15.9 — Friend classes and friend member functions

ALEX[1] APRIL 30, 2024

Friend classes

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

    // 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';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }

    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 `Display` class is a friend of `Storage`, `Display` members can access the private members of any `Storage` object they have access to.

This program produces the following result:

```
6.7 5
5 6.7
```

A few additional notes on friend classes.

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

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.

A friend class declaration acts as a forward declaration for the class being friended. This means we do not need to forward declare the class being friended before friending it. In the example above, `friend class Display` acts as both a forward declaration of `Display` and a friend declaration.

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 `Display::displayStorage` a friend member function. You might try something like this:

```cpp
#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';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }
};

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 single 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 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).

```cpp
#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';
        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::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 `Display::displayStorage()` 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 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 `Display::displayStorage()` 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 displayStorage(const Storage& storage); // forward declaration for Storage needed for reference here
};

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::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';
    else // display double first
        std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
}

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.

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:

```cpp
#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
    {
        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, 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))[2]

> Step #2

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

Show Solution (javascript:void(0))[2]

> Step #3

Reimplement the solution to the prior step 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 (javascript:void(0))[2]

Next lesson
15.10 Ref qualifiers

3

Back to table of contents

4

Previous lesson
15.8 Friend non-member functions

5

6

Junk
July 2, 2025 5:14 pm PDT

The point and vector examples for this are throwing me off because vectors act as points in computer graphics anyway, so the whole friend example is a bit contrived lol

Last edited 2 hours ago by Junk

0 Reply

Mr. F
June 7, 2025 9:42 am PDT

good heavens...

0 Reply

Copernicus
June 5, 2025 5:23 am PDT

Question #3
Main.cpp

```cpp
#include "Point3d.h"
#include "Vector3d.h"

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;
}
```

Point3d.h

```cpp
#pragma once

#include <iostream>
#include "Vector3d.h"

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

public:
    Point3d(double x, double y, double z);
    void print() const;
    void moveByVector(const Vector3d& v);
};
```

Point3d.cpp

```cpp
#include <iostream>
#include "Point3d.h"

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

void Point3d::print() const
{
    std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")\n";
}

void Point3d::moveByVector(const Vector3d& v)
{
    m_x += v.getX();
    m_y += v.getY();
    m_z += v.getZ();
}
```

Vector3d.h

```cpp
#pragma once

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

public:
    Vector3d(double x, double y, double z);
    void print() const;
    double getX() const;
    double getY() const;
    double getZ() const;
};
```

Vector3d.cpp

```cpp
#include <iostream>
#include "Vector3d.h"

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

void Vector3d::print() const
{
    std::cout << "Vector(" << m_x << ", " << m_y << ", " << m_z << ")\n";
}
double Vector3d::getX() const
{
    return m_x;
}
double Vector3d::getY() const
{
    return m_y;
}
double Vector3d::getZ() const
{
    return m_z;
}
```

0 Reply

Copernicus

June 5, 2025 4:48 am PDT

Question #2

```cpp
#include <iostream>

class Vector3d;

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);
};

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

    friend void Point3d::moveByVector(const Vector3d& v);
};

void Point3d::moveByVector(const Vector3d& v)
{
    m_x += v.m_x;
    m_y += v.m_y;
    m_z += v.m_z;
}

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;
}
```

Copernicus

June 5, 2025 4:39 am PDT

Question #1

```cpp
#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
    {
        std::cout << "Vector(" << m_x << ", " << m_y << ", " << m_z << ")\n";
    }

    friend class Point3d;
};

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)
    {
        m_x += v.m_x;
        m_y += v.m_y;
        m_z += v.m_z;
    }
};

