15.8 — Friend non-member functions

ALEX[1] MAY 8, 2024

For much of this chapter and last, we've been preaching the virtues of access controls, which provide a mechanism for controlling who can access the various members of a class. Private members can only be accessed by other members of the class and public members can be accessed by everyone. In lesson 14.6 -- Access functions (https://www.learncpp.com/cpp-tutorial/access-functions/)[2], we discussed the benefits of keeping your data private, and creating a public interface for non-members to use.

However, there are situations where this arrangement is either not sufficient or not ideal.

For example, consider a storage class that is focused on managing some set of data. Now lets say you also want to display that data, but the code that handles the display will have lots of options and is therefore complex. You could put both the storage management functions and the display management functions in the same class, but that would clutter things up and make for a complex interface. You could also keep them separate: the storage class manages storage, and some other display class manages all of the display capabilities. That creates a nice separation of responsibility. But the display class would then be unable to access the private members of the storage class, and might not be able to do its job.

Alternatively, there are cases where syntactically we might prefer to use a non-member function over a member function (we'll show an example of this below). This is commonly the case when overloading operators, a topic we'll discuss in future lessons. But non-member functions have the same issue -- they can't access the private members of the class.

If access functions (or other public member functions) already exist and are sufficient for whatever capability we're trying to implement, then great -- we can (and should) just use those. But in some cases, those functions don't exist. What then?

One option would be to add new member functions to the class, to allow other classes or non-member functions to do whatever job they would be otherwise unable to do. But we might not want to allow public access to such things -- perhaps those things are highly implementation dependent, or prone to misuse.

What we really need is some way to subvert the access control system on a case by case basis.

Friendship is magic

The answer to our challenge is friendship.

Inside the body of a class, a **friend declaration** (using the `friend` keyword) can be used to tell the compiler that some other class or function is now a friend. In C++, a **friend** is a class or function (member or non-member) that has been granted full access to the private and protected members of another class. In this way, a class can selectively give other classes or functions full access to their members without impacting anything else.

Key insight

Friendship is always granted by the class whose members will be accessed (not by the class or function desiring access). Between access controls and granting friendship, a class always retains the ability to control who can access its members.

For example, if our storage class made the display class a friend, then the display class would be able to access all members of the storage class directly. The display class could use this direct access to implement display of the storage class, while remaining structurally separate.

The friend declaration is not affected by access controls, so it does not matter where within the class body it is placed.

Now that we know what a friend is, let's take a look at specific examples where friendship is granted to non-member functions, member functions, and other classes. We'll discuss friend non-members functions in this lesson, and then take a look at friend classes and friend member functions in the next lesson [15.9 -- Friend classes and friend member functions (https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/)](3).

Friend non-member functions

A **friend function** is a function (member or non-member) that can access the private and protected members of a class as though it were a member of that class. In all other regards, the friend function is a normal function.

Let's take a look at an example of a simple class making a non-member function a friend:

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }

    // Here is the friend declaration that makes non-member function void print(const Accumulator& accumulator) a friend of Accumulator
    friend void print(const Accumulator& accumulator);
};

void print(const Accumulator& accumulator)
{
    // Because print() is a friend of Accumulator
    // it can access the private members of Accumulator
    std::cout << accumulator.m_value;
}

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

In this example, we've declared a non-member function named `print()` that takes an object of class `Accumulator`. Because `print()` is not a member of the Accumulator class, it would normally not be able

to access private member `m_value`. However, the Accumulator class has a friend declaration making `print(const Accumulator& accumulator)` a friend, this is now allowed.

Note that because `print()` is a non-member function (and thus does not have an implicit object), we must explicitly pass an `Accumulator` object to `print()` to work with.

Defining a friend non-member inside a class

Much like member functions can be defined inside a class if desired, friend non-member functions can also be defined inside a class. The following example defines friend non-member function `print()` inside the `Accumulator` class:

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }

    // Friend functions defined inside a class are non-member functions
    friend void print(const Accumulator& accumulator)
    {
        // Because print() is a friend of Accumulator
        // it can access the private members of Accumulator
        std::cout << accumulator.m_value;
    }
};

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

Although you might assume that because `print()` is defined inside `Accumulator`, that makes `print()` a member of `Accumulator`, this is not the case. Because `print()` is defined as a friend, it is instead treated as a non-member function (as if it had been defined outside `Accumulator`).

Syntactically preferring a friend non-member function

In the introduction to this lesson, we mentioned that there were times we might prefer to use a non-member function over a member function. Let's show an example of that now.

```cpp
#include <iostream>

class Value
{
private:
    int m_value{};

public:
    explicit Value(int v): m_value { v }  { }

    bool isEqualToMember(const Value& v) const;
    friend bool isEqualToNonmember(const Value& v1, const Value& v2);
};

bool Value::isEqualToMember(const Value& v) const
{
    return m_value == v.m_value;
}

bool isEqualToNonmember(const Value& v1, const Value& v2)
{
    return v1.m_value == v2.m_value;
}

int main()
{
    Value v1 { 5 };
    Value v2 { 6 };

    std::cout << v1.isEqualToMember(v2) << '\n';
    std::cout << isEqualToNonmember(v1, v2) << '\n';

    return 0;
}
```

In this example, we've defined two similar functions that check whether two `Value` objects are equal. `isEqualToMember()` is a member function, and `isEqualToNonmember()` is a non-member function. Let's focus on how these functions are defined.

In `isEqualToMember()`, we're passing one object implicitly and the other explicitly. The implementation of the function reflects this, and we have to mentally reconcile that `m_value` belongs to the implicit object whereas `v.m_value` belongs to the explicit parameter.

In `isEqualToNonmember()`, both objects are passed explicitly. This leads to better parallelism in the implementation of the function, as the `m_value` member is always explicitly prefixed with an explicit parameter.

You may still prefer the calling syntax `v1.isEqualToMember(v2)` over `isEqualToNonmember(v1, v2)`. But when we cover operator overloading, we'll see this topic come up again.

Multiple friends

A function can be a friend of more than one class at the same time. For example, consider the following example:

```cpp
#include <iostream>

class Humidity; // forward declaration of Humidity

class Temperature
{
private:
    int m_temp { 0 };
public:
    explicit Temperature(int temp) : m_temp { temp } { }

    friend void printWeather(const Temperature& temperature, const Humidity&
humidity); // forward declaration needed for this line
};

class Humidity
{
private:
    int m_humidity { 0 };
public:
    explicit Humidity(int humidity) : m_humidity { humidity } {  }

    friend void printWeather(const Temperature& temperature, const Humidity&
humidity);
};

void printWeather(const Temperature& temperature, const Humidity& humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}

int main()
{
    Humidity hum { 10 };
    Temperature temp { 12 };

    printWeather(temp, hum);

    return 0;
}
```

There are three things worth noting about this example. First, because `printWeather()` uses both `Humidity` and `Temperature` equally, it doesn't really make sense to have it be a member of either. A non-member function works better. Second, because `printWeather()` is a friend of both `Humidity` and `Temperature`, it can access the private data from objects of both classes. Finally, note the following line at the top of the example:

```cpp
class Humidity;
```

This is a forward declaration for `class Humidity`. Class forward declarations serve the same role as function forward declarations -- they tell the compiler about an identifier that will be defined later. However, unlike functions, classes have no return types or parameters, so class forward declarations are always simply `class ClassName` (unless they are class templates).

Without this line, the compiler would tell us it doesn't know what a `Humidity` is when parsing the friend declaration inside `Temperature`.

Doesn't friendship violate the principle of data hiding?

No. Friendship is granted by the class doing the data hiding with the expectation that the friend will access its private members. Think of a friend as an extension of the class itself, with all the same access rights. As such, access is expected, not a violation.

Used properly, friendship can make a program more maintainable by allowing functionality to be separated when it makes sense from a design perspective (as opposed to having to keep it together for access control reasons). Or when it makes more sense to use a non-member function instead of a member function.

However, because friends have direct access to the implementation of a class, changes to the implementation of the class will typically necessitate changes to the friends as well. If a class has many friends (or those friends have friends), this can lead to a ripple effect.

When implementing a friend function, prefer to use the public interface over direct access to members whenever possible. This will help insulate your friend function from future implementation changes and lead to less code needing to be modified and/or retested later.

Best practice

A friend function should prefer to use the class interface over direct access whenever possible.

Prefer non-friend functions to friend functions

In 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], we mentioned that we should prefer non-member functions over member functions. For the same reasons given there, we should prefer non-friend functions over friend functions.

For example, in the following example, if the implementation of `Accumulator` is changed (e.g. we rename `m_value`), the implementation of `print()` will need to be changed as well:

```
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 }; // if we rename this

public:
    void add(int value) { m_value += value; } // we need to modify this

    friend void print(const Accumulator& accumulator);
};

void print(const Accumulator& accumulator)
{
    std::cout << accumulator.m_value; // and we need to modify this
}

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

A better idea is as follows:

```cpp
#include <iostream>

class Accumulator
{
private:
    int m_value { 0 };

public:
    void add(int value) { m_value += value; }
    int value() const { return m_value; } // added this reasonable access function
};

void print(const Accumulator& accumulator) // no longer a friend of Accumulator
{
    std::cout << accumulator.value(); // use access function instead of direct access
}

int main()
{
    Accumulator acc{};
    acc.add(5); // add 5 to the accumulator

    print(acc); // call the print() non-member function

    return 0;
}
```

In this example, `print()` uses access function `value()` to get the value of `m_value` instead of accessing `m_value` directly. Now if the implementation of `Accumulator` is ever changed, `print()` will not need to be updated.

Best practice

Prefer to implement a function as a non-friend when possible and reasonable.

Be cautious when adding new members to the public interface of an existing class, as every function (even trivial ones) adds some level of clutter and complexity. In the case of `Accumulator` above, it's totally reasonable to have an access function to get the current accumulated value. In more complex cases, it may be preferable to use friendship instead of adding many new access functions to the interface of a class.

Next lesson
15.9 Friend classes and friend member functions

3

Back to table of contents

5

Previous lesson
15.7 Static 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.

528 COMMENTS

Newest

PollyWantsACracker

May 15, 2025 9:49 am PDT

Last edited 1 month ago by PollyWantsACracker

0 Reply

rasp

February 3, 2025 4:46 pm PST

"When implementing a friend function, prefer to use the public interface over direct access to members whenever possible. This will help insulate your friend function from future implementation changes and lead to less code needing to be modified and/or retested later."

please explain a little more.. or maybe an example :D

from what i get, it means using getters/setters in your friend funcs, but then what would be the point of friending it, so my interpretetion is prob wrong :p

Last edited 4 months ago by rasp

2 Reply

Alex Author

Reply to rasp[10] February 5, 2025 6:36 pm PST

Yes, it means using getters/setters when available.

Presumably you're making the function a friend for one of two reasons:

- There is at least one member you need access to that doesn't have a getter. However, for the other members, if they have getters, use them.
- You're doing it for syntactic reasons (e.g. operator overloading). In this case all members might have getters.

2 Reply

KLAP

January 16, 2025 4:30 pm PST

Is this implementation of friend violate the principle of data hiding?

```
#include <iostream>

class Accumulator
{
private:
    int m_value{ 0 };

public:
    void add(int value) { m_value += value; }

    friend void change(Accumulator& accumulator);

    int getValue() { return m_value; }
};

void change(Accumulator& accumulator)
{
    accumulator.m_value = 5;
}

int main()
{
    Accumulator acc{};
    acc.add(10);
    std::cout << acc.getValue() << '\n';

    change(acc);
    std::cout << acc.getValue() << '\n';

    return 0;
}

This prints:
10
5
```

In this code I managed to change the member variable value with non-member functions. Is this still considered not violating the principle of data hiding because technically the non-member function is a friend?

Last edited 5 months ago by KLAP

0 Reply

Bassem

Reply to KLAP [11] March 8, 2025 10:55 am PST

I'm still learning too but I will try to help with this
is this implementation violate the principle of data hiding ? Yes

yes because data hiding isn't like is it there or not , I think of it as levels , your class can be one way more encapsulated than the other way. like for example, preferring non-member function over member functions. because any change in the implementation won't break the code in the non-member function "unless the public interface changes". Conversely, if that exact function was a member and had direct access to the implementation, any changes might result in breaking the code in that function and changes need to be done.

So I think of encapsulation like this >> the more encapsulated a class is , the less code will break when changes done to its private members.

So it's a spectrum, not binary.

Now we agree on that?

by applying the same process of thinking.

in your code, if the private member `m_value` is renamed or replaced for example, by `double accumulated_value {};` it will break. and you'll need to update the code in your change() function. But if you have a setter function

