深度探索C++对象模型读书读书笔记

1 Perface

1.1 What Is the C++ Object Model?

包含两个层次的概念，一个是语言中用以支持面向对象编程的直接支持，这可以从关于 c++语言的书籍比如 c++ primer 上获得。另一种则是实现这种支持的底层实现细节，这很少有书涉及，而本书就是要关注这些细节。

对于实现的细节，语言并没有给出明确的规定，但可能存在一些通用的作法，比如为了实现虚函数调用，需要建立一个 virtual table，这样实现显得很自然。但是一个特别的实现完全可以选择其他的方式。

虽然没有一个确定的实现标准，但是理解通常的实现仍有意义。可以增加程序员的信心，对于如何编写程序，已经这样产生的效率损失提供清晰的认识。很多对于 c++的误解就是来源于对其底层实现模型的不清楚造成的，人们总是笼统的认为 c++编译器默默地在背后做了太多的东西。通过这本书，希望可以去除围绕在 c++周围的迷雾。

随着 c++的发展，一些底层机制也发生了变化。比如 static 初始化【实现模型】。
定义一个全局对象

```
X buf;
main()
{
   // buf must be constructed at this point
   cin >> setw( 1024 ) >> buf;    ...
}
```

语言的第一个层次保证 buf 在 main 函数执行前这个对象完成初始化。而到底第二层次如何实现的呢？

munch solutions：早期的 cfont，假设没有明确的目标平台和环境支持，使用 nm 命令实现。使用 cc 生成一个可执行文件，然后对这个文件运行 nm 命令，生成新的.c 文件，编译这个文件，然后重新链接出一个新的可执行文件。
patch solutions：另一种针对特定平台的解决方案是，采用 coff 文件格式，直接检查可执行文件，不需要使用 nm，cc，relink。

这两种方案都是基于程序层面的，也就是说，cfont 对每个需要静态初始化的.c 文件，都产生出一个 sti_函数，在里面完成初始化操作，这两个都需要搜索那些以 sti 开头的函数，然后将它们执行。(通过一个安插在 main()里第一行的库函数_main 以一种未定义的顺序执行)。

但是随着 c++的不断发展，出现了一些支持初始化段的格式，可以直接将 main 之前完成的工作放到这里。这就是上升到了环境层次上的支持。所以 cfont 只是提供了一种实现模型，由于它是第一个 c++编译器，姑且称为传统模型，而其他的实现则可能采用了不同的实现模型，以示区别。

2 Object Lessons

2.1 对象模型：

三种可供参考的实现模型，体现了对象上如何在内存中布置的。

2.1.1 简单对象类型：

所有的成员占用相同的空间，对象只是维护了一个包含成员指针的一个表。表中放的是成员的地址，无论上成员变量还是函数，都是这样处理。对象并没有直接保存成员而是保存了成员的指针。这个模型虽然简单，但是 this simple concept of an index or slot number is the one that has been developed into the C++ pointer-to-member concept。

2.1.2 表驱动对象模型：

这个模型可以保证所有的对象具有相同的大小，比如简单对象模型还与成员的个数相关。但是这个模型在简单对象的基础上又添加了一个间接层。将成员分成函数 member，和 data member，并且用两个表格保存，然后对象只保存了两个指向表格的指针。虽然该对象没有被实际应用，当思想仍然得到了采纳：the concept of a member function table has been the traditional implementation supporting efficient runtime resolution of virtual functions.被应用到了 vptr 和 vtbl 中。

2.1.3 C++对象模型：

真正的 c++采用的实现模型【实现模型】。这个模型从简单对象派生而来，并对内存存取和空间进行了优化。在此模型中，non static data member 被放置到对象内部，Static data members， Static and nonstatic function members 均被放到对象之外。对于虚函数的支持则分两步完成：
1.为每个类生成了一个虚函数表。
2.指向相关虚表的指针被插入到了每个对象中。
这个模型的优点在于访问和空间效率的提高，缺点如下：如果应用程序本身未改变，但当所使用的类的 non static 数据成员添加删除修改的时候，需要重新编译。

2.2 关键词的差异：

与 c 的兼容性 。

Struct 与 class 之争：

首先一个不同，struct 默认 public，其他基本与 class 一致。
另外在内存布局上可能有些不同。C++保证同一个 access section 内的变量按照声明次序进行内存的布局，但不保证不同 section 之间的布局。同样子类与父类的成员布局也没有明确的规定。而采用 struct 可以保证 c 风格的布局要求。只有在把 struct 作为一个成员时才能保证这样的与 c 兼容的布局，但当以一个 struct 作为基类时，无法保证子类会不会向基类的成员布局里安插新的数据成员，从而改变已有的布局。

2.3 对象的差异：

C++支持几种编程模式，The procedural model The abstract data type (ADT) model The object-oriented (OO) model
比如OO中，注意要实现多态，必须使用对象的指针或者引用。是一种运行时的行为，而对于对象实体则在编译期就已经确定了。

那么需要多大的内存来存下一个object呢？
将由以下部分组成：
所有的non static 数据member；由于内存对齐要求产生的填充；由于虚函数产生的开销，比如vptr；另外可能需要一个指向虚基类的开销。那么成员函数的指向又是如何确定的呢？

指针类型的不同在于其指向的内容的类型和大小不同。

当我们以一个父类指针指向子类的对象的时候究竟发生了什么变化？而与指向该子类对象的子类指针的不同究竟在哪类呢？对于多个基类的情况，vptr 是否在同一个位置呢，否则如何保证正确的找到这个 vptr？

不同类型的指针的不同在于指针的 span(跨度)不同，比如我们用指向子类的对象的父类指针便无法访问子类里的 public 成员变量或者函数(这体现了对指针类型的深刻理解)。也就是说即使具有多态的支持，父类的指针也只能对那些在该类出现的名字进行访问，无法访问子类特有的。通过多态，只是实现了同名函数不同实体的调用。对于父类指针所进行的调用编译时刻只会确定如下两项：固定的可用接口，接口的访问级别。而子类的指针则可以访问从父类继承下来的那些成员，但要注意可能发生的遮蔽(见下页)。

再看这样一个例子：

```
Bear b;
ZooAnimal za = b;
// ZooAnimal::rotate() invoked
za.rotate();
```

如果按照 memberwise 拷贝，为何 b 的 vptr 没有被拷贝给 za 呢？(包含了对于虚函数对象拷贝的深刻理解)

The answer to the question is that the compiler intercedes in the initialization and assignment of one class object with another. The compiler must ensure that if an object contains one or more vptrs, those vptr values are not initialized or changed by the source object .

可见这也是不能为含有虚函数的类采用 bit wise 的方式拷贝和构造的原因。需要编译器生成，以满足上面这种保证。

另外我们能否改变从父类继承的虚函数的访问级别吗？(实际上体现了对于访问控制的深刻理解)

实际测试发现可以，即可以在父类中定义的 public 虚函数，在子类中声明为 private。由此可见 public 和 private 只是作为编译时刻的一种坚持，而不会引起保存方式上的变化。因为实际上我们使用父类的指针访问，并没有涉及子类的名字。所以可以访问到子类中的那个 private 的 virtual 接口。

另外对于子类和父类如果拥有同名的 data member 或者 non virtual 成员函数，又会如何处理呢？如果 data member 是 static 类型的又有何不同？(实际上体现了对继承的深刻理解)

对于这样的同名函数或者成员，实际上由指针或者对象的类型决定了访问的具体是哪个？如果使用子类类型的对象或指针，则只能看到子类里的哪个函数或者成员，对于父类里的那个则会产生**遮蔽现象**，如果想访问父类成员，必须加上父类的名称限定，【**实现模型**】也就是说实际上编译器在实现中自动地对这种重名现象进行了处理，给父类对象加上了一个限定词合成了一个新的名称。但是如果没有发生这种重名现象，子类类型的对象或指针，依然可以正常访问从父类继承下来的那些成员。

3 构造函数语义学

人们经常抱怨编译器总是默默在背后做了很多工作。以一个cin的转换函数为例，说明了这样的默默的行为如何导致了一种称为Schwarz Error的错误。为了让cin支持这样的使用方式，if (cin) .... 【实现模型】Schwarz采用了给cin添加一个转换运算符operator int()，但是这样的一个运算符，却导致了使一种cin的错误用法变的合法，但是意思却并非我们所需要的。比如这样的行为 cin << intVal;将由转换操作解析为一个int的移位操作。解决方法就是使用一个 operator void*()代替operator int()

那为何还有要这种隐式的类型转换呢？可能最初的动力来自与string类，如果没有这样的一个转换，针对string将不得不重写大量的以字符串为参数的c运行式库，才能进行调用。.

而关键词expicit的加入，正是为了防止application of a single argument constructor as a conversion operator.从一定程度上限制了这种默认转换。但是这种默认转换实际上只是按照程序的字面意思进行的。而背后活动更可能发生在成员初始化以及具名返回值优化(named return value optimization (NRV))的过程中。

3.1 Default Constructor Construction

The C++ Annotated Reference Manual (ARM) [ELLIS90] (Section 12.1) tells us that "default constructors…are generated (by the compiler) where needed…."
这里的需要是指编译器的需要，而非程序员的需要。

The Standard has refined the discussion in the ARM, although the behavior, in practice, remains the same. The Standard states [ISO-C++95] (also Section 12.1) the following:

If there is no user-declared constructor for class X, a default constructor is implicitly declared…. A constructor is trivial if it is an implicitly declared default constructor….

