omitting the namespace name **std**. We could achieve the same result writing the more general instruction

```
using namespace std;
```

In these simple exercises, there are no reasons to write explicitly the **using** declarations instead of the using directive **using namespace std**, but this turns out to be a good practice in more complicated programs. Indeed, the purpose of namespaces is to avoid name clashes and thus it is better to maintain them unless we are sure that an omission does not generate ambiguous declarations.

In any case, **using** declarations or directives at namespace scope (i.e. out of any other block) should never appear in headers, since this defeats the purpose of namespaces and makes way for name clashes wherever those headers are included.

```
***** file Sales_archive.h *****
```

```

1 #ifndef SALES_ARCHIVE_H
2 #define SALES_ARCHIVE_H
3
4 #include <vector>
5
6 #include "Sales_data.h"
7
8 class Sales_archive {
9
10 public:
11     // constructors:
12     Sales_archive (void) = default;
13     Sales_archive (const Sales_data &s): archive({s}) {}
14     Sales_archive (const std::vector<Sales_data> &v): archive(v) {}
15
16     // public methods:
17     std::vector<Sales_data>::const_iterator
18     find_isbn (const std::string &) const;
19
20     void
21     add_elem (const Sales_data &sd) {archive.push_back(sd);}
22
23     Sales_archive
24     reduce_by_book (void) const;
25
26     void
27     print_archive (void) const;
28
29 private:
30     std::vector<Sales_data> archive;
31 }
```

```

32 // private methods:
33 std::vector<Sales_data>::iterator
34 find_isbn (const std::string &);
35
36 };
37
38 #endif /* SALES_ARCHIVE_H */

```

The class `Sales_archive` implements two different versions of the method named `find_isbn`. The first one (declared in lines 33 and 34 of the file "Sales_archive.h"), which is `private`, returns a non constant iterator and will be used within the implementation of the method `reduce_by_book` (for the reason why the method is `private` and not `const`, see exercise 1). The second version (lines 17 and 18), which, in turn, is a `public` method, returns a `const_iterator` and it is declared as `const`. It has been implemented because we wanted to give the user the possibility to search a `Sales_data` object in the archive passing its ISBN, but we did not want to allow him to use the returned iterator to modify the archive.

The class has three constructors: the default one (line 12), with no parameters, which creates an empty vector; one that receives a single `Sales_data` object (line 13) and one that receives a full vector of `Sales_data` (line 14).

The class implements a method called `add_elem` (lines 20 and 21) which receives as input a constant reference to a `Sales_data` object and adds that object to the vector `archive`. The argument is passed as a reference in order to avoid to make an useless copy and it is declared `const` because the method will not modify it.

***** file `Sales_archive.cpp` *****

```

1 #include "Sales_archive.h"
2
3 using std::vector;
4 using std::string;
5
6 // public methods:
7 vector<Sales_data>::const_iterator
8 Sales_archive::find_isbn (const string &s) const
9 {
10     vector<Sales_data>::const_iterator pos = archive.cbegin();
11     while (pos!=archive.cend() && pos->isbn()!=s)
12         pos++;
13     return pos;
14 }
15
16 Sales_archive
17 Sales_archive::reduce_by_book (void) const
18 {
19     vector<Sales_data>::const_iterator cit = archive.cbegin();
20     Sales_data sd = *cit;
21     Sales_archive new_sa(sd);
22     for (cit=archive.cbegin()+1; cit!=archive.cend(); cit++)
23     {
24         sd = *cit;
25         vector<Sales_data>::iterator pos = new_sa.find_isbn(sd.isbn());
26         if (pos==new_sa.archive.end())
27             new_sa.archive.push_back(sd);
28         else

```

```

29     pos->combine(sd);
30 }
31 return new_sa;
32 }

33

34 void
35 Sales_archive::print_archive (void) const
36 {
37     vector<Sales_data>::const_iterator cit;
38     for (cit=archive.cbegin(); cit!=archive.cend(); cit++)
39         cit->print();
40 }

41

42 // private methods:
43 vector<Sales_data>::iterator
44 Sales_archive::find_isbn (const string &s)
45 {
46     vector<Sales_data>::iterator pos = archive.begin();
47     while (pos!=archive.end() && pos->isbn()!=s)
48         pos++;
49     return pos;
50 }

```

The worst case complexity of the method `reduce_by_book` is $O(N^2)$, where N is the number of `Sales_data` objects stored in the archive. Indeed, within the for loop of lines 26 to 30, the method calls at most N times `find_isbn`, which has a worst case complexity of $O(N)$.

Note that the `reduce_by_book` method works properly under the assumption that the archive is not empty. How would you modify the code to relax this hypothesis?

Exercise 3 Roots of Polynomials

Given the C implementation of the Bisection, Newton and Robust methods to find the roots of a *function*, provide an implementation that relies on classes.

```

1 #include <iostream>
2 #include <cmath>
3 #include <cstdlib>
4 #include <limits>
5
6 /// Type for real numbers
7 typedef double real;
8
9 /// Type for convergence check
10 enum checkT {
11     INCREMENT, //! Check the difference between subsequent iterates
12     RESIDUAL, //! Check the residual
13     BOTH      //! Check both conditions
14 };
15
16 /// Function f(x)
17 real f (real x) {
18     return pow(x,2) - 0.5;
19 }

```

```

20
21 //! First derivative of function f(x)
22 real df (real x) {
23     return 2. * x;
24 }
25
26 bool converged (real increment, real residual,
27                 real tol, const checkT& check){
28     /*
29      Compares a parameter value against desired tolerance.
30      The parameter is chosen upon the value of check.
31     */
32     switch(check){
33     case INCREMENT:
34         return (increment < tol);
35     case RESIDUAL:
36         return (residual < tol);
37     case BOTH:
38         return ((increment < tol)&&(residual < tol));
39     default:
40         return false;
41     }
42 }
43
44 // BISECTION METHOD
45 real bisection (real a, real b,
46                  real tol, int maxit,
47                  const checkT& check, int& nit)
48 {
49     real u = f(a); // Evaluate f on boundary a
50     real l = b - a; // Interval length
51     real r;          // Residual
52     real c = a+l;   // Middle point
53
54     nit = 0;
55     r = f(c);
56
57     if (u*f(b)>=0.0) {
58         std::cout<<"Error Function has same sign at both endpoints"<<std::endl;
59         return - std::numeric_limits<double>::infinity();
60     }
61
62     while ( !(converged(fabs(l), fabs(r), tol, check))
63             && (nit <= maxit) ) {
64         /*
65          If f(c)f(a) < 0 then the new "right" boundary is c;
66          otherwise move the "left" boundary
67
68          The expression
69              test ? stat1 : stat2
70          means
71          if(test)

```

```

72         stat1
73     else
74         stat2
75     */
76     (u*r < 0.) ? (b = c) : (a = c, u = r);
77     l *= 0.5;
78     c = a + l;
79     r = f(c);
80     ++nit;
81 }
82 return c;
83 }

84

85

86 // NEWTON METHOD
87 real newton (real xp, real tol, int maxit,
88               const checkT& check, int & nit)
89 {
90     real v = f(xp);
91     real xnew;
92
93     nit = 0;
94     for(int k = 1; k <= maxit; ++k,++nit) {
95         double derv = df(xp);
96         if(!derv) {
97             std::cerr << "ERROR: Division by 0 occurred in Newton algorithm"
98             << std::endl;
99             exit(1);
100        }
101
102        xnew = xp - v / derv;
103        v = f(xnew);
104        if(converged(fabs(xnew - xp), fabs(v),tol,check)) break;
105        xp = xnew;
106    }
107    return xnew;
108 }

109

110

111

112 // ROBUST METHOD
113 real robust(real a, real b,
114             real tol, real cfratio,
115             int maxit, const checkT& check,
116             int& nit_coarse, int& nit_fine)
117 {
118     // Call bisection method (with a greater desired tolerance)
119     real tol_bis = cfratio * tol;
120     real x_bis = bisection(a, b, tol_bis, maxit, check, nit_coarse);
121
122     /*
123      Call a Newton algorithm which uses as an initial value

```

```

124     the solution given by bisection method
125 */
126 return newton(x_bis, tol, maxit, check, nit_fine);
127 }

128

129

130

131 // MAIN FUNCTION
132 int main() {

133

134     int nit_bis;
135     int nit_newt;
136     std::cout << std::numeric_limits<double>::infinity() << std::endl;
137     std::cout << "Bisection" << std::endl;
138     std::cout << bisection(0., 1., 1e-8, 100, INCREMENT, nit_bis);
139     std::cout << '\t' << nit_bis << std::endl;

140

141     std::cout << "Newton" << std::endl;
142     std::cout << newton(.1, 1e-8, 100, INCREMENT, nit_newt);
143     std::cout << '\t' << nit_newt << std::endl;

144

145     std::cout << "Robust" << std::endl;
146     std::cout << robust(0., 1., 1e-8, 1e4, 100, INCREMENT, nit_bis, nit_newt);
147     std::cout << '\t' << nit_bis << " " << nit_newt << std::endl;

148

149     return 0;
150 }
```

Exercise 3 - Solution

The header file "rootfinding.h" is used to define some useful structures that are needed in all the classes. In particular, it defines an alias to the type **double**, namely the type **real**. This can be useful because, if later on we want to modify the program using **floats** instead of **doubles**, we only need to change the definition of **real**, without modifying the rest of the code.

***** file rootfinding.h *****

```

1 #ifndef ROOTFINDING_H
2 #define ROOTFINDING_H
3
4 #include <iostream>
5 #include <cmath>
6 #include <limits>
7
8 //! Type for real numbers
9 typedef double real;
10
11 //! Type for convergence check
12 enum checkT {
13     INCREMENT, //! Check the difference between subsequent iterates
14     RESIDUAL, //! Check the residual
15     BOTH //! Check both conditions

```

```

16 };
17
18 #endif
```

The class **Function** implements a polynomial *function* of the type $a + bx + cx^2 + dx^3\dots$. In particular, it stores in a vector of **real** only the coefficients of the polynomial. The index of the coefficient in the vector gives the information about the exponent of the corresponding variable (namely, considering the example above, *a* has index 0 and it corresponds to the term x^0 , *b* has index 1 and it corresponds to x and so on). Consider the fact that the definition of the class **Function** (even if it is limited to polynomials) makes the code considerably more general with respect to the previous version. Indeed, there, every time we want to find the roots of a *function f*, we have to define manually two functions, one that represents *f* and the other that represents its first derivative. This implementation, in turn, gives us a general structure that allows us, first, to define all the *functions* that we want, by only passing to the constructor the vector of their coefficients and, second, to automatically compute their derivatives.

***** file **Function.h** *****

```

1 #ifndef FUNCTION_H_
2 #define FUNCTION_H_
3
4 #include <vector>
5 #include "rootfinding.h"
6
7 class Function {
8 private:
9     std::vector<real> coefficients;
10
11 public:
12     Function (const std::vector<real> &coeff): coefficients(coeff){};
13     real eval (real x) const;
14     Function derivative (void) const;
15     void print (void) const;
16 };
17
18 #endif /* FUNCTION_H_ */
```

The class implements a method **eval** (see the declaration in line 13 of the file "Function.h"), which returns the value of the *function* at the point passed as parameter. Moreover, it implements a method **derivative**, which returns, as a new **Function**, the first derivative of **this**.

Note that all the methods in this class are defined as **const** because they do not modify the members of the class itself.

***** file **Function.cpp** *****

```

1 #include "Function.h"
2
3 using std::vector;
4
5 real Function::eval(real x) const {
6
7     real val=0;
8     for (int i=0; i<coefficients.size();i++)
```

```

9     val+=coefficients[i]*pow(x,i);
10    return val;
11 }
12 }
13
14 Function Function::derivative(void) const {
15
16     vector<real> dcoefficients;
17     for (int i=0; i<coefficients.size()-1;i++)
18         dcoefficients.push_back((i+1)*coefficients[i+1]);
19     return Function(dcoefficients);
20 }
21 }
22
23 void Function::print(void) const {
24
25     for (int i=0; i<coefficients.size();i++)
26         std::cout<<coefficients[i]<< " ";
27     std::cout<<std::endl;
28 }
29 }
```

Notice that in the file "Function.cpp" the **using** declaration **using std::vector** is written in order to be able to use directly **vector** instead of **std::vector** (see Exercise 2 for additional discussion)³.

The class **Bisection** implements the bisection algorithm we had in the previous version. The constructor receives as parameters all the data we need for the computation, namely the *function*, the extrema of the interval in which we want to look for the roots, the tolerance, the information about the method we want to use in order to check convergence and the maximum number of iterations.

Note that, even if it is not mandatory, it is always a good practice, for performance considerations, to accept unknown size parameters as **const &**, if you do not need to modify them, or as **&**, when you do need. In particular, the type **Function** contains a vector, so every copy involves $O(N)$ operations, where N is the number of contained elements, whilst passing by reference is $O(1)$.

***** file **Bisection.h** *****

```

1 #ifndef BISECTION_H_
2 #define BISECTION_H_
3
4 #include "rootfinding.h"
5 #include "Function.h"
6
7 class Bisection{
8     private:
9         Function func;
10        real inf_limit;
```

³Here you can construct an example in which, in turn, the using directive **using namespace std** could create some issues: if you define your class as **function** instead of **Function** and, in any file where you declare a variable of type **function**, you write **using namespace std**, the compiler may raise an error. Indeed, the Standard Template Library defines a type named **std::function** and therefore the compiler may consider the name **function** to be ambiguous (this behaviour, however, is not straightforward because some compilers are more restrictive than others).

```

11     real sup_limit;
12     real tolerance;
13     checkT termination_criteria;
14     int max_iteration;
15
16     bool converged (real increment, real residual,
17                      real tol, const checkT& check ) const;
18 public:
19     Bisection (const Function &f, real inf_l, real sup_l, real tol,
20                const checkT& term_c, int max_i):
21         func(f), inf_limit(inf_l), sup_limit(sup_l), tolerance(tol),
22         termination_criteria(term_c), max_iteration(max_i){}
23     real find_root(int & nit) const;
24 };
25
26 #endif /* BISECTION_H_ */

```

The class defines a **private** method, namely `converged` (see lines 16 and 17 of the file "Bisection.h"), which returns `true` if the convergence is reached, and a **public** method, namely `find_root` (see line 22), which returns the root of the function computed via the bisection algorithm.

***** file Bisection.cpp *****

```

1 #include "Bisection.h"
2
3 #include <cstdlib>
4
5 bool Bisection::converged(real increment, real residual,
6                           real tol, const checkT& check ) const {
7
8     switch(check){
9         case INCREMENT:
10            return (increment < tol);
11        case RESIDUAL:
12            return (residual < tol);
13        case BOTH:
14            return ((increment < tol)&&(residual < tol));
15        default:
16            return false;
17    }
18
19 }
20
21 real Bisection::find_root(int & nit) const
22 {
23     real a=inf_limit;
24     real b=sup_limit;
25     real u = func.eval(a); // Evaluate f on boundary a
26     real l = b - a;       // Interval length
27     real r;               // Residual
28     real c = a+l;         // Middle point
29
30     nit = 0;

```

```

31     r = func.eval(c);
32
33     if (u*func.eval(b)>=0.0) {
34         std::cout<<"Error Function has same sign at both endpoints"<<std::endl;
35         return - std::numeric_limits<double>::infinity();
36     }
37
38     while ( !(converged(fabs(l), fabs(r), tolerance, termination_criteria))
39             && (nit <= max_iteration) ) {
40
41         (u*r < 0.) ? (b = c) : (a = c, u = r);
42         l *= 0.5;
43         c = a + l;
44         r = func.eval(c);
45         ++nit;
46     }
47     return c;
48 }
```

The implementation of both the methods is exactly the same we had in the previous version, a part from the fact that we rely on the methods of the class `Function`. For example, at line 25 of the file "Bisection.cpp" we call the method `eval` in order to evaluate the function at the given point.

Note that, since here all the information about the *function*, the extrema of the interval, etc are members of the class, we do not need to pass to the method `find_root` other variables than `nit`, which, exactly as in the previous version, will store the information about the number of iterations truly performed by the algorithm.

The class `Newton` implements the Newton algorithm. The constructor receives as parameters the *function*, an initial point from which we start the computation, a fixed tolerance, the information about the method we want to use in order to check convergence and the maximum number of iterations. Note that the derivative of the *function* is not a parameter of the constructor: the variable `der` is initialized through a call to the method `Function::derivative()` defined in the class `Function`.

The methods defined in the class are exactly the same we had in `Bisection` (the implementation of `Newton::find_root` is, of course, different).

***** file `Newton.h` *****

```

1 #ifndef NEWTON_H_
2 #define NEWTON_H_
3
4 #include <cstdlib>
5
6 #include "rootfinding.h"
7 #include "Function.h"
8
9 class Newton {
10 private:
11     Function func;
12     Function der;
13     real x_init;
14     real tolerance;
15     checkT termination_criteria;
16     int max_iteration;
```

```

17     bool converged (real increment, real residual,
18                     real tol, const checkT& check ) const;
19
20 public:
21     Newton(const Function &f, real xp, real tol, const checkT& term_c,
22            int max_i):
23         func(f), der(f.derivative()), x_init(xp), tolerance(tol),
24         termination_criteria(term_c), max_iteration(max_i){
25     }
26     real find_root(int & nit) const;
27 };
28
29 #endif /* NEWTON_H_ */

```

***** file Newton.cpp *****

```

1 #include "Newton.h"
2
3 bool Newton::converged(real increment, real residual,
4                         real tol, const checkT& check) const {
5
6     switch(check){
7     case INCREMENT:
8         return (increment < tol);
9     case RESIDUAL:
10        return (residual < tol);
11    case BOTH:
12        return ((increment < tol)&&(residual < tol));
13    default:
14        return false;
15    }
16
17 }
18
19 real Newton::find_root(int & nit) const
20 {
21     real xp = x_init;
22     real v = func.eval(xp);
23     real xnew = std::numeric_limits<double>::quiet_NaN();
24
25     nit = 0;
26     bool stop = false;
27     for(int k = 1; k <= max_iteration && ! stop; ++k,++nit) {
28         real derv = der.eval(xp);
29         if(! derv) {
30             std::cerr << "ERROR: Division by 0 occurred in Newton algorithm"
31             << std::endl;
32         }
33     else
34     {
35         xnew = xp - v / derv;
36         v = func.eval(xnew);
37         stop = converged(fabs(xnew - xp), fabs(v),tolerance,termination_criteria);

```

```

38     xp = xnew;
39 }
40 }
41 return xnew;
42 }
```

The class `Robust` implements the robust algorithm. The constructor receives as parameters the *function*, the extrema of the interval in which we want to look for the roots, a fixed tolerance (that is intended to be used by the Newton algorithm), the ratio between the tolerance used by Newton and the one used by the bisection algorithm, the information about the method we want to use in order to check convergence and the maximum number of iterations.

***** file `Robust.h` *****

```

1 #ifndef ROBUST_H_
2 #define ROBUST_H_
3
4 #include "Bisection.h"
5 #include "Newton.h"
6
7 class Robust {
8 private:
9     Function func;
10    real inf_limit;
11    real sup_limit;
12    real tolerance;
13    real cratio;
14    checkT termination_criteria;
15    int max_iteration;
16
17 public:
18     Robust(const Function &f, real inf_l, real sup_l, real tol, real cr,
19             const checkT& term_c, int max_i):
20         func(f), inf_limit(inf_l), sup_limit(sup_l), tolerance(tol),
21         cratio(cr), termination_criteria(term_c), max_iteration(max_i){}
22     real find_root(int & nit_bis, int & nit_newt) const;
23 };
24
25 #endif /* ROBUST_H_ */
```

***** file `Robust.cpp` *****

```

1 #include "Robust.h"
2
3 real Robust::find_root(int & nit_bis, int & nit_newt) const {
4     Bisection bs(func, inf_limit, sup_limit, cratio * tolerance,
5                  termination_criteria, max_iteration);
6
7     // Call bisection method (with a greater desired tolerance)
8     real x_bis = bs.find_root(nit_bis);
9
10    /*
11     * Call a Newton algorithm which uses as an initial value
```

```

12     the solution given by bisection method
13 */
14
15 Newton nw(func, x_bis, tolerance, termination_criteria, max_iteration);
16 return nw.find_root(nit_newt);
17
18 }

```

The method `find_root` is defined on top of the corresponding methods `Bisection::find_root` and `Newton::find_root`. First of all, we instantiate an object of type `Bisection`, passing as tolerance the product (`cratio * tolerance`), and we compute `x_bis` using the method `Bisection::find_root`. Then, we instantiate an object of type `Newton`, using this `x_bis` as initial point, and we compute the final value of the root through `Newton::find_root`.

***** file Roots.cpp *****

```

1 #include "rootfinding.h"
2 #include "Function.h"
3 #include "Bisection.h"
4 #include "Newton.h"
5 #include "Robust.h"
6
7 using std::vector;
8 using std::cout;
9 using std::endl;
10
11 int main() {
12
13     int nit_bis;
14     int nit_newt;
15     /*
16         vector<double> coeffs;
17
18         coeffs.push_back(-0.5);
19         coeffs.push_back(0);
20         coeffs.push_back(1.0);
21     */
22
23     Function f({-0.5, 0., 1.}); //  $x^2 - 1/2!$ 
24
25     Bisection bis(f, 0., 1., 1e-8, INCREMENT, 100);
26     cout << "Bisection" << endl;
27     cout << bis.find_root(nit_bis);
28
29     cout << '\t' << nit_bis << endl;
30
31     Newton nw(f, .1, 1e-8, INCREMENT, 100);
32
33     cout << "Newton" << endl;
34     cout << nw.find_root(nit_newt);
35     cout << '\t' << nit_newt << endl;
36
37     Robust rob(f, 0., 1., 1e-8, 1e4, INCREMENT, 100);
38     cout << "Robust" << endl;

```

```

39     cout << rob.find_root(nit_bis,nit_newt);
40     cout << '\t' << nit_bis
41         << " " << nit_newt << endl;
42
43     return 0;
44 }
```

Exercise 4 Gradient Descent

Provide the implementation of the gradient descent for polynomial functions $f : \mathbb{R} \rightarrow \mathbb{R}$, relying on the implementation of the `Function` class provided in exercise 3.

The header for the class `FunctionMin` should be the following:

```

1  class FunctionMin
2  {
3      Function f;
4      double inf_limit;
5      double sup_limit;
6      double tolerance;
7      double step;
8      unsigned int max_iterations;
9
10     //gradient descent with x_init initial point
11     double solve (double x_init) const;
12
13 public:
14     FunctionMin (Function func, double a, double b, double tol,
15                   double s, unsigned int max_it)
16         : f (func), inf_limit (a), sup_limit (b), tolerance (tol),
17           step (s), max_iterations (max_it) {};
18
19     // gradient descent
20     double solve (void) const;
21     // gradient descent with multi-start
22     double solve_multistart (unsigned int n_trials) const;
23     // gradient descent with multi-start and domain decomposition
24     double solve_domain_decomposition (unsigned int n_intervals,
25                                         unsigned int n_trials) const;
26 }
```

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 *****

```

1 #include <iostream>
2
```

```

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 interval 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 substituted 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:

$$\frac{\text{sup_limit} - \text{inf_limit}}{n_intervals}$$

and then, at every iteration of the for loop, we instantiate a new object of type `FunctionMin`, defining as new interval $[\text{internal_inf_limit}, \text{internal_inf_limit} + \text{internal_step}]$ (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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

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 $\|\cdot\|_2$ 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 }
```

```

10 protected:
11   coords_type x;
12
13 public:
14   explicit Point (const coords_type & coords): x (coords) {};
15
16   //compute distance to Point p
17   double distance (const Point & p) const;
18
19   void print (void) const;
20
21   std::size_t get_n_dimensions (void) const;
22   double get_coord (std::size_t i) const;
23   void set_coord (std::size_t i, double val);
24   coords_type get_coords (void) const;
25
26   double euclidean_norm (void) const;
27   double infinity_norm (void) const;
28 };
29
30 #endif /* POINT_H_ */
```

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

```

1 #include <cmath>
2 #include <iostream>
3
4 #include "Point.h"
5
6 double Point::distance (const Point & p) const
7 {
8   double dist = 0.0;
9
10  for (std::size_t i = 0; i < x.size (); ++i)
11  {
12    const double delta = x[i] - p.x[i];
13    dist += delta * delta;
14  }
15
16  return sqrt (dist);
17 }
18
19 void Point::print (void) const
20 {
21   for (auto it = x.begin (); it != x.end (); ++it)
22   {
23     std::cout << *it;
24     std::cout << " ";
25   }
26
27   std::cout << std::endl;
28 }
```

```

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>
```

```

6 #include "Point.h"
7
8 class Monomial
9 {
10     double coeff;
11     std::vector<double> powers;
12
13 public:
14     Monomial (double c, const std::vector<double> & pows)
15         : coeff (c), powers (pows) {};
16     double eval (const Point & P) const;
17 };
18
19 #endif /* MONOMIAL_H_ */

```

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

```

1 #include <cmath>
2
3 #include "Monomial.h"
4
5 double Monomial:: eval (Point const & P) const
6 {
7     double value = 1;
8
9     for (std::size_t dim = 0; dim < powers.size (); ++dim)
10        value *= pow (P.get_coord (dim), powers[dim]);
11
12    return coeff * value;
13 }

```

A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ can be seen as the sum of different monomials. Therefore, we store in the class `FunctionRn` a `std::vector` of `Monomials`.

The class is implemented 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 *****

```

1 #ifndef FUNCTIONRN_H_
2 #define FUNCTIONRN_H_
3
4 #include <vector>
5
6 #include "Monomial.h"
7 #include "Point.h"
8
9 class FunctionRn

```

```

10 {
11     std::vector<Monomial> monoms;
12
13     static constexpr double h = 0.00001;
14
15 public:
16     double eval (const Point & P) const;
17
18     void addMonomial (const Monomial & m);
19
20     //evaluate derivative wrt dim j in P
21     double eval_deriv (std::size_t j,
22                         const Point & P) const;
23 };
24
25 #endif /* FUNCTIONRN_H_ */

```

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

```

1 #include "FunctionRn.h"
2
3 double FunctionRn::eval (const Point & P) const
4 {
5     double value = 0;
6
7     for (std::size_t i = 0; i < monoms.size (); ++i)
8         value += monoms[i].eval (P);
9
10    return value;
11 }
12
13 void FunctionRn::addMonomial (const Monomial & m)
14 {
15     monoms.push_back (m);
16 }
17
18 //evaluate derivative wrt dim j in P
19 double FunctionRn::eval_deriv (std::size_t j,
20                               const Point & P) const
21 {
22     Point P1 (P.get_coords ());
23     Point P2 (P.get_coords ());
24     P1.set_coord (j, P.get_coord (j) + h);
25     P2.set_coord (j, P.get_coord (j) - h);
26     return (eval (P1) - eval (P2)) / (2 * h);
27 }

```

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 P_0 (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 P_0 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 P_0 (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]` exceeds 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

⁴The 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 fall on Monday (namely the 4th of April will be Monday, the 6th of June will be Monday, etc.).

⁴Implementation by Samuele Vianello and Andrea Farahat

3. finally, we have to find the Doomsday closest to d and to compute the week day of d starting from that Doomsday.

Both in the instantiation and in the computation of a new date, the necessity to check the validity of a date entails, for example, that the class must be able to recognize leap years. In particular, an year is leap if:

- it is divisible by 4 but not by 100 OR
- it is divisible by 400

The class `Date` stores three integers that represents the day, the month and the year, and a `bool` which is `true` if the year is leap.

It has one constructor, which receives as parameters three values, for the day, the month and the year respectively (see the file "Date.cpp", lines 5 to 28, for the relative implementation). The class defines different `public` methods: first of all, the method `add_days` (see the file "Date.cpp", lines 32 to 79, for the implementation) allows to update the date adding or subtracting a certain number of days.

`leap_year`, `next_Sunday` and `next_Weekday` are `static` methods, which means that they are independent of any object of the class `Date` and they can be called directly without passing through an object of the class (see for example line 15 of the file "main_user.cpp": the method `next_Sunday` is called writing `Date::next_Sunday`, while, for instance, the method `show_date`, at line 17, must be called writing `d1.show_date()`).

Furthermore, the class implements four `private` methods. `get_max` (see lines 142 to 156 of "Date.cpp") returns the maximum admissible value for a day, according to the month and to the value of the `bool` `leap`. The other `private` methods are useful to compute `next_Sunday` and `next_Weekday` and they are described in "Date.cpp".

***** file Date.hpp *****

```

1 #ifndef _H_DATE_H_
2 #define _H_DATE_H_
3
4 #include <iostream>
5 #include <string>
6
7 class Date {
8
9 private:
10
11     int day, month, year;
12     bool leap;
13
14     int get_max (void) const;
15     int dooms_month (void) const;
16     int Doomsday (void) const;
17
18     static std::string day_name (int d);
19
20 public:
21
22     Date (int d, int m, int y);
23
24     void add_days (int n);

```

```

25
26     int get_day (void) const {return day;}
27     int get_month (void) const {return month;}
28     int get_year (void) const {return year;}
29
30     void show_date (void) const;
31
32     static bool leap_year (int y);
33     static Date next_Sunday (const Date &d);
34     static Date next_Weekday (const Date &d);
35
36 };
37
38 #endif /* _H_DATE_H_ */

```

The constructor, see lines 5 to 28 of the file "Date.cpp", allows to define a new date passing as parameters the values for day, month and year. It checks, first of all, if the year is leap, then it checks if the value inserted for the month is valid and, finally, it computes the maximum admissible value for the day and it checks if the number inserted is valid. In particular, if the values inserted are not admissible, it prints an error message and sets the corresponding variables to 1.

In order to implement the methods `Date::next_Sunday` and `Date::next_Weekday`, some **private** methods are used.

The method

```
std::string Date::day_name (int d);
```

(see lines 218 to 237), which is a **static** method, returns the name of the day passed as parameter.

The method

```
int Date::dooms_month (void) const;
```

(see lines 158 to 185) returns the Doomsday of the month, which will be the closest Doomsday to the date passed as parameter to `Date::next_Sunday` and `Date::next_Weekday`.

The method

```
int Date::Doomsday (void) const;
```

(see lines 187 to 216) implements the Doomsday algorithm as it is described above. In particular, the *anchor day cbd* (see line 194) is computed through the following formula:

$$\left[\left(5c + \lfloor \frac{c-1}{4} \rfloor \right) \%7 + 4 \right] \%7 = cbd$$

where c is the century to whom the date belongs.

Given this number, the Doomsday of the year (see line 201) is computed, starting from the last two digits of the year, namely ldy , using

$$\left[\lfloor \frac{ldy}{12} \rfloor + ldy \% 12 + \lfloor \frac{ldy \% 12}{4} \rfloor \right] \%7 + cbd$$

In line 205, we store in `ddm` the Doomsday of the month, i.e. the closest Doomsday to the given date. This value is used in lines 208 to 213 in order to compute the week day of the given date.

The method `Date::next_Sunday` (see lines 100 to 112) initializes a new date with the values of the one passed as parameter (note that, since we are inside the scope of the class, the **private**

members day, month and year can be accessed directly), uses the function Doomsday to compute the week day related to this date and then it adds 7-my_day to it in order to compute the next Sunday.

Finally, the method Date::next_Weekday (see lines 114 to 138) follows the same strategy in order to compute the next week day related to the date passed as parameter.

```
***** file Date.cpp *****
```

```
1 #include "Date.hpp"
2
3 // constructor
4
5 Date::Date (int d, int m, int y)
6 {
7     leap = leap_year(y);
8     year = y;
9     if (m>12 || m<1)
10    {
11        std::cerr << "Month value m = "<< m <<" invalid. Value set to default m = 1."
12        << std::endl;
13        month=1;
14    }
15 else
16     month=m;
17
18 int maxd = get_max();
19
20 if (d<1 || d>maxd)
21    {
22        std::cerr << "Day value d = "<< d <<" invalid. Value set to default d = 1."
23        << std::endl;
24        day=1;
25    }
26 else
27     day=d;
28 }
29
30 // public methods
31
32 void
33 Date::add_days(int n)
34 {
35     int max;
36     while (n > 0)
37    {
38        max = get_max();
39        if ( max-day >= n)
40            {
41                day += n;
42                n = 0;
43            }
44        else
45            {
```

```

46     n -= (max-day);
47     day = 0;
48
49     if (month==12)
50     {
51         month = 1;
52         year++;
53     }
54     else month++;
55 }
56 }
57
58 while (n<0)
59 {
60     if (day > -n)
61     {
62         day += n;
63         n = 0;
64     }
65     else
66     {
67         if (month==1)
68         {
69             month = 12;
70             year--;
71         }
72         else
73             month--;
74         max = get_max();
75         n += (day-1);
76         day = max+1;
77     }
78 }
79 }
80
81 void
82 Date::show_date (void) const
83 {
84     std::cout << "Date: "<<day <<"/"<<month <<"/"<<year << '\n'<< std::endl;
85 }
86
87 bool
88 Date::leap_year (int y)
89 {
90     bool l = true;
91     if (y%4 != 0)
92         l = false;
93     else if (y%100 != 0)
94         l = true;
95     else if (y%400 != 0)
96         l = false;
97     return l;

```

```

98 }
99
100 Date
101 Date::next_Sunday (const Date &d)
102 {
103     Date data(d.day, d.month, d.year);
104
105     int my_day = data.Doomsday();
106
107     std::cout << "The Weekday related to the date is " << day_name(my_day) << std::
108         endl;
109
110     data.add_days(7-my_day);
111
112     return data;
113 }
114
115 Date
116 Date::next_Weekday (const Date &d)
117 {
118     Date data(d.day, d.month, d.year);
119
120     int my_day = data.Doomsday();
121
122     if (my_day == 5)
123         { //if it is Friday, the next_weekday is Monday
124             data.add_days(3);
125             std::cout << "The next Weekday is " << day_name(1) << std::endl;
126         }
127     else if (my_day == 6)
128         { //if it is Saturday, the next_weekday is Monday
129             data.add_days(2);
130             std::cout << "The next Weekday is " << day_name(1) << std::endl;
131         }
132     else
133         {
134             data.add_days(1); //for every other day, the next_weekday is the following one
135             std::cout << "The next Weekday is " << day_name(my_day+1) << std::endl;
136         }
137
138     return data;
139 }
140
141 // private methods
142
143 int
144 Date::get_max (void) const
145 {
146     int max = 31;
147     if (month==4 || month==6 || month==9 ||month==11)
148         max = 30;
149     else if (month==2)

```

```

149     {
150         if (leap)
151             max = 29;
152         else
153             max = 28;
154     }
155     return max;
156 }
157
158 int
159 Date::dooms_month (void) const
160 {
161     int doom = 12;
162     if (month == 1)
163         doom = 24 + leap;
164     if (month == 2)
165         doom = 28 + leap;
166     if (month == 3)
167         doom = 7;
168     if (month == 4)
169         doom = 4;
170     if (month == 5)
171         doom = 9;
172     if (month == 6)
173         doom = 6;
174     if (month == 7)
175         doom = 11;
176     if (month == 8)
177         doom = 8;
178     if (month == 9)
179         doom = 5;
180     if (month == 10)
181         doom = 10;
182     if (month == 11)
183         doom = 7;
184     return doom;
185 }

186
187 int
188 Date::Doomsday (void) const
189 {
190     // Step 1)
191     int cbd, century;
192     century = (year / 100) + 1 ;
193     //cbd = century base day
194     cbd = ( ( 5*century + ( (century-1) / 4 ) )% 7 +4 )%7;

195
196     // Step 2)
197     int dooms, ldy;
198     //ldy = last two digit of year
199     ldy = year % 100;
200     //dooms = Doomsday of the year

```

```

201 dooms = (ldy/12 + ldy%12 + (ldy%12)/4)%7 + cbd;
202
203 // Step 3)
204 //ddm = Doomsday of the month
205 int ddm = dooms_month();
206
207 // Step 3.1)
208 int dd = day - ddm;
209 int dooms_day = dd + dooms;
210 int my_day = dooms_day % 7;
211
212 if (my_day < 0)
213     my_day += 7;
214
215 return my_day;
216 }
217
218 std::string
219 Date::day_name (int d)
220 {
221     std::string name = "This number does not represent a day";
222     if (d == 0)
223         name = "Sunday";
224     if (d == 1)
225         name = "Monday";
226     if (d == 2)
227         name = "Tuesday";
228     if (d == 3)
229         name = "Wednesday";
230     if (d == 4)
231         name = "Thursday";
232     if (d == 5)
233         name = "Friday";
234     if (d == 6)
235         name = "Saturday";
236     return name;
237 }
```

***** file main_user.cpp *****

```

1 #include "Date.hpp"
2
3 int main()
4 {
5     int d, m, y;
6     std::cout << "Insert day, month and year: " << std::endl;
7     std::cin >> d >> m >> y;
8     Date mydate(d, m, y);
9     mydate.show_date();
10
11     Date d1 = Date::next_Sunday(mydate);
12     std::cout << "Next Sunday is: " << std::endl;
13     d1.show_date();
```

```

14     Date d2 = Date::next_Weekday(mydate);
15     d2.show_date();
16
17     return 0;
18 }
```

2.4 Frequently Asked Questions

1. If I declare a method of my class as `const`, which variables am I allowed to modify within the method itself?

ANSWER: When you declare a member function to be `const`, you are guaranteeing that the instance against which you call the method will not be modified as a direct or indirect effect of that function call. This means that you cannot modify non-static members, call other non-static member functions that are not qualified as `const`, or return a plain reference to a non-static data member. The first two constraints are straightforward: since you promised not to modify the instance, you cannot perform such changes yourself or delegate to other member functions that do not provide the same guarantee. The last restriction is trickier to grasp, but fundamental: your member function is not directly changing anything in the instance, yet it is leaving this possibility open to whoever called it via the returned reference.

As an example, consider the class

```

1 #include <iostream>
2
3 class Myclass {
4
5 private:
6     int x = 3;
7     int y = 5;
8
9 public:
10    Myclass (void) = default;
11
12    void first_method (void) const;
13    void second_method (int n);
14
15    const int& third_method (void) const;
16    int fourth_method (void) const;
17
18 };
19
20 void Myclass::first_method (void) const
21 {
22     std::cout << x << " " << y << std::endl;
23 }
24
25 void Myclass::second_method (int n)
26 {
27     x += n;
28 }
29
30 const int& Myclass::third_method (void) const
```

```

31 {
32     return y;
33 }
34
35 int Myclass::fourth_method (void) const
36 {
37     return x;
38 }
```

As it is explained above, `first_method`, which is `const`, cannot directly modify the values of the members `x` and `y` as it is done in `second_method`. Moreover, it cannot call `second_method` (for instance, before printing the values) because it is non `const` and thus it may modify the values of `x` and `y`, which is not allowed by the `const` qualifier of `first_method`.

For what concerns `third_method`, the issues stay in the return type: if we want to return a reference, this must be a reference to a `const`. Indeed, a reference to a non constant variable can be used to change its value and this is in contrast with the fact that `third_method` is not allowed to modify, directly or indirectly, the value of a member of `Myclass` (another possibility would be to return an `int` by value, as it is done by `fourth_method`, because in this case a copy is performed and thus the changing of the returned `int` do not affect the value stored in the object of the class).

2. If I need to define two classes that depends one from the other, for example

```

1 class apple
2 {
3     void hit_on_head (Newton &);
4 };
5
6 class Newton
7 {
8     void bite (apple &);
9 };
```

I cannot compile the code because I get the error:

```
error: unknown type name 'Newton'
void hit_on_head (Newton &);
```

How can I fix this?

ANSWER: When two classes refer to each other in their definitions, there is a circular dependency. The compilation of this piece of code fails because you try to use the parameter `Newton` in the declaration of `apple::hit_on_head` before having defined it. Of course, the same would happen if you try to switch the definitions of the classes, because, similarly, you would try to use the type `apple` in the method `Newton::bite` before having defined it.

Forward declarations come in handy when, as it happens in this case, you do not actually need a full-blown type, but you only want the compiler to know you are talking about a type. In order to get this, in the example above, you can forward declare `Newton`, then you use the identifier to declare `hit_on_head` and, consequently, define `apple`; in the end, you provide also the definition of `Newton` and everything works fine. To be more precise, you have to write:

```

1 class Newton;
2
```

```
3 class apple
4 {
5     void hit_on_head (Newton &z);
6 };
7
8 class Newton
9 {
10    void bite (apple &z);
11};
```

In general, we can say that a forward declaration is enough when the compiler does not need to know the details of a type. Some common cases are pointers and references to forward declared classes, usage in parameter lists or as return types, and the like. Situations in which a type must be complete are definitions of variables or data members, inheritance, access to class members, or pointer arithmetics.

Chapter 3

Inheritance and Polymorphism

3.1 Exercises

Exercise 1 Employees

A company with three categories of employees (manager, developer and secretary) needs a program to manage the procedure of salary payment.

Each employee has the following information: `name`, `surname`, `employee_ID` and `pay_rate` (which is the amount of salary any kind of employee receives per hour). Managers and developers, in addition, have a workshop attendance rate (`wsh_rate`). Managers have also a mission rate (`m_rate`). Total work hours (`work_hours`), mission hours (`m_hours`) and workshop hours (`wsh_hours`) are set from employees' worksheets through a suitable method. Every employee is either manager, developer or secretary. Each of these positions provides its specific function to calculate monthly salary:

- Managers:
$$((work_hours - m_hours - wsh_hours) * pay_rate) + (m_hours * m_rate) + (wsh_hours * wsh_rate)$$
- Developers:
$$((work_hours - wsh_hours) * pay_rate) + (wsh_hours * wsh_rate)$$
- Secretaries:
$$(work_hours * pay_rate)$$

Provide the class definitions for the proposed problem.

Hint: consider to define class `Employee` as an `abstract` class.

Exercise 1 - Solution

First of all, the class `Employee` is designed as an abstract class. Indeed, it will be the base class for all the categories of employees and each of them has to specialize in a different way the method `salary_cal` that it needs to compute the salary.

The class stores the information common to all the categories of employees (see lines 11 to 15 of the file "Employee.h") and it implements a method to set the number of work hours (line 24) and a method to print the information about an employee.

Since, here and in the following, the methods are implemented in-class, there is no source file and only the header file will be reported.

***** file Employee.h *****

```

1 #ifndef EMPLOYEE_H
2 #define EMPLOYEE_H
3
4 #include <string>
5 #include <iostream>
6
7 class Employee {
8
9 protected:
10
11     std::string name;
12     std::string surname;
13     unsigned id;
14     const double pay_rate = 7.5;
15     unsigned work_hours;
16
17 public:
18
19     Employee (const std::string& n, const std::string& sn, unsigned id):
20         name(n), surname(sn), id(id) {}
21
22     virtual double salary_cal(void) const = 0;
23
24     void set_work_hours (unsigned n) {work_hours = n;}
25
26     void print (void) const
27     {
28         std::cout << name << " " << surname << " " << id << std::endl;
29     }
30
31 };
32
33 #endif /* EMPLOYEE_H */

```

The class `Secretary` inherits from `Employee` and it overrides the method `salary_cal` according to the formula reported in the text.

***** file `Secretary.h` *****

```

1 #ifndef SECRETARY_H
2 #define SECRETARY_H
3
4 #include "Employee.h"
5
6 class Secretary: public Employee {
7
8 public:
9
10    Secretary (const std::string& n, const std::string& sn, unsigned id):
11        Employee(n, sn, id) {}
12
13    double salary_cal (void) const override {return (work_hours * pay_rate);}
14
15 };

```

```

16
17 #endif /* SECRETARY_H */

```

The class `Developer` inherits from `Employee`. It stores two additional data fields about the workshop hours and the relative pay rate and it overrides the method `salary_cal` according to the formula reported in the text.

***** file `Developer.h` *****

```

1 #ifndef DEVELOPER_H
2 #define DEVELOPER_H
3
4 #include "Employee.h"
5
6 class Developer: public Employee {
7
8 protected:
9
10    const double wsh_rate = 8.0;
11    unsigned wsh_hours = 0;
12
13 public:
14
15    Developer (const std::string& n, const std::string& sn, unsigned id):
16        Employee(n, sn, id) {}
17
18    double salary_cal (void) const override
19    {
20        return (work_hours - wsh_hours) * pay_rate + wsh_hours * wsh_rate;
21    }
22
23    void set_wsh_hours (unsigned n) {wsh_hours = n;}
24
25};
26
27#endif /* DEVELOPER_H */

```

The class `Manager` inherits from `Developer` and it extends it adding information about the mission hours and the relative pay rate. Moreover, it overrides the method `salary_cal` according to the formula reported in the text.

***** file `Manager.h` *****

```

1 #ifndef MANAGER_H
2 #define MANAGER_H
3
4 #include "Developer.h"
5
6 class Manager: public Developer {
7
8 protected:
9
10    const double wsh_rate = 8.5;
11    const double m_rate = 9.5;
12    unsigned m_hours = 0;

```

```

13
14 public:
15
16     Manager (const std::string& n, const std::string& sn, unsigned id):
17         Developer(n, sn, id) {}
18
19     double salary_cal (void) const override
20     {
21         return (work_hours - wsh_hours - m_hours) * pay_rate +
22                 wsh_hours * wsh_rate +
23                 m_hours * m_rate;
24     }
25
26     void set_m_hours (unsigned n) {m_hours = n;}
27
28 };
29
30 #endif /* MANAGER_H */

```

Exercise 2 Roots of Polynomials

Given the implementation of the Bisection, Newton and Robust methods provided in Exercise 3 of Chapter 2 (Section 2.2) to find the roots of a function, provide an implementation that relies on inheritance.

Exercise 2 - Solution

The structure of the program is the same we had in the version of Exercise 3 (Section 2.2 of Chapter 2). In particular, the file `rootfinding.h` and the class `Function` are not modified (and thus they are not reported here).

With respect to the previous version, we create a new class, namely `RootFinder`, which will be the base class for all the methods we want to implement. In particular, this class will implement the method `converged`, which is the same for Bisection, Newton and Robust algorithm (see file "RootFinder.cpp" for the implementation).

`RootFinder` is an abstract class: the method `find_root` (see the declaration in line 20 of the file "RootFinder.h") is pure `virtual` because it must be specialized according to the different algorithms.

***** file `RootFinder.h` *****

```

1 #ifndef ROOTFINDER_H_
2 #define ROOTFINDER_H_
3
4 #include "rootfinding.h"
5 #include "Function.h"
6
7 class RootFinder {
8     protected:
9         Function func;
10        real tolerance;
11        checkT termination_criteria;
12        int max_iteration;

```

```

13     bool converged (real increment, real residual,
14                 real tol, const checkT& check ) const;
15
16 public:
17     RootFinder(const Function & f, real tol, const checkT& term_c, int max_i):
18         func(f), tolerance(tol), termination_criteria(term_c), max_iteration(max_i){}
19
20     virtual real find_root(int & nit) const = 0;
21 };
22
23 #endif /* ROOTFINDER_H_ */

```

The implementation of the method `converged` is the same we had in the previous version, replicated in all the classes `Bisection`, `Newton` and `Robust` (see Exercise 3, in Section 2.2 of Chapter 2).

***** file `RootFinder.cpp` *****

```

1 #include "RootFinder.h"
2
3 bool RootFinder::converged (real increment, real residual,
4                             real tol, const checkT& check ) const {
5     /*
6      * Compares a parameter value against desired tolerance.
7      * The parameter is chosen upon the value of check.
8      */
9     switch(check){
10     case INCREMENT:
11         return (increment < tol);
12     case RESIDUAL:
13         return (residual < tol);
14     case BOTH:
15         return ((increment < tol)&&(residual < tol));
16     default:
17         return false;
18     }
19 }

```

The class `Bisection` inherits from `RootFinder`. In particular, it overrides the method `find_root` (see the declaration in line 18 of the file "Bisection.h") in order to make it consistent with the Bisection algorithm. The implementation is exactly the same we had in the version of Exercise 3, in Section 2.2 of Chapter 2, thus it is not reported here.

***** file `Bisection.h` *****

```

1 #ifndef BISECTION_H_
2 #define BISECTION_H_
3
4 #include "rootfinding.h"
5 #include "Function.h"
6 #include "RootFinder.h"
7
8 class Bisection: public RootFinder{
9     protected:
10         real inf_limit;

```

```

11     real sup_limit;
12
13 public:
14     Bisection(const Function & f, real inf_1, real sup_1, real tol,
15             const checkT& term_c, int max_i):
16         RootFinder (f,tol,term_c,max_i), inf_limit(inf_1), sup_limit(sup_1){}
17
18     real find_root(int & nit) const override;
19 }
20
21 #endif /* BISECTION_H_ */
```

As the previous one, the class `Newton` inherits from `RootFinder`. In particular, it overrides the method `find_root` (see the declaration in line 19 of the file "Newton.h") in order to make it consistent with the Newton algorithm. The implementation is exactly the same we had in the version of Exercise 3, in Section 2.2 of Chapter 2, thus it is not reported here.

***** file `Newton.h` *****

```

1 #ifndef NEWTON_H_
2 #define NEWTON_H_
3
4 #include <cstdlib>
5 #include <cmath>
6 #include "rootfinding.h"
7 #include "Function.h"
8 #include "RootFinder.h"
9
10 class Newton: public RootFinder {
11     private:
12         Function der;
13         real x_init;
14
15     public:
16         Newton(const Function & f, real xp, real tol, const checkT& term_c, int max_i):
17             RootFinder (f,tol,term_c,max_i), der(f.derivative()), x_init(xp){}
18
19         real find_root(int & nit) const override;
20     };
21
22 #endif /* NEWTON_H_ */
```

Finally, also the class `Robust` inherits from `RootFinder`. In this case, however, the structure is more complicated: the class needs a method `find_root` with two parameters, namely the number of iterations of the Bisection and of the Newton algorithms it has to run. Since it is not possible to modify the number of parameters in overriding a method, we overload it, i.e. we define a new version of `find_root`, which can be declared `const` because it does not modify the object, which is implemented as in the version of Exercise 3 (Section 2.2 of Chapter 2). In addition, the class overrides the method `find_root` (with one single parameter as in the base version), using it to call the other one (see the file "Robust.cpp", lines 19 to 25, for the implementation).

***** file `Robust.h` *****

```
1 #ifndef ROBUST_H_
```

```

2 #define ROBUST_H_
3
4 #include "Newton.h"
5 #include "Bisection.h"
6
7 class Robust: public Bisection {
8
9 private:
10    real cratio;
11
12 public:
13    Robust(const Function & f, real inf_1, real sup_1, real tol, real cr,
14           const checkT& term_c, int max_i):
15        Bisection (f,inf_1, sup_1, tol,term_c,max_i),cratio(cr){}
16
17    real find_root(int & nit_bis, int & nit_newt) const;
18    real find_root(int & nit) const override;
19};
20
21 #endif /* ROBUST_H_ */

```

***** file Robust.cpp *****

```

1 #include "Robust.h"
2
3 real Robust::find_root(int & nit_bis, int & nit_newt) const
4 {
5     Bisection bs(func,inf_limit,sup_limit,cratio * tolerance, termination_criteria,
6                  max_iteration);
7
8     // Call bisection method (with a greater desired tolerance)
9     real x_bis = bs.find_root(nit_bis);
10
11    /*
12     * Call a Newton algorithm which uses as an initial value
13     * the solution given by bisection method
14     */
15
16     Newton nw(func, x_bis, tolerance, termination_criteria, max_iteration);
17     return nw.find_root(nit_newt);
18 }
19
20 real Robust::find_root(int & nit) const
21 {
22     int nit_bis,nit_newt;
23     const real ret_value = find_root(nit_bis, nit_newt);
24     nit= nit_bis + nit_newt;
25     return ret_value;
26 }

```

In the `main` function (see the file "Roots2.cpp"), we can see the usage of the two different methods `find_root` we defined in class `Robust`. In particular, in line 45 we use the version with two parameters, while in line 52 we call the overridden version with only one parameter.

***** file Roots2.cpp *****

```
1 #include <iostream>
2 #include <cmath>
3 #include <cstdlib>
4 #include <vector>
5
6 #include "rootfinding.h"
7 #include "Function.h"
8 #include "Bisection.h"
9 #include "Newton.h"
10 #include "Robust.h"
11
12 using std::cout; using std::endl;
13 using std::vector;
14
15 int main() {
16
17     int nit_bis;
18     int nit_newt;
19     vector<double> coeffs;
20
21
22     Function f({-0.5, 0., 1.});
23
24
25     Bisection bis(f,0., 1., 1e-8, INCREMENT, 100);
26     cout << "Bisection" << endl;
27     cout << bis.find_root(nit_bis);
28
29     cout << '\t' << nit_bis << endl;
30
31     Newton nw(f,.1,1e-8,INCREMENT,100);
32
33     cout << "Newton" << endl;
34     cout << nw.find_root(nit_newt);
35     cout << '\t' << nit_newt << endl;
36
37     Robust rob(f,0., 1., 1e-8, 1e4,INCREMENT, 100);
38     cout << "Robust" << endl;
39     cout << rob.find_root(nit_bis,nit_newt);
40     cout << '\t' << nit_bis
41         << " " << nit_newt << endl;
42
43     int nit_t;
44
45     cout << "Robust 2nd ver" << endl;
46     cout << rob.find_root(nit_t);
47     cout << '\t' << nit_t << endl;
48
49     return 0;
50 }
```

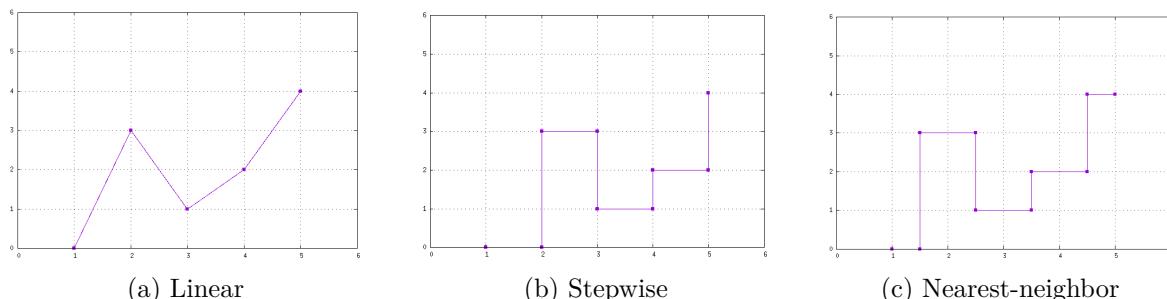


Figure 3.1

Exercise 3 Interpolation (*exam 01/09/2017*)

You have to develop a library that, given a vector of points, calculates their interpolation in a two dimensional Euclidean space with three different implementation methods, as shown in Figure 3.1. The figure shows three interpolation diagrams for the following vector of points:

$$\{(1.0, 0.0), (2.0, 3.0), (3.0, 1.0), (4.0, 2.0), (5.0, 4.0)\}$$

In particular the library provides:

- *Linear* interpolation, which connects the points with a straight line.
 - *Stepwise* interpolation, which provides, within an interval bounded by subsequent points, the y value of the left extreme.
 - *Nearest-neighbor* interpolation, which locates in the vector the nearest point and assigns the same value.

Assume that each point is defined as follows:

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

and that the interpolation methods have access to a vector of points, which are sorted according to their x -axis coordinates.

```
class Interpolation {  
    std::vector<Point> points; //sorted  
    //...  
};
```

You have to complete the definition of the class `Interpolation` and to provide a set of classes that implement your library. The `interpolate` methods, whose prototype is listed below, receive as input the x coordinate and return the corresponding y value for each interpolation option shown in the figure. If the input x does not belong to the domain spanned by the points, then your methods should return `NaN`.

```
double interpolate (double x) const;
```

Please note that it is not allowed to have an instance of class `Interpolation` without knowing which scheme it implements. Assume that the class `Point` provides all the getter and setter methods you need.

Exercise 3 - Solution

***** file Point.hpp *****

```
1 #ifndef POINT_HH
2 #define POINT_HH
3
4 class Point
5 {
6     double x;
7     double y;
8
9 public:
10    Point (double f1, double f2);
11
12    double get_x (void) const;
13    double get_y (void) const;
14};
15
16#endif // POINT_HH
```

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

```
1 #include "Point.hpp"
2
3 Point::Point (double f1, double f2)
4     : x (f1), y (f2) {}
5
6 double
7 Point::get_x (void) const
8 {
9     return x;
10}
11
12 double
13 Point::get_y (void) const
14 {
15     return y;
16}
```

The class `Interpolation` is the base class for all the interpolation methods we have to implement. In particular, it is an `abstract` class, because the method

`virtual double interpolate (double) const = 0;`

must be specialized by the derived classes according to the different interpolation methods. Therefore, no objects of the class `Interpolate` can be instantiated: the user must specify which method he wants to choose.

The class stores a `std::vector` of `Points`, sorted with respect to the x -coordinate. It implements the method

`bool range_check (double) const;`

that receives a parameter x and it returns `true` if the coordinate stays between the extrema of the interval in which we want to perform the interpolation (namely, if x stays between the x -coordinates of the first and the last point in the vector `points`).

Note that the constructor of the class `Interpolation`, and therefore the constructors of the derived classes, are declared `explicit`. This is usually a good practice with constructors

that receive a single parameter in order to avoid implicit conversions, which can lead to counterintuitive behaviours.

Moreover, note also that the class `Interpolation` explicitly defines a `destructor`, even if it is the `default` one. When this is done in a polymorphic structure, the constructor of the base class must always be declared `virtual`.

***** file `Interpolation.hpp` *****

```
1 #ifndef INTERPOLATION_HH
2 #define INTERPOLATION_HH
3
4 #include <limits>
5 #include <vector>
6
7 #include "Point.hpp"
8
9 class Interpolation
10 {
11 protected:
12     std::vector<Point> points; // sorted_vector
13     constexpr static double err_val = std::numeric_limits<double>::quiet_NaN();
14
15 public:
16     virtual double interpolate (double) const = 0;
17     bool range_check (double) const;
18
19     Interpolation (const std::vector<Point> &);
20     virtual ~Interpolation (void) = default;
21 };
22
23 #endif // INTERPOLATION_HH
```

***** file `Interpolation.cpp` *****

```
1 #include "Interpolation.hpp"
2
3 #include "Point.hpp"
4
5 bool
6 Interpolation::range_check (double x) const
7 {
8     return not (x < points.front().get_x() or x > points.back().get_x());
9 }
10
11 Interpolation::Interpolation (const std::vector<Point> & points)
12     : points (points) {}
```

The class `LinearInterpolation` inherits from `Interpolation`, thus it only modifies the constructor and the method `interpolate` (see the file "LinearInterpolation.cpp").

***** file `LinearInterpolation.hpp` *****

```
1 #ifndef LINEAR_INTERPOLATION_HH
2 #define LINEAR_INTERPOLATION_HH
3
```

```

4 #include <vector>
5
6 #include "Interpolation.hpp"
7
8 #include "Point.hpp"
9
10 class LinearInterpolation : public Interpolation
11 {
12 public:
13     explicit LinearInterpolation (const std::vector<Point> & points);
14     virtual double interpolate (double x) const override;
15 };
16
17 #endif // LINEAR_INTERPOLATION_HH

```

The constructor (see lines 5 and 6 of the file "LinearInterpolation.cpp") receives a vector of Points and it delegates the constructor of the base class to initialize the vector `points`.

The method `interpolate` (see lines 8 to 34) implements the specialization of the corresponding method in the base class. It receives as parameter a variable x and it computes the corresponding y through a linear interpolation of the points in `points`. In particular, it initializes the variable `result` with NaN , so that this value is returned if the interpolation is not possible, i.e. if the method `range_check` returns `false`. If this is not the case, the method loops over the points in `points` until it reaches the end of the vector or it finds a point whose x -coordinate is greater than or equal to the given x (remember that we assume the vector `points` to be sorted by the x -variable). At the end of the loop, it computes the value of the result.

***** file `LinearInterpolation.cpp` *****

```

1 #include "LinearInterpolation.hpp"
2
3 #include "Point.hpp"
4
5 LinearInterpolation::LinearInterpolation (const std::vector<Point> & points)
6     : Interpolation (points) {}
7
8 double
9 LinearInterpolation::interpolate (double x) const
10 {
11     double result (err_val);
12
13     if (range_check (x))
14     {
15         std::vector<Point>::const_iterator previous = points.cbegin (),
16             current = previous + 1;
17
18         while (current != points.cend () and current -> get_x () < x)
19         {
20             ++current;
21             ++previous;
22         }
23
24         if (current != points.cend ())
25         {

```

```

26     const double
27         x1 (previous -> get_x()), x2 (current -> get_x()),
28         y1 (previous -> get_y()), y2 (current -> get_y());
29         result = y1 + (y2 - y1) * (x - x1) / (x2 - x1);
30     }
31 }
32
33 return result;
34 }
```

The class `StepwiseInterpolation` inherits from `Interpolation`, thus it only modifies the constructor and the method `interpolate` (see the file "StepwiseInterpolation.cpp").

***** file `StepwiseInterpolation.hpp` *****

```

1 #ifndef STEPWISE_INTERPOLATION_HH
2 #define STEPWISE_INTERPOLATION_HH
3
4 #include <vector>
5
6 #include "Interpolation.hpp"
7
8 #include "Point.hpp"
9
10 class StepwiseInterpolation: public Interpolation
11 {
12 public:
13     explicit StepwiseInterpolation (const std::vector<Point> &);
14
15     double interpolate (double x) const override;
16 };
17
18 #endif // STEPWISE_INTERPOLATION_HH
```

The logic behind the implementation is exactly the same we had in the class `LinearInterpolation`; the only difference stays, of course, in the formula we use in order to compute the result of the method `interpolate` (see line 26 of the file "StepwiseInterpolation.cpp"). In particular, the result returned is, within an interval bounded by subsequent points, the y value of the left extreme.

***** file `StepwiseInterpolation.cpp` *****

```

1 #include "StepwiseInterpolation.hpp"
2
3 #include "Point.hpp"
4
5 StepwiseInterpolation::StepwiseInterpolation (const std::vector<Point> & points)
6     : Interpolation (points) {}
7
8 double
9 StepwiseInterpolation::interpolate (double x) const
10 {
11     double result (err_val);
12
13     if (range_check (x))
```

```

14 {
15     std::vector<Point>::const_iterator previous = points.cbegin(),
16     current = previous + 1;
17
18     while (current != points.cend () and current -> get_x () < x)
19     {
20         ++current;
21         ++previous;
22     }
23
24     if (current != points.cend ())
25     {
26         result = previous -> get_y ();
27     }
28 }
29
30 return result;
31 }
```

The class `NearestNeighborInterpolation` inherits from `Interpolation`, thus it only modifies the constructor and the method `interpolate` (see the file "NearestNeighborInterpolation.cpp").

***** file `NearestNeighborInterpolation.hpp` *****

```

1 #ifndef NEAREST_NEIGHBOR_INTERPOLATION_HH
2 #define NEAREST_NEIGHBOR_INTERPOLATION_HH
3
4 #include <vector>
5
6 #include "Interpolation.hpp"
7
8 #include "Point.hpp"
9
10 class NearestNeighborInterpolation: public Interpolation
11 {
12 public:
13     explicit NearestNeighborInterpolation (const std::vector<Point> &);
14
15     double interpolate (double) const override;
16 };
17
18 #endif // NEAREST_NEIGHBOR_INTERPOLATION_HH
```

As in the previous case, the only difference with respect to the class `LinearInterpolation` stays in lines 27 to 30 of the file "NearestNeighborInterpolation.cpp". In particular, the result returned by the method `interpolate` is the y -coordinate of the closest point to the given x .

***** file `NearestNeighborInterpolation.cpp` *****

```

1 #include "NearestNeighborInterpolation.hpp"
2
3 #include "Point.hpp"
4
5 NearestNeighborInterpolation::NearestNeighborInterpolation
6 (const std::vector<Point> & points)
```

```

7   : Interpolation (points) {}
8
9 double
10 NearestNeighborInterpolation::interpolate (double x) const
11 {
12   double result (err_val);
13
14   if (range_check (x))
15   {
16     std::vector<Point>::const_iterator previous = points.cbegin (),
17     current = previous + 1;
18
19   while (current != points.cend () and current -> get_x () < x)
20   {
21     ++current;
22     ++previous;
23   }
24
25   if (current != points.cend ())
26   {
27     const double first_distance = x - previous -> get_x (),
28     second_distance = current -> get_x () - x;
29     const bool first_closest = second_distance > first_distance;
30     result = first_closest ? previous -> get_y () : current -> get_y ();
31   }
32 }
33
34 return result;
35 }
```

As above mentioned, the `interpolate` implementations of the three classes mainly differ for the compute the result, but all require to find the left closest point (two of them also require the right closest point). Indeed there is a repeated `while` loop in all the implementations which can be factorized and included in the `range_check` function (changing its return type).

Exercise 4 Knapsack Problem (*exam 03/02/2017*)

The multidimensional knapsack problem (MKP) is an extension of the widely known knapsack problem, where items are characterized by multiple dimensions. In particular, let us consider a set of n items, with each item $1 \leq j \leq n$ having an associated value v_j , a weight w_j and a size s_j . The binary decision variable x_j is used to select the item. The objective is to pick some of the items, with maximal total value, while obeying that the maximum total weight and maximum total size of the chosen items must not exceed the knapsack capacities W and S . In other words, the MKP can be formulated as:

$$\max \sum_j v_j x_j$$

subject to:

$$\begin{aligned} \sum_j w_j x_j &\leq W \\ \sum_j s_j x_j &\leq S \\ x_j &\in \{0, 1\}, \quad \forall j \end{aligned}$$

Given the class hierarchy shown in Figure 3.2 and the declarations for the Item and MultiDimKnapsack classes:

***** file Item.h *****

```

1 #ifndef ITEM_H_
2 #define ITEM_H_
3
4 class Item
5 {
6     double weight;
7     double size;
8     double value;
9
10 public:
11     Item (double w, double s, double v)
12         : weight (w), size (s), value (v) {}
13
14     double get_size (void) const;
15     double get_weight (void) const;
16     double get_value (void) const;
17
18     void set_size (double size);
19     void set_weight (double weight);
20     void set_value (double value);
21
22     void print (void) const;
23 };
24
25 #endif /* ITEM_H_ */

```

***** file MultiDimKnapsack.h *****

```

1 #ifndef MULTIDIMKNAPSACK_H_
2 #define MULTIDIMKNAPSACK_H_
3
4 #include <vector>
5
6 #include "Item.h"
7
8 class MultiDimKnapsack
9 {
10 protected:
11     double w;
12     double s;

```

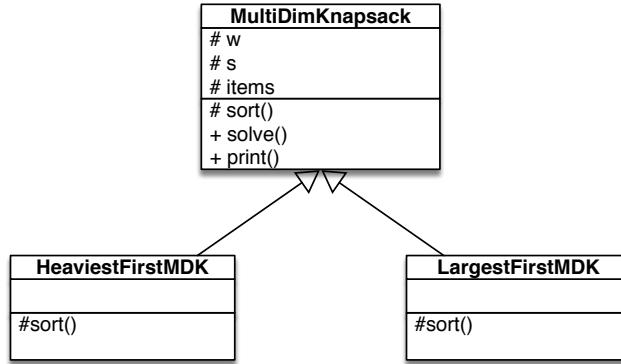


Figure 3.2: MKP program Class Diagram

```

13     std::vector<Item> items;
14
15     // returns a vector of sorted item indices
16     virtual std::vector<unsigned> sort (void) const = 0;
17
18 public:
19     // generate n_items random items and set knapsack capacities
20     MultiDimKnapsack (int n_items, double max_weight,
21                         double max_size);
22
23     // returns the set of items included in the solution
24     std::vector<unsigned> solve (void) const;
25
26     void print (void) const;
27 };
28
29 #endif /* MULTIDIMKNAPSACK_H_ */

```

Provide the declaration and the constructor of the class `LargestFirstMDK` (or, alternatively, of `HeaviestFirstMDK`), which heuristically solves the MKP by picking the items in nonincreasing order of size (weight, respectively). Moreover, provide the implementation of the `solve` method of the `MultiDimKnapsack` class. What is the time complexity of the `solve` method?

Exercise 4 - Solution

The files reported below contain the full implementation of the classes `MultiDimKnapsack`, `LargestFirstMDK` and `HeaviestFirstMDK`, even if it was not required by the exercise.

Note that the class `MultiDimKnapsack` is an `abstract` class. In particular, what we have to choose (and thus the different specializations of this class) is the way we use to sort the items that we want to insert in the knapsack. In particular, we will choose the class `LargestFirstMDK` if we want to select the items in non increasing order of size (`HeaviestFirstMDK` if we want to consider the weight).

The constructor (see lines 6 to 17 of the file "MultiDimKnapsack.cpp") receives as parameters the number of items we want to generate and the capacities of the knapsack. It generates and stores in the vector `items` the given number of objects, assigning them a weight, a size and a value.

The method `solve` (see lines 20 to 44) returns, in a `std::vector`, the indexes of the items that we choose to store in the knapsack. In particular, it sorts the items stored in `items` (line 27),

then it loops over them and it selects an item if it does not overcome the capacities of the knapsack.

```
***** file MultiDimKnapsack.cpp *****
1 #include <iostream>
2
3 #include "MultiDimKnapsack.h"
4
5 // generate n_items random items and set knapsack capacities
6 MultiDimKnapsack::MultiDimKnapsack (int n_items, double max_weight,
7                                     double max_size)
8   : w(max_weight), s(max_size)
9 {
10   for (std::size_t i = 0; i < n_items; ++i)
11   {
12     const double weight = i + 1.;
13     const double size = n_items - i;
14     const double value = (i + 1) * 100.;
15     items.push_back (Item (weight, size, value));
16   }
17 }
18
19 // returns the set of items included in the solution
20 std::vector<unsigned> MultiDimKnapsack::solve (void) const
21 {
22   std::vector<unsigned> solution;
23   double current_weight = 0;
24   double current_size = 0;
25
26   // compute solution
27   std::vector<unsigned> sorted_idx = sort();
28   for (unsigned idx: sorted_idx)
29   {
30     // check if adding the current item would lead to capacity saturation
31     const bool has_room = (current_size + items[idx].get_size () <= s)
32       and (current_weight + items[idx].get_weight () <= w);
33
34     // if there is still room add the item
35     if (has_room)
36     {
37       solution.push_back(idx);
38       current_weight += items[idx].get_weight();
39       current_size += items[idx].get_size();
40     }
41   }
42
43   return solution;
44 }
45
46 void MultiDimKnapsack::print (void) const
47 {
48   for (const Item & item: items)
```

```

49     {
50         item.print();
51         std::cout << std::endl;
52     }
53 }
```

The class `LargestFirstMDK` inherits from `MultiDimKnapsack`; it delegates the constructor of the base class to instantiate an object with a given number of items and a given capacity, while it overrides the method `sort` (see the file "LargestFirstMDK.cpp" for the implementation).

