Programmation 1
Lecture title: Classes

F.Marche

IMAG

2021/2022

Introduction

Classes and struct

Classes and structs are the main way to introduce user defined types in a C++ program. They are almost synonyms (the default member access rule changes) but it is generally accepted the **struct**s are used only for collection of data publicly accessible with no constructors and destructors, i.e. for aggregated.

class is used to indicate a more complex type where normally data is not publicly accessible but is manipulated through methods.

Type of class declarations

- Forward declaration. Used just to tell the compiler that a class type exists.

  ```
  class MyClass;
  ```

- Full declaration (or just declaration). It declares the public interface of the class and its private members. It is usually contained in a header file. It may contain also some definitions (in-class definitions). A method defined in-class may be automatically *inlined*.

```
class Name{
public:
        // Declarations of public members
private:
        // Declarations of private members
protected:
        // Declarations of protected members
};
```

Example 1 : a class Rational

We design a class for rational numbers, implemented by storing as integers the denominator and the numerator. We give here a very naive implementation that we will enhance gradually by introducing new concepts :

```
class Rational{
public:
  void set(int num, int den){M_n=num;M_d=den;}
  int den()const {return M_d;}
  int num()const {return M_n;}
private:
  int M_n;
  int M_d;
};
```

Access rules

Given a rational number

```
Rational a;
```

we have two private members, M_n and M_d. We cannot access them direcly, that is a statement like

```
a.M_d=3;
```

is forbidden, and would give a compilation error

```
error: int Rational::M_d is private
```

What we can do to initialize an object is

```
Rational a;
a.set(5,3); // 5/3
...
std::cout<<" The denominator is"<<a.den();
...
```

Access rules : Public vs Private

Making these members private helps to separate the implementation from the interface : for instance in the future we might decide to store the two integers into an array, which keeping the interface unchanged.

It is a common practice to use a consistent naming scheme for private members, here we have chosen to start their name with M_ (M stands for “my”).

We have here decided to interact with the data contained in a Rational using the methods set(**int**,**int**), which inserts the values, and num() and den() which returns the stored values. Methods of this sort ore often called *setters* and *getters*, respectively.

Methods

Methods behave in many respect as ordinary functions, the main difference is that they have authorized access to private members. Now let's have a look at

```
int num() const;
```

This is a constant method, it means that it does not alter the state of the object (i.e. it does not change the value of the members).

Methods of that sort should be declared constant by adding the keyword **const** after the parenthesis enclosing the arguments.

Only constant methods are allowed to operate on constant objects !

This keyword is *part of the signature of the method*. The compiler

may perform more aggressive optimizations on const methods than non-const ones.

Const methods

You may have at the same time a constant and a non-constant version of a method with the same name and arguments, and usual overloading rules apply. Let's have a look to a possible alternative implementation that exploits this fact :

```
class Rational{
public:
  void set(int num, int den);
  int & den(){return M_d;}
  int den()const {return M_d;}
  int num()const {return M_n;}
  int & num() {return M_n;}
private:
  int M_n;
  int M_d;
};
```

Const methods

The method **int** & den(), being non constant and returning a non constant reference, may be used as lvalue, that is a statement like

```
a.den()=5; // uses int & den()
```

is valid and assigns 5 to the denominator of a. Indeed with this technique we could also get rid of the set() method. We need also the constant version because otherwise we could not use the method den() on a constant object :

```
// A function that use a Rational in read mode
void myfun(Rational const & r,...){
    ...
    int a=r.den(); // uses int den() const;
}
```

If I have only the non constant version the compiler will issue an error and will refuse to compile the function.

Definition of class members

In our example, the definitions of the methods are given within the body of the class and we are giving a full definition of the class, i.e. the whole information needed to make a fully operative Rational object.

This is fine if the class is meant to be used only in the translation unit where it is defined. In general however, it is necessary to separate the declaration of the class from the definition of its member.

Definition of class members are usually made in the source file (a part in-class definitions and definition of inline methods).

The name of the member has to be qualified with the name of the class.

Pure declaration

A pure declaration (i.e. a declaration that does not contain any in-class definitions) for our Rational would read,

```
class Rational{
public:
  void set(int num, int den);
  int den()const;
  int num()const;
private:
  int M_n;
  int M_d;
};
```

and it would be normally contained in a *header file*, let's call it `rational.hpp`.

