15.9 — Friend classes and friend member functions

▲ ALEX SEPTEMBER 25, 2023

Friend classes

A **friend class** is a class that can access the private and protected members of another class.

Here is an example:

```
#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 }
    // Because Display is a friend of Storage, Display members can access the private members of Storage
    void displayStorage(const Storage& storage)
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';</pre>
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';</pre>
    }
    void setDisplayIntFirst(bool b)
         m_displayIntFirst = b;
};
int main()
    Storage storage { 5, 6.7 };
    Display display { false };
    display.displayStorage(storage);
    display.setDisplayIntFirst(true);
    display.displayStorage(storage);
    return 0:
```

Because the <code>Display</code> class is a friend of <code>Storage</code>, <code>Display</code> members can access the private members of any <code>Storage</code> object they have access to.



A few additional notes on friend classes.

First, even though <code>Display</code> is a friend of <code>Storage</code>, <code>Display</code> has no access to the *this pointer of <code>Storage</code> objects (because *this is actually a function parameter).

• • •

0

Second, friendship is not reciprocal. Just because Display is a friend of Storage 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.

Author's note

Sorry if this one hits a little close to home!

Class friendship is also not transitive. 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.

For advanced readers

Nor is friendship inherited. If class A makes B a friend, classes derived from B are not friends of A.

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 non-member function a friend, except the name of the member function is used instead.

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

```
#include <iostream>
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::displayStorage member function a friend of the Storage class
    friend void Display::displayStorage(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 displayStorage(const Storage& storage)
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';</pre>
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';</pre>
};
int main()
    Storage storage { 5, 6.7 };
    Display display { false };
    display.displayStorage(storage);
    return 0:
```

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.

• • •

0

Fortunately, this is easily resolved by moving the definition of class Display before the definition of class Storage (either in the same file, or by moving the definition of Display to a header file and #including it before Storage is defined).

```
#include <iostream>
class Display
private:
    bool m_displayIntFirst {};
public:
    Display(bool displayIntFirst)
        : m_displayIntFirst { displayIntFirst }
    }
    void displayStorage(const Storage& storage) // compile error: compiler doesn't know what a Storage is
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';</pre>
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';</pre>
};
class Storage
private:
    int m_nValue {};
    double m_dValue {};
    Storage(int nValue, double dValue)
        : m_nValue { nValue }, m_dValue { dValue }
    // Make the Display::displayStorage member function a friend of the Storage class
    friend void Display::displayStorage(const Storage& storage); // okay now
int main()
    Storage storage { 5, 6.7 };
    Display display { false };
    display.displayStorage(storage);
    return 0;
}
```

However, we now have another problem. Because member function <code>Display::displayStorage()</code> uses <code>Storage</code> as a reference parameter, and we just moved the definition of <code>Storage</code> below the definition of <code>Display</code>, the compiler will complain it doesn't know what a <code>Storage</code> 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 so the compiler will be okay with a reference to Storage before it has seen the full definition of the class.

Second, we can move the definition of <code>Display::displayStorage()</code> out of the class, after the full definition of <code>Storage</code> class.

Here's what this looks like:

. . .

```
#include <iostream>
class Storage; // forward declaration for class Storage
class Display
private:
    bool m_displayIntFirst {};
public:
    Display(bool displayIntFirst)
        : m_displayIntFirst { displayIntFirst }
    void displayStorage(const Storage& storage); // forward declaration for Storage needed for reference here
class Storage // full definition of Storage class
    int m_nValue {};
    double m_dValue {};
public:
    Storage(int nValue, double dValue)
        : m_nValue { nValue }, m_dValue { dValue }
    // Make the Display::displayStorage member function a friend of the Storage class
    // Requires seeing the full definition of class Display (as displayStorage is a member)
    friend void Display::displayStorage(const Storage& storage);
// Now we can define Display::displayStorage
// Requires seeing the full definition of class Storage (as we access Storage members)
void Display::displayStorage(const Storage& storage)
    if (m_displayIntFirst)
        std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';</pre>
    else // display double first
        std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';</pre>
int main()
    Storage storage { 5, 6.7 };
    Display display { false };
    display.displayStorage(storage);
    return 0:
```

Now everything will compile properly: the forward declaration of class Storage is enough to satisfy the declaration of Display::displayStorage() inside the Display class. The full definition of Display satisfies declaring Display::displayStorage() as a friend of Storage. And the full definition of class Storage is enough to satisfy the definition of member function Display::displayStorage().

If that's a bit confusing, see the comments in the program above. The key points are that a class forward declaration satisfies references to the class. However, accessing members of a class requires that the compiler have seen the full class definition.

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 be available in the .cpp files, and no rearranging of classes or functions is necessary!

Quiz time

Question #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, 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, Vector { 2.0, 0.0, 0.0 } would be a vector representing a direction along the positive x-axis (only), with length 2.0.

• • •

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 Point { 4.0, 1.0, 0.0 }.

Such points and vectors are often used in computer graphics (with points representing the vertices of a shape, and vectors represent movement of the shape).

Given the following program:

```
#include <iostream>
class Vector3d
private:
   double m_x{};
   double m_y{};
   double m_z{};
public:
   Vector3d(double x, double y, double z)
       : m_x{x}, m_y{y}, m_z{z}
   void print() const
       class Point3d
private:
   double m_x{};
   double m_y{};
   double m_z{};
public:
   Point3d(double x, double y, double z)
       : 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;
```

> Step #1

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

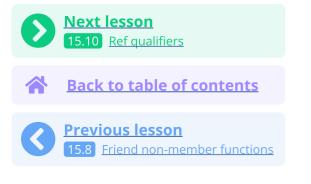
Show Solution (javascript:void(0))

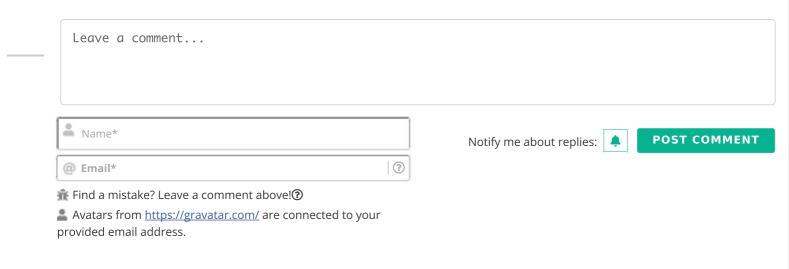
Instead of making class Point3d a friend of class Vector3d, make member function Point3d::moveByVector a friend of class Vector3d.

Show Solution (javascriptvoid(0))

> Step #3

Reimplement the solution to the prior step using 5 separate files: Point3d.h, Point3d.cpp, Vector3d.h, Vector3d.cpp, and main.cpp.





23 COMMENTS Newest ▼

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info:

OKAY

×