9.12 — Copy initialization

BY ALEX ON JUNE 5TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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.

Rule: 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:

```cpp
#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 Fra
        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 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:

```
class Something
{
};

Something foo()
{
  Something s;
  return s;
}

int main()
{
  Something s = foo();
}
```

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

9.13 -- Converting constructors, explicit, and delete

Index

9.11 -- The copy constructor

C++ TUTORIAL | PRINT THIS POST

37 comments to 9.12 — Copy initialization

David
January 22, 2020 at 5:34 am · Reply

In the following code,I have a question.As you mention,there are two occasions where the copy initialization is used.
First,it is pass a class by value. Second,it is return a class by value.However,the "the copy constructor is called" only appeared once.Does my program should be fixed? Thanks for replying.

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

class Fraction
{
    int m_num, m_deno;
public:
    Fraction(int num = 0, int deno = 1) :m_num{ num }, m_deno{ deno }
    {
        assert(deno != 0);
    }

    Fraction(const Fraction& frac):m_num{frac.m_num},m_deno{frac.m_deno}
    {
        std::cout << "The copy constructor is called\n";
    }

    int getNum()
    {
        return m_num;
    }

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

};

std::ostream& operator<<(std::ostream& out, const Fraction& frac)
{
    out << frac.m_num << "/" << frac.m_deno << "\n";
30        return out;
31    }
32
33    Fraction makeNegative(Fraction frac)
34    {
35
36        return Fraction{ -frac.getNum() };
37    }
38
39
40    int main()
41    {
42        Fraction six{ 6 };
43        std::cout << makeNegative(six);
44        return 0;
45    }
```

nascardriver
January 22, 2020 at 5:40 am · Reply

Because you're creating the `Fraction` in line 36 (Instead of creating a variable and returning that variable), the compiler has to elide the call to the copy constructor.
Even if you create a temporary variable, the compiler will probably optimize it away. Constructors are allowed to be elided, that's why you should only use them for initializations.

Louis Cloete
April 11, 2019 at 1:50 pm · Reply

Under "Other places copy initialization is used" the comment at the makeNegative() function says we should've done it by const reference. I don't think const reference will work, since it calls a non-const member function (which mutates state, so we can't just make the method const).

So: (I didn't compile to test)

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

Alex
April 11, 2019 at 5:30 pm · Reply

You'd have to implement the function in a different way. Maybe like this:

```
Fraction makeNegative(const Fraction &f)
{
    return Fraction(-f.getNumerator(), f.getDenominator());
}
```

This would require a few other changes (getNumerator() should be const, and getDenominator() would need to exist)

Asgar
February 27, 2019 at 11:18 am · Reply

Hi Alex,

I see there are two possible occurrences of calls made to the copy constructor in your example:
1. return s; will invoke the copy constructor to make a temporary object. Then local s will be destroyed when the method exits.
2. Something s = ... will invoke the copy constructor again to initialize the main() s using the temporary object. Then the temporary object will be destroyed.

```
class Something
{
};

Something foo()
{
  Something s;
  return s;
}

int main()
{
  Something s = foo();
}
```

Please let me know if there are other occurrences I missed. Also, how would the compiler elide a call to the copy constructor in this example? I mean, which of the 2 occurrences would be elided?

nascardriver
February 28, 2019 at 7:28 am · Reply

> how would the compiler elide a call to the copy constructor in this example?

```
void foo(Something &s)
{
  s = Something{};
}

int main(void)
{
  Something s;

  foo(s);

  return 0;
}
```

> which of the 2 occurrences would be elided?
The one that happens when @foo returns.

