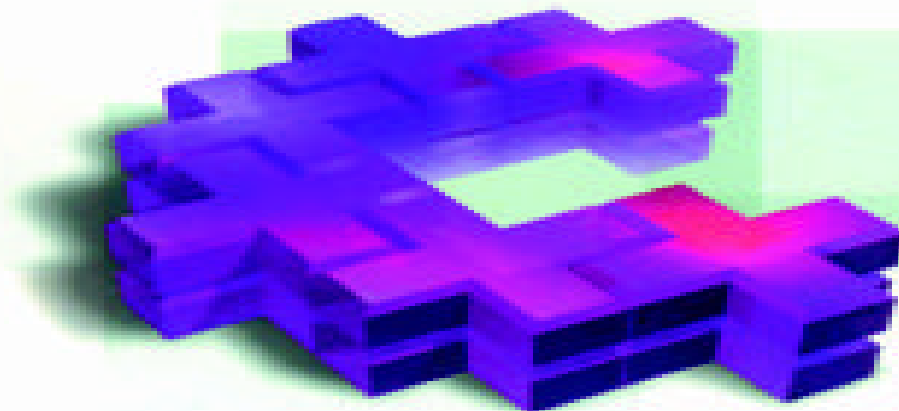




The Essential Guide to
Microsoft Visual C++ 6.0



Microsoft
Visual C++ 6.0
Programmer's Guide
程序员指南

 北京希望电脑公司

Microsoft Press

美国微软出版社授权的标准、权威、完整的中文图书

Microsoft Visual C++ 6.0

程序员指南

〔美〕 Beck Zaratian 著

希望图书创作室 译

希望图书创作室 审校

本书配套光盘包括两部分内容：

1. 本书英文原版附带光盘中的例子代码及可执行程序
2. 与本书配套的中文版电子书

北京希望电脑公司出品

1 9 9 8

内 容 提 要

本书是美国微软出版社授权的 Microsoft Visual Studio 系列中文版图书之一，它是 Visual C++ 6.0 程序员的实用参考书。全书共分成六个部分：第一部分介绍 Developer Studio 和 AppWizard 的基础知识；第二部分介绍文本、图形和对话框编辑器；第三部分是编程指南，展示了如何使用 ClassWizard 和 Gallery 来加速编程开发工作，创建自己的组件；第四部分介绍 ActiveX 控件及其使用，说明如何用 MFC 或 ATL 来写 ActiveX 控件；第五部分讨论调试器的各种能力，程序优化的各种选项及原因，如何定制 Visual C++，如何编制宏和附加实用程序；第六部分提供 ASCII 字符和 ANSI 字符的标准表格，并简要描述 ClassWizard 支持的 MFC 类和 VBScript。

书中讨论的许多主题均以范例程序进行说明，所有范例程序的项目文件都在配套光盘上。本书是从事 Visual C++ 6.0 应用与开发的所有技术人员极好的工具书，同时也是大专院校相关专业的师生的自学、教学参考用书。

本书配套光盘包括两部分内容：1. 本书英文原版附带光盘中的例子代码及可执行程序；2. 与本书配套的中文版电子书。

需要本书和配套光盘及需技术支持的读者请与北京海淀 8721 信箱书刊部联系，邮政编码：100080，联系电话：010-62562329，62541992，62531267，传真：010-62579874。

版 权 声 明

本书英文版名为《Microsoft Visual C++ 6.0 Programmer's Guide》，由 Microsoft 出版社出版，版权归 Microsoft 出版社所有。本书中文版由 Microsoft 出版社授权出版。未经出版者书面许可，本书的任何部分不得以任何形式或任何手段复制或传播。

本书封四上贴有防伪标签，无标签者不得销售，违者必究！

Microsoft Visual C++ 6.0 程序员指南

[美] Beck Zaratian 著

希望图书创作室 译

希望图书创作室 审校

责任编辑 刘晓融

北京希望电脑公司出品

北京海淀路 82 号 (100080)

印刷

新华书店、新华书店音像发行所发行、各地书店及软件专卖店经销

* * * * *

1998 年 12 月 第 1
版

1998 年 12 月 第 1
次 印 刷

开 本 : 787 × 1092

印 张 : 39.125

字 数 : 916 千 字

印 数 : 1-5000

新 出 音 管 [1998]164 号

ISBN 7-980021-23-1/TP · 16

定 价 : 68.00 元 (1CD, 含 配 套 书)

致 谢

第二版要比第一版容易写,因为在第二版中问题已解决了一半。我乐意写(或者更准确地说,是扩展)这本书的主要原因是,我又一次得到了具有献身精神的人们的帮助。尽管处理过较早版本的 Microsoft Press 小组已经转向出版书籍的其他任务,但他们的贡献仍在,他们认真负责的工作态度和敬业精神在本书留下了深深的烙印。

在第一版中,项目编辑 Lucinda Rowley 在百忙之中花费了许多时间,他审阅手稿具有敏锐的观察力(在书名为《Microsoft Visual C++ Owner's Manual》的版本中,我将 Lucinda 誉为“每位作家梦寐以求的编辑”,他的确名符其实)。手稿编辑 Vicky Thulman 推敲了每个句子,而技术编辑 Linda Ebenstein 和 Jim Johnson 确保所述内容的确切。

项目编辑 Saul Candib 和技术编辑 Jim Fuchs, Mary DeJong 和 Michael Hochberg 仔细审阅了新材料。旧金山的 Labrecque 出版社在编辑、版式、校对和制作管理等方面做了大量工作,为此谨向 Chrisa Hotchkiss、Curtis Philips、Lisa Bravo、Andrea Fox 和 Lisa Labrecque 表示谢意。

我不仅得到了来自 Press 和 Labrecque 的帮助,而且得到微软公司 42 楼 Visual C++ 分部的支持。如同以前一样, Laura Hamilton 充当了联络员的角色,我很感激,因为如果没有她的帮助,我将无法写出这本书来。Laura 是一名高级编辑,

她可以保证本书和在线《Visual C++ User's Guide》的质量。

这个版本离不开 Visual C++小组(以及其他领域)中许多人的帮助, 他们为第一版做了勘误, 提出了有价值的建议。其他人审阅了这一版的新材料, 集思广益确保本书的结构严谨和内容实用。在此谨向 Dennis Andersen、Cathy Anderson、Chuck Bell、Diane Berkeley、Patricia Cornette、Stacey Doerr、Chris Flaat、Jocelyn Garner、Anita George、Eric Gunnerson、Karl Hilsmann、Mark Hopkins、Simon Koeman、Chris Koziarz、Louis Lafreniere、Martin Lovell、Michael Maio、Bruce McKinney、Diane Melde、Daryn Robbins、Steve Ross、David Schwartz、Scott Semyan、Teeri Sharkey、George Shepherd、Kathy Shoesmith、Suzanne Sowinska、Yefim Sigal、Chuck Sphar、Yeong-Kan Tam、Donn Trenton 和 Laura Wall 表示谢意。这些人中的一部分是我认识多年的朋友, 其他则从未见过面, 仅仅通过电子邮件和联邦快递相互通信。

特别值得一提的是微软公司的 Barbara Ellsworth, 没有她, 本书将无法写成。感谢 Barb。

本书中文版的问世渗透了参与策划、翻译、录排、审校和出版人员的大量心血。本书由贺军、贺民、王雷、陈武、刘传凯、高胜友、佟大双、程宏筠等人翻译, 邓蛟龙、陈代川、龚亚萍、王晓娟、吴安定、孟丽艳、王学龙、张定军等人也做了大量的工作, 希望图书创作室技术人员对该书进行了仔细审校, 在此深表感谢。

前言

本书是 Microsoft Visual C++ 6.0 程序员指南，不是 C++ 语言，也不是 MFC 库，而仅仅与 Visual C++ 本身有关。

确实，Visual C++ 带有程序设计者的指导手册，即在线帮助。那庞大的帮助系统可能会激励你，使你自信些，因为你想知道的内容会在帮助系统的某个地方。但是，在线帮助的问题是，当你知道你要找什么时，它才能很好地起作用。这本书弥补了在线帮助的不足，但不能取代它。帮助系统和书面文字在目的和风格上存在着固有的差异，因而谁都不能取代另一个。它们一个是提供有关信息，另一个则是教你怎么做。一个范围广泛，而另一个则有深度。在线帮助尽可能简洁地提供清晰的事实，而不是详细的描述，它提供一个步骤列表，来指导完成一些任务，但很少花时间来整体描述。从中你可以知道“如何做”，而不是“为什么这样做”。

本书试图使你成为 Visual C++ 的熟练用户。它按材料的逻辑层次展开，展示了整体的各部分之间是如何相互关联的，它用简单的代码来阐明原理，并常充当指导老师的角色。而且，它携带方便。它确实拥有在线帮助所缺乏的优点。另一方面，在线帮助非常直接，而且范围广泛。数兆字节的帮助文本可以涉及到 Visual C++

的每个细节，而本书仅讨论一些要点。

首先，从本书上获得 Visual C++ 坚实的基础知识，然后，当你经验更丰富，而问题变得更深奥时，你可以转向在线帮助。有意思的是，你对产品越熟悉，在线帮助就越有用。

本书第一版的书名为《Microsoft Visual C++ Owner's Manual》，它成为微软关于 Visual C++ 5 版的正式指南。但微软把第二版改名为《程序员指南》，是为了把它作为 Visual Studio 98 系列丛书（共五本）的一部分，其余四本分别为《Visual Basic》、《Visual J++》、《Visual FoxPro》和《Visual InterDev》四种开发工具提供相关文档。本书独立成册，如果你的兴趣仅局限于 Visual C++，你就找对了地方。本“指南”和其他“指南”之间的相似之处仅在于封面，因为其他“指南”只是在线帮助的副本（每个产品所带的在线文档的副本）。你会发现本书同其他“指南”完全不同。

微软有充分的理由为本书改名，但我对丢掉原来的书名感到遗憾。我选择《Owner's Manual》（用户手册）来尽可能清楚地表达本书的重点，以便使读者对哪些是本书涉及的内容，哪些不是，可以有清醒的认识。如果是在一百多年前，那容忍冗长标题的时代，我可以加上一些东西，如“作为教程、伴侣和参考书，它深入阐述 Microsoft Visual C++ 的知识，它不介绍 C++ 编程语言和 Microsoft 基础类库”。显然，那个冗长的标题并不完全精确。Visual C++ 同 C++ 语言和 MFC 库完整地联系在一起，以致于不可能只谈论 Visual C++，而不提及其他两者。下

面的章节为了说明 Visual C++ 的某个方面而提供了许多范例和程序。代码必须有注释，否则它是没有用的。范例程序的说明也扩展到技术和 MFC 的主题。但这些事情是相互独立的，并且，不会分散我们如何使用编译器的重点。其他优秀的书籍可用来解释 C++ 编程和 MFC 库。

本书描述 Visual C++ 6 版，但较早版本的所有者也能从中受益。Visual C++ 的某些方面同以前的版本相比有相当大的变动，但其他许多地方变动甚微，甚至根本没有变动。近来，Visual C++ 附带发行了一个很小的包裹，其中有少量传单、一些印刷材料及一两个光盘。但是，由于你早就阅读了这些，你可能意识到，Visual C++ 中有大量的材料。我称它为“编译器”，仅仅是因为缺乏更好的名字。除编译器本身外，Visual C++ 提供了链接器、生成实用程序、调试器、文本编辑器、资源编辑器、开发环境、微软基础类库（MFC）、运行时间库、成千上万行源代码和其他许多东西。重申一次：本书不会阐述每件事情。我的目的是帮助你掌握 Visual C++，而不是在某些细节上纠缠。

读者要求

这类书不能从学习曲线的零点开始。如果起点太低，对预备知识的讨论会让人感到混乱。若起点太高，则作者会失去多数听众（除了作为无足轻重的东西，而给人留下印象）。诀窍在于，对有各式各样的技术和兴趣的读者，我们以同一种方式来说明，然而，在谈论深奥知识时，不能失掉一个听众，且在提供基础知识时，

也不会惹恼任何一个人。本书没有太多的要求。我假定你已经熟悉 C 和 C++ 编程语言，以前进行过 Windows 编程，并至少对 MFC 有基本的认识。无论如何，你不必是专家，但如果你理解一些基本概念，如指针、类和消息，你会发现，文本和范例代码更容易懂。幸运的是，关于编译器，这里并没有什么抽象的东西，它只是一个软件而已。

Visual C++ 简史

人们可以举出，Visual C++ 的根源不是开始于 Microsoft 而是 Borland 的事例。一些读者可能会记起，给 DOS 带来集成开发环境，或称 IDE 的想法的 Turbo Pascal。IDE 仍是某个领域内的另一个缩写，它们的状况已经改变了。IDE 只是意味着，编辑器和编译器共同工作，二者可从同一个地方访问。在编辑器中写下源代码，单击 Compile（编译）按钮来启动编译器。当编译器发现错误时，它将编辑光标设到出错语句处，以便于你改正问题。IDE 的想法是，为程序开发提供一个整体环境，程序员无须从中离开到其他程序中。

C 语言正是这个时候诞生的（公元 1987 年），Turbo Pascal 发行了 Turbo C。Microsoft 相应地推出相似的产品，称为 QuickC。我被邀做一些与 QuickC 相关的编程工作，并在结束时写些指导性文章，这些被包括在产品包装中，题为《C for Yourself》（标题不是我想的）。QuickC 作为一个产品单独发售，但它也包括在我们称之为 Big C 的 Microsoft's C 编译器中。当时，Big C 版本号为 5。它的

竞争对手包括：Computer Innovations、Datalight、Lattice、Manx，这些名字现在听起来很陌生。其他有些幸存下来，著名的有 Borland 和 Watcom（现在是 PowerSoft）。它们优秀的产品继续同 Microsoft 公司竞争着。

把 QuickC 同 Big C 放在一起的目的是，使程序员能在 QuickC 方便的 IDE 下编写代码。QuickC 编译起来很快，主要是因为它对代码仅作些微优化（本书后面会谈及优化，那时我们会看到它是如何影响编译时间的）。QuickC 优化时只是登记一些变量，插入几条 LEAVE 指令，仅此而已。结果是编译时间很短。程序调试过后，再在 QuickC 中运行时，程序员就能用 Big C 创建一个正式版，它在代码优化方面做得更为正式些。当用 Big C 编译时，程序的大小减少 15% 或更多，这并非怪事。

QuickC 和 Turbo C 给 C 编程引进了许多东西，但从没有对开发者产生永久性的影响。首要原因就是二者的编辑器都不怎么好（QuickC 编辑器后来合并到 Microsoft QuickBasic 中，现在它作为 DOS 编辑器 Edit.com 仍存在于 Microsoft Windows 95 中）。DOS 下 IDE 的另一个问题就是，它们占据大量内存，只给开发环境下的程序运行留下很少的空间。你经常不得不离开 IDE 来运行和调试程序。许多在开发工作中使用 QuickC 的程序员（包括我自己）仅仅使用它的命令行版本。

随后，Windows 3.0 出现了。

Windows 3.0 尤其是 3.1，为个人计算机引进了正式的 IDE 时代。内存的约束消

失了。而且，如果你准备做 Windows 编程，Windows 环境看来是天生的好地方。人们很容易明白，在 Windows 下进行 Windows 编程，会产生更好的结果。Windows 是个优秀的软件，长期在 Windows 下工作，会使你对一个程序应该做什么，不应该做什么，有更好的认识。

让许多人惊奇的是，Microsoft 把精力更多地集中在 C 编译器的内部支持上，而不是把它的界面升级到一个新的时代。当 7 版发行时，它仍然是一个基于 DOS 的产品，它既可以在 Windows 下的 DOS 模式下运行，也可以在扩展内存管理器下运行（它的包装盒中带有 Qualitas's 386Max）。作为让步，7 版提供了一个名为 Programmer's Workbench 的字符模式的 IDE，它已成为现有标准的累赘。然而，Workbench 展示了自 QuickC 时代以来的自然而然的演变结果。许多菜单中的命令看起来还是很现代化的，例如 New、Open、Save As、Build 和 Open Project。

7 版对编程工作的重要贡献不是它的 IDE，而是它对 C++ 的支持。Microsoft 首次把编译器指明为“C/C++”，来强调其新的双重性。它看起来就像细胞分裂一样。它不只是简单地扩展编译器，来辨认 C++ 超集的新命令，而是牵涉到更多的东西。C/C++ 7 版也引进了 Microsoft Foundation Class 库 1.0 版，完善了源代码。如果没有 Microsoft 发送给开发者的这套优秀的预写类，C++ 就不会成为现在这么流行的 Windows 编程工具。

微软下一个主要的让步是放弃了多数产品对 DOS 的依赖。Microsoft C/C++ 8 版成为真正的 Windows IDE，它就是大家所知道的 Visual C++ 1 版。这个名字利用

了早期 Visual Basic 的成功，但二者从未互相抗衡过。Visual Basic 允许开发者通过大量单击鼠标和少量代码来建立正在运作的 Windows 程序，而 Visual C++ 仅仅通过向导这样的特殊动态链接库来创建启动器源文件。在第 2 章中，我们将看到：向导挽救了众多开发过程中重复性的工作，这种对 Windows 程序来说非常普通的工作是由 MFC 编写的。

Visual C++ 1.5 版之后，微软决定不再将更多的努力花在支持 16 位编程上。Visual C++ 2 版仍提供对 16 位的支持，但自从那时起，Visual C++ 仅用来创建 32 位程序。没有 Visual C++ 3 版。版本号从 2 跳到 4 来，使 Visual C++ 和 MFC 同步，这样就结束了一个小小的混乱来源。然而，这种合并是短期的，因为 Visual C++ 和 MFC 又使用了不同的版本号。

Internet 的流行已经明显地影响了产品设计，在发行第 4 版时，Visual C++ 引进了为 Internet 编程而设计的新类库。5 版也增加了一些新类，但注意力更多地集中在改善产品的界面上，以提供一个更好的在线帮助系统、更高级的宏能力和对在开发者组内进行类和其它代码共享的支持。5 版也合并了 Active Template Library，并显著地改善了编译器优化代码的能力。在后面的章节中我们将看到：6 版作了更进一步的改进。

本书内容

本书分成六个部分，每个部分讨论一个与 Visual C++ 及它的开发环境相关的话题。一直到讨论文本编辑器的第 3 章，我们都有意地保持一些基本的讨论，这有助于确保每位读者，不论是新手还是专家，都能成功地操纵 Visual C++ 开发环境，及在文本编辑器中编写源代码。从第 4 章开始，讨论逐渐变得更专业一些。

第一部分：基础知识

我们通常所说的 Visual C++ 实际上是它的开发环境，它的名字是 Microsoft Developer Studio。二者之间的细小差别并不重要，通常这两个术语可以互换。但如果你不围绕 Developer Studio 来进行学习，就不能有效地使用 Visual C++（Developer Studio 听起来很像 Visual Studio，但它们之间没有关系，因此，在阅读本书时，你可以忘掉它）。

第 1 章可以说是方向性的，它介绍了 Developer Studio，并描述了你在这个环境下工作时将要碰到的主要的窗口。这一章也解释了如何使用 Microsoft Developer Network (MSDN)，它为包括 Visual C++ 在内的所有微软的程序产品提供在线帮助系统。

第 2 章介绍了 AppWizard，它是用 MFC 来为典型的 Windows 应用程序创建启动

器文件的 Visual C++ 向导程序。在本书中，我们将用 AppWizard 来建立一些范例程序。

第二部分：编辑器

Visual C++ 提供了三个不同的编辑器：一个用来创建文本源代码，另一个用来创建菜单和图形文件，而第三个用来创建对话框。每个编辑器都用一章来介绍，首先是在第 3 章中讲述文本编辑器。这一章仔细说明了重要的菜单命令，展示了打开文本文档的快捷方式，并介绍了宏。

第 4 章描述了 Visual C++ 多功能的图形编辑器，它可以用来创建包括菜单、位图、图标和工具栏在内的资源数据。这一章篇幅不小，因为它涉及到许多材料。随着内容的深入，将会产生一个名为 DiskPie1 的范例程序。每个部分首先描述如何用图形编辑器来创建一个特定的界面元素，如菜单或工具栏，然后，通过把该元素加入到 DiskPie1 程序来举例说明。到这一章的结尾，这个程序就成为一个以饼图形式显示磁盘和内存的使用情况的实用程序。

第 5 章讨论对话框编辑器，并展示如何用 Visual C++ 来设计对话框和创建基于对话框的应用程序，如 Windows Character Map 和 Phone Dialer 程序。这一章举了几个例子，包括一个能建立一个属性表（也称多选项卡对话框）的例子。

第三部分：编程指南

第三部分中的几章展示了如何使用 Visual C++ 中的两个重要工具来加速编程开发工作。第 6 章介绍了 ClassWizard，尽管它很难描述，但很容易让人喜欢。当开发 MFC 应用程序时，你会发现，ClassWizard 对于创建和维护类是非常有帮助的。

第 7 章中描述的 Gallery 提供了一个附加组件的集合，你只要点几下鼠标，就可以把它们合并到你的项目中去。Visual C++ 带有许多已经做好的由类的源代码和 ActiveX 控件组成的组件。第 7 章还演示如何为 Gallery 来创建你自己的组件。

第四部分：ActiveX 控件

第 8 章介绍 ActiveX 控件，并展示如何在应用程序中使用它们。第 9 章和第 10 章分别从不同方向来说明如何用 MFC 或 Active Template Library (ATL) 来写一个 ActiveX 控件。第 9 章提供了一个名为 Tower 的范例，来带领你一步一步地创建和编码一个依赖于 MFC 的 ActiveX 控件。然后，第 10 章用 Active Template Library 来建立一个同样的控件，从而清楚地说明这两种方法之间的不同之处。结果可以插入到任何支持 ActiveX 控件的应用程序中。

第五部分：高级主题

第 11 章讨论了调试器的重要主题，它是 Visual C++ 最完美的组成部分之一。这一章说明了调试器的内部特征，描述了调试器窗口和工具栏，然后通过修复范例程序中隐藏的缺陷来检验调试器的能力。

当程序调试过后，你将希望打开编译优化来创建一个发行版。第 12 章讨论编译器优化的一些通常难以理解的主题，同时，展示 Visual C++ 的诸多优化选项开关的作用及使用的原因。

到第 13 章的时候，你应已在 Developer Studio 环境上花费了大量时间，这些足够使你知道自己喜欢什么和想要改变什么。这一章展示了如何定制 Visual C++ 来适合你的喜好。它也通过范例演示了如何编制宏和附加实用程序，来使它们同 Developer Studio 紧密地结合在一起。

第六部分：附录

附录 A 提供了 ASCII 字符和 ANSI 字符的标准表格。你可能会发现附录 A 中的 ANSI 字符表比在线帮助中的类似信息要有用得多，因为该表显示了字符的八进制数形式。这样做是有理由的。我们在第 5 章中看到：在对话框文本中包含大写 ANSI 字符需要字符的八进制数形式。有了这些信息，你就可以在对话框显示的

文本字符串中添加一些有用的标志，例如 © 和 ¼。

附录 B 简要描述了 ClassWizard 支持的 MFC 类，它可以作为快速参考，用来帮助你给新类选择最合适的基类。

附录 C 介绍了 Microsoft Visual Basic Scripting Edition，即 VBScript。Visual C++ 把 VBScript 合并为它的宏语言，因此，如果你以前没有使用过 VBScript 或类似的 Visual Basic 语言，这里可以作为一本初级读物。尽管在 Visual C++ 中录制宏并不需要 VBScript 的有关知识，但是，你只有用 VBScript 编程，才能创建一个多用途的宏。

范例代码

本书中几乎每个范例程序都是以 C++ 语言写的，并使用了 MFC（两个例外是第 4 章中的一个光标演示程序，以及第 13 章的一个基于控制台的小程序）。但是，我在文中的一些程序片段中还是使用 C 语言。我发现 C++ 在简明地说明编程意图方面不如 C 好，并且除了清楚和简洁的优点外，C 还在现在的程序员间充当一种“通用语言”。从理论上来说，C++ 程序员可以直接理解 C，但反过来就不一定对。另一方面，C 没有地方来演示 MFC 应用程序。当我认为有些重要的想法，而且，二者间的差别太大而需要解释时，我偶尔也会同时给出 C 和 C++ 代码。

本书中诸多章节讨论的主题都有很好的范例说明，而我一直试图包括那些有趣的、有用的和能说明问题的范例程序。一些程序是用 AppWizard 创建的，另一些不是，这样就可以模拟范围广泛的编程实践。书中对几乎每个程序都进行了彻底的讨论。书中也包括了源代码列表，因此，你无须在编辑器中打开源文件，就可参加讨论。程序代码力求清楚，而不是优美。因此，毫无疑问，对一部分代码你可以在自己的开发工作中采取不同的处理。例如，我在程序中很少包含有错误检查标志。程序是在 Windows 95 下创建的，但多数是在 Windows NT 下测试的。

附带光盘

所有范例程序的项目文件都在书后所附的光盘上。为了把所有的项目复制到硬盘上，按下列步骤执行 Setup 程序：

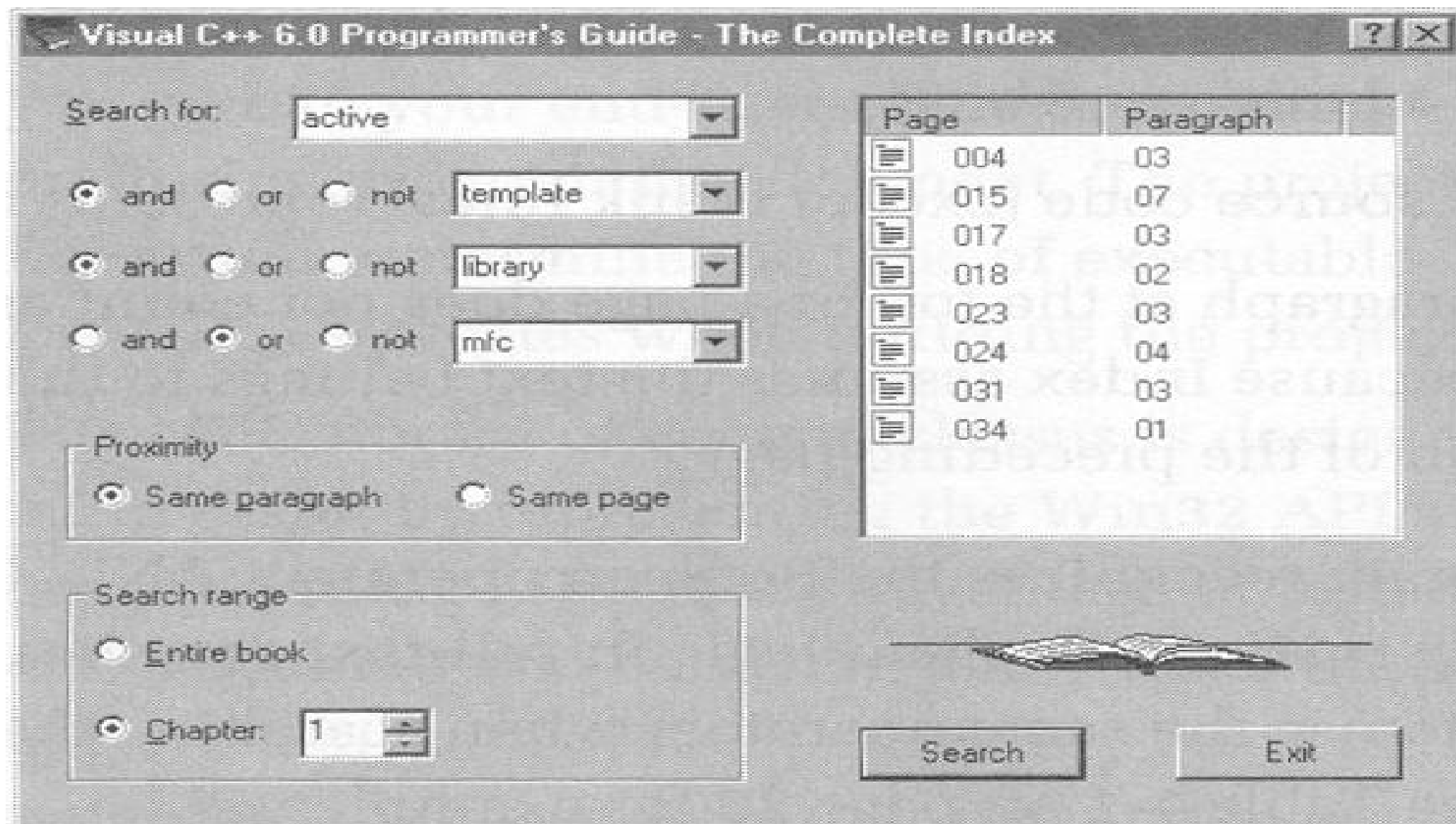
- 1.在 Windows 任务栏上单击 Start 按钮并选择 Run 命令。
- 2.在 Run 对话框中键入“d:\setup”，其中 d 代表光驱的驱动器号。

Setup 程序从光盘上复制 3M 多文件到硬盘上，它们放在名为 Visual C++ Programmer's Guide（或你指定的名字）的子目录下。运行 Setup 完全是可选的，如果你愿意，可以手动地从光盘上再次获得这些文件。你会发现所有的文件都位于 Code 子目录下。

嵌套的子文件夹指明了描述程序的章节号和项目名。例如，子文件夹 Chapter.05\MfcTree 中保留了生成第 5 章中的 MfcTree 程序所需的全部文件。每个项目文件夹都有一个名为 Release 的子文件夹，该目录包含了程序的可执行文件。因此，你可以直接试验范例程序，而不必先生成它。如果你想通过生成范例程序来参加文中的讨论，启动 Visual C++，从 File 菜单中选取 Open Workspace 命令。在硬盘上浏览项目夹，并双击项目的 DSW 文件。

范例程序的项目名保持为八个字符或更少。这一惯例适应了那些宁愿使用较早版本的文本编辑器的读者，这些编辑器可能不能识别长文件名。一些老版本的光盘驱动器也不能识别长文件名。

附带光盘包含了一个名为 Index 的程序。Index 不是一个范例程序，因此你不会在书中找到对它的说明。Index 为本书提供了索引，它在所有的章节和附录中进行全字符搜索。它保证只要某个主题在书中的某个地方提及过，你就可以找到它。这个程序实际上是书志学家所称的“词语索引”的电子形式：给出一个或更多的词，程序就会告诉你，它们出现在哪一页、哪一段。使用这个程序需要把文件 Index.exe、Index.hlp 和 Index.key 从光盘上复制到硬盘上，一定要把这三个文件放在同一目录下。如果你愿意的话，也可以直接从光盘上执行 Index。该程序外观如下图所示。



Index 窗口中的四个组合框每一个接受一个单词。这些单词可以组成一个短语，

如 “Active Template Library”，或仅仅是指定一些一同出现在同一段或同一页的毫无关联的几个词。程序也能搜索单词的复数和由 *-ed* 和 *-ing* 形成的派生词，并能考虑到单词拼写上细微的变化。例如，搜索单词 *edit*、*handle* 和 *debug*，程序同时会找出 *edits*、*handling* 和 *debugged* 的出现位置来。搜索单词的大小写无关紧要，不常见的简写可能会导致意外的匹配。例如，当搜索只取首字母的缩写 GUID 时，Index 会找到单词 *guiding*。单击 Search 按钮或书图标，即可开始搜索。

四个组合框会记住上一个搜索词，因此你无须再次输入。要调用以前输入过的单词，点开组合框的下拉列表，并选中该单词即可。弹出式帮助信息解释了程序的其他特色。单击对话框右上角的小问号按钮，然后单击控件窗口或组合框区，即可获得相关帮助。Windows NT 3.51 的用户必须按 F1 键才能获得帮助。

Index 把一页中的每一段标识为一个数字，如 2 或 7。当搜索一页来找出程序指出的特定的一段时，注意下面的规则，它决定程序认为哪一段是搜索到的：

- 图表的标题组成单独一段，如表格的每一行都是一段。
- 源代码的每一行（包括空行）都代表一段。
- 一页首端的一段的一部分不能作为单独的一段，因为 Index 认为这些文本属于上一页底端的一段。

Index 程序能识别布尔操作符 AND、OR 或 NOT。如果你对全文本搜索中的布尔

逻辑有些生疏的话，第 1 章描述了当搜索 MSDN 在线帮助系统时如何使用布尔操作符。参见表 1-1 的例子。

一些术语定义

在深入本书前，应定义一些术语，例如生成、项目、目标、配置和应用程序框架。由于在下面的章节中要用到这些词，因此最好现在说明它们的意义。

生成指编译和链接，把一组源文件转变成可执行的应用程序。编译源文件，链接对象文件，然后项目就生成了。项目有两个相互关联的含义。它可以指最终的结果，即生成的应用程序，但更准确的是指创建应用程序所需文件的集合，包括源文件、预编译头文件、资源脚本、图形文件和其他建立该程序所需的任何文件。Visual C++ 一次仅允许打开一个项目，这时你已经可以访问所有的项目文件，并可以进行编辑、生成和调试。每个项目可以拥有任意数量的嵌套子项目，当开发一个包含多个可执行元素的程序时，安排这些子项目是非常有意义的。例如，你

可以开发一个应用程序作为主项目，而把一个辅助的动态链接库作为独立的子项目。

生成一个项目后创建的应用程序有两种类型：发行版和调试版。Visual C++ 有时

用术语“目标”来指定生成类型。项目的正式目标是提供给最终用户的可执行程序。调试目标是计划开发过程中的可执行程序。项目设置，就是所说的配置，决定 Visual C++ 生成项目时创建的可执行文件是发行版的，还是调试版的。

一般类的 MFC 库通过提供 Win32 API 作为一套类对象，来使 Windows 编程更容易。使用 MFC 的程序利用测试过的代码作为一个应用程序框架来处理许多任务，否则，应用程序不得不自己来照顾这些事。这些隐藏服务的唯一代价是，对于大多数 MFC 程序来说，可执行文件尺寸可能更大，并存在某些内置的公共部分。应用程序框架通过类来指明应用程序的结构，而不是细节。然而，MFC 并没有严格限制程序员的创造力，许多不同的 Windows 应用程序是用 MFC 编写的这一事实就可证明这一点。

参考资料

推荐参考书可不容易做，我感到责任重大。书是很昂贵的，不仅是在金钱方面，更主要的是在时间方面。下面是一些我认为能给使用 Visual C++ 的程序员提供有价值的资料的书籍。它们碰巧都是 Microsoft Press 出版的，但这只是因为我没说更多的书名。

- 要开始了解 MFC，我相信，没有比 Jeff Prosise 的《Programming Windows 95 with MFC》更好的书了。我喜欢这本书。它写得很好，清楚明了，

主题明确、一致。

- 关于 MFC 的另一本好书是 David Kruglinski 的《Inside Visual C++》。不要被标题愚弄了，这本书的重点是 MFC，讨论了 ProsiSe 的书中没有讨论过的主题，例如数据库管理和 OLE。讨论和所有的范例程序都假定读者正使用 Visual C++。
- 如果你是 Windows 编程新手，想加强基础，并更愿意用 C 语言，而不是 C++ 语言编程，可以考虑 Charles Petzold 和 Paul Yao 的《Programming Windows 95》。该系列的最新版本首先出现在大约十年前，这本书仅仅是因为它对 Windows 编程这一主题清楚说明而出名的。注意书中的警告，尽管除了最后一章，该书没有提及 C++ 或 MFC。
- David Chappell 的《Understanding ActiveX and OLE》详细介绍了 ActiveX。尽管它对 Visual C++ 没有说什么，对编程也很少提及，这本值得一读的书对复杂的主题提供了全面的看法。

反馈

如果你对本书的再版有什么建议，写信给我。我会仔细阅读收到的电子邮件的每一行（我对这些事是很勤奋的），尽管我不能保证回信。在 Internet 的 beckz@witzendsoft.com 上可以访问到我。

目 录

第一部分 基础知识

第 1 章 环境

工具栏和菜单

环境窗口

在线帮助

MSDN 库

在开发环境之外工作

第 2 章 AppWizard（应用程序向导）

AppWizard（应用程序向导）的优点

运行 AppWizard（应用程序向导）

用 AppWizard（应用程序向导）来创建 DLL 文件

第二部分 编辑器

第 3 章 文本编辑器

启动文本编辑器

文档

浏览文档

文本搜索

编程辅助工具

Advanced（高级）命令

未结合命令

宏的基础

定制编辑器

在 Developer Studio 之外编辑文本

第 4 章 资源

系统资源

RC 资源脚本文件

Resource.h 头文件

AppWizard 资源的例子

例程 DiskPie1 简介

菜单和加速键

字符串资源和状态栏

位图、工具栏、图标和光标

向 DiskPie1 添加代码

再谈未结合命令

修整资源数据

第 5 章 对话框和控件

对话脚本

对话编辑器

例 1：修改 About 对话

例 2：简单的非模态对话框

例 3：向 AppWizard 程序添加对话

基于对话的应用程序

第三部分 编程帮助工具

第 6 章 ClassWizard（类向导）

访问 ClassWizard

ClassWizard 对话

WizardBar (向导栏)

ClassWizard 如何识别类

用 ClassWizard 创建对话类

第 7 章 Gallery (组件集)

例子：添加一个属性表

例子：添加初始屏幕和时钟

创建自定义的组件

第四部分 ActiveX 控件

第 8 章 使用 ActiveX 控件

背景知识简介

控件包容器

在包容器和 ActiveX 控件之间进行通信

编写容器应用程序

在没有对话框编辑器的情况下工作

第 9 章 使用 MFC 编写 ActiveX 控件

用于创建 ActiveX 控件的 Visual C++ 工具

ControlWizard

许可

例 1：什么都不做的 ActiveX 控件

例 2：Tower ActiveX 控件

向 ActiveX 控件项目添加属性页

第 10 章 用 ATL 编写 ActiveX 控件

ATL 和容器应用程序

ATL 和 ActiveX 控件

示例 1：Pulse ActiveX 控件

例子 2：TowerATL ActiveX 控件

对比组件模型

第五部分 高级主题

第 11 章 调试器

调试与发行

调试器的使用

断点

断点返回控制权给调试器的方式

建立调试版本

调试器界面

例子：开发和调试 ShockWave 程序

特殊的调试情况

第 12 章 编译器优化

优化入门

优化开关

从调试到发行

基准 Visual C++

第 13 章 自定义 Visual C++

Options（选项）对话框

Customize（自定义）对话框

工具栏

添加命令到 Tools（工具）菜单

宏

Developer Studio 附加项

第 六 部 分 附 录

附录 A ASCII 和 ANSI 文件格式

附录 B ClassWizard 支持的 MFC 类

附录 C VBScript 初步

第一部分 基础知识

第 1 章 环境

Visual C++软件包包含了许多单独的组件，如编辑器、编译器、链接器、生成实用程序、调试器，以及各种各样为开发 Microsoft Windows 下的 C/C++程序而设计的工具。幸运的是，它还包含有一个名为 Developer Studio 的开发环境。Developer Studio 把所有的 Visual C++工具结合在一起，集成为一个整体，通过一个由窗口、对话框、菜单、工具栏、快捷键及宏组成的和谐系统，你可以观察和控制整个开发进程。集成开发环境就像一间带有监视器、刻度盘和操纵杆（它使个人能操作工厂中的许多机器）的控制室。该环境几乎就是 Visual C++中所能“看见”的所有东西。所有其他事情都在其管理下在幕后运行。

产品 Visual C++和它的开发环境 Developer Studio 之间的区别几乎没有意义，因为后者完全取代了前者。为避免应付两个不同的名字，本书在一般场合下，用术语 Visual C++来既指代整个产品，又指代它的开发环境。微软自己已经接纳了这个做法，而以前版本的用户会注意到，原来标为 Developer Studio 的窗口标题已经改为 Visual C++。然而，在最后一章讨论实用程序如何同 Developer Studio 程序结合起来，而成为环境的一部分时，我们将恢复原来的名字。

开始，我们来概括一下 Visual C++环境为帮助程序开发而设计提供的诸多服务。

圆括号中的章节号指出了本书中详细讨论这些服务的地方。

- 对开发进程的不同方面，从类和源文件的列表到编译器消息，都提供了观察窗口（第 1 章）。
- 访问在线帮助的扩展系统的菜单（第 1 章）。
- 创建和维护源文件的文本编辑器（第 3 章），设计对话框用的优秀对话框编辑器（第 5 章），并创建其他界面组件，如位图、图标、鼠标及工具栏的图形编辑器（第 4 章）。
- 为程序创建启动器文件的向导在建立新项目的常规任务上提供一个良好的开端。Visual C++为不同类型的 Windows 程序提供向导，包括带有可选数据库和 Automation 支持的标准应用程序（第 2 章）、动态链接库、基于对话框的应用程序（第 5 章）、使用 Internet Server API(ISAPI) 的 Web 服务器的扩展程序及 ActiveX 控件（第 9 章和第 10 章）。
- 帮助 MFC 应用程序创建和维护类的帮手——ClassWizard（第 6 章）。
- Gallery 维护的内置可执行组件给程序增加方便的特征（第 7 章）。
- 优秀的调试器（第 11 章）。

- 通过对菜单和工具栏对命令进行合理和方便的访问。你可以定制 Visual C++ 中已有的菜单或创建新的菜单（第 13 章）。
- 通过宏和附加的动态链接库来添加自己的环境工具的能力（第 13 章）。你可以自己开发这些附加项，或从各种各样的供应商那里购买它们。

图 1-1 显示了一个典型的 Visual C++ 主窗口。开发环境的外观比以前的版本只有细微的改变，而它的风格和许多命令仍维持不变。如果你从较早版本的 Visual C++ 或其他微软产品中熟悉了 Developer Studio 环境，可以跳过这一章，而直接接触新特征，尤其是修订过的在线帮助系统。如果你以前从未用过 Visual C++，你会发现，像任何大的 Windows 程序一样，它需要一些时间来习惯。不要低估它的深度，只有当你认为已经发现了 Visual C++ 的每件事情时，才会峰回路转，出现一番新的景色。但是，它的界面是智能化的，而且，而且非常宽容，它鼓励你去实践，实践总是最好的教师。

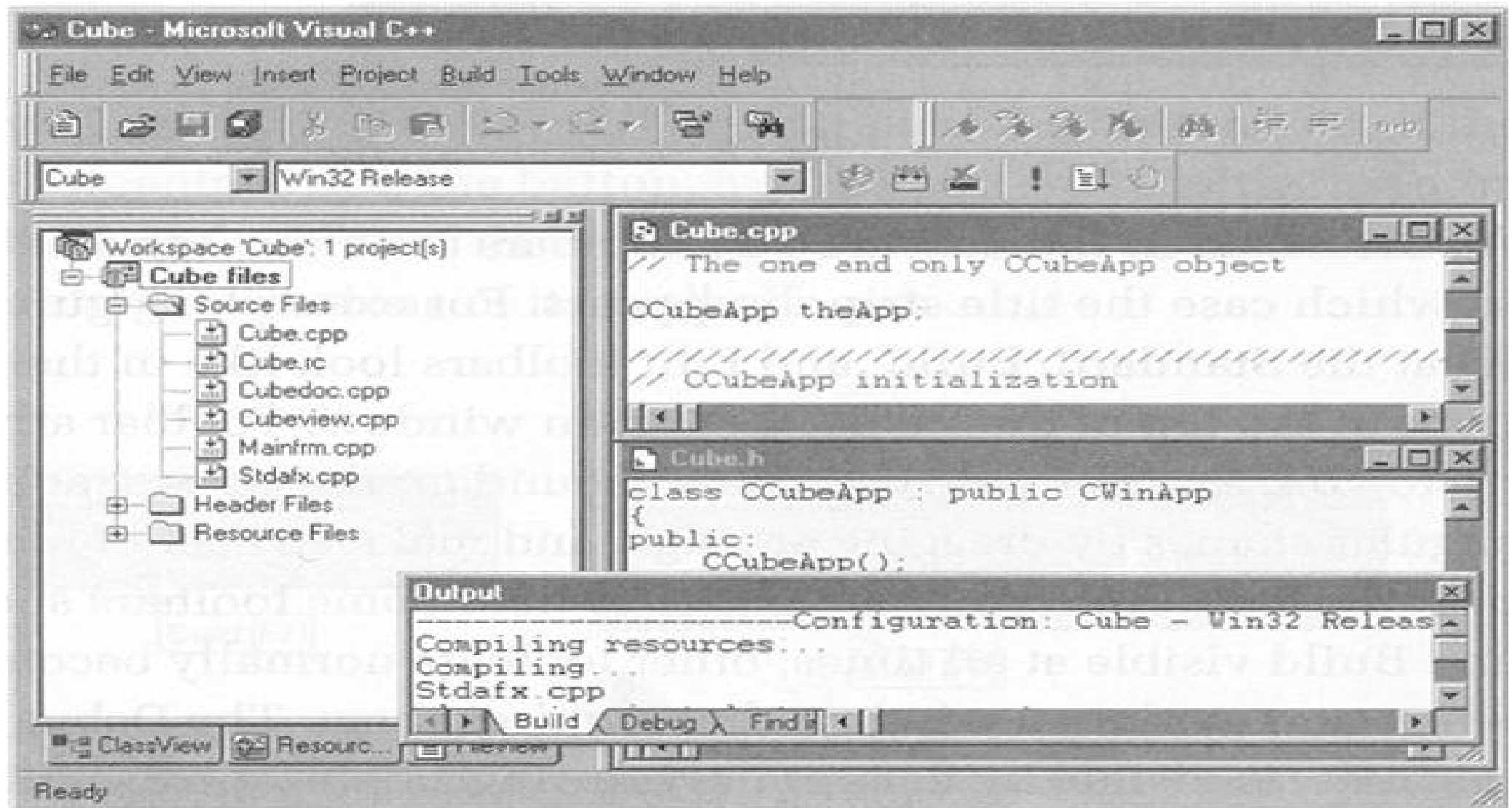
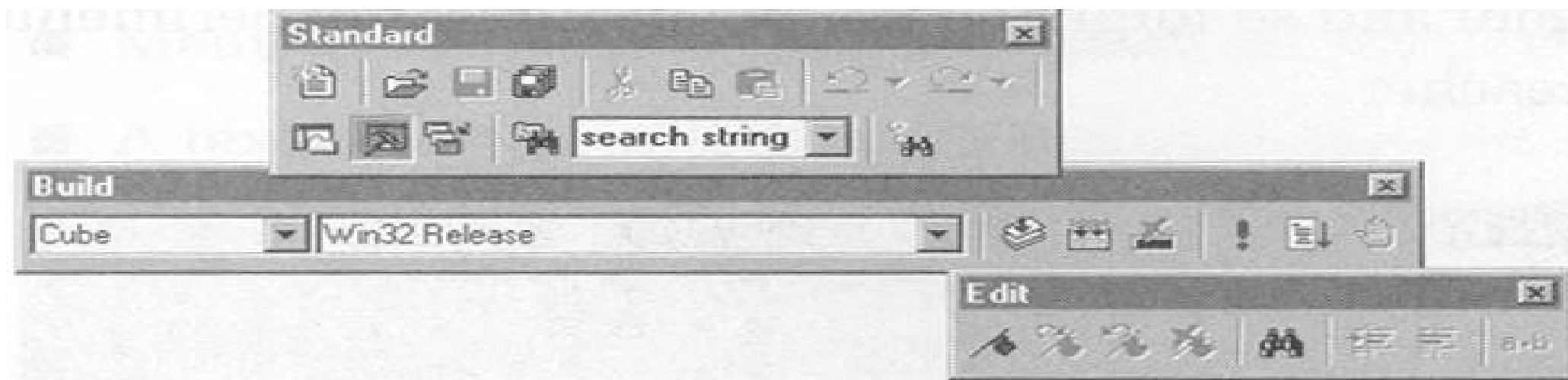


图 1-1 Visual C++主窗口的典型视图

本章是一个起点，它介绍 Visual C++的环境外壳，并描述开发项目时遇到的界面和窗口。我们不急于在本章中讨论单独的工具和菜单命令，因为在后面的几章中，都介绍了菜单和工具栏及它们包含的各种各样的命令。同时，Developer Studio 还充当 Visual J++和 Visual InterDev 的主环境，但这里，我们只集中讨论这个环境是如何应用到 Visual C++和 C/C++项目上的。

工具栏和菜单

Visual C++带有一个预先定义好的工具栏集，单击它们便可以访问它们。如果找不到需要的工具，可以用自己设计的定制工具栏来增大工具栏集。每个工具栏都由工具栏的标题栏上的名字标识：



正如下面部分所描述的那样，工具栏经常被定位在某个位置上，这时标题栏消失。例如，图 1-1 显示了 Standard、Build 和 Edit 工具栏定位在 Visual C++ 主窗口的顶端时的样子。工具栏的安排由你决定。你可以在屏幕四周移动工具栏，通过拖曳边框来调整它们的矩形形状，并使任何一套工具可见或隐藏。你可能愿意让某些工具栏（如 Standard 和 Build）一直都是可见的，至于其他工具栏，通常只在你工作在需要它们的窗口中时，才变得可见。例如，在默认设置中，Debug 工具栏只在调试过程中才是可见的。Colors 和 Graphics 工具栏（将在第 4 章中讨论）仅在图形编辑器中才是可见的，因为这是你需要它们的唯一地方。

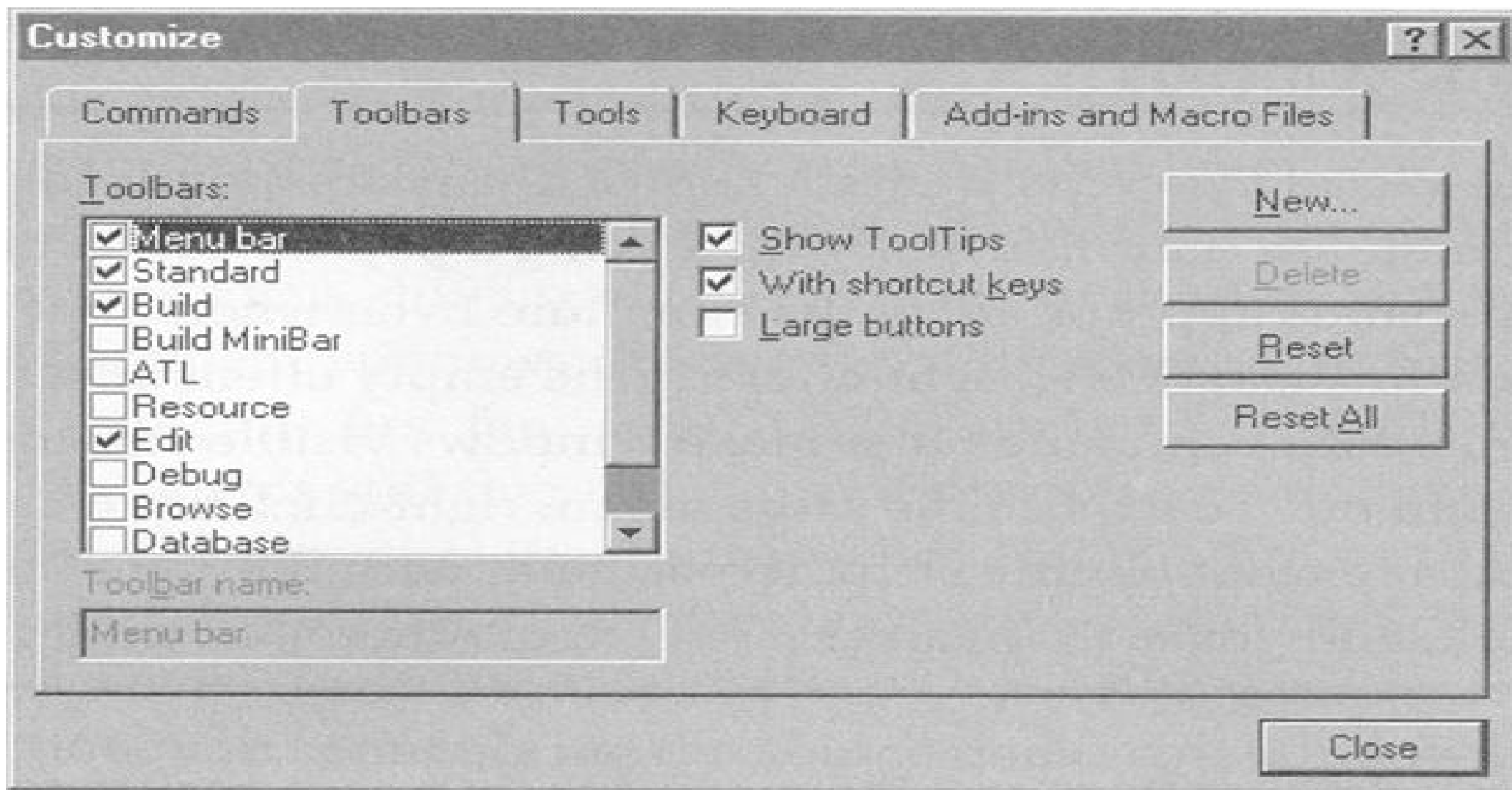
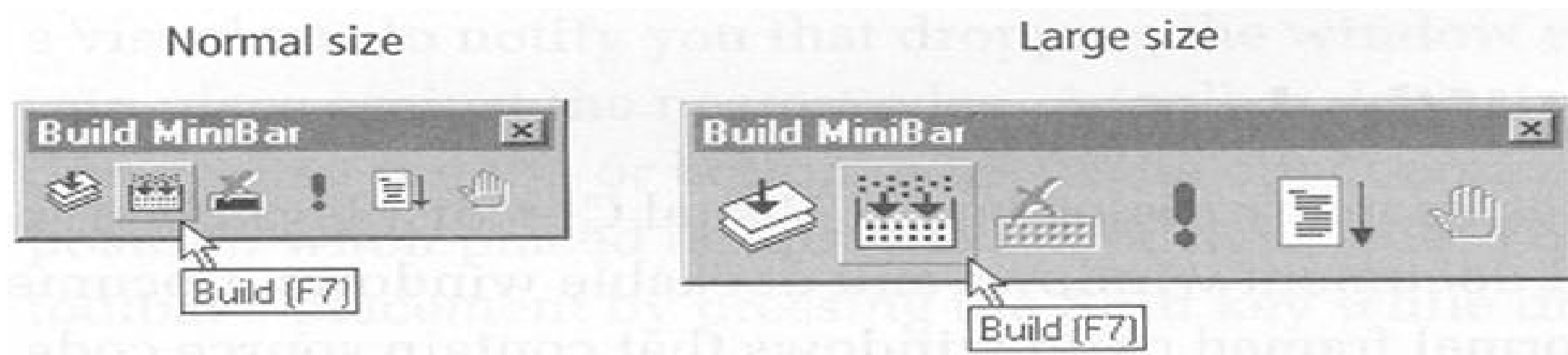


图 1-2 在 Customize（自定义）对话框中打开和关闭工具栏

图 1-2 显示了 Customize（自定义）对话框中包含的工具栏名称列表，这里你通过单击复选框来调整工具栏的可见性是打开或是关闭。在 Tools（工具）菜单中单击 Customize（自定义）命令即可打开这个对话框（第 13 章对 Customize（自定义）对话框有更多的叙述）。

当鼠标停留在工具栏按钮的上面时，按钮凸起。主窗口的底端的状态栏显示了对按钮的简短描述，并且，如果光标停留时间长一些，就会出现一个小的弹出式的“工具提示”窗口，它包含了按钮的名字。如果需要的话，Visual C++ 甚至可以显示放大的工具栏：



工具提示和放大的选项都是在图 1-2 显示的 Customize（自定义）对话框中控制的。

Visual C++ 菜单栏是一种特殊形式的工具栏。尽管只有在全屏模式下才能隐藏菜

单栏，其他情况下，它就像一个普通的工具栏。当鼠标停留在 Visual C++ 的菜单栏上时，菜单名像工具栏一样呈凸起的形状。单击菜单名来拉下菜单时，菜单名看起来像是凹进屏幕里。菜单打开后，把鼠标从一个菜单名滑动到另一个会拉下另一个菜单。

上下文相关菜单

Visual C++ 环境几乎总是都响应鼠标右键的单击，通常是显示一个弹出式的带有与位置相适应命令的上下文相关菜单。甚至当 Visual C++ 没有打开窗口时，在空白区右击会产生一个菜单，其中含有使窗口可见和调整工具栏开或关的命令。在工具栏上除标题栏外的任何地方右击，可打开同样的菜单。工作时试一试右键，你会发现许多其他的快捷方式。

通过在工具栏或菜单栏上不是按钮或菜单名的地方单击，并保持鼠标被按下，就可以把它们拖动到屏幕上新的地方。如果工具栏的标题栏不可见，工具栏上的垂直分隔处正是“抓取”工具栏拖动的好地方。由于停靠这一特征，在 Visual C++ 中移动工具栏，有时并不如你所想像的那样简单。下一部分深入讨论重新在屏幕上定位窗口和工具栏的秘密。

环境窗口

除许多对话框外，Visual C++显示两种类型的窗口，即文档窗口和停靠窗口。文档窗口是一般的带边框子窗口，其中包含有源代码文本和图形文档。Window（窗口）菜单中列出了在屏幕上以平铺方式还是以层叠方式显示文档窗口的命令。所有其他的 Visual C++窗口，包括工具栏甚至菜单栏，都是停靠的。开发环境有两个主要的停靠窗口——Workspace（工作空间）窗口和 Output（输出）窗口，它们通过 View（查看）菜单中的命令变成可见的窗口。另一个停靠窗口 Debugger（调试器）窗口在第 10 章中讨论，它在调试过程中显示。这一部分首先看看所有停靠窗口的一些共同特征，然后单独地讨论 Workspace（工作空间）窗口和 Output（输出）窗口。