***** file `LargestFirstMDK.h` *****

```

1 #ifndef LARGESTFIRSTMDK_H_
2 #define LARGESTFIRSTMDK_H_
3
4 #include "MultiDimKnapsack.h"
5
6 class LargestFirstMDK: public MultiDimKnapsack
7 {
8     // returns a vector of sorted item indices
9     virtual std::vector<unsigned> sort (void) const override;
10
11 public:
12     LargestFirstMDK (int n_items, double max_weight, double max_size)
13         : MultiDimKnapsack (n_items, max_weight, max_size) {}
14 };
15
16 #endif /* LARGESTFIRSTMDK_H_ */
```

The method `sort` (see lines 7 to 25 of the file "LargestFirstMDK.cpp") returns a vector of `unsigned` that will store the indices of the elements of the vector `items`, taken in ascending order of size. In order to get this result, it copies all the indices of `items` in `sorted_idx` (see lines 12 and 13), then it loops over all the elements in `items` and it swaps the corresponding indices if an object has greater size than the following one.

Note that it is more convenient in terms of memory to return a vector with only the indices of the items instead of generating a new vector of `Items` stored in the proper order. Indeed, in principle, an `Item` object can be huge, while we know exactly which is the size in memory of an `unsigned`.

***** file `LargestFirstMDK.cpp` *****

```

1 #include <iostream>
2 #include <utility>
3
4 #include "LargestFirstMDK.h"
5
6 // returns a vector of sorted item indices with nonincreasing size
7 std::vector<unsigned> LargestFirstMDK::sort (void) const
8 {
9     std::vector<unsigned> sorted_idx;
10
11    // first copy all indices
12    for (unsigned i = 0; i < items.size (); ++i)
13        sorted_idx.push_back (i);
14 }
```

```

15 // sort indices
16 for (std::size_t i = 0; i < items.size()-1; ++i)
17     for (std::size_t j = i+1; j < items.size(); ++j)
18         if (items[sorted_idx[i]].get_size() < items[sorted_idx[j]].get_size())
19             {
20                 using std::swap;
21                 swap (sorted_idx[i], sorted_idx[j]);
22             }
23
24     return sorted_idx;
25 }
```

The class `HeaviestFirstMDK` inherits from `MultiDimKnapsack`; it delegates the constructor of the base class to instantiate an object with a given number of items and a given capacity, while it overrides the method `sort` (see the file "HeaviestFirstMDK.cpp" for the implementation).

***** file `HeaviestFirstMDK.h` *****

```

1 #ifndef HEAVIESTFIRSTMDK_H_
2 #define HEAVIESTFIRSTMDK_H_
3
4 #include "MultiDimKnapsack.h"
5
6 class HeaviestFirstMDK: public MultiDimKnapsack
7 {
8     // returns a vector of sorted item indices
9     virtual std::vector<unsigned> sort (void) const override;
10
11 public:
12     HeaviestFirstMDK (int n_items, double max_weight, double max_size)
13         : MultiDimKnapsack (n_items, max_weight, max_size) {}
14 };
15
16
17 #endif /* HEAVIESTFIRSTMDK_H_ */
```

The method `sort` (see lines 7 to 25 of the file "HeaviestFirstMDK.cpp") returns a vector of `unsigned` that will store the indices of the elements of the vector `items`, taken in ascending order of weight. In order to get this result, it copies all the indices of `items` in `sorted_idx` (see lines 12 and 13), then it loops over all the elements in `items` and it swaps the corresponding indices if the an object has greater weight than the following one.

***** file `HeaviestFirstMDK.cpp` *****

```

1 #include <iostream>
2 #include <utility>
3
4 #include "HeaviestFirstMDK.h"
5
6 // returns a vector of sorted indices with nonincreasing weight
7 std::vector<unsigned> HeaviestFirstMDK::sort (void) const
8 {
9     std::vector<unsigned> sorted_idx;
```

```

11 // first copy all indices
12 for (unsigned i = 0; i < items.size (); ++i)
13     sorted_idx.push_back (i);
14
15 // sort indices
16 for (std::size_t i = 0; i < items.size () - 1; ++i)
17     for (std::size_t j = i + 1; j < items.size (); ++j)
18         if (items[sorted_idx[i]].get_weight () < items[sorted_idx[j]].get_weight ())
19             {
20                 using std::swap;
21                 swap (sorted_idx[i], sorted_idx[j]);
22             }
23
24 return sorted_idx;
25 }
```

The `solve` has a complexity of $O(n^2)$, where n is the `items` size.

Indeed, the worst-case cost of the `push_back` method is $O(n)$ and it is invoked at every iteration of the loop in lines 28 to 41 (see file "MultiDimKnapsack.cpp"), thus n times.

Moreover, the sorting algorithm used in `sort` (selection sort) has complexity $O(n^2)$, thus the overall complexity for `solve` is $O(2n^2) \sim O(n^2)$.

Important note: it is always possible to implement smarter versions of the `sort` algorithm, with complexity $O(n \log n)$. However, the overall complexity of the method `solve` would remain $O(n^2)$ as long as the solution is stored in a `std::vector`, that introduces the term $O(n^2)$ due to the subsequent `push_backs`.

A possible way to reduce the overall complexity of the method is, in addition to the introduction of a smarter `solve`, to use a different container provided by the STL as, for instance, a `std::set`. As we will discuss in Chapter 6, the method `insert` of `std::sets` has a complexity of $O(\log k)$, where k is the number of elements already stored in the container. In the loop, at every iteration we might add an item to the solution, then the complexity is the sum of $O(\log k)$ with k in the range $1 : n$. The summation of the logarithms is $O(\log n!)$ and $\log(n!) = O(n \log n)$. Roughly speaking, we can say that, if we are in the middle of our item list, adding a new item has complexity $O(\log(n/2))$ hence $O(\log n)$. Then we have to add the remaining $n/2$, hence we get $O(n/2 \cdot \log n) = O(n \log n)$.

Exercise 5 University Information System (*exam 21/07/2017*)

Find the problems in the following code, if any, explain the reasons why it does or does not compile and write the correct version.

```

1 #include <vector>
2 #include <string>
3
4 using namespace std;
5
6 class course
7 {
8     // ...
9 };
10
11 class department
12 {
```

```

13 // ...
14 };
15
16 class student
17 {
18 protected:
19     string name;
20     string surname;
21     size_t id;
22     department & dpt;
23     std::vector<double> grades;
24
25 public:
26     student (const string & name, const string & surname, size_t id)
27         : name (name), surname (surname), id (id) {}
28 };
29
30 class freshman: public student
31 {
32     std::vector<course &> classes;
33
34     std::vector<double> show_grades (student & s)
35     {
36         return s.grades;
37     }
38 };

```

Exercise 5 - Solution

There are three problems in the presented code:

- The attribute `department & dpt` in `student` class has never been initialized, which is not possible because a reference always needs initialization. We have to include it in the constructor.

Note also that, since `dpt` is a reference, we should pass the argument we want to use to initialize it as a reference, i.e. we should write

```
student (const string & name, const string & surname, size_t id, department & d):
    name(name), surname(surname), id(id), dpt(d) {}
```

Indeed, if we pass the parameter `d` by value, the reference `dpt` will be binded to an object allocated in the stack, which therefore will be destroyed when we exit the scope of the constructor. Moreover, we cannot pass `d` as a `const` reference because `dpt` is declared as non `const`.

- `std::vector<course &> classes` in `freshman` class is problematic since creating a vector of references is not allowed. It could be replaced by a vector of (shared) pointers.
- `std::vector<double> show_grades (student & s)` cannot access a `protected` member of the base class. `grades` could be retrieved by a getter function defined in `student` class.

```

1 #include <vector>
2 #include <string>
```

```

3
4 using namespace std;
5
6 class course
7 {
8     // ...
9 };
10
11 class department
12 {
13     // ...
14 };
15
16 class student
17 {
18
19 protected:
20     string name;
21     string surname;
22     size_t id;
23     department & dpt;
24     std::vector<double> grades;
25
26 public:
27     student (const string & name, const string & surname, size_t id, department & d)
28         : name (name), surname (surname), id (id), dpt (d) {}
29
30     const std::vector<double> & get_grades (void) const
31     {
32         return grades;
33     }
34 };
35
36 class freshman: public student
37 {
38     std::vector<course * > classes;
39
40     std::vector<double> show_grades (const student & s)
41     {
42         return s.get_grades();
43     }
44 };

```

Exercise 6 Matrix Computation (*exam 17/02/2017*)

You have to implement a program for matrix computation. The program needs to manage both dense and block matrices of double precision numbers. Dense matrices are allocated when created. Vice versa, block matrices are ideal to store sparse data and their size and memory storage is changed dynamically when blocks are added. The software architect requires you to implement the class hierarchy reported in Figure 3.3. The implementation of the **Block** class (which includes a dense matrix and is characterized by the indexes of its top

left and bottom right elements) is also provided and it is reported below, in the `block.hpp` file. Note that the values of a new block are provided per row in a single vector `vals`.

```

1 #ifndef BLOCK_H_
2 #define BLOCK_H_
3
4 #include <vector>
5
6 #include "dense_matrix.hpp"
7
8 class Block
9 {
10 private:
11     std::size_t top_left_row;
12     std::size_t top_left_col;
13     std::size_t bottom_right_row;
14     std::size_t bottom_right_col;
15
16     DenseMatrix dm;
17
18 public:
19     Block (std::size_t top_left_r, std::size_t top_left_c,
20             std::size_t bottom_right_r, std::size_t bottom_right_c,
21             const std::vector<double> & vals);
22
23     std::size_t get_bottom_right_col() const;
24     std::size_t get_bottom_right_row() const;
25
26     std::size_t get_top_left_col() const;
27     std::size_t get_top_left_row() const;
28
29     double & operator () (std::size_t i, std::size_t j);
30
31     double operator () (std::size_t i, std::size_t j) const;
32 };
33
34 #endif /* BLOCK_H_ */

```

You have to provide the implementation of the `BlockMatrix` class. In particular:

1. Complete "matrix.hpp", provided below, for the ancestor class:

```

1 #ifndef MATRIX_H_
2 #define MATRIX_H_
3
4 class Matrix
5 {
6     ???
7     ??? n_rows;
8     ??? n_cols;
9
10 public:
11     Matrix (??? rows, ??? cols);
12

```

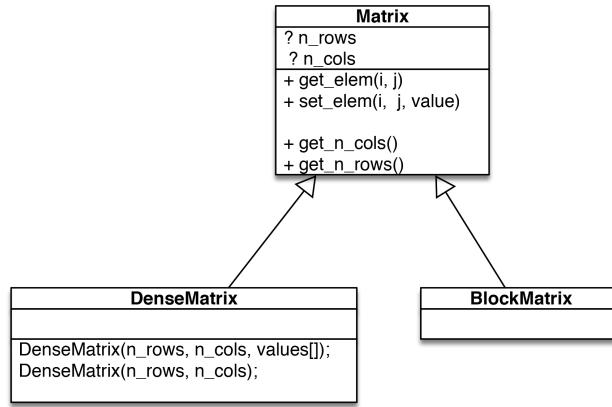


Figure 3.3: Matrix program Class Diagram.

1	2	3		0
4	5	6		
			7	8
			10	11
0			12	13
			14	15

Figure 3.4: Matrix `add_block()` example.

```

13     ??? operator () (std::size_t i, std::size_t j) ???;
14
15     ??? operator () (std::size_t i, std::size_t j) ???;
16
17     ??? get_n_cols() ???
18     ??? get_n_rows() ???
19 };
20
21 #endif /* MATRIX_H_ */

```

2. Provide the implementation of the `add_block()` method, which receives a new block as parameter. The method should also update the matrix size accordingly.
3. Provide the get-like implementation of `operator()`, which receives as parameters the row and column indexes of the element to be read and return its value. If the indexes are out of range, the method prints a message on the error stream. If the indexes are within the range, but for an element not explicitly initialized in a block, 0 is returned as default value.
4. Provide the set-like implementation of the method `operator()`, which receives as parameters the row and column indexes of an element and return a reference to the element in the matrix. If the element was not previously allocated in a block, a new block with a single element is added.

Figure 3.4 shows, as an example, a block matrix including two blocks: *block 1*, including numbers between 1 and 6, whose top left indexes are (0, 0) and bottom right indexes (1, 2); *block 2*, including numbers between 7 and 15, characterized by top left indexes (2, 3) and bottom right (4, 5).

Exercise 6 - Solution

Since the implementation of both the get-like and set-like `operator()` depends on the structure of the matrix (namely on the fact that it can be a dense matrix or a block matrix), the base class `Matrix` is designed as an `abstract` class. The get-like `operator()` it is a `const` method (i.e., it does not modify the object) which returns the value of a cell of a matrix. The set-like `operator()` it cannot be `const` (in the `BlockMatrix` modifies the internal data structure) which returns a reference to the internal cell of the matrix.

```
***** file matrix.hpp *****
```

```
1 #ifndef MATRIX_H_
2 #define MATRIX_H_
3
4 #include <cstddef>
5
6 class Matrix
7 {
8 protected:
9     std::size_t n_rows;
10    std::size_t n_cols;
11
12 public:
13     Matrix (std::size_t rows, std::size_t cols);
14
15     virtual double & operator () (std::size_t i, std::size_t j) = 0;
16
17     virtual double operator () (std::size_t i, std::size_t j) const = 0;
18
19     std::size_t get_n_cols() const;
20     std::size_t get_n_rows() const;
21
22     virtual ~Matrix (void) = default;
23 };
24
25 #endif /* MATRIX_H_ */
```

The class `DenseMatrix`, whose implementation was not required by the exercise, is designed as a `std::vector<double>`, where the matrix rows are stored one after the other. The two `operator()` are overridden from the base class on top of this choice.

As you can see in the corresponding header file in the text above, the class `Block` is implemented on top of the class `DenseMatrix` (namely it stores a dense matrix and it extends it with information about the indices of the block).

Note that the keyword `override` in the derived class is not mandatory. However, it is always a good practice to specify them: since, for example, when you override a method you cannot change in any way its parameters or its return type, if you mark it with `override` and then, by mistake, you define the parameters as `int` instead of `unsigned`, the compiler raises an error, which makes the code easier to debug.

```
***** file dense_matrix.hpp *****
```

```
1 #ifndef DENSE_MATRIX_HPP
2 #define DENSE_MATRIX_HPP
3
4 #include <iostream>
```

```

5 #include <vector>
6
7 #include "matrix.hpp"
8
9 class DenseMatrix : public Matrix
10 {
11     private:
12         std::vector<double> m_data;
13
14         std::size_t sub2ind (std::size_t i, std::size_t j) const;
15
16     public:
17         DenseMatrix (std::size_t rows, std::size_t columns,
18             double value = 0.0);
19
20         DenseMatrix (std::size_t rows, std::size_t columns,
21             const std::vector<double>& values);
22
23         void read (std::istream & );
24
25         void swap (DenseMatrix & );
26
27         double & operator () (std::size_t i, std::size_t j) override;
28
29         double operator () (std::size_t i, std::size_t j) const override;
30
31         DenseMatrix transposed (void) const;
32
33         double * data (void);
34
35         const double * data (void) const;
36     };
37
38 DenseMatrix operator * (const DenseMatrix &, const DenseMatrix & );
39
40 void swap (DenseMatrix &, DenseMatrix & );
41
42 #endif // DENSE_MATRIX_HH

```

A BlockMatrix is implemented as a vector of Blocks. It has two **private** methods (see lines 12 and 14 of the file "block_matrix.hpp"); the first one, given two indices, returns **true** if they do not exceed the dimension of the global matrix. The second one, given two indices and a Block, returns **true** if the indices are internal to the block.

The class overrides the two public **operator()** (see the file "BlockMatrix.cpp" for the relative implementation).

***** file block_matrix.hpp *****

```

1 #ifndef BLOCKMATRIX_H_
2 #define BLOCKMATRIX_H_
3
4 #include <vector>
5
6 #include "block.hpp"

```

```

7
8 class BlockMatrix: public Matrix
9 {
10     std::vector<Block> blocks;
11
12     bool indexes_in_range (std::size_t i, std::size_t j) const;
13
14     bool indexes_in_block (std::size_t i, std::size_t j, const Block & block) const;
15
16 public:
17     BlockMatrix();
18
19     void add_block (const Block & block);
20
21     double & operator () (std::size_t i, std::size_t j) override;
22
23     double operator () (std::size_t i, std::size_t j) const override;
24 };
25
26 #endif /* BLOCKMATRIX_H_ */

```

The get-like `BlockMatrix::operator()` (see lines 21 to 37 of the file "block_matrix.cpp") is implemented on top of the corresponding operator for the class `Block`. In particular, it returns `Nan` if the indices passed as parameters exceed the matrix dimension. Otherwise, it loops over all the blocks. If `indexes_in_block` returns `true`, the value given by get-like `Block::operator()` is returned. If `indexes_in_block` is `false` for every block, the value 0 is returned.

The set-like `BlockMatrix::operator()` (see lines 39 to 54) has a similar structure. The main difference is given by the fact that, if the indices of the element we want to add do not exceed the matrix dimension but they are not internal to any block, a new block must be added. If, at the end of the loop (line 50), the function does not exit, we need to create a new block, which will store only the element passed as parameter to the method.

Note that the method `add_block`, which is called in line 52 of `operator()` and is implemented in lines 56 to 62, not only appends the new `Block` to the vector of blocks, but it also updates the number of rows and the number of columns of the global matrix (which can be accessed directly because they are `protected` members of the base class). This is needed also because the class is defined in such a way that we can initialize only empty `BlockMatrices`, that can be populated with new blocks only through this method. Moreover, the block which is actually stored in the `blocks` vector is a copy of the variable `block` created at line 51. Since we want to return the reference to the cell stored in the former, at line 53 we cannot invoke `operator()` on `block`, but we must invoke it on `blocks.back()` (which returns a reference to the last element of the vector).

Important note: a possible alternative would be to design the set-like `operator()` so as to add a new block only when the given indexes are inside the matrix, even if they do not belong to any existing block, while returning an error message if the required element is outside the global matrix. In this case, we could add before line 41

```

if (! indexes_in_range(i,j))
{
    std::cerr << "indexes out of range!" << std::endl;
    return operator()(0,0);
}

```

Notice that, given how the method is defined, we must always return a reference to an existing element. Therefore, we cannot return NaN as in the get-like version of `operator()`, since in that case we would return a reference to a variable that is destroyed as soon as we exit the scope of the function.

```
***** file block_matrix.cpp *****
1 #include <limits>
2 #include <iostream>
3
4 #include "block_matrix.hpp"
5
6 BlockMatrix::BlockMatrix()
7   : Matrix (0, 0) {}
8
9 bool BlockMatrix::indexes_in_range (std::size_t i, std::size_t j) const
10 {
11   return (i < n_rows) && (j < n_cols);
12 }
13
14 bool BlockMatrix::indexes_in_block (std::size_t i, std::size_t j,
15                                     const Block & block) const
16 {
17   return (i >= block.get_top_left_row()) && (i <= block.get_bottom_right_row())
18     &&
19   (j >= block.get_top_left_col()) && (j <= block.get_bottom_right_col());
20 }
21
22 double BlockMatrix::operator()(std::size_t i, std::size_t j) const
23 {
24   if (!indexes_in_range (i, j))
25   {
26     std::cerr << "indexes out of range!" << std::endl;
27     return std::numeric_limits<double>::quiet_NaN();
28   }
29   for (const Block & block : blocks)
30   {
31     // if i, j in block coordinates
32     if (indexes_in_block (i, j, block))
33       // access proper indexes element and return
34     return block(i - block.get_top_left_row(), j - block.get_top_left_col());
35   }
36
37   return 0;
38 }
39
40 double & BlockMatrix::operator() (std::size_t i, std::size_t j)
41 {
42   for (Block & block : blocks)
43   {
44     // if i, j in block coordinates
45     if (indexes_in_block (i, j, block))
46     {
```

```

46     // change proper indexes element and return
47     return block(i - block.get_top_left_row(), j - block.get_top_left_col());
48 }
49 }
50 // Create a single element block
51 Block block (i, j, i, j, {0});
52 add_block (block);
53 return blocks.back()(0, 0);
54 }
55
56 void BlockMatrix::add_block (const Block & block)
{
57     blocks.push_back (block);
58     // Update matrix size
59     n_rows = std::max (n_rows, block.get_bottom_right_row() + 1);
60     n_cols = std::max (n_cols, block.get_bottom_right_col() + 1);
61 }
62 }
```

Exercise 7 Input Scaler (*exam 19/02/2018*)

You have to develop a C++ library for face recognition that takes as input a bi-dimensional matrix representing an image data. As the source of data may vary (for the sake of simplicity, assume every pixel is a **double**), the library includes classes for the scaling of the input data. In particular, the library includes two methods for scaling a $\mathbb{R}^{m \times n}$ matrix per column: a **StandardScaler** and a **MinMaxScaler**. For each method, the transformation is defined as follows:

- **StandardScaler**:

$$A'_{ij} = \frac{A_{ij} - \bar{A}_j}{S_j},$$

where \bar{A}_j is the sample mean of the entries in column j and S_j is the square root of the unbiased estimator for their variance.

- **MinMaxScaler**:

$$A'_{ij} = m + \frac{M - m}{M_j - m_j} (A_{ij} - m_j),$$

where m and M are the minimum and maximum values among which we want to perform the scaling; m_j is the minimum entry in column j , while M_j is the maximum.

You have to elaborate a solution for the aforementioned scaling approaches, taking as input a bi-dimensional matrix along with its dimensions. Each scaler class must declare the following methods: **fit**, **fit_transform**, and **inverse_transform**. The **fit** method should parse the columns of the matrix and obtain the data structure used by that specific scaler; the **fit_transform** method should perform the transformation (scaling); finally, the **inverse_transform** should perform the inverse transformation (reverse the scaling).

Provide only the header file(s) for the class(es) of your solution.

Exercise 7 - Solution

The solution consists of three classes: an **abstract** class **Scaler** and two derived classes: **StandardScaler** and **MinMaxScaler**. The base class contains three **pure virtual** member functions — namely, **fit**, **fit_transform**, and **inverse_transform** — which are overridden by each derived class. It also contains a vector of vectors of **doubles** representing the bi-dimensional matrix and two **unsigned** integers representing the number of rows (**unsigned r**) and columns (**unsigned c**).

In each case (standard and min-max), the method **fit** should parse the columns of the matrix and obtain the corresponding data structures used by the transformation: two vectors of **doubles** representing the mean and the standard deviation of each matrix column in the **StandardScaler** class, and two vectors of **doubles** representing the min and max of each matrix column in the **MinMaxScaler** class. In particular, the derived **MinMaxScaler** class presents two additional data members: a **double** **min** and a **double** **max**, which represent the range in which the min-max scaling of the matrix should happen.

Finally, the **fit_transform** and **inverse_transform** methods correspond to the actual scaling and inverse scaling of the matrix. The implementation of each of these methods by the derived classes (not required in this exercise) should be consistent with their corresponding formulas.

The header files of the three classes are provided below:

***** file **Scaler.hpp** *****

```
1 #ifndef __SCALER_HEADER__
2 #define __SCALER_HEADER__
3
4 #include <vector>
5
6 class Scaler
7 {
8 public:
9     typedef std::vector<std::vector<double>> matrix;
10
11 private:
12     matrix m;
13     unsigned r;
14     unsigned c;
15
16 public:
17     Scaler(const matrix & m_, unsigned r_, unsigned c_) : m(m_), r(r_), c(c_) {};
18
19     virtual void fit(void) = 0;
20     virtual void fit_transform(void) = 0;
21     virtual void inverse_transform(void) = 0;
22 };
23
24 #endif
```

***** file **StandardScaler.hpp** *****

```
1 #ifndef __STANDARD_SCALER_HEADER__
2 #define __STANDARD_SCALER_HEADER__
3
```

```

4 #include "Scaler.hpp"
5 #include <vector>
6
7 class StandardScaler : public Scaler
8 {
9     std::vector<double> means;
10    std::vector<double> sigmas;
11
12 public:
13
14     StandardScaler(const matrix & m__, unsigned r__, unsigned c__) : Scaler(m__, r__, c__)
15         {};
16
17     void fit(void) override;
18     void fit_transform(void) override;
19     void inverse_transform(void) override;
20 };
21
22 #endif

```

***** file MinMaxScaler.hpp *****

```

1 #ifndef __MIN_MAX_SCALER_HEADER__
2 #define __MIN_MAX_SCALER_HEADER__
3
4 #include "Scaler.hpp"
5 #include <vector>
6
7 class MinMaxScaler : public Scaler
8 {
9     std::vector<double> mins;
10    std::vector<double> maxs;
11    double min;
12    double max;
13
14 public:
15
16     MinMaxScaler(const matrix & m__, unsigned r__, unsigned c__, double min__, double
17                 max__)
18         : Scaler(m__, r__, c__), min(min__), max(max__) {};
19
20     void fit(void) override;
21     void fit_transform(void) override;
22     void inverse_transform(void) override;
23 };
24
25 #endif

```

Exercise 8 K - means *

Provide an implementation for the K-means algorithm described in the following.

K-means method¹:

Suppose that we have a set of n observations of some quantitative variables

$$C = \{\underline{x}_1, \dots, \underline{x}_n\}$$

where

$$\underline{x}_i \in \mathbb{R}^p, \forall i \in I := \{1, \dots, n\}$$

We also have a set of labels (at this stage we will consider their number as given and such that $\text{card}(\mathcal{L}) = k, k \in [1, n] \cap \mathbb{N}$)

$$\mathcal{L} := \{l_1, \dots, l_k\}, K := \{1, \dots, k\}$$

Our goal is to construct a (measurable) function to associate each and every element of C to one label in \mathcal{L} . It is easy to see that it makes no difference to work with \mathcal{L} or K as we can always find a bijection between the two sets.

We have reasons to believe that the elements belonging to one particular group are somewhat more similar to each other than the ones that are to members of other groups and that one element can belong to one group and one group only. We also assume that there are no empty groups. Translating this into mathematical terms, we can say that we want to find a collection

$$\{C_i\}_{i \in K}$$

such that

$$C_i \cap C_j = \emptyset, i \neq j$$

$$\bigcup_{j=1}^k C_j = C$$

or, in other words, we want to partition C into k clusters.

We also want to define a function

$$\delta : \underline{x}_i \longmapsto l_i$$

such that

$$\forall i \in K, \delta^{-1}(l_i) \neq \emptyset.$$

One immediate idea is to consider the following:

$$\delta(\underline{x}) = \sum_{i=1}^k \chi_{C_i}(\underline{x}) i,$$

where

$$\chi_A(\underline{x}) = \begin{cases} 1, & \text{if } \underline{x} \in A \\ 0, & \text{if } \underline{x} \notin A \end{cases}$$

so that we have a function that takes as an input a value from C and gives as an output the index of the label we are considering (and we know they are finite).

¹Notes by Andrea Mascaretti

References:

- Richard, Johnson and Wichern, Dean. *Applied Multivariate Statistical Analysis*. 6th ed. New York City: Pearson Education, 2008
- James, Gareth., Witten, Daniela., Hastie, Trevor., Tibshirani, Robert. *An Introduction to Statistical Learning*. 1st ed. New York City: Springer, 2013

The main issue is to find a method to partition the set C into the clusters. First of all, we need to define in a more precise manner the meaning of similarity: we will consider the squared *Euclidean distance* for this purpose

$$d(\underline{x}_i, \underline{x}_j) = \sum_{k=1}^p (x_{ik} - x_{jk})^2.$$

We now need to define some *loss* function and try to minimize it. The most natural idea follows from the assumption that members of one given cluster will be closer to each other than to members of other clusters: this is equivalent to stating that the correct partitioning will minimize the variability within the clusters. We need to translate this key concept into an objective function.

We define

$$I_k = \{i \in I \text{ s.t. } \underline{x}_i \in C_k\}$$

and in this case we will have

$$W(C_k) = \frac{1}{\text{card}(C_k)} \sum_{i,j \in I_k} \sum_{l=1}^p (x_{il} - x_{jl})^2.$$

and the problem becomes as follows

$$\min_{C_1, \dots, C_K} \left\{ \sum_{k=1}^K W(C_k) \right\}$$

Now we want to find a way to solve this problem and here we find the first issue as there are almost K^n ways to partition n observations into K clusters. We shall therefore look for a local optimum as follows.

It is important to notice that this function surely decreases for increasing k and in the extreme case of $k = n$ we have that each cluster contains only one point and we have no variability whatsoever. It is important to understand that we are looking for classes that do have some internal variability.

Algorithm for K-means clustering

1. Randomly assign a number, from 1 to K , to each observation: they will serve as initial cluster assignments. Set $\mu := 0$.
2. Iterate what follows until termination criterion holds:
 - (a) For each cluster, compute the centroid

$$\underline{c}_i^\mu = \frac{1}{\text{card}(C_i^\mu)} \sum_{j \in I_i^\mu} \underline{x}_j,$$

- (b) Assign each observation to the cluster whose centroid is closest (with respect to the Euclidean distance) as follows:

$$\forall j \in I, i^* = \min_{i \in K} \{d(\underline{x}_j, \underline{c}_i^\mu)\} = \min \{d(\underline{x}_j, \underline{c}_1^\mu), \dots, d(\underline{x}_j, \underline{c}_K^\mu)\} \implies \underline{x}_j \in C_{i^*}^\mu,$$

- (c) Set $\mu := \mu + 1$
3. Termination occurs when cluster assignments stop changing.

Note that the algorithm we have just defined is descent, that is it improves the solution at each iteration so that when the termination condition is true, we are certain to have reached a local optimum. Therefore, it is a good practice to iterate the algorithm various times and pick the best solution.

There is also one last important point to state. Any time we perform the K-means methods, we will find k clusters on our data. Now, the question is: are we really separating actual subgroups or are we just clustering the noise? Moreover, suppose that we are dealing with a case in which the vast majority of data belongs to a small number of subgroups, but for a small subset that is quite different from the rest. As our method forces our data into a fixed number of clusters, the presence of an outlying class might bring to distorted results. So, it is good practice not only to try various values of k to see what happens, but also to try and apply the algorithm to subsets of our initial set, in order to check that results remain somewhat stable.

Exercise 8 - Solution

The first element we need in order to implement this program is a `Point`. Since we do not know *a priori* the dimension p of the space, we can use, for example, the class `Point` implemented in Exercise 5 of Chapter 2 (Section 2.2).

The class `Centroid` inherits from `Point` (a centroid is, in fact, a point). It implements a method that, given a vector of pointers to `Points`, updates the coordinates of the centroid, using the formula reported in section (2.a) of the algorithm description above.

Note that, in line 11 of file "Centroid.cpp", since `ps` is a vector of pointers, we use the arrow operator instead of the dot operator to call the method `Point::get_coord`.

***** file Centroid.h *****

```

1 #ifndef CENTROID_H_
2 #define CENTROID_H_
3
4 #include <iostream>
5 #include <vector>
6
7 #include "Point.h"
8
9 class Centroid : public Point
10 {
11 public:
12     Centroid (const std::vector<double>& coords)
13     : Point (coords) {};
14
15     void
16     update_coords (const std::vector<Point *>& ps);
17 };
18
19 #endif /* CENTROID_H_ */

```

***** file Centroid.cpp *****

```

1 #include "Centroid.h"
2
3 void
4 Centroid::update_coords (const std::vector<Point *> & ps)
5 {

```

```

6 std::vector<double> new_coords(x.size(),0);
7
8 for (std::size_t i = 0; i < ps.size (); ++i)
9 {
10    for (std::size_t j = 0; j < x.size (); ++j)
11        new_coords[j] += ps[i] -> get_coord (j);
12    }
13
14 for (std::size_t j = 0; j < x.size (); ++j)
15    new_coords[j] /= ps.size ();
16 x.swap (new_coords);
17 }
```

The class `Clustering` implements the algorithm described in the notes above. In particular, it stores a vector of `Points`, a vector of `labels`, represented by `unsigned`, a vector of `Centroids` and a `std::vector<std::vector<Point *>>` (hidden in the user defined name `clusters_type`, see line 13 of the file "Clustering.h") which represent the clusters.

It implements a `private` method (lines 28 and 29), which, given a `Point`, returns the index of the closest `Centroid`, and a `public` method (lines 45 and 46), which performs the clustering. Note that we could have chosen to see the label as an attribute of the class `Point`, instead of saving it in a separate vector. This has not been done for two reasons: the first one is that, in this way, the `Point` class remains more general, it is not extended by a property that is needed only in this exercise. The second one is that, in this way, we have an information that allows to connect easily the clusters to the points, i.e. it gives the possibility to access directly the point or the cluster we need. If we had considered the label as an attribute of each point, in order to read its value we would have to scan all the vector `points`, which increases the overall complexity of the algorithm.

***** file Clustering.h *****

```

1 #ifndef CLUSTERING_H_
2 #define CLUSTERING_H_
3
4 #include <vector>
5
6 #include "Centroid.h"
7 #include "Point.h"
8
9 class Clustering
{
10
11    typedef std::vector<unsigned> f_labels_type ;
12    typedef std::vector<Centroid> centers_type;
13    typedef std::vector<std::vector<Point *>> clusters_type;
14
15    static constexpr double MAX_COORD=1000.0;
16
17    std::vector<Point> points;
18    f_labels_type labels; //label function
19    unsigned p; //number of dimensions
20    unsigned n; //number of points
21    unsigned k; //number of clusters
22
23    centers_type centers;
24    clusters_type clusters;
```

```

25
26     unsigned max_it; //maximum number of iterations
27
28     unsigned
29         min_dist_index (const Point& point) const; //returns the index of the centroid
30                         //closest to point
31
32     void
33         print_labels (void) const;
34
35     void
36         print_centers (void) const;
37
38 public:
39     Clustering (unsigned dimensions, unsigned n_points,
40                 unsigned k_cluster, unsigned max_iterations);
41
42     void
43         print (void) const;
44
45     void
46         calc_cluster (void);
47
48 #endif /* CLUSTERING_H_ */

```

First of all, the constructor (see lines 6 to 36 of the file "Clustering.cpp") generates, in lines 10 to 25, n random points (the observations), assigning to all of them the label 0. Then, it initializes all the k centroids to the origin and it initializes all the clusters to `nullptr`.

Lines 54 to 59 of the method `calc_cluster` implement the step (1) of the algorithm described in the notes above. Indeed, they assign to any element of the vector `labels` a random value between 0 and $k - 1$ which will serve as initial cluster assignment.

Then, until the maximum number of iterations is reached or `term_cond` becomes `true`, the method updates the clusters according to the labels (namely, a point with label h will be assigned to cluster h). Note that, since the points are already stored in the corresponding vector `points`, there is no need to replicate them in the clusters: it is enough to store in the clusters their address (i.e. to define each cluster as a vector of pointers to `Point` instead of defining it as a vector of `Points`).

Then, the coordinates of the centroid of any cluster are updated through the method `Centroid::update_coord` (see lines 77 and 78).

Finally, the new labels are computed following the idea that any `Point` should be assigned to the cluster whose `Centroid` is closest (with respect to the Euclidean distance).

The variable `term_cond` becomes `true` when the new labels are equal to the old ones, i.e. when no reassignment is performed.

***** file Clustering.cpp *****

```

1 #include <iostream>
2 #include <random>
3
4 #include "Clustering.h"
5
6 Clustering::Clustering (unsigned dimensions, unsigned n_points,
7                         unsigned k_cluster, unsigned max_iterations)
8     : p (dimensions), n (n_points), k (k_cluster), max_it (max_iterations)

```

```

9  {
10    std::default_random_engine generator;
11    std::uniform_real_distribution<double> distribution (0.0, MAX_COORD);
12
13    for (unsigned i = 0; i < n; ++i)
14    {
15      //Generate n random points
16      // Generate random coordinates
17      std::vector<double> coords;
18
19      for (unsigned j = 0; j < p; ++j)
20        coords.push_back (distribution (generator));
21
22      // Generate a new Point
23      points.push_back (coords);
24      labels.push_back (0); // assign every point to class 0
25    }
26
27    std::vector<double> origin (p, 0);
28
29    // Create and intialize centroids to origin
30    for (unsigned i = 0; i < k; ++i)
31      centers.push_back (origin);
32
33    // Create and initialize clusters to nulls
34    for (unsigned i = 0; i < k; ++i)
35      clusters.push_back ({nullptr});
36  }
37
38 void
39 Clustering::print (void) const
40 {
41   // Print points and labels
42   for (std::size_t i = 0; i < points.size (); ++i)
43   {
44     points[i].print ();
45     std::cout << " : ";
46     std::cout << labels[i];
47     std::cout << std::endl;
48   }
49 }
50
51 void
52 Clustering::calc_cluster (void)
53 {
54   std::default_random_engine generator;
55   std::uniform_int_distribution<int> distribution (0, k - 1);
56
57   //Randomly initialize labels
58   for (auto it = labels.begin (); it != labels.end (); ++it)
59     *it = distribution (generator);
60

```

```

61  bool term_cond = false;
62  unsigned i;
63
64  for (i = 0; i < max_it && ! term_cond; ++i)
65  {
66      f_labels_type old_labels (labels);
67
68      clusters_type new_clusters (k);
69
70      // update clusters according to new labels
71      for (std::size_t j = 0; j < points.size (); ++j)
72          new_clusters[labels[j]].push_back (&points[j]);
73
74      clusters.swap (new_clusters);
75
76      // update centroids
77      for (std::size_t j = 0; j < centers.size (); ++j)
78          centers[j].update_coords (clusters[j]);
79
80      // assign points to new centroids
81      for (std::size_t j = 0; j < points.size (); ++j)
82      {
83          const unsigned min_dist_ind = min_dist_index (points[j]);
84          labels[j] = min_dist_ind;
85      }
86
87      term_cond = (old_labels == labels);
88  }
89
90  std::cout << "Number of iterations: " << i << std::endl;
91  std::cout << "Final result!" << std::endl;
92  print_labels ();
93  std::cout << std::endl;
94 }
95
96 unsigned
97 Clustering::min_dist_index (const Point& point) const
98 {
99     int min_dist_ind = 0;
100    double min_dist = point.distance (centers[0]);
101
102    for (std::size_t i = 1; i < centers.size (); ++i)
103    {
104        const double dist = point.distance (centers[i]);
105        if (dist < min_dist)
106        {
107            min_dist = dist;
108            min_dist_ind = i;
109        }
110    }
111
112    return min_dist_ind;

```

```

113 }
114
115 void
116 Clustering::print_labels (void) const
117 {
118     for (auto it = labels.begin (); it != labels.end (); ++it)
119     {
120         std::cout << *it << " ";
121     }
122 }
123
124 void
125 Clustering::print_centers (void) const
126 {
127     std::cout << "Centers size: ";
128     std::cout << centers.size ();
129     std::cout << std::endl;
130
131     for (std::size_t i = 0; i < centers.size (); ++i)
132         centers[i].print ();
133 }
134
135 void
136 Clustering::print_clusters (void) const
137 {
138     for (unsigned i = 0; i < k; ++i)
139     {
140         std::cout << "Cluster: " << i << std::endl;
141         for (Point* c: clusters[i])
142         {
143             c->print ();
144             std::cout << std::endl;
145         }
146     }
147 }
```

***** file KMeans.cpp *****

```

1 #include <iostream>
2 #include "Clustering.h"
3
4 int main()
5 {
6     Clustering c(2,10,2,10000);
7     c.print();
8     c.calc_cluster();
9     c.print();
10
11     return 0;
12 }
```

Additional exercise:

Extend the program so that, instead of randomly generate the points, it reads their coordi-

nates from a file.

Solution²

The only files that are modified are "Clustering.h" and "Clustering.cpp". In particular, a **private** member named **name** is added to the class **Clustering** to store the name of the file from which we read the coordinates (this value is intended to be provided within the constructor).

Moreover, we add two **public** methods:

```
bool checkfile (void) const;
```

which checks if the file can be opened and if it has the right format, and

```
std::vector<Point> readfile (void) const;
```

which reads the file and generates a vector of points with the given coordinates.

Those methods are intended to be used within the constructor (that must be modified accordingly) in order to generate the points.

The implementation of both the methods is provided below:

```
1 //checkfile function
2 bool Clustering::checkfile (void) const
3 {
4     int n_points(n);
5     bool is_correct = true;
6     std::ifstream ist(name);
7     if (!ist)
8     {
9         std::cerr << "Problem with file opening" << std::endl;
10    is_correct = false;
11 }
12
13 { //new block
14     std::string aux;
15     int i(0);
16     std::string number;
17     while(getline(ist,aux))
18     {
19         ++i;
20         int j = 0;
21         std::istringstream temp(aux); // I want to read the single line
22         while(temp >> number)      // I count the distinct entries on a line
23             ++j;
24         if (j != p)
25             std::cerr << "Wrong dimension on a line" << std::endl;
26     }
27
28     if (i != n_points)
29     {
30         std::cerr << "The file contains too many or too few elements" << std::endl;
31         is_correct = false;
32     }
33 }
34
35 return is_correct;
```

²Implementation by Nicola Ischia and Alessandra Colli

```

35 }
36
37 //Readfile function
38 std::vector<Point> Clustering::readfile (void) const
39 {
40     std::ifstream ist(name);
41     std::vector<Point> ret;
42     double val;
43
44     for(int i=0; i<n; i++)
45     {
46         std::vector<double> coords;
47         for(int i=0; i<p; i++)
48         {
49             ist >> val;
50             coords.push_back(val);
51         }
52         Point P(coords);
53         ret.push_back(P);
54     }
55     return ret;
56 }
```

In particular, the method `checkfile` (see lines 2 to 35) reads the file line by line and it checks whether the number of lines is equal to the number of points we want to generate and whether the number of entries in every line is equal to the number of coordinates that any point must have. The method `readfile` (see lines 38 to 56), in turn, opens the file and, assuming that it has the right format, it reads the values and generates the vector of points.

Note (1): since opening and reading a file is usually a demanding operation, it is always better to optimize it as much as possible, for example reading the data and checking if the format is correct all at once, thus avoiding a second pass. Think how to optimize the code above.

Note (2): the implementation is designed in order to read the coordinates from a *txt* file, where the values are separated by white spaces. If you want to read the points from, for example, a *csv* file, you have to modify the functions in order to take care of the fact that the entries are separated by either a comma or a semicolon.

3.2 Frequently Asked Questions

1. Since an **abstract** class only has **pure virtual** methods and since no objects of an **abstract** class can be instantiated, can we define a constructor inside it? And, in this case, does the class become non abstract?

ANSWER: In general, a class is **abstract** if it has at least one **pure virtual** method. This does not necessarily mean that it does not have members (see, for example, class **vehicle**, Exercise ?? in Section 3.1). Since those members may need initialization, you can always define a constructor in an **abstract** class (which remains **abstract**); this constructor will be used by the derived classes in order to initialize the parent members. The only difference with respect to non **abstract** classes is that the constructor cannot be used directly to instantiate an object.

2. When I **override** a method, do I necessarily have to call it through a reference or a pointer to the class object?

ANSWER: To override a method means that, in the derived class, you want a different behaviour with respect to the one you have in the base class.

You can call the method either through a reference (or pointer) or directly through an object, depending on what you want to do: if you do not use neither a reference nor a pointer, you rely on static binding instead of dynamic binding, which means that you can only access the implementation of the class from which you are calling the method, not the implementation of the derived classes. As an example, suppose you write the following program:

```
1 class Father {
2     protected:
3         double x;
4     public:
5         Father (double d): x(d) {}
6         virtual void print (void) const {std::cout << "x is: " << x << std::endl;}
7     };
8
9 class Son: public Father {
10    protected:
11        double y;
12    public:
13        Son (double d1, double d2): Father(d1), y(d2) {}
14        void print (void) const override {std::cout << "x is: " << x
15                                << " y is: " << y << std::endl;}
16    };
17
18 int main (void)
19 {
20     Father f(3.3);
21     f.print();
22
23     Son s(4.4,5.5);
24     s.print();
25
26     Father * f2 = &s;
27     f2->print();
28 }
```

You will get the output:

```
x is : 3.3
x is : 4.4 y is : 5.5
x is : 4.4 y is : 5.5
```

Indeed, in line 21 you call the method `print` through an object of type `Father`, which means you print only the value of `x`. In line 24, you call the method `print` through an object of type `Son`, which means you print both the value of `x` and the value of `y`. Finally, in line 26 you initialize a pointer to `Father` type with the address of a `Son` object. Therefore, when, in line 27, you call the method `print` through the pointer, the compiler relies on dynamic binding to call not the method declared in `Father` class, but the one overridden in `Son` class.

You can find other examples of dynamic binding in Exercise ?? of Section 3.1.

3. When I want to override a method from a based to a derived class, do I have to explicitly write `virtual` and `override`?

ANSWER: When you want to override a method from a base class to a derived one, you must write **virtual** in the base class.

A **virtual** method in the base class is automatically **virtual** in the derived one, then you can either specify **virtual** or not in the derived class, as you prefer.

Moreover, it is not compulsory to write **override** in the declaration of the method in the derived class, but it is better to do it, so that the compiler will check that you are overriding the function and not redefining another one with different parameters (which then is overloading and does not lead to dynamic binding). An example of the difference between overriding and overloading a method can be found in class **Robust** of Exercise 2 (Section 3.1).

4. Supposing that I have two classes, namely

```
1 class Base {  
2     protected:  
3         double x;  
4     public:  
5         Base (double d): x(d) {}  
6     };  
7  
8 class Derived: public Base {  
9     protected:  
10        double y;  
11     public:  
12        Derived (double d1, double d2): x(d1), y(d2) {}  
13    };
```

Since **x** is a **protected** member of **Base** and thus it is inherited and visible by **Derived**, why, if I try to initialize it in the constructor of **Derived**, I get the compilation error:

```
In constructor 'Derived::Derived(double, double)':  
error: class 'Derived' does not have any field named 'x'
```

ANSWER: When you want to initialize a member of a parent class within the derived one, you must rely on the delegating constructor, which means that you must call the constructor of the class **Base** from the one of the class **Derived**. In your example, you must replace line 12 with:

```
Derived (double d1, double d2): Base(d1), y(d2) {}
```

Chapter 4

Copy Control

4.1 Exercises

Exercise 1 Taxi

The management system of a taxi-transportation company, represented by the class `Agency`, includes classes to keep track of drivers, taxis and receipts of travels.

Every `Taxi` is characterized by an identification number and by the zone where it can work (both implemented as `const std::strings`). Moreover, it keeps track of its current `Driver` and stores in a `vector` the `Receipts` related to the travels of the current week.

Consider that, since each taxi can operate in a single zone, in order to be sure that the amount of work is fairly divided among drivers, the agency randomly assigns them a different taxi every day.

Starting from the code excerpt provided below, you have to **complete the implementation of `Taxi` and `Agency` classes** in order to **minimize the memory usage**. In particular, you have to:

1. Identify the best type to store the information about the current `Driver` in the class `Taxi`, in order to minimize the memory consumption, and complete the declaration of method `Taxi::assign_driver` accordingly.
2. Identify the best type to store the set of all `Drivers` in the class `Agency`, in order to minimize the memory consumption, and complete the implementation of the constructor accordingly.
3. Provide the implementation of the method `Agency::perform_assignment` that assigns the taxis to the available drivers. Consider that:
 - If the number of drivers is not enough to assign all taxis, the method should not perform any assignment, but it should give an error message.
 - The index of the first driver who should receive a taxi is extracted at random through the following procedure:

```
std::uniform_int_distribution<> distribution(0, drivers.size() - 1);
unsigned initial_idx = distribution(generator);
```

Note that the class `Date` is not provided, since a possible implementation can be found in Exercise 2.3 of Section 2.3.

***** file `Receipt.hpp` *****

```

1 #ifndef RECEIPT_HH
2 #define RECEIPT_HH
3
4 #include "Date.hpp"
5
6 class Receipt {
7
8     std::string address;
9     Date date;
10    double price;
11
12 public:
13     Receipt (const std::string & a, const Date &d, double p):
14         address (a), date(d), price (p) {}
15
16 };
17
18 #endif /* RECEIPT_HH */

```

***** file Driver.hpp *****

```

1 #ifndef DRIVER_HH
2 #define DRIVER_HH
3
4 #include <string>
5
6 class Driver {
7
8     std::string name;
9     std::string surname;
10    std::size_t id;
11
12 public:
13     Driver (const std::string & n, const std::string & s, std::size_t i):
14         name (n), surname (s), id (i) {}
15
16     std::string to_string (void) const;
17 };
18
19 #endif /* DRIVER_HH */

```

***** file Taxi.hpp *****

```

1 #ifndef TAXI_HH
2 #define TAXI_HH
3
4 #include <vector>
5 #include <memory>
6
7 #include "Receipt.hpp"
8 #include "Driver.hpp"
9
10 class Taxi {

```

```

11 // identification number and zone
12 const std::string id;
13 const std::string zone;
14
15 // current driver
16 /* YOUR CODE GOES HERE */ driver;
17
18 // the receipts of the current week
19 std::vector<Receipt> receipts;
20
21 public:
22     // constructor
23     Taxi (const std::string& i, const std::string& z): id(i), zone(z) {}
24
25     void assign_driver /* YOUR CODE GOES HERE */ (d) {driver = d;}
26     void save_receipt (const Receipt& r) {receipts.push_back(r);}
27
28     std::string to_string (void) const;
29 };
30
31 #endif /* TAXI_HH */

```

***** file Agency.hpp *****

```

1 #ifndef AGENCY_HH
2 #define AGENCY_HH
3
4 #include "Taxi.hpp"
5
6 #include <random>
7 #include <iostream>
8
9 class Agency {
10
11 private:
12     std::vector</* YOUR CODE GOES HERE */> drivers;
13     std::vector<Taxi> taxis;
14
15     std::default_random_engine generator;
16
17 public:
18     Agency /* YOUR CODE GOES HERE */(, const std::vector<Taxi>&);
19
20     void perform_assignment (void);
21
22     void print (void) const;
23 };
24
25 #endif //TAXISYSTEM_AGENCY_H

```

***** file Agency.cpp *****

```

1 #include "Agency.hpp"

```

```

2
3 Agency::Agency /* YOUR CODE GOES HERE */ (ds, const std::vector<Taxi>& ts):
4     drivers(ds), taxis(ts)
5 }
6
7 void Agency::perform_assignment (void)
8 {
9     /* YOUR CODE GOES HERE */
10}
11
12 void Agency::print (void) const
13 {
14     for (const Taxi& t : taxis)
15         std::cout << t.to_string() << std::endl;
16 }
```

Exercise 1 - Solution

In order to reduce the memory requirements of the taxi-transportation company management program, we rely on shared pointers. In particular, each `Taxi` is designed to store a `shared_ptr` to its current driver, while also the class `Agency` stores a vector of `shared_ptrs` to `Drivers`.

Given this structure, the implementation of the method `Agency::perform_assignment` proceeds as reported in file "Agency.cpp": if the number of drivers is greater than or equal to the number of taxis, the index of the first driver is selected at random in lines 11 and 12. Then, we loop over all taxis and we assign each taxi to a driver, starting from the one in position `initial_idx` and going back to 0 when we reach the end of the vector.

***** file `Taxi.hpp` *****

```

1 #ifndef TAXI_HPP
2 #define TAXI_HPP
3
4 #include <vector>
5 #include <memory>
6
7 #include "Receipt.hpp"
8 #include "Driver.hpp"
9
10 class Taxi {
11     // identification number and zone
12     const std::string id;
13     const std::string zone;
14
15     // pointer to the current driver
16     std::shared_ptr<Driver> driver;
17
18     // the receipts of the current week
19     std::vector<Receipt> receipts;
20
21 public:
22     // constructor
23     Taxi (const std::string& i, const std::string& z): id(i), zone(z) {}
```

```

24
25     void assign_driver (const std::shared_ptr<Driver>& d) {driver = d;}
26     void save_receipt (const Receipt& r) {receipts.push_back(r);}
27
28     std::string to_string (void) const;
29 };
30
31 #endif /* TAXI_HH */

```

***** file Taxi.cpp *****

```

1 #include "Taxi.hpp"
2
3 std::string Taxi::to_string (void) const
4 {
5     std::string line1 = "Taxi " + id + "(" + zone + ")\n";
6
7     std::string line2;
8     if (driver)
9         line2 = "\tDriver: " + driver->to_string() + "\n";
10    else
11        line2 = "\tNot assigned\n";
12
13    return line1 + line2;
14 }

```

***** file Agency.hpp *****

```

1 #ifndef AGENCY_HH
2 #define AGENCY_HH
3
4 #include "Taxi.hpp"
5
6 #include <random>
7 #include <iostream>
8
9 class Agency {
10
11 private:
12     std::vector<std::shared_ptr<Driver>> drivers;
13     std::vector<Taxi> taxis;
14
15     std::default_random_engine generator;
16
17 public:
18     Agency (const std::vector<std::shared_ptr<Driver>>&, const std::vector<Taxi>&);
19
20     void perform_assignment (void);
21
22     void print (void) const;
23 };
24
25 #endif //TAXISYSTEM_AGENCY_H

```

```
***** file Agency.cpp *****
```

```
1 #include "Agency.hpp"
2
3 Agency::Agency (const std::vector<std::shared_ptr<Driver>>& ds, const std::vector<
4     Taxi>& ts):
5     drivers(ds), taxis(ts)
6 {}
7
8 void Agency::perform_assignment (void)
9 {
10     if (taxis.size() <= drivers.size())
11     {
12         std::uniform_int_distribution<> distribution(0, drivers.size() - 1);
13         unsigned initial_idx = distribution(generator);
14
15         unsigned idx = initial_idx;
16         for (Taxi &t : taxis) {
17             if (idx == drivers.size())
18                 idx = 0;
19
20             t.assign_driver(drivers[idx]);
21             ++idx;
22         }
23     }
24     else
25         std::cerr << "ERROR: not enough drivers to perform the assignment" << std::endl;
26 }
27
28 void Agency::print (void) const
29 {
30     for (const Taxi& t : taxis)
31         std::cout << t.to_string() << std::endl;
```

Exercise 2 Boxes

Starting from the code provided below, complete it so that the program keeps count of all the Box objects created and it outputs this information before exiting.

```
***** file Box.hpp *****
```

```
1 #ifndef BOX_hpp
2 #define BOX_hpp
3
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 class Box
10 {
```

```

11 public:
12
13     Box(double l, double b, double h): length(l),breadth(b),height(h)
14     {
15         cout <<"Constructing a box" << endl;
16     }
17
18     double Volume() const;
19
20 private:
21     double length; // Length of a box
22     double breadth; // Breadth of a box
23     double height; // Height of a box
24 };
25
26 #endif

```

***** file Box.cpp *****

```

1 #include "Box.hpp"
2
3 Box::double Volume() const
4 {
5     return length * breadth * height;
6 }

```

Exercise 2 - Solution

In order to make the `Box` class able to keep track of the number of boxes that are created, we use a `static` counter. The reason why we make it `static` is that the counter has to be related not to a single `Box` object, but to the class itself, i.e. it stores an information that must be shared by all the objects of that type.

In particular, we declare `count` as a `private static` member, so that its value cannot be directly modified by the user. For this reason, a `public` method, namely

`static unsigned getcount();`

(see line 29 of the file "Box.hpp") is implemented in order to make the value of `count` readable by the user, but it is not provided any method to set a different value for `count`.

Notice that, first of all, the method `getcount` is declared as `static`. This, in principle, is not mandatory, but it is more reasonable. Indeed, since `count` stores an information which is not related to a specific `Box` object, it is better to allow the user to read this information without passing through an explicit instance of the class. In particular, if the method `getcount` were not `static`, the user would be forced to write, for example:

```

Box b(7,7,7);
unsigned c = b.getcount();

```

explicitly calling `getcount` through `b`. This may even lead to an error: if no objects of the class `Box` have been created, the user is forced to create one only to call the method `getcount`, while maybe he only wants to read this information (getting, in this case, 0), not to generate a new `Box`.

On the other hand, if the method `getcount` is `static`, the user can call it without passing through a `Box` object, writing:

```

unsigned c = Box::getcount();

```

even if no objects of type `Box` have been created (in this case, he will get, of course, 0). Moreover, notice that the method `getcount`, since it is `static`, has no `const` qualifier. A `static` method cannot be declared as `const` because, since it is not called on a particular instance of the class, but it is generally related to the class itself, it does not have a pointer to `this`, thus the `const` qualifier turns out to be meaningless.

Note: the counter implemented in this exercise keeps track of all the `Boxes` that are created during the lifetime of the program, even if, at a certain moment, they go out of scope and they are deleted. Because of this, some strange situations might occur, where the number registered in `count` is completely different from the one we expect (see Question 2 in section 4.2).

Other than the general counter required by the exercise, the `Box` class provides any object with an identification number (see line 39 of file "Box.hpp"), which is automatically initialized when the object is created after having incremented the current `count` number (see what happens in the constructor, in lines 12 to 17, and in the copy constructor, in lines 20 to 25). The `Box` class, even if it does not implement explicitly a destructor because, since it does not manage the memory dynamically, it can safely rely on the synthesized one, needs both a copy constructor and an assignment operator. This is common in classes that have a non constant `static` member, because its value cannot be trivially copied as it would happen if we rely on the automatically synthesized copy constructor or assignment operator. Indeed, if we rely, for example, on the automatic copy constructor, the following happens: suppose that the user has initialized three objects of type `Box` using the general constructor of line 12 and then suppose that he writes:

```
Box b4(b1);
```

using the copy constructor. If, later on, he asks the program to print out the identification number of the fourth box and the total number of boxes:

```
std::cout << "box id: " << b4.getid() << std::endl;
std::cout << "total number of boxes: " << Box::getcount() <<
std::endl;
```

he receives as output:

```
box id: 3
total number of boxes: 3
```

even if the actual number of boxes is 4 (and the id of the last box should be 4 as well). Indeed, the values of `count` and `id` are copied from `b1` to `b4` by the copy constructor, while they would have to be incremented because, actually, a new object is created. A similar problem happens with the assignment operator, where the `id` of the right hand side is copied in the left hand side, while we want the lhs to get all the characteristic of the rhs, namely `length`, `breadth` and `height`, but to keep its original `id`.

To sum up, the copy constructor implemented in lines 20 to 25 of file "Box.hpp" generates a new, distinct serial number for the object being created, while it copies all the other data members from the given object. On the other hand, the assignment operator (see lines 14 to 21 of file "Box.cpp") copies all the information from the right hand side, but it does not modify the identification number of the left hand side. The idea is: we use the assignment operator when we want to modify an existing object so that it gets the characteristics of the right hand side (namely `length`, `breadth` and `height`), but it keeps its original identification number. A possible alternative would be to implement the assignment operator as the copy constructor, i.e. to increment the counter and to give to the left hand side a new identification number.

***** file Box.hpp *****

```

1 #ifndef BOX_hpp
2 #define BOX_hpp
3
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 class Box
9 {
10 public:
11
12     Box(double l, double b, double h): length(l),breadth(b),height(h)
13     {
14         cout <<"Constructing a box" << endl;
15         count++;
16         id = count;
17     }
18
19 // copy constructor
20 Box(const Box & b): length(b.length),breadth(b.breadth),height(b.height)
21 {
22     cout <<"Using copy constructor" << endl;
23     count++;
24     id = count;
25 }
26
27 // member functions
28 double Volume() const;
29 static unsigned getcount() {return count;}
30 unsigned getid() const {return id;}
31
32 // assignement operator
33 Box& operator=(const Box & z);
34
35 private:
36     double length; // Length of a box
37     double breadth; // Breadth of a box
38     double height; // Height of a box
39     unsigned id; // Identification Number of the box
40     static unsigned count; // Number of boxes
41 };
42
43 #endif

```

Note that the **static** member `count` is initialized not in the header file, but in the source file (line 4 of "Box.cpp"). This is done because any **static** member must be initialized only once, while, if we write the initialization in the header file, we end up having a multiple definition of the same data member every time we include this header file. In particular, the initialization of a **static** data member can be done in-class if it is a constant integral type (thus, for example, a **const int** or a **const char**), while it must be performed in-class (and it cannot be written in the source file) if it is a **constexpr**, which is always evaluated at compile time.

***** file Box.cpp *****

```
1 #include "Box.hpp"
2
3 // initialization of the static variable
4 unsigned Box::count = 0;
5
6 // member functions:
7
8 double Box::Volume() const
9 {
10     return length * breadth * height;
11 }
12
13 // assignment operator
14 Box& Box::operator=(const Box &b)
15 {
16     std::cout << "Using assignment operator" << std::endl;
17     length = b.length;
18     breadth = b.breadth;
19     height = b.height;
20
21     return *this;
22 }
```

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

```
1 #include "Box.hpp"
2 #include <vector>
3 int main(void)
4 {
5     // Initialize a couple of Box objects
6
7     Box b1(10,10,10);
8     Box b2(10,20,30);
9     Box b3(5,5,5);
10    Box b4(1,3,6);
11    Box b5(b4);
12
13    cout << endl;
14    double v1 = b1.Volume();
15    cout << "volume of box 1 (10x10x10) is: " << v1 << endl;
16    cout << "box id: " << b1.getid() << endl;
17    cout << endl;
18
19    double v2 = b2.Volume();
20    cout << "volume of box 2 (10x20x30) is: " << v2 << endl;
21    cout << "box id: " << b2.getid() << endl;
22    cout << endl;
23
24    double v3 = b3.Volume();
25    cout << "volume of box 3 (5x5x5) is: " << v3 << endl;
26    cout << "box id: " << b3.getid() << endl;
```

```

27 cout << endl;
28
29 double v4 = b4.Volume();
30 cout << "volume of box 4 (1x3x6) is: " << v4 << endl;
31 cout << "box id: " << b4.getid() << endl;
32 cout << endl;
33
34 double v5 = b5.Volume();
35 cout << "box 5 is equal to box 4" << endl;
36 cout << "volume of box 5 is: " << v5 << endl;
37 cout << "box id: " << b5.getid() << endl;
38 cout << endl;
39
40 cout << "now box 5 is equal to box 3" << endl;
41 b5 = b3;
42 double v6 = b5.Volume();
43 cout << "new volume of box 5 is: " << v6 << endl;
44 cout << "new box id: " << b5.getid() << endl;
45 cout << endl;
46
47 // Print total number of objects bellow
48
49 unsigned tot_count = Box::getcount();
50 cout << "the total number of boxes is: " << tot_count << endl;
51
52 return 0;
53 }
```

Exercise 3 Calendar

The goal of the source code provided is to implement a simple calendar. Class `Calendar` stores `Events` that are characterized by a name (implemented as a `std::string`) and a date (implemented as `time_t`). Stored events can be visualized through method `print` of the class `Calendar`. The header and implementation files of `Events` are provided, while you are asked to complete the header and the implementation of the class `Calendar` in order to obtain a like-a-pointer behavior.

After carefully reading the code, you have to:

1. Complete the implementation of `Calendar` in order to implement a *like-a-pointer* behavior.
2. List and motivate the program **output** considering the main file provided.

***** file `Event.hpp` *****

```

1 #ifndef EVENT_H
2 #define EVENT_H
3
4 #include <string>
5
6 class Event {
```

```

8 private:
9     time_t date;
10    std::string name;
11
12 public:
13     Event(time_t d, const std::string& n): date(d), name(n) {};
14
15     time_t getTime (void) const;
16     std::string getName (void) const;
17 };
18
19 #endif // EVENT_H

```

***** file Event.cpp *****

```

1 #include "Event.hpp"
2
3 time_t Event::getTime (void) const
4 {
5     return date;
6 }
7
8 std::string Event::getName (void) const
9 {
10    return name;
11 }

```

***** file Calendar.hpp *****

```

1 #ifndef CALENDAR_H
2 #define CALENDAR_H
3
4 #include <iostream>
5 #include <vector>
6
7 #include "Event.hpp"
8
9 class Calendar {
10
11 private:
12     /* YOUR CODE GOES HERE */ events;
13
14 public:
15     /* YOUR CODE GOES HERE */
16     void addEvent (const Event&);
17     void print (void) const;
18 };
19
20 #endif // CALENDAR_H

```

***** file Calendar.cpp *****

```

1 #include "Calendar.hpp"

```

```

2
3 void Calendar::addEvent (const Event &event)
4 {
5     /* YOUR CODE GOES HERE */
6 }
7
8 void Calendar::print() const
9 {
10    for (const Event &e : /* YOUR CODE GOES HERE */)
11        std::cout << e.getName() << std::endl;
12 }
```

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

```

1 #include "Calendar.hpp"
2
3 using std::cout;
4 using std::endl;
5
6 int main()
7 {
8     Calendar c0;
9     Event e0(time(0), "Important Meeting");
10    c0.addEvent(e0);
11
12    Calendar c1;
13    Event e1(time(0), "Andrew's Birthday");
14    c1.addEvent(e1);
15    c1 = c0;
16
17    Calendar c2(c0);
18    Event e2(time(0), "Trip to Cairo");
19    c2.addEvent(e2);
20
21    c2 = Calendar();
22    Event e3(time(0), "Visit to Museum");
23    c2.addEvent(e3);
24
25    cout << "c0:" << endl;
26    c0.print();
27    cout << "c1:" << endl;
28    c1.print();
29    cout << "c2:" << endl;
30    c2.print();
31
32    return 0;
33 }
```

Exercise 3 - Solution

In order to implement a like-a-pointer behavior, we define `events` in the class `Calendar` as a shared pointer to a vector of `Events`. Thanks to this, the events stored in a calendar will be shared among all calendars obtained as copies of the first one.

We define only the constructor with no parameters, that creates a new empty object of type `std::vector<Event>` through the method `std::make_shared`. We implement moreover the method `addEvent`, by using the arrow operator to access to the method `push_back` of `std::vector` class.

***** file `Calendar.hpp` *****

```

1 #ifndef CALENDAR_H
2 #define CALENDAR_H
3
4 #include <iostream>
5 #include <vector>
6 #include <memory>
7
8 #include "Event.hpp"
9
10 class Calendar {
11
12 private:
13     std::shared_ptr<std::vector<Event>> events;
14
15 public:
16     Calendar (void): events(std::make_shared<std::vector<Event>>()) {}
17     void addEvent (const Event&e);
18     void print (void) const;
19 };
20
21 #endif // CALENDAR_H

```

***** file `Calendar.cpp` *****

```

1 #include "Calendar.hpp"
2
3 void Calendar::addEvent (const Event &event)
4 {
5     events->push_back(event);
6 }
7
8 void Calendar::print() const
9 {
10     for (const Event &e : *events)
11         std::cout << e.getName() << std::endl;
12 }

```

For what concerns the output of the program, we can notice that:

- `c0` and `c1` share the same data because of the assignment `c1 = c0`.
- `c2` adds an event to the data originally pointed by `c0` since it is created using the copy constructor.
- After adding `e2`, `c2` is reinitialized and starts pointing to different data locations.

Therefore, the output of the program is:

```

c0:
Important Meeting
Trip to Cairo
c1:
Important Meeting
Trip to Cairo
c2:
Visit to Museum

```

Exercise 4 Document Store

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 vectors of `Documents`: one for completed documents (instance variable `docs`) and one for drafts (instance variable `docsDraft`). The size of the vector `docs` is set in the `DocumentStore` constructor and can be configured by the users. The size of `docsDraft`, defined as `const DRAFT_SIZE`, 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. 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 `DocumentStores`, 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.

***** file `Document.hpp` *****

```

1 #ifndef DOCUMENT_H
2 #define DOCUMENT_H
3
4 #include <string>
5 #include <iostream>
6
7 class Document {
8
9 private:
10     std::string text = "";
11     std::size_t id = 0;
12
13 public:
14     Document () = default;
15     Document (const std::string& text, std::size_t id): text(text), id(id) {}
16
17     const std::string& getText () const;

```

```

18     std::size_t getId () const;
19
20     void print () const;
21 };
22
23 #endif /* DOCUMENT_H */

```

***** file Document.cpp *****

```

1 #include "Document.hpp"
2
3 const std::string& Document::getText () const
4 {
5     return text;
6 }
7
8 std::size_t Document::getId () const
9 {
10    return id;
11 }
12
13 void Document::print () const
14 {
15     std::cout << "id: " << id << "\n\ttext: " << text << std::endl;
16 }

```

***** file DocumentStore.hpp *****

```

1 #ifndef DOCUMENTSTORE_H
2 #define DOCUMENTSTORE_H
3
4 #include <vector>
5
6 #include "Document.hpp"
7
8 const unsigned DRAFT_SIZE = 10;
9
10 class DocumentStore {
11
12 private:
13     /* YOUR CODE GOES HERE */ docs;
14     /* YOUR CODE GOES HERE */ docsDraft;
15     std::size_t size;
16     std::size_t curr;
17     std::size_t currDraft;
18
19 public:
20     explicit DocumentStore (std::size_t);
21
22     /* YOUR CODE GOES HERE */
23
24     void addDocument (const Document&);
25     void saveAsDraft (const Document&);

```

```

26     void print () const;
27 };
28
29 #endif /* DOCUMENTSTORE_H */

```

***** file DocumentStore.cpp *****

```

1 #include "DocumentStore.hpp"
2
3 DocumentStore::DocumentStore (std::size_t s):
4     /* YOUR CODE GOES HERE */, size(s), curr(0), currDraft(0)
5 {
6
7     /* YOUR CODE GOES HERE */
8
9     void DocumentStore::addDocument (const Document& doc)
10    {
11        if (curr < size)
12            docs[curr++] = doc;
13    }
14
15    void DocumentStore::saveAsDraft (const Document& draft)
16    {
17        if (currDraft < DRAFT_SIZE)
18            docsDraft[currDraft++] = draft;
19    }
20
21    void DocumentStore::print () const
22    {
23        std::cout << "List of Documents:" << std::endl;
24        for (std::size_t j = 0; j < curr; ++j)
25            docs[j].print();
26
27        std::cout << "List of Drafts:" << std::endl;
28        for (std::size_t j = 0; j < currDraft; ++j)
29            docsDraft[j].print();
30    }

```

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

```

1 #include "DocumentStore.hpp"
2
3 int main()
4 {
5     std::size_t id = 945;
6     Document d0("Apple", id++);
7     Document d1("Orange", id++);
8     Document d2("Melon", id++);
9     Document d3("Peach", id++);
10    Document d4("Strawberry", id++);
11
12    DocumentStore ds0(3);
13    ds0.addDocument(d0);

```

```

14     ds0.saveAsDraft(d1);
15
16     DocumentStore ds1(ds0);
17     ds1.addDocument(d2);
18     ds1.addDocument(d3);
19
20     DocumentStore ds2(2);
21     ds2.addDocument(d1);
22     ds2.addDocument(d4);
23     ds2.saveAsDraft(d2);
24     ds2.saveAsDraft(d3);
25
26     DocumentStore ds3(3);
27     ds2.addDocument(d0);
28     ds2.addDocument(d1);
29     ds3 = ds2;
30     ds3.addDocument(d3);
31
32     std::cout << "----- ds0 -----" << std::endl;
33     ds0.print();
34
35     std::cout << "\n----- ds1 -----" << std::endl;
36     ds1.print();
37
38     std::cout << "\n----- ds2 -----" << std::endl;
39     ds2.print();
40
41     std::cout << "\n----- ds3 -----" << std::endl;
42     ds3.print();
43
44     return 0;
45 }
```

Exercise 4 - Solution

In order to implement a like-a-value behavior for the `DocumentStore` class, we define both `docs` and `docsDraft` as vectors of `Documents` (see lines 13 and 14 of file "DocumentStore.hpp"). The two vectors will be initialized, in the body of the constructor implemented at lines 3 to 5 of file "DocumentStore.cpp", through the given size and the constant `DRAFT_SIZE`, respectively.

