15.10 — Ref qualifiers

ALEX[1] SEPTEMBER 25, 2024

Author's note

This is an optional lesson. We recommend having a light read-through to familiarize yourself with the material, but comprehensive understanding is not required to proceed with future lessons.

In lesson 14.7 -- Member functions returning references to data members (https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/)[2], we discussed how calling access functions that return references to data members can be dangerous when the implicit object is an rvalue. Here's a quick recap:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Employee
{
private:
    std::string m_name{};

public:
    Employee(std::string_view name): m_name { name } {}
    const std::string& getName() const { return m_name; } //  getter returns by const reference
};

// createEmployee() returns an Employee by value (which means the returned value is an rvalue)
Employee createEmployee(std::string_view name)
{
    Employee e { name };
    return e;
}

int main()
{
    // Case 1: okay: use returned reference to member of rvalue class object in same expression
    std::cout << createEmployee("Frank").getName() << '\n';

    // Case 2: bad: save returned reference to member of rvalue class object for use later
    const std::string& ref { createEmployee("Garbo").getName() }; // reference becomes dangling when return value of createEmployee() is destroyed
    std::cout << ref << '\n'; // undefined behavior

    return 0;
}
```

In case 2, the rvalue object returned from `createEmployee("Garbo")` is destroyed after initializing `ref`, leaving `ref` referencing a data member that was just destroyed. Subsequent use of `ref` exhibits undefined behavior.

This presents somewhat of a conundrum.

- If our `getName()` function returns by value, this is safe when our implicit object is an rvalue, but makes an expensive and unnecessary copy when our implicit object is an lvalue (which is the most common case).
- If our `getName()` function returns by const reference, this is efficient (as no copy of the `std::string` is made), but can be misused when the implicit object is an rvalue (resulting in undefined behavior).

Since member functions are typically called on lvalue implicit objects, the conventional choice is to return by const reference and simply avoid misusing the returned reference in cases where the implicit object is an rvalue.

Ref qualifiers

The root of the challenge illustrated above is that we only want one function to service two different cases (one where our implicit object is an lvalue, and one where our implicit object is an rvalue). What's optimal for one case isn't ideal for the other case.

To help address such issues, C++11 introduced a little known feature called a **ref-qualifier** that allows us to overload a member function based on whether it is being called on an lvalue or an rvalue implicit object. Using this feature, we can create two versions of `getName()` -- one for the case where our implicit object is an lvalue, and one for the case where our implicit object is an rvalue.

First, let's start with our non-ref-qualified version of `getName()`

```
const std::string& getName() const { return m_name; } // callable with both lvalue and rvalue implicit objects
```

To ref-qualify this function, we add a `&` qualifier to the overload that will match only lvalue implicit objects, and a `&&` qualifier to the overload that will match only rvalue implicit objects:

```
const std::string& getName() const &  { return m_name; } //  & qualifier overloads function to match only lvalue implicit objects, returns by reference
std::string        getName() const && { return m_name; } // && qualifier overloads function to match only rvalue implicit objects, returns by value
```

Because these functions are distinct overloads, they can have different return types! Our lvalue-qualified overload returns by const reference, whereas our rvalue-qualified overload returns by value.

Here's a full-example of the above:

```cpp
#include <iostream>
#include <string>
#include <string_view>

class Employee
{
private:
    std::string m_name{};

public:
    Employee(std::string_view name): m_name { name } {}

    const std::string& getName() const &  { return m_name; } //  & qualifier overloads function to match only lvalue implicit objects
    std::string        getName() const && { return m_name; } // && qualifier overloads function to match only rvalue implicit objects
};

