M.x — Chapter M comprehensive review

ALEX SEPTEMBER 28, 2021

A smart pointer class is a composition class that is designed to manage dynamically allocated memory, and ensure that memory gets deleted when the smart pointer object goes out of scope.

Copy semantics allow our classes to be copied. This is done primarily via the copy constructor and copy assignment operator.

Move semantics mean a class will transfer ownership of the object rather than making a copy. This is done primarily via the move constructor and move assignment operator.

std::auto_ptr is deprecated and should be avoided.

An r-value reference is a reference that is designed to be initialized with an r-value. An r-value reference is created using a double ampersand. It's fine to write functions that take r-value reference parameters, but you should almost never return an r-value reference.

If we construct an object or do an assignment where the argument is an l-value, the only thing we can reasonably do is copy the l-value. We can't assume it's safe to alter the l-value, because it may be used again later in the program. If we have an expression "a = b", we wouldn't reasonably expect b to be changed in any way.

However, if we construct an object or do an assignment where the argument is an r-value, then we know that r-value is just a temporary object of some kind. Instead of copying it (which can be expensive), we can simply transfer its resources (which is cheap) to the object we're constructing or assigning. This is safe to do because the temporary will be destroyed at the end of the expression anyway, so we know it will never be used again!

You can use the delete keyword to disable copy semantics for classes you create by deleting the copy constructor and copy assignment operator.

std::move allows you to treat an l-value as r-value. This is useful when we want to invoke move semantics instead of copy semantics on an l-value.

std::move_if_noexcept will return a movable r-value if the object has a noexcept move constructor, otherwise it will return a copyable l-value. We can use the noexcept specifier in conjunction with std::move_if_noexcept to use move semantics only when a strong exception guarantee exists (and use copy semantics otherwise).

std::unique_ptr is the smart pointer class that you should probably be using. It manages a single non-shareable resource. std::make_unique() (in C++14) should be preferred to create new std::unique_ptr. std::unique_ptr disables copy semantics.

std::shared_ptr is the smart pointer class used when you need multiple objects accessing the same resource. The resource will not be destroyed until the last std::shared_ptr managing it is destroyed. std::make_shared() should be preferred to create new std::shared_ptr. With std::shared_ptr, copy semantics should be used to create additional std::shared_ptr pointing to the same object.

std::weak_ptr is the smart pointer class used when you need one or more objects with ability to view and access a resource managed by a std::shared_ptr, but unlike std::shared_ptr, std::weak_ptr is not considered when determining whether the resource should be destroyed.

Quiz time

1. **Explain when you should use the following types of pointers.**

1a) std::unique_ptr

Show Solution

1b) std::shared_ptr

Show Solution

1c) std::weak_ptr

Show Solution

1d) std::auto_ptr

Show Solution

2. **Explain why move semantics is focused around r-values.**

Show Solution

3. **What's wrong with the following code? Update the programs to be best practices compliant.**

3a)

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
 Resource() { std::cout << "Resource acquired\n";
}
 ~Resource() { std::cout << "Resource
destroyed\n"; }
};

int main()
{
 auto* res{ new Resource{} };
 std::shared_ptr<Resource> ptr1{ res };
 std::shared_ptr<Resource> ptr2{ res };

 return 0;
}
```

Show Solution

3b)

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Something; // assume Something is a class that can throw an exception

int main()
{
 doSomething(std::shared_ptr<Something>{ new Something{} }, std::shared_ptr<Something>{ new Something{}
});

 return 0;
}
```

Show Solution

Next lesson

21.1 The Standard Library

Back to table of contents

Previous lesson

M.8 Circular dependency issues with std::shared_ptr, and std::weak_ptr

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest