POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Labs of

NLA

taught by Professor Paola Antonietti

Master Degree in Mathematical Engineering, Polimi, 2024/25

By

Teo Bonfa

Contents

Chapter 1

Classes-1

In C++ we use classes to define our own **abstract data types** (ADT). By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

1.1 Classes basics

- Classes are user-defined type, i.e. specifies a blueprint for objects.

> ! Most fundamentally, a class defines a new type and a new scope.

- Each class consists of a set of members. Members can be either data (**data members**), functions (**member functions** or **methods**), or type definitions.
- Member functions can define the meaning of creation (constructor), initialization, assignment, copy, and cleanup (destruction).
- A class may contain multiple `public`, `private`, and `protected` sections. Members defined in the `public` section are accessible to all code that uses the type; those defined in the `private` section are accessible (only) to other class members. We'll have more to say about `protected` when we discuss inheritance.

> ! The public members provide the class **interface** and the private members provide **implementation details** that we want to hide.

- First example:

```
class X { // this class' name is X

private: // private members <-- that's the implementation details
    // (accessible by members of this class only)
    // functions
    // types
    // data

public: // public members <-- that's the interface to users
    // (accessible by all)
    // functions
    // types
    // data (often best kept private!)
};
```

- Members are accessed using . (dot) for objects and -> (arrow) or (*). (dereference plus dot) for pointers:

```
// in a header file
class Complex {
private:
    double m_real;
    double m_img;
    int mf1();
```

```
public:
    int mf2();
};

// in the main
Complex num; // object (variable) num of type Complex
Complex *ptr = &num; // pointer of type Complex to num

double re = num.m_real; // ERROR: m_real is private
int i = num.mf1(); // ERROR: m_real is private
int j = ptr->mf2(); // good
int k = (*ptr).mf2(); // same as previous
```

> ! Class members are private by default. So in the example you might omit the `private` key.

▷ Also operators, such as +, !, and [ ], can be defined as well (e.g. what is the meaning of + between two object from the class `Complex`).

1.2 Classes and Struct

A struct is a class where members are public by default. So

```
struct X {
    int m;
};
```

is equivalent to

```
class X {
public:
    int m;
};
```

But then:

- which are the benefits of classes (i.e. public/private)?
- when shall we use a class and when a struct?

Let's answer.

▷ We do not make everything public
  - to provide a clean interface (data and messy functions can me hidden)
  - to allow a change of representation (we'll se in the complex numbers example)

  > ! If internal representation is hidden (**information hiding** principle):
  > – it's easier to support code evolution
  > – we can change the internals without changing the remaining code

  - to make sure that sensitive members are not used/copied/modified by anyone
  - to ease debugging
  - to maintain an **invariant**

▷ What are invariants? Let's consider the `Date` example.

To verify if the value of a date is valid (e.g. the month cannot be 13) we have to check for validity all the time, or we try to design our types so that values are guaranteed to be valid.

A rule for what constitutes a valid value is called an **invariant**.

The invariant for dates ("a date must represent a date in the past, present, or future") is usually hard to state precisely because of February 29th, leap years (*anni bisestili*), etc.

> ! If we can't thing of a good invariant, we can probably use struct.

Let's implement `Date` using struct:

```
struct Date {
    int day, month, year;
}

int main() {
    Date my_birthday;
    my_birthday.day = 15;
    my_birthday.month = 5;
    my_birthday.year = 2000;
}
```

We have to add a few helper functions for convenience:

```
void init_date(Date& date, int day, int month, int year) {
    // check for valid date and initialize ...
}

void add_day(Date& date, int n) {
    // increase date by n days ...
}
```

Admitting for a moment a trivial invariant, we have this:

```
#include <iostream>

struct Date {
    int day, month, year;
};

void init_date(Date& date, int day, int month, int year);
void add_day(Date& date, int n);

int main() {
    Date my_birthday;
    init_date(my_birthday, 15, 5, 2000);
    std::cout << my_birthday.day << "/" << my_birthday.month
              << "/" << my_birthday.year << "\n";

    Date your_birthday = my_birthday;
    add_day(your_birthday, 11);
    std::cout << your_birthday.day << "/" << your_birthday.month
              << "/" << your_birthday.year << "\n";

    return 0;
}

void init_date(Date& date, int day, int month, int year) {
    // check for valid date and initialize ...
    if (day<=31 && month<=12) {
        date.day = day;
        date.month = month;
        date.year = year;
    }
}

void add_day(Date& date, int n) {
    // increase date by n days ...
    init_date(date, date.day + n, date.month, date.year);
}
```

The output is the following: `15/05/2000` and `26/05/2000`.

▷ There is a better way to do this, using a **constructor**:

```
#include <iostream>

struct Date {
    int day, month, year;
    Date (int d, int m, int y) {
        if (d<=31 && m<=12) {
            day = d;
            month = m;
            year = y;
        }
    };
};

int main() {
    Date my_birthday(15, 5, 2000);
    std::cout << my_birthday.day << "/" << my_birthday.month
              << "/" << my_birthday.year << "\n";

    return 0;
}
```

> **!** When we create an object of a class type, the compiler automatically uses a constructor to initialize the object. A constructor is a special member function that has the same name as the class. Its purpose is to ensure that each data member is set to sensible initial values.

Moving on to classes, we can prevent someone from manually editing, for example, the month of a date, making it invalid.

