——谈 VC++对象模型

（美）简.格雷
程化　译

译者前言

一个 C++程序员，想要进一步提升技术水平的话，应该多了解一些语言的语意细节。对于使用 VC++的程序员来说，还应该了解一些 VC++对于 C++的诠释。 Inside the C++ Object Model 虽然是一本好书，然而，书的篇幅多一些，又和具体的 VC++关系小一些。因此，从篇幅和内容来看，译者认为本文是深入理解 C++对象模型比较好的一个出发点。
这篇文章以前看到时就觉得很好，旧文重读，感觉理解得更多一些了，于是产生了翻译出来，与大家共享的想法。虽然文章不长，但时间有限，又若干次在翻译时打盹睡着，拖拖拉拉用了小一个月。
一方面因本人水平所限，另一方面因翻译时经常打盹，错误之处恐怕不少，欢迎大家批评指正。

本文原文出处为 MSDN。如果你安装了 MSDN，可以搜索到 C++ Under the Hood。否则也可在网站上找到 http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/jangrayhood.asp 。

1 前言

了解你所使用的编程语言究竟是如何实现的，对于 C++程序员可能特别有意义。首先，它可以去除我们对于所使用语言的神秘感，使我们不至于对于编译器干的活感到完全不可思议；尤其重要的是，它使我们在 Debug 和使用语言高级特性的时候，有更多的把握。当需要提高代码效率的时候，这些知识也能够很好地帮助我们。

本文着重回答这样一些问题：
1* 类如何布局？
2* 成员变量如何访问？
3* 成员函数如何访问？
4* 所谓的“调整块”（adjuster thunk）是怎么回事？
5* 使用如下机制时，开销如何：
  * 单继承、多重继承、虚继承
  * 虚函数调用
  * 强制转换到基类，或者强制转换到虚基类
  * 异常处理
首先，我们顺次考察 C 兼容的结构（struct）的布局，单继承，多重继承，以及虚继承；
接着，我们讲成员变量和成员函数的访问，当然，这里面包含虚函数的情况；
再接下来，我们考察构造函数，析构函数，以及特殊的赋值操作符成员函数是如何工作的，数组是如何动态构造和销毁的；
最后，简单地介绍对异常处理的支持。

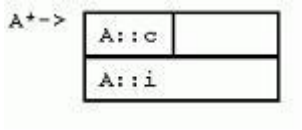
对每个语言特性，我们将简要介绍该特性背后的动机，该特性自身的语意（当然，本文决不是“C++入门”，大家对此要有充分认识），以及该特性在微软的 VC++中是如何实现的。这里要注意区分抽象的 C++语言语意与其特定实现。微软之外的其他 C++厂商可能提供一个完全不同的实现，我们偶尔也会将 VC++的实现与其他实现进行比较。

2 类布局

本节讨论不同的继承方式造成的不同内存布局。

2.1 C 结构（struct）

由于 C++基于 C，所以 C++也“基本上”兼容 C。特别地，C++规范在“结构”上使用了和 C 相同的，**简单的内存布局原则：成员变量按其被声明的顺序排列，按具体实现所规定的对齐原则在内存地址上对齐。** 所有的 C/C++厂商都保证他们的 C/C++编译器对于有效的 C 结构采用完全相同的布局。这里，A 是一个简单的 C 结构，其成员布局和对齐方式都一目了然



view plaincopy to clipboardprint?

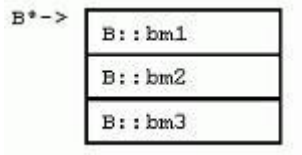
```
1. struct A {
2.     char c;
3.     int i;
4. };
```

译者注：从上图可见，A 在内存中占有 8 个字节，按照声明成员的顺序，前 4 个字节包含一个字符（实际占用 1 个字节，3 个字节空着，补对齐），后 4 个字节包含一个整数。A 的指针就指向字符开始字节处。

2.2 有 C++特征的 C 结构

当然了，C++不是复杂的 C，C++本质上是面向对象的语言：包 含 **继承、封装，以及多态** 。原始的 C 结构经过改造，成了面向对象世界的基石——类。除了成员变量外，C++类还可以封装成员函数和其他东西。然而，有趣的是，**除非 为了实现虚函数和虚继承引入的隐藏成员变量外，C++类实例的大小完全取决于一个类及其基类的成员变量！成员函数基本上不影响类实例的大小。**

这里提供的 B 是一个 C 结构，然而，该结构有一些 C++特征：控制成员可见性的“public/protected/private”关键字、成员函数、静态成员，以及嵌套的类型声明。虽然看着琳琅满目，实际上，**只有成员变量才占用类实例的空间** 。要注意的是，C++标准委员会不限制由“public/protected/private”关键字分开的各段在实现时的先后顺序，因此，不同的编译器实现的内存布局可能并不相同。**（ 在 VC++中，成员变量总是按照声明时的顺序排列）。**



view plaincopy to clipboardprint?

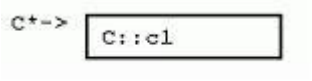
```
1. struct B {
2. public:
3.     int bm1;
4. protected:
5.     int bm2;
6. private:
7.     int bm3;
8.     static int bsm;
9.     void bf();
```

```
10.     static void bsf();
11.     typedef void* bpv;
12.     struct N { };
13. };
```

译者注：B 中，为何 static int bsm 不占用内存空间？因为它是**静态成员，该数据存放在程序的数据段** 中，不在类实例中。

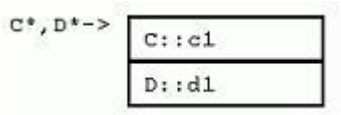
2.3 单继承

C++ 提供继承的目的是在不同的类型之间提取共性。比如，科学家对物种进行分类，从而有种、属、纲等说法。有了这种层次结构，我们才可能将某些具备特定性质的东西归入到最合适的分类层次上，如“怀孩子的是哺乳动物”。由于这些属性可以被子类继承，所以，我们只要知道“鲸鱼、人”是哺乳动物，就可以方便地指出“鲸鱼、人都可以怀孩子”。那些特例，如鸭嘴兽（生蛋的哺乳动物），则要求我们对缺省的属性或行为进行覆盖。
C++中的继承语法很简单，在子类后加上“:base”就可以了。下面的 D 继承自基类 C。




```
view plaincopy to clipboardprint?
```

```
1.  struct  C  {
2.          int  c1;
3.          void  cf();
4.  };
```




```
view plaincopy to clipboardprint?
```

```
1.  struct  D  :  C  {
2.          int  d1;
3.          void  df();
4.  };
```

