M.2 — R-value references

ALEX OCTOBER 30, 2021

Way back in chapter 1, we mentioned l-values and r-values, and then told you not to worry that much about them. That was fair advice prior to C++11. But understanding move semantics in C++11 requires a re-examination of the topic. So let's do that now.

L-values and r-values

Despite having the word "value" in their names, l-values and r-values are actually not properties of values, but rather, properties of expressions.

Every expression in C++ has two properties: a type (which is used for type checking), and a **value category** (which is used for certain kinds of syntax checking, such as whether the result of the expression can be assigned to). In C++03 and earlier, l-values and r-values were the only two value categories available.

The actual definition of which expressions are l-values and which are r-values is surprisingly complicated, so we'll take a simplified view of the subject that will largely suffice for our purposes.

It's simplest to think of an **l-value** (also called a locator value) as a function or an object (or an expression that evaluates to a function or object). All l-values have assigned memory addresses.

When l-values were originally defined, they were defined as "values that are suitable to be on the left-hand side of an assignment expression". However, later, the const keyword was added to the language, and l-values were split into two sub-categories: modifiable l-values, which can be changed, and non-modifiable l-values, which are const.

It's simplest to think of an **r-value** as "everything that is not an l-value". This notably includes literals (e.g. 5), temporary values (e.g. x+1), and anonymous objects (e.g. Fraction(5, 2)). r-values are typically evaluated for their values, have expression scope (they die at the end of the expression they are in), and cannot be assigned to. This non-assignment rule makes sense, because assigning a value applies a side-effect to the object. Since r-values have expression scope, if we were to assign a value to an r-value, then the r-value would either go out of scope before we had a chance to use the assigned value in the next expression (which makes the assignment useless) or we'd have to use a variable with a side effect applied more than once in an expression (which by now you should know causes undefined behavior!).

In order to support move semantics, C++11 introduces 3 new value categories: pr-values, x-values, and gl-values. We will largely ignore these since understanding them isn't necessary to learn about or use move semantics effectively. If you're interested, cppreference.com has an extensive list of expressions that qualify for each of the various value categories, as well as more detail about them.

L-value references

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 sometimes 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 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;
    int m_denominator;

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.

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

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

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.

One interesting note:

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

actually calls the l-value version of the function! Although variable ref has type *r-value reference to an integer*, it is actually an l-value itself (as are all named variables). The confusion stems from the use of the term r-value in two different contexts. Think of it this way: Named-objects are l-values. Anonymous objects are r-values. The type of the named object or anonymous object is independent from whether it's an l-value or r-value. Or, put another way, if r-value reference had been called anything else, this confusion wouldn't exist.

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

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;
}
```

Hide Solution

B, E, and G won't compile.

Next lesson
M.3 Move constructors and move assignment

Back to table of contents

Previous lesson
M.1 Intro to smart pointers and move semantics

65 COMMENTS Newest

Mohammed S
January 29, 2022 3:46 pm

E in the exercise will not compile as well, I tried it!

0 Reply

mursu
October 27, 2021 1:34 pm

>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.

0 Reply

Tanisha
October 13, 2021 3:47 am

I want to know the difference between these:
what happens in case 2?

suppose there is function defined as it takes rvalue ref
void fun(String && str);

and calling it via 2 different method, here you can not use str1 once it is moved:

1. String str1 = "hello";

fun(std::move(str1)); // here you can not use str1 as it would be null

2. but here str1 would still be used. why?? and

String str1 = "hello";
fun(std::string(str1));

0 Reply

Alex Author
Reply to Tanisha October 14, 2021 12:15 pm

In case 1, std::move() causes str1 to be converted into an r-value reference. Because std::string has a move constructor, the contents of str1 will then be moved (transferred) into function parameter str, which leaves str1 blank.
In case 2, we're creating a temporary copy of str1 using the std::string copy constructor, and then this r-value is move constructed into function parameter str. Because we've moved the copy rather than str1, str1 is not impacted.

1 Reply

solomon
September 29, 2021 1:16 am

This is for additional reading

https://www.internalpointers.com/post/understanding-meaning-lvalues-and-rvalues-c

0 Reply

New here
July 23, 2021 4:12 am

Hi,

Functions do type checking or value category checking when receiving arguments (or both)?

Because from my understanding the following argument makes sense only when functions do value category checking (or at least give it higher priority than type checking) when matching arguments passed to them!

"
One interesting note:

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

actually calls the l-value version of the function! Although variable ref has type r-value reference to an integer, it is actually an l-value itself (as are all named variables). The confusion stems from the use of the term r-value in two different contexts. Think of it this way: Named-objects are l-values. Anonymous objects are r-values. The type of the named object or anonymous object is independent from whether it's an l-value or r-value. Or, put another way, if r-value reference had been called anything else, this confusion wouldn't exist.
"

0 Reply

Quentin
February 27, 2021 4:48 am

Hi Alex,

" An r-value reference is a reference that is designed to be initialized with an r-value (only)."

But for code

```
int x{11};
auto &&rref{x};
std::cout << rref << '\n';
```

This works fine and rvalue reference binds to lvalue of x. I am confused by the result.

0 Reply

nascardriver Sub-admin
Reply to Quentin March 4, 2021 7:59 am

&&, when used with templates or `auto` can collapse. Your `auto` turns into `int&`, so you're forming an r-value reference to an `int`-reference, which collapses to an `int`-reference.

```
#include

// @decltype() "returns" the type of @rref
// @std::is_same_v checks if 2 types are the same
std::cout << std::is_same_v << '\n';
```

See https://en.cppreference.com/w/cpp/language/reference section "Reference collapsing"

0 Reply

yolo
Reply to nascardriver March 23, 2021 12:18 pm

I don't really get why collapsing forms a problem. What kind of problems does it create? The link doesn't say anything in particular.

0 Reply

nascardriver Sub-admin
Reply to yolo March 24, 2021 8:06 am

Why do you think collapsing causes problems?

0 Reply

goiu
January 23, 2021 6:13 am

Hi there!
What is the difference between an anonymous object and a temporary value?

0 Reply

yolo
Reply to goiu March 23, 2021 12:27 pm

Take it with a grain of salt. Anonymous objects can be temporary values but temporary values cannot be anonymous objects. Anonymous objects can be

```
Class{5} //anonymous object. Gets destroyed immediately
Class class{5} // instead of this normal variable. Follows scope.
```

0 Reply

yeokaiwei
January 7, 2021 10:20 pm

1. Feedback on "R-value references as function parameters" example

Based on the tutorial settings in Chapter 1, there will be warnings for the void fun() examples, while using MSVS2019.

"C4100 'lref', 'rref' unreferenced formal parameters"

0 Reply

yeokaiwei
January 7, 2021 10:09 pm

```
#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;
}
```

```
#include <iostream>

int main()
{
    int rref{ 5 }; // I just removed the &&
    rref = 10;
    std::cout << rref << '\n';

    return 0;
}
```

In the less intuitive example, what's the difference between the above code and the latter code?

Both print 10.

0 Reply

Hovsep_Papoyan
December 1, 2020 3:55 am

#include <iostream>

// example 1
/*
struct s {};

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

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

const s f()
{
return s{};
}

int main()
{
const s obj;
fun(std::move(obj)); // l-value reference
fun(f()); // l-value reference
return 0;
}
*/

// example 2

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

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

const int f()
{
return 0;
}

int main()
{
const int obj{};
fun(std::move(obj)); // l-value reference to const
fun(f()); // r-value reference
return 0;
}

Pointless but nevertheless legal cases:

1. function that returns a const value, like f() above;
2. call std::move() on a const object, like above.

0 Reply