Virtual Base Classes

Visual Studio 2012 2 out of 4 rated this helpful

Because a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.

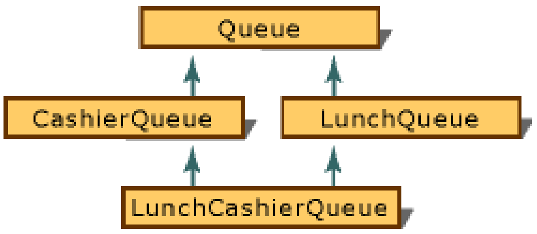
Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them.

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base.

When declaring a virtual base class, the **virtual** keyword appears in the base lists of the derived classes.

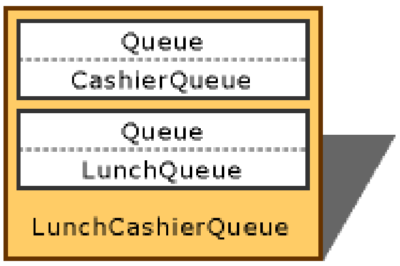
Consider the class hierarchy in the following figure, which illustrates a simulated lunch line.

Simulated Lunch-Line Graph



In the figure, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`, one from `CashierQueue` and the other from `LunchQueue`. The following figure shows the conceptual memory layout (the actual memory layout might be optimized).

Simulated Lunch-Line Object

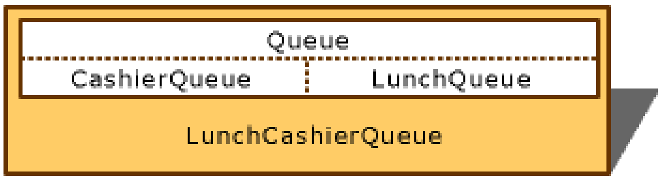


Note that there are two `Queue` subobjects in the `LunchCashierQueue` object. The following code declares `Queue` to be a virtual base class:

```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

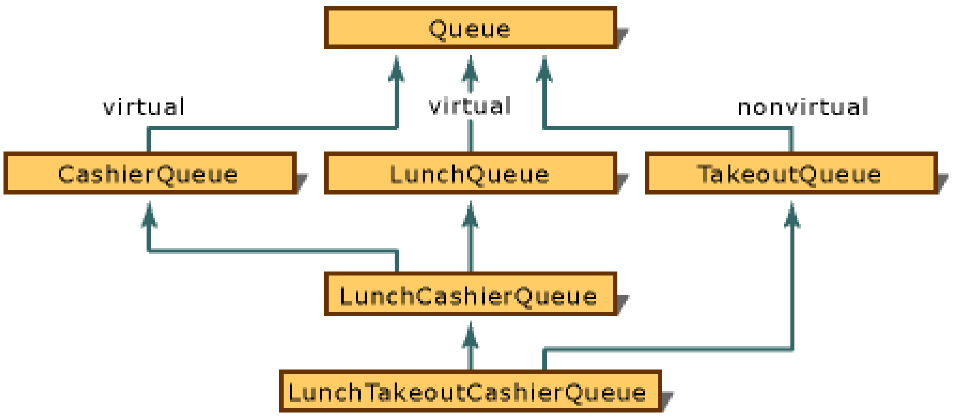
The `virtual` keyword ensures that only one copy of the subobject `Queue` is included (see the following figure).

Simulated Lunch-Line Object with Virtual Base Classes



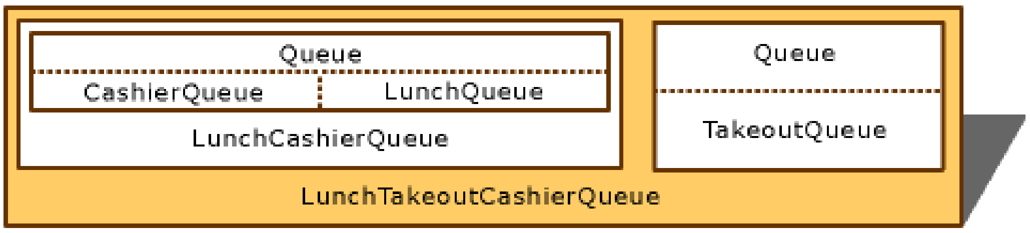
A class can have both a virtual component and a nonvirtual component of a given type. This happens in the conditions illustrated in the following figure.

Virtual and Nonvirtual Components of the Same Class



In the figure, `CashierQueue` and `LunchQueue` use `Queue` as a virtual base class. However, `TakeoutQueue` specifies `Queue` as a base class, not a virtual base class. Therefore, `LunchTakeoutCashierQueue` has two subobjects of type `Queue`: one from the inheritance path that includes `LunchCashierQueue` and one from the path that includes `TakeoutQueue`. This is illustrated in the following figure.

Object Layout with Virtual and Nonvirtual Inheritance



Note

Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce extra processing overhead.

If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or a destructor for the derived base class calls that function using a pointer to the virtual base class, the a destructor for the derived base class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into the classes with virtual bases. The /vd0 compiler option suppresses the addition of the hidden vtordisp constructor/destructor displacement member. The /vd1 compiler option, the default, enables them where they are necessary. Turn off vtordisps only if you are sure that all class constructors and destructors call virtual functions virtually.

The /vd compiler option affects an entire compilation module. Use the **vtordisp** pragma to suppress and then reenable vtordisp fields on a class-by-class basis:

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

See Also

Reference
Multiple Base Classes

Community Additions

© 2013 Microsoft. All rights reserved.