

## 22.1 — Introduction to smart pointers and move semantics

👤 **ALEX<sup>1</sup>** ⌚ **FEBRUARY 4, 2025**

Consider a function in which we dynamically allocate a value:

```
1 void someFunction()
2 {
3     Resource* ptr = new Resource(); // Resource is a struct or class
4
5     // do stuff with ptr here
6
7     delete ptr;
8 }
```

Although the above code seems fairly straightforward, it's fairly easy to forget to deallocate `ptr`. Even if you do remember to delete `ptr` at the end of the function, there are a myriad of ways that `ptr` may not be deleted if the function exits early. This can happen via an early return:

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

or via a thrown exception:

```

1  #include <iostream>
2
3  void someFunction()
4  {
5      Resource* 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 }

```

In the above two programs, the early return or throw statement execute, causing the function to terminate without variable `ptr` being deleted. Consequently, the memory allocated for variable `ptr` is now leaked (and will be leaked again every time this function is called and returns early).

At heart, these kinds of issues occur because pointer variables have no inherent mechanism to clean up after themselves.

### Smart pointer classes to the rescue?

One of the best things about classes is that they contain destructors that automatically get executed when an object of the class goes out of scope. So if you allocate (or acquire) memory in your constructor, you can deallocate it in your destructor, and be guaranteed that the memory will be deallocated when the class object is destroyed (regardless of whether it goes out of scope, gets explicitly deleted, etc...). This is at the heart of the RAII programming paradigm that we talked about in lesson [19.3 -- Destructors](#) (<https://www.learncpp.com/cpp-tutorial/destructors/>)<sup>2</sup>.

So can we use a class to help us manage and clean up our pointers? We can!

Consider a class whose sole job was to hold and “own” a pointer passed to it, and then deallocate that pointer when the class object went out of scope. As long as objects of that class were only created as local variables, we could guarantee that the class would properly go out of scope (regardless of when or how our functions terminate) and the owned pointer would get destroyed.

Here’s a first draft of the idea:

```

1  #include <iostream>
2
3  template <typename T>
4  class Auto_ptr1
5  {
6      T* m_ptr {};
7  public:
8      // Pass in a pointer to "own" via the constructor
9      Auto_ptr1(T* ptr=nullptr)
10         :m_ptr(ptr)
11     {
12     }
13
14     // The destructor will make sure it gets deallocated
15     ~Auto_ptr1()
16     {
17         delete m_ptr;
18     }
19
20     // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
21     T& operator*() const { return *m_ptr; }
22     T* operator->() const { return m_ptr; }
23 };
24
25 // A sample class to prove the above works
26 class Resource
27 {
28 public:
29     Resource() { std::cout << "Resource acquired\n"; }
30     ~Resource() { std::cout << "Resource destroyed\n"; }
31 };
32
33 int main()
34 {
35     Auto_ptr1<Resource> res(new Resource()); // Note the allocation of memory here
36
37     // ... but no explicit delete needed
38
39     // Also note that we use <Resource>, not <Resource*>
40     // This is because we've defined m_ptr to have type T* (not T)
41
42     return 0;
43 } // res goes out of scope here, and destroys the allocated Resource for us

```

This program prints:

```

Resource acquired
Resource destroyed

```

Consider how this program and class work. First, we dynamically create a Resource, and pass it as a parameter to our templated Auto\_ptr1 class. From that point forward, our Auto\_ptr1 variable res owns that Resource object (Auto\_ptr1 has a composition relationship with m\_ptr). Because res is declared as a local variable and has block scope, it will go out of scope when the block ends, and be destroyed (no worries about forgetting to deallocate it). And because it is a class, when it is destroyed, the Auto\_ptr1 destructor will be called. That destructor will ensure that the Resource pointer it is holding gets deleted!

As long as Auto\_ptr1 is defined as a local variable (with automatic duration, hence the "Auto" part of the class name), the Resource will be guaranteed to be destroyed at the end of the block it is declared in, regardless of how the function terminates (even if it terminates early).

Such a class is called a smart pointer. A **Smart pointer** 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. (Relatedly, built-in pointers are sometimes called "dumb pointers" because they can't clean up after themselves).

