friend declaration

The friend declaration appears in a class body and grants a function or another class access to private and protected members of the class where the friend declaration appears.

Syntax

`friend` *function-declaration*	(1)	
`friend` *function-definition*	(2)	
`friend` *elaborated-class-specifier* `;`	(3)	
`friend` *simple-type-specifier* `;` `friend` *typename-specifier* `;`	(4)	(since C++11)

Description

1) Designates a function or several functions as friends of this class

```
class Y {
    int data; // private member
    // the non-member function operator<< will have access to Y's private members
    friend std::ostream& operator<<(std::ostream& out, const Y& o);
    friend char* X::foo(int); // members of other classes can be friends too
    friend X::X(char), X::~X(); // constructors and destructors can be friends
};
// friend declaration does not declare a member function
// this operator<< still needs to be defined, as a non-member
std::ostream& operator<<(std::ostream& out, const Y& y)
{
    return out << y.data; // can access private member Y::data
}
```

2) (only allowed in non-local class definitions) Defines a non-member function, and makes it a friend of this class at the same time. Such non-member function is always inline.

```
class X {
    int a;
    friend void friend_set(X& p, int i) {
        p.a = i; // this is a non-member function
    }
 public:
    void member_set(int i) {
        a = i; // this is a member function
    }
};
```

3) Designates the class, struct, or union named by the *elaborated-class-specifier* (see elaborated type specifier) as a friend of this class. This means that the friend's member declarations and definitions can access private and protected members of this class and also that the friend can inherit from private and protected members of this class. The name of the class that is used in this friend declaration does not need to be previously declared.

4) Designates the type named by the *simple-type-specifier* or *typename-specifier* as a friend of this class if that type is a (possibly cv-qualified) class, struct, or union; otherwise the `friend` declaration is ignored. This declaration will not forward declare new type.

```
class Y {};
class A {
    int data; // private data member
    class B { }; // private nested type
    enum { a = 100 }; // private enumerator
    friend class X; // friend class forward declaration (elaborated class specifier)
    friend Y; // friend class declaration (simple type specifier) (since c++11)
```

```
};

class X : A::B { // OK: A::B accessible to friend
    A::B mx; // OK: A::B accessible to member of friend
    class Y {
        A::B my; // OK: A::B accessible to nested member of friend
    };
    int v[A::a]; // OK: A::a accessible to member of friend
};
```

Notes

Friendship is not transitive (a friend of your friend is not your friend)

Friendship is not inherited (your friend's children are not your friends)

Storage class specifiers are not allowed in friend function declarations. A function that is defined in the friend declaration has external linkage, a function that was previously defined, keeps the linkage it was defined with.

Access specifiers have no effect on the meaning of friend declarations (they can appear in `private:` or in `public:` sections, with no difference)

A friend class declaration cannot define a new class (`friend class X {};` is an error)

When a local class declares an unqualified function or class as a friend, only functions and classes in the innermost non-class scope are looked up, not the global functions:

```
class F {};
int f();
int main()
{
    extern int g();
    class Local { // Local class in the main() function
        friend int f(); // Error, no such function declared in main()
        friend int g(); // OK, there is a declaration for g in main()
        friend class F; // friends a local F (defined later)
        friend class ::F; // friends the global F
    };
    class F {}; // local F
}
```

A name first declared in a friend declaration within class or class template X becomes a member of the innermost enclosing namespace of X, but is not visible for lookup (except argument-dependent lookup that considers X) unless a matching declaration at the namespace scope is provided - see namespaces for details.

Template friends

Both function template and class template declarations may appear with the `friend` specifier in any non-local class or class template (although only function templates may be defined within the class or class template that is granting friendship). In this case, every specialization of the template becomes a friend, whether it is implicitly instantiated, partially specialized, or explicitly specialized.

```
class A {
    template<typename T>
    friend class B; // every B<T> is a friend of A

    template<typename T>
    friend void f(T) {} // every f<T> is a friend of A
};
```

Friend declarations cannot refer to partial specializations, but can refer to full specializations:

```
template<class T> class A {}; // primary
template<class T> class A<T*> {}; // partial
template<> class A<int> {}; // full
class X {
    template<class T> friend class A<T*>; // error!
    friend class A<int>; // OK
};
```

When a friend declaration refers to a full specialization of a function template, the keyword `inline` and default arguments cannot be used.

```
template<class T> void f(int);
template<> void f<int>(int);

class X {
    friend void f<int>(int x = 1); // error: default args not allowed
};
```

A template friend declaration can name a member of a class template A, which can be either a member function or a member type (the type must use elaborated-type-specifier). Such declaration is only well-formed if the last component in its nested-name-specifier (the name to the left of the last `::`) is a simple-template-id (template name followed by argument list in angle brackets) that names the class template. The template parameters of such template friend declaration must be deducible from the simple-template-id.

