25.8 — Virtual base classes

ALEX[1] SEPTEMBER 11, 2023

Last chapter, in lesson 24.9 -- Multiple inheritance (https://www.learncpp.com/cpp-tutorial/multiple-inheritance/)[2], we left off talking about the "diamond problem". In this section, we will resume this discussion.

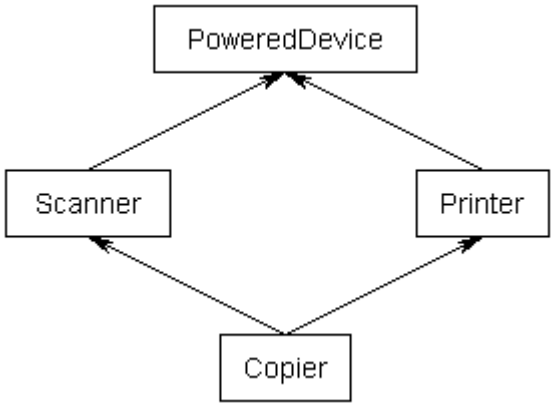
Note: This section is an advanced topic and can be skipped or skimmed if desired.

The diamond problem

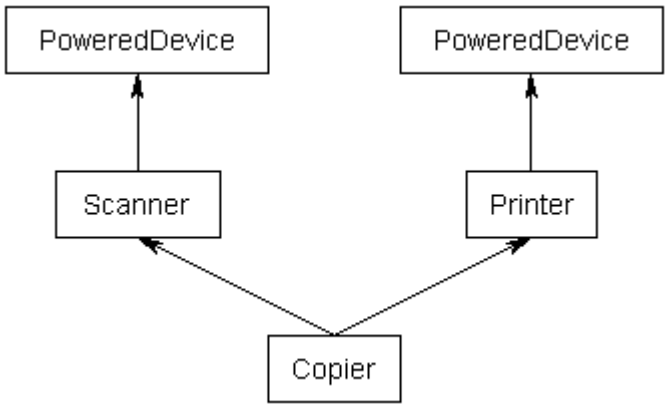
Here is our example from the previous lesson (with some constructors) illustrating the diamond problem:

```
#include <iostream>

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

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

class Printer: public PoweredDevice
{
public:
    Printer(int printer, int power)
        : PoweredDevice{ power }
    {
        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 }
    {
    }
};
```

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:

```
1  int main()
2  {
3      Copier copier{ 1, 2, 3 };
4
5      return 0;
6  }
```

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. The base object is shared between all objects in the inheritance tree and it is only constructed once. 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 object, you will get only one copy of PoweredDevice per Copier 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:

```cpp
#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)
        : PoweredDevice{ power }, // PoweredDevice is constructed here
        Scanner{ scanner, power }, Printer{ printer, power }
    {
    }
};
```

This time, our previous example:

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

    return 0;
}
```

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, for the constructor of the most derived class, 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 singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier is still responsible for creating PoweredDevice.

Fourth, all classes inheriting a virtual base class will have a virtual table, even if they would normally not have one otherwise, and thus instances of the class will be larger by a pointer.

Because Scanner and Printer derive virtually from PoweredDevice, Copier will only be one PoweredDevice subobject. Scanner and Printer both need to know how to find that single PoweredDevice subobject, so they can access its members (because after all, they are derived from it). This is typically done through some virtual table magic (which essentially stores the offset from each subclass to the PoweredDevice subobject).

Next lesson

25.9 Object slicing

3

Back to table of contents

4

Previous lesson

25.7 Pure virtual functions, abstract base classes, and interface classes

5

6

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[8] are connected to your provided email address.

133 COMMENTS

Newest

Nidhi Gupta

May 6, 2025 9:19 am PDT

If both Scanner and Printer inherit non-virtually from PoweredDevice, and Copier inherits from both, two separate PoweredDevice subobjects are created. That's often not what we want—it leads to duplication and ambiguity.

class Scanner : virtual public PoweredDevice { };

class Printer : public PoweredDevice { }; // ← NOT virtual

class Copier : public Scanner, public Printer {

Copier(...) : PoweredDevice{...}, Scanner{...}, Printer{...} { }

};

0 Reply

EmtyC

December 29, 2024 2:04 pm PST

uh, I am being deep fried :>, but I kinda used to it at this point :D

Question:

```cpp
#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 : public PoweredDevice // note: no longer inherits virtually
{
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)
        : PoweredDevice{ power }, // PoweredDevice is constructed here
        Scanner{ scanner, power }, Printer{ printer, power }
    {
    }
};

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

    return 0;
}
```

Compiler error: error C2385: ambiguous access of 'PoweredDevice'
Is it because we aren't allowed to call a non-directly inherited parent's Constructor for Printer, and we can for Scanner, so the compiler is confused? Hmmm, but why doesn't it use the Printer constructor to construct the PoweredDevice part for it Printer part, and an other PoweredDevice for the virtuals (so practically the diamond problem is some what back, but at the same time it feels weird).
Lol, that's why you recommend to stay away from polymorphism as much as possible :>

1 Reply

Alex Author

Reply to EmtyC [9] January 4, 2025 3:47 pm PST

I think this may be something that Visual Studio just doesn't handle correctly. You should be getting 2 PoweredDevice, one shared by all instances of Scanner, and one for each Printer.

Clang compiles your example just fine.

3 Reply

EmtyC

Reply to Alex [10] January 5, 2025 3:02 am PST

I considered to move to clang in VS a while back, but MSVC is better supported with intellisense, which is quite useful as it messages are more understandable (the above example compiles but triggers 8 intellisense errors related to the implementation files)

0 Reply

EmtyC

Reply to Alex [10] January 5, 2025 2:56 am PST

Hmmm, interesting. Thank you <3

0 Reply

Kirill

February 6, 2024 5:21 am PST

Is there reason why keyword "virtual" is used into child class declaration and not into parent class declaration (into class that we actually want to make virtual base class)? Something like this:

```
1 virtual class PoweredDevice
2 {
3 };
```

Also, virtual base classes were made specifically to resolve “diamond problem” or there are more cases in which we want to use them?

4 Reply

Asicx

Reply to Kirill [11] October 1, 2024 11:35 am PDT

I feel like it should have been called "base class inherited virtually" instead of "virtual base class". Makes more sense to me.

2 Reply

Alex Author

Reply to Kirill [11] February 6, 2024 2:07 pm PST

It's so that the base class can be used in either configuration (either virtually or non-virtually).

AFAIK they are largely to solve the diamond problem.

6 Reply

noctis

June 30, 2023 12:37 am PDT

Can you please explain why are these concepts - `virtual function`, `pure virtual function` and `virtual base class` use the term `virtual` in them ?

Last edited 2 years ago by noctis

0 Reply

Alex Author

Reply to noctis [12] July 1, 2023 6:56 pm PDT

A pure virtual function is a virtual function that doesn't have a body, so that's one's easy.
I have no idea why a virtual base class is called virtual.

8 Reply

Horu

May 20, 2023 7:17 am PDT

```cpp
#include <iostream>
#include <array>
#include <string>
#include <string_view>

class GrandParent
{

public:
    GrandParent(std::string_view gPname)
    {
        std::cout << "GrandParent: " << gPname << '\n';
    }

};


class Mother : virtual public GrandParent
{
public:
    Mother(std::string_view mName, std::string_view gPname) : GrandParent{ gPname }
    {
        std::cout << "Mother: " << mName << '\n';
    }

};

class Father : virtual public GrandParent
{
public:
    Father(std::string_view fName, std::string_view gPname) : GrandParent{gPname}
    {
        std::cout << "Father: " << fName << '\n';
    }
};

class Child : public Mother, public Father
{
public:
    Child(std::string_view mName, std::string_view fName, std::string_view gPname ) :
GrandParent{gPname}, Mother{mName, gPname}, Father{fName, gPname}
    {}


};


int main()
{

    Child c{ "Alex2", "Alex3", "Alex1" };


    return 0;
}
```


Bubun
March 20, 2023 9:32 am PDT

"First, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes."
So if we have inheritance like this

```
class A
{
};

class B: virtual public A
{
};

class C: public B
{
};

class D: virtual public C
{
};
```

the order of construction is A, C, B, D, right?

0 Reply

Alex Author

Reply to Bubun [13] March 21, 2023 4:25 pm PDT

No, that would cause issues because C is derived from B and the code in C will assume B has already been constructed. The order of construction for this is A, B, C, D.

I updated the sentence to read, "First, for the constructor of the most derived class, virtual base classes are always created before non-virtual base classes, which ensures all bases get created before their derived classes."

So in this case, virtual base class C will be constructed before any other non-virtual base classes of D (but there aren't any). The rule does not apply recursively.

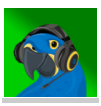
1 Reply

Bubun

Reply to Alex [14] March 21, 2023 4:37 pm PDT

Got it, thanks!

1 Reply



Avtem

January 21, 2023 5:11 am PST

i can not believe it, but it is actually the solution i needed for developing my framework. Very nicely explained!

1 Reply

Edman

January 9, 2023 8:56 pm PST

Because Scanner and Printer derive virtually from PoweredDevice, Copier will only be one PoweredDevice subobject. Scanner and Printer both need to know how to find that single PoweredDevice subobject, so they can access its members (because after all, they are derived from it). This is typically done through some virtual table magic (which essentially stores the offset from each subclass to the PoweredDevice subobject).

The most hazy part of the topic. I guess this part needs more details or may be examples to clarify.

7 Reply

Fab

Reply to Edman [15] August 6, 2023 8:14 am PDT

I believe "Copier will only be one PoweredDevice subobject" means that since Scanner and Printer derive virtually from PoweredDevice, they share and are derived from the same virtual base class. This also means Copier is only ONE PoweredDevice subobject, since there will only be one PoweredDevice virtual base class in the inheritance tree.

If Scanner and Printer didn't derive virtually, they'd each have their own separate PoweredDevice base class they derive from in the inheritance tree (shown in the 2nd diagram of this lesson). This would mean Copier will be a subobject of 2 PoweredDevice base classes because 2 PoweredDevice base classes exist in the inheritance tree.

Last edited 1 year ago by Fab

0 Reply

JustABug

June 8, 2022 11:13 pm PDT

`#include <iostream>` is missing in the first example.

1 Reply

Kuba

May 9, 2022 12:34 am PDT

Sometimes i get the feeling like classes arent as robust as some other concepts in programming. Like they work some way (for example most derived class creates base virtual class) because thats how it has to work. It doesnt make sense and goes against the rules. Still im very new to programming so im cautious making such statements. Its very cool to learn about something that didnt excist before. Like with physics you make obesrvations on how stuff works in real life and make documentations on it. In programming its our job to set all the rules and guard them. We arent restricted because its our world were working on, but its in our very interest that this world has rules that we can follow and refer to, that it can be intuitive.
Just some random thoughts, once again great explanations on the subject, big thanks!

6 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/
3. https://www.learncpp.com/cpp-tutorial/object-slicing/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/
6. https://www.learncpp.com/virtual-base-classes/
7. https://www.learncpp.com/breaktime/break-time-saint-petersburg/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-605901
10. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-606242
11. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-593305
12. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-582892
13. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-578551
14. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-578584
15. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/#comment-576033
16. https://g.ezoic.net/privacy/learncpp.com