22.2 — R-value references

ALEX[1] MAY 15, 2024

In chapter 12, 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/)[2]), 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:

```cpp
#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:

```cpp
#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.

```cpp
#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 an 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:

```cpp
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 `5` 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:

```cpp
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))[3]

Next lesson
22.3 Move constructors and move assignment

4

Back to table of contents

5

Previous lesson
22.1 Introduction to smart pointers and move semantics

6

7

Avatars from https://gravatar.com/[8] are connected to your provided email address.

132 COMMENTS Newest

Hari
May 26, 2025 11:31 pm PDT

```
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 5 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.
```

This is so mind-breakingly hilarious xD

0 Reply

Lith
January 17, 2025 6:54 pm PST

In an earlier lesson about lvalue and rvalue, there's a part about lvalue to rvalue conversion, which is lvalue will be converted to an rvalue wherever its needed. My question is why isnt that the case here. For example, (E) and (G) from the quiz above. Is there any other exception like this

0 Reply

Sergei
January 17, 2025 2:28 am PST

The goal is simple: to be able to pass big r-values by reference w/o copying. But the solution finally is too complicated as I see... the construction

```
const smthVeryComplcatedAndHuge a;
...
int function s(const smthVeryComplcatedAndHuge* const d);
...
r = function(&a);
```

looks simpler.

Last edited 5 months ago by Sergei

0 Reply

EmtyC

December 29, 2024 11:37 am PST

Well after tinkering, I have been blessed (or cursed) with this discovery:

Code 1:

```cpp
#include <iostream>

int main()
{
    const char (&& lel)[6] { static_cast<const char(&&)[6]>("hello") };

    std::cout << lel << '\n';

    char(&& bel)[6]{const_cast<char(&&)[6]>(lel)};

    std::cout << bel << '\n';

    bel[1] = 'a';

    std::cout << bel << '\n';
}
```

Output:

```
hello
@=´Â÷
@a´Â÷
```

(program closed normally)

Code 2:

```cpp
#include <iostream>

int main()
{
    const char (& lel)[6] { "hello" };

    std::cout << lel << '\n';

    char(& bel)[6]{const_cast<char(&)[6]>(lel)};

    std::cout << bel << '\n';

    bel[1] = 'a';

    std::cout << bel << '\n';
}
```

Output:

```
hello
hello

C:\Users\HP\source\repos\main\x64\Debug\main.exe (process 17508) exited with code
-1073741819 (0xc0000005).
Press any key to close this window . . .
```

(access violation error code)

What in the C++ is happening, where to even start :D (I believe you Alex, const_cast is unsafe)
my main question is, why is the first a success (with mess up data), and the second is a failure (but printed a correct data for the second output)?

Sorry for bothering you a lot lately Alex :>

0 Reply

Alex Author
Reply to EmtyC 9 January 4, 2025 3:23 pm PST

Honestly, I'm not sure. It seems like both should print hello hello (undefined). Binding a const rvalue reference to an const lvalue should be okay. Casting away const should also be okay so long as you don't write to the object. It's the assignment to the non-const object that should be invoking undefined behavior.

I'm not sure if you're running into a compiler bug or there is something I don't understand about how references, const_cast, and C-style arrays all intermingle.

1 Reply

EmtyC
Reply to Alex 10 January 18, 2025 8:12 am PST

After some research, and my own testing, it seems your first guess is right (compiler bug, MSVC... :>), both clang and gcc print hello hello (mostly crash for both cases) and based on https://en.cppreference.com/w/cpp/language/string_literal
we have: String literals are not convertible or assignable to non-const CharT*. An explicit cast (e.g. const_cast) must be used if such conversion is wanted.
And
The effect of attempting to modify a string literal object is undefined.

It's safe to deduce that const_casting a string literal is safe, although they specified pointers here and not r-value references, the fact that gcc and clang have no issue is telling :)

Edit: as far as I read in value categories, an x-value (result of const_cast<someType&&>) binds normally to r-value reference and per https://en.cppreference.com/w/cpp/language/const_cast in section Casting away constness, If a cast from a prvalue of type T1* to the type T2* casts away constness, casting from an expression of type T1 to a reference to T2 will also cast away constness. and the note below:

const_cast makes it possible to form a reference or pointer to non-const type that is actually referring to a const object or a reference or pointer to non-volatile type that is actually referring to a volatile object. Modifying a const object through a non-const access path and referring to a volatile object through a non-volatile glvalue results in undefined behavior.

We can deduce from this that what MSVC does cannot go under any undefined behavior and it's a compiler bug :>

Last edited 5 months ago by EmtyC

0 Reply

EmtyC

Reply to Alex [10] January 4, 2025 3:54 pm PST

Thanks for responding, I will add this to my `things_to_research` buffer :>, If I succeed, I will report back my findings

0 Reply

EmtyC

December 29, 2024 11:12 am PST

I was tinkering with r-value reference and then wrote this

