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:

```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 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