老『码』识途

从机器码到框架的系统观逆向修炼之路

韩宏 李林 ◎ 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

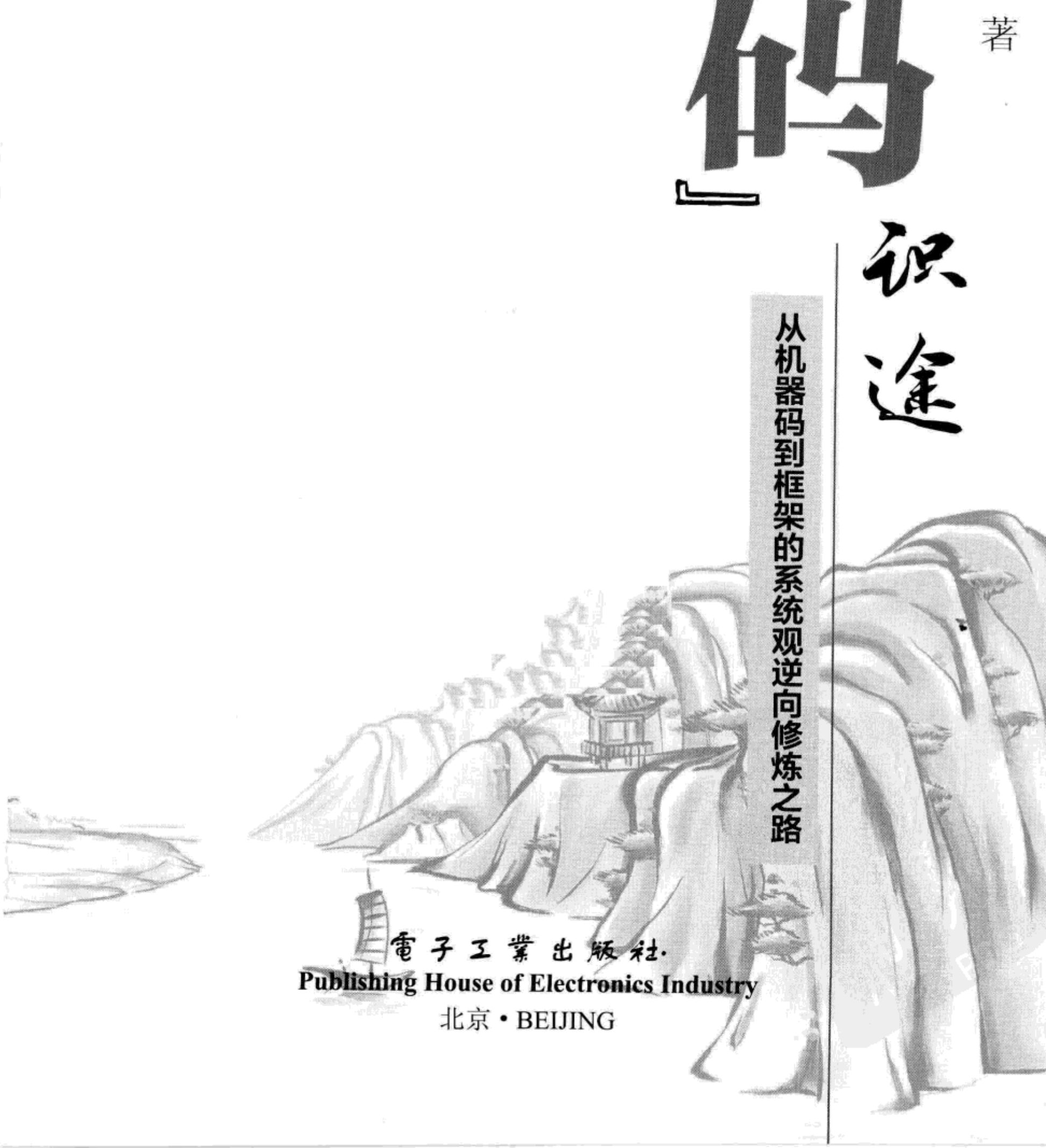
HTTP://WLSAM168.400GB.COM

ISBN 978-7-121-17382-0
9 787121 173820 >
定价：56.00元
策划编辑：章海涛
责任编辑：章海涛
封面设计：一克米工作室

老『码』识途

从机器码到框架的系统观逆向修炼之路

韩宏 李林◎著



電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书以逆向反汇编为线索，自底向上，从探索者的角度，原生态地刻画了对系统机制的学习，以及相关问题的猜测、追踪和解决过程，展现了系统级思维方式的淬炼方法。该思维方式是架构师应具备的一种重要素质。本书内容涉及反汇编、底层调试、链接、加载、钩子、异常处理、测试驱动开发、对象模型和机制、线程类封装、跨平台技术、插件框架、设计模式、GUI框架设计等。书中包含不少工业级或非公开案例。读者不仅能以底层观和调试技巧解决各种实际问题；还可掌握一套学习方法，如“猜测—实证—构建”，调构学习法。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

老“码”识途：从机器码到框架的系统观逆向修炼之路 / 韩宏，李林著 . —北京：电子工业出版社，2012.8

ISBN 978-7-121-17382-0

Ⅰ . ① 老… Ⅱ . ① 韩… ② 李… Ⅲ . ①程序设计 Ⅳ . ① TP311.1

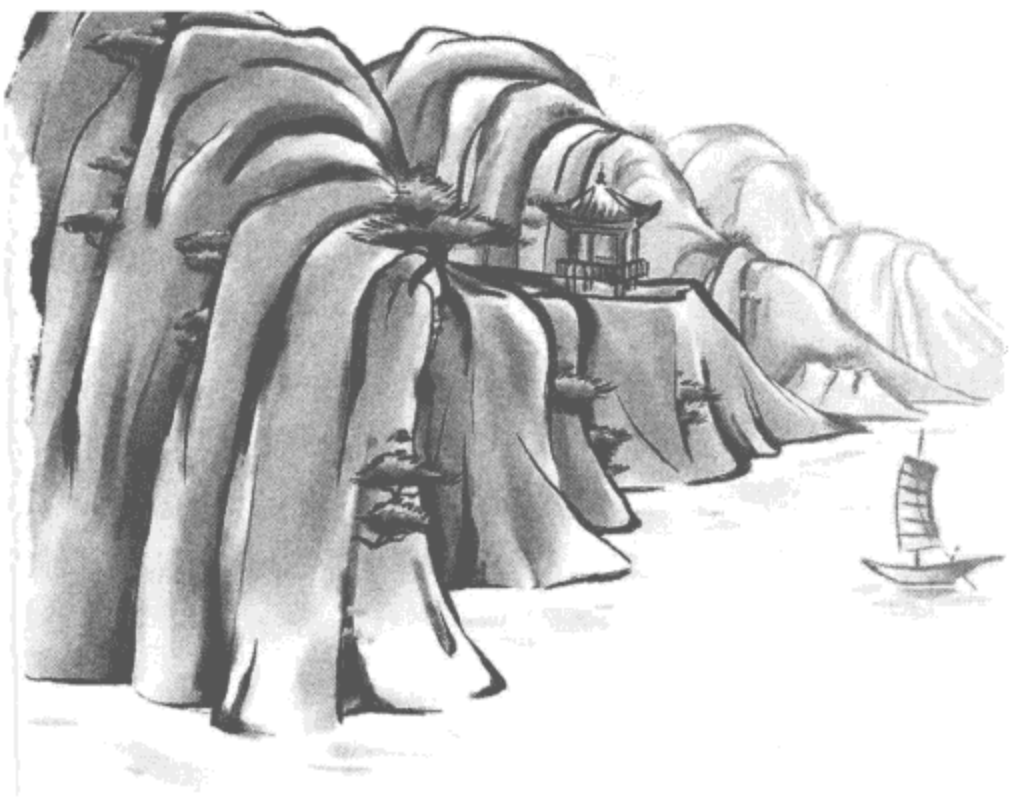
中国版本图书馆 CIP 数据核字（2012）第 136517 号

策划编辑：章海涛
责任编辑：章海涛　　特约编辑：何　雄
印　　刷：
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×1092　1/16　　印张：21.5　　字数：560 千字　　插页：4
印　　次：2012 年 8 月第 1 次印刷
定　　价：56.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。



P 序 reface

——“码”途有道何为径

你主要能得到什么

如果你想成为高级程序员或软件架构师，什么才是技术上的核心竞争力？仅仅是知识？在这个随时可求助于谷歌和百度的年代，知识似乎已变得非常廉价了。而青春的流失并不能给我们留下技术财富，似乎只是将我们变成自嘲的“码奴”。

核心竞争力究竟在哪里？本书认为，一个关键要素就是“系统观”，这是高级软件人才必备的素质。系统观是美妙的，它能自我生长，自我完善，有了它，你就拥有了一颗能成长的原核。系统观是核心知识架构和对它们不断运用所形成的思维方式的复合体。本书的首要任务就是帮助大家建立系统观。

◎ 本书采用独特的自底向上贯通的方式帮助读者完成系统观的构建，从反汇编、机器码入手，以逆向分析贯穿，运用一点点增加的知识不断探索出新知识并最终上升到框架。

◎ 本书采用生长式的学习方法，每个后续章节都是运用前面内容探索而来的，既让你制造“砖头”，又让你用它建造“大厦”。读完本书后，你不仅能掌握一个小而全的可自我发展的核心知识架构，更能掌握一种被笔者称为“猜测—实证—构建”的系统观的思维与学习方法。你将经历一次畅快的知识探索的发现之旅。

你还能得到什么

学完本书后，除了系统观这一主要成果，你还能获得一些有意思的能力，比如：

◎ 掌握被称为“调构学习法”的第三方源代码学习方法（成为架构师的重要能力）。

◎ 在C语言中模拟面向对象的继承、封装、虚函数覆盖等。

◎ 用病毒常用的自定位代码技术解决框架级软件的优雅构造。

◎ 利用钩子技术，在只有执行程序的条件下改变第三方软件的行为。

◎ 运用链接原理自己动手链接OBJ文件，使其可执行。

◎ ……

对笔者而言，这是一本关于“探索知识、让知识融入生命”的书；是一本关于能力与学习的书；是一本展示如何将看似琐碎的知识碎片不断打磨成珠玉，慢慢串成美丽珠宝的书；

是一本展示编程技艺如何自我锻造的书。也可以看成是与大家分享码途人生的飞絮呓语。

学什么？少即多的系统观

学什么？记得在Internet还没有成为主流的时代，有个学通信的硕士被一个IP路由器问题困扰了一个多星期。笔者最好的朋友，学空气动力学的，他玩了10多年DOS系统，从没实际接触过网络，只是“看”过一本资料，还不是IP网络，仅花两天就解决了这个问题。为什么有人学了很多技术和语言却不如未学习该技术的人呢？该问题另一个问法就是“什么才是我们最需要掌握的？”

笔者的看法是：“少就是多，将这个少变成多”。下足工夫，将“少”做深、做厚、变多，就是将来应对千变万化的舵。这个“少”就是计算机的系统观。看看上面的例子，那个朋友对DOS的熟悉程度已经到了对任何一段地址空间作用都了如指掌的地步，自己还实现过一个汉化的DOS。他无疑是建立起了坚实的系统观。这个系统观能指导他按照计算机的“逻辑”方式思考，而路由器不过是这种思维方式的一个例子罢了。有了系统观，学习新东西时，用它去猜测、构想学习对象的可能做法并实证分析，是一种迅捷的办法。笔者在处理SaaS软件中的hibernate将同一持久化对象存入不同库的问题时就是这样，虽然没用过hibernate，但通过猜测和调试，用了约半天时间就解决了问题（搜索并非银弹，该问题当时在网络的中英文查询中均未查到）。

到底什么是系统观？有些东西只可意会难以言传，且见仁见智。这时，纠结含义不如能够运用。诗词意境颇似系统观，诗人自身可能都说不清其内涵，可他能运用这种意境，“运用为王”。其实，我们只要能建立系统观就好了。

如何建立？一种方法是从无到有自己撰写一个系统级软件（哪怕是实验性的），如嵌入式数据库、小型操作系统或编译器。这非常有效，那位朋友得益于撰写汉化DOS的经历，在没有学过嵌入式的情况下，一个月时间就解密了一个单片机太阳能系统，并汉化。但这种方法需要较长一段时间完全投入，且难度不小。针对这些问题，本书给出了另一种方法。

本书内容：逆向入道，自底向上建立系统观

在建立系统观方面，本书没有选择正规的站马步的方式——构建一个操作系统或编译器，而是提供了一种剑走偏锋的方法——自底向上，以逆向反汇编入道，贯穿从机器码至框架的学习。因为，逆向是一把匕首，小巧，却能劈开黑箱直见根本，实乃实证利器。

本书涉及范围既广也窄。从“广”来说，覆盖了汇编、反汇编、逆向、调试、链接、线程、插件风格编程、设计模式、对象模型和机制、框架源代码学习等内容。从“窄”来讲，本书紧紧围绕运用基础知识这把“瑞士军刀”，披荆斩棘，最终建立系统观这一主旨。同时，本书不仅让我们多了一种能力（书中各章节均给出一些调试技巧），更将OS、Compiler等相关知识从压箱底处翻到台面上，并通过逆向思维有效整合在一起。你会发现，这些原来似乎抽象的知识，完全渗透到我们破疑解惑的过程中了。

本书由7章构成，其中第6章（除6.5节）为李林著，其余为韩宏著。

第1章是最重要的基础，以逆向反汇编角度建立了语言的物理模型，充分利用GUI调试环境，将所有知识点做到可“把玩”。例如，在分析mov指令机器码时，通过断点，并利用内存窗体修改机器码，从而改变C语言赋值语句的效果，也展现了一种学习汇编的方法——基于分

析RTL（runtime library）的学习方式。本章还充分展示了“猜测—实证—构建”的思想。

第2章展现基础知识的底层力量，给出多个运用前章逆向知识解决问题的例子。其中有的例子应该是首次公开的。本章揭示出逆向分析的力量，不仅可运用于加密、解密，而且真的能指导我们解决平常的开发难题。部分内容覆盖了钩子、异常、注入等技术。

第3章呈现以构建为驱动的软件开发学习方式，知识覆盖链接器、测试驱动开发、数据结构、软件重用、文件访问等。内容有其特点，如以构建的角度学习了链接器相关知识，并创建了一个最小“Linker”演示。在后续动态链接库运用中，以此为基础解决了棘手问题。

第4章以创造面向对象语言方式，展示了对象模型、机制及其重要特点和使用模式，涉及构造、析构、基于栈构造技巧、虚函数机制等。这种创造的方式集中体现了猜测、实证的思想，在理解的基础上深入探讨了在C语言中如何运用面向对象思想，给出了相关宏；并分析了一些重要设计原则及其在C++中的利弊和不同语种的权衡。以逆向思维贯穿所有问题的解决是本章的特点。

第5章构建了一个跨平台线程类，充分利用了第4章的知识，最终创建了跨平台线程库，并展示不同语言下虚函数机制导致的软件设计差异以及bug的解决过程。从设计到debug的过程体现了系统观和底层调试的重要性。

第6章在第4章基础上展示了一个实用插件框架的演化，在不修改已有代码的前提下，运行时为系统添加新功能。其中，DLL内存释放bug的解决方案充分展示了调试和系统观的力量。

第7章讨论了阅读优秀源代码的方法与技巧。这不仅是程序员成长的重要手段，也是开源时代的重要能力（对一个框架，在资料匮乏之下，你必须快速扩展框架，加入自己需要的功能）。以分析VCL框架为例，并对比MFC，结合框架阅读和创建，展现了被笔者称为“调构学习法”的源代码学习方法。调试、追踪、分析了自定位机器码解决this指针获取的问题，并模仿这一机制创建了可重用的窗体控件框架。本章将猜测—实证—构建思想贯穿到了框架源代码学习和构建中。

我的码道：三步斩“码”刀

修炼系统观不是简单知识的积累，需要一种“道”。笔者的“道”就是：猜测—实证—构建。

“猜测”让我们主动思考，发现问题。这样学习不再是按部就班，而是“寻找式”，对症下药，有选择地吸收，迅速且准确，直指症结。如1.3.1节学习call指令时，不直接看其解释，而是先分析机器码，猜测哪部分可能包含转跳地址，最后经多次猜测实证才搞清楚。这样，将一般被动接受式学习变成了主动研究式的破疑。简单地说，学习任何不懂的技术或框架前先不要急于看它怎么做，而想想自己怎么做，要想到可实证的细节，否则无意义。

“实证”体现为每个知识点都可触摸，或调试、或编程、或使用，能验证。许多人猜测后认为一些东西“很显然”，而不验证，此时“真知”就从指间溜走了。如2.9.3节猜测状态保存时，细分了局部变量、参数，并考虑优化。实证与猜测相悖处即产生前进动力：我们常感叹自己不善提问，而实证就是最好的设问及解答的过程。实证时往往一个知识点牵扯了很多复杂因素，看来几乎无法实证。此时关键就是“领悟精髓，删繁就简”。本书第3章展示了如何通过摸索理清链接的关键逻辑，构建一个2~3天能完成的Linker核心小程序。

“构建”要求尽量将学到的知识通过编写程序模仿出来。这非常有助于以系统观思考问题。而构建中会出现意想不到的问题，它们又是进步的阶梯。比如，多年前学习《Windows

核心编程》中内存访问API时，笔者抛开书上的例子，用这些API编写了游戏修改器。之后就发现了新问题：遍历整个线性地址空间寻找某个整数耗时太长，速度完全无法与真正的修改器比拟。最后，基于最基本的对齐原理解决这一问题，就不用挨个字节搜索了。基础知识在一个个构建、意外及调试解决中被贯穿起来，终成系统观。本书每章均有展现构建的威力。

其实，“码”途之道最重要的就是“快乐”。有了它，你就有了自己的“道”。在本书中，你能看到面对问题的兴奋和努力探索的快乐。如第2章最后两个例子，就是笔者大年初一连续几天疯狂逆向的结果。一个个失望和惊喜后，明媚的阳光穿过0/1 bit的迷雾晒到皮肤上，这就是生命的味道。

如何使用本书

读者有一定的C和C++语言的基础知识就可上路了，你不需要很懂汇编，我们会一点点探索。中途遇到什么概念不懂，可上网查阅学习。既然强调体验式，你就将本书当成游戏攻略吧，你一定要在计算机上“玩”成这个“游戏”，而不是在床头“看”它，要通关哦。随书附带了源代码，但你不要直接用，请自己实现一次，将源代码当做答案吧。（所有源代码并没有采用附盘的形式提供给读者，而是采取网站免费下载的方式，读者可以登录http://www.phei.com.cn或者http://www.hxedu.com.cn进行下载。）

本书以VS 2008为主要调试环境，还用到其他工具：Delphi、Ultraedit、PELord、PEView和IDA pro。

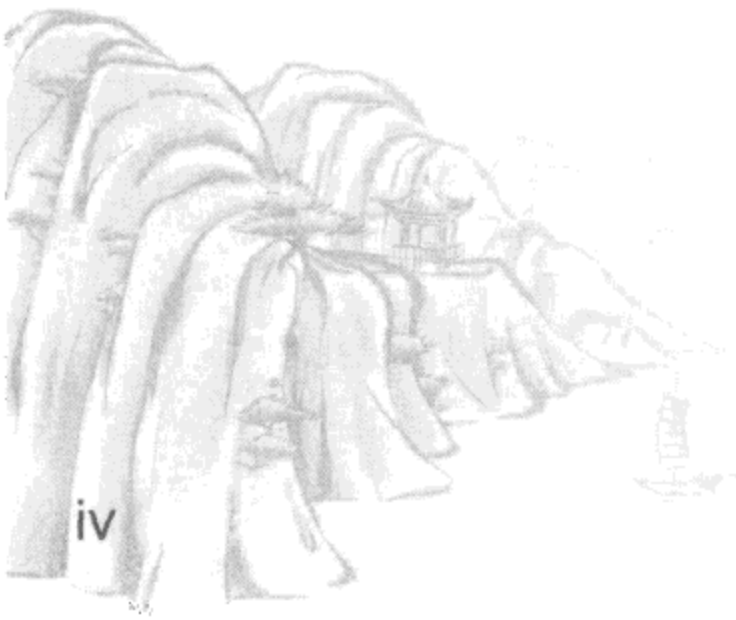
书中解决问题的体验式过程只从文字看（特别纠缠了反汇编后）容易迷失，所以每章都挑选了一个较难的案例编写了思维导图（除第7章外），读者可配合使用。通过导图的符号能索引正文；通过页侧的标记又可快速定位导图中位置，以便来回参阅，理顺思路。

结束本文前，用一首打油诗以抒心意：

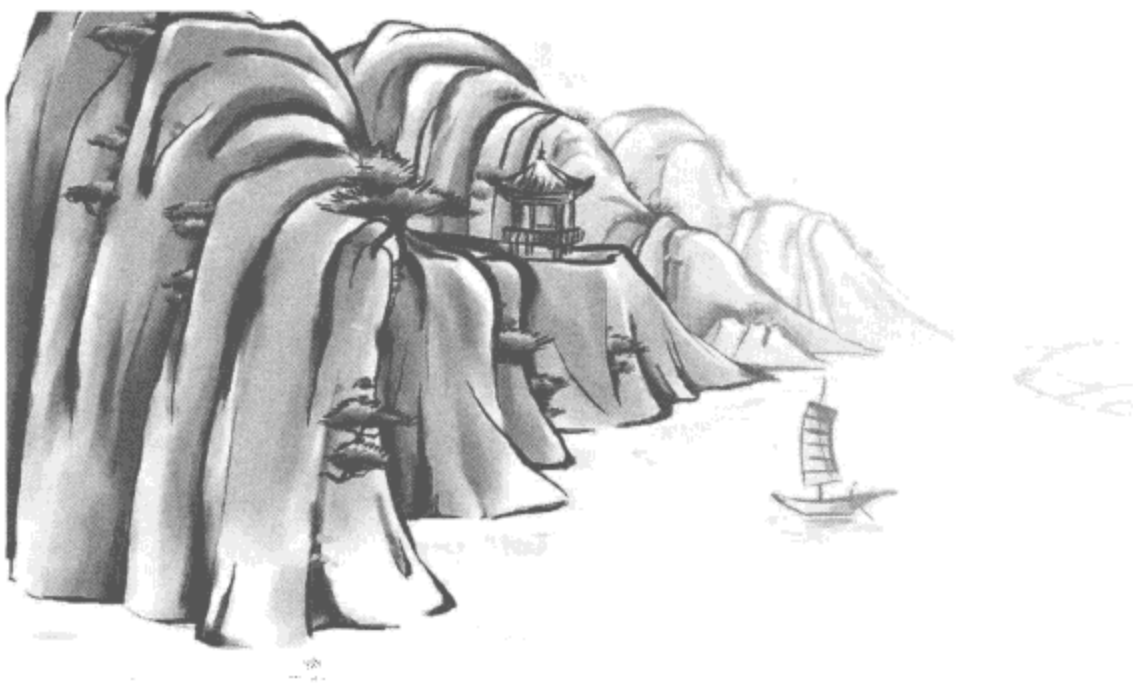
人被“码”驯为码奴，
何日驯“码”骋康途。
老“码”识途途何在，
三步斩“码”刀自出。

我的斩“码”刀就是“猜测—实证—构建”，你的呢？

最后感谢家人、朋友与学生（特别是陈松进行了细致的校对），他们在著书的过程中给予了笔者莫大支持并提供了宝贵意见。还要致谢高辉老师，他就strlen算法给出了证明。



韩 宏
二〇一二年八月



如何阅读本书

1. 通读式阅读

如果你是有一定基础但缺乏用底层的视角建立系统观的经验和经历，请从头到尾完全按照书上所展示的实验一步步分析，最好能自己提出新问题并验证。如果确有难以解决的问题，可发到作者邮箱hhan@uestc.edu.cn。对于常见问题，笔者也会在博客中与大家分享、探讨，http://blog.sina.com.cn/u/2103052793。本书更重体验式，所以单从简单阅读来看，多会产生一种知识点散漫之感，但如能静心用程序实证，所有的方法和系统感自会悄然建立。真正的知识探索过程本就是多线索、离散化、过程式的，在一种反复的淬炼中自然会升华，这也是工程性手艺活的特点。最典型的就是中国功夫的训练，或站桩，或一招两式，反复打磨，似乎全无体系，一朝顿悟，自能豁然贯通。这是东方式的体悟和西方式的解析解构的差别。

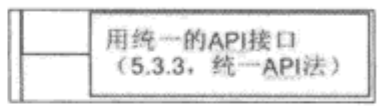
当然，如果你在经历了实证过程后，还需要有人引领你提炼出书中的知识架构，可参考第2种方法。如果你在之后需要快速查找一些知识点，可参考第2种方法。

2. 快速索引式

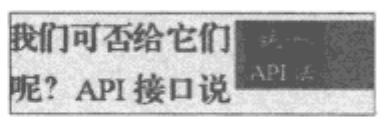
如果你是教师或有很深厚功力的读者，希望从本书中快速查找自己感兴趣的部分阅读，可参阅本节提供的知识导图和索引，快速找到知识点。

总体知识导图是笔者总结的本书所涉及的知识点的架构，可帮助读者回忆和总结，由三部分组成：一是简略总图，能帮助你了解其概况；二是较细的总图，其中主要知识点均有相关索引标注，你能快速跳到感兴趣处；三是分图，针对各主要部分展示最全面的知识索引和架构。

标注分两种：一种是直接给出章节号，另一种是给出章节及其中的关键词索引。后者使用方式如下：例如，知识导图中某知识点如下，先说明该知识点在5.3.3节中，关键词是“统一API法”，然后在5.3.3节正文的外侧查找如下所示的图标，“统一API法”标注的行就是相关内容。



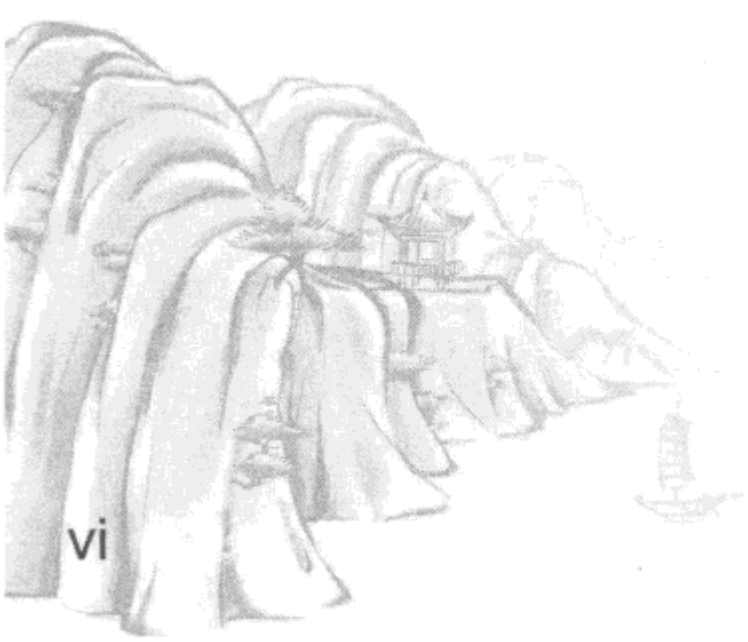
知识导图的标注

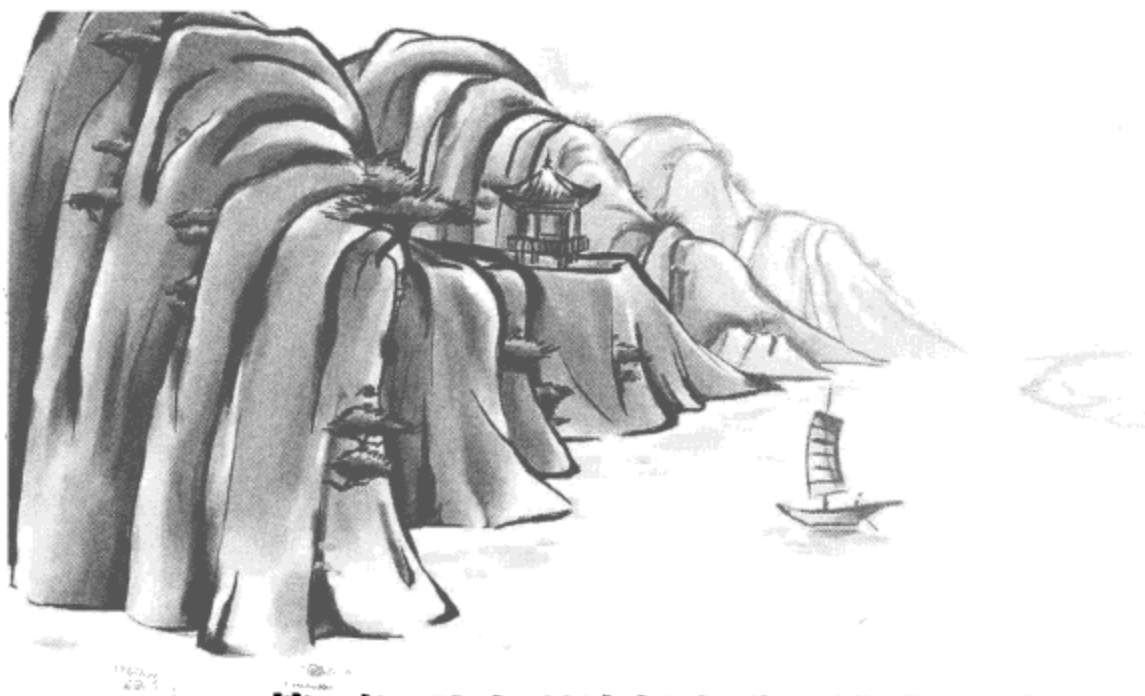


正文中的标注

3. 章首页

每章都有首页，其中展示了本章所需前导知识点及其所在位置，以及本章对后续章节知识的支撑作用。如果你直接切入到某章，“需准备的知识”辅助你有效阅读，“为你提供的能力则”让你明了本章内容有何价值。





C目 录 Contents

第1章 欲向码途问大道，锵锵bit是吾刀 …… 1

1.1 全局变量引发的故事 …… 2
1.1.1 剖析赋值语句机器码 …… 2
1.1.2 修改赋值语句机器码 …… 6
1.1.3 直接构建新的赋值语句 …… 7
1.1.4 小结 …… 10
1.2 理解指针和指针强制转换 …… 11
1.2.1 指针和它丢失的类型信息 …… 11
1.2.2 指针强制转换 …… 13
1.3 函数调用和局部变量 …… 15
1.3.1 计算指令中的跳转地址 …… 15
1.3.2 返回故乡的准备 …… 16
1.3.3 给函数传递参数 …… 17
1.3.4 函数获取参数 …… 18
1.3.5 局部变量 …… 20
1.3.6 返回故乡 …… 20
1.3.7 返回点什么 …… 23
1.3.8 扫尾工作 …… 28
1.3.9 调用惯例 …… 30
1.3.10 函数指针 …… 31
1.4 数组、结构体 …… 34
1.4.1 数组 …… 34
1.4.2 结构体 …… 35
1.5 无法沟通——对齐的错误 …… 37
1.5.1 结构体对齐 …… 37
1.5.2 无法沟通 …… 41
1.6 switch语句的思考 …… 44
1.6.1 switch机制探索 …… 45
1.6.2 switch语句一定比if-else语句快吗 …… 50
1.6.3 switch的再次优化 …… 50
1.7 关于其他高级语言要素的反汇编学习 …… 54
1.8 全局变量的疑问——重定位和程序结构 …… 54
1.8.1 独一无二的全局变量 …… 54
1.8.2 不变的地址和重定位 …… 56
1.8.3 动态链接库中的重定位 …… 64
1.9 汇编的学习之路——阅读RTL …… 68
1.10 程序设置说明 …… 76
习题1 …… 76

第2章 庖丁解“码”：底层的力量与乐趣 …… 79

2.1 解密之hello world …… 80
2.2 奇怪的死循环 …… 83
2.3 我们都犯过的错——指针的指针 …… 85
2.4 互通的障碍（跨语种调用） …… 87
2.5 错误的目的地 …… 90
2.6 网络发送出错了 …… 91
2.7 为什么代码运行完毕却出错 …… 93
2.8 失效的管道 …… 96
2.8.1 管道的力量 …… 97
2.8.2 我要控制Telnet客户端 …… 101
2.8.3 不是所有管道都可抽象等价 …… 101
2.8.4 一动不动的48小时 …… 103
2.9 异常世界历险记 …… 112
2.9.1 学习基础概念 …… 112
2.9.2 如何返回 …… 113
2.9.3 那些状态保存到哪里了 …… 117
2.9.4 意外的秘密 …… 120
习题2 …… 127

第3章 成长：与程序一起茁壮 …… 131

3.1 初写系统 …… 132
3.1.1 代码风格 …… 132
3.1.2 常量 …… 133
3.1.3 最简单的电话簿（1）：功能设计和相关库函数学习 …… 134
3.1.4 最简单的电话簿（2）：系统实现，分割函数 …… 141
3.1.5 空字符结尾串的警觉 …… 143
3.1.6 让程序更有组织性 …… 144
3.2 有序的世界：可测试与可跟踪的系统 …… 146
3.2.1 电话簿扩展（1）：硬盘结构体数组 …… 146
3.2.2 指针的陷阱 …… 148
3.2.3 动态数组 …… 149
3.2.4 变化的压力与出路：重构、单元测试和日志 …… 151
3.2.5 电话簿扩展（2）：可测试的恩赐 …… 155
3.3 优雅的积木 …… 155
3.3.1 可复用硬盘数组 …… 155
3.3.2 分享它（1）：理解编译链接过程 …… 161
3.3.3 分享它（2）：我的丑陋链接器 …… 167
3.3.4 分享它（3）：静态链接库 …… 173
3.3.5 分享它（4）：动态链接库 …… 175
3.3.6 搭积木的艺术 …… 178
习题3 …… 182

第4章 让我们创造面向对象语言吧 …… 185

4.1 “封装”数据函数合一，陈仓暗度this传递 …… 186
4.1.1 那些讨厌的事 …… 186
4.1.2 像芯片一样工作（1）：数据合一 …… 187

4.1.3 像芯片一样工作（2）：行为与数据合一 …… 188
4.1.4 不想让你传递“自己” …… 189
4.1.5 创造吧，新的语言 …… 190
4.1.6 是这样吗？我们需要证明 …… 191
4.2 太麻烦了，需要更简单的创造与销毁 …… 194
4.2.1 创造构造函数和析构函数 …… 194
4.2.2 构造中分配资源，析构中释放资源 …… 197
4.3 对比C语言的“对象”和面向对象 …… 199
4.4 体验封装的力量 …… 201
4.4.1 生死原点，整体资源管理 …… 201
4.4.2 文件流 …… 203
4.5 整体资源管理的爱恨 …… 204
4.5.1 扩展技巧：保证成对出现，巧妙的自动线程锁 …… 204
4.5.2 美丽的幻影：不可靠的自动析构 …… 205
4.5.3 隐藏的敌人：不请而至的析构和拷贝构造 …… 206
4.6 封装之强化：内外之别，亲疏之分 …… 209
4.6.1 私有的诞生 …… 209
4.6.2 私有？阻止不了我 …… 210
4.6.3 理解继承的机制（1）：模型 …… 211
4.6.4 理解继承的机制（2）：在C语言中“玩”继承 …… 214
4.6.5 保护的诞生 …… 218
4.7 “变”的烦恼与出路：创造虚函数 …… 218
4.7.1 “三变”之苦：格式化字符串 …… 218
4.7.2 函数指针，请带我走出不断修改的泥潭 …… 220
4.7.3 再进一步：做成对象 …… 221
4.7.4 我们需要性能更好的版本 …… 223
4.7.5 我们需要新语法，创造虚函数吧 …… 225
4.7.6 验证虚表机制（1）：反汇编分析 …… 226
4.7.7 验证虚表机制（2）：直接用虚表来调用虚函数 …… 227
4.8 虚函数的那些事儿 …… 227
4.8.1 理解“=” …… 227
4.8.2 纯虚函数，从DLL导入对象 …… 230
4.8.3 C语言实现虚函数 …… 231
4.8.4 魂归何处：析构之“虚” …… 232
4.8.5 理解运行期类型判断dynamic_cast …… 232
4.9 静态覆盖 …… 235
4.10 静态与非静态成员函数的区别 …… 235
4.11 遥远的风景：管窥 .NET对象 …… 235
习题4 …… 236

第5章 底层与抽象的混沌：一个跨平台线程类的封装、错误与进化 … 239

5.1 先学习多线程编程吧 …… 240
5.1.1 概念 …… 240
5.1.2 Windows下的线程接口 …… 240
5.1.3 第一个线程程序 …… 242
5.1.4 那些复杂的参数和bug …… 243
5.2 简单、重用，让我们构造线程类吧 …… 247
5.2.1 无赖的尝试，原来是它——static …… 248
5.2.2 可爱的virtual和可恨的this …… 249

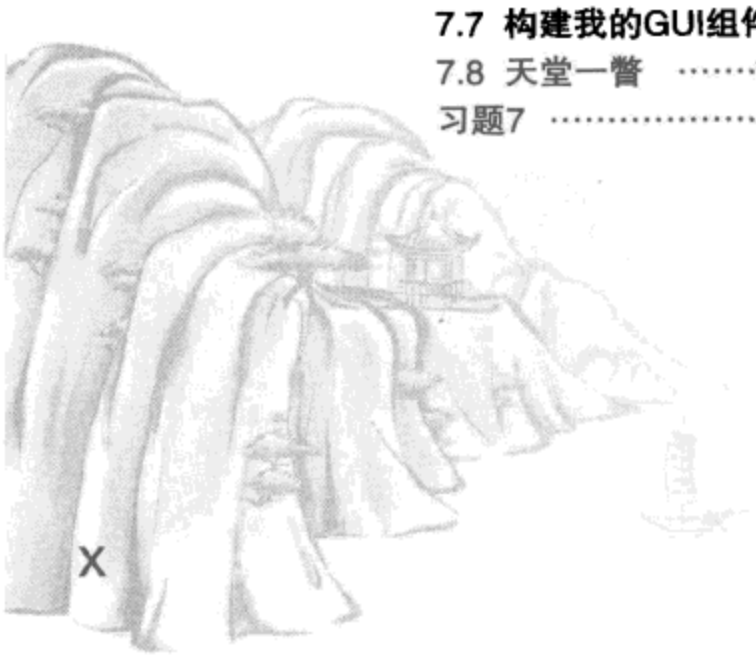
5.2.3 私有、保护、公有、只读、纯虚函数，一个都不能少 …… 251
5.2.4 析构中释放资源 …… 252
5.2.5 我们发现了一个设计模式 …… 252
5.2.6 我关心，你通知——我们的第二个设计模式 …… 253
5.3 跨平台的线程设计 …… 255
5.3.1 讨厌的Linux版本 …… 255
5.3.2 源代码跨平台技术 …… 256
5.3.3 跨平台的版本 …… 257
5.4 崩溃，哪里出错了 …… 262
5.4.1 寻找错误 …… 262
5.4.2 C++下整体资源管理的反思 …… 265
5.4.3 生生死死虚表误，剥离策略世界殊——重生 …… 267
习题 5 …… 268

第6章 插件养成记 …… 271

6.1 一个修改已有功能的实例 …… 272
6.2 一个可以动态添加功能的简单实例 …… 273
6.3 一个可以动态添加功能的复杂实例 …… 276
6.4 从函数到插件对象 …… 280
6.5 delete的灾难：谁的书 …… 283
6.5.1 释放内存的崩溃 …… 283
6.5.2 解决之道：新生活，各管各 …… 288
习题6 …… 291

第7章 天堂的阶梯 …… 293

7.1 遥望天堂，那些美丽与简洁我向往 …… 294
7.2 从最基础开始吧，SDK编写窗体程序 …… 295
7.2.1 hello window和基本原理 …… 295
7.2.2 来个复杂点的窗体程序 …… 298
7.3 构建我的GUI组件（1）：简单组件 …… 300
7.4 构建我的GUI组件（2）：天堂的机器码跳板 …… 304
7.4.1 调试，我要看清你 …… 304
7.4.2 我们的自定位代码 …… 313
7.4.3 自定位代码版Button类 …… 314
7.4.4 自定位代码版Form类 …… 315
7.4.5 为什么不错呢 …… 316
7.5 构建我的GUI组件（3）：更多的组件 …… 317
7.6 天堂阶梯，玩赏框架那如花散落的繁复与如索串珠的简洁之美 …… 319
7.7 构建我的GUI组件（4）：我的天堂 …… 326
7.8 天堂一瞥 …… 330
习题7 …… 332



第1章

欲向码途问大道，锵锵bit是吾刀

需准备的知识 ⋙

- 了解一些计算机的基本结构，如CPU、寄存器、内存……
- 基本掌握C语言。
- 了解一点汇编更好。

为你提供的能力 ⋙

- 本章的调试技巧 → 支撑整本书的底层调试和探索。
- 1.1.3节和1.8.2节的机器码构造能力 → 2.1节解密之hello world和7.4节的构造自定位代码。
- 1.2节指针机制 → 2.3节指针的指针bug、5.2.2节线程函数的this指针传递。
- 1.3节的函数调用机制 → 2.1节、2.2节、2.3节、2.4节、2.6节、2.7节、2.8节和2.9节的错误调试和逆向分析，3.3.5节的bug分析，4.1.6节和4.7.6节的this指针传递机制和虚函数机制分析，5.1.4节的bug分析，6.5节内存释放错误分析和7.4节分析自定位代码。
- 1.4.1节数组模型 → 2.2节的bug分析。
- 1.4.2节结构体模型 → 4.1节的this指针机制，4.3节对比C中“对象”和C++对象，4.6.3节继承机制和4.6.4节在C中玩面向对象。
- 1.5.1节对齐 → 1.5.2节的bug，1.9节的分析strlen中的对齐读，3.3.3节的结构体对齐bug。
- 1.6节的switch和1.9节的分析 → 学会一种分析方法和汇编学习方法。
- 1.8节的加载期重定位 → 3.3节的链接期重定位的理解和程序构造。
- 充分理解猜测、实证和构建方法。

如果你是一个穿越小说的主角来到计算机世界，面对着像尘沙般枯燥简单的 0/1 bit、五光十色的软件高楼大厦和像烟火一样绽放又凋谢的各种时髦技术，你的路在哪里？你怎样把握根本大道，创建一个属于自己的王国？是尘沙，那看似枯燥的底层程序观就是你真正纵横捭阖的根基。你想在面对一个没有源代码的黑盒似的执行程序时，为它点石成金般地添加新功能吗？那是上帝说要有光，即有光的神奇。你想在众人困惑于某个诡异 bug 时，以逆向之刀，游刃于乱花迷人眼的反汇编迷网，一记小李飞刀直指那乱网后的真相吗？那是红尘游侠寂寞的潇洒。你想看透各种辉煌技术的本质，举重若轻吗？那是羽扇纶巾的将帅风华。这些能力都来自本章的底层程序观，它是你的马步，你的正法眼。

来看看本章的功效。如果你从 1.1 节和 1.8 节中“玩”会了构造机器码，就能理解并完成第 7 章的构造自定位代码，解决 this 指针的传递问题。如果你领悟了 1.3 节中函数调用的所有细节，就能自己解决 2.4 节语言互通的奇怪 bug。如果你对 1.3 节和 1.4 节有了把握，就奠定了整个第 4 章对象模型的基础。第 4 章从底层剖析了对象，又为第 5 章巧妙的线程封装和第 6 章中从 DLL 导出对象提供了根基。这条通天大路，环环相扣，均根植于本章的点点滴滴的淬炼。

好了，我们的主角，开始枯燥、琐碎又起伏跌宕的修炼之旅吧。

1.1 全局变量引发的故事

1.1.1 剖析赋值语句机器码

我们从下面这段简单地为全局变量赋值语句的反汇编开始：

```
int gi;
void main(int argc, char* argv[])
{
    gi = 12;
}
```

使用 VS 2008 作为调试环境。将光标放在“gi=12;”语句行上，然后按 F9 键，在该行上设置断点（如果继续按 F9 键，将消除该断点），出现如图 1.1 所示的标志●。

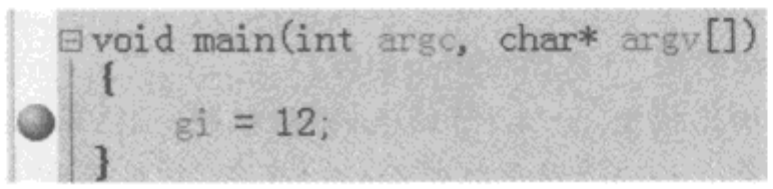


图 1.1

> 断点（break point）：当程序运行时，经过该点所在语句，程序将暂停运行。可在此时观察程序的各种状态，如变量值、内存值、寄存器，甚至可以修改这些相关状态。可以在程序运行前设置断点，也可以在运行中设置断点。

然后保证程序为 Debug 模式（默认模式），如图 1.2 中椭圆圈定部分。因为在 Debug 模式下可进行代码的调试跟踪。VS 2008 的 Release 版本可调试，因为生成了调试信息，而 VC 6.0 必须手动配置，否则无法调试跟踪。

按 F5 键，进入调试运行方式（按 Ctrl+F5 组合键，为非调试运行方式，设置的断点无效）。程序在断点处暂停，见图 1.3。

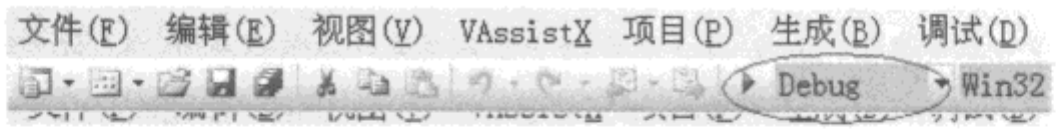


图 1.2

```
void main(int argc, char* argv[])
{
    gi = 12;
}
```

图 1.3

选择菜单的“调试→窗口→反汇编”命令（见图 1.4），进行反汇编（disassemble）（快捷键Ctrl+Alt+D），反汇编的结果见 DM1-1。

图 1.4

第2行是第1行对应的汇编码，左边的数字0041138E是十六进制数表示的赋值指令mov的存放地址，右边是该条指令的汇编表示形式。（**说明：内存地址和其中的值均采用十六进制数表示，在反汇编中一般省略末尾的H或h。全书同。**）学习过汇编的读者会奇怪，全局变量的名字gi怎么会出现在汇编指令中呢？汇编语言没有变量名。因为在VC环境中，为了易理解，开发环境特意将操作对应的符号名也显示在语句中。为了看到纯正的汇编语句，我们选择暂时将符号名从反汇编语句中去除。在代码窗口中单击右键，在弹出的快捷菜单中选择“显示符号名”，取消其选中状态（如果需要显示符号名，则再次单击），见图1.5。此时，反汇编结果见DM1-2，mov指令中没有C语言的变量名gi了，是把0ch（h代表十六进制，0c即十进制数12）赋值给内存。该内存地址为指令方括号中的十六进制数，即00417140h。

DM1-1

```
1   gi = 12;
2   0041138E     mov   dword ptr [gi (417140h)],0ch
3   }
```

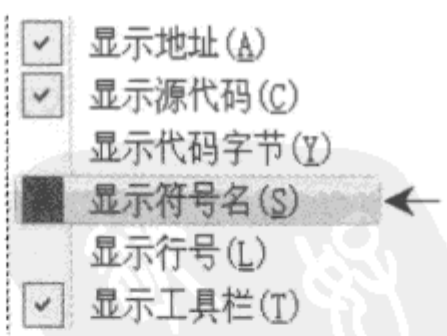


图 1.5

从这里开始，我们就开始探索式学习了。“问题”是探索式学习的关键。我们必须学会存疑发问，然后寻找解决方法求证。让我们来发问吧：

DM1-2

```
  gi = 12;
  0041138E     mov   dword ptr ds:[00417140h],0ch
}
```

计算机的信息存储在内存中，mov 指令将数据存放在指定地址的内存中。C 语言的全局变量本质上是一块内存，所以对它的存取也是通过地址进行的。

“mov [地址值], 存储值”指令将存储值存入括号中的地址指向的内存中。DS 是数据段寄存器，在 x86 寻址中是段寄存器和段内偏移合在一起定位的（这有点像用街道名和门牌号一起定位的方式）。在 Windows 和 Linux 系统中，所有段寄存器都指向一个位置，所以相当于只有一个段，段寄存器可以不用考虑。

① 内存 00417140h 真的是 gi 的地址吗？

② mov 指令真的放在内存 0041138eh 地址中吗？

先来解决第一个问题。打印是我们最熟悉的基本方法。修改源代码如下：

```
gi=12;
printf("gi address=%x\n", &gi);
```

这样太麻烦了，如果每次想看不同值，岂非要不断修改程序？这是不能容忍的！因此我们利用调试环境的一个能力，用监视窗口来观察程序状态，如变量和表达式。选择如图 1.6 所示的菜单命令，激活一个监视（watch）窗口。

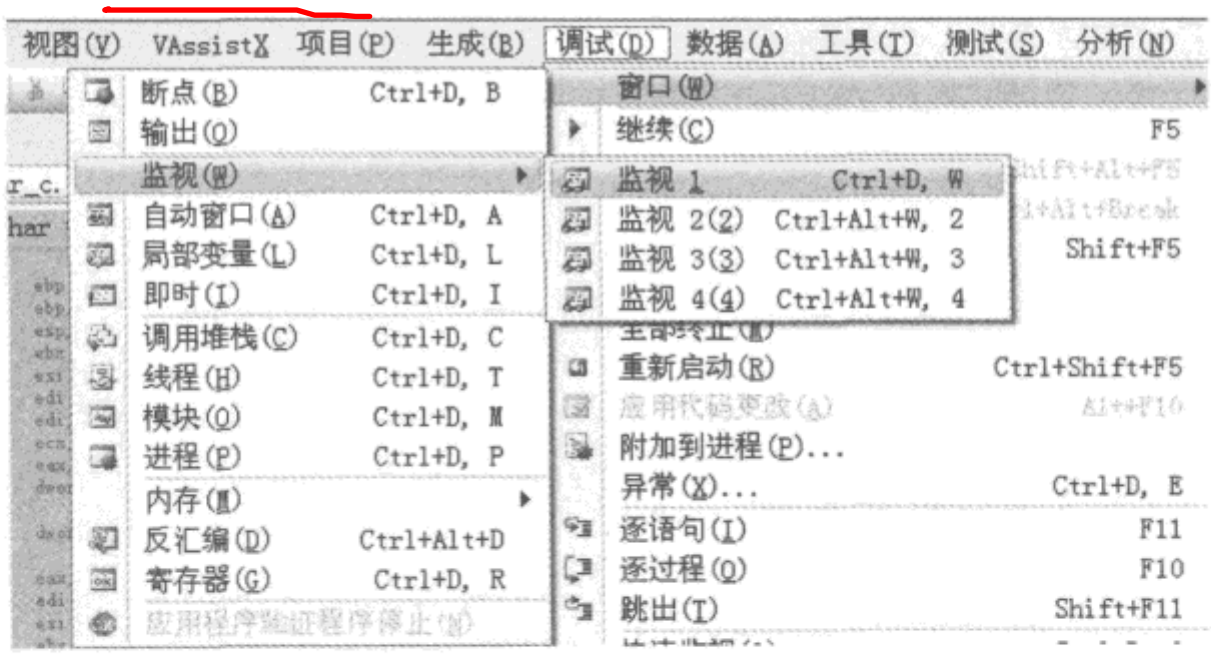


图 1.6

选择一个空行，双击“名称”列，输入“&gi”，回车，则在“值”列中显示其值 0x00417140（**与尾部加 h 一样，头部加 0x 也是十六进制数的一种表示方法**），见图 1.7。证明前面 mov 指令中的地址确实是 gi 的地址。

我们暂时将第二个问题放一放，进一步探索 mov 指令，来看它的机器码。在代码窗体中单击右键，在弹出的快捷菜单中选择“显示代码字节”（见图 1.8），则反汇编的结果如下：

```
gi = 12;
0041138E   c7 05 40 71 41 00 0c 00 00 00      mov   dword ptr ds:[00417140h], 0ch
```

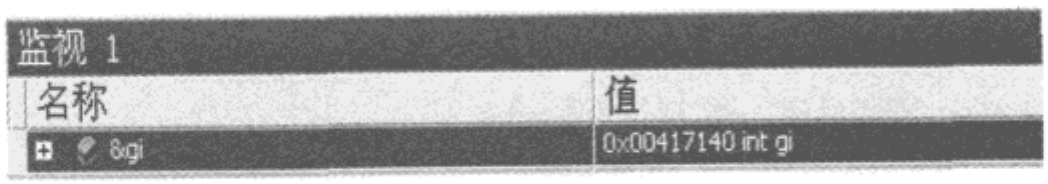


图 1.7

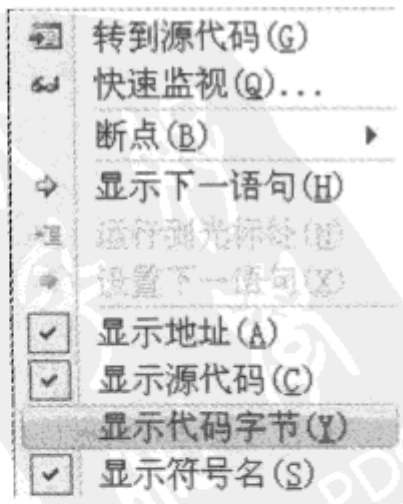


图 1.8

mov 指令的机器码处于指令存放地址和汇编指令中间，即黑体显示部分。

现在来分析这个机器码中包含的与 mov 指令相关的信息。首先还是猜测，该指令应该包含了 3 方面的信息：要赋的值 12，要赋值的内存地址 00417140h，该指令是 mov 指令。按照猜测，我们在给出的机器码 c7 05 40 71 41 00 0c 00 00 00 中来实证。

我们需要观察、分析和猜测。赋值的 12，即 0ch，就被包含其中，从右边数第 4 字节。赋值的地址 00417140h 好像没有，但是其中包含每个单独的字节，00、41、71、40。从右往左看，则右数第 5～8 字节分别是 00 41 71 40。我们因此猜测上面的机器码分成如下 3 块：

| c7 05 (代表 mov 指令) | 40 71 41 00 （地址 00417140h）| 0c 00 00 00（代表值 12）|

套用前面对地址的分析，第 3 块从右往左看就是 0000000c，即 12。而 int 类型在 32 位机上就是 4 字节，所以这条指令中用 4 字节表示了 12。

这样我们似乎发现了一个规律，计算机上表示的整数在内存中是按字节倒序存储的。0000000c 在内存中就是 0c 00 00 00，00417140h 在内存中就是 40 71 41 00。

下面来验证。将“gi = 12;”改成“gi=0x12345678;”，再反汇编：

```
gi = 0x12345678;
0041138E c7 05 40 71 41 00 78 56 34 12     mov   dword ptr ds:[00417140h],12345678h
```

其中，机器码 c7 05 40 71 41 00 78 56 34 12 的后 4 字节“78 56 34 12”应该是被赋的值，倒序排列为“12345678”。可知，我们的猜测是正确的。

这里引出“整数在计算机上的表示方式”的知识。在内存中如何存储整数只是一种规范，分为大端机和小端机两种方式。

小端机：整数逻辑上的最低字节放在内存的最低地址，次低字节放在内存的次低地址，依次存放。比如，0x12345678 放到内存中就是 78 56 34 12。Intel 的 x86 系列 CPU 是小端机。
大端机：与小端机刚好相反。比如，0x12345678 放到内存中就是 12 34 56 78。PowerPC、SUN 的 SPARC、Motorola 6800 是大端机。

小端机
大端机

下面来看 mov 指令的效果。要看赋值效果，我们需要用另一个调试利器——内存观察窗体。在 VC 2008 中选择如图 1.9 所示的菜单命令，激活内存窗体，见图 1.10。

在“地址”栏中输入地址，其值为十六进制数表示（以 0x 开头），然后回车。下方为内存面板区，其中显示内存的值，左边是内存地址，地址从左到右、从上到下增加，显示单位为字节。

首先执行程序，在语句“gi = 12;”断点处停下。在“地址”栏中输入要观察的 gi 的地址 0x00417140，回车，结果见图 1.11。

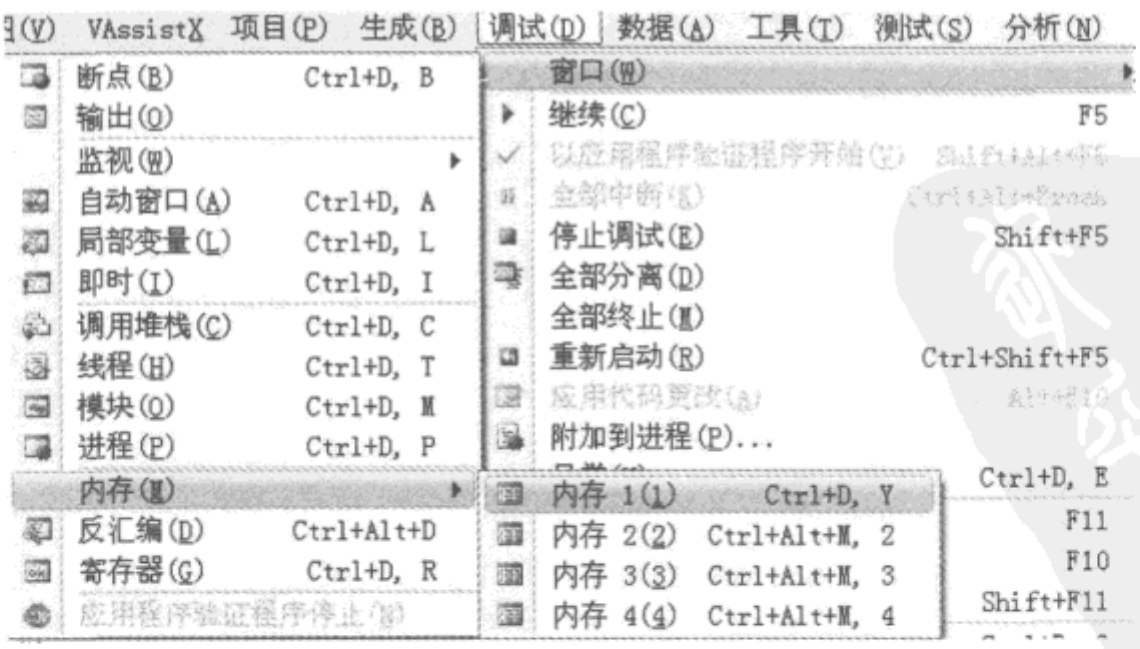


图 1.9

图 1.10

单步执行“gi = 12;”，再观察 gi 所在内存的值，见图 1.12。

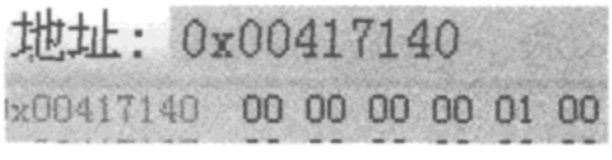


图 1.11

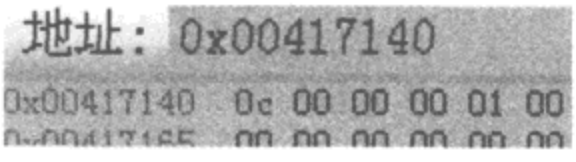


图 1.12

单步执行

单步执行：只执行当前语句。根据单步执行函数的方式，单步执行分为两种：step into（快捷键 F11），跟踪执行进当前函数；step over（快捷键 F10），不跟踪执行进入当前函数，直接到下一语句。

gi 所在内存 0x00417140 中的值从 00 变为 0c。可知，赋值 12 产生了效果。但到底是修改了 1 字节还是 4 字节呢？我们需要证明。（虽然我们知道 int 是 4 字节。）怎么办？我们运用内存窗体的另外一个能力——修改内存的值。

修改内存值

修改内存值：单击要修改的字节，然后直接输入要修改的新值。

重新执行程序，当断点中断时，将 0x00417140 指向的 4 字节全部修改为 11，见图 1.13。然后单步执行“gi = 12;”语句，我们发现，总共修改了 4 字节：0c 00 00 00。至此验证了整数 gi 是 4 字节。最后我们还要看小端机的效果，将赋值语句修改为“gi = 0x12345678;”，单步执行，并在内存窗体中查看，其值确实是“78 56 34 12”。

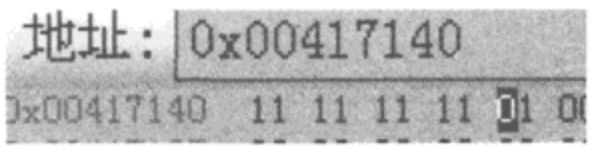


图 1.13

1.1.2 修改赋值语句机器码

现在来看之前提出的问题：

② mov 指令真的放在内存 0041138eh 地址中吗？

怎样来设计这个实验？既然指令依然存放在内存中，本质也是内存中的一些数据，那么我们是否可以通过内存窗体来观察？对于“gi=12;”，该地址的值是 c7 05 40 71 41 00 0c 00 00 00。在内存窗体的“地址”栏中输入 0x0041138E，回车，结果见图 1.14，确实该内存段存放的是 mov 指令的机器码。

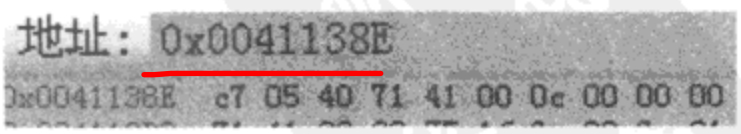


图 1.14

既然指令存储在内存中，并且可以修改内存中的值，那么我们是否可以修改其值，从而达到修改指令的效果？还记得刚才验证，最后 4 字节代表要赋的值，那我们来看修改其内容之后的效果，

如修改为“gi = 894567;”。首先，计算 894567 的十六进制值。可采用 Windows 系统自带的“计算器”来求取，其十六进制数表示为 0xda667。

十六进制数、二进制数与十进制数之间的转换

打开 Windows 系统自带的“计算器”，选择“查看→科学型”菜单命令（Windows 7 下是“查看→程序员”），就可以选择十进制、二进制和十六进制了。比如，在十进制下输入 12，然后选择十六进制，就变成了 c。反向转换也如此。

如果我们要修改内存窗体中代表 mov 指令的字节串“c7 05 40 71 41 00 0c 00 00 00”的后 4 字节，应该怎样将 0xda667 填进去？是“c7 05 40 71 41 00 00 0d a6 67”吗？（**想不清楚？没关系，实验一下。**）调试运行并在断点“gi = 12;”处停止，修改内存窗体，见图 1.15。再按 F10 键，单步执行，在监视窗体中查看 gi 的值，见图 1.16。gi 的值变为“1738935552”，不是 12，也不是我们希望的 894567。那么，说明我们修改还是有部分正确，它确实修改了 mov 指令所赋值，只是值不对。可将该值输入到计算器中看看其十六进制表示，也可用一点小技巧更方便观察其十六进制值。

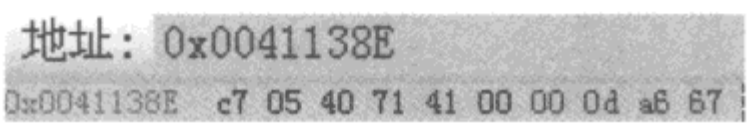


图 1.15

图 1.16

在监视窗体中用十六进制查看结果：因为调试器用十六进制数显示指针变量值，我们只需骗骗它，将被查看的变量强制转换为 void *。比如，我们要查看变量 gi，它本身是 int 类型，输入“(void *) gi”，就可用十六进制查看了。

用十六进制查看 gi 的结果，见图 1.17。

图 1.17

前面计算出的十六进制数应该是 0xda667，与 0x67a60d00 有什么关联？把“0xda667”写为“0x000da667”，并将这两个数按字节分开，即

我们发现，这两个值刚好按字节倒序排列了。（**前面提到过 Intel 计算机是小端机。**）因此，修改后的机器指令应该是“c7 05 40 71 41 00 67 a6 0d 00”，而不是“c7 05 40 71 41 00 00 0d a6 67”。经过这次修改且如前执行并观察，gi 的结果真的变成了我们希望的值。

练习：再定义一个全局变量，然后修改 mov 指令，将其地址部分修改为该变量地址，观察执行结果。

这样我们就全面测试了前面关于该指令各部分含义的猜测。这是一个有意思的关于基础知识学习的游戏，它的名字叫“猜测和实证”。希望大家从中感悟，找到自己的游戏方法，并运用到之后的学习中。在实证后是什么？就是“构建”，见 1.1.3 节。

1.1.3 直接构建新的赋值语句

既然我们发现了指令不过就是一些字节的组合，能否可以抛开 C 语言，自己构造指令来执行？我们完全可以分配一段内存，然后将 mov 指令的机器码填入。但是，如何让 CPU 执行我们的代码？

选择如图 1.18 所示的菜单命令，可在 VC 2008 中激活寄存器窗体，见图 1.19，从中可以查看当前寄存器的值。

关于代码的执行

CPU 中有一组寄存器，其中 IP（Instruction Pointer）寄存器（又叫指令寄存器）与代码执行相关，其大小为 16 位（bit）。CPU 从该寄存器指向的内存读入指令，并执行。执行后，IP 自动加上所执行指令的长度，于是计算机就不停地重复取指和执行过程。后来内存变大，用 16 位存储地址不够了，于是将 IP 寄存器变成了 32 位的 EIP（Extended Instruction Pointer）寄存器。总之，在 Intel 系统中，EIP 寄存器指向哪里，CPU 就将该地址作为将执行指令的入口，即使错误地指向了数据区。

jmp 指令可设定 EIP 寄存器，使 CPU 跳转到设定地址执行。JMP 指令有多种形式，如“jmp [地址]”。

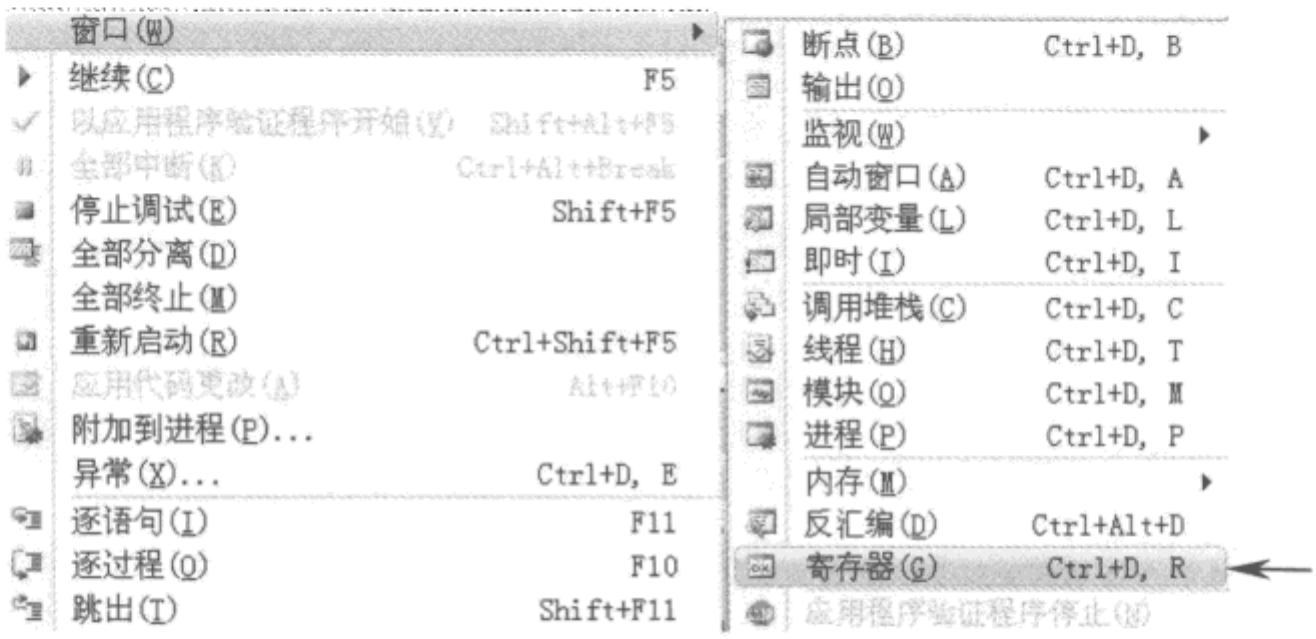


图 1.18

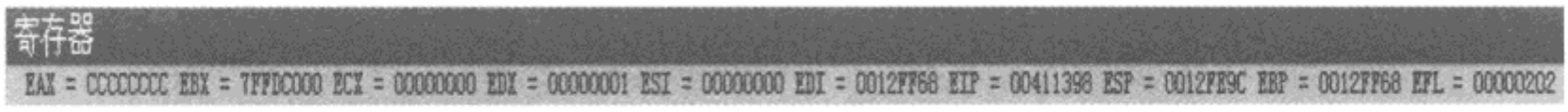


图 1.19

寄存器窗体：在 VC 中可以激活寄存器窗体，它显示所有寄存器的当前值。如果单步执行，寄存器的值与上次不同，将用红色标注。

在“gi = 12;”语句的断点处停下，查看其反汇编显示的指令地址是否与 EIP 寄存器中的值相同。从图 1.20 中两个箭头指示的地址可知，EIP 的值与当前执行的指令的地址相同，都是 0041138e（十六进制数，数字末尾省略了“h”）。按 F10 键，单步执行，发现下一条指令的地址也是 EIP 寄存器中的值。

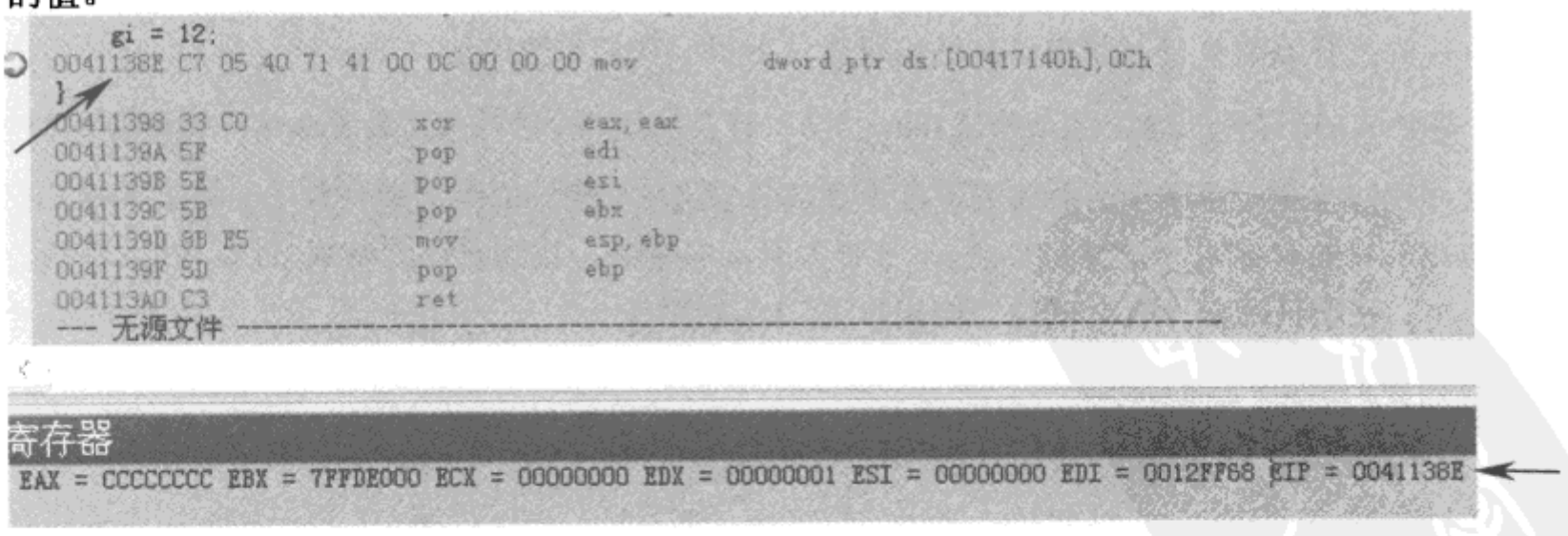


图 1.20

假定我们构造了一段 mov 指令，需要用 jmp 语句跳转到这段指令。这段指令执行完毕后，EIP 将指向下面的地址，即箭头指向的内存地址。如果不加处理，EIP 所指内存的值是不确定的。CPU 就会将该不确定值解释为对应指令，这将导致不可预料的行为。因此，我们在 mov 指令后面应该放一条指令，让程序回到正常流程。jmp 指令正好可达成该目的，见图 1.21。正常代码用 jmp 指令跳到我们构造的新代码，在新代码执行了 mov 指令后，用 jmp 指令跳回正常代码中 jmp 指令之后的代码，即 xxxx 代表的代码。

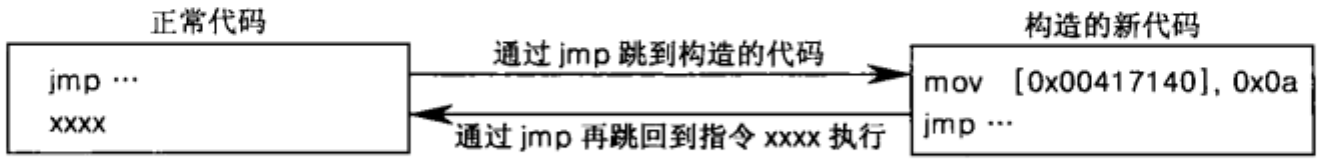


图 1.21

因为在构造的新代码中需要构造 jmp 指令，所以先来看 jmp 指令的机器码。到网上查找吗？其实很多时候，使用反汇编调试，我们不需求助他人。只需构造一段包含 jmp 指令的代码，然后通过反汇编跟踪查看机器码即可，见 DM1-3。**（这就是基础的力量，它可以生化万物，支撑我们的学习探索。）**

DM1-3

```
int i, gi;   //工程见“code\第 1 章\asmjmp”
void * address;
{
1       _asm {
2               mov  address, offset _lb1
3               jmp  address
4       }
5       i = 2;
6   _lb1:
7       gi = 12;
}
```

第 2 行将标签_lb1 即第 6 行语句的地址设定给变量 address。**（C 语言中嵌入的汇编能够识别高级语言中的符号，如变量 address，这与纯粹的汇编不同。）**

第 3 行跳转到 address 变量指向的地址，即执行第 6 行语句。

下面来看关键代码的反汇编：

```
                              jmp   address
004113C8 ff 25 cc 74 41 00    jmp   dword ptr ds:[004174cch]
```

在 C 语言程序中嵌入汇编

嵌入汇编有两种方式：① _asm{ …. }，在花括号中填写多条汇编语句；② _asm 一条汇编语句，如“asm mov eax, 1”。例如，C 语言中获取标签的地址值：

```
void * address;
_lb1:
gi = 12;
_asm mov address, offset _lb1;
```

变量 address 中保存的是标签_lb1 所在的地址，即“gi=12;”语句的地址。

机器码后 4 字节是 cc 74 41 00，正好是 jmp 指令中地址 004174cch 的小端机表示。因此，我们能够推断出代表 jmp 指令的操作码是 ff 25。现在构建机器码（见 DM1-4）并调用之，工程见“code\第 1 章\自己构造 mov 和 jmp 代码”。

DM1-4

```
   void main()
   {
1      void * code = buildCode();
2      _asm {
3          mov  address, offset _lb1
4      }
5      gi = 12;
6      printf("gi = %d\n", gi);
7      _asm  jmp code          //执行我们自己构建的代码
8      gi = 13;
9  _lb1:
10     printf("gi = %d\n", gi);  //打印的结果为 18，而不是 13
11     getchar();
   }
```

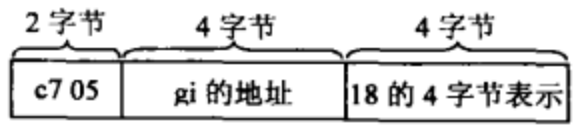
第 1 行，主函数调用 buildCode 获取新构建代码的首地址。第 2～4 行将第 10 行代码的地址赋值给 address。第 5～6 行对 gi 赋值为 12 并打印。第 7 行跳转到指针 code 指向的地址，即第 1 行分配的新代码首址。新代码将对 gi 赋值为 18，并跳转到 address 指向的地址，即第 10 行。第 10 行打印 gi 的新值 18。

我们来分析 buildCode 函数，它要构建的汇编指令如下：

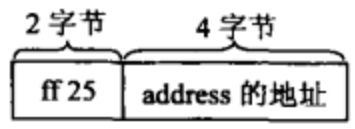
```
mov gi, 18;
jmp address
```

而这两条指令的机器码如下：

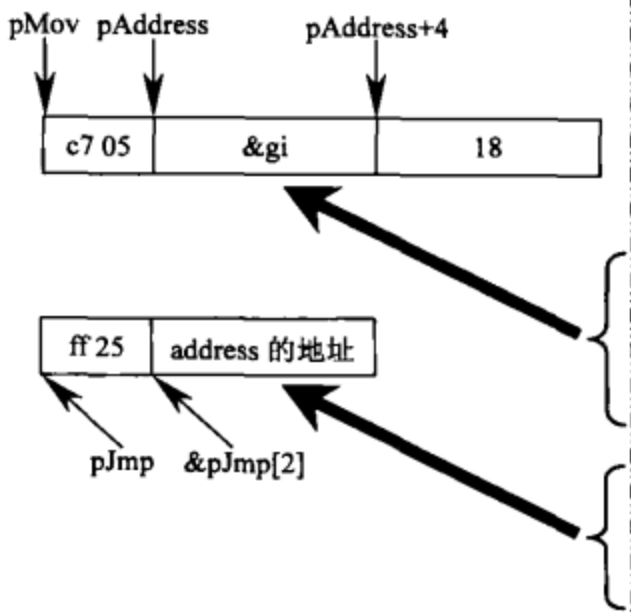
mov 指令



jmp 指令



这两条指令共 16 字节，因此需要用 malloc 分配 16 字节，然后将相关数据填入到其中，见 DM1-5。



DM1-5

```
void * buildCode()
{
   1   char * code = (char *)malloc(16);
   2   char * pMov = code;
   3   char * pJmp = code + 10;
   4   char * pAddress;
   5   //mov   gi, 18
   6   pMov[0] = 0xc7;
   7   pMov[1] = 0x05;
   8   pAddress = pMov + 2;
   9   *((int *)pAddress) = (int)&gi;
   10  *((int *)(pAddress + 4)) = 18;
   11  //jmp   address
   12  pJmp[0] = 0xff;
   13  pJmp[1] = 0x25;
   14  *((int *)(&pJmp[2])) = (int)&address;
   15  return code;
}
```

第 1 行，分配 16 字节存储这两条指令。第 2 行，pMov 指向 mov 指令的开始。第 3 行，pJmp 指向 jmp 指令头，因为 mov 指令为 10 字节，所以 pJmp=code+10。第 6～7 行，将 mov 的机器码填入，即 c7 05。第 8～9 行，将 gi 地址的 4 字节赋值给机器码第 3 字节开始。

第 10 行，将“18”这个要被赋值的 4 字节数设定给机器码第 7 字节开始的内存。第 12～13 行，设定 jmp 指令的机器码，即 ff 25。第 14 行，填写 address 的地址，从 jmp 指令的第 3 字节开始，填写 4 字节。第 15 行，将构造出的代码首址，即 code，返回。

第 9、10、14 行填写整数时，因为赋值指令采用小端机赋值，所以不需我们控制。

1.1.4 小结

通过本节，我们了解了调试环境的反汇编、监视窗口、内存窗口、单步、断点、mov 指令、全局变量赋值的反汇编、大端机/小端机，对指令建立了形象的认知，并直接构建了指令。

全局变量赋值引发的学习过程就是我们探索式学习很好的缩影，分为由浅及深的三阶段：深入剖析 → 部分修改 → 全新构建。我们要善于将自己的探索过程分解为递进的阶段，而这些阶段又

由诸多设问和实证组成、推进。通过这样的锻炼，我们能渐渐磨炼出自己的学习和探索能力。记住，设问和实证为心灵的自由插上了一对翱翔的翅膀。

1.2 理解指针和指针强制转换

1.2.1 指针和它丢失的类型信息

我们在学习 C 语言的时候，最难把握的是指针，特别是遇到指针强制转换时。通过反汇编，我们能够直接透视其本质，重新建立起清晰的认知。按照猜测实证的原则，首先应该对指针有一个简单的猜想，然后通过实验求证。

指针存储了内存的地址，同时指针是有类型的，如 int *、float *，那么，一个自然的猜想就是指针变量应该存储这两方面的信息：地址和指针类型。比如，就像下面的结构体：

```
struct pointer{
    long address;                    //地址
    int type;                        //指针的类型
}
```

先做一个简单的实验，打印 sizeof(int *)。出乎意料，这个值是 4！再打印 sizeof(float *)，还是 4。在 1.1 节中对全局变量赋值及 mov 指令的分析中，我们已经了解，全局变量的地址是 4 字节，即下面代码中的 00417140h。

```
gi = 12;
0041138E      mov   dword ptr ds:[00417140h],0ch
```

既然 4 字节是存储内存地址用的，反过来就说明指针并没有存储类型信息的地方。那么，为什么指针要有类型？为什么还要有指针类型强制转换？而且不转换，不同类型的指针无法赋值（**如 int *不能直接赋值给 float *，会产生编译错误**）。要实验就用反汇编直指其根本。反汇编一段简单代码，见 DM1-6。

如果还觉得不清晰，右键单击代码，在弹出的快捷菜单中选取“显示符号名”，见 DM1-7。

从反汇编结果可知，在第 1 条 mov 指令中，417168h 是 gi 的地址，而 417164h 是 pi 的地址（可通过监视窗体验证）。“pi = &gi;”就是通过一条 mov 指令将 gi 的地址放入一个 4 字节的变量 pi 中。指针确实就只有 4 字节（417164h 指向的 4 字节）存储了地址信息，且没有存储指针类型（即 int *）的代码。因为“pi = &gi;”只有一条 mov 指令与它对应，难道指针变量真的没有类型？如果没有类型，为什么 C 语言还有 int *、float *这些指针类型？

DM1-6

```
int gi;
int *pi;
void main()
{
    ...
    pi = &gi;
    00411452      mov   dword ptr ds:[00417164h], 417168h
    * pi = 12;
    0041145C      mov   eax,dword ptr ds:[00417164h]
    00411461      mov   dword ptr [eax],0ch
    ...
}
```

DM1-7

```
pi = &gi;
00411452    mov   dword ptr [pi(417164h)], offset gi(417168h)
* pi = 12;
0041145C    mov   eax,dword ptr [pi(417164h)]
00411461    mov   dword ptr [eax],0ch
```

在汇编语言中没有指针的概念，只有地址。为什么 C 语言包装了一个指针的概念？从表面看，指针除了存储地址信息，还有类型区别。**（思考问题往往要从源头去想，想想它的用途和存在价**

想想数据指针的用途，我们不是拿地址来玩，实际上要用它来访问内存。是否“访问内存，只要地址就可以了”？（**一定要想清楚细节，每个细节。**）把整数 1 存储到地址 0x12345678 中，需要从这个地址开始写几字节？1 字节、2 字节还是 4 字节？在 CPU 中，它们都是 1 的合法表示，对应到 C 语言的数据类型分别是 char、short 和 int。那么，到底要写几字节？实际上，该信息应该反映到赋值语句中。“* pi = 12;”语句的反汇编如下：

```
mov   eax, dword ptr [pi(417164h)]        //将 pi 中存储的 gi 的地址存入 eax
mov   dword ptr [eax], 0ch                //将 12（0ch）存储到 eax 指向的地址
```

在第 2 条赋值语句中，除了要赋的值（0ch）、被赋值的地址（在 eax 中）外，还有一个符号我们之前未注意——dword。是它回答了我们的问题：“写几字节？”dword 表明写 4 字节。到此我们发现，其实指针的类型信息决定了赋值/读取时写/读多少字节。

读/写多少字节的信息不是存放在指针变量中，而是放到了与该地址相关的赋值指令中，mov 指令中的 dword 指明了这个信息。

C 语言之所以要包装出指针的概念，是在汇编地址的内涵上增加了另一层含义，即读/写多少字节。不同类型指针，访问字节数不同。int *访问 4 字节，short *访问 2 字节。这样就方便我们操控一个地址，否则如果只有地址信息，每次访问它还要附加说明访问的字节数。这时，我们也能理解指针加/减 1 不是加/减 1 字节，而是加/减长度为该指针指向类型的长度的字节数。比如，int *指针加 1 是加 4 字节，short*指针则是加 2 字节。我们也能理解，void *类型的指针为什么无法进行加、减运算。因为它只是汇编语言中的地址，没有类型信息，加、减的时候不知道加/减多少字节。

DM1-8

```
short gi;
short *pi;
void main()
{
    ...
    pi = &gi;
    00413762     mov   dword ptr [pi(417164h)], offset gi(417168h)
    * pi = 12;
    0041376C     mov   eax, 0ch
    00413771     mov   ecx, dword ptr [pi(417164h)]
    00413777     mov   word ptr [ecx], ax
    ...
}
```

为了进一步确认“指针类型信息放入了赋值指令中”这一结论，我们再做一个实验：对 short 指针指向的地址赋值，见 DM1-8。

“pi = &gi;”的反汇编与前面是一样的，只有一条 mov 指令将 gi 的地址放入 pi 中，并没有指针类型信息的相关操作。

注意“*pi = 12;”对应的 3 条指令：

① mov eax, 0ch：将 12 放入 eax 中，eax 为 4 字节，12 存放在 eax 的低 2 字节即 ax 中。

② mov ecx, dword ptr [pi (417164h)]：将 pi 存储的地址即 gi 的地址放入 ecx 中（pi 的地址是 417164h，[417164h]中存储的是 gi 的地址）。

③ mov **word** ptr [ecx], ax：将 eax 的低 2 字节存储的内容（就是要赋值的 12）存入 ecx 指向的地址（即 gi 的地址）中。“word”表明了如果向 gi 所在地址存储，将写入 2 字节（前面对 int *指向地址的写操作涉及的 mov 指令是 dword，即 double word，为 4 字节）。我们再次看到，指针类型信息 short *体现在赋值指令 mov 中，而不是存放在指针变量中，指针变量只存放了地址。

还不够？我们继续这个实验，将 gi 改为 char 类型，将 pi 改为 char *类型。通过反汇编，则最后将 12 赋值给*pi：

```
mov   byte ptr [eax], 0ch
```

byte 指明了这个赋值语句是写 1 字节，因为 pi 为 char *类型。

至此，我们已经可以得出结论：

C语言的指针类型包含两方面信息：一是地址，存放在指针变量中；二是类型信息，关乎读写的长度，没有存储在指针变量中，位于用该指针读写时的 mov 指令中，不同的读写长度对应的 mov 指令不同。

看清了这一点，我们就要进一步直面许多人都有点发虚的指针强制转换，以及相关的转换安全性问题。

1.2.2 指针强制转换

问题，还是从问题开始我们的猜测和实证之旅，看如下代码：

```
int *pi;
short * ps;
...
pi = ps;
```

问题：

① 为什么，类型不同的指针变量不能赋值，如果将一个 short *指针变量 ps 赋值给 int *变量 pi（pi = ps），编译时会报错？

② 为什么，可以强制转换后赋值 “pi = (int *)ps;”，强制转换到底发生了什么？

对于第一个问题，1.2.1 节的分析已可给出答案。如果允许将 ps 指向的地址赋值给 pi，那么将来生成 “*pi = 2;” 赋值语句的指令时，必然生成 mov dword 指令，即写 4 字节。而 pi 指向的是 ps 指向的变量，该变量为 short，只有 2 字节，那么赋值 4 字节必然导致越界。所以，编译器为了帮助读者避免越界错误，便产生了编译错误。

对于第二个问题，我们只有去看看汇编才知道真相。还是先猜想：这种强制转换有类型信息的转换吗？根据 1.2.1 节的结论，指针变量本身不存储类型信息，只存储地址信息，那么，似乎强制转换只是形式上的，“pi = (int *)ps;” 应该只是将 ps 指向的值存入 pi 而已。所谓类型转换的效果，应该还是体现在对该地址指向内存的存取上吧？猜想到此，开始我们的实验。

在代码 DM1-9 中，整数 i 的地址赋值给了 int *、short*和 char *（即 pi、ps、pc）三种类型的指针变量。其中，对 ps 和 pc 的赋值进行了指针强制转换，最后 3 行是用它们对同一地址进行赋值操作。

DM1-9

```
int i;
int *pi;
short *ps;
char * pc;
void main(int argc, char* argv[])
{
    pi = &i;
    ps = (short *)&i;
    pc = (char *)&i;
    *pi = 0x1234;
    *ps = 0x1234;
    *pc = 0x12;
}
```

先看看主函数中的前 3 句对指针变量的赋值语句的反汇编：

```
pi = &i;
0041138e   mov   dword ptr [pi(417148h)], offset i (41714ch)
ps = (short *)&i;
00411398   mov   dword ptr [ps(417144h)], offset i (41714ch)
pc = (char *)&i;
004113a2   mov   dword ptr [pc(417140h)], offset i (41714ch)
```

这 3 条 mov 指令都是将 41714ch 即 i 的地址赋值给相关变量。只有赋值地址的 3 条指令没有对产生任何与类型相关的指令。可知，在指针变量赋值上，强制转换只是编译器的一个善意提醒，没有产生实际的指令。

再看后面 3 条表面上与强制转换无关的赋值语句的反汇编：

```
*pi = 0x1234;
```

```
004113ac      mov   eax, dword ptr [pi(417148h)]
004113b1      mov   dword ptr [eax], 1234h
*ps = 0x1234;
004113b7      mov   eax, 1234h
004113bc      mov   ecx, dword ptr [ps(417144h)]
004113c2      mov   word ptr [ecx], ax
*pc = 0x12;
004113c5      mov   eax,dword ptr [pc (417140h)]
004113ca      mov   byte ptr [eax], 12h
```

箭头所示的 3 条指令是同一地址在 3 种指针身份下的对应 mov 指令。注意其中的黑体部分，对 int * pi 指向地址的赋值语句是 mov dword，对 short * ps 指向地址的赋值语句是 mov word，对 char * pc 指向地址的赋值语句是 mov byte。指针强制转换的影响不是在转换的时候发生，而是在用转换后的身份去访问内存时体现到了指令中。

那么，什么情况下转换是安全的呢？就要看用这个转换后的身份去访问内存是否安全。简单地说有以下原则：

如果转换后指针指向的数据类型大小小于原数据类型大小，那么用该转化后的指针访问就不会越过原数据的内存，是安全的，否则危险，要越界。

对上例而言，“ps = (short *)&i;”强制转换后，用 ps 来访问内存是 2 字节，而 i 本身是 4 字节，所以不会越界，是安全的。而下面代码就是危险的：

```
short s;
int * p;
p = (int *)&s;
```

因为 p 指向的是 short 变量 s，大小为 2 字节，而 p 为整数指针，用它访问指向的内存将生成访问 4 字节的指令，访问会越界。

我们清晰了指针强制转换的本质，也了解了安全性原则，来看下面的例子（**我们真地能运用这些原则了吗？**）：

```
int * pi;
short si = 12;
pi = (int *) &si;
printf( "%d,    %x" , * pi, * pi);
```

打印的结果是什么？是“12, 0c”？笔者发现，即使讲完了以上分析后马上问学生该问题，也几乎无人发现其问题。（**在了解原则的情况下，还需要多多练习，多多怀疑，让猜测和实证将知识深深植入到思想和灵魂中，成为“活生生”的一部分。程序是手艺，而手艺是熟练和思想的交点。**）来看这段代码的反汇编结果：

```
-859045876    cccc000c
```

哪里有错？这次我们没有头绪，没有猜测。（**没关系，调试吧，观察吧，或许观察能够给我们线索。**）在“pi = (int *) &si;”语句上设置断点，然后用内存窗体查看 si 的值，看它是否为 12。先在监视窗体中输入“&si”，获得 si 的地址 0x0012FF54，然后在内存窗体中输入这个值，见图 1.22。

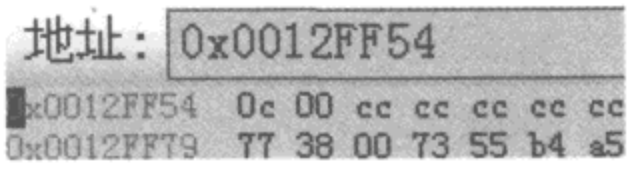


图 1.22

第 1 字节是 0c（0c 就是 12），第 2 字节是 00。（**也对，在小端机的内存中 0c 00 代表 0x000c，即 12。**）12 还可以用 4 字节表示，图中 0x0012FF54 开始的 4 字节 0c 00 cc cc，它代表的整数是

0xcccc000c。（**这个值就是打印结果的十六进制表示。找到问题了。**）将 si 的地址强制转换为 int * 类型，然后赋值给 pi（即 pi = (int *) &si;），那么*pi 会访问 4 字节。这时越界了，将 si 后的 2 字节纳入了范围，即 cc cc，它们与 0c 00 合在一起正好与结果吻合。（**我们违反了强制转换的原则，越界了。**）

1.3 函数调用和局部变量

1.3.1 计算指令中的跳转地址

函数是 C 语言为我们提供的一份大礼，它封装了许多细节。掌握一个函数的调用细节，对培养计算机的思维方式有莫大帮助，现在就开始这一探索之旅吧。

DM1-10

```
int Add(int x, int y){
    int sum;
    sum=x+y;
    return sum;
}
void main(){
    int z;
    z = Add(1, 2);
    printf("z=%d\n", z);
}
```

首先，给出 DM1-10 所示的代码。然后在 main()函数的“z=Add(1,2);”行设置断点，按 F5 键，调试运行并停止在该断点处。反汇编代码如下：

```
z = Add(1, 2);
00413762    6a 02          push   2
00413764    6a 01          push   1
00413766    e8 7a da ff ff  call   004111e5
0041376B    83 c4 08       add    esp, 8
0041376E    89 45 f8       mov    dword ptr [ebp-8], eax
```

执行了 call 指令后，程序就跳到 Add()函数。可知，call 指令完成了函数调用，而且执行 call 指令后，函数确实跳转到了 0x004111e5 地址。（**还是从机器码开始分析来寻找这个跳转信息。**）对于机器码 e8 7a da ff ff，先来猜测它的含义。习惯上，e8 似乎代表 call 指令，后面 4 字节似乎与跳转地址相关。怎样证明？一时也看不出后 4 字节是否与跳转地址相关，那么再加一个新的函数，来看是否第 1 字节代表 call 指令。添加一个 sub()函数，其调用反汇编如下：

```
sub(2, 1);
004137BC 6a 01             push   1
004137BE 6a 02             push   2
004137C0 e8 25 da ff ff    call   004111ea
```

可看到新的 call 指令的机器码是 e8 25 da ff ff。e8 依然在第 1 字节，后面 4 字节则变化了，因此可推定 e8 代表了 call 指令。下面看后 4 字节是否与跳转地址相关。

```
00413766 e8 7a da ff ff    call   004111e5
```

要跳转去的地址是 0x004111E5，而机器码后 4 字节是 7a da ff ff。这个数也与地址 0x004111E5 无关。根据对 mov 指令的经验，小端机内存中的 7a da ff ff 代表的值就是 0xffffda7a。（**但这个似乎也不对。**）我们需要分析可能的跳转机制。生活中的例子能给我们最好的提示，程序不过是自然和社会中各种机制的一种模拟罢了。生活中要找一个地方有两种方法：一是用经纬度这种方式绝对定位出位置，二是用一种指路方式“从这里左拐，然后右拐（右转弯），再左拐……”。这是用指路人位置作为坐标进行相对偏移。其实，计算机寻址（定位某个地址）也有这两种方式。绝对定位就是直接给出地址值，如 mov 指令中用全局变量地址直接赋值，相对定位就是用偏移量。那么，call 机器码中后 4 字节代表的 0xffffda7a 是否是偏移量呢？

从上面可知，call 指令在 0x00413766 地址处，加上 0xffffda7a 应该是一个非常大的数，从“call

004111e5”指令给出的提示可知是要跳转到 0x4111e5，这个似乎也不对。偏移量是区分正负的（**与往前还是往后跳转相关**），这个看上去很大的偏移量 0xffffda7a 是否是一个负数？这里需要用到一个基础知识：计算机中负整数的表示。

负整数的表示

补码表示：即其正整数值求反加 1。比如，-1 就是 1 求反，它只有最后一位是 0，其他位均为 1，再加 1，那么所有位均为 1，即 0xffffffff。对有符号数，最高 bit 为 1，则说明这它是一个负数。

如果对一个补码形式的负数求其正数值，就是求反加 1。比如，对-1 求其正数值，就是对 0xffffffff 求反，即 0，再加 1，就是 1。

0xffffda7a 最高位为 1，是负数。将其求反加 1 后，该值的二进制表示是 1111 1111 1111 1111 1101 1010 0111 1010；求反，为 0000 0000 0000 0000 0010 0101 1000 0101；再加 1，其十六进制表示为 0x2586。来看 call 指令，打开计算器做一个减法。

```
00413766 e8 7a da ff ff          call   004111e5
```

call 指令的地址 0x00413766 - 偏移量 0x2586 = 0x4111e0

这个减法的结果是 0x4111E0，理论上应该是跳转到的地址 004111E5，如上面虚线箭头指示的地址。可是两者不相等。（还好，这两个值已经非常接近了，它们只差 5 字节。）这 5 字节是怎么回事？（**我们需要思考、分析，不要放过任何一个理所当然的要素。很多时候，分析程序有点像脑筋急转弯，你忽略掉的理所当然的因素恰恰是关键点。**）在用相对偏移量寻址时是怎么计算的？是基址+偏移量。基址的选择必须是 call 指令的地址吗？是的，但并非必须。call 指令有 5 字节，而刚才的计算恰好差了 5 字节。这是巧合吗？

如果我们尝试将基址从 call 的起始处放到结束处来看，即 call 的下一条指令的起始地址来看新计算结果，那么 call 指令转跳的地址是 0x00413766+call 指令长度 5–偏移量 0x2585 = 0x4111e5。0x4111e5 就是 call 指令指示出来的跳转地址。正确了。

call 指令的偏移量寻址

x86 系列 CPU 的 call 指令的寻址方式为：用与 call 指令相关的偏移量定位转跳到的地址。
偏移量计算如下：偏移量 = 转跳到的地址 －call 指令后一条指令的起始地址。

1.3.2 返回故乡的准备

函数的特点是调用后会返回，那么如何得知这个返回地址？它是存储在函数代码中吗？就像 mov 指令中的地址那样？如果这样，岂非只能返回到一个固定地址？因为代码在运行期是不变的，但函数可在不同地方被调用，这意味着可返回到不同地址。那么，函数返回地址是否可以像参数一样传递给函数？

我们来看 call 指令除了跳转外，还对系统施加了什么影响。首先，call 指令如下：

```
00413796   e8 4a da ff ff      call   004111e5
0041379b   83 c4 08            add    esp, 8
```

call 的返回地址是 0x0041379b。call 执行后是否存储了该地址？如果是，该地址存储在何处？一般而言，数据或者存储在内存中，或者存储在寄存器中。

如果返回地址存储在内存中，那么从哪里获取该内存地址？不外乎指令、内存和寄存器。放在指令中已不可能，因为 call 指令的 5 字节只能放下偏移量。放到内存更不行，要访问该内存，其地址又从何获取？这回到了本欲解决的问题。只剩下从寄存器获取。

如果返回地址存储在寄存器中，那么 call 执行后某个寄存器中就是返回地址。

综合以上两点，某个寄存器中的值或者是返回地址，或者指向某内存地址，它存储了返回地址。（**那就观察寄存器的变化来实证吧**。）在 call 指令处设置断点，按 F11 键，单步执行一次，这时 call 执行后变化的寄存器值在图 1.23 中标注出来。变化的寄存器有两个，它们是我们的关注点。EIP 中存储的是 call 跳转到的指令的地址，从反汇编代码窗体中可证明（见图 1.24），箭头所指的位置就是 EIP 指示的地址 0x004111e5。因此，EIP 寄存器与我们的分析无关。

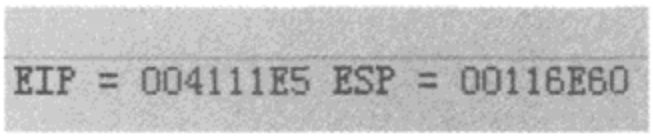


图 1.23

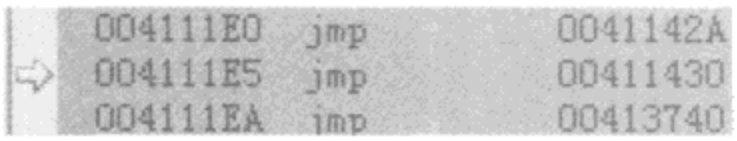


图 1.24

现在关注另一个变化的寄存器 ESP，其值为 0x00116e60。它或者是返回地址，或者是存放返回地址内存的地址。显然，它并非 call 指令的返回地址 0x0041379b。那么，它指向内存中存储的值是什么？在内存窗体中输入地址 0x00116e60，结果见图 1.25。根据小端机原则，内存中 9b 37 41 00 代表的值是 0x0041379b，这正是 call 指令返回地址。可知，call 指令将返回地址保存在内存中，而且 ESP 寄存器指向了该内存。1.3.6 节将讨论该返回地址如何被使用。

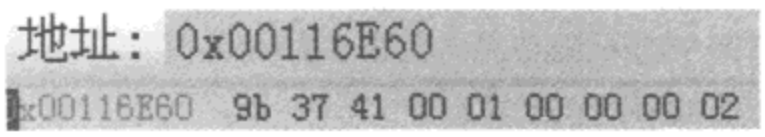


图 1.25

栈（stack）

栈是一个先进后出的数据结构，先压栈的数据后出栈。出栈操作总是将当前栈顶的数据弹出。如果当前栈为空，压栈 2 后，栈顶存储的就是 2，再压栈 4，那么栈顶存储的就是 4。执行出栈操作，就会将栈顶的 4 弹出，此时当前栈顶存储的就是 2，如果再次执行出栈操作，2 将出栈。这就是先进后出（详细可参阅数据结构的书籍）。

ESP 寄存器（Extended Stack Pointer）存储栈顶内存地址。我们可在调试环境中查看寄存器窗体，获取当前的 ESP 值，然后将该值输入到内存窗体中（用十六进制形式），即可查看栈中数据。

压栈的指令是 push，如“push 1”将 1 压栈，这时 ESP 指向存储 1 的内存地址。再执行“push 0c”，则 0c 将存入当前栈顶，ESP 将指向存储 0c 的内存地址。每次压栈，32 位机上都将用 4 字节存储压栈值。在 x86 中，push 指令使 ESP 的值减 4。换句话说，越压栈，栈顶地址越小。有的 CPU 中，压栈是增加 ESP 的值。

出栈指令为“pop 寄存器”。比如，“pop EAX”就是将当前 ESP 指向的栈顶中存储的值赋值给 EAX，并且 ESP 的值加 4。

call 指令功能

call 指令相当于以下两条指令的组合：

```
push   返回地址
jmp    函数入口地址
```

其中，返回地址通过第一步保存在栈上，函数返回时会用到。

1.3.3 给函数传递参数

函数需要参数，现在来看参数是如何传递的。

还是将之前的语句“z = Add(1, 2);”进行反汇编：

```
004139a2   6a 02              push   2
004139a4   6a 01              push   1
004139a6   e8 3a d8 ff ff     call   p (4111e5h)
004139ab   83 c4 08           add    esp, 8
```

两条 push 指令应该是传递 2 和 1 两个参数，通过栈来传递。我们还是应该实证。最简单的方法就是将该语句改成“z=Add(2,1);”，调换参数的顺序。再次反汇编，则先压栈 1 后压栈 2。基本可证明这个猜测，甚至还推断出 C 语言的参数传递是从右往左压栈传递参数。

下面单步分析执行这三条指令时栈的变化。在“push　2”处设置断点，在该断点中断，此时 ESP 的值是 0x00116e6c，将该值输入内存窗体，然后用滚动条往上滚动一行，我们查看整个栈的情况，其结果见图 1.26。

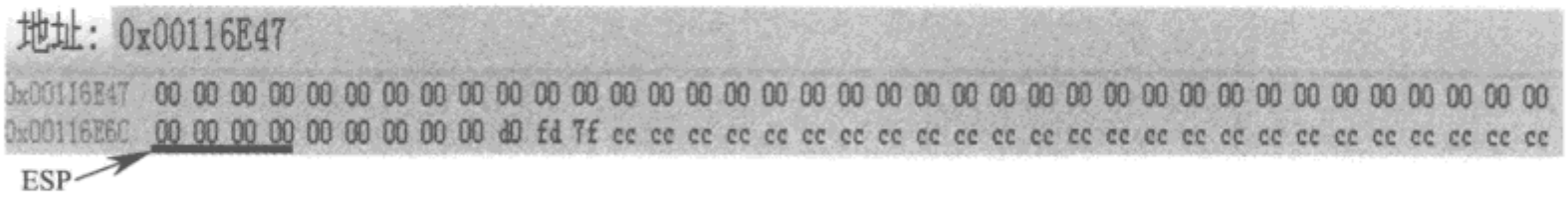


图 1.26

单步执行“push　2”，查看 ESP 的值为 0x00116e68，比执行前的 0x00116e6c 少了 4 字节（x86 栈顶地址小于栈底，压栈则使栈顶地址减小）。内存窗体中显示栈布局，见图 1.27。此时 ESP 指向的栈顶所存值如图中黑线所示为 2（小端机内存值 02 00 00 00 代表 2）。

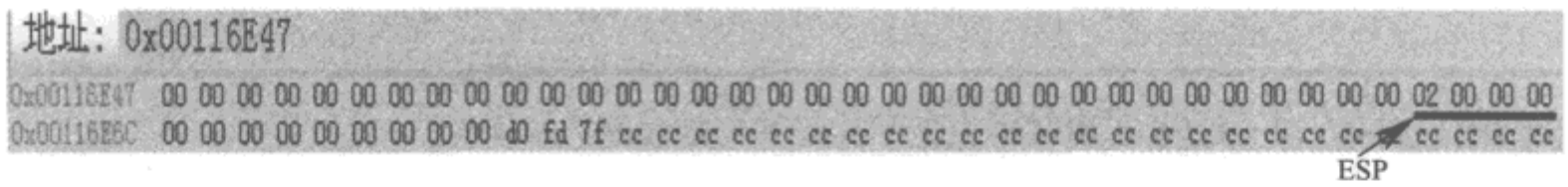


图 1.27

再次单步执行“push　1”指令，查看 ESP 中的值为 0x00116e64，比执行前的值 0x00116e68 也少了 4 字节。而内存窗体中显示栈布局见图 1.28。此时 ESP 指向的栈顶所存值如图中黑线所示为 1（小端机内存值 01 00 00 00 代表 1）。

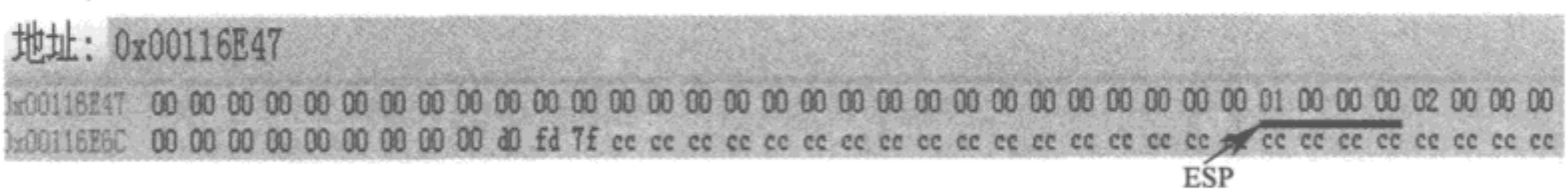


图 1.28

单步执行 call 指令后，栈内存见图 1.29，ESP 又减少了 4 字节，为 0x00116e60，其指向的内容 ab 39 41 00 正好是 call 指令的返回地址 0x004139ab。（**该地址见本节开始的反汇编，是 call 指令后的指令“004139ab add esp,8”的地址。**）

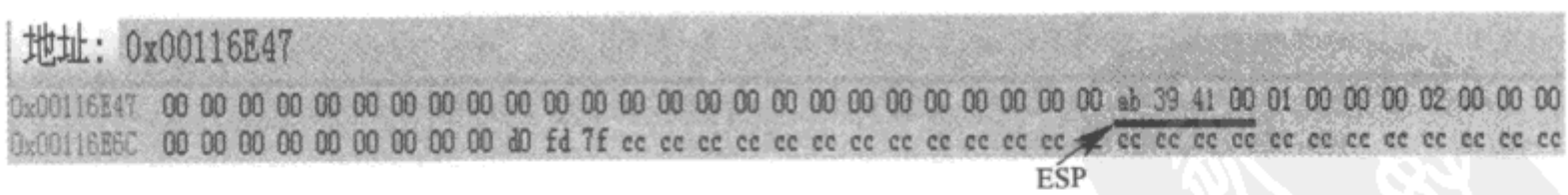


图 1.29

由此我们验证了参数传递时，栈顶地址不断减小的过程，并查看到参数压栈到栈内存的布局。

1.3.4　函数获取参数

被调函数如何获取栈上 1 和 2 这两个参数？我们先来看执行了 call 指令后的栈映像，见图 1.30，

反映了图 1.26～图 1.29 的结果。

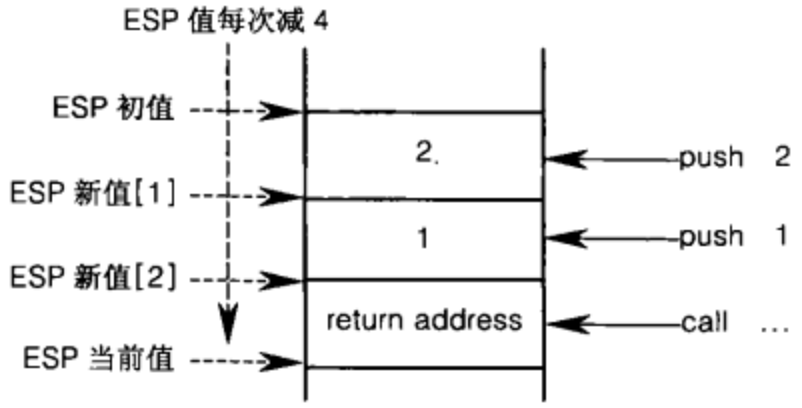


图 1.30

指令“push 2”执行后，ESP 指向存储 2 的地址；指令“push 1”执行后，ESP 减 4 指向存储 1 的地址，然后执行 call 指令，返回地址被压栈，ESP 内容再次减 4，指向保存返回地址的内存。

要获取到栈上存储的参数 1 和 2，必须得到存储 1 和 2 内存的地址。如何得到它们？是将它们再次压栈传递吗？肯定不行。存储地址的内存的地址又怎样获取？我们观察，1 和 2 是连续存放的，只要获取一个基点，就能通过偏移量计算出 1 和 2 存储的地址。从哪里获取基点的地址？它似乎不应该是保存在内存中，否则该内存的地址又无法获取。在计算机中，除了内存，就是寄存器用来存储信息，如在图 1.30 中，ESP 寄存器指向当前栈顶。图中一槽内存是 4 字节，那么 1 的存储地址就是 esp+4，2 的存储地址就是 esp+8。由于 ESP 将随栈的变化而变化，所以为了计算简便，可以将该值存储到另一个寄存器，如 EBP（Extended Base Pointer，扩展基址指针寄存器）。如果将 ESP 存入 EBP，那么刚才的两个地址就是 ebp+4 和 ebp+8。

不过由于 EBP 本身存储了值，如果直接赋值修改它，当 EBP 原值要使用时，就没有办法了。比如，函数 f1()调用函数 f2()，如果 f1()和 f2()都用 EBP 存放基点地址，f2()后执行，修改了之前 f1()设定的 EBP 值。当 f2()返回时，如不还原 EBP，f1()用 EBP 做基点就要出错。因此，必须有一个保存和恢复 EBP 值的过程。下面代码完成了这一过程，将 EBP 压到栈上，最后用“pop ebp”指令将栈顶存储的 EBP 值弹出来，赋值给 EBP，从而还原。

```
push   ebp
...
pop   ebp
```

因此可以猜想，在被调用的函数开始处存在如下代码：

```
push   ebp
mov   ebp, esp
```

它保存了 EBP 的值，并将 ESP 的值赋值给 EBP。压栈 EBP 后，ESP 将再次减 4 字节，见图 1.31。

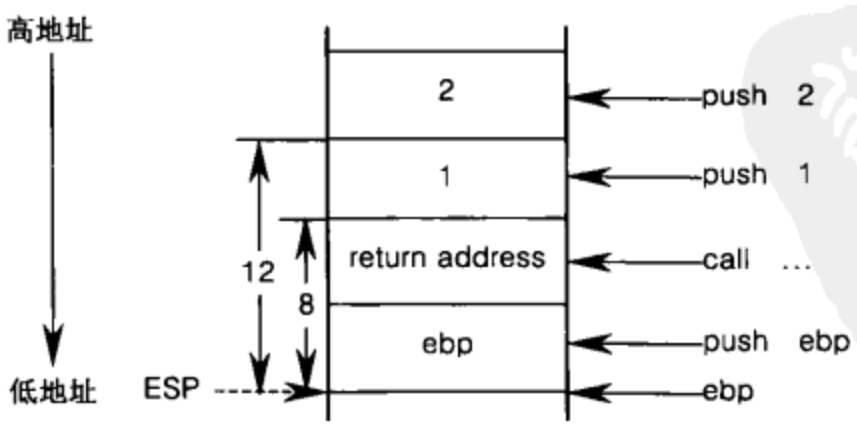


图 1.31

DM1-11

```
int Add(int x,int y)
{
  00411430    push    ebp
  00411431    mov     ebp, esp
  00411433    sub     esp, 0cch
  00411439    push    ebx
  0041143a    push    esi
  0041143b    push    edi
  0041143c    lea     edi, [ebp+ffffff34h]
  00411442    mov     ecx, 33h
  00411447    mov     eax, 0cccccccch
  0041144c    rep     stos dword ptr es:[edi]
  int sum;
  sum=x+y;
  0041144e    mov     eax, dword ptr [ebp+8]
  00411451    add     eax, dword ptr [ebp+0ch]
}
```

此时用 EBP 来计算参数 1 和 2 的存储地址。存储 1 的地址是 ebp+8，因为这时距离 1 存放的地方有 EBP 和返回地址，共 8 字节。存储 2 的地址是 ebp+0ch（即 12）。现在反汇编 Add()函数，查看是否与我们分析的一样。

验证 1：见 DM1-11。

反汇编代码中，黑体标注的前 2 行正如我们分析的那样。最后两条指令（黑体标注的）是“sum=x+y;”语句的对应指令。mov 指令将地址为 ebp+8 的内存中存储的值赋给 EAX，从图 1.31 看，ebp+8 正好是存储 1 的地址，符合之前分析，执行后，EAX 的值为 1。第 2 条 add 指令将内存地址为 ebp+0ch 中存的值，就是 2，与 EAX 相加并将其和存回 EAX。执行完，该指令后 EAX 就是参数 1 与 2 的和 3。一切如猜测。

验证 2：在单步状况下停在最后两条指令处，用监视窗体查看&x 和&y，获取两个参数的地址；然后通过寄存器，将 ebp+8 和 ebp+0ch 的值求出，与监视窗体中的&x 和&y 值比较，发现 ebp+8、ebp+0ch 确实是 x 和 y 的地址。

1.3.5 局部变量

局部变量在哪里分配？其特点与参数一样，当函数调用完毕就不再使用，所以仿效参数，就是将其分配在栈上。栈上方已经被参数等使用，我们只有使用栈更低地址的空间，也就是继续压栈分配局部变量，见图 1.32，即 ebp–4 指向的内存。

反汇编，来看这个假设是否正确。

```
sum=x+y;
0041144e    mov   eax, dword ptr [ebp+8]
00411451    add   eax, dword ptr [ebp+0ch]
00411454    mov   dword ptr [ebp-8], eax
```

从前面分析已知，第 1、2 两条指令将 x 和 y 求和，并放入 EAX 中。最后的指令将 EAX 的值存入 ebp–8 指向的内存中（断点后单步执行到第 3 条指令，用监视器查看&sum 的值确实是 ebp–8），并非 ebp–4，见图 1.33。（**请用内存窗体和寄存器窗体单步跟踪，查看这些相关内存的变化。**）奇怪的是，与猜测一致，在 Visual C++ 6.0 中，反汇编代码如下：

```
00401038    mov   eax, dword ptr [ebp+8]     //获取参数 x
0040103b    add   eax, dword ptr [ebp+0ch]   //获取参数 y，将 x、y 求和放入 EAX
0040103e    mov   dword ptr [ebp-4], eax     //将和存到局部变量 sum，其地址是 ebp-4
```

在 VS 2008 中，为了防止溢出攻击，所以产生了该现象。详细见习题 4。

不过，至此我们应该有一个简单的结论：在使用了 EBP 寻址的函数中，ebp+偏移量就是参数的地址，ebp–偏移量就是局部变量的地址。

1.3.6 返回故乡

函数执行完毕就要返回调用的地方，从 1.3.2 节知道，call 指令将其后指令的地址压栈保存，我们要查看它如何用该地址返回。首先介绍返回指令 ret。

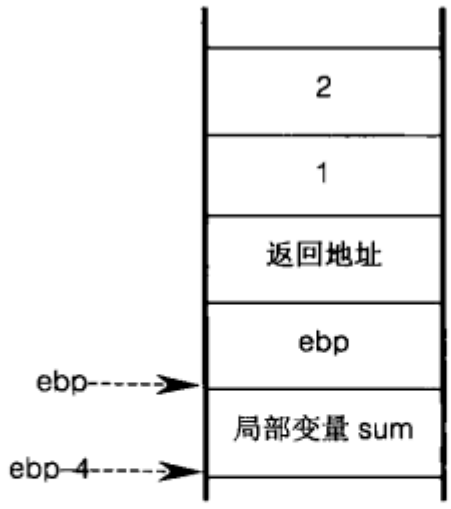


图 1.32

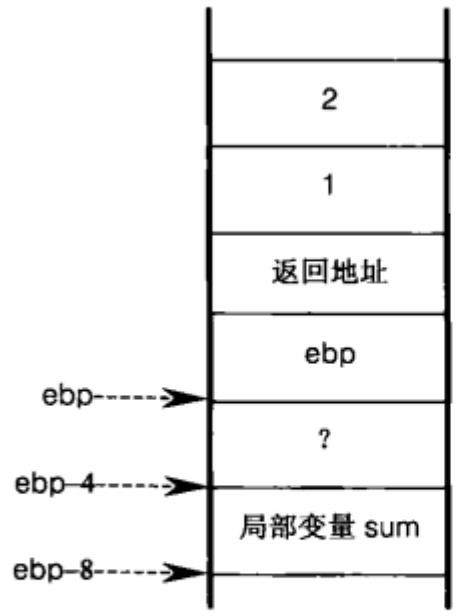


图 1.33

ret 指令：将栈顶保存的地址弹入指令寄存器 EIP，相当于“pop　eip”，从而让程序跳转到该地址。执行 ret 指令后，寄存器 EIP（存储了栈顶中保存的那个地址）和 ESP（在 32 位 x86 中加 4）的值有变化。

从 ret 的功能看，编译器要保证执行该命令时，ESP 正好指向 call 指令压栈保存返回地址的那段内存，我们跟踪的代码见 DM1-13。push 指令的压栈顺序是 ebx → esi → edi，pop 指令的出栈顺序是 edi → esi → ebx。正好体现了栈的先进后出原则。push 与 pop 之间的指令没有影响 ESP 的值（可在 0041143c 指令处观察 ESP 的值，并在 0041145a 处观察 ESP 的值，两者相同），所以这两组指令完成了对 EBX、ESI、EDI 的值的保存和恢复。

DM1-13

```
{
  00411430      push   ebp
  00411431      mov    ebp, esp
  00411433      sub    esp, 0cch
  00411439      push   ebx
  0041143a      push   esi
  0041143b      push   edi
  0041143c      lea    edi, [ebp+ffffff34h]
  00411442      mov    ecx, 33h
  00411447      mov    eax, 0cccccccch
  0041144c      rep    stos dword ptr es:[edi]
  int sum;
  sum=x+y;
  0041144e      mov    eax, dword ptr [ebp+8]
  00411451      add    eax, dword ptr [ebp+0ch]
  00411454      mov    dword ptr [ebp-8], eax
  return sum;
  00411457      mov    eax, dword ptr [ebp-8]
}
0041145a    pop   edi
0041145b    pop   esi
0041145c    pop   ebx
0041145d    mov   esp, ebp
0041145f    pop   ebp
00411460    ret
```

最后 3 条斜体标注的指令中，0041145d 处的指令将 EBP 赋值给 ESP。注意，函数执行 00411431 处指令后，栈映像见图 1.33。当执行 0041145d 处指令时，EBP 未改变。图 1.34 给出了执行其前后的栈映像。

在执行“mov　esp, ebp”指令后，ESP 正好指向保存 EBP 值的栈顶，因此其后的“pop　ebp”指令弹出保存的 EBP 旧值给 EBP，恢复 EBP，使得 ESP 加 4，指向保存返回地址的内存，见图 1.35。这时执行 ret 指令，正好将栈顶保存的返回地址弹给 EIP，完成了返回。

我们用调试器来跟踪观察以上分析，完成实证过程。首先，call 指令执行完后，进入 Add()函数时程序的状态见图 1.36，包括代码窗体、内存窗体和寄存器状态，call 压栈的返回地址为 0x004139ab。

执行“push　edi”指令后的 ESP 的值见图 1.37。然后，在执行“pop　edi”前观察此时 ESP 的值，与图 1.37 中的 0x00116D84 相同，见图 1.38。

读者可以自己跟踪之前 3 条 push 指令压栈到内存中的值，然后配合 ESP 查看指向的内存，查看 pop 指令恢复这些寄存器的过程，体验先进后出的栈结构在保存和恢复寄存器的值所起的作用。

下面省略“mov　esp, ebp”和“pop　ebp”两条指令的观察（读者可仿效上面过程自己查看），直接在 ret 指令前停下，查看状态，见图 1.39。ESP 恢复到了 0x00116E60，与图 1.36 执行 call 指令

后 ESP 的值一样，说明这时栈已经还原到刚刚进入 Add()函数时的状态了，观察 ESP 指向的内存，图 1.39 中正好是 0x004139ab，即 call 指令压栈的返回地址。这时执行 ret，就会将当前栈顶（esp 指向的内存）保存的地址 0x004139ab 压入 EIP，并将 esp 加 4，见图 1.40。

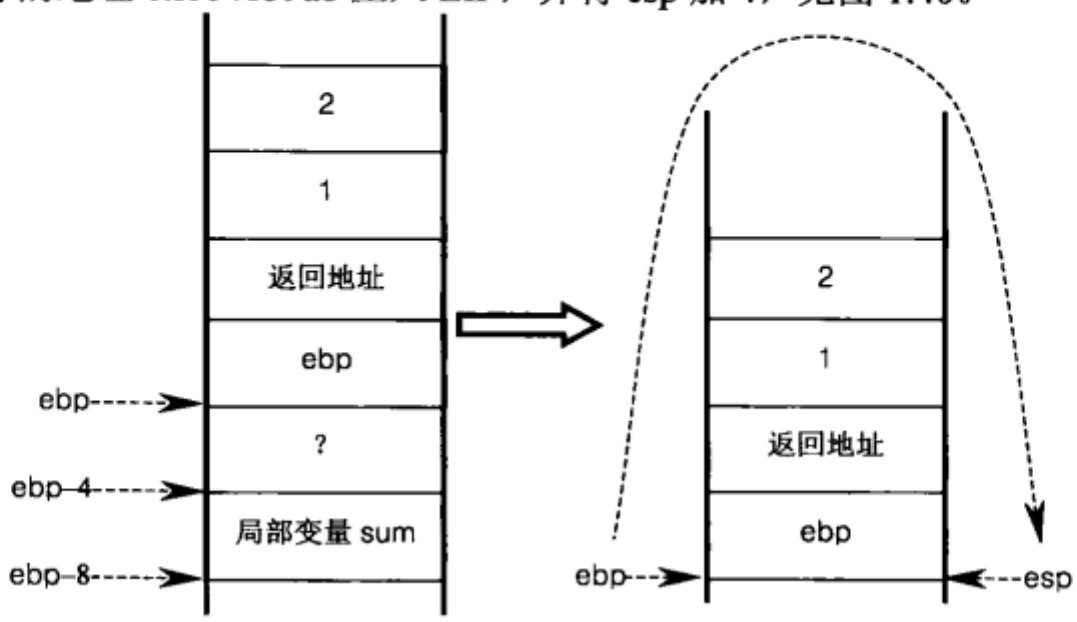


图 1.34

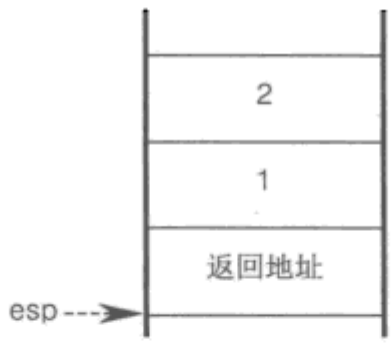


图 1.35

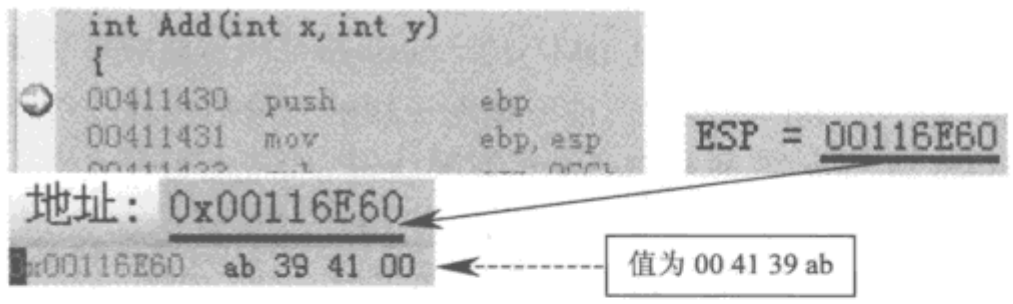


图 1.36

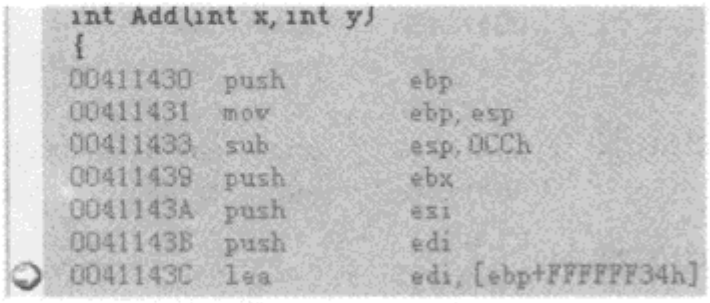


图 1.37

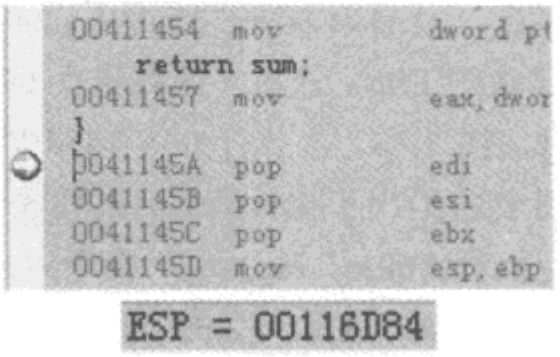


图 1.38

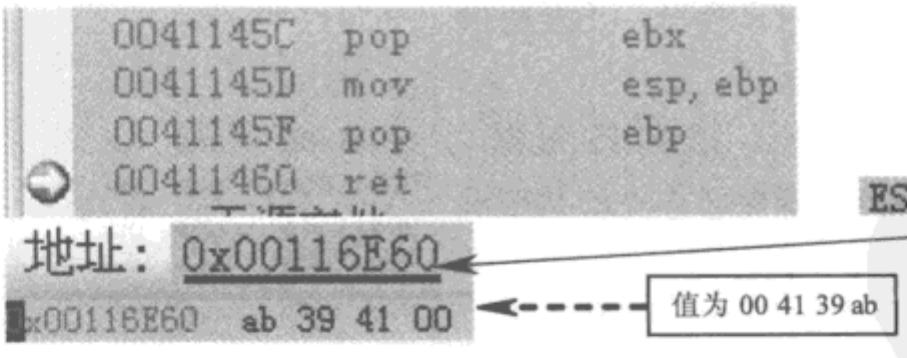


图 1.39

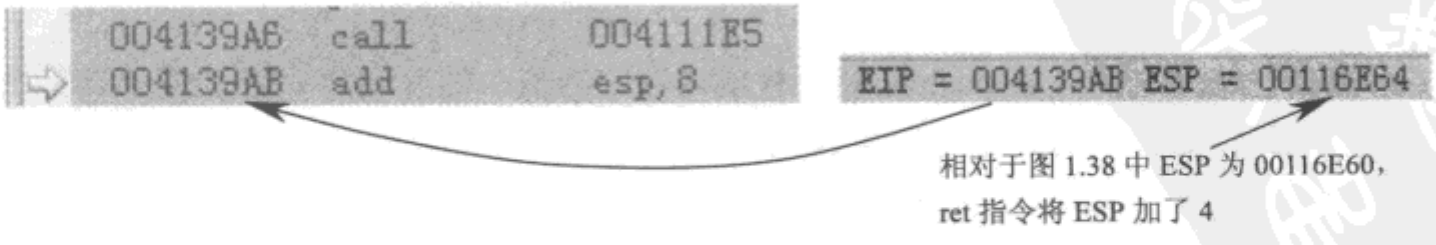


图 1.40

1.3.7 返回点什么

函数可有返回值，那么如何返回它们？通过寄存器。

函数返回值：编译器习惯上用 eax 作为存储返回值的寄存器，被调用方在 ret 前设定 eax，返回后，调用方从 eax 获取到该值。

返回小于4字节的数

例如，对于代码 DM1-14，第 1 条 mov 指令将 ebp–8 指向的内存（即 sum 变量的内存，见 1.3.5 节）中存储的值赋给了 EAX，之后的指令不修改 EAX 的值（**在寄存器窗体中查看该状况**）。

DM1-14

```
    return sum;
    00411457      mov   eax, dword ptr [ebp-8]
}
    0041145a      pop   edi
    0041145b      pop   esi
    0041145c      pop   ebx
    0041145d      mov   esp, ebp
    0041145f      pop   ebp
    00411460      ret
```

调用方代码如下：

```
    z = Add(1, 2);
004139a2      push   2
004139a4      push   1
004139a6      call   004111e5
004139ab      add    esp, 8
004139ae      mov    dword ptr [ebp-8], eax
```

call 指令返回后，最后一条指令将 EAX 的值赋值给了 ebp–8 指向的内存。之前我们已经有这样的概念，ebp 减一个偏移量就是局部变量的地址，该指令说明它将返回值赋给了一个局部变量。如果在该指令设置断点，查看 ebp–8 的值，并在监视窗体中查看&z，就会发现两者的值相同。

返回结构体

对于返回值的传递大家还有问题吗？似乎一切都很好，但仔细分析，EAX 只能返回长度小于等于 4 字节的数据，如果返回值大于 4 字节怎么办？double 类型（8 字节）、用户自定义数据类型——结构体（有多少成员就有多大）的数据大于 4 字节。对于这些“过大”的数据怎样返回？（还是先猜测。）再大的数据，只要拿到地址就能访问。那么，是传给函数这个“大”数据的地址由函数去填写其内容，还是在函数中分配这个“大”数据的内存并返回其指针呢？如果是后者，会存在一个问题：该数据在何处分配？因为这个数据与局部变量一样只在函数中有效，自然会在栈上分配。可是当函数返回后，函数分配的内存“逻辑上”无效了，用“无效”指针访问于理不通，见图 1.41。

如果是调用者先分配一块内存，然后将其指针传入给函数由它写入返回值，那么被调函数返回时该内存并不失效，似乎可行，见图 1.42。其中保存返回值的内存在栈的上部分配，其地址为 0x12ff58，然后该地址作为一个参数被压栈传递给被调用函数。

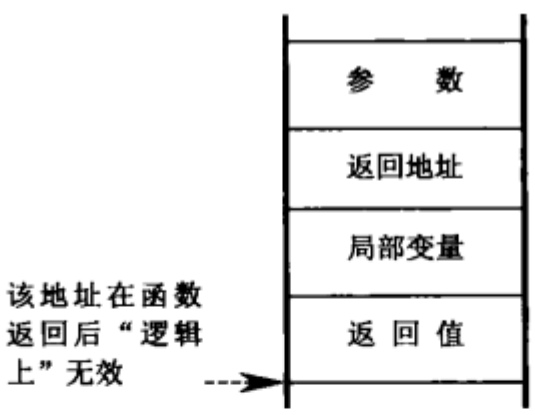


图 1.41

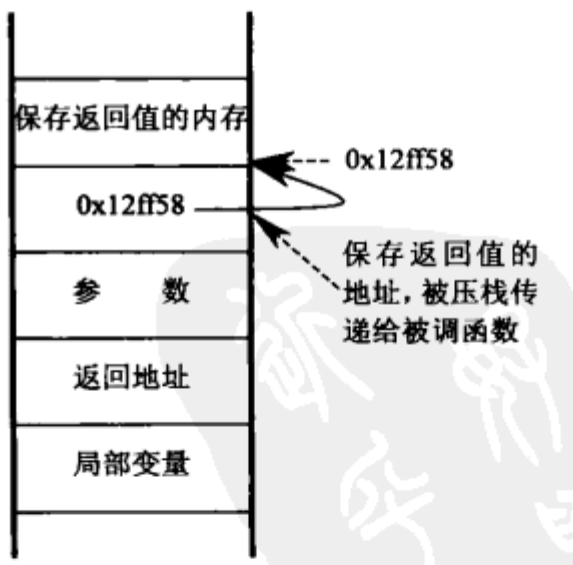


图 1.42

有了猜测，我们需要用实验来证明，用结构体来验证。需要用到结构体，请大家先参照 1.4 节理解结构体的相关知识。（**在学习中经常会遇到这样的状态，在研究一个知识点时，发现必须先了**

解另外一个知识点，因此需要先去解决它，然后回头继续当前问题。某些时候，这样的跳转会发生多次，只要是在我们能力之内，跳转下去直到把一条知识线索都搞通也是一种学习方法，并非一定要所谓的“系统化”。）

相关代码见 DM1-15。在 main()函数的“r=myfunc();”处设置断点，执行并反汇编如下：

DM1-15

```
struct myrd{
    int i1;
    int i2;
};
myrd myfunc(){
    myrd r1;
    r1.i1 = 1;
    r1.i2 = 2;
    return r1;
};
void main(int count, char** args){
    myrd r;
    r = myfunc();
}
```

```
    myrd r;
    r = myfunc();
0040120E    call        00401180
    ...
```

在 call 之前并没有如想象那样传入保存返回值内存的地址。（怎么回事？）在 call 指令处跟踪进函数（step into）：

```
    myrd r1;
    r1.i1 = 1;
0040119e    mov   dword ptr [ebp-0ch], 1
    r1.i2 = 2;
004011a5    mov   dword ptr [ebp-8], 2
    return r1;
004011ac    mov   eax, dword ptr [ebp-0ch]
004011af    mov   edx, dword ptr [ebp-8]
    ...
    ret
```

0040119e 和 004011a5 处的两条指令将 1 和 2 赋值给了 r1 的两个成员变量，而 004011ac 和 004011af 处的两条指令又将 r1 的两个成员变量赋值给 EAX 和 EDX。难道要返回的 8 字节的结构体是通过 EAX 和 EDX 返回的吗？如果是，在 ret 指令执行时，这两个寄存器应该是 1 和 2。我们在之后的 ret 指令处设置断点，发现运行到 ret 时，EAX 和 EDX 的值确实如此。

下面来检查调用方的代码，见 DM1-16。

DM1-16

```
myrd r;
r = myfunc();
0040120e    call   00401180
00401213    mov    dword ptr [ebp+ffffff24h], eax
00401219    mov    dword ptr [ebp+ffffff28h], edx
0040121f    mov    eax, dword ptr [ebp+ffffff24h]
00401225    mov    dword ptr [ebp-0ch], eax
00401228    mov    ecx, dword ptr [ebp+ffffff28h]
0040122e    mov    dword ptr [ebp-8], ecx
```

call 指令后的两条指令将 EAX 和 EDX 的值分别赋给了地址为 ebp+FFFFFF24h 和 ebp+FFFFFF28h 的内存，确实是 EAX 和 EDX 用于返回结构体的值。剩下要证明的就是这个返回值赋给了变量 r。我们发现，EAX 的值最终赋值给了 ebp–0ch，而 edx 最终赋值给了 ebp–8，自然猜想 ebp–0ch 这个更小的地址应该是结构体 myrd 中靠前定义的成员变量 i1 的地址，也就是结构体 r 的首部地址。在监视窗体中输入如下表达式验证猜想，见图 1.43，可知 r 的地址和 ebp–0ch 是一个值。（在表达式中用(void *)强制是为了用十六进制显示。）同理，请读者自己证明 ebp–8 就是 r.i2 的地址。

ebp+ffffff24h 和 ebp+ffffff28h 两个地址的内存似乎被用来中转，EAX 和 EDX 保存的 8 字节返回值先赋给这两个地址，然后从它们中读出，赋值给变量 r。那么，这块内存在哪里？它们有何用？我们先将上面两个地址转换成容易读懂的东西，最高位为 1 就是负数，所以这两个地址其实可以转换为“ebp–xxx”的形式。将 ffffff24h 和 ffffff28h 求反加 1，获得其正值，分别为 dch 和 d8h。这两块内存的地址就是 ebp–dch 和 ebp–d8h。它们明显小于 r 的地址 ebp–0ch，可知这块内存位于局部变量 r 的下方，似乎是一个临时分配存储返回值的缓冲区。

我们不解的是，为何不直接将 EDX 和 EAX 中的返回值直接赋值给 r？我们用的是 Debug 版，代码没有优化。这恰恰证明了之前的猜想：调用方分配一块缓冲存储返回值。优化的 Release 版本会用这个中转缓冲区吗？选择调试界面中的 Release 菜单（在 VC 6.0 中，调试 Release 需要专门设

置，VS 2008 不需要），见图 1.44，反汇编的代码见 DM1-17。

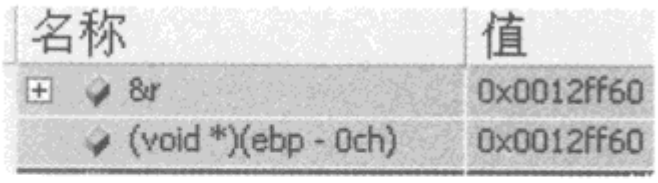


图 1.43

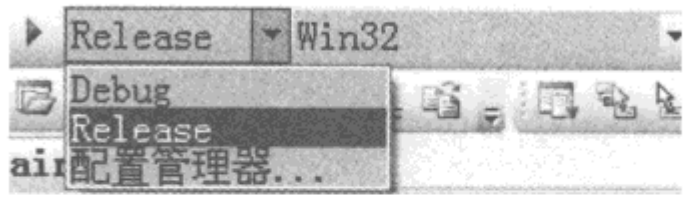


图 1.44

我们惊讶地发现，call 之后那些赋值语句全部消失了！为什么将返回值赋值给 r 的逻辑给“优化”掉了？（**优化的一个原则是看有用否。**）r 是 main()的局部变量，而 main()并未使用该局部变量，当然不用赋值。那么，只要让编译器认为变量 r 有用，它就不会被优化掉。如果将 r 定义为全局变量，要判断 r 是否有用就必须分析所有代码。（**因为全局变量可被所有代码使用**）。为了提高编译速度，编译器会“偷懒”，凡是对全局变量的操作会认为有用，不优化。

DM1-17

```
myrd r;
r = myfunc();
getchar();
00401085        call
00401474
}
0040108a        xor   eax, eax
```

代码见 DM1-18。果然，在 call 指令执行后，EAX 和 EDX 直接赋值给了变量 r。用监视窗体证明 0041c000h 是 r 的地址即可。

到此猜想并没有直接被证实。如果用 EAX、EDX 来传递返回值，结构体很大，寄存器不够用时怎样处理呢？是否与猜想一致呢？现在我们要做的就是将结构体变大，多定义一些成员变量。

```
struct myrd {
    int i1;
    int i2;
    int i3;
};
myrd myfunc() {
    myrd r1;
    r1.i1 = 1;
    r1.i2 = 2;
    r1.i3 = 3;
    return r1;
};
void main(int count, char** args) {
    myrd r;
    r = myfunc();
    getchar();
}
```

DM1-18

```
myrd r;
void main(){
r = myfunc();
00401085    call   0040107e
0040108a    mov    dword ptr ds:[0041c000h], eax
0040108f    mov    dword ptr ds:[0041c004h], edx
getchar();
```

DM1-19

```
myrd r;
r = myfunc();
0040121e    lea    eax, [ebp+ffffff1ch]
00401224    push   eax
00401225    call   00401180
0040122a    add    esp, 4
0040122d    mov    ecx,dword ptr [eax]
0040122f    mov    dword ptr [ebp+ffffff08h], ecx
00401235    mov    edx, dword ptr [eax+4]
00401238    mov    dword ptr [ebp+ffffff0ch], edx
0040123e    mov    eax, dword ptr [eax+8]
00401241    mov    dword ptr [ebp+ffffff10h], eax
00401247    mov    ecx, dword ptr [ebp+ffffff08h]
0040124d    mov    dword ptr [ebp-10h], ecx
00401250    mov    edx, dword ptr [ebp+ffffff0ch]
00401256    mov    dword ptr [ebp-0ch], edx
00401259    mov    eax, dword ptr [ebp+ffffff10h]
```

反汇编见 DM1-19。本来 myfunc 没有参数，但是 call 指令之前的“push　eax”指令向函数传递了一个参数。（**这似乎印证了我们的猜想。**）该参数是之前保存返回值的那个临时变量的地址吗？

> lea 指令：lea eax, [ebp + 10h]。其逻辑为 eax=ebp + 10h。如执行该指令时，ebp 为 3，执行后，eax 为 13h。

我们可以跟踪进函数。先记录这个压栈的值，即 EAX 的值 0x0012fe88。函数 myfunc()的反汇编如下：

```
myrd r1;
r1.i1 = 1;
```

```
0040119e   mov   dword ptr [ebp-10h], 1
r1.i2 = 2;
004011a5   mov   dword ptr [ebp-0ch], 2
r1.i3 = 3;
004011ac   mov   dword ptr [ebp-8], 3
return r1;
004011b3   mov   eax, dword ptr [ebp+8]
004011b6   mov   ecx, dword ptr [ebp-10h]
004011b9   mov   dword ptr [eax], ecx
004011bb   mov   edx,dword ptr [ebp-0ch]
004011be   mov   dword ptr [eax+4], edx
004011c1   mov   ecx, dword ptr [ebp-8]
004011c4   mov   dword ptr [eax+8], ecx
004011c7   mov   eax, dword ptr [ebp+8]
```

根据 1.3.4 节，该参数的地址应该是 ebp+8。我们要找到哪条指令用了该地址，004011b3 处的语句正是将 ebp+8 中存的值赋给了 EAX。ebp+8 中的值正好是 0x0012fe88，即 call 指令前压栈的 EAX 的值，见图 1.45。*(void **) (ebp+8)将地址 ebp+8 强制转为指针的指针(void **)，它指向的值被显示。

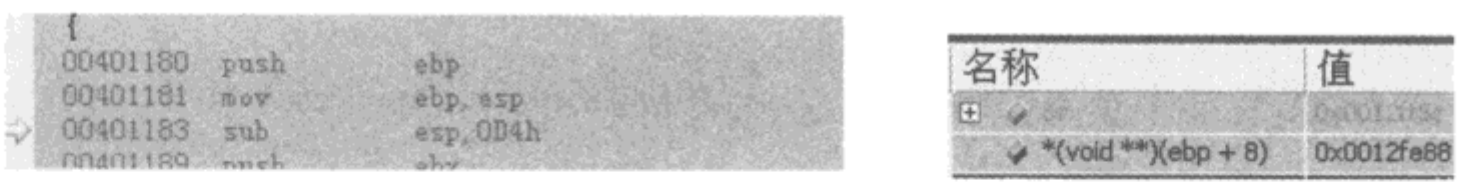


图 1.45

“mov eax, dword ptr [ebp+8]”指令使 EAX 存放了函数调用前分配的那个临时变量的地址，那么，我们可将注意力集中在和 EAX 相关的后续指令：

```
mov   dword ptr [eax]..
mov   dword ptr [eax+4]..
mov   dword ptr [eax+8]..
```

这正好是对结构体 3 个成员变量的赋值。i1 在开始处，所以其地址就是 eax，i2 离首部 4 字节（对应 eax+4），i3 离首部 8 字节（对应 eax+8）。再分析代码并简单运用调试工具分析，上面三条赋值指令确实是将局部变量 r1 的 3 个字段赋值给调用方为保存返回值分配的临时内存。代码中的最后一条指令“004011c7 mov eax, dword ptr [ebp+8]”再次将传入的保存返回值的内存的地址（在内存 ebp+8 中），赋值给 EAX。直到 ret 执行前，该 EAX 的值一直没有变化，那么，函数返回后，EAX 指向的就是保存返回值的临时变量的地址。

下面分析返回后 r=myfunc(1)的机理。call 指令返回后的代码如下：

```
0040122d      mov   ecx, dword ptr [eax]
0040122f      mov   dword ptr [ebp+ffffff08h], ecx    //为临时内存 2.i1 赋值
00401235      mov   edx, dword ptr [eax+4]
00401238      mov   dword ptr [ebp+ffffff0ch], edx    //为临时内存 2.i2 赋值
0040123e      mov   eax, dword ptr [eax+8]
```

其中黑体标注的 3 条指令分别从 eax 偏移 0、4、8 字节的地址获取值，而此时 eax 指向保存返回值的结构体的首部，所以这 3 个偏移量正好是成员变量 i1、i2、i3 的地址。可以验证，它们之后的 mov 指令的目标地址（即 ebp+ffffff08h 等）不是指向变量 r 的相关成员。而在这些代码后还有一堆 mov 指令，可以证明 ebp–10h 就是变量 r 的地址，这些赋值命令将最终为变量 r 赋值。

```
0040124d      mov   dword ptr [ebp-10h], ecx        //为 r.i1 赋值
00401250      mov   edx, dword ptr [ebp+ffffff0ch]
00401256      mov   dword ptr [ebp-0ch], edx        //为 r.i2 赋值
00401259      mov   eax, dword ptr [ebp+ffffff10h]
0040125f      mov   dword ptr [ebp-8], eax          //为 r.i3 赋值
```

我们可得到内存映像以及与指令的对应关系，见图 1.46。

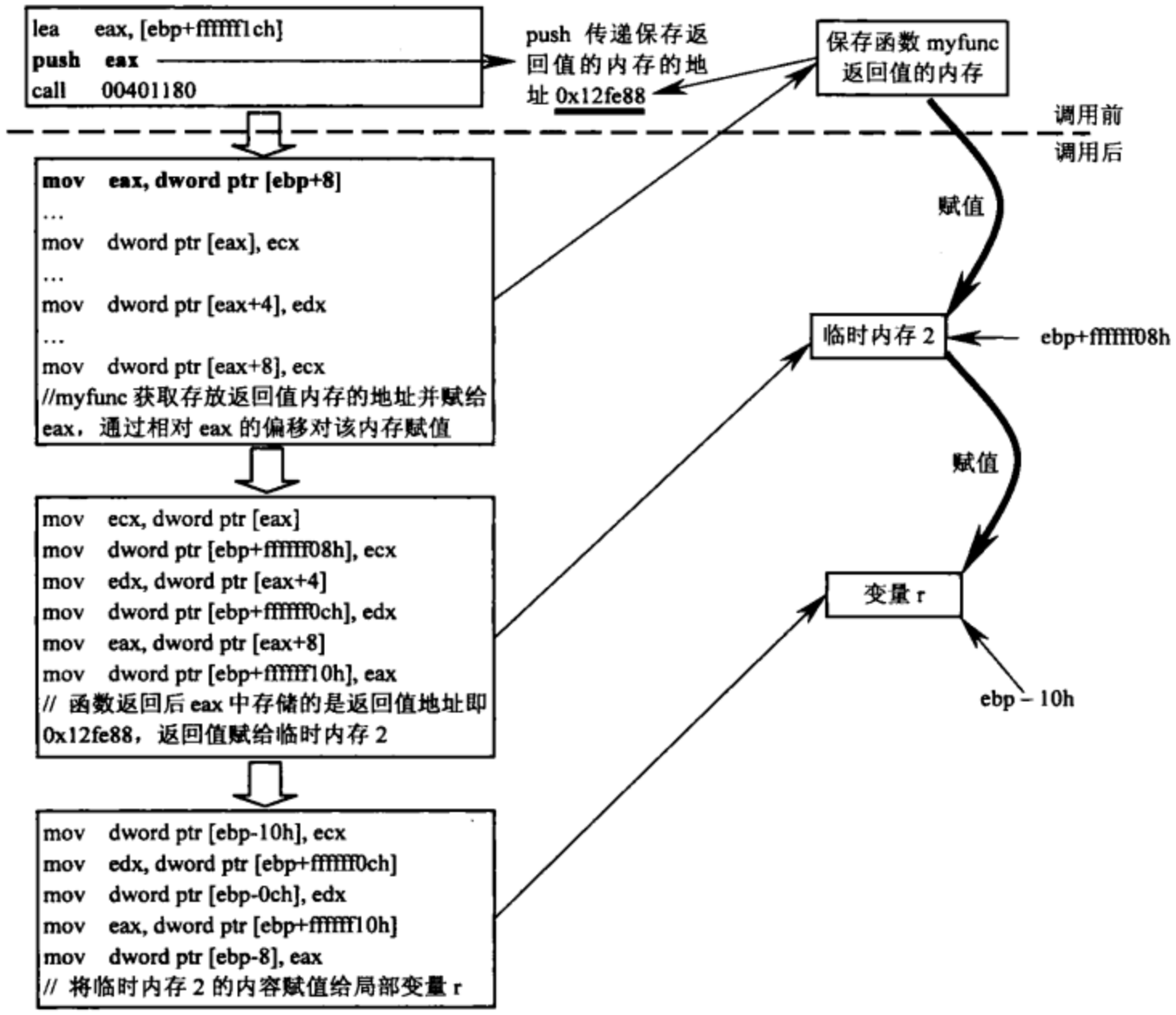


图 1.46

但是，返回值先赋值给临时内存 2，再从临时变量 2 赋值给 r。这样的中转意义何在？我们可以仿效前面全局变量的 Release 版本来看它的反汇编。将 main()函数中的 r 变为全局变量，反汇编如下：

```
r = myfunc();
0040109b      lea     eax, [esp+4]
0040109f      push    edi
004010a0      push    eax
004010a1      call    0040107e
004010a6      mov     esi, eax
004010a8      mov     edi, 41c000h
004010ad      movs    dword ptr es:[edi], dword ptr [esi]
004010ae      movs    dword ptr es:[edi], dword ptr [esi]
004010af      add     esp, 4
004010b2      movs    dword ptr es:[edi], dword ptr [esi]
```

从 call 之前的“push　eax”指令来看，应该是 eax 指向保存返回值的内存。然后将 eax 赋值给 esi，更可推断函数通过 eax 将该内存的地址再次返回。而之后出现了 3 条奇怪的“movs　dword ptr es:[edi], dword ptr[esi]”指令，几乎可以推断出是为结构体的 3 个成员变量赋值，esi 既然指向返回值内存的地址，那么 edi 指向的应该是全局变量 r 的地址。这 3 条一模一样的指令完成了 3 个不同内存的赋值，可以猜想该指令执行完后会将源地址 esi 和目标地址 edi 的值分别加 4。猜测完毕，剩下的就是实证我们的猜测：edi 是否是 r 的地址，esi 是否是返回值的地址，movs 指令是否按我们的猜想执行？如果这样，我们在 Release 版本中看到了更清晰的“大数据”返回的算法。实证的过程留给大家来做。

1.3.8 扫尾工作

在学习过程中，我们应该关注每个细节，如果不能马上明白，也要记下来将来分析。在 1.3.7 节中，返回代码如下：

```
z = Add(1, 2);
004139a2   push   2
004139a4   push   1
004139a6   call    004111e5
004139ab   add   esp, 8
004139ae   mov   dword ptr [ebp-8], eax
```

调用返回后，在 call 指令下有一条黑体标注的指令，将栈加了 8 字节。这条指令与返回值没有关系，那它有何用？我们知道，栈是保存参数、返回地址、局部变量的，那么函数调用完毕后，这些参数、返回值还有用吗？当然无用。如果它们还消耗栈的空间，将发生什么情况？很明显，随着每次函数调用，栈空间将不断减少直至耗尽栈内存，即栈溢出（stack overflow）。请大家写一段简单的代码，让栈溢出（提示：递归）。局部变量在被调用函数中释放了，保存返回地址消耗的栈被 ret 指令弹出也释放了。那么参数呢？“push　1”和“push　2”传参消耗了 8 字节，我们应该恢复它。栈结构是如何释放内存的？压栈，栈顶地址变小，弹栈，栈顶地址变大。这时，原来压栈使用的内存即可被再次压栈，存入新内容。所以我们只需将压栈减去的 8 字节加回来就可以了。“add　esp, 8”指令正好来完成这个工作。大家还可以参考 1.3.7 节的例子，保存返回值的内存地址作为参数传递给函数了，一个 push 指令消耗了 4 字节的栈，那么 call 返回后应该有“add　esp, 4”这样的代码来清理平衡栈，如下面黑体标注的指令：

```
r = myfunc();
0040121e   lea      eax, [ebp+ffffff1ch]
00401224   push     eax
00401225   call     00401180
0040122a   add      esp, 4
```

大家不妨将参数的个数增加，来看看“add　esp, xxx”指令中 xxx 的变化。

一切似乎到此就很清楚了，我们来看如下代码：

```
Add(1, 2);
Add(4, 5);
Add(100, 1);
...
```

大家想想其反汇编代码（见 DM1-20），觉得其中有什么问题？

DM1-20

```
push   2
push   1
call   ..
add    esp, 8
push   5
push   4
call   ...
add    esp, 8
push   1
push   100
call   ...
add    esp, 8
...
```

每个 Add()函数调用后都要执行相同的指令“add esp, 8”。这会带来什么影响？100 次调用就有 100 条 add 指令。而这 100 条 add 指令完全是重复的！什么办法可防止代码重复产生的空间开销？那就是将重复代码变成函数。如果我们将“add esp, 8”变成 Add()函数的一部分，调用返回后则不需“add esp, 8”指令（因为函数中已经执行了该指令）。100 次调用可节省 99 条 add 指令。如何来做？是否可将“add esp, 8”直接放入 Add()函数中？我们要养成一个好习惯，不要只想到“可放入”就不管了，一定要想如何放入。（**在“猜测，实证”这种方法中，猜测必须到细节，细节决定成败！这样你才会走得更深入，发现更多问题和有意思的东西。**）

不妨做一个理想实验，我们将指令放在 ret 之后：

```
ret
add  esp, 8
```

这明显无意义，执行 ret 后下面指令是无法执行的。那么，放在 ret 前？

```
add  esp, 8
ret
```

一下也看不出端倪，继续关注细节。虚拟出栈布局，分析指令执行过程，图 1.47 为调用 Add(1, 2)时的执行栈。

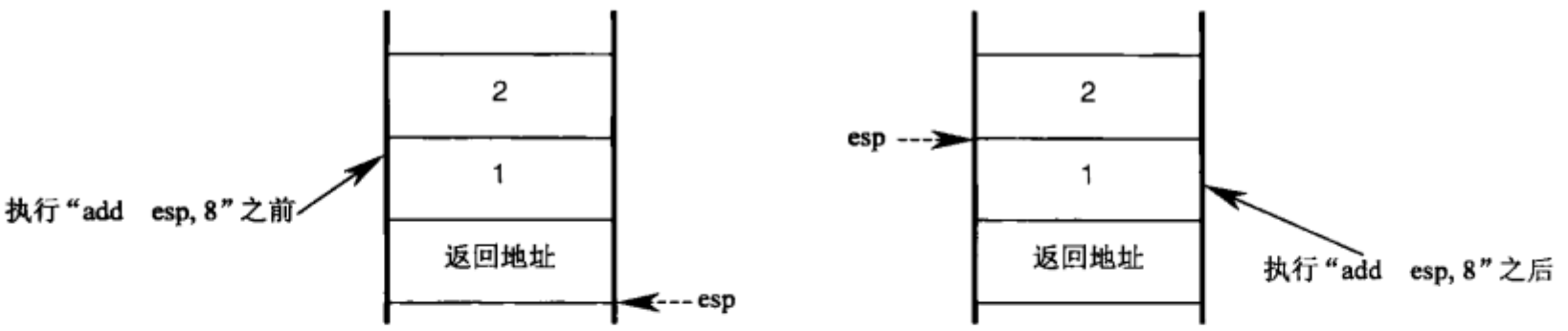


图 1.47

执行“add esp, 8”指令之前，ESP 应指向存返回地址的内存，那么执行后 ESP 将指向存储参数 2 的地址。再执行 ret 命令，2 将压入 EIP，程序将执行地址为 2 的内存中存储的东西，自然不对。这里的关键是“add esp, 8”必须在 ret 后执行，因为 ret 要弹出的返回地址在参数 2 和 1 的下方，只有弹出了下方的返回地址，才能以“add esp, 8”平衡栈。而之后执行已经被证明是不行的。所以，我们并没有办法像重用代码那样，将“add esp, 8”指令加入到被调用函数中！平常大家都觉得难以发现问题，很多时候老老实实地将每个想法想清楚“怎么做”，就能发现问题。现在问题出来了，要重用代码就要将“add esp, 8”的功能加入到被调用函数中，而执行顺序和栈布局又妨碍我们这样做（需常注意 ret 和栈的关系），该怎么办？

如果一个东西是合理的，我们就需要有支持它的机制。这里的关键是 ret 后执行栈顶增加的逻辑，两个动作无法分开来做，那么就需要新指令，支持返回和增加栈顶二合一的逻辑。

> ret 字节数：该指令将弹出栈顶的值作为返回值，并在弹出后将 esp 值加字节数。相当于以下指令二合一：ret 和 add esp, 字节数。

ret x

我们来做一个实验，在 Add()函数的名字前、返回类型后加上关键字_stdcall，代码如下：

```
int _stdcall Add(int x, int y){
    int sum;
    sum=x+y;
    return sum;
}
```

下面来看被调用函数和调用者的相关反汇编。首先是 Add()函数的，见 DM1-21。最后一条指

DM1-21

```
    return sum;
    00411457        mov   eax, dword ptr [ebp-8]
}
0041145a    pop   edi
0041145b    pop   esi
0041145c    pop   ebx
0041145d    mov   esp, ebp
0041145f    pop   ebp
00411460    ret   8
```

令表明返回后 esp 要加 8 字节。

再来看调用者的反汇编：

```
004139a2      push   2
004139a4      push   1
004139a6      call   004111f4
004139ab      mov    dword ptr [ebp-8], eax
printf("z=%d\n",z);
...
```

call 指令返回后，只有赋值的 mov 指令，之前的“add esp, 8”指令消失了，因为栈顶在 Add()函数中通过最后的“ret 8”指令得以增加。

现在我们不禁要问：什么原因导致 C 语言选择了调用方清理栈这种耗费代码空间的做法？获得了什么好处？从性能上分析，不过是时间和空间开销。空间上已经开销大了，难道时间上更快？两条指令（ret 和 add esp, 8）会比一条指令（ret 8）更快？似乎也不像。除性能还有什么？功能。C 语言支持一种变参函数，即函数参数个数不确定，如我们最早接触过的函数 printf()，根据字符串参数中格式化符号的不同，参数个数可不同（如 printf(“%d\n”, i)与 printf(“%s is %d”, “this”, i)）。ret x 方式不支持变参函数吗？在编译器中，在一个函数最后生成 ret 代码时，根据参数的不同，它生成的指令是固定的，如 Add(1, 2)生成“ret 8”，sum(1, 2, 3)生成“ret 0c”。代码一经生成就固定了，“ret x”指令运行期中不可变，决定了其参数只能是固定值。否则，如果调用 Add(1, 2, 3)，岂非 ret 指令要从“ret 8”变为“ret 0c”？这是不可能的。世间万物都是有代价的，功能、性能、空间、时间，难以两全。为了灵活性，C 语言舍弃了一点点空间性能。

现在我们知道，扫尾工作（平衡栈/清栈）有两种方式：① 调用方清栈，call 返回后，执行“add esp, x”指令；② 被调用方清栈，执行“ret x”指令。前者用空间代价换取了变参功能。

1.3.9 调用惯例

我们来回顾函数调用的环节：传参、清栈。清栈有两种选择，传参呢？前面参数通过栈传递，其实从寄存器也可传递，就如同返回值通过寄存器传递。寄存器传递不需要寻址内存空间，直接操控 CPU 中的寄存器速度快。但寄存器数目有限，特别是可自由使用的非常少，所以难以应对大量参数的传递。

我们来看 VC 中用寄存器传递参数的例子。将 Add()函数申明为_fastcall，其实从名字上也透露了寄存器传参的特点：fast（快速）。

```
int _fastcall Add(int x,int y){
    int sum;
    sum=x+y;
    return sum;
}
```

对应的调用方代码如下：

```
z = Add(1, 2);
004139A2      mov   edx,2
004139A7      mov   ecx,1
004139AC      call   004111F9
```

我们发现，2 和 1 通过寄存器 edx 和 ecx 分别传递。如果参数多了，往往编译器会将前两个参数用寄存器传递，后面的用压栈传递。比如，将 Add()函数变为有 3 个参数，我们看到，第 3 个参

数实际是 push 压栈传递的。

```
z = Add(1, 2, 3);
004139A2      push   3
004139A4      mov    edx,2
004139A9      mov    ecx,1
004139AE      call   004111FE
```

另外，即使栈传递参数，先压谁后压谁也是个问题，通过观察已知，C 语言的参数是从右往左压栈。至于从右往左还是左往右，不同语言有自己的选择，原因见习题 16。函数调用的习惯如下：

- ⊙ 是寄存器还是栈传递参数？
- ⊙ 栈传递时，参数是从右往左还是从左往右压栈？
- ⊙ 谁来清栈，是调用方还是被调用方？

而这几点的组合就是调用的习惯，即调用惯例（calling convention）。C 语言的调用方式是栈传参，参数从右往左压栈，调用方清栈。Pascal 是栈传参，参数从左往右压栈，被调用方清栈。_stdcall 是微软系统调用采用的惯例，除清栈是被调用方外，其他同 C 语言方式。_fastcall 是寄存器传递，不同语言在选择寄存器时有所不同，没有统一的规定。调用惯例非常重要，它涉及调用方和被调用方的约定，涉及它们能否正常沟通，在第 2 章中我们可以看到这个要点在跨语种编程中的重要影响。

1.3.10 函数指针

这时，我们可接触 C 语言中第 2 种重要指针——函数指针了。从使用角度出发，仿照数据指针的使用逻辑，可推出函数指针的内涵：数据指针是用来访问数据的，所以包含指向的内存地址；同时因为访问的长度需要确定，所以指针类型决定了从该地址访问多少字节。

函数指针是用来调用函数的，所以要包含函数代码的入口地址。函数指针有类型吗？调用函数并非只需要入口地址，还需要说明函数的参数表、返回类型、调用惯例，这样编译器才能生成正确的调用代码（如 push 多少参数、是否调用方清栈等），这三点合在一起就是函数原型。因此指针类型就是函数原型。所以，函数指针包含入口地址和函数原型两方面的信息。

比如，对于 int Add(int i, int b)函数，我们怎样描述其指针？首先，随便定义一个名字如 pfunc，然后在其左边加上“*”代表指针，即“* pfunc;”，接着需要描述其函数原型。先来描述其参数表（两个整数参数）：

```
* pfunc(int, int);
```

然后描述其返回类型：

```
int * pfunc(int, int);
```

问题：如果返回类型是 int *怎么办？难道写成“int ** pfunc(int, int);”？这样表示无法与返回值类型 int **区分，因此我们表示函数 Add()的函数指针类型如下：

```
int (*pfunc)(int, int);
```

最后，如果调用惯例是 C 方式，则完毕，否则需要将其标识出来，如_stdcall。例如，下面定义了一个函数指针变量 pfunc：

```
int (_stdcall *pfunc)(int, int);
```

然后可将函数原型匹配该指针类型的函数的入口地址赋值给它。这时用函数的名字代表入口地址，或在其前加“&”：

```
pfunc = Add;        或        pfunc = &Add;
```

DM1-22

```
int (*pfunc)(int, int);
int Add(int a, int b){
    return a + b;
}
void main(){
        push   ebp
        mov   ebp, esp
        sub   esp, 8
1       pfunc = Add;
2       mov   dword ptr ds:[00403374h],401000h
3       pfunc = &Add;
4       mov   dword ptr ds:[00403374h],401000h
5       pfunc(1, 2);
6       push   2
7       push   1
8       call    dword ptr ds:[00403374h]
9       add    esp,8
10      Add(1, 2);
11      push   2
12      push   1
13      call    00401000
14      add    esp, 8
}
```

当函数指针指向一个函数后，我们就能用函数指针访问该函数：

```
int sum = pfunc(1, 2);
```

反汇编代码见 DM1-22。代码标注的第 2 行、第 4 行分别是函数指针赋值的两种表示法，都是将函数 Add()入口地址 401000h 赋给变量 pfunc。

第6～9行和第11～14行分别是用函数指针和函数名两种方式调用的反汇编，通过比较发现，除了 call 指令，后者是用偏移量跳转，其他相同，执行结果也相同。

函数指针赋值的原则是：只能将与指针原型匹配的函数的入口地址赋值给它。

例如，如果将 Add()变成 int Add(int a, int b, int c)，Add()还可赋给 pfunc 吗？答案是“否”，因为两者的函数原型不同（**参数表不同**）。

如果将 Add()改成 float Add(int a, int b)，Add()可赋给 pfunc 吗？答案是“否”，因为两者的函数原型不同（**返回类型不同**）。

如果将 Add()改成 int _stdcall Add(int a, int b)，Add()可赋给 pfunc 吗？答案是“否”，因为两者的函数原型不同（**调用惯例不同**）。

我们就最后一种情况，分析为什么不可以赋值，其他情况自行分析。假定能够赋值将是怎样的状况？分析之前的反汇编码：

```
5     pfunc(1, 2);
6     push   2
7     push   1
8     call    dword ptr ds:[00403374h]
9     add    esp, 8
```

第 9 行的指令将压栈传参的 8 字节清栈。如果 Add()是_stdcall，那么在函数中已通过“ret 8”清栈，这里再次清栈 8 字节，势必影响栈布局。简单来说，当包含这段代码的函数返回时，返回地址必然错误。

函数指针变量申明比较麻烦，如果多次在不同的地方写，很有可能少写一个参数或调用惯例写错等导致编程错误。为了减少这种麻烦，如果能将一种函数指针声明为一种类型，然后所有定义函数指针变量的地方都用该类型定义就好了。例如，typedef int (*funcPtrType)(int, int)声明了一个函数指针变量类型 funcPtrType。funcPtrType pfunc 了定义一个函数指针变量 pfunc。之前的代码可如下修改：

```
typedef int (*funcPtrType)(int, int);

funcPtrType pfunc;

int Add(int a, int b){
    return a + b;
}
...
```

问题：对于数据指针，在遵循一定原则下，可安全地进行指针强制转换（见 1.2 节），是否函数指针也可遵循某种原则进行指针强制转换？在回答这个问题前，我们用强制转换处理前面分析的调用惯例不匹配例子，并分析其反汇编，见 DM1-23。注意 test 函数中的第一行语句，调用 Add()前，用指针强制转换将 Add()的指针类型进行了强制转换，因为 Add()的函数原型是_stdcall，而 funcPtrType 定义的函数指针类型是 C 方式，如果不强制转换，编译器会报错：

```
错误 1    error C2440:  "=" :无法从 "int (__stdcall *) (int,int)"
转换为 "funcPtrType"
    e:\project\cprj\test2\test2\test2.cpp    18        test2
```

错误提示无法将 Add()的类型转换为 pfunc 的类型 funcPtrType，因为两者的调用惯例不同。

DM1-23

```
typedef int (*funcPtrType)(int, int);

funcPtrType pfunc;

int _stdcall Add(int a, int b){
    return a + b;
}
void test(){
    pfunc = (funcPtrType)Add;
    pfunc(1, 2);
    Add(1, 2);
}
void main(){
    test();
}
```

下面来看强制转换后的反汇编，见 DM1-24。

首先，第 2 行强制转换没有多余的逻辑，只是“欺骗编译器”，将入口地址赋值给了 pfunc 变量。与数据指针的强制转换一样，强制转换本身并没有对应的指令，仅仅是将地址赋值给指针变量。

如果大家执行该程序会产生错误。我们看看错误与正常方式的差别吧。

① 错误方式：函数指针发起的调用，call 后第 7 行将栈顶上移 8 字节（这是正常方式没有的），因为这时 pfunc 的类型的调用惯例是 C 方式，所以是调用方清栈（第 7 行）。

② 正常方式：用函数名调用，第 11 行 call 指令后无清栈指令（因为 Add()是_stdcall，被调用方清栈，自然不需 call 后再加 ESP 中的值）。

DM1-24

```
  void test(){
        push   ebp
        mov    ebp, esp
1       pfunc = (funcPtrType)Add;
2       mov    dword ptr ds:[00403374h], 401000h
3       pfunc(1, 2);
4       push   2
5       push   1
6       call   dword ptr ds:[00403374h]
7       add    esp, 8
8       Add(1, 2);
9       push   2
10      push   1
11      call   00401000
  }
```

被调用的 Add()函数中本已用 ret 8 清栈了（ESP 加了 8），而用函数指针调用时，第 7 行自作多情地再次对 ESP 加 8 清栈，这必会导致错误！

具体来看错误是什么。先看 test()函数需返回的地址，单步跟踪进 test()函数中，查看 ESP 的值为 0x0012FF70，指向内存中的 4 字节整数是 0x0040104b（用小端机顺序解读），即返回地址。因为第 7 行将栈顶多上移了 8 字节，所以 test()函数执行 ret 指令前，ESP 没指向刚才保存返回地址的内存，即 0x0012FF70，而是 0x0012FF78，这时执行 ret 命令，将会把 0x0012FF78 中保存的值作为返回地址弹出。下面证明该论断。

在 ret 指令处设置断点，按 F5 键执行，在该断点停下，ESP 的值确实是 0x0012FF78。而此时在内存窗体中输入 0x0012FF78，可知其中保存的整数值是 0x78542201，那么 ret 执行后，程序将执行该地址的指令。按 F11 键，单步执行一次，我们可以看到代码确实飞到这个地方了：

```
78542201   add   esi, 4
```

这当然是错误的返回位置，如果执行下去，谁也不知道会有什么结果，最终必然出现异常。

简单总结一下函数指针强制转换的作用：与数据指针一样，函数指针在转换赋值的时候，除了赋值地址没有多余动作。起效时是在调用函数的时候，编译器会按其原型产生不同的代码。

通过这个例子，我们能回答之前的问题“函数指针是否可通过某种规则强制转换”。答案是：几乎不可。不能赋值，一定是因为指针变量和函数两者的函数原型不同，函数原型不同必然是参数

顺序、个数、清栈等有差异，势必造成某种错误。因此，可以说 99%的函数指针强制转换都必然导致错误发生，至于那 1%，大家也许将来能遇到，这里不再详谈。

最后一个问题：函数指针有什么用？用函数名字调用就可以了。那么，用函数指针调用函数，如果不看指针的值，肯定不知道是哪个函数，因为这个指针指向的是什么函数并不清楚。也就是说，同样的调用代码可能调用到不同的函数。这个好像面向对象的某个高级特征：虚函数调用。同样的代码，调用到的函数可能是父类的，也可能是子类的。后面我们将看到函数指针的威力：超强的灵活性，模块化编程的支撑力量！

1.4 数组、结构体

将数组和结构体放在一起讨论，主要是因为它们都是复合数据类型。

1.4.1 数组

数组在 C 语言中是比较简单的结构，谈它之前我们先从一个问题开始。笔者学习 C 时就感到很奇怪，为什么数组的第一个元素的索引要从 0 开始，而不是 1。后来看到 Pascal 语言可用索引 1 开始访问第一个元素，就非常鄙视 C 语言的“非人性做法”。当打开这个黑匣子分析清楚后，才明白自己的无知，一切选择都有原因。

让我们先猜测数组的访问方式。一个最基本的想法是，要访问一个内存只要拿到其地址即可。如何获取到第 i 个元素的地址？最直接的想法是，如果能拿到数组首部地址，加上相对偏移就能计算出第 i 个元素的地址。这个偏移比较好计算，因为每个元素大小一样，用元素个数乘以元素大小就能获得偏移量。图 1.48 给出了数组的内存结构。

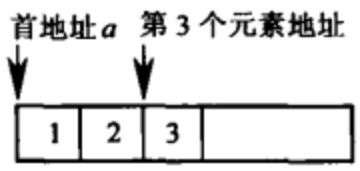


图 1.48

第 1 个元素的地址 = 首地址 a

第 2 个元素的地址 = 首地址 a + 1×元素大小

第 3 个元素的地址 = 首地址 a + 2×元素大小

.....

