



[首页](#)
[最新文章](#)
[IT 职场](#)
[前端](#)
[后端](#)
[移动端](#)
[数据库](#)
[运维](#)
[其他技术](#)

- 导航条 -

[伯乐在线](#) > [首页](#) > [所有文章](#) > [C/C++](#) > 浅析 C++ 继承与派生

浅析 C++ 继承与派生

2016/12/03 · [C/C++](#) · [C++](#), [派生](#), [继承](#)

分享到： ² 原文出处：[Fireplusplus](#)

测试环境：

Target: x86_64-linux-gnu

gcc version 5.3.1 20160413 (Ubuntu 5.3.1-14ubuntu2.1)

定义

要分析继承，首先当然要知道什么是继承：继承是面向对象程序设计中使用代码可以复用的最重要的手段，它允许程序员在原有类特性的基础上进行扩展，增加功能。这样产生的新类，就叫做派生类（子类）。继承呈现了面向对象程序设计的层次结构，体现了由简单到复杂的认知过程。

继承的格式

class 子类名 : 继承权限 基类名

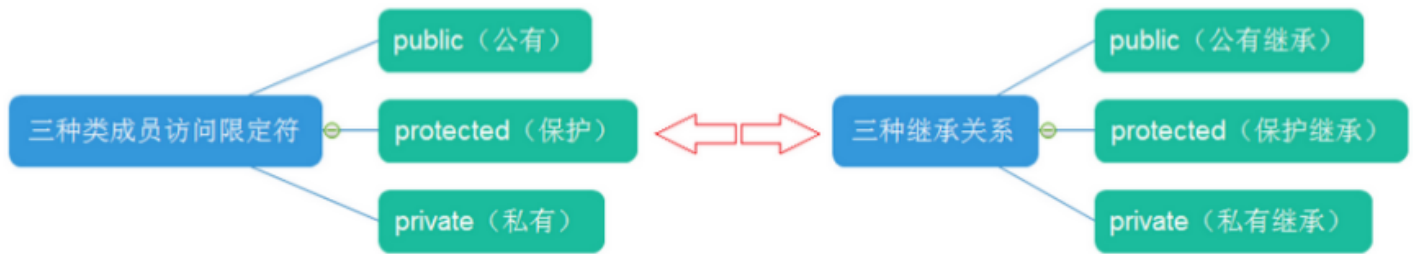
比如下面分别定义了两个类：

```
1 class A
2 {
3 public:
4     int pub;
5 protected:
6     int pro;
7 private:
8     int pri;
9 };
10
11 class B: public A
12 {
13 };
```

如上我们就说类B继承了类A，类B叫做类A的派生类或者子类，A类叫做B类的基类或者父类。

继承关系&访问限定符

之前学习类的成员访问限定符的时候都知道public, protected, private 这三种访问限定符的作用，public修饰的类成员可以在类外被访问，而protected与private则不可以。这三种访问权限又对应这三种继承关系：



继承关系可以影响子类中继承自父类的成员变量的访问权限，还是在上个栗子的基础上，我们定义一个B类对象进行如下操作；

```
1 int main()
2 {
3     B b1;
4     b1.pub;
5     b1.pro;
6     b1.pri;
7
8
9     return 0;
10 }
```

编译则会报错：

```
gaoliang@acer: ~/code/cpp/inherit
gaoliang@acer:~/code/cpp/inherit$ g++ inherit.cpp
inherit.cpp: In function 'int main()':
inherit.cpp:43:6: error: 'int A::pro' is protected
    int pro;
    ^
inherit.cpp:156:5: error: within this context
    b1.pro;
    ^
inherit.cpp:45:6: error: 'int A::pri' is private
    int pri;
    ^
inherit.cpp:157:5: error: within this context
    b1.pri;
    ^
gaoliang@acer:~/code/cpp/inherit$
```

会提示pro与pri变量访问权限分别为protected和private，我们不能在类外使用它们。类似的，在B中定义这样一个成员函数：

```
1 class B: public A
2 {
3     void fun()
4     {
5         cout<<pub;
6         cout<<pro;
7         cout<<pri;
8     }
9 };
```

会报这样的错：

```
gaoliang@acer:~/code/cpp/inherit$ g++ inherit.cpp
inherit.cpp: In member function 'void B::fun()':
inherit.cpp:45:6: error: 'int A::pri' is private
    int pri;
    ^
inherit.cpp:69:9: error: within this context
    cout<<pri;
    ^
gaoliang@acer:~/code/cpp/inherit$
```

