《深度探索 C++对象模型》读书笔记

整理日期：20120610

主页：http://chuanqi.name

E-mail：chuanqi.tan@gmail.com

这本书能让你对 C++有脱胎换骨的新认识，值得任何 C++程序员一看！

欢迎交流、指导；本文采用"CC BY 2.5"许可协议

By 谭川奇

前言

侯捷："这本书就是一位伟大的 C++编译器设计者在向你阐述他如何处理各种 explicit（明白出现于 C++程序代码）和 implicit（隐藏于程序代码背后）的 C++语意"。

为什么要学习 C++的对象模型，似乎这是 C++编译器实现者的工作：

- Lippman 这样说："程序员如果了解 C++对象模型，就可以写出比较没有错误倾向而且比较有效率的代码。"
- 侯捷这样说："这本书解决了过去令我百思不解的诸多疑惑"。
- 我同样认为：只有理解了 C++的对象模型，才能从根本上真正的理解 C++，才能在遇到错误时不惊慌，能够根据自己对于底层的了解去分析问题的根本原因并解决问题。

计算机是一门奇怪的科学，其它的科学技术都需要从上往下看才能有大局观，而计算机科学却必须从下往上看才能真正的理解。

但愿本笔记能为你的学习提供一些帮助！

谭川奇 2012.06.10 于北京

chuanqi.tan@gmail.com

本文采用的约定

蓝色：概念、讨论主题

下划线：值得注意的内容

紫红色：比较重要的地方

红色：必须理解的地方

绿色：一些评论、注解

斜体：一些总结性的话

粗体：对上述所有标记的加强、警示作用

目　录

第 1 章：关于对象..4
第 2 章：构造函数语意学..7
第 3 章：Data 语意学...9
第 4 章：Function 语意学...14
第 5 章：构造、解构、拷贝 语意学...18
第 6 章：执行期语意学..22
第 7 章：站在对象模型的尖端..26

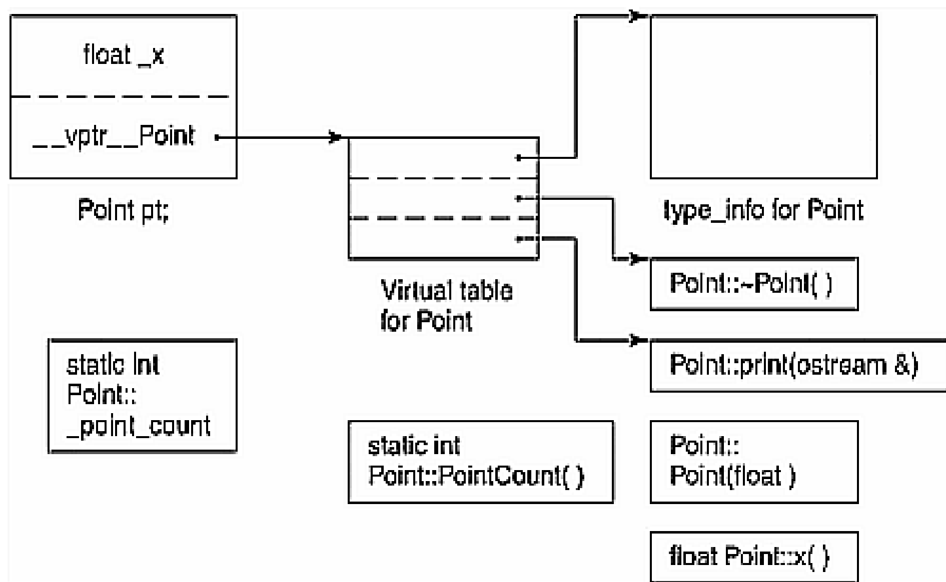
第 1 章：关于对象

介绍了 C++如何在背后实现一个对象，内存中的布局以及空间上的关系。

- 1.在 C++中以类的形式实现对象之后，会增加多少布局成本？

  ```
  template<class Type>
  class Point3d{
  public:
    Point3d(Type x, Type y, Type z) : x_(x), y_(y), z_(z) {}
    Type x() { return x_; }
  private:
    Type x_, y_, z_;
  }
  ```

  一般来说，不会增加任何成本。三个 data members 直接内含于一个 class object 中，就像 C 中的 struct 一样。而 member functions 虽然含在 class 的声明中，却不出现在 object 中（更像是类命名空间中的普通函数）。看，类的封装居然没有带来任何额外的成本。
- C++在内存布局以及存取时间上的主要额外负担都是由 virtual 引起的，包括：
  - virtual function 机制：引起了保存 vtable 和透过 vtable 找到函数地址；
  - virtual base class 机制：引起了透过指针来找到基类的成员。
- *C++的成本来源就是 vitrual 机制，无 virtual 就无额外的开销，这太棒了！*
- **C++对象模型（The C++ Object Model）**：



Nonstatic data members 被配置于每一个 class object 之内；

Static data members 则被存放在所有的 class object 之外；

Static 和 Nonstatic function members 也被放在所有的 class object 之外。

- Virtual function 机制则由以下的 2 个的两个步骤来支持：
    - 每一个 class 产生出一系列指针 Virtual function 的指针，**放在**一个被称为 virtual table(vtbl, vtable)的表格中；
    - 每一个 class object 被添加了一个指针 vptr，**指向**相对应的 vtable。vptr 的设置由编译器全权负责，程序员无需关心。
- RTTI：一般来说，每一个 class 相关联的 **type_info 对象的指针**通常也保存在 **vtable 的第一个 slot** 中。
- 需要清楚的明白一点是：

  一个 vtable 对应一个 class，一个 vptr 才对应一个 class object，必须区分开这 2 个概念。
- 引入继承后的对象模型成本：
    - 如果是普通的继承，父对象被直接包含在子对象里面，这样对父对象的存取也是直接进行的，没有额外的成本；
    - 如果是虚拟继承，则父对象会由一个指针被指出来，这样的话对父对象的存取就添加了一层间接性，必须经由一个指针来访问，添加了一次间接的额外成本。
- C++优先判断一个语句为声明：当语言无法区分一个语句是声明还是表达式时，就需用用一个超越语言范围的规则 —— C++优先判断为声明。
- struct 和 class 关键字的意义：
    - 它们之间在语言层面并无本质的区别，更多的是概念和编程思想上的区别。
    - struct 用来表现那些只有数据的集合体 POD（Plain OI' Data）、而 class 则希望表达的是 ADT（abstract data type）的思想；
    - 由于这 2 个关键字在本质是无区别，所以 class 并没有必须要引入，但是引入它的确非常令人满意，因为这个语言所引入的不止是这个关键字，还有它所支持的封装和继承的哲学；
    - 可以这样想象：struct 只剩下方便 C 程序员迁徙到 C++的用途了。
- C++只保证处于同一个 access section 的数据，一定会以声明的次序出现在内存布局当中。C++标准只提供了这一点点的保证。
- 与 C 兼容的内存布局： 组合，而非继承，才是把 C 和 C++结合在一起的**唯一可行**的方法。只有使用组合时，才能够保证与 C 拥有相同的内存布局，使用继承时的内存布局是不受 C++ Standard 所保证的（很多编译器也可行，但是标准未定义！）。

```
struct C_point {...};
class Point{
public:
  operator c_point() { return c_point_; }
private:
  C_point c_point_;
```

}

- C++支持三种形式的编译风格（或称典范 paradigm）：
  - 面向过程的风格：就像 C 一样，一条语句接一条语句的执行或者函数跳转；
  - **基于对象**的风格（object-based）（或称 ADT）：仅仅使用了 class 的封装，很多人都是在用基于对象的风格却误以为自己在使用面向对象的风格；
  - **面向对象**的风格（object-oriented）：使用了 class 的封装和多态的编程思维（*多态才是真正的面向对象的特征*）。
  - 纯粹以一种 paradigm 写程序，有助于整体行为的良好稳固。
- 还有一种能引起多态的写法：

  (*point).vf(); //当 point 指向派生类时也能引发多态，已实验验证。等价于 point->vf();
- 一个 reference 通常是以一个指针来实现的，所以 point 和 reference 并没有本质的区别。

  注：Lippman 这里只是说通常的编译器实现，但是 C++标准并未给予保证。
- 一个对象的内存布局大小（通常由 3 部分组成）：
  - 其 nonstatic data member 的总和大小；
  - 任何由于位对齐所需要的填补上去的空间；
  - 加上了为了支持 virtual 机制而引起的额外负担。
  - 这也印证了前面的一个结论：***C++中的额外成本通常都是由于 virtual 机制所引起的。***
- 指针的类型：
  - 对于内存来说，不同类型的指针并没有什么不同。它们都内是占用一个 word 的大小，包含一个数字，这个数字代表内存中的一个地址；
  - 感觉上，指针的类型是编译器的概念，对于硬件来说，并没有什么指针类型的概念；
  - 转型操作也只是一种编译器的指令，它改变的内是编译器对被指内存的解释方式而已！
- 多态只能由"指针"或"引用"来实现，根本原因在于：
  - 指针和引用（通常以指针来实现）的大小是固定的（一个 word），而对象的大小却是可变的。其类的指针和引用可以指向（或引用）子类，但是基类的对象永远也只能是基类，没有变化则不可能引发多态。
  - 一个 point 或 reference 绝不会引发任何"与类型有关的内存委托操作"，在指针类型转换时会受到的改变的只有它们所指向内存的解释方式而已。（例如指针绝不会引发 slice，因为它们大小相同）
- 在初始化、assignment 等操作时，编译器会保证对象的 vptrs 得到正确的设置。这是编译器的职责，它必须做到。一般都是通过在各种操作中插入编译器的代码来实现的。

