22.3 — Move constructors and move assignment

ALEX[1] FEBRUARY 18, 2025

In lesson 22.1 -- Introduction to smart pointers and move semantics (https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/)[2], we took a look at std::auto_ptr, discussed the desire for move semantics, and took a look at some of the downsides that occur when functions designed for copy semantics (copy constructors and copy assignment operators) are redefined to implement move semantics.

In this lesson, we'll take a deeper look at how C++11 resolves these problems via move constructors and move assignment.

Recapping copy constructors and copy assignment

First, let's take a moment to recap copy semantics.

Copy constructors are used to initialize a class by making a copy of an object of the same class. Copy assignment is used to copy one class object to another existing class object. By default, C++ will provide a copy constructor and copy assignment operator if one is not explicitly provided. These compiler-provided functions do shallow copies, which may cause problems for classes that allocate dynamic memory. So classes that deal with dynamic memory should override these functions to do deep copies.

Returning back to our Auto_ptr smart pointer class example from the first lesson in this chapter, let's look at a version that implements a copy constructor and copy assignment operator that do deep copies, and a sample program that exercises them:

```cpp
#include <iostream>

template<typename T>
class Auto_ptr3
{
    T* m_ptr {};
public:
    Auto_ptr3(T* ptr = nullptr)
        : m_ptr { ptr }
    {
    }

    ~Auto_ptr3()
    {
        delete m_ptr;
    }

    // Copy constructor
    // Do deep copy of a.m_ptr to m_ptr
    Auto_ptr3(const Auto_ptr3& a)
    {
        m_ptr = new T;
        *m_ptr = *a.m_ptr;
    }

    // Copy assignment
    // Do deep copy of a.m_ptr to m_ptr
    Auto_ptr3& operator=(const Auto_ptr3& a)
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Copy the resource
        m_ptr = new T;
        *m_ptr = *a.m_ptr;

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};

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

Auto_ptr3<Resource> generateResource()
{
    Auto_ptr3<Resource> res{new Resource};
    return res; // this return value will invoke the copy constructor
}

int main()
{
    Auto_ptr3<Resource> mainres;
    mainres = generateResource(); // this assignment will invoke the copy assignment

    return 0;
}
```

In this program, we're using a function named generateResource() to create a smart pointer encapsulated resource, which is then passed back to function main(). Function main() then assigns that to an existing Auto_ptr3 object.

When this program is run, it prints:

```
Resource acquired
Resource acquired
Resource destroyed
Resource acquired
Resource destroyed
Resource destroyed
```

(Note: You may only get 4 outputs if your compiler elides the return value from function generateResource())

That's a lot of resource creation and destruction going on for such a simple program! What's going on here?

Let's take a closer look. There are 6 key steps that happen in this program (one for each printed message):

1. Inside generateResource(), local variable res is created and initialized with a dynamically allocated Resource, which causes the first "Resource acquired".
2. Res is returned back to main() by value. We return by value here because res is a local variable -- it can't be returned by address or reference because res will be destroyed when generateResource() ends. So res is copy constructed into a temporary object. Since our copy constructor does a deep copy, a new Resource is allocated here, which causes the second "Resource acquired".
3. Res goes out of scope, destroying the originally created Resource, which causes the first "Resource destroyed".
4. The temporary object is assigned to mainres by copy assignment. Since our copy assignment also does a deep copy, a new Resource is allocated, causing yet another "Resource acquired".
5. The assignment expression ends, and the temporary object goes out of expression scope and is destroyed, causing a "Resource destroyed".
6. At the end of main(), mainres goes out of scope, and our final "Resource destroyed" is displayed.

So, in short, because we call the copy constructor once to copy construct res to a temporary, and copy assignment once to copy the temporary into mainres, we end up allocating and destroying 3 separate objects in total.

Inefficient, but at least it doesn't crash!

However, with move semantics, we can do better.

Move constructors and move assignment

C++11 defines two new functions in service of move semantics: a move constructor, and a move assignment operator. Whereas the goal of the copy constructor and copy assignment is to make a copy of one object to another, the goal of the move constructor and move assignment is to move ownership of the resources from one object to another (which is typically much less expensive than making a copy).

Defining a move constructor and move assignment work analogously to their copy counterparts. However, whereas the copy flavors of these functions take a const l-value reference parameter (which will bind to just about anything), the move flavors of these functions use non-const rvalue reference parameters (which only bind to rvalues).

Here's the same Auto_ptr3 class as above, with a move constructor and move assignment operator added. We've left in the deep-copying copy constructor and copy assignment operator for comparison purposes.

```cpp
#include <iostream>

template<typename T>
class Auto_ptr4
{
	T* m_ptr {};
public:
	Auto_ptr4(T* ptr = nullptr)
		: m_ptr { ptr }
	{
	}

	~Auto_ptr4()
	{
		delete m_ptr;
	}

	// Copy constructor
	// Do deep copy of a.m_ptr to m_ptr
	Auto_ptr4(const Auto_ptr4& a)
	{
		m_ptr = new T;
		*m_ptr = *a.m_ptr;
	}

	// Move constructor
	// Transfer ownership of a.m_ptr to m_ptr
	Auto_ptr4(Auto_ptr4&& a) noexcept
		: m_ptr { a.m_ptr }
	{
		a.m_ptr = nullptr; // we'll talk more about this line below
	}

	// Copy assignment
	// Do deep copy of a.m_ptr to m_ptr
	Auto_ptr4& operator=(const Auto_ptr4& a)
	{
		// Self-assignment detection
		if (&a == this)
			return *this;

		// Release any resource we're holding
		delete m_ptr;

		// Copy the resource
		m_ptr = new T;
		*m_ptr = *a.m_ptr;

		return *this;
	}

	// Move assignment
	// Transfer ownership of a.m_ptr to m_ptr
	Auto_ptr4& operator=(Auto_ptr4&& a) noexcept
	{
		// Self-assignment detection
		if (&a == this)
			return *this;

		// Release any resource we're holding
		delete m_ptr;

		// Transfer ownership of a.m_ptr to m_ptr
		m_ptr = a.m_ptr;
		a.m_ptr = nullptr; // we'll talk more about this line below

		return *this;
	}

	T& operator*() const { return *m_ptr; }
```

```
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};

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

Auto_ptr4<Resource> generateResource()
{
    Auto_ptr4<Resource> res{new Resource};
    return res; // this return value will invoke the move constructor
}

int main()
{
    Auto_ptr4<Resource> mainres;
    mainres = generateResource(); // this assignment will invoke the move assignment

    return 0;
}
```

The move constructor and move assignment operator are simple. Instead of deep copying the source object (`a`) into the destination object (the implicit object), we simply move (steal) the source object's resources. This involves shallow copying the source pointer into the implicit object, then setting the source pointer to null.

When run, this program prints:

```
Resource acquired
Resource destroyed
```

That's much better!

The flow of the program is exactly the same as before. However, instead of calling the copy constructor and copy assignment operators, this program calls the move constructor and move assignment operators. Looking a little more deeply:

1. Inside generateResource(), local variable res is created and initialized with a dynamically allocated Resource, which causes the first "Resource acquired".
2. Res is returned back to main() by value. Res is move constructed into a temporary object, transferring the dynamically created object stored in res to the temporary object. We'll talk about why this happens below.
3. Res goes out of scope. Because res no longer manages a pointer (it was moved to the temporary), nothing interesting happens here.
4. The temporary object is move assigned to mainres. This transfers the dynamically created object stored in the temporary to mainres.
5. The assignment expression ends, and the temporary object goes out of expression scope and is destroyed. However, because the temporary no longer manages a pointer (it was moved to mainres), nothing interesting happens here either.
6. At the end of main(), mainres goes out of scope, and our final "Resource destroyed" is displayed.

So instead of copying our Resource twice (once for the copy constructor and once for the copy assignment), we transfer it twice. This is more efficient, as Resource is only constructed and destroyed once instead of three times.

Related content

Move constructors and move assignment should be marked as `noexcept`. This tells the compiler that these functions will not throw exceptions.

We introduce `noexcept` in lesson [27.9 -- Exception specifications and noexcept](https://www.learncpp.com/cpp-tutorial/exception-specifications-and-noexcept/) (https://www.learncpp.com/cpp-tutorial/exception-specifications-and-noexcept/)[3] and discuss why move constructors and move assignment are marked as `noexcept` in lesson [27.10 -- std::move_if_noexcept](https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/) (https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/)[4].