即基类中的私有成员在子类中是不可见的。关于三种继承方式的成员访问权限总结如下表：

继承方式	基类的 public 成员	基类的 protected 成员	基类的 private 成员	继承引起的访问控制关系变化概括
public 继承	仍为 public 成员	仍为 protected 成员	不可见	基类的非私有成员在子类的访问属性都不变
protected 继承	变为 protected 成员	变为 protected 成员	不可见	基类的非私有成员都成为子类的保护成员
private 继承	变为 private 成员	变为 private 成员	不可见	基类中的非私有成员都成为子类的私有成员

总结:

1. 基类的 private 成员在派生类中是不能被访问的,如果基类成员不想在类外直接被访问,但需要在派生类中能访问,就定义为 protected 。可以看出保护成员限定符是因继承才出现的。
2. public继承是一个接口继承,保持is-a原则,每个父类可用的成员对子类也可用,因为每个子类对象也都是一个父类对象。
3. protected/private继承是一个实现继承,基类的部分成员并非完全成为子类接口的一部分,是 has-a 的关系原则,所以非特殊情况下不会使用这两种继承关系,在绝大多数的场景下使用的都是公有继承。
4. 不管是哪种继承方式,在派生类内部都可以访问基类的公有成员和保护成员,基类的私有成员存在但是在子类中不可见(不能访问)。
5. 使用关键字class时默认的继承方式是private,使用struct时默认的继承方式是public,不过最好显示的写出继承方式。
6. 在实际运用中一般使用都是public继承,极少场景下才会使用protected/private继承

继承关系中构造/析构函数调用顺序

在现有类的基础上添加如下的构造与析构函数：

```
1 class A
2 {
3 public:
4     A()
5     {
6         cout<<"A()"<<endl;
7     }
8
9     ~A()
10    {
11        cout<<"~A()"<<endl;
12    }
13
14 public:
15     int pub;
16 protected:
17     int pro;
18 private:
19     int pri;
20 };
21
22 class B: public A
23 {
24 public:
25
```

```

28     cout<<"B()"<<endl;
29 }
30
31 ~B()
32 {
33     cout<<"~B()"<<endl;
34 }
35 };

```

然后，在main函数中定义一个类B的对象：B b; 编译运行，看看输出语句的顺序：

```

gaoliang@acer: ~/code/cpp/inherit
gaoliang@acer:~/code/cpp/inherit$ ./a.out
A()
B()
~B()
~A()
gaoliang@acer:~/code/cpp/inherit$

```

先基类构造，后子类构造；析构的时候先析构子类，后析构基类。依旧和以前一样，先构造的后析构（因为在栈上）。

让我们走进几行代码的反汇编世界：

```

42         B b;
1: x/3i $pc
=> 0x4009ce <main()+24>:      lea     -0x30(%rbp),%rax
    0x4009d2 <main()+28>:      mov     %rax,%rdi
    0x4009d5 <main()+31>:      callq  0x400ab4 <B::B(>

```

这是程序现在运行到了b的定义语句。=> 所指，是当前运行的汇编语句。可以看到，第三条汇编语句调用了B类的构造函数。咦？怎么跟我们刚刚看到的顺序不太一样！不急，先往下看。直接 ni 运行到第三条汇编，然后用 si 命令跟进去：

```

(gdb)
0x0000000000400ab9      30      B()
2: x/5i $pc
=> 0x400ab9 <B::B()+5>: sub     $0x18,%rsp
    0x400abd <B::B()+9>: mov     %rdi,-0x18(%rbp)
    0x400ac1 <B::B()+13>: mov     -0x18(%rbp),%rax
    0x400ac5 <B::B()+17>: mov     %rax,%rdi
    0x400ac8 <B::B()+20>: callq  0x400a5c <A::A(>

```

可以看到，程序在正式进入B类的构造函数之前，先调用了A类的构造函数，照这么来看，可以推测出是编译器自动的在B类的构造函数的初始化列表位置调用了A类的构造函数。还是让我们把程序看完：

```

8         A()
2: x/5i $pc
=> 0x400a5c <A::A(>:      push    %rbp
    0x400a5d <A::A()+1>: mov     %rsp,%rbp
    0x400a60 <A::A()+4>: sub     $0x10,%rsp
    0x400a64 <A::A()+8>: mov     %rdi,-0x8(%rbp)
    0x400a68 <A::A()+12>: mov     $0x400bf4,%esi