有四种情况生成的 default constructor is nontrivial. 【深刻理解】实际上这四种情况，都可以归纳为一种情况，就是这些对象，存在一些必须要进行的行为，而这些行为 **trivial** 的构造器的并不会进行，有必要由编译器生成一个构造器去完成这些行为，比如成员的构造函数的调用，基类的构造器的调用，虚函数表指针 **vptr** 的设置，虚基类指针 **vbc** 的设置。

3.1.1 Member Class Object with Default Constructor

If a class without any constructors contains a member object of a class with a

default constructor, the implicit default constructor of the class is nontrivial and the compiler needs to synthesize a default constructor for the containing class. This synthesis, however, takes place only if the constructor actually needs to be invoked.
可见有默认构造器的生成有三个条件，该类本身没有任何构造器，另外成员对象有一个默认构造器，该构造器真正被调用了。如果该类以及含有构造器，或者成员对象有一些含参数的构造器，这个默认构造器就无法生成出来。

还有一个问题，如何避免为多个.c 文件合成出多份构造器呢？**【实现模型】，解决方法就是把生成的函数作为 inline，inline 函数本身具有 static**linkage，如果 inline 太复杂就生成一个 an explicit non-inline static instance。

即便是用户已经定义了构造函数，这个函数仍然可能被扩展。
Consider the case of each constructor defined for a class containing one or more member class objects for which a default constructor must be invoked. In this case, the compiler augments the existing constructors, inserting code that invokes the necessary default constructors prior to the execution of the user code.
如果有多个 class member objects 需要被初始化，又会如何呢？
The language requires that the constructors be invoked in the order of member declaration within the class. This is accomplished by the compiler. It inserts code within each constructor, invoking the associated default constructors for each member in the order of member declaration. This code is inserted just prior to the explicitly supplied user code.

还有一个问题需要考虑，这个定义的顺序与 constructors explicitly listed 的关系，因为这样的调用关系要保证合理的顺序，是否保证所有的可以按照声明顺序来调用呢？这个在 3.4 部分讨论。

3.1.2 Base Class with Default Constructor

如果一个类没有任何构造器，且继承的父类含有一个默认构造器。则生成一个默认构造器，在里面调用父类的默认构造器。

Similarly, if a class without any constructors is derived from a base class containing a default constructor, the default constructor for the derived class is considered nontrivial and so needs to be synthesized. The synthesized default constructor of the derived class invokes the default constructor of each of its immediate base classes in the order of their declaration. To a subsequently derived class, the synthesized constructor appears no different than that of an explicitly provided default constructor.

如果设计者提供了 multiple constructors，但是没有默认构造器，不会生成默认构造器，而是为所有的现有构造器自动插入对父类默认构造器的调用。

What if the designer provides multiple constructors but no default constructor? The compiler augments each constructor with the code necessary to invoke all required default constructors. However, it does not synthesize a default constructor because of the presence of the other user-supplied constructors. If member class objects with default constructors are also present, these default constructors are also invoked after the invocation of all base class constructors.

3.1.3 Class with a Virtual Function

包括两种情况：
1. 声明或者继承了一个虚函数
2. 派生自一个继承体系，其中有一个或多个虚基类

如果没有任何声明的构造器，则会合成一个默认的构造器。完成两项工作：
1. 产生虚函数表 vtbl，内放 class 的虚函数成员
2. 在每个对象里产生一个指针成员 vptr，指向虚函数表

另外所有的多态调用，会被改写比如
const Widget& widget; widget.flip();
会变成 widget.flip()　　(* widget.vptr[1]) (&widget)

为了完成这样的一个机制，编译器还需要完成一项工作，即为生成的对象安放正确的 vptr。**【实现模型】对于所有声明的构造器，编译器插入一段代码完成这些工作。而对于没有声明构造器的类，编译器会为它合成一个构造器，保证完成这项工作。这也是为何不能进行 trival 初始化的原因。**

3.1.4 Class with a Virtual Base Class

对于从虚基类继承来的子类来说，如何访问虚基类的成员呢？这个行为是无法在编译时期确定的，因为多态的原因，这个指针指向的对象类型并不确定，所以也就无法确定虚基类成员在对象中的具体偏移位置。所以需要提过运行时的支持。
【实现模型】在原始的 cfont 实现中，通过向每一个子类对象中插入一个指向虚基类的指针_vbc 来完成的，毫无疑问，这个指针也是在构造器中完成设置的。
All reference and pointer access of a virtual base class is achieved through the associated pointer.这样一个对于虚基类成员 i 的修改 void foo(const A* pa) { pa->i = 1024; }
，可能被编译器改写成：void foo(const A* pa) { pa->__vbcX->i = 1024; }

总结：
Classes that do not exhibit these characteristics and that declare no constructor at all are said to have implicit, trivial default constructors. In practice, these trivial default

constructors are not synthesized.
也就是说实际上所谓的 trivial default constructors，在实现中可能根本就没有合成出来。而在合成出来的构造函数中，只有 base class subobjects and member class objects 被初始化了，而那些 nonstatic data members, such as integers, pointers to integers, arrays of integers, and so on, are not initialized.它们的初始化依赖于程序员而非编译器实现。

实际上还有更复杂的情况，比如多个虚基类，多重继承，及其混合使用。

3.2 Copy Constructor Construction

使用它的情况可能有三种：
赋值初始化式：X xx = x;函数参数传递；返回值。
实际上这个与 default construction 很类似。【注意】但有些区别，比如生成的这个默认 copy，对内置类型也会进行拷贝，而 default construction 却不对内置内型进行初始化。还有一个问题需要考虑，如果用户定义了 copy 构造器，但其中有些内置类型变量未被包括，编译器是否会自动添加呢？答案是不会。但如果是一些用户定义类型的变量又如何呢？答案可参考 effective c++读书笔记 2.8 复制对象时勿忘其每一个成分。

用户可以自定义，如果没有自定义的，当用到它时，就要依赖编译器生成一个。
What if the class does not provide an explicit copy constructor? Each class object initialized with another object of its class is initialized by what is called default memberwise initialization. Default memberwise initialization copies the value of each built-in or derived data member (such as a pointer or an array) from the one class object to another. A member class object, however, is not copied; rather, memberwise initialization is recursively applied.