Definition in a separate source file

The corresponding definitions are put in a separate *source file*, let's call it `rational.cpp`, by using the following syntax :

```
#include "Rational.hpp"
...
void Rational::set(int num, int den){
 M_n=num;
 M_d=den;}
int Rational::den()const{
 return M_d;}
int Rational::num()const{
 return M_n;}
```

The inclusion of the header file is necessary because the compiler needs the declaration of the class to have a full view of all its members.

Getting it works

At this point we can compile the source file

```
g++ -Wall -c -std=c++11 rational.cpp
```

producing the object file `rational.o`. If the header file is not in the same directory as the source file (which is often the case), we need to use the compiler option `-I<dir>`, where `<dir>` indicates the directory with the header file. The compiler will try to inline methods defined in-class.
A possible main file could read

```
#include "rational.hpp"
...
int main(){
  Rational a;
  ....
}
```

Getting it works

It must include the header file`rational.hpp`, since the compiler need to have the declaration of Rational to use that type. We can now compile the main file, which we call `main_rational.cpp` and link it with Rational.o to produce the executable

```
g++ -Wall -c -std=c++11 main_rational.cpp
g++ -o main_rational main_rational.o rational.o
```

A pure declaration is sufficient for the compiler to know the size needed for an object.

Constructors

Constructor

Now, we would like to be able to create a Rational by giving directly numerator and denominator, like Rational a(5,3) or just Rational b(5) for the rational number 5/1.

Up to now, we executing the instruction

```
Rational a;
```

an object of 8 bytes is created to hold the two member variables by using the default automatic constructor provided by the language. This basic constructor allocates the memory but does not initialise the members.

We therefore need to provide a constructor *explicitly*. The constructor is a special method of the class with the the same name of the class and *no return value* (indeed its "return value" is an object of the class).

It may take an arbitrary number of arguments, and a constructor that takes no arguments is called *default constructor*.

Constructor

The pure declaration is just

```
ClassName(argument list);
```

while the definition takes the form

```
ClassName::ClassName(argument list): initialisation list
{
  // Constructor body (it may be empty).
}
```

The initialisation list contains the initialisation of the members separated by commas.

The constructor is usually public (otherwise we will not be able to construct the object).

Let's then add a default constructor that initialise with the value 0/1 a Rational.

Constructor

For the constructor we may write the definition within the body of the class declaration, in this case we may write

```
class Rational{
public:
  Rational():M_n(0),M_d(1) {}
  ... // the other members
};
```

If we keep declarations separated, we would add to the class declaration just

```
class Rational{
public:
Rational();
...
};
```

and in the file with the definitions we set

```
Rational::Rational():M_n(0),M_d(1) {}
```

Constructor

The usual overloading rules apply also to constructors, so we can add a constructor that takes two **int**s as arguments to initialise numerator and denominator.

We show here only the case of declaration separate from definitions. In the class declaration we add

```
Rational(int n, int d);
```

and in the definition file

```
Rational::Rational(int n, int d):M_n(n),M_d(d) {}
```

In a code we may now write statements like

```
...
Rational a; // uses Rational(), a is now 0/1
Rational b(5,3); // uses Rational(int,int). b = 5/3.
```

Constructor

Now we want the rational to be normalized and we use Euclid rule to find the prime factors of numerator and denominator. The function is only needed internally and not meant for general use, so it is implemented as a *private method*. Let us call it M_normalize(). The declaration of the class is now

```
class Rational{
public:
Rational();
Rational(int n, int d);
...
private:
M_normalize();
};
```

and the definition of the constructor will become

```
Rational::Rational(int n, int d):M_n(n),M_d(d){
M_normalize();
}
```

Solution for normalize

```
void Rational::M_normalize()
{
  ASSERTM(M_d!=0,"Zero denominator in Rational");
  // handle particular cases
  if (M_d==0) return; // 0 Denominator, do nothing!

  if(M_n==0){
    M_d=(M_d>0?1:-1); // M_d=sign(M_d)
    return;
  }
  // Fix the sign
  bool negative=(M_n*M_d)<0;

  // I use the absolute values here
  M_n = std::abs(M_n);
  M_d = std::abs(M_d);
```

