


14.15 — Class initialization and copy elision

 learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/

Way back in lesson [1.4 -- Variable assignment and initialization](#), 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:

```

#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](#).
- 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](#).

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:

```

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 -- Constant expressions and compile-time optimization](#).

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:

```

#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;
}

```

Without optimization, 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, and if you expect to see your copy constructor called and it isn't, copy elision is probably why.

Copy elision errata

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 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).