# Advanced Methods for Scientific Computing (AMSC)
# Lecture title: Classes

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

## Classes and struct

Classes are one of the main features of C++ and are the basic tools for object-oriented (OO) programming. I wish to recall, however, that C++ is not an OO language: it's a language that supports object-oriented, functional and generic programming, and you normally obtain the best by combining those features.

Classes are meant to nucleate concepts and functionalities, and classes may collaborate in different ways for implementing more complex concepts.

## The single responsibility principle

Every class should be responsible of a single and well identifiable task, part of the implementation of your code.

Classes expose a well define public interface which should not change, but may be extended (open-closed principle). Only this way classes can be easily reused and collaborate with each other to make up your program or library.

Often classes are designed with a bottom-up approach: classes representing simple well identified components, are developed and tested separately, and then this basic components collaborate to form classes representing more complex functionalities.

## An example

The concept of a mesh, i.e. a partition of a nD domain into simple polygons is a complex concept made of

- Geometric elements: a single type (e.g. triangle) or of a family of types (polygons) which themselves need the concept of Points, and of elements at the boundary.
- Containers for geometric elements and their relation;
- Tools to compute quantity of interest: area/volume, barycenter, etc. etc.

Identifying the basic components and enucleating them in classes allows a better re-use, documentation and testing.

## Classes in C++

A class object is defined by using either the keyword **class** or **struct**. They are "almost" synonyms: the only difference is the default access rule: public for **struct** and private for **class**.

It is a common habit (but not compulsory) to limit the use of **struct** to define just collection of data publicly accessible, more precisely aggregates, while **class** is used to indicate a more complex type where data can be not publicly accessible.

With a class you are effectively introducing a new type in your code.

From now on I use the term *class* and *class-type object* to indicate a type, and the corresponding object, introduced by a **class** or **struct** declaration.

## An example of simple class

```
    class Foo
    {
    public:
    using Real = double; //a type alias
    Real a=10.0; // in-class initialization
    int  j; // Another non-static member variable
    // To initialize a static member variable I need inline (C++17)
    inline static Real c=10.89;
    static constexpr Real pi=3.1415;
    auto getV()const {return v;} // a getter
    void setV(const Real & a){v=a;} //a setter
Real & setV{return v;}// A more C++ style setter!
    static void setX(Real const & a){x= a;}// sets a static variable
    private:
    inline static Real x{0.};// A static member variable
    Real v{5.};
mutable int c{0};// a mutable data member initialised to 0
    };
```

## An example of simple class usage

```
// Setting a static variable
// I do not need an object since it is static
Foo::setX(10.0);
// Object default constructed;
Foo p;
// I can get the static variable
// also with member access operator
auto constexpr x=p.pi;
// But I can do also:
// auto constexpr x=Foo::pi;
p.setV(10.0); // setting value of p.v
p.setV()=10.0;// using the more C++ style setter
using Real = Foo::Real; // extracting type
Real y{p.getV()};// Initializing y with p.v
auto pf=std::make_unique<Foo>(); // a unique_ptr<Foo>
pf->setV(90.);// Accessing through pointers
```

## What can a class can contain?

Class members can be:

- Types: either type alias, defined with the **using** construct, or nested classes;
- (non-static) member variables. They form the state of an object of that class.
- static member variables. They are an attribute of the class, shared by all class objects. Introduced by the keyword **static**.
- (non-static) methods (or function members). They can access data members (static and non static).
- static methods. They can access only static members of the class. Introduced by the keyword **static**.
- static constant expressions. Only static (well it would not make sense otherwise)

Variables and constant expressions in a class are generically called data members, in opposition to function members also called methods.

## Member access

Non-static members are accessed via the member access operator (e.g. a.x), if you have a pointer to an object of the class you use the other form of member access operator: pa−>x.

Static members may be accesses also with the scope resolution operator (e.g. Myclass::x): we do not necessarily need to have an object of that class to access them.

Non-static member functions can access member variables of the class or just using their name, or with the **this** pointer: **this**−>x.

Static member functions can access only static member variables of the class, using their name or the :: operator: MyClass::s.