```

果然，又进入了类A的构造函数。

```

B::B (this=0x7fffffffdd30) at test.cpp:32
32         cout<<"B()"<<endl;
2: x/5i $pc
=> 0x400acd <B::B()+25>:      mov     $0x400bfd,%esi
    0x400ad2 <B::B()+30>:      mov     $0x602080,%edi
    0x400ad7 <B::B()+35>:

```

从A类构造函数出来后，才正式进入类B构造函数。

```

=> 0x4009e3 <main()+45>:      mov     %rax,%rdi
0x4009e6 <main()+48>:      callq   0x400b0c <B::~B()>
0x4009eb <main()+53>:      mov     %ebx,%eax
0x4009ed <main()+55>:      mov     -0x18(%rbp),%rdx
0x4009f1 <main()+59>:      xor     %fs:0x28,%rdx

```

出main函数作用域时，先调用了B类的构造函数

```

0x0000000000400b39      36      {
2: x/5i $pc
=> 0x400b39 <B::~B()+45>:      mov     %rax,%rdi
0x400b3c <B::~B()+48>:      callq   0x400a88 <A::~A()>
0x400b41 <B::~B()+53>:      jmp     0x400b5d <B::~B()+81>
0x400b43 <B::~B()+55>:      mov     %rax,%rbx
0x400b46 <B::~B()+58>:      mov     -0x18(%rbp),%rax

```

在B类构造函数的末尾调用了A类构造函数。整个过程与我们看到的输出信息一致。

如果类B中还有一个成员变量是一个类对象，那么构造与析构调用顺序又是怎样？

```

1  class T
2  {
3
4  public:
5      T(int i = 1)
6      {
7          cout<<"T()"<<endl;
8      }
9
10     ~T()
11     {
12         cout<<"~T()"<<endl;
13     }
14 };
15
16 class A
17 {
18 public:
19     A()
20     {
21         cout<<"A()"<<endl;
22     }
23
24     ~A()
25     {
26         cout<<"~A()"<<endl;
27     }
28
29 public:
30     int pub;
31 protected:
32     int pro;
33 private:
34     int pri;
35 };
36
37 class B: public A
38 {
39 public:
40
41     B()
42     {
43         cout<<"B()"<<endl;
44     }
45
46     ~B()
47     {
48         cout<<"~B()"<<endl;
49     }
50 public:
51     T t;
52 };

```

还是刚刚的main函数，在运行一下程序：

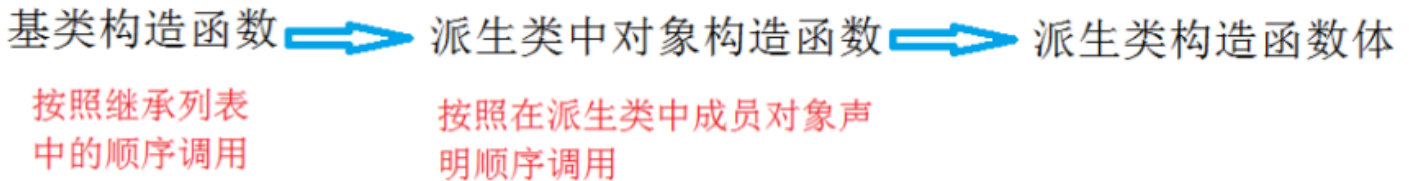
```

T()
B()
~B()
~T()
~A()

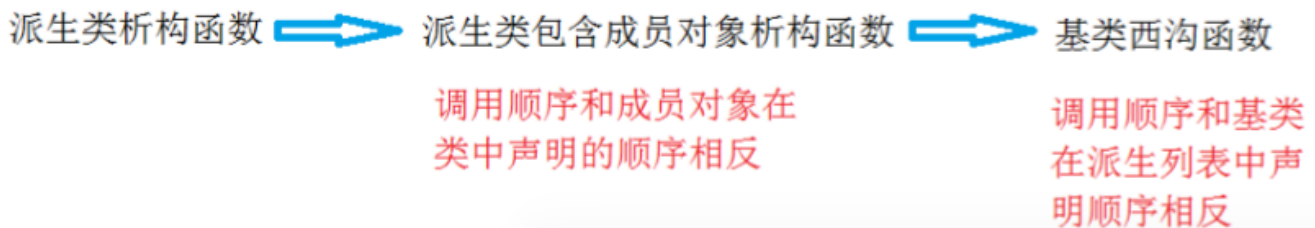
```