```
#include <iostream>

int main()
{
    [[maybe_unused]] const char (&& lel)[6] { static_cast<const char(&&)[6]>("sdbck") }; // unrelated

     // [[maybe_unused]] int& c{}; // compiler error

    [[maybe_unused]] int&& c{}; // HUH !?

    char(&& bel)[6]{}; // Works without an initializer

        double&& g; // doesn't work ??!!

    std::cout << bel << '\n';

}
```

Look it up in the web, didn't find anything

0 Reply

EmtyC

Reply to EmtyC [11] December 29, 2024 11:21 am PST

Absolute wisdom from chatgpt:

```
3. [[maybe_unused]] int&& c{};
This compiles because:

An rvalue reference (int&&) can exist in an uninitialized state.
c is default-initialized but unbound, so its value is indeterminate.
This behavior is legal but dangerous if you attempt to dereference c.
```

Note: it seems I forgot what is `zero initialization`, for int&& c, c is bound to 0, for char(&& bel)[6]{}, bel is an r-value reference to r-value array {'\0','\0','\0'...} (\0 is the null terminator) This is just a deduction from my part, didn't find any resource talking about this, so the compiler construct a default r-value of the type and bind the reference to it?

Last edited 6 months ago by EmtyC

0 Reply

Alex Author

Reply to EmtyC 12 January 4, 2025 3:09 pm PST

`{}` does value initialization, will generally perform zero-initialization (but also default initialize class types).

I'm surprised your example compiles, because references must be initialized to an object. But apparently C++ treats this as a syntax for creating a reference to a lifetime-extended temporary. It works for const lvalue references too (but not non-const lvalue references).

It's covered here: https://en.cppreference.com/w/cpp/language/list_initialization, under explanation, second to last bullet ("Otherwise, if T is a reference type that is not compatible with the type of the initializer clause").

1 Reply

EmtyC

Reply to Alex 13 January 4, 2025 3:52 pm PST

thank you as always for your response :> (I find it quite hard to understand cppreference in the details so your response are a real help)

0 Reply

LVNA

August 13, 2024 5:42 pm PDT

I am just curious, what is exactly the rationale behind some of the compiler warnings? There is warning when you return a variable by reference like this:

```
std::string& hanging_return()
{
    std::string abc {};
    std::cout << "> ";
    std::cin >> abc;
    return abc;
}
```

But not warning if you create a reference beforehand. I know you probably can't speak for the gcc compiler developers, but it seems to be really inconsistent. This provides no warnings:

```
std::string& hanging_return()
{
    std::string abc {};
    std::cout << "> ";
    std::cin >> abc;
    std::string& a_ref {abc};
    return a_ref;
}
```

0 Reply

Alex Author

Reply to LVNA [14] August 17, 2024 3:21 pm PDT

I get a warning from GCC when I compile that function (on https://wandbox.org/#). Try updating your GCC.

0 Reply

Strain

May 11, 2024 4:40 am PDT

That `auto&&` is something unique, may I say: if initialised with an l-value, it is an l-value reference (maybe `const` or not), and if initialised with an r-value, it is an r-value reference.

1 Reply

Asicx

Reply to Strain [15] September 19, 2024 2:12 pm PDT

Yeah i had an argument with chat gpt about it and i think it does bind to anything (r/lvalues/const..). auto&& is called a universal (forwarding) reference.

0 Reply

yagni

Reply to Strain [15] September 19, 2024 3:49 am PDT

As stated in the article, r-value&& cannot point to l-value objects

0 Reply

rafal

March 11, 2024 12:53 pm PDT

Hello

I excepted it to not compile at all or, to create temporary object in this expression.
`int&& rValueRef2 {static_cast<int&&>(lValue)};`
However none of those happened. Instead of doing any of above it does run (Tested in c++20).
I'am lost so I commented the code with what I think is happening and added arrows for easier location and reference.