The information that, when copying `DocumentStores` objects, only the complete documents should be copied, while drafts should be neglected, dictates the need of defining a suitable copy constructor and assignment operator. These are defined in lines 7 to 19 of file "DocumentStore.cpp". It is easy to notice that all fields concerning documents (namely `docs`, `curr` and `size`) are copied, while `currDraft` is initialized by 0. Notice that, since we do not want to copy drafts and since the size of the vector `docsDraft` is fixed, in the implementation of `operator=` there is no need to modify its content: by setting `currDraft` to 0, its old value will be replaced when calling the method `saveAsDraft`. Of course, the vector `docsDraft` must instead be initialized in the copy constructor, since otherwise it is default initialized as an empty vector, leading the method `saveAsDraft` to an undefined behavior.

***** file DocumentStore.hpp *****

```

1 #ifndef DOCUMENTSTORE_H
2 #define DOCUMENTSTORE_H
3
4 #include <vector>
5
6 #include "Document.hpp"
7
8 const unsigned DRAFT_SIZE = 10;
9
10 class DocumentStore {
11
12 private:
13     std::vector<Document> docs;
14     std::vector<Document> docsDraft;
15     std::size_t size;
16     std::size_t curr;
17     std::size_t currDraft;
18
19 public:
20     explicit DocumentStore (std::size_t);
21     DocumentStore (const DocumentStore&);

22     DocumentStore& operator= (const DocumentStore&);

23     void addDocument (const Document&);
24     void saveAsDraft (const Document&);
25     void print () const;
26 };
27
28 #endif /* DOCUMENTSTORE_H */

```

***** file DocumentStore.cpp *****

```

1 #include "DocumentStore.hpp"
2
3 DocumentStore::DocumentStore (std::size_t s):
4     docs(s), docsDraft(DRAFT_SIZE), size(s), curr(0), currDraft(0)
5 {}
6
7 DocumentStore::DocumentStore (const DocumentStore& rhs):
8     docs(rhs.docs), docsDraft(DRAFT_SIZE), size(rhs.size), curr(rhs.curr), currDraft(0)
9 {}
10
11 DocumentStore& DocumentStore::operator= (const DocumentStore& rhs)
12 {
13     docs = rhs.docs;
14     size = rhs.size;
15     curr = rhs.curr;
16     currDraft = 0;
17     return *this;
18 }
19
20 void DocumentStore::addDocument (const Document& doc)

```

```

21 {
22     if (curr < size)
23         docs[curr++] = doc;
24 }
25
26 void DocumentStore::saveAsDraft (const Document& draft)
27 {
28     if (currDraft < DRAFT_SIZE)
29         docsDraft[currDraft++] = draft;
30 }
31
32 void DocumentStore::print () const
33 {
34     std::cout << "List of Documents:" << std::endl;
35     for (std::size_t j = 0; j < curr; ++j)
36         docs[j].print();
37
38     std::cout << "List of Drafts:" << std::endl;
39     for (std::size_t j = 0; j < currDraft; ++j)
40         docsDraft[j].print();
41 }

```

Given the provided implementation, the content of the vectors `docs` and `docsDraft` is not shared among different `DocumentStore` objects. Therefore, after having created `ds1` using `ds0`, the vectors `docs` of the two instances contain the same documents but they are completely separated in memory. In fact, after adding new documents to `ds1`, no changes are made to `ds0`.

Moreover, since the program is designed not to add documents after the maximum size defined in the constructor is reached, the two instructions in lines 27 and 28 of file "main.cpp" have no effect, because two documents are already stored in `ds2`. For the same reason, since we assign to `ds3` the value of `ds2` (and therefore we reduce its maximum size to 2), also the instruction in line 30 is neglected. The output of the given program is therefore:

```

----- ds0 -----
List of Documents:
id: 945
    text: Apple
List of Drafts:
id: 946
    text: Orange

----- ds1 -----
List of Documents:
id: 945
    text: Apple
id: 947
    text: Melon
id: 948
    text: Peach
List of Drafts:

----- ds2 -----
List of Documents:

```

```

id: 946
    text: Orange
id: 949
    text: Strawberry
List of Drafts:
id: 947
    text: Melon
id: 948
    text: Peach

----- ds3 -----
List of Documents:
id: 946
    text: Orange
id: 949
    text: Strawberry
List of Drafts:

```

4.2 Frequently Asked Questions

1. Is there any rule to decide whether to overload an operator as a method of the class or as a free function?

ANSWER: Some of the operators, namely assignment (`=`), call (`()`), subscript (`[]`) and arrow (`->`), must be members of the class. For the others, the choice is left to implementors' sensibility, with some reasonable rules of thumb:

- Compound assignment operators (`+=`, `*=`, and the like) are generally implemented as members because it seems logical, being graphically similar to plain assignment;
- Stream insertion and extraction (`<<` and `>>`) have to be nonmember, because the object from the user defined class has to be the second parameter;
- Increment and decrement (`++` and `--`, both prefix and postfix) generally are members, since they modify the state;
- Arithmetic (`+`, `-`, both unary and binary, `*`, `/`, `%`) are usually nonmember to be robust to implicit conversions. Moreover, generally they are just a one-liner that calls the corresponding compound assignment, in order to avoid code duplication (see for example the alternative implementation of `operator +` in Exercise ?? of Section 4.1);
- Relational (`<`, `==`, etc) are generally nonmember for similar considerations.

2. Why, if I use the code written in Exercise 2 and I create a `std::vector<Box>` through an initializer list, namely I write:

```

Box b1(10,10,10);
Box b2(10,20,30);
Box b3(5,5,5);
Box b4(1,3,6);
Box b5(b4);
std::vector<Box> bv = {b1,b2,b3,b4,b5};
unsigned tot_count = Box::getcount();

```

I end up having `tot_count` equal to 15 instead of 5?

ANSWER: This happens because, when you use an initializer list, you need, first of all, to construct the list in contiguous memory, then you use it as argument for the non-explicit `std::vector<Box>` constructor that converts from `std::initializer_list<Box>`. After constructing `bv`, the braced list is destroyed, since it is a temporary whose extent is only the expression.

You get a similar behaviour also if you initialize an empty vector and then you add the Boxes through `push_back`. In particular, if you write:

```
std::vector<Box> bv;
bv.reserve(5);
bv.push_back(b1);
bv.push_back(b2);
bv.push_back(b3);
bv.push_back(b4);
bv.push_back(b5);
unsigned tot_count = Box::getcount();
```

you get `tot_count` equal to 10. Indeed, any time you call `push_back` you create a copy of the object passed as parameter and you plug it in the vector `bv` (only one copy and not two as it happens with the initializer list because the argument is passed to `push_back` as a constant reference, thus avoiding copies).

Notice that if you do not specify in advance the space that you will need in your vector, i.e. you do not call `bv.reserve(5)` as we did above, the number of copies might increase due to possible reallocations of the vector (for a further discussion about reallocations, see).

Chapter 5

Smart Pointers

5.1 Exercises

Exercise 1 TripAdvisor[®] (*exam 17/02/2017*)

Implement a program that, similarly to TripAdvisor[®], lets users share their comments about different places. You need to define three classes, namely `user`, `place` and `comment`, such that:

- Each user has a profile that shows name (`string`), surname (`string`) and comments about different places.
- Each place has a profile as well, that shows its name (`string`), description (`string`) and comments that different users wrote about it.
- Each comment has a text.

Hint: users and places have access to their corresponding comments through shared pointers, so that only one copy of each is stored in memory.

In addition, implement the following method in the `user` class to write comments about a place:

```
void leave_a_comment (const string&, place &);
```

Exercise 1 - Solution

```
***** file comment.h *****
```

```
1 #ifndef COMMENT_HH
2 #define COMMENT_HH
3
4 #include <string>
5
6 class comment
7 {
8     const std::string text;
9     public:
10    comment (const std::string & txt): text (txt) {}
11}
```

```

12     const std::string & get_text (void) const;
13 }
14
15 #endif

```

***** file comment.cpp *****

```

1 #include "comment.h"
2
3 const std::string & comment::get_text (void) const
4 {
5     return text;
6 }

```

***** file place.h *****

```

1 #ifndef PLACE_HH
2 #define PLACE_HH
3
4 #include <vector>
5 #include <memory>
6
7 #include "comment.h"
8
9 class place
10 {
11     const std::string name;
12     const std::string description;
13     std::vector<std::shared_ptr<comment>> comments;
14
15 public:
16     place (std::string const & n, std::string const & desc)
17         : name (n), description (desc) {}
18
19     const std::vector<std::shared_ptr<comment>> & get_comments (void) const;
20
21     void add_comment (std::shared_ptr<comment> cmnt);
22 };
23
24 #endif

```

The class `place` stores a `std::vector` of shared pointers to `comments`, which represents the list of all the comments relative to `this`. It implements a method which receives a `std::shared_ptr<comment>` and adds it to the vector `comments`.

***** file place.cpp *****

```

1 #include "place.h"
2
3 const std::vector<std::shared_ptr<comment>> & place::get_comments (void) const
4 {
5     return comments;
6 }
7

```

```

8 void place::add_comment (std::shared_ptr<comment> cmnt)
9 {
10     comments.push_back (cmnt);
11 }

```

***** file user.h *****

```

1 #ifndef USER_HH
2 #define USER_HH
3
4 #include "comment.h"
5 #include "place.h"
6
7 class user
8 {
9     const std::string name;
10    const std::string surname;
11    std::vector<std::shared_ptr<comment>> comments;
12
13 public:
14     user (const std::string &n, const std::string &s):
15         name(n), surname(s) {}
16
17     void leave_a_comment (const std::string&, place &);
18 };
19
20 #endif

```

The class `user` stores, exactly as `place`, a `std::vector` of shared pointers to `comments`, which in this case represents the list of all the comments left by this user (possibly on different places).

It implements a method that receives a `std::string` (the text of the new comment) and a `place` and allows the user to leave that comment to the given place. In order to do this (see lines 3 to 8 of file "user.cpp"), it initializes a `std::shared_ptr<comment>` with the new comment, it adds this pointer to the vector `comments` and finally it calls the method `place::add_comment` to add the new comment to the list of comments relative to that place.

The use of shared pointers allows to have only one instance of every comment which is created within the method `user::leave_a_comment` and it is shared by the `user` and the `place`.

***** file user.cpp *****

```

1 #include "user.hh"
2
3 void user::leave_a_comment (const std::string& txt, place & p)
4 {
5     std::shared_ptr<comment> c = std::make_shared<comment> (txt);
6     comments.push_back (c);
7     p.add_comment (c);
8 }

```

Exercise 2 Doctor (*exam 23/01/2018*)

Given the code below:

```

1 #include <iostream>
2 #include <memory>
3 #include <string>
4
5 class Doctor
6 {
7     protected:
8         unsigned doctor_id;
9         std::shared_ptr<Doctor> assistant;
10
11 public:
12
13     Doctor(unsigned id) : doctor_id(id){}
14
15     virtual void print_id (void) const
16     {
17         std::cout << "Doctor id: " << doctor_id << "\n";
18     }
19
20     void assign_assistant(const std::shared_ptr<Doctor> &ptr)
21     {
22         this -> assistant = ptr;
23     }
24
25     virtual ~Doctor (void) {}
26 };
27
28 class Surgeon : public Doctor
29 {
30     public:
31
32     std::string specialty;
33
34     Surgeon(unsigned id, std::string s) : Doctor(id), specialty(s){}
35
36     virtual void print_id (void) const override
37     {
38         std::cout << "Surgeon id: " << doctor_id << "\n";
39     }
40
41     virtual void print_specialty (void) const
42     {
43         std::cout << "Specialty: " << specialty << "\n";
44     }
45
46     virtual void operate(void) const
47     {
48         std::cout << "Surgery Responsible: " << std::endl;
49         this -> print_id ();
50         std::cout << "Surgery Assistant: " << std::endl;
51         this -> assistant -> print_id ();
52     }

```

```

53
54     virtual ~Surgeon (void) {}
55 };
56
57 int main (void)
58 {
59     Doctor dt1 (1);
60     Doctor dt2 (2);
61     Surgeon dt3 (3, "General");
62     Surgeon dt4 (4, "Neurological");
63
64     dt2 = dt3;
65     dt3 = dt1;
66
67     Doctor &refY = dt3;
68     Doctor * ptrX = &dt4;
69
70     refY.print_id();
71     ptrX->print_id();
72     ptrX->print_specialty();
73
74     std::shared_ptr<Doctor> ptrA = std::make_shared<Doctor> (dt1);
75     std::shared_ptr<Doctor> ptrB = std::make_shared<Doctor> (dt2);
76     std::shared_ptr<Doctor> ptrC = std::make_shared<Surgeon> (dt3);
77     std::shared_ptr<Surgeon> ptrD = std::make_shared<Surgeon> (dt4);
78
79     ptrB -> print_id ();
80     ptrC -> assign_assistant (ptrB);
81     ptrD -> assign_assistant (ptrA);
82
83     ptrC -> operate ();
84     ptrD -> operate ();
85     return 0;
86 }
```

You have to identify the line(s) with compilation error(s), explaining why they would fail to be compiled. Additionally, assuming this(these) line(s) have been commented out, provide the expected output from the `main` function once the program has been compiled and executed.

Exercise 2 - Solution

The errors are at lines 65 (we cannot assign a base class object to a derived class variable), 72 (the pointer `Doctor *ptrX` cannot be used to access members of the derived `Surgeon` class), and 83 (the shared pointer `shared_ptr<Doctor> ptrC` cannot be used to access members of the derived `Surgeon` class).

Once the aforementioned lines have been commented out and the program compiled, the expected output is:

```

Surgeon id: 3
Surgeon id: 4
Doctor id: 3
Surgery Responsible:
Surgeon id: 4
Surgery Assistant:
```

Exercise 3 Local Search*

The aim of this exercise is to implement a program which computes the scheduling of a given number of jobs on a single machine, minimizing the weighted tardiness of the schedule.

In particular, the tardiness of a scheduled job is 0 if its completion time is less than the due date (i.e. the deadline), while it is computed as *completion time – due date* otherwise. If to a job corresponds a weight (which represents the importance of the job itself), then the weighted tardiness is computed by multiplying the tardiness by the weight of the given job. For example, consider the list of jobs in Table 5.1. Every job is characterized by an id, a submission time (which represents, if it is greater than zero, the time before which it is not possible to start the job. This can be important in the case of tasks that must be executed one strictly after the other), a completion time and a due date.

A possible schedule for the given jobs is represented by Table 5.2. The first column represents the sequence in which the jobs are executed. The start time of the first job is, of course, zero, because it has no submission time and it does not have to wait for the completion of the previous task. Notice that we compute the schedule following the assumption that all the tasks must be executed sequentially: it is not possible to start a job until the previous one has been completed.

Implement a program which computes the schedule of a given list of jobs by relying on the local search algorithm presented below.

In order to improve the performance of the program, consider to evaluate moves without generating copies of the current solution and to limit the number of evaluations of the objective function (i.e., evaluate the variation of the function instead of computing its value from scratch).

Local Search algorithm:¹

Consider the optimization problem:

$$\min_{s \in \bar{S}} \hat{f}(s)$$

where \bar{S} is a discrete set. The function \hat{f} is called objective function.

Starting from a given $s_0 \in \bar{S}$, at any iteration we want to explore the set of all the neighbouring points of the current s^* . In particular, the neighborhood of s^* , i.e. the set of all the points that can be reached from s^* , is denoted by $N(s^*)$ and it can be traversed completely or not, according to the chosen exploration scheme. A new current point s'' is chosen from $N(s^*)$ if the objective function \hat{f} is improved. If no points in $N(s^*)$ can be accepted, the algorithm terminates:

```

1 s* ← s0
2 loop
3   s" = s*
4   for all s' mathin N(s") do
5     if f(s') < f (s") then
6       s" = s'
7       {we can stop the search on the best or on the first improving point}
8     end if
9   end for
10  if f(s") < f(s*) then
11    s* ← s"

```

¹Credits: Mirko Maischberger, COIN-OR METSlab a Metaheuristics Framework in Modern C++.

Table 5.1: List of jobs to be scheduled

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

Table 5.2: Sequence of scheduled jobs

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

```

12     else
13         return s*
14     end if
15 end loop

```

Algorithm 5.1: Local Search

In Figure 5.1, you can see an example of how the Algorithm 5.1 works. In particular, you can see two iterations of the algorithm, where the improved solution is selected among the possible solutions in $N(s'')$ (i.e. the orange dots) in order to reach the minimum of the given function.

Note: the local search algorithm, as it is described above, can lead to some issues when the objective function has also local minima. Indeed, see what happens in Figure 5.2b: starting from the current solution, the algorithm improves it until it reaches a local optimum (following the black arrow in the image). When the local optimum is found, the algorithm stops, because any further iteration (represented by the red arrows in the image) leads to a worse result for the objective function (indeed, of course, the value of the objective function in any point within the neighborhood of the local minimum is greater than its value in the minimum itself). This means that, starting from the current solution in Figure 5.2b, the global minimum can never be achieved.

Exercise 3 - Solution

```
***** file Job.hh *****
1 #ifndef JOB_H_
2 #define JOB_H_
3
4 #include <iostream>
5
6 class Job
7 {
8 public:
```

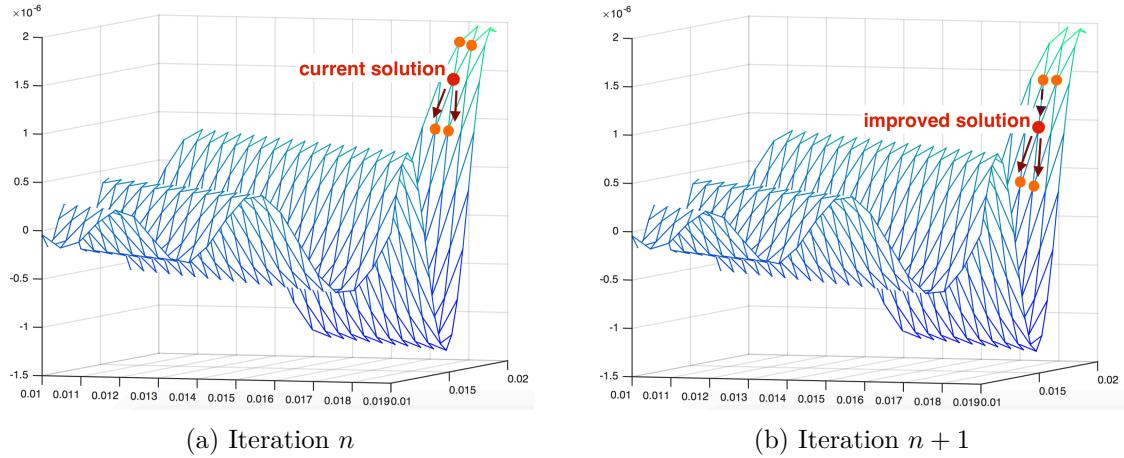


Figure 5.1: Iterations of Local Search Algorithm

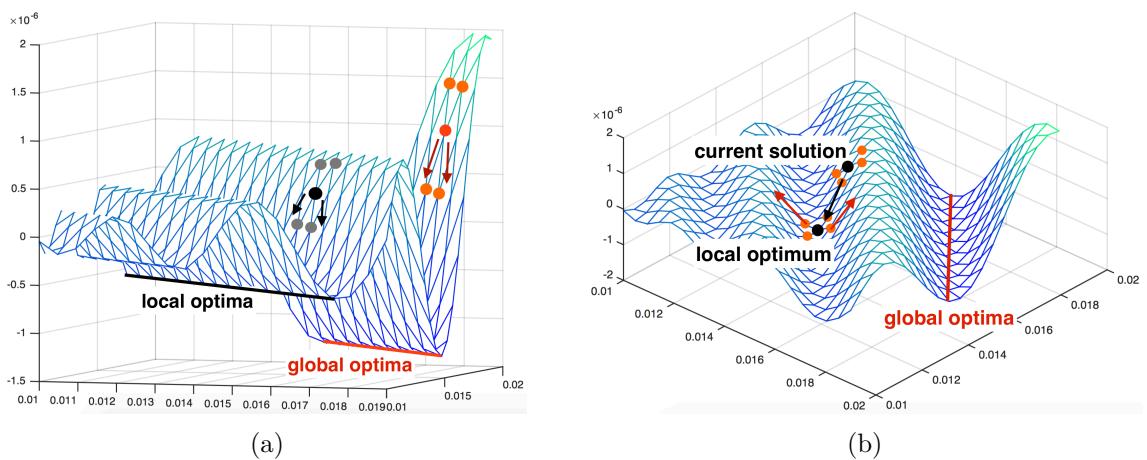


Figure 5.2: Finding local or global optima

```

9  typedef long int time_instant;
10 typedef unsigned int id_type;
11
12 Job (id_type j_id, time_instant sub_time, time_instant exe_time,
13       time_instant dline, double w = 1.0)
14   : id (j_id), submission_time (sub_time),
15     execution_time (exe_time), deadline (dline), weight (w)
16 {
17     adjust_deadline();
18 }
19
20 time_instant get_deadline() const;
21 time_instant get_execution_time() const;
22 time_instant get_submission_time() const;
23 id_type get_ID() const;
24
25 double get_weight() const;
26
27 void set_deadline (time_instant deadline);
28 void set_execution_time (time_instant execution_time);
29 void set_submission_time (time_instant submission_time);
30 void set_weight (double weight);
31 void print() const;
32
33 private:
34     id_type id;
35
36     time_instant submission_time;
37     time_instant execution_time;
38     time_instant deadline;
39     double weight;
40
41     void
42     adjust_deadline (void);
43 };
44
45 #endif /* JOB_H_ */
```

An object of class `Job` is characterized by an `id`, a submission time, an execution time, a deadline and a weight (see for example Table 5.1, where the execution time is called Process Time and the weight of all the listed jobs is equal to 1). Notice that all the time variables are represented as `long int` (see line 9 of file "Job.hh").

The class implements a `private` method, namely `adjust_deadline` (see the implementation in lines 58 to 62 of file "Job.cc"), which checks whether the value assigned to `deadline` is admissible and corrects it otherwise. In particular, a deadline is not admissible if it is less than the submission time of the job plus its execution time.

***** file Job.cc *****

```

1 #include "Job.hh"
2
3 Job::time_instant Job::get_deadline() const
4 {
5     return deadline;
```

```

6 }
7
8 Job::time_instant Job::get_execution_time() const
9 {
10    return execution_time;
11}
12
13 Job::time_instant Job::get_submission_time() const
14 {
15    return submission_time;
16}
17
18 double Job::get_weight() const
19 {
20    return weight;
21}
22
23 Job::id_type Job::get_ID() const
24 {
25    return id;
26}
27
28 void Job::set_deadline (time_instant d)
29 {
30    deadline = d;
31    adjust_deadline();
32}
33
34 void Job::set_execution_time (time_instant t)
35 {
36    execution_time = t;
37    adjust_deadline();
38}
39
40 void Job::set_submission_time (time_instant t)
41 {
42    submission_time = t;
43    adjust_deadline();
44}
45
46 void Job::set_weight (double w)
47 {
48    weight = w;
49}
50
51 void Job::print() const
52 {
53    std::cout << "id: " << id << " submission_time " << submission_time
54        << " execution_time " << execution_time << " deadline "
55        << deadline << " weight " << weight;
56}
57

```

```

58 void Job::adjust_deadline (void)
59 {
60     if (deadline < submission_time + execution_time)
61         deadline = submission_time + execution_time;
62 }

```

***** file ScheduledJob.hh *****

```

1 #ifndef SCHEDULEDJOB_H_
2 #define SCHEDULEDJOB_H_

3
4 #include <memory>
5
6 #include "Job.hh"
7
8 class ScheduledJob
9 {
10    std::shared_ptr<Job> ptr;
11    Job::time_instant start_time;
12
13    void adjust_start_time (void);
14
15 public:
16    ScheduledJob (const std::shared_ptr<Job> & j_ptr,
17                  Job::time_instant st_time)
18        : ptr (j_ptr), start_time (st_time)
19    {
20        adjust_start_time();
21    }
22
23    ScheduledJob (Job j, Job::time_instant st_time)
24        : ptr (std::make_shared<Job> (j)), start_time (st_time)
25    {
26        adjust_start_time();
27    }
28
29    explicit ScheduledJob (const std::shared_ptr<Job> & j_ptr)
30        : ptr (j_ptr),
31          start_time (j_ptr->get_submission_time())
32    {}
33
34    Job::time_instant get_end_time() const;
35
36    Job::time_instant get_start_time() const;
37
38    void set_start_time (Job::time_instant st_time);
39
40    void print() const;
41
42    double evaluate() const;
43
44    Job::time_instant get_deadline() const;
45

```

```

46     Job::time_instant get_execution_time() const;
47
48     Job::time_instant get_submission_time() const;
49 }
50
51 #endif /* SCHEDULEDJOB_H_ */

```

The class `ScheduledJob` stores a shared pointer to `Job` and the start time of the job itself, which of course cannot be less than its submission time (see the implementation of the `private` method `adjust_start_time` in lines 50 to 54 of file "ScheduledJob.cc").

The end time of a scheduled job can be computed (see the implementation of the method `get_end_time` in lines 5 to 8 of the file "ScheduledJob.cc") as the sum between the start time and the execution time of the job itself.

The method `evaluate` (see lines 17 to 22 of file "ScheduledJob.cc") returns the weighted tardiness of the scheduled job.

***** file `ScheduledJob.cc` *****

```

1 #include "ScheduledJob.hh"
2
3 using namespace std;
4
5 Job::time_instant ScheduledJob::get_end_time() const
6 {
7     return start_time + ptr -> get_execution_time();
8 }
9
10 void ScheduledJob::print() const
11 {
12     ptr -> print();
13     cout << " start_time " << start_time
14     << " end_time " << get_end_time();
15 }
16
17 double ScheduledJob::evaluate() const
18 {
19     return get_end_time() < ptr -> get_deadline()
20         ? 0
21         : ptr -> get_weight() * (get_end_time() - ptr -> get_deadline());
22 }
23
24 Job::time_instant ScheduledJob::get_start_time() const
25 {
26     return start_time;
27 }
28
29 void ScheduledJob::set_start_time (Job::time_instant st_time)
30 {
31     start_time = st_time;
32     adjust_start_time();
33 }
34
35 Job::time_instant ScheduledJob::get_deadline() const
36 {

```

```

37   return ptr -> get_deadline();
38 }
39
40 Job::time_instant ScheduledJob::get_execution_time() const
41 {
42   return ptr -> get_execution_time();
43 }
44
45 Job::time_instant ScheduledJob::get_submission_time() const
46 {
47   return ptr -> get_submission_time();
48 }
49
50 void ScheduledJob::adjust_start_time (void)
51 {
52   if (start_time < ptr -> get_submission_time())
53     start_time = ptr -> get_submission_time();
54 }
```

***** file Schedule.hh *****

```

1 #ifndef SCHEDULE_H_
2 #define SCHEDULE_H_
3
4 #include <vector>
5 #include <memory>
6 #include <initializer_list>
7
8 #include "ScheduledJob.hh"
9
10 class Schedule
11 {
12   std::vector<ScheduledJob> order;
13
14   // guarantees that the schedule is valid, i.e. a job
15   // starts after the end of the one before
16   void validate();
17
18 public:
19   typedef std::vector<ScheduledJob>::size_type size_type;
20
21   Schedule() {};
22
23   virtual ~Schedule() {};
24
25   size_type size (void) const;
26
27   // evaluate total weighted tardiness for the schedule;
28   double evaluate() const;
29
30   void print() const;
31   void add (const std::shared_ptr<Job> & j);
32 }
```

```

33 // swap jobs at position i and j
34 void swap (std::size_t i, std::size_t j);
35 };
36
37 #endif /* SCHEDULE_H_ */

```

The class `Schedule` will be used to implement the local search algorithm in order to compute the best schedule for a collection of jobs, i.e. the one that minimizes the overall tardiness. The class stores a `std::vector` of `ScheduledJobs` and it allows to compute the overall tardiness of the schedule (through the method `evaluate`, see line 28 of file "Schedule.hh") and to swap two jobs within the vector in order to change the relative order (through the method `swap`, see line 34 of the file "Schedule.hh").

***** file Schedule.cc *****

```

1 #include "Schedule.hh"
2
3 Schedule::size_type Schedule::size () const
4 {
5     return order.size();
6 }
7
8 double Schedule::evaluate() const
9 {
10    double ret_val = 0;
11
12    for (std::size_t i = 0; i < order.size(); ++i)
13    {
14        ret_val += order[i].evaluate();
15    }
16
17    return ret_val;
18 }
19
20 void Schedule::print() const
21 {
22    for (std::size_t i = 0; i < order.size(); ++i)
23    {
24        order[i].print();
25        std::cout << std::endl;
26    }
27 }
28
29 void Schedule::add (const std::shared_ptr<Job> & j)
30 {
31     // this is the first job
32     if (order.empty())
33         order.push_back (ScheduledJob (j));
34     else
35     {
36         ScheduledJob sj (j, order.back().get_end_time());
37         order.push_back (sj);
38     }
39 }

```

```

40
41 // swap jobs at position i and j
42 void Schedule::swap (std::size_t i, std::size_t j)
43 {
44     if (i >= order.size() || j >= order.size())
45         return;
46
47     using std::swap;
48     swap (order[i], order[j]);
49     validate();
50 }
51
52 void Schedule::validate()
53 {
54     // guarantees that the schedule is valid,
55     // i.e. a job starts after the end of the one before
56
57     // This can be optimized, check only after min(i,j)
58
59     //an empty schedule is ok
60     if (order.empty())
61         return;
62
63     order[0].set_start_time (order[0].get_submission_time());
64     ScheduledJob pred = order[0];
65
66     for (std::size_t i = 1; i < order.size(); ++i)
67     {
68         // adjust current job start time
69         order[i].set_start_time (std::max (pred.get_end_time(),
70                                         order[i].get_submission_time()));
71         pred = order[i];
72     }
73 }
```

The method `evaluate` (see lines 8 to 18 of file "Schedule.cc") loops over all the `ScheduledJobs` stored in `order` and it sums the relative tardinesses.

The method `add` (see lines 29 to 39) allows to append a new `ScheduledJob` to the vector `order`. If the given `Job` is the first to be scheduled, we use the constructor of `ScheduledJob` class which receives as parameter only the shared pointer. Indeed, the start time of the job, since it is the first, surely corresponds to its submission time. If, however, the vector `order` already stores other jobs, the start time of the new one will correspond to the end time of the previous one (i.e. the last which is stored in `order`).

The method `swap` (see lines 42 to 50) checks whether the two given indices are internal to the vector `order`. If this is the case, it swaps the two `ScheduledJobs` stored in the given positions and it calls the method `validate` in order to guarantee that the new schedule is admissible.

The `private` method `validate` (see lines 52 to 73) is based on the assumption that a schedule is valid if every job starts after the end of the one before. In particular, the start time of the first job in `order` corresponds to its submission time. Then, the method loops over all the `ScheduledJobs` stored in `order` and sets the start time of the current job to the maximum between its submission time and the end time of the previous job.

***** file main.cc *****

```

1 #include <iostream>
2 #include <memory>
3
4 #include "Job.hh"
5 #include "ScheduledJob.hh"
6 #include "Schedule.hh"
7
8 using namespace std;
9
10 int main()
11 {
12     //Job(id_type j_id, time_instant sub_time, time_instant exe_time, time_instant
13     //dline, double w =1.0):
14     auto j1 = std::make_shared<Job> (0, 0, 1, 1, 2.0);
15     auto j2 = std::make_shared<Job> (1, 0, 2, 2, 3.0);
16     auto j3 = std::make_shared<Job> (2, 10, 4, 4);
17     Schedule s;
18     j1 -> print();
19     cout << endl;
20     j2 -> print();
21     cout << endl;
22     j3 -> print();
23     cout << endl;
24     cout << endl;
25     s.add (j3);
26     s.add (j2);
27     s.add (j1);
28     s.print();
29     cout << "Total: " << s.evaluate() << endl << endl;
30     std::size_t dim = 3;
31     Schedule s_opt = s;
32     double best_val = s.evaluate();
33     bool improved = true;
34     unsigned max_iterations = 10;
35     cout << "Start Neigh exploration " << endl;
36     s.print();
37     cout << "Initial value " << best_val << endl;
38     cout << endl;
39     for (unsigned iter = 0; iter < max_iterations && improved; ++iter)
40     {
41         cout << "Iteration " << iter << endl << endl;
42         Schedule current_best = s_opt;
43         double current_best_val = best_val;
44         improved = false;
45
46         for (unsigned i = 0; i < dim - 1; ++i)
47         {
48             cout << "swap " << i << " " << i + 1 << endl;
49             Schedule s_current = s_opt;
50             s_current.swap (i, i + 1);
51             cout << "Evaluating " << endl;

```

```

52     s_current.print();
53     cout << endl;
54     const double current_val = s_current.evaluate();
55     cout << "Value: " << current_val << endl;
56
57     if (current_val < current_best_val)
58     {
59         cout << "Improving " << endl;
60         current_best = s_current;
61         improved = true;
62         current_best_val = current_val;
63         cout << "New best " << endl;
64         current_best.print();
65         cout << endl;
66     }
67 }
68
69     std::swap (s_opt, current_best);
70     best_val = current_best_val;
71 }
72
73 cout << "Final optimal solution found" << endl;
74 s_opt.print();
75 cout << endl;
76 cout << "Total: " << s_opt.evaluate() << endl;
77 return 0;
78 }
```

In the `main` function we can find the actual implementation of the local search algorithm. First of all, a `Schedule` object is instantiated by three jobs (see lines 12 to 27) and the total value of its tardiness is computed in line 28.

In lines 29 to 33, we instantiate the starting point of the local search algorithm, by setting the initial `Schedule` to be equal to `s` (line 30), the best value of the overall tardiness to `s.evaluate()` and the maximum number of iterations to 10.

In lines 39 to 71, we loop until the maximum number of iterations is reached or until the value of `improved` become `false`, i.e. the new iteration does not enhance the value of the objective function.

In particular, lines 42 and 43 of the file "main.cc" correspond to line 3 of the algorithm description reported in Algorithm 5.1, where we initialize the current solution to the previous result. Then, in lines 46 to 67 we implement lines 4 to 9 of Algorithm 5.1. Notice that, in this context, the neighborhood of the `Schedule` `current_best` corresponds to the set of all the possible schedules we get by initializing `s_current` to `s_opt` and by calling `Schedule::swap` over two jobs of `s_current`. For all the possible schedules in the neighborhood, we compute the overall tardiness and we compare it to the one of the current best solution, i.e. to `current_best_val`. If the new solution is better than the old one, i.e. its tardiness is smaller, we set `current_best` to `s_current` and `improved` to `true` (see lines 57 to 66 of the file "main.cc" and lines 5 to 8 of Algorithm 5.1).

Exercise 4 Fraud Detection

Note: In order to complete this exercise, the knowledge of streams is required.

Starting from the code attached below, implement a program for fraud detection. In particular, provide the implementation of the following methods:

- `csv_manager::parse_csv()`, which receives as input the name of a csv file (whose fields are separated by ';') and parses it, storing the data in the matrix `fields` (see file "csv_manager.hpp").
- `csv_manager::create_payers()`, which receives as input a vector of `Tax_Payers` and it populates this vector starting from the elements of the matrix `fields` (see file "csv_manager.hpp").

Write a function which identifies the tax payers whose total credit card expenses, in a specific fiscal year, were more than 10% of the income tax they payed (assume every payer pays one and only one income tax per year).

In the following we discuss the provided classes and functions. Within the `namespace Date_Util`, we provide a function which receives a string that represents a date and it returns the corresponding year (see the file "date_util.cpp" for the implementation).

***** file `date_util.hpp` *****

```

1 #ifndef DATE_UTIL_HPP
2 #define DATE_UTIL_HPP
3
4 #include <string>
5 #include <sstream>
6
7 namespace fraud_detection
8 {
9     namespace Date_Util
10    {
11        unsigned get_year_from_string(std::string);
12    }
13 }
14
15 #endif

```

The implementation of the function `get_year_from_string` is based on the assumption that the dates are represented in the format "day/month/year", where, in place of "/", we can have whatever `char` delimiter.

***** file `date_util.cpp` *****

```

1 #include "date_util.hpp"
2
3 namespace fraud_detection
4 {
5     namespace Date_Util
6     {
7         unsigned get_year_from_string(std::string date)
8         {
9             std::istringstream is(date);
10            unsigned d, m, y;
11            char delimiter;
12            if (is >> d >> delimiter >> m >> delimiter >> y) {
13                return y;
14            }
15        }
16    }
17 }

```

```

14     }
15     return 0;
16   }
17 }
18 }
```

The class `Credit_Transaction` is used to represent a transaction, thus it stores the `date` (in a string format), the `amount` and the `address` of the operation. It implements the relevant getters to read such information, plus a method that, relying on `Date_Util::get_year_from_string()`, returns the year corresponding to the date.

***** file `credit_transaction.hpp` *****

```

1 #ifndef Credit_Transaction_HPP
2 #define Credit_Transaction_HPP
3
4 #include <iostream>
5 #include <utility>
6 #include <vector>
7
8 #include "date_util.hpp"
9
10 namespace fraud_detection
11 {
12   class Credit_Transaction
13   {
14     private:
15       /*The transaction date*/
16       std::string date;
17
18       /*The transaction amount*/
19       double amount;
20
21       /*The transaction address*/
22       std::string address;
23
24     public:
25
26       /*Constructor*/
27       Credit_Transaction (const std::string & dt, const double & e, const std::string
28       & adrs) :
29         date (dt),
30         amount (e),
31         address (adrs)
32     {
33       std::cout << "Credit_Transaction(" << dt << "," << e << "," << adrs << ")" << std
34       ::endl;
35     }
36
37     const std::string & get_date (void) const;
38
39     unsigned get_year (void) const;
40
41     double get_amount (void) const;
```

```

40         const std::string & get_address (void) const;
41     };
42 }
43 #endif
44

***** file credit_transaction.cpp *****
1 #include "credit_transaction.hpp"
2
3 namespace fraud_detection
4 {
5     const std::string & Credit_Transaction::get_date (void) const
6     {
7         return date;
8     }
9
10    unsigned Credit_Transaction::get_year (void) const
11    {
12        return Date_Util::get_year_from_string(date);
13    }
14
15    double Credit_Transaction::get_amount (void) const
16    {
17        return amount;
18    }
19
20    const std::string & Credit_Transaction::get_address (void) const
21    {
22        return address;
23    }
24 }

```

The class `Credit_Card` stores a vector of shared pointers to `Credit_Transactions`, which represents the operations performed by the credit card itself. Moreover, each `Credit_Card` has an identification number and an expiry date.

Note that we refer to a `std::vector<std::shared_ptr<Credit_Transaction>>` by the user defined type `transactions_vector` (see line 16 of file "credit_card.hpp").

Other than the relevant getters, the class implements a method to add a new credit transaction to the vector `transactions` (see the declaration in line 39 of the file "credit_card.hpp") and a method, that, given a year, returns the total amount of transactions in that year (see line 44).

```
***** file credit_card.hpp *****
```

```

1 #ifndef CREDIT_CARD_HPP
2 #define CREDIT_CARD_HPP
3
4 #include "credit_transaction.hpp"
5 #include "date_util.hpp"
6
7 #include <iostream>
8 #include <vector>
9 #include <memory>

```

```

10 #include <utility>
11
12 namespace fraud_detection
13 {
14     class Credit_Card
15     {
16         typedef std::vector<std::shared_ptr<Credit_Transaction>> transactions_vector;
17
18         std::string number;
19         std::string expiry_date;
20         transactions_vector transactions;
21
22     public:
23
24     /*Constructor*/
25     Credit_Card(const std::string & s, const std::string & d):number(s), expiry_date(d)
26     {
27         std::cout << "Card(" << s << "," << d << ")" << std::endl;
28     }
29
30     /*Public interface*/
31
32     const std::string & get_expiry_date (void) const;
33
34     const std::string & get_number (void) const;
35
36     const transactions_vector & get_transactions (void) const;
37
38     /*Adds a transaction to the transactions vector*/
39     void add_transaction(const Credit_Transaction & new_trs);
40
41     /* Returns the total amount of all transactions in a specific date
42      * Receives: the date of the transactions
43      * Returns: the total amount of transactions in the specific date */
44     double get_trs_amount (unsigned year) const;
45 };
46 }
47 #endif

```

***** file credit_card.cpp *****

```

1 #include "credit_card.hpp"
2
3 namespace fraud_detection
4 {
5     const std::string & Credit_Card::get_expiry_date (void) const
6     {
7         return expiry_date;
8     }
9
10    const std::string & Credit_Card::get_number (void) const
11    {

```

```

12     return number;
13 }
14
15 const Credit_Card::transactions_vector & Credit_Card::get_transactions (void)
16     const
17 {
18     return transactions;
19 }
20
21 void Credit_Card::add_transaction(const Credit_Transaction & new_trs)
22 {
23     auto p = std::make_shared<Credit_Transaction>(new_trs);
24     this->transactions.push_back(p);
25 }
26
27 double Credit_Card::get_trs_amount (unsigned year) const
28 {
29     double total = 0;
30     for (const auto & transaction: transactions)
31     {
32         unsigned transaction_year = Date_Util::get_year_from_string(transaction->
33         get_date());
34         if (year == transaction_year)
35             total += transaction->get_amount();
36     }
37     return total;
38 }
39

```

The class Declaration represents a tax declaration. It stores the number, the type, the date and the amount of the declaration and it provides the relevant getters to read these information.

***** file declaration.hpp *****

```

1 #ifndef DECLARE_HPP
2 #define DECLARE_HPP
3
4 #include <memory>
5 #include <iostream>
6 #include <vector>
7 #include <string>
8
9 #include "date_util.hpp"
10
11 namespace fraud_detection
12 {
13     class Declaration
14     {
15         private:
16             /*Declaration tax number*/
17             std::string number;
18             /*The type of declaration tax*/
19             std::string tax;

```

```

20     /*The date of the declaration*/
21     std::string date;
22     /*The declared amount*/
23     double amount;
24
25 public:
26     /*Constructor*/
27     Declaration(const std::string & n, const std::string & t, const std::string & d,
28     const double & a) :
29         number(n),
30         tax(t),
31         date(d),
32         amount(a)
33     {
34         std::cout << "Declaration(" << n << "," << t << "," << d << "," << a << ")" <<
35         std::endl;
36     }
37
38     /*Public interface*/
39
40     const std::string & get_number (void) const;
41
42     const std::string & get_date (void) const;
43
44     unsigned get_year (void) const;
45
46     double get_amount (void) const;
47
48     const std::string & get_tax (void) const;
49
50     static std::string INCOME_TAX;
51     static std::string PROPERTY_TAX;
52
53 };
54
55 #endif

```

***** file declaration.cpp *****

```

1 #include "declaration.hpp"
2
3 namespace fraud_detection
4 {
5
6     const std::string & Declaration::get_number (void) const
7     {
8         return number;
9     }
10
11    const std::string & Declaration::get_date (void) const
12    {
13        return date;

```

```

14 }
15
16     unsigned Declaration::get_year (void) const
17 {
18     return Date_Util::get_year_from_string(date);
19 }
20
21     double Declaration::get_amount (void) const
22 {
23     return amount;
24 }
25
26     const std::string & Declaration::get_tax (void) const
27 {
28     return tax;
29 }
30
31     std::string Declaration::INCOME_TAX = "income_tax";
32     std::string Declaration::PROPERTY_TAX = "property_tax";
33
34 }

```

The class `Tax_Payer` stores all the information about a tax payer, i.e. the name, the birth place, the current address, a vector of `Credit_Cards`, a vector of old addresses and a vector of shared pointers to `Declarations` (hidden in the user defined type `declarations_vector`, see line 17 of the file "tax_payer.hpp").

The usage of shared pointers to `Declarations` is justified by the fact that multiple tax payers can share the same declaration (for example, in case of wife and husband).

Other than the relevant getters, the class provides a method to update the address of the tax payer, adding the current one to the vector of old addresses and replacing it with a new one (see the declaration in line 58 of the file "tax_payer.hpp").

It implements two methods, see lines 63 and 67 respectively, to add a new `Declaration` and a new `Credit_Card` to the corresponding vectors.

The method `find_card` (line 72) receives as parameter the number of a `Credit_Card` and it searches it among the `Credit_Cards` of the tax payer (stored in the vector `cards`). It returns the index of the credit card in the vector if it exists, -1 otherwise. Note that in the implementation of this method we need to return a value of type `int` while variable `i` is declared as `size_t` (look at line 57 of file "tax_payer.cpp"). To do this, we can use the operator `static_cast<TYPE>(EXPR)` that converts an expression `EXPR` to type `TYPE` (in this case we would use `static_cast<int>(i)`). Note that in C the same thing could be done by preceding an expression with a type in round parenthesis (e.g., `(int) i`).

The method `append_transaction` (see line 76) receives a new `Credit_Transaction` and the number of the credit card to whom the transaction must be appended. As before, in the implementation of this method (look at line 67 of file "tax_payer.cpp") we rely on the `static_cast` operator to explicitly convert variable `pos` into a `size_t` object, writing `static_cast<size_t>(pos)`.

The method `check_fraud` (see line 83) receives an year as parameter and it returns `true` if a fraud is detected in the given year.

Finally, the methods `get_declared_year_income` and `get_year_transaction_amount` return, respectively, the amount from the declared income tax and the total amount of credit transactions in a given year. Note that the first method (see the implementation in lines 83 to 92 of the file "tax_payer.cpp") uses the `static` variable `INCOME_TAX`, defined in `Declaration`, in order to identify the income tax within the vector of declarations.

```

***** file tax_payer.hpp *****
1 #ifndef TAX_PAYER_HPP_
2 #define TAX_PAYER_HPP_
3
4 #include <iostream>
5 #include <vector>
6 #include <string>
7 #include <memory>
8
9 #include "credit_card.hpp"
10 #include "declaration.hpp"
11
12 namespace fraud_detection
13 {
14     class Tax_Payer
15     {
16         public:
17             typedef std::vector<std::shared_ptr<Declaration>> declarations_vector;
18
19         private:
20             /*Tax payer's name*/
21             std::string name;
22             /*Tax payer's birth place*/
23             std::string birth_place;
24             /*Tax payer's current address */
25             std::string current_address;
26             /*Tax payer's credit cards*/
27             std::vector<Credit_Card> cards;
28             /*Tax payer's old addresses*/
29             std::vector<std::string> old_addresses;
30
31             /*Tax payer's declarations.
32                 One declaration may be shared by multiple tax payers
33                 E.g.: wife and husband may share a declaration*/
34             declarations_vector declarations;
35
36         public:
37             /*Constructor*/
38             Tax_Payer(const std::string & s1, const std::string & s2, const std::string & s3)
39             :
40                 name(s1),
41                 birth_place(s2),
42                 current_address(s3)
43             {
44                 std::cout << "Tax_Payer(" << s1 << "," << s2 << "," << s3 << ")" << std::endl;
45             }
46
47             /*Public interface*/
48             const std::string & get_name (void) const;
49
50             const std::string & get_birth_place (void) const;

```

```

51     const std::string & get_current_address (void) const;
52
53     const std::vector<std::string> & get_old_addresses (void) const;
54
55     /* Adds current address to the old addresses vector and
56      * replaces the current address with a new address
57      */
58     void update_address(const std::string &);

59     const declarations_vector & get_declarations (void) const;
60
61     /*Adds a declaration to the declarations vector*/
62     void add_declaration(const Declaration &);

63     const std::vector <Credit_Card> & get_cards (void) const;
64
65     void add_card(const Credit_Card &);

66     /* Finds a card by name among the vector of cards of this tax payer
67      * Receives: the name of the card to be found
68      * Returns: the found card's index in the vector or -1 if not found */
69     int find_card(const std::string &) const;

70     /* Appends a transaction to a given credit card from the tax payer
71      * Receives: the name of the credit card and the transaction to be appended*/
72     void append_transaction(const std::string &, const Credit_Transaction &);

73     /* Checks for a fraud in a specific year
74      * A fraud is detected for a tax payer if his/her credit card expenses,
75      * in a specific fiscal year, were more than 10% of the total income taxes they
76      have payed
77      * Receives: the fiscal year
78      */
79     bool check_fraud(unsigned) const;

80     /* Gets the amount from the declared income tax of a given year for this tax
81     payer
82      * Receives: the year of the income declaration
83      * Returns: the amount of the income declaration for the year
84      */
85     double get_declared_year_income(unsigned) const;

86     /* Gets the total amount of all credit transactions of a given year for this tax
87     payer
88      * Receives: the year of the income
89      * Returns: the total amount of all credit transactions of a given year
90      */
91     double get_year_transaction_amout(unsigned) const;
92
93 };
94
95 #endif

```

```

***** file tax_payer.cpp *****
1 #include "tax_payer.hpp"
2 #include <string>
3
4 namespace fraud_detection
{
5
6     const std::string & Tax_Payer::get_name (void) const
7     {
8         return name;
9     }
10
11    const std::string & Tax_Payer::get_birth_place (void) const
12    {
13        return birth_place;
14    }
15
16    const std::string & Tax_Payer::get_current_address (void) const
17    {
18        return current_address;
19    }
20
21    void Tax_Payer::update_address (const std::string & new_address)
22    {
23        old_addresses.push_back (current_address);
24        current_address = new_address;
25    }
26
27    std::vector<std::string> const & Tax_Payer::get_old_addresses (void) const
28    {
29        return old_addresses;
30    }
31
32    Tax_Payer::declarations_vector const & Tax_Payer::get_declarations (void) const
33    {
34        return declarations;
35    }
36
37    void Tax_Payer::add_declaration (const Declaration & d)
38    {
39        auto p = std::make_shared<Declaration> (d);
40        this->declarations.push_back (p);
41    }
42
43    const std::vector<Credit_Card> & Tax_Payer::get_cards (void) const
44    {
45        return cards;
46    }
47
48    void Tax_Payer::add_card (const Credit_Card & c)
49    {
50        cards.push_back (c);
51    }

```

```

52
53     int Tax_Payer::find_card(const std::string & card_number) const{
54         for(size_t i = 0; i < cards.size(); i++){
55             Credit_Card card = cards[i];
56             if(card.get_number() == card_number)
57                 return static_cast<int>(i);
58         }
59         return -1;
60     }
61
62     void Tax_Payer::append_transaction(const std::string & card_number, const
63                                         Credit_Transaction & transaction)
64     {
65         int pos = find_card(card_number);
66         if(pos != -1){
67             Credit_Card & card = cards[static_cast<size_t>(pos)];
68             card.add_transaction(transaction);
69         }
70     }
71
72     bool Tax_Payer::check_fraud(unsigned year) const
73     {
74
75         double total_year_transaction_amout = get_year_transaction_amout(year);
76         std::cout << "Total Year Credit Transaction Amount: " <<
77         total_year_transaction_amout << std::endl;
78         double declared_year_income = get_declared_year_income(year);
79         std::cout << "Declared Income: " << declared_year_income << std::endl;
80         double ratio = 100 * total_year_transaction_amout / declared_year_income;
81         std::cout << "Ratio: " << ratio << "%" << std::endl;
82         return ratio > 10;
83     }
84
85     double Tax_Payer::get_declared_year_income(unsigned year) const
86     {
87         for(auto & d : declarations)
88         {
89             if( d->get_year() == year &&
90                 d->get_tax() == Declaration::INCOME_TAX)
91                 return d->get_amount();
92         }
93         return 0;
94     }
95
96     double Tax_Payer::get_year_transaction_amout(unsigned year) const
97     {
98         double total_year_amout = 0;
99         for(Credit_Card card : cards)
100            total_year_amout += card.get_trs_amount(year);
101    }

```

The class `csv_manager` provides the methods to read the csv files with all the information about tax payers, credit cards, credit transactions and declarations. It allows to store all those information in a matrix, called `fields`, to create a vector of `Tax_Payers` and to add to each `Tax_Payer` the corresponding vectors of `Credit_Cards` (populated with the corresponding `Credit_Transactions`) and of `Declarations`.

Moreover, it provides two methods to find a specific payer and a specific declaration, providing the name and the number, respectively (as the method `Tax_Payer::find_card`, both return -1 if the desired object is not stored in the corresponding vector).

All the methods listed above are **private**. Indeed, the user is only intended to employ the method **parse_files** (see the declaration in line 70 of the file "csv_manager.hpp"), providing the vector of **Tax_Payers** that he wants to populate through the information coming from the csv files (see the **main** function in file "main.cpp").

***** file csv_manager.hpp *****

```

1 #ifndef csv_manager_hpp
2 #define csv_manager_hpp
3
4 #include <iostream>
5 #include <vector>
6 #include <fstream>
7 #include <sstream>
8 #include <vector>
9 #include <string>
10
11 #include "tax_payer.hpp"
12 #include "credit_transaction.hpp"
13 #include "credit_card.hpp"
14
15 namespace fraud_detection
16 {
17     class csv_manager
18     {
19         private:
20             /* The file names of the csv files with data about payers,
21              * cards, card transactions, and declarations */
22             std::string tax_payers_csv, cards_csv, card_transactions_csv, declarations_csv
23             ;
24
25             /*A vector of vectors of strings used as data structure for the fields in the csv
26             files*/
27             std::vector<std::vector<std::string>> fields;
28
29             /*Common method that parses the csv file formatted as text with ';' as field
30             separators*/
31             void parse_csv(std::string const&);
32
33             /* Creates a vector of tax payer objects based on the fields from the parsed csv
34             file
35             * Receives: the vector of tax payer objects to be populated */
36             void create_payers(std::vector<Tax_Payer> &);
37
38             /* Creates vectors of card objects based on the fields from the parsed csv file
39             */
40     };

```

```

35     * and adds them to their respective tax payer object
36     * Receives: the vector of tax payer objects already populated */
37 void create_cards(std::vector<Tax_Payer> &z);
38
39     /* Creates vectors of card objects based on the fields from the parsed csv file
40     * and adds them to their respective tax payer object
41     * Receives: the vector of tax payer objects already populated */
42 void create_transactions(std::vector<Tax_Payer> &z);
43
44     /* Creates vectors of declaration objects based on the fields from the parsed csv
file
45     * and adds them to their respective tax payer object
46     * Receives: the vector of tax payer objects already populated */
47 void create_declarations(std::vector<Tax_Payer> &z);
48
49     /* Finds a specific tax payer by name from a vector of tax payers
50     * Receives: the name of the payer to be found and the vector of tax payers
51     * Returns: the index of the tax payer in the vector or -1 if not found*/
52 int find_payer(std::string const&, std::vector<Tax_Payer> const&);

53     /* Finds a specific declaration by number from a vector of declarations
54     * Receives: the number of the declaration to be found and the vector of
declarations
55     * Returns: the index of the declaration in the vector or -1 if not found */
56 int find_declaration(std::string const&, Tax_Payer::declarations_vector const&)
57 ;
58
59 public:
60     /* Constructor
61     * Receives: the file names of the three csv files */
62     csv_manager(std::string const& t_p_csv, std::string const& c_csv, std::string
const& c_t_csv, std::string const& d_csv) :
63         tax_payers_csv(t_p_csv),
64         cards_csv(c_csv),
65         card_transactions_csv(c_t_csv),
66         declarations_csv(d_csv){};

67     /* Parses the csv files and populates a vector of tax payers along with each
payer's
68     * info, including credit cards and credit card transactions */
69     void parse_files (std::vector<Tax_Payer> &z);
70 };
71 }
72
73 #endif

```

As it can be seen in lines 6 to 21 of the file "csv_manager.cpp", the method `parse_files` calls the `private` methods `parse_csv`, `create_payers`, `create_cards`, `create_transactions` and `create_declarations` in order to read all the csv files and to populate the vector `payers` passed as parameter with all the relevant information.

Note that, since we are in the scope of the class, we could avoid to write `csv_manager::` in order to refer to a method of the class itself.

```

***** file csv_manager.cpp *****
1 #include "csv_manager.hpp"
2 using namespace std;
3
4 namespace fraud_detection
{
5
6     void
7         csv_manager::parse_files(std::vector<Tax_Payer> & payers)
8     {
9         //parses the tax payers csv file
10        csv_manager::parse_csv(tax_payers_csv);
11        csv_manager::create_payers(payers);
12        //parses the cards csv file
13        csv_manager::parse_csv(cards_csv);
14        csv_manager::create_cards(payers);
15        //parses the card transactions csv file
16        csv_manager::parse_csv(card_transactions_csv);
17        csv_manager::create_transactions(payers);
18        //parses the card transactions csv file
19        csv_manager::parse_csv(declarations_csv);
20        csv_manager::create_declarations(payers);
21    }
22
23    void
24        csv_manager::parse_csv (const string & file_name)
25    {
26    }
27
28
29    void
30        csv_manager::create_payers(std::vector<Tax_Payer> & payers)
31    {
32    }
33
34
35    void
36        csv_manager::create_cards(std::vector<Tax_Payer> & payers)
37        //Assumption: the correspondig tax payers exist
38    {
39        for (const auto & row : fields)
40        {
41            int pos = find_payer(row[2], payers);
42            if(pos != -1){
43                Tax_Payer & p = payers[pos];
44                Credit_Card c(row[0], row[1]);
45                p.add_card(c);
46            }
47        }
48    }
49
50    void
51        csv_manager::create_transactions(std::vector<Tax_Payer> & payers)

```

```

52     //Assumption: the correspondig tax payers exist
53 {
54     for (const auto & row : fields)
55     {
56         std::string::size_type sz;
57         double temp = std::stod (row[2], &sz);
58         Credit_Transaction new_trs(row[1],temp,row[3]);
59         for(Tax_Payer & p: payers)
60         {
61             const std::vector<Credit_Card>& cards = p.get_cards();
62             bool stop = false;
63             for (std::vector<Credit_Card>::const_iterator cit = cards.cbegin();
64                 cit != cards.cend() && ! stop; ++cit)
65             {
66                 if (cit->get_number() == row[0])
67                 {
68                     p.append_transaction(cit->get_number(), new_trs);
69                     stop = true;
70                 }
71             }
72         }
73     }
74 }
75
76 void
77 csv_manager::create_declarations(std::vector<Tax_Payer> & payers)
78 //Assumption: the correspondig tax payers exist
79 {
80     Tax_Payer::declarations_vector all_declarations;
81     for (const auto & row : fields)
82     {
83         int pos = find_payer(row[4], payers);
84         if(pos != -1){
85             Tax_Payer & tp = payers[pos];
86             std::string::size_type sz;
87             double temp = std::stod (row[3], &sz);
88             pos = find_declaration(row[0], all_declarations);
89             if(pos == -1){//if the declaration has not been created before
90                 Declaration d(row[0], row[1], row[2], temp);
91                 auto p = std::make_shared<Declaration> (d);
92                 all_declarations.push_back(p);
93                 tp.add_declaration(d);
94             }
95             else{//if the declaration has already been created for another tax payer
96                 Declaration & d = *all_declarations[pos];
97                 tp.add_declaration(d);
98             }
99         }
100    }
101 }
102
103 int

```

```

104     csv_manager::find_payer(const std::string & name, const std::vector<Tax_Payer>
105     & payers)
106     {
107         for(size_t i = 0; i < payers.size(); i++){
108             Tax_Payer p = payers[i];
109             if(p.get_name() == name)
110                 return i;
111         }
112         return -1;
113     }
114
115     int
116     csv_manager::find_declaration(const std::string & declaration_number,
117                                   const Tax_Payer::declarations_vector & declarations)
118     {
119         for(size_t i = 0; i < declarations.size(); i++){
120             Declaration d = *declarations[i];
121             if(d.get_number() == declaration_number)
122                 return i;
123         }
124         return -1;
125     }

```

The method `create_cards` (see lines 35 to 48) is based on the assumption that the csv file which stores information about the credit cards is organized in the following way: the first column stores the symbol of the card, the second stores the expiry date and the third the name of the owner (of course, after the call of `parse_csv`, the matrix `fields` has the same structure). The method scans the rows of `fields` and, for any row, if the name of the credit card corresponds to the name of a tax payer (namely, if the index returned by `find_payer` is not -1), it adds the card to his vector of cards. Notice that the instruction `static_cast<size_t>(pos)` in line 43 is used to explicitly convert the integer variable `pos` in a `size_t` and it is equivalent to the more C-style instruction `(size_t) pos`.

The method `create_transactions` (see lines 50 to 74) is based on the assumption that the corresponding csv file, and thus the matrix `fields`, stores in the first column the symbol of the credit card which has performed the transaction, in the second one the date, in the third one the amount and in the last one the address. It scans the rows of the matrix `fields` and, for any row, first of all it converts the amount from a `std::string` (which is what we have in `fields`) to a `double` through the function `std::stod (const std::string &, std::size_t *)`. Then, it creates a new `Credit_Transaction` and, for any `Tax_Payer` in `payers`, it checks if the credit card associated to the transaction is one of the tax payer's credit cards; in this case, it appends the transaction to the corresponding vector.

Finally, the method `create_declarations` (see lines 76 to 101) is based on the assumption that the corresponding csv file stores in the first column the tax number, in the second the type of tax, in the third the date, in the fourth the amount and in the last the name of the tax payer. As the previous ones, it scans all the rows of `fields`. For any row, it checks if the name of the tax payer in the declaration corresponds to a tax payer in `payers`. In this case, it converts the amount in a `double` (as in `create_transactions`) and it looks for the declaration in the vector of `all_declarations` defined in line 80. If it does not exists, it creates it (lines 89 to 94), otherwise it calls the method `Tax_Payer::add_declaration` (see file "tax_payer.hpp").

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

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 #include "tax_payer.hpp"
6 #include "credit_transaction.hpp"
7 #include "credit_card.hpp"
8 #include "csv_manager.hpp"
9
10 using namespace std;
11 using namespace fraud_detection;
12
13 int main()
14 {
15     std::vector<Tax_Payer> tax_payers;
16     csv_manager csv_mngr ("data/tax_payers.csv", "data/cards.csv", "data/credit_trs.
17     csv", "data/declarations.csv");
18     csv_mngr.parse_files (tax_payers);
19
20     for(const Tax_Payer & tp : tax_payers){
21         if(tp.check_fraud(2017))
22             cout << "Fraud detected with income declaration of user " << tp.get_name() <<
23             endl;
24     }
25     return 0;
26 }
```

Exercise 4 - Solution

The first method that has to be implemented is the method `parse_csv`. It opens the file whose name is passed as parameter, calls the method `std::vector::clear()` on the matrix `fields` in order to erase all its elements, then it reads the file line by line and it adds the elements to `fields`.

```

22 void csv_manager::parse_csv (const string & file_name)
23 {
24     ifstream in(file_name);
25     fields.clear();
26     if (in)
27     {
28         string line;
29         while (getline(in, line))
30         {
31             stringstream sep(line);
32             string field;
33             fields.push_back(vector<string>());
34             while (getline(sep, field, ';'))
35             {
36                 fields.back().push_back(field);
37             }
38         }
39     }
```

```

40     else
41     {
42         std::cerr << "Error accessing the file " << file_name << std::endl;
43     }
44 }
```

We can implement the second method, `create_payers`, knowing that the corresponding csv file stores in the first row the name of the tax payer, in the second one its birth place and in the third one its address (note that a tax payer can appear more than once in the file if its address has changed; the last instance will be its current address). The method scans all the rows of `fields`. For any row, if the tax payer is not already in the vector `payers`, it adds him, otherwise it calls the method `Tax_Payer::update_address` (see the file "tax_payer.hpp").

```

46 void csv_manager::create_payers(std::vector<Tax_Payer> & payers)
47 {
48     for (const auto & row : fields)
49     {
50         int pos = find_payer(row[0], payers);
51         if(pos == -1){
52             payers.push_back(Tax_Payer(row[0],row[1],row[2]));
53         }
54         else{
55             Tax_Payer & p = payers[static_cast<size_t>(pos)];
56             p.update_address(row[2]);
57         }
58     }
59 }
```

Chapter 6

Standard Template Library - use of containers

6.1 Exercises

Exercise 1 Stock Quotes (*exam 01/09/2017*)

As in Exercise 1 of Chapter 2 (Section 2.2), we want to implement a program for the U.S. stock quotes market, in order to manage real-time stock quotes data on all public companies in the U.S.

The only difference with respect to the version we have seen before is that the `Stock_quote_archive` class must store stock quote data in a `std::map` and not it a `std::vector`.

Exercise 1 - Solution

The implementation of the class `StockQuotes` is exactly the same we had in Exercise 1 of Chapter 2 (Section 2.2), thus it is not reported here.

***** file `Stock_quote_archive.h` *****

```
1 #ifndef STOCKQUOTEARCHIVE_H
2 #define STOCKQUOTEARCHIVE_H
3
4 #include <map>
5
6 #include "Stock_quote.h"
7
8 class Stock_quote_archive
9 {
10     std::map<std::string, Stock_quote> stock_quotes;
11 public:
12     void add_stock_quote (const Stock_quote & s);
13     void add_last_sale_price (const std::string & stock_symbol, float value);
14     void print (void) const;
15 };
16
17 #endif //STOCKQUOTEARCHIVE_H
```

The class `Stock_quote_archive` stores the stock quote data in a `std::map`, using as keys the `std::strings` that represent the symbols of the stock quotes.

The method `add_stock_quote` (see lines 3 to 6 of the file "Stock_quote_archive.cpp" for the implementation) receives as parameter the `Stock_quote` object we want to add and it uses the method `insert` available for `std::maps` in order to save it in `stock_quotes`. In particular, it uses the function `std::make_pair`, passing as parameters the symbol of the `Stock_quote`s and the stock quote itself, in order to create a `std::pair<std::string, Stock_quote>`, which is the type stored in the `std::map<std::string, Stock_quote>` `stock_quotes`.

The method `add_last_sale_price` (see lines 8 to 16) relies on the corresponding method implemented for the `Stock_quote` class (see Exercise 1 of Chapter 2, Section 2.2). In particular, in lines 11 and 12, it uses the method `std::map::find` in order to check whether a `Stock_quote` with the symbol `stock_symbol` passed as parameter is stored in the map or not. If this is the case, namely if the iterator returned by `find` is not `stock_quotes.end()`, it calls the method `Stock_quote::add_last_sale_price` on this iterator in order to add the given value to the proper stock quote.

***** file `Stock_quote_archive.cpp` *****

```

1 #include "Stock_quote_archive.h"
2
3 void Stock_quote_archive::add_stock_quote (const Stock_quote & s)
4 {
5     stock_quotes.insert (std::make_pair (s.get_symbol(), s));
6 }
7
8 void Stock_quote_archive::add_last_sale_price (const std::string & stock_symbol,
9                                     float value)
10 {
11     std::map<std::string, Stock_quote>::iterator it =
12     stock_quotes.find (stock_symbol);
13
14     if (it != stock_quotes.end())
15         it->second.add_last_sale_price (value);
16 }
17
18 void Stock_quote_archive::print (void) const
19 {
20     std::map<std::string, Stock_quote>::const_iterator it = stock_quotes.cbegin();
21
22     while (it != stock_quotes.end())
23     {
24         it->second.print();
25         ++it;
26     }
27 }
```

For what concerns the complexity of the method

`Stock_quote_archive::add_last_sale_price`,

let Q be the number of stock quotes items within the `stock_quotes` map and let P be the number of items within the vector `last_prices` of the stock quote whose symbol is given as parameter. In the worst case scenario, `map::find()` has complexity $O(\log Q)$, while `vector::push_back()` has complexity $O(P)$, hence the worst case overall complexity is $O(P + \log Q)$.

Exercise 2 Instant messenger (*exam 23/01/2018*)

Design and implement a class for an instant messaging service. This class owns a data structure to store both messages and their sending times. Represent messages as `std::strings` and timestamps with an integral type. The service offers mainly three methods:

1. a procedure to send messages, which takes the sending time and text as arguments;
2. 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;
3. a function that returns all the messages whose content includes a given word provided as argument.

Consider as most common use case clients that remain online quite often, with frequent sends and receives (and this latter on recent messages), but rare searches based on message content. State the complexity of the implemented methods and motivate your design choices, particularly regarding data structures.¹

Exercise 2 - Solution

We need an associative container to keep track of the relation between messages and the relative timestamps, thus a `std::map` or a `std::unordered_map`. In particular, we will use the timestamp, represented by an `unsigned long`, as key and the text, represented by a `std::string`, as value of the container. Since one of the procedures we want to write allows to recover a collection of messages more recent than a given timestamp, it would be useful to have an ordered container, which makes this operation simpler. Therefore, a `std::map` is to be preferred over a `std::unordered_map`, because it remains automatically sorted and allows to retrieve the latest messages with an $O(\log N)$ search, where N is the number of elements in the map.

```
***** file instant_messenger.hh *****
```

```
1 #ifndef __INSTANT_MESSENGER__
2 #define __INSTANT_MESSENGER__
3
4 #include <list>
5 #include <map>
6 #include <string>
```

¹Some useful routines available in STL are listed below:

- `<map> std::map::find`
 `const_iterator find(const Key& key) const;`
- `<unordered_map> std::unordered_map::find`
 `iterator find(const Key& key); const_iterator find(const Key& key) const;`
- `<string> std::basic_string::find`
 `size_type find(const basic_string& str, size_type pos = 0) const;`
 `size_type find(const CharT* s, size_type pos, size_type count) const;`
 `size_type find(const CharT* s, size_type pos = 0) const;`
 `size_type find(CharT ch, size_type pos = 0) const;`
- `<map> std::map::lower_bound`
 `iterator lower_bound(const Key& key);`
 `const_iterator lower_bound(const Key& key) const;`
- `<map> std::map::upper_bound`
 `iterator upper_bound(const Key& key);`
 `const_iterator upper_bound(const Key& key) const;`

For the relative documentation, see www.cppreference.com or www.cplusplus.com

```

7
8 namespace im
9 {
10    class messenger
11    {
12        typedef std::map<unsigned long, std::string> container_type;
13
14        container_type m_data;
15
16    public:
17        typedef container_type::key_type key_type;
18        typedef container_type::mapped_type mapped_type;
19        typedef container_type::value_type value_type;
20
21        void send (key_type, const mapped_type &);
22        std::list<mapped_type> receive (key_type) const;
23        std::list<mapped_type> search (const std::string &) const;
24    };
25 }
26
27 #endif // __INSTANT_MESSENGER__

```

The method `send` (see lines 5 to 8 of the file "instant_messenger.cc" for the implementation) relies on the subscript operator available for `std::maps`. In particular, when we call `m_data[timestamp] = message;`

(as in line 7), if an element with key equal to `timestamp` is already stored in the map, its value is modified by the value of `message`; if there was not an element with the given key, it is created and it is added to the map.

The method `receive` (see lines 10 to 22) returns a `std::list` of messages (remeber that `messenger::mapped_type` is the type of the values stored in the map, namely a `std::string`). We choose to use a `std::list` instead of, for example, a `std::vector`, because we do not need random access and because, since the list is not a sequential container, the complexity of the method `push_back` (see line 18) is always constant. With a `std::vector` which contains, for example, N elements, however, the worst case complexity of the method `push_back` is $O(N)$ because we might need a reallocation (for a further discussion about reallocations, see).

In line 15, we call the method `std::map::lower_bound`, which returns a `const_iterator` to the first element in the map with key greater than or equal to the one passed as parameter (or `m_data.cend()` if no elements with key greater than or equal to `timestamp` are stored in the map).

The method `search` (see lines 24 to 40) receives as parameter the word we want to look for in the messages. It loops over all the pairs stored in the map (see lines 29 to 37) and, for each one of them, it calls the method `std::string::find` in order to search the given word in the message. In particular, notice that the method `std::string::find` looks for the first substring equal to the given character sequence; it returns the position of the first character of the found substring or `std::string::npos` if no such substring is found.