Solution for normalize

```
    int a, b, r;  // Euclid's algorithm...
    if (M_d > M_n){
      a = M_d;
      b = M_n;
    }
    else{
      a = M_n;
      b = M_d;
    }
    r = 1;
    while (r > 0 && b > 0){
      r = a % b;
      a = b;
      b = r;
    }
    M_n /= a;
    M_d /= a;
    if (negative) M_n = -M_n;   // Get back the right sign
}
```

Constructor : default arguments

Constructors, like ordinary functions and methods, can take default arguments, that is you can assign default values to the right most arguments (even all arguments). For instance we could write *in the class declaration*

```
class Rational{
...
Rational(int n,int d=1);
```

by which

```
Rational a(5);
```

will call Rational(5,1), effectively initialising a to the value 5/1.

The default arguments are present only in the declarations. The definitions remains unchanged.

Constructor and conversion

A constructor that takes only one argument (*even if obtained by defaulting the other arguments*) defines an *implicit conversion*.
Unless it is declared *explicit* (see next paragraph).
Since with the constructors just provided the statement

```
Rational a(13); // calls Rational(13,1) -> a=13/1
```

is valid, the compiler knows *how to convert an integer to a* Rational, Therefore the following statements are also valid,

```
Rational a=24; // calls Rational(24,1) -> a=24/1
a=2; // becomes a=Rational(2,1) -> a=2/1
```

Another example : a vector class

We develop a simple class "Vector" of **double** elements, starting with a very basic API which will be upgraded in the following. Our "Vector" class should rely on dynamic allocation and have the following features

- a private **int** member M_size for the vector size, and a private **double*** member M_data for the data,
- 2 constructors :
  - a constructor Vector(**int** i) which allocates the vector space of size i, and set all the values to 0,
  - a constructor Vector(**int** i, **double** v) which initializes all the vector values to v.
- a public print method that prints the vector values on screen

The vector class

```
class Vector{
public:

 Vector(int i):M_data(new double[i]),M_size(i)
 {
   for (int k=0;k<M_size;++k)M_data[k]=0;
 };

 Vector(int i, double v):M_data(new double[i]),M_size(i)
  {
    for (int k=0;k<M_size;++k)M_data[k]=v;
  }
   void Print(){
     // To be defined
   }
private:
 double * M_data;
 int M_size;
};
```

Explicit constructor

Sometimes we do not want to have implicit conversions. For instance, assume that we are creating a class for vectors and a constructor that takes an integer to set the size of the vector. The class declaration looks like

```
class Vector{
public:
 Vector(); // a default constructor
 Vector(int i); // Constructs a vector of size i
...  // The remaining part of the declaration
};
```

By that declaration the statement

```
 Vector a=1; // converts into Vector a(1). Creates a vector of size
 ...
 a=5; // converts into a=Vector(5). Assigns a vector of size 5
to a!
```

are valid.

Explicit constructor

Of course this is confusing and not what we would like to have. To avoid implicit conversions you need to declare your constructor *explicit* by putting the keyword **explicit** before the name of the constructor *in the declaration*. Therefore, with this declaration

```
class Vector{
public:
 Vector(); // a default constructor
 explicit Vector(int i);
...  // The remaining part of the declaration
};
```

the statements

```
 Vector a=1;
 ...
 a=5;
```

are invalid. If we want to create a vector of size 1 we need to write Vector a(1) and if we want to have a temporary vector of size 5 we need to write Vector(5) explicitly.

Destructor

What happens to an object when it exits its scope? A part from static variables in a function, the object is destroyed and its memory area returned to the operative system. Let's have a more detailed look

```
...
 if(some condition){
     Rational a(5, 3)
     ... // some operations
  }
...
```

When a exits the scope defined by the body of the **if** construct it is destroyed and since we have not provided any destructor explicitly, the system will use the automatic destructor, which destroys the object members one by one in the order opposite with that used in the construction process.

Destructor

For our Rational this is fine. Yet there are two cases where an explicit destructor is necessary :

1. when the automatic destructor will not do what is necessary to do ;
2. when the class is a base class in a polymorphic hierarchy of classes. In this case we need a *virtual destructor*.

We deal here with the first case, leaving the second item to a next chapter, as it is related to inheritance. Let us consider the Vector class mentioned before. What would happen with the automatic destructor ? The members are destroyed, so the system will eliminate M_size and M_data, but NOT the memory area possibly pointed by M_data ! Indeed, if we look at the class constructors we see that they use the **new**[]() operator, but there is no matching **delete**[].