第 2 章：构造函数语意学

详细的讨论了 constructor 如何工作，讨论构造一个对象的过程以及构造一个对象给程序带来的影响。

1. C++中对于默认构造函数的解释是：默认的构造函数会在**需要的时候**被编译器产生出来。

   这里非常重要的一点是：谁需要？是**程序的需要**还是**编译器的需要**？如果是程序的需要，那是程序员的责任；只有在是编译器的需要时，默认构造函数才会被编译器产生出来，而且被产生出来的默认构造函数只会执行编译器所需要的行动，而且这个产生操作只有在默认构造函数真正被调用时才会进行合成。

   例如：成员变量的初始化为 0 操作，这个操作就是程序的需要，而不是编译器的需要。

2. 区分 trivial 和 notrivial：
   1. 只有编译器需要的时候（为什么会需要？后面讲的很清楚），合成操作才是 notrivial 的，这样的函数才会被真正的合成出来；
   2. 如果编译器不需要，而程序员又没有提供，这时的默认构造函数就是 trivial 的。虽然它在概念上存在，但是编译器实际上根本不会去合成出来，因为他不做任何有意义的事情，所以当然可以忽略它不去合成。*trivial 的函数只存在于概念上，实际上不存在这个函数。*

3. 总结变量的初始化：只有全局变量和静态变量才会保证初始化，其中静态变量可以视为全局变量的一种，因它静态变量也是保存在全局变量的存储空间上的。

   Golbal objects 的内存保证会在程序激活的时候被清 0；Local objects 配置于程序的堆栈中，Heap objects 配置于自由空间中，都不一定会被清为 0,它们的内容将是内存上次被使用后的痕迹！

4. 类声明头文件可以被许多源文件所包含，如何避免合成默认构造函数、拷贝构造函数、析构函数、赋值拷贝操作符（4 大成员函数）时不引起函数的重定义？

   解决方法是以 inline 的方式完成，如果函数太复杂不适合 inline，就会合成一个 explicit non-inline static 实体（Static 函数独立于编译单元）。

5. 如果 class A 内含一个或以上的 member objects，那么 A 的 constructor 必须调用每一个 member class 的默认构造函数。

   具体方法是：编译器会扩张 constructors，在其中安插代码使得在 user code 被调用之前先调用 member objects 的默认构造函数（当然如果需要调用基类的默认构造函数，则放在基类的默认构造函数调用之后：基类构造函数->成员构造函数->user code）。

   C++要求以"member objects 在 class 中的声明次序"来调用各个 construtors。这就是声明的次序决定了初始化次序（构造函数初始化列表一直要求以声明顺序来初始化）的根本原因！

6. 带有 virtual functions 的类的默认构造函数毫无疑问是 notrivial 的，需要编译器安插额外的成员 vptr 并在构造函数中正确的设置好 vptr，这是编译器的重要职责之一。

   带有 virtual base class 的类的默认构造函数同样也毫无疑问的 notrivial，编译器需要正确设置相关的信息以使得这些虚基类的信息能够在执行时准备妥当，这些设置取决于编译实现虚基类的手法。

7. 编译器有 4 种情况会使得编译器真正的为 class 生成 nontrivial 的默认构造函数，这个 nontrivial 的默认构造函数只满足编译器的需要（调用 member objects 或 base class 的默认构造函数、初始化 virtual function 或 virutal base class 机制）。其它情况时，类在概念上拥有默认构造函数，但是实际上根本不会被产生出来（前面的区分 trivial 和 notrivial）。
8. C++新手常见的 2 个误区：
   1. ERROR: 如果 class 没有定义 default constructor 就会被合成一个；
      首先定义了其它的 constructor 就不会合成默认构造函数，再次即使没有定义任何构造函数也不一定会合成 default constructor，可能仅仅是概念上有，但实际上不合成出来。
   2. ERROR: 编译器合成出来的默认构造函数会明确设定每一个 data member 的默认值；
      明显不会，区分了 Global objects, Stack objects, Heap objects 就非常明白了只有在 Global 上的 objects 会被清 0，其它的情况都不会保证被清 0。
9. Copy constructors 和默认构造函数一样，只有在必须的时候才会被产生出来，对于大部分的 class 来说，拷贝构造函数仅仅需要按位拷贝就可以。满足 bitwise copy semantics 的拷贝构造函数是 trivial 的，就不会真正被合成出来（与默认构造函数一样，只有 nontrivial 的拷贝构造函数才会被真正合成出来）。
   对大多数类按位拷贝就够了，什么时候一个 class 不展现出 bitwise copy semantics 呢？分为 4 中情况，前 2 种很明显，后 2 种是由于编译器必须保证正确设置虚机制而引起的。
   1. 当 class 内含一个 member object 而后者声明了（也可能由于 nontrivial 语意从而编译器真正合成出来的）一个 copy constructor 时；
   2. 当 class 继承自一个存在有 copy constructor 的 base class（同样也可能是合成）时；
   3. 当 class 声明了一个或多个 virtual functions 时；（vf 影响了位语意，进而影响效率）
   4. 当 class 派生自一个继承串链，其中一个或多个 virtual base classes 时。
10. NVR 优化：编译器会把返回值作为一个参数传到函数内，比如：
    X foo() {...}
    会被自动更改（也可以自己手动做这个优化）为：void foo(X &__result) { ... }
    从使用者的角度，用 return X(...)代替 X x; return x;能够辅助这个优化操作。
11. 不要随意提供 copy constructor，对于满足 bitwise copy semantics 的类来说，编译器自动生成的拷贝构造函数自动地使用了位拷贝（这是效率最高的），如果你自己随意提供 copy constructor 就会压抑掉编译器的这个行为，画蛇添足还影响了效率。
12. 成员初始化列表：在成员初始化列表背后实际发现的事情是什么呢？
    编译器会一一操作初始化列表，把其中的**初始化操作以 member 声明的次序**在 constructor 内安插初始化操作，并且在任何 explicit user code 之前。
    "以 member 声明的次序来决定初始化次序"和"初始化列表中的排列次序"之间的外观错乱，可能会导致一些不明显的 Bug。
    不过 GCC 已经可以强制要求使用声明次序来进行初始化以避免了这个陷阱。

13. 理解了初始化列表中的实际执行顺序中"以 member 声明的次序"来决定的，就可以理解一些很微妙的错误了。比如：

    A() : i(99), j(66), value(foo()) {… }

    int i, value, j;

    这会不会产生错误取决于成员函数 foo()是依赖于 i 还是 j：

    如果 foo 依赖于 i，由于 i 声明在 value 之前，所以不会产生错误；

    如果 foo 依赖于 j，由于 j 声明在 value 之后，就产生了使用未初始化成员的错误。

第 3 章：Data 语意学

C++对象模型的细节，讨论了 data members 的处理。

1. 空类也有 1Byte 的大小，因为这样才能使得这个 class 的 2 个 objects 在内存中有独一无二的地址。
2. 一个对象的内存布局大小（通常由 3 部分组成）Copy 自第 1 章：
    1. 其 nonstatic data member 的总和大小；
    2. 任何由于位对齐所需要的填补上去的空间；
    3. 加上了为了支持 virtual 机制而引起的额外负担。
3. 对 member functions 本身的分析会直到整个 class 的声明都出现了才开始。所以 class 的 member functions 可以引用声明在后面的成员，C 语言就做不到。
4. 和第 3 条对比，需要十分注意的一点是：class 中的 typedef 并不具备这个性质。
    因此，类中的 typedef 的影响会受到函数与 typedef 的先后顺序的影响。
    ```
    typedef int length;
    class Point3d{
    public:
      void f1(length l){ cout << l << endl; }
      typedef string length;
      void f2(length l){ cout << l << endl; }
    };
    ```
    这样 f1 绑定的 length 类型是 int；而 f2 绑定的 length 类型才是 string。

    所以，对于 typedef 需要防御性的程序风格：始终把 nested type 声明（即 typedef）放在 class 起始处！
5. 传统上，vptr 被安放在所有被明确声明的 member 的最后，不过也有些编译器把 vptr 放在最前面（MSVC++就是把 vptr 放在最前面，而 GCC 是把 vptr 放在最后面）。
6. 在 C++中，直观上来说，由一个对象存取一个 member 会比由一个指针存取一个 member 更

快捷。但是对于经由一个对象来存取和由一个指针来存取一个静态的member来说，是完全一样的，都会被编译器所扩展。

7. 经由一个函数调用的结果来存取静态成员，C++标准要求编译器必须对这个函数进行求值，虽然这个求值的结果并无用处。

   foo().static_member = 100;

   foo()返回一个类型为X的对象，含有一个static_member，foo()其实可以不用求值而直接访问这个静态成员，但是C++标准保证了foo()会被求值，可能的代码扩展为：

