12.4 — Lvalue references to const

ALEX[1] FEBRUARY 23, 2025

In the previous lesson (12.3 -- Lvalue references (https://www.learncpp.com/cpp-tutorial/lvalue-references/)[2]), we discussed how an lvalue reference can only bind to a modifiable lvalue. This means the following is illegal:

```cpp
int main()
{
    const int x { 5 }; // x is a non-modifiable (const) lvalue
    int& ref { x }; // error: ref can not bind to non-modifiable lvalue

    return 0;
}
```

This is disallowed because it would allow us to modify a const variable (`x`) through the non-const reference (`ref`).

But what if we want to have a const variable we want to create a reference to? A normal lvalue reference (to a non-const value) won't do.

Lvalue reference to const

By using the `const` keyword when declaring an lvalue reference, we tell an lvalue reference to treat the object it is referencing as const. Such a reference is called an **lvalue reference to a const value** (sometimes called a **reference to const** or a **const reference**).

Lvalue references to const can bind to non-modifiable lvalues:

```cpp
int main()
{
    const int x { 5 };    // x is a non-modifiable lvalue
    const int& ref { x }; // okay: ref is a an lvalue reference to a const value

    return 0;
}
```

Because lvalue references to const treat the object they are referencing as const, they can be used to access but not modify the value being referenced:

```
#include <iostream>

int main()
{
    const int x { 5 };    // x is a non-modifiable lvalue
    const int& ref { x }; // okay: ref is a an lvalue reference to a const value

    std::cout << ref << '\n'; // okay: we can access the const object
    ref = 6;                  // error: we can not modify an object through a const
reference

    return 0;
}
```

Initializing an lvalue reference to const with a modifiable lvalue

Lvalue references to const can also bind to modifiable lvalues. In such a case, the object being referenced is treated as const when accessed through the reference (even though the underlying object is non-const):

```
#include <iostream>

int main()
{
    int x { 5 };          // x is a modifiable lvalue
    const int& ref { x }; // okay: we can bind a const reference to a modifiable
lvalue

    std::cout << ref << '\n'; // okay: we can access the object through our const
reference
    ref = 7;                  // error: we can not modify an object through a const
reference

    x = 6;                // okay: x is a modifiable lvalue, we can still modify it
through the original identifier

    return 0;
}
```

In the above program, we bind const reference `ref` to modifiable lvalue `x`. We can then use `ref` to access `x`, but because `ref` is const, we can not modify the value of `x` through `ref`. However, we still can modify the value of `x` directly (using the identifier `x`).

Best practice

Favor `lvalue references to const` over `lvalue references to non-const` unless you need to modify the object being referenced.

Initializing an lvalue reference to const with an rvalue

Perhaps surprisingly, lvalues references to const can also bind to rvalues:

```cpp
#include <iostream>

int main()
{
    const int& ref { 5 }; // okay: 5 is an rvalue

    std::cout << ref << '\n'; // prints 5

    return 0;
}
```

if const was not there, then this is an error

When this happens, a temporary object is created and initialized with the rvalue, and the reference to const is bound to that temporary object.

Related content

We covered temporary objects in lesson 2.5 -- Introduction to local scope (https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/)[3].

Initializing an lvalue reference to const with a value of a different type

Lvalue references to const can even bind to values of a different type, so long as those values can be implicitly converted to the reference type:

```cpp
#include <iostream>

int main()
{
    // case 1
    const double& r1 { 5 };  // temporary double initialized with value 5, r1 binds to temporary

    std::cout << r1 << '\n'; // prints 5

    // case 2
    char c { 'a' };
    const int& r2 { c };     // temporary int initialized with value 'a', r2 binds to temporary

    std::cout << r2 << '\n'; // prints 97 (since r2 is a reference to int)

    return 0;
}
```

In case 1, a temporary object of type `double` is created and initialized with int value `5`. Then `const double& r1` is bound to that temporary double object.

In case 2, a temporary object of type `int` is created and initialized with char value `a`. Then `const int& r2` is bound to that temporary int object.

In both cases, the type of the reference and the type of the temporary match.

Key insight

If you try to bind a const lvalue reference to a value of a different type, the compiler will create a temporary object of the same type as the reference, initialize it using the value, and then bind the reference to the temporary.

Also note that when we print `r2` it prints as an int rather than a char. This is because `r2` is a reference to an int object (the temporary int that was created), not to char `c`.

Although it may seem strange to allow this, we'll see examples where this is useful in lesson 12.6 -- Pass by const lvalue reference (https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/)[4].

Warning

We normally assume that a reference is identical to the object it is bound to -- but this assumption is broken when a reference is bound to a temporary copy of the object or a temporary resulting from the conversion of the object instead. Any modifications subsequently made to the original object will not be seen by the reference (as it is referencing a different object), and vice-versa.

Here's a silly example showing this:

```cpp
#include <iostream>

int main()
{
    short bombs { 1 };          // I can has bomb! (note: type is short)

    const int& you { bombs };  // You can has bomb too (note: type is int&)
    --bombs;                   // Bomb all gone

    if (you)                   // You still has?
    {
        std::cout << "Bombs away!  Goodbye, cruel world.\n"; // Para bailar la
bomba
    }

    return 0;
}
```

In the above example, `bombs` is a `short` and `you` is a `const int&`. Because `you` can only bind to an `int` object, when `you` is initialized with `bombs`, the compiler will implicitly convert `bombs` to an `int`, which results in the creation of a temporary `int` object (with value `1`). `you` ends up bound to this temporary object rather than `bombs`.

When `bombs` is decremented, `you` is not affected because it is referencing a different object. So although we expect `if (you)` to evaluate to `false`, it actually evaluates to `true`.

If you would stop blowing up the world, that would be great.

Const references bound to temporary objects extend the lifetime of the temporary object

Temporary objects are normally destroyed at the end of the expression in which they are created.

Given the statement `const int& ref { 5 };`, consider what would happen instead if the temporary object created to hold rvalue `5` was destroyed at the end of the expression that initializes `ref`. Reference `ref` would be left dangling (referencing an object that had been destroyed), and we'd get undefined behavior when we tried to access `ref`.

To avoid dangling references in such cases, C++ has a special rule: When a const lvalue reference is *directly* bound to a temporary object, the lifetime of the temporary object is extended to match the lifetime of the reference.

```
#include <iostream>

int main()
{
    const int& ref { 5 }; // The temporary object holding value 5 has its lifetime extended to match ref

    std::cout << ref << '\n'; // Therefore, we can safely use it here

    return 0;
} // Both ref and the temporary object die here
```

In the above example, when `ref` is initialized with rvalue `5`, a temporary object is created and `ref` is bound to that temporary object. The lifetime of the temporary object matches the lifetime of `ref`. Thus, we can safely print the value of `ref` in the next statement. Then both `ref` and the temporary object go out of scope and are destroyed at the end of the block.

Key insight

Lvalue references can only bind to modifiable lvalues.

Lvalue references to const can bind to modifiable lvalues, non-modifiable lvalues, and rvalues. This makes them a much more flexible type of reference.

For advanced readers

Lifetime extension only works when a const reference is directly bound to a temporary. Temporaries returned from a function (even ones returned by const reference) are not eligible for lifetime extension.

We show an example of this in lesson 12.12 -- Return by reference and return by address (https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/)[5].

For class type rvalues, binding a reference to a member will extend the lifetime of the entire object.

So why does C++ allow a const reference to bind to an rvalue anyway? We'll answer that question in the next lesson!

Constexpr lvalue references Optional

When applied to a reference, `constexpr` allows the reference to be used in a constant expression. Constexpr references have a particular limitation: they can only be bound to objects with static duration (either globals or static locals). This is because the compiler knows where static objects will be instantiated in memory, so it can treat that address as a compile-time constant.

A constexpr reference cannot bind to a (non-static) local variable. This is because the address of local variables is not known until the function they are defined within is actually called.

```cpp
int g_x { 5 };

int main()
{
    [[maybe_unused]] constexpr int& ref1 { g_x }; // ok, can bind to global

    static int s_x { 6 };
    [[maybe_unused]] constexpr int& ref2 { s_x }; // ok, can bind to static local

    int x { 6 };
    [[maybe_unused]] constexpr int& ref3 { x }; // compile error: can't bind to non-static object

    return 0;
}
```

When defining a constexpr reference to a const variable, we need to apply both `constexpr` (which applies to the reference) and `const` (which applies to the type being referenced).

```cpp
int main()
{
    static const int s_x { 6 }; // a const int
    [[maybe_unused]] constexpr const int& ref2 { s_x }; // needs both constexpr and const

    return 0;
}
```

Given these limitations, constexpr references typically don't see much use.

Next lesson
12.5 Pass by lvalue reference

6

Back to table of contents

7

Previous lesson
12.3 Lvalue references

2

8

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[11] are connected to your provided email address.

