Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Classes

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Classes and struct

I assume you already know what a class is, we will recall here only the main concepts.

Classes and structs are they 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 user defined constructors or destructors, i.e. for aggregates.

While, **class** is used to indicate a more complex type where normally data is not publicly accessible but is manipulated through methods. From now on we use the term class of class-type object to indicate a type and the corresponding object introduces by a **class** or **struct** declaration.

Automatic (or synthetic) methods

When you create a class T you automatically have at disposal the following fundamental methods

Default constructor	T()
Copy-constructor	T(T **const** &)
Move-constructor	T(T&&)
Copy-assignment op.	T & **operator**=(T **const** &)
Move-assignment op.	T & **operator**=(T&&)
destructor	~T()

They can be overloaded, or even deleted (a part the destructor)

Rules for automatic method generation

The synthetic methods are generated automatically only if the user do not declare them explicitly. Moreover:

- A default constructor is generated only if the class do not declare any other constructor;
- Move operations are generated automatically for the class only if: a) No copy operations are declared; b) No move operations are declared; c) No destructor is declared;

If you want the synthetic methods (even when they would not be generated automatically) use the keyword **default**. If you do not want them to be generated use the keyword **delete**.

Automatic methods

```
class Foo{
  public:
  Foo(int i); // overloads constructor
  Foo()=default; // but I want the autom. default one
  // This class has no copy-assignment operator
  Foo& operator=(const Foo &)=delete;
  operator
};
```

What do the automatically generated method do?

The default constructor apply the default constructor to all non-static variable members of the class in the order they are declared in the class.

The other methods (a part the destructor) ...do the same..., just changing the verb: a copy-constructor copy-construct the non-static variable members, assignment operators assign them (by copying or by moving), always in the order they are declared in the class.

The destructor calls the destructor of all non-static member, in the opposite order with respect their declaration in the class.

Default constructor

A default constructor is any constructor that takes no argument. A class with a default constructor is called default constructible.

```
class FirstClass{
 public:
  FirstClass(); // a default const.
  ...};
class SecondClass{
 public:
  // Also this is a def. constr.!!
  SecondClass(int i=5):my_i(i):{}
 private:
  int yy_i;
};
```

Constructors in general

The in-class definition of a constructor has the form

`ClassName(<Param>):<Init. List>{<Body>}`

Here <...> indicates optional parts. An out-of-class definition:

`ClassName::ClassName(<Param>):<Init. List>{<Body>}`

The initialization list is used to initialize variable members. Members not explicitly initialized in the initialization list are initialized by the default constructor. Members are initialized in the order they declared in the class, irrespectively of the order in the initialization list. The body may contain other actions, different from initialization. When entering the body all members have been constructed and are available. --> Gli attributi sono stati costruiti da noi tramite initialization list o da default constructor Initialize members in the initialization list, do not assign them in the body of the constructor, it is more efficient!. The body can be empty (but you need the { }).

Destructor

The destructor is a special method of the class with name `~ClassName`. It has no return type nor arguments.
Normally, it is required to write a destructor only if the class handles memory dynamically through pointers, as our MyMat0:

```
MyMat0::~MyMat0() { delete[] data;}
```

Note that `delete[]` is safe if we make sure that `data` is always a valid pointer, eventually setting it to the null pointer.

Important note: If the class is a base class of a hierarchy the destructor must be declared `virtual`!

Copy constructor

Also the copy constructor is normally automatically given by the compiler (synthetic copy-constructor) and it is called when we construct an object copying it from another object of the same type: MyMat0 a(b). The synthetic one just copies the members (using their corresponding copy-constructor) one by one, in the order of declaration.
If I store pointers I need to write my own copy constructor if I want a deep copy (i.e. to copy the pointed data and not the pointer itself.
A class with an accessible copy constructor is called copy-constructable. You may make a class not copy constructable by declaring the copy-constructor private, but in C++11 it is better to use the `delete` keyword for this purpose.

