14.15 — Class initialization and copy elision

ALEX[1] NOVEMBER 14, 2024

Way back in lesson 1.4 -- Variable assignment and initialization (https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/)[2], we discuss 6 basic types of initialization for objects with fundamental types:

```
int a;          // no initializer (default initialization)
int b = 5;      // initializer after equals sign (copy initialization)
int c( 6 );     // initializer in parentheses (direct initialization)

// List initialization methods (C++11)
int d { 7 };    // initializer in braces (direct list initialization)
int e = { 8 }; // initializer in braces after equals sign (copy list initialization)
int f {};       // initializer is empty braces (value initialization)
```

All of these initialization types are valid for object with class types:

```cpp
#include <iostream>

class Foo
{
public:

    // Default constructor
    Foo()
    {
        std::cout << "Foo()\n";
    }

    // Normal constructor
    Foo(int x)
    {
        std::cout << "Foo(int) " << x << '\n';
    }

    // Copy constructor
    Foo(const Foo&)
    {
        std::cout << "Foo(const Foo&)\n";
    }
};

int main()
{
    // Calls Foo() default constructor
    Foo f1;            // default initialization
    Foo f2{};          // value initialization (preferred)

    // Calls foo(int) normal constructor
    Foo f3 = 3;        // copy initialization (non-explicit constructors only)
    Foo f4(4);         // direct initialization
    Foo f5{ 5 };       // direct list initialization (preferred)
    Foo f6 = { 6 };    // copy list initialization (non-explicit constructors only)

    // Calls foo(const Foo&) copy constructor
    Foo f7 = f3;       // copy initialization
    Foo f8(f3);        // direct initialization
    Foo f9{ f3 };      // direct list initialization (preferred)
    Foo f10 = { f3 }; // copy list initialization

    return 0;
}
```

In modern C++, copy initialization, direct initialization, and list initialization essentially do the same thing -- they initialize an object.

For all types of initialization:

- When initializing a class type, the set of constructors for that class are examined, and overload resolution is used to determine the best matching constructor. This may involve implicit conversion of arguments.
- When initializing a non-class type, the implicit conversion rules are used to determine whether an implicit conversion exists.

Key insight

There are three key differences between the initialization forms:

- List initialization disallows narrowing conversions.

- Copy initialization only considers non-explicit constructors/conversion functions. We'll cover this in lesson 14.16 -- Converting constructors and the explicit keyword[3].
- List initialization prioritizes matching list constructors over other matching constructors. We'll cover this in lesson 16.2 -- Introduction to std::vector and list constructors[4].

It is also worth noting that in some circumstances, certain forms of initialization are disallowed (e.g. in a constructor member initializer list, we can only use direct forms of initialization, not copy initialization).

Unnecessary copies

Consider this simple program:

```
#include <iostream>

class Something
{
    int m_x{};

public:
    Something(int x)
        : m_x{ x }
    {
        std::cout << "Normal constructor\n";
    }

    Something(const Something& s)
        : m_x { s.m_x }
    {
        std::cout << "Copy constructor\n";
    }

    void print() const { std::cout << "Something(" << m_x << ")\n"; }
};

int main()
{
    Something s { Something { 5 } }; // focus on this line
    s.print();

    return 0;
}
```

In the initialization of variable `s` above, we first construct a temporary `Something`, initialized with value `5` (which uses the `Something(int)` constructor). This temporary is then used to initialize `s`. Because the temporary and `s` have the same type (they are both `Something` objects), the `Something(const Something&)` copy constructor would normally be called here to copy the values in the temporary into `s`. The end result is that `s` is initialized with value `5`.

Without any optimizations, the above program would print:

```
Normal constructor
Copy constructor
Something(5)
```

However, this program is needlessly inefficient, as we've had to make two constructor calls: one to `Something(int)`, and one to `Something(const Something&)`. Note that the end result of the above is the same as if we had written the following instead:

```
1 | Something s { 5 }; // only invokes Something(int), no copy constructor
```

This version produces the same result, but is more efficient, as it only makes a call to `Something(int)` (no copy constructor is needed).

Copy elision

Since the compiler is free to rewrite statements to optimize them, one might wonder if the compiler can optimize away the unnecessary copy and treat `Something s { Something{5} };` as if we had written `Something s { 5 }` in the first place.

