

## 8.13 — Friend functions and classes

BY ALEX ON SEPTEMBER 20TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 4TH, 2018

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:

```
1  class Accumulator
2  {
3  private:
4      int m_value;
5  public:
6      Accumulator() { m_value = 0; }
7      void add(int value) { m_value += value; }
8
9      // Make the reset() function a friend of this class
10     friend void reset(Accumulator &accumulator);
11 };
12
13 // reset() is now a friend of the Accumulator class
14 void reset(Accumulator &accumulator)
15 {
16     // And can access the private data of Accumulator objects
17     accumulator.m_value = 0;
18 }
19
20 int main()
21 {
22     Accumulator acc;
23     acc.add(5); // add 5 to the accumulator
24     reset(acc); // reset the accumulator to 0
25
26     return 0;
27 }
```

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:

```
1  class Value
2  {
3  private:
4      int m_value;
5  public:
6      Value(int value) { m_value = value; }
7      friend bool isEqual(const Value &value1, const Value &value2);
8  };
9
10 bool isEqual(const Value &value1, const Value &value2)
11 {
12     return (value1.m_value == value2.m_value);
13 }
```

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 in chapter 9 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:

```
1  class Humidity;
2
3  class Temperature
4  {
5  private:
6      int m_temp;
7  public:
8      Temperature(int temp=0) { m_temp = temp; }
9
10     friend void printWeather(const Temperature &temperature, const Humidity &humidity);
11 };
12
13 class Humidity
14 {
15 private:
16     int m_humidity;
17 public:
18     Humidity(int humidity=0) { m_humidity = humidity; }
19
20     friend void printWeather(const Temperature &temperature, const Humidity &humidity);
21 };
22
23 void printWeather(const Temperature &temperature, const Humidity &humidity)
24 {
25     std::cout << "The temperature is " << temperature.m_temp <<
26         " and the humidity is " << humidity.m_humidity << '\n';
27 }
28
29 int main()
30 {
31     Humidity hum(10);
32     Temperature temp(12);
33
34     printWeather(temp, hum);
35
36     return 0;
37 }
```

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:

```
1 | 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:

```
1 | class Storage
2 | {
3 | private:
4 |     int m_nValue;
5 |     double m_dValue;
6 | public:
7 |     Storage(int nValue, double dValue)
8 |     {
9 |         m_nValue = nValue;
10 |        m_dValue = dValue;
11 |    }
12 |
13 |    // Make the Display class a friend of Storage
14 |    friend class Display;
15 |};
16 |
17 | class Display
18 | {
19 | private:
20 |     bool m_displayIntFirst;
21 |
22 | public:
23 |     Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
24 |
25 |     void displayItem(Storage &storage)
26 |     {
27 |         if (m_displayIntFirst)
28 |             std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
29 |         else // display double first
30 |             std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
31 |     }
32 | };
33 |
34 | int main()
35 | {
36 |     Storage storage(5, 6.7);
37 |     Display display(false);
38 |
39 |     display.displayItem(storage);
40 |
41 |     return 0;
42 | }
```

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:

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:

```
1  class Display; // forward declaration for class Display
2
3  class Storage
4  {
5  private:
6      int m_nValue;
7      double m_dValue;
8  public:
9      Storage(int nValue, double dValue)
10     {
11         m_nValue = nValue;
12         m_dValue = dValue;
13     }
14
15     // Make the Display::displayItem member function a friend of the Storage class
16     friend void Display::displayItem(Storage& storage); // error: Storage hasn't seen the full definiti
17 on of class Display
18 };
19
20 class Display
21 {
22 private:
23     bool m_displayIntFirst;
24
25 public:
26     Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
27
28     void displayItem(Storage &storage)
29     {
30         if (m_displayIntFirst)
31             std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
32         else // display double first
33             std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
34     }
35 };
36
```

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.

```
1  class Display
2  {
3  private:
4      bool m_displayIntFirst;
5
6  public:
7      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
8
9  }
```

```

8
9     void displayItem(Storage &storage) // error: compiler doesn't know what a Storage is
10    {
11        if (m_displayIntFirst)
12            std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
13        else // display double first
14            std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
15    }
16 };
17
18 class Storage
19 {
20 private:
21     int m_nValue;
22     double m_dValue;
23 public:
24     Storage(int nValue, double dValue)
25     {
26         m_nValue = nValue;
27         m_dValue = dValue;
28     }
29
30     // Make the Display::displayItem member function a friend of the Storage class
31     friend void Display::displayItem(Storage& storage); // okay now
32 };

```

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:

```

1  class Storage; // forward declaration for class Storage
2
3  class Display
4  {
5  private:
6      bool m_displayIntFirst;
7
8  public:
9      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
10
11     void displayItem(Storage &storage); // forward declaration above needed for this declaration line
12 };
13
14 class Storage // full definition of Storage class
15 {
16 private:
17     int m_nValue;
18     double m_dValue;
19 public:
20     Storage(int nValue, double dValue)
21     {
22         m_nValue = nValue;
23         m_dValue = dValue;
24     }
25
26     // Make the Display::displayItem member function a friend of the Storage class (requires seeing the
27     // full declaration of class Display, as above)
28     friend void Display::displayItem(Storage& storage);
29 };
30
31 // Now we can define Display::displayItem, which needs to have seen the full declaration of class Storage
32

```

```

33 void Display::displayItem(Storage &storage)
34 {
35     if (m_displayIntFirst)
36         std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
37     else // display double first
38         std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
39 }
40
41 int main()
42 {
43     Storage storage(5, 6.7);
44     Display display(false);
45
46     display.displayItem(storage);
47
48     return 0;
49 }

```

Now everything will compile properly: the forward declaration of class `Storage` is enough to satisfy the declaration of `Display::displayItem()` inside the `Display` class, 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 uncommonly used when two or more classes need to work together in an intimate way, or much more commonly, when defining overloading operators (which we'll cover in chapter 9).

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:

```

1  #include <iostream>
2
3  class Vector3d
4  {
5  private:
6      double m_x, m_y, m_z;
7
8  public:
9      Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)

```

```

10     : m_x(x), m_y(y), m_z(z)
11     {
12     }
13     }
14
15     void print()
16     {
17         std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
18     }
19 };
20
21 class Point3d
22 {
23 private:
24     double m_x, m_y, m_z;
25
26 public:
27     Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
28         : m_x(x), m_y(y), m_z(z)
29     {
30     }
31
32     void print()
33     {
34         std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
35     }
36
37     void moveByVector(const Vector3d &v)
38     {
39         // implement this function as a friend of class Vector3d
40     }
41 }
42 };
43
44 int main()
45 {
46     Point3d p(1.0, 2.0, 3.0);
47     Vector3d v(2.0, 2.0, -3.0);
48
49     p.print();
50     p.moveByVector(v);
51     p.print();
52
53     return 0;
54 }

```

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

#### Hide Solution

```

1     #include <iostream>
2
3     class Vector3d
4     {
5     private:
6         double m_x, m_y, m_z;
7
8     public:
9         Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)
10             : m_x(x), m_y(y), m_z(z)
11         {
12         }
13
14         void print()
15         {
16             std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";

```

```

18     }
19
20     friend class Point3d; // Point3d is now a friend of class Vector3d
21 };
22
23
24 class Point3d
25 {
26 private:
27     double m_x, m_y, m_z;
28
29 public:
30     Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
31         : m_x(x), m_y(y), m_z(z)
32     {
33
34     }
35
36     void print()
37     {
38         std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
39     }
40
41
42     void moveByVector(const Vector3d &v)
43     {
44         m_x += v.m_x;
45         m_y += v.m_y;
46         m_z += v.m_z;
47     }
48 };
49
50
51 int main()
52 {
53     Point3d p(1.0, 2.0, 3.0);
54     Vector3d v(2.0, 2.0, -3.0);
55
56     p.print();
57     p.moveByVector(v);
58     p.print();
59
60     return 0;
61 }

```

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

### **Hide Solution**

```

1  class Vector3d; // first, we need to tell the compiler that a class named Vector3d exists
2
3  class Point3d
4  {
5  private:
6      double m_x, m_y, m_z;
7
8  public:
9      Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
10         : m_x(x), m_y(y), m_z(z)
11     {
12
13     }
14
15     void print()
16     {
17         std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";

```



