14.6 — Access functions

ALEX[1] DECEMBER 29, 2024

In previous lesson 14.5 -- Public and private members and access specifiers (https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/)[2], we discussed the public and private access levels. As a reminder, classes typically make their data members private, and private members can not be directly accessed by the public.

Consider the following `Date` class:

```
#include <iostream>

class Date
{
private:
    int m_year{ 2020 };
    int m_month{ 10 };
    int m_day{ 14 };

public:
    void print() const
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }
};

int main()
{
    Date d{};  // create a Date object
    d.print(); // print the date

    return 0;
}
```

While this class provides a `print()` member function to print the entire date, this may not be sufficient for what the user wants to do. For example, what if the user of a `Date` object wanted to get the year? Or to change the year to a different value? They would be unable to do so, as `m_year` is private (and thus can't be directly accessed by the public).

For some classes, it can be appropriate (in the context of what the class does) for us to be able to get or set the value of a private member variable.

Access functions

An **access function** is a trivial public member function whose job is to retrieve or change the value of a private member variable.

Access functions come in two flavors: getters and setters. **Getters** (also sometimes called **accessors**) are public member functions that return the value of a private member variable. **Setters** (also sometimes called **mutators**) are public member functions that set the value of a private member variable.

Nomenclature

The term "mutator" is often used interchangably with "setter". But more broadly, a **mutator** is any member function that modifies (mutates) the state of the object. Under this definition, a setter is a specific kind of mutator. However, it is also possible to have non-setter functions that qualify as mutators.

Getters are usually made const, so they can be called on both const and non-const objects. Setters should be non-const, so they can modify the data members.

For illustrative purposes, let's update our `Date` class to have a full set of getters and setters:

```
#include <iostream>

class Date
{
private:
    int m_year { 2020 };
    int m_month { 10 };
    int m_day { 14 };

public:
    void print()
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }

    int getYear() const { return m_year; }        // getter for year
    void setYear(int year) { m_year = year; }     // setter for year

    int getMonth() const  { return m_month; }     // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() const { return m_day; }          // getter for day
    void setDay(int day) { m_day = day; }         // setter for day
};

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

    return 0;
}
```

This prints:

```
The year is: 2021
```

Access function naming

There is no common convention for naming access functions. However, there are a few naming conventions that are more popular than others.

- Prefixed with "get" and "set":

```
int getDay() const { return m_day; }  // getter
void setDay(int day) { m_day = day; } // setter
```

The advantage of using “get” and “set” prefixes is that it makes it clear that these are access functions (and should be inexpensive to call).

- No prefix:

```
int day() const { return m_day; }  // getter
void day(int day) { m_day = day; } // setter
```

This style is more concise, and uses the same name for both the getter and setter (relying on function overloading to differentiate the two). The C++ standard library uses this convention.

The downside of the no-prefix convention is that it is not particularly obvious that this is setting the value of the day member:

```
d.day(5); // does this look like it's setting the day member to 5?
```

Key insight

One of the best reasons to prefix private data members with “m_” is to avoid having data members and getters with the same name (something C++ doesn’t support, although other languages like Java do).

- “set” prefix only:

```
int day() const { return m_day; }     // getter
void setDay(int day) { m_day = day; } // setter
```

Which of the above you choose is a matter of personal preference. However, we highly recommend using the “set” prefix for setters. Getters can use either a “get” prefix or no prefix.

Tip

Use a “set” prefix on your setters to make it more obvious that they are changing the state of the object.

Getters should return by value or by const lvalue reference

Getters should provide “read-only” access to data. Therefore, the best practice is that they should return by either value (if making a copy of the member is inexpensive) or by const lvalue reference (if making a copy of the member is expensive).

Because returning data members by reference is a non-trivial topic, we’ll cover that topic in more detail in lesson [14.7 -- Member functions returning references to data members (https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/)](https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/)[3].

Access functions concerns

There is a fair bit of discussion around cases in which access functions should be used or avoided. Many developers would argue that use of access functions violates good class design (a topic that could easily fill an entire book).

For now, we'll recommend a pragmatic approach. As you create your classes, consider the following:

- If your class has no invariants and requires a lot of access functions, consider using a struct (whose data members are public) and providing direct access to members instead.
- Prefer implementing behaviors or actions instead of access functions. For example, instead of a `setAlive(bool)` setter, implement a `kill()` and a `revive()` function.
- Only provide access functions in cases where the public would reasonably need to get or set the value of an individual member.

Why make data private if we're going to provide a public access function to it?

So glad you asked. We'll answer this question in upcoming lesson [14.8 -- The benefits of data hiding (encapsulation) (https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/)](https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/)[4].