```
***** file instant_messenger.cc *****
1 #include "instant_messenger.hh"
2
3 namespace im
4 {
5     void messenger::send (key_type timestamp, const mapped_type & message)
6     {

```

```

7     m_data[timestamp] = message;
8 }
9
10 std::list<messenger::mapped_type>
11 messenger::receive (key_type timestamp) const
12 {
13     std::list<mapped_type> result;
14
15     for (container_type::const_iterator it = m_data.lower_bound (timestamp);
16          it != m_data.cend (); ++it)
17     {
18         result.push_back (it -> second);
19     }
20
21     return result;
22 }
23
24 std::list<messenger::mapped_type>
25 messenger::search (const std::string & word) const
26 {
27     std::list<mapped_type> result;
28
29     for (const container_type::value_type & pair : m_data)
30     {
31         const mapped_type & message = pair.second;
32
33         if (message.find (word) != mapped_type::npos)
34         {
35             result.push_back (message);
36         }
37     }
38
39     return result;
40 }
41 }
```

If N is the number of elements stored in the map, the complexity of `send` is $O(\log N)$, whilst `search` is $O(MNW)$, where M is the maximum length attained by any message and W the length of the searched word. Since we can reasonably expect $M \ll N$ to hold, `search` is fundamentally linear in N for our purposes.

For what concerns the method `receive`, being L the number of returned messages, its computational complexity can be expressed as $O(L + \log N)$. Indeed, $O(\log N)$ is the complexity of the method `std::map::lower_bound`, while $O(L)$ comes from the method `std::list::push_back` (which has a complexity of $O(1)$ and it is called L times if L messages satisfy the required condition).

If the timestamp argument is quite old, the complexity of the method `receive` is dominated by the linear contribution of the sequential access needed to build the returned list, which becomes $O(N)$ in the worst case. However, under the assumption that clients remain constantly online, we can expect only recent timestamps to be frequently accessed, thus keeping L a small value and yielding an expected $O(\log N)$ complexity in the average case.

Exercise 3 Counter Queue (*exam 19/02/2018*)

A well known supermarket chain, BZR, tasked you with the implementation of a piece of software for the management of queues at counters in their stores. After an environmentally responsible choice, BZR will install card readers at each counter, hence customers will not pick a number from the usual paper roll, but they will virtually enter a line via scanning their fidelity card.

The existing store management software represents customer's identifiers with **strings**. Your class has to provide three main methods:

1. the first method takes an ID as argument and adds the corresponding customer in line
2. the second method returns the ID of the next client to be served
3. the third method checks whether there is any client who is waiting.

Beware that customers must not be allowed to reserve more than one position in the queue and the system is expected to notify such failures to get in line. Your contract requires methods to be optimized for the **average case**.

In addition, state the complexity of all the implemented methods, both in the average and worst case, with a particular focus on the design choices you made with respect to data structures and algorithms.

At last, discuss at high level how you would change the project if BZR wanted to give precedence to premium clients and how this would affect the computational complexity.

Exercise 3 - Solution

We need a **std::queue** to keep track of customers' arrival order. Furthermore, we need a fast way to check who is in line, then we also use a **std::unordered_set** for its quick lookups. Hash tables are better than binary trees in the average case, which is why we prefer the unordered alternative over **std::set**.

***** file **counter_queue.h** *****

```
1 #ifndef _COUNTER_QUEUE_
2 #define _COUNTER_QUEUE_
3
4 #include <queue>
5 #include <string>
6 #include <unordered_set>
7
8 namespace supermarket
9 {
10    class counter_queue
11    {
12        std::queue<std::string> m_order;
13        std::unordered_set<std::string> m_in_line;
14
15    public:
16        bool
17        pick_number (const std::string & customer_id);
18
19        std::string
20        next_customer (void);
21 }
```

```

22     bool
23     empty (void) const;
24   };
25 }
26
27 #endif // _COUNTER_QUEUE_

```

The method `pick_number` (see lines 5 to 15 of file "counter_queue.cpp" for the implementation) relies on the methods `std::unordered_set::insert` and `std::queue::push`. Indeed, in line 11, it calls `insert`, which returns a `std::pair` containing an `iterator` (that points either to the newly inserted element or to an element with the same key, if it was already stored in the set) and a `bool` (which is `true` if the element is successfully inserted). Then, in line 13, the method checks if the new element is effectively added to the set and, in this case, it calls `std::queue::push` in order to append the new customer to the end of the queue.

The method `next_customer` (see lines 17 to 24) picks the first element of the queue, which is the next customer to serve, and it erases it both from the queue and from the set of customers in line.

***** file counter_queue.cpp *****

```

1 #include "counter_queue.h"
2
3 namespace supermarket
4 {
5     bool
6     counter_queue::pick_number (const std::string & customer_id)
7     {
8         typedef
9             std::pair<std::unordered_set<std::string>::iterator, bool>
10            insert_return_type;
11         const insert_return_type outcome = m_in_line.insert (customer_id);
12         const bool added_in_queue = outcome.second;
13         if (added_in_queue) m_order.push (customer_id);
14         return added_in_queue;
15     }
16
17     std::string
18     counter_queue::next_customer (void)
19     {
20         const std::string next = m_order.front ();
21         m_order.pop ();
22         m_in_line.erase (next);
23         return next;
24     }
25
26     bool
27     counter_queue::empty (void) const
28     {
29         return m_order.empty ();
30     }
31 }

```

By the choice of containers we have made, the methods `pick_number`, `next_customer`, and `empty` have $O(1)$ complexity on average, while, being N the number of customers in line,

`pick_number` and `next_customer` have complexity $O(N)$ in the worst case. In order to prioritize some premium customers over others, the `std::queue` should be substituted with a `std::priority_queue`, with some other minor changes. The main difference in terms of complexity is in `pick_number` and `next_customer`, which become $O(\log N)$ both in the average and worst case, due to the underlying heap.

Exercise 4 Streaming System (*exam 21/07/2017*)

You have to develop a micro-batch streaming system. In particular, you have to compute the average, minimum, and maximum values of the data samples received within a time window. A time window includes a fixed integer number of time slots and the data coming from each time slot are stored in a micro-batch. The time window is moved as the time elapses and old data are dropped according to the overlapping window paradigm. As an example, Figure 6.1 shows a sliding window including four time slots, hence four micro-batches.

The definition of the `MicroBatch` class is the following:

```

1  class MicroBatch
2  {
3      std::vector<double> samples;
4
5  public:
6      MicroBatch (const std::vector<double> & v)
7          : samples (v) {}
8
9      MicroBatch() {}
10
11     double get_min() const;
12     double get_max() const;
13     double get_sum() const;
14
15     unsigned get_n_samples() const;
16
17     void set_samples (const std::vector<double> & v);
18 };

```

In particular a `MicroBatch` object includes a varying number of samples received within a single time slot.

Provide the implementation of the `ContinuousStream` class, which includes the following methods:

1. The constructor, which receives as input the number of time slots.
2. The `get_min()`, `get_max()`, and `get_average()` methods, which return the minimum, maximum, and average of all the values currently stored in the time window.
3. The `add_batch` method that adds a new micro-batch in the time window, discarding the oldest.

Store the micro batches in a circular vector (see Figure 6.2) in a way that, when a new micro-batch is added in the time window, the oldest one is removed. In particular, store in the circular vector shared pointers to micro-batch objects. Moreover, since you can receive multiple requests to the `get` methods within a time slot, cache the minimum, maximum, and sum of the samples in a micro-batch in vectors and update their values within the `add_batch` method.

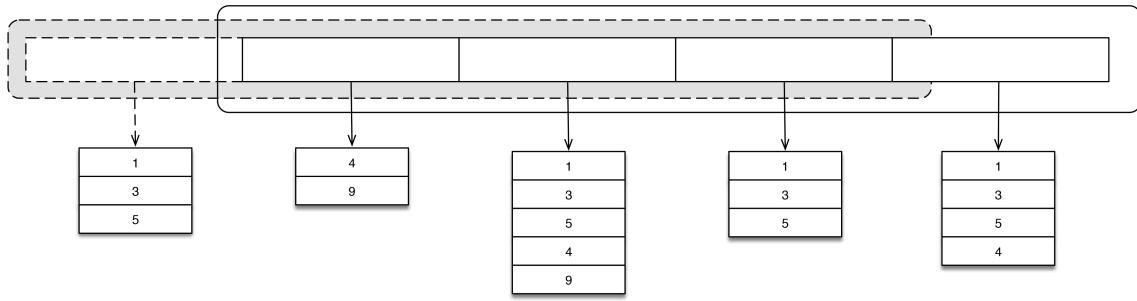


Figure 6.1: Streaming system with a four-slot time window.

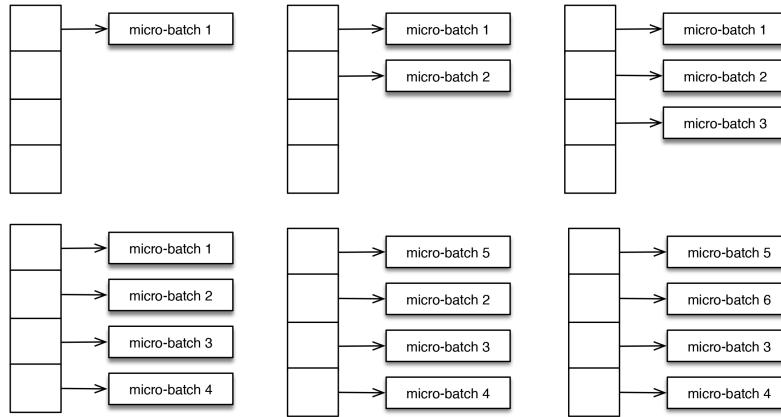


Figure 6.2: Circular vector data management.

Exercise 4 - Solution

The class `ContinousStream` stores a `std::vector` of shared pointers to `const MicroBatch` objects, three vectors that store, respectively, the minima, the maxima and the sums of the elements in every `MicroBatch` and an `unsigned` which represents the number of timestamps included in a time window. Moreover, it stores a `size_t`, which by default is 0, that stores the index of the next batch in which we will save a new element, possibly overwriting the oldest one (for instance, in the three examples of the first line of Figure 6.2, `next_batch` is equal to 1, 2 and 3, while in the ones of the second line it is equal to 0, 1 and 2, respectively). Notice that the choice to store in the vector `batches` shared pointers to `MicroBatches` allows to avoid duplicating objects.

Moreover, the `MicroBatches` are declared as `const` because a streaming system is an object from which we only read information, thus there is no reason to allow modifications of the batches. The fact that those objects are `const` allows us to cache the values of minima and maxima, which in principle could change if, in turn, the `MicroBatches` were not constant. Of course, since the batches are `const`, we cannot modify them through any method of the class.

***** file `ContinousStream.h` *****

```

1 #ifndef CONTINUOUSSTREAM_H
2 #define CONTINUOUSSTREAM_H
3
4 #include <memory>
5 #include <vector>
6

```

```

7 #include "MicroBatch.h"
8
9 class ContinousStream
10 {
11     std::vector< std::shared_ptr<const MicroBatch> > batches;
12
13     std::vector<double> mins;
14     std::vector<double> maxs;
15     std::vector<double> sums;
16
17     size_t next_batch = 0;
18
19     unsigned window_size;
20
21 public:
22     ContinousStream (unsigned time_window)
23         : window_size (time_window) {}
24
25     double get_min() const;
26     double get_max() const;
27     double get_average() const;
28
29     void add_batch (std::shared_ptr<const MicroBatch> b);
30 };
31
32 #endif // CONTINUOUSSTREAM_H

```

The methods `get_min` and `get_max` (see lines 5 to 8 and 10 to 13 of file "ContinousStream.cpp", respectively) return the minimum and the maximum element stored in the vectors `mins` and `maxs`. Notice that the functions `std::min_element` and `std::max_element` return an iterator pointing to the minimum or maximum element in the range passed as argument, thus we have to dereferenciate it in order to return a `double`.

The method `get_average` returns the average computed over all the `MicroBatches`. Since we want a global average and not the average per single `MicroBatch`, it is more convenient to cache in a vector (namely, `sums`) the sum of the elements stored in every `MicroBatch` and not the relative average.

Finally, the implementation of the method `add_batch` (see lines 29 to 52) is different depending on whether the window of `MicroBatches` is already complete or not. Indeed, in the first case (see the first line of Figure 6.2) we simply call `push_back` to add the new element to the vector `batches`, while in the second case we overwrite the element that was previously stored in position `next_batch` (i.e. the oldest element that was stored in the vector).

***** file ContinousStream.cpp *****

```

1 #include "ContinousStream.h"
2
3 #include <iostream>
4
5 double ContinousStream::get_min() const
6 {
7     return * std::min_element (mins.cbegin(), mins.cend());
8 }
9
10 double ContinousStream::get_max() const

```

```

11 {
12     return * std::max_element (maxs.cbegin(), maxs.cend());
13 }
14
15 double ContinousStream::get_average() const
16 {
17     double sum = 0.;
18     unsigned n_samples = 0;
19
20     for (size_t i = 0; i < sums.size(); ++i)
21     {
22         sum += sums[i];
23         n_samples += batches[i]->get_n_samples();
24     }
25
26     return sum / n_samples;
27 }
28
29 void ContinousStream::add_batch (std::shared_ptr<const MicroBatch> b)
30 {
31     // still creating the window
32     if (batches.size() < window_size)
33     {
34         batches.push_back (b);
35
36         sums.push_back (b->get_sum());
37         mins.push_back (b->get_min());
38         maxs.push_back (b->get_max());
39
40         next_batch = (next_batch + 1) % window_size;
41     }
42     else // replace one existing batch
43     {
44         batches[next_batch] = b;
45
46         sums[next_batch] = b->get_sum();
47         mins[next_batch] = b->get_min();
48         maxs[next_batch] = b->get_max();
49
50         next_batch = (next_batch + 1) % window_size;
51     }
52 }

```

Exercise 5 Erdos numbers (exam 12/07/2018)

Paul Erdős is well known for his prolific scientific production. Over time, his many academic collaborations led friends to devise Erdős numbers as a tribute. Somebody's *Erdős number* is the degree of separation between them and Erdős himself in the graph where nodes are scientific authors and undirected edges represent publications written in collaboration. Paul Erdős himself has number 0, his direct coauthors have 1, their own collaborators have 2, and so on. For example, Sandro Salsa's Erdős number is 3 because, as shown in Figure 6.3, there

is a path separating him from Paul Erdős with three hops.

Given a graph with mathematicians and their collaborations, Erdős numbers can be computed via Dijkstra's algorithm. This technique computes the shortest paths spanning tree rooted in a source node. In particular, using Paul Erdős as source node, the obtained distances are the relevant Erdős numbers.² In the following, you can find: i) the definition of the `author` class, and ii) an excerpt from Wikipedia that presents Dijkstra's algorithm. Building upon these, write a class that keeps track, through `std::shared_ptr`, of a collection of authors and computes Erdős numbers using Dijkstra's algorithm. Specifically: a) decide an appropriate data structure and motivate your choice, b) implement the member function that computes the spanning tree, whose prototype is

```
void erdos_calculator::compute_erdos_numbers (void)
```

and c) discuss the complexity of your implementation in terms of number of vertices (V) and edges (E) in the graph. Note that **you do not need to store shortest paths and return them as results**, but you only have to modify appropriately `m_erdos_number` members in `author` objects. Furthermore, for the sake of simplicity **assume** that the considered **graph** is **connected** and authors' **names** are **distinct**.

```
1  class author
2  {
3      private:
4          std::string m_name;
5          std::string m_affiliation;
6          std::vector< std::shared_ptr<author> > m_coauthors;
7          std::size_t m_erdos_number = std::numeric_limits<std::size_t>::max ();
8
9      public:
10         author (void) = delete;
11         explicit author (const std::string & name);
12         author (const std::string & name, const std::string & affiliation);
13
14         const std::string & get_name (void) const;
15
16         const std::string & get_affiliation (void) const;
17
18         const std::vector< std::shared_ptr<author> > & get_coauthors (void) const;
19
20         std::size_t get_erdos_number (void) const;
21
22         void set_affiliation (const std::string & z);
23         void add_coauthor (const std::shared_ptr<author> & );
24         void set_erdos_number (std::size_t);
25     };
26 }
27
28 #endif // AUTHOR_HH
```

Dijkstra's algorithm:³

²Actually they are upper bounds: if you leave out some mathematicians, thus cutting some paths, the solution might overestimate the affected authors.

³Excerpt extracted from https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=847620594

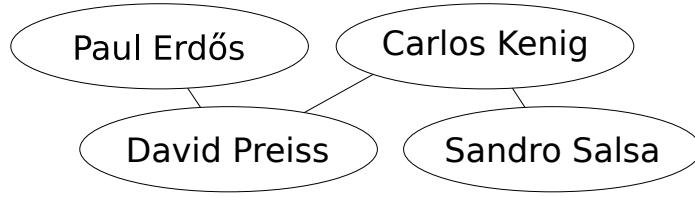


Figure 6.3: Collaboration path leading from Paul Erdős to Sandro Salsa

Let call the nodes from which we want to compute the distances as the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step according to the following steps:

1. Mark all nodes as unvisited and put all of them in the set of unvisited nodes.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set as current node the initial one.
3. Consider all the unvisited neighbors of the current node and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smallest one. For example, if the current node A has current distance equal to 6, and the edge connecting it with a neighbor B has length of 2, then the distance to B through A will be $6 + 2 = 8$. If the previous distance of B was larger than 8, then it is updated to 8, otherwise it keeps the current value.
4. When we are done considering all of the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when computing the shortest path among two nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when computing the shortest paths from a node of all the other nodes), then the algorithms finish. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new “current node”, and go back to step 3.

Exercise 5 - Solution

The class `erdos_calculator`, in `namespace mathematicians`, allows to compute the Erdős number of a given collection of authors. The data structure chosen to store the authors is a `std::map`, with the authors' names as keys and `std::shared_ptr<author>` as values.

A `std::map` is to be preferred over a `std::set` because the default relational operator for shared pointers compares their addresses and not the pointed objects. On the other side, since in the text it is specified that the authors' names are distinct, we can easily use them as keys for a `std::map`, avoiding the definition of a custom comparator for the set.

The choice is motivated also by the fact that, in real applications, it is always useful to be able to access directly an author, given his name.

Notice that, both in the class `author` and in the class `erdos_calculator`, we avoid every possible copy of `authors`, using shared pointers both to store the coauthors (see line 5 of the declaration of class `author` reported above) and to store the collection of authors of which we want to compute the Erdős number (see lines 15 and 18 of file "erdos_calculator.hh").

***** file erdos_calculator.hh *****

```

1 #ifndef ERDOS_CALCULATOR_HH
2 #define ERDOS_CALCULATOR_HH
3
4 #include <map>
5 #include <memory>
6 #include <string>
7
8 #include "author.hh"
9
10 namespace mathematicians
11 {
12     class erdos_calculator
13     {
14         public:
15             typedef std::map< std::string, std::shared_ptr<author> > container_type;
16
17         private:
18             container_type m_all_authors;
19
20         static container_type::const_iterator find_closest_to_erdos (const
21             container_type &);
22
23         public:
24             void add_author (std::shared_ptr<author>);
25
26             void compute_erdos_numbers (void);
27
28             const container_type & get_all_authors (void) const;
29     };
30 }
31 #endif // ERDOS_CALCULATOR_HH

```

The class `erdos_calculator` implements three `public` methods: `add_author` (see lines 24 and 25 of file "erdos_calculator.hh") receives as parameter a shared pointer to `author` and adds it to the map `m_all_authors`; `compute_erdos_numbers` (see lines 27 and 28) implements the Dijkstra's algorithm in order to compute the Erdős numbers of all the authors stored in `m_all_authors`; finally, `get_all_authors` (lines 30 and 31) returns the map with the collection of all the authors.

The `private` method `find_closest_to_erdos` (lines 20 and 21), in turn, is a `static` method which receives a `std::map` of shared pointers to `authors` and returns a constant iterator to the one whose Erdős number is the lowest.

***** file `erdos_calculator.cc` *****

```

1 #include "erdos_calculator.hh"
2
3 #include <limits>
4
5 namespace mathematicians
6 {
7     constexpr std::size_t largest_erdos = std::numeric_limits<std::size_t>::max ();
8
9     void erdos_calculator::add_author (std::shared_ptr<author> new_author)

```

```

10  {
11      const std::string & name = new_author -> get_name ();
12      m_all_authors.insert ({name, new_author});
13  }
14
15  erdos_calculator::container_type::const_iterator erdos_calculator::
16      find_closest_to_erdos (const container_type & authors)
17  {
18      container_type::const_iterator minimum_distance_author = authors.cbegin ();
19
20      for (auto it = ++ authors.cbegin (); it != authors.cend (); ++it)
21      {
22          const auto & current_author = it -> second;
23
24          if (current_author -> get_erdos_number () <
25              minimum_distance_author -> second -> get_erdos_number ())
26          {
27              minimum_distance_author = it;
28          }
29      }
30
31      return minimum_distance_author;
32  }
33
34  void erdos_calculator::compute_erdos_numbers (void)
35  {
36      container_type unvisited_authors = m_all_authors;
37
38      for (const auto & pair : m_all_authors)
39      {
40          if (pair.first == "Erdos, Paul")
41              pair.second -> set_erdos_number (0);
42          else
43              pair.second -> set_erdos_number (largest_erdos);
44      }
45
46      bool has_connections = true;
47
48      while (not unvisited_authors.empty () and has_connections)
49      {
50          const auto current = find_closest_to_erdos (unvisited_authors);
51          std::shared_ptr<author> current_author = current -> second;
52          unvisited_authors.erase (current);
53
54          if (current_author -> get_erdos_number () < largest_erdos)
55          {
56              const std::size_t neighbors_erdos_number =
57                  current_author -> get_erdos_number () + 1;
58
59              for (auto coauthor : current_author -> get_coauthors ())
60              {
61                  if (neighbors_erdos_number < coauthor -> get_erdos_number ())

```

```

61         coauthor -> set_erdos_number(neighbors_erdos_number);
62     }
63 }
64 else has_connections = false;
65 }
66 }
67
68 const erdos_calculator::container_type & erdos_calculator::get_all_authors (void)
69 {
70     const
71     {
72         return m_all_authors;
73     }
74 }
```

The method `find_closest_to_erdos` (see the implementation in lines 16 to 33 of file "erdos_calculator.cc") loops over all the shared pointers to `authors` stored in the container `authors` passed as parameter and it saves in `minimum_distance_author` the iterator of the one whose Erdős number is the lowest.

The method `compute_erdos_numbers` (see lines 35 to 69) implements the Dijkstra's algorithm presented above, with the assumption that the length of every edge of the authors' graph is equal to 1. In line 38, it initializes the `std::map` of unvisited authors, which at the beginning contains all the authors stored in `m_all_authors`. Then, it loops over all the authors stored in `m_all_authors` and it sets their Erdős number to `largest_erdos` (a `constexpr` initialized in line 7 with the maximum value that can be represented by a `std::size_t`). Of course, the Erdős number of Erdős himself is 0.

In lines 50 to 68, while the `std::map` of unvisited authors is not empty, the method selects the unvisited author which is closest to Erdős and it erases him from `unvisited_authors`. If his Erdős number is less than `largest_erdos`, the method sets the Erdős number of all his coauthors to the minimum between their original Erdős number and `neighbors_erdos_number`. If, in turn, the Erdős number of `current` is equal to `largest_erdos`, this means that he has no connections with the other authors (notice that this situation cannot happen in our context, where we assume the authors' graph to be connected).

This basic version of the algorithm has $O(V^2)$ complexity, due to the two nested loops. In full details, the `while` loop has complexity $O(V \cdot (V + \log V + S))$, because we loop over all the authors and, at every iteration, we scan all the unvisited authors in the method `find_closest_to_erdos`, we remove the current one (the method `std::map::erase` has complexity $O(\log V)$) and we check his S coauthors. Since the `for` loop of lines 40 to 46 adds a term $O(V)$, we end up with an overall cost of $O(V + V \cdot (V + \log V + S))$. For a realistic collaboration graph certainly $S \ll V$ holds, hence that term can be neglected and we finally find $O(V^2)$.

For what concerns the cost complexity of the Dijkstra's algorithm, we can notice that the choice of a `std::list` would give a slightly better result, since we always have to scan all the container (and therefore we do not get advantages by the fact that, using a `std::map` with N elements, we can recover any of them in $O(\log N)$ using its key), but the cost of the method `std::list::erase` is only $O(1)$. This, however, does not affect the overall result, since the cost of the method `erase` has to be added to other terms that are linear in V . Moreover, using a `std::list` over a `std::map`, we would need to take care of the fact that we do not want to duplicate authors, which is automatically done by `std::maps` since the keys are unique.

Exercise 6 National Healthcare

National healthcare system needs to provide doctors with an application to manage their

patients and the relative prescriptions.

Each medicine has a compatibility range defined. Each time a patient is prescribed with a new medicine, the application checks its compatibility with the medicines that patient already takes; in particular, two medicines are compatible if their compatibility ranges have some overlap (e.g., (1,3) has overlap with (2,4)).

Thus, the patient class keeps a history of taken medicines in a `std::multimap` that map dates to medicines and it also provides a method declared as:

```
void patient::add (const medicine& m)
```

to do the following:

1. add new medicines and remove from the history of the patient all the old medicines which are not compatible with the new one.
2. add the new medicine to the prescriptions of the patient.
3. remove all the prescriptions whose proposed period of use is over.
4. removes all the expired prescriptions.

The class `patient` also has a method

```
std::list<medicine> active_prescriptions (void) const;
```

that returns all the active prescriptions of a patient.

Class `medicine` has two boolean methods:

```
bool medicine::compatible (const medicine&) const;
```

that checks the compatibility of `this` and `m`;

```
bool medicine::medicine_expiry (void) const;
```

to control the termination date of the prescription.

Exercise 6 - Solution

***** file date.h *****

```
1 #ifndef DATE_HH_
2 #define DATE_HH_
3
4 ///STD include
5 #include <string>
6
7 class Date
8 {
9 private:
10     int day = 1, month = 1, year = 1972;
11
12 public:
13     Date (void) = default;
14     Date (const Date&) = default;
15
16     //format (year-month-day) needs to be checked
17     Date (const std::string& str);
18 }
```

```

19 const std::string Print (void) const;
20
21 // comparing two dates
22 friend bool operator< (const Date&, const Date&);
23 };
24
25 bool operator< (const Date&, const Date&);
26
27 #endif

```

In the class `Date`, we implement a constructor that receives as parameter a `std::string` storing the new date in the format "year-month-day" (notice that we should check if the given date is admissible, as we did, for example, in Exercise 2.3 of Section 2.3, Chapter 2).

We need to overload the `operator<` in order to be able, in class `medicine`, to check whether the expiry date of a given medicine is already elapsed and in order to be able to use, in class `patient`, a `Date` as key of a `std::multimap` (which is an ordered container).

***** file `date.cpp` *****

```

1 //Header include
2 #include "date.h"
3
4 //std include
5 #include <iostream>
6 #include <sstream>
7
8 //STL include
9 #include <vector>
10
11 Date::Date (const std::string& str)
12 {
13     std::vector<int> vect;
14     std::istringstream ss(str);
15     int i;
16
17     while (ss >> i)
18     {
19         vect.push_back(i);
20         if (ss.peek() == '-') ss.ignore();
21     }
22     day = vect[2];
23     month = vect[1];
24     year = vect[0];
25 }
26
27 const std::string Date::Print (void) const
28 {
29     std::ostringstream converter;
30     converter << day << "-" << (month) << "-" << year;
31     return converter.str();
32 }
33
34 bool operator< (const Date &d1, const Date &d2)
35 {

```

```

36 if (d1.year < d2.year)
37     return true;
38 else if (d1.year == d2.year)
39 {
40     if (d1.month < d2.month)
41         return true;
42     else if (d1.month == d2.month)
43     {
44         if (d1.day < d2.day)
45             return true;
46     }
47 }
48 return false;
49 }
```

***** file medicine.h *****

```

1 #ifndef MEDICINE_HH
2 #define MEDICINE_HH
3
4 #include "date.h"
5
6 class medicine
7 {
8     std::string name;
9     Date expiration_date;
10    Date proposed_end_of_use;
11
12    /* assuming that each medicine has an assigned range. Other medicines
13       are considered compatible if they have an overlapping range,
14       and incompatible otherwise*/
15    std::pair <unsigned, unsigned> compatibility_range;
16
17 public:
18    /*the constructor*/
19    medicine(const std::string& s, const Date& e_d, const Date& p_d, unsigned p1,
20             unsigned p2):
21        name(s), expiration_date(e_d), proposed_end_of_use(p_d),
22        compatibility_range(std::make_pair(p1,p2))
23    {}
24
25    /*returns the compatibility range */
26    std::pair <unsigned, unsigned> get_compatibility_range (void) const;
27
28    /*returns the name of the medicine */
29    std::string get_name (void) const;
30
31    /*returns the expiration date */
32    Date get_expiration_date (void) const;
33
34    /*returns the proposed_end_of_use date */
35    Date get_proposed_end_of_use_date (void) const;
```

```

36  /*returns true if m has overlapping compatibility range with this */
37  bool compatible (const medicine &m) const;
38
39  /*returns true if the medicine is expired */
40  bool medicine_expiry (void) const;
41 };
42
43 #endif

```

An object of class `medicine` is characterized by its name, two `Dates` (namely, the expiry date and the proposed end of use) and a `std::pair` of `unsigned` representing the compatibility range (in particular, two `medicines` are compatible if their compatibility ranges overlap).

***** file `medicine.cpp` *****

```

1 #include "medicine.h"
2 #include <ctime>
3 #include <iostream>
4 #include <string>
5
6 std::pair <unsigned, unsigned> medicine::get_compatibility_range (void) const
7 {
8     return compatibility_range;
9 }
10
11 std::string medicine::get_name (void) const
12 {
13     return name;
14 }
15
16 Date medicine::get_expiration_date (void) const
17 {
18     return expiration_date;
19 }
20
21 Date medicine::get_proposed_end_of_use_date (void) const
22 {
23     return proposed_end_of_use;
24 }
25
26 bool medicine::compatible (const medicine &m) const
27 {
28     return !(m.compatibility_range.first > this->compatibility_range.second ||
29             m.compatibility_range.second < this->compatibility_range.first);
30 }
31
32 bool medicine::medicine_expiry (void) const
33 {
34     /*creates a date from now and compares it with expiration date*/
35     time_t now = time(0);
36     struct tm tstruct;
37     char *buf = new char[80];
38     tstruct = *localtime(&now);
39     strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);

```

```

40     std::string s (buf, buf+10);
41     Date d = Date(s);
42     delete []buf;
43     return (expiration_date < d);
44 }
```

The method `medicine::medicine_expiry` (see lines 32 to 44) relies on the utilities available in header `<ctime>` in order to generate a `Date` corresponding to the current date in which the program is run and it checks whether the expiry date of `this` is less than the current date.

***** file patient.h *****

```

1 #ifndef PATIENT_HH
2 #define PATIENT_HH
3
4 #include "medicine.h"
5 #include <iostream>
6 #include <string>
7 #include <list>
8 #include <map>
9
10 class patient
11 {
12     std::string name;
13
14     //maps proposed_end_of_use date to medicine
15     //it is a multimap since several medicines can
16     //share the same proposed_end_of_use date
17     std::multimap <Date, medicine> medicines;
18
19 public:
20     patient (const std::string &n): name(n) {}
21     void add (const medicine& m);
22     std::list<medicine> active_prescriptions (void) const;
23 };
24
25 #endif
```

The `patient` class stores the name of the patient and its medical history, represented by a `std::multimap` of `medicines` whose key is the proposed end of use date (in particular, we use a multimap and not a map because several medicines can share the same proposed end of use date).

The class implements a method to add a `medicine` to the medical history of the patient and a method to return the list of active prescriptions.

***** file patient.cpp *****

```

1 #include "patient.h"
2
3 void patient::add (const medicine& m)
4 {
5     //removing medicines whose proposed period of use is over
6     auto beg = medicines.begin(),
7         lb = medicines.lower_bound(m.get_proposed_end_of_use_date());
8     auto new_beg = medicines.erase(beg,lb);
```

```

9
10 //removing incompatible or expired medicine
11 auto i = new_beg;
12 while (i != medicines.end()) {
13     if (!m.compatible (i->second)) i = medicines.erase(i);
14     else if (i->second.medicine_expiry()) i = medicines.erase(i);
15     else i++;
16 }
17
18 //add new medicine if it is not already expired
19 if (!m.medicine_expiry())
20     medicines.insert(std::make_pair(m.get_proposed_end_of_use_date(), m));
21 }
22
23 std::list<medicine> patient::active_prescriptions (void) const
24 {
25     std::list<medicine> l;
26     for (auto i = medicines.begin(); i != medicines.end(); i++)
27         l.push_back(i->second);
28     return l;
29 }
```

The method `patient::add` (see lines 3 to 21 of the file "patient.cpp") receives as parameter the medicine `m` we want to add to the medical history of the patient. First of all, the method erases from the multimap `medicines` all the ones whose proposed use period is over. In particular (see line 7) it uses the method `std::multimap::lower_bound` to get an iterator to the first element in the map whose proposed end of use date is greater than or equal to the one of `m`. Then, it calls the method `std::multimap::erase` to delete from `medicines` all the elements between `medicines.begin()` and this lower bound.

Having removed those medicines, the method removes all the ones that are not compatible with `m` or that are already expired.

Finally (see lines 19 and 20), if `m` is not expired, the method adds it to the medical history of the patient.

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

```

1 #include "medicine.h"
2 #include "patient.h"
3 #include "date.h"
4
5 #include <iostream>
6 #include <time.h>
7 #include <string>
8
9 using namespace std;
10 int main(int argc, char const *argv[]) {
11
12     Date d1("2018-7-8");
13
14     Date d2("2018-7-14");
15
16     patient p("Aldo");
17
18     //create a new medicine
```

```

19 medicine m1 = medicine ("m1", d1, d2, 2, 3);
20 medicine m2 = medicine ("m2", d1, d2, 1, 6);
21 medicine m3 = medicine ("m3", d1, d2, 0, 7);
22 medicine m4 = medicine ("m4", d1, d2, 8, 9);

23
24 //add them to medicines of a patient
25 p.add(m1);
26 p.add(m2);
27 p.add(m3);

28
29 std::list<medicine> l = p.active_prescriptions();
30 for (auto i = l.begin(); i != l.end(); i++) {
31     std::cout << i->get_name() << '\n';
32 }

33
34 std::cout << "adding an element with incompatible range (m4)" << '\n';
35 p.add(m4);
36 //when m4 is added, others are deleted since they are not compatible with m4
37 l = p.active_prescriptions();
38 for (auto i = l.begin(); i != l.end(); i++) {
39     std::cout << i->get_name() << '\n';
40 }
41 std::cout << "adding an already expired medicine (m5)" << '\n';
42 medicine m5 = medicine ("m5", Date("2014-12-12"), d2, 5, 10);
43 p.add(m5);
44 l = p.active_prescriptions();
45 for (auto i = l.begin(); i != l.end(); i++) {
46     std::cout << i->get_name() << '\n';
47 }

48
49 return 0;
50 }
```

Note: if you try to run the program, remember to change the dates `d1` and `d2` in lines 12 and 14 of file "main.cpp": since the method `medicine::medicine_expiry` compares the expiry date with the current date in which the program is run, `d1` and `d2` must be set accordingly in order to get the expected output.

Exercise 7 Student Search

Write a program that manages the list of students of a class and their grades. Students are uniquely identified by their ID, represented as an `unsigned int`. Create a function that can find and return the student identified by a given id.

As a starting point, you are provided with an implementation that uses a `std::vector` to collect students. Also a `binary_search` function has been implemented.

Implement your solution once by using a `std::set` to collect students, and then using a map (unordered).

Moreover, modify the initial vector-based solution in a way that you can rely on the STL implementation of `binary_search`.

The initial code you must rely on is provided below:

```
***** file timing.h *****
```

```

1 #ifndef TIMING_HH
2 #define TIMING_HH
3
4 #include <chrono>
5
6 namespace timing
7 {
8     typedef std::chrono::time_point<std::chrono::system_clock> time_point;
9
10    void elapsed_between (const time_point & start, const time_point & finish);
11 }
12
13 #endif // TIMING_HH

```

In the `namespace timing` we provide a function (implemented using the utilities available in `<ctime>`) to compute the time taken by the program in order to perform a specific operation. For example, we will use it to measure the performance of the binary search algorithm (see line 34 of the file "search.cpp").

***** file timing.cpp *****

```

1 #include <ctime>
2 #include <iostream>
3
4 #include "timing.h"
5
6 namespace timing
7 {
8     void elapsed_between (const time_point & start, const time_point & finish)
9     {
10         std::chrono::duration<double> elapsed_seconds = finish - start;
11         std::time_t end_time = std::chrono::system_clock::to_time_t (finish);
12         std::cout << "finished computation at " << std::ctime (&end_time)
13             << "elapsed time: " << elapsed_seconds.count() << " s"
14             << std::endl;
15     }
16 }

```

***** file Student.h *****

```

1 #ifndef STUDENT_H_
2 #define STUDENT_H_
3
4 #include <vector>
5
6 namespace university
7 {
8     class Student
9     {
10         unsigned id;
11         std::vector<unsigned> grades;
12
13     public:
14         Student (unsigned, const std::vector<unsigned> &);

```

```

15     explicit Student (unsigned);
16
17     const std::vector<unsigned> & getGrades (void) const;
18     unsigned getId (void) const;
19
20     void setGrades (const std::vector<unsigned> & );
21     void setId (unsigned);
22
23     void print (void) const;
24 };
25 }
26
27 #endif /* STUDENT_H_ */

```

An object of class `Student` is characterized by an `id` and a `std::vector` of grades. Notice that the constructor declared in line 15 of file "Student.h" receives only one parameter and therefore it is marked as `explicit` in order to avoid possible implicit conversions between `unsigned` and `Student` objects.

***** file `Student.cpp` *****

```

1 #include <iostream>
2
3 #include "Student.h"
4
5 namespace university
6 {
7     Student::Student (unsigned id, const std::vector<unsigned> & grades)
8         : id (id), grades (grades) {}
9
10    Student::Student (unsigned id): id (id), grades () {}
11
12    const std::vector<unsigned> & Student::getGrades() const
13    {
14        return grades;
15    }
16
17    unsigned Student::getId (void) const
18    {
19        return id;
20    }
21
22    void Student::setGrades (const std::vector<unsigned> & new_grades)
23    {
24        grades = new_grades;
25    }
26
27    void Student::setId (unsigned new_id)
28    {
29        id = new_id;
30    }
31
32    void Student::print (void) const
33    {

```

```

34     std::cout << id << ":\n";
35
36     for (unsigned grade : grades)
37         std::cout << grade << " ";
38
39     std::cout << std::endl;
40 }
41 }
```

***** file search.h *****

```

1 #ifndef SEARCH_HH
2 #define SEARCH_HH
3
4 #include <vector>
5
6 #include "Student.h"
7
8 namespace university
9 {
10     bool binary_search (const std::vector<university::Student> & stud_vec, unsigned
11                     stud_id);
12 }
13 #endif // SEARCH_HH
```

***** file search.cpp *****

```

1 #include <iostream>
2
3 #include "search.h"
4 #include "timing.h"
5
6 namespace university
7 {
8     bool binary_search (const std::vector<university::Student> & stud_vec, unsigned
9                     stud_id)
10    {
11        timing::time_point start = std::chrono::system_clock::now();
12
13        std::vector<university::Student>::const_iterator begin = stud_vec.cbegin(),
14                     end = stud_vec.cend(),
15                     // original midpoint
16                     mid = begin + (end - begin) / 2;
17        unsigned n_iter = 1;
18
19        while (begin != end and mid->getId() != stud_id)
20        {
21            /* end is meant to be invalid, so in both cases
22             * we are ignoring mid at the following iteration
23             */
24            if (stud_id < mid->getId())
25                end = mid;
26            else
```

```

26     begin = mid + 1;
27
28     mid = begin + (end - begin) / 2;
29     ++n_iter;
30 }
31
32 std::cout << "Number of Iterations " << n_iter << "\n";
33
34 timing::time_point finish = std::chrono::system_clock::now();
35 timing::elapsed_between (start, finish);
36
37 if (mid != stud_vec.cend() and stud_id == mid->getId())
38 {
39     return true;
40 }
41 else
42 {
43     return false;
44 }
45 }
46 }
```

The function `binary_search` (see lines 8 to 45 of the file "search.cpp") implements the binary search algorithm to look for a specific `Student`, identified by his id, within a `std::vector` of `Students`. Notice that the class `Student` is contained in the `namespace university`, thus we have to use the scope operator `university::` any time we refer to an object of that type. The function uses iterators in order to scan the vector looking for the required `Student`. In particular, we can use the binary search algorithm when we can assume the `std::vector<university::Student>` to be sorted by the id of the `Students`. If this is true, indeed, we can start the search from the `Student` in the middle of the vector (see line 15) and move to the left or to the right according to the order of the `Students'` identification numbers.

***** file RandomStudentGenerator.h *****

```

1 #ifndef RANDOMSTUDENTGENERATOR_H_
2 #define RANDOMSTUDENTGENERATOR_H_
3
4 #include <random>
5
6 namespace university
7 {
8     class Student;
9
10    class RandomStudentGenerator
11    {
12        unsigned next_id;
13
14        std::mt19937 generator;
15        std::uniform_int_distribution<unsigned> grade_distribution;
16        std::uniform_int_distribution<unsigned> number_distribution;
17
18    public:
19        RandomStudentGenerator (void);
20        RandomStudentGenerator (unsigned first_id, unsigned lowest_grade,
```

```

21         unsigned highest_grade, unsigned n_grades,
22         unsigned seed = 0u);
23
24     Student nextStudent (void);
25 }
26
27
28 #endif /* RANDOMSTUDENTGENERATOR_H_ */
```

The class `RandomStudentGenerator` is used to automatically generate a collection of students, assigning them a vector of random grades. It stores an `unsigned` that represents the next id to be assigned to a student, a pseudo-random numbers generator (the type `std::mt19937` represents a generator of pseudo-random 32-bit unsigned integers, with an almost uniform distribution in the range $[0, 2^{19937} - 1]$) and two `std::uniform_int_distribution<unsigned>`, i.e. two random number distributions that produce integer values in a given range, according to an uniform discrete distribution.

***** file `RandomStudentGenerator.cpp` *****

```

1 #include <vector>
2
3 #include "RandomStudentGenerator.h"
4 #include "Student.h"
5
6 namespace university
7 {
8     RandomStudentGenerator::RandomStudentGenerator (void)
9     : RandomStudentGenerator (1, 18, 30, 20) {}
10
11    RandomStudentGenerator::RandomStudentGenerator (unsigned first_id,
12                                                 unsigned lowest_grade,
13                                                 unsigned highest_grade,
14                                                 unsigned n_grades,
15                                                 unsigned seed)
16    : next_id (first_id), generator (seed),
17      grade_distribution (lowest_grade, highest_grade),
18      number_distribution (1, n_grades) {}
19
20    Student RandomStudentGenerator::nextStudent (void)
21    {
22        const unsigned n_grades = number_distribution (generator);
23        std::vector<unsigned> grades (n_grades);
24
25        for (unsigned & grade : grades)
26            grade = grade_distribution (generator);
27
28        return Student (next_id++, grades);
29    }
30}
```

The class `RandomStudentGenerator` provides two constructors: the second one, namely the one whose implementation can be found in lines 11 to 18 of the file "RandomStudentGenerator.cpp", receives as parameters the id that must be assigned to the first student, the range

of admissible grades, the maximum number of grades to be generated for every student and the seed through which initialize the `std::mt19937` generator (remember that, since every generator produces numbers that are only pseudo-random and not truly random, in general we use seeds as entropy sources: two generators initialized with the same seed should produce exactly the same sequence of pseudo-random numbers any time we execute the program, which can be useful for debugging⁴).

The first constructor, the one with no parameters that is implemented in lines 8 and 9 of file "RandomStudentGenerator.cpp", calls the other one in such a way that the first id number is 1 and that at most 20 grades between 18 and 30 are generated for every student. The seed is not passed as parameter because it has a default value (equal to 0, see line 22 of file "RandomStudentGenerator.h"; to write `0u` instead of simply 0 explicitly tells that the type of 0 must be `unsigned int`).

The method `nextStudent` (see lines 20 to 29 of file "RandomStudentGenerator.cpp") generates a new `Student` together with a vector of random grades.

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

```

1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 #include "RandomStudentGenerator.h"
6 #include "search.h"
7 #include "Student.h"
8 #include "timing.h"
9
10 typedef std::vector<university::Student> stud_vec_type;
11
12 int main (void)
13 {
14     constexpr unsigned n_students = 1000;
15     stud_vec_type stud_vec;
16
17     university::RandomStudentGenerator rs;
18
19 // create the vector with students sorted by id
20 for (unsigned i = 0; i < n_students; ++i)
21 {
22     university::Student random_stud = rs.nextStudent();
23     stud_vec.push_back (random_stud);
24 }
25
26 const unsigned stud_ok_id = stud_vec[stud_vec.size() / 2].getId();
27 const unsigned stud_ko_id = stud_vec.back().getId() + 1;
28
29 std::cout << "*****initial binary search on vector*****\n";
30 std::cout << (university::binary_search (stud_vec, stud_ko_id) ? "Found!\n" : "Not
   found\n");
31 std::cout << (university::binary_search (stud_vec, stud_ok_id) ? "Found!\n" : "Not
   found\n");
32

```

⁴Mind that this property might not hold if you execute the program on different architectures

```

33     return 0;
34 }
```

In the `main` function, first of all we generate 1000 random `Students` (see lines 14 to 24 of file "main.cpp"), then we select two id numbers: the first one (line 26) is the `id` of a student who is really stored in the vector `stud_vec`, while the second one (line 27) is an identification number that does not correspond to any student. Notice that we can extract those `id` in this way because the method `RandomStudentGenerator::nextStudent` generates `Students` with an increasing `id`, thus the vector `stud_vec` turns out to be sorted by the identification number of the students.

Finally, in lines 30 and 31 we use the `binary_search` function to look for the selected `id`, which allows to compare the time taken by the algorithm when it actually finds the element it is looking for or not.

Exercise 7 - Solution

We list below only the files or part of files that must be modified in order to write the solution.

***** file Student.cpp *****

```

42     bool operator < (const Student & lhs, const Student & rhs)
43     {
44         return lhs.getId() < rhs.getId();
45     }
```

First of all, if we want to use a `std::set` to store the `Students`, since the sets are ordered containers, we need to overload the `operator<` for the class `Student`. As it usually happens for all the relational operators, it is implemented as a free function (see the discussion in Question 1, Section 4.2 of Chapter 4). It compares the identification numbers of the given `Students` and it returns `true` if the first is smaller than the second.

Notice that having the `operator<` enables us also to check if two `Students` are equal, even if we do not have the `operator==`. Indeed, in order to check whether `Student s1` is equal to `Student s2` is enough to check whether both `s1 < s2` and `s2 < s1` return `false`.

Vice versa, we do not need to modify the class `Student` in order to be able to use a `std::unordered_map`. Indeed, we will use the identification numbers of the `Students` as keys and, since they are `unsigned`, both the `operator=` and the `hash` function are already available.

***** file search.cpp *****

```

47     bool set_search (const std::set<university::Student> & stud_set, unsigned stud_id)
48     {
49         timing::time_point start = std::chrono::system_clock::now();
50         const university::Student s(stud_id);
51         const std::set<university::Student>::const_iterator it = stud_set.find (s);
52
53         timing::time_point finish = std::chrono::system_clock::now();
54         timing::elapsed_between (start, finish);
55
56         return it != stud_set.cend();
57     }
58
59     bool map_search (const std::unordered_map<unsigned, university::Student> &
60                      stud_map, unsigned stud_id)
61     {
```

```

61     timing::time_point start = std::chrono::system_clock::now();
62     const std::unordered_map<unsigned, university::Student>::const_iterator it =
63         stud_map.find (stud_id);
64
65     timing::time_point finish = std::chrono::system_clock::now();
66     timing::elapsed_between (start, finish);
67
68     return it != stud_map.cend();
69 }
```

Other than the `binary_search` function we had before, we need to add two functions, namely `set_search` and `map_search`, that we will use when the container in which we store the Students is either a `std::set` or a `std::unordered_map` (we have above the implementation of those functions in file "search.cpp"; of course, the corresponding declaration must be added in "search.h").

In particular, both these functions rely, respectively, on the method `find` available for `std::set` and `std::unordered_map` (see lines 51 and 62-63, respectively). Notice, however, that, in `set_search`, since the keys and the values in `std::sets` are the same, we need to pass to the method `find` not only the `stud_id`, but an object of type `Student`, which is instantiated in line 50 with the proper identification number.

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

```

1 #include <algorithm>
2 #include <chrono>
3 #include <iostream>
4 #include <string>
5 #include <set>
6 #include <unordered_map>
7 #include <vector>
8
9 #include "RandomStudentGenerator.h"
10 #include "search.h"
11 #include "Student.h"
12 #include "timing.h"
13
14 typedef std::vector<university::Student> stud_vec_type;
15 typedef std::set<university::Student> stud_set_type;
16 typedef std::unordered_map<unsigned, university::Student> stud_map_type;
17
18 int main (void)
19 {
20     constexpr unsigned n_students = 1000;
21     stud_vec_type stud_vec;
22     stud_set_type stud_set;
23     stud_map_type stud_map;
24
25     university::RandomStudentGenerator rs;
26
27     for (unsigned i = 0; i < n_students; ++i)
28     {
29         university::Student random_stud = rs.nextStudent();
30         stud_vec.push_back (random_stud);
31         stud_set.insert (random_stud);
```

```

32     stud_map.insert (std::make_pair (random_stud.getId(), random_stud));
33 }
34
35 const unsigned stud_ok_id = stud_vec[stud_vec.size() / 2].getId();
36 const unsigned stud_ko_id = stud_vec.back().getId() + 1;
37
38 std::cout << "*****initial binary search on vector*****\n";
39 std::cout << (university::binary_search (stud_vec, stud_ko_id) ? "Found!\n" : "Not
    found\n");
40 std::cout << (university::binary_search (stud_vec, stud_ok_id) ? "Found!\n" : "Not
    found\n");
41
42
43 std::cout << "*****set search*****\n";
44 std::cout << (university::set_search (stud_set, stud_ko_id) ? "Found!\n" : "Not
    found\n");
45 std::cout << (university::set_search (stud_set, stud_ok_id) ? "Found!\n" : "Not
    found\n");
46
47 std::cout << "*****map search*****\n";
48 std::cout << (university::map_search (stud_map, stud_ko_id) ? "Found!\n" : "Not
    found\n");
49 std::cout << (university::map_search (stud_map, stud_ok_id) ? "Found!\n" : "Not
    found\n");
50
51 timing::time_point start, finish;
52 std::cout << "*****STL::binary_search*****\n";
53 university::Student s1(stud_ko_id);
54 start = std::chrono::system_clock::now();
55 std::cout << (std::binary_search (stud_vec.cbegin (), stud_vec.cend(), s1) ? "Found!\n"
    : "Not found\n");
56 finish = std::chrono::system_clock::now();
57 timing::elapsed_between (start, finish);
58
59 university::Student s2(stud_ok_id);
60 start = std::chrono::system_clock::now();
61 std::cout << (std::binary_search (stud_vec.cbegin (), stud_vec.cend(), s2) ? "Found!
    !\n" : "Not found\n");
62 finish = std::chrono::system_clock::now();
63 timing::elapsed_between (start, finish);
64
65 return 0;
66 }
```

In the `main` function, lines 31 and 32, we call the specific method `insert` to add the new student generated by `RandomStudentGenerator` to the set `stud_set` or the unordered map `stud_map` respectively, then we extract two id as before (lines 35 and 36) and we perform the search using the different containers in order to compare the resulting performance. Finally, in lines 51 to 63 we perform again the search on the vector `stud_vec` using the function `std::binary_search` provided by the STL instead of the one implemented in "search.cpp". As we can see in the output reported below, there is an increasing in speed using a `std::unordered_map` over a `std::vector` or a `std::set`. Indeed, both the computational cost of the binary search algorithm and of the method `std::set::find` are $O(\log N)$, where N is the

number of elements stored in the container, while the one of the method `std::unordered_map` `::find` is $O(1)$. In order to get more evident differences, you can try to run the same program with a larger number of students.

```
*****initial binary search on vector*****
Number of Iterations 10
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 0.00040552 s
Not found
Number of Iterations 1
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 0.000402834 s
Found!
*****set search*****
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 4.563e-06 s
Not found
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 3.457e-06 s
Found!
*****map search*****
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 8.61e-07 s
Not found
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 9.68e-07 s
Found!
*****STL::binary_search*****
Not found
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 4.144e-06 s
Found!
finished computation at Fri Jul 13 10:10:47 2018
elapsed time: 3.437e-06 s
```

Exercise 8 Dataframe (*exam 07/07/2017*)

A *DataFrame* is a 2-dimensional labeled data structure with columns of the same type. You can think of it like a spreadsheet, a table, or a Python `NamedTuple` (with the constraint to have column data of the same type!). The nice property of a *DataFrame* is that you can access its columns by name and apply to every column functions like `mean`, `max`, etc. Figure 6.4 shows, as an example, a *DataFrame* including two columns storing temperatures and humidity for a weather forecast application.

You have to provide a class definition for a `DataFrame` class, storing elements of type `double`. In particular, provide the implementation of the following methods:

1. A constructor that receives one single `string` as parameter. Using spaces as word separators, it will initialize the column names with the words contained in the argument. For example, the `DataFrame` in Figure 6.4 can be obtained passing "`temperatures` `humidity`".
2. A `set_column` method that, given a `vector` of values of type `double` and a column

weather_conditions_dataframe	
temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7

Figure 6.4: A weather application DataFrame.

name, replaces the entire column content.

3. The method `set_element_at` that, given a column name, an index i , and a value of type `double`, updates the i -th value of the column.
4. `get_mean()`, which returns the mean of a given column.
5. `select_equal` that, given a column name and a value, returns a new `DataFrame` including only the set of rows for which the column equals the value. For instance, `select_equal ("temperatures", 31.4)` called on the `DataFrame` in Figure 6.4 would yield a new one with both columns, but only the second row.

Take particular care to error conditions, e.g., access to a wrong column or element index out of range. Moreover, check that all the columns have the same number of rows (which will be known only when creating the first column). For the constructor implementation you can rely on the `split()` function:

```
vector<string> split(const string & s, char d)
```

which returns the names of columns in `s` separated by the delimiter `d`. In other words, you are not required to implement `split()` yourselves.

Discuss the worst case complexity of the setter methods.

Exercise 8 - Solution

For this exercise, two different solutions are provided: the first one uses only `std::vectors` as data structures, while the second one, which is the one required during the exam, relies on a `std::unordered_maps`. Moreover, the first solution is written without taking care of possible errors which may occur if, for example, we try to access a column which is not stored in the dataframe.

***** First solution:

As it is specified above, this solution uses two `std::vectors` in order to store the dataframe. In particular, a `std::vector<std::string>`, identified as a `key_container`, is used to store the names of the columns, while every column is represented through a `std::vector<double>`, identified as `mapped_type`, so that the whole dataframe is stored as a `std::vector<std::vector<double>>`.

Notice that all the types used for elements and containers are hidden in a couple of user-defined names (see lines 12 to 16 of file "DataFrame.hpp"). This is useful because it allows to easily change types and containers (provided that the methods are compatible) without having to rewrite the whole code.

```
***** file DataFrame.hpp *****
```

```
1 #ifndef DATAFRAME_HH
2 #define DATAFRAME_HH
```

```

3
4 #include <sstream>
5 #include <iostream>
6 #include <string>
7 #include <vector>
8
9 class DataFrame
10 {
11 public:
12     typedef std::vector<std::string> key_container;
13
14     typedef key_container::value_type key_type;
15     typedef std::vector<double> mapped_type;
16     typedef mapped_type::size_type size_type;
17
18 private:
19     key_container df_keys;
20     std::vector<mapped_type> df_values;
21     bool added_first_column;
22
23     // return the names of columns in s separated by delim
24     key_container split (const key_type & s, char delim) const;
25
26     // return the index of column key, or df_keys.size () if not found
27     size_type look_up (const key_type & key) const;
28
29 public:
30     explicit DataFrame (const key_type & c_names);
31     explicit DataFrame (const key_container & names);
32
33     const mapped_type & get_column (const key_type & column_name) const;
34
35     double get_element_at (const key_type & column_name,
36                           size_type index) const;
37
38     void set_element_at (const key_type & column_name, size_type index,
39                          double value);
40
41     double get_mean (const key_type & column_name) const;
42
43     // add a new column with data
44     void set_column (const key_type & column_name,
45                      const mapped_type & column_data);
46
47     // return a copy of the DataFrame with rows i such
48     // that "c_name[i] == value"
49     DataFrame select_equal (const key_type & c_name,
50                           double value) const;
51
52     void print (void) const;
53 };

```

```
55 #endif // DATAFRAME_HH
```

The class `DataFrame` stores the vector of keys `df_keys`, the vector of columns `df_values` and a `bool` which is `true` if we have already added the first column to the dataframe (see line 21). It implements two `private` methods:

- `split` (line 24), which receives a `std::string` and a `char` and it returns a `std::vector<std::string>` built by splitting the string received as parameter using the given `char` as delimiter.
- `look_up` (line 27), which receives a key as parameter and returns the index in `df_values` which corresponds to the column identified by the given key (or `df_values.size()` if the key is not stored in the dataframe).

Moreover, it implements two constructors: the one which receives a `std::string`, that was required in the text, and another one which receives a vector of keys and which is actually used to initialize the dataframe (see the declarations in lines 30 and 31, respectively). Notice that, since they receive a single parameter, they are both marked as `explicit`.

In addition, there are seven `public` methods. Other than the ones required by the exercise, there are two getters, which return, respectively, a column or an element in a specific position, and a method which can be used to print out the dataframe (see lines 33, 35 and 52 for the declarations).

```
***** file DataFrame.cpp *****
```

```
1 #include "DataFrame.hpp"
2
3 DataFrame::DataFrame (const key_type & c_names)
4   : DataFrame (split (c_names, ' '))
5 {}
6
7 DataFrame::DataFrame (const key_container & names)
8   : df_keys (names), df_values (df_keys.size ()),
9     added_first_column (false)
10 {}
11
12 DataFrame::key_container
13 DataFrame::split (const key_type & s, char delim) const
14 {
15   key_type word;
16   key_container keys;
17   std::istringstream columns (s);
18   while (std::getline (columns, word, delim))
19     keys.push_back (word);
20   return keys;
21 }
22
23 DataFrame::size_type
24 DataFrame::look_up (const key_type & key) const
25 {
26   size_type index = 0;
27   while (index < df_keys.size () and df_keys[index] != key)
28     ++index;
29   return index;
30 }
```

```

31
32 const DataFrame::mapped_type &
33 DataFrame::get_column (const key_type & column_name) const
34 {
35     const size_type index = look_up (column_name);
36     return df_values[index];
37 }
38
39 double
40 DataFrame::get_element_at (const key_type & column_name,
41                           size_type index) const
42 {
43     const size_type column_index = look_up (column_name);
44     return df_values[column_index][index];
45 }
46
47 void
48 DataFrame::set_element_at (const key_type & column_name, size_type index,
49                           double value)
50 {
51     const size_type column_index = look_up (column_name);
52     df_values[column_index][index] = value;
53 }
54
55 double
56 DataFrame::get_mean (const key_type & column_name) const
57 {
58     const size_type index = look_up (column_name);
59     const mapped_type & column = df_values[index];
60     double sum = 0.;
61     for (double v : column)
62         sum += v;
63     return sum / column.size ();
64 }
65
66 void
67 DataFrame::set_column (const key_type & column_name,
68                       const mapped_type & column_data)
69 {
70     const size_type index = look_up (column_name);
71
72     if (added_first_column)
73     {
74         mapped_type & column = df_values[index];
75
76         for (size_type i = 0; i < column_data.size (); ++i)
77             column[i] = column_data[i];
78     }
79     else
80     {
81         added_first_column = true;
82         df_values[index] = column_data;

```

```

83
84     for (mapped_type & column : df_values)
85         column.resize (column_data.size ());
86     }
87 }
88
89 DataFrame
90 DataFrame::select_equal (const key_type & c_name,
91                         double value) const
92 {
93     DataFrame result (df_keys);
94
95     const size_type index = look_up (c_name);
96     const mapped_type & column = df_values[index];
97
98     std::vector<size_type> indices;
99
100    // select rows indices satisfying the selection criterion
101    for (size_type j = 0; j < column.size (); ++j)
102        if (column[j] == value)
103            indices.push_back (j);
104
105    for (size_type i = 0; i < df_keys.size (); ++i)
106    {
107        mapped_type values;
108        const key_type & current_name = df_keys[i];
109        const mapped_type & current_column = df_values[i];
110
111        for (size_type j : indices)
112            values.push_back (current_column[j]);
113
114        result.set_column (current_name, values);
115    }
116
117    return result;
118 }
119
120 void
121 DataFrame::print (void) const
122 {
123     for (size_type i = 0; i < df_keys.size (); ++i)
124     {
125         std::cout << df_keys[i] << " :: ";
126
127         for (double v : df_values[i])
128             std::cout << v << " ";
129
130         std::cout << std::endl;
131     }
132 }
```

The first constructor (see lines 3 to 5), delegates the second one to initialize all the members of the class, passing to it the vector of keys returned by the method `split`. The second

constructor, in turn (see lines 7 to 10), initializes the vector of keys through `names`, sets the `bool added_first_column` to `false` and it initializes `df_values` as an empty vector with the size of `df_keys` (the number of columns we will have in the dataframe is equal to the number of keys).

The `private` method `look_up` (see lines 23 to 30) receives a key as parameter. It loops over the keys stored in `df_keys` and it returns the index which corresponds to the given one. This index will be used when we have to access a specific column in the dataframe because it links the name of a specific key to the position of the corresponding column (see for example the method `get_column`).

Notice that, since `size_type` is a type defined within the class, when we want to use it as the return type of a method we have to use the scope operator, i.e. to specify `Dataframe::size_type` (see line 23).

The method `get_column` (see lines 32 to 37) calls `look_up` in order to get the index corresponding to the given key and it returns the column which is stored in `df_values` at that index. Notice that the method does not check whether the given key is actually stored in the vector of keys. If this is not the case, the method `look_up` returns `df_keys.size()` and therefore line 36 turns out to have an undefined behavior, because it tries to access an element which is not stored in the vector.

The methods `get_element_at` and `set_element_at` (see lines 39 to 45 and 47 to 53, respectively) simply call `look_up` in order to get the index of the required column and then they rely on the `operator[]` of `std::vectors` in order to access the element at the proper index.

The method `get_mean` (lines 55 to 64) computes and returns the average of the values in a given column. Notice that, in line 59, we extract from the dataframe the column whose average we want to compute. The type we use is a constant reference to `mapped_type`; by doing this, we avoid to create an useless copy of the given column (which is always a good practice in order to save memory, since we do not know how many elements it contains). The `const` qualifier is needed because we are in a constant method and therefore operations which could possibly lead to modifications in the members of the class are not allowed. If, instead of using a reference, we write, in line 59:

```
mapped_type column = df_values[index];
```

i.e. we create a copy of the required column, we could drop the `const` qualifier without getting compiler errors, because the fact that we create a copy prevents the dataframe to be modified by following operations. However, since even the new vector `column` is used only to read elements (and therefore it is not modified), it can be better to write:

```
const mapped_type column = df_values[index];
```

The method `set_column` (see lines 66 to 87) acts differently when the first column has already been added to the dataframe and when this is not the case. In particular, in the first case (see lines 72 to 78) it extracts from the dataframe the required column and it loops over all the elements in order to set the values specified by `column_data`. In the second case, in turn, it sets the column which corresponds to the given key to be equal to `column_data` and then it resizes all the columns of the dataframe. Indeed, we want all the columns of the dataframe to have the same number of rows; therefore, once we know the size of the first column, we can prepare all the table, so that we avoid possible reallocations of the data structure when we add all the other columns.

Finally, the method `select_equal` (lines 89 to 118) returns a new `DataFrame` whose elements in column `c_name` are equal to `value`. First of all, a new `DataFrame` is initialized, passing as parameter `df_keys` because it is intended to have exactly the same keys of the former one. In lines 95 to 103, the method extracts the column whose name is given by `c_name` and it loops over this column in order to select the indices of the elements whose value is equal to `value`. Once we have this vector of indices (lines 105 to 115), we loop over all the elements

of `df_keys`. For any key, we extract the corresponding column of `df_values` (line 109) and we initialize a vector (named `values`) with the elements of `current_column` whose index is stored in `indices`. Finally, we set `values` as column of the dataframe `result` at the position specified by the current key.

Notice that selecting the indices of the elements that are stored in the column `c_name` and whose value is `value` allows us to reduce the number of operations we perform, because in lines 111 and 112 we do not need to loop over all the elements of `current_column`, but we can directly select the elements we need.

Complexity: in the worst case, `get_column` and `set_column` have complexity $O(N + M)$, where N is the number of columns and M is the number of rows. The get and set of a single element have complexity $O(N)$.

In both cases, $O(N)$ comes from the `look_up` method and therefore it cannot be dropped even in the average case (unless the key we are looking for is stored at the very beginning of the dataframe).

***** Second solution:

As previously mentioned, the second solution uses as data structure for the dataframe a `std::unordered_map`, which has `std::strings` as keys (the names of the columns) and `std::vector<double>` as values (the columns themselves). Notice that we choose a `std::unordered_map` instead of a `std::map` because we do not need the column to have a specific order and `std::unordered_maps` guarantee a constant complexity, on average, to access a given column (while `std::maps` have an average complexity of $O(\log N)$).

The type definitions in the file "DataFrame.h" have been modified accordingly (see lines 14 to 19).

In this case, the provided solution takes also care of the possible errors we might have if, for example, we try to access an element which is not stored in the dataframe.

The class `DataFrame` stores the map `df_data`, the `bool added_first_column`, which is `true` if we have already added the first column to the dataframe (see line 22 and 23) and `n_rows`, which stores the number of rows of the dataframe. This last variable, that was not used in the first solution, can be useful to check directly if the number of elements in a new column is equal to the number of elements we have in all the other columns of the dataframe.

***** file DataFrame.h *****

```

1 #ifndef DATAFRAME_H
2 #define DATAFRAME_H
3
4 #include <iostream>
5 #include <limits>
6 #include <sstream>
7 #include <string>
8 #include <unordered_map>
9 #include <vector>
10
11 class DataFrame
12 {
13 public:
14     typedef std::unordered_map<std::string, std::vector<double>> df_type;
15
16     typedef df_type::key_type key_type;
17     typedef df_type::value_type value_type;
18     typedef df_type::mapped_type mapped_type;

```

```

19  typedef mapped_type::size_type size_type;
20
21 private:
22   df_type df_data;
23   bool added_first_column;
24   size_type n_rows;
25
26   // return the names of columns in s separated by delim
27   std::vector<key_type> split (const key_type & s, char delim) const;
28
29   // returns true if column_name is valid (included in the constructor list)
30   bool check_column_name (const key_type & column_name) const;
31
32   // set a column data checking only right number of rows
33   void set_column_data (const key_type & column_name,
34                         const mapped_type & column_data);
35
36 public:
37   explicit DataFrame (const key_type & c_names);
38   DataFrame();
39
40   mapped_type get_column (const key_type & column_name) const;
41
42   double get_element_at (const key_type & column_name,
43                          size_type index) const;
44
45   void set_element_at (const key_type & column_name, size_type index,
46                         double value);
47
48   double get_mean (const key_type & column_name) const;
49
50   // set the values of an existing column
51   void set_column (const key_type & column_name,
52                     const mapped_type & column_data);
53
54   // add a new column with data
55   void add_column (const key_type & column_name,
56                     const mapped_type & column_data);
57
58   // return a copy of the DataFrame with rows i such
59   // that "c_name[i] == value"
60   DataFrame select_equal (const key_type & c_name,
61                         double value) const;
62
63   void print (void) const;
64 };
65
66 #endif //DATAFRAME_H

```

The class implements three **private** methods:

- `split` (line 27), which behaves exactly as it is described in the first solution.
- `check_column_name` (line 30), which receives a string as parameter and it returns

true if that string corresponds to the name of one column of the dataframe. Notice that, while this method can be useful to check the existence of a column, the method `look_up` that we had in the first solution is not required any more: since `df_data` is a `std::unordered_map`, we can access directly each column by its name.

- `set_column_data` (line 33 and 34), which receives the name of a column and the column itself and inserts it in the dataframe checking if it has the correct number of rows.

Moreover, the class implements two constructors (the one which receives a `std::string`, that was required in the text, and the one with no parameters) and eight `public` methods. Other than the ones required by the text, we have the two getters (see lines 40 and 42), which return, respectively, a column or an element in a specific position, and a method to print the full dataframe.

Notice that there are two different `public` methods, `set_column` (lines 51 and 52) and `add_column` (lines 55 and 56): the first one is intended to be used when we want to modify the values of an existing column, while the second when we want to add a new column to the dataframe.

***** file `DataFrame.cpp` *****

```

1 #include "DataFrame.h"
2
3 DataFrame::DataFrame (const key_type & c_names)
4   : DataFrame ()
5 {
6   std::vector<key_type> columns_names = split (c_names, ' ');
7
8   for (const key_type & name : columns_names)
9     df_data[name];
10 }
11
12 DataFrame::DataFrame (void)
13   : added_first_column (false), n_rows (0)
14 {}
15
16 std::vector<DataFrame::key_type>
17 DataFrame::split (const key_type & s, char delim) const
18 {
19   key_type word;
20   std::vector<key_type> v;
21   std::istringstream columns (s);
22
23   while (std::getline (columns, word, delim)) v.push_back (word);
24
25   return v;
26 }
27
28 bool
29 DataFrame::check_column_name (const key_type & column_name) const
30 {
31   return df_data.find (column_name) != df_data.end();
32 }
```

```

34 void
35 DataFrame::set_column_data (const key_type & column_name,
36                           const mapped_type & column_data)
37 {
38     if (! added_first_column)
39     {
40         n_rows = column_data.size();
41         added_first_column = true;
42         for (df_type::iterator it = df_data.begin(); it != df_data.end(); it++)
43         {
44             (it->second).resize(n_rows);
45         }
46         df_data[column_name] = column_data;
47     }
48     else
49     {
50         // for next assignments check the column has the same number of rows
51         if (n_rows == column_data.size())
52             df_data[column_name] = column_data;
53         else
54         {
55             std::cerr << "Error, " << column_name
56                 << " has a different number of rows"
57                 << std::endl;
58         }
59     }
60 }
61
62 DataFrame::mapped_type
63 DataFrame::get_column (const key_type & column_name) const
64 {
65     if (check_column_name (column_name))
66         return df_data.at (column_name);
67     else
68     {
69         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
70         return std::vector<double> ();
71     }
72 }
73
74 double
75 DataFrame::get_element_at (const key_type & column_name,
76                            size_type index) const
77 {
78     if (check_column_name (column_name))
79         if (index < n_rows)
80             return df_data.at (column_name)[index];
81         else
82         {
83             std::cerr << "Error, index out of bound" << std::endl;
84             return std::numeric_limits<double>::quiet_NaN();
85         }

```

