14.7 — Member functions returning references to data members

ALEX[1] MARCH 29, 2024

In lesson 12.12 -- Return by reference and return by address (https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/)[2], 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:

```cpp
#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:

```cpp
#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 (https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/)[3].

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

```
1 | 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 (https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/)[4].

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:

```cpp
#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:

```cpp
#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.

Next lesson

14.8 The benefits of data hiding (encapsulation)

5

Back to table of contents

6

Previous lesson

14.6 Access functions

7

8

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[11] are connected to your provided email address.

Notify me about replies:

POST COMMENT

51 COMMENTS

Newest

Tobito

May 14, 2025 9:25 pm PDT

std::cout << createEmployee("Frank").getName()

const std::string& ref { createEmployee("Garbo").getName()

std::string val { createEmployee("Hans").getName()

Isn't Frank, Garbo, and Hans are c style string which its life time until end of program? Why it be destroyed before end of program?

1 Reply

AlexorVorgo

April 14, 2025 2:36 pm PDT

Mixing the uni lectures and this course is helping me understand OOP (and C++ in general) much better and faster. Very nicely written Alex!

2 Reply

Aditya

March 14, 2025 8:14 am PDT

Why in case of this code, where I have intialized the private member variable m_year inside the getYear member function instead leads to the m_year value not being changed when I try returing non const private data member using reference, while in the section above with heading: * *Do not return non-const references to private data members**, you intialized the data member outside the member function, and this leads the data member value being able to be changed using reference in the calling environment:

```cpp
#include <iostream>

class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    int& getYear()
    {
        return m_year = 4;
    }
};

int main()
{
    Date today;
    std::cout << today.getYear() << '\n';
    today.getYear() = 5;
    std::cout << today.getYear() << '\n';

    return 0;
}
```

0 Reply

shayan ahmad

Reply to Aditya [12] March 27, 2025 4:47 am PDT

the first call std::cout << today.getYear() << '\n'; during returning assign 4 to m_year and return the m_year,now m_year with cout prints 4;
second call to today.getYear() = 5; first again assign 4 at call but as you did = 5 this time assign 5;
third call to std::cout << today.getYear() << '\n'; again assign 4 to m_year ,cout prits again 4 so that it,
btw i never saw this type of code where we assign to returned value ,so i save your code as "some_strange_code.cpp"

1 Reply

Teretana

Reply to Aditya [12] March 26, 2025 6:01 pm PDT

I think i see the problem lets go line by line

Alex initializes his year to be 4
You zero initialize. This part is not important.

Line 20.
You call getYear which returns m_year=4. you re saying, i want this 0 to be 4.
Alex just says give me whatever m_year is, or

```cpp
return m_year;
```

Line 21.
You call today.getYear()=5, which says basically, this 4 in line 20 is gonna be 5 now.

Alex says, whatever value getYear returns, its gonna be 5 now.

Finally, Line 22. the problem
Your call of getYear() here says: that 5 i just set in line 21. i want it to be 4 now.
because of line 13. That getYear sets ANY value m_year has previously had to 4, every single time you call it.
Meanwhile Alex`s getYear just says, gimme whatever value m_year is.

Hope this helps. if you change your code to this

```
class Date
{
private:
    int m_year{};
    int m_month{};
    int m_day{};

public:
    int& getYear()
    {
        return m_year; //no longer returns 4 every time
    }
};

int main()
{
    Date today;
    std::cout << today.getYear() << '\n';
    today.getYear() = 5;
    std::cout << today.getYear() << '\n';

    return 0;
```

It should sort the problem.

DISCLAIMER, please note i m also a student, thus i m prone to making a mistake. Await alex's response. He s the sensei we need but dont deserve. Cheers and gl learning!

Last edited 3 months ago by Teretana

1 Reply

KLAP

December 30, 2024 4:31 pm PST

In the "Rvalue implicit objects and return by reference" sections, it was late and my brain was fried so I couldn't fully understand all the technical terms. But in simple understanding of scope, `createEmployee("Garbo").getName()` returned `Employee e` with `std::string m_name{"garbo"}` but `e` is destroyed when function `createEmployee()` finished executed, which led to `Employee& ref` referencing nothing right? And if we copy "Employee e" instead of referencing it, no fouls or harm done?

Last edited 5 months ago by KLAP

3 Reply

Alex Author

Reply to KLAP [13] January 4, 2025 10:15 pm PST

Yup.

5 Reply

NordicCat

December 21, 2024 5:58 am PST

```cpp
#include <iostream>
#include <string>
#include <string_view>
using namespace std;


class Car {
private :
    string m_name{};
    string m_color{};
    int m_speed{};
    int m_year{};

public :
    // Set Value to Private Variable name
    void setName(string& name) {
        m_name = name;
    }
    // get Value of Name
    string_view getName() const{
        return m_name;
    }
    // Set Value to Color
    void setColor(string& color) {
        m_color = color;
    }
    // get Value of Color
    string_view getColor() const{
        return m_color;
    }
    // Set Value to Speed
    void setSpeed(int& speed) {
        m_speed = speed;
    }
    // get Value of Speed
    int getSpeed() const{
        return m_speed;
    }