Let's create a class in CLion:

(a) create a new project → right click on the folder name → new → C++ class

(b) insert the name of the class, our is `Date`

(c) the `Date.h` file will open

(d) write this:

```
#ifndef MYDATECLASS_DATE_H
#define MYDATECLASS_DATE_H

class Date {
    int d, m, y; // private by default
public:
    Date(int d, int m, int y);
    void add_day(int n);
    int month() const;
};

#endif //MYDATECLASS_DATE_H
```

(e) right click on the file → generate → generate definitions

(f) hold Shift and click on every members → ok → the `Date.cpp` file will open:

```
#include "Date.h"

Date::Date(int d, int m, int y) {

}

void Date::add_day(int n) {

}

int Date::month() const {
    return 0;
}
```

(g) now go to `main.cpp`, add `#include "Date.h"` and you can write your own main

For example, you can write `Date birthday(15,5,2000);` and the compiler will create the object `birthday` of type `Date` intialized at 15/5/2000.

Some important things:

- Now you can't write `std::cout << birthday.day << "\n";` because the member `day` is private!

  > **!** To print values of private members you have to use a print function within the class. Why? Because private members are accessible by other members of the same class! You can't access them directly, you have to go through the class!

- A constructor gets called automatically when we create the object of the class.
- There are plenty of ways to create a constructor. In this example, the constructor is

```
Date::Date(int d, int m, int y) {

}
```

  This is the *default style*, because if you don't declare one, the compiler will automatically create a constructor like that.

- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class, though they can be declared in the private section too.
- Constructors do not return values; hence **they do not have a return type**.
- Constructors can be defined inside or outside the class declaration. In this case, we have declared the constructor in a header but we have defined it in a source file. So we needed to use the **scope operator** :: ("member of") to clarify that the constructor is the constructor of the `Date` class.
- Member functions may be declared `const` (`int Date::month() const`). A `const` member may not change the data members of the object on which it operates.

▷ Another example:

```
class Line {
public:
    void setLength(double len); // <-- setter
    double getLength(); // <-- getter
    Line(double len); // <-- constructor
private:
    double length;
};
// member functions definition
Line::Line(double len) {
    cout<<"Object is being created, length = "<< len <<endl;
    length = len;
}
void Line::setLength(double len) {
    length = len;
}
double Line::getLength() {
    return length;
}
// main function for the program
int main() {
    Line line(10.0);
    std::cout<<"Length of line : " << line.getLength() << std::endl;
    line.setLength(6.0);
    std::cout<<"Length of line : " << line.getLength() << std::endl;
}
```

The output is:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

▷ Write a class named Person that represent the name and address of a person. Provide a constructor, and some getters and setters.

```cpp
// Person.h
#ifndef PERSON_PERSON_H
#define PERSON_PERSON_H

#include <string>

class Person {
    std::string name;
    std::string address;
public:
    // Constructor
    Person(std::string, std::string);
    // Setter
    void set_name(std::string);
    void set_address(std::string);
    // Getter
    std::string get_name() const;
    std::string get_address() const;
    // Print
    void print() const;
};

#endif //PERSON_PERSON_H
```

```cpp
// Person.cpp
#include "Person.h"
#include <iostream>

Person::Person(std::string Name, std::string Address) {
    name = Name;
    address = Address;
}

void Person::set_name(std::string Name) {
    name = Name;
}

void Person::set_address(std::string Address) {
    address = Address;
}

std::string Person::get_name() const {
    return name;
}

std::string Person::get_address() const {
    return address;
}

void Person::print() const {
    std::cout << "Person is " << name << ", at " << address << std::endl;
}
```

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

int main() {
    Person you("Lorenzo", "Milano");
    you.print();
    return 0;
}
```

1.3 The implicit this pointer

Member functions have an extra implicit parameter that is a **pointer to an object of the class type**. This implicit parameter is named `this`, and it is bound to the object on which the member function is called.

So member functions access the object on which they were called through the `this` pointer.

> **!** Member functions may not define the `this` parameter, the compiler does so implicitly.
> The body of a member function may explicitly use the `this` pointer, but is not required to do so; there are only two exception: for `static` members and when we need to refer to the object as a whole rather than to a member of the object.

Take a look at this:

```
Date my_birthday(15, 5, 2000);
int may = my_birthday.m();
```

Here we used the dot operator to run `month()` on the object named `my_birthday`. So the method `month()` is referring implicitly to the members of the object on which the function was called.

When we call a member function, **`this` is initialized with the address of the object on which the function was invoked**. For example, when we call `my_birthday.m()` the compiler passes the address of `my_birthday` to the implicit `this` parameter; then the method `month()` return `this->m`.

```
int may = my_birthday.month();
```

- create a pointer to `my_birthday`:
  `Date *this = &my_birthday`
- call the `month` member passing the address of `my_birthday`
- when `month` return `m`, it is implicitly returning `this->m`

This is how it works, but in reality we do not do that: the `this` parameter is defined for us implicitly.

> **!**
> - It is illegal for us to define a parameter or variable named `this`
> - `this` is a **const pointer**, we cannot change the address that it holds.

`PlayWithThis` example

Let's create a trivial class, with a constructor and a getter:

```
// X.h
class X {
private:
    int x;
public:
    X(int x); // <-- constructor
    int get_x() const; // <-- getter
};
```

```
// X.cpp
#include "X.h"

X::X(int x) {
    x = x;
}

int X::get_x() const {
    return x;
```

```
}
```

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

int main() {
    X obj(3);
    std::cout << obj.get_x() << std::endl;
    return 0;
}
```

Rather than the expected 3 output, we get a different value every time we run the code. This occurrence of random values is typical in situations where the variable is not initialized.

So where's the mistake here?

Surely, it's in the constructor. Because of **variable scope**, when we said `x=x` we were actually doing a self-assignment with the function input `x`, we're not touching the class member `x`.