```

86     else
87     {
88         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
89         return std::numeric_limits<double>::quiet_NaN();
90     }
91 }
92
93 void
94 DataFrame::set_element_at (const key_type & column_name,
95                           size_type index, double value)
96 {
97     if (check_column_name (column_name))
98     {
99         if (index < n_rows)
100            df_data[column_name][index] = value;
101        else
102            std::cerr << "Error, index out of bound" << std::endl;
103        else
104            std::cerr << "Error, " << column_name << " is unknown" << std::endl;
105    }
106
107 double
108 DataFrame::get_mean (const key_type & column_name) const
109 {
110     if (check_column_name (column_name))
111     {
112         double sum = 0.;

113         for (double d : df_data.at (column_name))
114             sum += d;

115         return sum / n_rows;
116     }
117     else
118     {
119         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
120         return std::numeric_limits<double>::quiet_NaN();
121     }
122 }
123 }

124 void
125 DataFrame::set_column (const key_type & column_name,
126                        const mapped_type & column_data)
127 {
128     if (check_column_name (column_name))
129         set_column_data (column_name, column_data);
130     else
131         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
132 }
133

134 void
135 DataFrame::add_column (const key_type & column_name,
136                        const mapped_type & column_data)

```

```

138 {
139     if (! check_column_name (column_name))
140         set_column_data (column_name, column_data);
141     else
142         std::cerr << "Error, " << column_name
143             << " is already included in the DataFrame"
144             << std::endl;
145 }
146
147 DataFrame
148 DataFrame::select_equal (const key_type & c_name,
149                         double value) const
150 {
151     if (! check_column_name (c_name))
152     {
153         std::cerr << "Error, " << c_name << " is unknown" << std::endl;
154         return DataFrame();
155     }
156
157     std::vector<size_type> indexes; // solution indexes
158     const mapped_type & relevant_column = df_data.at (c_name);
159
160     // select rows indexes satisfying the selection criterion
161     for (size_type i = 0; i < n_rows; ++i)
162         if (relevant_column[i] == value)
163             indexes.push_back (i);
164
165     DataFrame result;
166
167     for (const value_type & element : df_data)
168     {
169         mapped_type v; //column values
170         v.reserve (indexes.size ());
171
172         // select proper column values
173         for (size_type idx : indexes)
174             v.push_back (element.second[idx]);
175
176         result.add_column (element.first, v);
177     }
178
179     return result;
180 }
181
182 void
183 DataFrame::print (void) const
184 {
185     for (const value_type & element : df_data)
186     {
187         std::cout << element.first << " :: ";
188
189         for (const double & d : element.second)

```

```

190     std::cout << d << " ";
191
192     std::cout << std::endl;
193 }
194 }
```

The first constructor (see lines 3 to 10) calls the **private** method `split` in order to get the vector of column names, passing as parameters the string `c_names` and a space as delimiter. Then, it relies on the `operator[]` of `std::unordered_maps` in order to generate the empty columns of the new dataframe. In line 4, it delegates the constructor with no parameters (see the implementation in lines 12 and 13) in order to initialize `added_first_column` to `false` and the number of rows to 0.

The method `set_column_data` (see lines 34 to 60) has a different behavior whether the first column has already been added or not. In particular, in case we are adding the first column (lines 38 to 47), it sets the value of `n_rows` equal to the size of the vector `column_data`, it loops over all the dataframe in order to resize the columns with the proper number of rows (see the corresponding method in the first solution) and, finally, it adds the given column to the dataframe. When, in turn, `added_first_column` is `true` (see lines 48 to 59), if the size of the new column is equal to `n_rows`, it is added to the dataframe, otherwise an error message is printed.

The method `set_column_data` will be called both by `set_column` (see lines 125 to 133) and by `add_column` (see lines 135 to 145). This is possible, without having to modify the implementation, because it relies on the `operator[]` of `std::unordered_maps`, which can be used both to modify the value corresponding to an existing key and to add a new element if the given key is not stored in the container.

The method `get_column` (see lines 62 to 72) returns the `std::vector<double>` corresponding to the given key if it exists, an empty vector otherwise. Notice that the corresponding method in the first solution was designed to return the vector by `const&`. This is not possible if we want to return an empty vector when the required column is not in the dataframe, since returning a reference to an element initialized in a function results in an invalid reference.

Both in `get_column` method and in `get_element_at` method (see lines 74 to 91), which returns the element stored in the given position or `NaN` if it does not exist, we use the method `std::unordered_map::at` to access the given column. We cannot use the `operator[]`, as we do in line 80 with the vector selected by `df_data.at()`, because both `get_elem` and `get_element_at` are declared as `const` (see the discussion in Question 1 of Section 6.2).

The method `get_mean` (see lines 106 to 123) computes and returns the mean of a given column (or `NaN` if the column is not in the dataframe).

Finally, the method `select_equal` (lines 147 to 180) returns a new `DataFrame` whose elements in column `c_name` are equal to `value`. First of all (see lines 157 to 163), it extracts the column whose name is given by `c_name` and it loops over this column in order to select the indices of the elements whose value is equal to `value`. Once we have this vector of indices, we instantiate a new `DataFrame`, we loop over all the elements of `this` and, for any column, we create a vector with the elements whose index is stored in `indexes`. Finally, we add this vector as a new column of the dataframe `result`.

Notice that having selected the indices of the elements that are stored in the column `c_name` and whose value is `value` allows us to reduce the number of operation we perform, because in lines 173 and 174 we do not need to loop over all the elements of the column `element.second`, but we can directly select the elements we need.

Complexity: in the worst case, `get_column` and `set_column` have complexity $O(N + M)$, where N is the number of columns and M is the number of rows. Indeed, in the very worst case all columns are stored in a single bucket (and the bucket is implemented at least as a `forward_list`) and therefore you pay $O(N)$ to access a given column. Then, when you have the column, you have to store or read all M elements and you pay $O(M)$. If you consider the

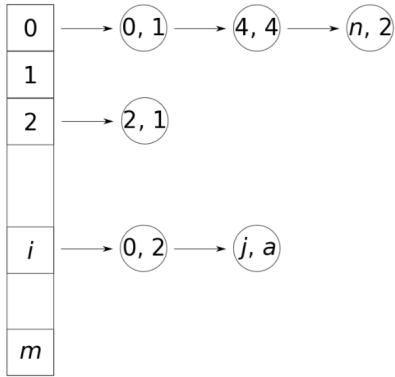


Figure 6.5: LIL representation of a sparse matrix.

average case, however, to access the column you pay $O(1)$, but then you still have to store or read M elements (and hence complexity is $O(M)$). The get and set of a single element have worst-case complexity $O(N)$. If the *average* case is considered, `get_column` and `set_column` have complexity $O(M)$, while the complexity for accessing single elements is $O(1)$. This is due to the fact that accessing an element in an `unordered_map` (both through the `operator[]` and through the method `at`) has complexity $O(1)$ in the average case, while it is linear in container size (i.e. it is $O(N)$) in the worst case.

Exercise 9 Sparse Matrix

Consider a sparse matrix class, in particular the simple LIL representation, which consists of a vector of lists, where every list represents a row. Since the matrix is sparse, all the zero-valued entries are not represented, whilst the nonzero entry a_{ij} appears in the i -th list as the pair (j, a) (see Figure 6.5).

The class provides some type names: `size_type` is an unsigned integral type, while `value_type` should be a `double`.

Provide an implementation of the following methods:

- `void set (size_type i, size_type j, value_type a);`
- `value_type get (size_type i, size_type j) const;`

State the complexity of the implemented methods for an $m \times n$ matrix. What would the complexity be if the outer container were a `std::list`? How could you change the data structure in order to improve the complexity of these methods?

Exercise 9 - Solution

The `lil_sparse` class stores a `std::vector<std::list<std::pair<size_type, double>>>`, where the type `double` is hidden under the user-defined name `value_type`.

***** file `lil_sparse.hpp` *****

```

1 #ifndef __LIL_SPARSE__
2 #define __LIL_SPARSE__
3
4 #include <list>
5 #include <utility>
6 #include <vector>

```

```

7
8 namespace numeric
9 {
10    class lil_sparse
11    {
12        public:
13            typedef double value_type;
14            typedef std::size_t size_type;
15
16        private:
17            typedef std::list<
18                std::pair<const std::size_t, value_type>> inner_list;
19            typedef inner_list::iterator inner_iterator;
20            typedef inner_list::const_iterator const_inner_iterator;
21            typedef std::vector<inner_list> data_structure;
22
23            data_structure nonzero;
24
25            size_type m_rows;
26            size_type m_columns;
27
28            inner_iterator
29            search (size_type, size_type);
30            const_inner_iterator
31            search (size_type, size_type) const;
32
33        public:
34            explicit lil_sparse (size_type rows = 1,
35                                size_type columns = 0);
36
37            size_type
38            rows (void) const {return m_rows;}
39            size_type
40            columns (void) const {return m_columns;}
41
42            void
43            set (size_type, size_type, value_type);
44
45            value_type
46            get (size_type, size_type) const;
47
48    }; /* end of class declaration */
49
50}
51
52 #endif

```

The class stores `m_rows` and `m_columns`, which are the number of rows and the number of columns of the matrix. The constructor of the class (see lines 34 and 35 of file "lil_sparse.hpp" for the declaration and lines 5, 6 of file "lil_sparse.cpp" for the definition) receives as parameters the number of rows and columns of the matrix. Notice that both parameters have a default value, which means that the constructor can be called even without passing anything as parameter; in this case, since the default value for the number of rows is 1, a single row-

vector will be created. The default value for the number of columns is 0 because it can be incremented dynamically by adding new elements.

Notice that the implementation presented below is based on the fact that the vector of lists has fixed size (i.e. we have to decide *a priori* the number of rows the sparse matrix), while the size of the lists themselves can grow dinamically as soon as we decide to add a new element in a certain row (and therefore the number of elements per column is not fixed). This is due to the fact that to change the size of a vector, while keeping the elements in a certain order (which is something we need to preserve the order of the rows in the matrix), is an expensive operation, while it has lower complexity in case of lists. For what concerns the choice of different containers and the complexity of the methods, see the discussion at the end of the answer.

The class implements the **private** method **search**, both in the constant and in the non-constant version, that returns an iterator to the element stored in a specific position. If the element is not stored in the matrix, **search** returns an iterator to the following element (or **nonzero.end()** if we reached the end of the row).

Moreover, it implements two methods to read the number of rows and columns in the matrix and two methods to get or set an element in a specific position.

***** file **lil_sparse.cpp** *****

```

1 #include "lil_sparse.hpp"
2
3 namespace numeric
4 {
5     lil_sparse::lil_sparse (size_type rows, size_type columns)
6         : nonzero (rows), m_rows (rows), m_columns (columns) {}
7
8     lil_sparse::inner_iterator
9     lil_sparse::search (size_type i, size_type j)
10    {
11        inner_iterator it = nonzero[i].begin ();
12        while (it != nonzero[i].end () && it -> first < j) ++it;
13        return it;
14    }
15
16    lil_sparse::const_inner_iterator
17    lil_sparse::search (size_type i, size_type j) const
18    {
19        const_inner_iterator it = nonzero[i].cbegin ();
20        while (it != nonzero[i].cend () && it -> first < j) ++it;
21        return it;
22    }
23
24    void
25    lil_sparse::set (size_type i, size_type j, value_type a)
26    {
27        inner_iterator it = search (i, j);
28        if (it != nonzero[i].end () && it -> first == j)
29        {
30            it -> second = a;
31        }
32        else
33        {

```

```

34     nonzero[i].insert (it, std::make_pair (j, a));
35 }
36 }
37
38 lil_sparse::value_type
39 lil_sparse::get (size_type i, size_type j) const
40 {
41 // If you can't find an entry in nonzero, then it's zero.
42 value_type result = 0.;
43 const_inner_iterator it = search (i, j);
44 if (it != nonzero[i].cend () && it -> first == j)
45 {
46     result = it -> second;
47 }
48 return result;
49 }
50
51 }

```

The method `set` (see the implementation in lines 24 to 36) can be used either to modify the value of an element which is already stored in the matrix or to insert a new element in a given row and column. In particular, in the second case we rely on the method `std::list::insert` in order to insert in the list which corresponds to the given row a new pair with the index of the required column and the value of the element. Notice that `std::list::insert` receives as parameter not only the new pair, but also an iterator to the position in which the element must be inserted (the iterator returned by `search` in line 27).

The worst case complexity of the method `set` is $O(n)$, where n is the number of columns, which is the cost of the `search` method. Indeed, vectors allow random access, which means that, in `search`, we reach in $O(1)$ the required row and then we have to walk over the corresponding list until we find the element. On the other hand, both the assignment we perform in line 30 and the method `std::list::insert` have complexity $O(1)$ (in general, we have that the cost of `std::list::insert` is linear in the number of inserted elements).

Having as external container a `std::list` instead of a `std::vector` increases the overall complexity of the method. Indeed, lists do not support random access and, therefore, the complexity of `search` becomes $O(m + n)$, where m is the number of rows in the matrix.

In order to improve the performances of the program, we can consider to replace the given data structure by a `std::vector<std::map<std::size_t, double>>`, where we have as keys the indices of the rows and as values the same lists we were storing before in the vector. A similar data structure produces an overall complexity of the method `set` which is $O(\log n)$. Indeed, we can modify the method `search` to rely on `std::map::find`, which has complexity $O(\log n)$; moreover, the method `std::map::insert` we would use in line 34 has again complexity $O(\log n)$.

Exercise 10 **HashMap*** (*exam 03/02/2017*)

`HashMap` is a structure that can be implemented as a vector of lists. Each list is a bucket to keep elements with the same hash value. Elements are pairs of `<key, value>` (see Figure 6.6). When an element is inserted in `HashMap`, a hash function computes the hash value from the key and inserts the element in the relevant bucket.

Given the following definition:

```

1 #ifndef HASHMAP_HH
2 #define HASHMAP_HH

```

```

3
4 #include <list>
5 #include <iostream>
6 #include <vector>
7 #include <utility>
8
9 class HashMap {
10
11     typedef int key_type;
12     typedef int value_type;
13
14     typedef std::vector<std::list<std::pair<key_type, value_type>>> table;
15
16     double max_load_factor; //max_load_factor
17     size_t bucket_count = 0; //vector length
18     size_t count = 0; //total inserted elements
19
20     std::hash<key_type> hash_code;
21
22     void
23     resize (void);
24
25     table hashtable;
26
27 public:
28
29     HashMap():
30         max_load_factor(0.75), bucket_count(8),
31         count(0), hashtable(bucket_count) {}
32
33     void
34     add (const key_type & key, const value_type & value);
35
36     void
37     remove (const key_type & key);
38
39     void
40     dump_internal_structure() const;
41
42 }; /* end of class declaration */
43
44 #endif /* HASHMAP_HH */

```

Implement the two methods below and discuss their complexity:

1. `add (int key, int value)` computes the hash value of `key` and places `value` in the resulting bucket.
2. Hash values depend on the table size, thus they change for each entry when resizing the table. `resize()` should create a double sized vector when `max_load_factor` is exceeded, recompute the hash value of the old elements and reallocate them in the new container.

Exercise 10 - Solution

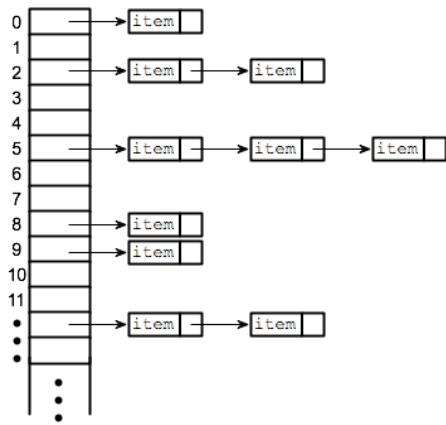


Figure 6.6: HashMap

The `HashMap` is represented by a `std::vector<std::list<std::pair<int,int>>`, hidden in the user defined name `table` (see line 14 of the header file).

The class stores a `double`, called `max_load_factor`. The load factor represents the ratio between the number of elements in the container and the number of buckets in which these elements are stored; in particular, it gives information about the probability that two elements are stored in the same bucket. The `max_load_factor`, as the initial number of buckets, is fixed when the container is initialized; when it is exceeded, the number of buckets must be increased so that the probability of collisions is kept reasonably small.

`bucket_count` represents the number of buckets that are stored in the `HashMap`; it is fixed during the initialization and then it is increased any time the `max_load_factor` is exceeded, so that the ratio between the number of elements and `bucket_count` itself remains small.

Finally, `count` is the number of elements actually stored in the container, which is of type `table` (see line 25).

The class stores also an object of type `std::hash<int>`, named `hash_code`. This object takes an element of type `int` (a key) and it computes its hash value, i.e. a value of type `size_t` such that, if `k1` and `k2` are equal, `hash_code(k1)` and `hash_code(k2)` are equal, while the probability of the hash values to be equal is very small if `k1` and `k2` are different keys. Thanks to this property, hash values are used to compute the position that two elements should have in a container: if the elements are characterized by the same key, their hash values are equal and this prevents from creating multiple elements with the same key and allows to use this key to find the element in the container. On the other hand, if the elements have different keys, their hash values are different and therefore they will be stored in different buckets, which prevents from having one single bucket filled with a lot of elements.

The implementation of the required method, written in file "HashMap.cpp", is reported below:

- Resize:

```

3 void
4 HashMap::resize (void)
5 {
6     bucket_count *= 2;
7
8     table temp (bucket_count);
9     hashtable.swap (temp);
10
11    count = 0;
12
13    // iterate over old hash table and add each entry to a new table.

```

```

14  for (const auto & bucket: temp)
15  {
16      for (const auto & couple: bucket)
17      {
18          add (couple.first, couple.second);
19      }
20  }
21 }
```

The method `resize`, reported above, doubles the value of `bucket_count` (i.e. the number of elements that can be stored in the vector), initializes a new `table` with this size and it uses the method `std::vector::swap` in order to insert in `temp` all the elements previously stored in `hashtable`. Then it sets to 0 the number of elements in `hashtable` (which, after the `swap`, is an empty container), loops over all the buckets in `temp` and over all the elements in each bucket and it calls the method `add` in order to insert the element in `hashtable`.

- Add:

```

23 void
24 HashMap::add (const key_type & key, const value_type & value)
25 {
26     const std::size_t idx = hash_code (key) % bucket_count;
27
28     // look if the key is already inserted
29     auto it = hashtable[idx].cbegin ();
30     while (it != hashtable[idx].cend () and it->first != key)
31         ++it;
32
33     // if you reach the end, you could not find the pair
34     if (it == hashtable[idx].cend ())
35     {
36         hashtable[idx].push_back ({key, value});
37         ++count;
38     }
39
40     if (static_cast<double>(count) / bucket_count > max_load_factor)
41         resize();
42 }
```

The method `add` receives as parameters the key and the value of the element that must be added to the hash table. In line 26, it computes the index of the bucket in which the element should be added. In order to do this, it computes the modulo between the value given by `hash_code(key)` and the number of buckets in the container (remember that `hash_code(key)` returns the hash value of the key, i.e. a `size_t` which can be any integer number, and we have to relate it with the number of buckets in the container in order to get an admissible index).

Given the index, the method loops over the corresponding bucket in order to check whether the key passed as parameter is already stored in the hash table (lines 29 to 31). If this is not the case, a new pair `{key,value}` is added at the end of the list which represents the bucket and the number of inserted elements is incremented (lines 33 to 38).

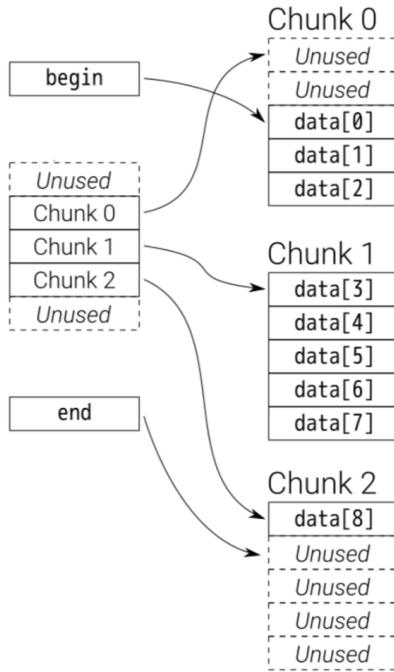


Figure 6.7: Structure of a deque.

Finally, the ratio between `count` and `bucket_count` is computed in order to check whether it is necessary to resize the hash table, incrementing the number of buckets.

If N is the number of elements stored in the hash table, the `resize` method is definitely $O(N)$, since it loops over all the stored items to copy them in the freshly created double-size data structure. On the other hand, the time complexity of `add` requires some more thought. Obviously, the worst case is $O(N)$, when it calls `resize` because of a high load factor. In every other case the time complexity is dominated by the search in the bucket: calling b the (maximum) list size, we have $O(b)$. The performance of the hash table depends on the quality of the hash function: if collisions remain rare, `add` is amortized constant in complexity, but in the worst case b can still become comparable to N .

Exercise 11 Deque*

`Deque` is a sequence container with dynamic size that can be expanded or contracted on both ends (either its front or its back). In fact, each `Deque` keeps a double ended queue of fixed-size chunks. As it is shown in Figure 6.7, each `Deque` can be implemented as a vector of arrays with fixed size; the number of arrays, i.e. the size of the vector, can grow dynamically as we push or pop elements from the `Deque`.

Write `Deque` as a class that implements `push_back` (a new entry is added inside the last chunk; if the last chunk is full, a new chunk is allocated) and `push_front` (the entry is added inside the first chunk; if the first chunk is full, a new chunk is allocated) methods and overloads `operator[]`. Instead of using a vector of arrays, implement your data structure as a `std::vector<std::vector<int>>`, where the internal vectors have fixed size.

Discuss the complexity of the methods above and explain when you would choose to use a `Deque` rather than a `vector`.

Exercise 11 - Solution

***** file deque.hpp *****

```
1 #ifndef __DEQUE__
2 #define __DEQUE__
3
4 #include <vector>
5
6 namespace ds
7 {
8     class Deque
9     {
10         typedef std::vector<std::vector<int> > container_type;
11
12     public:
13         typedef int value_type;
14         typedef typename container_type::size_type size_type;
15
16     private:
17         container_type chunks;
18
19         constexpr static size_type chunk_length = 8;
20         constexpr static size_type initial_chunk_number = 3;
21
22         size_type first_offset;
23         size_type off_end_offset;
24
25         inline bool
26         on_boundary (size_type offset) const
27         {
28             return offset % chunk_length == 0;
29         }
30
31         void
32         make_room_for (size_type);
33
34     public:
35         Deque (void)
36             : first_offset (0), off_end_offset (0) {}
37
38         inline size_type
39         size (void) const
40         {
41             return off_end_offset - first_offset;
42         }
43
44         inline bool
45         empty (void) const
46         {
47             return size () == 0;
48         }
49
50         value_type &
51         operator [] (size_type);
```

```

52 const value_type &
53 operator [] (size_type) const;
54
55 void
56 push_front (const value_type & z);
57 void
58 push_back (const value_type & z);
59 };
60
61 }
62
63 #endif

```

The class `Deque` stores a vector of chunks (see line 17) and two indices, `first_offset` and `off_end_offset`. The first one represents the global index of the first inserted element, while the second one is the global index of the first empty element, i.e. the global index of the first free position in the container. Both are not fixed because a `Deque` can dynamically grow both in the front and in the back.

Notice that both `first_offset` and `off_end_offset` are global indices, meaning that, considering, for example, the situation in Figure 6.7, `first_offset` is equal to 2 and `off_end_offset` is equal to 11.

The number of elements that can be stored in a chunk and the initial number of chunks in the container are set in lines 19 and 20, respectively.

The constructor of the class `Deque` (see lines 35 and 36) initializes to 0 both `first_offset` and `off_end_offset`. Notice that, for what concerns the initialization of the vector `chunks`, we rely on the default constructor of `std::vectors`, meaning that `chunks` is empty. We will use `initial_chunk_number` only to resize `chunks` when we call `push_back` or `push_front` for the first time.

The class implements two `private` methods: `on_boundary` returns `true` if the index passed as parameter is on the boundary of a chunk. In particular, this is needed in the method `push_front` when the index is equal to 0 and in the method `push_back` and the index is a multiple of `chunk_length`. Indeed, in the first case we need to add the new element in the last position of the previous chunk, while in the second case we have to add the new element at the first position of the following one.

The method `make_room_for`, in turn, receives as parameter the number of elements we are adding to the `Deque` and it ensures that the container has enough space to store them.

The class implements also six `public` methods. Other than the three required by the text (notice that the `operator[]` is overloaded twice, because it has both a constant and a non constant version), we have `size` (lines 38 to 42), which returns the number of elements stored in the `Deque`, and `empty` (lines 44 to 48), which returns `true` if no elements are stored in the container.

The implementation of all these methods is described below.

***** file deque.cpp *****

```

1 #include "deque.hpp"
2
3 namespace ds
4 {
5     void
6     Deque::make_room_for (size_type additions)
7     {
8         const size_type next_size = size () + additions;

```

```

9   const size_type needed_chunks = next_size % chunk_length > 0
10  ? next_size / chunk_length + 1
11  : next_size / chunk_length;
12  size_type next_chunks = chunks.size();
13
14 if (next_chunks == 0 and needed_chunks > 0)
15 {
16     next_chunks = initial_chunk_number;
17 }
18
19 if (first_offset == 0 or off_end_offset == chunks.size() * chunk_length)
20 {
21     next_chunks *= 2;
22 }
23
24 while (needed_chunks > next_chunks)
25 {
26     next_chunks *= 2;
27 }
28
29 if (next_chunks > chunks.size())
30 {
31     const size_type center_before = chunks.size() / 2;
32     chunks.resize(next_chunks);
33     const size_type center_after = chunks.size() / 2;
34     const size_type shift = center_after - center_before;
35
36     for (size_type i = off_end_offset / chunk_length;
37          i < next_chunks and i >= first_offset / chunk_length;
38          --i)
39     {
40         using std::swap;
41         swap(chunks[i], chunks[i + shift]);
42     }
43
44     if (empty())
45     {
46         first_offset = off_end_offset =
47             center_after * chunk_length + chunk_length / 2;
48         chunks[center_after].resize(chunk_length);
49     }
50     else
51     {
52         const size_type offset_shift = shift * chunk_length;
53         first_offset += offset_shift;
54         off_end_offset += offset_shift;
55     }
56 }
57 }
58
59 Deque::value_type &
60 Deque::operator [] (size_type i)

```

```

61 {
62     const size_type real_idx = i + first_offset;
63     const size_type chunk_idx (real_idx / chunk_length),
64         position (real_idx % chunk_length);
65     return chunks[chunk_idx][position];
66 }
67
68 const Deque::value_type &
69 Deque::operator [] (size_type i) const
70 {
71     const size_type real_idx = i + first_offset;
72     const size_type chunk_idx (real_idx / chunk_length),
73         position (real_idx % chunk_length);
74     return chunks[chunk_idx][position];
75 }
76
77 void
78 Deque::push_front (const value_type & element)
79 {
80     make_room_for (1);
81
82     if (on_boundary (first_offset --))
83     {
84         const size_type new_chunk = first_offset / chunk_length;
85         chunks[new_chunk].resize (chunk_length);
86     }
87
88     operator [] (0) = element;
89 }
90
91 void
92 Deque::push_back (const value_type & element)
93 {
94     make_room_for (1);
95
96     if (on_boundary (off_end_offset ++))
97     {
98         const size_type new_chunk = off_end_offset / chunk_length;
99         chunks[new_chunk].resize (chunk_length);
100    }
101
102    operator [] (size () - 1) = element;
103 }
104 }
```

The most important method is `make_room_for` (see the implementation in lines 5 to 57). It is called both in `push_front` and in `push_back` (see lines 80 and 94, respectively), passing as parameter 1, which is the number of elements we want to add in both cases.

The new size of the container, namely `next_size`, is computed as the current size plus the parameter `additions` (line 8). The index `needed_chunks` (line 9 to 11) represents the number of chunks we need to store all the elements (both the ones that are already in the container and the ones we want to add). The index `next_chunks` is initialized in line 12 with the number of chunks currently stored in the `Deque` and it is updated within the method to

represent the new number of chunks we want to have in the container.

Since any object of type `Deque` is initialized as an empty container, at the first call of `push_front` or `push_back` the value of `chunks.size()` is equal to zero, while `needed_chunks` is greater than zero because we need space to store the new elements. Therefore, in lines 14 to 17, `next_chunks` becomes equal to the initial number of chunks we set for the container. On the other hand, if `first_offset` is equal to zero or `off_end_offset` is equal to the maximum number of elements we can store in the existing chunks (this is true, for example, at the beginning, when both `first_offset` and `off_end_offset` are zero), the value of `next_chunks` is doubled. The same happens as long as the number of needed chunks is greater than `next_chunks`.

In lines 29 to 56, we perform any modification of the container we possibly need to store the new elements. Notice that the main goal of the `Deque` structure is to avoid reallocations of the internal chunks, so that any pointer to an element in the `Deque` is never invalidated when we add new elements. The same does not happen with vectors, since when we call `push_back` we may incur in a reallocation of the entire data structure.

The structure of a `Deque` is designed so that the container is filled starting from the element in the middle of the central chunk. When, for example, the method `push_back` is called for the first time, `make_room_for(1)` is called. When the method arrives to line 29, `next_chunks` is equal to 6 (indeed, `next_size` is 1, `needed_chunks` is 1 and `next_chunks` becomes equal to 3 in line 16 and it is doubled in line 21 since both `first_offset` and `off_end_offset` are zero). In line 31, `center_before` is set to zero, while, in line 33, `center_after` is set to 3 after having resized the vector of chunks, so that, when we have 6 chunks, the fourth one is considered as central chunk. The first element to be filled, therefore, will be the central element of the fourth chunk.

Suppose now that the container is not empty, but that, for example, we are in the following situation:

```
chunk 0:  
  
chunk 1:  
  
chunk 2:  
  
chunk 3:  
    1  2  3  4  5  6  
  
chunk 4:  
    7  8  9  10 11 12 13 14  
  
chunk 5:  
    15 16 17 18 19 20 21 22
```

where the first three chunks are empty, the fourth has been partially filled and the last two are completely full (therefore `first_offset` is equal to 26, which is the global index of the first existing element, and `off_end_offset` is equal to 48). If we run `push_back(25)`, a new element must be added at the end of the `Deque` and, in order to do this, we need to allocate new chunks. The method `make_room_for(1)` is called. When we reach line 29, `next_size` is equal to 23, `needed_chunks` is 3 and `next_chunks` is 12 (it is initialized to 6 in line 12 and it is doubled in line 21). This means that we want to double the number of chunks in the vector. Any time we resize the vector `chunks`, we do it in such a way that an equal number of chunks is inserted at the beginning and at the end of the vector `chunks`. This is achieved by the operations performed in lines 31 to 42. Indeed, `center_before` is set to 3 (the index of the current central chunk), while `center_after` is set to 6 (the index of the

central chunk after the resize). The variable `shift`, in line 34, is set to 3. In lines 36 to 42, we loop over all the indices between `off_end_offset / chunk_length` (which in this case is 6) and `first_offset / chunk_length` (which in this case is 3). At every iteration, we swap the chunk stored in position `i` with the one stored in position `i + shift`, in such a way that, in the example we are considering, when we reach line 42 we have:

```
chunk 0:
chunk 1:
chunk 2:
chunk 3:
chunk 4:
chunk 5:
chunk 6:
    1   2   3   4   5   6
chunk 7:
    7   8   9   10  11  12  13  14
chunk 8:
    15  16  17  18  19  20  21  22
chunk 9:
chunk 10:
chunk 11:
```

i.e. three chunks are added at the beginning and three are added at the end of the vector `chunks`.

In lines 44 to 49, if the `Deque` is still empty, both `first_offset` and `off_end_offset` are set to the index of the element in the middle of the central chunk, which is resized by `chunk_length`. If the `Deque` is not empty, in turn, both `first_offset` and `off_end_offset` are shifted, so that, considering the example above, `first_offset` becomes equal to 50 and `off_end_offset` becomes equal to 72.

The `operator[]` (both in the non constant version, in lines 59 to 66, and in the constant version in lines 68 to 75) needs to compute, starting from the index `i` passed as parameter, the number of chunk and the position in which the required element is stored. In particular, in order to compute those values, we need to know the position of the required element starting from `first_offset` (see for example line 71). Given this information, both `chunk_idx` and `position` are computed dividing `real_idx` by the length of a chunk and by considering once the quotient and once the remainder.

Both the methods `push_front` (lines 77 to 89) and `push_back` (lines 91 to 103) are implemented on top of `make_room_for` and on top of the `operator[]`. Notice that, in both the cases, we have to check whether either `first_offset` or `off_end_offset` are on the boundary of the current chunk and, if this is the case, we have to resize either the previous or the following chunk in order to be able to add the new element.

Complexity: The complexity of the `operator[]` is $O(1)$ because we rely on the corresponding

`operator[]` of `std::vectors`, which supports random access.

On the other hand, both the methods `push_front` and `push_back` have a worst-case complexity of $O(N)$, where N is the number of elements already stored in the `Deque`. Indeed, since the internal size of the chunks is fixed, the computational cost of the resize we possibly perform in lines 85 and 99 is not considered when computing the overall complexity, which therefore coincides with the one of the method `make_room_for`. Thanks to the fact that, in `make_room_for`, we use `std::swap` instead of copying the elements from one chunk to the other, the computational cost of the method is given by the `chunks.resize` we perform in line 32, which is $O(N)$.

Deques are useful when adding elements to both sides of the container is frequently required (`std::vectors` are already efficient, but only when we need to increase the size from the end, since `push_back` has complexity $O(N)$ in the worst case, but $O(1)$ in the average case, while `push_front` has always complexity $O(N)$).

Exercise 12 Heap*

A **Heap** is a data structure consisting of an almost complete binary tree satisfying the so-called *heap property*: for instance, in a *min-heap* every node hosts a value not less than the one stored in the parent node, meaning that the root of the tree holds the minimum. As you can see in figure Figure 6.8, where it is represented a *max-heap*, it is easy to store the implicit binary tree in a regular array, computing the parent, left and right child node positions algebraically.

Most operations on heaps boil down to moving elements up or down the tree in order to restore the heap property. Considering a *min-heap*, you can push a new element on the **Heap** appending it and then comparing to its parent: if the new element happens to be less than its parent, then they should be swapped and the procedure should be repeated, until either the new element is greater than its current parent or it reaches the root position. On the other hand, you can remove the root element by copying the last one in root position and moving it down the tree if one of its children is smaller.

Consider the class **Heap**, storing elements of type `int` and representing a *min-heap*, defined in file "minHeap.hpp".

***** file minHeap.hpp *****

```
1 #ifndef __MINHEAP__
2 #define __MINHEAP__
3
4 #include <initializer_list>
5 #include <vector>
6
7 namespace ds
8 {
9     class MinHeap
10    {
11        typedef std::vector<int> container_type;
12
13    public:
14        typedef int value_type;
15        typedef typename container_type::size_type size_type;
16
17    private:
```

```

18 container_type m_data;
19
20 void
21 sift_down (size_type i);
22 void
23 sift_up (size_type i);
24
25 void
26 build_Heap (void);
27
28 inline size_type
29 left (size_type i) const
30 {
31     return 2 * i + 1;
32 }
33
34 inline size_type
35 right (size_type i) const
36 {
37     return 2 * i + 2;
38 }
39
40 inline size_type
41 parent (size_type i) const
42 {
43     return (i + 1) / 2 - 1;
44 }
45
46 public:
47     MinHeap (void) = default;
48
49     MinHeap (container_type::const_iterator first,
50             container_type::const_iterator last);
51
52     MinHeap (std::initializer_list<value_type> il);
53
54     inline size_type
55     size (void) const
56     {
57         return m_data.size ();
58     }
59
60     inline bool
61     empty (void) const
62     {
63         return m_data.empty ();
64     }
65
66     inline const value_type &
67     peek (void) const
68     {
69         return m_data.front ();

```

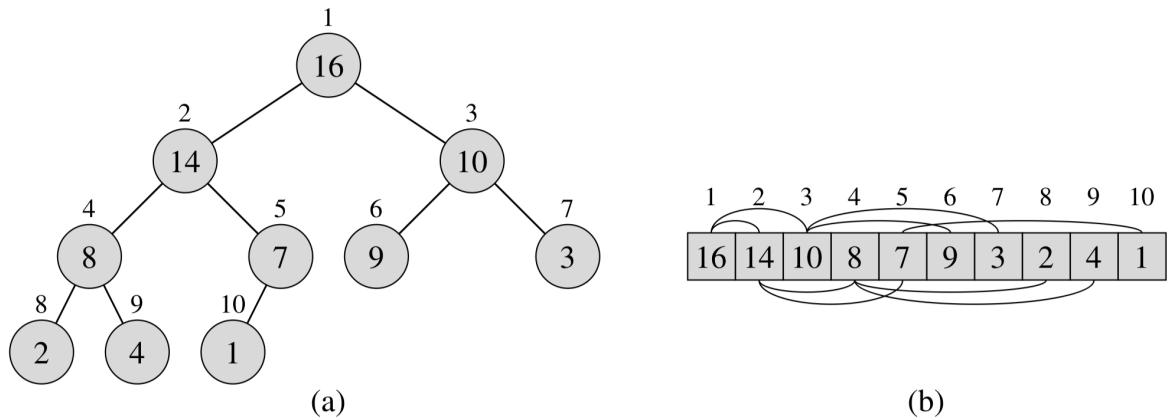


Figure 6.8: Max-heap.

```

70 }
71
72     void
73     pop (void);
74
75     void
76     push (const value_type &);
77
78     void
79     replace (const value_type &);
80
81 };
82
83 }
84
85 #endif

```

Such class declares the following members:

```
- std::vector<int> m_data;
```

Implement two procedures

- **void sift_down (size_type i);** that, given an index in the array, moves the corresponding element down in the tree.
- **void sift_up (size_type i);** that, given an index in the array, moves the corresponding element up in the tree.

You can assume that the class already provides an implementation of the **parent()**, **left()** and **right()** methods, which return respectively the parent, left or right child index of the one passed as argument.

State the asymptotic complexity of the implemented methods, providing an informal motivation for your answer.

Exercise 12 - Solution

The **Heap** class implements three constructors: the first one (see line 47 of file "minHeap.hpp") is the default constructor with no parameters. The second one (see lines 49 and 50) receives

two iterators, while the third one (see line 52) receives an `initializer_list` with the elements that must be stored in the tree.

As it is shown in Figure 6.8, the indices of the elements in the tree (and therefore in the array) start from 1, while indices of C++ `std::vectors` start from zero. Suppose, for example, we are in node 3, which stores the value 10. The index, in the array, of its parent is 1 and it is computed as $\lfloor \frac{idx}{2} \rfloor$ where idx is equal to the current index, i.e. 3. The indices of the left and the right children, in turn, are computed as $2 * idx$ and $2 * idx + 1$, respectively.

Once we have the indices in the array, we must compute the indices of the elements in the `std::vector<int>` which represents the data structure in the `Heap` class. Since the indices of C++ vectors start from 0, we get the parent as $\frac{idx+1}{2} - 1$, the left child as $2 * idx + 1$ and the right child as $2 * idx + 2$.

The methods `left`, `right` and `parent` (see lines 28 to 44 of file "minHeap.hpp") are implemented exactly in this way (notice that the keyword `inline` is not mandatory, since the methods defined in-class are automatically inlined).

***** file minHeap.cpp *****

```

1 #include "minHeap.hpp"
2
3 namespace ds
4 {
5     void
6     MinHeap::sift_down (size_type i)
7     {
8         bool keep_sifting = true;
9
10        while (keep_sifting and i < m_data.size ())
11        {
12            keep_sifting = false;
13
14            const size_type l = left (i), r = right (i);
15            size_type best = i;
16
17            if (l < m_data.size () and m_data[l] < m_data[best])
18            {
19                best = l;
20                keep_sifting = true;
21            }
22
23            if (r < m_data.size () and m_data[r] < m_data[best])
24            {
25                best = r;
26                keep_sifting = true;
27            }
28
29            if (keep_sifting)
30            {
31                using std::swap;
32                swap (m_data[best], m_data[i]);
33                i = best;
34            }
35        }
36    }
}
```

```

37
38     void
39     MinHeap::sift_up (size_type i)
40     {
41         while (i > 0 and m_data[i] < m_data[parent (i)])
42         {
43             using std::swap;
44             swap (m_data[i], m_data[parent (i)]);
45             i = parent (i);
46         }
47     }
48
49     void
50     MinHeap::build_Heap (void)
51     {
52         for (size_type i = parent (m_data.size ());
53             i >= 0 and i < m_data.size (); --i)
54         {
55             sift_down (i);
56         }
57     }
58
59     MinHeap::MinHeap (container_type::const_iterator first,
60                       container_type::const_iterator last)
61     : m_data (first, last)
62     {
63         build_Heap ();
64     }
65
66     MinHeap::MinHeap (std::initializer_list<value_type> il)
67     : m_data (il)
68     {
69         build_Heap ();
70     }
71
72     void
73     MinHeap::pop (void)
74     {
75         m_data.front () = m_data.back ();
76         m_data.pop_back ();
77         sift_down (0);
78     }
79
80     void
81     MinHeap::push (const value_type & element)
82     {
83         m_data.push_back (element);
84         sift_up (m_data.size () - 1);
85     }
86
87     void
88     MinHeap::replace (const value_type & element)

```

```

89  {
90      m_data.front () = element;
91      sift_down (0);
92  }
93 }
```

The methods required by the exercise are the **private** methods `sift_down` (see the declaration in lines 20 and 21 of file "minHeap.hpp") and `sift_up` (lines 22 and 23 of file "minHeap.hpp"). The first one is implemented in lines 5 to 36 of file "minHeap.cpp". It receives as parameter the index of the element we want to push down in the tree; it loops over all its children until it reaches a leaf and the *heap property* is not satisfied. At any iteration, the method computes the indices of the left and the right children of the given node (line 14) and it initializes the value of the index `best` with the current position. In lines 17 to 21, it uses the `operator<` to compare the values stored in the tree in position 1 and the one stored in position `best` (i.e. it compares the current value with the value of the left child). If the left child actually exists ($1 < m_data.size()$, i.e. we are not in a leaf) and the value stored in the left child is lower than the one in the current position, `best` takes the value of 1. The same procedure is performed with the right child. Finally, if `keep_sifting` is `true`, the element stored at position `best` and the current one are swapped.

A similar strategy is used in the method `sift_up` (see lines 38 to 47): the main difference is that here we check that the index `i` is not the root of the tree and we compare the value stored in the current node with the one stored in its parent.

The method `sift_down` is intended to be used, for example, during the construction of an Heap. Indeed, consider the two constructors, implemented in lines 59 to 64 and 66 to 70, respectively. In both cases, we use either the given iterators or the initializer list to initialize the data structure `m_data` (lines 61 and 67). This operation copies in `m_data` the values we pass as parameters, but the resulting data structure does not satisfy, in principle, the *heap property*. In order to recover this property, in lines 63 and 69 we call the **private** method `build_Heap`.

The method `build_Heap` (see lines 49 to 57) considers as first index `parent(m_data.size())` and, until the index is greater than or equal to zero, it calls `sift_down` in order to restore the *heap property* of the tree.

Suppose, for example, to define an `Heap` object as follows:

```
ds::minHeap min_Heap {17, 18, 5, 7};
```

The constructor of lines 66 to 70 runs: the container `m_data` is initialized to [17, 18, 5, 7] and the method `build_Heap` is called. Within `build_Heap`, `m_data.size()` is equal to 4, therefore the first index `i` is 1. The method `sift_down` compares the value stored at the index 1, which is 18, with the value stored in its left child, which is 7; since 7 is smaller than 18, the two elements are swapped. Notice that the element with index 1 does not have a right child. After the swap, the container `m_data` stores [17, 7, 5, 18]. The new value of `best`, in `sift_down`, is 3; the node with index 3 has neither left nor right child (it is a leaf) and therefore the method ends. The *heap property* is still not satisfied. In `build_Heap`, the index `i` becomes equal to 0 and `sift_down` is called again. The element in position 0, i.e. 17, is compared with its left child, i.e. 7, and its right child, i.e. 5. The best value according to `operator<` is 5, therefore `m_data` becomes [5, 7, 17, 18]. The new value of the index `best` is 2, which corresponds to a leaf. In `build_Heap`, `i` becomes equal to -1, which is not an admissible index; the method ends and the *heap property* of the tree is restored.

The complexity of the methods `sift_up` and `sift_down` is $O(\log N)$, where N is the number of elements stored in the heap.

6.2 Frequently Asked Questions

1. Is there any difference in using `std::size_t` or, for example, `std::vector::size_type`?

ANSWER: The type `std::size_t` is intended to store the maximum size of a theoretically possible object of any type, thus it is commonly used for array indexing and loop counting. From a theoretical point of view, if we want to define a type to index or count elements in a C++ container (for example a `std::vector` or a `std::string`), we should use the member `typedef` provided by the container itself, which means, in this case, `std::vector::size_type` or `std::string::size_type`. Typically, however, those types are defined as synonyms of `std::size_t`, then, for the standard cases, they are the same. On the other side, to use either `std::size_t` or `std::container::size_type` instead of, for example, `unsigned int`, to index, for instance, a `std::vector`, is more general and thus safer, because a program which uses `unsigned int` might fail on, e.g., 64-bits systems when the index exceeds the maximum value representable by `unsigned ints` or if it relies on 32-bit modular arithmetic.

Chapter 7

MPI

7.1 Exercises

Note: all exercises presented in the following are intended to be compiled from the commandline. This can be done by using the following command:

```
mpicxx -std=c++11 -Wall <list of the source files> -o <name of the executable>
```

In order to execute the program, the command to be used is:

```
mpiexec -np <number of required processors> ./<name of the executable>
```

Notice moreover that, if the number of required processors is greater than the number of processors actually present on the machine, it may be necessary to add in the execution command:

```
mpiexec --oversubscribe -np <required processors> ./<name of the executable>
```

More detailed examples are provided at the end of the exercises' solutions.

Exercise 1 Command line arguments

In order to practice with command line arguments, starting from the "Hello, world!" provided below, implement a modified version where it is possible to change the name for the greeting. The intended usage is:

```
$ mpiexec -np 2 ./hello  
Hello, world, from 0 of 2.  
Hello, world, from 1 of 2.
```

for the default version and

```
$ mpiexec -np 2 ./hello Eugenio  
Hello, Eugenio, from 0 of 2.  
Hello, Eugenio, from 1 of 2.
```

to change the name for the greeting.

The initial code is:

```
1 #include <iostream>  
2 #include <sstream>  
3 #include <string>  
4 #include <mpi.h>  
5  
6 int main (int argc, char *argv[]) {
```

```

7   MPI_Init(&argc,&argv);
8
9   int rank, size;
10
11  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
12  MPI_Comm_size (MPI_COMM_WORLD, &size);
13
14  std::ostringstream builder;
15  builder << "Hello, world, from " << rank << " of " << size;
16
17  std::string message(builder.str());
18  unsigned length = message.size();
19
20
21 if (rank > 0)
22 {
23     MPI_Send (&length,1,MPI_UNSIGNED,0,0,MPI_COMM_WORLD);
24     MPI_Send (&message[0],length,MPI_CHAR,0,1,MPI_COMM_WORLD);
25 }
26
27 else
28 {
29     std::cout << message << std::endl;
30     for (int r=1; r<size; r++)
31     {
32         MPI_Recv (&length,1,MPI_UNSIGNED,r,0,MPI_COMM_WORLD,
33                   MPI_STATUS_IGNORE);
34         message.resize(length);
35         MPI_Recv (&message[0],length,MPI_CHAR,r,1,MPI_COMM_WORLD,
36                   MPI_STATUS_IGNORE);
37         std::cout << message << std::endl;
38     }
39 }
40
41 MPI_Finalize();
42
43 return 0;
44 }
```

Exercise 1 - Solution

In order to have the required behaviour, it is enough to add to the `main` function, between line 13 and line 15, the following lines:

```

std::string greeting = "world";
if (argc > 1)
    greeting = argv[1];
```

in order to check if an argument has been provided during the execution and, in that case, to read it. Then, we have to modify the builder writing in line 15 and 16

```

std::ostringstream builder;
builder << "Hello, " << greeting << ", from " << rank << " of " << size;
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall hello.cc -o hello
```

Exercise 2 Discrete Gradient

Provide a parallel implementation with MPI of a function that computes the discrete gradient of a real function on \mathbb{R}^n . Your function should work irrespective of n. The prototype is:

```
nd_vector  
compute_gradient (std::function<double (const nd_vector &),  
                  const nd_vector &, double h = 0.01);
```

The class `nd_vector` 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. Consider the centered finite differences to approximate the partial derivatives and make the gradient available on all the processes.

Exercise 2 - Solution

The aim of this exercise was only to implement the function `compute_gradient` (see files "gradient.hh" and "gradient.cc"). However, the solution shows also the implementation of the class `nd_vector`.

```
***** file nd_vector.hh *****
```

```
1 #ifndef __ND_VECTOR__  
2 #define __ND_VECTOR__  
3  
4 #include <initializer_list>  
5 #include <vector>  
6  
7 namespace numeric  
8 {  
9     class nd_vector  
10    {  
11        typedef std::vector<double> container_type;  
12  
13        container_type x;  
14  
15    public:  
16        typedef container_type::value_type value_type;  
17        typedef container_type::size_type size_type;  
18        typedef container_type::pointer pointer;  
19        typedef container_type::const_pointer const_pointer;  
20        typedef container_type::reference reference;  
21        typedef container_type::const_reference const_reference;  
22  
23        explicit nd_vector (size_type n = 0);  
24        nd_vector (std::initializer_list<double>);  
25  
26        size_type  
27        size (void) const;
```

```

29     reference
30     operator [] (size_type);
31     value_type
32     operator [] (size_type) const;
33
34     pointer
35     data (void);
36     const_pointer
37     data (void) const;
38   };
39 }
40
41 #endif

```

***** file nd_vector.cc *****

```

1 #include "nd_vector.hh"
2
3 namespace numeric
4 {
5   nd_vector::nd_vector (size_type n)
6     : x (n, 0.) {}
7
8   nd_vector::nd_vector (std::initializer_list<double> il)
9     : x (il) {}
10
11  nd_vector::size_type
12  nd_vector::size (void) const
13  {
14    return x.size ();
15  }
16
17  nd_vector::reference
18  nd_vector::operator [] (size_type idx)
19  {
20    return x[idx];
21  }
22
23  nd_vector::value_type
24  nd_vector::operator [] (size_type idx) const
25  {
26    return x[idx];
27  }
28
29  nd_vector::pointer
30  nd_vector::data (void)
31  {
32    return x.data ();
33  }
34
35  nd_vector::const_pointer
36  nd_vector::data (void) const
37  {

```

```

38     return x.data ();
39 }
40 }
```

In the file "mpi_helpers.hh", we declare (for the definition, see "mpi_helpers.cc"), in namespace `mpi`, two helper functions, `rank()` and `size()`, which return the number of available cores and the rank, respectively.

***** file `mpi_helpers.hh` *****

```

1 #ifndef __MPI_HELPERS__
2 #define __MPI_HELPERS__
3
4 namespace mpi
5 {
6     int
7     rank (void);
8
9     int
10    size (void);
11 }
12
13#endif
```

***** file `mpi_helpers.cc` *****

```

1 #include <mpi.h>
2
3 #include "mpi_helpers.hh"
4
5 namespace mpi
6 {
7     int
8     rank (void)
9     {
10         int rk;
11         MPI_Comm_rank (MPI_COMM_WORLD, &rk);
12         return rk;
13     }
14
15     int
16     size (void)
17     {
18         int sz;
19         MPI_Comm_size (MPI_COMM_WORLD, &sz);
20         return sz;
21     }
22 }
```

REMEMBER: the default values of the parameters (namely, in the file below, `h = 0.01`) must be written only in the declaration of the function (or only in the definition, but the declaration is usually a better choice).

***** file `gradient.hh` *****

```

1 #ifndef __GRADIENT__
2 #define __GRADIENT__
3
4 #include <functional>
5
6 #include "nd_vector.hh"
7
8 namespace numeric
9 {
10     nd_vector
11     compute_gradient (std::function<double (const nd_vector &) >,
12                       const nd_vector & z, double h = 0.01);
13 }
14
15 #endif

```

In order to solve this exercise, we start from the assumption that the values stored in the vector `x` are known in all the processes (otherwise, we would have to broadcast the size of `x` and then, for example, to scatter the vector in a suitable way in order to split the computation between the processes). Thanks to this assumption, we can choose to use a cyclic partition instead of a block partition (see lines 16 to 22 of the file "gradient.cc"): this allows to avoid useless replications of the data in the different processes (which is what we would get if we used in this context an MPI_Scatter). In particular, the cyclic partition works properly both when the number of available cores evenly divides the size of `x` and when it does not.

At the end of the computation, we need to collect in `gradient` the proper values computed by every process. In order to do this, in lines 24 to 27 we scan all the elements in `gradient` and, at every iteration, we perform a broadcast in which the process that have computed a certain element of `gradient` (namely the core with rank equal to `i%size`) communicates its value to all the others. This way, at the end of the for loop, all the processes store the correct value of `gradient` and they can return it to the function caller.

***** file gradient.cc *****

```

1 #include <mpi.h>
2
3 #include "gradient.hh"
4 #include "mpi_helpers.hh"
5
6 namespace numeric
7 {
8     nd_vector
9     compute_gradient (std::function<double (const nd_vector &) > f,
10                      const nd_vector & x, double h)
11     {
12         nd_vector gradient (x.size ());
13
14         const unsigned rank = mpi::rank (), size = mpi::size ();
15
16         for (nd_vector::size_type i = rank; i < x.size (); i += size)
17         {
18             nd_vector x_plus_h (x), x_minus_h (x);
19             x_plus_h[i] += h;
20             x_minus_h[i] -= h;
21             gradient[i] = (f (x_plus_h) - f (x_minus_h)) / 2 / h;

```

```

22     }
23
24     for (nd_vector::size_type i = 0; i < gradient.size (); ++i)
25     {
26         MPI_Bcast (&gradient[i], 1, MPI_DOUBLE, i % size, MPI_COMM_WORLD);
27     }
28
29     return gradient;
30 }
31 }
```

Note: here the computation and the communication are kept separate in two different loops, that work on different indices. You could not write, for example
`MPI_Bcast(&gradient[i],1,MPI_DOUBLE,rank,MPI_COMM_WORLD);`
within the first loop, otherwise you end up with every process trying to send its own result, while nobody is listening.

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall main.cc gradient.cc nd_vector.cc mpi_helpers.cc -o main
```

Exercise 3 Discrete Laplacian (*exam 03/02/2017*)

Provide a parallel implementation with MPI of a function to compute the discrete Laplace operator of a real function on \mathbb{R}^n . It should work irrespective of n . The prototype is:

```

double
compute_laplacian (std::function<double (const nd_vector &)>,
                   const nd_vector & z, double h = 0.01);
```

The class `nd_vector` stores a vector in \mathbb{R}^n and can be initialized both by copy and providing the size n . Furthermore, it provides a `size()` method to read n and an unchecked indexing operator to access the values. Consider the second order centered finite difference to approximate the partial derivatives, then make the Laplacian available on all the processes.

Exercise 3 - Solution

Note: for a possible implementation of the class `nd_vector` (not required by the exercise), see Exercise 2.

The solution of this exercise is written under the hypothesis that the function f , the vector v and the increment h are available on all the processes. The communication of h and v could also be performed within the function itself, but we must be aware of the parallelization scheme that we choose: in order to evaluate $f : \mathbb{R}^n \rightarrow \mathbb{R}$, we need a vector of n dimensions, therefore we cannot split v across the processes. In this exercise, actually, the only possibility is to split the partial derivatives that any process has to compute (either with a cyclic partition, as it is done in the proposed solution, or with a block partition).

```

1   double
2   compute_laplacian (std::function<double (const nd_vector &)> f,
3                      const nd_vector & x, double h)
4  {
5      double laplacian (0.);
6      const double h2 = h * h;
```

```

7
8     int rank, size;
9     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
10    MPI_Comm_size (MPI_COMM_WORLD, &size);
11
12    for (nd_vector::size_type i = rank; i < x.size (); i += size)
13    {
14        nd_vector x_plus_h (x), x_minus_h (x);
15        x_plus_h[i] += h;
16        x_minus_h[i] -= h;
17        laplacian += (f (x_plus_h) - 2 * f (x) + f (x_minus_h)) / h2;
18    }
19
20    MPI_Allreduce (MPI_IN_PLACE, &laplacian, 1, MPI_DOUBLE,
21                   MPI_SUM, MPI_COMM_WORLD);
22
23    return laplacian;
24 }
```

Exercise 4 Ascending Sort (*exam 17/02/2017*)

Write with MPI a parallel program to sort integer numbers into ascending order. Assume that input is done and a `std::vector<long>` containing all the numbers to sort is available on process 0. In the end, the master should print the sorted values on standard output.

One possible strategy to parallelize such a task builds on the general *divide et impera* concept underlying mergesort. Similarly to this serial sorting algorithm, it is possible to *split* the overall dataset in smaller portions, *sort* them locally on the different processes, then *merge* them back to obtain a single sorted range on the master node.

Hint: to perform sort and merge operations, you can rely on the following methods, available in `<algorithm>`¹:

- a procedure called `sort`, that receives two iterators `first` and `last` and sorts the elements in the range `[first,last)`. In particular, given for instance
`std::vector<int> vec = {1,3,7,2,5};`
it is possible to sort the entire vector by calling:
`sort(vec.begin(), vec.end());`
- a function called `merge`, that receives five iterators `first1`, `last1`, `first2`, `last2` and `result` and combines the elements in the sorted ranges `[first1,last1)` and `[first2,last2)` into a new range beginning at `result` with all its elements sorted. Notice that the elements in the two ranges should already be sorted, while the container where we want to store the result must already have the correct dimension. Given, for instance, two sorted vectors `v1` and `v2` and a new vector `v` with size equal to `v1.size() + v2.size()`, it is possible to merge `v1` and `v2` in `v` by calling:
`std::merge(v1.cbegin(), v1.cend(), v2.cbegin(), v2.cend(), v.begin());`
- a procedure called `inplace_merge`, that receives three iterators `first`, `middle` and `last` and merges the two consecutive sorted ranges `[first,middle)` and `[middle,last)`, putting the result into the combined sorted range `[first,last)`. As before, notice that the two ranges should already be sorted. Given for instance

¹see www.cppreference.com or www.cplusplus.com for further details

```

std::vector<int> vec = {1,3,5,2,4,6,8,10};
it is possible to merge inplace vec, obtaining [1,2,3,4,5,6,8,10] by calling:
inplace_merge(vec.begin(), vec.begin() + 3, vec.end());

```

Exercise 4 - Solution

The solution reported below implements also the reading from standard input of the `std::vector<long>` that we want to sort (see lines 15 to 20; the text allowed to assume that the vector was already known by process 0). In particular, both the vector `numbers` and its size `numbers_count` are defined on all the processes. However, until line 23 only process 0 stores the proper value of those variables, while in all the other cores `numbers` is an empty vector and, therefore, `numbers_count` is equal to 0. In line 23, the proper value of `numbers_count` is broadcasted from process 0 in order to make every other core able to compute the length of the local portion it is going to sort. In the following lines, the vector `numbers` is resized on process 0 before the scattering, adding a tail of zeros in order to make the program work if the number of elements is not a multiple of the number of available cores.

This parallelization scheme leads to the fact that the last process will receive at the end of the vector `local` a number of zeros equal to `tail`. These zeros must not be considered by the sorting; therefore, in line 40, we call the function `std::sort` passing as last parameter `local.end()` if the process is not the last one and `(local.end() - tail)` otherwise. If we do not take care of this, the following can happen: if, for example, `numbers` stores `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]` and we work on 4 cores, the last process receives, in `local`, `[1, 0, 0]`. Therefore, after the `std::sort`, in the last process we have `[0, 0, 1]`, which of course makes the overall result wrong.

After the sorting, the local results are gathered on process 0 and saved in `numbers` (see line 45).

In lines 49 to 67 we work only on process 0; we resize again `numbers` in order to drop the tail we had added before (note that if we do not sort the local vectors properly, as it is shown in the example above, this operation deletes from `numbers` elements that we must not cancel) and then we merge the vector in order to obtain the final result (again, we have to take into account the fact that the portion of `numbers` given by the last process can be shorter than the ones given by all the other cores).

In order to merge the results, we use `std::inplace_merge` instead of `std::merge` in order to avoid to waste memory defining a new variable.

```

1 #include <algorithm>
2 #include <iostream>
3 #include <mpi.h>
4 #include <vector>
5
6 int
7 main (int argc, char *argv[])
8 {
9     MPI_Init (&argc, &argv);
10
11    int rank, size;
12    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
13    MPI_Comm_size (MPI_COMM_WORLD, &size);
14
15    std::vector<long> numbers;
16    if (rank == 0)
17    {
18        long value;

```

```

19     while (std::cin >> value) numbers.push_back (value);
20 }
21
22 unsigned numbers_count = numbers.size ();
23 MPI_Bcast (&numbers_count, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
24
25 unsigned local_share = numbers_count / size;
26 unsigned tail = 0;
27 if (numbers_count % size > 0)
28 {
29     ++local_share;
30     tail = size - numbers_count % size;
31 }
32 // Add elements to call the straightforward MPI_Scatter
33 if (rank == 0) numbers.resize (local_share * size);
34
35 std::vector<long> local (local_share);
36 MPI_Scatter (numbers.data (), local_share, MPI_LONG,
37               local.data (), local_share, MPI_LONG,
38               0, MPI_COMM_WORLD);
39
40 std::sort (local.begin (),
41             // If this process is not the last in the communicator
42             size - rank > 1
43             ? local.end () : local.end () - tail);
44
45 MPI_Gather (local.data (), local_share, MPI_LONG,
46             numbers.data (), local_share, MPI_LONG,
47             0, MPI_COMM_WORLD);
48
49 if (rank == 0)
50 {
51     // Drop the surplus elements added to call MPI_Scatter
52     numbers.resize (numbers_count);
53
54     std::vector<long>::iterator middle = numbers.begin () + local_share,
55     last = middle + local_share;
56
57     while (last < numbers.end ())
58     {
59         std::inplace_merge (numbers.begin (), middle, last);
60         middle = last;
61         last += local_share;
62     }
63
64     if (middle < numbers.end ())
65     {
66         std::inplace_merge (numbers.begin (), middle, numbers.end ());
67     }
68
69     for (long value: numbers) std::cout << value << std::endl;
70 }
```

```

71     MPI_Finalize ();
72     return 0;
73 }

```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall mpi-mergesort.cc -o mergesort
```

Exercise 5 Dot Product (*exam 21/07/2017*)

Implement with MPI a parallel program that reads from standard input two vectors of arbitrary dimension, computes the dot product, and prints the result on standard output.

The input consists of one natural number, n , representing the common dimension of both vectors, followed by all the n elements in the first and then in the second one. For example, the input content shown below corresponds to the output 8:

```

3
1 2 1
0 3 2

```

Hint: The scalar product is already implemented in the STL, in the header `<numeric>`². It receives three iterators, `first1`, `last1` and `first2`, and an initial element `init`, and it returns the result of accumulating `init` with the inner products of the pairs formed by the elements of two ranges starting at `first1` and `first2`. It can be used as follows: given for instance two vectors of `doubles` `vec1` and `vec2`, that are assumed to have the same dimension, the scalar product between `vec1` and `vec2` is given by:

```
double p = inner_product (vec1.cbegin(), vec1.cend(), vec2.cbegin(), 0.0);
```

Exercise 5 - Solution

The parallelization scheme followed by this exercise is similar to the one used in exercise 4. Indeed, after having read the input vectors from standard input, we resize them adding to both an useless tail of zeros (see lines 37 to 41). In this case, however, we compute the partial result of the dot product in all the cores in the same way, without discarding the zeros added to the vectors in the last process. This can be done without consequences because, of course, adding zeros to the partial sum does not affect the result (this would not be the case if, instead of a sum, we would have to perform a product or to sort elements as in the exercise 4. If we do not want to differentiate the computation done in the last core, we must be sure to resize the initial vectors with a value that does not bother the result).

In any case, to distinguish between the last core and the others allows to save some useless operations, which in general is a good practice, especially when we perform more complicated tasks.