第 i 个元素的地址 =首地址 a + $(i-1)$×元素大小

这里是用 1 作为元素第一个编号，在计算地址时总要进行一次减法运算。现在应该发现 C 语言数组索引编号的奥妙了吧，奇怪的方式一定有其原因。如果从 0 开始编号，计算地址时就没有了减法，用违背常规的方式换来了计算速度：

第 0 个元素的地址 = 首地址 a

第 1 个元素的地址 = 首地址 a + 1×元素大小

第 2 个元素的地址 = 首地址 a + 2×元素大小

.....

第 i 个元素的地址 =首地址 a + i×元素大小

好了，按照我们的习惯，到了实证的阶段。程序代码如下：

```
int array[5];
void main(){
    array[0] = 1;
```

```
    array[1] = 2;
    array[2] = 3;
}
```

其反汇编如下：

```
array[0] = 1;
    mov   dword ptr ds:[004174c4h], 1
array[1] = 2;
    mov   dword ptr ds:[004174c8h], 2
array[2] = 3;
    mov   dword ptr ds:[004174cch], 3
```

可知，数组的首部地址是 004174c4h；第 2 个元素是 004174c8h，比前一个增加了 4 字节；第三个是 004174cch，离首部偏移了 8 字节，用监视器查看 array[0]的地址正好是 004174c4h。因为 int 是 4 字节，所以偏移量是按 4 的整数倍递增的。

我们再来看数组是局部变量的情况，其代码如下：

```
int array[5];
array[0] = 1;
    mov   dword ptr [ebp-18h], 1        // mov dword ptr[ebp - 18 h + 0 * 4], 1
array[1] = 2;
    mov   dword ptr [ebp-14h], 2        // mov dword ptr[ebp - 18 h + 1 * 4], 2
array[2] = 3;
    mov   dword ptr [ebp-10h], 3        // mov dword ptr[ebp - 18 h + 2 * 4], 3
```

每行后添加的注释等价代码清晰地显示了数组是按首部地址+偏移量的做法。这些代码还看不出基于 0 索引的数组的优势。就用下面一段典型的 for 循环访问数组来实证这一优势。

```
for(i = 0; i < 10; i++)
    ...
a[i] = 1;
00412036      mov   eax, dword ptr [ebp-8]
00412039      mov   dword ptr [ebp+eax*4-38h], 1
```

其中，最后黑体所示的 eax * 4 正如前所示为偏移量计算：ebp–38h 是 a 数组的首址，eax 存储的是 i 的值（请自己证实）。

从以上反汇编我们也理解了 C 语言为什么会发生数组越界错误，因它只是拿到首部地址然后加偏移量，如果索引值超出范围，那么求得的元素地址也就超过了范围。请大家自己实验超越范围的数组元素访问，并查看其反汇编代码。

1.4.2 结构体

相对数组而言，更自由和更复杂的数据结构是结构体。还是老规矩，大家先猜测它在内存中会是什么样子。例如：

```
struct Person{
  int age;
  int no;
}
```

两个整数成员分配 8 字节应该就可以了。而且为了不浪费，这两个成员变量应该是连续的。如

何访问其中的成员变量呢？与数组的思路相仿，如果拿到结构体首部地址，然后求偏移量即可。偏移量如何计算？因为代码是编译器生成的，它自然知道哪个字段在哪个位置，如 Person 的 no 成员是在偏 4 字节的地方，因为第一个成员占了 4 字节。

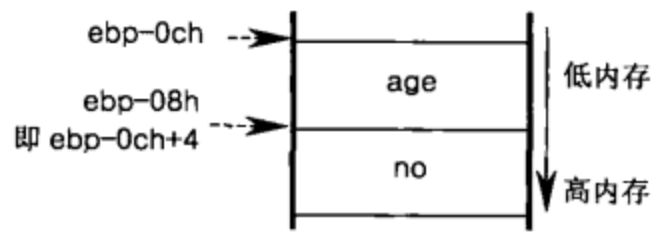


图 1.49

下面用真实代码实证：

```
Person p;
p.age = 1;
mov   dword ptr [ebp-0Ch], 1
p.no = 2;
mov   dword ptr [ebp-8], 2
```

从反汇编我们能得出其内存结构，见图 1.49。

第一条 mov 指令对 ebp–0ch 地址赋值。第二条指令对 ebp–8 地址赋值，该地址正好比 ebp–0ch 大 4 字节，说明赋值给了结构体首部偏 4 字节的 no 成员：ebp–8=ebp–0ch（结构体的首部地址）+4。

用监视窗体来证明分析，见图 1.50。可知，age 的地址&p.age 和 ebp–0ch 相等，为 0x0012ff5c；no 的地址&p.no 与 ebp–8 相等，为 0x0012ff60。

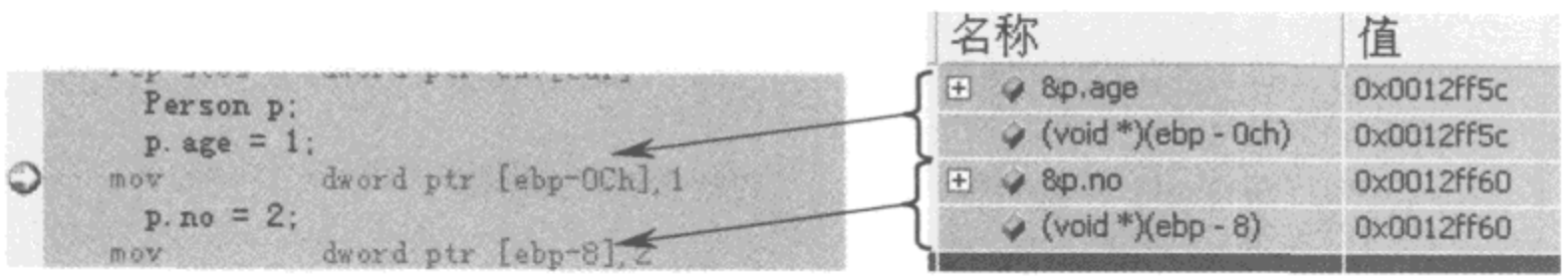


图 1.50

我们再来看一段代码的编译结果，并将版本改为 Release 版本，且关掉优化选项（在“项目属性”中的“配置属性→C/C++→优化→优化”选项设定为禁用）。该效果在 VC 6.0 中的 Debug 版本中可以查看到。

```
int no;
int age;
age = 1;
    mov   dword ptr [ebp-8], 1
no = 2;
    mov   dword ptr [ebp-4], 2
```

在该编译状态下，将代码改为如下：

```
Person p;
p.age = 1;
    mov   dword ptr [ebp-8], 1
p.no = 2;
    mov   dword ptr [ebp-4], 2
```

我们发现，两者的反汇编代码一模一样。换句话说，无法从反汇编确定某块内存到底是结构体还是相互比邻的局部变量。结构体并未在内存中有更多特殊性，没有用一段内存（如标志位之类）来表示这是一个结构体，内存大小看来就是全部成员变量之和（其实未必，请看 1.5 节），不过是编译器提供给我们的一个方便自动求取偏移量的方法：我们给定要访问的成员，编译器就能自动为我们确定出偏移的位置，如 p.no 就自动偏移 4 字节。

1.5 无法沟通——对齐的错误

1.5.1 结构体对齐

在 1.4 节我们了解了结构体的内存布局，简单地说，就是依照定义顺序依次为每个成员变量分配空间。下面从一个小例子展开：

```
struct Person{
    char c;
    short a;
}
...
printf("size of Person=%d\n", sizeof(Person));
```

其结果是什么？按照前面的结论，就是 sizeof(short)+sizeof(char)=3 字节。可是实证的结果是 4 字节。这 1 字节是从哪里冒出来的？代码如下：

```
Person p;
p.a = 1;
p.c = 2;
```

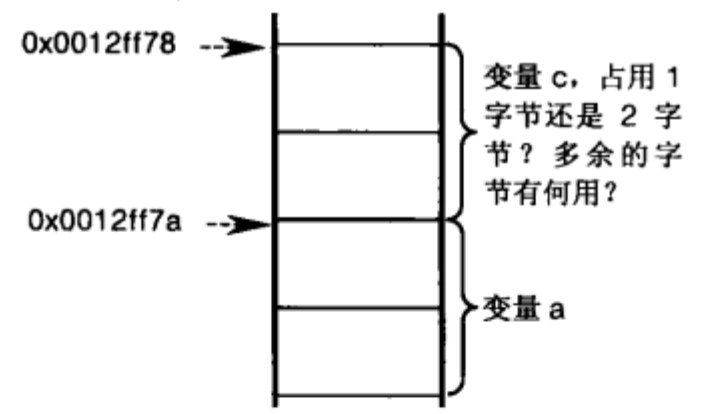


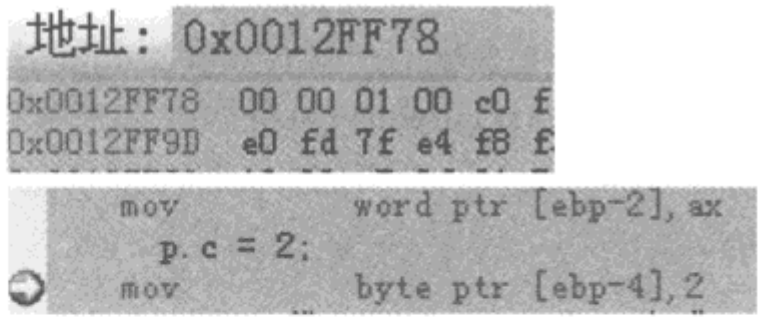
图 1.51

在 p.a=1 处设置断点，执行。在监视界面中输入&p.c 和&p.a 的值：0x0012ff78 和 0x0012ff7a。其对应内存布局见图 1.51。

c 和 a 之间相差了 2 字节，而理论上 c 只占用了 1 字节，那多余的 1 字节是在哪里？有何用处？首先我们要确定多余字节的位置。观察对变量 c 赋值的反汇编：

```
p.c = 2;
    mov   byte ptr [ebp-4], 2
```

它对 ebp–4 地址赋值了 2，而且从指令看，赋值长度是 1 字节（mov　byte ptr）。在监视器中输入(void *)(ebp-4)，可知其值是 0x0012ff78，因此成员变量 c 处于结构体首部，占用了 1 字节。为了进一步确认，我们在 p.c=2 处设置断点，执行到该处中断，并将地址 0x0012ff78 输入到内存窗体，并故意将 0x0012ff78 的前 2 字节修改为 0，然后看执行 mov 指令后到底修改了哪些字节，见图 1.52。单步执行，结果见图 1.53。



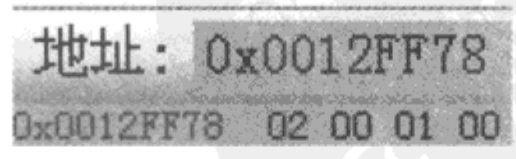


图 1.53

从结果看，只有第 1 字节被修改为 02 了，第 2 字节没有动，可知对于 Person 而言，第一个成员变量 c 虽然占用了 2 字节，但第 2 字节似乎并没有用到。我们需要弄清这种现象的原因。

对齐和结构体成员变量多了一些字节有何联系呢？我们假定刚才的 Person 变量 p 放在偶数地址，其第 1 字节存放 char 成员变量 c，如果第 2 字节存放 a，而第 2 字节是奇数地址，存放 short

对齐：基本数据（如单字节、双字节、四字节整数）存放处的地址必须能被自己数据类型的大小整除。比如，双字节整数存放的地址必须被 2 整除，四字节整数存放的地址要被 4 整除。如果不满足要求，不同体系结构的 CPU 反应不同。有的会直接结束进程的执行，有的可以容忍，但访问速度变慢（如 x86）。

类型的 a，奇数地址无法被 2 整除（a 是两字节大小），就造成成员变量 a 存放的位置没有对齐。当然，如果 p 放在奇数地址，似乎也可以满足对齐条件。但当我们将所有变量整体来考察，必须将一个结构体内部按照某种方式排列，所有的变量就能够满足对齐要求。这是什么规律？在分析它之前，先看看编译器的另外一个编译选项，即选择按多少字节对齐，见图 1.54。

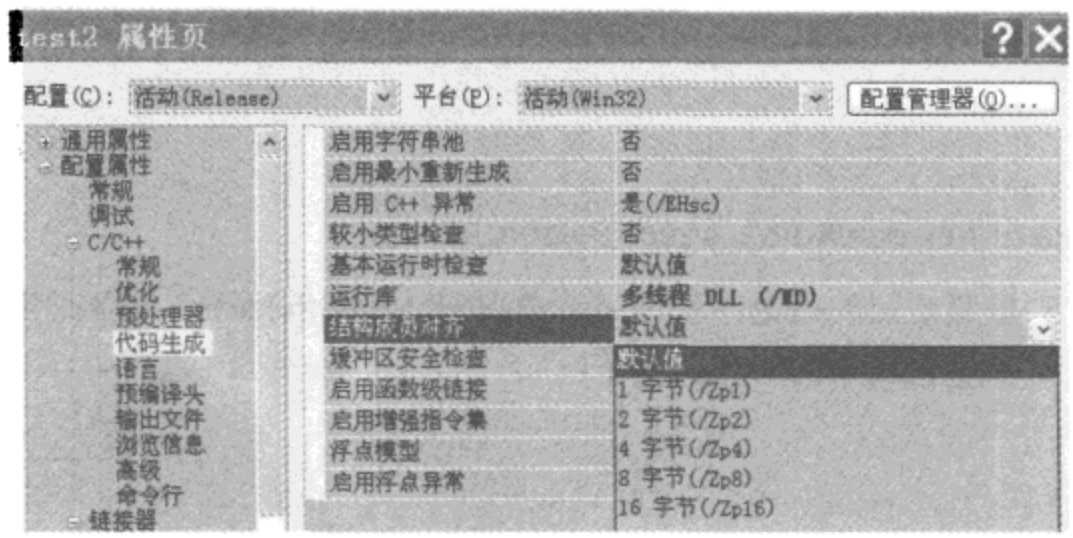


图 1.54

先来看看它的作用，选择按 16 字节对齐，Person 的大小是 4。选择 8 字节、4 字节、2 字节对齐，结果都是 4。选择 1 字节对齐，结果是 3，即没有无效填充字节，所见即所得！

不妨来探索一下结构体对齐的规律。从前面来看，当对齐方式选为 1 字节时，影响了对齐结果，那么不妨将成员变量改成 int 类型，来看在不同对齐方式下的影响。

```
struct Person {
    char c;
    int a;
}
```

在 1、2、4、8、16 字节的对齐方式下，其大小分别为 5、6、8、8、8，而成员变量 a 相对结构体头部的偏移量分别是 1、2、4、4、4。可知不同的对齐方式影响到了填充字节的多少。计算填充后 c 所占长度，见表 1.1。

首先，变量 c 所占的长度不会大于结构体中最长的字段 a 的大小，a 为 short 时，c 所占长度不大于 2；a 为 int 时，c 所占长度不大于 4。其次，占用长度能否达到最大值还取决于对齐的长度，似乎当对齐长度小于成员变量的最大长度时，c 所占长度就是对齐的长度。如 a 为 short 时，如果对齐选 1，那么 c 所占长度为 1（1<2），而 a 为 int 时，对齐长度选 1，c 所占长度为 1（1<4），对齐长度选 2，c 所占长度为 2（2<4）。所以，c 所占长度 = min{max{sizeof(成员变量)}, 对齐长度}。其中，min 和 max 代表取最小和最大值，max{sizeof(成员变量)}代表取所有成员变量大小的最大值。

为了证实这个猜想，我们将结构体修改为：

```
struct Person{
    char c;
    double a;
}
```

下面来看在成员变量的最大长度为 8 时的状况，见表 1.2。

表 1.1　成员变量 c 所占长度（一）

对齐（字节） / a 大小（字节）	1	2	4	8	16
2（short）	1	2	2	2	2
4（int）	1	2	4	4	4

表 1.2　成员变量 c 所占长度（二）

对齐（字节） / a 大小（字节）	1	2	4	8	16
2（short）	1	2	2	2	2
4（int）	1	2	4	4	4
8（double）	1	2	4	8	8

a 变成 double 时（占 8 字节），其规律确实如我们所推断。现在的问题是，每个成员变量都会填充吗？从道理上讲不应该这样。我们再做实验来验证：

```
struct Person{
  char c1;
  char c2;
  short a;
}
```

这时，不论选择多少对齐，整个长度就是 4，每个成员字段所占的长度都是其长度本身，填充字节不见了。再将 a 的类型改为 int：

```
struct Person{
   char c1;
   char c2;
   int a;
}
```

对齐为 1、2、4、8、16 时，结构体长度分别为 6、6、8、8、8，a 离头部的偏移量为 2、2、4、4、4。也就是说，c1 和 c2 一起所占长度分别为 2、2、4、4、4 字节。似乎 c1 和 c2 连续存放作为一体进行了填充，在对齐为 4、8、16 时，在它们后面填充了 2 字节。结合前面分析填充字节的规律，我们可以这样推断：

首先选定一个盒子，然后依序将字段往盒子中放，当盒子放不下后，再用另一个盒子存放，直至所有字段都存放完毕。

这里有几个要确定的要素：盒子的长度、放入盒子中的限制条件。从上面的例子看，这个盒子的长度如下计算：

盒子长度= min{ max{ sizeof(成员变量) }, 对齐长度 }

其中，min 和 max 代表取最小值和最大值，max{ sizeof(成员变量) }代表取所有成员变量大小的最大值。

而放入盒子的限制条件会怎样？例如：

```
struct Person{
   char c1;
   short s;
   int a;
}
```

选中对齐为 4 时，盒子长度= {成员变量最大长度(4 字节, int a), 对齐为 4} = 4 字节。我们打印后发现，该结构体的长度为 8。而字段 s 距头部偏移量为 2，不是直接放在 c1 后，在 c1 后有一个填充字节，见图 1.55(a)。如果将结构体改为

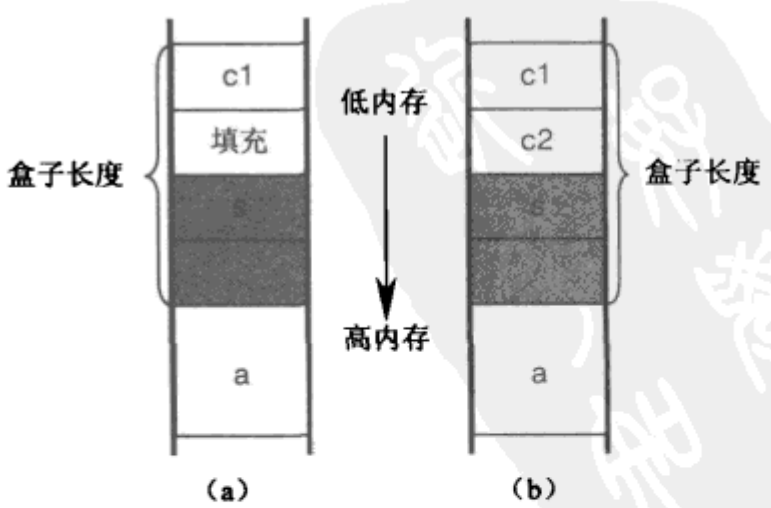


图 1.55

```
struct Person{
    char c1;
    char c2;
    short s;
    int a;
}
```

结构体长度依然是 8，字段 s 依然在头部偏移 2 字节的地方，见图 1.55(b)。

对比两图，我们发现，1 字节的数据可任意放入盒子中，如 c1、c2 的放置，2 字节的数据只能放在离盒子头部偏 0 和 2 字节的位置，如字段 s 只能放置偏 2 字节的地方。在图 1.55(a)中，c1 与 s 之间就产生了一个填充字节。由此，我们推断成员变量放入盒子的规则如下：

离盒子头部偏移字节数= n×sizeof(成员变量)　　（n=0，1，2，…）

比如，对于 short 类型的变量，如盒子长度为 4，只能放置偏移 0 和 2 字节的地方，见图 1.56(a)；对于 short 类型的变量，如盒子长度为 8，只能放置偏移 0、2、4 和 6 字节的地方，见图 1.56(b)；对于 int 类型的变量，如盒子长度为 4，只能放置偏移 0 字节的地方，见图 1.56(c)；对于 int 类型的变量，如盒子长度为 8，只能放置偏移 0 和 4 字节的地方，见图 1.56(d)。

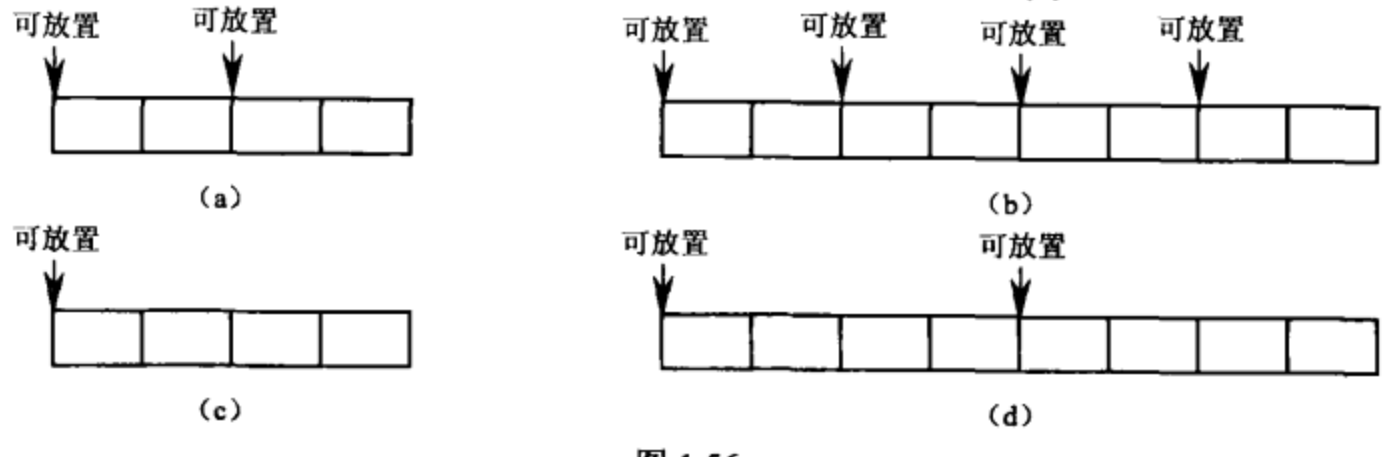


图 1.56

对于这个推断，我们修改结构体定义来证明：

```
struct Person{
    char c1;
    short s;
    char c2;
    int a;
}
```

在对齐为 4 的情况下，盒子长 4，字段 c1 放在首部，字段 s 只能放在偏 2 字节的地方，c1 与 s 之间有一个填充字节，因此第一个盒子用完。分配第二个盒子，c2 放在其首部，剩下 3 字节无法放字段 a，因此只有再次分配第 3 个盒子，c2 后的填充位占 3 字节，字段 a 放入第三个盒子。所以，总长为 3×4=12 字节。布局见图 1.57。

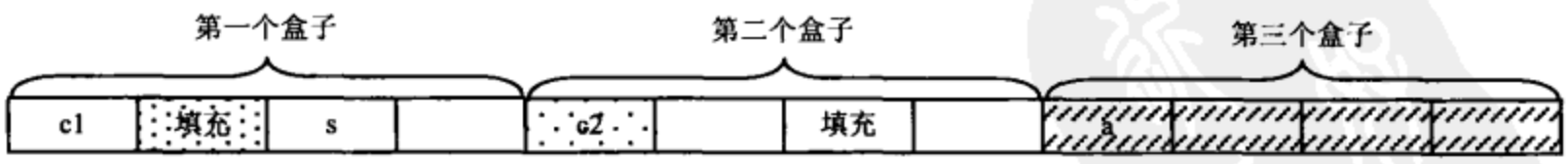


图 1.57

通过设置断点，在监视窗口中验证，见图 1.58。

最后，我们正式得出对齐的规律：

首先选定一个盒子，然后依序将字段往盒子中放，当盒子放不下后，又用下一个盒子存放，直至所有字段都存放完毕。

图 1.58

其中相关限制条件为：① 盒子长度 = min{max{sizeof(成员变量)}，对齐长度}，其中 min 和 max 代表取最小和最大值，max{sizeof(成员变量)}代表取所有成员变量大小的最大值；② 字段放入盒子的可放置位置如下：

$$\text{离盒子头部偏移字节数} = n \times \text{sizeof(成员变量)} \quad (n=0, 1, 2, \cdots)$$

1.5.2 无法沟通

结构体对齐影响我们什么？似乎什么也没有，要求取其大小时，用 sizeof 即可；平常我们也不需理解每个成员变量相对其首部的偏移量，访问一个字段时，编译器会自动计算偏移量进行访问。

下面来看一个网络编程中经常使用的技巧。报文一般有一个头部，在拿到头部协议格式后，我们会将它翻译成一个结构体。下面给出了简化的 IP 头部的结构体表示（从 Linux 简化）：

```
struct ip_hdr {
    char version : 4;                    //该字段为 4 位
    char ihl : 4;                        //该字段为 4 位
    unsigned char tos;
    unsigned short tot_len;
    unsigned short id;
    unsigned short frag_off;
    unsigned char ttl;
    unsigned char protocol;
    unsigned short check;
    unsigned int saddr;
    unsigned int daddr;
};
```

假定有如下协议：第 1 字节为标志位，代表报文类型；第 2～5 字节为整数，代表报文数据部分长度；其后是报文数据部分。

我们会将协议转换为一个结构体：

```
struct hdr{
  char flag;
  int len
};
```

下面是一网络程序伪代码，分为发送方和接收方代码。发送方的代码见 DM1-26。

DM1-26

```
1    char buf[128];
2    int * pdataLen;
3    buf[0] = 1;                         //设定标志类型
4    pdataLen = (int *)&buf[1];
5    * pdataLen = htonl(12);             //设定数据长度
6    //填充数据部分 12 字节.
7    buf[5] = 1;
8    buf[6] = 2;
     ...
9    buf[16] = 12;
     ...
10   //将 buf 开始的 17 字节发送出去，包括头部 5 字节、数据 12 字节
11   send(sockethnd, buf, 5 + 12, 0);
```

第 3 行填写标志位 flag，第 4 行将第 2 字节的地址强制为整数指针，第 5 行将数据长度 12 设定到第 2 字节开始的 4 字节。其中，为了让报文在大端机/小端机中处理整数的表示统一化，整数 12 将用 htonl 转换为网络顺序表示（网络顺序就是大端机顺序），报文接收方用 ntohl

将网络顺序再次转换为本地顺序。之后从 buf 偏移 5 字节处填写数据部分的内容。

这个协议头部很简单，只有两个字段，如果自己计算协议字段的偏移量还可以接受。但如果像上面 IP 协议那样复杂的头部，如此处理既麻烦又容易出错，因此我们常常采用一个技巧：将 buffer 的头部地址强制转换为代表协议头部的结构体指针，然后用该指针去访问结构体成员。该方法利用以下事实：访问结构体字段，将由编译器自动计算字段偏移量（见 1.4.2 节），不需我们手动加偏移，从而极大简化了编程。

接收方代码见 DM1-27。第 4 行将从网络接收 5 字节数据，放入 buf 中，它是协议头部。第 5 行将收到的数据起始地址（buf 的地址），强制转换为协议头部指针 struct hdr *类型，这样就可以用 phdr->len 来访问任何一个协议成员字段了，非常简洁。第 7 行将收到的 len 字段的值用 ntohl 从网络顺序转换为本地顺序。

DM1-27

```
1  struct hdr * phdr;
2  char buf[128];
3  int dataLen;
4  recv(sockethnd, buf, 5, 0);                    //接收协议头部 5 字节
5  phdr = (struct hdr *)buf;
6  printf("hdr flag is %d\n", phdr->flag);        //获取协议头部类型 flag
7  dataLen = ntohl(phdr->len);                    //获取协议头部关于数据的长度信息
8  printf("data length is %d\n", dataLen);
```

可是当这两个程序联调时，结果是错误的。为什么？如果接收方用发送方的方法改写程序，即手动计算偏移量，buf[1]的地址为 len 的起始地址，程序又是正确的，则

```
recv(sockethnd, buf, 5, 0);                    //接收协议头部 5 字节
dataLen = ntohl(*((int *)&buf[1]));            //获取协议头部关于数据长度的信息
printf("data length is %d\n", dataLen);
```

为了方便演示和分析该问题，我们把它从网络编程中分离出来，变成一个本地程序，见“code\第 1 章\对齐的错误\alignerror”。其实这种简化体现了另一种可能出问题的地方：如内核和用户空间的程序交互时，也会定义协议，一方发出信息，一方接收处理信息，这时依然可能出现上述问题。本地版本的程序如下：

```
#include <stdio.h>
struct Hdr{
char flag;
    int len;
};
void decodeHdr1(char * buf){
    struct Hdr * phdr;
    phdr = (struct Hdr *)buf;
    printf("version1 len = %d\n", phdr->len);
}
void decodeHdr2(char * buf){
    int * pdataLen = (int *)&buf[1];
    printf("version2 len = %d\n", * pdataLen);
}
void main(){
    char buf[64];
    buf[0] = 1;                                //填写标志位
    *((int *)&buf[1]) = 12;                    //填写数据长度为 12
```

```
    decodeHdr1(buf);
    decodeHdr2(buf);
}
```

main()函数中将数据长度部分设为 12（模拟发送方用手动求偏移方式构造报文），decodeHdr1 和 decodeHdr2 接收填写好的数据的头部地址，分别用两种方式解析头部并打印出长度信息，打印结果分别为-858993664 和 12。明显，decodeHdr2 的结果是正确的。为什么用结构体指针强制转换的 decodeHdr1 反而不正确呢？如果一时分析不清楚，直接面对最原始的信息（反汇编、内存等）是最简单也是最好的办法。在 decodeHdr1 处设置断点，反汇编跟踪，见 DM1-28。

DM1-28

```
1  phdr = (struct hdr *)buf;
2      mov   eax, dword ptr [ebp+8]
3      mov   dword ptr [ebp-8], eax
4  printf("version1 len = %d\n", phdr->len);
5      mov   si, esp
6      mov   ax, dword ptr [ebp-8]
7      mov   ecx, dword ptr [eax+4]
8      push  ecx
9  push          41573ch
10 call          dword ptr ds:[004182bch]
```

第 2 行是将 buf 指针指向的地址赋值给 EAX。

第 3 行将 eax 赋值给指针 phdr，phdr 变量的地址是 ebp–8，这时 phdr 也指向 buf 指向的内存。

第 10 行调用 printf()函数，那么它的两个参数应该是通过压栈传递的。我们要关注的是 phdr->len 的传递，该参数是最右边的参数，在 C 语言调用方式下是第一个压栈。call 之前第一个压栈的是第 8 行的“push ecx”指令，因此 ECX 中存放了 len。这时只需分析第 6～7 行，看赋予了 ECX 什么值即可。

第 6 行将 ebp–8 指向内存中存的值赋给 EAX。由第 3 行解释可知，ebp–8 指向内存存放的是 buf 首地址。所以，该指令结束后 eax 指向 buf 首地址。

第 7 行中的 eax+4 说明将 buf 偏移 4 字节处存储的整数赋值给了 ECX，然后在第 8 行压栈 ECX 传递给 printf()函数。可知，偏移 4 字节是有问题的，我们的协议是第 1 字节为 flag，第 2 字节开始是长度信息，不是偏移 4 字节！（**哪里出了问题？**）对比 decodeHdr2 函数的反汇编（见 DM1-29）就会发现，它只从 buf 头部偏移了 1 字节。

DM1-29

```
1   int * pdatalen = (int *)&buf[1];
2       mov   eax, dword ptr [ebp+8]
3       add   eax, 1
4       mov   dword ptr [ebp-8], eax
5   printf("version2 len = %d\n", * pdatalen);
6       mov   esi, esp
7       mov   eax, dword ptr [ebp-8]
8       mov   ecx, dword ptr [eax]
9       push  ecx
10      push  415754h
11      call  dword ptr ds:[004182bch]
```

第 9 行 ecx 的值被作为长度传递给了 printf()函数。从第 7、8 行可以看到，将 ebp–8 指向内存中的值作为地址，该地址指向内存中的值赋给了 ECX。那么，ebp–8 指向内存中的值是什么？第 2～4 行对 ebp–8 内存进行了赋值。第 2 行将 buf 地址赋值给了 EAX（ebp+8 是参数 buf 的地址）。第 3 行将 eax+1，即 buf 偏移 1 字节。因此，该代码是从 buf 偏移 1 字节获取的长度信息。

为什么 decodeHdr1 会从 buf 偏移 4 字节的地方获取长度信息？因为它用结构体指针来访问：

```
phdr->len;
```

结构体会对齐，而对齐会引入填充字节。那么，这个结构体有填充吗？

```
struct hdr{
  char flag;
  int len
};
```

不论当前是以 4、8、16 的方式对齐，“盒子”的大小都是 4，flag 放入后，就无法放入 len（长 4），那么 len 和 flag 之间就有 3 字节的填充位，所以 len 的地址离头部偏移了 4 字节。这正是前面反汇编分析看到的那个奇怪的“偏移 4 字节”。

知道原因后，怎样修改？

① 可以将项目的对齐方式设定为 1，那么结构体的排列永远所见即所得，这个问题就解决了。但这必须要求双方都按一个模式选择对齐方式（大家请做实验验证一下），这从工程实践是很难满足的。

② 我们能否从协议设计出发，无论编程选择什么样的对齐方式，结构体都不会产生填充字节？例如：

```
struct hdr{
    int len
    char flag;
};
```

将两个字段交换，flag 确实是在 len 之后，之间没有填充字节。但我们考虑问题要“瞻前顾后”。首先，这个结构体的大小是多少？它依然是 8 字节，而非 5 字节，那么 flag 后面就有 3 字节的填充。**（这个填充我们不能视而不见，它会影响程序。）**按照协议，读完头部后，就要定位数据部分的地址，最简单的做法是 data=buf+sizeof(struct hdr)，拿到头部地址加上协议头的大小（sizeof(struct hdr)）。这就从缓冲头部偏移 8 字节，而不是按照协议的 5 字节。**（在处理数据的时候，我们依然是错误的，少读 3 字节！）**怎么办？只能调整字段的顺序，在没办法的情况下，甚至主动添加一些字段，防止填充情况发生。比如，将协议修改为：

```
struct hdr{
    char flag;
    char reserved[3];
    int len;
};
```

我们用 reserved 字段的 3 字节占据了可能发生的填充（选择 4、8、16 对齐时）。这样无论用户选择哪种对齐方式，都不会发生填充，都是所见即所得。更多的时候，我们应该尽量调整字段顺序，而不是主动添加填充字节来防止自动填充。比如，对如下协议头：

```
struct hdr{
    char flag;
    short len;
    char type;
};
```

应该调整为：

```
struct hdr{
    char flag;
    char type;
    short len;
};
```

参考本节前面的结构体 ip_hdr，它的字段排列就是典型的“无填充”排列方式。如果有兴趣，请参考 TCP/IP 协议簇的参考书，其中有很多用图展示的协议：一行 4 字节，协议中的字段排列非常“整齐”，没有“错位”的感觉，其实那就是无填充对齐的最形象展示。

1.6 switch 语句的思考

在学习 switch 语句之前，我们先提一个问题：

为什么 switch 只能判断整数类型的变量（包括 char、short、int 等，char 是 1 字节整数），而不能判断浮点数？

关于 switch，似乎还有一些说法，如它比等价的 if-else 组合要更快速。这一切都需要用反汇编揭开其工作原理。

1.6.1 switch 机制探索

许多时候，我们从猜测开始探索之旅。有时候，当毫无头绪时，观察可能是探索之旅的最好开端。源代码见 DM1-30，运行并反汇编调试。

DM1-30

```
void main(){
    int flag;
    flag = 20;
    switch (flag)
    {
        case 1:
            printf("1\n");
            break;
        case 2:
            printf("2\n");
            break;
        case 3:
            printf("3\n");
            break;
        case 4:
            printf("4\n");
            break;
        case 5:
            printf("5\n");
            break;
        default:
            printf("%d\n", flag);
    }
}
```

为了易于观察，选择 Release 版本（在“项目属性”页中将“配置属性 → C/C++ → 优化 → 优化”选项设定为禁用），这样可以避免很多干扰主题的不必要的代码。反汇编如下：

```
1    flag = 20;
2    00401006    mov   dword ptr [ebp-4], 14h
3    switch (flag)
4    0040100d    mov   eax, dword ptr [ebp-4]
5    00401010    mov   dword ptr [ebp-8], eax
6    00401013    mov   ecx, dword ptr [ebp-8]
7    00401016    sub    ecx, 1
8    00401019    mov   dword ptr [ebp-8], ecx
9    0040101c    cmp   dword ptr [ebp-8], 4
10   00401020    ja    0040107c
11   00401022    mov   edx, dword ptr [ebp-8]
12   00401025    jmp   dword ptr [edx*4+0040109ch]
13   {
14    case 1:
15        printf("1\n");
16   0040102c    push   4020f4h
17   00401031    call   dword ptr ds:[004020a4h]
18   00401037    add    esp, 4
19        break;
20   0040103a    jmp    0040108e
...
```

前面代码让我们迷惑，不知到底要干什么，这是经常遇到的情况。我们要找到一个支撑点，从该点用倒推的方法，常常就能找到突破口。这段代码的支撑点是什么？跳转。我们应该找到这些 case 分支语句跳转过来的地方。第 14 行是 case 1 开始的地方，那么应该在它前面找到跳转向它的语句。只找到两条跳转指令，分别是第 10 行和第 12 行。（哪个是我们的支撑点？）

第 10 行的跳转语句只会跳往 0x0040107c，当然不太可能跳转到 case 分支语句（0x0040107c 是 default 分支入口）。比如，case 1 对应的代码是第 16 行，其地址是 0x0040102c，并非第 10 行跳转的 0x0040107c。那么，这个跳转肯定不是我们寻找的“支撑点”。

第 12 行是一个无条件跳转指令，5 个 case 分支是“有条件”的。（是否也放弃呢？）如果放弃，真的就无法找到跳转语句了。更关键的是，我们无法迈过该指令，因为程序顺序执行下来必然执行到该跳转。该跳转后就是 case 1 分支的入口。该语句是跳转到 edx*4+0040109ch 指向内存中存储的地址。存储地址不同，跳转的地方就不同。只要执行该指令时 edx 值不同，即可能跳转到不同地址。

比如，edx 是 0，就会跳转到内存 0040109ch 存储的地址；edx 是 1，就会跳转到内存 00401010h 存储的地址……如果 0040109ch、00401010h 中存的地址不同，就跳转到不同代码块。

标志寄存器（Flags Register，FR）：存储一些运算状态和控制结果，如加减乘除运算均可影响它。其中的 ZF 表示判断结果是否为 0，CF 表示计算时是否有借位和进位。两者均以 1 代表真，0 代表假。一些条件转跳指令以它为判据。

sub 减法指令：执行减法，如“sub　ecx, 1”是将 ecx 的值减 1 然后赋值给 ecx。

cmp 比较指令：比较两个值的大小，如“cmp　dword ptr [ebp-8], 4”将 4 和地址为 ebp–8 的内存中存的值比较，相当于减法“dword ptr [ebp-8]–4”。

cmp 指令常与转跳指令配合，检测比较结果。

ja（jump if above，大于跳转指令），根据比较结果，如果是大于，则转跳，转跳条件是标志位 CF=0 且 ZF=0；后面跟转跳偏移量。

jb（jump if below，小于跳转指令），根据比较结果，如果是小于，则转跳，转跳条件是标志位 CF=1；后面跟转跳偏移量。

je（jump if equal，等于跳转指令），根据比较结果，如果是相等，则转跳，转跳条件是标志位 ZF=1；后面跟转跳偏移量。

条件转跳针对有符号和无符号数有两套指令：ja、jb 为无符号数转跳，jg、jl 为对应的有符号数转跳。转跳其他类型指令请自行上网查找资料。

用倒推法来分析，当 flag 取不同值时，是否将影响 edx 的值，从而跳转向不同分支块。

第 11 行将 ebp–8 地址中的值赋给 edx。再往上的代码就是不断对 ebp–8 地址指向的内存进行操控。

第 2 行将 14h 赋值给 ebp–4 指向的内存，即变量 flag。而 14h 就是 20，即要比较的值。

第 4～5 行的代码将 flag 即 20 赋给 ebp–8 指向的内存。

第 6～8 行将 ebp–8 内存中存的值减 1 后赋给 ebp–8 指向的内存。

第 9～10 行并没有改变 ebp–8 中的值，而是涉及了一个分支，我们可以暂时不管。

为了看得更清楚，我们将第 2～12 行画流程图，见图 1.59。

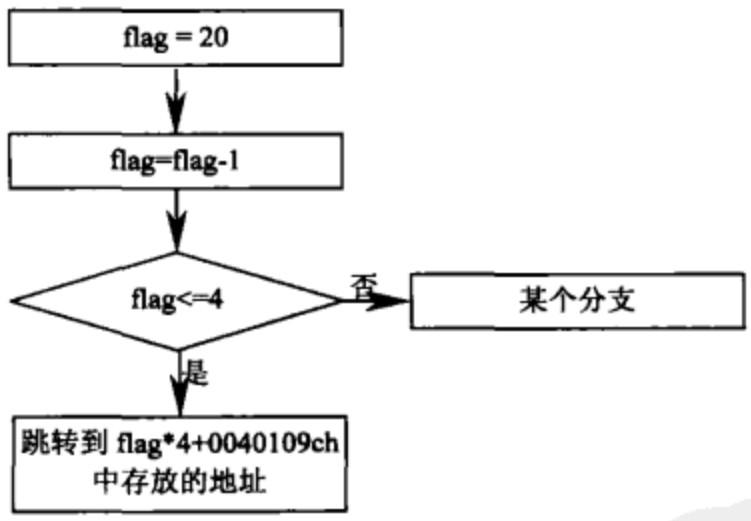


图 1.59

还是没有头绪？我们不妨将 flag 命中 1～5 分支的值都带入这个算法中看看。

flag=1 时，flag=flag–1，则 flag 为 0。而 0≤4，所以走 jmp 分支。flag * 4 + 0040109ch = 0040109ch，因此跳转到 0040109ch 地址指向的内存中存储的值。

flag=2 时，flag=flag–1，则 flag 为 1。而 1≤4，所以走 jmp 分支。flag * 4 + 0040109ch = 004010a0h，因此跳转到 004010a0h 地址指向的内存中存储的值。

flag=3 时，flag=flag–1，则 flag 为 2。而 2≤4，所以走 jmp 分支。flag * 4 + 0040109ch = 004010a4h，因此跳转到 004010a4h 地址指向的内存中存储的值。

flag=4 时，flag=flag–1，则 flag 为 3。而 3≤4，所以走 jmp 分支。flag * 4 + 0040109ch = 004010a8h，因此跳转到 004010a8h 地址指向的内存中存储的值。

flag=5 时，flag=flag–1，则 flag 为 4。而 4≤4，所以走 jmp 分支。flag * 4 + 0040109ch = 004010ach，因此跳转到 004010ach 地址指向的内存中存储的值。

0040109ch、004010a0h、004010a4h……每个地址间隔 4 字节，展示了一个以 0040109ch 为首址、元素大小为 4 的数组。图 1.60 指出了当 flag=1、2、3、4、5 时跳转到地址 1、2、3、4、5。存放这 5 个地址的内存是一个连续数组，跳转计算中 flag * 4 中的 4 是因为数组元素大小（所存地址的大小）为 4 字节。

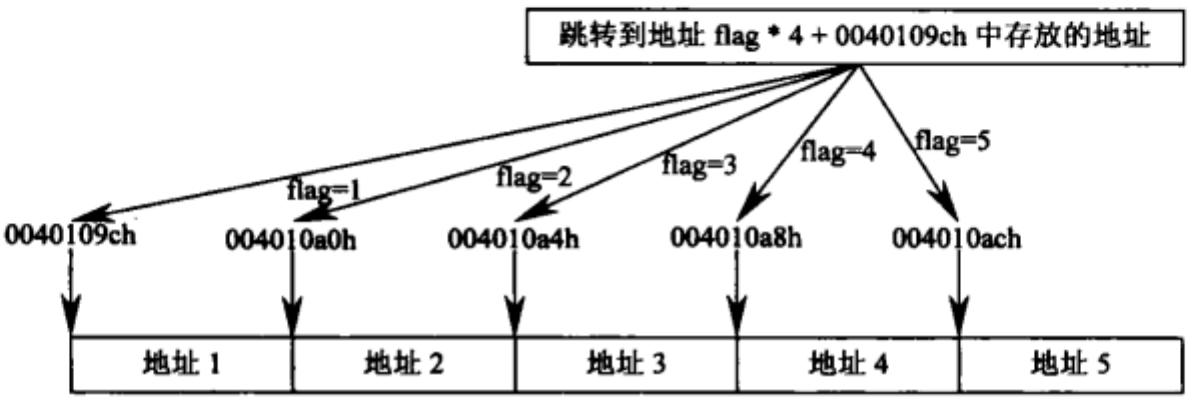


图 1.60

我们可以猜测，数组中存放的这 5 个地址应该是 case 的 5 个分支代码块的入口地址。（终于有了头绪，好吧，该实证了。）我们将第 1 行代码改为 flag=1，来看数组第一个元素 0040109ch 中是否存储的就是 case 1 的分支入口地址。设置断点，开始分析。图 1.61 为内存值和代码反汇编。

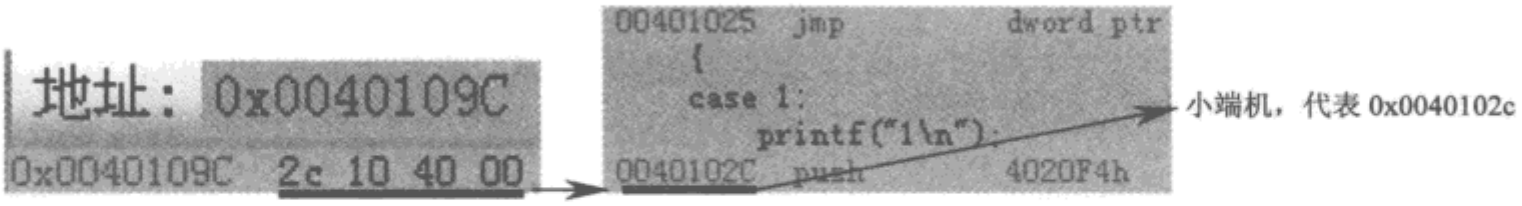


图 1.61

依此，分别观察数组的另外 4 个元素，可以证明其中分别是 case 2~case 5 代码块的入口地址。我们可以总结这个算法：

① i = flag – a，a 是 case 分支中最小的整数，本例中就是 1～5 中最小的 1，flag 是 switch 语句要比较的变量。

② 跳转到 addresses[i]中存储的地址。Addresses 是一个预先准备好的数组，其中存储了对应 case 分支的起始地址。

这真是一个绝妙的算法，不需要多次比较，不管 case 分支多少都直接获得跳转的地址，确实比 if-else 的等价组合快。因为它将 flag 映射为数组的索引，所以 flag 不能是浮点数。

仔细的同学会问，我们并没有将图 1.59 中的判断步骤描述到算法中。再进一步分析，发现没有考虑两个问题：

① 如果 flag 小于 1 或大于 5，怎样处理？

② 如果 case 分支涉及的整数并非连续，怎样处理？本例中分支涉及 1～5 的连续 5 个整数。如果考虑下面的代码，缺少了 case 2 会怎样？

```
switch (flag)
{
    case 1:
```

```
            printf("1\n");
            break;
        case 3:
            printf("3\n");
            break;
        case 4:
            printf("4\n");
            break;
        case 5:
            printf("5\n");
            break;
        default:
            printf("%d\n", flag);
    }
```

先分析问题②。如果 flag 为 2，应该执行 default 分支，因此我们只需将数组的第 2 个元素填写成 default 分支的代码入口即可，这样 jmp 语句跳转时将执行 default 分支。按此修改代码实验。反汇编代码见 DM1-31。算法并没有改变。从最后一行的命令分析，地址数组的起始地址变成了 0040108ch，第 2 个元素的地址是 0040108ch+4=00401090h。按照推理，这个地址中存储的值应该是 default 分支的地址。00401020 处的 ja 指令是跳转到 default 分支，其入口地址为 0040106ch，那么地址 00401090h 中应该存储 0040106ch。观察内存（见图 1.62），确实如此。

DM1-31

```
switch (flag)
0040100d        mov   eax, dword ptr [ebp-4]
00401010        mov   dword ptr [ebp-8], eax
00401013        mov   ecx, dword ptr [ebp-8]
00401016        sub   ecx, 1
00401019        mov   dword ptr [ebp-8], ecx
0040101c        cmp   dword ptr [ebp-8], 4
00401020        ja    0040106c
00401022        mov   edx, dword ptr [ebp-8]
00401025        jmp   dword ptr [edx*4+0040108ch]
```

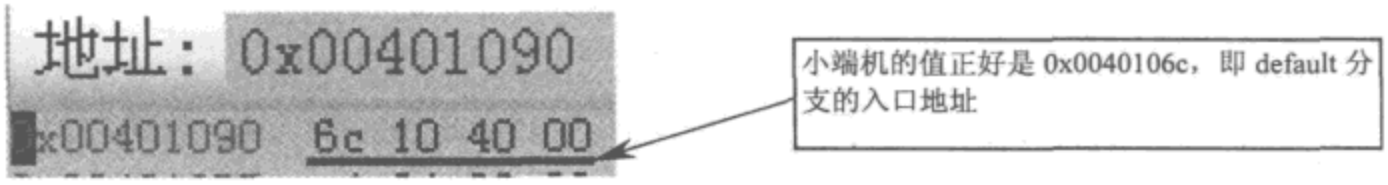


图 1.62

下面给出整个地址数组 5 个元素的值及其对应的分支入口地址，见图 1.63，椭圆提示了相关 case 分支的入口地址，虚线指向的是代码段该分支的入口地址，实线指向的是地址数组中对应元素，元素存储了该分支入口地址。

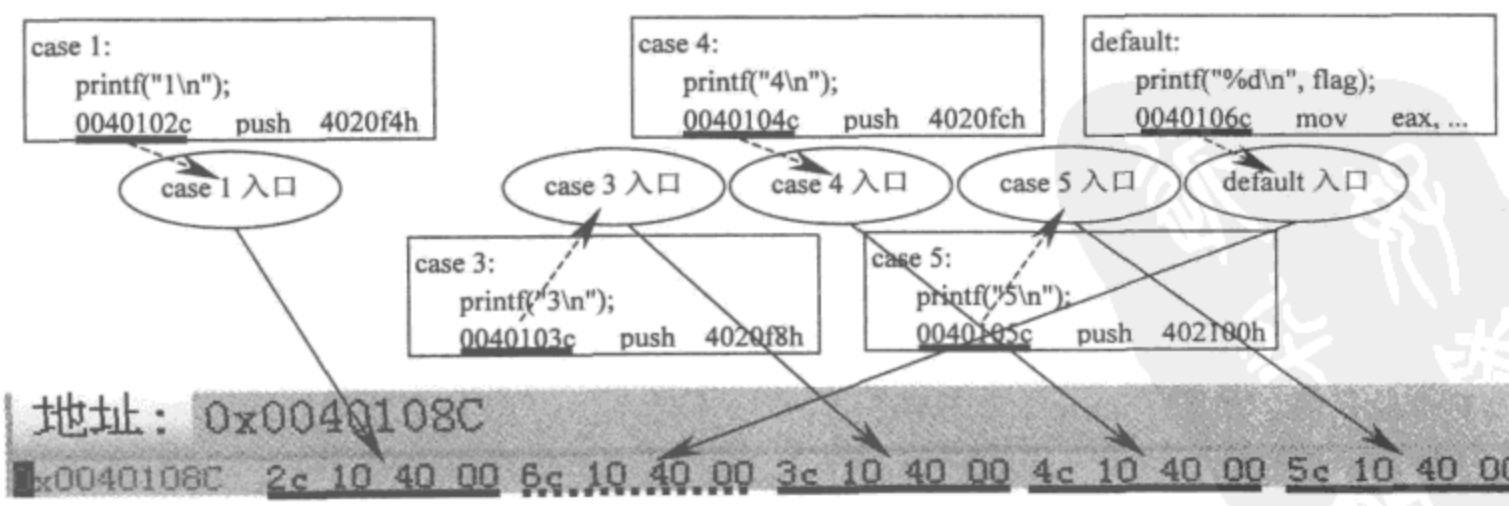


图 1.63

从图 1.63 可知，每个分支的入口地址放入数组对应元素，而缺少的分支（如该图中没有 case 2 分支），通过 flag-1 计算出对应元素的索引，其中（即索引为 1，图中内存中用虚线标出的 4 字节）存放的是 default 分支的入口地址。

下面分析问题①，对于 flag 大于 5 的情况。根据图 1.59，当 flag>4 时（因 flag=flag–1），转向的分支应该是 default 分支。因为如果 flag–1>4 意味着 flag> 5。从反汇编可看出，ja 指令后面跳转的地址正是 default 分支：

```
00401020   ja    0040106c
```

default 分支的反汇编如下，其入口地址是 0040106C（十六进制数）：

```
default:
    printf("%d\n", flag);
0040106c      mov   eax, dword ptr [ebp-4]
```

问题是，flag 小于 1 时也应该执行 default 分支，它能满足 flag–1>4 的条件吗？好像不满足。似乎我们应该写成如下判断：如果(flag–1) > 4 || (flag–1) < 0)，则执行 default 分支。这实际是两个条件判断，为什么代码只有一个对大于 4 的条件判断？

```
0040101c      cmp   dword ptr [ebp-8], 4
00401020      ja    0040106c
```

先看运行结果，将语句改成“flag=0;”。执行 flag–1 后，其值为–1，在内存中表示是 0xffffffff。执行“cmp dword ptr [ebp-8], 4”与 4 比较时，就是 0xffffffff 与 4 比较。如果是将 0xffffffff 当成无符号数，那么它就大于 4；如果当成有符号数，它是–1，就小于 4。所以关键要用什么样的类型比较，这取决于比较指令的选择。条件判断有两套指令集：针对有符号数，针对无符号数。ja 就是按无符号数来进行判断的，这样结果是满足大于 4 的条件。

我们可以再做一个简单的分析：当 flag 小于 case 的最小分支的值时，flag–case 的最小分支的值就是一个负数（本例中就是 flag–1），负数在用补码时，其最高位为 1，因此一个负数至少是 0x80000000，这是一个非常大的无符号整数，不出意外，它将大于我们 case 分支中的最大值。当然，大家可以从更严谨的角度去证明它，这已经超出本次任务的内容。这真是一个非常神奇的效果，利用有符号和无符号数的差异，我们将本来需要进行两次比较的过程变成一次比较。这也给我们一个重要的启示：

在数学的基础上，要结合计算机自身特点，才能最大限度地发挥其潜力和威力，不要将计算机程序完全等价于纯数学的过程。

最后，考虑了边界限制后的 switch 语句的算法，switch 描述见 DM1-32。

i_1，i_2，…，i_n 为整数，i_1 最小，i_n 最大。case i_m 的入口地址为 $address_m$，default 分支入口地址为 address_default。Addresses 数组称为跳转表，Addresses[index]中存放相关分支入口地址，index=i_m–i_1，im 属于 i_1，i_2，…，i_n。Addresses 数组中存放的是 $address_m$；当 i_m> i_1 且 i_m<i_n，但 i_m 不属于 i_1，i_2，…，i_n 时，Addresses 数组中存放的是 address_default。

DM1-32

```
switch(flag) {
    case i1: …
    case i2: …
    …
    case in: …
    default:…
}
```

（1）i = flag–i_1（i_1 是 case 分支中值最小的）。

（2）如果 i > i_n –i_1，则跳转到 default 分支，否则执行第（3）步。而这个条件比较用无符号比较指令 ja，从而可用一条指令判断 flag 是否属于[i_1, i_n]。（优化技巧。）

（3）跳转到 Addresses[i]中存储的地址。

1.6.2 switch 语句一定比 if-else 语句快吗

在 1.6.1 节，我们确实发现用跳转地址表的方法，switch 语句比 if-else 语句快，它的计算复杂度并不因分支的增加而增加。但真的一定快吗？应该从时间、空间代价上好好考察 switch 算法，体现工程中非常有价值的边界意识：哪里是我能承受的边界？哪里是正确的临界点？

首先，case 分支中的最小值和最大值决定了跳转地址数组的元素个数。我们可将一些情况推到极限，如 case 分支判断的数一个是 1，一个是 1M，请问这个跳转表有多少元素？1M 个，一个元素是 4 字节，那么，为了两个 case 分支的速度，需要付出 4MB 的跳转表，除了一头一尾，中间全是 default 的地址。（**太浪费了。**）时间还得与空间一起考虑，否则根本不适用。其次，即使 case 判断的数是 1 和 2，就一定会用跳转表方式吗？这时内存代价只有 8 字节，但该方式要做两次减法（一次 sub，一次 cmp），两条 if-else 组合也是两条减法，所以速度并不占优。

DM1-33

```
switch (flag)
0040100d        mov   eax, dword ptr [ebp-4]
00401010        mov   dword ptr [ebp-8], eax
00401013        cmp   dword ptr [ebp-8], 1
00401017        je    00401027
00401019        cmp   dword ptr [ebp-8], 3
0040101d        je    00401037
0040101f        cmp   dword ptr [ebp-8], 5
00401023        je    00401047
00401025        jmp   00401057
```

下面来看 case 数目对编译器选择算法的影响。

① 将前面代码只保留 3 个 case 分支：1、3、5。反汇编结果见 DM1-33。

3 条条件跳转指令就是对 1、3、5 的判断，1 条无条件跳转指令则跳转到 default 分支。这时完全与 if-else 等价。

② 在 case 分支：1、3、4、5（见 DM1-31）的 4 个分支情况时，采用跳转表方式。

再看跳转表大小的影响。现在将上面代码的“case 5”改成“case 0x12345678”，如果用跳转表，其元素个数将是 0x12345678 个，内存消耗太大，那么编译器肯定不应该采用跳转表。下面是相关反汇编代码见 DM1-34。其中 4 条黑体的比较语句就是对 4、1、3 和 0x12345678 进行判断，这时 switch 已经退化成一个 if-else 的等价实现了。因此，跳转表的空间代价决定了是否会采用该机制。

DM1-34

```
switch (flag)
0040100d        mov   eax, dword ptr [ebp-4]
00401010        mov   dword ptr [ebp-8], eax
00401013        cmp   dword ptr [ebp-8], 4
00401017        jg    0040102d
00401019        cmp   dword ptr [ebp-8], 4
0040101d        je    00401058
0040101f        cmp   dword ptr [ebp-8], 1
00401023        je    00401038
00401025        cmp   dword ptr [ebp-8], 3
00401029        je    00401048
0040102b        jmp   00401078
0040102d        cmp   dword ptr [ebp-8], 12345678h
00401034        je    00401068
00401036        jmp   00401078
```

实践是最会提问的老师，经过分析，我们发现了新问题：为什么 vc 在 4 个分支时用跳转表，见情况②；三个分支却是 if-else 方式，见情况①？仔细看指令，在 3 个分支下，if-else 组合是 3 条减法指令（cmp），跳转表方式有 2 条减法：一条 sub、一条 cmp。而且，跳转指令“jmp dword ptr [edx*4+0040108ch]”包含了一个乘法和加法，至少相当于 1 条减法。所以 3 个分支不用跳转表有其道理，占用空间却无速度优势。

所以，不能武断地说 switch 语句一定比 if-else 语句快，这里面有很多条件制约了它。希望大家从本例中学会用边界意识去思考问题，那将获益良多。

1.6.3 switch 的再次优化

上面看到了边界条件的力量，这里还没有考虑一种情况：分支条件比较多、但其整数比较“稀疏”的情况。分支数目多应该采用跳转表，而“稀疏”的整数又导致中间很多元素填写的都是 default 分支，似乎效率不高。同时，跳转表元素也不是“太多”，因此又允许我们采用跳转表。这种情况

应该如何处理？

```
switch (flag){
    case 10:
        printf("10\n");
        break;
    case 20:
        printf("20\n");
        break;
    case 30:
        printf("30\n");
        break;
    case 40:
        printf("40\n");
        break;
    case 50:
        printf("50\n");
        break;
    default:
        printf("%d\n", flag);
}
```

其中，跳转表元素 41 个（50–10+1），只有 5 个元素填写的是对应 5 个分支块的入口地址，其他 36 个填写的都是 default 的入口地址。从有效性而言，该数组有效元素非常“稀疏”。似乎 36 个重复的 default 入口地址有点浪费。有更好的办法吗？让我们开始探索之旅，看编译器怎样做，见 DM1-35。

DM1-35

```
switch (flag)
1 0040100d      mov   eax, dword ptr [ebp-4]
2 00401010      mov   dword ptr [ebp-8], eax
3 00401013      mov   ecx, dword ptr [ebp-8]
4 00401016      sub   ecx, 0ah
5 00401019      mov   dword ptr [ebp-8], ecx
6 0040101c      cmp   dword ptr [ebp-8], 28h
7 00401020      ja    00401083
8 00401022      mov   edx, dword ptr [ebp-8]
9 00401025      movzx  eax, byte ptr [edx+004010bch]
10 0040102c     jmp    dword ptr [eax*4+004010a4h]
{
    case 10:
```

有了 1.6.1 节的经验，我们很容易找到这段新代码的支撑点，即黑体所示的 3 条指令，从这里程序跳转到了不同分支。这段代码其实与之前版本有许多相似，如 00401016 处的“sub ecx, 0ah”指令与前面一样（0ah 为十进制的 10）。00401020 处的“ja 00401083”是跳到 default 分支，00401083 就是 default 的地址（从反汇编查看 default 地址可容易证明这点）。

第 1～4 行，执行 flag = flag–10。

第 5～7 行，将 flag 与 40（即 28h）比较，如果大于，则说明 flag 不属于 10～50，则跳到 default 分支。这同样使用了无符号比较的优化技巧。

第 8 行，将 flag–10 后的值赋给了 edx，在之前的算法中，edx 中就是跳转表数组的索引值了，flag 为 10 时，edx 为 0；flag 为 20 时，edx 为 10。

第 9 行，004010bch 似乎是跳转表的首地址，但与以前的[edx * 4+004010bch]不同，之前一个元素占 4 字节，此时 edx 没有乘 4，似乎元素大小是 1 字节。将 edx+004010bch 值简列如下：

flag = 10，则 edx = 0，edx+004010bch = 004010bch；

flag = 11，则 edx = 1，edx+004010bch = 004010bdh；

flag = 12，则 edx = 2，edx+004010bch = 004010beh；

……

从第 9 行指令中的 byte ptr 可知，这条 movzx 指令只会从 edx+004010bch 指向的地址读取 1 字节给 eax，而 movzx 会把 eax 的其他字节清零。由 byte ptr 可确定该数组元素大小为 1 字节。猜测第 10 行跳转到对应分支的语句，因为其后是 case 10 分支，只能从这里跳转到不同分支。那么，

eax*4+004010a4h 地址中存储的只能是分支的入口地址。同时，从 eax*4 能推测出 eax 应该是数组的索引。索引*4 为距数组首部偏移量，所以元素占 4 字节。以上分析见图 1.64。

（1）flag = 10（见图 1.64(a)）

flag =10;

edx = flag-10 = 0;

edx+004010bch = 004010bch;

movzx　eax, byte ptr [edx+004010bch]　　→　eax = 004010bch 地址存储的值=i1

执行 jmp　dword ptr [eax*4+004010a4h]指令，跳转到 i1*4 + 004010a4h 中存储的地址 address1（见图 1.64(b)），它应该是 case 10 的入口地址，数组元素大小为 4 字节。

（2）flag = 11（见图 1.64(c)）

flag = 11;

edx = flag-11 = 1;

edx+004010bch = 004010bdh;

movzx　eax, byte ptr [edx+004010bch]　　→　eax = 004010bdh 地址存储的值=i2

执行 jmp　dword ptr [eax*4+004010a4h]指令，跳转到 i2*4 + 004010a4h 中存储的地址 address2（见图 1.64(d)），它应该是 default 分支的入口地址，数组元素大小为 4 字节。

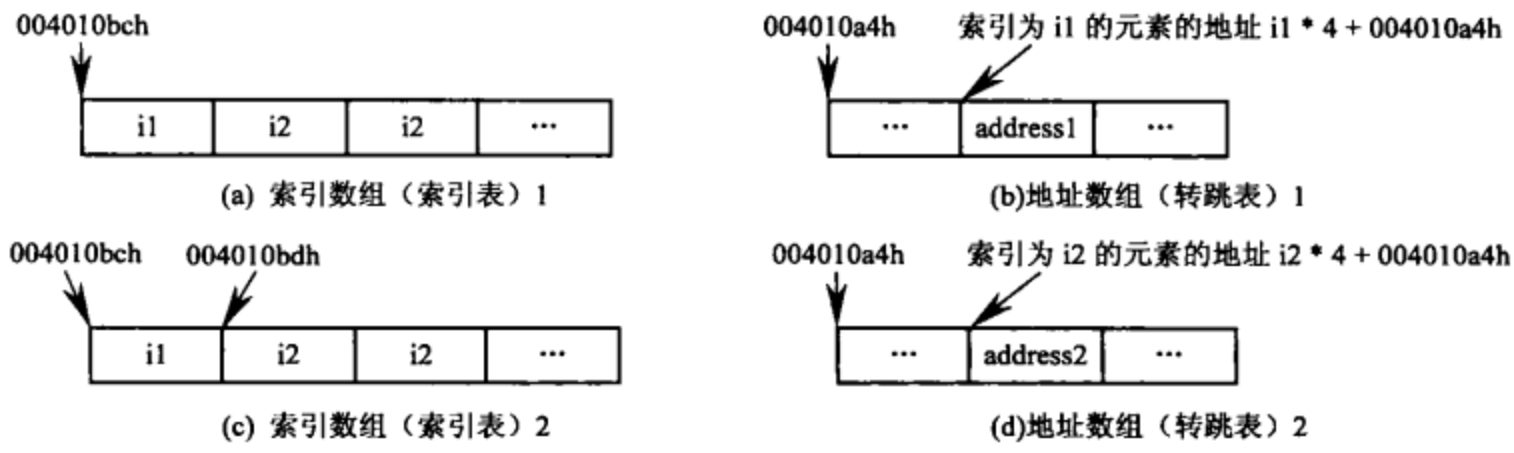


图 1.64

这时可推断，第 1 个数组（索引数组）存放的是第 2 个数组的索引号，元素大小为 1 字节。第 2 个数组存放的是跳转地址即跳转表，元素大小为 4 字节。switch 的做法是用 flag 换算出索引数组的索引号，然后从索引数组中获取该 flag 在跳转表中的索引号，再从跳转表中用该索引号查找存放的入口地址。

movzx　eax, byte ptr [edx+004010bch]指令说明 004010bch 是索引数组首地址，edx 是索引数组的索引，获取索引为 edx 的元素的值并赋值给 eax。eax 存储了跳转表中对应元素的索引号。

jmp　dword ptr [eax*4+004010a4h]指令执行后，跳转到 flag 在跳转表中对应元素所存的分支入口地址。其中，004010a4h 是跳转表首地址，eax 是上条指令计算出的跳转表的索引。eax*4+004010a4h 为跳转表中对应元素的地址，该元素存储分支入口地址。

这时，我们可以看出该算法的好处，见图 1.64(a)。对于 flag=12，它在索引数组中是第 3 个元素，而其中存的值为 i2，与第 2 个元素一样。因为它也是对应 default 分支，所以对应跳转表元素的索引均为 i2。这样，所有在 case 分支范围内并指向 default 分支的 flag 在跳转表中均共用索引为 i2 的元素。如果是像 1.6.1 节只有跳转表，则其他每个指向 default 分支的 flag 都要占用 4 字节存放 default 分支地址。本算法中，索引数组元素大小为 1 字节，所以非常节省。

下面来计算这两种方法的内存开销。

直接跳转表：跳转表共 50–10+1=41 个元素，每元素 4 字节，共 164 字节

索引表-跳转表：索引表元素 50–10+1=41 个元素，每元素 1 字节，共 41 字节。跳转表存储所有 case 分支入口地址和 default 分支地址，共 5+1=6 个地址，共 24 字节。合计 65 字节。

从时间开销来看，该算法依然不会因为分支数目增加而增加指令执行次数，比 if-else 好，只是比直接跳转表多了一次索引数组访问。

到目前，这些都是猜测，需要实证。图 1.65 是相关内存图，展示了两级索引表-跳转表的结构。

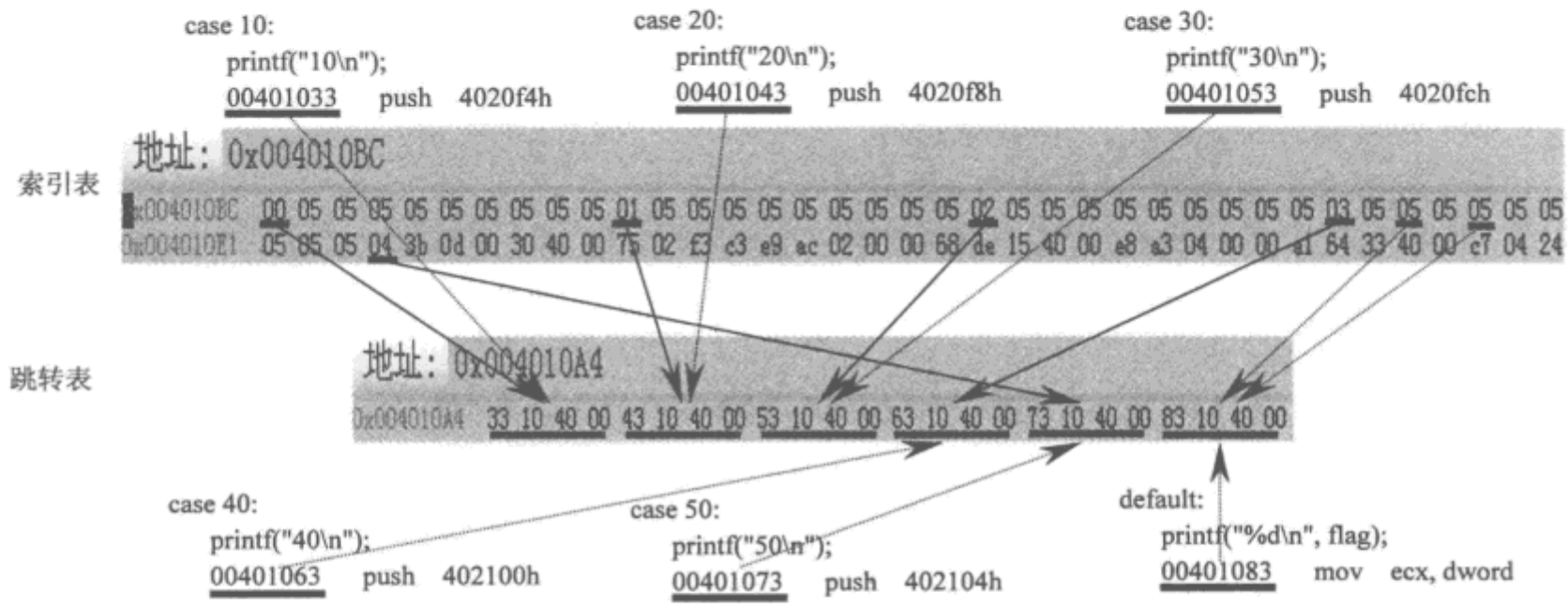


图 1.65

从指令 movzx eax,byte ptr [edx+004010bch]知道，索引表首地址是 004010bch，将其输入内存窗体。从指令 jmp dword ptr [eax*4+004010a4h]知道，跳转表的首地址是 004010a4h，将其输入内存窗体（在 VS 2005 及以后版本可打开多个内存窗体）。

索引表对应内存的第 1 行到第 2 行第 4 字节共 41 字节，一个元素 1 字节，与我们分析的索引数组的大小一致。其中黑色粗线标记的 5 个元素分别是 0、1、2、3、4，正好对应跳转表中 5 个元素的索引，索引对应的跳转表元素分别存储的是 case 10、case 20、case 30、case 40、case 50 的入口地址。

索引数组中的其他元素均为 5，说明索引为 5 的跳转表元素存储的是 default 分支的地址。索引时，如 flag 为 20，flag–10 的索引为 10，对应到索引数组其索引为 10 元素的值为 01，那么用 01 为索引，在跳转表中找到第 2 个元素，其值正好是 case 20 的入口地址 00401043h。

另外，每个分支的代码片段的入口地址用黑线标出，同时虚线指向了跳转表的对应元素。跳转表共 6 个元素。以 case 10 为例，其入口地址为 00401033h，虚线指向的跳转表第 1 项 33 10 40 00，其小端机表示的值为 00401033h，正好一一对应的。

如果大家还有疑惑，请根据索引表-跳转表的结构单步跟踪执行一次，查看相关变量，就能完全明白。这种最“笨”的方法有时也是最直接、最“聪明”的方法。

现在，我们总结 switch 语句的新算法：

```
switch(flag){
    case i1: …
    case i2: …
    case in: …
    default:…
}
```

i_1，i_2，…，i_n 为整数，i_1 最小，i_n 最大。case i_m 的入口地址为 $address_m$，default 分支入口地址为

address_default。Addresses 数组为跳转表，元素大小为 4 字节，其中存放所有 case 分支入口地址 $address_m$ 和 address_default，索引分别为 ia_m 和 $ia_{default}$。Indexs 数组为索引表，元素大小为 1 字节，Indexs[index]中存放对应 Addresses 数组中的相关索引 ia_m 和 $ia_{default}$，$index=i_m-i_1$，i_m 属于 i_1，i_2，…，i_n 时，存放的是对应的 ia_m；$i_m > i_1$ 且 $i_m < i_n$ 但 i_m 不属于 i_1，i_2，…，i_n 时，存放的是 $ia_{default}$。

（1）$i=flag-i_1$。

（2）如果 $i>i_n-i_1$，则跳转到 default 分支，否则执行第（3）步。这个条件比较用无符号比较指令 ja，从而用一条指令判断 flag 是否属于$[i_1, i_n]$。

（3）i_2 = Indexs[i]。

（4）跳转到 Addresses[i_2]中存储的地址。

最后还要考虑这个算法适用的“边界条件”：

① 当 case 分支中的整数比较“稀疏”时，可以压缩大量 default 分支入口地址消耗的空间。

② 由于索引数组元素大小为 1 字节，那么它指向跳转数组的容量限制为 256 个。也就是说，$i_n-i_1>255$ 时该方法失效。

思考：

针对分支数大于 255 时，我们能继续优化吗？（不妨看看编译器怎样处理。）

1.7 关于其他高级语言要素的反汇编学习

在目前为止，我们学习了赋值、函数调用、指针强制转换、结构体、switch 语句的反汇编，没有涉及 goto（非常简单，等价于 jmp 指令）、for 循环、while、if-else 等，请大家自行学习。遇到不熟悉的指令，请在网上查找。

1.8 全局变量的疑问——重定位和程序结构

1.8.1 独一无二的全局变量

全局变量是整个程序独一的吗？一个全局变量对应了一个地址，当该地址内存被赋值后，程序中任何函数访问该地址内容均可获得修改后的内容。既然一个全局变量对应一个地址，是否该地址分配给该程序用，其他程序就无法用了呢？或者，一个程序运行两份实例（即两次双击同时运行一个 EXE 程序），如果两个实例的全局变量指向同一个地址，那么当其中一个实例修改全局变量时，另外一个实例是否也看到这个修改后的值呢？全局变量地址相同，则共享一个内存。

设计一个实验程序，见 DM1-36（工程见“code\第 1 章\testglobalvarvalue”）。

控制台（选择“开始 → 运行”，然后输入“cmd”，回车）中要运行一个程序时，经常在要运行的程序后输入一些命令开关，如

```
C:\>netstat -a -n
```

netstat 是要运行程序的名字，-a 和-n 是参数。这些参数通过 main()函数传给程序，这样命令行程序才能与用户交互。访问这些参数需要两个信息：一是以空格分开的参数个数，二是参数本身。main()函数第 1 个参数就是参数个数。注意：有一个隐含参数就是该程序名本身，如 netstate 是第 1

个参数（如果启动命令是“C:\Windows\netstat...”，则本参数是 C:\Windows\netstat）。如何访问参数？main()函数的第 2 个参数是数组的指针，数组元素类型是 char *，如果要访问第 1 个参数，获取该参数串的地址就用 args[0]。

DM1-36

```
#include <stdio.h>
int gi;
void usage() {
    printf("please input s to set gi or d to dispaly gi\n");
}
void main(int argCount, char ** args){
    if (argCount != 2) {
        usage();
        return;
    }
    if (args[1][0] == 's') {
        gi = 12;
        printf("gi = %d\n", gi);
        getchar();
    }
    else if (args[1][0] == 'd') {
        printf("gi = %d\n", gi);
    }
    else
        usage();
}
```

以上代码编译出的程序叫 testglobalvarvalue.exe。该程序目的如下：先以 s 为参数运行，即“testglobalvarvalue.exe s”，这时程序会走 args[1][0]=='s' 分支，将全局变量 gi 设定为 12，然后在 getchar 处等待用户回车。这时该程序在运行，用 d 参数再次运行该程序，此时会打印 gi 变量，因为此时前一个实例在运行状态。这两个变量的地址相同，是否应该打印 12？这里有一个小技巧：激活 cmd 窗体，将资源管理器中 testglobalvarvalue.exe 的图标拖入到 cmd 窗体，那么该执行程序的全路径会自动出现在 cmd 窗体中，只需要回车就可运行该程序了。图 1.66 是执行结果，表明第 2 个实例（即带参数 d 的那个）并没有打印出 12，依然是 0。似乎该程序的两个实例的全局变量并没有共享同一块内存。

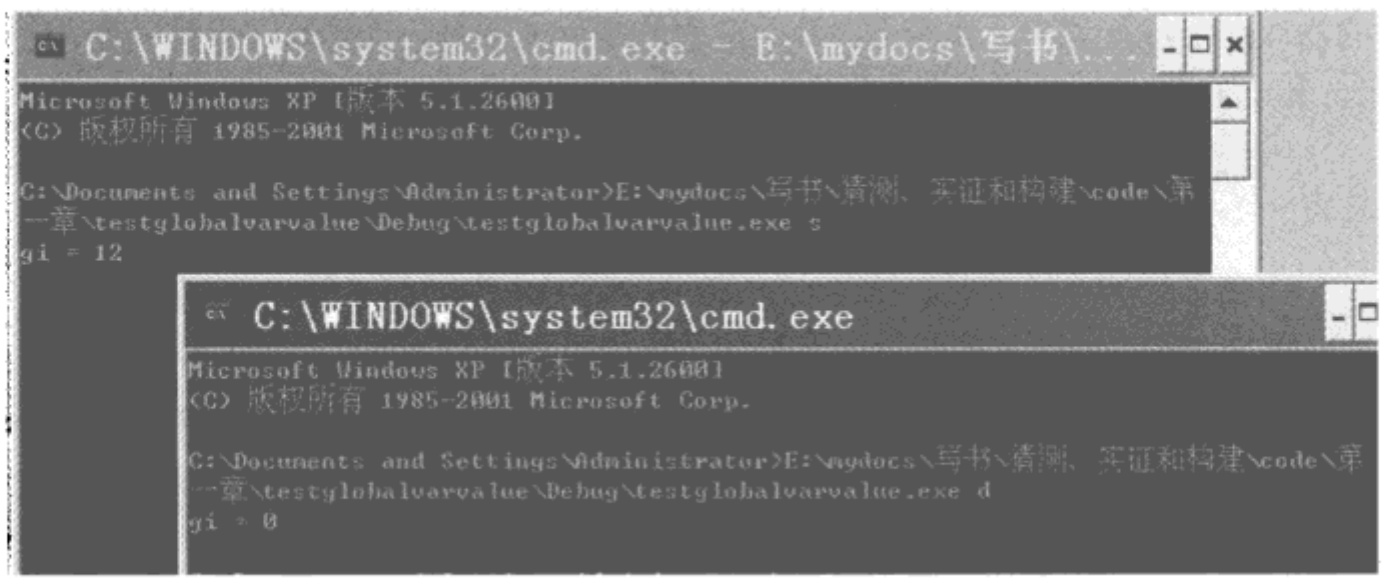


图 1.66

难道这两个实例全局变量 gi 的地址不同？我们首先回忆与全局变量赋值相关的 mov 指令，全局变量的地址值是嵌入到指令中的，两个运行的实例都是从一个执行程序运行的，其地址不可能是两个吧？为了保险，我们在 main()函数一开始加入一条 printf 语句，将该 gi 的地址打印，确实两者地址是同一值。为什么一个实例设定该地址内容却不影响另一实例读该地址时读到的内容？它们没有共享一块内存吗？

在 CPU 保护模式下，每个执行进程（程序的一个实例）都拥有自己独立的线性地址空间，这种机制叫虚存系统。我们可以举一个简单的例子，见图 1.67。

有两个寄存处，它们的包裹都放到同一个仓库中。仓库有 1000 个寄存位，编号 1～1000。寄存处对外提供 100 个寄存位，对外发放寄存位的号牌从 1～100 进行编号，这些编号显然并非仓库中寄存位的编号。如果用户要获取自己寄存的东西，向寄存处提交它发放的号牌（上有客户编号），

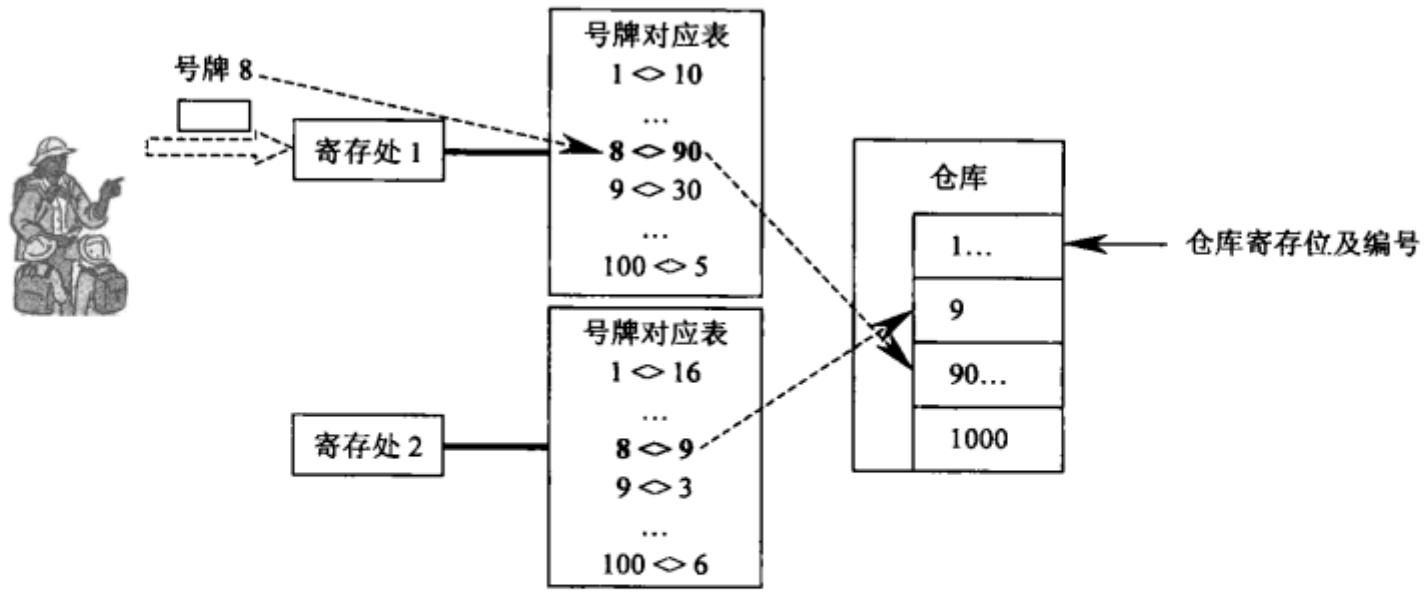


图 1.67

管理者在客户编号与仓库编号的对应表中找到仓库的编号，然后从仓库中提取该物品。比如，寄存处 1 的某客户提交的编号是 8，从图 1.67 可知对应表中的仓库编号是 90，那么管理者就从仓库编号 90 的位置提取客户物品。而寄存处 2 的 8 号牌对应的仓库编号是 9，不同寄存处的同样编号其实对应不同的仓库编号，并不是同一仓库寄存位。

这就是一个简单的虚存系统，寄存处就是进程（运行的程序实例），仓库是真正的物理存储系统，仓库编号是物理内存的地址，寄存处编号是线性地址。程序访问内存用的是线性地址（mov 指令中包含的地址信息是线性地址，用户态程序无法直接访问物理内存），而虚存管理系统用该地址找到对应的物理地址进行访问。表面上，两个进程的线性地址是一个地址（就如两个寄存处同一编号的号牌），其实对应的物理内存是不同的，并不会冲突。

1.8.2 不变的地址和重定位

在分析全局变量时，有如下代码：

```
gi = 0x12345678;                          //特殊值是个好东西，0x12345678 是标记，便于查找
00401003 C7 05 74 33 40 00 78 56 34 12 mov dword ptr ds:[00403374h],12345678h
```

如果将 gi 的地址打印，会发现每次运行其值都是 0x00403374。为什么 gi 总是能加载到这个地址？进一步说，该反汇编中的黑体部分的机器码是 mov 指令中全局变量的地址 0x00403374，也就是说，双击运行程序之前，在编译时，编译器就知道该 gi 的地址是 0x00403374 了。难道编译器能未卜先知？为了进一步证明，我们可以在执行程序中查找这段机器码（被赋值 0x12345678 形成的特殊机器码 78 56 34 12）是否是直接生成到执行程序中。

用 UltraEdit 打开该程序。（安装 UltraEdit 后，在要打开的程序图标上单击右键，然后选择“UltraEdit-32”）。按 Ctrl+F 组合键激活 UltraEdit 的查找界面，见图 1.68，输入要查找的十六进制数，并取消“查找 ASCII”复选框的选中。

单击“查找下一个”按钮，只能查找到一处，说明整个执行程序只有这样一条 mov 指令。我们发现，确实在生成执行程序时编译器已经知道了 gi 的地址。如果 gi 不在 0x00403374 处，那么 mov 指令必然不是对 gi 赋值，整个结果都是错误的。

为什么 gi 总是能被放在 0x00403374 呢？这个地址不会被其他程序占用吗？不会，因为这个地址是线性地址。从 1.8.1 节可知，每个进程都有自己独立的 0～4GB 的线性地址空间，线性地址不会相互干扰。即使对同一程序，编译器怎样知道它一定会加载到同一个固定地址，从而 gi 地址总

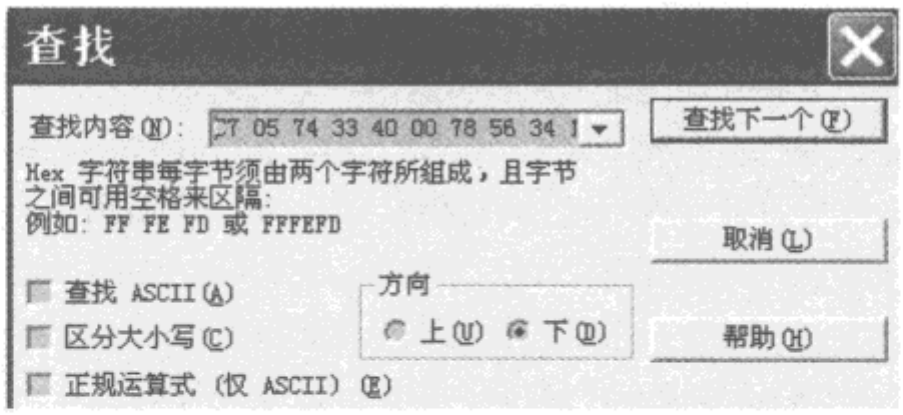
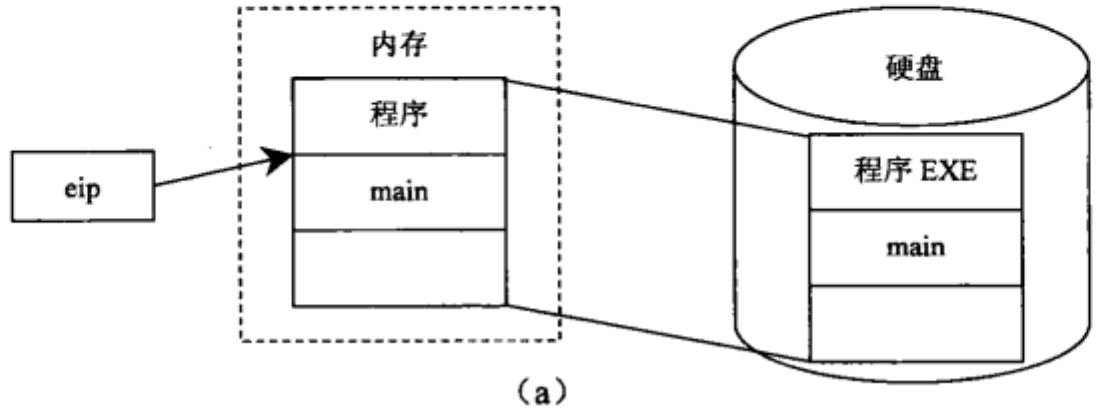


图 1.68

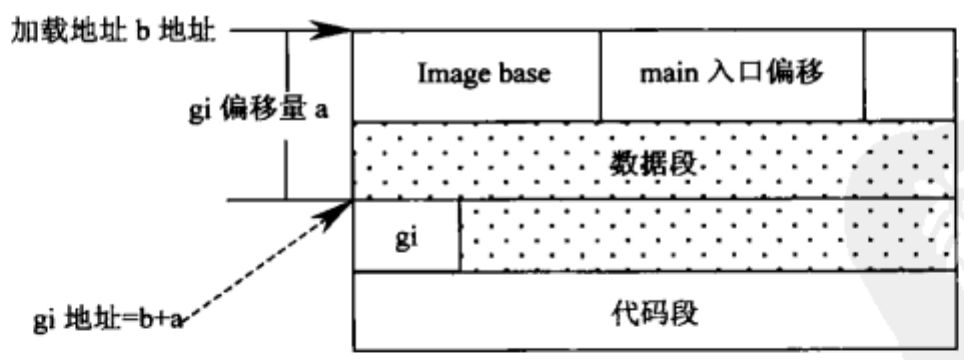
是 0x00403374 呢？（**如果我们完成这样的效果，应该怎样做？这种“我来做、怎么做”的朴素想法往往是分析和思维深入的基础。希望大家养成这种习惯。**）来构思如何让一个执行程序从硬盘加载到内存运行，本节进行猜测、提出自己的方案并实现，1.8.3 节将在实际程序中实证猜测结果。

假想程序就是一块数据，指令的机器码、全局变量都在这块数据中，我们可称存放代码的区域叫代码段，存放全局变量的区域叫数据段。执行程序就是这样的数据块。如果将它读出并放入内存，再将 EIP 寄存器指向 main()函数的入口，那么程序就被加载且从 main()开始运行。

图 1.69(a)表示从右边硬盘将执行程序完全读入左边的内存，然后 EIP 寄存器中的值指向 main()入口。完成加载程序并将 EIP 寄存器中的值指向 main()入口的程序称为加载器。那么，加载器如何知道 main()的入口？可以构想，执行程序就是一个表，这个表有很多表格，其中某个表格给出了 main()的位置。加载器读取表格，获取这些信息即可。那么，main()的入口位置怎样表示？是加载后 main()的线性地址还是其他什么？其实，main()放在程序的某个位置可以用它相对程序头部的偏移量表示，这样不管程序加载在哪个地址，用偏移量+程序头部地址都能够计算出 main()的地址，这样更灵活，见图 1.69(b)。



（a）



（b）

图 1.69

如果用程序头部中一个部分记录 main()入口相对程序头部的偏移量，可以计算出 main()的地址，让其得以执行。但程序加载到内存后头部的地址如何确定？

一种做法是加载器自由选择可加载地址，将程序加载到该处。一种是程序告诉加载器它希望被加载到哪里，让加载器保证加载到该地址。在选择这两种方法之前，先分析来自全局变量赋值的一个限制，也是我们到目前为止一切分析的初衷。全局变量赋值的语句是一条 mov 指令，它要赋值的地址固化在该指令中，意味着编译时就知道全局变量地址。（**一个限制，一个非常关键的限制。**）

因为 gi 相对程序头部的偏移量 a 是确定的，所以 gi 地址= b（加载地址）+a（见图 1.69(b)）。如果加载器自由加载程序到任意地址，那么每次加载，b 就是随意的，则 gi 的地址就是随意的，但 mov 指令中包含的是 gi 的固定地址 0x00403374。因此，不能随便加载到一个地址。倒过来推，如果 gi 要加载到 0x00403374，那么 b=0x00403374–a，b 必然是一个固定地址。如果程序要求加载器将程序成功加载到 0x00403374–a，那么 gi 的地址就是 0x00403374，则 mov dword ptr ds:[00403374h], 12345678h 指令就是正确的。因为整个程序都是编译器生成的，那么它在编译期就能确定所有的全局变量相对头部的偏移量，只要程序加载到编译器希望加载的地址，则所有全局变量地址在编译期都可计算出：

全局变量地址=程序头部加载地址 b+全局变量相对程序头部的偏移量 a

因此，在程序加载地址问题上，为了编译器能计算出全局变量地址，我们应该采取由程序指定它希望被加载的地址，而非加载器随机加载。为了告诉加载器希望加载的地址，我们将执行程序中分配一个区域来存储这个希望加载的地址，称为 image base（基址），见图 1.69(b)。

现在又出现一个新的问题：

加载器真的能保证程序加载到我们希望的 image base 吗？如果不成功，该怎么办？

首先，从 1.8.1 节的保护模式下，程序拥有自己独立的 0～4GB 空间。只要程序模块是第一个加载到这个空间的，那么肯定能保证它可加载成功。不过这个地址空间并非全部被我们的程序占用，在 Windows 系统中，其 2～4GB 地址空间分配给了内核空间，用户程序在低 2GB 空间。进一步说，低 2GB 空间的高地址部分其实也被一些系统模块（如 kernel32.dll）使用，它们不是用户程序的一部分。但只要我们指定合适的位置（一般编译器会指定加载地址为 0x00400000），剩下的空间已经足够加载我们的程序了。当然限制是，我们的程序是第一个加载的用户模块，所以能随便使用剩下的可用地址空间。如果一个程序有多个模块需要加载，比如一个程序要加载多个动态链接库（dynamic link library，即后缀名为 .dll 的文件），那么就很难保证它指定的加载位置没有被其他模块占用。因此程序只能被加载到非指定的位置，就产生了一个问题。在全局变量赋值代码 mov dword ptr ds:[00403374h], 12345678h 中，全局变量地址是固化到指令中的，但现在全局变量不在希望的地址，那么这条指令必然出错，见图 1.70。

希望加载的地址 0x00400000 与实际加载地址 0x00500000 之间的偏移量为 0x00100000。此时，gi 的地址也要在原来的 0x00403374 基础上加上这个偏移量，其新地址为 0x00503374。由于代码段的 mov 指令依然是访问 0x00403374 地址，这就不对了，所以需要加载后，将该指令中代表地址的 4 字节修改成 0x00503374。我们称因加载到非希望地址而产生的修改过程为重定位。

可是，加载器如何知道哪些指令需要修改？加载器解析出所有的包含绝对地址赋值的 mov 指令，然后修改吗？且不论这个方法可行否，仅仅反汇编整个机器码的开销就非常大了。有更简单快速的方法吗？仔细分析，需要修改的信息都是绝对地址信息，因为整个模块移位了，所以这些地址信息需要修订。而这些指令都是编译器生成的，在生成的时候就知道哪些指令需要修改。比如，上例中的 mov 指令相对程序头部的偏移量是确定的，而其包含地址部分的起始位置相对程序头部也是确定的，见图 1.71。

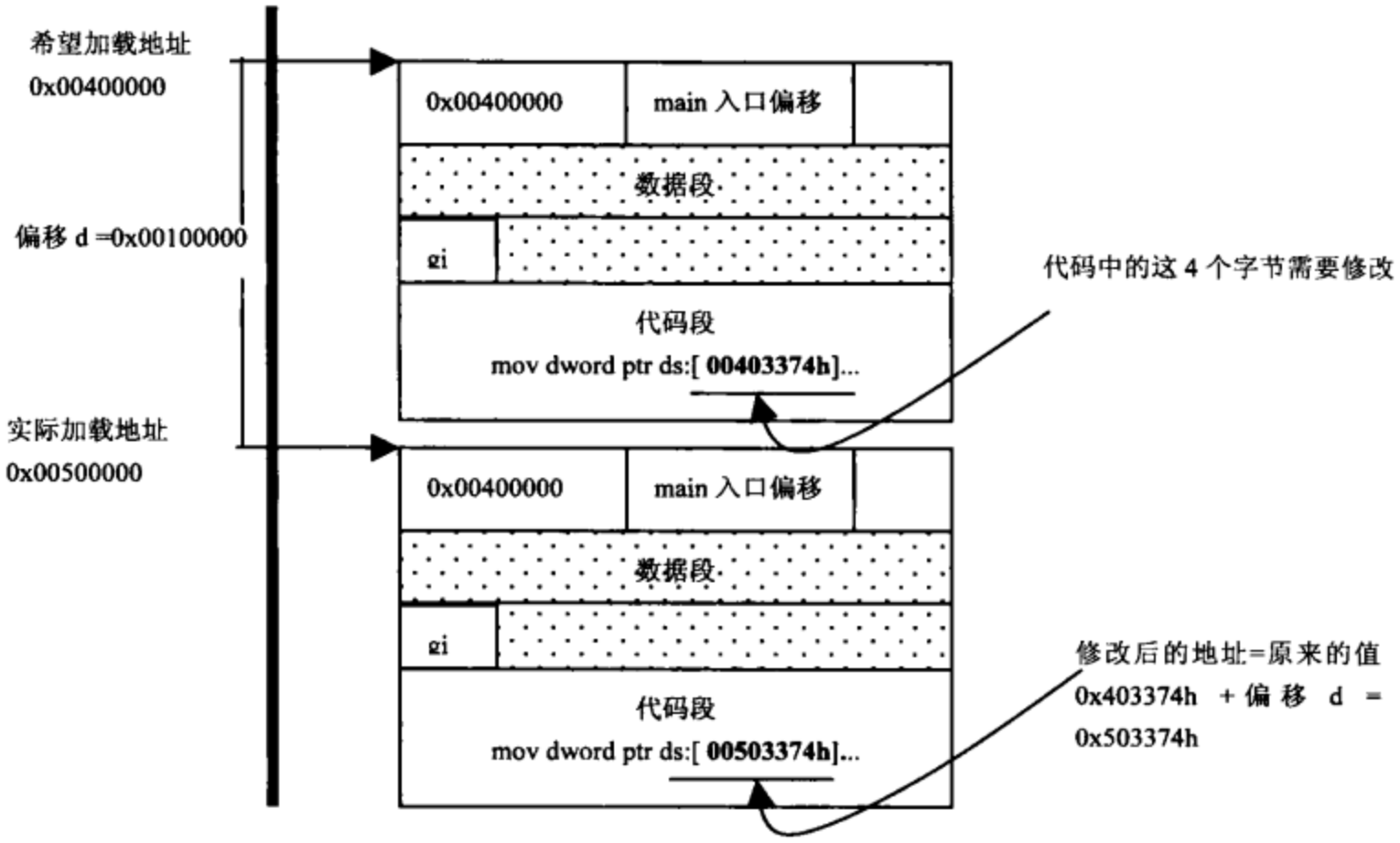


图 1.70

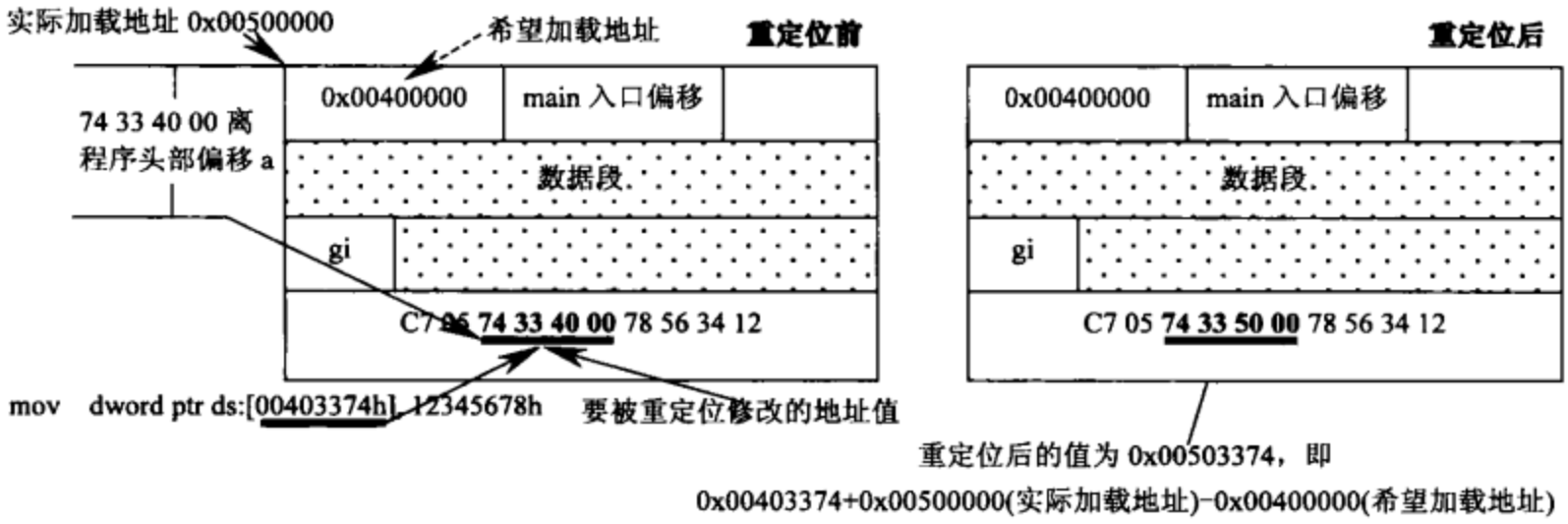


图 1.71

只需给出这个偏移量 *a*，就可如下计算，甚至完全没有必要去理解指令类型就能完成重定位。

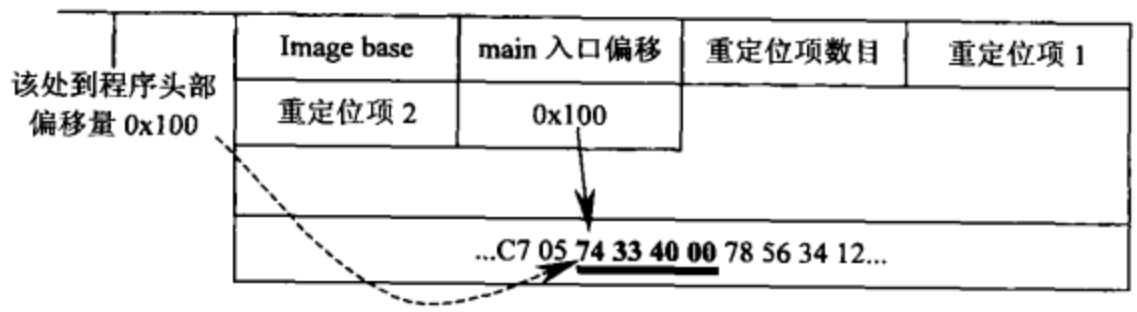
$$\text{relocAddr} = \text{actualBase} + a \tag{1}$$

$$*\text{relocAddr} = *\text{relocAddr} + \text{actualBase} - \text{expectedAddr} \tag{2}$$

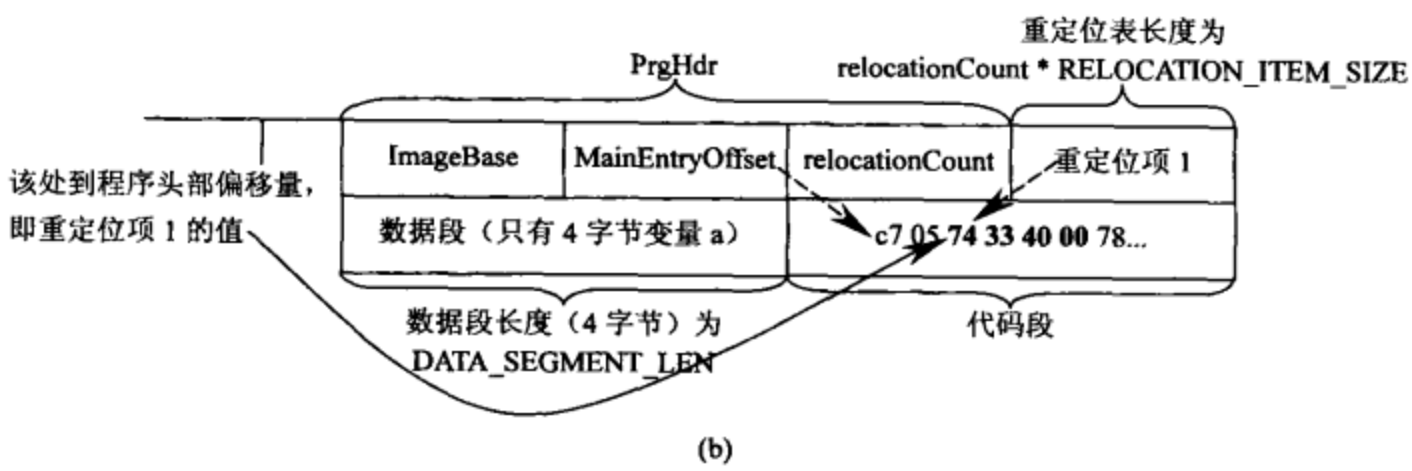
relocAddr 是加载后指令中存地址信息的地址，actualBase 是实际加载基址，*a* 是指令中存地址信息的起始位置相对程序头部偏移量，expectedAddr 是希望加载的基址。第二个等式右边的 *relocAddr 代表重定位前 mov 指令中的地址值。在图 1.71 中，actualBase 的值为 0x00500000，expectedAddr 的值为 0x00400000，原*relocAddr 的值为 0x00403374，重定位后为 0x00503374。

现在，剩下的问题就是加载器从哪里获得参数 a。既然是编译器知道 a，那么就应该由它来提供。只要编译器将这些信息放入执行文件，那么加载器就可利用。试想程序中有很多对全局变量的访问，以及对函数指针、全局变量指针引用的语句，它们都涉及对相关指令中地址信息的重定位，因此 a 不是一个，而是有许多条。那么，执行程序应该设计一个表来存储这些偏移量，即重定位表（relocation table）。我们如下简单设计，在 main()入口偏移段后就是重定位表，前 4 字节表示有多少个重定位项，后续的 4 字节的重定位项（即偏移量 a）连续存放，见图 1.72(a)：第 3 个重定位项内

容为 0x100，当程序加载到非希望地址需要重定位时，该重定位项表示从程序头部偏移 0x100 字节的位置存储的地址信息需重计算，它指向的位置就是之前例中 mov 指令中包含赋值地址的位置（c7 05 ...是 mov 指令的机器码，74 33 40 00 是要重定位的地址值）。



(a)



(b)

图 1.72

我们用如下结构体来概括设计的程序头部（因为是对齐的，所以不会出错）：

```
struct PrgHdr{
    int imageBase;              //程序希望被加载的地址
    int mainEntryOffset;        //主函数相对程序头部的偏移量
    int relocationCount;        //重定位项数目
};
```

给出重定位算法如下：

```
struct PrgHdr *pHdr;
```

① 从执行文件中读取 12 字节到内存，其地址为 pHdr。

② offset = actualBase - pHdr->imageBase，actualBase 为实际加载基址。

③ 从文件头部第 13 字节开始，循环 pHdr->relocationCount 次。

每次读 4 字节，获得重定位偏移量赋值给 relocationOffset。

对内存地址为 actualBase+relocationOffset 的 4 字节设定新值，新值=旧值+offset，即

```
*(int *)(actualBase + relocationOffset) = *(int *)(actualBase + relocationOffset) + offset
```

现在已经完成第一步，我们找到了一种方案处理全局变量寻址的重定位问题。（**当然不止全局变量地址**。）那么，下一步就是通过构建来看看该想法是否可行。最后在实际程序中印证重定位的想法，这放在 1.8.3 节。

为了简洁，我们如下设计验证程序，工程见“code\第 1 章\myprogram_load”。首先分配一块内存代表程序，由头部、重定位表、数据段和代码段组成，格式见图 1.72(b)。内存块头部 12 字节按结构体 PrgHdr 的格式设计。重定位表紧随头部的 12 字节。数据段只有一个 int 变量表示全局变量 a，紧随重定位表。代码段紧随数据段，由两条指令组成：一条 mov 指令对 a 赋值，一条指令为 ret 返回。重定位表有一个重定位项，它指向 mov 指令中的被赋值地址的起始。然后将该程序块复制到另

外一段内存，如果需要重定位，则重定位，并根据 mainEntryOffset 的指示，跳到代码段入口函数（即上面那条 mov 指令）执行。然后我们分别在加载执行前和执行后，打印数据段全局变量 a 的值，展示执行前后的不同。

```
//这个例子将展示加载以及重定位的效果按之前的格式构造出程序块，返回其首部地
//址赋给 prgLoaded
void main(){
    void * prgLoaded = makeProgramWillRelocate();
    char prgLocation[PROGRAM_LEN];
    //将程序从 prgLoaded 加载到 prgLocation，这模拟从硬盘加载到内存
    loadProgram(prgLoaded, prgLocation);
    //执行前，先打印出程序块中全局变量的初始值
    printTheGlobalVar(prgLocation);
    //执行加载的程序
    runProgram(prgLocation);
    //执行后，打印出程序块中全局变量的初始值，此时值为模拟程序执行后改变的值
    printTheGlobalVar(prgLocation);
    free(prgLoaded);
    //------------
    //这个例子将展示加载以及不重定位的效果
    prgLoaded = makeProgramWithoutRelocate();
    ...
}
```

关键函数包括：makeProgram、buildCode、loadProgram。buildCode 将代码构建到 codeStart 开始处，varAddress 为要赋值的全局变量的地址。

```
void buildCode(char * codeStart, int varAddress) {
    //mov  [00403374h], 12345678h 的机器码 c7 05 74 33 50 00 78 56 34 12，其中
    // c7 05 表示 mov，第 3~6 字节为被赋值地址，后 4 字节为被赋值
    //构建 mov 指令码---
    codeStart[0] = 0xc7;
    codeStart[1] = 0x05;
    //----
    int * pVarAddr;
    //第 3 字节开始的 4 字节是 mov 指令中的地址
    pVarAddr = (int *)&codeStart[2];
    //构建 mov 指令的地址部分
    *pVarAddr = varAddress;
    int * pVar;
    //从第 7 字节开始是 mov 中被赋值
    pVar = (int *)&codeStart[6];
    //构建 mov 指令中被赋值，该值为 0x12345678
    *pVar = 0x12345678;
    //构建 ret 指令，RET_CODE 宏定义了 ret 的机器码，ret 紧邻 mov 指令存放
    *(codeStart + MOV_CODE_LEN) = RET_CODE;
}
```

buildCode 代码块由两条指令组成：

```
mov  [varAddress], 0x12345678
ret
```

makeProgram 分配一块内存，并按格式构建整个程序块，expectedBase 为期望的加载地址。

```
void * makeProgram(int expectedBase){
    int relocationTbLen;
    //分配程序块
    struct PrgHdr * prg = (struct PrgHdr *)malloc(PROGRAM_LEN);
    //填写希望加载地址
    prg->imageBase = expectedBase;
    //设定重定位项个数为 1 项
    prg->relocation Count = 1;
    relocationTbLen = (prg->relocationCount * RELOCATION_ITEM_SIZE);
    //代码离首部偏移为 PrgHdr 长度+重定位项长度+数据段长度
    prg->mainEntryOffset = sizeof(struct PrgHdr) + relocationTbLen + DATA_SEGMENT_LEN;
    //填写重定位项，位于头部后所以 prg+1，mov 指令在代码段开始，其地址值部分从指令第 3
    //字节开始，该部分也离代码段首部 2 字节。因此离头部的偏移量为 prg->mainEntryOffset+2
    *(int *)(prg + 1) = prg->mainEntryOffset + 2;
    char * codeStart = (char *)prg + prg->mainEntryOffset;
    //计算在期望加载地址下全局变量地址
    int expectedGlobalVarAddress = expectedBase + sizeof(PrgHdr) + relocationTbLen;
    buildCode(codeStart, expectedGlobalVarAddress);
    int * pGlobalVar = (int *)((char *)prg +  sizeof(PrgHdr) + relocationTbLen);
    //给全局变量赋初值为 1，相当于 C 语言中的 int gi = 1，即程序加载后执行 main 之前的值
    * pGlobalVar = 1;
    //返回该程序模块的地址
    return (void *)prg;
}
```

loadProgram 加载程序，将 prgLoad 程序块的内容，加载到 prgInMem 中。

```
void loadProgram(void * prgLoaded, void * prgInMem) {
    memcpy(prgInMem, prgLoaded, PROGRAM_LEN);
    struct PrgHdr * prgHdr = (struct PrgHdr *) prgInMem;
    //从头部的 imageBase 字段获取希望加载地址
    int expectedBase = prgHdr->imageBase;
    //如果希望加载的地址等于实际加载的地址，则不需重定位
    if (expectedBase == (int)prgInMem) {
        printf("need not relocate\n");
        return;
    }
    //需要重定位
    else {
        printf("need relocate, expected address is 0x%x, but actualAdress is 0x%x\n",expectedBase, prgInMem);
        printf("relocation count is %d\n", prgHdr->relocationCount);
        int *pRelocationItem = (int *)(prgHdr + 1);
        //实际加载地址和希望值的差
        int offsetLoad = (int)prgInMem - expectedBase;
        //获取要重定位的所有地址
        for (int i = 0; i < prgHdr->relocationCount; i++) {
            //获取原来的地址值
            int * pRelocateAddress = (int *)((char *)prgInMem + *pRelocationItem);
            int oldValue = * pRelocateAddress;
```

```
            //计算新的地址值
            * pRelocateAddress = oldValue + offsetLoad;
            //计算下一个重定位项
            pRelocationItem ++;
            printf("make a relocation,the value from 0x%x to 0x%x\n",oldValue,*pRelocateAddress);
        }
        return;
    }
}
```

runProgram 从 mainEntryOffset 获得程序代码入口的偏移量，并用 call 指令调用它。

```
void runProgram(void * prgInMem){
    struct PrgHdr * prgHdr = (struct PrgHdr *)prgInMem;
    //代码入口相对头部偏移(mainEntryOffset)+程序加载后首址=代码入口地址
    void * mainFunctionAddress = (void *)((char *)prgInMem + prgHdr->mainEntryOffset);
    //调用入口函数
    _asm call mainFunctionAddress;
}
```

程序展示了重定位和没有重定位两种情况，并在程序执行前和执行后打印了全局变量值。执行前打印结果为 1，这是全局变量的初值；执行“mov [address], 0x12345678”，对该全局变量赋值后，打印结果为 0x12345678，见图 1.73。

```
need relocate, expected address is 0x400000, but actualAdress is 0x12ff34
relocation count is 1
make a relocation, the value from 0x400010 to 0x12ff44
global varable in the loaded progran is 0x1
global varable in the loaded progran is 0x12345678
need not relocate
global varable in the loaded progran is 0x1
global varable in the loaded progran is 0x12345678
```

图 1.73

真实的重定位机制什么样？参考 Windows 的 PE 格式（参考文章“Peering Inside the PE: A Tour of the Win32 Portable Executable File Format”，并可用 LordPE 来查看和学习），其差异在于为了节省重定位项的空间，不是用 4 字节表示到程序头部的偏移量。而将需要重定位的部分划分成一个个区（section）：

总偏移=区起始的偏移 +2 字节表示的区中的偏移

区的起始偏移为 4 字节，区内重定位项只占 2 字节，这样就大大节省了空间。PE 格式中希望加载的地址是 ImageBase，程序的入口偏移是 AddressOfEntryPoint，重定位表入口是 DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC]。

下面一段代码来自笔者以前的一个安全实验。为了能将后门 dll 隐藏，加载后便将其从硬盘上删除，因此不能用 LoadLibrary 加载，需自己加载。这段代码完成了重定位工作（代码是 Delphi 编写的，ptr^等价于 C 语言的*ptr，$等价于 C 语言的 0x，:=是等号）：

```
RelocationRVA := (FPCurrentItem^ and $0FFF) + FRelocationRegionRVA;
FixupAddress := PCardinal(PChar(ActualImageBase) + RelocationRVA);
FixupAddress^:=FixupPointAddress^ + Cardinal(ActualImageBase) - FExpectedImageBase;
```

其中，FRelocationRegionRVA 是一个重定位区首部相对程序头部的偏移，FPCurrentItem^ and $0FFF 将重定位项的 12 位取出作为区内相对偏移。RelocationRVA 是重定位处相对程序头部的偏移。

第 2 行代码用实际头部地址+RelocationRVA 计算重定位被修改部分的入口地址（FixupAddress）。最后一行修改该地址指向的 4 字节，完成重定位，其值为 FixupAddress 指向的旧值+实际加载地址和希望加载地址的差。

1.8.3 动态链接库中的重定位

在 1.8.2 节中展示了自定义的重定位机制及其实现，对于真实的重定位过程我们通过实验来实证。由于执行程序（*.exe）不会被重定位，所以编译器不会生成重定位表（例外情况请做习题 13），需要用动态链接库验证重定位机制。

先介绍动态链接库的工作原理和编程接口。有了 1.8.2 节对加载的认识，可如下简单理解动态链接库。首先，结构上，它与 EXE 文件一样是一个模块，包含代码段、数据段等，甚至有入口点，只不过这个入口点是当 DLL 被加载时执行一次，而非执行程序那样，作为整个程序的起始点。其次，dll 的目的是将一些程序的功能块从物理上分离出来，需要时由程序加载到内存。它对外暴露了可调用的函数，加载程序能获取这些函数地址，从而调用它们。这样做的好处有以下几点：① 如果代码可复用，那么只需要一份文件副本，节省空间。② 形成一定程度的抽象和隔离，用户只是通过函数名获取函数指针，并不知道函数的实现。那么，我们可通过替换新 DLL 以提供新实现，从而在不影响程序其他部分的情况下修改程序实现。插件的机制就要依赖动态加载。③ 能将大项目分解成彼此独立的 DLL，便于任务分配，并行开发调试。

Windows 下动态链接库相关有 3 个重要的 API：LoadLibrary，将 DLL 从硬盘加载到内存；GetProcAddress，接收函数名作为输入，返回该函数入口地址，当获得函数地址后，即可调用它；FreeLibrary，卸载已加载库。

① HINSTANCE LoadLibrary(LPCTSTR lpLibFileName)：传入参数为字符串指针，类型即 char *，指向被加载文件的名字，可以是包含路径的全名，如“C:\\tem\\mydll.dll”，也可只有文件的名字。如果是后者，系统将搜索加载该 dll 的执行程序所在目录、当前目录、系统目录等(具体请参考 MSDN 之 LoadLibray 条目)。返回为句柄类型，是一个整数，其实质上就是被加载库的加载基地址（即 1.8.2 节中的 imagebase）。如果失败，返回 NULL。该函数有两个版本 LoadLibrayA 和 LoadLibraryW，前者参数为单字节字符，后者参数为 Unicode 字符。

② FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName)：第 1 个参数为 LoadLibray 的返回值，表示欲获取函数所在模块；第 2 个参数为 char *，指向函数的名字。返回值为函数入口地址，如果失败，返回为 NULL。注意，返回值被定义为一个函数指针 FARPROC，其函数原型在这里没有意义，因此可将其安全地强制转换为被获取函数的真正原型，如后面的例子。

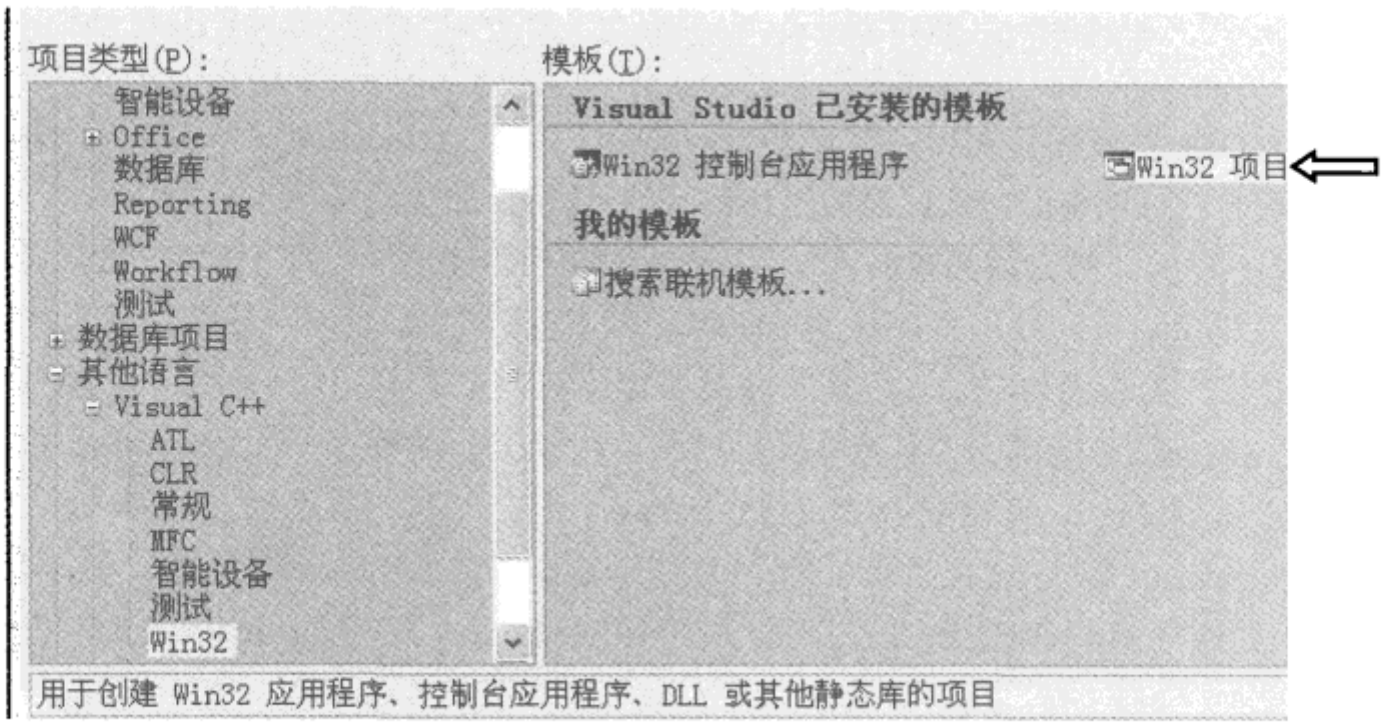
③ BOOL FreeLibrary(HMODULE hLibModule)：卸载已加载库，传入参数为 LoadLibray 返回的句柄。操作系统采用引用计数方式控制库加载和卸载。如果一个进程多次调用 LoadLibray 加载同一个文件，则引用计数记录调用 LoadLibray 的次数。FreeLibrary 将引用计数减 1，如果为 0，则真正从内存中卸载该 dll。卸载成功，返回非 0 值，否则为 0。

在 VS 2008 中，用图 1.74 的方式建立一个动态链接库（见“code\第 1 章\dll_relocation\dllload”）。

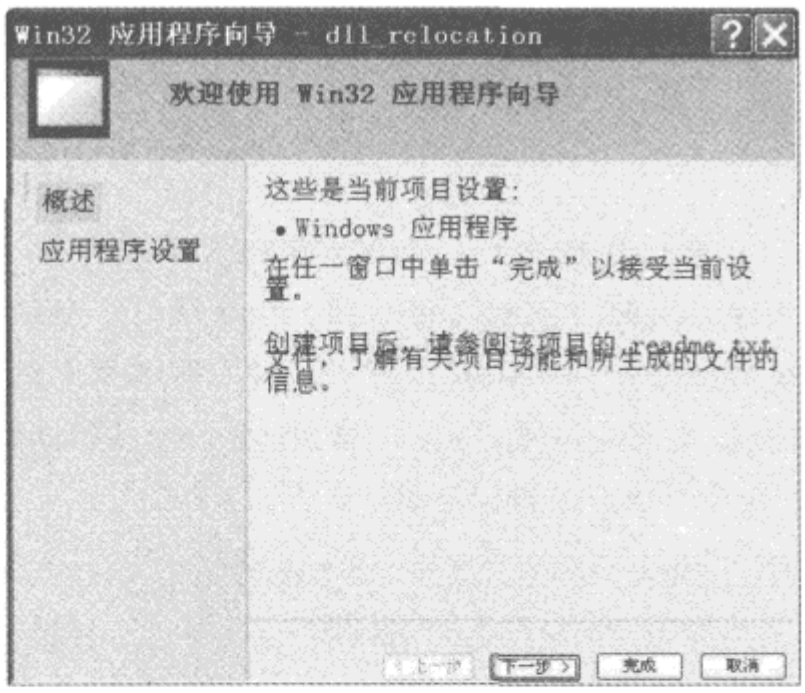
我们将项目如前设定为不使用预编译头部，并删掉所有其他文件，只保留 dll_relocation.h 和 dll_relocation.cpp。再添加一个 sub()函数到头文件中：

```
extern "C" int __declspec(dllexport) sub(int x, int y)
```

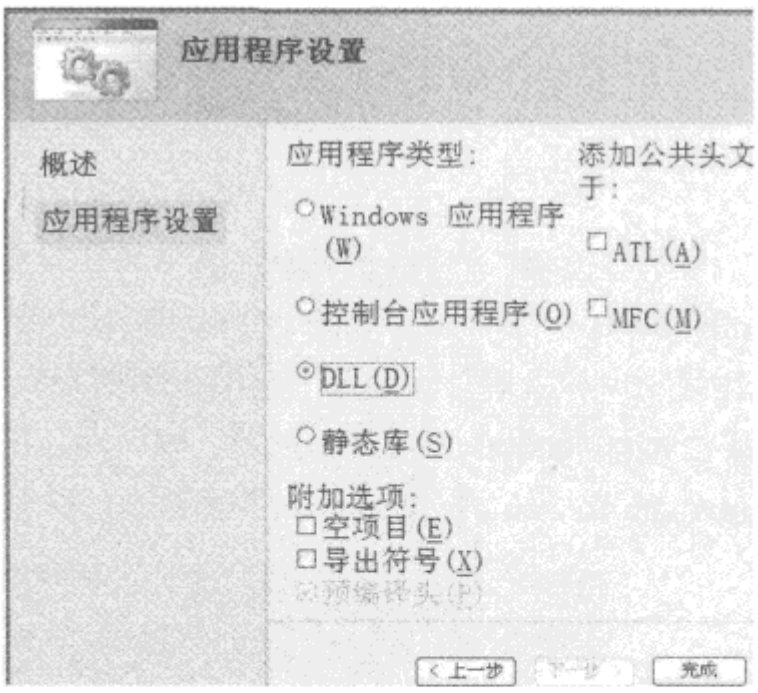
其中，extern "C"和__declspec(dllexport)代表 sub 是 dll 对外输出的函数，这些函数才能被 GetProcAddress 查找并返回入口地址。



(a)



(b)



(c)

图 1.74

选择资源管理器中根节点的“解决方案”并单击右键，在弹出的快捷菜单中选择“添加→新建项目”，再建立一个 Windows 控制台程序来调用该 DLL。

```
#include <stdio.h>
#include <windows.h>
typedef int (*SubPtr)(int, int);
void main() {
    //dll_relocation.dll 和 testdll.exe 被自动生成到一个目录，所以 dll 名字没包含路径
    HMODULE h = LoadLibraryA("dll_relocation.dll");
    if (NULL == h) {
        printf("cannot load\n");
        return;
    }
    SubPtr fp = (SubPtr)GetProcAddress(h, "sub");
    if (NULL == fp) {
        printf("cannot get function sub\n");
        return;
    }
```

```
    printf("3 - 2 = %d\n", fp(3, 2));
}
```

现在可开始 DLL 的重定位实验，工程见“code\第 1 章\dll_relocation”。将同一个 DLL 重命名后放在同一目录下，对系统而言这就是两个 DLL，加载时会加载两份副本。因为两个 DLL 规定的 imagebase 是一样的，势必后加载的 DLL 无法被加载到希望地址，而 DLL 中包含访问全局变量的代码,必然发生重定位。我们只要能调试跟踪进后加载的 DLL 的对应函数，反汇编，则可看到其代码被修改了，指令中与全局变量地址相关的部分被重定位了。为了对比，必然要保证第一个加载的 dll 的 imagebase 能满足，不能与执行程序（加载 DLL 的程序）的 imagebase 重合。那么，查看执行程序的 imagebase，再设置 DLL 的 imagebase，使之与执行程序的 imagebase 不重合。用 LordPE 查看 imagebase（解压 tools 目录下的 LPE-DLX.zip，双击 loadpe.exe 即可），单击 PE Editor，通过打开的“文件”对话框选择要打开的文件 dllload.exe，其中的 ImageBase 字段的值是 0x00400000，见图 1.75。

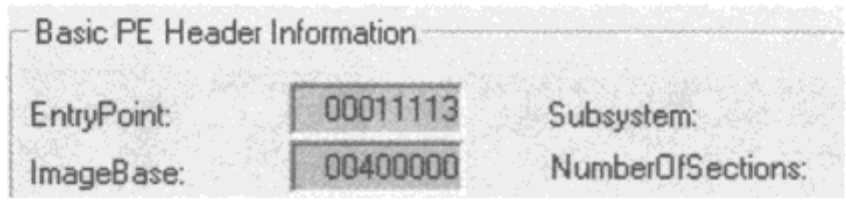


图 1.75

然后设定 dll 的 imagebase。执行文件约 30KB，只要选择地址大于 0x00400000+30K 即可，如 0x00500000。然后让 VC 生成的 dll 基址为该值。右键单击解决方案的 dll_relocation 项目，在弹出的快捷菜单中选择“属性”，在出现的对话框中选择“链接器 → 高级”，在“基址”后的文本框中“输入 0x00500000”（见图 1.76）。编译后，用 LordPE 查看生成的 DLL，其 imagebase 值变成了 0x00500000。如果在 Windows 7 平台上，执行文件工程需禁止“随机基址”（见图 1.76）。

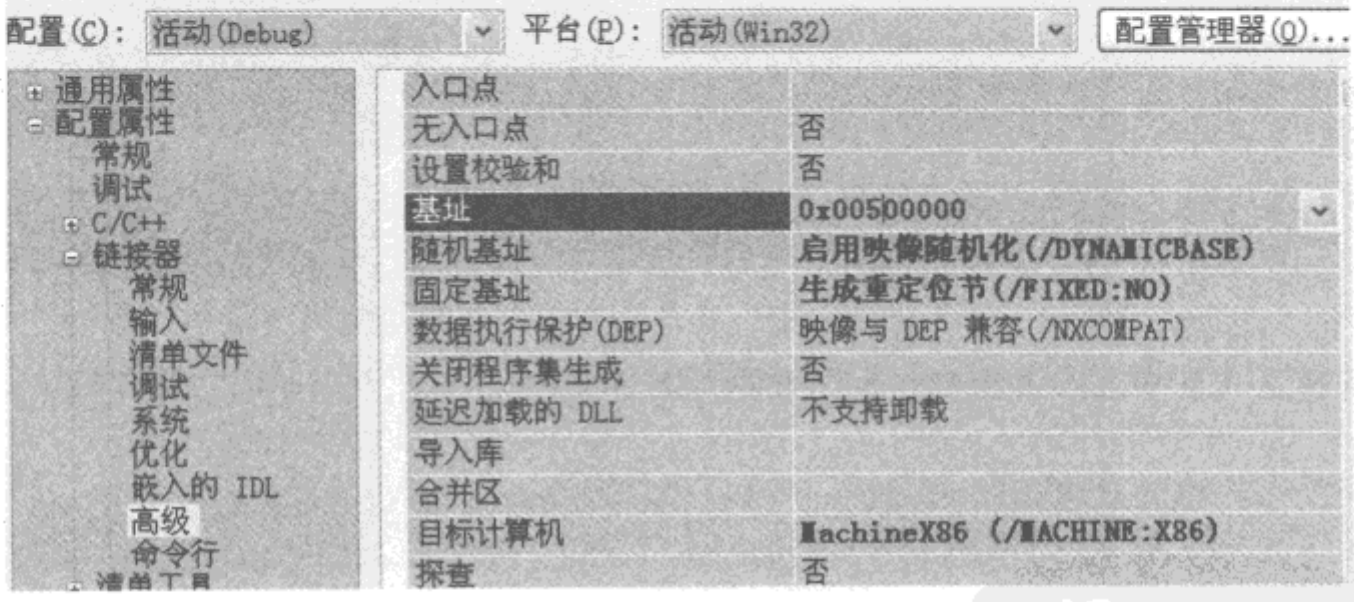


图 1.76

为了简单，我们在 sub()函数中增加一段对全局变量的访问“gi=10”，其对应的 mov 指令将会包含重定位部分，在调试时即可查看。将 dll_relocation.dll 复制到和同一目录下，重命名为 dll2.dll。然后建立一个新的控制台工程“testrelocation”，加载两份 DLL。

Testrelocation 工程代码如下：

```
#include <stdio.h>
#include <windows.h>
typedef int (*SubPtr)(int, int);
```

```
void main(){
    HMODULE h_withoutRel = LoadLibraryA("dll_relocation.dll");
    if (NULL == h_withoutRel) {
        printf("cannot load\n");
        return;
    }
    SubPtr fpWithoutRel = (SubPtr)GetProcAddress(h_withoutRel, "sub");
    if (NULL == fpWithoutRel) {
        printf("cannot get function sub without relocation\n");
        return;
    }
    HMODULE h_withRel = LoadLibraryA("dll2.dll");
    if (NULL == h_withRel) {
        printf("cannot load\n");
        return;
    }
    SubPtr fpWithRel = (SubPtr) GetProcAddress(h_withRel, "sub");
    if (NULL == fpWithRel) {
        printf("cannot get function sub with relocation\n");
        return;
    }
    fpWithoutRel(3, 2);
    fpWithRel(3, 2);
}
```

LoadLibray 分别加载了 dll_relocation.dll 和 dll2.dll 两个内容一样的动态链接库，dll2.dll 希望加载到 0x00500000 处，而该处已经被先加载的 dll_relocation.dll 占用，所以会发生重定位。即调用 fpWithRel 时，用反汇编跟踪进即可查看到重定位现象。

这个实验包含两部分：① 应该先证明第一个加载的 DLL 没有重定位；② 应该证明第 2 个加载的 DLL 重定位了。关于①，跟踪 fpWithoutRel，查看其 gi = 0x12345678 对应 mov 的机器码，然后用 UltraEdit 打开 dll_relocation.dll，输入该机器码，如果只查找到一处，即证明文件中的指令加载到内存后没有被修改。关于②，我们首先可反汇编跟踪进 fpWithRel，对照 gi=0x12345678 的 mov 指令，其 mov 指令中地址部分的 4 字节应该被修改了。然后用重定位原理计算第 2 个 DLL 文件中 gi 的地址，看它是否等于反汇编 mov 指令中显示的地址。第 2 个 DLL 文件加载后中，gi 地址=第 2 个 DLL 加载地址 + gi 到程序头部的偏移量（第 1 个 gi 的地址值-第 1 个加载地址即 0x00500000）。

右键单击解决方案中的 testrelocation 项目，在弹出的快捷菜单中选择“设为启动项目”(调试运行时会启动该项目)。在 main()函数中，在 fpWithoutRel(3, 2)处设置断点，执行后在断点处停止，打开反汇编，按 F11 键，跟踪进该函数的 call 指令。对应 gi=0x12345678 的指令如下：

```
gi = 0x12345678;
0051131e c7 05 38 71 51 00 78 56 34 12      mov   dword ptr ds:[00517138h],12345678h
```

用 UltraEdit 打开 dll_relocation.dll，将 c7 05 38 71 51 00 78 56 34 12 输入查找，见图 1.68，确实只能查到一条指令。

下面验证重定位。在最后一行处设置断点（该函数属于 dll2）。选择“调试→窗口→模块”,查看加载的动态链接库的清单，其地址列即加载地址。dll2 的加载地址为 0x003b0000。然后如上方式获得反汇编：

```
gi = 0x12345678;
003c131e c7 05 38 71 3c 00 78 56 34 12      mov   dword ptr ds:[003c7138h],12345678h
```

此时 gi 的地址是 0x003c7138，与 DLL 文件中的 0x00517138 不同，产生了重定位，且该指令中包含的地址也如黑体所示的新值。dll_relocation.dll 加载到了 0x00500000，而其 gi 的地址是 0x00517138，所以 gi 相对程序头部偏移是 0x00517138-0x00500000=0x00017138。而 dll2 的加载地址为 0x003b0000，那么 dll2 加载后 gi 的地址就应该是：0x003b0000（加载地址）+0x00017138（gi 到头部的偏移）=0x003c7138。这确实是我们跟踪到的情况（见上面的反汇编），由此证明了重定位过程。

接着来分析一个结论（来自学习操作系统时关于动态链接的论断）：

动态链接库不仅可以让所有程序都共享相同的 DLL 文件，从而节省磁盘空间，而且所有使用该 DLL 的进程共享相同代码，从而节省内存空间。虽然不同进程有自己独立的线性地址空间，但将 DLL 的代码段映射到同一段物理内存即可实现内存共享（因代码相同且只读，所以可共享）。

这一论断真的成立吗？

节省硬盘是显而易见的，因为整个硬盘只需要一份副本，要使用的进程加载它即可。节省内存也是显而易见的吗？回头看看刚才的重定位分析，如果不同进程将同一 dll 加载到不同地址，那么重定位后相关指令中包含的地址信息必然不同（如 mov 指令中被写内存的地址）。即使共享内存使不同进程享有同一段物理内存，但代码因重定位产生的差异（如本来是同一条 mov 指令，却因被写内存地址不同而不同）也使得不同进程无法共享代码段。

反过来，要共享代码段，必须加载到同一地址，这样代码才完全一样。这就是为什么在一个 Windows 版本上，其系统 dll（如 kernel32.dll）总是被加载到一个固定地址（见习题 14），被所有进程使用。为了节省空间、共享代码段，系统选择了将它们加载到固定的线性地址。当我们知道以下安全事实时，不要过分责怪 Windows 的不作为（当然也并非完全没有折中方案）：系统 DLL 的固定加载基址被 Windows 下的溢出攻击利用，甚至如果该特性不存在，Windows 下的溢出攻击几乎可以消除。（Windows 7 中终于引入了 dll 随机加载的选项，VC 编译时可指定，参见 http://technet.microsoft.com/zh-cn/magazine/2007.04.vistakernel.aspx。）

现在我们回味一下捕猎知识的过程：全局变量赋值中地址不变的疑惑 → 分析、实践构建自己的重定位机制 → dll 的重定位实验 → dll 共享代码内存的思考 → 对溢出攻击特性的分析和反思。发问和实证真的是我们知识和能力增长的“渔”。我们不应放过任何一个疑惑和实证它的机会。不要疑虑这样会偏离所谓的“学习主题”和“非系统化”，千万条这样的探索线路，不断汇聚、总结、系统化，就能直至梦想的顶峰。

1.9 汇编的学习之路——阅读 RTL

以前笔者总有些疑惑：如何学好汇编。之前的反汇编主要练习的是阅读理解，主要锻炼的是从纷乱的指令中如何找出关键点，显得支离破碎。怎样系统锻炼自己汇编编写能力呢？在 Windows 或 Linux 用户级很难完整地用汇编来编写程序（内核级指令无法使用）。笔者想了一些方法，如先将 C 语言语句写成等价的汇编代码，然后分析它与 C 语言语句反汇编结果的差异。这种方式比较轻松，也能很容易地与“答案”对比，但从兴趣和意义上都难一直持续下去。后来笔者发现了一个不错的方法。

如何获得源代码

如果是 VC，有两个途径：① 在 VS 安装目录...\Program Files\Microsoft Visual Studio 9.0\VC\crt\src 下；② 写一个程序调用该函数，在调用处设置断点，然后运行时在该断点处 step into，即可看到源代码。

如果是 Delphi，则是在其安装目录\source\rtl 下，快速方法是打开工程，输入该函数，按住 Ctrl 键并用鼠标单击该函数，即跳到其源码实现。

如何分析源代码

调试是最好的学习。在 VC 中，按 F11 键能跟踪。在 Delphi 中，选择菜单“Project → Options”，然后选择弹出对话框中的 Compiler 标签，再选择“Debugging” → “use Debug DCUs”。

Delphi 6 的 IDE 有点问题，设置后要强制编译一次，即选择菜单“Project → Build xxx”。按 F7 键，即可跟踪进库函数。之后单步跟踪，并观察相关寄存器和内存的值即可分析。Delphi 中有与 VC 相似甚至更好的反汇编调试界面（栈窗体、代码窗体可任意 goto 到一个希望的地址，标志寄存器窗口、对系统 API 的入口能直接给出函数名等），相关功能可以选择菜单“View → Debug Widows”。

如何分析代码性能

可将两个对比的函数连续调用许多次，调用前记录时间，调用后记录时间，然后对比。

Windows 或 Linux 系统都有高精度的时间函数可调用。在 Intel 平台，精确的方法就是用 Vtune 来分析。

如何学习新的汇编指令

不要专门去背，阅读代码时看到不懂的，去网上查，或者下载一个指令表速查，或找示例代码参考。

首先，用汇编实现开发环境所带的运行时库（Runt Time Library，RTL）中的函数，如 C 语言的 RTL 包括 strlen、strcpy 等。然后，分析系统库实现的这些函数。因为它们调用频繁，所以要求有很高的效率，基本都用汇编撰写。（好好阅读能学到很多高级技巧，能从高手的经典中汲取到很多营养。）再做性能实验，测试自己版本与系统版本的差异，并分析修改。（这一分析过程可以用到指令级分析工具 Vtune，从而在优化方面深入实战。）最后分析不同库实现的异同和好坏，如 VC、C++ Builder、Delphi、GCC。（“优秀的程序员都要阅读好代码”，让我们展开这一愉悦、痛苦又兴奋的旅程吧。）

下面以 StrLen()函数为例来分析，源代码（见 DM1-37）来自 Delphi 的 StrLen，与 C 语言一样都是用空字符判断字符串结束（即遇到字节 0 则认为字符串结束）。

初略一看，我们有以下两处不清楚：① 参数 str，即字串的地址是如何传入的；② repne scasb 的含义。对于①，编写一段代码直接调用它，并分析调用代码反汇编即可。对于②，我们上网查资料即可。

分析 StrLen()的调用代码，我们发现如下反汇编：

```
mov   eax, $0045b720        //$代表十六进制
call    strlen
```

自然的猜测就是通过 EAX 传递了字串地址，0x0045b720 应该是该地址。通过内存窗体输入该地址，看到该地址的内容确实是我们传递的字串。然后查询 repne scasb。scasb 在一段内存中查找 1 字节，要查找的数存在 EAX 中，字符串的地址存在 EDI 中，相当于把查找字节和当前指向的字节比较，如果相同，ZF 标志位为 1。该指令执行后，EDI 寄存器的内容会修改，是加 1 还是减 1 由标志位 DF 决定，0 代表地址增加，1 代表地址减少。repne 是一个重复前缀操作命令，当 ZF=0 且 ECX 的值不等于 0 时，重复其右边的指令。ECX 代表搜索次数，每执行一次 scasb，ECX 减 1。

DM1-37

```
function strlen(const str: pchar): cardinal; assembler;
asm
1       mov   edx, edi
2       mov   edi, eax
3       mov   ecx, 0ffffffffh
4       xor    al, al
5       repne   scasb
6       mov   eax, 0fffffffeh
7       sub    eax, ecx
8       mov   edi, edx
end;
```

下面分析代码。第 2 行设定 EDI，将传入的查询字串的首址（在 EAX 中），赋值给 EDI，以供搜索。由于要修改 EDI 寄存器，所以第 1 行将其赋值给 EDX 保存，在第 8 行返回前恢复。一般情况是保存寄存器值用栈，这里将 EDI 保存在 EDX 中，EDX 被篡改。怎样保证 StrLen()执行后 EDX 恢复原值？这里只能说明 Delphi 内部代码编译方式使得 edx 可自由使用。

第 3 行设定了查询的长度给 ECX，为 4 字节整数的最大值 0xffffffff，字符串不会超过它。

第 4 行将 EAX 寄存器的最低字节 AL 用异或 XOR 设定为 0，表明要查找的字节是 0，这符合空字符结尾字串的定义。

第 5 行开始循环查找，每次会增加 1 字节。其实这里我们是有疑问的，在该函数中并没有设定 DF 为 0 的语句，怎样保证查找的方向是地址增加的方向呢？我们稍后再来分析。

第 6、7 行操控了 EAX。EAX 一般保持返回值，如果 EAX 中是字串的长度，我们需要分析为什么它是长度。ECX 存储的是当前要搜索的次数，循环退出后，循环的次数就是字串长度+1。因为长度不包括最后的 0 字节，而循环是找到最后的 0 字节才退出，那么，循环次数=希望查找的次数（0xffffffff）–剩下的查找次数（在 ecx 中），即 0xffffffff–ecx。因为循环次数=字串长度+1，所以字串长度=0xffffffff–1–ecx=0xfffffffe–ecx。第 6、7 行正好完成该工作。大家观察寄存器的值即可更形象化、清晰地理解这一过程。

现在来看看查找方向的问题，就是 DF 位设置的问题。从调试时的标志寄存器窗口可知，DF 为 0。不妨将 DF 设定为 1 看其效果。首先，分配 4 字节的字符数组，第 1 字节和第 4 字节为 0，第 2 字节、第 3 字节非零。将第 2 字节的地址传入，并故意在调用 StrLen()之前用 std 指令将 DF 设定为 1，返回后，调用 cld 将其设定为 0。（这两个指令也可以在网上查找，如输入“设置 DF 标志位”之类。）从结果看，确实长度是 1，而非 DF 为 0 时的正确结果 2。这是代码作者的疏忽吗？笔者觉得在如此一个生产级的代码中，不应该有这种不设定 DF 的疏忽，也许对库作者而言，约定了一般 DF 为 0 的假定？就可以节约 cld 指令，而将 DF 恢复为 0 的责任推给了那些调用 std 的代码？为了确认这个分析，笔者用“STD”（注意用大小写敏感查找，否则太多无关信息，因为 Delphi 库中其汇编都用大写编写，所以这样好查到有用信息）查找了 StrLen()所在的文件，发现 StrRScan()函数从尾部向前扫描关键字。代码如下：

```
        STD
        DEC     EDI
        MOV     AL, CHR
        REPNE   SCASB
        MOV     EAX, 0
        JNE     @@1
        MOV     EAX, EDI
        INC     EAX
@@1:    CLD
```

黑体的指令调用了 std，完毕后就调用了 cld。这基本证明了笔者的猜测。如果对于自己编写的嵌入汇编代码，遵循其习惯，也就不会出错了。（对任何一个疑惑我们要多猜想，多实证，特别对于库一类的生产级代码，不要轻易下论断否定别人。）

下面来看 VC 的 StrLen()，看看两者的差异。在 VC 的工程中调用 StrLen()，然后按 F11 键跟踪，可看到代码如下：

```
                string  equ [esp+4]
1               mov   ecx, string       ; ecx -> string
2               test    ecx, 3          ; test if string is aligned on 32 bits
```

```
3                       je      short main_loop
    str_misaligned:                         ; simple byte loop until string is aligned
4                       mov   al, byte ptr [ecx]
5                       add   ecx, 1
6                       test  al, al
7                       je    short byte_3
8                       test  ecx, 3
9                       jne   short str_misaligned
10                      add   eax, dword ptr 0    ; 5 byte nop to align label below
11                      align  16                               ; should be redundant
12  main_loop:          mov   eax, dword ptr [ecx]          ; read 4 bytes
13                      mov   edx, 7efefeffh
14                      add    edx, eax
15                      xor   eax, -1
16                      xor   eax, edx
17                      add    ecx, 4
18                      test  eax, 81010100h
19                      je    short main_loop               ; found zero byte in the loop
20                      mov   eax, [ecx-4]
21                      test  al,al                         ; is it byte 0
22                      je    short byte_0
23                      test  ah, ah                        ; is it byte 1
24                      je    short byte_1
25                      test  eax,00ff0000h                 ; is it byte 2
26                      je    short byte_2
27                      test  eax, 0ff000000h               ; is it byte 3
28                      je    short byte_3
29                      jmp   short main_loop ;taken if bits 24-30 are clear and bit 31 is set
30  byte_3:             lea   eax, [ecx - 1]
31                      mov   ecx, string
32                      sub    eax, ecx
33                      ret
34  byte_2:             lea   eax, [ecx - 2]
35                      mov   ecx, string
36                      sub    eax, ecx
37                      ret
38  byte_1:             lea   eax, [ecx - 3]
39                      mov   ecx, string
40                      sub    eax, ecx
41                      ret
42  byte_0:             lea   eax, [ecx - 4]
43                      mov   ecx, string
44                      sub    eax, ecx
45                      ret
```

首先感觉这个代码比 Delphi 版本的多很多指令，但它真的不好吗？我们先要读懂这段代码。下面描述并没有直接给出流程图，而是展示了真正分析过程。读者可看懂后可以自己绘制流程图。

第 1 行似乎不是标准汇编的，可用反汇编查看其真实命令：

```
                mov   ecx, string                       ; ecx -> string
1026fee0        mov   ecx, dword ptr [esp+4]
```

这个 mov 指令中的 string 变量（字串的地址）就是放在 esp+4 的地址中。因此，该指令上有一条 string equ [esp + 4]语句，它定义了 string 的地址就是在 esp+4 中。