从概念上来说，对象的拷贝是通过拷贝构造器完成的。在实际中，一个好的编译器可能针对含有 bit-wise copy 语义的对象进行 bit-wise copy。【注意】也就是说实际中，对于一个不含有拷贝构造函数的类，它们的拷贝构造函数也不一定必须生成出来，只有需要时才会生成，也就是不满足 bit-wise copy 语义的情形下。

是否体现出 bit-wise copy 语义，主要取决于成员的类型，如果数据成员均是简单内置类型一般可以理解成 bit-wise copy 语义。当然也有例外，比如涉及到 virtual。

违反 bit-wise copy 语义的情况，主要有以下四种，实际上就是上面 3.1 中的四种情况。

3.2.1 成员对象含有拷贝构造函数

When the class contains a member object of a class for which a copy constructor exists (either explicitly declared by the class designer, as in the case of the previous String class, or synthesized by the compiler, as in the case of class Word)

编译器需要往 copy constructors 里插入成员的拷贝构造函数调用。

3.2.2 基类含有拷贝构造函数

When the class is derived from a base class for which a copy constructor exists (again, either explicitly declared or synthesized)

编译器需要往 copy constructors 里插入基类的拷贝构造函数调用。

3.2.3 声明或继承了虚函数

When the class declares one or more virtual functions
为了保证 vptr 的正确设置，必须拒绝 bit-wise copy。【注意】如果是同类对象的赋值 bit-wise copy 可以使用，关键是涉及到用一个子类对象初始化一个基类对象的时候，这种 bit-wise copy 就会产生错误，因为基类的 vptr 现在被指向了子类的 vptr 指向的内容。但是在赋值时，子类的部分已经被切割掉了，如果调用一个虚函数并且虚函数包含了对子类独有数据成员的操作，这样会造成 blow up。

也就是说合成出来的 copy constructors 主要说为了完成这个 vptr 的正确设置。

3.2.4 虚基类相关

When the class is derived from an inheritance chain in which one or more base classes are virtual

碰到的问题与 3.2.3 是一样的，不过这里说为了防止不同 class 的对象的赋值中，the virtual base class pointer/offset 被误设。

3.3 Program Transformation Semantics

3.3.1 参数传递如何实现呢？

【实现模型 1】引入临时对象，使用拷贝构造函数初始化。然后利用 bitwise copy 将其拷贝到 x0 的位置。比如：

void foo(X x0);

```
X xx;
foo( xx );
```

改写成

X __temp0;

```
__temp0.X::X ( xx );
foo( __temp0 );
```

还有一件事需要做，修改 foo 的声明，可以避免 bit-wise copy 的那一步。

```
void foo( X& x0 );
```

也就是生成一个临时对象，然后调用拷贝构造函数用实参初始化这个临时对象。然后往函数里传递这个临时对象的引用。

【实现模型 2】直接在程序栈上的活动记录里进行拷贝构造。

3.3.2 返回值实现

【实现模型 1】cfont 的实现采用了双阶段转化。1.首先声明一个额外的参数，类型上类对象的引用，用来存放返回结果。2.对这个参数利用返回值进行拷贝初始化。过程类似于参数传递，也是要定义一个临时对象，用来保存返回值，然后在函数内部调用拷贝构造函数用那个 return 值进行初始化。

```
X bar()
{
   X xx;
   // process xx ...
   return xx;
}
```

编译器转化后

```
// function transformation to reflect
// application of copy constructor
// Pseudo C++ Code
void bar( X& __result )
{
   X xx;
   // compiler generated invocation
   // of default constructor
   xx.X::X();
   // ... process xx
```

```
    // compiler generated invocation
    // of copy constructor
    __result.X::X( xx );
    return;
}
```

【实现模型 2】Named Return Value (NRV) optimization，具名返回值优化，实现这种优化有个前提，就是必须提供 **copy constructor，因为 NRV 优化的目的就是为了剔除 copy constructor 的使用。只有有了才能被剔除，否则谈不上剔除。一般的如果不优化 NRV，其实现就是类似于模型 1 中的过程，而实现了优化的过程则上这样的。**

```
X bar()
{
    X xx;
    // ... process xx
    return xx;
}

__result is substituted for xx by the compiler:

void bar( X &__result )

{
    // default constructor invocation
    // Pseudo C++ Code
    __result.X::X();
    // ... process in __result directly
    return;
}
```

