14.14 — Introduction to the copy constructor

ALEX[1] DECEMBER 19, 2024

Consider the following program:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

int main()
{
    Fraction f { 5, 3 };  // Calls Fraction(int, int) constructor
    Fraction fCopy { f }; // What constructor is used here?

    f.print();
    fCopy.print();

    return 0;
}
```

You might be surprised to find that this program compiles just fine, and produces the result:

```
Fraction(5, 3)
Fraction(5, 3)
```

Let's take a closer look at how this program works.

The initialization of variable `f` is just a standard brace initialization that calls the `Fraction(int, int)` constructor.

But what about the next line? The initialization of variable `fCopy` is also clearly an initialization, and you know that constructor functions are used to initialize classes. So what constructor is this line calling?

The answer is: the copy constructor.

The copy constructor

A **copy constructor** is a constructor that is used to initialize an object with an existing object of the same type. After the copy constructor executes, the newly created object should be a copy of the object passed in as the initializer.

An implicit copy constructor

If you do not provide a copy constructor for your classes, C++ will create a public **implicit copy constructor** for you. In the above example, the statement `Fraction fCopy { f };` is invoking the implicit copy constructor to initialize `fCopy` with `f`.

By default, the implicit copy constructor will do memberwise initialization. This means each member will be initialized using the corresponding member of the class passed in as the initializer. In the example above, `fCopy.m_numerator` is initialized using `f.m_numerator` (which has value `5`), and `fCopy.m_denominator` is initialized using `f.m_denominator` (which has value `3`).

After the copy constructor has executed, the members of `f` and `fCopy` have the same values, so `fCopy` is a copy of `f`. Thus calling `print()` on either has the same result.

Defining your own copy constructor

We can also explicitly define our own copy constructor. In this lesson, we'll make our copy constructor print a message, so we can show you that it is indeed executing when copies are made.

The copy constructor looks just like you'd expect it to:

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    // Copy constructor
    Fraction(const Fraction& fraction)
        // Initialize our members using the corresponding member of the parameter
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

int main()
{
    Fraction f { 5, 3 };  // Calls Fraction(int, int) constructor
    Fraction fCopy { f }; // Calls Fraction(const Fraction&) copy constructor

    f.print();
    fCopy.print();

    return 0;
}
```

When this program is run, you get:

```
Copy constructor called
Fraction(5, 3)
Fraction(5, 3)
```

The copy constructor we defined above is functionally equivalent to the one we'd get by default, except we've added an output statement to prove the copy constructor is actually being called. This copy constructor is invoked when `fCopy` is initialized with `f`.

A reminder

Access controls work on a per-class basis (not a per-object basis). This means the member functions of a class can access the private members of any class object of the same type (not just the implicit object).

We use that to our advantage in the `Fraction` copy constructor above in order to directly access the private members of the `fraction` parameter. Otherwise, we would have no way to access those members directly (without adding access functions, which we might not want to do).

A copy constructor should not do anything other than copy an object. This is because the compiler may optimize the copy constructor out in certain cases. If you are relying on the copy constructor for some behavior other than just copying, that behavior may or may not occur. We discuss this further in lesson 14.15 -- Class initialization and copy elision (https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/)[2].

Best practice

Copy constructors should have no side effects beyond copying.

Prefer the implicit copy constructor

Unlike the implicit default constructor, which does nothing (and thus is rarely what we want), the memberwise initialization performed by the implicit copy constructor is usually exactly what we want. Therefore, in most cases, using the implicit copy constructor is perfectly fine.

Best practice

Prefer the implicit copy constructor, unless you have a specific reason to create your own.

We'll see cases where the copy constructor needs to be overwritten when we discuss dynamic memory allocation (21.13 -- Shallow vs. deep copying (https://www.learncpp.com/cpp-tutorial/shallow-vs-deep-copying/)[3]).

The copy constructor's parameter must be a reference

It is a requirement that the parameter of a copy constructor be an lvalue reference or const lvalue reference. Because the copy constructor should not be modifying the parameter, using a const lvalue reference is preferred.

Best practice

If you write your own copy constructor, the parameter should be a const lvalue reference.

Pass by value and the copy constructor

When an object is passed by value, the argument is copied into the parameter. When the argument and parameter are the same class type, the copy is made by implicitly invoking the copy constructor.