   (void) foo();

   X::static_member = 100;

8. 对一个nonstatic data member进行存取操作，编译器会进行如下扩展：

   origin.y_ = 10; 扩展为=> &origin + (&Point3d::y_ - 1)

   注意其中的-1操作：指向data member的指针，其offset值总是被加上1。这样可以使编译系统区分出"一个指针data member的指针，用以指向class的第一个member"和"一个指向data member的指针，但是没有指向任何member"两种情况（*成员指针也需要有个表示NULL的方式，0相当于用来表示NULL了，其它的就都要加上1了*）。

9. X x; x.x = 0.0;

   X *px; px->x = 0.0;

   这两种写法在什么时候有重大的区别呢？

   答案是当类型X是一个继承体系中有虚基类的子类，并且x成员又正好是虚基类中的成员时，这两种写法在编译器看来就会有重大的区别了（*这种情况下的offset计算方式不同，其它情况的offset也可以直接算出*）。因为x的类型是固定的，编译器可以直接向第8条中所述的进行offset扩展，但是对于pt来说，由于无法确定pt指向的真正类型，所以只能借由一个执行期的间接来得到成员的具体地址。

10. 派生类的成员和基类成员的排列次序并未在C++ Standard中强制指定；理论上编译器可以自由安排，但是对于大部分的编译器实现来说，都是把基类成员放在前面，但是virtual base class除外。（一般而言，任何一条规则一旦碰到virtual base class就没辙了）。

11. 一般而言，具体继承（相对于虚拟继承）并不会增加时间和空间上的负担，前面说过多次：C++中的额外成本大多来自于virtual机制。

12. C++ Standard保证："**出现在派生类中的base class subobject有其完整原样性！**"

    子类会被放在父类的对齐空白字节之后，因为父类的完整性必须得以保证，父类的对齐空白字节也是父类的一部分，也是不可分割的。

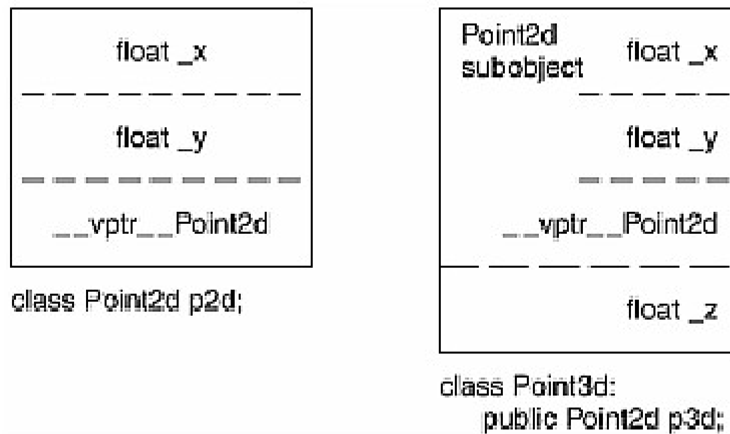
13. 支持多态所带来的4个负担：

    1. 导入virtual table用来存放每一个virtual functions的地址。这个Table的元素数目一般而言是被声明的virtual functions数目再加上一个或两个slots（用以支持RTTI）；
    2. 在每一个class object中安插一个vptr指向相应的vtable；

3. 在 constructor 中安插代码以正确设置 vptr；
4. 在 destructor 中安插代码以正确设置 vptr；

14. 单一继承并含有虚拟函数时的内存布局（考虑一般把 vptr 放在尾部的设计）：

Figure 3.3. Data Layout: Single Inheritance with Virtual Inheritance

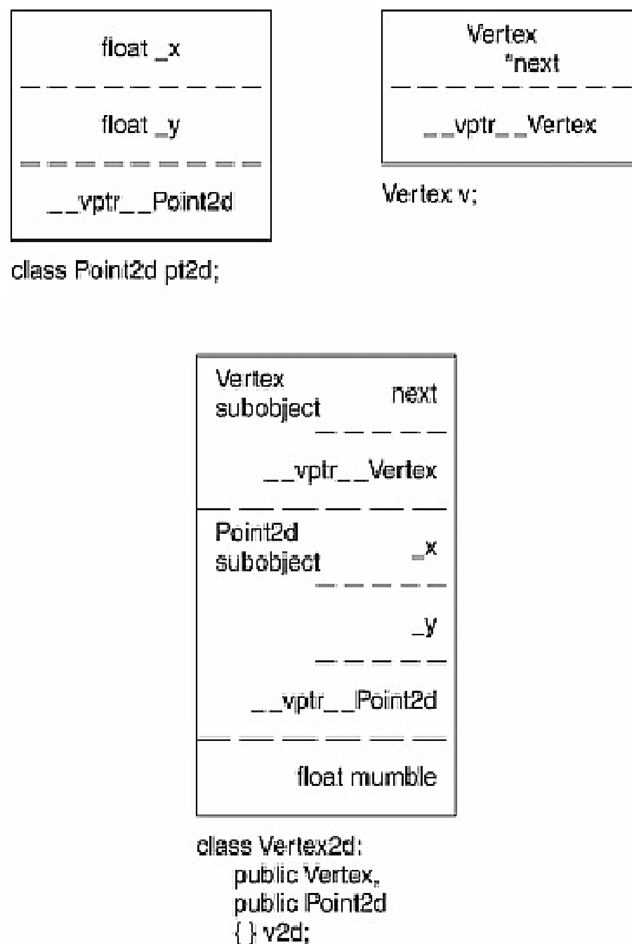


单一继承时 vptr 被放在第一个子类的末尾，产生这样布局的根本原因在于"**基类的完整性必须在子类中得以保存**"。

这个名叫 vptr_Point2d 的 vptr 可以这样理解，由 Point2d 而引发的 vptr，在 Point2d 的对象中，这个 vptr_Point2d 所指向的是与 Point2d 所对应的 point2d_vtable，而在 Point3d 的对象中，这个 vptr_Point2d 所指向的却是与 Point3d 所对应的 point3d_vtable。

15. 多重继承时的内存布局（多重继承时的主要问题在于派生类与非第 1 基类之间的转换）：

Figure 3.4. Data Layout: Multiple Inheritance



在多重继承的派生体系中，将派生类的地址转换为第1基类是成本与单继承是相同的，只需要改换地址的解释方式而已；而对于转换为非第1基类的情况，则需要对地址进行一定的offset操作才行。

C++ Standard并未明确base classes的特定排列次序，但是目前的编译器都是按照声明的次序来安放他们的。（*有一个优化：如果第1基类没有vtable而后继基类有，则可能把它们调个位置*）。

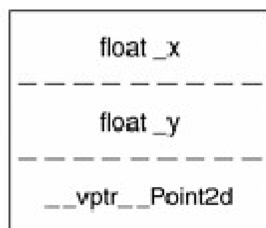
多重继承中，可能会有多个vptr指针，视其继承体系而定：派生类中vptr的数目最等于所有基类的vptr数目的总和。

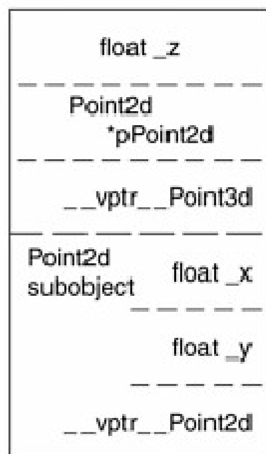
16. 虚拟继承：虚拟继承把一个类切割为2部分：一个不变局部和一个共享局部。

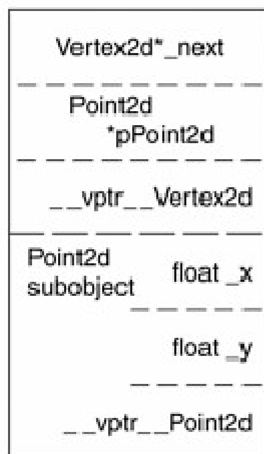
这个共享局部必须通过编译器安插的一些指针指向virtual base classobject来间接的存取，这样才能够实现共享。对于这个安插指针来实现共享的技术，有两种主流的做法。