Next lesson
14.7 Member functions returning references to data members

3

Back to table of contents

5

Previous lesson
14.5 Public and private members and access specifiers

2

6

---

djlwadhwah
April 12, 2025 11:36 pm PDT

So instead of simply setting Data.year = 2025, which is simple, fast, efficient and just works. You've introduced an extra function, more code, more spaghetti and more places for bugs. And then have the audacity to say that OOP is somehow an improvement. Amazing.

Also if your function allows division by zero, the solution isn't to make a new function that stops you from setting the data to zero, it's to make sure the function itself doesn't allow dividing by zero. Because no matter how much OOP spaghetti you add trying to control state, the bug is still inside the function. It's like giving a house new paint while it's rotting on the inside. All you're doing is covering up your error. At some point that function will receive a zero, your program will die, and the rocket will explode.

Don't use OOP kids. Learn math instead.

0 Reply

Dietskittles
Reply to djlwadhwah 9 May 1, 2025 12:18 pm PDT

The audacity to critique a tutorial when it clearly stated the example was for **illustrative purposes**.

1 Reply

Jeremy
Reply to djlwadhwah 9 April 25, 2025 2:25 pm PDT

Someone doesn't understand abstraction on large-scale projects.

5 Reply

paulisprouki
Reply to djlwadhwah 9 April 23, 2025 5:00 pm PDT

he is just teaching with simple examples. In a program of thousands of lines using classes and structs correctly would be a massive benefit compared to making everything public.

5 Reply

Teretana
April 7, 2025 7:22 pm PDT

If we have a private member called m_random that is a std::string. Getter returning a const reference is the norm. Would returning std::string_view, despite not being the norm, work too. Is there an inherent advantage/disadvantage to it, or just common practice (my colleagues will hate me for using std::string view)?

I m asking because i originally made that sv getter in one of the exercises here and went to stackoverflow in pursuit of answers. Here is what i found , but i dont understand it properly. I m not at a level where what i have learnt can be implemented in real world applications and code.

Anyways the quote is.
"string has one notable semantic difference to an arbitrary string_view: it's null-terminated by guarantee. We know this. Maybe some downstream user needs to rely on that information. If they need null-termination (e.g. they need to pass to some C API that requires it) and you give a string_view, they have to make a string out of it themselves. You save nothing, but potentially make downstream users do more work."

I assume not being null terminated by default has some serious problems down the pipe, however after thinking about it for days i dont see the problem, besides possible buffer overflow. Yet there are ways, as you have shown to protect against that. It is more work though.

Furthermore, both std::string_view and const ref suffer from "dangling" issues, so there shouldn't be a difference there.

Finally i saw this comment, and i dont understand why he says this:
"I guess that I would only use string view when I have to deal with a char*. If I already have a std::string, I would transfer it to a string_view only if I needed a substring."

If there was a private member called m_randomtwo that was a char pointer that pointed to some string, pointers are still lvalues, thus able to be referenced? Or since string_view is a hidden pointer, its easier to work with it?

And as a PS, most of these questions are very basic i m sure and i m happy if you dont bother to reply at all. I understand its a learning curve, and i ll read and read till i get it right. Save your time and efforts into making the guide better.

Thank you for everything Alex, you re a good man

Edit 1 found this nice article: https://blog.hiebl.cc/posts/how-to-use-std-string-view/

"Return types:

Don’t use string_view, as it is hard for the callee to guarantee that the string_view’s data outlive the returned object
Exception: The returned string_view is a substring of a passed argument.
Don’t use string_view in getters, not even in getters for constant members
Class members:

Don’t use string_view, unless you are sure they have a static lifetime"

Edit 2.
"❌ Getter
Returning a string_view in a getter method is almost a guarantee for trouble. The string_view becomes invalid as soon as the underlying (member-)value is changed. " (by a setter)

Last edited 2 months ago by Teretana

0 Reply

NordicCat
December 21, 2024 5:08 am PST

What will happen if we do not want to set the values manually just like we did when we use "setter-functions" instead we want to take a value from user?

Last edited 6 months ago by NordicCat

0 Reply

Alex Author
Reply to NordicCat [10] December 29, 2024 6:04 pm PST

We do this via constructors. Keep reading.

2 Reply

Cpp Learner
December 12, 2024 11:55 am PST

I think this chapter would be excellent time to explain the use case of `[[nodiscard]]` I remember in previous chapters `[[nodiscard]]` being used but for beginner it would be hard to understand at that time the real purpose of `[[nodiscard]]`

0 Reply

Alex Author
Reply to Cpp Learner [11] December 19, 2024 8:21 pm PST

Perhaps. But I'm thinking it might be more suitable to discuss in the chapter on error detection and handling. Thoughts?

0 Reply

Colin
December 12, 2024 7:11 am PST

Hey Alex,
Coming from Java, I find it a little hard to prefix private variables with `m_`. Is it acceptable if I alternatively prefix the function arguments?

