25.6 — The virtual table

ALEX[1] DECEMBER 7, 2024

Consider the following program:

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

class Base
{
public:
    std::string_view getName() const { return "Base"; }                // not virtual
    virtual std::string_view getNameVirtual() const { return "Base"; } // virtual
};

class Derived: public Base
{
public:
    std::string_view getName() const { return "Derived"; }
    virtual std::string_view getNameVirtual() const override { return "Derived"; }
};

int main()
{
    Derived derived {};
    Base& base { derived };

    std::cout << "base has static type " << base.getName() << '\n';
    std::cout << "base has dynamic type " << base.getNameVirtual() << '\n';

    return 0;
}
```

First, let's look at the call to `base.getName()`. Because this is a non-virtual function, the compiler can use the actual type of `base` (`Base`) to determine (at compile-time) that this should resolve to `Base::getName()`.

Although it looks almost identical, the call to `base.getNameVirtual()` must be resolved differently. Because this is a virtual function call, the compiler must use the dynamic type of `base` to resolve the call, and the dynamic type of `base` is not knowable until runtime. Therefore, only at runtime will it be determined that this particular call to `base.getNameVirtual()` resolves to `Derived::getNameVirtual()`, not `Base::getNameVirtual()`.

So how do virtual functions actually work?

The virtual table

The C++ standard does not specify how virtual functions should be implemented (this detail is left up to the implementation).

However, C++ implementations typically implement virtual functions using a form of late binding known as the virtual table.

The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”. In C++, virtual function resolution is sometimes called **dynamic dispatch**.

Nomenclature

Here’s an easier way of thinking about it in C++:
Early binding/static dispatch = direct function call overload resolution
Late binding = indirect function call resolution
Dynamic dispatch = virtual function override resolution

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it’s a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) has a corresponding virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer that is a member of the base class, which we will call `*__vptr`. `*__vptr` is set (automatically) when a class object is created so that it points to the virtual table for that class. Unlike the `this` pointer, which is actually a function parameter used by the compiler to resolve self-references, `*__vptr` is a real pointer member. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that `*__vptr` is inherited by derived classes, which is important.

By now, you’re probably confused as to how these things all fit together, so let’s take a look at a simple example:

```cpp
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    void function1() override {};
};

class D2: public Base
{
public:
    void function2() override {};
};
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer member to the most base class that uses virtual functions. Although the compiler does this automatically, we’ll put it in the next example just to show where it’s added:

```cpp
class Base
{
public:
    VirtualTable* __vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    void function1() override {};
};

class D2: public Base
{
public:
    void function2() override {};
};
```

When a class object is created, `*__vptr` is set to point to the virtual table for that class. For example, when an object of type Base is created, `*__vptr` is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, `*__vptr` is set to point to the virtual table for D1 or D2 respectively.