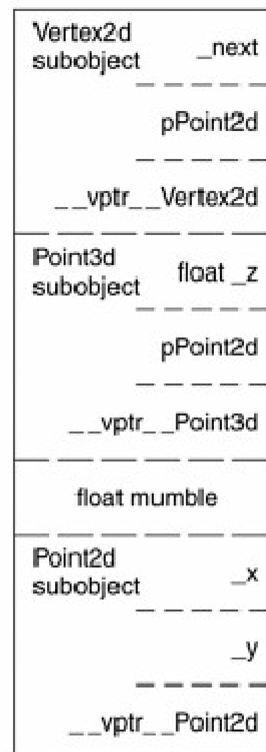
一种做法是直接使用一个指针指向虚基类：

Figure 3.5(a). Data Layout: Virtual Inheritance with Pointer Strategy



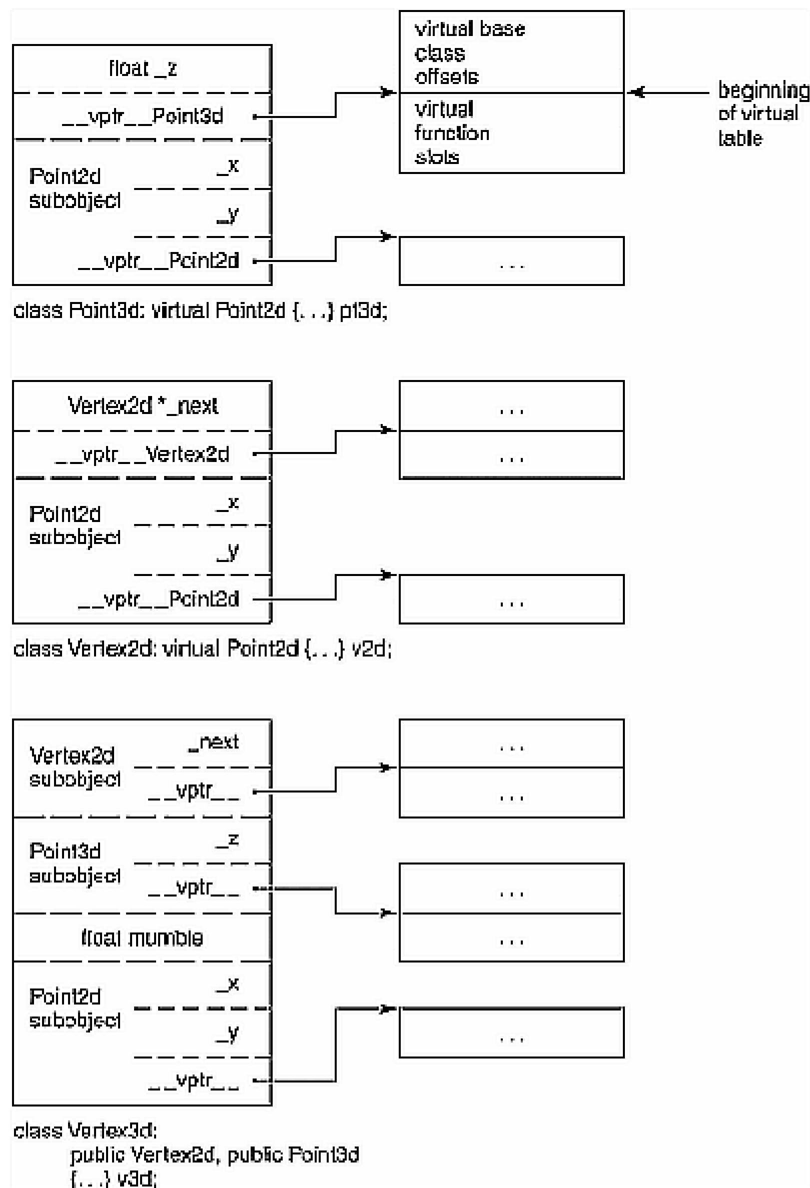






另一种做法是在 vtable 中放置 virtual base class 的 offset：

Figure 3.5(b). Data Layout: Virtual Inheritance with Virtual Table Offset Strategy



这种使用偏移地址的方式好处在于：vptr 是已经存在的成本，而 vtable 是 class 的所有 objects 所共享的成本。对于每一个 class object 没有引入任何的额外成本，仅仅在 vtable 多存储了一个 slot 布局，而前一种方式却对每一个 object 都引了两个指针的巨大成本。

这两种方式都是把虚基类放在内存中模型中的最后面，然后借由一层间接性（指针或 offset）来访问。

现在理解了 virtual base classes 之后，才觉得原来虚拟继承也一点都不难以理解，就是把普通的继承中的直接访问添加了一层间接访问布局，还是很好理解的。

17. 一般而言：virtual base classes 最有效的一种运用形式就是：一个抽象的 virtual base class，没有任何 data members。
18. 普通封装不会带来任何执行期的成本，编译器可以轻松优化掉普通封装带来的任何成本。

    但是一旦涉及到虚拟继承，效率就会大幅降低，在有 n 层的虚拟继承体系中，普通的访问就要经过 n 次间接，普通访问的成本就变为了 n 倍。

再次表示，C++中的额外成本基本都是由于 virtual 机制引起的。

19. 指向 Data Members 的指针内部实际保存的是这个 data member 相对于对象起始地址的偏移地址（offset）（但需要另外加 1 以区分空指针，前面有讲过了）！
20. 使用指向 Data Members 的指针时也不会损失效率，成本与直接存取相同。就像第 18 条所说的，普通访问没有额外成本，但是遇到虚拟继承效率就大幅降低。

第 4 章：Function 语意学

C++对象模型的细节，讨论了 member functions，尤其是 virtual function。

1. C++的设计准则之一就是：nonstatic member function 至少必须和一般的 nonmember function 有相同的效率。

   实际上，nonstatic member function 会被编译器进行如下的转换，变成一个普通函数：

   Type1 X::foo(Type2 arg1) { … }

   会被转换为如下的普通函数：

   void foo(X *const this, Type1 &__result, Type2 arg1) { … }
2. *实际上，**普通函数、普通成员函数、静态成员函数到最后都会变成与 C 语言函数类似的普通函数**，只是编译器在这些不同类型的函数身上做了不同的扩展，并放在不同的 scope 里面而已。*
3. 各家的 C++编译器对 man mangling 的做法还不统一，造成了各家编译器编译出的 C++代码不能兼容。不过，还有太多的原因使得各家的代码不能兼容了。
4. 虚拟成员函数：ptr->normalize();

   会被转换为如下形式的调用：(* ptr->vptr[1])(ptr);

   事实上，vptr 的名称会被加上 mangled，因为对于一个复杂的派生体系，可能会有多个 vptr。前面总结过了，一个派生类中的 vptr 数目等于其基类的总和。
5. 静态成员函数其实就是带有类 scope 的普通函数，它也没有 this 指针，所以它的地址类型并不是一个指向成员的指针而仅仅是一个普通的函数指针而已。

   静态成员函数是作为一个callback 的理想对象，在类的scope 内，又是普通的函数指针。
6. 识别一个 class 是否支持多态，唯一的适当方法就是看它是否有任何的 virtual function。只有 class 声明有任意一个 virtual function，那么它就需要额外的执行期信息 vtable。
7. 单一继承时的内存布局：

   前面讨论过，对于单一继承的情况，每个类最多只会有一个 vptr 指针，并放在第一个拥有 virtual function 的类的后面（考虑面前提过的，父类必须保证子类对象的完整性，就一下子明白了这种内存布局的原因）。

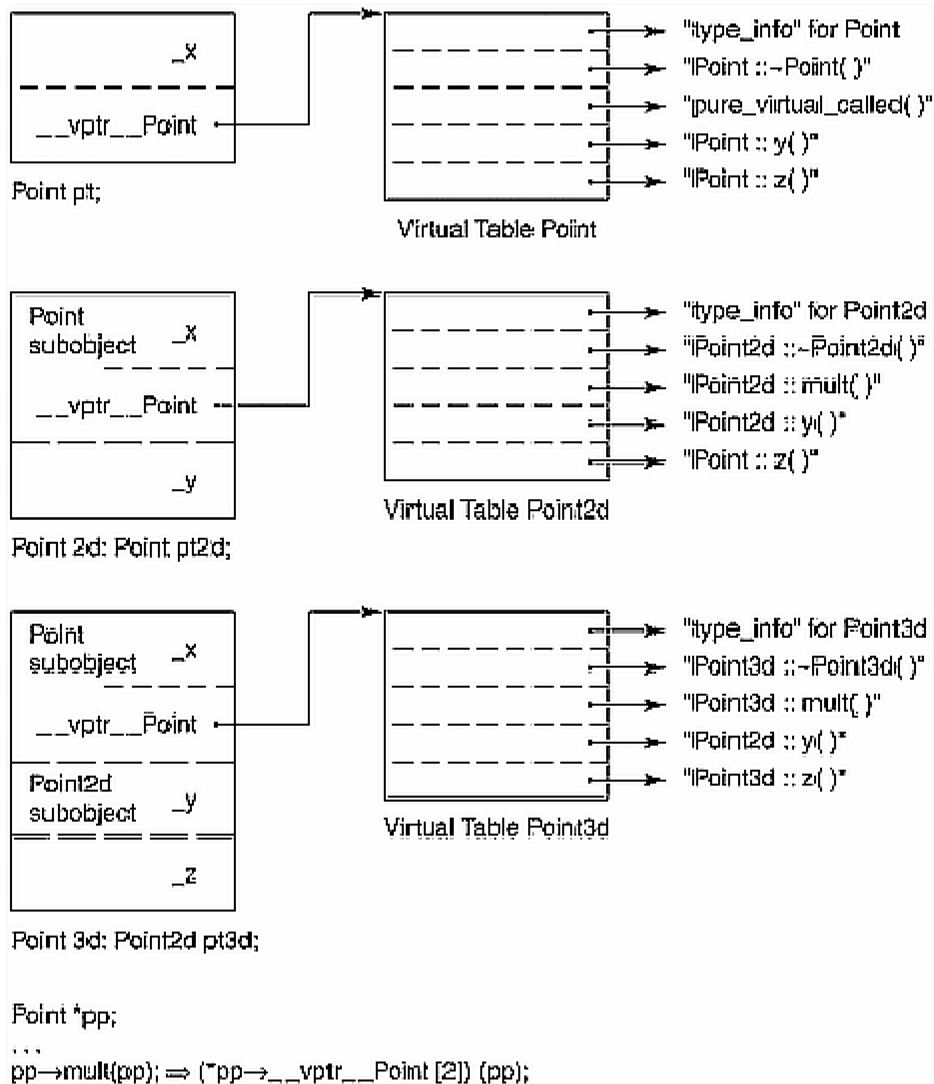
   这种内存布局下，virtual function 是如何工作的呢？因为在调用一个成员函数 ptr->z()时：