This is illustrated in the following example:

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    // Copy constructor
    Fraction(const Fraction& fraction)
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n";
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

void printFraction(Fraction f) // f is pass by value
{
    f.print();
}

int main()
{
    Fraction f{ 5, 3 };

    printFraction(f); // f is copied into the function parameter using copy
constructor

    return 0;
}
```

On the author's machine, this example prints:

```
Copy constructor called
Fraction(5, 3)
```

In the above example, the call to `printFraction(f)` is passing `f` by value. The copy constructor is invoked to copy `f` from `main` into the `f` parameter of function `printFraction()`.

Return by value and the copy constructor

In lesson 2.5 -- Introduction to local scope (https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/)[4], we noted that return by value creates a temporary object (holding a copy of the return value) that is passed back to the caller. When the return type and the return value are the same class type, the temporary object is initialized by implicitly invoking the copy constructor.

For example:

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    // Copy constructor
    Fraction(const Fraction& fraction)
        : m_numerator{ fraction.m_numerator }
        , m_denominator{ fraction.m_denominator }
    {
        std::cout << "Copy constructor called\n";
    }

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

void printFraction(Fraction f) // f is pass by value
{
    f.print();
}

Fraction generateFraction(int n, int d)
{
    Fraction f{ n, d };
    return f;
}

int main()
{
    Fraction f2 { generateFraction(1, 2) }; // Fraction is returned using copy
constructor

    printFraction(f2); // f2 is copied into the function parameter using copy
constructor

    return 0;
}
```

When `generateFraction` returns a `Fraction` back to `main`, a temporary `Fraction` object is created and initialized using the copy constructor.

Because this temporary is used to initialize `Fraction f2`, this invokes the copy constructor again to copy the temporary into `f2`.

And when `f2` is passed to `printFraction()`, the copy constructor is called a third time.

Thus, on the author's machine, this example prints:

```
Copy constructor called
Copy constructor called
Copy constructor called
Fraction(1, 2)
```

If you compile and execute the above example, you may find that only two calls to the copy constructor occur. This is a compiler optimization known as *copy elision*. We discuss copy elision further in lesson 14.15 -- Class initialization and copy elision (https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/)[2].

Using `= default` to generate a default copy constructor

If a class has no copy constructor, the compiler will implicitly generate one for us. If we prefer, we can explicitly request the compiler create a default copy constructor for us using the `= default` syntax:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    // Explicitly request default copy constructor
    Fraction(const Fraction& fraction) = default;

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

int main()
{
    Fraction f { 5, 3 };
    Fraction fCopy { f };

    f.print();
    fCopy.print();

    return 0;
}
```

Using `= delete` to prevent copies

Occasionally we run into cases where we do not want objects of a certain class to be copyable. We can prevent this by marking the copy constructor function as deleted, using the `= delete` syntax:

```cpp
#include <iostream>

class Fraction
{
private:
    int m_numerator{ 0 };
    int m_denominator{ 1 };

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

    // Delete the copy constructor so no copies can be made
    Fraction(const Fraction& fraction) = delete;

    void print() const
    {
        std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
    }
};

int main()
{
    Fraction f { 5, 3 };
    Fraction fCopy { f }; // compile error: copy constructor has been deleted

