Unique Pointers

In this lesson, we will examine the first type of smart pointer – the unique pointer. It limits access to its resource, thereby maintaining smart pointer's privacy.

WE'LL COVER THE FOLLOWING

Introduction

An std::unique_ptr automatically and exclusively manages the lifetime of its resource according to the RAII idiom. `std::unique_ptr` should be our first choice since it functions without memory or performance overhead.

std::unique_ptr exclusively controls its resource. It automatically releases the resource if it goes out of scope. No copy semantics are required, and it can be used in containers and algorithms of the Standard Template Library. `std::unique_ptr` is as cheap and fast as a raw pointer when no special delete function is used.

Characteristics

Before we go into the usage of `std::unique_ptr`, here are its characteristics in a few bullet points.

The `std::unique_ptr`:

- can be instantiated with and without a resource.
- manages the life cycle of a single object or an array of objects.
- transparently offers the interface of the underlying resource.
- can be parametrized with its own deleter function.
- can be moved (move semantics).
- can be created with the helper function `std::make_unique`.

Replacement for `std::auto_ptr`

> ⚠ **Don't use std::auto_ptr**
>
> Classical C++98 has a smart pointer `std::auto_ptr`, which exclusively addresses the lifetime of a resource. `std::auto_ptr` has a conceptual issue. If we implicitly or explicitly copy an `std::auto_ptr`, the resource is moved. Therefore, rather than utilizing copy semantic, we have a hidden move semantic, leading to undefined behavior. So `std::auto_ptr` is deprecated in C++11 and removed in C++17. Instead, use `std::unique_ptr`. We can neither implicitly nor explicitly copy an `std::unique_ptr`; only moving it is possible.

The code below will generate an error since we are using the deprecated `auto_ptr`.

```
#include <iostream>
#include <memory>
int main(){
  std::auto_ptr<int> ap1(new int(2011));
  std::auto_ptr<int> ap2 = ap1;              // OK

  std::unique_ptr<int> up1(new int(2011));
  //std::unique_ptr<int> up2 = up1;            // ERROR
  std::unique_ptr<int> up3 = std::move(up1); // OK

  return 0;
}
```

Runtime Error

Methods

These are the methods of `std::unique_ptr`:

Name	Description
`get`	Returns a pointer to the resource.
`get_deleter`	Returns the delete function.
`release`	Returns a pointer to the resource and releases it.
`reset`	Resets the resource.
`swap`	Swaps the resources.

Methods of std::unique_ptr

In the next lesson, we can examine the application of these methods.

Special Deleters

`std::unique_ptr` can be parametrized with special deleters: `std::unique_ptr<int, MyIntDeleter> up(new int(2011), myIntDeleter())`. `std::unique_ptr` uses, by default, the deleter of the resource.

std::make_unique

The helper function std::make_unique was unlike its sibling std::make_shared, and was forgotten" in the C++11 standard. Therefore, `std::make_unique` was added with the C++14 standard. `std::make_unique` enables us to create an `std::unique_ptr` in a single step:

```
std::unique_ptr<int> uniqPtr1= std::make_unique<int>(2011);
auto uniqPtr2= std::make_unique<int>(2014);
```

Using `std::make_unique` in combination with automatic type deduction means typing is reduced to its bare minimum.

> 💡 **Always use std::make_unique.**

If we use

```
func(std::make_unique<int>(2014), functionMayThrow());
func(std::unique_ptr<int>(new int(2011)), functionMayThrow());
```

and `functionMayThrow` throws, we have a memory leak with `new int(2011)` for this possible sequence of calls:

```
new int(2011)
functionMayThrow()
std::unique_ptr<int>(...)
```

Rarely, when we create `std::unique_ptr` in an expression and the compiler optimizes this expression, a memory leak may occur with an `std::unique_ptr` call. Using `std::make_unique` guarantees that no memory leak will occur.

Under the hood, `std::unique_ptr` uses the **perfect forwarding pattern**. The same holds for the other factory methods such as `std::make_shared`, `std::make_tuple`, `std::make_pair`, or an `std::thread` constructor.

Further information

- std::unique_ptr
- std::make_unique
- std::make_shared

Now let's take a look at examples of unique pointers in the next lesson.