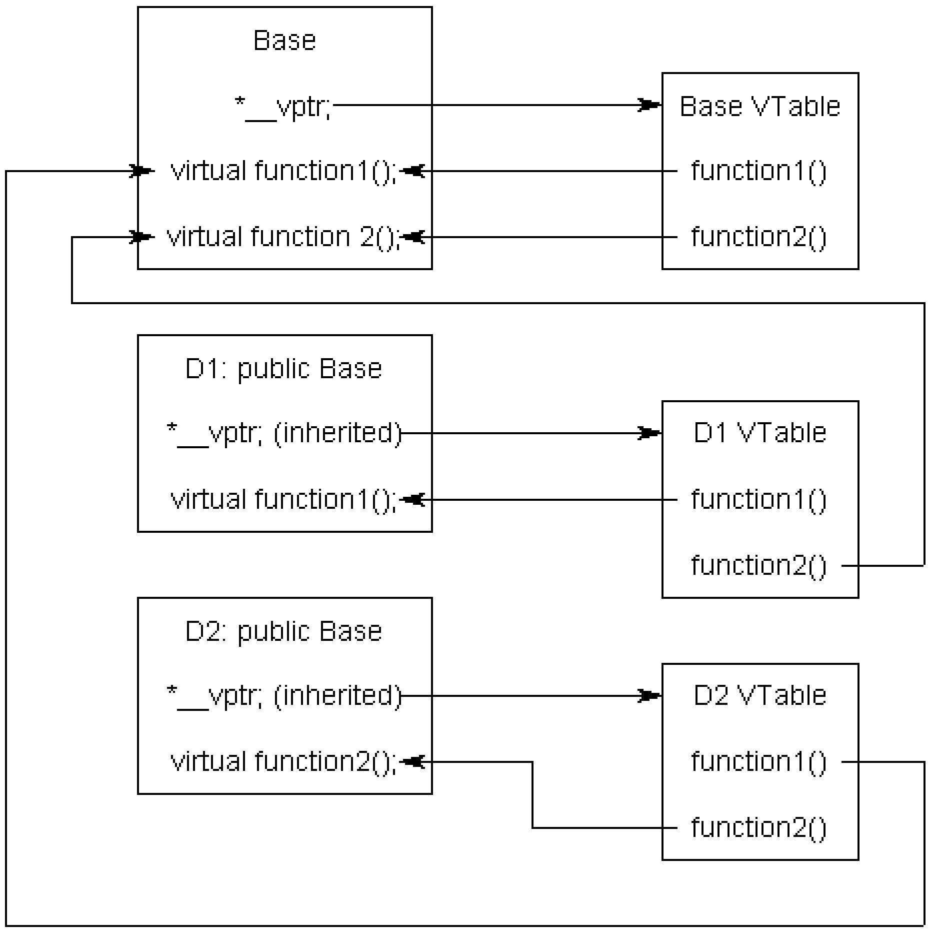
Now, let's talk about how these virtual tables are filled out. Because there are only two virtual functions here, each virtual table will have two entries (one for function1() and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

The virtual table for Base objects is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1() and the entry for function2 points to Base::function2().

The virtual table for D1 is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn't overridden function2(), so the entry for function2 will point to Base::function2().

The virtual table for D2 is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here's a picture of this graphically:



Although this diagram is kind of crazy looking, it's really quite simple: the `*__vptr` in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function that objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

```cpp
int main()
{
    D1 d1 {};
}
```

Because d1 is a D1 object, d1 has its *__vptr set to the D1 virtual table.

Now, let's set a base pointer to D1:

```cpp
int main()
{
    D1 d1 {};
    Base* dPtr = &d1;

    return 0;
}
```

Note that because dPtr is a base pointer, it only points to the Base portion of d1. However, also note that `*__vptr` is in the Base portion of the class, so dPtr has access to this pointer. Finally, note that `dPtr->__vptr` points to the D1 virtual table! Consequently, even though dPtr is of type `Base*`, it still has access to D1's virtual table (through `__vptr`).

So what happens when we try to call dPtr->function1()?

```cpp
int main()
{
    D1 d1 {};
    Base* dPtr = &d1;
    dPtr->function1();

    return 0;
}
```

First, the program recognizes that function1() is a virtual function. Second, the program uses `dPtr->__vptr` to get to D1's virtual table. Third, it looks up which version of function1() to call in D1's virtual table. This has been set to D1::function1(). Therefore, `dPtr->function1()` resolves to D1::function1()!

Now, you might be saying, "But what if dPtr really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?". The answer is no.

```cpp
int main()
{
    Base b {};
    Base* bPtr = &b;
    bPtr->function1();

    return 0;
}
```

In this case, when b is created, b.__vptr points to Base's virtual table, not D1's virtual table. Since bPtr is pointing to b, `bPtr->__vptr` points to Base's virtual table as well. Base's virtual table entry for function1() points to Base::function1(). Thus, `bPtr->function1()` resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you're only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the `*__vptr` to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

Also as a reminder, any class that uses virtual functions has a `*__vptr`, and thus each object of that class will be bigger by one pointer. Virtual functions are powerful, but they do have a performance cost.

[Next lesson](#)
25.7 [Pure virtual functions, abstract base classes, and interface classes](#)

2

[Back to table of contents](#)

3

[Previous lesson](#)
25.5 [Early binding and late binding](#)

4

5

simo
May 27, 2025 12:57 am PDT

i think that is the best explanation i read

3 Reply

David
February 15, 2025 7:23 am PST

This is the part where it really “clicked”.
I think it’s worth re-reading for those who didn’t understand)

"Note that because dPtr is a base pointer, it only points to the Base portion of d1. However, also note that __*vptr is in the Base portion of the class, so dPtr has access to this pointer. Finally, note that dPtr->__vptr points to the D1 virtual table! Consequently, even though dPtr is of type Base*, it still has access to D1's virtual table (through __vptr)."