The answer is yes, and the process of doing so is called *copy elision*. **Copy elision** is a compiler optimization technique that allows the compiler to remove unnecessary copying of objects. In other words, in cases where the compiler would normally call a copy constructor, the compiler is free to rewrite the code to avoid the call to the copy constructor altogether. When the compiler optimizes away a call to the copy constructor, we say the constructor has been **elided**.

Unlike other types of optimization, copy elision is exempt from the "as-if" rule. That is, copy elision is allowed to elide the copy constructor even if the copy constructor has side effects (such as printing text to the console)! This is why copy constructors should not have side effects other than copying -- if the compiler elides the call to the copy constructor, the side effects won't execute, and the observable behavior of the program will change!

Related content

We discussed the as-if rule in lesson 5.4 -- The as-if rule and compile-time optimization (https://www.learncpp.com/cpp-tutorial/the-as-if-rule-and-compile-time-optimization/)[5].

We can see this in the above example. If you run the program on a C++17 compiler, it will produce the following result:

```
Normal constructor
Something(5)
```

The compiler has elided the copy constructor to avoid an unnecessary copy, and as a result, the statement that prints "Copy constructor" does not execute! Our program's observable behavior has changed due to copy elision!

Copy elision in pass by value and return by value

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. However, in certain cases, these copies may be elided. The following program demonstrates some of these cases:

```cpp
#include <iostream>

class Something
{
public:
    Something() = default;
    Something(const Something&)
    {
        std::cout << "Copy constructor called\n";
    }
};

Something rvo()
{
    return Something{}; // calls Something() and copy constructor
}

Something nrvo()
{
    Something s{}; // calls Something()
    return s;      // calls copy constructor
}

int main()
{
    std::cout << "Initializing s1\n";
    Something s1 { rvo() }; // calls copy constructor

    std::cout << "Initializing s2\n";
    Something s2 { nrvo() }; // calls copy constructor

        return 0;
}
```

In C++14 or older, with copy elision disabled, the above program would call the copy constructor 4 times:

- Once when `rvo` returns `Something` to `main`.
- Once when the return value of `rvo()` is used to initialize `s1`.
- Once when `nrvo` returns `s` to `main`.
- Once when the return value of `nrvo()` is used to initialize `s2`.

However, due to copy elision, it's likely that your compiler will elide most or all of these copy constructor calls. Visual Studio 2022 elides 3 cases (it doesn't elide the case where `nrvo()` returns by value), and GCC elides all 4.

It's not important to memorize when the compiler does / doesn't do copy elision. Just know that it is an optimization that your compiler will perform if it can. If you expect to see your copy constructor called and it isn't, copy elision is probably why.

Mandatory copy elision in C++17 C++17

Prior to C++17, copy elision was strictly an optional optimization that compilers could make. In C++17, copy elision became mandatory in some cases. In these cases, copy elision will be performed automatically (even if you tell your compiler not to perform copy elision).

Running the same example as above in C++17 or newer, the copy constructor calls that would otherwise occur when `rvo()` returns and when `s1` is initialized with that value are required to be elided. The initialization of `s2` with `nvro()` is not a mandatory elision case, and thus the 2 copy constructor calls that occur here may or may not be elided depending on your compiler and optimization settings.

In optional elision cases, an accessible copy constructor must be available (e.g. not deleted), even if the actual call to the copy constructor is elided.

In mandatory elision cases, an accessible copy constructor need not be available (in other words, mandatory elision can happen even if the copy constructor is deleted).

For advanced readers

In cases where optional copy elision isn't performed, move semantics may still allow an object to be moved instead of copied. We introduce move semantics in lesson 16.5 -- Returning std::vector, and an introduction to move semantics (https://www.learncpp.com/cpp-tutorial/returning-stdvector-and-an-introduction-to-move-semantics/)[6].

Next lesson
14.16 Converting constructors and the explicit keyword

3

Back to table of contents

7

Previous lesson
14.14 Introduction to the copy constructor

8

9

Aswin

June 27, 2025 11:14 pm PDT

Have a question.

```
#include <iostream>

class Something
{ int x,y;
public:
    Something() = default;
    Something(const Something& x) : x{other.x},y{other.y}
    {
        std::cout << "Copy constructor called\n";
    }
const int& getX(){return x;}
};

Something option1(const Something& P)
{
        int y  = 2
    return Something{P.getX(),y}; // calls Something() and copy constructor
}

Something option2(const Something& P)
{
        int y  = 2
        int x  = P.getX()
    return Something{x,y}; // calls Something() and copy constructor
}

int main()
{
    std::cout << "Initializing s1\n";
    Something s1 { option1() }; // calls copy constructor

    std::cout << "Initializing s2\n";
    Something s2 { option1() }; // calls copy constructor

        return 0;
}
```

Looking at the above code snippet, would function option1 or option2 be better? And why?