第 11 行明显不是汇编指令，还是反汇编，结果如下：

```
                add   eax, dword ptr 0                    ; 5 byte nop to align label below
1026fefd        add   eax, 0
1026ff02        lea   esp, [esp+00000000h]
1026ff09        lea   esp, [esp+00000000h]
                align  16                                 ; should be redundant
```

这里有 3 条指令，第 1 条与 add 指令对应的，后 2 条的作用相同，就是将 esp 的值赋值给 esp。很明显是无意义的。从 align 16 可以猜测，这是为了指令对齐而产生的填充用字节。也可将这些字节全部换成 nop 指令（即占时钟周期但不干活的指令）。这两个小问题解决后，就可分析代码了。

第 1～3 行有点让人晕。第 3 行满足时跳到第 12 行，main_loop 从 ecx 指向的地址读了 4 字节（mov ... dword ptr），也就是说，这条指令是读 4 字节整数的指令。根据对齐原则，4 字节整数必须存放在能被 4 整除的地址上。第 2 行的注释“test if string is aligned on 32 bits”正是测试 string 的地址是否放在能被 4 整除的地址。（看看基础知识的重要性，否则我们无法理解其含义！）因为字符串地址赋值给了 ECX，那么第 2 行就是在测试该地址是否能被 4 整除。test 指令执行 and 操作，若结果为 0，则 ZF 标志位 1。为什么 a and 3 就可检测 a 是否能被 4 整除？能被 4 整除的数二进制形式为 xx..x00，其中“x”表示 0 或 1。xx..x00 and 3 等价于 xx..x00 and 00..011，因此 and 后结果为 0。

这里我们学习到一个小技巧：指令“test　xxx, 3　　je ...”就是来判断是否被 4 整除。

如果无法被 4 整除，则第 4 行取出当前地址指向的字节给 al，第 5 行将当前字串地址后移 1 字节。然后第 6 行判断该字节是否为 0，是，则遇到字串结尾，所以结束，并转向第 30 行的分支。否则，第 8 行再次判断当前字串地址（ECX）能否被 4 整除，不可整除（未对齐），则跳回第 4 行，重复查看当前字节是否为 0。最多 3 次，就会遇到能被 4 整除的地址，从而进入第 12 行。

第 30 行已经找到了 0 字节，将其地址传给 EAX（因为字符串地址 ecx 先加 1 再比较，见第 5 行，第 6 行才比较是否为 0，所以这里要减 1），第 31 行将字符串首址传给 ECX，第 32 行的 eax–ecx 就是字符串的长度。

第 12 行开始的 main_loop 又显得不知所云了。看汇编有时要前后连贯分析，往往能有相互提示作用。第 19 行后就出了循环，那之后在做什么？如果之后的事情清楚了，循环中的工作就可反推出来。

第 20 行将 4 字节整数读入 EAX，第 21、23、25、27 行分别在判断这个整数的最低字节（AL）、次低字节（AH）、次高字节和最高字节是否为 0。总之，就是在找这个整数中哪个字节为 0。这是在 main_loop 跳出后的判断，也就是说，main_loop 在查找一个 4 字节整数中是否包含 0 字节（第 21～27 行），是，则跳出循环。

此时算法的轮廓基本如下：

① 游标 char * p = 字串首部地址。

② 判断 p 是否被 4 整除，是，则进入主循环，即步骤③，否则判断当前字节*p 是否为 0，是，则结束，并计算长度，否则 p++，然后跳转到步骤②。

③ 每次读 4 字节，判断其中是否有 0 字节，有，则进入步骤④，否则读入下 4 字节，继续本步骤。

④ 判断整数中哪个字节为 0，找到后，转向对应分支计算长度并返回，否则转回步骤③。这里有点怪，从步骤③退出就意味着找了 0 字节，怎么还会有第 29 行的 jmp 语句，跳回步骤③呢？现在只有先放放，理出主线。

第 1～11 行为步骤①和②，第 12～19 行为步骤③，第 20～29 行为步骤④，第 30 行至最后对应步骤④中找到 0 字节后的长度计算。

现在有两个问题：

一、步骤③的一个判断语句就能求出 4 字节中有 0 吗？

二、如果能判断出是否有 0 字节，怎么会出现第 29 行的 jmp 指令？

这两个问题其实也是一个问题，我们必须清楚步骤（3）的算法。逆推法有时候能找到关键。第 18 行 test 后，如果值不为 0，就说明整数中有 0 字节（因不为 0 就退出了 main_loop 循环）。我们将第 12～18 行用数学形式表达如下（第 18 行等价于，如果 h 不等于 0，则 a 中可能包含一个 0 字节。以下证明由高辉老师提供）。令

f=a + 7efefeffh	表达式(1)
g=a xor ffffffffh	表达式(2)
b= f xor g	表达式(3)
h= b and 81010100h	表达式(4)

采用格式 a1_a2_a3_a4 表示 a 的从左到右的 4 字节（分别称为第 1～4 字节），f1_f2_f3_f4 表示 f 的从左到右的 4 字节，以此类推。只要证明 h1 到 h4 中任一字节不为 0，即证明了 h 不等于 0。

（1）若 a4 为 0

① 若 a3 的最右位为 1，则 f3 的最右位为 1，g3 的最右位为 0，b3 的最右位为 1，h3 的最右位为 1，即 h3 不等于 0。

② 若 a3 的最右位为 0，则 f3 的最右位为 0，g3 的最右位为 1，b3 的最右位为 1，h3 的最右位为 1，即 h3 不等于 0。

（2）若 a4 不为 0，a3 为 0

因 a3 为 0，因此表达式(1)中第 3 字节的加法没有向第 2 字节进位。

① 若 a2 的最右位为 1，则 f2 的最右位为 1，g2 的最右位为 0，b2 的最右位为 1，h2 的最右位为 1，即 h2 不等于 0。

② 若 a2 的最右位为 0，则 f2 的最右位为 0，g2 的最右位为 1，b2 的最右位为 1，h2 的最右位为 1，即 h2 不等于 0。

（3）若 a4 不为 0，a3 不为 0，a2 为 0

因 a2 为 0，因此表达式(1)中第 2 字节的加法没有向第 1 字节进位。

① 若 a1 的最右位为 1，则 f1 的最右位为 1，g1 的最右位为 0，b1 的最右位为 1，h1 的最右位为 1，即 h1 不等于 0。

② 若 a1 的最右位为 0，则 f1 的最右位为 0，g1 的最右位为 1，b1 的最右位为 1，h1 的最右位为 1，即 h1 不等于 0。

（4）若 a4 不为 0，a3 不为 0，a2 不为 0，a1 为 0

① 因 a4 不为 0，a3 不为 0，a2 不为 0，因此表达式(1)中第二字节的加法向第一字节进位。

② 因 a1 为 0，因此 f1 的最高位为 0，g1 的最高位为 1，b1 的最高位为 1，h1 的最高位为 1，即 h1 不等于 0。

（5）若 a4 不为 0，a3 不为 0，a2 不为 0，a1 不为 0

因 a4 不为 0，a3 不为 0，a2 不为 0，所以表达式(1)中第 4 字节、第 3 字节和第 2 字节的加法都有进位。

① 首先，从表达式(4)可知 h4=0。

② 若 a3 的最右位为 1，则 f3 的最右位为 0，g3 的最右位为 0，b3 的最右位为 0，h3 的最右位为 0，即 h3 等于 0；若 a3 的最右位为 0 则 f3 的最右位为 1，g3 的最右位为 1，b3 的最右位为 0，h3 的最右位为 0，即 h3 等于 0；因此 h3=0。

③ 若 a2 的最右位为 1，则 f2 的最右位为 0，g2 的最右位为 0，b2 的最右位为 0，h2 的最右位为 0，即 h2 等于 0；若 a2 的最右位为 0 则 f2 的最右位为 1，g2 的最右位为 1，b2 的最右位为 0，h2 的最右位为 0，即 h2 等于 0；因此 h2=0。

④ 若 a1 的最高位为 0。

(a) 若 a1 的最右位为 1，则 f1 的最右位为 0，最高位为 1（因表达式(1)的第 2 字节有进位），g1 的最右位为 0，最高位为 1，b1 的最右位为 0，最高位为 0，h1 的最右位为 0，最高位为 0，即 h1 等于 0。

(b) 若 a1 的最右位为 0，则 f1 的最右位为 1（因表达式(1)第 2 字节有进位），最高位为 1（因 a1 不为 0，而其最右位和最高位为 0，因此必然其他 6 位至少一位为 1，与表达式(1)中的 7e 相加，必然产生进位至最高位），g1 的最右位为 1，最高位为 1，b1 的最右位为 0，最高位为 0，h1 的最右位为 0，最高位为 0，即 h1 等于 0。因此，h1=0。

⑤ 若 a1 的最高位为 1，且 a1<>80h。

(a) 若 a1 的最右位为 1，则 f1 的最右位为 0（因表达式(1)第 2 字节有进位），最高位为 0，g1 的最右位为 0，最高位为 0，b1 的最右位为 0，最高位为 0，h1 的最右位为 0，最高位为 0，即 h1 等于 0。

(b) 若 a1 的最右位为 0，则 f1 的最右位为 1（因表达式(1)第 2 字节有进位），最高位为 0，g1 的最右位为 1，最高位为 0，b1 的最右位为 0，最高位为 0，h1 的最右位为 0，最高位为 0，即 h1 等于 0；因此 h1=0。

⑥ 若 a1==80h，则 f1 的最右位为 1（因表达式(1)第 2 字节有进位），最高位为 1，g1 的最右位为 1，最高位为 0，b1 的最右位为 0，最高位为 1，h1 的最右位为 0，最高位为 1，即 h1 不等于 0。这是唯一例外情形：此时即使 h 不等于 0，a 中也没有一个字节为 0。对应 29 行语句。

我们发现，VC 的 StrLen()通过一次读出 4 字节，然后以 5 条加法与 XOR 指令，一次判断的代价降低了内存 I/O 次数及判断次数（否则要一字节一字节地读和判断）。表面看指令多了，但实际其效率应该更高。在这个分析中，我们看到了基础知识的力量，如果不知道对齐，无法理解第 2～11 行代码的意义。而这一判断的重要性在于，如果不对齐，在有的体系下程序就会结束执行。

剩下就是实测这两个算法了。其程序如下（见“code\第 1 章\strlen 性能测试”）：

```
...
#define LOOP_COUNT 20000000
#define STR_BUF_LEN 1024
__declspec(naked) int strlen2(char * str) {
    _asm{
        mov   edx, edi
        //获取 str，因先压栈 str 后压返回地址，所以 esp+4 为参数 str 的地址
        mov   edi, [esp + 4]
        mov   ecx, 0ffffffffh
        xor    al, al
        repne   scasb
        mov   eax, 0fffffffeh
        sub    eax, ecx
        mov   edi, edx
```

```
        ret
    }
}
void main(){
    char str[STR_BUF_LEN];
    ...
    unsigned long start = GetTickCount();
    for (int i = 0; i < LOOP_COUNT; i++) {
        strlen(str);
    }
    unsigned long end = GetTickCount();
    int u1 = end - start;
    start = GetTickCount();     //获取系统当前毫秒数
    _asm   push edx;            //strlen2 使用了 edx，所以保存一下，循环完毕用 pop  edx 恢复
    for (int i = 0; i < LOOP_COUNT; i++) {
        strlen2(str);
    }
    _asm pop edx;
    end = GetTickCount();
    int u2 = end - start;
    //后面打印测试结果…
}
```

注意，对于循环次数 LOOP_COUNT 的确定要合理，我们可先将对 strlen2()的调用改成依然对 strlen()的调用，通过不断增大 LOOP_COUNT，如果两个值比较接近，那么循环次数就确定，否则循环少了没有统计意义。

对于_declspec(naked) int strlen2(char * str)，_declspec(naked)属于 VC 的特殊关键字，表示该函数的左右括号不会生成任何代码，此时 strlen2()的代码就是汇编本身。

照我们分析，当字符串很长时，strlen 应该有优势，因为一次读出 4 字节且其中没有 0 字节，就只有一条判断指令。但字符串太短时应该没有什么优势，如在 4 字节以内，还得一字节一字节地判断。

改变 BUF_LEN 的大小，结果 strelen 执行时间约为 strlen2 的 23%～29%。string 的长度为 1～1023。越长，性能越优越。令人费解的是，即使长度为 1，strlen 耗时也仅为 strlen2 的 29%。我们可以猜测，因为即使是短字符串（长度小于 4），strlen 也只用一条 mov 指令做一次内存读，其他计算就没有与内存打交道。（因为串长小于 4，加上结束符，总长也不会大于 4，一条 mov 指令能读 4 字节即可全部读取。见第 12 行代码 mov eax, dword ptr [ecx]。）而 strlen2 即便当字符串长为 1 时，也要读两次内存：第一次把第 1 字节读入，第二次把结束符 0 读入。虽然在短字符串的时候，从判断逻辑上两个算法应该差不多，但 strlen2 可能因为 CPU 实现为一字节一字节地读内存，所以时间就多了。

我们可实现两个算法，均一字节一字节地判断，然后比较读内存的代价：① 一次读 4 字节到 EAX，然后每字节用 AL、AH、0x00ff0000、0xff000000 进行 test 判断（与 strlen()的区别是，没有一次性判断 4 字节是否存在 0 的优化）；② 一字节一字节地从内存读取，然后判断，相当于①的 4 倍内存读写。

可知，分析和实证很多时候有差距，特别是当我们才涉足一个领域时。只有实证，才能获得更多、学到更多、发现更多。

关于学习 RTL 汇编代码的提示

在授课中，笔者发现很多同学只是简单地看看，写一两句“意思意思”的说明似乎就搞定了。这样是没法深入的。有一个简单的做法来测试你是否理解了，如果你不能丢开别人的源码，写出与它相同的代码。就说明你没有真的理解。

另外一个例子或许更清晰。一次某学生看到 strlen()的注释中有 32 对齐的说明，其学习报告中就说需要对齐。笔者提问：“为什么 32 对齐，是位还是字节？为什么不按 16 对齐？”。他茫然无措。

1.10 程序设置说明

本章的程序，如果用 VC 6.0 编译或者在 Windwos XP 上运行，没有任何问题。如果在 Windwos 7 上，并用 VS 2008 编译，则需要进行设置：右键单击项目，在弹出的快捷菜单中选择“属性”，出现“配置属性”对话框，然后选择“链接器”→“高级”，“随机基址”选择“禁用映像随机化(/DYNAMICBASE:NO)”，“数据执行保护”选择“映像与 DEP 不兼容(/NXCOMPAT:NO)”。

参考资源

本章主要试图通过反汇编去触摸程序的基础，并展示一种学习程序的方式：猜测、实证和构建，揭示了这种方法发现和探索的本质，以及培养提出问题的能力。同时，这一过程采用的底层机制，也是我们在可能的各方面深入的根基。这里再提及一种融汇底层知识的方法：通过编写攻防软件，将操作系统、汇编、编译原理、网络等知识在极为幽微处（如字节层次）贯通起来。

当然，本章的底层知识还远远不够，下面给出一些参考：

① 硬件和软件的结合：《深入理解计算机系统——程序员的观点》、《卓越编程之道》（卷一、卷二）、《黑客反汇编解密》。

② 调试和跟踪：《软件调试》、《加密与解密》、《黑客调试技术揭秘》。

③ 链接（本章中的重定位属于其中的一个知识点）：《Link & Loader》。

④ 高效编码（如 1.9 节中 VC 的 strlen()中判断 4 字节中包含 0 的算法）：《高效程序的奥秘》。

④ 操作系统：《操作系统设计与实现》、《自己动手写操作系统》、《Linux from scratch 从头构建 Linux 的好东西》。

这些资料希望能“玩”起来，而不是“看”下去，否则不如不看。

习 题 1

1．分析多维数组的结构和寻址方式。

2．很多时候，当遇到问题时，大家喜欢上网查。这个习惯在学习成长期并不好，因为可能失去了运用基础知识和技巧自己实验分析问题的机会。一方面，思维能力没有锻炼好；另一方面，底层技巧和知识也没有锻炼到。特别第 1 章涉及的底层知识很琐碎，难以集中系统训练，如果能在遇到问题时主动用它们去分析，自然就训练出来了。如果能仔细领会下面问题的启示，并运用到之后的学习中，将很有益。如果你记不起 C 语言指针加一到底移动了多少字节，怎么办？（不要上网查和问人。）

3．查看 C++中的 bool 类型是如何表示的，长度、真假值是多少。

4．在 1.3.5 节中，第一个局部变量地址在 ebp–8 而不是 ebp–4 中。请分析其原因及 ebp–4 指向

内存的功用。（提示：VS 从 2005 版开始吸取了 Stack Guard 的溢出攻击防护机制，可由用户选择是否提供溢出攻击防护机制。该现象与此机制运用有关，读者须自学相关知识点并分析。）

5．图 1.41 所示“逻辑上”无效的内存实际上有效吗？为什么？

6．自己设计一个参数可变的加法函数，用汇编实现，且用汇编展示其不同参数下的调用。

7. 在 Delphi 中实验 Pascal 方式的调用惯例。提示：其调用 Pascal 方式的调用惯例关键字是 pascal。

8．分析 GCC 的 switch 语句的反汇编，得出其算法，并与 VC 的算法比较。

9．查看 1.6.3 节描述的“稀疏”switch 类型，当 i_n–i_1>255 时，VC 所选择的 switch 算法。

10．重定位除了修改指令外，还可能修改什么部分？举例。

11．猜测加上实证，往往能发现许多出乎意料的东西。在 1.1.1 节对全局变量 gi 赋值语句“gi = 12;”的对应指令“mov [地址], 12”进行实证时，发现其机器码由三部分组成：c7 05、地址、被赋值的数。其中，c7 05 是 mov 的操作码。而这一点推断并未证明。请用内存窗体将 05 改成 15、25、35、45，在反汇编窗口中查看结果，并分析其原因。

12．不用嵌入汇编，用一个比较判断完成 1 <= i <= 5 的比较（不是两个判断 0<=i && i <=5），其中 i 的类型为 int。

13．用 PElord 验证 VC 6.0 编译的执行程序没有重定位表（Release 版），而 DLL 有重定位表。再查看 VS 2008 默认生成的执行程序是否有重定位表，分析其原因。

14. 用 VS 2008 查看进程的系统 DLL（如 kernel32.dll）的实际加载基址总是被加载到同一位置。例如，打开记事本程序，然后选择 VS 2008 的菜单“工具”→“附加到进程”，从弹出窗体的进程列表中选择 notepad，并确定。然后如 1.8.3 节查看动态链接加载基址那样查看 kernel32 基址。

15．反汇编分析如下代码：

```
struct SS {
    int a;
    int b
};
void put(struct SS s) {
    s.a = 1;
    s.b = 2;
}
void main(){
    struct SS s1;
    put(s1);
}
```

如何传递结构体变量 s1？分析与传递结构体变量指针的差异。

16．C 语言中从右往左压栈，原因在于支持变参函数。请分析原理。（提示：可分析 printf()，从如何确定出变参个数入手思考该问题。另外，变参相关函数 va_start、va_arg）。

17．学习 Windows 下 PE 格式中重定位表项相关内容，然后设计一个实验，验证重定位表的效果和功能。（提示：可用 PELord 和 UltraEdit。）

第2章

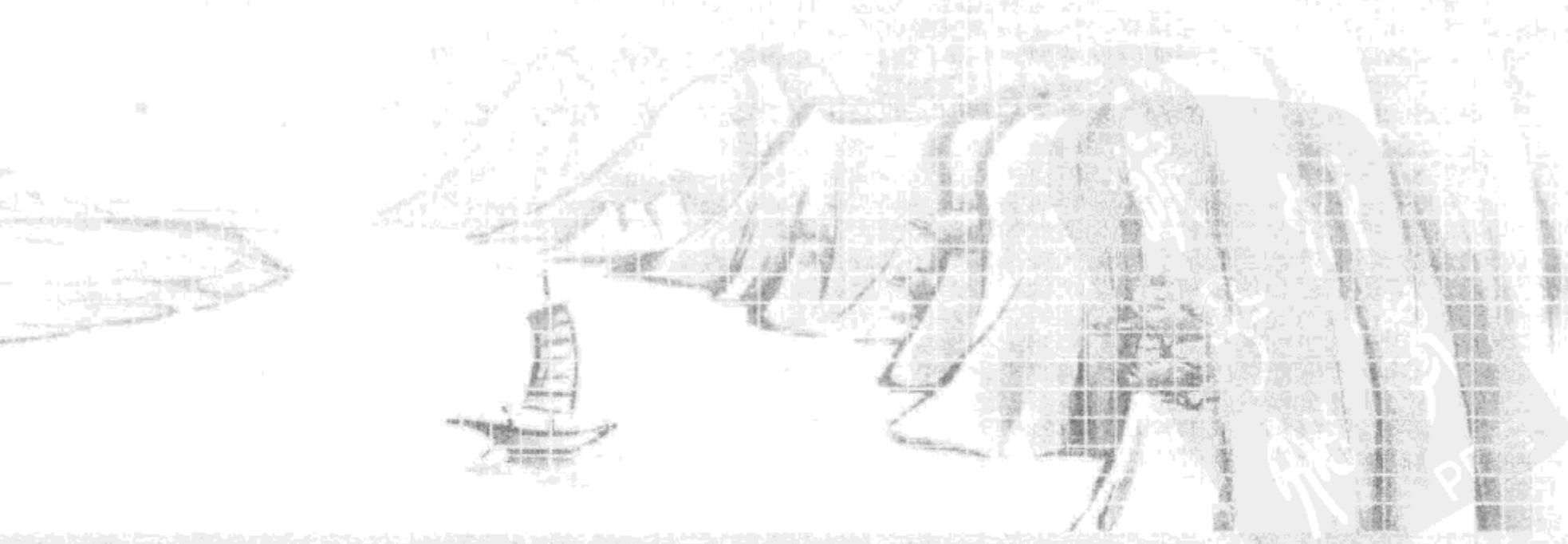
庖丁解“码”：底层的力量与乐趣

需准备的知识 »

- 2.1节需要1.1.2节、1.3.1节和1.3.3节的知识。
- 2.2节需要1.4.1节和1.3.5节的知识。
- 2.3节需要1.3.3节～1.3.5节的知识。
- 2.4节需要1.3.8节～1.3.9节的知识。
- 2.5节需要第1章的反汇编调试技巧。
- 2.6节需要1.3.3节和1.3.5节的知识。
- 2.7节～2.9节需要第1章的反汇编调试技巧。

为你提供的能力 »

- 进一步从问题解决中巩固底层程序观。
- 展示了分析底层问题的方法和技巧。
- 充分理解猜测—实证—构建方法。



在第 1 章漫长的旅途后，大家也许会问：“这些底层的东西到底有什么用？”笔者自己也觉得很不好回答，常答曰“相信我，这些很有用”。回忆起来，它们常常在关键处生效，但又显得零敲碎打，不成体系。这些不成体系既让人难以把握，也难以表达。借着成书的机会，笔者整理了思绪，选择了一些曾经历的东西，形成了本章内容。就是想在大家迷惑之时，展示底层威力，权当是一针强心剂吧。本章顺序值得商榷，因为部分内容涉及比较高级的主题，放在本章似乎“越级了”。如果有些困难，可将来返回再看。另外，一些运用底层分析的例子融汇到了后面章节，未单独列出。

2.1 解密之 hello world

先来个轻松的热身吧，它能用上我们之前学习的许多知识，并且最后的工作量只需要修改一两字节。本节将给出一个简单的需要输入密码的程序，然后对其执行程序解密。解密就是对执行程序动手术，自然我们应该对手术对象“生理构造”了如指掌。前面的基础知识学习，已初具认知，现在就来个牛刀小试。

DM2-1

```
1   void main()
2   {
3       char buf[128];
4       memset(buf, 0, 128);
5       gets(buf);
6       if (strcmp(buf, "itspassword") != 0)
7       {
8           printf("wrong password, try again\n");
9           return;
10      }
11      printf("you are right\n");
12  }
```

一般认为，要进行类似口令判断，只需要一个简单的 if 判断即可。用户似乎根本无法跨过这道坎。DM2-1 展示了这一逻辑，其中第 6 行判断用户输入到 buf 的口令是否与预先定义的 itspassword 相同，如果不同，则转入第 7～9 行打印失败并退出。完整工程见“code\第 2 章\password”。

我们来演示最基本的解密过程。首先运行该程序，随便输入一个字符串，会看到“wrong password, try again”的提示。那么，自然的联想就是，在 printf 语句的附近应该存在一条比较语句，判断口令正误（即第 6 行）。如果找到了打印错误提示的语句，那么在附近就能查找到判断语句，然后分析它，就能解密了。**（有没有办法快速找到打印语句呢？）**

对于 printf()函数，打印字符串是传入其地址。（这是空间和时间都很高效的做法。）这个被打印的字符串常量放在哪里？由 1.8.2 节知道，程序的数据段存放了全局变量或字符串常量等数据。反汇编，分析执行程序，从为 printf()传参的指令即“push　字符串地址”中可以获得字符串地址。由 1.8.2 节的“重定位机制”知道，从指令中包含的字符串地址能很容易计算出该字符串在执行文件中离首部偏移量，同时可获取一个程序中所有引用了该字符串的指令（因为它们压栈传递的地址都是同一个地址）。

手动分析太麻烦，有没有工具能做呢？有，就是反汇编之王者 IDA。IDA 可以列出程序数据段所有字符串，指定某字符串，可给出引用该字符串的所有指令。我们只需查看这些指令，即可找到目标。

将编译好的实验程序复制到“code\第 2 章\password\ida_data”中，然后打开 IDA pro，见图 2.1。单击 New 按钮，然后在图 2.2 所示对话框中选择 PE Executable，在弹出的对话框中选择要解密的文件“code\第 2 章\password\ida_data \test.exe”。然后一直单击 Next，直到完成。出现如图 2.3 所示的选择 Strings 页面。

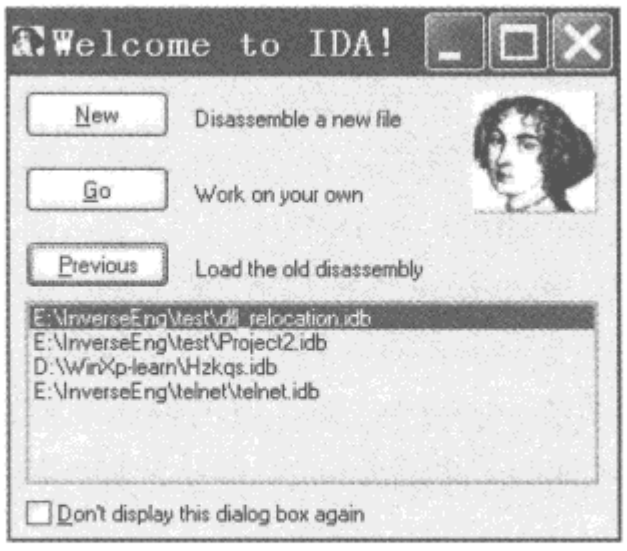


图 2.1

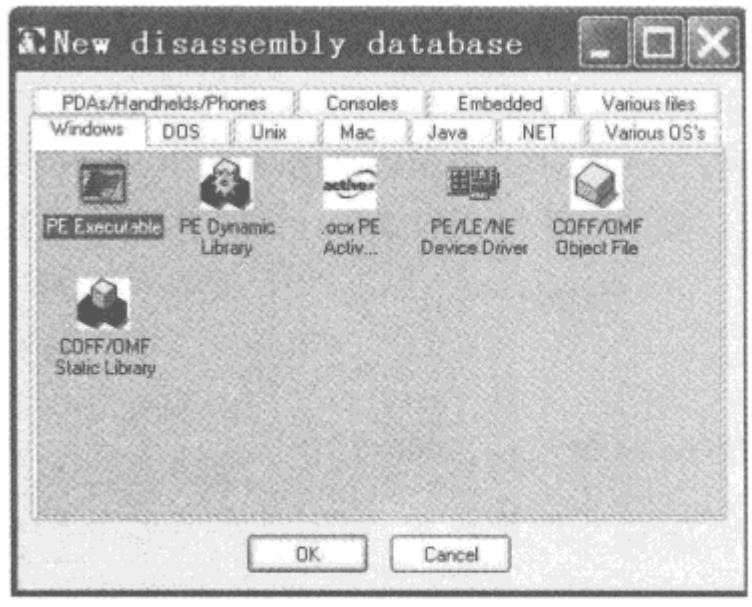


图 2.2

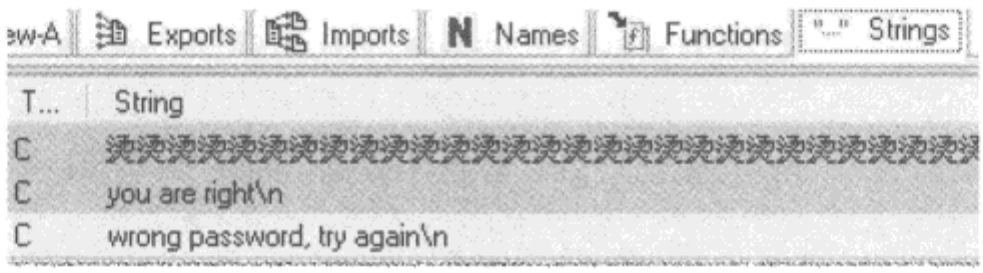


图 2.3

图 2.3 中的“wrong password, try again”正是我们想寻找的。双击该条目，将转跳到字符串数据段，然后按 Ctrl+X 键，将显示有哪些指令引用了该字符串，见图 2.4。图 2.4 中显示只有一条指令引用该字符串，双击它，将转到该指令，见 DM2-2 中的第 3 行。

图 2.4

DM2-2

```
1    text:0040105e              test  eax, eax
2    text:00401060              jz    short loc_401071
3    text:00401062              push  offset awrongpasswordt          ; "wrong password, try again\n"
4    text:00401067              call  _printf
5    text:0040106c              add   esp, 4
6    text:0040106f              jmp   short loc_40107e
7    text:00401071 ;哪哪哪
```

那么修改就不会影响到其他部分。从 DM2-2 第 3 行和第 2 行地址(左边的 16 进制数)的差 00401062 – 00401060=2 可知，jz 指令由 2 字节组成。由 1.3.1 节知，call 指令由操作码和偏移量组成，自然联想到与它相似的 jz 指令应该也由操作码和偏移量组成。既然 jz 为 2 字节，那么第 2 字节应该是偏移量。我们来看 jz 的机器码，证明该猜想。怎样获取 jz 的机器码？我们在 C 语言中嵌入汇编，然后在反汇编窗体中查看机器码即可，见 DM2-3。

DM2-3

```
1   __asm{
2                       jz      lb1
3   0041275e 74 01      je      00412761
4   nop;
5   00412760 90         nop
6                 lb1:
7   nop;
8   00412761 90         nop
9    }
```

首先猜测 jz 的偏移量。第 2 行 jz 跳转到第 7 行 nop，偏移量计算如 call 指令，从 jz 的下条指令即第 4 行计算。它与跳转目标第 7 行之间存在一条 1 字节的 nop 指令，所以 jz 的偏移量应为 1。

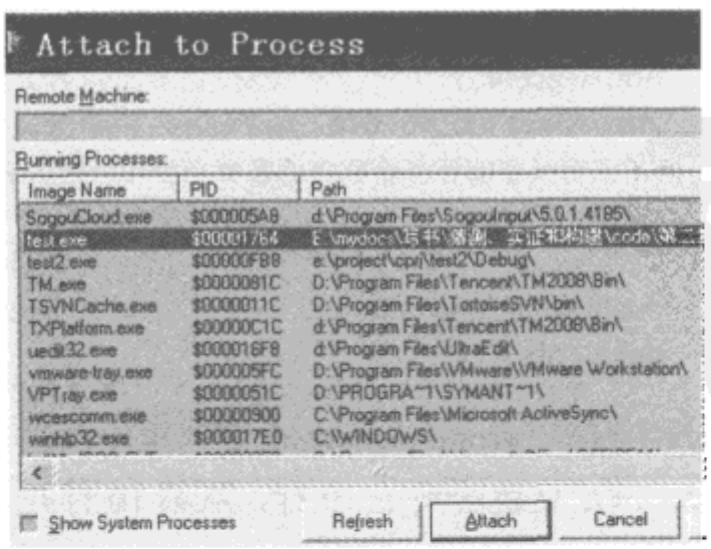
第 3 行地址后的 74 01 即 jz 的机器码，74 为 jz 的操作码，01 为偏移量。符合猜测。

现在需要将 DM2-3 中的 jz 改为 jmp，用相同方法获取 jmp 的机器码，为 eb 01。其中，eb 为 jmp 的操作码，01 为偏移量。

至此，只要能在执行程序中查找到 DM2-2 中第 2 行的机器码，即可将 jz 的操作码 74 修改为 jmp 的操作码 eb。对于偏移量，因为 jz 和 jmp 要转跳的位置一致，并且都用 1 字节代表偏移量，所以不用改。为了查找 jz 指令，我们需要获得 jz 的全部机器码，即它的偏移量。从 DM2-2 来看，第 2 行要转跳到地址 0x00401071，起始地址要从 jz 的下条指令即第 3 行的地址 0x00401062 计算。偏移量=0x00401071 – 0x00401062=0x0f。那么，第 2 行 jz 指令的机器码应为 74 0f。

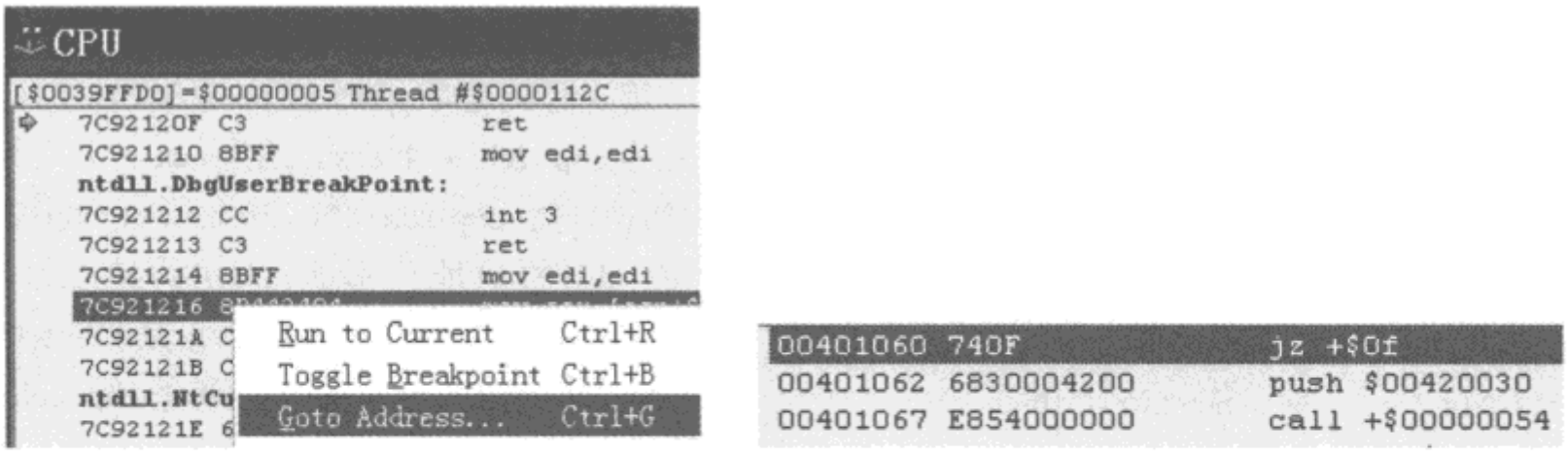
有了 jz 的机器指令，怎样在执行程序中查找这 2 字节？**(用 UltraEdit 即可，方法见 1.8.2 节的图 1.68。)** 用 UltraEdit 打开执行程序后，选择二进制搜索，输入“74 0f”。但 74 0f 太多，因此需要输入更多的字节来精确定位。如果能在调试方式下找到 jz 指令，然后查看其后面指令的机器码即可。可是我们怎样知道 jz 指令的地址呢？DM2-2 的第 2 行左边给出了指令地址 0x00401060，这是否是程序运行时 jz 指令的地址？这里要用第 1 章学习的重定位机制分析。IDA 之所以能在该指令旁边给出地址值，应该是用“程序希望加载基址+指令到文件头的偏移量”计算的。执行程序总能加载到希望地址，也就是说，运行时 jz 指令的地址应该依然是 0x00401060。

现在的工作就是调试这个执行程序。这里没有用专门的调试工具，笔者采用了 Delphi 的调试器。没有源代码不知道怎样调试吗？没关系，先运行 test.exe，然后选择 Delphi 菜单中的“Run|Attach Process”。在图 2.5(a)所示的界面中双击 test.exe 条目，即可将调试器附着在 test 进程上，调试器能跟踪查看整个 test 进程了。



(a)

图 2.5



(b) (c)

图 2.5（续）

然后在 CPU 窗口界面中（相当于 VC 的反汇编窗口）单击右键，在弹出的快捷菜单中选择 Goto Address，见图 2.5(b)。输入 IDA 中 jz 的地址$401060，回车（Delphi 中的十六进制数前用“$”表示），窗体将显示该地址的反汇编，见图 2.5(c)。jz 指令的机器码果然如前分析，是 74 0F。

从图 2.5(c)可见，jz 之后的机器码是 68 30 00 42 00，加上 jz 本身的 74 0f，我们可在 UltraEdit 中查看字节 74 0f 68 30 00 42 00，这次只查找到了一处。因此只需将该处的 74 修改为 eb，即将 jz 变成了 jmp。（**在 UltraEdit 中将光标放在 74 上，然后输入 eb，再保存。**）1 字节搞定。

2.2 奇怪的死循环

我们调试程序的时候经常会遇到一些奇怪的行为：完全与源代码所指示的不同。在不会反汇编分析的时候，听到最多的一个说法就是“系统疯了，不是我的错，看看源代码，我没有那样写”。通过反汇编，我们能认清自己的错误。

DM2-4

```
void main(){
    int i;
    int a[10];
    for (i = 0; i <= 10; ++i){
        a[i] = 0;
        printf("%d\n", i);
    }
}
```

本节的例子来自某年某科研院所招收研究生的题目。代码见 DM2-4，请分析其结果。

该代码是循环 11 次，打印 i 的值。但仔细看，其中有问题。数组 a 是 10 个元素，而循环中对 a[i]赋值，当 i 为 10 时，a[10]=0 将越界。这明显是错误的。但这个错误会造成什么结果？根据第 1 章的知识，局部变量在栈上分配，数组 a 自然也在栈上分配，对 a[10]赋值就是向 a 的首地址偏移 40 字节的内存赋值。这个赋值会产生什么结果呢？我们知道，参数、局部变量、函数返回地址等均放在栈上。被越界赋值的内存区到底是谁的“奶酪”？来画栈区映像，见图 2.6。局部变量 i 在哪里呢？不是在 a[0]的下方就是在 a[9]的上方。如果在上方，应该紧邻 a[9]，而这个位置就是逻辑上 a[10]的位置。代码每次循环都会将 a[i]设定为 0，如果 i 的地址就是 a[10]的地址，岂非每次循环到 11 次时都将 i 赋值为 0？那就是死循环。现在需要用代码来验证。

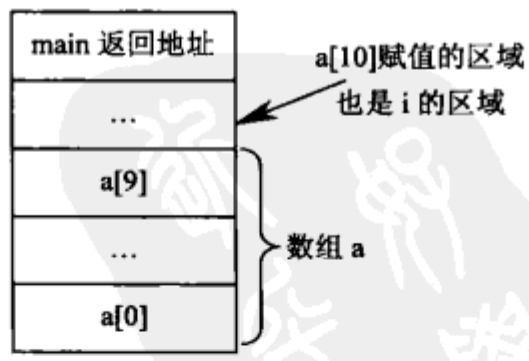


图 2.6

在 VC 6.0 下直接选择 Debug 版本，完整代码见“code\第 2 章\InfiniteLoop”。运行结果果为死循环，意味着 i 的地址应该正好在 a[9]的上方紧邻处。反汇编代码见 DM2-5。

DM2-5

```
1   a[i] = 0;
2   00401040        mov   ecx, dword ptr [ebp-4]
3   00401043        mov   dword ptr [ebp+ecx*4-2Ch], 0
4   printf("%d\n", i);
5   0040104B        mov   edx, dword ptr [ebp-4]
6   0040104E        push  edx
7   0040104F        push  offset string "%d\n" (0042001c)
8   00401054        call  printf (00401090)
9 }
```

第 2 行将 i 赋值给 ecx，i 的地址为 ebp–4。

第 3 行就是对 a[i]赋值 0。a[i]地址的计算方式为 ebp–2ch + 4*ecx。因为 ecx 为 i，所以 a[i]的地址为 ebp-2ch+4*i。由此可知 ebp–2ch 就是 a[0]的地址。那么 a[10]的地址就是 ebp–2ch+ 4*0ah = ebp–4，ebp–4 正是 i 的地址。在 Watch 中验证，a[10]的地址就是 ebp–2ch + 4*10，也是 i 的地址，见图 2.7。

Name	Value
(void *)(ebp - 2ch + 4 * 10)	0x0012ff7c
⊞ &a[10]	0x0012ff7c
⊞ &i	0x0012ff7c

图 2.7

这就是死循环的原因，i 和 a[10]的地址相同，每次循环执行到 a[10]=0 时就将 i 归零。这也是那个考试的答案。（但真的是这样吗？）我们选择 Release 版本再看看，循环 11 次，程序就结束了。在 VC 6.0 中，因为 Release 版本默认不生成符号表，所以无法进行源代码级调试。因此需要改变项目属性，生成符号表，见图 2.8，选择 Program Database，并选中 Generate debug info。反汇编结果见 DM2-6。

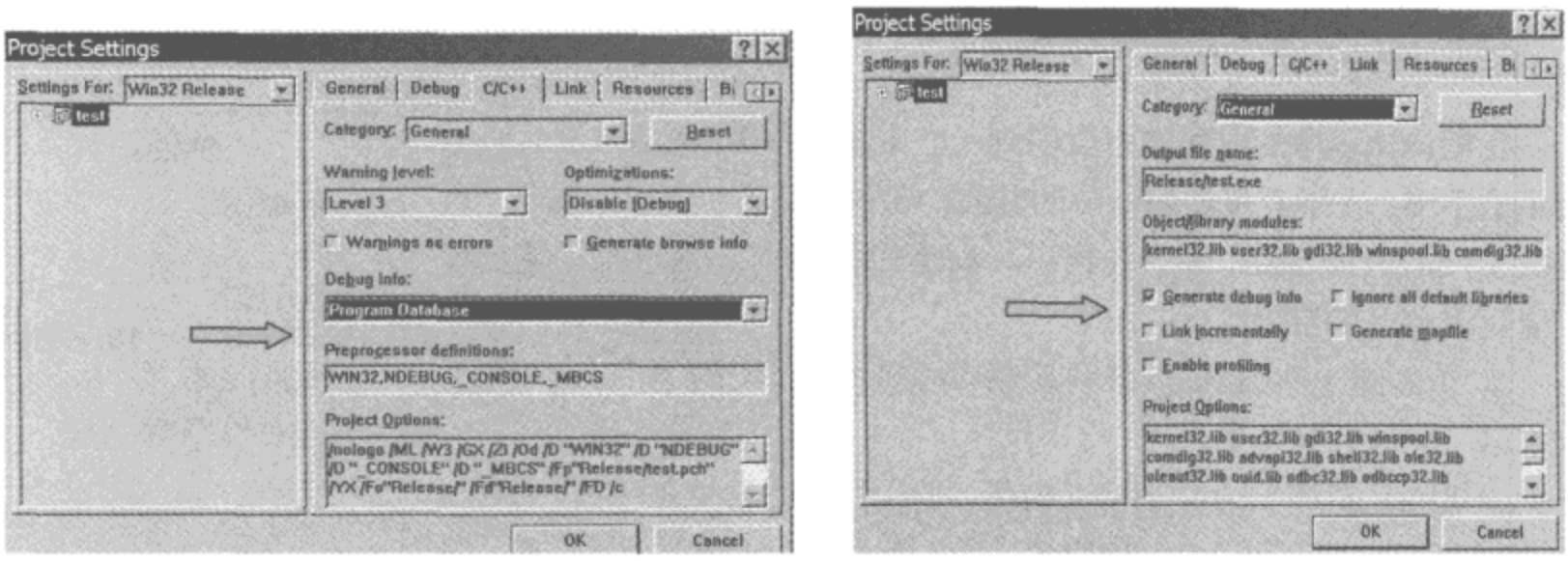


图 2.8

第 1 行的 esi 就是局部变量 i，因为从第 3 行 call printf 倒推，i 为最后一个参数，所以第一个被 push 入栈。从第 5 行也可见 inc esi 就是实现了循环中++i 的效果。这个代码比 DM2-5 所示的调试版代码简单多了。调试版在栈上为 i 和 a 都分配了内存，每次从内存获取 i 的值，加 1 后再写回该内存。a[i]=0 也确实向相应的内存赋值，但发行版优化了。（没有意义的代码不要！）没有必要给 i 分配内存，它就是个计数器，放 ESI 中即可。没有必要给数组 a 分配内存，这个局部变量没有传递给其他函数或做他用，a[i]=0 根本没有意义，所以既没有为数组 a 分配内存，也没有生成 a[i]=0 的代码。那么，

DM2-6

```
    for (i = 0; i <= 10; ++i) {
        a[i] = 0;
        printf("%d\n", i);
1   00401013        push  esi
2   00401014        push  offset string "%d\n" (00411adc)
3   00401019        call  _printf (0040b660)
4   0040101E        add   esp,8
5   00401021        inc   esi
6   00401022        cmp   esi,0Ah
7   00401025        jle   main+3 (00401013)
```

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

自然不会产生死循环。但是调试版不会随意改掉、kill 掉你的代码。忠实原著才好调试。这也是为什么有时候调试版对但发行版错的原因。对于错的代码，发行版和调试版不等价。

不同的编译策略，同样的源代码差异可以非常大。如果用 VS 2008 编译，则死循环在调试版下也不会发生。通过反汇编跟踪，a[10]地址比 i 地址低。a[10]=0 的越界访问根本覆盖不到 i，而且 a 与 i 之间有许多似乎无用的填充字节。

通过调试分析，我们知道了这道考题正确的一面，也看到很多状况下是不成立的。（**动手实证才能获得真正的知识与智慧。**）

VS 2008 调试版下，如果希望代码接近 VC 6.0（没有加入许多运行时检查），则激活程序的项目属性窗体，选择“项目属性 | 配置属性 |C++| 代码生成 | 基本运行时检查”，设定为默认值。

2.3 我们都犯过的错——指针的指针

下面的例子来自笔者所做项目：一次是带本科生做毕业设计，用 C 语言编写 Windows 内核防火墙；一次是带硕士用 C++做一个远程控制的项目。当时将防火墙规则定义为一个结构体，添加规则时，本科生定义了一个函数，将结构体指针传递给该函数，函数中分配一个新的规则，设定给传入的指针。当时由于错误发生在内核中，直接引起蓝屏死机。两年后，另外一个硕士向我讲述其问题时，笔者感觉似曾相识：一个函数内，new 新对象设定给函数外的一个待分配对象的指针。

下面将这个问题简化为 DM2-7，完整代码见“code\第 2 章\ptrptr”。该代码逻辑似乎很清晰，在第 7 行定义了整数指针 i，然后第 8 行传递给函数 allocateInt。第 2 行为指针分配了 4 字节内存，并在第 3 行将该内存的值赋为 3。从函数返回后，第 9 行打印 i 指向内存的值。该值是 3 吗？面对问题我们要先分析，再实证。用大脑虚拟机刻画一下函数调用栈，看看每条相关语句操作的结果。（**这个基本方法我们将不断使用，需要大家形象思维的时刻到了。**）栈映像见图 2.9。

DM2-7

```
1   void allocateInt(int * i){
2       i = (int *) malloc(sizeof(int));
3       *i = 3;
4    }
5
6   void main(){
7       int * i;
8       allocateInt(i);
9       printf("*i = %d", *i);
10  }
```

图 2.9(a)、(b)、(c)分别表示调用 allocateInt 之前、执行中和之后的过程，以及相关语句对内存的修改。关键问题是 allocateInt 应该修改栈最上面一槽内存，即 main()函数的局部变量 i，并将其修改为新分配内存的地址，即 0x338800。从图(b)看，allocateInt 修改的是其参数 i 所占的内存，即栈中间槽位的内存被修改为 0x338800，而非 main()的局部变量 i。因此在 main()函数中打印时，i 作为指针，其内容依然是没有初始化的，会打印其初值 10 作为地址指向的内存中的值，在图(c)中的值未知，用问号表示。（**其实，甚至这个内存都可能不存在。**）

调试，来看以上分析是否正确，反汇编代码见 DM2-8。以上代码如在 VS 2008 下运行，需要将工程中与空指针检查相关的选项取消，否则代码会因为生成了一段空指针检查而无法执行 allocateInt。（选择“项目属性|配置属性 | C++ | 代码生成 | 基本运行时检查”，设定为默认值。）

第 8 行调用了 malloc 函数，其返回的新内存地址保存在 eax 中。

第 12 行将 eax 即新内存地址赋给 ebp+8 指向的内存。从第 6、7 行可知，这是典型的函数栈帧，参数 i 在第 6 行压栈的 ebp 的上方，其地址正好是 ebp+8（下方为返回地址和 ebp，共 8 字节，见

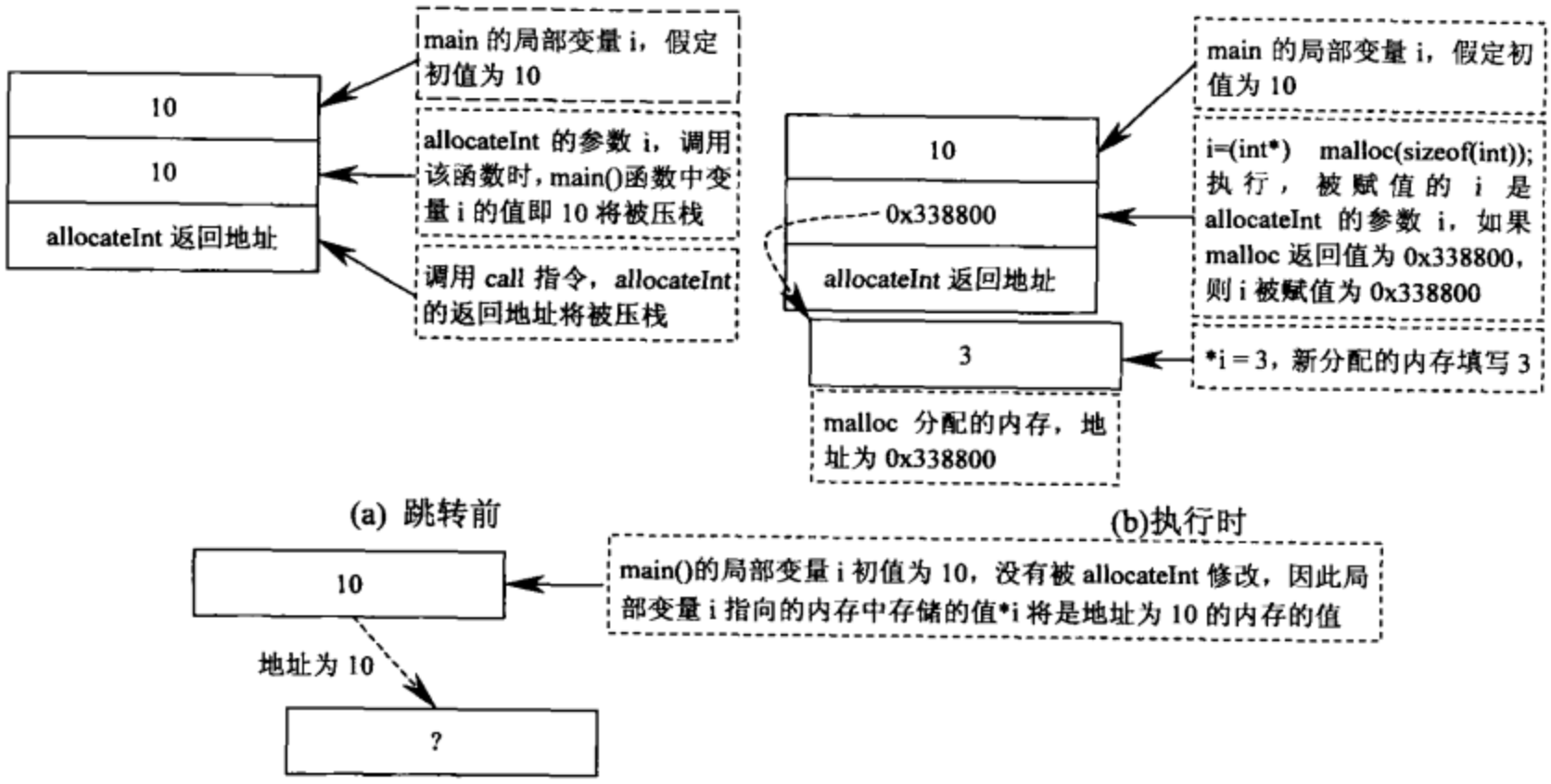


图 2.9

图 1.31 和 1.3.4 节)。也就是说，新分配内存的地址赋给了参数 i，而非 main()函数局部变量 i。从第 2～3 行看，将 main()的局部变量存储的值（地址为 ebp–4 中的初始值）压栈了。其实我们希望第 12 行修改 ebp–4 指向内存的值。

DM2-8

```
1   allocateint(i);
2       mov   eax, dword ptr [ebp-4]
3       push   eax
4       call    00411050
        ...
5   void allocateint(int * i){
6       push   ebp
7       mov   ebp, esp
        ...
8   i = (int *) malloc(sizeof(int));
9       push   4
10      call    dword ptr ds:[0041723ch]
11      add    esp, 4
12      mov   dword ptr [ebp+8], eax
```

DM2-9

```
1   void allocateInt(int ** i) {
2       int * pi = (int *) malloc(sizeof(int));
3       *pi = 3;
4       *i = pi;
5   }
6   void main() {
7       int * i;
8       allocateInt(&i);
9       printf("*i = %d", *i);
10  }
```

请自己调试，单步跟踪用监视窗体、内存窗体和寄存器窗体，观察相关语句执行的影响，进一步证实以上分析。

通过栈映像的头脑虚拟，我们能够发现错误，但是如果这样才能得到正确做法（目前还没有找出正确解）也太麻烦了。有时候复杂的东西要从最基本原理出发分析反而非常简单。我们的初衷是通过 allocateInt 为 main()函数的指针变量 i 分配一块内存，就是将新内存地址写入指针变量 i。最基本的概念是“要修改一块内存，就需要拿到其地址”。变量 i 是一块内存，要在 allocateInt 中修改它，就要传给该函数 i 的地址吧，即

```
allocateInt(&i);
```

这样就能决定该函数的参数类型了：首先 i 的类型为 Type，传递它的地址就是 Type *。因此函数的原型就是 void allocateInt(Type * pi)。

对 main()函数而言，其局部变量 i 的 Type 是 int *，所以用 int *替换 Type，则函数原型是

```
void allocateInt(int ** pi);
```

int**，指针的指针，似乎也不太难以理解。有了该函数原型，那么函数的实现就水到渠成，见 DM2-9。第 4 行将新内存地址值 pi 赋给了 main()函数的局部变量 i。

2.4 互通的障碍（跨语种调用）

本节探讨程序中比较有价值的一个主题，跨语种编程。比如，Visual C++处理界面不是强项，在工程中经常用 Visual Basic 开发界面，用 C++做逻辑层。而 Matlab 进行数学计算时很强大，用 C++编程时需要调用它的数学函数库。动态链接库可让不同的语种之间交互，因为它本质上将为使用者提供一种从函数名字获取函数地址的方法。获得函数地址，我们就能调用它。1.8.3 节介绍了动态链接库的基本使用接口。

下面用 Delphi 编写一个动态链接库，用 Visual C++来调用。Delphi 编写动态链接库比较简单，首先选择菜单“File | New | Other...”，图 2.10 中选择 Dll Wizard，单击 OK，建立一个 DLL 的工程。选择菜单“File | New | Unit”，生成新的单元文件 unit1.pas。添加 Add()函数，代码见 DM2-10。

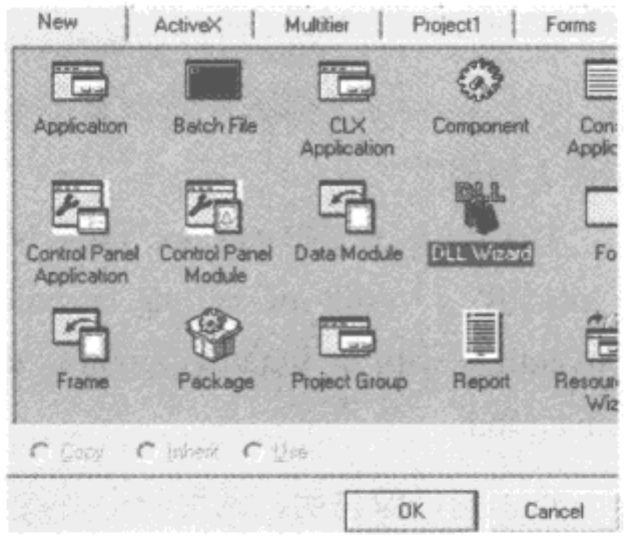


图 2.10

Delphi 有返回值的函数以关键字 function 开头，后跟函数名，参数表以圆括号标记。返回类型在参数表的右括号后加“: 类型”，返回类型 Integer 相当于 C 语言中的 int。函数体以 begin 开始、end 结束。返回值写入关键字 Result 中。与 C 语言不同，对 Result 赋值并不导致函数返回（C 语言执行“return 返回值;”将导致函数返回），程序将顺序执行。DM2-10 中的 Add 函数有 4 个整数参数。

DM2-10

```
function Add(I1: Integer; I2: Integer; I3: Integer; I4: Integer): Integer;
begin
  Result := I1 + I2 + I3 + I4;
end;
```

选择菜单“File | Save All”，在弹出的对话框中选择目录保存本工程，如 DelphiDll，包含 Add()函数的单元文件为 unit1.pas。在 DelphiDll 文件中加入如下代码，将输出 Add()函数，其他程序就可调用该函数了：

```
exports
  Add;
```

再次选择菜单“File | Save All”，保存改动。按 Ctrl+F9 键，即可在源文件目录下生成 DelphiDll.dll 文件。整个工程见“code\第 2 章\cross_language\DelphiDll”。

下面构建 Visual C++工程，调用该动态库，主程序见 DM2-11，工程见“code\第 2 章\cross_language\vcmainprogram”。对应 DLL 中的 Add()函数，在第 1 行定义相应的函数指针，接收 4 个 int 参数（与 Delphi 的 Integer 参数对应），返回类型为 int。第 3 行加载 DLL，第 8 行获取 Add()函数的指针，第 13 行调用 Add()函数。

编译该程序，将 DelphiDll.dll 复制到“code\第 2 章\cross_language\vcmainprogram\Debug”下，

DM2-11

```
1   typedef int (*AddPtr) (int, int, int, int);
2   void main(){
3       HMODULE h = LoadLibraryA("DelphiDll.dll");
4       if (NULL == h){
5           printf("cannot load\n");
6           return;
7       }
8       AddPtr fp = (AddPtr) GetProcAddress(h, "Add");
9       if (NULL == fp){
10          printf("cannot get function sub\n");
11          return;
12      }
13      int i = fp(1,2,3, 4);
14      printf("3 + 2 = %d\n", i);
    }
```

然后执行该程序。运行却产生错误，见图 2.11。左侧的箭头指向的正是 DM2-11 的第 13 行，对 Add() 函数的调用。我们需要跟踪，在第 13 行设置断点，然后反汇编单步跟踪，目的是找到出错的指令，反汇编代码见 DM2-12。

第 9 行中抛出了错误。第 2～7 行是调用 Add() 函数的指令，第 2～5 行压栈参数，第 6 行调用 Add() 函数，fp 的地址是 ebp-14h，call 指令转向其中存放的函数地址。第 7 行平衡堆栈，4 个整数参数共 16 字节，压栈使得 esp 减小了 16 字节，所以平衡时使用 add esp, 10h 指令，使栈还原，也就是使得 esp 还原到执行第 2 行指令之前的值。为什么直到第 9 行调用了不知为何物的函数后报错了呢?报的什么错呢？

DM2-12

```
int i = fp(1, 2, 3, 4);
1     mov   esi, esp
2     push  4
3     push  3
4     push  2
5     push  1
6     call  dword ptr [ebp-14h]
7     add   esp, 10h
8     cmp   esi, esp
9     call  0041113b
}
```

The value of ESP was not properly saved across a function call. This is usually a result of calling a function declared with one calling convention with a function pointer declared with a different calling convention.

第 1 句好像说明，调用一个函数后，ESP 的值没有正确保存。但第 7 行的作用正是恢复 ESP。（**第 1 行和第 8、9 行似乎与调用函数一点关系都没有。**）第 1 行将 ESP 的值保持到 ESI。在函数返回并平衡栈后，第 8 行比较当前的 ESP 和调用函数前的 ESP 值（保存在 ESI 中）是否相同。可以猜测，第 9 行调用的函数中应该判断第 8 行比较的结果，如果不相同（栈不平衡），就要报错。

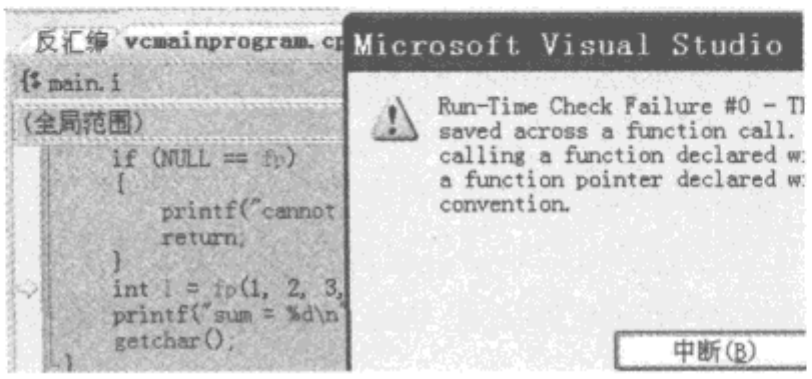


图 2.11

跟踪进第 9 行调用的函数，我们发现如下代码：

```
                _RTC_CheckEsp:
004114c0            jne   004114c3
004114c2            ret
                esperror:
004114c3            push  ebp
004114c4            mov   ebp, esp
```

很明显，第一条指令判断如果调用函数前的 esp 不等于调用后的 esp，就执行 004114c3 分支，也就是报错的分支。通过跟踪我们发现，确实走了该分支，而不是从 004114c2 分支返回。严重了，

调用前后，esp 不同。为了谨慎，我们在代码执行 DM2-12 的第 2 行之前在寄存器窗体查看 esp，在第 8 行处再查看其值，之前是 0x0012fe78，之后是 0x0012fe7c。也就是说，执行函数调用后 esp 向上多移动了 4 字节。而且，加法计算的结果应该在 EAX 中，此时其值是 0xf968e24e，根本不是希望的 10。也就是说调用 dll 的函数后，不仅栈平衡被破坏了，而且计算逻辑也错误。到底出了什么问题？看看上面那段英文错误信息，第 2 句似乎在说函数指针申明的调用惯例和它指向函数的实际调用惯例不同。（调用惯例是我们在 1.3.9 节中学习过的知识。）仔细想想调用方和被调用方如何交互沟通应该能发现问题所在。其实，两方代码要交互必须解决以下几点（在 1.3 节中分析过）：

① 如何将参数传递给函数？通过栈还是寄存器，如果是栈，那么从左压栈参数还是从右？

② 如何平衡栈？调用方还是被调用方清理栈？

③ 返回值怎样返回？

我们根本不知道 Delphi 的函数是如何处理以上 3 个问题的。看来必须分析 Delphi 的 Add()函数的是怎样传递参数的，在 DM2-12 的第 6 行按 F11 键跟踪进 Add()，反汇编代码见 DM2-13。

DM2-13

```
1 003B89BC   push    ebp
2 003B89BD   mov     ebp, esp
3 003B89BF   add     edx, eax
4 003B89C1   add     ecx, edx
5 003B89C3   add     ecx, dword ptr [ebp+8]
6 003B89C6   mov     eax, ecx
7 003B89C8   pop     ebp
8 003B89C9   ret     4
```

第 1～2 行是典型的保存 ebp 的代码，见 1.3.4 节。

第 3～4 行涉及 EDX、ECX、EAX 三个寄存器的值相加，其和保存在 ECX 中。就像 3 个参数的求和，这个函数本来就是将 4 个参数求和。

第 5 行将前三者之和与 ebp+8 中存储的值相加，再放入 ECX 中。在 1.3 节我们已经知道，一般 ebp+xx 就是参数的地址，而 ebp–xx 就是局部变量的地址，所以这里的 ebp+8 应该是第 4 个参数的地址。总之本指令完成了 4 个参数的求和，并将其存入 ECX 中。

第 6 行将 4 个参数的和从 ECX 赋值给 EAX。Delphi 也是用 EAX 返回值（之后 EAX 也没有变化了），这与 C 语言相同。

第 7 行恢复 EBP 中的值。

第 8 行的指令是 ret　4。在第 3～5 行，我们已经猜想 Delphi 用了 3 个寄存器传 3 个参数，另 1 个参数从栈传递。从本行可印证该猜想。因为本函数采用了被调用函数自身平衡栈，通过 ret　4 指令，将 ESP 因压栈传参减小的 4 字节补了回来。（该指令返回后 ESP 中的值将加 4。）

总结一下：EDX、ECX、EAX 传递 3 个参数，另一个参数压栈，由被调用函数平衡栈，返回值由 EAX 传递。这类似于 C++的 _fastcall。即使都是_fastcall，在选择传递参数的寄存器、从第几个参数开始压栈传递等方面，VC 和 Delphi 的策略是有出入的（请做习题 6 验证）。这个“出入”的结果就是：

调用方用栈传递参数（DM2-12 的第 2～5 行），而被调用方从寄存器和栈取参数（DM2-13 的第 3～5 行），求和的结果必然错误。

调用方（DM2-12 第 7 行）平衡了栈，而被调用函数最后的 ret　4 将栈多加了 4 字节。这与之前观察的结果符合。（调用前 esp 是 0x0012fe78，之后是 0x0012fe7c，多了 4 字节。）

目前来看，只有用 EAX 传递返回值是一致的。怎样让两个语言写的代码“步调一致”？这个调用的“步调”其实就是调用惯例。只要用同一种调用惯例就可以了（传参方式、顺序、谁负责清栈完全相同），刚才报错的英文提示说错误可能因为“...with a different calling convention”。有几种方式可选？cdecl 和 stdcall 均可。现在我们将 Delphi 的 Add()函数修改为 cdecl，在申明该函数的最后加上 cdecl 即可（工程见“code\第 2 章\cross_language\right_solution\DelphiDll”）：

```
function Add(I1: Integer; I2: Integer; I3: Integer; I4: Integer): Integer;cdecl;
```

下面介绍另一种调试 DLL 的方式。之前是从调用方用反汇编跟踪进 DLL。如果想以源代码方式调试 DLL，该怎样？下面的方式同样在 Visual C++中也有相似的模式（见习题 7）。（**如果我们练就了猜测的本领，这些功能完全可以用猜测来搞定。**）分析如下：

要调试 DLL 就要加载该 DLL 并调用其中的相关输出函数，就需要告诉 IDE 运行哪个程序来加载 DLL（用这一思路在 Visual C++的环境中也可找到对应的界面）。Run 菜单应该与这个主题相关，其 Parameter 子菜单比较像，因为用哪个程序加载 DLL 就是一个参数，其菜单界面见图 2.12。其中，Host Application（宿主程序）旁边有 Browse 按钮，一般用于选择文件（暗示与加载程序有关），所以我们将调用程序 vcmainprogram.exe 复制到 DLL 的目录下，并通过 Browse 按钮选择该文件，单击 OK。在 Add()函数的语句上按 F5 键，设置断点，按 F9 键，调试 DLL。（**其实熟悉一个 IDE 后，可以不用再看说明，用这种猜测加“点击”的方式就会用其他 IDE 了。**）这时代码在断点处停止，按 Ctrl+Alt+C 键，即可激活 Delphi 的反汇编界面，见图 2.13。

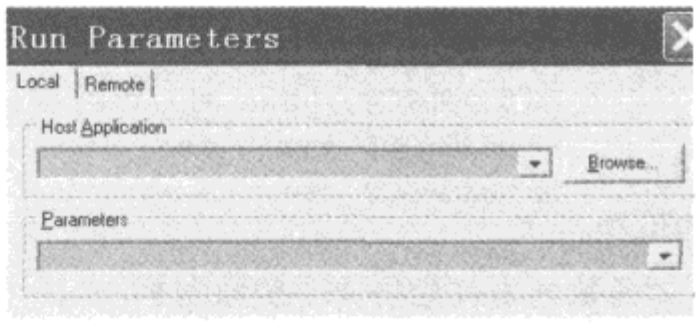


图 2.12

```
Unit1.pas.10: begin
003C89BC 55                 push ebp
003C89BD 8BEC               mov ebp,esp
003C89C2 03450C             add eax,[ebp+$0c]
003C89C5 034510             add eax,[ebp+$10]
003C89C8 034514             add eax,[ebp+$14]
Unit1.pas.12: end;
003C89CB 5D                 pop ebp
003C89CC C3                 ret
```

图 2.13

可知，参数从栈传递，其地址分别为 ebp+0x08、ebp+0x0c、ebp+0x10、ebp+0x14。将它们指向的内存中的值用 add 指令求和，其和均放在 EAX 中，因此返回时其中就是最终结果。最后的指令为 ret，不是 ret　4，这样函数没有负责清栈。当然，最终的结果也就正确了。

有了调用惯例的知识和反汇编调试能力，遇到这些貌似复杂的问题，我们也能分析原因并找到答案，这就是底层的力量。

2.5　错误的目的地

笔者的一个硕士比较迷 Linux，但他基础并不扎实。一天他匆匆而来，告诉笔者，一个 Linux 上的程序调了好几天了，总报错。而且前一天晚上还请教了一个 Linux 高手，研究了一个晚上也没有结果。在他看来是系统错了，因为是执行 strcpy()时报错的。笔者看到他的代码似乎也没有调用 Linux 系统，就迁移到 VC 下调试，结果发现了错误。这个错误有两方面意义：① 程序结构的知识在编程中也能用到；② 更有普遍意义。出错时不要只盯着出错的地方，错误的“凶案现场”可能在其他地方，但真正产生影响时，已经早不在原地了。

DM2-14

```
void enc(char * src, char * des, char key){
    char * ptr = src;
    char * tem = (char *)malloc(strlen(src) + 1);
    int i = 0;
    while (* ptr != 0){
        tem[i] = *ptr ^ key;
        ptr++;
        i++;
    }
    tem[strlen(src)] = 0;
    strcpy(des, tem);
}
```

DM2-14 为其出错代码，完成了简单的异或加密。（除了有点瑕疵，也不至于引起异常。）用调试查看。运行后直接弹出对话框“写入 0x415744 时发生访问冲突”。单击

中断，异常居然发生在 strcat.asm 中：

```
main_loop:                                  ; edx contains first dword of sorc string
        mov   [edi], edx                    ; store one more dword
        add    edi,4                        ; kick dest pointer
```

在执行 mov 指令时出错了。为什么？用调用堆栈界面查看，enc()函数与 strcat.asm 之间确实没有其他函数。到 strcat.asm 文件的头部可以看到注释，说明 strcat.asm 和 strcpy()共用了该代码。

由于没有头绪，所以暂时分析了出错的代码逻辑。其实整个代码有点像 1.9 节分析过的 strlen()的代码，似乎它也是一次复制了 4 字节，就如同 strlen()一次判断 4 字节中是否有 0 字节存在。（**呵呵，多学习有好处啊。**）需要分析 strcpy()的全部代码吗？不用，这里几乎可以假定 edi 就是当前要复制 4 字节的目标地址，edx 是被复制的 4 字节。之后 add　edi, 4 指令应该是将目标地址加 4，准备下一次复制。

mov 出错，难道复制越界了？当笔者准备查看是否越界时，突然想起报错消息中提示的写入出错的地址是“0x415744”，查看此时 edi 的值确实是 0x415744。（这个地址有点奇怪。）我们在第 1 章中学习重定位时知道，程序有一个希望加载基址，一般编译器都会设定为 0x400000，因此 0x4xxxxx 地址一般不是代码段，就是全局数据段。（代码段的属性！“只读”）。只读的东西不能被写入。除了代码段还有哪里是只读的？常量段，如著名的 hello world 中字符串"hello world\n"就是放在常量段中。（**后面我们探索面向对象时，类的虚表也放在其中。**）莫非这个地址指向常量段？（**失策啊，怎么老是关注出错的地方呢？错误现场可能在其他地方。**）我们不用重新执行程序（有些错误不是简单再执行一次就可复现），只需在调用堆栈界面中点击出错函数的上一层函数，即 enc()，然后在监视窗体中查看相关变量即可。

第一要看的自然是 DM2-14 中 strcpy()复制的目标地址 des。地址是 0x415744，居然是报错时提到的发生写错误的地址。也就是说，strcpy()在第一次写入时就出错了。这个 des 是作为参数传入的，明显传入了错误的 des，它指向了常量段！（**终于到了查看真正凶案现场的时候了。**）在调用堆栈中单击调用 enc()的函数，看到 DM2-15 中的代码。从第 3 行可知，复制的目标地址是 pDes。再看谁为 pDes 关联了内存？第 1 行中，pDes 指向了"12345678"，即指向了常量区"12345678"的地址。（**居然传递了一个常量的地址然后往里面写数据！**）为了验证，笔者在调用 enc()之前直接加了“*pDes = 'a';”语句，结果在执行该语句时抛出了一模一样的错误信息。（**出现这样的问题应该是没有熟悉程序的基本结构的概念，基础真的很有用。**）

DM2-15

```
1  char* pDes = "12345678";
2  char* encryed = "hello";
3  enc(encryed, pDes, 0);
```

其实我们可以往常量区写入，笔者告诉了学生 Windows 下的相关接口。后来他告诉笔者，“真的可以哦”。大家做一下习题 10 吧。本章涉及的完整程序见“code\第 2 章\WrongDestine”。

2.6　网络发送出错了

很多时候，调试完毕，发现问题是一个非常简单的错误，只是常常没有想到。这也许是需要调试的原因吧，它能逼迫人们一步一步走过一些看似完全正确却可能蕴含错误的代码，然后在某个地方突然提醒你，关注那些完全不曾怀疑的东西。下面的例子是一个 Delphi 项目，除了再次体现函数调用过程这一基础知识的重要性，也展示了代码分析中的“脑筋急转弯”。

两年前，为了测试一个 erlang 写的服务器性能，一个很擅长 Delphi 的程序员用 Delphi 写了一

个网络客户端程序，由它向服务器发送约定报文。预计半天能完成，可两三天过去了，完全没有进展。程序员说，关键问题是：Delphi 撰写的程序虽发出了 UDP 报文，但内容非约定格式。这一点通过跟踪服务器收到的内容甚至抓包分析均可证明。更麻烦的是，其代码非常简单，完全找不出问题（直接用 socket 相关函数发送，函数返回结果也正确）。笔者只能判断还是其代码有问题，没有发出正确内容。怎样证明？反汇编。

DM2-16

```
1  var
2    sock: Cardinal;
3    addr: sockaddr_in;
4    pbuf: PChar;
5  begin
6    sock := socket(AF_INET,SOCK_DGRAM,0);
   ...
7  GetMem(pbuf, 6);
8  StrCopy(pbuf, 'hello');
9  sendto(sock, pbuf, 5, 0, addr, sizeof(sockaddr_in));
   ...
```

DM2-16 是相关代码（**被完全简化了，如果对于更复杂的生产代码，大家要学会抓住关键点**），工程请参考“code\第 2 章\sendwrongcontent”。这段代码似乎一点问题都没有。PChar 相当于 char *。第 7 行为 char *指针 pbuf 分配了 6 字节。第 8 行 StrCopy() 相当于 C 语言的 strcpy()函数，将字符串 hello 复制到 pbuf。第 9 行调用 UDP 的发送函数 sendto()。反汇编，在第 9 行按 F5 键设置断点，执行后停下，按 Ctr+Alt+C 键，将反汇编窗体激活。

先查看 pbuf 的值。按 Ctr+Alt+W 键，激活监视窗体，双击弹出的界面，然后在弹出窗体的 expression 输入框中输入“@pbuf[0]”，获取 pbuf 指向的字符串的地址，回车。其值为$D43888。

在内存窗体中查看地址$D43888 的内容。在反汇编窗体的下部面板中单击右键，在弹出的快捷菜单中选择“Goto Address...”，在弹出对话框中输入“$D43888”，回车。内存窗体见图 2.14，左侧是行的起始地址，为 00D43888H。该地址指向的内容在面板中间，十六进制表示为 68 65 6C 6C 6F 00。其 ASCII 码表示为 hello，在面板右边。可见，发送前，缓存 pbuf 指向的内存确实是要发送的 hello 字符串。

```
00D43888 68 65 6C 6C 6F 00 00 00  hello...
00D43890 00 00 00 00 00 00 00 00  ........
```

图 2.14

sendto 确实发送了该缓存内容吗？由 DM2-16 中第 9 行可知，第 2 个参数是要发送的内存地址，第 3 个参数为发送长度 5 字节。查看现在已经激活的反汇编窗体，见图 2.15。从最后一行 call 指令向上分析，因为 sendto 是 Windows 的 API，调用惯例是 stdcall，所以参数从右往左压栈传递。call 指令上方的 push　ebx 是压栈 sock 参数，上数第 2 行 push　eax 应该压栈发送缓存的地址，上数第 4 行 push　$05 显然是压栈发送长度 5。现在我们需要老老实实看看压栈的值到底正确与否。单步 f7 每执行一步查看相应压栈的值是否匹配。执行到压栈 pbuf 的 push　eax 指令时，查看 eax 的值（寄存器面板在反汇编窗体的右上角，见图 2.16），是 0012F3D0H。它与图 2.14 中的 00D43888H 不同。也就是说，压栈传递的不是缓存的首址 00D43888H，而是 0012F3D0H。

```
Unit1.pas.51: sendto(sock, pbuf, 5, 0, addr,  sizeof(sockaddr_in))
0044DD0F 6A10            push $10
0044DD11 8D442408        lea eax,[esp+$08]
0044DD15 50              push eax
0044DD16 6A00            push $00
0044DD18 6A05            push $05
0044DD1A 8D442410        lea eax,[esp+$10]
0044DD1E 50              push eax
0044DD1F 53              push ebx
0044DD20 E80BFDFFFF      call -$000002f5
```

图 2.15

图 2.16

验证一下。在内存面板中查看 0012F3D0H 地址，见图 2.17，其中根本不是存储的 hello。

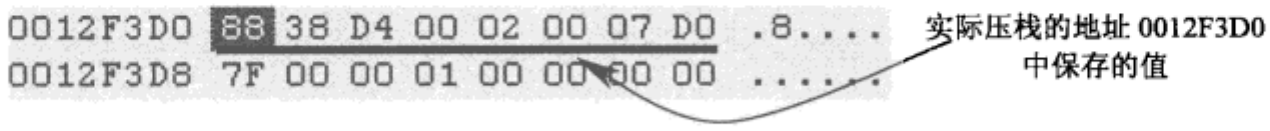


图 2.17

这个 0012F3D0H 到底是什么？为什么会被编译器传递给 sendto？难道编译器错了？sendto 在 Delphi 中的原型如下：

```
function sendto(s:TSocket; var Buf; len, flags:Integer;
                var addrto:TSockAddr; tolen:Integer):Integer; stdcall;
```

第 2 个参数 buf 前面加了 var，即 C++中的引用。（传递变量时，它帮你传递变量的指针。）而在 C 语言中，sendto 的第 2 个参数被定义为地址。习惯了 C 语言的人下意识就会将指针，也就是 pbuf 传入（见 DM2-16 的第 9 行），但这时按照 pascal 的语法，要写成 pbuf[0]：

```
sendto(sock, pbuf[0], 5, 0, addr,  sizeof(sockaddr_in));
```

这样表面上传入的是 pbuf[0]的值，其实传入的是其地址&pbuf[0]。这就是反汇编调试的意义。

2.7 为什么代码运行完毕却出错

有时候源码虽然简单，但高级语言会暗地里加入许多指令。如果我们没有理解这背后的规则，那错误离我们就不远了。当你面对一段源码百思不得其解时，用反汇编打开黑盒是前进的不二法门。

下面的例子展示了这一过程。大概是 2002 年，我们用 Delphi 开发一个网络自动防护系统。当时引入了单元测试 DUnit。在一次更新后，测试总报错。最后定位到一个测试用例，惊讶地发现，错误并非在有效的源码中产生，而是源码执行完毕，在执行类似 C 语言函数的右大括号时出错了！（**都没有代码执行了，还错？**）没有办法，祭出我们的法宝“反汇编调试”。（**它就像溺水人的最后稻草，又像忠实的挚友，总在最需要的时候给我们有效支持**。）为了展示错误，必须先陈述一些相关知识。

首先简单介绍一个相关概念，interface 是 Delphi 的一个关键字，与 Java、C#中的含义基本一样，可以在单根继承的限制下，获得多继承的好处：一个类可实现多个 interface 但只能继承一个类。为了与微软的 COM 结合，所有接口必须实现 COM 的最基本方法：

```
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
```

其中，_AddRef 和_Release 用于实现引用计数管理对象生存期，QueryInterface 是查询接口。引用计数是在没有 Java 的垃圾回收机制的情况下一种简易实用的内存管理机制。当一块代码要引用一

个对象时，就调用它的_AddRef，增加引用计数；不用时，就调用_Release 减少引用计数。当计数归零，该对象将在_Release 中被释放。下面例子的_Release 方法实现了如上机制：

```
function TTestObj._Release: Integer;
begin
  FRef := FRef - 1;                    //FRef 是类 TTestObj 的成员变量，实现引用计数功能
  Result := FRef;
  if FRef = 0 then                     //如果引用计数归零，则释放该对象
    Free;
end;
```

在本例中，我们定义一个接口 ITest：

```
ITest = interface
    procedure DoIt;
end;
```

该接口需要实现方法 DoIt，然后用类 TTestObj 实现了该接口。类 TTestObj 继承于 TObject 类，实现了 ITest 接口：

```
TTestObj = class(TObject, ITest)
  private
    FRef: Integer;
  public
    constructor Create;                //构造函数
    destructor Destroy;override;       //析构函数
    procedure DoIt;                    //实现 ITest 接口
    //实现根接口 IInterface，支持 COM 的最基本接口
    function QueryInterface(const IID: TGUID; out Obj) : HResult; stdcall;
    //引用计数增加
    function _AddRef: Integer; stdcall;
    //引用计数减小
    function _Release: Integer; stdcall;
end;
```

DM2-17 是按照原理简化的出错代码，工程见“code\第 2 章\interface”。需要说明的是，Delphi 中所有对象都是指针。Obj: TTestObj 相当于 C++语言的 TTestObj *obj。第 7 行构造了 TestObj 的对象，赋值给类型为 TTestObj 的 Obj 指针。因为 Obj 是 TTestObj 类，而该类实现了 ITest 接口，且 IT 变量的类型是 ITest 类型，所以第 8 行可以将 Obj 赋值给变量 IT。第 9 行以接口 ITest 的身份调用了接口的方法 DoIt。第 10 行调用了 Obj 的析构函数释放该对象。第 11 行将全局变量 G 赋值为 1，G 在第 1 行定义。

DM2-17

```
1 var G: Integer;
2 procedure TForm1.Button1Click(Sender: TObject);
3 var
4   IT: ITest;
5   Obj: TTestObj;
6 begin
7   Obj := TTestObj.Create;
8   IT := Obj;
9   IT.DoIt;
10   Obj.Free;
11   G := 1;
12 end;
```

程序执行出现异常，源代码级单步跟踪揭示出执行 11 行“G := 1”时出错了。一个简单的全局变量赋值怎样可能出错呢？用反汇编来看个究竟，见图 2.18。

按 F8 键，单步跟踪，执行“G := 1”后，在执行 mov　[G], $00000001 指令时并没有出错，而是执行到 call　@IntfClear（倒数第 2 行）时才报错的。现在需要搞清楚，从第 1 行 mov 指令后至

倒数第 2 行的这段代码到底在做什么？这段代码肯定与 G :=1 没有关系，与哪段代码有关？我们可采用二分法来寻找这段代码的关联性。

```
Unit1.pas.71: G := 1;
0044DCEE C70524FC44000100 mov [G],$00000001
0044DCF8 33C0             xor eax,eax
0044DCFA 5A               pop edx
0044DCFB 59               pop ecx
0044DCFC 59               pop ecx
0044DCFD 648910           mov fs:[eax],edx
0044DD00 6815DD4400       push $0044dd15
0044DD05 8D45FC           lea eax,[ebp-$04]
0044DD08 E8A781FBFF       call @IntfClear
0044DD0D C3               ret
```

图 2.18

二分法寻找关联代码：当一些反汇编代码无法找到对应的源代码时，可以先将这段反汇编代码可能涉及的语句全部注释，看看该汇编码是否消失。如果没有，就属于函数调用本身。如果消失，就属于某行源代码。

DM2-23(b)（正常版本）

二分寻找法

注释 DM2-17 的第 9～11 行，这段调用 IntfClear 的指令依然在，见图 2.19。

```
Unit1.pas.68: IT := Obj;
0044DCCE 8D45FC           lea eax,[ebp-$04]
0044DCD1 8BD3             mov edx,ebx
0044DCD3 85D2             test edx,edx
0044DCD5 7403             jz +$03
0044DCD7 83EAF8           sub edx,-$08
0044DCDA E8ED81FBFF       call @IntfCopy
0044DCDF 33C0             xor eax,eax
0044DCE1 5A               pop edx
0044DCE2 59               pop ecx
0044DCE3 59               pop ecx
0044DCE4 648910           mov fs:[eax],edx
0044DCE7 68FCDC4400       push $0044dcfc
0044DCEC 8D45FC           lea eax,[ebp-$04]
0044DCEF E8C081FBFF       call @IntfClear
0044DCF4 C3               ret
```

第二次注释后消失的代码

图 2.19

图 2.19 中矩形框圈定的代码与图 2.18 中 xor　eax, eax 开始的代码还是一样。可见，这段代码还没有消失。继续查找，将第 8~11 行注释，即多注释了一行 IT := Obj。其反汇编代码见图 2.20。

对比图 2.19，指令“lea　eax, [ebp -$4]”和“call　@IntfClear”消失了。这应该与注释的第 8 行有关。图 2.20 中矩形框圈定的指令与图 2.19 中依然相同。可见，这应该是与函数相关的一段共同代码（其实是异常链恢复代码）。而且，删除第 8 行后还消失了一段代码，就是图 2.19 中花括号圈定的代码。

```
Unit1.pas.67: Obj := TTestObj.Create;
0044DCBD B201             mov dl,$01
0044DCBF A108DC4400       mov eax,[$0044dc08]
0044DCC4 E897FFFFFF       call TTestObj.Create
0044DCC9 33C0             xor eax,eax
0044DCCB 5A               pop edx
0044DCCC 59               pop ecx
0044DCCD 59               pop ecx
0044DCCE 648910           mov fs:[eax],edx
0044DCD1 68DEDC4400       push $0044dcde
0044DCD6 C3               ret
```

图 2.20

可以推断，有了第 8 行后，加入了 call @IntfCopy 和 call @IntfClear 相关指令。对比图 2.18 和 2.19 可以发现，call @IntfClear 相关指令总是放在函数要返回前、所有源代码执行后。在图 2.18 中，源代码最后一行是 G := 1，call @IntfClear 相关指令就在其后。在图 2.19 中，源代码最后一行是 IT := Obj，call @IntfClear 就在该行之后。可以推断出以下规律：

将一个对象赋值给一个接口类型变量时（IT := Obj，IT 是接口变量），会加入 call @IntfCopy 调用，同时在函数结束时将悄悄加入 call @IntfClear 调用。

但现在还不清楚这两个调用做了什么，只有跟踪才能明白。如何跟踪？断点回溯法。

跟踪反汇编码和自己相关代码的方法

方法一：按 F7 键单步跟踪，遇到 call 指令就 step into，直到发现感兴趣的东西。这样可能比较慢。

方法二：在可能涉及自己书写的代码中设置断点，遇到 call 指令就按 F8 键，此时如果该 call 指令执行的代码最后调用到自己书写的代码，那么它就会停在事先设置的断点。这样能更快理清自己写的代码与编译器生成代码间的互动关系，更容易理解程序的逻辑流。

用方法二，在实现的 TTestObj 所有方法中设置断点。执行 call @IntfCopy 后会调用 TTestObj 的_AddRef 方法，而执行 call@IntfClear 后会调用 TTestObj 的_Release 方法。这时我们可还原出以下调用逻辑：

当一个对象赋值给接口变量时（IT := Obj）将自动调用该接口的引用计数方法_AddRef。

在该接口变量的作用域结束时（如上例中，IT 是局部变量，其作用域是整个函数，call @IntfClear 总是发生在函数最后一行源代码之后，这时用户代码不会使用该接口变量，因此调用_Release 减少引用计数），自动调用该接口的_Release，减少引用计数。

编译器这一做法保证了接口变量使用时自动增加引用计数，而其作用域结束时，自动减少引用计数。这样从编译器层面保证了引用计数策略的正确实施，否则程序员必须手动调用，存在忘记调用的可能。这是引用计数策略的最大弱点。除了局部变量这种情况，请读者自行分析还可能有哪些情况编译器会自动调用_AddRef 和_Release，见习题 11。

现在再来看 DM2-17 的错误就比较清楚了。第 8 行会自动调用_AddRef，导致对象 Obj 的引用计数为 1。因为第 10 行调用 Obj.Free 释放了该对象，而函数结束前又因为自动调用_Release 再次释放该对象（调用_Release 导致引用计数归零释放该对象）。两次释放同一个对象肯定出错。

当时笔者出错是因为之前一直不用 interface，养成了自己调用析构函数释放对象的习惯，同时又不知道对于 interface 类型的变量编译器会悄悄加入_Release 调用。这一错误发生在局部变量使用时还比较好定位，如果接口变量是类的成员变量，就很棘手了，因为即使用反汇编分析，错误可能不在出错现场。这里顺便提醒 Delphi 的使用者，如果用了接口，请在它被构造出以后，一直以接口身份去使用它，不要再主动编写代码释放该对象，否则错误极难发现。

反汇编代码很多，要找到关键代码，理出关心的逻辑线索，除了分析、猜测，还要一些小小的技巧。大家应该充分利用调试器，总结技巧，快速找出问题关键。

2.8 失效的管道

在这个网络普遍化的时代，知识似乎很廉价了。只要在 Google 中输入一些东西，那些深埋在世界角落的东西就会马上呈现在你眼前。但真正的秘密隐藏在这些廉价的泡沫和耀眼后，我们必须

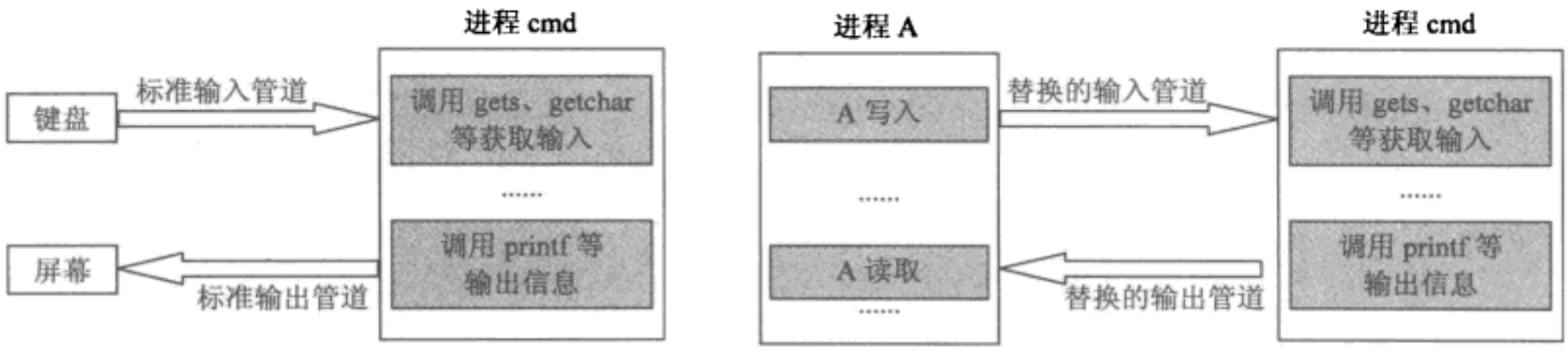
用自己的努力去挖出来。从本节开始，我们需要在黑暗中用艰辛的分析和考古似的细致去点亮光芒，叩开那些所罗门宝藏，欣赏一幅幅精美的智慧浮雕，并在这旅途中悄然成长。这是智慧的夜曲，也是底层的力量。

要开始本节，我们先得领会管道的强大功能，再去面对和分析奇怪的一幕。

2.8.1 管道的力量

假定现在想撰写一个远程 Shell，它收到从网络发来的命令，就在本地执行。比如，收到 dir 命令，将目录下的文件项列出来；收到 del 命令，将相关的文件删除等。这些命令是 Windows 下的 Shell 程序 cmd.exe 实现的功能。Shell 的命令非常多，难道需要我们将一条一条命令都实现出吗？工程量太巨大了，而且似乎是做无用功，因为 cmd.exe 都实现了这些功能。能否利用 cmd.exe 呢？

cmd.exe 的界面就是 DOS 控制台，从键盘输入命令，结果显示在屏幕上。如果写远程 Shell，命令都来自网络，通过网络收到的命令字符串如何交给 cmd.exe？cmd 是以键盘作为输入。UNIX 将所有的输入、输出信息抽象成一个管道（pipe），输入者从管道的一头灌入字节流，接收者从管道的另一头接收字节流。程序必须有一个基本的输入和输出管道，才能与外界交互。这个基本输入、输出管道称为标准输入/输出。对于 cmd 而言，可想象它的标准输入、输出管道是这样构建的：输入管道的输入端接的是键盘，输出端接的是程序，于是键盘敲击被转换为一个个字节流入管道，并被程序接收（如调用 gets()、getchar()、scanf()之类的函数接收流入管道的字节），输出管道的输入端是 cmd，程序通过诸如 printf()之类的函数将字节灌入到管道，就显示到管道的另一端“屏幕”上。其实，程序根本就不知道它拥有的标准输入、输出管道连接的是键盘还是屏幕。只要有机制在启动 cmd 程序时将标准输入、输出管道替换成其他管道，我们就可通过该输入管道输入 Shell 命令字符串，控制 cmd 执行这些命令，并通过输出管道接收 cmd 执行命令后的输出（那些显示在屏幕上的结果也是字符串）。这样程序启动 cmd 后，替换其标准输入、输出管道，就可与 cmd 的进程交互，见图 2.21。



(a)未替换管道前　　(b)替换管道后

图 2.21

Windows 提供构造新进程的函数为 CreateProcess，它的一个参数可将输入/出管道的 Id 传入给新进程并替换其标准输入出管道。CreateProcess 说明如下：

```
BOOL CreateProcess(LPCTSTR lpApplicationName, LPTSTR lpCommandLine,
                   LPSECURITY_ATTRIBUTES lpProcessAttributes,
                   LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL bInheritHandles,
                   DWORD dwCreationFlags, LPVOID lpEnvironment,
                   LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo,
                   LPPROCESS_INFORMATION lpProcessInformation);
```

第 1 个参数可传入空指针 NULL；第 2 个参数为包括程序执行文件名（可以是全路径名）和命

令参数的字符串，如要执行 netstat –a –n 就可输入该字符串；第 3、4 个参数也可传入 NULL；第 5 个参数一般用 false，指出新进程是否可使用父进程的所有句柄；第 6 个参数是进一步控制生成新进程的相关方式、属性，一般用 CREATE_DEFAULT_ERROR_MODE；第 7、8 个参数可用 NULL；第 9 个参数为结构体 STARTUPINFO 的指针，该结构体有 3 个成员变量 hStdInput、hStdOutput、hStdError，均为句柄，如果用新的管道替换标准输入、输出和错误输出管道，就将新管道句柄赋值给这 3 个成员变量，这涉及利用管道进行跨进程通信；第 10 个参数为指针，指向 PROCESS_INFORMATION 结构体，函数返回后该结构体成员变量被填写，包括进程 id、句柄、主线程 id 和主线程句柄。

CreatePipe()构造管道，返回两个整数，代表管道的两端。代表读端的整数传递给 ReadFile()即可读管道，代表写端的整数传递给 WriteFile()，即可写管道。函数原型如下：

```
BOOL CreatePipe(PHANDLE hReadPipe, PHANDLE hWritePipe,
                LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize);
```

第 1 个参数为 HANDLE 指针，返回后其指向变量填写了代表读端的整数；第 2 个参数为 HANDLE 指针，返回后，其指向变量填写了代表写端的整数；第 3 参数个为安全属性，一般为 NULL；第 4 个参数为管道长度，当写管道时，如果管道中尚未读走的字节长度达到该值，写动作就会阻塞等待，直到管道中字节被读走方可继续写。

ReadFile()可读取任何抽象的文件，包括磁盘文件、驱动、管道等，原型如下：

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
```

第 1 个参数为文件句柄，读管道时就传入 CreatePipe()第 1 个参数指向的整数；第 2 个参数是要读入的缓冲地址；第 3 个参数为要读的长度；第 4 个参数是 unsigned long 的指针，返回后，被填入实际读出的数据长度；第 5 个参数如无特殊需求，填入 NULL 即可。成功读取数据后管道中已读的数据将被删除。

WriteFile()可写任何抽象的文件，包括磁盘文件、驱动、管道等。原型和 ReadFile()一样，第 1 个参数为文件句柄，写管道时就传入 CreatePipe()第 2 个参数指向的整数。其他类似。

由于在没有数据可读时，调用 ReadFile()将阻塞，所以引入另一函数 PeekNamedPipe()。它可复制数据，并不删除管道中的数据，且能返回可读数据的长度，还可只返回可读数据的长度，而不复制数据（如第 2 个参数为 NULL，就不复制数据）。更关键的是，PeekNamedPipe()不阻塞，因此可来探测管道中可读数据的多少，然后用 ReadFile()读走它们。该函数原型：

```
BOOL PeekNamedPipe(HANDLE hNamedPipe, LPVOID lpBuffer, DWORD nBufferSize,
                   LPDWORD lpBytesRead, LPDWORD lpTotalBytesAvail,
                   LPDWORD lpBytesLeftThisMessage);
```

第 1 个参数为管道读端的句柄，是 CreatePipe()第 1 个参数指向的整数；第 2 个参数为读缓存的地址，如果为 NULL，则不复制数据；第 3 个参数为缓冲大小，如果第 2 个参数为 NULL，将忽略该值；第 4 个参数为指针，函数调用成功，则其指向变量为实际读到的长度，如果不读数据，可为 NULL；第 5 个参数为指针，函数调用成功，则其指向变量表示有多少数据可读，也可为 NULL；第 6 个参数可为 NULL。

例子程序用 CreateProcess()建立 cmd.exe 的新进程，并替换其标准输入、输出和错误管道。用户输入的命令，程序通过输入管道传递给 cmd；而 cmd 的输出，程序从输出管道读取，并打印在屏幕上。工程见“code\第 2 章\ctrlshell”。该程序关键由以下几部分组成：DM2-18，CreateShell 的主要部分，创建进程 cmd 并构造管道替换 cmd 进程的标准输入、输出和错误管道；DM2-19，printMsg

的主要部分，读取管道，将 cmd 输出信息打印到屏幕；DM2-20，main()函数框架，程序主流程。

DM2-18

```
1   bool CreateShell(HANDLE * readHnd, HANDLE * writeHnd, PROCESS_INFORMATION * pi){
2       if (!CreatePipe(&rh1, &wh1, &sa, 1024)){
3           ...
4       }
5       *readHnd = rh1;
6       if (!CreatePipe(&rh2, &wh2, &sa, 1024)){
7           ...
8       }
9       *writeHnd = wh2;
10      memset(&si, 0, sizeof(si));              //必须执行它，为 si 赋默认值，否则第 16 行无法成功构造进程
11      si.dwFlags = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
12      si.wShowWindow = SW_HIDE                 ;//隐藏新建的 cmd 进程，不会显示黑色的控制台窗体
13      si.hStdInput = rh2;
14      si.hStdOutput = wh1;
15      si.hStdError = wh1;
16      if (!CreateProcessA(NULL,"cmd.exe",NULL,NULL,true,CREATE_DEFAULT_ERROR_MODE,NULL,NULL,&si,pi)){
            ...
```

DM2-19

```
1   bool printMsg(HANDLE readH){
2       char output[1024];
3       unsigned long n, n2;
4       for (int i = 0; i < LOOP_NUM_GET_OUTPUT; i++){
5           while (true){                                          //循环几次将剩余的输出数据读完
6               if (!PeekNamedPipe(readH, NULL, 0, NULL, &n2, NULL)){
7                   printf("cannot get msg\n");
8                   return false;
9               }
10              if (n2 == 0)
11                  break;
12              if (!ReadFile(readH, output, READ_NUM, &n2, NULL)) {
13                  printf("cannot read file\n");
14                  continue;
15              }
16              output[n2] = NULL;
17              printf("%s", output);
18          }
19          Sleep(LOOP_MS_GET_OUTPUT);                             //休眠一下等待剩余的输出数据
20      }
21      return true;
22  }
```

DM2-20

```
void main(){
1   ...
2   if (!CreateShell(&readH, &writeH, &pi)) {          //建立 cmd 进程
3       return;
4   }
5   unsigned long n;
```

DM2-20（续）

```
6   printMsg(readH);                                  //cmd 启动将输出一些信息，读出它们并显示
7   while (true){
8       gets(cmd);                                    //接收输入命令
9        if (strcmp(cmd, "exit") == 0) {              //如果输入命令是 exit，就退出本程序
10          TerminateProcess(pi.hProcess, 0);         //结束 cmd 进程
11          return;
12      }
13      int cmdLen = strlen(cmd);
14      cmd[cmdLen] = '\n';                           //必须在用户输入命令后加入\n，相当于回车
15      WriteFile(writeH, cmd, cmdLen + 1, &n, NULL); //给 cmd 输入命令
16      getMsg(readH, cmdLen + 1);                    //因输出的第一行为输入命令本身，将其读出但不显示
17      if (!printMsg(readH)) {                       //读取 cmd 的输出，即 cmd 执行后打印在屏幕上的结果
18          TerminateProcess(pi.hProcess, 0);
19          return;
        }
    }
}
```

DM2-18 中，生成的管道句柄值将赋给指针 readHnd 和 writeHnd 指向的变量（第 5 行和第 9 行）。读取 readHnd 可获取 cmd 输出到屏幕的信息。写 writeHnd 则代替键盘，将命令串输入给 cmd。第 11 行标志 STARTF_USESTDHANDLES 令第 13～15 行有效，另一个标志使第 12 行有效。

第 2 行构造一个管道，其读端句柄在第 5 行设定给 readHnd，其写端 wh1 在第 14 和第 15 行替换新建 cmd 进程的标准输出和错误输出管道。这样 cmd 输出的信息从管道写端 wh1 写入，程序在 DM2-19 中将从该管道的读端 readHnd 中读入（第 12 行读入后就可打印在屏幕上了）。第 6 行构造一个管道，其写端在第 9 行设定给 writeHnd，其读端 rh2 在第 13 行替换新建 cmd 进程的标准输入管道。这样 cmd 将从 rh2 读入本应从键盘读入的数据。

程序在 DM2-20 中通过 writeHnd 代替键盘写入命令（第 15 行）。第 16 行创建 cmd 进程，最终结果建立两个新管道成功替换原来的信息通道，见图 2.22。

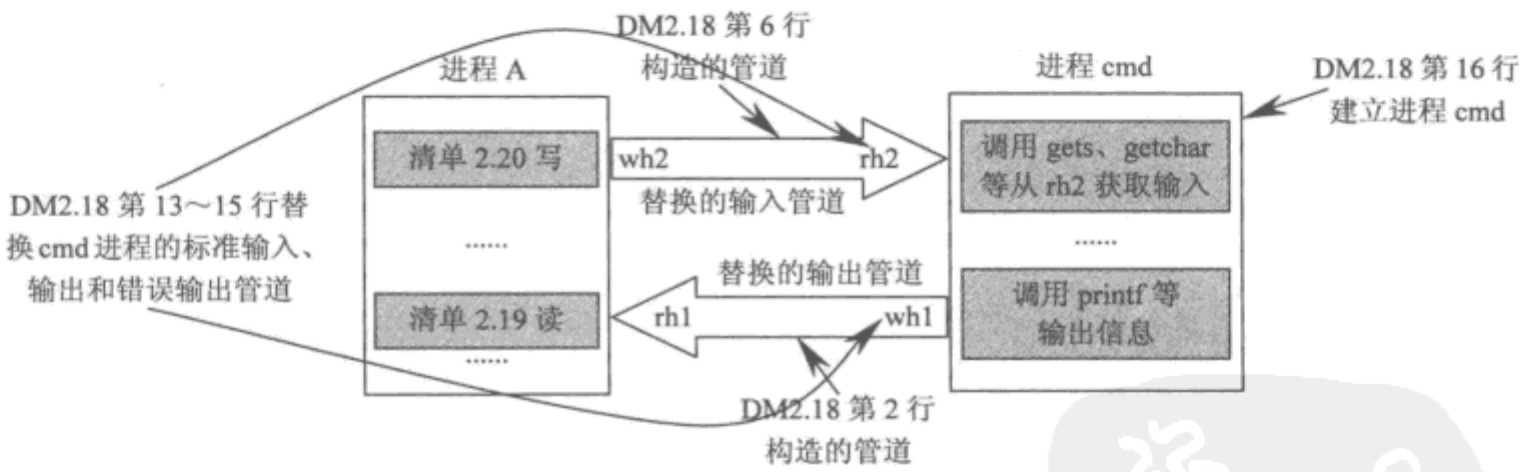


图 2.22

DM2-19 展示了从管道读 cmd 输出数据的逻辑。因为有时输出比较多，所以需要等待才能接收完所有输出，第 4～20 行是一个循环读取的过程，每次循环都要在第 19 行休眠等待一会。

第 5～18 行是循环读取 cmd 的输出，先在 6 行测试有多少数据可读，在第 10 行测试如果可读数据为 0，则退出本次读循环。否则在第 12 行用 ReadFile 从管道读取输出。读到后，第 16 行在最后一个字符后添加 NULL，这样第 17 行打印时才能正常打印。

DM2-20，主程序框架，逻辑见注释。这个程序很容易修改为远程 shell，只要将第 8 行从键盘

获取命令的语句 gets 修改为从网络接收命令，将 DM2-19 中的第 17 行输出结果到屏幕的 printf 语句修改为向网络发送结果信息即可。

2.8.2 我要控制 Telnet 客户端

曾经在一个项目中需要控制 Cisco 的 2970 交换机（编程控制关闭和打开其某个端口）。由于无法获得编程接口，所以只有分析交换机的控制程序。发现：可用 HTTP 控制，或者用 Telnet 登录控制。HTTP 的相关控制协议并没有文档，是通过网页直接控制的。Telnet 的控制命令有相关文档说明。那么，我们自然希望编写代码通过 Telnet 发送相关控制命令操纵交换机。其实，很多网络设备都提供由 Telnet 控制的方式，如果编写出这样的代码，会有非常普遍的用途。

但自己实现 Telnet 客户端协议还是有点麻烦。在 Windows 平台上，C/C++代码一时又查找不到可用模块。有了 2.8.1 节的经验，我们希望通过管道控制 Telnet 客户端程序（就像控制 cmd.exe 那样），向 Telnet 发出交换机控制命令，再由 Telnet 发到交换机。于是笔者编写了替换管道控制 Telnet 的代码，工程见“code\第 2 章\ctrltelnet”。

但奇怪的是进程列表中根本没有 Telent，而 CreateProcess()返回值显示成功。难道程序执行后又马上退出了？将类似 DM2-18 的第 12 行隐藏窗体的选项去掉，一个黑色的窗体晃了一下就消失了。可见，Telnet 确实启动又退出了。经过多方修改参数未果后，笔者查找百度，几乎没有信息。在 Google 中搜索纯英文信息，英文论坛上有不少人在问这个问题，但是无人回答。为了不耽误进程，笔者放弃了 C 语言，用 Delphi 下的 Indy 封装的 Telnet 组件继续开发（很快就完成了）。自己用 IDA pro 反汇编了 Telnet.exe，还真发现了问题。

2.8.3 不是所有管道都可抽象等价

为了保证新建 Telnet.exe 代码的正确性，首先做了一个对比试验：不替换管道，直接新建 Telnet，其他什么都不动，这时能成功启动 Telnet。说明确实是因为替换管道而造成的。现在的问题是为什么替换了管道就不行了呢？

这时我们必须做一些猜想，否则根本无法分析。代码一定是获取到替换后的管道句柄，然后做出某种操作时出错的，因此首先应该获取到替换后的句柄。一般而言，Windws 的 API 中要获取某个东西，喜欢用 Get 做前缀，如 GetCurrentProcess 之类。这里既然是替换了标准输入/输出，那么可能是 GetStandardInput/Output 之类的函数吧？微软的帮助中 GetS 做前缀的太多了，我们可利用一个小技巧快速定位。

程序要调用动态链接库中某个函数，除了用 LoadLibray + GetProcAddress 这种显式加载方式，还有一种叫隐式加载，即执行程序在一个导入表中申明要调用函数的名字及其归属 DLL，那么执行程序在开始运行前，加载器就会帮程序载入这些 DLL。利用这个导入表，可查看程序依赖的函数。IDA 反汇编 Telnet.exe，单击 imports，结果见图 2.23。

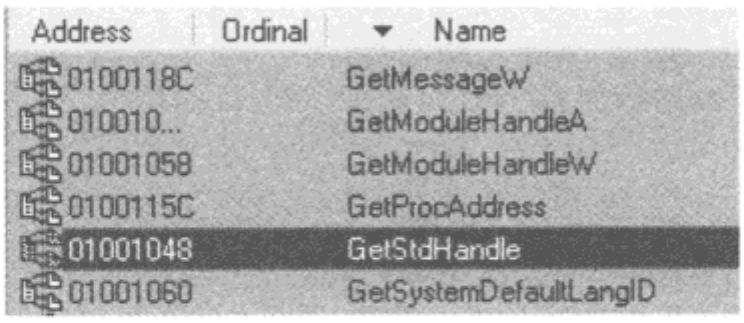


图 2.23

然后单击 Name 表头，导入的函数名将字母排序。找到 GetS 开始的函数，第一项就是一个叫 GetStdHandle 的函数。Std 即 standard 的缩写，Handle 即句柄，连在一起就是获取标准句柄嘛。查看该函数的原型如下：

```
HANDLE GetStdHandle(DWORD nStdHandle);
```

当传入为 STD_INPUT_HANDLE、STD_OUTPUT_HANDLE、STD_ERROR_HANDLE 三个不同整数时，返回值分别代表标准输入、输出和错误句柄。

DM2-21

```
1                       push   0fffffff5h
2   .text:0100c83c      mov    hconsoleinput, eax
3   .text:0100c841      call   edi                    ; GetStdHandle
4   .text:0100c843      cmp    eax, ebx
5   .text:0100c845      jz     loc_100c92f
6   .text:0100c84b      push   offset consolescreenbufferinfo
7   .text:0100c850      push   eax
8   .text:0100c851      mov    hconsoleoutput, eax
9   .text:0100c856      mov    dword_1025d18, eax
10  .text:0100c85b      call   ds:getconsolescreenbufferinfo
11  .text:0100c861      test   eax, eax
12  .text:0100c863      jz     loc_100c92f
```

双击图 2.23 中的 GetStdHandle 条目，将转到一个界面，列出了调用该函数的代码。我们选择其中一处，转跳过去（或按 Ctrl+X 键，在弹出的窗体中选择一处代码转跳），见 DM2-21。

第 3 行调用了 GetStdHandle，其参数是在第 1 行压入的。那个压栈的整数应该是 STD_INPUT_HANDLE 那三个宏中的一个。将 STD_INPUT_HANDLE 随便写入到一个 VC 工程源文件中，然后右键单击该字，在弹出的快捷菜单中选择“转到定义”，有如下宏定义：

```
#define STD_INPUT_HANDLE    ((DWORD)-10)
#define STD_OUTPUT_HANDLE   ((DWORD)-11)
#define STD_ERROR_HANDLE    ((DWORD)-12)
```

把十进制数换算成十六进制数，第 1 个宏为 0xfffffff6，第 2 个宏为 0xfffffff5，第 3 个宏为 0xfffffff4。在 DM2-21 中，第 1 行压栈的是 0xfffffff5，对应宏 STD_OUTPUT_HANDLE，可见第 3 行是想获取标准输出句柄。

第 10 行调用了函数 GetConsoleScreenBufferInfo，其函数原型为：

```
BOOL GetConsoleScreenBufferInfo(HANDLE hConsoleOutput,
                                PCONSOLE_SCREEN_BUFFER_INFO lpConsoleScreenBufferInfo);
```

第 1 个参数是标准输出句柄，第 2 个参数是结构体 CONSOLE_SCREEN_BUFFER_INFO 的指针。

第 3 行获取标准输出句柄后放在 eax 中，第 10 行调用 GetConsoleScreenBufferInfo 前在第 7 行压入 eax，即标准输出句柄，为其传递第 1 个参数（参数从右往左压，所以第一个参数最后压栈）。要获取的 CONSOLE_SCREEN_BUFFER_INFO 包括光标位置、屏幕为几行几列等信息。按照推理，CreateProcess()将管道的句柄传入替换了真正的标准输入/出句柄，而 Telnet 通过 GetStdHandle()获取到的应该是那个替换了的管道句柄，用这个句柄去获取屏幕信息（GetConsoleScreenBufferInfo），根本不可能成功。因为这个句柄是标识管道的，用它怎样能找到屏幕相关信息？为了证明这一点，我们编写如下代码：

```
if (!CreatePipe(&rh1, &wh1, &sa, 1024)){
    printf("cannot create pipe1\n");
    return ;
}
CONSOLE_SCREEN_BUFFER_INFO info;
if (!GetConsoleScreenBufferInfo(wh1, &info)){
    printf("cannot get screen info\n");
```

```
}
```

将管道句柄 wh1 传递给 GetConsoleScreenBufferInfo，结果证明确实失败。这样很自然地推想出 Telnet 退出的原因：因为没有真正的标准输出句柄（被换成管道句柄了），所以无法正确执行类似 GetConsoleScreenBufferInfo 的函数（它们必须用真正的标准输出句柄），因此程序只有退出。

我们要证明的就是 Telnet 用 GetStdHandle 获得的句柄就是传入的那个管道句柄。思路如下：

建立两个程序，程序 A 用 GetStdHandle 将句柄打印出来，程序 B 调用 CreateProcess 新建 A 的进程，并随意传入两个整数替换标准输入出管道。如果 A 程序运行后打印的就是 B 传入的整数，那么就证明了 GetStdHandle 获取的管道句柄就是 CreateProcess 替换的那些。

工程见“code\第 2 章\testPipeAndGetStdHandle”，包含两个项目：一个是 ctrlshell 扮演程序 B 的角色，一个是 myprg 扮演程序 A 的角色。需要说明的是，ctrlShell 中 3 个被替换的标准管道的数被设定成 3 个特殊的数：123456、888888 和 654321，即

```
si.hStdInput = (HANDLE)123456;
si.hStdOutput = (HANDLE)888888;
si.hStdError = (HANDLE)654321;
```

而 myprg 因为标准输出被替换了，因此将无法打印，只有将 3 个句柄值写入到文件（写入了 C:\result.txt）。见下面代码，经试验，这 3 个句柄确实就是 123456、888888 和 654321。

```
HANDLE ih = GetStdHandle(STD_INPUT_HANDLE);
HANDLE oh = GetStdHandle(STD_OUTPUT_HANDLE);
HANDLE eh = GetStdHandle(STD_ERROR_HANDLE);
FILE * fp = fopen("C:\\result.txt", "w");
char str[10];
itoa((int)ih, str, 10);                    //将标准输入句柄的整数值转换为字符串形式
fwrite(str, strlen(str), 1, fp);           //写入该句柄的字符串值
fwrite("\r\n", 2, 1, fp);                  //写入回车换行
...
```

至此，我们完全证实了为什么 Telnet 在替换标准管道后无法工作的原因了。因为 Telnet 要通过标准句柄获取屏幕信息，而这时的句柄却是管道句柄。管道是非常优雅的抽象，扮演数据传递者角色时，所有管道均等价，不过是一段连接输入、输出的“管子”；扮演其他角色时，管道就不是可以随便等价替换了，如需从管子追溯到连接的东西的相关信息时（用 GetConsoleScreenBufferInfo 获取连接在管道上的屏幕的信息）。

至此，就是笔者之前在这个问题上的全部分析了。在写该部分的时候，笔者觉得应该有更完美的结局，便花了 2 天时间来解决它。这一个过程异常艰苦却也充满乐趣。更多的时候，笔者觉得它已不是技术的问题，而是心理的战争。面对未知的黑箱，你必须冷静，并有着渴求最后光明的热望。下面笔者将这两天不断解决问题的过程写下来，权当一篇日记来看吧。

2.8.4 一动不动的 48 小时

笔者当年想过，如果用钩子技术，应该能解决用句柄操控屏幕信息的问题。从图 2.24 可简单理解钩子的实现原理。

> 钩子技术（hook）：一种通过直接修改执行代码改变程序流程的技术（对于系统支持的钩子，不用直接修改执行码，如 Windows 套接字的 SPI 钩子接口）。钩子有多种实现方法，如：修改函数入口指令，改为 jmp 到用户代码（也有不同实现风格）；修改导入表函数入口地址等。本节用微软提供的开源工程 Detours 实现。

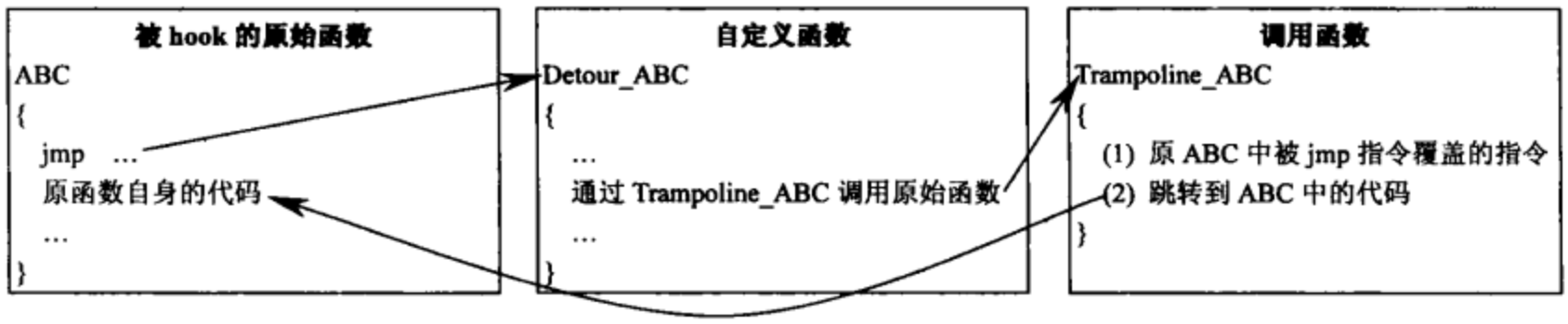


图 2.24

假定函数 ABC()需要被 hook。其被 hook 的效果就是当执行 ABC()时，马上执行我们的函数 Detour_ABC()。然后 Detour_ABC()执行了要做的工作后，通过调用 Trampoline_ABC()（它不是一个规范的函数，没有返回，只包含跳转到 ABC 的相关指令，执行 ABC()的代码，最后从 ABC()中返回）来调用原始的 ABC()。这种技术在没有源代码时能够改变程序逻辑，常常被黑客使用，也可用于正常软件中。Detour_ABC()和 Trampoline_ABC()等函数被编译成一个动态库，然后利用远线程技术注入到被 Hook 的进程。

有了钩子技术，笔者对解决问题做了相关设想。能否钩挂使用标准输入、输出句柄读写的函数，让它们使用管道读写，而非键盘和屏幕？这样也能完成与控制程序的互动。而在使用标准输入、输出句柄完成诸如 GetConsoleScreenBufferInfo 这种必须与真正的输入、输出设备打交道才能成功的函数时，就依然使用真正的标准输入、输出句柄。真正的标准输入已经被替换了，可以拿到它们吗？**（其实可以调用 Telnet 时将管道句柄作为参数传入，不替换标准输入、输出，既然这样做了，就懒得修改了。）**

首先要知道有哪些与 Console 相关的函数，通过微软的帮助查看到相关所有函数如下（帮助中有一个链接，指向了与当前函数为一类的所有函数的名字）：

```
AllocConsole
CreateConsoleScreenBuffer
FillConsoleOutputAttribute
.....
WriteConsoleOutputAttribute
WriteConsoleOutputCharacter
```

然后在图 2.23 的 IDA 的 imports()函数界面中检查 Telnet 用了哪些函数。（结果有近 30 个。）是否都需要被 hook 呢？原则是，不用标准输入、输出作为参数的函数就不用挂接，因为一切问题源于“只有用标准输入、输出句柄才可工作”。**（还剩下 24 个。）**有构造新标准输入、输出的函数吗？如果能构造新的标准输入、输出给那些需要标准输入、输出的系统 API，就能让这些 API 正常工作了。看函数说明，CreateConsoleScreenBuffer()有点像。动手实验，将 CreateConsoleScreenBuffer()返回的句柄传递给 GetConsoleScreenBufferInfo()，确实可以成功。这是标准输出，那标准输入呢？**（找不到，也没有心思一句句看 30 个函数的英文说明。动手试试看思路是否可行才是王道。）**之前也没用过 Detours，只是自己用 Delphi 写过钩子的框架。学习 Detours 还是很快的，在之前的基础上只要将它的示例代码调试一下就可以了。**（剩下就是挂接 24 个函数啊。完全是体力活！）**从微软帮助文档复制函数原型，然后用 Detours 的方式写 Detour_XX 和 Trampoline_XX 函数。代码见“code\第 2 章\hook_Telnet1\hooker\hooker.cpp”。一个多小时就这样在复制、粘贴中过去了。

程序的策略如下：

① 挂接 ReadConsoleW，从管道读入数据，返回给调用者。它们本来通过标准输入，从键盘读入数据。

② 挂接 WriteConsoleW/WriteConsoleA，将调用者输入的数据写入管道。它们本来通过标准输出，打印到屏幕上。

③ 初始化时用 CreateConsoleScreenBuffer()构造代替标准输出的句柄 gMockConsoleOutputHandle，将其他使用标准输出作为参数的函数用 gMockConsoleOutputHandle 作为其标准输出句柄参数。比如（见黑体部分）：

```
DETOUR_TRAMPOLINE(BOOL WINAPI Trampoline_WriteConsoleOutputW(
                                    HANDLE hConsoleOutput, CONST CHAR_INFO *lpBuffer,
                                    COORD dwBufferSize, COORD dwBufferCoord,
                                    PSMALL_RECT lpWriteRegion), WriteConsoleOutputW);
static BOOL WINAPI Detour_WriteConsoleOutputW( HANDLE hConsoleOutput,
                                     CONST CHAR_INFO *lpBuffer, COORD dwBufferSize,
                                     COORD dwBufferCoord,PSMALL_RECT lpWriteRegion) {
   BOOL ret = Trampoline_WriteConsoleOutputW(gMockConsoleOutputHandle, lpBuffer,
                                                    dwBufferSize, dwBufferCoord, lpWriteRegion);
   return ret;
}
```

为了让 Telnet 使用 gMockConsoleOutputHandle 作为输入，必须在 Telnet 开始执行 main()函数前先挂接并构造 gMockConsoleOutputHandle 句柄。所以调用 CreateProcess()启动 Telnet 时要用悬挂启动方式。这时 Telnet 虽然加载到内存，但不会执行。当钩子 DLL 注入到 Telnet 并在其 dllmain 中完成所有初始化工作后，用 ResumeThread 恢复进程的主线程，即执行 main()的线程，运行即可。

④ 标准输入怎么办？没有找到替代的方法。这个简单，“骗”。剩下的函数都返回 TRUE。不管是否能成功，第一目标是让程序运行起来，而不是像 2.8.2 节，启动后又退出。只要返回成功，至少启动时程序可能不会退出。

（**在晚饭后，终于可以试验一下了。**）黑屏，什么也没有！出什么问题了？在进程列表中看，Telnet 就在那里。成功了，只是什么也看不见。似乎问题在于两个 console 的程序共享了一个屏幕，做法肯定也有些问题，这样就不好分析了，程序见“code\第 2 章\hook_Telnet1\ctrlshell_no_work”。用窗体程序来替代 console。还是用 Delphi，程序见“code\第 2 章\hook_Telnet1\ctrlTelnet”。将 C 语言版的代码移植过来。（**天已经黑了，这个冬夜太冷。**）运行，在界面中看见了 Telnet 启动时的提示信息，可开始是乱码，然后似乎还有一些可识别的”Microsoft...”。应该是编码问题。而且，似乎显示信息的长度远不够。在调试情况下观察内存窗体，可看见每个字符都由 2 字节组成。如'M'由其 ASIIC 码值和'0'组成。（**好像是 Unicode 编码。**）而且挂接的函数 WriteConsoleW 是 W 结尾，在 Windows 的惯例中它是 Unicode 字符集的函数。看来我们收到了 WriteConsoleW 输出的信息。信息长度不够，通过调试、分析函数原型，因为函数参数中输出长度是字符长度，不是字节长度。笔者当成了字节长度，自然长度几乎减半。

调试一下钩子。钩子是一个被注入到 Telnet 进程中的 DLL，用 printf()无法打印。Telnet 的标准输入、输出被替换了，printf()会输出到 Delphi 书写的程序：

调试无法使用界面的代码

本来 printf()是调试的最原始也是最可依赖的方法，但有些情况下做不到，如图形界面（信息又不能输出到界面上）、钩子 DLL 被注入到其他进程中。解决办法如下：① 用 AllocConsole 为图形程序创建一个控制台，然后从中能看到输出消息；② 用 OutputDebugString 输出消息，后跟'\n'，否则不会输出，用图形工具 DebugView 可将消息展示在窗体中；③ 写一个 Log 函数，将消息写入文件。

笔者使用第③种方法。通过调试发现，Telnet 的欢迎信息确实是 Unicode 版本的 WriteConsoleW 输出的。那么，就是上网查找转换函数资料。终于，能看见非乱码的输出了，但输出信息依然不完整，少了一点。仔细分析，原来通过 WideCharToMultiByte 计算的字符串长度还包含了结尾字符'0'的长度。（**网上的代码是错误的，这个没计算进去的结尾字符导致信息不完整。兴奋！**）在界面中输入命令“d”回车，就看到了正常的输出信息。（**已经晚上 12 点了，想在睡前更完美一点。**）在 Windows 服务面板中启动 Telnet 服务器，在自己的程序中输入“o 127.0.0.1”连接服务器。看到了部分信息，即连接 127.0.0.1，然后死机了！（**撑不住了，只有睡了，遗憾的夜晚。**）

（**早上起来继续奋战。**）初看，无法分析，没有 Telnet 的源代码，怎样调试？首先，要看到底是 Telnet 连接并输出了消息而程序没正确执行，还是其他什么原因。因此，笔者想看看输出信息的两个函数 WriteConsoleA()和 WriteConsoleW()在这个方案和正常 Telnet 两个版本的执行中有何不同。为了防止其他因素的干扰，准备直接调试这两个版本。这时用汇编调试。（2.1 节的图 2.5(a)中介绍了用 Delphi 去 attach 正在执行进程、分析调试的方法。）现在要在 WriteConsoleA()和 WriteConsoleW()的入口处设置断点，就需要知道这两个函数的地址。

查找系统 API 入库

获取当下 Windows 系统的某个 API 的入口地址

由第 1 章的重定位知识，Windows 的系统 DLL 在所有进程中将加载在同一位置，其中的函数也会加载到同一地址。调用某 API 函数，然后用反汇编跟踪到该函数入口，记录下该地址即可。当要在其他进程中调试时，只需跳到该地址，设置断点即可。在 Delphi 中可以直接用 module 窗体查看进程加载的 DLL，并找到 DLL 中相关函数的地址。可以参见第 7 章的图 7.15。

通过比较证实，确实在笔者的钩子版本中 WriteConsoleW()被少执行了一次。为什么？这一定与笔者的方案有关：很多被钩住的函数直接返回了 TRUE，也许正常执行要依赖真正正确地执行了这些函数，而不是用返回值“骗”调用代码说成功了。

于是（**新的体力活开始了**）修改每个 Detour_XX 函数，让它们调用对应 Trampline_XX 的版本，并在日志中记录下是否成功调用。结果 WriteConsoleInput()函数执行时失败。但是帮助中并没有有价值的信息。（**都想放弃了，可是多年的积累给了笔者在完全无头绪时的冷静和信念。这时需要猜想，否则完全无法进展。**）也许程序是在某个时刻调用了该函数，将信息放入 Console 的 Input 队列（所以叫 Write Console Input），又用某些函数从 Input 队列提取该信息，才能进行后续工作。在 WriteConsoleInput 处设置断点并调试发现，正常版本的确是在执行该函数后再次调用了 WriteConsoleW()来输出信息。之前的方案根本没有找到替代的 Input 句柄，自然无法成功。那么，解决思路有了——提供可以正常操作的 Input 句柄。

粗看，之前的 30 个函数中似乎没有。（**不要简单放弃，要从 Windows 的角度思考。从构建者的角度思考，这时能展现“猜测、实证和构建”这一习惯的威力了！**）既然能替换标准输入、输出，万一又需要怎么办？这 30 个函数中，之前替代输出的句柄用 CreateConsoleScreenBuffer 生成了一个。从日志记录来看，凡是涉及输出句柄的函数操作都成功了。输出能造出新句柄，输入可以吗？这时需要仔细读某些文档（全部读太慢）。选择哪些函数细读？要构造新的输入句柄就是因为用 SetStdHandle 替换了旧句柄，与句柄操作相关的函数描述中也许有相关信息。GetStdHandle 中没有，再看 SetStdHandle，在最后有这样一段话：

> if the standard handles have been redirected, you can specify the **CONIN$** value in a call to the CreateFile function to get a handle of a console's input buffer. Similarly, you can specify the **CONOUT$** value to get a handle of the console's active screen buffer.

原来可以用 CreateFile 来构造。它正是所有抽象设备的构造入口，标准输入出也是一个抽象的

文件。现在那一堆包含 Console 的函数的作用还不清晰。微软文档中也没有例子。那么，不急于修改代码，先构想一下其作用模型。（**否则还是瞎子摸象。**）

所谓输入（input）队列，应该是将键盘的输入转换成一种描述信息放入队列中。而系统通过提供相关读接口，就能将它们读出来。WriteConsoleInput 的关键参数如下：

```
typedef struct_INPUT_RECORD {
    WORD EventType;
    union {
        KEY_EVENT_RECORD KeyEvent;
        MOUSE_EVENT_RECORD MouseEvent;
        WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
        MENU_EVENT_RECORD MenuEvent;
        FOCUS_EVENT_RECORD FocusEvent;
    } Event;
} INPUT_RECORD;
```

其中的 KEY_EVENT_RECORD 正是描述键盘事件。猜想一下，WriteConsoleInput 写入的数据应该能被 ReadConsoleInput、ReadConsole 甚至 ReadFile 读取。跟踪 VC 的 getc()发现，该函数通过 ReadFile 读取了键盘。ReadFile 可读取任何设备，Input 队列属于 console 设备，那么 ReadFile 应该能读取 console 设备的 Input 队列。实证结果是，CreateFile 构造的句柄用 ReadConsoleInput 可读，用 WriteConsoleInput 却失败。只好再次看那一堆函数，也许有什么激活句柄的函数，如 SetConsoleActiveScreenBuffer 就是将输出队列激活。输入队列的激活呢？既然 SetStdHandle 可以设定标准输入、输出，就用它来试验。结果成功。经过多个试验发现，所有的读都是阻塞式的，没有它关心的数据就不返回，ReadFile 也可读 WriteConsoleInput 写入的数据。终于可以改造代码了：① 在初始化部分构造新的标准输入和输出句柄；② 原来直接返回 TRUE 的钩子函数改成要调用真正的原函数（不过这在之前分析问题时已经改造）。

紧张地输入“o 127.0.0.1”。终于有了返回信息了，但仍然是乱码。利用在每个被钩挂的函数开始输出的“某某函数 begin”这样的信息，发现这次是调用非 Unicode 版的 WriteConsoleA 输出时出错的。当时的处理策略是，钩子端没有进行 Unicode 到多字符集的转换，而是在 Delphi 写的控制程序中做的。控制程序这次接收到 WriteConsoleA 输出的多字符集串并非 Unicode 版本，自然出错。控制程序无法区分两者，只有将转换工作放入钩子函数 Detour_WriteConsoleW 中去做，让控制程序接收到的都是统一的多字符集。修改代码，成功！

（**准备收兵了。**）但鉴于之前每多输入一次命令就多一次问题的前车之鉴，还是输入了“dir”命令，回车，结果什么返回都没有。（**崩溃，似乎没有一次输入的命令能正确执行。**）检查钩子端，通过日志发现，当控制程序用管道输入“dir”命令后，挂接的函数没有一个执行。也就是说，输入命令，Telnet 没有读取，或者说它通过其他渠道读取的，而这个不知道的渠道没有被挂接。

第一步是要找到到底从哪里读入输入。这似乎是一个难以解答的问题。用反汇编分析完全是一个遥遥无期的工作。就算有源代码也不见得能找到，所以采用 API 断点法分析。

快速定位关心部分的代码——系统 API 断点法

如果关心的功能调用了系统 API，那么用前面介绍的方法获取到该 API 的入口地址后，就可通过在其入口处设置断点进行调试。在断点处观察栈当前的值（即返回地址）。然后在这个地址处设置断点，继续执行。同时在 IDA 中找到该地址相关的代码。方法是通过 IDA 的 imports 表找到关心的系统 API，然后查看引用它的地方，根据 IDA 给出的地址很容易换算出实际运行中的地址（一般就是原值，因为不会重定位）。调试，配合 IDA 就很容易找出执行序列。

现在是不知道哪里接收键盘输入。输入命令后的动作是该命令通过网络被发送到服务器。Telnet使用 TCP，应该调用 send()函数发送。用 IDA 查看 Telnet 的导入表，确实有该函数。接下来在该函数入口处设置断点，然后启动 Telnet 并 attach 进去，运行，调试。对于 dir 命令，并非回车后才调用 send()函数，而是 d 键按下去还没弹起来就进入断点了。也许，Telnet 遵循某些约定，将客户端的输入马上送给服务器（有些输入需要立刻生效，所以马上就发送）。**（有这个表现也正常。）**假定程序用函数 A 接收了该键盘输入，最后调用到 send()，剩下的就是回溯到这个函数 A。**（这是一个体力活，如果结合 IDA 的分析可以减轻一些工作量。）**为了能边跟踪边分析，利用 IDA 的调试能力即可。可是笔者当时找不到在某个指定代码地址打断点的功能（后来发现只要在 Add breakpoint 中输入地址即可），因此只有先用其他调试器找到从 send()返回后的地址，再在 IDA 中找到该地址对应的代码并设置断点。再次从 IDA 中直接启动 Telnet 调试。终于有结果了，从 ReadConsoleInputA 接收到键盘输入。也就是说，当连接上 Telnet 服务器后，Telnet 并没有从被替换的输入管道直接读数据，而是从 Console 的输入队列读数据。而数据此时在替换后的输入管道中，Console 的输入队列并没有数据，所以无法取到命令并发送给服务器。**（问题回到了之前实验过的 WriteConsoleInput()等函数。）**已知 WriteConsoleInput()写入队列的信息可用 ReadFile、ReadConsoleInputA 和 ReadConsole 读取，这个输入/输出模式的好处是，真正的键盘输入转换成 INPUT_RECORD 结构，可放入队列，还可通过代码直接写入队列，模拟出键盘输入。这样就能完成一些复杂的功能，且将输入/输出从具体设备的依赖转换成抽象的模型。那么，既然连接服务器后是从 ReadConsoleInputA 读入的键盘输入，解决方案自然出来了，而且非常简单，工程见“code\第 2 章\hook_Telnet_ok”：

① 在 Telnet 中构造一个线程，从管道读入数据，然后用 WriteConsoleInputA 放入输入队列，Telnet 程序就能通过标准的函数读出它们，包括 ReadConsoleInputA。

② 挂接 WriteConsoleW/WriteConsoleA 的函数逻辑不变，依然从管道写入输出数据。而 ReadConsoleW 不需挂接，因为我们不通过它读管道，所以只需挂接 WriteConsoleW 和 WriteConsoleA 函数（其实挂接 WriteConsoleW 即可，因为 WriteConsoleA 将调用 WriteConsoleW。似乎 Windows 7 下不是这样）。程序异常简单。

③ 初始化部分与上一版本一致。

这一版本代码最少，但是付出了 14 小时的艰辛。遇到如下困难：

首先，连欢迎信息都收不到了。ReadConsoleW 根本收不到数据，甚至笔者都以为之前做实验得出的结论是错误的。最后定位发现，ReadConsole 需要接收到回车符才返回**（微软说明中没提到！）**，而之前实验时笔者恰巧用了回车符（字节值 13，C 语言的'\r'）。可项目代码后来用的是'\n'，因为 ReadConsole 被挂接后没有调用原函数所以并没阻塞，该问题就未暴露。修改后，该版本确实在连接服务器后能将命令送过去，但返回“该命令为非法命令”的提示。**（这就是苦难的开始。为什么会出错？没有半点思路，最笨的办法是对比编写的程序与直接运行 Telnet 在发送命令上到底有何区别。）**祭起调试法宝，在 send 入口设置断点，然后根据函数调用模型，在栈上找到属于 buf 的指针，再到内存窗体观察该地址中要发送的内容（参考 2.6 节），发现确实发出的不是希望的字符。

（笔者当时使用 WriteConsoleInput 模拟的键盘输入，只模拟了按下键盘，没有模拟弹起。难道是这个原因？虽然觉得不太像，但马上修改代码，可是无效。其后的工作非常枯燥。）

一边调试，一边用 IDA 分析两个版本的程序从收到字符到调用 send 之间到底在哪里不一样。先在 IDA 中分析正常版本的执行轨迹。按 F9 键，启动 Telnet，然后输入“o 127.0.0.1”登录。登录成功后，再在 send 入口处设置断点（因为不想关注之前连接命令导致调用 send 的逻辑，笔者的 Windwos XP 系统上 send 的入口地址是 0x71a2428a）。在 IDA pro 菜单中选择 Debugger →

Breakpoints → Add Breakpoint，在弹出的窗体中输入“71a2428a”，回车，在 Telnet 中输入字符 'd'。程序在 send 入口停下来。连续按 F8 键，单步执行，直到遇到 ret 指令返回。这时看到了调用 send 的代码。该函数比较小，基本就是直接调用了 send 函数。笔者将它命名为 willsend2，其代码见 DM2-22。从最后一行 send 的压栈顺序和 send 函数的原型 int send (SOCKET s, const char FAR * buf, int len, int flags)可得出，第 4 行压栈 flag，第 5 行压栈 len，第 6 行压栈 buf。willSend2 的第 2 个参数是 buf，第 3 个参数为发送长度 len。既然 send 发送的字符是错误的，就要观察这个参数。从 DM2-22 分析，该函数没有修改 buf 的内容而是直接发送，因此要看调用 willSend2 的代码。

DM2-22

```
    .text:01004652   ; int __stdcall willsend2(int,char *buf,int len)
    .text:01004652   arg_0= dword ptr 8
    .text:01004652   buf= dword ptr 0Ch
    .text:01004652   len= dword ptr 10h
1   .text:01004652   mov  edi, edi          //函数开始部分
2   .text:01004654   push ebp
3   .text:01004655   mov  ebp, esp
    ...
4   .text:0100466A   push 0                                  ; flags
5   .text:0100466C   push [ebp+len]                          ; len
6   .text:0100466F   push [ebp+buf]                          ; buf
7   .text:01004672   push dword ptr [esi+104h]               ; s
8   .text:01004678   call ds:send    .
```

继续按 F8 键，直到遇到 ret 指令返回。这时就来到了自命名的函数 willSend1。（IDA 支持用户将缺省的以地址命名的函数名修改成用户想要的名字。）这时需要分析 willSend1 执行过程看看它当中在调用 willSend2 之前为什么修改了发送的内容。willsend1 的代码见 DM2-23(a)。经过整理两个版本的执行路径，获得了 DM2-23。DM2-23(a)的第 6 行中，两个版本走了不同的路径：正常版为 DM2-23(b)，错误版为 DM2-23(c)。下面来分析错误版。

倒推法是识别参数含义的关键，由此才能分析代码逻辑。DM2-23(c)的第 18 行调用了 willsend2，从参数的压栈规则看，第 14 行压栈发送长度 1（当前输入为 d，因此发送 1 字节），第 15 行压栈了 eax，发送缓存的地址。第 12 行将地址 ebp+inputC+3 赋值给了 eax。因此，ebp+inputC+3 即发送缓存地址，第 11 行为该缓存赋值。（第 10 行将 al 的内容减 40h，然后第 11 行赋值给了 ebp+inputC+3 指向的内存。）通过调试发现，在第 10 行前，al 就是输入的字符“d”，第 10 行修改了，减了 40h。分析代码也可得到结论，第 10 行执行前的 al 就是要发送的字符。因为在 DM2-23(c)中追溯可见，al 是在第 3 行被赋值的，最终通过第 1 行和第 3 行将 bl 的值赋值给了 al。也就是说，我们需要追溯 bl 的赋值情况。分析整个 willsend1 函数，可知 DM2-23(a)第 1 行将输入字符由输入参数 ebp+inputC 赋值给 bl 后就没有变化，因此 bl 中就是要发送的字节，DM2-23(c)第 10 行的 al 存的是该字节的值。

要发送的字节为什么会在 DM2-23(c)的第 10 行被减了 40h 然后发送呢？追根溯源，是 DM2-23(a)中第 6 行的转跳，其转跳的条件就是 DM2-23(a)的第 5 行：cl 和 0ch 进行 and 的结果不为 0，cl 就在 DM2-23(a)中的第 3 行 mov cl, [ebp+10h]被赋值。而地址 ebp+10h 是 willsend1 参数的地址。IDA 给出了该函数的原型：

```
int __stdcall willsend1(COORD dwSize, int inputC, __int16);
```

该函数入口代码：

```
.text:0100977c      mov  edi, edi
.text:0100977e      push ebp
.text:0100977f      mov  ebp, esp
```

说明它是标准的用 EBP 寻址参数和局部变量的。从帮助文档看，COORD 是 4 字节，inputC 是 4 字节，那么第 3 个参数的地址应该是 ebp+8+4+4=ebp+16=ebp+10h。DM2-23(a)第 3 行就是将第 3 个参数赋值给 cl。错误版本和正确版本的差异就是第 3 个参数的值不同！

DM2-23(a)

```
1  .text:01009789    mov   bl, byte ptr [ebp+inputC]
2  .text:0100978c    cmp   bl, 3
3  .text:0100978f    mov   cl, [ebp+10h]
   ...
4  .text:010097c9    jz    loc_1009A4B
5  .text:010097cf    test  cl, 0ch
6  .text:010097d2    jz    short loc_100981a
```

DM2-23(b)（正常版本）

```
   .text:0100981a  ...
1  .text:01009aa0    loc_1009aa0:
2  .text:01009aa0          push   [ebp+inputC]
3  .text:01009aa3          push   edi
4  .text:01009aa4
5  .text:01009aa4          push   esi
6  .text:01009aa5          call   willsend2
```

DM2-23(c)（错误版本）

```
1  .text:010097d5    mov   [ebp+arg_B], bl
2  .text:010097d8    call  sub_10077ca
3  .text:010097dd    mov   al, [ebp+arg_B]
4  .text:010097e0    cmp   al, 20h
5  .text:010097e2    jnz   short loc_010097ea
   ...
6  .text:010097ea    cmp   al, 40h
7  .text:010097ec    jb    short loc_10097f2
8  .text:010097ee    cmp   al, 5dh
9  .text:010097f0    jbe   short loc_1009805
   ...
10 .text:01009805    sub   al, 40h
11 .text:01009807    mov   byte ptr [ebp+inputC+3], al
12 .text:0100980a    lea   eax, [ebp+inputC+3]
13 .text:0100980d    jmp   short loc_01009812
   ...
14 .text:01009812    push  1
15 .text:01009814    push  eax
16 .text:01009815    jmp   loc_01009aa4
.  ...
17 .text:01009aa4    push  esi
18 .text:01009aa5    call  willsend2
```

al 中为要发送的字节，第 10～11 行将其减 40h，并赋值给发送缓存，地址为 ebp + input + 3

第 14～18 行调用 willsend2。其中，eax 是发送缓存的地址。第 12 行将 ebp+inputC+3 赋给 eax，因此将地址 ebp+inputC+3 中的字节发送出去

那么，需要继续追踪调用 willsend1 的调用代码，为什么传入第 3 个参数值是不同的。继续跟踪，返回到调用 willsend1 的代码，见 DM2-24。

DM2-24

```
1  .text:0100a695    mov   eax, [ebp-24h]
2  .text:0100a698    test  ah, 1
3  .text:0100a69B    push  eax
4  .text:0100a69C    jnz   short loc_100a6a3
5  .text:0100a69E    cmp   [ebp-26h], bl
6  .text:0100a6a1    jnz   short loc_100a6ae
   ...
7  .text:0100a6ae    push  dword ptr [ebp-26h]
8.text:0100A6B1      push  esi
9.text:0100A6B2      call  willsend1
```

willsend1 的第 3 个参数压栈应该是从第 9 行往前的第 3 条 push 指令，所以参数值就是第 3 行压栈的 eax。在第 1 行将地址 ebp–24h 指向的值赋值给了 eax。为了快速找到修改 ebp–24h 地址的语句，开始在 IDA 中仔细查看。要不由 mov [ebp–24h], ...类似的指令直接修改了该地址，要不就是该地址作为参数传递给了某个函数，然后在该函数中被赋值。但是调用 willsend1 的函数的全部代码中没有任何代码符合以上假设。（**记得当时看到一个地址 ebp–30h，心里想如果是它就好了，也没太在意。**）由于静态分析不通，就想了一个笨办法。既然 ebp–24h 中存的值在两个版本中不同，那么肯定在哪个语句修改了它。于是在调用 willsend1 的函数中不断用 F8 键单步执行，然后关注 ebp–24h 内存的变化。结果在执行 DM2-25 的第 10 行后，即执行了 ReadConsoleInputA 后，内存变化了。但这段代码似乎并没涉及地址 ebp–24h。

DM2-25

```
1   .text:0100a29b                  push  1                          ; nLength
2   .text:0100a29d                  lea   eax, [ebp+buf]
3   .text:0100a2a0                  push  eax                        ; lpBuffer
4   .text:0100a2a1                  push  dword ptr [esi+7bch]       ; hConsoleInput
5   .text:0100a2a7                  jnz   short loc_100a2b1
6   .text:0100a2a9                  call  ds:ReadConsoleInputW
7   .text:0100a2af                  jmp   short loc_100a2b7          ; hConsoleInput
8   .text:0100a2b1
9   .text:0100a2b1  loc_100A2B1:
10  .text:0100A2B1                  call  ds:ReadConsoleInputA
```

先分析该函数的原型：

```
BOOL ReadConsoleInput(HANDLE hConsoleInput, PINPUT_RECORD lpBuffer,
                      DWORD nLength, LPDWORD lpNumberOfEventsRead);
```

传入第 2 个指针参数 lpBuffer，返回时将读出一个输入队列中的数据，填充到 lpBuffer 指向的 INPUT_RECORD 记录。DM2-25 中，第 3 行将 eax 的值压入作为 lpBuffer。第 2 行将 ebp+buf 赋值给 eax，这就说明地址 ebp+buf 是 INPUT_RECORD 记录的地址。本段代码中关于 buf 的定义为 buf= byte ptr-34h，则 INPUT_RECORD 记录的地址就是 ebp–34h。既然执行 ReadConsoleInput 后 ebp–24h 的内容变化了，那么合理的解释就是 ebp–24h 是 INPUT_RECORD 记录的一个成员变量的地址。（可能吗？）ebp–34h 为结构体首址，其成员变量的地址应该大于该地址。而 ebp–24h 大于 ebp–34h，所以 ebp–24h 有可能是结构体成员的地址。这两个地址相差 10h，即 16 字节。于是需要查看结构的哪个成员偏移 16 字节。前面给出了 INPUT_RECORD 的定义：

```
typedef struct _INPUT_RECORD {
    WORD EventType;
    union {
        KEY_EVENT_RECORD KeyEvent;
        ...
    } Event;
...
```

由于是键盘事件，其 union 成员 Event 就是结构体_KEY_EVENT_RECORD，定义如下：

```
typedef struct _KEY_EVENT_RECORD {
    BOOL bKeyDown;
    WORD wRepeatCount;
    WORD wVirtualKeyCode;
    WORD wVirtualScanCode;
    union {
        WCHAR UnicodeChar;
        CHAR  AsciiChar;
    } uChar;
    DWORD dwControlKeyState;
} KEY_EVENT_RECORD;
```

应用在 1.5 节中对齐的知识，可计算出正好在离首部偏移 10h 的地方就是 dwControlKeyState（可写代码验证，WCHAR 是 short，BOOL 是 int，因此 EventType 和 bKeyDown 占 8 字节，剩余字段占 8 字节）。该字段表示是否同时按下了大写键、Shift 键等控制键。笔者用 WriteConsoleInput 模拟输入时根本就没有设定该字段，因为是 Debug 版，所有局部变量的初值被设定为 cch。那么，这个

字段的值就是 cch。也就是说，当 Telnet 用 ReadConsoleInputA 读取 INPUT_RECORD 时，因为键盘事件的 dwControlKeyState 值的不同（笔者的版本是 cccccccch，正确版本是 00400000h），导致了处理的不同。DM2-24 中，第 1 行的 ebp–24h 地址就是结构体中 dwControlKeyState 的地址，将 dwControlKeyState 赋值给了 eax，并在之后压栈作为第 3 个参数传递给了 willsend1。这样导致 willsend1 因该参数不同执行了不同路径。因为错误版的值是 cch，相当于表示某个控制键同时被按下，自然发出去的字符就不对了。

这次应该找到原因了，将 dwControlKeyState 设定为 0 即可（设定为 0，则 DM2-23a 的第 6 行将转向 DM2-23b 的正常分支。因为此时 cl 的值就是 dwControlKeyState）。成功了。（**此时已是夜里 12 点半了。**）一条语句“dwControlKeyState=0;”等于 10 个小时。回头看这个版本的代码确实简单多了，笔者却付出了更大的代价，总共花了 48 小时。

最后一个问题 dwControlKeyState 的解决值得反思。分析两个版本执行过程到底差异在哪里花费了很长时间，但是如果冷静一点，先分析和猜想也许会快得多。应该从两个版本差异去寻找分析点。因为笔者并没有修改 Telnet 的代码，所以差异就是对键盘输入，正常版本是由系统产生的 INPUT_RECOED 记录放入输入队列，而笔者的版本是自己调用 WriteConsoleInputA 产生的，因此这个输入记录可能产生了差异。同时，认真阅读 WriteConsoleInputA 的参数 INPUT_RECOED 应该能发现 dwControlKeyState 的问题。

本节部分知识远远超出了一般读者的基础，但作为解决真实问题的忠实记录，涉及许多思考、工具、技巧和基础知识，几乎一步一错的过程很有启发意义，也展示了基础和底层知识的力量。读者可之后再深入阅读和调试配套代码。

2.9 异常世界历险记

之前的例子展示了反汇编和调试分析在纠错和特殊性构建程序方面的作用。本节将展示在学习未知事物方面的重要作用。探索未知机制也是我们增强逆向功力的重要途径。只要学过 C++、Java、C#或 Delphi 等都知道 try-catch 语句，就是操作系统的异常处理机制。异常处理机制对逆向、安全和部分底层代码（如 coroutine 库）都有关键作用。下面将用第一人称叙事的方式展现这种学习的本质，希望能启发大家。

2.9.1 学习基础概念

前面学了 C 语言的反汇编，特别是函数的调用和返回过程，似乎把握到了字节的运行轨迹。我喜欢这个感觉，记得有书上说“顶级的程序员，程序的字节在头脑中如何流动他都知道”。当我审视 try-catch 语句时，这个流动的字节发生了凝滞。哪里出问题了？仔细分析每个环节。函数返回时通过栈保存返回地址，当执行了 ret 指令，就将该地址弹入 EIP 寄存器，程序就返回了。可异常发生时呢？函数 f1 调用 f2，f2 调用 f3，异常发生在 f3 中，而 try 语句在 f1 中，程序怎样就知道 f1 中 catch 语句的地址并返回？try 后面是有奥秘的。先在网上搜索，输入“异常处理”、“机制”、“try”等关键字，如果信息太杂，再输入“Windows”。搜索到不少文章，很多都参考了“A Crash Course on the Depths of Win32Structured Exception Handling”。参考这篇英文，总算了解了大致原理。

图 2.25 中有 3 个 8 字节长异常处理块。第一字段是 next，目的是将块串起来形成链表。第二字段为异常处理函数入口地址。当发生异常时，操作系统将捕获异常，从 fs:[0]指向的内存获取第一

个处理块首址。然后调用块第 2 字段指向的异常处理函数，根据函数返回值决定其下一步动作，如是否继续调用下一个异常处理块处理异常（如当前块不处理则下一块将处理）。每条 try 语句会生产一个异常处理块，注册到操作系统（插入到异常处理链首）。当整个 try-catch 语句结束时，语句负责从异常处理链中注销刚才注册的块。

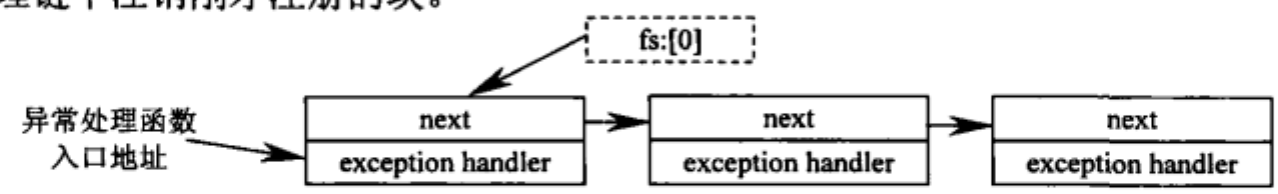


图 2.25

异常处理函数的原型如下：

```
EXCEPTION_DISPOSITION __stdcall ErrorHandler(PEXCEPTION_RECORD PtrExceptionRecord,
                    void * PtrSEH, PCONTEXT PtrContext, void * PtrDispatcherContext);
```

其参数暂时不深究，来实验第一个程序。**（我的习惯是，资料看懂了自己构建一个，不抄袭。这样才能真正验证是否理解，往往 1 字节的错误都无法成功，而简单运行别人的代码很可能不知道这些细节的存在。）**工程见“code\第 2 章\structexception\simple”。在 VS 2008 下，代码在 Debug 版方可运行（原因见 2.9.4 节）。图 2.26 和 DM2-26 描述了插入异常处理块和弹出块的过程，其中虚线指向的是完成对应指令后的内存映像（因为异常处理块在栈上，所以地址为上高下低）。如果输入 0 后回车，将执行第 12 行，此时 eax 为 0，会对地址 0 赋值 0，将产生异常。操作系统从异常处理链的第一个处理块的第二字段获取到异常处理函数入口，程序将转入函数 ErrorHandler 运行。

异常处理函数见 DM2-27。因为执行 DM2-26 的 mov [eax], 0 时出错，所以通过 DM2-27 第 2 行，将 eax 指向一个有效地址（&gi），返回值为 ExceptionContinueExecution，表示继续从出错的 mov 指令执行。此时因为 eax 已经指向了 gi 地址，所以成功执行。注意，ErrorHandler 的第 3 个参数包含了很多寄存器字段，我们可在 ErrorHandler 中设定这些寄存器字段，当该函数返回后，操作系统将这些寄存器值设定到寄存器中。

2.9.2 如何返回

我不太满意这个例子，因为它没有展示出如何返回到 try-catch 中的 catch 块执行（例子是继续从出错指令执行）。查阅了几个网上教程，均没有展示如何返回到 catch 块。分析 DM2-27，通过修改 PtrContext 的 eax，继续执行时就用修改后的 eax。如果该结构体有 EIP 寄存器，并修改它，是否就能转到希望的 catch 块执行呢？查看该结构体的定义，确实有 eip 成员变量。马上尝试。工程见“code\第 2 章\structexception\simpleReturnAddr”。修改非常简单，在 DM2-26 的开始增加一条指令，将全局变量 normalAddr 赋值为 Normal 代码块地址：

```
_asm mov normalAddr, offset Normal
```

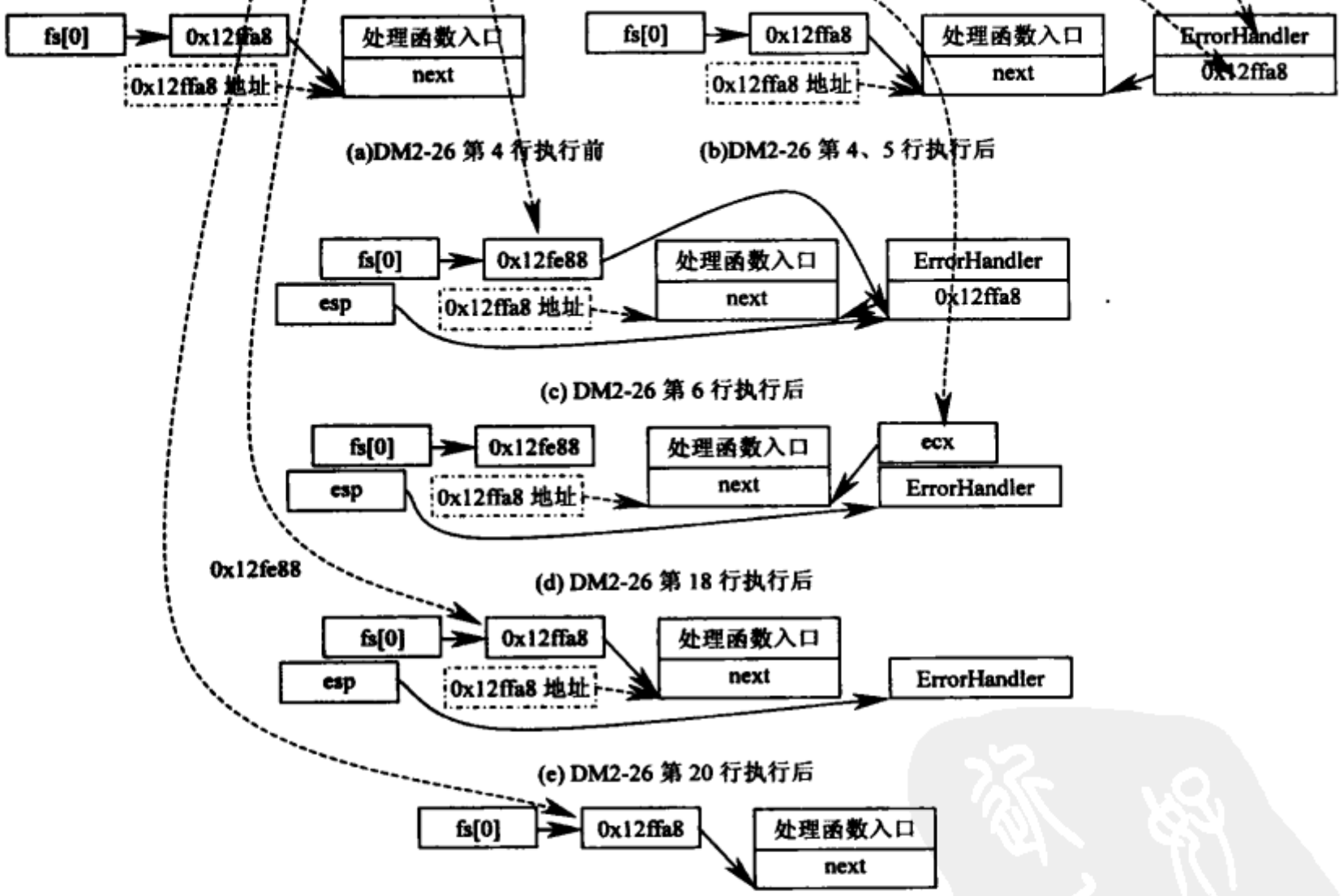
在 DM2-27 中将第 2 行替换为修改 eip，让其指向 Normal 代码块：

```
PtrContext->Eip = normalAddr;
```

这个实现有缺陷，对 eip 的赋值是固定在代码中。如果为每个 try 语句生成的代码除了正常返回地址值，其他一模一样，这也太浪费了吧？能否将正常返回地址传递给异常处理函数呢？可是这个函数似乎没有参数可传入用户自定义的数据。这时我很想看看编译器是怎样解决这个问题的。编写 Delphi 代码如下（except 相当于 catch，习惯上称 except 包括的代码为 catch 代码块）：

DM2-26

