9.4 — Lvalue references to const

ALEX JANUARY 19, 2022

In the previous lesson (9.3 -- Lvalue references), 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:

```cpp
#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;     // okay: we can access the const object
    ref = 6;              // error: we can not modify a const object

    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):

```cpp
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;     // 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; // prints 5

    return 0;
}
```

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

A **temporary object** (also sometimes called an **anonymous object**) is an object that is created for temporary use (and then destroyed) within a single expression. Temporary objects have no scope at all (this makes sense, since scope is a property of an identifier, and temporary objects have no identifier). This means a temporary object can only be used directly at the point where it is created, since there is no way to refer to it beyond that point.

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.

However, consider what would happen in the above example 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 bound to a temporary object, the lifetime of the temporary object is extended to match the lifetime of the reference.

```cpp
#include <iostream>

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

    std::cout << ref; // 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.

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

Next lesson
9.5 Pass by lvalue reference

Back to table of contents

Previous lesson
9.3 Lvalue references