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;
}
```

Kyle
May 17, 2025 9:12 pm PDT

Hey Alex, I was wondering what you believe is the best way to go about creating the 5-files answer, but from scratch. Would you normally build out the classes in main.cpp, test it, then separate it out from there? Or is it more practical to write it all separately to begin with?

Also, does the order in which you #include multiple headers files(like in Point3d.cpp) matter? Both orders compiled an ran fine for me, but I think I'm developing 'Error PTSD' and no longer trust anything I'm doing anymore.

Side note: I'm was pulling my hair out trying to interpret 10 seemingly unrelated errors. I really wish VS had a more elegant way of saying "You forgot to put 'const' here". Though, I must say...it sure is glorious when a minor fix like that clears out every remaining error!

Last edited 1 month ago by Kyle

0 Reply

paulisprouki
Reply to Kyle [10] June 22, 2025 1:13 pm PDT

I would start building from the headers and build along with the cpp files that use it respectively, after im done with the file or a section of it im going to make the connection and logic in main. This is a bit annoying to keep up but it gets easier with time.

0 Reply

smart
May 14, 2025 9:26 am PDT

I've started immediately clicking "Show Solution" whenever something doesn't work, and I don't know how to make myself actually search for the solution on my own and stop being lazy...

0 Reply

aki
Reply to smart [11] May 19, 2025 12:00 pm PDT

This is normal at the very beginning however a good rule of thumb is to only look at the answer if you haven't made any progress in 45 min of serious attempts, the only way one can truly learn is through struggle. Here are some things I've found useful:

- While reading the lessons, boot up your IDE of choice and try doing things with the current topic you are learning OUTSIDE of the examples shown, many times you will find that even though you read the lesson, you won't be able to implement it in your first attempt without forgetting something essential, refer back to the lesson and see what you missed. It can be overwhelming if you go straight to the quiz without implementing some trivial examples yourself.
- If something doesn't work, use the debugger and find out why, is the program behaving as you expect? At what point does SHTF? Using the IDE's debugger is EXTREMELY useful don't miss out on how to use it

- Just move code around, do random things that you think might work even if they sound dumb in your head, ask yourself, why does this adjustment work/not work?

It's good that your getting to the point of something not working because **that is when an opportunity to learn and become a better programmer is presenting itself, take advantage of it and struggle**, Alex did not come out of the womb with unbelievable programming knowledge like he has now, even he at one point had to struggle like we are now to get to the point that he is today. **Never give in to seeing the solution right without learning because you will never be able improve**, Remember, programming is not about memorizing solutions but about becoming a better problem solver, the former you will forget but the latter you will keep for the rest of your life.

thanks for reading if you made it this far and i wish you luck on your journey, aki

Last edited 1 month ago by aki

2 Reply

smart

Reply to aki [12] May 19, 2025 5:00 pm PDT

Thank you for the support :) I used to be able to spend an entire day, even longer, working through those quizzes, but lately I've been getting constant headaches, and other obligations leave me with very little time, which is really pressing on me. I know it would be better not to rush, but right now I only have the energy to read through lessons and study the quiz solutions. Someday I'll come back and solve them all myself, although of course you won't get that great memorization boost you get when you struggle to find the answer on your own.

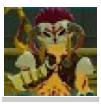
0 Reply

vitrums

May 3, 2025 4:35 am PDT

In step 3 moving one-liner definitions out of the respective headers stinks.

1 Reply



sebby

April 1, 2025 2:33 pm PDT

Point3d.h

```
#ifndef POINT_3D_H
#define POINT_3D_H
#include <iostream>

class Vector3d;
class Point3d {
    double m_x{};
    double m_y{};
    double m_z{};

  public:
    Point3d(double x, double y, double z);
    void print() const;
    void moveByVector(const Vector3d& v);
};

#endif // POINT_3D_H
```

Point3d.cpp

```
#include "Point3d.h"
#include "Vector3d.h"
#include <iostream>

Point3d::Point3d(double x, double y, double z) : m_x{x}, m_y{y}, m_z{z} {
}
void Point3d::print() const {
    std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")\n";
}
void Point3d::moveByVector(const Vector3d& v) {
    m_x += v.m_x;
    m_y += v.m_y;
    m_z += v.m_z;
}
```

Vector3d.h

```
#ifndef VECTOR_3D_H
#define VECTOR_3D_H
#include "Point3d.h"
#include <iostream>

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

  public:
    Vector3d(double x, double y, double z);
    void print() const;
    friend void Point3d::moveByVector(const Vector3d& v);
};

#endif // VECTOR_3D_H
```

Vector3d.cpp

```cpp
#include "Vector3d.h"
#include <iostream>


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

void Vector3d::print() const {
    std::cout << "Vector(" << m_x << ", " << m_y << ", " << m_z << ")\n";
}
```

main.cpp

```cpp
#include "Point3d.h"
#include "Vector3d.h"

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;
}
```

Last edited 3 months ago by sebby

0 Reply

Dinny

March 12, 2025 5:05 am PDT

I made the mistake of adding '#include "Vector3d.h"' in Point3d.h. It didn't compile and took a long time to figure out what's going on. I don't really understand why. Originally, I had wanted to put all the #include into the header file. I guess I need to reread the header file lesson....

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/ref-qualifiers/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/
6. https://www.learncpp.com/friend-classes-and-friend-member-functions/

7. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/
8. https://www.learncpp.com/cpp-tutorial/chapter-15-summary-and-quiz/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/#comment-610146
11. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/#comment-610055
12. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/#comment-610204
13. https://g.ezoic.net/privacy/learncpp.com