LEARN C++
Skill up with our free tutorials

18.6 — The virtual table

ALEX[1] JUNE 8, 2022

To implement virtual functions, C++ uses a special 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".

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) is given its own 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 instance 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. Consequently, it makes each class instance 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:

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

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

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

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:

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

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

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

When a class instance 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.

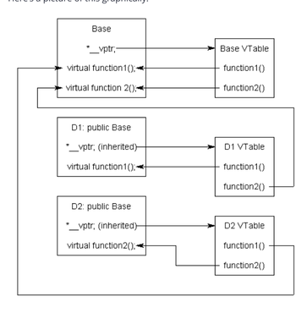
Now, let's talk about how these virtual tables are filled out. Because there are only two virtual functions here, each virtual table will have two entries (one for function1() and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived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

```
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:

```
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()?

```
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.

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

    return 0;
}
```

In this case, when b is created, __vptr points to Base's virtual table, not D1's virtual table. Consequently, bPtr->__vptr will also be pointing to Base's virtual table. 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
18.7 Pure virtual functions, abstract base classes, and interface classes
2
Back to table of contents
3
Previous lesson
18.5 Early binding and late binding
4
5

288 COMMENTS Newest

JustABug
June 7, 2022 11:10 pm

`The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.`

Isn't `they` missing here (`… they are allowed to call.`)?

0 Reply

Alex Author
Reply to JustABug[7] June 8, 2022 9:29 am

It's definitely missing a word. I rewrote as, "The entries in the virtual table point to the most-derived version of the function that objects of that class are allowed to call"

0 Reply

Amir
March 25, 2022 3:23 am

Hello,
The picture isn't working

0 Reply

Justin
March 18, 2022 11:17 pm

I know that when you say "class object" you mean an instance object of a particular class. However, coming from a Python background, this can be confusing since classes *are* objects in Python; A "class object" refers to the class itself, not an instance of it. I prefer "instance" or "instance object." I forgot if you disambiguate the term in an earlier chapter, but it might be a good idea to do so if you haven't.

0 Reply

Alex Author
Reply to Justin[8] March 20, 2022 8:39 am

Updated to `class instance`, at least in this lesson.

1 Reply

Radhey Saykar
March 4, 2022 10:40 pm

I understood it really well. Thank you

3 Reply

Ali Tariq
January 30, 2022 6:49 pm

Please tell me reason why the private function of D2 is calling?

#include <iostream>
using namespace std;

class Base
{
public:
// VirtualTable* __vptr;
virtual void function1() { cout << "Base\n"; };
virtual void function2(){};
};

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

class D2 : public Base
{
private:
virtual void function1() { cout << "Derived 1" << endl; };

public:
virtual void function2() { cout << "Derived" << endl; };
};

int main()
{
D2 d;
Base &b = d;
b.function1();

return 0;
}

0 Reply

Frank
Reply to Ali Tariq[9] May 11, 2022 12:09 pm

If you program like this, you will get base.
Base b = (Base b)d.
b.function1()

0 Reply

Alex Author
Reply to Ali Tariq[9] January 31, 2022 6:14 pm

Because b is a Base, and function1() is public in Base. The fact that function1() was made private in D2 only affects access through objects with type D2.

1 Reply

Stefan
November 23, 2021 2:51 am

Let's consider the simple example below:

```
//
class Base
{
public:
virtual void printClassName(){ cout<<"Base"<<endl;};
};

class Derived:public Base
{
public:
virtual void printClassName(){ cout<<"Derived"<<endl;};
};

int main()
{
//create a reference to Base class that refers to a Derived object
Derived derivedInst{};
Base& baseRef = derivedInst;

//the line below would print Derived
baseRef.printClassName();
return 0;
}
```

Firstly, it is known that each non static method name is mangled with *this when called. That said, the virtual method's definition in the Base class becomes:
virtual void printClassName(Base* const this)

Q1: Then, what happens with virtual method's name mangling in the Derived class? Concretely, to which of the versions below is it changed upon compilation?