```
/*
"We cannot bind an rvalue reference to an l-value (E & G)". But.. somehow it runs
*/

#include <iostream>

void fun(int&& rref){
    std::cout << "rvalue reference: " << rref << '\n';
}

int main()
{
    int lValue{3};
    int& lValueRef {lValue};

    //int&& rValueRef {lValue}; // error: cannot bind rvalue reference of type
                                // ‘int&&’ to lvalue of type ‘int’

    int&& rValueRef2 {static_cast<int&&>(lValue)};
        // rvlaue reference to ??????
        // rvalue reference to lvalue of type int&& ? What happened here?       <---
QUESTION A

    std::cout << '\n' << rValueRef2 << '\n'; // prints 3

    lValue = 5;

    std::cout << '\n' << rValueRef2 << '\n'; // prints 5
                // rvalue reference is bound to lValue variable?                <---
QUESTION B

    lValueRef = 7;
    std::cout << '\n' << rValueRef2 << '\n'; // prints 7
                // rValueRef2 references lvalue variable lValue?                <---
QUESTION C


    rValueRef2 = 1;
    // rvalue refernece modifies lvalue ?                                       <---
QUESTION D
    std::cout << '\n' << rValueRef2 << '\t' << lValue << '\n'; // prints 1 1


    fun(static_cast<int&&>(rValueRef2)); // prints "rvlaue reference: 1"
                // This now works too . Why?                                    <---
QUESTION E


    return 0;
}
```

0 Reply

rafal

Reply to rafal 16 March 12, 2024 12:12 pm PDT

I found similar question 2 lessons forward.

https://www.learncpp.com/cpp-tutorial/stdmove/#comment-584180

Based on that I think that this should be correct:

So `static_cast<int&&>(lValue)`, from question A, will create temporary rvalue reference that will be then used to initialize `rValueRef2` ?

B)
Since temporary reference was referring to lvalue `lValue`
rValueRef2 is rvalue reference to `lValue` ?

C)
Just makes sense if A and B are right.

D)
Provided C makes sense.

E)
We create temporary object that is rvalue so function accepts it.

0 Reply

Alex Author
Reply to rafal 17 March 12, 2024 6:10 pm PDT

Yep.

0 Reply

Dongbin
March 6, 2024 3:33 am PST

Regarding dangling reference from returning a r-value reference from a function, why doesn't lifespan extension apply?

0 Reply

Alex Author
Reply to Dongbin 18 March 8, 2024 9:42 pm PST

Lifetime extension doesn't work across function boundaries.

1 Reply

Rohit
February 3, 2024 6:18 am PST

C-style string literals are lvalues. Then why do they get bound to rvalue reference?

For e.g. following code works fine.

#include <iostream>

void fun(const char* &&str){
std::cout<<str;
}

int main()
{

fun("hello");

return 0;
}

Last edited 1 year ago by Rohit

0 Reply

Alex Author

Reply to Rohit 19 February 3, 2024 12:21 pm PST

The type of "hello" is actually `const char[6]`. When used as an argument, a C-style string decays into a temporary pointer holding the address of the first element of the array. This temporary pointer is an rvalue, and thus will bind to an rvalue reference parameter.

Similarly:

```cpp
#include <iostream>

void go(const int* && x)
{
    std::cout << *x;
}

int main()
{
    const int x { 5 };
    go(&x); // &x is a temporary pointer rvalue

    return 0;
}
```

4 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/
7. https://www.learncpp.com/rvalue-references/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-605898
10. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-606235

11. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-605895
12. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-605897
13. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-606234
14. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-600857
15. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-596947
16. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-594550
17. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-594590
18. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-594351
19. https://www.learncpp.com/cpp-tutorial/rvalue-references/#comment-593187
20. https://g.ezoic.net/privacy/learncpp.com