[C++中虚函数工作原理和(虚)继承类的内存占用大小计算](http://blog.csdn.net/hackbuteer1/article/details/7883531)

分类： [面试珠玑](http://blog.csdn.net/Hackbuteer1/article/category/899947)2012-08-19 17:30 31782人阅读 [评论](http://blog.csdn.net/hackbuteer1/article/details/7883531#comments)(43) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/hackbuteer1/article/details/7883531#report)

[工作](http://www.csdn.net/tag/%e5%b7%a5%e4%bd%9c)[c++](http://www.csdn.net/tag/c%2b%2b)[class](http://www.csdn.net/tag/class)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[fun](http://www.csdn.net/tag/fun)

**转载请标明出处，原文地址：[http://blog.csdn.net/hackbuteer1/article/details/7883531](http://blog.csdn.net/hackbuteer1/article/details/7883531" \t "_blank)  
一、虚函数的工作原理**  
      虚函数的实现要求对象携带额外的信息，这些信息用于在运行时确定该对象应该调用哪一个虚函数。典型情况下，这一信息具有一种被称为 vptr（virtual table pointer，虚函数表指针）的指针的形式。vptr 指向一个被称为 vtbl（virtual table，虚函数表）的函数指针数组，每一个包含虚函数的类都关联到 vtbl。当一个对象调用了虚函数，实际的被调用函数通过下面的步骤确定：找到对象的 vptr 指向的 vtbl，然后在 vtbl 中寻找合适的函数指针。  
      虚拟函数的地址翻译取决于对象的内存地址，而不取决于数据类型(编译器对函数调用的合法性检查取决于数据类型)。如果类定义了虚函数，该类及其派生类就要生成一张虚拟函数表，即vtable。而在类的对象地址空间中存储一个该虚表的入口，占4个字节，这个入口地址是在构造对象时由编译器写入的。所以，由于对象的内存空间包含了虚表入口，编译器能够由这个入口找到恰当的虚函数，这个函数的地址不再由数据类型决定了。故对于一个父类的对象指针，调用虚拟函数，如果给他赋父类对象的指针，那么他就调用父类中的函数，如果给他赋子类对象的指针，他就调用子类中的函数(取决于对象的内存地址)。  
      虚函数需要注意的大概就是这些个地方了，之前在More effective C++上好像也有见过，不过这次在Visual C++权威剖析这本书中有了更直白的认识，这本书名字很牛逼，看看内容也就那么回事，感觉名不副实，不过说起来也是有其独到之处的，否则也没必要出这种书了。  
      每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就会为这个类创建一个虚函数表（VTABLE）保存该类所有虚函数的地址，其实这个VTABLE的作用就是保存自己类中所有虚函数的地址，可以把VTABLE形象地看成一个函数指针数组，这个数组的每个元素存放的就是虚函数的地址。在每个带有虚函数的类 中，编译器秘密地置入一指针，称为v p o i n t e r（缩写为V P T R），指向这个对象的V TA B L E。 当构造该派生类对象时，其成员VPTR被初始化指向该派生类的VTABLE。所以可以认为VTABLE是该类的所有对象共有的，在定义该类时被初始化；而VPTR则是每个类对象都有独立一份的，且在该类对象被构造时被初始化。  
      通过基类指针做虚函数调 用时（也就是做多态调用时），编译器静态地插入取得这个V P T R，并在V TA B L E表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生。为每个类设置V TA B L E、初始化V P T R、为虚函数调用插入代码，所有这些都是自动发生的，所以我们不必担心这些。

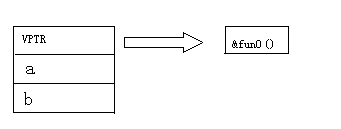
**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. #include<iostream>
2. **using** **namespace** std;
4. **class** A
5. {
6. **public**:
7. **virtual** **void** fun1()
8. {
9. cout << "A::fun1()" << endl;
10. }
11. **virtual** **void** fun2()
12. {
13. cout << "A::fun2()" << endl;
14. }
15. };
17. **class** B : **public** A
18. {
19. **public**:
20. **void** fun1()
21. {
22. cout << "B::fun1()" << endl;
23. }
24. **void** fun2()
25. {
26. cout << "B::fun2()" << endl;
27. }
28. };
30. **int** main()
31. {
32. A \*pa = **new** B;
33. pa->fun1();
34. **delete** pa;
36. system("pause");
37. **return** 0;
38. }

