12.7 — Virtual base classes

BY ALEX ON JANUARY 28TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 14TH, 2017

Last chapter, in lesson **11.7 -- Multiple inheritance**, we left off talking about the “diamond problem”. In this section, we will resume this discussion.

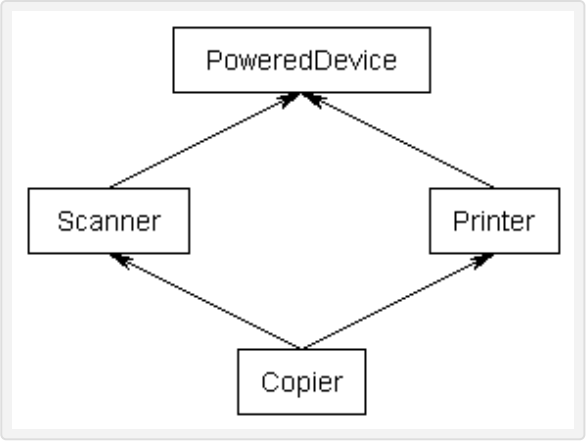
Note: This section is an advanced topic and can be skipped or skimmed if desired.

The diamond problem

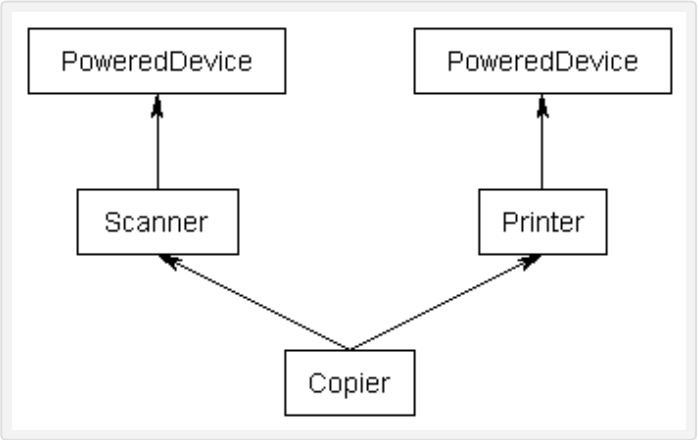
Here is our example from the previous lesson (with some constructors) illustrating the diamond problem:

```
class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
        cout << "PoweredDevice: " << power << '\n';
    }
};

class Scanner: public PoweredDevice
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice(power)
    {
        cout << "Scanner: " << scanner << '\n';
    }
};

class Printer: public PoweredDevice
{
public:
    Printer(int printer, int power)
        : PoweredDevice(power)
    {
        cout << "Printer: " << printer << '\n';
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : Scanner(scanner, power), Printer(printer, power)
    {
    }
};
```

Although you might expect to get an inheritance diagram that looks like this:



If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:



We can create a short example that will show this in action:

```
int main()
{
    Copier copier(1, 2, 3);
}
```

This produces the result:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

As you can see, PoweredDevice got constructed twice.

While this is often desired, other times you may want only one copy of PoweredDevice to be shared by both Scanner and Printer.

Virtual base classes

To share a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object that is shared. Here is an example (without constructors for simplicity) showing how to use the virtual keyword to create a shared base class:

```
class PoweredDevice
{
};

class Scanner: virtual public PoweredDevice
{
};

class Printer: virtual public PoweredDevice
```

```
{
};

class Copier: public Scanner, public Printer
{
};
```

Now, when you create a Copier class, you will get only one copy of PoweredDevice that will be shared by both Scanner and Printer.

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

```
#include <iostream>

class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
        std::cout << "PoweredDevice: " << power << '\n';
    }
};

class Scanner: virtual public PoweredDevice // note: PoweredDevice is now a virtual base class
{
public:
    Scanner(int scanner, int power)
        : PoweredDevice(power) // this line is required to create Scanner objects, but ignored in this
 case
    {
        std::cout << "Scanner: " << scanner << '\n';
    }
};

class Printer: virtual public PoweredDevice // note: PoweredDevice is now a virtual base class
{
public:
    Printer(int printer, int power)
        : PoweredDevice(power) // this line is required to create Printer objects, but ignored in this
 case
    {
        std::cout << "Printer: " << printer << '\n';
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power)
        : Scanner(scanner, power), Printer(printer, power),
        PoweredDevice(power) // PoweredDevice is constructed here
    {
    }
};
```

This time, our previous example:

```
int main()
{
    Copier copier(1, 2, 3);
}
```

produces the result:

```
PoweredDevice: 3
Scanner: 1
```

```
Printer: 2
```

As you can see, PoweredDevice only gets constructed once.

There are a few details that we would be remiss if we did not mention.

First, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the Scanner and Printer constructors still have calls to the PoweredDevice constructor. When creating an instance of Copier, these constructor calls are simply ignored because Copier is responsible for creating the PoweredDevice, not Scanner or Printer. However, if we were to create an instance of Scanner or Printer, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the *most* derived class is responsible for constructing the virtual base class. In this case, Copier inherits Printer and Scanner, both of which have a PoweredDevice virtual base class. Copier, the most derived class, is responsible for creation of PoweredDevice. Note that this is true even in a single inheritance case: if Copier was singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier is still responsible for creating PoweredDevice.

Fourth, a virtual base class is always considered a direct base of its most derived class (which is why the most derived class is responsible for its construction). But classes inheriting the virtual base still need access to it. So in order to facilitate this, the compiler creates a virtual table for each class directly inheriting the virtual class (Printer and Scanner). These virtual tables point to the functions in the most derived class. Because the derived classes have a virtual table, that also means they are now larger by a pointer (to the virtual table).

 **12.8 -- Object slicing**

 **Index**

 **12.6 -- Pure virtual functions, abstract base classes, and interface classes**

Share this:

C++ TUTORIAL | PRINT THIS POST

51 comments to 12.7 — Virtual base classes

Ritesh
June 5, 2018 at 12:00 am · Reply

Hey Alex,
I didn't understood the fourth point completely.
what did you mean by
"But classes inheriting the virtual base still need access to it. So in order to facilitate this, the compiler creates a virtual table for each class directly inheriting the virtual class (Printer and Scanner). These virtual tables point to the functions in the most derived class."
does this mean that the virtual table of the printer and scanner will point to Copier's functions