很明显，先调用基类构造，然后是成员对象的构造函数，最后是该类自身的构造函数，析构函数顺序则相反。具体的汇编代码就不演示了。总结一下：

【继承关系中构造函数调用顺序】



【继承关系中析构函数调用过程】



【说明】

- 1、基类没有缺省构造函数,派生类必须要在初始化列表中显式给出基类名和参数列表。
- 2、基类没有定义构造函数,则派生类也可以不用定义,全部使用缺省构造函数。
- 3、基类定义了带有形参构造函数,派生类就一定要定义构造函数。

继承体系中的作用域

1. 基类和派生类是不同的作用域
2. 同名隐藏：子类 and 父类中有同名成员时,子类成员将屏蔽父类对成员的直接访问。(在子类成员函数中,可以使用 `基类::基类成员` 访问父类成员)
3. 在实际中在继承体系里面最好不要定义同名的成员

```

1  class A
2  {
3  public:
4      int pub;
5  };
6
7  class B: public A
8  {
9  public:
10     int pub;
11 };
12
13 int main()
14 {
15     B b;
16     b.pub = 1;    //访问的是派生类的成员变量，基类同名被隐藏
17     b.A::pub = 2; //指明作用域，访问基类成员变量
18
19     return 0;
20 }

```

继承与转换—赋值兼容规则—（前提：public继承）

1. 子类对象可以赋值给父类对象
2. 父类对象不能赋值给子类对象

4. 子类的指针/引用不能指向父类对象(但可以通过强制类型转换完成)

友元与继承

友元关系不能继承,也就是说基类友元不能访问子类私有和保护成员。

```
1 class Person
2 {
3     friend void Display(Person &p , Student&s);
4 protected :
5     string _name ;
6 };
7
8 class Student: public Person
9 {
10 protected :
11     int _stuNum ;
12 };
13
14 void Display(Person &p , Student &s)
15 {
16     cout<<p._name<<endl;
17     cout<<s._name<<endl;
18     cout<<s._stuNum<<endl; //error
19 }
20 int main()
21 {
22     Person p;
23     Student s;
24     Display (p, s);
25     return 0;
26 }
```

继承与静态成员

基类定义了static成员,则整个继承体系里面只有一个这样的成员。无论派生出多少个子类,都只有一个static成员实例。如下:

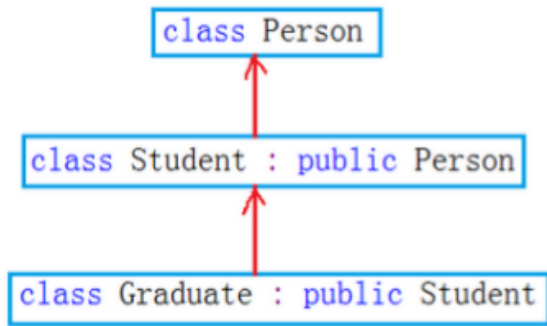
```
1 class A
2 {
3 public:
4     static int i;    //注意这里只是声明
5 };
6
7 int A::i = 0;
8
9 class B : public A
10 {};
11
12 int main()
13 {
14     A a;
15     B b;
16
17     cout<<"a.i="<<a.i<<" "<<"b.i="<<b.i<<endl;
18     a.i++;
19     b.i++;
20     cout<<"a.i="<<a.i<<" "<<"b.i="<<b.i<<endl;
21
22     return 0;
23 }
```