Now let's go back to our someFunction() example above, and show how a smart pointer class can solve our challenge:

```
1  #include <iostream>
2
3  template <typename T>
4  class Auto_ptr1
5  {
6      T* m_ptr {};
7  public:
8      // Pass in a pointer to "own" via the constructor
9      Auto_ptr1(T* ptr=nullptr)
10         :m_ptr(ptr)
11     {
12     }
13
14     // The destructor will make sure it gets deallocated
15     ~Auto_ptr1()
16     {
17         delete m_ptr;
18     }
19
20     // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
21     T& operator*() const { return *m_ptr; }
22     T* operator->() const { return m_ptr; }
23 };
24
25 // A sample class to prove the above works
26 class Resource
27 {
28 public:
29     Resource() { std::cout << "Resource acquired\n"; }
30     ~Resource() { std::cout << "Resource destroyed\n"; }
31     void sayHi() { std::cout << "Hi!\n"; }
32 };
33
34 void someFunction()
35 {
36     Auto_ptr1<Resource> ptr(new Resource()); // ptr now owns the Resource
37
38     int x;
39     std::cout << "Enter an integer: ";
40     std::cin >> x;
41
42     if (x == 0)
43         return; // the function returns early
44
45     // do stuff with ptr here
46     ptr->sayHi();
47 }
48
49 int main()
50 {
51     someFunction();
52
53     return 0;
54 }
```

If the user enters a non-zero integer, the above program will print:

```
Resource acquired
Hi!
Resource destroyed
```

If the user enters zero, the above program will terminate early, printing:

```
Resource acquired
Resource destroyed
```

Note that even in the case where the user enters zero and the function terminates early, the Resource is still properly deallocated.

Because the ptr variable is a local variable, ptr will be destroyed when the function terminates (regardless of how it terminates). And because the Auto\_ptr1 destructor will clean up the Resource, we are assured that the Resource will be properly cleaned up.

### **A critical flaw**

The Auto\_ptr1 class has a critical flaw lurking behind some auto-generated code. Before reading further, see if you can identify what it is. We'll wait...

(Hint: consider what parts of a class get auto-generated if you don't supply them)

(Jeopardy music)

Okay, time's up.

Rather than tell you, we'll show you. Consider the following program:

```

1  #include <iostream>
2
3  // Same as above
4  template <typename T>
5  class Auto_ptr1
6  {
7      T* m_ptr {};
8  public:
9      Auto_ptr1(T* ptr=nullptr)
10         :m_ptr(ptr)
11     {
12     }
13
14     ~Auto_ptr1()
15     {
16         delete m_ptr;
17     }
18
19     T& operator*() const { return *m_ptr; }
20     T* operator->() const { return m_ptr; }
21 };
22
23 class Resource
24 {
25 public:
26     Resource() { std::cout << "Resource acquired\n"; }
27     ~Resource() { std::cout << "Resource destroyed\n"; }
28 };
29
30 int main()
31 {
32     Auto_ptr1<Resource> res1(new Resource());
33     Auto_ptr1<Resource> res2(res1); // Alternatively, don't initialize res2 and then
34     assign res2 = res1;
35
36     return 0;
37 } // res1 and res2 go out of scope here

```

This program prints:

```

Resource acquired
Resource destroyed
Resource destroyed

```

Very likely (but not necessarily) your program will crash at this point. See the problem now? Because we haven't supplied a copy constructor or an assignment operator, C++ provides one for us. And the functions it provides do shallow copies. So when we initialize res2 with res1, both Auto\_ptr1 variables are pointed at the same Resource. When res2 goes out of the scope, it deletes the resource, leaving res1 with a dangling pointer. When res1 goes to delete its (already deleted) Resource, undefined behavior will result (probably a crash)!

You'd run into a similar problem with a function like this:

```

1 void passByValue(Auto_ptr1<Resource> res)
2 {
3 }
4
5 int main()
6 {
7     Auto_ptr1<Resource> res1(new Resource());
8     passByValue(res1);
9
10    return 0;
11 }

```

In this program, `res1` will be copied by value into parameter `res`, so both `res1.m_ptr` and `res.m_ptr` will hold the same address.

