14.14 — Introduction to the copy constructor

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:

```
#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.

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).

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 return 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. Similarly, when an object is returned back to the caller by value, the copy constructor is implicitly invoked to make the copy.

We see both happen in the following example:

```
#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 f{ 5, 3 };

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

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

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

```
    return 0;
}
```

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`.

When `generateFraction` returns a `Fraction` back to `main`, the copy constructor is implicitly called again. And when `f2` is passed to `printFraction`, the copy constructor is called a third time.

On the author's machine, this example prints:

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

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:

```
#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 called by the public). However, a private copy constructor *can* still be called from 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 and 19.3 -- Destructors, and copy assignment in lesson 21.12 -- Overloading the assignment operator.

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?

Show Hint

Show Solution