输出:

```
gaoliang@acer:~/code/cpp/inherit$ ./a.out
a.i=0 b.i=0
a.i=2 b.i=2
```

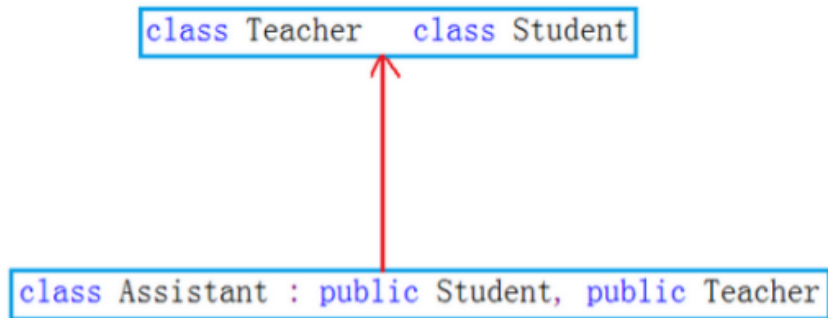
单继承&多继承&菱形继承

【单继承】

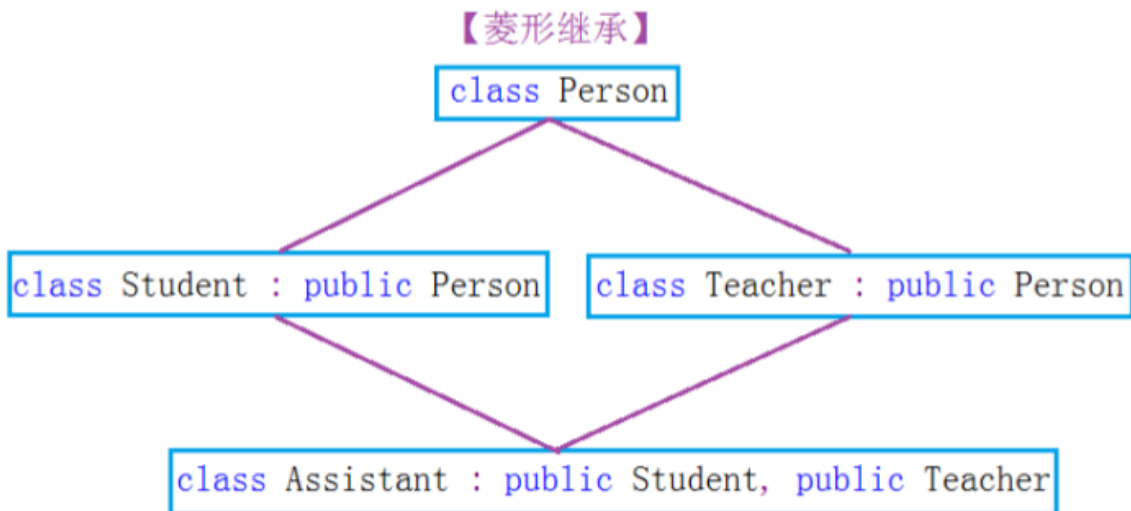


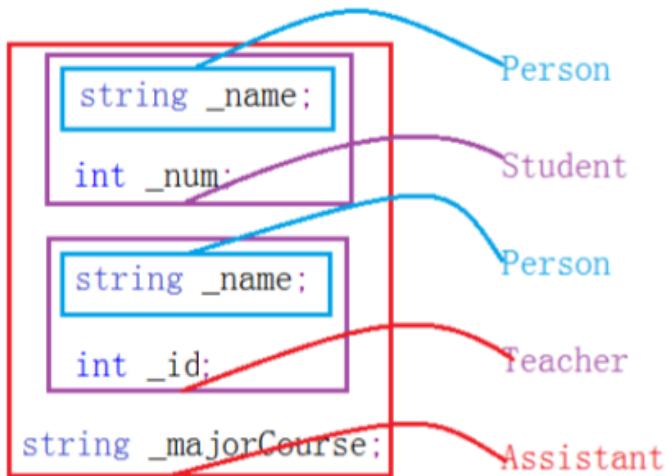
【多继承】

一个子类有两个或以上直接父类时称这个继承关系为多继承。



【菱形继承】





Assistant的对象中有两份Person成员
菱形继承存在二义性和数据冗余的问题

例：

```

1 class Person
2 {
3 public :
4     string _name ; // 姓名
5 };
6 class Student : public Person
7 {
8     protected :
9     int _num ; //学号
10 };
11 class Teacher : public Person
12 {
13     protected :
14     int _id ; // 职工编号
15 };
16 class Assistant : public Student, public Teacher
17 {
18     protected :
19     string _majorCourse ; // 主修课程
20 };
21 void Test ()
22 {
23     // 显示指定访问哪个父类的成员
24     Assistant a ;
25     a.Student::_name = "xxx";
26     a.Teacher::_name = "yyy";
27 }