## An example of static variables and functions

```
class  TriaElement{
public:
...
// a constexpr member must be static!
static constexpr int numnodes=3;
// Here a static constexpr function
static constexpr int NNod(){return 3;}
};
...
//I can access the static variable with no object
std::array<int,TriaElement::numnodes> nodeID;
//But I can use also the constexpr method instead
std::array<int,TriaElement::NNod()>;
```

## const methods of a class

A (non-static) member function that does not change the state of the class should be declared **const**.
Only const methods are available in a const object of the class.
The **const** specifier is part of the member function signature and participates to overloading. If the same method is present with its const and non-const version, the non-const version is used only on non-const objects.

```
Foo foo;
Const Foo cfoo;
foo.fun(); // double fun() is called
cfoo.fun();//double fun() const is called
cfoo.gun();// SYNTAX ERROR! gun() is not conts!
```

I repeat: always declare const methods that do not change the state of the class.

## The **this** pointer

Any non-static function members of a class has access to a special pointer that is the pointer to the object that is calling that member function:

```
struct Goo{
double fun(); // declaration
double fun()const; // declaration
double gun();// another method
double y;
};
double Goo::fun(){
auto x = this->y; // this is a Goo*
...
}
double Goo::gun() const{
auto x = this->y; // this is a Goo const *
...
}
```

C++ spares you the need to indicate **this** explicitly when accessing a member.

## mutable members

If a class object is declared **const**, you can use only const methods and not change its state. In some particular cases, it may be necessary to have some data members modifiable even on constant objects. You may use **mutable**.

```
class Foo{
 public:
  Foo(double a):a{a}{}
  double getA() const {
    return a;
    done=true;          }
 private:
  double a=0.;
  mutable bool done=false;
}
....
const foo s{3.}// a=3.0, done=false
auto b=s.getA(); // done=true
```

## Why mutable data members

If a member variable is only part of the implementation of the class, that is i sonly used internally to cache data which is not exposed to the "general public", make it mutable, you allow to modify them in methods that are "morally constant".

With *morally constant method* I mean a method that you expect to be able to call in a constant object of the class, and thus is bound to be marked **const**.

We will see some examples in practice.

## Access rules

Class members can be given different access rules:

- public: For everybody!. All can access the member. We say that the member is part of the public interface of the class.
- private: Only for me!. Only member functions of the class can access a private member. No one else!
- protected: For me and my family!. The member is occessible only by methods of the class and of publicly (or protected) derived classes.

## Implicit (or synthesized) methods

When you create a class T you have automatically at disposal following fundamental operators automatically:

| | |
|---|---|
| Default constructor | `T()` |
| Copy-constructor | `T(T const &)` |
| Move-constructor | `T(T&&) noexcept` |
| Copy-assignment op. | `T & operator=(T const &)` |
| Move-assignment op. | `T & operator=(T&&) noexcept` |
| Destructor | `~T()` |

The ones automatically provided by the compiler are called implicit or syntesized operators, but also automatic.

They can be user defined, or even deleted (a part the destructor, which cannot be deleted)

## Why **noexcept**

I have used the **noexcept** specifier for the move operations. It is not compulsory but it's better having move operators that do not throw exceptions and to indicate it!

In general, if a function or method does not throw an exception you can indicate it with the **noexcept** specifier. This allow the compiler to avoid setting up the exception trapping mechanism when the function is called, increasing efficiency.

As for the move operations, being exception free allows to optimize certain operations on object stored in standard containers. So its good practice have exception free move operations and indicate it.

The noexcept specifier is not compulsory, your code will work also without

## What do the synthesized operators do?

```
// a simple class with all synthetis
class Foo{
  public:
  double aMethod();
  private:
  std::vector<double> x ={3.4,5.0};//in-class initialization
}
Foo createFoo();// a function that creates a Foo object
...
Foo foo; // Foo::Foo()
Foo toto=foo;// Foo::Foo(Foo cont&)
foo = toto;// Foo:operator=(const Foo &)
Foo pippo{createFoo()};//Foo::Foo(Foo &&)
foo = createFoo();// Foo:operator=(Foo &&)
toto  = std::move(foo);//Foo:operator=(Foo &&)
// foo.x is now an empty vector!
```

## Rules for implicit methods

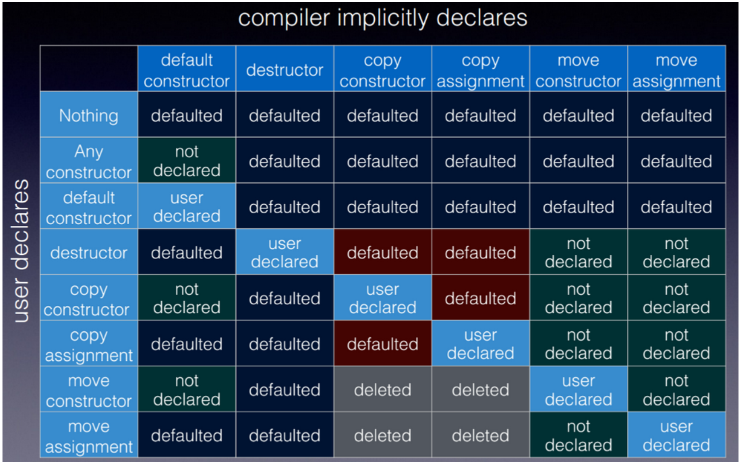
The synthetic methods are defined by the compiler automatically only if the user do not declare them explicitly.

Moreover:

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

If you want to "resurrect" implicit methods when they are not generated automatically, use the keyword **default**. If you do not want them to be generated use the keyword **delete**.

# The rules for the generation of implicit methods



compiler implicitly declares

| user declares | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---|---|---|---|---|---|---|
| Nothing | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| Any constructor | not declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| default constructor | user declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| destructor | defaulted | user declared | defaulted | defaulted | not declared | not declared |
| copy constructor | not declared | defaulted | user declared | defaulted | not declared | not declared |
| copy assignment | defaulted | defaulted | defaulted | user declared | not declared | not declared |
| move constructor | not declared | defaulted | deleted | deleted | user declared | not declared |
| move assignment | defaulted | defaulted | deleted | deleted | not declared | user declared |

Not declared: not declared by the compiler, but the user may recall it using the `default` keyword. Deleted: not provided by the compiler, but the user may still define one. See here for details.

## Another example

```
class Foo{
  public:
  Foo(int i):i{i}{}; // constructor taking an int
  Foo()=default; // but I want also the implicit default one
  // This class has no copy-assignment operator
  Foo& operator=(const Foo &)=delete;
  // And consequently neither move-assignement
  private:
  int i=0;// I give a default value
};
...
Foo a; // calls synth. default const
Foo b(3)// calls Foo(int)
a=b; // SYNTAX ERROR: no copy-assign.
```

## What do the synthesized methods do?

The synthesized default constructor constructs all non-static variable members of the class (using default c. or the appropriate one if variable is initialized in-class) in the order they are declared in the class.

The other methods (a part the destructor) ...operate similarly..., just changing the verb: a copy-constructor copy-construct the non-static member variables, 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.

Static variable members are constructed "at compile time".

## The rule of five/zero

- If a class requires a user-defined destructor or a user-defined copy or move constructor, or a user-defined copy or move assignment operator, it almost certainly requires all five.

If you can avoid defining a special operator prefer using the synthesized ones.

```
struct Foo{
  Foo(){};// why are you doing it?
  // It is equivalent to Foo()=default, but why do you need it?
...
```

When a base class is intended for polymorphic use, its destructor may have to be declared public and virtual. This blocks implicit moves (and deprecates implicit copies), and so the special member functions have to be declared as defaulted (if you want them).

## Default constructor

A default constructor is any constructor that may take no arguments. A class with a default constructor is called default constructible.

```
class FirstClass{
 public:
 // declaration of a user defined default const
  FirstClass(){// does something};
  ...};

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

Only if a class is default constructible you can do

```
MyClass m; // or
MyClass b{};
```

## 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 are initialized by their default constructor. Members are initialized in the order declared in the class, irrespectively of the order in the initialization list. The body may contain other actions, different from initialization. In the scope of the body all members have been constructed and are available. 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 { }).

## Delegating constructors (part I)

You can put in the initialization list a previously defined constructor. It helps avoiding code repetition.

```
class Foo{
public:
Foo(int i=0);// A constructor taking a int
Foo(int i, double x):Foo(i)
  {... /*do something/* }
}
```

I use Foo::Foo(**int**) inside the initialization list of Foo::Foo(**int**,**double**).

## 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:

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

An important note: If the class is used as a base class the destructor should be declared `virtual`!.

Another note: Use containers or unique pointers to store dynamic data owned by the class. You avoid the fuss of writing a destructor (and possible headaches).

## Copy constructor

The copy constructor is called when we initialize an object by copying from another object of the same type: MyMat0 a(b), or MyMat0 a=b calls the copy constructor of MyMat0.

You normally need to write your own copy constructor if you store a pointer to dynamically allocated data and want a deep copy (i.e. to copy the pointed data and not the pointer itself). But again, if you use a standard container you are spared the problem.

A class with an accessible copy constructor is called copy-constructable.

## When do you need to write your own copy constructor?

Typically, it happens when you handle memory dynamically with pointers, or when your class you aggregate resources through pointers. For instance, let assume that MyMat0 is a matrix that stores the data via a unique pointer `unique_ptr<double[]> M_data`. We can do

