In C++, what is a virtual base class?

Asked 16 years, 4 months ago Modified 1 year, 6 months ago Viewed 365k times

476

I want to know what a "*virtual base class*" is and what it means.

Let me show an example:

```
class Foo
{
public:
    void DoSomething() { /* ... */ }
};

class Bar : public virtual Foo
{
public:
    void DoSpecific() { /* ... */ }
};
```

c++ virtual-inheritance

Share
Improve this question
Follow

edited Mar 1, 2014 at 13:37
manlio
18.8k ● 14 ● 80 ● 133

asked Aug 22, 2008 at 1:13
popopome
12.5k ● 15 ● 45 ● 36

1 should we use virtual base classes in 'multiple inheritance' because if class A has member variable int a and class B also has member int a and class c inherits class A and B how do we decide which 'a' to use ? – Namit Sinha May 1, 2014 at 19:01

2 @NamitSinha no, virtual inheritance does *not* solve that problem. The member a would be ambiguous anyway – Ichthyo Dec 8, 2017 at 20:18

@NamitSinha Virtual inheritance isn't a magical tool to remove multiple inheritance related ambiguities. It "solves" a "problem" of having an indirect base more than once. Which is only a problem if it was intended to be shared (often but not always the case). – curiousguy Dec 22, 2019 at 8:20

11 Answers

Sorted by: Highest score (default)

595

Virtual base classes, used in virtual inheritance, is a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritance.

Consider the following scenario:

```
class A { public: void Foo() {} };
class B : public A {};
class C : public A {};
class D : public B, public C {};
```

The above class hierarchy results in the "dreaded diamond" which looks like this:

```
  A
 / \
B   C
 \ /
  D
```

An instance of D will be made up of B, which includes A, and C which also includes A. So you have two "instances" (for want of a better expression) of A.

When you have this scenario, you have the possibility of ambiguity. What happens when you do this:

```
D d;
d.Foo(); // is this B's Foo() or C's Foo() ??
```

Virtual inheritance is there to solve this problem. When you specify virtual when inheriting your classes, you're telling the compiler that you only want a single instance.

```
class A { public: void Foo() {} };
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {};
```

This means that there is only one "instance" of A included in the hierarchy. Hence

```
D d;
d.Foo(); // no longer ambiguous
```

This is a mini summary. For more information, have a read of this and this. A good example is also available here.

Share

Improve this answer

Follow

edited Jan 23, 2020 at 11:02

NoDataDumpNoContribution

10.8k ● 9 ● 67 ● 110

answered Aug 22, 2008 at 1:45

OJ.

29.4k ● 6 ● 58 ● 71

20 @Bohdan use virtual keyword as much as less, because when we use virtual keyword, a heavy weight mechanism is applied. So, your program efficiency will be reduced. – Sagar Feb 2, 2014 at 14:05

6 @OJ. Just want to know one more thing, after using virtual keyword then which instance is called like `d.Foo() // is this B's Foo() or C's Foo() (after using virtual keyword)` – Sagar Feb 2, 2014 at 14:08

5 @Viktor, it is not important, it is not belong to `B` or `C`. Just think it is a merged method actually belongs to `grandfather A` – Fredrick Gauss Feb 15, 2014 at 16:39

97 Your "dreaded diamond" diagram is confusing, although it seems to be commonly used. This is actually a diagram showing class inheritance relationships -- *not* an object layout. The confusing part is that if we do use `virtual`, then the object layout looks like the diamond; and if we do not use `virtual` then the object layout looks like a tree structure that contains two `A`s – M.M Jul 23, 2015 at 4:19

10 I have to downvote this answer for the reason outlined by M.M -- the diagram expresses the opposite of the post. – David Stone Oct 2, 2016 at 17:47

303

About the memory layout

As a side note, the problem with the Dreaded Diamond is that the base class is present multiple times. So with regular inheritance, you believe you have:

```
  A
 / \
B   C
 \ /
  D
```

But in the memory layout, you have:

```
A   A
|   |
B   C
 \ /
  D
```

This explain why when call `D::foo()`, you have an ambiguity problem. But the **real** problem comes when you want to use a data member of `A`. For example, let's say we have:

```
class A
{
    public :
       foo() ;
```

```
        int m_iValue ;
} ;
```

When you'll try to access `m_iValue` from `D`, the compiler will protest, because in the hierarchy, it'll see two `m_iValue`, not one. And if you modify one, say, `B::m_iValue` (that is the `A::m_iValue` parent of `B`), `C::m_iValue` won't be modified (that is the `A::m_iValue` parent of `C`).

This is where virtual inheritance comes handy, as with it, you'll get back to a true diamond layout, with not only one `foo()` method only, but also one and only one `m_iValue`.

What could go wrong?

Imagine:

- `A` has some basic feature.
- `B` adds to it some kind of cool array of data (for example)
- `C` adds to it some cool feature like an observer pattern (for example, on `m_iValue`).
- `D` inherits from `B` and `C`, and thus from `A`.

With normal inheritance, modifying `m_iValue` from `D` is ambiguous and this must be resolved. Even if it is, there are two `m_iValues` inside `D`, so you'd better remember that and update the two at the same time.

With virtual inheritance, modifying `m_iValue` from `D` is ok... But... Let's say that you have `D`. Through its `C` interface, you attached an observer. And through its `B` interface, you update the cool array, which has the side effect of directly changing `m_iValue`...