```

看一下菱形继承的构造与析构函数调用顺序：（main函数中创建了一个D类对象）

B类和C类继承A类，D类继承B和C类：

```

1 class A
2 {
3 public:
4     A(){cout<<"A()"<<endl;}
5
6     ~A(){cout<<"~A()"<<endl;}
7 };
8
9 class B: public A
10 {
11 public:
12
13     B(){cout<<"B()"<<endl;}
14

```

```

17
18 class C : public A
19 {
20 public:
21     C(){cout<<"C()"<<endl;}
22
23     ~C(){cout<<"~C()"<<endl;}
24 };
25
26 class D : public B, public C
27 {
28 public:
29     D(){cout<<"D()"<<endl;}
30
31     ~D(){cout<<"~D()"<<endl;}
32 };

```

```

gaoliang@acer: ~/code/cpp/inherit
gaoliang@acer:~/code/cpp/inherit$ ./a.out
A()
B()
A()
C()
D()
~D()
~C()
~A()
~B()
~A()
gaoliang@acer:~/code/cpp/inherit$

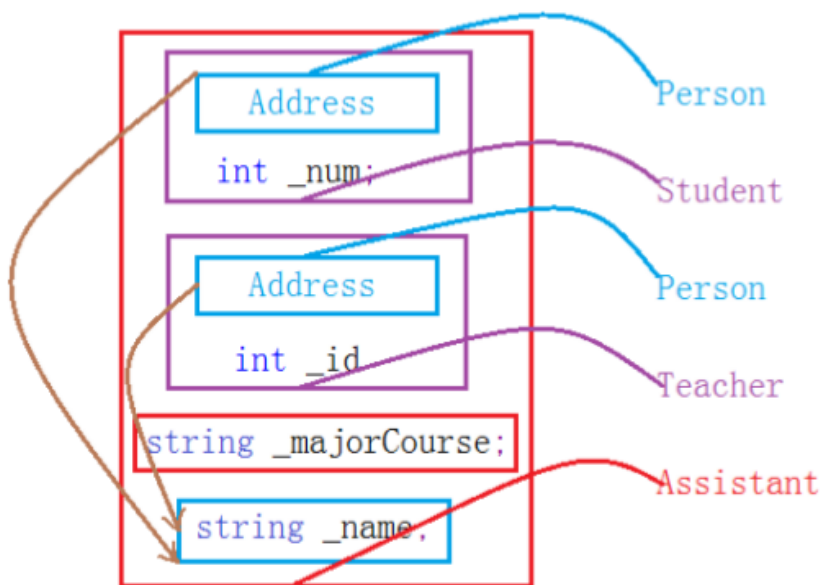
```

对照对象模型来看会很清楚。

虚继承—解决菱形继承的二义性和数据冗余的问题

1. 虚继承解决了在菱形继承体系里面子类对象包含多份父类对象的数据冗余&浪费空间的问题。
2. 虚继承体系看起来好复杂,在实际应用我们通常不会定义如此复杂的继承体系。一般不到万不得已都不要定义菱形结构的虚继承体系结构,因为使用虚继承解决数据冗余问题也带来了性能上的损耗。

【菱形虚拟继承对象模型】



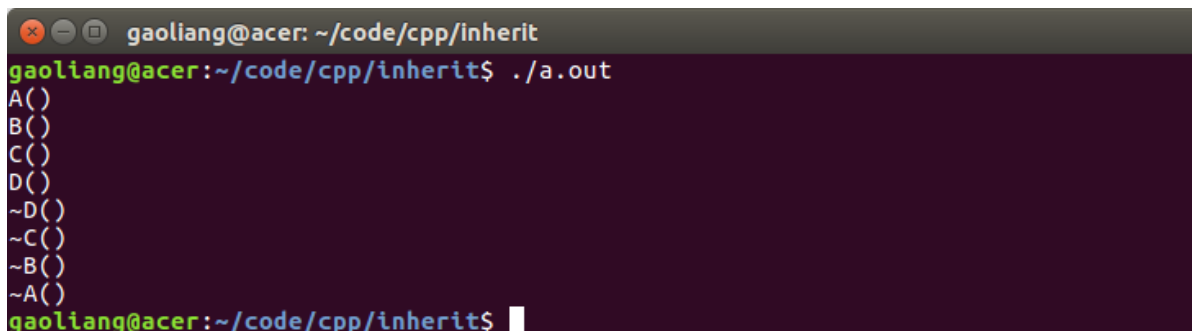
图中解决二义性时在vs下使用的是偏移量,而不是图中直接的指针指向,这里只是为了更直观的展示。

