

## 22.5 — std::unique\_ptr

👤 **ALEX<sup>1</sup>** ⌚ **AUGUST 20, 2024**

At the beginning of the chapter, we discussed how the use of pointers can lead to bugs and memory leaks in some situations. For example, this can happen when a function early returns, or throws an exception, and the pointer is not properly deleted.

```
1  #include <iostream>
2
3  void someFunction()
4  {
5      auto* ptr{ new Resource() };
6
7      int x{};
8      std::cout << "Enter an integer: ";
9      std::cin >> x;
10
11     if (x == 0)
12         throw 0; // the function returns early, and ptr won't be deleted!
13
14     // do stuff with ptr here
15
16     delete ptr;
17 }
```

Now that we've covered the fundamentals of move semantics, we can return to the topic of smart pointer classes. Although smart pointers can offer other features, the defining characteristic of a smart pointer is that it manages a dynamically allocated resource provided by the user of the smart pointer, and ensures the dynamically allocated object is properly cleaned up at the appropriate time (usually when the smart pointer goes out of scope).

Because of this, smart pointers should never be dynamically allocated themselves (otherwise, there is the risk that the smart pointer may not be properly deallocated, which means the object it owns would not be deallocated, causing a memory leak). By always allocating smart pointers on the stack (as local variables or composition members of a class), we're guaranteed that the smart pointer will properly go out of scope when the function or object it is contained within ends, ensuring the object the smart pointer owns is properly deallocated.

C++11 standard library ships with 4 smart pointer classes: `std::auto_ptr` (removed in C++17), `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. `std::unique_ptr` is by far the most used smart pointer class, so we'll cover that one first. In the following lessons, we'll cover `std::shared_ptr` and `std::weak_ptr`.

---

### std::unique\_ptr

`std::unique_ptr` is the C++11 replacement for `std::auto_ptr`. It should be used to manage any dynamically allocated object that is not shared by multiple objects. That is, `std::unique_ptr` should completely own the object it manages, not share that ownership with other classes. `std::unique_ptr` lives in the `<memory>` header.

Let's take a look at a simple smart pointer example:

```

1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3
4  class Resource
5  {
6  public:
7      Resource() { std::cout << "Resource acquired\n"; }
8      ~Resource() { std::cout << "Resource destroyed\n"; }
9  };
10
11 int main()
12 {
13     // allocate a Resource object and have it owned by std::unique_ptr
14     std::unique_ptr<Resource> res{ new Resource() };
15
16     return 0;
17 } // res goes out of scope here, and the allocated Resource is destroyed

```

Because the `std::unique_ptr` is allocated on the stack here, it's guaranteed to eventually go out of scope, and when it does, it will delete the `Resource` it is managing.

Unlike `std::auto_ptr`, `std::unique_ptr` properly implements move semantics.

```

1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3  #include <utility> // for std::move
4
5  class Resource
6  {
7  public:
8      Resource() { std::cout << "Resource acquired\n"; }
9      ~Resource() { std::cout << "Resource destroyed\n"; }
10 };
11
12 int main()
13 {
14     std::unique_ptr<Resource> res1{ new Resource{} }; // Resource created here
15     std::unique_ptr<Resource> res2{}; // Start as nullptr
16
17     std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
18     std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");
19
20     // res2 = res1; // Won't compile: copy assignment is disabled
21     res2 = std::move(res1); // res2 assumes ownership, res1 is set to null
22
23     std::cout << "Ownership transferred\n";
24
25     std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
26     std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");
27
28     return 0;
29 } // Resource destroyed here when res2 goes out of scope

```

This prints:

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

Because `std::unique_ptr` is designed with move semantics in mind, copy initialization and copy assignment are disabled. If you want to transfer the contents managed by `std::unique_ptr`, you must use move semantics. In the program above, we accomplish this via `std::move` (which converts `res1` into an r-value, which triggers a move assignment instead of a copy assignment).

## Accessing the managed object

`std::unique_ptr` has an overloaded operator\* and operator-> that can be used to return the resource being managed. Operator\* returns a reference to the managed resource, and operator-> returns a pointer.