To fix the error:

```
X::X(int x) {
    this->x = x;
}
```

because with `this` we are able to access the members of the class we're dealing.

In real life we simply use different names:

```
X::X(int x) {
    x = xx;
}
```

1.4 Helper functions

What makes a good **class interface** (the set of public functions)?

- Minimal (as small as possible)
- Complete (and no smaller)
- Invariant preserving
- Const correct

We keep a class interface minimal because it

> simplifies understanding
> simplifies debugging
> simplifies maintenance

but we need extra (non-member) "helper functions" outside the class, for example `next_weekday` or the opeators `==` and `!=` to compare two dates.

!
- Declare helper functions in the class header
- Define helper functions the class source file

1.5 Operator overloading

- C++ let us redefine the meaning of the operators when applied to objects of class type.
- Judicious use of **operator overloading** can make class types as intuitive to use as the built-in types.
- For example, the standard library defines several overloaded operators for the container classes. These classes define the subscript operator [ ] to access data elements and * and -> to dereference container iterators. The fact that these library types have the same operators makes using them similar to using built-in arrays and pointers. Allowing programs to use expressions rather than named functions can make the programs much easier to write and read.
- Overloaded operators are functions with special names: the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list: `Complex operator+(const Complex&, const Complex&);`
- There are some rules and some advices to follow:
  (1) An overloaded operator has the same number of parameters as the operator has operands (e.g., no `operator<=` with only one parameter (unary), or `operator!` (not) with two parameters (binary)), except for the function-call operator: `operator()`

  (2) There are operators that may be and may not be overloaded:

Table 14.1. Overloadable Operators

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr>
<tr><td>&</td><td>|</td><td>~</td><td>!</td><td>,</td><td>=</td></tr>
<tr><td><</td><td>></td><td><=</td><td>>=</td><td>++</td><td>--</td></tr>
<tr><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td></tr>
<tr><td>|=</td><td>*=</td><td><<=</td><td>>>=</td><td>[]</td><td>()</td></tr>
<tr><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td></tr>
</table>

Table 14.2. Operators That Cannot Be Overloaded

::	.*	.	?:

  (3) New operators may not be created by concatenating other legal symbols (e.g., it would be illegal to attempt to define an `operator**` to provide exponentiation)

> ! You can define only existing operators, you can not create new ones.

  (4) The meaning of an operator for the built-in types may not be changed (e.g., the built-in integer addition operation cannot be redefined)

  Nor may additional operators be defined for the built-in data types (e.g., an `operator+` taking two operands of array types cannot be defined)

> ! An overloaded operator must have at least one user-defined type as operand. This rule enforces the requirement that an overloaded operator may not redefine the meaning of the operators when applied to objects of built-in type.

  (5) The precedence, associativity, or number of operands of an operator cannot be changed (e.g., regardless of the type of the operands and regardless of the definition of what the operations do, the expression `x == y+z` always binds the arguments `y` and `z` to `operator+` and uses that result as the right-hand operand to `operator==`)

  (6) Four symbols (+, -, * , &) serve as both unary (with one operand) and binary (with two operands) operators, which operator is being defined is controlled by the number of operands

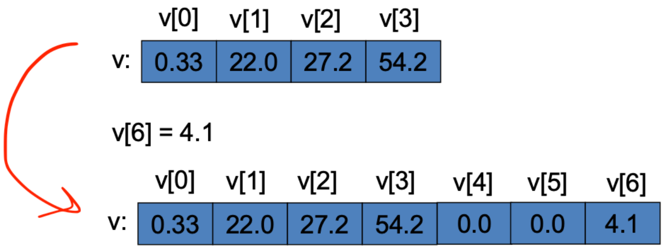
  (7) Overload operators only with their conventional meaning (e.g., + should be addition)

  (8) Don't overload &&, ||, and the comma operator (because the order in which those operands are evaluated is not stipulated or defined)

(9) **Don't overload unless you really have to**

▷ Class example: `MatlabVector`

We want to implement Matlab-like vectors in C++, so vectors can grow as in Matlab



but we want to keep the C++ convention for elements indexing (i.e. the first element has index 0).

So our goals are:

- provide `operator+`
- implement the product of `MatlabVector` with a scalar: `operator*`
- provide `operator[]` to access individual elements
- neglect, in the beginning, errors (e.g., vectors size do not match)

Our first implementation is the following:

```
class MatlabVector {

    std::vector<double> elem;

public:

    void set(size_t n, double v); // access: write
    double get(size_t n); // access: read

    size_t size() const; // return number of elements

    void print() const;

    MatlabVector operator*(double scalar) const;
};

MatlabVector operator+(MatlabVector& v1, MatlabVector& v2);
```

Note that:

- the getter is not const, beacuse if the index is out of dimension we want the automatically growth of the vector
- the `operator*` is const, because we're not changing the object which we're computing on
- the `operator+` is not in the class, it's a helper function, and so it uses references to access the members of the class (and the references are not const because the getter is not const!)

Let's see the functions in details:

```
void MatlabVector::set(size_t n, double v)
{
    while (elem.size() < n+1)
        elem.push_back(0.);

    elem[n] = v;
}

double MatlabVector::get(size_t n)
{
    while (elem.size() < n+1)
        elem.push_back(0.);

    return elem[n];
```

```
}

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

MatlabVector MatlabVector::operator*(double scalar) const
{
    MatlabVector result;

    for (unsigned i=0; i<elem.size(); ++i)
        result.set(i, scalar * elem[i]);

    return result;
}

MatlabVector operator+(MatlabVector& v1, MatlabVector& v2)
{
    MatlabVector result;

    for (unsigned i=0; i<v1.size(); ++i)
        result.set(i, v1.get(i) + v2.get(i));

    return result;
}
```

! Thanks to the loop

```
while (elem.size() < n+1)
    elem.push_back(0.);
```

we're able to automatically grow the vector.

Now we can test it.

(i) Do vectors grow automatically? Yes:

```
MatlabVector v;

v.set(0, 1);
v.set(1, 3);
cout << "v content: [ ";
v.print();
cout << "]\n\n";

v.set(3,4);
cout << "v content: [ ";
v.print();
cout << "]\n\n";
```

```
v content:  [ 1 3 ]
v content:  [ 1 3 0 4 ]
```

```
double d1 = v.get(1);
cout << "d1: " << d1 << endl;

cout << "v content after gettin d1: [ ";
v.print();
cout << "]\n\n";

double d2 = v.get(6);
cout << "d2: " << d2 << endl;

cout << "v content after gettin d2: [ ";
v.print();
```

```
cout << "]\n\n";
```

```
                     d1:  3
v content after gettin d1:  [ 1 3 0 4 ]
                     d2:  0
v content after gettin d2:  [ 1 3 0 4 0 0 0 ]
```

(ii) What happens in the sum between vectors if the dimensions are different?

```
MatlabVector v1;

v1.set(0, 1);
v1.set(1, 3);
v1.set(3,4);
cout << "v1 content: [ ";
v1.print();
cout << "]\n\n";

MatlabVector v2;

v2.set(0, 4);
v2.set(1, 2);
cout << "v2 content: [ ";
v2.print();
cout << "]\n\n";

MatlabVector sum1, sum2;

sum1 = v2 + v1; // smallest + biggest
cout << "v2 + v1 content: [ ";
sum1.print();
cout << "]\n\n";

sum2 = v1 + v2; // biggest + smallest
cout << "v1 + v2 content: [ ";
sum2.print();
cout << "]\n\n";
```

```
     v1 content:  [ 1 3 0 4 ]
     v2 content:  [ 4 2 ]
v2 + v1 content:  [ 5 5 ]
v1 + v2 content:  [ 5 5 0 4 ]
```

- if the first addendum is smaller, we neglect the *extra* terms of the bigger addendum, and the sum has the same size as the smallest one;
- if the first addendum is bigger, the second automatically grow to match the size of the first and then the sum is computed.

Remember: the sum always has the size of the first addendum $\Longrightarrow$ the sum is'nt commutative.

(iii) Is the scalar product commutative? No:

```
MatlabVector v3 = v1 * 3;
// v3 print ...

MatlabVector v33 = 3 * v1;
// v33 print ...
```

The first print is

```
v3 content:  [ 3 9 0 12 ]
```

but of course if we try to run the second one CLion would be mad. This is because

`v3 = v1 * 3`   actually is   `v3 = v1.operator*(3)` instead, `v33 = 3.operator*(v1)` is not defined thus `v33 = 3 * v1` uses the standard C++ operator *, not the one we defined.

Conclusion: neither the addition nor the the scalar product is commutative.

(iv) How do we create a vector of ordered numbers?

```
MatlabVector v0;

cout << "v0 content: [ ";
for (unsigned i = 0; i < 10; ++i) {
    v0.set(i,i);
    cout << v0.get(i) << " ";
}
cout << "]\n\n";
```

```
v0 content:  [ 0 1 2 3 4 5 6 7 8 9 ]
```

Look: this is pretty ugly! I'd prefer something like this:

```
MatlabVector v0;

cout << "v0 content: [ ";
for (unsigned i = 0; i < 10; ++i) {
    v0[i] = i;
    cout << v0[i] << " ";
}
cout << "]\n\n";
```

How we improve our implementation?

First of all, we define the subscript operator to speed up access to vectors:

```
// in MatlabVector.h, under the public part
double & operator[](size_t n);

// in MatlabVector.cpp
double & MatlabVector::operator[](size_t  n)
{
    while (elem.size() < n+1)
        elem.push_back(0.);

    return elem[n];
}
```

This is similar to the implementation of the getter and the setter, however here we're using **references**!! Now `return elem[n]` is not returning a copy of `elem[n]`, but a reference to the real `elem[n]`.

! Thanks to the subscript operator, we don't need anymore the getter and the setter.

Moreover, we make the product commutative by adding the following helper function:

```
// in MatlabVector.h, outside the class
MatlabVector operator*(double scalar, const MatlabVector v);

// in MatlabVector.cpp
MatlabVector operator*(double scalar, const MatlabVector v)
{
    return v * scalar; // or v.operator*(scalar)
}
```

In the end, this is the new header

```
class MatlabVector {

    std::vector<double> elem;

public:
```

```
    double & operator[](size_t n);

    size_t size() const;

    void print() const;

    MatlabVector operator*(double scalar) const;
};

MatlabVector operator+(MatlabVector& v1, MatlabVector& v2);

MatlabVector operator*(double scalar, const MatlabVector v);
```

and this the new source file

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

MatlabVector MatlabVector::operator*(double scalar) const
{
    MatlabVector result;

    for (unsigned i=0; i<elem.size(); ++i)
        result[i] =scalar * elem[i];

    return result;
}

MatlabVector operator+(MatlabVector& v1, MatlabVector& v2)
{
    MatlabVector result;

    for (unsigned i=0; i<v1.size(); ++i)
        result[i] = v1[i] + v2[i];

    return result;
}

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

    return elem[n];
}

MatlabVector operator*(double scalar, const MatlabVector v)
{
    return v*scalar; // cioè v.operator*(scalar)
}
```

Thanks to these, the main in really Matlab-like:

```
// 1: create
MatlabVector v;
v[0] = 1;
v[1] = 3;
v[3] = 4;
cout << "v content: [ ";
v.print();
cout << "]\n\n";

// 2: extract and automatic growth
double d1 = v[6];
cout << "d1 = v[6] = " << d1 << endl;
cout << "v content after gettin d1: [ ";
v.print();
```

```
cout << "]\n\n";

// 3: ordered vector
MatlabVector v0;
cout << "v0 content: [ ";
for (unsigned i=0; i<10; ++i) { // Not ugly anymore!
    v0[i] = i;
    cout << v0[i] << " ";
}
cout << "]\n\n";

// 4: commutative product
MatlabVector v1 = v0*3;
cout << "v1 content: [ ";
v1.print();
cout << "]\n\n";

MatlabVector v2 = 3*v0;
cout << "v2 content: [ ";
v2.print();
cout << "]\n\n";
```

```
// 1: create
v content: [ 1 3 0 4 ]

// 2: extract and automatic growth
d1 = v[6] = 0
v content after gettin d1: [ 1 3 0 4 0 0 0 ]

// 3: ordered vector
v0 content: [ 0 1 2 3 4 5 6 7 8 9 ]

// 4: commutative product
v1 content: [ 0 3 6 9 12 15 18 21 24 27 ]
v2 content: [ 0 3 6 9 12 15 18 21 24 27 ]
```

▷ Now we aim to define the operators more thoroughly, whether they are inside or outside the class.

In the `MatlabVector` class there is only one `MatlabVector` and so:

- scalar product is between one `MatlabVector` and one scalar ⇝ defined within the class
- sum is between two `MatlabVector` ⇝ defined outside, as an helper function

But there is a way to implement the `operator+` inside the class: since the left hand side (lhs) operand is bounded to `this`, we can define the operation just with one input parameter:

```
class MatlabVector {

    std::vector<double> elem;

public:

    double & operator[](size_t n);

    size_t size() const;

    void print() const;

    MatlabVector operator*(double scalar) const;

    MatlabVector operator+(const MatlabVector& rhs) const;
};

MatlabVector operator*(double scalar, const MatlabVector v);
```

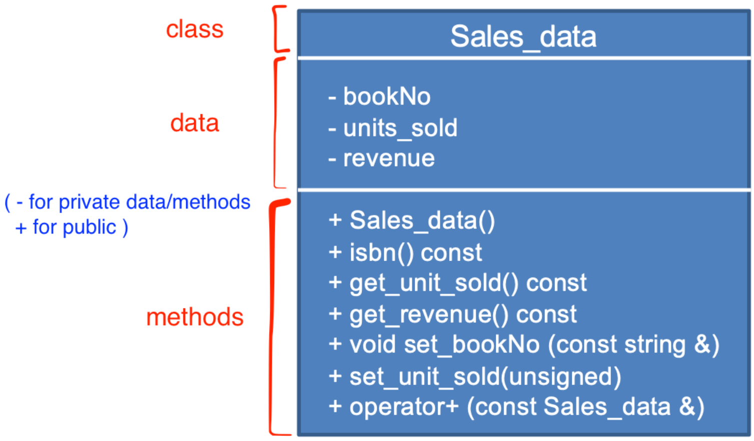
```
MatlabVector MatlabVector::operator+(const MatlabVector &rhs) const
{
    MatlabVector result;
```

```
    for (unsigned i=0; i<elem.size(); ++i)
        result[i] = elem[i] + rhs.elem[i];

    return result;
}
```

▷ Class example: `Sales_data`

We want to manage book sales (when discussing inheritance, we'll take this example more deeply). Let's see the **class diagram**:



Here there is the header:

```
class Sales_data {

    string bookNo;
    unsigned units_sold;
    double revenue;

public:

    // Constructor
    Sales_data() :
        bookNo(""),
        units_sold(0),
        revenue(0.0)
    {}

    // Getters and setters
    string isbn() const;
    unsigned get_unit_sold() const;
    double get_revenue () const;
    void set_bookNo (const string & bn);
    void set_unit_sold(unsigned u);
    void set_revenue (double r);
    Sales_data operator+(const Sales_data &rhs) const;
}
```

The in-class implementation of `operator+` is the following:

```
Sales_data Sales_data::operator+(const & Sales_data rhs) const {
    Sales_data ret;
    ret.bookNo = bookNo;
    ret.units_sold = units_sold + rhs.units_sold;
    ret.revenue = revenue + rhs.revenue;
    return ret;
}
```

Instead, if the operation was an helper function:

```
Sales_data operator+(const Sales_data & lhs, const Sales_data & rhs) {
    Sales_data ret;
```

```
    ret.set_bookNo(lhs.isbn());
    ret.set_units_sold(lhs.get_units_sold() + rhs.get_units_sold());
    ret.set_revenue(lhs.get_revenue() + rhs.get_revenue());
    return ret;
}
```

▷ To wrap up:

- Must be member
  `=  []  ()  ->`
  (function call)
- Should be member
  - Compound assignments `+=  -=  /=  %=  %=  ^=  &=  |=  *=  <<=  >>=`
  - Modify operators `++  --  *`
- Better non-member
  - Arithmetic operators `+  -  *  %`
  - Bitwise operators `^  &  |`
  - Equality operators `<  >  <=  =>  !=  ==`
  - Relational operators `!  &&  ||`

1.6 Defining a function to return "this" object

In the last example, we defined

