12.4 — Lvalue references to const

In the previous lesson (12.3 -- Lvalue references), we discussed how an lvalue reference can only bind to a modifiable lvalue. This means the following is illegal:

```
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:

```
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:

```
#include <iostream>

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

    std::cout << ref << '\n'; // 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.

Related content

We covered temporary objects in lesson 2.5 -- Introduction to local scope.

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:

```
#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.

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 *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.

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.

```
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).

```
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.