```
MyMat0(MyMat0 const & m):nr(m.nr),nc(m.nc)
{
M_data=make_unique<double[]>(m.nr*m.nc);
for (unsigned i=0;i<m.nr*m.nc;++i)M_data[i]=m.M_data[i];
}
```

Note: If you do not write the copy constructor you have a shallow copy, probably not what you want! If you use a `std::vector<double>` to store the data you are saved the fuss!

## Copy-assignment

Again, you normally need to write your own copy-assignment if you aggregate resources through pointers. It is important to ensure correct behavior when you do a=a, which should be a no-operation. And remember to return a reference to the object, in order to be able to do a=b=c.

```
MyMat0 & operator=(MyMat0 const & m){
if (this!=& m); // handles a=a
{
 M_data.reset(new double[m.nr*n.nc]);// release old data
 for (unsigned i=0;i<m.nr*m.nc;++i)M_data[i]=m.M_data[i];
 ...
}
return *this;
}
```

A class that has a copy-assignment operator is copy-assignable, and you can do a=b on objects of the class.

## Move-constructor and move-assignment

In this course we skip move semantic.

I only anticipate here that you need to define your own move operators only if your class is handling big data and you want to define how the data can by "moved" between objects without unnecessary copies. If you use standard containers to store your data the synthesized operators are fine, since containers know how to move the elements saving memory:

```cpp
std::vector<double> v;//a vector of 3 doubles
v.fill(1000,3.0);// filled with 1000 values
std::vector<double> b=std::move(v);// move cosntructor is called
// b.size()=1000 while v is empty: v.capacity()=0!
```

## 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 implicit or user declared methods.

If the implicit methods are good for you, there is no need to define your own. If you want to "resurrect" a implicit method that has been hidden because of the rules previously described, use the **default** keyword.

If, for some reason, you want to make sure not to have some of them, use **delete**.

```
struct pippo
{
pippo()=default;// use the sinthetised default constructor
pippo(const pippo &)=delete; // No copy allowed!
...};
```

## Explicit constructors and implicit conversions

Any constructor that may take a single argument (including the possible effect of defaulted parameters) defines an implicit conversion unless declared explicit. Sometimes it's what you want, sometimes it's not...in the latter case use the **explicit** specifier.

```
struct Foo
{
  Foo(int);
...
struct Foo2
{
   explicit Foo2(int);
...
Foo a(10); // I create an object of Foo type
a=9;// OK int->Foo conversion
Foo2 b(10);// An object of Foo2 type
b=9;// Error!!
b=static_cast<Foo2>(9);// ok, explicit cast
```

## Casting operator

You want to convert a Foo to an int

```
struct Foo
{
    operator int(){return i;}
    int i;
};
struct Foo2
 {
     explicit operator int(){return i;}
     int i;
 };
 ..
 Foo a{3}; // Foo is an aggregate
 int b = a;// OK Foo->int
 Foo2 c{9};// OK
 int z = c// ERROR!
 int z = static_cast<int>(c);//OK explicit cast
```

Use with care!

## Constant and non constant methods

A method that does not alter the state of a class must be declared **const**, like the method get() in a previous example. Constant objects can call only constant methods!

```
class MyClass{...
  double get(int);// Should be const!
  ...};
void afun(Myclass const & m, int i){
  auto g=m.get(i)// ERROR m is const}
```

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

**const** is your friend, use it!.

## The inline directive

**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];}
   ...
 };
 // Definition (inline not needed here)
 double foo::medhod1(int i){...}
```

Definitions of inlined member functions should be given in the header file. Methods defined in-class are implicitly **inline** (but is not an error being redundant).

## Aggregates

An aggregate is a particularly simple class type:

- Only public non-static data members;
- No user declared constructor;
- No virtual member functions;
- Inheritance allowed but only public and from a base class that is an aggregate.

It is customary to use **struct** to define an aggregate. Note that std::array, std::pair, std::tuple and C-style fixed size array like **double** v[10] are all aggregates.