```
Sales_data operator+(const Sales_data & lhs, const Sales_data & rhs);
```

to sum two sales. It is possibile to add the following *combined* operator `+=` (with integer, the one that make `x=x+3` equal to `x+=3`):

```
Sales_data & operator+=(const Sales_data & rhs);
```

With `operator+=` we're returning a reference to a `Sales_data` object.

In this way, in the main we can say

```
Sales_data s1, s2;
s1 += s2; // or s1.operator+=(s2);
```

instead of

```
Sales_data s1, s2;
s1 = s1 + s2; // or s1.operator+(s1, s2);
```

Let's see the implementation of `operator+=`:

```
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;   // add the members of rhs
    revenue += rhs.revenue;         // into the members of "this" object

    return *this; // return the object on which the function was called
}
```

Comments:

- the operator is defined within the class (is not an helper function), so when we say

  `units_sold` or `revenue`

  we're accessing the private part of the class (we're within the class, so this is possible), in particular the private part of "this" object (and twe do not need to write `this->units_sold`) however, we do need to use this to access the object as a whole:

```
return *this;     // return the object on which the function was called
```

Here the return statement dereferences this to obtain the object on which the operator is executing

- the +='s in the function body are the default += (like x+=3)
- it is not mandatory to overload the +=, you can use (as Lippman does):

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;   // add the members of rhs
    revenue += rhs.revenue;         // into the members of "this" object

    return *this; // return the object on which the function was called
}
```

This is what tipically happens in real life:

```
Sales_data trans;
/* modify trans */

Sales_data total;
/* modify total */

total += trans; // update the running total
```

- the address of total is bound to the implicit this parameter and rhs is bound to trans
- Thus, when += executes:
  units_sold += rhs.units_sold;
- the effect is to add total.units_sold and trans.units_sold, storing the result back into total.units_sold
- the same happens for revenues

We remark that:

operator=, operator+=, etc, must return a referece, not a copy!

Why? Because with user-defined types (operators) we want to mimic built-in types (operators).

Let's stress a little bit more this concept.

▷ The standard = operator (assignment):

```
int a=0, b1, c=2;
cout << "a=" << a << " b=" << b << " c=" << c << endl;

a = b = c;
cout << "a=" << a << " b=" << b << " c=" << c << endl;
```

```
a=0 b=1 c=2
a=2 b=2 c=2
```

```
int a=0, b1, c=2;
cout << "a=" << a << " b=" << b << " c=" << c << endl;

( a = b ) = c;
cout << "a=" << a << " b=" << b << " c=" << c << endl;
```

```
a=0 b=1 c=2
a=2 b=1 c=2
```

In the first case: a=2 b=2 c=2, because a=b=c; means b=c; a=b;

Instead, in the second case: a=2 b=1 c=2 i.e. b remains with his old value, because (a=b)=c; means a=b; a=c; and the assignment b=c is not performed

▷ Now we want to mimic the = operator with copy:

```
class BaseKO{
    int x;
public:
    BaseKO() { x = -1; };
    BaseKO(int val) : x(val) {};
    BaseKO operator=(const BaseKO& rhs); // COPY
    int get_x() const;
    void set_x(int val);
};
```

```
int BaseKO::get_x() const{
    return x;
}
void BaseKO::set_x(int val){
   x = val;
}
BaseKO BaseKO::operator=(const BaseKO& rhs) { // COPY
    x = rhs.x;
    return *this;
}
```

```
BaseKO a_(0); BaseKO b_(1); BaseKO c_(2);
a_ = b_ = c_;

a_.set_x(0); b_.set_x(1); c_.set_x(2);
(a_ = b_) = c_;
```

```
a_=2 b_=2 c_=2
a_=1 b_=1 c_=2
```

and this makes us sad: `a_=b_=c_;` is okay, but `(a_=b_)=c_;` does not mimic the standard behaviour.

▷ So the only way to mimic the = operator is using reference:

```
class BaseOK{
    int x;
public:
    BaseOK() { x = -1; };
    BaseOK(int val) : x(val) {};
    BaseOK & operator=(const BaseOK& rhs); // REFERENCE
    int get_x() const;
    void set_x(int val);
};
```

```
int BaseOK::get_x() const{
    return x;
}
void BaseOK::set_x(int val){
   x = val;
}
BaseOK & BaseOK::operator=(const BaseOK& rhs) { // REFERENCE
    x = rhs.x;
    return *this;
}
```

```
BaseOK a__(0); BaseOK b__(1); BaseOK c__(2);
a__ = b__ = c__;

a__.set_x(0); b__.set_x(1); c__.set_x(2);
(a__ = b__) = c__;
```

```
a__=2 b__=2 c__=2
a__=1 b__=1 c__=2
```

and now the behavior is coherent.

Chapter 2

Inheritance - Overview

2.1 Basics

- A program is an object oriented program (OOP) if it provides the following (PIE) properties:

  $\boxed{\text{Polymorphism}}$ $\boxed{\text{Inheritance}}$ $\boxed{\text{Encapsulation}}$

- Encapsulation: binds the data & functions in one *Class*; by thinking the system as composed of independent objects, we keep sub-parts really independent and this:
  → allows different groups of programmers to work on different parts of the project
  → allows extreme modularity
  → increases code-reuse
- Inheritance: provides a way to create a new class from an existing class

base class	→	derived class
or super class		or sub class
or father class		or child class
or parent class		

Example:

High level (more general)	Father class: Insect
Low level (more specific)	↪ Child class: Bumble Bee ↪ Child class: Grasshopper