10 Reply

Jeremy

Reply to David [7] May 4, 2025 9:29 pm PDT

Same for me. Good note.

1 Reply

Rykard

January 28, 2025 3:33 am PST

I think it's kinda confusing that in the diagramm D1 and D2 have the __*vptr as part of their class, because doesn't the *__vptr stay in the base class and is updated in the construction process of the derived object,so that a pointer Base* to object of D1/D2 still has access to the right virtual table? Thinking about it, obviously this has to be the case but maybe it's just me.

1 Reply

Alex Author

Reply to Rykard [8] January 29, 2025 7:50 pm PST

Yep. That's why it says inherited. I agree it could probably be clearer but I'm not sure how to do so and still keep the picture simple.

0 Reply

Robert Reimann

Reply to Alex [9] March 30, 2025 9:59 am PDT

I think the "(inherited)" in the diagram could be highlighted in BOLD text or just a different color that stands out more. Because I did not think about it at all when reading through it... But only now understood it when reading this comment.

0 Reply

Kaus05

January 19, 2025 12:14 am PST

Does the size of vtable increase based on number of virtual functions? is it a static array of function pointers or integer that would hold addresses

0 Reply

Gavin
Reply to Kaus05 [10] March 5, 2025 5:50 pm PST

Yes, the size of vtable will increase as more virtual functions are added. Vtable is a static array constructed by compiler at compiling time, so are the offsets to entries.

0 Reply

Artem
July 31, 2024 12:23 am PDT

It was a bit difficult to understand. It would be good if you add something like this "When we create a reference or pointer to the base class, the vptr remains the same as in the derived class, since it is initially inherited pointer"

Last edited 11 months ago by Artem

2 Reply

Alex Author
Reply to Artem [11] July 31, 2024 4:11 pm PDT

Setting a reference or pointer to an object never changes anything about the object. The vptr is a member of the base class, so it is always accessible regardless of the type of the object accessing it.

0 Reply

wangzhichao
July 26, 2024 9:11 pm PDT

I have presented the elements of vtables by gdb, which may be another way.

```cpp
#include <iostream>

// the virtual table

class Base
{
public:
    virtual void function1()
    {
        std::cout << "Base::function1\n";
    }
    virtual void function2()
    {
        std::cout << "Base::function2\n";
    }
};

class D1: public Base
{
public:
    void function1() override
    {
        std::cout << "D1::function1\n";
    }
};

class D2: public Base
{
public:
    void function2() override
    {
        std::cout << "D2::function2\n";
    }
};

int main()
{
    Base b {};
    Base* bPtr { &b };
    bPtr->function1();
    bPtr->function2();

    D1 d1 {};
    Base* dPtr { &d1 };
    dPtr->function1();
    dPtr->function2();

    D2 d2 {};
    Base* dPtr2 { &d2 };
    dPtr2->function1();
    dPtr2->function2();

    return 0;
}

/*
g++ (Ubuntu 13.1.0-8ubuntu1~20.04.2) 13.1.0

(gdb) info vtbl b
vtable for 'Base' @ 0x555555557d60 (subobject @ 0x7fffffffdac8):
[0]: 0x555555555256 <Base::function1()>
[1]: 0x555555555282 <Base::function2()>
(gdb) info vtbl d1
vtable for 'D1' @ 0x555555557d40 (subobject @ 0x7fffffffdad0):
[0]: 0x5555555552ae <D1::function1()>
[1]: 0x555555555282 <Base::function2()>
(gdb) info vtbl d2
vtable for 'D2' @ 0x555555557d20 (subobject @ 0x7fffffffdad8):
[0]: 0x555555555256 <Base::function1()>
[1]: 0x5555555552da <D2::function2()>
```

```
(gdb)

Output:
Base::function1
Base::function2
D1::function1
Base::function2
Base::function1
D2::function2
*/
```

4 Reply

Dck
April 6, 2024 9:12 pm PDT

Hi Alex, I'm curious about the performance of finding the most derived version for each function. I think it's fairly straightforward since we will just traverse upstream to the Base class, but since we have to do this for every function in every virtual table, is the performance cost significant?

