Home Articles

C++ Template Classes and Friend Function Details

Introduction

When writing C++, there are a couple common situations where you may want to create a friend function, such as implementing `operator<<`. These are straightforward to write for non-template classes, but templates throw a real wrench in the works here. In C++, there are a few different ways to set up template classes with friend functions. Each behaves in a subtly different way, which causes all sorts of confusion when trying to finish your homework the night before it's due. We will examine the possible approaches to this, but first...

Some terminology

- **Template instantiation**: A specific instance of a templated item; for example, if we have a templated class `A`, `A<int>` is a specific instantiation of `A`.
- **Non-member function**: A function that is not a member of a class. We will examine these in the context of friend functions in this article.

Approach #1

```
template<class T>
class A
{
  public:
    A(T a = 0): m_a(a) {}

    template<class U>
    friend A<U> foo(A<U>& a);

  private:
    T m_a;
};

template<class T>
A<T> foo(A<T>& a)
{
  return a;
}
```

This approach is the most permissive approach to declaring a templated friend function of a templated class. This will work for all cases where you want to explicitly pass an instance of class `A` to `foo()`.

However, there is one unapparent side effect of this approach: All template instantiations of `A` are friends with all template instantiations of `foo()`. So, for example, `A<int>` is friends with `foo<int>` but also `foo<double>`. To see why this is a Bad Thing, consider the following perfectly valid C++:

```
19 A<double> secret_pie(3.14);
20
21 struct dummy {};
22 template<>
23 A<dummy> foo<dummy>(A<dummy>& d)
24 {
25   cout << "Hacked! " << secret_pie.m_a << endl;
26   return d;
27 }
```

In an ideal world, line 25 should not work, since it is in a function that by all rights should know nothing about `A<double>`! To prevent this potential programmer error, we must take a different approach to generate more antisocial friend functions.

Approach #2

```
 1 template<class T>
 2 class A;
 3
 4 template<class T>
 5 A<T> foo(A<T>& a);
 6
 7 template<class T>
 8 class A
 9 {
10   public:
11     A(T a = 0): m_a(a) {}
12
13     friend A foo<T>(A& a);
14
15   private:
16     T m_a;
17 };
18
19 template<class T>
20 A<T> foo(A<T>& a)
21 {
22   return a;
23 }
```

This approach is a bit more ugly, but it fixes the issue with approach #1. Here, `foo()` is declared as a template function using a declared (but not yet defined) templated class `A`. Then, when we define `A`, we make each template instantiation of `A` friends with the corresponding template instantiation of `foo()`. (In case you're wondering why on line 13 A is used with no `<T>` after it: inside class definitions, C++ assumes that any reference to that class is templated, so adding the `<T>` is redundant.)

This may seem like a minor difference from approach #1, but in this case `A<int>` is friends with `foo<int>`, but `foo<double>` is not a friend of `A<int>` since the type in their template parameters do not match.

In general, this is how template friend functions are done in best practices. They are explicit in the type of objects they will take as parameters and do not allow different template instantiations to be friends.

However, in certain edge cases, such as numeric type objects, these friend functions have an undesirable side effect: since they are template functions, the parameters passed to them must be explicitly associated with the expected parameter types for template argument deduction to succeed. To illustrate,

```
 1 template<class T>
 2 class A;
 3
 4 template<class T>
 5 A<T> foo(A<T>& a);
 6
 7 template<class T>
 8 class A
 9 {
10   public:
11     A(T a = 0): m_a(a) {}
12
13     friend A foo<T>(A& a);
14
15   private:
16     T m_a;
17 };
18
19 template<class T>
20 A<T> foo(A<T>& a)
21 {
22   return a;
23 }
24
25 int main()
26 {
27   A<int> a(5);
28   int i = 4;
29
30   foo(a); // Succeeds
31   foo(i); // Cannot deduce template parameters
32
33   return 0;
34 }
```

On line 31, despite there being an implicit conversion from `int` to `A<int>`, that conversion is not made. This is because C++ never considers implicit conversions for template parameter deduction on non-member template functions. For the most part, this is not a problem because it prevents certain amounts of ambiguity when dealing with calling template functions, but in the case where we are working with numeric type classes, we do want the implicit conversion to occur.

Approach #3

As we saw in approach #2, C++ won't implicitly convert types for template function parameters, even if such a conversion exists. But in the case where we are working with numeric types, we want that conversion to occur. So, how can we achieve this? Well, C++ will perform implicit type conversion on non-templated non-member functions. What if we were to create some non-template friend function that was automatically created for each template instantiation of some class? As it turns out, that is possible:

```cpp
template<class T>
class A
{
  public:
    A(T a = 0): m_a(a) {}

    friend A operator+(const A& lhs, const A& rhs)
    {
      return lhs.m_a + rhs.m_a;
    }

  private:
    T m_a;
};

int main()
{
  A<int> a(5);
  A<int> b(7);
  int i = 4;

  a + b;    // Succeeds
  i + a;    // Also succeeds
  return 0;
}
```

Here, `operator+` is a friend function, but is not templated. Instead, it is a function that is automatically created for each template instantiation of `A` when that template is instantiated. The generic definition must be inlined in the class definition because `operator+` doesn't exist outside the class definition of `A` until a template instantiation of `A` is generated, which happens during the compilation process.