When `res` is destroyed at the end of the function, `res1.m_ptr` is left dangling. When `res1.m_ptr` is later deleted, undefined behavior will result.

So clearly this isn't good. How can we address this?

Well, one thing we could do would be to explicitly define and delete the copy constructor and assignment operator, thereby preventing any copies from being made in the first place. That would prevent the pass by value case (which is good, we probably shouldn't be passing these by value anyway).

But then how would we return an `Auto_ptr1` from a function back to the caller?

```

1 ??? generateResource()
2 {
3     Resource* r{ new Resource() };
4     return Auto_ptr1(r);
5 }

```

We can't return our `Auto_ptr1` by reference, because the local `Auto_ptr1` will be destroyed at the end of the function, and the caller will be left with a dangling reference. We could return pointer `r` as `Resource*`, but then we might forget to delete `r` later, which is the whole point of using smart pointers in the first place. So that's out. Returning the `Auto_ptr1` by value is the only option that makes sense -- but then we end up with shallow copies, duplicated pointers, and crashes.

Another option would be to overload the copy constructor and assignment operator to make deep copies. In this way, we'd at least guarantee to avoid duplicate pointers to the same object. But copying can be expensive (and may not be desirable or even possible), and we don't want to make needless copies of objects just to return an `Auto_ptr1` from a function. Plus assigning or initializing a dumb pointer doesn't copy the object being pointed to, so why would we expect smart pointers to behave differently?

What do we do?

## Move semantics

What if, instead of having our copy constructor and assignment operator copy the pointer ("copy semantics"), we instead transfer/move ownership of the pointer from the source to the destination object? This is the core idea behind move semantics. **Move semantics** means the class will transfer ownership of the object rather than making a copy.

Let's update our `Auto_ptr1` class to show how this can be done:





```

1  #include <iostream>
2
3  template <typename T>
4  class Auto_ptr2
5  {
6      T* m_ptr {};
7  public:
8      Auto_ptr2(T* ptr=nullptr)
9          :m_ptr(ptr)
10     {
11     }
12
13     ~Auto_ptr2()
14     {
15         delete m_ptr;
16     }
17
18     // A copy constructor that implements move semantics
19     Auto_ptr2(Auto_ptr2& a) // note: not const
20     {
21         // We don't need to delete m_ptr here. This constructor is only called when
22         // we're creating a new object, and m_ptr can't be set prior to this.
23         m_ptr = a.m_ptr; // transfer our dumb pointer from the source to our local
24         // object
25         a.m_ptr = nullptr; // make sure the source no longer owns the pointer
26     }
27
28     // An assignment operator that implements move semantics
29     Auto_ptr2& operator=(Auto_ptr2& a) // note: not const
30     {
31         if (&a == this)
32             return *this;
33
34         delete m_ptr; // make sure we deallocate any pointer the destination is
35         // already holding first
36         m_ptr = a.m_ptr; // then transfer our dumb pointer from the source to the
37         // local object
38         a.m_ptr = nullptr; // make sure the source no longer owns the pointer
39         return *this;
40     }
41
42     T& operator*() const { return *m_ptr; }
43     T* operator->() const { return m_ptr; }
44     bool isNull() const { return m_ptr == nullptr; }
45 };
46
47 class Resource
48 {
49 public:
50     Resource() { std::cout << "Resource acquired\n"; }
51     ~Resource() { std::cout << "Resource destroyed\n"; }
52 };
53
54 int main()
55 {
56     Auto_ptr2<Resource> res1(new Resource());
57     Auto_ptr2<Resource> res2; // Start as nullptr
58
59     std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
60     std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");
61
62     res2 = res1; // res2 assumes ownership, res1 is set to null
63
64     std::cout << "Ownership transferred\n";
65
66     std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");
67     std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");
68
69     return 0;
70 }

```

This program prints:

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

Note that our overloaded operator= gave ownership of m\_ptr from res1 to res2! Consequently, we don't end up with duplicate copies of the pointer, and everything gets tidily cleaned up.