Destructor

This is a case where a user defined destructor is mandatory, otherwise you have memory leaks. A destructor declaration takes (only) the following form

Listing 1 – Destructor declaration

```
~ClassName();
```

The definition looks like

```
ClassName::~ClassName()
{
// Destructor body }
```

The body of the destructor may be empty (but the curly brackets should be present all the same) and in that case it will behave exactly as the automatic destructor.

Destructors are called automatically when an object exit its scope

Destructor : the vector class

A destructor for our Vector class may be written as

```
class Vector{
public:
  ...
~Vector(){delete[] M_data;}
  ... //the rest of class definition
};
```

where here we have put the definition in-class.
The destructor operates by first executing the statements in its body and then destroying the object members in the normal way. In this case, **delete**[] M_data; will eliminate the pointed area (before M_data is destroyed). Here you see that having set M_data to the null pointer in the case of an empty Vector is very appropriate : deleting a null pointer is a valid statement (which does nothing at all).

Copy constructors

Copy constructors

If we write

```
Rational b(6,7);
Rational c(b); // initialise c with b
```

the Rational object c is *copy constructed* with b, i.e. is a copy of b.

Since we have not defined a copy constructor explicitly the compiler graciously provides one to us which *copies of all member variables*, in the same order as they are found in the declaration.

So in our case M_n is initialised using the value of b.M_n and b.M_d analogously.

For our Rational this is fine, however there are situations where the automatic copy construction process is not what we want.

Copy constructor

A copy constructor should always be declared using the following pattern

```
ClassName(ClassName const &);
```

So we will need to add to the class interface

```
Rational(Rational const &);
```

and its definition may look like, for our class "Rational"

```
Rational::Rational(Rational const & r):M_n(r.M_n),M_d(r.M_d){}
```

Note that, being a member of the class, the copy constructor may access private members of objects of that class, and in particular those of r.

Copy constructor : the vector class

Now, if no copy constructor is provided, the statements

```
Vector a(5);
...
Vector b(a);
...
```

are valid : `b` is build by copying the value of the members of `a`.

So after that statement `b.M_data` is equal to `a.M_data` and *refers to the same memory area of* `a` !

Changing an element of `b` will then change the corresponding element of `a`. Certainly this is NOT what we wanted.

We want to copy the stored values not the pointers !. This is a typical case where defining a special copy constructor is mandatory.

Correction : the copy constructor for the vector class

```
class Vector{
public:
 // The other constructors
   ...
Vector(Vector const & v):
  M_data(v.M_size==0?0: new double[v.M_size]), M_size(v.M_size){
for (unsigned int k=0;k<M_size;++k)M_data[k]=v.M_data[k];
}

.... // the other members
private:
 double * M_data;
 unsigned int M_size;
};
```

The copy constructor for the vector class

The only peculiar part is the handling of the case of an empty vector.

In that case M_data should be initialized to the null pointer and we should not call **new**[]().

To this aim, we have used the conditional operator, which here comes quite handy. If the size of the given Vector is zero it returns 0 (which is automatically interpreted as the null pointer).

Otherwise, it returns the pointer to a newly allocated area.

A class which has a copy constructor (automatic or not) is called *Copy constructable* and this is a requirement for storing objects of that class on STL containers.

Pointers and class members

Addressing members by pointers

Having defined a class we can use pointers to an object to that class. So the following code is correct :

```
Rational * a=new Rational(4,5);
...
cout<<(*a).den();
```

However, there is an operator able to address members of a class through a pointer, eliminating the need of the cumbersome expression (*a).den(). This operator is sometimes called *the arrow operator* and its signature is

```
ClassName & ->(ClassName *);
```

The previous statements may then be written as

```
Rational * a=new Rational(4,5);
...
cout<<a->den();
```

The keyword **this**

How can a method access a member of the calling object ? It does it through a special pointer, called **this**, which is indeed a special variable indicating the pointer to the calling object.
All (non static) methods of a class have **this** as an hidden leftmost formal argument, which is assigned to the pointer to the calling object when the method is called. Therefore, **this** can be used inside the body of a method as an ordinary pointer. For instance, the method

```
int Rational::den() const{ return M_d;}
```

could be written equivalently as

```
int Rational::den() const{ return this->M_d;}
```

The keyword **this**

Here we access member M_n by dereferencing the pointer **this** which indeed points to the calling object.

