C++对象面面观

学习 C++应该看过不少关于 C 与 C++的口水贴，以及关于各种对比 C 与 C++效率比较的帖子， 最有影响力的恐怕当属 linus 对 C++的炮轰——《糟糕程序员的垃圾语言》。但无论如 何，我正喜欢着这样一种垃圾，我当然对 linus 充满敬意，但这不妨碍我口食垃圾而对其 仰慕。

无需太在意站在山巅的巨人们的言论，每个人都有不同的道路来追求真理。与其听着 Linus 的嗤笑之声，不妨跟随 Lippman 一起探索 C++对象模型的"内心世界"。若要对事物褒贬，总 得先对其了解。唯有了解才能有负责的发声。

——仅以此区区百余字为之前记。

C++的额外成本

C++较之 C 的最大区别，无疑在于面向对象。类相较于 C 的 struct 不仅只包含了数据，同时还 包括了对于数据的操作。在语言层面上 C++带来了很多面向对象的新特性，类、继承、多态 等等。新特性使得 C++更加强大，但同时却伴随着空间布局和存取时间的额外成本。作为一 个以效率为目标的语言，C++对于面向对象的实现，其实不大，这些额外成本主要由 virtual 引起，包括：

- virtual function 机制，用来支持"执行期绑定"。
- virtual base class ——虚基类机制，以实现共享虚基类的 subobject。

除此之外 C++没有太多理由比 C 迟缓。

三种对象模型

C++类包含两种数据成员：静态数据成员和非静态数据成员；同时包含成员函数，静态函数 和虚函数三种成员函数，这些机制在 C++对象是如何被表现的？下面有三种模型可以用以表 现它们——简单对象模型、表格驱动对象模型以及 C++对象模型。也许你没兴趣去了解有几种 方式可以实现 C++的对象模型，只想了解 C++对象模型。然则，C++对象模型是在前两种对象 模型上发展而来的，甚至于局部上直接用到前两种对象模型。

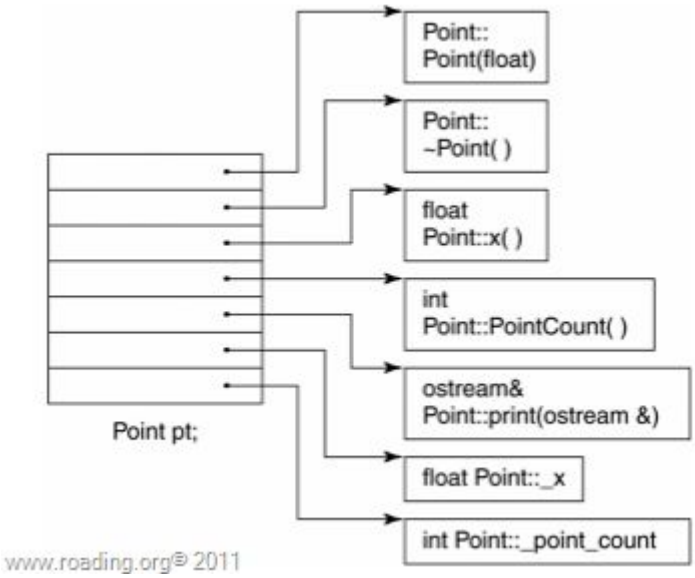
假定有一个 Point 类，我们将用三种对象模型来表现它。Point 类如下：

```
class Point { public:
    Point( float xval );
    virtual ~Point();
    float x() const;
    static int PointCount();protected:
```

```
    virtual ostream& print( ostream &os ) const;
    float _x;
    static int _point_count; };
```

简单对象模型

简单对象模型：一个 C++对象存储了所有指向成员的指针，而成员本身不存储在对象中。也 就是说不论数据成员还是成员函数，也不论这个是普通成员函数还是虚函数，它们都存储 在对象本身之外，同时对象保存指向它们的指针。



简单对象模型对于编译器来说虽然极尽简单,但同时付出的代价是空间和执行期的效率.显而 易见的是对于每一个成员都要额外搭上一个指针大小的空间以及对于每成员的操作都增加了 一个间接层。因此 C++并没有采用这样一种对象模型，但是被用到了 C++中"指向成员的指针" 的概念当中。