（1）：虽然不能确定 ptr 直接指向的类型，但是可以经由 ptr 找到它的 vtable，而 vtable 记录了所指对象的真正类型（一般对象的 type_info 放在 vtable 的第 1 个 slot 里）；

（2）：虽然不能确定应该调用的 z 函数的真正地址，但是可以知道它在 vtable 中被放在哪一个 slot，于是就直接去 vtable 中相应的 slot 中取出真正的函数地址加以调用。

单一继承时的内存布局图如下所示：

Figure 4.1. Virtual Table Layout: Single Inheritance

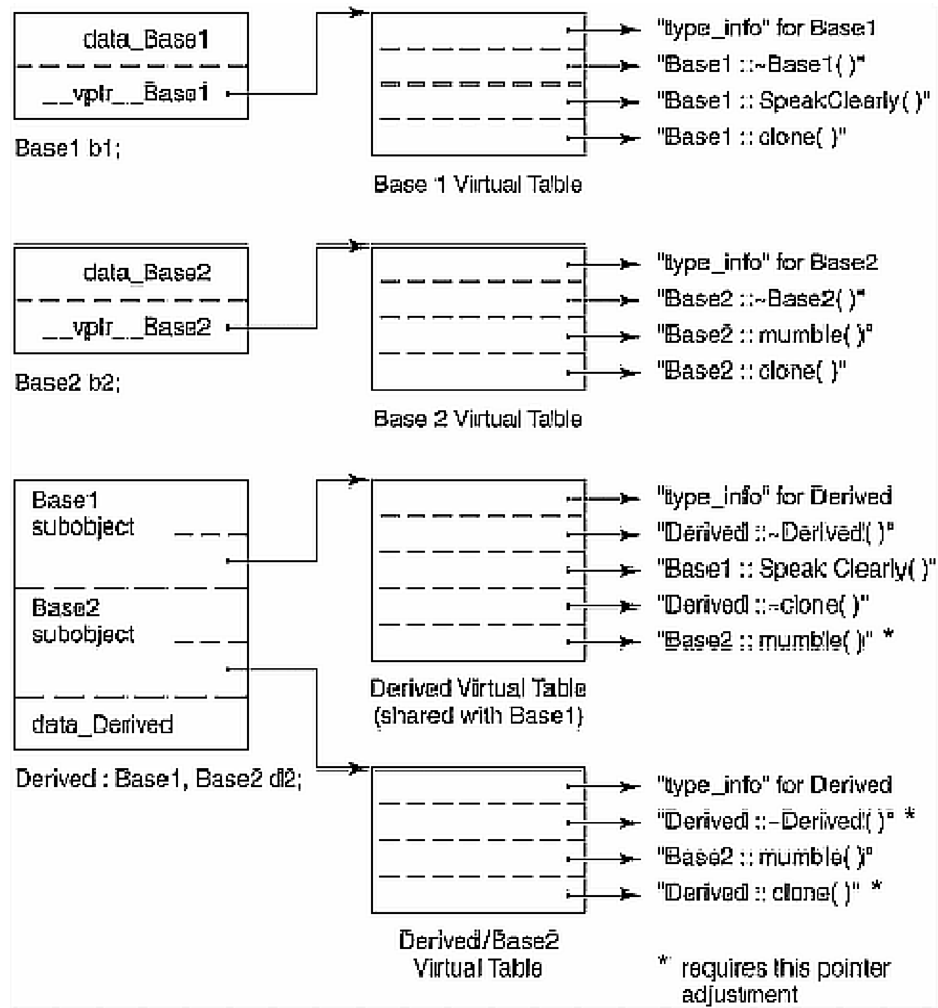


8. 多重继承下的内存布局：

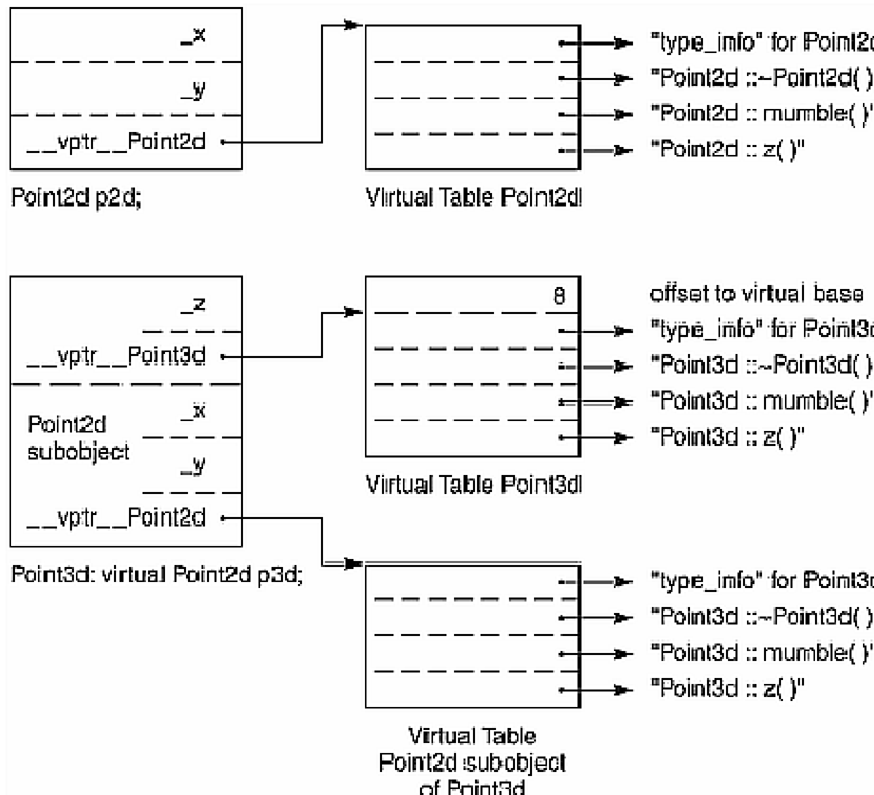
在多重继承中比单一继承更复杂的地方在于对非第 1 基类的指针和引用进行操作时，必须进行一些执行期的调整 this 指针的操作。比如对于简单的 delete 操作： deleta base2;由于 base2 可能没有指向对象的起始地址，这样简单的删除操作会引发灾难，所以需要对 base2 做执行期的调整才能正确的 delete 对象。

在多重继承下，一个 Derived Class 可能同时含有多个 vptrs 指针，这取决于它的所有基类的情况（想想基类完整性）。也可能有对应的多个 vtables（如 cfront），但也可能无论如何只有一个 vtable（把所有的 vtables 合成一个，并使得所有的 vptrs 都指向这一个合成的 vtable + offset，如 Sun 的编译器），这都取决于编译器的策略。

多重继承下的内存布局如下所示：



9. 虚拟继承下的内存布局：



Lippman 建议：不要在一个 virtual base class 中声明 nonstatic data members。如果一定要这么做，那么你会距离复杂的深渊愈来愈近，终不可拔。

这样看来 virtual 继承技术就更加像是其它高级语言里的 Interface 了。但是：如果我的基类中没有 nonstatic data members，那还我要 virtual 继承做什么呢？普通的多重继承不就可以了啊？反正没有 data members，就根本不需要去共享什么啊？

10. 这里的函数性能测试表明，inline 函数的性能如此之高，比其它类型的函数高的不是一个等级。

    因为 inline 函数不只能够节省一般函数调用所带来的额外负担，也给编译器提供了程序优化的额外机会。

11. 取一个 nonstatic member function 的地址，如果该函数是 nonvirtual，则得到的结果是它在内存中的真正地址。然而这个值是不完全的，它需要被绑定于某个 class object 的地址上，才能够通过它调用该函数。

    很明显，这个 nonstatic member function 被编译器添加了一个参数 this,如果不绑定于 class object 就无法传递这个 this 指针。

    (origin.*fptr)(); 会被转换为 (fptr)(&origin);

12. 在 GCC 和 VC++2010SP1 中，对所有成员函数输出它们的取地址操作的结果都是"1"。

    这是为什么呢？

```
class A{
public:
	int a;
	int b;
	int c;
	int d;
};
int main(){
	cout << (&A::a) << " ";
	cout << (&A::b) << " ";
	cout << (&A::c) << " ";
	cout << (&A::d) << " ";
	return 0;
}
```