In this case, the member of any specialization of A becomes a friend. This does not involve instantiating the primary template A: the only requirements are that the deduction of the template parameters of A from that specialization succeeds, and that substitution of the deduced template arguments into the friend declaration produces a declaration that would be a valid redeclaration of the member of the specialization:

```
// primary template
template<class T>
struct A {
  struct B { };
  void f();
  struct D { void g(); };
  T h();
  template<T U> T i();
};
// full specialization
template<>
struct A<int> {
  struct B { };
  int f();
  struct D { void g(); };
  template<int U> int i();
};
// another full specialization
template<>
struct A<float*> {
    int *h();
};
// the non-template class granting friendship to members of class template A
class X {
  template<class T>
  friend struct A<T>::B; // all A<T>::B are friends, including A<int>::B

  template<class T>
  friend void A<T>::f();  // A<int>::f() is not a friend because its signature
                          // does not match, but e.g. A<char>::f() is a friend

//  template<class T>
//  friend void A<T>::D::g(); // ill-formed, the last part of the nested-name-specifier,
//                            // D in A<T>::D::, is not simple-template-id

  template<class T>
  friend int* A<T*>::h();  // all A<T*>::h are friends: A<float*>::h(), A<int*>::h(), etc

  template<class T>
  template<T U>       // all instantiations of A<T>::i() and A<int>::i() are friends,
  friend T A<T>::i(); // and thereby all specializations of those function templates
};
```

Default template arguments are only allowed on template friend declarations if the declaration is a definition and no other declarations of this function template appear in this translation unit. (since C++11)

Template friend operators

A common use case for template friends is declaration of a non-member operator overload that acts on a class template, e.g. `operator<<(std::ostream&, const Foo<T>&)` for some user-defined `Foo<T>`

Such operator can be defined in the class body, which has the effect of generating a separate non-template `operator<<` for each T and makes that non-template `operator<<` a friend of its `Foo<T>`

Run this code

```
#include <iostream>

template<typename T>
class Foo {
 public:
    Foo(const T& val) : data(val) {}
 private:
    T data;

    // generates a non-template operator<< for this T
    friend std::ostream& operator<<(std::ostream& os, const Foo& obj)
    {
        return os << obj.data;
    }
};

int main()
{
    Foo<double> obj(1.23);
    std::cout << obj << '\n';
}
```

Output:

```
1.23
```

or the function template has to be declared as a template before the class body, in which case the friend declaration within `Foo<T>` can refer to the full specialization of `operator<<` for its T:

Run this code

```
#include <iostream>

template<typename T>
class Foo; // forward declare to make function declaration possible

template<typename T> // declaration
std::ostream& operator<<(std::ostream&, const Foo<T>&);

template<typename T>
class Foo {
 public:
    Foo(const T& val) : data(val) {}
 private:
    T data;

    // refers to a full specialization for this particular T
    friend std::ostream& operator<< <> (std::ostream&, const Foo&);
    // note: this relies on template argument deduction in declarations
    // can also specify the template argument with operator<< <T>"
};

// definition
template<typename T>
std::ostream& operator<<(std::ostream& os, const Foo<T>& obj)
{
    return os << obj.data;
}

int main()
{
    Foo<double> obj(1.23);
    std::cout << obj << '\n';
}
```

Example

stream insertion and extraction operators are often declared as non-member friends

Run this code

```
#include <iostream>
#include <sstream>

class MyClass {
    int i;

    friend std::ostream& operator<<(std::ostream& out, const MyClass& o);
    friend std::istream& operator>>(std::istream& in, MyClass& o);
 public:
    MyClass(int i = 0) : i(i) {}
};

std::ostream& operator<<(std::ostream& out, const MyClass& mc)
{
    return out << mc.i;
}

std::istream& operator>>(std::istream& in, MyClass& mc)
{
    return in >> mc.i;
}

int main()
{
    MyClass mc(7);
    std::cout << mc << '\n';
    std::istringstream("100") >> mc;
    std::cout << mc << '\n';
}
```

Output:

```
7
100
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 45 (https://wg21.cmeerw.net/cwg/issue45)	C++98	members of a class nested in a friend class of T have no special access to T	a nested class has the same access as the enclosing class
CWG 500 (https://wg21.cmeerw.net/cwg/issue500)	C++98	friend class of T cannot inherit from private or protected members of T, but its nested class can	both can inherit from such members

References

- C++11 standard (ISO/IEC 14882:2011):
  - 11.3 Friends [class.friend]
  - 14.5.4 Friends [temp.friend]
- C++98 standard (ISO/IEC 14882:1998):
  - 11.3 Friends [class.friend]
  - 14.5.3 Friends [temp.friend]

See Also

Class declaration
Access specifiers