```

1 #include <iostream>
2 #include <mpi.h>
3 #include <numeric>
4 #include <vector>
5
6 int
7 main (int argc, char *argv[])

```

²see www.cppreference.com or www.cplusplus.com for further details

```

8  {
9      MPI_Init (&argc, &argv);
10
11     int rank, size;
12     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
13     MPI_Comm_size (MPI_COMM_WORLD, &size);
14
15     std::vector<double> x, y;
16
17     if (rank == 0)
18     {
19         unsigned long dimensions = 0;
20         std::cin >> dimensions;
21
22         unsigned long counter = 0;
23         double value;
24         while (std::cin >> value)
25         {
26             if (counter++ < dimensions) x.push_back (value);
27             else y.push_back (value);
28         }
29     }
30
31     if (x.size () != y.size ())
32     {
33         std::cerr << "Wrong input vectors, dimension mismatch" << std::endl;
34     }
35     else
36     {
37         unsigned long aux_dimension = x.size ();
38         while (aux_dimension % size != 0) ++aux_dimension;
39         // The 0s added by resize do not affect the result
40         x.resize (aux_dimension);
41         y.resize (aux_dimension);
42
43         unsigned long portion = aux_dimension / size;
44         MPI_Bcast (&portion, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
45
46         std::vector<double> local_x (portion), local_y (portion);
47         MPI_Scatter (x.data (), portion, MPI_DOUBLE,
48                     local_x.data (), portion, MPI_DOUBLE,
49                     0, MPI_COMM_WORLD);
50         MPI_Scatter (y.data (), portion, MPI_DOUBLE,
51                     local_y.data (), portion, MPI_DOUBLE,
52                     0, MPI_COMM_WORLD);
53
54         const double partial = std::inner_product (local_x.cbegin (),
55                                         local_x.cend (),
56                                         local_y.cbegin (),
57                                         0.0);
58
59         double product;

```

```

60         MPI_SUM, 0, MPI_COMM_WORLD);
61
62     if (rank == 0)
63     {
64         std::cout << product << std::endl;
65     }
66 }
67
68 MPI_Finalize ();
69 return 0;
70 }
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall dot_product.cc -o dot_product
```

Exercise 6 Monte Carlo Method (exam 01/09/2017)

Implement with MPI a parallel *function* to approximate definite integrals on limited intervals $\Omega \subset \mathbb{R}$ with the Monte Carlo method. The required prototype is as follows:

```
std::pair<double, double>
montecarlo (const std::function<double (double)> & f, unsigned long N);
```

where f is the integrand and N is the sample size. Assume N is known only on the master node. For this exercise, consider $\Omega = [-1, +1]$. The pair returned by the function must contain the approximation of the integral as first element, while the second one is its estimated variance. This applies to all the MPI processes.

Monte Carlo method:

Recall that the Monte Carlo quadrature method draws N observations from a random variable $X \sim \mathcal{U}(\Omega)$ and estimates the integral with:

$$Q_N = |\Omega| \bar{Y}_N,$$

where $Y = f(X)$. Since $|\Omega| = \int_{\Omega} dx$, by the law of large numbers:

$$\lim_{N \rightarrow +\infty} Q_N = I = \int_{\Omega} f(x) dx.$$

Let $\sigma^2 = \text{Var}(Y)$, then the variance of the quadrature formula is:

$$\text{Var}(Q_N) = \frac{|\Omega|^2}{N} \sigma^2,$$

whence, exploiting the unbiased variance estimator S_N^2 for σ^2 , it is possible to derive an estimator of the quadrature variance:

$$S_{Q_N}^2 = \frac{|\Omega|^2}{N} S_N^2.$$

Random numbers:

The `<random>` header provides implementations of pseudo-random number generators and statistical distributions. See the following example and recall that every process should seed differently its random engine.

Initialization:

```

std::default_random_engine engine (289);
std::uniform_real_distribution<double> distro (-1., 1.);
Pseudo-random number generation:
const double x = distro (engine);

```

Exercise 6 - Solution

The parallelization scheme considered in this exercise is the following: after having made the sample size N available on all the cores (see line 15), we subdivide it into a certain number of subsamples (i.e. the number of random points that every core has to generate in order to compute its local result) in such a way that, if N is not a multiple of `size`, every core between 0 and $N\%size$ works on one subsample more than the others (lines 16 and 17).

In lines 19 and 20, every process initializes its own random engine. The argument `rank*rank *size*size` through which we initialize `engine` is needed in order to avoid that different cores generate the same random points.

From line 22 to 29, every core generates the random points, evaluates the function f in them and it computes the partial sum. The values of the function in the random points are stored in a vector `ys` because we will need them to compute the variance (alternatively, we could have saved in a vector the random point generated at every iteration, but in this way we would have to evaluate again the function, which in principle can be an expensive operation). In line 30, the global sum is computed and made available on all the processes through an `MPI_Allreduce`, so that every core can compute the proper result of the mean.

After that, every core computes the partial sum of the distances between the values in `ys` and the mean; the function `MPI_Allreduce` is used again to get the global sum in order to compute the variance on all the processes.

Note that, both in line 30 and in line 36, the function `MPI_Allreduce` is called passing as first argument `MPI_IN_PLACE`: this means that the result of the operation performed by the reduce (namely, the `MPI_SUM`) is stored in the same variable used as sendbuffer, avoiding the creation of another useless variable.

The values of the integral and of the estimated variance are then computed according to the formulas of the Monte Carlo method (see the text above and note that $|\Omega| = 2$) and they are returned in a `std::pair` (in line 43, we could have written

```
return {integral, integral_variance}
instead of using the function std::make_pair).
```

```

1 #include <mpi.h>
2 #include <random>
3
4 #include "montecarlo.hh"
5
6 namespace quadrature
7 {
8     std::pair<double, double>
9     montecarlo (const std::function<double (double)> & f, unsigned long N)
10    {
11        int rank, size;
12        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
13        MPI_Comm_size (MPI_COMM_WORLD, &size);
14
15        MPI_Bcast (&N, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
16        const unsigned long local = N / size;
17        const unsigned long subsample = (rank < N % size) ? local + 1 : local;

```

```

18
19     std::default_random_engine engine (rank * rank * size * size);
20     std::uniform_real_distribution<double> distro (-1., 1.);
21
22     double mean = 0.;
23     std::vector<double> ys (subsample);
24     for (double & y: ys)
25     {
26         const double x = distro (engine);
27         y = f (x);
28         mean += y;
29     }
30     MPI_Allreduce (MPI_IN_PLACE, &mean, 1, MPI_DOUBLE,
31                     MPI_SUM, MPI_COMM_WORLD);
32     mean /= N;
33
34     double variance = 0.;
35     for (double y: ys) variance += (y - mean) * (y - mean);
36     MPI_Allreduce (MPI_IN_PLACE, &variance, 1, MPI_DOUBLE,
37                     MPI_SUM, MPI_COMM_WORLD);
38     variance /= (N - 1);
39
40     const double integral = 2 * mean;
41     const double integral_variance = 2 * 2 * variance / N;
42
43     return std::make_pair (integral, integral_variance);
44 }
45 }
```

Exercise 7 Matrix Multiplication

The goal of this exercise is to write a parallel program to perform matrix multiplication. Recall that you can represent a dense matrix in memory as a vector, or array, where rows are stored one after another.

The initial code already implements a `DenseMatrix` class with an `operator *` performing serial matrix multiplication. The code of `DenseMatrix` class is the same that has been provided in Exercise 6 of Section 3.1, Chapter 3), with the only difference that the first constructor has default values 0 both for the number of rows and the number of columns, so that an empty matrix can be instantiated by writing:

```
DenseMatrix A;
```

In addition, you have a skeleton for the `main` function, where the input matrices are read from file and the final result is printed to screen.

Consider, as parallelization scheme, to subdivide the left hand operand in stripes by row and replicate the right hand matrix on all the processes. When you collect the partial slices, make the full result available on all processes.

Write the program assuming that all the matrix dimensions are multiples of the communicator size.

The initial code, and the text files are provided below. In particular, the program is intended to be compiled by the command

```
mpicxx -std=c++11 -Wall main.cpp matrix.cpp dense_matrix.cpp -o matrix_mult
```

and then to be run, for instance, by writing:

```
mpiexec -np 4 ./matrix_mult A.txt B.txt
```

(of course, you can choose any reasonable number of cores), i.e. passing the files A.txt and B.txt as parameters (the program will compute the product between the matrices stored in A.txt and in B.txt). The file "C_in.txt" stores the expected output of the product. Therefore, if the program is run by writing

```
mpiexec -np 4 ./matrix_mult A.txt B.txt > C_out.txt
```

it is possible to check if the computed solution is correct through the command `diff`:

```
diff C_in.txt C_out.txt
```

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

```
1 #include <fstream>
2 #include <iostream>
3
4 #include <mpi.h>
5
6 #include "dense_matrix.hpp"
7
8 int
9 main (int argc, char *argv[])
10 {
11     MPI_Init (&argc, &argv);
12
13     int rank (0), size (0);
14     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
15     MPI_Comm_size (MPI_COMM_WORLD, &size);
16
17     DenseMatrix full_A, full_B;
18
19     if (rank == 0)
20     {
21         std::ifstream first (argv[1]), second (argv[2]);
22         full_A.read (first);
23         full_B.read (second);
24     }
25
26 // Here goes your code
27
28     if (rank == 0)
29     {
30         std::cout << full_C.get_n_rows() << " " << full_C.get_n_cols() << "\n";
31
32         for (std::size_t i = 0; i < full_C.get_n_rows(); ++i)
33             for (std::size_t j = 0; j < full_C.get_n_cols(); ++j)
34             {
35                 std::cout << full_C(i, j);
36                 std::cout << (full_C.get_n_cols() - j == 1 ? "\n" : " ");
37             }
38     }
```

```

39     MPI_Finalize ();
40     return 0;
41 }
42
43 **** file dense_matrix.hpp ****
44
45 #ifndef DENSE_MATRIX_H_
46 #define DENSE_MATRIX_H_
47
48 #include <iostream>
49 #include <vector>
50 #include <cmath>
51
52 #include "matrix.hpp"
53
54 class DenseMatrix : public Matrix
55 {
56
57     public:
58         typedef std::vector<double> container_type;
59
60     private:
61         container_type m_data;
62
63         std::size_t sub2ind (std::size_t i, std::size_t j) const;
64
65     public:
66         DenseMatrix (std::size_t rows = 0, std::size_t columns = 0,
67                     double value = 0.0);
68         DenseMatrix (std::size_t rows, std::size_t columns,
69                     const container_type& values);
70
71         void read (std::istream &z);
72
73         void swap (DenseMatrix &z);
74
75         double & operator () (std::size_t i, std::size_t j) override;
76         double operator () (std::size_t i, std::size_t j) const override;
77
78         DenseMatrix transposed (void) const;
79
80         double * data (void);
81         const double * data (void) const;
82         container_type get_data (void) const;
83     };
84
85     DenseMatrix operator * (const DenseMatrix &, const DenseMatrix &);
86
87     void swap (DenseMatrix &, DenseMatrix &);
88
89     //double norm (const DenseMatrix&);
90
91 #endif // DENSE_MATRIX_HH

```

```
***** file A.txt *****
```

```
8 16
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
```

```
***** file B.txt *****
```

```
16 12
1 2 3 4 5 6 7 8 9 10 11 12
2 3 4 5 6 7 8 9 10 11 12 13
3 4 5 6 7 8 9 10 11 12 13 14
4 5 6 7 8 9 10 11 12 13 14 15
5 6 7 8 9 10 11 12 13 14 15 16
6 7 8 9 10 11 12 13 14 15 16 17
7 8 9 10 11 12 13 14 15 16 17 18
8 9 10 11 12 13 14 15 16 17 18 19
9 10 11 12 13 14 15 16 17 18 19 20
10 11 12 13 14 15 16 17 18 19 20 21
11 12 13 14 15 16 17 18 19 20 21 22
12 13 14 15 16 17 18 19 20 21 22 23
13 14 15 16 17 18 19 20 21 22 23 24
14 15 16 17 18 19 20 21 22 23 24 25
15 16 17 18 19 20 21 22 23 24 25 26
16 17 18 19 20 21 22 23 24 25 26 27
```

```
***** file C_in.txt *****
```

```
8 12
17 19 21 23 25 27 29 31 33 35 37 39
18 20 22 24 26 28 30 32 34 36 38 40
19 21 23 25 27 29 31 33 35 37 39 41
20 22 24 26 28 30 32 34 36 38 40 42
21 23 25 27 29 31 33 35 37 39 41 43
22 24 26 28 30 32 34 36 38 40 42 44
23 25 27 29 31 33 35 37 39 41 43 45
24 26 28 30 32 34 36 38 40 42 44 46
```

Exercise 7 - Solution

Since the `dense_matrix` class is already implemented, the only part of the program that has to be modified is the `main` function.

In particular, it can be modified adding between line 24 and line 28 the following lines, which perform the parallel multiplication:

```
1 unsigned m = full_A.get_n_rows(), n = full_A.get_n_cols(), p = full_B.
    get_n_cols();
```

```

2 MPI_Bcast (&m, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
3 MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
4 MPI_Bcast (&p, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
5
6 DenseMatrix local_B (n, p);
7 if (rank == 0)
8 {
9     MPI_Bcast (full_B.data (), n * p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
10    local_B = full_B;
11 }
12 else MPI_Bcast (local_B.data (), n * p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13
14 const unsigned stripe = m / size;
15 DenseMatrix local_A (stripe, n);
16 MPI_Scatter (full_A.data (), stripe * n, MPI_DOUBLE,
17               local_A.data (), stripe * n, MPI_DOUBLE,
18               0, MPI_COMM_WORLD);
19
20 DenseMatrix local_C = local_A * local_B;
21
22 DenseMatrix full_C (m, p);
23 MPI_Allgather (local_C.data (), stripe * p, MPI_DOUBLE,
24                 full_C.data (), stripe * p, MPI_DOUBLE,
25                 MPI_COMM_WORLD);

```

First of all, the number of rows of the matrix `full_A`, its number of columns and the number of columns of the matrix `full_B` must be broadcasted in order to make them available for all the processes. Note that it is not necessary to communicate the number of rows of `full_B` since the multiplication between two matrices can be done only if the number of rows of the second matrix is equal to the number of columns of the first one (for this reason, we can consider to add a check on this condition when we read the data from A.txt and B.txt, in lines 19 to 24 of "main.cpp", printing out an error message if the number of columns of `full_A` and the number of rows of `full_B` mismatch).

After these broadcasts, a local matrix `local_B` must be defined on all the processes. This passage is necessary even if the second matrix won't be splitted across the processes; indeed, the matrix `full_B`, read from B.txt, is available only on process 0 and then the other cores must receive a copy of those data. In particular, the definition of `local_B` is done in the following way: process 0 sends (through a broadcast) the values stored in `full_B` and it defines `local_B` so that it is equal to `full_B`; the other cores receive from process 0 the proper values and they store them in `local_B`.

The following step consists in splitting the first matrix across the different processes, so that each one of them will work only on some of its rows. In particular, since we are working using the assumption that the matrix dimensions are multiples of the communicator size, the number of rows that each process will receive is simply given by

`const unsigned stripe = m / size.`

Finally, each process computes its local product between `local_A` and `local_B`, storing the result in `local_C`.

All the local results are gathered and stored in `full_C`. The usage of `MPI_Allgather` instead of `MPI_Gather` assures that the final result will be available on all the processes.

Additional exercises:

1. Continue the exercise by relaxing the assumption on matrix dimensions. The provided test uses two processes and has matrices with even dimensions, so you can easily check

the general solution just by changing the number of processes for odd values when you run the program.

2. Assess the speedup you can obtain when the number of processes increases.

Take into account that, in order to observe a tangible difference in execution times, you might need to generate larger matrices and use a virtual machine with several processors on Azure, or at least configure the Vagrant box so that it uses as many CPUs as you have available on your laptop.

Exercise 8 Power Method (*exam 23/01/2018*)

Implement with MPI a parallel *function* to approximate the dominant eigenvector of a square matrix A via the power method.

Recall that the power method is an iterative algorithm. Let us assume that the matrix A has a dominant eigenvalue with corresponding dominant eigenvectors. If we choose an initial nonzero approximation \mathbf{x}_0 of one of the dominant eigenvectors of A and we form the sequence given by:

$$\begin{aligned}\mathbf{x}_1 &= A\mathbf{x}_0 \\ \mathbf{x}_2 &= A\mathbf{x}_1 = A^2\mathbf{x}_0 \\ \mathbf{x}_3 &= A\mathbf{x}_2 = A^3\mathbf{x}_0 \\ &\vdots \\ \mathbf{x}_k &= A\mathbf{x}_{k-1} = A^k\mathbf{x}_0\end{aligned}$$

for large powers k , we will obtain a good approximation of the dominant eigenvector of A . In particular provide:

- the implementation of the function `power_method`:

```
DenseMatrix power_method(const DenseMatrix &z, std::size_t iterations);
```

- the `main` function that calls `power_method`.

For matrix operations, you can rely on all the methods of the class `DenseMatrix`, whose header file is provided in Exercise 7, and you can neglect normalizing the vector used in the loop. Note that matrices are stored row major in contiguous memory.

The input matrix is available only on `rank 0` process and it is read from a file passed as first argument to `main`. For the sake of simplicity, assume that the matrix size evenly divides the number of processes. Describe at high level how your read and write functions work and in particular explain how you split the input matrix across processes. The read and write function implementation is not requested.

Exercise 8 - Solution

As in Exercise 4, the solution reported below implements also the function `read_matrix`, which is used to read the matrix from a file whose name is passed as parameter, and the function `print_eigen_vector`, even if their implementation was not required during the exam. In particular, in the function `read_matrix`, whose implementation can be seen in lines 36 to

66, process 0 reads from file the values of matrix A and then it scatters them across the different processes, so that the returned `DenseMatrix` is already the portion that every process must consider in the computation.

Since what we are intended to compute at every iteration in order to approximate the dominant eigenvector is actually the product between a matrix and a vector (see the description of the power method reported in the text and observe that it is easier and less expensive to compute x_k as the product between A and x_{k-1} instead of computing A^k at every iteration), the parallelization scheme is exactly the same we used in exercise 7.

In particular, remember that we are working in the assumption that the matrix size evenly divides the number of available cores.

```

1 #include <fstream>
2 #include <iostream>
3
4 #include <mpi.h>
5
6 #include "dense_matrix.hpp"
7
8 DenseMatrix power_method(const DenseMatrix & local_A, std::size_t iterations);
9
10 DenseMatrix read_matrix(const std::string & file_name);
11
12 void print_eigen_vector(const DenseMatrix & eigen_vector);
13
14 int
15 main (int argc, char *argv[])
16 {
17     MPI_Init (&argc, &argv);
18
19     int rank (0), size (0);
20     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
21     MPI_Comm_size (MPI_COMM_WORLD, &size);
22
23     DenseMatrix local_A;
24
25     local_A = read_matrix(argv[1]);
26
27     DenseMatrix eigen_vector = power_method(local_A, 6);
28
29     print_eigen_vector(eigen_vector);
30
31     MPI_Finalize ();
32     return 0;
33 }
34
35
36 DenseMatrix read_matrix(const std::string & file_name)
37 {
38     int rank (0), size (0);
39     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
40     MPI_Comm_size (MPI_COMM_WORLD, &size);
41
42     unsigned n(0);

```

```

43
44 DenseMatrix full_A;
45
46 if (rank == 0)
47 {
48     std::ifstream f_stream (file_name);
49     full_A.read (f_stream);
50     n = full_A.get_n_cols();
51 }
52
53
54 MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
55
56 const unsigned stripe = n / size;
57
58 DenseMatrix local_A(stripe,n);
59
60 MPI_Scatter (full_A.data (), stripe * n, MPI_DOUBLE,
61               local_A.data (), stripe * n, MPI_DOUBLE,
62               0, MPI_COMM_WORLD);
63
64 return local_A;
65
66 }
67
68 void print_eigen_vector(const DenseMatrix & eigen_vector)
69 {
70
71     int rank (0), size (0);
72     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
73     MPI_Comm_size (MPI_COMM_WORLD, &size);
74
75     if (rank == 0)
76     {
77         std::cout << eigen_vector.get_n_rows() << " "
78             << eigen_vector.get_n_cols() << "\n";
79
80         for (std::size_t i = 0; i < eigen_vector.get_n_rows(); ++i)
81             for (std::size_t j = 0; j < eigen_vector.get_n_cols(); ++j)
82             {
83                 std::cout << eigen_vector(i, j);
84                 std::cout << (eigen_vector.get_n_cols() - j == 1 ? "\n" : " ");
85             }
86     }
87
88 }
89
90
91 DenseMatrix power_method(const DenseMatrix & local_A, std::size_t iterations)
92 {
93     int rank (0), size (0);
94     MPI_Comm_rank (MPI_COMM_WORLD, &rank);

```

```

95 MPI_Comm_size (MPI_COMM_WORLD, &size);
96
97 const unsigned n = local_A.get_n_cols();
98
99 const unsigned stripe = local_A.get_n_rows();
100
101 DenseMatrix full_eigen_vector(n, 1.0);
102
103 for (std::size_t it_n=0; it_n < iterations; ++it_n)
104 {
105     DenseMatrix local_X = local_A * full_eigen_vector;
106
107     MPI_Allgather (local_X.data (), stripe , MPI_DOUBLE,
108                     full_eigen_vector.data (), stripe , MPI_DOUBLE,
109                     MPI_COMM_WORLD);
110 }
111
112 return full_eigen_vector;
113 }
```

Note: the program can be compiled through the following command:

```
mpicxx -Wall -std=c++11 main.cpp matrix.cpp dense_matrix.cpp -o power_method
```

and it is intended to be run by passing as argument the file with the input matrix, i.e. by writing, for instance:

```
mpiexec -np 2 ./power_method A.txt
```

Exercise 9 Search Engine (*exam 07/07/2017*)

An external library defines in the `search_engine` namespace a class, `engine`, that allows to look up keywords in a set of web pages (knowledge base in the following) and retrieve a vector of relevant URLs. Starting from the `main` function presented below, develop with MPI a parallel program with the same functionality.

```

1 int
2 main (void)
3 {
4     // create the knowledge base
5     const search_engine::knowledge_base kb = search_engine::create_default_kb ();
6     // create the search engine and initialize it to work on the knowledge base
7     const search_engine::engine libero (kb);
8
9     std::list<std::string> searched_sentence;
10    std::string word;
11    // read keywords to be serched in the KB
12    while (std::cin >> word) searched_sentence.push_back (word);
13
14    std::vector<std::string> links =
15        libero.search (searched_sentence.cbegin (), searched_sentence.cend ());
16
17    for (const std::string & url: links)
```

```

18  {
19      std::cout << url << std::endl;
20  }
21
22  return 0;
23 }
```

Notice that the search engine accesses an internal knowledge base in order to associate words to prebuilt lists of interesting URLs and returns the intersection of all the relevant sets after a lookup. In the MPI version assume that the knowledge base can be safely created and accessed across all the processes. Furthermore, the master node is in charge of collecting the partial search results and performing the final intersection.

Hint: to perform the set intersection, you can rely on the function `set_intersection`, implemented in `<algorithm>`³. It receives five iterators as parameters: the boundaries of two sorted ranges `[first1, last1]` and `[first2, last2]` and an iterator to the location where the result should be saved. The resulting intersection is sorted. The function returns an iterator to the end of the constructed range. Notice that the two ranges are assumed to be already sorted, while the container where we want to store the result must have enough space to store all elements. As an example, the method can be used as follows:

```

std::vector<int> v1 = {1,2,3,4,5,6};
std::vector<int> v2 = {2,4,6,8};
std::vector<int> v(std::min(v1.size(),v2.size()));
auto it = std::set_intersection(v1.cbegin(), v1.cend(), v2.cbegin(), v2.cend(),
                               v.begin());
std::resize(it - v.begin());
```

Exercise 9 - Solution

In the serial version, `searched_sentence` was defined as a `std::list<std::string>`; here we go with a `std::vector<std::string>` because we need the possibility to randomly access the elements. In order to perform the parallelization, indeed, we have to keep track of which rank should take care of which word; the choice of a `std::vector` allows us to use just one variable to both loop through the container and associate words with ranks.

An alternative would be to keep a `std::list` and to use an iterator and a counter in $\mathbb{Z}/p\mathbb{Z}$, where p is the number of processes.

Starting from line 41, the master node sends out the keywords in evenly divided portions; every other process receives them and stores them in a partial vector.

In this context, we cannot use `MPI_Scatter` on the global vector of keywords `searched_sentence` as we do when we have, for example, a vector of `int` or `doubles` (see Exercise 4 or Exercise 5). Each `std::string` in C++, indeed, is represented as an array of characters and it is not a type that can be directly communicated in MPI (as datatype, you can only send or receive an `MPI_CHAR`).

In order to communicate each string, therefore, as we did, for instance, in Exercise 1 or in Exercise 11, first of all we send the length of the string itself, in order to be able to resize the receiving buffer with the proper size and thus to avoid reading or writing improperly on the memory.

Notice that, in order to be able to use the method `std::set_intersection`, as it is done in line 113, the containers to be intersected must be sorted. Therefore, the method `std::sort` is called in line 83 on the partial result computed by every processor.

³see www.cppreference.com or www.cplusplus.com for further details

The sorting procedure can be executed either by each processor with its partial result (as it is done in the proposed implementation) or by rank 0 right before the intersection. In this second case, notice that only the vector `its_urls` should be sorted between lines 109 and 112, because the result of `std::set_intersection`, i.e. `my_urls`, is still sorted. Since, due to the implementation of `std::sort`, each call of the method requires to scan the whole vector, the number of times the function is used should be limited as much as possible in order to avoid useless operations.

```

1 #include <algorithm>
2 #include <cstring>
3 #include <iostream>
4 #include <iterator>
5 #include <string>
6 #include <vector>
7
8 #include <mpi.h>
9
10 #include "engine.hh"
11 #include "knowledge_base.hh"
12
13 int
14 main (int argc, char * argv[])
15 {
16     MPI_Init (&argc, &argv);
17
18     int rank, size;
19     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
20     MPI_Comm_size (MPI_COMM_WORLD, &size);
21
22     std::vector<std::string> searched_sentence;
23
24     // read keywords from master
25     if (rank == 0)
26     {
27         std::string word;
28         while (std::cin >> word) searched_sentence.push_back (word);
29     }
30
31     unsigned word_count = searched_sentence.size ();
32     MPI_Bcast (&word_count, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
33
34     // compute how many words every process will take care of
35     unsigned local_share = word_count / size;
36     if (rank < word_count % size) ++local_share;
37     std::vector<std::string> my_part (local_share);
38
39     if (rank == 0)
40     {
41         for (std::size_t i = 0; i < searched_sentence.size (); ++i)
42         {
43             const std::size_t dest = i % size;
44
45             if (dest > 0)

```

```

46 {
47     std::string send_buffer = searched_sentence[i];
48     unsigned buffer_size = send_buffer.size();
49     MPI_Send (&buffer_size, 1, MPI_UNSIGNED, dest, 0,
50               MPI_COMM_WORLD);
51     MPI_Send (&send_buffer[0], buffer_size, MPI_CHAR,
52               dest, 1, MPI_COMM_WORLD);
53 }
54 else
55 {
56     my_part[i/size] = searched_sentence[i];
57 }
58 }
59 }
60 else
61 {
62     for (unsigned i = 0; i < local_share; ++i)
63     {
64         MPI_Status status;
65         unsigned buffer_size;
66         MPI_Recv (&buffer_size, 1, MPI_UNSIGNED, 0, 0,
67                   MPI_COMM_WORLD, &status);
68         std::string recv_buffer(buffer_size,'\\0');
69         MPI_Recv (&recv_buffer[0], buffer_size, MPI_CHAR,
70                   0, 1, MPI_COMM_WORLD, &status);
71         my_part[i] = recv_buffer;
72     }
73 }
74
75 // create the knowledge base
76 const search_engine::knowledge_base kb = search_engine::create_default_kb ();
77 // create the search engine and initialize it to work on the knowledge base
78 const search_engine::engine libero (kb);
79 // do partial search with the engine
80 std::vector<std::string> my_urls =
81     libero.search (my_part.cbegin (), my_part.cend ());
82
83 std::sort (my_part.begin (), my_part.end ());
84
85 /* everyone sends its partial search result, while master
86 * both collects and intersects them
87 */
88 if (rank == 0)
89 {
90     for (int r = 1; r < size; ++r)
91     {
92         unsigned its_url_count = 0;
93         MPI_Status status;
94         MPI_Recv (&its_url_count, 1, MPI_UNSIGNED,
95                   r, 2, MPI_COMM_WORLD, &status);
96
97         std::vector<std::string> its_urls (its_url_count);

```

```

98
99     for (std::string & url: its_urls)
100    {
101        unsigned buffer_size;
102        MPI_Recv (&buffer_size, 1, MPI_UNSIGNED, r, 3,
103                  MPI_COMM_WORLD, &status);
104        std::string recv_buffer(buffer_size,'\\0');
105        MPI_Recv (&recv_buffer[0], buffer_size, MPI_CHAR,
106                  r, 4, MPI_COMM_WORLD, &status);
107        url = recv_buffer;
108    }
109
110    std::vector<std::string> tmp;
111    tmp.resize (std::min (my_urls.size (), its_urls.size ()));
112
113    auto it = std::set_intersection (my_urls.cbegin (),
114                                    my_urls.cend (),
115                                    its_urls.cbegin (),
116                                    its_urls.cend (),
117                                    tmp.begin());
118    tmp.resize (it - tmp.begin());
119
120    my_urls.swap (tmp);
121 }
122 }
123 else
124 {
125     const unsigned url_count = my_urls.size ();
126     MPI_Send (&url_count, 1, MPI_UNSIGNED, 0, 2, MPI_COMM_WORLD);
127
128     for (const std::string & url: my_urls)
129     {
130         unsigned buffer_size = url.size();
131         MPI_Send (&buffer_size, 1, MPI_UNSIGNED, 0, 3, MPI_COMM_WORLD);
132         MPI_Send (&url[0], buffer_size, MPI_CHAR,
133                   0, 4, MPI_COMM_WORLD);
134     }
135 }
136
137 // output from master
138 if (rank == 0)
139 {
140     for (const std::string & url: my_urls)
141     {
142         std::cout << url << std::endl;
143     }
144 }
145
146 MPI_Finalize ();
147 return 0;
148 }
```

Exercise 10 Word-Count

The goal of this exercise is to implement a parallel program that counts the occurrences of all the words in a given file (we assume that it has already been splitted in subfiles, whose number should be equal to the number of available ranks). The counter must be case-insensitive and must discard the punctuation marks, so that, for example both an input file like this:

```
Cplusplus  
CPlusPlus  
cPlusPlus  
CPLUSPLUS  
cplusplus  
CPlusPLUS
```

and an input file like this:

```
cplus?plus  
c-plus-plus  
?cplusplus?  
cplusplus$plus$\ncplusplus^^  
_c_plus_plus_  
return cplusplus: 6.
```

Exercise 10 - Solution

The steps performed by the algorithm are the following: reading from file, parsing the different words, counting them and finally printing out the results. Parallelization scheme: in this case, the parallelization does not concern the data that will be read from each core (since every process has its own input file), but the words that it has to count and store after the initial reading. The partition is done according to the alphabetical order (for example: if two cores are available, `process 0` will count the words that belong to the first half of the alphabet, while `process 1` will count the remainig ones). In particular, if we have two cores (and thus two input files), the program works in the following way:

- `rank 0` and `rank 1` read their input files, they parse the words that they find there and they count the occurences without considering the alphabetical order
- `rank 0` communicates to `rank 1` the words that it found in its input file, but that belong to the second half of the alphabet (and then that are supposed to be stored by `rank 1`) and viceversa.

The container chosen to store the words and their relative counter is a `std::map<std::string,unsigned>`.

In this way, every time the program reads a word, it can easily check if it is already stored in the map (and then it only has to increment by one the relative counter) or not (and then it has to create a new element with counter equal to one).

The full code of the solution is provided below.

Other than `rank()` and `size()` (see Exercise 2, file "mpi_helpers.hh"), the most important function declared in file "MPI_helpers.hh" is

```
std::pair<char,char> get_processor_bounds (unsigned);
```

(see the implementation in MPI_helpers.cc). It performs the block partition of the alphabet, i.e., it is intended to return in a `std::pair<char,char>` the first and the last letter that one specific processor (whose rank is passed as a parameter to the function) will take care of.

```
***** file MPI_helpers.hh *****
```

```

1 #ifndef __WORD_COUNTING_MPI_HELPERS__
2 #define __WORD_COUNTING_MPI_HELPERS__
3
4 #include <utility>
5
6 namespace mpi
7 {
8     int
9     rank (void);
10
11    int
12    size (void);
13
14    std::pair<char, char>
15    get_processor_bounds (unsigned);
16 }
17
18 #endif // __WORD_COUNTING_MPI_HELPERS__

***** file MPI_helpers.cc *****
1 #include <mpi.h>
2
3 #include "MPI_helpers.hh"
4
5 namespace mpi
6 {
7     int
8     rank (void)
9     {
10         int rk;
11         MPI_Comm_rank (MPI_COMM_WORLD, &rk);
12         return rk;
13     }
14
15     int
16     size (void)
17     {
18         int sz;
19         MPI_Comm_size (MPI_COMM_WORLD, &sz);
20         return sz;
21     }
22
23     std::pair<char, char>
24     get_processor_bounds (unsigned rk)
25     {
26         constexpr unsigned letters = 26;
27         const unsigned sz = size ();
28         const unsigned portion = letters / sz;
29         unsigned first, second;
30         if (rk < letters % sz)
31         {
32             first = rk * (portion + 1);

```

```

33     second = first + portion + 1;
34 }
35 else
36 {
37     first = (letters % sz) * (portion + 1) +
38         (rk - letters % sz) * portion;
39     second = first + portion;
40 }
41 return std::make_pair ('a' + first, 'a' + second);
42 }
43 }
```

The class `word_counter` implements the methods useful to parse and count the words in a given file. In particular, the method `void update_counts (std::istream &)` receives as parameter an object of type `std::istream` and, until it reads a word, it calls the method

`void update_word (key_type const&, mapped_type)`

in order to update the relative counter. `Update_word`, in turn, calls the method

`key_type parse_key (key_type const&),`

which is defined as private because it is not intended to be used outside of the class. The method `parse_key` removes from the word that it receives as parameter all the punctuation marks and it transforms all the characters in lower case, so that the counter will consider the words independently of the letters' cases. Once the parsing has been performed, `update_word` looks for the word in the map `counts`; if it already exists, it increments by `count`, which is equal to 1, the relative counter. Otherwise, it creates a new instance in the map with key equal to the word and counter equal to `count`.

The method `void keep_between(char,char)` erases from the map `counts` all the words that begin with a letter which does not stay, in the alphabet, between the characters passed as parameters. This method will be used at the end of the program, after the communication unit: indeed, every process is intended to count all the words read from its input file, independently of their position in the alphabet, but it has to store only those belonging to its portion of the alphabet (determined by the function `get_processor_bounds`, see `MPI_helpers.hh`).

***** file `word_counter.hh` *****

```

1 #ifndef __WORD_COUNTER_HH__
2 #define __WORD_COUNTER_HH__
3
4 #include <iostream>
5 #include <map>
6 #include <string>
7
8 namespace word_counting
9 {
10     class word_counter
11     {
12     private:
13         typedef std::map<std::string, unsigned> count_container;
14
15     public:
16         typedef count_container::key_type key_type;
17         typedef count_container::mapped_type mapped_type;
18         typedef count_container::value_type value_type;
19         typedef count_container::const_iterator const_iterator;
20 }
```

```

21 private:
22     count_container counts;
23
24     key_type parse_key (key_type const&z);
25
26 public:
27     void update_counts (std::istream &z);
28
29     void update_word (key_type const&, mapped_type);
30
31     void keep_between (char begin, char end);
32
33     const_iterator lower_bound (key_type const&) const;
34
35     const_iterator upper_bound (key_type const&) const;
36
37     const_iterator cbegin (void) const;
38
39     const_iterator cend (void) const;
40
41     const_iterator begin (void) const;
42
43     const_iterator end (void) const;
44 };
45 }
46
47 #endif // __WORD_COUNTER_HH__

```

***** file word_counter.cc *****

```

1 #include <algorithm>
2 #include <cctype>
3 #include <iterator>
4
5 #include "word_counter.hh"
6
7 namespace word_counting
8 {
9     word_counter::key_type
10    word_counter::parse_key (key_type const& word)
11    {
12        key_type key;
13        std::remove_copy_if (word.cbegin (), word.cend (),
14                             std::back_inserter (key), ispunct);
15        std::transform (key.begin (), key.end (),
16                       key.begin (), tolower);
17        return key;
18    }
19
20    void
21    word_counter::update_counts (std::istream & input)
22    {
23        key_type word;

```

```

24     while (input >> word) update_word (word, 1);
25 }
26
27 void
28 word_counter::update_word (key_type const& word, mapped_type count)
29 {
30     const key_type key = parse_key (word);
31     if (not key.empty ())
32     {
33         count_container::iterator match = counts.find (key);
34         if (match != counts.end ()) match -> second += count;
35         else counts[key] = count;
36     }
37 }
38
39 void
40 word_counter::keep_between (char begin, char end)
41 {
42     const std::string b {begin}, e {end};
43     const const_iterator first = lower_bound (b),
44         last = upper_bound (e);
45     counts.erase (counts.cbegin (), first);
46     counts.erase (last, counts.cend ());
47 }
48
49 word_counter::const_iterator
50 word_counter::lower_bound (key_type const& k) const
51 {
52     return counts.lower_bound (k);
53 }
54
55 word_counter::const_iterator
56 word_counter::upper_bound (key_type const& k) const
57 {
58     return counts.upper_bound (k);
59 }
60
61 word_counter::const_iterator
62 word_counter::cbegin (void) const
63 {
64     return counts.cbegin ();
65 }
66
67 word_counter::const_iterator
68 word_counter::cend (void) const
69 {
70     return counts.cend ();
71 }
72
73 word_counter::const_iterator
74 word_counter::begin (void) const
75 {

```

```

76     return cbegin ();
77 }
78
79     word_counter::const_iterator
80     word_counter::end (void) const
81 {
82     return cend ();
83 }
84 }
```

In the **main** function (see below the file "main.cpp"), first of all, every process must get its own input file. The main assumption lines 25 to 35 are based on is the fact that the input file, whose name is, for example, InputFile, has been splitted into a number of subfiles equal to the number of available cores. In case of two cores, for example, the first one (i.e. **rank 0**) will read a file called InputFile_0.txt, while the second one will read InputFile_1.txt. In lines 25 to 35, the base name of the input file (in this example, the string "InputFile_") is read by **process 0**, which communicates it to the other cores, so that each one of them is able to construct the name of its own input, adding to the base name the number of its rank and the extension ".txt".

From line 36 to 38, given the input file, each core opens it and it calls the method

word_counter::update_counts

(see "word_counter.hh") to read and count the words that are stored there.

At line 40, each core has in **wc** a **std::map<std::string,unsigned>** in which all the words from its input file are stored with their relative counter. The next step of the program is the communication unit: every process must communicate the words that it counted to the core that is intended to store them, given their position in the alphabet. For example, if the input files for **rank 0** and **rank 1** are, respectively,

<pre># file InputFile_0.txt Cplusplus cplusplus python</pre>	<pre># file InputFile_1.txt Cplusplus !python pyThoN</pre>
--	--

rank 0 counts in **wc** "cplusplus: 2" and "python: 1", while **rank 1** counts "cplusplus: 1" and "python: 2". In case of two available cores, the first one is intended to store all the words between a and m, while the second one is intended to store those between n and z. Therefore, **rank 0** must keep "cplusplus: 2" and communicate to **rank 1** "python: 1" (and the other way round).

The choice of a **std::map<std::string,unsigned>** as container to store the words in **wc** make the process simpler: since it is an ordered container (with respect to the alphabetical order of the strings used as keys), the methods **upper_bound** and **lower_bound** (conveniently overloaded in the **word_counter** class in order to make them work with an object of that type) automatically look for the words that stay inside the boundaries specified by **get_processor_bounds**.

At the end of the communication unit, at line 102, every process stores all the words that belong to its portion of the alphabet (counted by itself and by the other cores) and it has communicated those that it had counted but that had to be stored by other ranks. Therefore, the method **word_counter::keep_between** is called in order to discard those words that do not belong to the proper portion of the alphabet (keeping the example considered before, **rank 0** now stores "cplusplus: 3" and it has to discard "python: 1").

Finally, each core prints out its own result ("cplusplus: 3" for **rank 0** and "python: 3" for **rank 1**).

***** file main.cc *****

```

1 #include <fstream>
2 #include <iostream>
3 #include <mpi.h>
4 #include <sstream>
5 #include <string>
6 #include <vector>
7
8 #include "MPI_helpers.hh"
9 #include "word_counter.hh"
10
11 int
12 main (int argc, char * argv[])
13 {
14     constexpr int counts_tag = 1;
15     constexpr int words_tag = 2;
16
17     MPI_Init (&argc, &argv);
18
19     const unsigned rank = mpi::rank ();
20     const unsigned size = mpi::size ();
21
22     word_counting::word_counter wc;
23
24     {
25         std::string basename;
26         if (rank == 0) std::cin >> basename;
27
28         unsigned length = basename.size ();
29         MPI_Bcast (&length, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
30
31         basename.resize (length);
32         MPI_Bcast (&basename[0], length, MPI_CHAR, 0, MPI_COMM_WORLD);
33
34         std::ostringstream filename;
35         filename << basename << rank << ".txt";
36         std::ifstream input (filename.str ());
37         if (input) wc.update_counts (input);
38     }
39
40     for (unsigned receiver = 0; receiver < size; ++receiver)
41     {
42         if (rank != receiver)
43         {
44             std::ostringstream words_builder;
45             std::vector<unsigned> counts;
46
47             const std::pair<char, char> bounds = mpi::get_processor_bounds (receiver);
48             const std::string low {bounds.first}, up {bounds.second};
49             typedef word_counting::word_counter::const_iterator const_iterator;
50             const const_iterator end = wc.upper_bound (up);
51
52             for (const_iterator it = wc.lower_bound (low);

```

```

53     it != end; ++it)
54     {
55         words_builder << it->first << std::endl;
56         counts.push_back (it->second);
57     }
58
59     const std::string words = words_builder.str ();
60     const unsigned words_length = words.size ();
61     MPI_Send (&words_length, 1, MPI_UNSIGNED, receiver,
62               words_tag, MPI_COMM_WORLD);
63     MPI_Send (&words[0], words_length, MPI_CHAR,
64               receiver, words_tag, MPI_COMM_WORLD);
65
66     const unsigned num_counts = counts.size ();
67     MPI_Send (&num_counts, 1, MPI_UNSIGNED, receiver,
68               counts_tag, MPI_COMM_WORLD);
69     MPI_Send (counts.data (), num_counts, MPI_UNSIGNED,
70               receiver, counts_tag, MPI_COMM_WORLD);
71 }
72 else
73 {
74     for (unsigned sender = 0; sender < size; ++sender)
75     {
76         if (rank != sender)
77         {
78             unsigned length {0};
79
80             MPI_Recv (&length, 1, MPI_UNSIGNED, sender, words_tag,
81                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
82             std::string words (length, '?');
83             MPI_Recv (&words[0], length, MPI_CHAR, sender,
84                       words_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
85
86             MPI_Recv (&length, 1, MPI_UNSIGNED, sender, counts_tag,
87                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
88             std::vector<unsigned> counts (length);
89             MPI_Recv (counts.data (), length, MPI_UNSIGNED, sender,
90                       counts_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
91
92             std::istringstream words_stream (words);
93             for (unsigned count: counts)
94             {
95                 std::string word;
96                 words_stream >> word;
97                 wc.update_word (word, count);
98             }
99         }
100    }
101 }
102 }
103
104 const std::pair<char, char> bounds = mpi::get_processor_bounds (rank);

```

```

105 wc.keep_between (bounds.first, bounds.second);
106
107 for (word_counting::word_counter::value_type const& pair: wc)
108 {
109     std::ostringstream builder;
110     builder << pair.first << ":" << pair.second << "\n";
111     std::cout << builder.str ();
112 }
113
114 MPI_Finalize ();
115 return 0;
116 }
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 main.cc MPI_helpers.cc word_counter.cc -o wordcount
```

Notice that all the header and source files the program is composed by are provided in the subdirectory `src/`. An additional subdirectory, namely `test/`, is provided: it stores a list of input files that can be used to test the program. They are organized as follows: for each possible test (e.g. "capitalization", "punctuation", and so on), the two files "`<test name>_0.txt`" and "`<test name>_1.txt`" (e.g. "capitalization_0.txt" and "capitalization_1.txt") are the input files to be read by `rank 0` and `rank 1`, respectively. The expected output of the program is provided in the corresponding file "`expected_<test name>.txt`".

The program is intended to be run by passing through the command line the basename of the files that should be read by all processors (for instance, "capitalization_").

Since the program is built in folder `src/`, the produced executable and the file to be used for testing are in different directories. You can proceed in different ways; among them:

1. you can execute the program from `src/` folder through

```
mpiexec -np 2 ./wordcount
```

and pass to `std::cin` the relative path of the required test, i.e. type, for instance

```
..//test/capitalization_
```

2. you can copy the files "`<test name>_0.txt`" and "`<test name>_1.txt`" in the `src/` folder, execute the program through

```
mpiexec -np 2 ./wordcount
```

and then type the basename of the test, for instance

```
capitalization_
```

3. you can go in folder `test/` and execute the program through

```
mpiexec -np 2 ..//src/wordcount
```

then proceed as before to pass the test name to `std::cin`.

4. you can rely on the files "`input_<test name>.txt`" that you have in folder `test/`, that store exactly the relative path `..//test/<test name>_` considered in example 1. In this case, you can execute the program from folder `src/` by writing, for instance:

```
mpiexec -np 2 ./wordcount < ..//test/input_capitalization.txt
```

or you can move the file "`input_<test name>.txt`" in folder `src/` and write:

```
mpiexec -np 2 ./wordcount < input_capitalization.txt
```

Exercise 11 Parallel Grep (*exam 19/02/2018*)

grep is a popular Linux command that helps in searching strings within text files. For example, this simple invocation of the command:

```
$ grep pattern filename
```

will print all the lines where pattern appears preceded by their number in filename. So if the file "dogs.txt" includes the following text:

```
Dido is my dog. He loves to get outdoors  
and in the forest, surrounded by a world of exciting scents,  
he is very happy. I love taking Dido with me  
when I go on holiday; sure that means staying reasonably close to  
home — no exotic beaches for a start!
```

The output of:

```
$ grep Dido dogs.txt
```

is:

```
1: Dido is my dog. He loves to get outdoors  
3: he is very happy. I love taking Dido with me
```

Implement in MPI a parallel program similar to grep. Assume that any input file filename is already split in even parts and every process can access a partial file called filename-<rank>, where <rank> is a placeholder for the rank of each process. In the above example, there are "dogs.txt-0", "dogs.txt-1", and so on. In particular, you are only required to provide the main source file and the implementation of the print_result function, starting from the code that follows.

PS: If you try grep at home, pay attention that its default behavior is to output only lines that match pattern, to enable the behavior presented in this exercise, i.e. to print also line numbers, you need the -n flag.

```
1 // grep.hh  
2 #ifndef GREP_HH  
3 #define GREP_HH  
4  
5 #include <vector>  
6 #include <string>  
7 #include <utility>  
8  
9 namespace grep  
10 {  
11     typedef std::pair< unsigned, std::string > number_and_line;  
12  
13     typedef std::vector< number_and_line > lines_found;  
14  
15     void search_string (const std::string & file_name,  
16                         const std::string & search_string,  
17                         lines_found & lines, unsigned & local_lines_number);  
18 }
```

```

19 void print_result (const lines_found & lines, unsigned local_lines_number);
20 }
21
22 #endif // GREP_HH

```

```

1 // grep.cc
2 #include <fstream>
3 #include <iostream>
4 #include <sstream>
5
6 #include <mpi.h>
7
8 #include "grep.hh"
9
10 namespace grep
11 {
12     void search_string (const std::string & file_name,
13                         const std::string & search_string,
14                         lines_found & local_filtered_lines,
15                         unsigned & local_lines_number)
16     {
17         int rank (0);
18         MPI_Comm_rank (MPI_COMM_WORLD, &rank);
19         const unsigned rk (rank);
20         local_lines_number = 0;
21         std::ostringstream file_name_builder;
22         file_name_builder << file_name << '-' << rk;
23         const std::string local_file_name = file_name_builder.str ();
24         std::ifstream f_stream (local_file_name);
25
26         // read input file line by line
27         for (std::string line; std::getline (f_stream, line); )
28         {
29             //increment local number of lines
30             ++local_lines_number;
31
32             if (line.find (search_string) != std::string::npos)
33             {
34                 local_filtered_lines.push_back ({local_lines_number, line});
35             }
36         }
37     }
38
39     void print_result (const lines_found & local_filtered_lines,
40                       unsigned local_lines_number)
41     {
42         ...
43     }
44 }

```

Exercise 11 - Solution

First of all, the implementation of the function `grep::print_result`, written in file "grep.cc", is reported below. The idea is the following: the function receives as parameters a vector of `std::pair<unsigned, std::string>` (hidden in the user defined type `lines_found`, see lines 11 and 13 in the file "grep.hh") and an `unsigned`, which store, respectively, the number and content of the lines found by the function `search_string` and the number of lines in the input file read by each process. The function `search_string` (see lines 12 to 37 in "grep.cc"), indeed, receives the name of the file, the string that has to be searched and two references to a `lines_found` object and to an `unsigned` in which it saves the results of the research. During the execution of the function, each process generates the name of its input file using the assumption described in the text (the same that we did, for example, in exercise 10), it reads its input file line by line and it checks the presence of the string that has to be searched. It finally stores in `local_lines_number` the total number of lines of its input file. This information, in particular, is needed in order to reconstruct the line number of each string in the global file. Indeed, consider the example seen in the text of the exercise: if the global input file "dogs.txt" is subdivided in

```
# dogs.txt-0
```

```
Dido is my dog. He loves to get outdoors  
and in the forest, surrounded by a world of exciting scents,
```

```
# dogs.txt-1
```

```
he is very happy. I love taking Dido with me  
when I go on holiday; sure that means staying reasonably close to  
home — no exotic beaches for a start!
```

the function `search_string`, receiving as parameters "dogs.txt" and "Dido", will produce on rank 0

1: Dido is my dog. He loves to get outdoors

and, on rank 1

1: he is very happy. I love taking Dido with me

because in both files the line with the word "Dido" is the first one. We need to know the total number of lines in "dogs.txt-0" and "dogs.txt-1", respectively, in order to reconstruct the fact that the first line of "dogs.txt-1" is actually the third one in the global file "dogs.txt".

Given these informations, the function `print_result` initializes on the master node the variable `global_filtered_lines` with the lines found by process 0 and then it appends the lines found by all the other processes.

In order to do this, rank 0 receives, first of all, the number of lines in each local file, then the number of lines found by each core and, finally, in the for loop of lines 65 to 79, the single elements (i.e. the content of the lines and the relative numbers).

```
39 void print_result (const lines_found & local_filtered_lines,  
40                      unsigned local_lines_number)  
41 {  
42     int rank (0), size (0);  
43     MPI_Comm_rank (MPI_COMM_WORLD, &rank);  
44     MPI_Comm_size (MPI_COMM_WORLD, &size);  
45     const unsigned rk (rank);  
46     const unsigned sz (size);  
47 }
```

```

48 if (rk == 0)
49 {
50     lines_found global_filtered_lines (local_filtered_lines);
51
52     // append to global_filtered_lines the lines found by other processes
53     for (unsigned remote_process = 1; remote_process < sz; ++remote_process)
54     {
55         unsigned remote_lines;
56         // receive the number of lines in the local file
57         MPI_Recv (&remote_lines, 1, MPI_UNSIGNED, remote_process, 0,
58                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
59         // receive the number of lines to be received
60         unsigned filtered_lines;
61         MPI_Recv (&filtered_lines, 1, MPI_UNSIGNED, remote_process,
62                   0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
63
64         // receive lines
65         for (unsigned i = 0; i < filtered_lines; ++i)
66         {
67             unsigned current_line_number;
68             MPI_Recv (&current_line_number, 1, MPI_UNSIGNED, remote_process,
69                       0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
70             unsigned length = 0;
71             MPI_Recv (&length, 1, MPI_UNSIGNED, remote_process,
72                       0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
73             std::string new_line (length, '\0');
74             MPI_Recv (&new_line[0], length, MPI_CHAR, remote_process,
75                       0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
76             // add new line and compute offset
77             global_filtered_lines.push_back (std::make_pair (local_lines_number +
78                                         current_line_number, new_line));
79         }
80
81         local_lines_number += remote_lines;
82     }
83
84     // print all filtered lines
85     for (const number_and_line & n_l : global_filtered_lines)
86     {
87         std::cout << n_l.first << ":" << n_l.second << std::endl;
88     }
89 }
90 else
91 {
92     // send to rank 0 the number of lines of the local file
93     // in a way the line number can be updated
94     MPI_Send (&local_lines_number, 1, MPI_UNSIGNED, 0, 0,
95               MPI_COMM_WORLD);
96     // send to rank 0 the number of filtered lines
97     const unsigned filtered_lines = local_filtered_lines.size();
98     MPI_Send (&filtered_lines, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);

```

```

99     // send lines
100    for (const number_and_line & n_l : local_filtered_lines)
101    {
102        // send line number
103        MPI_Send (&n_l.first, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);
104        const unsigned length = n_l.second.size();
105        // send string length
106        MPI_Send (&length, 1, MPI_UNSIGNED, 0, 0, MPI_COMM_WORLD);
107        // send string
108        MPI_Send (&(n_l.second)[0], length, MPI_CHAR, 0, 0,
109                  MPI_COMM_WORLD);
110    }
111 }
112 }
113 }
```

```

1 // main.cc
2 #include <mpi.h>
3
4 #include "grep.hh"
5
6 int main (int argc, char * argv[])
7 {
8     MPI_Init (&argc, &argv);
9     grep::lines_found local_filtered_lines;
10    unsigned local_lines_number;
11    grep::search_string (argv[1], argv[2], local_filtered_lines, local_lines_number);
12    grep::print_result (local_filtered_lines, local_lines_number);
13    MPI_Finalize();
14    return 0;
15 }
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall grep-main.cc grep.cc -o grep
```

Exercise 12 Parallel K-means*

The aim of this exercise is to write a parallel version of k-means clustering using MPI, starting from the serial version presented in Chapter 3 Exercise 8.

Work under the assumptions that the dimension p of the Euclidean space where points belong and the number of clusters k are both way smaller than the number of points, n . The assumption on k likely applies in any case; regarding p , there might be applications where it does not hold, but this is not our problem here. These assumptions influence the choice of the parallelization scheme: to keep the communication pattern simple, consider replicating centroids everywhere, whilst the collection of points should be sliced across processes.

Design your classes to be parallelism aware. Ideally, your `main` should rely on MPI routines only for I/O and, clearly, MPI initialization, whilst constructing the `Clustering` object and launching the k-means algorithm should require exactly the same code as the serial version.

Exercise 12 - Solution

Since the solution of this exercise is based on the serial version provided in Exercise 8 of Chapter 3 (Section 3.1), here there are only the functions that somehow differ from the original solution, due to the parallelization scheme.

***** file Point.hh and Point.cc *****

These files are essentially equivalent to the corresponding files "Point.h" and "Point.cpp" presented in Exercise 8 of Chapter 3 (Section 3.1).

The main difference stays in the function `print`, which in this case returns a `std::string`. This is due to the fact that, using MPI, it is better to allow only process 0 to use standard I/O. For this reason, it is useful to define every `print` function so that it returns a `std::string`, in order to be able to communicate it easily to process 0 and make it truly print on standard output.

```

1 std::string
2 Point::print (void) const
3 {
4     std::ostringstream buffer;
5     auto it = x.cbegin ();
6     if (it != x.cend ())
7     {
8         buffer << *it++;
9     }
10    while (it != x.cend ())
11    {
12        buffer << " " << *it++;
13    }
14    buffer << std::endl;
15    return buffer.str ();
16 }
```

This choice can be useful even in more general contexts, because it enables to easily concatenate prints; for example, if we have a `Point p` that we want to print, we can write

```
std::cout << "Point p is: " << p.print() << std::endl;
```

instead of

```
std::cout << "Point p is: ";
p.print();
std::cout << std::endl;
```

The files "MPI.hh" and "MPI.cc" provide some useful routines, such that two functions that return the size and the rank of a particular process (see Exercise 2, file "mpi_helpers.hh").

***** file MPI.hh *****

```

1 #ifndef __KMEANS_MPI_HH__
2 #define __KMEANS_MPI_HH__
3
4 #include <iostream>
5 #include <mpi.h>
6 #include <string>
7
8 namespace mpi
9 {
10     int rank (MPI_Comm const&);
```

```

12 int size (MPI_Comm const&);
13
14 void tidy_output (std::ostream &, std::string const&, MPI_Comm const&);
15 }
16
17 #endif // __KMEANS_MPI_HH__

```

The most interesting function is `tidy_output`, which is designed to allow all the processes to send to rank 0 the text that must be printed; process 0, in turn, receives the text and prints it on `std::ostream &os` (which in this case will be `std::cout`, as we can see in the `main` function in file "Kmeans.cc", but in principle can be, for example, a given output file).

***** file MPI.cc *****

```

1 #include <vector>
2
3 #include "MPI.hh"
4
5 namespace mpi
6 {
7
8     int
9     rank (MPI_Comm const& comm)
10    {
11        int r;
12        MPI_Comm_rank (comm, &r);
13        return r;
14    }
15
16    int
17    size (MPI_Comm const& comm)
18    {
19        int s;
20        MPI_Comm_size (comm, &s);
21        return s;
22    }
23
24    void
25    tidy_output (std::ostream & os, std::string const& text, MPI_Comm const& comm)
26    {
27        const unsigned rk = rank (comm);
28        const unsigned sz = size (comm);
29
30        if (rk > 0)
31        {
32            const unsigned length = text.size ();
33            MPI_Send (&length, 1, MPI_UNSIGNED, 0, 0, comm);
34            MPI_Send (&text[0], length, MPI_CHAR, 0, 0, comm);
35        }
36        else
37        {
38            os << text;
39
40            for (unsigned i = 1; i < sz; ++i)
41            {

```

```

41     unsigned length = 0;
42     MPI_Recv (&length, 1, MPI_UNSIGNED, i, 0, comm,
43                 MPI_STATUS_IGNORE);
44     std::string more_text (length, '\0');
45     MPI_Recv (&more_text[0], length, MPI_CHAR, i, 0,
46                 comm, MPI_STATUS_IGNORE);
47     os << more_text;
48 }
49 }
50 }
51 }
```

The header file of the class `Centroid` is equivalent to the file "Centroid.h" that we had in Exercise 8 of Chapter 3 (Section 3.1), thus it is not reported here.

The implementation of the function `update_coords`, written in "Centroid.cc", is slightly different because of the parallelization.

In particular, since every process generates only a portion of the points that are intended to be subdivided in clusters, when we compute the new coordinates of the centroid we must collect all the partial sums given by the different cores, performing an `MPI_Allreduce` in order to get the total sum. The same method must be applied on the number of points in order to get the total value.

***** file Centroid.cc *****

```

1 #include <mpi.h>
2
3 #include "Centroid.hh"
4
5 void
6 Centroid::update_coords (std::vector<Point *> const & points)
7 {
8     std::vector<double> new_coords (x.size (), 0);
9
10    for (std::vector<Point *>::size_type i = 0; i < points.size (); ++i)
11    {
12        for (size_type j = 0; j < x.size (); ++j)
13        {
14            new_coords[j] += points[i] -> get_coord (j);
15        }
16    }
17
18    MPI_Allreduce (MPI_IN_PLACE, new_coords.data (), new_coords.size (),
19                  MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
20
21    unsigned cluster_size = points.size ();
22    MPI_Allreduce (MPI_IN_PLACE, &cluster_size, 1,
23                  MPI_UNSIGNED, MPI_SUM, MPI_COMM_WORLD);
24
25    for (double & coord: new_coords)
26    {
27        coord /= cluster_size;
28    }
29
30    x.swap (new_coords);
```

As before, the header file of the class `Clustering` is equivalent to the file "Clustering.h" that we had in Exercise 8 of Chapter 3 (Section 3.1).

In the file "Clustering.cc" there is the portion of the program where the parallelization scheme is exploited.

In particular, in the constructor (see lines 8 to 41) we have the first communication unit: since the dimension of the space p , the number of points n , the number of clusters k and the maximum number of iterations are known only in process 0, which reads them from standard input, after the initialization we have to broadcast them in order to make the proper values available on all the processes (line 15 to 18).

Furthermore, in lines 20 and 21 we fix the number of random points that every core will generate in order to perform the clustering. In particular, we assign to every core between 0 and `n%size` one point more than to the remaining processes.

The construction of the random points and their first assignment to the different clusters (lines 23 to 40) is performed exactly as in the serial version.

In the function `calc_cluster` (see lines 56 to 103), the initialization of the labels and the assignment of the points to the different clusters are performed exactly as in the serial version. The program exits the loop (see line 72) when it reaches the maximum number of iterations or when the termination condition is `true` on all the processes (that is when the old labels are equal to the new ones, i.e. when the algorithm does not perform any improvement). In order to check if the termination condition is true on all the processes, an `MPI_Allreduce` is performed (see line 98), applying the operation `MPI_LAND`, which implements the logical "and" between the arguments. This means that the variable `term_cond`, which stores the termination condition, cannot be defined, in line 69, as a `bool`: MPI routines are written in C and C language did not define the `bool` type.

Another difference between the serial and the parallel version is given by the return type of the function: in the serial version, indeed, it was defined as

```
void Clustering::calc_cluster (void)
```

while in this case it returns an `unsigned`. This is again due to the fact that, using MPI, we allow only process 0 to interact with standard output. In the serial version, indeed, at the end of the for loop that assigns the points to the different clusters, we had

```
std::cout << "Number of iterations: " << i << std::endl;
std::cout << "Final result!" << std::endl;
print_labels ();
std::cout << std::endl;
```

This cannot be done in the parallel version, unless we want every core to print out its own result (in random order). Here as in `Point` class, the functions `Clustering::print()`, `Clustering::print_labels()`, `Clustering::print_centers()` and `Clustering::print_clusters()` are designed in such a way that they return a `std::string` (the implementation is not reported here because it is basically analogous to that of `Point::print()`, see above the file "Point.cc"). Moreover, the number of iterations performed by every core is the same, because we have a common termination condition. Therefore, we chose to return the number of iteration and to print it, together with the labels generated by `calc_cluster`, directly from the `main` function (see below the file "KMeans.cc"). If we want to keep `void` as the return type of the function, we could, for example, add at the end of the for loop, between lines 95 and 97, the following lines:

```
int rank = mpi::rank(MPI_COMM_WORLD);
if (rank==0)
    std::cout << "Number of iterations: " << i << "\n";
    mpi::tidy_output (std::cout, print_labels (), MPI_COMM_WORLD);
```

relying on the function `mpi::tidy_output` (see above the file "MPI.cc" for the implementation), passing as parameter the `std::string` returned by `print_labels`, to print out the labels.

***** file Clustering.cc *****

```

1 #include <iostream>
2 #include <random>
3 #include <sstream>
4
5 #include "Clustering.hh"
6 #include "MPI.hh"
7
8 Clustering::Clustering (unsigned int dimensions, unsigned int n_points,
9                         unsigned int k_cluster, unsigned int max_iterations)
10                        : p (dimensions), n (n_points), k (k_cluster), max_it (max_iterations)
11 {
12     const unsigned rank = mpi::rank (MPI_COMM_WORLD);
13     const unsigned size = mpi::size (MPI_COMM_WORLD);
14
15     MPI_Bcast (&p, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
16     MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
17     MPI_Bcast (&k, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
18     MPI_Bcast (&max_it, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
19
20     local_n = n / size;
21     if (rank < n % size) ++local_n;
22
23     std::default_random_engine generator (rank * size * local_n);
24     std::cerr << "Process " << mpi::rank (MPI_COMM_WORLD)
25             << " value: " << generator () << std::endl;
26     std::uniform_real_distribution<double> distribution (0.0, MAX_COORD);
27
28     for (unsigned i = 0; i < local_n; ++i)
29     {
30         std::vector<double> coords (p);
31         for (double & value : coords)
32         {
33             value = distribution (generator);
34         }
35         points.push_back (coords);
36     }
37
38     labels.assign (local_n, 0);
39     centers.assign (k, std::vector<double> (p, 0.));
40     clusters.assign (k, {nullptr});
41 }
```

```

56     unsigned
57     Clustering::calc_cluster (void)
58     {
59         std::default_random_engine generator;
60         std::uniform_int_distribution<unsigned> distribution (0, k - 1);
```

```

62 //Randomly initialize labels
63 for (unsigned & label : labels)
64 {
65     label = distribution (generator);
66 }
67
68 // MPI is C, unfortunately
69 int term_cond = false;
70 unsigned i;
71
72 for (i = 0; i < max_it and not term_cond; ++i)
73 {
74     f_labels_type old_labels (labels);
75     clusters_type new_clusters (k);
76
77     // update clusters according to the labels
78     for (std::size_t j = 0; j < points.size (); ++j)
79     {
80         new_clusters[labels[j]].push_back (&points[j]);
81     }
82     clusters.swap (new_clusters);
83
84     // update centroids
85     for (std::size_t j = 0; j < centers.size (); ++j)
86     {
87         centers[j].update_coords (clusters[j]);
88     }
89
90     // assign points to new centroids
91     for (std::size_t j = 0; j < points.size (); ++j)
92     {
93         const unsigned int min_dist_idx = min_dist_index (points[j]);
94         labels[j] = min_dist_idx;
95     }
96
97     term_cond = (old_labels == labels);
98     MPI_Allreduce (MPI_IN_PLACE, &term_cond, 1,
99                     MPI_INT, MPI_LAND, MPI_COMM_WORLD);
100 }
101
102 return i;
103 }
```

***** file KMeans.cc *****

```

1 #include <iostream>
2
3 #include "Clustering.hh"
4 #include "MPI.hh"
5
6 int
7 main (int argc, char * argv[])
8 {
```

```

9 MPI_Init (&zargc, &zargv);
10
11 unsigned dimensions, points, clusters, iterations;
12 dimensions = points = clusters = iterations = 0;
13
14 const unsigned rank = mpi::rank (MPI_COMM_WORLD);
15
16 if (rank == 0)
17 {
18     std::cin >> dimensions >> points >> clusters >> iterations;
19 }
20
21 Clustering c (dimensions, points, clusters, iterations);
22
23 mpi::tidy_output (std::cout, c.print (), MPI_COMM_WORLD);
24
25 const unsigned iter = c.calc_cluster ();
26
27 if (rank == 0)
28 {
29     std::cout << "Number of iterations: " << iter << "\n";
30 }
31
32 mpi::tidy_output (std::cout, c.print_labels (), MPI_COMM_WORLD);
33 mpi::tidy_output (std::cout, c.print (), MPI_COMM_WORLD);
34
35 MPI_Finalize ();
36
37 return 0;
38 }
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall KMeans.cc Clustering.cc MPI.cc Centroid.cc Point.cc -o
KMeans
```

Additional exercise: Try to implement a second parallel version of the K-means program that reads points from the file system, instead of generating them randomly. In this case, you will need a different constructor that takes a `std::istream &` instead of the number of points. Since you cannot have more than one process read from the same file, you should prepare one file per process and use ranks to give them predictable, but unique to each process, names.⁴

Exercise 13 Parallel Gradient Descent*

The aim of this exercise is to provide a parallel program that implements the gradient descent for functions defined from \mathbb{R}^n to \mathbb{R} . The scheme of the implementation is the same seen in Exercise 5, Section 2.2 of Chapter 2.

Exercise 13 - Solution

⁴Hint: you can see, for example, what is done in exercise 10

The only files which have to be modified, with respect to the version seen in Exercise 5, Section 2.2 of Chapter 2, in order to implement the parallelization, are "FunctionMinRn.cc" and "main.cc", which therefore are the only ones reported below.

In particular, in the `main` function, we must take into account the fact that only process 0 should print the results on standard output.

```
***** file main.cc *****
1 #include <iostream>
2 #include <mpi.h>
3 #include <vector>
4
5 #include "FunctionMinRn.hh"
6 #include "MPI_helpers.hh"
7
8 int
9 main (int argc, char *argv[])
{
10    MPI_Init (&argc, &argv);
11
12    constexpr double step = 1e-2;
13    constexpr double tolerance = 1e-5;
14
15    std::vector<Monomial> terms;
16    terms.push_back (Monomial (2, {2, 0}));
17    terms.push_back (Monomial (2, {1, 1}));
18    terms.push_back (Monomial (2, {0, 2}));
19    terms.push_back (Monomial (-2, {0, 1}));
20    terms.push_back (Monomial (6, {0, 0}));
21
22    FunctionRn f;
23    for (const Monomial & m: terms)
24        f.addMonomial (m);
25
26    const Point P1 ({0, 0});
27    const Point P2 ({2, 2});
28
29    const unsigned rank = mpi::rank ();
30    if (rank == 0)
31    {
32        std::cout << "Initial Points values" << std::endl;
33        std::cout << f.eval (P1) << " " << f.eval (P2) << std::endl;
34    }
35
36    constexpr unsigned int max_iterations = 2e5;
37    FunctionMinRn minRn (f, tolerance, step, max_iterations, {-5, -4}, {7, 9});
38
39    const Point P = minRn.solve ();
40    if (rank == 0)
41    {
42        std::cout << "Final solution standard grad: "
43                  << f.eval (P) << std::endl;
44        P.print ();
```

```

46     }
47
48     const Point Q = minRn.solve_multistart (1000);
49     if (rank == 0)
50     {
51         std::cout << "Final solution multi-start: "
52             << f.eval (Q) << std::endl;
53         Q.print ();
54     }
55
56     constexpr unsigned int n_domain_steps = 100;
57     const Point R = minRn.solve_domain_decomposition (n_domain_steps, 100);
58     if (rank == 0)
59     {
60         std::cout << "Final solution domain decomposition: "
61             << f.eval (R) << std::endl;
62         R.print ();
63     }
64
65     MPI_Finalize ();
66     return 0;
67 }
```

The functions

`FunctionMinRn::solve (const Point & P)`

and

`FunctionMinRn::solve (void)`

are not affected by the parallelization, because it concernes only the way in which the points and/or the subsets of the domain are splitted between the different cores in case of multistart and domain decomposition.

In particular, the implementation of the parallel gradient descent algorithm with multistart is reported in lines 101 to 174 of the file "FunctionMinRn.cc".

First of all, an initial random point is generated on all the processes and a first evaluation of the function `FunctionMinRn::solve (const Point & P)` is given starting from that point (lines 103 to 137). Then, if `n_trials` is the number of random points that we want to use as initial data for the computation, we split it across processes passing one more element to every rank from 0 to `n_trials%size` (lines 139 to 141).

In this exercise, we do not have, as we did, for example, in Exercise 7, some data that we have to split across processes; every core, indeed, will compute its own random points. In particular, the number that every rank will compute and store in `my_trials` is precisely the number of random points that it has to generate (and thus the number of evaluations of `FunctionMinRn::solve (const Point & P)` that it has to run).

We do not need to broadcast the value of `n_trials` because it is already known by all processes.

After the splitting (lines 143 to 163), every process generates `my_trials` random points and it passes them to

`FunctionMinRn::solve (const Point & P)`,

computing its local minimum; in particular, at every iteration it stores in `f_new` the evaluation of the function f in the point computed by `FunctionMinRn::solve (const Point & P)` and it compares it with the minimum value `f_min` reached in the previous iterations, then it updates `p_min` and `f_min` if the new result is better than the previous one.

At the end of the for loop (line 165), each process stores in a `std::pair<double,int>` the value of the function at the point `p_min` and the value of its own rank.

Now comes the communication unit (lines 166 to 170). At the end, we want every process to return the point which corresponds to the global minimum, not to the local one (i.e., we want all the cores to return the same value). This is, in general, a good practice when we have a parallel function; indeed, the return value could be used, later on in the program, for further computations and therefore it is better to have the proper result in every core. In order to get this, we have to split the communication unit into two steps.

First of all, an `MPI_Allreduce` is performed in order to get in all the cores the minimum value of `f_min` and the value of the rank that has achieved this minimum. In particular, the receive buffer is set to `MPI_IN_PLACE`, which means that the correct value of the pair `minimum` computed by the `MPI_Allreduce` is stored in the pair itself (this allows not to create another useless variable and then it enables to save some memory). Furthermore, the operation `MPI_MINLOC`, performed on a pair `<value, index>` returns the global minimum of the values in `value` and the corresponding index (in our case, the rank of the process that has computed the minimum).

The previous operation is necessary because the rank saved in `minimum`, i.e. the value of `minimum.second`, must be used as root in the following broadcast, whose aim is to communicate to all the processes the value of the global minimum, which will be the return value of the function. In particular, every process constructs a `std::vector<double>`, called `min_coords`, with the coordinates of its local minimum. Then, an `MPI_Bcast` is performed: the core whose rank is equal to `minimum.second` sends its `min_coords.data()` to all the other processes, so that, after the broadcast, they all store the proper coordinates of the global minimum. The function will return a `Point` constructed by every process passing as parameter `min_coords`, which is now known everywhere.

Without `MPI_MINLOC`, each process would have to communicate its local minimum to all the other processes, so that, instead of performing the final `MPI_Bcast` described above, they would be able to directly compute the global minimum.

The last function that has to be modified in order to make the program work in parallel is `FunctionMinRn::solve_domain_decomposition`,

in which the algorithm of gradient descent with multi-start is performed after having splitted the initial domain into a given number of subsamples. In particular, the parallelization scheme will concern exactly the number of subsamples that every core has to take into account.

The full code of the function is provided in lines 177 to 267.

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 194), 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 to divide the domain in 100 rectangles).

Given the number of subspaces, we define in every process `first` and `last` (lines 201 to 212): they will be the indices of the first and the last subspace that the process must consider (in particular, if the number of subspaces does not divide evenly the number of cores, the first `n_subspaces%size` cores will consider one subspace more than the others). After the definition of `first` and `last` (line 217), every core generates its own subspaces using the function `FunctionMinRn::next_inf_limit` (see Exercise 5, Section 2.2 of Chapter 2, for the implementation).

The computation of the local minimum in every core (lines 214 to 256) follows as in the non parallel case, using at every iteration of the for loop that runs from `first` to `last` the function `FunctionMinRn::solve_multistart` presented above.

At the end of the computation (lines 258 to 267), the global minimum is communicated and returned by all the cores exactly as in `FunctionMinRn::solve_multistart`.

