22.2 — R-value references

ALEX JANUARY 8, 2024

In chapter 9, we introduced the concept of value categories (12.2 -- Value categories (lvalues and rvalues) (https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/)), which is a property of expressions that helps determine whether an expression resolves to a value, function, or object. We also introduced l-values and r-values so that we could discuss l-value references.

If you're hazy on l-values and r-values, now would be a good time to refresh on that topic since we'll be talking a lot about them in this chapter.

L-value references recap

Prior to C++11, only one type of reference existed in C++, and so it was just called a "reference". However, in C++11, it's called an l-value reference. L-value references can only be initialized with modifiable l-values.

L-value reference	Can be initialized with	Can modify
Modifiable l-values	Yes	Yes
Non-modifiable l-values	No	No
R-values	No	No

L-value references to const objects can be initialized with modifiable and non-modifiable l-values and r-values alike. However, those values can't be modified.

L-value reference to const	Can be initialized with	Can modify
Modifiable l-values	Yes	No
Non-modifiable l-values	Yes	No
R-values	Yes	No

L-value references to const objects are particularly useful because they allow us to pass any type of argument (l-value or r-value) into a function without making a copy of the argument.

R-value references

C++11 adds a new type of reference called an r-value reference. An r-value reference is a reference that is designed to be initialized with an r-value (only). While an l-value reference is created using a single ampersand, an r-value reference is created using a double ampersand:

```
int x{ 5 };
int& lref{ x }; // l-value reference initialized with l-value x
int&& rref{ 5 }; // r-value reference initialized with r-value 5
```

R-values references cannot be initialized with l-values.

R-value reference	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	Yes

R-value reference to const	Can be initialized with	Can modify
Modifiable l-values	No	No
Non-modifiable l-values	No	No
R-values	Yes	No

R-value references have two properties that are useful. First, r-value references extend the lifespan of the object they are initialized with to the lifespan of the r-value reference (l-value references to const objects can do this too). Second, non-const r-value references allow you to modify the r-value!

Let's take a look at some examples:

```
#include <iostream>

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

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

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

int main()
{
    auto&& rref{ Fraction{ 3, 5 } }; // r-value reference to temporary Fraction

    // f1 of operator<< binds to the temporary, no copies are created.
    std::cout << rref << '\n';

    return 0;
} // rref (and the temporary Fraction) goes out of scope here
```

This program prints:

```
3/5
```

As an anonymous object, Fraction(3, 5) would normally go out of scope at the end of the expression in which it is defined. However, since we're initializing an r-value reference with it, its duration is extended until the end of the block. We can then use that r-value reference to print the Fraction's value.

Now let's take a look at a less intuitive example:

```
#include <iostream>

int main()
{
    int&& rref{ 5 }; // because we're initializing an r-value reference with a literal, a temporary with value 5 is created
here
    rref = 10;
    std::cout << rref << '\n';

    return 0;
}
```

This program prints:

```
10
```

While it may seem weird to initialize an r-value reference with a literal value and then be able to change that value, when initializing an r-value reference with a literal, a temporary object is constructed from the literal so that the reference is referencing a temporary object, not a literal value.

R-value references are not very often used in either of the manners illustrated above.

R-value references as function parameters

R-value references are more often used as function parameters. This is most useful for function overloads when you want to have different behavior for l-value and r-value arguments.

```
#include <iostream>

void fun(const int& lref) // l-value arguments will select this function
{
    std::cout << "l-value reference to const: " << lref << '\n';
}

void fun(int&& rref) // r-value arguments will select this function
{
    std::cout << "r-value reference: " << rref << '\n';
}

int main()
{
    int x{ 5 };
    fun(x); // l-value argument calls l-value version of function
    fun(5); // r-value argument calls r-value version of function

    return 0;
}
```

This prints:

```
l-value reference to const: 5
r-value reference: 5
```

As you can see, when passed an l-value, the overloaded function resolved to the version with the l-value reference. When passed an r-value, the overloaded function resolved to the version with the r-value reference (this is considered a better match than a l-value reference to const).

Why would you ever want to do this? We'll discuss this in more detail in the next lesson. Needless to say, it's an important part of move semantics.

Rvalue reference variables are lvalues

Consider the following snippet:

```
	int&& ref{ 5 };
	fun(ref);
```

Which version of `fun` would you expect the above to call: `fun(const int&)` or `fun(int&&)`?

The answer might surprise you. This calls `fun(const int&)`.

Although variable `ref` has type `int&&`, when used in an expression it is an lvalue (as are all named variables). The type of an object and its value category are independent.

You already know that literal `6` is an rvalue of type `int`, and `int x` is an lvalue of type `int`. Similarly, `int&& ref` is an lvalue of type `int&&`.

So not only does `fun(ref)` call `fun(const int&)`, it does not even match `fun(int&&)`, as rvalue references can't bind to lvalues.

Returning an r-value reference

You should almost never return an r-value reference, for the same reason you should almost never return an l-value reference. In most cases, you'll end up returning a hanging reference when the referenced object goes out of scope at the end of the function.

Quiz time

Question #1

State which of the following lettered statements will not compile:

```
int main()
{
    int x{};

    // l-value references
    int& ref1{ x }; // A
    int& ref2{ 5 }; // B

    const int& ref3{ x }; // C
    const int& ref4{ 5 }; // D

    // r-value references
    int&& ref5{ x }; // E
    int&& ref6{ 5 }; // F

    const int&& ref7{ x }; // G
    const int&& ref8{ 5 }; // H

    return 0;
}
```

Show Solution (javascript:void(0))

Next lesson
22.3 Move constructors and move assignment

Back to table of contents

Previous lesson
22.1 Introduction to smart pointers and move semantics