## A reminder

Deleting a nullptr is okay, as it does nothing.

### std::auto\_ptr, and why it was a bad idea

Now would be an appropriate time to talk about std::auto\_ptr. std::auto\_ptr, introduced in C++98 and removed in C++17, was C++'s first attempt at a standardized smart pointer. std::auto\_ptr opted to implement move semantics just like the Auto\_ptr2 class does.

However, std::auto\_ptr (and our Auto\_ptr2 class) has a number of problems that makes using it dangerous.

First, because std::auto\_ptr implements move semantics through the copy constructor and assignment operator, passing a std::auto\_ptr by value to a function will cause your resource to get moved to the function parameter (and be destroyed at the end of the function when the function parameters go out of scope). Then when you go to access your auto\_ptr argument from the caller (not realizing it was transferred and deleted), you're suddenly dereferencing a null pointer. Crash!

Second, std::auto\_ptr always deletes its contents using non-array delete. This means auto\_ptr won't work correctly with dynamically allocated arrays, because it uses the wrong kind of deallocation. Worse, it won't prevent you from passing it a dynamic array, which it will then mismanage, leading to memory leaks.

Finally, auto\_ptr doesn't play nice with a lot of the other classes in the standard library, including most of the containers and algorithms. This occurs because those standard library classes assume that when they copy an item, it actually makes a copy, not a move.

Because of the above mentioned shortcomings, std::auto\_ptr has been deprecated in C++11 and removed in C++17.


### Moving forward

The core problem with the design of std::auto\_ptr is that prior to C++11, the C++ language simply had no mechanism to differentiate "copy semantics" from "move semantics". Overriding the copy semantics to implement move semantics leads to weird edge cases and inadvertent bugs. For example, you can write `res1 = res2` and have no idea whether res2 will be changed or not!

Because of this, in C++11, the concept of "move" was formally defined, and "move semantics" were added to the language to properly differentiate copying from moving. Now that we've set the stage for why move

semantics can be useful, we'll explore the topic of move semantics throughout the rest of this chapter. We'll also fix our Auto\_ptr2 class using move semantics.

In C++11, std::auto\_ptr has been replaced by a bunch of other types of “move-aware” smart pointers: std::unique\_ptr, std::weak\_ptr, and std::shared\_ptr. We'll also explore the two most popular of these: unique\_ptr (which is a direct replacement for auto\_ptr) and shared\_ptr.



Next lesson


22.2 [R-value references](#)

3



Back to table of contents

4



Previous lesson

21.y [Chapter 21 project](#)

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/><sup>8</sup> are connected to your provided email address.

306 COMMENTS

Newest ▼



Swaminathan R

 May 15, 2025 3:47 pm PDT

So if you allocate (or acquire) memory in your constructor, you can deallocate it in y

Hi Alex,what's the difference between allocate and acquire here?

👍 0    ➡ Reply



**Danil**

🕒 February 2, 2025 8:05 am PST

I don't understand. Why is object res2 destroyed during copying `Auto_ptr1<Resource> res2(res1);`? from which scope does it come out?

📝 *Last edited 4 months ago by Danil*

👍 2    ➡ Reply



**Alex**

Author

↻ Reply to [Danil](#)<sup>9</sup>    🕒 February 4, 2025 8:44 pm PST

It's not. Both res1 and res2 are destroyed at the end of the function. And because they are both trying to manage the same Resource, you get a double-delete.

👍 0    ➡ Reply

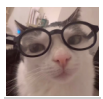


**Sergei**

🕒 January 15, 2025 6:54 am PST

A lot of problems could be solved if `delete/delete[]` operator assigned the NULL into the pointer variable... C++ creators solved the problem on 50% - starting from some version `delete NULL` pointer is a legal operation but deleting already freed pointer still make problems. Why they still go this way?

👍 0    ➡ Reply



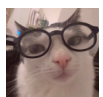
**NordicCat**

🕒 January 14, 2025 10:30 am PST

Does the whole point of `res1 = res2` is that "it does not specifies what would happen with res2"right? bc it sets the `res2 = nullptr` and transfer all resources via copying them. I am quite curious about: How should it supposed to perform (as expected by c++ devs) and How it is really behaving.