```
***** file FunctionMinRn.cc *****
```

```
1 #include <cmath>
2 #include <iostream>
3 #include <mpi.h>
4 #include <random>
5 #include <utility>
6
7 #include "FunctionMinRn.hh"
8 #include "FunctionRn.hh"
9 #include "Monomial.hh"
10 #include "MPI_helpers.hh"

101 Point FunctionMinRn::solve_multistart (unsigned int n_trials) const
102 {
103     const unsigned rank = mpi::rank ();
104     std::default_random_engine generator (rank * big_number);
105     std::uniform_real_distribution<double> distribution (0, 1);
106     std::vector<double> random_coords;
107     debug_info ("Running multi-start");

108     //generate random coords
109     for (std::size_t i = 0; i < inf_limits.size (); ++i)
110     {
111         debug_info ("Pick random value");
112         const double rand_val = distribution (generator);
113         random_coords.push_back (inf_limits[i] +
114             (sup_limits[i] - inf_limits[i])
115             * rand_val);
116     }
117
118     if (debug)
119     {
120         for (double rc: random_coords)
121             std::cout << rc << " ";
122         std::cout << std::endl;
123     }
124
125     debug_info ("Creating random point");
126     Point random_point = Point (random_coords);
127
128     if (debug)
129     {
130         std::cout << "First random point" << std::endl;
131         random_point.print ();
132     }
133
134     Point p_min = solve (random_point);
135     double f_min = f.eval (p_min);
136     debug_info ("First random point val", f_min);
137
138     const unsigned size = mpi::size ();
139     const unsigned portion = n_trials / size;
```

```

141 const unsigned my_trials = rank < n_trials % size ? portion + 1 : portion;
142
143 for (unsigned n = 1; n < my_trials; ++n)
144 {
145     //generate random coords
146     for (std::size_t i = 0; i < inf_limits.size(); ++i)
147     {
148         const double rand_val = distribution(generator);
149         random_coords[i] = inf_limits[i] +
150             (sup_limits[i] - inf_limits[i]) * rand_val;
151     }
152
153     random_point = Point(random_coords);
154     const Point p_new = solve(random_point);
155     debug_info("Local optimum found: ", f.eval(p_new));
156
157     const double f_new = f.eval(p_new);
158     if (f_new < f_min)
159     {
160         p_min = p_new;
161         f_min = f_new;
162     }
163 }
164
165 std::pair<double, int> minimum(f_min, rank);
166 MPI_Allreduce(MPI_IN_PLACE, &minimum, 1, MPI_DOUBLE_INT,
167               MPI_MINLOC, MPI_COMM_WORLD);
168
169 std::vector<double> min_coords = p_min.get_coords();
170 MPI_Bcast(min_coords.data(), min_coords.size(), MPI_DOUBLE,
171            minimum.second, MPI_COMM_WORLD);
172
173 return Point(min_coords);
174 }
```

```

177 Point FunctionMinRn::solve_domain_decomposition(unsigned int n_intervals,
178                                                 unsigned int n_trials) const
179 {
180     std::vector<double> internal_steps;
181     // compute steps along each axis
182     for (std::size_t i = 0; i < inf_limits.size(); ++i)
183     {
184         internal_steps.push_back((sup_limits[i] - inf_limits[i]) / n_intervals);
185     }
186
187     debug_info("Temp minimum");
188     Point p_min = Point(inf_limits);
189     double f_min = f.eval(p_min);
190
191     if (debug)
192         p_min.print();
193 }
```

```

194 const unsigned int n_subspaces = pow (n_intervals, inf_limits.size ());
195 debug_info ("N subspaces :", n_subspaces);
196
197 const unsigned rank = mpi::rank ();
198 const unsigned size = mpi::size ();
199 const unsigned portion = n_subspaces / size;
200 const unsigned remaining = n_subspaces % size;
201 unsigned first, last;
202 if (rank < remaining)
203 {
204     first = rank * (portion + 1);
205     last = first + portion + 1;
206 }
207 else
208 {
209     first = (rank - remaining) * portion + remaining * (portion + 1);
210     last = first + portion;
211 }
212 if (debug) std::cerr << "First " << first << ", last " << last << std::endl;
213
214 for (unsigned int n = first; n < last; ++n)
215 {
216     debug_info ("Compute next sub-interval");
217     const std::vector<double> cur_inf_limit =
218         next_inf_limit (n, n_intervals, internal_steps);
219
220     // compute cur_sup_limits
221     debug_info ("compute cur_sup_limits");
222     std::vector<double> cur_sup_limit;
223     for (std::size_t j = 0; j < cur_inf_limit.size (); ++j)
224         cur_sup_limit.push_back (cur_inf_limit[j] + internal_steps[j]);
225
226     debug_info ("Create local solver");
227     //That's inefficient!! Think how to improve this!
228     FunctionMinRn minf_int (f, tolerance, step, max_iterations,
229                             cur_inf_limit, cur_sup_limit);
230
231     if (debug)
232     {
233         std::cout << "Inf intervals" << std::endl;
234         for (double elem: cur_inf_limit)
235             std::cout << elem << " ";
236         std::cout << std::endl;
237
238         std::cout << "Sup intervals" << std::endl;
239         for (double elem: cur_sup_limit)
240             std::cout << elem << " ";
241         std::cout << std::endl;
242     }
243
244     debug_info ("Compute new minimum");
245     const Point p_iter = minf_int.solve_multistart (n_trials);

```

```

246     const double f_iter = f.eval (p_iter);
247
248     if (debug)
249         p_iter.print ();
250
251     if (f_iter < f_min)
252     {
253         p_min = p_iter;
254         f_min = f_iter;
255     }
256 }
257
258 std::pair<double, int> minimum (f_min, rank);
259 MPI_Allreduce (MPI_IN_PLACE, &minimum, 1, MPI_DOUBLE_INT,
260                 MPI_MINLOC, MPI_COMM_WORLD);
261
262 std::vector<double> new_coords = p_min.get_coords ();
263 MPI_Bcast (new_coords.data (), new_coords.size (), MPI_DOUBLE,
264             minimum.second, MPI_COMM_WORLD);
265
266 return Point (new_coords);
267 }
```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall main.cc FunctionMinRn.cc FunctionRn.cc MPI_helpers.cc
Monomial.cc Point.cc -o main
```

7.2 Challenges Solutions

Vector Norm

⁵The most important aim of the challenge is to compute in parallel, using MPI, the p norm, for $p \in [1, +\infty]$, of a vector in \mathbb{R}^n .

The full code of the program is presented below.

The functions `read_pn` and `read_vector` (see the file "get.hh"), both in `namespace get`, are intended to read data from standard input and to make them available on the different processes.

First of all, the function `read_pn` reads from standard input the values of p and n . In particular, only `process 0` reads the values and then a copy of this data is delivered to all the processes through a broadcast. The hardest part of this implementation was to understand how to assign an infinity value to p , in order to ask the infinity norm. Indeed, with our basic C++ input/output we are not able to read at the same time numbers (as we need for the finite norms) and input such as "inf", "+inf", "INFINITY", and so on. For this reason, since for the finite norms $p \geq 1$ and since the program stores this value in an `unsigned int` (and therefore it must be a positive number), the value $p = 0$ has been related to the infinity norm. It could be possible to use a `constexpr` variable, named "infinity", and assign to it value 0 in order to be able to call the function in a more intuitive way: `const double inf_norm = norm (infinity, v)`

The function `read_vector` reads from standard input the vector v . As usual, only `process 0` reads from standard input, but, in this case, it sends only a portion of the data to the other

⁵Implementation by Eva Capovin

processes, in order to perform the parallelization (in particular, the code is designed to work whether the number of elements in the vector is multiple of the number of available cores or not).

The implementation of the functions is written in the file "get.cc".

***** file get.hh *****

```

1 #ifndef GET_HH
2 #define GET_HH
3 #include <mpi.h>
4 #include <vector>
5 #include <string>
6 #include <iostream>
7 #include <limits>
8 #include <cmath>
9
10 namespace get
11 {
12     void read_pn (unsigned int & p, unsigned int & n);
13
14     std::vector<double> read_vector (unsigned int & n, MPI_Comm const & comm);
15 }
16
17 #endif

```

***** file get.cc *****

```

1 #include "get.hh"
2
3 namespace get
4 {
5     void read_pn (unsigned int & p, unsigned int & n)
6     {
7         int rank;
8         MPI_Comm_rank (MPI_COMM_WORLD, &rank);
9
10        if (rank == 0)
11        {
12            std::cout << "Please insert p\n" << std::endl;
13            std::cout << "Assigning 0 to p means that you want to "
14                           << "calculate infinity norm\n" << std::endl;
15            std::cin >> p;
16            if (p==0)
17                std::cout << "Infinity norm selected" << std::endl;
18            else
19                std::cout << "Finite norm selected" << std::endl;
20            std::cout << "Please insert n" << std::endl;
21            std::cin >> n;
22        }
23
24        //Deliver a copy of the data in p and n to all the processes
25        MPI_Bcast (&p, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
26        MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

```

```

27 }
28
29 std::vector<double> read_vector (unsigned int & n, MPI_Comm const & comm)
30 {
31     int rank, size;
32     MPI_Comm_rank (comm, &rank);
33     MPI_Comm_size (comm, &size);
34
35     //It is necessary to declare them in order to be able to calculate
36     //the norm of any vector, independently of its size
37     const unsigned local_n = n % size > 0 ? n / size + 1 : n / size;
38     const unsigned communication_n = local_n * size;
39
40     //Initialize a new vector of doubles whose size is local_n
41     std::vector<double> result (local_n);
42
43     if (rank == 0)
44     {
45         std::vector<double> input(n);
46         std::cout << "Enter vector" << "\n";
47         for (double & e : input)
48             std::cin >> e;
49         //Add zeros at the end of the vector input in order to make its length multiple
50         //of size without affecting the computation of the norms
51         input.resize (communication_n, 0);
52
53         //Send a portion of the data in input to all the processes storing it in result
54         MPI_Scatter (input.data (), local_n, MPI_DOUBLE, result.data (),
55                         local_n, MPI_DOUBLE, 0, comm);
56     }
57     else
58     {
59         MPI_Scatter (nullptr, local_n, MPI_DOUBLE, result.data (),
60                         local_n, MPI_DOUBLE, 0, comm);
61     }
62
63     return result;
64 }
65 }
```

The function `calculate_norm`, see the declaration in the file "calculate.hh" and the definition in the file "calculate.cc", computes the desired norm of the vector passed as an input and it returns the correct result on every process.

In computing the norm, we need to distinguish between the finite and the infinity case. The calculation of the local result on every process and the following assembly are performed through different functions (see the file "operation.hh") in order to take care of the definition of the norms.

***** file calculate.hh *****

```

1 #ifndef CALCULATE_HH
2 #define CALCULATE_HH
3 #include "operation.hh"
4 #include "mpi_assembling.hh"
```

```

5 #include <iostream>
6 #include <mpi.h>
7 #include <vector>
8 #include <cmath>
9
10 namespace calculate
11 {
12     double calculate_norm(const unsigned int & p, std::vector<double> & v);
13 }
14
15 #endif

```

***** file calculate.cc *****

```

1 #include "calculate.hh"
2
3 namespace calculate
4 {
5     double calculate_norm (const unsigned int & p, std::vector<double> & v)
6     {
7         //Calculating the norm we need to differentiate between finite and infinity norm
8
9         //Infinity norm
10        if(p==0)
11        {
12            const double local = operation::partial_max_module(v);
13            double norm = mpi_assembling::max (local);
14            return norm;
15        }
16        //Finite norm
17        else
18        {
19            const double local = operation::partial_sum (p, v);
20            double norm= mpi_assembling::sum_power(local, p);
21            return norm;
22        }
23    }
24 }

```

The function `partial_max_module`, which is obviously intended to be used in case $p = +\infty$, returns the maximum, in absolute value, between the elements of the vector passed as parameter. Since this function is called by `calculate::calculate_norm` after the scatter of the original vector `input`, the result computed by `partial_max_module` is only a local result. The function `partial_sum`, used in case $p < \infty$, returns the sum of the absolute values of the elements in vector `v`, raised to the power p . As before, the vector `v` passed as parameter is only a portion of the input read by `get::read_vector` at the beginning of the program.

***** file operation.hh *****

```

1 #ifndef OPERATION_HH
2 #define OPERATION_HH
3 #include <vector>
4 #include <cmath>
5

```

```

6  namespace operation
7  {
8      double partial_max_module(std::vector<double> & v);
9
10     double partial_sum (const unsigned int & p, std::vector<double> & v);
11 }
12
13 #endif

```

***** file operation.cc *****

```

1 #include "operation.hh"
2
3 namespace operation
4 {
5     double partial_max_module(std::vector<double> & v)
6     {
7         double n=0;
8         double m=0;
9         for (unsigned int i=0; i<v.size(); i++)
10         {
11             n=std::abs(v[i]);
12             if (n>m)
13                 m=n;
14         }
15         return m;
16     }
17
18     double partial_sum (const unsigned int & p, std::vector<double> & v)
19     {
20         double n=0.0;
21         double m=0.0;
22         for ( unsigned int i=0; i<v.size(); i++)
23         {
24             m=std::abs(v[i]);
25             n += pow(m,p);
26         }
27         return n;
28     }
29 }

```

Given the partial results, they must be assembled in order to compute the global norm and to make it available on all the processes (see the files "mpi_assembling.hh" and "mpi_assembling.cc").

Even in the assembling, the procedure is different whether $p = +\infty$ or $p < \infty$.

In the first case, the function we use is `max`. It receives as parameter the local result computed by `operation::partial_max_module` and it performs an `MPI_Allreduce`, applying `MPI_MAX` in order to store in `total`, in all the processes, the global maximum, which is exactly the infinity norm.

In the case $p < \infty$, we use the function `sum_power`. It receives as parameters both the local result and the index p of the norm we want to compute, it applies an `MPI_Allreduce` in order to get the global sum on all the processes and then it raises the result to the $1/p$ in

order to compute the p -norm.

***** file mpi_assembling.hh *****

```
1 #ifndef MPI_ASSEMBLING_HH
2 #define MPI_ASSEMBLING_HH
3
4 #include <iostream>
5 #include <cmath>
6
7 namespace mpi_assembling
8 {
9     double max (double local);
10
11    double sum_power (double local,const unsigned int & p);
12 }
13
14 #endif
```

***** file mpi_assembling.cc *****

```
1 #include <mpi.h>
2
3 #include "mpi_assembling.hh"
4
5 namespace mpi_assembling
6 {
7     double max (double local)
8     {
9         double total(0.);
10        //Store the result -apply MPI_MAX to the portions of data in local- on all
11        //the processes
12        MPI_Allreduce (&local, &total, 1, MPI_DOUBLE, MPI_MAX,
13                      MPI_COMM_WORLD);
14        return total;
15    }
16
17    double sum_power (double local, const unsigned int & p)
18    {
19        int rank;
20        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
21
22        double total (0.);
23        double a=(1.0)/p;
24        //Store the result -apply MPI_SUM to the portions of data in local- on all
25        //the processes
26        MPI_Allreduce (&local, &total, 1, MPI_DOUBLE, MPI_SUM,
27                      MPI_COMM_WORLD);
28
29        double totalfinal = pow(total, a);
30    }
}
```

```

31 }
32 }

***** file main.cc *****
1 #include <iostream>
2 #include <mpi.h>
3 #include <vector>
4 #include "get.hh"
5 #include "calculate.hh"
6 using namespace std;

7
8 int main (int argc, char *argv[])
9 {
10    MPI_Init (&argc, &argv);
11
12    //Initialize size and rank
13    int rank, size;
14    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
15    MPI_Comm_size (MPI_COMM_WORLD, &size);

16
17    unsigned int p;
18    unsigned int n;

19
20    //Read from standard input the values of p and n
21    //Moreover it delivers a copy of the data in p and n to all the processes
22    //through broadcast
23    get::read_pn (p, n);

24
25    //Read from standard input the vector v
26    //Moreover it sends a portion of the data to all the processes
27    //through scatter
28    vector<double> v = get::read_vector(n, MPI_COMM_WORLD);

29
30    //Calculate the p norm of the n-dim. vector v
31    double norm = calculate::calculate_norm(p,v);

32
33    //Print the result
34    if (rank==0)
35    {
36        if (p==0)
37            cout<<"inf";
38        else
39            cout<<p;
40        cout<<"-norm="<<norm<<endl;
41    }

42
43    MPI_Finalize ();
44
45    return 0;
46 }

```

Note: the program can be compiled through the following command:

```
mpicxx -std=c++11 -Wall main.cc calculate.cc get.cc mpi_assembling.cc operation.cc -o main
```

7.3 Frequently Asked Questions

7.3.1 General questions

1. If I define a variable and then I write `MPI_Init`, is that variable already available in all ranks?

ANSWER: `MPI_Init` starts all the MPI runtime environment to support message passing and this is not related to variables availability into processes. When you declare variables in a block, they are all allocated on the Stack and they are de-allocated when the block gets out of scope. This rule applies to all ranks independently on where you write `MPI_Init`. All processes are born when you run `mpiexec` at the command line even before you reach `MPI_Init` in your code.

2. When you do not specify the rank, which process performs the relative operations?

ANSWER: Any process which has access to the variables involved in the operation will do it if the rank is not specified. Therefore, if you write

```
1 // ... code
2 // operation 1
3 if (rank == 0)
4 {
5     // operation 2
6 }
7 else
8 {
9     // operation 3
10 }
11 // ... code
```

"operation 1", which is written outside of any condition, is performed by every rank; "operation 2" is done only by process 0 and "operation 3" is performed by every core with `rank > 0`.

3. When I define only a parallel function (not an entire program), where do I have to call `MPI_Init` and `MPI_Finalize`?

ANSWER: `MPI_Init` and `MPI_Finalize` are supposed to be called only once in the `main` function; when you define another function that will work in parallel, you just have to define properly `MPI_Comm_size(MPI_COMM_WORLD, &zsize)` and `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` if the process should know the number of available cores and its own rank (see for example Exercise 12 in Section 7.1, where both a parallel function and the `main` function are provided, respectively in files "Clustering.cc" and "KMeans.cc").

4. Supposing that I have to compute on every process two boundary values, `local_a` and `local_b`, so that the starting point for every process is the ending point of the previous one, can I communicate the values to the adjacent processes like this:

```
1 // ... code
2 const unsigned local_n = rank < remainder ? long_n : short_n;
```

```

4     double local_a = a; // process 0 never reassigned local_a, others do
5     if (rank > 0) // process 0 skips this
6     {
7         MPI_Recv(&local_a, 1, MPI_DOUBLE, rank - 1, 0,
8                 MPI_COMM_WORLD, MPI_STATUS_IGNORE)
9     }
10    double local_b = local_a + h * local_n;
11    if (rank < size - 1) // last process skips this
12    {
13        MPI_Send(&local_b, 1, MPI_DOUBLE, rank + 1, 0,
14                MPI_COMM_WORLD)
15    }
16    // ... code

```

ANSWER: If you write a code like this, process 0 computes its value of local_b, then it sends it to process 1. process_1 waits until it receives from process 0 a value that it stores in local_a, then it computes local_b and send it to process 2 and so on. This is correct in the sense that, at the end, every process will have the proper values stored in local_a and local_b. However, this is not a good parallelization scheme: since every process has to wait for the result of the previous one before starting its computation, you lose any advantage you would get by performing the computations in parallel. A better way to go is to make available to every process the starting point (in this example, the value of a) and then to let each one compute its end point without relying on the results of the other processes (see, for example, what is done in Exercise 13 in Section 7.1 to compute the domain decomposition).

7.3.2 Broadcast

1. Which is the difference between

```

1     // ... code
2     unsigned N = 0;
3     if (rank == 0)
4     {
5         std::cin >> N;
6         MPI_Bcast (&N, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
7     }
8     else
9         MPI_Bcast (&N, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
10    // ... code

```

and

```

1     // ... code
2     unsigned N = 0;
3     if (rank == 0)
4         std::cin >> N;
5     MPI_Bcast (&N, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
6     // ... code

```

ANSWER: the two codes are functionally equivalent. Any collective routine (broadcast, scatter, gather) must be written for every process in the communicator. This means that you can either call them once outside from every conditional that depends on the rank (as happens in the second example) or you must call them twice: once for the rank that is sending data and the other for those that are intended to receive.

- When I broadcast the data in a vector from process 0 to all the other processes, do I need to resize the vector?

ANSWER: When you write something as

```

1 // ... code
2 unsigned n = 0;
3 std::vector<int> v1;
4 if (rank == 0)
5 {
6     v1 = {1,2,3,4};    //or any more interesting way to read data
7     n = v1.size();
8 }
9 MPI_Bcast (&n, 1, MPI_UNSIGNED,0,MPI_COMM_WORLD);

```

only process 0 knows not only the proper values of the data in `v1`, but also the new size of the vector: in all the other processes `v1` is still empty. Therefore, if you write only

```
MPI_Bcast (&v1[0], n, MPI_INT,0,MPI_COMM_WORLD);
```

the program will fail: `MPI_Bcast` does not see the vector, it only receives a pointer and a counter and it assumes to have enough memory both to read from the send buffer and to write in the receive buffer. In this case, the send buffer stores (and it tries to communicate) `n` elements, but the receiver is an empty vector. In order to make things work properly, you must write

```
v1.resize(n);
```

before the broadcast (and, of course, after having communicated the value of `n`), so that `v1` now stores enough memory to receive all the data from process 0.

7.3.3 Scatter and Gather

- If I write a code like this, supposing that the value of `n` is already known in all the processes and that I only need to read and scatter the input:

```

1 // ... code
2 unsigned local_n = n%size > 0 ? n/size : n/size + 1;
3 unsigned comm_n = local_n * size;
4 std::vector<int> local_v(local_n);
5 if (rank == 0)
6 {
7     std::vector<int> input(n);
8     for (int & elem : input)
9         std::cin >> elem;
10    input.resize(comm_n,0);
11    MPI_Scatter(input.data(),local_n,MPI_INT,local_v.data(),local_n,
12                MPI_INT,0,MPI_COMM_WORLD);
13 }
14 else
15     MPI_Scatter(nullptr,local_n,MPI_INT,local_v.data(),local_n,MPI_INT,0,
16                 MPI_COMM_WORLD);
17 // ... code

```

why do I need the second `MPI_Scatter` and why do I fix the send buffer to `nullptr`?

ANSWER: As it is reported in the answer of Question 1 in Section 7.3.2, if you write

any collective routine within an **if** condition that depends on the rank, you have to write the corresponding routine in the **else** part. In particular, in case of an MPI_Scatter, since in the **else** you are only receiving data, you can set the send buffer to the `nullptr`. The possible alternative, i.e. to write only one

```
MPI_Scatter(input.data(),local_n,MPI_INT,local_v.data(),local_n,MPI_INT,  
           0,MPI_COMM_WORLD);
```

outside of any condition that depends on the rank, is not admissible in this case because the vector `input` is defined only in the scope of `if(rank==0)` (if you want to write only one MPI_Scatter, you have to put also the definition of the vector outside of any condition that depends on the rank, so that it is defined in every process; see for example Exercise 4 in Section 7.1).

- When we write an MPI_Scatter, does the number of elements communicated to the different process have to be the same, or is it possible to send a different number of values depending on the core that is receiving?

ANSWER: The number of elements passed to every process, i.e., if we write

```
MPI_Scatter(input.data(),local_n,MPI_INT,local_v.data(),local_n,MPI_INT,  
           0,MPI_COMM_WORLD);
```

the value of `local_n`, must be the same. This can lead to some problems when the number of elements that you have to send (referring again to this example, the size of the vector `input`) does not evenly divide the number of available cores. To get rid of this, you have two possibilities: either you resize `input`, adding an useless tail (the so called *padding*) in order to make its size be a multiple of the number of cores (see for example Exercise 4 or Exercise 5 in Section 7.1), or you use a cyclic partition (which means you do not use MPI_Scatter any more; see for example Exercise 2 in Section 7.1 or Exercise 3 in Section 7.1).

Chapter 8

Appendix

8.1 Advanced topics on pointers

Pointers to const variables

A pointer to a **const** variable is defined as:

```
const int ci = 1024;
const int * p1 = ci;
```

Example:

Find the problem in the following code:

```
1 const double pi = 3.14;
2 double *ptr = &pi;
3 const double *cptr = &pi;
4 *cptr = 42;
5 double dval = 7.5;
6 cptr = &dval;
```

The first issue stays in line 2. Indeed, **pi** is declared **const** and we cannot bind a non constant pointer to a constant variable.

Moreover, also line 4 is a syntax error. Indeed, **cptr** is defined in line 3 as a pointer to a **const double**, which means that the value pointed by **cptr** cannot be modified by the pointer itself.

Note that a pointer to a **const double**, as **cptr**, can always be initialized through a non constant **double** (while the opposite is not allowed, as we discussed before).

Moreover, note that **cptr** is not a constant pointer; thus, in line 6, we are allowed to change its value, making it point to another variable, namely **dval**. If we want a **const** pointer, i.e. a pointer which is forced to point forever to the variable through which it is initialized, we have to write

```
7 double * const c_ptr = &dval;
8 *c_ptr = 2.1;
9 double x = 3.2;
10 c_ptr = &x;
```

In this way, line 8 is admissible, because the value pointed by **c_ptr**, namely **dval**, is not **const** and then it can be modified. Line 10, on the other hand, gives a syntax error because **c_ptr** is a **const** pointer to **double** and so its value cannot be modified by assigning to it a new address.

Of course, we can also define a **const** pointer to a **const double**, writing

```
11 const double * const c_ptr_2 = &pi;
```

so that neither the value of the pointer nor the value of the pointed variable can be modified.

References to Pointers

Which one is possible: a pointer to a reference or a reference to a pointer?

1. A reference is not an object. Hence, we cannot have a pointer to a reference.
2. A pointer is an object, thus we can define a reference to a pointer.

```
1 int i = 42;
2 int *p; // p is a pointer to int
3 int *&r = p; // r is a reference to the pointer p
4 r = &i; // r refers to a pointer; assigning &i to r makes p point to i
5 std::cout << "i: " << i << " r: " << *r << " p: " << *p << std::endl;
6 *r = 0; // dereferencing r yields i, the object to which p points; changes i to 0
7 std::cout << "i: " << i << " r: " << *r << " p: " << *p << std::endl;
```

This is the output you get:

```
i: 42 r: 42 p: 42
i: 0 r: 0 p: 0
```

Exercise

What is the output of the following program?

```
1 #include <iostream>
2 using namespace std;
3 int main (void)
4 {
5 int a[2][4] = {3,6,9,12,15,18,21,24};
6 char *arr[] = {"C", "C++", "Java", "VBA"};
7 char *(*ptr)[4] = &arr;
8 cout << ++(*ptr)[3] << endl;
9 cout << *(a[1] + 3) << *(*(a+1) + 3) << ++(*ptr)[3] << endl;
10 return 0;
11 }
```

ANSWER: in line 5, **a** is defined as an array of array of **ints** (a multidimensional array, i.e. a matrix). In this kind of initialization, each array of **ints** is initialized with the elements of the given initialization list (in the example, `{3,6,9,12,15,18,21,24}`) until either all positions have been initialized or there are no more elements in the list (in this example, each row has dimension equal to 4, therefore the first row **a[0]** is initialized with the first four elements while the second row **a[1]** with the remaining ones).

arr, in turn, is defined in line 6 as an array of pointers to **chars**. To understand this definition, remember that you can define an array of **char** using the following syntax:

```
char char_array[] = "Hello World";
```

In this case, each character in the initialization string becomes a **char** in the array. In contrast,

```
char *arr[] = {"C", "C++", "Java", "VBA"};
```

defines an array of pointers to **char**. Each of these pointers points to the **char** at the first position of the array of **chars** created from the strings in the initialization list.¹

For instance,

¹**Note:** if you try to compile this code, the compiler will give you a warning. Indeed, each **char** array within **arr** is initialized through list initialization. The contents of the list, i.e. "C", "C++" and so on, are provided as constant strings, while the **char** arrays have not been declared as `const`. Since C compilers implement strings

```
cout << *(arr[3]) << endl;
```

would output V, which is the first **char** of the fourth string in the initialization list. Instead, if you miss the dereference operator, i.e. you write:

```
cout << arr[3] << endl;
```

the output would be all the characters starting from the given position up to the end of the **char** array (i.e. you would see as output VBA).

You can go from arrays to pointers and vice versa (see "Pointers and Arrays" in section 1.1.1). Therefore, when you write, in line 7,

```
char *(*ptr)[4] = &arr;
```

you get the following declaration: ***ptr** is a pointer to an array of size 4, which stores pointers to **char**. In particular, this pointer is initialized with the address of **arr** (in particular, the address of its first element). In this kind of definition, you must specify the size of the array, according to the size of the array used for the initialization.

Similarly, when you write, in line 8:

```
cout << ++(*ptr)[3] << endl;
```

you are performing the following operation: first, **ptr** is dereferenced, which gives the array of pointers to **char** we had before (i.e. **arr**). In particular, the pointer in the fourth position of this array (index 3, which is the pointer to "VBA") is accessed and then incremented. Therefore, the output will be BA.

Through the last instruction, in line 9, we access, first of all, the array **a**. The first part is:

```
*(a[1] + 3)
```

Here you access the second of the two arrays of **ints** stored in **a** (given by **a[1]**) and, adding 3, you get a pointer to the fourth element of this array. Then you dereference this pointer, getting the value 24.

The second part is:

```
*(*(a+1) + 3)
```

The inner arithmetic operation, **a+1**, results in a pointer to the second position of the array of arrays of **ints** (i.e. a pointer to **a[1]**). Dereferencing this pointer, you get the corresponding array of **ints**, i.e. **a[1]**; the other operations are exactly the same you performed in the previous instruction, then you get again 24.

The third and last operation is:

```
++(*ptr)[3]
```

This instruction is called for the second time, then also the resulting pointer is incremented a second time; this is the reason why the output now is A instead of BA.

The overall output of the program is:

```
BA  
2424A
```

in a different way with respect to C++, the C++ compiler keeps compatibility by just raising a warning and not an error. The same would not apply to other types:

```
const int i = 0;  
int *j = &i;
```

leads to an error message because the conversion from **const int *** to **int *** is invalid.

8.2 Templates and STL

Note: since templates are not part of the syllabus of the course, the Exercises reported in this Section should be considered as advanced readings.

Exercise 1 Dataframe (*exam 07/07/2017*)

A *DataFrame* is a 2-dimensional labeled data structure with columns of the same type. You can think of it like a spreadsheet, a table, or a Python `NamedTuple` (with the constraint to have column data of the same type!). The nice property of a *DataFrame* is that you can access its columns by name and apply to every column functions like `mean`, `max`, etc. Figure 8.1 shows, as an example, a *DataFrame* including two columns storing temperatures and humidity for a weather forecast application.

You have to provide a class template definition for `DataFrames`. It does not need to be implemented in extreme generality, rather you can consider as specialization target *arithmetic* types. In particular provide the implementation of the following methods:

1. A constructor that receives one single `string` as parameter. Using spaces as word separators, it will initialize the column names with the words contained in the argument. For example, the `DataFrame` in Figure 8.1 can be obtained passing "`temperatures` `humidity`".
2. A `set_column` method that, given a `vector` of values of type `T` and a column name, replaces the entire column content.
3. The method `set_element_at` that, given a column name, an index i , and a value of type `T`, updates the i -th value of the column.
4. `get_mean()`, which returns the mean of a given column.
5. `select_equal` that, given a column name and a value, returns a new `DataFrame` including only the set of rows for which the column equals the value. For instance, `select_equal ("temperatures", 31.4)` called on the `DataFrame` in Figure 8.1 would yield a new one with both columns, but only the second row.

Take particular care to error conditions, e.g., access to a wrong column or element index out of range. Moreover, check that all the columns have the same number of rows (which will be known only when creating the first column). For the constructor implementation you can rely on the `split()` function:

```
vector<string> split(const string & s, char d)
```

which returns the names of columns in `s` separated by the delimiter `d`. In other words, you are not required to implement `split()` yourselves.

Discuss how you organize your template in files. Finally, discuss the worst case complexity of the setter methods.

Exercise 1 - Solution

First of all, remember that the full definition of a **template class** must be provided in the header file. We cannot write the declaration in the header file and the definition in a different source file as we usually do.

For this exercise, two different solutions are provided: the first one uses only `std::vectors` as data structures, while the second one, which is the one required during the exam, relies on `std::maps`. Moreover, the first solution is written without taking care of possible errors which

weather_conditions_dataframe	
temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7

Figure 8.1: A weather application DataFrame.

may occur if, for example, we try to access a column which is not stored in the dataframe.

***** First solution:

As it is specified above, this solution uses two `std::vector`s in order to store the dataframe. In particular, a `std::vector<std::string>` is used to store the names of the columns, while every column is represented through a `std::vector<T>`, so that the whole dataframe is stored as a `std::vector<std::vector<T>>`.

Notice that all the types used for elements and containers are hidden in a couple of user-defined names (see lines 12 to 17). This is useful because it allows to easily change types and containers (provided that the methods are compatible) without having to rewrite the whole code.

The class `DataFrame` stores the vector of keys `df_keys`, the vector of columns `df_values` and a `bool` which is `true` if we have already added the first column to the dataframe (see line 22). It implements two `private` methods:

- `split` (line 25), which receives a `std::string` and a `char` and it returns a `std::vector<std::string>` built by splitting the string received as parameter using the given `char` as delimiter.
- `look_up` (line 28), which receives a key as parameter and returns the index in `df_values` which corresponds to the column addressed by the given key (or `df_values.size()` if the key is not stored in the dataframe).

Moreover, it implements two constructors: the one which receives a `std::string`, that was required in the text, and another one which receives a vector of keys and which is actually used to initialize the dataframe (see the declarations in lines 31 and 32, respectively).

In addition, there are seven `public` methods. Other than the ones required by the exercise, there is a method which is used to print out the dataframe and there are two getters, which return, respectively, a column or an element in a specific position (see lines 34, 36 and 38 for the declarations).

```

1 #ifndef DATAFRAME_HH
2 #define DATAFRAME_HH
3
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 template <typename T>
9 class DataFrame
10 {
11 public:
12     typedef std::vector<std::string> key_container;
13     typedef std::vector<T> mapped_container;

```

```

14
15     typedef key_container::value_type key_type;
16     typedef typename mapped_container::value_type mapped_type;
17     typedef typename std::vector<mapped_container>::size_type size_type;
18
19 private:
20     key_container df_keys;
21     std::vector<mapped_container> df_values;
22     bool added_first_column;
23
24     // return the names of columns in s separated by delim
25     key_container split (const key_type & s, char delim) const;
26
27     // return the index of column key, or df_keys.size () if not found
28     size_type look_up (const key_type & key) const;
29
30 public:
31     explicit DataFrame (const std::string & c_names);
32     explicit DataFrame (const key_container & names);
33
34     void print (void) const;
35
36     mapped_container get_column (const key_type & column_name) const;
37
38     mapped_type const & get_element_at (const key_type & column_name,
39                                         size_type index) const;
40     void set_element_at (const key_type & column_name, size_type index,
41                          const mapped_type & value);
42
43     T get_mean (const key_type & column_name) const;
44
45     // add a new column with data
46     void set_column (const key_type & column_name,
47                      const mapped_container & column_data);
48
49     // return a copy of the DataFrame with rows i such that "c_name[i] == value"
50     DataFrame select_equal (const key_type & c_name,
51                           const mapped_type & value) const;
52 };
53
54 template <typename T>
55 DataFrame<T>::DataFrame (const std::string & c_names)
56   : DataFrame (split (c_names, ' '))
57 {}
58
59 template <typename T>
60 DataFrame<T>::DataFrame (const key_container & names)
61   : df_keys (names), df_values (df_keys.size ()),
62     added_first_column (false)
63 {}
64
65 template <typename T>

```

```

66 typename DataFrame<T>::size_type
67 DataFrame<T>::look_up (const key_type & key) const
68 {
69     size_type index = 0;
70     while (index < df_keys.size () and df_keys[index] != key) ++index;
71     return index;
72 }
73
74 template <typename T>
75 typename DataFrame<T>::mapped_container
76 DataFrame<T>::get_column (const key_type & column_name) const
77 {
78     const size_type index = look_up (column_name);
79     return df_values[index];
80 }
81
82 template <typename T>
83 T DataFrame<T>::get_mean (const key_type & column_name) const
84 {
85     const size_type index = look_up (column_name);
86     mapped_container const & column = df_values[index];
87     T sum = T();
88     for (const mapped_type & v : column)
89         sum += v;
90     return sum / column.size ();
91 }
92
93 template <typename T>
94 void DataFrame<T>::print (void) const
95 {
96     for (size_type i = 0; i < df_keys.size (); ++i)
97     {
98         std::cout << df_keys[i] << " :: ";
99
100        for (const mapped_type & v : df_values[i])
101            std::cout << v << " ";
102
103        std::cout << std::endl;
104    }
105 }
106
107 template <typename T>
108 typename DataFrame<T>::key_container
109 DataFrame<T>::split (const std::string & s, char delim) const
110 {
111     std::string word;
112     key_container keys;
113     std::istringstream columns (s);
114     while (std::getline (columns, word, delim)) keys.push_back (word);
115     return keys;
116 }
117

```

```

118 template <typename T>
119 typename DataFrame<T>::mapped_type const &
120 DataFrame<T>::get_element_at (const key_type & column_name,
121                               size_type index) const
122 {
123     const size_type column_index = look_up (column_name);
124     return df_values[column_index][index];
125 }
126
127 template <typename T>
128 void DataFrame<T>::set_element_at (const key_type & column_name,
129                                   size_type index, const mapped_type & value)
130 {
131     const size_type column_index = look_up (column_name);
132     df_values[column_index][index] = value;
133 }
134
135 template <typename T>
136 void DataFrame<T>::set_column (const key_type & column_name,
137                                const mapped_container & column_data)
138 {
139     const size_type index = look_up (column_name);
140
141     if (added_first_column)
142     {
143         mapped_container & column = df_values[index];
144
145         for (size_type i = 0; i < column_data.size (); ++i)
146             column[i] = column_data[i];
147     }
148     else
149     {
150         added_first_column = true;
151         df_values[index] = column_data;
152
153         for (mapped_container & column : df_values)
154             column.resize (column_data.size ());
155     }
156 }
157
158 template <typename T>
159 DataFrame<T> DataFrame<T>::select_equal (const key_type & c_name,
160                                            const mapped_type & value) const
161 {
162     DataFrame<T> result (df_keys);
163
164     const size_type index = look_up (c_name);
165     const mapped_container & column = df_values[index];
166
167     std::vector<size_type> indices;
168
// select rows indices satisfying the selection criterion

```

```

170 for (size_type j = 0; j < column.size (); ++j)
171     if (column[j] == value)
172         indices.push_back (j);
173
174 for (size_type i = 0; i < df_keys.size (); ++i)
175 {
176     mapped_container values;
177     key_type const & current_name = df_keys[i];
178     mapped_container const & current_column = df_values[i];
179
180     for (size_type j : indices)
181         values.push_back (current_column[j]);
182
183     result.set_column (current_name, values);
184 }
185
186 return result;
187 }
188
189 #endif // DATAFRAME_HH

```

The first constructor (see lines 54 to 57), delegates the second one to initialize all the members of the class, passing to it the vector of keys returned by the method `split`. The second constructor, in turn (see lines 59 to 63), initializes the vector of keys through `names`, it sets the `bool added_first_column` to `false` and it initializes `df_values` as an empty vector with the size of `df_keys` (the number of columns we will have in the dataframe is equal to the number of keys).

The `private` method `look_up` (see lines 65 to 72) receives a key as parameter. It loops over the keys stored in `df_keys` and it returns the index which corresponds to the given one. This index will be used when we have to access a specific column in the dataframe because it links the name of a specific key to the position of the corresponding column (see for example the method `get_column`).

Notice that, since `size_type` is a type defined within the class, when we want to use it as the return type of a method we have to use the scope operator, i.e. to specify `Dataframe<T>::size_type` (see line 66). Moreover, since `Dataframe<T>` is a `template` class, we have to use the keyword `typename` in order to declare that the dependent name that we are looking for (i.e., in this case, `size_type`) is a type.

The method `get_column` (see lines 74 to 80) calls `look_up` in order to get the index corresponding to the given key and it returns the column which is stored in `df_values` at that index. Notice that the method does not check whether the given key is actually stored in the vector of keys. If this is not the case, the method `look_up` returns `df_keys.size()` and therefore line 79 turns out to have undefined behaviour, because it tries to access an element which is not stored in the vector.

The method `get_mean` (lines 82 to 90) computes and returns the average of the values in a given column. Notice that, in line 86, we extract from the dataframe the column whose average we want to compute. The type we use is a constant reference to `mapped_container`; by doing this, we avoid to create an useless copy of the given column (which is always a good practice in order to save memory, since we do not know how many elements it contains). The `const` qualifier is needed because we are in a constant method and therefore operations which could possibly lead to modifications in the members of the class are not allowed. If, instead of using a reference, we write, in line 86:

```
mapped_container column = df_values[index];
```

i.e. we create a copy of the required column, we could drop the `const` qualifier without getting compiler errors, because the fact that we create a copy prevents the dataframe to be modified by following operations. However, since even the new vector `column` is used only to read elements (and therefore it is not modified), it can be better to write:

```
const mapped_container column = df_values[index];
```

The methods `get_element_at` and `set_element_at` (see lines 118 to 125 and 127 to 133, respectively) simply calls `look_up` in order to get the index of the required column and then they rely on the `operator[]` of `std::vectors` in order to access the element at the proper index. The method `set_column` (see lines 135 to 156) acts in different ways when the first column has already been added to the dataframe and when it does not. In particular, in the first case (see lines 141 to 147) it extracts from the dataframe the required column and it loops over all the elements in order to set the values specified by `column_data`. In the second case, in turn, it sets the column which corresponds to the given key to be equal to `column_data` and then it resizes all the columns of the dataframe. Indeed, we want all the columns of the dataframe to be made by the same number of rows; therefore, once we know the size of the first column, we can prepare all the table, so that we avoid possible reallocations of the data structure when we add all the other columns.

Finally, the method `select_equal` (lines 158 to 187) returns a new `DataFrame` whose elements in column `c_name` are equal to `value`. First of all, a new `DataFrame` is initialized, passing as parameter `df_keys` because it is intended to have exactly the same keys of the former one. In lines 164 to 171, the method extracts the column whose name is given by `c_name` and it loops over this column in order to select the indices of the elements whose value is equal to `value`. Once we have this vector of indices (lines 174 to 184), we loop over all the elements of `df_keys`. For any key, we extract the corresponding column of `df_values` (line 178) and we initialize a vector (named `values`) with the elements of `current_column` whose index is stored in `indices`. Finally, we set `values` as column of the dataframe `result` at the position specified by the current key.

Notice that selecting the indices of the elements that are stored in the column `c_name` and whose value is `value` allows us to reduce the number of operations we perform, because in lines 179 and 180 we do not need to loop over all the elements of `current_column`, but we can directly select the elements we need.

Complexity: in the worst case, `get_column` and `set_column` have complexity $O(N + M)$, where N is the number of columns and M is the number of rows. The get and set of a single element have complexity $O(N)$.

In both cases, $O(N)$ comes from the `look_up` method and therefore it cannot be dropped even in the average case (unless the key we are looking for is stored at the very beginning of the dataframe).

***** Second solution:

As previously described, the second solution uses as data structure for the dataframe a `std::unordered_map`, which has `std::strings` as keys (the names of the columns) and `std::vector<T>` as values (the columns themselves).

In this case, the provided solution takes also care of the possible errors we might have if, for example, we try to access an element which is not stored in the dataframe.

The class `DataFrame` stores the map `df_data`, the `bool first_column_added` which is `true` if we have already added the first column to the dataframe (see line 31) and `n_rows` which stores the number of rows of the dataframe.

It implements three `private` methods:

- `check_column_name` (line 26), which receives a string as parameter and it returns `true` if that string corresponds to the name of one column of the dataframe.

- `split` (line 29), which behaves exactly as it is described in the first solution.
- `set_column_data` (line 35 and 36), which receives the name of a column and the column itself and inserts it in the dataframe checking if it has the correct number of rows.

Moreover, the class implements two constructors (the one which receives a `std::string`, that was required in the text, and the one with no parameters) and eight `public` methods. Other than the ones required by the text, we have the two getters (see lines 50 and 52), which return, respectively, a column or an element in a specific position, and a method to print the full dataframe.

Notice that there are two different `public` methods, `set_column` (lines 47 and 48) and `add_column` (lines 60 and 61): the first one is intended to be used when we want to modify the values of an existing column, while the second when we want to add a new column to the dataframe.

```

1 #ifndef DATAFRAME_H
2 #define DATAFRAME_H
3
4 #include <iostream>
5 #include <limits>
6 #include <sstream>
7 #include <string>
8 #include <unordered_map>
9 #include <vector>
10
11 template<typename T>
12 class DataFrame
13 {
14 public:
15     typedef std::unordered_map<std::string, std::vector<T> > df_type;
16
17     typedef typename df_type::value_type value_type;
18     typedef typename df_type::key_type key_type;
19     typedef typename df_type::mapped_type mapped_type;
20     typedef typename mapped_type::size_type size_type;
21
22 private:
23     df_type df_data;
24
25     // returns true if column_name is valid (included in the constructor list)
26     bool check_column_name (const key_type & column_name) const;
27
28     // return the names of columns in s separated by delim
29     std::vector<key_type> split (const key_type & s, char delim) const;
30
31     bool first_column_added;
32     size_type n_rows;
33
34     // set a column data checking only right number of rows
35     void set_column_data (const key_type & column_name,
36                           const mapped_type & column_data);
37
38 public:
```

```

39  DataFrame (const key_type & c_names);
40
41  DataFrame();
42
43  DataFrame & operator= (const DataFrame &) = default;
44
45  void print (void) const;
46
47  void set_column (const key_type & column_name,
48                  const mapped_type & column_data);
49
50  mapped_type get_column (const key_type & column_name) const;
51
52  T get_element_at (const key_type & column_name, size_type index) const;
53
54  void set_element_at (const key_type & column_name, size_type index,
55                      const T & value);
56
57  T get_mean (const key_type & column_name) const;
58
59 // add a new column with data
60 void add_column (const key_type & column_name,
61                  const mapped_type & column_data);
62
63 // return a copy of the DataFrame with rows such that the c_name = value
64 DataFrame<T> select_equal (const key_type & c_name, const T & value) const;
65
66 }; /** end of class declaration ***/
67
68 template<typename T>
69 DataFrame<T>::DataFrame (const key_type & c_names)
70   : DataFrame ()
71 {
72   std::vector<key_type> columns_names = split (c_names, ' ');
73
74   for (const key_type & name : columns_names)
75     df_data[name];
76 }
77
78 template<typename T>
79 DataFrame<T>::DataFrame (void)
80   : first_column_added (false), n_rows (0) {}
81
82 template<typename T>
83 bool DataFrame<T>::check_column_name (const key_type & column_name) const
84 {
85   return df_data.find (column_name) != df_data.end();
86 }
87
88 template<typename T>
89 void DataFrame<T>::set_column (const key_type & column_name,
90                               const mapped_type & column_data)

```

```

91 {
92     if (check_column_name (column_name))
93         set_column_data (column_name, column_data);
94     else
95         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
96 }
97
98 template<typename T>
99 typename DataFrame<T>::mapped_type
100 DataFrame<T>::get_column (const key_type & column_name) const
101 {
102     if (check_column_name (column_name))
103         return df_data.at (column_name);
104     else
105     {
106         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
107         return std::vector<T> ();
108     }
109 }
110
111 template<typename T>
112 T DataFrame<T>::get_mean (const key_type & column_name) const
113 {
114     if (check_column_name (column_name))
115     {
116         T sum = T();
117
118         for (T d : df_data.at (column_name))
119             sum += d;
120
121         return sum / n_rows;
122     }
123     else
124     {
125         std::cerr << "Error, " << column_name << " is unknown" << std::endl;
126         return std::numeric_limits<T>::quiet_NaN();
127     }
128 }
129
130 template<typename T>
131 void DataFrame<T>::print (void) const
132 {
133     for (const value_type & element : df_data)
134     {
135         std::cout << element.first << " :: ";
136
137         for (const T & d : element.second)
138             std::cout << d << " ";
139
140         std::cout << std::endl;
141     }
142 }

```

```

143
144 template<typename T>
145 std::vector<typename DataFrame<T>::key_type>
146 DataFrame<T>::split (const key_type & s, char delim) const
147 {
148     key_type word;
149     std::vector<key_type> v;
150     std::istringstream columns (s);
151
152     while (std::getline (columns, word, delim)) v.push_back (word);
153
154     return v;
155 }
156
157 template<typename T>
158 T DataFrame<T>::get_element_at (const key_type & column_name,
159                                 size_type index) const
160 {
161     if (check_column_name (column_name))
162         if (index < n_rows)
163             return df_data.at (column_name)[index];
164         else
165             {
166                 std::cerr << "Error, index out of bound" << std::endl;
167                 return std::numeric_limits<T>::quiet_NaN();
168             }
169         else
170             {
171                 std::cerr << "Error, " << column_name << " is unknown" << std::endl;
172                 return std::numeric_limits<T>::quiet_NaN();
173             }
174 }
175
176 template<typename T>
177 void DataFrame<T>::set_element_at (const key_type & column_name,
178                                     size_type index, const T & value)
179 {
180     if (check_column_name (column_name))
181         if (index < n_rows)
182             df_data[column_name][index] = value;
183         else
184             std::cerr << "Error, index out of bound" << std::endl;
185         else
186             std::cerr << "Error, " << column_name << " is unknown" << std::endl;
187 }
188
189 template<typename T>
190 void DataFrame<T>::add_column (const key_type & column_name,
191                               const mapped_type & column_data)
192 {
193     if (! check_column_name (column_name))
194         set_column_data (column_name, column_data);

```

```

195     else
196         std::cerr << "Error, " << column_name
197             << " is already included in the DataFrame"
198             << std::endl;
199 }
200
201 template<typename T>
202 void DataFrame<T>::set_column_data (const key_type & column_name,
203                                     const mapped_type & column_data)
204 {
205     if (! first_column_added)
206     {
207         n_rows = column_data.size();
208         first_column_added = true;
209         for (typename df_type::iterator it = df_data.begin();
210              it != df_data.end(); it++)
211         {
212             (it->second).resize(n_rows);
213         }
214         df_data[column_name] = column_data;
215     }
216     else
217     {
218         // for next assignments check the column has the same number of rows
219         if (n_rows == column_data.size())
220             df_data[column_name] = column_data;
221         else
222         {
223             std::cerr << "Error, " << column_name
224                 << " has a different number of rows"
225                 << std::endl;
226         }
227     }
228 }
229
230 template<typename T>
231 DataFrame<T> DataFrame<T>::select_equal (const key_type & c_name,
232                                         const T & value) const
233 {
234     if (! check_column_name (c_name))
235     {
236         std::cerr << "Error, " << c_name << " is unknown" << std::endl;
237         return DataFrame<T> ();
238     }
239
240     std::vector<size_type> indexes; // solution indexes
241     const mapped_type & relevant_column = df_data.at (c_name);
242
243     // select rows indexes satisfying the selection criterion
244     for (size_type i = 0; i < n_rows; ++i)
245         if (relevant_column[i] == value)
246             indexes.push_back (i);

```

```

247
248     DataFrame<T> result;
249
250     for (const value_type & element : df_data)
251     {
252         mapped_type v; //column values
253         v.reserve (indexes.size ());
254
255         // select proper column values
256         for (size_type idx : indexes)
257             v.push_back (element.second[idx]);
258
259         result.add_column (element.first, v);
260     }
261
262     return result;
263 }
264
265 #endif //DATAFRAME_H

```

The first constructor (see lines 68 to 76) calls the **private** method `split` in order to get the vector of column names, passing as parameters the string `c_names` and a space as delimiter. Then, it relies on the `operator[]` of `std::maps` in order to generate the empty columns of the new dataframe. In line 70, it delegates the constructor with no parameters (see the implementation in lines 78 to 80) in order to initialize `first_column_added` to `false` and the number of rows to 0.

The method `set_column` (see lines 88 to 96) calls the **private** method `set_column_data` if the string passed as parameter corresponds to the name of an existing column, otherwise it prints an error message and exits without modifying the dataframe.

The same **private** method is called by `add_column` (see lines 189 to 199), which, in turn, adds the new column when its name does not correspond to any column already stored in the dataframe.

The method `set_column_data` (see lines 201 to 228) has a different behaviour whether the first column has already been added or not. In particular, in case we are adding the first column (lines 205 to 215), it sets the value of `n_rows` equal to the size of the vector `column_data`, it loops over all the dataframe in order to resize the columns with the proper number of rows (see the corresponding method in the first solution) and, finally, it adds the given column to the dataframe. When, in turn, `first_column_added` is `true` (see lines 216 to 227), if the size of the new column is equal to `n_rows`, it is added to the dataframe, otherwise an error message is printed.

Notice that `set_column_data` can be called both by `set_column` and by `add_column` without having to modify the implementation. Indeed, it relies on the `operator[]` of `std::maps`, which can be used either to modify the value corresponding to an existing key or to add a new element if the given key was not stored in the map.

The method `get_column` (see lines 98 to 109) returns the `std::vector<T>` corresponding to the given key if it exists, an empty vector otherwise.

Notice that, both here and in the method `get_element_at` (see lines 157 to 174), which returns the element stored in the given position or `Nan` if it does not exist, we use the method `std::map::at` to access the given column. We cannot use the `operator[]`, as we do in line 163 with the vector selected by `df_data.at()`, because both `get_elem` and `get_element_at` are declared as `const` (see the discussion in Question 1 of Chapter 6, Section 6.2).

The method `get_mean` (see lines 111 to 128) computes and returns the mean of a given column (or `Nan` if the column is not in the dataframe).

Finally, the method `select_equal` (lines 230 to 263) returns a new `DataFrame` whose elements in column `c_name` are equal to `value`. First of all (see lines 240 to 246), it extracts the column whose name is given by `c_name` and it loops over this column in order to select the indices of the elements whose value is equal to `value`. Once we have this vector of indices, we instantiate a new `DataFrame`, we loop over all the elements of `this` and, for any column, we create a vector with the elements whose index is stored in `indexes`. Finally, we add this vector as a new column of the dataframe `result`.

Notice that having selected the indices of the elements that are stored in the column `c_name` and whose value is `value` allows us to reduce the number of operation we perform, because in lines 256 and 257 we do not need to loop over all the elements of the column `element.second`, but we can directly select the elements we need.

Complexity: in the worst case, `get_column` and `set_column` have complexity $O(N + M)$, where N is the number of columns and M is the number of rows. Indeed, in the very worst case all columns are stored in a single bucket (and the bucket is implemented at least as a `forward_list`) and therefore you pay $O(N)$ to access a given column. Then, when you have the column, you have to store or read all M elements and you pay $O(M)$. If you consider the average case, to access the column you pay $O(1)$, but then you still have to store or read M elements (and hence complexity is $O(M)$). The get and set of a single element have complexity $O(N)$. If the *average* case is considered, `get_column` and `set_column` have complexity $O(M)$, while the complexity for accessing single elements is $O(1)$.

This is due to the fact that to access an element in an `unordered_map` (both through the `operator[]` and through the method `at`) has complexity $O(1)$ in the average case, while it is linear in container size (i.e. it is $O(N)$) in the worst case.

Exercise 2 **HashMap** (*exam 03/02/2017*)

`HashMap` is a structure that can be implemented as a vector of lists. Each list is a bucket to keep elements with the same hash value. Elements are pairs of `<key, value>` (see Figure 8.2). When an element is inserted in `HashMap`, a hash function computes the hash value from the key and inserts the element in the relevant bucket.

Given the following definition:

```

1 #ifndef HASHMAP_HH
2 #define HASHMAP_HH
3
4 #include <list>
5 #include <iostream>
6 #include <vector>
7 #include <utility>
8
9 template <typename K, typename T>
10 class HashMap {
11
12     typedef std::vector<std::list<std::pair<K, T>>> table;
13
14     double max_load_factor; //max_load_factor
15     size_t bucket_count = 0; //vector length
16     size_t count = 0; //total inserted elements
17
18     std::hash<K> hash_code;
19
20     void

```

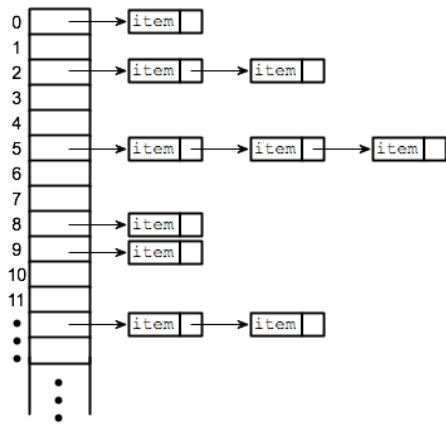


Figure 8.2: HashMap

```

21  resize (void);
22
23  table hashtable;
24
25 public:
26
27  HashMap():
28      max_load_factor(0.75), bucket_count(8),
29      count(0), hashtable(bucket_count) {}
30
31  void
32  add (const K & key, const T & value);
33
34  void
35  remove (const K & key);
36
37  void
38  dump_internal_structure() const;
39
40 };
```

Implement the two methods below and discuss their complexity:

1. `add (K key, T value)` computes the hash value of `key` and places `value` in the resulting bucket.
2. Hash values depend on the table size, thus they change for each entry when resizing the table. `resize()` should create a double sized vector when `max_load_factor` is exceeded, recompute the hash value of the old elements and reallocate them in the new container.

Exercise 2 - Solution

The class `HashMap<K,T>` (see the header file reported above) is a **template** class which depends on both the type of the keys and the one of the values. An `HashMap` is represented by a

`std::vector<std::list<std::pair<K,T>>>`, hidden in the user defined name `table` (see line 12 of the header file).

The class stores a **double**, called **max_load_factor**. The load factor represents the ratio between the number of elements in the container and the number of buckets in which these elements are stored; in particular, it gives information about the probability that two elements are stored in the same bucket. The **max_load_factor**, as the initial number of buckets, is fixed when the container is initialized; when it is exceeded, the number of buckets must be increased so that the probability of collisions is kept reasonably small.

bucket_count represents the number of buckets that are stored in the **HashMap**; it is fixed during the initialization and then it is increased any time the **max_load_factor** is exceeded, so that the ratio between the number of elements and **bucket_count** itself remains small.

Finally, **count** is the number of elements actually stored in the container, which is of type **table** (see line 24).

The class stores also an object of type **std::hash<K>**, named **hash_code**. This object takes an element of type **K** (a key) and it computes its hash value, i.e. a value of type **size_t** such that, if **k1** and **k2** are equal, **hash_code(k1)** and **hash_code(k2)** are equal, while the probability of the hash values to be equal is very small if **k1** and **k2** are different keys. Thanks to this property, hash values are used to compute the position that two elements should have in a container: if the elements are characterized by the same key, their hash values are equal and this prevents from creating multiple elements with the same key and allows to use this key to find the element in the container. On the other hand, if the elements have different keys, their hash values are different and therefore they will be stored in different buckets, which prevents from having one single bucket filled with a lot of elements.

- Resize:

```

38     dump_internal_structure() const;
39
40 }; /* end of class declaration */
41
42 template <typename K, typename T>
43 void
44 HashMap<K, T>::resize (void)
45 {
46     bucket_count *= 2;
47
48     table temp (bucket_count);
49     hashtable.swap (temp);
50
51     count = 0;
52
53     // iterate over old hash table and add each entry to a new table.
54     for (const auto & bucket: temp)
55     {
56         for (const auto & couple: bucket)
57         {

```

The method **resize**, reported above, doubles the value of **bucket_count** (i.e. the number of elements that can be stored in the vector), initializes a new **table** with this size and it uses the method **std::vector::swap** in order to insert in **temp** all the elements previously stored in **hashtable**. Then it sets to 0 the number of elements in **hashtable** (which, after the **swap**, is an empty container), loops over all the buckets in **temp** and over all the elements in each bucket and it calls the method **add** in order to insert the element in **hashtable**.

- Add:

```

59     }
60   }
61 }
62
63 template <typename K, typename T>
64 void
65 HashMap<K, T>::add (const K & key, const T & value)
66 {
67   const std::size_t idx = hash_code (key) % bucket_count;
68
69   // look if the key is already inserted
70   auto it = hashtable[idx].cbegin ();
71   while (it != hashtable[idx].cend () and it->first != key)
72     ++it;
73
74   // if you reach the end, you could not find the pair
75   if (it == hashtable[idx].cend ())
76   {
77     hashtable[idx].push_back ({key, value});
78     ++count;
79   }

```

The method `add` receives as parameters the key and the value of the element that must be added to the hash table. In line 63, it computes the index of the bucket in which the element should be added. In order to do this, it computes the modulo between the value given by `hash_code(key)` and the number of buckets in the container (remember that `hash_code(key)` returns the hash value of the key, i.e. a `size_t` which can be any integer number, and we have to relate it with the number of buckets in the container in order to get an admissible index).

Given the index, the method loops over the corresponding bucket in order to check whether the key passed as parameter is already stored in the hash table (lines 66 to 68). If this is not the case, a new pair `{key,value}` is added at the end of the list which represents the bucket and the number of inserted elements is incremented (lines 71 to 75).

Finally, the ratio between `count` and `bucket_count` is computed in order to check whether it is necessary to resize the hash table, incrementing the number of buckets.

If N is the number of elements stored in the hash table, the `resize` method is definitely $O(N)$, since it loops over all the stored items to copy them in the freshly created double-size data structure. On the other hand, the time complexity of `add` requires some more thought. Obviously, the worst case is $O(N)$, when it calls `resize` because of a high load factor. In every other case the time complexity is dominated by the search in the bucket: calling b the (maximum) list size, we have $O(b)$. The performance of the hash table depends on the quality of the hash function: if collisions remain rare, `add` is amortized constant in complexity, but in the worst case b can still become comparable to N .

Exercise 3 Sparse Matrix

Consider a sparse matrix class, in particular the simple LIL representation, which consists of a vector of lists, where every list represents a row. Since the matrix is sparse, all the zero-valued entries are not represented, whilst the nonzero entry a_{ij} appears in the $i - th$ list as the pair (j, a) (see Figure 8.3).

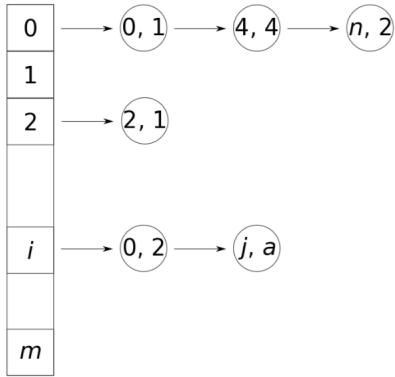


Figure 8.3: LIL representation of a sparse matrix.

The class provides some type names: `size_type` is an unsigned integral type, while `value_type` should be a **template**.

Provide an implementation of the following methods:

- `void set (size_type i, size_type j, value_type a);`
- `value_type get (size_type i, size_type j) const;`

State the complexity of the implemented methods for an $m \times n$ matrix. What would the complexity be if the outer container were a `std::list`? How could you change the data structure in order to improve the complexity of these methods?

Exercise 3 - Solution

The `lil_sparse` class is a **template** class which stores a `std::vector<std::list<std::pair<size_type, T>>>`,

where the **template** parameter `T` is hidden under the user-defined name `value_type`.

The class stores `m_rows` and `m_columns`, which are the number of rows and the number of columns of the matrix. The constructor of the class (see lines 35 and 36 for the declaration, 51 to 53 for the definition) receives as parameters the number of rows and columns of the matrix. Notice that both parameters have a default value, which means that the constructor can be called even without passing anything as parameter; in this case, since the default value for the number of rows is 1, a single row-vector will be created. The default value for the number of columns is 0 because it can be incremented dynamically by adding new elements. Notice that the implementation presented below is based on the fact that the vector of lists has fixed size (i.e. we have to decide *a priori* the number of rows the sparse matrix), while the size of the lists themselves can grow dinamically as soon as we decide to add a new element in a certain row (and therefore the number of elements per column is not fixed). This is due to the fact that to change the size of a vector, while keeping the elements in a certain order (which is something we need to preserve the order of the rows in the matrix), is an expensive operation, while it has lower complexity in case of lists. For what concerns the choice of different containers and the complexity of the methods, see the discussion at the end of the answer.

The class implements the **private** method `search`, both in the constant and in the non-constant version, that returns an iterator to the element stored in a specific position. If the element is not stored in the matrix, `search` returns an iterator to the following element (or `nonzero.end()` if we reached the end of the row).

Moreover, it implements two methods to read the number of rows and columns in the matrix and two methods to get or set an element in a specific position.

```

1 #ifndef __LIL_SPARSE__
2 #define __LIL_SPARSE__
3
4 #include <list>
5 #include <utility>
6 #include <vector>
7
8 namespace numeric
9 {
10    template <typename T>
11    class lil_sparse
12    {
13        public:
14            typedef T value_type;
15            typedef std::size_t size_type;
16
17        private:
18            typedef std::list<
19                std::pair<const std::size_t, value_type>> inner_list;
20            typedef typename inner_list::iterator inner_iterator;
21            typedef typename inner_list::const_iterator const_inner_iterator;
22            typedef std::vector<inner_list> data_structure;
23
24            data_structure nonzero;
25
26            size_type m_rows;
27            size_type m_columns;
28
29            inner_iterator
30            search (size_type, size_type);
31            const_inner_iterator
32            search (size_type, size_type) const;
33
34        public:
35            explicit lil_sparse (size_type rows = 1,
36                                size_type columns = 0);
37
38            size_type
39            rows (void) const {return m_rows;}
40            size_type
41            columns (void) const {return m_columns;}
42
43            void
44            set (size_type, size_type, value_type);
45
46            value_type
47            get (size_type, size_type) const;
48
49    }; /* end of class declaration */
50
51    template <typename T>
52    lil_sparse<T>::lil_sparse (size_type rows, size_type columns)

```

```

53     : nonzero (rows), m_rows (rows), m_columns (columns) {}
54
55     template <typename T>
56     typename lil_sparse<T>::inner_iterator
57     lil_sparse<T>::search (size_type i, size_type j)
58     {
59         inner_iterator it = nonzero[i].begin ();
60         while (it != nonzero[i].end () && it -> first < j) ++it;
61         return it;
62     }
63
64     template <typename T>
65     typename lil_sparse<T>::const_inner_iterator
66     lil_sparse<T>::search (size_type i, size_type j) const
67     {
68         const_inner_iterator it = nonzero[i].cbegin ();
69         while (it != nonzero[i].cend () && it -> first < j) ++it;
70         return it;
71     }
72
73     template <typename T>
74     void
75     lil_sparse<T>::set (size_type i, size_type j, value_type a)
76     {
77         inner_iterator it = search (i, j);
78         if (it != nonzero[i].end () && it -> first == j)
79         {
80             it -> second = a;
81         }
82         else
83         {
84             nonzero[i].insert (it, std::make_pair (j, a));
85         }
86     }
87
88     template <typename T>
89     typename lil_sparse<T>::value_type
90     lil_sparse<T>::get (size_type i, size_type j) const
91     {
92         // If you can't find an entry in nonzero, then it's zero.
93         value_type result = 0.;
94         const_inner_iterator it = search (i, j);
95         if (it != nonzero[i].cend () && it -> first == j)
96         {
97             result = it -> second;
98         }
99         return result;
100    }
101
102 }
103
104 #endif

```

The method `set` (see the implementation in lines 73 to 86) can be used either to modify the value of an element which is already stored in the matrix or to insert a new element in a given row and column. In particular, in the second case we rely on the method `std::list::insert` in order to insert in the list which corresponds to the given row a new pair with the index of the required column and the value of the element. Notice that `std::list::insert` receives as parameter not only the new pair, but also an iterator to the position in which the element must be inserted (the iterator returned by `search` in line 77).

The worst case complexity of the method `set` is $O(n)$, where n is the number of columns, which is the cost of the `search` method. Indeed, vectors allow random access, which means that, in `search`, we reach in $O(1)$ the required row and then we have to walk over the corresponding list until we find the element. On the other hand, both the assignment we perform in line 80 and the method `std::list::insert` have complexity $O(1)$ (in general, we have that the cost of `std::list::insert` is linear in the number of inserted elements).

Having as external container a `std::list` instead of a `std::vector` increases the overall complexity of the method. Indeed, lists do not support random access and, therefore, the complexity of `search` becomes $O(m + n)$, where m is the number of rows in the matrix.

In order to improve the performances of the program, we can consider to replace the given data structure by a

```
std::vector<std::map<std::size_t, T>>,
```

where we have as keys the indices of the rows and as values the same lists we were storing before in the vector. A similar data structure produces an overall complexity of the method `set` which is $O(\log n)$. Indeed, we can modify the method `search` to rely on `std::map::find`, which has complexity $O(\log n)$; moreover, the method `std::map::insert` we would use in line 84 has again complexity $O(\log n)$.

Exercise 4 Heap

A **Heap** is a data structure consisting of an almost complete binary tree satisfying the so-called *heap property*: for instance, in a *min-heap* every node hosts a value not less than the one stored in the parent node, meaning that the root of the tree holds the minimum. As you can see in figure Figure 8.4, where it is represented a *max-heap*, it is easy to store the implicit binary tree in a regular array, computing the parent, left and right child node positions algebraically.

Most operations on heaps boil down to moving elements up or down the tree in order to restore the heap property. Considering a *min-heap*, you can push a new element on the **Heap** appending it and then comparing to its parent: if the new element happens to be less than its parent, then they should be swapped and the procedure should be repeated, until either the new element is greater than its current parent or it reaches the root position. On the other hand, you can remove the root element by copying the last one in root position and moving it down the tree if one of its children is smaller.

Consider the **template** class `Heap<T, Compare>`, where `T` is the value type and `Compare` is a type providing a binary call `operator` that defines an ordering on `T`. Such class declares the following members:

- `std::vector<T> m_data;`
- `std::vector<T>::size_type m_size;`
- `Compare m_compare;`

Implement the two procedures that, given an index in the array, move the corresponding element up or down in the tree. You can assume that the class already provides an imple-

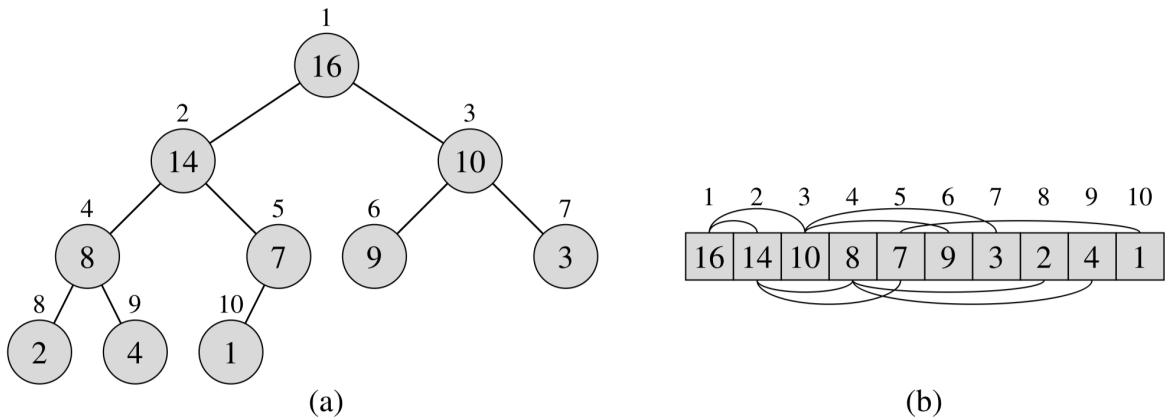


Figure 8.4: Max-heap.

mentation of the `parent()`, `left()` and `right()` methods, which return respectively the parent, left or right child index of the one passed as argument.

State the asymptotic complexity of the implemented methods, providing an informal motivation for your answer.

Exercise 4 - Solution

The `Heap` class is a **template** class depending on two parameters, namely the type of the elements stored in the tree and the comparison rule we want to use in order to build the tree. It implements three constructors: the first one (see lines 52 and 53) receives as parameter an object of type `Compare`, i.e. a function which will be used to compare two elements in the tree. The second one (see lines 55 to 57) receives two `iterators` and a `Compare` object, while the third (see lines 59 and 60) receives an `initializer_list` with the elements that must be stored in the tree and the function used for comparison.

Notice that the **template** parameter `Compare` and all the `Compare` objects in the constructors have a default value, which is fixed to `std::less<T>` (see lines 10 and 11). This means that, using, for example, the first constructor, we can instantiate an object of type `Heap` both writing:

```
comparison_rule mycomp;
Heap<double, comparison_rule> myheap_1(mycomp);
```

where `comparison_rule` is a user-defined type which represents a comparison rule (and `mycomp` is an object of that type), and writing:

```
Heap<double> myheap_2;
```

relying on the default value of the **template** parameter `Compare`.

As it is shown in Figure 8.4, the indices of the elements in the tree (and therefore in the array) start from 1, while indices of C++ `std::vectors` start from zero. Suppose, for example, we are in node 3, which stores the value 10. The index, in the array, of its parent is 1 and it is computed as $\lfloor \frac{idx}{2} \rfloor$ where `idx` is equal to the current index, i.e. 3. The indices of the left and the right children, in turn, are computed as $2 * idx$ and $2 * idx + 1$, respectively.

Once we have the indices in the array, we must compute the indices of the elements in the `std::vector<T>` which represents the data structure in the `Heap` class. Since the indices of C++ vectors start from 0, we get the parent as $\frac{idx+1}{2} - 1$, the left child as $2 * idx + 1$ and the right child as $2 * idx + 2$.

