MODERN C++ PROGRAMMING COURSE – SMART POINTERS

Teacher: Nicolò Genesio (@Nicogene)

PREFACE: RULE OF 5

ISTITUTO ITALIANO
DI TECNOLOGIA

The **rule of five** is a modern expansion of the **rule of three**.

These functions are usually required when a class manually manage a dynamically allocated resource, and so all of them must be implemented to manage the resource safely.

PREFACE: RULE OF 5

In addition, the *rule of five* identifies that it usually appropriate to also provide the following functions to allow for optimized copies from temporary objects:

- move constructor
- move assignment operator

The **rule of zero** is the complementary of the rule of five, if you don't define none of these 5 functions for a specific class, they will be automatically generated by the compiler in function of the copiability or movability of its members.

1.INTRODUCTION

Dynamic allocations are useful and in some cases necessaries but they arise several questions, and troubles:

- Who allocates/deallocates the memory?
- How the memory is allocated/deallocated?
  - `new`? `malloc`? `delete`? `delete []`? `free`?
- Have I left some memory leaked?

1.INTRODUCTION

ISTITUTO ITALIANO DI TECNOLOGIA

Usually these questions are addressed by the documentation, it is necessary to specify explicitly the memory management and ownership.

A very good documentation preserves from API misuse and memory troubles when using memory dynamically allocated. E.g.:

```
class Widget {
public:
    ~Widget() {
        delete m_layout;
    }

    //! The ownership of @p layout is transferred to the Widget.
    void setLayout(Layout *layout) {
        m_layout = layout;
    }

private:
    Layout* m_layout = nullptr;
};

//! @return The created layout, whose ownership is transferred to the caller
Layout* createLayout() {
  return new Layout();
}
```

I am forced to write the destructor then violating the rule of 0.

From the signature it is not clear the memory ownership.

Documentation saves the day here

1.INTRODUCTION

The dynamic allocations and memory management is a very powerful tool but risky and that gives a lot of headaches.

For this reason, languages both interpreted(Python, Lua, Ruby etc) and compiled(Java, C# etc) "younger" than C and C++, prefer to not let manage the memory to the user.

C++ with the new standards introduced with c++11 and 14, moved in this direction introducing the smart pointers.

1.INTRODUCTION

Smart pointers are classes that can be used transparently as pointers but additionally provide additional semantics to help in memory management.

As many of new semantics introduced in c++11, they allow to write self-documented, avoiding any misunderstandings or misuses in memory ownership.

They are:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

II. UNIQUE_PTR

ISTITUTO ITALIANO
DI TECNOLOGIA

`std::unique_ptr` is used to own *exclusively* a memory area

When a **`std::unique_ptr`** is destroyed, it will *automatically* free the memory.

`unique_ptr` cannot be copied, but it can be *moved*, in this sense the ownership of the memory is moved to another pointer.

The default function called during the destruction is **`delete`**, but it can be defined a different custom deleter one as 2nd argument of the template.

II.UNIQUE_PTR

The **`unique_ptr`** it can be initialized in two ways

- `std::unique_ptr<MyClass> ptr = new MyClass();`

Or using **`make_unique()`** (available in `c++14`)

- `std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>();`

`make_unique` forwards the arguments passed to it to the constructor of **`MyClass`**, less memory allocation is involved and allows to not write the keyword **`new`**, that make people worry if a delete has to be called somewhere.

II.UNIQUE_PTR

ISTITUTO ITALIANO
DI TECNOLOGIA

By using **`std::unique_ptr`** at the interface level, the ownership model is clearly documented:

```
class Widget
{
public:
    void setLayout(std::unique_ptr<Layout> layout)
    {
        m_layout = std::move(layout);
    }

private:
    std::unique_ptr<Layout> m_layout;
};

std::unique_ptr<Layout> createLayout(bool switch)
{
  if (switch) return std::make_unique<Layout>("a");
  else return std::make_unique<Layout>("b");
}
```

In this case the rule of 0 is not violated, there is no need to implement the destructor.
We let the compiler generate copy or move operators and constructors.

Quick question:

Is this class copyable?Is it Movable?

It is movable but <u>not</u> copyable. The copy assignment operator and copy constructor are <u>not</u> generated by the compiler.

Here the pointer is passed by value, then is copied?
But the unique_ptr can't be copied…
What then?