Remember that `std::unique_ptr` may not always be managing an object -- either because it was created empty (using the default constructor or passing in a nullptr as the parameter), or because the resource it was managing got moved to another `std::unique_ptr`. So before we use either of these operators, we should check whether the `std::unique_ptr` actually has a resource. Fortunately, this is easy: `std::unique_ptr` has a cast to bool that returns true if the `std::unique_ptr` is managing a resource.

Here's an example of this:

```
1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3
4  class Resource
5  {
6  public:
7      Resource() { std::cout << "Resource acquired\n"; }
8      ~Resource() { std::cout << "Resource destroyed\n"; }
9  };
10
11 std::ostream& operator<<(std::ostream& out, const Resource&)
12 {
13     out << "I am a resource";
14     return out;
15 }
16
17 int main()
18 {
19     std::unique_ptr<Resource> res{ new Resource{} };
20
21     if (res) // use implicit cast to bool to ensure res contains a Resource
22         std::cout << *res << '\n'; // print the Resource that res is owning
23
24     return 0;
25 }
```

This prints:

```
Resource acquired
I am a resource
Resource destroyed
```

In the above program, we use the overloaded operator\* to get the Resource object owned by std::unique\_ptr res, which we then send to std::cout for printing.

---

## std::unique\_ptr and arrays

Unlike std::auto\_ptr, std::unique\_ptr is smart enough to know whether to use scalar delete or array delete, so std::unique\_ptr is okay to use with both scalar objects and arrays.

However, std::array or std::vector (or std::string) are almost always better choices than using std::unique\_ptr with a fixed array, dynamic array, or C-style string.

### Best practice

Favor std::array, std::vector, or std::string over a smart pointer managing a fixed array, dynamic array, or C-style string.

---

## std::make\_unique

C++14 comes with an additional function named std::make\_unique(). This templated function constructs an object of the template type and initializes it with the arguments passed into the function.

```

1  #include <memory> // for std::unique_ptr and std::make_unique
2  #include <iostream>
3
4  class Fraction
5  {
6  private:
7      int m_numerator{ 0 };
8      int m_denominator{ 1 };
9
10 public:
11     Fraction(int numerator = 0, int denominator = 1) :
12         m_numerator{ numerator }, m_denominator{ denominator }
13     {
14     }
15
16     friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
17     {
18         out << f1.m_numerator << '/' << f1.m_denominator;
19         return out;
20     }
21 };
22
23
24 int main()
25 {
26     // Create a single dynamically allocated Fraction with numerator 3 and denominator
27     5
28     // We can also use automatic type deduction to good effect here
29     auto f1{ std::make_unique<Fraction>(3, 5) };
30     std::cout << *f1 << '\n';
31
32     // Create a dynamically allocated array of Fractions of length 4
33     auto f2{ std::make_unique<Fraction[]>(4) };
34     std::cout << f2[0] << '\n';
35
36     return 0;
37 }

```

The code above prints:

```

3/5
0/1

```

Use of `std::make_unique()` is optional, but is recommended over creating `std::unique_ptr` yourself. This is because code using `std::make_unique` is simpler, and it also requires less typing (when used with automatic type deduction). Furthermore, in C++14 it resolves an exception safety issue that can result from C++ leaving the order of evaluation for function arguments unspecified.

## Best practice

Use `std::make_unique()` instead of creating `std::unique_ptr` and using `new` yourself.

[\(\)](#) **The exception safety issue in more detail** [\(#smartpointerexceptionsafety\)](#)<sup>2</sup>

For those wondering what the “exception safety issue” mentioned above is, here’s a description of the issue.

