C++对象的数据成员

数据成员的布局

对于一个类来说它的对象中只存放非静态的数据成员,但是除此之外，编译器为了实现 virtual 功能还会合成一些其它成员插入到对象中。我们来看看这些成员的布局。

C++ 标准的规定

- 在同一个 Access Section（也就是 private,public,protected 片段）中， 要求较晚出现的数据成员处在较大的内存中。这意味着同一个片段中的数据成员并不需要紧密相连，编译器所做的成员对齐就是一个例子。
- 允许编译器将多个 Acess Section 的顺序自由排列，而不必在乎它们的声明 次序。但似乎没有编译器这样做。
- 对于继承类，C++标准并未指定是其基类成员在前还是自己的成员在前。
- 对于虚基类成员也是同样的未予规定。

一般的编译器怎么做？

- 同一个 Access Section 中的数据成员按期声明顺序，依次排列。 但成员与成员之间因为内存对齐的原因可能存在空当。
- 多个 Access Section 按其声明顺序排放。
- 基类的数据成员总放在自己的数据成员之前，但虚基类除外。

编译器合成的成员放在哪？

为了实现虚函数和虚拟继承两个功能，编译器一般会合成 Vptr 和 Vbptr 两个指针。那么这两个指针应该放在什么位置？C++标准肯定是不曾规定的，因为它甚至并没有规定如何来实现这两个功能，因此就语言层面来看是不存在这两个指针的。

对于 Vptr 来说有的编译器将它放在末尾，如 Lippman 领导开发的 Cfront。有的则将其放在最前面，如 MS 的 VC，但似乎没人将它放在中间。为什么不放在中间？没有理由可以让人这么做，放在末尾，可以保持 C++类对 C 的 struct 的良好兼容性，放在最前可以给多重继承下的指针或引用调用虚函数带来好处。

看一小段代码：

```
class X{public:
    int a;
```

```
    virtual void vfc(){};};int main(){
    using namespace std;
    X x;
    cout<<&x.a<<" "<<&x<<endl;
    system("pause");}
```

在 VS2010 和 VC6.0 中运行的结果都是地址值&x.a 比&x 大 4，可见说 vc 的 vptr 放在对象的最前面此言非虚。

对于 Vbptr 来说，有好几种方法，在这儿我们只看看 VC 的实现原理：

对于由虚拟继承而得的类，VC 会在其每一个对象中插入一个 Vbptr,这个 Vbptr 指向 vitual base class table（我称之为虚基类表）。虚基类表中则存放有其虚基类子对象相对于虚基类指针的偏移量。例如声明如 `class Y:virtual public X` 的类的 virtual base class table 的虚基类表中当存储有 X 对象相对于 Vbptr 的偏移量。

对象成员或基类对象成员后面的填充空白不能为其它成员所用

看一段代码：

```
class X{public:
    int x;
    char c;};class X2:public X{public:char c2;};
```

X2 的布局应当是 x(4),c(1),c2(1),这么说来 sizeof(X2)的值应该是 8？错了，实际上是 12。原因在于 X 后面的三个字节的填充空白不能为 c2 所用。也就是说 X2 的大小实际上为：X(8)+c2(1)+填补（3）=12。这样看来编译器似乎是那么的呆板，其实不然，看一下下面的语句会发生什么？

```
X2 x2;X x;x2=x;
```

如果 X 后面的填充空白可以被 c2 使用的话，那么 X2 和 X 都将是 8 字节。上面的语句执行后 x2.c2 的值会是多少？一个不确定的值！这样的结果肯定不是我们想要的。

Vptr 与 Vbptr[1]

- 在多继承情况下，即使是多虚拟继承，继承而得的类只需维护一个 Vbptr； 而多继承情况下 Vptr 则可能有要维护多个 Vptr，视其基类有几个有虚函数。

- 一条继承线路只有一个 Vptr，但可能有多个 Vbptr，视有几次虚拟 继承而定。换言之，对于一个继承类对象来说，不需要新合成 vptr，而是使用其基类子对象的 vptr。而对于一个虚拟继承类来说，必须新合成一个自己的 Vbptr。

如：

```
class X{
    virtual void vf(){};};class X2:virtual public X{
  virtual void vf(){};};class X3:virtual public  X2{
    virtual void vf(){};}
```

X3 将包含有一个 Vptr，两个 Vbptr。确切的说这两个 Vbptr 一个属于 X3，一个属于 X3 的子对象 X2,X3 通过其 Vbptr 找到子对象 X2，而 X2 通过其 Vbptr 找到 X。

其中差别在于 vptr 通过一个虚函数表可以确切地知道要调用的函数，而 Vbptr 通过虚基类表只能够知道其虚基类子对象的偏移量。这两条规则是由虚函数与虚拟继承的实现方式，以及受它们的存取方式和复制控制的要求决定的。

数据成员的存取

静态数据成员相当于一个仅对该类可见的全局变量，因为程序中只存在一个静态数据成员的实例，所以其地址在编译时就已经被决定。不论如何静态数据成员的存取不会带来任何额外负担。

非静态数据成员的存取，相当于对象起始地址加上偏移量。效率上与 C struct 成员的效率等同。因为它的偏移量在编译阶段已经确定。但有一种情况例外：`pt->x=0.0`。当通过指针或引用来存取——x 而 x 又是虚基类的成员的时候。因为必须要等到执行期才能知道 pt 指向的确切类型，所以必须通过一个间接导引才能完成。

小结

在 VC 中数据成员的布局顺序为：

1. vptr 部分（如果基类有，则继承基类的）
2. vbptr （如果需要）
3. 基类成员（按声明顺序）
4. 自身数据成员
5. 虚基类数据成员（按声明顺序）

参考：《深度探索 C++对象模型》

这部分内容只是自己试验而得，并非放诸各编译器皆适合的准则。