> !
> To recognize the relationship between father and child class you can use this trick:
>
> a `DerivedClassName` is a `BaseClassName` with ...
>
> and then enumerate the special characteristic of that subclass.
>
> E.g. A triangle is a polygon with three edges

Motivations/Advantages of Inheritance:

→ **reuse** the superclass members
(this point expresses code-reuse)

→ **extend** the superclass by adding new members

→ **specialize** the superclass by changing implementation without changing the interface (e.g., by overloading its methods with your own implememntations)
(these last two point express code-evolution)

2.2 Inheritance

- Inheritance is a mean of specifying hierarchical relationships between types
- C++ classes
  - → inherit both data and function members (although such members may not always be accessible in the derived class, just wait!)
  - ← does not inherit the base class constructors, destructor, assignment operator and friends (because these functions are class-specific)
- Syntax:

```
class DerivedClassName : access-level BaseClassName { // body... }
```

  The access-level specifies the type of derivation i.e. **the inheritance type**:
  - private (by default)
  - public
  - protected

    ⚠ we will always use public inheritance (because we want to inherit the interface)
- Example:

```
// base class
class Point
{
    protected:
        float x;
        float y;
    public:
        void set_coord(float xx, float yy);
};

// derived class
class 3dPoint: public Point
{
    private:
        float z;
    public:
        void set_coord(float xx, float yy, float zz);
}
```

  Remark: here there is the overloading of the method `set_cord`

2.3 Protected members and Class access

In a base class, the `public` and `private` labels have their ordinary meanings:

- ▷ `public` members are accessibile anywhere (outside the class and also from objects)
- ▷ `private` members are accessible only in the class itself

A derived class has the same access as any other part of the program to the `public` and `private` members of its base class: it may access the `public` members and has no access to the `private` members.

Sometimes a class used as a base class has members that it wants to allow its derived classes to access, while still prohibiting access to those same members by other users. The `protected` access label is used for such members:

- ▷ `protected` members are accessibile in the class and in subclasses

In the example above: our `Point` based class expects its derived class `3dPoint` to redefine the `set_coord` function; to do so, this class will need access to the `x` and `y` members. So those members have to be `protected`. (If I wanted to create a `4dPoint` class by taking `3DPoint` as the base class then I would have to declare the `z` coordinate `protected` too.)

The protected access label can be thought of as a blend of `private` and `public`:

- ▷ Like `private` members, `protected` members are inaccessible to users of the class.
- ▷ Like `public` members, the `protected` members are accessible to classes derived from this class.

In addition, `protected` has another important property:

- ▷ A derived object may access the `protected` members of its base class only through a derived object. The derived class has no special access to the `protected` members of base type objects.

Dataframe (exam 7/7/17 - exercise 13)

A `DataFrame` is a 2-dimensional labeled data structure with columns of the same type. You can think of it like a spreadsheet or a table. 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 1 shows, as an example, a `DataFrame` including two columns storing temperatures and humidity for a weather forecast application.

You have to provide the definition of the `DataFrame` class, storing elements of type `double` and optime your choice for the **worst case complexity** under the assumption your data structure have a very large number of columns. You can assume that the columns are dense. In particular, you have to:

1. Provide the declaration of the `DataFrame` internal data structure.

and you have to provide the implementation of the following methods:

2. 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 1 can be obtained passing "temperatures humidity".

   For the constructor implementation you can rely on the

   ```
   vector<string> split(const string & s, char d)
   ```

   function, which returns the names of columns in `s` separated by the delimiter `d`. In other words, you are not required to implement `split()` yourselves.

weather_conditions_dataframe

temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7

Figure 1: A weather application `DataFrame`.

3. A `set_column` method that, given a vector of values of type `double` and a column name, replaces the entire column content.

   In order to complete this task it is better to rely on the following functions:

   - `check_column_name` checks if a given string is a existing column name in the `DataFrame`;
   - `set_column_data` stores, given the column name and the column data (i.e., a vector of `double`), the data in the column.

4. A `get_column_names` method, which returns in a **set** of `strings` the column names.
5. A `unique` method, which return the unique values (i.e., without duplicates) stored in the column whose name is passed as parameter.
6. A `drop_column` method, which removes from the `DataFrame` the column whose name is passed as parameters.
7. A `set_element_at` method that, given a column name, an index `i`, and a value of type `double`, updates the i-th value of the column.

   Try also to implement `get_column` and `get_element_at` methods.

8. A `get_mean` method, which returns the mean of a given column.

9. A `sum_by_rows` method, which returns in a vector of `double` the sum of values in each individual row of the `DataFrame`.
10. A `select_equal` method 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 1 would yield a new one with both columns, but only the second row.
11. A `concatenate` method, which returns as a copy a `DataFrame` obtained by concatenating the Dataframe with a second `DataFrame` passed as parameter (both original `DataFrame` are left unchanged). An example of concatenation is shown in Figure 2.

df_1

temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7

df_2

temperatures	humidity
28	0.9
21.2	0.5
23.4	0.1
26.3	0.7
27.7	0.6

df_1.concatenate(df_2)

temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7
28	0.9
21.2	0.5
23.4	0.1
26.3	0.7
27.7	0.6

Figure 2: A concatenation of `DataFrame`s example.