    输出的结果为：1 1 1 1

    甚至于，把所有的 fooX 换成 virtual 的成员函数，输出的结果依然全是"1"，为什么呢？

13. 指向 virtual member functions 的指针：

    对一个 virtual member function 取其地址，所能获得的只是一个 vtable 中的索引值。

14. inline 函数扩展时的实际参数取代形式参数的过程，会聪明地引入临时变量来避免重复求值。

    对于函数： inline int min(int i, int j) { return i < j : i : j; }

    如果这样调用：minval = min(foo(), bar() + 1);

编译器就会自动的引入临时变量并赋值 t1=foo(), t2=bar()，避免了对这个函数的多次调用。*因此，知道编译器会自动的做这些优化，就没有必须自己去画蛇添足的手动引入临时变量了。*

15. inline 中再调用 inline 函数，可能使得表面上一个看起来很平凡的 inline 却因连锁的复杂性而没有办法扩展开来。

    对于既要安全又要效率的程序，inline 函数提供了一个强而有力的工具，然后与 non-inline 函数比起来，它们需要更加小心的处理。

第 5 章：构造、解构、拷贝 语意学

C++对象模型的细节，讨论了 class 的整个模型，一个对象的完整生命周期。

- 纯虚函数也可以被调用，方式如下：

```
class A{
public:
 virtual ~A(){}
 virtual void f() = 0;
};
void A::f(){cout << "pure virtual" << endl;} //纯虚函数必须定义在类声明的外部
class D : public A{
public:
 virtual void f(){ A::f(); }  //纯虚函数必须经由派生类显式的要求调用
};
int main() {
 D d;
 d.f();
 return 0;
}
```

  输出的结果为“pure virtal”。

  这里需要注意的几点：

  - 纯虚函数不能在类的声明中提供实现，只能在类声明的外部来提供默认的实现；
  - 基类的纯虚函数的默认实现必须由派生类显式的要求调用；
  - 派生类不会自动继承这个纯虚函数的定义，如果派生类 D 未定义 f()，那么 D 依然是一个抽象类型；
  - 这种 pure virtual 函数还提供实现的方案比较好的应用场景为：*基类提供了一个默认的实现，但是不希望自动的继承给派生类使用，除非派生类明确的要求。*
  - 还需要注意这个纯虚函数为析构函数的情况。C++语言保证继承体系中的每一个 class object 的 destructors 都会被调用。所以编译器一定会扩展派生类的析构函数去显式地调

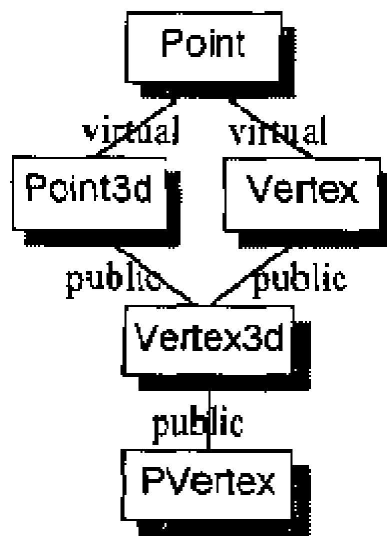
用基类的析构函数。

    - 另外一个重要的应用场景：有些情况下会把析构函数声明为纯虚。这时，必须为纯虚析构函数提供一个默认的实现。否则，派生类的析构函数由于编译器的扩展而显式的调用基类的析构函数时会找不到定义。同时编译器也无法为已经声明为纯虚的析构函数生成一个默认的实现。

- 虚函数中的 const 哲学：一个虚函数该不该被定义为 const 呢？一个虚函数在基类中不需要修改 data member 并不意味着派生类改写它时一定不会修改 data member。
  所以除非有十足的把握，一般就不声明为 const。
- Lippman 认为把所有的函数都声明为 virtual function，然后再靠编译器的优化操作把非必须的 virtual invocation 去除，并不是好的设计观念。不过，Java 和.NET 很可能都是这么干的。
- 对象能从三个地方产生出来：Global 内存、Local 内存、Heap 内存。
- 观念上，编译器会为每一个类产生 4 个函数：
  dafault constructor, destructor, copy constructor, copy assignment operator。
  但是，切记这仅仅是观念上的，trivial 的函数不会被真正的产生出来。
- C 和 C++的又一个不同点，就是"C 语言的临时性定义"。
  像 Point global; 这样的定义：
  在 C 中会被视为一个"临时性定义"，可以在程序中出现多次，这些实例最后会被链接器折叠成起来，最终只留下一个实体；
  但是在 C++会被视为一个"完全定义"，所以只能出现一次，在实现 C 一样的临时性语意，C++中必须把它声明为 extern，即：extern Point global;
- 理论上：A *pa1 = new A; 和 A *pa2 = new A(); 之间是有差别的，前一个应该不会调用默认构造函数而后一个会。但是在 GCC 和 VS2010 的实验中发现，这 2 个写法是完全没有区别的，默认的构造函数都被调用了。
- 对于可以视为 POD 的 class（没有声明构造函数、没有 virtual 机制等等），就可以使用 POD 结构特有的 initialization list 进行初始化。
  Point p = {2, 3};
  C++11 中的 initialization list 被大量使用。
- 引入 virtual function 会给对象的构造、拷贝和析构等过程带来的负担如下：
    - constructor 必须被安插一些代码以便将 vptr 正确的初始化，这些代码需要被安插在任何 base class constructors 的调用之后，但必须在任何 user code 的代码之前；
    - 合成 copy constructor 和 copy assignment operator，因为它们不再是 trivial 的了，它们必须安插代码以正确的设置 vptr；
- C++ Standard 要求尽量延迟 nontrival members 的实际合成操作，直到真正遇到其使用场合为止。
- constructor 会被编译器安插大量的代码，一般而言编译器所做的扩充操作大约如下：

- 初始化成员：使用 member initialization list 或者调用默认构造函数；
- 在那之前，如果 class object 有 vptr，它们必须被正确的设置；
- 在那之前，所有的上一层的 base class construcotrs 必须被调用，以 base classes 声明的顺序。使用 member initialization list 或者调用默认构造函数，同时如果 base class 是多重继承下的非第 1 基类，还需要调整 this 指针；
- 在那之前，所有的 virtual base class constructors 必须被调用，从左到右，从深到浅。并同时设置好 virtual base class 所需要使用的各种机制；
- 即处理顺序为：virtual base classes → base class → vptr → member。

- 赋值运算符中切记要记得进行自我检查。
- 虚拟继承时，共享基类必须由最底层的 class 负责初始化操作：
这是虚拟继承时非常重要的一点，共享基类的初始化操作必须由最底层的类来负责，中层层次的



类调用这个共享基类初始化的操作会被编译器所压抑掉。
考虑对于如下继承体系的类：

编译器如何压抑非底层类对共享基类的初始化操作呢？是通过对 Point3d 和 Vertex 的构造安插一个额外的参数__most_derived 来解决的。

```
Point3d* Point3d::Point3d( Point3d *this, bool __most_derived, float x,
float y, float z ) {
 if ( __most_derived != false )
   this->Point::Point( x, y);
 this->__vptr__Point3d = __vtbl__Point3d;
 this->__vptr__Point3d__Point = __vtbl__Point3d__Point;
 this->_z = rhs._z;
 return this;
}
Vertex3d* Vertex3d::Vertex3d( Vertex3d *this, bool __most_derived, float
x, float y, float z ) {
```

```
 if ( __most_derived != false )
   this->Point::Point( x, y);
 this->Point3d::Point3d( false, x, y, z );
 this->Vertex::Vertex( false, x, y );
 return this;
}
```

当 Point3d 是作为最底层类来构造时，__most_derived 参数会被设置为 true，于是 Point 的构造函数就会被调用；当 Point3d 的构造函数是被 Vertex3d 间接调用时，__most_derived 参数会被设置为 false，于是调用 Point 构造函数的操作就被压抑掉了。

这种由最底层类来负责初始化共享基类的手法貌似有一点不优雅，但是这却是共享基类唯一可能正确的确定初始化的地方。

*Point3d 和 Vertex 对于 Point 的初始化要求不同，该听的呢？只能由 Vertex3d 或 PVertex 来作出**唯一的决定**了。*

- 在构造函数中调用 virtual function 是没有多态性的，因为在构造函数中，对象还不完整，派生类的部分还没有开始构造，当然不能调用它们的成员函数，否则在它们的成员函数中可能会访问还不存在的成员变量。

  由于在构造函数中没有多态性，所以催生了一种在构造函数中清 0，再提供一个 init()进行真正的初始化的保护性手法。

- 要保证在构造函数中没有多态性，虚拟机制就必须知道一个调用操作是否来源自于构造函数之中，这是如何实现的呢？

  答案是编译器在构造函数中安插代码时会保证：先调用所有基类的构造函数，再设置 vptr，然后再调用 member initialization 操作。这是构造函数中没有多态性的根本原因！

  在任何 User code 和 member initialization 被调用之前，vptr 被正确的设置为了当前类的类型，于是在调用 virtual function 时从 vtable 中取出来的函数地址就是正确的当前类的成员方法地址。

- 一个很容易犯的错误：

```
struct A : public Base{
   A() : Base(foo()) , valueA(10) {}
   int foo() {return valueA;}
   int valueA;
}
```