Great Ape
February 6, 2019 at 3:06 pm · Reply

I still don't really understand the difference between copy, direct and uniform initialisation except that they have different syntaxes. I get that direct/uniform initialisation is better for initialising class objects because elision isn't guaranteed for copy initialisation, but I still don't know what direct, uniform and copy initialisation actually mean in a general sense, even for fundamental data types. In other words, what is actually happening behind the scenes for each type of initialisation? I've been wondering this ever since the first chapter.

nascardriver

February 7, 2019 at 6:27 am · Reply

Copy elision is guaranteed for copy- and direct initialization since C++17.
I don't know if there's still a difference between copy- and direct initialization at all.
As for copy/direct vs brace initialization:
1. You want to default-initialize a value

```
int i = /* Can't do it */;
int i(); // Can't do it. @i is a function (Most vexing parse).
int i{}; // Initialized to 0 (Or whatever the default value of the type is.
```

2. You want to list-initialize

```
int arr[] = { 1, 2, 3 }; // We need to mix copy/brace
int arr[]({ 1, 2, 3 }); // We need to mix direct/brace
int arr[]{ 1, 2, 3 }; // No mixing. Brace-initializers work almost everywhere (Hence "
```

nascardriver
January 8, 2019 at 7:41 am · Reply

Hi Alex!

Paragraph "And as you learned in the previous lesson [...]".
This is no longer true in C++17 and later. I'm not entirely certain about it, please double-check. If I'm wrong, let me know.
https://en.cppreference.com/w/cpp/language/copy_elision

Alex
January 9, 2019 at 5:25 pm · Reply

That's my understanding as well. I've updated the lesson to note that elision is mandatory in this case.

Udit
December 9, 2018 at 2:29 am · Reply

Hi, I want to ask that:
Whether there is any major performance gain in uniform initialization compared to copy initialization?
Since copy constructor take argument as reference only, so it should not be that much slow compared to direct initialization?

nascardriver
December 9, 2018 at 2:34 am · Reply

Hi Udit!

```
A a = A(1, 2, 3);
```

This could create an @A on the right side (Constructed via A(int, int, int)) and an @A on the left side (Constructed via (A(const A&)).
Not only do you have 1 unnecessary call (to the copy constructor), but you also have to move/copy all the data from the right to the left, which is slow.
Direct- and uniform initialization don't have this problem. Use uniform initialization.

Yusef Abdullah
August 29, 2018 at 6:00 pm · Reply

Hello Alex, first of all thanks for the great lesson,

I really don't understand this piece of code

```
[/code]
Fraction(const Fraction &copy) :
    m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
[/code]
```

Why can't I do something like this(just like the constructor)

```
[/code]
Fraction(const Fraction &copy) :
    m_numerator(copy
numerator), m_denominator(copy.denominator)
[/code]
```

Alex
September 3, 2018 at 11:56 am · Reply

Sorry, what's the difference? You replaced a period with a line break?

YTXbaiaLrs
March 31, 2019 at 3:53 pm · Reply

I think Yusef means replacing "m_numerator" with "numerator" and "m_denominator" with "denominator".

You can't do this because "numerator" and "denominator" are not members of "copy".

This works in the constructor because "numerator" and "denominator" are local variables (look at the parameters of the constructor Alex has).

Siddharth Sharma
March 13, 2018 at 4:59 pm · Reply

Hey Alex, why is the copy constructor elided in this case?

```
class Something
{
};

Something foo()
{
  Something s;
  return s;
}

int main()
{
  Something s = foo();
}
```

Alex

March 16, 2018 at 2:43 pm · Reply

Because it can be. :)

The rules around what can and can't be elided are a bit complicated, but this example qualifies as one of those cases.

DOG_TRAINER
November 25, 2017 at 7:49 am · Reply

From my understanding (WARNING: I might be wrong), copy constructor is elided because C++ treats it as a direct or uniform initialization.

KnowMore
June 22, 2017 at 1:43 am · Reply

"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."
What is the specific criteria that has to be met?

Alex
June 23, 2017 at 8:42 am · Reply

The answer to that question is fairly complicated. See this page on **copy elision**.

Omri
June 6, 2017 at 10:32 pm · Reply

Thank you for the answer.
So, a ref var can bind to an l-value, meaning memory locations that hold l-values and can be reached by memory address.
A const ref var can bind to these memory locations and also to "temporary locations" where, for example, anonymous values are stored for a duration dictated by the context of their creation (for whatever this means)...
When a const ref var binds to such a temporary and non addressable value (perhaps living in a register?), is there a copy of this value being created elsewhwere so it can be further referenced by the const ref var during its lifetime or otherwise...? Thank you again.