```

18     }
19
20     void moveByVector(const Vector3d &v); // so we can use Vector3d here
21     // note: we can't define this function here, because Vector3d hasn't been declared yet (just for
22     ward declared)
23 };
24
25 class Vector3d
26 {
27 private:
28     double m_x, m_y, m_z;
29
30 public:
31     Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)
32         : m_x(x), m_y(y), m_z(z)
33     {
34
35     }
36
37     void print()
38     {
39         std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
40     }
41
42     friend void Point3d::moveByVector(const Vector3d &v); // Point3d::moveByVector() is now a friend of
43     class Vector3d
44 };
45
46 // Now that Vector3d has been declared, we can define the function Point3d::moveByVector()
47 void Point3d::moveByVector(const Vector3d &v)
48 {
49     m_x += v.m_x;
50     m_y += v.m_y;
51     m_z += v.m_z;
52 }
53
54 int main()
55 {
56     Point3d p(1.0, 2.0, 3.0);
57     Vector3d v(2.0, 2.0, -3.0);
58
59     p.print();
60     p.moveByVector(v);
61     p.print();
62
63     return 0;
64 }

```

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.

### Hide Solution

Point3d.h:

```

1 // Header file that defines the Point3d class
2
3 #ifndef POINT3D_H
4 #define POINT3D_H
5
6 class Vector3d; // forward declaration for class Vector3d for function moveByVector()
7
8 class Point3d
9 {
10     private:

```

```

11     double m_x;
12     double m_y;
13     double m_z;
14
15     public:
16     Point3d(double x = 0.0, double y = 0.0, double z = 0.0) : m_x(x), m_y(y), m_z(z) {}
17
18     void print();
19     void moveByVector(const Vector3d &v); // forward declaration above needed for this line
20 };
21
22 #endif

```

Point3d.cpp:

```

1 // Member functions of the Point3d class defined here
2
3 #include <iostream> // for std::cout
4 #include "Point3d.h" // Point3d class declared in this
5 #include "Vector3d.h" // for the parameter of the function moveByVector()
6
7 void Point3d::moveByVector(const Vector3d &v)
8 {
9     // Add the vector components to the corresponding point coordinates
10    m_x += v.m_x;
11    m_y += v.m_y;
12    m_z += v.m_z;
13 }
14
15 void Point3d::print()
16 {
17     std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
18 }

```

Vector3d.h:

```

1 // Header file that defines the Vector3d class
2
3 #ifndef VECTOR3D_H
4 #define VECTOR3D_H
5
6 #include "Point3d.h" // for declaring Point3d::moveByVector() as a friend
7
8 class Vector3d
9 {
10     private:
11         double m_x;
12         double m_y;
13         double m_z;
14
15     public:
16         Vector3d(double x = 0.0, double y = 0.0, double z = 0.0) : m_x(x), m_y(y), m_z(z) {}
17
18         void print();
19         friend void Point3d::moveByVector(const Vector3d &v);
20 };
21
22 #endif

```

Vector3d.cpp:

```

1 // Member functions of the Vector3d class defined here
2
3 #include <iostream>
4 #include "Vector3d.h" // Vector3d class declared in this file
5
6 void Vector3d::print()
7 {

```

```
8   std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
9 }
```

main.cpp:

```
1  #include "Vector3d.h" // for creating Vector3d object
2  #include "Point3d.h" // for creating Point3d object
3
4  int main()
5  {
6      Point3d p(1.0, 2.0, 3.0);
7      Vector3d v(2.0, 2.0, -3.0);
8
9      p.print();
10     p.moveByVector(v);
11     p.print();
12
13     return 0;
14 }
```



[8.14 -- Anonymous objects](#)



[Index](#)



[8.12 -- Static member functions](#)

Share this:



Facebook



Twitter



G+ Google



Pinterest

[C++ TUTORIAL](#) | [PRINT THIS POST](#)

## 207 comments to 8.13 — Friend functions and classes

[« Older Comments](#) [1](#) [2](#) [3](#)



Benjamin Collins

[June 18, 2018 at 3:44 pm](#) · [Reply](#)

Friend classes? More like friend classes with benefits, since they give each other access to their privates ; )

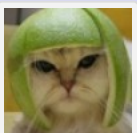


DecSco

[June 15, 2018 at 12:40 am](#) · [Reply](#)

I'm using Code::Blocks as IDE and when trying to find out why my program doesn't compile, I found that the ticks for "Compile file" and "Link file" are not set for header files (and indeed, they mustn't be).

Why is that so? I couldn't find an answer elsewhere online.



Alex

[June 18, 2018 at 3:58 pm](#) · [Reply](#)