```
1   void main(){
2       __asm{                          //在栈上生成异常处理节点**************
3           xor     eax, eax
4           push    ErrorHandler;       //设定异常块第二字段，异常处理函数入口地址
5           push    fs:[eax]            //设定异常块 next 字段，即前一块首址(在 fs:[0]中)
6           mov     fs:[eax], esp       //修改异常链首址,当前块首址即 esp，赋于 fs:[0]
7       };
        ...
8       if (input != '0')
9           printf("ok\n");
10      else{
11          __asm{
12              xor     eax, eax
13              mov     [eax], 0;       //写空地址，触发异常!
14          }
15      }
16      Normal:
17          __asm{                      //销毁 8 字节异常块
18              pop     ecx             //将 next 字段值，即前一节点地址弹入 ecx
19              xor     eax , eax
20              mov     fs:[eax], ecx   //将前一节点的地址作为首节点地址存入 fs:[0]
21              pop     ecx             //随意弹出异常块第 2 字段，最终栈平衡
22          };
23  }
```



(a)DM2-26 第 4 行执行前

(b)DM2-26 第 4、5 行执行后

(c) DM2-26 第 6 行执行后

(d) DM2-26 第 18 行执行后

(e) DM2-26 第 20 行执行后

(f) DM2-26 第 21 行执行后

图 2.26

DM2-27

```
EXCEPTION_DISPOSITION __stdcall ErrorHandler(PEXCEPTION_RECORD PtrExceptionRecord,
                                    void * PtrSEH, PCONTEXT PtrContext, void * PtrDispatcherContext) {
1    printf("hello error\n");
2    PtrContext->Eax = (DWORD)&gi;
3    return ExceptionContinueExecution;                    //让程序继续运行，不停止
}
```

```
try
    PIntger(0)^:= 1;                //相当于*((int *)0) = 1；即往 0 地址赋值 1，将引发异常
except
    gi := 2;
end;
```

将其反汇编，见图 2.27。

```
Unit1.pas.30: try
0044DBB2 33C0                xor eax,eax
0044DBB4 55                  push ebp
0044DBB5 68D2DB4400          push $0044dbd2
0044DBBA 64FF30              push dword ptr fs:[eax]
0044DBBD 648920              mov fs:[eax],esp
Unit1.pas.31: Pinteger(0)^ := 1;
0044DBC0 33C0                xor eax,eax
0044DBC2 C70001000000        mov [eax],$00000001
0044DBC8 33C0                xor eax,eax
0044DBCA 5A                  pop edx
0044DBCB 59                  pop ecx
0044DBCC 59                  pop ecx
0044DBCD 648910              mov fs:[eax],edx
0044DBD0 EB14                jmp +$14
0044DBD2 E9955DFBFF          jmp @HandleAnyException
Unit1.pas.33: gi := 2;
0044DBD7 C70524FC44000200 mov [gi],$00000002
0044DBE1 E8EE60FBFF          call @DoneExcept
Unit1.pas.36: end;
```

图 2.27

图 2.27 中虚框所示的代码正是 2.9.1 节中注册异常处理块的代码。push $0044dbd2 压入的就是异常处理函数的入口地址。这并非是一个函数入口，是一条 jmp 语句，位于 try-catch 语句块中。它跳向了 HandleAnyException 函数。其实，call HandleAnyException 等价于以下两条指令组合：

```
call   0x0044dbd2
...
0x0044dbd2   jmp   HandleAnyException
```

为什么不像 DM2-26 第 4 行直接 push HandleAnyException 呢？这样先转到一个 jmp 指令，再跳过去似乎又费代码又费时间。更不能理解的是，catch 代码块的第一行 gi := 2 对应的指令 44dbd7h 并没有传递给异常处理函数 HandleAnyException。那么，该函数怎样能让程序流最后转向 44dbd7h？只有跟踪了。在 jmp 指令处设置断点，然后按 F9 键，执行程序，在断点处停止。通过单步跟踪发现，Delphi 并非像猜想的一样设定成员变量 eip 达到返回目的，而是直接通过一条 jmp 语句自己跳到 catch 代码块的入口（见图 2.28），最后一条 jmp ebx 指令跳回到了 catch 块的入口地址 44dbd7h。而 00403a66h 处的指令 add ebx, 05 将 ebx 加了 5 字节才是 44dbd7h。那么，之前 ebx 就是 44dbd7h–5=44dbd2h。参考图 2.27，地址 44dbd2h 正是被压栈的异常处理代码入口地址，即 jmp HandleAnyException 的地址。如果能获取异常块的地址，那么偏移 4 字节就是第 2 字段的地址，该段存储异常处理函数地址，即 44dbd2h，再加 5，就是 catch 块的入口地址（catch 块刚刚放在 jmp

HandleAnyException 后，而该 jmp 指令为 5 字节），见图 2.29。

```
00403A4D FFB000000000     push dword ptr [eax+$00(
00403A53 89A000000000     mov [eax+$00000000],esp
00403A59 8B6F08           mov ebp,[edi+$08]
00403A5C 8B5F04           mov ebx,[edi+$04]
00403A5F C74704703A4000   mov [edi+$04],$00403a70
00403A66 83C305           add ebx,$05
00403A69 E866FEFFFF       call NotifyAnyExcept
```

图 2.28

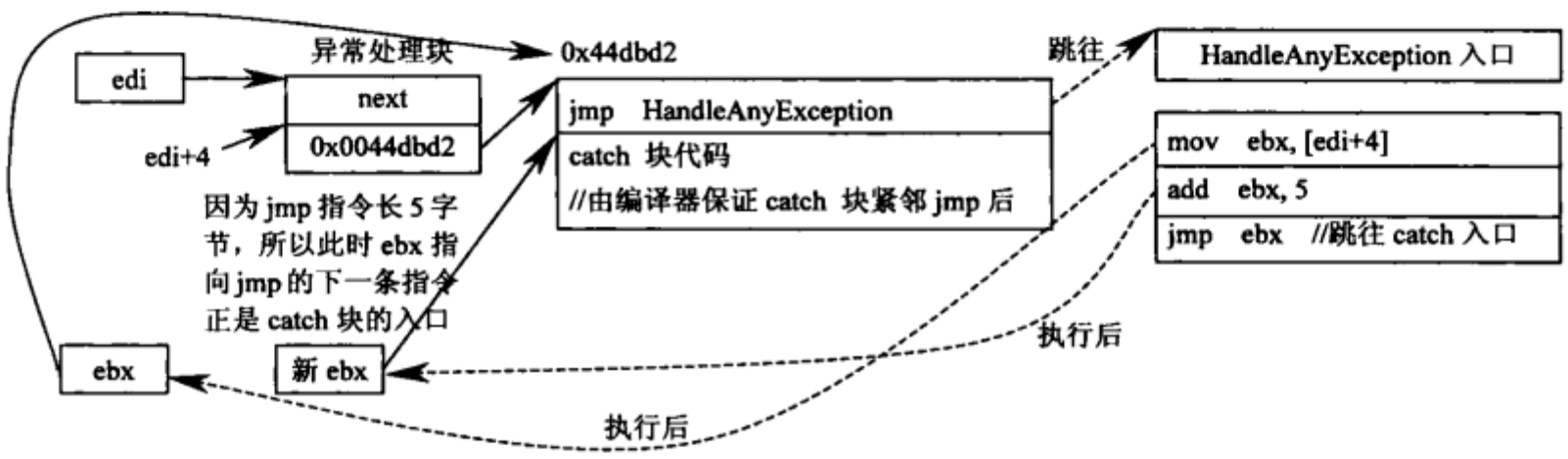


图 2.29

回头看前面异常处理函数的原型，第 2 个参数就是异常处理块的地址，也就是说，假设可行。从图 2.28 看，mov ebx, [edi + 04]指令是 add ebx, 05 执行前最后对 ebx 赋值之处。从图 2.25 可知，异常处理函数地址存于异常块首址偏 4 字节处，而执行 mov ebx, [edi + 04]后，ebx 为异常处理函数地址，倒推可得 edi 就是异常块的首址，即异常处理函数的第 2 个参数。

现在需要证明。再次运行程序，在图 2.27 中的 44dbd2h 处 jmp 指令查看栈映像，如图 2.30 所示。箭头指向的槽位保存的是返回地址，而之上第 2 个槽位在图中标记的就是第 2 个参数的值：0012f3cch（对比异常处理函数原型分析即知）。

```
  0012F004 0012F3CC
  0012F000 0012F0E4
➪ 0012EFFC 7C9232A8
```

图 2.30

执行图 2.28 中 mov ebx, [edi + 04]指令时，edi 的值是 0012f3cch，即证明了猜想。再次运行到这条指令时，查看寄存器窗体，edi 的值果然是 0012f3cch。同时，edi 就在其上方的 mov edi, [esp + 28h]处被赋值的。至此可知，Delphi 用如下方式获取 catch 块的地址：将 catch 块代码放在异常处理函数入口 jmp HandleAnyException 指令之后，将在异常处理函数中通过第 2 个参数获取到异常块地址，再从异常块获得异常处理的入口指令 jmp HandleAnyException 的地址，然后加该指令的 5 字节，就换算成其后 catch 块的入口地址。这就是 Delphi 在异常处理代码入口放 jmp HandleAnyException 指令的原因。

这个做法了启示我，在异常块后再压入 catch 块入口地址，在异常处理函数中获取到异常块地址后，加 8 字节，就能得到存储 catch 块入口地址的内存。这样就不用在异常处理函数中把 catch 块地址固定，而相当于参数传入。在 DM2-28 中，左边是 main()函数，增加的代码是第 1 行，压入

了 catch 块的入口地址 Normal。这样异常块结构体如下：

```
struct she {
    seh * next;
    //异常函数入口地址
    void * exceptionhandler;
    void * CatchEntryAddress;
}
```

DM2-28

main()函数	异常处理函数
push offset Normal	DWORD * pNormalAddr =
push ErrorHandler;	(DWORD *)((char *)PtrSEH + 8);
push fs:[eax]	PtrContext->Eip = * pNormalAddr;
mov fs:[eax], esp	…
…	

DM2-28 的右边是异常处理函数，获得 seh 地址（PtrSEH）后，加 8 字节，就是 seh 中 CatchEntryAddress 字段的地址，之后将其值赋给 eip，完成返回到 catch 入口地址的工作。工程见“code\第 2 章\structexception\betterReturnAddr”。与 Delphi 不压栈异常处理函数地址而是压栈 jmp HandleAnyException 地址相比，哪个更好？从时间性能来看，压栈 jmp 的地址，发生异常后要多执行一次 jmp 指令，正常执行却少压栈一次 catch 块入口地址（因为能通过 jmp 指令地址加 5 直接计算获得，不用像自己编写的代码要多 push 一次）。毕竟正常流程是绝大多数情况，所以反而快。再有，从栈消耗来看，因为没有压栈 catch 的入口地址，所以也少用 4 字节。

看到 Delphi 是通过直接 jmp 返回到 catch 块代码时，我想到：难道它不从异常处理函数返回了（**我的方案都要返回的**）？因此就在图 2.27 的 jmp HandleAnyException 断点处观察 esp 指向内存的值，此时操作系统刚 call 了该处理函数，esp 指向的内存应保存返回地址，是 7c9232a8h。跳到该地址，设置断点。然后不断按 F8 键，想看看到底是否在某条指令返回到了断点处。在调用了 ntdll 的 RTLUnwind 后，代码居然神奇地返回到 7c9232a8h，然后继续运行，又再次跳回异常处理函数（当然，现在它不会再返回了，而是跳到 catch 代码入口）。真的返回了吗？如果在 7c9232a8h 前面的 call 指令也设置断点，可以看到，调用了 ntdll 的 RTLUnwind 后再次经过了 call 指令，然后才执行 7c9232a8h 处的指令。所以异常处理函数并未真正返回，而是因为调用了 RTLUnwind 导致操作系统的异常分派函数重入了。至于 RTLUnwind 函数的作用，请大家参考“A Crash Course on the Depths of Win32Structured Exception Handling”。

2.9.3 那些状态保存到哪里了

虽然异常处理后能返回到 catch 块中执行，但这时的执行环境如果没有恢复，代码照样无法正确执行，如 EAX、ECX 等寄存器的值似乎应该与出异常前一样。如 2.2 节的 DM2-6 中，循环变量被放入 ESI 寄存器，如果 catch 代码块中要访问循环变量，就应该恢复 ESI 寄存器。Delphi 为了 catch 块代码的正常执行，恢复寄存器了吗？先看下面代码的反汇编。

```
try
    asm
    mov [0], 1
end;
except
    gi := 3;
end;
```

```
Unit1.pas.84: begin
0044DBAC 55              push ebp
0044DBAD 8BEC            mov ebp,esp
0044DBAF 53              push ebx
0044DBB0 56              push esi
0044DBB1 57              push edi
Unit1.pas.86: try
0044DBB2 33C0            xor eax,eax
0044DBB4 55              push ebp
0044DBB5 68D4DB4400      push $0044dbd4
0044DBBA 64FF30          push dword ptr fs:[eax]
0044DBBD 648920          mov fs:[eax],esp
Unit1.pas.88: mov [0], 1
0044DBC0 C705000000000100 mov [$00000000],$00000001
```

图 2.31

在图 2.31 开始处有 push ebx、push esi、push edi 指令。（**这是为 try 语句块恢复环境而设定的吗？验证。**）如果注释掉 try 语句块，这段代码确实消失了，看来与之有关。接下来，当代

码停在 catch 块入口“gi := 3”时，其环境包括刚才保存的 ebx、esi、edi 是否恢复到了图 2.31 中引发异常的断点语句 mov　[0], 1 执行时的值了吗？实验发现，除了 EBP，其余寄存器全部不同。

从这个例子看，“gi := 3”的成功执行确实与环境无关，因为 gi 是全局变量，指令中直接包含赋值地址。如需验证 catch 块是否涉及回复某些寄存器的工作，则在 catch 语句块中需要有与环境相关的操作：如用 stdcall 来定义函数，Delphi 就会用 EBP 寄存器相对寻址参数和局部变量。下面定义一个参数，并在 catch 段中访问。

```
procedure mycall(I: Integer);stdcall;
begin
    try
        asm
            mov  [0], 1
        end;
    except
        gi := i;                    //如果不恢复环境，这个赋值将错误
    end;
    gi:= i + 2;                     //查看前面那组保存的寄存器何时弹出
end;
```

在图 2.32 中，第 2 条指令在异常块后面压入了 EBP，可通过异常处理函数的第 2 个参数 seh 指针偏移 8 字节寻址。所以恢复 EBP 没问题。实验表明，进入 catch 段时，EBP 恢复了。而 i 是用 ebp + 8 寻址的，所以 catch 块执行 gi:=i 没有问题。

```
Unit1.pas.32: try
0044DBB2 33C0              xor eax,eax
0044DBB4 55                push ebp
0044DBB5 68D4DB4400        push $0044dbd4
0044DBBA 64FF30            push dword ptr fs:[eax
0044DBBD 648920            mov fs:[eax],esp
```

图 2.32

不用 stdcall，默认方式下，Delphi 会用寄存器直接传递参数，这时执行“gi := i”时要恢复传递参数值 i 的那个寄存器吗？反汇编，见图 2.33。其中，地址为 0044dbd3h 的语句将参数 I 赋给了地址为 ebp–4 的内存(此时参数 i 用 EAX 传入的)。而最后一行 gi := i 的反汇编首先是 mov　eax, [ebp–4] 从内存 ebp–4 取得了 i!(**本来是用 EAX 传递参数 i，又转换为 EBP 寻址了。**) catch 块运行环境的恢复又绕回到 ebp 上。(**之前已知 EBP 能恢复。**)

那么，是否没有 try 处理就不会用 ebp 寻址呢？见下面剔除 try 后的代码：

```
procedure mycall3(I: Integer);
begin
    asm
        mov  [0], 1
    end;
    gi := i;
    gi:= i + 2;
end;
```

反汇编，见图 2.34，没有 push　esi 等压栈，也没有 EBP 寻址，直接用 EAX 传递的参数（地址为 0044DBB6 的指令）。看来为了从异常中恢复执行，编译器放弃了许多优化。

代码正确运行必须依赖的环境是什么？必须恢复所有的寄存器吗？

代码从异常转入 catch 后能正常执行包括两部分：

① 执行完 catch 后，从包含 catch 的函数（如上例的 mycall）正确返回到调用该函数的地方。只要保证 ESP 能恢复为 catch 块所在函数的值即可。异常处理函数的第 2 个参数，即当前异常处理

```
Unit1.pas.43: begin
0044DBAC 55                  push ebp
0044DBAD 8BEC                mov ebp,esp
0044DBAF 51                  push ecx
0044DBB0 53                  push ebx
0044DBB1 56                  push esi
0044DBB2 57                  push edi
0044DBB3 8945FC              mov [ebp-$04],eax
Unit1.pas.45: try
0044DBB6 33C0                xor eax,eax
0044DBB8 55                  push ebp
0044DBB9 68D8DB4400          push $0044dbd8
0044DBBE 64FF30              push dword ptr fs:[eax]
0044DBC1 648920              mov fs:[eax],esp
Unit1.pas.47: mov [0], 1
0044DBC4 C705000000000100 mov [$00000000],$00000001
0044DBCE 33C0                xor eax,eax
0044DBD1 59                  pop ecx
0044DBD2 59                  pop ecx
0044DBD6 EB12                jmp +$12
Unit1.pas.50: gi := I;
0044DBDD 8B45FC              mov eax,[ebp-$04]
```

图 2.33

```
Unit1.pas.59: mov [0], 1
0044DBAC C705000000000100 mov [$0000000
Unit1.pas.61: gi := I;
0044DBB6 A3C0EB4400          mov [gi],eax
Unit1.pas.62: gi:= i + 2;
0044DBBB 83C002              add eax,$02
0044DBBE A3C0EB4400          mov [gi],eax
```

图 2.34

块地址，就是要恢复的 ESP（因为异常块放在栈上），不需专门保存 ESP。

② 包含 catch 的函数代码正确执行。只要保证能正常寻址该函数的变量和参数即可。如果所有变量和参数用 EBP 寻址，就只需保存 EBP。其他寄存器需要保存和恢复吗？取决于这些寄存器是否保留了运行状态。如作为循环的计数器，如果在一个循环中出现异常，那么该寄存器就要恢复。如果将计数器都放入内存 ebp–xx，就不会有问题。关键问题是看后续代码（catch 中及跳出 catch 后的）要依赖的寄存器。可将 try 语句块放入一个循环中，catch 块中的代码涉及计数器 i 的访问：

```
procedure mycall4(x: Integer);
var
    i: Integer;
begin
    for i := 0 to gi do
    begin
        try
            asm
                mov   [0], 1
            end;
        except
            gi := i;
        end;
    end;
    gi:= i + 2;
end;
```

从图 2.35 可知，ebp–4 是 i 的地址（见 0044dbf0h 处指令），地址 ebp–8 存储的是循环上限（见 0044dc00h 处指令）。优化全部没有了，本来这些都可直接存寄存器中。但为了 catch 块正确执行都存到内存中，并以 EBP 寻址。不妨看看去掉 try 语句块的循环优化代码：

```
procedure mycall5(x: Integer);
var
    i: Integer;
begin
    for i := 0 to gi do
```

```
    begin
      asm
        mov  [0], 1
      end;
      gi := I;
    end;
  end;
```

反汇编，见图 2.36。循环等状态直接存到了 EDX、EAX 寄存器中（见地址为 0044dbc8h 和 0044dbc9h 的两处指令），根本就不用 EBP 寻址。

```
Unit1.pas.69: for i := 0 to gi do
0044DBB5 A1C0EB4400      mov eax,[gi]
0044DBBA 85C0            test eax,eax
0044DBBC 7C47            jl +$47
0044DBBE 40              inc eax
0044DBBF 8945F8          mov [ebp-$08],eax
0044DBC2 C745FC00000000  mov [ebp-$04],$00000000
Unit1.pas.71: try
0044DBC9 33C0            xor eax,eax
0044DBCB 55              push ebp
0044DBCC 68EBDB4400      push $0044dbeb
0044DBD1 64FF30          push dword ptr fs:[eax]
0044DBD4 648920          mov fs:[eax],esp
Unit1.pas.73: mov [0], 1
0044DBD7 C705000000000100 mov [$00000000],$00000001
0044DBE1 33C0            xor eax,eax
0044DBE3 5A              pop edx
0044DBE4 59              pop ecx
0044DBE5 59              pop ecx
0044DBE6 648910          mov fs:[eax],edx
0044DBE9 EB12            jmp +$12
0044DBEB E97C5DFBFF      jmp @HandleAnyException
Unit1.pas.76: gi := I;
0044DBF0 8B45FC          mov eax,[ebp-$04]
0044DBF3 A3C0EB4400      mov [gi],eax
0044DBF8 E8D760FBFF      call @DoneExcept
Unit1.pas.78: end;
0044DBFD FF45FC          inc dword ptr [ebp-$04]
Unit1.pas.69: for i := 0 to gi do
0044DC00 FF4DF8          dec dword ptr [ebp-$08]
0044DC03 75C4            jnz -$3c
```

图 2.35

```
Unit1.pas.87: for i := 0 to gi do
0044DBAC A1C0EB4400      mov eax,[gi]
0044DBB1 85C0            test eax,eax
0044DBB3 7C17            jl +$17
0044DBB5 40              inc eax
0044DBB6 33D2            xor edx,edx
Unit1.pas.90: mov [0], 1
0044DBB8 C705000000000100 mov [$00000000],$00000001
Unit1.pas.92: gi := I;
0044DBC2 8915C0EB4400    mov [gi],edx
Unit1.pas.93: end;
0044DBC8 42              inc edx
Unit1.pas.87: for i := 0 to gi do
0044DBC9 48              dec eax
0044DBCA 75EC            jnz -$14
Unit1.pas.94: end;
0044DBCC C3              ret
0044DBCD 8D4000          lea eax,[eax+$00]
Unit1.pas.98: mycall5(3);
0044DBD0 B803000000      mov eax,$00000003
0044DBD5 E8D2FFFFFF      call mycall5
```

图 2.36

可见，确实由编译器保证除 EBP 外，不依赖其他寄存器保存状态，这样恢复环境时就可只恢复寄存器 EBP 和 ESP。而 ESP 从异常处理函数的第 2 个参数直接恢复，不需保存，因此居然只需要保存一个 EBP 值。恢复 ESP 的代码在图 2.27 调用的 DoneExcept 代码中，恢复 EBP 的代码为图 2.28 的地址为 00403a59h 的指令，在转跳回 catch 代码块之前。

既然恢复 catch 运行只需保存 EBP，那么图 2.31 中只要有 try 的函数，函数一开始必然要压栈，保存一些寄存器，又是为什么？笔者认为，合理的解释是，虽然恢复包含 try-catch 块的函数执行不需恢复其他寄存器，但难免调用该函数的函数不使用它们，所以进行了选择性保存（就如同 C 语言的 longjmp 也没有保存全部寄存器）。

最后模拟 Delphi，恢复 EBP、ESP，并不从异常处理函数返回的方式实现新的版本，参见“code\第 2 章\structexception\delphilike”。

2.9.4 意外的秘密

实践让人真正学习进步，因为它面临太多的意外和未知。当我写完上节的例子后，写下了一行注释，提示在发行版中对局部变量 i 赋值将出错。因为想象中 i 可能优化后不用 EBP 寻址。为了严谨，在大年廿九的深夜临睡前，我运行了一次发行版。居然无法抓住异常！（**应该是两个版本编译上有区别吧**。）让自己好好休息了一天，在新春初一的阳光里，我开始了一天奇妙的探索。

开始用一个比较笨的办法，打开两个版本的工程配置，一个个比较并修改，让发行版和调试版

相同。让人沮丧的是毫无结果。是否是执行程序格式不同？文件分析难以入手，因为编译出的代码就有很大不同。只有跟踪，看 Windows 到底干了什么。我在 Delphi 中写了一段 try-except 代码。然后在图 2.27 中地址 0044dbd2h 的指令设置断点，停下后查看栈顶保存的返回地址为 7c9232a8h。在该地址之前是一条 call　ecx 指令（其地址是 7c9232a6h）。

难道是在 VC 发行版中，这条指令中的 ecx 不对，所以转不到异常处理代码吗？于是设计了下面的实验：在 call 指令处（7c9232a6h）设置断点，调试观察 ecx。为了调试器能 attach 到发行版进程中，在程序开始加入了 getchar 语句。在 getchar 等待时，用 Delphi 调试器 attach 到被测试进程中，并在 7c9232a6h 处设置断点，然后恢复执行，即可在异常发生时停留在断点并查看 ecx。令人惊讶的是，根本没有在该断点停下。（**看来之前就走了不同路径，这太令人意外了**。）现在需要将包含这段代码的系统 DLL 找出，反汇编并分析它。在 Delphi 中，按 Ctrl+Alt+M 快捷键查看所有加载的 DLL，其中 ntdll.dll 的加载基址是 7c920000h。而 call 指令在地址 7c9232a6h 处，离 ntdll.dll 基址 7c920000h 在 64k 之内，远小于 ntdll.dll 大小，所以该代码应该属于 ntdll.dll。将 ntdll.dll 用 IDA pro 反汇编（在 C 盘下搜索，即可找到该文件），反汇编，原始文件见“code\第 2 章\structexception\ntdll 反汇编\原始版本\ntdll.idb”。

下面需要找到函数调用该 call 指令所经历的途径。可以用 IDA 的“查看调用者”来静态分析。如果一个函数的调用者太多，就需要结合调试，不断单步执行，使其从函数中返回，并记录下返回地址。然后在 IDA 中找到这些地址所在的函数，即可获得调用的序列。比如，要找到之前地址 7c9232a6h 处的 call 指令，选择“jump”→“jump to address”，在弹出对话框中输入“7c9232a6”，回车，就转到了函数 sub_7C923282 中。然后在函数名右边的 CODE XREF 上单击右键，在弹出的快捷菜单中选择“jump to cross reference”，在弹出的对话框中选择唯一调用它的代码。继续这一静态回溯时，后面将出现多个调用者，因此转入前面提到的调试结合 IDA 静态分析的方法。获得如下调用序列：KiUserExceptionDispatcher → sub_7C94A487 → sub_7C923247 → sub_7C923282。

这是异常能捕获时执行的序列。异常不能捕获时执行的序列，到底从哪里开始产生了不同？要分析以下 3 种情况：① VC 发行版用 try...catch 语句成功捕获异常，简称发行正常版；② VC 发行版自己构造异常处理块，无法捕获异常，简称发行错误版；③ Delphi 编写的可成功捕获异常的版本，简称 Delphi 版。为了方便对比，将①和②合成为一个程序，当用户输入 1 时，执行发行正常版的代码，否则执行发行异常版的代码（如果输入 0，则会执行对空地址赋值逻辑，触发异常）。工程见“code\第 2 章\structexception\test2”。（**这个程序出了一点状况，在 VS 2008 中用 try 语句捕获 0 除法错误时，竟然捕获不到，尝试修改其工程属性配置，发现将其选项“配置属性”→“C/C++”→“代码生成”→“启用 C++异常”选择为“/EHa”才能捕获**。）运行 test2.exe，输入“0”，回车，再输入“0”，回车，程序结束，并没打印“hello error”。

为了找出 3 种情况执行序列的差异，我选择了在 4 个函数调用指令设置打断点的方式，然后查看在 3 种情况中没有执行到哪个断点，再逐步调试对比，最终定位出不同的执行序列。

首先是 Delphi 版与发行正常版的区别，它们都执行了以下调用序列：sub_7C94A487 → sub_7C94A567 → sub_7C94A5E8 → sub_7C94A691。在后 3 个函数，两个版本产生了差异。（**必须理解到底做了什么，才能进一步找到办法**。）先在 sub_7C94A691 中看到了对两个函数的调用：RtlImageNtHeader 和 RtlImageDirectoryEntryToData。从名字看，应该是两个获取执行文件相关数据的函数。（现在正是在寻找执行程序属性的不同。）为了理解程序的含义，要尽量识别出相关参数和局部变量的含义。这一过程可能无法一次完成，需要猜测并调试证明，然后利用 IDA 的功能将参数名修改为能代表其含义的名字。例如，在 sub_7C94A691 中有如下代码：

```
push    ecx
push    [ebp+arg_0]
call    RtlImageNtHeader
```

根据第 1 章重定位中的经验，执行程序具有一定格式并被加载到内存，而要解析这个格式至少应该知道该程序被加载到内存中的地址(基址)，这样才能根据相对偏移量拿到想要的信息。Windows 系统调用都是 stdcall，所以参数都是压栈的，那么 push [ebp + arg_0]指令可能就是传递执行程序的基址。当然，也许是 push ecx 传递的。只要确定了 RtlImageNtHeader 函数只有一个参数即可解决问题。因为是 stdcall，所以只要看该函数最后的返回指令 ret x，这个 x 如果是 4，就可推定它只有一个参数。双击 call 指令右边该函数的名字，就能转到该函数，查看其 ret 指令，证明该函数只有一个参数。那么，离 call 最近的 push 指令就是压栈参数。所以，这时可将该指令中的 arg_0 修改为 peImageBase（右键单击 arg_0，在弹出的快捷菜单中选择“rename”，然后在弹出的对话框中输入“peImageBase”)。现在需要证明猜想，只需调试时在 call 指令所在的地址 7c94a69ah 处设置断点并查看，其值为 400000h。这是编译器选择的默认值。没有经验也没关系。执行 PeLord (tools\LPE-DLX.ZIP)，在窗体右上角处单击“PE Editor”按钮，在弹出的对话框中选择被调试的执行程序，然后在窗体中查看 ImageBase 项，确实是 400000h，见图 2.37。

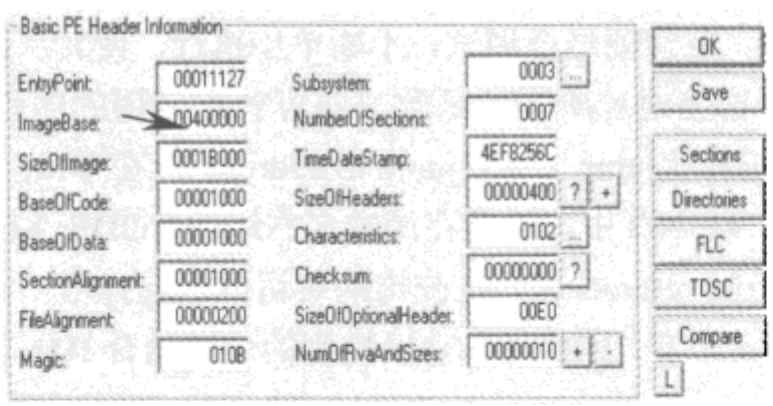


图 2.37

sub_7C94A691 的另外两个参数呢？一下无法知道其含义，分析是否为指针类型可找到端倪。因为如果是指针，基本可推断被调用函数会为该指针指向的内存填写相关信息。也就是说，该函数会通过该参数获取一些信息。这时要分析调用该函数的代码才能推测，见 DM2-29，第 3 行是传递第 2 个参数，压栈 eax。eax 是在第 2 行赋予的。查看函数的开始 var_4，即–4，那么 ebp+var_4 就是 ebp–4，这是典型的局部变量的地址（见 1.3.5 节）。第 2 行正是将变量的地址，ebp+var_4，赋给 eax。(**这个 lea … 在编译器编译的代码中往往都是取地址的指令**。)第 3 个参数是在第 1 行压入的，即参数 arg_8，它是代码所在函数的第 3 个参数。要确定其是否为指针，就需要往外一层，看调用包含代码的函数压栈参数的情况，见 DM2-30。第 1~2 行正是传递第 3 个参数的代码。明显与前面一样，这也是一个指针变量。

DM2-29

```
1  .text:7c94a658    push   [ebp+arg_8]
2  .text:7c94a65b    lea    eax, [ebp+var_4]
3  .text:7c94a65e    push   eax
4  .text:7c94a65f    push   esi
5  .text:7c94a660    call   sub_7C94A691
```

DM2-30

```
1  lea    eax, [ebp+var_8]
2  push   eax
3  lea    eax, [ebp+var_14]
4  push   eax
5  xor    ebx, ebx
6  push   esi
7  mov    [ebp+var_C], ebx
8  call   sub_7C94A5E8
```

最后整理一下对 sub_7C94A691 的分析：① 第 1 个参数为执行程序基址；② 第 2、3 个参数为指针变量，本函数可能通过这两个参数返回调用者感兴趣的信息。

现在需要修订该函数的声明。在函数开始的 sub_7C94A691 proc near 处，右键单击该函数名，在弹出的快捷菜单中选择“set function type”，将弹出对话框中的第 2、3 个参数从 int 暂时改为 int *，将两个

参数的名字暂时改为 pInfo1 和 pInfo2，便于以后分析猜测。通过调试可知，Delphi 版和发行版在 sub_7C94A691 中产生了执行路径的差异。

DM2-31

```
1  lea    eax, [ebp+var_4]
2  push   eax
3  push   0Ah
4  push   1
5  push   [ebp+peImageBase]
6  call   RtlImageDirectoryEntryToData
7  test   eax, eax
8  jz     loc_7C955EFC
```

在 DM2-31 中，因为前面判断出 arg_0 是执行程序基址，已经修改为 peImageBase 了，所以第 5 行中的 ebp+arg_0 就自动变成了第 5 行中的 ebp+peImageBase。(**这就是 IDA 的强大，通过重命名函数、变量、参数，之前的分析结果就会慢慢累积，并在某个时候豁然贯通。**)第 6 行的函数返回结果在 eax 中(见 1.3.7 节)。第 7～8 行对返回值进行自身求 and，以测试是否返回值为 0。Delphi 版返回值为 0，而 VC 发行版不为 0，因此 Delphi 版将在第 8 行跳转。(这已经证明了开始的猜想，即 VC 发行版编译出的执行程序某个属性不同。) 从第 6 行函数名来看，似乎在查找 pe 格式的某个目录。这个函数有 4 个参数，第 3 行和第 4 行传递的是常量，估计其中一个应该是指出要查询的目录（0ah 更像，1 似乎是一个布尔属性)。既然查询某个目录，则应该有返回目录相关属性的地方。还剩第 4 个参数和返回值两个地方。第 1～2 行与前面相似，是在传递指针，自然可推定这个参数是一个返回相关属性的地方。(返回值呢？) 通过第 7～8 行，检验返回值是否为 0，可推断要不返回值代表布尔值，要不就是指针。(**这时需要用调试来验证。**)如果非零的返回值不是 1，而是一个不太规则的整数，那么基本可以推定返回值为指针。调试结果支持指针这一论断。这时对该函数的意义较清晰了，它查询了 pe 格式中一个特定的目录(或特定目录的指定子属性)，这个目录（或特定目录的指定子属性）的存在与否影响了异常处理函数是否被调用。如果存在，返回值似乎是一个指针。

进一步似乎无路可走，没关系，调试观察往往是拨开迷雾的契机。现在需要查看发行正常版和发行错误版的区别。刚才分析的函数 sub_7C94A691，两个版本有区别吗？没有，它们都在 DM2-31 的第 8 行不转跳。根据前面找到的调用序列，不断追踪，在 DM2-32 的第 2～3 行检验 sub_7C94A567 函数的返回值是否为 0 时发现：错误版返回值为 0，从而转跳；发行正常版返回非 0。可推断 sub_7C94A567 的返回值是布尔类型，因为能够在该函数返回指令前看到对 eax 设定的指令 mov al, 1（其后虽然有一个函数调用，但该函数中没有设定 eax，所以 mov al,1 是最后设定 eax 的指令，是设置返回值的指令)，意味着将返回值设定为真，而非指针。(**函数 sub_7C94A567 的返回值也许意味着是否可以调用异常处理函数？**) 为了印证这一猜测，调试 Delphi 版本时，可知 sub_7C94A567 返回的是 1，而因为这个 1，该版本能调用其异常处理函数。

DM2-32

```
1  call   sub_7C94A567
2  test   al, al
3  jz     loc_7C93B2D8
```

跟踪，看在哪里导致发行错误版返回值为 0，见 DM2-33。

发行错误版进入了一个循环：第 4～9 行、第 14～16 行形成了一个循环体。第 7 行中的 eax 类似数组的起始地址，ecx 相当于循环计数器，将数组当前元素（地址为 eax+ecx*4）的值赋值给 ebx。第 8 行将该值与 esi 比较，如果大于 esi 的值，就转到第 14 行。循环完毕，则从第 17 行跳转，并最终转到第 18 行。注意此时返回值 eax 的低字节 al 被赋值为 0，意味着函数返回值为 0（这正是导致发行错误版无法调用异常处理函数的原因)，并从第 19 行跳至第 12 行，退出函数。

这个循环中，第 8 行与 esi 的比较是一个关键。从第 1 行可知，esi 和 ebp+var_14 中存放的值相减，并将差赋回给 esi。调试时，在第 1 行，发现 esi 的值是 401000h，而地址 ebp+var_14 中存储的值是 400000h。(400000h **就是执行程序的基址。**) 0x401000 是什么？在 VS 环境中调试发行版，然后在反汇编情况下寻找该地址，容易发现，它是发行错误版中“push ErrorHandler”压栈的异常处

DM2-33

```
1   sub    esi, [ebp+var_14]
2   xor    edx, edx
3   cmp    edi, ebx
4   jl     loc_7c9700ec
5   lea    ecx, [edi+edx]          ;循环体开始
6   sar    ecx, 1
7   mov    ebx, [eax+ecx*4]
8   cmp    esi, ebx
9   jb     loc_7c93b2e1            ;转跳到第 14 行
10  ja     loc_7c94d9f4            ;最终会转到第 15 行进入循环
11  mov    al, 1                   ;第 8 行比较结果，相等，则不会在第 9 行进入循环，返回结果为 1
12  mov    ecx, [ebp+var_4]
    ...
13  ret    4
14  lea    edi, [ecx-1]
15  cmp    edi, edx
16  jge    loc_7c94a5bc            ;继续循环转跳到第 5 行
17  jmp    loc_7c9700ec            ;循环完毕退出，最终转跳到第 18 行
    ...
18  xor    al, al
19  jmp    loc_7c94a5d4            ;转跳到第 12 行，退出函数
```

理函数 ErrorHandler 的入口地址。第 1 行将这个地址与基址相减，得到的就是代码相对首部的偏移量。1.8.2 节还讨论过这个偏移量可以用于重定位。在 PE 格式中，它叫 RVA（Relative Virtual Address，相对虚拟地址）。从第 7～8 行将异常处理函数入口地址的 RVA 表示与 eax 指向的数组中存储的值比较，自然猜想到，数组中的值也应该是一个 RVA。既然被比较的是异常处理函数入口的 RVA，那么数组中存储的 RVA 应该也是代码的 RVA。只需在第 8 行，将 ebx 的值加上基址 400000h，就应该是 RVA 对应的绝对地址。通过跟踪发现，这个数组有两个元素，内存中的字节值分别为“25 19 00 00”和“50 1a 00 00”。因为是小端机，所以对应的整数值为 1925h 和 1a50h。加上基址 400000h，则两个绝对地址分别为 401925h 和 401a50h。在 VS 调试环境的反汇编窗体中查看着两个地址，分别对应 DM2-34 中的两段代码。

DM2-34

```
_except_handler4:
00401925        mov   edi, edi
...
__ehhandler$_main:
00401A50        mov   edx, dword ptr [esp+8]
```

DM2-35

```
00401035        push   401A50h
0040103A        mov    eax,dword ptr fs:[00000000h]
00401040        push   eax
```

从注释可知，这两个地址是两个异常处理函数的入口，后一个似乎是 main()函数中的异常处理函数入口。在 VS 中调试，并展开反汇编窗口，则 main()函数一开始有如 DM2-35 所示的压栈异常处理函数入口的代码，第 1 行正是压栈了 main()函数中 try 语句对应的异常处理函数入口：401a50h。这与 DM2-34 最后一行的结果吻合。（**此时似乎真正触摸到问题的核心。**）

归纳之前的分析如下：

似乎从执行程序中获取到了一个数组，当中保存了所有编译器生成的异常处理函数入口的地址的 RVA（我称其为**异常处理函数注册数组**）。循环比较当前异常处理函数入口 RVA 是否与数组中的某个项相同（DM2-33 中第 9、10 行均不相等，都将转入循环查找），相同，则退出循环并返回真（DM2-33 的第 11 行将 eax 赋值为 1）。如果找不到，则循环完毕退出，返回为假（最终会执行 DM2-33

的第 18 行，将 al 赋值为 0）。通过调试发行正常版，此时执行 DM2-33 第 1 行，esi 的值是 401a50h，就是 main()中 try 语句对应的异常处理函数入口，它存在于异常处理函数注册数组中，所以找到匹配后转向了第 11 行，退出函数，返回为 1。也就是说，在发行版下，执行程序中应该包含了编译器生成的异常处理函数入口 RVA 数组。

用 UltraEdit 来看执行程序中是否真的存在这个数组，即查找当中是否存在“25 19 00 00”和“50 1a 00 00”。果然查到，且只有一处。现在已经获得第一个让发行错误版正确执行的方案。用 UltraEdit 修改这个数组中最后一项的值，将其改为注册的异常处理函数入口的 RVA。这样执行 DM2-33 时，就能找到匹配项，就会调用注册的异常处理函数。

我注册的异常处理函数入口是 401000h，执行程序基址是 400000h，那么该函数的入口 RVA 的地址为 401000h–0x400000h =1000h。因为是小端机，所以存储的顺序为“00 10 00 00”。修改后的结果如图 2.38 黑线所示，并保存为“code\第 2 章\structexception\Release\test2_pack1.exe”。运行输入 0，回车，再输入 0，回车即可看到自定义异常处理函数执行。

```
00001230h: 25 19 00 00 00 10 00 00
```

图 2.38

到这里可以停下了，但我很想知道异常处理函数注册数组到底存在程序的哪个段。这时可回溯到之前分析的函数 sub_7C94A691，来确定之前留下的问题：

① RtlImageDirectoryEntryToData 的返回值到底是什么？

② 其第 2 个和第 3 个参数是指针，通过函数 sub_7C94A691 将填入什么信息？

关于问题①，既然 RtlImageDirectoryEntryToData 是查询 PE 文件目录。根据 PE 相关格式，我们知道，定位某个目录可用其索引值，这些值被定义为宏。DM2-31 中，第 3 行压栈的 0ah 可能是其中某个索引值，用它指定被查询目录是最自然不过。通过分析宏定义发现，IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG 的值正是 10（即 0ah）。是否意味 DM2-31 的代码是在查询 LoadConfig 目录呢？用 PELord 分析 Delphi 版和 VC 发行版发现，Delphi 版没有 LoadConfig 目录，而 VC 发行版有。从 DM2-31 执行时第 6 行调用 RtlImageDirectoryEntryToData 的返回结果看，Delphi 版返回 0，VC 发行版返回 402120h，似乎证明确实在查找 LoadConfig 目录。因为 Delphi 版没有该目录，所以返回 0，而 VC 发行版有，所以返回其地址。接下来需要证实查询的是 LoadConfig 目录。思路是先用 PeLord 查看该目录的内容，然后在 DM2-31 的第 7 行设置断点。调试 VC 发行版，如果此时 EAX（值为 402120h）指向 LoadConfig 目录，那么通过内存窗体查看，内存 402120h 的内容与 PELord 的内容相同。

在图 2.37 的 PELord 的界面下单击右侧的 Directories 按钮，然后在弹出的对话框中查看 LoadConfig 项目的 RVA 是 2120h，大小为 40h，再单击其右边的大写 H 的按钮，可显示该目录中的二进制信息，见图 2.39。

调试时观察到的内存 402120h 区的内容如图 2.40 所示。

可知，图 2.39 中黑色背景的数据块与图 2.40 中黑色框圈定的内存块内容一模一样。这就是 LoadConfig 段的内容。刚刚通过 PELord 已查到 LoadConfig 目录的 RVA 是 2120h，加上执行程序基址 400000h，等于 402120h，它是程序执行时该目录的内存首址。该值正是调试跟踪时 DM2-31 的第 6 行 RtlImageDirectoryEntryToData 的返回值 402120h。由此可见，函数 sub_7C94A691 确实查询的是 LoadConfig 目录。可是 PELord 显示该目录只有 40h 大小，即图 2.39 中黑色块的大小，没有异

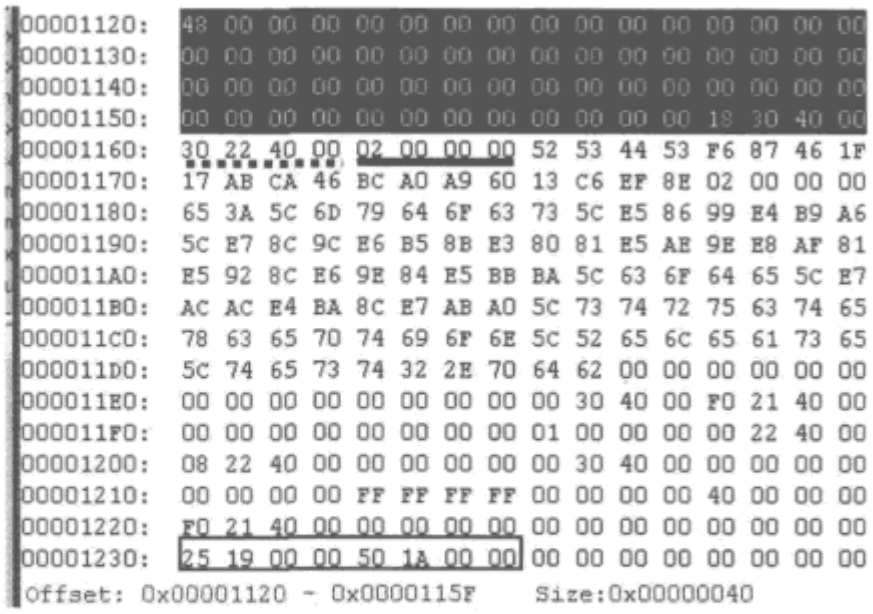


图 2.39

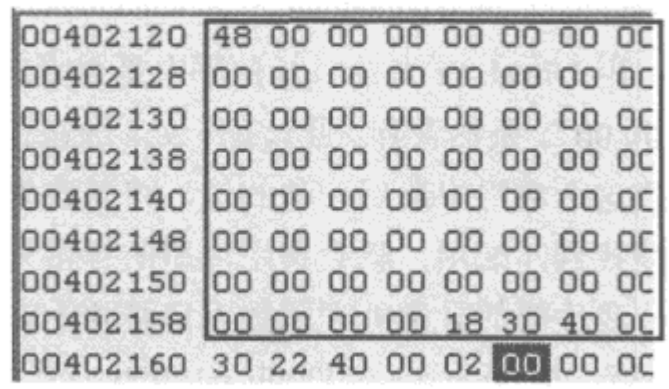


图 2.40

常处理函数注册数组。但在图 2.39 的最后一行，由黑色框圈定的 8 字节正是前面分析的异常处理函数入口的 RVA：25 19 00 00 50 1a 00 00。这说明这个数组不是 LoadConfig 目录的一部分，而是“隐藏”在其后的非公开文档（undocument）。（**微软就喜欢玩这些把戏。**）

在 DM2-31 之后就是用 Loadconfig 首址求取异常处理函数注册数组首址和元素个数的代码，见 DM2-36。通过分析和跟踪很容易得出清单注释中给出的解释，这里不详细分析。第 3 行从 Loadconfig 首址偏移 40h（Loadconfig 大小）获得了异常处理函数注册数组内存首址。图 2.39 中第 5 行虚线标定的 4 字节，即整数值为 402230h（容易算出该值指向图中下方黑色方框的起始位置）。DM2-36 第 10 行从 Loadconfig 首址偏移 44h 获得了数组元素个数。图 2.39 中第 5 行实线标定的 4 字节，即整数值 00000002h（有两个元素）。简而言之：

异常处理函数注册数组的首址和元素个数信息紧邻 LoadConfig 目录放在其后，共 8 字节，前 4 字节为加载后异常处理函数注册数组内存首地址，后 4 字节为元素个数。这就是微软异常处理在 PE 格式方面的一个“秘密”。

DM2-36

```
1   call  RtlImageDirectoryEntryToData
2   test  eax, eax
    //如果查询到了 LoadConfig 目录就执行下面的代码
    //eax 是 LoadConfig 的首址
    ...
    //LoadConfig 大小是固定的 40h，它后面 4 字节就是异常处理函数注册数组加载后在内存中的首址
3   mov   ecx, [eax+40h];
4   test  ecx, ecx
5   jz    loc_7C955EFC
    //Loadconfig 后偏移 4 字节的地方存储了异常处理函数注册数组元素的个数，如果没有元素，就另处理
6   cmp   dword ptr [eax+44h], 0;
7   jz    loc_7C955EFC
8   mov   edx, [ebp+arg_4]
9   mov   [edx], ecx            //将异常处理函数注册数组的首址赋给本函数第 2 个参数 arg_4 指向的内存
10  mov   eax, [eax+44h]        //获取异常处理函数注册数组元素的个数
11  mov   ecx, [ebp+arg_8]
12  mov   [ecx], eax            //将异常处理函数注册数组元素的个数赋给本函数第 3 个参数 arg_8 指向的内存
```

为什么 VC 的调试版可以执行我们自定义的异常处理函数呢？它如果没有生成 Loadconfig 目

录，那么就与 Delphi 版一样，不会进入 DM2-33 的循环比较逻辑。用 PeLord 很容易证实这一点。那么，利用刚才的分析还有什么办法让 VC 发行版依然能执行我们自定义的异常处理函数呢？只要将异常处理函数注册数组首址和元素个数这两个属性都改成 0，就相当于没有异常处理函数注册数组了。DM2-36 的第 4～5 行和第 6～7 行正是在判断这两个属性是否为 0。这次直接在图 2.39 所示的 PELord 中直接修改，将第 5 行中虚线和实线标定的 8 字节全部改为 0（选择要修改的字节，然后输入 0 即可）。修改版本见“code\第 2 章\structexception\Release\test2_pack2.exe”。运行输入 0，回车，再输入 0，回车，即可看到自定义异常处理函数执行。本书提供了注释较好的 ntdll.dll 的 IDA 反汇编文件，包括修改的函数名、参数名、变量名、注释等，见“code\第 2 章\structexception\ntdll 反汇编\ ntdll.idb”。

为什么 VC 发行版要校验当前异常处理函数是否是编译器生成的异常处理函数注册数组中的一员？也许是安全需求。比如，溢出攻击代码的一种技巧就是自定义异常处理函数查找 kernel32 的基址。如此限制后，异常处理函数只能是编译器生成的代码了。

回顾学习异常机制的过程，绝大多数人在完成 2.9.1 节后就止步了，他们甚至也动手实现了资料中展示的例子。而提出 2.9.2 节的问题并探索的人就比较少了，追问 2.9.3 节的问题并分析的人就更少。遇到 2.9.4 节的问题，没有底层跟踪分析及 PE 格式的基本知识是根本不可能解决的，因为涉及的知识点就是一个“非公开文档”，难以在网上查找。真正的学习就是要用反汇编去“庖丁解牛”，要猜测并自己去实证。这一过程中的猜测至关重要，否则就没有深入分析的可能。同时，实证又能发现新问题（如 2.9.4 节），两得益彰。

小　结

限于篇幅，本章涉及的有些内容删减了。比如，如何通过把玩，构建溢出攻击来学习底层编码；用反汇编和调试分析 Servu，为了一个空格付出的溢出代价，以及攻击之后获取执行权限的机制；分析 .NET 的 JIT 机制等。

反汇编和跟踪应用领域较广，但似乎没有成套路的规范方式学习（像编写正常程序那样），往往学习了基础知识，却无从继续深入运用，最终无缘感受它的魅力。似乎前辈们也没有给出很好的建议。笔者简单总结了以下几点建议：

① 对于遇到分析源码难以搞清的问题，要敢于用反汇编分析。

② 遇到不知究竟的机制，要敢于用反汇编跟踪，而不是止步于简单从网上搜索资料（如现在大家可以动手思考和分析对象到底怎样做的，因为归根到底它都被编译成了机器指令）。

③ 从编写安全对抗程序入手，如编写溢出攻击和防御的程序，编写钩子程序（不是用 Detours，而是自己写钩子掌握底层机制）。

其中前两条需要慢慢积累，后一条能够较快和集中地学习，不过因为篇幅原因，就无法展示了。

更多扩展

学习 Windows 系统下异常处理机制“A Crash Course on the Depths of Win32Structured Exception Handling”。

从网上下载 Detours，学习其原理。

习 题 2

1．从 DM2-2 的第 4 行可见，IDA 知道 call 指令调用的是 printf()。我们知道，call 指令不过由

5 字节组成：e8 和 4 字节偏移量。被调用函数也只是一堆指令组成，为什么 IDA 能够分析出是在调用 printf()呢？请试构想其背后的机制。

2．请给出 2.1 节更多的解密方式。

3．如果被调用函数为_stdcall，调用方将其视为 cdecl 调用惯例调用，是否一定会出错？请举例并分析。

4．有函数指针被定义为“typedef int (_stdcall *PtrHandle)(int, int, int, int);”，调用惯例为_stdcall。有如下代码：

```
Callhook(PtrHandle hnd) {
    int a = hnd(1, 2, 3, 4);
}
```

如传入的 hnd 不是_stdcall 而是 cdecl（即 C 语言调用惯例），如下：

```
int hook1(int i1, int i2, int i3, int i4) {...}
Callhook((PtrHandle)hook1);
```

结果会正确吗？请指出原因。如果希望 Callhook 调用 hnd 时，既能调用符合 PtrHandle 所定义函数原型的函数，也可正确调用调用惯例是 cdecl 的函数（如上面的 hook1），该怎样做？

Windows 回调窗体处理函数时，窗体函数既可以是_stdcall（符合文档标准），也可以是 cdecl 的函数（不符合文档标准）。估计是为了兼容早期代码。

5．对于习题 4 提到的兼容调用_stdcall 和 cdecl，请自学 Windows 窗体编程，并用反汇编分析 Windows 兼容调用_stdcall 和 cdecl 的做法，且比较与第 4 题做法的异同。

6．分析 Visual C++的_fastcall 和 Delphi 的默认调用方式分别选用哪些寄存器传递参数，什么条件下会用栈混合寄存器传递参数。

7．用 Visual C++编写一个 DLL，并寻找办法用源代码方式调试该 DLL 输出的函数。

8．将本章 2.4 节例子中的 DLL 换成 Visual C++实现，其 Add()函数是_stdcall。另外，将激活程序的项目属性窗体设定为“项目属性 | 配置属性 | C++ | 代码生成 | 基本运行时检查”，设定为默认值。此时代码不会生成 DM2-12 的第 1、8、9 行。请问是否会出错？分析其运行结果和原因。

9、采用 VS 2008 的 Debug 版，用反汇编分析编译器检测指针未初始化的机制（可参考 2.4 节分析 DM2-12 的检测栈平衡机制）。

10．Windows 下有一个函数叫 VirtualProtect()，可修改内存页的属性。运用这个函数，让 2.5 节相关的程序可正确写常量串"12345678"的内存区。之后，请将该代码移植到 Linux 下（关键要找到等价 API）。

11．分析 Delphi 除了对局部变量类型的接口变量自动调用_AddRef 和_Release 保证引用计数策略自动实施外，还有哪些情况应该自动调用这两个函数，何时、何地调用。并用反汇编分析自己的猜测是否成立。

12．分析 Visual C++和 GCC 的异常处理机制，分析其关键技术，并用汇编实现之。

13．理解了 2.9.4 节的过程，请用 IDA pro 和调试器分析。

14．在 Windows 下运行并分析以下程序出错的原因，并给出一种以上的解决方案。

```
void main(int argc, char * argv[]) {
    int i1, i2;
    char sample[1024][1024];
    for (i1 = 0; i1< 1024; i1++)
    for(i2 = 0; i2 < 1024 ; i2++)
        sample[i1][i2] = 'c';
}
```

15．在 1.6 节中对 Visual C++中的 switch 的实现进行了相关分析。请对 Java 的 switch 指令进行分析。Java 没有提供反汇编一级的调试分析，因此无法像 Visual C++一样方便分析。Java 是 JIT 即时编译字节码为机器码，所以无法运行前事先对执行时才即时生成的代码打断点。请设计一个可调试的代码，运用调试器，attach 到 Java 进程中分析。

16．模拟 Delphi 的用“jmp 地址+5”计算 catch 入口地址的方法，修改“code\第 2 章\structexception\betterReturnAddr”。

第3章

成长：与程序一起茁壮

需准备的知识 »

- 3.1.3节需要第1章的内存窗体使用知识，见图1.9和图1.10。
- 3.2.3节需要1.4.1节的知识。
- 3.3节需要1.8.2节的知识。
- 3.3.3节的结构体对齐bug需要1.5.1节的知识。
- 3.3.5节需要1.3.1节的知识。
- 3.3.5节和3.3.6节需要1.8.3节的动态链接库工程创建知识。

为你提供的能力 »

- 3.1节 → 学会一种工程驱动的自我学习方法，养成基本的良好编码习惯。
- 3.2节 → 测试驱动的开发，用测试重构和改善设计，形成良好的工程习惯。
- 3.3节 → 理解了链接机制，能解决诡异的编译链接错误，进一步强化底层程序观。
- 3.3.2节 → 理解动态数组，进一步理解动态分配内存的使用，强化内存必须释放的概念。
- 3.3.1节 → 理解结构化数据的存储基本原理和机制，搞懂硬盘数组（这是构造数据库的最小原核）。
- 3.3.6节 → 深入理解动态链接库的使用机制，理解其中各细微的环节，防止诡异错误。
- 充分理解猜测—实证—构建方法。

如何做好软件开发的自我学习呢？除了把握基础，善用底层的力量猜测、求证，只有写程序一途。写好一个个具有完整功能的“能用”的程序，而非仅仅是那些“示例”程序。即使在初学状态下，也应该从一开始投入到“能用”程序的开发。它能激发自学能力，知识可马上学以致用，建立系统的整体观并激发兴趣。在一个系统上不断地深入、调试、改善中能获得非常珍贵的、切实的“痛苦”。这些“痛苦”和解决它的渴望才是自我进步提升的根源动力。这些“痛苦”能揭开一切优雅、高级和复杂的技术的面纱，直指它存在的价值。由此，在无人指点的茫然中，我们能自己提出一个个问题，解决它们，从而自我成长。本章用一个小程序的不断完善，展示了开发“能用”程序的力量。在解决因此而生的一个个困难中，我们甚至对软件开发中一些关键技术做了小小的漫游。

3.1 初写系统

写点什么“能用”的程序呢？有个学生说，学了很久的算法、C 语言、Java 语言，程序一关闭，什么都消失了。就写个电话本吧，它必须能将信息写入到硬盘中。本节介绍一些写代码的好习惯，重点展示在编程中学习的方法。

3.1.1 代码风格

先看看下面的两段代码。

DM3-1

(a)

```
bool CWinControl::HandleNeeded(){
    if (m_handle == 0) {
        if (m_parent != NULL) m_parent->HandleNeeded();
            CreateHandle();
    };
}
```

(b)

```
bool myclass::fun1(){
if(h==0){
if(pw!=NULL)pw->fun1();
fun2();
};
}
```

哪个舒服？直觉就是 DM3-1(a)的代码。细细分析有以下几点：

① DM3-1(b)的代码没有缩进，行与行之间都完全是对齐的，让人分不清两个 if 代码块的边界。而 DM3-1(a)就比较清晰，第一个 if 块后退了两格，而其中语句又后退了两格。

② DM3-1(a)的“==”和“!=”两头都有空格，让两个比较符号显得清晰；而 DM3-1(b)没有空格，粘成一块了。

③ DM3-1(a)中的类名前加了大写“C”，所以我们容易知道 CWinControl 是一个类，而 DM3-1(b)的处理很随意。DM3-1(a)中的类名、变量和函数名都很容易读出其含义，如 HandleNeeded 从名字看就知道是一个判断语句，同时两个单词首字母均为大写，两个词一下就被区分开了，也明白了它们的含义“Handle Needed”：是否需要句柄。这样的好处还体现在对整体代码含义的解读上：如果句柄（m_handle）是 0，就创建句柄（CreateHandle），且在创建自己句柄前还要判断自己的父对象（m_parent）是否需要构建句柄，从而让父对象也能有机会创建自己的句柄。（这是一个典型的 lazy load 技巧。）而 DM3-1(b)的代码命名随意，难以理解。

易读、易理解是代码非常重要的特点，否则自己将来都无法维护自己的代码，更不说团队中与

他人协作。

解决该问题就是遵循一定的命名规范（naming convention）。不同的语言都有自己约定俗成的规范。微软的 C++开发用匈牙利命名法（Hungarian naming convention）、Delphi 有自己的命名法、C#用 Pascal 命名法(源于对 Delphi 和 C#缔造者的尊敬)、Java 用驼峰命名法(camel naming convention)。以驼峰命令法为例，如果一个符号（类名、函数、参数和变量名）由多个单词组成，如 handleNeeded，每个词的第一个字母要大写。除了类名外，符号的第一个字母要求小写。对于私有成员变量多数命名法都没有要求，不少人喜欢用下划线“_”开头。DM3-1(a)近似于 Pascal 命名法，DM3-2(a)和(b)分别展示了驼峰命名法和 Delphi 命名法。

DM3-2

(a)

```
bool WinControl::handleNeeded(){
    if (_handle == 0) {
        if (_parent != NULL) _parent->handleNeeded();
            createHandle();
    };
}
```

(b)

```
bool TWinControl::HandleNeeded(){
   if (FHandle == 0) {
       if (FParent != NULL) FParent->HandleNeeded();
           CreateHandle();
   };
}
```

进一步精确按代码规范编程（包括其缩进等格式），大家可直接模仿相关库的源代码，如 Visual C++的 MFC 库（安装 Visual C++后即安装了其源码）采用匈牙利命名法；下载 Java 的源代码即可参考驼峰命名法。鉴于匈牙利命名法太复杂，本书代码借鉴了驼峰命名法的部分习惯。

3.1.2 常量

记得多年前遇到一个在一家大游戏公司做项目经理的学生，他感慨进入公司一年多后最大的进步就是养成了定义宏的习惯。（宏真的这么重要？）设想编写一个处理单词的程序，其功能如下：如果读到单词 hello，就按指定格式改变它。例如，选择大写按钮时，hello 变成大写，选择小写按钮时，它变成小写，按红色按钮时，它变成红色。假设有 3 个源文件 file1.c、file2.c、file3.c，分别将 hello 格式化为大写、小写和红色格式，见 DM3-3。

DM3-3

(a) file1.c

```
word = getAword(readALn(fileRead));
if (word == "hello")
    word = upperCase(word);
...
```

(b) file2.c

```
word = getAword(readALn(fileRead));
if (word == "hello" )
    word = lowerCase(word);
...
```

(c) file3.c

```
word = getAword(readALn(fileRead));
if (word == "hello" )
    word = changeColor(word);
...
```

一切 OK 了，但客户要求变了，不是 hello 而是遇到 love 做格式化。我们不得不在 3 个文件中找到 if 语句中的 hello，并把它改为 love，本例子仅改动 3 个文件可达到目的，如果项目有成百上千个文件，这将是一个巨大的工作量。如果少一个地方没有改到，将引起错误。比如，file3.c 没有被修改，那么被格式化的词还是 hello 而不是 love，这就是 bug。（**我们第一次轻松感受到了修改代码（没有真正改代码）的艰辛!**）有人说，如果用开发环境的查找替换功能，将 hello 统一替换成 love，就轻松无漏的搞定了。是吗？如果一个函数名中有 hello 这个词，如 gethello()，那么自动替换结果就是 getlove，直接导致编译错误。更可怕的是，如果在程序其他逻辑判断某个字符串是否为 hello 并做出相关处理，这个 hello 也会被替换成 love。这时不会有编译错误提示。一个新 bug 就悄悄诞生了。就算每

个替换时都进行人工确认，如果替换的地方多了，确认也会有误差。怎么办？用常量（在 C/C++中是宏，在 Delphi 和 C#中是 const）。在 consts.h 中定义宏#define KEYWORD "hello"，然后在 file1.c、file2.c、file3.c 中引用头文件 consts.h，并用 KEYWORD 替换 hello，见 DM3-4。

DM3-4

(a) consts.h

```
#define KEYWORD "hello"
...
```

(b) file1.c

```
#include consts.h
...
if (word == KEYWORD)
    word =  upperCase(word);
...
```

(c) file2.c

```
#include consts.h
...
if word == (KEYWORD)
    word =  lowerCase(word);
...
```

当关键词变成 love 时，我们只需要修改 DM3-4a 中 KEYWORD 的定义，将 hello 变成 love，然后重新编译即可。一处修改，处处修改。除了维护容易修改方便，常量还利于阅读、理解程序，见 DM3-5。采用宏的 switch 语句（见 DM3-5(a)）至少让别人能猜出它与大小写等处理有关，而 DM3-5(b)就不知所云了。

常量还有一个好处，避免大意的错误。比如，DM3-3(a)中的 hello 的字母“o”写成了数字“0”，编译器并不会报错，但实际处理逻辑错误了。可如 DM3-4(b)和 DM3-4(c)中如果将 KEYWORD 写错一个字母，编译也会报错提醒你。

DM3-5

(a)

```
#define UPPERCASE  1
#define LOWERCASE  2
#define REDCOLOR   3
...
switch(flag) {
    case: UPPERCASE ···
    case: LOWERCASE ···
    case: REACOLOR ···
}
```

(b)

```
switch(flag) {
    case: 1 ···
    case: 2 ···
    case: 3 ···
}
```

3.1.3 最简单的电话簿（1）：功能设计和相关库函数学习

我们要先构想程序的功能，然后根据功能查找相关库函数，并做相应的实验搞清其使用方法。程序的主要功能：① 添加电话信息；② 将电话信息保存；③ 程序运行后，自动显示所有电话信息；④ 用户可将电话信息导出备份。

电话信息现在只包含简单的两个部分：姓名和电话。用记事本可查看的文本文件（*.txt 文件）保存，一行一条电话信息。为了简单，现在用命令行方式编写程序，那么，每个功能需要相关的命令，如下：

- ⊙ 导入电话簿：-ld path。path 代表要导入的电话簿的文件全名（包括路径）。
- ⊙ 导出电话簿：-sa path。path 代表要导出的电话簿的文件全名（包括路径）。
- ⊙ 保存：-se，将信息保存到默认文件中，在执行文件目录下。
- ⊙ 添加：-ad name phone 添加的电话信息。

既然有了命令输入，自然应该有一个循环不断等待用户输入命令并解析，见 DM3-6。现在要看里面哪些功能需要借助库函数支持。第 2 行要等待获取键盘输入，怎么做？第 4 行导入文件，如何读文件？第 7 行导出文件，如何写文件？而且是读写“文本文件”。简而言之就是两个问题：如何获取键盘输入；如何读写文本文件。

DM3-6

```
1  while(1) {
2      等待用户输入并获取输入命令
3      if (命令是 “-ld”) {
4          导入文件;
5      }
6      else if (命令是”-sa”) {
7          导出文件
8      }
9      …
10 }
```

（先解决第一个问题。过去我们会买本书来查阅，但在网络时代就问百度和 Google 吧。当百度查不到时，就用英文 Google 查找英文。）先整理关键字，精确的关键字组合能加快查找过程。因为是编写 C 语言，那么关键字要包含“C 语言”；要获取命令，所以包含“获取命令”。那么，就用“C 语言 获取命令”作为关键字查找。但效果不好。这时要变换查找关键词使其更准确或用同义词替换以精确化搜索结果。“获取命令”是比较模糊的，从哪里呢？是从键盘，所以关键词变成“获取键盘输入”。有的被搜索资料可能没有将“获取”和“键盘输入”两词连在一起，因此将两词间用空格分开更合适。最终关键词变成“C 语言 获取 键盘输入”。效果似乎还是不好。根据前面的原则，我们可尝试更精确的描述，再增加关键词。因为要查找哪个库函数支持键盘输入，所以再加入“函数”关键词，最后查找的输入为“C 语言 获取 键盘输入 函数”。快速扫描查找结果的标题有两条似乎有点靠谱：

（1）如何在 WIN32API 中实现键盘输入?像 C 语言中的 SCANF 函数一样...

（2）C 语言编写程序：通过键盘输入 10 个数，利用三个函数分别...

第 1 个明显提到“SCANF 函数”，第 2 个似乎要从键盘获取 10 个数，与我们的需求相同。点开两个链接，发现均与 scanf()函数相关，特别第 2 个中包含了 DM3-7(a)的代码，明显是从键盘获取整数输入的代码。因此需要进一步查找 scanf 的说明。基本百度百科中就会有。我们依然在百度搜索中输入 scanf。第一条就是百度百科关于 scanf 的内容，关于读入字符串的代码见 DM3-7(b)，用 scanf 读入字符串到数组 str 中，然后打印。阅读说明发现，读入一个字符串时，如遇空格、制表符或换行符函数，就要返回。而按照命令设计，加载命令是-ld path，在-ld 和 path 之间要有空格。显然将在空格处返回，这是不行的。耐心往下看，就发现了关于该问题的说明（如果没有，可以再用百度查找），如 DM3-7(c)黑体语句所示，即可获取包含空格的字符串。

DM3-7

(a)

```
int i ,x;
scanf("%d",&x);
max=x;
```

(b)

```
char str[80];
scanf("%s",str);
printf("%s",str);
```

(c)

```
char str[80];
scanf("%[^\n]",str);
printf("%s",str);
```

现在搭建一个测试程序来实验 scanf()，见 DM3-8，完整程序见“code\第 3 章\3-1\test\testscanf”。令人惊讶的是，输入“123 12”并回车后，程序死循环，不断打印“123 12”。似乎 scanf()读取了“123 12”后并没有清除它，于是一次次读入。不妨求助百度吧，既然似乎没有清除，那么关键字就包含“清除”，如“scanf 清除 上次 读入”。获得的信息中有一条如下：

DM3-8

```
1 while (1) {
2    scanf("%[^\n]", cmd);
3    printf("%s\n", cmd);
4 }
```

“一个用 scanf 出现的奇怪问题，哪位大哥... 显然在上一次输入后没有象正常情况一样清除...”

打开查看，这是一个论坛的帖子，最后提示要在 scanf()前加入 fflush(stdin)，将上次残存在输入队列中的字符清空。似乎很符合我们的情况。我们只需在 DM3-8 第 2 行后插入 fflush(stdin)。测试果然如此。同时发现一个问题，测试是死循环，无法退出。设想中的命令解析代码（见 DM3-6）也是死循环，无法退出。看来应该加入退出命令“-qt”，以便从 while 循环退出。（**看来早做测试有利于发现问题和改善设计**。）

从 DM3-6 第 3 行可知，其逻辑是比较输入串是否与“-ld”相同。所以，在前面的测试代码上加入一个判断，见代码 DM3-9。结果并非如预想一样，输入“-ld”没有打印“ok”。（**哪里有问题？用调试来分析**。）在第 4 行设置断点，输入“-ld”，停在断点后查看 cmd 的值，确实是“-ld”。（**难道程序疯了？**）用反汇编来看第 4 行到底在比较什么，见 DM3-10。第 2 行是将 ebp–28h 赋给 eax，应该就是 cmd 的首址（从 1.3.5 节知，ebp–xx 一般是局部变量的地址），在监视窗体中查看 ebp–28h 和 cmd 的值，发现这一猜测正确。第 3 行将 cmd 的地址与 415740h 比较。“415740h”是谁的地址？直觉认为是字符串“-ld”的地址，因为 if 语句是将 cmd 和“-ld”比较。（**验证**。）在内存窗体中输入 0x415740，可见内存中对应值确实是“-ld”。现在清楚了，用“==”比较两个字符串是比较字符串首址是否相同，而我们比较的是内容，当然逻辑错误了。

如何比较两个字符串是否相同？查查百度，输入“C 语言 字符串比较 函数”，很容易找到关于 strcmp 的说明。其原型是 int strcmp(char * str1, char * str2)，比较 str1 与 str2 是否相同，如果返回值为 0，代表相同，如果返回值大于 0，代表 str1 大于 str2，如果返回值小于 0，代表 str1 小于 str2。那么，用 strcmp 替代 DM3-9，见 DM3-11（strcmp 的头文件是 string.h），工程见“code\第 3 章\3-1\test\testscanf_strcmp”。

DM3-9

```
1 while (1) {
2     scanf("%[^\n]", cmd);
3     fflush(stdin);
4     if (cmd == "-ld")
5         printf("ok\n");
6 }
```

DM3-10

```
1 if (cmd == "-ld")
2     lea     eax, [ebp-28h]
3     cmp     eax, 415740h
4     jne     00413631
5 }
```

DM3-11

```
1 while (1) {
2     scanf("%[^\n]", cmd);
3     fflush(stdin);
4     if (strcmp(cmd,"-ld") == 0)
5         printf("ok\n");
6     else
7         break;
8 }
```

输入“-ld”和其他字符，程序都正确。但先输入“-ld”，回车，再直接回车，此时程序并没有像想象一样因为输入的不是“-ld”而从 DM3-11 的第 6 行跳出循环，居然打印了 ok。通过跟踪当第二次回车时，cmd 中确实还是上次输入的“-ld”。第二次是直接回车，没有输入其他字符，因此此时的 cmd 应该没有字符，也就是说，第一个字符 cmd[0]的值应该是 0（因为 c 语言字符串是以字节值 0 代表结束）。难道对 scanf 而言，直接回车相当于失败，因此它就没有像我们想象一样处理 cmd 中的内容？scanf 应该有返回值表示失败与否？我们之前只管正常功能了，似乎没在意错误处理。回头再看 scanf 的说明，果然它用返回值表示成功读入的变量个数。例如，scanf("%d%d", i1, i2)，如果成功读入两个整数，则返回 2；如果只读入了一个（如 i2），则返回 1；如果一个都没读入，则返回 0。

在 DM3-11 上稍作测试即可发现，当无输入直接回车时，返回值为 0；而敲入几个字符再回车，则返回 1（因为本例中 scanf 只有一个输入）。可见，对字符串而言，如果输入长度为 0（直接回车）就代表没有输入。而 DM3-11 并没有判断这种情况，所以出错。下面我们将要接触到程序开发中一

个重要理念：编写处理错误的代码，往往比编写正常代码更有价值，它决定了程序的健壮性。DM3-11就没有编写处理 scanf 失败的代码，但世界不只按正常状态运行。实际上，很多有经验的招聘者往往更倾向于考察程序员的错误处理意识。修改后的代码见 DM3-12。第 2 行判断返回值是否为 0，如果是，则清除缓冲并重新等待输入。

DM3-12

```
1 while (1){
2     if (scanf("%[^\n]", cmd) == 0) {
3         fflush(stdin);
4         continue;
5     }
6     fflush(stdin);
7     …
8 }
```

程序在实验中一点点成型了，还剩一个重要功能，读写文件。在百度中刚输入“C 语言文件”就提示“C 语言文件操作”，看来相关内容很多啊。我们很容易找到关心的内容。文件必须用 fopen 打开方能操控（读/写）。打开的文件必须用 fclose 关闭，否则会有资源泄漏。文件操控的方式由文件打开方式决定。fopen 函数原型如下：

```
FILE * fopen(const char * pathname, const char * type);
```

其中，pathname 为文件名（包含全路径名，否则将在当前目录下查找）；第 2 个参数为字符串，代表不同的打开方式，如“w”代表创建新文件（如果文件已存在，则覆盖），只能写，不能读，具体见表 3.1。

表 3.1　文件打开权限

限制 \ type	r	w	a	r+	w+	a+
文件必须存在	√			√		
覆盖已存在文件		√			√	
可读	√			√	√	√
可写		√	√	√	√	√
只能写到尾部			√			√

如何读写一行文本文件？fputs 可写一个字符串到文件，原型如下：

```
int fputs(char *string, FILE *stream)
```

string 为要写入的字符串，stream 为被写入的文件指针，由 fopen 返回。

fgets 读取字符串，原型如下：

```
char *fgets(char *buf, int n, FILE *stream)
```

stream 是要读取的文件指针。读取 *n*-1 个指定字符到缓存 buf 中，如遇到换行标记，即使未读够 *n* 个也返回。成功返回后，buf[*n*]的值会被改写为 0，如果失败，则返回 NULL。细心的读者会问，读写文件时，从文件的哪个位置开始？答案就是当前位置。读写成功后，文件当前位置会后移，跳过刚才读入的数据指向其后。当以“w”或“r”方式打开文件时，文件当前位置在起始位置，因此如果要读一个文件，就不断调用 fgets 即可，直到返回为 NULL，即无数据可读。

函数 int feof(FILE* fp)判断文件当前位置是否是尾部，是，则返回 1，否则返回 0。据此，我们编写了测试程序，见 DM3-13，工程见“code\第 3 章\3-1\test\testfile”。

执行的结果失败，C 盘下根本就没有生成 temp.txt 文件。通过调试发现，第 3 行 fopen 返回 NULL，意味着创建文件 temp.txt 失败。C 盘是有空间的，那么应该检查 fopen 的参数是否传递成功（2.6 节就是参数传递错误造成的 bug）。DM3-14 反汇编指示，文件名字符串的地址是 0x420f80，因此用内存窗体观察该地址，结果如图 3.1 所示，根本不是“C:\temp.h”。字符 "\t" 似乎被换成了一个奇怪的符号。

DM3-13

```
1    FILE * f;
2    char buf[30];
3    f = fopen("C:\temp.txt", "w");                    //写方式打开文件
4    if (f != NULL) {                                  //如果文件打开成功，则写入两行字串
5        fputs("han hong", f);
6        fputs("han heee", f);
7        fclose(f);                                    //必须关闭文件，否则资源泄漏
8    }
9    f = fopen("C:\temp.txt", "r");                    //读方式打开刚才的文件
10   if (f != NULL) {                                  //循环读，并打印，直到读完
11       while (fgets(buf, sizeof(buf), f) != NULL)
12           printf("%s", buf);
13       if (feof(f) != 0)
14           printf("********file end********\n");
15       fclose(f);
16   }
```

DM3-14

```
5:  fopen("C:\temp.h", "w");
        push   offset string "w"            (0042001c)
        push   offset string "C:\temp.h"    (00420f80)
        call   fopen                        (004010a0)
        add    esp, 8
// 字符串传递的是该串的指针，也就是说，第二个 push 时传递的是"C:\temp.h"的起始地址，即 0x420f80
```

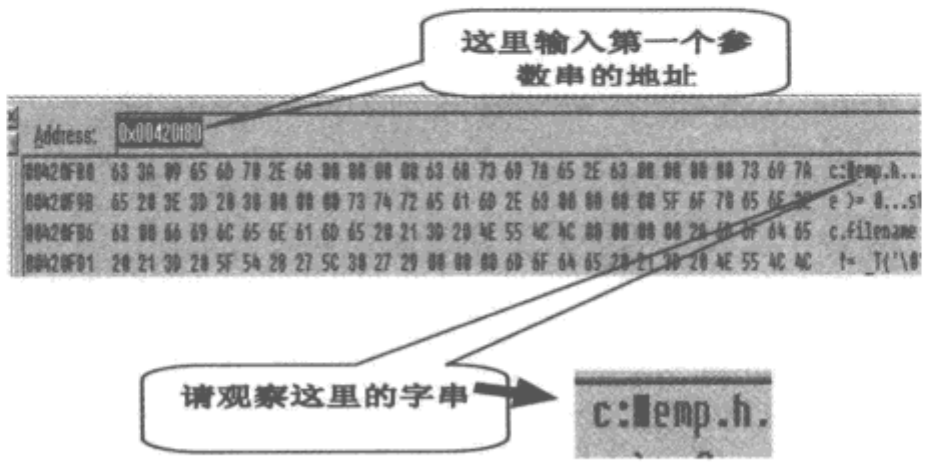


图 3.1　文件名实际值

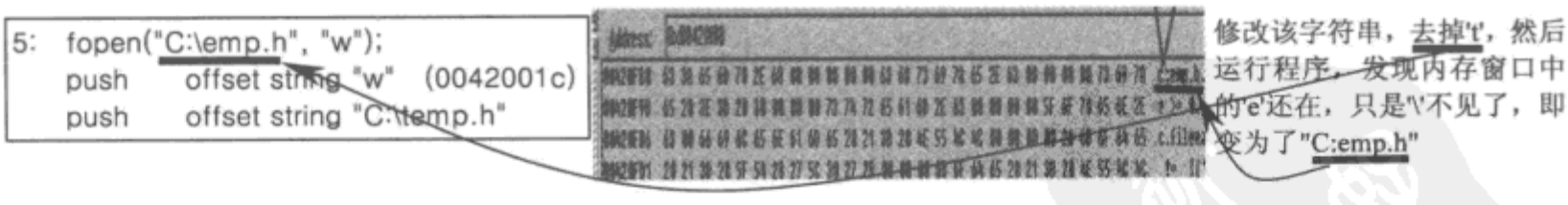


图 3.2　修改后文件名的实际值 1

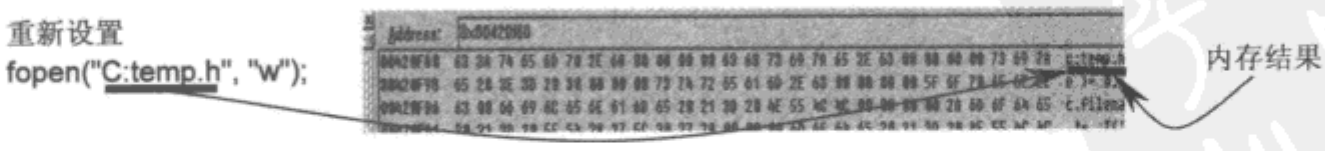


图 3.3　修改后文件名的实际值 2

通过图 3.2 的实验我们发现'\'不见了。'\' 似乎总不见，而其后的字母是否不见则不定。这次去掉 '\' 来验证这一论断，实验结果如图 3.3。内存中字符串变为了"C:temp.txt"。说明，如果无 '\'，后边的 't' 就不受影响。所以，'\' 可能使后边的字符变成另外的内容，如 '\t'（不再是 t）；也可能不变，如 '\e'（依然是 e）。'\' 其实就是转义符，就像 printf()中常用的 '\n'。字符串的 '\' 应该用 '\\' 表示，则 DM3-13 的第 3 行和第 9 行的文件名修改为"C:\\temp.txt"即可。

通过调试分析，看到真实状况，我们能独立分析清晰很多问题（这个“\\”的问题并非笔者杜撰，而是来自初学者多次发生的状况）。

该程序执行，虽然成功，但是文件中并非两行字符串，而是一行。怎样才能写入一行？（**运用猜测和求证的习惯吧**。）读文件的代码怎样才能判断一行结束了呢？可用特殊标记代表换行。常见的符号（大、小写字母和各种符号等）有 94 个，可用 94 个整数值代表。1 字节的取值范围是 0～255，所以只要从中选取一个非 94 字符的整数代表一行结束即可。

下面来证实猜测。用 UltraEdit 打开一个文本文件，按快捷键 Ctrl+H，以十六进制显示，见图 3.4，其中标注的 2 字节 0D 0A 出现在每行的结束，应该是结束标志。在 UltraEdit 中再添加一行字符串，可见行的结束处又多了 2 字节 0D 0A。

十六进制查看

```
68 61 6E 20 68 6F 6E 67 0D 0A 68 61 6E 20 68 65 ; han hong..han he
65 65 0D 0A                                     ; ee..
```

图 3.4　换行标志

可是这 2 字节怎样通过 fputs()写到文件中？什么字符代表换行符？在学习 printf()时我们知道，“\n”代表换行。不妨将 DM3-13 的第 5、6 行字符串的尾部加上“\n”（fputs("han hong\n", f)），结果成功了。但有一个问题，“\n”应该代表 1 字节，为什么写到文件中就变成 2 字节了？先证实“\n”是一个字符，将代码反汇编后，将调用 fputs()的 call 指令前的 push 指令中的地址输入到内存窗体，可见字符串的结尾是 0A，并非 0D 0A。估计是 fputs()在写文件时将 0A 翻译成 0D 0A 再写入。为了证实这一点，将 fputs()写入的字符串尾部的“\n”去掉，然后主动写入 0D 0A，看是否具有相同效果。如 DM3-15，第 2～3 行形成了换行标志 0D 0A，第 6 行将该换行标志写入文件。按照推理，最后写入的应该是 0D 0D 0A，因为 fputs()将 0A 替换为 0D 0A。用 UltraEdit 查看果然如此。

DM3-15

```
1  char retc[3];
2  retc[0] = 0x0d;
3  retc[1] = 0x0a;
4  retc[2] = 0;      //代表字串 retc 结束
5  fputs("han hong", f);
6  fputs(retc, f);
```

Windows 的文本文件换行标志为 0D 0A，而 UNIX/Linux 的换行标志为 0A。这导致 Linux 下生成的文本文件用 Windows 的记事本打开时，没有换行效果，其中还夹杂黑色方块符号。

为了验证文本文件的读写过程，我们能否自己完成一行的读写呢？那就需要找到新的读写函数。在百度中输入“C 语言 文件操作 函数”，然后在给出的函数中可找到函数 fgetc()和 fwrite()。fgetc()的参数为 FILE *，从文件当前位置读取 1 字节。fwrite()的功能是把 ptr 指向的缓冲中的数据写到文件指针 fp 指向的文件中，写入字节为 nobj 乘以 size。其中，nobj 是数据块个数，size 是数据块大小，返回实际写入的数据块数目（换算成字节数要乘以 size）。例如，size_t fwrite(void *ptr, size_t size, size_t nobj, FILE* fp)中的 size_t 就是 int。

我们编写的读取一行的函数如下：

```
int readLn(FILE * fp, char * ln)
```

从 fp 中读入字符串到 ln 中，返回值为读入的一行字符串的长度，不含换行符，代码见 DM3-16。

通过简单实验知道，'\r'就是 0x0d。函数逻辑就是一个一个读入字符，寻找 0x0a（即'\n'），然后判断是否一行结束。其中，第 14～19 行判断 UNIX 还是 Windows 风格的文本文件，计算一行的长度。

用 readLn 和 fwrite 修改 DM3-13 的代码，见 DM3-17，工程见“code\第 3 章\3-1\test\testtxtfile”。第 7 行和第 9 行写入 0x0d 0x0a，即 Windows 的换行标志。

DM3-16

```
1    int readLn(FILE * fp, char * ln) {
2        int c, ret;
3        ret = 0;
4        while (1){
5            c = fgetc(fp);    //读入 1 字节
6            if (c == EOF){    //如果遇到文件结束
7                *ln = 0;      //将字串最后填写 0
8                return ret;
9            }
10           *ln = (char)c;    //将读到字节写入 ln 中
11           ln++;             //后移动指针，下一次读入
12           ret++;            //并将读入长度加 1
13           if (c == '\n'){   //读到 0x0a，则遇到换行
14               if (*(ln - 2) == '\r'){//处理 windows
15                   *(ln - 2) = 0;  //风格的文本文件
16                   ret--;
17               }
18               else
19                   *(ln - 1) = 0;  //否则就是 UNIX 文本
20               return ret;
21           }
22       }
23       return ret;
24   }
```

DM3-17

```
1   char retc[2];
2   retc[0] = 0x0d;
3   retc[1] = 0x0a;
4   f = fopen("C:\\temp.txt", "w");
5   if (f != NULL) {
6       fwrite("han hong", strlen("han hong"), 1, f);
7       fwrite(retc, 2, 1, f);
8       fwrite("han heee", strlen("han heee"), 1, f);
9       fwrite(retc, 2, 1, f);
10      fclose(f);
11  }
12  f = fopen("C:\\temp.txt", "r");
13  if (f != NULL) {
14      while (readLn(f, buf) != 0)
15          printf("%s\n", buf);
16      if (feof(f) != 0)
17          printf("***file end***\n");
18      fclose(f);
19   }
20  }
```

运行结果出乎意料，通过 UltraEdit 查看，DM3-17 中第 7 行依然在字符串的最后写入的是 0d 0d 0a。很明显，通过 fwrite()写入的 0x0a 依然被转成了 0d 0a。难道 C 语言读写文件还区分文件的模式，是文本文件就将一些特殊字符转换？如果有设定模式的地方，又在哪里？在打开文件的代码处最适合指定模式，在 fopen()中吗？百度中查找到的说明没有这方面信息，这时应该求助往往更全面的英文文档。

在中文搜索效果不佳时应该用 Google 尝试英文搜索，且在搜索界面中选择 search in English（即强制搜索英文文档，否则搜索以中文为主），往往能搜索到许多中文完全没有的内容。

通过 Google 英文搜索 fopen()，查到了以下资料：

> With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a "b" character has to be included in the *mode* string. This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

它明确指出，如果在模式中加入 b 或者 t 参数，就会指定 fopen()以何种形式打开文件，前者为二进制，后者为文本；同时，对 UNIX（包括 Linux）操作系统，两种模式没有区别，虽然一般不特别指出就是文本模式，但为了 C 语言代码可以不加修改地在不同操作系统上移植，即使是文本模

式也最好指明是 t 模式。这时只要修改 DM3-17 的第 4 行为“fopen("C:\\temp.txt", "wb");”，指明为二进制模式即可。实验结果完全符合预期。

我们在文件读写上花在猜测、思考和实验方面的工作表面看似乎太繁琐，不如直接用 fputs()和 fgets()编写快速，但这样做是值得的：对于一些“所以然”的认知（即文件模式、'\r'的处理等）是有回报的，能解决很多麻烦的 bug。有了文件模式的认知，3.2 节的代码就不会出错，否则出现的 bug 将很难追踪。

另外，本节展示了一种在开发中学习的过程，强调了查找资料、测试、猜测和实证的过程，展示了调试分析在其中扮演的重要角色。这一过程中还接触到两个重要的开发理念：① 尽早测试；② 要严格编写错误处理代码，能避免很多 bug 及由此导致的调试工作。

3.1.4 最简单的电话簿（2）：系统实现，分割函数

经过 3.1.3 节的探索学习，我们终于可以开始系统构建了。可在 main()函数一开始，我们就必须停下了。电话本功能（3.1.3 节开始处）要求“自动加载默认电话文件，并在输入“-se”命令时将信息保存到默认文件中”。可是，这个默认文件放在哪个目录中？放在与执行程序相同的目录是一个自然的选择。但是怎样知道执行程序所在的目录？在 1.8.1 节知道，main()函数接收的命令行参数的第一个参数是执行程序的名字。如果启动程序时用了包括目录的全名，如“E:\myprg\debug\my.exe”，那么参数值就是“E:\myprg\debug\my.exe”；如果当前目录在执行程序所在目录（E:\myprg\debug）中，那么“E:\myprg\debug>my.exe”可以启动程序。此时，从 main()函数中取得的参数就是“my.exe”。如果启动时不跟后缀名（系统会自动查找后缀为 .exe 的文件），如“E:\myprg\debug>my”，则该参数就是“my”。因此，我们或者能从该参数获得执行程序所在目录，或者当前目录就是执行程序所在目录（定义了环境变量指向执行程序所在目录则是一种例外）。这涉及如下问题：fopen()中输入文件名参数，在不带路径的情况下，是否就是指的执行程序所在的当前目录，如果是，则能成功打开文件。

DM3-18 为通过 main()参数获取执行程序路径的代码，工程见“code\第 3 章\3-1\test\testfiledir”，且在执行程序所在目录下事先生成文本文件 temp.txt。第 3～8 行从字符串尾部查找是否第一个参数中包含目录分隔符'\'。如果找到，i 就是最右边'\'的索引，否则 i 小于 0。当用户在执行程序所在目录启动程序时，由于没有输入路径名，所以进入第 10～12 行，此时 fopen()打开文件名中没有包含路径，仅仅是“temp.txt”。如果用户启动程序时包含了程序的路径，则第 14～23 行用包含路径的文件名打开文件。在控制台界面中分别将两种情况都试验，可证明按第 10～23 行的逻辑总能正常打开文件。至此，我们终于将程序涉及的所有关键技术点都试验完毕了（其实当前目录并非一定就是执行程序所在目录，请参考习题 14）。

DM3-18

```
1   printf("第一个参数是: %s\n", args[0]);
2   i = strlen(args[0]) - 1;
3   while (i >= 0){
4       if (args[0][i] == '\\') {
5           break;
6       }
7       i--;
8   }
9   char * fileName = "temp.txt";
10  if (i < 0) {
11      printf("第一个参数无路径\n");
12      fp = fopen(fileName, "r");
13  }
14  else {
15      char file[256];
16      memset(file, 0, sizeof(file));
17      strcpy(file, args[0]);
18      file[i] = 0;
19      printf("dir is %s\n", file);
20      file[i] = '\\';
21      strcpy(file + i + 1, fileName);
22      printf("open file %s\n", file);
23      fp = fopen(file, "r");
24  }
25  ...                              //打印整个文件内容
```

如何来组织代码？我们需要把握一条简单的原则：分割函数，甚至用函数来代替注释。运用得好，就能让代码井井有条。这一原则来自极限编程（在有的领域，其他因素更重要，这一原则并非完全适应）。还有一个经验原则：当一段代码复用 3 次时，就将其变成函数。初学者当然无法一下就想到所有的功能及对应的函数。不妨先在主函数中写伪代码，之后整理时，看哪些语句出现多次，然后将它们定义成函数名，再根据其需要传递的信息，自然就定义出它的参数。熟练后，根据功能分解就能直接定义出函数了，见代码 DM3-19。第 3 行和第 11 行都有“加载电话本信息”，就可以用 loadPhoneInfo()来替换。第 9、16、23、30 行都有“展示命令用法”，可用 usage()替换。这样就有两个函数了。

usage()只是打印函数用法，不需参数和返回值，所以其原型为 void usage()。loadPhoneInfo()的功能是从文件中读入每一行的电话信息，然后打印，因此需要文件名作为参数，而且从代码看，一个需要传入默认文件加载，一个需要传入用户指定的文件加载。暂时看不到其他重复的语句，那么可暂且不管，在编写代码时，如果重复，就变成一个函数。典型的例子是第 7、14、21 行，是从用户输入的命令中取出其参数，初看没有头绪，无法形成函数，但因为命令本身都是占 3 个字符，格式相同（命令后是参数），所以最终在编写代码时能够提炼出一个函数 getArg()。最后，应用“与其写注释，不如写函数”原则，有的代码似乎只出现了一次，但也被变成了函数，如获取默认电话簿文件名的代码（函数 getDefaultFilePath()）和判断默认文件是否存在的代码（函数 hasDefaultPhoneInfo()）。DM3-20 给出了两种方式的对比。

DM3-19

```
1   获取默认文件全名
2   if (默认文件存在否)
3       给定默认文件名，加载电话本信息
4   while(1) {
5       获取用户输入命令
6       if (命令是 "-ld") == 0) {
7           获取被加载文件名
8           if (未获取到) {
9               提示错误，并展示命令用法
10          }
11          给定文件名，加载电话本信息
12      }
13      else if (命令是 "-sa")) {
14          获取被存储的文件名
15          if (未获取到) {
16              提示错误，并展示命令用法
17          }
18          存储文件
19      }
20      else if (命令是 "-ad")) {
21          获取被加入的电话信息
22          if (未获取到)
23              提示错误，并展示命令用法
24          添加电话信息
25      }
26      else if (命令是 "-qt")) {
27          退出
28  }
29  else
30      展示命令用法
```

DM3-20

(a)

```
getDefaultFilePath(argv[0]);
if (hasDefaultPhoneInfo()) {
    loadPhoneInfo(gPhoneInfoPath);
}
...
```

(b)

```
//------获取默认文件名-------
int i;
i = strlen(arg[0]) - 1;
strcpy(gPhoneInfoPath, arg[0]);
while (i >= 0)      {
    if (gPhoneInfoPath[i] == '\\'){
        i++;
        break;
    }
    i--;
}
if (i < 0) {
    strcpy(gPhoneInfoPath, EFAULT_PHONE_INFO);
}
//make the path
strcpy(gPhoneInfoPath + i, DEFAULT_PHONE_INFO);
FILE * p = fopen(gPhoneInfoPath, "r");
if (p != NULL){          //如果默认文件存在，加载
    fclose(p);
    loadPhoneInfo(gPhoneInfoPath);
}
```

DM3-20(a)的代码逻辑非常清晰，只要会英语基本能知道大概。DM3-20(b)的代码逻辑即使有注释，开发者也只有先认真界定边界，才能看出代码的主体逻辑。简单地说，前者像流程图，没有细节，一下就看见了主体；后者将细节和主线混在一起，纷繁难理。

DM3-21

```
getDefaultFilePath(argv[0]);
if (hasDefaultPhoneInfo()) {
    loadPhoneInfo(gPhoneInfoPath);
}
while(1) {
    printf("please input:");
    scanf("%[^\n]", cmd);    //用户输入命令
    fflush(stdin);
    char oldC = cmd[3];
    cmd[3] = 0;              //0 结尾才能用 strcmp 比较
    if (strcmp(cmd, "-ld") == 0) {
        cmd[3] = oldC;
        arg = getArg(cmd + 3);      //获取文件名
        if (arg == NULL){
            printf("lose load file arg\n");
            usage();                //打印正确用法
            continue;
        }
        loadPhoneInfo(arg);  //加载指定电话簿
    }
    else if (strcmp(cmd, "-sa") == 0) {
        ...
    }
    else if (strcmp(cmd, "-ad") == 0){
        ...
        addPhoneInfo(arg);   //添加电话信息
    }
```

有了 3.1.3 节的实验，其他函数的逻辑均较清晰。将 DM3-19 的伪代码细化为真实代码，见 DM3-21，工程见“code\第 3 章\3-1\PhoneNote1”。（scanf 语句后的“cmd[3] = 0;”需解释。因此涉及 C 语言串操作的相关注意事项将在 3.1.5 节分析）。另外，C 语言中没有布尔类型，一般用 0 代表 false，用 1 代表 true。但为了代码可阅读性，进行了如下定义：#define BOOLEAN int，#define TRUE 1，#define FALSE 0。这是 C 语言中常用的一种方式。

下面指出一个非常重要的原则：重复就是罪恶。为什么前面的函数虽然只有几行，一旦重复就要抽取成一个函数？loadPhoneInfo()使用了两处，如果没有抽取成函数，相当于一模一样的代码存在于两个地方。一旦函数逻辑修改，则两处代码都必须修改。如果调用处有 100 处，就得改 100 次！这个枯燥的过程可能出错。更麻烦的是，必须找到这 100 处。**（如果没找到呢？）**所以，一旦出现重复的代码，往往就是程序修改维护的噩梦。

重复即罪恶

3.1.5 空字符结尾串的警觉

先不管 3.1.4 节 DM3-21 的 cmd[3] = 0 是何意，万事应追根溯源，先思考，怎样在内存中表示字符串？这与 3.1.3 节中分析如何表示一行文本一样。既然 1 字节能表示 256 种可能，而文本可见字符数目小于 256 个，那么我们可以用 1 字节（非可见字符）来表示字符串结束。C 语言选用了 0（从图 3.3 的内存窗体中可证明）。字符串"C:temp.h"共 8 字节，内存窗体第 1 行的第 9 字节的值就是 0。图 3.5 为示意图。还有一种方式是直接给出字符串的长度，见图 3.6。该字符串为"yes"，其地址指向'y'的开始，而字符串地址减 4 字节指向的内存存储了字符串的长度 3。采用该方式的语言有 Delphi（其结构还要复杂些）。**（两种字符串表示方式的优劣请自己做练习题 1。）**

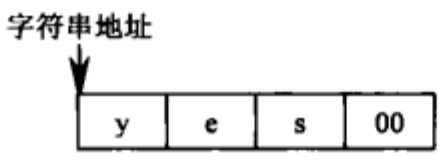


图 3.5 字符串地址（一）

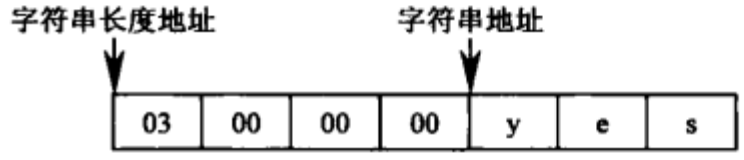


图 3.6 字符串地址（二）

以下是 C 语言中字符串函数要注意的问题：

① strlen()返回的长度不包括结束的 0 字节，所以图 3.5 所示字符串的长度是 3 不是 4。而用 strcpy()、strcat()等函数复制或连接字符串时，这些函数会自动将最后的 0 字节写入目标缓冲，因此我们分配内存存储字符串时，要多 1 字节，即 malloc(strlen(str1) + 1)。因为所有字符串函数是依赖字符串最后字节为 0 界定长度的，所以 DM3-21 中的“cmd[3] = 0”就是将命令串中命令部分（正好长度为 3 字节，如“-se”）的尾部即第 4 字节设定为 0，这样下一条语句才能用 strcmp()正常工作**（如果它不知道被比较字符串的长度，又怎样比较两个字符串是否相同？）**。

② 安全问题，可以想象，如果一个被复制的字符串没有 0 字符结尾，那么 strcpy()就会不断复

制下去（只有遇到 0 字节才会停止）。这样迟早会超过目标缓冲的边界，就会引发一种破坏力非常强大的攻击——溢出攻击。解决办法是采用带字母“n”的安全版本的字符串函数。如 strncpy()多了一个参数，指出目标缓冲的大小，当复制长度达到该大小时，即使没有遇到 0 字节，也停止复制。

③ 比较隐晦的问题：“字符串函数只能处理以 0 字节结尾其他字符不为 0 的数据”。本来有了前面两条论述，这一条似乎画蛇添足，我们不妨问一个问题：如果通过抓包软件从网上收一个 IP 报文，其内容放入缓冲区 char * frame，那么，能否用 strlen(frame)来判断报文长度？答案是“否”，因为该缓存能放入任意字节组合的数据，中间很可能出现 0。用 strlen()就会少计算报文长度。笔者当年用 winpcap 抓包并在 Delphi 中处理时，就疏漏了，其结果导致一个极难发生且难分析的 bug。程序连续运行几天后会出现内存访问违规错误，直到后来分析了 C 语言字符串的原理后，才发现了该错误。可是，为什么会发生内存访问违规？调用 strlen()都可发生这种错误。这里涉及 1.8.1 节提到的虚存管理系统。在操作系统中，物理内存分配是按页分配的（Windows 页的默认大小为 4096 字节）。假设一个报文的内容在某一页内存中，但其中没有一个字节为 0，那么 strlen()判断时，就会一直往下读，直到地址超过该页内存。假定该页最后字节的地址是 0x400000，strlen()因为没读到 0，就会读取线性地址 0x00400001。（当然，实际 C++的算法是一次读 4 字节，见 1.9 节。）又假定这个线性地址上并没有真正分配物理内存，那么读该地址就在读取无效内存，自然会出内存访问违规错误。

DM3-22 给出了示例 strlen 调用出错的代码，工程见“code\第 3 章\3-1\strlenError”。其原理是先用 VirtualAlloc 分配 2 个地址连续的内存页面，将第一个页面（m1 指向页面）全赋值为 1，并将第 2 页（m2 指向页）第 1 字节赋值为 0（m2[0] = 0）。那么，strlen()读完第一页将读取到 m2[0]，从而返回。将 m2 用 VirtualFree()释放，再次调用 strlen()时，读取 m2[0]对应内存必然导致内存访问违规。

3.1.6 让程序更有组织性

如果我们审视自己的电话簿代码，发现它完全放在一个文件中。就像你将所有东西堆在一个房间里一样，有些没条理。下面按照功能，将相关函数放在一个 C 文件中，便于复用和维护，然后通过同名头文件暴露出来，供其他程序调用。我们分解出下面几个文件：① PhoneInfo.h、PhoneInfo.cpp，支持操作电话簿相关的函数和数据结构；② types.h，因类型如自定义 BOOLEAN 在其他工程也会重用，所以为独立的头文件；③ utils.h，utils.cpp，如 readLn，getArg 等函数也可被其他工程重用，所以独立出来；④ PhoneNote1.cpp，包括主函数。项目结构见图 3.7，工程见“code\第 3 章\3-1\PhoneNote_good_prj_err”。

因为头文件可能被多个其他文件 include，可能出现下面状况：b.h 引用 a.h，c.h 引用 a.h，d.h 引用 b.h 和 c.h。而 include 相当于将被引用文件复制、粘贴进去，那么 d.h 在复制 b.h 时将 a.h 复制了一次，在复制 c.h 时又将 a.h 复制了一次。假定 a.h 中定义了一个变量 gInt，对 d.h 而言，相当于定义了两次 gInt，必然导致错误。用条件编译可解决该问题，DM3-23 展示了 utils.h。第 1、2 和 6 行是条件编译（见 5.3.2 节），头文件的有效内容放在其中。第 1～2 行表示，如果没有定义宏 UTILS_H_XXXXX，就定义该宏，且第 3～5 行才存在。因为 UTILS_H_XXXXX 已定义，下次出现这 6 行代码，第 2～5 行对编译器而言就不存在，自然不会发生重定义。这个宏必须是整个项目中独一无二的，否则会出错，有一种做法是用 guid（全球唯一的编码号）。

因为 main()函数所在的 PhoneNote1.cpp 需要使用全局变量 char gPhoneInfoPath[100]，而 PhoneInfo.cpp 中的代码也要使用该变量，所以将其定义在 PhoneInfo.h 中。编译时报错：变量 gPhoneInfoPath 重定义错误，见图 3.8。

DM3-22

```
SYSTEM_INFO sSysInfo;
GetSystemInfo(&sSysInfo);                                    //获取系统信息.
int pageSize = sSysInfo.dwPageSize;                          //获取页大小
char * m1, *m2;
m1 = (char *)VirtualAlloc(0, pageSize * 2, MEM_RESERVE, PAGE_READWRITE);
if (m1 == NULL) {
    printf("fail to reserve\n");
    return;
}                    //下面函数将两页内存从保留状态变成提交状态，因此程序可以访问这两页内存
if (VirtualAlloc(m1, pageSize * 2, MEM_COMMIT, PAGE_READWRITE) == NULL) {
    printf("fail to commit\n");
    return;
}                    //虽然 m1 指向页的字节全为 1，strlen()将读取 m2 指向页，但 m2[0]=0 为 strlen()界定了长度
memset(m1, 1, pageSize);
m2 = m1 + pageSize;
m2[0] = 0;       //当 strlen 读取 m2 指向内存时，将遇到 0，从而停止
strlen(m1);
printf("first strlen ok\n");
if (!VirtualFree(m2, pageSize, MEM_DECOMMIT)) {              //m2 指向的页释放，导致读该块内存失败
    printf("cannot free\n");
    return;
}
strlen(m1);   //该次调用将出现异常，如果在 Visual C++中调试可见提示为 "内存访问违规"，错误访问的地址正是 m2
```

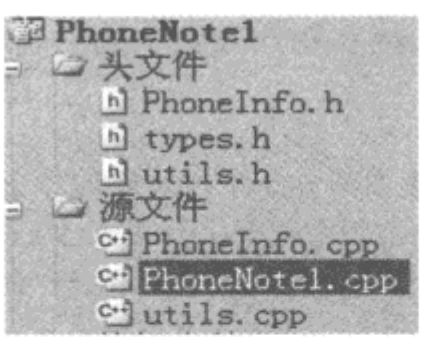


图 3.7　工程结构

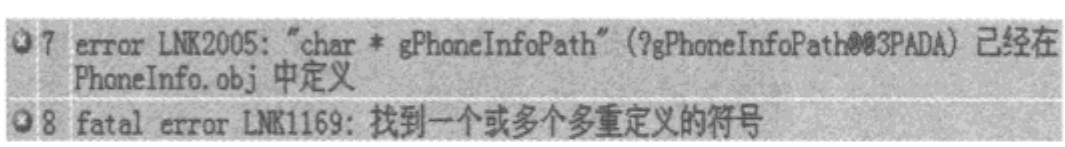


图 3.8　重定义错误

为什么会有这个错误？用我们的原则："我来做，将怎么做"来分析编译器的工作过程。编译器会先将 CPP 文件编译成对应的 OBJ 文件，通过链接，将它们链接为一个 EXE 可执行文件。在"code\第 3 章\3-1\PhoneNote_good_prj_err\PhoneNote1\Debug"下可看到 PhoneNote1.obj 和 PhoneInfo.obj，提示链接时（LNK2005）重定义。原来 PhoneInfo.cpp 和 PhoneNote1.cpp 引用了 PhoneInfo.h，而该头文件又定义了全局变量 gPhoneInfoPath。因此相当于 PhoneNote1.cpp 和 PhoneInfo.cpp 中都定义了该变量。各自的 OBJ 文件中应该都包含了符号 gPhoneInfoPath。用 UltraEdit 打开两个 OBJ 文件均可查到字符串"gPhoneInfoPath"。这样当链接 OBJ 文件时，因两个 OBJ 文件中都有"gPhoneInfoPath"，链接自然报错。图 3.8 的第二个错误提示，"找到一个或多个多重定义的符号"，我们隐隐猜到：链接时为了定位一个符号（变量，函数名之类），将在所有的 OBJ 文件中查找，查找谁到就是谁。所以，不能有两个 OBJ 文件包含相同符号定义。

DM3-23

```
1  #ifndef UTILS_H_XXXXX
2  #define UTILS_H_XXXXX
3  #include <stdio.h>
4  int readLn(FILE * fp, char * ln);
5  char * getArg(char * start);
6  #endif
```

问题分析清楚了，怎么解决呢？只要在头文件 PhoneInfo.h 中定义的变量前加入 extern 即可，extern char gPhoneInfoPath[100]，这样即使其他 CPP 文件引用了该文件，编译器就知道该 CPP 文件不会在其 OBJ 文件中分配 gPhoneInfoPath，就不会出现重定义错误。但是，总得有一个 CPP 要定义该变量，否则所有 OBJ 文件中都没有该符号，链接时又要报错："找不到 gPhoneInfoPath"。我们选

择在 PhoneInfo.cpp 中定义该变量，工程见“code\第 3 章\3-1\PhoneNote_good_prj”。

3.2 有序的世界：可测试与可跟踪的系统

本节通过软件开发中功能扩展和优化，展示程序的一些重要基本元素，并通过软件变化造成的困难，自然引入单元测试和日志。

3.2.1 电话簿扩展（1）：硬盘结构体数组

3.1 节的实现还不能查找电话信息，如输入名字查号码、输入号码查名字。怎样做？我们现在是采用文本文件存储电话信息。从格式上讲，信息的结构很松散。比如，“小李␣ 81102567”也可以存成“小李 81102567”，即姓名和号码之间没有空格，甚至“81102567 小李”，这样在实现查询功能时程序处理逻辑就不好控制。如果电话簿信息中将来再存储更多信息，如地址、邮编等，就很难处理了。用文本存储信息的一个缺点是比较浪费空间。例如，整数 1000000 用文本要 7 字节（0x31 0x30 0x30 0x30 0x30 0x30 0x30），用二进制存储只需 4 字节（0x000f4240）。另外，如果这个整数要进行加、减、乘、除等运算，就需要将字符串"1000000"转换成 4 字节表示的整数。这要花时间，特别数据多时。如果直接在硬盘上用 4 字节存储（1000000 就是 0x40 0x42 0x0f 0x00，小端机表示），则读入时不需做任何处理。

> 文本表示也有优越性，如任何体系结构的硬件均为同一表示。“1000000”的表示均为 0x31 0x30 0x30 0x30 0x30 0x30 0x30。而用 4 字节整数的二进制方式，大端机为 0x00 0x0f 0x42 0x40，小端机为 0x40 0x42 0x0f 0x00。这一优势特别体现在不同硬件、软件系统间交换数据时，所以现在用 XML 文档作为交互协议比较流行。另外，文本表示不仅机器能理解，人也很容易理解。这就是 UNIX 坚持文本化输入和输出（相对 Windows 图形界面而言）的原因：方便人们自由地将不同程序的文本输入、输出用管道串成更大和更复杂的程序。

我们将电话信息改为二进制存储。但如何解决信息的结构问题，从而便于查询呢？一个自然的想法是用结构体表示，如

```
struct PhoneInfo{
    char name[30];
    char phone[30];
};
```

那么，在内存中就容易编写查询逻辑，如将所有电话信息存储在一个 PhoneInfo 数组 gInfo 中，遍历该数组比较成员变量即可，如 if (strcmp(gInfo[i].name, queryName) == 0)。现在的问题是，如何将结构体存储到硬盘？从 1.4.2 节知道，结构体就是一块连续的内存。fwrite()是指定内存地址和写入长度，将内存值写入到硬盘上。那么，我们可以直接将结构体写入到硬盘。内存地址就是结构体的指针，长度就是 sizeof(PhoneInfo)，代码即 fwrite(&gInfo[i], sizeof(PhoneInfo), 1, fp)。同理，只要文件当前位置是硬盘上结构体的起始位置，用 fread()就可将结构体直接从文件读入内存中的一个结构体中，即 fread(&gInfo[i], sizeof(PhoneInfo), 1, fp)。3.1.3 节结尾处已知，如果用文本模式读写，会把'\n'转义变成"\r\n"，所以用二进制模式打开，见 DM3-24，工程见“code\第 3 章\3-2\test\structfile”。

仔细分析 DM3-24，要应用于电话本工程还有些问题需要解决：① 如何知道硬盘数组有多长？② 读入内存的数组必须有确定的长度，如果硬盘数组的长度超过了内存数组的长度怎么办？

对于问题①，一直读直到文件结尾就可以了，读到几个就有多长。对于问题②，我们先简化处

理，定义一个固定长度的内存数组，只要限制存取不超过边界就可以了见 DM3-25。

DM3-25(a)是加载电话簿信息。第 6 行做循环读，如果没到结尾就添加电话簿信息。第 10 行如果判断是否成功读入了整个结构体。第 13～15 行成功读入后打印并修改相关信息。gCount 为全局变量，指示当前有多少条信息。pCurrentInfo 指向当前可存放信息的数组元素。在开始时，gCount 为 0，pCurrentInfo 指向全局数组的起始元素。因此每加载一条，这两个变量都要加一。第 5 行和第 7 行涉及的 initInfo()和 canAdd()两个函数充分体现了分割函数的思想。两者都非常小，一个有两行代码，一个只有一行。虽然小，但是完成了一定功能，我们还是将其独立成函数。如将来变化也便于修改，比如要修改是否可添加的逻辑就只需修改 canAdd()。initInfo()用于初始化数组的相关信息，所以将 gCount 清零，并让 pCurrentInfo 指向全局数组的起始位置 gPhoneInfoList。

DM3-24

```
FILE * fp = fopen("C:\\temp.dat", "wb");
PhoneInfo info[2];
info[0].no = 10;
strcpy(info[0].name, "h1");
...
for (int i = 0; i < 2; i++) {
    fwrite(&info[i], sizeof(PhoneInfo), 1, fp);
}
fclose(fp);
fp = fopen("C:\\temp.dat", "rb");
memset(info, 0, sizeof(info));
for (int i = 0; i < 2; i++) {
    fread(&info[i], sizeof(PhoneInfo), 1, fp);
    printf("no: %d\n", info[i].no);
    ...
}
fclose(fp);
```

DM3-25(a)

```
1   void loadPhoneInfo(char * file) {
2       ...
3       fp = fopen(file, "rb");
4       ...
5       initInfo();
6       while (feof(fp) == 0) {
7           if (canAdd() == FALSE){
8               printf("长度超界%d\n", MAX_PHONE_NUM);
7               break;
9           }
10          if (fread(pCurrentInfo, sizeof(PhoneInfo), 1, fp) == 0) {
11              break;
12          }
13          printf("%s %s\n", pCurrentInfo->name, pCurrentInfo->phone);
14          pCurrentInfo++;
15          gCount++;
16      }
17      fclose(fp);
18  }
```

DM3-25(b)定义了函数 initInfo()。DM3-25c 中涉及宏 MAX_PHONE_NUM，它定义了数组的最大长度。我们用该宏定义数组 PhoneInfo gPhoneInfoList[MAX_PHONE_NUM]。其好处是，需要修改其大小时，只需修改宏，那么数组大小、与其大小相关的代码都自动修正，使得代码便于维护。

DM3-25(b)

```
void initInfo(){
pCurrentInfo = gPhoneInfoList;
    gCount = 0;
}
```

DM3-25(c)

```
BOOLEAN canAdd(){
    return gCount<MAX_PHONE_NUM ? TRUE: FALSE;
}
```

DM3-25(d)

```
1   void addPhoneInfo(PhoneInfo info) {
2       FILE * pf;
3       if (canAdd() == FALSE){
4           printf("cannot add any more\n");
5           return;
6       }
7       pf = fopen(gPhoneInfoPath, "ab");
8       ...
9       fwrite(&info, sizeof(PhoneInfo), 1, pf);
10      * pCurrentInfo = info;
11      pCurrentInfo++;
12      gCount++;
13      fclose(pf);
14  }
```

DM3-25(d)添加一条信息。第 3 行重用了 canAdd()函数，虽小但也展示了重用的力量。试想，如果我们将来提供内存动态数组，没有了大小的限制，只需要将 canAdd()直接返回为 TRUE，而第 3 行与 DM3-25(a)的第 7 行根本不用修改。第 7 行是添加模式打开文件，这样记录就会写入文件尾

部，之前的内容不会被覆盖。第 9 行将结构体 info 写入文件。第 10 行将添加的内容也写入到内存数组当前元素。第 11～12 行改变数组的状态，类似 DM3-25(a)的第 14～15 行。整个项目见“code\第 3 章\3-2\PhoneNoteFixedArray”。

在结构化存储的基础上，代码实现查询功能就非常简单，请查阅工程项目。

3.2.2 指针的陷阱

3.2.1 节中遗留了一个问题：存储信息的内存数组是静态的，只能在编译期指定大小，可否提供一个随需求变化大小的数组呢？从原理上讲（见 1.4.1 节），数组就是一块连续的内存。只要能动态分配一块内存就可以做一个动态数组。从第 1 章的分析中我们知道，局部变量在栈上分配，全局变量在执行文件的数据段中分配，加载时直接导入对应内存。两者都是编译器、加载器等控制，与编程者无关。因此，无论是操作系统还是 C 库，都提供了相应的机制让编程者自己控制内存分配。DM3-22 中的 VirtualAlloc()是操作系统提供的内存分配函数（实质按页分配，粒度太大），而 C 库利用操作系统的函数提供了更细粒度的分配能力：malloc()和 free()。它们带给程序莫大自由，也带来深重的苦难。

这两个函数代表内存分配的两方面：malloc()分配内存，free()释放内存。如果一个程序只知道分配内存而不释放，那么最终内存会耗尽，导致程序越来越慢，无法工作，并累及其他程序无法运行，只有关闭这个问题程序才能恢复。该问题称为内存泄漏（精确的描述后面再讨论）。内存泄漏好解决吗？似乎记住释放就可以了。但事情远没这样简单：其一，人总有记不住的时候，在数百甚至上千、万行代码中找到泄漏点是非常困难的；其二，就算不会忘记，但可能出现错误释放，却是记忆无法解决的问题。当一块内存被多处使用，一处释放了，而另一处还要使用，就会导致错误，我们称为指针悬空。可以说，C/C++程序中非常棘手的 bug 相当部分都与内存的这两个问题相关。为了解决这两个有点矛盾的问题，才有了垃圾回收技术（其中 Java 在这方面比较著名，当然它并非鼻祖），让程序员脱离了苦海，只管分配，有人来做释放。

DM3-26

(a)

```
void test(){
    int * p = (int *)malloc(sizeof(int));
}
```

(b)

```
int * p;
void test(){
    p = (int *)malloc(sizeof(int));
}
```

只分配内存不释放就是泄漏吗？如果确实需要内存，当然不能释放，这总不能算泄漏吧。这里的关键还不是“是否”释放，而是“能否”释放，见 DM3-26(a)。

malloc()分配的内存地址只赋值给了局部变量 p，test()返回后，就无法访问 p 了，自然无法获取到保存在 p 中的新分配内存地址。没有该地址，又如何通过调用 free()传入该地址来释放这块内存？即使想释放，也无法释放了，这才是内存泄漏的关键。而 DM3-26(b)就不能说一定有泄漏，因为虽然 test()中没有释放，但内存地址保存在全局变量 p 中，那么其他代码在任何时候都可用 free(p)释放。

指针悬空其实是与释放联系在一起的，没有释放就没有悬空。引用计数是一个不错的想法。其方式是，每个内存块上有一个字段叫引用计数，存储当前使用该块内存的使用者数目。当一段代码要使用它时，调用一个函数将引用计数加 1；不使用时，调用另一个函数将引用计数减 1，当该函数判断计数归 0 时，就释放该内存。当然，这个方法并非完善，如果使用者忘记调用函数增加或减少引用计数，照样会出现泄漏或悬空错误，而且存在其他问题。实际上，微软的 COM 机制就采用了该方案，每个 COM 对象都有_AddRef()和_Release()函数增加和减少引用计数。引用计数代码见 DM3-27，工程见“code\第 3 章\3-2\test\refcount”。

无法享受垃圾回收的语言，如C/C++，如何避免内存泄漏？有专门的工具可完成，如boundschecker。其基本原理并不复杂，只要监视每次分配和回收，就可以检测是否有内存泄漏。因此，Visual C++的库就提供了两种版本的malloc()和free()，其中一种就是监视每次调用和分配。当然，如果运行时没有执行到泄漏的代码，自然无法检测到。Visual C++ 2008中可在调试版中定义_CRTDBG_MAP_ALLOC宏，并加入#include <crtdbg.h>，然后在main()函数退出时调用“_CrtDumpMemoryLeaks();”展示泄漏的情况。

DM3-27

(a)

```
1   void addRef(void * p) {
2       int * pCount = (int *)p - 1;
3       *pCount = *pCount + 1;
4       printf("refcount=%d\n", *pCount);
5   }
6   void releaseRef(void * p){
7       int * pCount = (int *)p - 1;
8       printf("refcount=%d\n", *pCount);
9       if (*pCount - 1 <= 0) {
10          printf("free memory\n");
11          free(pCount);
12          return
13      }
14      *pCount = *pCount - 1;
15  }
```

(b)

```
1   void * mallocRef(int size){
2       void * p = malloc(size + sizeof(int));
3       if (p == NULL)
4           return NULL;
5       *(int *)p = 0;
6       return (int *)p + 1;
7   }
```

DM3-27

(c)

```
1   void useGlobalP(void * p) {
2       addRef(p);
3       if (gp != NULL)
4           releaseRef(gp);
5       gp = p;
6   }
7   void freeGlobalP() {
8       if (gp != NULL)
9           releaseRef(gp);
10  }
```

(d)

```
1   void main() {
2       char * p = (char *)mallocRef(10);
3       addRef(p);
4       useGlobalP(p);
5       memset(p, 0, 10);
6       releaseRef(p);
7       freeGlobalP();
8   }
```

DM3-27(a)的第1～5行对于引用计数加1，因为用户传入的地址前4字节才是引用计数的地址，所以第2行对p指针以int *的身份减1，获得引用计数的地址。第9行判断如果引用计数归零，就释放内存。DM3-27(b)分配了内存，将内存前4字节用于记录引用计数（用户看不见），返给用户的地址是第5字节的地址。其原理与图3.6相似，所以DM3-27(b)的第2行所分配大小是用户请求的size加存储引用计数的大小（sizeof(int)）。第5行将引用计数设为0，第6行是将内存的第5字节的地址返回给用户使用。DM3-27(c)展示了对全局变量gp的赋值，useGlobalP()因为在第5行将p指针赋值给了gp，所以前面（第2行）就对该内存块的引用计数加1，freeGlobalP()则对引用计数减1。DM3-27(d)的第3行和第6行、第4行和第7行分别展示了引用内存和释放引用。运行该代码，将看到引用计数的变化，以及最后一次完成了真正的释放内存。

3.2.3 动态数组

学习内存分配后，就可正式做动态数组了。我们当然想做一个通用的，能存储int、float、结构体等类型的数据，再用它去搞定电话簿的全局静态数组。从功能上讲，有以下3点：① 动态分配指定大小的内存；② 读取指定索引的元素；③ 写指定索引的元素。后两个都与元素地址相关，所以转换为一个问题：如何给定索引计算地址。从1.4.1节可知：

$$数组索引为\ i\ 的元素的地址=数组首址\ +i*\text{sizeof(元素类型)}$$

除了 i，还需要知道元素大小和数组首址。我们自然联想到用结构体存储这些信息，其中成员 ptrArray 代表数组首址，elementSize 代表元素大小。另一个直觉的想法是，在该结构体中用成员变量 count 记录元素个数。（**当然，不少时候是因为在编写中需要用到某个成员，就添加上去。即使现在不添加该成员，之后编写设定数组大小的逻辑也要用到，自然会添加。**）例如：

```
struct DynamicArray{
    char * ptrArray;
    int count;
    int elementSize;
};
```

关于动态设定数组大小的逻辑有几点需分析。首先，用给定个数乘以 elementSize 可得新内存大小。但并非仅此而已，我们还需将老数组的内容（如果存在）赋值给新内存，否则设定新的大小值后，原来的值就不见了。即使复制旧数组内容，新的大小值与旧的大小值的关系也影响到复制策略。其次，如何复制旧内容？我们习惯的数组间赋值方式是 a[i] = b[i]，显然不适合，因为通用动态数组无法使用方括号操作符。但从数组本质是一块内存来看，将这一复制动作看成是将一段内存赋值给另一段内存，这样就与数组、元素的类型等完全无关。在 C 语言中，memcpy()可以指定复制字节数，setArrayLength()可以设定动态数组大小的任务，见 DM3-28，工程“code\第 3 章\3-2\test\dynamicArray”。

DM3-28

```
1   void setArrayLength(PDynamicArray ptrArray, int elementCount) {
2       char *p;
3       int moveSize, newSize;
4       if (ptrArray->count == elementCount)
5           return;
6       if (elementCount <= 0) {
7           free(ptrArray->ptrArray);
8           ptrArray->count = 0;
9           return;
10      };
11      newSize = elementCount * ptrArray->elementSize;
12      p = (char *)malloc(newSize);
13      if (ptrArray->count > 0) {
14          if (ptrArray->count > elementCount)          //pack
15              moveSize = newSize;
16          else                                         //extension
17              moveSize = ptrArray->count * ptrArray->elementSize;
18          memcpy(p, ptrArray->ptrArray, moveSize);
19          free(ptrArray->ptrArray);
20      };
21      ptrArray->ptrArray = p;
22      ptrArray->count = elementCount;
23  }
```

DM3-28 中，第 4 行判断，如果新数组的大小与旧的相同，就不做内存分配，直接返回。第 6～10 行判断，如果用户设定的新数组大小小于等于 0，就释放老内存（否则内存泄漏），但不分配新内存，并将元素个数归零。第 11～12 行计算并分配新内存，不要直接赋给结构体（不要用 ptrArray->ptrArray = malloc...直接代替第 12 行），会引起内存泄漏。第 13 行判断，如果 count 大于 0，则代表之前有元素需要复制到新内存中。第 14～17 行通过判断新数组是大于还是小于旧数组来计算复制尺寸。第 18 行将老数组内容复制给新数组。第 19 行释放旧内存（否则引起内存泄漏）。第 21 行再将新内存地址赋值给结构体的相关成员，成为数组的首址。这里面最容易犯的错误是内存泄漏和一些边界条件的判断，初学者要注意。

setArrayItem()将指针 ptrValue 指向内存的值赋给索引为 index 的元素，见 DM3-29。第 2 行判断，如果 index 超出数组边界，就不赋值。（我们的动态数组有了边界判断的功能，C 语言的数组没有该能力。）第 4 行将 ptrValue 指向内存的值复制给索引为 index 的元素。memcpy()的第一个参数是被赋值的地址，即元素的地址为 anArray.ptrArray+(index)*anArray.elementSize，要复制的长度就是元素的大小，即 anArray.elementSize。

DM3-29

```
1   void setArrayItem(int index, DynamicArray anArray, void *ptrValue) {
2       if (index + 1 > anArray.count)
3           return;
4       memcpy(anArray.ptrArray + (index) * anArray.elementSize, ptrValue, anArray.elementSize);
5   }
```

代码 DM3-30 展示了动态数组的使用。第 3 行将元素大小设定给结构体 array，并将其 count 设定为 0。第 5 行将数组大小设为 1，第 6 行将 i 的值赋给第 1 个元素。第 7～9 行将数组大小增大为 2，并将 4 赋值给第 2 个元素。第 10～12 行是循环打印出数组的所有元素值。

DM3-30

```
1   void main() {
2       DynamicArray array;
3       initDynamicArray(&array, sizeof(int));
4       int i = 3;
5       setArrayLength(&array, 1);
6       setArrayItem(0, array, &i);
7       setArrayLength(&array, 2);
8       i = 4;
9       setArrayItem(1, array, &i);
10      for (i = 0; i < array.count; i++) {
11          printf("array[%d]=%d\n", i, *(int *)getArrayItem(i, array));
12      }
13  }
```

这套代码如何复用呢？存储浮点数，那么第 3 行中的 sizeof(int)改为 sizeof(float)；如果存储结构体 Person，就改为 sizeof(Person)。

3.2.4 变化的压力与出路：重构、单元测试和日志

下面开始动手用动态数组修改静态数组版本。不少代码将修改，而程序主体框架又不变，功能也无变化，这一过程称为重构，即在功能不变的情况下改变程序结构。我们必须手动输入各条命令，测试新版本。这个手动输入并判断结果枯燥、易错，而且如果测试的情况比较多，还会忘记一些测试，那么新 bug 就可能诞生了。有什么方法应对？

一种被称为自动单元测试的做法正是针对该问题提出的，而且已经风靡了整个软件开发社区。例如，Java 有其单元测试框架 JUnit，C++有 cppunit，Delphi 有 DUnit，甚至脚本语言都有。软件功能本质上讲就是，当一组输入产生后会产生希望的输出。如果输入可以用代码产生，而输出也可以用代码获取，那么整个测试过程就可自动化。（**注意是过程，不是逻辑，谁也不能自动猜出你的测试逻辑**。）我们只需编写代码产生输入，并用代码获取输出，按照希望和实际结果比对，如果出错就提示，那么，每次测试就直接运行写好的测试代码（测试用例），不需手工输入，用眼、脑判断了。这样测试必然是快、准、无遗漏，因为测试过程自动化了。但是测试代码得自己写。举个简单例子，我们要测试 add()函数，可编写这样一段代码

```
if (add(1,3) != 4)
    printf("add error, 1 + 3 should be 4, but not\n");
```

然后将其放入专门的测试工程的 main()函数中。该程序一执行，就立刻能知道结果。这个例子因为输入数据（1 和 3）、输出数据（返回值）均可代码完成，所以就能用代码测试。

如果与界面相关，就有些无能为力了。比如，有一个窗体，从两个文本框中输入两个加数，其和显示在另一个文本框中。输入是人们用鼠标和键盘完成，代码很难获取，输出直接显示到界面，也难以用代码获取，所以自动单元测试就无能为力了。但反过来想，好的设计就是要将与界面的交互代码做得非常薄，与真正的逻辑代码分离（逻辑代码由很多函数组成，可以用代码施加输入和获取输出，因而能够测试）。那么，做到这一点后，你就能测试大部分的逻辑代码，从而很好地保证工程质量。所以，有另外一个说法“便于测试的代码就是良好设计的代码”。如果发现程序难以用代码来测试，实际已经发出了警告，可能代码设计不好，如可能是界面控制代码和逻辑层代码“绞”在一起了。

下面来审视静态数组版的工程是怎样测试。这里偷个懒，只将因静态数组变动态数组涉及的代码纳入单元测试范畴。我们将测试 queryName、loadPhoneInfo 和 addPhoneInfo。新建解决方案见“code\第 3 章\3-2\PhoneNoteFixedArray_test”，包括之前的工程，并建立 test 工程，然后右键单击 test 工程图标，选择“添加 | 现有项”，从出现的对话框中选择 utils.h、utils.cpp、PhoneInfo.h、PhoneInfo.cpp 和 types.h 加入工程；最后在 test 工程中建立 main()函数，并在其中调用测试函数。

DM3-31

(a)

```
1   void queryName(char * name){
2       ...
3       for (i = 0; i < gCount; i++) {
4           if (strcmp(gPhoneInfoList[i].name, name) == 0) {
5               printf("%s   %s\n", gPhoneInfoList[i].name, gPhoneInfoList[i].phone);
6               return;
7           }
8       }
9       printf("cannot find the name: %s\n", name);
10  }
```

(b)

```
1   BOOLEAN queryName(char * name) {
2       int i;
3       for (i = 0; i < gCount; i++) {
4           if (strcmp(gPhoneInfoList[i].name, name) == 0) {
5               pShowPhoneInfo(&gPhoneInfoList[i]);
6               return TRUE;
7           }
8       }
9       printf("cannot find the name: %s\n", name);
10      return FALSE;
11  }
```

```
void main() {
    printf("begin test\n");
    //测试查询到某个姓名
    testQueryName_CanFind();
    //测试查询不到某个名字
    testQueryName_CanNotFind();
    //测试添加记录和加载整个文件
    testloadPhoneInfo_addPhoneInfo();
    printf("end test\n");
}
```

先测试容易的，如 queryName 代码，见 DM3-31(a)。第 5 行在查询到结果后，将数据打印。我们无法测试是否屏幕上输出了需要的信息，只有用代码获得打印信息才能确定 queryName()是否执行了显示，故定义函数指针变量 ShowPhoneInfoPtr pShowPhoneInfo。ShowPhoneInfoPtr 定义为 typedef void (*ShowPhoneInfoPtr)(PhoneInfo * info)。然后定义函数 showPhoneInfo：

```
void showPhoneInfo(PhoneInfo * info) {
    printf("%s    %s\n", info->name, info->phone);
}
```

真正的电话簿程序运行时，将 showPhoneInfo 设定给 pShowPhoneInfo：

```
ShowPhoneInfoPtr pShowPhoneInfo = showPhoneInfo;
```

修改 DM3-31(a)为 DM3-31(b)。将 DM3-31(a)的第 5 行替换为 DM3-31(b)的第 5 行，这样测试时就可用 setShowPhoneInfo(mockShowInfo)函数将测试用的假显示函数 mockShowInfo()赋值给 pShowPhoneInfo。那么，测试时就调用到 mockShowInfo()函数，而不是真正的打印函数。

```
void mockShowInfo(PhoneInfo * info) {
```

```
    isCallShowInfo = TRUE;
    gInfo = *info;
};
```

mockShowInfo()函数将要打印信息 info 赋值给全局变量 gInfo，并将 bool 类型的 isCallShowInfo 设定为真。那么，测试程序可先将 isCallShowInfo 赋值为假，再调用 queryName。如果查询到了信息，那么 mockShowInfo 就会被调用。通过检查 isCallShowInfo 是否为真，以及 gInfo 的成员变量 name 和 phone 是否正确，就能判定 queryName 是否调用了显示函数。（我们居然可以通过函数指针的间接性，测试了似乎无法测试的打印逻辑！）如果要测试查询到与否，因为 DM3-31(a)中的函数 queryName()没有返回值，是无法测试的，因此 DM3-31(b)中的 queryName()的返回值为 bool 类型。这样就可根据返回结果检测是否查询到了。（因为编写测试，我们改进了生产代码。）我们写测试代码和因此改进生产代码的目的是为了将来重构系统的时候，测试代码和生产代码都尽量少改动，甚至不变。下面以此原则来分析测试代码的好坏。

少改甚至不变

为了模拟全局数组 gPhoneInfoList 中有两条 PhoneInfo 信息，进行初始化，见 DM2-32(a)。很明显，动态数组肯定不支持 gPhoneInfoList[0]这样的代码，而其最后一行将 gCount 赋值为 2，也肯定与动态数组无关。这就意味着对动态数组，DM2-32(a)中的相关测试代码要修改。这违反了“少改甚至不变”的原则。究其原因是我们将实现细节——全局静态数组——暴露给测试代码了。测试代码因为利用了这个细节，才有“gPhoneInfoList[0]”和“gCount= 2;”。而一旦转换为动态数组，实现细节的改变必然导致利用该细节的测试代码修改。（怎么办？隐藏细节！）通过函数调用，将实现细节隐藏到函数中。这样将来变化时，调用者不变，变的是被调用者。编写 addInfo()函数来向全局数组添加电话信息，见 DM3-32(b)，将 info 的值赋给指针 pCurrentInfo 指向的元素，然后该指针向后移动一个元素，并将总个数加 1。

DM3-32

(a)

```
strcpy(gPhoneInfoList[0].name, "hh");
strcpy(gPhoneInfoList[0].phone, "123");
strcpy(gPhoneInfoList[1].name, "ww");
strcpy(gPhoneInfoList[1].phone, "456");
gCount = 2;
…
```

(b)

```
void addInfo(PhoneInfo * info) {
    * pCurrentInfo = *info;
    pCurrentInfo++;
    gCount++;
}
```

(c)

```
void testQueryName_CanFind() {
1    initTestEnv();
2    PhoneInfo info;
3    strcpy(info.name, "hh");
4    strcpy(info.phone, "123");
5    addInfo(&info);
6    strcpy(info.name, "ww");
7    strcpy(info.phone, "456");
8    addInfo(&info);
9    if (queryName("hh") == FALSE)
10       showError("…");
11   if (isCallShowInfo == FALSE)
12       showError("…");
13   if ((strcmp(gInfo.name, "hh") != 0) || (strcmp(gInfo.phone, "123") != 0))
14       showError("…");
}
```

测试代码见 DM3-32(c)，用 addInfo()来修改 DM2-32(a)中的代码，其等价代码是第 2～8 行。即使将来变成动态数组版，DM3-32(c)的代码也不会变，到时只需修改 addInfo()和 queryName()即可。第 9～14 行是检测代码，其中第 11～14 行检测是否 queryName()在查询到“hh”的情况下调用了显示函数（即 DM3-31(b)的第 5 行）。第 1 行，函数 initTestEnv()将 mockShowInfo 设定给 pShowPhoneInfo，这样才会有第 11～14 行的测试，因为 mockShowInfo 会将显示信息赋给 gInfo，并设定 isCallShowInfo。另外，showError()会调用 exit 指令直接退出测试。

继续测试，加载和添加信息。在 DM3-33(a)中，第 2～5 行准备测试环境。第 6～11 行向文件中添加了两条记录。第 12 行加载该文件，应该在全局数组中找到加载的 2 条记录。为了在第 13 行测

试记录是否为 2 条，我们添加了 getInfoCount()函数。没有直接用 gCount()而是用 getInfoCount()返回 gCount。这也是为了隐藏细节，将来变化为动态数组时，第 13 行的测试代码不用修改（只改 getInfoCount()的实现）。同理，第 16 行和第 19 行获取数组的第 1 个和第 2 个元素，我们也不是直接访问全局变量 gPhoneInfoList[0]、gPhoneInfoList[1]，而是用 getInfo(0)和 getInfo(1)。同样，因为隐藏了细节，之后动态数组版测试代码不用修改。

下面来看 loadPhoneInfo 因测试发生了什么改变。DM3-25(a)的第 10 行、第 14～15 行完成的任务其实就是上一个测试时加入的函数 addInfo()完成的任务，所以直接替换为 addInfo()。加载时显示信息的逻辑也由 printf()改变为通过函数指针 pShowPhoneInfo 显示（与 queryName()相似）。整个 loadPhoneInfo()不会因为升级成动态数组而修改代码，见 DM3-33(b)。原来的版本中，addPhoneInfo()代码见 DM3-25(d)，第 10～12 行完成的也是 addInfo()的功能，因此替换，见 DM3-33(c)。此时，addPhoneInfo()的代码也不会随动态数组版的变化而修改。

DM3-33

(a)

```
void testloadPhoneInfo_addPhoneInfo() {
1    initTestEnv();
2    setShowPhoneInfo(mockShowInfo2);
3    ...
4    getDefaultFilePath("test.dat");
5    clearFile("test.dat");
6    PhoneInfo info;
7    strcpy(info.name, "hh");
8    strcpy(info.phone, "123");
9    addPhoneInfo(info);
10   ...
11   addPhoneInfo(info);
12   loadPhoneInfo(gPhoneInfoPath);
13   int count = getInfoCount();
14   if (count != 2)
15       showError("...");
16   info = getInfo(0);
17   if ((strcmp(info.name, "hh") != 0) || (strcmp(info.phone, "123") != 0))
18       showError("...");
19   info = getInfo(1);
20   if ((strcmp(info.name, "ww") != 0) || (strcmp(info.phone, "456") != 0))
21       showError("...");
22   ...
}
```

DM3-33

(b)

```
void loadPhoneInfo(char * file) {
  ...
  while (feof(fp) == 0) {
    ...
    PhoneInfo info;
    if (fread(&info, sizeof(PhoneInfo), 1, fp) == 0)
      break;
    pShowPhoneInfo(&info);
    addInfo(&info);
  }
  ...
```

(c)

```
void addPhoneInfo(PhoneInfo info) {
  ...
  fwrite(&info, sizeof(PhoneInfo), 1, pf);
  addInfo(&info);
  fclose(pf);
}
```

现在复查 queryName()的代码是否会因动态数组而修改。在 DM3-31(b)中，第 3 行访问了属于静态数组的 gCount，第 4 行访问了静态数组 gPhoneInfoList[i]，所以还有问题。其实只需用测试时新增函数 getInfoCount()替换第 3 行，用 getInfo(i)替换 gPhoneInfoList[i]，即可隐藏细节，使得 queryName()也不需随版本的变化而修改。（测试改善生产代码！）

至此，通过满足测试需求修改生产代码，最终让大部分生产代码和全部测试不会因将来的动态数组升级而修改。这就是所谓的“测试驱动的开发”，测试导致设计的完善。在生成代码最终调试前，我们就小块小块地测试了所有代码，既减轻了调试压力，又为未来维护修改提供了坚实基础。而 JUnit 之类的单元测试框架，为测试提供了更多方便。

有了单元测试，我们已经向有序的世界迈进了一大步。有了单元测试，你可以轻松地将一个庞然大物分割得很小来开发，而且能开发一点测试一点。最后是单元测试的一点小经验：

（1）尽量测试边界条件，如删除最后一条记录。

（2）不测试被测者内部数据结构，只测试对外输出是否正确，这样内部实现变化后，测试代码可重用而不需修改。

（3）开发中出现了单元测试没测到的错误，如下处理：① 找出错误；② 增加单元测试，让该测试出错；③ 修改生产代码，让新增测试通过。不要先修改再测试。

如何在整个程序部署运行后（无法在集成开发环境下调试）依然有很好的调试性？我们要借助“日志”技术。最简单的就是用 printf()打印信息。但如果在发布的程序中嵌入太多打印语句，比较影响性能。聪明的程序员就用“宏+条件编译”来处理。与直接函数调用相比，宏可少一次函数调用。条件编译可在不声明调试宏_DEBUG 时没有 printf()输出，从而进一步提高性能。

日 志

```
#ifdef _DEBUG
#define DEBUG_MSG(msg) printf("%s", msg)
#else
#define DEBUG_MSG(msg) {}
#endif
```

当然，更复杂的日志技术可以通过配置文件决定输出的信息是写到屏幕还是文件，是 E-mail 还是数据库。log4j 和 log4net 是 Java 和 .NET 平台上两个常用的日志框架。

3.2.5 电话簿扩展（2）：可测试的恩赐

为了将电话薄升级为动态数组版本，我们将 3.2.4 节的工程复制过来，做以下修改：① 将动态数组相关的操作放入 utils.h 和 utils.cpp；② 修改与数组实现相关的函数，见 DM3-34。单元测试不用修改。

DM3-34

```
int getInfoCount() {
  return gPhoneInfoList.count;
}
PhoneInfo getInfo(int index) {
  return *(PhoneInfo *)getArrayItem(index, gPhoneInfoList);
}
void showPhoneInfo(PhoneInfo * info) {
  printf("%s    %s\n", info->name, info->phone);
}
void setShowPhoneInfo(ShowPhoneInfoPtr pfunc) {
  pShowPhoneInfo = pfunc;
}
void addInfo(PhoneInfo * info) {
  setArrayLength(&gPhoneInfoList, gPhoneInfoList.count+1);
  setArrayItem(gPhoneInfoList.count-1, gPhoneInfoList, info);
}
void initInfo() {
  initDynamicArray(&gPhoneInfoList, sizeof(PhoneInfo));
}
```

运行测试程序，通过。再运行生产代码，通过。我们没有花任何精力，轻松迁移成功。这得益于隐藏细节，也得益于单元测试。工程见“code\第 3 章\3-2\PhoneNoteDynamicArray”。

3.3 优雅的积木

在前面编程历练中，慢慢接触到软件开发中一些核心的东西。从最初提炼一些函数到动态数组，已经触摸到软件复用的边缘。如何才能深入这个深刻的主题？除了学习各种理论外，其实最简单、最直接的方式就是分析手边的程序，抽象并剥离出可复用部分，不断改造现有程序。有痛苦，才有进步。下面开始我们的积木游戏。

3.3.1 可复用硬盘数组

电话薄程序的哪个部分还可以被其他程序重用？仔细分析，可锁定为结构体从硬盘读写的过程，用索引访问硬盘类似于用数组访问数据：① 给定索引，从硬盘读出结构体；② 给定索引，往

硬盘相关位置写入结构体；③ 删除给定索引的结构体；④ 将新的结构体添加到文件尾部。这些逻辑能够复用吗？是否觉得有点抽象？查看写过的代码，然后分析是否有共性。DM3-25(a)的第 10 行“if (fread(pCurrentInfo, sizeof(PhoneInfo), 1, fp) == 0)”展示了从文件游标指向的当前位置读入结构体 PhoneInfo 的逻辑。如果能将数组索引换算成在文件中的偏移量，并移动文件游标到该偏移位置，那么用 fread()就可完成任务。偏移量如何计算？index * sizeof(PhoneInfo)。我们可以编写伪代码，见 DM3-35(a)。这段代码可重用吗？不妨再假想其他应用场景，要读写一个类型为 Person 的结构体，见 DM3-35(b)。

DM3-35

(a)

```
//移动文件游标
moveFileCurrentPosition(index * sizeof(PhoneInfo));
fread(pPhoneInfo, sizeof(PhoneInfo), 1, fp)
```

(b)

```
//移动文件游标
moveFileCurrentPosition(index * sizeof(Person));
fread(pPerson, sizeof(Person), 1, fp)
```

两段代码的唯一差别就是粗体所示部分，即代表结构体大小的部分。在代码的相同位置，只是值的差异，这就是可参数化的表现。只需把 sizeof(PhoneInfo)替换成参数 elementSize 即可。而 fread()的第一个参数不管是 pPhoneInfo 还是 pPerson，本质上就是存储读入信息的地址，所以通过另外一个参数 pValue 传入即可，见 DM3-36(a)。对于数组的操作，每次都传入相同的 elementSize 和 fp，烦琐、易错，参考动态数组的思路，将 elementSize、fp 等相关变量集中为结构体 DiskArray，将结构体指针传入，从成员变量获取 elementSize 和 fp，就解决了以单独参数传入的麻烦，见 DM3-36(b)。

DM3-36

(a)

```
void getValue(FILE * fp, void * pValue, int index, int elementSize) {
    moveFileCurrentPosition(index * elementSize);
    fread(pValue, elementSize, 1, fp)
}
```

(b)

```
void getValue(DiskArray * array, void * pValue, int index) {
    moveFileCurrentPosition(index * array->elementSize);
    fread(pValue, array->elementSize, 1, array->fp)
}
```

再分析往指定 index 写入数据这个过程，向文件尾部添加数据与上面读数据一样，通过移动文件游标，向相关位置写入固定长度的数据即可，也能参数化。删除呢？这个要麻烦些，因为中间元素删除后所有元素要前移，并修改文件大小，从文件尾部截断 elementSize 字节。

现在与功能有关的未知函数有两个：如何定位文件游标，如何截断文件。对第一个问题，在百度中输入关键字组合“C 语言 文件 读写 任意位置 函数”，就能查到 fseek()函数，其原型如下：

```
int fseek( FILE *fp, long offset, int origin )
```

offset 为偏移量，可正负。Origin 指示从哪里开始计算偏移量，有三个宏可选：SEEK_SET，从文件开头开始计算；SEEK_CUR，从当前位置；SEEK_END，从文件结尾。

对第二个问题，在百度中输入“C 语言 截断 文件”，可以搜索到 ftruncate()和 chsize()。实验发现，前者在 Visual C++中似乎根本没有相关头文件，应该是 Linux 平台的。chsize()呢？有相关头文件吗？如果有，是什么？这个问题经常让初学者不知所措。用下面的小方法可解决，该头文件是 io.h。

> 查找定义一个函数的头文件：安装 Visual Assist，随便在 Visual C++中建立一个工程，然后在源文件中输入要查找的函数，按组合键 Alt+G，如存在该函数，即会打开对应头文件。

函数 int chsize(int handle, long size)是将文件大小设定为 size，如果成功，则返回 0。handle 是文件句柄。例子程序都是用 open()函数返回的句柄。但我们都是用 C 语言的文件操作函数，它们用 FILE 类型代表文件指针。如果没有好办法，要做删除元素操作，只能调用 fclose()关闭正在操控的

文件，用 open()再次打开，然后调用 chsize()。能否在借助 fopen()打开文件后调用 chsize()？（**猜测如下。**）fopen()本质上要调用操作系统的 API 完成文件打开，那么文件句柄肯定是放入 FILE 结构体中。我们用上面寻找头文件的小技巧，可看到其定义，见 DM3-37。从名字看，_file 有点像，而且也是 int 类型。那么做个实验，见 DM3-38。

实验显示，chsize()成功修改了大小（因为第 4 行打印了“ok”），但打开文件内容发现依然是“123”，而非想象中的“1”（因为先写入“123”，然后将大小设定为 1 字节，自然最后应该只有字符'1'）。为了尽量与资料上的例子程序相似。在 DM3-38 第 5 行后加入 DM3-39 的代码。可是这次连“ok”都不打印了。为什么？这里的区别仅仅是 fopen()的打开模式，成功用的是“wb+”，失败用的“rb”，即后者只有读的能力，前者有读和写的能力。

DM3-37

```
struct _iobuf {
   char *_ptr;
   int    _cnt;
   char *_base;
   int    _flag;
   int    _file;
   int    _charbuf;
   int    _bufsiz;
   char *_tmpfname;
};
typedef struct _iobuf FILE;
```

DM3-38

```
1  FILE * fp = fopen("C:\\temp.txt", "wb+");
2  fwrite("123", 3, 1, fp);
3  if (chsize(fp->_file, 1)== 0)
4     printf("ok\n");
5  fclose(fp);
```

DM3-39

```
fp = fopen("C:\\temp.txt", "rb");
if (chsize(fp->_file, 1)== 0)
   printf("ok\n");
fclose(fp);
```

简单推理：修改文件大小至少也是“写”的行为，在“只读”模式下自然要失败。将“rb”改为“rb+”，使其有写的能力就可以了。这次实验不仅成功修改，而且打开文件确实也看到了效果。问题出在哪里？（**分析，冷静分析。**）DM3-38 的第 3 行返回确实是 0，说明修改大小应该成功。而实际文件内容又确实是“123”，有一种可能，chsize()执行后才发生了“123”的写入，但明明 fwrite()在 chsize()前面。为了验证“123”的写入时机，我们设计了一个实验：在 DM3-38 的第 2 行设置断点，执行后，看文件内容是否变化。如果没有变化，就单步执行，看到底哪条语句执行时文件内容变化了。为此，我们先将 C:\temp.txt 删除。在执行 fopen()后，该文件再次被生成，但内容是空的。fwrite()执行后果然文件还是空的，写动作没有发生。单步执行 fclose()后，文件内容变为“123”。果然是先发后至。为什么并非在 fwrite()时就真正写硬盘呢？想想，如果用户调用 fwrite()每次写 1 字节，共写 100 次，与调用一次 fwrite()写入 100 字节，哪个效率高？自然是后者。怎样让前者的效率提高？自然是增加一个内存缓存，fwrite()写入时写到缓存，速度就比写硬盘快。当缓存满了就写入到硬盘。在 DM3-37 中，FILE 定义中的_bufsiz 就有点像缓存大小的意思。看来 C 语言的文件操控真的有缓冲。通过单步调试，我们甚至能发现 FILE 的成员_base 指向的就是缓存的起始，在内存窗体中甚至可查看到执行 fwrite()后，_base 指向的内存被写入了“123”。如果有函数能强迫将缓存中的内容写入硬盘，就不会出现“先发后至”了。从“合理”这一角度思考，系统应该给用户控制刷新缓存的能力。在百度中输入“C 语言 文件 刷新 缓存 函数”，就可查到 fflush()。该函数只有一个参数 FILE *。在 DM3-38 的第 2 行与第 3 行间增加“fflush(fp);”语句，我们终于看到希望的效果，最终文件内容变成截断后的“1”，而非先前的“123”。并且也实验出，fopen()的打开模式要有写能力，chsize()才能成功。

现在我们需要考虑在对硬盘数组进行增、删、改、读操作时，文件打开操作应如何配合的问题。直观地考虑，这 4 个操作对应 4 个函数，在这些函数一开始处打开文件，在函数结束时关闭。似乎没有问题。对照电话簿的使用场景分析。一个场景是打开文件，遍历读出所有记录，并显示在屏幕

上。之前的做法是打开一次文件，不断读取，现在就变成了读取每条记录都要开关文件一次，效率明显不如前者。所以一个合理的设计是：提供 open()和 close()函数，对应打开和关闭文件，在打开状态下，用户能自由对硬盘数组实施操作，即被打开的文件需要同时具有读、写的权限。是否“wb+”、“rb+”和“ab+”三种具有读、写权限的模式任选一种？注意，“wb+”是要删除已存在的文件；“ab+”虽然不删除已存在文件，但 fwrite()只能向尾部写入（fseek()不起效），所以被否决（我们用索引定位必须依赖 fseek 任意定位）；“rb+”要求文件必须存在，不能像“wb+”一样创建新文件。所以，我们只能将“wb+”和“rb+”结合。当文件不存在时，用“wb+”，否则用“rb+”，代码如下：

```
array->fp = fopen(array->fileName, "rb+");
if (array->fp == NULL) {                    //如果文件不存在，就用写方式打开，创建新文件
    array->fp = fopen(array->fileName, "wb+");
```

下面来设计文件格式。由于模仿数组，我们希望在用户打开文件后能快速获取到元素的个数信息（因为常见如下逻辑：用元素个数作为数组遍历的循环上限）。因此，我们在文件的前 4 字节存储元素个数，其后才是每个被存储的元素，见图 3.9。

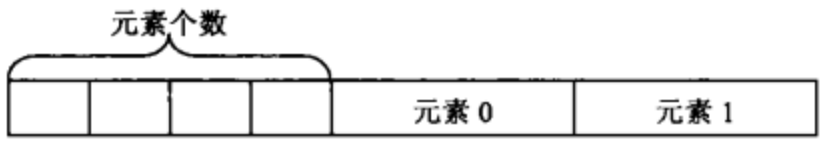


图 3.9　硬盘数组文件格式

最终我们设计出代表硬盘数组的结构体和操作函数：

```
struct SimpleDiskArray{
    FILE * fp;                  //文件指针
    int elementSize;            //元素大小，如存储 int 元素，该字段就是 sizeof(int)
    int count;                  //元素个数
    char * fileName;            //文件的名字
    void * tmpBlock;            //临时存放读入数据的缓存
};
//打开文件，读入元素个数，设定给 array->count
BOOLEAN openSimpleDiskArray(SimpleDiskArray * array);
//关闭文件
void closeSimpleDiskArray(SimpleDiskArray * array);
//设定相关参数
void initSimpleDiskArray(SimpleDiskArray * array, int elementSize, char * fileName);
//清除相关参数，如释放内存
void finalizeSimpleDiskArray(SimpleDiskArray * array);
//添加记录
BOOLEAN insertSimpleDiskArray(SimpleDiskArray * array, void * pValue);
//删除记录
BOOLEAN deleteSimpleDiskArray(SimpleDiskArray * array, int index);
//更新指定索引记录
BOOLEAN updateSimpleDiskArray(SimpleDiskArray * array, int index, void * pValue);
//读取指定索引记录
BOOLEAN getSimpleDiskArray(SimpleDiskArray * array, int index, void * pValue);
//获取数组长度
int getSimpleDiskArrayCount(SimpleDiskArray * array);
```

在阅读代码时，注意提炼可重用函数。如 seekSimpleDiskArray()定位索引为 index 的元素对应的文件游标，在 3 个有 index 作为参数的函数中均被用到。如果将来文件头格式变化，依然只需修改这一个函数即可（见习题 4）。

```
void seekSimpleDiskArray(SimpleDiskArray * array, int index) {
// index * array->elementSize 是元素部分的偏移量，sizeof(array->count)是元素数目字段的大小
    fseek(array->fp, index * array->elementSize + sizeof(array->count), SEEK_SET);
}
```

下面主要给出删除、更新和读取涉及的函数，工程见“code\第 3 章\3-3\simplediskarray\simplediskarray”。

getSimpleDiskArray 指定元素的 index，从文件中读出数据，并写入指针 pValue 指向的内存，见 DM3-40。第 1 行的 CHECK_BOUND 为宏函数，如果 index 超过数组边界，就返回 FALSE。

```
#define CHECK_BOUND(index, array) { if (index + 1 > array->count)  return FALSE; }
```

什么是宏函数？它有什么作用？假设有一段代码要反复重用，又不想作为函数重用（为了节省函数压栈传参、call 和返回的开销），就用宏函数。其本质依然是复制替换。比如，源代码中出现 CHECK_BOUND(i, myarray)，那么编译器会替换宏函数右边定义的表达式“{if (index+1 > array->count) return FALSE;}”，用 i 替换 index，myarray 替换 array，替换结果为“{if (i+1 > myarray->count) return FALSE;}”。定义宏函数时注意两点：

① 定义宏函数时用花括号将整个代码括起来是一个好习惯。如果没有，则 CHECK_BOUND 定义如下：

```
#define CHECK_BOUND(index, array)  if (index + 1 > array->count)  return FALSE;
```

在源代码中出现“CHECK_BOUND(i, myarray);”时，替换结果就会包含结尾的分号。结果为：

```
if (i + 1 > myarray->count)  return FALSE;;              //最后有两个分号
```

末尾的两个分号，一个是宏函数中 FALSE 后的，一个是调用宏函数时“...,myarray);”最后的分号。这样就有问题。但如果整个宏函数用大括号括起来，替换结果如下，就没问题：

```
{if (i + 1 > myarray->count)  return FALSE;};
```

② 形式上有多行代码的宏函数，最后一行之前的每行结尾要用'\'结束，因为宏本身必须在一行结束，否则编译器无法识别。如果逻辑上有多行，写成一行不容易理解。因此，在分解成多行后，每行末尾加上'\'，告诉编译器下面的行和本行是一行。比如，CHECK_BOUND 可如下定义(注意第一行末尾的'\'):

```
#define CHECK_BOUND(index, array) {if (index + 1 > array->count) \
 return FALSE;}
```

在 DM3-40 中，第 2 行定位文件游标，第 3 行从 array->fp 中读取一块大小为 array->elementSize 字节的数据，读到 pValue 指向的内存。

DM3-40

```
BOOLEAN getSimpleDiskArray(SimpleDiskArray * array, int index, void * pValue) {
1   CHECK_BOUND(index, array);
2   seekSimpleDiskArray(array, index);
3   if (fread(pValue, array->elementSize, 1, array->fp) != 1)
4       return FALSE;
5   return TRUE;
}
```

在 DM3-41 中，第 1～2 行逻辑同 DM3-40，第 3 行将 pValue 指定地址开始一块 array->elementSize 字节的数据，写入 array->fp 所指文件的当前位置。

DM3-42 展示了删除元素的逻辑。第 5～10 行将被删除元素后的元素依次往前移动。因为删除了索引为 index 的元素，所以被前移的元素索引从 index+1 开始，因此第 5 行循环初始值 i=index+1。

DM3-41

```
BOOLEAN updateSimpleDiskArray(SimpleDiskArray * array, int index, void * pValue) {
1   CHECK_BOUND(index, array);
2   seekSimpleDiskArray(array, index);
3   if (fwrite(pValue, array->elementSize, 1, array->fp) != 1)
4       return FALSE;
5   return TRUE;
}
```

第 6 行将索引为 i 的元素读出到临时内存 pTmp 中。第 8 行将其复制到前一个元素，索引为 i-1。第 11 行将现有数组个数减 1，然后第 12 行调用函数 changeSimpleDiskArrayCount()，不仅将该数目写入硬盘头 4 字节，且将重新设定文件大小(因为删除了一个元素)。注意，第 2～3 行对临时缓存 pTmp 采取了一种叫懒加载的策略（lazy load）。因为只有删除逻辑才会用到它，所以没有在 openSimpleDiskArray 中直接分配，而是当调用了删除函数时，才分配。这样如果没有做删除操作，就可少分配一块内存。第 2 行判断是否为 NULL 就是实现“如果没有分配则分配的逻辑”。另外，将 pTmp 保存到结构体中，可避免每次删除函数调用时，均需分配和释放临时内存 pTmp 的低效。

DM3-42

```
BOOLEAN deleteSimpleDiskArray(SimpleDiskArray * array, int index){
1 CHECK_BOUND(index, array);
2 if (array->tmpBlock == NULL)
3       array->tmpBlock = malloc(array->elementSize);
4 void * pTmp = array->tmpBlock;
5 for (int i = index + 1; i < array->count; i++){
6   if (getSimpleDiskArray(array, i, pTmp) == FALSE)
7       return FALSE;
8   if (updateSimpleDiskArray(array, i - 1, pTmp) == FALSE)
9       return FALSE;
10}
11 array->count--;
12 changeSimpleDiskArrayCount(array, array->count);
13 return TRUE;
}
```

DM3-43 实现了改变文件大小的逻辑。第 5 行在调用 chsize()之前调用了 fflush()，就是处理前面实验遇到的缓存问题，保证之前可能的修改操作在硬盘中生效。计算文件当前大小=结构体 count 成员的大小 + 当前元素个数×元素大小。而第 1～2 行将指定的 count 值写入到文件开始处。该工程的实现依然用单元测试保证，见“code\第 3 章\3-3\simplediskarray\testsimplediskarray”。

DM3-43