Its type for an non-constant method is ClassName * **const**, so you are not supposed to change it.

For a constant method the type of the **this** argument is instead ClassName **const** * **const**.

We will see situations where the use of **this** is compulsory, and many programmers suggest to use it in all situations, even when non strictly needed, since it helps code readability when you are dealing with complex classes with many members.

By using **this**, a method can also return a reference to the calling object, and we will see that this is essential for the correct implementation of some operators, like the one we will be dealing next.

Operators overloading

Overloading operators

Operator overloading is a very powerful feature of the C++ language.

It enables, for instance, to express in a synthetic form common operations on user defined classes.

If I have created a class for storing matrices, A+B is much more intuitive to express matrix addition than, say, addMatrix(A,B,R);.

Yet, be very careful in not overusing operator overloading. It must be adopted *only* when the semantic of the operator applied on the given class is evident, and possibly near to that of in-built types.

If this is not the case, use an ordinary method, or free function to perform the wanted operation, not operator overloading.

We refer to some reference textbook for examples of other operators overloeading.

Overloading operators

Operators in general may be implemented as free functions or as members of a class. Assignment ($=$), subscript ([]), conversion, function call (()) as well as $->$ operators can be implemented *only as members*. In general a binary operator in its symbolic form a σ b translates to

```
operator σ(a,b)
```

when implemented as a free function, and to

```
a.operator σ(b)
```

when implemented as a method. Conversely, a unary operator in prefix form, σa translates to

```
operator σ(a)
```

or

```
a.operator σ()
```

Assignment operator

Assignment is another basic operation that the C++ language provides automatically also for class types.

It means that even if the current implementation says nothing about an assignment operator for a Rational, the piece of code

```
Rational a(3,5)
Rational b;
b=a; // The value 3/5 is assigned to b
```

works as expected.

The assignment operator = is a member of class and its automatic version copies the member variables of the object on the right hand side to the one on the left hand side (which is the calling object). The order by which members are copied is that in which they appear in the class declaration.

Assignment operator

The standard declaration of the operator is

```
class Classname{
...
ClassName& operator =(ClassName const &);
...
};
```

It takes a constant reference as argument, thus it should not modify the status of the expression at its right hand side (which is a reasonable requirement). It returns a reference to the calling object because the statement

```
a=b=c;
```

is valid in C++ for in-built types, so generally you want it so also for user defined types. Consequently, b=c must return a valid object to be assigned to a (remember that assignments are right associative).

Assignment operator

If we want to write explicitly an assignment operator for our Rational (even if not strictly necessary) we need to add to the interface of the class the statement

```
Rational & operator=(Rational const &);
```

while the definition may read

```
Rational & Rational::operator=(Rational const & r){
if (this !=&r ){
 M_d=r.M_d;
 M_n=r.M_n;
}
return *this
};
```

Having kept the definition separate we need the fully qualified name Rational::**operator** =. The **if** construct tests whether we are assigning an object to itself. Indeed a=a; is a valid statement and should do nothing, a part returning a reference to a.

Assignment operator : the vector class

Here we give a possible definition,

```
Vector Vector::operator= (Vector const & v){
if (this != &v){
 assert(v.M_size==M_size);
 for(unsigned int k=0;k<M_size;++k)M_data[k]=v.M_data[k];
}
return *this;
}
```

Here we have used the macro assert() that will be described in a later section (it requires the header <cassert>) to test whether the two vectors have the same size and abort the program if this condition is not satisfied. Of course, this is a design choice, we could have decided to treat this situation otherwise.

Assignment operator

A class where the assignment operation is defined (either explicitly or implicitly by the default operator) and has the following semantic :

- it assigns the value of the right operand to the left operand ;
- it does not modify the state of the left operand ;
- it returns a reference to the left operand ;

is called *Assignable*. This property is a requirement to store objects of the class in a STL container. In general, a part particular cases (like for instance in a smart pointer implementing exclusive ownership), you should make sure that your class is assignable.

Subscript operator []

Another operator which is implementable only as a member of the class is the subscript operator.

No automatic version is provided as its implementation strongly depends on how the data is organized.

It is meant for container classes which behave "like" an array. It takes as argument an "integer type" and returns either a value or a reference to the contained object.