             毫无疑问，调用了B::fun1()，但是B::fun1()不是像普通函数那样直接找到函数地址而执行的。真正的执行方式是：首先取出pa指针所指向的对象的vptr的值，这个值就是vtbl的地址，由于调用的函数B::fun1()是第一个虚函数，所以取出vtbl第一个表项里的值，这个值就是B::fun1()的地址了，最后调用这个函数。因此只要vptr不同，指向的vtbl就不同，而不同的vtbl里装着对应类的虚函数地址，所以这样虚函数就可以完成它的任务，多态就是这样实现的。  
      而对于class A和class B来说，他们的vptr指针存放在何处？其实这个指针就放在他们各自的实例对象里。由于class A和class B都没有数据成员，所以他们的实例对象里就只有一个vptr指针。  
      **虚拟函数使用的缺点  
　　优点讲了一大堆，现在谈一下缺点，虚函数最主要的缺点是执行效率较低，看一看虚拟函数引发的多态性的实现过程，你就能体会到其中的原因，另外就是由于要携带额外的信息（VPTR），所以导致类多占的内存空间也会比较大，对象也是一样的。**  
     含有虚函数的对象在内存中的结构如下:

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** A
2. {
3. **private**:
4. **int** a;
5. **int** b;
6. **public**:
7. **virtual** **void** fun0()
8. {
9. cout<<"A::fun0"<<endl;
10. }
11. };

  
1、直接继承  
那我们来看看编译器是怎么建立VPTR指向的这个虚函数表的，先看下面两个类：

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** base
2. {
3. **private**:
4. **int** a;
5. **public**:
6. **void** bfun()
7. {
8. }
9. **virtual** **void** vfun1()
10. {
11. }
12. **virtual** **void** vfun2()
13. {
14. }
15. };
17. **class** derived : **public** base
18. {
19. **private**:
20. **int** b;
21. **public**:
22. **void** dfun()
23. {
24. }
25. **virtual** **void** vfun1()
26. {
27. }
28. **virtual** **void** vfun3()
29. {
30. }
31. };

两个类的VPTR指向的虚函数表（VTABLE）分别如下：  
base类  
                     ——————  
VPTR——>    |&base::vfun1 |  
                      ——————  
                    |&base::vfun2 |  
                    ——————  
        
derived类  
                      ———————  
VPTR——>    |&derived::vfun1 |  
                     ———————  
                   |&base::vfun2     |  
                    ———————  
                   |&derived::vfun3 |  
                    ———————  
      每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个VTABLE，如上图所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为virtual的函数的地址。如果在这个派生类中没有对在基类中声明为virtual的函数进行重新定义，编译器就使用基类 的这个虚函数地址。（在derived的VTABLE中，vfun2的入口就是这种情况。）然后编译器在这个类中放置VPTR。**当使用简单继承时，对于每个对象只有一个VPTR**。VPTR必须被初始化为指向相应的VTABLE，这在构造函数中发生。  
       一旦VPTR被初始化为指向相应的VTABLE，对象就"知道"它自己是什么类型。但只有当虚函数被调用时这种自我认知才有用。   
       没有虚函数类对象的大小正好是数据成员的大小，包含有一个或者多个虚函数的类对象编译器向里面插入了一个VPTR指针(void \*)，指向一个存放函数地址的表就是我们上面说的VTABLE，这些都是编译器为我们做的我们完全可以不关心这些。所以有虚函数的类对象的大小是数据成员的大小加上一个VPTR指针(void \*)的大小。  
  