```
void changeSimpleDiskArrayCount(SimpleDiskArray * array, int count){
1   fseek(array->fp, 0, SEEK_SET);
2   fwrite(&count, sizeof(array->count), 1, array->fp);
3   array->count = count;
4   fflush(array->fp);
5   chsize(array->fp->_file, array->count *   array->elementSize + sizeof(array->count));
}
```

3.3.2 分享它（1）：理解编译链接过程

如何让其他程序使用我们的硬盘数组呢？当然，像之前那样直接将源代码引用是一个办法，但存在一些问题：① 你可能不想别人看到代码（特别商业程序）；② 如果修改了源代码，你得将被修改的代码找出来，让别人更新，或者全部发布出去更新。这个过程也会遇到麻烦，假定本代码又依赖了其他代码，过程会很麻烦。

有没有办法直接将编译好的模块让别人使用呢？有，一种是静态链库方式，一种是动态链接库方式。为了将来遇到编译链接问题时（实际上，C/C++编程经常遇到诡异的编译链接问题），有可用于分析的基础知识，我们先简单理解编译链接过程，再对两种链接方式给出相应的说明。

程序从源代码变成最后可运行的程序由两个步骤完成：① 编译（compile），每个 xxx.c/cpp 文件被编译成目标文件 xxx.obj，*.obj 中存储有源代码编译成的机器码；② 链接（link），将相关*.obj 文件组合成一个可执行文件（当然也可能是动态链接库）。

以工程“code\第 3 章\3-2\PhoneNoteFixedArray”为例，用命令行 cl.exe 来编译。先建立一个目录存储编译链接出的文件，如 C:\mytest。然后在 CMD 中用 cd 命令将当前目录转到“C:\mytest”（cl.exe 生成的文件会放在当前目录下），然后输入命令：

```
cl /c 全路径名\PhoneInfo.cpp 全路径名\PhoneNote1.cpp 全路径名\utils.cpp
```

其中 3 个文件是包含全路径的（否则 cl 无法知道它们在哪里），用空格分开，包含了工程的所有文件。“/c”选项代表只生成 OBJ 文件。在笔者的计算机上就是：

```
cl /c E:\mydocs\写书\猜测、实证和构建\code\第 3 章\3-2\ PhoneNoteFixedArray\
PhoneNote1\PhoneInfo.cpp E:\mydocs\写书\猜测、实证和构建\code\第 3 章\3-2\
PhoneNoteFixedArray\PhoneNote1\PhoneNote1.cpp E:\mydocs\写书\猜测、实证和构建\code\
第 3 章\3-2\PhoneNoteFixedArray\PhoneNote1\utils.cpp
```

于是 C:\mytest 目录中出现了 PhoneInfo.obj、PhoneNote1.obj 和 utils.obj。这就是编译产生的目标文件。因为此时当前目录就在 obj 文件所在目录，所以再输入命令并执行：

```
link PhoneInfo.obj PhoneNote1.obj utils.obj
```

其含义是将这 3 个 OBJ 文件链接成可执行文件。运行后，C:\mytest 目录下多了 PhoneInfo.exe 文件。当然，如果 cl 命令不加“/c”选项，编译和链接将一起完成。在这里，link.exe 就是链接器。基于对该过程的理解，有些错误我们就能知道究竟了。比如，下面一段代码：

```
void mytest();
void main(){
    mytest();
}
```

在没有实现 mytest()函数的情况下，如果将第 1 行注释掉，编译器就会报错“error C3861: "mytest"：找不到标识符”。这是编译错误，很好理解，因为确实没有声明 mytest。如果用“cl /c”编译本代码所在的文件，就发现没有生成 OBJ 文件，即编译失败。如果上述代码不修改，就会链接错误“error LNK2001: 无法解析的外部符号 "void __cdecl mytest(void)" (?mytest@@YAXXZ) test2.obj”。仔细分析会发现，这个错误明确说是 LNK（链接）且找不到外部符号 mytest。如果用“cl /c”命令编译，会发现编译通过，且生成了对应的 OBJ 文件。可见，错误发生在链接时找不到符号 mytest。为什么要找 mytest？合理的推测是编译时只是在 OBJ 文件中说明要调用 mytest()函数，因为它可能在另外一个 .c/.cpp 文件对应的 OBJ 文件中，所以编译期不处理，仅仅留下记号。（**这个推测我们需要证明。**）如果下次遇到同样找不到符号的链接错误（特别是网上下载的工程），我们就应

该猜想到可能没有包含该函数实现的目标文件（.obj 或静态库.lib）供链接器链接。有了这个提示，自然就可在项目中寻找这个“失落”的目标文件，或在网上查找。

对于下面的代码，为什么会找不到符号“mytest”？

```
void main(){
    mytest();
}
void mytest() { }
```

也许 C++编译器（C 语言狡猾些，不报错。Visual C++环境亦可按 C 语言方式编译）比较老实，遇到一个符号时会查看文件中该符号前的部分是否声明该符号，没有就要报错。因此才有头文件放在 .c/.cpp 的头部，事先声明函数和变量（不需实现）。为了验证这一猜想，在 main()之前加上“void mytest();”语句。结果正确。

我们想进一步理解编译和链接的过程，能否借助基本知识、猜测、实验搞定呢？首先猜想：

① 源代码编译成的机器码被放入到 OBJ 文件中，链接时，链接器再将它们拼接成执行程序。

② 前面已猜测，在目标文件中对一个函数的调用被翻译成对一个符号的引用，所以如果找不到实现该符号的 OBJ 文件，链接期就会出错。

先证明第一个猜测。查看“code\第 3 章\3-3\simplediskarray\simplediskarray\simplediskarray.cpp”生成的头文件 simplediskarray.obj（将其复制到...3-3\coff_obj 下），查看 DM3-41 对应的函数 updateSimpleDiskArray 的机器码是否在 simplediskarray.obj 中。

思路：在调试方式下，在 updateSimpleDiskArray 处设置断点，查看反汇编，展开其机器码；然后用 UltraEdit 打开 simplediskarray.obj，查找这段机器码。反汇编见 DM3-44。将每行指令地址右边的字节（就是粗体部分）复制出来连在一起，如下：

```
8b 45 0c 83 c0 01 8b 4d 08 3B 41 08 7E 04 33 C0 EB 43 8B 45 0C 50 8B 4D 08 51
```

该字符串被发现有 2 处。（**肯定有相同的代码出现在了同一源文件的其他地方。**）在 DM3-40 中，第 1～2 行与 DM3-41 中的相同。整个代码中也只有这两个函数在第 1～2 行是相同的，对应 OBJ 文件中两处相同的机器码。这基本已证明了第一个猜想，OBJ 文件中有函数机器码。

DM3-44

```
  CHECK_BOUND(index, array);
0041285e 8B 45 0C          mov  eax, dword ptr [ebp+0ch]
00412861 83 C0 01          add  eax, 1
00412864 8B 4D 08          mov  ecx, dword ptr [ebp+8]
00412867 3B 41 08          cmp  eax, dword ptr [ecx+8]
0041286a 7E 04             jle  00412870
0041286c 33 C0             xor  eax,eax
0041286e EB 43             jmp  004128b3
  seekSimpleDiskArray(array, index);
00412870 8B 45 0C          mov  eax, dword ptr [ebp+0ch]
00412873 50                push eax
00412874 8B 4D 08          mov  ecx, dword ptr [ebp+8]
00412877 51                push ecx
00412878 E8 8D E7 FF FF    call 0041100a
0041287d 83 C4 08          add  esp, 8
  if (fwrite(pvalue, array->elementsize, 1, array->fp) != 1)
```

继续往下看，就能发现与第二个猜想有关的信息。DM3-44 倒数第 3 行地址为 00412878 对应的指令是一条用相对偏移跳转的 call 指令，e8 为相对 call 的操作码，后面的 8d e7 ff ff 为相对偏移量。图 3.7 是 UltraEdit 查找到的对应代码，其中深色块标注的就是 DM3-44 粗体所示的机器码，虚线标注的最后 3 字节是 4D 08 51，其后就是 E8，即 call 指令的操作码。但它后面的 4 字节偏移量是全 0。DM3-44 中倒数第 2 行地址 0041287d 对应的机器码是 83 C4 08，与图 3.10 中实线标注的完全吻合，说明仅仅是 call 指令的偏移量被置 0，其余都与运行期的机器码一致。再继续看 DM3-44 中 fwrite()对应的 call 指令也会发现，操作码后的 4 字节被清零。

图 3.10　obj 中的机器码

为什么要将转跳的偏移量清零？被调用函数的机器码可能在其他 OBJ 文件中，只有链接成一个执行程序时，链接器才知道被调用函数入口到调用代码的偏移，在仅仅生成 OBJ 文件的编译期无法知道这个偏移量。(也许清零只是指示它无效，总之链接时要重新计算。)链接器怎样知道这里的 call 指令到底要调用哪个函数？这段机器码没有记录任何与函数名相关的信息，所以应该是存到其他地方了。现在我们不知道 OBJ 文件的格式，也无法知道链接器从哪里获取被调用函数的名字。我们可以查找资料看看 OBJ 文件是什么格式，并分析这些格式。但在此前可以猜测，再去查找相关资料，就比较容易找到想要的东西。下面是我们的猜测。

这个猜测受 1.8.2 节图 1.72 所示的重定位项的启发。实际只要指出指令中应该被重新计算值的部分是哪个位置，且这种重计算属于什么类型即可。以图 3.10 为例，其实就是箭头所示的位置（离文件头部偏移 2e9fh 处）开始的 4 字节要重新计算。给出该部分离文件头部的偏移量，即可标识该位置。重计算为什么要给出类型？在本例中，很明显这需要计算距离被调用函数的偏移量。我们必须知道这是函数调用偏移量的重计算，需获得被调函数的名字，才能在链接期重计算。假如源代码是对全局变量的赋值 gi = 12，且 gi 并非定义在本赋值语句所在的 OBJ 文件中，那么，该语句对应的 mov 指令中要被重计算的就是链接后 gi 的地址值。因为编译时无法确定 gi 将放在数据段的哪个位置，这时重计算逻辑（取得 gi 的地址）明显与函数调用偏移量的计算逻辑（取得偏移量）不同，所以重计算应该有类型区别。我们可以假想如下数据结构和重计算逻辑：

```
struct Recalculation {
    int offset;   //离文件头部偏移量，在图 3.10 的例子中，就是箭头所示字节离文件头部偏移
    int type;                    //重计算类型
    char symbolName[];       //函数或全局变量名字
};
recalculate() {
  1 获取 OBJ 文件中重计算信息，即 Recalculation 表（一个元素为 Recalculation 的硬盘数组）
  2 遍历该 Recalculation 数组，计算新的值:
    如果 type 是函数调用的偏移量计算，则用 symbolName 查找链接后该函数的入口地址，并
与当前 call 指令后指令的地址相减，然后将相减的结果写入 Recalculation.offset 指示的位置
    如果 type 是全局变量地址计算，则用 symbolName 找到链接后该变量的地址，并将该地址
写入 Recalculation.offset 指示的位置。
}
```

下面来看真正的 OBJ 文件格式与我们猜测的有何异同。在百度上输入“VC OBJ 文件格式”，容易知道它是 COFF 格式。然后查找 COFF 格式解释，也能找到资料。剩下就是找一款 COFF 查看器。这次在百度上搜索结果不佳，求助 Google，检索关键字“COFF viewer”，很容易看到一款叫 PEview 的软件。(**下载后用它打开 simplediskarray.obj，边消化资料边对照查看器分析。**)

COFF 文件由文件头（file header)、节头（section header)、节（section)、表（table）数据块组成。节存放数据、机器码，其数目不确定。节头是对节属性的描述，相当于索引表，能定位到与节相关的各种信息。表有符号表（symbol table)、字符串表（string table)、重定位表（relocation table）和行号表（Line Numbers table)。节中一些信息都与表相关，见图 3.11（现在只看最左边的表即可，该图是我们做实验整理出来的，实线指示位置，虚线代表内部结构的展示)。

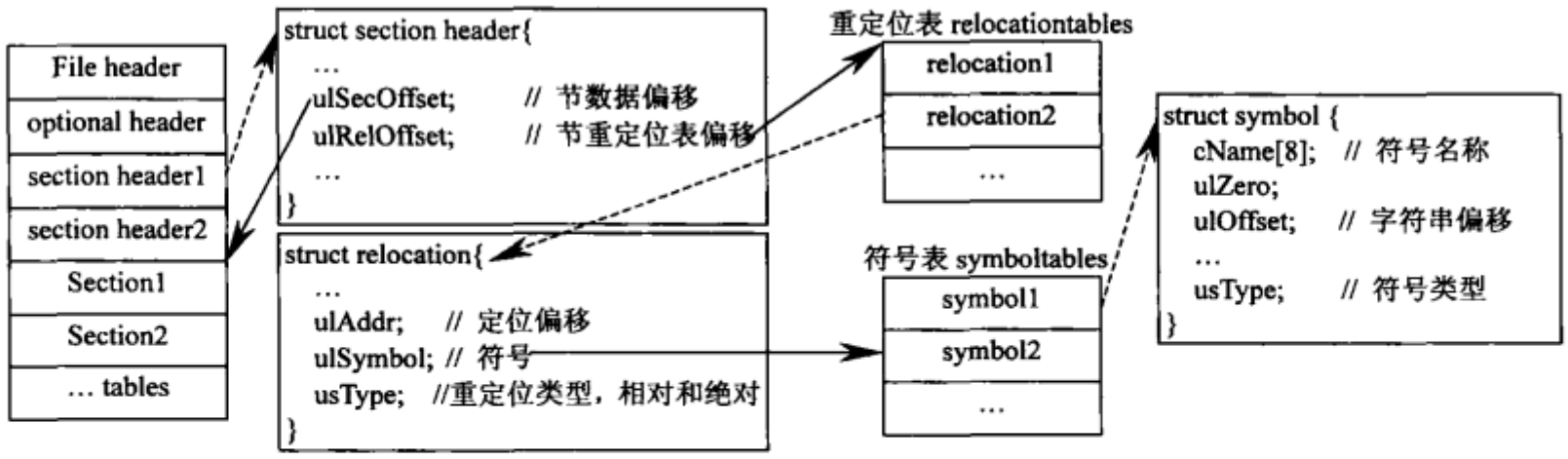


图 3.11　COFF 中重定位相关结构

导图 3.1 m7

哪个与重计算相关？单单看节和表的名称、字符串表、行号表等，都不太像。剩下 relocation 表（重定位表），资料上又没说清楚。**（重定位什么？重定位，重计算，似乎两者有某种文字上的神似。）**我们前面猜测的重计算过程似乎可称为重定位，因为是重新计算（定位）指令中的地址信息。**（怎样验证这个猜测？）**既然重计算与指令相关，而指令又放在节中，那么我们可查看节的描述（section header）中是否有与重计算有关的信息，如果有且只有一个，即 relocation 表的话，则该表很可能与重计算相关。图 3.11 中，节头（section header）有多个字段指向了相关信息块（由相对偏移表示），信息块包括一个节（ulSecOffset 指向节的起始）和多个表。节中包含的是指令和数据，自然不应包含重计算信息。这些被指向的表从名字和解释看大多与重计算无关，如 ulLNOffset 指向行号表。唯有 ulRelOffset 指向 relocation 表，这与我们之前的猜测已隐隐相对。因此，下面从该表下手。**（COFF 结构复杂难明，唯有先猜测，才能快速缩小学习范围。）**

要进一步确认 relocation 表是否包含了重计算信息，我们必须分析其格式。relocation 表是一个数组，由多个 struct relocation（见图 3.11 中部下方）组成。这一点与我们之前假想的 Recalculation 表一样，也由多个 struct Recalculation 组成。因此，只需对比 struct relocation 和 struct Recalculation 的成员字段，即可找到答案。

先对比 Recalculation 的 offset。从资料上看，relocation 的成员 ulAddr（资料描述为“ulAddr 是要定位的内容在段内偏移”）和 offset 可能等价（后待证明），虽然无法确定这个“要定位信息”是什么。同时两者有点差异：ulAddr 是用节内偏移表示位置（节省空间，段内偏移可用较短的整数表示），我们的 offset 是用文件头部偏移表示位置，但两者本质上等价。

导图 3.1 m8

再分析 Recalculation 的 symbolName。COFF 的 relocation 项没有直接包含符号名，但其成员 ulSymbol 是重定位所涉及符号在符号表中的索引。用该索引能检索到符号项，而符号项的结构（见图 3.11 中的 struct symbol）中有符号的名字信息（长度小于 9，从 cName 获取，否则 ulZero 为 0，将用 ulOffset 从字符串表中检索到对应的名字信息）。所以，通过 ulSymbol 可以得到 symbol 名。因此，我们认为，Recalculation.symbolName 和 relocation.ulSymbol 是等价的。**（为什么 COFF 的重定位项不像我们假想的那样直接包含被引用的符号名，而通过 ulSymbol 索引获取？猜测与实际的差异就是我们学习的机会（下面马上又将遇到），搞清楚，就是成长。）**符号会被多次引用，如我们的设计，符号名将多次出现在重定位项中，比较耗费空间。如果是一个 ulSymbol 索引，则大大节约了空间。

最后是 Recalculation.type。relocation 有一个同名的 type 字段，资料显示类型为：绝对定位和相对定位。回头来看我们的 recalculation 伪代码：

“如果 type 是函数调用的偏移量，计算……；如果 type 是全局变量地址，计算……”

第一个“如果...”描述的就是相对偏移定位，第二个“如果...”描述的就是绝对定位。在我们看来，函数名造成的重定位就是相对定位（DM3-41 中第 2 行函数调用产生的是 call offset 指令，参考其反汇编 DM3-44 中地址 00412878 的指令，所以是相对定位），从图 3.11 中所示符号项 struct symbol 的 usType 成员可知符号是否为函数，从而得出是否为相对定位。既然从 symbol.usType 能判断是否相对定位，为什么 COFF 还要在 relocation 中再定义一个 type？难道有的函数名并非产生相对定位？（这个疑问就是我们猜测、实证方法的一个关键点：自己和别人做法的差别往往是关键所在，如果目前确实搞不清楚，也要记下将来再琢磨。）注意，源代码出现函数名的地方并非都是调用函数，如果将函数的入口地址赋给一个函数指针变量“pfunc = mytest;”，虽然这条语句的重定位也是因为 mytest 符号造成的，但它是绝对定位。因为该语句被翻译成一条 mov 指令，指令的一部分就是函数 mytest()的入口地址，需要在链接时才能计算出（重定位）。重定位时并不计算相对值，只是将函数入口地址直接填入 mov 指令的对应部分即可。

从图 3.11 的 struct symbol 中我们能获取到符号的类型（usType 指示是函数还是变量）。如果是相对定位，产生重定位的符号又是函数，则偏移量=调用函数入口-(重定位处+4 字节)，加 4 字节是因为相对转移指令（jmp、call）从该指令的下一条计算偏移；如果是绝对定位，则直接将符号的地址写入重定位处即可。与我们猜测的算法对比，有了新的算法假设，见 DM3-45。

DM3-45

```
void relocate(){
  从每个 section header 的 ulRelOffset 获取到重定位表 relocationtables，见图 3.11
  遍历该表每个重定位项 relocationtables[i]，做如下任务：
    1. 从 relocationtables[i].ulSymbol 获取到需要重定位的符号的描述 struct symbol；用 symbol.cName 或 symbol.ulOffset 获取到符号的名字 symbolName；用 symbolName 在链接后的全局表中查找到该符号对应的绝对地址 symbolAddress（如果是函数就是入口地址，全局变量就是变量地址）。
    2. 用 relocationtables[i].ulAddr + 段偏移（即 section header 的 ulSecOffset，估计是段离文件头部的偏移，需要证明）计算重定位离文件头部偏移量 actualRelOffset。
    3. if relocationtables[i].type 是绝对定位 {
         从文件头部偏移 actualRelOffset，写入 symbolAddress
       }
       else {                                                    //是相对定位
         if symboletables[relocationtables[i].ulSymbol].usType 是函数（symboletables 见图 3.11）
           从文件头部偏移 actualRelOffset，写入相对偏移值（symbolAddress-(重定位处的地址+4)）
       }
}
```

到此，一切都是假设，需要用 PEView 查看验证。思路如下：从 section header 中找到它指向的 section 的位置，然后分析图 3.10 中机器码的离文件头的偏移量是否在该 section 中；如果找到，则从 section header 中获得重定位表入口，从重定位项的 ulAddr 值分析是否它就是指向图 3.10 箭头所示位置（**这个位置是我们的猜测，需要证实**）。

用 PEview 打开 simplediskarray.obj，图 3.12 是 section header 的信息，实线标出了 section 的偏移量 2E66（估计是离文件头部的偏移量），虚线是重定位表的偏移量 2EED。下面需要找到对应 section 的数据，在 PEview 中选择每个 section 查看，其内容见图 3.13，找到一个 section 起始的偏移量是 00002E66（第 1 行 VA 列的量）。这个偏移是否是离文件头部的偏移？只需在 UltraEdit 中查看 00002E66 处的内容是否为图 3.13 所示，如是，则说明确该偏移是离文件头部偏移量。Ultraedit 的内容见图 3.14，其中标注的块所含字节与图 3.13 中 00002E66 开始的块的内容一模一样。由此证明，section header 中指示某个块的位置是用相对文件头部偏移量描述的。哪个 section 包含了图 3.10 所示的机器码？从图 3.10 可知，机器码到文件头部偏移为 2E90H 左右，我们只需遍历 PEview 中的每个以 .text 命名的 section header 的图标（从资料可知，.text 是代码节的默认名字，要查找包含机器

VA	Data	Description
00000424	2E 74 65 78	Name
00000428	74 00 00 00	
0000042C	00000000	Virtual Size
00000430	00000000	RVA
00000434	00000087	Size of Raw Data
00000438	00002E68	Pointer to Raw Data
0000043C	00002EED	Pointer to Relocations
00000440	00000000	Pointer to Line Numbers
00000444	0004	Number of Relocations
00000446	0000	Number of Line Numbers
00000448	60501020	Characteristics

图 3.12　一个 section header 的结构

VA	Raw Data
00002E66	55 8B EC 81 EC C0 00 00 00 53 56 57 8D BD 40 FF
00002E76	FF FF B9 30 00 00 00 B8 CC CC CC CC F3 AB 8B 45
00002E86	0C 83 C0 01 8B 4D 08 3B 41 08 7E 04 33 C0 EB 43
00002E96	8B 45 0C 50 8B 4D 08 51 E8 00 00 00 00 83 C4 08
00002EA6	8B F4 8B 45 08 8B 08 51 6A 01 8B 55 08 8B 42 04
00002EB6	50 8B 4D 10 51 FF 15 00 00 00 80 83 C4 10 3B F4
00002EC6	E8 00 00 00 00 83 F8 01 74 04 33 C0 EB 05 B8 01
00002ED6	00 00 00 5F 5E 5B 81 C4 C0 00 00 00 3B EC E8 00
00002EE6	00 00 00 8B E5

E8 后那个 00 到文件头部偏移为 2e9fh

图 3.13　section header 对应 section 的内容

```
00002e60h: 6C 00 00 00 0A 00 55 8B EC 81 EC C0 00 00 00 53
00002e70h: 56 57 8D BD 40 FF FF FF B9 30 00 00 00 B8 CC CC
00002e80h: CC CC F3 AB 8B 45 0C 83 C0 01 8B 4D 08 3B 41 08
00002e90h: 7E 04 33 C0 EB 43 8B 45 0C 50 8B 4D 08 51 E8 00
00002ea0h: 00 00 00 83 C4 08 8B F4 8B 45 08 8B 08 51 6A 01
00002eb0h: 8B 55 08 8B 42 04 50 8B 4D 10 51 FF 15 00 00 00
00002ec0h: 00 83 C4 10 3B F4 E8 00 00 00 00 83 F8 01 74 04
00002ed0h: 33 C0 EB 05 B8 01 00 00 00 5F 5E 5B 81 C4 C0 00
```

图 3.14　ultraedit 显示的 section 内容

VA	Data	Description	Value
00002EED	00000039	RVA	
00002EF1	00000055	Symbol Table Index	?seekSimpleDiskArray@@
00002EF5	0014	Type	IMAGE_REL_I386_REL32
00002EF7	00000057	RVA	
00002EFB	00000020	Symbol Table Index	__imp__fread
00002EFF	0006	Type	IMAGE_REL_I386_DIR32

图 3.15　section 关联的重定位表

码的节，自然是查找代码节.text)，查看图 3.12 中实线标注的哪个字段（section 的起始偏移量）的值与 2E90H 接近。很容易找到图 3.12 所示的 section。section 的起始偏移为 00002E66H，其上一字段注明为“size of Raw Data”，说明是 section 的大小。其值为 87H。也就是说，该 section 的结尾离文件头部的偏移是 2E66H+ 87H= 2EEDH。我们关注的机器码的偏移值 2E90H 满足 2E66H<2E90H<2EEDH，属于该 section。现在既然找到了该 section。我们需要找它的重定位表，如图 3.12 的虚线所示，重定位表起始处相对偏移为 00002EEDH。单击其 section 项紧邻的 IMAGE_RELOCATION 项，可看到图 3.15 所示的重定位表。其第一项的相对偏移（第一列 VA 列的值）正是 00002EED。可见，该表就是我们要找的重定位表。还可发现，该表应该对应 DM3-40 而非 DM3-41，因为图 3.15 还展示了一个函数相关的重定位项_imp_fread，它是属于 DM3-40 第 3 行调用 fread()产生的重定位。从图 3.15 第一项 Symbol table index 成员所指的 Value 即 seekSimpleDiskArray 来看，这个重定位项就是由调用 seekSimpleDiskArray 产生的。而该项的 RVA 成员（即前面 struct relocation 的 ulAddr）的值是 00000039H，即 39H 是要被重定位的位置。**(记住，它可能是相对 section 首部的偏移。)** 图 3.12 所示本 section 首部的偏移量是 2E66H，加上 39H，就是重定位处相对文件头部的偏移 2E66H + 39H=2E9FH。图 3.10 中猜测的那个箭头所示的重定位值的位置是 2E9FH（UltraEdit 的这个位置是从文件头部开始计算的），也是图 3.13 箭头所示的位置**(猜测和实证完全吻合!)**，relocation.ulAddr 是段内偏移。

另外，从图 3.15 可知，重定位表的 4 个项正好就是 DM3-40 中函数 getSimpleDiskArray()的 4 次函数调用产生的重定位。**(另外 2 次函数调用是因为在 Debug 版下，对动态链接库函数 fread()的调用，会在调后用 CheckEsp()函数检测栈是否出现调用惯例不匹配的错，此时 C 库以动态库方式引用，而这个函数本身退出前也要调用 CheckEsp 检测，所以多了 2 次隐藏的调用。)**

到此还没有完毕，实证只是证明了重定位项中的 ulAddr 的位置如预期，但 DM3-45 涉及的几个问题尚未验证：① 是否用函数名调用产生的重定位项其 type 是相对定位，而函数指针赋值产生的重定位项就是绝对定位，全局变量产生的重定位项是否是绝对定位；② 当重定位由函数名产生时，其 ulSymbol 对应的符号属性是否确实是函数，全局变量产生的符号属性是否就是变量。

关于问题①，从图 3.15 第 3 行可见，type 属性的值是 14H，右侧 Value 列对该值的描述为 IMAGE_REL_i386_REL32，自然猜想到 REL32 就是 relative32 的缩写，即相对定位。不过还是让我

们写一个程序来验证该猜测，见 DM3-46。工程见“code\第 3 章\3-3\coff_obj\test”。

明显第 1 行应该产生绝对定位，第 2 行应该产生绝对定位，第 3 行应该产生相对定位。编译后，用 PEview 打开 test.obj，只有 1 个与 .text 相关的重定位节 IMAGE_RELOCATION（紧跟 .text 节后），包含符号 gi 和 testfunc 的就是我们要找的节，见图 3.16。注意：tesfunc 符号有两个重定位项，分别为 1 和 3，第 2 项是 gi 的重定位项。这个顺序与 DM3-46 的 main()中语句的顺序一致，基本可以推断第一个重定位项是第 1 行产生的，第 3 项是第 3 行产生的。重定位项 1 和 3 的类型不同，前者为 IMAGE_REL_i386_DIR32，后者为 IMAGE_REL_i386_REL32。因为第 1 行翻译为“mov [ebp + …], testfunc 地址”，所以该指令需要的是 testfunc 的地址，不是相对偏移，是绝对定位。IMAGE_REL_i386_DIR32 中的“DIR32”即“directly32”，即直接寻址。根据第 2 项对全局变量 gi 的重定位 type 也是 DIR32 类型，也可以证明。如果还不放心第 1 项是代码第 1 行产生的，只需按照前面的分析，用 UltrEdit 查看该位置的机器码是否与程序运行调试的反汇编结果相同即可完全确定。

DM3-46

```
int gi;
void testfunc() {
}
void main(){
1   void (*f)() = testfunc;
2   gi = 12;
3   testfunc();
}
```

VA	Data	Description	Value
000009B2	00000021	RVA	
000009B6	00000012	Symbol Table Index	?testfunc@@YAXXZ
000009BA	0006	Type	IMAGE_REL_I386_DIR3
000009BC	00000027	RVA	
000009C0	0000000B	Symbol Table Index	?gi@@3HA
000009C4	0006	Type	IMAGE_REL_I386_DIR3
000009C6	00000030	RVA	
000009CA	00000012	Symbol Table Index	?testfunc@@YAXXZ
000009CE	0014	Type	IMAGE_REL_I386_REL

图 3.16 test.obj 的重定位表

第 2 个问题的验证比较简单，如图 3.16 的第一项第 2 行是对应符号在符号表中的索引——12。单击 PEview 左边列表中唯一的一个图标“IMAGE_SYMBOL Table”，并注意右边展示的每个符号项第一行给出的索引值，找其值为 12 的即可（注意从实践看，这个索引 12 是十六进制数）。其类型为 DT_FUNCTION，十六进制值为 20H，同理找到 gi 的值为 0。

这次探索学习中，我们不仅对链接的原理深入了一层，还再次体会了猜测、实证方法在学习未知事物方面的强大威力。希望读者多多体会。

3.3.3 分享它（2）：我的丑陋链接器

到此，我们的探索似乎应该告一个段落了——猜测、实证、构建。DM3-45 算法真的有效吗？虽然 3.3.2 节从很多方面有所实证，但毕竟没有最直接的证明。能否“构建”一个简单的验证程序实现该算法？链接器？太难了。在学习探索中，一定要想办法化简，否则真实的问题都很复杂，岂非永远无法验证？简化的关键是找出主要矛盾并加以解决。如同 1.8.2 节中为了构建我们想象中的重定位过程，其代码非常简化，以内存代替文件，并只实现了一个简单 mov 指令的重定位。因为其关键是验证重定位逻辑（定位表、重定位计算等），不是重定位代码从哪里来，也不是要做太复杂代码的重定位，所以需要删繁就简。

当下关键矛盾是：从 OBJ 中找到代码，并处理其中重定位，将可运行的代码提取出来，以 main()函数为入口，将所有相关函数组装成执行代码（不是程序）。所以我们不链接一个完整的执行程序，否则意味着要掌握 COFF 文件所有项的含义和细节，以及 PE 执行文件格式的细节（这是一个短期内不可能完成的任务）。具体方案如下：

从 OBJ 文件中找到函数 f1()，f1()使用了全局变量 gMsg，还调用本 OBJ 文件中的另一个函数 f2()。在 f2()中，将该全局变量 gMsg 的值打印。本方案将验证函数调用相对定位和全局变量的绝对定位。编写一个程序，加载该 OBJ 文件，对代码进行重定位处理，并分配 gi 的内存，返回 f1()的指针给程序调用。本方案不涉及链接成执行程序，所以回避了 PE 格式的复杂性，而只处理这个简单场景，又不必关注 COFF 格式的过多细节。细想这个方案，有点麻烦，f2()中要调用 printf()，printf()

不在该 OBJ 文件中，需要从 C 运行库中链接，这就复杂了。（**能否简化？**）将打印函数的指针作为参数传给 f2()，那么 f2()就不用关注链接 C 库的问题了。

开始编程时，我们遇到了问题：如果无法获得 f1()函数的入口地址及其长度，就完成不了以上任务。其实 DM3-45 中的 symbolAddress 如何获取也是目前该问题的另外一个说法。（**这就是“构建”在探索三部曲中的重要性。不构建，一些关键性细节会被忽略，构建程序就迫使我们在此过程中某个时候直面它们。构建就是一个“最好的、严格的、善于发问的老师”。**）

从资料上查不到清晰说明，那就“猜测”吧。分析 3.3.2 节的 testfunc 和 main，根据图 3.11，它们最合理的存放场所应该在符号项 symbol 中。既然符号名、类型什么的都在里面，“入口地址”是否也在里面呢？用 PEview 查看符号项的结构，见图 3.17 和图 3.18。从名字来看，图 3.17 中标注的项 Section Number 有点像。因为它表明了函数 testfunc 所在的节。可是只给了节，如果一个节中有多个函数，没有节内偏移也不行。Value 似乎可以扮演这个节内偏移的角色。（验证如下。）DM3-46 中只有两个函数 testfunc()和 main()。调试跟踪用反汇编，得到它们的机器码，再将机器码和 PEview 中的两个 .text 节对比，确定 testfunc()和 main()分别属于哪个节。

		Symbol Table Index	00000012
00000CC8	00000000	Long Name	?testfunc@@YA
00000CCC	00000004	Offset into String Table	
00000CD0	00000000	Value	
00000CD4	0004	Section Number	.text
00000CD6	0020	Type	DT_FUNCTION
00000CD8	02	Storage Class	IMAGE_SYM_C
00000CD9	00	Number of Aux Symbols	

图 3.17　符号项的结构（一）

		Symbol Table Index	00000023
00000DFA	5F 6D 61 69	Short Name	_main
00000DFE	6E 00 00 00		
00000E02	00000000	Value	
00000E06	0006	Section Number	.text
00000E08	0020	Type	DT_FUNCTION
00000E0A	02	Storage Class	IMAGE_SYM_
00000E0B	00	Number of Aux Symbols	

图 3.18　符号项的结构（二）

DM3-47 给出了 main()的反汇编，粗体部分是机器码，将其放在一起如 DM3-48(a)所示。斜体部分是在链接期必须重定位的，如果用 00 将其替换，那么该段机器码的值如 DM3-48(b)所示。

DM3-47

```
004113d0 55                    push …
004113d1 8B EC                 mov  …
004113d3 81 EC CC 00 00 00     sub  …
004113d9 53                    push …
004113da 56                    push …
004113db 57                    push …
004113dc 8D BD 34 FF FF FF     lea  …
004113e2 B9 33 00 00 00        mov  …
004113e7 B8 CC CC CC CC        mov  …
004113ec F3 AB                 rep  stos…
    void (*f)() = testfunc;
004113ee C7 45 F8 AA 10 41 00  mov…
    gi = 12;
004113f5 C7 05 40 71 41 00 0C 00 00 00 mov testfunc();
004113ff E8 A6 FC FF FF        call …
}
00411404 33 C0                 xor  …
00411406 5F                    pop  …
00411407 5E                    pop  …
00411408 5B                    pop  …
00411409 81 C4 CC 00 00 00     add  …
0041140f 3B EC                 cmp  …
00411411 E8 2A FD FF FF        call …
00411416 8B E5                 mov  …
00411418 5D                    pop  …
00411419 C3                    ret
```

DM3-48

(a)

```
55 8B EC 81 EC CC 00 00 00 53
56 57 8D BD 34 FF FF FF B9 33 00 00 00 B8 CC
CC CC CC F3 AB C7 45 F8 AA 10 41 00 C7 05
40 71 41 00 0C 00 00 00 E8 A6 FC FF FF 33 C0
5F 5E 5B 81 C4 CC 00 00 00 3B EC
E8 2A FD FF FF 8B E5 5D C3
```

(b)

```
55 8B EC 81 EC CC 00 00 00 53
56 57 8D BD 34 FF FF FF B9 33 00 00 00 B8 CC
CC CC CC F3 AB C7 45 F8 00 00 00 00 C7 05
00 00 00 00 0C 00 00 00 E8 00 00 00 00 33 C0
5F 5E 5B 81 C4 CC 00 00 00 3B EC
E8 2A FD FF FF 8B E5 5D C3
```

```
00000968  55 8B EC 81 EC CC 00 00  00 53 56 57 8D BD 34 FF
00000978  FF FF B9 33 00 00 00 B8  CC CC CC CC F3 AB C7 45
00000988  F8 00 00 00 00 67 05 00  00 00 00 0C 00 00 00 E8
00000998  00 00 00 00 33 C0 5F 5E  5B 81 C4 CC 00 00 00 3B
000009A8  EC E8 00 00 00 00 8B E5  5D C3
```

节的起始偏移

图 3.19

查看两个.text 节的内容，发现其中一个如图 3.19 所示，与 DM3-48(b)中的内容完全一样！

也就是说，该.text 节的开始就是 main()函数的机器码，即 main()起始位置的节内偏移为 0。这与图 3.18 的 Value 值相同。同理，可发现 testfunc 在另一个.text 节的起始，这与图 3.17 中的 Value 为 0 也是一致的，所以 Value 可能是节内偏移量。我们还需找到一个节内有两个 symbol 的例子，通过在源代码中添加更多函数，编译后查看，最终证明这个猜测：Value 是符号在节内的偏移。现在要从 section number 成员找到 section 起始地址。但从资料上看，section number“当它大于 0 时，才是符号所在节的索引值”，却没有提到索引值从 1 开始，还是从 0 开始。(**验证。**) 图 3.18 中，main()函数的 section number 是 8。从文件头后数 8 个节头，正好是一个 .text 的头，见图 3.20，显示该节偏移是 968H。图 3.19 显示的节起始位置的偏移量就是 00000968（第 1 行最左侧的偏移量）。由此可知，从符号表项中 section number 获得的是从 1 开始计数的节头编号（而非 0 开始的索引），然后找到节头中的节起始偏移量及节的大小（见图 3.20），就可以将某个函数符号的对应代码块找到。至此，要实现本次目标的所有技术要点都搞定了。(**一直没有分析，如果函数等符号在其他 OBJ 文件中定义如何处理，见习题 11。**)

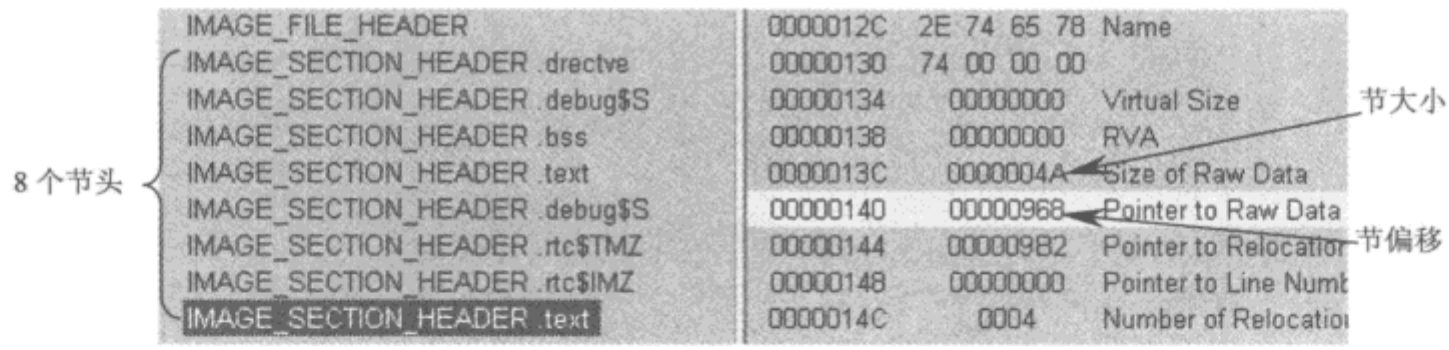


图 3.20　节的验证

开始“构建”。编写要被链接用的代码，见 DM3-49，文件见“code\第 3 章\3-3\coff_obj\objtest_for_linker\mytest.cpp”。DM3-49 会测试链接的特点：① f2()使用了全局变量 gMsg，需要绝对重定位；② f1()对 gMsg 赋值，需要绝对重定位；③ f1()调用 f2()，call 指令是用的偏移量，所以是相对重定位。在控制台中，用 cd 命令将目录定位到 mytest.cpp 所在目录，然后执行“cl /c mytest.cpp”，即可生成 mytest.obj，它将用于将来的链接。

DM3-49

```
typedef void (*FuncPtr)(char * msg);
char * gMsg;
void f2(FuncPtr pfunc){
   pfunc(gMsg);
}
void f1(FuncPtr pfunc, char * msg){
   gMsg = msg;
   f2(pfunc);
}
```

为了找到 f1()的入口地址，必须比较符号项的名字，直到发现“f1”。先看符号项的声明。ulZero 不为 0，则名字是一个短名字，用 cName 的 8 个字符表示，否则在字符串表中查找长名，在字符串表中的偏移为 ulOffset。但字符串表在哪里？COFF 文件头结构见 DM3-50。但其中只能看到 ulSymbolOffset 可定位符号表，其他都没有。从资料的文件示意图来看，字符串表在符号表的后面。如果真的是这样，那么从文件头获得符号表的起始，再用其 ulNumSymbol * sizeof(COFF_Symbol)计算该表大小。从符号表起始偏移表大小就定位出字符串表起始了。可是网上的资料总是不严谨，只字未提“结构如图排列”。那就只有动手验证了。用 PEview 打开前面编译的 mytest.obj 文件，单击符号表和字符串表的图标，查看符号表最后一项的偏移是 2D7H，字符串表的第一项偏移是 2D8H，说明两者首尾相接。现在需要验证是否 DM3-51 中的 ulOffset 就是从字符串表起始处计算偏移，“?f2@@YAXP6AXPAD@Z@Z”的长度大于 8，所以要用 ulOffset

DM3-50

```
struct COFF_FileHdr {
    unsigned short usMagic;          //魔法数字
    unsigned short usNumSec;         //节数
    unsigned long ulTime;            //时间戳
    unsigned long ulSymbolOffset; //符号表偏移
    unsigned long ulNumSymbol;  //符号数
    unsigned short usOptHdrSZ;   //可选头长度
    unsigned short usFlags;          //文件标记
};
```

DM3-51

```
struct COFF_Symbol {
    union {
        char cName[8];              //符号名称
        struct {
            unsigned long ulZero;   //字符串表标识
            unsigned long ulOffset; //字符串偏移
        } longName;
    } name;
    unsigned long ulValue;          //符号值
    short iSection;                 //符号所在节
    unsigned short usType;          //符号类型
    unsigned char usClass;          //符号存储类型
    unsigned char usNumAux;         //符号附加记录数
};
```

索引字符串表，见图 3.21 中的标注，即 DM3-51 中的 ulOffset。因为 COFF_Symbol 开始是一个 union，所以 ulOffset 是图 3.21 中的第 2 项，其值为 11h（十进制为 17）。图 3.22 为字符串表，其中实线标注了 13 字节，该行上面是4字节(第1行和第2行的偏移量2DCH–2D8H，即第 1 行 4 字节)，所以 13+4=17。串表头偏移 17 字节正是图 3.22 中的“?f2@@...”，这说明 ulOffset 确实是字符串表头的偏移。容易计算出图 3.22 第 1 行的 3AH 就是整个字符串表（包含开始的 4 字节）的字节长度。同时，实线标注的最后是 0，说明字符串以 0 结尾。另外，本来源代码中的“gMsg”变量变成了“?gMsg@@...”，意味着，在符号表中寻找函数 f1()产生的符号时要去掉'? '和'@'来判断。**(注意，短名字如果为 8 字节，会占满 cName 的 8 字节，不用 0 字节结尾。)**

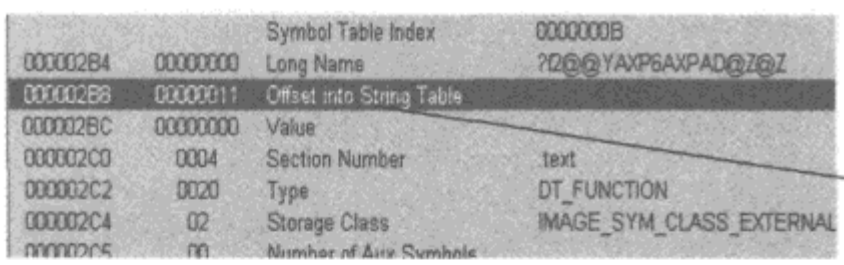


图 3.21 ulOffset 偏移计算（一）

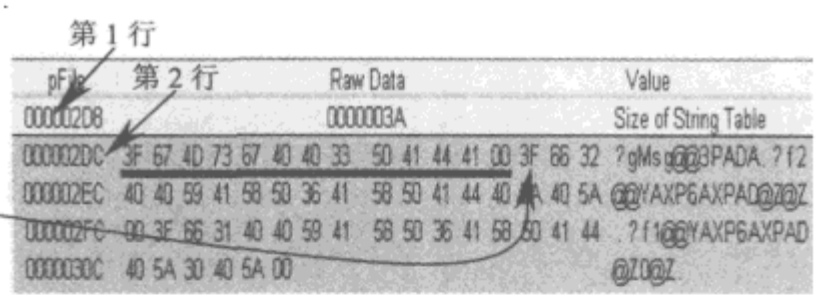


图 3.22 ulOffset 偏移计算（二）

在上面的基础上，我们重新梳理链接算法。遍历所有节头，从 DM3-52 所示的 ulFlags 判断节是否为 .text 节，如是（见习题 19），则从 COFF_SectionHdr 成员 ulSecOffset 获得代码节的起始地址，并通过成员 ulRelOffset（从文件开始的偏移）找到该节的重定位表。遍历重定位项，根据重定位符号是否为函数及重定位性质进行定位（即重计算偏移还是地址）。

如何判断节是否为.text 节？在 PEview 中打开.text 节头，如图 3.23 所示，可以看到对应 ulFlags 的属性（ulFlags 在结构体中为最后一个，Characteristics 也是该图中最后一个，所以两者应该是同一个）。Characteristics 的值由 4 个值 20H、50000H、20000000H 和 40000000H 组成。一个 4 字节的数表示 4 个值，明显是将 4 个值进行了或运算，图中的 60500020H 确实为该 4 个数之或。图 3.23 中标注的 00000020 被描述为 IMAGE_CNT_CODE，应该代表代码段，所以只需用 ulFlag & 0x20 == 0x20 就可验证是否为 .text 节。DM3-53 为主函数逻辑，见“code\第 3 章\3-3\coff_obj\mysimplelinker”。用户在命令行中输入目标文件的名字，第 5 行取得该文件名，第 7 行调用 linkFunction()进行链接，并返回描述函数 f1()的结构体 prg。prg->entry 是该函数的入口地址。第 9 行调用链接好的 f1()，传入 printMsg 指针，用于打印，最终打印“my link ok”。

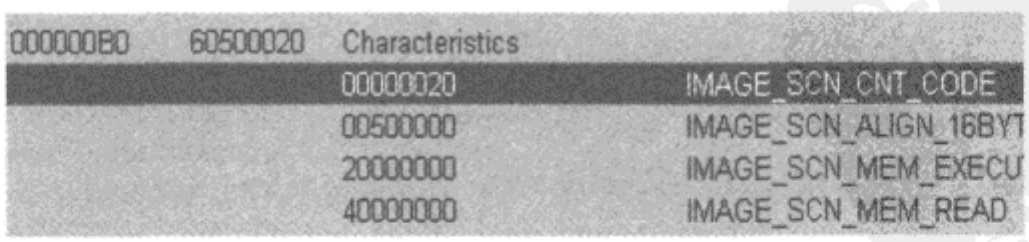


图 3.23

程序通过 initLinkInfo 将整个 OBJ 文件读入内存，见图 3.24，左边为重定位前内存映像，右边为重定位后映像。解析文件，将 LinkInfo 的几个相关成员指针指向内存映像中的对应地址。

DM3-52

```
struct COFF_SectionHdr{
   char cName[8];              //节名
   unsigned long ulVSize;      //虚拟大小
   unsigned long ulVAddr;      //虚拟地址
   unsigned long ulSize;       //节长度
   unsigned long ulSecOffset; //节数据偏移
   unsigned long ulRelOffset; //节重定位表偏移
   unsigned long ulLNOffset;//行号表偏移
   unsigned short ulNumRel; //重定位表元素个数
   unsigned short ulNumLN; //行号表长度
   unsigned long   ulFlags;   //节标识
};
```

DM3-53

```
void main(int argCount, char ** args) {
1    if (argCount != 2) {
2       printf("mysimplelinker C:\temp\abc.obj\n");
3       return;
4    }
5    char * objFile = args[1];
6    char * entryFunctionName = "f1";
7    struct Program * prg = linkFunction(entryFunctionName, objFile);
8    FuncPtr func = (FuncPtr)prg->entry;
9    func(printMsg, "my link ok\n");
10   freeProgram(prg);
}
```

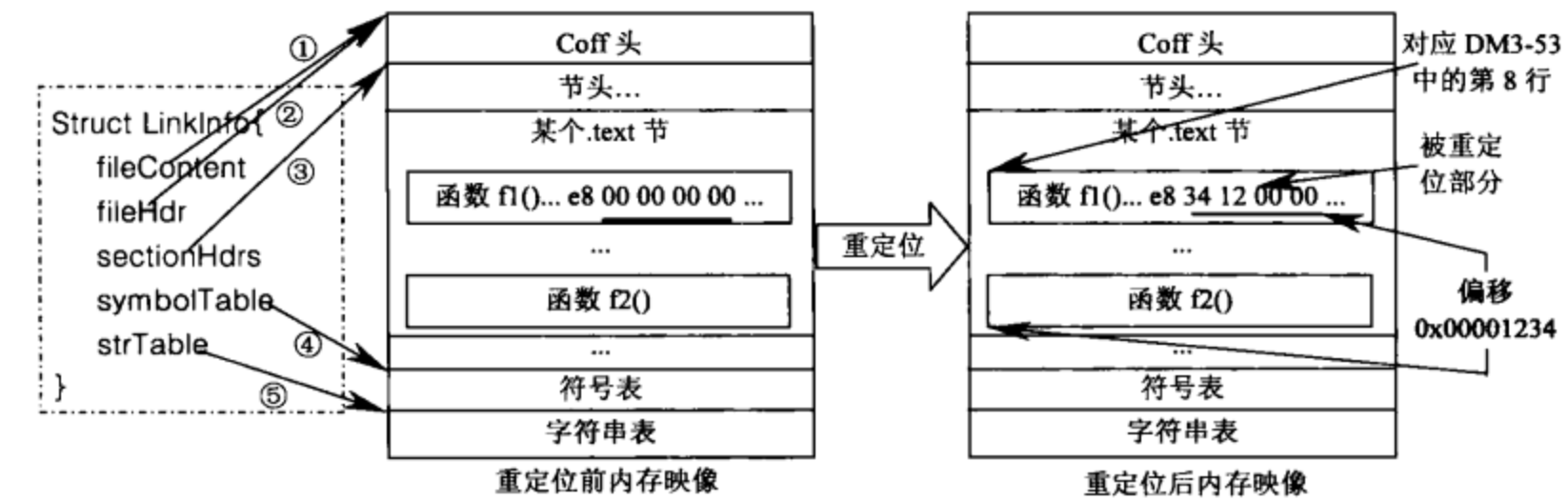


图 3.24　重定位前、后的内存映像

fileContent 是内存首址，也是文件起始的地方，对应图 3.24 中的箭头①，定位代码是 plinkInfo->fileContent = (char *)malloc(size)。

fileHdr 指向文件头，也是首址位置，对应图 3.24 中箭头的②，定位代码是 plinkInfo->fileHdr = (struct COFF_FileHdr *)plinkInfo->fileContent。

sectionHdrs 指向节头，对应图 3.24 中的箭头③，定位代码为：

```
//从文件起始 plinkInfo->fileContent 偏移文件头的大小 sizeof(COFF_FileHdr)
plinkInfo->sectionHdrs=(struct COFF_SectionHdr *)(plinkInfo->fileContent+sizeof(COFF_FileHdr));
if (plinkInfo->fileHdr->usOptHdrSZ != 0) //如果有可选头，那么还要再偏移其大小 usOptHdrSZ
   plinkInfo->sectionHdrs = (struct COFF_SectionHdr *)((char *)(plinkInfo->sectionHdrs) +
                                                    plinkInfo->fileHdr->usOptHdrSZ);
```

symbolTable 指向符号表，对应图 3.24 中的箭头④，定位代码为：

```
//从文件起始偏移文件头中指示的符号表偏移 ulSymbolOffset
plinkInfo->symbolTable = (struct COFF_Symbol *)(plinkInfo->fileContent +
                                                plinkInfo->fileHdr->ulSymbolOffset);
```

strTable 指向字符串表，对应图 3.24 中的箭头⑤，定位代码为：

```
//符号表大小为文件头指示的个数 ulNumSymbol 乘以符号项的大小 sizeof(struct COFF_Symbol)
int symbolTbSize = plinkInfo->fileHdr->ulNumSymbol * sizeof(struct COFF_Symbol);
//字符串表在符号表后面，所以其开始是首址+符号表入口偏移+符号表的大小
plinkInfo->strTable = plinkInfo->fileContent + plinkInfo->fileHdr->ulSymbolOffset + symbolTbSize;
```

图 3.24 中，重定位前 f1()调用 f2()的机器码是 e8 00 00 00 00（左侧），e8 为 call 指令操作码，后 4 字节要重定位成 f1 下条指令到 f2 的偏移。右侧为重定位后的映像，该指令变成了 e8 34 12 00 00，因为偏移为 0x00001234。

调试该程序时出了一个 bug，表现为重定位值不正确。后来发现导致该问题的原因是

sizeof(COFF_Symbol)的值为 20，而不是 PEView 中的 18 字节。在分析了 DM3-51 后，根据 1.5.2 节的对齐错误，我们发现，这个结构体为了对齐出现了填充位，消除填充位即可，即“所见即所得”按 1 对齐。(**这就是基础知识的力量**。) 如果按图 1.54 的方式，整个程序都将按 1 对齐。如何让一些指定的结构体按某种方式对齐？笔者其实也不清楚，但基础知识的力量又发挥作用了。在 MSDN 中输入“align”就能最终找到答案，见 DM3-54。开始行代表下面的对齐方式要用 1 对齐，而最后一行表示恢复第一行前的对齐方式。这样，两行间的结构体都按 1 对齐。

DM3-54

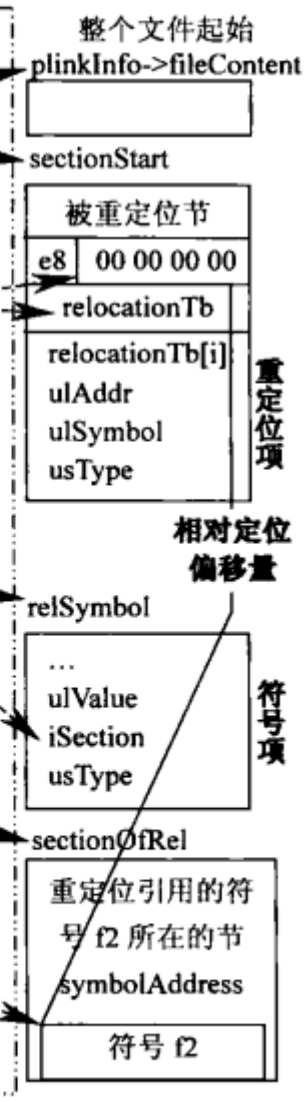
```
#pragma pack(push, 1)

struct COFF_FileHdr{
  unsigned short usMagic;   //魔法数字
  ...
};
....
#pragma pack(pop)
```

限于篇幅，本文只给出了关键重定位代码见 DM3-55。由于程序开始将整个文件读入内存，所以 plinkInfo->fileContent 指向该内存的起始，即相当于指向文件开始。

DM3-55

```
void relocateSection(struct LinkInfo * plinkInfo, struct COFF_SectionHdr * sectionHdr){
1 int relocationCount = sectionHdr->ulNumRel;
2 char * sectionStart = plinkInfo->fileContent + sectionHdr->ulSecOffset;
3 if (relocationCount > 0)                    //if has relocation {
4   struct COFF_Relocation * relocationTb = (struct COFF_Relocation *)(plinkInfo->fileContent +
5                                                                     sectionHdr->ulRelOffset);
6   for(int i = 0; i < relocationCount; i++) {
7     char * symbolAddress;
8     char * relocationAddress = sectionStart + relocationTb[i].ulAddr;
9     int symbolIndex = relocationTb[i].ulSymbol;
10    struct COFF_Symbol * relSymbol = plinkInfo->symbolTable + symbolIndex;
11    if (relSymbol->usClass != IMAGE_SYM_CLASS_EXTERNAL)
12      fatalError("i do not know how to relocate non IMAGE_SYM_CLASS_EXTERNAL symbol\n");
13    char * sectionOfRel = GET_SECTION_ADDRESS(plinkInfo, relSymbol->iSection - 1);
14    symbolAddress = sectionOfRel +   relSymbol->ulValue;
15    if (relSymbol->usType == COFF_SYMBOL_TYPE_FUNC) {
16     if (relocationTb[i].usType == IMAGE_REL_I386_REL32)
17       symbolAddress = (char *)(symbolAddress - (relocationAddress + 4));
18     } else {
19       if (relocationTb[i].usType != IMAGE_REL_I386_DIR32)
20           fatalError("i do not know how to relative relocate a symbol not function\n");
21     }
22    *((void **)relocationAddress) = symbolAddress;          //relocate the address
23 }
24}
}
```



函数检查 sectionHdr 指向的节是否需要重定位并处理。第 1 行获取该节的重定位项数目。第 2 行用文件头地址 plinkInfo->fileContent+节的偏移 sectionHdr->ulSecOffset 获得被重定位节的起始地址 sectionStart。第 3 行判断是否有重定位项，如果有，则遍历每个项并重定位。第 4 行用文件头地址+节头指出的重定位偏移 ulRelOffset 获得重定位表的入口地址 relocationTb。第 8 行确定要被重定位改写的地址 relocationAddress。假定被重定位的是一条相对 call 指令 e8 00 00 00 00，那么 relocationAddress 就是指向的 e8 后的第一个 00。第 9 行获得产生重定位的符号在符号表中的索引，如要调用的函数名 f2。第 10 行用该索引获取到对应的符号项 f2 的首地址 relSymbol。第 13 行获得该符号所在节的首地址 sectionOfRel。第 14 行通过节首址+节内偏移获得该符号（**不是符号项**）的内存地址 symbolAddress。第 15～21 行根据该符号是否为函数进行不同计算，如果是函数且为相对寻址时，进行偏移计算，即偏移为 symbolAddress–(relocationAddress+4)（4 是相对 call 指令 E8 00 00

00 00 后的地址）。第 22 行用这个相对偏移或符号的绝对地址填写到 relocationAddress 指向的 4 字节，本例中就是 e8 指令后的 4 字节。

上面的做法存在缺陷，我们将整个 OBJ 文件复制到内存，然后重定位，相当于整个 OBJ 文件都是最后的执行代码。其实运行时，并非 OBJ 文件中所有部分都能用到，如符号表、重定位表、节头等就无用。真正有用的是节中的数据，甚至并非节中所有的数据都有用。比如，实现了 4 个函数，但从入口代码开始涉及的所有函数只有 2 个，那么就不应将所有代码节完整地链接到执行程序中，只需链接 2 个函数的代码即可。例如，本例中，只需链接 f1()和 f2()的代码。不需要的代码不链接入执行程序是链接器的一种优化。请大家参考习题 10 实现优化的链接器。

知其所以然的知行合一是有回报的。比如，当我们再次审视 3.3.2 节的提出的符号无法找到的错误时，就非常清晰了。编译产生 OBJ 文件时，只在机器码中留下占位符，代表之后要处理，并且把引起该占位符的符号（函数名或变量名）存到符号表。正常情况下是等待链接时，从本 OBJ 文件或其他目标文件中找到真正定义了该符号的文件，并将该符号使用的节合并到执行程序中。如果找不到该符号，只有报错。

3.3.2 节和 3.3.3 节完整地展示了猜测、实证和构建的做法和本质。希望读者能够感悟并改变一些编码学习的习惯和方法。

通过本节，我们展示了“构建”的力量。而链接知识本身也为将来解决可能遇到的编译链接错误提供了坚实的分析基础。下面利用这些知识来分析一个常见的错误。

初学者为了重用函数（如 test2()），常将函数实现放到头文件（如 test.h）中，然后被不同 CPP 文件引用。当两个 CPP 文件引用了 test.h 时，就会出现类似如下错误：

“testmain.obj :error LNK2005: "int _cdecl test2(int,int)" (?test2@@YAHHH@Z) already defined in test.obj”

其意思是函数 test2()在 testmain.obj 和 test.obj 中都被定义了。为什么？因为 test2()实现在 test.h 中，该头文件又被引用到 testmain.cpp 和 test.cpp 中，那么 test2()的实现相当于复制到这两个 CPP 文件中。编译后，它们对应的 OBJ 文件中均会生成 test2()的符号及对应实现。链接时，如果要调用 test2()函数，链接器就不知道用哪个 OBJ 文件中的实现（用 Preview 容易观察到两个 OBJ 文件的符号表中都有 test2 符号，且都有对应的代码 section）。怎样解决？将 test2 的实现移到一个 CPP 文件中即可，头文件中只有 test2 的声明，无实现代码，这样只有这个 CPP 文件生成的 OBJ 文件会包含 test2 的实现。

有人提出另一种做法：依然在头文件中实现 test2()，但在 test2()前加 static 关键字。编译确实能通过。可这与标准做法（头文件中无实现）的区别是什么？static 关键字使所修饰符号（如 test2）只在本 CPP 文件中有效。如果头文件中有 test2()实现，引用它的 CPP 文件必然会在 OBJ 文件中包含 test2()的机器码。当 CPP 文件调用 test2()时，链接器会因为 test2（）是 static，必然只调用本 OBJ 文件中包含的这段 test2()代码。如果多个 CPP 文件都引用 test.h 并调用 test2()函数，则会调用各自 OBJ 文件中的 test2()代码。这样，执行程序中将有多份相同的 test2()的机器码。通过调试“code\第 3 章\3-3\test_static”，反汇编跟踪，查看两个 CPP 文件中调用的 test2()入口地址，根据其不同，即可证明该结论。

3.3.4 分享它（3）：静态链接库

在 3.3.3 节的基础上，我们自己都可以设想到共享代码的方案了。将要复用函数所在的 OBJ 文

件全部复制到新项目中，让新项目链接时链接它们即可。如 3.3.1 节的硬盘数组，只需将包含其相关函数的 simplediskarray.obj 在链接时链入执行程序即可。但当 OBJ 文件太多时，管理和使用它们都比较麻烦。一个想法是将相关 OBJ 文件打包成一个文件，就像压缩程序打包一样。这种文件称为静态库，扩展名为 .lib。在创建“code\第 3 章\3-3\simplediskarray”项目时，选择静态库，则编译链接时将生成静态库文件，调试版就在“code\第 3 章\3-3\simplediskarray\Debug”目录下生成了 simplediskarray.lib。其链接使用的目标文件 simplediskarray.obj 也在同一目录下。用 PEview 查看 simplediskarray.lib，会发现 PEview 左边列表中存在名为.\Debug\simplediskarray.obj 的节点，查看其内容，发现它与 UltraEdit 打开的 simplediskarray.obj 的内容一模一样。这说明静态库文件确实将相关 OBJ 文件打包在一起了。

依据单元测试的习惯，我们在同一个解决方案中建立了测试工程，见“code\第 3 章\3-3\simplediskarray\testsimplediskarray”，将引用 simplediskarray.lib，否则其中涉及对硬盘数组的函数调用将在链接期找不到对应的代码。例如，testsimplediskarray.cpp 调用了 insertSimpleDiskArray() 函数，编译时生成的 testsimplediskarray.obj 中只会在符号表中添加 insertSimpleDiskArray 项，并标定为外部引用。链接时，因为链接器不知从哪里找到 insertSimpleDiskArray 对应的代码，所以无法成功。既然链接需要获取从哪个 LIB 文件中查找 insertSimpleDiskArray 的信息，那就告诉它。右键单击 testsimplediskarray 项目图标，选择“项目依赖项...”，在弹出的窗体中勾选 simplediskarray 项目。这样，在链接时，集成开发环境会将 simplediskarray.lib 作为 testsimplediskarray 项目链接时的链接库。

本例中，静态库项目有源代码，且与引用它的项目在一个解决方案中。如果只有 LIB 文件，无源代码，怎么办？从前面分析知道，只要链接时链接 simplediskarray.lib 库即可。因此需要找到在 VS 中设定该信息的地方。工程“code\第 3 章\3-3\PhoneNote_lib”将 3.2 节的相关代码修改为对硬盘数组函数的调用，并将静态库 simplediskarray.lib 复制到“code\第 3 章\3-3\PhoneNote_lib\lib”下，让链接时链接该库。这次在理解了原理后，我们完全可以在 VS 界面中猜测哪个属性与链接静态库 simplediskarray.lib 相关（**基础的力量**）。如果将所有要引用的 LIB 文件放到某个目录下，就需要设定该目录。当然，还应指明依赖的静态库的名字“simplediskarray.lib”，否则如果不需要的库恰巧提供了相同的函数名，会造成混乱。因此，基本可猜测需要设定库的路径和库名。

在项目属性的链接器相关项（**与链接相关，所以首先查找它**）下可以找到设定界面。通过猜测和实验（就是看能否编译成功），发现常规项中的“附加库目录”就是静态库的目录，输入“$(ProjectDir)..\lib”即可。其中，“$(ProjectDir)”代表项目的路径，即“..code\第 3 章\3-3\PhoneNote_lib\PhoneNote1”，“..”代表上一级目录，合在一起就是“..code\第 3 章\3-3\PhoneNote_lib\lib”。而在输入项中，“附加依赖项”就是输入静态库名的地方，输入“simplediskarray.lib”即可。要依赖多个库怎么办？不外乎这些库名之间用空格、逗号、分号之类隔开，只要实验一下即可。

> 大家对编译链接的选项往往都很迷惑，换了 IDE，如从 VC6 到 VS 就不知如何配置了，更不用说换到 Delphi 环境下。其实，只要知道了编译链接的基本原理，在图形界面的开发环境中，只需通过用基本原理去猜测、实验，就非常容易找到相关选项。对 VC 命令行的编译程序 cl 而言，就是将依赖的库（包含全名，跟在源文件后，上面的例子中依赖 detours.lib）：
>
> cl /LD C:\PhoneNote_good_prj\hooker\dllmain.cpp C:\PhoneNote_good_prj\hooker\hooker.cpp
> **C:\PhoneNote_good_prj\lib\deours.lib**

3.3.5 分享它（4）：动态链接库

除了静态库的方式，还有什么方式呢？我们知道，静态方式将使用到的代码全部链接到执行程序中，如果有 10 个执行程序都用了相同的函数，每个程序中都有一份相同的代码，很浪费硬盘和内存。我们学习了动态链接库，现在能非常清晰地理解“动态”和“链接”的含义了。执行程序需要的代码没有链接到程序中，而是运行时“动态”从其他 DLL 文件中加载到内存。

之前介绍动态链接库有两种使用方法：① 通过 LoadLibray 和 GetProcAddress 获得函数的入口地址即可使用（有些不方便）；② 用导入表机制，在源代码中可直接用函数名调用，程序员不用关心函数入口地址的获取问题。

我们先“猜测”构想第 2 种机制。假定现在要联系张三和李四（调用函数），但没有他们的电话号码（相当于入口地址）。我们可以在门口挂两个邮箱，贴上张三和李四的名字。然后约定投递员，将这两人的号码放入我们的两个邮箱中。我们从邮箱里取出号码就可访问了。其实，邮箱就相当于全局变量。我们设计一个表，将不同的函数名字（张三、李四）指向对应的全局变量。当程序被加载时，加载器（投递员）就会按表中的函数名去加载包含该函数的 DLL，并在加载后将该函数的地址设定给对应的全局变量。图 3.25 是我们假想机制的原理图。所有调用 fun1()的代码都编译成“call [fun1 对应的全局变量地址]”，等执行该指令时，全局变量中已经是函数的入口地址了。

在图 3.25 中，func1()所在的动态库加载前，call 指令调用的函数的入口地址存储在地址为 0x00048800 的变量中，该值是 0。加载 DLL 后，从表中可查找到程序要使用 func1()函数，并且存储该函数地址的变量为 0x00048800，加载器就将 func1()的入口地址 0x00768800 写入地址为 0x00048800 的变量。这样，当 call 指令执行时，就会调用入口地址为 0x00768800 的函数 func1()，见图 3.26。描述一个程序要调用某 DLL 哪些函数的表被称为导入表（import table）。

下面实证以上猜想。像 Sleep()这样的函数属于 kernel32.dll，随便写一段代码调用 Sleep()，然后反汇编。为了代码简洁，用 Release 版调试。对照 DM3-56、图 3.27 和图 3.26，两者完全吻合。

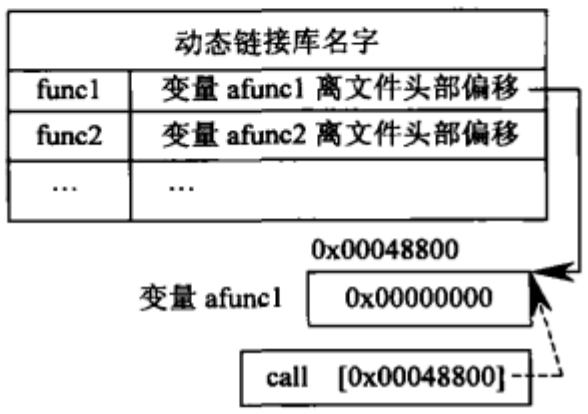


图 3.25 加载 func1 所在 DLL 前

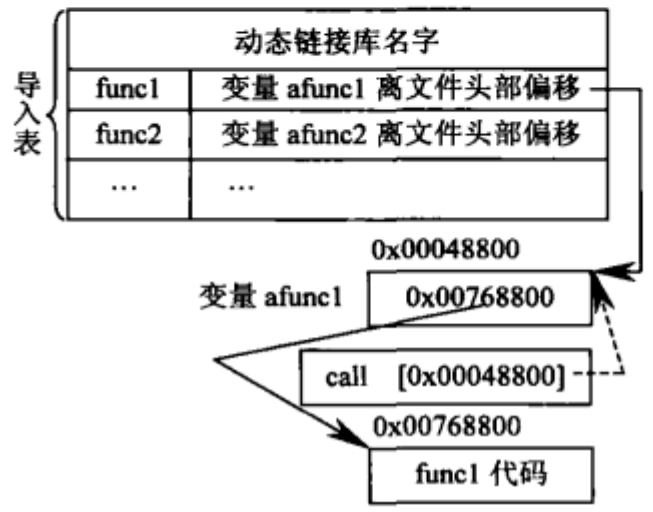


图 3.26 加载 func1 所在 DLL 后

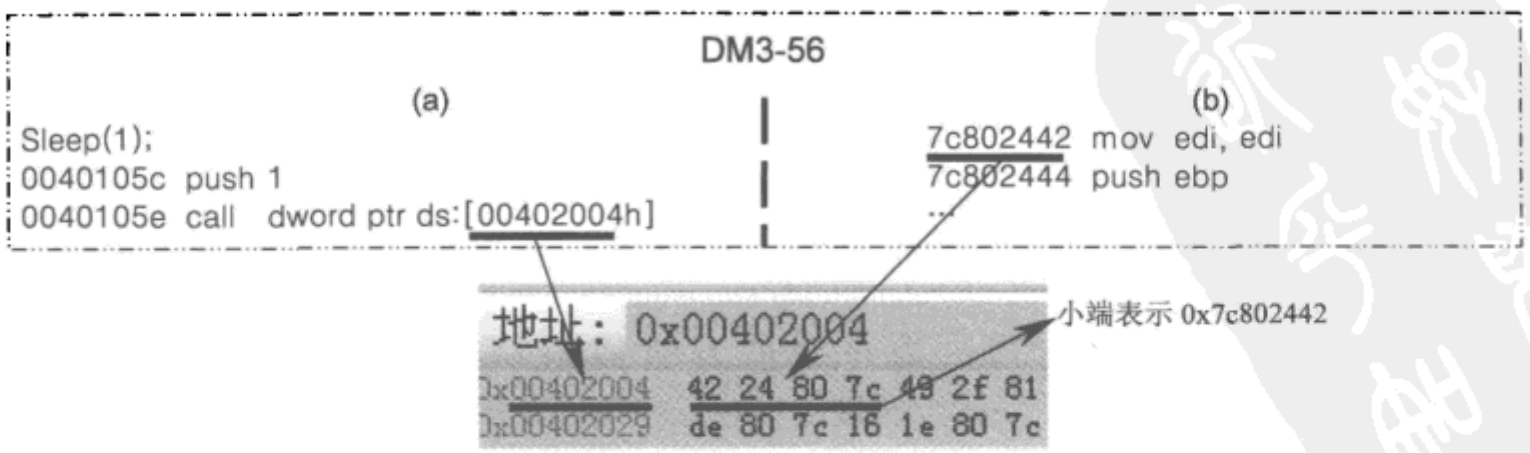


图 3.27 调用 Sleep()后的内存

现在需要思考如何通过链接生成相关调用代码。如果编译器在编译阶段看到 Sleep(1)这样的代码，自然应该翻译成相对 call 指令：call　offset，然后等链接器定位出符号 Sleep 再确定 offset 的值。可是，DM3-56(a)中的 call 指令却是“call　[地址]”形式。编译器看到什么特殊标识才做出这种翻译？右键单击 Sleep，选择弹出菜单的“转到定义”，可看到如下代码：

```
WINBASEAPI
VOID
WINAPI
Sleep(
    __in DWORD dwMilliseconds
);
```

WINAPI 就是_stdcall，只剩下 WINBASEAPI 和__in 不清楚意图，但__in 是修饰输入参数的，应该与我们的问题关系不大，那就继续用“转到定义”的方法查看 WINBASEAPI。发现，它等价于__declspec(dllimport)。可能是它导致了 call　[地址]的编译结果。我们自己声明 Sleep，不要它，编译的 call 指令就应该是 call　offset 这种了。注释掉“#include<windows.h>”，然后自己定义如下：

```
int __stdcall Sleep(int time)
```

结果出现链接错误，“无法解析的外部符号 "int __stdcall Sleep(int)" (?Sleep@@YGHH@Z)”。“?Sleep@@..”在前面做链接器的时候遇到过：函数名在 OBJ 文件中的符号并非名字本身，在其前、后加了莫名其妙的符号。既然找不到，就意味着调用 Sleep()的 OBJ 文件中引用了该符号。用 PEview 打开该 OBJ 文件，确实重定位项中指出的符号就是“(?Sleep@@YGHH@Z)”。

> 在编译生成 OBJ 文件时，C 编译器和 C++编译器对符号名字并非直接存入 OBJ 文件中，而是要做一定变化，两者的变化规则不一样，同一个符号最终形成的结果将不同。
>
> 选择项目“属性 | C/C++ | 高级 | 编译为 | 编译为 C 代码/TC”，即按 C 语言代码编译。用 PEview 查看 OBJ 文件，则 Sleep 的符号为_Sleep@4，确实与 C++的不同。如果 C++中引用 C 语言的函数，在形成 OBJ 文件时，就应该按 C 语言的风格形成符号名，否则引用处是 C++风格的符号名，实际实现该符号的 OBJ 文件中又是 C 语言风格表示的符号名，链接时必然找不到 C++风格的符号。这时用“extern "C"”修饰，如 extern "C" void _stdcall Sleep(int time)，那么调用 Sleep()的代码涉及的 Sleep 符号就按 C 语言风格形成。
>
> 因此，这两种风格的代码可混合编程。

在 Sleep()声明前加上“extern "C"”。运行结果如所料，见 DM3-57(a)。call 指令正是 call offset 方式(注意 e8 为操作码，偏移量为 0ah)。而跳转到一个 jmp [地址]指令，从调试可知，地址 00402000h 中存放的就是 Sleep 函数的入口地址。因为 Sleep()的声明没有特殊性(无__declspec(dllimport)修饰)，所以编译器自然按一般的函数调用来处理，生成了 call　offset 这样的指令。而 Sleep()的真正入口地址存放在相关导入表项中（地址 00402000h），因此只好用一个 jmp　[00402000h]做跳板，跳到 Sleep()入口。

DM3-57

(a)

```
Sleep(1);
00401003 6a 01          push 1
0040105e e8 0a 00 00 00 call  00401014
```

(b)

```
00401014  jmp  dword ptr ds:[00402000h]
...
```

我们来分析这两种方式如何导入 DLL 的函数：

① 用__declspec(dllimport)修饰函数声明的方式需要修改编译阶段的代码，遇到__declspec(dllimport)就直接生成“call　[地址 a]”形式的代码。然后需要一个目标文件提供“地址 a”相关的符号。通过 PEview 查看调用 Sleep()的 OBJ 文件，可见该符号名是__imp__Sleep@4。

② 不用__declspec(dllimport)的方式，不需修改编译阶段的代码，生成的 OBJ 文件与是否调用 DLL 代码一点没关系。直接翻译成“call offset”。call 调用的符号名是 _Sleep@4。然后需要一个目标文件提供符号_Sleep@4，以便链接，并且在链接时添加 DM3-57(b)类似的跳转指令。

总之，这两种方法都需要一个 OBJ 目标文件提供要导入的动态链接库的函数名以及库名信息以供链接使用。它在 VC 中就是被称为导入库的 LIB 文件。VC 的动态链接库项目编译后都会生成和 DLL 文件同名的 LIB 文件。我们建立工程“code\第 3 章\3-3\simplediskarray_dll”，将硬盘数组封装为一个 DLL 文件。编译后其 debug 目录下找到 diskarray.dll 和 diskarray.lib。然后仿照静态库的方式在解决方案中建立 testsimplediskarray 工程。并用导入库的方式使用 DLL 文件的函数，而不用 LoadLibrary 和 GetProcAddress。这时测试代码完全不用修改，与静态库版本的一模一样。既然要依赖静态导入库 diskarray.lib，自然需要按 3.3.4 节设定附加库目录和附加依赖项指向 diskarray.lib。

借助PEview，我们可以清晰地分析链接器如何用静态导入库链接生成文件的。为了生成DM3-57所示的代码。首先，导入库应该有一个目录查看到所有它能提供的符号（目录是任何 LIB 文件都必须提供的）。当查到该符号后，链接器就知道该符号属于哪个 DLL，以及对应 DLL 中的哪个函数，这样才能生成导入表并生成 DM3-57(b)所示的 jmp 代码。

用 PEview 可以大致猜出 LIB 文件的格式。在浏览了图 3.28 左边的树型浏览面板后，可以看到 Offset Table 中最后一行（00000060）标注的_updateSimpleDiskArray 项，其 Description 为 Offset to Header，说明可能是定位某个 Header（树型浏览面板中有很多以 IMAGE_ARCHIVE_MEMBER_HEADER 为前缀的项，应该就是 Header）的偏移量。_updateSimpleDiskArray 项的 Data 段的值为 00000DE0，于是在树型浏览面板中查看哪个 Header 的起始偏移是 DE0H。

如图 3.29(a)发现了一项 Header 是 IMAGE_ARCHIVE_MEMBER_HEADER 00000DE0，单击之，从右边列表的第一行第一列（该 Header 起始位置）可知，其文件偏移正好就是 DE0（选择用 RVA 方式查看偏移）。所以，图 3.28 中猜测的 Offset 不仅是偏移量，还是离文件头部的偏移。再单击图 3.29(a)中的 IMPORT_OBJECT_HEADER 和 IMPORT_OBJECT_HEADER Data，分别出现图 3.29(b)和图 3.29(c)。图 3.29(b)中的 Type 说明符号 updateSimpleDiskArray 是 IMPORT_OBJECT_CODE（应该是代码之意）。图 3.29(c)中的“_updateSimpleDiskArray.diskarray.dll”说明该符号关联的函数名字为 updateSimpleDiskArray，所属动态链接库为 diskarray.dll。

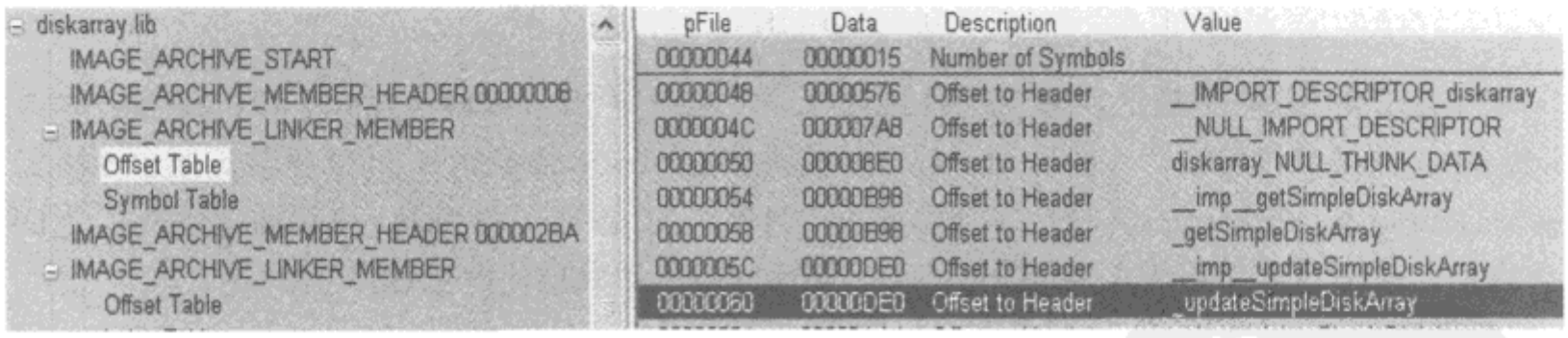


图 3.28 _updateSimpleDiskArray 项

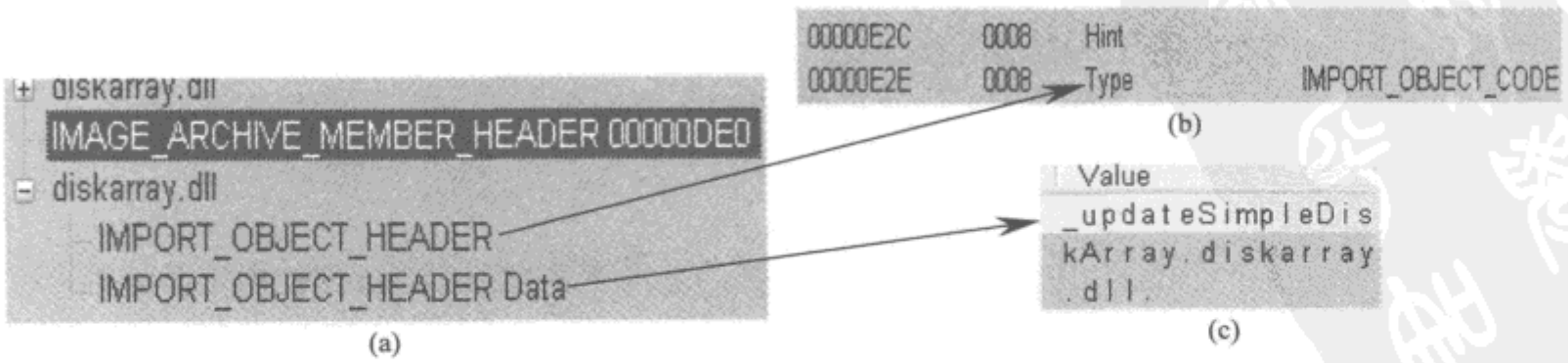


图 3.29 寻找 Header 的起始偏移

因此，我们可以得出链接动态库导入函数的大致算法如下：

① 从图 3.28 的 IMAGE_ARCHIVE_LINKER_MEMBER 表项（即符号的目录）中可获取所有的符号名及所在的 Header 的偏移量。遍历可找到需要查找的符号如 symbol2。

② 查看相关的 IMPORT_OBJECT_HEADER（见图 3.29(b）中的 Type 字段，如果其值是 IMPORT_OBJECT_CODE，则表明符号 symbol2 代表的是动态库的导出函数。因此，生成 DM3-57(b) 的 jmp 代码。这还不够，因为 jmp 代码是跳转到导入表所存地址（见图 3.26），所以链接器还要构造导入表。根据 IMPORT_OBJECT_HEADER Data（见图 3.29(c)）存储的“函数名.xx.dll”获得的库名 xx.dll 和函数名，即可构建导入表。

对 COFF、LIB 库及导入库格式感兴趣的读者，可阅读参考资料 1（网上容易下载），并用 PEview 辅助学习。

3.3.6 搭积木的艺术

DLL 除了可节约空间、共享功能，还有什么好处？DLL 是动态加载的，那么我们完全可以用一个全新的 DLL（但导出函数的名字和原型与旧 DLL 一样）覆盖老 DLL，就实现系统升级了。如习题 6 所示，3.3.1 节实现的硬盘数组效率不高，我们完全可以重新实现一套函数，但保持导出接口（导出函数名、原型）与旧 DLL 相同，就可不修改任何已经发布的代码，只是替换硬盘数组的 DLL 文件即可。只要将系统分解成一组合理的 DLL 文件，并抽象出好的函数接口，系统就能很好地模块化，像搭积木一样灵活可替换。

如何用 DLL 将电话簿程序分解？从功能块分解是一种直接又清晰的做法，见图 3.30。

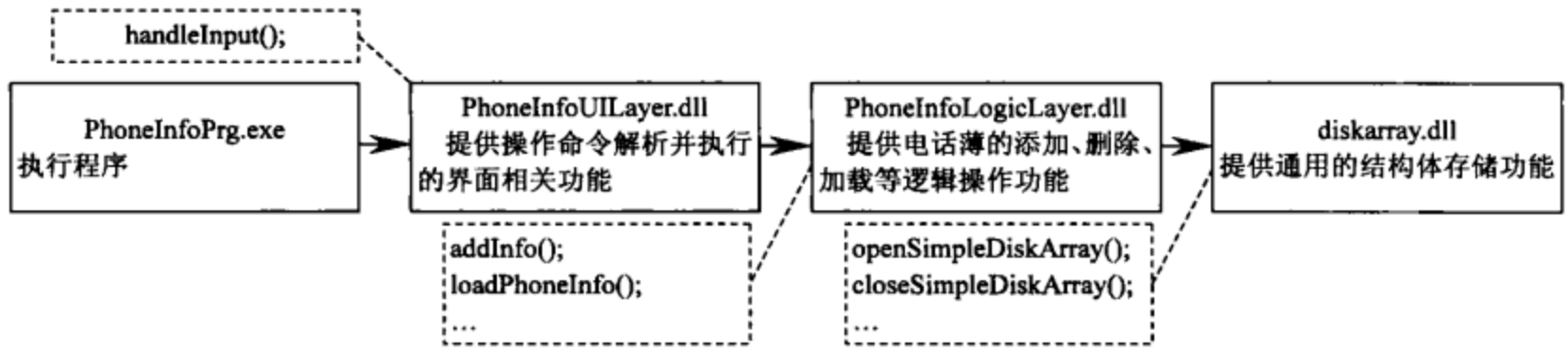


图 3.30　DLL 分解

程序被分解为 4 个模块，最上层是执行程序，它依赖 PhoneInfoUILayer.dll。该 dll 的函数 handleInput 从键盘接受指令，然后调用逻辑层模块 PhoneInfoLogicLayer.dll 的函数。而该逻辑层又依赖 diskarray.dll，diskarray.dll 封装了存储层及硬盘数组的逻辑。这样分解，可以随意替换 3 个 DLL 升级程序。而即使将来系统升级为图形界面，至少也能复用底层的两个 dll。

3.3.5 节提供了 diskarray.dll。现将 3.3.4 节中的 PhoneInfo.cpp 封装为 PhoneInfoLogicLayer.dll。其中的变量 gPhoneInfoPath 被主函数使用，如何通过动态链接库导出变量？先从原理上分析实现的可能性。利用可执行模块的导入表项，存储要引用的 DLL 的导出变量的地址，就可访问导出变量了。比如，gInt 为导出变量，那么“gInt = 2;”就要翻译成指令：

```
mov eax, [gInt 导入表项的地址]          //gInt 导入表项中存储 gInt 的地址
mov [eax], 1
```

而不是通常的一条指令“mov　[gInt], 1”（因为通常的全局变量在程序链接期能确定其地址，所以只需一条指令）。编译后，在头文件中将导出变量 gPhoneInfoPath 定义为 extern，那么使用该变量的 CPP 文件生成的 OBJ 文件中重定位项“gPhoneInfoPath”将标定为外部引用。链接时，从提供该变

量的动态链接库对应的静态导入库(.lib)中，查找到该符号即可。并因为该符号在导入库中被描述为动态库的导出变量（非代码），那么 linker 就最后会生成对应的导入表。

在解决方案中建立 dll 项目 PhoneInfoLogicLayer 生成 PhoneInfoLogicLayer.dll。将依赖的库 diskarray.lib 复制到工程的 lib 子目录下，并设定项目的依赖库和相关路径属性。尝试在头文件 PhoneInfo.h 中定义，见 DM3-58。

DM3-58

头文件 PhoneInfo.h

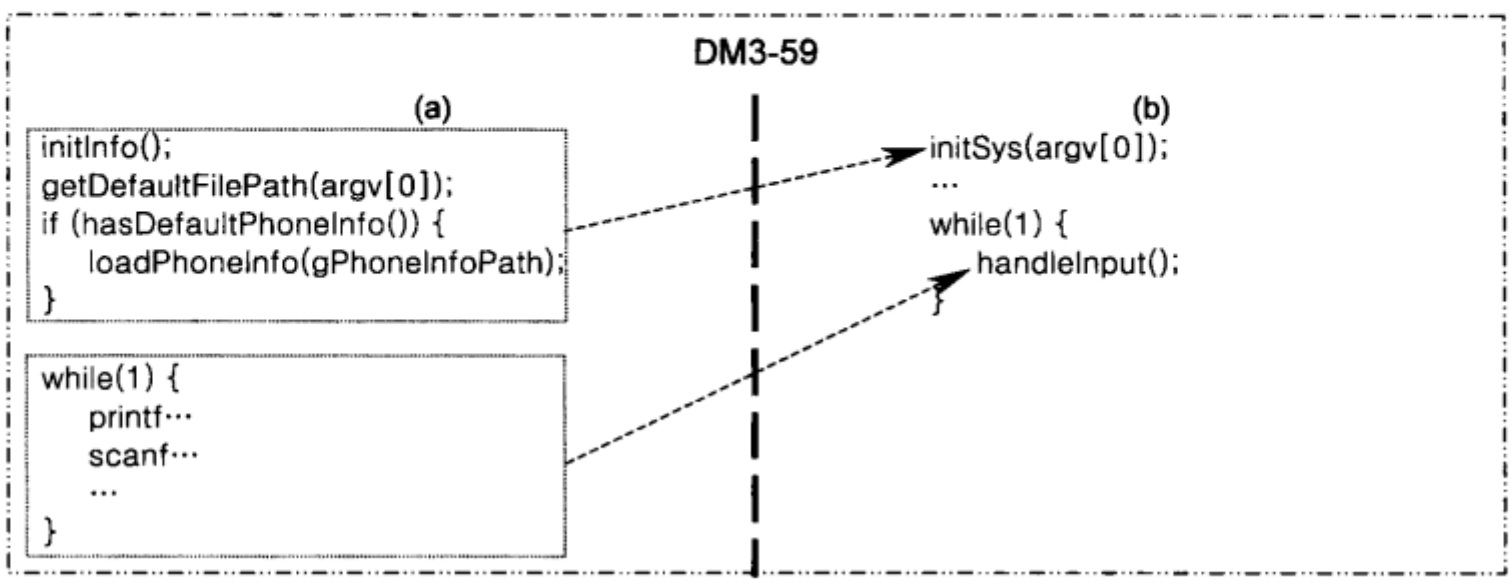
```
...
extern char __declspec(dllexport) gPhoneInfoPath[100];
...
```

编译通过。为了保险起见，用 PElord 查看编译成功的动态链接库，在 Directoies 的 Export Table 中查找到 gPhoneInfoPath。（**搞定！**）

Export Table 即 PE 格式中的导出表。如果 DLL 文件中没有这个表，那么 GetProcAddress(h, "Sleep")无法用字符串"Sleep"从 DLL 文件中找到对应函数的入口地址。GetProcAddress 就是遍历了 DLL 的导出表，查看是否存在指定函数名的表项，表项中存储了函数名字和函数入口相对偏移量。用该偏移量能最终换算出 DLL 加载到内存后函数的入口地址。

剩下的工作是在解决方案中建立的 DLL 项目生成 PhoneInfoUILayer.dll。我们将“code\第 3 章\3-3\PhoneNote_lib\PhoneNote1\PhoneNote1.cpp”中的 main()函数加以分解。从 DM3-59(a)中的代码转换为 DM3-59(b)的代码，(a)中第一个虚框中的代码转换为 PhoneInfoLogicLayer.dll 的导出函数 initSys，(a)中第 2 个虚框中的代码（即整个 while 循环体）转化为界面层 PhoneInfoUILayer.dll 的导出函数 handleInput。此时，右键单击项目属性，并选择“项目依赖项...”，指明依赖上面的 DLL 项目 PhoneInfoLogicLayer 即可（这样编译器才会自动链接本项目依赖的 PhoneInfoLogicLayer.lib）。

DM3-59



之后在解决方案中建立执行程序项目 PhoneInfoPrg，它的代码极为少，只有一个 main()函数，其内容就是 3.59(b)所示的 3 行代码。再次设定其依赖项目为前两个 DLL 项目。

编译 PhoneInfoUILayer.dll 时，报链接错误找不到符号“?gPhoneInfoPath@...”。有了前面的链接知识，知道应该去查找 PhoneInfoLogicLayer.dll 的导入库 PhoneInfoLogicLayer.lib。用 PEview 查看，包含的符号只有“__imp_?gPhoneInfoPath@@...”。由 3.3.5 节末分析导入库知道，对于导出函数 updateSimpleDiskAarray()，它导出的符号在图 3.28 右边最后两项：_imp_update…和_update…。仔细观察 PhoneInfoLogicLayer.lib，对于函数导出了两个符号，一个“__imp__”打头，另一个则没有；全局变量只导出了一个“__imp__”打头的符号，这样自然找不到“?gPhoneInfoPath..”。有无其他方式导出 DLL 的符号（函数或变量），从而在导出库的导出符号中对导出变量生成“?符号名..”的版本？有，则问题解决。

用 DEF 文件导出 DLL 的函数

在动态链接库项目图标上单击右键，选择“新建项”，在弹出的窗体中选择“模块定义文件.def”。生成后，在该文件中添加 EXPORTS 字段，即导出，然后将要导出的符号写在其后用分号分隔：

```
EXPORTS
  gPhoneInfoPath;
  addInfo;
  ...
```

编译后用 PEview 查看，发现这次导出了“__imp__?gPhoneInfoPath...”和“?gPhoneInfoPath...”两个符号。那就意味着问题解决了。果然没有发生链接错误，成功。

根据本节开始的分析，对于 DLL 导出的全局变量的赋值访问应该由两条 mov 指令组成。我们还是应该看看使用方的代码是否真的这样。在 PhoneInfoUILayer.dll 的相关代码处设置断点（全局变量 gPhoneInfoPath 是 PhoneInfoLogicLayer.dll 导出的，由 PhoneInfoUILayer.dll 使用，所以应该反汇编查看 PhoneInfoUILayer.dll 中使用该变量的代码），见 DM3-60。其中，第 6 行为 CopyFileA 传递了第 1 个参数，即 gPhoneInfoPath 的地址。这条指令竟然直接是 push 了 100110DCh 地址，它根本不可能在编译代码时就知道该地址，而应该是从导入表中获取的，由以下两条指令组成：

DM3-60

```
1  if (CopyFileA(gPhoneInfoPath, arg, FALSE) == 0)
2     mov  esi, esp
3     push 0
4     mov  eax, dword ptr [ebp-28h]
5     push eax
6     push 100110dch
7     call    ...
```

```
mov eax, [gPhoneInfoPath 的导入表项地址];
push eax
```

（**太诡异了。**）似乎 PhoneInfoUILayer.dll 在链接期事先为这个变量分配了一份内存，否则无法生成 DM3-60 第 6 行代码。这意味着它当中的该变量和 PhoneInfoLogicLayer.dll 中的该变量实际是两份内存空间（本该是一份内存），它们的地址肯定不同！

在 PhoneInfoLogicLayer.dll 的函数 getDefaultFilePath 中使用该变量的地方设置断点，反汇编获得 DM3-61。从第 4 行可知，正是压栈该变量的地址为 strcpy 传递第一个参数，该地址为 3a8148h，与 DM3-60 第 6 行显示的地址 100110dch 不同。（**确实为两块内存！**）从程序执行结果看，也出现了错误。使用 gPhoneInfoPath 的代码并没有按照导入表的机制生成代码。（**原因何在？**）

DM3-61

```
1  strcpy(gPhoneInfoPath, exePath);
2     mov  eax, dword ptr [ebp+8]
3     push eax
4     push 3a8148h
5     call   003a10dc
   ...
```

先分析使用 DLL 的工程，即 DM3-59 的头文件声明的全局变量 extern __declspec(dllexport) char gPhoneInfoPath[100]。在 3.3.5 节分析 Sleep()的例子时就发现：调用 DLL 的函数时，如果该函数在头文件声明为“__declspec(dllexport)”，其效果与没有用“__declspec(dllexport)”修饰的效果一样。没有该修饰，DM3-58 的声明其实就是 extern char gPhoneInfoPath[100]。在编译器看来，该符号就是一个与 DLL 无关的全局变量。自然，生成的指令也不可能从导入表获取变量地址。

剩下的问题是，应该怎样让编译器按导入表机制生成访问 gPhoneInfoPath 的代码。3.3.5 节分析 Sleep()的例子时，用__declspec(dllimport)修饰一个函数的声明，生成的代码就是“call [函数导入表项地址]”。那么，是否意味着用__declspec(dllimport)修饰全局变量的声明，编译器生成的变量访问代码就会从变量的导入表项获取变量地址？如果这样，从 DEF 导出全局变量反而是危险的，因

为这时的导出库.lib 会生成“? + 变量名 + ...”的形式，如“?gPhoneInfoPath@@...”。如果使用者使用导出变量 gPhoneInfoPath，却没有在其声明时加上“__declspec(dllimport)”，即 extern char gPhoneInfoPath[100]，就会产生不使用导入表的错误代码。而该代码引用的变量符号是以问号打头的符号“?gPhoneInfoPath@@...”，该符号却能在 DEF 文件方式导出的导入库中找到，自然通过链接。如果用“__declspec(dllexport)”方式导出，因为只会导出“__imp__”打头的符号，反而不会发生该错误。如果不用 DEF 文件导出函数或变量，就需要在头文件 PhoneInfo.h 中定义：

```
extern char __declspec(dllexport) gPhoneInfoPath[100];
```

而其他项目中通过引用头文件 PhoneInfo.h 来使用 gPhoneInfoPath 时，却需采用如下形式：

```
extern char __declspec(dllimport) gPhoneInfoPath[100];
```

同一个头文件中对同一个变量的声明有两种方式，怎么办？用条件编译，见 DM3-62。其中，每个 DLL 项目的预处理宏中都自动定义了一个“项目名称+_EXPORTS”的宏。在 PhoneInfoLogicLayer.dll 项目中是 PHONEINFOLOGICLAYER_EXPORTS，代表为动态链接库导出而使用。如上条件编译导致在动态链接库的工程中编译时，因为 PHONEINFOLOGICLAYER_EXPORTS 存在，所以第 2 个含 dllexport 的声明就有效；而对其他项目使用时，因为没有定义该宏，自然是含 dllimport 的第一个声明有效。

DM3-62

```
#ifndef PHONEINFOLOGICLAYER_EXPORTS
extern char __declspec(dllimport) gPhoneInfoPath[100];
#else
extern char __declspec(dllexport) gPhoneInfoPath[100];
#endif
```

编译运行，我们看到DM3-60涉及的代码有新的编译结果，见DM3-63。第6～7行相当于DM3-60的第 6 行，此时代码确实是从导入表获取导出变量的地址。当然，整个程序运行结果也正确。工程见 code\第 3 章\3-3\PhoneInfo_dll。不过，建议大家不要导出变量，它违反了封装的思想，不利于隐藏实现，可能导致无法替换新的 DLL 实现。

简单总结微软编译器处理动态库静态导入的细节（在 C++ name mangling 规则下）：

DM3-63

```
1  if (CopyFileA(gPhoneInfoPath, arg, FALSE) == 0)
2      mov  esi, esp
3      push 0
4      mov  eax, dword ptr [ebp-28h]
5      push eax
6      mov  ecx, dword ptr ds:[1001833ch]
7      push ecx
8      call  dword ptr ds:[100181ech]
```

① 用 DEF 导出函数名和变量名，每个函数名或变量名在静态导入库（.lib）中均将生成两个符号。一个以“__imp__?”为前缀，一个以“?”为前缀。

② 用__declspec(dllexport)导出函数名，结果如规则①。而导出变量，则只导出一个符号，在 C++ 规则下，以“?”为前缀。

③ 如果以__declspec(dllimport)定义要引用的函数，则编译器知道被调用函数是 dll 的，所以调用方代码会生成“call [导入表项地址]”风格的代码。如果不用该符号定义，则编译器认为被调函数非动态库提供，因此生成如 DM3-57 所示的 call 和 jmp 两条指令。其中，call 指令是编译期生成到 OBJ 文件中，jmp 指令在链接期链入执行模块中。

④ 如果以__declspec(dllimport)定义要引用的变量，则编译器知道变量是 dll 提供，生成的代码将从导入表获取变量地址。如果不用该符号定义，因为没有从导入表取出地址，虽然编译通过，运行则将发生错误。

⑤ 用__declspec(dllimport)符号定义的函数或变量名，在 OBJ 文件中，其对应的引用符号以“__imp__”为前缀。如果不用“__declspec(dllimport)”定义，OBJ 文件中对应的引用符号以“?”为前缀。

⑥ 在定义 DLL 导出符号的头文件中，应该用条件编译定义，见 DM3-62。

参考资料

1. Microsoft Portable Executable and Common Object File Format Specification.

http://msdn.microsoft.com/en-us/windows/hardware/gg463119

微软的 PE 格式和 COFF 格式及静态库 lib 涉及的格式。

习 题 3

1．请对比 0 结尾和直接指出字符串长度这两种方式表示字符串长度各自优劣。

2．用反汇编分析 Delphi 的字符串结构。

3．分析 DM3-25(a)和 DM3-25(d)的代码，能用分割函数原则做出怎样的改变？

4．如果 3.3.1 节中的文件格式中加入元素的大小，请修改相关函数接口和实现。

5．如果仿效 fopen 和 fwrite 的缓冲机制，给出 3.3.1 节硬盘数组的改进方案。是否在 fopen 中就分配写缓冲区？从空间效率上分析，何时分配更合理？并实现。

6．3.3.1 节实现的硬盘数组除了习题 4 和 5 的改进外，还有什么严重的性能问题？请给出改进方案。

分析电话簿程序采用 DLL 方式调用硬盘数组功能时（如 3.3.6 节的动态链接库共享方案），本题实现方案与 3.3.1 节实现方案能否做到在不修改电话簿程序情况下，只替换硬盘数组 DLL 即可。硬盘数组在这一要求下提供怎样的函数接口更合理？并实现。

7．能否参考测试 LoadPhoneInfo 中的打印逻辑采用的函数指针方法，将 3.2.5 节涉及的工程“code\第 3 章\3-2\PhoneNoteDynamicArray”中的 main()函数的逻辑（注意不是代码，而是逻辑）也测试？

提示：可以将 while 循环中的代码变成一个函数 handleInput()。

8．局部变量在 OBJ 目标文件中会产生重定位项吗？“无法找到符号”这样的链接错误会发生在局部变量上吗？

9．验证 DM3-46 对应的目标文件中，图 3.13 的第一个重定位项确实是 DM3-46 中第 1 行代码产生的。

10．参考 3.3.3 节最后部分的描述，修改 3.3.3 节链接代码的实现。当指定入口函数后（如其中就是函数 f1()），只将与该函数调用涉及的所有函数和变量链接到一起，并比较两者链接出的大小（3.3.3 节中的大小就是 OBJ 文件的大小），且所有相关的全局变量放在一起，所有代码放在一起。

提示：从入口函数开始形成符号引用图，找到一个函数所引用的符号及符号（变量和代码）的长度信息是关键（有的信息也许需要计算而非直接能获取）。感兴趣的读者可参见 1.8.2 节的加载重定位设计（图 1.72(b)所示文件格式）以及实现“code\第 1 章\myprogram_load”，将链接结果生成文件。也就是说，链接除了找出了相关代码和变量并合并生成了数据段与代码段，还生成了执行时需要的重定位段、程序入口地址、加载基址。

11．如果被链接的符号不在 OBJ 文件中定义，而只是被引用，会在引用它的 OBJ 文件中形成一个符号项。如何区分符号是本模块还是外部模块定义的？用 PEview 观察。

提示：查看调用 printf()之类库函数形成的符号项，或者调用其他 CPP 中实现的函数形成的符号项，即结构体 COFF_Symbol 的相关字段。

12．基于 11 题分析的结果，扩展 3.3.3 节的链接代码，使其可链接多个 OBJ 文件。并进一步结

合 10 题的结果实现。

13．用习题 6 实现的新硬盘数组的 DLL 升级 3.3.6 节的程序。

14．在 Visual C++中调试运行如下代码，查看 tem.txt 是否与执行程序在一个目录。

```
FILE * f = fopen("tem.txt", "w");
fclose(f);
```

如果不在，分析其原因，并找出解决方案。

15．用 PEview 分析“code\第 3 章\3-3\test_static”生成的两个 OBJ 文件，并查看证实其中均包含 test2 符号及其实现；验证 test2 的符号项的 Storage Class 类型是 IMAGE_SYSTEM_CLASS_STATIC（注意与非 static 函数的不同）。

16．根据 static 的含义（符号的实现将在 OBJ 文件内部），在工程“code\第 3 章\3-3\test_static”的 test.h 中定义 static int g 的“全局变量”，从 testmain.cpp 和 test.cpp 中访问 g。这个 g 是同一块内存吗？从链接的角度加以解释。

17．C++模板函数的实现定义在头文件中，当两个不同的 CPP 文件通过引用该头文件使用模板函数时（在工程“code\第 3 章\3-3\test_template”中，两个 CPP 文件均调用了模板函数 add(1, 2)）。两个 CPP 文件中均有该模板函数（add()）的实现吗？如果只有一个存在，会是哪个（两者均是同等的）？如果两个均有，链接为什么不报“符号已定义的错”？给出原因，并证明。例如，修改 OBJ 文件，将本来重复定义的两个函数符号通过某种修改，链接时不报错，单独链接用 link.exe。

提示，用 PEview 分析，并用参考资料 1 对照分析。

18．仿照习题 17，在 C++中，对比在类定义的头文件中直接实现成员函数与在 CPP 文件中实现成员函数，对生成的 OBJ 文件有何区别？

19．本书的重定位算法只处理了.text 节中需重定位的部分，参考第 1 章的习题 10，如有其他需要定位的节，请修改该算法。

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

PDG

第4章

让我们创造面向对象语言吧

需准备的知识

- 整章需要第1章的反汇编分析支持。
- 4.1节和4.6节需要1.3.3节、1.3.4节和1.4.2节的知识。
- 4.4.2节需要3.1.3节的文件操作相关知识。
- 4.7节需要1.3节的知识。

为你提供的能力

- 4.1节对象模型是第5章设计的关键。
- 4.7节虚函数机制是第6章的支撑。
- 4.1节和4.7节是第7章自定位代码的支撑。
- 有了对象模型，将进一步具备分析面向对象程序的基础。
- 通过反汇编，进一步深入理解了C++的各种机制，能应对各种诡异的设计错误。
- 全方面理解构造分配资源，析构释放资源。
- 充分理解“猜测—实证—构建”方法。

在第 3 章的学习中，我们已经稍稍领略了软件开发中重用和维护的力量与甘苦。本章在与重用和维护的角力中，将以解决问题的方式“创造”出面向对象语言，并用反汇编利器，游刃于那些关于“面向对象”神秘背后的真相。

4.1 “封装”数据函数合一，陈仓暗度 this 传递

4.1.1 那些讨厌的事

通过第 3 章，我们已经比较熟悉文件操作了，梳理相关代码，会发现存在以下容易出错的地方，见 DM4-1。

DM4-1

(a)

```
FILE *fp;
fread(buf, 20, 1, fp);          //错误，未打开文件
```

(b)

```
FILE * f1; FILE * f2;
f1 = fopen("...","r");
fread(buf, 20, 1, f2);   //错误，打开文件 f1，读文件 f2…
```

DM4-1(a)，在没有打开文件的情况下，读文件。DM4-1(b)，打开的文件关联指针 f1，却“张冠李戴”地用未初始化的文件指针 f2 来读。

再看一个关于第 3 章中硬盘数组函数的设计：

```
int load_item(const void *buf, int index, unsigned size, FILE *fp)
```

它有什么问题？① 每次调用该函数时，必须传入元素大小 size 和 fp，容易传递错误。比如，总得记住某个结构体有多大，就有 load_item(..., sizeof(struct Person), ...)这样的调用代码。(使用起来不爽。) ② 无法透明升级。假如想给这个函数增加一个读缓冲区，以提高性能，该怎么办（这样只要读取的数据在缓冲区中，每次读时可不读硬盘而读内存）？如果按照上面 load_item 的设计，只能再增加一个参数 char * readBuf，传入读缓冲地址：

```
int load_item(const void *buf, int index, unsigned size, FILE *fp, char * readBuf)
```

如此，不仅已有的调用 load_item 的代码必须修改，无法透明升级，而且完全不合理。因为如何分配读缓冲，如何读取都是硬盘数组函数自己的事，通过参数暴露出来，会让调用者不知所措。

面对这些讨厌的东西，先来看看现实生活中的一个例子能否启发我们。如果甲让乙付钱，乙肯定会掏“自己的”钱包，而不会掏“别人的”钱包。看看上面的两个错误，调用 fread 就相当于甲要求乙付钱，还指定了乙用哪个钱包付钱（fp 相当于钱包），因此导致甲错误地指定了用丙的钱包付钱。而平常生活中，乙知道“自己的”钱包在哪里，不用别人告诉，所以不会做出错误动作去掏别人的钱包，即乙的行为——“付钱”和行为操作的东西——“钱包”是一体的。如果能模仿人付钱的方式，那么在 DM4-1 的代码中就不需给“行为”fread 传递文件指针，自然就不会出现张冠李戴的错误。

再看现实中的另外一个例子，就更清楚“行为和其操作的东西是一体”这句话的含义。芯片工作时，通过引脚和外部沟通，外部永远不用告诉芯片电路应该操作哪个器件，因此芯片也永远不会操作到另一个芯片的内部器件。如将 DM4-1 中的 fread 看成是芯片的功能，将传递的参数 fp 变成芯片内部的器件，那么张冠李戴的错误就不会发生。

这两个例子背后的抽象就是“行为和其要操作的东西是一体”。这符合自然界本身的客观规律。

对芯片而言，行为就是其电路，能完成功能；而被操作的东西就是其存储信息的元件，是电路操控的对象。类比到程序，芯片的电路就是程序中的函数（行为），它们都是完成一定任务而存在；存储元件就是程序中的变量（数据）。芯片的强大在于通过封装将行为和数据合一。这种合一形成了一个“能动体”，它有“自己的”行为和数据，自己的行为只操控自己的数据。

细心的读者或许认为：函数本身就是一个芯片，代码就是电路，局部变量就是存储元件，函数的代码只会操作自己的局部变量，不会操作其他函数。注意，局部变量的生存期只在函数被调用执行期间，而芯片的存储元件能“长期”保存内部状态，有点全局变量的效果。全局变量却不是每个芯片一份，而是全局只有一份。所以，芯片的工作方式本质上有别函数的执行，它是“能动体”。

有了能动体后，文件操作就会变成 DM4-2 中的样子。fileOperation 是一个能动体，只需在其 open 方法中告知要打开文件的名字，之后其读取（readLn）、获取文件名（getFileName）和关闭文件（close）等行为，它们均自己知道要操作哪个文件，而不需像 fread 那样必须外部传递文件指针 fp。DM4-1 的问题就解决了。关于 load_item 增加读缓冲的问题，我们在 4.1.2 节中将看到答案。

4.1.2 像芯片一样工作（1）：数据合一

能否找到一种形式，将程序的行为与数据合一，即函数和变量合一呢？芯片用封装外壳做到了合一，在 C 语言中有这样的“外壳”存在吗？要将函数和变量合一，先要将相关的变量放在一起，这自然让我们联想到结构体。下面就以构建一个文本文件读写的能动体为目标来开始我们的探索。

先看看熟悉的过程式方法编写文本文件读取并获取文件行数的代码，见 DM4-3，以便后面对比。readLn 参考 DM3-16。既然要将变量整合在一起，就要将相关的信息放在一起，见 DM4-4，结构体中包含了文件指针、文件名等相关信息。用结构体编程的代码见 DM4-5。

DM4-2

```
fileOperation.open(fileName);
fileOperation.readLn(ln) ;
   ….
fileOperation.getFileName();
fileOperation.getCountLine();
fileOperation.close();
```

DM4-3

```
FILE * f;
char buf[100];
int lnCount;
f = fopen("C:\\temp.txt", "r");
if (f == NULL)
    return;
lnCount = 0;
while (!feof(f)) {
    readLn(fp, buf);
    lnCount++;
}
fclose(f);
```

DM4-4

```
struct TTextFileInfo {
    FILE * F;              //文件指针
    char FileName[256];
    int CurrentLine;       //当前读到的行数
    int TotalLine;         //总行数
    char* Mode;            //文件打开模式
};
    typedef TTextFileInfo * PTextFileInfo ;
```

对比 DM4-3，我们发现 DM4-5 的代码反而变复杂了。仔细分析有以下可重用的函数：

① 打开文件：BOOLEAN OpenTextFile(PTextFileInfo Info)，返回 0 代表失败，1 代表成功。

② 读取一行：BOOLEAN ReadLine(PTextFileInfo FileInfo, char * ln)，0 代表失败，1 代表成功。

③ 写一行：void WriteLine(PTextFileInfo FileInfo, char * ln)。

④ 关闭文件：void CloseTextFile(PTextFileInfo FileInfo)。

用这组函数修改代码 DM4-5，得到 DM4-6，相对于 DM4-3，显得比较精简。工程见“code\第 4 章\DataEncapsulateOnly”。

DM4-5

```
char Ln[100];
TTextFileInfo Info;
strcpy(Info.FileName, "C:\\temp.txt");
Info.Mode = "r";
Info.F = fopen(info.FileName, info.Mode);
Info.CountLine = 0;
while (!feof(Info.F)) {
    readLn(Info.F, Ln);
    Info.CurrentLine = Info.CurrentLine + 1;
}
fclose (Info.F);
```

注意，这 4 个函数均将结构体指针 PTextFileInfo 作为参数传入。这些函数可能要写结构体的成员变量，如 OpenTextFile 用语句“info->F = fopen...”给 info 的成员变量 F 赋值，要完成赋值逻辑必须将被写内存地址传入，所以为了操作自己的数据，必须将结构体的地址传入。

DM4-6

```
char Ln[100];
TTextFileInfo Info;
strcpy(info.FileName,
"C:\\temp.txt");
Info.Mode = "r";   //this is a constant
OpenTextFile(&Info);
while (ReadLine(&Info, Ln));
CloseFile(&Info);
```

DM4-7(a)

```
struct DiskArray{
    FILE * fp; //文件指针
    int elementSize;//元素的大小
}
void  load_Item(struct  DiskArray  *
pArray, void * pValue);
```

DM4-7(b)

```
struct DiskArray{
    FILE * fp; //文件指针
    int elementSize;//元素的大小
    void * readBuf;//读缓冲
}
void  load_Item(struct  DiskArray  *
pArray, void * pValue);
```

回头审视 4.1.1 节中关于函数 load_item 增加读缓冲区的问题。根据数据整合的原则，我们将文件指针、元素大小等集中到一个结构体 DiskArray 中，见 DM4-7(a)。load_item 将结构体指针传入即可，该函数从结构体上取得文件指针 fp 和元素大小用于读硬盘。需要增加读缓冲时，只需在结构体中增加成员变量 readBuf，并不用修改 load_item 的函数原型，函数会从成员变量 readBuf 指向的缓冲中读取相关元素。这样所有调用 load_item 的代码都不用修改，load_item 透明升级了！究其原因就是数据整合，外部调用者不用关心结构体内部的细节，这些细节是相关函数关心的东西，我们已经初见整合的力量，但现在仅仅是数据的整合，我们需要函数和数据的合一。

4.1.3 像芯片一样工作（2）：行为与数据合一

怎样将数据与函数合一形成能动体？基本的想法是通过结构体调用到相关函数。参考 DM4-2 中对于能动体的假想来修改 DM4-6，使其函数与数据合一。其代码将有如下变换，也就是通过结构体 info 能够调用函数 OpenTextFile。

```
OpenTextFile(&info);     →     info.OpenTextFile(&info);
```

问题就转化为能否通过结构体的成员变量调用函数？先想清楚如何才能调用一个函数。从第 1 章知道，本质上只要知道函数的入口地址就可跳转过去。1.3.10 节的函数指针就是保存函数入口地址的，通过它就能调用函数。而函数指针是变量，所以可放入结构体。这样，只要在结构体中定义函数指针 OpenTextFile，即可实现 info.OpenTextFile(&info)。

首先，针对 4.1.2 节定义的 4 个函数分别定义 4 个函数指针类型，如下：

```
//关闭文件的函数指针类型
typedef void (*TCloseTextFile)(PTextFileInfo self);
//打开文件的函数指针类型
typedef BOOLEAN (*TOpenTextFile)(PTextFileInfo Self);
//从文件读取一行的函数指针类型
typedef BOOLEAN (*TReadLine)(PTextFileInfo Self, char * Ln);
//向文件写一行的函数指针类型
typedef void TWriteLine(PTextFileInfo Self, char * Ln);
```

然后修改 DM4-4 中结构体 TTextFileInfo 的定义，见 DM4-8。其最后 4 个新增加的粗体显示的成员变量即代表 4 种操作的函数指针。

DM4-8

```
struct TTextFileInfo{
    FILE * F;          //文件指针
    char FileName[256];
    int CurrentLine;   //当前读到的行数
    int TotalLine;     //总行数
    char* Mode;        //文件打开模式
    TOpenTextFile Open;
    TCloseTextFile Close;
    TReadLine ReadLn;
    TWriteLine WriteLn;
};
typedef TTextFileInfo * PTextFileInfo;
```

用新结构体重写 DM4-6，得到 DM4-9。这时新代码已经从形式上做到了函数和数据合一，即 DM4-9

中第 4～6 行中粗体标记的“info.函数名(...)”的形式。工程见“code\第 4 章\DataEncapsulateFunc”。

注意，DM4-9 第 3 行调用了函数 InitTextFileInfo2，它通过调用 InitTextFileInfo 对结构体 info 的 4 个函数指针变量初始化，使其指向 4.1.2 小节实现的 4 个函数，见 DM4-10。否则 info 的这 4 个成员变量将指向随机的地址，势必导致 DM4-9 的 4~6 行执行的函数调用失败。

DM4-9

```
1 char Ln[100];
2 TTextFileInfo   Info;
3 InitTextFileInfo2(&Info, "C:\\temp.txt", "r");
4 Info.Open (&Info);
5 while Info.ReadLn(&Info, Ln);
6 Info.Close(&Info);
```

DM4-10

```
void InitTextFileInfo(PTextFileInfo pInfo) {
    pInfo->Open = OpenTextFile;
    pInfo->Close = CloseTextFile;
    pInfo->ReadLn = ReadLine;
    pInfo->WriteLn = WriteLine;
}
void InitTextFileInfo2(PTextFileInfo pInfo,char * FileName,char * Mode) {
    InitTextFileInfo(pInfo);
    strcpy(pInfo->FileName, FileName);
    pInfo->Mode = Mode;
}
```

为什么说是“形式上”数据和函数合一了？因为芯片似的电路与存储元件合一，其本质是电路知道自己的元件在哪里，直接操作之，而不需芯片外部告诉它要操作的元件在哪里，这样芯片“自己操作自己的数据”才不会发生张冠李戴的错误。可是 DM4-9 的第 4～6 行调用的 info.open(&info)、info.ReadLn(&info,...)等都需要让外部调用者来传递&info，这意味着 open()、ReadLn()等函数并非知道自己要操作的数据在哪里，而是外部调用者传递。如果调用者传递错误，如 info.open(&info1,...)，那么依然要发生张冠李戴的错误。数据和函数合一的能动体本质需要“自己操作自己的数据”。

4.1.4 不想让你传递“自己”

首先我们约定几个名词，这样称呼比较方便。

① 将数据和函数指针放在一起的结构体称为“能动体”。

② 能动体上函数指针指向的函数称为“能动体的函数”。

③ 能动体上定义的变量称为“能动体的变量”。

④ 能动体的函数第一个参数总是传入“能动体自己的地址”，如 info.open(&info)，因此我们称这个在能动体函数中传入的结构体地址为“self 指针”。

⑤ 能动体的函数操作能动体自己的变量称为“自己操作自己的数据”。

现在面临一个两难的境地，为了自己操作自己的数据，能动体的函数必须传递 self 指针，否则函数无法获得能动体的首部地址，自然无法根据偏移量访问到其相关的变量。如果通过参数传递，则调用者可能传递错误。这一张冠李戴的错误就违反了“自己操作自己的数据”的观念。怎么办？

我们不需要“info.open(&info)”这样的形式，而需要“info.open()”这样的形式。后一种方式的关键和优越性在于，它保证了 open()只会操作自己的数据，而不可能操作到别人的数据。我们不妨先从形式上满足不直接从 open()函数传递 self 指针，但暗度陈仓地从其他地方传递 self 指针就可以了。由 1.3.9 节知道，我们可用不同方式传递参数，其中一种在 Visual C++中叫_fastcall，它采用寄存器传递。例如，ecx 在 32 位机上是 4 字节，长度正好可以放下 4 字节的地址——self 指针。于是有了基本的算法思路，见 DM4-11。调用方代码在调用 open 函数之前将 self 指针，即&info 传递给 ecx。然后调用 open()函数，这时 open()函数并不从参数列表中传递 self 指针。被调用函数 open()一进入函数，就从 ecx 获取到 self 指针并赋值给 p，之后就可用 p 访问结构体了。为了方便，DM4-11(a)

是伪代码。

DM4-11

(a) 调用方代码

```
1    _asm  mov  ecx, &info
2    info.open();
```

(b) 被调用方代码

```
void open() {
  struct TTextFileInfo * p;
  _asm mov p, ecx;
  ...
}
```

只要按照这种思路改造所有的能动体的函数，我们就可以将 self 指针从能动体函数的参数表中去除掉。从形式上达到了不传递 self 指针的效果（见 DM4-11(a)第 2 行的 open()函数调用）。在实际工程中，我们采用 edx 传递的 self 指针。（**为什么不能用 ecx？见习题** 1。）DM4-12 展示了对 Close 函数的实际调用代码。DM4-13 对比了被调用的 CloseTextFile 函数的 4.1.3 节版本和当前版本。从对比可见，能动体的函数由旧版本演化为新版本非常简单：只需将老版本中的 self 指针参数剪切，复制到函数体中，作为局部变量，并在其后添加一行汇编，从 edx 获取 self 指针，赋给局部变量 pInfo 即可。工程见“code\第 4 章\DataEncapsulateFuncPassSelf”。

DM4-12

```
1    TTextFileInfo info;
2    InitTextFileInfo2(&info,...);
3    void * self = &info;
     ....
     //传递 self 指针
4    _asm   mov edx, self;
5    info.Close();
```

DM4-13

(a) 旧版本

```
void CloseTextFile(PTextFileInfo pInfo) {
    fclose(pInfo->F);
}
```

DM4-13

(b) 新版本

```
void CloseTextFile() {
1    PTextFileInfo pInfo;
     //获取 self 指针
2    _asm mov pInfo, edx;
     //以下为与旧版本一模一样的代码
3    fclose(pInfo->F);
}
```

如果只看 DM4-12 第 5 行的 info.close()，那就是完美效果。可是再往上看一行，发现我们必须手动用汇编传递 self 指针给 edx。这比 DM4-13(a)所示的直接调用版本还要麻烦和易错。而且在 DM4-13(b)中还需手动加入汇编语句，从 edx 获取 self 指针，更麻烦。（**似乎我们越做越痛苦了，但黑暗的后面往往是黎明。**）

4.1.5 创造吧，新的语言

如果让编译器帮我们自动悄悄传递 self 指针，那就好了。因为它是不会犯错的，自然保证了“自己操作自己数据”。

如 DM4-12 第 5 行所示，所有能动体函数调用前都要加入一条 mov 指令，完成 self 指针的传递。而 DM4-13(b)所示每个能动体函数一开始都要加入一条 mov 指令，将 self 指针获取。这两个动作是多么机械地重复啊。（**机械就好，程序（编译器）就适合做机械的工作。**）

编译器能够明白我们的意图，并自动加入汇编传递 self 指针吗？现在我们需要做一个角色扮演，将自己想象成编译器。我们先看见了 DM4-12 的第 5 行 info.Close()，info 是一个 TTextFileInfo 类型。根据 DM4-8，TTextFileInfo 的类型是 struct。结构体 info 的成员变量 Close 是一个函数指针，那么 info.Close()就应该解释为“用 call 指令跳到成员变量 Close 保存的函数入口地址”。我们会加入 DM4-12 第 4 行的汇编吗？肯定不会，因为这就是一个普通的用 C 语言函数指针调用函数，不必加入第 4 行。接着开始编译 DM4-13(b)的代码。看到 CloseTextFile 这个函数时，我们会自动加入第 2 行代码吗？不会，因为这个函数就是最普通的 C 语言函数，也不必加入这句莫名其妙的从 edx 读取值的指令。这里的关键是，不论调用方还是被调用方，在 C 编译器看来，它们都是普通的函数调用，自然不会加入“多余”的指令去传递和获取 self 指针。

为什么我们知道要在 DM4-12 第 5 行之前加入传递 self 指针的代码？因为在我们的头脑中有一个概念：“能动体”函数调用时为了自己操作自己数据需要悄悄传递 self 指针。（注意，我们有了新

的概念——能动体。如果去掉上面那句话中双引号所括的能动体一词，整个逻辑将不成立！）只有能动体才有这样的需求。而在刚才角色扮演的过程中，我们发现通过关键字 struct，编译器无法知道 DM4-12 中的 info 是能动体，在它看来这就是个普通的结构体，自然不会悄悄添加相关汇编指令。只有让编译器知道 info 是能动体，我们的愿望才能实现。只有定义一个新的关键字，代表能动体，如“class”。我们用 class 替换 DM4-8 中的 struct，如下：

```
class TTextFileInfo…
```

这时，编译器看到 DM4-12 第 5 行的 info.Close()，通过分析 info 的类型 TTextFileInfo 是 class，就知道 info 是能动体，而能动体的函数调用必须悄悄传入 self 指针。自然，编译器会添加 DM4-12 的第 4 行指令。但此时 DM4-13(b)的问题依然存在，从文字上，CloseTextFile 并非体现出它是能动体的函数，只是一个普通的 C 语言函数，所以编译器不会加入第 2 行指令。那么，仿造前面的思路，只要让编译器理解 CloseTextFile 是“能动体的函数”就可以了。这自然需要指出该函数是哪个能动体的函数，所以语法形式只要能表述出函数隶属于哪个能动体即可。

一种形式是：

```
能动体类型名字 + 分隔符号 + 函数名字
```

分隔符可以选择任何可用符号，如“::”或者“.”，那么 DM4-13(b)的函数声明就可改写为 void TTextFileInfo::Close()或 void TTextFileInfo.Close()。

另一种形式是：

```
class TTextFileInfo {
    void Close() {
      ….
    }
}
```

因为 Close()函数被外层花括号包裹在能动体 TTextFileInfo 定义之内，这自然表明该函数属于 TTextFileInfo。编译器看到以上两种声明形式之一，就知道 Close 是能动体 TTextFileInfo 的函数。而能动体的函数需要在一开始从约定寄存器 edx 获取 self 指针，那么自然会添加 DM4-13(b)中第 2 行指令去获取 self 指针。

总结一下：

① 用新关键字 class 替换 struct，使得编译器理解一个变量是能动体，从而当能动体函数被调用时就会自动加入指令按约定寄存器传递 self 指针。

② 用“能动体名::函数名”或其等价语法形式，可让编译器为能动体的函数悄悄从约定寄存器取出 self 指针。（学过面向对象语言的读者知道：class 就是类，能动体的函数被称为成员函数，能动体的变量被称为成员变量。）“类名::函数名”的形式是 C++成员函数的表示方法，“类名.函数名”的形式是 Delphi 成员函数的表示方法，如上将成员函数直接放入类的定义方式被 C++、Java 和 C# 采用。

③ 用结构体定义的变量称为结构体变量，而用类定义的变量被称为“对象”。对象其实就是一个结构体！

4.1.6 是这样吗？我们需要证明

通过和结构体对比，我们看到了类和对象的本质，现在需要实证。有以下几点需要证明：

① 对象成员变量的内存布局和等价的结构体变量是相同的，即类型定义从 class 改为 struct 是

DM4-14

(a)

```
class TestObjClass {
 public:
    int a;
    int b;
    int c;
    void setC(int v);
};
```

(b)

```
struct TestObjStruct {
    int a;
    int b;
    int c;
};
```

(c)

```
int main() {
    TestObjClass t1;
    TestObjStruct t2;
    t1.c = 1;
    t1.setC(2);
    t2.c = 3;
}
```

没有影响的。

② 增加成员函数后，应该在对象上多 4 字节，以保存该函数的入口地址。

③ 调用对象的成员函数前要悄悄传递 self 指针，可能是某个寄存器。

④ 成员函数可能从寄存器取得传递的 self 指针，并用它访问成员变量。

为此我们这样设计实验：首先定义一个结构体和一个类，它们有完全相同的成员变量及声明顺序，如 DM4-14。类 TestObjClass 有一个 setC 的成员函数，将参数 v 的值赋给成员变量 c，DM4-14(c)中分别定义了对象 t1 和结构体 t2；然后，用反汇编观察对它们成员变量 c 的赋值，从而验证是否两者的内存布局相同；查看 t1.setC(2)的反汇编，可分析是否悄悄传递了 self 指针，即 t1 的地址传给函数 setC；再跟踪进该函数，验证如何获取 self 指针并通过它访问对象成员变量 c。工程见“code\第 4 章\testsimpleobj”。

先比较结构体和设想中的对象内存布局，再用反汇编证明。两者的内存布局见图 4.1，如果对象布局如设想，那么 c 的地址应该是从对象首地址偏移 8 字节。调用 setC，应该从首地址偏移 12 字节的地方获取到函数地址，并 call 过去。来看反汇编的结果。

先验证对象和结构体成员变量的内存布局一样。从 DM4-15 第 2 行可见，t1.c 的地址是 ebp–8。这时在监视窗体中输入&t1 和(void *)(ebp–8)，查看两者是否如预料一样差 8 字节。见图 4.2，第 1、2 行中 t1 的首地址 0x0012ff58 与 t1.c 的地址 ebp–8 的值 0x0012ff60 正好差 8 字节，符合预期。同理，t2 的首址与 t2.c 地址（由 DM4-15 第 8 行可知为 ebp–1ch）之间也是差 8 字节（如图 4.2 后两行所示），符合预期。

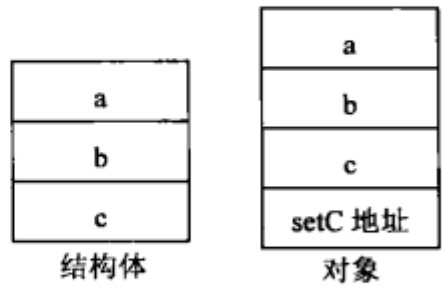


图 4.1

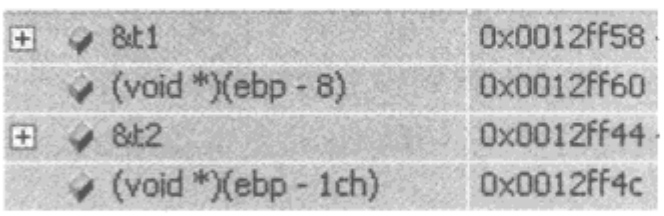


图 4.2

下面分析成员函数 setC 的调用过程。通过跟踪可知，DM4-15 第 6 行将跳转到函数 setC，第 4 行 push 2 正是传递参数 2。第 5 行对 ecx 赋值是否就是悄悄传递 self 指针呢？从 1.3.7 节的提示知道，lea 语句是将 ebp–10h 赋给 ecx。ebp–10h 是 self 指针，就是 t1 的首地址吗？从前面分析已知，t1.c 的地址是 ebp–8，减 8 字节就是 t1 的首址，t1 的首址=ebp – 8 – 8= ebp–10h，可知 ebp–10h 真的是 t1 的首址。大家应该用监视窗体核实。（**看来 VC 用 ecx 传递 self 指针。**）

DM4-15

```
1  t1.c = 1;
2  0041140e     mov   dword ptr [ebp-8], 1
3  t1.setc(2);
4  00411415     push 2
5  00411417     lea ecx,[ebp-10h]
6  0041141a     e8 c7 fc ff ff       call   004110e6
7  t2.c = 3;
8  0041141f     mov dword ptr [ebp-1ch], 3
```

接着跟踪进 setC 函数，查看代码是否从 ecx 获得 t1 的首地址，加 8 字节，获得成员变量 c 的地址，并赋值。setC 的反汇编见 DM4-16。在第 7 行前都没有修改 ecx，第 7 行将 ecx 压栈。中间省略了一些指令，第 8 行 ecx 又从栈中弹出，恢复为第 7 行压栈的值，即传入的 t1 首址（请在跟踪时用寄存器窗体查看 ecx 值验证之）。第 9 行将 t1 的首址赋给地址为 ebp–8 的内存，第 11 行又将它赋给 eax，最终导致 eax 存储了 t1 的首址，见图 4.3(a)。第 12 行将地址 ebp+8 指向内存中的值赋给了 ecx。现在需要分析 ebp+8 所指的内存，才能清楚 ecx 的值。

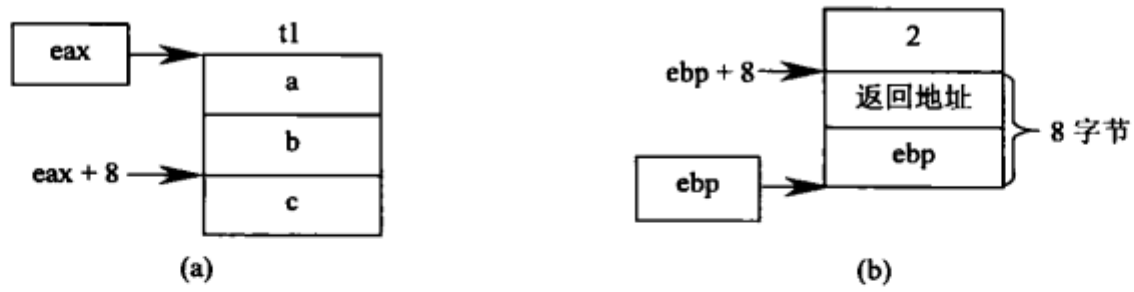


图 4.3

参考 1.3.4 节并对比 DM4-16，我们很容易画出栈帧的映像，见图 4.3(b)。其中，ebp+8 是被压栈参数 2 的地址，执行第 12 行后，ecx 存储的是 2，即参数 v 的值。第 13 行将 v 赋值给了地址为 eax+8 的内存。由图 4.3(a)分析可知，eax 此时是 t1 的首址，加 8 正好是成员变量 c 的地址。因此第 13 行将参数 v 赋值给了 t1 的成员变量 c，即第 10 行的源代码 c = v 要完成的逻辑。由此我们发现，Visual C++中，成员函数从 ecx 获取 self 指针。在 Visual C++中，第 10 行还可写成“this->c = v;”。原来 this 就是 self 指针，就是对象的首部地址，它被悄悄传入了。

DM4-16

```
void TestObjClass::setC(int v) {
1    004113a0    push ebp
2    004113a1    mov  ebp, esp
3    004113a3    sub  esp, 0cch
4    004113a9    push ebx
5    004113aa    push esi
6    004113ab    push edi
7    004113ac    push ecx
     ...
8    004113bf    pop  ecx
9    004113c0    mov  dword ptr [ebp-8], ecx
10   c = v;
11   004113c3    mov  eax, dword ptr [ebp-8]
12   004113c6    mov  ecx, dword ptr [ebp+8]
13   004113c9    mov  dword ptr [eax+8], ecx
```

梳理一下，发现本节开始要证明的 4 件事，还有问题②没有证明，即图 4.1 中我们假设对象有一个字段保存成员函数 setC 的地址，就像 4.1.3 节中 DM4-8 那样，定义了 4 个函数指针变量的效果。DM4-15 中，第 3 行源代码对应的汇编指令为第 4、5、6 行。第 4、5 行分别为传递参数和 self 指针，第 6 行调用 call 指令，似乎没有从 self 指针偏移 12 字节来获取 setC 的入口地址。按照我们的推理，应该是诸如 call [ecx + 0ch]这样的 call 指令（因第 5 行导致 ecx 中存储了 self 指针，想象中的 setC 的地址存储在离首址偏移 12 字节的地方，见图 4.1）。但第 6 行的指令实际是 call　004110e6，似乎直接就知道函数在哪里，如果查看其机器码，“e8 c7 fc ff ff”，参见 1.3.1 节知道，e8 是相对 call 指令的操作码，c7 fc ff ff 是偏移量。从小端机可知，偏移量是 ff ff fc c7，从最高位为 1 可知这是一个负数。对该值求反加 1，易计算出该值是-339h。由于偏移量从 call 指令之后的指令计算，即第 8 行的地址 0041141fh。那么，目标地址就是 0041141fh–339h=4110e6h，正好等于第 6 行汇编指令 call　004110e6 所指出的目标地址。所以，可以完全确定在面向对象的语法作用下，根本不需像 C 语言模拟数据与函数合一时那样，用函数指针变量保存成员函数地址。编译器因知道被调函数的地址，直接生成了相对 call 指令。**（为什么会这样？）**

来看 DM4-14(c)所示的 t1.setC(2)。编译器在编译阶段通过符号“t1.setC...”就明确知道这是调用类 TestObjClass 的成员函数 setC，即 TestObjClass::setC。TestObjClass::setC 就是编译器生成的代码，自然知道其入口地址并可算出 DM4-15 第 6 行指令的偏移量，所以可直接生成 call　004110e6。不妨与 C 语言调用函数对比，只要给出被调函数的名字，就可生成相对 call 指令调用它。t1.setC(2)被解释为 TestObjClass::setC(&t1, 2)，相当于直接给出被调用函数的名字，即 TestObjClass::setC，进行调用，自然可生成相对 call 指令。

如果运用第 3 章链接的知识，就连细节也能非常清晰。假定函数 TestObjClass::setC 在生成的 OBJ 文件中的符号是 TestObjClass::setC_XXXX。t1.setC(2)在编译生成的 OBJ 文件中的机器码是“e8 00 00 00 00”，后 4 字节的偏移量为 0，在链接时将重定位。在其对应的重定位项中，必然会指出相关符号是 TestObjClass::setC_XXXX，链接时就能算出 call 指令后续指令到函数 TestObjClass::setC

入口的偏移量，从而修改 call 指令的后 4 字节。用 PEview 打开 OBJ 文件，图 4.4 中将机器码和重定位表的相关部分放在一起，上部分为 OBJ 文件中对应 DM4-15 的机器码，它对应的重定位项位于图下部，横线标注的重定位位置的相对偏移量为 2bh。机器码“F8 01 …”开始于 RVA 值为 20h 的地方，20h 加 11 正好是 2bh，即图中箭头指示的 E8 后的第一个 00 字节，正是 DM4-15 第 6 行 call 指令中偏移量开始的地方。根据重定位项，关联符号是“?setC@TestObjClass@@QAEXH@Z”，代表函数 TestObjClass::setC，即链接时能计算出到符号“?set@TestObjClass@@QAEXH@Z”的偏移，并以之覆盖 E8 的后 4 字节即可。

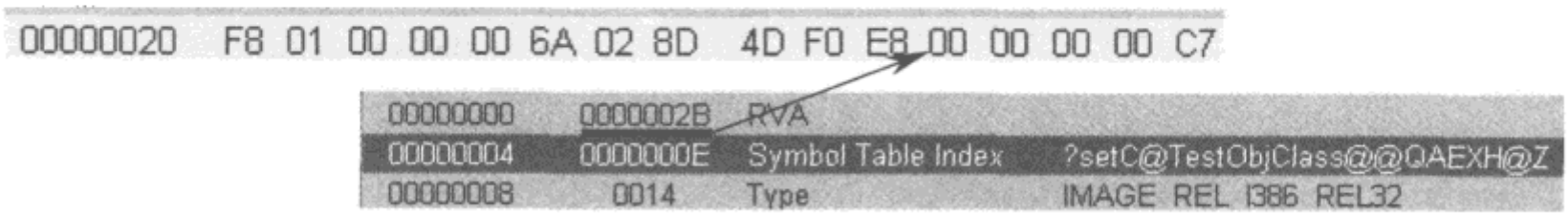


图 4.4

简单总结一下：

① 对象相当于结构体变量，其内存布局和去掉成员函数后的等价结构体完全相同。成员函数的数目不会影响对象的大小。（错误观念：不要在类上定义太多方法，否则对象的内存开销会很大。这是不了解面向对象本质导致的。）在内存耗费上，这要优于 4.1.3 节用 C 语言结构体模拟的对象，在那里，每个成员函数必须在结构体上对应一个函数指针变量存储其入口地址。

② 调用成员函数时都会自动悄悄传递对象首址，并在成员函数中自动获取该首址，用相对首址的偏移获取成员变量的地址，从而访问成员变量。通过自动给成员函数传递对象首址，面向对象语言保证了“自己操作自己的数据”，这与结构体必须外部手动传递结构体首址有本质区别。

③ 虽然我们用对象模拟了芯片，但两者还是有细节的不同。一个芯片都有一套自己的电路。虽然成员函数就像电路一样，但并非每个对象都有自己的一套独立函数，而是所有对象共享一套函数。通过悄悄传递不同对象首址，造成了对象有自己独立函数的假象。

4.2 太麻烦了，需要更简单的创造与销毁

4.2.1 创造构造函数和析构函数

有了“类就是结构体，对象就是结构体变量的认识”后，用对比分析有些东西就好思考了。怎么用类？该问题等价于“怎么用结构体”？结构体应该有两种用法：① 直接定义局部变量、参数或全局变量；② 定义结构体的指针，然后用 malloc 为该指针分配相应内存。关于①，实际在 DM4-14(c)中已经展示，对象相关的内存由编译器生成的代码自动分配。比如，局部变量和参数就在栈上，全局变量就在数据段。现在分析方式②。DM4-17(a)给出了结构体的实现，DM4-17(b)给出了对象版本。两个版本除了定义 Monitor 时一个用 struct 一个用 class，其余一模一样，都由 malloc 分配一块大小为 sizeof(Monitor)的内存，并用 free()释放。似乎没有什么特别之处。

既然对象是一个能做动作的能动体，是一个产品，那么产品往往在造出后会有一些默认参数设定。以 Monitor 为例，它代表显示器，有亮度 brightness 和伽马值 gamma 两个变量。如果造出一个显示器，这两个值都是 0，则表示什么也看不见。那么用户肯定很不满意，因为第一次打开时什么也看不见。所以，我们要给这个新构造出的显示器两个参数都赋予 70 的默认值，见 DM4-18(a)。

DM4-17

(a)

```
struct Monitor {
  int brightness;
  int gamma;
};
...
Monitor * pm;
pm = (struct Monitor *)malloc(sizeof(Monitor));
free(pm);
```

(b)

```
class Monitor {
  public:  int brightness;
           int gamma;
};
...
Monitor * pm;
pm = (Monitor *)malloc(sizeof(Monitor));
free(pm);
```

这样的代码意味着以后每构造一个新的对象，DM4-18(a)中的第 3～4 行必须重复。重复的过程中难保会粗心写错。从 3.1.4 节最后我们知道，“重复就是罪恶”，这是将来维护的祸根。所以，我们将第 3～4 行抽取为一个函数 initMonitor()，得到 DM4-18(b)。

在真实的世界，当一个产品（对象）报废的时候，往往需要先回收产品的零件，再行销毁。在代码中意味着调用 free()函数销毁对象前，需要调用函数 recycleMonitor()回收产品。在 DM4-18(c)中，该函数只是简单将所有参数赋为 0，第 9 行在 free 前调用 recycleMonitor()展示了这一过程。仔细审视该段代码，依然发现了“重复的罪恶”。因为造一个产品时，必然有两步：一是用 malloc()分配产品的内存空间，二是用初始化函数为它设定初始值。这意味着 DM4-18(c)中的第 7～8 行每次必须同时调用，顺序也必须正确。同理，第 9～10 行也必须同时调用。缺少其中任何一步或顺序错误都是不对的。这重复的两步应该消除，变成一步就好了。

既然有了新的面向对象编译器，我们来看能否借助新语法来解决这个问题。（**现在又是角色扮演的时候了。**）如果我们是编译器，总得知道哪个是类的初始化函数，哪个是回收函数，这样才能将两步合一。显然，必须让程序员通过某个特殊的标志将这两个函数标定出来。在 C++、Java 和 C#中，初始化函数的名字很特别，与类名一样；Delphi 中，初始化函数名前添加关键字 constructor（构造函数的英文表示）。而回收函数的标定，C++中是让函数名和类名相同，并且在名字前面加上符号“~”，如对 Monitor 类就是~Monitor()。在 Delphi 中，在该函数名前加上关键字 destructor（析构函数的英文表示）即可。至此，我们给初始化和回收函数分别取正式名字：构造和析构函数。

DM4-18

(a)

```
1   Monitor * pm;
2   pm = (Monitor *)malloc(sizeof(Monitor));
3   pm->brightness = 70;
4   pm->gamma = 70;
5   free(pm);
```

(b)

```
void initMonitor(Monitor * m) {
  m->brightness = 70;
  m->gamma = 70;
}
Monitor * pm;
pm = (Monitor *)malloc(sizeof(Monitor));
initMonitor(pm);
free(pm);
```

(c)

```
1   void initMonitor(Monitor * m)…
2   void recycleMonitor(Monitor * m) {
3       m->brightness = 0;
4       m->gamma = 0;
5   }
6   Monitor * pm;
7   pm = (Monitor *)malloc(sizeof(Monitor));
8   initMonitor(pm);
9   recycleMonitor(pm);
10  free(pm);
```

(d)

```
1   class Monitor {
2       …
3       Monitor(){ brightness = 70;… }
4       ~Monitor(){ brightness = 0;… }
5   }
6  }
7  pm = new Monitor();
8  delete pm;
```

最终在 C++下，我们获得了 DM4-18(d)的最终版本，很好地将分配对象内存和初始化，并将回收对象和释放内存合而为一。第 7 行的 new 相当于 DM4-18(c)的第 7 行分配内存，该操作符后面的“Monitor()”代表调用构造函数 Monitor()，相当于 DM4-18(c)的第 8 行。Monitor 的构造函数的代码完全与DM4-18(c)中的initMonitor()函数相同。同理，DM4-18(d)中第8行中的delete相当于DM4-18(c)的第 10 行，而它会先调用析构函数~Monitor()，相当于 DM4-18(c)的第 9 行。

至此，我们可简单总结构造和析构函数：

① 构造函数做初始化工作，并非分配新对象内存。析构函数做对象回收工作，并非释放对象内存。内存由 new 操作符分配，释放内存由 delete 释放。

② 因为①，所以构造和析构函数都要悄悄传递对象首址，否则无法完成初始化和回收工作。

③ 因为初始化方式可能有多种（如出口到不同地区的显示器，brightness 和 gamma 的默认值不一样），所以构造函数可以带参数，而且可以有多个。因为回收产品一种方式即可，所以析构函数没有参数，且只有一个。

④ 通过 new 配合构造函数，delete 配合析构函数，我们将内存分配与初始化，回收与释放内存二合一。消除了重复的弊端，简化了开发。

下面还是用实证来看看构造和析构函数是否符合以上总结。反汇编见 DM4-19。

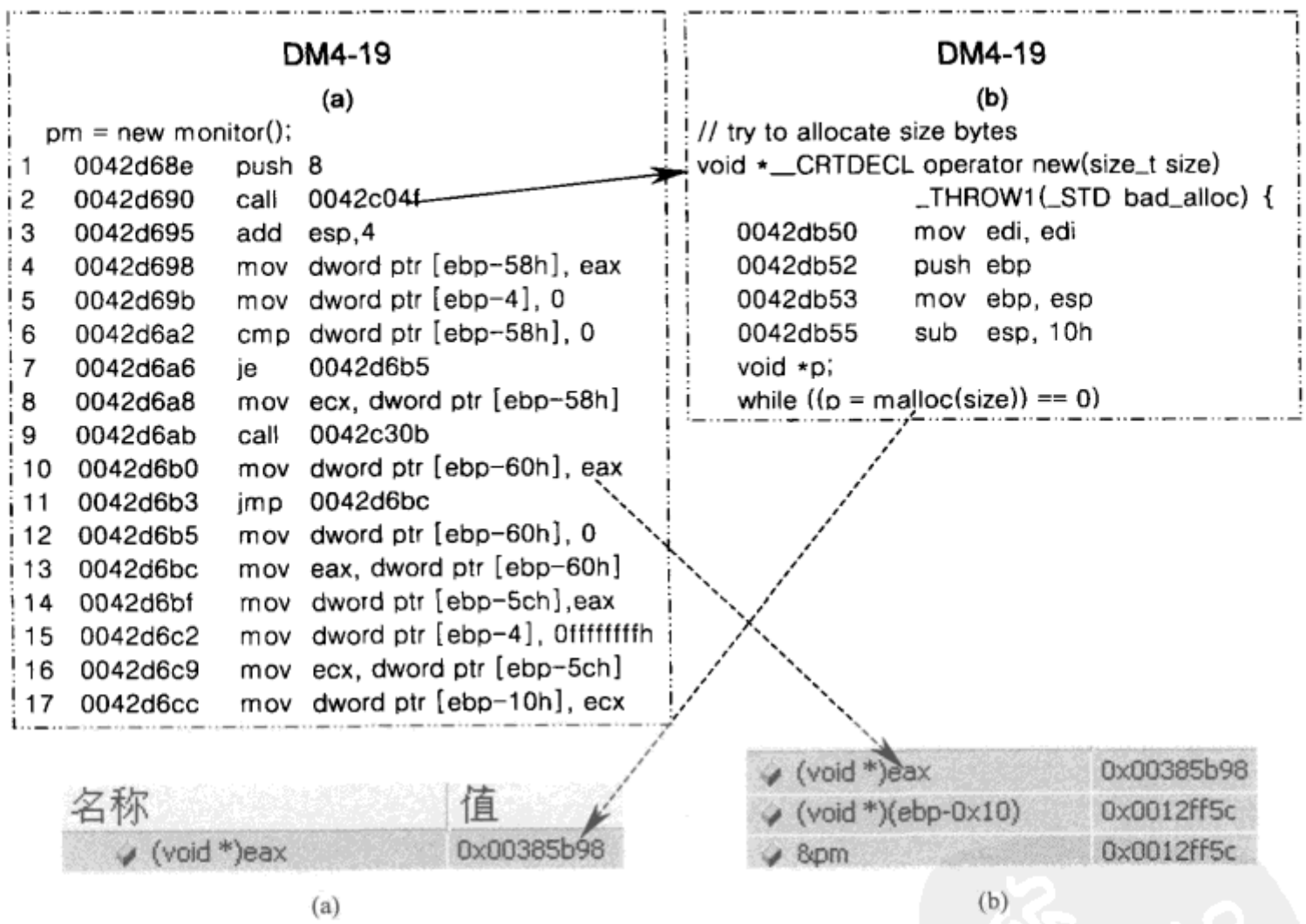


图 4.5

看反汇编代码的快捷办法是先要猜测可能的逻辑，再在代码中对照印证。因为代码太多且杂乱，如果没有事先猜测，很难下手。事先有了猜测，就很容易撇开非主线代码，从关键点下手。

按我们的猜测，“new 构造函数()”包含两步，分别对应分配内存和调构造函数。DM4-19(a)正好包含了两个 call 指令，分别为第 2 行和第 9 行，自然猜测第 2 行调用为分配内存。分配内存需要传入内存大小，因此第 1 行压栈传递 8 应该就是欲分配内存的大小。从 DM4-17(b)类的声明来看，它包含了 2 个 int 成员变量（int 大小为 4 字节）。按照前面已得出的对象模型，该对象就是 8 字节

大小。这与第 1 行传递 8 正好匹配。（**初步证明有门儿。**）再跟踪进函数，看到 DM4-19(b)的代码，“operator new(size_t size)”说明参数确实是内存大小，代码中的 malloc 也说明了这是在分配内存。malloc 的返回值即新内存的地址保存在 eax 中，只需追踪这个值是否就是最终在 DM4-19(a)中赋值给指针 pm 的值即可。这里先记录下 malloc 返回后 eax 的值，图 4.5(a)中为 0x385b98。然后在该函数返回前（执行到 ret 时）再查看 eax 值，如果同样是 0x385b98，则说明 new 操作符将此分配内存返给了调用者。实验结果正如猜测。

继续分析返回后的代码。第 4～8 行将新分配内存地址最终赋值给了 ecx。（**这是在做什么？**）分析可能关联的第 9 行，发现它调用了我们猜测的构造函数。构造函数要初始化对象就需要传入对象首址。由 4.1.6 节知，Visual C++用 ecx 向成员函数传递对象首址，那么构造函数也该从 ecx 传递。第 4～8 行正好完成了该工作。（**这就对上了。**）跟踪进第 9 行所调函数，证明果如猜测。但目前为止还没发现对局部变量 pm 赋值的指令。（**继续猜测。**）通过观察，第 9 行调用了构造函数后，第 10 行将 eax 的值赋值给了地址为 ebp–60h 的内存。由于 eax 扮演保存返回值的角色，自然猜想这个 eax 是对象首址。用调试器查看可证实该猜想，见图 4.5(b)，此时 eax 的值依然与图 4.5(a)指示的值相同，即 0x385b98（已经证明，该值是对象首址）。剩下的就是看首址的传递过程，从第 13～17 行我们发现，该地址被传递的序列是：地址为 ebp–60h 的内存 → 地址为 ebp–5ch 的内存 → ecx → 地址 ebp–10h 的内存。因此，ebp–10h 极有可能就是指针变量 pm 的地址。该值也符合第 1 章的结论“局部变量地址往往是 ebp–xxx”。从图 4.5(b)可知，ebp–10h 和&pm 的值都是 0x0012ff5c，因此 ebp–10h 就是 pm 的地址。通过第 13～17 行完成了将新构造对象首址赋给对象指针变量 pm 的逻辑。

4.2.2 构造中分配资源，析构中释放资源

4.2.1 节中，我们理解了构造函数用于初始化，析构函数用于最后回收。具体讲，一个产品制造时，在初始化阶段和回收时常常做的就是分配资源和回收资源。比如，生产显示器时要为它分配相关零件，报废时要回收部分可用零件。这样的资源分配和回收就常常出现在代码中。如果在构造函数中不分配相关资源，在使用对象时再分配就很容易出错。如果对象构造好时，资源也分配好了，使用起来就非常方便。析构函数中不释放相关资源，就意味对象已经销毁了，而资源还没有回收，岂非浪费？

来看一个很简单的情况，见 DM4-20。构造 Person 对象时，从构造函数传入身份证号地址 aID。在构造函数中，用 malloc 分配一块与字符串 aID 同样长的内存，然后将 aID 的内容复制到新内存中。成员变量 ID 保存新分配内存的地址。释放对象时，在析构函数中调用 free 释放 ID 指向的内存。如果不释放，就意味着内存泄漏。

也许大家觉得这样很麻烦，有人会写出如 DM4-21(a)所示的代码，直接将参数 aID 赋值给成员变量 ID，这样不仅构造函数简单，还可省略析构函数。来看 DM4-21(b)，工程见“code\第 4 章\Person”。最后打印时，p1 的 ID 打印正确，而 p2 的 ID 打印出来是乱码。为什么？此时，我们不外乎从两个角度去思考：分析可能原因，或者追踪过程，剥丝抽茧，观察端倪。

首先分析可能原因。p2 的 ID 内容错误有两个可能：① 成员变量 ID 的值被修改了，使得它指向另外一段包含乱码的内存；② ID 依然指向原来的内存，但其内容被修改。只需在 DM4-21(a)的赋值语句中设置断点，查看 ID 的值，并在 DM4-21(b)的第 12 行跟踪进 getID()函数，查看此时 ID 的值，就很容易排除原因①，因为 ID 值没有变化，指向的是同一块内存 0x12fe68。只能是原因②，该内存值被修改了。设置断点，每步执行都关注对象 p2 的成员变量 ID 指向的内存 0x12fe68，如图

4.6 所示。可确定，执行 DM4-21(b)的第 11 行后，内存值改变了。为什么执行了函数 createPerson1()后就出问题了？是该函数出错了吗？（**可该函数没一点问题。**）

DM4-20

```
class Person{
      char * ID;
      ...
   public:
      Person(char * aID);
      ~Person();
      ...
};

Person::Person(char * aID) {
    ID = (char *)malloc(strlen(aID) + 1);
    strcpy(ID, aID);
};
Person::~Person() {
    free(ID);
};
```

DM4-21

(a)

```
Person::Person(char * aID) {
    ID = aID;
};
```

(b)

```
1    Person * createPerson1() {
2       Person * p = new Person("123");
3       return p;
4    }
5    Person * createPerson2() {
6       char Id[4] = "123";
7       return new Person(Id);
8    }
9    void main(){
10      Person *p2 = createPerson2();
11      Person *p1 = createPerson1();
12      printf("p2->Id=%s p1->Id=%s\n", p2->getID(), p1->getID());
13   }
```

地址：0x0012FE68
0x0012FE68 31 32 33 00

(a) 正确的内存值

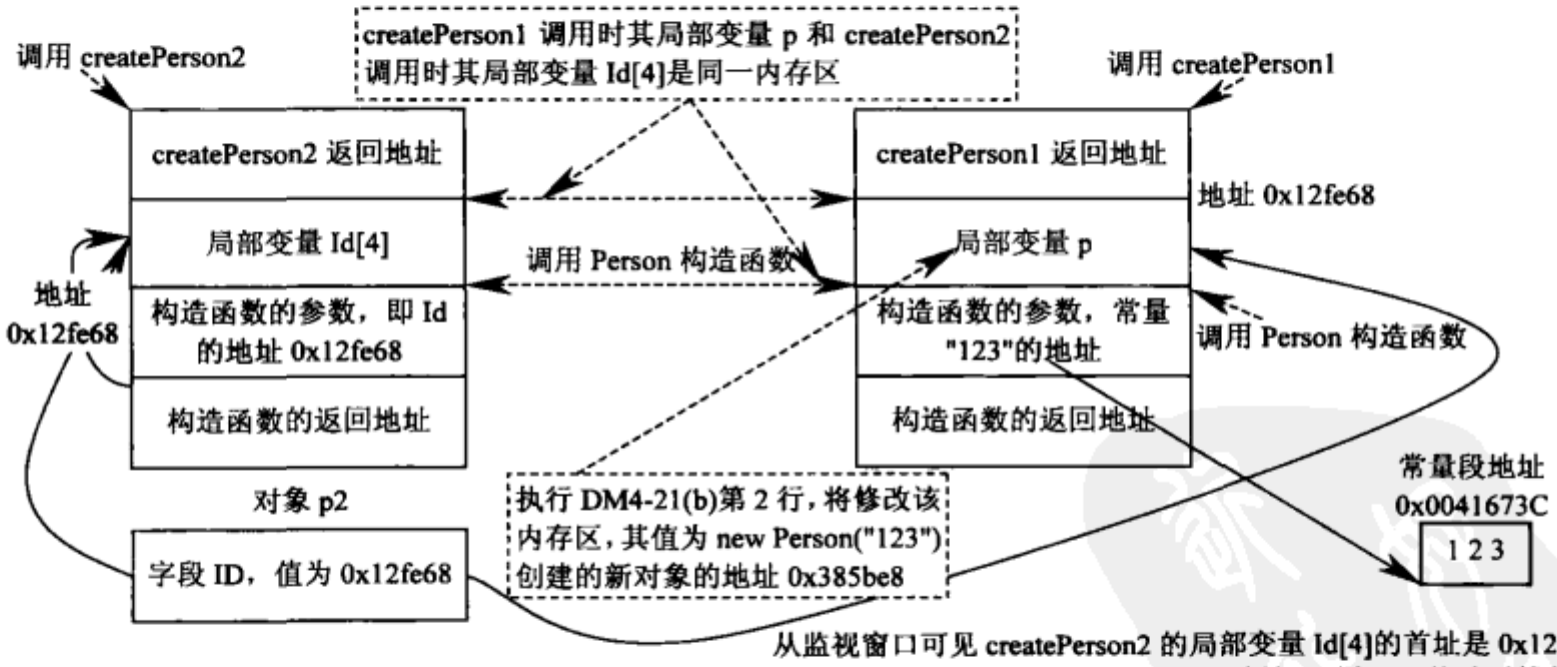
地址：0x0012FE68
0x0012FE68 e8 5b 38 00

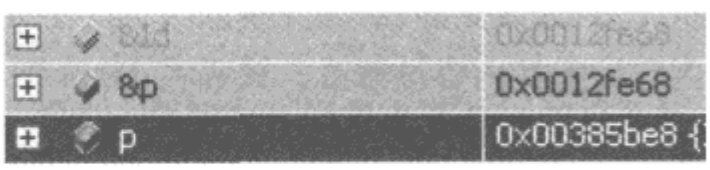
(b) 错误的内存值

图 4.6

其实，有时出现错误的地方并非错误发生的原因。如果只盯住 createPerson1，可能永远找不到原因。这时，“追踪过程，剥丝抽茧”可能就能带我们走出泥潭。请读者先画出 DM4-21(b)调用过程中栈及对象的变化，再看本书的描写。

在图 4.7 中，左边是执行第 10 行时的状态，右边是执行第 11 行的状态。





从监视窗口可见 createPerson2 的局部变量 Id[4]的首址是 0x12fe68，等于 createPerson1 的局部变量 p 的地址。对象 p2 构造时接收的是 Id 的地址，并直接赋给了成员变量 ID。那么，p2 的 ID 也指向 0x12fe68。执行了 createPerson1 后，p 的内容被修改为 0x00385be8。因为 p2 的 ID 指向的就是此时变量 p 的值，所以这时 p2 的 ID 指向的内存就变成了对象 p1 的地址。由 4.6(b)可知，在执行 DM4-21(b)的第 12 行打印时，对象 p2 的成员变量 ID 指向的内存中正是 e8 5b 38 00，从小端机来看，正是整数 0x00385be8。这就是打印乱码的根源。

图 4.7

由图 4.7 可知，createPerson2 将其局部变量 Id[4]的地址 0x12fe68 传递给 Person 构造函数，因此构造出的对象 p2 的成员变量 ID，保存的是局部变量 Id[4]的地址。当其返回后，调用 createPerson1，此时该函数的局部变量 p 正好与之前函数的局部变量 Id[4]占用同一内存空间。因此，createPerson1 返回后，这块空间的内容被改写为对象 p1 的首址 0x385be8。此时 p2 的成员变量 ID 虽仍指向该内存，但内容完全变了，形成乱码。其实之后调用 printf 时，该块内存区因用作其参数、局部变量，又被再次修改。这个错误并非是执行了 createPerson1 造成，根源在前面的 createPerson2。其实简单地想，DM4-21(a)直接保存了外部传入的指针，而没有保证该指针指向的内存不会被其他地方修改，因此结果就没有保证。所以，就如 DM4-20，对象自己分配一块内存复制传入字符串内容才是正确做法。为什么 new Person("123")不出错？从 2.5 节知道，双引号引用的字符串分配在常量段，这时传入地址属于常量段。常量段内存只读，不会被修改，所以不用担心其内容变化。而简单测试时往往会以该方式使用字符串，就会将错误隐藏起来。这个例子让我们再次感受到底层知识的力量。

4.3 对比 C 语言的“对象”和面向对象

在把玩了面向对象的基本要点后，是时候对比一下 C 语言版“对象”和真正的对象了。我们很容易将 4.1 节的 C 语言版对象从 DM4-22(a)的代码转换为 DM4-22(b)所示的真正对象。从字面上，两者的定义一个为 class，一个为 struct。结构体版本的 4 个成员函数以函数指针形式出现。相应的内存布局见图 4.8，C 语言版本的对象，每个函数指针都占据内存，而真正的对象只有成员变量，函数并不占用对象内存空间。工程见“code\第 4 章\EncapsulateRealObj”。

DM4-22

(a)

