14.7 — Member functions returning references to data members

learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/

In lesson 12.12 -- Return by reference and return by address, we covered return by reference. In particular, we noted, “The object being returned by reference must exist after the function returns”. This means we should not return local variables by reference, as the reference will be left dangling after the local variable is destroyed. However, it is generally okay to return by reference either function parameters passed by reference, or variables with static duration (either static local variables or global variables), as they will generally not be destroyed after the function returns.

For example:

```
// Takes two std::string objects, returns the one that comes first alphabetically
const std::string& firstAlphabetical(const std::string& a, const std::string& b)
{
	return (a < b) ? a : b; // We can use operator< on std::string to determine
which comes first alphabetically
}

int main()
{
	std::string hello { "Hello" };
	std::string world { "World" };

	std::cout << firstAlphabetical(hello, world); // either hello or world will
be returned by reference

	return 0;
}
```

Member functions can also return by reference, and they follow the same rules for when it is safe to return by reference as non-member functions. However, member functions have one additional case we need to discuss: member functions that return data members by reference.

This is most commonly seen with getter access functions, so we’ll illustrate this topic using getter member functions. But note that this topic applies to any member function returning a reference to a data member.

Returning data members by value can be expensive

Consider the following example:

```
#include <iostream>
#include <string>

class Employee
{
	std::string m_name{};

public:
	void setName(std::string_view name) { m_name = name; }
	std::string getName() const { return m_name; } //  getter returns by value
};

int main()
{
	Employee joe{};
	joe.setName("Joe");
	std::cout << joe.getName();

	return 0;
}
```

In this example, the `getName()` access function returns `std::string m_name` by value.

While this is the safest thing to do, it also means that an expensive copy of `m_name` will be made every time `getName()` is called. Since access functions tend to be called a lot, this is generally not the best choice.

Returning data members by lvalue reference

Member functions can also return data members by (const) lvalue reference.

Data members have the same lifetime as the object containing them. Since member functions are always called on an object, and that object must exist in the scope of the caller, it is generally safe for a member function to return a data member by (const) lvalue reference (as the member being returned by reference will still exist in the scope of the caller when the function returns).

Let’s update the example above so that `getName()` returns `m_name` by const lvalue reference:

```
#include <iostream>
#include <string>

class Employee
{
	std::string m_name{};

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

int main()
{
	Employee joe{}; // joe exists until end of function
	joe.setName("Joe");

	std::cout << joe.getName(); // returns joe.m_name by reference

	return 0;
}
```

Now when `joe.getName()` is invoked, `joe.m_name` is returned by reference to the caller, avoiding having to make a copy. The caller then uses this reference to print `joe.m_name` to the console.

Because `joe` exists in the scope of the caller until the end of the `main()` function, the reference to `joe.m_name` is also valid for the same duration.

Key insight

It is okay to return a (const) lvalue reference to a data member. The implicit object (containing the data member) still exists in the scope of the caller after the function returns, so any returned references will be valid.

The return type of a member function returning a reference to a data member should match the data member's type

In general, the return type of a member function returning by reference should match the type of the data member being returned. In the above example, `m_name` is of type `std::string`, so `getName()` returns `const std::string&`.

Returning a `std::string_view` would require a temporary `std::string_view` to be created and returned every time the function was called. That's needlessly inefficient. If the caller wants a `std::string_view`, they can do the conversion themselves.

Best practice

A member function returning a reference should return a reference of the same type as the data member being returned, to avoid unnecessary conversions.

For getters, using `auto` to have the compiler deduce the return type from the member being returned is a useful way to ensure that no conversions occur:

```
#include <iostream>
#include <string>

class Employee
{
	std::string m_name{};

public:
	void setName(std::string_view name) { m_name = name; }
	const auto& getName() const { return m_name; } // uses `auto` to deduce
return type from m_name
};

int main()
{
	Employee joe{}; // joe exists until end of function
	joe.setName("Joe");

	std::cout << joe.getName(); // returns joe.m_name by reference

	return 0;
}
```

Related content

We cover `auto` return types in lesson 10.9 -- Type deduction for functions.

However, using an `auto` return type obscures the return type of the getter from a documentation perspective. For example:

```
const auto& getName() const { return m_name; } // uses `auto` to deduce return type
from m_name
```

It's unclear what kind of string this function actually returns (it could be a `std::string`, `std::string_view`, `C-style string`, or something else entirely!).

For this reason, we'll generally prefer explicit return types.

Rvalue implicit objects and return by reference

There's one case we need to be a little careful with. In the above example, `joe` is an lvalue object that exists until the end of the function. Therefore, the reference returned by `joe.getName()` will also be valid until the end of the function.