停靠窗口可以固定在 Visual C++用户区的顶端、底端或侧面，或者浮动在屏幕上任何地方。停靠窗口，不论是浮动着的或是固定着的，总是出现在文档窗口的上面。这样，就保证了当焦点从一个窗口移到另一个时，浮动的工具栏一直都是可见的，但它也意味着，文档窗口偶尔会看起来像消失了似的。这种情况刚开始时可能会使人不安，但是，文档窗口仍在那儿。例如，如果你正在文本编辑器中编辑源代码，此时打开一个占据整个 Visual C++用户区的停靠窗口，源代码文档就会消失，它隐藏在新窗口之下。如果覆盖的窗口被定位在该位置上，你就无法使文档窗口回到上面来。唯一的解决方法是要么关了覆盖的窗口，要么把它拖到不挡眼的地方去。我们将看到如何快速地打开和关闭停靠窗口。

拖动一个停靠窗口时会出现一个移动的轮廓，它显示了释放鼠标左键时窗口的新的位置。这些轮廓线是灰色虚线，直到它与环境用户区的边界或另一个停靠窗口的边界接触时，它才变成细黑线。这种变化提示你，若放下窗口，会使它定位在边界附近的位置。工具栏在用户区的顶端或底端时定位为水平位置，而在左右边界时则定位为垂直位置。在拖动工具栏时，可通过按 Shift 键来改变工具栏的放置方向。

把窗口定位在希望的位置和尺寸上，有时会花费一些时间。要使窗口占据整个用户区，向上拉动边界，直到鼠标同用户区的上边界接触，然后释放鼠标按键。要把窗口恢复到较小尺寸，拖动窗口，直到光标接触到用户区的左边界。这迫使窗口成为浮动的，你可以通过其标题栏来把它拖动到其他地方。

在屏幕上移动一个停靠窗口时，窗口看起来像是有主意似的，顽强地紧贴着 Visual C++ 主窗口的某一边界，或任何其他它接触到的定位窗口。有两个办法可防止这个问题发生。第一种办法就是在移动窗口时按住 Ctrl 键，来暂时禁止它的停靠特征。第二种办法只对窗口有效，对工具栏无效，那就是禁止窗口的停靠能力，直到你再次使它生效。在窗口内部右击，从上下文相关菜单中，选择 Docking View（停靠视图）命令来关掉命令的复选标志。Window（窗口）菜单也提供对 Docking View（停靠视图）命令的访问，如图 1-3 所示。

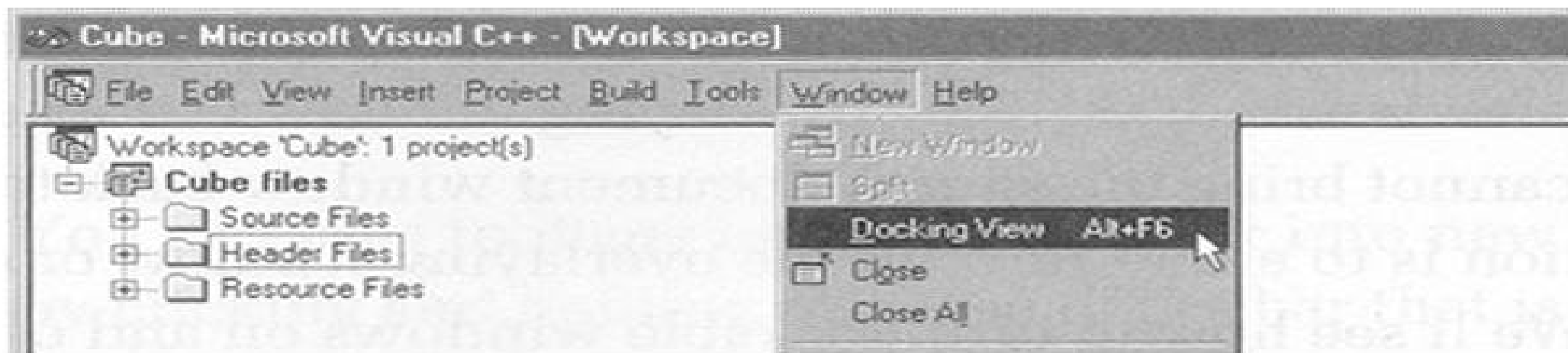


图 1-3 使用 Docking View（停靠视图）命令切换窗口的停靠模式

若关掉窗口的停靠特征，在下面几个方面影响窗口：

- 窗口就像一个普通的文档窗口，标题栏上有最小化、最大化和关闭窗口的按钮。
- 从 Window（窗口）菜单中选择 Cascade（层叠）或 Tile（平铺）命令时，窗口同其他打开的文档窗口一起排列位置。
- 窗口不能移到 Visual C++主窗口用户区的外面，当它处在停靠模式下时是可以的。
- 输入焦点移到该窗口上时，它可以由 Window（窗口）菜单上的 Close

（关闭）命令关闭。否则，Close（关闭）命令不能影响停靠模式下的窗口，即使它拥有焦点。

窗口或工具栏被定位后，凸起的硬节（有时称为把手）出现在窗口的顶端或左边界，如图 1-4 所示。双击把手使窗口变成浮动的；双击浮动窗口或工具栏的标题栏，使它回到原来的位置。也可以通过把手来把窗口拖动到另一个固定的或浮动的位置。

在 Visual C++ 中创建的窗口排列，将在项目的整个过程中维持，或直到你改变窗口排列方式。下一次打开项目时，窗口像上次离开时的样子排列。然而，在环境中运行的实用程序的窗口并不遵守这些规则。这些窗口既不是文档窗口，也不是停靠窗口，它们的特征是由实用程序决定的，而不是由 Visual C++ 决定。

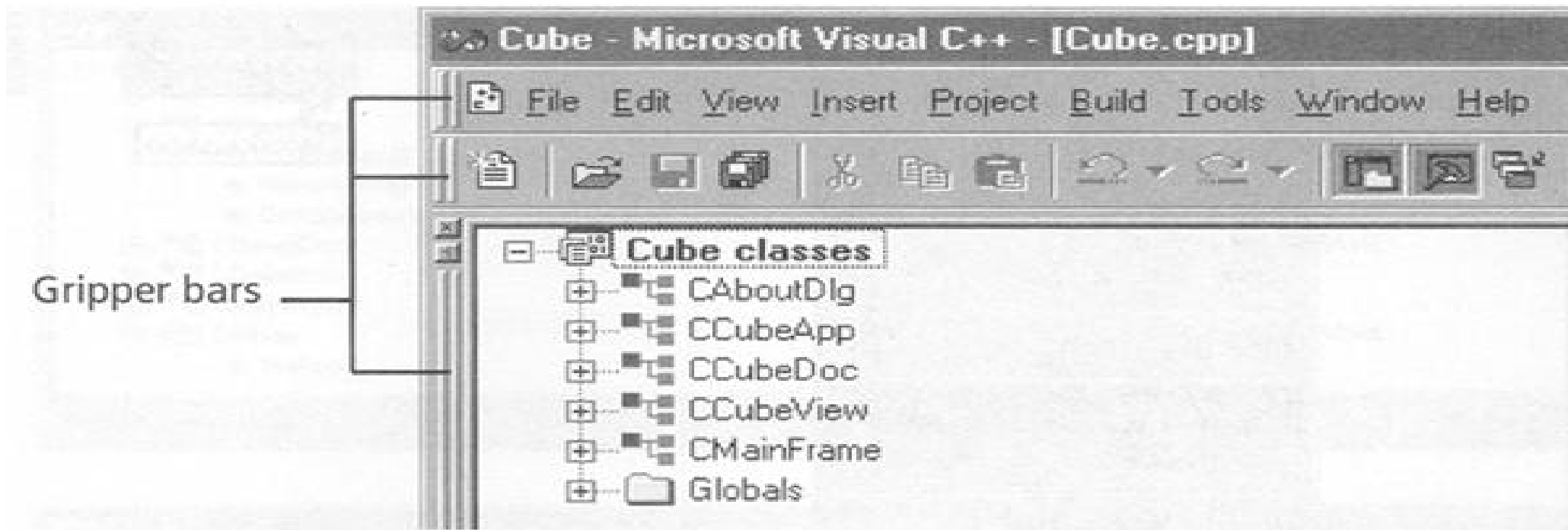


图 1-4 在停靠时，窗口、工具栏和菜单栏中出现一个凸起的把手

Workspace（工作空间）窗口和 Output（输出）窗口

Visual C++在停靠的 Workspace（工作空间）和 Output（输出）窗口中显示项目的有关信息，如图 1-1、图 1-6 和图 1-7 所示。在本书中，自始至终，我们都要遇到这些重要的窗口，尤其是 Workspace（工作空间）窗口，因此，花些时间来讨论它们如何工作是值得的。

要使 Workspace 或 Output（输出）窗口可见，在 View（查看）菜单中单击它们的名字即可，如图 1-5 所示（这个命令不是一个开关命令，因此，再单击它一次不会使窗口消失）。窗口也可以通过 Standard（标准）工具栏上的按钮来激活，这些按钮按下后会使窗口可见或消失。

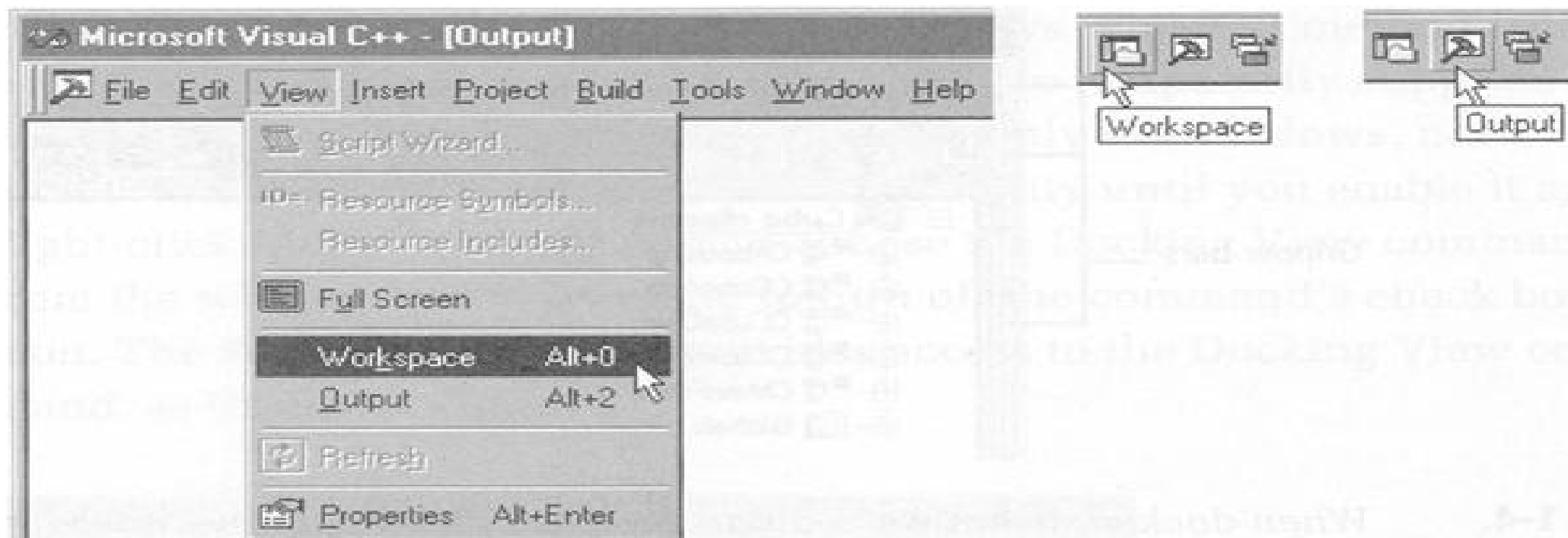


图 1-5 显示 Workspace（工作空间）和 Output（输出）窗口，工具按钮在 Standard（标准）工具栏上

除使用工具栏按钮外，还可以用下列方法来隐藏 Workspace（工作空间）和 Output

（输出）窗口：

- 如果窗口是浮动的，单击窗口的标题栏上的 Close（关闭）按钮。
- 如果窗口是固定的，单击窗口把手上面或右面的 X 形按钮（图 1-4）。
- 在窗口内任何地方右击，打开上下文相关菜单，选择 Hide（隐藏）或 Close（关闭）命令。菜单上出现哪个命令取决于窗口的停靠模式是打开的还是关闭的，但两个命令有同样的效果。
- 如果窗口的停靠特征被禁止，单击窗口把焦点移到它上面，然后从 Window（窗口）菜单中选择 Close（关闭）命令。

Workspace（工作空间）窗口显示了项目各个方面的信息。在窗口底端选择相应的选项卡来显示项目的类、资源、数据源和文件的列表。在窗口中单击小的加号（+）或减号（-）来展开或折叠列表。例如，展开类的列表会显示出成员函数的名字，如图 1-6 中的第一屏所示。双击列表开头靠近文件夹或书本形图标的文字，与单击表头的加号或减号有同样的效果。

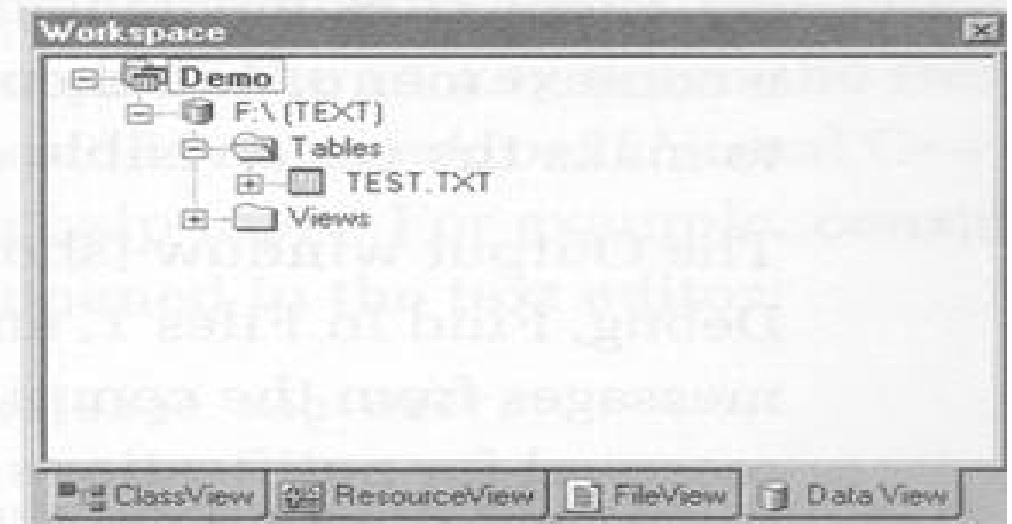
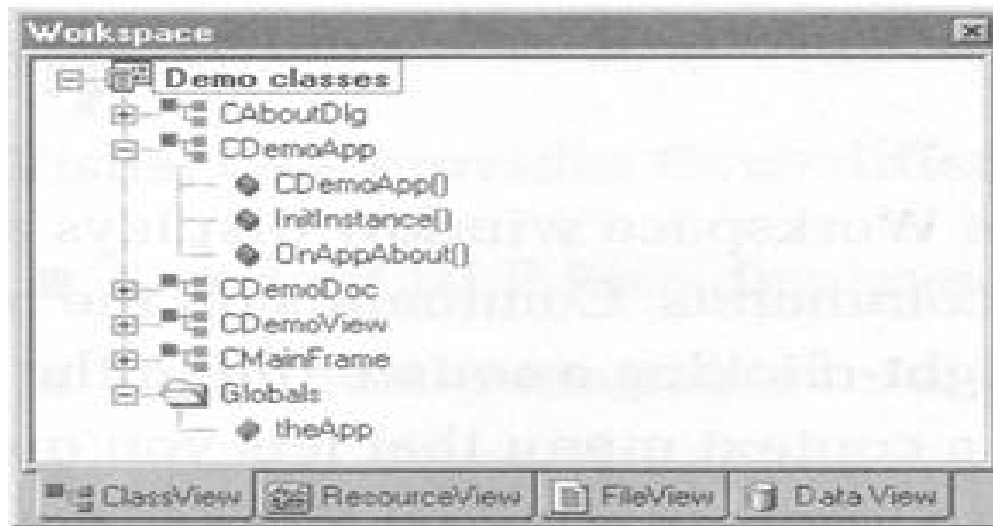


图 1-6 Workspace（工作空间）窗口的四个窗格

Workspace（工作空间）窗口可以显示多达四个窗格的信息，如下所述：

- **ClassView**：列出项目中的类和成员函数。要在 Visual C++ 文本编辑器中打开类的源文件，双击列表中的要打开的类或函数。
- **ResourceView**：列出项目的资源数据，如对话框和位图。同 ClassView 中一样，双击 ResourceView 列表中的数据项会打开合适的编辑器并加载资源。
- **FileView**：列出项目的源文件。把源文件复制到项目夹中，不会自动把文件添加到 FileView 窗格的列表中。你必须用 Project（项目）菜单中的 Add To Project（添加到项目）命令明确地把新文件添加到项目中。
- **Data View**：显示数据库项目的数据来源信息。Data View（数据视图）选项卡只出现在 Visual C++ 的企业版的数据库项目中，企业版遵循开放式数据库互连标准（ODBC）来同数据源相连。

在 Workspace（工作空间）窗口中的某项上右击，会显示一个含有常用命令的上下文相关菜单。菜单中的命令取决于单击在哪一项上。例如，在 FileView 窗格中的一个源文件上，右击，会显示一个快速打开或编译文件的上下文相关菜单。你也可以开关单个 Workspace（工作空间）窗格。在 Workspace（工作空间）窗口的底端的选项卡上右击来显示上下文相关菜单，然后从菜单中选择有关命令来

使窗格可见或消失。

Output（输出）窗口（如图 1-7 所示）有四个选项卡：Build（建立）、Debug（调试）、Find In Files 1（在文件 1 中查找）和 Find In Files 2（在文件 2 中查找）。Build（建立）选项卡显示编译器、链接器和其他工具的状态消息。Debug（调试）选项卡用于通知来自调试器的提示，这些提示对诸如未处理的异常异常和内存异常之类的情況提出警告。应用程序通过 OutputDebugString API 函数或 afxDump 类库产生的消息，也显示在 Debug（调试）选项卡中。

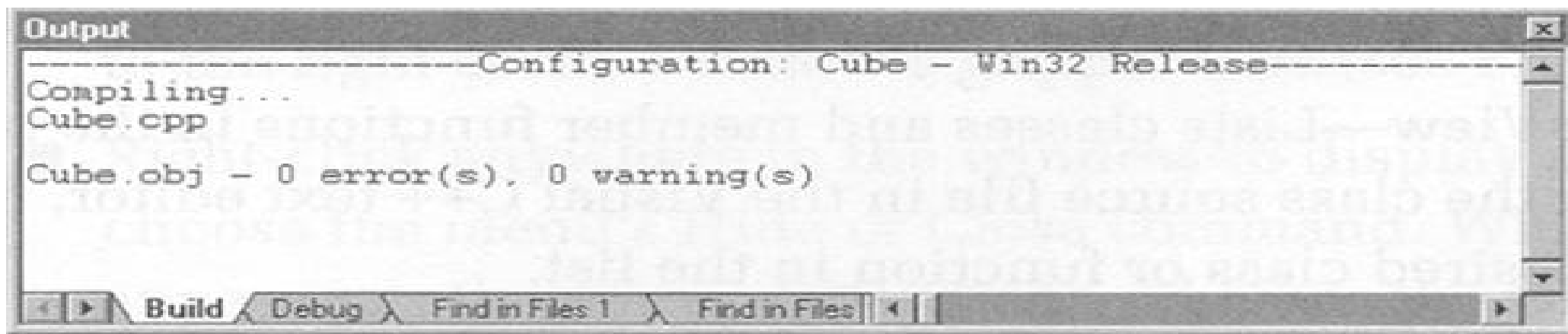


图 1-7 Output（输出）窗口

在 Output（输出）窗口中剩下的两个选项卡，显示从 Edit（编辑）菜单中选中的 Find In Files（在文件中查找）命令的执行结果（这个十分有用的特征与 UNIX 的 grep 命令相似，它将在第 3 章中详细讨论）。默认情况下，Find In Files 搜索结果

显示在 Output（输出）窗口的 Find In Files 1 选项卡中，但 Find In Files 对话框中的一个复选项，允许你把结果转移到 Find In Files 2 选项卡中。Output（输出）窗口中还可以包含其他的选项卡。在第 13 章中，我们将看到如何给 Visual C++ 添加能在 Output（输出）窗口中自己的选项卡里显示消息的定制工具。

在线帮助

Visual C++ 提供了三个不同的在线帮助来源：

- 由 WinHlp32 查看器显示的标准 HLP 文件。
- 对话框中的弹出式帮助信息。
- Microsoft Developer Network Library，即 MSDN。

标准的 HLP 文件内容包括环境的命令和窗口，当 Help（帮助）菜单的 Use Extension Help（使用扩展命令）命令被选中，或 Visual C++ 不能为帮助主题确定明确的上文时，按 F1 键就会显示这些帮助文件。例如，考虑文本编辑器中打开的一个典型的源文件文档中的这一行：

```
DECLARE_MESSAGE_MAP() //MFC message map macro
```

在这种情况下，按 F1 键的效果取决于文本编辑器中闪烁光标所在的位置。如果光标在宏名的内部或开始，而扩展帮助关闭了，按 F1 键会打开 MSDN 库窗口，并显示 DECLARE_MESSAGE_MAP 宏的有关信息。如果光标是停留在一个空行上，这里没有明确的在线帮助的上下文。这种情况下，按 F1 键会提供文本编辑器窗口本身的有关信息，它们显示在 WinHlp32 查看器中：



在线帮助的第二个来源是弹出式消息，在这个环境中的许多对话框中都可以使用。选项卡和偶尔的提示会尽量解释对话框中的编辑框和按钮的用途，但当标签说明不充分时，总可以通过下列几种方法之一来寻求特殊控件的更多解释：

- 把焦点移到该控件上，并按 F1 键。对于复选框或单选按钮，通过单击来给它焦点，可能会打开或关闭一个开关选项。如果你不希望这种结果，当你阅读完帮助信息后，要记得把开关恢复到原来的设置。
- 在该控件上右击，来打开 What's This?（这是什么）弹出式按钮。如果

该控件是一个编辑框，右击该控件的标签文本，比右击编辑框本身要更好一些。单击 What's This?（这是什么）按钮，将显示该控件的帮助文本。

- 单击对话框右上角的问号按钮，然后单击要了解的控件。

这三种方法效果完全一样：运行 WinHlp32，来显示一个如图 1-8 所示的简明的弹出式消息。单击鼠标或按任意键时，这些消息就消失了。

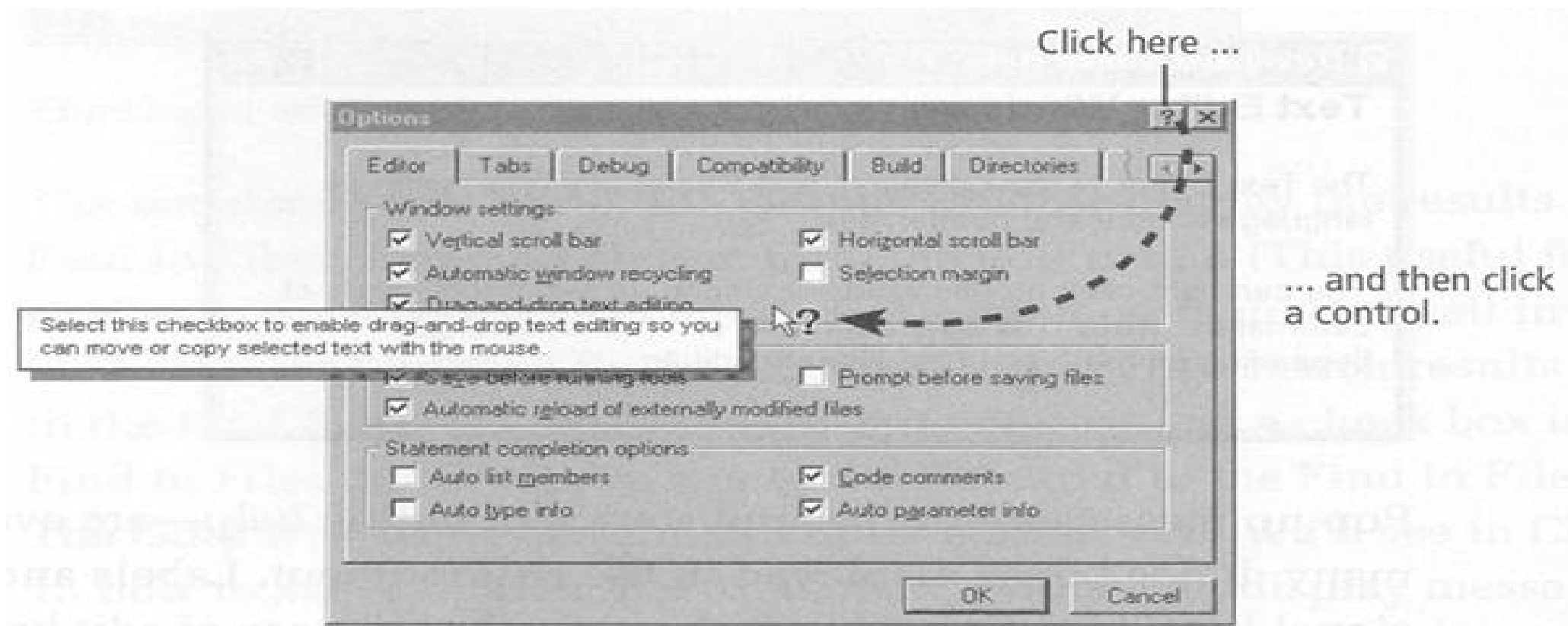


图 1-8 在典型的 Developer Studio 对话框获得帮助

在线帮助的第三个来源是你在 Visual C++ 中工作时最常用的一个。MSDN 通常是合乎逻辑的，而且易于使用，但它也十分庞大。正如下一部分我们将看到的一样，要充分利用 MSDN 库还需要进行一些练习。

MSDN 库

一旦通过预订获得 MSDN 库后，它就可以为包括 Visual C++ 在内的整套 Visual Studio 开发环境工具提供在线帮助。由于所有的工具都可以共享它，MSDN 作为一个应用程序单独运行，而没有同任何单个开发环境结合在一起。要从 Visual C++ 中访问 MSDN，Help（帮助）菜单中的 Use Extension Help（使用扩展帮助）命令必须没有被选中。

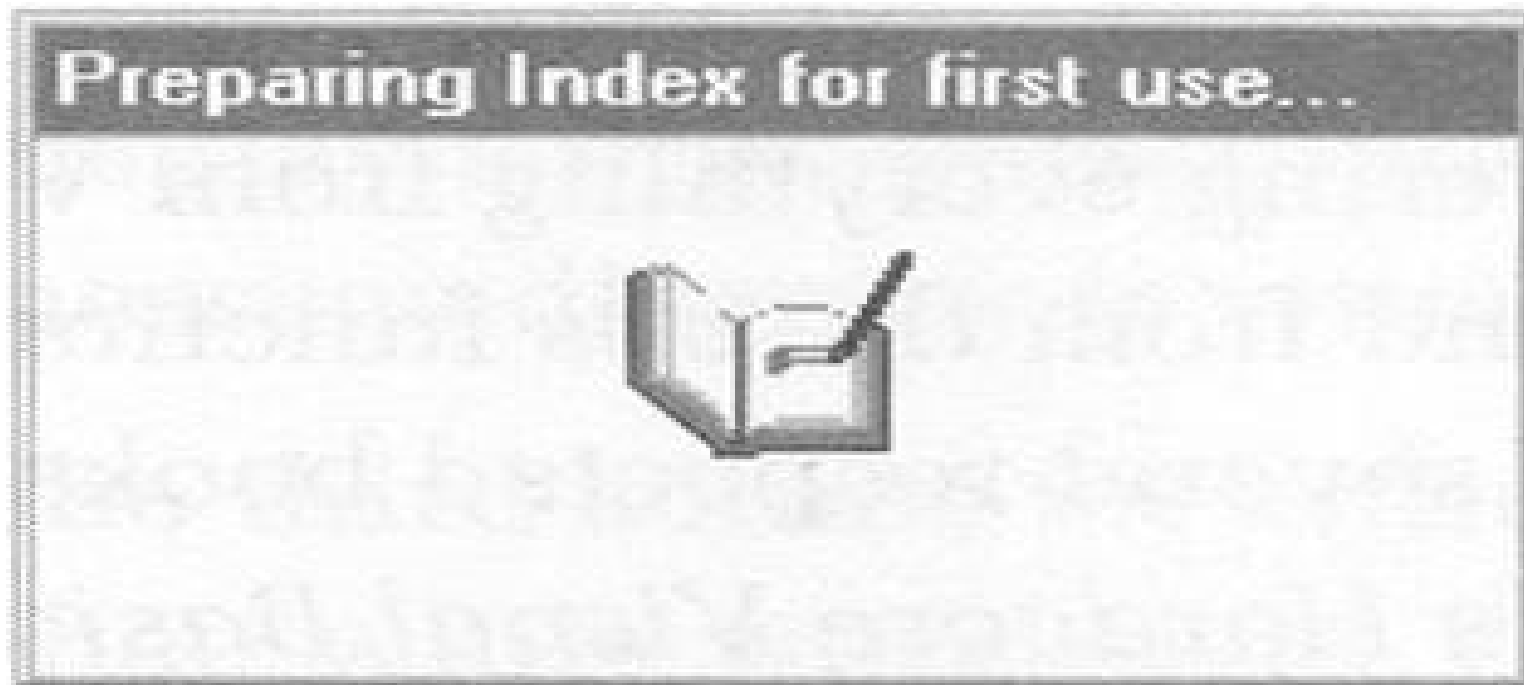
从 Visual C++ 的 Help（帮助）菜单中选择 Contents（目录）、Search（搜索）或 Index（索引）命令，会使环境通过执行 Windows\HH.exe 程序来运行 MSDN，该程序从位于 MSDN98\98VS\1033 文件夹中的文件 MSDNVS98.col 中加载 MSDN 目录列表（包含该文件的文件夹名反映了系统的定位设置，1033 是美国英语的语言代码）。

MSDN 库提供的庞大信息集几乎涉及到微软编程工具和 Win32 编程的每个方面。它包含了成千上万的主题，这些主题讨论了从 Visual C++ 到 Visual J++、从 MFC 到 ActiveX 和从 abs 函数到 z 排列（z-ordering）的每件事情。MSDN 也包括了几本微软出版的重要书籍的全部文字，例如 Bruce McKinney 的 Hardcore Visual Basic 和 Kraig Brockschmidt 的 Inside Ole 第二版。你也可以从中找到 Knowledge Base（知识库）主题、Microsoft Systems Journal 的最新论点、应用程序和设备驱动程序的开发工具包的全部文档、协议文件、范例源代码及其他许多东西。这一版的界面并未经过良好的构思，但 MSDN 中所包括的信息的完整性

确实让人吃惊。

MSDN 帮助系统把文本存在一系列的“伙伴”文件中，这些文件通过扩展名 CHM 来标识；该扩展名指出这些文件是以编译过的 HTML 格式写的。CHM 文件就像单册的 MSDN 百科全书，每一个都包含了讨论一个特殊主题的论文，如 ActiveX 或 Win32 API 参考大全。每个文件都与一个独立的索引文件配对，索引文件有同样的文件名和 CHI 扩展名。在安装 MSDN 的过程中，安装程序把所有的 CHI 索引文件都复制到硬盘上，但只从 CD-ROM 中复制你指定的那些 CHM 文件。CHM 文件占据大量的磁盘空间，因此，你可能会更愿意只安装那些你最常用的帮助主题。在执行过程中，MSDN 访问所有的 CHM 文件，不管它是在硬盘上，还是在光盘上。如果程序在硬盘上找不到需要的 CHM 文件，它会提示用光盘来代替。哪些 MSDN 主题应该安装到系统中，取决于你估计使用 MSDN 的频繁程度、哪些主题你最感兴趣，以及你是否愿意使用光盘。

第一次访问 MSDN 库来搜索一篇文章时，程序创建名为 MSDNVS98.chw 的关键字文件，该文件中包含了所有文章中使用的关键字列表，及指向它们在文本中的位置的指针。以这种方法编辑关键字，会加速特定的词和短语的搜索过程，一会儿，我们就可以看到这一点。创建关键字文件是一次性，只需要几分钟，在此过程中出现动画消息，会告诉你正在做什么：



该过程要求系统 TEMP 文件夹下有许多兆可用的磁盘空间。如果环境变量 TEMP 当前指向一个容量不够的 RAM，在 MSDN 中进行首次搜索之前，需要在 AutoExec.bat 文件中重新设置该变量，并重新启动计算机。MSDN 创建了关键字文件之后，可以恢复原来的 TEMP 设置。

图 1-9 展示了典型的 MSDN 文章，以两窗格显示。这两个窗格被设计成可以共同工作，左边的允许为你希望阅读的文章输入标准，而右边的则显示找到的文章。

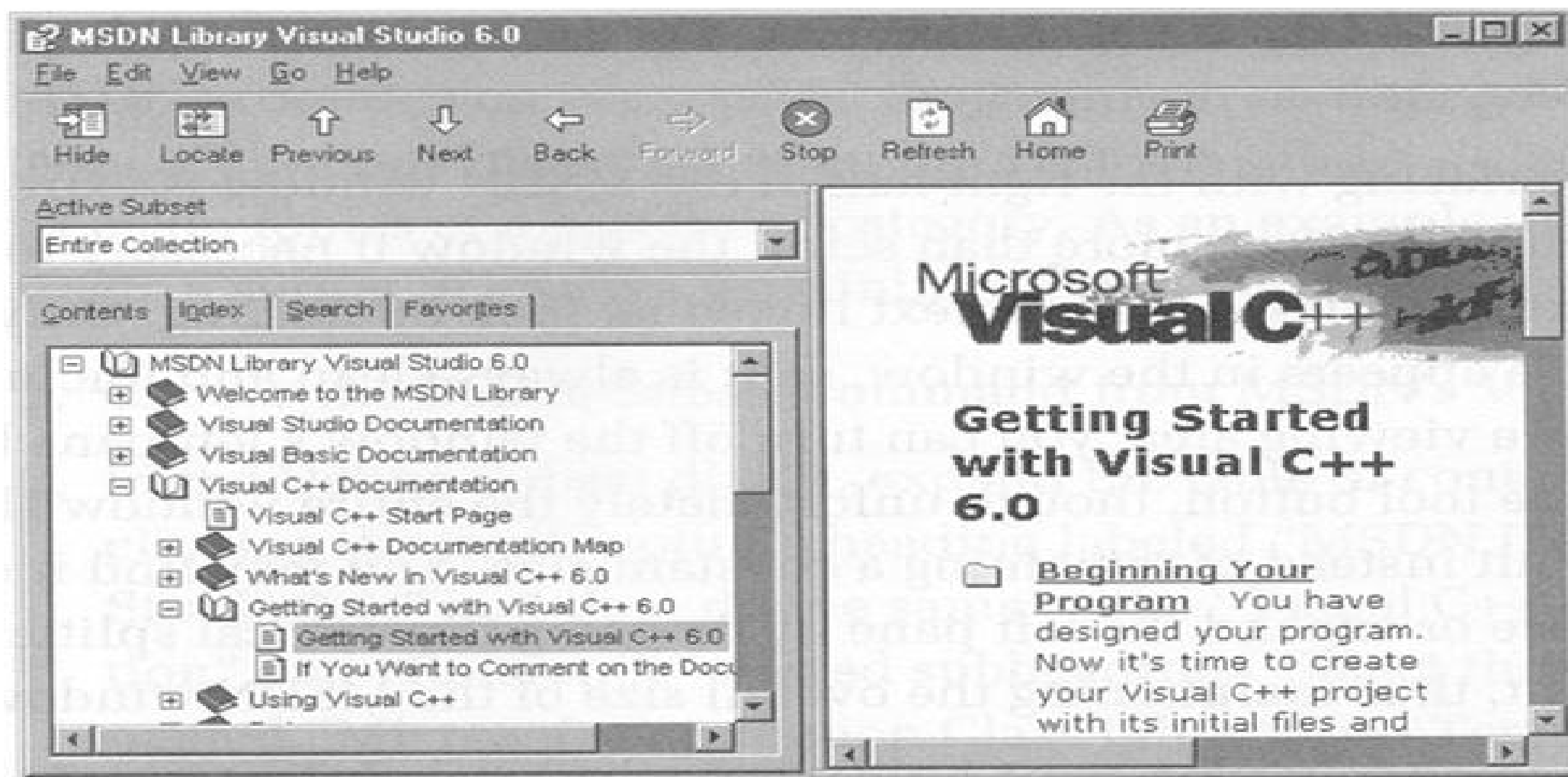


图 1-9 通过 MSDN Library 应用程序来访问在线帮助

在右窗格中，每次都出现主题，它通过超文本链接 web 连接到其他的相关主题。

超文本链接也称超级链接，它是文本中一些特殊的词和短语。链接带有下划线，并以突出的颜色显示，这使得它们能很快被识别。当光标移动到 MSDN 库窗口中的超文本链接上时，光标变成手的形状（图 1-10）。在链接上单击，会从 MSDN 窗口中清除掉当前主题，而代之以超文本链接所指向的新主题。这个效果很像在 Internet 上浏览 Web 网页。

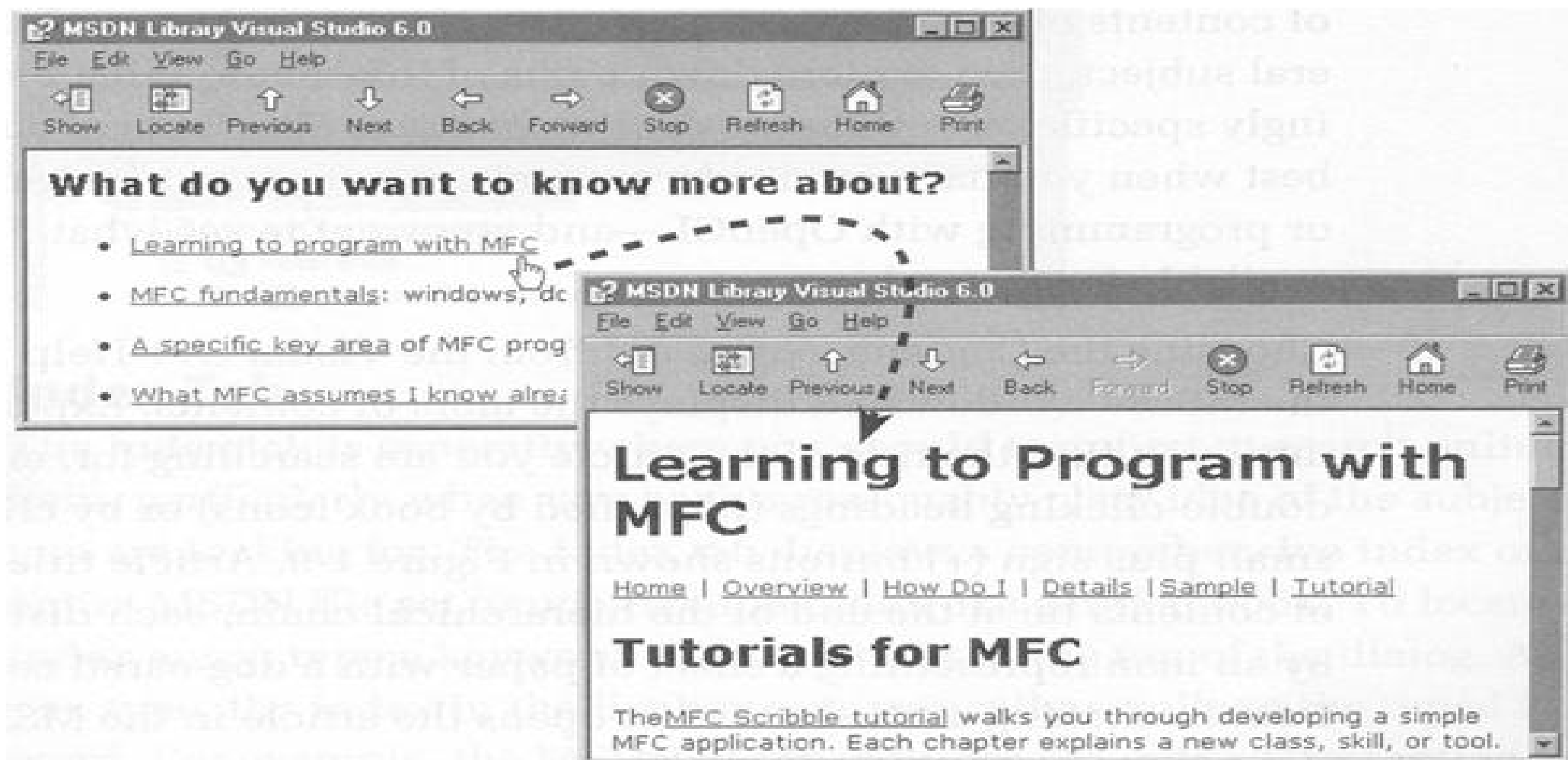


图 1-10 单击一个超文本链接可以从一个 MSDN 文章跳到另一个

MSDN 窗口右半部分的界面是非常友好的，如果需要阅读帮助文本的话，只要上下滚动窗口即可，此外，还可单击超文本链接，来查看感兴趣的主体。在窗口中，

一次只显示一个主题，因此，窗口总是保持干净整洁。为了观察更多的区域，你可以通过单击 Hide（隐藏）工具按钮来关掉窗口左边的一栏，这样，整个窗口会由于保持固定的大小而收缩。你会发现，通过拖动左边或右边的垂直分隔条，更容易关掉或展开左边的一栏，这样也维持了 MSDN 窗口的整体大小。

窗口左边的一栏中有四个选项卡：Contents（目录）、Index（索引）、Search（搜索）和 Favorites（收藏）。每个选项卡提供一种不同的在线帮助浏览方式。

Contents（目录）选项卡

MSDN 的主题组依照标题和副标题的排列方式形成一个目录表。它就像一本书的目录表一样，仅仅是交互式的。开始，可搜索一个大概的主题，然后逐层打开越来越明确的信息路径，来寻找你感兴趣的主题。当你头脑中有一个大概的主题，如调试器或用 OpenGL 编程，而你又想要看一看这个主题有哪些文档时，目录表是非常有用的。

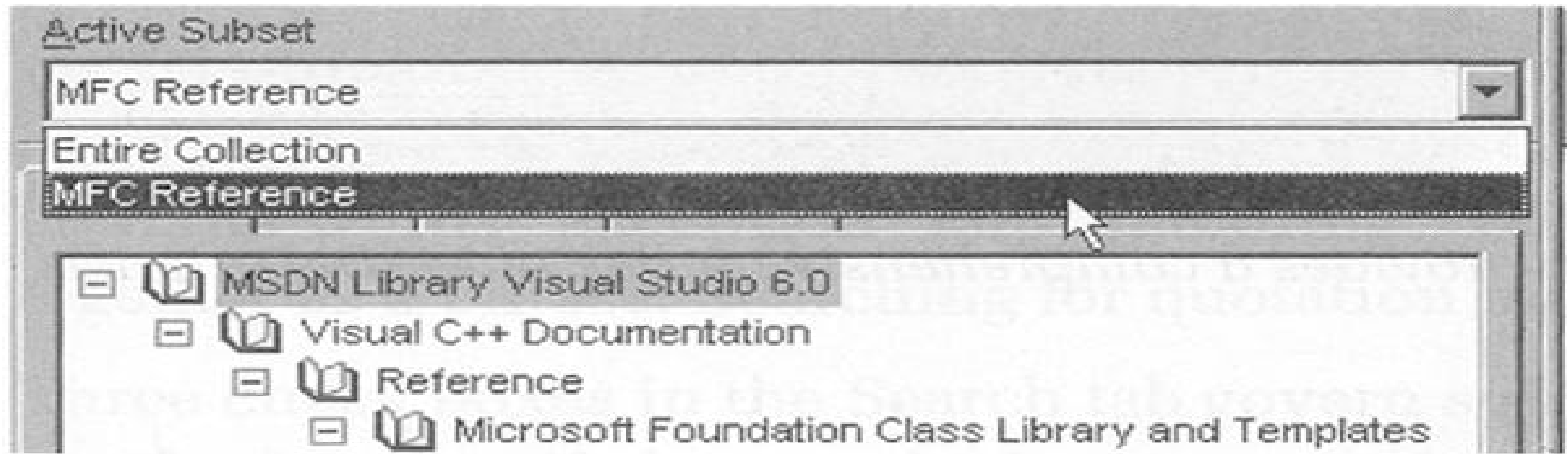
从 Visual C++ 的 Help（帮助）菜单中选择 Contents（目录）命令，会打开 MSDN 窗口，并显示目录列表。展开列表，直到找到要搜索的标题，这可以通过双击标题（通过书形图标来识别），或单击小的加号（+）按钮来完成，如图 1-9 所示。目录列表中的文章标题位于一连串分层的末端，每个标题通过一张折起一个角的

纸形图标来区别。双击列表中的标题，会在 MSDN 窗口的右边一栏中打开该文章。

默认情况下，目录列表概括了 MSDN 的全部文章。你可以通过把目录表的一个分支定义为一个信息子集，来缩短显示的目录。子集允许你集中一些特殊类别的主题。例如，下面是如何创建一个只属于 MFC 参考的文章子集：

1. 从 MSDN 的 View（查看）菜单中选择 Define Subset（定义子集）命令。
2. 在 Define Subset（定义子集）对话框中，通过双击名为 MSDN Library Visual Studio 6.0 的标题来展开目录列表，然后，同样，展开 Visual C++ Documentation（Visual C++文档）和 Reference（参考）副标题。选择名为 Microsoft Foundation Class Library and Templates（Microsoft 基础类库和模板）的副标题，并单击 Add（添加）按钮来创建子集。
3. 在 Define Subset（定义子集）对话框的底部的编辑框中键入新子集的名字，然后单击 Save（保存）和 Close（关闭）按钮。

使用在线帮助时，要在各个子集间切换，只需从名为 Active Subset（有效子集）的下拉列表中选择一个子集即可：



Index（索引）选项卡

搜索在线帮助时，通常首先要求助于 Index（索引）选项卡，尤其当你对要寻找的主题相当清楚时。Index（索引）选项卡显示了整套 MSDN 文件的全面索引，就像一本书的索引一样。要找到一个索引条目，在对话框顶部的编辑框中键入关键字即可。在你键入关键字的时候，列表框中的索引自动滚动到该关键字所在的位置。例如，MSDN 索引包含条目“exception handling（异常处理）”、“handling exceptions（处理异常）”和“C++ exception handling（C++异常处理）”。因此，输入这些词条中的任意一个，都可以找到有关“异常处理”的主题。找到所要的

索引条目后，双击它。如果该条目仅对应一篇文章，MSDN 会立刻显示它；否则会显示 Topic Found（找到的主题）对话框，其中列出了该条目指向的所有文章，如图 1-11 所示。通过双击列表中的标题，或选中该标题，并单击 Display（显示）按钮，均可在对话框中打开一篇文章。

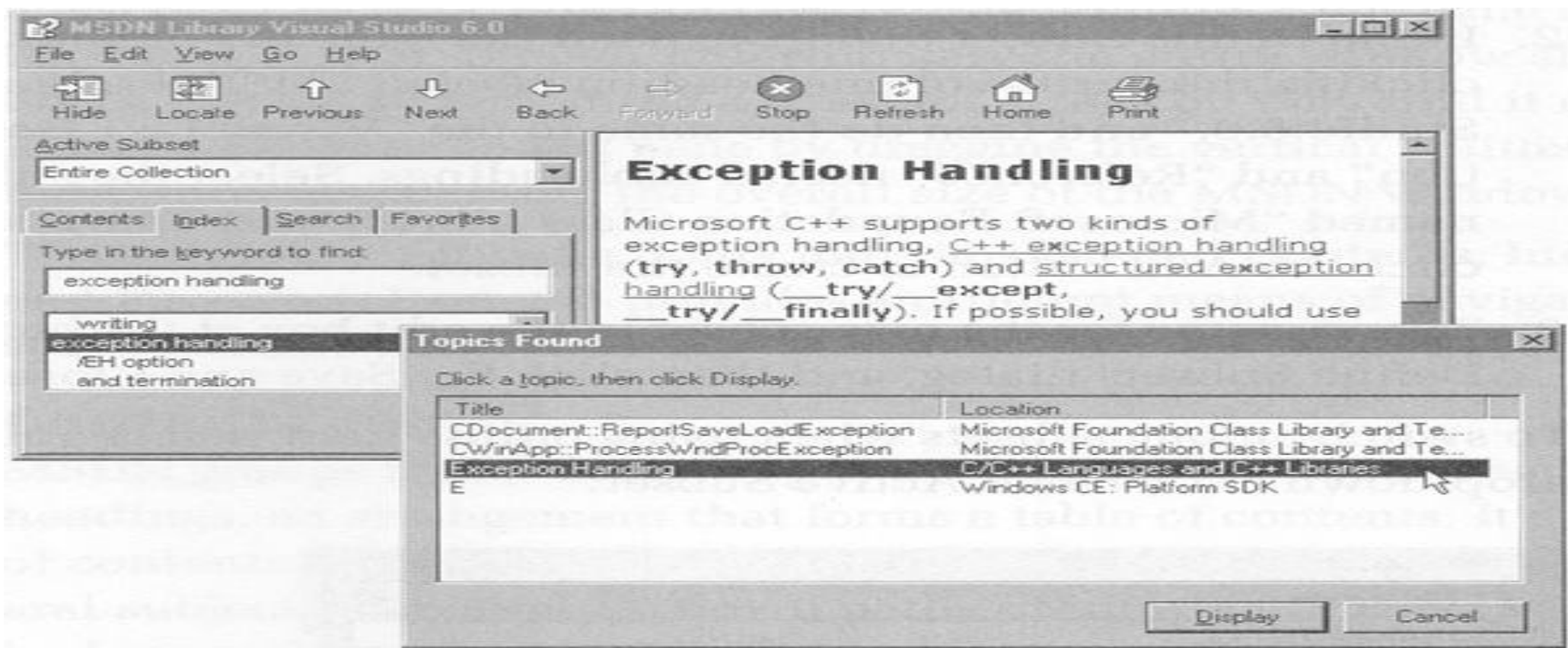


图 1-11 提供 MSDN 文章全面索引的 Index（索引）选项卡

Search（搜索）选项卡

MSDN 帮助文件集不仅仅是被动的，它还包含一个搜索引擎，该引擎扫描 MSDNVS98.chw 关键字文件，来确定哪些主题文件包含了指定的词或短语，这个过程称为全文本搜索。全文本搜索从 MSDN 的 Search（搜索）选项卡中启动（图 1-12），它允许你寻找包含指定词或短语的主题。MSDN 搜索引擎是非常优秀的，它能理解词的派生、通配符、布尔逻辑组合及 NEAR 运算符。尽管要有效地使用这些特征，要求认真地考虑和进行计划，它们允许你精选搜索参数，来增加只寻找你最感兴趣的那些主题的机会。

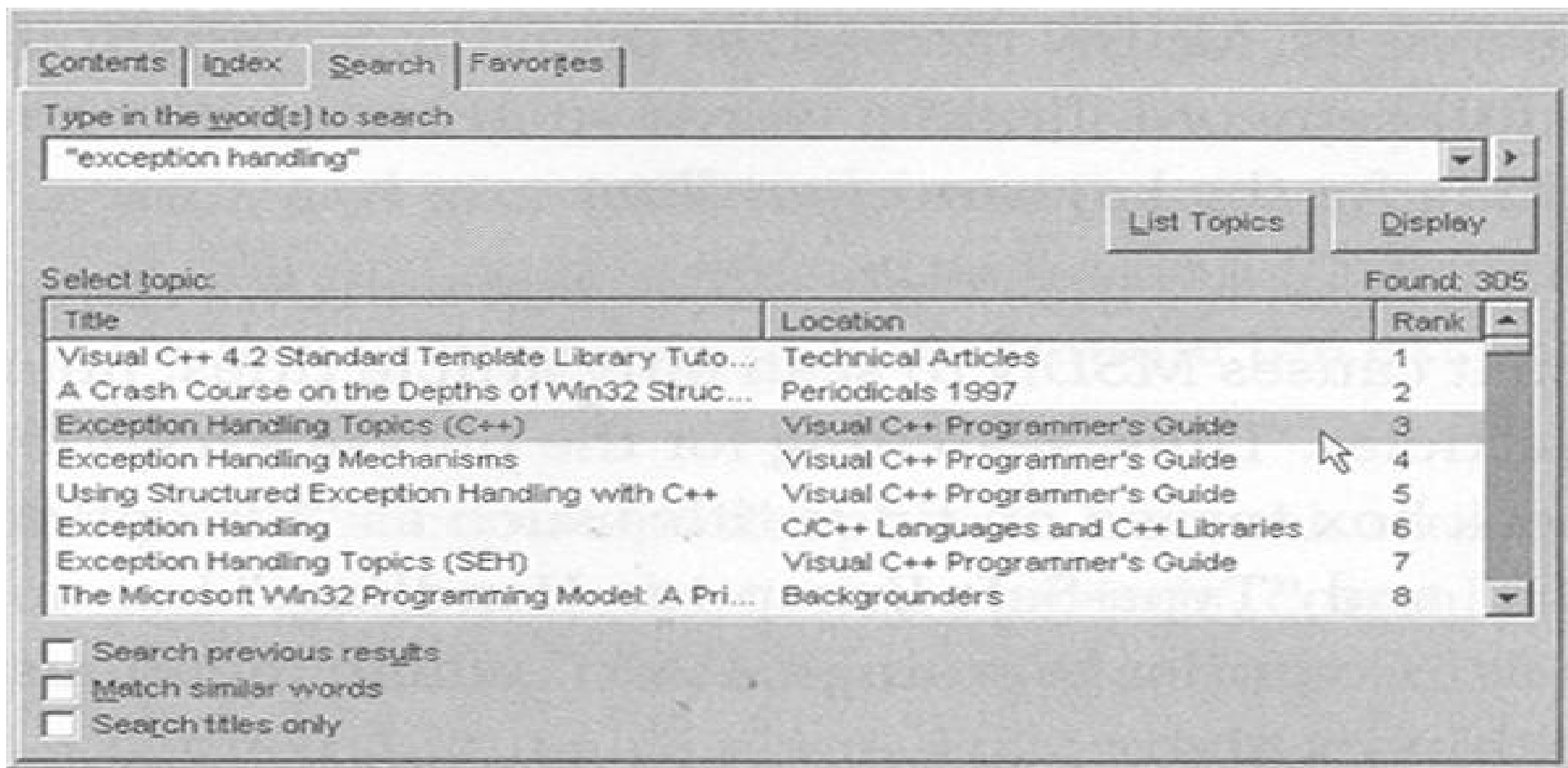


图 1-12 利用 Search（搜索）选项卡可以搜索包含特定词或短语的主题

讨论过 Search（搜索）选项卡中可用的各种各样的选项之后，我们将集中讨论如何用通配符和运算符来精炼搜索内容。

在选项卡的顶部键入要搜索的词或短语，用双引号把短语括起来，以区别于单个词（单引号会被忽略）。例如，搜索图 1-12 所示的词，会找到那些只包含短语“exception handling”的主题。键入同样的单词，而不带双引号，就表示你想搜索包含单词“exception”和“handling”的主题，它们不一定要作为一个短语同时出现。不能搜索引号。

利用 Search（搜索）选项卡中的三个复选项，你可以进一步指定如何和在哪里搜索。Search Previous Results（搜索以前的结果）复选项，可将搜索过程限制在已经出现在 Search（搜索）选项卡中的那些主题范围内。打开 Match Similar Words（匹配类似的词）复选项，会让 MSDN 接受在文本框中键入词的合乎文法的派生词。派生词包括通常的后缀，如 s、ed 和 ing，例如，它迫使 MSDN 认为 edits 和 edited 同关键字 edit 是相匹配的。这样来放宽搜索标准，当然会找到更多的主题。Match Similar Words 选项对编辑框中所有的搜索词都适用，打开这个选项时，搜索短语 handle exception，会找到包含相近的派生词，如 handled exceptions 的主题。MSDN 甚至能识别那些不包含整个关键字的派生词，例如，搜索关键字 handling 时，会找到 handler 和 handled。

打开 Search Titles Only（只搜索主题）复选项，会极大地限制搜索范围，因为它使 MSDN 只扫描文章标题，不扫描文章中的正文。这样，打开该选项时，搜索短语 exception handling，会找到如“Exception Handling Topics(SEH)”和“Type-Safe Exception Handling”之类的标题，但不能找到其他的相关主题，例如“Compiler Warning C4530”，尽管后者在文中提到了异常处理。

搜索完成后，MSDN 列出所有提到过给定搜索字符串的文章的标题，同时在列表的右上角显示找到的文章数目。列表是按搜索字符串在主题文档中的出现次数递减的次序排列的。要想按标题或文章位置排列，单击相应列表栏顶部的按钮即可。双击 Search（搜索）选项卡中的列表条目，会显示该文章，文中的匹配字符串以高亮显示，这使你可以很快地找到字符串的出现位置。文本中经常重复出现的高亮字符串看起来有点分散注意力，它使文章外表就像一篇笔记。要从显示中去掉高亮，从 View（查看）菜单中执行 Highlights（高亮）命令两次，或单击 Previous（前一个）和 Next（下一个）工具按钮，来临时转到另一篇文章，再返回来。你也可以用 Edit（编辑）菜单中的 Find In This Topic（在此主题中查找）命令在显示的文章中查找文本。