A copy constructor for MyMat0

Declaration:

```
MyMat0(MyMat0 const & m);
```

Definition:

```
MyMat0::MyMat0(MyMat0 const & m):
nr(m.nr),nc(m.nc),myPolicy(m.myPolicy),getIndex(m.getIndex)
{ if (nr*nc > 0){
    dati=new double[nr*nc];
    double * const & mcp=m.dati;
    // Deep copy
    for (size_type i=0;i< nr*nc;++i)dati[i]=mcp[i];
                }
// it is not necessary...but safer
else dati=0;
}
```

Move-constructor and move-assignment

We postpone the explanation to the lecture on move semantic.....

A general rule

If there is no special reason to do otherwise, make sure your class is default constructible, copy constructible and copy assignable (and possibly also move-constructable and move-assignable), using the synthetic or user specified methods.

If the synthetic methods are good for you there is no need to define your own. However, you may wish to use the keyword **default**, to make clear to the user that you are happy with the synthetic method.

Explicit constructors

Use the keyword **explicit** if you do not want implicit conversion!

```
class Foo
{
  Foo(int);
...
class Foo2
{
   explicit Foo2(int);
...
Foo a=sqrt(10.);// implicit conversion! Integer part!
Foo2 b=sqrt(10.);// Error! No implicit conversion
```

Types 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 MyClass{
  public:
  MyClass(int); //Constructor
  MyClass(MyClass const&); //Copy constructor
  int get()const {return my_i}; // in-class definitin
  void set(int i);// Pure declaration of a method
  private:
  int my_i; // private variable
}; // remember the ;
```

Constant and non constant methods

A method that does not alter the state of a class should be declared **const**, like the method get() in the example.
Only constant methods may operate on constant objects!

```
void afun(Myclass const & m, int i){
  m.set(i)// ERROR m is const}
```

The compiler will never allow you to use non-const methods on constant objects (the reason is obvious).

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

const is your friend, use it!.

API and implementation

All public members of a class form the so called public interface, sometimes indicated with API, even if API whould more properly be referred to a whole application or library non just a class.

All the private (and protected) members of a class are part of its implementation.

Definition of class members

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

ON out-of-class definitions the name of the member has to be qualified with the name of the class:

```
// Definition of the constructor
MyClass::MyClass(int i):my_i(i){}

// Definition of a method
void MyClass::set(int i){my_i=i;}
```

The design of a (not so) simple class

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

- Dynamic dimensions;
- Access to the elements by operator `(int i, int j)`, numbering starting from 0.
- Possibility of storing the matrix row-wise or column-wise.
- Possibility of obtaining the dimensions of the matrix.
- Having at disposal some tools to compute matrix norms and operations with vectors.
- Creation of a namespace to avoid name clashes.

The class definition is in MyMat0_pointer/MyMat0.hpp

A possible layout

```
namespace LinearAlgebra{
enum StoragePolicySwitch {ROWMAJOR,COLUMNMAJOR};
typedef std::size_t size_type;
class MyMat0{
 private:
  size_type nr,nc;
  double * data;
....
```

We use an enumerator to indicate whether to organize data row-wise or column-wise. I use a **typedef** to indicate the type used to address indexes. The standard library in <cstddef> provides size_t for this purpose. I rename it size_type to be consistent with the name used in vector<>.

Memory organization of a two dimensional matrix

Computer memory has normally a linear addressing so a matrix is in fact stored in a vector.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Row-wise ordering

$$[1, 2, 3, 4, 5, 6, 7, 8, 9,]$$

Column-wise ordering

$$[1, 4, 7, 2, 5, 8, 3, 6, 9]$$

Policy via pointers to function

The specification of different behavior of a class at compile or run time is normally called policy. Here the policy has to prescribe how the matrix is organized in data:

- ROWMAJOR: $(i, i) \rightarrow j + i * nc$
- COLUMNMAJOR: $(i, j) \rightarrow i + j * nr$

I can create two functions like