For example:

```
class Foo
{
private:
    int health{};

public:
    int getHealth() const
        return health;

    void setHealth(int f_health)
        health = f_health;
};
```

Thanks

Last edited 6 months ago by Colin

0 Reply

Alex Author

Reply to Colin [12] December 16, 2024 8:21 pm PST

Not really. It's not conventional to prefix function arguments in C++, and it means your data member names can still collide with your member function names.

1 Reply

Jacob

January 22, 2024 5:46 am PST

Hi Alex. I'm plodding along here, enjoying your course.
In the final topic here, "Access Function Concerns" you provide this advice:
> Prefer implementing behaviors or actions instead of access functions.
> For example, instead of a setAlive(bool) setter, implement a kill() function.

This is big DUH to me. Perhaps you might elaborate on what your hypothetical setAlive() and kill() subroutines (whoops - methods) do and how they would accomplish the same (or opposite) purposes?

3 Reply

Alex Author

Reply to Jacob [13] January 23, 2024 10:16 am PST

It really depends on the implementation, but a minimum viable `kill()` function would probably look like this:

```
void kill()
{
    m_alive = false;
}
```

The benefit here is that you could later extend this to have other on-death behaviors, whereas this is more difficult if you only have an access function.

10 Reply

NordicCat

Reply to Alex [14] December 21, 2024 4:58 am PST

mutators?
Mutators is a really good name when it comes to modifying or checking any specific condition and then changing the value rather than saying it is a setter function like for example :

void kill() should look like this :

```
void kill(){
if (monsterHealth <= 0) {
monsterKilled = true;

killCount++ ;
}
```

then, it will make more sense to everyone.

Is it necessary to call these functions setters? like it confuses me a lot.

Last edited 6 months ago by NordicCat

0 Reply

Alex Author

Reply to NordicCat [15] December 29, 2024 6:04 pm PST

A mutator is a general name for any member function that modifies state, whether it does so conditionally or not.

That implementation of kill() only makes sense if the monster actually has health and if health and the killed flag make sense to be decoupled.

I wouldn't call `kill` a setter.

0 Reply

NordicCat

Reply to Alex [16] December 29, 2024 11:47 pm PST

thanks a lot!

0 Reply

Afrost

October 2, 2023 11:29 am PDT

Hey, just got back and got some basic question that keeps pondering me while I'm learning here..

How do your read a code normally? I mean looking at the example from this chapter

```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
};

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

    return 0;
}
```

Do you read from int main, then go into the called functions?
or basically try to understand and read everything from the functions directly? (which is the class member and member function in here) - which is a top down way of reading

coz some/most of the times, I kinda dont really have a clear understanding when reading top down, but after I reach the main, everything became clear.

Last edited 1 year ago by Afrost

9 Reply

Alex Author

Reply to Afrost [17] October 3, 2023 10:38 am PDT

I'll normally scan a class/struct to get the gist of what it is doing. "So Employee represents some Employee who has a name".

Then I'll go to main() and start from there. "Okay, we're creating an Employee named Joe and printing his name".

C++ programs don't execute from top to bottom, so I wouldn't expect reading a program top down to be that useful in aiding comprehension.

22 Reply

Afrost

Reply to Alex [18] October 6, 2023 3:16 am PDT

Thx for the insight as usual, Alex!

2 Reply

D D

September 16, 2023 9:47 am PDT

The question arises. If we just have simple getter and setter, why do we use methods if we could declare a public member?
The second one. You say don't use a non-const reference. But why don't we use it for getter and setter simultaneously (a simple setter changes a value)?

0 Reply

Alex Author

Reply to D D [19] September 19, 2023 3:08 pm PDT

First question: The answer to this is so important we devote an entire article to it:
https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
Second question: Two primary reasons:
a) This doesn't work for const objects.
b) If we do this we might as well make the member public.

0 Reply

D D

September 16, 2023 6:14 am PDT

[[IMPORTANT TO READ]]
Hello, Alex. I appreciate your lessons a lot and your time spending for writing them and answering our questions. And our suggestions!

I want to leave an important remark that can lead to big mistakes. I change a little our class to show problems. I've added a constructor which will be covered later. The first problem: for const objects our getter won't work.

```cpp
class Employee
{
    std::string m_name{};

public:
    Employee(std::string_view name) : m_name{ name }
    {  }
    const std::string& getName() /*const*/
    {
        std::cout << "getName(): ";
        return m_name;
    }
};

int main()
{
    Employee joe{"Joe"};
    auto& name = joe.getName();
    std::cout << name << '\n';

    const Employee richard{ "Richard" };
    auto& nR = richard.getName(); // compiler error. Error (active) E1086   the object has type qualifiers that are not compatible with the member function "Employee::getName"
    std::cout << nR << '\n';

}
```

That's why we should add `const`-qualifier to the method. It'll be work with both const and non-const objects.