A temporary variable created inside the scope of a function are returned by <u>move by default</u>, if the object is movable, it is copied otherwise.

III.SHARED_PTR

`std::shared_ptr` is a reference-counting class which holds a pointer.

It represent the shared memory model.

When the *last* reference to the pointer is deleted, the pointer held is deleted.

It is composed by a pointer and a reference counter that keeps how many pointers are referring to the same memory area.

It is initialized using the method **`make_shared`** that is similar to **`make_unique`**, the arguments are forwarwed to the constructor of the class.

E.g. :

```
shared_ptr<Employee> e = make_shared<Employee>("Knopf", "Jim", 1950);
```

III.SHARED_PTR

The **`shared_ptr`** is movable and copyable.

The move semantics makes the new pointer point to the memory, the moved one is set to **`nullptr`** and the reference count is NOT incremented.

On the other hand the copy semantics increment the reference counter.

The multi-thread safeness is not guaranteed when accessing the shared memory. On the other hand the increment of the internal reference count is multi-thread safe, since the refcount is an atomic operation.

III.SHARED_PTR

ISTITUTO ITALIANO
DI TECNOLOGIA

As the **`unique_ptr`**, the **`shared_ptr`** can call a custom deleter.

Example:

```
void arrayDeleter(int* array) { delete[] array; }
std::shared_ptr<int> arrayPtr{new int[30], arrayDeleter};

std::shared_ptr<int> array { new int[66], std::default_delete<int[]>{} };

FILE *fh = fopen("myfile.txt", "r");
std::shared_ptr<FILE> fhPtr { fh,[ ] (FILE* f) { fclose(f); } };
```

`shared_ptr<int[]>` exists only in c++17, then there without specificing the custom deleter a simple delete is called since it is `shared_ptr<int>` causing memory leak.

The problem of the right deleter can be addressed also with the default deleters.

III.SHARED_PTR

```cpp
#include <memory>
#include <iostream>
#include <string>

class Data {
public:
    ~Data() {
        std::cout << "I'm being deleted!" << std::endl;
    }
    void echo(const std::string &context) {
        std::cout << context << ": " << this << std::endl;
    }
};


/// A function which allocates a Data pointer and returns it.
/// The ownership of the pointer is handled by std::shared_ptr.
std::shared_ptr<Data> makeData()
{
    // Pattern:
    // 1) Always one per line, always directly into the shared pointer
    // 2) Using make_shared() (see below)
    std::shared_ptr<Data> result{new Data};

    // Alternative below, but we don't use the auto keyword here to make it
    // clearer where the shared_ptrs are used.
    //auto result = std::make_shared<Data>();

    result->echo("within makeData");

    return result;
}

void useData()
{
    std::shared_ptr<Data> data = makeData();
    data->echo("returned from makeData");
    std::cout << "the pointer held is: " << data.get() << std::endl;

    // Just for show off:
    std::shared_ptr<Data> data2 = data;
    data->echo("data after copy");
    data2->echo("data2 after copy");

    // The actual Data object gets deleted when the last shared pointer pointing to
    // it goes out of scope, i.e. when both data and data2 get destroyed.
}

int main(int, char *[])
{
    useData();
    return 0;
}
```

Ref count? 1

Ref count? 1

Ref count? 1

Ref count? 2

ISTITUTO ITALIANO DI TECNOLOGIA

IV. WEAK_PTR

ISTITUTO ITALIANO
DI TECNOLOGIA

`std::weak_ptr` is a non-owning pointer to an object.

Merely observes resource, so it does not change refcount.

Useful to break `std::shared_ptr` cycles, have non-owning cache of objects, etc.

`lock()` returns `std::shared_ptr` to resource.

```
std::weak_ptr<Resource> wptr;
void f() {
    shared_ptr<Resource> sptr = wptr.lock();
    if (!sptr)
    return; // resource was deleted
~~~
}

{
    auto sptr = make_shared<Resource>(...);
    wptr = sptr;
    f (); // Will be able to lock resource
}
f(); // Won't be able to lock resource
```

The name is unhappy, it doesn't lock anything, checks that the resource exists and it that case give the ownership in the form of `shared_ptr`

Exiting from the scope the memory is freed since no pointers are pointing to the memory

ISTITUTO ITALIANO
DI TECNOLOGIA

Thanks for the attention!

Any question?