```
struct TTextFileInfo {
    FILE * F;
    char FileName[256];
    int CurrentLine;
    int TotalLine;
    char* Mode;          //the open mode of the file
    TOpenTextFile Open;
    TCloseTextFile Close;
    TReadLine ReadLn;
    TWriteLine WriteLn;
};
```

(b)

```
class TTextFileInfo {
  public:
    FILE * F;
    char FileName[256];
    int CurrentLine;
    int TotalLine;
    char* Mode;
    TTextFileInfo(char * FileName, char * Mode);
    BOOLEAN Open();
    void Close();
    BOOLEAN ReadLn(char * Ln);
   void WriteLn(char * Ln);
};
```

DM4-23 展示了成员函数的区别。C 语言版必须从参数显式传入结构体的首部地址，以供访问成员变量使用。而真正的对象不用显式传递参数。对象首部地址（即 this 指针）是悄悄被编译器传入的，也用于访问成员变量。因此 C 语言版用代码“pInfo->F”访问成员变量 F，而对象版用代码“F”或“this->F”访问成员变量 F，形式上似乎解脱了程序员主动传递内存首址的麻烦，以及防止用户传错的可能。实际上，编译器悄悄传递 this 指针却导致了“自己操作自己数据”这一封装理念的诞生。这是一个简单而又伟大的转变。可以说，面向对象语言玩了两次“指针的把戏”，就创造了崭新的面向对象方法学。这次是 this 指针暗度陈仓的把戏，另一次将在 4.7 节中亲历其诞生。

DM4-23

(a)

```
BOOLEAN ReadLine(PTextFileInfo pInfo, char *Ln) {
    if (feof(pInfo->F))
        return FALSE;
    readLn(pInfo->F, Ln);
    pInfo->CurrentLine++;
    return TRUE;
}
```

(b)

```
BOOLEAN TTextFileInfo::ReadLn(char *Ln){
    if (feof(F))
        return FALSE;
    readLn(F, Ln);
    CurrentLine++;
    return TRUE;
}
```

说到封装，笔者想到了多年前学习面向对象的一段场景，而且类似的场景在后来很多书中也不断提及。如 DM4-14(a)所示，一个类有一个成员变量（如 int c），然后编写读取和设定该变量的成员函数（如 setC）。这些我们称为 getter()和 setter()的简单函数，仅仅是直接对相关变量写或读。这些最简单的 getter()和 setter()函数就被宣扬为体现了面向对象的封装特性。

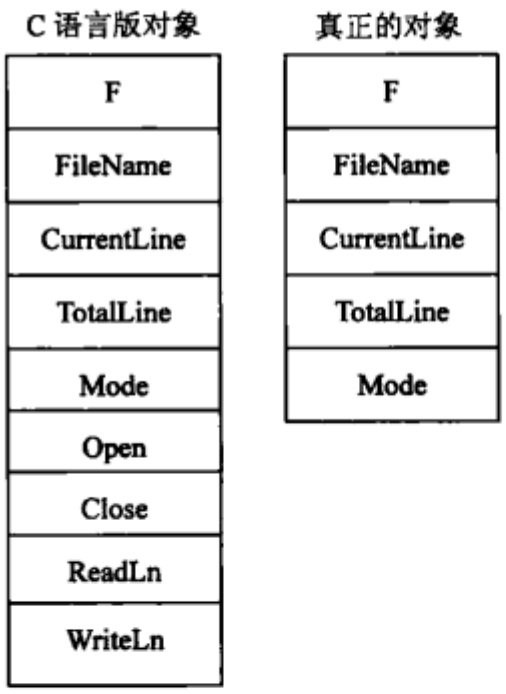


图 4.8

```
void TestObjClass::setC(int v) {
    c = v;
}
```

很不理解，这到底怎么就封装了？怎么就先进了？记得那时就假想出如下代码：

```
TestObjClass o;
o.c = 12;
o.setC(12);
```

对成员变量 c 的赋值不论是直接赋值还是用 setC 赋值，反正都要赋值，直接赋值还比函数调用少了参数传递、跳转和返回，明显要快。功能完全一样，直接赋值性能还好。当时也就不了了之。

其实，面向对象的一大优势是体现在代码变化时，如需要维护、需要扩展的时候，通过封装，我们能改变类自身的代码，而不让调用方代码修改，能将变化的影响遏制在涉及类的内部，而不传播出去。试想，一个对象就是一朵小雪花，一座雪峰就是一个工程。如果一朵雪花的变化不断导致相邻雪花的变化，最终可能导致雪崩。因此，将变化遏制在内部是很有价值的。下面来看 setter()、getter()函数是如何遏制“雪崩”的。

如果程序需要变化，设定变量 c 时要检验设定值的有效性。如果是直接赋值成员变量 c 的方式，则代码如下：

```
TestObjClass o;
if (i < VALIDVALUE)
    o.c = i;
```

这时必须将所有对成员变量 c 赋值的代码全部修改，加入 if 判断。如果没有修改，则产生 bug。而将来的赋值也要加入这个判断，忘记加入（太容易忘记了，谁记得这个有效性要求呢？）也是错误。这里其实又体现了“重复即罪恶”的理念，因为 if 语句被不断重复。怎么办？我们知道，重复的代码就放入函数，由重复变重用。实际上，如果我们开始是通过 setC 设定成员变量 c，那么将 if 判断加入到成员函数 setC 中，调用方代码根本就不变。而且将来调用 setC 时，自动使用了有效性判断逻辑，不会再发生疏漏。

重复即罪恶

```
void TestObjClass::setC(int v) {
    if (v >= VALIDVALUE)
        throw new Exception….
```

```
        c = v;
    }
```

也就是说，在最简单的 getter()和 setter()函数中，我们确实没有直接读取/赋值成员变量有效。但通过它们，我们设置了一道防火墙，将来代码演变时，保护调用方代码不变，将所有变化隐藏在 setter()和 getter()内部。根据“自己操作自己数据的封装”原则，外部调用代码不清楚内部数据的细节，所以将来这些细节变化时，也自然影响不到它。这就是封装的关键意义所在。

现在一点小小的遗憾就是在最简单的 getter()和 setter()下，效率有点损失。不过在 .NET、Java 这些支持 JIT 的平台上，这个遗憾将不存在。因为运行时要即时编译，虚拟机发现 getter()、setter()函数很小，就会用内联消除对它们的调用（会复制 getter()、setter()的指令替换调用指令）。未来如果修改 getter()、setter()，导致其代码量增加，即时编译结果又会变成函数调用形式。因此，在这类平台上，我们更会不加考虑地将所有成员变量的访问通过函数来完成。

4.4 体验封装的力量

下面将把 C 语言中最常用的随机文件操作的函数封装成文件流类，从这个简单的任务中感受类的设计。如果能按照责任划分形成类，责任实现形成成员函数的方法，就能在面向对象设计上迈出有力的步伐了。文件流的责任是什么？自然是操控文件。操控文件应该包含哪些操作？打开/关闭文件，读/写文件，定位文件游标，判断文件是否结束。而一切都要从打开和关闭文件开始，这是其基本。在开始这些前，我们按照责任给这个类起一个名字“FileStream”。

4.4.1 生死原点，整体资源管理

打开关闭文件很简单，需要什么？我们定义两个成员函数：open()打开文件，close()关闭文件。假想使用它们的代码，见 DM4-24。这里存在以下问题：① 当创建了对象 f 后，忘记调用 open()，则后续操作会失败；② 在 delete f 执行前，忘记调用 close()函数，那么对象释放后打开的文件并没有关闭，将造成资源泄漏（文件开了就必须关）。如果调用了 close()却忘记调用 delete f，将导致内存泄漏。究其原因就是：文件流对象需要文件资源，构造对象时并没有打开该资源；对象销毁时，关联的资源并非连带关闭，而要通过另一个 close()函数完成。

DM4-24

```
FileStream f = new FileStream();
f->open("C:\\temp.txt");
//....读写文件
f->close();
delete f;
```

这让我们想起了在 4.2.2 节中讨论的一个设计原则：“在构造中分配资源，在析构中释放资源”。这个原则又被称为整体资源管理（general resource management）或 RAII（Resource Acquisition Is Initialization，资源获取就是初始化）。应用这一原则，我们将在构造函数中打开文件，在析构函数中关闭文件，这时使用者代码如 DM4-25 所示。这时，前面提到的问题均不复存在。

DM4-25

```
FileStream f = new FileStream("C:\\temp.txt");
//....读写文件
delete f;
```

FileStream 的构造和析构函数见 DM4-26(a)，传入要打开的文件名和打开模式。fopen()返回的文件指针存到哪里呢？析构函数中关闭文件就要用到。本着“自己操作自己数据”的原则，不应该让调用者传入文件指针。“自己的数据”就是成员变量，因此该文件指针就定义为 FileStream 的成员变量，即第 2 行中的 fp。随之而来的好处就是析构函数中直接用此 fp 关闭文件，见 DM4-26(b)。（**自己操作自己数据，关闭文件时，我们不用担心关错文件了。**）对于 fclose()，如果用 C 语言风格编程，程序员可能传入不正确的文件指针。再看 DM4-25，“delete

DM4-26

(a)

```
1    FileStream::FileStream(char * fileName, char * openMode) {
2        fp = fopen(fileName, openMode);
3        openResult = fp != 0;
     };
```

(b)

```
FileStream::~FileStream(){
    if (fp != 0)
        fclose(fp);
};
```

(c)

```
class FileStream{
  public:
    FILE *fp;
    FileStream(char * fileName, char * openMode);
    ~FileStream();
}
```

f;”根本不需要我们传入文件指针变量，就不会出错了。这时类的定义见 DM4-26(c)。

DM4-25 中还有一点点瑕疵，如果程序员忘记删除对象，文件自然就没有关闭。能否让代码自动关闭文件和销毁对象呢？对于只能基于堆分配对象的语言（如 Java 必须用 new 操作符显示分配对象）我们没有更好的办法，但对于能基于栈或全局数据段分配对象的语言（不用 new 分配对象，定义了对象就可直接使用），我们就有些技巧了。下面来看 C++基于栈分配对象的能力，见 DM4-27。我们定义文件对象 f，然后可直接使用它，并没有显式调用 new 分配对象。从道理上讲，编译器应该保证调用了构造函数，同时在不需要对象时调用其析构函数。通过简单的代码跟踪，很容易证明以上结论。利用自动调用 FileStream 的构造函数，就自动打开了文件；而利用自动调用析构，编译器帮我们自动关闭了文件。对比 DM4-25，DM4-27 中虽然只是少了“**delete f;**”，却保证了内存和文件资源的双重自动回收！程序员更不易犯错了。为了清晰地把握这种方式中打开和关闭文件的时机，我们应该确定构造和析构函数自动调用的时机。还是遵循先猜测再实证的方法，大家觉得何时调用构造合理呢？不可能是定义对象变量前调用，那时还没有对象。为了能安全可靠使用对象，必须保证在使用该对象前就调用了构造函数。自然的位置应该是紧邻定义该对象的地方。析构呢？就是函数中再也不用该对象的时候，直觉就可在函数体的右花括号前调用析构，因为此时该对象变量无效了。下面就用反汇编跟踪来实证。

如 DM4-28 所示，定义变量的语句是第 1 行，第 2～3 行就是对应指令，第 2 行通过 ecx 传入对象 m 的首址，第 3 行调用了类 Monitor 的构造函数。（**这印证了我们的猜想。**）第 6 行是函数体结束的右花括号，对应指令为第 7～10 行。第 7 行传递析构需要的 this，第 8 行调用 Monitor 的析构函数。（**这也印证了我们的猜测。**）

如果只能在函数返回的右大括号才能释放局部变量，似乎还不够灵活。从上面分析可见，编译器是依据“何时变量无效何时调用析构”的原则。C++有没有更灵活的方法控制局部变量的有效范围呢？答案是肯定的：局部变量的有效范围就是声明它的语句所在的花括号所包含的范围。如 DM4-29 所示，变量 m 的有效范围就是内花括号的区域。是否与前面的猜测一样，在内花括号的结束处调用析构函数？用反汇编分析，很容易得到证明，此处不再给出结果。

DM4-27

```
void myfunc(){
    FileStream f("C:\\temp.txt", "w+");
    ...                          //读写文件
}
```

DM4-29

```
void myfunc(){
    {
        Monitor m;
    }
}
```

DM4-28

```
1    monitor m;
2    0042d679    lea    ecx, [m]
3    0042d67c    call   monitor::monitor(42c30bh)
4    m.brightness = 12;
5    0042d681    mov    dword ptr [m], 0ch
6 }
7    0042d688    lea    ecx, [m]
8    0042d68b    call   monitor::~monitor(42c310h)
9    ...
10   0042d698    ret
```

最后总结一下：

通过整体资源管理，将对象所依赖的资源在构造函数中分配，在析构中释放。在此基础上，定义局部变量可保证资源的自动分配和释放。

在指定的花括号中定义局部变量，能精确地控制构造和析构调用的时机。

4.4.2 文件流

解决了文件的打开和关闭，我们继续 FileStream 的设计之旅，还应该实现如下功能：

① int read(void* buf, int size)　　将 size 大小的数据读入 buf，实际读入的多少由返回值确定，返回-1 为错误。

② int write(void* buf, int size)　　将 size 大小的数据从 buf 中写入文件，返回值表示实际写入的大小，返回-1 为错误。

③ int locate(long offset, int from)　　移动文件流的游标。from 指定偏移方式：_FromCurrent 代表从现在位置计算偏移，_FromStart 从开始位置计算偏移，_FromEnd 从结束位置计算。offset 为偏移字节数。

④ bool eof()　　判定文件是否到达结尾。

它们几乎与 C 语言版函数一模一样。比如，C 语言版的 feof 需要传入文件指针参数，对象版的 eof 不传递，直接从自己的成员变量 fp 获取需要的文件指针。与 C 语言版函数一个参数的差别体现了面向对象自己操作自己数据的封装理念，不会发生 C 语言版存在的问题。而其余成员函数也均未从外部传入文件指针参数，直接访问成员变量 fp。

```
bool FileStream::eof() {
    if (fp == 0)
        return true;
    return feof(fp) != 0;
};
```

下面代码展示了文件流类的使用，以及自动打开和关闭文件的技巧。它打开一个文件，并打印其内容。工程见“code\第 4 章\FileStream”。

```
void main {
    ...
    {
        FileStream fStream("C:\\temp.txt", _OpenRead);          //此处自动打开文件
        int count;
        if (!fStream.openResult) {
            printf("open error\n");
        return;
        };
        while (!fStream.eof()) {
            count = fStream.read(buf, 100);
            if ((count > 0) && (count < 100)) {
                buf[count] = 0;
                printf("%s", buf);
            }
        }                                                       //此处自动关闭文件
}
```

4.5 整体资源管理的爱恨

4.5.1 扩展技巧：保证成对出现，巧妙的自动线程锁

“整体资源管理”+“局部变量自动调用构造和析构”保证了资源的自动分配和释放，是 C++这种支持基于栈分配对象语言（C#也可以，Java 不行）的独特技巧。如果进一步抽象这一方法，发现它保证了一些必须成对执行的行为被成对调用。除了资源的分配和释放要成对出现，还有什么东西要成对执行？那就是为了保证多线程同步而是用的锁机制。比如，Windows 下为了保证多线程访问临界区而提供了两个函数：

```
VOID EnterCriticalSection(                          //该函数返回意味获得了锁，可操作临界资源
    LPCRITICAL_SECTION lpCriticalSection            //临界区变量的指针
);
VOID LeaveCriticalSection(      //调用该函数，释放上面函数获得的锁，其他线程才能获得该锁
    LPCRITICAL_SECTION lpCriticalSection            //临界区变量的指针
);
```

为了保证临界资源的安全性，当多线程访问一个临界资源时，如一个全局变量，需要先调用 EnterCriticalSection 获得锁，再访问全局变量。访问完毕，调用 LeaveCriticalSection 释放锁，这样其他线程才能再次用 EnterCriticalSection 获得锁。代码如下：

```
LPCRITICAL_SECTION p;
...
EnterCriticalSection(p);
gloableVariable = ...
LeaveCriticalSection(p);
```

简单理解临界资源的原理也很容易：线程就像一个人，多线程就是多个人。临界资源就像一块黑板，当多个人要“共用”一块黑板时，必须协调他们，必须一个个“依次”使用黑板，不能“同时”使用，否则就会乱。假定有 A 和 B 线程都执行上面代码，当 A 线程调用 EnterCriticalSection 返回时，意味着当前无人运行其后的代码，它就可安全执行 globalVariable 的赋值语句。此时若线程 B 又调用了 EnterCriticalSection，操作系统就会阻塞 B 的运行，因为锁 p 已经被 A 获取了。当 A 调用 LeaveCriticalSection 释放 p 后，B 才会从 EnterCriticalSection 返回，使得 B 可执行其后代码。这就是同步，保证某时间只有一个线程执行 EnterCriticalSection、LeaveCriticalSection 之间的代码。

如果忘记写 LeaveCriticalSection 的后果是什么？那意味着 A 执行完毕后没有释放锁 p，B 线程将永远阻塞在 EnterCriticalSection 上不返回，就会形成所谓的“死锁”。这是命令式语言（imperative）进行多线程编程的硬伤。（**函数式语言似乎成为了这个问题的银弹**。）

这个必须保证 EnterCriticalSection 和 LeaveCriticalSection 成对出现的要求似乎很简单，只要养成习惯先写这两个函数，再在其间编写临界区代码就好了。注意，要程序员养成“好”习惯，这本身就引入了不可靠因素。另一方面，即使你有这个好习惯，也很难保证安全。假定有如下代码，临界区语句由很多条 if-else 分支嵌套组成。

```
EnterCriticalSection(p);
 if …
   else if ….
    else if…
    ….
LeaveCriticalSection(p);
```

当程序使用了较长时间后，因为功能变化需要修改。程序员费力地在条件分支丛林中找到了要增加的分支，他此时已经忘了临界区的问题，在该分支中添加了相关代码就直接 return 返回。自然，当代码执行该分支后，锁资源未释放，于是引发死锁。

既然 EnterCriticalSection 和 LeaveCriticalSection 有成对调用的要求，我们能借鉴 4.4.1 节文件自动打开和释放的技巧吗？显然可以。定义锁类 Lock，将 EnterCriticalSection 的调用放入其构造函数，而 LeaveCriticalSection 的调用放入其析构函数。并通过定义 Lock 类的局部变量就能因自动执行 Lock 的构造函数而执行 EnterCriticalSection；因自动执行析构函数而调用 LeaveCriticalSection。临界区代码放在 Lock 局部变量有效域中即可，如下代码：

```
{
    Lock alock(p);              //此处因自动调用构造函数，而执行 EnterCriticalSection
    if
        ...
    else if
        ....
    else if
        ...
    ...
}                               //此处因自动调用析构而执行 LeaveCriticalSection
```

剩下的工作就是根据以上代码来设计 Lock 类，见 DM4-30。构造函数中不仅进入临界区，而且将临界区指针记录到成员变量 pSec 中。析构时，直接调用 LeaveCriticalSection()。注意，由于析构从成员变量 pSec 获取临界区指针，就杜绝了调用 LeaveCriticalSection()时传错指针的错误。在前面直接调用两个函数的版本中，很可能 EnterCriticalSection() 传入的是临界区的指针 A，而 LeaveCriticalSection()却传入指针 B，即获取锁 A，释放的却是锁 B。这也是死锁的一个原因。这又一次展现了封装操作自己数据的力量。

DM4-30

```
class Lock{
    LPCRITICAL_SECTION pSec;
  public:
    Lock(LPCRITICAL_SECTION p){
      pSec = p;
      EnterCriticalSection(pSec);
    }
    ~Lock(){ LeaveCriticalSection(pSec); }
}
```

4.5.2 美丽的幻影：不可靠的自动析构

4.5.1 节的自动线程锁多么美妙，保证了锁一定会获取和释放。但现实世界真的如此美好么？运行 DM4-31 所示的代码，函数 test 中定义了局部变量 c，并在其后对空地址赋值以产生异常。异常发生后，会调用 TestClass 的析构函数吗？按理说，定义局部变量 c 将调用构造函数，而因异常离开 c 的作用域，应调用析构函数。如果不调用，那么 4.5.1 节的线程锁将失去作用，死锁将发生。

如果调用了，将打印“do constructor”。实验的结果是：Visual C++ 6.0 不调用析构，GCC 不调用析构（版本为 4.3.3）。VS 2008 中，如果将项目属性的选项“属性 | C/C++ | 代码生成 | 启用 C++ 异常”选择为“是，但有 SEH 异常(/EHa)”，就会调用析构，选择“否”就不调用析构。（如果 main 中不加 try..catch，无论选择什么，Visual C++都不会调用析构。这似乎也好理解，如果程序不显式用 try…catch 处理，程序就会终止。程序都终止了，也不在乎调不调用析构了。）G++有 Visual C++ 7.0 类似的选项控制。可见，编译器在处理时并没有遵循统一规范。

Lippman（Visual C++ 7.0 的编译器开发领导者）的大作《Essential C++》7.4 节中明确提到，“在异常处理机制终结某个函数前，C++保证，函数中所有局部对象的 destructor 都会被调用。”（**看来经**

DM4-31

(a)

```
class TestClass{
  public:
    TestClass() {
      printf("do constructor\n");
    }
    ~TestClass() {
      printf("do destructor\n");
    }
};
```

(b)

```
void test() {
  TestClass c;
  *(int *)0 = 1;
}
int main() {
  try {
    test();
  }
  catch(...) {
    printf("catch it\n");
  }
  return 0
}
```

典书上的知识也要通过实证才能确保。）到底是 C++的标准没有规定，还是编译器没有遵循，只有看标准才能知道了。

笔者通过搜索了解到，C++标准 5 年更新一次，上一次更新是 2003 年，编号是 14882。而这次更新基本已经定稿，笔者能搜到的最新版本是 2011 年 4 月 5 号，编号为 N3291 的草稿。将两者下载，却面临一个问题：1350 页的文档如何查找？其实与编程一样，先猜测，觉得相关信息放在哪里比较合适，再到文中找。在笔者看来，描述异常、析构函数、对象生存期等地方都合适。通过目录，容易定位出析构函数的描述章节，在老版本 12.4 节关于析构函数的小节第 10 条中有如下英文描述：

Destructors are invoked implicitly (1) for a constructed object with static storage duration (3.7.1) at program termination(3.6.3), ***(2) for a constructed object with automatic storage duration (3.7.2) when the block in which the object is created exits (6.7)***, (3) for a constructed temporary object when the lifetime of the temporary object ends (12.2), (4)for a constructed object allocated by a new-expression (5.3.4), through use of a delete-expression (5.3.5), ***(5) in several situations due to the handling of exceptions (15.3)***. A program is ill-formed if an object of class type or array thereof is declared and the destructor for the class is not accessible at the point of the declaration. Destructors can also be invoked explicitly.

段首粗体部分“invoked implicitly”正好描述了析构函数被隐式调用的情况。这正是我们要找的情况。当发生异常而离开局部对象的作用域，如果引起了析构调用，正是隐式调用。其第(2)条的描述有点像我们需要的，而第(5)条谈到了异常，这两条都有点像。通过阅读第(2)条提及的相关章节 3.7.2，我们确定“automatic storage”指的是局部对象的存储分配。而“when the block in which the object is created exits”的意思是“当创建对象的代码块退出时”。这也与前面正常退出时在右花括号调用析构符合。相关的 6.7 节更清晰描述了，对于局部对象，何时调用析构（包括在代码块中遇到 goto、break 等跳转语句时），但没有清楚提及如果遇到异常而退出，是否调用析构。第(5)条虽然提及了异常处理，但读完了相关 15.3 节也找不到我们寻找的情况。（**似乎标准没有清晰地说明这个问题。**）

是否标准旧了？其实从 lippman 书的出版日期比标准 14882 早好几年已经能说明问题。但本着严谨的态度，在阅读了之前提及的 N3291 后，我们发现，最新的标准只是在线程方面增加了新规定（为了适应汹涌而来的并发编程大潮），其余部分完全相同。

至此，我们至少可以推定，可能 C++标准在这个问题上描述的不明确性导致了不同编译器在实现上的差异。而这个问题让 C++美丽的自动析构能力变得脆弱了，那些巧妙的技巧也让人惴惴不安。（要保证一定执行，还是用 try...finally 吧，可这又不是 C++的标准语句。）

4.5.3 隐藏的敌人：不请而至的析构和拷贝构造

C++是复杂的，它暗地里做了太多事情，如果不明就里，结果往往出人意表。比如，一个熟悉只支持基于堆构造对象语言（如 Java）的程序员很容易被 C++的基于栈的对象构造能力蒙蔽，犯下错误还茫然不知。下面的例子从一个意外的问题开始，结束于对整体资源管理的反思和优化。

一次在做实验时，一位同学拿来一段程序，如 DM4-32 所示，其中 TestClass 类定义如 DM4-31(a)

所示，仅仅多了两个成员变量 i1、i2。他不明白为什么析构函数被调用了两次，而构造函数只被调用了一次，见图 4.9。工程见“code\第 4 章\copy_constructortest”。

```
do constructor
a.i1=1 a.i2=2
do destructor
do destructor
```

图 4.9

DM4-32

```
void test(TestClass a){
    printf("a.i1=%d a.i2=%d\n", a.i1, a.i2);
}
void main(){
    TestClass a;
    a.i1 = 1;
    a.i2 = 2;
    test(a);
}
```

main 中的局部变量 a 对应一次构造和析构的调用。为什么多出来一次？而从打印结果来看，第 2 次析构调用似乎发生在 test 函数的 printf 语句执行后。要想清楚这个问题，我们将先分析结构体的情况，因为对象就是一个结构体变量，不妨先看看结构体是怎么处理的。在之前工程中添加新代码。从猜想来看，如 DM4-33 调用 test2 传递的参数不应该是 s 的地址，而是 s 的值，即在栈上分配变量 s 大小的内存，并将 s 的值复制，否则 test3(TestStruct * ps)这样明确要求传递结构体地址的代码和 test2 的方式就重复。因此增加了 test3 作为对照。

DM4-33

```
struct TestStruct {
    int i1;
    int i2;
};
void test2(TestStruct s){
    printf("s.i1=%d s.i2=%d\n", s.i1, s.i2);
};
void test3(TestStruct* ps) {
    printf("ps.i1=%d ps.i2=%d\n", ps->i1, ps->i2);
};
void main() {
    ...
    TestStruct s;
    s.i1 = 3;
    s.i2 = 4;
    test2(s);
    test3(&s);
}
```

反汇编见 DM4-34。i1 是 s 的第一个成员，所以 i1 的地址也是 s 的地址。从第 2 行看，i1 的地址就是 ebp–28h，也是 s 的地址。第 10 行调用 test2 之前有两次 push 压栈，共 8 字节，自然联想这可能是传递 s 的值。结构体 TestStruct 正是 8 字节。细看第 6～7 行就是将内存 ebp–24h 中的值压栈，从第 4 行看，ebp–24h 正是 s.i2 的地址，所以第 6～7 行就是将 s.i2 的值压栈。同理，第 8～9 行是将 s.i1 的值压栈。x86 压栈使栈顶地址减小，所以第 6～7 行压栈的 4 存储的地址正好比第 8～9 行压栈的 3 存储的值高 4 字节。这样正好是结构体 s 内存布局的栈翻版，见图 4.10。（**至于 test2 访问参数时，是如何从栈上获取到 s 的该副本，请自行分析。**）

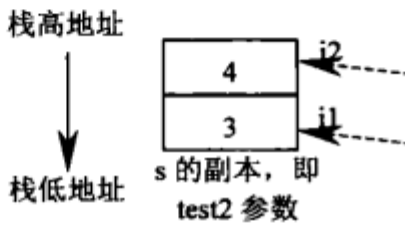


图 4.10

DM4-34

```
1   s.i1 = 3;
2   0041377a    mov   dword ptr [ebp-28h], 3
3   s.i2 = 4;
4   00413781    mov   dword ptr [ebp-24h], 4
5   test2(s);
6   00413788    mov   eax, dword ptr [ebp-24h]
7   0041378b    push  eax
8   0041378c    mov   ecx, dword ptr [ebp-28h]
9   0041378f    push  ecx
10  00413790    call  004111ea
11  00413795    add   esp, 8
12  test3(&s);
13  00413798    lea   eax, [ebp-28h]
14  0041379b    push  eax
15  0041379c    call  004111e5
16  004137a1    add   esp, 4
```

第 11 行是函数返回后清栈刚才压栈结构体耗费的 8 字节。第 13 行将 ebp–28h 存入 eax，上面分析 ebp–28h 就是 s 的地址。所以第 14 行压栈 eax，明显是传递 s 的地址。第 15 行转跳到 test3 的入口执行。（大家可自行分析 test3 利用参数访问结构体成员变量的过程。）

现在非常清楚，当函数的参数是结构体类型时，就在栈上复制被传参数的副本，是传递的值。而函数访问的也是栈上的这份副本，而非被复制的结构体本身。既然对象就是结构体变量，那么自然的推想是，DM4-32 中调用函数 test 时，也是将局部对象 a 的内容复制了一份副本到栈上，也就

是说，调用 test 使得栈上诞生了一个“新”的“对象 a 的克隆”对象。既然诞生了新对象，那么在新对象无效时自然应该调用析构函数。新对象是 test 的参数，所以其有效范围就是 test 函数。因此，在退出 test 函数时对象的析构就要被调用，这也与图 4.9 的打印结果吻合。

DM4-35

(a)

```
  a.i1 = 1;
0041375c  mov  dword ptr [ebp-18h], 1
  a.i2 = 2;
00413763  mov  dword ptr [ebp-14h], 2
  test(a);
0041376a  mov  eax, dword ptr [ebp-14h]
0041376d  push eax
0041376e  mov  ecx, dword ptr [ebp-18h]
00413771  push ecx
00413772  call  0041107d
00413777  add  esp, 8
```

(b)

```
  printf("a.i1=%d a.i2=%d\n", a.i1, a.i2);
  ...
00411c68  push  offset string "a.i1=%d a.i2=%d\n"
00411c6d  call  dword ptr [__imp__printf (4192c4h)]
00411c73  add  esp, 0ch
  ...
}
00411c7d  lea  ecx,[a]
00411c80  call  testclass::~testclass (4110ebh)
```

可是，为什么没有调用构造函数呢？在没有调用构造的情况下，如何在栈上克隆了对象 a 的一个副本？是否真的如结构体那样在栈上有一个对象 a 的副本？还是靠反汇编，见 DM4-35(a)，容易发现，它与结构体那个版本对应的 DM4-34 的第 1～11 行逻辑结构完全一样。可见，编译器就是将对象 a 的每个字段内容复制到栈上，完成了新对象创建工作。从 DM4-35(b)容易看到，test 函数在其右花括号对应的代码中调用了 TestClass 的析构。（在反汇编窗体中单击右键，选择弹出菜单中的“显示符号名”，可看到地址对应的符号，便于阅读。）

唯一的问题是，为什么在栈上克隆对象时不调用构造函数？不调用构造函数岂非很危险？如 DM4-20 的 Person 类的构造函数就分配了一块新的内存，按照当前的逻辑，只要调用函数 func1()，其原型为“func1(Person a)”，必然导致错误。因为传递参数时，克隆对象的成员变量 ID 与被克隆对象的 ID 值相同。这意味着两者都指向同一块内存区。当两个对象的析构被调用时，ID 指向的内存将被释放。后调用的那个将再次释放前面已经释放的内存。为什么在传参克隆对象时不调用构造函数？① 构造函数可能是带参数的，而克隆对象时不可能获得这些参数，自然无法调用。② 克隆，除了被克隆的对象，自然不需要其他参数。因此，从“不可能”到“不需要”都说明不应该调用构造函数。

但是如何解决上面提及的 Person 类问题？只要克隆时再给新对象分配一块内存，让 ID 指向它，那么析构就释放新内存，而非被克隆对象拥有的内存，自然不会发生两次释放同一内存块的问题。因此，专门在克隆时调用的函数产生了，它接收被克隆对象作为参数，这就是拷贝构造函数。（**原来拷贝构造是这样应需而生的，并不神秘**。）其原型就是“类名(const 类名&)”。其中，&是指针的另外一种便捷形式，叫引用。对于 Person 类，我们添加拷贝构造，见 DM4-36，其中分配新内存，并复制对象 src 的 ID 串的内容。

这时，回头看看整体资源管理的关键设计策略，在构造函数中申请资源，在析构函数中释放资源。拷贝构造函数和析构函数的调用会对此造成什么影响？以 FileStream 类为例，如果 FileStream 的对象 f（非&f）作为参数传递给函数 func(f)，在 func 返回前必然自动调用 FileStream 的析构，从而关闭文件。也就是说，如果从 func 返回，本已经打开的文件就关闭了。从 func 返回后再用对象 f 操作文件肯定失败。可仿照 Person 类，为 FileStream 类增加拷贝构造函数来解决吗？在拷贝构造函数中再打开一次文件吗？对于使用共享打开文件模式可以这样考虑，而独占模式就不允许同时打开一个文件。况且，同时打开还可能引起一些非常诡异的错误。如果在拷贝构造函数中我们无能为力，那该怎么办？其实只需禁止 FileStream 对象作为参数传递，允许传递 FileStream *，但不允许传递对象本身，这样就避免了错误。所以，将拷贝构造函数定义为私有就可达成目标，见 DM4-37 中的粗体部分。这样用 FileStream 对象作为参数传递将导致编译错误。这种禁止策略在 STL 中也用到了，

如其读文件流 ifstream 也在构造函数中打开文件，在析构中关闭，将面临 FileStream 同样的问题。DM4-38 的第 2 行无法通过编译，这样就避免了多次调用析构关闭文件的问题。另一个要注意的问题即“=”重载，见 4.8.1 节的论述。

DM4-36

```
class Person {
    char * ID;
    ...
 public:
  Person(char * aID);
  ~Person();
  Person(Person & src); {
     ID = (char *)malloc(strlen(src.ID) + 1);
     strcpy(ID, src.ID);
  }
  ...
};
```

DM4-37

```
class FileStream {
  private:
    FileStream(FileStream & src) { }
  public:
    FileStream(char * fileName, char * openMode);
    ~FileStream();
}
```

DM4-38

```
ifstream s("C:\\temp.txt");
test(s);
```

我们简单总结整体资源管理在 C++下要注意的问题：

① 用指针传递实现了整体资源管理策略的对象，不能直接用对象作为参数。

② 将拷贝构造和“=”操作符设为私有，保证第①条。

③ 可以用局部对象实现自动资源管理，但在不同编译器中，当异常发生时会产生不确定后果，无法保证析构一定被调用去释放资源。

④ 在无把握时，C++下，用对象的指针作为参数和局部变量比较简单，副作用小。

4.6 封装之强化：内外之别，亲疏之分

4.6.1 私有的诞生

在进行新内容前，将第 3 章的动态数组也封装为一个类。其方法与 4.3 节的 TTextFileInfo 封装相似，很容易就从 C 语言版变成真正的类。用它修改第 3 章的电话簿程序。工程见“code\第 4 章\DynamicArray”，类的定义见 DM4-39。其构造函数传入元素的大小。

为了方便使用，对于存储结构体 PhoneInfo 的动态数组，我们想封装为一个专门的类 PhoneInfoArray，这样用它访问元素时，不用像 C 语言版那样进行类型强制转换。如何重用已有的 DynamicArray 类？见 DM4-40，首先定义了一个 DynamicArray 指针类型的成员变量 m_array。秉承构造函数初始化资源的原则，我们在构造函数中，为 m_array 分配对象，并指定元素大小为 sizeof(struct PhoneInfo)，在析构中释放 m_array，见 DM4-41(a)。通过 DM4-41(b)所示的 getItem 和 setItem，调用者不用对获得的元素指针做“(PhoneInfo*)”强制转换，如函数 queryName

DM4-39

```
class DynamicArray {
  public:
    DynamicArray(int elementSize);
    char * ptrArray;              //数组内存首址
    int count;                    //数组个数
    int elementSize;              //元素大小
    void setArrayLength(int elementCount);
    void setArrayItem(int index, void *ptrValue);
    void* getArrayItem(int index);
};
```

DM4-40

```
class PhoneInfoArray {
  public:
    DynamicArray * m_array;
    PhoneInfoArray();
    ~PhoneInfoArray();
    void setLength(int size);
    int getLength();
    PhoneInfo * getItem(int index);
    void setItem(int index, PhoneInfo * pValue);
};
```

中的语句“pInfo = gPhoneInfoList.getItem(i);”。第 3 章中 C 语言版的强制转换为“(PhoneInfo *) getArrayItem(index, gPhoneInfoList)”，可见封装后使用非常简便。

DM4-41

(a)

```
PhoneInfoArray::PhoneInfoArray(){
    m_array = new DynamicArray(sizeof(PhoneInfo));
}
PhoneInfoArray::~PhoneInfoArray(){
    delete m_array;
}
```

(b)

```
PhoneInfo * PhoneInfoArray::getItem(int index) {
    return (PhoneInfo *) m_array->getArrayItem(index);
}
void PhoneInfoArray::setItem(int index, PhoneInfo * pValue) {
    m_array->setArrayItem(index, pValue);
}
```

下面来看如果 FileStream 和 DynamicArray 的一些成员变量被意外修改的状况。在 DM4-42(a)中，如果第 2 行将 fp 设为 NULL，意味着第 1 行执行构造函数打开文件而赋予 fp 的有效值被清零，之后的任何操作都是错的。在 DM4-42(b)中，第 2 行设定了动态数组长度 12 后，其成员变量 count 的值就是 12。第 3 行却将其赋为 14，必然导致将来访问越界。

DM4-42

(a)

```
1  FileStream f("C"\\temp.txt", _OpenRead);
2  f.fp = NULL;
3  f.read….
```

(b)

```
1  DynamicArray A(sizeof(int));
2  A.setArrayLength(12);
3  A.count =14;  ····· //error.
```

(c)

```
class FileStream{
  private:
    FILE *fp;
  public:
    ….
};
```

(d)

```
class DynamicArray {
  private:
    char * ptrArray;
    int count;
    int elementSize;
  public:
    …
};
```

为了解决该问题，面向对象模仿了现实中的限制方式定义了私有和公有两种方式。只需要将以上两个类的成员变量都放入私有域，类自身成员函数以外的代码就不能访问了，见 DM4-42(c)和(d)。这时，DM4-42(a)和(b)编译将报错。如果无法决定成员变量是公有还是私有，可全放入私有域，需要时再移入公有即可。其实 C++如果不标记公有、私有，默认是私有，也表明了这种思想。毕竟以“自己操作自己数据”的原则，先定义为私有也不会错到哪里。

4.6.2 私有？阻止不了我

公有、私有改变什么吗了？似乎是外部就不能访问私有变量。我们还是思考更基本的问题，有了公有、私有后，对象模型变了吗？如果公有、私有真地对程序运行产生了什么影响，那么一定会影响到对象模型，或影响到访问代码。编译器真的可以让一段内存只对某些代码可访问吗？我们学过内存访问控制机制，最多是以页为单位设定只读、可写等。一个对象往往只是几十字节的小块而已，是没法只针对几十字节设定属性的。再深入分析，对象本质就是一个结构体变量，获得首址求偏移，就应该可以访问其任意成员变量。似乎只要拿到对象的地址，我们应该可以直接修改对象相关内存区的。(**看穿对象的华丽外衣，它就是一块内存而已！**)

下面我们动手，设计如下实验来验证刚才的推测，见 DM4-43，工程见“code\第 4 章\directchangeprivatemember”。

对象就是去掉成员函数后的结构体，所以 MyTest 的对象共 8 字节。成员变量 b 在离首部 4 字节的地方。DM4-43 中，粗体标注的代码获取对象 o 的地址后，将其强制为 int *，然后加 1，因为 int *加 1 就是加 4 字节，所以加完后的值正是成员变量 b 的地址。对其赋值 3，所以之后打印将打出“b is 3”。这证明只要拿到对象地址，私有限制是没有用的。

为了进一步证明确实私有访问没有改变对象模型，我们只需像 4.1.6 节那样用反汇编比较添加 private 和注释 private 的指令，就可发现两者完全一样（此处不再给出反汇编清单）。说明私有、公

有访问并没有影响到对象模型和代码。

那么，面向对象语言添加公有、私有访问控制岂非骗人？仔细想想，如果像 DM4-42(a)和(b)那样以对象语法去访问，就无法编译通过了。也就是说，这种访问控制是编译期的保护：违反了原则，连编译都通不过，自然无法生成执行程序，也就保护了对象。

DM4-43

```
class MyTest {
  private:
    int a;
    int b;
  public:
    void setB(int v) {
      b = v;
    }
    int getB() {
      return b;
    }
};
void main() {
    MyTest o;
    o.setB(2);
    printf("b is %d\n", o.getB());
    *((int *)&o + 1) = 3;
    printf("b is %d\n", o.getB());
}
```

了解访问控制下的对象模型，知道外部能直接访问私有成员仅仅是满足我们知其究里的好奇吗？这些特殊的知识，在特殊的场合往往能发挥意想不到的效果。曾经做一个工程时，需要判断画板上的多个位图对象是否指向同一幅位图。位图类属于 VCL 系统库，翻译成 C++ 语言，定义如下：

```
class TBitmap {
  private:
    ...
    Map* _pMap;
    ...
}
```

_pMap 是真正指向位图缓存的指针。TBitmap 实现了优化，当多个对象打开同一幅图时，它们的成员变量_pMap 指向同一个缓存。当时不愿意麻烦地通过分析位图格式来判断是否两个对象指向同一幅图。只要能获得成员变量_pMap，并比较这个指针是否相同即可，相同就说明是同一幅图。可 TBitmap 没有公有方法获得_pMap，修改它又不太现实。这时对象模型就有用武之地了。只需算清楚_pMap 离对象头部的偏移量，然后获得对象的地址，并加上_pMap 的偏移量，即可读取到_pMap 的值：_pMap 的地址 =(char *)(&位图对象) + _pMap 的偏移字节数。

当然，该方法存在危险，不是正道。如果将来系统库在定义 TBitmap 时将_pMap 定义顺序变了，之前计算的_pMap 的偏移量就错了，代码也就错了。但考虑该类的稳定性和这个捷径惊人的简单性，当时还是采用了。

又如，有一次逆向分析 Delphi 撰写的一个程序，当中有一个定时器触发的代码非常影响调试跟踪。为了禁止它，笔者分析了 Delphi 的 TTimer 对象的内存布局，计算了控制定时器运行的 FEnabled 属性的偏移量，用调试器直接修改内存将其置为 0（代表 false 不运行）而令其停止。

4.6.3 理解继承的机制（1）：模型

面向对象从一开始就在模仿自然界的机制，软件重用是其重要目标。对象很好体现了封装理念。自然界中还存在一种很好的重用模式，就是儿子可通过继承获得父亲的能力。模仿这种关系的结果就产生了类的“继承”机制。如 4.6.1 节的 PoneInfoArray 类可以看成是 DynamicArray 类的“儿子”。如何体现这种“继承”关系？最关键的特点是父类的公有函数，子类对象不用自己实现就具有了。

为了实现这个目标，子类的对象从首部地址开始的内存布局必须与父类对象一样。举例来说，如果父类有两个 int 成员变量 a、b，共 8 字节，子类也必须如此。因为父类的函数会按照父类对象的布局去访问成员变量，如果子类在对象前半部不具有相同的布局，那么父类函数必然发生错误。

如图 4.11 所示，子类对象从开始的 8 字节和父类具有相同的内存布局，前 4 字节为 a，后 4 字节为 b。父类函数 setB 将完成对成员变量 b 的赋值。函数的第一步是获得隐藏参数 this，它指向了父类、子类两个对象的首址。第二步，用首址加 4 字节偏移获得 b 的地址，因为父类、子类对象开

始部分的布局相同，均在偏 4 字节处为 b 的地址，所以这一步对父类、子类对象均正确。第三步，将 v 赋给 b。由此可见，只要子类对象在首地址开始拥有和父类对象相同的内存布局，父类函数即可成功操作子类对象。

图 4.11

其次，子类之所以是子类，一定有自己的行为。因为自己操作自己的数据，所以有自己的行为，也往往意味着有自己的数据（即成员变量）。这又如何实现？只要将子类自己的成员变量放到不影响父类函数操作的地方就可以了，如紧邻父类的数据，即图 4.11 中子类对象成员变量 b 后那些“...”代表的位置。父类的函数只会操作父类所占据的前 8 字节，不会操作第 9 字节开始的内存，所以子类成员变量放在第 9 字节开始的地方不会影响父类的功能，也不会受父类函数影响。其原理见图 4.12，子类的函数 setC 对子类成员变量 c 进行设定。

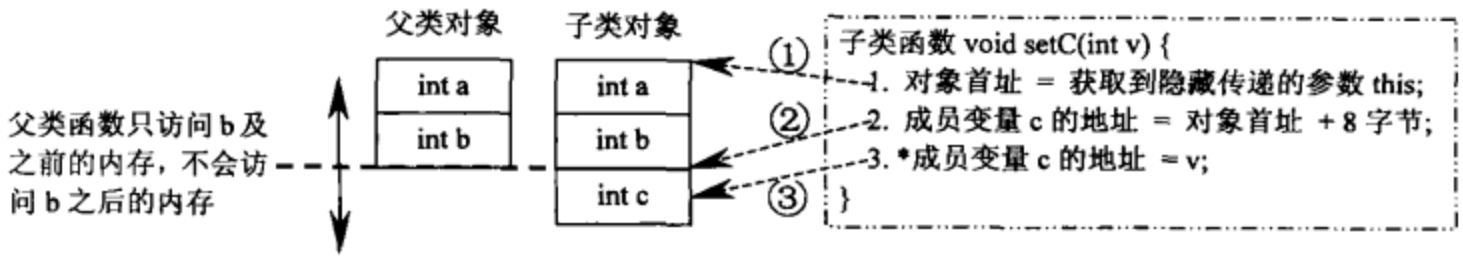


图 4.12

这种“继承”的思路可以远溯到结构化编程中。就像在生产级代码中，经常看到的链表数据结构总是将 next 成员放在起始位置。例如，第 2 章的图 2.25 所示的结构化异常由两个成员组成，第一个就是 next。Linux 内核中的 sk_buff 结构体是一个双向链表，其开始的两个成员就是双向链节点的 next 和 prev（特别注意其一开始的注释“These two members must be first”，特别强调必须在头部），定义见 DM4-44(a)。

为什么链表方式的结构体都要将 next 放在结构体的开始处，而不是放在中间或者最后呢？其实和我们的“继承”思路完全一样，可以直接将任何满足开始字段为 next 的结构体的首地址传递给通用的链表操作函数，从而完成链表的添加和删除，而不用为每个不同的结构体编写相同的函数。这就像重用父类的函数一样。DM4-44(b)定义了通用的 Node 结构体，其添加操作由 DM4-44(c)的 addNode 函数完成。DM4-44(a)和(b)分别定义了两个链表结构 sk_buff 和 MyNode，它们的第一个成员都是 next，似乎类型不同，但因为是指针，所以都只占 4 字节，因此实际内存布局与 Node 相同，即前 4 字节为指向下一个元素的地址。这样，不论添加 MyNode 节点还是 sk_buff 节点，均可重用 addNode 函数完成，见 DM4-44(d)，需要做的仅仅是将结构体的首址强制转换为(struct Node *)并传入即可。之所以这样做能成功，与前面分析继承可能的机制是一样的。因为 addNode 操作结构体的 next 属性，其逻辑是操作结构体首部的 4 字节。sk_buff 和 MyNode 的前 4 字节都是 next，所以成功。不论什么结构体，只要前 4 字节是 next 指针，调用 addNode 都能成功。这就是为什么定义支持链表操作的结构体时，要将链表的公用属性 next 放在头部的原因。反过来，这个例子从结构化编程角

度证明了我们对“继承”机制设想的可行性，因为两者的原理完全一样。

DM4-44

(a)

```
struct sk_buff {
   //These two members must be first
   struct sk_buff   *next;
   struct sk_buff   *prev;
   struct sock      *sk;
   ...
}
```

(b)

```
struct Node {
   struct Node * next;
};
struct MyNode {
   struct MyNode *next;
   int a;
}
```

DM4-44

(c)

```
void addNode(struct Node **root, struct Node * n) {
   if (*root != NULL) {
      n->next = *root;
      *root = n;
   }
}
```

(d)

```
MyNode * pm, *myRoot;
...
addNode((struct Node**)&myRoot, (struct Node*)pm);
sk_buf* ps, *skRoot;
...
addNode((struct Node **)&skRoot, (struct Node *)ps);
```

现在就需要实证继承机制了。将图 4.12 描述的结构转换成代码，见 DM4-45。首先从打印结果看，类 Parent 对象的大小为 8 字节，子类 Child 对象的大小为 12 字节，符合我们的推测。

接下来用反汇编验证函数调用是否符合前面的推定，见 DM4-46。DM4-46(a)的第 4 行和第 8 行分别对应父类对象 p 和子类对象 c 来调用父类函数 setB。从 call 指令看，它们确实都在调用 0x00401000 为入口地址的函数，父类和子类重用了一个函数。第 3 行中，代表父类调用时，将父类对象 p 的首址 ebp–8 通过 ecx 传递给了 setB。第 7 行中，代表子类调用时，将子类对象 c 的首址 ebp–14h 通过 ecx 传递给了 setB。

再来看 DM4-46(b)，即父类的 setB。第 1 行从 ecx 获得对象首址，正好对应我们推测的图 4.11 的①。第 5 行指令中的 eax+4 获得成员变量 b 的地址（因为它离首部偏移为 4 字节，且此时对象首址在 eax 中），这对应图 4.11 的②。第 5 行的 mov 指令将参数 v 赋值给 b，对应于图 4.11 的③。不论传递的是父类对象 p 还是子类对象 c 的首址，setB 都是这样做的，因为两个对象起始 8 字节的布局一样，所以 setB 的操作是正确的。

DM4-45

(a)

```
class Parent{
   public:
      int a;
      int b;
      void setB(int v) {
         b = v;
      }
};
```

(b)

```
class Child : public Parent {
   public:
      int c;
      void setC(int v) {
         c = v;
      }
};
```

(c)

```
void main() {
   Parent p;
   Child c;
   p.setB(1);
   c.setB(2);
   c.setC(3);
   printf("p size=%d, c size=%d\n", sizeof(p), sizeof(c));
}
```

接下来分析调用子类函数 setC 的过程，从 DM4-46a 可知，第 11 行传递了子类对象 c 的首址 ebp–14h 给 ecx。第 12 行调用 setC。DM4-46(c)是 setC 的反汇编，第 1 行说明它从 ecx 获得对象首址，对应于推测的图 4.12 的①。第 5 行的 eax+8（因为成员变量 c 离对象首部偏移 8 字节，且此时对象首址在 eax 中）对应于图 4.12 的②。第 5 行整个指令完成对 c 的赋值，对应图 4.12 的③。至此，我们已经完全证明了对象继承的机制：

子类的对象模型就是在父类对象后添加上自己定义的成员变量，内存布局顺序和成员变量定义的顺序一致。对象大小依然和成员函数的多少无关。

DM4-46

(a)

```
1  p.setB(1);
2    00401046    push  1
3    00401048    lea   ecx, [ebp-8]
4    0040104b    call  00401000
5  c.setB(2);
6    00401050    push  2
7    00401052    lea   ecx, [ebp-14h]
8    00401055    call  00401000
9  c.setC(3);
10   0040105a    push  3
11   0040105c    lea   ecx, [ebp-14h]
12   0040105f    call  00401020
```

(b)

```
void setB(int v) {
    ...
1  00401004      mov  dword ptr [ebp-4], ecx
2  b = v;
3  00401007      mov  eax, dword ptr [ebp-4]
4  0040100a      mov  ecx, dword ptr [ebp+8]
5  0040100d      mov  dword ptr [eax+4], ecx
```

(c)

```
void setC(int v) {
    ...
1  00401024    mov  dword ptr [ebp-4], ecx
2  c = v;
3  00401027    mov  eax, dword ptr [ebp-4]
4  0040102a    mov  ecx, dword ptr [ebp+8]
5  0040102d    mov  dword ptr [eax+8], ecx
```

(d)

```
Child c;
0040107e        lea   ecx,[c]
00401081        call  Child::Child(401030h)
...
Child() {
    ...
    00401034    mov  dword ptr [ebp-4], ecx
    00401037    mov  ecx, dword ptr [this]
    0040103a    call  Parent::Parent(401000h)
```

曾经有位同学提了一个问题：“子类对象怎么能拥有父类对象的私有变量呢？不是私有的吗？”。这反映了纯粹用形象比喻去理解技术机制的缺陷：“会发生似是而非的错误”。其实从正面讲，父类私有变量的含义是：子类函数用面向对象语法访问该变量时将报编译错误，并非这部分内存“不存在”。从反面讲，如果真的不存在，将会发生错误。因为从子类对象 c 上调用父类函数 setB、c.setB(...)，调用的确实就是父类函数 setB，见 DM4-46(b)。执行其第 5 行时，必然向对象首部地址偏移 4 字节的内存赋值。如果这块内存因为是“父类私有的”而不存在，那么这条 mov 指令岂非要出错？这个问题再次提醒我们：除了用比喻快速理解技术机制外，还要从代码层面清晰理解它。

另一问题是，在继承机制下，构造函数和析构函数的调用顺序是怎样的？自然的想法是，有了父才有子，就应该先调用父类构造函数，再调子类的构造函数。析构函数的顺序反之亦然。C++确实按照这种思路实现，不过只是保证了逻辑上这个顺序，而非实际执行顺序（并非所有语言强制保证执行顺序，可参见 5.4.2 节最后的提示）。也就是说，其实 C++是先调用子类构造函数，但在子类构造函数开始处悄悄插入了调用父类构造函数的指令，从而保证效果上是父类先调用。

我们在 DM4-45(a)和(b)的两个类添加构造函数，DM4-45(c)和 child 的反汇编见 DM4-46(d)。局部变量 c 的定义直接导致调用子类的构造函数 Child::Child()。Child 的构造中一开始就调用父类构造函数 Parent::Parent()，导致父类构造“先于”子类构造中“程序员编写的”代码执行。析构函数则最后自动调用父类析构函数。

4.6.4 理解继承的机制（2）：在 C 语言中“玩”继承

4.6.3 节通过链表例子已隐约看到继承机制在 C 语言中的影子。现在既然理解了面向对象的继承机制，是否能用 C 语言来模拟？一方面，通过构建性工作能更清晰理解；另一方面，真正在 C 语言下工作时，我们也能享受到对象及继承的好处。（**事实上，笔者在工程实践中，遇到必须用 C 语言编写时，就用模拟的继承机制享受了面向对象的部分优势。**）

来看一个简单的继承例子，以 DM4-45 为例，如果转换成结构体版本如 DM4-47 所示，工程见“code\第 4 章\structinherit1”。

DM4-47(a)的第 1～5 行将类 Parent 转换为一个 struct，成员变量不变，只是在第 4 行增加了一个函数指针变量 setB，用于保留类 Parent 的成员函数 setB 的入口地址。第 6～10 行是结构体版本的子类 Child。继承的关键就是第 7 行的成员变量 super，其类型为 Parent。super 变量必须放在第一个，这是保证继承模型的关键，对象起始的内存布局为父类的布局。这里没有直接复制 Parent 的定义，是因为复制就是重复，“重复即罪恶”。如果修改了 Parent 的定义，而 Child 的定义是复制 Parent 的，就必须修改 Child 的定义。现在如此定义 super，我们的 Child 就不用修改了。第 8～9 行定义了自己的成员变量 c 和 setC。Child 和 Parent 的内存布局见图 4.13。

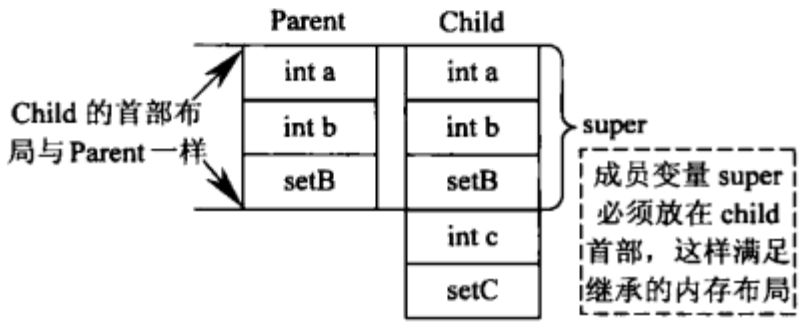


图 4.13

DM4-47(b)给出了 initParent 和 initChild，相当于对象的构造函数，对结构体初始化。注意，initChild 先要调用 initParent，相当于子类构造函数总是先调用父类的构造，先初始化属于父类部分的内存（将成员变量 setB 赋值为函数 setB 的入口地址），再将函数 setC 的指针赋给结构体的成员变量 setC。

DM4-47(c)模拟了对象版代码 DM4-45(c)的逻辑。第 5 行将结构体 p 初始化，第 6 行将结构体 c 初始化。第 8～9 行展示了结构体 c 如何调用“父类”结构体 Parent 的成员函数 setB。第 8 行将 c 的指针强转为 Parent *。这一步非常关键，从图 4.13 也可以看出，c 的前 12 字节就是 Parent 的布局，因此强制转换是合理的。这样第 9 行就可以用 Parent 的身份调用其成员函数 setB。第 10 行是调用 Child 自己的函数 setC。从第 11 行打印结果可知，一切 ok。

下面我们来看以上过程中哪些比较麻烦和重复，进而找到一些方法来减轻工作量。首先，为了定义继承，我们必须像见 DM4-47(a)的第 7 行那样，每次手动地定义 super 变量。如不仔细分析，难以一眼看出继承关系。能否模拟 C++继承语法“class 子类名 : 父类名”？

DM4-47

(a)

```
typedef void(*PtrParentIntSetter)(struct Parent *self, int v);
typedef void(*PtrChildIntSetter)(struct Child *self, int v);
1   struct Parent {
2       int a;
3       int b;
4       PtrParentIntSetter setB;
5   };
6   struct Child {
7       struct Parent super;
8       int c;
9       PtrChildIntSetter setC;
10  };
```

(b)

```
void setB(struct Parent *self, int v) {
    self->b = v;
}
void setC(struct Child * self, int v) {
    self->c = v;
}
void initParent(struct Parent * obj) {
    obj->setB =setB;
}
void initChild(struct Child * obj) {
    initParent((struct Parent *)obj);
    obj->setC = setC;
}
```

(c)

```
1 void main() {
2     struct Parent p;
3     struct Child c;
4     struct Parent *pp;
5     initParent(&p);
6     initChild(&c);
7     p.setB(&p, 1);
8     pp = (Parent *)(&c);
9     pp->setB(pp, 2);
10    c.setC(&c, 3);
11    printf("p.b=%d, c.b=%d c.c=%d₩n", p.b, c.super.b, c.c);
}
```

C 语言中有一种叫宏函数的机制，它是一个宏，但可以接收参数，如 Max 宏“#define Max(a, b) a > b ? a : b“。如果有 Max(3, 4)，则会被替换成“3 > 4 ? 3 : 4”形式的语句，而不是像真正的函数产生 call 指令调用。宏因为只能是一行，为了在格式上分成多行，必须用“\”放在行尾代表本行没结束，下面继续，如：

```
# define sum(x, y) x + \
                     y
```

下面给出了用宏函数模拟的方法，见 DM4-48(a)。第 1 行将 struct 定义为 CLASS。第 2 行定义了宏函数 EXTENDS 代表继承，参数传入基类的名字。第 3 行自动在子类结构体定义的第一行添加了成员变量 super，super 类型为传入的基类名 super_class。这样自动完成了基类成员 super 的定义，简化了操作。用这两个宏定义的 Child 类如第 4～7 行所示。CLASS Child EXTENDS(Parent)很神似地模拟了 class Child : public Parent，能一眼看出继承关系。我们不用手动定义 super 变量，靠 EXTENDS 完成，只用定义第 5～6 行属于 Child 的成员。DM4-48(a)的第 4～7 行完全等价于 DM4-47(a)的第 6～10 行。

DM4-48(a)

```
1   #define CLASS struct
2   #define EXTENDS(super_class) {
3       struct super_class super;
        ...
4       CLASS Child EXTENDS(Parent);
5       int c;
6       PtrChildIntSetter setC;
7   };
```

下面需要实现 new 和 delete 两个操作符了。因为它们将调用构造和析构，我们约定，所有的初始化函数（即类的构造函数）命名规范为“类名_init”，如 Parent 的初始化函数就是 Parent_init，Child 的初始化函数为 Child_init。析构函数命名为“类名_dtor”，则 Parent 的析构函数是 Parent_dtor。

new 操作符的模拟为 DM4-48(b)中的宏函数 CLASS_NEW，参数为要生成实例的类名。如要构造一个 Parent 的对象 CLASS_NEW(Parent)。第 4 行分配了对象，大小为 sizeof(struct class_name)。第 9 行在分配成功后调用类的构造函数初始化。“##”在宏函数中的意思是将其左右两边的字符串连成一个串。如用 CLASS_NEW(Parent)调用时，字符串"!class_name##_init(ret)"是"!Parent_init(ret)"，意味着调用 Parent 的构造函数 Parent_init。为了方便定义一个类的对象创建函数，我们定义了宏函数 CLASS_NEW_FUNC_DEFINE，见 DM4-48(c)，它调用宏 CLASS_NEW 创建对象，展开是一个函数定义。如 CLASS_NEW_FUNC_DEFINE(Child)的结果就是：

```
struct Child * Child_new() {
    CLASS_NEW(Child);
}
```

DM4-48(b)

```
1   #define CLASS_INSTANCE(class_name) (struct class_name *)MALLOC(sizeof(struct class_name))
2   #define CLASS_NEW(class_name)   \
3   {\
4       struct class_name * ret = CLASS_INSTANCE(class_name); \
5       if (ret == NULL) \
6       {    \
7           return NULL; \
8       }    \
9       if (!class_name##_init(ret)) return NULL; \
10      return ret; \
11  }
```

DM4-48(c)

```
#define CLASS_NEW_FUNC_DEFINE(class_name) struct class_name *   class_name##_new(){ \
    CLASS_NEW(class_name){ } \
};
```

delete 操作符的工作是先调用被删除对象的析构，然后调用 free()释放内存。与 CLASS_NEW 不同，delete 等价的宏函数 OBJ_DELETE(self)接收的参数是要被删除对象的指针，而不是类名，所以无法用“类名##_init”这样的语句得到析构函数的名字。这时为所有类定义一个基类 Object，该类只有一个成员变量 dtor，该指针指向析构函数。我们可用“((struct Object *)self)->dtor(self)”的形式调用析构函数了。最终，OBJ_DELETE 定义见 DM4-48(d)。

之后，我们发现 DM4-47(b)的 initChild 需要在一开始手动调用基类的构造函数 initParent。这是

所有子类初始化函数必然做的，能否将这个动作简化？如通过一个宏函数。另外，我们想吸取 Java、C#等语言的习惯，即在初始化时，将对象的内存开始自动清零，这往往能避免很多忘记设定初值的错误。是否可以在宏函数中将这个逻辑也做了呢？

DM4-48(d)

```
#define OBJ_DELETE(self){ \
    if (self != NULL) \
    {\
        ((struct Object *)self)->dtor(self); \
        free(self); \
    }\
}
```

见 DM4-48(e)，宏函数 SUPERCLASS_INIT 完成了该初始化工作。参数 super_class 为基类的名字，self_class 为子类的名字。假定我们如下使用宏函数 SUPERCLASS_INIT(self, Parent, Child)，则意味着 super_class 为 Parent，self_class 为 Child。第 3 行最终的结果就是“r_xxx = Parent_init((struct Parent *)self);”，它调用了 Parent 的初始化函数。第 5 行则完成了将对象相关字节清零的工作。(char *)(self)+sizeof(self->super)使得被清零的地址从对象首址偏移父类大小（sizeof(self->super)）处开始。清零的内存长度=子类大小–基类大小。

DM4-48(e)

```
1   #define  SUPERCLASS_INIT(self, super_class, self_class) \
2   BOOL r_xxxx;\
3   r_xxxx = super_class##_init((struct super_class *)self); \
4   if (!r_xxxx) return FALSE;\
5       memset((char *)(self) + sizeof(self->super), 0, sizeof(struct self_class) - sizeof(self->super));\
6   ((struct Object *)self)->dtor = self_class##_dtor;
```

所有类的基类 Object 有一个成员变量 dtor，相当于析构函数的指针，该变量应指向子类的析构函数。第 6 行完成了该工作，self_class##_dtor 可计算出子类的析构函数名。相应地，为了方便调用父类析构函数，定义了宏函数 SUPERCLASS_DTOR 调用父类的析构。

最终我们给出 Child 的实现和 main()函数，见 DM4-49。工程见“code\第 4 章\structinherit2\structinherit2”。注意，构造函数应该在一开始调用父类构造，见 DM4-49(a)，Child_init 函数即调用了 SUPERCLASS_INIT。析构函数在最后调用父类的析构，如 Child_dtor 在最后调用了 SUPERCLASS_DTOR。这样就与面向对象的构造析构机制一样了。

DM4-49

(a)

```
CLASS Child EXTENDS(Parent) {
    int c;
    PtrChildIntSetter setC;
};
CLASS_NEW_FUNC_DEFINE(Child);
void Child_dtor(void * self) {
    //do the staff of this class, let the
    // SUPERCLASS_DTOR call at the last
    SUPERCLASS_DTOR(self, Parent);
}
void Child_setC(struct Child *self, int v) {
    self->c = v;
}
BOOL Child_init(struct Child * self) {
    SUPERCLASS_INIT(self, Parent, Child);
    self->setC = Child_setC;
    return TRUE;
}
```

(b)

```
void main() {
    CLASS Child * pc;
    CLASS Parent * pp;
    CLASS Parent * p2;
    pc = Child_new();
    pp = Parent_new();
    pp->setB(pp, 1);
    p2 = (CLASS Parent *) pc;
    p2->setB(p2, 2);
    pc->setC(pc, 3);
    printf("pp->b=%d, pc->b=%d pc->c=%d\n", \
                        pp->b, pc->super.b, pc->c);
    OBJ_DELETE(pc);
    OBJ_DELETE(pp);
}
```

4.6.5 保护的诞生

有了继承机制，再回头看 4.6.1 节中类 PhoneInfoArray 与 DynamicArray 的关系，我们就有新观点了。之前，我们让 DynamicArray 成为 PhoneInfoArray 的成员变量 m_array，见 DM4-40。其中 setLength 并没有做有意义的事，只是调用成员变量 m_array->setArrayLength...。这种方式称为分派，就是 PhoneInfoArray 的函数只将任务分派给其成员（m_array）的函数了，像一个中介。如果这样的分派多了，编写 PhoneInfoArray 的函数是一种无聊的拖累。可是，如果让 PhoneInfoArray 成为 DynamicArray 的子类，我们就不用编写分派函数 setLength 了，可直接调用父类的 setArrayLength，成为它的子类，拥有它的能力。（**多好的重用方式！**）

下面我们重构 PhoneInfoArray 类，见 DM4-50。DM4-50(a)为类的定义，DM4-50(b)为构造函数。在构造函数中，将父类成员变量 elementSize 赋为结构体 PhoneInfo 的大小即可。可是，问题发生了，因为 elementSize 是父类的私有变量，子类无法访问，我们又不能将它定为公有，否则问题更大。其实，社会中有一种现象，一个家族中可以拥有家族内部才能知晓的信息。子类和父类相当于一个家族，elementSize 就是家族内部成员才能访问的变量。如此，我们需要一种新的访问控制范围，即 protected，将 elementSize 设为 protected 即可，新的 DynamicArray 定义见 DM4-50(c)所示，注意粗体部分。

DM4-50

(a)

```
class PhoneInfoArray : public DynamicArray{
  public:
    PhoneInfoArray();
    PhoneInfo * getItem(int index);
    void setItem(int index, PhoneInfo * pValue);
};
```

(b)

```
PhoneInfoArray::PhoneInfoArray() {
    elementSize = sizeof(PhoneInfo);
};
```

(c)

```
class DynamicArray {
  private:
    char * ptrArray;
    int count;
  protected:
      int elementSize;
  public:
    ....
};
```

4.7 “变”的烦恼与出路：创造虚函数

本节从软件重用的角度经历软件扩展性的困苦，并一路见证虚函数的诞生，然后讨论虚函数相关问题。

4.7.1 “三变”之苦：格式化字符串

我们从一个简单程序开始，其功能是将遇到的单词全部变成大写。这里要用到函数 strupr()，它将传入字符串全部变成大写，该函数定义在 string.h 中，关键代码见 DM4-51(a)，工程见“code\第 4 章\Format_1opt”。基本思路是将读到的字符串用 strupr()转换成大写并打印。

新的任务又来了：用户可以选择大写和小写两种方式，见 DM4-51(b)，工程见“code\第 4 章\Format_2opts”。于是我们设计了两个宏 FILE_FORMAT_UPPERCASE 和 FILE_FORMAT_LOWERCASE，代表大小写两种格式化选择。其中将字符串变成小写通过调用 strlwr()完成。对比 DM4-51(a)和(b)，其中粗体部分是修改的代码。

新的变化又来了：再提供一种选择，一个单词的第一个字母大写，其余小写。当然第一件事是实现这个功能，由自定义函数 strFirstCharUpperCase 完成。如何选择格式化的逻辑见 DM4-51(c)，工程见“code\第 4 章\Format_3opts”。

我们定义一个新宏 FILE_FORMAT_FIRST_UPPERCASE，代表首字母大写的格式化方式，仅在 switch 语句添加一个分支，如 DM4-51(c)中粗体所示。

到目前为止，似乎一切都很自然，这种改动从表面上看也是必须的。让我们先想象一个场景。在组装计算机时，可以选择不同显卡插到主板上。显卡能提供显示功能，与一个函数能提供一定逻辑的复用相似。主板使用显卡工作，就像调用函数的代码。（**看出问题了吗？**）如果我们认为 DM4-51(c)中的 formatTheFile 函数就是“主板”，它调用了三个不同的“显卡”，就是三个格式化字符串的函数 strupr()、strlwr()和 strFirstCharUpperCase()。为什么装配计算机时，可以随便换显卡，而不改动主板，但在代码中调用不同函数时就必须不断修改调用者（主板）的代码（电路）？DM4-51 中的粗体部分就是不断变化的部分。如果我们支持第 4 种格式化方式，就要提供新的格式化函数，DM4-51(c)中的 switch 语句必然增加新分支。

可见，现有方式就相当于使用不同的显卡要不断修改主板电路。（**这应该是个大问题了。**）主板和显卡的配合方式好在哪里？为什么要模仿？因为它灵活，专业术语就是“可扩展性好”。

下面做一个理想实验，就更能发现这种模式的好处：主板相当于显卡插在机箱上，机箱相当于显卡插在机柜里，机柜相当于显卡插在房子中，房子相当于显卡插在地球上……如果更换显卡要修改主板，而主板和机箱的关系就是显卡和主板的关系，那么主板被迫改了，是否就应改机箱呢？如此下去，简直就是多米诺骨牌（得改地球了）。这在软件开发中是必须防止的。如果一个模块修改了，要被迫修改其他使用它的模块，如此下去，工程很难维护。能够通过防止调用方代码被迫修改而控制这个多米诺的坍塌吗？从之前的程序功能添加和演化，我们发现在软件的世界里“只有变化才是永恒不变的主题”。出错要改代码，修改算法要改代码，增加功能要改代码，改变存储方式要改代码……在连绵不绝的代码变化中，我们能找到一种惬意的生存方式吗？

其实，主板和显卡的比喻就给我们了出路，我们只要模拟它们的配合，就能找到出路。简单地说，就是在某种情况下，被调用者（显卡）变化时，调用者代码（主板电路）不变。这是什么？即

DM4-51

(a)

```
void FormatFile::formatTheFile(char * fileName) {
    char ln[1024];
    TTextFileInfo file(fileName, "r");
    ...
    while (!file.endOf()){
        if (file.ReadLn(ln) > 0){
            strupr(ln);
            printf("%s\n", ln);
        }
    }
    file.Close();
}
```

(b)

```
void FormatFile::formatTheFile(char * fileName, int formatType){
    char ln[1024];
    TTextFileInfo file(fileName, "r");
    ...
    while (!file.endOf()) {
        if (file.ReadLn(ln) > 0) {
            switch (formatType) {
                case FILE_FORMAT_UPPERCASE:
                    strupr(ln);    break;
                case FILE_FORMAT_LOWERCASE:
                    strlwr(ln);   break;
                default:
                    printf("error formattype\n");
            }
            printf("%s\n", ln);
        }
    }
    file.Close();
}
```

(c)

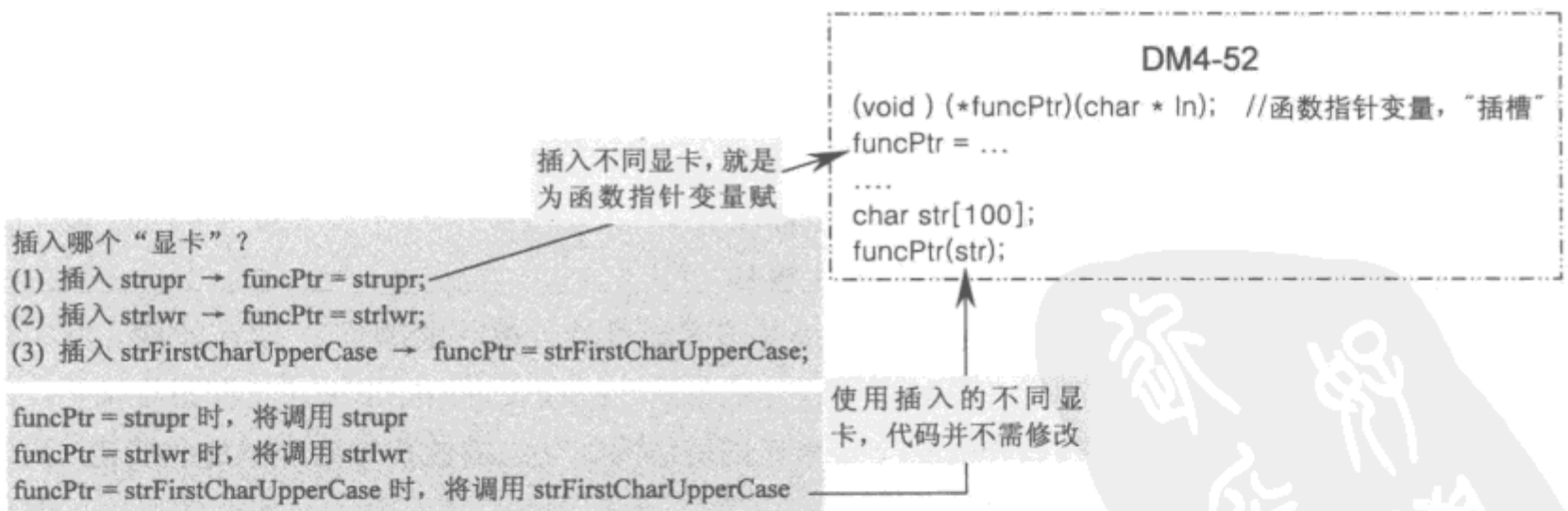
```
void FormatFile::formatTheFile(char * fileName, int formatType) {
    ...
    switch (formatType) {
        case FILE_FORMAT_UPPERCASE:
            strupr(ln);    break;
        case FILE_FORMAT_LOWERCASE:
            strlwr(ln);    break;
        case FILE_FORMAT_FIRST_UPPERCASE:
            strFirstCharUpperCase(ln);    break;
        default:
            printf("error formattype\n");
    }
    ...
}
```

术语“模块化”。硬件中非常典型，在软件中就是插件。比如一个媒体播放器，假定一种媒体格式的解码对应一个函数，就像前面的字符串格式化一样。如果要解码多种格式，就会有 DM4-51(c)中的 switch 分支调用不同解码函数。如果按照这种风格做程序，当加一个新媒体格式时，就需要增加新的分支，然后重新编译主程序。有了插件风格，主程序会自动下载新解码器插件，在不修改主程序的情况下，我们就能看新格式的电影了。这与“主板-显卡”的模式相似。这种方式有一个经典的原则——“开闭原则”：软件模块对修改闭，对添加新代码开，也就是说，增加新功能时，增加新代码，但不修改老代码。操作系统驱动程序就是一个典型例子。安装了新显卡，只需安装新驱动程序，并不需修改调用这些驱动程序的操作系统本身。否则，增加一个新驱动就要重编译操作系统，简直是灾难。

用 switch 或 if-else 的组合，就是典型的妨碍“主板-显卡”工作方式的代码。因为，每增加新的可用“显卡”类型时，我们必须修改调用者代码，使得插件风格无法实现，也违反了“开闭原则”。

4.7.2 函数指针，请带我走出不断修改的泥潭

让我们先审视“主板-显卡”模式，从中找出借鉴的思想。显卡为什么可随便换呢？因为所有显卡都有相同引脚，从数量到功能，所以都能插入到主板上。如果引脚数或引脚功能不对，是不可能被主板接受的。如果显卡被比喻成函数，引脚像函数中的什么？引脚规范就是与外界交互的边界，interface 嘛。函数和外界交互的边界就是函数的参数表和返回值等，统称函数原型。引脚就像参数，不同引脚的功能不同，就如同不同参数的含义不同。只要有相同引脚规范，就能“插”到主板上，那么函数只要具有相同原型，也可以被统一调用了。对于 DM4-51(c)，即使 3 个格式化函数有相同原型，但通过函数名显式调用，也无法实现调用新增函数而不修改代码。什么机制可用相同代码调用到具有相同函数原型的不同函数呢？我们再看看主板，它上面有插槽，插槽指向什么卡，就使用什么卡。既然可用函数指针变量来调用函数，那么变量就像一个插槽，只能“插入”（赋值）那些与它声明的函数原型相同的函数首址。只要一个函数指针变量被“插入”了相关函数首址，被调用的就是那个函数，调用代码都不变。DM4-52 展示了这一原理，其中 funcPtr 是函数指针，需要关注的是通过它调用函数，可用相同代码“functPtr(str);”调用到不同函数，关键在于第一步为它赋值什么函数入口地址。



分析 DM4-51(c)，我们发现 3 个格式化函数有相同原型，就可像 DM4-52 那样赋值给同一个函数指针变量。DM4-53 给出了用函数指针方式实现的关键代码，工程见“code\第 4 章\Format_3opts_func_ptr”。DM4-53(a)在类中定义了函数指针 m_formatHandler，其中粗体代码展示了通过该指针调用格式化函数。m_formatHandler 被赋值了什么函数的首址，就会调用到什么函数。

DM4-51(c)中那个讨厌的 switch 语句不见了，我们终于可以用不变的代码（DM4-53(a)的代码），调用到不同的函数，可以做到插件方式了。添加第 4 种格式化方式时，只需修改 DM4-53(b)代码，而 DM4-53(a)完全不用变动！

DM4-53

(a)

```
class FormatFile {
    ...
    fpFormatLn m_formatHandler;
};
void FormatFile::formatTheFile(char * fileName) {
    char ln[1024];
    ...
    while (!file.endOf()) {
        if (file.ReadLn(ln) > 0) {
            m_formatHandler(ln);
            printf("%s\n", ln);
        }
    }
    file.Close();
}
```

(b)

```
void FormatFile::doFormat() {
    ...          //获取用户输入的格式化选项，放入 cmd 中
    if (strcmp(cmd, FORMAT_UP) == 0) {
        m_formatHandler = strupr;
    }
    else if (strcmp(cmd, FORMAT_LW) == 0) {
        m_formatHandler = strFirstCharUpperCase;
    }
    else
        ...
    formatTheFile(arg);
}
```

DM4-53(b)从键盘获取用户输入命令，根据输入，将不同格式化函数首址赋值给 m_formatHandler。细心的读者会发现问题，这里依然有 if-else，4.7.1 节末尾已经指出 if-else 组合的使用与插件风格程序是违背的。那么，插件的梦想岂非空话？具体地说，假定我们增加第 4 种格式化，并为它分配一个新命令开关串，如“f4”，那么 DM4-53(b)就要再添加一个 if-else 分支语句：

```
else if (strcmp(cmd, "f4") == 0)
    m_formatHandler = format4
```

在回答该问题前，我们再审视“主板-显卡”模型。这个模型中似乎忽略了一个重要组件：组装系统的人。如果没有人这个角色，市场上有了新显卡时，谁将它纳入可选之列呢？谁来安装它？更直接地说，如果没有人，这个装配任务就只能推给主板或显卡。如果主板能完成“装配”功能，当它能装配新卡时，装配电路必然要修改！（**任意更换显卡而不修改主板，一个梦而已**。）DM4-53(b)就是在完成装配任务（为函数指针变量赋值的动作就像将卡插入槽中）。在实际系统中，如果我们能将装配模块独立出来，那么替换模块造成的修改也仅限于装配模块。那个修改卡并最后导致多米诺骨牌倾倒的事故就不会发生了。到此已经显著改善了软件的维护、扩展工作。

其实，如果用合适的设计，装配过程也可以不修改。比如，Java 语言中利用反射原理和 IOC（Inversion Of Control）容器，能通过只修改配置文件达成装配的变化。在我们的例子中能否完全实现插件呢？答案是可以的，请有兴趣的读者参考习题 7。

4.7.3 再进一步：做成对象

在 DM4-53(a)中，我们直接使用函数指针，这是过程式的做法，没有体现“自己操作自己数据”的面向对象思想。如果能将 3 种格式化方式看成 3 个类，这将是一个进步。

先定义基类 LnFormatter，见 DM4-54(a)，它拥有一个成员变量 m_handler，就是格式化函数指针。该指针设定为保护，因为子类要访问，但并不希望外部访问。对非子类而言，通过公有函数 getHandler 获得该指针。3 种格式化方式对应 LnFormatter 的 3 个子类，UpperLnFormatter 代表大写，LowerLnFormatter 代表小写，分别在其构造函数中将对应的格式化函数设定给 m_handler，见

DM4-54

(a)

```
typedef char* (*fpFormatLn)(char * ln);
class LnFormatter {
  public:
    fpFormatLn getHandler() {
      return m_handler;
    }
  protected:
    fpFormatLn m_handler;
};
```

(b)

```
class UpperLnFormatter : public LnFormatter {
  public:
    UpperLnFormatter(){
      m_handler = strupr;
    }
};
```

(c)

```
class LowerLnFormatter : public LnFormatter {
  public:
    LowerLnFormatter() {
       m_handler = strlwr;
    }
};
```

(d)

```
if (strcmp(cmd, FORMAT_UP) == 0) {
   m_formatHandler = new UpperLnFormatter();
}
else if (strcmp(cmd, FORMAT_LW) == 0) {
   m_formatHandler = new LowerLnFormatter();
}
else if (strcmp(cmd, FORMAT_FU) == 0)
   ...
```

(e)

```
1  void FormatFile::formatTheFile(char * fileName) {
2    char ln[1024];
     ...
3    fpFormatLn handler = m_formatHandler->getHandler();
4    while (!file.endOf()) {
5      if (file.ReadLn(ln) > 0) {
6        handler(ln);
7        printf("%s\n", ln);
8      }
9    }
10   file.Close();
   }
```

DM4-54(b)和(c)。

用户选择不同格式化方式时，我们构造不同的formatter对象，见DM4-54(d)。用户选择小写格式化时，我们执行m_formatHandler = new LowerLnFormatter()，构造LowerLnFormatter的对象，将指针赋值给m_formatHandler。

执行格式化功能时，见DM4-54(e)，第3行将调用m_formatHandler的getHandler，获得格式化函数的指针，赋值给handler。如果m_formatHandler指向的是LowerLnFormatter的实例，那么getHandler将返回strlwr的地址(因为DM4-54(c))；如果m_formatHandler指向的是UpperLnFormatter的实例，那么getHandler将返回strupr的地址（因为DM4-54(b)）。所以，在第6行调用格式化函数时可调用到不同的函数，达到调用不同函数而不修改调用代码的插件效果。工程见“code\第4章\Format_3opts_funcptr_obj1”。

引入对象的好处就是能将相关操作汇聚在一起，方便维护和操作。我们假定，用户希望查看当前采用的是什么格式化方式。例如，开始选择了大写格式，输入“-v”命令后，就打印“uppercase format”。如果依据“自己对自己行为负责”的原则，调用代码只负责问格式化对象“你的格式是什么”，然后将对象的回答打印出来，这样调用代码就不因格式化的不同而修改。这与DM4-54(e)格式化的逻辑一样：第3行调用代码问格式化对象：“你的格式化函数是什么？”，对象返回地址给handler变量，因此第6行能不修改代码调用到不同函数。

依照这个思路，在基类LnFormatter中再增加一个函数指针变量m_getFormatName，它指向的函数将返回格式化对象的名字，见DM4-55(a)。该指针将在子类的构造函数中初始化，DM4-55(b)中，UpperLnFormatter在其构造函数中将函数getFormatNameFunc()（返回字符串“uppercase format”代表大写格式化对象的名字）的地址赋给指针m_getFormatName。用户选择“-v”后，将调用当前对象的getFormatNameFunc()函数获得函数指针，返回格式化对象名字。代码如下：

```
printf("current format is : %s\n", m_formatHandler->getFormatNameFunc()())
```

“getFormatNameFunc()()”中第1个括号调用getFormatNameFunc()函数返回函数指针，第2个括号调用该函数指针指向的函数。如果m_formatHandler此时是UpperLnFormatter的对象，该语句的执行结果是打印“current format is : uppercase format”。工程见“code\第4章\Format_3opts_funcptr_obj2”。

DM4-55

(a)

```
class LnFormatter {
  public:
    fpFormatLn getHandler() {
      return m_handler;
    }
    fpGetFormatName getFormatNameFunc() {
      return m_getFormatName;
    }
  protected:
    fpFormatLn m_handler;
    fpGetFormatName m_getFormatName;
};
```

(b)

```
class UpperLnFormatter : public LnFormatter {
  private:
    static char * getFormatNameFunc() {
      return "uppercase format";
    }
  public:
    UpperLnFormatter() {
      m_handler = strupr;
      m_getFormatName = getFormatNameFunc;
    }
}
```

4.7.4 我们需要性能更好的版本

现在来看看 4.7.3 节中对象的内存布局，如图 4.14 所示。

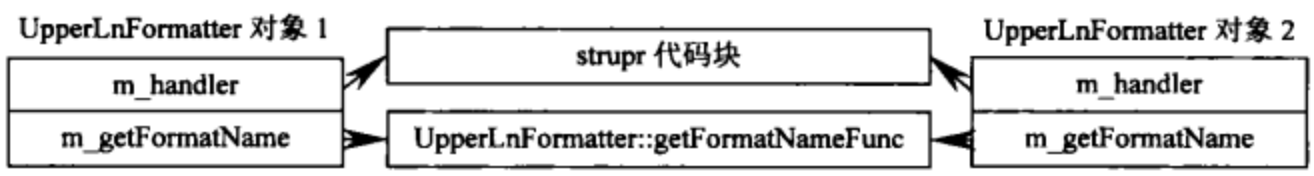


图 4.14

每个 UpperLnFormatter 对象均由两个函数指针变量组成，其中 m_handler 指向 strupr 的首址，m_getFormatName 指向 UpperLnFormatter::getFormatNameFunc。对于 UpperLnFormatter 的所有对象均如此。m_handler 能指向 strlwr 吗？不能，否则 UpperLnFormatter 的格式化将变成小写。这两个字段会变化吗？同理不能，否则 UpperLnFormatter 对象的这两个字段指向其他函数，意味着调用这两个字段指向的函数时，执行结果不是预期 UpperLnFormatter 应该有的行为。m_getFormatName 如果指向了 LowerLnFormatter::getFormatNameFunc，执行 upperFormatObj.getFormatNameFunc()()时，获得的是“lowercase format”而非“uppercase format”，执行 upperFormatObj.getHandler()("...")却是大写格式化效果，两者就不匹配了。因此，LnFormatter 子类所有对象的这两个字段值是不变的，相当于是常量。

这样，对象就非常浪费空间了。一个的类（如 UpperLnFormatter），所有对象前 8 字节的内容均相同，且不会修改。在现实中，如果阅读不会修改的信息，我们就会用公告的形式来发布这个信息。图 4.14 的方式相当于每个人都“复制”了一份 8 字节的“公告”，当然浪费。如果用一份 8 字节的公告记录两个函数首址，那每个对象怎样才能得到它？在现实中，每个人都知道公告栏在哪里。在程序中，让每个对象知道 8 字节公告（数组）的首址就可以。把这个首址放到对象上，于是有了图 4.15 的新内存布局。

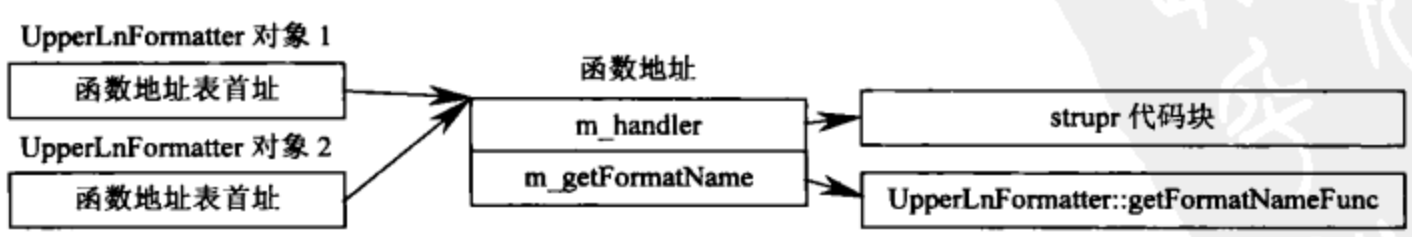


图 4.15

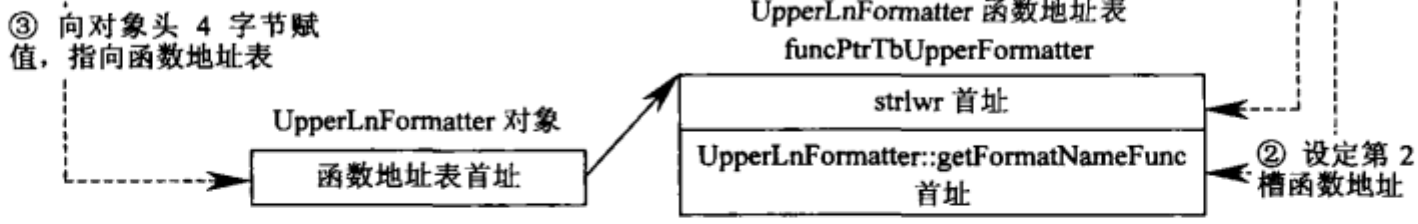
公告就是函数地址表，与图 4.14 的布局完全相同。新对象首部只有一个 4 字节字段，指向函数地址表。这样节省了很多内存空间，如果函数地址表有 100 个地址，而我们有 10 个对象，那么老方式的对象将占用 100×4×10 = 4000 字节，新方式的对象只占用 10×4 = 40 字节，如果加上函数地址表，也只有 40 + 100×4 = 440 字节。函数地址表越大，对象越多，空间节省效果越明显。按此思路，修改源代码，见 DM4-56，工程见“code\第 4 章\Format_3opts_funcptr_tb”。基类 LnFromatter 只有一个成员变量 m_handlerTb 指向函数地址表入口（void**代表指向的数组元素类型是 void *，因为存储不同函数指针，所以只能用 void * 来存储）。getHandlerTb 返回该函数地址表的首址。

DM4-56(a)

```
class LnFormatter {
  public:
    void** getHandlerTb() {
      return m_handlerTb;
    }
  protected:
    void ** m_handlerTb;
};
```

DM4-56(b)

```
1  void* funcPtrTbUpperFormatter[2];      //UpperLnFormatter 的函数地址表
2  int UpperLnFormatter::initClass() {                  //初始化函数地址表
3    funcPtrTbUpperFormatter[0] = strupr;
4    funcPtrTbUpperFormatter[1] = UpperLnFormatter::getFormatNameFunc;
5    return 0;
6  }
7  int gInitUpper = UpperLnFormatter::initClass();     //将在 main()之前执行
8  UpperLnFormatter::UpperLnFormatter() {
9    m_handlerTb = funcPtrTbUpperFormatter;
10 };
```



DM4-56(b)给出了 UpperLnFormatter 类的关键代码，其余两个类类似。第 1 行定义了 UpperLnFormatter 的函数地址表，为 void*全局数组。第 2 行函数负责初始化该数组，虚线①、②分别指示将数组的 0、1 两元素设定为对应函数的首址。第 7 行保证一定调用该初始化函数，且在 main()之前运行。那么，调用 new UpperLnFormatter()时，函数地址表已经初始化了。第 9 行是构造函数，构造新对象时，将对象前 4 字节赋值为函数地址表的入口地址，如虚线③所示。

至此，对象和函数地址表都准备好了，剩下的就是使用对象，见 DM4-56(c)。第 2 行调用对象的 getHandlerTb，从其前 4 字节获取函数指针表入口，如果此时对象 m_formatHandler 是 UpperLnFormatter，则返回值就是 DM4-56(b)中第 1 行定义的数组。获取其索引为 0 的元素即 strlwr 的地址（见 DM4-56(b)的说明），并赋值给 pFormat 函数指针。第 3 行通过 pFormat，将最终调用到 strlwr()，执行大写格式化效果。同理，第 6 行获得函数地址表索引为 1 的元素，即函数 UpperLnFormatter::getFormatNameFunc 的首址。最终调用时，将调用到该函数。

假定 m_formatHandler 是 LowerLnFormatter 的对象，以上过程最终获得小写格式化的效果。也就是说，LnFormatter 的各种子类就像我们的显卡，DM4-56(c)的代码相当于主板，来调用它们。m_formatHandler 指向不同的“显卡”，并不需修改 DM4-56(c)的代码。（**达到了插件效果，符合开闭原则**。）将来增加新格式化方式，只需“增加新代码”，即定义新的 LnFormatter 子类，而不需修改使用者即 DM4-56(c)的“旧代码”。同时，共享函数地址表的引入，在空间上比 4.7.3 节的方案获得了显著改善。

DM4-56(c)

```
1    if (file.ReadLn(ln) > 0) {
2        fpFormatLn pFormat = (fpFormatLn)(m_formatHandler->getHandlerTb()[0]);
3        pFormat(ln);
4        printf("%s\n", ln);
5    }
     ...
6    fpGetFormatName pGetName = (fpGetFormatName)(m_formatHandler->getHandlerTb()[1]);
7    printf("current format is : %s\n", pGetName());
```

4.7.5 我们需要新语法，创造虚函数吧

考察 DM4-56(c)，我们发现，使用并不方便，需要记住函数地址表中第 0 项为格式化函数，第 1 项为获取名字信息的函数，见第 2 行和第 6 行。这极易导致错误，且不利于开发和维护。DM4-56(b)中，第 2～7 行初始化函数地址表的逻辑，第 9 行初始化对象前 4 字节的逻辑对每个子类都有相似过程。人来做这些工作，重复且易错。有解决之道吗？从“悄悄传递 self 指针”可知，只要是机械动作，就特别适合程序来做，通过添加新语法，能告知编译器哪些代码需要进行自动处理。

现在面临的问题就有同样的特点：

① 每次调用格式化函数，机械地调用索引为 0 的元素指向的函数。

② 初始化子类相关的函数地址表和对象前 4 字节的工作重复、机械。

先分析第②个问题。为什么编译器要帮我们自动生成函数地址表？4.7.4 节中，所有代码都是一般的 C/C++语义，不能因为用了函数指针就一定要生成函数指针表吧？其实是因为要达到插件效果，我们将不同实现抽象出共同的函数接口，调用方想用同一段代码调用到不同实现的函数，才一步步走到现在。可见，正是因为调用这种有相同函数原型且目标相同的函数才会构建函数地址表并通过它来调用函数。（想想第 1 章中的内容，函数调用被翻译成一条相对 call 指令，而不是现在这样麻烦，从对象前 4 字节获取表入口，再偏移获取相关元素中存储的函数地址，最后调用。）只有让编译器理解这些函数是一种特殊的为插件方式而存在的函数，它才会帮我们构建函数地址表。在函数定义中加一个新关键字“virtual”，其实随便什么都可以。virtual 其实是虚函数的关键字，当编译器扫描一个类定义时，就可将该类所有标记为 virtual 的函数找出，然后分配一个全局数组，保存它们的地址。这时，我们给函数地址数组一个新称号“虚表”，因为是虚函数的地址表。

按什么顺序保存函数地址？这与问题①相关。按一定规则存入虚表，调用时自然会按相应槽位去获取地址。需要的只是一个规则。自然的想法是：写在前面的虚函数就放在虚表的前面。比如：

```
class A {
    virtual f1();
    virtual f2();
}
```

函数 f1()的地址就放在虚表的第 1 槽，函数 f2()放在第 2 槽。有人会问，如果在 A 的子类中，函数 f1()和 f2()的书写顺序反了呢？因为这两个函数已经在父类中出现了，在那里已经确定了顺序，所以不会再影响顺序了。准确地说，虚函数第一次声明时的顺序（覆盖版的顺序不影响）决定了虚表中存储的顺序。

生成虚表后，编译器在编译一个有虚函数的类（包括自己没有，但基类有）的构造函数时，就可在函数一开始悄悄加入代码，将虚表地址赋值给对象前 4 字节。

在调用虚函数时能够自动生成调用代码吗？查看以下代码：

```
A * pa;
...
pa->f2();
```

编译器遇到 pa->f2()时，它通过 A 的定义知道函数 f2()是“virtual”函数，然后根据前面虚表槽位存储的顺序，知道 f2()在第 2 槽（首次定义时在第 2 位）。那么，编译器根据虚函数的调用原则，就能从 pa 指向的对象前 4 字节获得虚表入口，然后偏移 4 字节，获得第 2 个元素值，即 f2()的地址。再用该地址跳转即可。

由此可见，有了关键字 virtual 后，编译器就能帮我们自动解决本节一开始谈到的麻烦。我们不仅获得了插件风格的代码，还获得了简洁、清晰、不会出错的语法。关键字 virtual 的诞生，让面向对象编译器又帮我们悄悄玩了一次指针——函数指针。

在继承机制下如何处理虚表呢？从 4.7.4 节我们看到，每个类有自己的虚表，因此在图 4.16 中，子类 B 也有自己的虚表。只要先将父类虚表内容复制一份，再查看子类 B 覆盖了哪些函数（实现了哪些自己的版本），然后将自己覆盖（实现）的函数入口地址填入到对应槽位即可。从图 4.16 可知，子类 B 的虚函数 p2()实现了自己的版本，所以它将虚表中第 2 槽的值用自己的函数 B::p2 的地址“覆盖”。该槽指向的是子类 B 自己的 p2 入口，其余两个函数没有实现自己的版本，所以没有“覆盖”原值，它们依然指向父类对应函数入口。(“覆盖”，override。确实覆盖了虚表中某个槽位，多么形象。)

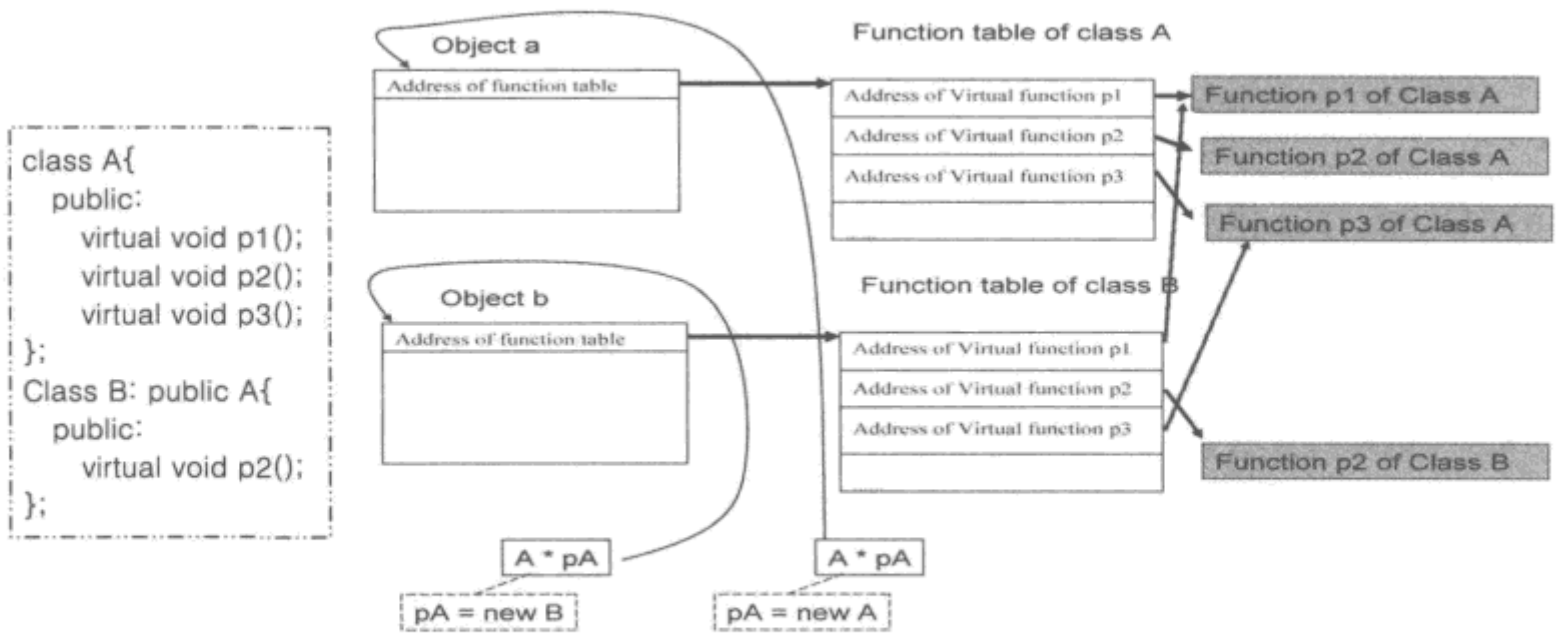


图 4.16

C++支持虚函数和函数重载两种多态，而真正支持模块化、插件风格编程的是虚函数机制。

插件风格需要不同的函数有相同的函数原型，因而调用方可用同样代码调用到不同的覆盖版本。而重载恰恰需要参数表不同，调用代码只有用 if-else 区别调用。

4.7.6 验证虚表机制（1）：反汇编分析

老规矩，现在我们要实证了。源代码见 DM4-57(a)，定义了两个虚函数：work 和 finish。那么，work 的地址应该在虚表第 1 槽，离虚表首部偏移 0 字节；finish 为第 2 槽，离虚表首部偏移 4 字节。

从 DM4-57(b)来看，第 1 行将 ebp–10h 内存中存储的对象首址赋值给 edx（用监视可以查看 ebp–10h 即&aClerk）。第 2 行将对象头 4 字节赋值给 eax，此时 eax 指向虚表入口。第 5 行取出虚表第 1 槽（前面已分析 work 应该在第 1 槽）的内容，跳转。第 4 行是为该函数传递 this 指针，即 aClerk 对象首址。第 13 行调用 finish 时，相对虚表首部 edx 偏移了 4 字节，从虚表第 2 槽取得 finish 入口，

这也符合结论。不过虚表不是虚函数实现的唯一机制。试想，这时每个类都有自己的一份虚表，如果子类完全没有覆盖父类虚表，将非常浪费。C++ Builder 以关键字 dynamic 替换 virtual，提供了新的实现方式。

DM4-57

(a)

```
class Clerk {
  public:
    virtual void work() {printf("i am clerk\n");};
    virtual void finish(){printf("i am clerk, i finish it\n");};
};
...
Clerk * aClerk;
aClerk = new Clerk;
aClerk->work();
aClerk->finish();
```

(b)

```
    aClerk->work();
1   0040d8ac   mov   edx, dword ptr [ebp-10h]
2   0040d8af   mov   eax, dword ptr [edx]
3   0040d8b1   mov   esi, esp
4   0040d8b3   mov   ecx, dword ptr [ebp-10h]
5   0040d8b6   call  dword ptr [eax]      //call work()
6   0040d8b8   cmp   esi, esp
7   0040d8ba   call  __chkesp(004019a0)
8   aClerk->finish();
9   0040d8bf   mov   ecx, dword ptr [ebp-10h]
10  0040d8c2   mov   edx, dword ptr [ecx]
11  0040d8c4   mov   esi, esp
12  0040d8c6   mov   ecx, dword ptr [ebp-10h]
13  0040d8c9   call  dword ptr [edx+4]    //call finish
```

4.7.7 验证虚表机制（2）：直接用虚表来调用虚函数

本节将利用虚函数调用原理，用代码从虚表获取函数地址并调用虚函数。其效果与直接调用虚函数一样。

DM4-58(b)第 5 行获得对象 vc 的首址，第 6 行从其头 4 字节获取到虚表入口赋值给 ptrVFT。第 7 行从 ptrVFT 指向内存的头 4 字节中获取到虚函数 p1 的地址，并赋值给 pVirFunc。第 8～11 行转跳到 pVirFunc 指向的函数 p1。第 9 行传递 this 指针。同理，第 12～15 行完成对 p2()的调用。第 12 行，ptrVFT+1 相对虚表首部 ptrVFT 偏移了 4 字节，从虚表数组第 2 槽中获得 p2 的入口（ptrVFT 为 void **，所指向的数据类型为 void*，其大小为 4 字节，所以加 1 就是加 4 字节）。执行结果证明，都成功地调用了对应函数。

DM4-58

(a)

```
class VirClass {
  public:
    virtual p1(){printf("hello p1\n");};
    virtual p2(){printf("hello p2\n");};
};
```

(b)

```
1   VirClass vc;
2   void ** aPtr;
3   void ** ptrVFT;
4   void * pVirFunc;
5   aPtr = (void **) &vc;
6   ptrVFT = (void **) * aPtr;
7   pVirFunc = * ptrVFT;
8   _asm {
9     mov   ecx, aPtr;
10    call  pVirFunc;
11  };
12  pVirFunc = * (ptrVFT + 1);
13  _asm {
14    mov   ecx, aPtr;
15    call  pVirFunc;
16  };
```

C++只要子类的函数名和原型与父类的虚函数相同，就默认是覆盖父类的，子类函数前是否加 virutal 并没有影响。在大型工程中，很有可能我们想在子类中定义新的虚函数，但恰巧名字和原型与父类的一个虚函数相同，就“偷偷被”覆盖了，虽然并没有定义新的虚函数。Delphi 和 C# 采用在覆盖函数上显式加入关键字 override 标记覆盖，否则即使名字和原型恰巧相同也不认为是覆盖。

4.8 虚函数的那些事儿

4.8.1 理解“=”

当语言的外衣让程序变得似乎越来越简单时，往往在这种简单背后隐藏了更多复杂性。在 C 语言中，等号就是 mov 指令的等价物，不过对全局和局部变量的 mov 不同罢了。在 C++中呢？下面

看看对象的赋值，如“对象 a=对象 b”会是什么样子？

用 4.5.3 节的方式猜测，既然“对象就是结构体变量”，就以结构体的赋值来对比分析。自然的思考是，结构体的每个成员变量都会产生一条 mov 指令赋值，有多少成员就产生多少条语句，那么对象也应该有相似行为，首先查看同一个类的两个对象赋值的反汇编。定义 Test 类有两个 int 成员 i1 和 i2，DM4-59 展示了将对象 t2 赋值给 t1 的过程。从第 3 行和第 5 行容易推知，ebp–8 和 ebp–4 分别是 t1 的成员 i1 和 i2 的地址。这样可推知第 7～8 行是将 t1.i1 的内容赋值给 t2.i1，第 9～10 行是将 t1.i2 的内容赋值给 t2.i2。证明这点，只需将第 8 行和第 10 行的地址 ebp–10h、ebp–0ch 在监视中查看，分别与&t2.i1 和&t2.i2 比较，就会发现两两相等。对于最简单情况，对象赋值和结构体是相同的（结构体赋值的验证请自己分析）。

DM4-59

```
1    Test t1, t2;
2    t1.i1 = 1;
3      mov  dword ptr [ebp-8], 1
4    t1.i2 = 2;
5      mov  dword ptr [ebp-4], 2
6    t2 = t1;
7      mov  eax,dword ptr [ebp-8]
8      mov  dword ptr [ebp-10h], eax
9      mov  ecx, dword ptr [ebp-4]
10     mov  dword ptr [ebp-0ch], ecx
```

在相同类的对象赋值中还有复杂情况吗？我们想到了 Person 类和 FileStream 类，想到了 4.5.3 节中探讨析构释放资源带来的麻烦。因为 Person 类在析构中释放内存，如果将 Person 对象 P1 赋值给 P2 后，两个对象的成员变量将相同，那么成员变量指针 ID 就指向同一内存，当其中一个对象析构后，共享的这块内存就无效了，之后另一个对象析构时将再次释放该内存，因此必然导致问题。拷贝构造函数本质上就是等号赋值的作用，但它会在赋值时被调用吗？如果调用，则不会出问题，否则必须重载“=”操作符。在之前的 Test 类中增加拷贝构造（见 DM4-60(a)第 3 行）和重载等号操作符（见 DM4-60(a)第 4～10 行）。其中，等号重载操作符函数所在的第 5～6 行是判断等号右边对象的地址与自己的是否相同，如果相等，则不用自己给自己赋值。

DM4-60

(a)

```
1   class Test{
2     ...
3     Test(Test& src) { *this = src; }
4     Test& operator = (const Test& right) {
5       if (this == &right)
6         return *this;
7       i1 = right.i1;
8       i2 = right.i2;
9       return *this;
10    }
11  };
```

(b)

```
1    t2 = t1;
2      lea    eax, [t1]
3      push   eax
4      lea    ecx, [t2]
5      call   Test::operator = (4329dch)
```

同样的“t2=t1;”代码的反汇编结果见 DM4-60(b)。我们选择直接显示地址对应的源代码符号，这样容易识别。第 5 行表明调用了=操作符函数。第 2～3 行获取右边赋值对象 t1 的地址，并压栈传参（operator=函数要接收等号右边对象的地址）。第 4 行将左边被赋值对象 t2 的地址传入 ECX（传递 this）。如果将 DM4-60(a)的第 4～10 行注销，我们发现，“t2=t1;”的反汇编结果与 DM4-59 一样。说明，如果没有重载等号，遇到对象赋值时，并不调用拷贝构造函数。由于两者执行了相同逻辑，为了避免重复即罪恶，我们在拷贝构造函数中调用重载的等号，如 DM4-60(a)第 3 行。这里对 4.5.3 节末尾对整体资源管理注意事项中再增加一条：重载等号并将其定义为私有，防止对象与对象赋值。因为，FileStream 在赋值函数中难以处理其打开的文件，这与其拷贝构造无法处理从而设为私有是一样的道理（见 4.5.3 节）。

下面分析引入继承后，父类对象赋值给子类对象或者子类对象赋值给父类对象会怎样。

从对象模型上讲，子类对象可能比父类多定义了一些字段，形象地说，子类对象比父类的“多”，子类上部和父类对象是相同的。非常自然的想法是，相同的部分应该可以相互赋值。如 DM4-45(a)和(b)分别定义了父类、子类，实验发现，子类可赋值给父类对象，反之不行。这也可理解，我们可以说马是动物，但动物却并非一定是马。赋值暗含的前提就是：等号右边的 xx 是左边的 yy，所以不支持父类对象赋值给子类对象。图 4.17 给出子类对象 child 赋值给父类对象 parent 时的反汇编跟踪结果，从中容易发现之前的推论：“当子类对象赋值给父类对象时，与父类共同的部分被赋值给

父类对象。”

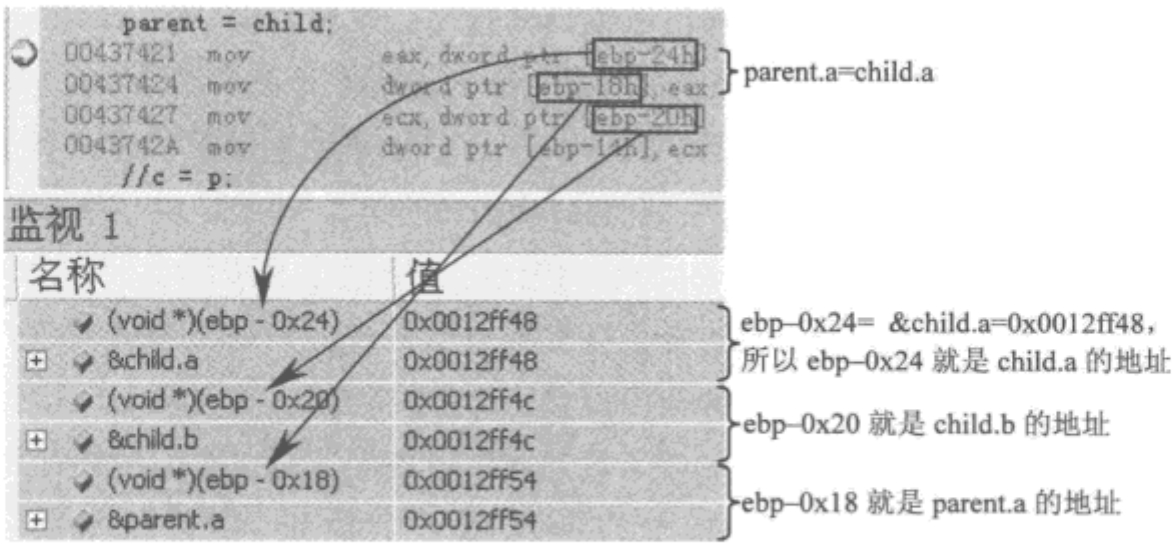


图 4.17

最后来分析：当有虚函数时，对象的赋值又是怎么回事？DM4-61(a)给出了父类 Clerk 和子类 WangClerk，子类对父类虚函数 work()进行了覆盖。DM4-61(b)展现了两种调用虚函数的情况，第 1～3 行执行 work 时调用的是 WangClerk::work，第 4～7 行调用的是哪个类的 work()？前一种情况很好理解，因为 aClerk 指向的是 WangClerk 的对象，其前 4 字节指向 WangClerk 的虚表入口，通过虚函数的访问机制从虚表中获得 WangClerk::work 的入口，调用的是该函数。这是我们需要的多态。而第 4～7 行呢？第 6 行进行了对象赋值，将 wang 赋给 clerk。它能调用 WangClerk::work 吗？能否调用关键在于，赋值后 clerk 的前 4 字节是否指向 WangClerk 的虚表入口。

DM4-61

(a)

```
class Clerk {
  public:
    virtual void work() {
      printf("i am clerk\n");
    };
};
class WangClerk: public Clerk {
  public:
    void work(){printf("i am wang\n");};
};
```

(b)

```
1   Clerk * aClerk;
2   aClerk = new WangClerk;
3   aClerk->work();
4   Clerk clerk;
5   WangClerk wang;
6   clerk = wang;
7   clerk.work();
```

(c)

```
class WangClerk: public Clerk {
    int i;
  public:
    void work(){ i = 4 };
};
```

如果仅仅将这前 4 字节看成与其他成员变量一样的字段，似乎可以赋值。但不同虚表代表不同行为特征，可以说，对象的虚表代表其身份，将 clerk 的虚表换成 WangClerk 的，岂非就是将人变成动物的行为？从技术层面分析，子类虚函数可能访问子类定义字段，该字段位置必然在父类对象内存区域后。如果将子类虚表入口赋给父类对象的前 4 字节，将调用子类虚函数，岂非访问越界？比如，将 WangClerk 的定义修改为 DM4-61(c)。调用 work()，就会写距对象头部 4 字节的内存，父类本身只有虚表指针所占的 4 字节，这样岂非越界？通过实际运行，确实 DM4-61(b)第 7 行执行的依然是父类的 work。

那么，我们强迫将 WangClerk 的虚表地址赋给父类对象 clerk 呢？这样似乎会让第 7 行执行子类的虚函数，见 DM4-62(a)，第 4 行完成了将 WangClerk 虚表地址赋值给父类对象 clerk 头 4 字节的工作。实验结果令人惊讶，依然调用父类的 work()！我们用反汇编展开，见 DM4-62(b)。注意，第 5 行将 ecx 设置为 clerk 对象的首地址，即 ebp–8。第 6 行并没有用这个地址去获取对象前 4 字节的虚表入口，而是直接用 call 指令调用了 Clerk::work，指令中的 0043217B 就是该函数的入口。如果选择显示符号名，那么第 6 行就是“call Clerk::work (43217bh)”。也就是说，此时编译器生成的调用代码根本没有按照虚函数的调用方法去调用函数，而是直接调用了 Clerk::work，因此替换前 4 字节的虚表指针自然无效。

DM4-61

(a)

```
1   Clerk clerk;
2   WangClerk wang;
3   clerk = wang;
4   *(void **)(&clerk)) = *(void **)(&wang));
5   clerk.work();
```

(b)

```
1   *(void **)&clerk = *(void **)&wang;
2     mov  eax, dword ptr [ebp-0ch]
3     mov  dword ptr [ebp-8], eax
4   clerk.work();
5     lea    ecx, [ebp-8]
6     call   0043217b
```

(c)

```
1   aClerk = &clerk;
2     lea    eax,[clerk]
3     mov  dword ptr [aClerk], eax
4   aClerk->work();
5     mov  eax, dword ptr [aClerk]
6     mov  edx, dword ptr [eax]
7     mov  ecx, dword ptr [aClerk]
8     mov  eax, dword ptr [edx]
9     call   eax
```

为什么不用虚函数方式调用 work()呢？work()不是虚函数吗？换位成编译器想想：既然明确写了 clerk.work()，clerk 是类 Clerk 的对象，那么说明就是调用 Clerk 的 work()。既然都明确调用 Clerk::work 了，如果用虚函数的方式调用，还会多出几条指令（见 DM4-62(c)的第 5～9 行，共 5 条指令，直接调用仅 DM4-62(b)中的第 5～6 行的 2 条指令），而效果又一样，编译器自然会选择直接调用方式。如果将 DM4-62(a)中的第 5 行改成指针调用方式，即 DM4-62(c)中的第 1 行和第 4 行，则编译器将按虚函数方式生成调用代码（因为用指针访问无法在编译时推断出应调用哪个函数）。实验结果确实调到了子类 WangClerk 的虚函数。由此，我们从机制上明白了，在 C++中调用虚函数要用对象的指针调用才有虚函数效果，也明白了，在有虚函数情况下，父子类对象的赋值是不能赋值对象前 4 字节的，因为虚表地址本质上代表了一个类的行为，是一个类之所以是该类的关键。

4.8.2 纯虚函数，从 DLL 导入对象

首先理解什么是纯虚函数。如果定义一个虚函数，在其后加入“= 0”，就是纯虚函数。“纯”就意味着它没有实现，如“virtual myfunc() = 0;”，此时函数没有函数体。纯虚函数并非是“virtual myfunc(){};”这样没有在函数体中填写代码的函数。这个空函数体的函数其实是有代码的，至少有一条 ret 指令。纯虚函数有什么影响呢？就是会在虚表中产生一个对应槽位。本来槽位是存储函数地址的，因为函数没有实现，这个槽位也似乎无可存储，挺浪费。一个没有实现又白占虚表空间的纯虚函数有何用？下面和后面的 5.2.3 节将展示其功能。

前面已经了解到，动态链接库可以导出函数供加载者调用，如果将对象用动态链接库的方式导出，该怎么办？初看这个问题有点茫然，这时我们又需要从已了解的机制上去分析。

先看非虚函数，其调用被翻译成一条相对 call 指令，这时编译器必须知道被调用函数的地址，才能计算出相对 call 指令中的偏移量。对于非 DLL 方式，我们已经在 4.1.6 节中分析过，因为调用指令和被调用代码均在一个执行程序中，所以编译器在编译阶段就能计算出这个偏移量。但是，现在被调用代码在 DLL 中，编译器无法事先知道其加载地址，所以无法计算这个偏移量了。（非虚函数如果从 DLL 导出应该怎么办，见习题 11。）

再看虚函数，它的调用过程是从对象前 4 字节拿到虚表入口，然后根据类定义中虚函数定义的位置计算出函数地址在虚表中存储的位置，获得函数地址，然后跳转。这一过程不需要在编译期事先知道被调用函数地址，而是在运行期获得的。（**正因如此，才有了一段相同代码可调用到不同虚函数的多态效果**。）那么，编译期生成虚函数调用的代码其实只需类定义的头文件就可以了，用它计算出函数地址在虚表中的位置，因此我们可以从 DLL 导出虚函数，生成一个头文件，其中类的定义“只包含纯虚函数”。该头文件被 DLL 工程和调用 DLL 的工程包含。对于调用方，只要拿到对象的指针 pObj，就可以用 pObj->virtualFunc1(...)这样的语句调用虚函数了（只需对象首址就能获得虚函数入口地址）。而 DLL 方需要导出函数 createObj，它返回对象指针即可。既然头文件中定义的类只有虚函数，那么意味着必须覆盖并实现它们。所以，在 DLL 的工程中需要继承该类并实现。

工程见“code\第 4 章\import_obj”，其中包含了 DLL 和调用 DLL 的两个工程，关键代码见 DM4-63。其中，DM4-63(a)给出了两个工程均引用的头文件 common.h 的内容，定义了 Parent 基类，其两个函数均为纯虚函数，并定义了构造对象的 createObj 函数，它的返回是 Parent *，意味着可能返回任何 Parent 的子类对象。该函数的实现在 DLL 中，并从 DLL 导出了该函数（工程中的 export.def 文件定义了 EXPORTS createObj，从而导出）。DM4-63(b)给出了 DLL 工程的实现，引用了 common.h，以 Child 类继承了 Parent，并覆盖了两个虚函数。同时，其 createObj()函数返回的是 Child 对象的指针。DM4-63(c)调用了 DLL 的 createObj()函数构造对象，此时自然调用到 DM4-63(b)中的 createObj()函数。那么，返回的对象是 Child 类，由指针 p 指向。再通过 p->setNum...，p->getNum...来调用虚函数，实际就调用到 Child 的覆盖版本。代码中的箭头表示执行语句前、后相关寄存器或符号指向的内存位置。从 DM4-63(d)可清晰看出，setNum 被翻译成虚函数调用的相关指令，这一翻译过程完全不依赖具体 Parent 子类的实现，仅依赖于 DM4-63(a)的 common.h 中 Parent 类的定义。换句话说，如果加载另外一个实现了 createObj()函数和 Parent 子类的 DLL，DM4-63(d)的代码依旧不变，但它能调用到不同实现版本。

DM4-63
(a) 两个工程共用的 common.h

```
class Parent {
  public:
    virtual void setNum(int v) = 0;
    virtual int getNum() = 0;
};
Parent* createObj();
```

DM4-63
(b) DLL 工程的实现代码

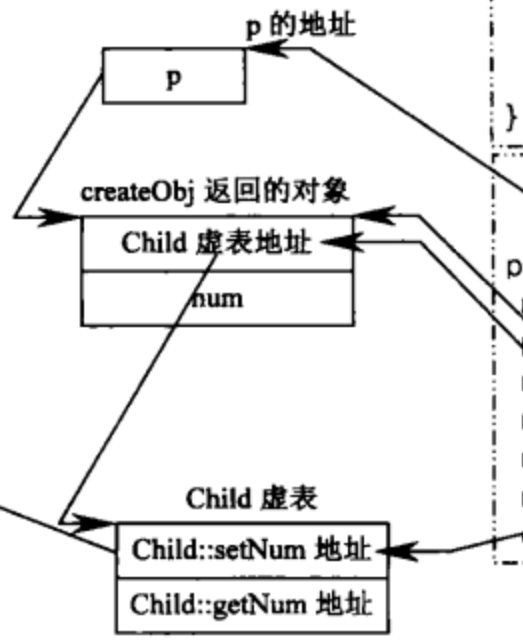
```
#include "../inc/common.h"
class Child : public Parent {
  public:
    void setNum(int v){ num = v; };
    int getNum(){ return num; };
  private:
    int num;
};
Parent* createObj() {
    return new Child();
}
```

DM4-63
(c) 调用 dll 工程的代码

```
#include "../inc/common.h"
void main() {
    Parent* p = createObj();
    p->setNum(2);
    printf("num=%d\n", p->getNum());
}
```

DM4-63
(d) 调用 setNum 的反汇编

```
p->setNum(2);
 mov  esi, esp
 push 2
 mov  eax, dword ptr [ebp-8]
 mov  edx, dword ptr [eax]
 mov  ecx, dword ptr [ebp-8]
 mov  eax, dword ptr [edx]
 call  eax
```



4.8.3 C 语言实现虚函数

由于 C 语言不会自动形成虚表，所以如果像 C++一样，让一个类的所有对象共享虚表会非常麻烦。我们又回到 4.7.3 节的做法，将函数指针定义到结构体（本节为了方便，称这些结构体为类、子类或父类）。利用 4.6.4 节以宏模拟类的做法，我们只需在相当于子类的初始化函数中，将函数指针变量赋值为子类版本的函数地址即可，工程见“code\第 4 章\structvirtual”。DM4-64(a)是子类实现的 setB()版本，该函数在 Parent 中已经实现，为 Parent_setB()。在 Child_setB 中，先调用父类实现完成赋值工作，然后打印“I am child ....”。DM4-64(b)是子类的初始化函数，粗体语句将父类 Parent 中定义的函数指针 setB 变量绑定为子类 Child 的版本 Child_setB。这时，如 DM4-64(c)所示最后一行，通过函数指针 setB 调用函数时，调用的就是 Child_setB，而非父类的 Parent_setB（因 Child_new 调用 DM4-64(b)的 Child_init 导致 setB 变量和 Child_setB 绑定）。这种虚函数思想在 Linux 中多有体现，如虚文件系统。

DM4-64

(a)

```
void Child_setB(struct Parent *self, int v) {
    Parent_setB(self, v);
    printf("I am Child::setB\n");
}
```

(c)

```
CLASS Child * pc;
CLASS Parent * p2;
pc = Child_new();
p2 = (CLASS Parent *) pc;
p2->setB(p2, 2);
```

(b)

```
BOOL Child_init(struct Child * self) {
    SUPERCLASS_INIT(self, Parent, Child);
    ((struct Object *)self)->dtor = Child_dtor;
    self->setC = Child_setC;
    ((struct Parent *)self)->setB = Child_setB;
    return TRUE;
}
```

4.8.4 魂归何处：析构之“虚”

下面来看 DM4-65 代码，分析它的结果。此时，DM4-65(c)的“delete p”并非如我们所想地调用了 WangClerk 的析构，而是调用了 Clerk 的析构。从编译器的角度也很好理解：当 delete 指针 p 时，p 的类型为 Clerk，所以只能调用 Clerk 的析构。因为仅从定义无法获知该指针到底指向 Clerk 或它的某个子类的对象。这种情况就可能发生内存泄漏。假如 WangClerk 的构造函数分配了内存，在其析构中就会释放该内存。但现在 delete 没有调用 WangClerk 的析构，自然内存泄漏。

DM4-65

(a)

```
class Clerk {
  public:
    ~Clerk(){ printf("i am clerk\n");}
};
```

(b)

```
class WangClerk: public Clerk{
  public:
    ~WangClerk() { printf("i am wang\n"); };
};
```

(c)

```
Clerk* p =
 new WangClerk();
delete p;
```

很多时候，我们以基类指针的方式来使用对象（如为了虚函数的多态效果），这时无法正确调用析构是很严重的问题。怎么办？只要能调到子类 WangClerk 的析构，一切都解决了。这样，子类析构会在结束自动调用父类析构（见 4.6.3 节最后），最终导致一个从子类到父类的析构函数调用序列。如何通过指针执行到实际对象的析构？这就是虚函数的效果。因此，将基类的析构定义为虚函数则解决了该问题。所以，为了“delete 对象指针”这种语句能成功调用到子类的析构，好习惯是，将所有类的析构定义为虚函数。

4.8.5 理解运行期类型判断 dynamic_cast

现在来看一个问题：通过一个对象的指针能否判断出它的类型是什么？比如，通过 DM4-65(c)中的指针 p，我们能在运行期判断 p 指向的是什么类的对象吗？Clerk 还是 WangClerk？在不考虑虚函数的情况下，一个对象等价于去除所有成员函数后的结构体变量。这个问题其实就变成了：给定一个结构体变量的指针，能判断出该变量是什么结构体吗？这明显是不可能的，因为没有类型信息（或者整数，或者字符串）放在结构体中。结构体指针指向的内存，除了存储成员变量，根本没有任何内存来存储类型信息，或者存储指向类型信息的地址。所以反汇编执行程序时，我们无法在源代码级恢复出对结构体的访问。这个类型判定问题到此似乎就很清楚了，无解。可我们知道 C++中有一个操作符 dynamic_cast 可达成以上目的。语句“dynamic_cast<WangClerk *>(p) != NULL;”为真，则指针 p 指向的对象就是类型 WangClerk。

让我们来想想这是怎么回事。先猜测，再实证。什么能代表类型呢？我们可分配一个独一无二的整数来代表一个类，如 Clerk 为 1，WangClerk 为 2，并将这个整数存储到对象中，那么，如果给

定对象的地址，就可用该地址求偏移量，找到存储该整数的地方获得该类型。这样每个对象都要增加 4 字节来存储代表类型的整数（当然，这 4 字节也可存储一地址，指向更复杂的结构描述类型信息，如类名等）。该做法似乎有点浪费空间。4.8.1 节已分析过，一个类的虚表决定着该类的行为，即虚表就是一个类的特征，能代表其身份。每个类有自己的虚表，这样虚表地址就是类的唯一标识（一个不会重复的整数），自然可以代表一个类。对于有虚函数的对象，非常符合我们的设想，因为虚表地址就放在对象的前 4 字节，所以我们很容易得出 dynamic_cast<类 X>（对象指针）算法，见 DM4-66。如果该推断为真，那么没有虚函数的类就无法使用 dynamic_cast。去掉类中的虚函数时，编译会报错：“"dynamic_cast": "Clerk"不是多态类型”。下面如 DM4-65(a)和(b)定义父类、子类，将 DM4-65(a)中的析构定义为虚函数，然后反汇编 DM4-67 的代码来验证猜测。

DM4-66

```
(1) 从对象指针指向的前 4 字节获取虚表地址;
(2) if (该地址 == 类 X 的虚表地址)
      则对象是类 X
```

DM4-67

```
Clerk* p = new WangClerk();
WangClerk *pw = dynamic_cast<WangClerk *>(p);
```

反汇编见 DM4-68，第 8 行表示调用了 dynamic_cast 函数，传入的参数有第 3 行压栈的 4af01ch，该值被注释为 WangClerk 'RTTI Type Descriptor'，由此可以推断 4af01ch 应该就是 WangClerk 的类型信息描述块的入口地址。这与猜测不符，我们猜测用于比较的应该是 WangClerk 的虚表地址。从图 4.18 最上部可知，对象指针 p 指向的是 0x00385bf0，而其前 4 字节是 WangClerk 虚表入口地址 0x49af90，显然不等于第 3 行压栈的 4af01ch，但也并非完全没有干系，第 6～7 行传入了对象首址。这意味着可通过它获得虚表地址。毕竟 dynamic_cast 要进行指针强制转换，并非仅仅进行类型判断，所以它可能需要更复杂的类型信息结构。合理的设想是，通过第 6～7 行传入的对象首址获得虚表指针，并通过虚表指针拿到该类的 RTTI Type Descriptor。然后与第 3 行传入的 WangClerk 的 RTTI Type Descriptor 地址 4af01ch 比较，最终判断该对象是否为 WangClerk。那么，如何通过虚表指针最终访问到类型信息入口地址 4af01ch？只有跟踪看看。

DM4-68

```
1   WangClerk *pw = dynamic_cast<WangClerk *>(p);
2       push 0
3       push offset WangClerk 'RTTI Type Descriptor' (4af01ch)
4       push offset Clerk 'RTTI Type Descriptor' (4af004h)
5       push 0
6       mov eax,dword ptr [p]
7       push eax
8       call @ILT+6635(___RTDynamicCast) (4329f0h)
-----------------------------------------
___RTDynamicCast 函数
...
9       mov   eax,dword ptr [ebp+8]
10      push eax
11      call FindCompleteObject (434F70h)
-----------------------------------------
FindCompleteObject 函数
....
12      mov eax,dword ptr [ebp+8]   //对象地址赋给 eax
13      mov ecx,dword ptr [eax]     //获得虚表地址赋给 ecx
14      mov edx,dword ptr [ecx-4]   //虚表向前偏 4 字节获取所存值
```

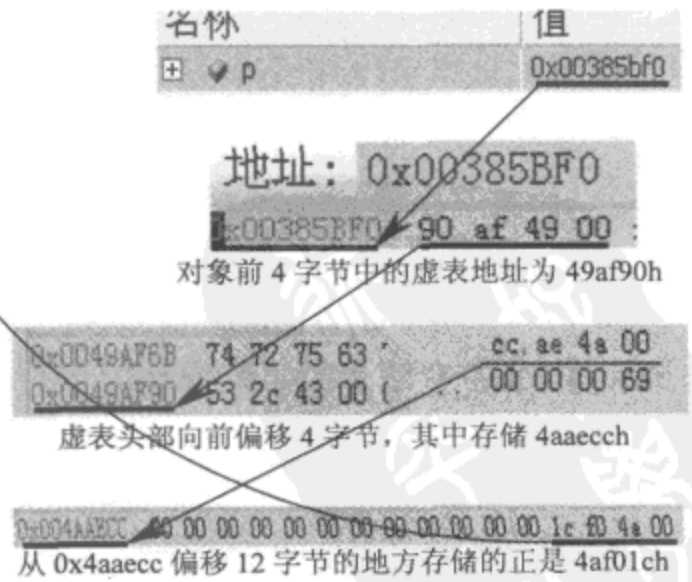


图 4.18

第 8 行调用了函数___RTDynamicCast，第 6～7 行将对象地址作为第一个参数最后压栈。在第

8 行跟踪进___RTDynamicCast 函数，我们关注的是它如何用对象地址获得 4af01ch。既然对象地址最后压栈，那么在函数中就是以[ebp + 8]来访问的。此时只需查看访问了 ebp + 8 地址的指令，我们发现了第 9～11 行。第 11 行调用函数 FindCompleteObject，又将对象地址作为其参数压栈。这时我们再次跟踪进 FindCompleteObject，同理，依然关注该参数的使用，即访问了 ebp + 8 地址的指令，发现了第 12～13 行，其效果是在虚表入口向前偏移（即减）4 字节的地方获取了一个值（向后偏移存放的是虚函数地址）。那么，这个值很可能是类型描述块的地址，从图 4.18 可知，它是 0x4aaecc，并非之前的 4af01ch。也许这依然是一个更复杂结构的地址，它指向的内容可能包含了 4af01ch。我们查看内存 0x4aaecc，见图 4.18 底部，在离该地址 12 字节的地方存放着 4af01ch！可见猜想基本思路正确，“虚表地址代表了类的身份”。只是这个地址经过了更复杂的转换才获得真正的类型信息块。整理图 4.18，得到图 4.19。为了证明猜想和分析正确，我们做了如下 3 个实验。

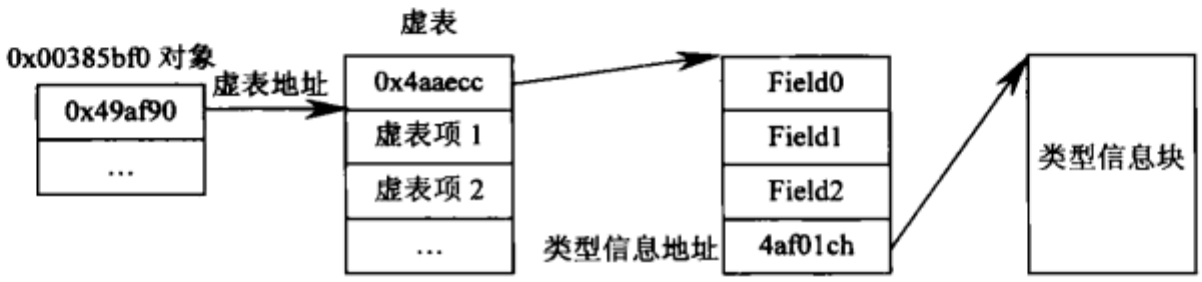


图 4.19

实验一：我们将对象前 4 字节指向另外一个类的虚表，则 dynamic_cast 就应该认为对象类型是指向的那个类，见 DM4-69。执行第 3 行时，因为 p 指向的是 Clerk 对象，所以 pw 为空。第 7 行将 WangClerk 的虚表指针赋给对象前 4 字节后，第 9 行执行时，pw 非空，被系统认为是 WangClerk，符合预期。虚表地址代表了身份！

DM4-69

```
1   Clerk* p = new Clerk();
2   WangClerk * pw = dynamic_cast<WangClerk *>(p);
3   if (pw == NULL)
4       printf("is not wangclerk\n");
5   else
6       printf("is wangclerk\n");
7   *(void **)p = *(void **)(new WangClerk());
8   pw = dynamic_cast<WangClerk *>(p);
9   if (pw == NULL)
10      printf("is not wangclerk₩n");
11  else
12      printf("is wangclerk\n");
13  delete p;
```

实验二：进一步我们尝试不修改对象头 4 字节，而将 Clerk 虚表首址向前偏 4 字节的值，也就是图 4.19 中虚表中保存 0x4aaecc 的这个位置的值替换成 WangClerk 虚表中该位置的值。发现 Clerk 的对象也被 dynamic_cast 判断为 WangClerk 类型。因为虚表是只读的，所以为了简便，我们通过打断点，在内存窗口中修改。

实验三：最后，我们尝试不修改实验一和二所涉及的位置，将图 4.19 中 Clerk 类型相关的从左数第 3 个矩形偏 12 字节处，即存储 4af01ch 处的值，修改为 WangClerk 该位置的值（也需要在内存窗体中修改），结果并未成功欺骗系统。经分析发现，紧邻该位置的一个整数值与前面的值很相近，都是 4axxxxh，可能又是一个与类型转换相关的数据。将该值输入反汇编窗体的地址栏中，发现它被称为 Clerk:: 'RTTI Class Hierarchy Descriptor'，明显与继承关系有关。继承关系和类型判断有何关系？试想，有 A 和 B 两个类，A 是 B 的父类。虽然虚表地址能代表类，但如果只比较 A 与 B 的虚表地址，两者肯定不同，但“B is A”肯定正确（p 指向类 B 的对象，此时 dynamic_cast<A *>(p)不为 NULL）。所以，合理的逻辑是，通过虚表地址获得 B 的运行期描述后，通过它能找到父类 A 的运行期描述，进行比较，从而确定“B is A”。而表示 A 和 B 类继承关系的结构应该是刚刚看到的 RTTI Class Hierarchy Descriptor，所以尝试将该值也修改为 WangClerk 对应位置的值，结果成功。关于继承关系与类型判断猜想的证明请做习题 15。

4.9 静态覆盖

DM4-70

```
1   class Parent{
2     public:
3       void doIt(){ printf("parent\n");}
4   };
5   class Child : public Parent {
6     public:
7       void doIt(){ printf("child\n");}
8   };
9   Child c;
10  Parent * pp = &c;
11  pp->doIt();
12  Child * pc = &c;
13  pc->doIt();
```

如 DM4-70 所示，如果子类和父类有某函数，它们同名、同原型，但非虚函数，这时会出现什么情况？第 11 行和第 13 行调用 doIt 时，第 11 行调用父类的，第 13 行调用子类的。

为什么会这样？换位成编译器来思考。执行第 11 行时，发现调用的 doIt 并非虚函数，就需要确定是哪个类的函数，然后用相对 call 直接跳转。pp 的类型是 Parent *，因此只能认为 doIt 是 Parent 的。执行第 13 行时，同理，pc 的类型是 Child *，所以确定为调用 Child 的 doIt。简单地说，在非虚函数下，父子类因函数名和原型相同产生的覆盖被称为静态覆盖。编译器根据以何种类型对象调用该函数来确定调用哪个函数。如何确定类型？如用对象指针调用函数，就是指针的类型；如用对象调用函数，就是对象的类型。

4.10 静态与非静态成员函数的区别

DM4-71

```
1   class Parent{
2     public:
3       void doIt(){ printf("parent\n");}
4       static void doStatic() { printf("parent\n"); }
5   };
6   ...
7   Parent p;
8   p.doIt();
9   004344c9    lea    ecx, [ebp-1]
10  004344cc    cal    004313f2
11  p.doStatic();
12  004344d1    call   00431d3e
```

静态函数是什么？记不清没关系，来看它的反汇编，见 DM4-71。第 9～10 行调用了非静态函数，第 12 行调用了静态函数。其关键差异在于第 9 行，从 4.1.6 节知，这就是悄悄传递对象地址到 ECX。静态函数调用时没有传递对象地址。没有传递对象地址，就无法通过该地址求偏移量访问对象的非静态成员变量。无法获取对象地址，也无法调用非静态成员函数，因为它们需要传递对象地址。这时，静态成员函数的限制就清楚了。

再问一个问题，静态函数可以是虚函数吗？有点摸不着头脑？还是从机制出发来分析。如果是虚函数，就需要通过虚表访问，获得虚表地址。C++通过对象前 4 字节拿到虚表地址。静态函数与对象毫无关系，本质就是一个 C 函数。与对象无关，自然无法获取虚表地址，当然就与虚函数无缘了。只要能拿到虚表地址，这个问题就能解决。Delphi 通过 class of class 机制可获得虚表地址，从而其静态函数也可以是虚函数。

4.11 遥远的风景：管窥 .NET 对象

汇编与 .NET 及 Java 这种管控平台似乎很远，就像计算世界的原始社会和信息时代。可是，手握反汇编这把石斧，我们却能剖析洞察那遥远的风景。因为褪去华丽的外衣，一切存在只剩下两样：内存和指令。从这里入手，就能看清 .NET 对象机制。这就是根源的力量。本节用 VS 中的反汇编调试来简单分析 .NET 对象模型和虚函数调用机制。

代码见 DM4-72，包含两个虚函数和两个成员变量。首先，分析对象内存布局是否与结构体类似且悄悄传递了 this 指针。反汇编调试，见 DM4-73(a)，调用 set()函数前的第 2 行对 ECX 进行了设定，这很像传递 this 指针。执行第 2 行后查看寄存器，ECX 的值为 0x0194f160。然后跟踪进 set()

函数，反汇编见 DM4-73(b)。其第 2 行之前没有修改 ECX，第 2 行将其保存在 ebp–3ch 指向的内存中。第 3 行又从该内存中取出其值并赋给 EAX。从第 4 和 7 行对成员变量 a、b 的赋值来看，EAX 应该是对象首部地址。因为从第 4 行和第 7 行可知，a 的地址是 eax+4，b 的地址是 eax+8。说明 a 和 b 相邻存放，这类似于 C++对象中成员变量的布局。但 a 离对象首部 4 字节，对象前 4 字节很可能存储的是类似虚表首址之类的东西。

为了验证该分析，在执行了第 7 行对 b 的赋值后，我们利用内存窗体修改 eax + 8 指向的内存区的值，由第 7 行赋值后的 2，改为 3，如图 4.20 所示，显示了从对象首址 0x0194f160（之前记录的 ECX 的值）开始的 12 字节。倒数第 3 字节被修改为 03，而此时执行 print 语句的打印结果正是 3，说明判断正确。而对象前 4 字节的值为 0x00149c78，疑似虚表首址。

```
0x0194F160  78 9c 14 00 01 00 00 00 03 00 00
```

图 4.20

DM4-73(c)调用了两个虚函数。第 2 行将对象首址放入 ECX 中，第 3 行从对象前 4 字节取出一个地址放入 EAX 中，而第 4 行从该地址偏移 38h，从中获取 print 的地址并 call 过去。如果虚函数 print2 像 C++那样，其入口地址的存储空间应该紧邻 print，那么存储地址应该是 eax+3ch。从第 9 行对 print2 的调用可证明这一点。

至此，我们发现，.NET 对象的这个“虚表”似乎很大，用户定义的第一个虚函数 print 的首址居然存在离头部 38h 的地方，而非 C++那样存在表起始。但 C++虚表的“真正”内容也不是只存储了虚函数地址。如 4.8.5 节中的图 4.19，表的更前方还存储了 0x4aaecc。可见，两者无本质不同，表内容都不是只存储虚函数地址，不过对象前 4 字节指向表的位置不同，但这并不妨碍通过它访问该表全部信息。事实上，即使没有定义虚函数，因为要检索类型信息，对象前 4 字节依然指向该表。对象模型见图 4.21。

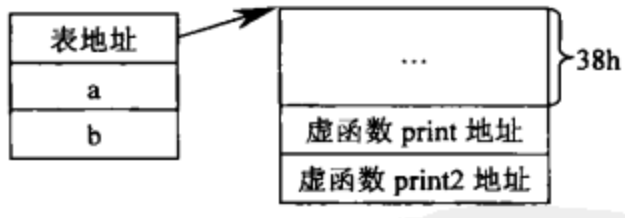


图 4.21

DM4-72

```
class Program{
    int a;
    int b;
    public virtual void print() {
      System.Console.Write(b + "\n");
    }
    public virtual void print2(){ }
    private void set(){
        a = 1;
        b = 2;
    }
    static void Main(string[] args) {
        Program c = new Program();
        c.set();
        c.print();
        c.print2();
    }
}
```

DM4-73

(a)

```
1    c.set();
2      mov  ecx, dword ptr [ebp-40h]
3      cmp  dword ptr [ecx], ecx
4      call   ffb6b010
```

(b)

```
1    private void set() {
       ...
2      mov  dword ptr [ebp-3ch], ecx
       ...
3    a = 1;
3      mov  eax, dword ptr [ebp-3Ch]
4      mov  dword ptr [eax+4], 1
5    b = 2;
6      mov  eax, dword ptr [ebp-3ch]
7      mov  dword ptr [eax+8], 2
```

(c)

```
1    c.print();
2      mov  ecx, dword ptr [ebp-40h]
3      mov  eax, dword ptr [ecx]
4      call   dword ptr [eax+38h]
5      nop
6    c.print2();
7      mov  ecx, dword ptr [ebp-40h]
8      mov  eax, dword ptr [ecx]
9      call   dword ptr [eax+3ch]
```

参考资料

1. 深度探索 C++对象模型. 武汉：华中科技大学出版社, 2007.

习 题 4

1．4.1.4 节在 VC 下采用 EDX 传递 self 指针，请试验能用 ECX 或 EAX 传递吗？并分析原因。

2．this 指针除了从寄存器传递，还可以从其他地方传递吗？请用_stdcall 修饰成员函数，并用反汇编分析 this 的传递方法。

3．请用反汇编和跟踪，分析 delete 执行过程，证明以下过程：先执行了析构函数，并向析构函数悄悄传递了对象首址，然后调用内存释放函数，释放对象。

4．分析 DM4-33 中 test2 是如何从栈上访问参数（结构体 s）的两个成员变量的。

5．如 DM4-43 那样手动计算被修改成员变量的偏移是麻烦而易犯错的，因为不仅成员变量可能较多，且考虑第 1 章提到的结构体对齐的影响，就更不容易计算偏移了。请给出一种更简单的方法，直接修改私有变量。

6．如果 DM4-43 中的粗体语句“*((int *)&o + 1) = 3;”修改为“*(int *)(&o + 4) = 3”，是否正确？3 被赋值到离头部偏移多少字节的地方？

7．在 4.7.2 节的基础上，实现最基本的插件。

提示：首先一种格式化相关的逻辑可以放入 DLL 中，它们具有相同的函数名、函数原型等，这样主程序可以利用函数名获取到相关处理函数。其次，提供一种机制，加入新的处理模块（DLL）时不用修改代码就能加载，因此必须利用 LoadLibray 动态加载（静态加载 DLL 必须修改代码）。最后，如何设计出函数接口是一个难题。一个简单而有效的办法就是，先实现出自己认可的方案（或者编写伪代码），然后从 main()函数开始，一行一行查看，如果发现为了加入一个新 DLL，必须修改主程序代码，那么就必须重构程序。遇到这种情况怎么办？根本上就是，为那些似乎个性化的功能抽象出共性的函数原型，就像 4.7.2 节对格式化函数原型的设计，然后在 DLL 中实现这个共性的函数原型，当主程序加载了这些 DLL 后，就可用函数指针调用。参考格式化是怎么通过函数指针让调用代码不修改即可调不同函数，模仿它即可。

8．试用反汇编分析 C++中的重载函数的调用，是否存在像虚函数调用一样特殊的机制？并说明原因。

9．有了反汇编调试，很多奇怪的问题都能看清。C++有太多特点，经常悄悄做了很多工作。如下代码中，e.Description()返回类型为类_bstr_t（其原型为_bstr_t _com_error::Description()），它是一个对象，而 MessageBoxW 的第 2 个参数是指针 LPCWSTR，它是一个指针。两者类型完全不匹配，为什么以下编译不报错呢？试用反汇编调试分析。

```
#include <comdef.h>
void main() {
    try {
        ...
    }
    catch(_com_error e) {
        MessageBoxW(NULL, e.Description(), L"e", MB_OK);
    }
}
```

10．4.7.6 节提到 C++ Builder 中用 dynamic 替换 virtual，实现了一种新的虚函数机制。请用反汇编调试跟踪分析，并给出说明。

11．Visual C++提供了自己的关键字，可将非虚函数甚至成员变量从 DLL 导出。请运用反汇编分析其实现机制。

12．证明定义纯虚函数将会在虚表中增加对应的槽位。

13．如果 4.8.2 节中定义的虚函数不是纯虚函数，有何影响？试分析。

14．分析引入静态成员变量对对象模型的影响。

15．自定义 A 和 B 两个类，B 是 A 的子类，然后用反汇编分析实证 4.8.5 节实验三中的推测是否正确：通过 RTTI Class Hierarchy Descriptor 能最终判断“B is A”的语义。

16．DM4-48(e)初始化对象区为 0 的方式比较冗长，每个子类的初始化函数都会清零一次，有更简洁的方案吗？

PDG

第5章

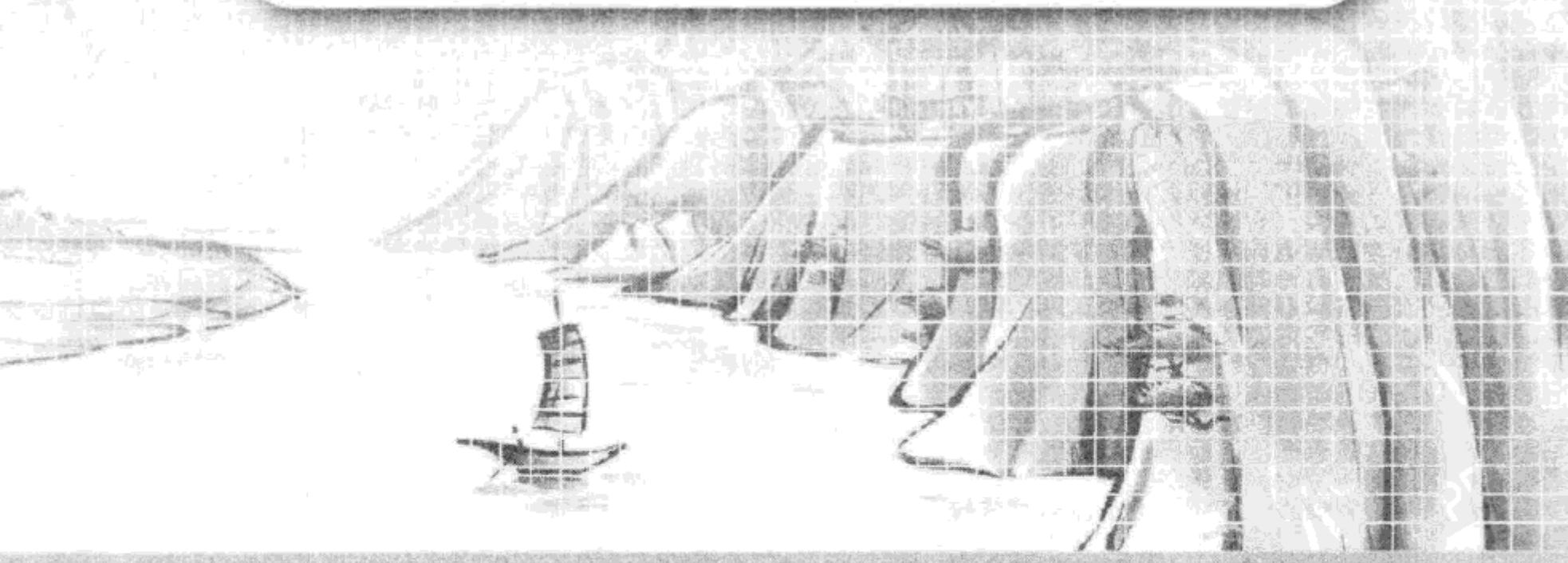
底层与抽象的混沌：一个跨平台线程类的封装、错误与进化

需准备的知识

- 整章需要第1章反汇编分析支持，最好完整阅读第4章。
- 5.1.4节需要1.3节的知识。
- 5.2.2节需要4.7.5节和4.7.6节的知识。
- 5.2.4节需要4.4.1节的知识。
- 5.4.1节和5.4.3节需要4.7.5和4.7.6的知识。

为你提供的能力

- 5.2.5节和5.2.6节提供对第7章对应设计模式的理解。
- 深入理解整体资源管理，学会一种在C++析构中实现虚函数多态的方法。
- 学会运用面向对象的各种要素，如static、virtual等，构造一个生产级的可重用线程类设计。
- 学会源代码级跨平台技巧，实现跨平台线程库。
- 充分理解猜测—实证—构建方法。



在之前我们探索了许多面向对象底层结构（如 this 指针的传递），也看到了一些语法要素的内在机制（如 static 函数），更透视了虚函数的机制等。但似乎难以将这些东西应用到实际代码编写中。通过本章，我们将利用所学习的底层机制知识以及第 1 章的调用惯例，构建一个强大的、可重用的线程类，既感受到底层的力量，也触摸到对象的强大；同时，通过一个设计 bug 的解决，展现底层机制和调试的威力，也反思语言机制对设计的影响。本章还会展示源代码跨平台技术，提供跨 Windows 和 Linux 线程类，为第 7 章奠定基础。

5.1 先学习多线程编程吧

5.1.1 概念

线程的详细概念大家可以参考任何一本操作系统的书。这里为了方便，只做一个简单介绍。CPU 是代码运行的场所，在单 CPU 时代，为了让多个程序同时在唯一的 CPU 上运行且相互不干扰，就提出了进程（后来是线程）的概念（在 Windows 系统中，我们能同时听音乐、编辑文档、下载资料等）。可以想象，CPU 就是一块黑板，一个线程就是一个写黑板的人。多个线程同时运行就是多个人写黑板，如果管理协调不好，那么一切都会乱套。简单地说，当 A 正在写黑板而轮到 B 写黑板了，系统首先对 A“点穴”停止它，然后保存 A 当前黑板的内容。在 B 用完后 A 再次使用黑板时，系统将之前保存的内容恢复到黑板上，然后“解穴”A。这样 A 根本察觉不到自己被冻结和解冻，就好像自己一个人占有黑板一样。从第 1 章学习的知识来考察，这个过程就极易理解（**大家一定要将学习时的比喻落实到技术细节上，否则无法落地。如果你发现无法对应到技术细节，在可能的情况下，一定要完成这一“比喻对应技术”过程。**）：一段代码一般都有很多函数调用，函数调用时要用栈传递参数，保存返回值，可能还要利用寄存器，这些东西可理解为黑板上记录的内容。如果系统不完成这些东西的自动保存和恢复，那么一切运行都会乱套。（依赖程序自觉退出 CPU 并自己保存状态可行吗？遇到不自觉的程序怎么办？另外，每个程序自己保存这些复杂的东西既重复又容易出错。）比如，A 线程用 ECX 传递的 this 指针是 0x123355，B 线程占用 CPU 时，它将 ECX 修改为 0x334477，那么当 A 再次占用 CPU 时，如 ECX 不恢复为 0x123355，程序访问的对象就是错误的。这一保存“黑板”状态并恢复的过程在操作系统的调度线程机制中叫 context switch。有了线程和操作系统的调度管理，我们就可以进行线程编写了。

还是从一段小代码来形象理解线程的效果吧。

```
...
getchar();
printf("hello world\n");
```

在以前的编码中，只有等到 getchar()返回后才可能打印。有了线程后，我们可在 getchar()之前构造一个线程，当该线程获得 CPU 时，即使 getchar()还未返回，也可在该线程中打印。线程让我们能够将一个大任务分解为多个“并发”的小任务。在多核时代，CPU 可更好地被利用，这也是目前大家关注的热点领域之一。

5.1.2 Windows 下的线程接口

先看下面的函数，它用于创建一个线程：

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES lpThreadAttributes,
                     DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
                     LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId)
```

lpThreadAttributes，一个与安全有关的指针，一般都不使用，默认为 NULL。

dwStackSize，线程栈的大小，如果为 0，则与主线程（即执行 main()函数的线程）的栈一样大。根据第 1 章，我们容易分析出为什么一个线程需要一个栈，因为函数调用的参数、返回值和局部变量都在栈上分配。如果多个线程共用一个栈，那么它们相互覆盖栈空间，这些数据就混乱了。

dwCreationFlags，代表创建线程后是否运行（为 0）还是悬挂（即不运行，CREATE_SUSPENDED）。如果是悬挂，用 ResumeThread 可恢复其运行。

lpThreadId，整数指针，如果成功构造线程，该整数指针指向的整数将是线程的 ID，它在整个系统中不重复，是线程的标识。

对于这个函数，如果函数成功，则返回线程的句柄，否则返回 0。这里要与 lpThreadId 的作用区分。句柄也是线程的标识，是一个整数，是在进程范围内唯一标识。ID 是在系统范围内的唯一标识。Windows 的习惯是：如果要获得使用线程对象，可用线程的 ID 获得句柄（如 threadHandle = OpenThread(threadId);），然后一切与线程相关的操作用句柄完成。这与 UNIX 只用一个整数（文件描述符）代表并使用的策略不同。

lpStartAddress 和 lpParameter 是两个最重要的参数。按照我们先猜测的习惯，先想想要启动一个线程到底需要哪些东西。（**在学习 API 时，先猜测应该有什么再到文档中印证的习惯依然是一种捷径。**）从根本上讲，一个线程需要执行一段代码。要执行一段代码需要告诉系统的是代码入口地址，自然联想到函数指针可胜任这一目标。函数指针是有类型的，它到底应该执行什么样的函数原型？或者说，重要的是应该传递什么样的参数？传一个整数、两个整数，还是浮点数？不同用处的函数的参数表千姿百态。函数指针指向的函数原型只有一个，什么样的参数可以万能呢？本质上所有的信息都是在内存中，只要传给函数这段内存的地址就可以了。因此参数应该是 void *（关于复杂参数的例子请见 5.1.4 节），所以很容易理解 lpStartAddress 函数指针指向函数的原型是 DWORD WINAPI ThreadFunc(LPVOID)。这里的 LPVOID 就是 void *，而 WINAPI 是调用惯例，通过 Visual C++的“转到定义查看”，就是_stdcall（见 1.3.9 节），该参数指向了线程函数（线程运行时执行的函数）的入口地址。此时，lpParameter 也好理解了。操作系统通过 CreateThread 知道了线程函数的地址，当它调度该线程运行时，执行 call 指令，将 EIP 寄存器指向该地址即可。但系统在 call 之前还要压栈传递该线程函数需要的参数，参数从哪里来？自然是从 CreateThread 传入并保存起来，执行线程函数时，再取出传递给它即可。这个 lpParameter 就是线程函数执行需要传入的参数 void *。下面是以上逻辑的伪代码：

```
CreateThread( …, lpStartAddress, lpParameter, …) {
    pThdDes = 分配线程描述块;
    pThdDes->funcAddress = lpStartAddress;
    pThdDes->arg = lpParameter;
    将 pThdDes 放入线程调度队列…
}
```