When are the move constructor and move assignment called?

The move constructor and move assignment are called when those functions have been defined, and the argument for construction or assignment is an rvalue. Most typically, this rvalue will be a literal or temporary value.

The copy constructor and copy assignment are used otherwise (when the argument is an lvalue, or when the argument is an rvalue and the move constructor or move assignment functions aren't defined).

Implicit move constructor and move assignment operator

The compiler will create an implicit move constructor and move assignment operator if all of the following are true:

- There are no user-declared copy constructors or copy assignment operators.
- There are no user-declared move constructors or move assignment operators.
- There is no user-declared destructor.

These functions do a memberwise move, which behaves as follows:

- If member has a move constructor or move assignment (as appropriate), it will be invoked.
- Otherwise, the member will be copied.

Notably, this means that pointers will be copied, not moved!

Warning

The implicit move constructor and move assignment will copy pointers, not move them. If you want to move a pointer member, you will need to define the move constructor and move assignment yourself.

The key insight behind move semantics

You now have enough context to understand the key insight behind move semantics.

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" (where b is an lvalue), 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!

C++11, through r-value references, gives us the ability to provide different behaviors when the argument is an r-value vs an l-value, enabling us to make smarter and more efficient decisions about how our objects should behave.

> **Key insight**
>
> Move semantics is an optimization opportunity.

Move functions should always leave both objects in a valid state

In the above examples, both the move constructor and move assignment functions set a.m_ptr to nullptr. This may seem extraneous -- after all, if `a` is a temporary r-value, why bother doing “cleanup” if parameter `a` is going to be destroyed anyway?

The answer is simple: When `a` goes out of scope, the destructor for `a` will be called, and `a.m_ptr` will be deleted. If at that point, `a.m_ptr` is still pointing to the same object as `m_ptr`, then `m_ptr` will be left as a dangling pointer. When the object containing `m_ptr` eventually gets used (or destroyed), we’ll get undefined behavior.

When implementing move semantics, it is important to ensure the moved-from object is left in a valid state, so that it will destruct properly (without creating undefined behavior).

Automatic l-values returned by value may be moved instead of copied

In the generateResource() function of the Auto_ptr4 example above, when variable res is returned by value, it is moved instead of copied, even though res is an l-value. The C++ specification has a special rule that says automatic objects returned from a function by value can be moved even if they are l-values. This makes sense, since res was going to be destroyed at the end of the function anyway! We might as well steal its resources instead of making an expensive and unnecessary copy.

Although the compiler can move l-value return values, in some cases it may be able to do even better by simply eliding the copy altogether (which avoids the need to make a copy or do a move at all). In such a case, neither the copy constructor nor move constructor would be called.

Disabling copying

In the Auto_ptr4 class above, we left in the copy constructor and assignment operator for comparison purposes. But in move-enabled classes, it is sometimes desirable to delete the copy constructor and copy assignment functions to ensure copies aren’t made. In the case of our Auto_ptr class, we don’t want to copy our templated object T -- both because it’s expensive, and whatever class T is may not even support copying!

Here’s a version of Auto_ptr that supports move semantics but not copy semantics:

```
#include <iostream>

template<typename T>
class Auto_ptr5
{
	T* m_ptr {};
public:
	Auto_ptr5(T* ptr = nullptr)
		: m_ptr { ptr }
	{
	}

	~Auto_ptr5()
	{
		delete m_ptr;
	}

	// Copy constructor -- no copying allowed!
	Auto_ptr5(const Auto_ptr5& a) = delete;

	// Move constructor
	// Transfer ownership of a.m_ptr to m_ptr
	Auto_ptr5(Auto_ptr5&& a) noexcept
		: m_ptr { a.m_ptr }
	{
		a.m_ptr = nullptr;
	}

	// Copy assignment -- no copying allowed!
	Auto_ptr5& operator=(const Auto_ptr5& a) = delete;

	// Move assignment
	// Transfer ownership of a.m_ptr to m_ptr
	Auto_ptr5& operator=(Auto_ptr5&& a) noexcept
	{
		// Self-assignment detection
		if (&a == this)
			return *this;

		// Release any resource we're holding
		delete m_ptr;

		// Transfer ownership of a.m_ptr to m_ptr
		m_ptr = a.m_ptr;
		a.m_ptr = nullptr;

		return *this;
	}

	T& operator*() const { return *m_ptr; }
	T* operator->() const { return m_ptr; }
	bool isNull() const { return m_ptr == nullptr; }
};
```

If you were to try to pass an Auto_ptr5 l-value to a function by value, the compiler would complain that the copy constructor required to initialize the function parameter has been deleted. This is good, because we should probably be passing Auto_ptr5 by const l-value reference anyway!