📝 *Last edited 5 months ago by NordicCat*

👍 0    ➡ Reply



**NordicCat**

🕒 January 14, 2025 8:40 am PST

btw does `&&` means this :

Here's a real-world analogy: Imagine you're moving to a different country permanently. You've got two friends:

Friend A (regular reference &) wants to borrow your stuff but keep it in good condition to return to you  
Friend B (rvalue reference &&) knows you're leaving forever, so they can take your stuff and do whatever they want with it - repaint it, modify it, even throw parts away

👍 0    ➡ Reply



**Avinash**

🕒 January 7, 2025 9:50 pm PST

Hello everyone,

I have a question regarding the arrow operator's overload process. The overloading makes it seem like it is a unary operator, while the usage makes it appear as a binary operator. Also the return value isn't also intuitive. Only a raw pointer is returned.

```

1  #include <iostream>
2
3  template <typename T>
4  class Auto_ptr1
5  {
6      T* m_ptr {};
7  public:
8      // Pass in a pointer to "own" via the constructor
9      Auto_ptr1(T* ptr=nullptr)
10         :m_ptr(ptr)
11     {
12     }
13
14     // The destructor will make sure it gets deallocated
15     ~Auto_ptr1()
16     {
17         delete m_ptr;
18     }
19
20     // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
21     T& operator*() const { return *m_ptr; }
22     T* operator->() const { return m_ptr; }
23 };
24
25 // A sample class to prove the above works
26 class Resource
27 {
28 public:
29     Resource() { std::cout << "Resource acquired\n"; }
30     ~Resource() { std::cout << "Resource destroyed\n"; }
31     void sayHi() { std::cout << "Hi!\n"; }
32 };
33
34 void someFunction()
35 {
36     Auto_ptr1<Resource> ptr(new Resource()); // ptr now owns the Resource
37
38     int x;
39     std::cout << "Enter an integer: ";
40     std::cin >> x;
41
42     if (x == 0)
43         return; // the function returns early
44
45     // do stuff with ptr here
46     ptr->sayHi();
47 }
48
49 int main()
50 {
51     someFunction();
52
53     return 0;
54 }

```

In the example `ptr->sayHi()`, "`ptr->`" should be replaced by the raw pointer, but apparently only "`ptr`" is replaced by the pointer, and anything that follows is coherent with intuition. The intermediate step is what bugs me. Is the overloading process of the arrow operator an exceptional case?

👍 0    ➡ Reply

**Alex**

Author

Reply to [Avinash](#)<sup>10</sup> January 21, 2025 11:09 am PST

1. The operator is overloaded as a member, and members have an implicit operand. So even though it looks unary, it's binary.
2. Yes, operator-> is a special case.



1



Reply

**Avinash**Reply to [Alex](#)<sup>11</sup> January 22, 2025 7:51 am PST

My query is settled. Thank you very much

Last edited 5 months ago by Avinash



0



Reply

**Eugene**

December 1, 2024 5:51 pm PST

Does the code you had for move semantics also share the same problem with auto\_ptr? If we create another function and pass in res2 by value, we would expect a segmentation fault as well since we leave the function scope and now res2 is a dangling pointer?



0



Reply

**Alex**

Author

Reply to [Eugene](#)<sup>12</sup> December 3, 2024 2:51 pm PST

Yes.



0



Reply

**frank**

October 24, 2024 5:09 pm PDT

```
Auto_ptr1<Resource> res1(new Resource())
```

what is `<Resource>` doing here? is it telling the template to specifically handle a Resource object or something?



0



Reply

**Alex**

Author

Reply to [frank](#)<sup>13</sup> October 25, 2024 5:17 pm PDT

Yes, it's being explicit about the fact that the type template argument for `Auto_ptr1` is `Resource`.

In C++17 it's not required, as we can use CTAD in this case.

👍 0    ➡ Reply



**Jonathan**

🕒 October 4, 2024 5:35 pm PDT

Hi, 2 questions on terminology re: "Move semantics means the class will transfer ownership of the object rather than making a copy."

