

逆向 C++

这些年来，逆向工程分析人员一直是凭借着汇编和 C 的知识对大多数软件进行逆向工程的，但是，现在随着越来越多的应用程序和恶意软件转而使用 C++ 语言进行开发，深入理解 C++ 面向对象方式开发的软件的反汇编技术就显得越发的必要。本文试图通过分析在反汇编时如何手工识别 C++ 对象，进而讨论如何自动完成这一分析过程最终介绍我们自己开发的自动化工具，一步一步的帮助读者掌握逆向 C++ 程序的一些方法。

Team 509

作者：Paul Vincent Sabanal
Mark Vincent Yason

译者：Hannibal509@gmail.com

逆向C++.....	1
I. 引言和必要性	3
II. 手工方法	3
A. 识别类及其构造函数.....	3
B. 识别类.....	10
1) 识别构造函数和析构函数.....	10
2) 利用RTTI识别多态类.....	12
C. 判别类与类之间的关系.....	18
1. 通过分析构造函数来分析类与类之间的关系.....	18
2. 通过RTTI分析类与类之间的关系	19
D. 辨别类的成员	21
III. 自动化	21
A. OOP_RE.....	21
B. 为什么选择静态分析的方式？	22
C. 自动化分析的策略.....	22
通用算法.....	22
1. 利用RTTI识别多态类.....	23
2. 利用虚函数表识别多态类（不使用RTTI）	24
3. 通过搜索构造/析构函数来识别类	25
4. 识别类与类之间的继承关系	28
5. 类的成员的识别	29
D. 显示结果	29
1. 注释各种结构体	29
2. 改进过的调用图表.....	30
E. 分析结果可视化：UML图	30
IV. 小结	32

Jean 509

I. 引言和必要性

对于逆向工程分析人员来说，能从一个二进制可执行文件中识别出 C++ 程序的结构，并且能标识出各个主要的类，以及这些类之间的关系（继承、派生等）是非常重要的。为了能做到这一点，逆向工程分析人员就必须要（1）能识别出这些类（2）能识别出这些类之间的关系（3）识别出类中的各个成员。本文就是要教大家能做到上述三点。首先我们先来讨论如何手工的分析一个 C++ 程序编译的二进制可执行文件，从中提取出有关的类的信息。然后我们再来讨论如何自动化这一手工分析的过程。

当然，要做到这一点需要你花上不少的功夫学习很多技巧，但是为什么我们要学习并掌握这些东西呢？我认为有下面这三点理由要求我们这么做：

1) 用 C++ 开发的恶意软件越来越多了

根据我们分析恶意软件的经验，现在我们要分析的恶意软件中使用 C++ 开发的恶意软件越来越多了。你知道，把这些恶意软件扔到 IDA 里去进行静态分析的难度会比较大，因为相对于 C 中的直接函数调用而言，静态分析 C++ 中的虚函数调用就比较困难，因为 C++ 中的调用虚函数是采用间接调用的方式，有时你甚至都能难确定某个函数是否被调用过。比如臭名昭著的 Agobot 病毒就是用 C++ 写的，另外我自己的蜜罐里最近也捕获了一些新的 C++ 写的恶意软件。

2) 用 C++ 开发的现代的应用程序也越来越多了。

随着操作系统和应用程序的规模和复杂度的与日俱增，C++ 越来越受软件开发人员的青睐。这也导致了在漏洞发掘等逆向工程任务中面对 C++ 语言编写的软件的可能性也就越来越大。所以逆向分析人员必须要掌握 C++ 相关的逆向工程技术

3) 关于 C++ 的逆向工程资料极少

我们相信把 C++ 的逆向工程资料整理成册，提供给逆向工程分析人员是一件功德无量的好事，因为这一方面的资料实在是太少了。（译注：在《黑客反汇编揭密》一书中有部分讨论）

注意：本文中讨论的 C++ 可执行文件仅限于使用 Microsoft Visual C++ 编译器编译出的 C++ 可执行文件。

II. 手工方法

这一节，主要讨论手工分析 C++ 可执行文件的方法。主要讨论如何识别类及其成员（变量，函数以及构造函数和析构函数）以及类与类之间的关系。

A. 识别类及其构造函数

要识别出类的成员及类与类之间的关系，我们首先要把各个类给识别出来，所以我们先来识别类及其构造函数。我们可以通过下列特征从一个可执行文件中把类和它的构造函数识别出来：

1) 大量的使用 ECX 寄存器（作为 this 指针）。我们应该首先注意到的是在反汇编代码中会大量出现使用 ECX 寄存器（用来传递 this 指针）的情况。如下图，我们看到在给 ECX 寄存器赋值之后，马上调用了一个函数。

```
.text:004019E4      mov     ecx, esi
.text:004019E6      push    0BBh
.text:004019EB      call    sub_401120
```

另外，我们在函数中可能会经常看到 ECX 寄存器还没有初始化就直接被使用的情况（如下图），这时我们基本上就可以猜出来：这个函数应该就是某个类的成员函数。

```
.text:004010D0 sub_4010D0      proc near
.text:004010D0          push    esi
.text:004010D1          mov     esi, ecx
.text:004010DD          mov     dword ptr [esi], offset off_40C0D0
.text:00401101          mov     dword ptr [esi+4], 0BBh
.text:00401108          call    sub_401EB0
.text:0040110D          add    esp, 18h
.text:00401110          pop    esi
.text:00401111          retn
.text:00401111 sub_4010D0      endp
```

2) 调用约定。这一点与 1) 有关，类的成员函数在被调用时基本上是把函数的参数压入栈中，而使用 ECX 传递 this 指针。如下面这个例子，在为类新建了一个对象之后，new 返回的指针（该指针指向分配给对象的地址）EAX 的值马上被传给了 ECX，然后就调用了构造函数。

```
.text:00401994 push    0Ch
.text:00401996 call    ??2@YAPAXI@Z      ; operator new(uint)
.text:004019AB mov     ecx, eax
...
.text:004019AD call    ClassA__ctor
```

另外，我们有时还会遇到一些间接函数调用，这很可能是调用类的虚函数，当然，在静态分析的情况下（即不是在调试器中进行动态分析）如果不是事先明确的知道这个虚函数是哪个类的，要深入跟踪这个虚函数还是很困难的。我们考虑下面这个例子：

```
.text:00401996 call    ??2@YAPAXI@Z      ; operator new(uint)
...
.text:004019B2 mov     esi, eax
...
.text:004019FF mov     eax, [esi]       ; EAX = vftable
.text:00401A01 add    esp, 8
.text:00401A04 mov     ecx, esi
.text:00401A06 push   0CCh
.text:00401A0B call    dword ptr [eax]
```

在这个例子里，我们首先要知道 ClassA 的虚函数表（virtual function table）在哪里，然后才能根据虚函数表来确定虚函数的代码所在的位置。

3) STL（标准模板库 Standard Template Library）中的代码和可执行文件导入的 DLL。另外，如果我们在检查二进制可执行文件时发现这个可执行文件使用了 STL 中的代码，这一点可以通过分析可执行文件要求导入的函数或者通过 IDA 的 FLIRT 之类的库签名识别方法来做到：

Address	Ordinal	Name	Library
00402030		InterlockedExchange	KERNEL32
00402038		?endl@std@@YAAV?\$basic_ostream@DU?\$char_traits@D@std...	MSVCP80
0040203C		?setstate@?\$basic_ios@DU?\$char_traits@D@std@@@std@@QA...	MSVCP80
00402040		?cout@std@@@3V?\$basic_ostream@DU?\$char_traits@D@std@@@...	MSVCP80
00402044		?uncaught_exception@std@@YA_NZ	MSVCP80
00402048		?sputn@?\$basic_streambuf@DU?\$char_traits@D@std@@@std@...	MSVCP80
0040204C		?_Osfx@?\$basic_ostream@DU?\$char_traits@D@std@@@std@@@...	MSVCP80

下面是调用 STL 中的代码的情况:

```
.text:00401201    mov      ecx, eax
.text:00401203    call
ds: ?sputc@?$basic_streambuf@DU?$char_traits@D@std@@@std@@QAEHD@Z
; std::basic_streambuf<char, std::char_traits<char>>::sputc(char)
```

类的实例

在我们进一步深入讨论之前，逆向工程分析人员还应该熟悉对象（或者说一个类的实例）在内存中是个什么样子，说的文绉绉一点就是类在内存中的布局情况。我们先来看一个简单的类：

```
class Ex1
{
    int var1;
    int var2;
    char var3;
public:
    int get_var1();
};
```

这个类在内存中是这个样子的：

```
class Ex1           size(12):
+---+
0   | var1
4   | var2
8   | var3
| <alignment member> (size=3)
+---
```

最后一个类的成员变量后面有 3 个字节的填充，这是因为要求 4 字节对齐。在 Visual C++ 中，类的成员变量是按照其声明的大小依次排列在内存中的。

看 PPT 里的更清楚一点：

```

class Ex1
{
    int var1;
    int var2;
    char var3;
public:
    int get_var1();
};

```

-d1reportAllClassLayout
compiler switch to generate a
.layout file

class Ex1	size(12):
	+---
0	var1
4	var2
8	var3
	<alignment member> (size=3)
	+---

那么怎么才能得到上面这张图呢？我们可以使用-d1reportAllClassLayout这个编译开关，它可以让 MSVC 编译器（译注：至少是 MSVC 6.0 以上的版本）生成一个.layout 文件，在该文件中包含有大量的极具价值的类的布局信息，包括基类在派生类中的位置，虚函数表，虚基类表（virtual base class table 我们下面会深入讨论），类的成员变量等信息（实际上我们这些图表都是从.layout 文件中取出的）。

那么，如果在一个类中含有虚函数呢？

```

class Ex2
{
    int var1;
public:
    virtual int get_sum(int x, int y);
    virtual void reset_values();
};

```

下面是这个类在内存中的存在形式：

class Ex2	size(8):
	+---
0	{vfptr}
4	var1
	+---

注意指向虚函数表的指针（vfptr）是被添加在最前面的，而在虚函数表里面，各个虚函数是按照其声明的顺序排列的。类 Ex2 的虚函数表如下：

```

Ex2::$vftable@:
0      | &Ex2::get_sum
4      | &Ex2::reset_values

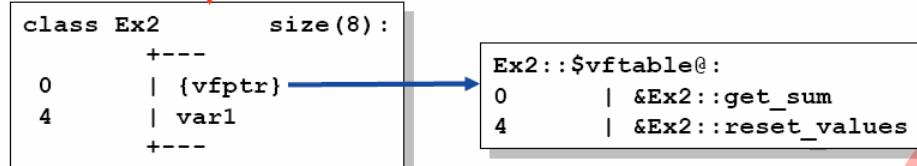
```

下面这个图是 PPT 里的更清楚一点：

```

class Ex2
{
    int var1;
public:
    virtual int get_sum(int x, int y);
    virtual void reset_values();
};

```



当一个类是继承另一个类的话，情况又会怎么样呢？下面讨论一个简单的单一继承关系

```

class Ex3: public Ex2
{
    int var1;
public:
    void get_values();
};

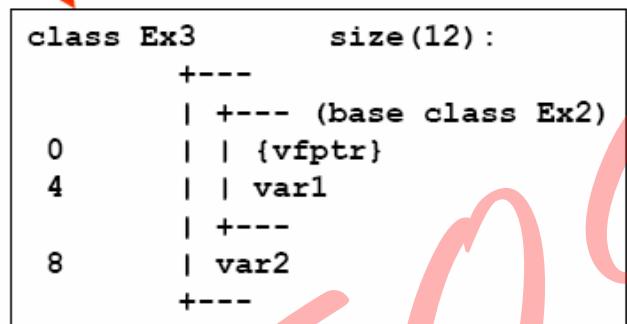
```

在内存中这个类的情况是这样的：

class Ex3 size(12):	
0	+--- (base class Ex2)
4	{vfptra}
8	var1
+---	
var1	
+---	

还是 PPT 上的图好看：

```
class Ex3: public Ex2
{
    int var2;
public:
    void get_values();
};
```



正如您所看到的，派生类只是简单的把基类嵌入到自己内部就完事了。但是
万一要是有多重继承有会有什么情况发生呢？

```
class Ex4
{
    int vari;
    int var2;
public:
    virtual void func1();
    virtual void func2();
};

class Ex5: public Ex2, Ex4
{
    int vari;
public:
    void func1();
    virtual void v_ex5();
};


```

内存中的情况会是这样：

```
class Ex5      size(24):  
    +---  
    | +--- (base class Ex2)  
0     | | {vptr}  
4     | | var1  
| +---  
| +--- (base class Ex4)  
8     | | {vptr}  
12    | | var1  
16    | | var2  
| +---  
20    | var1  
+---
```

```
Ex5::$vftable@Ex2@:  
0     | &Ex2::get_sum  
1     | &Ex2::reset_values  
2     | &Ex5::v_ex5
```

```
Ex5::$vftable@Ex4@:  
| -8  
0     | &Ex5::func1  
1     | &Ex4::func2
```

看 PPT 上的图更清楚一点：

```
class Ex4  
{  
    int var1;  
    int var2;  
public:  
    virtual void func1();  
    virtual void func2();  
};  
  
class Ex5: public Ex2, Ex4  
{  
    int var1;  
public:  
    void func1();  
    virtual void v_ex5();  
};
```

```
class Ex5      size(24):  
    +---  
    | +--- (base class Ex2)  
0     | | {vptr}  
4     | | var1  
| +---  
| +--- (base class Ex4)  
8     | | {vptr}  
12    | | var1  
16    | | var2  
| +---  
20    | var1  
+---
```

509

```
Ex5::$vftable@Ex2@:
```

0	&Ex2::get_sum
1	&Ex2::reset_values
2	&Ex5::v_ex5

```
Ex5::$vftable@Ex4@:
```

	-8
0	&Ex5::func1
1	&Ex4::func2

派生类将每个基类都嵌入了自身，而且每个基类还都保留有自己的虚函数表。但是请注意，第一个基类的虚函数表是被派生类共享的，派生类的虚函数将会被例在基类虚函数表的后面。另外要注意的是，因为 Ex5 中也有一个和 Ex4 的虚函数同名的 func1()，所以根据 C++ 的规则，Ex4 虚函数中的 func1() 函数的指针已经被 Ex5 的 func1() 的函数指针给替换掉了。

B. 识别类

我们上面已经讨论了如何判断一个程序是不是用 C++ 写的，讨论了类的构造函数以及内存中类的实例的组织形式，这一节我们来讨论 C++ 的类在可执行文件中的使用情况。我们先来讨论如何确定内存中哪些部分是类（或者称为对象）下一节再来讨论如何确定类之间的关系以及类中的成员。

1) 识别构造函数和析构函数

为了能从二进制可执行文件中把类识别出来，我们必须先要理解这些类的实例——对象是怎样被创建的。因为这个创建过程在汇编级别上具体是怎样实现的会给我们反汇编时如何识别这些类提供依据。

1) 全局对象。全局对象顾名思义就是那些被声明为全局变量的对象。这些对象的内存空间是在编译时就被分配好了的，它们位于可执行文件的数据段中。这些对象的构造函数是在这个程序启动之后，main() 函数被调用之前被调用执行的，而它们的析构函数则是在程序退出(exit) 时被调用的。

一般来讲，如果我们发现一个函数调用时，传入的 this 指针（一般是使用 ecx 寄存器）是指向一个全局变量的话，我们基本可以确定，这是一个全局对象，而要找到这个全局对象的构造函数和析构函数，我们一般要借助于交叉引用（cross-references）的功能。我们观察所有使用指向这个全局对象的函数的位置，如果某个函数位于程序的入口点（entry point）和 main() 函数之间，那么它就很有可能就是这个对象的构造函数。

PPT 里的图很说明问题：

这是源码：

```

class X
{
public:
    X() { printf("X::X()\n"); x1 = 0xff; }
    ~X() { printf("X::X()\n"); x1 = 0xaa; }
    int x1;
};

X xobj;

int main()
{
    xobj.x1 = 3;

    return 0;
}

```

这是反汇编以后的代码:

```

.data:00408000          ;org 408000h
.data:00408000 dword_408000 dd 0           ; DATA XREF: __cinit+1F o
.data:00408004             dd offset X_Ctor_Dtor_atexit
.data:00408008 unk_408008 db 0           ; DATA XREF: __cinit+1A o

.text:00401000 X_Ctor_Dtor_atexit proc near ; DATA XREF:
.text:00401000                 call  X_ctor
.text:00401005                 jmp   setup_at_exit
.text:00401005 X_Ctor_Dtor_atexit endp

.text:00401030
.text:00401030 setup_at_exit:
.text:00401030     push  offset X_dtor
.text:00401035     call   _atexit
.text:0040103A     pop   ecx
.text:0040103B     retn

```

2) 局部对象。同全局对象，局部对象就是被生命为局部变量的对象。这些对象的作用域起始于该对象被声明的地方，结束于声明该对象的模块退出之时（比如函数结尾或者分支结束的地方，下面例子里就是一个 if 语句块结束的地方调用析构函数的）。局部对象在内存中是位于栈（stack）里的。它们的构造函数在该对象声明的地方被调用，而在对象离开其作用域时调用对象的析构函数。

局部对象的构造函数还是比较容易识别的，如果你发现一个函数调用，传递过去的 this 指针竟然是指向了栈中一个未被初始化过的变量的话，你基本上可以确定这个函数是一个对象的构造函数，同时也就发现了一个对象。析构函数一般则是与构造函数位于同一个模块（也就是声明该对象的模块）的最后一个使用指向该对象的 this 指针的函数。

下面是一个简单的例子:

```

.text:00401060 sub_401060      proc near
.text:00401060
.text:00401060 var_C           = dword ptr -0Ch
.text:00401060 var_8           = dword ptr -8
.text:00401060 var_4           = dword ptr -4
.text:00401060
...(some code)...
.text:004010A4             add    esp, 8
.text:004010A7             cmp    [ebp+var_4], 5
.text:004010AB             jle    short loc_4010CE
.text:004010AB { < block begin
.text:004010AD             lea    ecx, [ebp+var_8] ; var_8 is uninitialized
.text:004010B0             call   sub_401000 ; constructor
.text:004010B5             mov    edx, [ebp+var_8]
.text:004010B8             push   edx
.text:004010B9             push   offset str->WithinIfX
.text:004010BE             call   sub_4010E4
.text:004010C3             add    esp, 8
.text:004010C6             lea    ecx, [ebp+var_8]
.text:004010C9             call   sub_401020 ; destructor
.text:004010CE } < block end
.text:004010CE
.text:004010CE loc_4010CE:          ; CODE XREF: sub_401060+4Bj
.text:004010CE             mov    [ebp+var_C], 0
.text:004010D5             lea    ecx, [ebp+var_4]
.text:004010D8             call   sub_401020