既然派生类要保留基类的所有属性和行为，自然地，每个派生类的实例都包含了一份完整的基类实例数据。在 D 中，并不是说基类 C 的数据一定要放在 D 的数据之前，只不过这样放的话，能够保证 D 中的 C 对象地址，恰好是 D 对象地址的第一个字节。这种安排之下，**有了派生类 D 的指针，要获得基类 C 的指针，就不必要计算偏移量 了。几乎所有知名的 C++厂商都采用这种内存安排（基类成员在前）。 在单继承类层次下，每一个新的派生类都简单地把自己的成员变量添加到基类的成员变量之后 。** 看看上图，C 对象指针和 D 对象指针指向同一地址。

2.4 多重继承

大多数情况下，其实单继承就足够了。但是，C++为了我们的方便，还提供了多重继承。

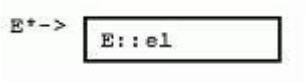
比如，我们有一个组织模型，其中有经理类（分任务），工人类（干活）。那么，对于一线经理类，即既要从上级经理那里领取任务干活，又要向下级工人分任务的角色来说，如何在类层次中表达呢？单继承在此就有点力不胜任。我们可以安排经理类先继承工人类，一线经理类再继承经理类，但这种层次结构错误地让经理类继承了工人类的属性和行为。反之亦然。当然，一线经理类也可以仅仅从一个类（经理类或工人类）继承，或者一个都不继承，重新声明一个或两个接口，但这样的实现弊处太多：多态不可能了；未能重用现有的接口；最严重的是，当接口变化时，必须多处维护。最合理的情况似乎是一线经理从两个地方继承属性和行为——经理类、工人类。

C++就允许用多重继承来解决这样的问题：

```
view plaincopy to clipboardprint?
```

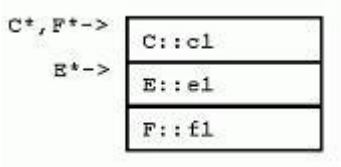
```
1. struct Manager ... { ... };
2. struct Worker ... { ... };
3. struct MiddleManager : Manager, Worker { ... };
```

这样的继承将造成怎样的类布局呢？下面我们还是用“字母类”来举例：




```
view plaincopy to clipboardprint?
```

```
1. struct E {
2.         int e1;
3.         void ef();
4. };
```




```
view plaincopy to clipboardprint?
```

```
1. struct F : C, E {
2.         int f1;
3.         void ff();
4. };
```

结构 F 从 C 和 E 多重继承得来。**与单继承相同的是，F 实例拷贝了每个基类的所有数据。 与单继承不同的是，在多重继承下，内嵌的两个基类的对象指针不可能全都与派生类对象指针相同：**

```
view plaincopy to clipboardprint?
```

```
F f;
// (void*)&f == (void*)(C*)&f;
// (void*)&f < (void*)(E*)&f;
```

译者注：上面那行说明 C 对象指针与 F 对象指针相同，下面那行说明 E 对象指针与 F 对象指针不同。

观察类布局，可以看到 F 中内嵌的 E 对象，其指针与 F 指针并不相同。正如后文讨论强制转化和成员函数时指出的，这个偏移量会造成少量的调用开销。

具体的编译器实现可以自由地选择内嵌基类和派生类的布局。 VC++ 按照基类的声明顺序 先排列基类实例数据，最后才排列派生类数据。 当然，派生类数据本身也是按照声明顺序布局的（**本规则并非一成不变** ，我们会看到，当一些基类有虚函数而另一些基类没有时，内存布局并非如此）。

2.5 虚继承

回到我们讨论的一线经理类例子。让我们考虑这种情况：如果经理类和工人类都继承自“雇员类”，将会发生什么？

```
view plaincopy to clipboardprint?
```

```
struct Employee { ... };
struct Manager : Employee { ... };
struct Worker : Employee { ... };
struct MiddleManager : Manager, Worker { ... };
```

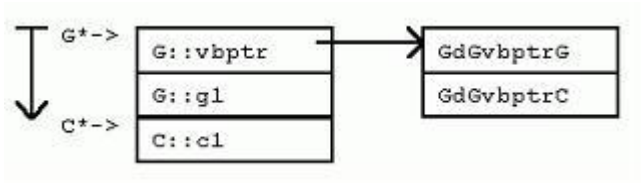
如果经理类和工人类都继承自雇员类，很自然地，它们每个类都会从雇员类获得一份数据拷贝。**如 果不作特殊处理，一线经理类的实例将含有两个 雇员类实例，它们分别来自两个雇员基类 。** 如果雇员类成员变量不多，问题不严重；如果成员变量众多，则那份多余的拷贝将造成实例生成时的严重开销。更糟的是，这两份不同的雇员实例可能分别被修改，造成数据的不一致。因此，我们需要让经理类和工人类进行特殊的声明，说明它们愿意共享一份雇员基类实例数据。

很不幸，在 C++中，这种“共享继承”被称为**“虚继承”** ，把问题搞得似乎很抽象。虚继承的语法很简单，在指定基类时加上 virtual 关键字即可。

```
view plaincopy to clipboardprint?
```

```
struct Employee { ... };
struct Manager : virtual Employee { ... };
struct Worker : virtual Employee { ... };
struct MiddleManager : Manager, Worker { ... };
```

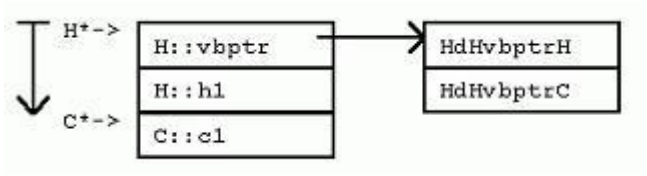
使用虚继承，比起单继承和多重继承有更大的实现开销、调用开销。回忆一下，**在单继承和多重继承的情况下，内嵌的基类实例地址比起派生类实例地址来，要么地址相同（单继承，以及多重继承的最靠左基类） ，要么地址相差一个固定偏移量（多重继承的非最靠左基类） 。 然而，当虚继承时，一般说来，派生类地址和其虚基类地址之间的偏移量是不固定的，因为如果这个派生类又被进一步继承的话，最终派生类会把共享的虚基类实例数据放到一个与上一层派生类不同的偏移量处。** 请看下例：



view plaincopy to clipboardprint?

