【C++】Day73

Class	C++
Date	@March 14, 2022
Material	
Series Number	
Summary	Override Virtual Functions

【Ch15】OOP

15.3 Virtual Functions

In C++, dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type. Because we don't know which version of a function is called until run time, virtual functions must always be defined.

Because we don't know which version is called until run time, virtual functions must always be defined.

Calls to Virtual Functions May be Resolved at Run Time

When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call. The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference.

As an example, consider our `print_total` function. This function calls `net_price` on its parameter named `item`, which has type `Quote&`.

Because item is a reference, and because `net_price` is virtual, the version of `net_price` that is called depends at run time on the actual type of the argument bound to item.

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10); // calls Quote::net_price
```

```
Bulk_quote derived("0-201-82470-1", 50, 5, 0.19);
print_total(cout, derived, 10); // calls Bulk_quote::net_price
```

Note: Virtuals are resolved at run time only if the call is made through a reference or pointer. Only in these cases is it possible for an object's dynamic type to differ from its static type.

Virtual Functions in a Derived Class

When a derived class override a `virtual` function, it may, but not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains virtual in all the derived classes.

A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.

The return type of a `virtual` in the derived class also must match the return type of the function from the base class. An exception applies to virtuals that return a reference to types that are themselves related by inheritance.

That is, if `D` is derived from `B`, then a base class virtual can return a `B*` and the version in the derived class can return a `D*`.

Note: A function that is virtual in a base class is implicitly virtual in its derived classes. When a derived class overrides a virtual, the parameters in the base and derived classes must match exactly.

The final and override Specifiers

It is legal for a derived class to define a function with the same name as a virtual in its base class but with a different parameter list. The compiler considers such a function to be independent from the base-class function.

However, this is always a mistake-the class author intended to override a virtual from the base case but made a mistake specifying the parameter list.

Finding such bugs can be hard. Under the new standard we can specify `override` on a virtual function in a derived class. Doing so makes our intention clear and (more importantly) enlists the compiler in finding such problems for us.

The compiler will reject a program if a function marked override does not override an existing virtual function:

```
struct B {
  virtual void f1(int) const;
  virtual void f2();
  void f3();
};

struct D1 : B {
  void f1(int) const override; // ok: f1 matches B::f1
  void f2(int) override;  // error: B has no f2(int) function
  void f3() override; // error: f3 not virtual
  void f4() override; // error: B doesn't have a function named f4
};
```

We can also designate a function as `final`. Any attempt to override a function that has been defined as `final` will be flagged an error:

```
struct D2 : B {
  void f1(int) const final; // subsequent classes cannot override f1(int)
};

struct D3 : D2 {
  void f2(); // ok: overrides f2 inherited from the indirect base B
  void f1(int) const; // error: D2 declared f2 as final
};
```

Virtual Functions and Default Arguments

When a call is made through a reference or pointer to base, the default arguments will be those defined in the base class. The base-class arguments will be used even when the derived version of the function is run.

Best Practices: Virtual functions that have default arguments should use the same argument values in the base an derived classes.

Circumventing the Virtual Mechanism

In some cases, we want to prevent dynamic binding of a call to virtual function. We want to force the call to use a particular version of that virtual. We can use the scope operator to do so. For example:

```
// calls the version from the base class regardless of the dynamic type of baseP
double uniscounted = baseP -> Quote::net_price(42);
```

calls the `Quote` version of `net_price` regardless of the type of the object to which `baseP` actually points.

Note: Ordinarily, only code inside member functions should need to use the scope operator to circumvent the virtual mechanism.

Exercise

Exercise 15.11: Add a virtual `debug` function to your `Quote` class hierarchy that displays the data members of the respective classes.

See 15_11.cpp for code