C++之成员函数调用

c++支持三种类型的成员函数，分别为 static,nostatic,virtual。每一种调用方式都不尽相同。

非静态成员函数（Nonstatic Member Functions）

保证 nostatic member function 至少必须和一般的 nonmember function 有相同的效率是 C++的设计准则之一。事实上在 c++中非静态成员函数（nostatic member function）与普通函数的调用也确实具有相同的效率，因为本质上非静态成员函数就如同一个普通函数,如一个非静态成员函数 X`float Point::X();`就相当于一个普通函数 `float X(Point* this);`。编译器内部会将成员函数等价转换为非成员函数，具体是这样做的:

1.改写成员函数的签名，使得其可以接受一个额外参数，这个额外参数即是 this 指针：

```
float Point::X();//成员函数 X 被插入额外参数 thisfloat Point:: X(Point* this );
```

当然如果成员函数是 const 的，插入的参数类型将为 const Point* 类型。

2.将每一个对非静态数据成员的操作都改写为经过 this 操作。

3.将成员函数写成一个外部函数，对函数名进行“mangling”处理，使之成为独一无二的名称。

可以看出，将一个成员函数改写成一个外部函数的关键在于两点，一是给函数提供一个可以直接读写成员数据的通道； 二是解决好有可能带来的名字冲突。第一点通过给函数提供一个额外的指针参数来解决，第二点则是通过一定的规则将名字转换，使之独一无二。

由此可以做出一点总结:一个成员函数实际上就是一个被插入了一个接受其类的指针类型的额外参数的非成员函数， 当然还要额外对函数的名称进行处理。额外插入的参数用来访问数据成员，而名称的特殊处理用来避免名字冲突。

对于名称的特殊处理并没有统一的标准，各大编译器厂商可能有不同的处理规则。 在 VC 下上述的成员函数 X()的名称 X 处理后就成了`?X@Point@@QAEMXZ` 更多信息可以参见维基百科的 Visual C++名字修饰。

于是在 VC 中对于上面的例子中的成员函数的调用将发生如下的转换：

```
//p->X();被转化为?X@Point@@QAEMXZ(p);//obj.X();被转化为?X@Point@@QAEMXZ(&obj);
```

虚拟成员函数(Virtual Member Functions)

如果 function()是一个虚拟函数，那么用指针或引用进行的调用将发生一点特别的转换——一个中间层被引入进来。例如：

```
// p->function()//将转化为(*p->vptr[1])(p);
```

- 其中 vptr 为指向虚函数表的指针，它由编译器产生。vptr 也要进行名字处理，因为一个继承体系可能有多个 vptr。
- 1 是虚函数在虚函数表中的索引，通过它关联到虚函数 function().

何时发生这种转换？答案是在必需的时候——一个再熟悉不过的答案。当通过指针调用的时候，要调用的函数实体无法在编译期决定，必需待到执行期才能获得，所以上面引入一个间接层的转换必不可少。但是当我们通过对象（不是引用，也不是指针）来调用的时候， 进行上面的转换就显得多余了，因为在编译器要调用的函数实体已经被决定。此时调用发生的转换，与一个非静态成员函数(Nonstatic Member Functions)调用发生的转换一致。

静态成员函数(Static Member Functions)

静态成员函数的一些特性：

1. 不能够直接存取其类中的非静态成员（nostatic members），包括不能调用非静态成员函数(Nonstatic Member Functions)。
2. 不能够声明为 const、voliatile 或 virtual。
3. 它不需经由对象调用，当然，通过对象调用也被允许。

除了缺乏一个 this 指针他与非静态成员函数没有太大的差别。在这里通过对象调用和通过指针或引用调用，将被转化为同样的调用代码。

需要注意的是通过一个表达式或函数对静态成员函数进行调用，被 C++ Standard 要求对表达式进行求值。如：

```
(a+=b).static_fuc();
```

虽然省去对 a+b 求值对于 static_fuc()的调用并没有影响，但是程序员肯定会认为表达式 a+=b 已经执行，一旦编译器为了效率省去了这一步，很难说会浪费多少程序员多少时间。这无疑是一个明智的规定。