    return 0;
}
```

In the example, when the compiler goes to find a constructor to initialize `fCopy` with `f`, it will see that the copy constructor has been deleted. This will cause it to emit a compile error.

As an aside...

You can also prevent the public from making copies of class object by making the copy constructor private (as private functions can't be used by the public). However, a private copy constructor *can* still be used by other members of the class, so this solution is not advised unless that is desired.

For advanced readers

The **rule of three** is a well known C++ principle that states that if a class requires a user-defined copy constructor, destructor, or copy assignment operator, then it probably requires all three. In C++11, this was expanded to the **rule of five**, which adds the move constructor and move assignment operator to the list.

Not following the rule of three/rule of five is likely to lead to malfunctioning code. We'll revisit the rule of three and rule of five when we cover dynamic memory allocation.

We discuss destructors in lesson 15.4 -- Introduction to destructors (https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/)[5] and 19.3 -- Destructors (https://www.learncpp.com/cpp-tutorial/destructors/)[6], and copy assignment in lesson 21.12 -- Overloading the assignment operator (https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/)[7].

Quiz time

Question #1

In the lesson above, we noted that the parameter for a copy constructor must be a (const) reference. Why aren't we allowed to use pass by value?

When we pass a class type argument by value, the copy constructor is implicitly invoked to copy the argument into the parameter.

If the copy constructor used pass by value, the copy constructor would need to call itself to copy the initializer argument into the copy constructor parameter. But that call to the copy constructor would also be pass by value, so the copy constructor would be invoked again to copy the argument into the function parameter. This would lead to an infinite chain of calls to the copy constructor.

Show Hint (javascript:void(0))[8]

Show Solution (javascript:void(0))[8]

Next lesson
14.15 Class initialization and copy elision

2

Back to table of contents

9

Previous lesson
14.13 Temporary class objects

10

11

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/[13] are connected to your provided email address.

230 COMMENTS Newest

jorge98
May 24, 2025 3:29 pm PDT

finished this lesson after 4 months! i hate my adhd and school

1 Reply

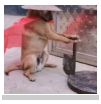
Nidhi Gupta
April 15, 2025 9:43 pm PDT

There is a copy constructor which works by as seen in example below:

Fraction(const Fraction& fraction) = delete;

void print() const
{
std::cout << "Fraction(" << m_numerator << ", " << m_denominator << ")\n";
}

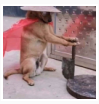
0 Reply



LechugaPlayer
April 12, 2025 6:09 am PDT

'If you compile and execute the above example, you may find that only two calls to the copy constructor occur'. Mine only called once ._.

2 Reply



LechugaPlayer
Reply to LechugaPlayer [14] April 12, 2025 6:17 am PDT

nvm, answer is in the next chapter

1 Reply

LapTQ
March 22, 2025 8:16 pm PDT

In the previous lesson, you wrote that "default constructor" is the constructor that has no argument. But in this lesson, you commented the following constructor, which has arguments, as "default constructor":

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

Can you clarify? Thanks!

0 Reply

Badger Patcher
Reply to LapTQ [15] March 30, 2025 9:28 am PDT

Here "default constructor" refers to the default values that the author has specified in the function parameter (int numerator = 0, int denominator = 1) so that if we do something like Fraction f {}; it defaults the numerator to 0 and denominator to 1, that is, f.numerator = 0 and f.denominator = 1. Hope it makes sense.

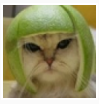
0 Reply

Manas Ghosh

December 14, 2024 3:24 am PST

`a private copy constructor can still be called from other members of the class`. but a constructor can't be called from other function including constructors. Then how is this possible?

0 Reply



Alex Author

Reply to Manas Ghosh [16] December 23, 2024 9:38 pm PST

I changed the word "called" to "used". If a member function creates an object of the class, the private constructor is a viable candidate for a match.

1 Reply

Amitrisu

December 3, 2024 3:03 am PST

Yeah, on my Clang 18.1.8 compiler I also get only one invocation of the copy constructor (**Return by value and the copy constructor** part), like several people below

0 Reply

vivien

September 30, 2024 12:40 pm PDT

I know this question shouldn't come in this chapter but please consider why this prints yes
cout<< ((x < 2) ? "yes" : "no");
And this prints 1
cout<< (x < 2) ? "yes" : "no";

0 Reply

Alex Author

Reply to vivien [17] October 1, 2024 3:11 pm PDT

See https://www.learncpp.com/cpp-tutorial/the-conditional-operator

0 Reply

Steins;Pointer
September 19, 2024 10:02 am PDT

in section "Return by value and the copy constructor", last paragraph could be a bit modified, because like in the case of previous commenter, compiler can elide more than one copy constructor call.

Thank You for amazing tutorial.

Last edited 9 months ago by Steins;Pointer

1 Reply

Droy
July 15, 2024 8:52 pm PDT

Hi,

When I'm running the example of the 'Pass by value (and return by value) and the copy constructor' section I'm getting the following output instead of the output given below the example -

**Copy constructor called
Fraction(5, 3)
Copy constructor called
Fraction(1, 2)**

Can you please tell me why is it happening?

I'm running the code in VS 2022, C++ 20.

0 Reply

Alex Author
Reply to Droy [18] July 17, 2024 7:33 pm PDT

See the paragraph immediately following the output box.

0 Reply

Ali
June 9, 2024 12:57 pm PDT

what should a copy ctor do when my member variables are allocated on the heap (using dynamic allocation)?
1- The member variable should point to the same thing as the other object does.
2- I should first ALLOCATE a NEW member variable and then set its value to the value of the other object's value (so that this->var has distinct address than other.var, but value of the two objects are the same.)

0 Reply

Strain

Reply to Ali [19] June 15, 2024 2:15 pm PDT

Second option. But, as Alex said, covered deeper in future lessons.

0 Reply

Alex Author

Reply to Ali [19] June 9, 2024 9:30 pm PDT

Covered in lesson https://www.learncpp.com/cpp-tutorial/shallow-vs-deep-copying/

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/
3. https://www.learncpp.com/cpp-tutorial/shallow-vs-deep-copying/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/
5. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/
6. https://www.learncpp.com/cpp-tutorial/destructors/
7. https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/
8. javascript:void(0)
9. https://www.learncpp.com/
10. https://www.learncpp.com/cpp-tutorial/temporary-class-objects/
11. https://www.learncpp.com/introduction-to-the-copy-constructor/
12. https://www.learncpp.com/cpp-tutorial/overloading-typecasts/
13. https://gravatar.com/
14. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-609203
15. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-608722
16. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-605228
17. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-602530
18. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-599637
19. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/#comment-598185
20. https://g.ezoic.net/privacy/learncpp.com