

Template Friends

01/03/2018 • 4 minutes to read


In this article

- [Example](#)
- [Example](#)
- [Example](#)
- [See Also](#)

The latest version of this topic can be found at [Template Friends](#).

Class templates can have [friends](#). A class or class template, function, or function template can be a friend to a template class. Friends can also be specializations of a class template or function template, but not partial specializations.


C++ 11: A type parameter can be declared as a friend by using the form `friend T; .`

 Copy

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Example

In the following example, a friend function is defined as a function template within the class template. This code produces a version of the friend function for every instantiation of the template. This construct is useful if your friend function depends on the same template parameters as the class does.

 Copy

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;
```

```
template <class T> class Array {
    T* array;
    int size;

public:
    Array(int sz): size(sz) {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }

    Array(const Array& a) {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }

    T& operator[](int i) {
        return *(array + i);
    }

    int Length() { return size; }

    void print() {
        for (int i = 0; i < size; i++)
            cout << *(array + i) << " ";

        cout << endl;
    }

    template<class T>
    friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
    Array<T>* a = new Array<T>(a1.size + a2.size);
    for (int i = 0; i < a1.size; i++)
        (*a)[i] = *(a1.array + i);

    for (int i = 0; i < a2.size; i++)
        (*a)[i + a1.size] = *(a2.array + i);

    return a;
}

int main() {
    Array<char> alpha1(26);
    for (int i = 0 ; i < alpha1.Length() ; i++)
        alpha1[i] = 'A' + i;

    alpha1.print();
}
```

```
Array<char> alpha2(26);
for (int i = 0 ; i < alpha2.Length() ; i++)
    alpha2[i] = 'a' + i;

alpha2.print();
Array<char>*alpha3 = combine(alpha1, alpha2);
alpha3->print();
delete alpha3;
}
```

 Copy

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Example

The next example involves a friend that has a template specialization. A function template specialization is automatically a friend if the original function template is a friend.

It is also possible to declare only the specialized version of the template as the friend, as the comment before the friend declaration in the following code indicates. If you do this, you must put the definition of the friend template specialization outside of the template class.

```
// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
    }
};
```

 Copy

```
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {
        return size;
    }
    void print()
    {
        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
```

```
f(ac);

Array<int> a(10);
f(a);
}
```

Copy

10 generic
10 int

Example

The next example shows a friend class template declared within a class template. The class template is then used as the template argument for the friend class. Friend class templates must be defined outside of the class template in which they are declared. Any specializations or partial specializations of the friend template are also friends of the original class template.

Copy

```
// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
private:
    T* data;
    void InitData(int seed) { data = new T(seed); }
public:
    void print() { cout << *data << endl; }
    template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
    U* GetNewObject(int seed)
    {
        U* pu = new U;
        pu->InitData(seed);
        return pu;
    }
};
```

```
int main()
{
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
```

 Copy

65
97
A
a

See Also

[Default Arguments](#)