C类是C结构体面向对象的扩展，因此，在结束数据结构的讨论时，我们有必要介绍一下已编译的C代码的各种特性。牢固掌握C++语言的基础知识，对于你理解已编译C++代码将大有裨益。在源代码层次上完全掌握继承和多态等面向对象的概念会非常困难。尝试在汇编语言层次上深入这些概念，但却不了解它们在源代码层次上的意义，毫无疑问，你会陷入困境。

this指针

this可以看成是传递到所有非静态成员函数的第一个隐藏参数,Microsoft Visual C++利用thiscall调用约定，并将this传递到ECX寄存器中。GNU g++编译器则把this看做是非静态成员函数的第一个（最左边）参数，并在调用该函数之前将用于调用函数的对象的地址作为最后一项压入栈中。

从逆向工程的角度看，在调用函数之前，将一个地址转移到ECX寄存器中可能意味着两件事情。首先，该文件使用Visual C++编译；其次，该函数是一个成员函数。如果同一个地址被传递给两个或更多函数，我们可以得到结论，这些函数全都属于同一个类层次结构。

另外，如果发现一个函数向其他函数传递this指针，则这些函数可能和传递this的函数属于同一个类。

虚函数

虚函数是指在声明前使用关键字virtual的函数，如 virtual void func()，作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

纯虚函数，在虚函数后加“=0”，如 virtual void func()=0，类似于Java中的abstract方法，子类必须实现。

如果父类或者祖先类中函数func()为虚函数，则子类及后代类中，函数func()是否加virtual关键字，都将是虚函数。

虚函数是C++中用于实现多态的机制。核心理念就是通过基类访问派生类定义的函数。

虚表

C++实现虚函数的方法是：为每个类对象添加一个隐藏成员，隐藏成员保存了一个指针，这个指针叫虚表指针（vptr），它指向一个虚函数表（virtual function table, vtbl）。在运行时创建对象时，对象的虚表指针将设置为指向合适的虚表。如果该对象调用一个虚函数，则通过在该对象的虚表中进行查询来选择正确的函数。因此，虚表是在运行时解析虚函数调用的基本机制。

每个类使用一个虚函数表，每个类对象用一个虚表指针。

使用虚表指针导致的一个后果是，在操纵IDA中的类时，你必须考虑到虚表指针。。前面我们讲过，C类是C结构体的一种扩展。因此，我可以利用IDA的结构体定义来定义C类的布局。对于包含虚函数的类，你必须将一个虚表指针作为类中的第一个字段。在计算对象的总大小时，也必须考虑到虚表指针。这种情况在使用new操作符[插图]动态分配对象时最为明显，这时，传递给new的大小值不仅包括类（以及任何超类）中的所有显式声明的字段占用的空间，而且包括虚表指针所需的任何空间。

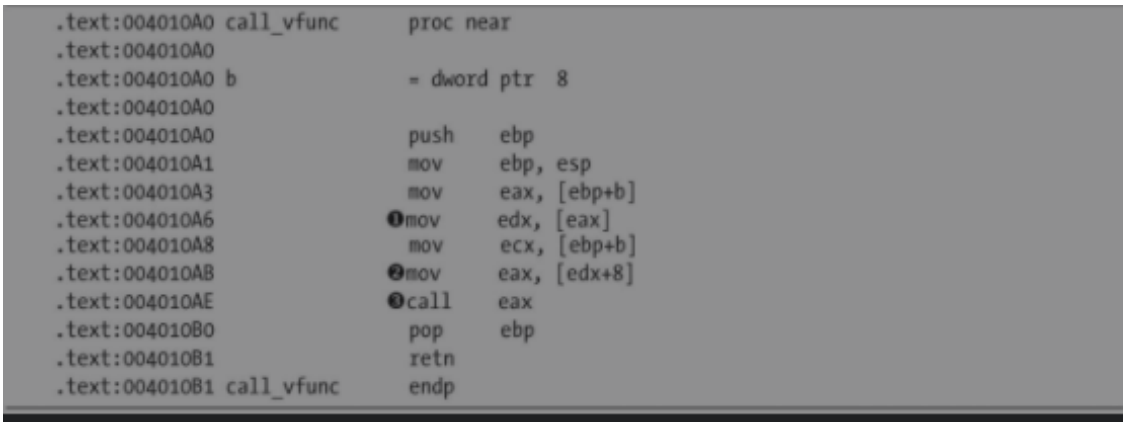
实际例子

```
void call_vfunc(BaseClass *b){
    b->vfunc3();
}

int main(){
    // SubClass是BaseClass的子类，vfunc3是BaseClass中声明的虚函数
    BaseClass *bc = new SubClass();
    call_vfunc(bc);
}
```


上述的例子通过new创建了一个SubClass的堆对象，地址保存在BaseClass的一个指针中，然后这个指针传递给一个函数并用刚才的指针来调用虚函数vfunc3。

下面是call_vfunc的反汇编指令：



很难读懂，b是一个指向SubClass对象的地址，在1处读出了某个基于b的偏移地址，然后在2 处读出了基于b的一个偏移地址（从后面call可以推断出来是vfunc3的地址）并调用，因此可以发现，2处的操作和结构体成员访问的方式非常类似，因此我们可以利用创建一个结构体来优化这段代码的可读性，因为虚表中存放的是每个函数的指针，因此结构体的大小是虚方法个数，大小为dd，最终2处的汇编指令将被优化为mov eax,[edx+SubClass_vtable.vfunc3]类似的指令。

对象的生命周期

了解对象的构建和撤销机制，有助于明确对象的层次结构和嵌套对象关系，并有助于迅速确定类构造函数和析构函数。

对全局和静态分配的对象来说，构造函数在程序启动并进入main函数之前被调用。栈分配的对象的构造函数在对象进入声明对象的函数作用域中时被调用。

如果对象在程序堆中动态分配，则创建对象分为两个步骤。第一步，调用new操作符分配对象的内存。第二步，调用构造函数来初始化对象。

执行一个构造函数的步骤：（与Java类似）

- 如果类拥有一个超类，则调用超类的构造函数。
- 如果类包含任何虚函数，则初始化虚表指针，使其指向类的虚表。这样做可能会覆盖一个在超类中初始化的虚表指针
- 如果类拥有本身就是对象的数据成员，则调用这些数据成员的构造函数。
- 最后，执行特定于代码的构造函数。这些是程序员指定的、表示构造函数C++行为的代码。

析构函数基本上按相反的顺序调用。

C++名称重载机制

为了给重载函数生成唯一的名称，编译器用其他字符来修饰函数名称，用来编码关于函数的各种信息。编码后的信息通常描述函数的返回类型、函数所属的类、调用该函数所需的参数序列（类型和顺序），是C++编译器用于区分重载函数的机制。

但如何修改并不是C++语言的规范，因此各编译器都有自己的名称改编约定。

默认情况下，在程序中遇到一个改编名称时，IDA会在反汇编代码清单中该名称出现的位置以注释的形式显示该名称的原始名称，也可以通过Options -> Demangled Names设置在注释中出现还是直接以名称出现。

改编名称能提供大量与函数签名有关的信息，它们减少了IDA为理解传递给函数的参数的数量和类型所需的时间。

运行时类型识别

C提供各种操作符，可进行运行时检测，以确定（typeid）和检查（dynamic_cast）一个对象的数据类型。为实现这些操作，C编译器必须将类型信息嵌入到一个程序的二进制文件中，并执行能够百分之百确定一个多态对象的类型的过程，但是typeid是如何知道指针指向的对象的真正类型？

问题的答案非常简单。因为每个多态对象都包含一个指向虚表的指针，编译器将类的类型信息与类虚表存储在一起。具体来说，编译器在类虚表之前放置一个指针，这个指针指向一个结构体，其中包含用于确定拥有虚表的类的名称所需的信息。需要注意的是，只有使用typeid或dynamic_cast操作符的C++程序才需要RTTI信息。多数编译器都提供一些选项，禁止不需要RTTI的二进制文件生成RTTI。因此，如果RTTI信息并不一定都存在。

继承关系

确定某个继承体系的最简单方法是，观察在创建对象时被调用的超类构造函数的调用链。内联（）构造函数是这种方法成功与否的唯一最大障碍。如果使用了内联构造函数（被inline关键字声明的函数，将被编译器作为宏处理，并且调用该函数的调用指令将会被替换为该函数的代码），我们就不可能知道一个超类构造函数实际上已经被调用。

分析和比较虚表是另一种用于确定继承关系的方法：

- 如果两个虚表包含相同数量的条目，则与这两个虚表对应的类之间可能存在着某种继承关系。
- 如果类X的虚表包含的条目比类Y多，则X可能是Y的子类。
- 如果X包含的条目也可以在Y的虚表中找到，则必定存在下面一种关系：X是Y的子类，Y是X的子类，或者X和Y全都是同一个超类Z的子类。
- 如果X包含的条目也可以在类Y的虚表中找到，并且X的虚表中至少包含一个纯调用条目（purecall，如果子类没有实现父类的纯虚函数，那么编译器会插入一个错误处理函数的地址通常名为purecall，这个方法被调用时会令程序终止），而Y的虚表中并没有这个条目，那么Y是X的子类。
-