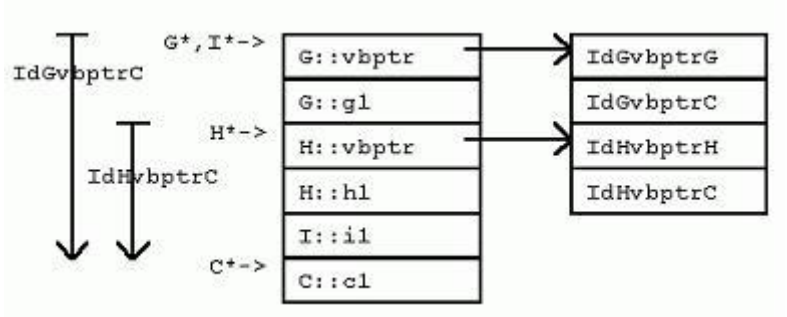
```
1. struct G : virtual C {
2.         int g1;
3.         void gf();
4. };
```

译者注：GdGvbptrG（In G, the displacement of G’s virtual base pointer to G）意思是：在G中，G对象的指针与G的虚基类表指针之间的偏移量，在此可见为0，因为G对象内存布局第一项就是虚基类表指针；GdGvbptrC（In G, the displacement of G’s virtual base pointer to C）意思是：在G中，C对象的指针与G的虚基类表指针之间的偏移量，在此可见为8。



view plaincopy to clipboardprint?

```
1. struct H : virtual C {
2.         int h1;
3.         void hf();
4. };
```



view plaincopy to clipboardprint?

```
1. struct I : G, H {
2.         int i1;
3.         void _if();
4. };
```

暂时不追究 vbptr 成员变量从何而来。从上面这些图可以直观地看到，在 G 对象中，内嵌的 C 基类对象的数据紧跟在 G 的数据之后，在 H 对象中，内嵌的 C 基类对象的数据也紧跟在 H 的数据之后。但是，在 I 对象中，内存布局就并非如此了。VC++实现的内存布局中，G 对象实例中 G 对象和 C 对象之间的偏移，不同于 I 对象实例中 G 对象和 C 对象之间的偏移。当使用指针访问虚基类成员变量时，由于指针可以是指向派生类实例的基类指针，所以，编译器不能根据声明的指针类型计算偏移，而必须找到另一种间接的方法，从派生类指针计算虚基类的位置。

在 VC++ 中，对每个继承自虚基类的类实例，将增加一个隐藏的*“虚基类表指针”（vbptr）*成员变量，从而达到间接计算虚基类位置的目的。该变量指向一个全类共享的偏移量表，表中项目记录了对于该类 而言，“虚基类表指针”与虚基类之间的偏移量。

其它的实现方式中，有一种是在派生类中使用指针成员变量。这些指针成员变量指向派生类的虚基类，每个虚基类一个指针。这种方式的优点是：获取虚基类地址时，所用代码比较少。然而，编译器优化代码时通常都可以采取措施避免重复计算虚基类地址。况且，这种实现方式还有一个大弊端：从多个虚基类派生时，类实例将占用更多的内存空间；获取虚基类的虚基类的地址时，需要多次使用指针，从而效率较低等等。

在 VC++中，G 拥有一个隐藏的“虚基类表指针”成员，指向一个虚基类表，该表的第二项是 G dGvbptrC。（在 G 中，虚基类对象 C 的地址与 G 的“虚基类表指针”之间的偏移量 （ 当对于所有的派生类来说偏移量不变时，省略“d”前的前缀））。比如，在 32 位平台上，GdGvptrC 是 8 个字节。同样，在 I 实例中的 G 对象实例也有 “虚基类表指针”，不过该指针指向一个适用于“G 处于 I 之中” 的虚基类表，表中一项为 IdGvbptrC，值为 20。

观察前面的 G、H 和 I， 我们可以得到如下关于 VC++虚继承下内存布局的结论：

1 首先排列非虚继承的基类实例；
2 有虚基类时，为每个基类增加一个隐藏的 vbptr，除非已经从非虚继承的类那里继承了一个 vbptr；
3 排列派生类的新数据成员；
4 在实例最后，排列每个虚基类的一个实例。

该布局安排使得虚基类的位置随着派生类的不同而“浮动不定”，但是，非虚基类因此也就凑在一起，彼此的偏移量固定不变。

3 成员变量

介绍了类布局之后，我们接着考虑对不同的继承方式，访问成员变量的开销究竟如何。

没有继承： 没有任何继承关系时，访问成员变量和 C 语言的情况完全一样：从指向对象的指针，考虑一定的偏移量即可。

```
view plaincopy to clipboardprint?
```

```
1.  C*  pc;
2.  pc->c1;  //  *(pc  +  dCc1);
```

译者注：pc 是指向 C 的指针。
a. 访问 C 的成员变量 c1，只需要在 pc 上加上固定的偏移量 dCc1（在 C 中，C 指针地址与其 c1 成员变量之间的偏移量值），再获取该指针的内容即可。

单继承： 由于派生类实例与其基类实例之间的偏移量是常数 0，所以，可以直接利用基类指针和基类成员之间的偏移量关系，如此计算得以简化。

```
view plaincopy to clipboardprint?
```

```
1.  D*  pd;
2.  pd->c1;  //  *(pd  +  dDC  +  dCc1);  //  *(pd  +  dDc1);
```

```
3.  pd->d1;  //  *(pd  +  dDd1);
```

译者注：D 从 C 单继承，pd 为指向 D 的指针。
a. 当访问基类成员 c1 时，计算步骤本来应该为“pd+dDC+dCc1”，即为先计算 D 对象和 C 对象之间的偏移，再在此基础上加上 C 对象指针与成员变量 c1 之间的偏移量。然而，由于 dDC 恒定为 0，所以直接计算 C 对象地址与 c1 之间的偏移就可以了。
b. 当访问派生类成员 d1 时，直接计算偏移量。

多重继承 ：虽然派生类与某个基类之间的偏移量可能不为 0，然而，该偏移量总是一个常数。只要是个常数，访问成员变量，计算成员变量偏移时的计算就可以被简化。可见即使对于多重继承来说，访问成员变量开销仍然不大。

view plaincopy to clipboardprint?

```
1.  F*  pf;
2.  pf->c1;  //  *(pf  +  dFC  +  dCc1);  //  *(pf  +  dFc1);
3.  pf->e1;  //  *(pf  +  dFE  +  dEe1);  //  *(pf  +  dFe1);
4.  pf->f1;  //  *(pf  +  dFf1);
```

译者注：F 继承自 C 和 E，pf 是指向 F 对象的指针。
a. 访问 C 类成员 c1 时，F 对象与内嵌 C 对象的相对偏移为 0，可以直接计算 F 和 c1 的偏移；
b. 访问 E 类成员 e1 时，F 对象与内嵌 E 对象的相对偏移是一个常数，F 和 e1 之间的偏移计算也可以被简化；
c. 访问 F 自己的成员 f1 时，直接计算偏移量。

虚继承： **当类有虚基类时，访问非虚基类的成员仍然是计算固定偏移量的问题。然而，访问虚基类的成员变量，开销就增大了** ， 因为必须经过如下步骤才能获得成员变量的地址：
1. 获取“虚基类表指针”；
2. 获取虚基类表中某一表项的内容；
3. 把内容中指出的偏移量加到“虚基类表指针”的地址上。