205 COMMENTS

Newest

Trystan Henriques
June 28, 2025 1:42 pm PDT

Best c++ resource out

Last edited 3 days ago by Trystan Henriques

0 Reply

Diddy
June 21, 2025 9:17 pm PDT

this is the most incredible reading i have read

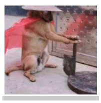
Last edited 10 days ago by Diddy

0 Reply

Rober
May 23, 2025 3:55 pm PDT

Damn constexpr, damn consteval, damn references, damn numeric conversions hahahahaja

0 Reply



LechugaPlayer
March 31, 2025 7:02 am PDT

I'll finish this tutorial with trauma related to constexpr and bombs

10 Reply

RSH

Reply to LechugaPlayer [12] May 11, 2025 10:38 pm PDT

Fr, constexpr has gave me trauma aswell

0 Reply

Kania

March 4, 2025 6:20 am PST

Alex gotta stop with these bombs examples

15 Reply

Leni

March 2, 2025 8:36 pm PST

This article provides a thorough explanation of lvalue references to const in C++, highlighting their flexibility and best practices. Why does C++ allow lvalue references to const to bind to rvalues, and how does this impact memory management?

0 Reply

Jack

December 15, 2024 6:47 am PST

When this code from this lesson is copied into Visual Studio it crashes, but when replace "int g_x { 5 };" with "int g_x = 5;" it works fine.

```
int g_x { 5 };

int main()
{
    [[maybe_unused]] constexpr int& ref1 { g_x }; // ok, can bind to global

    static int s_x { 6 };
    [[maybe_unused]] constexpr int& ref2 { s_x }; // ok, can bind to static local

    int x { 6 };
    [[maybe_unused]] constexpr int& ref3 { x }; // compile error: can't bind to non-static object

    return 0;
}
```

0 Reply

Alex Author

Reply to Jack [13] December 26, 2024 12:29 pm PST

It should behave identically. Try updating your compiler?

1 Reply

Jack
Reply to Alex [14] January 7, 2025 7:06 am PST

I reported this issue in the Visual Studio developer community and they fixed this bug on 17.13.0 Preview 2.0. I checked this on 17.13.0 Preview 2.1 and everything works fine.

10 Reply

Kyutae Lee
December 6, 2024 10:52 pm PST

Thank you for this great tutorials! It really helps!
But I have some difficulties what the following text means. Could you provide some example code to demonstrate the behavior?

> For class type rvalues, only the entire object can have its lifetime extended. Individual members can not have their lifetimes extended.

1 Reply

Alex Author
Reply to Kyutae Lee [15] December 9, 2024 11:52 pm PST

The way I wrote that sentence was misleading at best. I've rewritten it as "For class type rvalues, binding a reference to a member will extend the lifetime of the entire object."

Is that clearer?

3 Reply

Kyutae Lee
Reply to Alex [16] December 10, 2024 6:27 am PST

Now it is clear! Thank you!

0 Reply

NordicPeace
December 2, 2024 12:28 am PST

The best practices confuses my mind a lot.It is just like saying "Ohh! do X and don't do Y, Z sucks, Like- it is good for knowing but pls let me make mistakes and try myself.Sometimes it overwhelms my mind.

C++ is a great language- But I will never understand the meaning of best practices even after countless hours of reading and writing.

Like OOP "Everything is a class" every time a new thing comes that overtake the previous thing and make that practice obsolete; Same goes for languages.

0 Reply

Alex Author

Reply to NordicPeace [17] December 3, 2024 3:04 pm PST

Best practices are tried and true methods that work in most cases. You should follow them unless you have a specific reason not to. In many cases, not following them will initially appear to work as well, but you'll eventually run into whatever issue it was that caused the best practice to be a better option than the one you chose...

Why make the same mistakes that other people have already made? It'll slow your learning down significantly if you have to discover every principle yourself. Much better is to follow the best practices and ensure you understand WHY that thing is a best practice. And in doing so, you'll not only speed up your learning, you'll also build good habits along the way.

2 Reply

NordicPeace

Reply to Alex [18] December 4, 2024 10:33 am PST

I learned this the hard way. While working on my university assignment, I ended up submitting it with messy code and ignored naming conventions. As a result, I barely passed and received low grades. My project was full of spaghetti code and looked awful. I should have followed your advice and stuck to best practices.

Thanks for replying, by the way!

0 Reply

KLAP

October 20, 2024 1:29 pm PDT

I'm starting a `constexpr` hate club after the optional.

18 Reply

Yagy

Reply to KLAP [19] October 30, 2024 4:53 am PDT

I really would like to join your club, mate! :)

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/lvalue-references/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/
4. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/
5. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/
6. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/
7. https://www.learncpp.com/
8. https://www.learncpp.com/lvalue-references-to-const/
9. https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/
10. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-608930
13. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-605274
14. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-605749
15. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-604985
16. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-605077
17. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-604765
18. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-604830
19. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/#comment-603372
20. https://g.ezoic.net/privacy/learncpp.com