9.12 — Copy initialization

BY ALEX ON JUNE 5TH, 2016 | LAST MODIFIED BY ALEX ON JULY 22ND, 2016

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, it's better to avoid copy initialization for classes, and use direct or 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:

```
#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 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

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

22 comments to 9.12 — Copy initialization

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()
{