然而，事情并非永远如此。正如下面访问 I 对象的 c1 成员那样，**如果不是通过指针访问，而是直接通过对象实例，则派生类的布局可以在编译期间静态获得，偏移量也可以在编译时计算，因此也就不必要根据虚基类表的表项来间接计算了。**

view plaincopy to clipboardprint?

```
1.  I*  pi;
2.  pi->c1;  //  *(pi  +  dIGvbptr  +  (*(pi+dIGvbptr))[1]  +  dCc1);
3.  pi->g1;  //  *(pi  +  dIG  +  dGg1);  //  *(pi  +  dIg1);
4.  pi->h1;  //  *(pi  +  dIH  +  dHh1);  //  *(pi  +  dIh1);
5.  pi->i1;  //  *(pi  +  dIi1);
6.  I  i;
7.  i.c1;  //  *(&i  +  IdIC  +  dCc1);  //  *(&i  +  IdIc1);
```

译者注：I 继承自 G 和 H，G 和 H 的虚基类是 C，pi 是指向 I 对象的指针。
a. 访问虚基类 C 的成员 c1 时，dIGvbptr 是“在 I 中，I 对象指针与 G 的“虚基类表指针”之间的偏移”，*(pi + dIGvbptr)是虚基类表的开始地址，*(pi + dIGvbptr)[1]是虚基类表的第二项的内容（在 I 对象中，G 对象的

“虚基类表指针”与虚基类之间的偏移），dCc1 是 C 对象指针与成员变量 c1 之间的偏移；
b. 访问非虚基类 G 的成员 g1 时，直接计算偏移量；
c. 访问非虚基类 H 的成员 h1 时，直接计算偏移量；
d. 访问自身成员 i1 时，直接使用偏移量；
e. **当声明了一个对象实例，用点“.”操作符访问虚基类成员 c1 时，由于编译时就完全知道对象的布局情况，所以可以直接计算偏移量。**

当访问类继承层次中，多层虚基类的成员变量时，情况又如何呢？比如，访问虚基类的虚基类的成员变量时？一些实现方式为：保存一个指向直接虚基类的指针，然后就可以从直接虚基类找到它的虚基类，逐级上推。VC++优化了这个过程。VC++在虚基类表中增加了一些额外的项，这些项保存了从派生类到其各层虚基类的偏移量。

4 强制转化

如果没有虚基类的问题，将一个指针强制转化为另一个类型的指针代价并不高昂。如果在要求转化的两个指针之间有“基类-派生类”关系，编译器只需要简单地在两者之间加上或者减去一个偏移量即可（并且该量还往往为 0）。

view plaincopy to clipboardprint?

```
1.  F*  pf;
2.  (C*)pf;  //  (C*)(pf  ?  pf  +  dFC  :  0);  //  (C*)pf;
3.  (E*)pf;  //  (E*)(pf  ?  pf  +  dFE  :  0);
```

C 和 E 是 F 的基类，将 F 的指针 pf 转化为 C*或 E*，只需要将 pf 加上一个相应的偏移量。转化为 C 类型指针 C*时，不需要计算，因为 F 和 C 之间的偏移量为 0。转化为 E 类型指针 E*时，必须在指针上加一个非 0 的偏移常量 dFE。**C ++规范要求 NULL 指针在强制转化后依然为 NULL** ，因此在做强制转化需要的运算之前，VC++会检查指针是否为 NULL。当然，这个检查只有当指针被显示或者隐式转化为相关类型指针时才进行；当在派生类对象中调用基类的方法，从而派生类指针在后台被转化为一个基类的 Const “this” 指针时，这个检查就不需要进行了，因为在此时，该指针一定不为 NULL。

正如你猜想的，**当继承关系中存在虚基类时，强制转化的开销会比较大。具体说来，和访问虚基类成员变量的开销相当。**

view plaincopy to clipboardprint?

```
1.  I*  pi;
2.  (G*)pi;  //  (G*)pi;
3.  (H*)pi;  //  (H*)(pi  ?  pi  +  dIH  :  0);
4.  (C*)pi;  //  (C*)(pi  ?  (pi+dIGvbptr  +  (*(pi+dIGvbptr))[1])  :  0);
```

译者注：pi 是指向 I 对象的指针，G,H 是 I 的基类，C 是 G,H 的虚基类。
a. 强制转化 pi 为 G*时，由于 G*和 I*的地址相同，不需要计算；
b. 强制转化 pi 为 H*时，只需要考虑一个常量偏移；
c. 强制转化 pi 为 C*时，所作的计算和访问虚基类成员变量的开销相同，首先得到 G 的虚基类表指针，再从虚基类表的第二项中取出 G 到虚基类 C 的偏移量，最后根据 pi、虚基类表偏移和虚基类 C 与虚基类表指针之间的偏移计算出 C*。

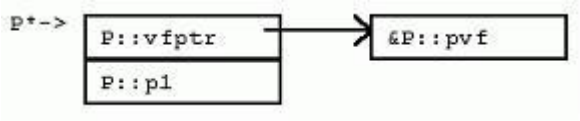
一般说来，当从派生类中访问虚基类成员时，应该先强制转化派生类指针为虚基类指针，然后一直使用虚基类指针来访问虚基类成员变量。这样做，可以避免每次都要计算虚基类地址的开销。 见下例。

```
/* before: */                          ... pi->c1 ... pi->c1 ...
/* faster: */ C* pc = pi; ... pc->c1 ... pc->c1 ...
```

译者注：前者一直使用派生类指针 pi，故每次访问 c1 都有计算虚基类地址的较大开销；后者先将 pi 转化为虚基类指针 pc，故后续调用可以省去计算虚基类地址的开销。

5 成员函数

一个 C++成员函数只是类范围内的又一个成员。**X 类每一个非静态的成员函数都会接受一个特殊的隐藏参数——this 指针，类型为 X* const。** 该指针在后台初始化为指向成员函数工作于其上的对象。同样，在成员函数体内，成员变量的访问是通过在后台计算与 this 指针的偏移来进行。



view plaincopy to clipboardprint?

```
1.  struct  P  {
2.          int  p1;
3.          void  pf();  //  new
4.          virtual  void  pvf();  //  new
5.  };
```

P 有一个非虚成员函数 pf()，以及一个虚成员函数 pvf()。很明显，虚成员函数造成对象实例占用更多内存空间，因为虚成员函数需要虚函数表指针。这一点以后还会谈到。这里要特别指出的是，声明非虚成员函数不会造成任何对象实例的内存开销。现在，考虑 P::pf()的定义。

view plaincopy to clipboardprint?

```
1.  void  P::pf()  {  //  void  P::pf([P  *const  this])
2.          ++p1;       //  ++(this->p1);
3.  }
```

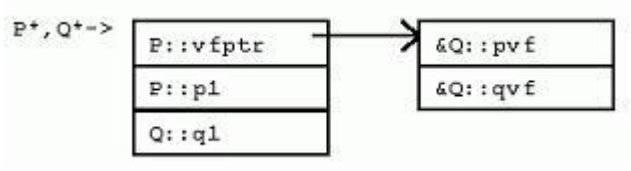
这里 P:pf()接受了一个**隐藏的 this 指针参数** ，对于每个成员函数调用，编译器都会自动加上这个参数。同时，注意成员变量访问也许比看起来要代价高昂一些，因为成员变量访问通过 this 指针进行，在有的继承层次下，this 指针需要调整，所以访问的开销可能会比较大。然而，从另一方面来说，编译器通常会把 this 指针缓存到寄存器中，所以，成员变量访问的代价不会比访问局部变量的效率更差。

译者注：访问局部变量，需要到 SP 寄存器中得到栈指针，再加上局部变量与栈顶的偏移。在没有虚基类的情况下，如果编译器把 this 指针缓存到了寄存器中，访问成员变量的过程将与访问局部变量的开销相似。

5.1 覆盖成员函数

和成员变量一样，成员函数也会被继承。与成员变量不同的是，通过在派生类中重新定义基类函数，一个派生类可以覆盖，或者说替换掉基类的函数定义。**覆盖是静态 （根据成员函数的静态类型在编译时决定）还是动态 （通过对象指针在运行时动态决定），依赖于成员函数是否被声明为“虚函数”。**

Q 从 P 继承了成员变量和成员函数。Q 声明了 pf()，覆盖了 P::pf()。Q 还声明了 pvf()，覆盖了 P::pvf()虚函数。Q 还声明了新的非虚成员函数 qf()，以及新的虚成员函数 qvf()。



view plaincopy to clipboardprint?

```
1. struct Q : P {
2.         int q1;
3.         void pf(); // overrides P::pf
4.         void qf(); // new
5.         void pvf(); // overrides P::pvf
6.         virtual void qvf(); // new
7. };
```

对于非虚的成员函数来说，调用哪个成员函数是在编译时，根据“->”操作符左边指针表达式的类型静态决定的。特别地，即使 ppq 指向 Q 的实例，ppq->pf()仍然调用的是 P::pf()，因为 ppq 被声明为“P*”。（注意，“->”操作符左边的指针类型决定隐藏的 this 参数的类型。）

view plaincopy to clipboardprint?

```
1. P p; P* pp = &p; Q q; P* ppq = &q; Q* pq = &q;
2. pp->pf(); // pp->P::pf(); // P::pf(pp);
3. ppq->pf(); // ppq->P::pf(); // P::pf(ppq);
4. pq->pf(); // pq->Q::pf(); // Q::pf((P*)pq); （错误!）
5. pq->qf(); // pq->Q::qf(); // Q::qf(pq);
```

译者注：标记“错误”处，P*似应为 Q*。因为 pf 非虚函数，而 pq 的类型为 Q*，故应该调用到 Q 的 pf 函数上，从而该函数应该要求一个 Q* const 类型的 this 指针。

对于虚函数调用来说，调用哪个成员函数在运行时决定。不管“->”操作符左边的指针表达式的类型如何，调用的虚函数都是由指针实际指向的实例类型所决定。比如，尽管 ppq 的类型是 P*，当 ppq 指向 Q 的实例时，调用的仍然是 Q::pvf()。

view plaincopy to clipboardprint?

```
1. pp->pvf(); // pp->P::pvf(); // P::pvf(pp);
2. ppq->pvf(); // ppq->Q::pvf(); // Q::pvf((Q*)ppq);
3. pq->pvf(); // pq->Q::pvf(); // Q::pvf((P*)pq); （错误！）
```

译者注：标记“错误”处，P*似应为 Q*。因为 pvf 是虚函数，pq 本来就是 Q*，又指向 Q 的实例，从哪个方面来看都不应该是 P*。

为了实现这种机制，引入了隐藏的 vfptr 成员变量。 一个 vfptr 被加入到类中（如果类中没有的话），该 vfptr 指向类的虚函数表（vftable）。类中每个虚函数在该类的虚函数表中都占据一项。每项保存一个对于该类适用的虚函数的地址。因此，调用虚函数的过程如下：取得实例的 vfptr；通过 vfptr 得到虚函数表的一项；通过虚

函数表该项的函数地址间接调用虚函数。 也就是说，在普通函数调用的参数传递、调用、返回指令开销外，虚函数调用还需要额外的开销。

回头再看看 P 和 Q 的内存布局，可以发现，VC++编译器把隐藏的 vfptr 成员变量放在 P 和 Q 实例的开始处。这就使虚函数的调用能够尽量快一些。实际上，**VC++的实现方式是，保证任何有虚函数的类的第一项永远是 vfptr。** 这就可能要求在实例布局时，在基类前插入新的 vfptr，或者要求在多重继承时，虽然在右边，然而有 vfptr 的基类放到左边没有 vfptr 的基类的前面（如下）。

```
view plaincopy to clipboardprint?
```

```
1. class CA
2. {    int a;};
3. class CB
4. {    int b;};
5. class CL : public CB, public CA
6. {    int c;};
```

对于 CL 类，它的内存布局是：
int b;
int a;
int c;
但是，改造 CA 如下：

```
view plaincopy to clipboardprint?
```

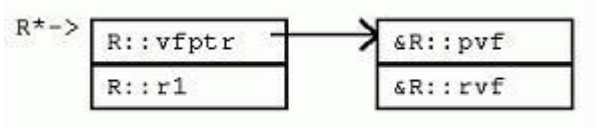
```
1. class CA
2. {
3.     int a;
4.     virtual void seta( int _a ) { a = _a; }
5. };
```

对于同样继承顺序的 CL，内存布局是：
vfptr;
int a;
int b;
int c;

许多 C++的实现会共享或者重用从基类继承来的 vfptr。比如，Q 并不会有一个额外的 vfptr，指向一个专门存放新的虚函数 qvf()的虚函数表。Qvf 项只是简单地**追加** 到 P 的虚函数表的末尾。如此一来，单继承的代价就不算高昂。一旦一个实例有 vfptr 了，它就不需要更多的 vfptr。新的派生类可以引入更多的虚函数，这些新的虚函数只是简单地在已存在的，“每类一个”的虚函数表的末尾追加新项。

