13.13 — Copy initialization

ALEX SEPTEMBER 3, 2021

Consider the following line of code:

```
int x = 5;
```

This statement uses copy initialization to initialize newly created integer variable x to the value of 5.

However, classes are a little more complicated, since they use constructors for initialization. This lesson will examine topics related to copy initialization for classes.

Copy initialization for classes

Given our Fraction class:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
        : m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}
```

Consider the following:

```
int main()
{
    Fraction six = Fraction(6);
    std::cout << six;
    return 0;
}
```

If you were to compile and run this, you'd see that it produces the expected output:

```
6/1
```

This form of copy initialization is evaluated the same way as the following:

```
Fraction six(Fraction(6));
```

And as you learned in the previous lesson, this can potentially make calls to both Fraction(int, int) and the Fraction copy constructor (which may be elided for performance reasons). However, because eliding isn't guaranteed (prior to C++17, where elision in this particular case is now mandatory), it's better to avoid copy initialization for classes, and use uniform initialization instead.

> **Best practice**
>
> Avoid using copy initialization, and use uniform initialization instead.

Other places copy initialization is used

There are a few other places copy initialization is used, but two of them are worth mentioning explicitly. When you pass or return a class by value, that process uses copy initialization.

Consider:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
        : m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

        // Copy constructor
    Fraction(const Fraction& copy) :
        m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
    {
        // no need to check for a denominator of 0 here since copy must already be a valid Fraction
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction& f1);
        int getNumerator() { return m_numerator; }
        void setNumerator(int numerator) { m_numerator = numerator; }
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

Fraction makeNegative(Fraction f) // ideally we should do this by const reference
{
    f.setNumerator(-f.getNumerator());
    return f;
}

int main()
{
    Fraction fiveThirds(5, 3);
    std::cout << makeNegative(fiveThirds);

    return 0;
}
```

In the above program, function makeNegative takes a Fraction by value and also returns a Fraction by value. When we run this program, we get:

```
Copy constructor called
Copy constructor called
-5/3
```

The first copy constructor call happens when fiveThirds is passed as an argument into makeNegative() parameter f. The second call happens when the return value from makeNegative() is passed back to main().

In the above case, both the argument passed by value and the return value can not be elided. However, in other cases, if the argument or return value meet specific criteria, the compiler may opt to elide the copy constructor. For example:

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

Something foo()
{
    return Something(); // copy constructor normally called here
}
Something goo()
{
    Something s;
    return s; // copy constructor normally called here
}

int main()
{
    std::cout << "Initializing s1\n";
    Something s1 = foo(); // copy constructor normally called
here

    std::cout << "Initializing s2\n";
    Something s2 = goo(); // copy constructor normally called
here
}
```

The above program would normally call the copy constructor 4 times -- however, due to copy elision, it's likely that your compiler will elide most or all of the cases. Visual Studio 2019 elides 3 (it doesn't elide the case where s is returned), and GCC elides all 4.

Next lesson
13.14 Converting constructors, explicit, and delete

Back to table of contents

Previous lesson
13.12 The copy constructor

68 COMMENTS Newest

Rekt0707
November 30, 2021 9:12 pm

Car s1= temp(6);
Car s2{temp(6)};

Would both behave in the same manner?

where Car is class name and s1, s2, temp are it's objects.

Last edited 3 months ago by Rekt0707

0 Reply

> **Alex** Author
> Reply to Rekt0707 December 2, 2021 4:38 pm
>
> If the copy constructor is explicit, the former won't even compile, whereas the latter will.
>
> But assuming temp is an instance of Car and the copy constructor is not explicit, I believe these will behave identically.
>
> 0 Reply

George
October 14, 2021 1:51 am

I don't quite get the difference in both functions, so to warrant a different behaviour. In both cases we return copy to an object, in both cases the original object is created within the function (f is local variable due to passing by value, as well as s).

```
Something goo()
{
    Something s;
    return s; // copy constructor normally called here
}
```

and this