表格驱动对象模型

表格驱动模型则更绝，它将对象中所有的成员都抽离出来在外建表，而对象本身只存储指向这个表的指针。右图可以看到，它将所有的数据成员抽离出来建成数据成员表，将所有的函数抽取出来建成一张函数成员表，而对象本身只保持一个指向数据成员表的指针。

侯大大认为，在对象与成员函数表中间应当加一个虚箭头，他认为这是 Lippman 的疏漏之处， 应当在对象中保存指向函数成员表的指针。

然而我在这儿还是保留原书（而非译本）的截图，因为以我之拙见，不保存指向成员函数表的指针也没有妨碍。因为形如 `float Point::x()`的成员函数实际上相当于 `float x(Point*)` 类型的普通函数，因此保存指向成员函数表的指针当属多此一举。

当然 C++也没有采用这一种对象模型，但 C++却以此模型作为支持虚函数的方案。

C++对象模型

所有的非静态数据成员存储在对象本身中。所有的静态数据成员、成员函数（包括静态与非静态）都置于对象之外。另外，用一张虚函数表（virtual table)存储所有指向虚函数的指 针，并在表头附加上一个该类的 type_info 对象，在对象中则保存一个指向虚函数表的指 针。如下图：

class 和 struct

按照 lippman 的意思是，struct 仅仅是给想学习 C++的 C 程序员攀上高峰少一点折磨。但遗憾的 是当我开始学 C++的时候这个问题给我带来更多的疑惑。以我的认识 class 与 struct 仅限一个 默认的权限（后者为 public 前者为 private）的不同。有时我甚至觉得只有一点畸形，他们不 应当如此的相像，我甚至认为 struct 不应该被扩充，仅仅保存它在 C 中的原意就好了。[1]

一个有意思的 C 技巧（但别在 C++中使用）

在 C 中将一个一个元素的数组放在 struct 的末尾，可以令每个 struct 的对象拥有可变数组。看代码：

```
struct mumble {
  /* stuff */
  char pc[ 1 ]; }; // grab a string from file or standard input  // allocate memory both for struct & string  struct mumble *pmumb1 = ( struct mumble* )
  malloc(sizeof(struct mumble)+strlen(string)+1); strcpy( &mumble.pc, string );
```

这是一个很有意思的小技巧，但是别在 C++中使用。因为 C++的内存布局相对复杂。例如被继 承，有虚函数... 问题将不可避免的发生。

三种编程典范

- 程序模型
- ADT 模型
- 面向对象模型

纯粹使用一种典范编程，有莫大的好处，如果混杂多种典范编程有可能带来意想不到的后果 ，例如将继承类的对象赋值给基类对象，而妄想实现多态，便是一种 ADT 模型和面向对象模型 混合编程带来严重后果的例子。

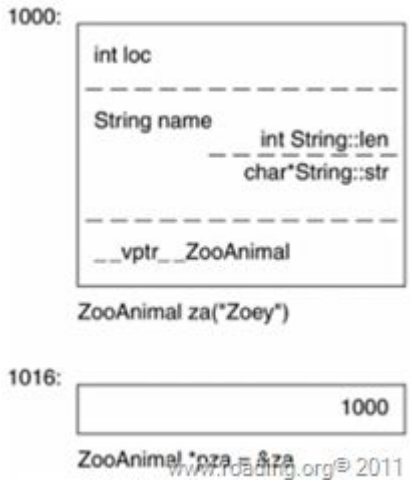
一个类的对象的内存大小包括：

- 所有非静态数据成员的大小。
- 由内存对齐而填补的内存大小。
- 为了支持 virtual 有内部产生的额外负担。

如下类：

```
class ZooAnimal {public:
ZooAnimal();
virtual ~ZooAnimal();
virtual void rotate();protected:
int loc;
String name;};
```

在 32 位计算机上所占内存为 16 字节：int 四字节，String8 字节（一个表示长度的整形，一个 指向字符串的指针），以及一个指向虚函数表的指针 vptr。对于继承类则为基类的内存大小 加上本身数据成员的大小。在 cfront 中其内存布局如下图：



实际上 struct 还要复杂一点，它有时表现的会和 C struct 完全一样，有时则会成为 class 的胞兄弟。