  使用了派生类的成员方法去初始化基类，注意在这个时候派生类还没有开始构造，调用它的成员方法的行为当然是未定义的！

- 一个 class 的默认 copy assignment operator，以下情况不会表现出 bitwise copy 语意：
  - 当含有一个或以上的成员有 copy assignment operator 时；
  - 当基类有 copy assignment operator 时；
  - 当 class 中有 virtual function 时（需要正确设置 vptr）；

- 当 class 的继承体系中有 virtual base class 时；

- C++语言中的虚继承时 copy assignment operator 弱点：
  C++标准没有规定在虚继承时 copy assignment operator 中是否会多次调用共享基类的 copy assignment operator。这样就有可能造成共享基类被赋值多次，造成一些错误，所以程序员应该在使用了 virtual base class 时小心检验 copy assignment operator 里的代码（以确保这样的多次赋值没有问题或者查看编译器是否已经提供了解决方案）。
  因此，尽可能不要允许一个 virtual base class 的拷贝操作，甚至根本不在要任何 virtual base class 中声明数据。
- C++隐式生成的 4 大成员函数，在不是真正需要的情况下都不要自己去声明。
  因为如果是 trivial 的，这些函数不会被真正的合成出来（只存在于概念上），当然也就没有调用的成本了，去提供一个 trivial 的成员反而是不符合效率的。
- 析构函数的执行顺序（**这里侯捷认为的顺序是错误的，而 Lippman 的才是正确的**）：
  - 如果 object 内带有 vptr，那么首先重设相关的 vtable；
  - destructor 函数本身现在会被执行，也就是说 vptr 会在程序员的代码执行之前被重设；
  - 以声明顺序的相反顺序调用 members 的析构函数；
  - 如果有任何直接的（上一层）nonvirtual base classed 拥有 destructor，那么会以其声明顺序的相反顺序被调用；
  - 如果有任何 virtual base classes 拥有 destructor，而当前讨论的这个 class 是最尾端的，那么它们会以其原来的构造顺序的相反顺序被调用。
- 由于析构函数中的重设 vptr 会在任何代码之前被执行，这样就保证了在析构函数中也不具有多态性，从而不会调用子类的函数。*因为此时子类已经不完整了，子类中的成员已经不存在了，而子类的函数有可能需要使用这些成员。*
- **构造函数和析构函数中都不具有多态性**：这并不是语言的弱点，而是正确的语意所要求的（*因为那个时候的对象不完整*）。

第 6 章：执行期语意学

讨论执行期的对象模型的行为，包括临时对象的生命周期和 new、delete 运算符的行为。

1. C++中过多的隐式变换有时候不太容易从程序代码看出来表达式的复杂度。
2. C++保证：全局变量会在第一次用到之前构造好，在 main()结束之前析构掉。
3. C++程序中所有的 Global object 都放置在程序的 data segment 中并清 0，但是它的 constructor 在程序激活时才会被调用。
4. Lippman 建议不要使用那些需要使用静态初始化的 global object（Google C++编程规范也

是如此建议的）。

5. 现在的 C++ Standard 已经强制要求局部静态对象在第一次被使用时才被构造出来。

   这也是 Effective C++ 中 Singleton 手法所利用的。

   而且在程序结束时会被以构造的相反次序被摧毁。

6. 对象的数组都是通过编译器安插一个函数调用的代码来实现的：

```
void* vec_new(void *array,          // address of start of array
  size_t elem_size,                 // size of each class object
  int elem_count,                   // number of elements in array
  void (*constructor)( void* ),        //构造函数的指针
  void (*destructor)( void*, char ))   //析构函数的指针
{}
void* vec_delete(void *array,  // address of start of array
  size_t elem_size,            // size of each class object
  int elem_count,              // number of elements in array
  void (*destructor)( void*, char ))
{}
```

   由于把数组的声明转换为 vec_new 的函数调用，产生的问题是构造函数是通过指针调用的，因此无法使用任何参数，默认参数也不行。

   对于那些声明了默认参数从而实际上拥有无参构造函数的类，编译器会产生一个绝对无参的构造函数，再从这个构造函数里调用这个默认参数的构造函数。（这样，编译器实际上违反了语言的规定，拥有了 2 个没有参数的构造函数，但是这样的特例只能由编译器自己来违反）

   这里之所以传进了析构函数的指针，是为了在构造函数抛出异常时，把已经构造好的对象给析构掉，这是 vec_new 义不容辞的任务。

7. new 的两步曲：

   1. 分配内存；
   2. 调用构造函数。

8. delete 的两步曲：

   1. 调用析构函数；
   2. 释放内存。

9. 一般的 library 对 new 运算符的实现：

```
extern void* operator new(size_t size) {
  if (size == 0)
    size = 1;
  void *last_alloc;
  while (!(last_alloc = malloc(size))) {
    if (_new_handler)
      (*_new_handler)();
    else
```

```
        return 0;
    }
    return last_alloc;
}
```

有2个精巧之处。第一：new 操作符至少会返回1个字节的内存；第二：_new_handler 会给予内存分配不足时以补救的机会。

10. 虽然 C++ Standard 并没有规定，但是实际上的 new 运算符都是以 C malloc()完成；同样 delete 运算符也都是以 C free()完成的。
11. trivial 的 vec_new()：

    如果要分配的数组的类型并没有定义默认构造函数，那么这个 vec_new()的调用就是 trivial 的，完全可以仅仅分配内存就可以了，new 操作符（注意区分 new 操作符和 new 运算符）足以胜任这个任务。只有在定义了默认构造函数时，vec_new 才需要被调用起来。
12. delete 和 delete []

    寻找数组维度给 delete 运算符带来了效率上的影响，所以出现了这个妥协。只有在[]出现时，编译器才会去寻找数组的维度，否则它就假设只有一个 object 需要被删除。

    （1）：delete 数组时，只有第 1 个元素会被删除；

    （2）：delete [] 单个对象时，1 个元素都不会被删除，没有任何析构函数被调用。
13. 数组的大小会被编译器记录在某个地方，所以编译器能够直接查询出来某个数组的大小。
14. **数组和多态行为的天生不兼容性**：

    永远不要把数组和多态扯到一起，他们天生是不兼容的。当你对一个指向派生类的基类指针进行 delete [] pbase; 操作时，它是不会有正确的语意的。

    这是由于 delete []实际上会使用 vec_delete()类似的函数调用代替，而在 vec_delete()的参数中已经传递了元素的大小，在 vec_delete 中的迭代删除时，会在删除一个指针之后将指针向后移动 item_size 个位置，如果 DerivedClass 的 size 比 BaseClass 要大的话（通常都是如此），指针就已经指向了一个未知的区域了（如果 Derived 与 Base 大小相同，那碰巧不会发生错误，delete []可以正确的执行）。
15. placement operator new 应该与 placement operator delete 搭配使用，也可以在 placement operator new 出来的对象上显式的调用它的析构函数使得原来的内存又可以被再次使用。

    一般而言，placement operator new 并不支持多态，因为 Derived Class 往往比 Base Class 要大，已经存在的类型为 Base 内存并不一定能够容纳 Derived 类型的对象。
16. 一段比较晦涩隐暗的代码：

```
class Base{
public:
    virtual ~Base(){}
    virtual void f(){ cout << "f in Base" << endl; }
```

```
  int value;
};
class Derived : public Base{
public:
  virtual void f(){cout << "f in Derived" << endl;}
};
int main() {
  Base b;
  b.f();    //这个调用很明朗
  b.~Base();
  new (&b) Derived;
  b.f();    //这里应该调用哪个 f()呢？？？
  return 0;
}
```

大部分人认为这里应该输出"f in Derived"，但实际上 GCC 输出的是"f in Base"。

如果理解了前面的编译器如何扩展函数调用，就会明白输出"f in Base"才是正确的。因为 b 是一个对象而不是指针或者引用不具有多态性，所以编译器会：

把 b.f()直接扩展为 Baes::f(&b);

*因此，可以想象，如果把 b 换成是 Base *类型，则由于指针会引发多态，所以才调用 Derived 的 f()函数：*

```
int main() {
  Base *b = new Base();
  b->f();
  b->~Base();
  new (b) Derived;
  b->f();
  return 0;
}
```

这次，GCC 输出了"f in Derived"。

17. C++ Standard 允许编译器对临时性对象的产生有完全的自由度。
18. 临时对象的摧毁时机：摧毁临时对象应该在产生它的**完整的表达式**的最后一个步骤。
    切记是完整的表达式，比如一连串的逗号或一堆的括号，只有在完整的表达式最后才能保证这个临时对象在后面不会再被引用到。
19. 如果一个临时性对象被绑定于一个 reference，对象将残留，直到被初始化之 reference 的生命结束，或者直到临时对象的生命范畴（scope）结束—视哪一种情况先到达而定。
20. ***总结**：临时性对象的确在一些场合、一定程度上影响了 C++的效率。但是这些影响完全可以通过良好的编码和编译器的积极优化而解决掉临时性对象带来的问题（至少在很大的程度上），所以对临时性对象的影响不能大意但也不必太放在心上。*