1. virtual void printClassName(Base* const this)
2. virtual void printClassName(Derived* const this)

Q2: If it is changed to version 1, would it mean that *this pointer needs dynamic_casted to Derived class in order to access the Derived vptr?

Q3: Is the vptr accessed by dereferencing *this? Concretely, the following line:
baseRef.printClassName();
would become:
this->vptr->printClassName()
or
dynamic_cast<Derived*>(this)->vptr->printClassName()

Last edited 6 months ago by Stefan

1 Reply

Alex Author
Reply to Stefan[10] November 24, 2021 4:43 pm

1. virtual void printClassName(Derived* const this)
2. Yes, but this happens implicitly.
3. BaseRef.printClassName() would call Derived::printClassName(dynamic_cast(baseRef))

1 Reply

Stefan
Reply to Alex[11] November 25, 2021 1:47 am

Thanks, Alex!

Nonetheless, one aspect still remains unclear to me you mentioned the following: "First, we have to use the *__vptr to get to the appropriate virtual table."
When the vptr is accessed, is it accessed via *this, such as this->vptr->vtable[whatever_index]? Where *this should have been previously dynamic_casted

Additionally, is there a good reason for *this to be always dynamic_casted instead of being static_casted?

Last edited 6 months ago by Stefan

1 Reply

Alex Author
Reply to Stefan[12] November 26, 2021 2:50 pm

1. Yes, *__vptr is a member, so it gets accessed via the this pointer.
2. Upon reflection, my previous answer was probably incorrect: it likely uses static_cast since it doesn't need the runtime checking that dynamic_cast does.

1 Reply

mursu
October 26, 2021 11:32 am

>The virtual table is actually quite simple, though it's a little complex to describe in words.

I think you did a great job and it was all really simple and understandable on the first read already

2 Reply

Ishtmeet
October 26, 2021 12:05 am

Can you clarify on this statement

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).

If `dPtr` is a pointer to the base part of `d1`, doesn't the `d1->__vPtr` returns the V Table of the Base class instead of the D1 class?

Last edited 7 months ago by Ishtmeet

1 Reply

Alex Author
Reply to Ishtmeet[13] October 26, 2021 12:05 pm

No. __vptr is set when the object is created. If the object was created as a D1, then the object's __vptr will point to the D1 virtual table. The fact that dPtr is a Base pointer is irrelevant to this.

2 Reply

George
October 21, 2021 12:13 am

Do we have access to __vptr pointer? Can we print the adress it contains? I'd like to compare these pointers between objects.

1 Reply

Alex Author
Reply to George[14] October 21, 2021 11:02 am

You don't have access to it in your programs (remember, a __vptr is an implementation specific detail). You may be able to see it in a debugger. For example, in Visual Studio's debugger, I can see a __vfptr pointer in class objects that contain virtual functions.

1 Reply

George
October 21, 2021 12:08 am

Why copy initialization (instead of reference) of a base object will not work?
D1 d1;
Base b{d1};
will call the base virtual table. But I was thinking, we construct a base object b, and copy assign the base portion of d1 to b; So we copy the content of __vptr from d1 into _vptr of b, and it will point to the same place as d1, i.e. to the d1 virtual table. But function call resolves to the base version.

0 Reply

Alex Author
Reply to George[15] October 21, 2021 10:59 am

When you do Base b{d1}, d1 gets sliced. b doesn't contain a D1 portion, so it doesn't make sense for b's __vptr to point to any D1 functions. Virtual tables aren't copied, they're set up when the object is created based on the object's actual type, not the initializer's type.

2 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-569504
8. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-566617
9. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-564912
10. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562930
11. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562968
12. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562979
13. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562307
14. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562138
15. https://www.learncpp.com/cpp-tutorial/the-virtual-table/#comment-562137
16. https://www.ezoic.com/what-is-ezoic/