【C++】Day25

Class	C++
Date	@December 16, 2021
Material	
Series Number	
Summary	

【Ch7】Classes

7.3.3 Class Types

Every class defines a unique type. Two different classes define two different types even if they define the same members.

```
struct First {
  int member;
};

struct Second {
  int member;
};

First obj1;
Second obj2 = obj1; //error: obj1 and obj2 have different types
```

We can refer to a class type directly, by using the class name as a type name. Alternatively, we can use the class name following the keyword `class` or `struct`:

```
Sales_data item1; //default-initialized object of type Sales_data
class Sales_data item1; //equivalent declaration
```

The second method is inherited from C and also available in C++.

Class Declarations

Just as we can declare a function apart from its definition, we can also declare a class without defining it:

```
class Screen; //declaration of the Screen class
```

This declaration, sometimes referred to as a forward declaration, introduces the name `Screen` into the program and indicates that Screen refers to a class type.

After a declaration and before a definition is seen, the type `Screen` is an incomplete type-it's known that `Screen` is a class type but not known what members that type contains.

We can use an incomplete type in only limited ways:

- We can define pointers or references to such types
- We can declare(but not define) functions that use an incomplete type as a parameter or return type.

A class must be defined before we can write code that creates objects of that type. Otherwise, the compiler does not know how much storage such objects need.

Similarly, the class must be defined before a reference or pointer is used to access a member of the type.

Exercise

Exercises Section 7.3.3

Exercise 7.31: Define a pair of classes `X` and `Y`, in which `X` has a pointer to `Y`, and `Y` has an object of type `X`.

```
class Y;

class X {
  Y* item;
```

```
};

class Y {
  X item;
};
```

7.3.4 Friendship Revisited

A class can make another class its friend or it can declare specific member functions of another(previously defined) class as friends.

Friendship between Classes

As an example, our Window_mgr class will have members that will need access to the internal data of the Screen object it manages.

Screen can designate Window_mgr as its friend:

```
class Screen {
  //Window_mgr members can access the private parts of class Screen
  friend class Window_mgr;
  //rest of the Screen class
};
```

We can write the clear member of Window_mgr as follows:

```
class Window_mgr {
public:
using ScreenIndex = std::vector<Screen>::size_type
void clearScreen(ScreenIndex);

private:
  std::vector<Screen> screens{ Screen(24, 80, ' ') };
};

inline void Screen::clearScreen(ScreenIndex i) {
  Screen &s = screens[i];
  s.contents = string(s.height * s.width, ' ');
}
```

Note: Each class controls which classes or functions are its friends.

Making A Member Function a Friend

Rather than making the entire `Window_mgr` class a friend, Screen can instead specify that only the clear member is allowed access.

When we declare a member function to be a friend, we must specify the class of which that function is a member:

```
class Screen {
  //Window_mgr::clear must have been declared before class Screen
  friend void Window_mgr::clear(ScreenIndex);
};
```

In this case, we must order our program as follows:

1. First, define the `Window_mgr` class, which declares, but not define, `clear`. `Screen` must be declared before clear can use the members of `Screen`.
2. Next, define class `Screen`, including a friend declaration for `clear`
3. Finally, define `clear`, which can now refer to the members in Screen

Friend Declaration and Scope

Even if we define the function inside the class, we must still provide a declaration outside of the class itself to make that function visible:

```
struct X {
  friend void f() {}
  X() { f(); } //error:no declaration for f
  void g();
  void h();
};

void X::g() { return f(); } //error:f hasn't been declared
void f();
void X::h() { return f(); } //ok: declaration for f is now in scope
```

Exercise

Exercises Section 7.3.4

Exercise 7.32: Define your own versions of `Screen` and `Window_mgr` in which `clear` is a member of `Window_mgr` and a friend of `Screen`.

```
class Window_mgr;

class Screen {
public:
  using pos = int;
  Screen() = default;
  Screen(pos wd, pos ht, char c) : width(wd), height(ht), contents(wd * ht, c) {}
  friend Window_mgr;
  string print() {
    return contents;
  }

private:
  pos width = 0, height = 0;
  std::string contents;
};

class Window_mgr {
public:
  using ScreenIndex = std::vector<Screen>::size_type;
  void clearScreen(ScreenIndex);
  std::vector<Screen> screens{ Screen(24, 80, '#') };
};

inline void Window_mgr::clearScreen(ScreenIndex i) {
  Screen &s = screens[i];
  s.contents = string(s.height * s.width, ' ');
}


int main() {
  Window_mgr window = Window_mgr();
  cout << window.screens[0].print();
  window.clearScreen(0);
  cout << window.screens[0].print();

  return 0;
}
```