13.12 — The copy constructor

ALEX AUGUST 19, 2021

Recapping the types of initialization

Since we're going to talk a lot about initialization in the next few lessons, let's first recap the types of initialization that C++ supports: direct initialization, uniform initialization or copy initialization.

Here are examples of all of those, using 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;
}

```

We can do a direct initialization:

```
int x(5); // Direct initialize an integer
Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int)
constructor
```

In C++11, we can do a uniform initialization:

```
int x { 5 }; // Uniform initialization of an integer
Fraction fiveThirds {5, 3}; // Uniform initialization of a Fraction, calls Fraction(int, int)
constructor
```

And finally, we can do a copy initialization:

```
int x = 6; // Copy initialize an integer
Fraction six = Fraction(6); // Copy initialize a Fraction, will call Fraction(6, 1)
Fraction seven = 7; // Copy initialize a Fraction.  The compiler will try to find a way to convert 7 to
a Fraction, which will invoke the Fraction(7, 1) constructor.
```

With direct and uniform initialization, the object being created is directly initialized. However, copy initialization is a little more complicated. We'll explore copy initialization in more detail in the next lesson. But in order to do that effectively, we need to take a short detour.

The copy constructor

Now consider the following program:

```
#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;
}

int main()
{
 Fraction fiveThirds { 5, 3 }; // Direct initialize a Fraction, calls Fraction(int, int)
constructor
 Fraction fCopy { fiveThirds }; // Direct initialize -- with what constructor?
 std::cout << fCopy << '\n';
}
```

If you compile this program, you'll see that it compiles just fine, and produces the result:

```
5/3
```

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

The initialization of variable fiveThirds is just a standard direct initialization that calls the Fraction(int, int) constructor. No surprises there. But what about the next line? The initialization of variable fCopy is also clearly a direct initialization, and you know that constructor functions are used to initialize classes. So what constructor is this line calling?

The answer is that this line is calling Fraction's copy constructor. A copy constructor is a special type of constructor used to create a new object as a copy of an existing object. And much like a default constructor, if you do not provide a copy constructor for your classes, C++ will create a public copy constructor for you. Because the compiler does not know much about your class, by default, the created copy constructor utilizes a method of initialization called memberwise initialization. Memberwise initialization simply means that each member of the copy is initialized directly from the member of the class being copied. In the above example, fCopy.m_numerator would be initialized from fiveThirds.m_numerator, etc...

Just like we can explicitly define a default constructor, we can also explicitly define a copy constructor. The copy constructor looks just like you'd expect it to:

```
#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& fraction)
        : m_numerator{fraction.m_numerator}, m_denominator{fraction.m_denominator}
        // Note: We can access the members of parameter fraction directly, because we're inside the Fraction class
    {
        // no need to check for a denominator of 0 here since fraction 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);
};

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

int main()
{
 Fraction fiveThirds { 5, 3 }; // Direct initialize a Fraction, calls Fraction(int, int) constructor
 Fraction fCopy { fiveThirds }; // Direct initialize -- with Fraction copy constructor
 std::cout << fCopy << '\n';
}
```

When this program is run, you get:

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

The copy constructor we defined in the example above uses memberwise initialization, and is functionally equivalent to the one we'd get by default, except we've added an output statement to prove the copy constructor is being called.

Unlike with default constructors, it's fine to use the default copy constructor if it meets your needs.

One interesting note: You've already seen a few examples of overloaded operator<<, where we're able to access the private members of parameter f1 because the function is a friend of the Fraction class. Similarly, member functions of a class can access the private members of parameters of the same class type. Since our Fraction copy constructor takes a parameter of the class type (to make a copy of), we're able to access the members of parameter fraction directly, even though it's not the implicit object.

Preventing copies

We can prevent copies of our classes from being made by making the copy constructor private:

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

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

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

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;
}

int main()
{
 Fraction fiveThirds { 5, 3 }; // Direct initialize a Fraction, calls Fraction(int, int) constructor
 Fraction fCopy { fiveThirds }; // Copy constructor is private, compile error on this line
 std::cout << fCopy << '\n';
}
```

Now when we try to compile our program, we'll get a compile error since fCopy needs to use the copy constructor, but can not see it since the copy constructor has been declared as private.

The copy constructor may be elided

Now consider the following example:

```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 &fraction)
  : m_numerator{fraction.m_numerator}, m_denominator{fraction.m_denominator}
 {
  // no need to check for a denominator of 0 here since fraction 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);
};

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

int main()
{
 Fraction fiveThirds { Fraction { 5, 3 } };
 std::cout << fiveThirds;
 return 0;
}
```

Consider how this program works. First, we direct initialize an anonymous Fraction object, using the Fraction(int, int) constructor. Then we use that anonymous Fraction object as an initializer for Fraction fiveThirds. Since the anonymous object is a Fraction, as is fiveThirds, this should call the copy constructor, right?

Run this and compile it for yourself. You'd probably expect to get this result (and you may):

```
copy constructor called
5/3
```

But in actuality, you're more likely to get this result:

```
5/3
```

Why didn't our copy constructor get called?

Note that initializing an anonymous object and then using that object to direct initialize our defined object takes two steps (one to

create the anonymous object, one to call the copy constructor). However, the end result is essentially identical to just doing a direct initialization, which only takes one step.

For this reason, in such cases, the compiler is allowed to opt out of calling the copy constructor and just do a direct initialization instead. This process is called elision.

So although you wrote:

```
Fraction fiveThirds { Fraction{ 5, 3 }
};
```

The compiler may change this to:

```
Fraction fiveThirds{ 5, 3
};
```

which only requires one constructor call (to Fraction(int, int)). Note that in cases where elision is used, any statements in the body of the copy constructor are not executed, even if they would have produced side effects (like printing to the screen)!

Prior to C++17, copy elision is an optimization the compiler can make. As of C++17, some cases of copy elision (including the example above) have been made mandatory.

Finally, note that if you make the copy constructor private, any initialization that would use the copy constructor will cause a compile error, even if the copy constructor is elided!

Next lesson
13.13 Copy initialization

Back to table of contents

Previous lesson
13.11 Overloading typecasts

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest