15.8 — Friend non-member functions

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, 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 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). Even if the class is ours, we simply might not want to have certain things exposed to the general public.

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.

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:

```cpp
#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.

```
#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:

```
#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:

```
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 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.

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), 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:

```cpp
#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:

```
#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.