下面是在 Search（搜索）选项卡中设计搜索参数的一些基本的规则和告诫：

- 搜索过程不区分大小写，因此用输入搜索短语时，用大写或小写都可以。
- 默认情况下，MSDN 只能查找整个单词。例如，搜索 key 不会找到 keyboard。通配符可以克制这种默认动作，就像前面解释的那样。
- 可以搜索字母和数字的任意组合，包括单个字符（a，b，c，1，2，3 等），但不包括简单的单词，如 an，and，as，at，be，but，by，do，for，from，have，he，in，it，near，not，of，on，or，she，that，the，

there, these, they, this, to, we, when, which, with 和 you。MSDN 在匹配文本时会忽略这些词，因此，搜索 `handle exceptions` 时，也会找到包含短语“`handle the exception`”或“`handle an exception`”的主题。

- MSDN 允许搜索字符串中有省略号，但它忽略其他标点符号，如句号、逗号、分号、冒号及连字号。这使搜索字符串时可以不管上下文，但它也提供了出现错误匹配的机会。例如，搜索短语 `exception handling` 时，可能会找到包含下面文本的毫不相干的主题：`Messages are an exception. Handling a message...`

通配符和运算符

搜索字符串可以用标准的问号（`?`）和星号（`*`）通配符来组成一个一般的表达式，这些通配符可以代替不在双引号之内的字符。问号通配符在表达式中代表单个字符，因此，搜索字符串 `80?86` 时，可以找到“`80286`”、“`80386`”和“`80486`”（但没有“`8086`”）。星号通配符可以代表任意次序的零个到多个字符。例如，搜索 `*wnd*` 可以找到如“`wnd`”、“`CWnd`”、“`HWND`”和“`wndproc`”之类的文本。星号通配符使 MSDN 可以找到包含有共同的词根的所有单词。例如，查找诸如“`keyboard`”、“`keystroke`”和“`keypress`”之类的单词时，可以键入 `key*` 来代替 `key`，作为搜索字符串。自然，这种方法可能会发现诸如“`keyword`”和“`key_type`”之类的不相干的搜索结果。运算符可以进一步提炼搜索标准，来减少这种不希望发生的结果。

MSDN 能识别布尔运算符 AND、OR、NOT 及近程运算符 NEAR。描述这些运算符作用的最好办法是通过表 1-1 中例子来说明。NEAR 运算符认为，当字符串之间相隔不超过八个可标识单词时，字符串是“近程的”。MSDN 没有为近程的确定提供另外的标准。

表 1-1 在 MSDN 的 Search 选项卡中字符串运算符的效果

运算符	例子	结果
AND	debug AND window	搜索文中含有 debug 和 window 的主题，但只含有一个的主题不包括在内
OR	mfc OR “foundation library”	搜索含有该字符串中的任何一个或两个的主题
NOT	ellipse NOT cdc	搜索仅含有第一个给定字符串，而不含有第二个字符串的主题。左边的例子指明了那些既包含有 ellipse 又包含有 cdc 的主题应该被跳过，这样就忽略了关于 CDC::Ellipse 函数的主题

续表

NEAR	handl* NEAR exception	搜索文中给定字符串相隔不超过八个单词的主题

要用运算符连接两个单词，在单词之间用空格分开，并键入运算符，如表 1-1 所示；字母的大小写无关紧要。你也可以单击靠近组合框的箭头按钮（▸），来从小的弹出式菜单中选择要用的运算符。

运算符没有确定的优先权，MSDN 按从左到右的次序求表达式的值。如果要用运算符来明确地组合字符串，可以使用圆括号。MSDN 忽略双引号内部的圆括号，因此，不能搜索带圆括号的注释。在没有其他运算符、圆括号或双引号的情况下，Search（搜索）选项卡把搜索字符串中的空格视为 AND 运算符。因此在 Search（搜索）选项卡中输入下列搜索字符串中的任意一个，都会产生同样的效果：

debug AND window AND breakpoint

(debug AND window) breakpoint

debug window AND breakpoint

debug window breakpoint

早先版本的 InfoViewer 帮助系统允许使用 C 语言中布尔运算符的等同符号：用符号 &、竖线 (|) 和惊叹号 (!) 来分别代替 AND、OR 和 NOT。MSDN 忽略这些字符，因此它们都等同于 AND 运算符。

搜索策略

搜索在线帮助的方法不是取决于要搜索的东西，更多的是取决于如何能描述好想要搜索的东西。如果能用一两个明确的关键字说明要搜索的主题，通过 MSDN 索引来搜索通常是最有效的方法。就像一本书的索引一样，MSDN 索引在关键字和一个相对较小的相关文章列表之间建立了联系，这使你可以快速找到所需要的信息。另一方面，全文本搜索覆盖范围更宽，它比索引提供更多的文章以供选择。搜索结果取决于你如何仔细地用短语描述搜索字符串，并充分利用搜索运算符。经过一次全文本搜索之后，在搜索列表中仔细地寻找最适合于自己的问题的文章，将是一个单调乏味的过程。

如果该主题对你来说是陌生的，你可能需要概述信息，并了解一般的背景知识。这种情况下，MSDN 目录列表可能是最好的求助对象。首先看看目录列表的整体结构来了解其内容。有时，少量索引或全文本搜索能帮助你找到感兴趣的列表区。用这种办法找到一个感兴趣的主题后，你可以通过单击 MSDN 工具栏上的 Locate（定位）按钮，来确定该主题标题在目录列表中的出现位置。Previous（前一个）和 Next（下一个）按钮选择列表中相邻的文章，它使你能够通过按顺序排列的相关文章来浏览整个列表。许多主题开头都有几行非常有帮助的超文本链

接，它们能带你到一个主页、一个主题综述、一个常见问题列表等等。

Favorites（收藏）选项卡

在帮助文本中从一篇文章跳到另一篇文章时，你会不可避免地想要回到早先阅读过的文章。图 1-13 所示的 Favorites（收藏）选项卡这时就有用了，它维护着一个标记选择过的文章的书签列表，这样，你就可以很快地返回到以前阅读过的文章了。这些标记就像一个由 Web 浏览器维护着的书签，并且你会发现，它们对你在探究在线帮助时折回原来的位置很有帮助。如果不删除，添加到该列表中的标题会一直保留，因此，当你启动 MSDN 程序时，Favorites（收藏）选项卡保持上次离开时的样子。

当前文章（即显示在右边窗格中的文章）的标题出现在 Favorites（收藏）选项卡的底部。单击 Add（添加）按钮，将该标题添加到列表中；双击列表中的一个标题会回到该文章。

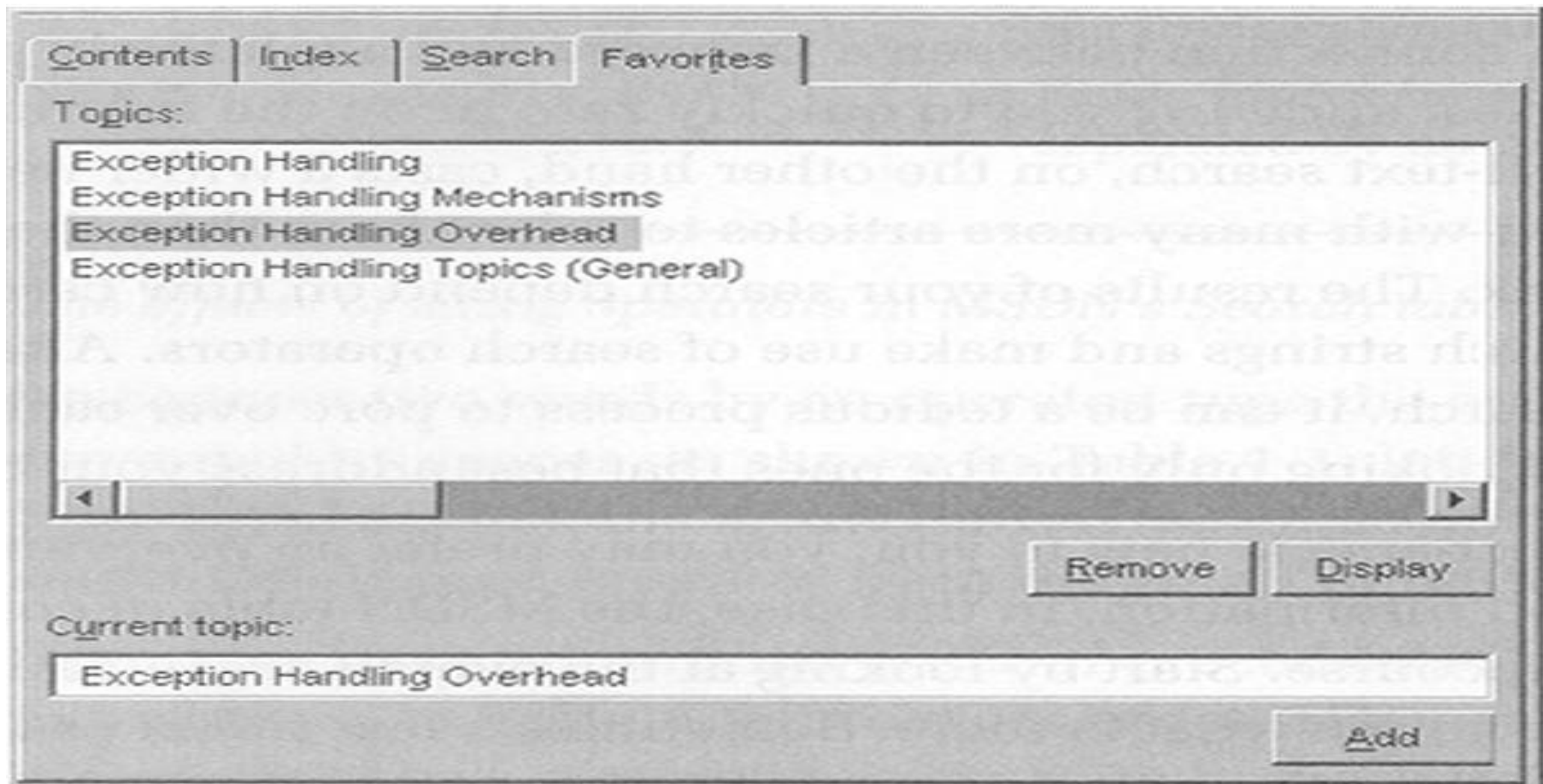


图 1-13 Favorites（收藏）选项卡保持想要重新访问的文章列表

访问万维网

通过嵌入 Internet Explorer 浏览器，MSDN 可以接触到其他 Web 站点上的信息来源，不论它们在哪里。文章可以包含 Internet 地址（通用资源定位符或 URL）作为超文本链接，因此，MSDN 可以像打开其他任何库中的文章一样来打开一个 Web 网页。



图 1-14 通过 MSDN 来访问 Web 站点

要指定目标 Web 网点，从 Go（转向）菜单中选择 URL 命令，并输入网点的网址。图 1-14 演示了一个范例。

在开发环境之外工作

大多数 Visual C++ 工具只有在 Developer Studio 环境内部才是可用的，但编译器、链接器、资源编辑器和生成程序例外。这些程序像 32 位控制台实用程序一样运行。通过编译和链接来生成应用程序时，Visual C++ 用生成程序来执行两个编译器和一个链接器。它们的输出消息通常送到系统的标准输出设备，这些消息被捕获，并显示在环境的 Output（输出）窗口中。不用环境也可以生成应用程序，从命令行执行四个程序 NMake.exe、CL.exe、Link.exe 和 RC.exe 即可。

但在 Developer Studio 之外工作是不切实际的。本章开头的特征列表说明了环境给程序开发带来的好处，尤其（不是唯一）是对使用 MFC 库的 C++ 开发工作。除非你有遗留下来的 C 语言源文件以及还要使用的生成文件，你一定会发现，在环境内部的开发工作更容易，且更具创造性。第 3 章解释如何使用旧的文本编辑器（如果你愿意的话），但 Developer Studio 远不只是一个文本编辑器而已。没有了它，Visual C++ 就去掉了精华部分。本书的每一章都描述了如何使用环境的某些部分来创建和维护 C/C++ 程序。本章仅仅是个开头。

第 2 章 AppWizard（应用程序向导）

Visual C++最显著的技术之一就是它的“向导”。每个向导擅长为一种特殊类型的程序建立项目，它在创建新项目时提供一个很好的开端，这样，你就不必从头做起了。它像其他动态链接库一样，在 Developer Studio 环境下运行，并且要求你提供新程序的特征，然后生成启动器源文件，其中这些新特征的大部分通用代码都已经替你写好了。Visual C++为专业项目，如 ActiveX 控件和 Developer Studio 附加实用程序，提供了各种各样的向导。甚至有向导来帮助你建立自己的定制向导。在后面的章节中，我们会遇到其中一些类型的项目，但本章着重讨论 Visual C++的主向导：AppWizard（应用程序向导）。除了措辞上有所差别外，第 6 版的 AppWizard（应用程序向导）同以前的版本没有什么改变。如果以前你曾经使用过 AppWizard（应用程序向导），可以跳过这一章。

AppWizard（应用程序向导）的优点

AppWizard（应用程序向导）擅长为使用微软基础类库的典型的 C++ Windows 应用程序建立开发项目。

如果你想用 C 来写程序，或不愿意使用 MFC，请忘掉 AppWizard（应用程序向导）。用 Win32 Application Wizard 来代替 AppWizard（应用程序向导），不会对你有什么影响。较早版本的 AppWizard（应用程序向导）是专门为创建基于文档/视图结构的应用程序而设计的，这种结构中程序的数据由文档对象来维护，并通过视图对象提供给用户。MFC 本身就倾向于这样一种程序结构。在 Visual C++ 的较晚版本中，AppWizard（应用程序向导）已经变得更加灵活了，它在没有内置文档支持的时候，也能准备应用程序，这个选项对许多不读取或创建文件的较小的程序来说，是非常合适的。你可以把这种简洁的逻辑形式称为“视图单一结构”，用来代替文档/视图结构，因为这样创建的应用程序包含一个处理显示的视图类，但没有为文档对象提供通信类。AppWizard（应用程序向导）也能创建一个不依赖于文档/视图的基于对话框的应用程序，它通过对话框来与用户交流。第 5 章描述如何在 Visual C++ 中使用 AppWizard（应用程序向导），来创建基于对话框的应用程序，及不使用 AppWizard（应用程序向导）时又如何创建。

在生成的项目中，每个类都有它自己的执行文件和头文件。文件中的全部源代码从空的存根函数到完全成型的程序元素，如工具栏和用户可从 Help（帮助）菜单中调用的 About（关于）对话框，都应该包括在内。AppWizard（应用程序向导）为各种各样的程序特征提供了代码，这些特征包括：

- 单文档、多文档和对话框界面。

- 停靠工具栏、状态栏和打印机支持。
- 带有典型的操作命令，如 Open（打开）、Save（保存）、Print（打印）、Cut（剪切）、Copy（复制）和 Paste（粘贴）的菜单。
- 上下文相关帮助的启动器文件。
- 显示程序信息和 MFC 图标的 About（关于）对话框。
- 数据库支持。
- 对复合文档、Automation 和 ActiveX 控件的 OLE/ActiveX 支持。
- 对消息应用编程接口（MAPI）和 Windows Sockets 的支持。

在这一章中，我们将着眼于如何用 AppWizard（应用程序向导）来创建一个预先载有这些和其他一些特征的新项目。为使你了解 AppWizard（应用程序向导）可节省你多少工作，图 2-1 演示了从 AppWizard（应用程序向导）生成的项目文件中生成的一个典型的应用程序，它只有一个窗口。这个例子中不需要再做其他的编程工作。

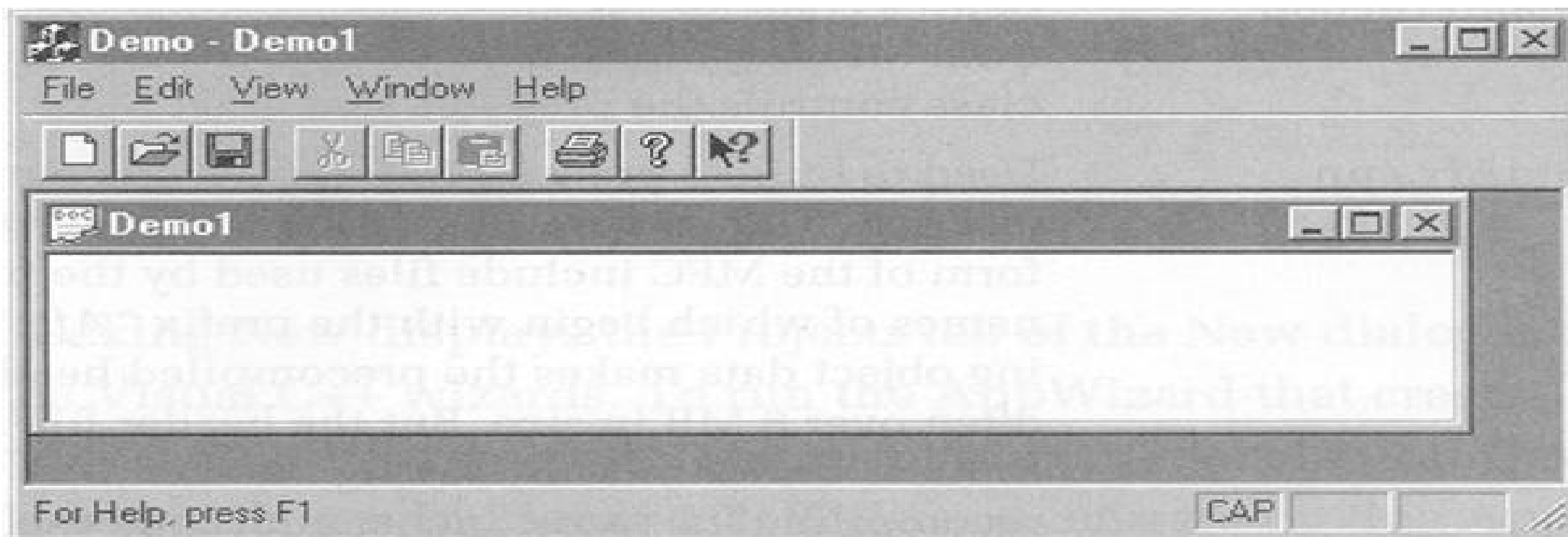


图 2-1 AppWizard (应用程序向导) 创建的基本应用程序

AppWizard (应用程序向导) 仅在建立项目的开头运行一次，它在开始时，提供足够多的选项，但随后就不再提供了。然而，你并不是完全漫无目的地漂游，因为 AppWizard (应用程序向导) 建立项目的方式允许你使用其他的 Visual C++ 工具，如 ClassWizard 来继续开发过程。例如，当查看 AppWizard (应用程序向导) 源文件时，你会注意到一些特殊的注释语句。在第 6 章中，我们将看到 ClassWizard 用这些注释来监控项目的类。

AppWizard（应用程序向导）为项目生成的源文件的数目取决于你指定的特征。
表 2-1 显示了一个典型的列表。表中每个执行文件都有一个相应的同名头文件。

表 2-1 AppWizard（应用程序向导）生成的典型的源文件，斜体的项目表示项目名

文件	说明
<i>project.cpp</i>	主应用程序源文件
<i>ProjectView.cpp</i>	程序的视图类源代码
<i>ProjectDoc.cpp</i>	程序的文档类源代码
MainFrm.cpp	类 CMainFrame 的源代码。这个类是从 MFC 的 CFrameWnd 或 CMDIFrameWnd 派生来的，它控制程序的主窗口

续表

StdAfx.cpp

用来建立一个名为 project.pch 的预编译头文件。预编译头文件采用 MFC 的编译形式，它包括项目引用的文件，这些文件带有前缀“ Afx ”。对象的计算结果数据使预编译头文件很大，通常超过 6 MB。但头文件通过节省对未变动的代码的编译工作，从而显著地缩短了编译时间

project.rc

包含项目的资源数据（在第 4 章中讨论）

Resource.h

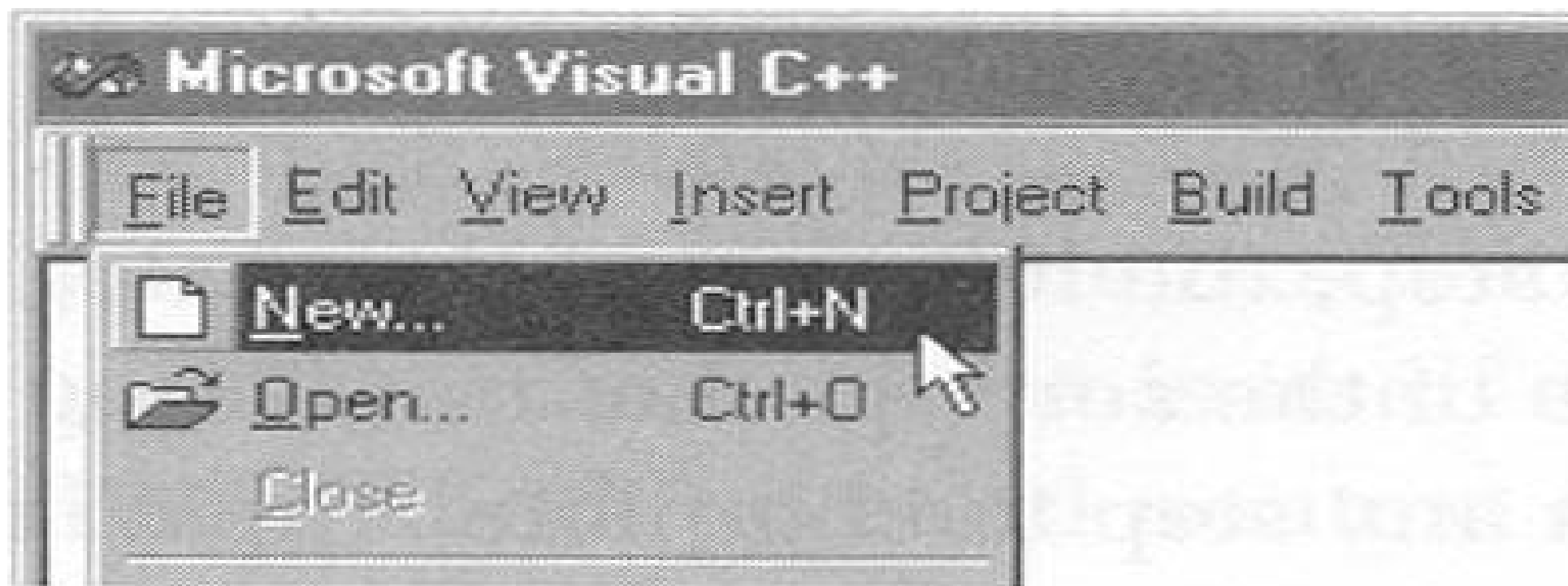
包含 #define 语句来为项目声明常量

刚开始时，你可能试图想放弃 AppWizard（应用程序向导），它对于核心编程人员来说限制太严。而且，如果你有规律地创建一些相同类型的项目，对你来说，通过复制和修改以前的项目源文件来开始一个新项目，比起用 AppWizard（应用程序向导）来创建一套新文件来说，要容易得多。但 Visual C++ 中的向导技术已经成熟到这种地步：它不知何故，看起来相当容易，以致于不用 AppWizard（应用程序向导）简直是一种错误。在不到 60 秒的时间内，你就可以完成一个开发项目的单调乏味的常用设置过程，而直接开始代码开发。并且，你可以依赖 AppWizard（应用程序向导）写的源代码，它保证是没有错误的，这是你在项目

间剪切和粘贴代码所不能保证的。如果你所计划的程序正是 AppWizard（应用程序向导）所指定的类型，请不要犹豫。你通过用 AppWizard（应用程序向导）来建立项目，可以节省大量时间。

运行 AppWizard（应用程序向导）

通过环境的 File（文件）菜单中的 New（新建）命令可开始一个 AppWizard（应用程序向导）项目：



单击 New（新建）显示 New（新建）对话框的 Project（项目）选项卡，其中列出了 Visual C++ 中的向导。要运行向导来为一个典型的 Windows 应用程序创建项目，选择标有 MFC AppWizard(exe) 的图标，如图 2-2 所示。现在我们集中讨论这个 AppWizard（应用程序向导）。通过 MFC AppWizard（应用程序向导）(dll) 图标调用的 AppWizard（应用程序向导），为你建立动态链接库开发项目，下一章我们将讨论这个问题。

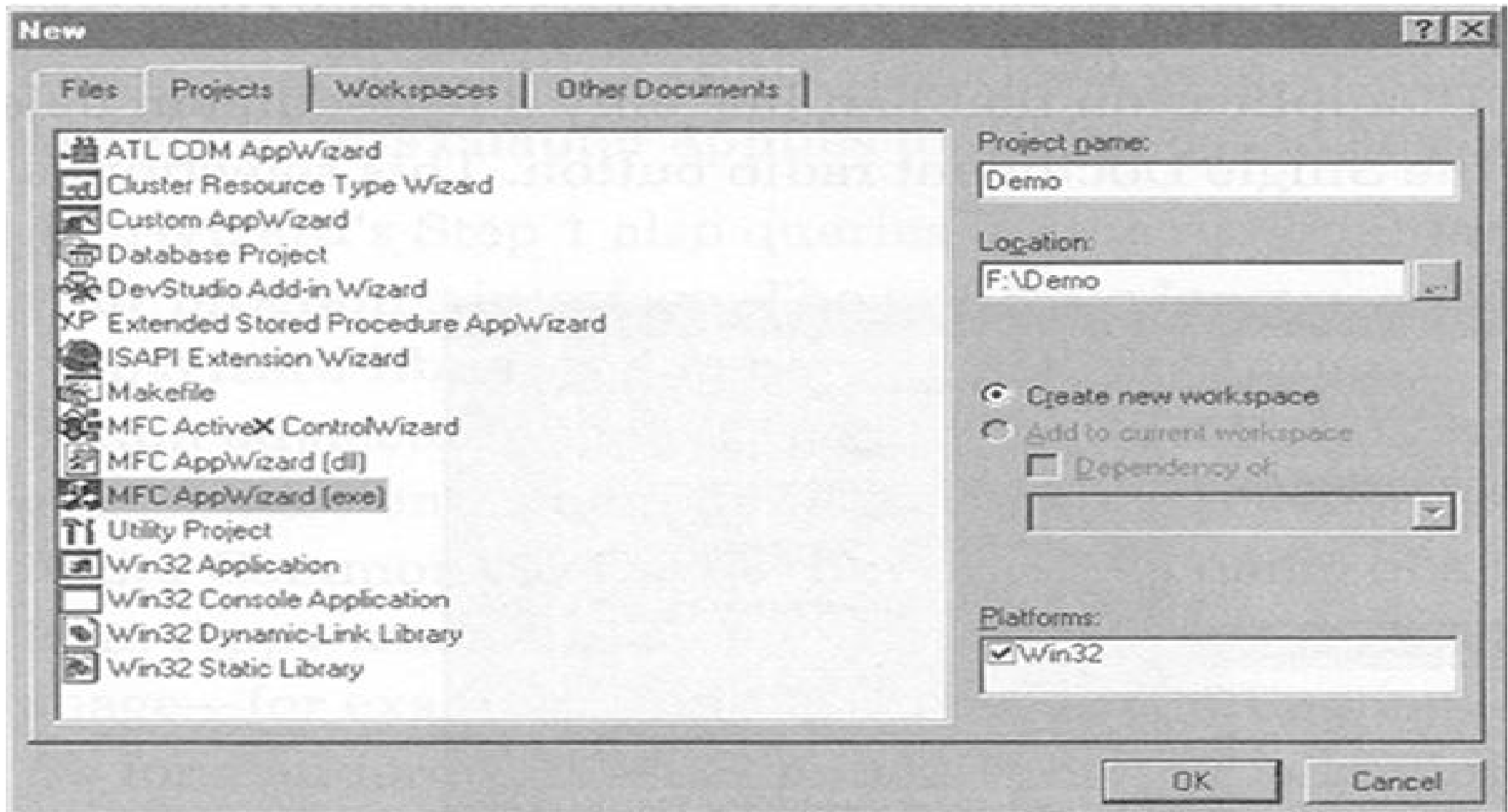


图 2-2 要为典型的应用程序创建项目，需选择 MFC AppWizard(exe)图标

为项目输入一个名字。正如前面提到过的那样，AppWizard（应用程序向导）用项目名来区别项目中不同的文件，因此，项目名要适当地短一些。项目一旦建立以后，就没有办法改变它的名字。默认情况下，Visual C++把 AppWizard（应用程序向导）项目放在 Common\MsDev98\MyProjects 目录下；如果你想用其他目录，在 Location（位置）文本框中指定一个路径。直到你选择列表中的一个图标，并输入一个项目名，OK（确定）按钮才能用。

单击 OK（确定）时，AppWizard（应用程序向导）以对话框的形式提供多达 6 个步骤。在每个步骤中，对话框的左边显示一个图片，来给出设置的视觉提示。在每一步中，单击 Finish（完成）按钮，将结束 AppWizard（应用程序向导），并接受剩下步骤中的默认设置。为了在一系列步骤中前进或后退，单击 Next（下一步）或 Back（后退）按钮即可。

步骤 1：程序界面

如图 2-3 所示，AppWizard（应用程序向导）的第一步指定了应用程序的类型，选择单文档界面(SDI)、多文档界面(MDI)或基于对话框的界面。要创建不需要文档对象来从磁盘文件中读取数据的简单的 Windows 应用程序，把标为 Document/View Architecture Support（文档/视图结构支持）复选项禁用即可。

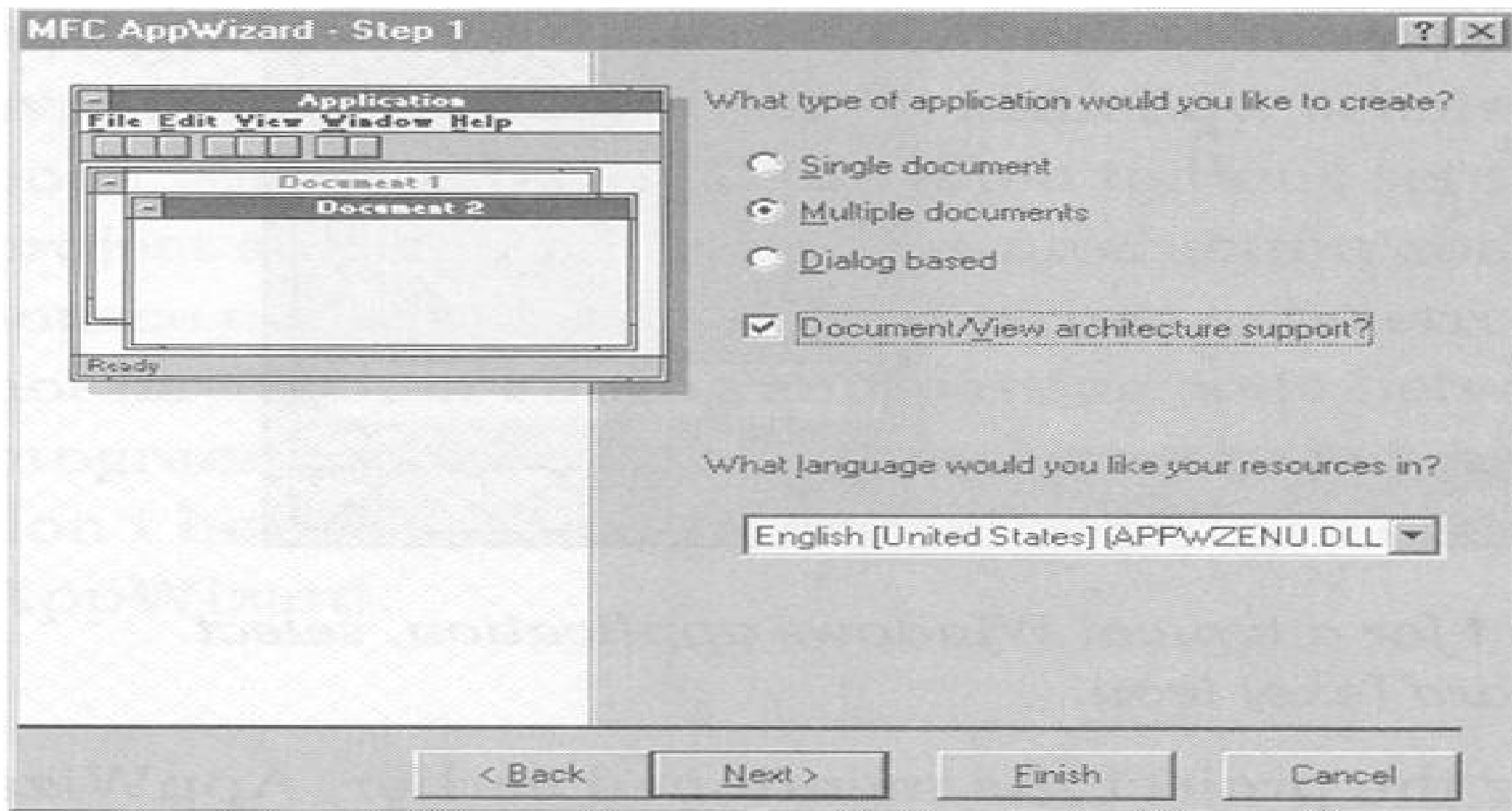


图 2-3 在 AppWizard（应用程序向导）的第一步中选择应用程序的界面

对于一次只处理一个文档对象的 SDI 应用程序，请打开 Single Document（单文档）单选按钮。这个选项对那些不能明确地遵循文档/视图结构的应用程序也是适用的。一个 SDI 应用程序的一般性维护程序，比起相应的 MDI 应用程序来说，要小得多，因此，SDI 应用程序的可执行文件要更小一些。

MDI 应用程序的优点是能一次处理任意数量的文档，它在每个分离的窗口中显示一个文档。用户可以在不同的文档窗口中工作，并把每个文档单独保存为一个文件。在后两章中我们将看到，Visual C++ 本身就是 MDI 应用程序的一个例子，它能在不同的编辑窗口中显示文本或非文本数据。

第三种界面选项创建一个基于对话框的应用程序。这种选择对于那些不需要一个主窗口的小实用程序非常适合，因为用户通过对话框就可与程序交流。基于对话框的界面并不像听起来那样限制严格，第 5 章演示了如何创建显示一个属性表的基于对话框的应用程序，该对话框能接受并显示大量信息。Windows 所带的 Phone Dialer（电话拨号程序）实用程序就是基于对话框的应用程序的一个例子。

因为第 5 章将详细讨论基于对话框的应用程序，这里就不再讨论基于对话框的界面这一选项。然而，本章中大部分信息对基于对话框的应用程序也是适用的。

AppWizard（应用程序向导）的步骤 1 还要求你提供程序界面的语种。可选的语言取决于你安装到系统上的 AppWizard（应用程序向导）库；单击文本框附近的箭头按钮，可显示可选的语言。每种语言都依赖它自己的动态链接库，默认情况下，这些动态库都安装在 Common\MSDev98\bin\ide 文件夹中。库文件名以

Appwzxxx.dll 的形式表示，其中 *xxx* 表示一个三位的语言代码。例如，*enu* 代表美国英语，*deu* 代表德语，而 *fra* 代表法语。图 2-4 显示了在三种不同语言中由 AppWizard（应用程序向导）生成的应用程序的 File（文件）菜单的样子。



图 2-4 在三种不同的语言中的应用程序的 File（文件）菜单

步骤 2：数据库支持

在图 2-5 中所示的 AppWizard（应用程序向导）的第二步中，要求你提供项目所需要的数据库支持种类。这一步和下面的步骤假定你在第一步中选择了带文档/视图支持的 Single Document（单文档）或 Multiple Documents（多文档）选项。

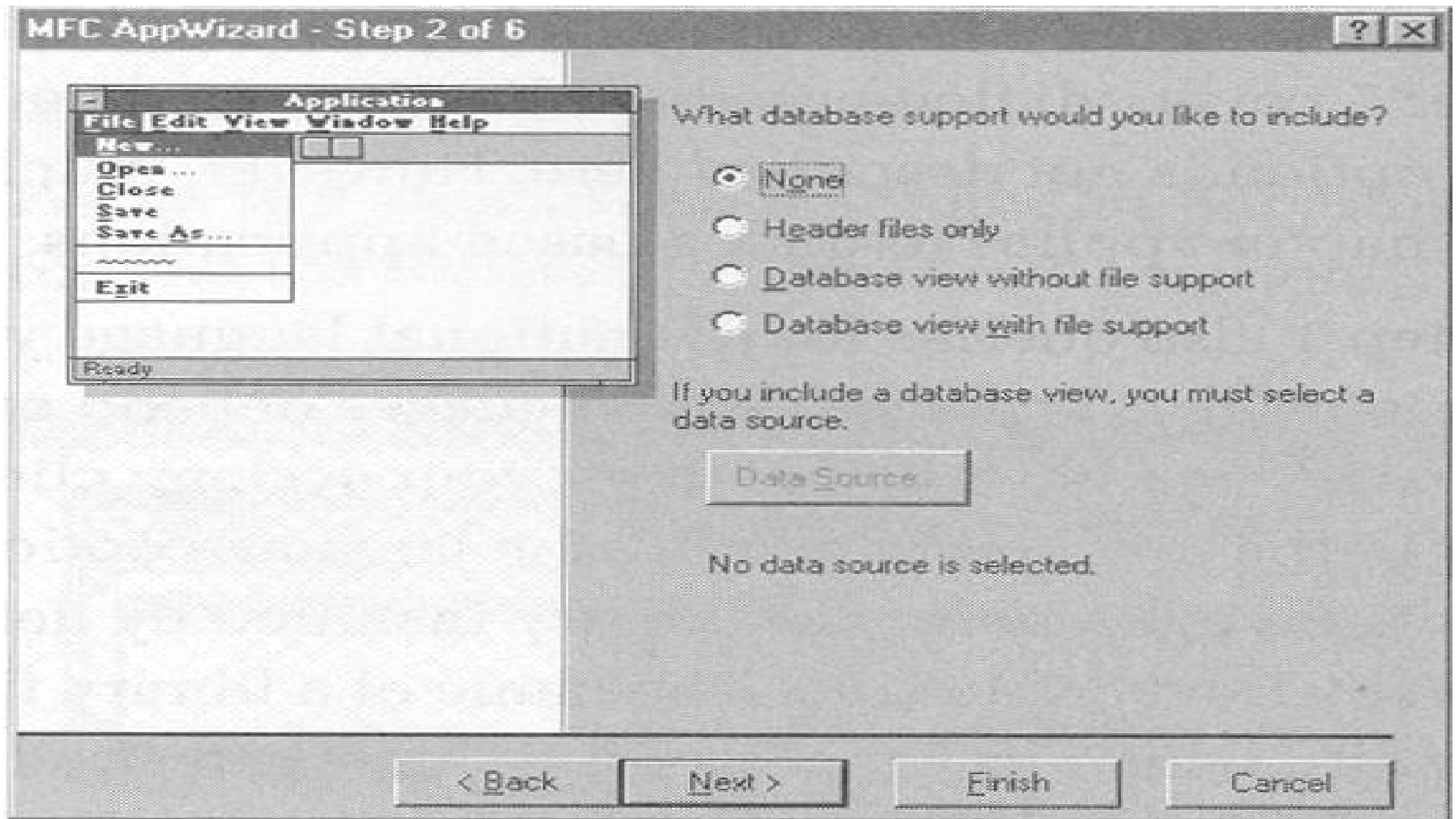


图 2-5 在 AppWizard（应用程序向导）的步骤 2 中选择数据库支持

如果你的项目不使用数据库，单击 **Next**（下一步）按钮跳过这一步，进到步骤 3。如图 2-5 所示，四个单选按钮决定了 AppWizard（应用程序向导）添加到项目的数据库支持程度：

- **None（无）**：在生成项目时，把数据库支持的库文件排除在外。如果你的项目不使用数据库，选择 **None（无）** 单选按钮，来避免在项目文件增加不必要的代码。你可以以后再往项目中添加数据库支持。
- **Header files only（只有头文件）**：在生成过程中，包含数据库头文件和库文件，但 AppWizard（应用程序向导）不为数据库类生成源代码。你必须自己写所有的源代码。这个选项适用于那些开始时不使用数据库，但你打算以后添加数据库支持的项目。
- **Database view without file support（数据库视图，没有文件支持）**：包含数据库头文件和库文件，同时也创建一个记录视图和记录集。生成的应用程序支持文档，但不支持串行操作。
- **Database view with file support（数据库视图，具有文件支持）**：设置同上，除了生成的应用程序既支持数据库文档，又支持串行操作。

如果你用最后两个选项中的一个来包含数据库视图，那么，除非你指出数据源，否则不能继续到下一步。

数据来源

要指定数据来源，单击 Data Source（数据来源）按钮显示图 2-6 所示的 Database Options（数据库选项）对话框。

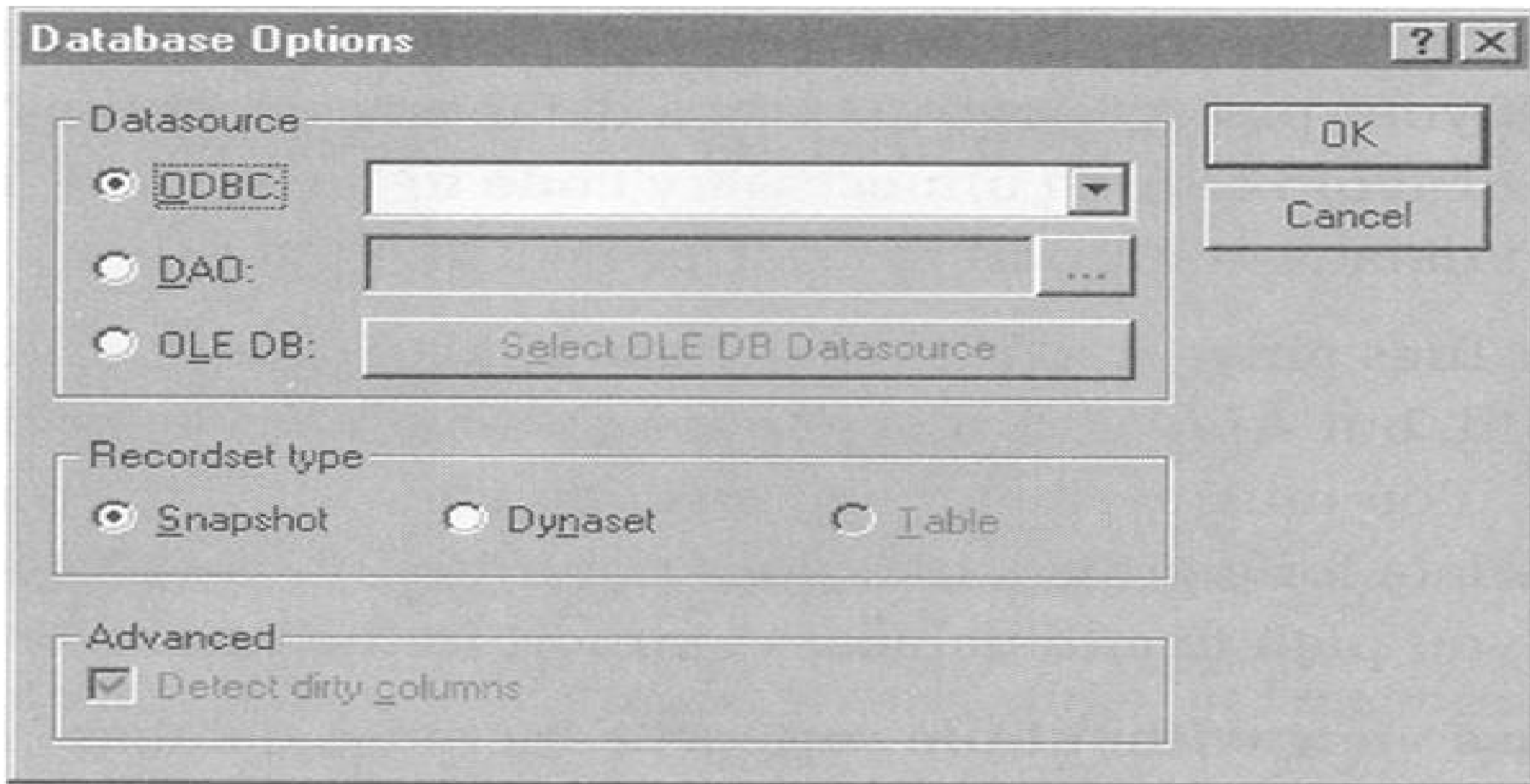


图 2-6 在 Database Options (数据库选项) 对话框中确定数据来源

Database Options (数据库选项) 对话框提示输入数据源是遵循开放式数据库互接

(ODBC)标准，还是微软的数据访问对象(DAO)标准，或 OLE 数据库(OLE DB)标准。ODBC 功能在数据库管理系统，如 Microsoft Access、Oracle 或 dBase 的特定的驱动程序中实现。Visual C++提供了一个 ODBC 驱动程序的集合；其他的可从经销商那里买到。要查看 Visual C++中包含的驱动程序列表，参看标题为“ODBC Driver List”的在线帮助文档。

选择 ODBC 作为程序的数据源类型时，AppWizard（应用程序向导）生成代码来调用“ODBC 驱动管理”，并通过它来调用合适的驱动程序。驱动程序用结构化查询语言(SQL)来同目标数据库管理系统发生联系。ODBC 支持保证应用程序能访问不同格式和不同结构的数据。

选择 ODBC 会激活一个列有 ODBC 数据源管理器中注册过的所有数据来源的下拉列表。数据源既包含数据，也包含访问数据所需的信息。要注册数据源，或取消数据源的注册，在 Control Panel（控制面板）中双击 32 位 ODBC 图标来运行管理器。在安装过程中，Visual C++通常会装入数据源管理器，但如果你是定制安装 Visual C++的，你的系统中可能会没有数据源管理器。如果 Control Panel（控制面板）中没有 32 位 ODBC 图标，需重新运行 Visual C++安装程序，并安装必要的 ODBC 数据库支持文件。

DAO 是微软产品如 Access 和 Visual Basic 的标准。DAO 用 Microsoft Jet 数据库引擎来提供一套访问对象，包括数据库对象、表定义和查询定义对象及记录集对象。尽管 DAO 在 MDB 文件（例如那些由 Microsoft Access 创建的文件）上工作

得最好，一个 DAO 程序也能通过 Microsoft Jet 来访问 ODBC 数据源。

OLE DB 是一种新的数据访问方法，它允许一个称为“使用者”的客户应用程序从任何装备了被称为“供应者”的数据翻译器的数据源中取回数据。“供应者”对应用程序来说，就像一组组分对象模型(COM)对象，它通常不创建数据，而作为一个中间人以自己所带的格式访问数据，并以可识别的形式把数据传递给“使用者”。图 2-7 说明了“使用者”如何同“供应者”(而不是同原始的数据源创立者)通信。

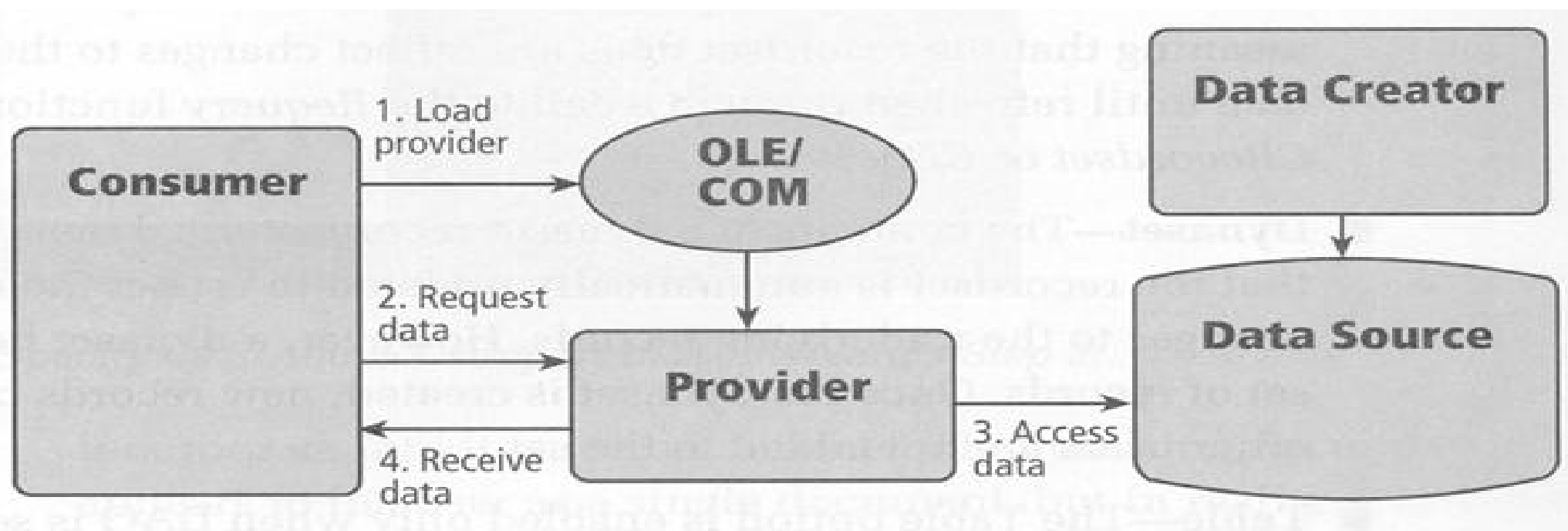


图 2-7 在 OLE DB 中，在数据客户与提供者之间的典型交互

OLE DB 的优点是在“使用者”和“供应者”间不需要任何关于数据格式的预定协议。至少“供应者”负责把数据翻译成“使用者”理解的形式，通常是表格式的格式。“供应者”也能对未加工的数据进行改进，例如，按照指定的标准进行查询处理和排序。在 AppWizard（应用程序向导）的 Database Options（数据库选项）对话框中选择 OLE DB 选项，是创建一个使用数据的应用程序（而不是数据供应者）的第一步。这个选项从称为“OLE DB 使用者模板”的类库中生成代码，该类库提供了对 OLE DB 类对象，例如 CDataSource 和 CSession 的包装。Visual C++ 提供了另一个向导（ATL COM AppWizard）来帮助编写“供应者”应用程序。第 10 章对 ATL COM AppWizard 和 Active Template Library 有更多的描述，尽管它是从编写 ActiveX 控件的角度，而不是从 OLE DB 供应者的角度上来说的。

记录集类型

通过在 Database Options（数据库选项）对话框中的 Recordset Type（记录集类型）部分中选择三个单选按钮中的一个，可以指定程序使用的记录集类型。三个单选按钮作用描述如下：

- **Snapshot（快照）**：快照记录集保持快照创建时的数据视图。快照是静态的，这意味着，记录集不会反映出原始数据的改变，除非通过调用类 CRecordset 或 CDaoRecordset 的 Requery 函数来刷新它。

- **Dynaset（动态集）**：动态记录集的内容是动态的，这意味着，记录集自动更新，来反映记录的最新改变。然而，动态设备保持一套固定的记录。一旦动态集建立，其他用户创建的新记录就不会添加到这个集合中去。
- **Table（表）**：Table（表）选项只有数据源类型选择为 DAO 时才是可用的。该选项允许程序使用 DAO 对象以表的形式来操作数据。当选择 Table(表)单选按钮,并单击 OK(确定)按钮,来关闭 Database Options（数据库选项）对话框时，另一个对话框就会出现。在这个对话框中，你可以选择程序使用的列表。

步骤 3：OLE 和 ActiveX 支持

在 AppWizard（应用程序向导）的第三步(图 2-8)中，设置程序的 OLE 和 ActiveX 支持类型。对话框上半部分的五个单选按钮控制着 AppWizard（应用程序向导）添加到程序中的复合文档支持的类型。下面是这些选项的说明：

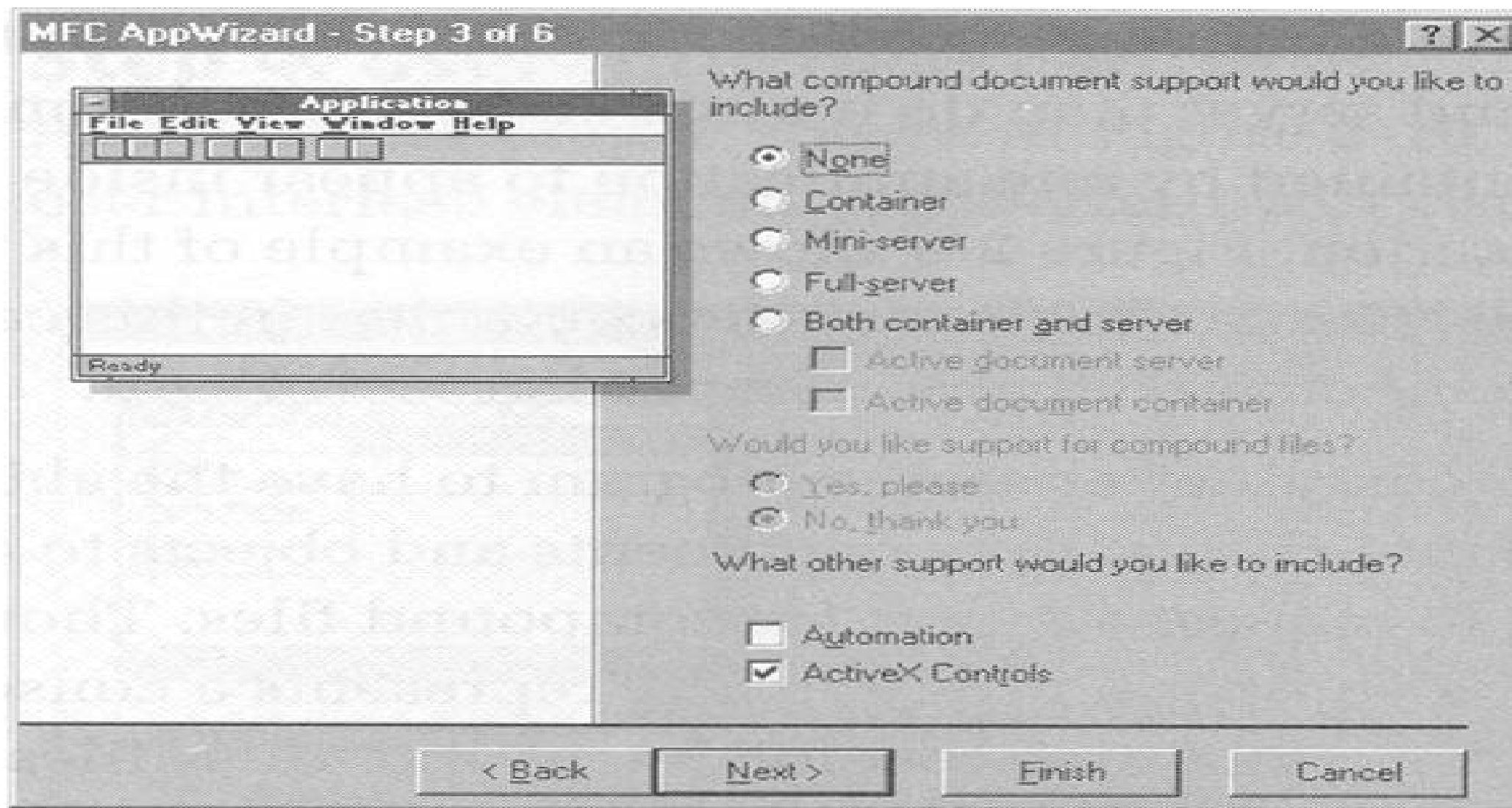


图 2-8 在 AppWizard (应用程序向导) 的第三步中指定 OLE、ActiveX 支持

- **None（无）**：AppWizard（应用程序向导）不为复合文档支持生成任何代码。
- **Container（包容器）**：AppWizard（应用程序向导）创建能包含链接和嵌入对象的程序。
- **Mini-server（微型服务器）**：程序充当一个微型服务器，它能创建复合文档对象，集成应用程序能把这些对象合并到自己的文档中。这样的文档对用户来说，就像单一文档一样，但实际上它是由不同的来源形成的。微型服务器直接把数据写在集成应用程序的文档中，而不是写在磁盘文件中，因此，对微型服务器创建的对象集成应用程序能嵌入，但不能链接。微型服务器应用程序不能作为独立的程序运行，而必须由集成应用程序来启动。Microsoft Draw 就是微型服务器的一个例子。
- **Full-server（全服务器）**：AppWizard（应用程序向导）创建的程序能充当一个全服务器应用程序，它具有微型服务器的全部特性，再加上附加的性能。像微型服务器一样，全服务器应用程序能由集成应用程序来启动，但它也能作为一个独立的 Windows 应用程序来运行。AppWizard（应用程序向导）增加了对把数据存储到磁盘文件中的支持，因此，全服务器应用程序能像支持嵌入一样支持链接。
- **Both container and server（包容器和服务器）**：AppWizard（应用程序向导）生成代码使程序既能像集成应用程序那样嵌入对象，又能像服

务器应用程序那样提供对象。

通过集成应用程序或服务器应用程序来选择一个复合文档选项，会让你为活动文档选择附加支持。活动文档在客户和服务器之间提供一个比普通的嵌入文档更高层次的集成，它允许由一个应用程序维护的文档出现在另一个应用程序的窗口中。图 2-9 显示了这种集成的一个例子，其中 Internet Explorer (本例中的包容器) 打开了一个由服务器 Microsoft Word 创建的文档。

如果你希望你的包容器或服务器程序能串行化复合数据，也就是说，把文档和对象保存到磁盘上，打开单选按钮，来请求对复合文件的支持。尽管从概念上来说，复合文档是单纯的文档，实际上，它代表不同文件的组合，一个文件包含文档，而其他文件则包含链接到文档的对象。当保存复合文档时，包容器负责把自己的文档对象写到磁盘上。然后它给服务器传递一个请求，来把包容器正在使用的各个对象保存到同一个“存储器”中。