Consider an expression like this one:

```
1 | some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

The compiler is given a lot of flexibility in terms of how it handles this call. It could create a new `T`, then call `function_that_can_throw_exception()`, then create the `std::unique_ptr` that manages the dynamically allocated `T`. If `function_that_can_throw_exception()` throws an exception, then the `T` that was allocated will not be deallocated, because the smart pointer to do the deallocation hasn't been created yet. This leads to `T` being leaked.

`std::make_unique()` doesn't suffer from this problem because the creation of the object `T` and the creation of the `std::unique_ptr` happen inside the `std::make_unique()` function, where there's no ambiguity about order of execution.

This issue was fixed in C++17, as evaluation of function arguments can no longer be interleaved.

---

## Returning `std::unique_ptr` from a function

`std::unique_ptr` can be safely returned from a function by value:

```
1 | #include <memory> // for std::unique_ptr
2 |
3 | std::unique_ptr<Resource> createResource()
4 | {
5 |     return std::make_unique<Resource>();
6 | }
7 |
8 | int main()
9 | {
10 |     auto ptr{ createResource() };
11 |
12 |     // do whatever
13 |
14 |     return 0;
15 | }
```

In the above code, `createResource()` returns a `std::unique_ptr` by value. If this value is not assigned to anything, the temporary return value will go out of scope and the `Resource` will be cleaned up. If it is assigned (as shown in `main()`), in C++14 or earlier, move semantics will be employed to transfer the `Resource` from the return value to the object assigned to (in the above example, `ptr`), and in C++17 or newer, the return will be elided. This makes returning a resource by `std::unique_ptr` much safer than returning raw pointers!

In general, you should not return `std::unique_ptr` by pointer (ever) or reference (unless you have a specific compelling reason to).

---

## Passing `std::unique_ptr` to a function

If you want the function to take ownership of the contents of the pointer, pass the `std::unique_ptr` by value. Note that because copy semantics have been disabled, you'll need to use `std::move` to actually pass the variable in.

```

1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3  #include <utility> // for std::move
4
5  class Resource
6  {
7  public:
8      Resource() { std::cout << "Resource acquired\n"; }
9      ~Resource() { std::cout << "Resource destroyed\n"; }
10 };
11
12 std::ostream& operator<<(std::ostream& out, const Resource&)
13 {
14     out << "I am a resource";
15     return out;
16 }
17
18 // This function takes ownership of the Resource, which isn't what we want
19 void takeOwnership(std::unique_ptr<Resource> res)
20 {
21     if (res)
22         std::cout << *res << '\n';
23 } // the Resource is destroyed here
24
25 int main()
26 {
27     auto ptr{ std::make_unique<Resource>() };
28
29     // takeOwnership(ptr); // This doesn't work, need to use move semantics
30     takeOwnership(std::move(ptr)); // ok: use move semantics
31
32     std::cout << "Ending program\n";
33
34     return 0;
35 }

```

The above program prints:

```

Resource acquired
I am a resource
Resource destroyed
Ending program

```

Note that in this case, ownership of the Resource was transferred to `takeOwnership()`, so the Resource was destroyed at the end of `takeOwnership()` rather than the end of `main()`.

However, most of the time, you won't want the function to take ownership of the resource.

Although you can pass a `std::unique_ptr` by const reference (which will allow the function to use the object without assuming ownership), it's better to just pass the resource itself (by pointer or reference, depending on whether null is a valid argument). This allows the function to remain agnostic of how the caller is managing its resources.

To get a raw pointer from a `std::unique_ptr`, you can use the `get()` member function:

```

1  #include <memory> // for std::unique_ptr
2  #include <iostream>
3
4  class Resource
5  {
6  public:
7      Resource() { std::cout << "Resource acquired\n"; }
8      ~Resource() { std::cout << "Resource destroyed\n"; }
9  };
10
11 std::ostream& operator<<(std::ostream& out, const Resource&)
12 {
13     out << "I am a resource";
14     return out;
15 }
16
17 // The function only uses the resource, so we'll accept a pointer to the resource, not
18 // a reference to the whole std::unique_ptr<Resource>
19 void useResource(const Resource* res)
20 {
21     if (res)
22         std::cout << *res << '\n';
23     else
24         std::cout << "No resource\n";
25 }
26
27 int main()
28 {
29     auto ptr{ std::make_unique<Resource>() };
30
31     useResource(ptr.get()); // note: get() used here to get a pointer to the Resource
32
33     std::cout << "Ending program\n";
34
35     return 0;
36 } // The Resource is destroyed here

```

The above program prints:

```

Resource acquired
I am a resource
Ending program
Resource destroyed

```

## std::unique\_ptr and classes

You can, of course, use `std::unique_ptr` as a composition member of your class. This way, you don't have to worry about ensuring your class destructor deletes the dynamic memory, as the `std::unique_ptr` will be automatically destroyed when the class object is destroyed.

However, if the class object is not destroyed properly (e.g. it is dynamically allocated and not deallocated properly), then the `std::unique_ptr` member will not be destroyed either, and the object being managed by the `std::unique_ptr` will not be deallocated.

## Misusing std::unique\_ptr

There are two easy ways to misuse `std::unique_ptr`s, both of which are easily avoided. First, don't let multiple objects manage the same resource. For example:



```
1 Resource* res{ new Resource() };
2 std::unique_ptr<Resource> res1{ res };
3 std::unique_ptr<Resource> res2{ res };
```

While this is legal syntactically, the end result will be that both `res1` and `res2` will try to delete the `Resource`, which will lead to undefined behavior.

Second, don't manually delete the resource out from underneath the `std::unique_ptr`.

```
1 Resource* res{ new Resource() };
2 std::unique_ptr<Resource> res1{ res };
3 delete res;
```

If you do, the `std::unique_ptr` will try to delete an already deleted resource, again leading to undefined behavior.

Note that `std::make_unique()` prevents both of the above cases from happening inadvertently.

---

## Quiz time

### Question #1

Convert the following program from using a normal pointer to using `std::unique_ptr` where appropriate:

```

1  #include <iostream>
2
3  class Fraction
4  {
5  private:
6      int m_numerator{ 0 };
7      int m_denominator{ 1 };
8
9  public:
10     Fraction(int numerator = 0, int denominator = 1) :
11         m_numerator{ numerator }, m_denominator{ denominator }
12     {
13     }
14
15     friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
16     {
17         out << f1.m_numerator << '/' << f1.m_denominator;
18         return out;
19     }
20 };
21
22 void printFraction(const Fraction* ptr)
23 {
24     if (ptr)
25         std::cout << *ptr << '\n';
26     else
27         std::cout << "No fraction\n";
28 }
29
30 int main()
31 {
32     auto* ptr{ new Fraction{ 3, 5 } };
33
34     printFraction(ptr);
35
36     delete ptr;
37
38     return 0;
39 }

```

[Show Solution](#)(javascript:void(0))<sup>3</sup>



**Next lesson**

**22.6** [std::shared\\_ptr](#)

4



**Back to table of contents**

5



**Previous lesson**

**22.4** [std::move](#)

6

7

**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/><sup>9</sup> are connected to your provided email address.**250 COMMENTS**

Newest ▼

**Jamison**

June 19, 2025 8:49 pm PDT

After reading these chapters, I wonder how anyone could even stand working on C++ before move semantics. You have all these great language features but have to use 'unsafe' C equivalents to avoid expensive copies EVERYWHERE. What a fantastic feature. It really makes C++ a great language.



0



Reply

**Sanderson**

May 1, 2025 2:51 am PDT

Alex, I hope you could add additional section on the customized deleter using a functor for this lesson, and when and how to use this technique:

```

1 struct XPacket
2 {
3     int size { 0 };
4     unsigned char* data { nullptr };
5 };
6
7 class PacketDeleter
8 {
9 public:
10     void Close() {
11         std::cout << "call PacketDeleter::Close()" << '\n';
12     }
13     void operator()(XPacket* x)
14     {
15         std::cout << "call PacketDeleter()" << '\n';
16         delete x->data;
17         delete x;
18     }
19 };
20
21 int main()
22 {
23     std::unique_ptr<XPacket, PacketDeleter> xPacketDeleter(new XPacket);
24     return 0;
25 }

```

 Last edited 2 months ago by Sanderson

 0  Reply



**NordicCat**

 January 15, 2025 9:12 pm PST