Last edited 2 minutes ago by Aswin

0 Reply

Erad

May 17, 2025 10:42 pm PDT

You wrote:
"Running the same example as above in C++17 or newer, the copy constructor calls that would otherwise occur when rvo() returns and when s1 is initialized with that value are required to be elided. **`The initialization of s2 with nvro() is not a mandatory elision case,`** and thus the 2 copy constructor calls that occur here may or may not be elided depending on your compiler and optimization settings."

Why is the highlighted not a mandatory elision case? From the code, I see that it's a bit different from the rvo() case since there is an actual declared object, s, here. However, since it's been returned by value, I think it's still going to be returned to main() as a temporary object; right? So why is rva's case mandatory and nrva's isn't?

Last edited 1 month ago by Erad

0 Reply

frank

September 17, 2024 7:03 pm PDT

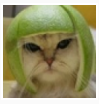
```
Something rvo()
{
    return Something{}; // calls Something() and copy constructor
}

Something nrvo()
{
    Something s{}; // calls Something()
    return s;      // calls copy constructor
```

I'm not sure why either of the returns call a copy constructor? as I understand it a copy constructor is just a way to construct something using an object as opposed to manually inputting values, but what are the objects in either return statement? I especially don't understand this:

```
return s;      // calls copy constructor
```

1 Reply



Alex Author

Reply to frank [12] September 22, 2024 7:03 pm PDT

Return by value creates a temporary object (in the scope of the caller) that is initialized using the object in the return statement. Since the temporary object and the object in the return statement (which may or may not be a temporary object itself) have the same type, the copy constructor will be used to make the copy.

Let's take rvo() for example. First, we create a default Something{} object using the default constructor. Then we return a temporary Something, initialized with this default Something. That invokes the copy constructor.

Same with the nrvo() case, except in this case we're returning a named Something rather than an unnamed Something.

1 Reply

Jan Schultke

May 23, 2024 11:23 pm PDT

I feel like this article could need some more C++17 modernization.

```
return Something{}; // calls Something() and copy constructor
```

In C++17, that's just not how it works. The prvalue in the return statement initializes the returned object, which is the same object as `s1`.

This has absolutely nothing to do with RVO, which is an ABI-related optimization. This is not optimizing away the call to the copy constructor; there was never any call to the copy constructor to begin with.

It should also be mentioned that even if NRVO doesn't happen, `return s;` will call the move constructor if possible because `s` is an implicitly movable entity, and `s` is an xvalue in this context. The example doesn't have a `Something` with a move constructor, but if it had one, the distinction would matter.

1 Reply

PollyWantsACracker

Reply to Jan Schultke [13] May 13, 2025 9:34 am PDT

master came back to check his knowledge

0 Reply

Alex Author

Reply to Jan Schultke [13] May 26, 2024 6:37 pm PDT

Thanks for the thoughts. Made a few tweaks to the lesson accordingly:

1. Made clearer that the initial discussion of the code in the example applies to C++14 or older (with copy elision disabled).
2. Renamed the bottom section "Mandatory copy elision in C++17".
3. Added an advanced box with a link to the lesson that introduces move semantics.

3 Reply

Baker

May 3, 2024 6:29 am PDT

When you do return something{}, a copy construction is happening because something{} is essentially being copied as the return value?

0 Reply

Alex Author

Reply to Baker [14] May 4, 2024 11:53 am PDT

Yes.

1 Reply

Rohit

February 3, 2024 10:39 pm PST

I have created my own string class using char* and I have implemented copy and move semantics and also overloaded plus operator.

```
MyString s1{"Hello "};
MyString s2{"World"};

MyString s3{s1+s2};
```

In this case only parameterized constructor got called 3 times.
(1 for s1, 1 for s2 and 1 for creating temporary inside operator+)

Does that mean call to move constructor got elided or something else is happening here?

Last edited 1 year ago by Rohit

0 Reply

Alex Author

Reply to Rohit [15] February 5, 2024 3:00 pm PST

I would have expected s1 and s2 to call converting constructor `MyString(const char*)`, and s3 to call the move constructor, since s1+s2 should be a temporary MyString. Hard to know why it isn't doing so without seeing the code.