In addition, how do compilers usually find the entry for a function in the virtual table? Is this similar to how we can access an entry in a hash table in O(1) time?

Thank you!

0 Reply

> **Alex** Author
> Reply to Dck 12 April 8, 2024 3:59 pm PDT
>
> You're thinking about it incorrectly. Each class has a table that contains a link to the most derived function for objects of that class. These tables are constructed at compile time so they have no runtime cost.
>
> The virtual pointer points to the table for the actual type of object being constructed. Calling the most derived function requires two operations: one to get to the virtual table for the class (through the virtual pointer), and another to call the function.
>
> I'm not sure how compilers find the actual entry in the table, as this is an implementation-specific detail. But I don't see why the virtual table couldn't have the same set of functions as the class in the same order and just look them up based on position.
>
> 0 Reply

>> **Dck**
>> Reply to Alex 13 April 8, 2024 4:17 pm PDT
>>
>> I got it now, thanks!
>>
>> 0 Reply

George D.
April 1, 2024 2:20 am PDT

Hi Alex,

How can both of the following statements be true? :
a) *__vptr is inherited by derived classes
b) When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively

Judging also from the diagram, I understand that each function, derived or not, has its own __vptr, right? So what does the "inherited" refers to?

0 Reply

Alex Author
Reply to George D. 14 April 1, 2024 9:59 am PDT

a) happens at compile time, b) happens at runtime.

> Judging also from the diagram, I understand that each function, derived or not, has its own __vptr, right? So what does the "inherited" refers to?

No, each base class with virtual functions has its own __vptr. Any class derived from this base class will inherit this __vptr from the base class.

1 Reply

Strain
March 29, 2024 9:31 am PDT

Fun fact:

This approach increases the size of every object that uses virtual things. In Rust, another approach is used: wide pointers. The v-table is part of the pointer instead of the object. This approach is very hard to be implemented in C++, however, as the existence of incomplete types imposed a challenge: would a wide pointer or a thin pointer be used for it?

0 Reply

sdk
January 25, 2024 8:56 am PST

if __vptr* is public as in the example, it should be unique for the entire hierarchy? so no need each class to have a __vptr*. when a new object is created the __vptr will points to the vtable of the class form which the object was instantiated? is that correct?
or it is other way around each class of the hierarchy has its private member __vptr that points to its vtable?

1 Reply

Alex Author
Reply to sdk 15 January 26, 2024 11:19 am PST

Assuming single inheritance, the base class will have a __vptr, which will be inherited by derived classes. When the object is created, the __vptr points to the virtual table of the type that was created.

1 Reply

Roman

Reply to Alex [16] March 27, 2024 7:33 am PDT

I still have the same question as above. __vprt*, if it's ONE for the whole hierarchy (say, class A parent of class B, which is in turn parent of class C) with addresses for created object of class C, where we will get address for virtual function of class B (when we access through pointer of type B)?
In other words shouldn't be __vptr* virtual member (in each object in the hierarchy, actually as shown on the diagram in the article) instead of just one for some entire hierarchy (as in the code sketch in the article, where there is just one non virtual VirtualTable* __vptr)?
Thank you!

Last edited 1 year ago by Roman

0 Reply

Alex Author

Reply to Roman [17] March 29, 2024 10:52 am PDT

It's one per hierarchy. If the object created is type C, a B* or B& will still call C's override of the virtual function (as will an A* or A&). This is the point of virtual functions -- you always get the most derived version of the function for the object's actual type. Therefore, the virtual table only needs to store a pointer to the most derived version of each function.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/
5. https://www.learncpp.com/the-virtual-table/
6. https://gravatar.com/
7. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-607763
8. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-607095
9. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-607170
10. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-606740
11. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-600323

12. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-595497
13. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-595547
14. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-595302
15. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-592857
16. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-592918
17. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-595161
18. https://g.ezoic.net/privacy/learncpp.com