【C++】Day75

Class	C++
Date	@March 17, 2022
Material	
Series Number	
Summary	Class Scope under Inheritance

【Ch15】OOP

15.6 Class Cope Under Inheritance

Each class defines its own scope within which its members are defined. Under inheritance, the scope of a derived class is nested inside the scope of its base classes.

If a name is unresolved within the scope of the derived class, the enclosing base-class scopes are searched for a definition of that name.

It is this hierarchical nesting of class scopes that allows the members of a derived class to use members of its base class as if those members were part of the derived class.

For example, when we write

```
Bulk_quote bulk;
cout << bulk.isbn();
```

the use of the name `isbn` is resolved as follows:

- Because we called `isbn` on an object of type `Bulk_quote`, the search starts in the `Bulk_quote` class. The name `isbn` is not found in that class.
- Because `Bulk_quote` is derived from `Disc_quote`, the `Disc_quote` class is searched next. The name is still not found.
- Because `Disc_quote` is derived from `Quote`, the `Quote` class is searched next. The name `isbn` is found in that class; the use of `isbn` is resolved to the `isbn` in `Quote`.

Name Lookup Happens at Compile Time

The static type of an object, reference, or pointer determines which member of that object are visible. Even when the static and dynamic types might differ, the static type determines what members can be used.

As an example, we might add a member to the `Disc_quote` class that returns a pair holding the minimum quantity and the discounted price:

```
class Disc_quote : public Quoe {
public:
  std::pair<size_t, double> discount_policy() const {
    return {quantity, discount};
  }
};
```

We can use `discount_policy` only through an object, pointer, or reference of type `Disc_quote` or of a class derived from `Disc_quote`:

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk; // static and dynamic type are the same
Quote *itemP = &bulk; // static and dynamic type differs
bulkP->discount_policy(); // ok: BulkP has type Bulk_quote*
itemP->discount_policy(); // error: itemP has type Quote*
```

Even though bulk has a member named `discount_policy`, that member is not visible through `itemP`.

The type of `itemP` is a pointer to `Quote`, which means that the search for `discount_policy` starts in class `Quote`. The Quote class has no member named `discount_policy`, so we cannot call that member on an object, reference, or pointer of type `Quote`.

Name Collisions and Inheritance

Like any other scope, a derived class can reuse a name defined in one of its direct or indirect base classes.

As usual, names defined in an inner scope(e.g., a derived class) hides users of that name in the outer scope(e.g., a base class).

```
struct Base {
  Base() : mem(0) { }
protected:
  int mem;
};

struct Derived : Base {
  Derived(int i) : mem(i) {} // initializes Derived::mem to i
  // Base::i is default initialized
  int get_mem() { return  mem; } // returns Derived::mem
protected:
  int mem; // hides mem in the base
};
```

The reference to ~~mem~~ inside `get_mem` is resolved to the name inside `Derived`. Were we to write:

```
Derived d(42);
cout << d.get_mem() << endl; // print 42
```

Then the output would be 42.

Note: A derived-class member with the same name as a member of the base class hides direct use of the base-class member.

Using the Scope Operator to Use Hidden Members

We can use a hidden base-class member by using the scope operator:

```
struct Derived : Base {
  int get_base_mem() { return Base::mem; }
};
```

Best Practices: Aside from overriding inherited virtual functions, a derived class usually should not reuse names defines in its base class.

Name Lookup Happens before Type Checking

If a member in a derived class has the same name as a base class member(i.e., a name defined in an outer scope), then the derived member hides the base-class member within the scope of the derived class. The base member is hidden even if the functions have different parameter lists:

```
struct Base {
  int memfcn();
};

struct Derived : Base {
  int memfcn(int); // hides memfcn in the base
};
Derived d; Base b;
b.memfcn(); // calls Base::memfcn
d.memfcn(10); // calls Derived::memfcn
d.memfcn(); // error: memfcn with no arguments is hiddend.Base::memfcn(); // ok: calls Base::memfcn
```

Virtual Functions and Scope

If the base and derived members took arguments that differed from one another, there would be no way to call the derived version through a reference or pointer to the base class. For example:

```
class Base {
public:
  virtual int fcn();
};

class D1 : public Base {
public:
  // hides fcn in the base; this fcn is not virtual
  // D1 inherits the definition of Base::fcn()
  int fcn(int); // parameter list differs from fcn in Base
  virtual void f2(); // new virtual function that does not exist in Base
};

class D2 : public D1 {
public:
  int fcn(int); // nonvirtual function hides D1::fcn(int)
  int fcn(); // overrides virtual fcn from Base
  void f2(); // overrides virtual f2 from D1
};
```