第 7 章：站在对象模型的尖端

讨论了 C++的三个著名扩展：template, exception handling, RTTI。

1. 编译器在看到一个模板的声明时会做出什么反映呢？实际上编译器没有任何反映！编译器的反映只有在真正具现化时才会发生。
   明白了这个，就明白了为什么在模板内部有明显的语法错误，编译器也不会报错，除非你要具现化出这个模板的一具实体时编译器才会发出抱怨。
   在这点上，似乎 GCC 做的比 MSVC++要好的多。GCC 好像会做完全的解析，但是除了类型的检验；而 MSVC++似乎就是放任不管，只有在具现化的时候才去检查。
   *在学习了C++ Templates 就明白了，编译器实际上会做**二阶段查找**，而且这种延迟到实例化时的具体行为是：延迟定义，而不是声明。*
2. 声明一个模板类型的指针是不会引起模板的具现化操作的，因为仅仅声明指针不需要知道 class 的实际内存布局。
3. 只有在某个 member function 真正被使用时，它才会被真正的具现化出来，这样的延迟具现化至少有 2 个好处：
   1. 空间和时间上的效率；
   2. 如果使用的类型并不完全支持所有的函数，但是只需要不去用那些不支持函数，这样的部分具现化就能得以通过编译。
4. int 和 long 的一致性：int 和 long 在大多数的机器上都是相同的，但是如果编译器看到如下声明：
   Point<int> p1;
   Point<long> p2;
   目前的所有编译器都会具现化 2 个实体。
   可以想象，编译器用一些 mangling 的手法把具现出来的 2 个实体分别叫做：__Point_Int, __Point_Long 之类的东西。
5. 涉及 Template 时的错误检查太弱了，template 中那些与语法无关的错误，程序员可能认为十分明显，编译器却放它通过了，只有在特定的实体被具现化时，编译器才发出抱怨，这是目前实现技术上的一个大问题（*二阶段查找的必然结果*）。
6. Template 中的名称决议方式：**scope of the template definition（定义模板的地方）** 和 **scope of the template instantiation（具现出模板实体的地方）**。示例如下：

```
// scope of the template definition
extern double foo ( double );
template < class type >
class ScopeRules
{
```

```
public:
  void invariant() {              //情况 1
    _member = foo( _val );
  }
  type type_dependent() {         //情况 2
    return foo( _member );
  }
  // …
private:
  int _val;
  type _member;
};

//scope of the template instantiation
extern int foo( int );
ScopeRules< int > sr0;
```

Template 中，对于一个 nonmember name 的决议结果是根据这个 name 的使用是否与"用以具现出该 template 的参数类型"有关而决定的。

*当时觉得这样的规则很诡异，然后在学习了 C++ Templates 之后就很清晰的明白了。这是因为前面一个是**非依赖名称**；而后面的使用是**依赖名称**，所以会在不同的时机进行查找（**二阶段查找嘛！**）。*

1. 情况 1：如果其使用互不相关，那就以 scope of template declaration 来决议 name；
2. 情况 2：如果其使用互有关系，那就以 scope of template instantiation 来决议 name；
3. 这个看似很诡异的规则，实际上是非常必要的！这给予了一个调用者可以进行自定义的机会。模板的使用者往往可以在使用时，根据具体的调用类型来提供一个更好的函数给模板（就像示例中，提供了一个完全符合 int 类型的函数，可以视为一个更好的函数）。
4. 与参数无关的调用，就是站在模板设计者的角度来看，自然就使用 scope of template declaration；

   而与参数相关的调用，就是站在模板使用者的角度来看，当然也就使用 scope of the template instantiation。
5. 还需要非常注意的一点就是：这里依据是否与类型相关而决定的是使用了哪一个 scope，然后再其中搜寻适当的 name。示例中的代码，在调用 sr0.type_dependent();时，由于使用了 scope of the template instantiation，使得 2 个 foo()函数同时成为备选函数，但是由于 foo(int)更加的符合，所以最后才决议使用 foo(int)这个版本。

   如果 sr0 是 ScopeRules<double>类型的话，最后调用的依然是 foo(double)那个版本。
6. 编译器维持了 2 个 scope contexts:
   1. scope of template declaration：用以专注一般的 template class；

2. scope of template instantiation：用以专注于特定的实体；

7. 这种关联性不能简单的使用一个宏扩展来重现，是一种很新奇的关联。

7. 一种具现化的策略：先不具现任何的 member function，链接器会登记缺少哪些函数的定义，然后再重新调用编译器把登记在册的缺乏的定义重写编译出来，最后在把这些缺乏的定义和以前的链接结果链接起来形成最后的可执行文件或者库。

8. 如果 vtable 被具现出来，那么每一个 virtual function 也都必须被具现。
   这就是为什么 C++ Standard 中有如下的描述："如果一个虚函数被具现出来，其具现点紧跟在其 class 的具现点之后"。（也就是说，virtual function 是一口气被具现出来的）

9. 一般而言，exception handling 机制需要与编译器所产生的数据结构以及执行期的一个 exception library 紧密合作而实现。

10. 编译器为了支持异常的机制，又需要把程序员的代码进行大量的扩展才能保证异常机制的正确执行。但是处理这些问题是编译器的责任，不过程序员应该明白这里把做的事情以及有可能付出的代价。

11. 以值类型抛出异常，以引用类型进行捕获：
    被抛出的异常类型，一定会被先复制一份，真正被抛出的实际上是这份复制器；
    即使是以值类型来进行捕获异常也可以捕获该值类型和其派生类的异常，但是在 catch 语句中会引发切割。

12. 对于每一个被丢出的 exception，编译器必须产生一个类型描述器，对 exception 类型进行编码。如果那是一个 derived type，则编码内容还必须包括其所有 base class 的类型信息。

13. 当一个 exception 被丢出时，exception object 会被产生出来并通常放置在相同形式的 exception 数据堆栈中。从 throw 端传染给 catch 子句的是 exception object 的地址、类型描述器（或是一个函数指针，该函数会返回该 exception type 有关的类型描述器对象），以及有可能还有的 exception object 的析构函数的地址（如果有的话）（**中文书翻译有误**）。

14. 只有在一个 catch 子句评估完毕并且知道它不会再丢出 exception 之后，真正的 exception object 才会被摧毁。

15. 支持异常机制的代价：与其它语言特征相比较，C++编译器支持 EH 机制所付出的代价最大。
    C++对异常机制所付出的代价大概为：空间 10%、时间 5%。不算小，但也不是不可以接受吧。
    有一个问题：如果编译器开启了异常支持，但是在某一段未使用异常的代码中，也会为编译器开启了异常支持而付出代价吗？

16. 在 C++中，一个具备多态性质的 class，就是指内含 virtual functions 的类（直接声明或者继承而来的）。

17. 由于具备多态性质的 class 都已经含有一个 vptr 指向 vtable 了，C++把类型信息放在 vtable 的第 1 个 slot 中（一个 type_info 的指针指向一个表示当前类型的 type_info 对象），从而几乎没有付出代价的支持了 RTTI（1byte per class, not 1byte per class object）（**中文书翻译有误，侯捷错误的写成了 1byte per class object**）。
    由于 RTTI 所需要的信息放在 vtable 中，自然的：只有含有 vptr 的类才支持 RTTI。

18. 有了 RTTI 机制的支持，就可以实施保证安全的动态转型操作 dynamic_cast<>();
19. 在 dynamic_cast 中使用指针和引用的区别在于当转型失败时：

    指针版本会返回 0，使用者需要进行检查；

    引用的版本会抛出一个 bad_cast exception（***因为没有空引用啊***）；

    这两种机制各有用处吧，视需求而用。
20. type_info 类型的 copy 构造函数和 operator=操作符都被声明为私有，禁止了赋值和拷贝操作。而且只提供了一个受保护的带有一个 const char *参数的构造函数，因为不能直接得到 type_info 对象，只能通过 typeid()运算符来得到这类对象。
21. RTTI 只适用于多态类型（RTTI 信息存于 vtable 的原因），事实上 type_info object 也适用于非多态类型。typeid()使用于非多态类型时的差异在于，这时候的 type_info object 是静态取得的（编译器直接给扩展了），而非像多态类型一样在执行期通过 vtable 动态取得。

    这之间的区别看下面的这个例子就会很快明白了：

```
struct A{};                    //A 是非多态类型
struct B : public A{};         //B 也是非多态类型
int main() {
  A *pa = new B;
  cout << typeid(pa).name() << endl;
  cout << typeid(*pa).name() << endl;
}
```

    将输出：

    Struct A *

    Struct A

    这没有检测出 pa 所指的真正类型，原因就在于 typeid 运算符用在非多态类型上时，会被编译器在编译期间静态的扩展了。

    也许是类似的扩展：

```
typeid(pa).name()  => typeid(A*).name()
typeid(*pa).name() => typeid(A).name()
```

    如果给 struct A 添加一个虚拟函数，从而使得类型 A 和 B 都变成多态类型，于是 typeid 运算符就会在运行期间动态的去获取它们的真正类型了。

```
struct A{
  virtual ~A(){}            //A 包含了一个虚函数，从而把 A 变成了多态类型
};
struct B : public A{};      //B 从 A 继承了一个虚函数，所以也是多态类型
int main() {
  A *pa = new B;
  cout << typeid(pa).name() << endl;
  cout << typeid(*pa).name() << endl;
}
```

将输出：

Struct A *

Struct B

22. 效率和弹性始终是对矛盾体！