Auto_ptr5 is (finally) a good smart pointer class. And, in fact the standard library contains a class very much like this one (that you should use instead), named std::unique_ptr. We'll talk more about std::unique_ptr later in this chapter.

Another example

Let's take a look at another class that uses dynamic memory: a simple dynamic templated array. This class contains a deep-copying copy constructor and copy assignment operator.

```
#include <cstddef> // for std::size_t

template <typename T>
class DynamicArray
{
private:
    T* m_array {};
    int m_length {};

    void alloc(int length)
    {
        m_array = new T[static_cast<std::size_t>(length)];
            m_length = length;
    }
public:
    DynamicArray(int length)
    {
        alloc(length);
    }

    ~DynamicArray()
    {
        delete[] m_array;
    }

    // Copy constructor
    DynamicArray(const DynamicArray &arr)
    {
        alloc(arr.m_length);
        std::copy_n(arr.m_array, m_length, m_array); // copy m_length elements from arr to m_array
    }

    // Copy assignment
    DynamicArray& operator=(const DynamicArray &arr)
    {
        if (&arr == this)
            return *this;

        delete[] m_array;

        alloc(arr.m_length);

        std::copy_n(arr.m_array, m_length, m_array); // copy m_length elements from arr to m_array

        return *this;
    }

    int getLength() const { return m_length; }
    T& operator[](int index) { return m_array[index]; }
    const T& operator[](int index) const { return m_array[index]; }
};
```

Now let's use this class in a program. To show you how this class performs when we allocate a million integers on the heap, we're going to leverage the Timer class we developed in lesson [18.4 -- Timing your code](https://www.learncpp.com/cpp-tutorial/timing-your-code/) (https://www.learncpp.com/cpp-tutorial/timing-your-code/)[5]. We'll use the Timer class to time how fast our code runs, and show you the performance difference between copying and moving.

```cpp
#include <algorithm> // for std::copy_n
#include <chrono> // for std::chrono functions
#include <iostream>

// Uses the above DynamicArray class

class Timer
{
private:
    // Type aliases to make accessing nested type easier
    using Clock = std::chrono::high_resolution_clock;
    using Second = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<Clock> m_beg { Clock::now() };

public:
    void reset()
    {
        m_beg = Clock::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
    }
};

// Return a copy of arr with all of the values doubled
DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
{
    DynamicArray<int> dbl(arr.getLength());
    for (int i = 0; i < arr.getLength(); ++i)
        dbl[i] = arr[i] * 2;

    return dbl;
}

int main()
{
    Timer t;

    DynamicArray<int> arr(1000000);

    for (int i = 0; i < arr.getLength(); i++)
        arr[i] = i;

    arr = cloneArrayAndDouble(arr);

    std::cout << t.elapsed();
}
```

On one of the author's machines, in release mode, this program executed in 0.00825559 seconds.

Now let's update `DynamicArray` by replacing the copy constructor and copy assignment with a move constructor and move assignment, and then run the program again:

```
#include <cstddef> // for std::size_t

template <typename T>
class DynamicArray
{
private:
    T* m_array {};
    int m_length {};

    void alloc(int length)
    {
        m_array = new T[static_cast<std::size_t>(length)];
        m_length = length;
    }
public:
    DynamicArray(int length)
    {
        alloc(length);
    }

    ~DynamicArray()
    {
        delete[] m_array;
    }

    // Copy constructor
    DynamicArray(const DynamicArray &arr) = delete;

    // Copy assignment
    DynamicArray& operator=(const DynamicArray &arr) = delete;

    // Move constructor
    DynamicArray(DynamicArray &&arr) noexcept
        :  m_array { arr.m_array }, m_length { arr.m_length }
    {
        arr.m_length = 0;
        arr.m_array = nullptr;
    }

    // Move assignment
    DynamicArray& operator=(DynamicArray &&arr) noexcept
    {
        if (&arr == this)
            return *this;

        delete[] m_array;

        m_length = arr.m_length;
        m_array = arr.m_array;
        arr.m_length = 0;
        arr.m_array = nullptr;

        return *this;
    }

    int getLength() const { return m_length; }
    T& operator[](int index) { return m_array[index]; }
    const T& operator[](int index) const { return m_array[index]; }
};

#include <iostream>
#include <chrono> // for std::chrono functions

class Timer
{
private:
    // Type aliases to make accessing nested type easier
    using Clock = std::chrono::high_resolution_clock;
    using Second = std::chrono::duration<double, std::ratio<1> >;