1. Does this mean if I have a class that satisfies the Rule of 5 (Destructor, CC, CA, MC, MA), it does NOT have Move Semantics because it still allows copying?
2. As I also understand, the Rule of 3 is only about Copying (D, CC, CA) so such a class with Move Semantics (D, MC, MA; deleted CC, CA) does not fit into any of the Rule of 3/5/0. Is this correct?

(referring to [https://en.cppreference.com/w/cpp/language/rule\\_of\\_three](https://en.cppreference.com/w/cpp/language/rule_of_three))

✎ Last edited 8 months ago by Jonathan

👍 0    ➡ Reply



**Alex** Author

➡ Reply to Jonathan<sup>14</sup>    🕒 October 6, 2024 5:12 pm PDT

1. No. Whether a class does or does not allow copy semantics is independent from whether it supports move semantics.
2. The rule of 3 morphed into the rule of 5 to account for move semantics. A class with move semantics definitely fits into the rule of 5, which says that if you define or delete a move constructor and move assignment for your class, then you should define or delete the other three.

👍 0    ➡ Reply



**Jonathan**

➡ Reply to Alex<sup>15</sup>    🕒 October 6, 2024 6:01 pm PDT

Thanks for your response and clarification. I had a misunderstanding that the Rule of 3/5 was about only defining all 3/5. I did not realize it also included deleting. It's also good to know that copy and move semantics are independent.

I guess I took that quote out of context, where `Auto_ptr2` was meant to be a building-block to `Auto_ptr4/std::unique_ptr`. I still do find it slightly confusing as it implies move semantics "overrides" copy semantics; which it does from `Auto_ptr1` to `Auto_ptr2`, but not in the general C++ world.

👍 0    ➡ Reply



**Alex** Author

➡ Reply to Jonathan<sup>16</sup>    🕒 October 9, 2024 9:46 am PDT

Move semantics does take precedence over copy semantics where applicable. But since



most objects aren't movable, in most cases, a move just does a copy anyway...

👍 0

➡ Reply



**Mohamed**

🕒 September 17, 2024 6:20 pm PDT

I have a question just to challenge a bit the benefits of using smart pointers, why don't we implement a destroy method within resource class something like:

```
void Resource::destroy{delete this;} then call it from the main() as a normal method.
```

Also why we're not delegating the allocation of memory to the smart pointer and the client is the one responsible for that ?

Thanks.

👍 0

➡ Reply



**Alex**

Author

➡ Reply to [Mohamed](#) <sup>17</sup> 🕒 September 22, 2024 6:20 pm PDT

1. Because your call to `destroy()` will never be executed if the function exits early, or there is an exception, or any number of other things happen. Making deletion tied to the lifetime of a local object means we don't have to worry about how the function exits.
2. Not delegating the allocation of memory to the smart pointer gives the client the flexibility to either allocate a new object or to pass in an object that has previously been allocated. If you want to delegate allocation, write a separate function that does the allocation and returns the smart pointer. That way you can use that function if you want delegated allocation, and not if you don't.

👍 1

➡ Reply



**yagni**

➡ Reply to [Mohamed](#) <sup>17</sup> 🕒 September 19, 2024 3:20 am PDT

It seems to me that you can forget to delete the allocated memory, but it is unrealistic to forget to allocate memory. That's why it seems to me that they don't delegate it to the smart pointer.

👍 0

➡ Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/destructors/>

3. <https://www.learncpp.com/cpp-tutorial/rvalue-references/>
4. <https://www.learncpp.com/>
5. <https://www.learncpp.com/cpp-tutorial/chapter-21-project/>
6. <https://www.learncpp.com/introduction-to-smart-pointers-move-semantics/>
7. <https://www.learncpp.com/cpp-tutorial/chapter-27-summary-and-quiz/>
8. <https://gravatar.com/>
9. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-607288>
10. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-606369>
11. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-606818>
12. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-604752>
13. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-603504>
14. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-602716>
15. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-602775>
16. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-602784>
17. <https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/#comment-602068>
18. <https://g.ezoic.net/privacy/learncpp.com>