## Aggregates features

Aggregates have some nice features

```
struct NewtonOptions{
double tolerance=1e-8;
unsigned int maxIter=100;
};
```

- aggregate initialization:

```
NewtonOptions aa{.001,300};
NewtonOptions bb ={5e-4,60};
```

- structured binding:

```
auto [tol,maxIt]=aa; // create a copy
auto & [tol2,maxIt2]=bb // creates a reference
```

## A note: brace initialization does not allow narrowing

Brace initialization does not allow narrowing (loss of precision) on POD types:

```
double x=9.8;
...
int sum1{x};// ERROR! doubles->int conversion is narrowing
int sum2(x);// OK, will be truncated to integer part!
std::cout<<sum2;// prints 9
float y;
double w;
...
double z{y}; //Ok, no narrowing here.
float k{w};// Error: narrowing!
```

## Beware!

Sometimes the meaning can be different!. This problem arises in several dynamic std containers, since their public interface was defined before C++11.

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

This is due to the old design of std::vector. But here there is nothing one can do if backward compatibility has to be ensured.

## How classes collaborate with each other

Classes introduce functionalities that are normally related each other. Different type of "collaboration" among classes are possible.

Let's start with the strongest way to relate classes: creating a hierarchy with inheritance and polymorphism.

## Hierarchy of concepts

Very often mathematical concepts (but not only) are hierarchical: a triangle and a square are polygons; a P1 and P2 are finite elements, normal and binomial are distributions.

It means that we expect to be able to carry out on triangles and squares operations that are in fact common to all polygons.

This type of relationship is termed "is-a" is expressed in C++ through public inheritance and polymorphism.

However inheritance and polymorphism are related but independent concepts: you need inheritance to apply polymorphism, but you may use inheritance without polymorphism.

Inheritance is a indeed also possible way to implement more complex concepts starting from simpler ones (composition via public inheritance).

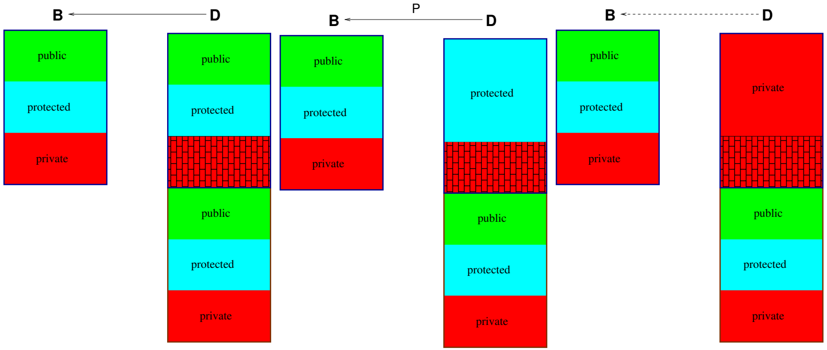
## Inheritance

Inheritance can be public, protected, private. The most commonly used is the public one, but let's explain the different choices.

- Public: Get everything from my parent, , maintaining its privacy levels, apart its private stuff;
- Protected: Get from my parent its public and protected members, but make all them protected: only I and my siblings will be able to use them.
- Private: Get the public and protected members from my parent, but make them private: only I can use them!.

The most used inheritance is the public one.

## A graphical view



Public inheritance Protected inheritance Private inheritance

## Public inheritance: details

```
class D: public B{....
```

The mechanism of public inheritance is simple:

1. Public and protected members of `B` are accessible by `D`. If they are redefined in `D`, methods with the same name in `B` are hidden, but you can still access them using the qualified name (`B::..`).
2. Public members of `B` are public also in `D`.
3. Protected members of `B` are protected in `D` and thus accessible only by `D` and possible classes publicly derived from `D`.
4. Private members of `B` are inaccessible by `D`.

Public inheritance is the default for structs

## Delegating constructor

In the constructor of a derived class I can call the constructor of the base class. Useful if I need to pass arguments. Otherwise, the default constructor of the base class is used (in which case the base class must be default constructible)

```
class B
{
public:
B(double x){...};
...
};
class D : public B
{
D(int i, double x):B(x) My_i{i}{}
private:
int My_i;
};
```

## Inheriting constructors

Constructors are not inherited, but can be recalled,

```
class B
{
public:
B(double x){...};
...
};
class D : public B
{
using B::B; //Inherits B constuctors
private:
int My_i=10;
};
```

Now, D d{12.0} calls B constructor: d.x is set to 12.0 and d.My_i takes the default value 10.