```

```
    std::chrono::time_point<Clock> m_beg { Clock::now() };

public:
    void reset()
    {
        m_beg = Clock::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
    }
};

// Return a copy of arr with all of the values doubled
DynamicArray<int> cloneArrayAndDouble(const DynamicArray<int> &arr)
{
    DynamicArray<int> dbl(arr.getLength());
    for (int i = 0; i < arr.getLength(); ++i)
        dbl[i] = arr[i] * 2;

    return dbl;
}

int main()
{
    Timer t;

    DynamicArray<int> arr(1000000);

    for (int i = 0; i < arr.getLength(); i++)
        arr[i] = i;

    arr = cloneArrayAndDouble(arr);

    std::cout << t.elapsed();
}
```

On the same machine, this program executed in 0.0056 seconds.

Comparing the runtime of the two programs, (0.00825559 - 0.0056) / 0.00825559 * 100 = 32.1% faster!

Deleting the move constructor and move assignment

You can delete the move constructor and move assignment using the `= delete` syntax in the exact same way you can delete the copy constructor and copy assignment.

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Name
{
private:
    std::string m_name {};

public:
    Name(std::string_view name) : m_name{ name }
    {
    }

    Name(const Name& name) = delete;
    Name& operator=(const Name& name) = delete;
    Name(Name&& name) = delete;
    Name& operator=(Name&& name) = delete;

    const std::string& get() const { return m_name; }
};

int main()
{
    Name n1{ "Alex" };
    n1 = Name{ "Joe" }; // error: move assignment deleted

    std::cout << n1.get() << '\n';

    return 0;
}
```

If you delete the copy constructor, the compiler will not generate an implicit move constructor (making your objects neither copyable nor movable). Therefore, when deleting the copy constructor, it is useful to be explicit about what behavior you want from your move constructors. Either explicitly delete them (making it clear this is the desired behavior), or default them (making the class move-only).

Key insight

The **rule of five** says that if the copy constructor, copy assignment, move constructor, move assignment, or destructor are defined or deleted, then each of those functions should be defined or deleted.

While deleting only the move constructor and move assignment may seem like a good idea if you want a copyable but not movable object, this has the unfortunate consequence of making the class not returnable by value in cases where mandatory copy elision does not apply. This happens because a deleted move constructor is still a declared function, and thus is eligible for overload resolution. And return by value will favor a deleted move constructor over a non-deleted copy constructor. This is illustrated by the following program:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Name
{
private:
    std::string m_name {};

public:
    Name(std::string_view name) : m_name{ name }
    {
    }

    Name(const Name& name) = default;
    Name& operator=(const Name& name) = default;

    Name(Name&& name) = delete;
    Name& operator=(Name&& name) = delete;

    const std::string& get() const { return m_name; }
};

Name getJoe()
{
    Name joe{ "Joe" };
    return joe; // error: Move constructor was deleted
}

int main()
{
    Name n{ getJoe() };

    std::cout << n.get() << '\n';

    return 0;
}
```

Issues with move semantics and `std::swap` Advanced

In lesson [21.12 -- Overloading the assignment operator (https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/)][6], we mentioned the copy and swap idiom. Copy and swap also works for move semantics, meaning we can implement our move constructor and move assignment by swapping resources with the object that will be destroyed.

This has two benefits:

- The persistent object now controls the resources that were previously under ownership of the dying object (which was our primary goal).
- The dying object now controls the resources that were previously under ownership of the persistent object. When the dying object actually dies, it can do any kind of cleanup required on those resources.