// createEmployee() returns an Employee by value (which means the returned value is an rvalue)
Employee createEmployee(std::string_view name)
{
    Employee e { name };
    return e;
}

int main()
{
    Employee joe { "Joe" };
    std::cout << joe.getName() << '\n'; // Joe is an lvalue, so this calls std::string& getName() & (returns a reference)

    std::cout << createEmployee("Frank").getName() << '\n'; // Frank is an rvalue, so this calls std::string getName() && (makes a copy)

    return 0;
}
```

This allows us to do the performant thing when our implicit object is an lvalue, and the safe thing when our implicit object is an rvalue.

For advanced readers

The above rvalue overload of `getName()` above is potentially suboptimal from a performance perspective when the implicit object is a non-const temporary. In such cases, the implicit object is going to die at the end of the expression anyway. So instead of having the rvalue getter return a (possibly expensive) copy of the member, we can have it try to move the member (using `std::move`).

This can be facilitated by adding the following overloaded getter for non-const rvalues:

```cpp
// If the implicit object is a non-const rvalue, use std::move to try to move m_name
std::string getName() && { return std::move(m_name); }
```

This can either coexist with the const rvalue getter, or you can just use this instead (since const rvalues are fairly uncommon).

We cover `std::move` in lesson 22.4 -- std::move (https://www.learncpp.com/cpp-tutorial/stdmove/)[3].

Some notes about ref-qualified member functions

First, for a given function, non-ref-qualified overloads and ref-qualified overloads cannot coexist. Use one or the other.

Second, similar to how a const lvalue reference can bind to an rvalue, if only a const lvalue-qualified function exists, it will accept either lvalue or rvalue implicit objects.

Third, either qualified overload can be explicitly deleted (using `= delete`), which prevents calls to that function. For example, deleting the rvalue-qualified version prevents use of the function with rvalue implicit objects.

So why don't we recommend using ref-qualifiers?

While ref-qualifiers are neat, there are some downsides to using them in this way.

- Adding rvalue overloads to every getter that returns a reference adds clutter to the class, to mitigate against a case that isn't that common and is easily avoidable with good habits.
- Having an rvalue overload return by value means we have to pay for the cost of a copy (or move) even in cases where we could have used a reference safely (e.g. in case 1 of the example at the top of the lesson).

Additionally:

- Most C++ developers are not aware of this feature (which can lead to errors or inefficiencies in use).
- The standard library typically does not make use of this feature.

Based on all of the above, we are not recommending the use of ref-qualifiers as a best practice. Instead, we recommend always using the result of an access function immediately and not saving returned references for use later.

Next lesson
15.x Chapter 15 summary and quiz

4

Back to table of contents

5

Previous lesson
15.9 Friend classes and friend member functions

6

7

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/[9] are connected to your provided email address.

27 COMMENTS

Newest

Kania
March 15, 2025 5:03 am PDT

These comments savage LOL

1 Reply

Tim
February 18, 2025 7:37 am PST

We gotta stop going in depth about some part of C++ and after 10-15 minutes of reading and note taking you hit us with the:

So why don't we recommend using ref-qualifiers?

16 Reply

Mikhail
November 18, 2024 12:05 am PST

Is it correct that copy and move elision eliminates the need of this feature?

Last edited 7 months ago by Mikhail

1 Reply

Alex Author
Reply to Mikhail[10] November 18, 2024 2:54 pm PST

No. Ref qualifiers allow us to handle lvalue implicit objects and rvalue implicit objects differently. Elision only comes into play in this case due to the conversions being performed by these functions. Ref qualified functions do not need to perform conversions, or even return a value.

3 Reply

moamen
September 26, 2024 8:04 pm PDT

I was reading one of Bjarn Stroustrup's books, and he used a const reference return type to implement the vector.
myCode:
https://github.com/notpythonics/vector_implemention/blob/master/vector_implemention/vector.h

0 Reply

Friend
July 19, 2024 8:11 am PDT

According to this sentence "**... if only an lvalue-qualified overload is provided (i.e. the rvalue-qualified version is not defined), any call to the function with an rvalue implicit object will result in a compilation error ...**", the following code should not compile, however it runs normally:

```
struct A
{
    int x{ 10 };

    const int& get() const &
    {
        std::cout << "Hi: ";
        return x;
    }
};

int main()
{
    std::cout << A{}.get() << '\n';
    return 0;
}
```

Prints "**Hi: 10**". Did I misunderstand the sentence? Or why does this run?

Last edited 11 months ago by Friend

0 Reply

Alex Author
Reply to Friend 11 July 21, 2024 6:39 pm PDT

My mistake. It's true for non-const lvalue-qualified functions, but it appears const lvalue-qualified functions will accept rvalue implicit objects. Lesson text fixed. Thanks for pointing this out.

2 Reply

Spesader
May 28, 2024 8:52 am PDT

Now I can confidently say that I know what most C++ developers don't. Thanks Alex.

27 Reply

Cruiser
May 24, 2024 8:57 am PDT

Hey, I've noticed it in a few examples in a few different lessons, and I don't remember if it was explained, but what's up with the const between round and curly parenthesis?

`const std::string& getName() const & { return m_name; } // & qualifier overloads function to match only lvalue implicit objects`
`std::string getName() const && { return m_name; } // && qualifier overloads function to match only rvalue implicit objects`

What do they do and why are there in that weird place?

0 Reply

Alex Author
Reply to Cruiser [12] May 28, 2024 9:09 am PDT

The right-most const (between the round and curly brackets) makes the function a const member function. See https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/

0 Reply

Cruiser
Reply to Alex [13] May 28, 2024 10:06 am PDT

Thanks!

0 Reply

X-x
May 6, 2024 2:16 pm PDT

```
const std::string& getName() const &  { return m_name; } //  & qualifier overloads
function to match only lvalue implicit objects, returns by reference
```

In fact, `const std::string& getName() const &` can be matched with rvalue object. Is the code below safe?

```cpp
class A {
public:
    const std::string& bar() const & { return m_data; }
    std::string m_data = "a data";
};


int main() {
   auto& const_data = A().bar(); // OK, because returned by const std::string& that extends lifetime of m_data?
}
```

0 Reply

Alex Author

Reply to X-x 14 May 8, 2024 2:49 pm PDT

Not really. `const_data` is a reference to the m_data of a temporary object. That temporary object will be destroyed at the end of the full expression in which it is created. This will leave `const_data` dangling, and use in subsequent statements will result in undefined behavior.

You should not store references to the members of temporary objects. This is covered in https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/

0 Reply

Baker

May 4, 2024 8:35 am PDT

“First, for a given function, non-ref-qualified overloads and ref-qualified overloads cannot coexist. Use one or the other.” So you either have functions overloaded with an ampersand or you don’t?

0 Reply

Alex Author

Reply to Baker 15 May 8, 2024 10:49 am PDT

Yep. You can't mix ref-qualified and non-ref-qualified overloads.

0 Reply

Phargelm

May 1, 2024 7:27 am PDT

It looks like rvalue overloading isn’t a solution for all cases. Let’s say:

```cpp
const std::string& ref { createEmployee("Garbo").getName() }; // getName() is rvalue overloading and returns by copy, it prevents dangling reference.
std::cout << ref << '\n';
```

Here it fixes the problem with dangling reference. But:

```cpp
std::cout << createEmployee("Frank").getName() << '\n';
```

In this example rvalue overloading returns a copy of std::string, but as we use it immediately in the same expression we are actually Ok if we would get a reference instead to avoid a copy.

0 Reply

Alex Author

Reply to Phargelm[16] May 3, 2024 12:11 pm PDT

Yep, this is pretty much what is covered in the "So why don't we recommend using ref-qualifiers?" section.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/
3. https://www.learncpp.com/cpp-tutorial/stdmove/
4. https://www.learncpp.com/cpp-tutorial/chapter-15-summary-and-quiz/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/
7. https://www.learncpp.com/ref-qualifiers/
8. https://www.learncpp.com/cpp-tutorial/deleting-functions/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-604252
11. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-599847
12. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-597495
13. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-597635
14. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-596758
15. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-596651
16. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/#comment-596525
17. https://g.ezoic.net/privacy/learncpp.com