## Multiple inheritance

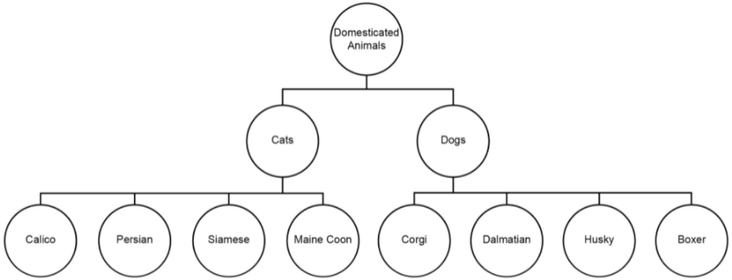
In C++ it is possible to derive from more than one base.
The derivation rules apply to each base class.

Possible ambiguity in the names may be resolved using the qualified name:

```
class D: public B, public C
{
public:
void fun(){
// if both B and C define foo() here I
// resolve the ambiguity using the method of B
auto x=B::foo();
...
}
...
}
```

# Polymorphism

Public inheritance is also the mechanism by which we implement polymorphism: the ability of objects belonging to different class of a hierarchy to operate each one according to an appropriate type-specific behavior.

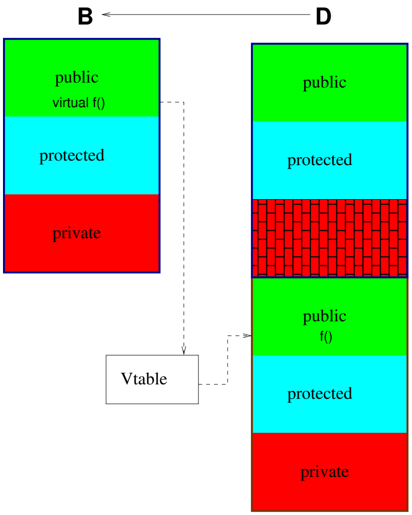


## The mechanism of (public) polymorphism

1. A pointer or a reference to `D` is implicitly converted to a pointer (reference) to `B` (upcasting). A pointer or a reference to `B` can be explicitely converted to a pointer (reference) to `D` if `B<-D` (downcasting), using `static_cast` (statically) or `dynamic_cast` (dynamically), if possible.
2. Methods declared virtual in `B` are overridden by methods with the same signature in `D`.
3. If `B* b=new D` is a pointer to the base class converted from a `D*`, calling a virtual method (`b->vmethod()`) will in fact invoke the method defined in `D` (it applies also to references).

Note: overridden virtual methods must have same return type, with one exception: a method returning a pointer (reference) to a base class may be overridden by a method returning a pointer (reference) to a derived class.

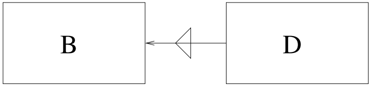
# Public inheritance and polymorphism



## Is-a relationship

Public inheritance polymorphism (simply called polymorphism) must be used only when the relation between base and derived class is an is-a relation: the public interface of the derived class is a superset of that of the base class.

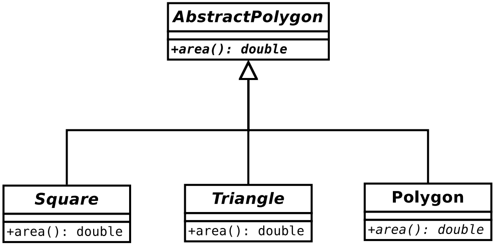
It means that one should be able to use safely from an object of the derived class any member of the public interface of the base.



PROMISE NO LESS, REQUIRE NO MORE (H. Sutter)

This implies that the base class must define the public interface common to all members of the hierarchy.

## An example



See the example in Polygon/Polygon.hpp

## Overriding

```
void f (AbstractPolygon const & p){
...
auto a=p.area();
...
}
int main(){
Square s;
...
f(s);
```

I am giving to `f()` a `Square` as argument, passed by reference. It is possible since we have the implicit conversion
Square& −> AbstractPolygon&.
Yet, `p.area()` in `f()` will now call the method defined in `Square` and not that of `AbstractPolygon`, since `area()` is a *virtual method* of `AbstractPolygon`.