You wrote the next statement: `Getters should return by value or const reference` – yes, I agree with you, but... not always. Your case will be only work with Lvalue references. But we may encounter some big problemes with R-value ref (a temporary object). Because it lives to the end of the full expression. We have a const reference to an object which is already dead.

```cpp
auto& badN = Employee{ "BadUser" }.getName();
std::cout << badN << '\n';
```

Now the reference to `badN` is dangling. If we wrote just auto, we would get a copy of our string (We wanted to get rid of any copies). But we know that a copy may be very expensive. And when we print badN, we could see nothing, but also there can be garbage. And we can write much code but mistakes will happen when we don't expect them.

So, there is some advice how we can handle these problems. In C++11 appeared ref-qualifiers in class methods. They can protect us from mistakes. When we return any references from a method: both const ref and just ref, we must mark it with &. When we add one &, it'll be work with lvalue. Temporarily delete `const`-ref to show another problem.

```cpp
class Employee
{
    std::string m_name{};

public:
    Employee(std::string_view name) : m_name{ name }
    {  }
    const std::string& getName() &
    {
        std::cout << "getName(): ";
        return m_name;
    }
};

int main()
{
    Employee joe{"Joe"};
    auto& name = joe.getName();
    std::cout << name << '\n';

    auto& badN = Employee{ "BadUser" }.getName(); // compiler error
    std::cout << badN << '\n';

    return 0;
}
```

So, our Rvalue will be an error of a compiler as we wanted. Let's undo our edit and add `const` to the `getName()`

```cpp
const std::string& getName() const &
{
    return m_name;
}
```

We'll see that our compiler error with a temporary object has disappeared. Did we win? Hah-hah. Unfortunately, no. We still have a dangling ref, how to fix it?
In previous lessons we learnt that const ref can work with Lvalue, Const LValue and **Rvalue**. For rvalue there is its own qualifier. It looks like two `&&`. To exclude any variants with creating temporarily objects we must forbid function with `R-value` references.

```cpp
class Employee
{
    std::string m_name{};

public:
    Employee(std::string_view name) : m_name{ name }
    {  }
    const std::string& getName() const&
    {
        std::cout << "getName(): ";
        return m_name;
    }
    const std::string& getName() const&& = delete;
};

int main()
{
    Employee joe{"Joe"};
    auto& name = joe.getName();
    std::cout << name << '\n';

    auto& badN = Employee{ "BadUser" }.getName(); // compiler error as we wanted
    std::cout << badN << '\n';

    const Employee richard{ "Richard" };
    auto& nR = richard.getName();
    std::cout << nR << '\n';
    return 0;
}
```

Last edited 1 year ago by D D

0 Reply

Alex Author

Reply to D D [20] September 19, 2023 2:56 pm PDT

1. I'll address the const issue more thoroughly in a bit.
2. I added a section discussing this topic, and put the ref-qualifier tip into an advanced bubble. Thanks for the suggestion.

0 Reply

D D

Reply to Alex [21] September 22, 2023 6:49 am PDT

You could add the information about it in 'Last changes of the site'

0 Reply

D D

September 15, 2023 5:39 am PDT

Hello, Alex.

1. Because getters don't change object and `print()` just show results in a console, we should mark those methods with `const`-qualifier

```
class Date
{
    int m_year{ 2020 };
    int m_month{ 10 };
    int m_day{ 14 };
public:
    void print() const
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }

    [[nodiscard]] int getYear() const { return m_year; }
    void setYear(int year) { m_year = year; }

    [[nodiscard]] int getMonth() const { return m_month; }
    void setMonth(int month) { m_month = month; }

    [[nodiscard]] int getDay() const { return m_day; }
    void setDay(int day) { m_day = day; }

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

    return 0;
}
```

2. It seems the next example is difficult, because you use here a constructor, but you'll explain it in the next lessons.

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

Last edited 1 year ago by D D

0 Reply

Alex Author

Reply to D D [22] September 19, 2023 10:39 am PDT

Thanks for the feedback!

#1 will be fixed when I move the lesson on const member functions into this chapter (soon).
#2 was resolved from a prior comment.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/
3. https://www.learncpp.com/cpp-tutorial/member-functions-returning-references-to-data-members/
4. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
5. https://www.learncpp.com/
6. https://www.learncpp.com/access-functions/
7. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-609213
10. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-605521
11. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-605194
12. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-605183
13. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-592680
14. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-592743
15. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-605520
16. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-605920
17. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-588098
18. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-588177
19. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-587313
20. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-587307
21. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-587430
22. https://www.learncpp.com/cpp-tutorial/access-functions/#comment-587256
23. https://g.ezoic.net/privacy/learncpp.com