Advanced Methods for Scientific Computing (AMSC)
Lecture title: Smart pointers

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

The need of smart pointers

In C and old C++ pointers are use for two different purposes:

1. Point to an existing resources, with no ownership:

```
double a=10;
double * a_p=&a;
```

2. Owning the resorce they point to

```
double * a_p= new double; // acquire the resource
...
delete a_p;// free the resource
```

The second case is problematic: when and where to release the resource? In complex code it is easy to create situations where you have memory leaks, or, even worse segmentation faults.
Some languages have tackled the problem implementing computationally expensive garbage collection mechanisms. In modern C++ we have smart pointers to ease the problem.

Smart pointers

We have:

unique_ptr<T>	Implements unique ownership. The resource is released (deleted) when the pointer goes out of scope
shared_prt<T>	Implements shared ownership. The resource is released when the last shared pointer goes out of scope
weak_prt<T>	A non-owning pointer to a shared resources. For special usage.

They all require the `<memory>` header.

The need of unique_ptr

Let's look at this example, where Polygon is the base class of several polygons:

```
class myClass{
setPolygon(Polygon * p){my_polygon=p;}
...
private:
Polygon * my_polygon;// Polymorphic object
};
// A Factory of Polygons
Polygon * polyFactory(std::string t){
switch(t){
case ''Triangle'': return new Triangle;
case ''Square'':      return new Square;
..
default: return nullprt;}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

A poor design

This design is prone to errors.

First of all objects of type MyClass have now to take care of handling of the resource my_polygon (whenever you see a **new**, always ask yourself: "where is the **delete**?".)

We have to build constructors, destructor, assignment operator very carefully, and to account for possible exceptions to avoid memory leaks and dangling pointers.

There is always the risk that the user calls polyFactory and forgets to delete the returned pointer when it is required, causing a memory leak that may be very difficult to detect!

The version with unique_ptr

```
class myClass{
setPolygon(unique_ptr<Polygon> p){my_polygon=std::move(p);}
...
private:
unique_ptr<Polygon> my_polygon;
}
// A Factory of Polygons
unique_ptr<Polygon> polyFactory(std::string t){
switch(t){
case ``Triangle'': return std::make_unique<Triangle>();
case ``Square'':   return std::make_unique<Square>();
..
default: return unique_ptr<Polygon>();// null ptr
}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

Complete example in SmartPointers

How a unique_ptr<> works

A unique_ptr<T> serves as unique owner of the object (of type T) it refers to. The object is destroyed automatically when its unique_ptr gets destroyed.

It implements the * and the -> dereferencing operators, so it can be used as a normal pointer.

But it can be initialized to a pointer only through the constructor:

```
std::unique_ptr<int>up=new int; // ERROR!
std::unique_ptr<unt> down(new int); //OK!
```

or, much better, using the utility std::make_unique<T>()

```
auto p=std::make_unique<Triangle>();
```

The default constructor produces an empty (null) unique pointer. You can check if a unique_ptr is empty just with **if**(prt) (smart pointer are also contextually convertible to bool).

Moving a unique_ptr around

Unique pointers cannot be copied, for abvious reasons, but they can be moved (for details wait the lecture on move semantic).

Ownership can be transferred using the std::move utility:

```
unique_ptr<double> c(new double);
unique_ptr<double> b;
b= std::move(c);
```

Now c is empty and b points to the double originally held by c.

Dealing with C-arrays

By default a unique_ptr calls **delete** for an object of which it loses ownership. Unfortunately, this will not work properly if the object is an array. However, there is a specialization that works for arrays:

```
unique_ptr<string> up(new string[10]);// A SERIOUS ERROR!!
unique_ptr<string[]> up(new string[10]);// OK
auto up=make_unique<string[]>(10);     // EVEN BETTER!
```

Suggestion: try to avoid C-style arrays. Use std::array, or std::vector, for which you do not need all this fuss.

Main methods and utilities of unique_ptr

swap(ptr1,ptr2)	Swaps ownership
pt1=std::move(pt2)	Moves resources from `pt2` to `pt1`. The previous resource of `pt1` is deleted. `pt2` remains empty.
pt.reset()	Resource is deleted. `pt` is now empty.
pt.reset(pt2)	as pt=std::move(pt2)
pt.release()	returns a standard pointer. It releases the resource without deleting it. `pt` is now empty.

unique_ptrs can be stored in a standard container:

```
vector<unique_ptr<Polygon>> polygons;
```

Shared pointers

While unique_ptr do not cause any computational overhead (they are just a light wrapper around an ordinary pointer), shared pointers do, so use them only if it is really necessary.

For instance you have several objects that "refer" to a resource (a Matrix, a Mesh...) that is build dynamically (and maybe is a polymorphic object). You want to keep track of all the references in such a way that when (and only when) the last one gets destroyed the resource is also destroyed.

To this purpose you need a shared_ptr<T>. It implements the semantic of "clean it up when the resource is no used anymore".

See the example in SmartPointers

The std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference (here reference is used in a generic sense) to an object that is managed by std::shared_ptr.

It must be converted to std::shared_ptr in order to access the referenced object.

It may be used to test if a resource associated to a shared_ptr has been deleted, in a thread-safe way.

However, it's usage is rather special and we omit the details here. You may find them in any good reference.

References vs pointers

There is often confusion on reference and pointers. Let's try to make things clear

- A pointer is a variable, it uses memory (normally 8 bytes) to store a memory address. When dereferenced it returns the value associated to the address it points to (provided it is a valid address). But for the rest, it is like any other "first class object": it can be reassigned, changed etc.;
- You can have null pointers, i.e. pointers that points to nothing, which you can assign to the address of an object later on;
- A pointer type is indeed a different "base type" than that of the pointed object. A `double*` is a complete different type than `double`.
- A reference is just an alias to an existing object. A reference to a variable gives a "secondary name" to that variable. You cannot have unbound references, nor you can bind a reference to another object. Reference and referenced object are bound for life! But a reference cannot outlive the referenced object.
- A reference does not define a different "basic type", sometimes it is said that it is an adornment to a type. A `double&` behaves exactly as a `double`.