The declaration of the subscript operator may take one of the two forms

```
value_type & operator [](integer_type);
```

which enable us to change the content of our "container", or

```
value_type operator[] (integer_type) const;
```

which is used on constant objects. Usually you do need both.

Subscript operator []

```
class Vector{
public:
  ... // the other methods
  double & operator[](unsigned int i){return M_data[i];}
  double operator[](unsigned int i) const {return M_data[i];}
  ...
};
```

If the Vector object is an `lvalue`, the non constant version will be used. If the object is constant only the constant version is available.

Subscript operator []

The overloading mechanism applies because the keyword **const** is part of the signature. Here two code snippets to explain the situation :

```
...
Vector a(10); // A vector of 10 elements
a[5]=7; // uses the non constant version.
...
```

and

```
#include <algorithm> // for the max() function
#include <cmath> // for the abs() function
  // Definition of a function that computes the maximum norm
double norminf(Vector const & v){
double temp(0.0);
// The constant version is used here:
for (unsigned int i=0;i<v.size();++i)
                    temp=std::max(temp,std::abs(v[i]));
return temp;
}
```

Overloading operators

We have seen some examples of definition of operators for user defined types. C++ allows to redefine almost all operators. This operation is called *operator overloading*.

The operators that cannot be overloaded are the following :

Operator	*Description*
::	Scope resolution
.	Member selection
.*	Member selection through pointer to function
?:	Ternary condition
sizeof()	Computing the size of an object of given type
typeid()	Type identification

The move operations (C++ 11)

The move constructor

If a vector has a lot of elements, it can be expensive to copy. So we should copy vectors only when we need to.
Consider the following case :

```
vector fill( istream& is)
{
va = vector(10000)
... // instruction to fill va from a file
return va;
}

void use()
{
...
vector vec = fill(cin)
...
}
```

The move constructor

In this example, we fill the local vector va from the input stream and return it to use(). Copying va out of fill () and into vec could be expensive. But why copy ? We don't want a copy, we can never use the original va after the return (and va is destroyed as part of te return from fill ().

C++ 11 introduces the move operations to complement the copy operations :

```
class vector{
        int sz;
        double*elem;
public:
    vector (vector&& a): // move constructor
    vector& operator=(vector&&);
    // ... other methods
        }
```

The move constructor

The && notation is called an "rvalue reference". We use it for defining the move operations.

Part of the purpose of a move operation is to modify the source, to make it "empty" : move operations do not take **const** arguments.

By defining a move constructor, we make it cheaper to move large amounts of data, such as a vector with many elements.

Given a move constructor, and calling again the function fill (istream& is), the move constructor is implicitly used by the compiler to implement the return. It knows that the local value returned va is about to go out of scope, so it can move from it, rather than copying.

Using the move operations prevents us from dealing with pointers and references to get a similar behavior.

The vector class : move operations

```
vector::vector(vector&& a)
    :sz{a.sz}, elem{a.elem}
    {
        a.sz = 0; //make a the empty vector
        a.elem = nullptr;
    }
```

Friendship

Friendship

Only public members can be accessed directly outside the scope of the class and this is normally done by calling the appropriate public method.

However it is allowed to specify that a free function has the right to access private member, by declaring it **friend** of the class.

You need to add to the class declaration the full specification of the function prefixed by the keyword friend. A friend declaration can be placed anywhere within the class body.

```
class X{
public:
  ...
friend double foo(X& x); // foo is a friend of X
...
private:
double a;
int b;
};
```

Friendship

Being foo() a friend it can access the private members of class X :

```
double foo(X& x){
...
double c=X.a; // allowed because friend
...
```

There is a peculiar rule that related friendship with visibility.
Besides the **friend** statement, for a function to be friend of a class one of the following conditions must be satisfied :

1. its declaration is in a scope enclosing that of class declaration ;
2. it takes an argument of that class.

Friendship

For instance

```
  ...  // there is NO declaration of f here
  ... // nor of foo()
double g(); // declaration of g
class X{   // Declaration of X
public:
  ...
friend double f(); // USELESS!
friend double g();
friend int foo(X const &);
...
private:
double a;
int b;
};
```

Function f() is NOT friend of X even if declared so, because its declaration was not visible (i.e. it was in an enclosing scope).

