15.10 — Ref qualifiers

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, 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:

```
#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()`

```
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:

```
#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:

```
// 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.

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, 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. This provides a useful way to completely prevent use of a 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.