As the change of `m_iValue` is done directly (without using a virtual accessor method), the observer "listening" through `C` won't be called, because the code implementing the listening is in `C`, and `B` doesn't know about it...

Conclusion

If you're having a diamond in your hierarchy, it means that you have 95% probability to have done something wrong with said hierarchy.

edited Jun 15, 2023 at 11:07

Jan Schultke

38.3k ● 8 ● 87 ● 168

answered Sep 21, 2008 at 23:06

paercebal

83.2k ● 38 ● 134 ● 160

Follow

Your 'what could go wrong' is due to direct access to a base member, not due to multiple inheritance. Get rid of 'B" and you have the same problem. Basic rule of: 'if its not private, it should be virtual' avoids the problem. m_iValue is not virtual and therefor should be private – Chris Dodd Dec 17, 2009 at 15:32

4 @Chris Dodd: Not exactly. What happens with m_iValue would have happened to any symbol (*e.g. typedef, member variable, member function, cast to the base class, etc.*). This really is a multiple inheritance issue, an issue that users should be aware to use multiple inheritance correctly, instead of going the Java way and conclude "Multiple inheritance is 100% evil, let's do that with interfaces". – paercebal Oct 30, 2010 at 6:38

Hi, When we use virtual keyword, there will be only one copy of A. My question is how do we know whether it is coming from B or C? Is my question valid at all? – user875036 Jun 29, 2014 at 18:40

@user875036 : A is coming both from B and C. Indeed, virtuality changes a few things (e.g. D will call A's constructor, not B, nor C). Both B and C (and D) have a pointer to A. – paercebal Jun 30, 2014 at 16:44

4 FWIW, in case someone's wondering, member variables *cannot* be virtual -- virtual is a specifier for *functions*. SO reference: stackoverflow.com/questions/3698831/… – rholmes Sep 23, 2016 at 19:21

38

Explaining multiple-inheritance with virtual bases requires a knowledge of the C++ object model. And explaining the topic clearly is best done in an article and not in a comment box.

The best, readable explanation I found that solved all my doubts on this subject was this article: http://www.phpcompiler.org/articles/virtualinheritance.html

You really won't need to read anything else on the topic (unless you are a compiler writer) after reading that...

Share

Improve this answer

Follow

edited Jul 14, 2016 at 18:14

Jeffrey Bosboom

13.7k ● 16 ● 81 ● 94

answered Jun 13, 2009 at 22:59

lenkite

1,102 ● 13 ● 9

10

> A virtual base class is a class that cannot be instantiated : you cannot create direct object out of it.

I think you are confusing two very different things. Virtual inheritance is not the same thing as an abstract class. Virtual inheritance modifies the behaviour of function calls; sometimes it resolves function calls that otherwise would be ambiguous, sometimes it

defers function call handling to a class other than that one would expect in a non-virtual inheritance.

Share Improve this answer Follow

answered Aug 22, 2008 at 1:47

wilhelmtell

58.6k • 20 • 97 • 131

7

I'd like to add to OJ's kind clarifications.

Virtual inheritance doesn't come without a price. Like with all things virtual, you get a performance hit. There is a way around this performance hit that is possibly less elegant.

Instead of breaking the diamond by deriving virtually, you can add another layer to the diamond, to get something like this:

```
   B
  / \
D11 D12
 |   |
D21 D22
 \   /
  DD
```

None of the classes inherit virtually, all inherit publicly. Classes D21 and D22 will then hide virtual function f() which is ambiguous for DD, perhaps by declaring the function private. They'd each define a wrapper function, f1() and f2() respectively, each calling class-local (private) f(), thus resolving conflicts. Class DD calls f1() if it wants D11::f() and f2() if it wants D12::f(). If you define the wrappers inline you'll probably get about zero overhead.

Of course, if you can change D11 and D12 then you can do the same trick inside these classes, but often that is not the case.

Share

Improve this answer

Follow

edited Apr 13, 2017 at 8:13

Mathieu K.

283 • 3 • 14

answered Aug 22, 2008 at 2:03

wilhelmtell

58.6k • 20 • 97 • 131

3 This is not a matter of more or less elegant or of resolving ambiguities (you can always use explicit xxx:: specifications for that). With non-virtual inheritance, every instance of class DD has *two* independent instances of B. As soon as the class has a single non-static data member, virtual and non-virtual inheritance differ by more than just syntax. – user3489112 Jun 25, 2014 at 18:02

@user3489112 As soon as ... nothing. Virtual and non virtual inheritance differ semantically, period. – curiousguy Nov 27, 2018 at 23:44

6

In addition to what has already been said about multiple and virtual inheritance(s), there is a very interesting article on Dr Dobb's Journal: Multiple Inheritance Considered Useful

Share Improve this answer Follow

answered Sep 22, 2008 at 0:58

Luc Hermitte
32.9k ● 7 ● 72 ● 88

3

Diamond inheritance runnable usage example

This example shows how to use a virtual base class in the typical scenario: to solve diamond inheritance problems.

