12.15 — Friend functions and classes

ALEX JULY 19, 2021

For much of this chapter, we've been preaching the virtues of keeping your data private. However, you may occasionally find situations where you will find you have classes and functions outside of those classes that need to work very closely together. For example, you might have a class that stores data, and a function (or another class) that displays the data on the screen. Although the storage class and display code have been separated for easier maintenance, the display code is really intimately tied to the details of the storage class. Consequently, there isn't much to gain by hiding the storage classes details from the display code.

In situations like this, there are two options:

1. Have the display code use the publicly exposed functions of the storage class. However, this has several potential downsides. First, these public member functions have to be defined, which takes time, and can clutter up the interface of the storage class. Second, the storage class may have to expose functions for the display code that it doesn't really want accessible to anybody else. There is no way to say "this function is meant to be used by the display class only".
2. Alternatively, using friend classes and friend functions, you can give your display code access to the private details of the storage class. This lets the display code directly access all the private members and functions of the storage class, while keeping everyone else out! In this lesson, we'll take a closer look at how this is done.

Friend functions

A friend function is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may be either a normal function, or a member function of another class. To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

Here's an example of using a friend function:

```cpp
class Accumulator
{
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Make the reset() function a friend of this class
    friend void reset(Accumulator& accumulator);
};

// reset() is now a friend of the Accumulator class
void reset(Accumulator& accumulator)
{
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}

int main()
{
    Accumulator acc;
    acc.add(5); // add 5 to the accumulator
    reset(acc); // reset the accumulator to 0

    return 0;
}
```

In this example, we've declared a function named reset() that takes an object of class Accumulator, and sets the value of m_value to 0. Because reset() is not a member of the Accumulator class, normally reset() would not be able to access the private members of Accumulator. However, because Accumulator has specifically declared this reset() function to be a friend of the class, the reset()

function is given access to the private members of Accumulator.

Note that we have to pass an Accumulator object to reset(). This is because reset() is not a member function. It does not have a *this pointer, nor does it have an Accumulator object to work with, unless given one.

Here’s another example:

```cpp
class Value
{
private:
    int m_value;
public:
    Value(int value) { m_value = value; }
    friend bool isEqual(const Value& value1, const Value&
value2);
};

bool isEqual(const Value& value1, const Value& value2)
{
    return (value1.m_value == value2.m_value);
}
```

In this example, we declare the isEqual() function to be a friend of the Value class. isEqual() takes two Value objects as parameters. Because isEqual() is a friend of the Value class, it can access the private members of all Value objects. In this case, it uses that access to do a comparison on the two objects, and returns true if they are equal.

While both of the above examples are fairly contrived, the latter example is very similar to cases we’ll encounter later when we discuss operator overloading!

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;

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

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

class Humidity
{
private:
    int m_humidity;
public:
    Humidity(int humidity=0) { 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 two things worth noting about this example. First, because printWeather is a friend of both classes, it can access the private data from objects of both classes. Second, note the following line at the top of the example:

```cpp
class Humidity;
```

This is a class prototype that tells the compiler that we are going to define a class called Humidity in the future. Without this line, the compiler would tell us it doesn't know what a Humidity is when parsing the prototype for printWeather() inside the Temperature class. Class prototypes serve the same role as function prototypes -- they tell the compiler what something looks like so it can be used now and defined later. However, unlike functions, classes have no return types or parameters, so class prototypes are always simply `class ClassName`, where ClassName is the name of the class.

Friend classes

It is also possible to make an entire class a friend of another class. This gives all of the members of the friend class access to the private members of the other class. Here is an example:

```cpp
#include <iostream>

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Make the Display class a friend of Storage
    friend class Display;
};

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(const Storage& storage)
    {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue <<
'\n';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue <<
'\n';
    }
};

int main()
{
    Storage storage(5, 6.7);
    Display display(false);

    display.displayItem(storage);

    return 0;
}
```

Because the Display class is a friend of Storage, any of Display's members that use a Storage class object can access the private members of Storage directly. This program produces the following result:

```
6.7 5
```

A few additional notes on friend classes. First, even though Display is a friend of Storage, Display has no direct access to the *this pointer of Storage objects. Second, just because Display is a friend of Storage, that does not mean Storage is also a friend of Display. If you want two classes to be friends of each other, both must declare the other as a friend. Finally, if class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

Be careful when using friend functions and classes, because it allows the friend function or class to violate encapsulation. If the details of the class change, the details of the friend will also be forced to change. Consequently, limit your use of friend functions and classes to a minimum.

Friend member functions

Instead of making an entire class a friend, you can make a single member function a friend. This is done similarly to making a normal function a friend, except using the name of the member function with the className:: prefix included (e.g. Display::displayItem).

However, in actuality, this can be a little trickier than expected. Let's convert the previous example to make Display::displayItem a friend member function. You might try something like this:

```cpp
class Display; // forward declaration for class Display

class Storage
{
private:
 int m_nValue;
 double m_dValue;
public:
 Storage(int nValue, double dValue)
 {
  m_nValue = nValue;
  m_dValue = dValue;
 }

 // Make the Display::displayItem member function a friend of the Storage class
 friend void Display::displayItem(const Storage& storage); // error: Storage hasn't seen the full
definition of class Display
};

class Display
{
private:
 bool m_displayIntFirst;

public:
 Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

 void displayItem(const Storage& storage)
 {
  if (m_displayIntFirst)
   std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
  else // display double first
   std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
 }
};
```

However, it turns out this won't work. In order to make a member function a friend, the compiler has to have seen the full definition for the class of the friend member function (not just a forward declaration). Since class Storage hasn't seen the full definition for class Display yet, the compiler will error at the point where we try to make the member function a friend.