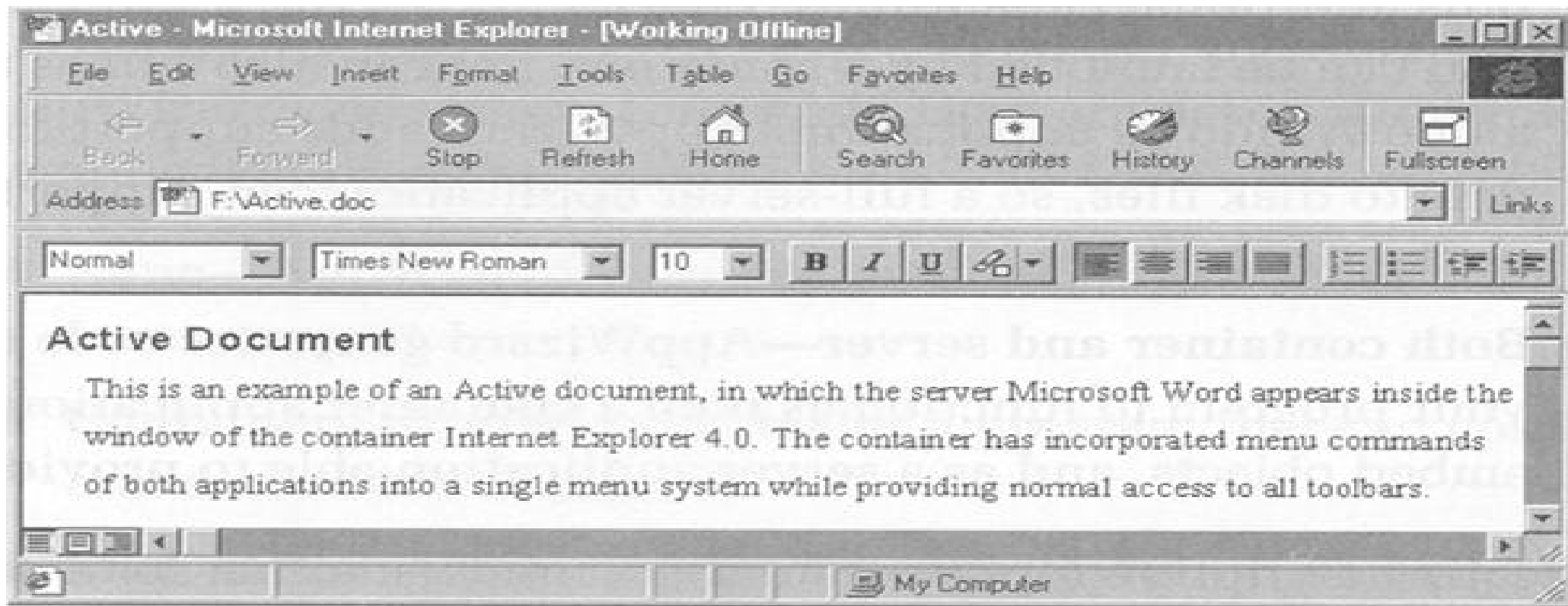


图 2-9 Active 文档容器和服务器的例子

第三步的对话框底部的两个复选项询问,是否要求 Automation 和 ActiveX 控件支持。默认情况下, AppWizard (应用程序向导) 激活 ActiveX Controls (ActiveX 控件) 选项;如果程序不会嵌入 ActiveX 控件,请清除该复选项。这个决定并不是不可取消的,以后,你可以轻易通过包含一行代码,来向 MFC 程序中添加 ActiveX 控件支持。要了解如何向已存在的 MFC 程序中添加 ActiveX 控件支持,

请参阅第 8 章。

步骤 4：用户界面特征

在图 2-10 中所示的 AppWizard（应用程序向导）的第四步中可以控制 AppWizard（应用程序向导）为程序创建哪些用户界面元素。

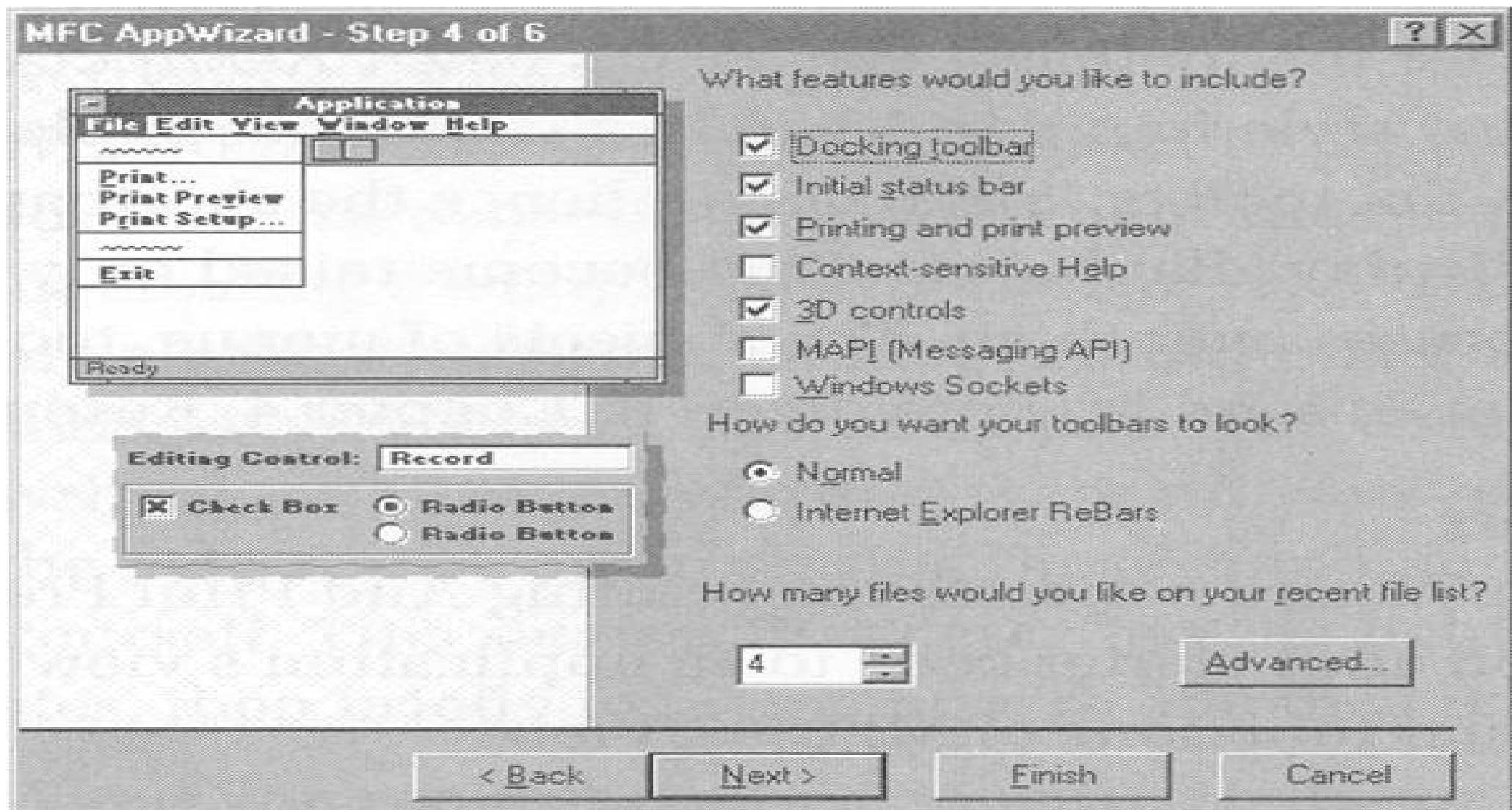
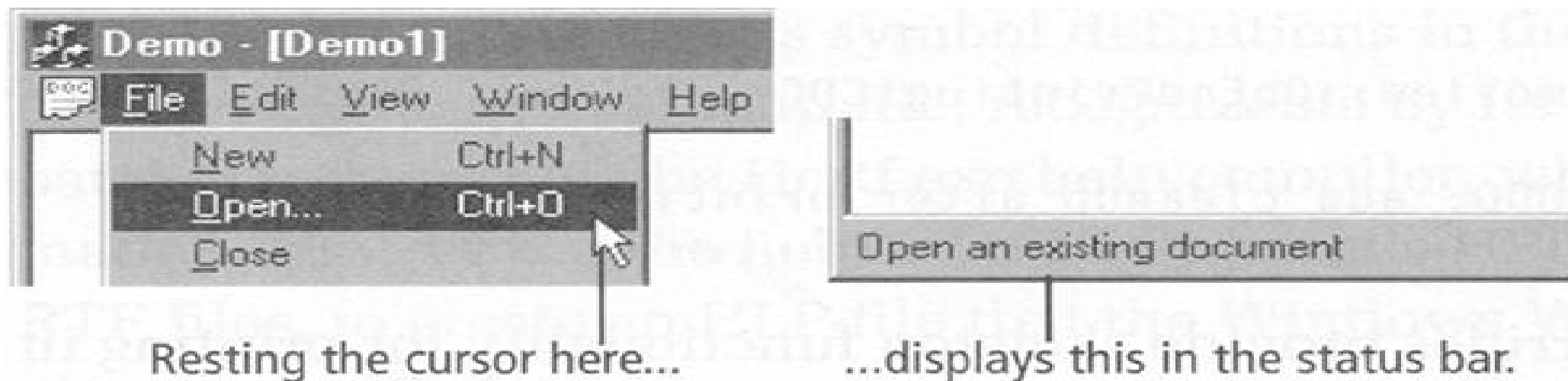


图 2-10 在 AppWizard (应用程序向导) 的第四步中选择程序的用户界面特征

AppWizard（应用程序向导）自动为程序主窗口的菜单系统、工具栏及状态栏生成代码和数据。工具栏包含与菜单命令效果一样的按钮，而状态栏则显示命令和工具栏按钮的描述信息。当光标在菜单命令或工具栏按钮上停留一段时间后，帮助信息会出现在状态栏上，如下图所示：



没有选中命令时，状态栏显示诸如“Ready”、“For Help, press F1”或任何其他信息。状态栏也包括了键盘的 Caps Lock、Num Lock 和 Scroll Lock 键的指示器。当程序运行时，MFC 框架自动更新指示器，因此，你不需要添加其他代码来并入这一特征。

Normal（常规）和 Internet Explorer Rebars 单选按钮为应用程序的工具栏提供两种不同的风格。选择 ReBars 选项，会用 MFC 的新的 CReBar 类来为工具栏生成代码，最终会生成 Visual C++、Internet Explorer 及其他应用程序中的平面的工具

栏风格。平面的工具栏是可以调整大小的，因此使用速记词“rebar”，只有当鼠标移到按钮上面时，该按钮才凸起。菜单、工具栏和状态栏的主题将在第4章详细讨论。

打印支持

默认情况下 AppWizard（应用程序向导）激活 Printing And Print Preview（打印和打印预览）复选项。这个选项向应用程序的视图类中添加启动器代码，它跨过了 MFC 的 Cview 的三个虚拟函数：

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//CDemoView printing
```

```
BOOL CDemoView::OnPreparePrinting(CPrintInfo*pInfo)
```

```
{
```

```
    // default prepartation
```

```
    return DoPreparePrinting(pInfo);
```

```
}
```

```
void CDemoView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo/>)
```

```
{  
  
    //TODO: add extra initialization before printing  
  
}  
  
void CDemoView::OnEndPrinting(CDC* /*pDC*/,CPrintInfo* /*pInof*/)   
  
{  
  
    // TODO:and cleanup after printing  
  
}
```

这些为在文档/视图程序中打印提供了功能框架，但在程序打印文档之前，你还需要做更多的工作。要更好地了解如何向 MFC 程序中添加打印功能，参阅 Jeff Prosise 的《Programming Windows 95 with MFC》的第 10 章。MSDN 在线帮助也提供了由“Printing and Print Preview Topics”开始的一系列主题信息。选中 Search Titles Only 复选项，并在 MSDN 的 Search（搜索）选项卡中键入该标题，即可找到该主题。

在线帮助

激活 Context-Sensitive Help（上下文相关）复选项，会告诉 AppWizard（应用程

序向导) 你希望程序提供在线帮助。AppWizard (应用程序向导) 会向项目中添加源代码和一系列文件来帮助你创建一个完整的帮助系统。AppWizard (应用程序向导) 会仔细地说明它添加到程序中的所有的命令和工具栏按钮, 例如 New、Open、Cut 和 Paste。这些说明清晰完整, 你不需要再做更多的工作, 只需要通过说明那些自己添加到程序中的命令来加强帮助文件。本部分首先描述了 AppWizard (应用程序向导) 创建的帮助系统, 然后简要解释如何用自己的帮助文本来加强帮助系统。

当程序需要上下文相关帮助时, AppWizard (应用程序向导) 在项目文件夹中创建一个名为 HLP 的子文件夹。HLP 子文件夹中的文件之一是名为 AfxCore.rtf 的主题文件。如果程序要求打印支持, AppWizard (应用程序向导) 还添加另一个名为 AfxPrint.rtf 的主题文件。以多信息文本格式写的 AfxCore.rtf 和 AfxPrint.rtf 包含了描述 AppWizard (应用程序向导) 给项目添加的特征的帮助文本。HLP 子文件夹也包含一个与该项目同名, 而扩展名为 HPJ 的帮助项目文件。当你生成项目时, Visual C++ 在启动编译器之前, 先执行名为 MakeHelp.bat 的批处理文件。MakeHelp.bat 运行 Makehm.exe Help Maintenance 实用程序, 它读入项目的 Resource.h 文件中的标志定义, 并创建扩展名为 HM 的帮助映像文件。然后该批处理文件运行 Hcrtf.exe 帮助编译器, 来收集取自帮助映像文件、项目的 HPJ 文件及 RTF 文件中的文本, 它用这些来创建一个 Windows WinHlp32 帮助文件查看器能读取的 HLP 文件。

AppWizard (应用程序向导) 也写一些源代码来运行 WinHlp32, 并加载项目的

HLP 文件来响应用户的帮助请求。整个帮助界面是通过在项目的 MainFrm.cpp 文件中将四个入口添加到消息映像中实现的，如下所示：

```
BEGIN_MESSAGE_MAP(CMainFrame,CFrameWnd)

    .
    .
    .

    ON_COMMAND(ID_HELP_FINDER,CFrameWnd::OnHelpFinder)
    ON_COMMAND(ID_HELP,CFrameWnd::OnHelp)
    ON_COMMAND(ID_CONTEXT_HELP,CFrameWnd::OnContextHelp)
    ON_COMMAND(ID_DEFAULT_HELP_ CFrameWnd::OnHelpFinder)

END_MESSAGE_MAP()
```

映像中的每个入口指向由 MFC 框架提供的三个函数之一。调用这些函数用以响应不同的事件，每个函数都调用 WinHlp32，并显示项目帮助文件中的相应文本。

下面的列表描述了什么时候调用函数：

被调用的函数	当用户做如下事情的时候
OnHelpFinder	从 Help(帮助)菜单中选择 Help Topics (帮助主题) 命令时
OnHelp	按 F1 键来接受关于当前上下文的帮助时
OnContextHelp	按 Shift+F1 或单击工具栏上的 Help(帮助) 按钮时

从 Help (帮助) 菜单中选择 Help Topics (帮助主题) 命令，会在 WinHlp32 查看器中显示一个典型的 Help Topics (帮助主题) 对话框，如图 2-11 所示。用户可以浏览 Help Topics (帮助主题) 对话框来寻找想得到的主题帮助。

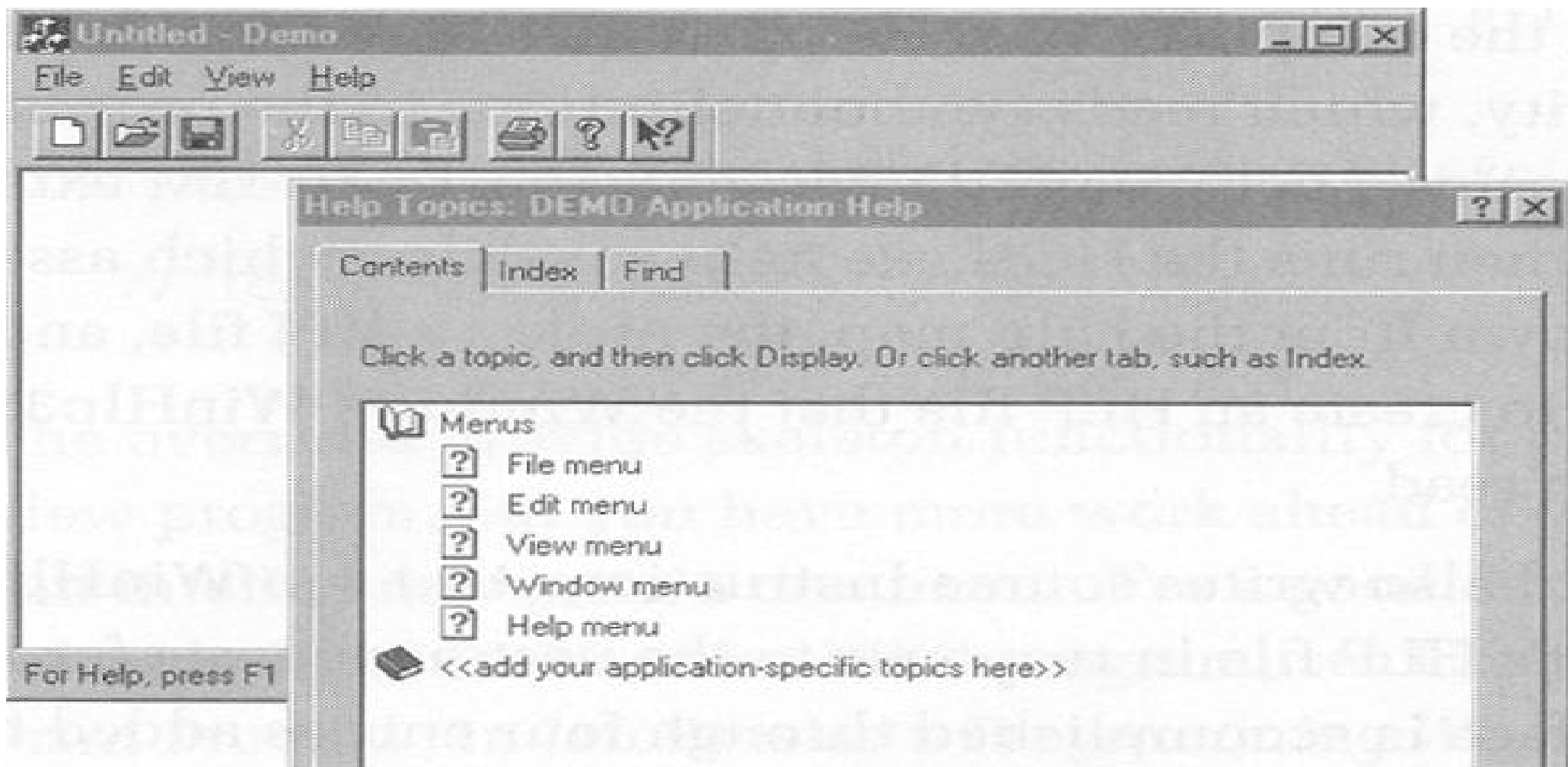
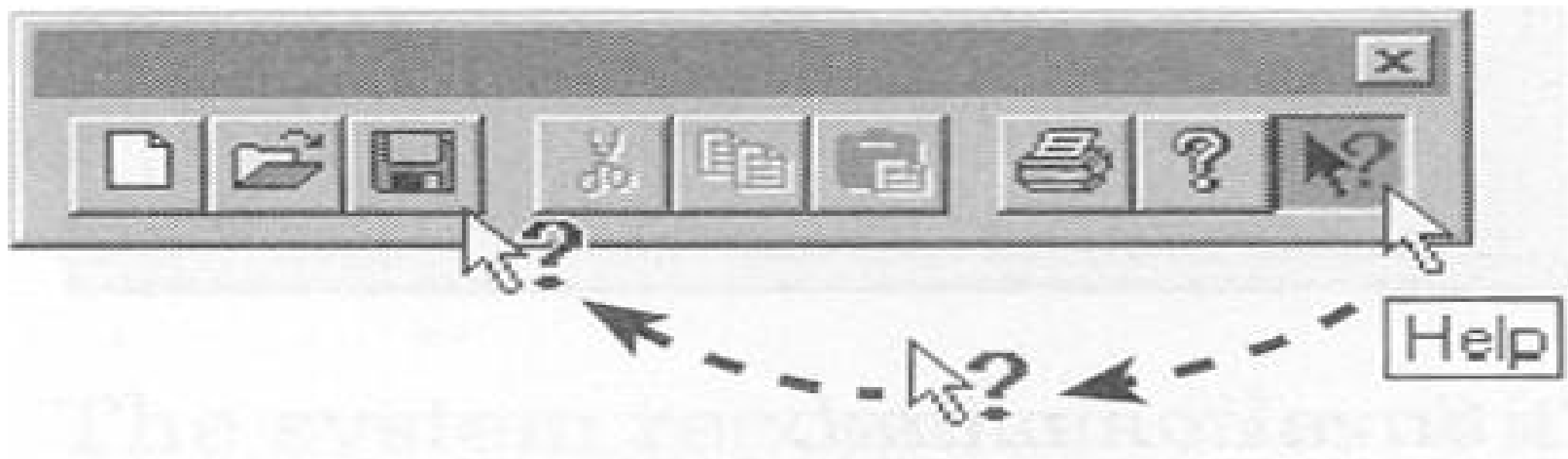


图 2-11 Help Topics（帮助主题）对话框

AppWizard（应用程序向导）在程序的工具栏上也添加了一个 Help（帮助）按钮，它同 Developer Studio 对话框右上角的小问号按钮相类似。单击 Help（帮助）按

钮，会把鼠标形状改变为一个带问号的箭头。然后，用户可以单击程序窗口的任何部分，包括菜单命令、状态栏和工具栏按钮：



调用 Help（帮助）工具，将引起程序执行 WinHlp32，它显示一个描述所单击元素的帮助窗口。例如，通过像上面所说的那样单击 Save 按钮以获取帮助时，会显示图 2-12 所示的帮助窗口。

通过描述其他程序自身的特征，可以加强帮助系统，在一个能识别多信息文本格式的字处理程序中加载 AfxCore.rtf。不要使用 Windows 所带的 WordPad 实用程序。尽管 WordPad 能读取多信息文本文档，但它不保存帮助编译器所需要的信息。多信息文本文档也是以普通的 ASCII 格式写的，因此，在紧急时候你可以通

过文本编辑器来作小小的改动。

创建自己的帮助文本的第一步是搜索字符串“ <<YourApp>> ”，并用应用程序名来取代它。双击文档中的角括号(<<>>)，会装入提示你应添加的文本类型的占位符文本。用新的文本来取代提示和角括号。去除应用程序不用的所有主题，并从AppWizard（应用程序向导）放在文件中的文本中取出对重要格式的提示。文件中的主题必须由硬分页符分隔开。

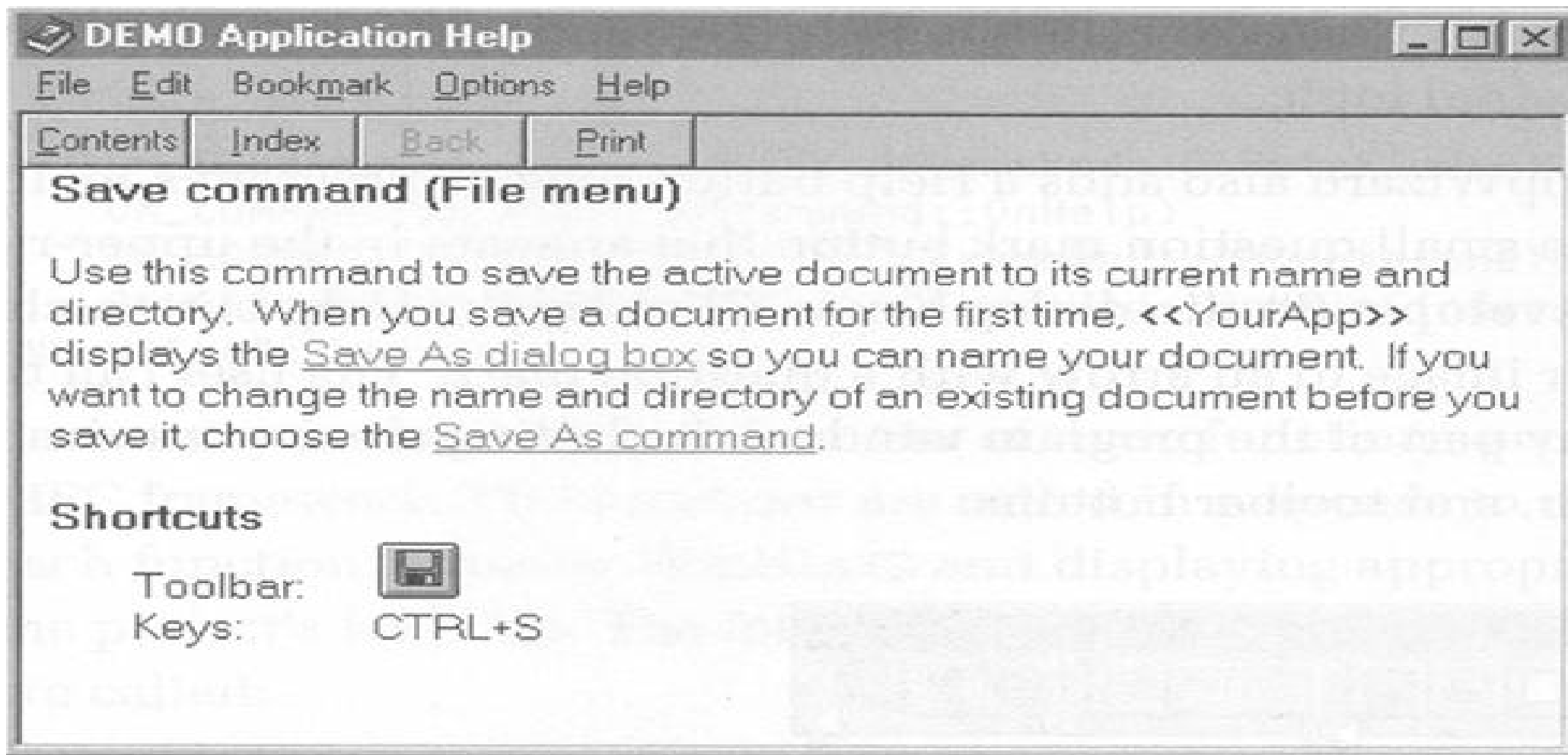


图 2-12 程序的 Save（保存）命令的帮助窗口

写作帮助文件这个主题范围广泛，这里不可能将它完全说清楚。在线帮助描述了如何用 Visual C++ 提供的 Help Workshop 实用程序来构造由 AppWizard（应用程

序向导) 生成的帮助系统。

Advanced (高级) 按钮

在 AppWizard (应用程序向导) 第四步的对话框的右下角, 是一个 Advanced (高级) 按钮, 单击它, 将显示一个标题为 “Advanced Options (高级选项)” 的两页的对话框。在第一页 Document Template Strings (文档模板字符串) 中, 可以改写储存在 Windows 和 MFC 框架使用的程序数据中的某些字符串。如果在第三步中选中了 Active Document Server (活动的文档服务器) 复选项, 你必须在 Document Template Strings 页中为应用程序的文档文件指定一个文件扩展名。在该页的第一个编辑框中键入扩展名, 如下页图所示。

系统把带这种扩展名的文件识别为属于你的服务器应用程序的文件, 这种方法与它用 Microsoft Word 来关联 DOC 文件的方法一样。AppWizard (应用程序向导) 用适合于应用程序的字符串来填写剩下的编辑框, 你可以接受它们, 或按你的想法来修正它们。我们将在第 4 章中详细讨论这些字符串。

Advanced Options



Document Template Strings

Window Styles

Non-localized strings

File extension:

File type ID:

Localized strings

Language:

English [United States]

Main frame caption:

Doc type name:

Filter name:

File new name (short
name):

File type name (long
name):

Close

对话框的第二页 Window Styles（窗口风格）允许你控制程序主窗口的外观，而且，如果你在第一步中选择了 Multiple Document（多文档）选项，在这里还可以控制程序的文档窗口的外观。选择 Window Styles（窗口风格）页顶部的 Use Split Window（使用分隔窗口）复选项，会在 SDI 项目的 MainFrm.cpp 文件中添加下面这个函数：

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
                                CCreateContext* pContext)
{
    return m_wndSplitter.Create(this,
                                  2, 2,
                                  Csize(10, 10),
                                  pContext);
}
```

MDI 项目在它的 ChildFrm.cpp 文件中接收到类似的 OnCreateClient 函数。对于 SDI 应用程序，OnCreateClient 函数启用程序主窗口中的分隔条。程序的 View（查

看) 菜单中的 Split (分隔) 命令打开分隔条, 它允许程序在同一窗口的一个、两个或四个窗格中显示数据。对于 MDI 应用程序, AppWizard (应用程序向导) 把 Split (分隔) 命令放在程序的 Window (窗口) 菜单中, 它控制每个子窗口的分隔条。图 3-4 显示了 Visual C++ 文本编辑器中使用分隔条的例子。

步骤 5: 使用 MFC 库

图 2-13 所示的 AppWizard (应用程序向导) 的第五步询问你想创建的程序的风格、你是否需要附加的源代码注释, 及你希望程序如何链接到 MFC 库。

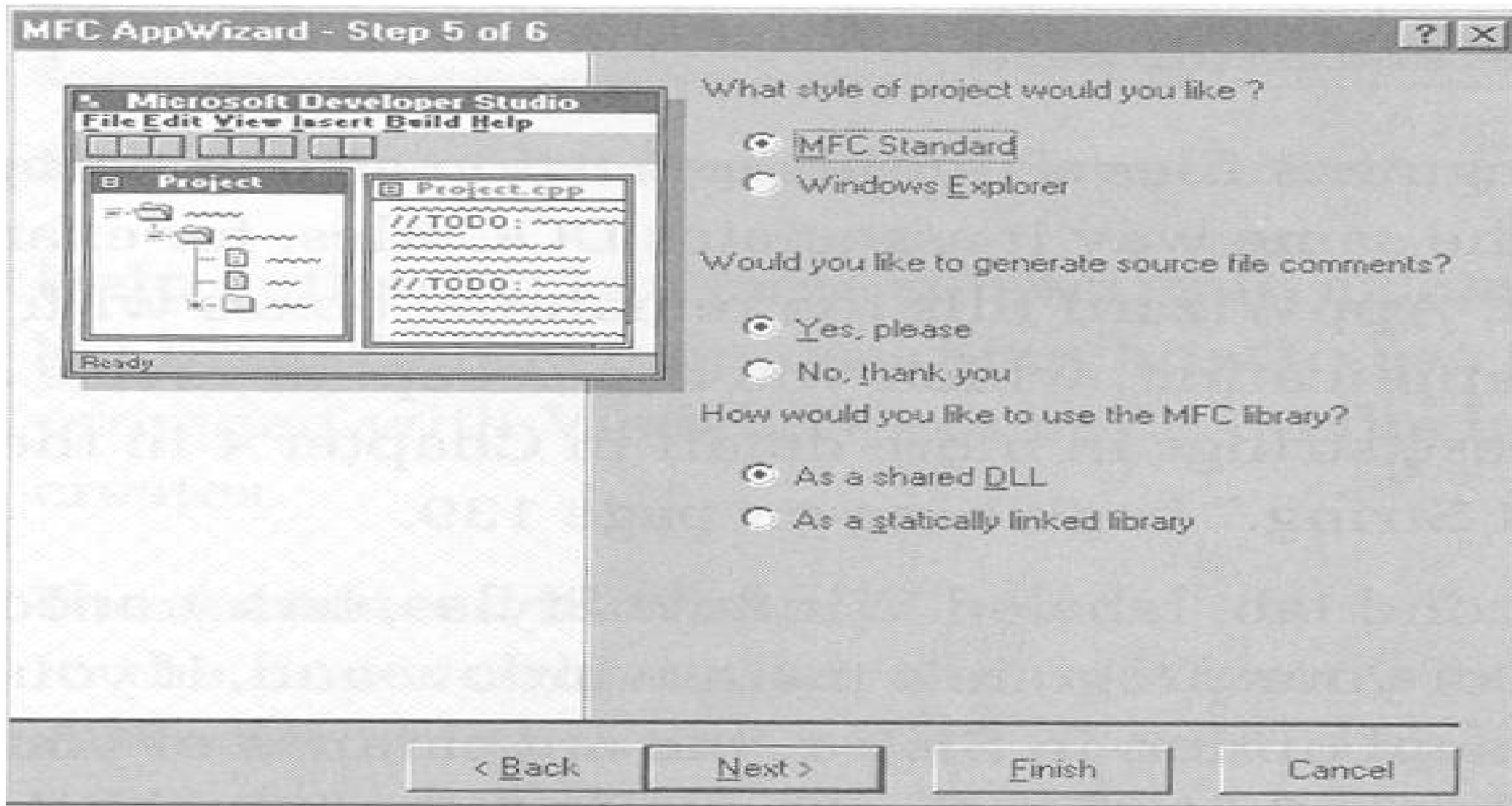


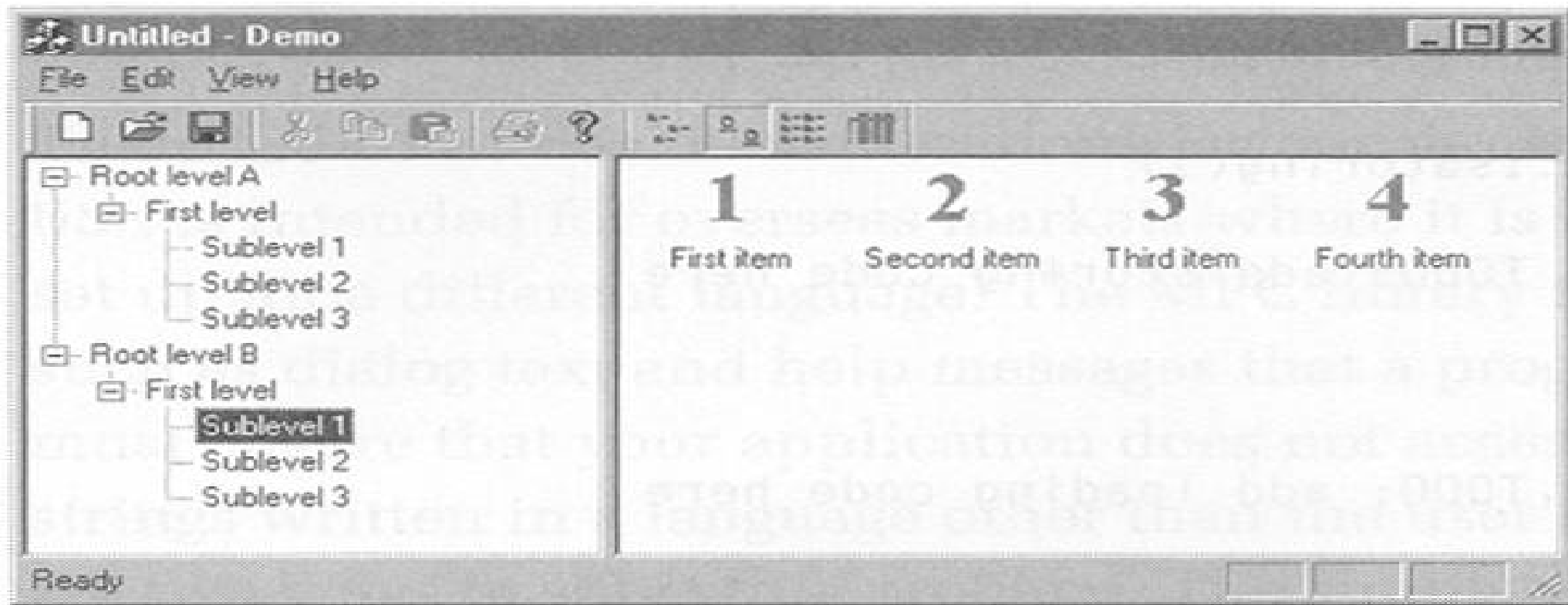
图 2-13 选择源文件注释和 MFC 库选项

项目风格

AppWizard（应用程序向导）提供了两种程序风格，它们由步骤 5 对话框顶部的两个单选按钮来决定。默认按钮为 MFC Standard（MFC 标准），对于创建带有从 CView 派生而来的视图类的普通的 Windows 应用程序，它是正确的选择。本书中后面的章节在用 AppWizard（应用程序向导）来创建范例程序时采用 MFC Standard（MFC 标准）设置，因此，这里花些时间来讨论一下第二个单选按钮的作用是值得的。按钮标题为 Windows Explorer（Windows 资源管理器），是因为这时 AppWizard（应用程序向导）创建的应用程序有着与大家熟知的 Windows 所带的 Explorer（资源管理器）实用程序相似的外观和用户界面。

Explorer（资源管理器）形式的应用程序的主窗口被分隔成相邻的两个窗格，每一个窗格显示不同的内容，而且每一个窗格都由它自己的类来控制。左窗格的视图类派生自 MFC 的 CTreeView，它使该窗格适合于显示一个通过树形层次相关联的条目列表，例如公司的职员列表、家谱表或硬盘上的目录和文件列表。对应于右边窗格的视图类派生自 CListView，它被设计用来显示在某些方面属于左边窗格当前选中的选项的条目列表。像 Explorer（资源管理器）一样，程序的工具

栏包含了四个调整窗格外观的附加按钮，它允许用户选择大图标或小图标的不同排列。下面显示了一个典型的 Explorer（资源管理器）形式的应用程序的外观：



左边窗格中包含的树形视图控制中的条目有小的加号和减号来展开或折叠列表。应用程序通过在窗口的 `CREATESTRUCT` 结构中设置风格标志，来添加按钮和分层线：

```
BOOL CLeftView::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
```

```
cs.style |= TVS_HASLINES | TVS_HASBUTTONS | TVS_LINESATROOT;  
return CTreeView::PreCreateWindow(cs);  
}
```

我们还没有讨论过使你能检查和修改程序的 Visual C++ 编辑器，但如果你对回顾上述简单程序的源代码感兴趣，可以在附带光盘的 Code\Chapter.02\Demo 文件夹下找到这些项目文件。Demo 项目是用 AppWizard（应用程序向导）来创建的，它在第一步中选择了 Single Document（单文档）选项，而在第五步中选择了 Windows Explorer（资源管理器）选项，并且接受了剩下的 AppWizard（应用程序向导）的默认选择。Demo 是一个基础项目，它仅创建了由 AppWizard（应用程序向导）生成的 Explorer（资源管理器）形式的应用程序所需的典型的代码。第 3 章和第 4 章更进一步地描述了如何用 Visual C++ 编辑器来访问和查看项目，例如那些在附带光盘上找到的项目。

源文件注释

要求源文件注释会使 AppWizard（应用程序向导）在生成的源代码中添加有用的所做事项的注解。这些注解外观类似于下面的注释，它建议你添加的源代码，

应使一个特征或函数变得可操作：

```
void CDemoDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        //TODO:add storing code here
    }
    else
    {
        //TODO: add loading code here
    }
}
```

选择添加源文件注释，也会使 AppWizard（应用程序向导）在项目文件夹中放置一个 ReadMe.txt 文件。ReadMe 文件作为整个项目的目录列表，提供了对

AppWizard（应用程序向导）生成的所有文件的简要说明。

链接到 MFC 库

第五步中的第三个选项决定程序如何链接到 MFC。默认情况下，As A Shared DLL（作为共享的 DLL）单选按钮被选中，这意味着 AppWizard（应用程序向导）建立应用程序，来动态地链接到包含在单独文件中的 MFC 库。这种链接方式显著地减小了应用程序的可执行文件的大小，并能更有效地利用系统资源。

然而，动态链接到 MFC 要求提供 Mfcnn.dll 库文件，文件名中的 *nn* 代表 MFC 的版本号。文件通常是在 Windows System 或 System32 文件夹下。如果你的应用程序动态链接到 MFC，而且应用程序通常是用在那些可能没有 Mfcnn.dll 库的系统上，你应该把这个库文件作为应用程序包的一部分提供给用户。如果应用程序使用 Unicode，则应用 Mfcnnu.dll 文件来代替上述的库文件。微软允许你自由地把这些库文件附在应用程序中。应用程序的安装程序可以搜索用户硬盘上已有的 MFC 库文件，并把文件复制到 System 文件夹中（如果文件夹中还没有的话）。如果还没有 Msvcrt.dll 文件，则还要复制它，因为 MFC 使用 C 运行时库的共享版。库文件同旧的应用程序保持向上兼容，因此如果用户的 System 文件夹中已经包含有更新版本的 MFC 和 C 运行时库文件，你的安装程序就不需要再复制这些文件。

如果你的动态链接应用程序打算在国外发行，在那里，它可能会运行在不同语言的系统上，那么你需要考虑更多的事情。MFC 库文件包含了字符串数据，例如

程序能访问的对话框文本和帮助信息。你必须保证应用程序不会访问和显示以用户母语以外的语言写的库文件字符串。有两个办法来解决这个问题。最简单的办法是写程序时使用自己专用的字符串数据，而不访问由库文件提供的文本（第 4 章更详细地讨论了这个问题）。然后，你在分发 `Mfcnn.dll` 时，就可以不用考虑用户的区域设置了。

第二种解决办法是改写安装程序，使它询问主机系统的当地语言、把重新分配的 `Mfcnnxxx.dll` 文件复制到 `System` 文件夹中，并重命名为 `Mfcnnloc.dll`（文件名中的 `xxx` 代表主机语言的三位代码，例如 *deu* 代表德语而 *fra* 代表标准的法语）。要了解这个主题的更多信息，参见在线帮助的 Technical Notes 56 和 57，你可以通过 MSDN 索引中的“MFC components”条目来找到它。

如果你愿意把应用程序静态链接到 MFC，选择标题为“`As A Statically Linked Library`（作为静态链接库）”的单选按钮。静态链接意味着，应用程序不依赖于 MFC 库文件的存在，尽管它可能仍需要 `Msvcrt.dll` 文件。静态链接的代价是可执行文件更大，而且，对内存的利用可能不够充分。在 Visual C++ 的学习版中，静态链接到 MFC 是不可能的。

你在第五步中选择的 MFC 链接选项仅仅是项目的初始设置，你可以在开发过程中的任何时候选择一个不同的选项。在生成项目之前，从 `Project`（项目）菜单中选择 `Settings`（设置）命令，并在对话框的 `General`（常规）选项卡中选择静态或动态链接。

步骤 6：类和文件名

AppWizard（应用程序向导）的第六步（也是最后一步）逐条列举 AppWizard（应用程序向导）将为项目创建的类。要改变类的名字，在列表选中它，并在 Class Name（类名）文本框中输入一个新名字。其他的文本框显示了 AppWizard（应用程序向导）为类源代码创建的文件名。这些名字只是一个建议，你可以为列表中除应用程序类(图 2-14 中名字为 CDemoApp 的类)外的所有类输入一个新的名字。包含应用程序类的源文件的名字取自项目，因此不能被改变。

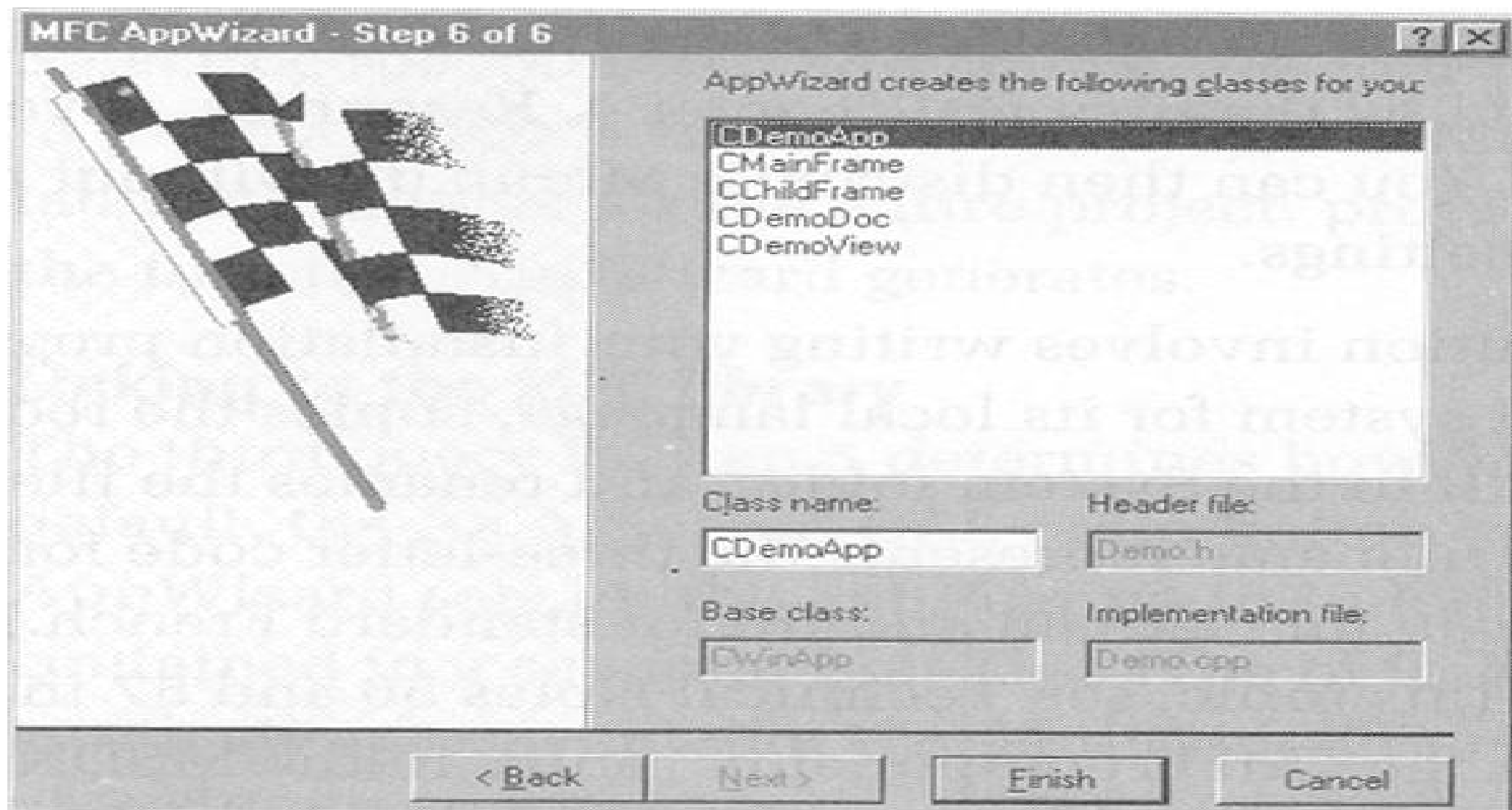


图 2-14 在 AppWizard 的第六步中指定类名

单击 Finish（完成）按钮后，AppWizard（应用程序向导）显示一个摘要表，其中列出了所选的项目特征(图 2-15)。这个摘要是你取消项目的最后机会。单击 OK（确定）会使 AppWizard（应用程序向导）在摘要表底部所列的目录下创建项目。

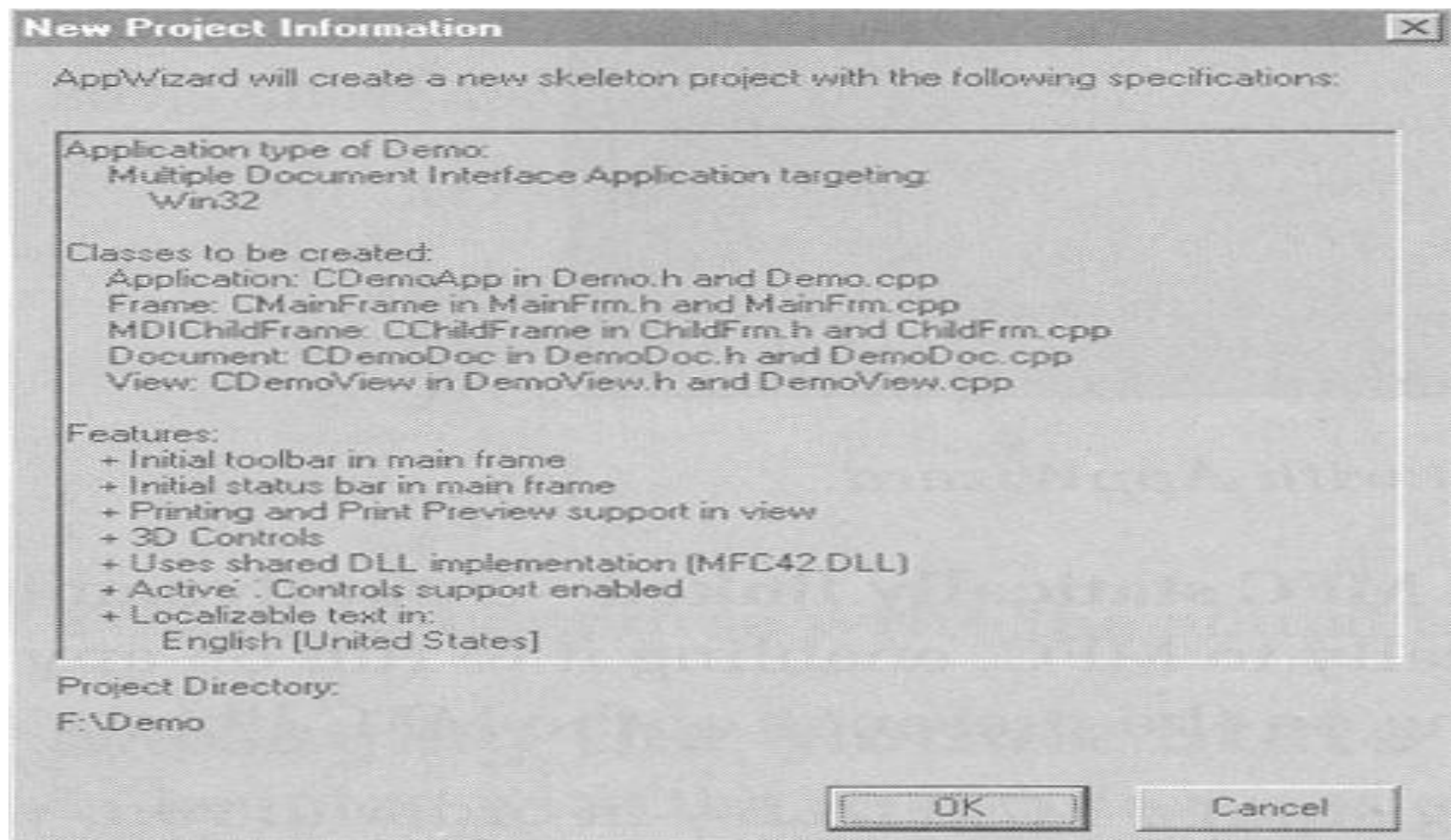


图 2-15 在 AppWizard (应用程序向导) 中选择的项目特征列表

用 AppWizard（应用程序向导）来创建 DLL 文件

如果你要开发一个动态链接库，而不是一个普通的 Windows 应用程序，在 New（新建）对话框中选择 MFC AppWizard(dll)图标(参见图 2-3)。这个特殊的 AppWizard（应用程序向导）仅显示图 2-16 所示的一个步骤，它询问诸如你的动态链接库应如何链接到 MFC 之类的信息。

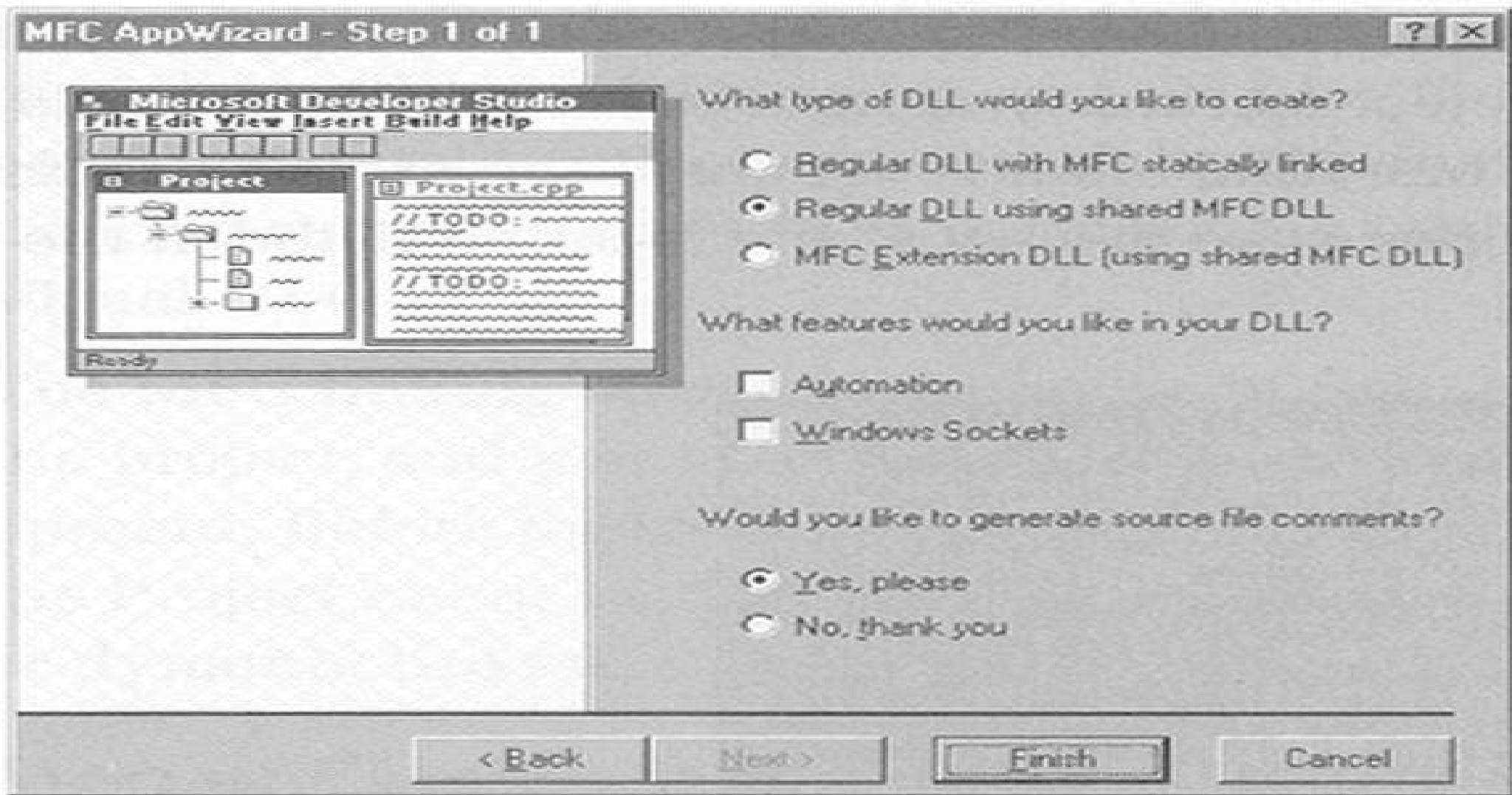


图 2-16 使用 AppWizard (应用程序向导) 设置 DLL 项目

这个向导提供了三个不同的链接选项，每个都有优点和缺点。前两个选项使动态链接库能被任何 Win32 程序访问。第三个选项限制较严，因为它创建的动态链接库只能用在使用 MFC 的应用程序和其他库文件上。下面描述了各个链接选项：

- Regular DLL with MFC statically linked (具有静态链接 MFC 的常规 DLL)：动态链接库静态地链接到 MFC，这使它能不依赖于 MFC 库文件的存在，而运行在任何的 Win32 系统上。
- Regular DLL using shared MFC DLL (使用共享 MFC DLL 的常规 DLL)：运行你的动态链接库需要访问正确版本的 MFC 库文件。这样就减小了最终的可执行文件大小，但正如前面解释过的那样，这可能需要在你的产品中附带 MFC 库文件。如果你的动态链接库是设计用在动态链接到同一版本的 MFC 的应用程序上时，你尤其需要考虑这个选项，因为这样一个 MFC 的实例就可以既用作调用的应用程序，又作为你的库文件。
- MFC Extension DLL(using shared MFC DLL) (MFC 扩展 DLL (使用共享 MFC DLL))：这个选项同前述的选项有些类似，最重要的差别是调用过程也必须动态链接到正确版本的 MFC 库文件。一个 MFC Extension DLL 提供的类加强或补充了原有的类的功能。要了解编写 MFC Extension DLL 的详细信息，参考 Technical Note 33，MSDN 在线

帮助中的“DLL Version of MFC”。

如果你想把动态链接库用在诸如 Microsoft Excel 和 Visual Basic 之类的 Automation 客户上，选中 Automation 复选项。选中 Windows Sockets (Windows 套接字) 复选项来添加对在 Internet 或其他任何使用 TCP/IP 协议的网络系统上进行通信的支持。选择 Windows Sockets 支持会使 AppWizard (应用程序向导) 增加一个对 MFC 的 AfxSocketInit 函数的调用：

```
BOOL CDemoApp::InitInstance()
```

```
{
```

```
    if (!AfxSocketInit())
```

```
    {
```

```
        AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
```

```
        return FALSE;
```

```
    }
```

```
    return TRUE;
```

```
}
```