当调度线程运行时：

```
pThdDes = 获取当前应该运行的线程描述块
pThdDes->funcAddress(pThdDes->arg);                    //运行该线程函数
```

函数

void ExitThread(DWORD dwExitCode)

在线程函数中调用，执行后线程函数结束，线程停止运行。那么，dwExitCode 有什么用呢？当线程结束后，我们可通过该参数获得一些信息（用 WaitForSingleObject 可等待线程结束，用 GetExitCodeThread 可查询退出码），如一个线程函数计算出了 CRC 校验值，就可通过该返回码返回。更复杂的数据结构怎样返回？从之前 CreateThread 的线程函数参数的分析可了解，指针是最强大的，再复杂的数据块有了指针就能访问。这个 DWORD 类型的返回码因为与指针长度相等，所以完全可以是返回数据的头部地址！当然，如果这个数据是动态分配的，请注意释放。该函数的调用还触发了很多相关结束动作，因此我们要养成习惯，在最后调用该函数。

函数

BOOL CloseHandle(HANDLE hObject)

是 Windows 系统中的一个通用函数，操作系统中的线程、进程、事件、信号灯等内核中的对象都用一个整数（称为句柄）代表，相当于 ID。调用该函数可能导致关闭内核对象、释放其相关联的资源的发生。内核对象采用引用计数释放，调用相关函数，引用计数会增一，CloseHandle 导致引用计数减一，计数归零则真正释放对象。如果有多个程序、多个模块使用一个内核对象，就很好管理，“关注”则引用计数增一，不想用了，CloseHandle 则减一。线程结束并非意味着线程对象从内核中释放，如 ExitThread 传递的退出码可能被查询，只有用户明确调用 CloseHandle 才会释放内核线程对象。

函数

DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds)

等待一个内核对象相关事件的发生，如果被等待者是线程、进程，则等待两者结束；如果是 Event（事件，一种同步机制），则等待事件状态变为真；等等。

hHandle 是被等待的句柄，对线程就是 CreateThread 返回值。dwMilliseconds 代表要等待的时间，单位为毫秒，宏 INFINITE 代表无限等待，其值为-1。返回值代表等待的结果，由 3 个宏代表：WAIT_OBJECT_0 代表在指定时限内等待成功；WAIT_TIMEOUT 代表在指定时限内等待失败。WAIT_ABANDONED 比较奇怪，等待就是成功/失败两种情况嘛，这个第 3 种情况是什么呢？大家想，如果我们等红绿灯转换，除了绿灯亮或不亮，还有一种情况：等待中突然一货车将灯撞坏了，我们肯定不会再等下去。这时既非成功，也非失败，而是“异常”。因为被等待对象被异常地销毁了。WAIT_ABANDONED 就表示该情况，比如等待时对象被 CloseHandle 关闭了。

5.1.3 第一个线程程序

现在开始令人兴奋的旅程。DM5-1 建立了一个线程，休眠 5 秒，然后打印一句话。

注意，threadProcedure 是线程函数，线程运行将执行它。第 3 行中的_stdcall 说明线程函数的调用惯例是标准调用。如果不写，C 语言默认是 cdecl 方式，那么在 11 行编译器将报错：

error C2664: "CreateThread" : 不能将参数 3 从"unsigned long (__cdecl *)(void *)"转换为"LPTHREAD_START_ROUTINE"

查看符号定义

在 Visual C++中查看不认识的标识符：右键单击标识符（变量、函数或宏）上，选择快捷菜单的“转到定义”。

我们要学会分析错误信息，第 13 行 CreateThread 的第 3 个参数就是 ThreadProcedure 函数指针，其原型是 unsigned long (__cdecl *)(void *)，不能转换为 LPTHREAD_START_ROUTINE，说明

DM5-1

```
1   #include <stdio.h>
2   #include <windows.h>
3   unsigned long __stdcall threadProcedure(void * arg) {
4       Sleep(5000);
5       printf("hello, I run in a thread\n");
6       ExitThread(0);
7       return 0;
8   };
9   void main() {
10      unsigned long threadID;
11      HANDLE threadHandle;
12      printf("please input\n");
13      threadHandle = CreateThread(NULL, 0, threadProcedure, NULL, 0, &threadID);
14      getchar();
15      printf("I am in main thread\n");
16      getchar();
17      if (threadHandle != NULL) {
18          WaitForSingleObject(threadHandle, INFINITE);
19          CloseHandle(threadHandle);
20      };
21  }
```

LPTHREAD_START_ROUTINE 和 ThreadProcedure 的函数原型不同。怎样看到该原型是什么呢？我们就要运用 IDE 的功能了，将 LPTHREAD_START_ROUTINE 写入到程序中，右键单击之，在弹出的快捷菜单中选择“转到定义”，看到其等价的一个定义 PTHREAD_START_ROUTINE，如法查看其定义：

```
DWORD (WINAPI PTHREAD_START_ROUTINE) ( LPVOID lpThreadParameter);
```

用同样的方法，可知 WINAPI 即_stdcall，说明该函数是 stdcall 调用惯例。不在 ThreadProcedure 前定义_stdcall 导致函数指针类型不匹配。第 2 章中已经讨论了调用惯例不匹配时的灾害性影响，因此加上_stdcall 即可。

第 13 行构造了线程，但此时并非意味着线程开始执行，只是将其放入了调度队列中等待调度。第 4 个参数为 NULL，因为线程函数 threadProcedure()中实际不需要参数，所以随意传递个 NULL。第 17 行判断是否成功构造了线程，如果是，则 WaitForSingleObject 一直等待其结束，之后会调用 CloseHandle 释放该内核对象。本程序如果在第 14 行等待 5 秒以上再回车，那么线程函数将先执行打印"hello, I run in a thread"，然后第 15 行才打印。反之，则主线程先打印。如果执行第 18 行时线程函数还未结束（如第 14、16 行回车非常迅速），它将一直等待线程函数执行完毕，线程结束才返回。主线程（执行 main 的线程）等待子线程（其他的线程）结束很重要，在早期的 Windows 系统中如不执行，将出现 bug。

5.1.4 那些复杂的参数和 bug

5.1.3 节的线程没有干什么有价值的工作。从 5.1.2 节知道，CreateThread 的第 4 个参数将在线程运行时作为线程函数的参数传递给线程函数。如何给线程传递一个复杂的参数？比如，线程要完成如下工作：将文件名 f1、f2 和 1 字节的 key 传给线程函数，它将文件 f1 的每字节读入，与 key 进行异或运算，再写入一个新的文件 f2，完成简单加密。如何将这 3 个参数传递给只接收一个参数的线程函数？5.1.2 节中提及“再复杂的信息也是一段内存，只要传递该内存地址就可访问该信息”，因此我们将 f1、f2、key 三个参数放入一个结构体，结构体本质上就是一块连续内存（见 1.4.2 节），

所以只要将结构体指针传递给线程函数，函数中再将其强制转换为结构体指针，即可访问到。

```
struct Info{
    char * f1;
    char * f2;
    char key;
};
…
struct info;
info.f1 = "C:\\tmp.txt"
info.f2 = …
//&info 传递结构体地址
threadHandle = CreateThread(NULL, 0, threadProcedure, &info, 0, &threadID);
…
unsigned long __stdcall threadProcedure(void * arg) {
    //将 CreateThread 传入的地址强转为 struct Info *，访问之
    struct Info *pInfo = (struct Info *)arg;
    pInfo->f1…
}
```

为了简化，我们将 f1、f2 变为两个数组，将 f1 中的字节与 key 异或后写入 f2 的数组。结构体定义如下，ARRAY_LEN 是一个宏：

```
struct Info {
    char f1[ARRAY_LEN];
    char f2[ARRAY_LEN];
    char key;
};
```

代码见 DM5-2，工程见“code\第 5 章\ThdXor”。

DM5-2

```
1   unsigned long __stdcall threadProcedure(void * arg){
2       struct Info * pInfo = (struct Info *)arg;
3       for (int i = 0; i < ARRAY_LEN; i++) {
4           pInfo->f2[i] = pInfo->f1[i] ^ pInfo->key;
5       }
6       printf("xor result: \n");
7       for (int i = 0; i < ARRAY_LEN; i++) {
8           printf("%d ", pInfo->f2[i]);
9       }
10      printf("\n");
11      ExitThread(0);
12      return 0;
13  };
14  void main() {
15      Info info;
16      for (int i = 0; i < ARRAY_LEN; i++){ info.f1[i] = i; }
17      info.key = 10;
18      unsigned long threadID;
19      HANDLE threadHandle;
20      threadHandle = CreateThread(NULL, 0, threadProcedure, &info, 0, &threadID);
21      if (threadHandle != NULL) {
22          WaitForSingleObject(threadHandle, INFINITE);
23          CloseHandle(threadHandle);
24      };
25  }
```

第 20 行，将加密需要的结构体 info 的指针通过 CreateThread 传递给线程。第 2 行将传入的这个指针又强制转换成 struct info *，这样线程函数就可用它访问被加密的数组 f1 及 key 了。第 3～5 行完成加密，并存入数组 f2。第 6～9 行将加密结果打印出来。

在线程传递参数方面有一些比较微妙的东西，不知大家能否看出 DM5-2 中存在的潜藏问题？来看代码 DM5-3，见“code\第 5 章\ThdXor-err”。第 14～22 行定义了一个 CreateThd 来产生线程，根据传入的 start 的不同，初始化 f1 数组的值不同。第 25～26 行分别构造了两个线程，第 27～33 行等待两个线程结束。

DM5-3

```
1   unsigned long __stdcall threadProcedure(void * arg) {
2       struct Info * pInfo = (struct Info *) arg;
3       for (int i = 0; i < ARRAY_LEN; i++) {
4           pInfo->f2[i] = pInfo->f1[i] ^ pInfo->key;
5       }
6       printf("xor result: \n");
7       for (int i = 0; i < ARRAY_LEN; i++) {
8           printf("%d ", pInfo->f2[i]);
9       }
10      printf("\n");
11      ExitThread(0);
12      return 0;
13  };
14  HANDLE CreateThd(char start) {
15      Info info;
16      for (int i = 0; i < ARRAY_LEN; i++) {
17          info.f1[i] = i + start;
18      }
19      info.key = 10;
20      unsigned long threadID;
21      return CreateThread(NULL, 0, threadProcedure, &info, 0, &threadID);
22  }
23  void main() {
24      HANDLE h1, h2;
25      h1 = CreateThd(1);
26      h2 = CreateThd(0);
27      if (h1 != NULL) {
28          WaitForSingleObject(h1, INFINITE);
29          CloseHandle(h1);
30      }
31      if (h2 != NULL) {
32          WaitForSingleObject(h2, INFINITE);
33          CloseHandle(h2);
34      };
35  }
```

运行该程序，结果出现异常，此时查看输出的结果是“7, 7, -38, -127, -125, -1, -1, …”。对比 DM5-2 的运行结果“10, 11, 8, 9, 14, 15, 12, 13, 2, 3”分析，这次打印出的数据竟然没有一个是我们希望的输出。进一步分析代码，发现出错是第 28 行执行 WaitForSingleObject 出现的内存访问违规。怎么回事？这时底层基础知识的力量就展现出来了。请构想出函数调用过程以及相关栈布局，在我们的大脑中浮现执行的境况。

从第 21 行可知，传递给线程的参数地址是 CreateThd 中的局部变量 info 的地址，一个局部变量的地址传递给函数之外的地方运用（线程函数中），这本身就有问题。两次调用 CreateThd 函数的主线程栈映像如图 5.1 所示。

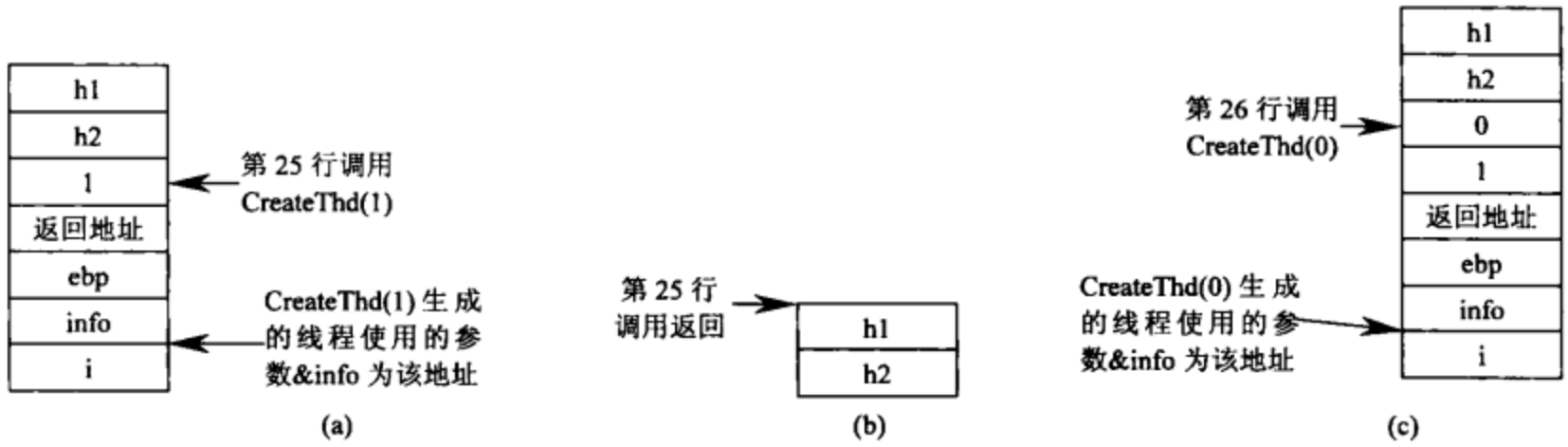


图 5.1　主线程栈映像

图 5.1(a)是 25 行调用 CreateThd 未返回前的映像；图 5.1(b)是返回后，下部因 CreateThd 使用的栈空间被平衡了，所以没被使用（当然，这段内存依然存在）；图 5.1(c)是第 26 行调用 CreateThd 未返回时的映像。我们发现，在两次调用 CreateThd 时，均由第 21 行传递&info 给 CreateThread，这两次传递的其实是一个地址。也就是说，当两个子线程开始运行时，它们操控的是同一块内存——主线程栈上属于 info 的那块空间（跟踪进 CreateThd，查看两次调用时 info 的地址即可证明）。其操控结果必然是混乱的。这也可解释为什么输出不是我们希望的值。

但为什么主线程在执行 28 行 WaitForSingleObject 时，出现了异常呢？对于多线程还真不好调试，我们需要在大脑中进一步冥想。继续构想主线程中栈映像，见图 5.2。

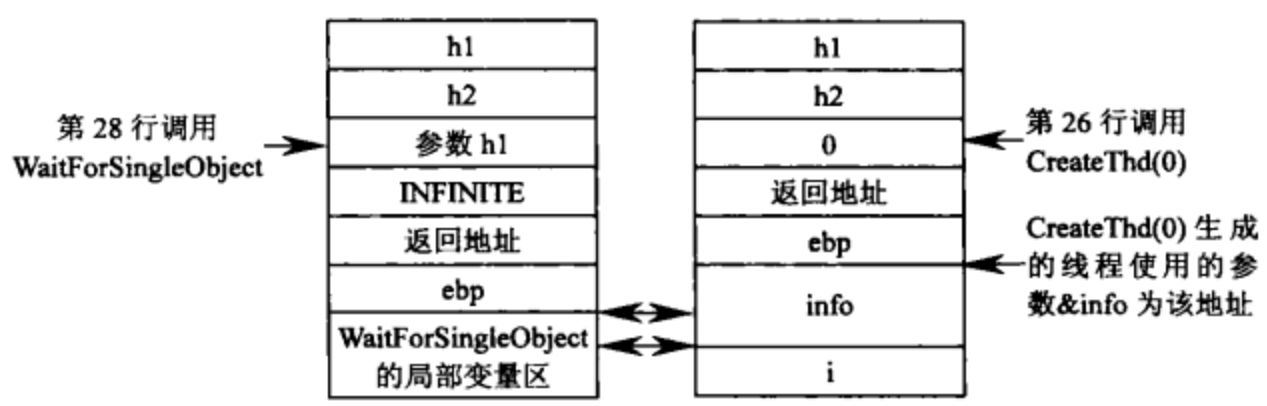


图 5.2

图 5.2 中，左边是调用 WaitForSingleObject 时的栈映像，最下方是该函数自己的局部变量区。图 5.2 右边是调用 CreateThd 时的栈映像，其中 info 的地址被传递给子线程访问。中间两条双向箭头指示的区域就是 info 和 WaitForSingleObject 局部变量重叠的区域。这块区域被两个子线程写入异或后的数据（DM5-3 第 4 行对 info 成员数组 f2 的写操作）。当这三个线程同时运行时，该交叠区域就是一片混乱。这个混乱体现在子线程输出结果上就是一些奇怪的打印值。对 WaitForSingleObject 而言，其局部变量被两个子线程“乱写”，必然导致其执行错误。

为了印证这个猜想，我们只需简单地将第 15 行的关于 info 的定义放到 CreateThd 外，成为全局变量（这样 info 的内存不在栈上分配，不会与 WaitForSingleObject 的局部变量“交叠冲突”），就不会在 WaitForSingleObject 时出现异常了。进一步，可以通过某种合适方式让主线程延迟执行 WaitForSingleObject，等两个子线程完毕后再执行 WaitForSingleObject（从而避免子线程“乱写”该函数局部变量区），那么这个异常应该不会发生了。**（在第 26 行后增加一个 Sleep 调用吗？这是错误的，请读者自己思考。）**我们在第 26 行后增加一个循环，循环中为一个全局变量赋值（请在 Debug 版下实验），将该循环的次数调整到足够大时（即大到两个子线程已经执行完毕），我们会发现，WaitForSingleObject 不会再报错。当然，由于两个子线程都操作同一内存，显示结果还是错误的。这里蕴含着一个内存相关的危险，如果 CreateThd 在一个子线程中被调用，那么 info 变量就是在该

线程栈上分配。如调用 CreateThd 的线程结束，info 所在栈即被释放，此时如果 CreateThd 生成的线程正在运行，那么它对 info 的访问必然导致对无效内存的访问异常（请设计程序验证）。

怎样修改这个程序？就是为每个线程分配一个内存，该内存存储线程函数的参数，然后将该内存的地址传递给 CreateThread，见 DM5-4，代码见“code\第 5 章\ThdXor-err-fix”。

DM5-4

```
1   void PrintResult(struct Info * pInfo) {
2       printf("xor result: \n");
3       for (int i = 0; i < ARRAY_LEN; i++) {
4           printf("%d ", pInfo->f2[i]);
5       }
6       printf("\n");
7   }
8   HANDLE CreateThd(char start, struct Info ** ppInfo) {
9       struct Info * info = (struct Info *)malloc(sizeof(Info));
10      * ppInfo = info;
11      for (int i = 0; i < ARRAY_LEN; i++) {
12          info->f1[i] = i + start;
13      }
14      info->key = 10;
15      unsigned long threadID;
16      return CreateThread(NULL, 0, threadProcedure, info, 0, &threadID);
17  }
18  void main() {
19      HANDLE h1, h2;
20      struct Info * p1, *p2;
21      h1 = CreateThd(1, &p1);
22      h2 = CreateThd(0, &p2);
23      if (h1 != NULL) {
24          WaitForSingleObject(h1, INFINITE);
26          CloseHandle(h1);
27          PrintResult(p1);
28          free(p1);
29  };
30  if (h2 != NULL) {
        //和 24~27 行类似，等待结束，并打印 p2，然后释放之
```

先将线程函数中打印的部分移到函数 PrintResult 中（见第 1～7 行），因为多线程执行的不确定性必然导致两个线程打印结果混在一起，为了分别清晰地打印，在线程结束后再调用 PrintResult，见第 26 行。为了给每个线程一个专属的存放线程参数的空间，在 CreateThd 中分配了 info 结构体，见第 9 行，同时该函数多了一个参数 struct Info**（关于指针的指针请看 2.3 节），将函数中分配的内存传递给函数之外，便于打印。第 26～27 行在等待 h1 指向的线程结束后，打印加密结果，并释放 CreateThd 中分配的结构体内存（否则内存泄漏）。

5.2 简单、重用，让我们构造线程类吧

看看 DM5-1，我们发现，线程编写比较烦琐，且容易出错：① 线程函数要记住最后调用 ExitThread；② 复杂的 CreateThread 函数；③ 等待线程的结束以及必须调用 CloseHandle；④ 参数传递的微妙错误（见 5.1.4 节）；⑤ 过程式操作，没有体现自己操作自己数据封装的优越。这里面有很多似乎重复、易错的代码，我们能解决这些问题吗？软件设计就是要让大家偷懒、轻松、少犯错误。一切的一切调动起来，见招拆招，开始我们艰苦而又兴奋的对象设计之旅。问题④、⑤见

5.2.1 节和 5.2.2 节；问题②见 5.2.3 节；问题①见 5.2.5 节；问题③见 5.2.4 节。

5.2.1 无赖的尝试，原来是它——static

从面向对象的观点来看，一个对象应完成线程要做的所有工作，这是很自然的事。而对象 A 做加密，对象 C 做网络收发，这也非常符合虚函数的多态性。我们先来设想需要的接口：

- Run，启动线程对象的运行。
- WaitFor，等待自己结束。
- 析构函数，在其中等待线程结束，并调用 CloseHandle 释放线程资源。

想象中是这样来使用线程对象：需要一个线程做特定的事情时就定义一个线程类，如 MyThread，然后如 DM5-5 使用。

DM5-5

```
MyThread* thread;
thread = new MyThread();
Printf("please input \n");
thread->Run();
delete thread;    //在析构中 WaitFor
```

我们对着它来写写心目中的线程类。Run()方法既然启动了线程，那么应该调用 CreateThread。FHandle 和 FThreadID 作为其成员变量。

```
void Run() {
    FHandle = CreateThread(NULL, 0, execute, arg, 0, &FThreadID);
}
```

关键是线程函数传入什么？arg 又传入什么？不同线程执行动作不同，execute 不同，且 arg 也不同。那么，是否将 Run()方法加入两个参数，将 execute 和 arg 作为参数传入呢？如果这样就违背了面向对象自己操作自己数据的精神，且 5.1.4 节中参数传递的问题也没有解决。我们好好分析一下 execute，其参数表相同，均为 void *。对不同类有自己不同的操作，这正是虚函数的特征，如果将 execute 变成基类的虚函数，传递给 CreateThread，而子类覆盖之，就好像解决这个问题了。

```
class TThread{
    unsigned long FThreadID;
    HANDLE FHandle;
    virtual unsigned long Execute(void * arg) { return 0;};
    void Run() {
        FHandle = CreateThread(NULL, 0, execute, arg, 0, &FThreadID);
    }
}
class MyThread : public TThread{
    unsigned long Execute(void * arg) { … };                    //子类覆盖父类虚函数
}
```

这段代码还有 arg 没有确定，不妨先将其设定为 NULL，然后编译该代码，提示如下错误：

"TThread::Execute"：函数调用缺少参数列表；请使用"&TThread::Execute"创建指向成员的指针。

错误指向了 CreateThread 所在行，意味着没有将 Execute 的指针成功地传递给 CreateThread。想想看，类的函数指针 Execute 与 C 语言函数指针的表示方式也许是不同的。C 语言只需函数名就可指明是哪个函数，有了类自然要指出是哪个类的函数，否则编译器无法知道是哪个类的 Execute 函数。错误信息“请使用"&TThread::Execute"”正是包含了类名的指针，于是将代码修改为

```
CreateThread(NULL, 0, &TThread::Execute, NULL, 0, &FThreadID)
```

可依然错误，如下：

"CreateThread"：不能将参数 3 从"unsigned long (_ _thiscall TThread::*)(void *)"转换为"LPTHREAD_START_ROUTINE"。

错误依然指向 CreateThread，这个错误与 5.1.3 节的错误一样，即传入的第 3 个参数，函数指针，其函数原型 unsigned long (_ _thiscall TThread::*)(void *)不是要求的原型。仔细分析，我们没写调用惯例，_stdcall。好，将父子类的 Execute 都加上_stdcall，再编译，还是错误：

"CreateThread"：不能将参数 3 从"unsigned long (__stdcall TThread::*)(void *)"转换为"LPTHREAD_START_ROUTINE"。

Execute 的函数原型到底与要求有何区别？错误信息中的 TThread::代表什么？不妨看看之前的错误信息，还多了一个提示__thiscall。this 的 call 就在第 4 章分析过类的静态和非静态函数区别时，非静态函数多了个悄悄传递的“this”。悄悄传递 this 的 call 就是__thiscall 吗？不管是不是，Execute 确实多传递了一个 this 指针，因此与线程函数原型不同，所以无法传递给 CreateThread。那么，可将其强制转换后传递 CreateThread 吗？我们在第 1 章提到过，函数指针的强制转换基本都是错误的。我们甚至可以想象出，如果传递给了 CreateThread，当操作系统调用 Execute 时，是不会传递 this 的。因此对 Visual C++来说，调用 Execute 时，ECX（传递 this 的寄存器）是随机数，通过 ECX 求偏移量访问对象成员变量必然出错。既然找到了 static 这个根源，我们是否一切都搞定了？

5.2.2 可爱的 virtual 和可恨的 this

将 static 加在 Execute 前，还是错误：

"virtual"不能和"static"一起使用。

为什么不能？我们在第 4 章有明确分析：对于 C++，如果不从对象前 4 字节获得虚表指针，是没有其他机制获得它的。static 函数与对象无关，可用“类名::函数名”直接调用，所以无法获得虚表指针，自然无法调用虚函数。我们定义一个 static 函数 Wrapper，与线程函数原型一样，传递给 CreateThread，再在该函数中调用 Execute 就可以了。代码如下（注意粗体部分代码）：

```
class TThread {
    unsigned long FThreadID;
    HANDLE FHandle;
    static unsigned long __stdcall Wrapper(void * arg) {
        Execute(arg);
    }
    virtual void Execute(void * arg) { return 0;};
    void Run() {
        FHandle = CreateThread(NULL, 0, Wrapper, arg, 0, &FThreadID);
    }
}
```

编译结果是 Wrapper 无法调用 Execute。静态函数中无法调用非静态函数，第 4 章中分析过原因，因为调用非静态函数需要给该函数传递 this，可静态函数没有参数传递 this，它无法获得 this，自然无法给非静态函数传递，即无法调用非静态函数了。

怎么办？难道一切努力都是泡影？不妨用倒推方式看看，CreateThread 需要的线程函数是 C 语言风格的，没有悄悄传递 this 指针，这个无法改变，所以静态的 Wrapper()函数是必须的。要实现多态，就必须在 Wrapper()中调用虚函数 Execute()。虚函数是非静态的，意味着必须在静态函数中

调用非静态函数。如果静态函数能够获得对象的首部地址（即 this 指针），就可以主动传递给非静态成员函数，且能获得虚表访问虚函数了。**（这个问题就搞定了。）**Wrapper()从哪里拿到对象首部地址呢？不要忘了 Wrapper 是线程函数，它有一个万能参数 void *（5.1.4 节中的复杂参数都是通过它传递地址的）。我们如法炮制，将对象首址通过该参数传入就可以了。代码如下：

```
class TThread {
    unsigned long FThreadID;
    HANDLE FHandle;
    static unsigned long __stdcall Wrapper(void * arg) {
        TThread * pobj = (TThread *)arg;
        pobj->Execute();
    }
    virtual void Execute() { return 0; };
    void Run() {
        FHandle = CreateThread(NULL, 0, Wrapper, this, 0, &FThreadID);
    }
}
```

CreateThread 的第 4 个参数传入 this（即对象首址），而 Wrapper()的 arg 参数就是 CreateThread 中传递的这个 this 指针，理所当然地将它强制转换为线程类指针，pobj = (TThread *)arg，这样就可用它访问虚函数 Execute 了，pobj->Execute()。

回顾 5.2 节开始提出的问题④和⑤，这两个最头痛的问题其实已经完美解决了。因为问题⑤的解决，问题④也自然解决：我们不需要为线程参数专门分配内存然后释放，线程需要的所有参数都作为对象成员变量即可。还是来看 5.1.4 节的对象版解决方案：

```
class EncThd : public TThread {
  public:
    char f1[ARRAY_LEN];
    char f2[ARRAY_LEN];
    char key;
  protected:
    virtual void Execute() {
        for (int i = 0; i < ARRAY_LEN; i++){
            pInfo->f2[i] = pInfo->f1[i] ^ pInfo->key;
        }
    }
}
...
void main(){
    EncThd * thd = new EncThd();
    // 对 Thd->f1 赋值设定要加密的数组内容，并设定 Thd->key
    thd->Run();
    delete thd;
}
```

这段代码非常简洁优雅，满足了“对象自己操作自己数据”的原则：Execute 在线程中操作了自己对象上的成员变量 f1、f2 和 key。这里提一个问题：可否将上面 main()函数的第 1 行改成 EncThd thd，而不用 new EncThd()？**（从对象内存分配位置及操作它的安全性考虑，考虑 5.1.4 节的错误。）**到此为止，我们已经可以像 DM5-5 中那样通过覆盖 TThread 子类的 Execute()函数，使线程编码简

化。回顾这两节的探索过程，无不彰显着底层的力量：如果我们对对象模型和函数调用惯例不知其所以然，是不可能有这个满意的结果。现在需要完善 TThread 类，让它能尽快跑起来，并且看看它怎么解决 5.2 节开始的其他几个问题。

5.2.3 私有、保护、公有、只读、纯虚函数，一个都不能少

来看看 TThread 类的定义，并分析其中缘由，见代码 DM5-6。

DM5-6

```
1   class TThread {
2     private:
3       unsigned long FThreadID;
4       HANDLE FHandle;
5       static unsigned long __stdcall Wrapper(void * arg);
6     protected:
7       int FReturnValue;
8       virtual void Execute() = 0;
9     public:
10      TThread();
11      ~TThread();
12      bool Run();
13      void WaitFor();
14      int GetThreadID();
15      HANDLE GetHandle();
16      int getReturnValue() { return FReturnValue; };
17  };
```

第3～4行，线程标识FThreadID和线程Handle，这两个变量应该是私有的，因为不想被外部直接操控。而且这两个变量不应被修改，但可能被外部读取（用它们可操控线程），所以我们提供两个公有的方法读取它们，见第 14～15 行。

Execute()函数是被线程调用的，只应该在线程中被调用执行，所以肯定不能是公有的。如果是私有的，子类又无法覆盖它了（私有对子类不可见），所以应该是保护的，见第 8 行。TThread 并不知道线程函数执行什么逻辑，所以无法实现 Execute()函数，因此将它设定为纯虚函数 Execute()=0。设定为纯虚函数有一好处，就是强迫子类覆盖它。如果不覆盖，在 new 子类对象时，编译器会报错。因为C++无法 new 包含纯虚函数的类。

是否应该实例化包含纯虚函数的类

该问题似乎很奇怪。如果实例化，当调用纯虚函数时将调用一个不存在的函数，因此 C++禁止实例化这种类。笔者也是后来用到一种叫 Template 的技术才明白 Delphi 为什么支持构造这种类：用户可将自定义窗体作为组件注册，拖拽即可重用。这必须实例化模版子类（才能可视化）。因重用性可能在模版类中定义纯虚函数，拖拽后在子类中覆盖它。此时因未调用纯虚方法所以不出错。Delphi 将规避调用纯虚函数的责任交给程序员，以灵活性换取部分安全性。这种权衡常体现在不同语言对同一问题的不同处理：如 5.4 节展示了构造函数、析构函数中调用虚函数的不同策略；C++采用了相对 Java、Delphi、C#等更安全但死板的做法。世界精彩，取舍不同。

退出码的设定是通过成员变量 FReturnValue 传递给 ExitThread 的。只能由线程函数执行，所以它不能放在公有域。但因要在子类 Execute 中访问，所以不能是私有的，最终它放在了保护域，见第 7 行。因退出码可能被外部代码读取，所以提供了公有读函数，见第 16 行。

启动线程的 Run()方法和等待结束的 WaitFor()方法均会被外部调用，所以为公有的。

WaiFor()实现如下：

```
void TThread::WaitFor(){
    if (FThreadID == 0)
        return;
    WaitForSingleObject(FHandle, INFINITE);
};
```

如果成功构造了线程（FThreadID != 0），则调用 WaitForSingleObject。这句代码也很好地反映了“自己操作自己数据”的思想。因为 WaitForSingleObject 等待的句柄是 FHandle，对象的成员变

量无需用户从外部作为参数传入，不会出现过程式编程传错的缺陷。

如果我们设计时对哪些需要外部访问不清楚，可全放入私有域。需要外部访问，再移入公有域，需要给子类使用，再移入保护域。

5.2 节开始的问题②，复杂的 CreateThread 参数用 Run()方法已经很好地解决。接下来，5.2.4 节将解决问题③，5.2.5 节将解决问题①。

5.2.4 析构中释放资源

问题③：

必须等待线程结束以及必须调用 CloseHandle。

线程是一种资源，当它结束时，代表它的线程对象也应该结束其生命。或者倒过来说，当线程对象要释放掉时，它应该等待它代表的线程结束。在第 4 章我们已经领略了“在构造函数中分配资源，在析构函数中释放资源”的思想（FileStream 类就是一个例子），如果我们将这一思想运用到 TThread 的析构函数中，让它等待线程结束，再 CloseHandle，岂非问题③就搞定了？代码如下：

```
TThread::~TThread() {
    if (FThreadID != 0) {
        WaitFor();
        CloseHandle(FHandle);
    }
};
```

这样只要记住执行一条语句“delete thd;”来释放线程对象，就会自动调用 WaitFor()等待线程结束，并用 CloseHandle()关闭线程资源，而不需如下代码分两次执行该过程：

```
thd->WaitFor();
delete thd;
```

在析构函数 WaitFor()中，避免了忘记调用 WaitFor()的错误。

细心的读者会问：既然在析构中释放资源，为什么不在构造中分配资源，也就是在构造函数中调用 CreateThread 呢？这个问题请参考 5.4.2 节的讨论，自行分析。

5.2.5 我们发现了一个设计模式

最后来看问题①：

线程函数要记住最后调用 ExitThread。

既然该函数的调用是在线程函数中调用的，我们将视线聚焦到 Wrapper()函数吧，见 DM5-7。

DM5-7

```
1  unsigned long __stdcall TThread::Wrapper(void * arg) {
2      TThread * Thd;
3      Thd = ((TThread *) arg);
4      Thd->Execute();
5      ExitThread(Thd->FReturnValue);
6      return Thd->FReturnValue;
7  };
```

第 4 行执行了子类虚函数 Execute()，之后应该调用 ExitThread()退出线程了。但程序员完全可以不管 ExitThread()的调用，因为基类 TThread 的 Wrapper()函数的第 5 行调用了 ExitThread()。将子类 Execute()函数中设定的 FReturnValue 传递给该函数。我们在基类中帮子类完成了它本来必须记住的工作——调用 ExitThread！（**子类完全不知道基类调用了它，编写子类的程序员是多么幸福和轻松啊。**）

这简单的 6 行代码，它不简单，里面有一个非常有用的设计模式——“模板方法模式”（Template

Method Design Pattern)。(**原来我们也可以发现模式。**)对 DM5-7 做点抽象，如下：

```
执行确定性的逻辑（指针强制转换）；
执行不确定的逻辑，调用虚函数，调到子类覆盖的版本（执行 Execute）；
执行确定性的逻辑（调用 ExitThread）；
...
```

有些场合，在父类中，我们确定了整个算法过程（先强制指针转换，再执行线程函数，然后执行 ExitThread()），但在某些步骤无法确定子类中到底要做什么工作（线程函数要执行什么逻辑，加密还是打印），那么将这些子类要做的工作定义为父类中的虚函数（Execute），在父类中实现这套算法的骨架（Wrapper()函数），在其中调用虚函数即可。骨架代码中（即上面的抽象描述），粗体所示部分通过调用子类虚函数覆盖版本，将不确定性工作推给子类完成（当前线程实例是 TThread 的哪个子类则调用该子类的 Execute）。而确定逻辑则在父类的骨架代码 Wrapper()中实现，为非粗体部分。可以用虚虚实实来形容这个模式，虚的是某些步骤调用虚函数，实的是父类中确定的骨架函数流程及代码。

这样做的好处是什么？重用了父类中实现的算法骨架。在 TThread 的子类中，我们不用重复地写强制指针转换及 ExitThread 的代码，这些流程已经固化在父类的 Wrapper()函数中，只需在子类中去覆盖相关虚函数，就完成了重用。

再举个现实中的例子，我们要撰写一个关于学校报到的代码。报到中有共同的流程手续，如先交钱，再用缴费证明去领取宿舍钥匙，然后到系上注册。但这三步曲中假定系上注册的过程因不同系而不同（如办公地点不同，一些手续环节都有差异），我们如何让不同系学生的报到都走这三步曲？撰写一个父类，它的一个函数实现了三步曲的逻辑。只是因为第三步涉及的每个系不同，那么将该步骤定义为一个虚函数，让子类覆盖。不同系定义成不同子类，覆盖第三步对应的虚函数。当调用父类的这个骨架函数时，所有系的学生注册都是按照先交钱、后领钥匙、再系上报道的过程。我们重用了整个算法。

除了重用的好处，还带来什么？因为公共部分是在父类的骨架函数中实现的，不论将来发现其中有 bug，还是要改变算法，只需修改骨架代码，一次修改处处修改（所有子类共享了这个改动）。如果这段算法是用复制再修改的方式重用，有几个系，就需要修改几次，几乎是不可维护的（谁都难以记起这段代码复制了几次，复制到哪里了）。

这个模式在哪里有很好的运用？大家可阅读图形界面类的封装框架，在 Delphi 中是 VCL 框架，在 Visual C++中是 MFC 框架，它们大量运用了该模式：在父类中定义虚函数，并在父类中调用，然后在子类覆盖这些虚函数。第 7 章也运用了这种模式。至此为止，TThread 的第一个版本分析完毕，工程参见“code\第 5 章\ThreadClass_Simple”。

5.2.6 我关心，你通知——我们的第二个设计模式

用现在的接口，如果一个主线程要等待子线程结束，只能用 WaitFor()阻塞等待。但是很多时候主线程是无法阻塞的，如 Windows 中的窗体程序，主线程是在一个循环中获取消息，再分派执行。一旦阻塞，消息无法处理，窗体将丧失所有功能响应（如我们在按钮单击事件中执行一个死循环），我们需要一个接口来通知外界“线程结束了”。由于线程结束时客户代码将做什么工作是无法确定的，所以必须设计一个可以扩展的接口。

为什么不直接在线程 Execute()中撰写该结束逻辑的代码呢？因为在不同环境下，同一个线程执行完毕后，可能要做的事不同。这意味着不可能将该段不确定的代码放入线程类，否则在不同执行

DM5-8

```
1   class TThreadListener{
2     public:
3       virtual void NotifyThreadFinished(TThread * Sender) = 0;
4   };
5   unsigned long __stdcall TThread::Wrapper(void * arg) {
6       TThread * Thd;
7       Thd = ((TThread *) arg);
8       Thd->Execute();
9       if (Thd->FListener != NULL)
10          Thd->FListener->NotifyThreadFinished(Thd);
11      ExitThread(Thd->FReturnValue);
12      return Thd->FReturnValue;
13  };
```

环境下，必须修改线程类实现。（**当然，模板方法模式也可提供解决办法。**）因此，这段不确定逻辑应该放入另外的类 A 中，从而与线程类隔离。这样线程结束时要调用不同代码，将只改动类 A，不会改动线程类自身。让线程类用同一个函数名调用不同的代码，这就是虚函数的长项，见 DM5-8。

我们将定义一个 TThreadListener 类，在其中定义一个虚函数，代表线程结束 NotifyThreadFinished，见第 1～4 行。然后在 Wrapper()函数中，执行了 Execute()后调用：判断客户是否注册了 Listener 到线程类的 FListener 成员变量中，如果是，则调用其 NotifyThreadFinished 虚函数，见第 9～10 行。注册不同子类，将执行它们各自的覆盖版本，从而实现了多态：将不同结束代码的不确定性封装到了 TThreadListener 子类中。

FListener 变量类型为 TThreadListener *，在线程构造函数中传入并赋值，代码如下：

```
TThread::TThread(TThreadListener * Listener) {
    FListener = Listener;
    ...
};
```

以下代码展示了生成一个 TThreadListener 子类，并将其实例注册到线程中：

```
class TSimpleListener : public TThreadListener {
  public:
    void NotifyThreadFinished(TThread * Sender) {          //子类覆盖了该虚函数
      printf("thread finish\n");
    }
};
...
void main() {
    TPrintThread * Thd;
    Thd = new TPrintThread(new TSimpleListener());         //注册 Listener 实例到线程中
    ...
}
```

如果注册 TThreadListener 的子类对象不同，就会执行不同的结束代码，而线程类本身并不修改。这一模式就是策略者模式（Strategy Design Pattern），其方法是：将代码中不能确定的逻辑分离成一个抽象的策略类（TThreadListener），其中的虚函数（NotifyThreadFinished）代表了该策略的实现逻辑；将不同的策略子类实例注册到环境类（TThread）中，在环境类中调用策略类的虚函数（Wrapper()函数中对 NotifyThreadFinished 的调用），从而执行不同的策略逻辑。策略者模式使得环境类的代码不随策略逻辑的变化而修改，不论注册什么 TThreadListener 子类，Wrapper()函数都不用改动。

策略模式从功能上与 5.2.5 节的模板方法模式可等价（见 Robert C. Martin 所著的《敏捷软件开发：原则、模式与实践》）。做法上，策略模式要复杂点，要实例化策略对象，但从功能上说，策略者模式要灵活得多。因为它可以不变动环境，而修改策略。等价模板方法的运用，为了变动策略需

要构造新的环境（这往往代价巨大）。因为该模式将策略和环境两个类合为一个类了。大家不妨设想，在一个集合上实现不同的排序逻辑，用两种模式来实现，并分析其优劣。策略模式的灵活性甚至让笔者有一种“万物皆策略”的感觉。**（还清晰地记得与某个学生一起用策略模式搞定了一个比较复杂的嵌套代码生成器后，他在其后一段时间内，言必称“策略”的状况。那是一种一点通处处皆能圆融无碍的境况。）**事件通知的领域，均可很好运用策略模式，如 Java 的图形框架中遍地皆是 Listener 类。

5.3 跨平台的线程设计

跨平台是软件开发中重要又非常昂贵的工作。Java 通过虚拟机翻译（解释运行或即时编译）达成了。对于没有虚拟机的语言呢，如 C++、Delphi 之类？本节中我们要学习源代码跨平台的技巧及模式，实现不同方式跨 Linux 和 Windows 平台的线程类。

5.3.1 讨厌的 Linux 版本

我们先来看看 Linux 下的线程接口，需要加入头文件 pthread.h。在学习了 Windows 系统后可用一种快速的方式去“查找” Linux 的相关接口。如果没学习过如何查找呢？本书强调的一个重要的方式是，先构想自己做该怎么做，再看别人的。那是在一无所知的情况下，而当已经知道了一种方式（Windows 的方式），自然可用它为范本去“查找” Linux 的做法。通过查找得到如下对比：

```
pthread_create←→CreateThread
pthread_detach←→CloseHandle
pthread_join←→WaitForSingleObject
pthread_exit←→ExitThread;
```

函数

```
int pthread_create(pthread_t *tidp, pthread_attr_t *attr,void*(*start_rtn)(void*), void *arg)
```

用于构造新线程，成功则返回 0，否则为出错编号。

第 1 个参数的类型 pthread_t *其实是 usigned long *，成功则其指向整数为线程标识。第 2 个参数操控线程的属性，一般传入 NULL，表示用默认属性。第 3、4 个参数分别为线程函数的入口地址和参数，与 CreateThread()的相当。

该函数不支持是否以悬挂方式启动，这与 Windows 不同。因为 POSIX 标准的线程语义不支持悬挂和恢复，与多线程死锁问题有关。如果一个拥有锁的线程 A 被线程 B 悬挂了，而线程 B 又在等待被悬挂线程 A 尚未释放的锁，那么就发生死锁了。

pthread_detach，只传入一个参数，即线程的 ID，它是 pthread_create 返回的第 1 个参数。基本等价于 Windows 系统中的 CloseHandle。

int pthread_join(pthread_t thid, void **thread_return)，调用该函数的线程等待指定线程的结束，由第 1 个参数指定要等待的线程。第 2 个参数保存线程自定义的返回值。

pthread_exit(void* retval)，退出线程，并返回指向某个用户自定义内存结构的指针。这里比较好理解 pthread_join 的第 2 个参数，因为线程指定了一块内存返回，所以 pthread_exit 将该内存地址用其参数告知内核。调用 pthread_join 的线程如何获得该地址？它只能传入一个指针 type *，让内核填写该指针指向的内存。type 就是 pthread_exit 传入的那个 void *，所以用 void *替换 type，则

pthread_join 的第 2 个参数就是 void **。

根据与 Windows 的对应函数，我们容易将 Windows 版本的线程类修改为 Linux 版本，工程参见“code\第 5 章\ThreadClass_Linux”。因为要为跨平台编写，这里主要将与 Windows 不同的部分列出，相同部分省略。

首先是头文件不同，一个是 windows.h，一个是 pthread.h。其次，因为 API 返回值、参数类型和线程函数原型的不同，导致类定义、代码实现不同，见 DM5-9。

DM5-9

(a) 类定义

```
class TThread{
  private:
    int FThreadID;                          //类型不同，windows 版为 HANDLE
    void * FReturnValue;                    //类型不同，windows 版为 int
    //函数原型和返回类型不同，Windows 为_stdcall，返回为 unsigned long，
    //即 static unsigned long _ _stdcall Wrapper(void * arg)
    static void * Wrapper(void * arg);
    ...
    void * GetReturnValue() {
       return FReturnValue
    };                                      //返回类型不同，Windows 为 int
};
```

(b) 代码实现

```
1   void * TThread::Wrapper(void * arg) {
2       TThread * Thd;
3       Thd = ((TThread *) arg);
4       Thd->Execute();
5       if (Thd->FListener != NULL)
6         Thd->FListener->NotifyThreadFinished(Thd);
7       pthread_exit(FReturnValue);
8   }
```

在 DM5-9(b)中，除第 7 行，所有代码与 Windows 版本一样。这种情况还存在于其他代码中。简单地说，它们拥有相同逻辑，具体与操作系统相关接口的调用会有差异。于是出现了这种夹心饼干似的代码，通用代码和差异化系统调用混在一起。这有何问题？重复就是罪恶。假想 DM5-9(b)有 bug 要修改，岂非要在两个版本的代码中修改（其实 5.2.6 节已展示了代码修改演化的情况，当时在简单版本中加入了 Listener）。如果我们维护了一个跨十几个操作系统的库（ACE 就跨十几个操作系统），岂非直接崩溃？如果能共享 DM5-9(b)中相同的第 1～6 行代码，那就舒服了。

重复即罪恶

5.3.2 源代码跨平台技术

怎样共享通用代码而剥离出不同的部分呢？这里要提到的是一种被称为条件编译的技术，在第 3 章中提到过。

```
#ifdef macro_name
   ...
#endif
```

只要申明了宏 macro_name，那么#ifdef 与#endif 之间由省略号代表的代码对编译器而言就是存在的，将被编译。如果没有定义该宏，则这些代码编译器会“视而不见”，当它们根本不存在。

如何定义一个宏而又不修改代码呢？在 gcc 下可在 make 文件中定义，在 VS 2008 这样的集成开发环境中，右键单击项目图标，选择“属性”菜单，然后在图 5.3 所示的预处理器定义中输入你定义的宏，这里是 marco_name。多个宏用分号分开。图 5.3 中显示定义了两个宏：WIN32 和_DEBUG。

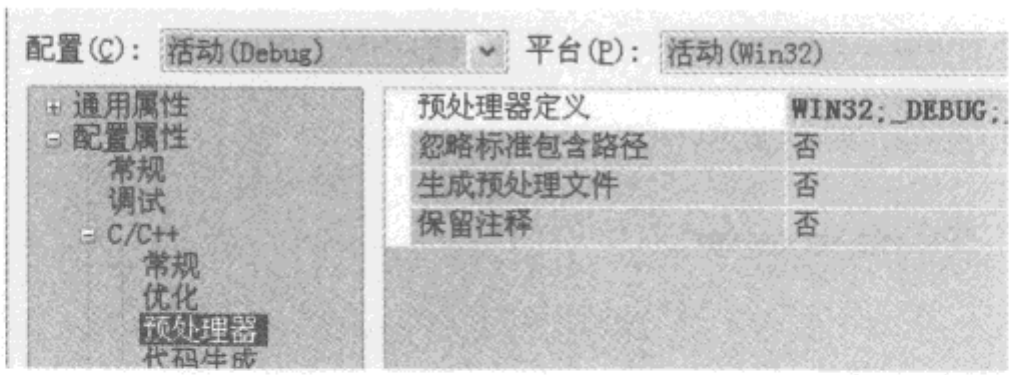


图 5.3

如果要生成跨多个操作系统的源代码，则会定义多个这样的宏，如 WIN32、Linux。这时用上面的方式不是很好。我们会采用下面的写法，它展示了关于头文件的条件编译：

```
#ifdef WIN32
#include <windows.h>
#elif defined(Linux)
#include <pthread.h>
#else
#error      "Please fix os version"
#endif
```

粗体部分显示了这种方式的条件编译结构，如果定义 WIN32，则加入头文件 windows.h；如果定义了 Linux，则加入头文件 pthread.h；如果两者都不是，#error 则会在编译时报错"Please fix os version"，其中的字符串，大家根据需求自定义。#else 和#error 的组合很重要，如果用户未定义操作系统版本相关的宏，或者它不属于 WIN32 和 Linux，那么编译会出错，起提醒作用。Linux 操作系统源代码就采用该方式。

5.3.3 跨平台的版本

下面对比 5.3.1 和 5.2 节中两个平台版本的差异，然后将差异部分用条件编译处理。工程见“code\第 5 章\ThreadClass_CrossPlatform1”。以头文件为例，下面的粗体代码展示了由于线程函数原型不同而导致的条件编译。对 WIN32 分支，线程函数为 static unsigned long __stdcall Wrapper(void * arg)；对于 Linux 分支，线程函数为 static void * Wrapper(void * arg)。


```
class TThread {
  private:
    unsigned long FThreadID;
    ...
    #ifdef WIN32
      static unsigned long __stdcall Wrapper(void * arg);
    #elif defined(Linux)
      static void * Wrapper(void * arg);
    #else
    #error"Please fix os ver"
    #endif
    ...
}
```

这样的条件编译分支在代码中比比皆是，如下面的代码展示了线程函数的实现。

```
#ifdef WIN32
```

```
    unsigned long _ _stdcall TThread::Wrapper(void * arg)
#elif defined(Linux)
    void * TThread::Wrapper(void * arg);
#else
    #error    "Please fix os ver"
#endif
{
    TThread * Thd;
    int i;
    Thd = ((TThread *) arg);
    Thd->Execute();
    if (Thd->FListener != NULL)
        Thd->FListener->NotifyThreadFinished(Thd);
#ifdef WIN32
    ExitThread(Thd->FReturnValue);
    return Thd->FReturnValue;
#elif defined(LINUX)
    pthread_exit(&Thd->FReturnValue);
    return (void *) (Thd->FReturnValue);
#else
    #error    "Please fix os ver"
#endif
};
```

该方式在跨平台中较常见，如 Delphi 6.0 中的 VCL 库跨平台实现。如下代码是对事件机制的跨平台封装，粗体部分为条件编译：

```
constructor TEvent.Create(EventAttributes : PSecurityAttributes; ManualReset,
                                                    InitialState : Boolean; const Name : string);
{ $IFDEF MSWINDOWS }
begin
    FHandle := CreateEvent(EventAttributes, ManualReset, InitialState, PChar(Name));
end;
{$ENDIF}
{$IFDEF LINUX}
var
    Value : Integer;
begin
    if InitialState then
        Value := 1
    else
        Value := 0;
    sem_init(FEvent, False, Value);
end;
{$ENDIF}
```

但是，要将上面 TThread::Wrapper 代码中关于操作系统的代码（粗体部分）与通用代码（非粗体部分）区分开很难。这些条件编译的存在大大降低了代码的可读性，让我们有点“头晕眼花”。而每个调用了线程 API 的函数均包含了这种条件编译，在 Wrapper()短短 26 行代码中竟有 20 行之多。随着支持的操作系统数目增加，如达到 10 种时，条件编译中的分支数将增加到 60 行。但操作

系统无关的代码只有 6 行。要在 60 多行中“分辨”出这“散碎”其中的 6 行，并理解其逻辑简直是大海捞针，可读性太差。

现实生活中是怎样隔离差异化、“统一”管理的呢？比如，一个省要管理多个市，每个市的情况可能很大不同，但一个省如何用同一流程管理它们呢？每个市都有相同的对外接口，则每个市内部的差异就可屏蔽。对于线程呢？每个操作系统 API 的名字和参数都不同，那么我们可否给它们规定一个对外统一接口，将这些差异都变成内部？对外的调用接口要统一哪些东西呢？API 接口说白了就是函数，函数的几个要件就是需统一化的东西：函数名，参数列表（包括个数、类型），返回类型，调用惯例。

对于函数名的处理，可定义一个函数，让它去调用不同操作系统的 API。比如，等待线程结束的接口可以定义为 WaitFor，在该函数中再用条件编译调用 Windows 的 WaitForSingleObject 和 Linux 的 pthread_join。

```
WaitFor(…) {
    #ifdef WIN32
        WaitForSingleObject(…);
    #elif defined(LINUX)
        pthread_join(…)
    #else
        #error    "Please fix os ver"
    #endif
}
```

线程类不直接用操作系统相关函数，而调用 WaitFor()，那么线程类中就不会“混”入条件编译代码了，它们被隔离在抽象出的统一接口 WaitFor()中。

要外部调用 WaitFor 的代码不变，还需要参数列表相同，否则，调用代码需根据不同操作系统而传递不同参数，最终导致无法用同一代码调用 WaitFor。但有的函数参数列表确实不同，如线程生成函数，Windows 和 Linux 的参数个数就不同。这时只有一个办法，就是抽象出共同的，忽略掉不同的，不同的参数只传递默认值。例如，CreateThread 支持是否悬挂方式构造，而 pthread_create 没有，那么，对 CreateThread，该参数只能传递一个固定值让它与 pthread_create 的效果相同，即传递 0，使其非悬挂启动。

对于抽象的统一接口的返回值，除了类型相同以外，还必须语义相同。在 WaitForSingleObject 和 pthread_join 上，我们应该处理之。

以上提供统一 API 让外部调用的方法，可很好地将调用代码中操作系统相关部分移除，使其易于维护。而系统相关代码被迁移到自定义的统一 API 中，也便于管理。这种做法的一个变体是将统一的 API 由函数变为宏，即 WaitFor 不是一个函数，而是一个宏定义。其好处是少了一次函数调用、返回和参数传递，从而速度会提高。用这种方法，每个 API 内部还有很多条件编译分支（支持多少操作系统就有多少分支）。有没有办法将其分支数减少，并维护容易呢？

下面给出一种笔者称为 APIBinder 的方法，它部分参考了来自 Delphi 的开源项目 Indy 的做法，并针对 C++做出了相关调整。首先申明一个类，为 4 个线程 API 定义了 4 个函数，其头文件中包括其申明：

```
class TThreadAPIBinder{
  public:
static HANDLE MakeThread(unsigned long * ThreadID, void * (* StartAddress)(void *) , void * arg, int *error);
```

```
    static long WaitFor(unsigned long Handle);
    static void ExitFromThread(unsigned long ExitCode);
    static void Cleanup(unsigned long Handle);
};
```

例如，WaitFor 提供了线程等待语义。线程类中所有对线程的操作均调用这 4 个相关函数。这样线程类实现代码将不包含操作系统相关代码。例如，线程的 WaitFor()方法就是调用了 TThreadAPIBinder 的 WaitFor()方法，代码如下，黑体部分在之前是操作系统相关 API 的调用，现在无关了：

```
void TThread::WaitFor() {
    int Result;
    void * P;
    if (!FSuccessMake)
        return;
    P = &Result;
    TThreadAPIBinder::WaitFor((unsigned long)FHandle);
};
```

然后对应不同操作系统 TThreadAPIBinder 的实现代码放在一个单独的 CPP 文件中，被一个对应的条件分支包括。下面的 ThreadBinder_Win.cpp 实现了对 windows 的封装，粗体部分是条件编译，决定到底启用哪个版本，Winows 版本对应的宏为 WIN32。

```
#include "ThreadAPIBinder.h"

#ifdef WIN32

#include <windows.h>
//TThreadBinder 对应的 Windows 实现…
long TThreadAPIBinder::WaitFor(unsigned long Handle) {
    return WaitForSingleObject((HANDLE) Handle, INFINITE);
};
…
#endif
```

而 ThreadBinder_Linux.cpp 实现了对 Linux 的封装，对应宏为 LINUX：

```
#include "ThreadAPIBinder.h"

#ifdef LINUX

#include <pthread.h>
//TThreadBinder 对应的 linux 的实现…
long TThreadAPIBinder::WaitFor(unsigned long Handle) {
    long Result;
    void * P;
    P = &Result;
    pthread_join(Handle, &P);
    return Result;
};
…
#endif
```

这个版本中没有用条件编译的#error 语句，所以如果用户没有定义 WIN32 或 LINUX，并不会

报编译错误。为了保持前面版本报错的好处。我们会在申明 TThreadAPIBinder 的头文件中加入如下条件编译，除了报错没有其他实际意义：

```
#ifdef WIN32
#elif defined(LINUX)
#else
    #error    "Please fix os ver"
#endif
```

回调指针 wrapper

这种为不同操作系统 API 抽象共同语义接口的方法遇到的困难并非只是将个性化的参数采用默认值的问题（如 CreateThread 的悬挂启动标志），有的参数差异是无法用默认值解决的。如 MakeThread 要传入线程函数的地址，该函数原型调用惯例一个为_stdcall，一个为 cdecl，这个参数无法取默认值。怎么办？笔者用如下方法解决，供参考：

① MakeThread 接收的是 cdecl 方式的函数指针。

② 对 Windows 版本，CreateThread 依然接收_stdcall 函数 WinWrapper()，由它调用用户的线程函数（调用惯例为 cdecl）。WinWrapper 的参数为一结构体：

```
struct ThreadWrapperArg {
    void * (* StartAddress)(void *);
    void * arg;
};
```

其中，StartAddress 指向通过 MakeThread 传入的线程函数入口地址（其所指函数是 cdecl），arg 为 StartAddress 所指函数的参数。这样在 WinWrapper()中就可调用 StartAddress 了。以下代码中必须用 malloc 为线程参数分配一个独立内存块（原因见 5.1.4 节）。

```
HANDLE TThreadAPIBinder::MakeThread(unsigned long * ThreadID,
                                    void * (* StartAddress)(void *) , void * arg, int *error) {
    struct ThreadWrapperArg* pWrapperArg;
    HANDLE handle;
    //allocate memory , to prevent from thread access local memory of MakeThread. it will be freed
    pWrapperArg = (struct ThreadWrapperArg*) malloc(sizeof(struct ThreadWrapperArg));
    pWrapperArg->StartAddress = StartAddress;
    pWrapperArg->arg = arg;
    handle = CreateThread(NULL, 0, WinWrapper, pWrapperArg, 0, ThreadID);
```

线程运行时，将执行 WinWrapper()函数。代码如下，它从传入的 arg 结构体两个成员变量中获取到用户真正要执行的函数 StartAddess()和参数，并调用 StartAdress。工程见“code\第 5 章\ThreadClass_CrossPlatform2”。

```
unsigned long __stdcall WinWrapper(void * arg){
    struct ThreadWrapperArg* theArg;
    unsigned long ReturnValue;
    theArg = (struct ThreadWrapperArg*) arg;
    ReturnValue = (unsigned long)theArg->StartAddress(theArg->arg);
    free(theArg);
    return ReturnValue;
};
```

注意：参数 arg 在返回前要释放。

这种方式的好处如下：

- ⊙ 生产代码中（TThread 的代码）不会混入操作系统相关代码，因此易维护和理解。
- ⊙ 每个操作系统相关代码被封装到一个单独的 CPP 文件中，容易维护；支持新操作系统，只需添加新的 CPP 文件。
- ⊙ 条件分支极少化，一个系统实现对应一个分支（在 CPP 文件的头尾），易于维护。

该方法在 Indy 中的实现是，利用 Delphi 静态函数可为虚函数的特性，不同操作系统用子类覆盖实现了这些虚函数，这样做有利于动态绑定。在 C 语言中可采用如下等价方法：定义一个结构体，包含线程相关的 4 个函数指针；定义一个该结构体类型的全局变量，对不同操作系统，该变量的函数指针绑定不同函数；线程类通过该全局变量上的函数指针访问系统 API。

5.4 崩溃，哪里出错了

5.4.1 寻找错误

导图 5.1

当年，笔者将线程类依照 Delphi 的实现迁移到 5.3.3 节所示版本，一切运行似乎良好。可有时就会出现执行纯虚方法错误。我们知道，包含纯虚方法的类在 C++中是无法实例化的，想弄出这种效果都难。到底哪里出错了？代码 DM5-10 展示了出错的状况（这段代码并不一定出错，不同操作系统及版本出错几率不同，工程见“code\第 5 章\ThreadClass_Simple_err”）。笔者当时完全没有头绪，所以就采用最笨的方法，单步跟踪。首先会遇到如图 5.4 所示的错误框。

DM5-10

```
1   TPrintThread * Thd;
2   Thd = new TPrintThread();
3   Thd->Run();
4   delete Thd;
```

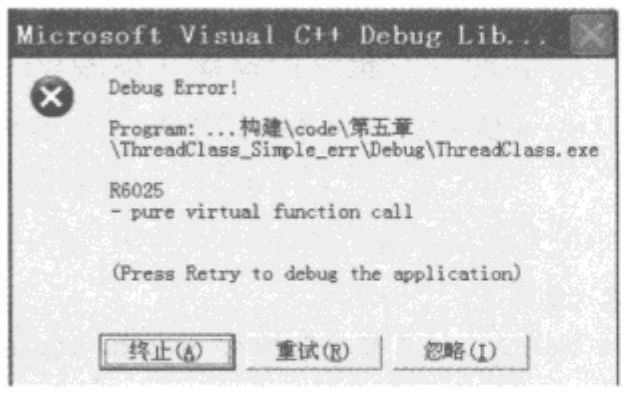


图 5.4

单击“重试”按钮，运行依然会弹出错误对话框。单击“终止”按钮，这时运行光标会暂停在 crt0msg.c 的一行代码处。没有头绪？没关系，如果是这一行执行时抛出的错误，如果能看到之前执行的代码，就会有些头绪。这时运用一个很有用的调试功能，C++和 Delphi 中都称为 call stack，即调用堆栈，Java 中也有类似功能。它能够展示在断点处，程序是经过哪些函数调用到达该断点的。例如，有如下调用关系：函数 f1 调 f2，f2 调 f3，而断点在函数 f3 中。激活调用堆栈功能时，我们能看到界面中展示出以上调用关系。图 5.5 给出了激活 call stack 的方法，选择“调试 | 窗口 | 调用堆栈”菜单命令，将看到如图 5.6 所示的调用堆栈界面。

图 5.6 显示了如下调用关系 TThread::Wrapper → _purecall → _NMSG_WRITE，其中箭头指示当前执行到的代码，可知执行 TThread::Wrapper 的时候出错了。双击图 5.6 中 TThread::Wrapper 所在的行，就转到了 Wrapper 中的出错语句——Thd->Execute()出错了。图 5.4 的错误提示正是显示执行了纯虚函数，而 Execute 就是一个虚函数，且 TThread 中申明的就是纯虚函数。那么，我们应该看得更清楚些，那就是反汇编。

激活反汇编后，代码见 DM5-11。

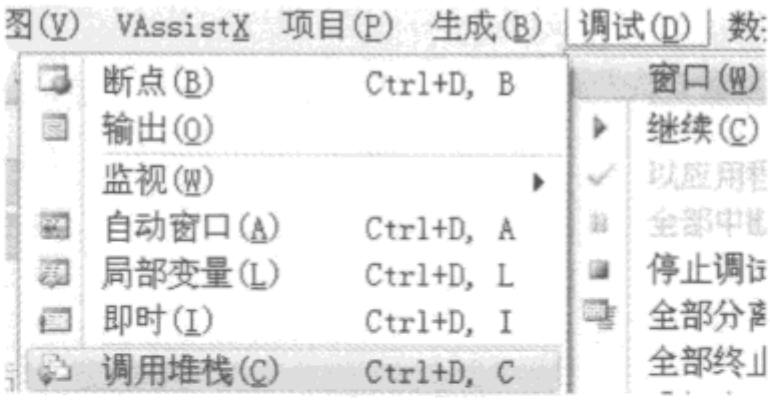


图 5.5

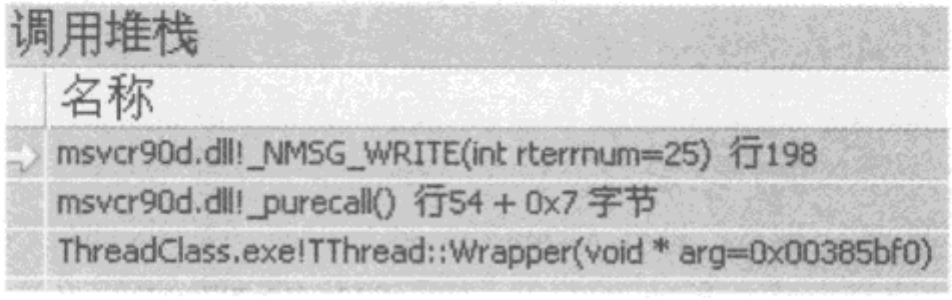


图 5.6

指示第 7 行 cmp 指令出错，可见是在执行 call 时出错。这段代码是在调用虚函数 Execute。第 4 章中分析过虚函数调用过程："从对象前 4 字节获取虚表入口地址，然后从对应偏移槽位取出虚函数入口地址。"TThread 只有一个虚函数 Execute()，所以其入口地址就在虚表第一个元素，偏移为 0。ebp–8 地址指向的内存[ebp–8]中存放的就是线程对象地址，第 1～2 行将对象前 4 字节的虚表地址存入了 EDX 中。第 5 行将虚表的第 1 项即 Execute()的入口地址放入 EAX，第 6 行调用虚函数 Execute()。这时，我们无法直接查看寄存器，如 EAX，因为它已经不是执行该代码时的值了。但我们可以通过监视窗口获得 Wrapper()函数的参数 arg，它就是线程对象地址。在获取到对象地址的情况下，只要按照虚表结构去查看内存窗体，我们就能还原出执行 Wrapper()时到底调用了哪个函数。

DM5-11

```
1  00411534  mov  eax, dword ptr [ebp-8]
2  00411537  mov  edx, dword ptr [eax]
3  00411539  mov  esi, esp
4  0041153b  mov  ecx, dword ptr [ebp-8]
5  0041153e  mov  eax, dword ptr [edx]
6  00411540  call eax
7  00411542  cmp  esi, esp
```

导图 5.1 m2

为什么出错时不能查看 eax 的值，却可以查看参数 arg 的值

从 1.3.4 节和 1.3.5 节可知，局部变量和参数是通过 EBP 寻址的（见图 1.32）。当问题发生时，只要错误代码没有修改调用者使用的栈区（栈帧），能恢复调用函数当时 EBP 的值，就可正确找到此时的局部变量和参数。能够恢复吗？由图 1.32 可知，只要每个函数进入时都保持 EBP 的值，就可追溯出来。为了证明 EBP 的值被恢复了，双击图 5.6 中不同的函数，观察寄存器窗体，可知 EBP 和 ESP 因选择不同函数而改变。为什么 ESP 也能恢复呢？请自己对照图 1.32 分析。

导图 5.1 m3

首先从监视窗口获得 arg 的值是 0x00385bf0（对象首址）。

然后在内存窗体中输入 0x00385bf0，回车查看，它开始的 4 字节就是 40 67 41 00，按照小端机，该数为 0x416740，它是虚表的地址。之后将 0x416740 输入内存窗体查看，前 4 字节对应的数是 0x411181，它是虚表的第一项存的值，即 Execute 的入口地址。

再在反汇编窗体的地址栏中输入 0x411181，该地址的代码是：

```
00411181      jmp  00411c76
```

同理，在反汇编窗体地址栏中输入转跳的地址 0x00411c76，显示的源码是：

```
_purecall:
00411c76      jmp  dword ptr ds:[00419314h]
```

可见，此时虚表中存储的根本不是类 TPrintThread 的虚函数 Execute()入口地址，而是 purecall()函数的入口地址。为什么会这样？难道对象前 4 字节存储的是其他类的虚表地址？（**这真是一个惊人的发现**。）难道编译系统错误地设定了虚表？从哪里下手找到设定虚表的代码？按理应该是构造函数中设定了该虚表地址，因为 new 操作符只是分配了内存，并没有类相关的任何信息。那么，在 TThread 和 TPrintThread 的构造函数中设置断点来看看吧。

执行后的第一件事令人惊讶，断点先停在子类 TPrintThread 的构造函数，而不是我们原来学的“先执行父类构造函数，然后执行子类的”。（**看来实践总能有新发现**。）不过通过跟踪分析发现，虽然先调用子类的函数，但编译器会在子类构造函数有效代码（左大括号后的代码）执行前，先调用父类构造。虽物理上先执行子类构造，但逻辑上先执行了父类构造。

我们关注的是，哪条指令将对象前 4 字节设为对应虚表地址，但又不想分析反汇编，解决思路就是：查看对象前 4 字节并单步跟踪，直到某指令让对象前 4 字节变化。如此，则需先获取 this 的值，然后输入到内存窗体中观察。进入 TPrintThread 的构造函数时，监视窗体中还无法观察 this。
导图 5.1m4
这难不倒我们，在第 4 章已经知道，C++通过 ECX 传递对象首部地址，只要查看 ECX 的值即可，它是 0x00385BF0。

剩下的工作就是在内存窗体中输入该值，此时 0x00385BF0 指向内存的内容为 cd cd cd cd。跟踪到父类 TThread 的构造函数时，发现在执行该函数的第一行有效代码（FThreadID = 0）之前有如下指令：

```
004114d3    mov   eax, dword ptr [ebp-8]
004114d6    mov   dword ptr [eax], 416740h
FThreadID = 0;
```

第 2 行指令执行时，内存窗体中跟踪的值被改变了。第 1 行将 ebp–8 指向的内存中保存的对象地址赋给了 EAX，第 2 行将 416740h 赋值给了 EAX 指向的内存，即对象的前 4 字节被赋予了 0x416740。这正是出错时的那个虚表地址，该地址应该是 TThread 的虚表地址。这个地址肯定不对，照理应该被设定为 TPrintThread 的虚表地址。

继续单步跟踪，关注内存窗体，看看该内存区是否还有被修改的时候。当父类构造函数返回后，在子类的构造函数中，我们发现下面两行代码修改了关注的值：

```
0041199b    mov   eax,dword ptr [ebp-8]
0041199e    mov   dword ptr [eax],41674ch
```

第 2 行将对象的前 4 字节修改为 41674ch。如果查看该地址的前 4 字节，即 Execute()函数入口地址，可发现，该地址指向代码最终转向 TPrintThread::Execute。可见，0x0041674c 才是子类 TPrintThread 的虚表地址。

仔细分析以上过程，我们发现，在父类 TThread 的构造函数中，对象前 4 字节被设定为 TThread
导图 5.1m5
的虚表地址，执行 TPrintThread 构造函数时，对象前 4 字节又被设定为 TPrintThread 的虚表地址。也就是说，执行哪个类的构造函数，就会将哪个类的虚表地址设定给对象。为什么会这样？为什么不始终设定为最终实例化的那个子类（TPrintThread）的虚表地址？这也好理解，因为父类构造函数先执行，此时虚表设定为子类的，当构造函数中调用虚函数时，必然调用到子类的虚函数。此时子类构造函数尚未执行，假定子类的虚函数必须依赖子类构造函数执行后才能正确执行（如子类构造函数中 new 了一个成员对象，子类的虚函数要使用该对象），那么势必造成在父类构造函数中调用子类虚函数出错。

导图 5.1m6
从构造函数的逻辑，我们也可对析构函数做出相似推理，调到哪个类的析构，将会把对象前 4

字节设定为该类虚表入口。在类 A 的析构函数中，只能调用到类 A 的虚函数，不会调到真正实例化子类的虚函数。通过反汇编，该结论很容易被证明，此处不给出。

从前面的分析可知，调用 Wrapper()函数出错时，对象前 4 字节指向父类 TThread 的虚表，而不是希望的子类 TPrintThread 的虚表。假定代码没有其他错误，那么这个虚表地址的变化一定是构造函数和析构函数造成的。此时因为已经执行了成员函数 Run()，所以构造函数已正常结束，应该不是它的问题。而且从之前跟踪结果看，在 TPrintThread 的构造函数中确实也正确设定了虚表地址。那么，似乎只可能是析构函数导致的。难道执行 Wrapper()函数时，析构函数已经执行了吗？

回顾 DM5-10，在第 3 行 Run()方法调用 CreateThread 后，第 4 行调用了 delete Thd，这里执行了析构函数。我们知道，并非一执行 CreateThread，线程就开始运行，要等待当前线程时间片执行完毕后，操作系统才调度下一线程运行。如果调度到新线程，新线程才会运行。Wrapper()函数是在新线程中运行的，即有一种情况：Run()执行后，当前线程时间片未结束，就会执行 delete Thd。TThread 中的析构函数做了什么？等待新线程结束，此时新线程还没运行。当主线程在父类析构函数中等待新线程结束时，新线程开始运行。此时因为执行了父类析构函数，所以此时对象前 4 字节已经指向 TThread 虚表，而新线程中执行 Wrapper()，调用 Execute()虚函数必然调用 TThread 的虚函数——那个纯虚的版本！为了证明这个分析，首先我们需要将该问题 100%复现，然后解决它。

现在的代码，如果 DM5-10 中执行了 Run()后，主线程时间片到了，然后新线程被调度，那么就不会因为主线程在新线程运行前执行析构函数替换虚表指针而导致错误。为了让错误一定产生，只需保证新线程调用虚函数 Execute()时主线程已经执行了析构。怎么办？让 Wrapper()在调用 Execute()前调用“Sleep(2000);”，休眠 2 秒，主线程必然执行了析构。（**因为 Windows 的线程时间片远小于 2 秒，即使主线程执行了 Run()后被切换，2 秒的时间也足以让系统再次调度它执行后续析构语句。**）

有了这个 100%复现问题，让这个问题消失就能证明前面的分析正确。我们只需在 delete Thd 之前加入“Thd->WaitFor();”语句即可。因为在等待时，没有执行 TThread 的析构函数，就不会改变对象前 4 字节的虚表地址，自然结果正确。请大家自行做这个验证。

简单总结，该错误非常微妙。即使我们知道了 C++构造函数和析构函数中调用的虚函数只是本类的实现，而不是被实例化的子类实现，也很难发现这个错误，因为我们并没有在构造和析构函数中调用虚函数。如果没有很好掌握虚函数机制，并具有灵活运用底层调试机制分析的能力，很难发现玄机，特别当问题时隐时现时。（**请用该例好好品味底层力量吧。**）

5.4.2 C++下整体资源管理的反思

值得思考的是，5.4.1 节的做法为什么在 Delphi 中可以呢？通过反汇编跟踪，我们发现，Delphi 中对象前 4 字节始终指向实例化子类的虚表，不会随构造函数而变化，其结果就是：当构造函数和析构函数中调用虚函数时，被调用的是实例化子类的虚函数，而不是当前被执行的析构函数所在类的虚函数。我们不禁要问：

① 难道 Delphi 不担心 C++想要避免的错误吗？

② 这样做有什么好处？

第一个问题的答案就是，为了获得第二个问题涉及的好处，Delphi 将避免可能错误的责任推给了语言的使用者（只要父类调用的子类虚函数中没有访问子类析构已经释放的资源即可）。这就如同 5.2.3 节提到的那个实例化纯虚函数的例子，不同的权衡有不同的策略。这个策略在 Java、C#中

获得了同样的支持。

我们在第 4 章中提到过面向对象软件设计的一个策略：在构造函数中分配资源，在析构函数中释放资源。这被称为整体资源管理（General Resource Management），也被称为 RAII（Resource Acquisition Is Initalization）。我们已经见识了在文件和临界区应用该策略。

再有，比如在 VCL 类库中封装了一个叫 THandleObject 的类，它的析构函数调用 CloseHandle 关闭资源，而其子类，如 TEvent 类，在其构造函数中调用 CreateEvent 分配资源产生句柄：

```
destructor THandleObject.Destroy;                    //父类 THandleObject 的析构函数
begin
   CloseHandle(FHandle);
   inherited Destroy;
end;
constructor TEvent.Create(EventAttributes: PSecurityAttributes; ManualReset,
   InitialState: Boolean; const Name: string);       //子类 TEvent 的构造函数
{$IFDEF MSWINDOWS}
begin
   FHandle := CreateEvent(EventAttributes, ManualReset, InitialState, PChar(Name));
end;
```

DM5-12

```
1   class Communicator {
2     protected:
3       virtual void CloseChannel();
4     public:
5       ~Communicator() {
6           SendCloseMsg();
7           FreeThreadPool();
8           ...
9           CloseChannel();
10      }
11  }
12  class ShareMemCommunicator : public Communicator {
13    protected:
14      virtual void CloseChannel(){ /*释放共享内存*/ }
15  }
```

整体资源管理的一个扩展如下：如果父类释放资源时，所涉及的释放方式需要具有多态性，即在释放资源时除了父类自己的一部分释放逻辑，还要释放一些父类中不可确定具体类型的资源（子类才能确定），那么，为了共用释放逻辑，就可采用模板方法模式，在父类中定义释放资源的虚函数，并在父类析构函数中调用，而子类覆盖该函数实现具体释放逻辑。伪代码见 DM5-12。

在基类 Communicator 中定义了虚函数 CloseChannel()，见第 3 行。该类的析构函数（第 5～10 行）会先发送关闭消息，然后释放线程池，最后会调用虚函数 CloseChannel 关闭通道。因为它不知道 Channel 的类型，所以无法执行具体的关闭逻辑，但为了共享前面的最终化逻辑（第 5～8 行），所以将 CloseChannel 定义为虚函数，让子类覆盖。子类 ShareMemCommunicator 提供了用共享内存实现的 CloseChannel 覆盖版本。

在支持父类构造和析构函数调用子类虚函数实现的语言中，该模式没有任何问题。假定 ShareMemCommunicator 的对象释放，Communicator 析构函数调用 CloseChannel 时，调用到的是 ShareMemCommunicator 的覆盖版本，而非 Communicator 的。C++中，调到的实际是 Communicator 的版本，这就不对了。此时可回答本节开始的第二个问题了，答案是：因为在析构中释放的资源有些具有多态性，父类无法确定其释放逻辑，如果要在父类中定义统一的析构逻辑，那么需要在父类析构中调用子类虚函数，由子类的虚函数实现那些差异化的释放逻辑。（**个人更喜欢这种策略，它让设计更灵活、强大。**）想来这也是后于 C++的 Java、C#、Delphi 都支持该机制的部分理由吧。

要在 C++中解决这个问题，一种办法是在子类中关闭子类的资源，即在 Communicator 析构中不调用 CloseChannel，而是在 ShareMemCommunicator 的析构函数中完成释放逻辑。但这样并非完美方案，如本例中，共享内存版 channel 会因为 ShareMemCommunicator 析构先执行而关闭，当

Communicator 析构函数调用 SendCloseMsg 时（DM5-12 第 6 行），因 channel 已经关闭，再用其发消息就必然失败。对于释放资源有顺序要求的状况，这个办法是不行的。

有更好的办法吗？其实，上面的模板方法模式有一个等价的策略者模式，这个模式调用的虚方法并非属于环境类（Context Class），而是属于策略者。那么，环境类的析构执行并不影响策略者类虚方法的多态性。在 Communicator 的例子中，只要定义一个 Channel 类，使其成为 Communicator 的一个成员 Channel *，然后 Channel 成为策略者基类，实现一个 ShareMemChannel 的子类即可。运用这一思想来修订 5.4.1 节中发现的线程类 bug 吧。

C++中，限制整体资源管理的另一个因素就是构造函数调用的顺序：必须先执行父类的构造，再执行子类的构造。一般情况下，这种策略很安全也没有问题。但它限制了资源创建的顺序，而有的资源可能虽然在父类中封装了创建逻辑，但创建时机却必须晚于子类资源的创建。这时就需要允许程序员手动控制构造函数调用顺序。

还是以线程为例，如果在基类 TThread 构造函数中调用 CreateThread 创建线程资源，线程就执行了。而线程执行却要操作子类 TMyThread 构造函数中分配的资源。这时的困境是，TMyThread 的构造后执行，所以可能当线程要使用 TMyThread 分配的资源时，TMyThread 的构造还没有执行，必然出错。Delphi 将构造函数和析构函数执行的顺序让程序员控制。首先执行子类构造，如果在子类构造中一开始就调用父类构造，效果上就等价于父类构造先执行，也可在其他位置调用，甚至不调用（当然就可能错误）。其实，这个机制与 C++保证先调用父类构造的机制是一样的，如第 4 章的 DM4-46(d)，依然先执行子类构造，只是编译器在一开始强行悄悄加入了对父类构造的调用。

有了程序员控制构造、析构调用顺序的机制，线程的困境也就能解决。我们在 TMyThread 的构造中先分配资源，然后调用父类构造函数。调用后必然创建线程执行，此时资源已经分配，所以线程访问这些资源不会出错。

5.4.3 生生死死虚表误，剥离策略世界殊——重生

5.4.1 节的错误在于为了实现整体资源管理，在父类析构中等待线程结束，如果此时线程还没执行，父类析构的执行就将对象入口 4 字节设定为父类虚表地址，从而导致线程接下来执行的虚函数将是父类的，而非子类。该问题出现必须同时具备下面两个条件：

- ⊙ 调用的虚函数 Execute 属于线程类。
- ⊙ 要在线程基类析构中等待线程结束。

第 2 个条件是整体资源管理的要求，无法回避。那么第 1 个条件能否被破坏？第 1 个条件实际上由两部分组成：① Execute 是虚函数；② 该函数属于线程类。虚函数的多态是我们复用的关键，只要子类覆盖即可重用整个逻辑，所以①似乎是不能动摇的，那就剩下②，该虚函数属于线程类 TThread。这个似乎能搞定，只要将虚函数 Execute()迁移到其他类上，就破坏了。线程中的 Wrapper()函数就是模板方法模式（见 5.2.5 节）。模板方法模式有一个等价的策略者模式，我们只要将 Execute()迁移到策略者类上，就搞定了。

策略者类叫 Runnable，代码如下：

```
class Runnable {
  public:
    virtual unsigned long execute() = 0;
};
```

TThread 在构造函数中传入 Runnable 的子类对象，并在 Wrapper()调用它的 Execute()方法：

```
class TThread {
```

```
    Runnable * FRunnable;
  public:
    TThread(Runnable * runnable){FRunnable = runnable};
}
void * TThread::Wrapper(void * arg) {
    TThread * Thd;
    Thd = ((TThread *) arg);
    unsigned long ret = Thd->FRunnable->execute();
    ...
};
```

定义 Runnable 的子类 TPrintRunnable：

```
class TPrintRunnable : public Runnable {
  protected:
    virtual unsigned long execute() { printf("hello I am in a thread\n"); ... };
};
```

使用 TPrintRunnable：

```
TThread * Thd;
Thd = new TThread(new TPrintRunnable());
Thd->run();
delete Thd;
```

TThread 执行析构等待线程结束时，并不影响 Wrapper()函数调用 Runnable 的虚方法 Execute()。因为此时是 TThread 对象的虚表被影响，并不影响其成员对象 FRunnable 的虚表，调用到的依然是 Runnable 子类 TPrintRunnable 的 execute。工程见“code\第 5 章\ThreadClass_CrossPlatform3”。

本章从线程类的构造到 bug 的发现与解决，无不体现了底层机制和分析的力量，也看到了不同语言机制处理问题时的多样性。希望大家通过本章的作业进一步强化，特别是习题 10，更是要深入到底层深处才能构建成功。

更多扩展

关于线程编程的进一步学习可参考《Windows 核心编程》和 Linux 的相关资料。

关于协程（coroutine），可参考 http://en.wikipedia.org/wiki/Coroutine，可根据“lightweight thread”查找学习用户级线程的概念。

关于跨平台实现，Delphi 中可阅读其 VCL（或 CLX）源代码（随机安装），或 Indy 源代码（关于网络编程接口的跨平台封装）。另外，这个库是学习 Internet 网络协议的好帮手，支持很多协议，只需简单运行其相关协议的例子并调试即可。C++中可学习 ACE 的源代码，它跨了 10 多种操作系统。Indy 和 ACE 源代码也是学习设计模式和网络编程的好范例。关于 GUI 跨平台封装的例子可参考 Delphi 的 CLX 框架和 QT 框架。

习 题 5

1．将跨平台线程类中定义的 Win32 去掉，编译并查看结果。加入 Linux 预定义宏，编译查看结果。在 Linux 上自己编写一个 make 文件，用 GCC 编译出可执行的版本。

2．将 5.3.3 节最后提到的函数指针方式实现 TThreadAPIBinder 的效果。

3．编写一个多线程应用，线程实现读取文件，并对整个文件求 CRC 校验值。预先生成 *N* 个线

程备用，当接收到 M 个文件处理任务后，用预先生成的线程处理这 M 个文件，处理完后线程不结束，返回等待任务状态。计算完毕将校验值和对应文件名信息保存。通过相关接口优雅结束整个程序的工作。提示：需要用到多线程同步机制，不用 C++，只用 C 语言实现，不用全局变量。

4．学习 Windows 和 Linux 多线程同步机制，封装跨平台类（封装的机制包括临界区和 0/1 事件（Windows 中的 Event）），并用该跨平台类以及跨平台线程类实现习题 3。

5．将习题 3 的实现方式修改为一个线程读文件和写文件，一个线程计算 CRC 校验，并比较两者的效率。

6．为什么本章中实现的线程类不能在构造函数中调用 CreateThread（除 5.4.2 节提示中给出的构造顺序的因素）？分析并给出证据，包括浮现错误的例子。

7 能否将本章中使用线程类的方式从 new 线程对象的方式修改为非指针方式，如下：

```
void SpawnThd(){
    MyThread thd;
}
```

如果是否定的，请给出分析并展示出错代码。

8．分析为什么在调试版本下，用调用堆栈界面观察调用序列中任意函数时，不能保证寄存器是有效的，却可以获取到该函数的有效参数和局部变量？参考 5.4.1 节的提示，通过画出栈映像给出详细说明。

9．请运用 5.4.2 节最后提及的策略者类实现整体资源管理多态化的思路，将该节中的 Communicator 类改造，实现 CloseChannel 的多态化。

10．用一个系统线程模拟多个线程运行，模拟的用户级线程（或轻量级线程 lightweight Thread，或协程 coroutine）要实现如下接口：

（1）线程函数原型为 void (*ThdFunc)(void *)。

（2）用户线程中调用 void YieldThd()完成切换到下一个线程运行，本线程停止运行。被切换运行的线程，如果是已经运行的，需要从上次它自己调用 YieldThd 之后代码运行。

（3）CreateThd(ThdFunc func, void * arg)，创建用户线程。

（4）StartThds()，开始调度线程。

（5）演示代码如下：

```
void thd2(void * arg) {
    for (int i = 4; i < 12; i++) {
        printf("thd2: arg=%d , i = %d\n", (int)arg, i);
        YieldThd();
    }
}
void thd1(void * arg) {
    if (CreateThd(thd2, (void *)2) == INVALID_THD) {
        printf("cannot create\n");
    }
    for (int i = 0; i < 12; i++){
        printf("thd1: arg=%d , i = %d\n", (int)arg, i);
        YieldThd();
    }
}
int main() {
    if (CreateThd(thd1, (void *)1) == INVALID_THD) {
```

```
            printf("cannot create\n");
        }
        StartThds();
        return 0;
    }
```

（6）测试

将（5）中的代码 thd1、thd2 的打印注释，然后将循环修改为无限循环，程序是否报错？

提示：① 可能要用__declspec(naked)函数；② 注意 context 切换，对于要保存的寄存器，可参考调试时寄存器窗口中给出的寄存器，尽量多保存和恢复寄存器；③ 要清晰函数调用的过程；④ 肯定要用汇编嵌入；⑤ 如果测试（6）出错，则认真分析调用和切换过程；⑥ 如果 C 库的函数在汇编中调用有问题，请用反汇编分析，C 库有动态和静态链接两种方式。

11．请用模板方法模式等价实现 5.2.6 节的 listener 模式。

第6章

插件养成记

需准备的知识

- 整章需要第1章的反汇编分析支持，特别是1.3节，最好完整阅读第4章。
- 整章需要4.7节和4.8.2节的知识。
- 6.5节需要1.8.2节和3.3节中的重定位及链接知识。

为你提供的能力

- 其基于接口编程的思想为第7章提供基础支撑。
- 学会一种实用的插件框架。
- 深入理解跨DLL内存释放问题，并找到两种解决方法。
- 6.5节帮助你进一步了解和掌握复杂问题的底层调试。
- 充分理解猜测、实证和构建方法。

在国内，几乎所有的程序员都或多或少地使用过 Microsoft Visual Studio。与笔者一样，也许你也遇到过同样的问题，想要统计当前工程中所有源文件的总行数。统计的方法有很多，从手工统计到自己动手写个小程序统计等。最简便的方法应该是在 Visual Studio 中安装使用代码行统计插件。笔者第一次接触这类插件时，感觉很神奇：一个已经通过编译、链接，并安装运行很久了的软件，居然还能动态地增加新功能。是的，完全不用重新编译链接已有的程序。不神奇吗？OK，来吧，本章会以若干个实例，将你带入一个插件系统的天地，让你感受如何完成变形金刚的壮举，为已有的软件系统动态地添加新功能。

6.1 一个修改已有功能的实例

如果你觉得不用重新编译链接已有的软件模块，就能添加新功能很厉害，那么，你是不是会觉得不重新编译链接已有的软件模块，就能修改已有的功能更厉害呢？我们还是从更厉害的入手吧。

可能有人会告诉你，“动态修改已有软件的功能，那是逆向工程师的事”。不不不，那些都是牛人，我们的目标只是告诉你基本的、简单的方法。现在要替换掉软件的某一项功能，但是又不需要像黑客一样去修改机器码，怎么办？一种最简单、最自然的想法很快浮现在我们的眼前：如果这个要被修改的功能是用一个动态链接库实现的，那就好办多了。动态链接库？如果你忘记了，请参考第 1 章和第 3 章。

现在，让我们隔着面纱看看这个例子吧。目前，该例子包含两个文件 main.exe 和 function.dll。main.exe 的主要代码见 DM6-1。在 DM6-1 中，第 7 行调用 LoadLibrary()函数加载动态链接库 Function.dll。之后，第 10 行调用 GetProcAddress()，获取名称为 Print 的函数地址，并在第 14 行调用该函数。DM6-2 给出了 Function.dll 的主要代码。

在 DM6-1 中，所要获取地址的函数即 DM6-2 中的 Print()函数。显然，该函数的主要功能就是向屏幕打印“Hello China!”。图 6.1 给出了 main.exe 程序的运行截屏。

DM6-1

```
1    int main(int argc, char **argv) {
2        HINSTANCE hinstLib;
3        PROC_PRINT Proc;                    //typedef void (__cdecl *PROC_PRINT)(void);
4        BOOL fFreeResult;
5        ...
6        // Get a handle to the DLL module.
7        hinstLib = LoadLibrary(TEXT("function.dll"));
8        ...
9        if (hinstLib != NULL) {
10           Proc = (PROC_PRINT) GetProcAddress(hinstLib, "Print");
11           ...
12           // If the function address is valid, call the function.
13           if (NULL != Proc) {
14               (Proc)();
15           }
16           ...
17           // Free the DLL module.
18           fFreeResult = FreeLibrary(hinstLib);
19       }
20       ...
21       getchar();
22   }
```

DM6-2

```
1   extern "C" __declspec(dllexport) void Print(void) {
2       cout<<"Hello China!"<<endl;
3   }
4   ...
5   BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved) {
6       ...
7   }
```

图 6.1

从图 6.1 可以看到，main.exe 的实际功能很简单，即向屏幕打印“Hello China!”。现在我们的任务是更改 main.exe 的功能，让其向屏幕打印“Hello World!”，但不能修改 main.exe 的源代码。如何实现？方法很简单，只需重新编写 function.dll 的源文件，见 DM6-3，再将重新编译链接好的 function.dll，替换掉旧的即可。图 6.2 给出了运行结果。

DM6-3

```
1   extern "C" __declspec(dllexport) void Print(void) {
2       cout<<"Hello World!"<<endl;
3   }
4   ...
5   BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved) {
6       ...
7   }
```

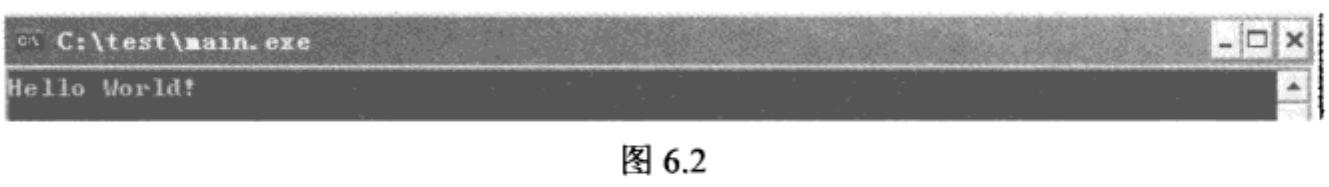


图 6.2

从这个例子中可以看出：不需更改 main.exe 的源代码，只需要替换 function.dll，就可以满足修改已有功能的要求。其根本原理在于两点：一是利用了动态链接库的基本特性，二是动态链接库导出了事先约定的接口。在这个例子里，事先约定的接口主要是指名为 Print 的函数及 Function.dll。换句话说，如果随意提供一个 DLL，它没有导出名为 Print 的函数，或者导出了但函数原型不一致，或者 DLL 的名称不是 Function.dll，则都将不能被 main.exe 程序所接受。当然在更通用的情形下，DLL 的名称不会被如此严格限制。因此，main.exe 程序之所以能被修改功能，关键在于所附属的动态链接库实现了事先约定的接口。通常，在插件系统中，这种动态链接库就被称为插件。

6.2 一个可以动态添加功能的简单实例

在 6.1 节的例子中，我们将打印“Hello China!”的功能动态修改为了打印“Hello World!”的功能。在本节中，我们做个小小的改动，既要打印“Hello China!”，又要打印“Hello World!”，甚至还要打印其他一些信息。也许你会说，“这还不简单吗”，在 main()函数中，直接调用 cout，想打印什么就打印什么。但是这里有个限制条件：程序事先并不知道需要向屏幕打印什么信息。而在这个程序编译链接完毕后，可以动态地向其增加打印何种信息的功能，即向已有程序动态添加功能。

怎样做到这一点？回顾 6.1 节的例子，我们用动态链接库来修改程序已有的功能。一个自然的

想法是：任何一种打印信息的功能都用一个动态链接库来实现。比如，用一个动态链接库实现打印“Hello China!”，另一个动态链接库实现打印“Hello World!”，若未来还需打印其他信息，也用一个动态链接库实现。

问题又来了：程序如何找到并加载这些动态链接库，以及如何调用这些库的打印功能？先解决后者。6.1 节其实已经给出了答案，就是规定所有的动态链接库必须导出事先约定的接口。这里包括导出函数的名称、原型必须一致。本节的例子继续沿用 6.1 节中介绍的接口函数 Print。

解决了后者，前一个问题又当如何？6.1 节的例子是为了修改已有的功能，所以两个动态链接库使用了同一个名称。显然，这里是行不通了。每个动态链接库都代表了某个打印功能，这些库都必须共存。因此，它们通常需要有不同的名称或在不同的路径中。另外，打印功能的数目事先不能确定，即动态链接库的数目不能确定，又当如何处理？一个简单的想法是：把所有的动态链接库都放在同一个目录中，假设该目录名称为“Plugin”；应用程序遍历这个目录，找出其中的动态链接库并加载之；加载之后，就可获取每个动态链接库的 Print 函数并调用，以打印有关信息。

DM6-4 给出了遍历 Plugin 目录、获取动态链接库名称的代码。遍历 Plugin 子目录，并获取插件名称，即动态链接库名称的功能，被放置在类 CPluginEnumerator 的 GetPluginNames()函数中。顾名思义，该类即进行插件的枚举。

DM6-4 的第 6 行是设置需要搜索的路径。为了简化起见，所有插件被放置在当前目录的 Plugin 子目录中。因此，常量 PLUGIN_PATH 的值被设置为"Plugin"。第 9 行调用 Windows 系统 API FindFirstFile，开始对插件所在的目录进行遍历。具体的遍历代码即第 16～44 行。对于 Plugin 子目录中的每个文件，都会进行如下类似的处理。在第 18 行，FindFileData.cFileName 即指向当前需要处理的文件名。由于我们需要查找出所有后缀为 .dll 的文件，因此不需要处理目录文件及文件名长度小于 4 的文件。所以在第 20～23 行中，如果该文件类型是目录，或者文件名长度小于 4，则不用进行任何操作，直接进入下一个循环，即获取下一个待处理的文件。

在 DM6-4 中，第 25～35 行的处理意图是判断文件名的后缀是否为 .dll。为了直观并便于阅读，我们采用了比较“死”的方法，即判断最后一个字符、倒数第二个字符是否为“l”或“L”，倒数第三个字符是否为“d”或“D”，倒数第四个字符是否为“.”。严格上讲，这种比较方法很不好，可扩展性和通用性都有很大问题，但我们讲述的重点并不是后缀名的比较，因此为了直观，姑且采用该方法。

当确认当前遍历的文件是动态链接库后，将形成带相对路径信息的动态链接库名，如 Plugin\function.dll，并将其保存在 vstrPluginNames 中，即第 37～42 行。因此，调用完上述函数后，调用者将获取到所有功能插件名称。

当获取到所有动态链接库后，程序需要逐一进行加载，并获取每个库的 Print 函数地址，并调用之。这一过程见 DM6-5。

DM6-5 第 5～7 行调用了类 CPluginEnumerator 的 GetPluginNames()函数，即 DM6-4，得到了插件或动态链接库名称列表。列表中的每个插件都将按照代码第 13～21 行处理。第 13 行调用了 LoadLibrary，加载某一插件。然后第 17 行调用 Windows 系统 API GetProcAddress()函数，获得 Print()函数的地址。第 21 行调用该 Print()函数。至于动态链接库的实现方式，与 DM6-3 类似。关键在于，这些动态链接库都必须要导出 Print()函数接口。

这样，放置在 Plugin 子目录中的所有动态链接库的 Print()函数都被调用了。显然，这是符合本节一开始提出的既要打印“Hello China!”，又要打印“Hello World!”，甚至还要打印未来才知晓的一些信息。图 6.3 给出了运行结果的截图。

DM6-4

```
1    bool CPluginEnumerator::GetPluginNames(std::vector<std::string>& vstrPluginNames)
2    {
3        WIN32_FIND_DATA FindFileData;
4        HANDLE hFind = INVALID_HANDLE_VALUE;
5        ...
6        std::string strDirSpec = PLUGIN_PATH;                          //可配置的路径
7        strDirSpec += "//*";
8        ...
9        hFind = ::FindFirstFile(strDirSpec.c_str(), &FindFileData);
10       ...
11       if (hFind == INVALID_HANDLE_VALUE)
12       {
13           return false;
14       }
15       ...
16       do
17       {
18           int length = strlen(FindFileData.cFileName);
19           ...
20           if((FindFileData.dwFileAttributes == FILE_ATTRIBUTE_DIRECTORY) || (length < 4))
21           {
22               continue;
23           }
24           ...
25           if(!((FindFileData.cFileName[length-1] == 'l') || (FindFileData.cFileName[length-1] == 'L')))
26               continue;
27           ...
28           if(!((FindFileData.cFileName[length-2] == 'l') || (FindFileData.cFileName[length-2] == 'L')))
29               continue;
30           ...
31           if(!((FindFileData.cFileName[length-3] == 'd') || (FindFileData.cFileName[length-3] == 'D')))
32               continue;
33           ...
34           if(!(FindFileData.cFileName[length-4] == '.'))
35               continue;
36           ...
37           std::string strName = PLUGIN_PATH;
38           strName += " \\";
39           ...
40           strName += FindFileData.cFileName;
41           ...
42           vstrPluginNames.push_back(strName);
43           ...
44       } while(::FindNextFile(hFind, &FindFileData));
45       ...
46       ::FindClose(hFind);
47       ...
48       return true;
49   }
```

DM6-5

```
1    int main(int argc, char **argv)
2    {
3        std::vector<std::string> vstrPluginNames;
4        ...
5        CPluginEnumerator enumerator;
6        if (!enumerator.GetPluginNames(vstrPluginNames))
7            return 0;
8        ...
9        for (unsigned int i=0 ; i<vstrPluginNames.size(); i++)
```

DM6-5（续）

```
10      {
11          PROC_PRINT DllProc;
12          ...
13          HINSTANCE hinstLib = LoadLibrary(vstrPluginNames[i].c_str());
14          ...
15          if(hinstLib != NULL)
16          {
17              DllProc = (PROC_PRINT)::GetProcAddress(hinstLib, "Print");
18              ...
19              if(NULL != DllProc)
20              {
21                  (DllProc)();
22              }
23              ::FreeLibrary(hinstLib);
24          }
25      }
26
27      getchar();
28  }
```

图 6.3

本节给出了为已有程序动态添加功能的方法。但是上述例子中所有动态链接库的打印功能都被调用了，即同时打印了“Hello China!”、“Hello World!”等信息。若是我们需要指定打印何种信息，即一次只调用一种指定的打印功能，又当如何？

6.3 一个可以动态添加功能的复杂实例

6.2 节描述了为已有程序动态增加功能的方法。但是，一次程序运行会调用所有动态链接库的打印功能。而现在需要做的是：一次只调用一种指定的打印功能。怎么办？既然要指定某种打印功能，那么至少需要让用户知道究竟有哪些打印功能。如何让用户知道当前程序拥有哪些功能？现在面临的情况是，程序不知道会有什么样的打印功能，以及未来会添加什么样的打印功能。是不是觉得很耳熟？对了，这就是我们 6.2 节所面临的问题。

与 6.2 节相同，每个打印功能依然由一个动态链接库实现。除了打印功能的导出接口 Print()函数外，每个动态链接库还需要导出一个 Help 接口，该接口通过向屏幕打印信息的方式，使用户知道其实现了何种打印功能，以及这种功能所对应的独一无二的标识。DM6-6 给出了该 Help 接口的实现方式示例。

DM6-6 中的第 1 行定义了该动态链接库所实现的功能的标识，而这个标识应该是独一无二的。通常为了保证这个标识的唯一性，会采用随机生成 128 位 GUID 的方式。但在这里为了简化讨论，我们采用整形类型保存功能 ID。第 8～11 行给出了 Help 接口的实现方式。如前所述，Help 接口的目的即在于使用户了解打印功能 ID 和具体的打印功能的对应关系。因此，Help 函数的实现很简单，即将插件所实现的功能 ID 和打印功能的描述输出到屏幕，以供用户查阅。

这样，每个插件或动态链接库的 Help 接口建立好了。那么，用户如何才能查看到每个插件所实现的功能说明呢？为了简化起见，我们采用通过命令行参数的方式进行处理，如图 6.4 所示。

DM6-6

```
1    const int FUNC_ID = 1;
2    ...
3    extern "C" __declspec(dllexport) void Print(void)
4    {
5        cout << "Hello World!" << endl;
6    }
7    ...
8    extern "C" __declspec(dllexport) void Help(void)
9    {
10       cout << "Function ID " << FUNC_ID << " : This function will print hello world." << endl;
11   }
12   ...
13   BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved)
14   {
15       ...
16   }
```

```
C:\test>main.exe help
```

图 6.4

输入命令行参数 help 时，即表示用户希望查看功能 ID 以及与之对应的打印功能描述。动态链接库的 Help 接口调用方式，同 6.2 节的示例类似。其处理过程见 DM6-7。

DM6-7

```
1    int main(int argc, char **argv)
2    {
3        ...
4        if(strcmp(argv[1], "help") == 0)
5        {
6            std::vector<std::string> vstrPluginNames;
7            ...
8            CPluginEnumerator enumerator;
9            if(!enumerator.GetPluginNames(vstrPluginNames))
10               return 0;
11           ...
12           for(unsigned int i=0 ; i<vstrPluginNames.size(); i++)
13           {
14               PROC_HELP DllProc;                          // typedef void (__cdecl *PROC_HELP)(void);
15               ...
16               HINSTANCE hinstLib = LoadLibrary(vstrPluginNames[i].c_str());
17               ...
18               if(hinstLib != NULL)
19               {
20                   DllProc = (PROC_HELP)::GetProcAddress(hinstLib, "Help");
21                   ...
22                   if(NULL != DllProc)
23                   {
24                       (DllProc)();
25                   }
26                   ::FreeLibrary(hinstLib);
27               }
28           }
29           ...
30           return 0;
31       }
32       ...
33       return 0;
34   }
```

在 DM6-7 中，第 4 行将命令行参数与字符串"help"比较，即判断当前用户是否希望查询功能 ID 及功能描述。若期望，即会按照与代码 DM6-5 相类似的处理方式，加载所有的动态链接库即插件，获取 Help 导出函数地址，并调用之。每个动态链接库的 Help 函数都会打印出功能 ID 和打印功能的描述。图 6.5 给出了“main.exe help”的输出结果。

```
C:\test>main.exe help
Function ID 1 : This function will print hello world.
Function ID 2 : This function will print hello china.
```

图 6.5

从图 6.5 可知，用户输入“main.exe help”后，即可获知功能 ID 及其对应的打印功能说明。为了简化起见，当用户需要调用 main.exe 的某项功能时，只需通过命令行参数指定该功能的 ID 即可。显然，程序能通过解析命令行参数获知用户需要调用何种打印功能。问题又出来了：main.exe 程序如何能知道究竟哪个动态链接库实现了用户所需功能？用户能告知 main.exe 的只是一个功能 ID，若动态链接库也能导出一个接口告知其所实现功能的 ID，那么 main.exe 就能比较这两个 ID 是否一致。若相同，显然该动态链接库实现了用户所需的打印功能。DM6-8 给出了动态链接库所实现的获取功能 ID 的导出接口 GetID。

DM6-8

```
1    extern "C" __declspec(dllexport) int GetID(void)
2    {
3        return FUNC_ID;
4    }
5    ...
6    BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved)
7    {
8        ...
9    }
```

从 DM6-8 可知，GetID()导出接口的实现非常简单，直接将动态链接库所实现功能的 ID 返回即可。这样，阻挡我们前进的关键技术都已经解决了。

程序处理的基本思路是：程序在初始化时，需要加载各个插件，并获取它们的 Print()函数、GetID()函数的地址，并保存；用户通过命令行参数传递其需要调用的功能的 ID；获取到该 ID 后，即可调用所有插件的 GetID()函数，直到 GetID()的返回值与用户指定的 ID 相同，即可由此确定实现了该功能的插件；再调用该插件的 Print 导出函数，以获取指定的打印功能；最后，程序结束时，进行各插件的卸载工作。DM6-9 给出了程序初始化时，加载各插件，以及获取相应接口函数的实现方法。

类 CPluginController 的主要工作在于初始化和反初始化插件及实现插件的调用。DM6-9 给出了该类的 InitializeController()函数，该函数实现了插件发现及插件的加载。同 DM6-5 的处理方法类似，DM6-9 的第 5～7 行调用了类 CPluginEnumerator 的 GetPluginNames()函数，即 DM6-4，得到了插件或动态链接库名称列表。列表中的每个插件都将按照代码第 11～22 行处理。

DM6-9 的第 11 行调用了 LoadLibrary 加载某一插件。加载成功之后，第 15 行即将该插件的实例句柄，保存在成员变量 m_vhForPlugin 容器中。其目的主要在于，当进行反初始化时，可以将加载了的动态链接库卸载掉。之后，代码第 17 行和第 18 行分别调用 GetProcAddress()函数，以获取每个插件的 Print()函数、GetID()函数的地址，并将这些地址分别保存在成员变量 m_vPrintFunc 容器和 m_vGetIDFunc 容器中。上述三个成员变量的定义，见 DM6-10。

DM6-9

```
1   bool CPluginController::InitializeController(void)
2   {
3       std::vector<std::string> vstrPluginNames;
4       ...
5       CPluginEnumerator enumerator;
6       if(!enumerator.GetPluginNames(vstrPluginNames))
7           return false;
8       ...
9       for(unsigned int i=0 ; i<vstrPluginNames.size(); i++)
10      {
11          HINSTANCE hinstLib = LoadLibrary(vstrPluginNames[i].c_str());
12          ...
13          if(hinstLib != NULL)
14          {
15              m_vhForPlugin.push_back(hinstLib);
16              ...
17              PROC_PRINT DllPrint = (PROC_PRINT)::GetProcAddress(hinstLib, "Print");
18              PROC_GETID DllGetID = (PROC_GETID)::GetProcAddress(hinstLib, "GetID");
19              if((NULL != DllPrint) && (NULL != DllGetID))
20              {
21                  m_vPrintFunc.push_back(DllPrint);
22                  m_vGetIDFunc.push_back(DllGetID);
23              }
24          }
25      }
26      ...
27      return true;
28  }
```

DM6-10

```
1   typedef void (__cdecl *PROC_PRINT)(void);
2   typedef void (__cdecl *PROC_HELP)(void);
3   typedef int (__cdecl *PROC_GETID)(void);
4
5   class CPluginController
6   {
7     public:
8       CPluginController(void);
9       virtual ~CPluginController(void);
10      ...
11      bool InitializeController(void);
12      bool UninitializeController(void);
13      ...
14      bool ProcessHelp(void);
15      bool ProcessRequest(int FunctionID);
16      ...
17    private:
18      std::vector<HINSTANCE> m_vhForPlugin;
19      std::vector<PROC_PRINT> m_vPrintFunc;
20      std::vector<PROC_GETID> m_vGetIDFunc;
21  };
```

从 DM6-10 可知，我们用 3 个 vector 容器分别保存动态链接库的实例句柄、Print()函数和 GetID()函数的地址。InitializeController()函数执行完毕，即加载完所有插件后，程序即可根据用户传递而来的功能 ID 查找具体插件，并调用其打印功能。DM6-11 描述了这一过程。

从 DM6-11 可知，CPluginController 类的 ProcessRequest()函数负责根据用户传递而来的功能 ID，查找实现了该对应功能的插件，并调用其打印功能接口。代码第 3 行，即对保存了功能 ID 的 vector

DM6-11

```
1   bool CPluginController::ProcessRequest(int FunctionID)
2   {
3       for(unsigned int i = 0; i < m_vGetIDFunc.size(); i++)
4       {
5           if((m_vGetIDFunc[i])() == FunctionID)
6           {
7               (m_vPrintFunc[i])();
8               break;
9           }
10      }
11      ...
12      return true;
13  }
```

容器进行遍历。在第 5 行，调用了插件的 GetID()接口获取其实现的功能的 ID，并与用户指定的功能 ID 进行比较。若两个 ID 相同，则说明当前插件实现了用户所需功能。因此，第 7 行调用了相应插件的 Print()函数接口。

CPluginController 类的 UninitializeController()函数实现方式很简单，即在程序退出之前，遍历 m_vhForPlugin 容器，对保留其中的每个句柄调用 FreeLibrary，以卸载动态链接库。

这样，我们就分析完了一个较复杂的动态增加功能的实例——可以让用户指定要调用何种功能。图 6.6 给出了本节实例的运行效果。

图 6.6

6.4 从函数到插件对象

在通常情况下，动态链接库提供给外界的使用方式就是各种各样的导出函数。同样，在 6.2 节的例子实现了具体打印功能的各插件，它们对外导出了 Print()函数、GetID()函数、Help()函数。按照 6.3 节讨论的实现方式，我们使用两个 vector 容器，分别保存所有插件的 Print()函数地址和 GetID()函数地址。6.3 节给出的实例的业务功能十分简单，在现实情况中，可能遇到十分复杂的情形，插件需要导出的函数远远不止这两个。这样，若还按照 6.3 节的处理方式，将会使用很多 vector 容器，以分别保存这些众多的导出函数。如此操作，加大了程序开发的难度，降低了代码的可维护性。

这时你可能会问，“有没有什么好的办法，可以有效地组织管理这些导出函数呢？”答案当然是肯定的。类这个东西，应该会很快浮现在你的脑中。对了，就是它，它能把若干函数和数据封装在一起。结合到 6.3 节例子，所有的动态链接库都需要导出 Print()、GetID()、Help()三个函数，自然会想到把这三个函数都放在一个类中。每个具体的需要实现打印功能的插件都可从这个类派生出自己的类，并给出自己的实现。这里，假设这个基类名称为 IPrintPlugin，其意思是实现打印功能的插件的接口。如何从一个 DLL 中导出对象？这个问题在 4.8.2 节中已经解决了。

DM6-12

```
1   class IPrintPlugin
2   {
3     public:
4         ...
5         virtual void Help() = 0;
6         virtual void Print() = 0;
7         virtual int GetID() = 0;
8   };
```

DM6-12 给出了 IPrintPlugin 的定义。类 IPrintPlugin 仅给出了三个纯虚函数，即 Help()、Print()和 GetID()。每个要实现打印功能的动态链接库都需要从 IPrintPlugin 派生类，并给出派生类的这三个函数的实现。这三个虚函数的实现方式与 6.3 节讨论的类似，这里不再赘述。

我们已经把三个函数统一管理起来了，现在需要解决的问题是，怎样才能把 IPrintPlugin 派生类的对象或者对象的指针，传递给插件的调用者，以使其能访问 Help()等三个函数。没有什么好办法了，只能利用动

态链接库可导出函数供外界使用这一基本特点了。在本节的示例中，我们假设所有的插件都需要导出 CreateObj()函数，通过该函数可获取到 IPrintPlugin 派生类对象的指针。DM6-13 描述了 CreateObj()函数。

DM6-13

```
1    extern "C" __declspec(dllexport) void CreateObj(IPrintPlugin **ppPlugin)
2    {
3        static CPrintPlugin plugin;
4        *ppPlugin = &plugin;
5    }
6    ...
7    BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved)
8    {
9        ...
10   }
```

在 DM6-13 中，第 3 行给出了类 CPrintPlugin 的一个静态对象。其中，CPrintPlugin 派生于 IPrintPlugin，它实现了具体的打印功能。第 4 行将该静态对象的地址，通过参数，传递给插件的调用者。这样，插件调用者即可使用该对象。这种对象被称为插件对象。这样，类 PluginController 将不会再直接使用 Print()等导出函数了，将通过调用 CreateObj()函数，获得 IPrintPlugin 接口指针，再通过该指针调用具体的方法，如 Print()、GetID()等。DM6-14 简要描述了这一过程。

DM6-14

```
1    bool CPluginController::InitializeController(void)
2    {
3        ...
4        for (unsigned int i=0 ; i<vstrPluginNames.size(); i++)
5        {
6            typedef int (_ _cdecl *PLUGIN_CREATE)(IPrintPlugin**);
7            PLUGIN_CREATE CreateProc;
8            ...
9            IPrintPlugin *pPlugin = NULL;
10           ...
11           HINSTANCE hinstLib = LoadLibrary(vstrPluginNames[i].c_str());
12
13           if (hinstLib != NULL)
14           {
15               m_vhForPlugin.push_back(hinstLib);
16               ...
17               CreateProc = (PLUGIN_CREATE)::GetProcAddress(hinstLib, "CreateObj");
18               ...
19               if (NULL != CreateProc)
20               {
21                   (CreateProc)(&pPlugin);
22                   ...
23                   if (pPlugin != NULL)
24                   {
25                       m_vpPlugin.push_back(pPlugin);
26                   }
27               }
28           }
29           ...
30       }
31   }
```

在 DM6-14 中，第 11 行和第 17 行加载插件并获得 CreateObj()函数地址后，即在第 21 行调用了该函数，获取到插件对象指针。第 25 行将该指针保存在了成员变量 m_vpPlugin 容器中。那么，

如何使用该插件对象指针，访问 IPrintPlugin 类的虚方法呢？DM6-15 给出了答案。

DM6-15

```
1    bool CPluginController::ProcessRequest(int FunctionID)
2    {
3        fo r(unsigned int i = 0; i < m_vpPlugin.size(); i++)
4        {
5            if (m_vpPlugin[i]->GetID() == FunctionID)
6            {
7                m_vpPlugin[i]->Print();
8                break;
9            }
10       }
11       ...
12       return true;
13   }
```

从 DM6-15 可知，IPrintPlugin 类虚方法的访问方式与一般类虚方法访问方式完全一致，并不会因为插件对象是在动态链接库中产生而任何不同。第 5 行调用了插件对象的 GetID()方法，获取其实现的功能的 ID。第 7 行则调用了 Print()方法，直接获取插件所实现的打印功能。

DM6-16

```
1   class IPlugin
2   {
3     public:
4       virtual bool Initial() = 0;
5       virtual bool Uninitial() = 0;
6       ...
7       virtual bool Start() = 0;
8       virtual bool Stop() = 0;
9       ...
10      virtual int GetID() = 0;
11  };
```

到目前为止，我们打印功能插件的基本原理的讨论可以告一段落了。但是尚有一些重要问题还未提及。DM6-12 给出了 IPrintPlugin 的定义，从该类的名称和其中定义的函数可以看出，IPrintPlugin 与本章所讨论的打印功能是密切相关的，其 Print()、Help()等函数与业务逻辑也是紧密联系的。实际上，许多插件接口都提供了诸如 Initial（初始化插件）、Uninitial（反初始化插件）、Start（启动插件）、Stop（停止插件）等函数。类似这些函数，通常应是各种类型插件共有的。因此，我们可以建立一个基类，如类 IPlugin，以包含这些共有的方法。这样，本节中的 IPrintPlugin 类即可从 IPlugin 类中派生。DM6-16 给出了 IPlugin 类的一种典型定义。

目前，插件的函数都是虚函数。必须如此吗，能否改成非虚的成员函数呢？下面以 Print()函数为例进行说明。让我们回忆第 3 章学到的编译知识并在头脑里模拟编译过程。编译阶段当编译器发现 Print 这个符号后，仅仅在引用该符号的 CPP 文件对应的 OBJ 文件中添加该符号（准确地说是添加 IPrintPlugin::Print 符号）。在 OBJ 文件中，对 Print 的调用是一条相对 call 指令，其偏移量为 0。链接时，找到真正实现 Print()函数的 OBJ，并将其代码链接入执行程序，这时就可计算出之前那条相对 call 指令的转跳偏移量了。如果找不到实现 Print 的 OBJ 呢？只有报链接错误。

Print()函数实现在哪里呢？在 DLL 中，不是 OBJ 中，自然只有报错了。当 Print()函数是虚函数时，情况就不一样了。这时第 4 章关于虚函数的调用原理及反汇编码给出了我们不一样的答案。当编译器遇到 Print 符号时，因为从头文件中发现是虚函数，那么它不会生产相对 call 指令，也不会像上面那样在链接阶段企图寻找实现 Print 的 OBJ 并计算 call 的偏移量重定位。这时它生成的指令序列是从对象指针指向的前 4 字节获取虚表，然后从虚表对应位置获取函数入口，并最终跳过去（指令见 DM4-57(b)的第 1～5 行）。这些指令不需要进行相对 call 指令那样的重定位，所以不会出现链接错误。

目前的代码还有一点隐藏的瑕疵，在代码 DM6-13 中，CreateObj()返回的是静态对象的地址。显然，这不需要由我们来释放对象内存。同时，其弊端就是只有一个对象，一种类型的插件需要生

成多个插件实例时，则不适用了。因此，需要在 CreateObj()中返回用 new 动态生成的插件实例；自然，也需要用 delete()来按需删除对象（否则内存泄漏）。这似乎直接而简单，我们完全不知道这意味着怎样的磨难。

6.5 delete 的灾难：谁的书

4.8.2 节给出了简单的对象导出例子，细心的读者会发现它没有释放对象。本节将深入探讨相关问题并寻求解决方案。

6.5.1 释放内存的崩溃

将工程“code\第 4 章\import_obj”复制并修改，见“code\第 6 章\import_obj”。首先做以下改动，对两个工程做如下操作：右键单击，在弹出的快捷菜单中选择“属性”，在弹出的窗体左边的树中选择“配置属性 | C/C++ | 代码生成”；然后在右边列表“运行库”下拉框中选择“多线程调试(/MTd)”；在左边的树中选择“配置属性 | 链接器 | 高级”，然后在右边“随机基址”的下拉框中选择“禁用映像随机化(/DYNAMICBASE:NO)”。接着修改 main 函数，见 DM6-17。

运行该程序，我们发现程序抛出异常，似乎是一个叫_CtrIsValidHeapPointer()的函数出错了。为了进一步分析，在 DM6-17 的代码第 3 行设置断点，断点命中后，按 F11 键，跟踪进 delete 操作符。跟踪到 DM6-18 所示的代码时出错，见图 6.7 的提示。

DM6-17

```
1   void main(){
2       Parent* p = createObj();
3       delete p;
4   }
```

DM6-18

```
return HeapValidate( _crtheap, 0, pHdr(pUserData) );
```

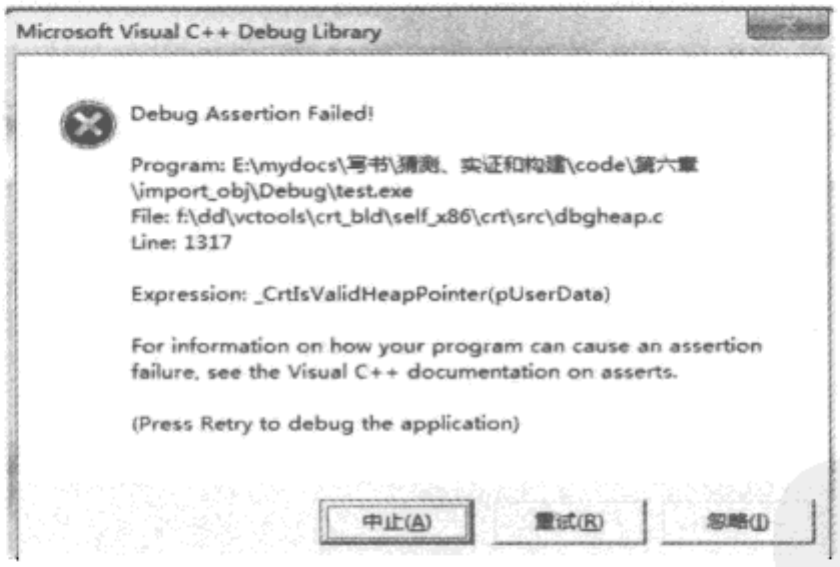


图 6.7

源码级跟踪进 HeapValidate()的企图失败，可见该函数可能是系统 API。上网查找发现，该函数确为 Windows 的 API，相关函数还有 HeapAlloc()、HeapFree()、HeapCreate()等，它们实现了小块内存分配。看来 Visual C++的 C 库用这些 HeapXXX 函数分配内存。为什么 HeapValidate 会失败？HeapValidate 的说明有如下内容：“You can also use the HeapValidate function to validate a single memory block within a specified heap.”看来该函数可验证一个内存块是否属于某个堆。HeapValidate

原型如下：

```
BOOL HeapValidate(
    HANDLE hHeap,     // handle to the heap of interest
    DWORD dwFlags,    // bit flags that control heap access during function operation
    LPCVOID lpMem     // optional pointer to individual memory block to validate
);
```

第 1 个参数标识要判断的堆，第 3 个为被判断的内存地址。（其他不懂的符号与主题无关，忽略。我们关注的是地址，什么类型的地址都是一个整数，所以忽略。要学会“忽略”，这是一种重要的探索能力。）

在 DM6-18 中调用 HeapValidate 失败，是否意味着 pUserData 指向的内存块不属于_crtheap 指向的堆？是否 pUserData 就是 DM6-17 中那个对象的地址 p？我们在 DM6-17 第 3 行设置断点，命中断点后，记录下 p 的值 0x003c3ca8，再在 DM6-18 所示语句处设置断点，然后按 F5 键，继续执行程序，断点命中后，查看 pUserData 的值也是 0x003c3ca8。也就是说，对象的内存确实不属于_crtheap 指向的堆。为什么会这样？对象从 DLL 中分配来的，这是否意味着执行程序和 DLL 有各自的堆？如果我们想象堆像一个图书馆，分配和释放内存，就像借书和还书，那么两个不同的堆就像有两个不同的图书馆，从一个馆借书还给另一个馆，肯定会造出混乱。一个图书馆的书，对另一个图书馆肯定就是无效书籍。HeapValidate 也许正是验证一本书是否属于某个馆的。

有了这个比喻及其猜测，那我们就要验证 DLL 和执行程序是否有不同的堆。既然 delete 时最终导致调用 HeapValidate，而该函数第 1 个参数是堆的标识句柄，那么我们在 DLL 和执行程序中分别执行 delete，来看 HeapValidate 的第 1 个参数是否不同。在 CreateObj 中增加构造和释放的代码，见 DM6-19，对比此 delete p 调用时最终传给 HeapValidate 的堆句柄和执行程序（DM6-17 第 3 行）所传堆句柄值是否相同，即可得证。如何调试动态链接库呢？动态链接库自身无法直接运行，因此首先需要一个调用 DLL 函数的执行程序，然后告诉调试环境该程序在哪里，调试器就会去启动该执行程序。执行程序调用 DLL 的函数时，就会在我们设定的断点处停下，激活 DLL 项目的工程属性界面，如图 6.8 所示，在命令项中输入调用该 DLL 的程序（输出目录下叫 test.exe 的程序），然后将 DLL 项目设为启动项。

DM6-19

```
Parent* createObj(){
    Parent * p = new Child();
    delete p;
    return new Child();
}
```

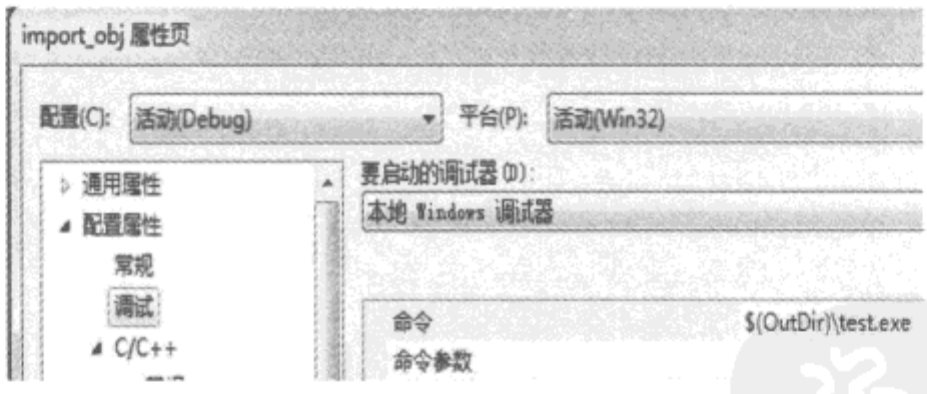


图 6.8

按 F5 键，调试运行即可跟踪 DLL。在 DM6-19 的 delete 处设置断点，断点命中后，在 DM6-18 的语句设置打断点，命中后，查看堆句柄为 0x005d0000。该句柄是 DLL 所用堆的值。实际执行该语句，没出错。这也部分验证了我们的猜测，因为自己堆上分配的对象自己释放，没出错，是合法的。继续执行程序，它将在执行程序的 delete 语句（DM6-17）的断点处暂停。下面执行 HeapValidate 时需判断传给它的堆句柄是否与 0x005d0000 相同。此时的句柄为 0x01500000，果然不同，说明 DLL 和执行程序的 C 库拥有各自不同的堆。

为了判断“图书馆比喻”的有效性，此时将传给 HeapValidate 的句柄修改为 0x005d0000，即从 0x005d0000 堆借出的书还给 0x005d0000 堆，这样应该不会报错了。不要直接修改全局变量_crtheap，这样可能导致后续用到_crtheap 时出错。此时反汇编，见 DM6-20。我们只需在第 6 行执行前通过寄存器窗体将 ECX 的值修改为 0x005d0000 即可。此时，程序成功通过了之前失败的 HeapValidate 调用（第 7 行）。当然，后续其他有效性判断依然失败。

DM6-20

```
    return HeapValidate( _crtheap, 0, pHdr(pUserData) );
1   00430c10    mov     eax, dword ptr [puserdata]
2   00430c13    sub     eax, 20h
3   00430c16    push    eax
4   00430c17    push    0
5   00430c19    mov     ecx, dword ptr [__crtheap (494874h)]
6   00430c1f    push    ecx
7   00430c20    call    dword ptr [__imp__heapvalidate@12 (497274h)]
```

现在已经证明了“图书馆比喻”的正确性。从技术层面，为什么执行程序和 DLL 各自有两个不同的堆？这要从本节一开始修改运行库（即 C 库）属性为“多线程调试(/MTd)”说起。它意味着以静态链接方式链入 C 库代码。静态链接成的程序必须包含所有要执行的代码和全局变量（因为没有动态库为它提供所需代码，它必须自己携带它们）。在第 3 章已清楚静态链接的过程：当程序调用 C 语言库函数 A 时，链接器将把 C 语言的静态库中函数 A 的代码复制到生成的程序中，要使用的全局变量（如 DM6-18 的_crtheap）也在程序全局数据区。也就是说，链接 DLL 时，因为 DLL 要用 C 语言库的 new、delete 代码和全局变量，它们被复制到 DLL 中；链接 EXE 时，因为 EXE 也需要它们，所以也被复制到 exe 中。DLL 和执行程序中会有两份一样的 C 库语言代码（new、delete 等）和全局变量（_crtheap），见图 6.9。

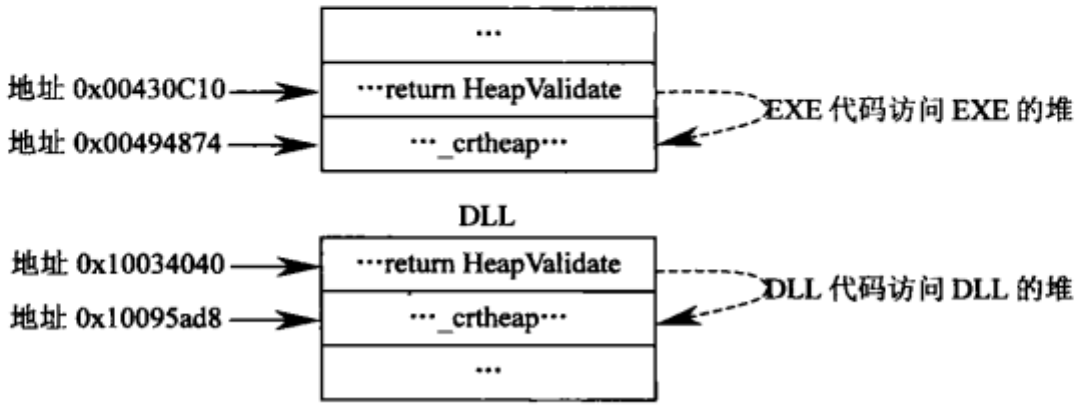


图 6.9

为什么有了两套相同的代码和全局变量，就会有不同的堆？不急，我们先证明“两套相同代码和全局变量”这一结论。为此，在 DLL 代码（DM6-19）的 delete 最终调用到 HeapValidate 语句（DM6-18）时，以及执行程序代码（DM6-17）的 delete 最终调用到 HeapValidate（DM6-18）时，查看 EIP 寄存器的值，如果不同，则说明貌似一样的代码其实是内容相同的两段指令，从而证明了图 6.9 的推论。先在 DM6-19 的 delete 语句设置断点，命中断点后，在 DM6-18 中的语句处设置断点，按 F5 键，恢复程序运行，在新断点处暂停，此时 EIP 为 0x10034040。然后取消 DM6-18 语句的断点（以防其他地方导致执行该语句时干扰视线），并在 DM6-17 的 delete 语句处设置断点，按 F5 键，恢复运行，命中新断点后再次在 DM6-18 中的语句处设置断点，然后按 F5 键，执行并停在该断点处。此时 EIP 的值为 0x00430c10。由此可见，虽然都是 DM6-18 的语句，其指令地址却不同，一个是 0x10034040，一个是 0x00430c10。在查看 EIP 时分别在监视窗体中查看&_crtheap 的值，发

DM6-21(a)

```
void * operater new(int size) {
    if (_crtheap == NULL)
        _crtheap = HeapCreate();
    return HeapAlloc(_crtheap, .., size);
}
```

现全局变量_crtheap 的地址也是不同的值，见图 6.9。由此推论成立。我们也发现，DLL 文件的 HeapValidate 使用的是 DLL 的_crtheap，EXE 文件也如此。

现在来分析两套相同代码和全局变量为什么就导致产生两个堆。现在我们从构建者角度来分析如何撰写 new 这样的函数。既然从 Windows 提供的小块内存分配 API 获取内存，那么自然要用到前面查到的 HeapAlloc。而它需要先分配一个堆，所以需先调用一次 HeapCreate 创建一个堆，然后每次分配内存时调用 HeapAlloc，从该堆上分配内存，伪代码见 DM6-21(a)。因为 DLL 和 EXE 文件都有如此一套代码和_crtheap 变量，当 DLL 的 new 操作符执行时，此时 DLL 的_crtheap 为 NULL，所以 HeapCreate 会执行。当 EXE 的 new 第一次执行时，因其 if 判断的是 EXE 的_crtheap 变量，它依然为 NULL（之前调用 dll 的 new、HeapCreate 的返回值赋给了 DLL 的_crtheap，而非 EXE 的），所以 HeapCreate 会再次被调用，并将返回值赋给 EXE 文件的_crtheap。这样就产生了两个堆。

猜测了还得实证，看看真实的情况。我们用 total commander 软件查找 C 语言库的源码（在 VS 安装目录\VC\crt\src 下，笔者的是 D:\Program Files\Microsoft Visual Studio 9.0\VC\crt\src），寻找其中包含 HeapCreate 字符串的 .c 文件，找到两个：smallheap.c 和 heapinit.c，其中均有一个_heap_init 函数调用了 HeapCreate。现摘录一个简单的，见 DM6-21(b)。它们均直接调用了 HeapCreate 创建堆，可见_heap_init 应该只被执行了一次。否则，调用两次就会创建两个堆。只有一个_crtheap 保存堆句柄，如果第二次赋值_crtheap 前不释放前一个堆，那么势必造成前一个堆无法释放（内存泄漏）。另外，如果按 F11 键跟踪进 new 操作符，我们发现，它最终调用了 HeapAlloc()函数，这一点与 DM6-21(a)的猜测符合。

DM6-21(b)

```
int _ _cdecl _heap_init ( int mtflag ) {
    if ( (_crtheap = HeapCreate( mtflag ? 0 : HEAP_NO_SERIALIZE, BYTES_PER_PAGE, 0 )) == NULL )
        return 0;
    return 1;
}
```

下面实证我们的分析：对于 DLL 和 EXE 各自 C 库的初始化而言，_heap_init 只执行了一次。在两个文件的 HeapCreate 语句处设置断点（在 VS 中手动打开 smallheap.c 和 heapinit.c，再设置断点），然后运行程序，发现 heapinit.c 中的_heap_init 被执行了 2 次，一次是调用 DLL 的（先执行），一次是调用 EXE 的。图 6.10 展示了两次命中断点时的状况，其对应截图分属虚线分隔的上下两部分。从跟踪发现，虽然执行了两次_heap_init，但实际是分属 DLL 和 EXE 的两套代码（中断点命中时 EIP 的值不同），而被赋值的_crtheap 地址也不同（见图 6.10 的监视窗体）。因此，对两套 C 语言库代码而言，堆初始化函数_heap_init 只执行了 1 次。从右下角的调用堆栈可知，一个是从 DLLMainCRTStartup 最终调用到_heap_init，一个是从 mainCRTStartup 最终调用到_heap_init。DLLMainCRTStartup 为 DLL 的 C 语言库提供了初始化，mainCRTStartup 为执行程序的 C 语言库提供了初始化。跟踪还发现，main()函数在这两个函数完成后才执行。这也好理解，因 C 语言库没在 new()函数中判断是否该分配堆，所以它必须在 main()之前执行_heap_init 分配堆，否则调用 new()时堆还未分配，就会出错。

虽然我们从“图书馆比喻”易知，不属于某个堆的内存块不能还给该堆。但从技术细节而言，为什么不能还，为什么要有 DM6-18 那样的判断语句？如果错了，会发生什么问题？因为堆的分配

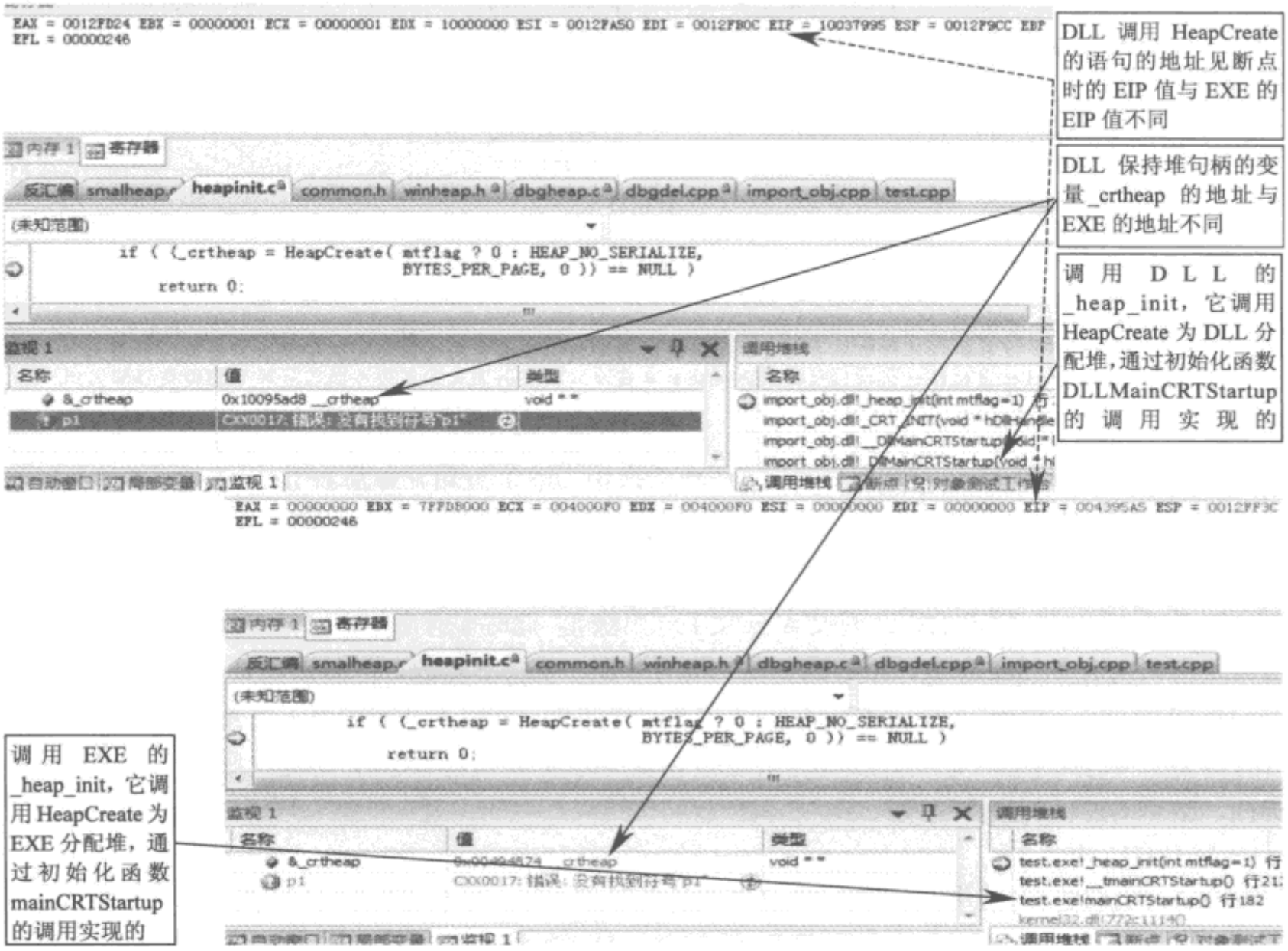


图 6.10

和回收算法是 kernel32 实现的，我们不易分析。但从构建角度，可做如下理想实验，以窥究竟。因为操作系统分配内存是按页（Windows 一页为 4KB）分配的，粒度很大，如果想分配 1 字节，也不得不先分配 4KB。为了不浪费，我们需要实现一套算法和数据结构，它将一页内存根据用户需求划分成更小的块，并将块地址返回给用户。这就是 malloc 完成的工作。为了分配和回收，自然的想法是将释放的块串成可用链。这就需要将块定义为链表结构。另外，我们调用 free(p)释放内存时，还需要知道该内存到底是多大，这也需要记录在块信息中。因为指针 p 指向的内存是分配给客户使用的，长度信息只能放在比 p 指向地址小的区域，类似第 3 章图 3.6 所示的字符串存储长度的方式。这样有图 6.11(a)所示的块结构，假定对应的释放函数为 free1()，再假定另外一个软件实现的块结构如图 6.11(b)所示。如果将该格式内存块传递给 free1()释放，free1()必然将长度与 next 信息搞反，必然出错。这相当于两个图书馆的书所帖的分类标签格式不同，条纹码不同，如果还错，这会让管理员（free1()）无所适从。

图 6.11 的猜测并非虚妄，编写代码“p = malloc(12);”，然后查看比 p 小的地址区是否真的存储了长度 12。在内存窗体地址栏输入 p 指向的内存 0x1423AA8，回车，然后往上滚动一行，如图 6.12 所示。实线所指为 p 指向的内存，虚线所指正好是 0c，即长度信息 12。修改 malloc 参数，反复实验，可知虚线所指数据均是 malloc 参数，从而可确定这就是长度信息存储处。明显，这是一个复杂的数据结构，因为长度信息右边还有 12 字节才是用户存储信息的区域，这 12 字节必有所指。可见实际结构已与我们假想的结构（图 6.11）完全不同了（如果单步跟踪 malloc，会发现这是 C 语言库自定义的内存块结构）。这种不同必然要求我们不能将一种算法分配的块让另外一种算法释放。因此，需要检验块是否合法（HeapValidate）。

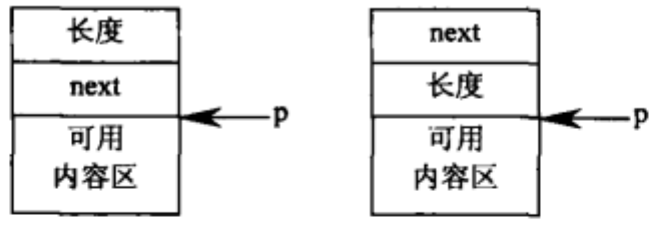


图 6.11

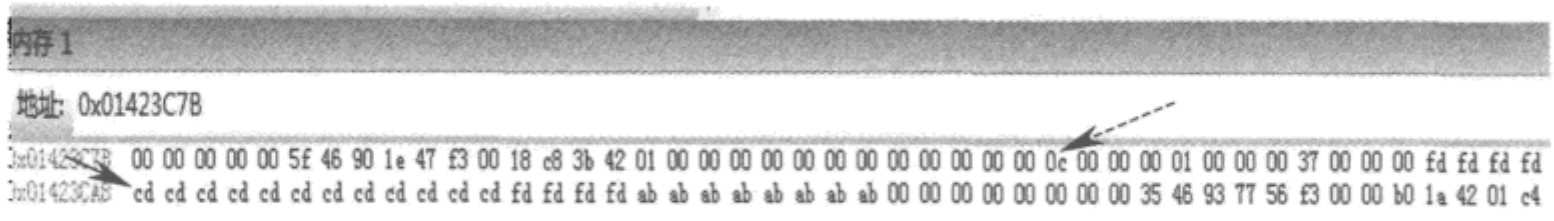


图 6.12

即使两个堆采用完全一样的分配算法和数据结构，就可以相互释放对方分配的块了吗？如果 A、B 两馆条纹码格式相同，但 A 馆的书编号段为 1～6，B 馆的为 10～16，那么将 A 编号为 2 的书还给 B，B 的管理员会发现多了一个无法归类的编号为 2 的书，同时 A 的管理员却始终认为丢了一本编号为 2 的书。技术上讲：假设 A 分配了一个块 a，此时 a 所在的页 P 只分配了 a 一块内存。如果正常调用 A 的 free 释放，A 必然发现 P 没有分配，就将 P 释放。但假如调 B 的 free 释放 a，B 在它管辖的堆中（如由_crtheap 指示）无法找到 a 所属页，所以无法释放 P，而 A 也失去正常释放 P 的机会了（因为最后一块分配的内存 a 已经错误地通过 B 假释放了）。这是一种另类内存泄漏。因此，依然需要检查块合法性。

6.5.2 解决之道：新生活，各管各

如何解决 6.5.1 节的问题呢？谁分配谁释放就不会出现还错“书”的问题。只要能让执行程序调到 DLL 的 delete()函数就好，因为 DLL 的 delete()将使用 DLL 的堆（见图 6.9），这样从 DLL 堆中分配的对象最终就还给 DLL 了。来个偷天换日，思路如下：在 DLL 的代码 DM6-19 的 delete 语句处设置断点，从其反汇编获取到调用操作符 delete 函数的指令，并记录下真正 delete()函数的地址；然后在执行程序代码 DM6-16 的 delete 语句处设置断点，程序运行到该断点后，展开反汇编，单步执行到 call 指令，它调用操作符的 delete()函数。此时，如果执行 call，将跳到执行程序的堆释放代码；如果让它跳到刚才记录下的 DLL 的 delete()函数的入口，需要怎么做？

移花接木法

第一，将 ESP 减 4，并通过内存窗体在栈上存储返回地址。这模拟 call 指令压栈效果（因为我们不直接调用 call 指令，所以必须手动模拟）。第二，通过寄存器窗体修改 EIP 的值，让它指向 DLL 的 delete()函数入口。该“移花接木”可常常用于改变程序流程分析结果。再按 F5 键，执行程序应该就完成了在执行程序中调用 DLL 的 delete()函数的效果。

下面实证。DM6-22(a)展示了调用 DLL 的 delete()函数的指令序列，其中 10031e2d 的 call 指令调用了 delete，而该指令中的“(1002b5f5h)”指示了 delete()函数的地址，后面就要用它修改 EIP。DM6-22(b)展示了调用执行程序 delete()函数的指令序列。单步执行到 0042d666 的 call 指令时，就展开“偷天大法”。在寄存器窗口中将当前 ESP 的值减 4，然后在内存窗体中输入 ESP 的值，回车，显示当前栈顶内容。修改栈顶内容为 DM6-22(b)的 call 指令后的指令地址，即 0042d66b。注意，该值内存顺序为小端机，所以是 6b d6 42 00，见图 6.13。此时已经模拟了 call 压栈返回地址的效果。然后，将 EIP 修改为刚才获得的 Dll 程序的 delete 函数入口地址 1002b5f5h。最后按 F5 键，恢复运行，程序如期正常结束，没有报任何错误。

DM6-22

(a)

```
  delete p;
10031e1d  mov   eax, dword ptr [p]
10031e20  mov   dword ptr [ebp-0e0h], eax
10031e26  mov   ecx, dword ptr [ebp-0e0h]
10031e2c  push  ecx
10031e2d  call  operator delete (1002b5f5h)
10031e32  add   esp, 4
```

(b)

```
  delete p;
0042d656  mov   eax, dword ptr [p]
0042d659  mov   dword ptr [ebp-0d4h], eax
0042d65f  mov   ecx, dword ptr [ebp-0d4h]
0042d665  push  ecx
0042d666  call  operator delete (42b5dch)
0042d66b  add   esp, 4
```

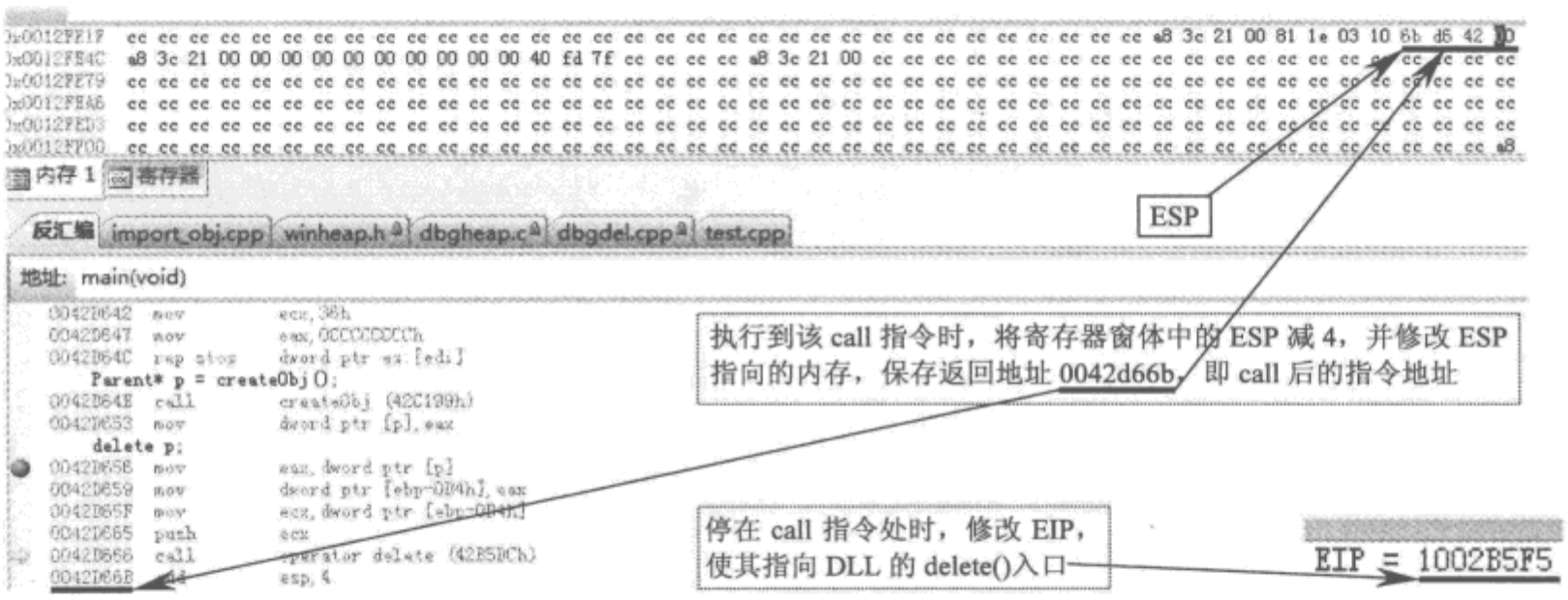


图 6.13

方案 1：在该实验证明了“让 DLL 的 delete 释放 DLL 的 new 分配的内存”这一思路正确后，我们就需要做出相关设计，让执行程序自然调用到 DLL 的 delete()。如果能获得 delete()函数的地址，函数地址就是函数指针，虚函数也是用函数指针。这两者能搭上线吗？让我们想想虚函数调用过程：从对象前 4 字节获取虚表地址，再从虚表中获取函数地址并调用。该函数地址是 DLL 中虚函数的地址，因此调虚函数总是调到 DLL 中的函数。如果在该虚函数中调用 delete()，就是通过“call 偏移量”指令调用 DLL 的 delete()函数，那么我们在 Parent 中定义一个虚函数叫 Destroy，它调用 delete(this)，而释放对象时调用它就可以。这时 DM6-17 的第 3 行变成“p->Destroy();”即可。工程见“code\第 6 章\import_obj-Destroy”。图 6.14 展示了 Destroy 调到的 delete()和 DLL 调到的 delete()为一个版本。不采用 Destroy 而将 Parent 的析构函数定义为虚函数可达到相同效果吗？见习题 3。

DLL 的 CreateObj()中调用 delete()

```
地址: createObj(void)
10002722  mov           dword ptr [ebp-0F4h], eax
10002728  jmp           createObj+54h (1002274Ah)
1000272A  mov           dword ptr [ebp-0F4h], 0
10002734  mov           eax, dword ptr [ebp-0F4h]
1000273A  mov           dword ptr [p], eax
    delete p;
10002740  mov           eax, dword ptr [p]
10002743  mov           dword ptr [ebp-0E0h], eax
10002749  mov           ecx, dword ptr [ebp-0E0h]
1000274F  push          ecx
10002750  call          operator delete (1002B5F5h)
10002755  add           esp, 4
```

EXE 通过 Destroy()中调用 delete()

```
地址: Child::Destroy(void)
10002BF3  mov           ecx, 38h
10002BF8  mov           eax, 0CCCCCCCCh
10002BFD  rep stos      dword ptr es:[edi]
10002BFF  pop           ecx
10002C00  mov           dword ptr [ebp-8], ecx
    delete this;
10002C03  mov           eax, dword ptr [this]
10002C06  mov           dword ptr [ebp-0D4h], eax
10002C0C  mov           ecx, dword ptr [ebp-0D4h]
10002C12  push          ecx
10002C13  call          operator delete (1002B5F5h)
10002C18  add           esp, 4
```

调用同一份 delete()代码，它是 DLL 的。这样，new()从 DLL 中分配内存，delete()也从 DLL 中释放

图 6.14

方案 2：其实还有另外一种解决办法，就是让分配、释放内存的代码只有一份。大家都从一个图书馆借书还书就对了，也就是说，不将 C 库以静态链接方式分别链入 DLL 和执行程序，而是让 C 库以动态链接库方式被使用，执行程序和 DLL 都访问这个 C 语言库 DLL 即可，见图 6.15（请对照图 6.9 的差异）。工程见“code\第 6 章\import_obj-c 库 dll”。实证，在 DLL 的 createObj 中添加一个 delete 语句，通过单步查看 DLL 中调用的 delete 函数的入口地址和 EXE 中调用的 delete 入口地址是否为同一地址。如是，则证明 EXE 和 DLL 是同一个 delete()函数，结果见图 6.16。其实 VS 已经帮大家考虑了，它生成的工程默认都是使用以动态链接库方式存在的 C 语言库函数（所以为了暴露错误，我们一开始要修改项目的运行库属性为“多线程调试(/MTd)”，这代表静态链接方式）。Visual C++ 6.0 的默认方式是静态链接 C 语言库，所以要特别注意。

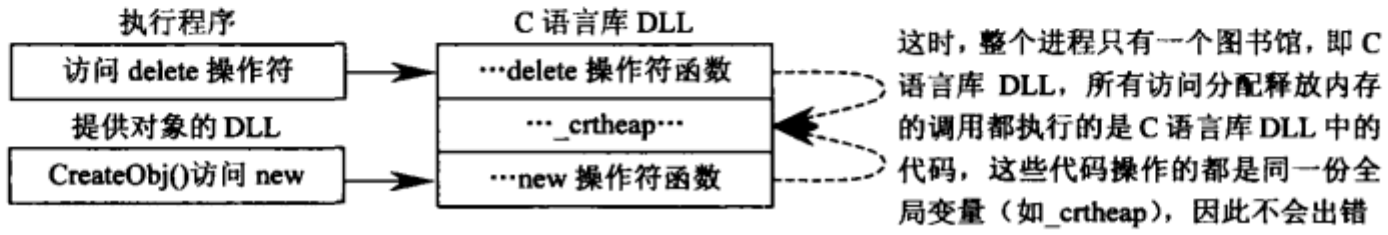


图 6.15

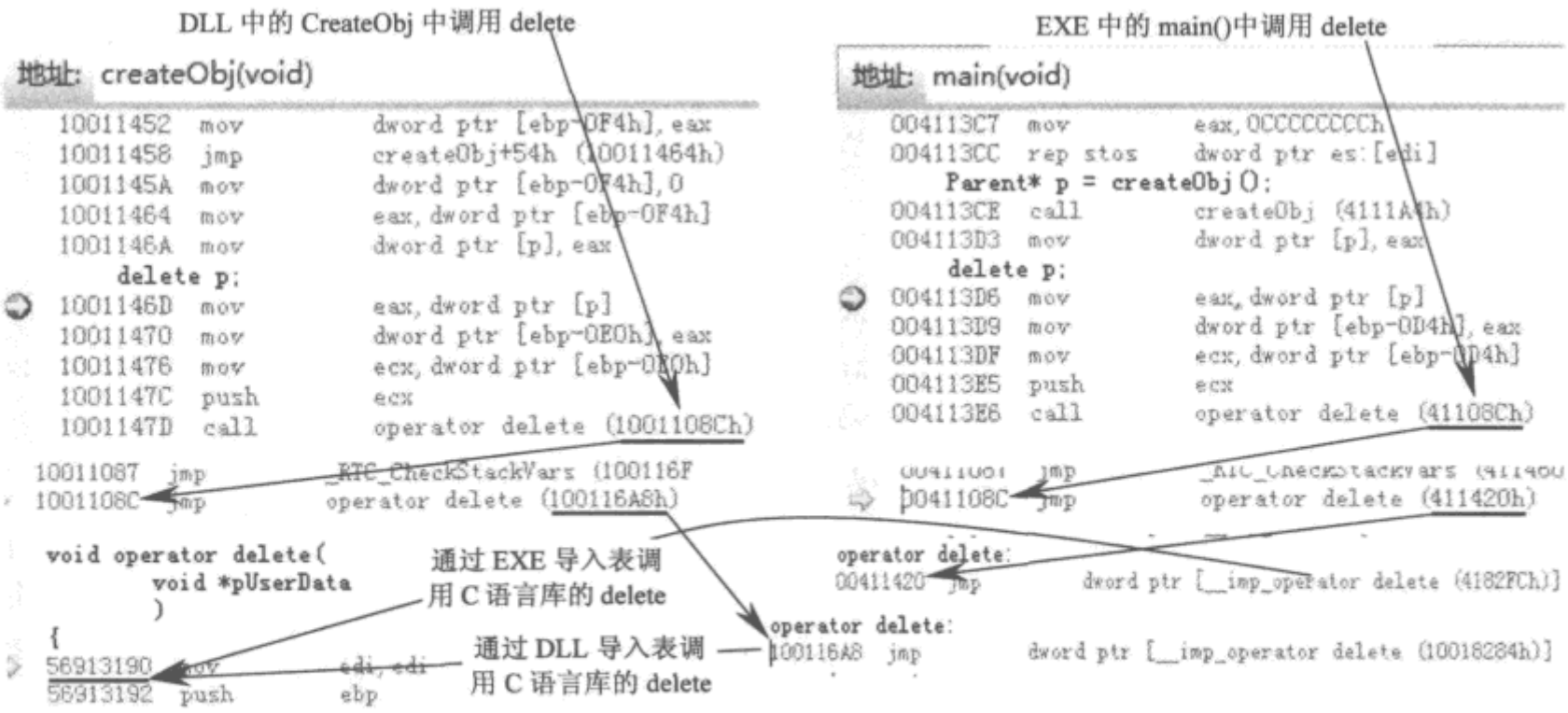


图 6.16

其实 VS 已经帮大家考虑了，它生成的工程默认都是使用以动态链接库方式存在的 C 语言库函数（为了暴露错误，首先要修改项目的运行库属性为“多线程调试(/MTd)”，这代表静态链接方式）。Visual C++ 6.0 的默认方式是静态链接 C 语言库，所以要特别注意。

通过虚函数调用 delete()的方案比动态链接 C 语言库更通用，因为作为第三方插件的 DLL 很难限制它必须以动态链接库方式使用 C 语言库。

习 题 6

1．现在需要实现一个软件系统，该系统可对文件进行统计分析。究竟对文件进行什么样的统计分析，最终用户自己也不是很清楚，目前只知道需要统计文件的行数。也就是说，本软件系统将会随时面临，增加文件统计新功能的难题。请结合本章内容，设计并实现这样一个面向文件统计功能的插件系统。（需要实现的插件包括：统计某个文件的行数，统计某个文件的字节数）

2．如果插件需要自己特殊的命令参数，应该怎么实现。

3．请问不采用 Destroy 释放对象，而将 Parent 的析构函数定义为虚函数可达到相同效果吗？提示：虚函数调用依赖的是函数指针，似乎将析构函数定义为虚函数就能通过函数指针调到 DLL 中子类的析构函数。但问题是出在析构函数上吗？

