

Advanced Programming for Scientific Computing (PACS)

Lecture title: Smart pointers and references

Luca Formaggia

MOX

Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

RAII

Resource **A**cquisition **I**s **I**nitialization is a big karma in C++. It basically means that an object should be responsible for the creation and destruction of the resources it owns. A terrible name, also the inventor admits it. Examples:

```
double * p = new double[10];
```

This is not RAII compliant! Who is destroying the resources pointed by p?? **You have to take care of it!**

```
std::array<double,10> p;
```

This is instead RAII compliant!. p is in charge of creating 10 doubles and of destroying them when it goes out-of-scope.

C++ smart pointers are another tool to implement RAII.

Pointers, smart and not

In modern C++ we use different types of pointers

- ▶ Standard pointers. Use them only to **watch** (and operate on) an object (resource) **whose lifespan is independent from that of the pointer** (but not shorter);
- ▶ Owning pointers. Also called smart pointers. **They control the lifespan of the resource they point to.** They are of 2 kinds:
 - ▶ `unique_ptr`, with **unique** ownership of the resource. The owned resource is destroyed when the pointer is destroyed (goes out of scope);
 - ▶ `shared_ptr` with **shared** ownership of a resource. The resource is destroyed when the last pointer owning it is destroyed.

Smart pointers implement the RAII concept. For just addressing a resource (maybe polymorphically) use **ordinary pointers**.

Smart pointers

We have:

<code>unique_ptr<T></code>	Implements unique ownership . The resource is released (deleted) when the pointer goes out of scope
<code>shared_ptr<T></code>	Implements shared ownership . The resource is released when the last shared pointer goes out of scope
<code>weak_ptr<T></code>	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 nullptr;}
...
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<int> 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 by testing **if**(prt.) or simply (but smart pointer are also contextually convertible to `bool`).

Moving a `unique_ptr` around

Unique pointers cannot be copied, for obvious 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`, less problems, less fuss.

Main methods and utilities of `unique_ptr`

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

`unique_ptr`s 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, its usage is rather special and we omit the details here. You may find them in any good reference.

The use of references

References are very important in C++ programming and essential in C++ for scientific computing. Their main usage is as parameter (and sometimes return type) of a function. Passing a reference instead of a value help saving memory, since you refer to an existing object, and also (if the reference is not const) it provides an alternative way to return data from the function.

The initialization of a reference is called **binding**, and the binding rules are **an essential** feature of references, particularly when coupled with function overloading.

Reference semantic in std containers

Std containers can hold only “first class” objects, but not references (they are not first class objects, but alias to already existing ones):

```
vector<unique_ptr<Polygon>> a; //OK  
vector<Polygon*> a; //OK  
vector<Polygon &> n; //ERROR!
```

Reference semantic in std containers

However, we have a way to store objects with the same semantics as references in a container. You need to use `std::reference_wrapper` defined in the header `<functional>`

```
Point p1(3,4);  
Point p2(5,6);  
std::vector<std::reference_wrapper<Point>> v;  
v.push_back(p1);  
v.push_back(p2);  
v[0].setCoord(7,8); // changes coordinates of p1
```


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 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!
- ▶ 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`.