0 Reply

Rohit

Reply to Alex [16] February 5, 2024 11:44 pm PST

```
MyString::MyString(const char* str) : m_size(strlen(str))
{
    std::cout << "Converting cunst\n";
    m_Buffer = new char[m_size + 1];
    memcpy(m_Buffer, str, m_size);
    m_Buffer[m_size] = '\0';
}

MyString::MyString(const String& other) : m_size(other.m_size)
{
        std::cout << "Copy cunst\n";
    m_Buffer = new char[other.m_size + 1];
    memcpy(m_Buffer, other.m_Buffer, m_size + 1);
}

MyString::MyString(String&& other) noexcept
    : m_Buffer{ other.m_Buffer }, m_size{ other.m_size }
{
    std::cout << "Move cunst\n";
    other.m_Buffer = nullptr;
    other.m_size = 0;
}
```

for following code:

MyStrings1 = " Hello";
MyStrings2 = "World";

MyStrings3 = s1 + s2;

Output:

Converting cunst
Converting cunst
Converting cunst

Last edited 1 year ago by Rohit

0 Reply

Alex Author

Reply to Rohit [17] February 6, 2024 1:16 pm PST

Best guess is that your move constructor is being elided. It is constructing `MyString3` directly (with the return value of `s1 + s2`).

0 Reply

Rohit

Reply to Alex [18] February 6, 2024 9:54 pm PST

Total 2 (One copy and one move) cunstructors are elided right?

Last edited 1 year ago by Rohit

0 Reply

Alex Author

Reply to Rohit [19] February 7, 2024 12:14 pm PST

No, either the move or the copy constructor would get invoked to initialize s3.

0 Reply

Suryaansh

December 13, 2023 3:54 am PST

At the end of the very first section you mention - "It is also worth noting that in some circumstances, certain forms of initialization are disallowed (e.g. in a constructor member initializer list, we can only use direct forms of initialization)."
Can you please elaborate what's meant by this line?

```
Foo(int x, int y): m_x { x }, m_y { y }
{
 //Constructor using a member initialization list that uses list initialization and
NOT direct/copy initialization
}
```

0 Reply

Alex Author

Reply to Suryaansh [20] December 15, 2023 3:07 pm PST

It means:

```
Foo(int x): m_x { x } // direct list init okay
{}
Foo(int x): m_x ( x ) // direct init okay
{}
Foo(int x): m_x = x  // copy init disallowed
{}
```

2 Reply

resident of flavourtown

November 18, 2023 1:13 pm PST

Hi, I have a question about some of the examples. In a lot of them, you create a copy constructor, but do not initialise the variables:

```
// Copy constructor
Foo(const Foo&)
{
std::cout << "Foo(const Foo&)\n";
}
```

I would just like to confirm that because you are not initialising any variables, they will not actually get the values of the object you're passing in, and instead will be zero initialised?

1 Reply

Alex Author

Reply to resident of flavourtown [21] November 18, 2023 3:40 pm PST

Almost. Because we're not initializing the members of the implicit object, those members will be initialized to their default values (if default member initializers are provided) or default-initialized otherwise.

2 Reply

CppLearner
October 11, 2023 8:15 am PDT

For this simple code below, does the instantiation of object `c` at line 7 involve any copy elision? My impression is that without any copy elision or optimization, a temporary `C` object `C{1, 2}` will be first created and followed by a call to the copy constructor from `C` to instantiate `c`.

```
1 class C
2 {
3 public:
4     C(int i, int j){}
5 };
6
7 C c = {C{1, 2}};
```

0 Reply

Alex Author
Reply to CppLearner [22] October 13, 2023 11:28 pm PDT

Yes, I'd expect this to be equivalent to `C c = { 1, 2 }`.

1 Reply

Pantera
September 28, 2023 10:28 pm PDT

Thank you for these awesome tutorials!

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/
3. https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/
5. https://www.learncpp.com/cpp-tutorial/the-as-if-rule-and-compile-time-optimization/
6. https://www.learncpp.com/cpp-tutorial/returning-stdvector-and-an-introduction-to-move-semantics/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/
9. https://www.learncpp.com/class-initialization-and-copy-elision/
10. https://www.learncpp.com/cpp-tutorial/overloading-operators-using-normal-functions/
11. https://gravatar.com/

12. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-602070
13. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-597488
14. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-596582
15. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-593219
16. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-593275
17. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-593300
18. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-593331
19. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-593348
20. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-590917
21. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-589984
22. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/#comment-588553
23. https://g.ezoic.net/privacy/learncpp.com