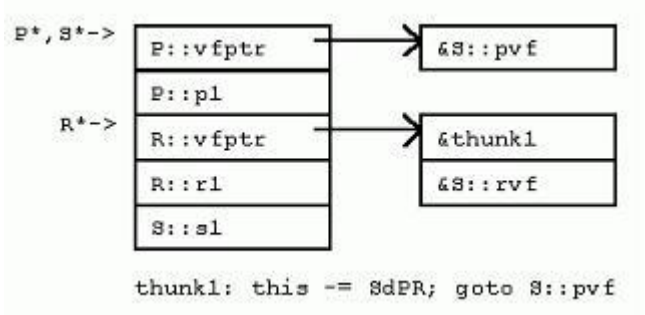
5.2 多重继承下的虚函数

如果从多个有虚函数的基类继承，一个实例就有可能包含多个 vfptr。考虑如下的 R 和 S 类：



view plaincopy to clipboardprint?

```
1.  struct  R  {
2.          int  r1;
3.          virtual  void  pvf();  //  new
4.          virtual  void  rvf();  //  new
5.  };
```



view plaincopy to clipboardprint?

```
1.  struct  S  :  P,  R  {
2.          int  s1;
3.          void  pvf();  //  overrides  P::pvf  and  R::pvf
4.          void  rvf();  //  overrides  R::rvf
5.          void  svf();  //  new
6.  };
```

这里 R 是另一个包含虚函数的类。因为 S 从 P 和 R 多重继承，S 的实例内嵌 P 和 R 的实例，以及 S 自身的数据成员 S::s1。注意，在多重继承下，靠右的基类 R，其实例的地址和 P 与 S 不同。 S::pvf 覆盖了 P::pvf()和 R::pvf()，S::rvf()覆盖了 R::rvf()。

view plaincopy to clipboardprint?

```
1.  S  s;  S*  ps  =  &s;
2.  ((P*)ps)->pvf();  //  (*(P*)ps)->P::vfptr[0])((S*)(P*)ps)
3.  ((R*)ps)->pvf();  //  (*(R*)ps)->R::vfptr[0])((S*)(R*)ps)
4.  ps->pvf();              //  one  of  the  above;  calls  S::pvf()
```

译者注：
调用((P*)ps)-•>pvf()时，先到 P 的虚函数表中取出第一项，然后把 ps 转化为 S*作为 this 指针传递进去；
调用((R*)ps)-•>pvf()时，先到 R 的虚函数表中取出第一项，然后把 ps 转化为 S*作为 this 指针传递进去；

因为 S::pvf()覆盖了 P::pvf()和 R::pvf()，在 S 的虚函数表中，相应的项也应该被覆盖。然而，我们很快注意到，不光可以用 P*，还可以用 R*来调用 pvf()。问题出现了：R 的地址与 P 和 S 的地址不同。表达式(R*)ps 与表达式(P*)ps 指向类布局中不同的位置。因为函数 S::pvf 希望获得一个 S*作为隐藏的 this 指针参数，虚函数必须把 R*转化为 S*。因此，在 S 对 R 虚函数表的拷贝中，pvf 函数对应的项，指向的是一个“调整块”的地址，该调整块使用必要的计算，把 R*转换为需要的 S*。
译者注：这就是“thunk1: this-= sdPR; goto S::pvf”干的事。先根据 P 和 R 在 S 中的偏移，调整 this 为 P*，也就是 S*，然后跳转到相应的虚函数处执行。

在微软 VC++实现中，对于有虚函数的多重继承，只有当派生类虚函数覆盖了多个基类的虚函数时，才使用调整块。

5.3 地址点与“逻辑 this 调整”

考虑下一个虚函数 S::rvf()，该函数覆盖了 R::rvf()。我们都知道 S::rvf()必须有一个隐藏的 S*类型的 this 参数。但是，因为也可以用 R*来调用 rvf()，也就是说，R 的 rvf 虚函数槽可能以如下方式被用到：

```
view plaincopy to clipboardprint?
```

```
1.  ((R*)ps)->rvf();  //  (*((R*)ps)->R::vfptr[1])((R*)ps)
```

所以，大多数实现用另一个调整块将传递给 rvf 的 R*转换为 S*。还有一些实现在 S 的虚函数表末尾添加一个特别的虚函数项，该虚函数项提供方法，从而可以直接调用 ps->rvf()，而不用先转换 R*。MSC++的实现不是这样，MSC++有意将 S::rvf 编译为接受一个指向 S 中嵌套的 R 实例，而非指向 S 实例的指针（我们称这种行为是“给派生类的指针类型与该虚函数第一次被引入时接受的指针类型相同”）。所有这些在后台透明发生，对成员变量的存取，成员函数的 this 指针，都进行“逻辑 this 调整”。

当然，在 debugger 中，必须对这种 this 调整进行补偿。

```
view plaincopy to clipboardprint?
```

```
1.  ps->rvf();  //  ((R*)ps)->rvf();  //  S::rvf((R*)ps)
```

译者注：调用 rvf 虚函数时，直接给入 R*作为 this 指针。

所以，当覆盖非最左边的基类的虚函数时，MSC++一般不创建调整块，也不增加额外的虚函数项。

5.4 调整块

正如已经描述的，有时需要调整块来调整 this 指针的值（this 指针通常位于栈上返回地址之下，或者在寄存器中），在 this 指针上加或减去一个常量偏移，再调用虚函数。某些实现（尤其是基于 cfront 的）并不使用调整块机制。它们在每个虚函数表项中增加额外的偏移数据。每当虚函数被调用时，该偏移数据（通常为 0），被加到对象的地址上，然后对象的地址再作为 this 指针传入。

```
view plaincopy to clipboardprint?
```

```
1.  ps->rvf();
2.  //  struct  {  void  (*pfn)(void*);  size_t  disp;  };
3.  //  (*ps->vfptr[i].pfn)(ps  +  ps->vfptr[i].disp);
```

译者注：当调用 rvf 虚函数时，前一句表示虚函数表每一项是一个结构，结构中包含偏移量；后一句表示调用第 i 个虚函数时，this 指针使用保存在虚函数表中第 i 项的偏移量来进行调整。

这种方法的缺点是虚函数表增大了，虚函数的调用也更加复杂。

现代基于 PC 的实现一般采用“调整—跳转”技术：

```
view plaincopy to clipboardprint?
```

```
1.  S::pvf-adjust:  //  MSC++
2.  this  -=  SdPR;
3.  goto  S::pvf()
```