当然，你必须自己写实际的通信代码。

管理模块状态

为保持动态链接库的简易特性，AppWizard（应用程序向导）为库文件项目生成的代码比起它为普通的应用程序所写的源代码来更简洁。项目的 CPP 文件包含一个消息映像和类的构造器。文件还包括一个注释块，它解释库文件中输出函数可能需要调用 MFC 的 AFX_MANAGE_STATE 宏。这取决于你的动态链接库是静态还是动态链接到 MFC 库。静态链接到 MFC 库不需要 AFX_MANAGE_STATE 宏，但是，因为项目的链接设置在开发过程中很容易改变，在开始的时候，最好先假设使用 MFC 的动态链接库是动态链接到 MFC 库文件的。因此，调用 MFC 的输出函数，应该像下面这样通过调用 AFX_MANAGE_STATE 宏来开始：

```
extern "c" _declspec( dllexport ) void WINAPI ExportedFunction()  
{
```

```
#ifndef _AFXDLL

    AFX_MANAGE_STATE( AfxGetStaticModuleState() );

#endif

}
```

在每个线程的局部存储器的内部，MFC 维持了一个指向被称为“模块状态”的结构的指针，该结构中包含了 MFC 库当前正在维护的过程模块的特定信息。当应用程序进入到动态链接库的输出函数中时，模块状态属于调用它的应用程序，而不是属于你的库文件。在把执行权转移到 MFC 时，输出函数应首先把指针改变为指向 DLL 自己的模块状态。这就是使用 AFX_MANAGE_STATE 宏的目的，它临时把模块状态指针指向当前的模块（即你的动态链接库），然后，当输出函数完成，并返回到调用它的应用程序时，这个宏恢复原来的指针。AFX_MANAGE_STATE 宏在由 MFC 库文件自己调用的输出函数中并不是必不可少的，例如，在消息映像中列出的 InitInstance 和句柄函数，这是因为 MFC 在调用函数之前已经正确设置好了模块状态。

AFX_MANAGE_STATE 宏应该放在函数的开头附近，甚至要在对象变量定义之前，因为它们的构造器本身可能就包括对 MFC 库文件的调用。程序段中的 #ifdef 条件块保证编译器只为动态链接到 MFC 的库文件包含宏代码。如果链接是静态

的，Visual C++ 编译器没有预定义 `_AFXDLL` 常量。要更详细地了解 `AFX_MANAGE_STATE` 宏，请参考 Technical Note 58, MSDN 在线帮助中的“MFC Module State Implementation”。

第二部分 编辑器

第 3 章 文本编辑器

Visual C++提供了一个专门为“代码剪切”而设计的文本编辑器。这个编辑器与各种环境工具（诸如调试器）组成一个完整的统一体，并提供了大量复杂的特征，包括 Undo/Redo（撤销/重做）、可定制的快捷键命令和对 Win32 和 MFC 参考的快速访问。

作为一个程序员，你一定已用过了一种或几种编辑器。本章将介绍 Visual C++文本编辑器最重要的几个方面，讲述一些有用的和隐含的特性，并展示怎样有效地使用编辑器。即使你仍愿使用现有的编辑器来完成大多数编码任务，也应浏览一下本章，以便了解 Visual C++的强大功能。不久你将发现，如果只想修改编译器错误，那么，在编辑文本时，将文本保持在 Developer Studio 环境中将是很方便的。在本章的最后，还有一些有用的建议。

由于 Visual C++是一个基于 Windows 的软件，它将其文本文件以 ANSI 格式存储。有关 ANSI 标准和 ASCII 及 ANSI 码表，请参考附录 A。

启动文本编辑器

刚进入 Visual C++ 时，看不到文本编辑器，也不会在任何菜单下看到“Start the text editor（启动文本编辑器）”或“editor（编辑器）”之类的选项。在 Visual C++ 通向对象的环境中，只需给出文件类型，而不必管应该使用哪一个工具。在这个环境中，除了文本编辑器，还有其他几个编辑器。只需指出想创建或修改的是一个文本文件，而不是其他类型（比如说，图表文件），Visual C++ 将会根据文件类型自动启动相应的编辑器。

进入编辑器后，如果想创建一个新文件，请单击 File（文件）菜单，再选择 New（新建）。在如图 3-1 所示的 New 对话框中，Visual C++ 按字母顺序列出可以创建的文件类型。

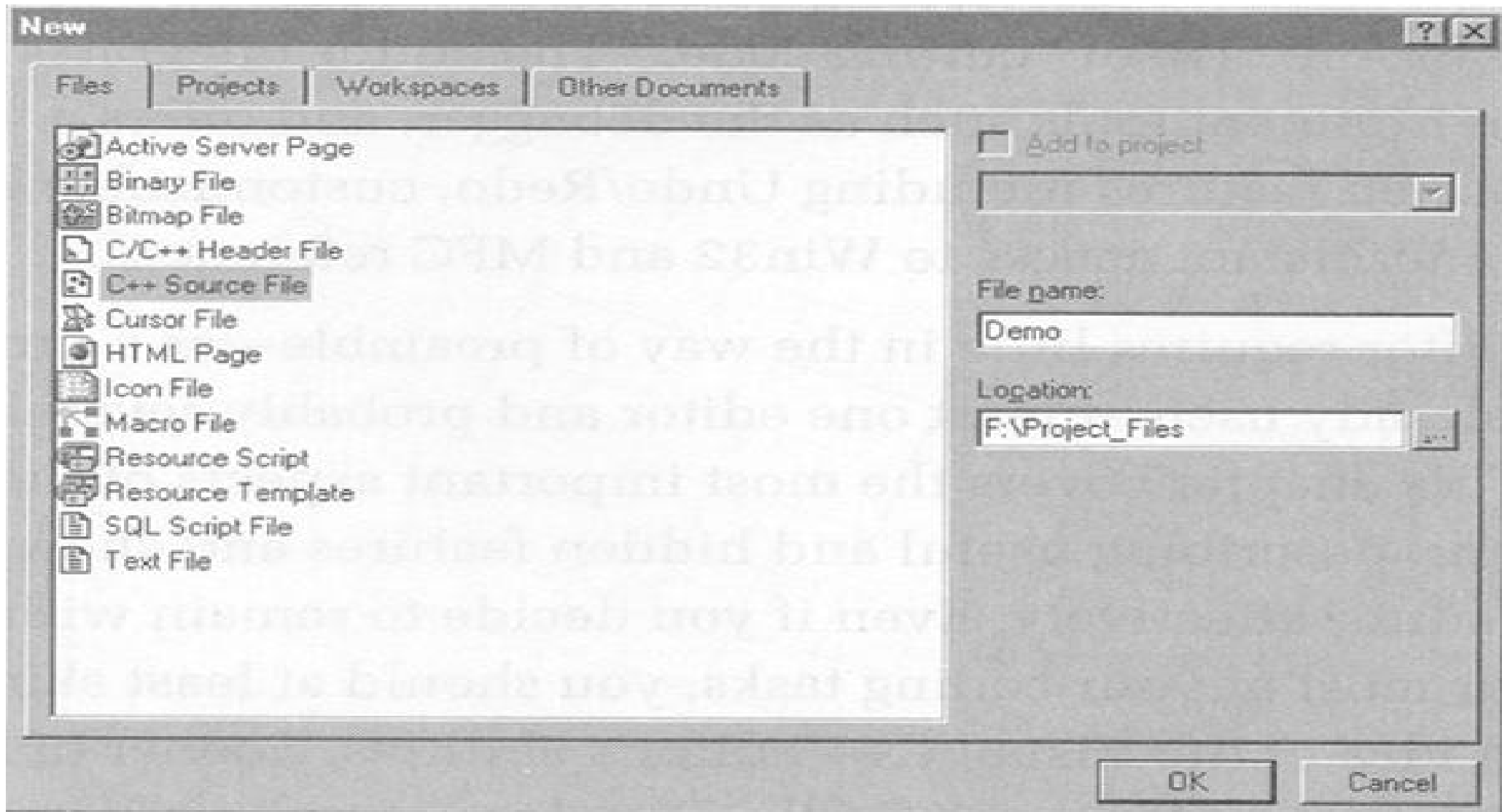


图 3-1 在 New 对话框中选择文档类型

输入文件名，然后在列表选定其类型，如 Active Server Page、C/C++ Header File、C++Source File、HTML Page、Macro File、SQL Script File 或 Text File。再单击 OK（确定）就会发现，文本编辑器以一个空文档窗口的形式出现在屏幕上。此时菜单和工具栏都基本不变，但会有一声短促的喇叭声。为了保持工作的连续性，Visual C++采用通用的外观和动作，以便使软件易学易用。例如，如果新文档满屏显示，那么，除了文档本身外，只有很少的几条可视觉线索能表明此时是在文本编辑器中，而不是在 Visual C++的主窗口中。其中之一是在菜单栏的左边缘有一个小页标；另一个是在主窗口顶部标题栏中有文档名称，它指出了文档的类型。如果在 New（新建）对话框中没有给出文档名称，编辑器将自动给此新文档赋一个临时名称，像 Text1 或 Cpp1，直到给文档赋了新名称为止。

在本章后面会看到，Visual C++的菜单在各种编辑器中是通用的。虽然表面的菜单和工具栏基本不变，但在内部会有一些变化。例如，在使用文本编辑器时，许多在主窗口中呈灰色的命令被激活，而变为正常体。对不同的编辑器，Visual C++会自动激活相应的命令。例如，由于文本只对文本编辑器有效，因此，Find（查找）命令在使用文本编辑器时被激活，而呈正常体，而在使用图形编辑器时，则因未被激活，而呈灰色。图 3-2 中简要列出了使用文本编辑器时可用的菜单命令。

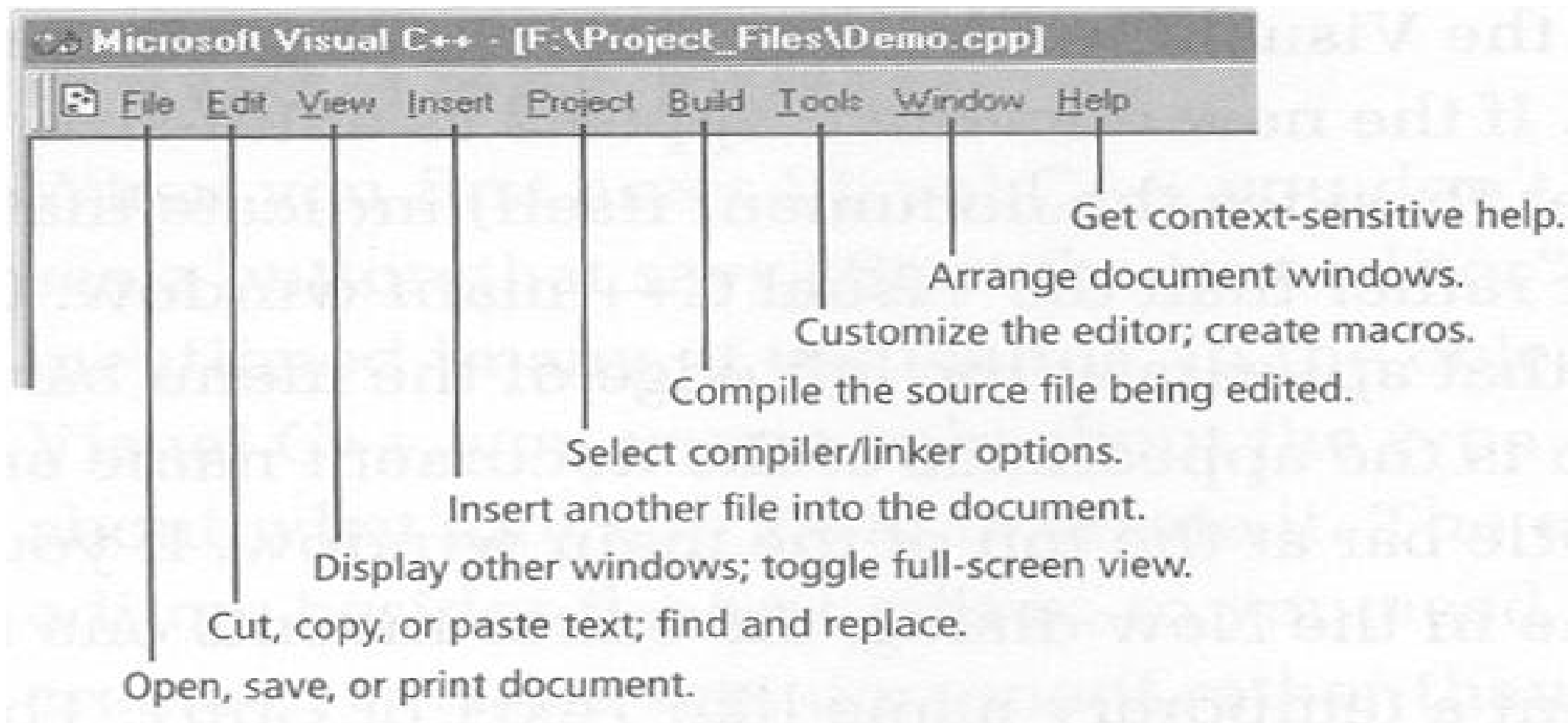


图 3-2 文本编辑器菜单

除了上图中所列的命令外，还有一些命令是可用的。例如，如果在编辑器中删除了文本，却又想恢复它，可使用 Edit（编辑）菜单下的 Undo（撤销）和 Redo（重做）命令，Undo（撤销）能记忆最近一次删除操作。如果要恢复更早删除

的文本，连续单击 Undo（撤销），或重复按 Ctrl+Z 直到恢复到需要的文本为止。不过，有时会多击一次，或多按一次，从而多恢复了无用的文本，这时候，可以使用 Edit（编辑）下的 Redo（重做）命令来取消最近一次 Undo（撤销）操作。

文档

这一节是本章最长的一节，讲述怎样创建、打开、浏览、存储和打印文本文档。如序言中提到的那样，如果已使用过基于 Windows 的文本编辑器或字处理器，那么本节所讲的一些知识只是对已有知识的复习。但是，即使对于有经验的 Windows 用户，在它们了解了 Visual C++ 的细节后，它们仍能在浏览和打印文档方面获益。

打开文档

Visual C++ 文本编辑器支持多文档操作，所以可以同时打开多个文档。再使用 New（新建）命令，就可以创建另一个新文档，系统自动默认文档名为 Text2 或 Cpp2。如果是满屏显示，则正在使用的文档名称显示在屏幕顶端标题栏中。可以通过按 Ctrl+F6 在已打开的文档间进行切换，也可通过从 Window（窗口）菜单中挑选所需的文件的名称来实现切换。

同一个文档只能创建一次。当将一个新文档存到盘上时，从那一刻起，它将成为一个文件，想再次对它进行处理时，就必须打开它，而不能再创建它。用下述办法可打开一个已存在的文件：

- 单击标准工具栏中的 Open（打开）按钮。



- 按 Ctrl+O。
- 选择 File 下拉菜单中的 Open（打开）命令。
- 选择 File 下拉菜单中的 Recent File（最新文件）命令,再选择所需文件

的名称。

前三种办法都会出现 Open（打开）对话框,在对话框中,可以通过浏览各个文件夹来寻找想要打开的文件。最后一种方法不需要对话框,系统自动列出最近编辑过的文件列表,称做 MRU 列表,从这个列表中,可以直接打开所需要的文件。

最近编辑的文件列表

在默认情况下,MRU 列表包含最近编辑过的四个文件,这四个文件并不只是文本编辑器编辑过的,也包含其他任意一种编辑器编辑过的文件。拉下 File（文件）下拉菜单,将鼠标光标放在 Recent File 命令行上,就可显示 MRU 列表。在表中挑选文件名,就可在相应的编辑器中打开文件。当同时对几个文件进行操作时,使用 MRU 列表是很方便的。然而,你会发现,即使对于一个很小的软件项目,也需同时对四个以上的文件进行操作。这时,MRU 列表就显得不够用了。幸运的是,Visual C++允许扩展 MRU 列表,以使它可容纳更多的文件名。单击 Tools（工具）菜单下的 Options（选项）命令,然后按左、右箭头键选择 Workspace（工作空间）项。在标有 Recent File List Contains（最新文件列表包含）的文本框中输入新值即可(如下图所示)。

Workspace 选项还提供了另一种影响 MRU 列表外观的选项。如果希望 MRU 列表直接出现在 File（文件）菜单中,而不是隐藏在下一级命令里,只需取消 Show Recently Used Items On Submenus（在子菜单上显示最近使用的项）选项即可。不过,由于文件名的增加,File（文件）菜单会显得很拥挤

Options



Debug

Compatibility

Build

Directories

Workspace

Data



Docking views:

- ☒ Output
- ☒ Watch
- ☒ Variables
- ☒ Registers
- ☒ Memory
- ☒ Call Stack
- ☐ Disassembly
- ☒ Workspace

- ☒ Display status bar
- ☐ Display clock on status bar
- ☒ Reload documents when opening workspace
- ☐ Reload last workspace at startup
- ☒ Automatically sort window menu
- ☐ Use screen reader compatible menus
- ☒ Show recently used items on submenus

Window menu contains:

10

windows

Recent file list contains:

6

files

Recent workspace list contains:

4

workspaces

OK

Cancel

Open（打开）对话框

如果所需的文件不在 MRU 列表中，那么，必须使用 Open（打开）命令，再在 Open（打开）对话框确定文件。这个对话框中的默认路径是当前软件所在的文件夹，因此不必花很大力气来寻找文件。

在选择要打开的文件时，按下 Ctrl 键再单击文件名，就可以同时打开一组文件，每单击一次就多打开一个文件。如果想取消某个已打开的文件，只需再次按下 Ctrl，同时单击该文件名即可。选定文件后，单击 Open（打开），就可将所有选中的文件分别打开成不同的文档。如果想打开的文件在其目录下是连续排列的，那么，有一种更快的方法来选中它们。可先单击选中第一个文件，再按住 Shift 单击最后一个文件，这样，在这两个文件间的所有文件都被选中了。如果想不要其中某个文件，只需按住 Ctrl 的同时单击该文件即可。

注意：在 Open（打开）对话框中列出的文件并不是该文件夹下的所有文件，这是由于 Files Of Type（文件类型）组合框中有过滤设置。“过滤”是指只选择一组相关的文件。例如，系统默认的 C++ Files 过滤器将使 Open（打开）对话框中只包含以 C、CPP、CXX、TLI、H、TLH、INL 和 RC 为扩展名的文件。如要列出其他类型的文件，如以 HPP 或 TXT 为扩展名的文件，则须在 Files Of Type（文件类型）框中选择相应的过滤器。

在 Open（打开）对话框中有一个 Open As Read-Only（作为只读文件打开）选项。

这是一个很重要的选项，一旦它被选中，将不能对打开的文件作任何改变，而只能对其浏览、打印，或将其复制到剪贴板上。通常，只需使用 File（文件）菜单下的 Save as（另存为）命令，将此只读文档以一个新名称保存，就可以解除只读限制。注意，此时原文件只读属性并未改变，只是新文件中没有只读限制。Visual C++ 6.0 还提供了一种设置，使得即使对于上面这种无害的操作，系统也不允许进行。Tools（工具）菜单下 Options（选项）对话框中 Compatibility（兼容性）选项卡下，有一个 Protect Read-Only Files From Editing（阻止编辑只读文件）选项，当它被选中后，只读文件将不能以新名称另存。

查看文档

尽管文本编辑器尽可能有效地使用屏幕空间，但当有好几个窗口同时显示时，空间常常显得很紧张。为了使可用空间尽可能大，选择 View（查看）菜单下的 Full Screen（全屏）命令（如图 3-3），或者按 Ctrl+V，就可使标题栏、菜单、工具栏都消失，以便最大限度地腾出空间。如果要返回正常显示，只需按 Esc，或单击浮动的 Full Screen（全屏）工具栏即可。在满屏状态下，由于菜单栏消失，不能再用鼠标来单击下拉菜单。如果要使用下拉菜单，可先按 Alt，再按所需菜单的首字母即可。例如，先按 Alt，再按 F，就可激活 File（文件）菜单，再按左、右箭头键，可激活相邻的菜单（在这里，按 Alt 起激活菜单栏的作用，因此两个键不必同时按下）。

如果 Full Screen (全屏) 工具栏令你感到眼花缭乱, 单击此工具栏上的关闭按钮, 就可移走它。不过这样的话, 将只能通过按 Esc 键来返回正常显示。如要恢复该工具栏, 按 Alt+T 显示 Tools (工具) 菜单, 再单击 Custmize (自定义) 命令, 在 Custmize (自定义) 对话框内的 Toolbar (工具栏) 选项卡下选中工具栏列表中的 Full Screen (全屏)。以同样的方法, 也可以在满屏状态下显示其他工具栏或菜单栏。

Window (窗口) 菜单提供了一个文档列表, 其中包含所有的 Visual C++ 编辑器打开的文件。例如, 图 3-3 中的文档表就包含了一个文本文件 Test.CPP 和一个图表编辑器下的图片文件 Dimg.bmp。可以通过拉下 Window (窗口) 菜单, 再单击想处理的文档名, 来实现各个文档间的切换。如果想同时将所有的文档都显示在屏幕上, 可选择 Cascade (层叠)、Tile Horizontally (水平平铺) 或 Tile Vertically (垂直平铺) 命令 (如图 3-3)。

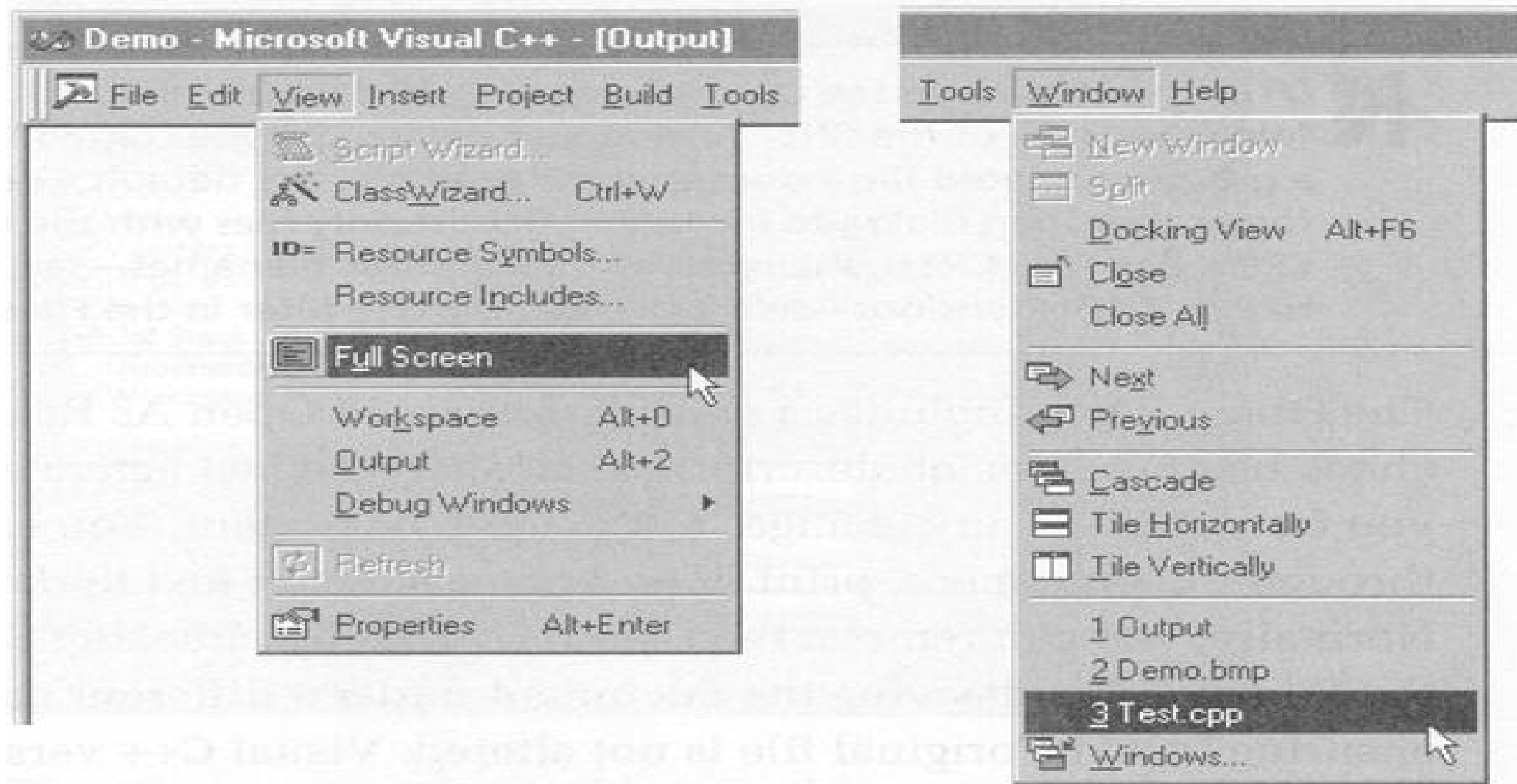


图 3-3 View (查看) 和 Window (窗口) 菜单

文本编辑器的文档窗口允许多窗格显示，可以将同一文档的一个、两个或四个不同部分同时显示在窗口上。图 3-4 就是同一文档的四窗格显示的例子。

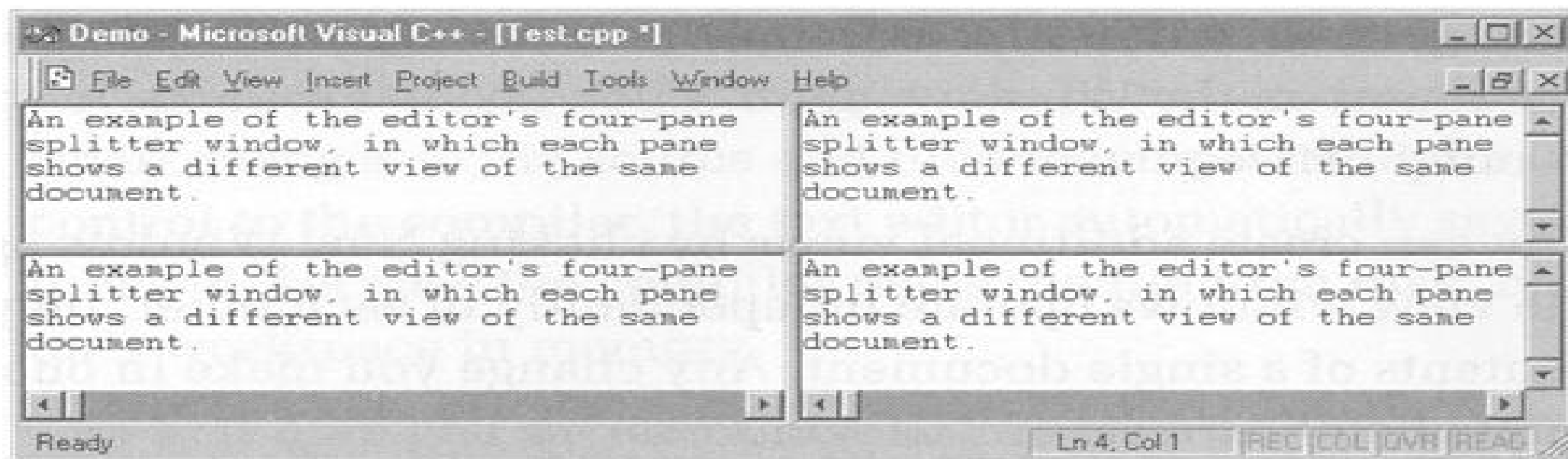


图 3-4 典型的文档窗口分为四个窗格

由于 Visual C++ 使用 MFC 的 `CSplitterWnd` 类创建每一个文本窗口，所以，当创建或打开一个文档时，多窗格功能自动生效。窗格分隔条最初显示为两个按钮，一个位于垂直滚动条的顶端，另一个位于水平滚动条的最左端。要想设定某个窗格分隔条的位置，只需拖动该窗格分隔条到所需位置再释放即可。也可以通过选择 Window（窗口）菜单下的 Split（分隔）命令，来一次设定两个窗格分隔条的位置。Split（分隔）命令设定了两个窗格分隔条所决定的分割线在编辑窗口中交

点的位置，移动鼠标，将此交点移到所需位置上，再单击，即可将两个窗格分隔条同时设定。

由于各窗格没有自己的滚动条，因此，最有效的方法是将显示窗口划分为一上一下两个窗格。要想做此划分，只需要将垂直分隔条（即位于水平滚动条上的分隔条）移到最左端或最右端，直到其消失为止。通过单击窗格或按 F6，可从一个窗格切换到另一窗格。

上面的两窗格显示对于上下划分的文档非常方便，但由于各窗格不能独立滚动，因此对于左右划分的文档就不大有效了。幸运的是，Window（窗口）菜单提供了另一个命令，它可以垂直视图的方式显示将一个文档分成两个或多个窗格。在文本编辑器只打开一个文档的情况下，单击 New Window（新窗口）命令，就会打开另一个包含同一文档的显示窗口。这并不是重新打开文件，而不过是在编辑器的工作空间中提供了原文档的第二个视图。这两个视图各有其滚动条及闪烁光标，所以可以同时浏览文档的不同部分。此时，再单击 Window（窗口）菜单下的 Tile Vertically（垂直平铺）命令，就可使两个浏览器实现左右显示。通过单击某个窗口，或按 F6 键，可实现视图之间的切换。

也可以通过 New Window（新窗口）命令再增加视图。不过，尽管各视图之间相互独立，但它们都只能显示同一文档的内容，因此，在一个视图对文档做的任何修改，都会立即显示在其他视图中。

保存文档

当开始在一个文档窗口中输入文本时，一个“*”符号会显示在标题栏和 Window（窗口）菜单下已打开文档表中的文档名后。这表明此时该文档已被改变，它和原文件已不相同。与文字处理器不同的是，Visual C++文本编辑器不能自动存盘，因此，在输入源代码时，一定要养成定时存盘的习惯。存盘可用以下方法：

- 单击标准工具栏上的 Save 按钮。



- 按 Ctrl+S。
- 选择 File（文件）菜单下的 Save（保存）命令。

保存文件时，文件名后的“*”符号将消失。当再次改变文本时，它又将出现。当“*”符号存在时，编辑器会提示先存盘。

定时存盘主要是针对长时间编辑一个文档的场合。在这种情况下，在处理文档时，偶尔问一下自己，“如果突然停电，我会不会因没存盘而很后悔？”。如果答案是肯定的，就请按 Ctrl+S。如果只是对源文件作一些小改动，再重新编译它，那么存盘就不那么重要了，因为在重新编译之前，系统会自动保存文档。

第一次保存一个未命名的文档时，会出现 Save as（另存为）对话框，在这个对话框里，你可给出文件名及其扩展名。请将扩展名定为 CPP 或 C，因为文本编辑器只能通过扩展名判断文件内容，再依内容将其按 C++ 或 C 程序编译。如果不给出扩展名，Visual C++ 将自动设一个与在 New（新建）对话框（图 3-1）中选择的文件类型相近的扩展名。例如，如果从此对话框中选择 C++ Source File（C++ 源文件），则新文件名的扩展名将为 CPP。

许多程序员常将 C++ 头文件的后缀名置为 HPP，在使用 Visual C++ 时，这会带来一点小麻烦。当选择 Open（打开）命令时，对话框中通常只显示以 C、CPP、CXX、TLI、H、TLH、INL、RC 等为扩展名的文件。要想打开一个扩展名为 HPP 的文件，必须将对话框下部的文件类型改为 C++ Include File（C++ 包括文件）或 All Files（所有文件）。除此之外，使用 HPP 作为头文件的扩展名再不会有别的麻烦。实际上，对于头文件可以使用任何扩展名，因为在创建项目时，Visual C++ 只根据“#include”语句来搜索源文件。无论扩展名是什么，只要被 #include 语句

包含，它就会自动加入到项目中，并显示在 Workspace（工作空间）窗口中的 FileView（文件视图）选项卡下的头文件表中。

当保存一个新文档，并给定其文件名时，这个文件名就代替系统默认的文件名，并显示在标题栏和 Window（窗口）菜单下。从此，当再保存此文件时，系统将自动覆盖前一个版本，并不再显示 Save as（另存为）对话框。而且，系统也不会以任何形式保存前一个版本。因此，一旦保存了一个文档，则它从前的版本将不复存在。如果源文件需要多个版本，则必须通过 File（文件）菜单下的 Save as（另存为）命令来给不同版本赋予不同名称。

打印文档

要打印一个正在编辑的文档，只需单击 File（文件）菜单下的 Print（打印）命令，或按 Ctrl+P，来打开 Print（打印）对话框。如果只想打印该文档中的一部分，先选择要打印的部分，再启用 Print（打印）对话框中的 Selection（选择）单选按钮（如图 3-5）。

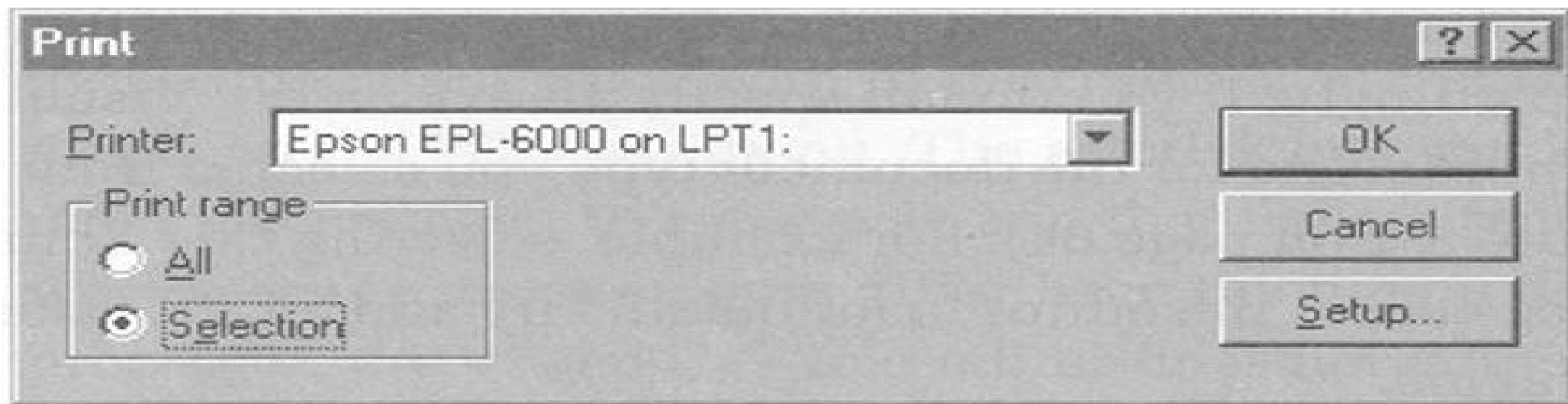


图 3-5 Print (打印) 对话框

Selection (选择) 单选按钮设定系统只打印用户选中的部分。如要取消它,只需单击图中的 All(所有)单选按钮即可。

上图所示的对话框也给出了 Windows 将使用的打印机。如要选择其他打印机,单击 Printer (打印机) 下拉框, 并从其列表中选择打印机类型。再单击 OK (确定) 就可开始打印。使用后台打印时, 系统会立即返回文本编辑器, 使你可以继续工作。尽管控制打印过程时需要浏览 Print 文件夹, 但还是可以迅速打印数个文档。单击任务栏中的 Start(启动)按钮, 再选择 Setting(设置), 然后单击 Printers (打印机), 选定打印机后再单击 File (文件) 菜单下的 Open (打开) 命令, 就可看到当前正在打印的文件列表。

当使用后台打印时，取消某个打印任务会稍微麻烦一些。一旦后台打印开始进行，Cancel（取消）按钮将从屏幕上消失，因此，只能通过 Printer 文件夹来取消打印任务。另外，如果后台打印不能用，那么，文本编辑器只能等待打印完成后才能被用户使用。

Visual C++ 文本编辑器为其打印的文本提供了有限格式，可以设置页边距，并指定每页的页眉和页脚。从 File（文件）菜单下选择 Page Setup（页面设置）命令，在 Page Setup（页面设置）对话框下将所要的页眉和页脚文本分别输入到对应的文本框中。可使用表 3-1 中的代码在页眉和页脚文本中加入某些实时信息。

表 3-1 在页眉和页脚中添加信息的打印代码

代 码	意 义
& F	文 件 名 称
& P	当 前 页 数
& T	系 统 时 间
& D	系 统 日 期
& L	页 眉 和 页 脚 与 文 本 左 边 缘 对 齐
& C	页 眉 和 页 脚 与 文 本 中 心 对 齐
& R	页 眉 和 页 脚 与 文 本 右 边 缘 对 齐

这些代码并不需要去记，只需单击 Page Setup（页面设置）对话框中文本框旁边的箭头，即可显示格式选项列表。单击其中某选项，就可将它插入到页眉或页脚中。打印代码就在此列表的最上端或最下端。

可将打印代码与正常文本以任意方式组合。例如，一个指明文件名称和打印日期的页眉可做如下安排：

```
&RFile:   &F;   Date:   &Dpage &p
```

这里，&R 代码使得页眉与右边缘对齐，这和字处理器中的右对齐是一样的。&P 代码打印页数，从“1”开始。这里的页数与打印作业相关，而不是与原文档相关。也就是说，如果选择文档的中间部分打印，&p 代码仍将被打印的第一页的页数定为“1”。Visual C++ 中没有代码可显示总页数，也就是说，不可能以类似于“Page 1 of 20”的形式来显示页数。另外，由于页眉和页脚中的打印代码在整个打印过程中都起作用，因此，不存在那种可交替使用左对齐与右对齐，以便设定奇数页为“&R”，而偶数页为“&L”的选项。

页眉和页脚中每行不能超过 40 个字符。每个打印代码算两个字符。Tab 键不允许使用。

浏览文档

当然，通过按上下左、右箭头键或拉动滚动条，可以浏览文档。可是，在下面将可以看到，通过一些其他办法可更方便、更有效地浏览文档。在此之前，先让我们统一一些术语。在 DOS 模式文本编辑器中很熟悉的光标，在 Windows 中有不

同的名称。Windows 把闪烁光标称为提示符，因为它的功能与用来表示可输入新文本的 ^ 符号相似。光标一词在 Windows 中仍被保留，它被用来指代表示鼠标位置的箭头（或其他标志）。Visual C++ 的在线帮助称插入符为“插入点”，作为一个 Windows 程序员，应该知道光标与提示符之间的区别。这样，当进入 API 函数（例如 ShowCaret 和 SetCaretPos）时，就不会感到迷惑了。

在文本编辑器中移动提示符的快捷键，对于任何一个使用过 Windows 字处理器的用户来说，都是很熟悉的。表 3-2 列出了主要的提示符移动快捷键。

表 3-2 文本编辑器插入符移动键

敲 键	插入符移动
左箭头，右箭头	向前或向后移动一个字符。如果提示符在最左端，则左移将移到前一行的末尾；如果提示符在右端，则右移将移到下一行的开头
上箭头，下箭头	向上或向下一行。如果目标行比当前行短，则移动后提示符的位置取决于提示符现在的具体位置
Ctrl+左箭头，Ctrl+右箭头	向前或向后移动一个单词。Visual C++编辑器将很多标点符号按独立单词对待。例如，必须按 7 次 Ctrl+右箭头，才能移过“CAN'T/WON'T”
Home，End	移到一行的开头或末尾
Ctrl+Home， Ctrl+End	移到文档的开头或末尾

续表

Page Up, Page Down	按照窗口中显示的行数来上下滚动文档。这里，编辑器将重叠滚动一行，也就是说，当按下 PageDown 键时，当前屏幕最底部的一行将变成下一屏的最顶行。这个滚动间距是不能改变的

在空白区内移动

当把插入符移到一行的末尾时，文本编辑器下面再怎样动作没有一个统一的规则。有些编辑器将不进行任何操作。另有一些编辑器将文本看作连续的流，当在行尾按右箭头键时，插入符会移到下一行的开头。如果长时间右箭头键不放，甚至可将插入符一直移到文档末尾。还有一些编辑器允许插入符继续向右进入空白区。这种方式的好处是，它使屏幕就像一页纸，你可以任意移动插入符，并可在任何地方输入。但同时它也有一些缺点。

想一想，当插入符位于下例中第一行的末尾时，想将它移到下一行，会出现什么情况：

```
b = SendMessgae( hwnd, MY_MESSAGE, wParam, lParam);
```

```
b *= 2;
```

如果没有空白区，按右箭头键就可很快移到第二行开头，按向下箭头键，可移到第二行的结尾。而对于有空白区的编辑器，则需要按两次键。必须先按向下箭头键，再按 End 或 Home 键。然而，在另一方面，有空白区的编辑器又可以很方便地给第二行加一个注释。只需把插入符向下移动到空白区即可：

```
b =SendMessage (hwnd, MY_MESSAGE, WParam, lParam);
```

```
b*=2;           // Double it
```

顺便说一下，Visual C++之外的编辑器大都将文档看作文本流。这样，在文档中就不能留有间隙。在原文档与在空白区加入的文档之间的间隙中，Visual C++编辑器用尽可能多的 TAB 键和空格键填充。

上面两种体系都有其麻烦之处。为了使操作方便，Visual C++文本编辑器将选择权交给你，可按以下步骤将空白区设置改为自己所喜好的类型：

1. 选 Tools（工具）菜单下 Options（选项）命令。
2. 单击 Compatibility（兼容性）选项卡。
3. 设置或清除 Enable Virtual Space（启用空白区）文本框中的值。

括号匹配

Visual C++文本编辑器可识别包围 C/C++源代码块的括号对，这样，只按一次键，就可将插入符由一个括号处移到与此括号相对应的括号处。编辑器可区别三种不同类型的括号：小括号（），大括号{}，方括号[]。

括号通常成对使用，用于源代码块的两端。每一对括号建立一个层次，并可能包含若干个次级层。下例就是一个用大括号来分层次的典型例子：

```
if (msg=WM_USER)

{
    //Begin level A

    for(i=0;i<5;i++)

    {
        //Begin level B

        .

        .

        .

    }
    //End level B
```

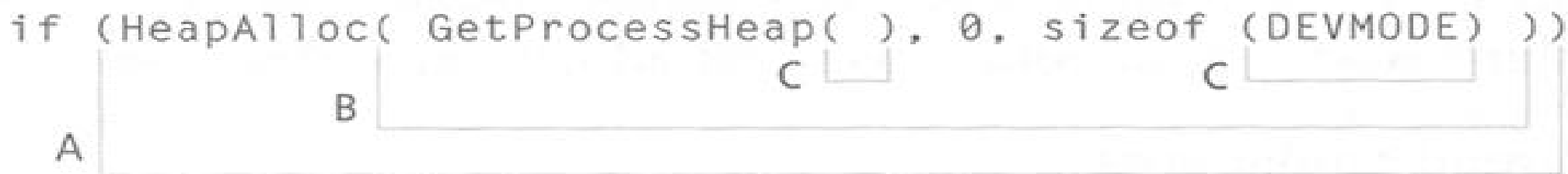


```
} //End level A
```

当光标在任何一个括号旁边时，按 **Ctrl+]** 可将插入符移到与之对应的另一个括号处。按 **Shift+Ctrl+]** 可选定两括号间所有文本。

C 和 C++ 中的小括号用来容纳不同的代码元素。然而，被小括号定义的层次不依赖于语句类型，而依赖于这些层次在文档中的角色。下面这个例子就举出了 A、B、C 三个层次的例子：

```
if (HeapAlloc( GetProcessHeap( ), 0, sizeof (DEVMODE) ))
```



这里，最里面的两对括号层次相同，都是 C 层，并且都包含在 A 层与 B 层内。当插入符位于第一个括号旁时，由于它代表 A 层的开始，因此，按 **Ctrl+]**，可将插入符移到代表 A 层结尾的最后一个括号处。

编辑器通过使用一个小程序计算源代码中括号的个数，从而确定每个括号属于那一层。任何一个完整层次中左右括号个数必须相等，否则就说明有错误。在把插入符从第一个括号处移到 A 层末尾的过程中，编辑器向前搜索匹配括号，并启动计数器。每发现一个左括号计数器加一，每发现一个右括号计数器减一，直到

编辑器发现层尾的括号时计数器又变为 0，于是，编辑器移动插入符到此括号处。

编辑器也把 `#if`，`#ifdef`，`#else`，`#elif` 和 `#endif` 这些条件编辑指令按括号对待，不过，相应的操作键是不同的。当插入符位于某个条件指令旁边时，按 `Ctrl+J` 可将它移到下一个指令旁。如果同时按下 `Shift` 键，则会在移动的同时选中这一区间的文本。

书签

文本编辑器的书签可以记忆一个位置，这样，无论在文档的任何位置，都可以迅速返回书签所在位置。如果以前是利用 `Go To` 命令来跳到想浏览的位置，那么，现在一定会发现书签的好处。`Go To` 命令是以行数为目标的，若在文档的某些地方增加或删减了若干行，则其后的每一行都将向后或向前偏离其原始位置。因此，当有行加入或删除时，`GoTo` 命令不能准确到达想去的位置。而书签就方便多了，你不必去记行数，书签会自动保留在所在行，并随文档的增减而改变位置。Visual C++ 文本编辑器提供了两种类型的书签：有名书签和无名书签。

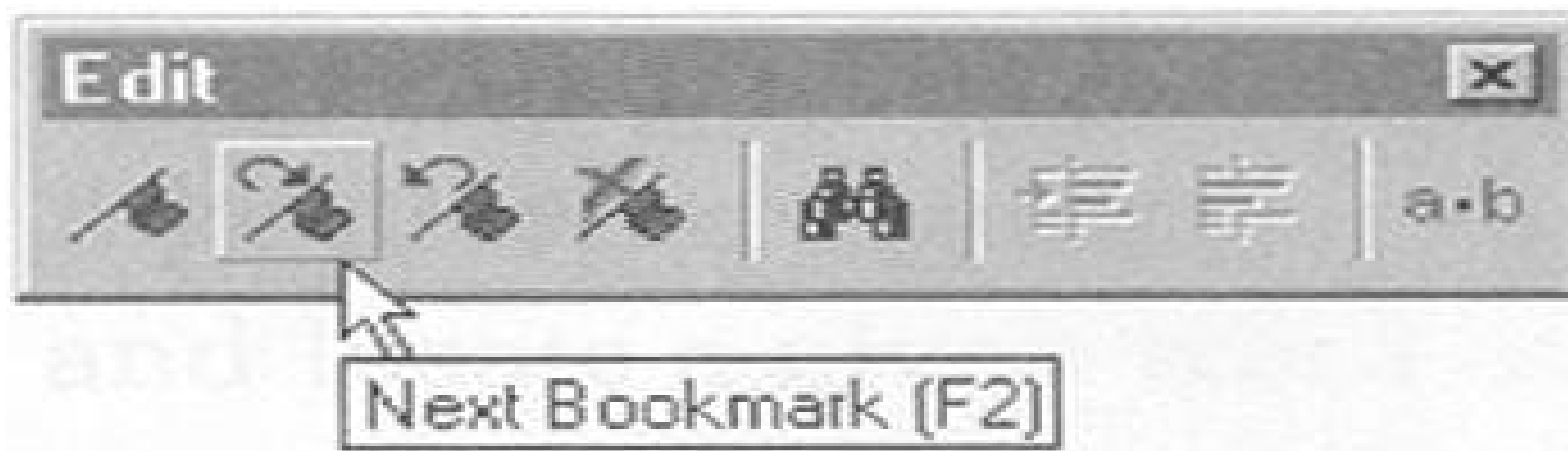
有名书签

一个命名的书签将会长久留在文档中，直到删掉它为止。它在文本中标志了一个精确的位置。实际上，当一个有书签的文档没有打开时，也可以从另一个已打开的文档中直接跳到该文档中。这里，编辑器自动打开该文档，并将插入符置于书

签所在的位置。

要创建一个有名书签，先将插入符置于所想标志的位置，再单击 Edit（编辑）菜单下的 Bookmarks（书签）命令，此时就会看到 Bookmark（书签）对话框。输入给此书签所起的名字，再单击 Add（添加）按钮，就可将此书签加入到列表中。当关闭对话框后，此新书签就开始起作用了。有两种方式可返回书签处：直接方式和间接方式。如果文档中的书签不很多时，用间接方式是最方便的。只需按 F2 就可跳到下一个书签处，按 Shift+F2 可跳到前一个书签处。单击 Edit（编辑）工

具栏中的 Next Bookmark（下一个书签）按钮，也会得到同样的结果，如下图所示：



通过直接方式移动插入符到书签处，需要再次访问 Boodmark（书签）对话框。双击书签名，或选择目标书签，再单击 GoTo（转到）按钮即可。也可以通过 Edit（编辑）菜单下的 GoTo（转到）命令来跳到某个有名书签处。

具体来说，有名书签是一个 32 位量，用以记录文本中一个特定位置。当在书签前的文本中添加或删除一个字节时，这个量会相应增加或减小。这样，书签就会继续指向其目标，文本怎样变更都不会影响它。与字处理器不同的是，关闭文档后，Visual C++ 编辑器不将有名书签保留在文档中，因为额外字符会使编辑器产生混乱。

无名书签

通常，上面使用书签的方式似乎有些大材小用。当想定义这样一个书签时，它标

志源代码的一处地方，在编辑文档的其他部分时，只想返回这个地方一两次，然后它就没用了，在这种情况下，有名书签就不是很方便了。此时，为方便起见，可使用一个无名书签。无名书签是暂时的，当取消它或关闭文档后，它就消失了。而且，无名书签只标志一行，而不是一个精确位置。当插入符跳到一个无名书签处时，插入符自动置于该行开头。如果删除被标志的行，则书签也将被删除。

无名书签的好处是，它易于设置，删除更简便。要使用一个无名书签，只需将插入符置于该行，按 `Ctrl+F2`，或单击标有旗帜的按钮（如图示）即可。



如果页边选择（本章后面部分将会描述）被激活，那么，一个小盒子图标就会显示在被标志行左边的页边处。否则，编辑器将以不同颜色来显示整个被标志行。

可通过单击工具栏按钮，或按 `F2/Shift+F2`，来移动插入符到无名书签处。每按一

次，插入符将向前或向后移到下一个书签处，无论该书签是有名书签，还是无名书签。

有以下几种方法可删除无名书签：

- 将插入符移到此行，再按 `Ctrl+F2` 即可删除此书签。
- 按 `Shift+Ctrl+F2`。这将删除所有无名书签。
- 不去理它。当文档关闭后，无名书签将自动消失。

文本搜索

Visual C++ 编辑器提供了三种文本搜索类型。 可以：

- 在已打开的文档中搜索文本。
- 在已打开的文档中替换文本。
- 在磁盘文件中搜索文本。

前两种类型在文本编辑器中是很普通的，而在磁盘文件中搜索文本，则是一个不常见但却极有用的类型，它可显示出包含一个特定词或短语的所有文件。以下我

们将详细说明这三个类型。

在打开的文档中搜索文本

像大多数文本编辑器一样，Visual C++编辑器能够扫描文档，并找出给定单词或短语（我们称之为目标字符串）的位置。有两种方法可设定目标字符串的内容。最方便的方法是使用标准工具栏中的下拉框，如下图所示：



在下拉框中输入字符串，或者单击框旁的箭头按钮，再从列表中选择以前输入的字符串。按 Enter 键开始搜索。当编辑器找到该字符串后，它将其高亮显示，并将插入符置于此高亮字符串的首字符旁。只要组合框允许，按 Enter 键就可继续按下一个已给出的目标字符串搜索。按 Esc 键，或单击文档窗口的任何部分，就可返回编辑模式。再按 F3 或 Shift+F3，就可继续向后或向前继续搜索。Visual C++ 提供了这些命令的工具按钮，不过需要你自己将它们加到工具栏中。本章后面的“为命令创建工具栏按钮”一节将说明怎样添加工具按钮。下面是它们被加到工具栏中后的样子：



第二种设定目标字符串的方法需要 Find（查找）对话框。虽然这种方法不如第一种直接，但它提供了更多的选择。例如，如果想搜索的字符串碰巧在屏幕中，可以直接借用它，而不必再输入它。如果目标字符串只是一个单词，只需通过单击这个单词或将插入符置于这个单词上；否则，可通过拖动鼠标光标来选中所需文本。接下来，通过按 Ctrl+F，或选择 Edit（编辑）菜单下的 Find（查找）命令，来打开 Find（查找）对话框。当对话框显示出来的时候，所选中的文本已经在对话框中了。

可通过设定是否区别大小写，或是否整词匹配，来使搜索更为精确。单击 Match Case（区别大小写）复选框，将设定一个区别大小写的搜索。在这个搜索过程中，编辑器只找出与目标串完全相同的字符串。例如，对于目标串“abc”，区分大小写的搜索只找出“abc”，而不区分大小写的搜索则会找出“abc”、“ABC”、“Abc”

等等。单击 Match Whole Word Only（整词匹配）复选框，可忽略包含于另一单词之中的目标串。例如，对于目标串“any”，在整词匹配搜索中，编辑器只找出以完整单词形式出现的“any”，而忽略掉类似“company”、“many”、“anywhere”之类的单词。

在 Find（查找）对话框中，单击 Mark All（标记所有的）按钮，就可在每一个搜索到的目标处设置一个无名书签。当继续使用 Find 命令搜索其他目标时，这种设置能够回到前面的目标处。

Visual C++ 编辑器中一个有用的变化是，有一个叫 Incremental Search（增量搜索）的命令，它使得搜索工作在输入目标串的同时就开始进行。在一个打开的文档中按 Ctrl+I，则“Incremental Search:”就会立即出现在屏幕左下角的状态栏中。在输入目标串的同时，编辑器就开始搜索，通常不等输完，搜索就已完成。当编辑器找到正在寻找的字符串时，按 Enter 键，或一个方向键就可回到编辑模式。如要再次搜索同样的目标串，按 F3 或单击适当的工具按钮即可。按 Shift+Ctrl+I 键可改变搜索方向，以使编辑器从插入符所在位置向前搜索。

在已打开的文档中替换文本

如果想替换文档中的某些字符串，可选择 Edit（编辑）命令下的 Replace（替换）命令。这个命令将给出一个类似于 Find（查找）对话框的对话框，只不过这个对话框有两个文本框，需要两个字符串。第一个文本框中放置普通的目标字符串。

第二个文本框中则要放置要用来替换的新字符串。如果将第二个文本框置空，则编辑器将把找到的所有目标串删除。为了有选择地搜索或替换，当编辑器找到目标串时，单击 Replace（替换）按钮，系统将自动跳到下一个目标串处。单击 Find

Next（查找下一个）按钮，系统将跳过这个字符串，而不改变它。单击 Replace All（全部替换）按钮，将会一次性完成所有替换。可以只向前，或只向后搜索或替换，但不能在多个文件中搜索或替换。

如果在激活 Replace（替换）对话框之前，已选中了超过一行的文本，则 Selection（选择）单选钮将自动打开，这表明编辑器将把搜索或替换操作限制在所选章节之中。单击 Whole File（整个文件）单选钮，将取消这一设置。虽然可通过在按下 Alt 键的同时向右下方拖动鼠标光标来选中一系列字符串，但通常情况下，不能进行列的替换。如果选中一系列字符串，Selection（选择）单选钮将不会打开。不过，有一个宏操作可克服这一困难，在第 13 章中举出了一个可对列进行搜索或替换的宏操作的例子。

在磁盘文件中搜索文本

UNIX 系统的用户都知道这一功能。给出一个目标字符串，Visual C++ 文本编辑器能找出某个文件夹下所有包含目标串的文件，包括位于子文件夹内的文件。单

击 Edit（编辑）菜单下的 Find In Files（在文件中查找）命令，可打开如图 3-6 所示的对话框。这个对话框需要指定目标串、文件类型以及搜索过程所在的文件夹。

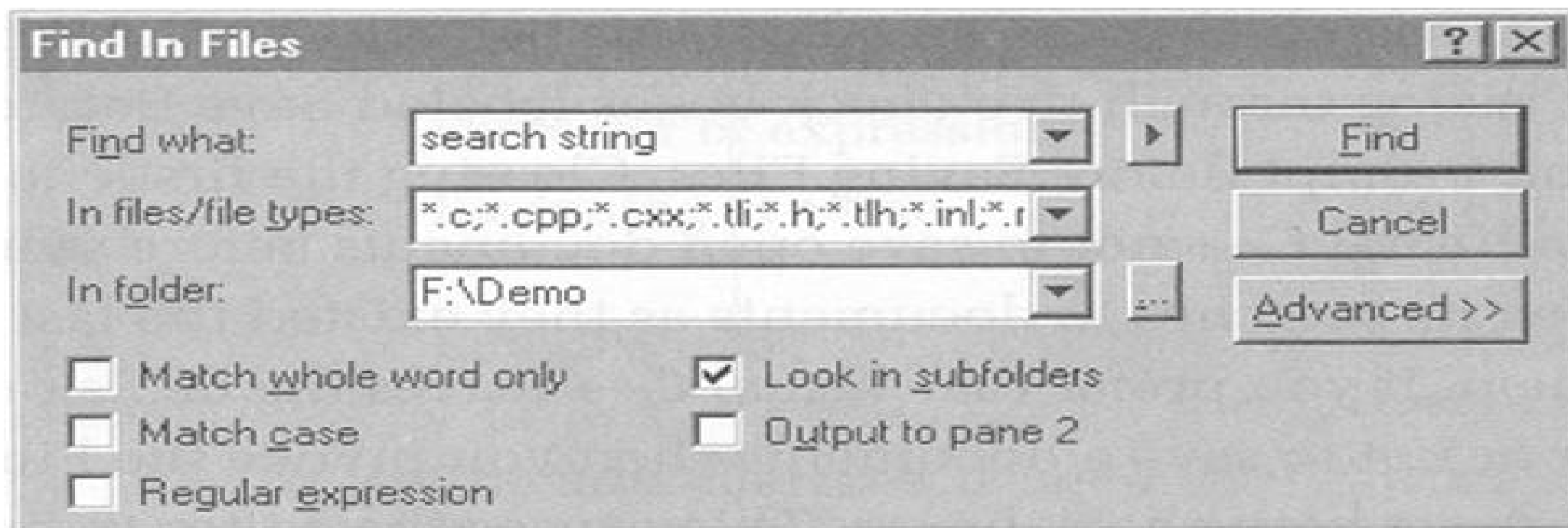


图 3-6 Find In Files 对话框，用于搜索磁盘文件

通常，默认文件夹是当前软件所在的文件夹；如果想搜索其他文件夹，在 In Folder（文件夹中）框中，输入其路径，或单击框旁的有三个点的按钮，来寻找所需文件夹。Look In Subfolders（在子文件夹中查找）复选框决定编辑器是否在子文件夹中进行搜索。通常其默认值为打开。单击 Advanced（高级）按钮，可设定除子文件夹之外的其他文件夹，搜索也将在这些文件夹中进行。

在第 1 章中我们已看到，Find In Files（在文件中查找）命令通常将其找到的文件

在 Output（输出）窗口中的 Find In Files 1 选项卡下列出（参看图 1-7）。如要将其在 Find In Files 2 选项卡下列出，只需激活 Output To Pane 2（输出到窗格 2）复选框，如图 3-6 所示。这个选项可同时显示两个独立的文件列表，这样，当前搜索的结果不会覆盖前一次搜索的结果。

如果已经设置好各项，单击 Find（查找）按钮。当 Visual C++ 找到一个包含所给出的目标串的文件时，它将其文件名和路径列在 Output（输出）窗口中，同时还列出此文件中第一个目标串所在行，这便可以看到目标串在文件中是怎么用的。双击某个文件，可在文本编辑器中打开它。

在进行文件搜索之前，Visual C++ 首先保存任何在文本编辑器中打开而未保存的文档，这将确保所搜索的文件都是最新版本的。可通过调整 Options（选项）对话框中 Editor（编辑器）选项卡下的两个标有“Save Before Running Tools（在运行工具之前保存）”和“Prompt Before Saving Files（在保存文件之前提示）”的复选框来改变这一属性。关闭“Save Before Running Tools”复选框，将使 Visual C++ 在搜索前不保存文档，这样就使搜索在上次保存的文档中进行。如果希望在激活 Find In Files（在文件中查找）命令之前，由自己决定是否保存文档，则关闭这两个选项。这将使编辑器在保存每个文档前先征求你的意见。

常规表达式搜索

迄今为止，我们见到的搜索对话框中，都包含一个标有 Regular Expression（常规

表达式) 的复选框。常规表达式是指有一个或多个特殊字符组成, 用以表示一系列文本的字符或字符串。在 Open (打开) 和 Save as (另存为) 对话框中, 我们已有过类似的使用, 例如, “*.CPP” 表示以 CPP 为扩展名的任何文件。这里的星号就是一个常规表达式, 它代表可组成有效文件的任何文本。

搜索中的目标字符串中所用的常规表达式则更为复杂, 这样可以更为精确地控制目标字符串。表 3-3 列出了系统默认的常规表达式。注意, 只有当搜索对话框中的 Regular Expression (常规表达式) 复选框打开时, 编辑器才把这些字符按常规表达式处理, 否则, 将把它们当作一般字符处理。

表 3-3 常规表达式字符

符号	含义	示例
.	任何单个字符	"..do"可以同 redo 和 undo 匹配, 但不能同 outdo 匹配
[]	任意字符或括号内的字符	"sl[aou]g"可以同 slag, slog 和 slug 匹配
[^]	插入符后所示范围之外的任何字符	"sl[^r-z]g"包含 slag, slog, 但不含 slug
*	没有或有更多的前一个字符	"re*d"包含 rd, red, reed
+	一个或多个前一个字符	"re+d"包含 red, reed, 但不含 rd
^	一行的开头	"^word"只包含位于行首的 word

续表

\$	一行的结尾	"word\$" 只包含位于行尾的 word
\	下一个字符不是常规表达式	"word\"\$"只与 word\$匹配，而不把\$看作常规表达式

不必去记这张表。在所有的 Find（查找）对话框中，都通过组合框旁的一个小按钮来给出表 3-3 的当前版本。单击此按钮，打开常规表达式菜单，然后就可以选择所需的常规表达式了。

用加号可设计一个字符串。例如，考虑常规表达式 `[a-zA-Z]`，它代表任何一个字母。如果再增加一个加号，意义就完全变了。加号的意义是“一个或多个这些字符”。因此，编辑器将 `[a-zA-Z]+` 看作为任意字符串，也就是任意一个单词。同样，常规表达式 `[0-9]` 代表一个数字，而 `[0-9]+` 则代表任意一个正数，无论其大小。

常规表达式搜索通常是区别大小写的。即使关闭 Find（查找）对话框中的 Match Case 复选框，对常规表达式 `[0-9a-f]+` 的搜索也只能找到类似于 `0x37ac` 的字符串，而不能找到 `0x7A4B`。为了找到后者，必须将大写字母包含在常规表达式中，例如：`[0-9a-fA-F]+`。

编程辅助工具

在 Windows 和 MFC 下编程，即使是最有经验的程序员，也不得不经常翻阅大量的参考资料，或浏览在线资料。我们还不得不花大量时间，去记住一些编程所必须的信息，例如参数表、函数或变量的拼写等。现在，Visual C++ 文本编辑器的一些特性将使程序员省去了这些麻烦。本章将介绍这其中最有用的一个工具：一种称作 Statement Completion（语句补全）的输入辅助工具。

Statement Completion（语句补全）是三个编程工具的总称，它们是：成员表（List Members），参数信息（Parameter Info），输入信息（Type Info）。这三个工具就如同是手边随时可用的 Win32 和 MFC 的参考资料。

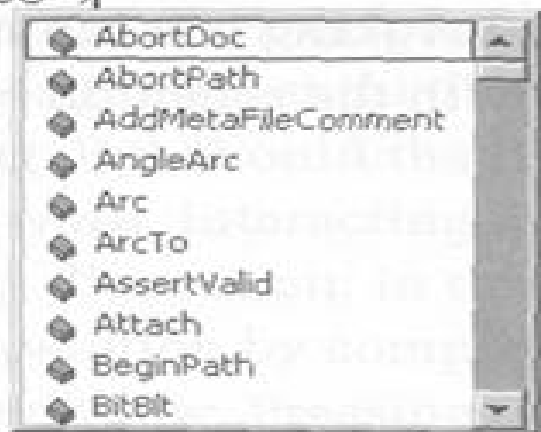
成员表

为了提高代码输入速度，并最大限度的减少输入错误，Visual C++ 编辑器的成员表可任随时调用。通过一个弹出式窗口，成员表给出了一个巨大的列表，其中包括 MFC 类成员、C 运行函数、显示常量、结构名称、Win32 API 函数、当前程序的类成员。可在这个列表找出要输入的单词，从而完成输入。当输入域运算符（::）、成员运算符（.）、指针运算符（->）时，成员表会自动弹出到屏幕上。此时，当继续输入一个成员名时，这个弹出式窗口的选择条会自动移到与输入最匹配的成员名上。图 3-7 说明了在函数 CDC::SetMapMode 下，成员表是如何在完

成输入之前找到匹配名的。

```
void CDemoView::OnDraw(CDC* pDC)
{
    CDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Set coordinate system
    pDC->
```



The window appears when the editor detects the pointer-member operator.

```
void CDemoView::OnDraw(CDC* pDC)
{
    CDemoDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Set coordinate system
    pDC->SetM
```



The selection bar locates the complete name as you continue typing.