总结一下VPTR 和 VTABLE 和类对象的关系:  
       每一个具有虚函数的类都有一个虚函数表VTABLE，里面按在类中声明的虚函数的顺序存放着虚函数的地址，这个虚函数表VTABLE是这个类的所有对象所共有的，也就是说无论用户声明了多少个类对象，但是这个VTABLE虚函数表只有一个。  
       在每个具有虚函数的类的对象里面都有一个VPTR虚函数指针，这个指针指向VTABLE的首地址，每个类的对象都有这么一种指针。  
2、虚继承  
     这个是比较不好理解的，对于虚继承，若派生类有自己的虚函数，则它本身需要有一个虚指针，指向自己的虚表。另外，派生类虚继承父类时，首先要通过加入一个虚指针来指向父类，因此有可能会有两个虚指针。  
**二、(虚)继承类的内存占用大小**  
     首先，平时所声明的类只是一种类型定义，它本身是没有大小可言的。 因此，如果用sizeof运算符对一个类型名操作，那得到的是具有该类型实体的大小。  
计算一个类对象的大小时的规律：  
    1、空类、单一继承的空类、多重继承的空类所占空间大小为：1（字节，下同）；  
    2、一个类中，虚函数本身、成员函数（包括静态与非静态）和静态数据成员都是不占用类对象的存储空间的；  
    3、因此一个对象的大小≥所有非静态成员大小的总和；   
    4、当类中声明了虚函数（不管是1个还是多个），那么在实例化对象时，编译器会自动在对象里安插一个指针vPtr指向虚函数表VTable；  
    5、虚承继的情况：由于涉及到虚函数表和虚基表，会同时增加一个（多重虚继承下对应多个）vfPtr指针指向虚函数表vfTable和一个vbPtr指针指向虚基表vbTable，这两者所占的空间大小为：8（或8乘以多继承时父类的个数）；  
    6、在考虑以上内容所占空间的大小时，还要注意编译器下的“补齐”padding的影响，即编译器会插入多余的字节补齐；  
    7、类对象的大小=各非静态数据成员（包括父类的非静态数据成员但都不包括所有的成员函数）的总和+ vfptr指针(多继承下可能不止一个)+vbptr指针(多继承下可能不止一个)+编译器额外增加的字节。  
示例一：含有普通继承

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** A
2. {
3. };
5. **class** B
6. {
7. **char** ch;
8. **virtual** **void** func0()  {  }
9. };
11. **class** C
12. {
13. **char** ch1;
14. **char** ch2;
15. **virtual** **void** func()  {  }
16. **virtual** **void** func1()  {  }
17. };
19. **class** D: **public** A, **public** C
20. {
21. **int** d;
22. **virtual** **void** func()  {  }
23. **virtual** **void** func1()  {  }
24. };
26. **class** E: **public** B, **public** C
27. {
28. **int** e;
29. **virtual** **void** func0()  {  }
30. **virtual** **void** func1()  {  }
31. };
33. **int** main(**void**)
34. {
35. cout<<"A="<<**sizeof**(A)<<endl;    //result=1
36. cout<<"B="<<**sizeof**(B)<<endl;    //result=8
37. cout<<"C="<<**sizeof**(C)<<endl;    //result=8
38. cout<<"D="<<**sizeof**(D)<<endl;    //result=12
39. cout<<"E="<<**sizeof**(E)<<endl;    //result=20
40. **return** 0;
41. }

前面三个A、B、C类的内存占用空间大小就不需要解释了，注意一下内存对齐就可以理解了。  
求sizeof(D)的时候，需要明白，首先VPTR指向的虚函数表中保存的是类D中的两个虚函数的地址，然后存放基类C中的两个数据成员ch1、ch2，注意内存对齐，然后存放数据成员d，这样4+4+4=12。  
求sizeof(E)的时候，首先是类B的虚函数地址，然后类B中的数据成员，再然后是类C的虚函数地址，然后类C中的数据成员，最后是类E中的数据成员e，同样注意内存对齐，这样4+4+4+4+4=20。  
示例二：含有虚继承

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** CommonBase
2. {
3. **int** co;
4. };
6. **class** Base1: **virtual** **public** CommonBase
7. {
8. **public**:
9. **virtual** **void** print1() {  }
10. **virtual** **void** print2() {  }
11. **private**:
12. **int** b1;
13. };
15. **class** Base2: **virtual** **public** CommonBase
16. {
17. **public**:
18. **virtual** **void** dump1() {  }
19. **virtual** **void** dump2() {  }
20. **private**:
21. **int** b2;
22. };
24. **class** Derived: **public** Base1, **public** Base2
25. {
26. **public**:
27. **void** print2() {  }
28. **void** dump2() {  }
29. **private**:
30. **int** d;
31. };

sizeof(Derived)=32，其在内存中分布的情况如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** Derived size(32):
2. +---
3. | +--- (base **class** Base1)
4. | | {vfptr}
5. | | {vbptr}
6. | | b1
7. | +---
8. | +--- (base **class** Base2)
9. | | {vfptr}
10. | | {vbptr}
11. | | b2
12. | +---
13. | d
14. +---
15. +--- (**virtual** base CommonBase)
16. | co
17. +---