Output stream operator

What about pretty printing our Rationals ? We may want that for a Rational a the statement cout<<a; prints something meaningful. In particular we may want it to print it

- as an integer, if the denominator is 1.
- as a fraction M_n/M_d if $|a| < 1$;
- In the form $x + c/M_d$ if $|a| > 1$, where x is the integer part of a.

We need to define an output stream operator << for a Rational. According to the norm, it has to be a free function, and, since we want to access directly the member of the class we make it **friend** :

```
class Rational{
...
friend std::ostream & operator<< (std::ostream &,
        Rational const &);
...
};
```

Output stream operator

A output stream operator takes a reference to an output stream as first argument and returns *the same reference*. In this way cout<<a would return cout, ready for another successive stream operator. Then, its definition is

```
std::ostream & operator<<(std::ostream & str,
                                    Rational const & r)
{
  if (r.M_d==1) str<< r.M_n;
  else
    if (int d=r.M_n / r.M_d)
      str<<d<<"+" << r.M_n%r.M_d << '/' << r.M_d;
    else
      str<< r.M_n << '/' << r.M_d;
  return str;
}
```

We recall that the operation / among two integers is an *integer division*.

Output stream operator

We also exploits the fact that an initialization returns the initialized value, so **if** (**int** d=r.M_n / r.M_d) computes the integer division r.M_n / r.M_d), uses the result to initialise d and tests the returned the value. If it zero it is interpreted as **false**, otherwise it is **true**.

The inline directive

The declaration inline may be applied also to methods of a class.

```
class foo{
public:
  // Declaration of a inlined method
  inline double method1(int);
  // In-class definitions implies inlining
  double & getValue(i){return my_v[i];}
  ...
};
```

Remember that you should inline only short functions. Moreover, inline only tells the compiler that it may inline that function, it is not sure that it will do it.
Definition of inlined functions should be made in the header file.
A last note. Inlining may make debugging difficult. One may use a preprocessor macro to disable it during debug phase.

Use of macros to disable inlining

An example of how to use a preprocessor macro to disable inlining. I want inlining disabled unless the option `-DNDEBUG` is given at compilation stage.

```
#ifndef NDEBUG
#define INLINE
#else
#define INLINE inline
#endif
...
// This function is inlined only if
// NDEBUG is defined
INLINE double fun();
```

Static members

A variable member of a class may be declared **static**. It means that all the object of the class share the same variable. It is a useful technique to store quantities common to all objects of the same type (often they are constants).

Also methods may be declared **static**. Only static methods can access static variables without the need of an object, since they are method at the class scope.

A first example of static variable

```
class TriaElement{public: TriaElement()
...
static const int numnodes=3; }
...
//I use the static variable
vector<int> nodeID(TriaElement::numnodes);
```

I had to use the scope resolution operator `::` to access the static variable. `TriaElement a; int n=a.numnodes` is an error since operator `.` access public object member not static class members (unless they are declared off-class, but this is a detail). Note : Only integers constant may be initialized in-class !

Exercice 1 : the vector class

- starting from the provided Vector class, add the output stream overloading, to allow some "pretty print" of the vector's values.

Exercice 2 : the vector class

In the next lecture, we will implement a polymorphism-based Conjugate-Gradient algorithm and need some additional operators. So for the next lecture :

- add the following overloaded operators
  - unary +, unary -, binary +, binary -, assignment += and -=
  - products for double×Vector and Vector×double
- add a reset method **void** reset(**double** = 0) ;
- add a size method **int** size() **const** (which returns the vector's size)
- add a twonorm and maxnorm functions with the prototypes

```
double maxnorm() const;
double twonorm() const;
```

- add a dot friend function with the following prototype

```
friend double dot(const Vector&, const Vector&);
```

Exercice 3 : design of a dynamic matrix class

We want to create a class for holding matrices of doubles with the following characteristics :

- Dynamic dimensions ;
- a matrix-vector product

The class declaration is provided in the exercice folder.

Another detail

- Data is stored in a dynamic array accessed through the pointer data. This means that I have to take care of memory management. In particular I have to define my own default constructor, destructor, copy constructor, and assignment operator : the automatic (synthetic) ones would do the wrong thing !.
  Indeed copy constructor and assignment operators should here perform a deep copy (copy the data not the pointer to the data !).
  Destructor has to call **delete**[] data