Consider the following working example:

main.cpp

```
#include <cassert>

class A {
    public:
        A(){}
        A(int i) : i(i) {}
        int i;
        virtual int f() = 0;
        virtual int g() = 0;
        virtual int h() = 0;
};

class B : public virtual A {
    public:
        B(int j) : j(j) {}
        int j;
        virtual int f() { return this->i + this->j; }
};

class C : public virtual A {
    public:
        C(int k) : k(k) {}
```

```
        int k;
        virtual int g() { return this->i + this->k; }
};

class D : public B, public C {
    public:
        D(int i, int j, int k) : A(i), B(j), C(k) {}
        virtual int h() { return this->i + this->j + this->k; }
};

int main() {
    D d = D(1, 2, 4);
    assert(d.f() == 3);
    assert(d.g() == 5);
    assert(d.h() == 7);
}
```

Compile and run:

```
g++ -ggdb3 -O0 -std=c++11 -Wall -Wextra -pedantic -o main.out main.cpp
./main.out
```

If we remove the `virtual` into:

```
class B : public virtual A
```

we would get a wall of errors about GCC being unable to resolve D members and methods that were inherited twice via A:

```
main.cpp:27:7: warning: virtual base ‘A’ inaccessible in ‘D’ due to ambiguity
[-Wextra]
   27 | class D : public B, public C {
      |       ^
main.cpp: In member function ‘virtual int D::h()’:
main.cpp:30:40: error: request for member ‘i’ is ambiguous
   30 |         virtual int h() { return this->i + this->j + this->k; }
      |                                        ^
main.cpp:7:13: note: candidates are: ‘int A::i’
    7 |         int i;
      |             ^
main.cpp:7:13: note:                 ‘int A::i’
main.cpp: In function ‘int main()’:
main.cpp:34:20: error: invalid cast to abstract class type ‘D’
   34 |     D d = D(1, 2, 4);
      |                    ^
main.cpp:27:7: note:   because the following virtual functions are pure within
‘D’:
   27 | class D : public B, public C {
      |       ^
main.cpp:8:21: note:    ‘virtual int A::f()’
    8 |         virtual int f() = 0;
      |                     ^
main.cpp:9:21: note:    ‘virtual int A::g()’
    9 |         virtual int g() = 0;
      |                     ^
```

```
main.cpp:34:7: error: cannot declare variable ‘d’ to be of abstract type ‘D’
   34 |     D d = D(1, 2, 4);
      |       ^
In file included from /usr/include/c++/9/cassert:44,
                 from main.cpp:1:
main.cpp:35:14: error: request for member ‘f’ is ambiguous
   35 |     assert(d.f() == 3);
      |              ^
main.cpp:8:21: note: candidates are: ‘virtual int A::f()’
    8 |         virtual int f() = 0;
      |                     ^
main.cpp:17:21: note:                 ‘virtual int B::f()’
   17 |         virtual int f() { return this->i + this->j; }
      |                     ^
In file included from /usr/include/c++/9/cassert:44,
                 from main.cpp:1:
main.cpp:36:14: error: request for member ‘g’ is ambiguous
   36 |     assert(d.g() == 5);
      |              ^
main.cpp:9:21: note: candidates are: ‘virtual int A::g()’
    9 |         virtual int g() = 0;
      |                     ^
main.cpp:24:21: note:                 ‘virtual int C::g()’
   24 |         virtual int g() { return this->i + this->k; }
      |                     ^
main.cpp:9:21: note:                 ‘virtual int A::g()’
    9 |         virtual int g() = 0;
      |                     ^
./main.out
```

Tested on GCC 9.3.0, Ubuntu 20.04.

Share

Improve this answer

Follow

edited Oct 2, 2020 at 8:38

answered Dec 7, 2016 at 19:52

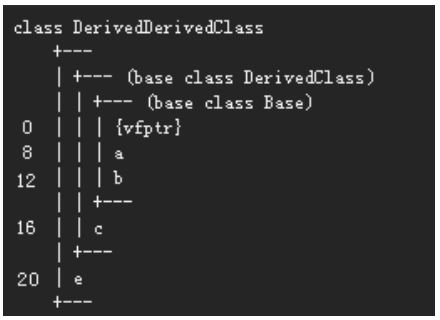
Ciro Santilli
OurBigBook.com
380k ● 116 ● 1.3k ● 1.1k

2 `assert(A::aDefault == 0);` from the main function gives me a compiling error : `aDefault is not a member of A` using gcc 5.4.0. What is it suppose to do? – clickMe Jan 20, 2017 at 11:57

@SebTu ah thanks, just something I forgot to remove from copy paste, removed it now. The example should still be meaningful without it. – Ciro Santilli OurBigBook.com Jan 20, 2017 at 13:00

Regular Inheritance

3



With typical 3 level non-diamond non-virtual-inheritance inheritance, when you instantiate a new most-derived-object, `new` is called and the size required for the object on the heap is resolved from the class type by the compiler and passed to new.

`new` has a signature:

```
_GLIBCXX_WEAK_DEFINITION void *
operator new (std::size_t sz) _GLIBCXX_THROW (std::bad_alloc)
```

And makes a call to `malloc`, returning the void pointer

This address is then passed to the constructor of the most derived object, which will immediately call the middle constructor and then the middle constructor will immediately call the base constructor. The base then stores a pointer to its virtual table at the start of the object and then its attributes after it. This then returns to the middle constructor which will store its virtual table pointer at the same location and then its attributes after the attributes that would have been stored by the base constructor. It then returns to the most derived constructor, which stores a pointer to its virtual table at the same location and and then stores its attributes after the attributes that would have been stored by the middle constructor.

Because the virtual table pointer is overwritten, the virtual table pointer ends up always being the one of the most derived class. Virtualness propagates towards the most derived class so if a function is virtual in the middle class, it will be virtual in the most derived class but not the base class. If you polymorphically cast an instance of the most derived class to a pointer to the base class then the compiler will not resolve this to an indirect call to the virtual table and instead will call the function directly `A::function()`. If a function is virtual for the type you have cast it to then it will resolve to a call into the virtual table which will always be that of the most derived class. If it is not virtual for that type then it will just call `Type::function()` and pass the object pointer to it, cast to Type.

Actually when I say pointer to its virtual table, it's actually always an offset of 16 into the virtual table.

```
vtable for Base:
        .quad   0
```

```
        .quad   typeinfo for Base
        .quad   Base::CommonFunction()
        .quad   Base::VirtualFunction()

pointer is typically to the first function i.e.

        mov     edx, OFFSET FLAT:vtable for Base+16
```

`virtual` is not required again in more-derived classes if it is virtual in a less-derived class because it propagates downwards in the direction of the most derived class. But it can be used to show that the function is indeed a virtual function, without having to check the classes it inherits's type definitions. When a function is declared virtual, from that point on, only the last implementation in the inheritance chain is used, but before that, it can still be used non-virtually if the object is cast to a type of a class before that in the inheritance chain that defines that method. It can be defined non-virtually in multiple classes before it in the chain before the virtualhood begins for a method of that name and signature, and they will use their own methods when referenced (and all classes after that definition in the chain will use that definition if they do not have their own definition, as opposed to virtual, which always uses the final definition). When a method is declared virtual, it must be implemented in that class or a more derived class in the inheritance chain for the full object that was constructed in order to be used.

`override` is another compiler guard that says that this function is overriding something and if it isn't then throw a compiler error.

`= 0` means that this is an abstract function

`final` prevents a virtual function from being implemented again in a more derived class and will make sure that the virtual table of the most derived class contains the final function of that class.

`= default` makes it explicit in documentation that the compiler will use the default implementation

`= delete` give a compiler error if a call to this is attempted

If you call a non-virtual function, it will resolve to the correct method definition without going through the virtual table. If you call a virtual-function that has its final definition in an inherited class then it will use its virtual table and will pass the subobject to it automatically if you don't cast the object pointer to that type when calling the method. If you call a virtual function defined in the most derived class on a pointer of that type then it will use its virtual table, which will be the one at the start of the object. If you call it on a pointer of an inherited type and the function is also virtual in that class then it will use the vtable pointer of that subobject, which in the case of the first subobject will be the same pointer as the most derived class, which will not contain a thunk as

the address of the object and the subobject are the same, and therefore it's just as simple as the method automatically recasting this pointer, but in the case of a 2nd sub object, its vtable will contain a non-virtual thunk to convert the pointer of the object of inherited type to the type the implementation in the most derived class expects, which is the full object, and therefore offsets the subobject pointer to point to the full object, and in the case of base subobject, will require a virtual thunk to offset the pointer to the base to the full object, such that it can be recast by the method hidden object parameter type.

Using the object with a reference operator and not through a pointer (dereference operator) breaks polymorphism and will treat virtual methods as regular methods. This is because polymorphic casting on non-pointer types can't occur due to slicing.

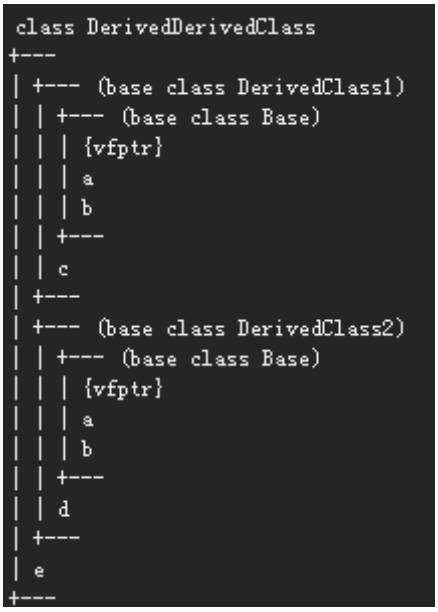
Virtual Inheritance

Consider

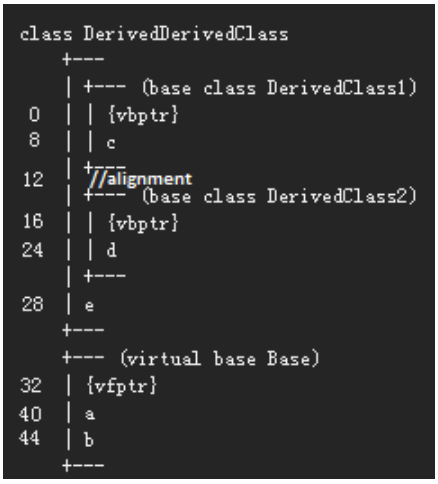
```
class Base
  {
      int a = 1;
      int b = 2;
  public:
      void virtual CommonFunction(){} ; //define empty method body
      void virtual VirtualFunction(){} ;
  };


class DerivedClass1: virtual public Base
  {
      int c = 3;
  public:
    void virtual DerivedCommonFunction(){} ;
     void virtual VirtualFunction(){} ;
  };

  class DerivedClass2 : virtual public Base
 {
     int d = 4;
 public:
     //void virtual DerivedCommonFunction(){} ;
     void virtual VirtualFunction(){} ;
     void virtual DerivedCommonFunction2(){} ;
 };

class DerivedDerivedClass :  public DerivedClass1, public DerivedClass2
 {
   int e = 5;
 public:
     void virtual DerivedDerivedCommonFunction(){} ;
     void virtual VirtualFunction(){} ;
 };

 int main () {
   DerivedDerivedClass* d = new DerivedDerivedClass;
```

```
    d->VirtualFunction();
    d->DerivedCommonFunction();
    d->DerivedCommonFunction2();
    d->DerivedDerivedCommonFunction();
    ((DerivedClass2*)d)->DerivedCommonFunction2();
    ((Base*)d)->VirtualFunction();
  }
```

Without virtually inheriting the bass class you will get an object that looks like this:



Instead of this:



I.e. there will be 2 base objects.

In the virtual diamond inheritance situation above, after `new` is called, it passes the address of the allocated space for the object to the most derived constructor `DerivedDerivedClass::DerivedDerivedClass()`, which calls `Base::Base()` first, which writes its vtable in the base's dedicated subobject, it then `DerivedDerivedClass::DerivedDerivedClass()` calls `DerivedClass1::DerivedClass1()`, which writes its virtual table pointer to its subobject as well as overwriting the base subobject's pointer at the end of the object by consulting the passed VTT, and then calls `DerivedClass1::DerivedClass1()` to do the

same, and finally `DerivedDerivedClass::DerivedDerivedClass()` overwrites all 3 pointers with its virtual table pointer for that inherited class. This is instead of (as illustrated in the 1st image above) `DerivedDerivedClass::DerivedDerivedClass()` calling `DerivedClass1::DerivedClass1()` and that calling `Base::Base()` (which overwrites the virtual pointer), returning, offsetting the address to the next subobject, calling `DerivedClass2::DerivedClass2()` and then that also calling `Base::Base()`, overwriting that virtual pointer, returning and then `DerivedDerivedClass` constructor overwriting both virtual pointers with its virtual table pointer (in this instance, the virtual table of the most derived constructor contains 2 subtables instead of 3).

The following is all compiled in debug mode -O0 so there will be redundant assembly

```
main:
.LFB8:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 24
        mov     edi, 48 //pass size to new
        call    operator new(unsigned long) //call new
        mov     rbx, rax  //move the address of the allocation to rbx
        mov     rdi, rbx  //move it to rdi i.e. pass to the call
        call    DerivedDerivedClass::DerivedDerivedClass() [complete object
constructor] //construct on this address
        mov     QWORD PTR [rbp-24], rbx  //store the address of the object on
the stack as the d pointer variable on -O0, will be optimised off on -Ofast if
the address of the pointer itself isn't taken in the code, because this address
does not need to be on the stack, it can just be passed in a register to the
subsequent methods
```

Parenthetically, if the code were `DerivedDerivedClass d = DerivedDerivedClass()`, the `main` function would look like this:

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 48 // make room for and zero 48 bytes on the stack for the
48 byte object, no extra padding required as the frame is 64 bytes with `rbp`
and return address of the function it calls (no stack params are passed to any
function it calls), hence rsp will be aligned by 16 assuming it was aligned at
the start of this frame
        mov     QWORD PTR [rbp-48], 0
        mov     QWORD PTR [rbp-40], 0
        mov     QWORD PTR [rbp-32], 0
        mov     QWORD PTR [rbp-24], 0
        mov     QWORD PTR [rbp-16], 0
        mov     QWORD PTR [rbp-8], 0
        lea     rax, [rbp-48] // load the address of the cleared 48 bytes
        mov     rdi, rax // pass the address as a pointer to the 48 bytes
cleared as the first parameter to the constructor
        call    DerivedDerivedClass::DerivedDerivedClass() [complete object
constructor]
        //address is not stored on the stack because the object is used
```

```
directly -- there is no pointer variable -- d refers to the object on the stack
as opposed to being a pointer
```

Moving back to the original example, the `DerivedDerivedClass` constructor:

```
DerivedDerivedClass::DerivedDerivedClass() [complete object constructor]:
.LFB20:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     QWORD PTR [rbp-8], rdi
.LBB5:
        mov     rax, QWORD PTR [rbp-8] // object address now in rax
        add     rax, 32 //increment address by 32
        mov     rdi, rax // move object address+32 to rdi i.e. pass to call
        call    Base::Base() [base object constructor]
        mov     rax, QWORD PTR [rbp-8] //move object address to rax
        mov     edx, OFFSET FLAT:VTT for DerivedDerivedClass+8 //move address
of VTT+8 to edx
        mov     rsi, rdx //pass VTT+8 address as 2nd parameter
        mov     rdi, rax //object address as first (DerivedClass1 subobject)
        call    DerivedClass1::DerivedClass1() [base object constructor]
        mov     rax, QWORD PTR [rbp-8] //move object address to rax
        add     rax, 16  //increment object address by 16
        mov     edx, OFFSET FLAT:VTT for DerivedDerivedClass+24  //store
address of VTT+24 in edx
        mov     rsi, rdx //pass address of VTT+24 as second parameter
        mov     rdi, rax //address of DerivedClass2 subobject as first
        call    DerivedClass2::DerivedClass2() [base object constructor]
        mov     edx, OFFSET FLAT:vtable for DerivedDerivedClass+24 //move this
to edx
        mov     rax, QWORD PTR [rbp-8] // object address now in rax
        mov     QWORD PTR [rax], rdx. //store address of vtable for
DerivedDerivedClass+24 at the start of the object
        mov     rax, QWORD PTR [rbp-8] // object address now in rax
        add     rax, 32  // increment object address by 32
        mov     edx, OFFSET FLAT:vtable for DerivedDerivedClass+120 //move this
to edx
        mov     QWORD PTR [rax], rdx  //store vtable for
DerivedDerivedClass+120 at object+32 (Base)
        mov     edx, OFFSET FLAT:vtable for DerivedDerivedClass+72 //store this
in edx
        mov     rax, QWORD PTR [rbp-8] //move object address to rax
        mov     QWORD PTR [rax+16], rdx //store vtable for
DerivedDerivedClass+72 at object+16 (DerivedClass2)
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax+28], 5 // stores e = 5 in the object
.LBE5:
        nop
        leave
        ret
```

The `DerivedDerivedClass` constructor calls `Base::Base()` with a pointer to the object offset 32. Base stores a pointer to its virtual table at the address it receives and its members after it.

```
Base::Base() [base object constructor]:
.LFB11:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi //stores address of object on stack (-
O0)
.LBB2:
        mov     edx, OFFSET FLAT:vtable for Base+16  //puts vtable for Base+16
in edx
        mov     rax, QWORD PTR [rbp-8] //copies address of object from stack to
rax
        mov     QWORD PTR [rax], rdx  //stores it address of object
        mov     rax, QWORD PTR [rbp-8] //copies address of object on stack to
rax again
        mov     DWORD PTR [rax+8], 1 //stores a = 1 in the object
        mov     rax, QWORD PTR [rbp-8] //junk from -O0
        mov     DWORD PTR [rax+12], 2  //stores b = 2 in the object
.LBE2:
        nop
        pop     rbp
        ret
```

`DerivedDerivedClass::DerivedDerivedClass()` then calls `DerivedClass1::DerivedClass1()` with a pointer to the object offset 0 and also passes the address of `VTT for DerivedDerivedClass+8`

```
DerivedClass1::DerivedClass1() [base object constructor]:
.LFB14:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi //address of object
        mov     QWORD PTR [rbp-16], rsi  //address of VTT+8
.LBB3:
        mov     rax, QWORD PTR [rbp-16]  //address of VTT+8 now in rax
        mov     rdx, QWORD PTR [rax]     //address of DerivedClass1-in-
DerivedDerivedClass+24 now in rdx
        mov     rax, QWORD PTR [rbp-8]   //address of object now in rax
        mov     QWORD PTR [rax], rdx     //store address of DerivedClass1-in-..
in the object
        mov     rax, QWORD PTR [rbp-8]  // address of object now in rax
        mov     rax, QWORD PTR [rax]    //address of DerivedClass1-in.. now
implicitly in rax
        sub     rax, 24                 //address of DerivedClass1-in-
DerivedDerivedClass+0 now in rax
        mov     rax, QWORD PTR [rax]    //value of 32 now in rax
        mov     rdx, rax                // now in rdx
        mov     rax, QWORD PTR [rbp-8]  //address of object now in rax
        add     rdx, rax                //address of object+32 now in rdx
        mov     rax, QWORD PTR [rbp-16]  //address of VTT+8 now in rax
        mov     rax, QWORD PTR [rax+8]   //derference VTT+8+8; address of
DerivedClass1-in-DerivedDerivedClass+72 (Base::CommonFunction()) now in rax
        mov     QWORD PTR [rdx], rax     //store at address object+32 (offset
to Base)
        mov     rax, QWORD PTR [rbp-8]  //store address of object in rax,
return
        mov     DWORD PTR [rax+8], 3    //store its attribute c = 3 in the
object
.LBE3:
```

```
        nop
        pop     rbp
        ret
```

```
VTT for DerivedDerivedClass:
        .quad   vtable for DerivedDerivedClass+24
        .quad   construction vtable for DerivedClass1-in-DerivedDerivedClass+24
//(DerivedClass1 uses this to write its vtable pointer)
        .quad   construction vtable for DerivedClass1-in-DerivedDerivedClass+72
//(DerivedClass1 uses this to overwrite the base vtable pointer)
        .quad   construction vtable for DerivedClass2-in-DerivedDerivedClass+24
        .quad   construction vtable for DerivedClass2-in-DerivedDerivedClass+72
        .quad   vtable for DerivedDerivedClass+120 // DerivedDerivedClass
supposed to use this to overwrite Bases's vtable pointer
        .quad   vtable for DerivedDerivedClass+72 // DerivedDerivedClass
supposed to use this to overwrite DerivedClass2's vtable pointer
//although DerivedDerivedClass uses vtable for DerivedDerivedClass+72 and
DerivedDerivedClass+120 directly to overwrite them instead of going through the
VTT