```

1 std::unique_ptr<vector<string>> res1 = std::make_unique<vector<string>>
  (initializer_list<string> {"hello", "puppy", "foo", "barr", "buzz", "looo"});
2
3 /*for (int i = 0; i < 10; i++) {
4     (*res1).at(i) = "Fear does not announce its arrival; it creeps in like mist.";
5 }*/
6
7 for (int i = 0; i < 3; i++) {
8     cout << (*res1).at(i) << endl;
9 }

```

So, this means we could initialize the vector of strings like this but not like in general??

 0  Reply



**Zendorr**

 January 6, 2025 11:04 pm PST

Got really confused about `std::unique_ptr::get`. So, just for clarification, because `unique_ptr` is a class, it is actually storing some resource while it is in scope, but with `get`, we just use raw pointer to get the address to the resource, which doesn't own it and thus doesn't need to manage it?

👍 1    ➡ Reply



**Alex**    Author

👤 Reply to [Zendorr](#)<sup>10</sup>    🕒 January 19, 2025 10:26 pm PST

Yeah, `std::unique_ptr` is really just a class that manages a raw pointer holding the address of a dynamically allocated resource. The destructor of `std::unique_ptr` ensures the pointer gets deleted when the `unique_ptr` goes out of scope. Calling `get()` just returns that pointer so the caller can use it in contexts that need a raw pointer (such as our example `void useResource(const Resource* res)` function). This raw pointer doesn't own the resource, but it can access it. When the raw pointer goes out of scope, nothing happens (which is fine, since the `std::unique_ptr` is still the owner).

👍 1    ➡ Reply



**EmtyC**

🕒 December 21, 2024 7:59 am PST

It seems the C++ lang attempts to implement automatic memory management while leaving manual possibilities, cool !!!

👍 3    ➡ Reply



**Phargelm**

🕒 August 18, 2024 4:26 pm PDT

> operator-> returns a pointer

As we've seen before with raw pointers, operator -> is used to access the members of an object being pointed to. But I can't figure out where it actually "returns a pointer"?

👍 0    ➡ Reply



**Alex**    Author

👤 Reply to [Phargelm](#)<sup>11</sup>    🕒 August 20, 2024 10:41 am PDT

See [https://en.cppreference.com/w/cpp/memory/unique\\_ptr/operator\\*](https://en.cppreference.com/w/cpp/memory/unique_ptr/operator*)

The `Auto_ptr3` class in lesson <https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/> has an overloaded `operator->` that returns a pointer.

👍 0    ➡ Reply



**Phargelm**

👤 Reply to [Alex](#)<sup>12</sup>    🕒 August 20, 2024 12:49 pm PDT

I've reviewed the `Auto_ptr3` implementation. I see that calling `operator->` works as expected but I can't get why it works. If `operator->` returns a pointer and it's unary operator then this expression should work: `auto* ptr{ res-> }` but it doesn't. We can't use the returned pointer explicitly like we do with other unary operators, but it's used implicitly only to access a member of the object being pointed. And this implicit logic is out of our control, our overloading function

actually doesn't do this member lookup. For other overloading operators we completely define what will be a result, but operator-> it's not the case.

Also I've found that "if the return value is another object of class type, not a pointer, then the subsequent member lookup is also handled by an `operator->` function. The language chains together the `operator->` calls until the last one returns a pointer."

<https://stackoverflow.com/questions/8777845/overloading-member-access-operators>

Maybe it worth to add to this lesson:

<https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/>

 Last edited 10 months ago by Phargelm

 1

 Reply



Alex

Author

 Reply to Phargelm<sup>13</sup>  August 21, 2024 10:46 am PDT

Yeah, `operator->` is a bit of a weird one, as it has the implicit recursion until it reaches a raw pointer. You can call an overloaded version of it directly via its function-like syntax:

```
res.operator->().
```

In normal use, `res->foo()` implicitly becomes `(*res.operator->()).foo()`.

I'm in the process of rewriting the chapter on operator overloading. I'm intending to add material related to overloading this operator then.

 3

 Reply



Asicx

 Reply to Alex<sup>14</sup>  September 7, 2024 2:00 pm PDT

I made a similar request in another lesson. This cleared the confusion for me:

"An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->` exists and if the operator is selected at the best match function by the overload resolution mechanism"

 0

 Reply



Phargelm

 August 18, 2024 9:00 am PDT

I just can't figure out this explanation:

> Although you can pass a `std::unique_ptr` by reference (which will allow the function to use the object without assuming ownership), you should only do so when the called function might alter or change the object being managed.

1. But if the called function might alter or change the object being managed then we can pass it as non-const pointer to non-const?

2. In contrast if the called function doesn't alter the object being managed, why can't we pass a `unique_ptr` object by constant reference instead of raw pointer? `void useResource(const std::unique_ptr<Resource>& res)`

 Last edited 10 months ago by Phargelm

 0  Reply



**Alex** Author

 Reply to [Phargelm](#)<sup>15</sup>  August 20, 2024 9:18 am PDT

1. "the object being managed" means the object owned by the `std::unique_ptr`. You can't do that without passing in the `std::unique_ptr` itself.
2. You can, but it's better to just pass in the resource itself, so the function doesn't have to match the kind of container the caller is using to manage the resource.

I tweaked some of the wording in this section.

 0  Reply



**Phargelm**

 Reply to [Phargelm](#)<sup>15</sup>  August 18, 2024 4:11 pm PDT

2. It seems I've got it. Passing `unique_ptr` as const reference still allows a function being called to modify underlying Resource as it's not treated as const.

 0  Reply



**Sai**

 August 15, 2024 10:15 am PDT

Hey, Alex,

Under `std::make_unique` section dont we have to use "new" operator to create Fraction and Fraction array dynamically? or it not required.

 0  Reply



**Alex** Author

 Reply to [Sai](#)<sup>16</sup>  August 17, 2024 9:53 pm PDT

`std::make_unique` calls `new` (to ensure it is the only owner of the object), so we don't need to do so ourselves.

 3  Reply



**Sai**

 Reply to [Alex](#)<sup>17</sup>  August 19, 2024 9:43 am PDT

Awesome!!! Thanks for this amazing work you do... God bless you man!

👍 0

➡ Reply



Kirill

🕒 July 30, 2024 2:49 pm PDT

Am I understanding this right that main reason why we don't want to pass `unique_ptr` by const reference is because `operator*` of `unique_ptr` returns NON-CONST reference to resource (even though overloaded `operator*` itself is marked as const)? So even if we pass `unique_ptr` by const reference we can still change it's resource:

```
1  #include <memory>
2  #include <iostream>
3
4  class Resource
5  {
6      int m_x{};
7  public:
8      Resource(int x)
9          : m_x{ x }
10     {}
11     ~Resource() { std::cout << "Resource destroyed\n"; }
12
13     int getX() const { return m_x; }
14     void setX(int x) { m_x = x; }
15 };
16
17 std::ostream& operator<<(std::ostream& out, const Resource& r)
18 {
19     out << "I am a resource, my value is: " << r.getX();
20     return out;
21 }
22
23 void useResource(const std::unique_ptr<Resource>& res) // passing by const reference
24 {
25     (*res).setX(6); // Resource's value changed since operator* returns non-const
26     reference
27     if (res)
28         std::cout << *res << '\n';
29     else
30         std::cout << "No resource\n";
31 }
32
33 int main()
34 {
35     auto r{ std::make_unique<Resource>(3) };
36     useResource(r);
37
38     return 0;
39 }
```

Considering this, why `unique_ptr` doesn't have another overloaded version of `operator*` which would actually marked as const and returned const reference to resource? Something like this:



```

1  #include <iostream>
2  class Resource
3  {
4      int m_x{};
5  public:
6      explicit Resource(int x)
7          : m_x{ x }
8      {}
9
10     int getX() const { return m_x; }
11     void setX(int x) { m_x = x; }
12 };
13
14 template <typename T>
15 class Manager
16 {
17
18     T m_val{};
19
20 public:
21     explicit Manager(T val)
22         : m_val{ val }
23     {}
24
25     const T& operator*() const
26     {
27         return m_val;
28     }
29
30     T& operator*()
31     {
32         return m_val;
33     }
34 };
35
36 void someConstFunc(const Manager<Resource>& c) // pass by const reference
37 {
38     (*c).setX(5); // compile error since operator* returns const reference
39     (*c).getX(); // but we can still use const functions of Resource
40 }
41
42 void someNonConstFunc(Manager<Resource>& c) // pass by non-const reference
43 {
44     (*c).setX(5); // works fine since non-const version of operator* is used
45 }
46
47 int main()
48 {
49     Manager<Resource> m{ Resource{5} };
50     someNonConstFunc(m);
51     someConstFunc(m);
52
53     return 0;
54 }

```

✍ Last edited 11 months ago by Kirill

👍 0    ➡ Reply



**Alex** Author

↩ Reply to Kirill<sup>18</sup> ⌚ July 30, 2024 7:06 pm PDT

My understanding is that a `const std::unique_ptr<T>` is a const pointer to T, not a pointer to const T. A `const std::unique_ptr<T>` can't transfer ownership (because it is const), but you can still modify the object being pointed to because the object itself is non-const.

As to your latter question, [https://stackoverflow.com/questions/44053034/why-does-stdunique\\_ptr-not-have-a-const-get-method](https://stackoverflow.com/questions/44053034/why-does-stdunique_ptr-not-have-a-const-get-method) asks the same thing.

👍 2

➡ Reply



**Strain**

🕒 May 25, 2024 1:05 am PDT

“As a reminder, a smart pointer is a class that manages a dynamically allocated object. Although smart pointers can offer other features, the defining characteristic of a smart pointer is that it manages a dynamically allocated resource, and ensures the dynamically allocated object is properly cleaned up at the appropriate time (usually when the smart pointer goes out of scope).”

Does that mean `std::vector` and `std::string` are smart pointers too? They do manage dynamically allocated data.

👍 0

➡ Reply



**Alex**

Author

↻ Reply to [Strain](#)<sup>19</sup> 🕒 May 28, 2024 9:36 am PDT

No. The difference is that smart pointers manage an object created by the user (either directly, or through a `make_xxx` function), whereas containers such as `std::vector` and `std::string` manage their own internal data.

I tweaked the sentence to read "Although smart pointers can offer other features, the defining characteristic of a smart pointer is that it manages a dynamically allocated resource **provided by the user** of the smart pointer".

👍 2

➡ Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#smartpointerexceptionsafety](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#smartpointerexceptionsafety)
3. `javascript:void(0)`
4. [https://www.learncpp.com/cpp-tutorial/stdshared\\_ptr/](https://www.learncpp.com/cpp-tutorial/stdshared_ptr/)
5. <https://www.learncpp.com/>
6. <https://www.learncpp.com/cpp-tutorial/stdmove/>
7. [https://www.learncpp.com/stdunique\\_ptr/](https://www.learncpp.com/stdunique_ptr/)
8. [https://www.learncpp.com/cpp-tutorial/stdinitializer\\_list/](https://www.learncpp.com/cpp-tutorial/stdinitializer_list/)

9. <https://gravatar.com/>
10. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-606352](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-606352)
11. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-601008](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-601008)
12. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-601070](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-601070)
13. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-601085](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-601085)
14. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-601126](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-601126)
15. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-600994](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-600994)
16. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-600911](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-600911)
17. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-600983](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-600983)
18. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-600286](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-600286)
19. [https://www.learncpp.com/cpp-tutorial/stdunique\\_ptr/#comment-597515](https://www.learncpp.com/cpp-tutorial/stdunique_ptr/#comment-597515)
20. <https://g.ezoic.net/privacy/learncpp.com>