实际存放情况:

坝，个便演示，可以查看反汇编。

再看一下上面的类中虚继承的情况下构造与析构函数调用顺序：B类和C类虚继承A类

```
1 class A
2 {
3 public:
4     A(){cout<<"A()"<<endl;}
5
6     ~A(){cout<<"~A()"<<endl;}
7 };
8
9 class B: virtual public A
10 {
11 public:
12
13     B(){cout<<"B()"<<endl;}
14
15     ~B(){cout<<"~B()"<<endl;}
16 };
17
18 class C : virtual public A
19 {
20 public:
21     C(){cout<<"C()"<<endl;}
22
23     ~C(){cout<<"~C()"<<endl;}
24 };
25
26 class D : public B, public C
27 {
28 public:
29     D(){cout<<"D()"<<endl;}
30
31     ~D(){cout<<"~D()"<<endl;}
32 };
```



对照着对象模型看，只需要调用一次B类构造函数即可。

👍 1 赞

🔖 2 收藏

💬 评论



相关文章

- [提升 C++ 技能的 7 种方法 · 5](#)
- [C++17 中那些值得关注的特性](#)
- [C++17 相比于 C++14 的所有重大变化](#)
- [C++ 中命名空间的 5 个常见用法](#)
- [MFC双缓冲绘图实例](#)

可能感兴趣的话题

- [工作压力大，然后出错多。。然后工作压力大。该怎么调整自己的状态.....](#) · 5
- [是去北上广深闯几年还是留在二线城市？](#)
- [从Python转Go的开发多吗？](#)
- [直播：过年7天乐，学一门新技术](#) · 19

登录后评论

新用户注册

直接登录



- [本周热门文章](#)
- [本月热门文章](#)
- [热门标签](#)

0 [刚开始学编程？这几款小工具能让你事...](#)

1 [从业 24 年独立开发者：大多数同行...](#)

2 [Linux 新用户？来试试这 8 款重...](#)

3 [du 及 df 命令的使用（附带示例）](#)

4 [Linux 中的“大内存页”（hugepage...](#)

5 [深入学习 Redis（1）：Redis 内...](#)

6 [如何在 Linux 上安装应用程序](#)



业界热点资讯

[更多 »](#)



[李文星家属诉 BOSS直聘：哪怕赔一分 能给个交代也值](#)

21 小时前 · 4



[FSF 宣布 2017 年度的自由软件奖得主](#)

21 小时前 · 2



[GitLab 发布全球开发者报告：开源仍是主流](#)

2 天前 · 2



5 天前 · 17 · 2



[Mozilla 的 Firefox 2018 路线图](#)

4 天前 · 2



[精选工具资源](#)

[更多资源 »](#)



[mlpack: 一个C++机器学习库](#) [C++](#), [机器学习](#)



[Whitewidow: SQL 漏洞自动扫描工具](#) [数据库](#) · 2



[Caffe: 一个深度学习框架](#) [机器学习](#) · 3



[静态代码分析工具清单：公司篇](#) [静态代码分析](#)



[HotswapAgent: 支持无限次重定义运行时类与资源](#) [开发流程增强工具](#)

[关于伯乐在线博客](#)

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为“快餐”添加一些“营养”元素。

快速链接

[网站使用指南 »](#)

[问题反馈与求助 »](#)

[加入我们 »](#)

[网站积分规则 »](#)

[网站声望规则 »](#)

新浪微博: [@伯乐在线官方微博](#)

RSS: [订阅地址](#)

推荐微信号



合作联系

Email: bd@Jobbole.com

QQ: 2302462408 (加好友请注明来意)

[更多频道](#)

- [小组](#) — 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) — 分享和发现有价值的内容与观点
- [相亲](#) — 为IT单身男女服务的征婚传播平台
- [资源](#) — 优秀的工具资源导航
- [翻译](#) — 翻译传播优秀的外文文章
- [文章](#) — 国内外的精选文章
- [设计](#) — UI,网页, 交互和用户体验
- [iOS](#) — 专注iOS技术分享
- [安卓](#) — 专注Android技术分享
- [前端](#) — JavaScript, HTML5, CSS
- [Java](#) — 专注Java技术分享
- [Python](#) — 专注Python技术分享