是否需要拷贝构造函数呢？

【注意】一方面要考虑默认的语义是否符合我们的需要。另一方面如果对象面临大量的拷贝操作，有必要实现一个拷贝构造函数以支持 NRV 优化。但是如果想使用底层的 memcpy 之类的直接进行 bit wise copy，注意是否真的是 bit wise copy 拷贝，比如如果是 virtual，这样可能破坏调 vptr。

3.3.3 对象拷贝

3.4 Member Initialization List

以下情况必须使用成员初始化列表。

1. When initializing a reference member
2. When initializing a const member
3. When invoking a base or member class constructor with a set of arguments

对于第四种情况，member class constructor with a set of arguments，如果存在一个默认构造器，程序可以编译运行但是效率不高。

对于成员初始化表究竟上如何处理的呢？**【实现模型】编译器首先保证初始化列表内的初始化动作，全部插入到 expicit user code 之前。同时按照成员的声明顺序安排初始化的次序。**

在构造函数内调用成员函数是可以的，因为 this 指针这时已经设定好了。但如果妄想调用子类的虚函数，则可能上危险的行为。比如：

```
// is the invocation of FooBar::fval() ok?
class FooBar : public X {
   int _fval;
public:
   int fval() { return _fval; }
   FooBar( int val )
      : _fval( val ),
        X( fval() )
      {}
   ...
};
```

实际代码可能被转化成 X::X(this, this->fval());因为这时的对象内的 vptr 还未被建构起来。或者已经建构起来的那个 vptr 是指向父类的虚函数表。故是十分危险的。

另外编译器如何组织代码顺序呢？比如如果父类的初始化在构造函数内写了，然后成员的却写到了初始化列表中？
实验表明编译器仍然会在所有的初始化之前，重新生成一个父类的构造函数。即使被调用两遍，也要生成。当如果将父类的写到了初始化列表，便不会生成。

4 The Semantics of Data

先从一个简单的对象实例看起：

```
class X {};
class Y : public virtual X {};
class Z : public virtual X {};
class A : public Y, public Z {};
```

这个例子里，对象的大小是多少呢？

在 devcpp 下，它们的大小分别是 1，8，8，12。

实际上对象的大小收到以下方面的影响：

1. 语言本身造成的额外负担。比如为了支持 virtual base class，子类对象可能被安插进一个指针，该指针指向虚基类子对象或者指向一个相关表格，该表格存放的是虚基类子对象的指针或者 offset。
2. 编译器对于特殊情况所做的优化处理。There is the 1 byte size of the virtual base class X subobject also present within Y (and Z).传统上，这个 X subobject 被放置在 Y Z 的末尾部分。然而有的编译器针对这种情况可以通过将虚基类部分放到对象布局的开头，优化掉那个为了区分两个空对象而额外添加的 1byte。
3. 内存对齐的要求。

Class Y，Z 组成包括了继承来的为了区分空对象额外添加的大小为 1byte 的 X 部分，还有一个 4bytes 的指针指向虚基类子对象。经过对齐处理后就是 8bytes。

classA 的内容，由以下部分组成：

1. 被大家共享的唯一一个 X 实体，大小为 1byte
2. Y，Z 减去 X 部分后的那些大小，每个为 4byte，总共为 8byte
3. class A 自己的大小 0byte
4. 经过对齐处理后的，变成了最终的 12bytes。

C++标准并没有规定，诸如基类子对象的排列顺序，不同访问 level 的数据的分布，也没有规定虚函数或者虚基类的实现细节。这些都是由编译器实现定义的。

对象的大小可能并不如你所预期的，因为

1. 编译器加入了一些额外的数据，用来支持比如 virtual 这样的机制
2. 对齐规则

4.1 数据成员绑定

注意早期的成员作用域解析规则与现在的是不同的。现在的 c++标准规定，对一个成员函数 body 的解析，必须等到 class 声明结束再进行。**【注意】然而对于成员函数的参数列表，它们仍然在第一次遇到的地方就进行解析。因此对于** extern and nested type names 的非预期绑定仍然可能发生。

```
typedef int length;
```

```
class Point3d
{
public:
   mumble( length val ) { _val = val; }
   length mumble() { return _val; }
private:
   typedef float length;
   length _val;
   // ...
};
```

这样 mumble(length val)会被解析成 int 而不是 float。因此作为防御性程序设计，最好把关于数据及类型的声明放到类的最开始处。

4.2 数据成员布局

C++标准仅仅规定了对于同一个 access section 的成员保证，"later members have higher addresses within a class object"。而对其他的则没有定义。
比如对于虚函数的 vptr 的放置位置就没有明确规定，传统的实现把它放到类中所有显示声明的成员的后面，但是最近的一些实现中把它放到了类对象的开头。另外编译器也运行不同访问区段的成员自由放置。

4.3 Datameber 的存取

```
Point3d origin, *pt = &origin;
origin.x = 0.0;
pt->x = 0.0;
```

两种访问方式有何区别呢？
【注意】当 x 是一个继承自虚基类的成员时，便会有所不同，因为无法确定 px 指向的类型，因此也就不知道 x 的 offset 的位置，pt->x 必须延迟到运行时决定。origin.x 则在编译时已经确定了。

