【C++】Day81(2)

Class	C++
Date	@April 12, 2022
Material	
Series Number	
Summary	Template Friendship

【Ch16】Input and Output

Simplifying Use of a Template Class Name inside Class Code

There is one exception to the rule that we must supply template arguments when we use a class template type.

Inside the scope of the class template itself, we may use the name of the template without arguments:

```
template <typename T> class BlobPtr {
public:
  BlobPtr() : curr(0) {}
  BlobPtr(Blob<T> &a, size_t sz = 0) : wptr(a.data), curr(sz) {}
  T& operator*() const {
    auto p = check(curr, "Dereference past end");
    return (*p)[curr];
  }
  // Prefix increment and decrement operators
  BlobPtr& operator++();
  BLobPtr& operator--();

private:
  // check returns a shared_ptr to the vector if the check succeeds
  std::shared_ptr<std::vector<T>> check(std::size_t, const std::string &) const;
  // stores a weak_ptr, which means the underlying vector might be destroyed
  std::weak_ptr<std::vector<T>> wptr;
  std::size_t curr; // current position within the array
};
```

When we are inside the scope of a class template, the compiler treats references to the template itself as if we had supplied template arguments.

Using a Class Template Name outside the Class Template Body

We must remember that we are not in the scope of the class until the class name is seen:

```
// Postfix operator
template <typename T>
BobPtr<T> BlobPtr<T>::operator++(int) {
  BlobPtr ret = *this;
  ++*this;
  return ret;
}
```

Class Templates and Friends

When a class contains a friend declaration, the class and the friend can independently be templates or not.

A class template that has a nontemplate friend grants that friend access to all the instantiations of the template.

When the friend is itself a template, the class granting friendship controls whether friendship includes all instantiations of the template or only specific instantiations.

One-to-One friendship

The most common form of friendship from a class template to another template establishes friendship between corresponding instantiations of the class and its friend.

In order to refer to a specific instantiation of a template, we must first declare the template itself.

```
// Forward declarations needed for friend declarations in Blob
template <typename> class BlobPtr;
template <typename> class Blob;
template <typename T> bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
  friend class BlobPtr<T>;
  friend bool operator==<T>(const Blob<T>&, const Blob<T>&);
};
```

For example, if we define a `Blob` variable as following:

```
Blob<char> a; // BlobPtr<char> and operator==<char> are friends
```

General and Specific Template Friendship

A class can make every instantiation of another template its friend, or it may limit friendship to a specific instantiation:

```
template <typename T> class Pal;
class C{
  friend class Pal<C>; // Pal instantiated with class C is a friend to C

  // All instances of Pal2 are friends to C
  // No forward declaration required when we befriend all instantiations of a class
  template <typename T> class Pal2;
};
```

Befriending the Template's Own Type Parameter

Under the new standard, we can make a template type parameter a friend:

```
template <typename Type> class Bar {
  friend Type; // Grans access to the type used to instantiate Bar
};
```

Thus, `Sales_data` would be a friend of `Bar<Sales_data>`