```

3) 动态分配的对象。这种对象是指哪些通过 new 操作符动态创建的对象。实际上 new 操作符会转变成两个函数调用：一个 new() 函数的调用再紧接着一个构造函数的调用。new() 函数是用来在堆中为对象分配空间的（对象的大小通过参数传递给 new() 函数），然后把新分配的地址放在 EAX 寄存器中返回出来。然后这个地址就被当作 this 指针传递给构造函数。同样 delete 操作符也会转变成两个函数调用，先调用析构函数，然后接着调用 free() 函数回收空间。

如下面这个简单的例子：

```

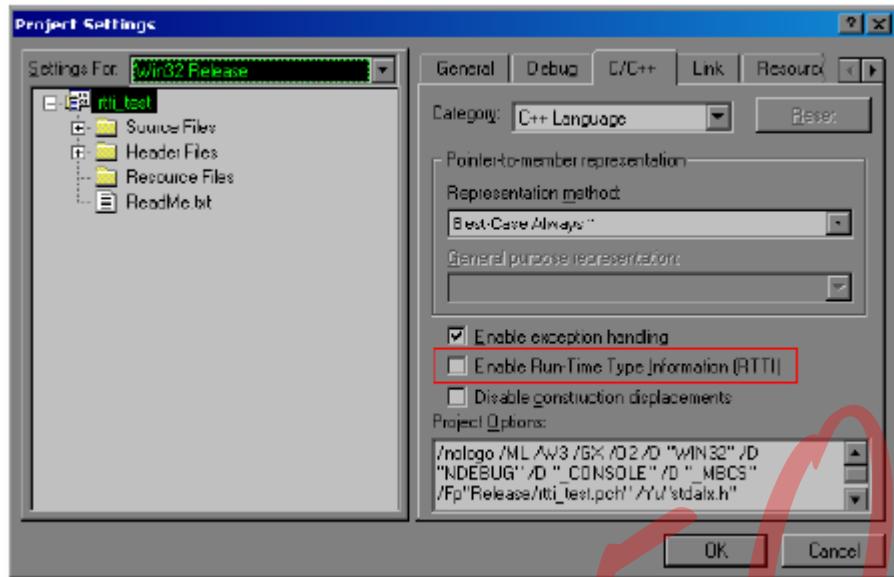
.text:0040103D _main      proc near
.text:0040103D argc        = dword ptr 8
.text:0040103D argv        = dword ptr 0Ch
.text:0040103D envp       = dword ptr 10h
.text:0040103D
.text:0040103D             push   esi
.text:0040103E             push   4
.text:00401040             call   ??2@YAPAXI@Z ; operator new(uint)
.text:00401045             test   eax, eax ; eax = address of allocated memory
.text:00401047             pop    ecx
.text:00401048             jz    short loc_401055
.text:0040104A             mov    ecx, eax
.text:0040104C             call   sub_401000 ; call to constructor
.text:00401051             mov    esi, eax
.text:00401053             jmp    short loc_401057
.text:00401055 loc_401055:          ; CODE XREF: _main+Bj
.text:00401055             xor    esi, esi ; CODE XREF: _main+16j
.text:00401057 loc_401057:          ; CODE XREF: _main+16j
.text:00401057             push   45h
.text:00401059             mov    ecx, esi
.text:0040105B             call   sub_401027
.text:00401060             test   esi, esi
.text:00401062             jz    short loc_401072
.text:00401064             mov    ecx, esi
.text:00401066             call   sub_40101B ; call to destructor
.text:00401068             push   esi ; void *
.text:0040106C             call   j__free ; call to free thunk function
.text:00401071             pop    ecx
.text:00401072 loc_401072:          ; CODE XREF: _main+25j
.text:00401072             xor    eax, eax
.text:00401074             pop    esi
.text:00401075             retn
.text:00401075 _main            endp

```

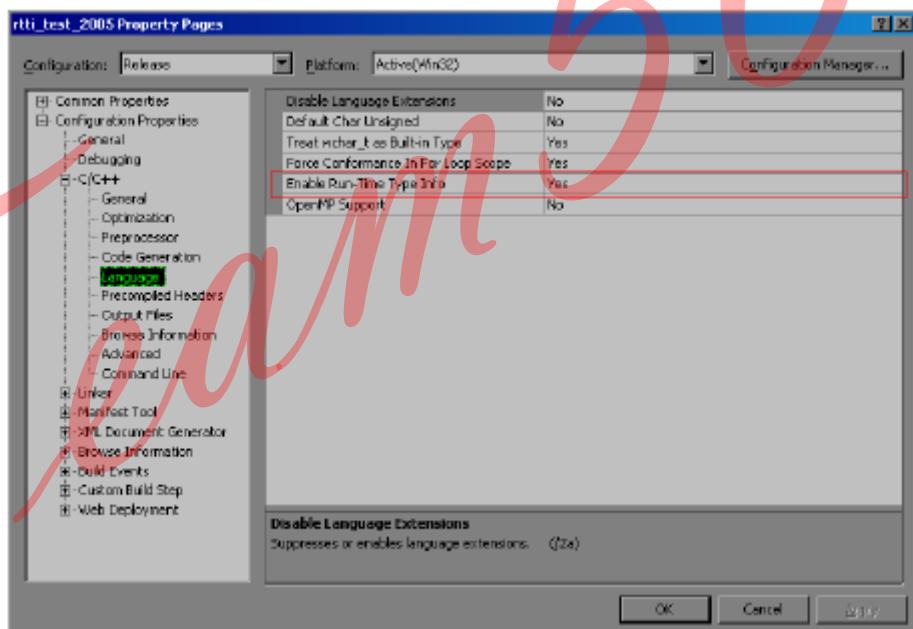
2) 利用 RTTI 识别多态类

如果 C++ 程序在编译时启用了 RTTI 功能，那么恭喜你！你又多了另一种识别类，特别是对多态类（即包含有虚函数的类），的方法——利用 RTTI（运行时类型信息 Run-time Type Information）。RTTI 是 C++ 中提供的一种在运行时确定对象的类型的机制。在 C++ 中我们一般使用 typeid 和 dynamic_cast

这两个操作符来实现这一机制。这两个操作符在实现时需要获得相关类的类名，类的层次等相关信息。在实际使用 VC 的过程中，如果你使用了 typeid 和 dynamic_cast 这两个操作符，却没有打开 RTTI 编译选项，编译器将会给你一个警告。在默认情况下 MSVC 6.0 是把 RTTI 给关闭掉的。



但是在 MSVC 2005 中，RTTI 默认是打开的。



为了实现 RTTI，编译器在编译完了的二进制可执行文件中加入一些结构体，这些结构体包含了代码中关于类（特别是多态类）的信息。这些结构体是：

1. RTTICompleteObjectLocator

这个结构体包含了 2 个指针，一个指向实际的类信息，另一个指向类的继承关系信息。

Offset	Type	Name	Description
0x00	DW	signature	Always 0?
0x04	DW	offset	Offset of vftable within the class
0x08	DW	cdOffset	?
0x0C	DW	pTypeDescriptor	Class Information
0x10	DW	pClassHierarchy Descriptor	Class Hierarchy Information

怎么找到这个 RTTICCompleteObjectLocator 结构体呢？我们先找虚函数表，在内存中虚函数表上面一个 DWORD 就是指向 RTTICCompleteObjectLocator 结构体的指针，不信？请看下面这两个例子，您上眼：

```
.rdata:00404128          dd offset ClassA_RTTICompleteObjectLocator
.rdata:0040412C ClassA_vftable dd offset sub_401000 ; DATA XREF:...
.rdata:00404130          dd offset sub_401050
.rdata:00404134          dd offset sub_4010C0
.rdata:00404138          dd offset ClassB_RTTICompleteObjectLocator
.rdata:0040413C ClassB_vftable dd offset sub_4012B0 ; DATA XREF:...
.rdata:00404140          dd offset sub_401300
.rdata:00404144          dd offset sub_4010C0
```

下面给出的是一个 RTTICCompleteObjectLocator 结构体的实例：

```
.rdata:004045A4 ClassB_RTTICompleteObjectLocator
    dd 0      ; COL.signature
.rdata:004045A8      dd 0      ; COL.offset
.rdata:004045AC      dd 0      ; COL.cdOffset
.rdata:004045B0      dd offset ClassB_TypeDescriptor
.rdata:004045B4      dd offset ClassB_RTTIClassHierarchyDescriptor
```

2. TypeDescriptor

您想必已经看见了，在 RTTICCompleteObjectLocator 结构体中，第四个 DWORD 域里是一个指向本类的 TypeDescriptor 结构体的指针。TypeDescriptor 这个结构体中记录了这个类的类名，我们逆向的时候一般可以根据类名大致猜出这个类是干什么的，这个结构体的结构如下图：

Offset	Type	Name	Description
0x00	DW	pVFTable	Always points to type_info's vftable
0x04	DW	spare	?
0x08	SZ	name	Class Name

下面是 TypeDescriptor 的一个实例：

```
.data:0041A098 ClassA TypeDescriptor ; DATA XREF: ....
    dd offset type_info_vftable   ; TypeDescriptor.pVFTable
.data:0041A09C      dd 0      ; TypeDescriptor.spare
.data:0041A0A0      db '.?AVClassA@0',0           ; TypeDescriptor.name
```

3. RTTIClassHierarchyDescriptor

RTTIClassHierarchyDescriptor 记录了类的继承信息，包括基类的数量，以及一个 RTTIBaseClassDescriptor 数组，RTTIBaseClassDescriptor 我们下面详细讨论，现在我只先说一点，就是 RTTIBaseClassDescriptor 最终将指向当前各个基类的 TypeDescriptor。

Offset	Type	Name	Description
0x00	DW	signature	Always 0?
0x04	DW	attributes	Bit 0 - multiple inheritance Bit 1 - virtual inheritance
0x08	DW	numBaseClasses	Number of base classes. Count includes the class itself
0x0C	DW	pBaseClassArray	Array of RTTIBaseClassDescriptor

比如说我们声明了一个类 ClassG，它虚继承了类 ClassA 和 ClassE：

```
class ClassA {...}  
class ClassE {...}  
class ClassG: public virtual ClassA, public virtual ClassE {...}
```

那么 ClassG 的 RTTIClassHierarchyDescriptor 就应该是下面这个样子的：

```
.rdata:004178C8 ClassG_RTTIClassHierarchyDescriptor ; DATA XREF: ...  
.rdata:004178C8          dd 0                      ; signature  
.rdata:004178CC          dd 3                      ; attributes  
.rdata:004178D0          dd 3                      ; numBaseClasses  
.rdata:004178D4          dd offset ClassG_pBaseClassArray ; pBaseClassArray  
.rdata:004178D8 ClassG_pBaseClassArray  
                      dd offset oop_re$RTTIBaseClassDescriptor@4178e8  
.rdata:004178DC          dd offset oop_re$RTTIBaseClassDescriptor@417904  
.rdata:004178E0          dd offset oop_re$RTTIBaseClassDescriptor@417920
```

它里面有 3 个基类（包括了 ClassG 本身），attribute 是 3 表示这个类是多继承加上虚继承。最后有一个 pBaseClassArrary 指针指向 RTTIBaseClassDescriptor 指针数组。

4. RTTIBaseClassDescriptor

这个结构体包含了关于基类的有关信息。它包括一个指向基类的 TypeDescriptor 的指针和一个指向基类的 RTTIClassHierarchyDescriptor 的指针，（译注：在 VC6.0 编译的结果中可能没有 pClassDescriptor）另外它还包含有一个 PMD 结构体，该结构体中记录了该类中各个基类的位置。RTTIBaseClassDescriptor 的结构如下图所示：

Offset	Type	Name	Description
0x00	DW	pTypeDescriptor	TypeDescriptor of this base class
0x04	DW	numContainedBases	Number of direct bases of this base class
0x08	DW	PMD.mdisp	vftable offset
0x0C	DW	PMD.pdisp	vtable offset (-1: vtable is at displacement PMD.mdisp inside the class)
0x10	DW	PMD.vdisp	Displacement of the base class vtable pointer inside the vtable
0x14	DW	attributes	?
0x18	DW	pClassDescriptor	RTTI Class Hierarchy Descriptor of this base class

虚基类表 (virtual base class table, vtable) 只会在多重虚继承的情况下才会出现。因为在多重虚继承的情况下，有时会需要 upclass, (译注：比如这个 ClassG 这个例子中 ClassA 和 ClassE 都继承自 ClassX，《掀起你的盖头来——谈 VC++ 对象模型》一文中第五节虚继承中讲的比较细，我懒一下直接引用了，呵呵，http://dev.yesky.com/136/2317136_1.shtml) 这时就需要精确定位基类。虚基类表包含了各个基类在派生类中的位置 (或者也可以说是各个基类的虚函数表在派生类中的位置，因为虚函数表是位于类的起始位置的)。

比如我们这个 ClassG，它在内存中是这个样子的：

```
class ClassG           size(28):
+---+
0   | {vfptr}
4   | {vbptr}
+---+
+--- (virtual base ClassA)
8   | {vfptr}
12  | class_a_var01
16  | class_a_var02
| <alignment member> (size=3)
+---+
+--- (virtual base ClassE)
20  | {vfptr}
24  | class_e_var01
+---+
```

虚函数表的指针被放在整个类偏移 +4 这个位置上，而虚基类表中则记录了各个基类在派生类中的位置：

ClassG::\$vtable@:
0 -4
1 4 (ClassGd(ClassG+4) ClassA)
2 16 (ClassGd(ClassG+4) ClassE)

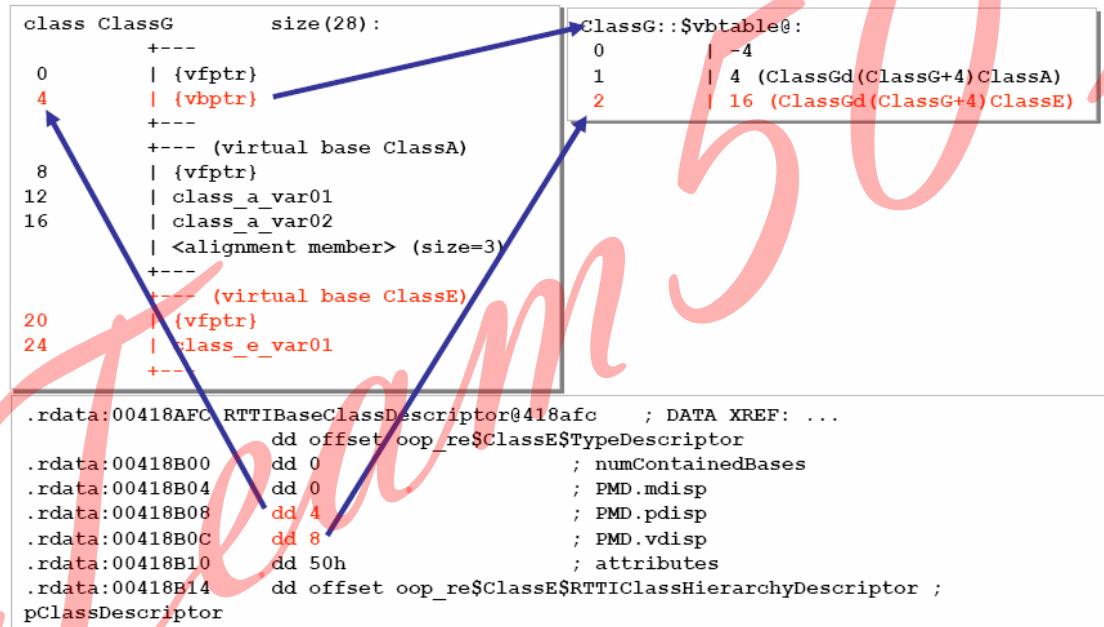
我们现在来试试通过虚基类表来确定 ClassE 在 ClassG 中的位置，我们先要知道虚基类表的偏移，嗯，它是 4，然后我们从虚基类表中读出 ClassE 的偏移，嗯，它是 16， $16+4=20$ ，所以 ClassE 在 ClassG 中位于偏移+20 这个位置上。

下面是 ClassG 中 ClassE 的 BaseClassDescriptor:

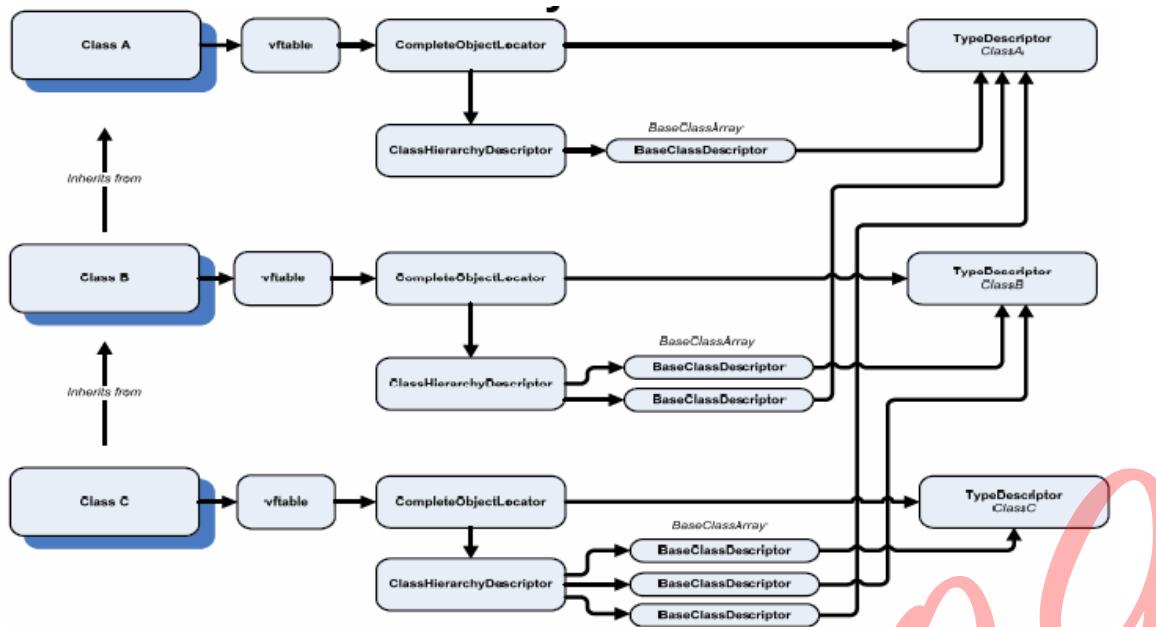
```
.rdata:00418AFC RTTIBaseClassDescriptor@418afc ; DATA XREF: ...
    dd offset oop_re$ClassE$typeDescriptor
.rdata:00418B00 dd 0 ; numContainedBases
.rdata:00418B04 dd 0 ; PMD.mdisp
.rdata:00418B08 dd 4 ; PMD.pdisp
.rdata:00418B0C dd 8 ; PMD.vdisp
.rdata:00418B10 dd 50h ; attributes
.rdata:00418B14 dd offset oop_re$ClassE$RTTIClassHierarchyDescriptor ;
pClassDescriptor
```

我们看到 PMD.pdisp 是 4，这个域表示的是 vtable（虚函数表）在 ClassG 中的偏移量，而 PMD.vdisp 是 8，表示 ClassE 在 ClassG 中的偏移量，是记录在虚函数表偏移+8 的位置上的。（也就是第三个 DWORD 域中）。

PPT 里的图实在是清楚啊：



下面这个图是对这一节的一个总结：



C. 判别类与类之间的关系

1. 通过分析构造函数来分析类与类之间的关系

构造函数是用来初始化对象的（好像是废话啊，呵呵），所以在构造函数中，它会调用基类的构造函数（如果有的话）以及设置自己的虚函数表。因此分析类的构造函数是我们分析类与类之间关系的一个很好的突破口。

下面是一个简单单继承的例子：

```
.text:00401010 ClassB_Ctor      proc near
.text:00401010
.text:00401010 var_4           = dword ptr -4
.text:00401010
.text:00401010
.text:00401011 push    ebp
.text:00401011 mov     ebp, esp
.text:00401012 push    ecx
.text:00401012 mov     [ebp+var_4], ecx ; get this ptr to current object
.text:00401013 mov     ecx, [ebp+var_4]
.text:00401014 call    sub_401000 ; call class A constructor
.text:00401015 mov     eax, [ebp+var_4]
.text:00401016 mov     esp, ebp
.text:00401017 pop    ebx
.text:00401018 ret
.text:00401022 ClassB_Ctor      endp
```

假定我们已经知道上面这段代码是某个类的构造函数，我们发现红颜色标出的这个函数使用了一个由 ecx 传递进来的一个当前对象的 this 指针。问题来了：这个函数究竟是当前对象的一个成员函数呢，还是当前对象的基类的构造函数呢？

对不起，我不能 100% 的确定。当然在实际的逆向工程里，很有可能这就是一个基类的构造函数。当然有时我们干脆事先就已经知道了这个函数是另一个类的构造函数，问题也就迎刃而解了。

接下来说正事，如果我们发现类 A 的构造函数在类 B 的构造函数中出现，而且还将当前对象（类 B）的指针当成（类 A 的）this 指针来使用的话，我们基本上就可以确定，这是类 A 是类 B 的基类。

在进行人工判别时，我们应该多多利用交叉引用（cross-references）功能，看看红颜色标出的这个函数有没有被其他类当成构造函数使用。自动判别的有关技巧我们稍后再进行讨论。

下面我们再来看一个复杂一点的多继承的情况:

```
.text:00401020 ClassC_Ctor      proc near
.text:00401020
.text:00401020 var_4           = dword ptr -4
.text:00401020
.text:00401020                 push    ebp
.text:00401021                 mov     ebp, esp
.text:00401023                 push    ecx
.text:00401024                 mov     [ebp+var_4], ecx
.text:00401027                 mov     ecx, [ebp+var_4] ; ptr to base class A
.text:0040102A                 call    sub_401000 ; call class A constructor
.text:0040102A
.text:0040102F                 mov     ecx, [ebp+var_4]
.text:00401032                 add     ecx, 4 ; ptr to base class B
.text:00401035                 call    sub_401010 ; call class B constructor
.text:00401035
.text:0040103A                 mov     eax, [ebp+var_4]
.text:0040103D                 mov     esp, ebp
.text:0040103F                 pop    ebp
.text:00401040                 retn
.text:00401040 ClassC_Ctor      endp
```

一开始还是和刚才那个单继承的情况一样，先有一个函数调用，使用了ecx把当前对象的指针当成this指针传给了这个函数。嗯，然后好像就有点不一样了，我们注意到当前对象的指针被加上4，然后又被当成另一个函数的this指针……呵呵，显然第二个函数是另一个基类的构造函数。

我们现在对这个类做一点解释，让你能比较直观的理解这一小节。上面这段代码是类D的构造函数，类D继承了类A和C，这三个类在内存中的布局如下：

```
class A size(4):
    +---+
    0 | a1
    +---+
class C size(4):
    +---+
    0 | c1
    +---+
class D size(12):
    +---+
    | +--- (base class A)
    0 | | a1
    | +---+
    | +--- (base class C)
    4 | | c1
    | +---+
    8 | | d1
    +---+
```

我们现在知道了各个基类的构造函数所使用的this指针是怎么来的了。基类的this指针是与派生类的this指针息息相关的，具体说，就是类A和C的this指针是类D的this指针加上类A和C各自在类D中的偏移量得出的。

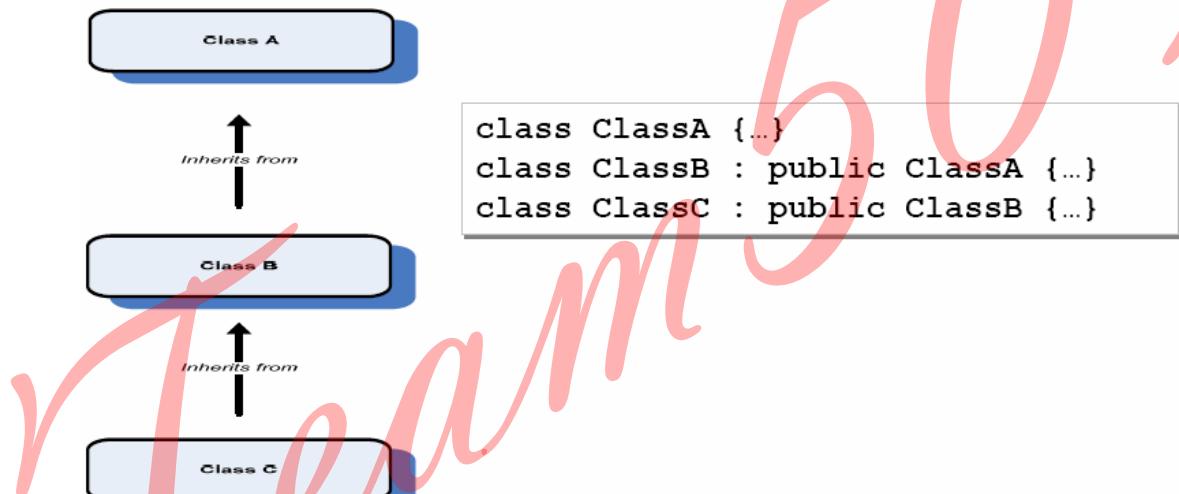
2. 通过RTTI分析类与类之间的关系

利用RTTI识别类我们在前面已经讨论过了，现在我们来讨论怎样利用RTTI来判别类与类之间的关系。现在我们要利用RTTIClassHierarchyDescriptor这个结构体。为了便于大家参考，我把这个结构体的结构在贴一遍：

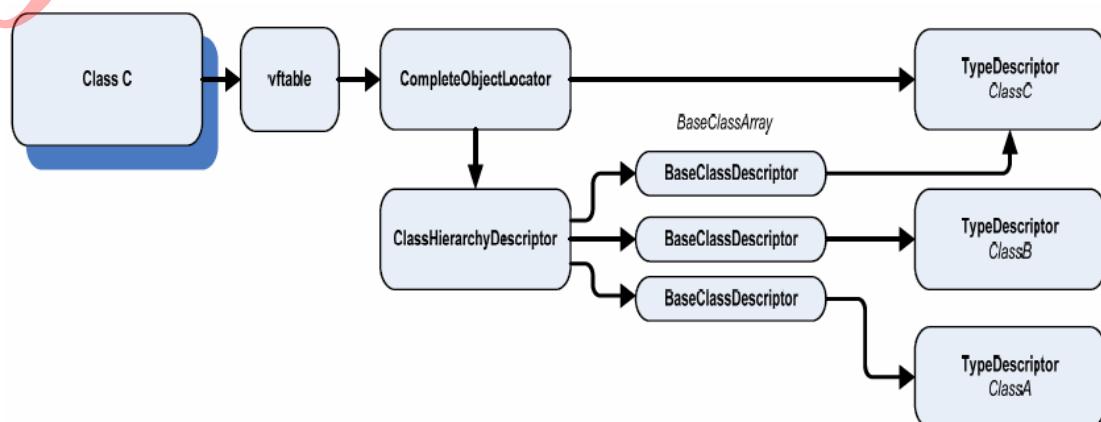
Offset	Type	Name	Description
0x00	DW	signature	Always 0?
0x04	DW	attributes	Bit 0 - multiple inheritance Bit 1 - virtual inheritance
0x08	DW	numBaseClasses	Number of base classes. Count includes the class itself
0x0C	DW	pBaseClassArray	Array of RTTIBaseClassDescriptor

我们现在注意最后一个域——pBaseClassArray。这个域里是一个指向RTTIBaseClassDescriptor(BCD)组成的数组的指针。而数组中各个BCD则是指向当前类的各个基类的TypeDescriptor结构体的指针(关于这一点我们前面已经讨论过了)。

比如下面这个例子:



下面是根据类 C 的 RTTIClassHierarchyDescriptor , RTTIBaseClassDescriptor 以及相关基类的 TypeDescriptor 画出的，A、B、C 三个类之间的关系：



仔细的看官可能已经发现一点问题了，在 pBaseClassArray 指向的 BaseClassArray 数组中，甚至列出了类 C 的非直接基类——A。这样以来类 A 和 B 之间的关系就比较模糊了。当然你可以再去分析类 B 的 RTTIClassHierarchyDescriptor，然后你就能知道类 A 实际上是类 B 的基类。所以类 A 就不可能再是类 C 的基类了。这样你就能正确推出 A、B、C 三个类的关系了。

D. 辨别类的成员

辨别类的成员的这一过程虽然有点枯燥乏味，但是相对而言技术难度却小的多。一般访问类的成员（读或者写）一般都会使用 this 指针加上该成员在类中的偏移量的方式实现。所以我们也利用这一特点来辨别类的成员，如下面这个例子：

```
.text:00401003      push    ecx
.text:00401004      mov     [ebp+var_4], ecx ; ecx = this pointer
.text:00401007      mov     eax, [ebp+var_4]
.text:0040100A      mov     dword ptr [eax + 8], 12345h ; write to 3rd member
                                         ; variable
```

一般调用虚函数都是使用读虚函数表中的偏移，然后进行间接调用的方式实现的，我们也利用这一特点来辨别类的虚函数，比如下面这个例子：

```
.text:00401C21      mov     ecx, [ebp+var_1C] ; ecx = this pointer
.text:00401C24      mov     edx, [ecx] ; edx = ptr to vftable
.text:00401C26      mov     ecx, [ebp+var_1C]
.text:00401C29      mov     eax, [edx+4] ; eax = address of 2nd virtual
                                         ; function in vftable
.text:00401C2C      call    eax ; call virtual function
```

那么类的非虚函数怎么来识别呢？我们可以利用 this 指针来做到这一点，一般 this 指针要通过 ecx 寄存器来传递给函数，比如下面这个例子：

```
.text:00401AFC      push    0CCh
.text:00401B01      lea     ecx, [ebp+var_C] ; ecx = this pointer
.text:00401B04      call    sub_401110
```

当然如果你觉得证据还不充分，你还可以进一步检查在被调用的函数中是不是没有初始化就直接使用了 ecx 寄存器，我们来具体看看 sub_401110 这个函数的实现代码：

```
.text:00401110      push    ebp
.text:00401111      mov     ebp, esp
.text:00401113      push    ecx
.text:00401114      mov     [ebp+var_4], ecx ; uninitialized ecx
used
```

III. 自动化

这一节我来介绍我们自动化类的识别过程的方法。我们介绍我们开发的一个工具，并告诉大家这个工具是这样实现的。

A. OOP-RE

OOP-RE 我们内部开发的一个自动化 C++ 对象识别的工具的名字。它能够从一个二进制可执行文件中识别出其中的各个类（如果 RTTI 打开的话甚至能得到类的名字），类与类之间的关系以及类的成员。总之就是上面讨论的所有内容。运行这个工具之后，它会用注释的方式把识别出来的东东在反汇编结果中标识出来。OOP-RE 使用 Ruby 语言写成的，它使用了 Sabre 的 Ida2sql。



B. 为什么选择静态分析的方式？

在 OOP-RE 开发之初我们就考虑过到底应该把它开发成一个动态分析工具还是一个静态分析工具。最终我们选择了把它开发成一个静态分析工具。为什么呢？因为有些系统，比如 Symbian，大量使用 C++，但是在这些系统上进行动态分析却十分困难——如果这个工具以后要考虑分析 Symbian 的应用程序的话。当然，不管怎么说，一个兼备了动态分析和静态分析的方案可能会更好，它应该可以产生更加准确的分析结果。

C. 自动化分析的策略

通用算法

Pointer Flow Graphs

我们使用了一些自动化分析技术要求我们要能够对一些寄存器或者是变量中的值进行跟踪，为了做到这一点，我们就必须要进行一个恰当的数据流分析。不过就像很多前辈已经证明了的，数据流分析是一个很难解决的问题。不过还算好，我们面对的都是一些特定条件下的简单的数据流分析，所以问题会比较简单。我们的数据流分析只要能够对几个特定的寄存器或者指针进行分析追踪就可以了。

我下面解释一下几个术语：

Block

正如很多研究数据流分析 (data flow analyzer) 的大牛已经证明过的那样，对一大段指令进行数据流分析是一个很难完成的任务，所以我们在设计 OOP-RE 的时候，一开始就不准备对大段的反汇编程序进行数据流分析的想法，代之而来的是我们分析一个一个的小块 (block)。这些小块的起始位置一般由一些特定的分析算法决定（比如寻找构造函数时我们可能会先寻找 new 函数，找到以后 new 函数下面一条指令就成了一个 block 的起点），block 的终点由以下几个条件决定：

- 1) 如果我们要分析的变量或者寄存器中的值被其他变量覆盖了，block 就随之结束，因为继续分析已经毫无意义了。
- 2) 如果我们分析的是 eax 寄存器，那么如果遇到一个函数调用（比如：call 指令），block 就随之结束。（因为我们假定所有的函数返回值都是通过 eax 寄存器返回的）
- 3) 如果遇到一个函数调用，下一条指令就会变成一个新的 block 的起点。

4) 如果遇到一个条件分支(比如一个 if 语句块),而且我们要跟踪的变量或者寄存器在两个分支里都会被使用,我们把每个分支都当成一个新的 block。

5) 如果我们要分析的变量或者寄存器的值被复制到了另一个变量中,我们就开始一个新的 block,在新的 block 中同时跟踪新的和旧的变量。

Pointer Flow Graphs

如果我们事先指定了一个要跟踪的寄存器或者变量,并且给出了一个 block 的起始位置,开始进行分析。我们就说我们构建了一个指针流图(Pointer Flow Graphs)。

识别成员函数

我们判断一个函数是不是类的成员函数的方法是:

- 1) 从头开始,逐一检查函数反汇编代码中的每一条指令。
- 2) 如果先遇到一条读 ecx 寄存器的指令,那么这个函数就很有可能是一个类的成员函数
- 3) 如果先遇到一条写 ecx 寄存器的指令,那么这个函数就只是一个普通的函数
- 4) 如果这个函数中就根本没有使用过 ecx 寄存器(不管是读是写),那么这个函数就只是一个普通的函数

1. 利用 RTTI 识别多态类

OOP-RE 利用 RTTI 可以获取下列信息:

- 1) 各个多态类
- 2) 多态类的类名
- 3) 各个多态类之间的继承关系
- 4) 各个多态类的虚函数和虚函数表
- 5) 各个多态类的构造/析构函数

为了找到 RTTI 相关的结构体,OOP-RE 先要去定位虚函数表。这是因为 RTTICreateObjectLocator 的指针就在虚函数表上面的一个 DWORD 中。为了找到虚函数表,OOP-RE 执行下面列出的这个检查:

- 1) 检查是不是一个 DWORD
- 2) 检查这个 DWORD 是不是一个指向代码的指针
- 3) 检查这个 DWORD 是不是被其他代码引用过,如果有,引用这个 DWORD 的指令是不是一条 MOV 指令(假设这条指令是用来分配虚函数表的)

一旦找到了虚函数表,OOP-RE 就开始检查虚函数表上面那个 DWORD 是不是一个 RTTICreateObjectLocator 的指针。这一点是通过分析 RTTICreateObjectLocator 并 检查

RTTICreateObjectLocator.pTypeDescriptor 是不是指向一个有效的 TypeDescriptor 来确定的。检查 TypeDescriptor 是否正确的一个办法是检查 TypeDescriptor.name 是不是一个“.?AV”开头的字符串。“.?AV”好像是 VC 给类名加上的前缀(译注:“.?AV”似乎应该是一个 DWORD 是 TypeDescriptor.Spare)。

下面是一个例子,我们在 004165B4 处找到一个虚函数表:

```
.rdata:004165B0          dd offset ClassB_RTTICompleteObjectLocator@00
.rdata:004165B4 ClassB_vftable
.rdata:004165B4          dd offset sub_401410 ; DATA XREF:...
.rdata:004165B8          dd offset sub_401460
.rdata:004165BC          dd offset sub_401230
```

OOP-RE 接下来就会去检查 RTTICompleteObjectLocator 是否有效。我们看到 OOP-RE 是去读 RTTICompleteObjectLocator 中指向的 TypeDescriptor 的指针：

```
.rdata:00418A28 ClassB_RTTICompleteObjectLocator@00
.rdata:00418A28          dd 0           ; signature
.rdata:00418A2C          dd 0           ; offset
.rdata:00418A30          dd 0           ; cdOffset
.rdata:00418A34          dd offset ClassB_TypeDescriptor
.rdata:00418A38          dd offset ClassB_RTTIClassHierarchyDescriptor
```

如果 TypeDescriptor 的 name 域指向的字符串是以 “.?AV” 开头的，那 RTTICompleteObjectLocator 就是有效的，否则 RTTICompleteObjectLocator 就是无效的。

```
.data:0041B01C ClassB_TypeDescriptor
                  dd offset type_info_vftable
.data:0041B020          dd 0           ;spare
.data:0041B024 a_?avclassb@@      db '.?AVClassB@@',0 ; name
```

如果上面这个分析能够通过的话，OOP-RE 就会分析 RTTICompleteObjectLocator 中记录的所有 RTTI 相关的结构体，并且创建一个新的类，类名就是 TypeDescriptor.name 中记录的那个。下面列出的是所有利用 RTTI 能够提取的信息：

发现新的类

——通过 TypeDescriptor 识别

找出新的类的名字

——就是 TypeDescriptor.name

找出新的类的虚函数表以及各个虚函数

——通过 RTTICompleteObjectLocator 与虚函数表之间的关系

找出新的类的构造函数和析构函数

——通过分析设置虚函数表的函数（应该就是那条 MOV 指令）

找出新的类的所有基类

——通过分析 RTTICompleteObjectLocator.pClassHierarchyDescriptor

2. 利用虚函数表识别多态类（不使用 RTTI）

如果被分析的二进制可执行文件在编译时没有使用 RTTI，我们还可以利用搜索虚函数表的方法来识别多态类（具体方法我们在 C.2 中已经详细讨论过了）。这样我们可以：

发现新的类

——通过 TypeDescriptor 识别

找出新的类的名字

——自动根据虚函数表所在的地址给类命名

找出新的类的虚函数表以及各个虚函数

——通过虚函数表获得

找出新的类的构造函数和析构函数

——通过分析设置虚函数表的函数（应该就是那条 MOV 指令）

注意这时，多态类的基类还是没办法识别，但是我们还是可以通过分析各个类的构造函数来分析各个类之间的继承关系，这一点我们下面详细讨论。

3. 通过搜索构造/析构函数来识别类

程序中，如果程序员在程序中使用了 new 函数在运行时动态生成某些类的对象，对于这些类来说，我们还可以通过下列算法进行识别：

- 1) 寻找 new 函数（比如：“j_??2@YAPAXI@Z”、“??2@ YAPAXI@Z”或者“operator_new”）
- 2) 以 new 函数之后的第一条指令为起点，对 new 函数的返回值（在 eax 中）进行追踪分析，看看 eax 的值什么时候传给了 ecx。
- 3) 然后我们在这个 block 中寻找第一个函数调用，然后跟入这个函数看看它是不是未初始化就使用了 ecx 寄存器。
- 4) 如果这是一个类函数（未初始化就使用了 ecx 寄存器），那么这个函数很有可能就是类的构造函数。

对于局部对象，我们也可以找出它的构造函数，算法如下：

Team 509

- 1) 寻找类似“lea ecx, [XXX]”的指令，这里 XXX 应该是当前函数栈中的一个内存地址。
- 2) 检查一下函数中“lea ecx, [XXX]”指令之前执行的指令中有没有向 XXX 地址上写一些的值的指令，如果有，则把找到这条“lea ecx, [XXX]”指令忽略掉。
- 3) 从“lea ecx, [XXX]”的下一条指令开始创建一个 pointer flow graph 跟踪 ecx。
- 4) 然后我们在这个 block 中寻找第一个函数调用，然后跟入这个函数看看它是不是未初始化就使用了 ecx 寄存器。
- 5) 如果这是一个类函数（未初始化就使用了 ecx 寄存器），那么这个函数很有可能就是类的构造函数。

(译注：下面这段原文实在是语意不清，译者按自己的理解来翻)这样识别有问题吗？嗯，有一点。比如有些函数使用默认的构造函数，不初始化任何成员变量。这时，VC 就不提供一个专门的构造函数了，而是代之以一条“mov dword ptr [eax], offset vtable”的指令。这样 new 之后的第一个被调用的函数，就不是构造函数了。比如下面这段程序

```
#include <iostream>

using namespace std;

class ClassA
{
    int i;
public :
    virtual void func();
};

void ClassA::func(){
    cout << "this is A" << endl;
    return ;
}

int main(){
    ClassA * p = new ClassA;
    p->func();
    return 0;
}
```

用 IDA 反汇编这段程序，请注意 main 函数：

```
.text:00401070 ; **** S U B R O U T I N E ****
.text:00401070
.text:00401070
.text:00401070 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401070 _main proc near ; CODE XREF: start+AF↓p
.text:00401070     push    8
.text:00401072     call    operator new(uint)
.text:00401072
.text:00401077     add     esp, 4      ; 平栈
.text:0040107A     test    eax, eax
.text:0040107C     jz      short loc_40108D ; 检查new函数是否成功分配了段内存
.text:0040107C             ; new函数如果返回0, 说明new失败, 跳转进行异常处理
.text:0040107C
.text:0040107E     mov     dword ptr [eax], offset OFF_40B0D0 ; 这是构造函数
.text:0040107E             ; 仅仅设置虚函数表
.text:0040107E             ; 不初始化任何成员变量
.text:00401084     mov     edx, [eax]
.text:00401086     mov     ecx, eax
.text:00401088     call    dword ptr [edx]
.text:0040108A     xor     eax, eax
.text:0040108C     retn
```

你看，如果按上面的识别算法，函数 func() 就会被当成 ClassA 的构造函数了。如果你把源码中的 virtual 去掉，连 “mov dword ptr [eax], offset vtable” 这条指令都会被省略掉，更容易把 func() 当成 ClassA 的构造函数：

```
.text:00401070 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401070 _main proc near ; CODE XREF: start+AF↓p
.text:00401070     push    4
.text:00401072     call    operator new(uint)
.text:00401072
.text:00401077     add     esp, 4
.text:0040107A     mov     ecx, eax
.text:0040107C     call    sub_401000 ; 这是函数func
.text:0040107C             ; 很容易当成是ClassA的构造函数吧：)
.text:0040107C
.text:0040107C     xor     eax, eax
.text:00401081     retn
.text:00401083
.text:00401083 _main endp
```

如果 ClassA 的成员函数多一点，每次 new ClassA 之后马上调用的成员函数又是不同的，这样 ClassA 就会有多个“构造函数”。比如下面这个例子：

```

#include <iostream>

using namespace std;

class ClassA
{
    int i;
public :
    void func1();
    void func2();
};

void ClassA::func1(){
    cout << "this is A.func1" << endl;
    return ;
}

void ClassA::func2(){
    cout << "this is A.func2" << endl;
    return ;
}

int main(){
    ClassA * p = new ClassA;
    p->func1();
    p = new ClassA;
    p->func2();
    return 0;
}

```

IDA 反汇编的结果:

```

.text:004010E0 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:004010E0 _main proc near                           ; CODE XREF: start+AF↓p
.text:004010E0
.text:004010E2
.text:004010E2
.text:004010E7
.text:004010EA
.text:004010EC
.text:004010EC
.text:004010EC
.text:004010F1
.text:004010F3
.text:004010F3
.text:004010F8
.text:004010FB
.text:004010FD
.text:004010FD
.text:00401102
.text:00401104
.text:00401104 _main    endp
.text:004010E0
.text:004010E2
.text:004010E2
.text:004010E7
.text:004010EA
.text:004010EC
.text:004010EC
.text:004010EC
.text:004010F1
.text:004010F3
.text:004010F3
.text:004010F8
.text:004010FB
.text:004010FD
.text:004010FD
.text:00401102
.text:00401104
.text:00401104 _main    endp

```

push 4
call operator new(uint)
add esp, 4
mov ecx, eax
call sub_401000 ; 这是 Func1
; 很容易被当成一个构造函数

push 4
call operator new(uint)
add esp, 4
mov ecx, eax
call sub_401070 | ; 这是 Func2
; 也很容易被当成一个构造函数

xor eax, eax
ret

为了避免出现这种情况，OOP-RE 会检查传给 new 函数的对象的大小，如果传给两个 new 函数的对象的大小是一样的话，OOP-RE 就会认为这是在 new 同一个类的对象。这样它就会去检查这些“构造函数”是不是类的成员函数。（不过这样的话，如果 2 个类的大小恰好一样大的话……）

4. 识别类与类之间的继承关系

就像我们之前讨论过的那样，类与类之间的继承关系可以通过分析构造函数

来获知(派生类的构造函数会调用基类的构造函数)。OOP-RE 中我们通过跟踪构造函数的当前对象的 this 指针的使用来自动实现这一分析过程。在单继承时, this 指针会被传递给构造函数中调用的基类的构造函数;而在多基层时, this 指针会被加上其他基类在该派生类中的偏移量, 传递给相关基类的构造函数。OOP-RE 完全能够应付这些情况。

5. 类的成员的识别

1) 类的成员变量的识别

因为程序中访问类的成员是靠 this 指针加上成员变量在类中的偏移量的方式实现的, 所以 OOP-RE 会跟踪 this 指针的使用, 列出所有可能的成员变量。

2) 类的非虚函数的识别

OOP-RE 通过跟踪分析指向当前对象的 this 指针(一般就是 ECX 寄存器)的使用情况来做到这一点。

3) 类的虚函数的识别

通过分析虚函数表就能得到。

做完了上述工作之后, 我们应该能够重构出目标代码中的类的基本情况了。

D. 显示结果

1. 注释各种结构体

一旦完成了上一节的工作, OOP-RE 将会生成一个 idc 文件, 使用 IDB 就可以用这个文件, 把分析结果以注释的形式显示在 IDA 的反汇编结果中。

对于 RTTI 相关的数据, OOP-RE 会重新定义有关结构体, 并且把结构体中各个成员的名字都注释出来。

比如下面这张图是 OOP-RE 没运行之前, IDA 分析的结果:

```
<Original>
.rdata:004165A0 dd offset unk_4189E0
.rdata:004165A4 off_4165A4 dd offset sub_401170 ; DATA XREF:...
.rdata:004165A8 dd offset sub_4011C0
.rdata:004165AC dd offset sub_401230
.rdata:004165B0 dd offset unk_418A28
```