4.3.1 静态数据成员

1. 除了可见性的不同，实际跟全局变量没有区别，不会产生什么额外开销。而从指令执行的观点看，它也是唯一一个通过对象和指针来存取完全相同的情况。
2. 如果这个静态数据成员从其他类中继承而来，比如从一个虚基类里。这也无关紧要。
3. 若取地址，则得到一个指向该数据类型的地址，而不是 a pointer to class member。比如&Point3d::chunkSize;得到的是一个真实的地址，而如果 chunkSize 只是个非静态的成员，那么得到的只是一个指向 class member 的偏

移。

4. 如果两个类含有同名的该类型，则对其进行 name-mangling。

4.3.2 非静态数据成员

实际上它们的访问，必须通过一个显式或者隐式的 object 进行。比如编译器默默传递个成员函数的 this 指针。

```
the address of &origin._y; is equivalent to the addition of &origin +
( &Point3d::_y - 1 );
```

之所以-1，是因为为了区分 a pointer to data member that is addressing the first member of a class and a pointer to data member that is addressing no member.c++给指向成员的指针都加了个 1.

4.4 继承与数据成员

加上多态后，考虑 vptr 的放置位置选择：

1. 放到对象末尾。最初就是这样做的，保持了基类部分的 c struct 布局。
2. 随着虚拟继承以及抽象基类概念的发展，开始把它放到开始。这对于 some virtual function invocations through pointers to class members under multiple inheritance，Otherwise, not only must the offset to the start of the class be made available at runtime, but also the offset to the location of the vptr of that class must be made available.当然这丧失了与 c 的兼容性。
   另外，这种情况下，如果基类没有虚函数(这样基类对象的布局中，就不会含有那个 vptr，而子类含有 vptr，而 vptr 又处在对象布局的开始，这样就得调整对象指针)，而子类含有，则在子类指针向基类指针转换中，必须调整指针的值。

4.4.1 多重继承

在多重继承和虚拟继承下，编译器的介入变得更加必要。在这里，子类向第二个基类的转换，以及虚函数的支持变的更加复杂。子类与第二父类指针间的转换变得复杂，必须对指针的值进行调整，比如加上第一父类所占据的大小。

```
Vertex3d *p3d;
Vertex   *pv;
// Pseudo C++ Code
pv = (Vertex*)((char*)p3d) + sizeof( Point3d );
```

【注意】简单的转换还不行，因为 p3d 可能是 null。

对于多重继承中基类在内存中的布局，标准并没有规定。传统的实现都按照声明的次序来安排。

多重继承下的 vptr 又是如何安排的呢？是否有多个 vptr 呢？

是的多重继承或完整的继承父类的数据布局，包括 vptr，也就是说一个子类可能具有继承自多个父类的不同的 vptr。
但是如果含有多个 vptr，又如何实现多态呢？
实际上是这样的，每个类的 vptr 都被放到了类对象的起始位置，这样即使子类从父类继承下来了父类的那个 vptr，也不会妨碍多态的实现，因为它们各自的 vptr 依然都在一个相同的 offset 上。

4.4.2 虚拟继承

多重继承语音产生的一个副作用就是需要支持一种共享子对象继承。语言级别的解决方案就是使用虚继承。
实现上的挑战就是提供一个有效的方法把不同对象维护的那个虚基类对象折叠成一个单一的对象，并且还要保存子类与父类之间多态指针的多态赋值操作。

【实现模型】常见的实现方案是将含有一个或多个虚基类的子类，分成两部分，一个不变局部和一个可变局部。对于不变局部来说，无论怎样继承，这个部分距离对象起始位置的 offset 是不变的，所以可以直接进行访问。共享局部则保存了虚基类子对象，每继承一次，里面的数据可能会变化，所以无法直接访问。必须采用间接访问，而编译器实现方式的不同就在于间接访问方式的不同。目前有三种主流的实现模型：

```
class Point2d {
public:    ...
protected:
   float _x, _y;
};

class Vertex : public virtual Point2d {
public:    ...
protected:
   Vertex *next;
};

class Point3d : public virtual Point2d {
public:    ...
protected:
   float _z;
};
class Vertex3d :
   public Point3d, public Vertex {
public:    ...
protected:
   float mumble;
};
```

通常的分布策略，是首先安排不变局部，然后建立共享局部。然而还有一个问题，

如何访问类的共享部分。

1. cfont 模型：在每个子类对象中安插一个指针，该指针指向该类的虚基类。这样对于虚基类的成员的访问就是通过这个指针间接实现的。比如下面的访问：

```
void
Point3d::
operator+=( const Point3d &rhs )
{
    _x += rhs._x;
    _y += rhs._y;
    _z += rhs._z;
};

under the cfront strategy, this is transformed internally into

// Pseudo C++ Code
__vbcPoint2d->_x += rhs.__vbcPoint2d->_x;
__vbcPoint2d->_y += rhs.__vbcPoint2d->_y;
_z += rhs._z;
```

子类与基类的转换：

Point2d *pv = pv3d;会被转换为

Point2d *pv = pv3d ? pv3d->__vbcPoint2d : 0;

该模型具有两个缺点：

(1):每个对象针对其每一个虚基类产生一个指针，如果我们希望每个对象具有固定的指针，不因虚基类个数而变化，该如何实现？

(2):如果虚拟继承串的加长，可能导致多次的间接访问。理想上我们希望有固定的访问时间，不应因虚拟继承串的增长而增加。

MetaWare and other compilers still using cfront's original implementation model solve the second problem by promoting (by copying) all nested virtual base class pointers into the derived class object. This solves the constant access time problem, although at the expense of duplicating the nested virtual base class pointers.