```
size_type RM(size_type const i, size_type const j)
{return j+i*nc;}
```

to be chosen in alternative by the use of a pointer to function. A problem `nc` e `nr` are private variable of the class, either I pass the matrix as a reference and I make the function friend (very cumbersome!) or I make the policies method of the class and I access them through a pointer to member function.

Pointers to member function

We recall the usage of pointers to members:

```
class Pippo{
public:
double c(double v);
double d(double v);
};
int main(){
// pointer to member function
double(Pippo::* pi)(double);
pi=&Pippo::c; //assign a member function
Pippo g;
double z=(g.*pi)(20)=; // z=g.c(20);
...
```

pi è un puntatore ad un metodo della classe Pippo che riceve un double e restituisce un double

Policy implementation

StoragePolicySwitch: enum (è anche attributo)
storagePolicy: attributo (è puntatore ad un metodo)

```
class MyMat0{
private:
 size_type nr,nc;
 double * dati;
 StoragePolicySwitch myPolicy;
 //! The policy for storing by rows
 size_type rowMajorPolicy(size_type const & i,
                          size_type const & j) const;
 //! The policy for storing by columns
 size_type columnMajorPolicy(size_type const & i,
                             size_type const & j) const;
 typedef size_type (MyMat0::*storagePolicy) (size_type const &,
                             size_type const &) const;
 storagePolicy getIndex;
public:
 MyMat0(StoragePolicySwitch sPolicy=ROWMAJOR);
  ....
```

storagePolicy è un tipo equivalente ad un puntatore ad un metodo di MyMat0 che riceve due size_type const & e restituisce size_type

Nel costruttore dovrò settare getIndex di modo che punti alla funzione esatta (rowMajorPolicy o columnMajorPolicy)

The constructor

```
MyMat0::MyMat0(StoragePolicySwitch sPolicy):
nr(0),nc(0),data(0),myPolicy(sPolicy){
    switch (this->myPolicy){
    case ROWMAJOR:
        getIndex = &MyMat0::rowMajorPolicy;
        break;
    case COLUMNMAJOR:
        getIndex = &MyMat0::columnMajorPolicy;
        break;
      } }
```

Note that data is initialized to the null pointer!. The preferred C++11 syntax would be data(nullptr) instead of data(0).
In C++11 it is better to declare the enum as
enum class StoragePolicySwitch{... to avoid conversion with integers.

Member access

```
inline double operator ()
(const size_type i, const size_type j) const
{
  return data[(this->*getIndex)(i,j)];
}
  inline double & operator ()
        (const size_type i, const size_type j)
{
   return data[(this->*getIndex)(i,j)];
}
```

getIndex non è necessario sia public in quanto viene chiamato all'interno di operator(), non viene chiamato direttamente dall'utente

I delegate to the policy getIndex the localization of the element in the array data. Note that I have implemented both const and non-const version and the latter returns a reference, so m(1,1)=1.5 will modify the matrix.

Note The use of **this**-> to address a class member is not strictly necessary in C++ (at least here...) but may help to understand the code better.

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

Matrix-vector product

I want to define the product of a MyMat0 matrix with a vector<**double**> with the standard rule.
An efficient implementation, however, depends on the internal organization of the matrix. We want to access the matrix elements sequentially in memory to reduce *cache misses*. The algorithm implemented are

- row-major: $\mathbf{r} = A\mathbf{v}$ is implemented as $r_i = \sum_j A_{ij} v_j$.
- column-major: $\mathbf{r} = A\mathbf{v} \rightarrow \mathbf{r} = \sum_j v_j \mathbf{r}_j$, being $\mathbf{r}_j$ the j-th column of A.

Implementation of matrix-vector

We recall that among integers the division operator / behaves as integer division. Moreover, % is the modulo operator (rest of the integer division)