当然，下面的代码序列更好（然而，当前没有任何实现采用该方法）：

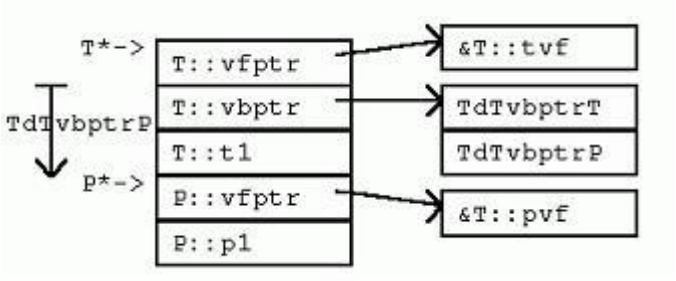
view plaincopy to clipboardprint?

```
1.  S::pvf-adjust:
2.  this  -=  SdPR;  //  fall  into  S::pvf()
3.  S::pvf()  {  ...  }
```

译者注：IBM 的 C++编译器使用该方法。

5.5 虚继承下的虚函数

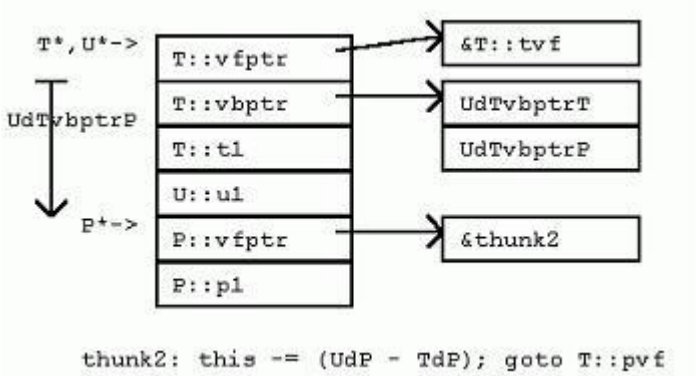
T 虚继承 P，覆盖 P 的虚成员函数，声明了新的虚函数。如果采用在基类虚函数表末尾添加新项的方式，则访问虚函数总要求访问虚基类。在 VC++中，为了避免获取虚函数表时，转换到虚基类 P 的高昂代价，**T 中的新虚函数通过一个新的虚函数表　获取**　，从而带来了一个新的虚函数表指针。该指针放在 T 实例的顶端。



view plaincopy to clipboardprint?

```
1.  struct  T  :  virtual  P  {
2.          int  t1;
3.          void  pvf();                    //  overrides  P::pvf
4.          virtual  void  tvf();  //  new
5.  };
6.  void  T::pvf()  {
7.          ++p1;  //  ((P*)this)->p1++;  //  vbtable  lookup!
8.          ++t1;  //  this->t1++;
9.  }
```

如上所示，**即使是在虚函数中，访问虚基类的成员变量也要通过获取虚基类表的偏移，实行计算来进行。这样做之所以必要，是因为虚函数可能被进一步继承的类所覆盖，而进一步继承的类的布局中，虚基类的位置变化了。**　下面就是这样的一个类：



view plaincopy to clipboardprint?

```
1.  struct  U  :  T  {
2.          int  u1;
3.  };
```

在此 U 增加了一个成员变量，从而改变了 P 的偏移。因为 VC++实现中，T::pvf()接受的是嵌套在 T 中的 P 的指针，所以，需要提供一个调整块，把 this 指针调整到 T::t1 之后（该处即是 P 在 T 中的位置）。

5.6 特殊成员函数

本节讨论编译器合成到特殊成员函数中的隐藏代码。

5.6.1 构造函数和析构函数

正如我们所见，在构造和析构过程中，有时需要初始化一些隐藏的成员变量。最坏的情况下，一个构造函数要执行如下操作：

1 * 如果是“最终派生类”，初始化 vbptr 成员变量，调用虚基类的构造函数；
2 * 调用非虚基类的构造函数
3 * 调用成员变量的构造函数
4 * 初始化虚函数表成员变量
5 * 执行构造函数体中，程序所定义的其他初始化代码

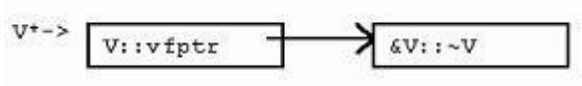
（注意：一个“最终派生类”的实例，一定不是嵌套在其他派生类实例中的基类实例）
所以，如果你有一个包含虚函数的很深的继承层次，即使该继承层次由单继承构成，对象的构造可能也需要很多针对虚函数表的初始化。
反之，析构函数必须按照与构造时严格相反的顺序来“肢解”一个对象。

1 * 合成并初始化虚函数表成员变量
2 * 执行析构函数体中，程序定义的其他析构代码
3 * 调用成员变量的析构函数（按照相反的顺序）
4 * 调用直接非虚基类的析构函数（按照相反的顺序）
5 * 如果是“最终派生类”，调用虚基类的析构函数（按照相反顺序）

在 VC++中，有虚基类的类的构造函数接受一个隐藏的“最终派生类标志”，标示虚基类是否需要初始化。对于析构函数，VC++采用“分层析构模型”，代码中加入一个隐藏的析构函数，该函数被用于析构包含虚基类的类（对于“最终派生类”实例而言）；代码中再加入另一个析构函数，析构不包含虚基类的类。前一个析构函数调用后一个。

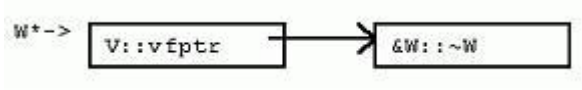
5.6.2 虚析构函数与 delete 操作符

假如 A 是 B 的父类，
A* p = new B();
如果析构函数不是虚拟的，那么，你后面就必须这样才能安全的删除这个指针：
delete (B*)p;
但如果构造函数是虚拟的，就可以在运行时动态绑定到 B 类的析构函数，直接：
delete p;
就可以了。这就是虚析构函数的作用。
实际上，很多人这样总结：当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。
考虑结构 V 和 W。



view plaincopy to clipboardprint?

```
1. struct V {
2.     virtual ~V();
3. };
```



view plaincopy to clipboardprint?

```
1. struct W : V {
2.     operator delete();
3. };
```

析构函数可以为虚。**一个类如果有虚析构函数的话，将会象有其他虚函数一样，拥有一个虚函数表指针，虚函数表中包含一项，其内容为指向对该类适用的虚析构函数的地址。这些机制和普通虚函数相同。虚析构函数的特别之处在于：当类实例被销毁时，虚析构函数被隐含地调用。调用地（delete 发生的地方）虽然不知道销毁的动态类型，然而，要保证调用对该类型合适的 delete 操作符。**例如，当 pv 指向 W 的实例时，当 W::~W 被调用之后，W 实例将由 W 类的 delete 操作符来销毁。

view plaincopy to clipboardprint?

```
1. V* pv = new V;
2. delete pv;      // pv->~V::V(); // use ::operator delete()
3. pv = new W;
4. delete pv;      // pv->~W::W(); // use W::operator delete() 动态绑定到 W 的析构函数，W 默认的析构函数调用{delete this;}
5. pv = new W;
6. ::delete pv; // pv->~W::W(); // use ::operator delete()
```