至于第一个问题，一般有两种解决方法，microsoft 引入所谓的虚基类表，如果有一个或者多个虚基类，则由编译器插入一个指针指向该表。而真正的虚基类指针，则保存在该表内。第二种方式，就是在虚函数表中，放置虚基类的 offset。

在第二种方式下：上面的 Point3d operator+=变成了

```
// Pseudo C++ Code
(this + __vptr__Point3d[-1])->_x += (&rhs + hs.__vptr__Point3d[-1])->_x;
(this + __vptr__Point3d[-1])->_y += (&rhs + hs.__vptr__Point3d[-1])->_y;
_z += rhs._z;
```

这一套机制只是为了处理多态的应用，即运行时的需求而设立的。对于

```
Point3d origin;
...
origin._x;
```

编译时就可以确定 origin._x 的地址。

4.5 成员访问效率

4.6 数据成员指针

【注意】为了区别指向第一个成员的指针与为指向任何成员的指针，编译器给所有的指针加上了 1.

【注意】实际上侯捷翻译错了，原文的：The result of

```
& origin.z
```

adds the offset of z (minus 1) to the beginning address of origin.

本身就是正确的。

另外 Under multiple inheritance, the combination of a second (or subsequent) base class pointer to a member bound to a derived class object is complicated by the offset that needs to be added.比如：

```
struct Base1 { int val1; };
struct Base2 { int val2; };
struct Derived : Base1, Base2 { ... };
void func1( int Derived::*dmp, d *pd )
{
   pd->*dmp;
}
void func2( d *pd )
{
   int Base2::*bmp = & Base2::val2;
   func1( bmp, pd )
}
```

Bmp必须进行调整，因为传递的是个基类 Base2的成员指针，而需要的是 Derived 的成员指针。

```
func1( bmp + sizeof( Base1 ), pd );
```

In general, however, we cannot guarantee that bmp is not 0 and so must guard against it:// internal transformation

```
// guarding against bmp == 0
func1( bmp ? bmp + sizeof( Base1 ) : 0, pd );
```

5 The Semantics of Function

5.1 Member 的各种调用方式

静态成员函数，普通成员函数，普通函数。经过编译器处理后，实际上最终效果上一样的，因此时间花费也是相同的。

5.2 虚成员函数

实现模型，vptr 与 vtbl。Vtbl 是在编译时，每个类(具有虚函数：继承或者自己声明的虚函数，或者纯虚函数)都会产生的一个虚函数表。每个类可能不止一个虚函数表，比如多重继承的情况下。

【注意】一个函数可能具有多个虚函数表，虚函数表中的内容包括该类自己具有的虚函数，从其他函数继承来的虚函数，纯虚函数实体，同时为了支持虚继承，某些实现会把虚基类地址也放到 vtbl 里。

Vtbl 则是每个对象都拥有的一个指向虚函数表的指针。在对象产生时的构造函数里设置，由编译器生成设置代码。一般而言我们不知道 ptr 所指对象的真正类型，但是经过 ptr 总是可以访问对象的虚表。虽然不知道那个 z()函数实体被调用，但是 z()地址总是会放到相同的 slot 中。于是一个多态的访问

Ptr->z();总是可以被转换成

(*Ptr->vptr[4])(ptr),这里 vptr 表示编译器安插的指针，4 表示 z()放置的 slot 编号。

5.3 函数效能

5.4 指向成员函数的指针

去一个非静态成员函数地址，如果它是非虚的，得到的将是它在内存中的真正地址。但是它的调用需要 this 指针，所以需要与对象绑定的.或者->调用

。

如果该函数是虚的话，则更加复杂。虚拟机制仍然实行。**【实现模型】因为编译时，并不能确定是哪个虚函数调用，所以得到的实际上是该虚函数的 slot，也就是在虚表中的索引值。但是如何区分某函数指针到底是实际地址还是 slot 呢？因为真正的函数地址都会很大，在地址空间的前部分由操作系统占据。而 slot 则是从 0 开始的很小的一些值，根据这个就可以区分了。**

多重继承，虚继承下的函数指针。Strustrop 设计了如下结构

```
// fully general structure to support
// pointer to member functions under MI
struct __mptr {
```

```
    int delta;
    int index;
    union {
        ptrtofunc  faddr;
        int        v_offset;
    };
};
```