图 3-7 在文本编辑器中显示的 List Member（成员表）

当输入一个无字符码（如空格或 semicolon）时，系统就将高亮显示的成员名插入到文档中。这个过程听起来挺复杂，但实际过程要简单得多，因为源代码中每一个成员名后都几乎总是跟着一个标志：函数名后跟一个左括号，变量名后跟一

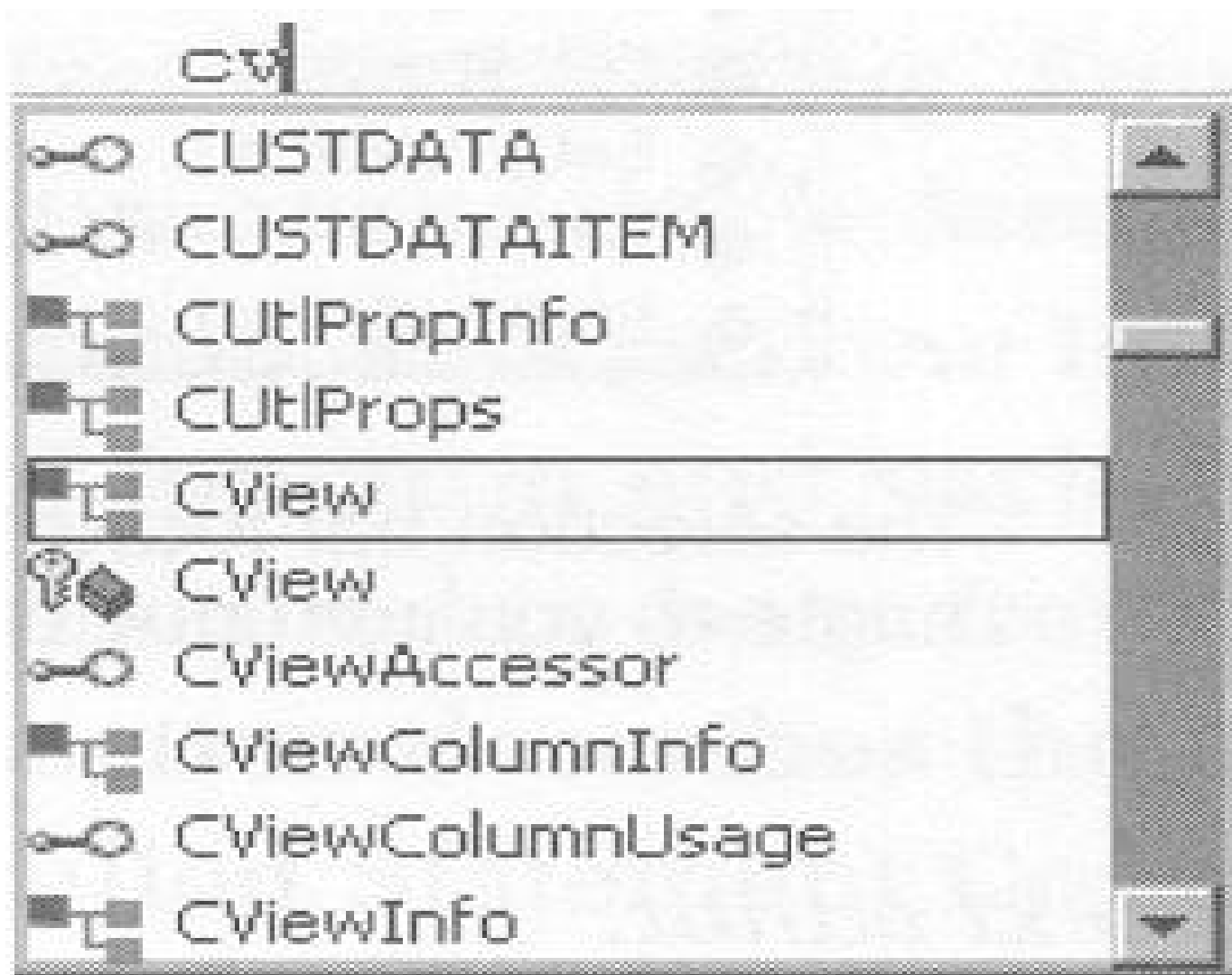
个等号或分号。例如，在图 3-7 右图中，输入一个左括号会产生如下结果：

```
//Set coordinate system
```

```
pDC->SetMapMode(
```

这里，成员表消失，高亮显示的 SetMapMode 被插入到插入符所在处，并在其后加一小括号，从而为继续输入参数做好准备。在下一节我们会看到，Statement Completion（语句补全）并没有到此为止，而是又激活了 Parameter Info（参数信息）工具来帮你继续完成参数输入。当成员表消失后，这个逻辑工具就自动取代了它的位置。

按 Tab 键或 Ctrl+Enter 键，可在不增添字符的情况下完成单词输入，并使成员表消失。按 Ctrl+Alt+T 键，可在任何时刻激活成员表窗口。编辑器总是以插入符左侧的文本为线索来定位选择条。例如，在键入 cv 后激活成员表，此时将显示成员表窗口，并将选择条定位在 CView 处：



这里输入的字符 `cv` 与高亮显示的名称 `CView` 大小写不同。不过，成员表在定位工具栏时忽略大小写，因此，可以全部用小写输入，只是最后需要按 `Tab` 或回车

来确认一下。这个特点有助于输入一些难以记忆的组合函数，如 `UnMapViewOfFile` 和 `UnMapAndLoad` 等。

尽管通过按组合快捷键很容易显示成员表窗口，但如果乐意，还可以通过 `Edit`（编辑）菜单，或在文档窗口内双击右箭头键，激活上下文菜单来打开成员表。成员表窗口中只包括那些在当前类或对象下有效的符号名。例如，当输入一个从 `CDialog` 继承来的目标时，成员表中将包括 `PrevDlgCtrl` 函数，因为这个函数属于这个类。而当输入一个从 `CString` 继承来的类时，这个函数将不存在于列表中。

在 `Edit`（编辑）菜单下还有一个叫 `Complete Word`（补全词）的命令，它看似 `Statement Completion`（语句补全）的另一个工具，其实不过是成员表的一种简短方式。如果不想通过按 `Ctrl+Alt+T` 来激活 `List Members`（成员表）命令，可通过按 `Ctrl+Space` 来激活 `Complete Word` 命令。通常情况下，这两种方法会有同样的结果，都显示成员表窗口，并将选择条置于与正输入的文本最匹配的成员名上。不过，当与所输入的文本最匹配的成员名只有一个时，仍显示成员表就显得多余了。在这种情况下，按 `Ctrl+Space` 将不显示成员表，而直接完成操作。例如，在输入 `CreateMul` 后按 `Ctrl+Spacebar`，就可一步完成输入，因为它可以明确判定想输入的是 `CreateMultiProfileTransform`，而不是 `CreateMutex` 或其他。

参数信息

参数信息包含函数的各个参数，当在函数后输入第一个括号时，参数信息将以一

个工具栏式窗口的形式弹到屏幕上。这种工具栏式窗口像一个屏幕提示，显示出函数原型,并要求输入参数。如下例：

```
m_wndStatusBar.SetIndicators(|  
BOOL SetIndicators (const UINT *lpIDArray, int nIDCount)
```

输入参数时，函数原型一直显示在屏幕上，当输入最后的右括号时，此窗口消失。如果这个函数是过载的，可接受多组参数，则参数信息窗口将显示一个函数原型，并在窗口左端给出数字，来说明此函数有多少过载的版本。按 Ctrl+PgUp，或单击参数信息窗口，可在一个函数原型循环到下一个。如下例：

```
m_wndStatusBar.GetPaneText (|  
1 of 2 CString GetPaneText (int nIndex)  
  
m_wndStatusBar.GetPaneText (|  
2 of 2 void GetPaneText (int nIndex, CString &rString)
```

有一个命令可使得当把插入符放在函数名上时，参数信息窗口会自动显示在屏幕上。可通过 Edit（编辑）菜单或按 Ctrl+Shift+Spacebar 来选择这个命令。也可在文本中用右击一个函数名，来打开上下文菜单，再从中选择该命令。和其他的 Visual C++ 命令一样，也可以设置一个快捷键来激活参数信息工具。本章后面的“未结合命令”一节解释这点。

输入信息

输入信息和参数信息类似，也以工具提示窗口出现，它包含有关函数和变量的信息。当系统认出鼠标光标处的符号时，输入信息窗口将自动显示出来，当把光标移开时，该窗口将自动消失。可通过 Edit（编辑）菜单、上下文菜单或按 Ctrl+T，来调用输入信息功能。如果想了解刚输入的，或刚从成员表中粘贴过来的代码信息，那么，最后一种方法是很方便的。当插入符位于一个函数名上时，输入信息窗口和参数信息窗口所提供的信息是一样的，都列出函数的原型。当调用一个定义的类型时，窗口中显示创建别名时的 typedef 语句：

OLECHAR

```
typedef unsigned short OLECHAR ;
```

当需要显示某个变量的信息时，输入信息就显得很有用。它给出了变量的说明，使得不再需要查询源代码或类的头文件来确定该变量类型。例如，如果变量 indicators 的类型不清楚，使用输入信息窗口，就会立即看到它是一个整数数组变量，如下图：

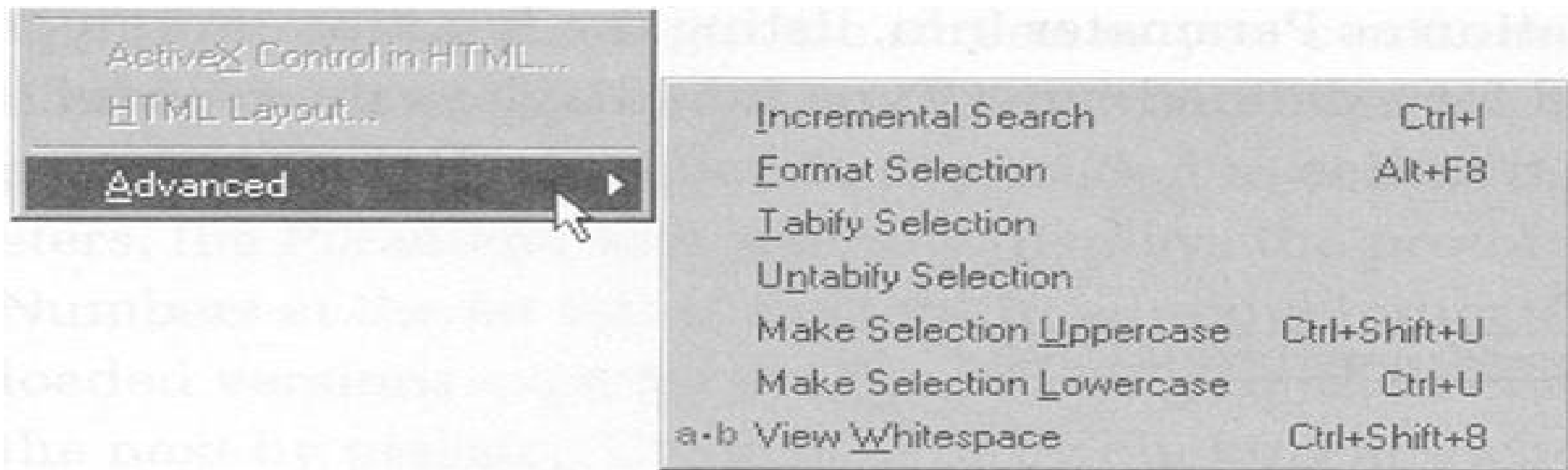
```
// Set status bar panes  
statusbar.SetIndicators( indicators, 3 );
```

```
unsigned int indicators[] ;
```

输入信息窗口以及其他类似的弹出式窗口也许会使你感到眼花缭乱。如果这样，单击 Tools（工具）菜单下的 Options（选项）命令，再清除对话框中 Editor（编辑器）选项卡下的相应复选框即可（参看图 3-11）。通过相应的快捷键或菜单命令，可再次激活这些工具。

Advanced（高级）命令

Edit（编辑）菜单底部附近的 Advanced（高级）命令提供了一个选项集合，它们在进行文本文档工作时非常有用。让光标在 Advanced（高级）命令上停留一会儿，会显示如下所示的子菜单。



正如你所看到的那样，菜单提供了对前述 Incremental Search（增量搜索）命令的访问，尽管按 Ctrl+I 是访问该命令的更为方便的办法。Format Selection（格式选择）命令通过插入制表符，来设置花括号内的 C/C++ 代码的缩进。这个命令把下面的代码：

```
if(msg=WM_USER)
```

```
{
```

```
for(i=0;i<5;i++)
```

```
{  
  
//Additional code  
  
}  
  
}
```

变成：

```
if(msg== WM_USER)  
{  
  
    for(i=0;i<5;i++)  
  
    {  
  
        //Additional code  
  
    }  
  
}
```

Format Selection (选择) 命令通过扫描为花括号选择的文本来决定嵌套层次。第一层的文本行缩进一个制表符，第二层的文本行缩进两个制表符，依此类推。

Tabify Selection (制表符选择) 命令把选中的一系列空格转换成等量的制表符。Untabify Selection (取消制表符选择) 则把这个操作反过来。通过打开 View Whitespace (查看工作空间) 双态开关，可清楚地看到两个命令的效果，这时文档中的空格和制表符都变成可见的。当该开关打开时，文本中的空格字符显示为小圆点(.)，而制表符显示为书名号(>>)。

子菜单中剩下的两个命令的作用就像它们的名字所表示的那样。Make Selection Uppercase (使选择内容成为大写) 命令使选中的所有字母都变成大写，而 Make Selection Lowercase (使选择内容成为小写) 命令则恰恰相反。选择中的非字母字符，如数字和标点符号，则不受这两个命令的影响。

未结合命令

每个 Visual C++ 命令都有一个描述性的内部名称。例如，刚刚说过的 Incremental Search (增量搜索) 和 Tabify (转换制表符) 命令的内部名称为 SearchIncremental、SelectionTabify 和 SelectionUntabify。在线帮助提到了许多你在菜单中找不到的其他命令，例如名为 GoTo (转到) NextErrorTag, LineTranspose 和 LineDeleteTo Start 的命令。有两个原因来解释为什么帮助系统宁愿用内部名称，而不用组合键，如

F4 或 Shift+Alt+T 来标识命令。首先，你可以把命令的组合键改成你喜欢的任意键。其次，许多命令还没有指派快捷键。这样的命令称为“未结合的”。要使用一个未结合命令，你必须首先给它指定一个组合键。

在 Developer Studio 环境下，隐藏着一大套命令，除那些出现在菜单和工具栏按钮上的命令之外，还有更多的命令可以使用。在 Help(帮助)菜单中单击 Keyboard Map (敲键映射) 来查看命令列表，如图 3-8 所示。

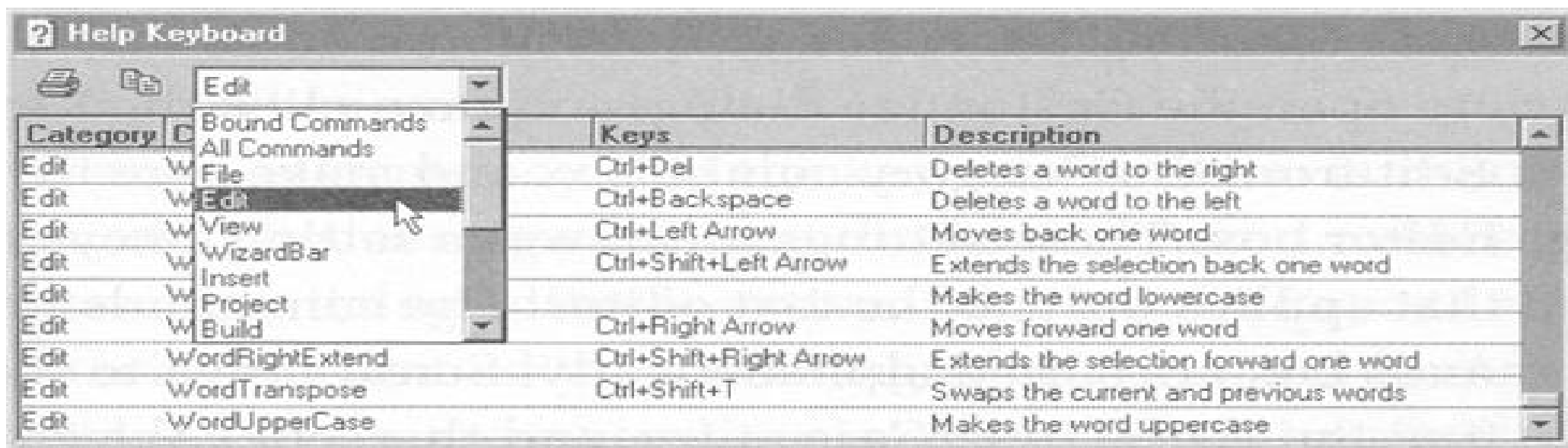


图 3-8 从 Help 菜单中选择 Keyboard Map，以显示 Visual C++ 的命令列表

在 Help Keyboard（帮助键盘）窗口中出现的默认列表称为“结合命令”，这意味着，这些命令已经指派了组合键。要查看只属于文本编辑器的结合命令和未结合命令列表，在下拉列表框中选择 Edit（编辑），并在第二栏的上边单击 Command（命令）按钮，来按命令的字母顺序排列列表。当你滚动列表时，你会在 Keys（键）栏中看到最常用的命令已经约定了组合键，但许多命令还没有。未结合命令的设置使许多特征变得可用，否则它们就不能通过菜单、工具栏或键盘来访问。

如何访问命令完全取决于你的喜好。Visual C++ 允许你向菜单或工具栏中添加任

何结合的和未结合命令，这将在第 13 章中详细讨论。但因为拥挤的菜单和工具栏反而容易误事，最好通过给未结合命令定义一个组合键来调用它。唯一的缺点是你必须记住访问命令的快捷键。

没有人会打算一次就启用所有的未结合命令。选择那些你认为最有用的命令，给它们指派最符合你的风格和最容易记忆的组合键。举个例子，让我们给文本编辑器添加两个非常有用的命令：WordUpperCase 和 WordLowerCase，它们改变当前文本中插入符之下的单词的大小写。默认情况下，WordUpperCase 和 WordLowerCase 没有被指派组合键，也没有工具栏按钮或菜单选项来访问它们。除非你给它们指定组合键，没有别的办法来使用这两个命令。

下面是调用命令的方式：从 Tools（工具）菜单中选择 Customize（自定义），来打开 Customize（自定义）对话框，然后单击 Keyboard（键盘）选项卡。从 Category（分类）下拉列表中选择 Edit（编辑），并确保 Text（文本）出现在 Editor 框中。这些设置表示我们正为一个只用于文本编辑器的命令定义一个组合键。在 Commands 列表框中出现的命令是按字母顺序排列的。向下滚动列表到底部，找到 WordUpperCase 条目，然后单击它来选中它。对话框的左下角会出现命令的简短说明，但 Current Keys（当前键）列表框中是一片空白，这表示目前没有给 WordUpperCase 定义快捷键。要定义一个快捷键，单击 Press New Shortcut Key（按新的快捷键）文本框，并按下你想用来访问该命令的组合键。如果你按下 Ctrl+U，对话框提示你该组合键目前已指定给 SelectionLowercase 命令。这并不意味着你不能把 Ctrl+U 指定给 WordUpperCase；它只是提示你，如果你这样做

的话，按 `Ctrl+U` 将不再调用 `SelectionLowercase` 命令，它会变成一个未结合命令。因为 `Ctrl+U` 已经使用了，`Alt+U` 对 `WordUpperCase` 来说是一个更好的选择。当你按下 `Alt+U` 时，对话框告诉你这个快捷键目前没有定义给别的命令（见图 3-9）。单击 `Assign`（指定）按钮后，快捷键就可以使用了。

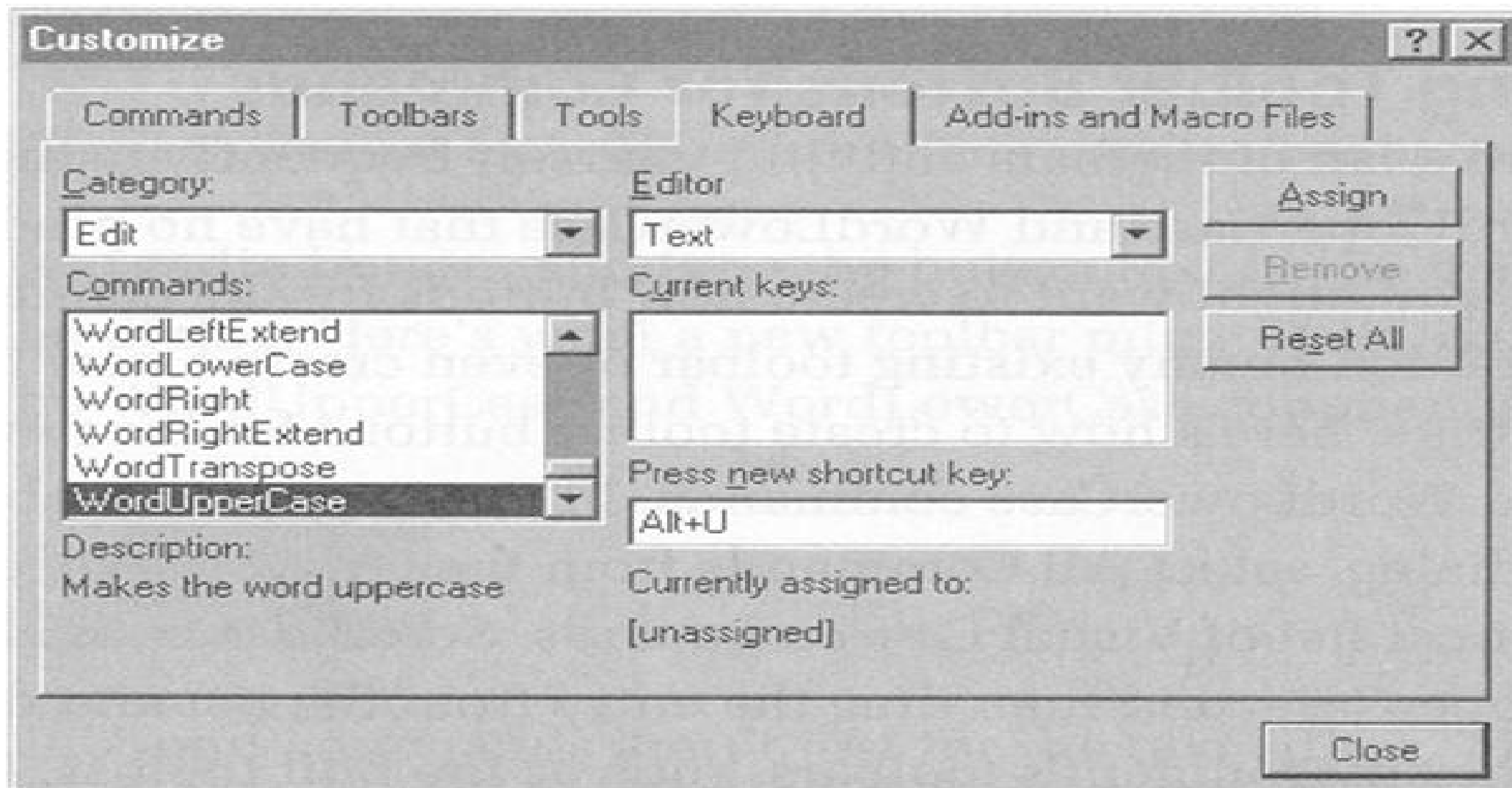


图 3-9 对文本编辑器命令指定一个键组合

同样给 WordLowerCase 命令定义快捷键 Alt+L。当你在 Press New Shortcut Key 文本框中按下 Alt+L 时，系统会提示你这个组合键被用来访问菜单。提示信息提到 Layout（布局）菜单，它只在对话框编辑器调用时才能使用。因为 Layout（布局）菜单对文本编辑器没有影响，选择 Alt+L 不会导致快捷键冲突。当文本编辑器调用时，Alt+L 调用 WordLowerCase 命令；当对话框编辑器调用时，Alt+L 像以前一样拉下 Layout（布局）菜单。

要使用新的 WordUpperCase 和 WordLowerCase 命令，打开一个文本文档，并随便在一个单词上放一个插入符。按下 Alt+U 或 Alt+L 调用命令来改变从插入符到单词末尾的所有字母的大小写。同样地，新的命令会覆盖 SelectionUppercase 和 SelectionLowercase 命令，因为它们作用在任何选中的文本块上，而不是在单一的单词上。现在 Selection（选择）命令是多余的了，没有关系。新的 Alt+U 和 Alt+L 快捷键比起等同的 Shift+Ctrl+U(SelectionUppercase) 和 Ctrl+U(SelectionLowercase)组合键来更容易使用和记忆。然而这种覆盖只适用于选中的文本，因为当没有选择文本时 SelectionUppercase 和 SelectionLowercase 会影响插入符附近的字符。

为命令创建工具栏按钮

如果你喜欢使用工具栏，并发现自己经常使用诸如 WordUpperCase 和 WordLowerCase 之类没有预定义工具栏按钮的命令，你可能想为这些命令创建新

的按钮。你可以把按钮放在任何已有的工具栏上，或者创建一个新的工具栏。下面演示如何为新的 WordUpperCase 和 WordLowerCase 命令创建工具栏按钮。在 Customize（自定义）对话框的 Commands 选项卡中，从 Category（分类）框中选择 All Commands（所有命令），来显示按字母顺序排列的 Visual C++ 命令列表。滚动列表，来找到 WordUpperCase，然后从列表中拖动该条目，把它放在环境的一个工具栏，如 Edit 工具栏上。如果你更愿意创建一个新的工具栏，只要把 WordUpperCase 条目拖出对话框，并放在屏幕上没有被工具栏覆盖的地方，就可以了。Visual C++ 自动创建一个新的工具栏来放置按钮。从列表中把 WordLowerCase 命令拖到同一个工具栏中。

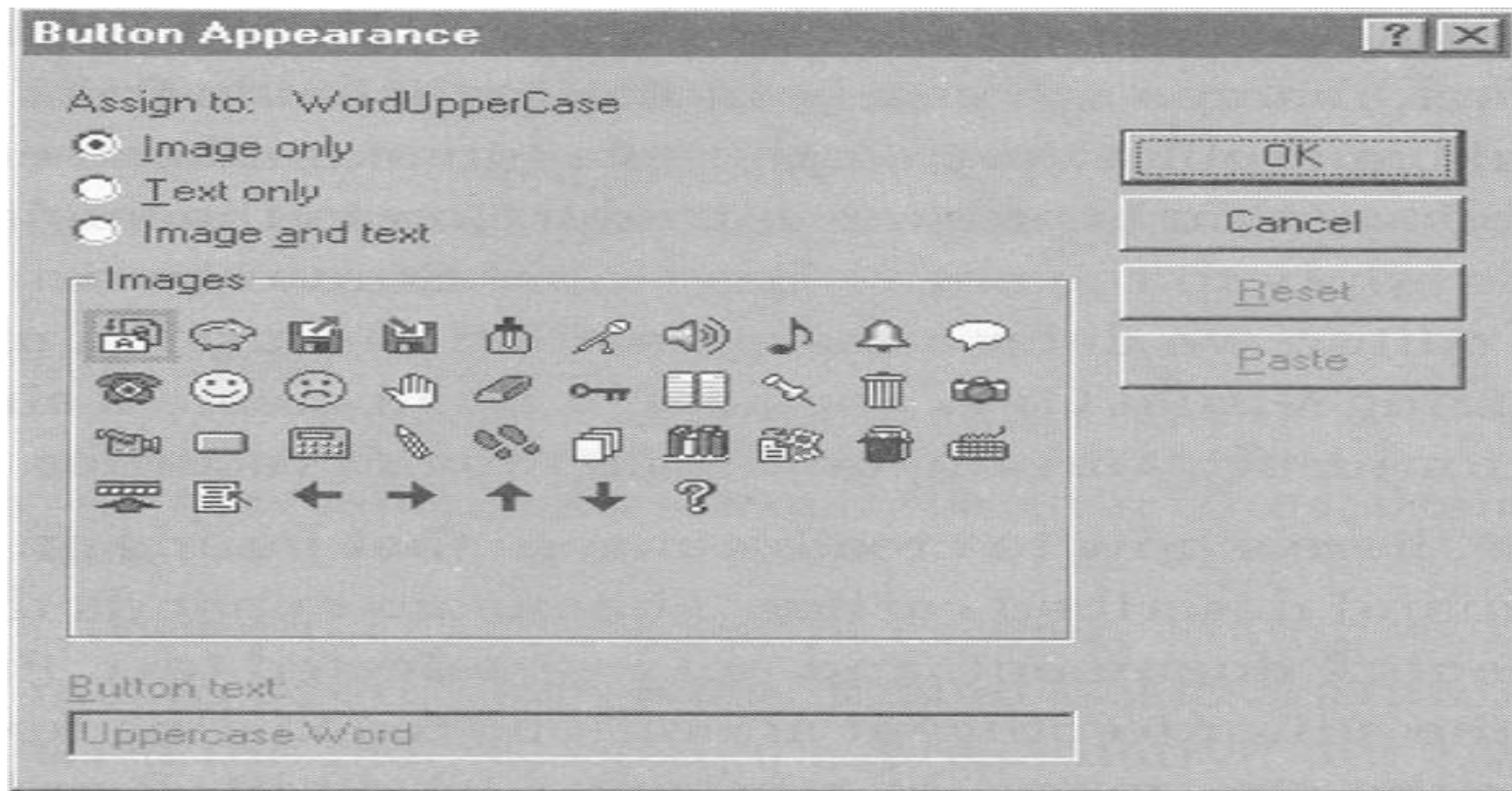


图 3-10 Visual C++ 提供新工具栏按钮的图标选择

因为 WordUpperCase 和 WordLowerCase 命令没有预定义的图标，Button

Appearance（按钮外观）对话框打开（图 3-10），在该对话框中，你可以为每个新的按钮选择一个图标。在可用的图标中，没有一个能反映 WordUpperCase 和 WordLowerCase 的不寻常的功能，但你可以把图像和文字组合起来，以使按钮的功能变得明确。在 Button Appearance（按钮外观）对话框中选择一个图标，单击 Image And Text（图像和文本）单选钮，然后，在对话框底部的文本框中键入按钮文本。下面是带有 WordUpperCase 和 WordLowerCase 命令的按钮的新工具栏的样子：



未结合命令不只是用于文本编辑器。下一章描述如何用同样的方法来通过快捷键或工具栏按钮来执行图形编辑器中非常有用的命令。第 13 章中详细地讨论了在 Visual C++ 环境中创建工具栏的主题，解释了如何重命名或删除工具栏，如何从一个工具栏中向另一个工具栏中复制按钮，以及如何定制按钮图标。

宏的基础

你可以把一个结合命令当作一个预定义的宏，即指派给一个组合键的用法说明。环境也允许你创建用于文本编辑器的宏，你可以通过记录按键和鼠标单击活动，来把它们组合成普通的 Visual C++ 命令中的一个可重复使用的命令。你可以像执行其他命令一样通过快捷键、菜单命令和工具栏按钮来执行一个宏。实际上，宏同普通的结合命令几乎没有区别，这使它们成为扩展环境功能的一个高级方法。这一部分只介绍一下宏的主题。因为宏用在整个 Developer Studio 环境中，而不只是用于文本编辑器，我们在第 13 章再作更详细的讨论。现在，我们可以简单地通过打开命令录制器来为文本编辑器创建一个宏。

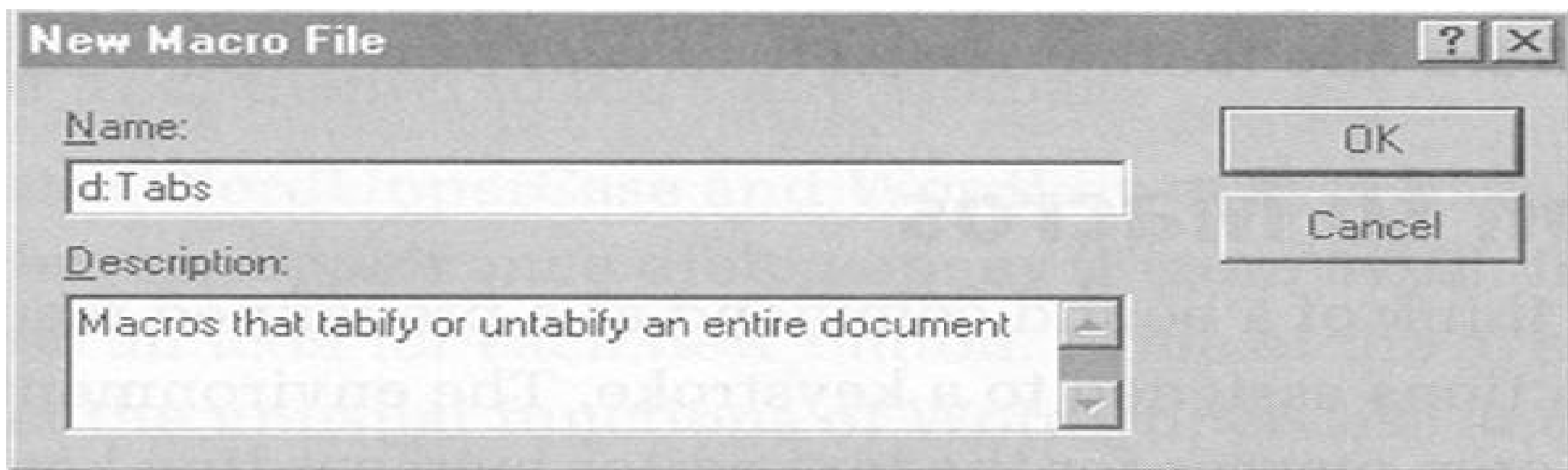
下面演示如何创建一个建立在前面说过的 Untabify Selection（解除制表符转换选择）命令上的宏。该宏把命令的功能扩展到整个文档，而不仅仅是选中的文本。首先关闭文本编辑器中所有的文档，来防止改变已有的文本，然后通过按下 Ctrl+Shift+R，或从 Tools（工具）菜单中选择 Record Quick Macro（快速录制宏）命令，来开始录制新宏。这会打开 Record（录制）工具栏，并把鼠标变成一个盒式磁带形的图像，这表示 Visual C++ 正在录制每一个按键和鼠标单击动作。该宏包括下面四个步骤：

1. 在 Edit（编辑）菜单中单击 Select All（全选）命令来选中整个文档。
2. 从 Edit（编辑）菜单中选择 Advanced，并在子菜单中单击 Untabify

Selection 命令。

3. 按下 Ctrl+Home 来在文档开头返回一个插入符。

4. 在 Record（录制）工具栏上单击 Stop Recording（停止录制）按钮来结束录制过程。



现在，我们有了一个新的宏。Visual C++以 GlobalTemporary.dsm（该扩展名代表 Developer Studio 宏）。的文件名把宏存储在 Common\MsDev98\Macros 的文件夹下。该文件包含一个简单的 Visual Basic 子程序，其中的语句调用了我们刚才录制的三个命令：

```
Sub GlobalTemporary
```

```
    ActiveDocument.Selection.SelectAll
```

```
    ActiveDocument.Selection.Untabify
```

```
    ActiveDocument.Selection.StartOfDocument
```

```
End Sub
```

附录 C 更详细地说明了宏的源文件语言：Visual Basic Scripting Edition。要检验这个宏，打开一个典型的文档，并在 Advanced（高级）子菜单中打开 View Whitespace（查看空白空间）命令，来使你能够看到宏的效果。现在通过按下 Ctrl+Shift+P 或在 Tools（工具）菜单中单击 Play Quick Macro（快速播放宏）命令来运行宏。效果同重新键入录制的动作一样。

GlobalTemporary 宏是独特的，因为它没有出现在图 3-8 所示的 Visual C++ 命令列表中。它被保留为通过 Record Quick Macro 命令创建的“快速”宏。通过同样的命令来录制另一个宏，就会覆盖前一个宏，因此，一次只有一个快捷宏存在。

定制编辑器

Visual C++ 文本编辑器可以改变自己的许多特征，以更好地适应你的工作风格。我们已经讨论过 Tools（工具）菜单中的 Customize（自定义）命令，它让你定制工具栏，并给命令约定快捷键。要改变编辑器界面的其他特征，从 Tools（工具）菜单中选择 Options（选项）命令。

Options（选项）命令显示图 3-11 所示的对话框，它允许你指定文本编辑器的下列特征：

- 外观、文档保存和 Statement Completion（语句补全）选项
- 制表符和缩进
- 模拟
- 字体

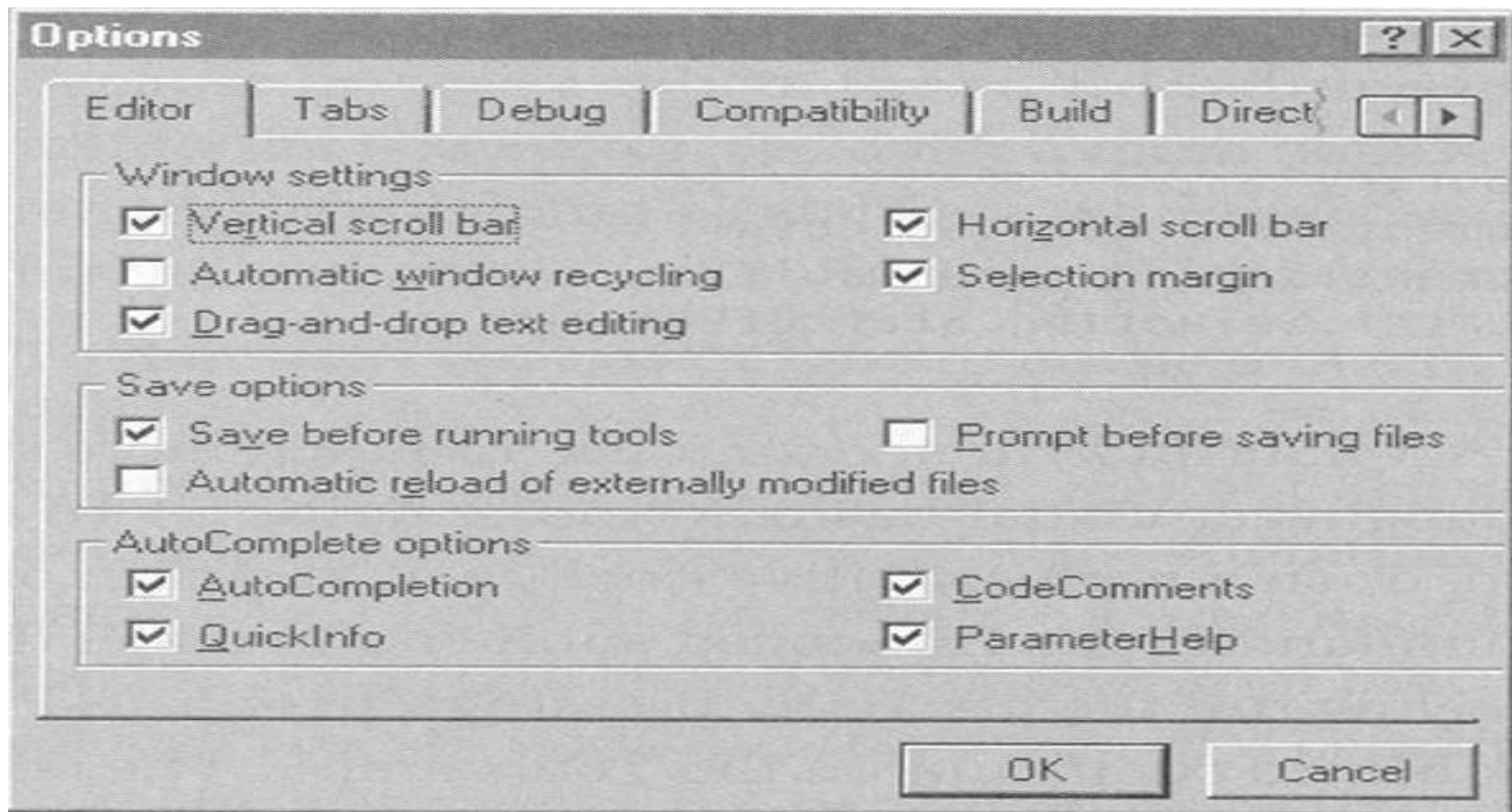


图 3-11 Options 对话框的 Editor 选项卡

在对话框的 Edit 选项卡中，根据你的爱好来选择编辑器如何及何时需要保存文档（下面将详细说明 Automatic Reload Of Externally Modified Files（外部修改的文件自动重新加载）复选项。你也可以指定编辑器在编译之前是否自动保存已改变的文件，以及在保存文档之前是否提示你。

值得一提的是同一页中的 Selection Margin（选择边界）复选项。选择边界是文档窗口左边的大约半英寸宽的一列阴影区。通过单击该边界，你就可以选中附近的一整行。该空格也可以保留书签的图标，以及后面我们将在第 11 章中提到的调试断点。如果你宁愿恢复这些空白来显示文档，清除掉此复选项，来禁用选择空格。

Visual C++ 文本编辑器可以在一定程度上模拟 BRIEF 或 Epsilon 程序员的编辑器。如果你习惯于这些产品，可以打开相应的模拟选项。单击 Compatibility（兼容性）选项卡，然后从列表中选择编辑器中的一个，或通过打开单个的复选项来设置想要的选项。

Format（格式）选项卡允许你给编辑器窗口指定字体类型和颜色。从 Category 列表中单击 Source Windows（源窗口），来查看当前的字体。默认情况下，字体为 10 号 Courier，但你可以把它改成任何你所想要的类型和大小。Color 区允许你为编辑器中不同的标记和文本（如源文件注释和 HTML 标记）调整背景和前景颜色。要改变颜色，从列表选择一个条目，然后从组合框中选择想要的颜色。

在 Developer Studio 之外编辑文本

文本编辑器同用户喜欢的字处理程序共享了许多特征。Visual C++ 给你提供了一个非常强大的程序编辑器，但是，如果你现在喜欢使用另一个编辑器，也可以。使用一个你非常熟悉的产品效率可能会更高一些。并且，如果你从来没有使用过 Developer Studio 环境，除了使用别的编辑器外，你没有别的选择了。Visual C++ 编辑器是 Developer Studio 整体的一部分，而不是一个独立的程序。你只能从 Developer Studio 中访问该编辑器。

Visual C++ 文本编辑器的一个很大的优点是，展示了一个集成的开发环境的整体应是什么样子。当编译器在源代码中发现错误时，它自动把编辑器的光标设置在第一个出错的语句上，准备让你改正错误。双击列表中的下一个错误，光标就会移动到源文件中的正确位置上。编辑完后，只需单击 Build（建立）工具栏上的 Compile（编译）按钮，就可以把修订过的文本重新提交给编译器。Visual C++ 自动把新的源文件保存到磁盘上。在环境之外的编辑器中工作，就需要更多的努力。你必须切换到编辑器，把光标移到编译出错的语句行，改正之后，保存文件，然后再切换回 Visual C++ 来重新编译。

如果你决定用另一个文本编辑器来编写和维护源代码，你应该对 Visual C++ 环境作两个小小的改动。首先，如果你打算长期使用另外的一个编辑器，你可能会发现从 Tools（工具）菜单中通过编辑器的专用命令来运行它更为方便。通过在 Tools（工具）菜单中放置一个新的命令，你可以在环境内部启动编辑器。如果你的编

辑器从命令行接受文件名，可以配置该命令，以使编辑器在启动时自动加载源文件。添加一个新的 Tools（工具）命令，来启动一个文本编辑器（或任何其他外部程序），这将在第 13 章中详细讨论。

应该做的第二个改变是对默认设置的细小改动。在另一个编辑器中打开一个文件时，通常在 Visual C++ 中会打开同一个文件。这发生在编译和调试代码的过程中，因为调试器要加载源文件。当你在编辑器中改变，并保存文件后，切换回 Visual C++ 来重新编译时，环境可以识别到打开的文件副本不再是当前的内容了。默认情况下，系统会显示图 3-12 所示的消息框，它提示你从磁盘上重新加载新的文件。你的回答通常会 Yes，或者至少你不会很在意。为了防止每次你切换回 Visual C++ 时出现这个消息，在 Tools（工具）菜单中单击 Options（选项）。选中图 3-11 所示的 Automatic Reload Of Externally Modified Files 复选项，来让编辑器自动加载改变了的文件，而不再提示你。

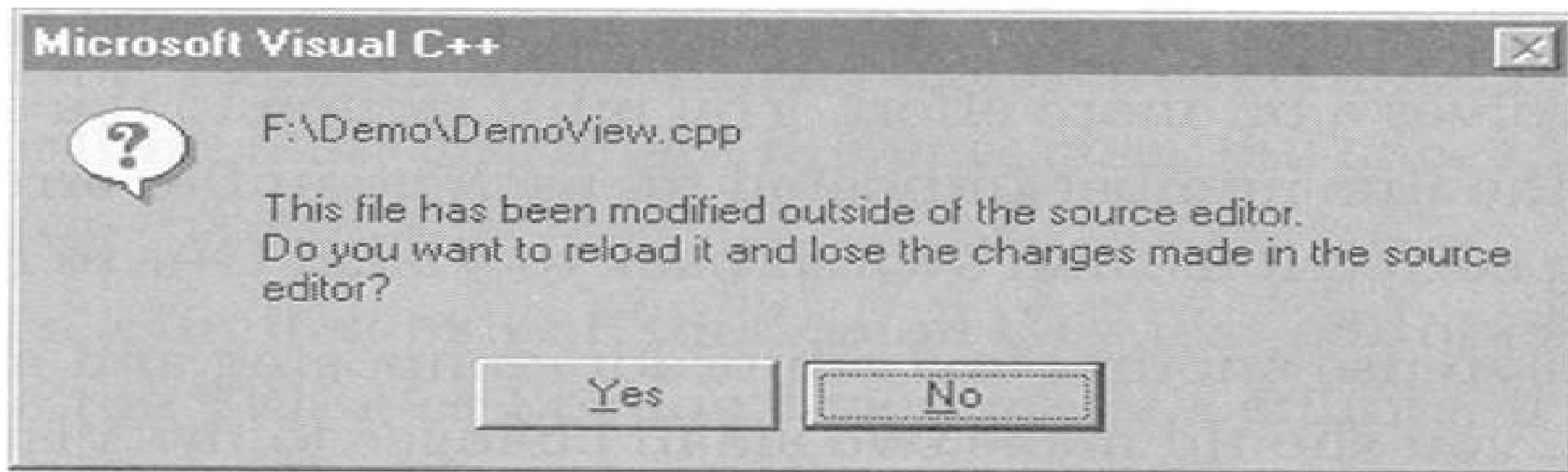


图 3-12 在外部编辑器已经改变了一个文件时，解析文档版本

当你使用另外的编辑器时，你就已经与 Visual C++ 签订了这样的协议：你不会在两个编辑器中同时改变文档。然而，环境可以处理对文档的同时改变。Visual C++ 承认外部改动，因为当环境重新获得输入焦点时，它检查所有在文本编辑器中打开的文件的日期和时间标志。出于安全的目的，环境重新加载拥有更新标志的文件，但 Automatic Reload Of Externally Modified Files 设置仅适用于那些编辑器自己的文档副本没有改变的情况下。如果编辑器中的文档已经被改变，即使是一个字符，不论 Automatic Reload Of Externally Modified Files 复选项如何设置，Visual C++ 都显示图 3-12 所示的消息。这使你可以控制文档的哪个版本是正确的。

第 4 章 资源

一般地，当我们说起一个程序的数据时，我们指的是那些用 `X` 或 `pString` 命名的源代码里的变量。一个典型的 Windows 应用程序也有另一类叫做资源的数据，它包括文本和图形，这些决定程序用户界面的外观和感觉。一个程序的资源定义界面元素如下：

- 菜单
- 加速键
- 位图、光标和图标
- 对话框和控件
- 字符串
- 工具栏

当 Windows 调用一个程序时，它从程序的可执行文件里读取程序代码和初始化

数据的数值，并把它们拷贝到分配的内存里。有些情况下，资源数据保留在程序硬盘上的可执行文件里。资源在运行时读取，而不是在装载时读取，实际上，它是程序创建窗口、显示对话框或装载位图时，从 EXE 文件或 DLL 文件里读取的。Visual C++ 提供一些资源编辑器，通过它，你可以创建或编辑一个项目的资源数据。出于某些考虑，本章是前面几章的延续，它描述怎样通过 Visual C++ 的文字编辑器创建和编辑文档。虽然这里“文档”这个词的定义已经扩展，它包括的内容不仅仅有包括纯文本，还有其他格式，这些格式的编辑原理是相同的。

有关资源和资源编辑器的主题范围广泛，它占了本章和下一章的内容。本章覆盖了用户在应用里首先会遇到的界面元素，这种资源数据包括菜单、工具栏、加速键、图标和鼠标光标。有关对话框和控件的主题放在第 5 章里。

系统资源

为了以后清楚所讲的内容，我也应该谈谈系统资源，它是 Windows 提供给应用程序的公共资源数据集。系统资源实际上是借给应用程序的，要知道，在应用程序释放一个句柄时，它以显式的方式归还给系统，在应用程序终止时，它以隐式的方式归还给系统。某些系统资源，例如鼠标光标，是自动提供的，这样，就不需每个应用程序创建自己的资源了。虽然一个应用程序可以显示它自己独特的光标（正如我们在本章后面将看到的），但是，使用箭头、沙漏和其他系统提供的位图更容易些。在对程序员提供方便的同时，它也可以保证用户在程序之间切换