```
// member function
void set_value(int value) { accumulated_value = value;}
```

```
// non-friend non-member function
void change(const Accumulate& a){
                a.set_value(5);
}
```

you will only have to update the setter function, which is inside the class, which is easier to manage.

in your example it doesn't make a difference because you'll have to update m_value in one place anyway, but imagine if you have many functions

```
// friends
void change_to_100(Accumulate& a){
                a.m_value = 100;
}
int calc_factorial(Accumulate& a){
                int factorial{};
                while(a.m_value){
                                        factorial += m_value--;
                }
                return factorial;
                //notice that here we changed the value of m_value accidently
                // and when we use m_value again after anywhere it will be
zero
                // this can be hard to notice and debug
}
// vs the non member non friend version with the gelp of getter function
int calc factorial(Accumulate& a){
                    int value = a.get_value();
                    // you can't make the same mistake here
                    // becuase you can't access and corrupt private data
                    int factorial{};
                    while(value){
                                            factorial += value--;
                    }
                    return factorial;
}
```

Then you'll have to change `m_value` to the new name in several places in all of your friend functions vs changing it once in the setter function.

"Less code is prone to break by doing this"

So in my books , this does violate the concept of data hiding because it wakens it and there's better you can do.

Two things worth noting :

Setters and getters will level up data hiding at the cost of :

1. allowing the public to set and get
2. if your program is **performance critical**, and the function will call setter or getter a **million times** inside a for loop for example, that will add a lot of performance overhead because function call requires saving and restoring registers that can cause measurable performance cost.

It's worth nothing to say that the compiler does inline propagation, and the setter/getter will behave exactly as if it was direct access with no performance overhead. but in many cases where there's branching in the setter or complex logic so it's defined in a separate .cpp file this will prevent inline propagation. vs If you have the class declaration in a header file and private members are included, then the friend function in a translation unit where that header is included will have direct access to the variable so better compiler optimization.

Finally, you don't take anything I just said for granted. You do your due diligence because I might be wrong on something as I'm still naive.

Last edited 3 months ago by Bassem

0 Reply

groovyest

June 5, 2024 9:12 am PDT

can there be friends of friends?

1 Reply

Alex Author

Reply to groovyest [12] June 5, 2024 2:57 pm PDT

Edit: A friend of a friend is not automatically your friend.

But you can always directly friend the friend of a friend so they will be your friend too.

Last edited 5 months ago by Alex

8 Reply

NordicCat

Reply to Alex [13] January 2, 2025 2:50 am PST

btw but If I become a friend of my friend's friend then?

0 Reply

NordicCat

Reply to Alex [13] January 2, 2025 2:50 am PST

this is such a friendful of you Alex.

1 Reply

Claire

May 27, 2024 4:10 pm PDT

Are there reasons, aside from convenience or maybe a little less typing on the front end (with the potential for more on the backend if things change), that I wouldn't implement some kind of interface? A basic set of getters seems pretty straightforward. Maybe hyper tight performance considerations?

1 Reply

Alex Author

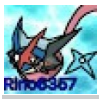
Reply to Claire [14] May 28, 2024 5:09 pm PDT

Trivial getters should be optimized away by the compiler, and therefore have no performance impact.

If a user should reasonably have access to the value of a member, a getter is fine. But there are cases where we don't want to allow this, particularly for things like the internal state of an object. Think about a PRNG, which has some state variables that get mixed up each time we generate a new random number. There's no value in letting the user see or alter these. However, if we were writing a function

that needed access to these, we might decide to implement it as a friend non-member function, so that single function could access the internal state without letting everyone do so.

6 Reply



Baker
May 4, 2024 6:55 am PDT

“When implementing a friend function, prefer to use the class interface over direct access whenever possible. This will help insulate your friend function from future implementation changes and lead to less code needing to be modified and/or retested later.
” you mean that friend functions should use access functions? Clarify please.

2 Reply

Alex Author
Reply to Baker [15] May 8, 2024 10:47 am PDT

Any function that is part of the public interface. This could be access functions, various overloaded operators, etc...

2 Reply

Phargelm
April 30, 2024 12:58 pm PDT

From the lesson 15.2:

> Unlike functions, which only need a forward declaration to be used, the compiler typically needs to see the full definition of a class (or any program-defined type) in order for the type to be used. This is because the compiler needs to understand how members are declared in order to ensure they are used properly, and it needs to be able to calculate how large objects of that type are in order to instantiate them. So our header files usually contain the full definition of a class rather than just a forward declaration of the class.

In this lesson we use only class declaration:

```
1 class Humidity; // forward declaration of Humidity
```

Is it because we don’t need to instantiate an object of this class when we encounter it for the first time? What is actually the rule when we are able to use only `class Humidity;` declaration and when we need the full definition of class from the header file? Before I've reached this lesson I thought that we always need to include header file with the corresponding class in order to reference it.

Last edited 1 year ago by Phargelm

0 Reply

Alex Author

Reply to Phargelm [16] April 30, 2024 5:16 pm PDT

When we forward declare a class, it is considered an incomplete type until the compiler has seen the full definition. The list of things we can do with an incomplete type is very limited. We can declare a pointer or reference to the incomplete class type, and we can friend the incomplete class type. Beyond that, there isn't a whole lot we can do, as the compiler doesn't know the size or details of how the class is implemented.

1 Reply

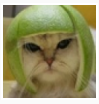
Phargelm

April 30, 2024 12:51 pm PDT

> But if the class is not ours (maybe it’s part of a third party library), we probably don’t want to modify the class (because if we update the library, those additions would be overwritten)

It seems that friends don’t solve the problem in this case as it requires adding changes to the third party class anyway in order to declare a friend function.

0 Reply



Alex Author

Reply to Phargelm [17] April 30, 2024 5:13 pm PDT

Good point. Sentence reworked. Thanks!

3 Reply

Yacine

March 23, 2024 7:38 am PDT

Hey Alex,

Thanks a lot for the valuable content you're delivering, I'm truly grateful to you for that.

Quick question, let's say we have a class that provides interfaces to access the data the we need, here, do we have cases where it's better to choose a go with a friend function over a non-friend function.

Last edited 1 year ago by Yacine

0 Reply

Alex Author

Reply to Yacine [18] March 24, 2024 1:43 pm PDT

Not that I can think of. Prefer non-member functions to friend functions if the interface allows.

1 Reply

IDontStopGoing
March 19, 2024 7:27 am PDT

i assume at this point (or maybe even earlier?) i could get started on making a 3D game (in a game engine) now just to play around with, but i have no idea how to start ... lol

How do games even start and run? Is it by some kind of class or something? or is there as many ways as there are stars?

Thank you for everything as always!
(im happy for all and any responses)

0 Reply

Alex Author
Reply to IDontStopGoing [19] March 19, 2024 2:41 pm PDT

Most games use some sort of framework (e.g. Unity, SFML). The game's `main()` typically initializes the framework and then kicks off an event loop of some kind.

https://www.sfml-dev.org/tutorials/2.6/window-window.php has an example of how this might be done for SFML.

2 Reply

IDontStopGoing
Reply to Alex [20] March 20, 2024 12:55 pm PDT

you're the best!
Thank you so much, i'll take a look at this, and still am loving your knowledge that you share here!
I'm so glad its taken me more than 3 months, because I honestly have loved every moment of it, and I dont ever want to leave xd

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/access-functions/
3. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/
4. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/static-member-functions/
7. https://www.learncpp.com/friend-non-member-functions/

8. https://www.learncpp.com/cpp-tutorial/introduction-to-operator-overloading/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-607356
11. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-606643
12. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-597985
13. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-598032
14. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-597619
15. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-596648
16. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-596479
17. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-596476
18. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-595017
19. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-594856
20. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/#comment-594867
21. https://g.ezoic.net/privacy/learncpp.com