Alex
June 7, 2017 at 9:26 am · Reply

The lifetime of a temporary object is extended to match the lifetime of the const reference in this case.

Omri
June 5, 2017 at 9:04 pm · Reply

In: frac class I defined, in addition to a default constructor, a copy constructor as follows leaving out the const qualifier:

```
frac(frac &f) :m_num(f.m_num), m_denom(f.m_denom)
{std::cout << "Copy constructor called\n";}
```

as opposed to:
frac(const frac &f) :m_num(f.m_num), m_denom(f.m_denom)
{std::cout << "Copy constructor called\n";}
In main:
frac f1(1,2); compiled. Following it: frac f2(f1); compiled. Following it frac f3(frac(3,6)); did not compile.

The compiler complained:
cannot bind non-const lvalue reference of type 'frac&' to an rvalue of type 'frac' frac f3(frac(3,6));

When I add the const qualifier into the copy constructor definition all goes well.
As I understand this message, It seems that: frac(3,6) does not bind to: frac &f but does bind to: const frac &f.
I understand that anonymous frac(3,6) is an rvalue.
Being anonymous It perhaps cannot be referenced later since its existance in memory is only upon its creation and immediate usage (what ever this means)...
How does the const qualifier solve this?

Alex
June 6, 2017 at 9:19 am · Reply

Non-const references can only bind to l-values. Const references can bind to l-values and r-values. Your anonymous frac is an r-value, so it can only be bound to a const reference.

Sihoo
July 8, 2017 at 3:26 pm · Reply

thank you.

susenj
May 28, 2017 at 4:02 am · Reply

Pardon my ignorance, but I still didn't get why the second call of copy constructor would be made in the last example of negative fractions.

Alex
May 28, 2017 at 11:32 am · Reply

The lesson explains it here:

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

Essentially, we get a call to the copy constructor whenever we have a class variable passed by value. That happens twice in this example: once passing fiveThirds to the function, and once when the return value is passed back.

Vijay
August 30, 2016 at 6:06 am · Reply

Hi Alex, why in this program compiler elide the copy constructor?

```
class Something
{
};
```

```

Something foo()
{
  Something s;
  return s;
}

int main()
{
  Something s = foo();
}
```

Nazime
July 19, 2016 at 9:48 am · Reply

In the first exemple in the main you forgot the " () " juste after the main
Consider the following:

```
int main    //Here no ()
{
    Fraction six = Fraction(6);
    std::cout << six;
    return 0;
}
```

Whene i copied that i get an error and i didn't notice that at the first place x)
i Like this website *_*

Alex
July 22, 2016 at 1:53 pm · Reply

Fixed, thanks!

Andrew Terry
July 15, 2016 at 4:28 am · Reply

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

6

No it is 6/1

Alex
July 15, 2016 at 8:53 am · Reply

Fixed. Thanks for pointing that out.

Andrew Terry
July 15, 2016 at 9:10 am · Reply

Ur welcome mate, and thanks for the Description and Site, really helpful!

Nyap
June 29, 2016 at 11:54 am · Reply

This is whats really getting me confused

However, because eliding isn’t guaranteed, it’s better to avoid copy initialization for classes, and use direct or uniform initialization instead.

But isn't that the same for all forms of initialization? that eliding is not guaranteed? or is the compiler just less likely to elide if its in copy init form?

Alex
July 1, 2016 at 10:40 am · Reply

There's no need to elide direct or uniform initialization because they are already as efficient as possible (there's no extraneous objects created).

anand
June 12, 2016 at 5:09 am · Reply

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

```
Fraction fiveThirds(Fraction(6))
```

;
is it not supposed to be:

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

;

Alex
June 13, 2016 at 11:31 am · Reply

Yes, thank you for pointing that out.