    //  Set Value to Year
    void setYear(int& year) {
        m_year = year;
    }
    // get Value of Year
    int getYear() const{
        return m_year;
    }

    // Print Values of Car Details
    void printValue() {
        cout << m_name << endl;
        cout << m_color << endl;
        cout << m_year << endl;
        cout << m_speed << endl;
    }
};

int main() {
    Car myCar;
    string color{ "Blue" };
    int speed{ 400 };
    int year{ 2001 };
    string name{ "Mustang" };
    myCar.setColor(color);
    myCar.setSpeed(speed);
    myCar.setName(name);
    myCar.setYear(year);

    myCar.printValue();
}
```

Okay, it is too long but isn't it okay to return by string_view rather than const reference?

0 Reply

Alex Author

Reply to NordicCat [14] December 29, 2024 6:34 pm PST

Getters should always return the same type as the member to avoid unnecessary conversions.

2 Reply

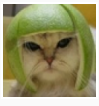
Cpp Learner

December 12, 2024 12:15 pm PST

Let's pretent the `int` is extremely complex type and this struct will be initialized in an array 1 million times. Is this getter and setter most efficient and performed way?

```
#include <iostream>

class Date
{
private:
    int Year{2020};
    int Month{10};
    int Day{14};

public:
    void print() const
    {
        std::cout << Year << '/' << Month << '/' << Day << '\n';
    }

    [[nodiscard]] const int& getYear() const { return Year; } // getter for year
    void setYear(const int& year) { Year = year; } // setter for year

    [[nodiscard]] const int& getMonth() const { return Month; } // getter for month
    void setMonth(const int& month) { Month = month; } // setter for month

    [[nodiscard]] const int& getDay() const { return Day; } // getter for day
    void setDay(const int& day) { Day = day; } // setter for day
};

int main()
{
    Date d{};
    d.setYear(2021);
    std::cout << "The year is: " << d.getYear() << '\n';
    return 0;
}
```

0 Reply



Alex Author

Reply to Cpp Learner [15] December 19, 2024 8:33 pm PST

It's a good start. If the expensive type is movable, you might want to use ref-qualifiers to add a move-

enabled setter.

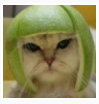
0 Reply

David Pinheiro
November 3, 2024 4:10 am PST

Do member variables benefit from lifetime extension? If so is it only the member or the whole object that gets its lifetime extended?

I don't think this is the case but think a note explicitly stating this would be useful in this lesson!

1 Reply



Alex Author
Reply to David Pinheiro [16] November 7, 2024 4:35 pm PST

No, members can't be extended. Only the entire object can have its lifetime extended.

I added a note about this in lesson https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/

This lesson is about member functions returning references. The objects returned by reference can never be extended because lifetime extension doesn't work across function boundaries.

1 Reply

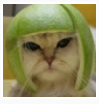
Swaminathan R
June 2, 2024 1:57 am PDT

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

But this is true for normal functions also correct? And I guess this holds too for return by value too? Then why did we specifically include this in the member function topic?

Last edited 1 year ago by Swaminathan R

1 Reply



Alex Author
Reply to Swaminathan R [17] June 4, 2024 8:56 pm PDT

Yes, this is also generally true for non-member functions (the return type of a non-member function returning a reference to any object should typically match the object's type).

It does not hold true for return by value. Since return by value constructs a new object anyway, it's fine if this new object has a different type.

This was included here because we just discussed getters, which typically return by reference.

2 Reply

emme
May 20, 2024 5:19 am PDT

"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."

Why is it a bad thing though? That way we don't even need any extra functions to get/set data, and just do away with references whenever we need to get some info from class or set some variable inside.

0 Reply

Alex Author
Reply to emme [18] May 21, 2024 5:21 pm PDT

Because it violates encapsulation. If you do that, you might as well just make all your members public and not bother with access functions.

5 Reply

Masdas
March 7, 2024 5:58 am PST

Why does this work:
`auto& getName() const { return m_name; }`
In this lesson it is stated that const member functions can’t return non-const references, so does auto recognize the return type should be const std::string& ?

0 Reply

Alex Author
Reply to Masdas [19] March 10, 2024 4:06 pm PDT

Yes, `auto` implicitly adds `const` in this case. It's not obvious this is occurring if you glance at the return type, so it's better to explicitly use `const auto&`.

5 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/
3. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/
5. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/

6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/access-functions/
8. https://www.learncpp.com/member-functions-returning-references-to-data-members/
9. https://www.learncpp.com/cpp-tutorial/alias-templates/
10. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-608537
13. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-605983
14. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-605526
15. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-605196
16. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-603805
17. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-597845
18. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-597327
19. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/#comment-594407
20. https://g.ezoic.net/privacy/learncpp.com