```
switch(this->myPolicy){
case ROWMAJOR:
  // Classic A*v row by row
   for (size_type i=0;i<nc*nr;++i)
    res[i/nc]+= data[i]*v[i%nc];
   break;
   case COLUMNMAJOR:
   // linear combination of the columns
   for (size_type i=0;i<nc*nr;++i)
     res[i%nr]+=data[i]*v[i/nr];
   break;}
```

The complete code in MyMat0_pointer/MyMat0.cpp

Why not to make life easier

The choice of storing matrix elements in a C-style dynamic array has as a consequence the necessity of taking care of memory management. Would not be better if the memory management could be taken care by data itself?

Indeed, we can decide to store the elements in a vector<**double**>. In this way the synthetic copy constructor, assignment and destructor are fine! I do not have to bother writing my own.

An implementation is in MyMat0_vector directory.

Note If MyMat0 is meant to be a base class for a hierarchy of classes (inheritance), I still need to declare a **virtual** destructor.

Se utilizzassi vector<double> e non volessi sfruttare inheritance: mi va bene il default constructor
Se utilizzassi vector<double> e volessi sfruttare inheritance: devo dichiarare un virtual destructor

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.

Definitions 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!

Another example

A more complex commented example is found in
StaticMembers/class_ws.hpp

Clonable classes

Often, it is useful that a class can clone itself. It is a way to implement the Prototype design pattern.

```
class MyClass{
public:
...
std::unique_ptr<MyClass> clone()const
  { return std::unique_ptr<MyClass>(new MyClass(*this))};
};
```

Or (C++14):

```
std::unique_ptr<MyClass> clone()const{
            return make_unique<MyClass>(*this);}
```

MyClass must be copy-constructible. Usage:

```
MyClass a;
....
auto b = a.clone();
```

Uniform initialization

C++11 has introduced uniform initialization with the $\{\cdots\}$ syntax. This may be confusing because now you can do

```
int x(10); // x is an int initialized with 10
int y=10; // y is an int initialized with 10
int z{10}; // z is an int initialized with 10
int k={10}; // k is an int initialized with 10
```

Let's try to understand better. The new syntax has been introduced to allow a way to initialize object that acts uniformly across different types. There are differences though!

brace initialization do not allow narrowing

The first difference is that brace initialization do not allow narrowing on built-in types:

```
double x,y,z;
...
int sum1{x+y+z};// ERROR! doubles are not expressible as int
int sum2(x+y+z);// Will be truncated to integer part
float y;
...
double z{y}; //Ok, no narrowing here.
```

Uniform with respect to 0 argument

```
class Foo
{
 Foo(); // default constr. (no arguments)
 Foo(int); // constr. taking a int
..
}
..
Foo a; // calls Foo()
Foo a(10); // calls Foo(10);
Foo a();// It is a function declaration!
Foo a{};// Calls Foo();
```

Use in member in-class initialization

```
class foo
{
private:
int x =0; // fine in-class initialization
int z(0); // ERROR!!! Non possible
int z{0}; // Ok
}
```

With user defined classes

```
class Foo
{
 Foo(int,bool); //
 Foo(int,double); // Another constructor
...};
Foo a(10,true); // calls Foo(int,bool);
Foo a{10,true}; // calls Foo(int,bool);
Foo a{10,10.0}; // calls Foo(int,double);
Foo a(1.5,10.0); // calls Foo(int,double);
Foo a{1.5,10.0}; // Error: narrowing not allowed;
```

Beware!

Sometimes the meaning can be different!

```
// Constructing a vector initialized with 10 and 20
vector<int> a{10,20};
// Constr. vector of 10 elements with value 20!
vector<int> b(10,20);
```

This is due to the poor design of std::vector. But here is nothing we can do.

std:: initializer_list <T>

You may use std:: initializer_list <T> to specify that you want to use the {} initialization for your class, for elements of the same type!

```
// A class that takes an initializer list
class Pippo
{
  Pippo(std::initializer_list<int> il);
  ...
}
```

However beware that it is not obvious. Read carefully a good book on C++11 (the Lippman book for instance).

To conclude

My suggestion is: use the `()` or `{}` way of initializing variables in a consistent way in your code. Use the other only when necessary.

Which on of the two to choose as "principal one" is up to your taste!