As an aside, it is possible to write specific definitions of `operator+` outside of A, but no generic definition can be written because operator+ is not a template function:

```cpp
template<class T>
class A
{
  public:
    A(T a = 0): m_a(a) {}

    friend A operator+(const A& lhs, const A& rhs);

  private:
    T m_a;
};

inline A<int> operator+(const A<int>& lhs, const A<int>& rhs)
{
  return lhs.m_a + rhs.m_a;
}

int main()
{
  A<int> a(5);
  A<int> b(7);
  int i = 4;

  a + b;    // Succeeds
  i + a;    // Also succeeds
  return 0;
}
```

However, this code won't compile without warnings in GCC, and is uninteresting anyway because we don't want to write specific implementations of `operator+` for each template instantiation of `A` (otherwise, why bother with templates to begin with?). The warning itself indicates that this is possible, but not usually what you want to do.

Now, you may be wondering why I switched my example function defnition. As it turns out, this sort of non-template friend function is unusual since the function is neither in the global scope nor a member of class `A`. This means that our function doesn't live in the "standard" places you'd expect it to; neither `::operator+` nor `A<int>::operator+` exist. Fortunately, C++ has a feature called ADL, or Argument Dependent Lookup, that can search through functions that aren't in the current scope, but exist in a class or namespace that is suggested by the type of the arguments handed to the function call:

```
 1 template<class T>
 2 class A
 3 {
 4   public:
 5     A(T a = 0): m_a(a) {}
 6 
 7     friend A operator+(const A& lhs, const A& rhs)
 8     {
 9       return lhs.m_a + rhs.m_a;
10     }
11 
12     friend A foo(const A& a)
13     {
14       return a;
15     }
16 
17   private:
18     T m_a;
19 };
20 
21 int main()
22 {
23   A<int> a(5);
24   A<int> b(7);
25   int i = 4;
26 
27   a + b;    // Succeeds
28   i + a;    // Also succeeds
29 
30   foo(a);   // Succeeds
31   foo(i);   // Fails: foo not declared in this scope
32   foo(A<int>(i)); // Succeeds
33   return 0;
34 }
```

Here, line 28 succeeds because of ADL: even though `operator+` is not in the current scope, C++ can see that the second argument to `operator+` is of type `A<int>` and therefore will attempt to implicitly convert `i` to `A<int>` because `operator+` is not a template function and thus follows standard implicit conversion rules for non-template non-member functions. Since C++ does have a way to convert an `int` to a `A<int>`, the call succeeds.

However, the call on line 31 fails because `foo()` is not in the current scope, and no parameter passed to `foo()` hints that C++ needs to use ADL to consider functions not in the current scope.

Line 32 succeeds because the parameter to `foo()` is explicitly of type `A<int>`, and thus C++ uses ADL to look up `foo()`.

This issue with ADL not always looking up a function when it exists is reason to not use this approach with most classes and instead explicitly convert arguments to the correct types when calling a templated friend function. And, if you're already explicitly converting types, you may as well use approach #2 and lose all the complexity of approach #3.

But, in the case where you require implicit type conversions on friend functions of template classes and can guarantee that ADL will succeed in your function calls, approach #3 will work, and in the worst case it is no more difficult to use than approach #2.

Conclusion

C++ has three approaches for implementing friend functions of template classes. The first approach, making each template instantiation of a function friends with each template instantiation of a class, has some unfortunate consequences for object-oriented encapsulation best practices. The second approach explicitly creates both a template class and a template function and makes each instantiation of the class friends with the corresponding instantiation of the function. This approach is generally what people want when they think of making a template function friends with a template class. Unfortunately, however, template functions do not perform implicit conversions on parameters passed to them. The third approach solves this problem by relying on C++'s template instantiation mechanism to generate a non-template friend function associated with each instantiation of a template class. However, calling this function depends on ADL to find that function, so compiler errors may be odd ("function not declared in this scope" rather than "cannot convert argument").

Of course, you should ask yourself whether you need a friend function to begin with! For example, if you implement `operator+=` as a member function, you can write a templated `operator+` function that just calls your public `operator+=` function and thus does not need to be a friend function in the first place. (However, be aware that this will not perform implicit conversions; you must use approach #3 if you want those.)

Overall, if you need implicit conversions from other types to instances of a template class, you should take approach #3. Otherwise, consider whether you can implement your non-member function by calling public member functions of the template class; if you can, you don't need a friend function at all. If you cannot, then take approach #2.

Sources and Further Reading

- Effective C++, Scott Meyers; particularly Items #24 and #46 (available via Safari Books)
- C++ Templates: The Complete Guide, David Vandervoorde, Nicolai M. Josuttis; pages 174-177
- The three different approaches to template class friend functions
- More on approaches to template class friend functions
- A few more details on the third approach
- ADL for template parameter deduction and shortcomings of ADL
- Wikipedia page for ADL