The methods `left`, `right` and `parent` (see lines 33 to 49) are implemented exactly in this way (notice that the keyword `inline` is not mandatory, since the methods defined in-class are automatically inlined).

```

1 #ifndef __HEAP__
2 #define __HEAP__
3
4 #include <functional>
5 #include <initializer_list>
6 #include <vector>
7
8 namespace ds
9 {
10     template <typename T,
11               typename Compare = std::less<T> >
12     class Heap
13     {
14         typedef std::vector<T> container_type;
15
16     public:
17         typedef T value_type;
18         typedef Compare value_compare;
19         typedef typename container_type::size_type size_type;
20
21     private:
22         container_type m_data;
23         value_compare m_compare;
24
25         void
26         sift_down (size_type i);
27         void
28         sift_up (size_type i);
29
30         void
31         build_Heap (void);
32
33         inline size_type
34         left (size_type i) const
35         {
36             return 2 * i + 1;
37         }
38
39         inline size_type
40         right (size_type i) const
41         {
42             return 2 * i + 2;
43         }
44
45         inline size_type
46         parent (size_type i) const
47         {
48             return (i + 1) / 2 - 1;
49         }
50
51     public:
52         explicit Heap (const value_compare & comp = value_compare ())

```

```

53   : m_compare (comp) {}
54
55 template <typename InputIterator>
56   Heap (InputIterator first, InputIterator last,
57         const value_compare & comp = value_compare ());
58
59   Heap (std::initializer_list<value_type> il,
60         const value_compare & comp = value_compare ());
61
62   inline size_type
63   size (void) const
64   {
65     return m_data.size ();
66   }
67
68   inline bool
69   empty (void) const
70   {
71     return m_data.empty ();
72   }
73
74   inline const value_type &
75   peek (void) const
76   {
77     return m_data.front ();
78   }
79
80   void
81   pop (void);
82
83   void
84   push (const value_type &);
85
86   void
87   replace (const value_type &);
88 };
89
90 template <typename T, typename Compare>
91   void
92   Heap<T, Compare>::sift_down (size_type i)
93   {
94     bool keep_sifting = true;
95
96     while (keep_sifting and i < m_data.size ())
97     {
98       keep_sifting = false;
99
100      const size_type l = left (i), r = right (i);
101      size_type best = i;
102
103      if (l < m_data.size () and m_compare (m_data[l], m_data[best]))
104      {

```

```

105     best = l;
106     keep_sifting = true;
107 }
108
109 if (r < m_data.size () and m_compare (m_data[r], m_data[best]))
110 {
111     best = r;
112     keep_sifting = true;
113 }
114
115 if (keep_sifting)
116 {
117     using std::swap;
118     swap (m_data[best], m_data[i]);
119     i = best;
120 }
121 }
122 }
123
124 template <typename T, typename Compare>
125 void
126 Heap<T, Compare>::sift_up (size_type i)
127 {
128     while (i > 0 and m_compare (m_data[i], m_data[parent (i)]))
129     {
130         using std::swap;
131         swap (m_data[i], m_data[parent (i)]);
132         i = parent (i);
133     }
134 }
135
136 template <typename T, typename Compare>
137 void
138 Heap<T, Compare>::build_Heap (void)
139 {
140     for (size_type i = parent (m_data.size ());
141          i >= 0 and i < m_data.size (); --i)
142     {
143         sift_down (i);
144     }
145 }
146
147 template <typename T, typename Compare>
148 template <typename InputIterator>
149 Heap<T, Compare>::Heap (InputIterator first, InputIterator last,
150                         const value_compare & comp)
151 : m_data (first, last), m_compare (comp)
152 {
153     build_Heap ();
154 }
155
156 template <typename T, typename Compare>

```

```

157 Heap<T, Compare>::Heap (std::initializer_list<value_type> il,
158                         const value_compare & comp)
159     : m_data (il), m_compare (comp)
160 {
161     build_Heap ();
162 }
163
164 template <typename T, typename Compare>
165 void
166 Heap<T, Compare>::pop (void)
167 {
168     m_data.front () = m_data.back ();
169     m_data.pop_back ();
170     sift_down (0);
171 }
172
173 template <typename T, typename Compare>
174 void
175 Heap<T, Compare>::push (const value_type & element)
176 {
177     m_data.push_back (element);
178     sift_up (m_data.size () - 1);
179 }
180
181 template <typename T, typename Compare>
182 void
183 Heap<T, Compare>::replace (const value_type & element)
184 {
185     m_data.front () = element;
186     sift_down (0);
187 }
188 }
189
190 #endif

```

The methods required by the exercise are the **private** methods **sift_down** (see the declaration in lines 25 and 26) and **sift_up** (lines 27 and 28).

The first one is implemented in lines 90 to 122. It receives as parameter the index of the element we want to push down in the tree; it loops over all its children until it reaches a leaf and the *heap property* is not satisfied. At any iteration, the method computes the indices of the left and the right children of the given node (line 100) and it initializes the value of the index **best** with the current position. In lines 103 to 107, it uses the object **m_compare** to compare the values stored in the tree in position 1 and the one stored in position **best** (i.e. it compares the current value with the value of the left child). If the left child actually exists ($1 < m_data.size()$, i.e. we are not in a leaf) and **m_compare** returns **true** (which means that the value stored in the left child is better than the one in the current position), **best** takes the value of 1. The same procedure is performed with the right child. Finally, if **keep_sifting** is **true**, the element stored at position **best** and the current one are swapped. A similar strategy is used in the method **sift_up** (see lines 124 to 134): the main difference is that here we check that the index **i** is not the root of the tree and we compare the value stored in the current node with the one stored in its parent.

The method **sift_down** is intended to be used, for example, during the construction of an **Heap**. Indeed, consider the two constructors, implemented in lines 147 to 154 and 156 to 162,

respectively. In both cases, we use either the given iterators or the initializer list to initialize the data structure `m_data` (lines 151 and 159). This operation copies in `m_data` the values we pass as parameters, but the resulting data structure does not satisfy, in principle, the *heap property*. In order to recover this property, in lines 153 and 161 we call the `private` method `build_Heap`.

The method `build_Heap` considers as first index `parent(m_data.size())` and, until the index is greater than or equal to zero, it calls `sift_down` in order to restore the *heap property* of the tree.

Suppose, for example, to define an `Heap` object as follows:

```
ds::Heap<int> min_Heap {17, 18, 5, 7};
```

The constructor of lines 156 to 162 runs: the container `m_data` is initialized to [17, 18, 5, 7] and the method `build_Heap` is called. Within `build_Heap`, `m_data.size()` is equal to 4, therefore the first index `i` is 1. The method `sift_down` compares the value stored at the index 1, which is 18, with the value stored in its left child, which is 7; since the comparison operator is `std::less<int>` and 7 is smaller than 18, the two elements are swapped. Notice that the element with index 1 does not have a right child. After the swap, the container `m_data` stores [17, 7, 5, 18]. The new value of `best`, in `sift_down`, is 3; the node with index 3 has neither left nor right child (it is a leaf) and therefore the method ends. The *heap property* is still not satisfied. In `build_Heap`, the index `i` becomes equal to 0 and `sift_down` is called again. The element in position 0, i.e. 17, is compared with its left child, i.e. 7, and its right child, i.e. 5. The best value according to `m_compare` is 5, therefore `m_data` becomes [5, 7, 17, 18]. The new value of the index `best` is 2, which corresponds to a leaf. In `build_Heap`, `i` becomes equal to -1, which is not an admissible index; the method ends and the *heap property* of the tree is restored.

The complexity of the methods `sift_up` and `sift_down` is $O(\log N)$, where N is the number of elements stored in the heap.

Exercise 5 Social Network

Create the classes to store information about a social network where the users are identified by their name and their surname. The social network must store information about friendships and provides the following functionalities:

- `const std::unordered_set<User> CGetUser()`, which returns the set of users.
- `const std::unordered_set<User> & CGetFriends(const User user)`, which returns the set of friends of a user.
- `const std::unordered_set<User> CGetFriendsofFriends(const User user, const unsigned int distance)`, which returns the set of users connected by a chain of friendships of length `distance`.
- `CheckSixDegreeofSeparation()`, which checks if the six degree of separation rule holds (everybody in the world is six or fewer steps away from each other).

Exercise 5 - Solution

As it is described in the text, the class `User` stores two strings which represent name and surname of the user himself.

In order to represent the social network, i.e. to store the `Users` keeping track of the friendship relations among them, we use a

```
std::unordered_map<User, std::unordered_set<User>>
```

(see file "social_network.hpp"). With this implementation, each user, which is a key of the `unordered_map`, is related to an `unordered_set` of other users which are his friends. We use `unordered_maps` and `unordered_sets` over `maps` and `sets` because we do not need to define an ordering relation among users.

The fact that the type `User` is used as key in an `unordered_map` entails, first of all, that we need to define an equivalence relation among `Users`, for example by overloading the `operator ==`. In particular, two `Users` are equivalent if they have the same name and the same surname.

Moreover, since `User` is not a built-in type, in order to use it as key we also need to specialize the `hash` function that the `unordered_map` uses in order to store the elements. This specialization is performed in lines 48 to 60 of file "user.hpp". In particular, the `hash` function for `Users` shifts by one bit the value returned by the `std::hash` applied to `user`'s surname and then it combines through the `xor` operator the values returned by the `hash` functions applied to the user's name and surname (since `name` and `surname` are `std::strings`, `std::hash` is already specialized to deal with them).

***** file user.hpp *****

```

1 #ifndef USER_HPP_
2 #define USER_HPP_
3
4 //STD include
5 #include <string>
6
7 namespace SocialNetworkNS
8 {
9     class User
10    {
11        private:
12            ///The name
13            const std::string name;
14
15            ///The surname
16            const std::string surname;
17
18        public:
19            /**
20             * Constructor
21             * @param name is the name of the user
22             * @param surname is the surname of the user
23             */
24            User(const std::string & name, const std::string & surname);
25
26            /**
27             * Comparison operator
28             */
29            bool operator == (const User & other) const;
30
31            /**
32             * Return the name
33             */
34            const std::string & CGetName() const;
35

```

```

36     /**
37      * Return the surname
38      */
39     const std::string & CGetSurname() const;
40
41     /**
42      * Return the name and surname of the user
43      */
44     std::string ToString() const;
45 };
46 }
47
48 namespace std
49 {
50     template <>
51     struct hash<SocialNetworkNS::User> : public unary_function<SocialNetworkNS::
52     User, size_t>
53     {
54         size_t operator()(const SocialNetworkNS::User & user) const
55         {
56             return std::hash<std::string>{}(user.CGGetName() + user.CGGetSurname());
57         }
58     };
59 #endif

```

***** file user.cpp *****

```

1 //Header include
2 #include "user.hpp"
3
4 namespace SocialNetworkNS
5 {
6     User::User(const std::string & _name, const std::string & _surname) :
7         name(_name),
8         surname(_surname)
9     {}
10
11     bool User::operator == (const User & other) const
12     {
13         return this->name == other.name and this->surname == other.surname;
14     }
15
16     const std::string & User::CGGetName() const
17     {
18         return name;
19     }
20
21     const std::string & User::CGGetSurname() const
22     {
23         return surname;
24     }

```

```

26     std::string User::ToString() const
27     {
28         return name + " " + surname;
29     }
30 }
```

The class `SocialNetwork` stores the friendship relations among users in an `unordered_map` whose keys are the `Users` and whose values are `unordered_sets` of `Users` which represent the friends of the key.

Other than the ones required by the exercise, the class implements a method `AddUser` (see the declaration in line 50 of the file "social_network.hpp"), which receives as parameters name and surname of a new user and inserts him in the social network, appending to him an empty set of friends, and a method `AddFriendship` (line 59 to 62), which receives as parameters names and surnames of two users and inserts one in the set of friends of the other (and viceversa).

***** file `social_network.hpp` *****

```

1 #ifndef SOCIAL_NETWORK_HPP_
2 #define SOCIAL_NETWORK_HPP_
3
4 ///. include
5 #include "user.hpp"
6
7 //STL include
8 #include <unordered_map>
9 #include <unordered_set>
10
11 namespace SocialNetworkNS
12 {
13     class SocialNetwork
14     {
15         private:
16             ///The existing friendships between users
17             std::unordered_map<User, std::unordered_set<User> > friendships;
18
19         public:
20             /**
21             * Return the set of users
22             */
23             const std::unordered_set<User> CGetUsers() const;
24             /**
25             * Get the friends of a user
26             * @param user is the user
27             * @return the set of his/her friends
28             */
29             const std::unordered_set<User> & CGetFriends(const User & user) const;
30
31             /**
32             * Get the friends of friends
33             * @param user is the starting user
34             * @param distance is the maximum length of friendship chain to be considered
35             * @return the set of friends of friends
36             */
```

```

37     const std::unordered_set<User> CGetFriendsOfFriends(const User & user, const
38     unsigned int distance) const;
39
40     /**
41      * Check if the social network satisfies the six degree of separation rules
42      * @return true if the rule is satisfied, false otherwise
43      */
44     bool CheckSixDegreeOfSeparation() const;
45
46     /**
47      * add a new user to the social network
48      * @param name is the name of the user
49      * @param surname is the surname of the user
50      */
51     void AddUser(const std::string & name, const std::string & surname);
52
53     /**
54      * Add a friendship between two users
55      * @param first_name is the name of the first user
56      * @param first_surname is the surname of the first user
57      * @param second_name is the name of the second user
58      * @param second_surname is the surname of the second user
59      */
60     void AddFriendship(const std::string & first_name,
61                        const std::string & first_surname,
62                        const std::string & second_name,
63                        const std::string & second_surname);
64
65 };
66 #endif

```

***** file social_network.cpp *****

```

1 //Header include
2 #include "social_network.hpp"
3 #include "user.hpp"
4
5 //STD include
6 #include <iostream>
7
8 namespace SocialNetworkNS
9 {
10     void SocialNetwork::AddUser(const std::string & name, const std::string & surname)
11     {
12         User user = User(name, surname);
13         friendships[user] = std::unordered_set<User>();
14     }
15
16     const std::unordered_set<User> SocialNetwork::CGetUsers() const
17     {
18         std::unordered_set<User> ret;
19         for(const auto user : friendships)

```

```

20     {
21         ret.insert(user.first);
22     }
23     return ret;
24 }
25
26 const std::unordered_set<User> & SocialNetwork::CGetFriends(const User & user)
27     const
28 {
29     return friendships.find(user)->second;
30 }
31
32 const std::unordered_set<User> SocialNetwork::CGetFriendsOfFriends(const User &
33     user, const unsigned distance) const
34 {
35     std::unordered_set<User> ret;
36     ret.insert(user);
37     for(unsigned int iteration = 0; iteration < distance; iteration++)
38     {
39         std::unordered_set<User> temp;
40         for(const auto current_user : ret)
41         {
42             const auto friends = CGetFriends(current_user);
43             temp.insert(friends.begin(), friends.end());
44         }
45         ret.swap(temp);
46     }
47
48     auto iterator = ret.find(user);
49     if(iterator != ret.end())
50         ret.erase(iterator);
51
52     return ret;
53 }
54
55 bool SocialNetwork::CheckSixDegreeOfSeparation() const
56 {
57     if(friendships.empty())
58         return true;
59     for(const auto user : friendships)
60     {
61         if(CGetFriendsOfFriends(user.first, 6u).size() < friendships.size()-1)
62         {
63             return false;
64         }
65     }
66     return true;
67 }
68
69 void SocialNetwork::AddFriendship(const std::string & first_name,
70                                     const std::string & first_surname,
71                                     const std::string & second_name,

```

```

70                     const std::string & second_surname)
71 {
72     const User first_user = User(first_name, first_surname);
73     const User second_user = User(second_name, second_surname);
74     if(friendships.find(first_user) == friendships.end()
75         or friendships.find(second_user) == friendships.end()
76         or first_user == second_user)
77     {
78         return;
79     }
80     friendships[first_user].insert(second_user);
81     friendships[second_user].insert(first_user);
82 }
83 }
```

The method `CGetUsers` (see the implementation in lines 16 to 24 of file "social_network.cpp" for the declaration) returns the set of all the `Users` enrolled in the social network (i.e. all the keys stored in the map `friendships`).

The method `CGetFriendsOfFriends` (lines 31 to 44) receives as parameters an `User` and a `distance` and it returns the set of all the `Users` who are at most `distance` steps apart from the one passed as parameter. In particular, the method proceeds in the following way: first of all, it initialized the set it will return and it inserts in this set the user himself (we can consider that the distance between any `User` and himself is equal to 0). Then, it loops until the iteration number is less than `distance`; for every iteration, it loops over the `Users` already inserted in `ret` and it saves in `temp` all their friends, then it swaps the two containers. Indeed, supposing that `distance` is 3, when `iteration` is equal to 1 the set `ret` stores already `user` and his friends (whose distance from `user` is 1). In lines 38 to 42, we loop over all of them and we recover again `user` and his friends, but we take also the friends of his friends, whose distance from `user` is equal to 2.

Finally, in lines 46 to 48 we erase `user` from the set `ret`.

The method `CheckSixDegreeOfSeparation` (see lines 53 to 65) returns `true` if either the social network is empty or the maximum distance between all the users is equal to 6. In particular, this condition is checked by looping over all the `Users` in `friendships` and calling the method `CGetFriendsOfFriends`, passing 6 as `distance`: if the size of the set returned by this method is less than the size of `friendships` minus one (i.e. there are `Users` whose distance from the current one is greater than 6), the method returns `false`.

Exercise 6 Cl

Considering the class below:

```

1 #ifndef CL_HH
2 #define CL_HH
3
4 template <typename T>
5 class cl
6 {
7     private:
8     T *val = new T;
9
10    public:
11        cl(){} }
```

```

12     ~cl(){ delete val; }
13     cl(const cl&z) = default;
14     T* get_val() const {return val;}
15 };
16
17 #endif /* CL_HH */

```

and the function:

```

1 template <typename T>
2 void f (cl<T> c1)
3 {
4     cl<T> c2;
5     c2 = c1;
6 }

```

Explain what could go wrong when executing the following piece of code and how it can be fixed.

```

1 int main (void)
2 {
3     cl<int> c;
4     f(c);
5     cout << c.get_val() << '\n';
6 }

```

Exercise 6 - Solution

When function `f` is called by `c`, it copies `c` into `c1`. In line 4 of function `f`, a new object of type `cl<T>` is instantiated via the `default` constructor, thus a new pointer `c2.val` is allocated in the stack. The `default` copy assignment operator which is executed in line 5 assigns to `c2.val` the same address stored in `c1.val`, which produces a memory leakage because the value previously stored in `c2.val` is not deleted and it is lost forever. Moreover, when we exit the scope of function `f` and both the destructors of `c1` and `c2` are called, since `c1.val` and `c2.val` point to the same area in the memory, a double deallocation happens.

In the `main` function we have two additional issues: first of all, the pointer stored by `c`, whose value is copied in `c1.val` when we execute the function call in line 4, becomes invalid, because `c1.val` is deleted when we exit the scope of `f`. Therefore, in line 5, by calling `c.get_val`, we read the value of an invalid pointer, which leads to an implementation defined behaviour. Furthermore, when we exit the scope of the `main` function at the end of the program, the destructor of `c` is executed and the value pointed by `c.val` is deleted for the third time.

In order to solve these multiple issues, we have to implement in `class cl` a copy constructor and an assignment operator. A possible definition of both these methods is provided below:

```

18 template <typename T>
19 cl<T>::cl (const cl & rhs): val(new T (*rhs.val)) {}
20
21 template <typename T>
22 cl<T> & cl<T>::operator= (const cl & rhs)
23 {
24     if (this != &rhs)
25     {
26         delete val;
27         val = new T (*rhs.val);
28     }

```

```
29     return *this;
30 }
```

Notice that, both in line 19 and in line 22, any time we refer to `class cl` outside the scope of the class itself, we have to specify the template parameter.

Exercise 7 Counting Sort

Consider the following code snippets: organize them in files, with the necessary additions.

```
1. template <typename PRNG, typename OutputIterator>
void
create_numbers (PRNG & rng, unsigned count, OutputIterator first)
{
    for (unsigned i = 0; i < count; ++i)
    {
        *first++ = rng ();
    }
}
```

```
2. std::vector<unsigned>
counting_sort (const std::vector<unsigned> & data, unsigned bound)
{
    std::vector<unsigned> counts (bound, 0);
    for (auto it = data.cbegin (); it != data.cend (); ++it)
    {
        ++counts[*it];
    }
    for (unsigned i = 1; i < bound; ++i)
    {
        counts[i] += counts[i - 1];
    }
    std::vector<unsigned> result (data.size ());
    for (auto it = data.crbegin (); it != data.crend (); ++it)
    {
        result[--counts[*it]] = *it;
    }
    return result;
}
```

```
3. counting_sort (const std::vector<unsigned> &, unsigned);
```

```
4. int main (void)
{
    constexpr unsigned bound = 8;
    std::mt19937 generator;

    std::uniform_int_distribution<unsigned> distribution (0, bound);

    auto pseudo_random = std::bind (distribution, generator);

    constexpr unsigned count = 10;
```

```

    std::vector<unsigned> numbers;

    generation::create_numbers (pseudo_random, count,
                               std::back_inserter (numbers));

    const std::vector<unsigned> sorted =
        sorting::counting_sort (numbers, bound + 1);

    for (unsigned value : sorted)
    {
        std::cout << value << std::endl;
    }

    return 0;
}

```

Exercise 7 - Solution

The file organization of the code snippets reported above can be recovered especially from the `main` function. Indeed, we can see that both the function `create_numbers` and the function `counting_sort` (which correspond to the first and the second code excerpts) are protected under a `namespace`. The fact that the two `namespaces` are different suggests that the two functions should be implemented in different files.

Once we observe this, we can easily understand that the function `create_numbers` must be declared and defined in an header file, because it is a `template` function (we call this file "generation.hpp").

On the other hand, declaration and definition of `counting_sort` are splitted in the code excerpts 2 and 3 and we can assume them to be written in two different files, namely "sorting.hpp" and "sorting.cpp".

Remember that in all these files we have to write the relevant `includes` and to define the `namespaces` we saw in the `main` function.

***** file generation.hpp *****

```

1 #ifndef GENERATION_HH
2 #define GENERATION_HH

3
4 namespace generation {
5
6     template <typename PRNG, typename OutputIterator>
7     void
8     create_numbers (PRNG & rng, unsigned count, OutputIterator first)
9     {
10         for (unsigned i = 0; i < count; ++i)
11         {
12             *first++ = rng ();
13         }
14     }
15
16 }
17 #endif /* GENERATION_HH */

```

The file "generation.hpp" stores the implementation of a **template** function, namely **create_numbers**, which is used to generate a given amount of numbers. The **template** parameter PRNG identifies the rule which is used to generate those numbers, which are created and inserted in a container starting from the position specified by **first**.

***** file sorting.hpp *****

```

1 #ifndef SORTING_HH
2 #define SORTING_HH
3
4 #include <vector>
5 namespace sorting {
6
7     std::vector<unsigned>
8     counting_sort (const std::vector<unsigned> &, unsigned);
9
10 }
11
12
13
14 #endif /* SORTING_HH */
```

The function **counting_sort** receives as parameters a vector of **unsigned** we want to sort and a variable which represents the maximum number of different elements we can have in the vector we want to sort. It returns the sorted vector.

***** file sorting.cpp *****

```

1 #include "sorting.hpp"
2
3 namespace sorting {
4
5     std::vector<unsigned>
6     counting_sort (const std::vector<unsigned> & data, unsigned bound)
7     {
8         std::vector<unsigned> counts (bound, 0);
9         for (auto it = data.cbegin (); it != data.cend (); ++it)
10         {
11             ++counts[*it];
12         }
13         for (unsigned i = 1; i < bound; ++i)
14         {
15             counts[i] += counts[i - 1];
16         }
17         std::vector<unsigned> result (data.size ());
18         for (auto it = data.crbegin (); it != data.crend (); ++it)
19         {
20             result[~counts[*it]] = *it;
21         }
22         return result;
23     }
24 }
```

First of all, the function `counting_sort` initializes a vector of zeros whose size is equal to `bound`, i.e. to the maximum number of different `unsigneds` we might have in the vector `data`. In lines 9 to 12, it loops over all the elements in `data` and it increments by one the corresponding counter. If, for example, `bound` is equal to 9 and the vector `data` stores:

```
6 5 1 3 5 4 0 7 4 1
```

at the end of the for loop the corresponding vector `counts` turns out to be:

```
1 2 0 1 2 2 1 1 0
```

because in `data` we have one zero, two ones, zero two and so on.

In lines 13 to 16, the vector `counts` is modified by adding to each element the value stored in the previous one, so that, keeping the example above, we have:

```
1 3 3 4 6 8 9 10 10
```

Finally, in lines 17 to 21 the vector `result` (which will store the sorted numbers) is initialized and then the values we have in `data` are copied there according to the information given by `counts`. The final value of `result` which is returned by the function is:

```
0 1 1 3 4 4 5 5 6 7
```

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

```

1 #include <functional>
2 #include <iostream>
3 #include <random>
4
5 #include "generation.hpp"
6 #include "sorting.hpp"
7
8 int main (void)
9 {
10    constexpr unsigned bound = 8;
11    std::mt19937 generator;
12
13    std::uniform_int_distribution<unsigned> distribution (0, bound);
14
15    auto pseudo_random = std::bind (distribution, generator);
16
17    constexpr unsigned count = 10;
18    std::vector<unsigned> numbers;
19
20    generation::create_numbers (pseudo_random, count,
21                               std::back_inserter (numbers));
22
23    const std::vector<unsigned> sorted =
24        sorting::counting_sort (numbers, bound + 1);
25
26    for (unsigned value : sorted)
27    {
28        std::cout << value << std::endl;
29    }
30
31    return 0;
32 }
```

Within the `main` function, we initialize a generator of random numbers (line 11) and a distribution of `unsigned` between 0 and `bound` (line 13). The instruction in line 15 instantiates an object, `pseudo_random`, which can be used to generate pseudo random numbers with the characteristics specified by `distribution` and `generator`.

In lines 20 and 21, we call the function `generation::create_numbers`, passing as parameters the object we use to generate random numbers, the number of elements we want to create and `std::back_inserter(numbers)`, so that the numbers are appended at the end of the vector `numbers`.

In lines 23 and 24, we use `sorting::counting_sort` to sort the vector returned by `generation ::create_numbers`. Of course, if the random numbers are generated between 0 and `bound`, the maximum number of different entries we can have in `numbers` is `bound + 1`.

Exercise 8 Genetic Algorithms (exam 31/08/2018)

Genetic algorithms (GAs) are a class of evolutionary optimization techniques. The basic idea is to mimic the biological processes that underline evolution for organisms, thus obtaining a meta-heuristic approach viable to optimize a vast range of problems.

In particular, genetic algorithms store a collection of candidate solutions, called *genotypes* or *individuals*. In every iteration, suggestively named *generation*, all the individuals in the *population* are evaluated according to their *fitness* (usually their objective function values). Subsequent generations consist of a fraction of individuals that simply survive, via *selection*, and of a complementary part with offspring deriving from the *recombination* of pairs of parent individuals. Additionally, the gene pool can be enriched by *mutation*.

The pseudo-code of a basic GA implementation is reported in Algorithm 8.1, where n is the number of individuals in the population, χ is the fraction of the population to be replaced by crossover in each iteration, and μ is the mutation rate.

```

1 Algorithm: GA ( $n, \chi, \eta$ )
2
3 // Initialize generation 0:
4 k := 0;
5  $P_k$  := a population of  $n$  randomly-generated individuals;
6
7 // Evaluate  $P_k$ :
8 Compute fitness( $i$ ) for each  $i \in P_k$ ;
9 do
10 {
11     // Create generation}  $k+1$ :
12     // 1. Copy:
13     Select  $(1 - \chi) \times n$  members of  $P_k$  and insert into  $P_{k+1}$ ;
14     // 2. Crossover:
15     Select  $\chi \times n$  members of  $P_k$ ;
16     pair them up;
17     produce offspring;
18     insert the offspring into  $P_{k+1}$ ;
19     // 3. Mutate:
20     Select  $\mu \times n$  members of  $P_{k+1}$ ;
21     invert a randomly-selected bit in each;
22     // Evaluate  $P_{k+1}$ :
23     Compute fitness( $i$ ) for each  $i \in P_k$ ;
```

¹From <http://www.cs.ucc.ie/~dgb/courses/tai/notes/handout12.pdf>

```

24     // Increment:
25     k := k+1;
26 }
27 while fitness of fittest individual in Pk is not high enough;
28
29 return the fittest individual from Pk;

```

Algorithm 8.1: GA pseudo-code

***** Serial Version *****

Your task is to search the optimal solution to a multi-dimensional knapsack problem (MKP) via a GA. Recall that the MKP is an extension of the widely known knapsack problem, where items are characterized by multiple dimensions $d \in D$. In particular, the knapsack has multiple capacities C_d and we can pick from a set N of items, with each item $1 \leq j \leq n$ having an associated value v_j and a tuple of weights w_{jd} for each $d \in D$. The binary decision variable x_j is used to select each item. The objective is to pick the optimal subset of items with maximum total value, while meeting the capacity constraints on all the relevant dimensions. In other words, the MKP can be formulated as:

$$\max \sum_{j \in N} v_j x_j$$

subject to:

$$\begin{aligned} \sum_{j \in N} w_{jd} x_j &\leq C_d, \quad \forall d \in D, \\ x_j &\in \{0, 1\}, \quad \forall j \in N. \end{aligned}$$

A third party template library provides the implementation of the GA meta-heuristic, hence you only need to complete the implementation of a **class** that meets the required specification.

```

1  class multi_knapsack
2  {
3      public:
4          using solution_type = std::vector<bool>;
5
6      private:
7          const std::size_t m_items;
8          std::vector<double> m_values;
9          std::vector< std::vector<double> > m_weights;
10         std::vector<double> m_capacities;
11         std::mt19937_64 m_random_engine;
12
13         bool
14         check_feasibility (solution_type const&) const;
15
16     public:
17         explicit multi_knapsack (std::size_t items, std::size_t seed = 0);
18
19         void
20         add_dimension (double capacity, std::vector<double> const& weights);
21         void

```

```

22 add_values (std::vector<double> const&);

23

24 solution_type
25 random_individual (void);
26 double
27 individual_fitness (solution_type const&) const;
28 std::pair<solution_type, solution_type>
29 cross_over (solution_type const&, solution_type const&);

30 void
31 mutate (solution_type &);

32 void
33 make_feasible (solution_type &);

34 };

```

The above excerpt provides the **class** definition for `multi_knapsack`. Individuals are represented through `solution_type`, where each gene is `true` if the corresponding item is in the knapsack, `false` otherwise. Clearly, all the individuals have the same length. **The only members you are required to implement** are `cross_over`, which takes two parents and produces two offsprings, `mutate`, which takes an individual and flips a random gene in place, and `make_feasible`, which enforces the feasibility of the given individual in place, if it is infeasible, by making it a one-item solution.

Figure 8.5 shows `cross_over` and `mutate` operations. `cross_over` is the operation performed during recombination. Given the two parents, extract with uniform probability an index, cut their genotypes at the obtained position, and create two children so that one has the first half of genes coming from the "father" and the second from the "mother", the other is formed as "mother" - "father". Similarly, `mutate` works by extracting with uniform probability an index and flipping the corresponding gene in the given individual. In the end, `make_feasible` first checks whether the solution is feasible. If this is the case, the function returns without further ado. Otherwise, it assigns all the genes `false` and picks with uniform probability a single item to include in the knapsack.

In the end, discuss the complexity of the implemented methods in terms of the size of each individual, or number of possible items in the knapsack problem.²

***** Parallel Version *****

Elitism in population based meta-heuristics like GAs is a general process for constructing a new population by allowing the best individuals from the current generation to carry over to the next one without changes. It is frequently considered to be an effective practical variant for selecting or constructing new populations.

Provide a parallel program that implements GA elitism. The basic ideas are the following:

²Some useful STL routines are listed below:

- `<algorithm> std::copy_n`
`template< class InputIt, class Size, class OutputIt >`
`OutputIt copy_n(InputIt first, Size count, OutputIt result);`
- `<algorithm> std::swap_ranges`
`template< class ForwardIt1, class ForwardIt2 >`
`ForwardIt2 swap_ranges(ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2);`
- `<random> std::uniform_int_distribution`
`template< class IntType = int >`
`class uniform_int_distribution;`
- `<random> std::uniform_real_distribution`
`template< class RealType = double >`
`class uniform_real_distribution;`

For the relative documentation, see www.cppreference.com or www.cplusplus.com

- Initially each process randomly initializes a population of n individuals.
- After a given number of generations, processes share with each other their *top-k individuals* (according to their fitness values).
- The parallel algorithm then proceeds with the same schema and stops when a given number of parallel iterations is reached.

For the sake of simplicity assume that n is a multiple of the number of available processes p and the best $k = n/p$ individuals are retrieved from each process' population.

In particular, complete the following code by providing also the implementation of the `share_top_k` function, which returns in each process the new population starting from the `local_top_k`. You can rely on all the members listed in the main and on the `set_population` member function (which receives as argument a new population) already implemented for you in the `genetic` class template. Finally, note that `population_type`, member type of the `genetic` class template, is defined in the appropriate header as follows:

```
using population_type = std::vector<typename problem_type::solution_type>;
```

Complete the "main.cc" file reported below with the required implementation.

```

1 // file: main.cc
2 #include <algorithm>
3 #include <iostream>
4 #include <iterator>
5 #include <mpi.h>
6 #include <vector>
7
8 #include "genetic.hh"
9 #include "multi_knapsack.hh"
10
11 using population_t =
12     optimization::genetic<optimization::multi_knapsack>::population_type;
13
14 population_t
15 share_top_k (const population_t & local_top_k);
16
17
18 int
19 main (int argc, char * argv[])
20 {
21     MPI_Init (&argc, &argv);
22     int rank{0}, size{0};
23     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
24     MPI_Comm_size (MPI_COMM_WORLD, &size);
25
26     // Problem instance definition
27     optimization::multi_knapsack global_instance (20, 17);
28     global_instance.add_dimension (36., {11., 20., 5., 10., 20., 1., 20., 18., 9., 2., 3., 15.,
29                                     16., 21., 23., 12., 11., 15., 11., 10.});
30     global_instance.add_dimension (28., {11., 10., 1., 15., 10., 7., 12., 19., 8., 18., 3., 19.,
31                                     12., 8., 15., 6., 3., 12., 8., 7.});
32     global_instance.add_values ({30., 21., 15., 10., 7., 32., 39., 8., 38., 3., 39., 32., 28.,
33                                15., 20., 23., 32., 8., 27., 30.});
34 }
```

```

32 // Create a GA solver
33 auto local_metaheuristic =
34     optimization::make_genetic (global_instance, 0.7, 0.05, 200 * rank * rank);
35
36 constexpr std::size_t parallel_iterations = 10;
37 constexpr std::size_t local_generations = 100;
38 constexpr std::size_t population_size = 100;
39
40 std::cout << "Initializing population\n";
41 local_metaheuristic.initialize_population (population_size);
42 local_metaheuristic.optimize (local_generations, std::cout);
43 const std::size_t top_k = population_size / size;
44
45 // Perform parallel iterations
46 // ----- YOUR CODE GOES HERE -----
```



```

64 // print outputs
65 if (rank == 0)
66 {
67     std::cout << "Final population:\n";
68     for (auto const & individual : local_metaheuristic.get_population())
69     {
70         for (bool item : individual) std::cout << item;
71         std::cout << "\n";
72     }
73
74     std::cout << "Top 10:\n";
75     for (auto const & individual : local_metaheuristic.get_top_k (10))
76     {
77         for (bool item : individual) std::cout << item;
78         std::cout << " fitness = "
79             << global_instance.individual_fitness (individual) << '\n';
80     }
81 }
82
83 MPI_Finalize();
84 return 0;
85 }
```

Exercise 8 - Solution

Note: in order to show the complete solution of this exercise, we report here also the header file "genetic.hh", which is included both in the solution of the serial version and of the parallel version. Consider this an advanced reading.

```

1 #ifndef GENETIC_HH
2 #define GENETIC_HH
3
4 #include <algorithm>
5 #include <cmath>
6 #include <functional>
7 #include <iostream>
8 #include <random>
```

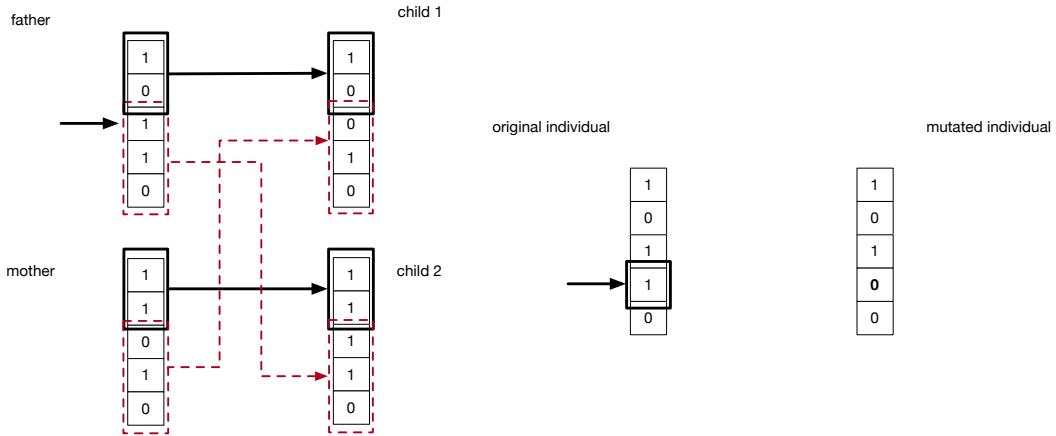


Figure 8.5: Crossover (left) and mutation (right) operations for MKP

```

9 #include <utility>
10 #include <vector>
11
12 namespace optimization
13 {
14     template <typename Problem>
15     class genetic
16     {
17     public:
18         using problem_type = Problem;
19         using population_type = std::vector<typename problem_type::solution_type>;
20         using fitness_type = std::vector<double>;
21
22     private:
23         problem_type m_problem;
24         const double m_crossover_fraction;
25         const double m_mutation_probability;
26         std::mt19937_64 m_random_engine;
27         population_type m_population;
28         fitness_type m_fitness;
29
30     public:
31         genetic (problem_type const&, double crossover,
32                  double mutation, std::size_t seed = 0);
33
34     private:
35         void
36         evaluate_fitness (void);
37         population_type
38         select (std::size_t);
39         population_type
40         recombine (void);
41         void
42         mutate (void);
43         void
44         enforce_feasibility (void);
45         void

```

```

46     evolve_generation (void);
47
48 public:
49     void
50     initialize_population (std::size_t population_size);
51     void
52     optimize (std::size_t generations, std::ostream &);

53     population_type
54     get_top_k (std::size_t k);

55     population_type const&
56     get_population (void) const;
57     void
58     set_population (population_type const&);

59 };
60
61
62 template <typename Problem>
63     genetic<Problem>
64     make_genetic (Problem const& problem, double crossover, double mutation,
65                   std::size_t seed = 0)
66 {
67     genetic<Problem> ga (problem, crossover, mutation, seed);
68     return ga;
69 }
70
71
72 template <typename Problem>
73     genetic<Problem>::genetic (problem_type const& problem, double crossover,
74                               double mutation, std::size_t seed)
75     : m_problem (problem), m_crossover_fraction (crossover),
76       m_mutation_probability (mutation), m_random_engine (seed) {}

77
78 template <typename Problem>
79     void
80     genetic<Problem>::initialize_population (std::size_t population_size)
81 {
82     m_population.clear ();
83     std::generate_n (std::back_inserter (m_population), population_size,
84                      std::bind (&problem_type::random_individual, &m_problem));
85 }

86
87 template <typename Problem>
88     void
89     genetic<Problem>::evaluate_fitness (void)
90 {
91     using namespace std::placeholders;
92     fitness_type population_fitness;
93     std::transform (m_population.cbegin (), m_population.cend (),
94                    std::back_inserter (population_fitness),
95                    std::bind (&problem_type::individual_fitness,
96                               &m_problem, _1));
97     using std::swap;

```

```

98     swap (population_fitness, m_fitness);
99 }
100
101 template <typename Problem>
102 typename genetic<Problem>::population_type
103 genetic<Problem>::select (std::size_t count)
104 {
105     std::discrete_distribution<typename population_type::size_type>
106     sampler (m_fitness.cbegin (), m_fitness.cend ());
107     population_type selected;
108     std::generate_n (std::back_inserter (selected), count,
109                     [&z] (void)
110                     {
111             const typename population_type::size_type index =
112                 sampler (m_random_engine);
113             return m_population[index];
114         });
115     return selected;
116 }
117
118 template <typename Problem>
119 typename genetic<Problem>::population_type
120 genetic<Problem>::recombine (void)
121 {
122     const std::size_t parent_count =
123         std::round (m_crossover_fraction * m_population.size () / 2);
124     population_type parent1 = select (parent_count),
125         parent2 = select (parent_count), recombinet;
126     typename population_type::iterator it1 = parent1.begin (),
127         it2 = parent2.begin ();
128     while (it1 != parent1.end ())
129     {
130         std::pair<typename problem_type::solution_type,
131                   typename problem_type::solution_type>
132         pair = m_problem.cross_over (*it1++, *it2++);
133         recombinet.push_back (pair.first);
134         recombinet.push_back (pair.second);
135     }
136     return recombinet;
137 }
138
139 template <typename Problem>
140 void
141 genetic<Problem>::mutate (void)
142 {
143     std::bernoulli_distribution bernoulli (m_mutation_probability);
144     for (typename problem_type::solution_type & individual : m_population)
145         if (bernoulli (m_random_engine))
146             m_problem.mutate (individual);
147 }
148
149 template <typename Problem>

```

```

150 void
151 genetic<Problem>::enforce_feasibility (void)
152 {
153     for (typename problem_type::solution_type & individual : m_population)
154         m_problem.make_feasible (individual);
155 }
156
157 template <typename Problem>
158 void
159 genetic<Problem>::evolve_generation (void)
160 {
161     evaluate_fitness ();
162     const std::size_t selected_count =
163         std::round ((1 - m_crossover_fraction) * m_population.size ());
164     population_type next_generation = select (selected_count),
165         crossed_over = recombine ();
166     next_generation.insert (next_generation.end (),
167                             crossed_over.cbegin (), crossed_over.cend ());
168     using std::swap;
169     swap (next_generation, m_population);
170     mutate ();
171     enforce_feasibility ();
172 }
173
174 template <typename Problem>
175 void
176 genetic<Problem>::optimize (std::size_t generations, std::ostream & out)
177 {
178     for (std::size_t generation = 1; generation <= generations; ++generation)
179     {
180         out << "Evolving generation " << generation << ": ";
181         evolve_generation ();
182         out << "population size = " << m_population.size () << ", ";
183         const typename problem_type::solution_type best =
184             get_top_k (1).front ();
185         const double fitness = m_problem.individual_fitness (best);
186         out << "best fitness = " << fitness << '\n';
187     }
188 }
189
190 template <typename Problem>
191 typename genetic<Problem>::population_type
192 genetic<Problem>::get_top_k (std::size_t k)
193 {
194     evaluate_fitness ();
195     std::vector<typename population_type::size_type> indices;
196     for (typename population_type::size_type i = 0;
197          i < m_population.size (); ++i)
198     {
199         indices.push_back (i);
200     }
201     std::nth_element (indices.begin (), indices.begin () + k, indices.end ());

```

```

202     [&] (typename population_type::size_type i,
203           typename population_type::size_type j)
204     {
205         return m_fitness[i] > m_fitness[j];
206     });
207     indices.resize (k);
208     population_type top_k;
209     for (typename population_type::size_type i : indices)
210         top_k.push_back (m_population[i]);
211     return top_k;
212 }
213
214 template <typename Problem>
215 typename genetic<Problem>::population_type const&
216 genetic<Problem>::get_population (void) const
217 {
218     return m_population;
219 }
220
221 template <typename Problem>
222 void
223 genetic<Problem>::set_population (population_type const& population)
224 {
225     m_population = population;
226 }
227
228 #endif // GENETIC_HH

```

***** Solution of the Serial Version *****

In the solution reported below, we show the full implementation of the class `multi_knapsack`, even if only the methods `cross_over`, `mutate` and `make_feasible` were required during the exam.

First of all, notice that the class `multi_knapsack` stores the dimension of every item (line 7 of file "multi_knapsack.hh" reported above), the values associated to every item, the corresponding matrix of weights and the vector of capacities. Moreover, it stores a generator of random numbers, that is initialized within the constructor by a given `seed` (for a more extended discussion about the initialization of a random engine, see Exercise 7 in Section 6.1, Chapter 6).

Other than the ones required by the text, the class implements a `private` method, `check_feasibility`, which receives as parameter a vector of `bool` and returns `true` if it is feasible, and four `public` methods. The implementation of all these methods is described below.

```

1 #include <algorithm>
2
3 #include "multi_knapsack.hh"
4
5 namespace optimization
6 {
7     multi_knapsack::multi_knapsack (std::size_t items, std::size_t seed)
8         : m_items (items), m_random_engine (seed) {}

```

```

9
10 void
11 multi_knapsack::add_dimension (double capacity,
12                               std::vector<double> const& weights)
13 {
14     m_capacities.push_back (capacity);
15     m_weights.push_back (weights);
16 }
17
18 void
19 multi_knapsack::add_values (std::vector<double> const& values)
20 {
21     m_values = values;
22 }
23
24 bool
25 multi_knapsack::check_feasibility (solution_type const& individual) const
26 {
27     bool feasible = true;
28     std::vector< std::vector<double> >::const_iterator w_it =
29         m_weights.cbegin ();
30     std::vector<double>::const_iterator c_it = m_capacities.cbegin ();
31     while (feasible and c_it != m_capacities.cend ())
32     {
33         std::vector<double> const& current_weights = *w_it++;
34         double total = 0.0;
35         for (std::size_t i = 0; i < m_items; ++i)
36             if (individual[i])
37                 total += current_weights[i];
38         feasible = (total <= *c_it++);
39     }
40     return feasible;
41 }
42
43 multi_knapsack::solution_type
44 multi_knapsack::random_individual (void)
45 {
46     std::bernoulli_distribution bernoulli (0.5);
47     solution_type individual;
48     for (std::size_t i = 0; i < m_items; ++i)
49         individual.push_back (bernoulli (m_random_engine));
50     return individual;
51 }
52
53 double
54 multi_knapsack::individual_fitness (solution_type const& individual) const
55 {
56     double fitness = 0.0;
57     if (check_feasibility (individual))
58     {
59         for (std::size_t i = 0; i < m_items; ++i)
60             if (individual[i])

```

```

61         fitness += m_values[i];
62     }
63     return fitness;
64 }
65
66 std::pair<multi_knapsack::solution_type, multi_knapsack::solution_type>
67 multi_knapsack::cross_over (solution_type const& one,
68                             solution_type const& two)
69 {
70     std::uniform_int_distribution<solution_type::size_type>
71     sampler (0, m_items - 1);
72     const solution_type::size_type index = sampler (m_random_engine);
73     std::pair<multi_knapsack::solution_type, multi_knapsack::solution_type>
74     offspring {one, two};
75     std::swap_ranges (offspring.first.begin () + index, offspring.first.end (),
76                       offspring.second.begin () + index);
77     return offspring;
78 }
79
80 void
81 multi_knapsack::mutate (solution_type & individual)
82 {
83     std::uniform_int_distribution<solution_type::size_type>
84     sampler (0, m_items - 1);
85     const solution_type::size_type index = sampler (m_random_engine);
86     individual[index] = not individual[index];
87 }
88
89 void
90 multi_knapsack::make_feasible (solution_type & individual)
91 {
92     if (not check_feasibility (individual))
93     {
94         individual.assign (m_items, false);
95         std::uniform_int_distribution<solution_type::size_type>
96         sampler (0, m_items - 1);
97         const solution_type::size_type index = sampler (m_random_engine);
98         individual[index] = true;
99     }
100 }
101 }
```

The method `check_feasibility` (see lines 24 to 41) receives as parameter an individual, represented by a vector of `bools`, where each element is `true` if the corresponding item is stored in the knapsack. It loops over all the vector of capacities; at every iteration, it extracts the corresponding vector of weights and it loops over all the genes in `individual`, summing up the weights of the items whose gene is `true`. The parameter `feasible` becomes `false` if the total sum of these wheights exceeds the current capacity.

The method `random_individual` (see lines 43 to 51) generates a random individual by setting every gene to `true` or `false` according to a Bernoulli distribution with parameter $p = 0.5$.

The method `individual_fitness` (see lines 53 to 64) receives as parameter an individual and computes the corresponding *fitness*. In particular, if the individual is feasible, its *fitness* is computed summing up the values of all the `true` genes.

The method `cross_over` (lines 66 to 78) performs the crossover operation described in Figure 8.5. It receives as parameters the two individuals that must be combined during the crossover, it instantiates an `uniform_int_distribution` and it uses this distribution to generate a random index between 0 and `m_items - 1`.

In lines 73 and 74, it instantiates a pair of individuals, called `offspring`, by passing as parameters the two initial individuals `one` and `two`. Finally, in lines 75 and 76 it relies on the function `std::swap_ranges` in order to perform the crossover. Indeed, the function:

```
std::swap_ranges( ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2 );
```

swaps the elements contained in the range $[first1, last1]$ with a corresponding number of elements starting from `first2`, so that, in this case, all the elements between

```
offspring.first.begin() + index
```

and

```
offspring.first.end()
```

(which correspond to the elements between `index` and the end of `one`) are swapped with the corresponding elements of `offspring.second`, which corresponds to `two`.

The method `mutate` (see lines 80 to 87) extracts with uniform probability an index between 0 and `m_items - 1` and it modifies the corresponding gene of the individual passed as parameter, so that it passes from `true` to `false` or viceversa.

Finally, the method `make_feasible` (lines 89 to 100) receives an individual as parameter and, if it is not feasible, it assings the value `false` to all its genes and it randomly selects an index, setting the corresponding gene to `true`.

Let n the number of items in the knapsack formulation, hence also the size of each solution. `mutate` is $O(1)$, as it only flips one random bit. `cross_over` is $O(n)$, since you need to create copies of the parents and then swap part of their values. Similarly, `make_feasible` is $O(n)$, given the fact that checking feasibility requires going through all the items in the solution.

***** Solution of the Parallel Version *****

First of all, notice that the `multi_knapsack` class we are considering in the parallel version is exactly the same whose implementation we saw in the serial version of the exercise.

In lines 27 to 30 of the "main.cc" file reported above, we instantiate a problem by defining a `multi_knapsack` object with dimension equal to 20 and `seed` equal to 17, then we call twice the method `add_dimension`, in order to add the capacity and the corresponding vectors of weights, we call `add_values` in order to set the values.

In lines 33 and 34, we initialize a GA solver by relying on the function `make_genetic` defined in header "genetic.hh". Finally, in lines 41 to 43, we set the number of parallel iterations we want to perform, the number of local generations and the size of the population we want to create.

Once we have initialized the population and we have performed an initial optimization (lines 41 and 42), we define the index `top_k` as the ratio between the size of the population and the number of available processes and then we start with the parallel optimization (see lines 48 to 62 reported below).

```
48 for (std::size_t it = 0; it < parallel_iterations; ++it)
49 {
50     // share top k among all processes
51     const population_t & new_global_population =
52         share_top_k (local_metaheuristic.get_top_k (top_k));
53     // set new population
54     local_metaheuristic.set_population (new_global_population);
```

```

55     // perform optimization again
56     local_metaheuristic.optimize (local_generations, std::cout);
57 }
58
59 // share again top_k to get the best result
60 const population_t & final_global_population =
61     share_top_k (local_metaheuristic.get_top_k (top_k));
62 local_metaheuristic.set_population (final_global_population);

```

The parallel optimization is performed in this way: we loop until the maximum number of iterations and, for every step, we instantiate a new population with the value returned by `share_top_k`, we set this population in the GA solver and we perform the optimization. Notice that, at the end of the for loop (lines 60 to 62), we have to call again `share_top_k` in order to ensure that all the processes share the same optimal value.

The implementation of the function `share_top_k` is reported below:

```

88 population_t
89 share_top_k (const population_t & local_top_k)
90 {
91     int rank{0}, size{0};
92     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
93     MPI_Comm_size (MPI_COMM_WORLD, &size);
94     // initialize new_population with the local top_k elements
95     population_t new_population = local_top_k;
96
97     // for all processes
98     for (int current_proc = 0; current_proc < size; ++current_proc)
99     {
100         // for every individual in local_top.
101         // Note that here we exploit the fact that each process has the
102         // same number of elements and that overall their number is equal to the
103         // initial population size otherwise you should send
104         // (or better compute) their number
105         for (auto const & individual : local_top_k)
106         {
107             std::vector<unsigned short> item_unsigned_short;
108             // convert bool vector into unsigned short vector
109             std::copy (individual.cbegin (), individual.cend (),
110                         std::back_inserter (item_unsigned_short));
111
112             MPI_Bcast (item_unsigned_short.data(), item_unsigned_short.size(),
113                         MPI_UNSIGNED_SHORT,
114                         current_proc, MPI_COMM_WORLD);
115
116             if (rank != current_proc)
117             {
118                 // add item to new_population
119                 optimization::multi_knapsack::solution_type received;
120                 // convert unsigned short vector into bool vector
121                 std::copy (item_unsigned_short.cbegin (),
122                             item_unsigned_short.cend (),
123                             std::back_inserter (received));
124                 new_population.push_back (received);

```

```

125     }
126   }
127 }
128
129 return new_population;
130 }
```

The function `share_top_k` is called by every process in order to share the *top-k* individuals passed as input. In particular, in line 95 a new population is initialized in every process by its own *top-k* individuals. Then, the function loops over all the ranks and it performs the communication.

The idea is the following: we want to loop over all the *top-k* individuals we have in `local_top_k` and we want to perform a broadcast in order to share every individual between all the processes. In order to do this, however, we need to convert the vectors of `bools` that correspond to every individual in vectors of, for example, `unsigned short`. Indeed, `bool` is not a type that can be directly communicated in MPI. Therefore, in lines 107 to 115 we convert the current individual in a vector of `unsigned_short` and then we perform the `MPI_Bcast` between all the processes (remember that, since `MPI_Bcast` is a collective routine, we can call it only once and it holds both when the current process is the sender and when it is the receiver). Finally, any time a process receives one of the *top-k* individuals from another rank, it has to update its population, so that, in the end, all the processes share the same population, made by the *top-k* individuals recovered from any rank. In order to do this, in lines 116 to 125, if `rank` is different from `current_proc`, i.e., the process is not the sender, we initialize a `bool` vector, namely `received`, in which we copy the values received from the sender, and we append this vector to the new population.

Exercise 9 Deque

`Deque` is a sequence container with dynamic size that can be expanded or contracted on both ends (either its front or its back). In fact, each `Deque` keeps a double ended queue of fixed-size chunks. As it is shown in Figure 8.6, each `Deque` can be implemented as a vector of arrays with fixed size; the number of arrays, i.e. the size of the vector, can grow dynamically as we push or pop elements from the `Deque`.

Write `Deque` as a `template` class that implements `push_back` (a new entry is added inside the last chunk; if the last chunk is full, a new chunk is allocated) and `push_front` (the entry is added inside the first chunk; if the first chunk is full, a new chunk is allocated) methods and overloads `operator[]`. Instead of using a vector of arrays, implement your data structure as a `std::vector<std::vector<T>>`, where the internal vectors have fixed size.

Discuss the complexity of the methods above and explain when you would choose to use a `Deque` rather than a vector.

Exercise 9 - Solution

The class `Deque` stores a vector of chunks (see line 18) and two indices, `first_offset` and `off_end_offset`. The first one represents the global index of the first inserted element, while the second one is the global index of the first empty element, i.e. the global index of the first free position in the container. Both are not fixed because a `Deque` can dynamically grow both in the front and in the back.

Notice that both `first_offset` and `off_end_offset` are global indices, meaning that, considering, for example, the situation in Figure 8.6, `first_offset` is equal to 2 and `off_end_offset` is equal to 11.

The number of elements that can be stored in a chunk and the initial number of chunks in

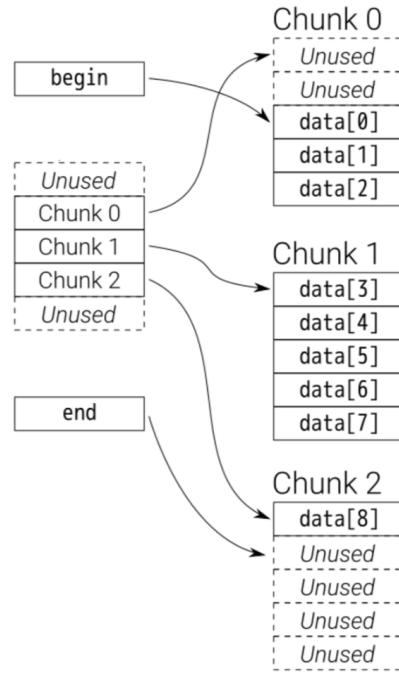


Figure 8.6: Structure of a deque.

the container are set in lines 20 and 21, respectively.

The class implements two **private** methods: `on_boundary` returns `true` if the index passed as parameter is on the boundary of a chunk. In particular, this is needed in the method `push_front` when the index is equal to 0 and in the method `push_back` and the index is a multiple of `chunk_length`. Indeed, in the first case we need to add the new element in the last position of the previous chunk, while in the second case we have to add the new element at the first position of the following one.

The method `make_room_for`, in turn, receives as parameter the number of elements we are adding to the `Deque` and it ensures that the container has enough space to store them.

The class implements also six **public** methods. Other than the three required by the text (notice that the `operator[]` is overloaded twice, because it has both a constant and a non constant version), we have `size` (lines 39 to 43), which returns the number of elements stored in the `Deque`, and `empty` (lines 45 to 49), which returns `true` if no elements are stored in the container.

The implementation of all these methods is described below.

```

1 #ifndef __DEQUE__
2 #define __DEQUE__
3
4 #include <vector>
5
6 namespace ds
7 {
8     template <typename T>
9     class Deque
10    {
11        typedef std::vector<std::vector<T>> container_type;
12
13    public:
14        typedef T value_type;
15        typedef typename container_type::size_type size_type;

```

```

16
17 private:
18     container_type chunks;
19
20     constexpr static size_type chunk_length = 8;
21     constexpr static size_type initial_chunk_number = 3;
22
23     size_type first_offset;
24     size_type off_end_offset;
25
26     inline bool
27     on_boundary (size_type offset) const
28     {
29         return offset % chunk_length == 0;
30     }
31
32     void
33     make_room_for (size_type);
34
35 public:
36     Deque (void)
37         : first_offset (0), off_end_offset (0) {}
38
39     inline size_type
40     size (void) const
41     {
42         return off_end_offset - first_offset;
43     }
44
45     inline bool
46     empty (void) const
47     {
48         return size () == 0;
49     }
50
51     value_type &
52     operator [] (size_type);
53     const value_type &
54     operator [] (size_type) const;
55
56     void
57     push_front (const value_type & z);
58     void
59     push_back (const value_type & z);
60 };
61
62 template <typename T>
63     void
64     Deque<T>::make_room_for (size_type additions)
65     {
66         const size_type next_size = size () + additions;
67         const size_type needed_chunks = next_size % chunk_length > 0

```

```

68     ? next_size / chunk_length + 1
69     : next_size / chunk_length;
70 size_type next_chunks = chunks.size ();
71
72 if (next_chunks == 0 and needed_chunks > 0)
73 {
74     next_chunks = initial_chunk_number;
75 }
76
77 if (first_offset == 0 or off_end_offset == chunks.size () * chunk_length)
78 {
79     next_chunks *= 2;
80 }
81
82 while (needed_chunks > next_chunks)
83 {
84     next_chunks *= 2;
85 }
86
87 if (next_chunks > chunks.size ())
88 {
89     const size_type center_before = chunks.size () / 2;
90     chunks.resize (next_chunks);
91     const size_type center_after = chunks.size () / 2;
92     const size_type shift = center_after - center_before;
93
94     for (size_type i = off_end_offset / chunk_length;
95          i < next_chunks and i >= first_offset / chunk_length;
96          --i)
97     {
98         using std::swap;
99         swap (chunks[i], chunks[i + shift]);
100    }
101
102 if (empty ())
103 {
104     first_offset = off_end_offset =
105         center_after * chunk_length + chunk_length / 2;
106     chunks[center_after].resize (chunk_length);
107 }
108 else
109 {
110     const size_type offset_shift = shift * chunk_length;
111     first_offset += offset_shift;
112     off_end_offset += offset_shift;
113 }
114 }
115 }
116
117 template <typename T>
118 typename Deque<T>::value_type &
119 Deque<T>::operator [] (size_type i)

```

```

120 {
121     const size_type real_idx = i + first_offset;
122     const size_type chunk_idx (real_idx / chunk_length),
123         position (real_idx % chunk_length);
124     return chunks[chunk_idx][position];
125 }
126
127 template <typename T>
128 const typename Deque<T>::value_type &
129 Deque<T>::operator [] (size_type i) const
130 {
131     const size_type real_idx = i + first_offset;
132     const size_type chunk_idx (real_idx / chunk_length),
133         position (real_idx % chunk_length);
134     return chunks[chunk_idx][position];
135 }
136
137 template <typename T>
138 void
139 Deque<T>::push_front (const value_type & element)
140 {
141     make_room_for (1);
142
143     if (on_boundary (first_offset --))
144     {
145         const size_type new_chunk = first_offset / chunk_length;
146         chunks[new_chunk].resize (chunk_length);
147     }
148
149     operator [] (0) = element;
150 }
151
152 template <typename T>
153 void
154 Deque<T>::push_back (const value_type & element)
155 {
156     make_room_for (1);
157
158     if (on_boundary (off_end_offset ++))
159     {
160         const size_type new_chunk = off_end_offset / chunk_length;
161         chunks[new_chunk].resize (chunk_length);
162     }
163
164     operator [] (size () - 1) = element;
165 }
166 }
167
168 #endif

```

The constructor of the class `Deque` (see lines 36 and 37) initializes to 0 both `first_offset` and `off_end_offset`. Notice that, for what concerns the initialization of the vector `chunks`, we rely on the default constructor of `std::vector`, meaning that `chunks` is empty. We will use

`initial_chunk_number` only to resize `chunks` when we call `push_back` or `push_front` for the first time.

The most important method is `make_room_for` (see the implementation in lines 62 to 115). It is called both in `push_front` and in `push_back` (see lines 141 and 156, respectively), passing as parameter 1, which is the number of elements we want to add in both cases.

The new size of the container, namely `next_size`, is computed as the current size plus the parameter `additions` (line 66). The index `needed_chunks` (line 67 to 69) represents the number of chunks we need to store all the elements (both the ones that are already in the container and the ones we want to add). The index `next_chunks` is initialized in line 70 with the number of chunks currently stored in the `Deque` and it is updated within the method to represent the new number of chunks we want to have in the container.

Since any object of type `Deque` is initialized as an empty container, at the first call of `push_front` or `push_back` the value of `chunks.size()` is equal to zero, while `needed_chunk` is greater than zero because we need space to store the new elements. Therefore, in lines 72 to 75, `next_chunks` becomes equal to the initial number of chunks we set for the container. On the other hand, if `first_offset` is equal to zero or `off_end_offset` is equal to the maximum number of elements we can store in the existing chunks (this is true, for example, at the beginning, when both `first_offset` and `off_end_offset` are zero), the value of `next_chunks` is doubled. The same happens as long as the number of needed chunks is greater than `next_chunks`.

In lines 87 to 114, we perform any modification of the container we possibly need to store the new elements. Notice that the main goal of the `Deque` structure is to avoid reallocations of the internal chunks, so that any pointer to an element in the `Deque` is never invalidated when we add new elements. The same does not happen with vectors, since when we call `push_back` we may incur in a reallocation of the entire data structure.

The structure of a `Deque` is designed so that the container is filled starting from the element in the middle of the central chunk. When, for example, the method `push_back` is called for the first time, `make_room_for(1)` is called. When the method arrives to line 87, `next_chunks` is equal to 6 (indeed, `next_size` is 1, `needed_chunks` is 1 and `next_chunks` becomes equal to 3 in line 74 and it is doubled in line 79 since both `first_offset` and `off_end_offset` are zero). In line 89, `center_before` is set to zero, while, in line 91, `center_after` is set to 3 after having resized the vector of chunks, so that, when we have 6 chunks, the fourth one is considered as central chunk. The first element to be filled, therefore, will be the central element of the fourth chunk.

Suppose now that the container is not empty, but that, for example, we are in the following situation:

```
chunk 0:  
  
chunk 1:  
  
chunk 2:  
  
chunk 3:  
    1  2  3  4  5  6  
  
chunk 4:  
    7  8  9  10 11 12 13 14  
  
chunk 5:  
    15 16 17 18 19 20 21 22
```

where the first three chunks are empty, the fourth has been partially filled and the last two

are completely full (therefore `first_offset` is equal to 26, which is the global index of the first existing element, and `off_end_offset` is equal to 48). If we run `push_back(25)`, a new element must be added at the end of the `Deque` and, in order to do this, we need to allocate new chunks. The method `make_room_for(1)` is called. When we reach line 87, `next_size` is equal to 23, `needed_chunks` is 3 and `next_chunks` is 12 (it is initialized to 6 in line 70 and it is doubled in line 79). This means that we want to double the number of chunks in the vector. Any time we resize the vector `chunks`, we do it in such a way that an equal number of chunks is inserted at the beginning and at the end of the vector `chunks`. This is achieved by the operations performed in lines 89 to 100. Indeed, `center_before` is set to 3 (the index of the current central chunk), while `center_after` is set to 6 (the index of the central chunk after the resize). The variable `shift`, in line 92, is set to 3. In lines 94 to 100, we loop over all the indices between `off_end_offset / chunk_length` (which in this case is 6) and `first_offset / chunk_length` (which in this case is 3). At every iteration, we swap the chunk stored in position `i` with the one stored in position `i + shift`, in such a way that, in the example we are considering, when we reach line 100 we have:

```
chunk 0:  
  
chunk 1:  
  
chunk 2:  
  
chunk 3:  
  
chunk 4:  
  
chunk 5:  
  
chunk 6:  
    1   2   3   4   5   6  
  
chunk 7:  
    7   8   9   10  11  12  13  14  
  
chunk 8:  
    15  16  17  18  19  20  21  22  
  
chunk 9:  
  
chunk 10:  
  
chunk 11:
```

i.e. three chunks are added at the beginning and three are added at the end of the vector `chunks`.

In lines 102 to 107, if the `Deque` is still empty, both `first_offset` and `off_end_offset` are set to the index of the element in the middle of the central chunk, which is resized by `chunk_length`. If the `Deque` is not empty, in turn, both `first_offset` and `off_end_offset` are shifted, so that, considering the example above, `first_offset` becomes equal to 50 and `off_end_offset` becomes equal to 72.

The `operator[]` (both in the non constant version, in lines 117 to 125, and in the constant version in lines 127 to 135) needs to compute, starting from the index `i` passed as parameter, the number of chunk and the position in which the required element is stored. In particular, in

order to compute those values, we need to know the position of the required element starting from `first_offset` (see for example line 131). Given this information, both `chunk_idx` and `position` are computed dividing `real_idx` by the length of a chunk and by considering once the quotient and once the remainder.

Both the methods `push_front` (lines 137 to 150) and `push_back` (lines 152 to 165) are implemented on top of `make_room_for` and on top of the `operator[]`. Notice that, in both the cases, we have to check whether either `first_offset` or `off_end_offset` are on the boundary of the current chunk and, if this is the case, we have to resize either the previous or the following chunk in order to be able to add the new element.

Complexity: The complexity of the `operator[]` is $O(1)$ because we rely on the corresponding `operator[]` of `std::vectors`, which supports random access.

On the other hand, both the methods `push_front` and `push_back` have a worst-case complexity of $O(N)$, where N is the number of elements already stored in the `Deque`. Indeed, since the internal size of the chunks is fixed, the computational cost of the resize we possibly perform in lines 146 and 161 is not considered when computing the overall complexity, which therefore coincides with the one of the method `make_room_for`. Thanks to the fact that, in `make_room_for`, we use `std::swap` instead of copying the elements from one chunk to the other, the computational cost of the method is given by the `chunks.resize` we perform in line 90, which is $O(N)$.

Deques are useful when adding elements to both sides of the container is frequently required (`std::vectors` are already efficient, but only when we need to increase the size from the end, since `push_back` has complexity $O(N)$ in the worst case, but $O(1)$ in the average case, while `push_front` has always complexity $O(N)$).

8.3 Frequently Asked Questions

1. **Template argument deduction:** Writing the code:

```

1 #include <iostream>
2
3 template <typename T> void fobj (T,T);
4 template <typename T> void fref (const T &, const T &);

5
6 int main (void)
7 {
8     int a[10], b[8];
9     fobj(a,b);
10    return 0;
11 }
12
13 template <typename T>
14 void fobj (T t1, T t2)
15 {
16     if (t1 != t2)
17         std::cout << "different" << std::endl;
18     else
19         std::cout << "equal" << std::endl;
20 }
21
22 template <typename T>
23 void fref (const T & t1, const T & t2)
24 {
```

```

25     if (t1 == t2)
26         std::cout << "different" << std::endl;
27     else
28         std::cout << "equal" << std::endl;
29 }
```

I get as output:

```
different
```

as expected, while, if in line 9 I try to call `fref` instead of `fobj`, I get the compiler error:

```

tempArgDed.cpp:10:2: error: no matching function for call to 'fref'
fref(a,b);

tempArgDed.cpp:5:28: note: candidate template ignored: deduced conflicting
      types
for parameter 'T' ('int [10]' vs. 'int [8]')
template <typename T> void fref (const T &, const T &);
```

Why?

ANSWER: An array's type consists in both the contained type and the size, hence `a` is of type `int[10]` and `b` is of type `int[8]`. This means that `fref` deduces two different types for `T`, which makes it ambiguous, so the template instantiation fails.

Instead, `fobj` is not offending because, when the function parameter is not a reference, the standard allows also array-to-pointer conversions even in template argument deduction. Since `fobj` receives its arguments by copy, both `a` and `b` appear as `int*` in the function call, so the template instantiation succeeds.

2. In a situation like the one we have in Exercise 5, where we have to use as key of an unordered container something that is not a built-in type, is there any difference between specializing `std::hash` by something as:

```

1 namespace std
2 {
3     template <>
4     struct hash<SocialNetworkNS::User> : public unary_function<SocialNetworkNS
5         ::User, size_t>
6     {
7         size_t operator()(const SocialNetworkNS::User & user) const
8         {
9             return std::hash<std::string>{}(user.CGetName() + user.CGetSurname());
10        }
11    };
}
```

as we did in the exercise and implementing a user-defined hasher as it is written below?

```

1 struct hasher
2 {
3     size_t operator() (const SocialNetworkNS::User & user) const
4     {
5         return std::hash<std::string>{}(user.CGetName() + user.CGetSurname());
6     }
7 };
```

ANSWER: The solutions are functionally equivalent, in the sense that you can choose to go with one or the other, as you prefer. However, notice that in the first case you can instantiate the unordered container simply writing, for example:

```
std::unordered_set<User> set_of_users;
```

(see what happens in Exercise 5), because the second **template** parameter has as default value `std::hash<K>`, where `K` is the type of your key.

In the second case, in turn, you have to pass explicitly your hasher as a **template** parameter of the unordered container, writing:

```
std::unordered_set<User,hasher> set_of_users;
```