# **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
- Characteristics
  - Replacement for std::auto\_ptr
- Methods
  - Special Deleters
  - std::make\_unique
- Further information

## 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.

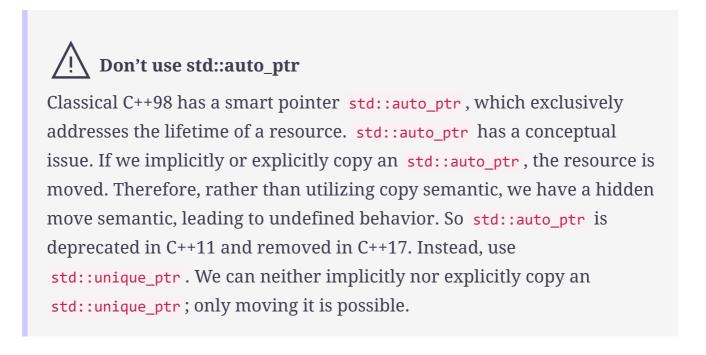
# Characteristics #

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 #



The code below will generate an error since we are using the deprecated auto\_ptr.

Runtime Error

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.