Take particular care to error conditions, for example:

- access to a wrong column or element index out of range;
- check that all the columns have the same number of rows (which will be known only when creating the first column)

Finally,

12. Discuss the worst case complexity of the setter methods.

Chapter 3

Evaluate by Column (Exercise 18)

Header file

class dense_matrix final	
private:	
	typedef std::vector<double> container_type
public:	
	typedef container_type::value_type value_type typedef container_type::size_type size_type typedef container_type::pointer pointer typedef container_type::const_pointer const_pointer typedef container_type::reference reference typedef container_type::const_reference const_reference
private:	
	size_type m_rows, m_columns container_type m_data size_type sub2ind (size_type i, size_type j) const
public:	
	dense_matrix (void) = default dense_matrix (size_type rows, size_type columns, const_reference value = 0.0) explicit dense_matrix (std::istream &) void read (std::istream &) void swap (dense_matrix &) reference operator () (size_type i, size_type j) const_reference operator () (size_type i, size_type j) const size_type rows (void) const size_type columns (void) const dense_matrix transposed (void) const pointer data (void) const_pointer data (void) const void print (std::ostream& os) const
outside:	
	dense_matrix operator * (dense_matrix const &, dense_matrix const &) void swap (dense_matrix &, dense_matrix &)

```cpp
dense_matrix::dense_matrix (size_type rows, size_type columns,
                            const_reference value)
  : m_rows (rows), m_columns (columns),
    m_data (m_rows * m_columns, value) {}

dense_matrix::dense_matrix (std::istream & in)
{
  read (in);
}

dense_matrix::size_type
dense_matrix::sub2ind (size_type i, size_type j) const
{
  return i * m_columns + j;
}

void
dense_matrix::read (std::istream & in)
{
  std::string line;
  std::getline (in, line);

  std::istringstream first_line (line);
  first_line >> m_rows >> m_columns;
  m_data.resize (m_rows * m_columns);

  for (size_type i = 0; i < m_rows; ++i)
    {
      std::getline (in, line);
      std::istringstream current_line (line);

      for (size_type j = 0; j < m_columns; ++j)
        {
          /* alternative syntax: current_line >> operator () (i, j);
           * or: current_line >> m_data[sub2ind (i, j)];
           */
          current_line >> (*this)(i, j);
        }
    }
}

void
dense_matrix::swap (dense_matrix & rhs)
{
  using std::swap;
  swap (m_rows, rhs.m_rows);
  swap (m_columns, rhs.m_columns);
  swap (m_data, rhs.m_data);
}

dense_matrix::reference
dense_matrix::operator () (size_type i, size_type j)
{
  return m_data[sub2ind (i, j)];
}

dense_matrix::const_reference
dense_matrix::operator () (size_type i, size_type j) const
{
  return m_data[sub2ind (i, j)];
}

dense_matrix::size_type
dense_matrix::rows (void) const
{
  return m_rows;
}
```

```
dense_matrix::size_type
dense_matrix::columns (void) const
{
  return m_columns;
}

dense_matrix
dense_matrix::transposed (void) const
{
  dense_matrix At (m_columns, m_rows);

  for (size_type i = 0; i < m_columns; ++i)
    for (size_type j = 0; j < m_rows; ++j)
      At(i, j) = operator () (j, i);

  return At;
}

dense_matrix::pointer
dense_matrix::data (void)
{
  return m_data.data ();
}

dense_matrix::const_pointer
dense_matrix::data (void) const
{
  return m_data.data ();
}

void
dense_matrix::print (std::ostream& os) const
{
  using size_type = dense_matrix::size_type;

  os << m_rows << " " << m_columns << "\n";

  for (size_type i = 0; i < m_rows; ++i)
  {
    for (size_type j = 0; j < m_columns; ++j)
      os << operator () (i,j) << " ";
    os << "\n";
  }
}

dense_matrix
operator * (dense_matrix const & A, dense_matrix const & B)
{
  using size_type = dense_matrix::size_type;

  dense_matrix C (A.rows (), B.columns ());

  for (size_type i = 0; i < A.rows (); ++i)
    for (size_type j = 0; j < B.columns (); ++j)
      for (size_type k = 0; k < A.columns (); ++k)
        C(i, j) += A(i, k) * B(k, j);

  return C;
}

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

Chapter 4

Chapter 5

Lab 01

Chapter 6

Lab 02

Chapter 7

Lab 03

Chapter 8

Lab 04: Iterative solvers and preconditioners with Eigen

In this lab we aim at evaluating some hand-made implementation of the most common iterative methods for solving linear systems.

8.1 Hand-made Conjugate Gradient method

First, we create a new directory called `iter_sol++` and we copy the following implementation of the Conjugate Gradient method in a cpp file called `eser1.cpp`:

```
namespace LinearAlgebra
{
template <class Matrix, class Vector, class Preconditioner>
int CG(const Matrix &A, Vector &x, const Vector &b, const Preconditioner &M,
   int &max_iter, typename Vector::Scalar &tol)
{
  using Real = typename Matrix::Scalar;
  Real   resid;
  Vector p(b.size());
  Vector z(b.size());
  Vector q(b.size());
  Real   alpha, beta, rho;
  Real   rho_1(0.0);

  Real   normb = b.norm();
  Vector r = b - A * x;

  if(normb == 0.0)
    normb = 1;

  if((resid = r.norm() / normb) <= tol)
    {
      tol = resid;
      max_iter = 0;
      return 0;
    }

  for(int i = 1; i <= max_iter; i++)
    {
      z = M.solve(r);
      rho = r.dot(z);

      if(i == 1)
        p = z;
      else
        {
          beta = rho / rho_1;
          p = z + beta * p;
        }
```

```
      q = A * p;
      alpha = rho / p.dot(q);

      x += alpha * p;
      r -= alpha * q;

      if((resid = r.norm() / normb) <= tol)
        {
          tol = resid;
          max_iter = i;
          return 0;
        }

      rho_1 = rho;
    }

  tol = resid;
  return 1;
}
} // namespace LinearAlgebra
```