22.x — Chapter 22 summary and quiz

ALEX[1] JANUARY 6, 2025

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::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 the 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 (javascript:void(0))[2]

1b) std::shared_ptr

Show Solution (javascript:void(0))[2]

1c) std::weak_ptr

Show Solution (javascript:void(0))[2]

1d) std::auto_ptr

Show Solution (javascript:void(0))[2]

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

Show Solution (javascript:void(0))[2]

3. What's wrong with the following code? Update the program 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 (javascript:void(0))[2]

Next lesson
23.1 Object relationships

3

Back to table of contents

4

Previous lesson
22.7 Circular dependency issues with std::shared_ptr, and std::weak_ptr

5

6

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

81 COMMENTS

Newest

AJAY.CHALLA
June 10, 2025 1:31 am PDT

QUIZ

Question::

```cpp
#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 ptr1{ std::make_shared<Resource>() };
    auto ptr2{ ptr1 };

    return 0;
}
```

Last edited 22 days ago by AJAY.CHALLA

0 Reply

Bhaskar Srivastava

January 2, 2025 5:04 am PST

```cpp
#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{ std::make_shared<Resource>() };
    std::shared_ptr<Resource> ptr1{ res };
    std::shared_ptr<Resource> ptr2{ res };

    return 0;
}
```

Is this the correct way to create multiple shared pointers or should it be

```cpp
#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{ std::make_shared<Resource>() };
    std::shared_ptr<Resource> ptr1{ res };
    std::shared_ptr<Resource> ptr2{ ptr1 };

    return 0;
}
```

Asking about line 14-15.

Thanks

0 Reply

Alex Author

Reply to Bhaskar Srivastava [9] January 6, 2025 4:11 pm PST

The quiz solution holds your answer.

0 Reply

EmtyC

December 21, 2024 9:09 am PST

To relationships!

3 Reply

Bromor

November 28, 2023 9:40 am PST

I don't know how the reference count is implemented for shared_ptr, but I've heard about it before getting to the shared_ptr chapter of this tutorial serie, and I feel it helped me a lot to get the process straight away

Maybe a small paragraph about it in the beginning of the lesson could make a difference for some people, if it may give you any idea Alex :)

0 Reply

Strain

Reply to Bromor [10] January 27, 2024 1:52 am PST

I know that Rust's version of it (`Rc`) uses a pointer to an allocated memory, and that memory has both the value and the reference count. Perhaps C++ does something similar.

0 Reply

learnccp lesson reviewer
August 1, 2023 7:21 am PDT

CHAPTER 20 DOWN LES GO

5 Reply

Emeka Daniel
May 26, 2023 6:01 am PDT

Awesome chapter, I understood all but the topic on std::move_noexpect, entirely because I skipped the chapter on expections, I just feel more at home with assert and return codes. Maybe in the nearest future when a company I work for requires the need of expections, then this is my goto site for that.

3 Reply

Alex Author
Reply to Emeka Daniel [11] May 29, 2023 11:49 pm PDT

I think you should read the lesson on std::move_if_noexcept. It provides a way for code to be optimized in cases where exceptions aren't possible. So even if you're only using assert and return codes, you can still get value out of it.

1 Reply

Emeka Daniel
Reply to Alex [12] May 30, 2023 2:17 am PDT

You're right, I shouldn't postpone the topic, who knows when I'll have the time again. Thanks

0 Reply

Emeka Daniel
May 26, 2023 5:58 am PDT

In question 3b, both shared_ptr and unique_ptr suffer from the same problem when passed by value to a function.

I know the compiler is given the freedom to start execution(pardon this term for now, just to make my question a bit more understandable) from either side of a parameter, but isn't the compiler supposed to be given some degree of restriction when it comes to which object to create first in cases like these. Cuz the problem with 3b is exactly that, but it also isn't exactly that, it's unpredictable, the compiler might choose which ever to create first.

0 Reply

Alex Author

Reply to Emeka Daniel [13] May 29, 2023 11:46 pm PDT

The issue really boils down to the fact that the order of evaluation of operands and function arguments is unspecified, and in C++14, the evaluation of expressions of separate function arguments can be interleaved.

I just learned that this issue was fixed in C++17 (as such evaluations can no longer be interleaved), so this is no longer a motivation to use either std::make_unique or std::make_shared. I've removed the quiz question.

0 Reply

Emeka Daniel

Reply to Alex [14] May 30, 2023 2:23 am PDT

Oh, okay.

0 Reply

Stanisław

September 7, 2022 11:12 pm PDT

Hi, one question that I can't understand

1. Who call destructor of object which pointed by pointer ?

```
#include <iostream>
#include <memory>

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

    friend std::ostream& operator<<(std::ostream& out, const Resource& res)
    {
        out << "I am a resource";
        return out;
    }
};

int main()
{
    Resource *pRes{ new Resource };
    std::shared_ptr<Resource> pShrRes1{ pRes };
    {
        std::shared_ptr<Resource> pShrRes2{ pShrRes1 };
        std::cout << "Killing one shared pointer\n";
    }
    //pShrRes1 = nullptr;
    std::cout << "Killing another shared pointer\n";

    return 0;
}
```

Output:

Resource acquired
Killing one shared pointer
Killing another shared pointer
~Resource destroyed~

We go out of main scope and pointer and pointed object was destructed. But now when I uncomment line 'pShrRes1 = nullptr' and nothing pointed to Resource object it will destroyed, but how it know that nothing pointing on it and destroyed themselves (but pointed object don't know nothing about pointer and how many of it pointing, so I think this option is not real), or last pointer will know that nothing else point to object and it(smart pointer) call object destructor("~Resource") to clean up, or maybe there is other mechanism that causes removal of object?

```
#include <iostream>
#include <memory>

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

    friend std::ostream& operator<<(std::ostream& out, const Resource& res)
    {
        out << "I am a resource";
        return out;
    }
};

int main()
{
    Resource *pRes{ new Resource };
    std::shared_ptr<Resource> pShrRes1{ pRes };
    {
        std::shared_ptr<Resource> pShrRes2{ pShrRes1 };
        std::cout << "Killing one shared pointer\n";
    }
    pShrRes1 = nullptr;
    std::cout << "Killing another shared pointer\n";

    return 0;
}
```

Output:

Resource acquired
Killing one shared pointer
~Resource destroyed~
Killing another shared pointer

Thank's for your time and help :)

Last edited 2 years ago by Stanisław

0 Reply

Alex Author

Reply to Stanisław [15] September 13, 2022 10:15 pm PDT

When a std::shared_ptr is destroyed, the destructor for the std::shared_ptr checks if there are any other std::shared_ptr that are sharing the resource. If not, then the std::shared_ptr destructor will destroy the resource.

1 Reply

Huân

August 8, 2022 6:38 am PDT

The last quiz could take my little brain forever to proceed :v

0 Reply

Gabriel

August 5, 2022 5:42 pm PDT

After all, smart pointer ain't that smart with all the exception thingy going on...

2 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/object-relationships/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/
6. https://www.learncpp.com/chapter-22-summary-and-quiz/
7. https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-606102
10. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-590321
11. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-580821
12. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-580980
13. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-580820
14. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-580979
15. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/#comment-572863
16. https://g.ezoic.net/privacy/learncpp.com