construction vtable for DerivedClass1-in-DerivedDerivedClass:
        .quad   32
        .quad   0
        .quad   typeinfo for DerivedClass1
        .quad   DerivedClass1::DerivedCommonFunction()
        .quad   DerivedClass1::VirtualFunction()
        .quad   -32
        .quad   0
        .quad   -32
        .quad   typeinfo for DerivedClass1
        .quad   Base::CommonFunction()
        .quad   virtual thunk to DerivedClass1::VirtualFunction()
construction vtable for DerivedClass2-in-DerivedDerivedClass:
        .quad   16
        .quad   0
        .quad   typeinfo for DerivedClass2
        .quad   DerivedClass2::VirtualFunction()
        .quad   DerivedClass2::DerivedCommonFunction2()
        .quad   -16
        .quad   0
        .quad   -16
        .quad   typeinfo for DerivedClass2
        .quad   Base::CommonFunction()
        .quad   virtual thunk to DerivedClass2::VirtualFunction()
vtable for DerivedDerivedClass:
        .quad   32
        .quad   0
        .quad   typeinfo for DerivedDerivedClass
        .quad   DerivedClass1::DerivedCommonFunction()
        .quad   DerivedDerivedClass::VirtualFunction()
        .quad   DerivedDerivedClass::DerivedDerivedCommonFunction()
        .quad   16
        .quad   -16
        .quad   typeinfo for DerivedDerivedClass
        .quad   non-virtual thunk to DerivedDerivedClass::VirtualFunction()
        .quad   DerivedClass2::DerivedCommonFunction2()
        .quad   -32
        .quad   0
        .quad   -32
        .quad   typeinfo for DerivedDerivedClass
```

```
        .quad   Base::CommonFunction()
        .quad   virtual thunk to DerivedDerivedClass::VirtualFunction()

virtual thunk to DerivedClass1::VirtualFunction():
        mov     r10, QWORD PTR [rdi]
        add     rdi, QWORD PTR [r10-32]
        jmp     .LTHUNK0
virtual thunk to DerivedClass2::VirtualFunction():
        mov     r10, QWORD PTR [rdi]
        add     rdi, QWORD PTR [r10-32]
        jmp     .LTHUNK1
virtual thunk to DerivedDerivedClass::VirtualFunction():
        mov     r10, QWORD PTR [rdi]
        add     rdi, QWORD PTR [r10-32]
        jmp     .LTHUNK2
non-virtual thunk to DerivedDerivedClass::VirtualFunction():
        sub     rdi, 16
        jmp     .LTHUNK3

        .set    .LTHUNK0,DerivedClass1::VirtualFunction()
        .set    .LTHUNK1,DerivedClass2::VirtualFunction()
        .set    .LTHUNK2,DerivedDerivedClass::VirtualFunction()
        .set    .LTHUNK3,DerivedDerivedClass::VirtualFunction()
```

Each inherited class has its own construction virtual table and the most derived class, `DerivedDerivedClass`, has a virtual table with a subtable for each, and it uses the pointer to the subtable to overwrite construction vtable pointer that the inherited class's constructor stored for each subobject. Each virtual method that needs a thunk (virtual thunk offsets the object pointer from the base to the start of the object and a non-virtual thunk offsets the object pointer from an inherited class's object that isn't the base object to the start of the whole object of the type `DerivedDerivedClass`). The `DerivedDerivedClass` constructor also uses a virtual table table (VTT) as a serial list of all the virtual table pointers that it needs to use and passes it to each constructor (along with the subobject address that the constructor is for), which they use to overwrite their and the base's vtable pointer.

`DerivedDerivedClass::DerivedDerivedClass()` then passes the address of the object+16 and the address of VTT for `DerivedDerivedClass+24` to `DerivedClass2::DerivedClass2()` whose assembly is identical to `DerivedClass1::DerivedClass1()` except for the line `mov     DWORD PTR [rax+8], 3` which obviously has a 4 instead of 3 for `d = 4`.

After this, it replaces all 3 virtual table pointers in the object with pointers to offsets in `DerivedDerivedClass`'s vtable to the representation for that class.

The call to `d->VirtualFunction()` in `main`:

```
        mov     rax, QWORD PTR [rbp-24] //store pointer to object (and hence
vtable pointer) in rax
```

```
        mov     rax, QWORD PTR [rax] //dereference this pointer to vtable
pointer and store virtual table pointer in rax
        add     rax, 8 // add 8 to the pointer to get the 2nd function pointer
in the table
        mov     rdx, QWORD PTR [rax] //dereference this pointer to get the
address of the method to call
        mov     rax, QWORD PTR [rbp-24] //restore pointer to object in rax (-O0
is inefficient, yes)
        mov     rdi, rax  //pass object to the method
        call    rdx
```

`d->DerivedCommonFunction();`:

```
        mov     rax, QWORD PTR [rbp-24]
        mov     rdx, QWORD PTR [rbp-24]
        mov     rdx, QWORD PTR [rdx]
        mov     rdx, QWORD PTR [rdx]
        mov     rdi, rax  //pass object to method
        call    rdx  //call the first function in the table
```

`d->DerivedCommonFunction2();`:

```
        mov     rax, QWORD PTR [rbp-24] //get the object pointer
        lea     rdx, [rax+16]  //get the address of the 2nd subobject in the
object
        mov     rax, QWORD PTR [rbp-24] //get the object pointer
        mov     rax, QWORD PTR [rax+16] // get the vtable pointer of the 2nd
subobject
        add     rax, 8  //call the 2nd function in this table
        mov     rax, QWORD PTR [rax]  //get the address of the 2nd function
        mov     rdi, rdx  //call it and pass the 2nd subobject to it
        call    rax
```

`d->DerivedDerivedCommonFunction();`:

```
        mov     rax, QWORD PTR [rbp-24] //get the object pointer
        mov     rax, QWORD PTR [rax] //get the vtable pointer
        add     rax, 16 //get the 3rd function in the first virtual table
(which is where virtual functions that that first appear in the most derived
class go, because they belong to the full object which uses the virtual table
pointer at the start of the object)
        mov     rdx, QWORD PTR [rax] //get the address of the object
        mov     rax, QWORD PTR [rbp-24]
        mov     rdi, rax  //call it and pass the whole object to it
        call    rdx
```

`((DerivedClass2*)d)->DerivedCommonFunction2();`:

```
//it casts the object to its subobject and calls the corresponding method in
its virtual table, which will be a non-virtual thunk

        cmp     QWORD PTR [rbp-24], 0
        je      .L14
```

```
        mov     rax, QWORD PTR [rbp-24]
        add     rax, 16
        jmp     .L15
.L14:
        mov     eax, 0
.L15:
        cmp     QWORD PTR [rbp-24], 0
        cmp     QWORD PTR [rbp-24], 0
        je      .L18
        mov     rdx, QWORD PTR [rbp-24]
        add     rdx, 16
        jmp     .L19
.L18:
        mov     edx, 0
.L19:
        mov     rdx, QWORD PTR [rdx]
        add     rdx, 8
        mov     rdx, QWORD PTR [rdx]
        mov     rdi, rax
        call    rdx
```

`((Base*)d)->VirtualFunction();`:

```
//it casts the object to its subobject and calls the corresponding function in
its virtual table, which will be a virtual thunk

        cmp     QWORD PTR [rbp-24], 0
        je      .L20
        mov     rax, QWORD PTR [rbp-24]
        mov     rax, QWORD PTR [rax]
        sub     rax, 24
        mov     rax, QWORD PTR [rax]
        mov     rdx, rax
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rdx
        jmp     .L21
.L20:
        mov     eax, 0
.L21:
        cmp     QWORD PTR [rbp-24], 0
        cmp     QWORD PTR [rbp-24], 0
        je      .L24
        mov     rdx, QWORD PTR [rbp-24]
        mov     rdx, QWORD PTR [rdx]
        sub     rdx, 24
        mov     rdx, QWORD PTR [rdx]
        mov     rcx, rdx
        mov     rdx, QWORD PTR [rbp-24]
        add     rdx, rcx
        jmp     .L25
.L24:
        mov     edx, 0
.L25:
        mov     rdx, QWORD PTR [rdx]
        add     rdx, 8
        mov     rdx, QWORD PTR [rdx]
        mov     rdi, rax
        call    rdx
```

Share

Improve this answer

Follow

edited Feb 20, 2021 at 22:22

answered Jun 9, 2020 at 11:41

Lewis Kelsey

4,667 ● 1 ● 39 ● 54

1

You're being a little confusing. I dont' know if you're mixing up some concepts.

You don't have a virtual base class in your OP. You just have a base class.

You did virtual inheritance. This is usually used in multiple inheritance so that multiple derived classes use the members of the base class without reproducing them.

A base class with a pure virtual function is not be instantiated. this requires the syntax that Paul gets at. It is typically used so that derived classes must define those functions.

I don't want to explain any more about this because I don't totally get what you're asking.

Share Improve this answer Follow

answered Aug 22, 2008 at 1:48

Baltimark

9,272 ● 12 ● 38 ● 35

1 A "base class" that is used in a virtual inheritance becomes a "virtual base class" (in the context of that precise inheritance). – Luc Hermitte Sep 22, 2008 at 1:15

1

It means a call to a virtual function will be forwarded to the "right" class.

C++ FAQ Lite FTW.

In short, it is often used in multiple-inheritance scenarios, where a "diamond" hierarchy is formed. Virtual inheritance will then break the ambiguity created in the bottom class, when you call function in that class and the function needs to be resolved to either class D1 or D2 above that bottom class. See the FAQ item for a diagram and details.

It is also used in *sister delegation*, a powerful feature (though not for the faint of heart). See this FAQ.

Also see Item 40 in Effective C++ 3rd edition (43 in 2nd edition).

Share
Improve this answer
Follow

edited Feb 23, 2015 at 11:58

Atul Kumar
363 ● 4 ● 16

answered Aug 22, 2008 at 1:42

wilhelmtell
58.6k ● 20 ● 97 ● 131

0

Virtual classes are **not** the same as virtual inheritance. Virtual classes you cannot instantiate, virtual inheritance is something else entirely.

Wikipedia describes it better than I can. http://en.wikipedia.org/wiki/Virtual_inheritance

Share Improve this answer Follow

answered Aug 22, 2008 at 1:52

bradtgmurray
14.3k ● 10 ● 39 ● 36

6 There is no such thing as "virtual classes" in C++. There are however "virtual base classes" which are "virtual" regarding a given inheritance. What you refer is what is officially called "abstract classes". – Luc Hermitte Sep 22, 2008 at 0:56

@LucHermitte, there are definitely virtual classes in C++. Check this: en.wikipedia.org/wiki/Virtual_class . – Rafid Nov 1, 2014 at 22:05

"error: 'virtual' can only be specified for functions". I don't know what language this is. But there is definitively no such thing as *virtual class* in C++. – Luc Hermitte Nov 2, 2014 at 23:27

This is all just a case of people using Java terminology to describe C++. I used to come across it a lot at University, where they traditionally teach Java before C++. I did a C++ course (but avoided the Java) and it used to drive me batty hearing "virtual" instead of "abstract". – Orwellophile Mar 9, 2021 at 18:28

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.