时，不会对各种光标感到不知所措。

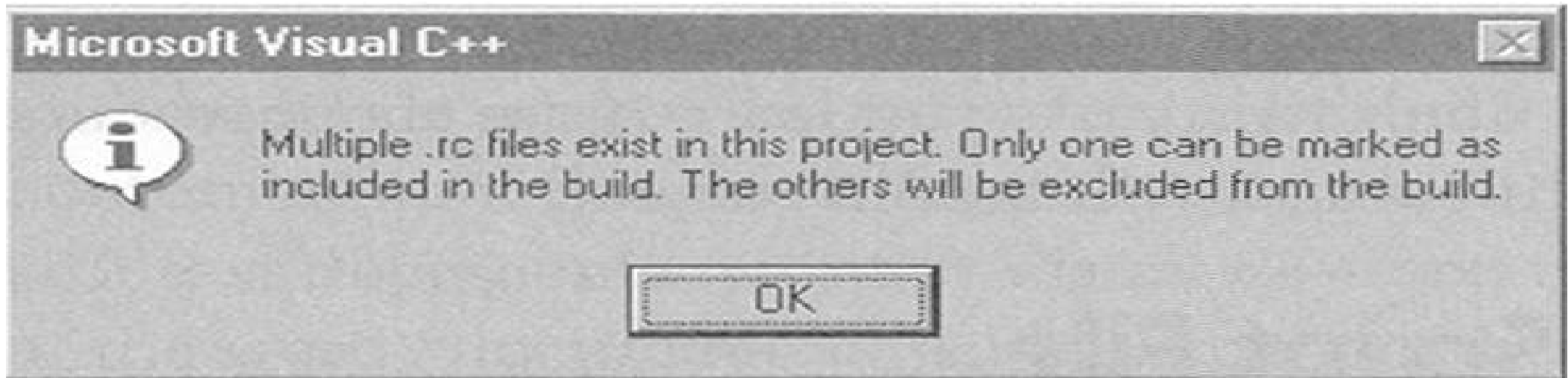
其他的系统资源，例如设备上下文和系统插入符，不能够被一个应用程序复制，也不能够被真正类比为本章中所描述的程序资源。虽然在一个程序的外观上，内存没有起到什么作用，但是，它也是一种被经常谈及的系统资源，因为它作为堆分配给程序，这是程序所要求的。控件是另一种当我们谈及资源时容易混淆的潜在的资源。在一个程序的资源数据里，程序声明的仅仅是它想使用的控件类型，例如窗口的坐标，或者可能是初始化状态。控件的显示和操作是由系统传递的。如果系统资源和程序资源的界限有时看起来不明了，那么不要着急。当你在本章中学习更多的资源时，两者之间的差别将变得非常明显。

RC 资源脚本文件

一个项目在源文件里定义它的资源，资源文件以 RC 为扩展名，并且一般和项目具有相同的名字。RC 文件只包含文本，它更像是一个程序资源文件，所以，你可以用一个文本编辑器查看它。在文件里，你将发现一些定义字符串和菜单内容的表格，但是，没有包含位图和图标的图形数据。图形资源保存在单独的文件里，它的名字和在磁盘中的位置保存在 RC 文件里。RC 这个扩展名指示出这个文件为资源编译器提供像源代码那样的服务，资源编译器是 Visual C++ 的一个独立部分，它把程序的文本和图形资源编译成目标文件形式，这样，链接器可以把它联编到 EXE 文件里。一个项目的 RC 文件经常叫做资源脚本或资源定义文件。

资源脚本文件是可选的。不与用户交互的窗口应用程序不需要资源，对一个程序来说，更可能在运行时随时创建所有的资源。但是，正如你将在本章中看到的那样，资源脚本利于开发者进行开发，因为资源脚本从源代码中分离出用户界面元素。通过使用 Visual C++ 资源编辑器，开发者可以设计一个界面，看到它是什么样子，并且可以通过少许鼠标单击来改变它。并且，由于程序要面向国际市场，所以资源脚本是必须的，因为它们允许转换者修改用户界面，而不用接触程序源代码。

一个 Visual C++ 项目只认可一个主要的 RC 文件。如果你想使用 Project (项目) 菜单里的 Add To Project (添加到项目) 命令添加一个额外的 RC 文件，集成环境会警告你，当编译这个项目时，这个文件将不会被编译，如下所示。



然而，一个项目可以有任意数目的资源脚本文件，但是，所有的二级文件只能通过 `#include` 语句添加到 RC 文件里。例如，AppWizard（应用程序向导）自动创建第二个资源文件，这个文件以项目文件名作为名字，以 RC2 作为扩展名，它提供了一个存放你以前开发和测试的任何资源的好地方，你不需进一步的修改。为了看看 RC2 文件怎样包含在 AppWizard（应用程序向导）创建的项目里，打开项目，并从 View（查看）菜单里选择 Resource Includes（资源包括）项，然后在 Compile-Time Directives（编译时伪指令）控件里向下滚动，到下面这一行（在这里 project 是项目名）：

```
#include "res\project.rc2"    //non-Microsoft Visual C++ resource
```

包含在补充文件里的所有资源均将编译，并链接到项目的可执行文件里。但是，

当你使用一个 Visual C++ 资源编辑器对主 RC 文件操作时，将不能访问它。这就是为什么只有完成的和测试过的资源才应该包含在 RC2 中的原因。资源编译器读取所有的脚本文件，并产生一个以 RES 做后缀的编译过的二进制文件，它类似于 C/C++ 编译器产生的 OBJ 文件。

在第 1 章中描述的作为环境的 Workspace（工作空间）窗口，列出了在主 RC 文件里定义的项目资源。当项目打开时，单击 ResourceView（资源查看）选项卡，并通过单击靠近文件夹图标的加号展开列表。为了在适当的编辑器里打开一个资源（我们将立刻做这件事情），双击列表里的资源。图 4-1 显示了 ResourceView（资源查看）窗格中的资源，这是名为 Demo 的 AppWizard（应用程序向导）项目中的。

通过 AppWizard（应用程序向导）创建的默认 RC 文件是广泛的，包括冗长的字符串表、菜单脚本和与交叉开发平台有关的代码。如果你接受创建新的应用程序时 AppWizard（应用程序向导）产生的所有默认资源，那么，最终将得到一个将近 400 行的 RC 文件。你可以尝试在一个文本编辑器里修改 RC 文件，删除通过 AppWizard（应用程序向导）产生的多余的代码行，减少文件尺寸到易于管理的大小。但是，这样做意味着，以后当你使用一个 Visual C++ 的资源编辑器修改一种资源时，就会有麻烦。虽然你可以得到一个有效的 RC 文件，但是，不幸的是，Visual C++ 不再认为它是 AppWizard（应用程序向导）产生的。你仍然可以通过一个编辑器修改资源，但是，当你保存修改后的文件时，Visual C++ 用一个新的 RC 文件重写你的最低要求的 RC 文件，它包含许多你在前面

已经删除的由 AppWizard（应用程序向导）生成的多余代码。唯一可供选择的是用一个不同的文件名保存修改后的文件，然后，使用文本编辑器从新文件里拷贝你想要的行，并粘贴到最初的资源脚本。我推荐你学习使用由 AppWizard（应用程序向导）产生的巨大的 RC 文件，除了一些小的变化外，只通过资源编辑器来修改资源。

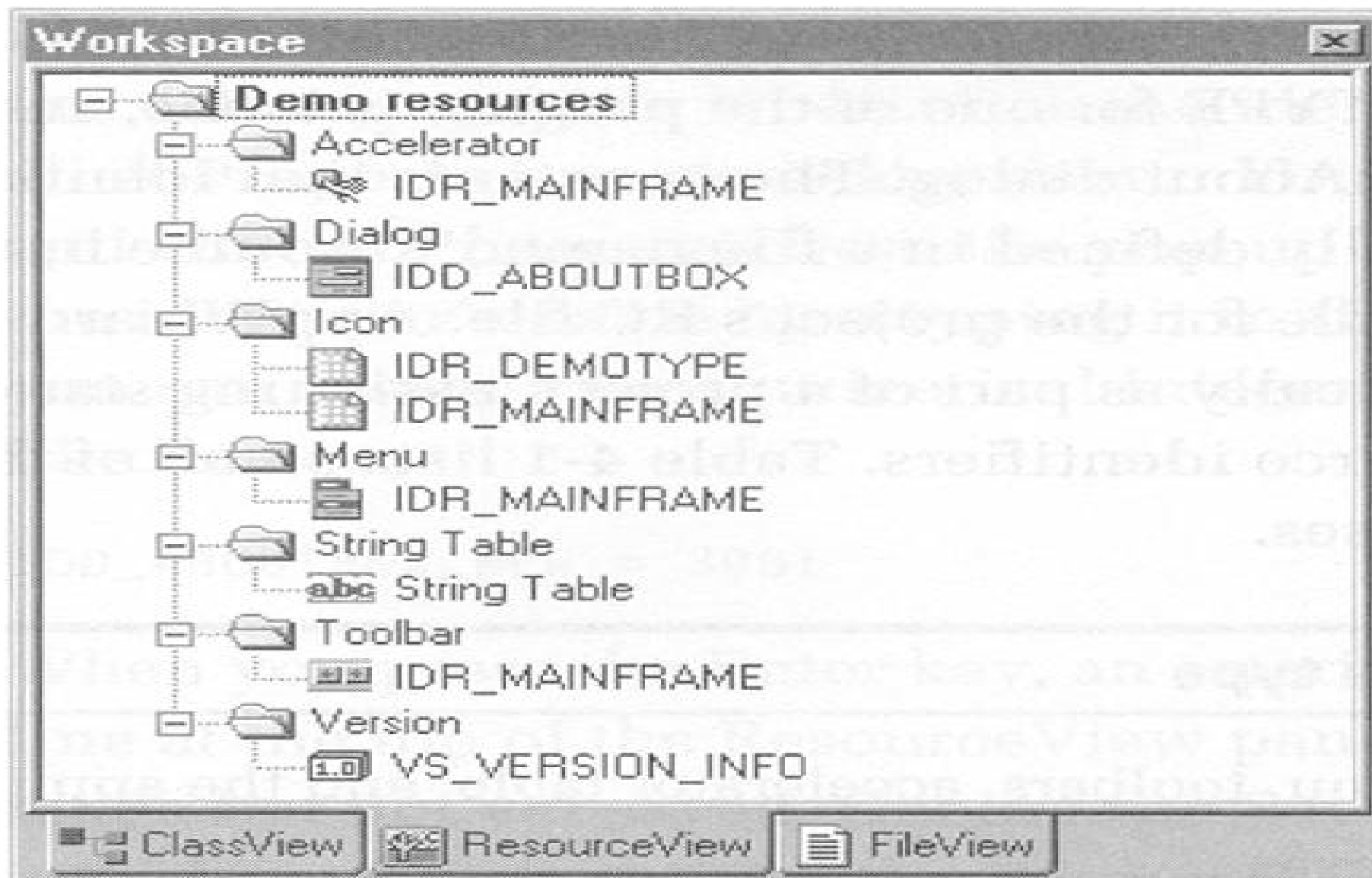


图 4-1 Workspace 窗口的 ResourceView 窗格

Resource.h头文件

项目里的每一个资源或者通过常量标识符，或者至少通过一个字符串名，来加以区别。例如，在图 4-1 中，假想的 Demo 程序的资源都用恒定的值标识：IDR_MAINFRAME 是菜单和工具栏，IDR_DEMOTYPE 是程序的一个图标，IDD_ABOUTBOX 是 About（关于）对话框。用于标识一个项目的资源常量一般是在以 Resource.h 命名的文件里。AppWizard（应用程序向导）自动创建 Resource.h 作为项目的一部分，对资源标识符指定标准 MFC 的资源前缀。表格 4-1 列出一些 MFC 使用的标识前缀。

表 4-1 标准 MFC 标识符前缀

前缀	资源类型
IDR_	主菜单、工具栏、加速键表和应用程序图标
IDD_	对话框

续表

IDC_	控件和光标
IDS_	字符串
IDP_	为消息框提示字符串
ID_	菜单命令

常量标识符可以由字母（大写或小写）、数字和下划线组成，但是不能以一个数字开头。

C 程序把标识符成员认为是明显的常量或“定义”，但是，Visual C++有时将它们作为符号。从技术上来说，符号是在源代码里标明内存地址的名字，例如一个变量或一个函数名字。我们将在第 11 章里看到，编译器怎样生成一个程序里的符号列表，通过阅读它，调试器可以知道程序里的变量和函数名。不要混淆资源符号和源代码里的符号。毫无疑问，Visual C++的设计者选择单词符号，他们认为资源脚本也是一种源代码和资源标识符，类似于程序源代码里的变量名字。

你可以改变列在 Workspace（工作空间）窗口里 ResourceView（资源查看）窗格里的一个资源标识符的名字，或以数字表示的值。首先通过展开适当的文件夹图标，找到标识符名字，如图 4-1 所示。然后，单击列表里的标识符选择它，并单

击 View (查看) 菜单里的 Properties (属性)。你也可以右击一个标识符, 并从弹出的上下文菜单里选择 Properties。在任何一种方法里, 都要通过在 ID 控件里重新输入标识符名字来改变它。同时, 你可以像下面这样给名字添加一个以数字表示的整型值, 来给它赋值:

```
IDD_ABOUTBOX_NEW    =    3001
```

按下 Enter 键时, 一个星号出现在 ResourceView (资源查看) 窗格顶部的项目行附近, 它表明你已经改变了它, 但是还没有保存。从 File (文件) 菜单里选择 Save (保存) 命令, 使 Visual C++ 重写 Resource.h 文件, 用新的标识符代替 #define 语句里旧的标识符。

从理论上来说, 你可以给一个资源标识符赋以从 1 到 65535 (0xFFFF) 中的任何一个值。然而, Windows 为系统菜单项保留了 0xF000 及它上面的值, MFC 为内部使用保留了从 0xE000 到 0xEFFF 之间的值, 所以, 你应该使自己的标识符值保持在 1 到 57343 之间 (0xDFFF)。标识符的值被限制在 WORD 大小, 而不是 DWORD 大小, 这是因为 WM_COMMAND 消息在 wParam 消息参数的低字节里传递标识符的值。

你也可以在 Resource Symbols (资源符号) 浏览器里改变、添加或删除标识符符号, 如图 4-2 所示, 这些标识符是由 AppWizard (应用程序向导) 或其他一种 Visual C++ 资源编辑器创建的 RC 文件提供的。要打开视图, 单击 View (查看) 菜单里的 Resource Symbols (资源符号)。视图将显示所有在 Resource.h 中定义的

标识符，要给标识符重命名或改变其值，从列表里选择它并单击 Change（更改）按钮。New（新建）按钮让你在 Resource.h 里添加新的标识符，并给它们赋值。在你关闭 Resource Symbols(资源符号)对话框之后，右击 ResourceView（资源查看）窗格里的项目行，并选择 Save（保存）命令，把新的值写到 Resource.h 文件里。

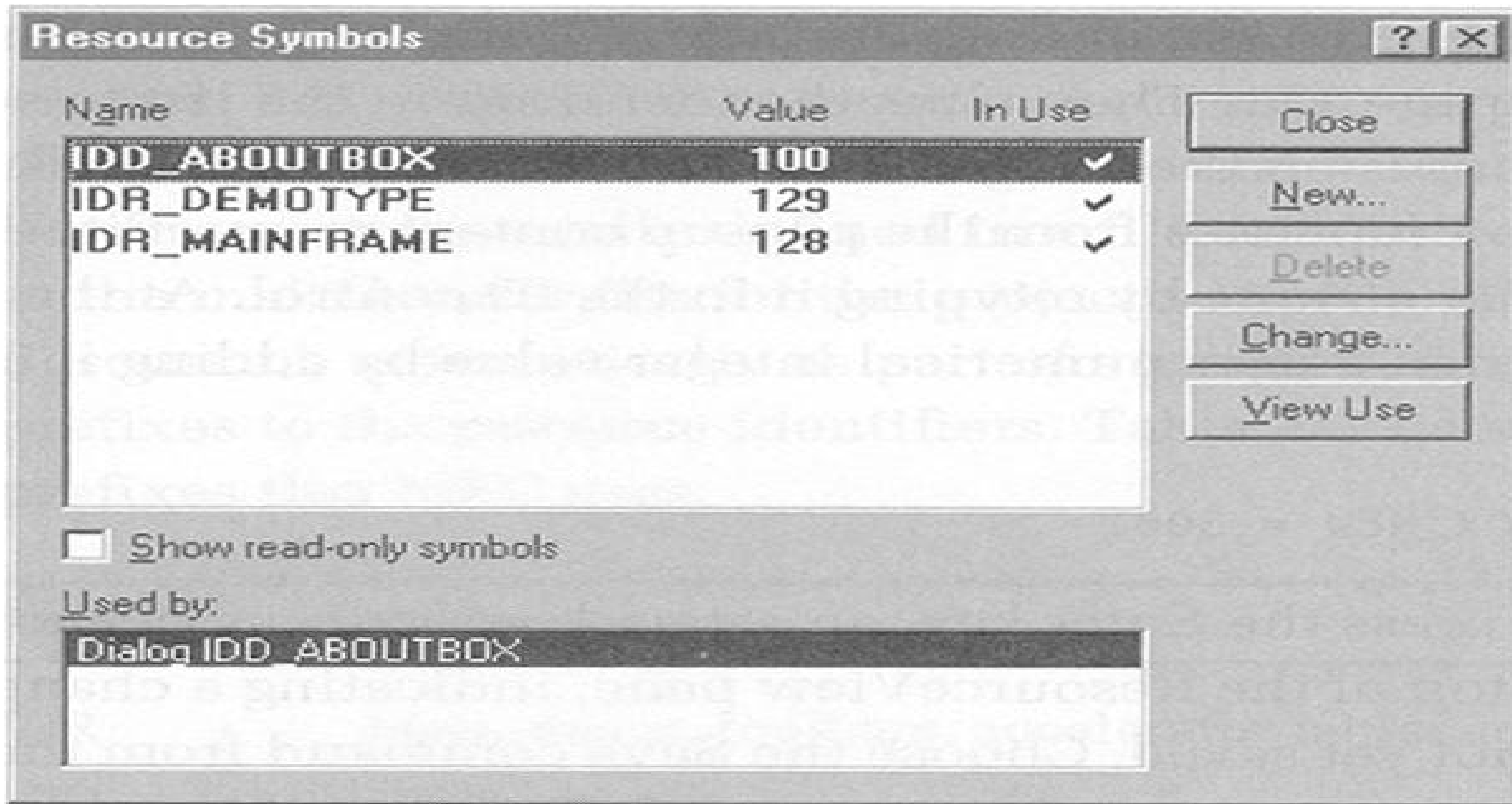


图 4-2 Resource Symbols 浏览器，它通过 View 菜单上的 Resource Symbols 命令调用

符号浏览器是为最佳利用由 AppWizard（应用程序向导）或其他一种 Visual C++ 资源编辑器创建的 RC 文件而设计的。你可以查看所有在项目 RC 文件里引用的标识符的定义，但是，浏览器只可以修改在 Resource.h 头文件里定义的标识符。它把定义在其他包含文件里的标识符当作只读的，并不提供修改它们名字和值的方法。要看这些标识符，单击浏览器对话框里的 Show Read-Only Symbols（显示只读符号）检查框。例如，AppWizard（应用程序向导）添加到 RC 文件里的这一行，我们将在本章后面再次见到它：

```
#include "afxres.h"
```

当你选择了 Show Read-Only Symbols 复选框，浏览器将包含在 Afxres.h 文件里的所有符号的列表。你可以区别出在列表里的只读符号，因为可修改的符号以黑体字显示。

在 In Use（正在使用）列里，一个复选标记表明，在 RC 文件里，一个符号标识一种资源。在你开发一个程序时，偶尔可能要改变标识符的名字，没有关系，但是 Visual C++ 不删除旧的名字，而为新的标识符在 Resource.h 中添加一个定义。一些标识符最终会成为孤标识符（orphans），它在 Resource.h 文件里定义，但是在 RC 文件里任何地方都没有使用。在 In Use 列中，我们可以很容易确定孤标识符。要删除一个标识符，就是从 Resource.h 文件中删除它的 #define 语句，那么从列表里选择它，并单击 Delete 按钮。删除将在你下一次单击 Save（保存）命令时生效。记住，虽然浏览器告诉你，只有复选的符号才会出现在 RC 文件里，

但是，这并不意味着这个符号不在源代码或其他资源文件的其他地方使用。

AppWizard资源的例子

在深入说明之前，让我们看看一部分 AppWizard（应用程序程序向导）为假想的 Demo 程序生成的资源脚本文件。我们已经见到，AppWizard（应用程序程序向导）自动的为 About（关于）对话框创建一个资源脚本，还有一个 MFC 图标。在生成的 Demo.rc 文件里，对话框脚本看起来像下面这样：

```
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

```
CAPTION "About Demo"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
        ICON                IDR_MAINFRAME,IDC_STATIC,10,22,20,20
```

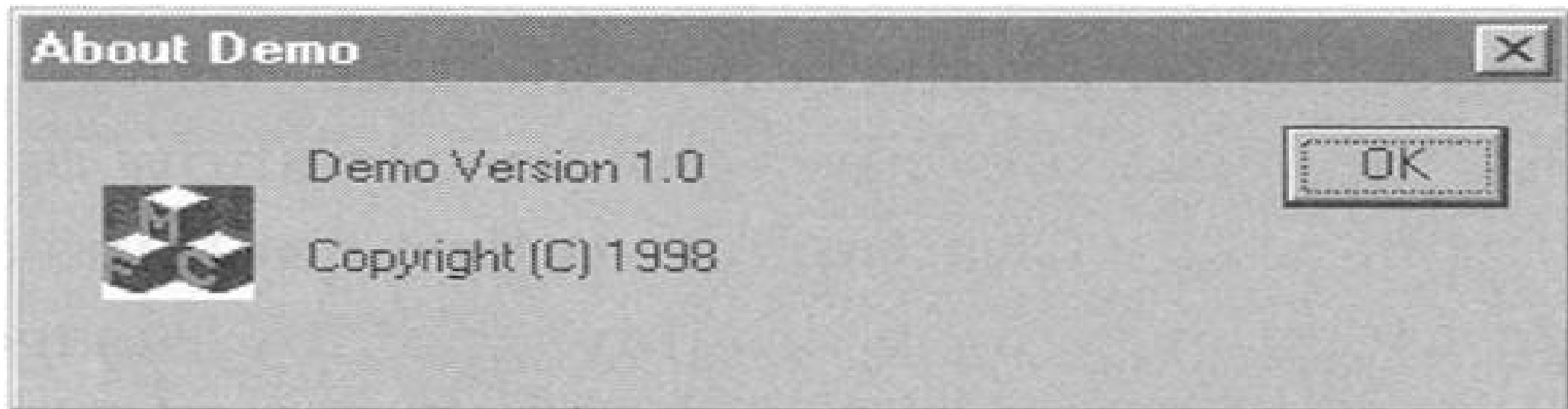
```
        LTEXT                "Demo Version 1.0",IDC_STATIC,45,10,115,8
```

```
LTEXT          "Copyright  1998 ", IDC_STATIC,40,25,119,8
```

```
DEFPUSHBUTTON  "OK",IDOK,178,7,32,14,WS_GROUP
```

```
END
```

上述命令定义了下面所示的对话框，它是通过选择 Demo 程序的 Help（帮助）菜单里的 About 来调用的：



虽然一行也没写，但是，效果还不错。然而，AppWizard（应用程序向导）并不是适合于任何场合。要看一下不使用 AppWizard（应用程序向导）将会是什么样，本章的其余部分开发了一个不使用 AppWizard（应用程序向导），而是从头开始的资源程序，并讨论这个方法的利弊。

例程 DiskPie1 简介

这里，我们将开始用连续的几节逐步开发一个叫做 DiskPie1 的例程。每一节集中一种资源类型，我们将从菜单和加速键开始，接着是状态栏、位图和工具栏。每一节都以总体讨论一种资源类型开始，以将这种资源加入 DiskPie1 结束，并

示范怎样用适当的 Visual C++ 编辑器创建或修改一种资源。当我们结束的时候，DiskPie1 将成为一个有用的工具，它快速查看当前内存使用情况和可用的磁盘空间。

为了更好地开发实践，我们将在写任何代码之前，首先对程序进行说明。这个说明将使你了解我们将加入到程序中的资源，并使我们可以很容易地看到它们怎样一起工作，形成一个和谐的界面。下面是 DiskPie1 的主要说明：

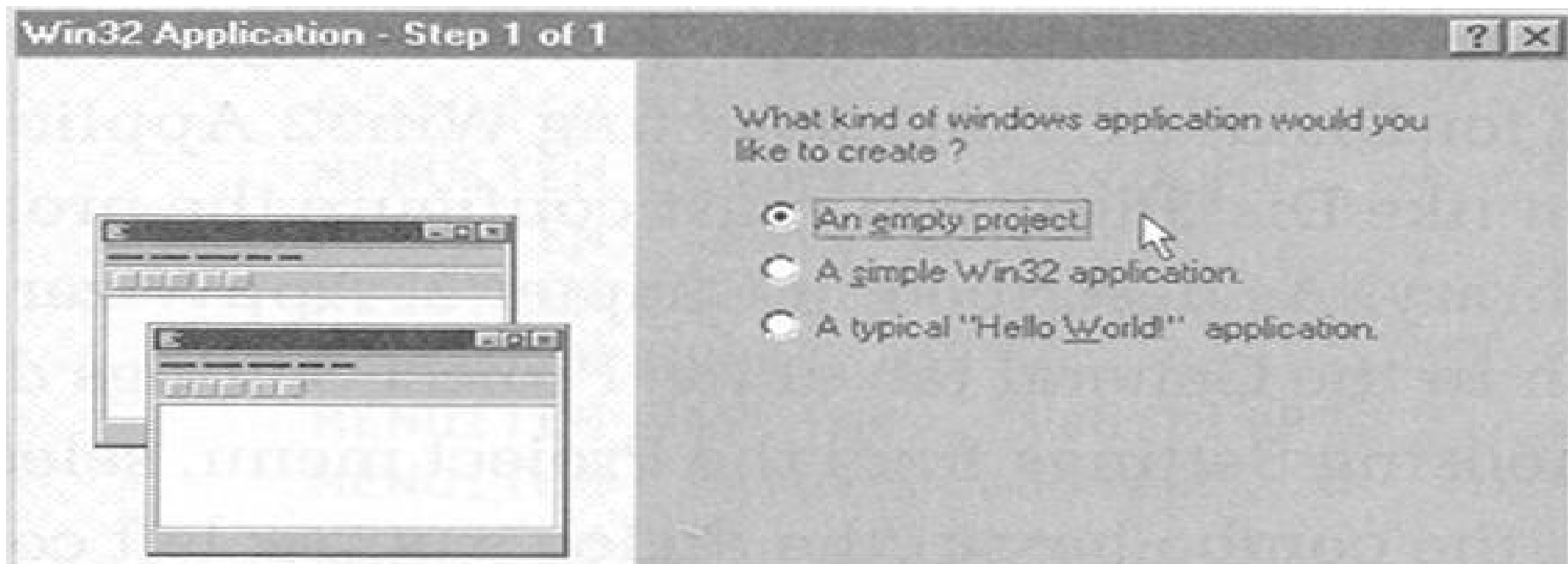
- **描述：**DiskPie1 是一个用 MFC 写的实用程序，它显示两份饼图。通过对菜单或工具栏的选择，这个图显示指定的磁盘驱动器上的存储空间分配。饼图的一部分描述已经分配的空间，而其余部分显示未分配的空间。有标签将两者明显区分开。
- **主窗口：**程序有四个菜单，叫做 File，Chart，View 和 Help。File（文件）菜单只包含一个 Exit 命令，Help（帮助）菜单有一个显示程序信息的 About 命令。View（查看）菜单允许用户显示或隐藏工具栏和状态栏。Chart 菜单开始时只包含一个命令，叫做 Memory，它显示内存使用状况。在运行的时候，DiskPie1 搜索与系统相连的磁盘驱动器，包括 RAM 磁盘和远程网络驱动器，并把它们添加到 Chart 菜单里。此程序忽略软盘驱动器、CD 驱动器及其他可卸式驱动器。
- **工具栏和加速键：**可停靠的工具栏和键盘命令补充了程序的菜单，允许用户通过单击按钮或通过按表示从 C 到 Z 的特定驱动器的键，来显

示表示使用状况的饼图。

- 状态栏：标识当前选择的菜单或工具栏。
- 上下文菜单：DiskPie1 不提供上下文菜单。

DiskPie1 可以很容易地通过 AppWizard 生成骨架代码，但是，我不选择这样做，有两个原因。首先，DiskPie1 并不是那种文档/视图形式的应用程序程序，而 AppWizard 创建文件时，注重的是这种形式，并且，删除 AppWizard 生成的多余资源描述可能是冗长乏味的过程。第二，我们已经谈到过 AppWizard。现在，应当看看在 Visual C++ 里从底层创建一个项目会怎样。下面的几节并没有忽略 AppWizard，不管怎样，在默认的情况下，它们都描述 AppWizard，这样，你能够看到，通过使用 AppWizard 创建 DiskPie1 这样的小项目将得到或失去什么。

讨论假设 DiskPie1 以一个没有资源文件的空项目开始。如果你喜欢在这里略述的步骤从头开始创建这个项目，那么从 File（文件）菜单选择 New（新建）命令，然后单击 Project（项目）选项卡，选择 Win32 Application 图标。输入项目名字，然后单击 OK（确定）按钮。Win32 Application Wizard 只显示一个单独的步骤，提供在三种不同的准备状态下，以设置一个新的项目，单击 Finish（完成）按钮，接受为空项目选择的默认设置。



如果你已经运行 Setup 程序，从所附 CD 中将 DiskPie1 拷贝到你的硬盘，那么你可能更喜欢打开项目，不用自己创建资源，而继续进行讨论。要打开已经完成的项目，从 File（文件）菜单选择 Open Workspace（打开工作空间）命令，并在硬盘驱动器上浏览 DiskPie1 项目所在的文件夹。双击 DiskPie1.dsw 打开项目。

打开已有的项目

新项目就像新文件一样，开始要使用 New 命令，并在一个 Visual C++向导的帮助下（如 AppWizard 或 Win32 Application），设置新的项目。一旦向导完成了它的工作，此项目就不再使用 New 命令了。要打开一个已有的项目，例如从配套光盘中安装的任何一个项目，需从 File 菜单中选择 Open Workspace 命令，并浏览到项目的文件夹。对于最近使用过的项目，相同菜单上的 Recent Workspace 命令提供了更好的方便性。

如果你愿意，Visual C++环境可以在启动时自动打开最近的项目。对于你已经花费了许多时间的冗长的项目，这个特征非常方便。单击 Tools（工具）菜单上的 Options（选项），并滚动到右边，找到 Workspace 选项卡，然后，激活标为 Reload Last Workspace At Startup（在启动时重新加载上次的工作空间）复选框。

配置 DiskPie1项目

Win32 Application 向导只生成几个文件，它们用于建立项目的主干内容。此向导也假设项目不使用 MF，对 DiskPie1 来说，这是不正确的假设。在选择 Win32 Application 创建像 DiskPie1 一样的 MFC 程序之后，你必须配置这个项目，来识别 MFC 库文件。正如在第 2 章里描述的那样，这是通过在 Project Settings 对话框 General 选项卡的一个开关来实现的。通过从 Project（项目）菜单选择 Setting 打开对话框，从对话框的左上角的组合框里选择 All Configurations（所有配置），

并为项目选择静态或动态链接库。图 4-3 显示后一种选择。

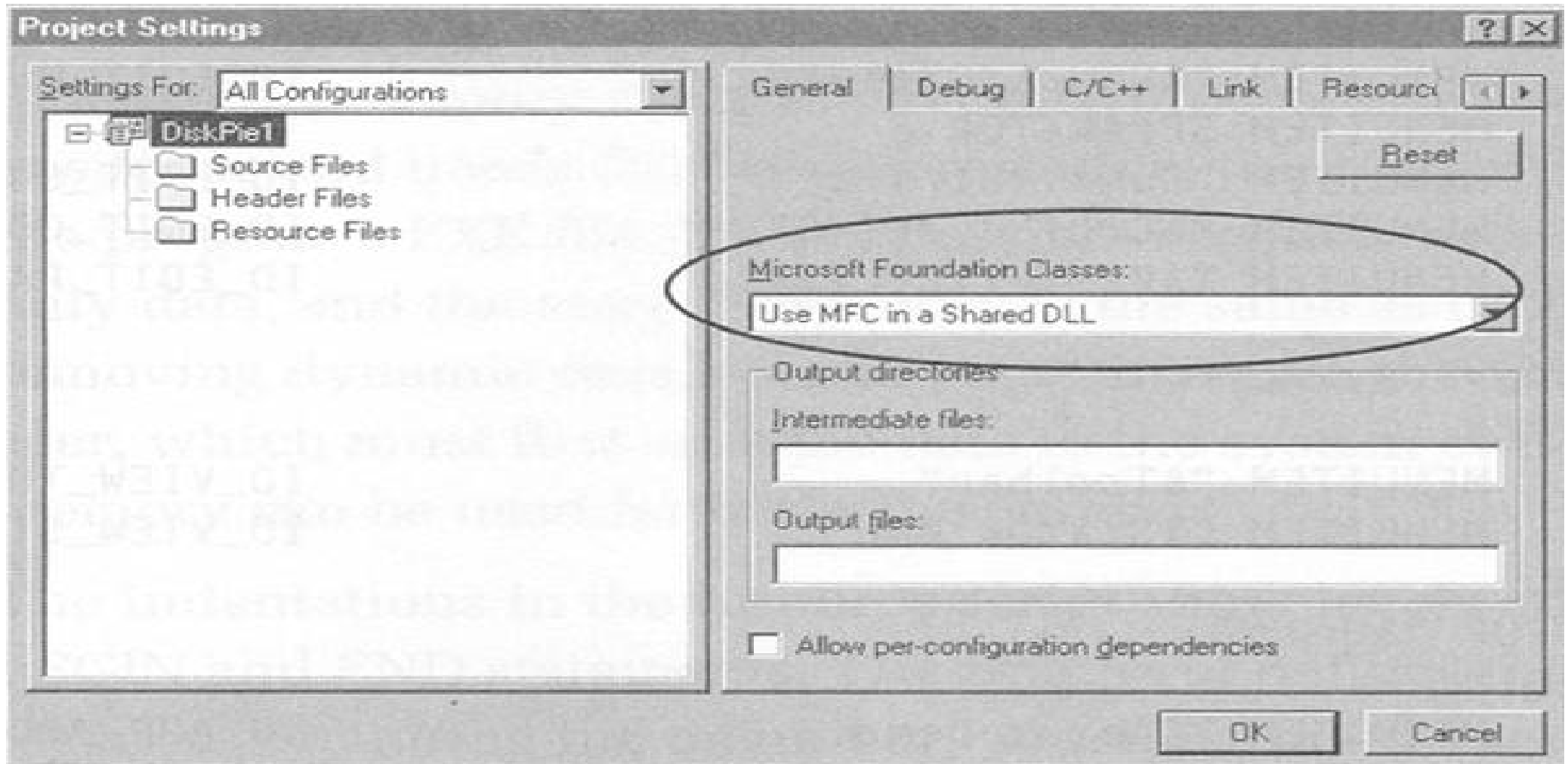


图 4-3 在 Project Settings 对话框中选择到 MFC 的动态链接

一旦打开了初具雏形的 DiskPie1 项目，并正确进行配置，我们便可以开始为它

创建资源，并添加 DiskPie1.rc 和 Resource.h 文件了。对于 DiskPie1 这样一个小程序，它使用了很多资源，所以，大部分工作是创建资源数据。在完成资源的创建之后，我们将为程序写出实际的代码。

菜单和加速键

图 4-4 显示了 AppWizard 创建的默认菜单系统。你可以看到，图中的菜单和 AppWizard 放在 RC 文件中的菜单脚本是对应的：

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
```

```
BEGIN
```

```
    POPUP "&File"
```

```
    BEGIN
```

```
        MENUITEM "&New\tCtrl+N",          ID_FILE_NEW
```

```
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
```

```
        MENUITEM "&Save\tCtrl+S",          ID_FILE_SAVE
```


BEGIN

MENUITEM "&Undo\tCtrl+Z", ID_EDIT_UNDO

MENUITEM SEPARATOR

MENUITEM "Cu&t\tCtrl+X", ID_EDIT_CUT

MENUITEM "&Copy\tCtrl+C", ID_EDIT_COPY

MENUITEM "&Paste\tCtrl+V", ID_EDIT_PASTE

END

POPUP "&View"

BEGIN

MENUITEM "&Toolbar", ID_VIEW_TOOLBAR

MENUITEM "&Status Bar", ID_VIEW_STATUS_BAR

END

```
POPUP "&Help"
```

```
BEGIN
```

```
    MENUITEM "&About Demo...",          ID_APP_ABOUT
```

```
END
```

```
END
```

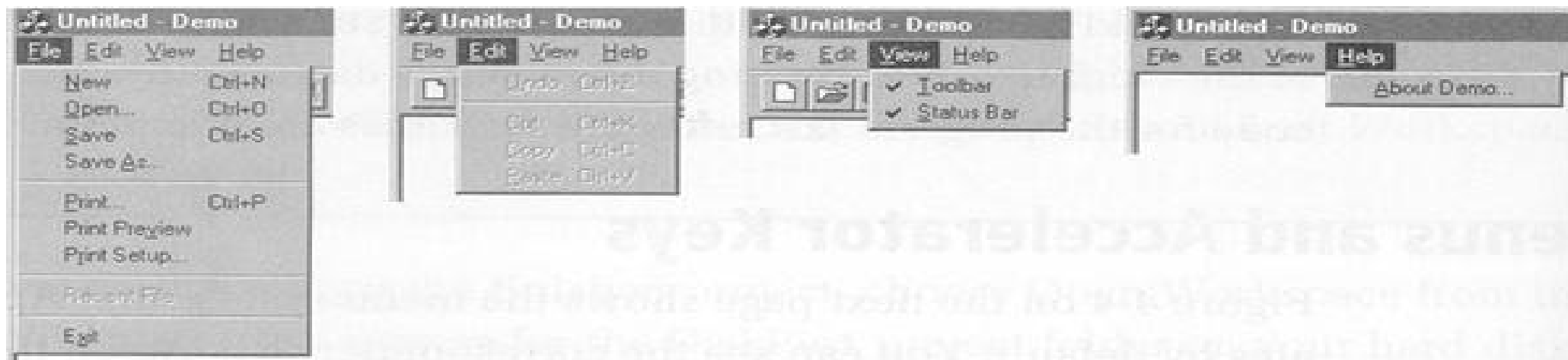



图 4-4 由 AppWizard 生成的菜单系统

脚本的第一行给菜单栏一个 `IDR_MAINFRAME` 标识号，它在 AppWizard 添加到项目的 `Resource.h` 文件里定义。像文件里的所有标识符一样，`IDR_MAINFRAME` 只是 AppWizard 的默认名字。你可以为资源指定你想要的任何名字或值。

`PRELOAD` 和 `DISCARDABLE` 伪指令在 Win32 应用程序程序脚本中并不是必须的。只对 16 位程序有意义的 `PRELOAD` 告诉 Windows，当它开始加载程序时，将菜单资源数据拷贝到内存，而不是以后当程序创建主窗口时，再打开程序的 `EXE` 文件，并读取菜单数据。不需要 `DISCARDABLE` 伪指令，是因为在 Win32 里，所有的资源都是可丢弃的。这意味着，操作系统可以自由地从物理内存中删除程序的资源数据，以使其他程序能够得到这些内存。当程序再次获得焦点，并需要那些删除的资源时，系统从程序的 `EXE` 文件中重新读取数据。这可能是因

为资源是静态只读数据，在内存里的副本和磁盘上的一样。相反，从内存中移去动态数据由虚拟内存管理器完成，但是，虚拟内存管理器必须首先在内存被用于其他目的之前，将数据保存到系统交换文件。

资源脚本文件的缩进排列显示了 BEGIN 和 END 语句之间的资源的级别。第一级定义了完整的菜单资源，包括菜单栏，它被叫做顶级菜单。BEGIN-END 对的第二级指明每一个下拉菜单的内容。每一个 POPUP 语句跟在一个显示在菜单栏上的菜单标题之后，随后的 MENUITEM 语句指明列在菜单上的命令。菜单上的一行叫做命令或菜单项。

一些菜单命令包含键盘组合键，例如 Ctrl+N 相当于 New（新建）命令，Ctrl+O 相当于 Open（打开）命令。如你所知，这些组合键提供像快捷键那样的服务，它让用户选择命令，而不用通过菜单系统。例如，Ctrl+O 立即显示 Open（打开）对话框，与从 File（文件）菜单里选择 Open（打开）命令的效果一样。加速键的麻烦是用户必须记住它们，它们出现在菜单中，是为了提醒用户，还有一种更容易的方法来选择命令。加速键要求在 RC 文件中附加的表格，我们将立即看到这些。

加速组合键之前的 \t 是一个制表符，它整齐地在菜单上排列加速键。你也可以使用 \a 代替 \t，在菜单上右对齐排列文本，以具有一致性。如果你使用 \a 来排列菜单上的任何加速组合键，那么，你应该在其他行上不使用 \t。如果不这样做，将使 Windows 混乱，导致菜单文本排列凌乱。 \a 字符比 \t 对菜单的控制更好些。如

果菜单上的文本看起来太拥挤，那么在 \a 字符前输入几个空格，可以加宽菜单，并使加速键和命令之间的间隔更远。在每一个菜单命令里的字符 & 把它后面的字符作为菜单的助记符。如图 4-4 所示，在菜单里，一个助记符以下划线显示，以便用户识别它。对于一个菜单或菜单栏来说，助记符应该是唯一的，例如，Format 菜单应该有一个不是“F”的其他助记符，以便避免同 File（文件）菜单冲突。但是，使用唯一的助记符只是一个建议，而不是一条规则，如果一个菜单栏或下拉菜单在两个或更多的地方包含相同的助记符，那么，当用户按下助记符时，Windows 将依次加亮每一个命令，并且只激活你按下 Enter 键时选择的命令。下一节描述的菜单编辑器可以通过在弹出的上下文菜单上的命令，来检查重复的助记符。在编辑器工作区域的任何地方右击，将打开这个菜单：



助记符和加速键并不是相同的东西。加速键不需要通过菜单系统，就可以激活一个命令，然而，下划线助记符只有当菜单可见时才是可用的。

每一个菜单命令都有一个以 ID_ 作为前缀，紧接着是描述命令的名字组成的辅助

标识符。包括 ID_前缀的标识符是完全由用户决定，前面的菜单脚本只是 AppWizard 提出的（然而，正如我们将看到的，使用 MFC 已经定义的特定符号名是有好处的）。只有通过菜单标识符，程序才能引用菜单事件。当用户单击一个菜单命令，或按下一个加速键时，Windows 用在 wParam 低字节里的命令标识符给主窗口发出 WM_COMMAND 消息。如果这个命令是用户按下加速键响应的，那么 wParam 的高字节是 TRUE，如果是通过菜单选择响应的，那么 wParam 的高字节是 FALSE。

按传统，C 程序通过一系列 switch-case 语句，来检查 WM_COMMAND 消息的 wParam 参数以传递菜单命令：

```
switch (msg)
{
    case WM_COMMAND:
        switch (LOWORD (wParam))
        {
            case ID_FILE_NEW
                OnFileNew ();
```

```
        break;

    case ID_FILE_OPEN:

        OnFileOpen ();

        break;

    .

    .

    .

}

}
```

MFC 程序通过一个消息映射来完成相同的事情：

```
BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)

    ON_COMMAND(ID_FILE_NEW, OnFileNew)

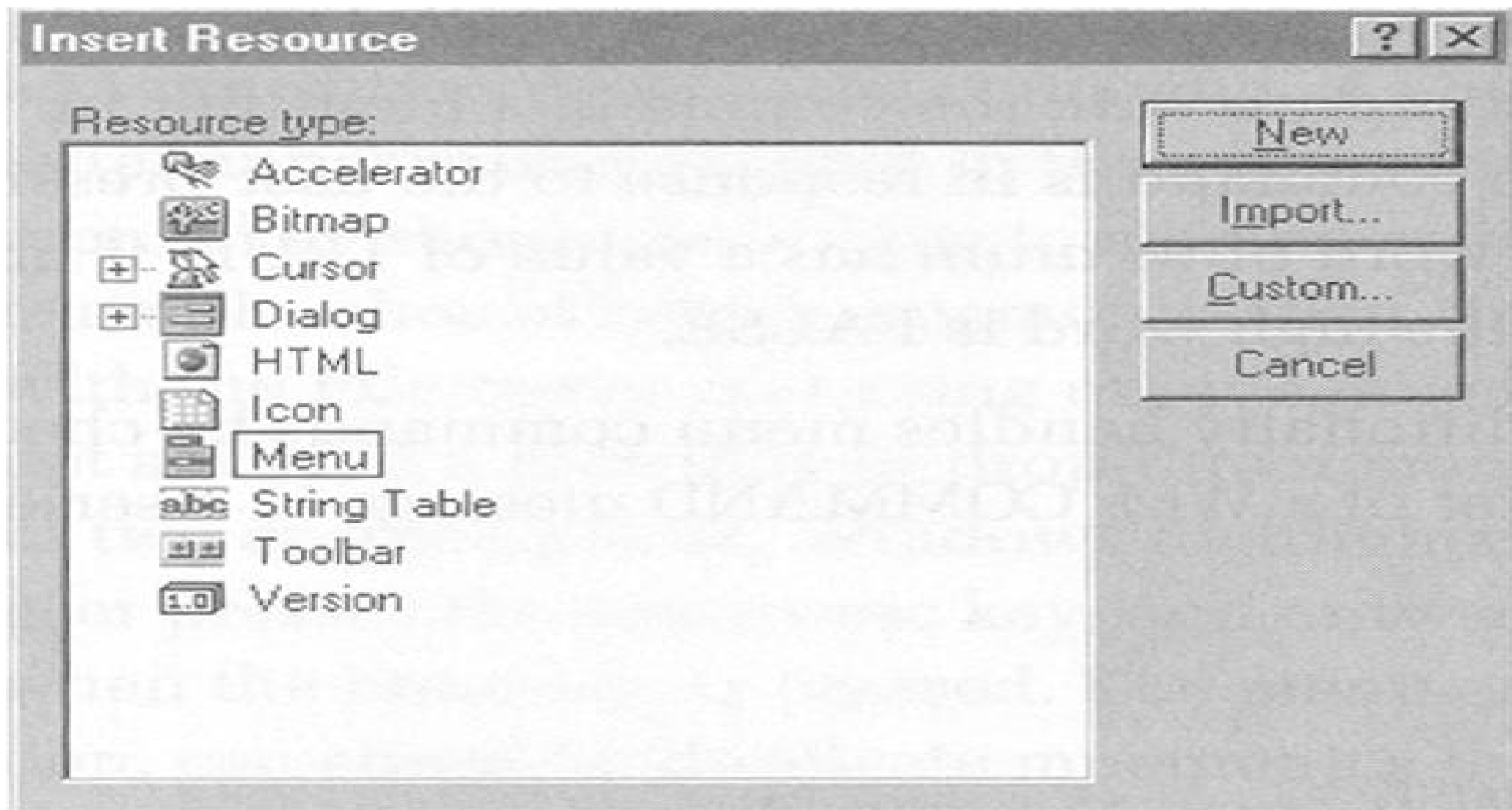
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
```

```
.  
.   
.
```

```
END_MESSAGE_MAP ()
```

为 DiskPie1 创建菜单

正如我们将为 DiskPie1 所做的那样，当我们从头开始创建菜单资源时，我们从 Insert 菜单选择 Resource，以显示资源类型列表，如下所示。然后，双击列表里的 Menu 条目打开菜单编辑器。



项目必须是打开的，你可能不得不隐藏 Workspace 或 Output（输出）窗口，以便不覆盖编辑器的工作区域。当你设计并保存菜单时，Visual C++把菜单脚本写到

项目的 RC 文件里，并把标识符 `#define` 语句写到 `Resource.h` 文件里。其后，当你打开菜单资源时，环境自动打开菜单编辑器。为了显示 Project（项目）菜单标识符的列表，双击 `ResourceView`（资源查看）窗格里的 Menu 条目（如图 4-1 所示）。通过双击列表里的资源标识符或右击标识符，并从上下文菜单里选择 `Open` 打开菜单编辑器。

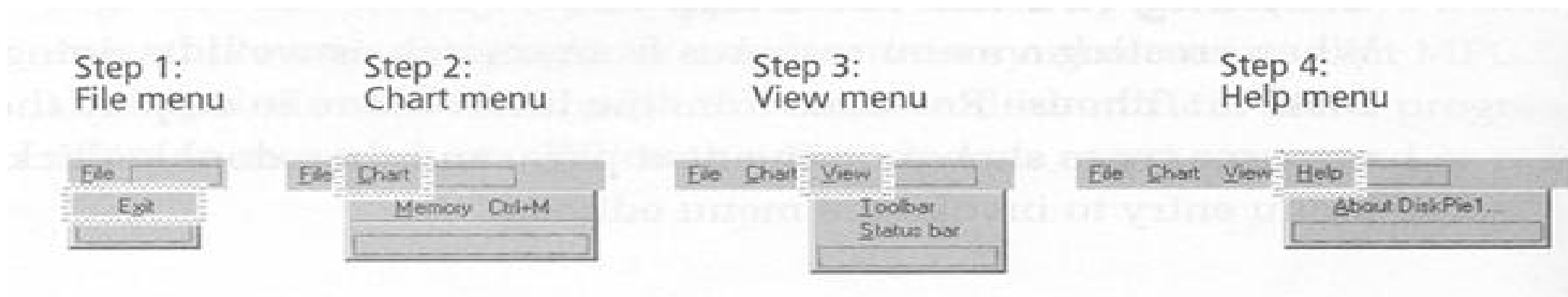


图 4-5 使用 Visual C++ 菜单编辑器创建 DiskPie1 的菜单

图 4-5 显示了当我们向 DiskPie1 项目进一步添加菜单时菜单编辑器的外观。顶级菜单（也就是菜单栏）包含一个点线矩形，叫做新项框，它指出菜单标题文本插入点。当你在菜单栏上输入一个条目，并按下 `Enter` 键，一个带有自己的新项框的下拉菜单出现了。模糊的边界表明新项框是激活的，而不管它是菜单栏上，还是在下拉菜单里。如果你想在没有击活的新项框里输入一项，那么，首先单击新项框来选择它。任何输入到新项框的东西将同时输入到图 4-6 所示的 `Menu Item Properties`（菜单项属性）对话框的 `Caption` 控件里。要返回并修改一个标题或菜

单项，或者选择这个项目，并输入新的文本，或者双击这个项目打开 Menu Item Properties（菜单项属性）对话框。当你正在菜单项之间跳转时，对话框会消失，有时，这是很不方便的。在这种情况下，单击对话框左上角带金属头的钢针按钮，可强制对话框保持可见。

DiskPie1 的 File（文件）菜单只有一个命令，叫做 Exit。要创建这个菜单，首先在菜单栏的新项框里输入 &File，按下 Enter 键，然后输入 E&xit 作为菜单项的文本。如果此时按下 Enter 键，菜单编辑器自动给这个命令一个叫做 ID_FILE_EXIT 的标识符，它是菜单标题和菜单项文本的混和物。它也为 ID_FILE_EXIT 添加 #define 语句到 Resource.h 文件中。

让我们停一会儿，想一下为什么这可能会导致以后的问题。当你保存这个新的菜单资源时，Visual C++ 看到这个项目没有 RC 文件，它自动为你创建一个。它也添加这些行到 RC 文件：

```
#include "afxres.h"
```

```
#include "resource.h"
```

MFC 提供了 Afxres.h 头文件，使你不必为每一个项目定义相同的公共标识符，它们显示在典型的 Windows 程序里。理论上，大多数 Windows 程序都有一个 File，Edit，View 和 Help（帮助）菜单，Afxres.h 定义了一些常用的标识符，例如，ID_FILE_OPEN，ID_EDIT_COPY 和 ID_APP_ABOUT 等。它为你定义自己

的资源标识符保留了 Resource.h。偶然情况下，Afxres.h 没有为 ID_FILE_EXIT 定义，但是，如果它定义了会怎样呢？在那种情况下，当编译 RC 文件时，你将碰到一个错误，因为 ID_FILE_EXIT 将定义两次，一次是在 Afxres.h 里，一次是在 Resource.h 里。

AppWizard 生成的资源脚本并不存在这个潜在的命名冲突问题。AppWizard 生成的所有的菜单项都在 Afxres.h 里定义，所以，AppWizard 并不为它们添加定义到 Resource.h 里。对于 DiskPie1 这样的非 AppWizard 项目，当使用资源编辑器时，你有三种选择来防止定义重复：

- 在文本编辑器里打开 RC 文件，并为 Afxres.h 移去 #define 语句。
- 不接受资源编辑器可能在 Afxres.h 文件里的默认标识符，用你自己定义的资源标识符。
- 编辑 Resource.h 文件，并删除任何已经在 Afxres.h 里定义的标识符。

第一种操作的问题是，它迫使你也从文件里移去了所有其他 Visual C++ 要求在 Afxres.h 里定义的跟踪信息。第二种解决方案更可靠。当命名资源标识符时，由于 IDM_ 在 Afxres.h 中从没有作为一种标识符前缀使用，MFC Technical Note 20 推荐你添加 IDM_ 到菜单标识符上。在 Menu Item Properties（菜单项属性）对话框里指定标识符名字，同时像下面那样随意为标识符设一个值：

```
IDM_FILE_EXIT=1001
```

当然，你要确定每一个菜单标识符有一个唯一的值。

采用上面列表选项中的第三种方案更好些，尽管这种方法有些粗野。考虑一下，如果你在程序里使用像 `IDM_FILE_EXIT` 这样的名字来标识 `Exit` 命令，将会发生什么。对于像 `DiskPie1` 这样的 MFC 应用程序，你必须接着为带有标识符的 `WM_COMMAND` 消息提供一个处理函数，同时也添加一个指向处理函数的消息映射。结果可能看起来像下面这样：

```
ON_COMMAND (IDM_FILE_EXIT,OnFileExit)           // In the message map

.
.
.

void  CMainFrame::OnFileExit()                    //Handler  for
IDM_FILE_EXIT

{

    SendMessage( WM_CLOSE, 0, 0);
```

```
}
```

Afxres.h 包含几个特定的标识符名字，MFC 为它提供自己的处理函数，这样做可免除应用程序的麻烦。这些特定标识符中的一个 ID_APP_EXIT，它通过一个关闭应用程序的 MFC 函数来自动调用。通过指定 ID_APP_EXIT 的值给 Exit 菜单命令，DiskPie1 不需要提供它自己的代码来处理 Exit 菜单选项。出于类似的原因，DiskPie1 的 View（查看）菜单上的两个菜单项的值被指定为 ID_VIEW_TOOLBAR 和 ID_VIEW_STATUS_BAR。MFC 认可这些特定值，并调用它自己的处理函数来显示或隐藏工具栏和状态栏。DiskPie1 在它自己的菜单脚本里为 Toolbar 和 Status Bar 命令简单地使用这些标识符，而 MFC 框架则处理其他每一件事情。

给命令特定的标识符（如 ID_APP_EXIT 或 ID_VIEW_TOOLBAR）的缺点是菜单编辑器在 Resource.h 里为这些名字书写定义，这样就重复了在 Afxres.h 里的定义。我们不得不在创建资源之后使用文本编辑器，来删除在 Resource.h 里的无用定义。

DiskPie1 菜单项标识符的名字是通过把它们输入到 Menu Item Properties（菜单项属性）对话框里来指定的。下面是结果概况：

菜单标题	项目标题	标识符
&File	E&xit	ID_APP_EXIT
&Chart	&Memory\tCtrl+M	IDM_MEMORY
&View	&Toolbar	ID_VIEW_TOOLBAR
&View	&Status bar	ID_VIEW_STATUS_BAR
&Help	&About DiskPie1...	ID_APP_ABOUT

在图 4-6 所示的 Menu Item Properties（菜单项属性）对话框中，可以优化菜单的外观。例如，如果程序刚开始时一个菜单命令是非激活的，那么，这个菜单的文本应该变灰，来提示用户这个命令当前是不可用的。通过单击对话框里的 Grayed（变灰）复选框，可以为一个菜单项指定变灰文本。要在菜单命令旁放置复选标记，单击 Checked（复选的）。在资源脚本里设定变灰文本或复选标记，对于 DiskPie1 这样的 MFC 程序是不需要的，因为框架会自动更正菜单。

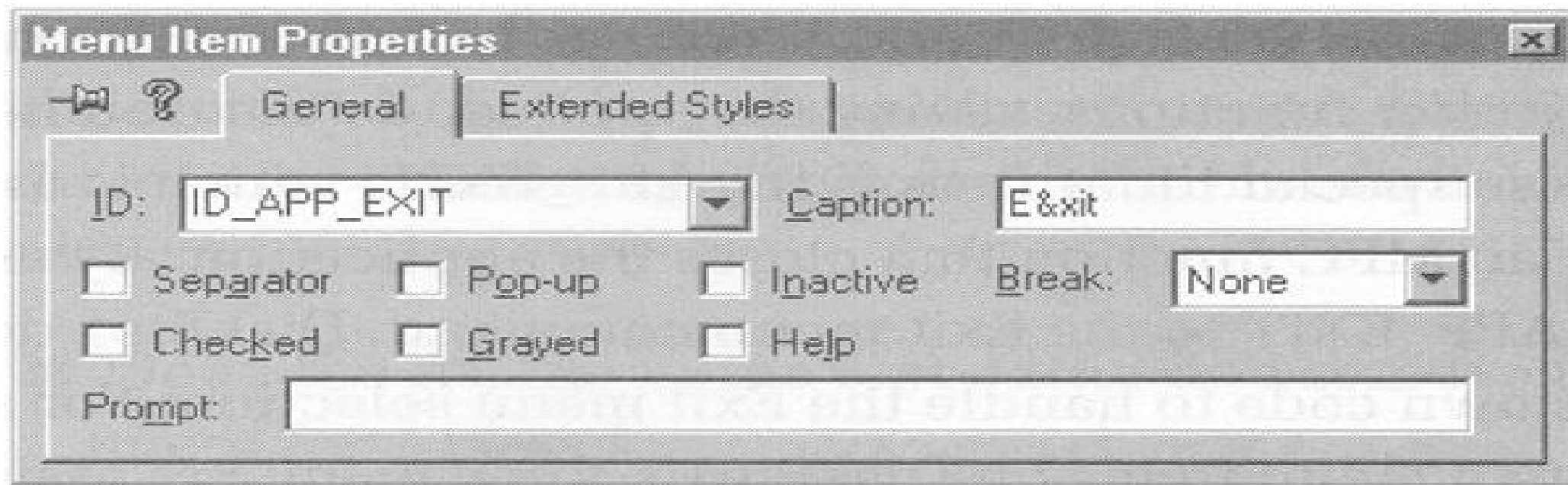


图 4-6 菜单资源的 Menu Item Properties 对话框

如果你想用菜单命令打开一个折叠的弹出菜单，单击在 Menu Item Properties（菜单项属性）对话框里的 Pop-up（弹出）复选框。显示在菜单项旁边的箭头符号告诉用户，这个命令显示一个嵌套的弹出菜单。编辑器为这个弹出菜单显示另一个新项框，你在里面输入命令，就像在其他任何菜单里一样（在环境的 File（文件）菜单上的 Recent File 命令给出一个折叠弹出菜单的例子）。