OOP-RE 运行之后:

```
<Processed>
.rdata:004165A0 dd offset oop_re$ClassA$RTTICompleteObjectLocator@00
.rdata:004165A4 oop_re$ClassA$vftable@00
.rdata:004165A8 dd offset sub_401170 ; DATA XREF: ...
.rdata:004165AC dd offset sub_4011C0
```

下面是一个 RTTICompleteObjectLocator 结构体的实例, OOP-RE 没运行之前:

```
<Original>
.rdata:004189E0 dword_4189E0      dd 0          ; DATA XREF:...
.rdata:004189E4                  dd 0
.rdata:004189E8                  dd 0
.rdata:004189EC                  dd offset off_41B004
.rdata:004189F0                  dd offset unk_4189F4
```

OOP-RE 运行之后：

```
<Processed>
.rdata:004189E0 oop_re$ClassA$RTTICompleteObjectLocator@00
              dd 0          ; RTTICompleteObjectLocator.signature
.rdata:004189E4      dd 0          ; RTTICompleteObjectLocator.offset
.rdata:004189E8      dd 0          ; RTTICompleteObjectLocator.cdOffset
.rdata:004189EC      dd offset oop_re$ClassA$TypeDescriptor
.rdata:004189F0      dd offset oop_re$ClassA$RTTIClassHierarchyDescriptor
```

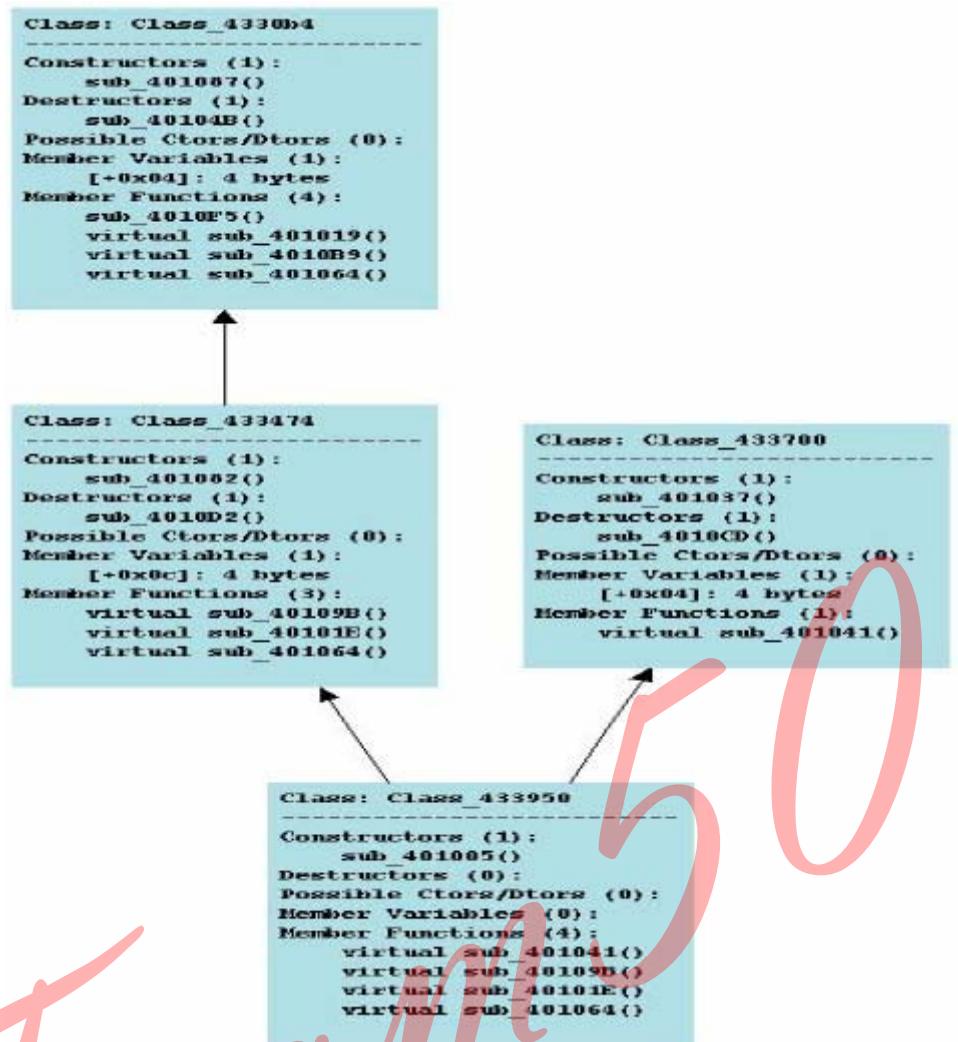
2. 改进过的调用图表

OOP-RE 的分析结果是以注释的形式添加到 IDA 里的，当然，程序代码对各个类的虚函数的调用也能在 IDA 里以交叉引用的方式表示出来。当然，也就能生成一个更好的调用图标，使之能用于 BinDiff 和 DarunGrim 之类的二进制比较工具。定位到的虚函数表也能用于有关二进制比较技术（见 Rafal Wojtczuk 的博客 <http://www.avertlabs.com/research/blog/?p=17>）

E. 分析结果可视化：UML 图

OOP-RE 把一个类画成 UML 图中的一个节点，然后按照继承关系把各个类连接起来。

下面是 OOP-RE 创建的一个 UML 图：



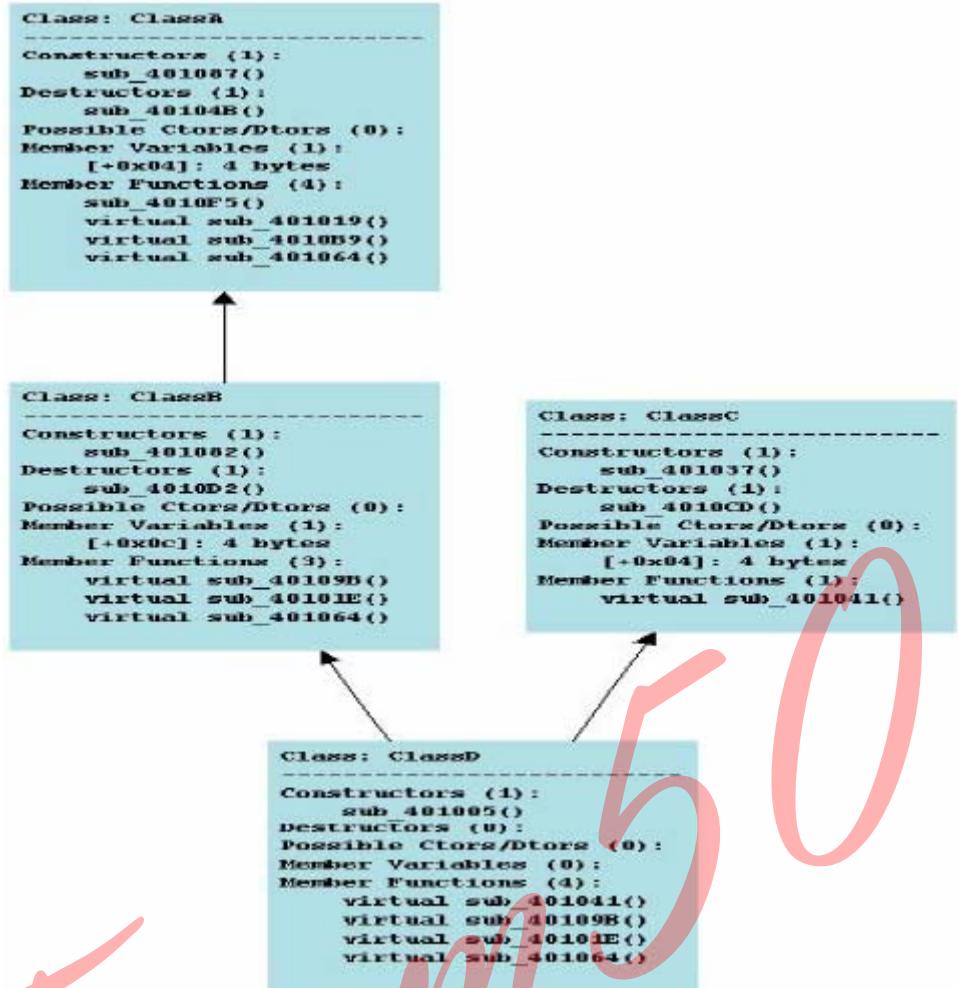
对应上面这张图的类声明是：

```

class ClassA {...}
class ClassB : public ClassA {...}
class ClassC {...}
class ClassD : public ClassB, public ClassC {...}

```

如果目标二进制可执行代码在编译时启用了 RTTI，分析结果会更好些，类的类名也会被显示出来，如下例：



这张 UML 图把类的结构以及类与类之间的关系以直观的方式显示给逆向分析人员，这样逆向分析人员在进行逆向分析时就能更加得心应手了。

IV. 小结

本文讨论了如何对一个 C++ 开发的二进制可执行文件进行分析的方法，特别是分析与类相关的信息以及类与类之间的关系的方法。我们希望能对大家在逆向 C++ 的代码时提供一些帮助。