What do these members represent? The index and faddr members, respectively, hold either the virtual table index or the nonvirtual member function address. (By convention, index is set to ? if it does not index into the virtual table.) Under this model, an invocation such as

```
( ptr->*pmf )() becomes
( pmf.index < 0 )
    ? // non-virtual invocation
    ( *pmf.faddr )( ptr )
    : // virtual invocation
    ( * ptr->vptr[ pmf.index ]( ptr );
```

微软则采用 vcall thunk 避免这种检查的时间耗费。

5.5 Inline 函数

6 Semantics of Construction, Destruction, and Copy

一般来说，成员必须初始化，而且在构造函数中完成，如果一个类的变量想让它的子类初始化值，那也应该提供一个构造函数接口，子类通过调用这个接口完成初始化。

【注意】一个纯虚函数是可以被定义并且调用的，不过必须经由静态机制调用，不能通过多态调用。而对于一个纯虚析构器，则必须提供它的定义，因为编译器默认以静态方式在子类析构器中调用它，而这个定义必然是可以在静态决议出来的。

6.1 无继承下的对象构造

Global 对象的初始化方法：
使用 new 运算符，delete，会调用 default constructor 和 destructor？
给类加入 virtual 函数，不仅会导致 vptr 的生成，也会导致构造器，拷贝构造器，析构器进行特定的非平凡的行为。

6.2 继承体系下的对象构造

【注意】构造函数所完成的工作：

The general sequence of compiler augmentations is as follows:

1. The data members initialized in the member initialization list have to be entered within the body of the constructor in the order of member declaration.
2. If a member class object is not present in the member initialization list but has an associated default constructor, that default constructor must be invoked.
3. Prior to that, if there is a virtual table pointer (or pointers) contained within the class object, it (they) must be initialized with the address of the appropriate virtual table(s).
4. Prior to that, all immediate base class constructors must be invoked in the order of base class declaration (the order within the member initialization list is not relevant).
    - If the base class is listed within the member initialization list, the explicit arguments, if any, must be passed.
    - If the base class is not listed within the member initialization list, the default constructor (or default memberwise copy constructor) must be invoked, if present.
    - If the base class is a second or subsequent base class, the this pointer must be adjusted.
5. Prior to that, all virtual base class constructors must be invoked in a left-to-right, depth-first search of the inheritance hierarchy defined by the derived class.
    - If the class is listed within the member initialization list, the explicit arguments, if any, must be passed. Otherwise, if there is a default constructor associated with the class, it must be invoked.
    - In addition, the offset of each virtual base class subobject within the class must somehow be made accessible at runtime.
    - These constructors, however, may be invoked if, and only if, the class object represents the "most-derived class." Some mechanism supporting this must be put into place.

6.2.1 虚拟继承：

如果避免多重继承下，虚基类的重复初始化调用。可以通过添加一个参数，控制

虚基类的构造函数的调用。注意类的所有虚基类的，它们的构造都直接出现在构造函数中，从左到右，由深到浅，而不像普通基类那样，只有上一层的基类构造函数被调用。

6.2.2 Vptr 初始化语义学

涉及到在构造函数内部调用虚函数会如何的机制？
通过将 vptr 的设置控制在，所有基类构造之后，用户代码之前。来保证这种对象构造过程中虚函数调用的行为。
如果在初始化列表调用一个虚函数，如果用于初始化该类的数据成员，可以保证该行为是安全的，因为这时该类的 vptr 已经设置。可以保证正确的调用那个函数，但无法保证所有依赖的数据的初始化已经完成。
但是初始化一个基类部分呢？
这是更不安全的，因为这时 vptr 没有被设定好，或者指向了另一个错误的类，所有依赖的数据成员也未初始化。

6.3 对象复制语义学

Copy constructor 不应该成为 copy assignment operator 出现的理由，因为有时候我们提供一个 Copy constructor 是为了打开 NRV 优化。
【注意】在这里与 copy assignment operator 又与 Copy constructor 又个不同，我们不需要修改 vptr 的值，因为它们在初始化的时候已经设定好了。而赋值并不应该改变 vptr。同时这个规定也说明了，当存在 vptr 时，使用 bitwise copy 是错误的。
【注意】但是对于虚拟继承下的=操作，其行为可能不佳，需要小心设计和说明。因为编译器并不尝试取得正确的语义，或者像 constructor 那样，通过一些方式保证虚基类的构造函数只被调用一次。在这里，虚基类的=操作符可能在调用每个基类实例时被调用多次。

所以尽量不要在虚基类里声明任何数据成员。

6.4 对象效率

6.5 析构语义学

如果 class 没有定义 destructor，编译器只有当 class 的成员对象或者基类，含有 destructor 时才会合成一个。如果不是上述情况，甚至存在虚函数，也不会被合成出来。

7 Runtime Semantics

7.1 对象的构造和析构

Global 对象，local static 对象如何保证构造一次和条件析构，对象数组
【注意】为何不能使用构造函数地址？

7.2 New 和 delete 操作符

Vec_new 的对于 new 数组的作用

7.3 临时对象

临时对象生命期

8 站在对象模型的顶端

总结：（结合 3，4，5 章），讨论对象在普通继承，虚继承，多重继承，多重虚继承下的详细布局，构造，拷贝，析构，初始化的行为。