When you think about swapping, the first thing that comes to mind is usually `std::swap()`. However, implementing the move constructor and move assignment using `std::swap()` is problematic, as `std::swap()` calls both the move constructor and move assignment on move-capable objects. This will result in an infinite recursion issue.

You can see this happen in the following example:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Name
{
private:
    std::string m_name {}; // std::string is move capable

public:
    Name(std::string_view name) : m_name{ name }
    {
    }

    Name(const Name& name) = delete;
    Name& operator=(const Name& name) = delete;

    Name(Name&& name) noexcept
    {
        std::cout << "Move ctor\n";

        std::swap(*this, name); // bad!
    }

    Name& operator=(Name&& name) noexcept
    {
        std::cout << "Move assign\n";

        std::swap(*this, name); // bad!

        return *this;
    }

    const std::string& get() const { return m_name; }
};

int main()
{
    Name n1{ "Alex" };
    n1 = Name{"Joe"}; // invokes move assignment

    std::cout << n1.get() << '\n';

    return 0;
}
```

This prints:

```
Move assign
Move ctor
Move ctor
Move ctor
Move ctor
```

And so on… until the stack overflows.

You can implement the move constructor and move assignment using your own swap function, as long as your swap member function does not call the move constructor or move assignment. Here’s an example of how that can be done:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Name
{
private:
    std::string m_name {};

public:
    Name(std::string_view name) : m_name{ name }
    {
    }

    Name(const Name& name) = delete;
    Name& operator=(const Name& name) = delete;

    // Create our own swap friend function to swap the members of Name
    friend void swap(Name& a, Name& b) noexcept
    {
        // We avoid recursive calls by invoking std::swap on the std::string member,
        // not on Name
        std::swap(a.m_name, b.m_name);
    }

    Name(Name&& name) noexcept
    {
        std::cout << "Move ctor\n";

        swap(*this, name); // Now calling our swap, not std::swap
    }

    Name& operator=(Name&& name) noexcept
    {
        std::cout << "Move assign\n";

        swap(*this, name); // Now calling our swap, not std::swap

        return *this;
    }

    const std::string& get() const { return m_name; }
};

int main()
{
    Name n1{ "Alex" };
    n1 = Name{"Joe"}; // invokes move assignment

    std::cout << n1.get() << '\n';

    return 0;
}
```

This works as expected, and prints:

```
Move assign
Joe
```

[Next lesson](#)
22.4 std::move

7

[Back to table of contents](#)

8

[Previous lesson](#)
22.2 R-value references

9

10

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/[11] are connected to your provided email address.

417 COMMENTS Newest

X_Taste
May 25, 2025 12:23 am PDT

Help! My Brain is not working.

0 Reply

Kania
April 19, 2025 11:10 pm PDT

My brain isn't braining

1 Reply

eren

Reply to Kania [12] April 27, 2025 12:24 am PDT

hold on, this is just the begining :D

0 Reply

Kania

Reply to eren [13] May 6, 2025 7:54 pm PDT

:D

0 Reply

eren

Reply to Kania [14] May 8, 2025 10:02 pm PDT

I found that if you are confusing with some concepts, it's a good method to use some LLM tools, you can ask them to explain some things you are not sure, try it, then you can build a more profound understanding.

1 Reply

sandersan

April 13, 2025 7:03 am PDT

I have tried many times the DynamicArray class in CLion on macOS, but there is no significant difference in time consumption between these two variants. Both of these variants consume ±0.06 seconds.

0 Reply

Bhargavi

March 21, 2025 7:26 pm PDT

I think found the answer to my question under section "Automatic l-values returned by value may be moved instead of copied".

0 Reply

Bhargavi

March 21, 2025 7:11 pm PDT

Hi Alex,

Auto_ptr3<Resource> generateResource()
{
Auto_ptr3<Resource> res{new Resource};

return res; // this return value will invoke the copy constructor
}

Here when move constructor is invoked with res, res is not a temporary object or rvalue right? then how it is passed to rvalue reference in move constructor?

May be I'm missing something here.. please help me out to understand this.

0 Reply

ArcShahi

Reply to Bhargavi [15] March 24, 2025 7:59 am PDT

Automatic l-values returned by value may be moved instead of copied

In the `generateResource()` function of the Auto_ptr4 example above, when variable res is returned by value, it is moved instead of copied, even though res is an l-value. **The C++ specification has a special rule that says automatic objects returned from a function by value can be moved even if they are l-values**.