But what if our implicit object is an rvalue instead (such as the return value of some function that returns by value)? Rvalue objects are destroyed at the end of the full expression in which they are created. When an rvalue object is destroyed, any references to members of that rvalue will be invalidated and left dangling, and use of such references will produce undefined behavior.

Therefore, a reference to a member of an rvalue object can only be safely used within the full expression where the rvalue object is created.

Tip

We covered what a full expression is in lesson 1.10 -- Introduction to expressions.

Warning

An rvalue object is destroyed at the end of the full expression in which it is created. Any references to members of the rvalue object are left dangling at that point.

A reference to a member of an rvalue object can only be safely used within the full expression where the rvalue object is created.

Let’s explore some cases related to this:

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

class Employee
{
	std::string m_name{};

public:
	void setName(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;
	e.setName(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();

	// 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; // undefined behavior

	// Case 3: okay: copy referenced value to local variable for use later
	std::string val { createEmployee("Hans").getName() }; // makes copy of
referenced member
	std::cout << val; // okay: val is independent of referenced member

	return 0;
}
```

When `createEmployee()` is called, it will return an `Employee` object by value. This returned `Employee` object is an rvalue that will exist until the end of the full expression containing the call to `createEmployee()`. When that rvalue object is destroyed, any references to members of that object will become dangling.

In case 1, we call `createEmployee("Frank")`, which returns an rvalue `Employee` object. We then call `getName()` on this rvalue object, which returns a reference to `m_name`. This reference is then used immediately to print the name to the console. At this point, the full expression containing the call to `createEmployee("Frank")` ends, and the rvalue object and its members are destroyed. Since neither the rvalue object or its members are used beyond this point, this case is fine.

In case 2, we run into problems. First, `createEmployee("Garbo")` returns an rvalue object. We then call `getName()` to get a reference to the `m_name` member of this rvalue. This `m_name` member is then used to initialize `ref`. At this point, the full expression containing the call to `createEmployee("Garbo")` ends, and the rvalue object and its members are destroyed. This leaves `ref` dangling. Thus, when we use `ref` in the subsequent statement, we're accessing a dangling reference, and undefined behavior results.

Key insight

The evaluation of a full expression ends *after* any uses of that full expression as an initializer. This allows objects to be initialized with an rvalue of the same type (as the rvalue won't be destroyed until after initialization occurs).

But what if we want to save a value from a function that returns a member by reference for use later? Instead of using the returned reference to initialize a local reference variable, we can instead use the returned reference to initialize a non-reference local variable.

In case 3, we're using the returned reference to initialize non-reference local variable `val`. This will cause the member being referenced to be copied into `val`. After initialization, `val` exists independently of the reference. So when the rvalue object is subsequently destroyed, `val` is not impacted by this. Thus `val` can be output in future statements without issue.

Using member functions that return by reference safely

Despite the potential danger with rvalue implicit objects, it is conventional for getters to return types that are expensive to copy by const reference, not by value.

Given that, let's talk about how we can use the return values from such functions safely. The three cases in the example above illustrate the three key points:

- Prefer to use the return value of a member function that returns by reference immediately (illustrated in case 1). Since this works with both lvalue and rvalue objects, if you always do this, you will avoid trouble.
- Do not "save" a returned reference to use later (illustrated in case 2), unless you are sure the implicit object is an lvalue. If you do this with an rvalue implicit object, undefined behavior will result when you use the now-dangling reference.
- If you do need to persist a returned reference for use later and aren't sure that the implicit object is an lvalue, using the returned reference as the initializer for a non-reference local variable, which will make a copy of the member being referenced into the local variable (illustrated in case 3).

Best practice

Prefer to use the return value of a member function that returns by reference immediately, to avoid issues with dangling references when the implicit object is an rvalue.

Do not return non-const references to private data members

Because a reference acts just like the object being referenced, a member function that returns a non-const reference provides direct access to that member (even if the member is private).

For example:

```
#include <iostream>

class Foo
{
private:
    int m_value{ 4 }; // private member

public:
    int& value() { return m_value; } // returns a non-const reference (don't do this)
};

int main()
{
    Foo f{};                // f.m_value is initialized to default value 4
    f.value() = 5;          // The equivalent of m_value = 5
    std::cout << f.value(); // prints 5

    return 0;
}
```

Because `value()` returns a non-const reference to `m_value`, the caller is able to use that reference to directly access (and change the value of) `m_value`.

This allows the caller to subvert the access control system.

Const member functions can’t return non-const references to data members

A const member function is not allowed to return a non-const reference to members. This makes sense -- a const member function is not allowed to modify the state of the object, nor is it allowed to call functions that would modify the state of the object. It should not be doing anything that might lead to the modification of the object.

If a const member function was allowed to return a non-const reference to a member, it would be handing the caller a way to directly modify that member. This violates the intent of a const member function.