译者注：

V 没有定义 delete 操作符，delete 时使用函数库的 delete 操作符；•

W 定义了 delete 操作符，delete 时使用自己的 delete 操作符；•

可以用全局范围标示符显示地调用函数库的 delete 操作符。•

为了实现上述语意，VC++扩展了其“分层析构模型”，从而自动创建另一个隐藏的析构帮助函数——“deleting 析构函数”，然后，用该函数的地址来替换虚函数表中“实际”虚析构函数的地址。析构帮助函数调用对该类合适的析构函数，然后为该类有选择性地调用合适的 delete 操作符。

6 数组

堆上分配空间的数组使虚析构函数进一步复杂化。问题变复杂的原因有两个：

1、 堆上分配空间的数组，由于数组可大可小，所以，数组大小值应该和数组一起保存。因此，堆上分配空间的数组会分配额外的空间来存储数组元素的个数；

2、 当数组被删除时，数组中每个元素都要被正确地释放，即使当数组大小不确定时也必须成功完成该操作。然而，派生类可能比基类占用更多的内存空间，从而使正确释放比较困难。

view plaincopy to clipboardprint?

```
1. struct WW : W { int w1; };
2. pv = new W[m];
3. delete [] pv; // delete m W's (sizeof(W) == sizeof(V))
4. pv = new WW[n];
5. delete [] pv; // delete n WW's (sizeof(WW) > sizeof(V))
```

译者注：WW 从 W 继承，增加了一个成员变量，因此，WW 占用的内存空间比 W 大。然而，不管指针 pv 指向 W 的数组还是 WW 的数组，delete[]都必须正确地释放 WW 或 W 对象占用的内存空间。
虽然从严格意义上来说，数组 delete 的多态行为 C++标准并未定义，然而，微软有一些客户要求实现该行为。因此，在 MSC++中，该行为是用另一个编译器生成的虚析构帮助函数来完成。该函数被称为“向量 delete 析构函数”（因其针对特定的类定制，比如 WW，所以，它能够遍历数组的每个元素，调用对每个元素适用的析构函数）。

7 异常处理

简单说来，异常处理是 C++标准委员会工作文件提供的一种机制，通过该机制，一个函数可以通知其调用者“异常”情况的发生，调用者则能据此选择合适的代码来处理异常。该机制在传统的“函数调用返回，检查错误状态代码”方法之外，给程序提供了另一种处理错误的手段。

因为 C++是面向对象的语言，很自然地，C++中用对象来表达异常状态。并且，使用何种异常处理也是基于“抛出的”异常对象的静态或动态类型来决定的。不光如此，既然 C++总是保证超出范围的对象能够被正确地销毁，异常实现也必须保证当控制从异常抛出点转换到异常“捕获”点时（栈展开），超出范围的对象能够被自动、正确地销毁。
考虑如下例子：

view plaincopy to clipboardprint?

```
1. struct X { X(); }; // exception object class
2. struct Z { Z(); ~Z(); }; // class with a destructor
3. extern void recover(const X&);
4. void f(int), g(int);
5.
6. int main() {
7.     try {
8.         f(0);
9.     } catch (const X& rx) {
10.        recover(rx);
11.    }
12.    return 0;
13. }
14.
15. void f(int i) {
16.    Z z1;
17.    g(i);
18.    Z z2;
```

```
19.         g(i-1);
20. }
21.
22. void  g(int  j)  {
23.         if  (j  <  0)
24.               throw  X();
25. }
```

译者注：X 是异常类，Z 是带析构函数的工作类，recover 是错误处理函数，f 和 g 一起产生异常条件，g 实际抛出异常。

这段程序会抛出异常。在 main 中，加入了处理异常的 try & catch 框架，当调用 f(0)时，f 构造 z1，调用 g(0)后，再构造 z2，再调用 g(-1)，此时 g 发现参数为负，抛出 X 异常对象。我们希望在某个调用层次上，该异常能够得到处理。既然 g 和 f 都没有建立处理异常的框架，我们就只能希望 main 函数建立的异常处理框架能够处理 X 异常对象。实际上，确实如此。当控制被转移到 main 中异常捕获点时，从 g 中的异常抛出点到 main 中的异常捕获点之间，该范围内的对象都必须被销毁。在本例中，z2 和 z1 应该被销毁。

谈到异常处理的具体实现方式，一般情况下，在抛出点和捕获点都使用“表”来表述能够捕获异常对象的类型；并且，实现要保证能够在特定的捕获点真正捕获特定的异常对象；一般地，还要运用抛出的对象来初始化捕获语句的“实参”。通过合理地选择编码方案，可以保证这些表格不会占用过多的内存空间。

异常处理的开销到底如何？让我们再考虑一下函数 f。看起来 f 没有做异常处理。f 确实没有包含 try，catch，或者是 throw 关键字，因此，我们会猜异常处理应该对 f 没有什么影响。错！编译器必须保证一旦 z1 被构造，而后续调用的任何函数向 f 抛回了异常，异常又出了 f 的范围时，z1 对象能被正确地销毁。同样，一旦 z2 被构造，编译器也必须保证后续抛出异常时，能够正确地销毁 z2 和 z1。

要实现这些“展开”语意，编译器必须在后台提供一种机制，该机制在调用者函数中，针对调用的函数抛出的异常动态决定异常环境（处理点）。这可能包括在每个函数的准备工作和善后工作中增加额外的代码，在最糟糕的情况下，要针对每一套对象初始化的情况更新状态变量。例如，上述例子中，z1 应被销毁的异常环境当然与 z2 和 z1 都应该被销毁的异常环境不同，因此，不管是在构造 z1 后，还是继而在构造 z2 后，VC++都要分别在状态变量中更新（存储）新的值。

所有这些表，函数调用的准备和善后工作，状态变量的更新，都会使异常处理功能造成可观的内存空间和运行速度开销。正如我们所见，即使在没有使用异常处理的函数中，该开销也会发生。

幸运的是，一些编译器可以提供编译选项，关闭异常处理机制。那些不需要异常处理机制的代码，就可以避免这些额外的开销了。

8 小结

好了，现在你可以写 C++编译器了（开个玩笑）。

在本文中，我们讨论了许多重要的 C++运行实现问题。我们发现，很多美妙的 C++语言特性的开销很低，同时，其他一些美妙的特性（译者注：主要是和“虚”字相关的东西）将造成较大的开销。C++很多实现机制都是在后台默默地为你工作。一般说来，单独看一段代码时，很难衡量这段代码造成的运行时开销，必须把这段代码放到一个更大的环境中来考察，运行时开销问题才能得到比较明确的答案。