Fortunately, this is easily resolved simply by moving the definition of class Display before the definition of class Storage.

```cpp
class Display
{
private:
 bool m_displayIntFirst;

public:
 Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

 void displayItem(const Storage& storage) // error: compiler doesn't know what a Storage is
 {
  if (m_displayIntFirst)
   std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
  else // display double first
   std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
 }
};

class Storage
{
private:
 int m_nValue;
 double m_dValue;
public:
 Storage(int nValue, double dValue)
 {
  m_nValue = nValue;
  m_dValue = dValue;
 }

 // Make the Display::displayItem member function a friend of the Storage class
 friend void Display::displayItem(const Storage& storage); // okay now
};
```

However, we now have another problem. Because member function Display::displayItem() uses Storage as a reference parameter, and we just moved the definition of Storage below the definition of Display, the compiler will complain it doesn't know what a Storage is. We can't fix this one by rearranging the definition order, because then we'll undo our previous fix.

Fortunately, this is also fixable in a couple of simple steps. First, we can add class Storage as a forward declaration. Second, we can move the definition of Display::displayItem() out of the class, after the full definition of Storage class.

Here's what this looks like:

```cpp
#include <iostream>

class Storage; // forward declaration for class Storage

class Display
{
private:
 bool m_displayIntFirst;

public:
 Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

 void displayItem(const Storage& storage); // forward declaration above needed for this declaration line
};

class Storage // full definition of Storage class
{
private:
 int m_nValue;
 double m_dValue;
public:
 Storage(int nValue, double dValue)
 {
  m_nValue = nValue;
  m_dValue = dValue;
 }

 // Make the Display::displayItem member function a friend of the Storage class (requires seeing the
full declaration of class Display, as above)
 friend void Display::displayItem(const Storage& storage);
};

// Now we can define Display::displayItem, which needs to have seen the full definition of class Storage
void Display::displayItem(const Storage& storage)
{
 if (m_displayIntFirst)
  std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
 else // display double first
  std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
}

int main()
{
    Storage storage(5, 6.7);
    Display display(false);

    display.displayItem(storage);

    return 0;
}
```

Now everything will compile properly: the forward declaration of class Storage is enough to satisfy the declaration of Display::displayItem(), the full definition of Display satisfies declaring Display::displayItem() as a friend of Storage, and the full definition of class Storage is enough to satisfy the definition of member function Display::displayItem(). If that's a bit confusing, see the comments in the program above.

If this seems like a pain -- it is. Fortunately, this dance is only necessary because we're trying to do everything in a single file. A better solution is to put each class definition in a separate header file, with the member function definitions in corresponding .cpp files. That way, all of the class definitions would have been visible immediately in the .cpp files, and no rearranging of classes or functions is necessary!

Summary

A friend function or class is a function or class that can access the private members of another class as though it were a member of that class. This allows the friend or class to work intimately with the other class, without making the other class expose its private members (e.g. via access functions).

Friending is commonly used when defining overloaded operators (which we'll cover in a later chapter), or less commonly, when two or more classes need to work together in an intimate way.

Note that making a specific member function a friend requires the full definition for the class of the member function to have been seen first.

Quiz time

1. **In geometry, a point is a position in space. We can define a point in 3d-space as the set of coordinates x, y, and z. For example, the Point(2.0, 1.0, 0.0) would be the point at coordinate space x=2.0, y=1.0, and z=0.0.**

In physics, a vector is a quantity that has a magnitude (length) and a direction (but no position). We can define a vector in 3d-space as an x, y, and z value representing the direction of the vector along the x, y, and z axis (the length can be derived from these). For example, the Vector(2.0, 0.0, 0.0) would be a vector representing a direction along the positive x-axis (only), with length 2.0.

A Vector can be applied to a Point to move the Point to a new position. This is done by adding the vector's direction to the point's position to yield a new position. For example, Point(2.0, 1.0, 0.0) + Vector(2.0, 0.0, 0.0) would yield the point (4.0, 1.0, 0.0).

Points and Vectors are often used in computer graphics (the point to represent vertices of shape, and vectors represent movement of the shape).

Given the following program:

```cpp
#include <iostream>

class Vector3d
{
private:
 double m_x{};
    double m_y{};
    double m_z{};

public:
 Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)
  : m_x{x}, m_y{y}, m_z{z}
 {

 }

 void print() const
 {
  std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z <<
")\n";
 }
};

class Point3d
{
private:
 double m_x{};
    double m_y{};
    double m_z{};

public:
 Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
  : m_x{x}, m_y{y}, m_z{z}
 {

 }

 void print() const
 {
  std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z <<
")\n";
 }

 void moveByVector(const Vector3d& v)
 {
  // implement this function as a friend of class Vector3d
 }
};

int main()
{
 Point3d p{1.0, 2.0, 3.0};
 Vector3d v{2.0, 2.0, -3.0};

 p.print();
 p.moveByVector(v);
 p.print();

 return 0;
}
```

1a) Make Point3d a friend class of Vector3d, and implement function Point3d::moveByVector()

Show Solution

1b) Instead of making class Point3d a friend of class Vector3d, make member function Point3d::moveByVector a friend of class Vector3d.

Show Solution

1c) Reimplement the solution to quiz question 1b using 5 separate files: Point3d.h, Point3d.cpp, Vector3d.h, Vector3d.cpp, and main.cpp.

Thanks to reader Shiva for the suggestion and solution.

Show Solution

Next lesson
12.16 Anonymous objects

Back to table of contents

Previous lesson
12.14 Static member functions

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X