```
Fraction makeNegative(Fraction f) // ideally we should do this by const reference
{
    f.setNumerator(-f.getNumerator());
    return f;
}
```

Last edited 4 months ago by George

0 Reply

> **Alex** Author
> Reply to George October 14, 2021 12:52 pm
>
> I'm not sure of the technical reason the former is eligible for copy elision and the latter isn't. There's a lot of information about copy elision here: https://en.cppreference.com/w/cpp/language/copy_elision, but it still doesn't always cover the "why".
>
> 0 Reply

Knight
October 10, 2021 9:34 pm

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

Hi, dear admins what is "Something&" referenced? Is referenced public copy constructor that created from the c++(Something()=default;)? Do not need the referenced name at here?

0 Reply

> **Alex** Author
> Reply to Knight October 11, 2021 6:11 pm
>
> I'm not quite sure I understand what you're asking. Are you asking why `const Something&` isn't `const Something& someName`? An identifier isn't required for a function parameter if the parameter isn't used in the body of the function.
>
> 0 Reply
>
> > **Knight**
> > Reply to Alex October 11, 2021 10:17 pm
> >
> > Sorry about my English.
> > Am I right: Because the Something& does not use body of the function, so the Something& can without the parameter: someName? And what is Something& reference to? I think references must be initialized.
> >
> > *Last edited 4 months ago by Knight*
> >
> > 0 Reply
> >
> > > **King**
> > > Reply to Knight October 14, 2021 4:42 am
> > >
> > > Ad it is initialized, but we don't have to provide identifier, because as Alex said: the parameter isn't used in the body of the function.
> > > "Something& can without the parameter" - the parameter exists, but it has no identifier.
> > >
> > > 0 Reply

T. A
September 2, 2021 1:08 pm

For the last example, you say:

> In this case, the compiler will probably elide the copy constructor, even though variable s is returned by value.

There are two cases in the code block where the compiler (pre c++17) will call the copy constructor:

1. When we're returning by value on line 8
2. When we're doing the copy initialization on line 13.

By saying the compiler will elide the copy constructor (from c++17), do you mean that will happen in both cases, or just in the first case?

Thanks for this fantastic resource, by the way.

0 Reply

> **Alex** Author
> Reply to T. A September 3, 2021 2:05 pm
>
> It can actually happen in both cases, even in pre-C++17 compilers. I updated the example a bit to make it slightly more interesting and show that compilers do different things.
>
> 0 Reply
>
> > **T. A**
> > Reply to Alex September 4, 2021 9:29 am
> >
> > Got it. Thanks, Alex.
> >
> > 0 Reply

Waldo Lemmer
July 24, 2021 1:07 am

1. `Fraction &f1`
2. "Rule" should be "Best practice"
3. I think the `:`s should be on the next lines at the member initializer lists

1 and 3 apply to the previous lesson as well.

0 Reply

typoman
May 27, 2021 2:45 am

"The first copy constructor call happens when fiveThirds passed as an argument into makeNegative() parameter f."
small typo

0 Reply

> **nascardriver** Sub-admin
> Reply to typoman June 6, 2021 7:49 am
>
> The small typo now fixed :)
>
> 0 Reply

Andreas Krug
March 27, 2021 12:03 am

Please put Rule: Avoid using copy initialization, and use uniform initialization instead. in a beautiful green box.

0 Reply

chai
February 17, 2021 10:07 am

hello.
Rule: Avoid using copy initialization, and use uniform initialization instead.
Is direct initialization (like ClassA x(4);), the same as copy initialization (like ClassA x = 4;) ?

thanks.

0 Reply

Copy_constructor
August 31, 2020 3:13 pm

I ran the very last example and I found that the compiler still called Copy constructor but in your example it says the compiler doesn't call Copy constructor. (I am using compiler C++20)

```
#include <iostream>
#include <cassert>
class Something
{
private:
    int m_something{};
public:
    Something(int something = 0) : m_something{ something }
    {

        std::cout << "DeFaul\n";

    }
    Something(const Something& object)
    {
        m_something = object.m_something;
        std::cout << "Copy\n";

    }
};
Something foo()
{

    Something s;
    return s;
}

int main()
{

    Something s = foo();
    return 0;

}
```

Output:
DeFault
Copy

0 Reply

> **nascardriver** Sub-admin
> Reply to Copy_constructor August 31, 2020 11:18 pm
>
> The compiler will "probably" elide the copy. If you increase your compiler's optimization level, you shouldn't get a copy constructor call either.
> If you change `foo`'s body to
>
> ```
> return {};
> ```
>
> The copy is guaranteed to be elided since C++17.
>
> 1 Reply

Passionate_programmer
August 31, 2020 12:36 pm

1. Why did I get the following messages and not any 'Copy constructor"?

2)I didn't even get 'Default Constructor' from the first statement.

Default Constructor
Default Constructor
Default Constructor

```
1  #include <iostream>
2  #include <cassert>
3
4  class Fraction
5  {
6  private:
7      int m_numerator;
8      int  m_denominator;
9
10 public:
11     Fraction(int numerator=0, int denominator=1):
12         m_numerator{numerator}, m_denominator{denominator}
13     {
14         std::cout << "Default Constructor \n";
15         assert(m_denominator != 0 );
16     }
17     // Copy constructor
18     Fraction(const Fraction& fraction) :
19         m_numerator(fraction.m_numerator+2), m_denominator(fraction.m_denominator+2)
20     {
21         // no need to check for a denominator of 0 here since fraction must already be a valid Fraction
22         std::cout << "Copy constructor called\n"; // just to prove it works
23     }
24     friend std::ostream& operator<<(std::ostream& output, const Fraction& fraction)
25     {
26         output << fraction.m_numerator << '/' << fraction.m_denominator;
27         return output;
28     }
29     int getNumberator()  const { return m_numerator; }
30     void setNumerator(int numerator) { m_numerator = numerator; }
31
32
33
34 };
35 Fraction makeNegative(Fraction fraction)
36 {
37     fraction.setNumerator(-fraction.getNumberator());
38     return fraction;
39 }
40 Fraction test()
41 {
42
43     return Fraction(1, 3);
44 }
45
46 int main()
47 {
48
49
50     Fraction test();
51     Fraction testMe{ test() };
52     Fraction testMeV2( test() );
53     Fraction testMeV3 = test();
54
55     return 0;
56
57 }
```

0 Reply

> **nascardriver** Sub-admin
> Reply to Passionate_programmer August 31, 2020 11:16 pm
>
> Line 50 is a forward declaration of a function. Use list initialization.
> All of the copies were elided.
>
> 0 Reply