## Polymorphisms is applied with pointers or references

This is an error:

```
void f (AbstractPolygon p){
...
p.area();
...
}
int main(){
Square s;
...
f(s);
```

Two compilation errors: a Square is not convertible to an AbstractPolygon.
Moreover, AbstractPolygon is abstract (see later) so I get a compiler error also in the definition of f().

## Overriding and hiding

```
class B{
  public:
  int foo();
  virtual int goo();
  virtual ~B=default();
}
class D; public B{
  public:
  int foo(); // hides
  int goo() override; // overrides
}
...
B b;
D d;
B & bd=d;
d.foo(); /// calls D::foo();
bd.foo();// Calls B::foo()
d.goo();// Calls D::goo()
bd.goo();// Calls D:goo()
```

Avoid name hiding, they are source of confusion.

## Virtual destructor

If you apply polymorphism the destructor of the base class should be defined `virtual`! This is compulsory if the derived class introduces new member variable.

The reason:

```
AbstractPolygon * p=new Square();
...
delete p;
```

In the last line I need to call the Square destructor! If I forgot **virtual** that of AbstractPolygon is called instead, and if Square has added new data mambers we have a memory leak

Note: If you add the warning `-Wnon-virtual-dtor` at compilation time, the compiler issues a warning if you have forgotten a virtual destructor. Most IDE will also warn you if virtual is missing.

## The final and override specifications

We have two possible specifications: final and override.

- final for a method means that the method cannot be overridden;
- final for a class means that you cannot inherit from that class;
- override tells that a method is overriding one of the base class.

Unfortunately, override is not compulsory (because of backward compatibility), but I suggest you to use it!.

Note: The option `-Wsuggest-override` tells the compiler to warn you if an override looks missing.

## Examples of final and override specifiers

```
struct A{
virtual void foo() final;
virtual double foo2(double);
...};
// in a struct inheritance is public by def.
struct B final : A {
void foo(); // Error: foo cannot be overridden:
//  it's final in A
...};
struct C : B // Error: B is final
{...};
```

## A nasty error spotted!

```
struct A{
virtual void foo();
void bar();
...};

struct B : A
{
void foo() const override; // Error:
//Has a different signature from A::foo
void foo() override; // OK: base class contains a
// virtual function with the same signature
void bar() override; // Error: B::bar doesn't
// override because A::bar is not virtual
};
```

The override specification when overriding virtual member functions makes a safer code. USE IT.

## Abstract classes

Sometimes the base class expresses just an abstract concept and does not make sense to have concrete objects of that type. In other words, the base class is meant to define the common public interface of the hierarchy, but not to implement it (at least not in full).

To this purpose, C++ introduce the idea of an *abstract class*, which is a class where at least one virtual method is defined as null.

## Example of abstract class

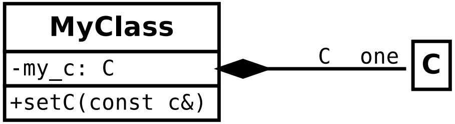
```
class AbstractPolygon{
public:
explicit AbstractPolygon(Vertices v, bool convex=false);
virtual  AbstractPolygon();
bool isConvex() const;
virtual double area() const=0; //null method
// ...
protected:
bool isconvex;
};
```

The method declared null must be overridden in a derived class.

## Composition

A basic but probably the most important collaboration between classes is that of composition, obtained by simply having an object of another class as member. Often, the composed object is kept private, and is used by methods of the class.

In composition, the lifespan of the composed object coincides with that of the composing object. In other words, the composer "owns" its components (sometimes referred as "resources").



Here, MyClass is composed with one C.

## Composition with polymorphic objects

What happens if I want to compose my class with a polymorphic object? The cleaner solution is to use a unique pointer. After all, it implements the "unique ownership" we need! For instance

```
class MyClass{
public:
// A constructor that moves the polygon
MyClass(std::unique_ptr<AbstractPolygon> p):my_p{std::move(p)}{}
// example of a setter
void setPoly(std::unique_ptr<AbstractPolygon> p){my_p.reset(p);}
...
private:
std::unique_ptr<AbstractPolygon> my_p;
..
}
```

`std::move()` is used to move the pointer, the method `reset()` replaces the pointer, deleting the possibly already stored polygon. That's look fine. But...Houston, we have a problem!

## The class is not copy-constructible/assignable

MyClass is move-constructible and move-assignable, but is not copy-constructible nor copy-assignable, since unique pointers can be moved but cannot be copied.

```
MyClass c{std::make_unique<Square>()};
//so far fine, c i composed with a  Square
MyClass b=c; // ERROR!
```

The copy construction of b fails!. Syntax error!