示例3：

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. **class** A
2. {
3. **public**:
4. **virtual** **void** aa() {  }
5. **virtual** **void** aa2() {  }
6. **private**:
7. **char** ch[3];
8. };
10. **class** B: **virtual** **public** A
11. {
12. **public**:
13. **virtual** **void** bb() {  }
14. **virtual** **void** bb2() {  }
15. };
17. **int** main(**void**)
18. {
19. cout<<"A's size is "<<**sizeof**(A)<<endl;
20. cout<<"B's size is "<<**sizeof**(B)<<endl;
21. **return** 0;
22. }

执行结果：A's size is 8  
              B's size is 16  
      说明：对于虚继承，类B因为有自己的虚函数，所以它本身有一个虚指针，指向自己的虚表。另外，类B虚继承类A时，首先要通过加入一个虚指针来指向父类A，然后还要包含父类A的所有内容。因此是4+4+8=16。  
  
两种多态实现机制及其优缺点  
除了c++的这种多态的实现机制之外，还有另外一种实现机制，也是查表，不过是按名称查表，是smalltalk等语言的实现机制。这两种方法的优缺点如下：  
（1）、按照绝对位置查表，这种方法由于编译阶段已经做好了索引和表项(如上面的call \*(pa->vptr[1]） )，所以运行速度比较快；缺点是：当A的virtual成员比较多（比如1000个），而B重写的成员比较少（比如2个），这种时候，B的vtableＢ的剩下的998个表项都是放Ａ中的ｖｉｒｔｕａｌ成员函数的指针，如果这个派生体系比较大的时候，就浪费了很多的空间。  
比如：ＧＵＩ库，以ＭＦＣ库为例，ＭＦＣ有很多类，都是一个继承体系；而且很多时候每个类只是１，２个成员函数需要在派生类重写，如果用Ｃ＋＋的虚函数机制，每个类有一个虚表，每个表里面有大量的重复，就会造成空间利用率不高。于是ＭＦＣ的消息映射机制不用虚函数，而用第二种方法来实现多态，那就是：  
（２）、按照函数名称查表，这种方案可以避免如上的问题；但是由于要比较名称，有时候要遍历所有的继承结构，时间效率性能不是很高。（关于ＭＦＣ的消息映射的实现，看下一篇文章）  
３、总结：  
如果继承体系的基类的virtual成员不多，而且在派生类要重写的部分占了其中的大多数时候，用Ｃ＋＋的虚函数机制是比较好的；  
但是如果继承体系的基类的virtual成员很多，或者是继承体系比较庞大的时候，而且派生类中需要重写的部分比较少，那就用名称查找表，这样效率会高一些，很多的ＧＵＩ库都是这样的，比如ＭＦＣ，ＱＴ。  
ＰＳ：其实，自从计算机出现之后，时间和空间就成了永恒的主题，因为两者在98%的情况下都无法协调，此长彼消；这个就是计算机科学中的根本瓶颈之所在。软件科学和算法的发展，就看能不能突破这对时空权衡了。呵呵。。  
何止计算机科学如此，整个宇宙又何尝不是如此呢？最基本的宇宙之谜，还是时间和空间。  
  
C++如何不用虚函数实现多态   
可以考虑使用函数指针来实现多态

**[cpp]** [view plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7883531)

1. #include<iostream>
2. **using** **namespace** std;
4. **typedef** **void** (\*fVoid)();
6. **class** A
7. {
8. **public**:
9. **static** **void** test()
10. {
11. printf("hello A\n");
12. }
14. fVoid print;
16. A()
17. {
18. print = A::test;
19. }
20. };
22. **class** B : **public** A
23. {
24. **public**:
25. **static** **void** test()
26. {
27. printf("hello B\n");
28. }
30. B()
31. {
32. print = B::test;
33. }
34. };

37. **int** main(**void**)
38. {
39. A aa;
40. aa.print();
42. B b;
43. A\* a = &b;
44. a->print();
46. **return** 0;
47. }