I think you missed this section.

Last edited 3 months ago by ArcShahi

2 Reply

JiaChen

March 9, 2025 8:36 am PDT

Is it correct that writing like this calls the move constructor?

```
Auto_ptr4<Resource> mainres {generateResource()};
```

Last edited 3 months ago by JiaChen

0 Reply

jia

Reply to JiaChen [16] March 24, 2025 1:18 am PDT

yes

0 Reply

jyc

March 3, 2025 6:52 am PST

Hey Alex,

Just wanted to make sure I understand:

1. In 14.15[17] you mentioned that the "copy constructor is normally called when an argument of the same type as the parameter is passed by value or return by value is used." In this chapter, you mentioned "automatic l-values returned by value may be moved instead of copied".

Does this mean when it comes to returning by value, move constructors have a higher precedence than copy constructors (assuming both are eligible)?

2. If I want to make sure that an object is only copyable and never movable, I should just declare the copy constructor? That way the compiler will not create an implicit move constructor/assignment. And as a result move semantics will not be invoked.

Thank you for your time!

0 Reply

jyc

Reply to jyc[18] March 19, 2025 7:26 am PDT

Attempting to answer my own questions after going through the lessons again:

1. The precedence and order are stated in the section "When are the move constructor and move assignment called?".
2. Yes, without move constructor/assignments, no move semantics will be invoked.
3. For heap-allocated, move will be safe. Stack-allocated should be copied.

0 Reply

jyc

Reply to jyc[18] March 6, 2025 5:25 am PST

More questions:

3. Regarding "automatic l-values returned by value may be moved instead of copied", does this depend on whether the value to be returned is allocated in stack vs heap? A copy should be made if the source is in stack right? Whereas for dynamically allocated memory, a move will be safe?

Again thanks for your time and help.

0 Reply

Alex(student)

March 1, 2025 12:38 pm PST

I tried DynamicArray for some calculations, and without -O3 it was about 20% faster than std::vector, but with -O3 it was 25% slower... strange ?

the arrays were 1000x4000 = 4M elements, and hitting about 2000 page faults a second with the very unoptimized calculation method

Last edited 4 months ago by Alex(student)

0 Reply

Sai
February 17, 2025 7:42 am PST

Is there a reason why move constructors use direct initialization over direct list initialization that copy and regular constructors use?

0 Reply

Alex Author
Reply to Sai [19] February 18, 2025 8:17 pm PST

Nope, just an oversight. I've corrected them to use direct list initialization.

0 Reply

Kaus05
February 7, 2025 4:45 am PST

Hi Alex
what would happen if i were to do

```
Name n1{"Alex"};
Name n2 {"Kaus"}:
n2=std::move(n1);
```

is it okay if n1 is not pointing to nullptr but actual data?
also after going through this https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom
i have no idea how does copy and swap work for copy assignment?
do we change the swap function to take in by value for second argument?

0 Reply

Alex Author
Reply to Kaus05 [20] February 10, 2025 2:21 pm PST

1. You'll get a compile error because this would use deleted function 'Name& Name::operator=(Name&&)'.
2. The data to be assigned is copied into the parameter, which is then swapped with the current object. At this point, the current object contains the data we wanted to assign, and the parameter contains the old data. The parameter then goes out of scope, destroying the data (and doing any necessary cleanup).

You don't need to change the swap function.

1 Reply

Kaus05

Reply to Alex [21] February 10, 2025 10:12 pm PST

Thank you Alex.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/
3. https://www.learncpp.com/cpp-tutorial/exception-specifications-and-noexcept/
4. https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/
5. https://www.learncpp.com/cpp-tutorial/timing-your-code/
6. https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/
7. https://www.learncpp.com/cpp-tutorial/stdmove/
8. https://www.learncpp.com/
9. https://www.learncpp.com/cpp-tutorial/rvalue-references/
10. https://www.learncpp.com/move-constructors-and-move-assignment/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-609398
13. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-609588
14. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-609839
15. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-608704
16. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-608403
17. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/
18. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-608213
19. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-607837
20. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-607469
21. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/#comment-607583
22. https://g.ezoic.net/privacy/learncpp.com