We should strive to have copy-constructible and copy-assignable classes as far as possible. How can we do it? Moreover, I need a deep copy: I do not want to copy the pointer, but the object pointed to!.
Unfortunately, doing it safely and in a scalable way is not so obvious.

## Clonable classes (virtual constructor)

It can be useful if a polymorphic object is able to clone itself. It's a way to implement the Prototype design pattern. Let's apply it to our Polygons:

```
class AbstractPolygon{ // base class
  ...
  protected: // but public is also ok
  virtual std::unique_ptr<AbstractPolygon> clone() const=0;
  };

class Square: public AbstractPolygon{
...
protected: // but public is also ok
 std::unique_ptr<AbstractPolygon> clone() const override
     {return std::make_unique<Square>(*this);}
};
```

std::make_unique<Square>(*this) copies the current object (*this), which is a Square, and the copy is handled by a unique pointer to Square that is then converted to a unique pointer to AbstractPolygon (possible since Square derives from AbstractPolygon), and returned.

## Why clonable classes are useful

Now I can do

```
class MyClass
{
  public:
  // copy constructor
  MyClass(MyClass const & c):my_p{c.my_p->clone()}{};
  // assignement
  MyClass & operator = (MyClass const & c)
  {
        if (&c != this) my_p=c.my_p->clone();
        return *this;
  }
  // but I have to bring back move operators!
  MyClass(MyClass&&)=default;
  MyClass & operator = (MyClass&&)=default;
};
```

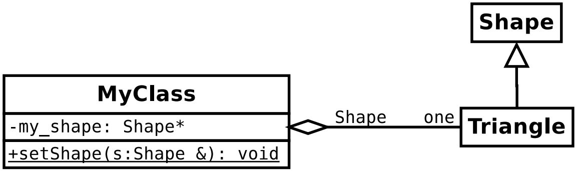
Et voilà: the class is copy constructible/assignable and copy operations make a deep copy. Houston, problem solved!

For the nerds, a better solution is the `PointerWrapper` in Utilities/CloningUtilities.hpp

## Aggregation

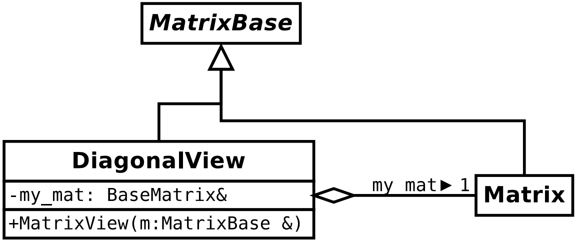
A second important type of collaboration is aggregation. The class stores a (classic) pointer or a reference to a (possibly) polymorphic object external to the class.

In the case of a reference, the latter must be initialized in the constructor and obviously you cannot reassign it The lifespan of the referenced object is independent from that of the containing one, and the former should outlive the latter. If the aggregated object is used to define part of the implementation of the class it is called a policy.



## Views (Proxies)

A view (or proxy) is a particular type of aggregation (usually performed with a reference), whose role is to enable to access members of the aggregating object using a different (usually more specialized) interface, for a particular use. For instance, accessing a general matrix as a diagonal matrix. It is a particular instance of the decorator design pattern.



## Implementation details

```
class Matrix: public MatrixBase{
public:
//! Returns A_ij
double & operator()(int i, int j);
...
};

class DiagonalView: public MatrixBase {
public:
 DiagonalView(Matrix & m):MM(m){}
 double & operator()(int i,int j){
 return i==j ? MM(i,i):0.0;}
 ...
private:
 Matrix &MM;
};
```

This is a working and simple view.

## Friendship

A class can be declared friend of another class. This way the friend class may access its private member.

```
class A{
...
friend class B;
private:
double x;
};
class B{
void f (A & a)
{ a.x=10.;// I can do it!
}
};
```

Friendship is a strong relationship: a friend class may be considered as part of the implementation. Use it only when necessary.

## Friend functions

Another type of friendship is made with functions. It gives those functions the possibility to access the private member of the friend class. Sometimes it is used to speed-up access or to do special operations that require access to the private member, a common example is the streaming operator.

```
class A{
...
friend std::ostream & operator<<(std::ostream & out, A const &);
...
};
// now operator<< can access private members of A.
```

Use only if necessary. And remember: "friends of my parents are not my friends" applies also to classes, friendship is not inherited.