第7章

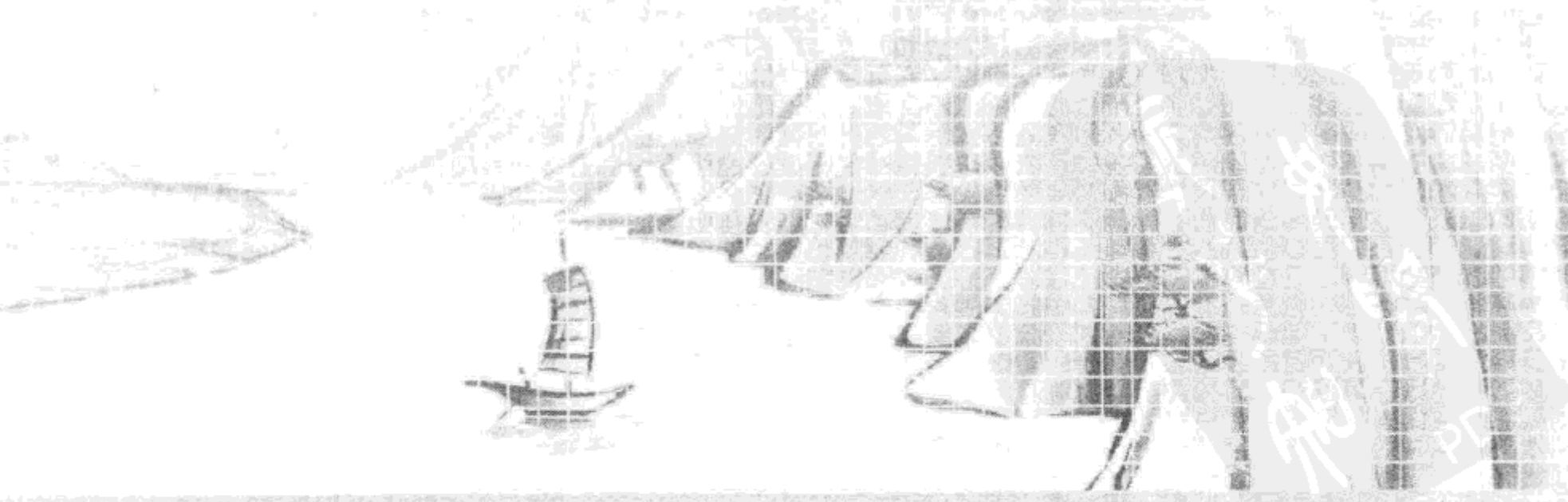
天堂的阶梯

需准备的知识 »

- 整章需要第1章的反汇编分析支持，特别是1.3节，最好完整阅读第4章。
- 7.4节需要4.1节和4.7节的知识。
- 7.4.5节需要1.3.3节和1.3.8节的知识。
- 7.6节需要5.2.5节和5.2.6节中的两个设计模式的知识。

为你提供的能力 »

- 了解并能运用源代码学习方法：调构学习法。
- 学习VCL的GUI框架，学会面向对象框架的分析和学习技巧。
- 了解Windows的消息驱动机制。
- 掌握自定位代码的运用，构建可扩展消息分发框架，并将VCL与MFC对比。
- 构建自己的微型GUI框架。
- 充分理解猜测—实证—构建方法。



在漫漫程序之旅中我们到底如何成长，如何从蹒跚学步走到遥远的天堂？除了实践、带着思考的不断实践，我们怎样将自己的手艺打磨得越来越精致，让作品越来越美丽？这里有两个阶梯：一个是从机器码到框架，一个是“构建—阅读—再构建”。

7.1 遥望天堂，那些美丽与简洁我向往

在学习编程时，看过一本书，它说“优秀的程序员都要阅读优秀的源代码”。当时笔者正热火朝天地学习着 Delphi，而其快速开发的便捷及支撑这些便捷的 VCL（Visual Component Library）类库深深地吸引了我。虽然 VCL 提供了源码，但我发现根本无法看懂。一般的执行程序从 main()开始阅读，而库从哪里开始？直到能力渐渐提升后，我终于开始了 VCL 库的探索。先来看在面向对象的支撑下，构建界面的简洁，见 DM7-1。

DM7-1

```
TMyForm = class(TForm)
    ...
    procedure ClickEventHandle(Sender: TObject);
end;
...
form := TMyForm.Create(nil);                    //创建窗体类
btn := TButton.Create(form);                    //创建按钮类
btn.Left := 10;                                 //设定按钮的左上角 x 坐标
btn.Top := 12;                                  //设定按钮的左上角 y 坐标
btn.Caption := 'hello';                         //设定按钮上显示的文字
btn.Parent := form;                             //将按钮放入父窗体 form
btn.OnClick := form.ClickEventHandle;           //将按钮响应函数设定为对象 form 的 ClickEventHandle()方法
form.Show();                                    //显示窗体，相应地，btn 也就显示了
```

学习了最原始的 GUI 编程后，我们会惊叹于这种设计的强大和简洁。库源码的学习很重要，特别在开源时代。很多重要的软件都是以库的方式提供，如 Hibernate、Spring、BerkeleyDB 等。曾经有人感叹，开源框架为我们做了 80%的工作，但我们的工作却越来越难做。究其原因，笔者认为，应用程序开发的个性导致必然对框架进行扩展，如加入插件、覆盖某个虚方法等，这让开发者需要直面框架的内部最核心处。开源软件文档的缺失又必然导致学习其内核无所依。（**其实即使像微软用巨大金钱支撑的帮助文档，学习起来也非常麻烦。**）

笔者一次做 SaaS 风格的软件，需要为每个注册的公司生成一套一模一样的数据库（表结构完全一样），用 Hibernate 存储时必须根据公司的不同存入对应的库。在已有的资料中没有涉及这方面的知识，而关键词难以界定，所以搜索引擎似乎也用处不大。最终通过分析源码，笔者找到了两种解决办法。这一过程让笔者同时经历了一次面向对象设计之美的洗礼。库的学习不仅能帮助我们解决困难任务，还能不断提高设计水平。套用一句老话，“熟读框架三百篇，不会设计也会编”。如何开始一个库的源码学习？请参考提示：基于调试的“构建—阅读—再构建”源代码学习法，简称调构学习法。

基于调试的“构建—阅读—再构建”源代码学习法

一、先学会使用库，越深入越好。
二、抛开库，自己用原始的方法完成相同的工作，体会其艰辛。对比第一步的便捷，理解库的需求。
三、从白手开始模仿构建一个类似的最基础功能，遇到的问题尽量想办法解决。
四、对比自己的做法和库，分析别人的长处。分析库源码时，采用后面的基于调试的源码学习技巧。

基于调试的“构建—阅读—再构建”源代码学习法（续）

五、按第四步学到的机制，重新构建自己的模拟库，并在该过程中尝试深入理解第四步无法理解的别人的做法。这非常重要，往往在进行到第四步时很多人就放弃了。但只有通过“构建”，我们才能懂得那些在第四步通过“看”无法理解的问题，并明悟他人的精妙。

7.2 从最基础开始吧，SDK 编写窗体程序

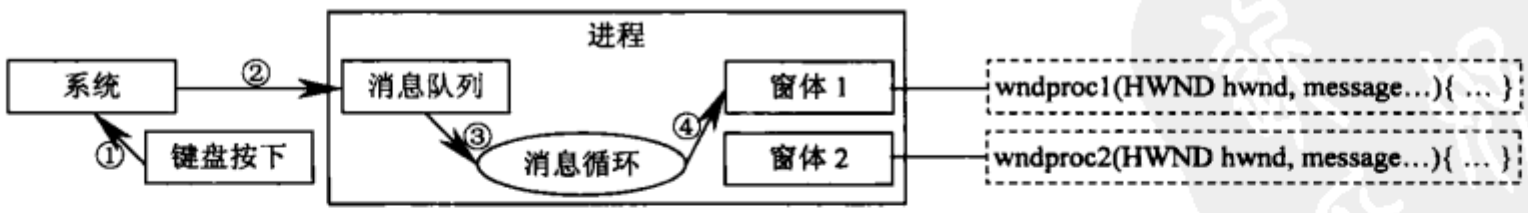
7.2.1 hello window 和基本原理

调构学习法第一步，我们省略，作为后面的对照，可参阅 DM7-1。现在进入第二步。希望深入学习 Windows 窗体编程的可参考文献 1。首先要了解 Windows 窗体工作的原理，DM7-2 是简化的典型窗体应用程序框架代码。

DM7-2

```
1   LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
2       switch (message) {
3           case WM_CREATE:
            ...
4           case WM_DESTROY:
5           PostQuitMessage (0) ;
6           return 0 ;
7       }
8       return DefWindowProc (hwnd, message, wParam, lParam) ;
9   }
10  int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
11      WNDCLASS   wndclass;
12      wndclass.lpfnWndProc = WndProc ;
        ...
13      wndclass.lpszClassName = "testwin";
14      RegisterClass (&wndclass);
15      hwnd = CreateWindow ("testwin", ...);
16      ShowWindow (hwnd, iCmdShow) ;
17      while (GetMessage (&msg, NULL, 0, 0)) {
18          TranslateMessage (&msg) ;
19          DispatchMessage (&msg) ;
20      }
21  }
```

窗体程序是通过处理消息运行的。操作系统将所有事件转换成一条条消息，发送给应用程序。比如，敲击键盘将生成键盘消息，其中包含哪个键被按下的信息。应用程序进入一个消息处理循环，见 DM7-2 中的第 17~19 行，不断接收并处理消息。其原理见图 7.1。



① 键盘中断被系统捕获；② 包装成消息投递；③ 获取消息；④ 调用相关窗体消息处理函数

图 7.1

图 7.1 中，第③步对应 DM7-2 的 17 行的 GetMessage 获取消息。第④步却存在两个问题：一、如何知道哪个窗体需要处理消息？二、如何知道窗体关联的消息处理函数是哪个？问题一需要分析

GetMessage 获取到的 msg 结构体的成员，见 DM7-3，其中第一个就是句柄 hwnd，它是整数，唯一地代表一个窗体。系统在分派消息时就知道应该将消息投递给哪个窗体处理。对于问题二，我们先来看消息、窗体和处理函数之间的关系。消息都是发往某个窗体的，窗体是一个抽象概念，代表界面上一个能接收消息的实体，如按钮、编辑框都是窗体。一个按钮被点击了，那么点击的消息就发往该按钮窗体。既然窗体负责接收并处理消息，就应该有一个与该窗体关联的消息处理函数，其原型为 DM7-2 的第 1 行。操作系统并不知道某个窗体的处理函数是什么，因此需要告知系统，即通过函数注册窗体句柄和处理函数的对应关系。系统用一个表记录了窗体句柄和相关处理函数的对应关系，在消息投递第④步时，只需要用句柄在表中查找到对应函数，并调用即可。

DM7-3

```
typedef struct tagMSG {
    HWND hwnd;          //处理该消息的窗体句柄
    UINT message;       //消息标识
    WPARAM wParam; //消息的参数
    LPARAM lParam;   //消息的参数
    DWORD time;
    POINT pt;
} MSG;
```

那么，登记窗体句柄和处理函数的对应关系有两种方法。

① 在窗体描述结构 WNDCLASS 中指出。窗体千差万别，如果让系统构造一个窗体，需要给系统一份窗体图纸。该图纸就是 WNDCLASS 结构体，其中有两个重要的成员变量，一个是 DM7-2 的第 13 行的 lpszClassName，标识这份图纸的名字，例中给定名字为“testwin”。后面构造窗体时，就用这个名字告知系统我们想用哪份图纸构造，第 15 行 CreateWindow 构造窗体，其第 1 个参数就是“testwin”，指出了图纸的名字。要用这份图纸自然需要将它注册给系统，所以第 14 行 RegisterClass 完成了该工作。WNDCLASS 结构体另一个成员是第 12 行的 lpfnWndProc，指出了窗体处理函数的地址，指示的函数就是第 1～9 行的函数 WndProc()。这种方法指出窗体的处理函数，意味着所有用名为“testwin”的 WNDCLASS 构造出的窗体都拥有同样的处理函数，如果构造了多个窗体，如何让函数区分到底当前在处理哪个窗体的消息？DM7-2 中，第 1 行窗体处理函数的第 1 个参数 hwnd 指出了被处理消息属于哪个窗体。只要在函数中用判断 hwnd 的不同，即可做出相应处理，这样一个窗体函数可被多个窗体共用，却又能各行其是。

② 第①种方法为某类型窗体指定相同处理函数，这种方法可为某个具体窗体对象指定处理函数。也就是说，可以用同一个 WNDCLASS 结构创建出的窗体拥有各自不同的处理函数。调用函数 SetWindowLong()即可，SetWindowLong(hwnd, GWL_WNDPROC, (LONG)wndProc)将消息处理函数指针 wndProc 设定给句柄 hwnd 指示的窗体。

消息处理函数怎样处理消息？从 DM7-2 第 1 行可知，它有 4 个参数。第 1 个已解释，是接收消息的窗体的句柄。第 2 个是无符号整数，代表消息类型，如宏 WM_PAINT 就是重绘消息。从 winuser.h 中可知，该宏声明为#define WM_PAINT 0x000F，即说明十进制数 15 代表重绘消息。一个消息可能包含相关参数，则由第 3、4 个参数指出，其类型为指针，但很多时候被直接当成整数。比如，鼠标左键按下消息 WM_LBUTTONDOWN 中，wParam 的整数值代表了此时其他键按下的状态，如 MK_SHIFT 代表键盘 Shift 键是否按下。读者可能会问，如果有 2 个甚至以上的参数需要表示怎么办？系统非常节约，比如还是 WM_LBUTTONDOWN 消息，需要包含此时鼠标的坐标 x 和 y，它用 lParam 的高 2 字节代表 y，低 2 字节代表 x。如果还有更多信息呢？因为 wParam 和 lParam 都声明为指针，那么就可以将任何结构体的指针作为 wParam 或 lParam 传递即可。这与第 5 章线程函数的参数只接收一个 void *类型却可传递任何信息是一样的。

DM7-2 的第 8 行消息处理函数为什么要调用 DefWindowProc？我们处理了自己感兴趣的消息，绝大多消息并没有处理。窗体要正常工作，这些消息实际需要处理，所以系统用 DefWindowProc()函数帮我们处理这些消息，只需在最后调用。（大家可注释该行代码，看看程序能否正常工作。）

现在来模拟消息产生并被处理的过程。单击窗体 1，这时操作系统内核将捕获中断。然后产生消息结构体变量，其 hwnd 成员为窗体 1 的句柄，其 message 值为 WM_LBUTTONDOWN，lParam 和 wParam 被填入了相关值。然后，内核将该消息变量放入产生该消息的进程消息队列，并通知进程可取消息。此时进程正阻塞在 DM7-2 的第 17 行 GetMessage()函数，因有消息可取，则恢复运行并返回，将消息队列中消息结构体的值复制到该函数传入的第 1 个参数指示的消息变量 msg 中。再调用 TranslateMessage()，最后调用 DispatchMessage()，将消息分派到窗体 1 的消息处理函数执行。这时，消息结构体中的成员 hwnd、message、lParam 和 wParam 分别对应传给消息处理函数的 4 个参数。消息处理函数返回后，进入下一轮消息循环。如果没有消息，那么进程就继续阻塞，在 GetMessage()等待消息。

程序何时退出？也是投递消息。当窗体被关闭时，窗体处理函数会收到 WM_DESTROY 消息，见 DM7-2 的第 4 行。这时再调用 PostQuitMessage()投递退出消息。GetMessage()收到退出消息，将返回，返回值为假，这时退出消息循环。

工程“code\第 7 章\win_hello”给出了最简单的窗体程序，DM7-2 来自该工程的简化（从 Petzold 的 Windows 程序设计简化而来）。在消息循环中，到底是哪个函数导致了系统调用窗体消息处理函数？从分析看，应该是 DispatchMessage()。在 win_hello 中，通过调试来验证。以调试方式运行程序，切换到开发环境，在窗体处理函数中随便设置断点，再切换回被调试程序。这时因为消息投递，程序将在断点处停止，如图 7.2 标注的所示，WinMain()函数的某行执行后，经过 user32.dll 中一些函数的调用，最终调用到最上面的消息处理函数 WndProc()。双击箭头所指的行，对应源代码正是 GetMessage()所在的行，即通过 GetMessage()直接调用了窗体函数分派消息。

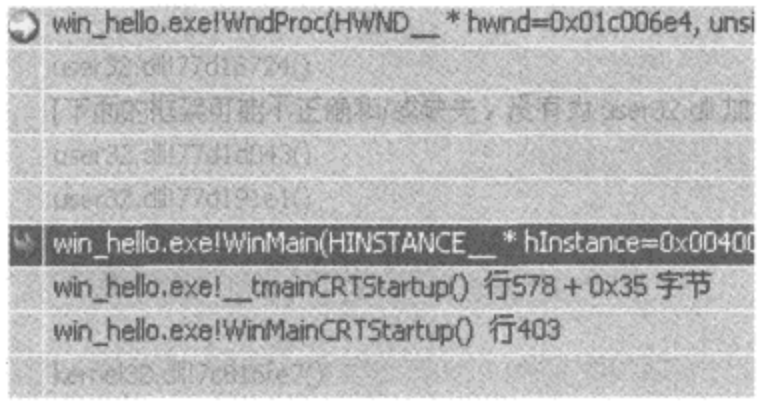


图 7.2

在 GetMessage()后设置断点，然后按 F5 键，继续执行程序，发现程序不断调用窗体函数，但没有从 GetMessage()返回，停在新加的断点处，可见有一批消息是直接通过 GetMessage()被派发给了处理函数，而 GetMessage()并不返回，因此避免了消息被 DispatchMessage()重复处理。然后在 DispatchMessage 语句处设置断点，当程序停在该断点后，按 F10 键执行，发现程序停在了消息处理函数的断点处。此时如前查看调用栈，确实是 DispatchMessage()调用了消息处理函数。简单总结就是，一部分消息在 GetMessage()中通过调用消息处理函数分派，GetMessage()不返回；一部分消息会被 GetMessage()取出，由 DispatchMessage()调用消息处理函数处理，与我们的估计有点不同，可见实践出真知。

前面多次提及，学习时一定要以构建者角度思考，“我来做将怎样做？”这样就会发现问题，学习深入。如果我们来构建消息处理机制，会怎样做呢？笔者当年这样思考后发现了一个很困惑的问题：既然系统知道消息属于哪个窗体句柄，又知道该句柄对应的消息处理函数地址，为什么不直接调用该函数？也就是说，不要 GetMessage 再 DispatchMessage 出去，也不用让我们来编写什么消息循环，系统直接调用相关消息处理函数，不可以吗？

要回答该问题，不妨做假想实验：程序像一个老师在讲课，操作系统是收发室的邮件接收者，收到老师的信件后，它应该做什么？方法一，直接“闯”进教室，不论老师在做什么，马上打断他，让他处理信件。方法二，将信件投递到教室门上的邮箱，老师定时查看邮件，并处理。方法二就是 Windows 消息循环的做法，方法一就是我们假想的“直接调用”方法。很明显，方法一将破坏课堂

现场。在程序中，可能的情况是，代码正在为全局链表 A 加入一个节点，刚刚处理到一半，突然消息来了，它被打断去执行另一段消息处理函数，该函数也要向全局链表 A 中加入一个节点，必然发生冲突，破坏 A 的数据结构。这一般发生在没有加锁的多线程编程中，但在单线程中采用方法一处理消息，也会导致该错误。另外，如果要以方法一的“中断方式”派发消息，还需保存被中断函数的现场并恢复现场（保存各种寄存器的值），这就更复杂、更低效了。

理解了消息处理原理，许多现象就能解释。比如，先前调试方式运行程序后，在消息处理函数中设置断点，再次切回到被调试程序，程序将停止在断点处，此时该程序的窗体无法显示，或者是白色一片。这种情况也常发生在我们所谓程序无响应时，窗体中一切元素都消失，只有一个白板。为什么？窗体需要重绘时，如从被遮盖的窗体变成看得见的窗体时，系统会发送 WM_PAINT 消息。窗体收到该消息，就会用相关代码处理重绘。而程序无响应，或强制设置断点迫使它停下，就意味着程序无法从当前消息处理函数中返回，自然无法进行下一次消息循环：用 GetMessage()获取 WM_PAINT 消息，因此无法执行处理 WM_PAINT 消息的重绘处理代码，只能看到一个“白板”。

如果我们运用构建者的角度思考这个问题，又会发现蹊跷：既然窗体上有哪些按钮、编辑框，操作系统都知道，为什么不让系统帮我们直接重绘？何必将这个工作以消息处理方式推给程序处理？（想想视频播放器就知道原因了，处于后台的播放器在放电影，回到前台后，系统帮助它恢复那些从前台变到后台时窗体上的旧画面是没有意义的。）

总之，学习新东西时，从构建者角度去思考，比被动接受知识能学到更多深层的东西。仅仅从构建者角度思考还不够，有些东西还要去构建。7.2.2 节将继续“调构学习法”的第二步——做一个复杂点的窗体应用。

7.2.2 来个复杂点的窗体程序

本节将在窗体中构建两个按钮，处理按钮鼠标左键弹起消息（即鼠标左键从按下状态到弹起时产生的消息），不同按钮响应该消息时弹出的消息框信息不同。程序还要处理窗体的鼠标左键按下消息，也弹出消息框。我们在窗体创建完毕后生成两个按钮，因此在窗体消息处理函数中处理 WM_CREATE 消息，见 DM7-4(a)的第 4 行。如何创建按钮？既然按钮依然是窗体的一种特例，那么就用 CreateWindow 创建。创建按钮的图纸，即 WNDCLASS 结构体早被系统预先注册好了，名字是“BUTTON”。因此，我们只需要在 CreateWindow()的第 1 个参数中传入“BUTTON”，见第 5 行；第 2 个参数是按钮上需显示的字串，这里传入“button1”；第 3 个参数是按钮的样式。按钮显示在哪个窗体上？倒数第 4 个参数就是该窗体句柄，我们将消息处理函数参数 hwnd（第 1 行）传递进去。因为此消息函数是窗体的处理函数，该函数被调用都是处理发往该窗体的消息，所以 hwnd 就是该窗体的句柄。按钮的位置坐标、宽和高呢？从第 4 个参数开始的后 4 个参数分别为按钮左上角点的 x、y、宽和高。

要让按钮能响应 WM_LBUTTONUP 消息，我们必须为按钮注册其消息处理函数。DM7-4(a)的第 6 行完成了该工作，CreateWindow 构造按钮后返回按钮的句柄为 b1，这时作为 SetWindowLong 的第 1 个参数传入。被注册消息处理函数的入口地址为第 3 个参数，此时传入的是函数 WndProcBtn()的指针，该函数见 DM7-4(b)。细心的读者会问，DM7-2 消息处理函数在处理了我们关心的消息后，在第 8 行调用了 DefWindowProc()处理消息，那么按钮的默认处理函数是什么，也是 DefWindowProc()吗？按钮点击后会凹下，然后弹起，这是收到消息后对按钮的特殊处理，明显 DefWindowProc()不会做这些事，它只是普通窗体的处理函数。如何拿到按钮本身的处理函数？SetWindowLong()的返

回值就是调用该函数之前原窗体处理函数地址，也就是按钮本身的默认消息处理函数。DM7-4(a)的第 6 行 oldBtnProc()保存了按钮本身的消息处理函数。按钮的新消息处理函数 WndProcBtn()通过 CallWindowProc 调用了 oldBtnProc 处理消息，见 DM7-4(b)的第 1 行。

DM7-4

(a)

```
1   LRESULT CALLBACK WndProc (HWND hwnd,
2   UINT message, WPARAM wParam, LPARAM lParam){
3       switch (message) {
4       case WM_CREATE :
5           b1 = CreateWindowA( "BUTTON", "button1", WS_CHILD | WS_VISIBLE |
                    BS_PUSHBUTTON, 0, 10, 60, 40, hwnd, 0, ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;
6           oldBtnProc = (WNDPROC)SetWindowLong(b1, GWL_WNDPROC, (LONG)WndProcBtn);
7           b2 = CreateWindowA( "BUTTON", "button2", WS_CHILD | WS_VISIBLE |
                    BS_PUSHBUTTON, 0, 70, 60, 40, hwnd, 0, ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;
8           SetWindowLong(b2, GWL_WNDPROC, (LONG)WndProcBtn);
9           return 0 ;
10      case WM_LBUTTONDOWN:
11          MessageBoxA(NULL, "form left button",NULL, NULL);   break;
12      ...
```

(b)

```
    LRESULT CALLBACK WndProcBtn(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
1       LRESULT result = CallWindowProcA(oldBtnProc, hwnd, message, wParam, lParam);
2       if (message == WM_LBUTTONUP){
3           if (b1 == hwnd)
4               MessageBoxA(NULL, "button1 left up",NULL, NULL);
5           else if (b2 == hwnd)
6               MessageBoxA(NULL, "button2 left up",NULL, NULL);
7       }
8       return result;
9   }
```

从构建者角度思考，我们又发现了问题，为什么 DM7-4(b)第 1 行不直接用函数指针 oldBtnProc 调用函数，即 oldBtnProc(hwnd, message, wParam, lParam)？为什么必须通过向 CallWindowProc 传入被调函数 oldBtnProc 指针来完成对 oldBtnProc 的调用？很明显，CallWindowProc 似乎不过做了以下伪代码的工作：

```
CallWindowProc(WNDPROC proc, HWND hwnd, UINT message, WPARAM wParam,
                                                    LPARAM lParam) {
    proc(hwnd, message, wParam, lParam);
}
```

（费解！事实是直接调用并没错。）这个问题在 7.5 节我们还会相遇。查看 DM7-4(b)的逻辑，第 2 行判断消息是否为左键弹起消息，如果是，则第 3 行和第 5 行用句柄判断是哪个按钮的消息，从而调用 MessageBox 显示时打印不同的消息。完整工程见“code\第 7 章\button”。

下面再将程序变化一下，在窗体中添加一个文本框，按钮单击时不弹出对话框，而是将文字写入文本框中，工程见“code\第 7 章\button_edit”。只需在 DM7-4(a)的 WM_CREATE 处理代码中添加编辑框的构造代码，依然用 CreateWindow()构造编辑框，WNDCLASS 的名字是“EDIT”：

```
edit = CreateWindowA("EDIT", "edit1", WS_CHILD | WS_VISIBLE | WS_BORDER ...)
```

向编辑框写入字符串需要向编辑框发送编辑消息 WM_SETTEXT，发送函数为 SendMessage()，代码如下：

```
SendMessage(edit, WM_SETTEXT, 0, (LPARAM)"b1 up")
```

第 1 个参数是接收消息的窗体句柄，这里是编辑框的句柄。第 2 个参数是消息号。最后 2 个参数分别为 wParam 和 lParam。对于 WM_SETTEXT 消息，lParam 为一指针，指向将要写入编辑框的字串地址。用该 SendMessage 语句替换 DM7-4(b)中第 4 行和第 6 行的代码即可。

7.3 构建我的 GUI 组件（1）：简单组件

重复即罪恶

在 7.2.2 节的编码中，我们挺辛苦，不断复制代码，然后粘贴修改。“重复就是罪恶”的断言缠绕着我们。看看 DM7-4(a)第 5 和 7 行构造按钮的复杂调用，即使拷贝修改也很容易出错。再对比 DM7-1 的代码，那种简洁就是我们要的目标。细细看来，有些代码也很不符合面向对象思想，如 DM7-4(b)，第 3 和 5 行需要区别句柄做出不同动作。这违反了自己操作自己数据的原则。我们的第一个目标是封装窗体和按钮，用面向对象改造之前工程 code\第 7 章\button。

首先解决 DM7-4(a)中的 5~6 行的问题，它负责构造按钮。自然地，根据前面屡次体验的“构造分配资源，析构释放资源原则”，我们将这两行代码移到即将创建的 Button 类构造函数中，而其析构中将按钮资源释放。好了，这时发现我们竟然不知道怎么释放创建的窗体资源。之前就知道不断 CreateWindow，完全不管释放 window 的事儿。有了 VCL 框架，我们就不用老问百度了，调试框架就很容易知晓答案。

基于调试的代码阅读技巧 1：未知功能实现机制的学习

有些功能我们不知道如何实现，但框架实现了，那么我们编写代码调用框架相关功能，然后用跟踪进入（step into）的方式跟踪进入分析即可。对于特殊的非直接调用的功能，还应该进行特殊处理。比如，发送消息后还要在消息处理函数中进行相关处理的情况，就应该在处理函数中设置断点，通过条件断点（即某个参数或变量满足某个条件才停止的断点）捕获消息处理。

例如，Delphi 调试 VCL 框架需要做特殊设置，否则无法跟踪进 VCL 代码。选择“Project | Options”菜单命令，在弹出的对话框的 Compiler 标签中勾选 Debugging 项。

为什么要用跟踪的方式？（**记得当年看不太懂源代码时，笔者的好友告诉自己“看不懂就调试”。他没有解释，后来发现这确实非常有效。**）记得一次修改 snort 源码，从下载代码到找出修改办法只用了一个早上。什么原因？笔者觉得这与程序的 2-8 原则有关。我们往往关心的是 20%的常用功能的代码逻辑，可要付 80%的“非”常用代码。如果看到 if 语句就跟踪进去，看到循环就慢慢分析，看到函数就跟进去分析，那么工作量将十分巨大。而在调试状态下，即使用 step over 方式（在 Visual C++中按 F10 键）将代码跟踪一遍，也只会面对那 20%的代码，工作将显著减轻。如果我们再分析一下，好的源代码基本通过函数名就能猜出其意图，就能再次滤除部分可能无关代码。调试几次，就能很容易发现关注点。再加上调试状态下能观察变量各种状态及调用栈等，甚至可以动用内存查看。遇到无法理解的东西，通过调试查看分析，也基本能得到答案。所以，大家在粗略浏览代码后，应该立刻动手调试。

先用 VCL 构造代码来创建编辑框，然后释放，观察对象的析构函数中是否存在编辑框控件销毁的代码。这里有一点推测，毕竟在析构中销毁编辑框是符合习惯和规律的。代码见 DM7-5(a)，第 3 行析构对象 E。在第 3 行处跟踪进 Destroy 函数。有 20 行代码，涉及 9 个函数的调用，为了快速捕捉到关键点，我们又要猜测了哦。看看哪个函数名像销毁编辑框的函数。既然编辑框用 CreateWindow 构造，返回的标识叫窗体句柄（window handle），那么 DM7-5(b)中的 DestroyWindowHandle 不是很像么？跟踪进该函数，看到代码如 DM7-5(c)，其中 DestroyWindow 函数应该就是销毁窗体句柄的函数，在微软帮助中一查，果然如此。

学习计算机英语的小窍门

不用专门去看专业英语书。每次读中文计算机资料时，想想看到的专业词汇是否自己知道对应的英文，不知道查查即可。比如，变量的英文是 variable。

计算机英语的关键是专业词汇。

DM7-5

(a)

```
1    E := TEdit.Create(nil);
2    E.Parent := Self;
3    E.Destroy;
```

(b)

```
  Destroying;
  ...
  if Parent <> nil   then RemoveFocus(True);
  if FHandle <> 0   then DestroyWindowHandle;
  ...
```

(c)

```
Include(FControlState, csDestroyingHandle);
try
  if not Windows.DestroyWindow(FHandle) then
      RaiseLastOSError;
  finally
    Exclude(FControlState, csDestroyingHandle);
end;
FHandle := 0;
```

通过调试框架，我们知道了控件销毁函数（其实设定编辑框中文字的方法也是通过调试 VCL 框架快速了解的），而且看到 VCL 也是在析构函数中销毁控件句柄的。因此，按钮类的构造和析构函数诞生了，m_handle 为私有变量，见 DM7-6。

接下来处理消息响应的问题。如同第 5 章线程函数一样，消息处理函数不该是静态成员函数，否则在消息处理函数中就没有了对象的封装能力（没有 this 指针，无法操作自己的数据）。从 DM7-4(b)可见，消息处理函数通过判断语句检查句柄来分辨消息属于哪个窗体。该做法在产生新按钮时，必然要增加判断语句，这明显不能组件化，是典型的非面向对象风格。不同按钮对象在响应同一事件（消息）时执行的逻辑不同，但消息处理函数是操作系统定义的不传递 this 的函数。这很像第 5 章线程中 5.2.6 节的情况：线程的回调函数也不传递 this，而线程结束事件的处理逻辑对不同线程可能不同。在第 5 章我们运用了策略者模式，向线程对象注册了一个 listener，在回调函数结束前调用了 listener 的虚函数 NotifyThreadFinished，见 DM5-8。在这里我们可模仿定义一个 ButtonListener 注册到按钮对象中，其虚方法 onLeftButtonUp 代表处理按钮的鼠标左键弹起消息。将消息处理函数申明为 static 以便通过 SetWindowLong 设定给新构造的按钮。并在窗体处理函数中遇到鼠标左键弹起消息时，调用 listener 的虚函数 onLeftButtonUp。有了这个构思，我们需要修改构造函数，添加 SetWindowLong 注册消息处理函数，并在 m_oldWndProc 中保留按钮之前的消息处理函数供新消息处理函数调用，见 DM7-7(a)。在 Button 类中定义 static 的消息处理函数 WndProc，见 DM7-7(b)。并实现消息处理函数 WndProc，调用注册的 Listener，见 DM7-7(c)。可我们在 DM7-7(c)的第 1 和 3 行处不得不停下来。注意其中中文字符“按钮对象指针”。在这个回调函数（static 函数）中，我们无法获取到按钮对象的地址！

DM7-6

(a)

```
Button::Button(int x, int y, int width, int height, char * caption, HWND parentHandle) {
    m_handle = CreateWindowA("BUTTON", caption,
                                WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                                x, y, width, height,
                                parentHandle, 0, applicationHandle, NULL
    );
}
```

(b)

```
Button::~Button(void) {
  DestroyWindow(m_handle);
}
```

DM7-7

(a)

```
Button::Button(int x, int y, int width, int height, char * caption, HWND parentHandle) {
    m_handle = CreateWindowA("BUTTON", …);
    m_oldWndProc = (WNDPROC)SetWindowLong(m_handle, GWL_WNDPROC, (LONG)WndProc);
    m_listener = NULL;
}
```

(b)

```
class Button{
  private:
    HWND m_handle;
    WNDPROC m_oldWndProc;
    ButtonListener * m_listener;
    static LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
};
```

(c)

```
    LRESULT CALLBACK Button::WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam){
1       LRESULT result = CallWindowProcA(按钮对象指针->m_oldWndProc, hwnd, message, wParam, lParam);
2       if (message == WM_LBUTTONUP)
3           按钮对象指针->m_listener->onLeftButtonUp…;
4       return result;
5   }
```

该麻烦与线程如出一辙，可是线程利用了“线程函数的参数是我们传入的”这一条件，将对象指针直接从线程函数传入了，只需进行指针强制转换即可获得对象地址，见 DM5-7 的第 2～3 行。而消息回调函数无法照搬，通过消息处理函数获取到的是按钮对象句柄，但它不是对象地址。怎样从句柄获取到对应对象地址？一个直观的想法就是，构造一个全局 hash 表，其键对应句柄，其值对应对象指针，在消息函数中就可用句柄从 hash 表中查到对象指针了。那么，在构造函数中向 hash 表添加“句柄-对象指针”键值，在销毁对象时即在析构函数中移除。hash 表采用 STL 中的 std::map 即可。考虑到多线程同时操作该表的可能，我们用临界区保护了对该表的存取。DM7-8 展示了注册、注销和由句柄获取对象指针几个函数的代码。以此改造构造和析构函数，构造函数见 DM7-9(a)，析构函数见 DM7-9(b)。

DM7-8

(a)

```
void initApp(HINSTANCE instance) {
    //初始化相关全局变量，在 WinMain 中被调用
    applicationHandle = instance;
    InitializeCriticalSection(&handlesLock);
}
void addWindowHandle(HWND handle, void * objPtr) {
    EnterCriticalSection(&handlesLock);    //加锁
    windowHandles[handle] = objPtr;  //注册"句柄-对象"
    LeaveCriticalSection(&handlesLock);  //解锁
}
```

(b)

```
void delWindowHandle(HWND handle){
    EnterCriticalSection(&handlesLock);//加锁
    windowHandles.erase(handle);//注销句柄
    LeaveCriticalSection(&handlesLock);//解锁
}
void * getObject(HWND handle){//用句柄获取对象指针
    EnterCriticalSection(&handlesLock); //加锁
    void * ret = windowHandles[handle];//用句柄获取对象指针
    LeaveCriticalSection(&handlesLock);//解锁
    return ret;
}
```

DM7-9

(a)

```
m_handle = CreateWindowA("BUTTON", …);
addWindowHandle(m_handle, this);  //注册句柄，对象指针
m_oldWndProc = (WNDPROC)SetWindowLong…;
…
```

(b)

```
delWindowHandle(m_handle);//注销句柄
if (m_listener != NULL) delete m_listener;
…
DestroyWindow(m_handle);
```

DM7-7(c)将变化为 DM7-9(c)。因为第 7 行调用 listener 的虚方法，所以不同的消息响应逻辑只需注册不同的 listener 即可，具有很好的扩展性。如果我们想提供更多的消息处理能力，怎么办？只需在 ButtonListener 中定义更多的类似 onLeftButtonUp()的虚函数，如 onKeyDown()，并在第 5～8 行之间包含的条件处理语句中如第 6～7 行那样添加更多的消息判断及处理语句，见 DM7-10。

DM7-9(c)

```
1   LRESULT CALLBACK Button::WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
2       Button * self = (Button *)getObject(hwnd);  //用句柄从句柄-对象指针表获取到对应的对象指针，并强转为 Button*供后使用
3       if (self == NULL)  return 0;
4       LRESULT result = CallWindowProcA(self->m_oldWndProc, hwnd, message, wParam, lParam);//调用消息处理老函数
5       if (self->m_listener != NULL) {              //如果注册了 listener，则判断是否收到相关消息
6           if (message == WM_LBUTTONUP)             //如果消息是鼠标左键弹起消息，则调用 listener 的 onLeftButtonUp 通知
7               self->m_listener->onLeftButtonUp(self);
8       }
9       return result;
10  }
```

DM7-10

(a)

```
class ButtonListener {
  public:
    virtual void onKeyDown(Button * button, int key) = 0;
    virtual void onLeftButtonUp(Button * button) = 0;
};
```

(b)

```
…Button::WndProc(HWND hwnd, …) {
    …
    if (message == WM_LBUTTONUP)
        self->m_listener->onLeftButtonUp(self);
    else if (message == WM_KEYDOWN)
        self->m_listener->onKeyDown(self,…);
    …
}
```

觉得这个设计就像打补丁一样慢慢改出来的，可这就是学习面向对象设计的真实过程。甚至在你熟悉这些手法后，依然不能完全杜绝“打补丁”。为了设定 Button 上显示的字符串，我们定义了 setCaption()函数，它向按钮发送 WM_SETTEXT 即可。（限于篇幅就不给出源代码。）

在 Button 类的基础上，比较容易翻版出 Form 类，封装了一般窗体类的逻辑，主要在消息处理函数上有些不同，因为用户可直接单击窗体右上角的“×”关闭窗体，这导致发送 WM_DESTROY 消息并销毁窗体，所以我们应该随之释放窗体对象。那么，在消息处理函数中加入 WM_DESTROY 消息的处理逻辑，如 DM7-11(a)粗体所示。

程序可能创建很多窗体，但当第一个创建的窗体，一般我们称为主窗体关闭后，就意味着程序退出。因此在主窗体退出时应该调用 PostQuitMessage()函数。修改析构函数，见 DM7-11(b)。其中，m_isMainForm 是私有成员变量，其值在构造函数中作为参数传入并设定给它。

DM7-11

(a)

```
LRESULT CALLBACK Form::WndProc(HWND hwnd, …) {
    …
    switch (message){
        case WM_DESTROY:  delete self;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, …) ;
}
```

(b)

```
Form::~Form() {
    delWindowHandle(m_handle);
    DestroyWindow(m_handle);
    if (m_isMainForm)
        PostQuitMessage(0);
}
```

下面用 Form 和 Button 类构建程序。程序将先构造一个主窗体，包含一个按钮。按钮注册的 listener 在鼠标左键弹起后将弹出消息框，然后生成另一个窗体。该窗体也包含一个按钮，单击后将改变按钮自己的显示文字。如何生成新窗体？有两种方法：一是 DM7-12(a)的第 4～8 行，构造 Form 类对象 form，然后构造相关组件类（如 Button），并将它们的父窗体设定为 form，再添加相关的 listener

对象（第 7～8 行）；方法二是从 Form 继承生成子类 MyForm，在其构造函数中生成相关组件并注册相应 listener 对象，见 DM7-12(b)，工程见“code\第 7 章\button_obj_simple”。

DM7-12

(a)

```
1    int WINAPI WinMain (HINSTANCE hInstance, ...) {
2        initApp(hInstance);
3        MSG msg;
4        Form * form = new Form(true, TEXT("hello"));                    //构造主窗体
5        HWND parent = form->getHandle();
6        Button * btn = new Button(10, 20, 100, 30, "button1", parent);
7        MyListener * bListener = new MyListener();
8        btn->setListener(bListener);
9        while (GetMessage (&msg, NULL, 0, 0)) {
10           TranslateMessage (&msg) ;
11           DispatchMessage (&msg) ;
12       }
13       return msg.wParam ;
14   }
```

(b)

```
class MyListener2 : public ButtonListener {
  public:
    virtual void onLeftButtonUp(Button * button){
        button->setCaption("change");                                    //修改按钮文字
    }
};
class MyForm : public Form {
  private:
    Button * m_button;
  public:
    ~MyForm() {
        delete m_button;                                                 //删除按钮对象
    }
    MyForm(): Form(false, TEXT("test")) {                                //非主窗体
        m_button = new Button(0, 0, 100, 30, "hello", this->getHandle());  //生成按钮
        m_button->setListener(new MyListener2());
    }
};
```

7.4 构建我的 GUI 组件（2）：天堂的机器码跳板

7.4.1 调试，我要看清你

7.3 节的代码与 7.2 节相比，令人满意。虽然还有些不太完善的地方，如 Form 和 Button 类相似的代码太多了。（**如果将来封装编辑框类，可以想象将又是复制、修改。现在这个问题暂时放放。**）“调构学习法”已经完成了三步，我们迫不及待要进入第四步，看看 VCL 类库怎样处理我们实现的那些功能，学习它的长处。

比较纠结的是那个全局变量的句柄-对象指针表。全局变量总让人不舒服，因为它违反了面向对象的基本原则，它也有明显弱点：消息处理函数是非常频繁被调用的一个函数，每次都要去查表，似乎效率不理想。另外，为了多线程可使用，我们加入了临界区控制，这将陷入内核，无疑更雪上加霜。更有甚者，如果这里处理不好，如发生某些异常，导致代码没有执行 LeaveCriticalSection，还会造成死锁。那让我们看看 VCL 的实现吧。

分析源码不能总一行一行“死”调试，这样效率太低。下面的提示给出了快速切入主题的小建议。该方法非常有效，下面就用它来分析 VCL 如何处理上面的问题。

基于调试的代码阅读技巧 2：快速定位和分析框架的扩展机制

不论是面向对象还是 C 语言，本质上，回调函数是它们扩展机制的基础。为了分析框架中我们关注的扩展机制，从相关回调函数入手则十分高效。首先，通过例子或说明了解如何使用这种扩展机制，哪里是相关的回调函数，并如何使用。比如，在 VCL 中，我们很容易知道响应按钮消息就是在按钮对象上挂接按钮事件的处理函数，该函数就是一个回调函数。然后在该函数中设置断点，运行程序，让程序在断点停止。接着调用栈（见 5.4.1 节），查看框架是通过哪些函数调用到断点所在的函数。这样就能抓出框架调用回调或虚函数的主线。最后，我们分析并根据情况反复调试这条调用线路即可。

第一步，构建涉及消息处理的代码。按钮的点击事件必然涉及消息处理，只需拖拽一个按钮，双击它，挂接一个 click 事件。之后随便在该事件函数中写点代码，设置断点。运行程序后，单击该按钮，程序在断点处停止。这时，按快捷键 Ctrl+Atl+S，激活调用栈界面，如图 7.3 所示。这个调用序列比较深，到底从哪里查看？从函数的名字看，多数函数都如 TControl.Click 这样的“类名.方法名”，说明它们是类的函数，所有非静态成员函数都不能直接作为消息处理函数（要隐蔽传递 this 指针），所以可以先不看有类名的函数（当然，如果它们是静态函数，则可作为回调）。这样只有 DoControlMsg 和 StdWndProc。不论是参数个数（消息处理函数有 4 个参数，只有 StdWndProc 符合）、函数所处的位置（越下面越可能是系统回调的处理函数），还是函数名字，我们第一选择应该 StdWndProc。双击 tdWndProc 条目，将转到图 7.4 所示的 StdWndProc()函数。从该函数的参数表可知其就是消息处理函数，关键字 stdcall 也符合消息处理函数的调用惯例。基本能推定这就是消息处理函数了。图 7.4 中深色标定的行（即 ADD　ESP, 12）就是当前执行序列返回到该函数时将执行的语句。明显，上面的 call 指令调用了图 7.3 中 StdWndProc 上面的函数，即 TWinControl.MainWndProc()。如果该函数是非静态函数，那么 StdWndProc 必须将对象指针传给它。可 StdWndProc 怎样取得对象地址？它的参数表可不传递这个，而图 7.4 似乎也与句柄-对象指针表无关。**（我们正一步步接近关注点！）**

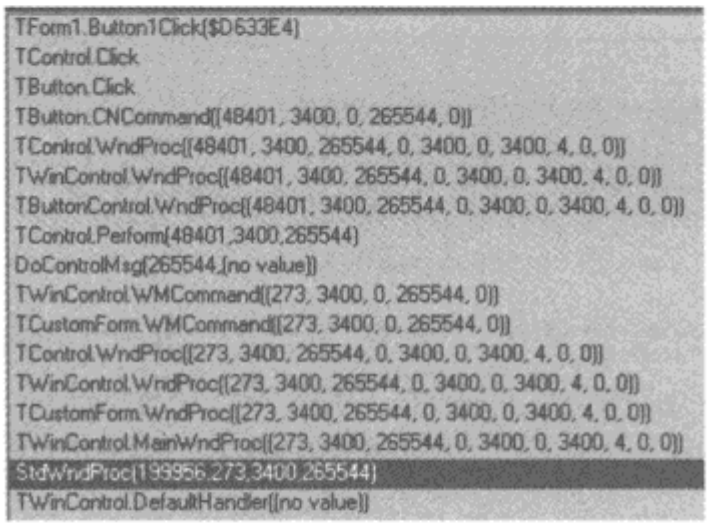


图 7.3

```
function StdWndProc(Window: HWND; Message, WParam: Longint;
  LParam: Longint): Longint; stdcall; assembler;
asm
        XOR     EAX,EAX
        PUSH    EAX
        PUSH    LParam
        PUSH    WParam
        PUSH    Message
        MOV     EDX,ESP
        MOV     EAX,[ECX].Longint[4]
        CALL    [ECX].Pointer
        ADD     ESP,12
        POP     EAX
end;
```

图 7.4

双击图 7.3 中的 TWinControl.MainWndProc 条目，查看该函数的定义，它确实不是静态成员函数。**（这里就是问题解决之匙了。）**图 7.4 中的 4 条 PUSH 指令看来与参数传递相关，this 指针从哪里传递的？在 CALL 指令前只有 MOV　EDX, ESP 和 MOV　EAX, [ECX].Longint[4]两条指令是可能的候选。但 MOV　EDX, ESP 肯定不是，因为它之前是一条 PUSH Message 指令，执行该指令后 ESP 就指向栈顶存放 MESSAGE 值的内存。所以接着执行 MOV　EDX, ESP，使得 EDX 也指向栈上存放 Message 的内存，这块内存并非我们的对象，而是消息处理函数临时生成的栈上的变量。它

似乎是将该变量的地址传递给 TWinControl.MainWndProc 函数。进一步查看该函数的定义如下：

```
//void MainWndProc(TMessage* Message)
procedure MainWndProc(var Message:TMessage);
```

DM7-13

```
struct TMessage{
    unsigned long Msg;
    unsigned long WParam;
    unsigned long LParam;
    unsigned long Result;
}
```

参数中的 var 相当于 C++的引用&，就是传递结构体 Message 的地址。如果再查看 TMessage 定义，它的 C 语言等价定义见 DM7-13。容易分析出图 7.4 中的 PUSH 指令就是在构造结构体 TMessage 变量的内存，并赋值，然后通过 MOV　EDX, ESP 指令将该变量地址传给 EDX。图 7.5 给出了分析。当然，这一分析我们马上用调试证明。

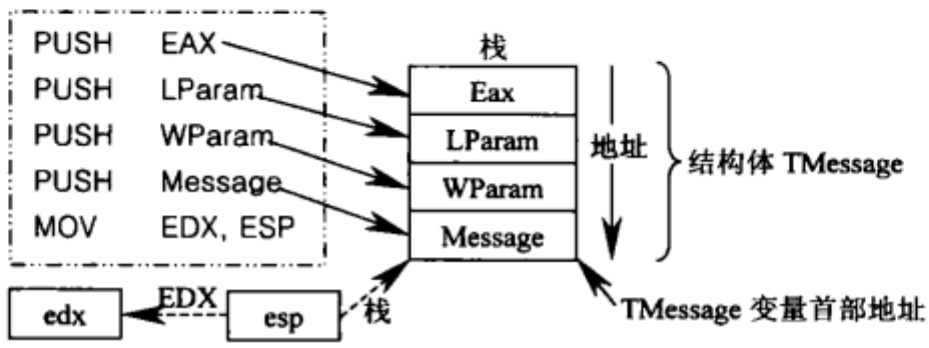


图 7.5

图 7.4 中标注的 ADD 指令是当调用序列返回到该函数后执行的指令。为了验证“EDX 传递了 TMessage 结构体变量地址”的结论，必须在之前的 CALL 指令处设置断点，然后等待下一次进入消息循环。按 F9 键，执行程序，再次停在 CALL 指令后，按 F7 键，跟踪进函数，发现它果然调用了 TWinControl.MainWndProc()函数。该函数的反汇编如图 7.6 所示，最后一行的 MOV 指令从 EDX 中取出了 TMessage 结构体的地址。此时，在查看窗体（按组合键 Ctrl+Alt+W 激活）中查看 Pointer(Self)，相当于在 C++中查看 this 指针。发现 this 指针与 EDX 的值并不相同，如图 7.7 所示（左部是查看窗体，右部是寄存器窗体），this 指针是 0xD63888，EDX 是 0x0012E818。这进一步说明 EDX 确实不是传递 this 指针。

```
TWinControl.MainWndProc:
0043D5F8 55              push ebp
0043D5F9 8BEC            mov ebp,esp
0043D5FB 83C4F8          add esp,-$08
0043D5FE 53              push ebx
0043D5FF 56              push esi
0043D600 57              push edi
0043D601 8955F8          mov [ebp-$08],edx
```

图 7.6

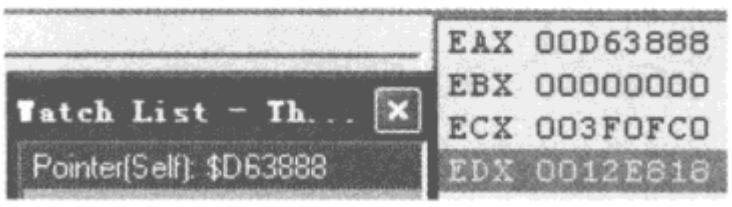


图 7.7

既然图 7.4 中的 MOV　EDX, ESP 不是传递对象的地址，就只下那条 MOV　EAX, [ECX].LONGINT[4]了。该指令并非标准汇编，而是 Delphi 内嵌汇编语法。虽然不能全部理解，但至少知道是对 EAX 赋值。也就是说，EAX 应该用来传递 this 指针。此时查看图 7.7，EAX 的值正好是查看窗体中 this 指针的值。这不是巧合，暗示 Delphi 就是用 EAX 传递 this 指针。如同 4.1.6 节中证明 Visual C++用 ECX 传递 this 一样，可模仿它编写一个 Delphi 的类，并反汇编跟踪分析，容易证明该结论，此处不再赘述。

还是回到我们关注的问题：在消息处理函数中如何获得 this 指针？现在的问题是搞懂指令 MOV EAX, [ECX].LONGINT[4]。不用查资料学习 Delphi 反汇编语法，只需在反汇编窗体中查看即可。我们发现，这条指令其实是 MOV　EAX, [ECX+4]，也就是说，从 ECX 指向的内存偏移 4 字节，从中取出的值就是 this 指针。再次回顾图 7.4，消息处理函数 StdWndProc()中并没有对 ECX 赋值的指令，说明 ECX 应该是调用该函数时作为参数被调用者设定的。但 StdWndProc()为窗体函数，是在

消息循环中调用 DispatchMessage()或 GetMessage()时被操作系统回调的。操作系统不可能知道我们所谓的对象，也不可能在调用 StdWndProc()前帮我们将 ECX 设定为对象的地址。(**诡异，太诡异了**。)如果只是简单静态阅读代码，到此我们基本不知所措了。“代码不决问调试”。在 StdWndProc 中设定断点，运行并停止在断点处，然后在反汇编下单步跟踪，直到该函数执行 ret 指令返回。我们要看看它到底返回到哪里，想看看调用它的函数怎么神奇地设定了 ECX。返回到的代码如图 7.8 所示，最后一行为 StdWndProc 返回后的指令。按组合键 Ctrl+Alt+M，激活模块窗体，从中可知这些指令地址为 0x77D1xxxx，而该地址段对应模块为系统的 USER32.dll（图 7.8 右部标注的条目指示其基址为 0x77D10000），也就是说，确实是操作系统代码调用了 StdWndProc。但操作系统不会帮我们设定 ECX。再分析一下指令，call 指令调用消息处理函数前的所有指令确实没有一条是设定 ECX 的。(**不可思议！**)

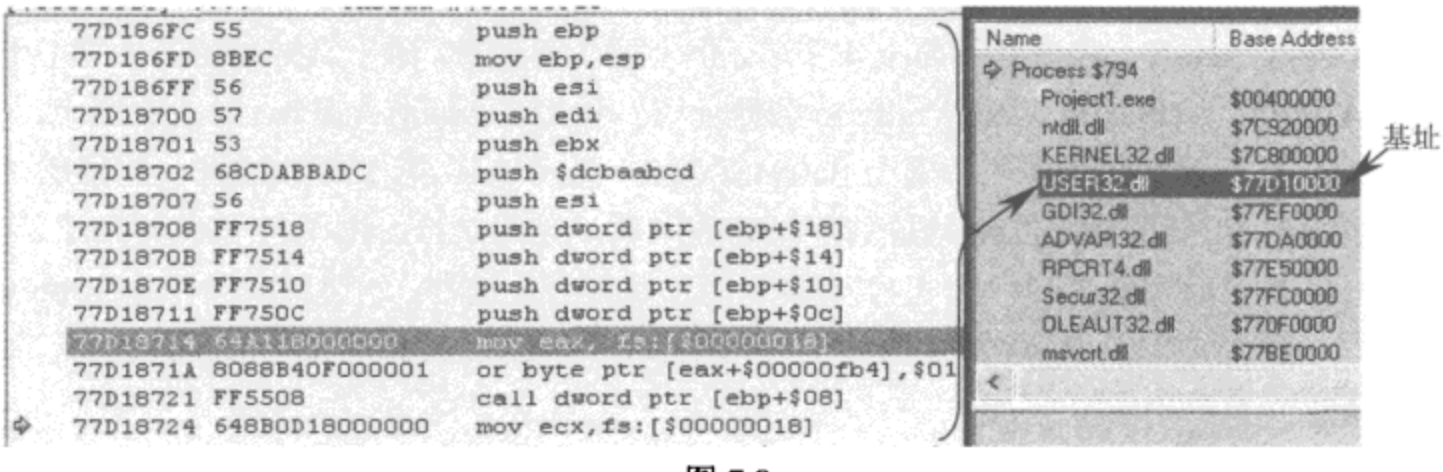


图 7.8

在图 7.8 的 call 指令处设置断点，然后等待下次消息处理经过这里时跟踪进去，看到底在哪里 ECX 被神奇地设定了。当程序再次停在 call 指令处后，按 F7 键，跟踪进函数跳转到一段代码，如图 7.9(a)所示。这里是一条 call 指令，再次 F7 跟踪进该指令，如图 7.9(b)所示，这是一条 pop ecx 指令。(**终于看到设置 ECX 的指令了，可它是什么意思？**)图 7.9(a)中的 call 指令将其下一条指令 clc 的地址 003F0FC0，压栈，接着图 7.9(b)中马上用 pop ecx 指令将这个地址出栈，放到 ECX 中。单步执行 pop ecx 后，见图 7.9(c)，其中寄存器 ECX 的值确实是 003F0FC0。之后的 jmp StdWndProc 就跳转到上面分析的 StdWndProc()函数。从图 7.8 中操作系统的 call 指令到 StdWndProc 被调用所执行的 4 条指令中，pop ecx 设定了 ECX。从图 7.9(a)的 call 到图 7.9(b)的 pop 指令，一个压栈一个出栈，两条指令执行后除了设定 ECX 外，ESP 的值未变，并未影响栈。图 7.4 中的 MOV EAX, [ECX+4]意味着 ecx + 4 内存中存放的是 this 指针。

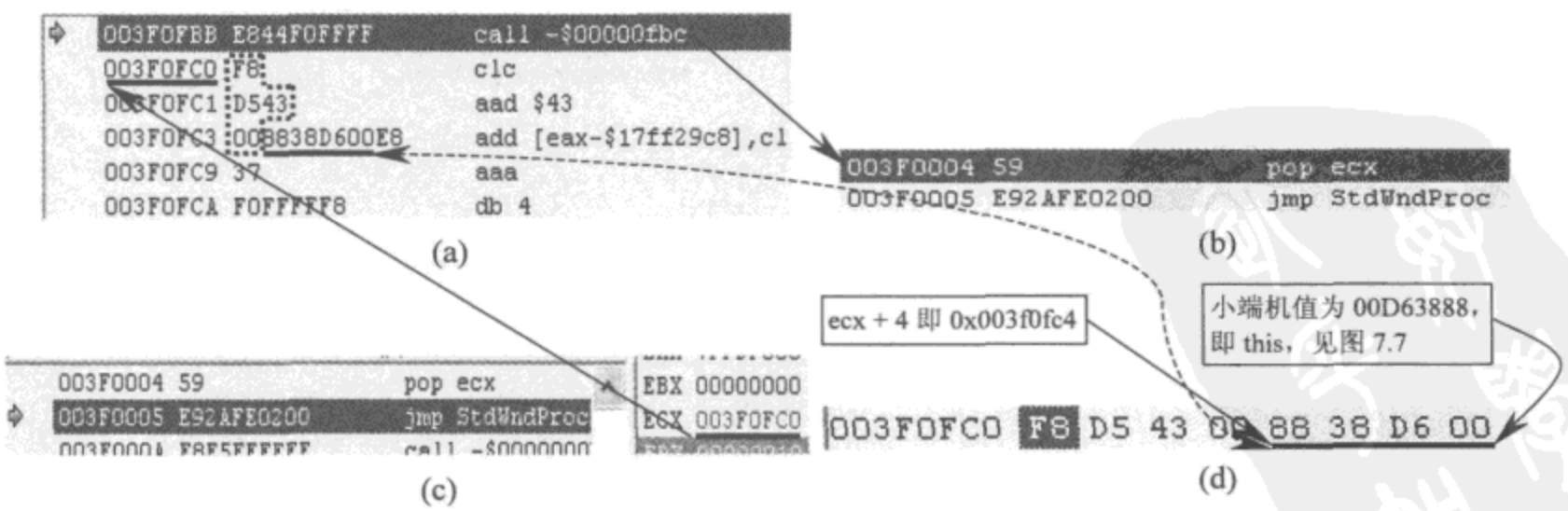


图 7.9

通过内存查看我们容易得知，ecx+4 内存的地址就是 003F0FC4，其中存储的值是 this 指针的值 0xD63888（见图 7.7），如图 7.9(d)所示。其实，ecx+4 就是图 7.9(a)中 call 指令后那些奇怪指令所在的地址。图 7.9(a)中第 4 条指令的机器码“00 88 38 D6 00 E8”中，其第 2 字节“88”（ecx+4 正好指向该字节）到第 5 字节“00”用小端机顺序表示正是 0x00D63888（见图中虚箭头指示）。也就是说，图 7.9(b)的 pop ecx 指令居然获取了“代码段中”存储对象首址（this）的内存的地址。虽然找到了 this 指针的定位代码，可愈发让人疑惑：与按钮相关的消息处理函数代码中包含了 this 指针数据！不仅如此，根据图 7.4 中的代码 CALL [ECX].Pointer（**其反汇编就是** call [ecx]）及图 7.3 的调用栈，我们知道，ECX 指向的内存存储的是 MainWndProc()函数的地址，即 ECX 指向的 8 字节中，前 4 字节存储的是消息处理函数地址，后 4 字节存储的是对象 this 指针。（**代码中居然存储了两个数据！**）也就是说，图 7.9(a)的代码第 2～4 条指令的机器码其实是函数地址和 this。我们已经分析了图 7.9(a)中第 4 条指令包含了 this 指针，它前面存储的应该是 MainWndProc 的地址。依据小端机，从第 4 条指令的第 1 字节开始向上 4 字节，止于第 2 条指令，图中虚框圈定的 4 字节 f8 d5 43 00，即 0x0043d5f8。该地址是 MainWndProc 的地址吗？我们在 MainWndProc 处设置断点，当程序停在该处时，从反汇编可知该其入口确为 0x0043D5F8，见图 7.10 第 1 行的地址。整理图 7.9(a)，我们给出包含 this 和 MainWndProc 地址的消息处理函数，见图 7.11。图 7.11 的头绪似乎还不很明晰，结合图 7.4，我们通过图 7.12～图 7.14 来演示消息处理的全过程，分别对应执行 call、pop 和 jmp 指令的状况，通过它们从代码中获取 this 和消息处理函数 MainWndProc()的地址，并调用 MainWndProc()。

```
Controls.pas.6132: begin
0043D5F8 55                 push ebp
0043D5F9 8BEC               mov ebp,esp
0043D5FB 83C4F8             add esp,-$08
```

图 7.10

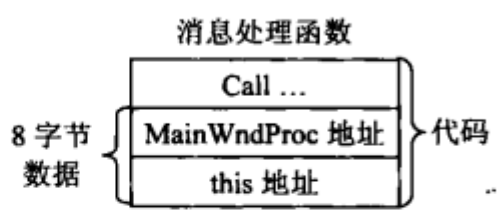


图 7.11

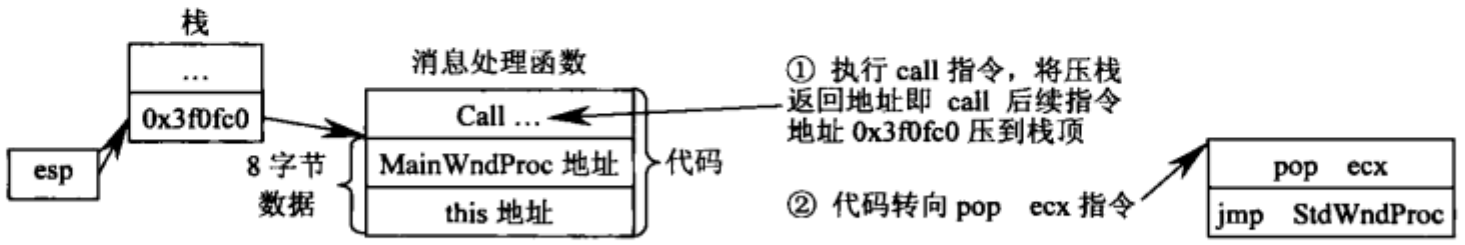


图 7.12

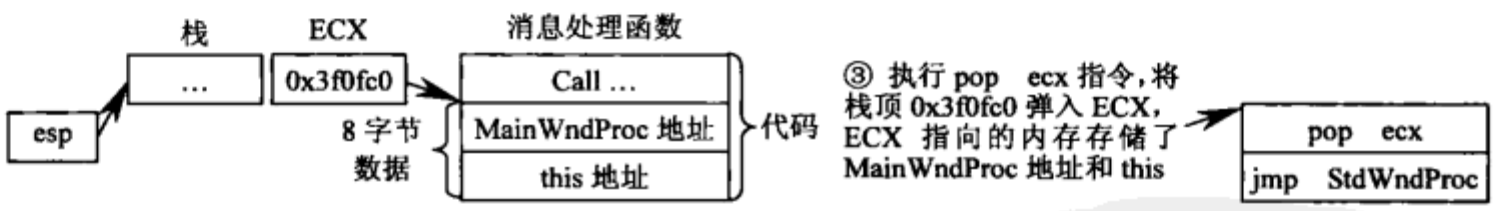


图 7.13

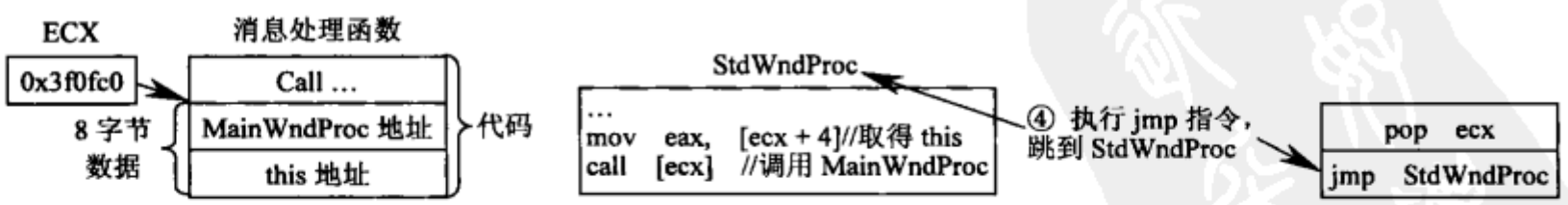


图 7.14

再次聚焦到图 7.11 所示代码，我们要看看这段指令混合数据的代码到底怎样写出来的。既然作为窗体消息处理函数被系统调用，那么它极可能通过 SetWindowLong 设定。只要找到调用 SetWindowLong 的代码，就可顺藤摸瓜找到这段诡异代码。哪里会调用这函数呢？一个个文件中查

吗？太慢。我们还是用“断点大法”吧。

为了跟踪到框架调用 SetWindowLong()，我们应该用编码方式（非拖曳）构造一个按钮，这时其消息响应函数应该通过 SetWindowLong() 设定。如果我们在 SetWindowLong 的入口处设置断点，则可快速追踪到想查看的东西。将按钮单击响应代码替换成 DM7-14，然后运行新代码。在 SetWindowLong()函数开始处设置断点：激活模块窗体，见图 7.15，选择左边的 USER32.dll，双击右边的 SetWindowLongA，将转到反汇编界面该函数起始指令，在此处设置断点，然后单击按钮，程序果然停止在断点处。

DM7-14

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Btn: TButton;
begin
  Btn := TButton.Create(nil);
  Btn.Parent := Self;
end;
```

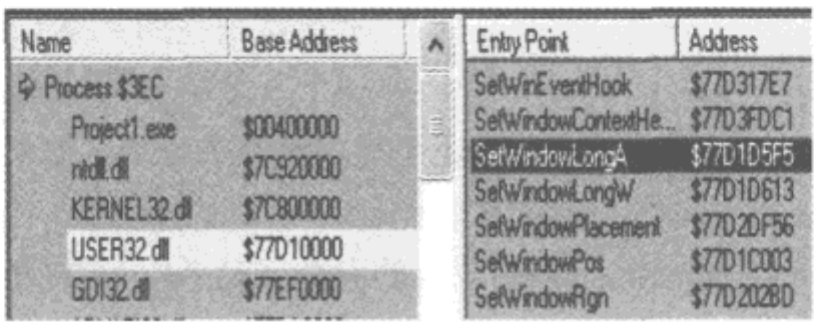


图 7.15

基于调试的代码阅读技巧 3：快速定位框架中和系统调用相关的代码

如果我们认为框架代码调用了某个系统函数，为了快速找到框架中调用该函数的代码，只需在系统函数的入口指令处设置断点，再运行程序，使其在断点处停止。然后，一步一步地单步返回，或通过查看此时栈顶的返回值直接找到调用代码。

在 Visual C++中给系统设置断点

Visual C++ 6.0 中没有模块窗体；VS 有，但无导出函数表。此时，可写一段代码调用 SetWindowLong()，然后用反汇编跟踪到其代码入口，记下该地址（见 1.8.3 节最后的分析，系统 DLL 为了共享代码段，会加载到固定地址，所以在同一系统中，系统 API 函数入口地址就固定不变，因此才能将一个程序中获得的 API 地址用到其他进程），然后在被调试程序中的反汇编窗口中，通过设定该地址转到 API 入口，设置断点。

我们不想一行一行单步跟踪返回到调用 SetWindowLong()的代码。查看栈窗体，见图 7.16，此时栈顶值为 0x0042F449，它就是调用 SetWindowLong()后要返回的地址，也是我们想要查看的代码所在地址。在反汇编代码窗体中单击右键，在弹出的快捷菜单中选择“Goto Address”，然后在弹出窗体中输入该地址并回车。在该地址设置断点，见图 7.17。之后按 F9 键，运行程序。最终程序停在新断点处——我们关注的代码。

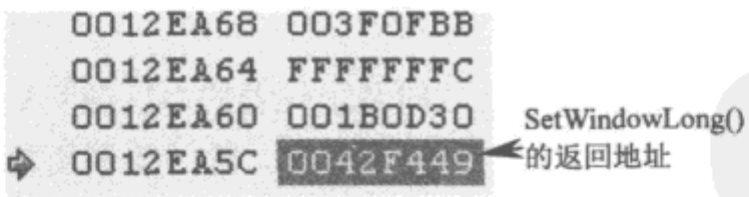


图 7.16

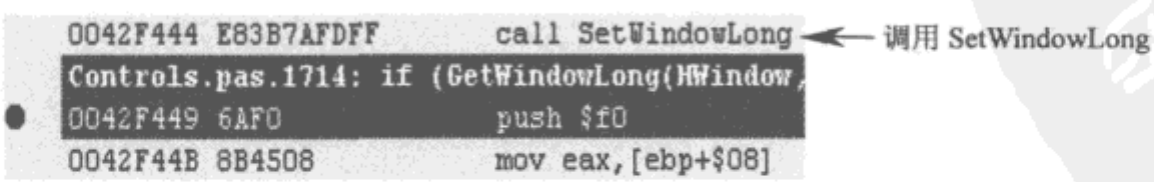


图 7.17

关键部分如下：

SetWindowLong(HWindow, GWL_WNDPROC, **Longint(CreationControl.FObjectInstance)**);

其中，第 3 个参数就是那段奇怪的包含 this 指针和函数地址的消息处理代码的入口地址。CreationControl 是一个全局变量，类型是类 TWinControl，而 FObjectInstance 是其私有变量。包含这段代码的文件为 Controls.pas。既然 SetWindowLong()使用了 FObjectInstance，那么必然有地方对其赋值。FObjectInstance 为 TWinControl 私有变量，且该类在 Control.pas 中定义，对 FObjectInstance 的赋值语句必然也在该文件中（与 C++不同，Delphi 的类只能定义在一个源文件中），按 Ctrl+F 组合键，激活搜索界面，查询字符串"FObjectInstance := "，在 TWinControl 的构造函数中找到，见 DM7-15。

DM7-15

```
constructor TWinControl.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FObjectInstance := Classes.MakeObjectInstance(MainWndProc);
    ...
end;
```

分析代码，MakeObjectInstance 返回值赋予了 FObjectInstance，而 FObjectInstance 又是消息处理函数的地址，显然 MakeObjectInstance 返回的就是消息处理函数地址。而该地址指向的代码中包含了 this 指针和一个函数（MainWndProc（见图 7.11））地址。注意，MakeObjectInstance 传入的参数就是 MainWndProc。看到这里，自然推测 MakeObjectInstance 接收了 MainWndProc 地址作为参数，然后“加工”成了一段代码，在其中嵌入了 MainWndProc 地址，并返回该段代码的入口地址（我们在第 1 章中做了不少直接构建机器码的工作）。MakeObjectInstance 生成了机器代码么？这里尚有一点不明确：生成的代码中不仅嵌入了 MainWndProc 地址，还包含了 this 地址，而 MakeObjectInstance 似乎只传入了 MainWndProc 地址，它怎能创建出包含了 this 的代码呢？既然这里有困惑，就在 DM7-15 的调用 MakeObjectInstance 处设置断点，用反汇编跟踪，见图 7.18，注意 call 指令前的 2 条 push 指令，其中 push $0043d5f8 明显是在传递 MainWndProc()的地址（图 7.10 及相关论述已证实 0x0043d5f8 就是 MainWndProc()的入口），另一条应该传递 this 指针。查看 EAX 和 Self 的值很容易证实该结论，见图 7.19。

```
FObjectInstance := Classes.MakeObjectInstance(MainWndProc);
        mov eax,[ebp-$04]
        push eax <----this 地址
        push $0043d5f8 <----MainWndProc 地址
        call MakeObjectInstance
```

图 7.18

图 7.19

原来，在 Delphi 中，类的非静态成员函数的名字代表的不仅像 C/C++那样是函数地址，还包含了对象的首址。想想也合理，非静态成员函数被回调时，除了需要函数入口地址，还需要传递对象首址。Delphi 有一个特有语法特性：“类的非静态回调函数指针”，不是单纯的入口地址，而是由对象首址和函数地址，共 8 字节。Delphi 的这一“发明”延续到.NET 中就是分派，在软件扩展性上，比 C++、Java 等基于虚函数回调更灵活，耦合更松。

有了这个发现，我们更加坚定了前面的猜测，MakeObjectInstance 应该构建了一段机器码，包含了 this 和消息函数地址。现在是剖析 MakeObjectInstance 的时候了，见 DM7-16。为了熟悉 C++的读者阅读方便，需要剖析的代码均转换成 C++代码了。

DM7-16

```
1    void* MakeObjectInstance(Method: TWndMethod) {
2        char BlockCode[2] = {
3            0x59,                                          // pop   ecx
4            0xE9    };                                     // jmp   StdWndProc
5        int PageSize = 4096;
6        PInstanceBlock Block;
7        PObjectInstance Instance;                          // PObjectInstance 是 TObjectInstance *
8        if (InstFreeList == NULL) {
9            Block = (PInstanceBlock)VirtualAlloc(NULL, PageSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
10           Block->Next = InstBlockList;
11           memcpy(Block->Code, BlockCode, sizeof(BlockCode));
12           Block->WndProcPtr = (void *)(CalcJmpOffset(&Block->Code[1], StdWndProc));
13           Instance = Block->Instances;
14           do{
15               Instance->Code = 0xE8;                     // call   near ptr Offset
16               Instance->Offset = CalcJmpOffset(Instance, &Block->Code);
17               Instance->Next = InstFreeList;
18               InstFreeList = Instance;
19               Instance++;
20           }while ((Longint)Instance - (Longint)Block < SizeOf(TInstanceBlock));
21           InstBlockList = Block;
22       }
23       void * Result = InstFreeList;
24       Instance = InstFreeList;
25       InstFreeList = Instance->Next;
26       Instance->Method = Method;
27       return Result;
```

阅读源代码往往是印证自己猜测和破除疑惑。有了前面猜测，我们容易注意到第 3、4 行的注释，“pop　ecx”正是图 7.12 中右边的代码，从图 7.9(b)查看其机器码可见：pop　ecx 的机器码为 0x59，jmp 的操作码为 0xE9。这正与 DM7-16 中的第 3、4 行中的 2 字节吻合。(**看来该函数真在构建机器码。**)根据这一点，容易发现第 11 行将数组 BlockCode 的机器码复制到 Block->Code。

那么，图 7.11 的代码片段在哪里构造的呢？既然 MakeObjectInstance 传入参数 Method 包含了 this 和函数地址，那么一定有某个地方将参数 Method 赋值给某块内存。易知第 26 行完成了该任务。DM7-16 中涉及的两个结构体见 DM7-17。

TObjectInstance 存储了图 7.11 所示的代码片段。其成员 Code 是 call 指令的操作码，在 DM7-16 的第 15 行被赋值为 0xE8。成员 Offset 为 call 指令的偏移量，在 DM7-16 的第 16 行被赋值，其值由 CalcJmpOffset 计算。成员 Method 前 4 字节为消息函数地址（离结构体头部 5 字节），后 4 字节为 this 指针（离结构体头部 9 字节），见图 7.20。

DM7-17

(a)

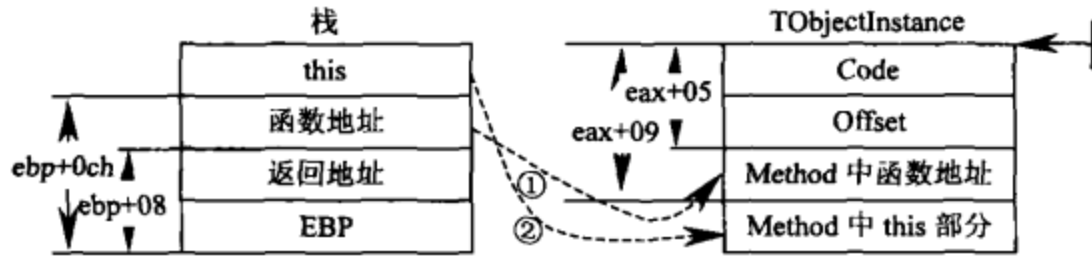
```
//这个结构体按 1 字节对齐，没有填充
struct TObjectInstance {
    char Code;
    int Offset;
    union {
        PObjectInstance Next;
        //这个指针包含 this 和函数地址 8 字节
        TWndMethod Method;
    }value;
};      //TObjectInstance 是 TObjectInstance*
```

(b)

```
//这个结构体按 1 字节对齐，没有填充
struct TInstanceBlock {
    PInstanceBlock Next;
    char Code[2];
    void *WndProcPtr;
     TObjectInstance
Instances[InstanceCount];
};// PInstanceBlock 是 TInstanceBlock*
```

下面通过反汇编展示 this 和函数地址赋值的逻辑。先确定 MakeObjectInstance 的参数地址，从图 7.18 可知，因先压栈 this，所以其存储地址为 ebp+0ch；后压栈函数地址，因此其存储地址是 ebp+08，见图 7.20 中的栈部分。图 7.21 展示了 MakeObjectInstance 对 TObjectInstance 的成员 Method 赋值的反汇编。第 1 条指令将 eax 设定为结构体 TObjectInstance 首址。第 2 条指令将函数地址参数（其地址 ebp+08）设定给 edx，第 3 条指令将函数地址（EDX 中的值）赋值给结构体首部偏移 5 字节处。

同理。最后一条指令将 this 赋值给结构体首部偏移 9 字节处。至此，图 7.11 所示代码的构造逻辑已基本清楚。



① 将参数函数地址赋值给 TObjectInstance；② 将参数 this 赋值给 TObjectInstance

图 7.20

```
Instance^.Method := Method;
    mov eax,[ebp-$0c]
①   mov edx,[ebp+$08]
    mov [eax+$05],edx
②   mov edx,[ebp+$0c]
    mov [eax+$09],edx
```

①、②完成图 7.20 对应的任务

图 7.21

在找到了构造机器码逻辑后，下面将整个 MakeObjectInstance 函数梳理一下。

首先是数据结构。TObjectInstance 用于构建图 7.11 的代码片段。它是一个单向链表，其 next 指针和 Method 为 union。如果该节点没使用，就用 next 串起来，形成可用链，一旦取下用于生成机器码，则 next 不起作用，该内存用于存储 this 和函数地址（这是缓冲池算法，速度快）。Code 存储 call 指令的操作码，Offset 存储 call 指令的偏移量，共 5 字节。

TInstanceBlock 由两部分构成（见 DM7-17(b)）：① 图 7.12～图 7.14 中的 pop 和 jmp 指令片段，由成员 Code 和 WndProcPtr 组成。Code 的 2 字节分别是 pop　ecx 的机器码（0x59）和 jmp 指令操作码（0xE9）。WndProcPtr 紧邻 jmp 的操作码，为 jmp 的偏移量。② 一个包含 InstanceCount 个元素的 TObjectInstance 数组。元素间通过 next 指针将其链接在一起。

然后是机器码生成算法的数据结构，见图 7.22（分配一个 TObjectInstance 用于构建代码前）和图 7.23（分配一个 TObjectInstance 用于构建代码后）。TInstanceBlock 块上的所有 TObjectInstance 块中的 call 指令均跳到它们所属的这个 TInstanceBlock 块上的 pop 和 jmp 指令片段。分配一个 TObjectInstance 后，其 Next 成员用于存储 this 和函数地址，见图 7.23。

在 DM7-16 中，第 8 行判断可用块 InstFreeList 为空时，执行第 9～22 行代码。第 9 行分配一个内存页作为结构体 TInstanceBlock。第 11～12 行填写 pop　ecx 和 jmp　StdWndProc 机器码。第 13～20 行将新分配的 TInstanceBlock 块的 Instances 数组的每个元素初始化：填写 call 指令机器码（它转跳到第 11～12 行构建的机器指令），并通过 Next 指针将元素链接起来。

第 23～25 行从链上取下一个 TObjectInstance，构建图 7.11 的机器码。其中，call 指令已经在第 15～16 行构建完毕，因此只是通过第 26 行将参数 Method 包含的 this 和函数地址设定到 TObjectInstance 块中，并在第 27 行返回该块的地址，作为消息处理函数的入口地址。

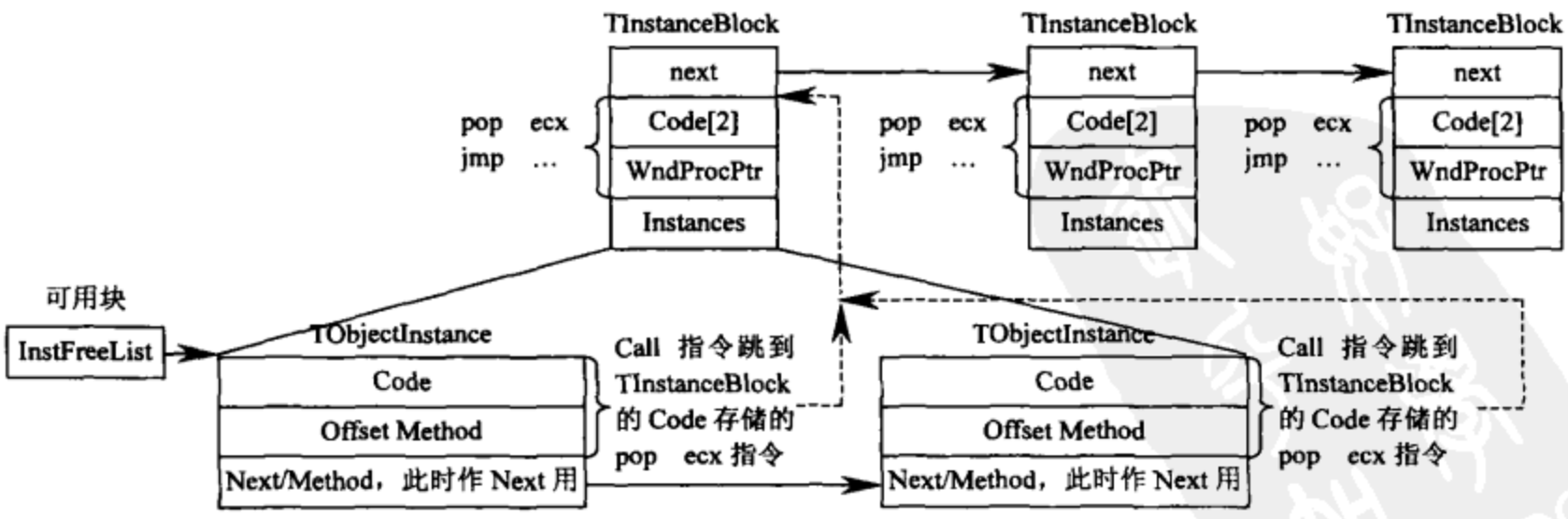


图 7.22

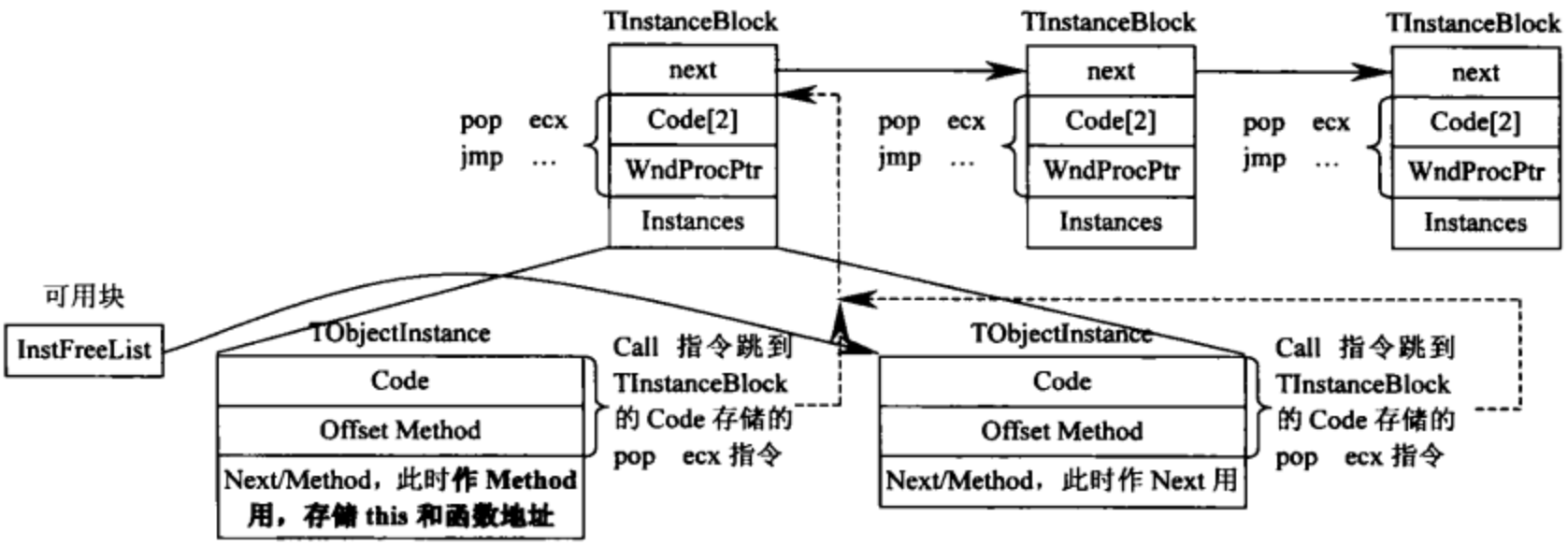


图 7.23

为了解决操作系统回调函数无法传递对象首址问题，Delphi 通过直接生成机器码，将对象首址和函数入口地址这些“数据”紧随 call 指令放在其后，通过 call 到一条 pop 指令“自动”计算出这些“数据”存放的地址。这种技术被称为自定位代码，在病毒设计时比较常用。既然通过调试和分析我们在迷雾中看清了这诡异而精致的技巧，同时完美地解决了 7.4.1 节开始提出的全局句柄-对象指针表的问题，那么我们就做一个自己的自定位版本吧。

7.4.2 我们的自定位代码

新版本的关键是自定位代码的构造。Delphi 的实现方案比较完善（**预分配一页，将代码块所需内存串成链，需要则从中取出，不需要则重新加入链。相当于一个缓存池，速度较快。其实还解决了另外一个问题，见** 7.4.3 **节。**），但作为实验自定位技巧似乎引入了更多复杂性。我们需要更简单、更纯粹的做法，排除其他因素干扰。

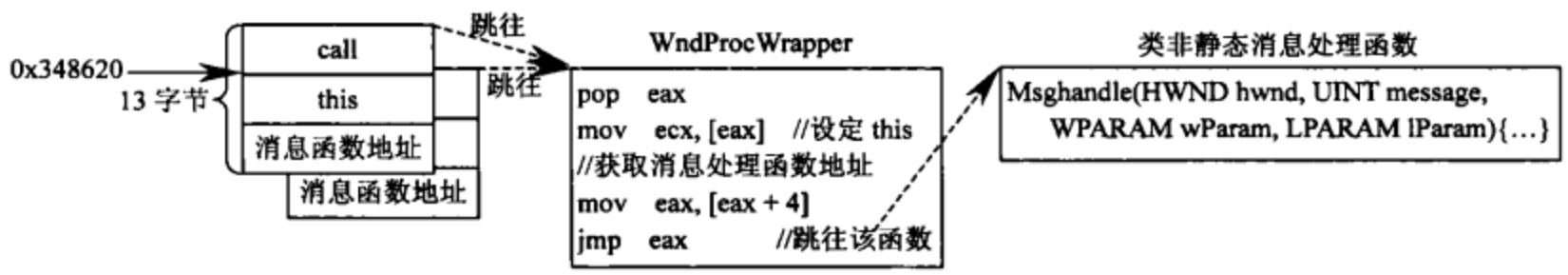


图 7.24

动态生成的机器码片段只有 call 指令片段，pop 指令被放入函数 WndProcWrapper 中并非动态生成。下面结合代码执行过程中栈的变化，给出该方案原理，见图 7.25。图 7.25(a)展示了操作系统调用用户注册的窗体函数后的栈，此时 EIP 指向图 7.24 的 call 指令。执行 call 后，其返回地址 0x348620 压栈，该值实为 call 指令后存储 this 指针信息的地址，见图 7.25(b)。call 将跳到 pop 指令，执行该指令后，0x348620 被压入 EAX，用 EAX 就可访问到 this 指针等，且栈恢复到图 7.25(a)的样子，见图 7.25(c)。之后，通过 EAX 获得 this 和消息处理函数地址，并调用该函数（图 7.24 中的 MsgHandle）。但该函数返回时，将图 7.25(c)栈顶的返回地址弹入 EIP，完成返回。代码不返回到 WndProcWrapper。

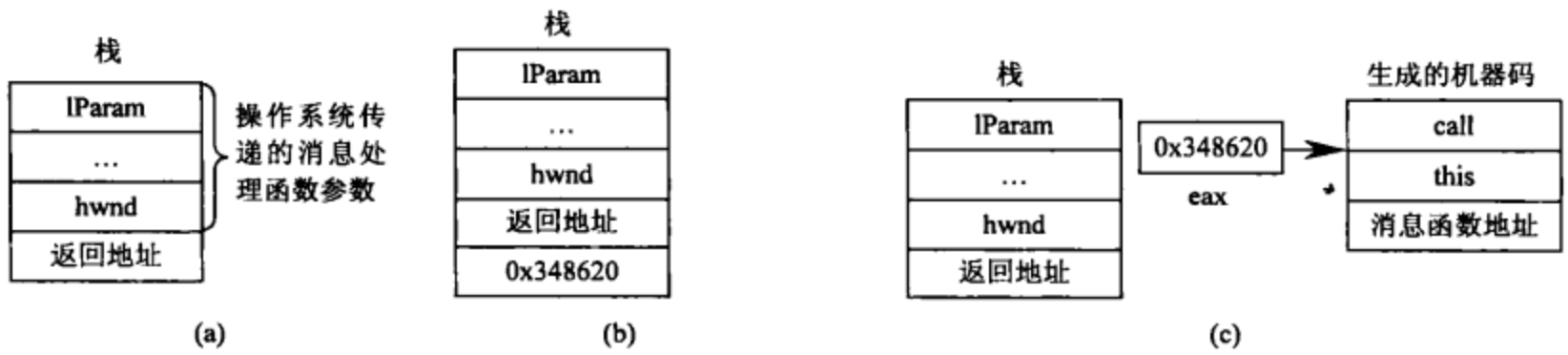


图 7.25

对照图 7.24，我们要写两块代码：WndProcWrapper()和 call 指令所在代码片段。函数 WndProcWrapper()为了避免被编译器悄悄加入机器指令（在一开始加入 push ebp 就会影响栈布局，使其与图 7.25 不同，从而发生错误），采用了 Visual C++的关键字__declspec(naked)，见 DM7-18(a)。

call 指令所在代码片段如何构造？模仿 Delphi 的做法，将分配内存并填入机器码和相关 this 等信息，由函数 makeWndProcStub()完成，见 DM7-18(b)。

DM7-18

(a)

```
__declspec(naked) LRESULT CALLBACK WndProcWrapper(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam){
    _asm{
        pop eax;
        mov ecx, [eax];
        mov eax, [eax + 4];
        jmp eax;
    }
}
```

(b)

```
1     void * makeWndProcStub(void * objPtr, void * msgHandler) {
2         char * code = (char *)malloc(13);
3         code[0] = 0xe8;
4         *((long*)&code[1])=(long)((char *)WndProcWrapper - (code+5));
5         *((void **)&code[5]) = objPtr;
6         *((void **)&code[9]) = msgHandler;
7         return code;
8     }
```

为了简化逻辑，我们没有事先分配一个页面的内存并管理它（如 DM7-16），而是需要一个代码片段，用 malloc 分配一片刚够用的内存。因此，DM7-18(b)的第 2 行分配了 13 字节，见图 7.24，其中 call 指令 5 字节（一个操作码+4 字节偏移），this 为 4 字节，消息函数地址 4 字节共 13 字节。第 3 行填写了 call 指令的操作码 0xe8，第 4 行写了 4 字节转跳偏移量。如何计算偏移呢？由 1.3.1 节知，偏移=转跳目的地址-call 指令后续指令地址。转跳到的函数为 WndProcWrapper，所以第 4 行中被减数为 WndProcWrapper。因为 call 指令占 5 字节，所以其后续指令地址为 code+5（call 指令放在 code 处）。第 5 行和第 6 行分别填入对象首址和消息处理函数地址。第 7 行返回这 13 字节，即代码片段的首地址，以便之后程序将其通过 SetWindowLong 注册到系统中。

7.4.3 自定位代码版 Button 类

为了避免引入太多不确定因素，我们修改程序应该验证一点改一点。所以先将 Button 类从全局句柄-对象指针表中解脱，Form 依然用老方式（事实证明后面它确实有点问题）。

首先要修改的是构造函数，见 DM7-19(a)。第 3～9 行只做了一件事，就是将 Button 的成员函

数 WndProc 地址设定给指针 func。为什么如此麻烦？因为 Visual C++编译器不允许我们将 Button::WndProc 赋值给 func，即使将 Button::WndProc 强制转换为 void *，编译也出错，所以嵌入汇编，绕过编译器的检查。第 10 行调用 makeWndProcStub()，将 this 和消息处理函数地址 func 传递给它构造 call 指令代码片段，并将代码片段地址返回。第 11 行将该代码片段首址作为消息处理函数入口地址注册到新生成的按钮窗体中，替换原始的消息处理函数。

然后，处理这段动态生成代码释放的问题。如不释放，内存泄漏。何时释放？自然是不需要按钮的时候，还是我们习惯性的析构释放资源，见 DM7-19(b)。

DM7-19

(a)

```
1    Button::Button(int x, int y, int width, int height, char * caption, HWND parentHandle) {
2        m_handle = CreateWindowA("BUTTON",...);
3        LRESULT (_ _thiscall Button::* ptrFunc)(HWND,UINT,WPARAM,LPARAM);
4        ptrFunc = (&Button::WndProc);
5        void * func;
6        _ _asm{
7            mov eax, ptrFunc;
8            mov func, eax;
9    }
10   m_wndProcStub = makeWndProcStub(this, func);
11   m_oldWndProc = (WNDPROC)SetWindowLong(m_handle, GWL_WNDPROC, (LONG)m_wndProcStub);
 ...
}
```

(b)

```
Button::~Button(void){
...
1        DestroyWindow(m_handle);
2        free(m_wndProcStub);
3        SetWindowLong(m_handle, GWL_WNDPROC, (LONG)m_oldWndProc);
4    }
```

比较好理解第 1~2 行，先销毁按钮窗体，再销毁消息处理函数代码片段。为什么还需要第 3 行？为了避免引入过多干扰，笔者开始并没有加入第 2 行和第 3 行，一切都很好。加入第 2 行后，就出现异常，提示访问了无效地址。当时分析可能是在释放代码（m_wndProcStub）后，按钮控件又接到了新消息，此时调用了这段已释放的代码，所以出现异常。为了证明该猜测，我们需要反汇编跟踪，查看是否 free 后该段指令被再次调用。自然的做法是在 m_wndProcStub 指向的代码中设置断点，但不能事先设置，应该在调用 free 后再设置，否则消息将不断导致断点中断。事实证明，在 free 后 m_wndProcStub 指向的代码再次被执行。而 Visaul C++调试版执行 free()后，被释放的内存被重新填写了其他字节，相当于修改了被执行代码。程序执行到这里，如果继续执行这些被篡改的指令，必然出现问题。解决办法就是，释放后用第 3 行将按钮的消息处理函数设置为以前的老函数，在处理后续消息时就不会执行已释放的代码了。

Button 类的其他代码不用修改。至此，自定位代码 Button 版升级成功。

7.4.4 自定位代码版 Form 类

似乎按照 Button 的修改方法改动 Form 类很顺理成章。让我们来写一下 Form 的构造函数，见 DM7-20。第 1 行构造 call 代码片段，第 2～3 行构建窗体类结构体，第 4 行注册窗体类结构，第 5 行用该注册窗体类构建窗体。第 2 行中，窗体类结构体的成员 lpfnWndProc 为该窗体的消息处理函数，设定为第 1 行构造的 call 代码片段首址 m_wndProcStub 吗？（有问题。）这段 call 代码片段混

合了 this 地址，也就是说，窗体类的共用消息处理代码总是与某个特殊的对象 this 绑定（即执行时将获取该对象首址）。

DM7-20

```
1   m_wndProcStub = makeWndProcStub(this, func);
2   wndclass.lpfnWndProc = ?;      //等号右边为 m_wndProcStub 吗
    ...
3   wndclass.lpszClassName = className ;
4   RegisterClass (&wndclass)
5   m_handle = CreateWindow (className, ...) ;
```

为什么 Button 类没有遇到这个问题？因为它不用构造自己的窗体类结构体，直接用名为“BUTTON”的窗体类 CreateWindow 即可（见 DM7-19(a)第 2 行）。这样，只需在构造按钮后调用 SetWindowLong 为每个按钮设定它“独有”的 call 指令片段。照此思路，我们将 DM7-20 中的问号替换成窗体的缺省处理函数 DefWindowProc，然后在第 5 行后加入“SetWindowLong(m_handle, GWL_WNDPROC, (LONG)m_wndProcStub)”即可。

析构函数参照 Button 析构的处理，如 DM7-21。运行结果出现内存访问违规错误。查看出错时调用栈，问题发生在 free()函数调用时。阅读 DestroyWindow 的帮助发现，当调用它时将发送 WM_DESTROY 消息。认真分析代码，在 7.3 节的 DM7-11(a)中，消息处理函数收到 WM_DESTROY 时调用了 delete()，析构自己。析构函数被调用后，执行 DestroyWindow()会再次发送 WM_DESTROY，则消息处理函数中又会调用析构函数。这将导致多次释放而出错。因此，只有直接调用析构函数删除窗体时才应该调用 DestroyWindow。如果是通过接收到 WM_DESTROY 消息释放窗体，在析构函数中不需要调用 DestroyWindow。修改消息处理函数和析构函数，见 DM7-22，完整代码见“code\第 7 章\button_obj_stubcode”。

DM7-21

```
Form::~Form(){
DestroyWindow(m_handle);
free(m_wndProcStub);
SetWindowLong(m_handle, GWL_WNDPROC,
(LONG)DefWindowProc);
...
}
```

DM7-22

(a)

```
LRESULT Form::WndProc(HWND hwnd, ...) {
    switch (message){
    ...
        case WM_DESTROY:
        if (m_destroyState == dsNo) { //表明因窗体关闭发送的消息
          m_destroyState = dsClose; //此时才调用 delete 析构对象
          delete this;
       }
       return 0 ;
    }
    return DefWindowProc(hwnd, ···);
}
```

(b)

```
Form::~Form() {
    if (m_destroyState == dsNo) {
        //如果直接释放对象，则调用 DestroyWindow
        m_destroyState = dsFree;
        DestroyWindow(m_handle);
    }
    free(m_wndProcStub);
    SetWindowLong(m_handle, GWL_WNDPROC,
                          (LONG)DefWindowProc);
...
}
```

7.4.5 为什么不错呢

笔者一次在给学生布置本节所述自定位代码版本练习时，发现了一件奇怪的事：学生的消息处理函数并没有标记为_stdcall，这意味着不是被调用函数清栈，即返回指令非 ret x，而是 ret，那么栈的平衡就会被破坏（其后果见 2.4 节）。但为什么他的程序运行正常？

这时，我们祭起反汇编调试大法，关注点就是 user32.dll 中消息函数调用返回后的代码逻辑。像 7.4.1 节那样设置好断点，反汇编，如图 7.26 所示。第 9 行 call 指令返回后，第 12 行做了奇怪的比较，判断栈上的值是否为 0xdcbaabcd。与该值的相关代码是第 1 行，它将 0xdcbaabcd 压栈。之后，第 3～6 行是将消息处理函数的 4 个参数压栈，见图 7.27。其中，①和②分别标明了被调函数为 stdcall

和 cdecl 时 call 指令返回后的 esp 和 esp+4 值（因 stdcall 要在被调函数中执行 ret 0x10，将 esp 指针在返回后再增加 16 字节）。

```
1  77D18702 68CDABBADC        push $dcbaabcd
2  77D18707 56                push esi
3  77D18708 FF7518            push dword ptr [ebp+$18]
4  77D1870B FF7514            push dword ptr [ebp+$14]
5  77D1870E FF7510            push dword ptr [ebp+$10]
6  77D18711 FF750C            push dword ptr [ebp+$0c]
7  77D18714 64A118000000      mov eax, fs:[$00000018]
8  77D1871A 8088B40F000001    or byte ptr [eax+$00000fb4],$01
9  77D18721 FF5508            call dword ptr [ebp+$08]
10 77D18724 648B0D18000000    mov ecx,fs:[$00000018]
11 77D1872B 80A1B40F000000    and byte ptr [ecx+$00000fb4],$00
12 77D18732 817C2404CDABBADC cmp [esp+$04],$dcbaabcd
13 77D1873A 0F85587C0200      jnz +$00027c58
14 77D18740 83C408            add esp,$08
```

图 7.26

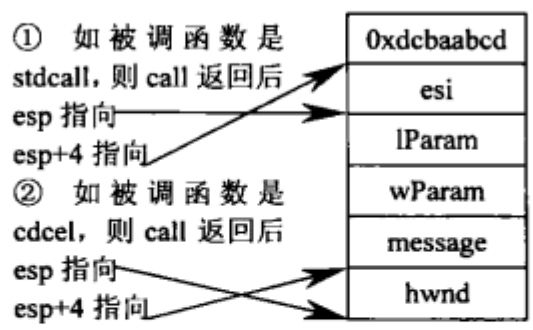


图 7.27

下面分析图 7.26 中的第 12 行代码。假定被调函数是 stdcall，那么第 9 行 call 返回后，[esp + 4] 存储的是 0xdcbaabcd。cmp 的结果是两者相同，所以将执行第 14 行代码，esp 增 8，将保存 0xdcbaabcd，与 ESI 占用的栈空间平衡了。假如被调函数是 cdecl，那么 call 返回后，[esp+4]保存的是 message 值，不等于 0xdcbaabcd。这时，第 13 行转跳到新的分支。明显该分支应该将 ESP 中的值加 16，平衡本该由被调函数平衡的栈。这时，我们用寄存器窗体故意将 ESP 中的值减 16，模拟出 cdecl 的栈情况，将执行图 7.28 所示的代码。

```
77D40398 813C24CDABBADC    cmp [esp],$dcbaabcd
77D4039F 7508              jnz +$08
77D403A1 83EC04            sub esp,$04
77D403A4 E99783FDFF        jmp -$00027c69
77D403A9 83C410            add esp,$10
77D403AC E98F83FDFF        jmp -$00027c71
```

这段代码很奇怪，似乎在兼容一种不同的消息处理函数原型。该原型会多压一个参数，因为 esp 减了 4

该段代码处理被调函数为 cdecl 的情况，将 4 个参数占用的 16 字节的栈平衡（将 esp 加 16）

图 7.28

图 7.28 中，第 5～6 行对应被调函数为 cdecl 的情况。第 3～4 行则是一种非常奇怪的情况，这时的被调消息处理函数是 stdcall，但多了一个参数作为最后一个参数，就是图 7.26 中第 2 行 push 的 esi。从调试来看，似乎这个参数就是被调用的消息处理函数地址。估计这段代码是兼容历史造成的消息处理函数不同原型。这些原型应该有三种：其中两种为 4 参数的 stdcall 和 cdecl，另一种为 5 参数的 stdcall。正是这段代码判断出被调函数的具体调用惯例和原型。（**有意思的技巧和为了兼容的可敬精神**。）其实这段代码的发现是偶然的，笔者以为 Visual C++编译成员函数默认是 cdecl 调用惯例，其实它除了通过 ECX 传递 this，默认就是 stdcall。

7.5 构建我的 GUI 组件（3）：更多的组件

本节继续按照前面思路将编辑框也包装成自定位版本的类，方式如同 Button 类，依然在构造函数中生成自定位代码，见 DM7-23。第 2～3 行构造编辑框。第 4～11 构造自定位代码。第 12 行将自定位代码注册为编辑框的新消息处理函数。

析构函数几乎与 Button 相同，请参见工程源文件。DM7-24 为设置和获取编辑框文本。第 2 行通过向编辑框发送消息 WM_SETTEXT 设定字串，消息的 lParam 为字串的地址。获取框中字串要复杂些，首先要获取字串长度，第 5 行通过发送消息完成该任务，然后分配该长度内存用于保存获取的字串。第 6～7 行释放之前的内存，以免泄漏。第 8 行分配新内存。第 9 行向编辑框发送消息

WM_GETTEXT，让系统将字串复制到 m_text 指向的新分配内存中。工程见“code\第 7 章\button_obj_stubcode_more”。

DM7-23

```
1    Edit::Edit(int x, int y, int width, int height, char * editName, HWND parentHandle) {
2        m_handle = CreateWindowA("EDIT", editName, WS_CHILD | WS_VISIBLE | WS_BORDER,
3                                                    x, y, width, height, parentHandle, 0, applicationHandle, NULL);
4        LRESULT (__thiscall Edit::* ptrFunc)(HWND,UINT,WPARAM,LPARAM);
5        ptrFunc = (&Edit::WndProc);
6        void * func;
7            __asm {
8                mov      eax, ptrFunc
9                mov      func, eax
10       }
11       m_wndProcStub = makeWndProcStub(this, func);
12       m_oldWndProc = (WNDPROC)SetWindowLong(m_handle, GWL_WNDPROC, (LONG)m_wndProcStub);
13       m_text = NULL;
14   }
```

DM7-24

```
1    void Edit::setText(char * value) {
2        SendMessageA(m_handle, WM_SETTEXT, 0, (LPARAM)value);
3    }
4    char * Edit::getText() {
5        int len = SendMessageA(m_handle, WM_GETTEXTLENGTH, 0, 0);
6        if (m_text != NULL)
7            free(m_text);
8        m_text = (char *)malloc(len + 1);
9        SendMessageA(m_handle, WM_GETTEXT, len + 1, (LPARAM)m_text);
10       return m_text;
11   }
```

没想到 7.2.2 节中关于为什么不直接调用 SetWindowLong 返回的消息处理函数，而要调用 CallWindowProc 的问题，在一个偶然中被解开了。为了处理编辑框的消息，用 SetWindowLong 替换了编辑框原有的消息处理函数。而一次下意识的偶然，在替换后的消息处理函数中，笔者没有用 CallWindowProc 而直接调用了旧消息处理函数，结果发生了内存访问违规异常。通过跟踪发现，SetWindowLong 返回的旧消息处理函数的地址是 0xffff0cd3，最高字节竟然是 ff，也就是说，最高位是 1。该地址大于 2GB。这是内核地址空间。用户空间直接访问当然不行。一个编辑框的消息处理函数怎么会在内核空间中？这太不合理了。这间接说明了为什么要通过 CallWindowProc 调用。是否真的在访问内核空间？用 IDA 反汇编 CallWindowProc 所在的 user32.dll，发现 CallWindowProc 存在 DM7-25 所示的代码，从跟踪和分析可知，ESI 中是消息处理函数的地址，其中第 4～7 行正是测试该地址高 2 字节是否为 ff。SetWindowLong 返回的那个怪异地址 0xffff0cd3 的高 2 字节就是 ff，可见这种看似内核的地址是有问题的，代码检出这种地址后在第 10 行跳到特殊分支处理。

CallWindowProc 问题

怎么处理的？通过跟踪，我们发现了 DM7-26 所示的代码。第 1 行 ESI 中依然是消息函数地址，是否是第 2 行调用的函数将该所谓“地址”转换成了真实地址？该函数应该是有返回值的，因为第 3 行检测 eax 是否为 0，所以 EAX 中很可能是转换后的地址。追踪 EAX 的使用否定了这一猜测，因为第 4 行将 eax +18h 指向的内存中存储的值取出存入 ESI。这个 ESI 是否才是真正的消息函数地址？从调试看，该值是 0x77D3B3D4，正是 user32.dll 中的代码。继续跟踪，发现第 14 行调用函数后整个 CallWindowProc 就返回了，因此第 14 行所调函数应该调用了真正的消息函数。分析发现，第 11 行压入的是窗体句柄，第 10 行压入的是消息号（此时 EDI 的值与调用 CallWindowProc 时传

入的参数 message 值相同），第 8、9 行压入了 lParam 和 wParam。自然的猜测是，第 12 行为第 14 行所调函数压入真实的消息函数地址，以便跳过去。为了证明这一点，我们将在 ESI 指示的地址即 0x77D3B3D4 处设置断点，然后按 F10 键，执行第 14 行的函数。如果断点触发，则证明我们的猜想。果然触发断点。SetWindowLong 返回的这个函数地址，在某些情况下，并非函数地址，而是一个编码，通过它可以计算出存储真正函数地址的内存地址。这个编码的标志就是高 2 字节均为 0xff。

DM7-25

```
1    mov    esi, [ebp+argPrMsgHandler]
2    test   esi, esi
3    jz     loc_77D41257
4    mov    eax, 0FFFF0000h
5    mov    ecx, esi
6    and    ecx, eax
7    cmp    ecx, eax
8    push   edi
9    mov    edi, [ebp+argMsg]
10   jz     local_77D1E8E9
```

DM7-26

```
1    mov    ecx, esi
2    call   sub_77D1B971
3    test   eax, eax
     ...
4    mov    esi, [eax+18h]
5    jmp    loc_77D1C5F8
     ...
77D1C5F8
     ...
6    push   0
7    push   0
8    push   [ebp+lParam]
9    push   dword ptr [ebp+wParam]
10   push   edi
11   push   [ebp+argHandle]
12   push   esi
13   push   eax
14   call   sub_77D1874F
```

7.6 天堂阶梯，玩赏框架那如花散落的繁复与如索串珠的简洁之美

现在到了“调构学习法”的第 4 步：分析别人框架，学习优点。其实 7.4 节自定位代码的分析已经开始了本步骤。不过现在是全面剖析学习的时候了。因为我们在自己构造时产生了不少疑问，带着疑问阅读，无疑效率更高。当然，阅读中又会发现对待相同问题各自不同的做法，这也会进一步让我们反思和进步。

先列出前面实现中的问题，最大的问题就是“重复”，太多重复相似的代码散布在不同类中。

重复即罪恶

① 构造函数重复。Form、Button 和 Edit 类除了 CreateWindow 传入的参数有差别，其他代码几乎一模一样，都要构建自定位代码，重新设定窗体消息处理函数。

② 析构函数也有相似代码。调用 DestroyWindow 销毁窗体，释放自定位代码，重设旧消息处理函数。

③ 消息处理函数重复。都要通过 Listener 通知诸如鼠标左键弹起消息。

但这些代码又并非完全重复，总因各自特点有些许差异。“重复即罪恶”，且看 VCL 如何面对这些问题，如何利用面向对象技术，使其既复用又可灵活扩展。

这时我们又要用到 7.4.1 节所讲述的方法：用 VCL 构建一个程序，然后通过打断点并使用调用栈（见图 7.3）查看学习。我们同样构造 Form、Button 和 Edit 三种对象。首先构造如 DM7-1 所示的代码。然后跟踪 TButton 类的构造函数，因为我们版本的 Form、Button 和 Edit 在该函数中有太多的重复，需看看 VCL 的实现。

跟踪 TButton 的构造函数，发现它分别调用了基类 TButtonControl 和 TWinControl 的构造函数。

只有在 TWinControl 的构造函数中调用了 Classes.MakeObjectInstance(MainWndProc)，构造自定位代码片段（见 DM7-15），没有与 CreateWindow 相关的。而且执行构造函数并没有按钮显示，这倒符合事实，我们知道，VCL 中只有设定了控件 Parent 属性（即在哪个窗体上显示）才会显示控件。如何找到调用 CreateWindow 的代码？断点设置，这意味着只要在 CreateWindow 设置断点，就能反过来看清调用 CreateWindow 的序列。采用图 7.15 所示方法，在 CreateWindow 入口设置断点（实际是 CreateWindowExA 和 CreateWindowExW 两处）。设定 Parent 属性时，存在调用序列如下：

SetParent → InsertControl → … → UpdateShowing → CreateHandle → …

先看看 SetParent 的逻辑（为方便，以下均用 C++等价代码），见 DM7-27(a)。其命名规范让我们基本能猜出其含义，如果一个控件新的父窗体 AParent 和旧的父窗体 m_parent 不是一个窗体，则要处理（见第 2 行）。代码很严密：第 3～4 行防止自己将自己作为父窗体；如果以前有父窗体，第 5～6 行将其移除；第 7～9 行表示如果新父窗体不是 NULL，则本控件插入其中。巧妙的是，如果 AParent 为 NULL，则只相当于将控件移除。

在调用序列中我们看到 UpdateShowing 函数，见 DM7-27(b)，第 4 行判断如果控件应该显示，即 ShowControl 为真，那么第 5 行判断如果本控件句柄 m_handle 等于 0 就创建句柄，句柄等于 0 就是没有创建嘛。如果需要时才创建句柄（即第 4 行判断应该显示时），这不就是第 3 章提到的 lazy load 技巧吗（DM3-42 的第 2～3 行）？据说该技巧大幅度优化了 VCL 的性能，特别在 windows 窗体控件个数有限的情况下，大幅度节约了资源。甚至看到资料说，当年 Borland 公司的 JBuilder 性能大幅度提高，其架构师提及主要是得益于 lazy load。这时，我们对比前面自己的做法，在构造函数中分配资源—“调用 CreateWindow”，便看出别人的高明了。之后 VCL 通过 CreateHandle 及后续调用，最后调用了 CreateWindowEx。好了，通过分析构造函数，得出 VCL 和我们第一个不同点。我们在构造函数中直接调用 CreateWindow；而它采用 lazy load，在需要显示时才调用。

DM7-27

(a)

```
1   void TControl::SetParent(TWinControl * AParent) {
2       if (m_parent != AParent){
3           if (AParent == this)
4               throw new EInvalidOperation();
5           if (m_parent != NULL)
6               m_parent->RemoveControl(this);
7           if (AParent != NULL) {
8               AParent.InsertControl(this);
9               UpdateAnchorRules();
10          }
11      }
12  }
```

(b)

```
1   void TWinControl::UpdateShowing() {
2       bool ShowControl;
3       ShowControl = m_visible ||… ;
4       if (ShowControl) {
5           if (m_handle == 0) CreateHandle();
6               …
7       }
8       …
9   }
```

VCL 如何解决不同控件在创建时代码重复度很高的问题呢？TButton 的继承关系如图 7.29 所示。在真正构造 TButton 控件过程中，涉及的调用序列 SetParent、InsertControl、UpdateShowing、CreateHandle、CreateWnd 等均属于基类 TWinControl。对照 Button、Edit 类来看，相似的创建逻辑均在各自构造函数中，主要是 CreateWindow 的参数不同。那么，可从它下手，看看 VCL 怎样将参数依据控件不同而差异化的。

谁调用了 CreateWindow？CreateWindowHandle，见 DM7-27(c)。传递给 CreateWindow 的不同参数是靠该函数的结构体 Params 传递的。那么，TWinControl 不同子类必然生成不同的 Params，构造这个 Params 的函数也许是虚函数，谁调用了 CreateWindowHandle，谁就负责传递该参数，就该

看到构建不同 Params 的代码。从调用栈可知，TWinControl::CreateWnd 调用了 CreateWindowHandle，见 DM7-27(d)。第 17 行调用 CreateWindowHandle，将 Params 传递给它。哪里构造了 Params？第 5 行调用 CreateParams，将 Params 的地址传递进去。自然的想法就是 CreateParams 通过地址将 Params 的成员填写了。那么，不同的类要产生不同的 Params，CreateParams 就该是虚函数。查看该函数，确实是虚函数。而 TButton 覆盖了该函数吗？直接在 DM7-27(d)的第 5 行设置断点，运行调试，单步跟踪进 CreateParams，果然调用到的是 TButton 子类的覆盖版，见 DM7-27(e)。

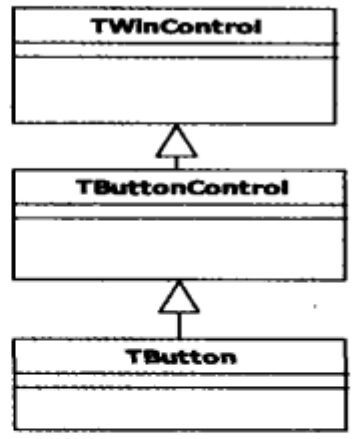


图 7.29

DM7-27

(c)

```
1   void TWinControl.CreateWindowHandle(TCreateParams Params) {
2       m_handle = CreateWindowEx(Params.ExStyle, Params.WinClassName,
3                                 Params.Caption, Params.Style, Params.X, Params.Y, Params.Width,
4                                 Params.Height, Params.WndParent, 0, Params.WindowClass.hInstance,
5                                 Params.Param);
6   }
```

(d)

```
1   void TWinControl::CreateWnd() {
2       TCreateParams Params;
3       TWndClass TempClass;
4       bool ClassRegistered;
5       CreateParams(&Params);
6       m_defWndProc = Params.WindowClass.lpfnWndProc;
7       ClassRegistered = GetClassInfo(Params.WindowClass.hInstance, Params.WinClassName, TempClass);
8       if (!ClassRegistered || (TempClass.lpfnWndProc != &InitWndProc)) {
9           if (ClassRegistered)
10              UnregisterClass(Params.WinClassName, Params.WindowClass.hInstance);
11          Params.WindowClass.lpfnWndProc = &InitWndProc;
12          Params.WindowClass.lpszClassName = Params.WinClassName;
13          if (RegisterClass(Params.WindowClass) == 0)
14              RaiseLastOSError();
15      };
16      CreationControl = this;                          //CreationControl 全局变量
17      CreateWindowHandle(Params);
18      if (m_handle == 0)
19          RaiseLastOSError();
20      ...
21  }
```

(e)

```
1   void TButton::CreateParams(TCreateParams * Params) {
2       TWinControl::CreateParams(&Params);
3       CreateSubClass(&Params, 'BUTTON');
4       Params.Style = Params.Style | ButtonStyles[m_default];
5   }
```

基于调试的代码阅读技巧 4：用跟踪定位虚函数调用

面向对象大量运用虚函数，而静态阅读几乎无法知道真正调用了什么代码，所以用跟踪是找出真实被调函数的有效方法。

DM7-27(e)的精妙之处是用第 2 行构造出许多控件 Params 中相同的部分。比如，CreateWindow 第 3 个参数，Edit 和 Button 均包含 WS_CHILD | WS_VISIBLE。这正对应 Params 参数的 Style 成员，我们只需在第 4 行将 Style 成员加上自己需要的部分。第 3 行传递“BUTTON”字符串，通过

CreateSubClass 调用系统 API——GetClassInfo，获得与按钮相关的窗体类 WndClass。从 DM7-27(e)我们看到，VCL 尽量提炼相同部分放入父类，把个性代码放到子类的虚函数。DM7-27(d)和 DM7-27(e)就是典型的模板方法模式（见 5.2.5 节）。

为了验证我们的理解，来看 TEdit 类是否覆盖了 CreateParams。TEdit 继承于 TCustomEdit 类，果然在 TCustomEdit 中覆盖了 CreateParams，其实现见 DM7-27(f)。其中，第 3～7 行实现了 TCustomEdit 的特殊逻辑。DM7-27(e)和 DM7-27(f)的第 2～3 行似乎可以融合成一个函数，重用性更高。为什么 VCL 不这样做？（**不要轻易否定他人**。）查看窗体类 TCustomForm 类也许就明白了。该类的确覆盖了虚函数 CreateParams，但其逻辑在开始就与另外两个类不同，见 DM7-27(g)。它一开始调用了 TScrollingWinControl 的 CreateParams，而不是调用 TWinControl，接下来也没有调用 CreateSubClass。所以，VCL 没有将 DM7-27(e)和 DM7-27(f)的第 2～3 行提炼成一个函数。

DM7-27

(f)

```
1   void TCustomEdit::CreateParams(TCreateParams* Params) {
2       TWinControl::CreateParams(Params);
3       CreateSubClass(Params, 'EDIT');
4       Params.Style = Params.Style | ... ;
5       if (NewStyleControls && Ctl3D ···) {
6           ...
7       }
8   }
```

(g)

```
void TCustomForm::CreateParams(TCreateParams* Params) {
    TScrollingWinControl::CreateParams(Params);
    InitAlphaBlending(Params);
    ...
}
```

我们总结一下在处理构造不同窗体控件的差异时，VCL 是如何实现重用与扩展的。

通过 TWinControl 封装了所有窗体控件的基础逻辑。将 lazy load 策略实现到其中，所有从它继承的子类，通过调用其 SetParent()函数自然共享这一策略。创建窗体控件的句柄时采用了模板方法设计模式，该方法为 TWinControl::CreateWnd()，见 DM7-28。其中，第 2 行为虚函数，不同的 TWinControl 子类覆盖该方法，其余代码均非虚函数。这样，不同的子类只需覆盖 CreateParams 即可重用整个窗体控件构造逻辑。执行 DM7-28 第 2 行时，创建的对象是哪个类，则执行该类的 CreateParams()方法，从而既重用了基类 TWincontrol 的其他逻辑，又针对不同类实现了其差异化功能。简略梳理控件显示的过程如图 7.30 所示。

DM7-28

```
1   void TWinControl::CreateWnd() {
2       CreateParams(&Params);
3       ...
4       CreateWindowHandle(Params);
5       ...
6   }
```

创建句柄的过程还存在一些不明之处，不理解 DM7-27(d)的第 6～16 行的复杂过程为了什么。如果是我们做，几乎可以直接省略。比照 7.4.1 节分析的自定位代码原理，我们发现，这里还有一道鸿沟：前面提到，在 TWinControl 构造函数中，通过调用 MakeObjectInstance 生成了自定位代码，但之后没见如何将该代码设定给控件的。在 7.4.1 节，为了直指问题关键，通过在 SetWindowLong 函数入口处设置断点，发现了一段代码通过调用 SetWindowLong 将自定位代码地址设定给控件，但并未分析这段代码如何才被调用。这段代码就是 InitWndProc()函数，见 DM7-29。

下面来分析和实证 InitWndProc 被调用的过程。

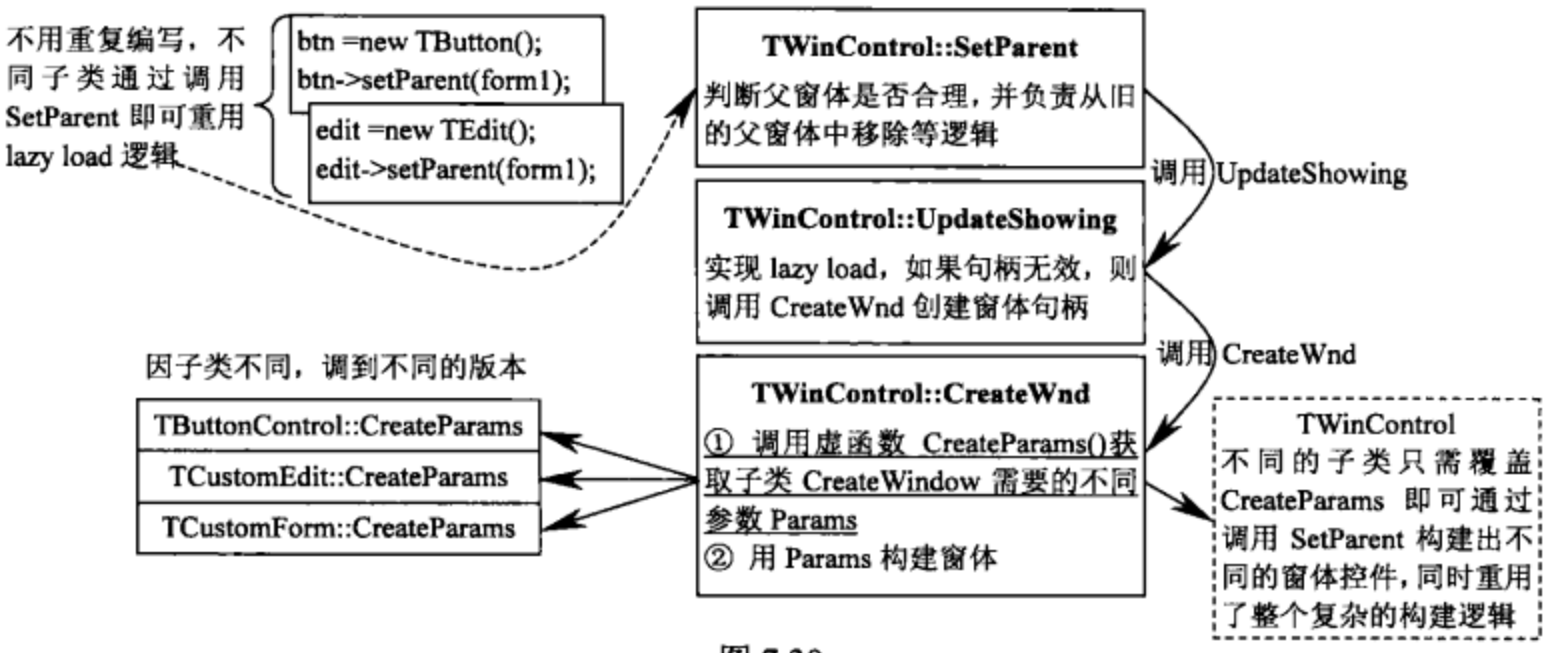


图 7.30

DM7-29

```
1    long InitWndProc(HWnd HWindow, LONG Message, LONG WParam, ... ){     // 接收第一个消息的消息处理函数
2        CreationControl.FHandle = HWindow;
3        SetWindowLong(HWindow, GWL_WNDPROC,              //设定自定位代码为消息处理函数
4        Longint(CreationControl.FObjectInstance));       //FObjectInstance 为构造函数创建的自定位代码入口
5        ...
6        _asm {      //调用自定位代码完成消息处理，因第 2 行设定的新消息处理函数在本次调用不会执行，需主动调用
7            push    LParam                               //传递 lparam
8            push    WParam                               //传递 wparam
9            push    Message                              //传递 Message
10           push    HWindow                              //传递窗体句柄
11           mov     eax, CreationControl                 //传递 this
12           mov     CreationControl,0                    //将全局变量清零
13           call    [eax], TWinControl.FObjectInstance   //通过自定位代码调用消息处理函数
14       }
15   }
```

DM7-27(d)的 17 行 CreateWindowHandle 中调用了 CreateWindowEx，按理该函数执行后，窗体控件消息处理函数就要被修改为自定位代码，也就是要执行 InitWndProc。但整个 DM7-27(d)没有看到直接调用 InitWndProc 的代码，第 11 行似乎引用了 InitWndProc。大致阅读这段代码，它实际是通过第 13 行注册一个 WndClass 结构体 Params.WindowClass，然后用该注册结构体创建控件。第 11 行就是将该结构体的消息处理函数成员 lpfnWndProc 设定为 InitWndProc。这是否意味着，可能是 CreateWindowEx 调用时，因为控件需要处理消息，所以就被系统直接调用了注册信息中的消息处理函数 InitWndProc？这是合理的解释，当该函数被调用，才通过其中的 SetWindowLong 修改控件的真正消息处理函数。

接下来需要证明这个猜测。思路如下：在 DM7-27(d)的第 17、18 行分别设置断点，并在 InitWndProc 的入口处设置断点。在第 17 行中断后，单步用 step over 执行，看是否在 InitWndProc 的断点处停止。如果在 InitWndProc 断点停止前，没在第 18 行的断点处停止，必然说明第 17 行函数调用的执行序列中直接由 CreateWindow 调用了 InitWndProc，否则会先在第 18 行断点处停止。(**这种情况可能发生在消息处理中，会先在第 18 行断点处停止，当函数返回并执行下一轮消息处理时调用** InitWndProc。) 当然，这里还有其他办法证明是 CreateWindowEx 调用了 InitWndProc。比如，在 InitWndProc 入口处设置断点，然后通过反汇编观察此时栈顶的返回值（在笔者机器上是 0x77d18724），再查看该值是否属于 user32.dll 的代码段（之前我们做过类似工作，user32.dll 在笔者

机器上基址为 0x77d10000）。事实证明，确实是 CreateWindowEx 调用了 WndClass 结构体中注册的消息处理函数，也就是 InitWndProc。

将自定位代码设置为控件消息处理函数的过程如图 7.31 所示。

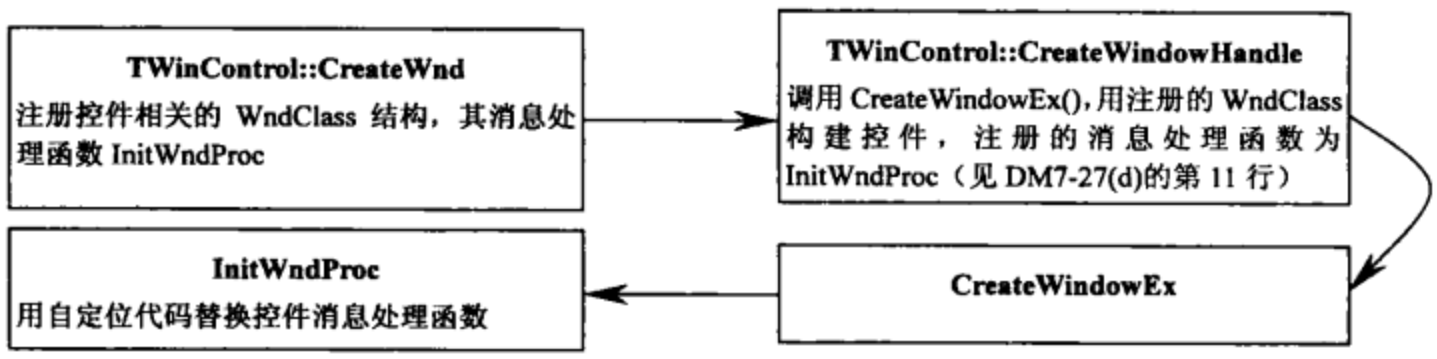


图 7.31

CreateWindowEx 为什么要调用消息处理函数 InitWndProc？合理的猜测是发送创建控件相关的消息。帮助中提及，会发送 WM_CREATE 消息。依然在 InitWndProc 入口处设置断点，查看其参数 Message 的值即可，实证该值为 129，从消息定义头文件得知，该值为 WM_NCCREATE，并非 WM_CREATE。再查看 WM_NCCREATE 说明，发现它是在 WM_CREATE 发送前发送的。为什么没有收到 WM_CREATE？因为 InitWndProc 执行时，就注册了自定位代码，替换自己作为新的消息处理函数，那么 WM_CREATE 消息被新消息处理函数接收。为了证明该猜测，我们只需在 CreateWindowEx 前后设置两个断点，并在 InitWndProc 入口处设置断点。调用 CreateWindowEx 前的断点中断时，再在真正的消息处理函数 StdWndProc 处设置断点（函数见图 7.4，这样做可避免因其他消息反复进入 StdWndProc）。只要在 CreateWindowEx 执行后，其后断点中断前，StdWndProc 被调用，说明 InitWndProc 成功将消息处理函数替换成 StdWndProc。在 StdWndProc 的 Message 参数中查看其值是否有 WM_CREATE 即可。通过实验发现，在 CreateWindowEx 返回前，StdWndProc 连续被调用，按顺序收到了以下消息：

WM_NCCREATE → WM_NCCALCSIZE → WM_CREATE → WM_SIZE
→ WM_MOVE → WM_PARENTNOTIFY

可见，第 3 个收到的消息是 WM_CREATE。

VCL 消息处理函数替换过程明显不同于我们的版本，我们是先用 CreateWindow 构建控件，再自己调用 SetWindowLong，修改控件的消息处理函数。VCL 看来似乎复杂些，先将所有控件的消息处理函数都定义成 InitWndProc，再依赖调用 CreateWindowEx 时，该函数自动调用 InitWndProc 来修改控件消息处理函数。

通过上面的实验，我们已经窥探到 VCL 这样做的端倪：让控件对象能够接收到创建控件时的相关消息（上面的实验共有 7 个消息）。CreateWindowEx 首先调用 InitWndProc，通过 DM7-29 的第 7～13 行调用自定位消息处理函数。此时，自定位代码还没有注册为控件消息函数，不会被系统调用，为了第一个消息（WM_NCCREATE）依然能被客户代码处理（通过 StdWndProc 派发给客户代码），所以主动调用自定位代码。之后，因 InitWndProc 将消息处理函数设为自定位代码，那么后续消息处理就会调用到 StdWndProc。这样在 CreateWindowEx 期间的所有消息一个不剩地被消息处理函数 StdWndProc 处理。（**精妙的设计！**）以上分析见图 7.32。

反观我们的做法，CreateWindow 返回后才调用 SetWindowLong 改变消息处理函数，自然 CreateWindow 期间的所有消息不会被真正的消息函数（即其后被 SetWindowLong 设定的自定位代码）处理，而是被默认处理函数处理。

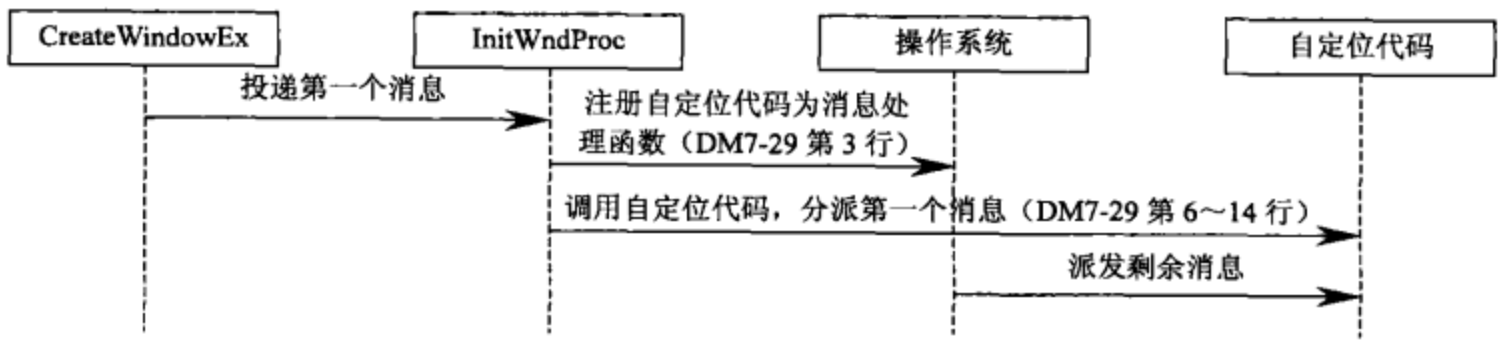


图 7.32

现在还有一点不明，DM7-27(d)的第 7～10 行为什么要做那么复杂判断再注册窗体类呢？我们的实现就是简单地注册了事。这个问题还得靠调构学习法第五步来解决（见 7.7 节）。因为按 VCL 的思路实际构建时出现了问题，在解决该问题时，对此段代码有了更深刻认知。**（实践是吾师！）**

回到本节开始的第二个问题，我们的版本每个控件类的析构函数重复很多。来看 VCL 如何处理析构函数。只需在前面构造组件对象的实验代码后加入释放对象的代码，即可用 step into 跟踪进析构函数查看。我们发现，TButton 和 TEdit 等组件均将释放工作交给了其共同基类 TWinControl。构建工作交给了基类 TWinControl，释放也交给它，看来 VCL 很好地解决了重用问题。TWinControl 的析构函数见 DM7-30。第 5～6 行的含义是：如果窗体句柄有效，就删除窗体句柄。再细查 DestroyWindowHandle 的内容，发现它调用了 DestroyWindow 函数。第 8～9 行则将自定位代码释放。

DM7-30

```
1   TWinControl::~TWinControl() {
2       ...
3       if (m_parent != NULL)
4           RemoveFocus(true);
5       if (m_handle != 0)
6           DestroyWindowHandle;
7       ...
8       if (FObjectInstance != NULL)
9        Classes.FreeObjectInstance(FObjectInstance);
10  }
```

分析 FreeObjectInstance，发现它并没有释放内存本身，只是将自定位代码所占内存块再次挂到可用链。DestroyWindowHandle 是一个虚函数。为什么？自然的想法是，不同组件在释放时，可能销毁句柄的方式会不同。在源文件中查找该函数名，发现 TForm 类覆盖了该函数，见 DM7-31。因为 TForm 类封装了多文档窗体和单窗体逻辑，所以在析构时如果是多文档窗体，则第 2～3 行将发送消息，否则调用基类的 DestroyWindowHandle。这里再次让我们看到了 5.4.2 节中的痛苦：对于 C++，在其构造和析构中无法实现多态，自然无法用模板方法模式（在这里 Delphi 可以）。事实上，不同子类在析构函数中销毁资源的策略可能有所不同。

DM7-31

```
1   void TCustomForm::DestroyWindowHandle(){
2       if (fsCreatedMDIChild in FFormState)
3           SendMessage(Application.MainForm.ClientHandle…
4       else
5           TWinControl::DestroyWindowHandle();
6       FClientHandle = 0;
7   }
```

最后，我们来看本节开始的第三个问题：重复的消息处理函数怎样重用？其实在 7.4.1 节已经分析过消息处理的调用序列，这里给出鼠标左键弹起消息处理经过的函数调用：

自定位代码 → StdWndProc → TWinControl::MainWndProc → TWinControl::WndProc
→ TControl::WndProc → TControl::WMLeftButtonUp → … → TControl::MouseUp
→ 通知实际的事件响应函数

7.4.1 节指出该响应函数是 Delphi 特有的由 this 和函数地址共 8 字节组成的对象版函数指针。

注意，为什么 StdWndProc 要先调用 TWinControl 的 MainWndProc 再由 MainWndProc 调用 TWincontrol 的 WndProc？MainWndProc 和 WndProc 都是 TWinControl 的函数，何必那么麻烦？首先，不同类型组件有自己特有的消息处理逻辑，如我们实现的窗体类就要处理 WM_DESTROY，这

意味着同样的消息处理过程，在不同类型控件执行时要执行到不同代码。这显然是虚函数的特点。StdWndProc 调用消息处理函数时，是从 ECX 中指向的内存中获取函数地址（对应图 7.4 中的 call [ecx]）。如果 ECX 中存储的函数地址不变，则总会调用到同一函数。该函数地址是在 TWinControl 的构造函数中设定的，总被设定为 MainWndProc（见 DM7-15）。这意味着，StdWndProc 只能调用到 TWinControl::MainWndProc，自然无法调到 TWinControl 子类们的消息处理代码。所以，为了达成消息处理的多态化，我们必须在 MainWndProc 中调用虚函数。

来看 MainWndProc 和它调用的 WndProc 的函数类型：MainWndProc 为非虚函数，WndProc 是 TWinControl 基类 TControl 声明的虚函数，完全符合我们的推断。需要注意的是，MainWndProc 函数不仅起到了调用虚函数使消息处理多态化的效果，而且利用 try...catch 结构，保证所有消息处理中，如果出错均会释放相关资源。这又是一个模板方法模式，让客户代码撰写更容易，客户代码不用关注错误处理，系统库处理了相关工作，见下面的 C++模拟代码。

```
try {
    WndProc(Message);
}
catch(…) {
    Application->HandleException(this);
}
```

上面的消息处理序列中存在如下调用：TControl::WndProc → TControl::WMLeftButtonUp，是靠 Delphi 特有的语言机制（Dynamic 方法，一种类似 virtual 的方法）完成的，这里就不探讨。有兴趣的读者可以用反汇编分析 TControl::WndProc()，其最后一行 Dispatch 函数是关键，将消息值 message 和 dynamic 的消息处理函数的 ID 巧妙对应。在调用客户撰写的消息回调函数这个问题上，VCL 的机制本质等同于我们的 listener 模式。

从消息处理的重用和扩展我们发现，VCL 思路有 3 条：① 在基类中定义虚函数的消息处理函数（WndProc），这样当我们继承定义自己子类控件时，可以覆盖它处理子类特有逻辑（当然最后要调用基类的 WndProc 重用基类处理逻辑）；② 在基类中定义一些基本消息的事件回调，这样非子类的其他类也可以获得处理消息的能力，使得灵活性极大提高，这等同于我们的 listener；③ 其实是①的优化，子类可声明特有的 dynamic 消息处理方法获得消息处理能力，否则在①的方式下必须有很多 if...else 判断组合。

我们的版本只需要学习其①和②即可，先将消息处理函数定义成可覆盖的虚函数，放入控件基类中。（目前我们的版本是每个控件类重复定义消息处理函数，复制，然后粘贴。）其次将每个控件类的消息处理 listener 提炼成一个基类，并将通知 listener 的代码也放入控件基类，这样所有控件子类不用重复书写就直接拥有消息事件通知能力。

目前的分析只是一个起点，真正的航程需要大家扬帆。框架因为要处理极其繁复的细节，往往有着非常复杂的细节代码。而撑起其框架重用扩展的基本骨架确实简单直接。有些细节必须在自己去构建相似框架时才能领悟。如何在细节的繁复与骨架的简洁间游历？不断阅读、思考、调试、验证应该是通向我们天堂的阶梯，阶梯上路过的繁复与简洁都是风景。

7.7 构建我的 GUI 组件（4）：我的天堂

上面我们领略了优雅的 VCL，针对自己存在的问题专门考察了 VCL 相关处理。是到了修改我

们的版本建立自己天堂的时候了。

首先，为不同窗体控件定义共同基类 Control。将自定义代码构建逻辑封装到该类构造函数中，并模仿 lazy load，在 show 函数中才真正调用 CreateWindow 构建组件。在析构函数中，释放窗体控件句柄和自定位代码内存。这样所有子类，如 Button、Edit 等，均不用再书写任何这方面的代码，全面重用基类 Control 的。

其次，消息处理方面，模仿 VCL 在基类中定义 WndProc()虚函数，使得子类可以有自己的消息处理逻辑。将 listener 通知和注册代码放入基类 Control。这样原来 Edit、Button 等类中的重复代码不复存在。而替换消息处理函数为自定位代码的过程模仿 VCL。

另外，将一些重用代码也移入到 Control 类，如 setCaption、getCaption，这样 Form、Edit 和 Button 不用再自己专门定义 setCaption()等函数。

最后，让 Form 等类从 Control 继承，并删除这些子类中移入到 Control 类的代码，只保留与自己特殊逻辑相关的代码。

下面就几个关键点加以阐述，工程见“code\第 7 章\button_obj_stubcode_good”。

先来看析构函数，Button 和 Edit 逻辑都是相同的，问题出在 Form，见 DM7-22(b)，开始的 3 行判断窗体到底是通过直接调用析构销毁还是通过鼠标关闭窗体销毁。这个逻辑是其他 Control 子类没有的，而我们希望 Control 子类在重用父类的析构逻辑的同时可加入自己的逻辑。但 C++的析构函数中调用虚函数没有多态，因此我们借鉴 5.4.3 节中将虚函数从原先的类分离到一个策略者对象的做法，定义 CtrlDestructor 类，它有两个虚函数 tryDestroyHandle()和 afterDestroy()。该类的对象在 Control 析构中被调用，不同的 Control 子类如果需要自己的析构逻辑，则定义 CtrlDestructor 的子类并覆盖两个虚函数，见 DM7-32(a)。

DM7-32(b)是一个典型的策略者模式，在第 5 行和第 11 行无法确定代码逻辑，所以调用注册的 m_ctrlDtor 的虚函数。

Control 构造函数中注册了默认的 CtrlDestructor 的实例，见 DM7-32(c)。

Form 的构造函数见 DM7-32(d)，第 1 行先删除基类注册的 m_ctrlDtor，然后在第 3 行注册自己的处理策略对象，FormDestructor 的实例。

FormDestructor 的 tryDestroyHandle 的实现是将 DM7-22(b)开始的 3 行稍修改（通过 m_ctrl 访问）后的等价代码，见 DM7-32(e)。其 afterDestory()则类似将之前版本的 Form 析构函数的代码迁入，如果是主窗体，则发送退出消息，见 DM7-32(f)。当生成 Form 对象时，在构造函数中注册 FormDestructor 对象，此时 m_ctrlDtor 是 FormDestructor 的实例。Control 的析构执行时，DM7-23(b)的第 5 行调用 m_ctrlDtor 的 tryDestroyHandle，调用到的实际是 FormDestructor 的版本。第 11 行中，

DM7-32

(a)

```
class CtrlDestructor {
  public:
    CtrlDestructor(Control * ctrl) {
        m_ctrl = ctrl;
    }
    ~CtrlDestructor() { }
    virtual void tryDestroyHandle();
    virtual void afterDestroy();
  protected:
    Control * m_ctrl;
};
```

(b)

```
1  Control::~Control() {
2      if (m_caption != NULL)
3          free(m_caption);
4      if (m_handle != 0){
5          m_ctrlDtor->tryDestroyHandle();
6          free(m_wndProcStub);
7          SetWindowLong(m_handle, …);
8      }
9      if (m_listener != NULL)
10         delete m_listener;
11     m_ctrlDtor->afterDestroy();
12     delete m_ctrlDtor;
13 }
```

(c)

```
Control::Control(int x, int y , …) {
    m_ctrlDtor = new CtrlDestructor(this);
    …
}
```

DM7-32（续）

(d)

```
1  Form(bool isMainForm,char* caption) : Control(...){
2      delete m_ctrlDtor;
3      m_ctrlDtor = new FormDestructor(this);
4      m_isMainForm = isMainForm;
5      m_destroyState = dsNo;
6  }
```

DM7-32（续）

(e)

```
1  void FormDestructor::tryDestroyHandle() {
2      if (((Form *)m_ctrl)->m_destroyState == dsNo) {
3          ((Form *)m_ctrl)->m_destroyState = dsFree;
4          DestroyWindow(((Form *)m_ctrl)->getHandle());
5      }
6  }
```

(f)

```
void FormDestructor::afterDestroy() {
    if (((Form *)m_ctrl)->m_isMainForm)
        PostQuitMessage(0);
}
```

m_ctrlDtor 的 afterDestroy() 调到的也是 FormDestructor 的版本。这样，析构时，Form 既重用了 Control 的所有逻辑，又在需要处执行了自己的逻辑，满足了扩展性。根据 VCL，TForm 类并没有在析构时进行 DM7-32(e)那样的逻辑，因为它是收到 WM_CLOSE 消息时尝试释放 TForm 对象，不是在接收 WM_DESTROY，从而避免了 DM7-32(e)的逻辑。因而也可暂不要 CtrlDestructor 这样的设计。在框架中寻找我们不会的功能和难题的答案是一捷径。

构造函数没有特殊部分，与 VCL 不同的是，我们没有在构造中分配自定位代码内存，而是在需要显示时才分配的。

关于自定位代码生成和消息处理函数调用的关键代码，见 DM7-33。其中，DM7-33(a)判断 m_handle 是否为 0，如果是，则构造自定位代码并调用 CreateWindow()构建实际控件。该函数通过在 show 中调用 changPosition()而最终被调用，从而实现 lazy load。

DM7-33(a)

```
1   void Control::needHandle() {
2       WndCreateParams params;
3       if (m_handle == 0) {
4           LRESULT (__thiscall Control::* ptrFunc)(HWND,UINT,WPARAM,LPARAM);
5           ptrFunc = (&Control::WndWrapper);                    //将自定位代码跳往的函数设定为 Control::WndWrapper
6           void * func;
7           __asm {
8               mov eax, ptrFunc;
9               mov func, eax;
10          }
11          m_wndProcStub = makeWndProcStub(this, func);        //构建自定位代码
            //消息处理函数第一次被调时，将调 initWndProc，它必须通过全局变量 gCtrlNoFinished 访问对应组件对象
12          gCtrlNoFinished = this;
13          createParams(&params);                              //通过调用虚函数 createParams，不同子类具有自己的构造参数
14          m_oldWndProc = params.wndClass.lpfnWndProc;         //保存老的处理函数，析构时要设置回去。
15          WNDCLASSA tmpClass;
16          bool registered = GetClassInfoA(applicationHandle, params.className, &tmpClass);
17          if (!registered || params.wndClass.lpfnWndProc != initWndProc){ //重新用 RegisterClassA 注册保证非 unicode 版
18              if (registered)
19                  UnregisterClassA(params.className, applicationHandle);
20              params.wndClass.lpfnWndProc = initWndProc;
21              params.wndClass.lpszClassName = params.className;
22              RegisterClassA(&params.wndClass);
23          }
24          m_handle = CreateWindowA(params.wndClass.lpszClassName, ...);
25      }
26  }
```

DM7-33(a)第 5 行将自定位代码转跳的函数设定为 Control::WndWrapper，而该函数运行时调用虚函数 WndProc，从而实现 Control 子类消息处理的多态化，见 DM7-33(b)。Control::WndProc 通过调用 listener 的消息处理虚函数实现了一种更灵活的消息处理扩展，见 DM7-33(b)。

下面进一步分析自定位代码是如何最终注册给控件的。从 7.6 节的分析知道，在调用 CreateWindowEx 时，该函数将自动多次调用消息处理函数处理多个消息。因此，VCL 的方式可让控件接收到所有消息，即先通过注册的 WndClass 结构使得第一次消息处理指向的函数为 initWndProc()，见 DM7-33(a)的第 20 行。在 initWndProc 最终被 CreateWindowEx 调用时，initWndProc

将控件消息处理函数设定为自定位代码，见 DM7-33(c)的第 2 行。因为 initWndProc 非自定位代码，也不可能是自定位代码（之所以有自定位代码是要将 this 和指令放在一起，而 initWndProc 作为整个控件类的消息处理函数，基本无法和某个对象 this 绑定在一起。）,所以必须通过全局变量 gCtrlNoFinished 传递当前控件对象的 this（这一点和 VCL 相同，DM7-29 的第 2 和第 4 行中的 CreationControl 就是全局变量，作用等同于 gCtrlNoFinished）。DM7-33(a)的第 12 行为该传递完成了设置工作。前面通过调试我们知道，initWndProc 是被 CreateWindowEx()调用的，所以执行 initWndProc 时，CreateWindowEx 还没返回，那么对象的成员变量 m_handle 就没有被设置（DM7-33(a)第 24 行）。initWndProc 的第 10 行调用了对象的消息处理函数，该函数有可能使用 m_handle，所以必须设定 m_handle，这就有了 DM7-33(c)的第 1 行（等价于 DM7-29 的第 2 行）。框架的每句话都是精巧和慎密的。第 3～10 行完成对消息处理函数调用，其目的与 VCL（DM7-29）一样，为了第一个消息也被处理。

DM7-33

(b)

```
LRESULT Control::WndWrapper(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    return WndProc(hwnd, message, wParam, lParam);
}
LRESULT Control::WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    LRESULT result = CallWindowProcA(m_oldWndProc, hwnd, message, wParam, lParam);
    if (m_listener != NULL) {
        switch (message) {
            case WM_LBUTTONUP:
                m_listener->onLeftButtonUp(this);
                break;
            }
        }
        return result;
    }
}
```

(c)

```
LRESULT CALLBACK Control::initWndProc(HWND hwnd, UINT message, …) {
//此时 createwindow 产生 wm_create 并调到该函数，所以 createwindow 未返回，因此 m_handle 没有设定值。为了
//保证第 10 行调用的消息处理函数能使用成员变量 m_handle，所以设定
    gCtrlNoFinished->m_handle = hwnd;
    SetWindowLongA(hwnd, GWL_WNDPROC, (LONG)gCtrlNoFinished->m_wndProcStub);
    void * func = gCtrlNoFinished->m_wndProcStub;
    __asm {                                          //以下 4 个 push 对应传递消息处理函数的参数，从右向左压栈传参
        push    lParam;
        push    wParam;
        push    message;
        push    hwnd;
        mov     ecx, gCtrlNoFinished;                //设定 this 指针，VC 用 ecx 传递
)       call    func;          //调用消息处理函数，最后不清栈，因为被调函数返回采用了 ret  xxx 指令自己清栈
    }
 }
```

现在回头仔细分析 DM7-33(a)的逻辑。当第 3 行判断应该进行窗体创建后，第 4～11 行构建了自定位代码。第 12 行为了配合 initWndProc 访问控件对象，将 this 赋值给了全局变量 gCtrlNoFinished。这导致该版本多线程不安全，也是 VCL 中要求控件必须由一个线程创建的因素之一。我们可通过对 DM7-33(a)的第 4～24 行代码加锁解决该问题。即使加锁，也只涉及控件构造时，这与 7.3 节每次消息函数调用都要通过锁访问全局句柄-对象指针表不可同日而语。如果你连这一点点锁的运用都不能接受，采用线程局部变量替代 gCtrlNoFinished 即可。

为了构建不同的控件，必须为 CreateWindow 提供不同的参数，所以 DM7-33(a)的第 13 行调用虚函数 createParams()，这样不同子类的覆盖版本将被调用。该函数会按照子类需要设置 params 参数，以供第 24 行真正创建控件使用。第 14 行保留老的消息处理函数，在析构函数中为了自定位代码释放后能正常处理消息，老处理函数将会被设定给控件（见 DM7-32(b)第 7 行）。第 16～23 行注册了被创建控件对应的窗体类 WndClass，特别将 initWndProc 设定为控件的消息处理函数（第 20 行）。第 24 行创建控件。

乱码解决

第 16～19 行似乎有点多余，直接注册即可。问题发生在实现 Control::getCaption 的时候。DM7-33(b)的 Control::WndProc()函数中调用了非 Unicode 版本的 CallWindowProcA（对应 Unicode 版本是 CallWindowProcW）来处理消息，这时获得的字符串是乱码。如果在消息处理函数中设置断点，还可看到传入的就是 Unicode 字符串（0x31 变成了 0x00 0x31，结束也由两个 0x00 代表）。

通过查资料发现，Windows 默认使用 Unicode 版本，我们需要用 RegisterClassA 注册窗体类。可是当我们尝试这样做时，一点效果都没有。分析一下，这也正常：像标准控件，如编辑框控件，本来就已经注册了，因此操作系统可能因为你注册的窗体类已经注册而根本就没有再次注册，所以无效。我们尝试先 UnregisterClass 注销，事实是，如果消息处理函数和注销前窗体类的消息处理函数一样，即使再注册，也没有效果。这时我们终于明白了，VCL 在 DM7-27(d)中的第 7～10 行的含义：它们被迁移到我们的版本即 DM7-33(a)的第 16～19 行。为了使用非 Unicode 版本，我们必须用 RegisterClassA 注册窗体类，见 DM7-33(a)的第 22 行。在此之前，必须有效注销老类。什么是有效注销呢？如果窗体类的消息处理函数不是我们的 initWndProc，则应该注销，这就有了第 16～17 行的判断。这段经历告诉我们，实践是吾师，有些问题在阅读甚至调试后都难明了，就必须由实践构建阶段来解决。在这个阶段，无法事先预料自己将遇到什么问题，而成功解决这些问题，往往能让我们看清那些辉煌的框架后隐藏的微妙。这也是笔者为什么一直秉承“猜测—实证—构建”作为自学求知之道的原因。

7.8 天堂一瞥

我们的天堂是构建在我们看到的仙境之上。VCL 就是那唯一的仙境吗？徜徉在自己的天堂，应该向窗外遥望，那些云与苍穹背后还有美丽仙境。这也是求学之道，没有谁来告诉你，来教你什么是优雅，我们应该在一个个经过的优雅中去品味和辨别，去思考和比较。见识宽广了，境界自然就高了。因此，我们应该去看看其他窗体控件框架是怎么做的。有了之前构建的基础，这时的阅读学习不再是一行行按部就班死读，而是从前面关注的问题为出发点，直奔主题。

前面最关注的一个问题就是：消息处理函数不传递对象 this。7.3 节采用的全局句柄-对象指针表存在缺陷，VCL 采用了自定位代码。其他框架怎样处理？第一步是看有哪些框架存在，容易查找到 MFC、WTL、QT 等。这里仅以 MFC 为例展现学习过程，其他请大家各自分析。

首先，我们依然是构建一个简单的 MFC 应用，然后模仿调试 VCL，随便找一个必然是因处理消息而被调用的函数，如按钮点击响应函数。在该函数中设置断点，运行中断后，就可在 Visual C++ 中通过调用堆栈查看，找到操作系统回调的那个函数即可。

调用堆栈如图 7.33 所示，最后一行 user32.dll 表明了这次函数调用是 user32 中发出的。从这里往上查看，点击到 AfxWndProc 项时（见 DM7-34），我们发现它是消息函数。直奔主题，快速忽略掉其他，发现了第 5 行，函数 CWnd::FromHandlePermanent()传入的是窗体句柄，返回的却是一

图 7.33

DM7-34

```
1   LRESULT CALLBACK AfxWndProc(HWND hWnd,
2   UINT nMsg, WPARAM wParam, LPARAM lParam) {
3       if (nMsg == WM_QUERYAFXWNDPROC)
4           return 1;
5       CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
6       ...
7       return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}
```

个 CWnd *，这就是对象的指针。没有前面的构建和疑问，我们很难这样犀利地直指命门。几乎可推定，这个函数执行了与 7.3 节类似的逻辑，用句柄查找对象 this。查看该函数，果如预料，见 DM7-35。

DM7-35

```
1   CWnd* PASCAL CWnd::FromHandlePermanent(HWND hWnd) {
2       CHandleMap* pMap = afxMapHWND();
3       CWnd* pWnd = NULL;
4       if (pMap != NULL) {
5           pWnd = (CWnd*)pMap->LookupPermanent(hWnd);
6           ASSERT(pWnd == NULL || pWnd->m_hWnd == hWnd);
7       }
8       return pWnd;
9   }
```

从 DM7-35 的第 5 行看到，通过调用 map 对象的 LookupPermanent，用句柄 hWnd 获取对应对象的指针 pWnd。再查看该 map 的类定义，即 CHandleMap 定义，就看到了函数 void SetPermanent(HANDLE h, CObject* permOb)的声明，可见应该是用这个函数设定了句柄和对象指针的对应关系，之后才在 DM7-35 中用 LookupPermanent 查找。此时我们知道 MFC 采用了 7.3 节相同的思路。

乘胜追击，我们几乎能推定，之后框架会用这个对象指针调用其虚函数版本的消息处理函数，以达到消息处理的多态。在图 7.33 中点击 AfxWndProc 上层的项，即它调用的函数 AfxCallWndProc。其代码见 DM7-36。其中通过句柄查询到对象指针 pWnd，第 3 行用它调用了函数 WindowProc()。完成了对象版非静态消息处理函数的调用。通过查看 WindowProc()定义，发现它果然是虚函数。

至此，我们的版本、VCL 和 MFC 在消息处理上有了融合点：通过调用父类定义的消息处理虚函数让子类有自主处理消息的能力。

限于篇幅，本节就不再深入讨论。我们应该将这一学习过程深入到以后的源代码学习中。

DM7-36

```
1   LRESULT AFXAPI AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,WPARAM wParam = 0, LPARAM lParam = 0) {
2       ...
3       lResult = pWnd->WindowProc(nMsg, wParam, lParam);
4       ...
5   }
```

更多扩展

1．Charles Petzold. Windows 程序设计（第 5 版）. 清华大学出版社, 2010.

习 题 7

1．请用反汇编跟踪证实 Delphi 默认方式下，用 EAX 传递 this 指针。

2．请修改源代码，让消息处理函数调用惯例符合系统要求。

3．请采用线程局部变量解决 DM7-33(a)和 DM7-33(c)中对全局变量 gCtrlNoFinished 的依赖。

4．请分析 QT、WTL 中如何解决在消息处理函数中获取对应控件对象指针的问题。

5．请分析 MFC、QT、WTL 中在构造窗体控件（即调用 CreateWindow）的时机上是否采用了 lazy load。

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

[General Information]
书名=老“码”识途　　从机器码到框架的系统观逆向修炼之路
作者=韩宏，李林著
页码=332
ISBN=332
SS号=13070007
dxNumber=0000083549 34
出版时间=2012.08
出版社=该引擎未能查询到
定价：56.00
试读地址=http://book.duxiu.com/bookDetail.jsp?dxNumber=000008354934&d=190AFFCCC87B06C6AFE78D8206E13DA8&fenlei=18170403010 1&sw=%C0%CF%A1%B0%C2%EB%A1%B1%CA%B6%CD%BE++%B4%D3%BB%FA%C6%F7%C2%EB%B5%BD%BF%F2%BC%DC%B5%C4%CF%B5%CD%B3%B9%DB%C4%E6%CF%F2%D0%DE%C1%B6%D6%AE%C2%B7
全文地址=http://dkj.5read.com/image/ss2jpg.dll?did=b13&pid=286C327DF092E2D0E9ED4618B1E89E6E60080D7BCD8C6EC7BA289535CF252A39883 6AC473F21623CD8 71D3119C9CAEB03DE9AE6F91CDC3B2D5052 53CB1AF882DDE9897C8089 9D6C2B97AD6DEFCC568A1644D68F54D8188E4CC83D29255F1037259603DB3BDCDCECFC0ED918609 4C6BAFE02D&jid=/