对于像 DiskPie1 这样有状态栏的程序，当用户加亮菜单上的命令时，对话框里的 Prompt（提示）文本框提供了一个方便的地方，来输入显示在状态栏里的描

述。我们将在以后的一节里，使用 Visual C++ 字符串编辑器添加 DiskPie1 的菜单描述。当你看到重复的描述时，你将会同意，使用字符串编辑器是最好的选择。

DiskPie1 的菜单是标准的。唯一有趣的添加项是 Chart 菜单底部的分隔条。在一个菜单最后放置一个分隔条，乍看起来是很古怪的。但是，DiskPie1 在执行时向 Chart 菜单添加更多的命令：顶部的 Memory 命令和底部的 Disk 命令，像 Disk C 和 Disk D。要在菜单上创建一个分隔条，需单击图 4-6 里的 Separator（分隔条）复选框。

如果你想插入一个新的菜单或菜单命令，将新项框拖到想要的地方。当你拖拉这个方框时，一个水平或垂直插入线显示在光标附近。放开鼠标按钮来放下新项框，然后一般是输入新菜单的标题。你也可以拖动独立的菜单项或菜单栏目，来改变它显示的顺序。例如，要改变 Chart 和 View（查看）菜单的顺序，拖动 View（查看）菜单到左边，直到你看到一个显示在 File 和 View 之间的垂直插入线。释放鼠标，就完成了操作。

如前所述创建一个新菜单资源时，菜单编辑器想给这个新资源起一个像 IDR_MENU1 这样的名字。资源的符号名字显示在 RC 文件里菜单脚本的第一行：

```
IDR_MENU1    MENU PRELOAD DISCARDABLE
```

菜单应当有一个像 IDR_MENU1 这样的名字，但是，对 DiskPie1 这样的 MFC 程

序来说，它不是很好的选择。作为一个单文档界面程序(SDI)，DiskPie1 可以为它的资源注册模板，使用到 CSingleDocTemplate 构造器的单独调用，提供具有相同标识符值的所有资源。标识符的值是什么，甚至给资源不同的标识符名字，这些并不重要，只要菜单、工具栏、加速键表 and 状态栏资源是通过同一不变的数值描述的就行。如果你的程序不调用 CSingleDocTemplate 或同 MDI 等效的 CMultiDocTemplate，那么，你不需要担心像菜单和加速键那样用同样的符号值来标识资源。

在默认状态下，资源编辑器给所有的主窗口资源标识符提供不同的名字和值，所以，Resource.h 文件可能最终是这样：

```
#define IDR_MENU1 革命者          101

#define IDR_ACCELERATOR1          102

#define IDR_ICON1                 103

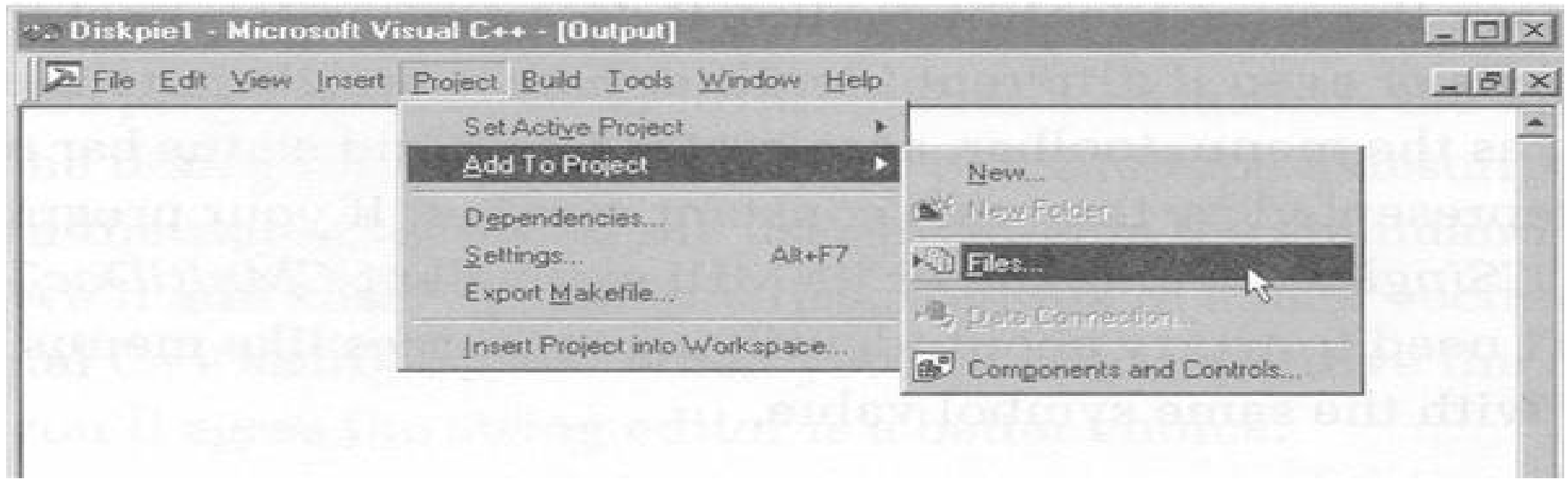
#define IDR_TOOLBAR1              104
```

如果创建资源时，你接受默认的名字，那么必须接着编辑 Resource.h 文件，在使用 CSingleDocTemplate 前给标识符一个通用值。对于 DiskPie1，我们将不接受默认的名字，而改为给菜单和其他资源指定相同的普通符号标识符，就是被 AppWizard 使用的 IDR_MAINFRAME。这将保证 CSingleDocTemplate 总是得到一

个对所有的资源来说公共单一的值。要改变菜单的标识符，双击菜单栏的任何地方，但必须是在一个菜单名字上，来打开 Menu Properties 对话框，输入 IDR_MAINFRAME，并按下 Enter 键。

这时，项目的 RC 文件还没有存在。要保存项目的第一个资源，单击 File（文件）菜单上的 Save 或 Save As，并给文件起一个同项目相同的名字，在此是 DiskPie1。Visual C++ 就创建 DiskPie1.rc 文件，将菜单资源脚本写到它里面，并创建 Resource.h 文件，以保存新的定义。不要忘记以后用文本编辑器编辑 Resource.h 文件，为 ID_APP_EXIT，ID_VIEW_TOOLBAR，ID_VIEW_STATUS_BAR 和 ID_APP_ABOUT 删除那些不需要的定义。

下一步，是将 DiskPie1.rc 文件添加到项目中。从 Properties 菜单中选择 Add To Project 命令，然后单击显示在下面的级联菜单中的 Files：



双击在文件列表里的新的 DiskPie1.rc 文件，将它添加到项目中。对 Resource.h 文件不需要进行相同的操作，因为 Visual C++ 自动识别这个头文件作为项目的支持文件。从这点来看，DiskPie1 是一个实际的项目。接着，我们将为 DiskPie1 创建一个资源，由于 DiskPie1.rc 文件已经存在，我们将使用 Save（保存）命令来保存资源，而不是用 Save As（另存为）命令。

你可能使用文本编辑器来查看 Visual C++ 写到 DiskPie1.rc 文件的菜单脚本。通过单击 File（文件）菜单上的 Open（打开）命令，将 RC 文件作为文本文档调入，来显示 Open（打开）对话框。从对话框底部的 Open As（打开类型）组合框里选择 Text，然后，双击文件列表里的 DiskPie1.rc。这里是显示在文件里的新菜单

脚本：

```
IDR_MAINFRAME MENU DISCARDABLE
```

```
BEGIN
```

```
    POPUP "&File"
```

```
    BEGIN
```

```
        MENUITEM "E&xit",                                ID_APP_EXIT
```

```
    END
```

```
    POPUP "&Chart"
```

```
    BEGIN
```

```
        MENUITEM "Memory\tCtrl+M",                        IDM_MEMORY
```

```
        MENUITEM SEPARATOR
```

```
    END
```

```
POPUP "&View"

BEGIN

    MENUITEM "&Toolbar",                ID_VIEW_TOOLBAR

    MENUITEM "&Status Bar",            ID_VIEW_STATUS_BAR

END

POPUP "&Help"

BEGIN

    MENUITEM "&About DiskPie1...",      ID_APP_ABOUT

END

END
```

图 4-7 显示了 DiskPie1 已经完成的菜单外观。Chart 菜单的 Disk C 和 Disk D 命令不会显示在上面的菜单脚本中，因为这些命令是在运行时添加到菜单上的。标题栏上的图标在本章后面创建。

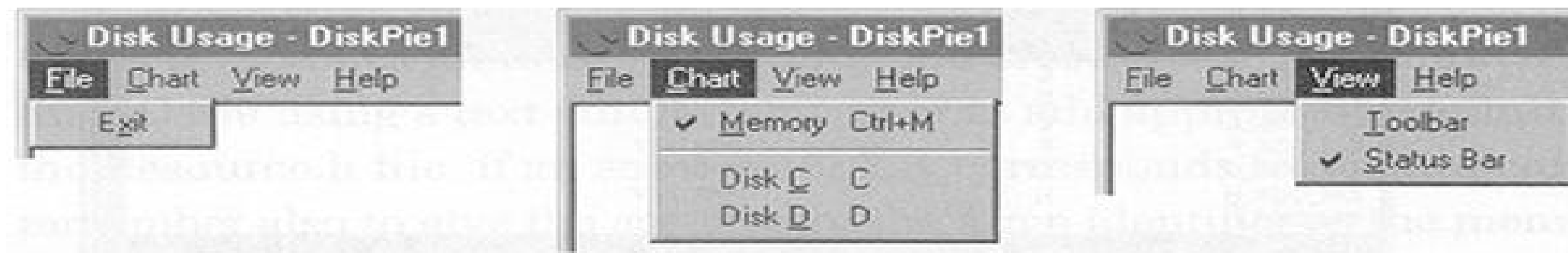


图 4-7 DiskPie1 的菜单系统

如果你想在继续下一节之前从屏幕上消除菜单编辑器，那么从 File（文件）菜单或从 Windows 菜单上选择 Close（关闭）命令。在使用这个命令时，要确定编辑器具有输入焦点。

为 DiskPie1 创建加速键

在 Visual C++ 将来的版本中，可能可以在菜单脚本里搜索加速组合键，并自动生成一个相应的加速键表。但是现在，我们将不得不手工做这些事。由于上面的脚本只有一个加速键，对应于 Memory 命令的 Ctrl+M，你可能以为 DiskPie1.rc 文件里的加速键表很小，并且很简单。但是，实际上这个表是相当大的，因为我们不得不为那些还没有在菜单上但可能在运行时被添加的项添加按键。我们将使用 Visual C++ 编辑器做这些。

对于已经存在的加速键表，例如一个通过 AppWizard 创建的加速键表，从项目的 ResourceView（资源查看）窗格里打开加速键编辑器，就如同你打开其他任何资源编辑器一样。双击在 Accelerator 文件夹里的条目来启动这个编辑器。要为 DiskPie1 这样的项目从头开始创建一个新表，通过从 Insert 菜单里选择 Resource，并双击在 Resource Type 列表里的 Accelerator 打开加速键编辑器。

图 4-8 显示了 DiskPie1 加速键列表的一部分，它包括对应于 Memory 命令的 Ctrl+M，以及从 C 到 Z 的 24 个键。这些字母键表示磁盘驱动器，作为 DiskPie1 在运行时添加到 Chart 菜单的 Disk 命令的加速键。因为磁盘驱动器（包括远程驱动器）可以指定最多到 Z 的任何字母，所以，DiskPie1.rc 必须在加速键表里包括所有可能的加速键。当程序运行时，这不会造成任何麻烦，因为 DiskPie1 忽略不与已有的驱动器相对应的按键。

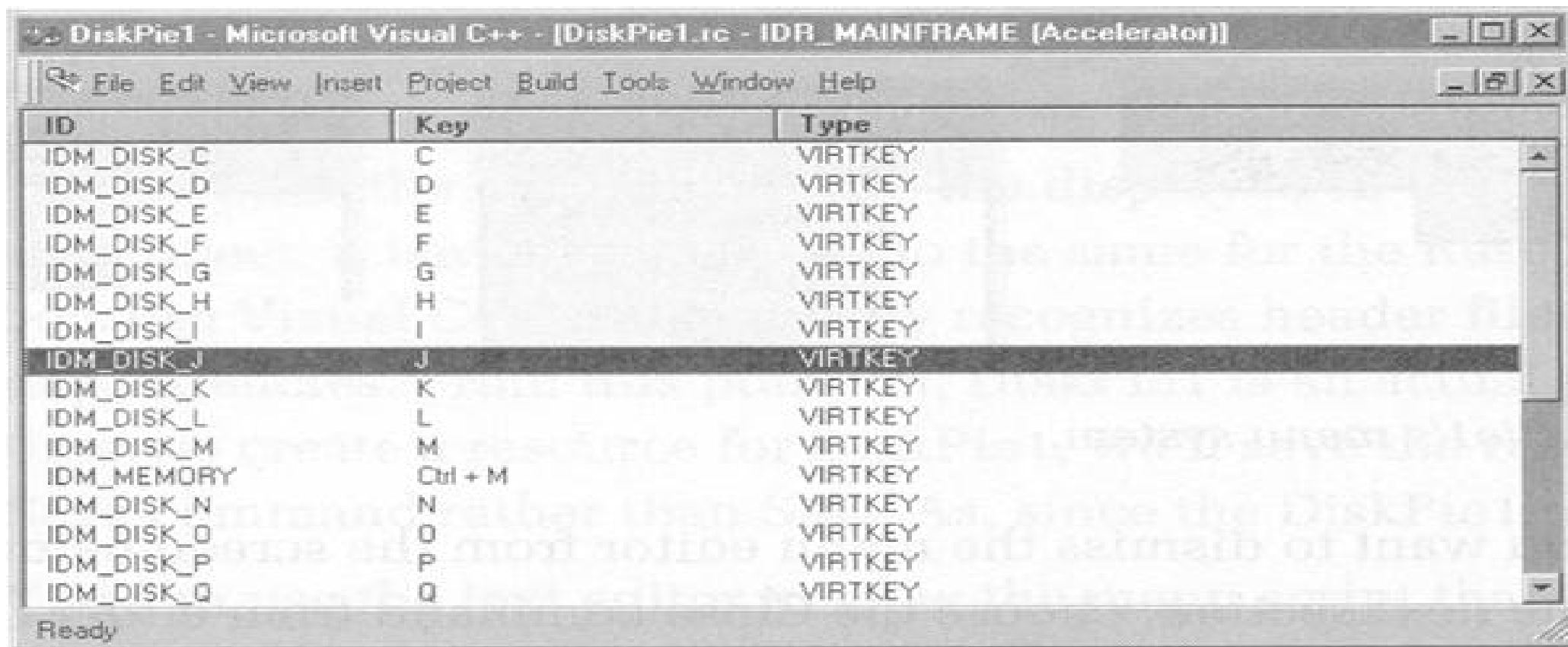


图 4-8 Visual C++ 加速器编辑器创建 DiskPie1 的加速器表

要添加一个加速键到表中，双击新项框(它以一个点框显示)，来打开 Accel Properties 对话框，然后输入加速键和它的标识符。例如，在 Key 控件里和在 ID 控件里的 IDM_MEMORY 中输入 M, 为 DiskPie1 的 Memory 命令添加 Ctrl+M 加速键。在菜单编辑器里给 Ctrl+M 加速键指定同 Memory 命令相同的标识符，我

们保证在 DiskPie1 里按下 Ctrl+M 和从 Chart 菜单里选择 Memory 命令具有相同的效果。在任何一种情况下，会调用相同的过程，以显示表示内存使用状况的饼图。

对应于 DiskPie1 的 Disk 命令的加速键都有像 IDM_DISK_C、IDM_DISK_D 等的标识符。注意，在图 4-8 里，这些加速键没有一个是同其他的键（像 Ctrl 或 Shift）进行组合，这样就允许用户仅仅按下一个字母键，例如 C 或 D，来显示 C 或 D 磁盘的使用状况饼图。要为一个加速键设置或移去一个组合键，在 Accel Properties 对话框里的 Modifiers 域选定或不选定 Ctrl, Alt 或 Shift 复选框。

在加速键编辑器里，从表里移去加速键是很容易的：仅仅选择表里的条目，然后按下 Delete 键。要选择一批条目，单击这些条目中的第一个条目，然后按下 Shift 键，并单击最后一个条目。在表中添加名称是比较麻烦的工作，特别是如果要添加一大批键的时候。对于有连续名字的符号标识符，例如那些在 DiskPie1 表中的，使用兼容宏的文本编辑器有时会更方便。如果你使用一个文本编辑器在 RC 文件里创建加速键表，那么，记住在 Resource.h 文件里添加适当的定义。如果一个加速键对应于一个菜单命令，要记住，也需要给加速键同菜单项相同的标识符。

当在加速键编辑器里添加一组具有相近名字（例如 IDM_DISK_C 直到 IDM_DISK_Z）的加速键时，可以利用剪贴板。完整地输入第一个条目，从列表里选择它，然后按下 Ctrl+C，把它拷贝到 Windows 剪贴板，然后重复按 Ctrl+V，

粘贴一系列的副本到加速键编辑器里。接着，双击第一个条目打开 Accel Properties 对话框，然后单击图钉按钮，以便对话框保留在屏幕上。你可以下移此列表选择条目，并按需要修改标识符名字和键。

单击 File 菜单上的 Save 来保存新的加速键表。编辑器给这个资源提供标识符，如 IDR_ACCELERATOR1，你可以从 Workspace(工作空间)窗口的 ResourceView (资源查看) 窗格中看到。对于 DiskPie1 来说，这不是令人满意的名字，出于相同的原因，IDR_MAINFRAME1 对菜单资源来说也不令人满意。右击 ResourceView (资源查看) 窗格里的标识符，并单击弹出的上下文菜单里的 Properties，然后将资源的符号名字改为 IDR_MAINFRAME，如图 4-9 所示。它与以前给菜单资源的名字相同。

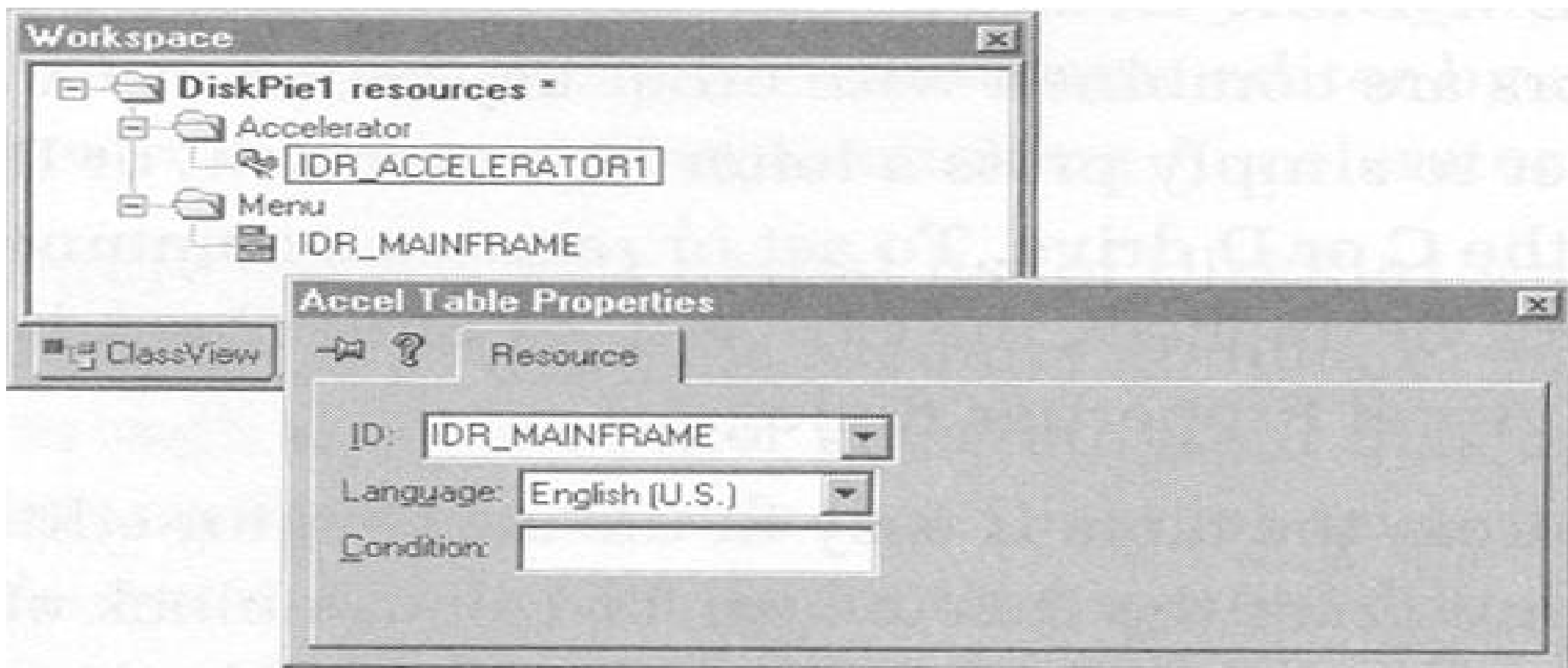


图 4-9 为加速键表更改标识符名

再次选择 Save(保存)命令,来为加速键表设置新的 IDR_MAINFRAME 标识符。在更新的 DiskPie1.rc 文件里,如下代码段显示了加速键表目前的样子:

```
IDR_MAINFRAME ACCELERATORS DISCARDABLE
```

BEGIN

"C",	IDM_DISK_C,	VIRTKEY, NOINVERT

"D",	IDM_DISK_D,	VIRTKEY, NOINVERT

"E",	IDM_DISK_E,	VIRTKEY, NOINVERT

.

.

.

."X",	IDM_DISK_X,	VIRTKEY, NOINVERT

续表

"Y",	IDM_DISK_Y,	VIRTKEY, NOINVERT
"Z",	IDM_DISK_Z,	VIRTKEY, NOINVERT

END

DiskPie1 的 Resource.h 文件包含了与这些标识符相应的定义：

#define IDM_MEMORY	130
#define IDM_DISK_C	131
#define IDM_DISK_D	132
#define IDM_DISK_E	133

.

.

```
#define IDM_DISK_X 152  
  
#define IDM_DISK_Y 153  
  
#define IDM_DISK_Z 154
```

IDM_DISK 标识符的实际值是什么无所谓，但是，有两个理由应当保持这些值连续。第一，从 IDM_DISK_C 到 IDM_DISK_Z 之间使用连续值，便可以在 DiskPie1 里用单一的过程处理所有的菜单命令，或者，通过使用 MFC 的 ON_COMMAND_RANGE 宏，来处理从 C 到 Z 之间的加速键。当我们开始添加代码到 DiskPie1 中时，我们略去了细节。第二个使用连续标识符值的理由是，它与 Windows 装载包含在一个程序资源数据里的字符串有关。那是下一节的内容。

字符串资源和状态栏

当用户加亮一个菜单上的命令时，DiskPie1 显示了在窗口左下角状态栏里的命令描述。当鼠标滑过工具栏上的按钮时，将发生同样的事情。这些描述是字符串资源那样的程序数据，它们是以文本形式保存在可执行程序的资源部分中。一个 Windows 程序可以保存任何种类的只读文本作为字符串资源，状态栏的描述文本

只是一个例子。

这一节以字符串资源的一般性讨论开始，接着，缩小它的范围，来看看状态栏描述文本。最后，介绍以使用 Visual C++ 字符串编辑器写 DiskPie1 的状态栏描述文本。

字符串资源

字符串资源同程序数据里其他字符串没有什么不同，只是它必须从可执行的文件读到缓冲区中。C 程序通过调用 LoadString API 函数读取字符串资源，MFC 程序可以调用 CString 类的 LoadString 成员函数读取。并且，至少在状态栏描述文本的情况下，MFC 程序甚至还不必做这些。MFC 提供了默认代码，可以自动加载描述字符串，我们在后面将看到这点。

字符串资源是在程序 RC 文件里的字符串表里定义的，它是用 STRINGTABLE 键标识的字符串列表，并用 BEGIN-END 语句或大括号括在一起：

```
STRINGTABLE
{
    ID_STRING1    “Text for string resource #1”
```

```
ID_STRING2    "Text for string resource #2"
```

```
}
```

在 Win32 里，字符串资源被限制在 4096 个字符之内，并在资源脚本里只能占一个独立的行。一个 RC 文件可以有多个字符串表，每一个表有任何多个字符串。

与一般的数据字符串相比，字符串资源具有两种主要的优点。第一，字符串资源只在需要时才加载到内存中，并允许程序“脱机”保存可能从不使用的文本数据。考虑一个向用户显示帮助提示的程序，它可能在消息框里显示。虽然很多用户可能会喜欢这个特征，但是，其他用户可能不需要它，并快速关闭这个选项。通过将提示以字符串资源的形式保存，程序在每次运行时，不会因不使用的文本而浪费内存。字符串的第二个优点是，通过将程序的字符串组织成资源，开发者将字符串保存在一个地方，而不是零散地放置在几个资源模块中。同其他优点相比，这样，翻译人员只修改 RC 文件中的脚本，便可以创建一个外国语言的版本，修改完之后，开发者可以重新编译和链接，而不需要接触程序源代码。

你可以通过文本编辑器或者 Visual C++ 字符串编辑器，在 RC 文件添加或修改字符串资源，如果改变了一个字符串标识符的名字，那么，字符串编辑器还可以自动为新的标识符添加定义到 Resource.h 文件中。然而，字符串编辑器没有在源代码里放置新的标识符来代替旧的标识符。

提示字符串和工具提示

状态栏里的命令描述（又名提示字符串或定点描述）迅速成为 Windows 应用程序的标准过程。对其简洁的菜单命令，以及抽象艺术形式的令人难以理解的工具栏按钮，由于它们几乎没有提供有关其功能的线索，我们都曾经感到困惑。你可以在状态栏里仔细浏览，那里有提示字符串，使新用户不再需要有经验的用户的帮助。

好在要显示提示字符串，在程序中并不需要做很多工作。你通过给所有三个元素相同的资源标识符，便可以将一个提示字符串绑定到一个指定的菜单命令和工具栏按钮上。接着，创建一个状态栏，并添加 `CBRS_FLYBY` 标记到工具栏风格里。MFC 处理其余的工作，当一个菜单命令处于高亮状态，或鼠标光标停在一个工具栏按钮上时，便会显示正确的字符串。

添加 `CBRS_TOOLTIPS` 标记到工具栏风格里，可以生成工具提示，这是提示字符串的变种。工具提示是一个小弹出窗口，当鼠标光标从一个工具栏按钮上移过时，它显示简要的描述信息。Visual C++ 环境使用工具提示来标识它自己的工具栏按钮，可是，这个特征是可选的（要在 Visual C++ 中打开工具提示，通过单击 Tools(工具)菜单上的 Customize 打开 Customize(自定义)对话框，再打开 Toolbars 选项卡里的 Show ToolTips（显示工具提示）复选框。工具提示文本是提示字符串的一部分，像下面这样，在字符串结尾添加一个 `\n` 换行字符：

ID_PROMPT1 “Prompt string text in the status bar \nTooltip text”

这里是一个简单的例子，它描述了菜单、工具栏、提示字符串和工具提示之间的关系。虽然接下来的代码段描述了一个只打开和保存文档的 MFC 程序，但是，用黑体字显示的相关步骤也同样可应用程序于非 MFC 程序。

1. 给 RC 资源脚本文件里的相应菜单项、工具栏和提示字符串相同的标识符

IDR_MAINFRAME MENU

BEGIN

 POPUP “&File”

 BEGIN

 MENUITEM “&Open”,

 ID_FILE_OPEN

 MENUITEM “&Save”,

 ID_FILE_SAVE

 END

END

```
IDR_MAINFRAME BITMAP "res\\Toolbar.bmp"
```

```
IDR_MAINFRAME TOOLBAR 16, 15
```

```
BEGIN
```

```
    BUTTON    ID_FILE_OPEN
```

```
    BUTTON    ID_FILE_SAVE
```

```
END
```

```
STRINGTABLE
```

```
BEGIN
```

```
    ID_FILE_OPEN    "Open an exitsting document\nOpen"
```

```
    ID_FILE_SAVE    "Save the active document\nSave"
```

```
END
```

2. 在源代码里创建菜单

要用一步过程创建主窗口，并配上菜单，需调用 `CFrameWnd::Create` 成员函数：

```
Create(NULL, " Simple Demo", WS_OVERLAPPEDWINDOW, rectDefault,
        NULL,

        MAKEINTRESOURCE (IDR_MAINFRAME));
```

或者，在创建主窗口后，使用此代码来单独加载菜单。

```
Cmenu      menu;

menu.LoadMenu(IDR_MAINFRAME);

SetMenu(&menu);

menu.Detach();
```

3. 创建工具栏

```
CToolBar toolbar;                                     //In the header file

//Style defaults to WS_CHILD | WS_VISIBLE | CBRS_TOP
```

```
toolbar.Create(this);

toolbar.LoadToolBar(IDR_MAINFRAME);

//Add flyby and tooltip flags to style

toolbar.SetBarStyle(toolbar.GetBarStyle()|CBRS_FLYBY|CBRS_TOOLTIPS);
```

4.创建并初始化状态栏

```
CStatusBar          statusbar;                                //In the header file

UINT                nIndicator=ID_SEPARATOR;                  //Single pane in status bar

statusbar.Create(this);

statusbar.SetIndicators(&nIndicator,1);
```

AppWizard 生成相似的代码，为你做所有这些工作。你只需要打开字符串编辑器，并移去不需要的字符串资源，用描述程序里新菜单命令的字符串代替它们，但是，首先将遇到一个或更多的字符串资源。

文档字符串

当我们完成 DiskPie1 时，它将有六个程序资源，每一个都用 IDR_MAINFRAME 标识符进行标记：

- 应用程序图标
- 主窗口的菜单
- 菜单的加速键表
- 工具栏窗口
- 工具栏位图
- 表示文档的字符串资源

列表里的最后一项叫做文档字符串，它由 7 个子串组成，它们通过 \n 换行符分开。当在 AppWizard 里创建一个 SDI 或 MDI 项目时，你可以通过单击在 AppWizard 第四个步骤里的 Advance 按钮，来预先查看文档字符串的 7 个子字符串元素（Advance 按钮是在第 2 章里描述的）。单击这个按钮显示 Advanced Options（高级选项）对话框，如果你想改变它们，那么，可以在这里重新输入项目的默认子字符串。

如果我们使用 AppWizard 创建 DiskPie1，那么，AppWizard 将在 DiskPie1.rc 文件里像这样重新输入一个默认的文档字符串：

```
STRINGTABLE PRELOAD DISCARDABLE
```

```
BEGIN -
```

```
    IDR_MAINFRAME    “DiskPie1\n\nDiskPie1\n\n\nDiskPie1.Document\nDiskPie1 Document”
```

```
END
```

像其他字符串资源一样，文档字符串必须在 RC 文件里以单行显示，但是，由于空间的原因，在上面以两行显示。子字符串包含 MFC 指定给程序和它创建的文档的文本和名字。按显示的顺序，这些子字符串指定：

- 在主窗口标题栏上显示的程序名。
- 指定给程序创建的新文档名。
- 在 MDI 应用程序里使用的文档描述，它可以打开同一类型的多个文档。
- 以常规表达式文件形式组合的文档描述，就如同显示在 Open 和 Save As（另存为）对话框的文件类型列表一样。

- 给程序创建的文档的默认扩展名。
- 在系统注册表里表示文档类型的名字。
- 对程序创建的文档类型的描述。

由于 DiskPie1 并不创建文档，所以，我们对文档字符串感兴趣只是因为第二个子字符串。当这个子字符串为空时，MFC 给出新文档一个叫做“Untitled”的默认名字。你可能已经注意到，通过 AppWizard 创建的程序经常在它们的标题栏里带有“Untitled”文件名。对于像 DiskPie1 这样不需要存储它的数据的程序，调用它的“Untitled”显示，只能使用户感到混乱。一个解决办法是提供描述程序（而不是文档）的第二个子字符串的文本，像“Disk Usage”。详见下一节介绍。

为 DiskPie1 创建字符串资源

在 RC 脚本文件里已经有一个字符串表的 AppWizard 项目，通过双击 Workspce 窗口 ResourceView（资源查看）窗格里的 String Table 打开字符串编辑器。要为一个像 DiskPie1 这样的项目建一个新的字符串表，从 Insert 菜单里选择 Resource，然后双击 Resource Type（资源类型）列表里的 String Table。

字符串编辑器同字符串本身一样普通。唯一有趣的部分是在编辑器窗口的水平线，你可以在图 4-10 看到。这些线是字符串表里的分隔线，介于“段”的字符

串组之间，每段最多有 16 个字符串。字符串标识符的值决定了字符串属于哪一个段。标识符的值在 0 到 15 之间的字符串属于第一个段，标识符值在 16 到 31 之间的字符串属于第二个段，如此类推。

有时，一个段可以作为预读缓冲区，在许多磁盘驱动器上都有这种缓冲区，在预读缓冲里，磁盘控制器不仅读取需要的磁盘上的扇区，而且同时读取许多连续的扇区，把它们储存在一个内存缓冲区里，以备以后使用。预读缓冲区加快了磁盘使用的速度，因为一次磁盘操作通常会伴随有更多的磁盘操作，这时，控制器可以读取缓冲区的内容，而不是磁盘中的内容。用同样的逻辑，系统从可执行文件里一次读取一个段的字符串资源。当一个程序 LoadString API 函数从它的资源数据里读取独立的字符串时，Windows 读取字符串所属的那个完整的段，如果程序现在想要一个字符串，那么它将很快会要求其他的字符串。出于这个原因，你应该尝试通过给字符串连续的标识符值将相关的字符串分组。你可能已经这样做了，但现在你知道了这样做的原因。

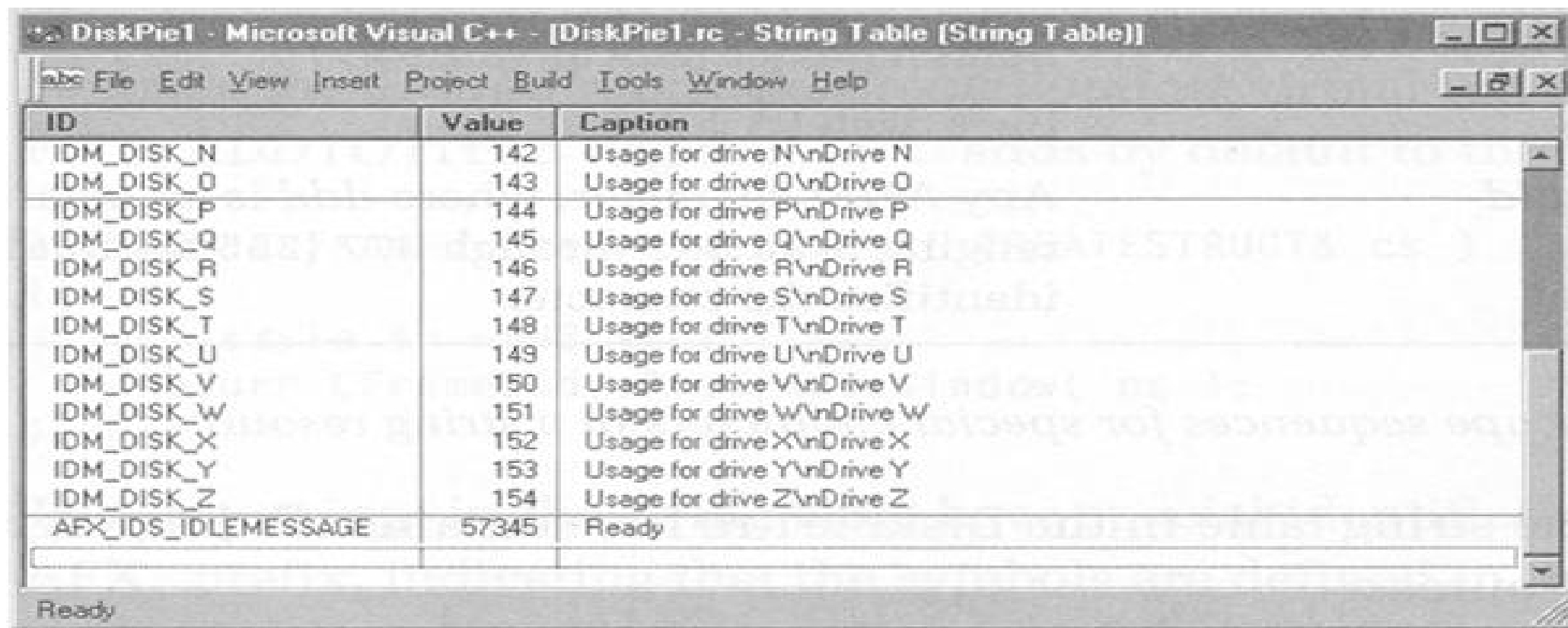


图 4-10 Visual C++ String 编辑器

同其他 Visual C++ 资源编辑器一样，字符串编辑器通过显示一个以点框表示的新项框指出表中下一个字符串的位置。要输入一个新的字符串，可选择新项框，然后开始输入字符串文本。String Properties 对话框显示一个默认的字符标识符。如果你需要，那么重新写标识符的名字，或从对话框组合框的下拉列表里选择一个名字。这个列表包含了为这个项目定义的所有标识符，包括那些在 Afxres.h 文件

里的标识符，它允许你选择一个名字，例如 `IDM_MEMORY`，而不是输入它。字符串编辑器自动用标识符的值来对表里的条目排序。

通过输入表 4-2 里的转义字符，可以将字符同字符串分开。此表涉及到 ASCII 和 ANSI 值的列表，请参阅附录 A。

表 4-2 在字符串资源中特殊字符的转义序列

转义序列	意义
<code>\n</code>	新行（ASCII 值 #10）
<code>\r</code>	回车（ASCII 值 #13）

续表

\t	制表符 (ASCII 值 #9)
\a	响铃字符 (ASCII 值 #7)
\\	反斜线 (\)
\ddd	任何 ANSI 字符, 其中 ddd 是一个 8 位字节数字, 范围从 001 到 377(10 进制数 255), 用于标识字符

DiskPie1.rc 文件里的字符串表包含程序所有的字符串资源：

STRINGTABLE DISCARDABLE

BEGIN

IDR_MAINFRAME "DiskPie1\nDisk Usage\n\n\n\n\n\n\n"

IDM_MEMORY "Memory usage\nMemory"

IDM_DISK_C "Usage for drive C\nDrive C"

IDM_DISK_D	"Usage for drive D\nDrive D"
IDM_DISK_E	"Usage for drive E\nDrive E"
.	
.	
.	
IDM_DISK_X	"Usage for drive X\nDrive X"
IDM_DISK_Y	"Usage for drive Y\nDrive Y"
IDM_DISK_Z	"Usage for drive Z\nDrive Z"
AFX_IDS_IDLEMESSAGE	"Ready"
AFX_IDS_SCSIZE	"Change the window size"
AFX_IDS_SCMOVE	"Change the window position"
AFX_IDS_SCMINIMIZE	"Reduce the window to an icon"
AFX_IDS_SCMAXIMIZE	"Enlarge the window to full size"

```
AFX_IDS_SCCLOSE
```

```
"Close the DiskPie1 application"
```

```
AFX_IDS_SCRESTORE
```

```
"Restore the window to normal size"
```

```
END
```

列表里的第一个字符串是 DiskPie1 的文档字符串，它同程序的其他资源有相同的标识符：

```
IDR_MAINFRAME
```

```
“DiskPie1\nDisk Usage\n\n\n\n\n\n\n”
```

这个指定的文本代替“Untitled”标题，否则，MFC 将会把“Untitled”写在标题栏中。注意一下图 4-20 里 DiskPie1 的标题文本，你将看到，MFC 怎样从文档字符串的前两个子串获取标题栏的文本。虽然这比仅仅删除资源行里的第二个子串更麻烦，但是，还会可能从标题栏里完整地移去“Disk Usage”。你必须找到 CMainFrame::PreCreateWindow 虚拟函数，清除 MFC 默认添加给窗口风格的 FWS_ADDTOTITLE 标记：

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
```

```
    cs.style &= ~FWS_ADDTOTITLE;
```

```
return CFrameWnd::PreCreateWindow(cs);
```

```
}
```

在字符串表里，最后的字符串有指定的 AFX_前缀标识符，它指示这些符号是在 Afxres.h 文件中定义的。例如，当程序在等待用户输入时，MFC 认可 AFX_IDS_IDLEMESSAGE 标识符，并在状态栏里显示被赋予的字符串值。为了方便起见，字符串只是读取“Ready”。其他的 AFX_符号为 DiskPie1 的系统菜单标识提示字符串，你可以通过单击程序的图标或右击标题栏来打开系统菜单。当用户从系统菜单里选择一项时，状态栏中将显示适当的 AFX_字符串。我们将看到，当我们在本章结尾写 DiskPie2 程序时，一般根本不需要包含这些提示字符串。

位图、工具栏、图标和光标

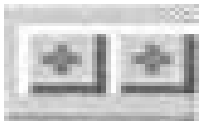
Visual C++ 图形编辑器是创建和修改程序的图形资源的地方，图形资源包括位图、工具栏、图标和光标。图标和光标是位图，用途很小。图标显示在任务栏按钮上或在目录列表里，光标就是为鼠标光标而设计的，当鼠标放置在程序的用户区域里时，它将提供服务。工具栏是包含几个位图图形的窗口，它在一个水平行中显示几个按钮。别的东西，例如，窗口里的图形，或对话框里的动态图片，均叫做位图。

Visual C++环境提供一个适用于所有场合的图形编辑器，所以，你不需要去学四种不同的用法。你可能看到在线帮助里的“toolbar editor”或“icon editor”参考，但是，这些只是意味着，Visual C++图形编辑器可以应用于任何一种特定类型的资源。图形编辑器可以处理不同资源类型的多个文档。图 4-11 显示了打开两种不同文档的编辑器，一个是我们所熟悉的 16×15 的位图，另一个是两色的 32×32 的鼠标光标。

对于每一种资源类型，编辑器的外观差别不大，文档窗口的左上角的图标指示你正在使用的资源类型。表 4-3 显示每一种资源文档对应的图标，并列出这个图形编辑器可以读写的文件类型的扩展名。

当使用 Open（打开）命令打开已有的资源文档，而且这个文档还具有表 4-3 第三列里的某个扩展名时，Visual C++自动打开图形编辑器，并调整自己，以适合这种资源类型。

表 4-3 图形编辑器图标和文件类型

资源	图标	输入文件类型	输出文件类型
Bitmap (位图)		BMP, DIB, EPS, GIF, JPG	BMP
Toolbar (工具栏)		BMP	BMP
Cursor (光标)		CUR	CUR
Icon (图标)		ICO	ICO

DiskPie1 有菜单、加速键和字符串数据，但还没有图形资源。对于 DiskPie11 这样的项目，可以用两种稍有不同的方法来创建一种新的图形资源。选择什么方法取决于你想象的这个图形文件是什么样子。如果打开一个项目，并想添加一个新的图形资源到项目中，从 Insert 菜单中选择 Resource，并双击位图或光标，或者

是在列表里的工具栏。Visual C++打开图形编辑器，在标题栏里显示为这种资源文档指定的标识符。根据资源类型的不同，标识符是像 IDB_BITMAP1，IDC_CURSOR1，IDI_ICON1 或 IDR_TOOLBAR1 这样的名字。以后在编辑器里打开的资源将得到相似的标识符，只是在数字上加 1，例如 IDI_ICON2，IDI_ICON3 等等。保存图形资源到文件中时，Visual C++给这个文件起的名字是标识符减去前缀，然后在项目的 Resource.h 文件里定义这个标识符。例如，要保存名为 IDB_BITMAP1 的资源，需添加如下一行到 RC 文件：

```
IDB_BITMAP1          BITMAP          DISCARDABLE          "res\\bitmap1.bmp"
```

并添加下面一行到 Resource.h 文件里：

```
#define IDB_BITMAP1    130
```

然而，你没有必要接受这些没有什么特征的名字。在保存资源前，通过单击 View（查看）菜单上的 Properties，并输入一个新的标识符名字来给它提供一个描述性的标识符。在相同的 Properties 对话框里，也可以给这个图形文件指定一个名字。

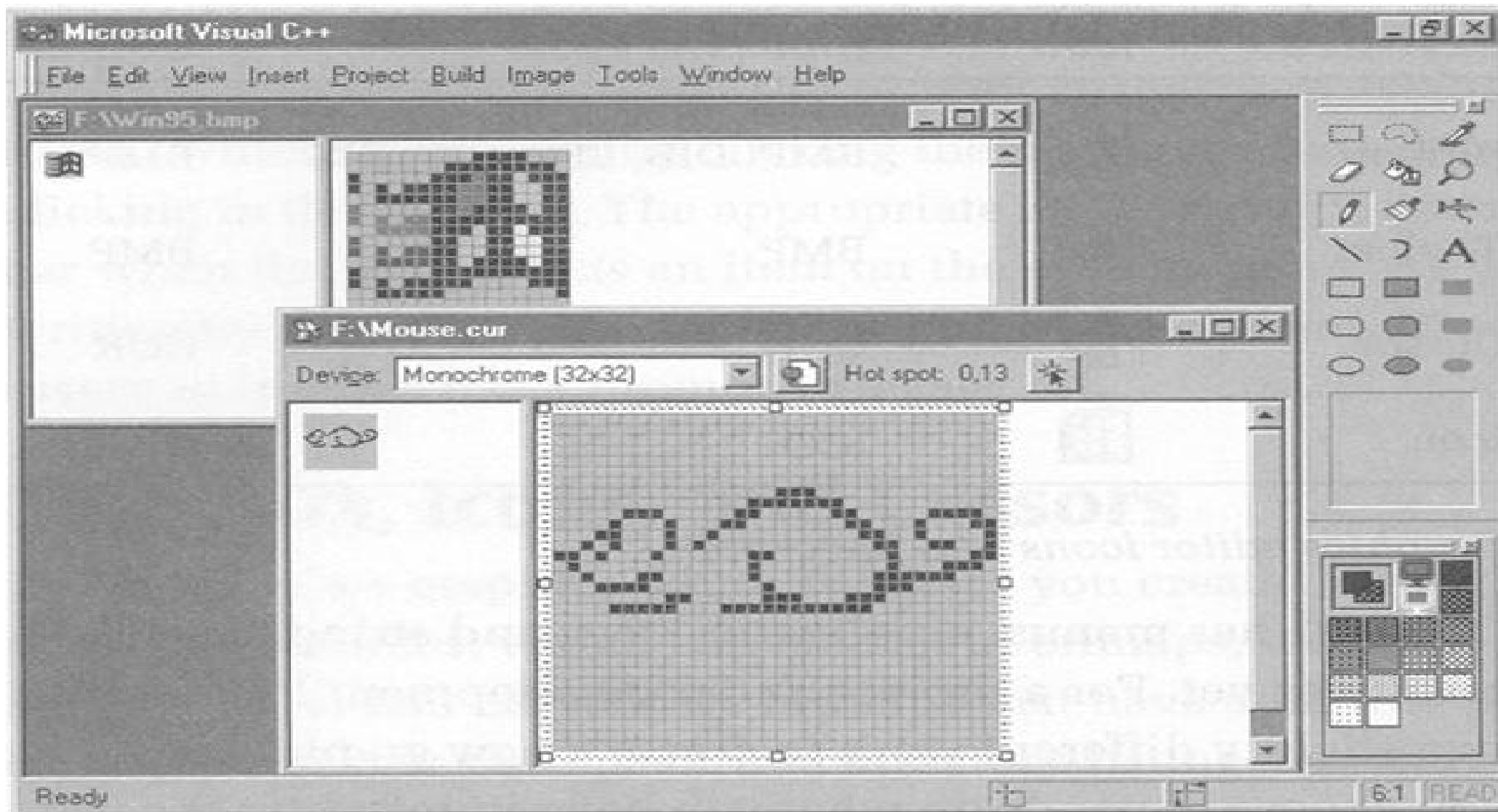


图 4-11 Visual C++ 图形编辑器具有两个打开的文档

在启动图形编辑器的第二种方法中，你可以创建一个新的图形资源，但不添加它到项目文件列表里。你可能想这样做，例如，创建一个资源库，或为工具栏按钮设计位图。这种方法不要求打开项目，只单击 File（文件）菜单上的 New，并在 File 选项卡里，从列表里选择你想要的资源类型：Bitmap File（位图文件），Icon File（图标文件）或 Cursor File（光标文件）。

图形编辑器的工作区是分成两个窗口。默认状态下，左边窗口以图形的实际尺寸显示图形，右边窗口显示的图形放大到大约 36（ 6×6 ）倍。放大的图形有一个覆盖形式的网格，网格中，每一个方块表示实际图形上的一个像素。如果你以前使用过画图程序，例如 Microsoft Windows 的 Paint 应用程序，那么，就会对 Visual C++ 图形编辑器很熟悉。

通过单击左边或右边窗口中的任何地方，来选择要绘制的图形。如果要处理细节，可以在大的工作区域里进行，并在实际尺寸图形窗口里观察效果。将两个窗口分割开的分隔条是可以移动，只要用鼠标向左或向右拖动它即可。要开始画图形，通过单击显示在图 4-12 里 Graphics 工具栏上的一个按钮，来选择一个合适的工具。与在 Visual C++ 环境里使用其他工具栏一样，你可以通过向内或向外拖动工具栏来缩小或放大它。

Graphics toolbar

Colors palette

Transparency selector

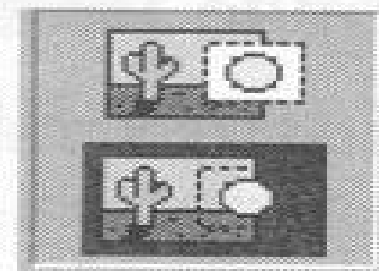
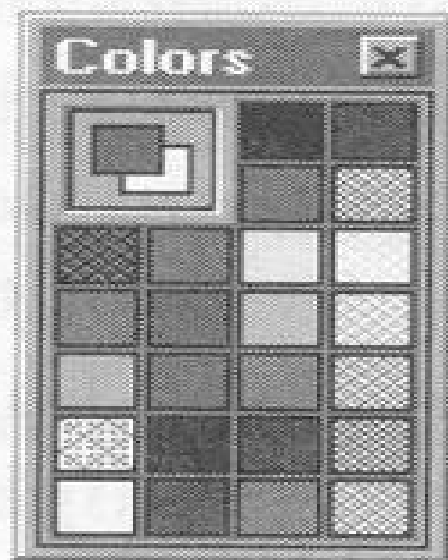
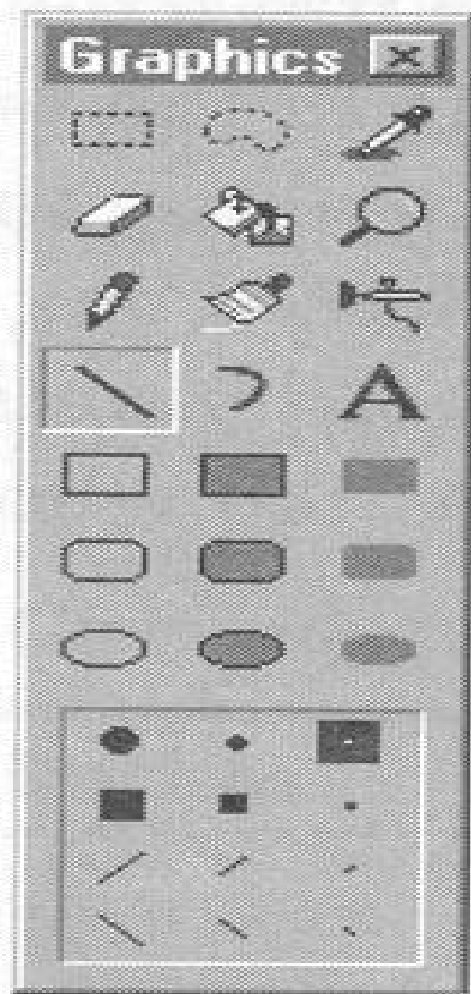


图 4-12 Visual C++图形编辑器中的工具

图形编辑器工具很灵活，而且很友好，只需要几分钟就可以学会，然而，有几点可能看起来不那么直观，所以要简要讨论一下。首先，图形的背景颜色依赖于图形的类型。图标和光标只有一种透明的背景色，在编辑器窗口里用蓝绿色进行的着色。当 Windows 在屏幕上画图标或光标时，它只画前景色，在透明背景色下的所有像素均不擦除。另一方面，位图的背景色是不透明的。一个位图在屏幕上作为一块画出来的，它将覆盖它下面的任何东西。背景色是否透明，可能是这两种图形资源类型之间的最大区别。

当你单击 Rectangle Selection（矩形选择），Irregular Selection（不规则图形选择）或 Text 按钮时，透明选择框显示在 Graphics 工具栏的底部，它也是编辑器 Image 菜单上 Draw Opaque（不透明画图）命令的工具栏版本。要理解选择框的用途，考虑一下一个分为两层的图形，一层覆盖在另一层上。上层是一个漂浮的图形，你可以移动，或设定它的位置。下面这一层叫做底层，是固定的。利用选择框，你可以设定顶层的背景透明度，但不能影响底层的透明度。选择透明度选择框的顶部图标，意味着覆盖图形层上的背景像素应该当作前景颜色，使上面的一层成为实心的矩形框。选择框的底部图标使上面一层透明，实际上，是从这一层移去背景像素。例如，下面是当打开或关闭透明度时，在底层图形上输入字母“A”时的外观：



不管透明度的设定如何，底层位图图形的背景颜色（在当前情况下是白色）均保持不透明，以便在屏幕上显示的位图覆盖位图矩形区域内的任何像素。然而，当显示一个位图时，程序可以模拟透明的像素，首先掩盖位图的背景颜色，然后用位图将出现的屏幕区域的副本来替代它。如果你对这个技术感兴趣，完全可以从

Jeff Prosise's Programming Windows 95 with MFC 里题为 “ Bitmaps, Palettes, and Regions ” 那章的透明化位图教程中，找到详细的说明。通常，你只是想确保一个位图与显示它的窗口拥有一样的背景颜色，以便得到一种透明的假象。有一种简单的方法可以实现这种操作，下一节将详细说明。

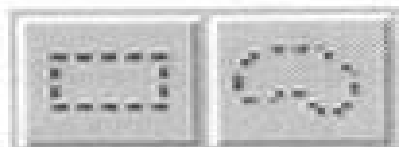
另一种图形编辑器的隐藏特征是，鼠标的左右按钮分别对应于前景色和背景色。例如，用鼠标左键单击 Colors 调色板上的一种颜色，来选择前景色，用鼠标右击选择背景色。你可以在图形上通过拖动或单击适当的鼠标按钮，来选择使用一种颜色画图。

表 4-4 总结了在图形编辑器里能够看到的工具栏按钮。如图 4-12 所示，在 Graphics 工具栏里，按钮是以平面方式显示的。表 4-4 显示了处于凸起状态下的按钮，这可以帮助我们区别每一个按钮。

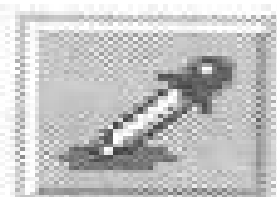
图 4-4 Graphics 工具栏按钮

按钮

说明

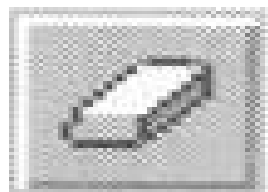


利用 Rectangle Selection 和 Irregular Selection 工具, 可以标记要移动、清除或复制的图像部分。单击按钮, 然后, 拖动交叉十字或光标, 指向要标记的矩形或区域。在释放鼠标按钮时, 在标记的区域周围将出现选择框。你可以移动选择内容, 方法是使用鼠标拖动它; 要清除它, 则按 Del 键; 复制它到剪贴板的方法是按 Ctrl+C

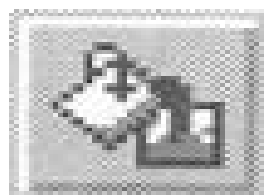


利用 Select Color (取色) 工具, 可以从图像中选取颜色, 而不是从 Colors 调色板中。单击工具栏上的按钮, 然后单击要选取颜色的图像上的区域。使用鼠标左键可以选取前景色, 使用鼠标右键可以选取背景色

续表

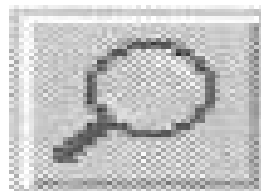


Erase (橡皮擦) 工具将光标改为一个方块，可以在图像上拖动它，来擦除像素。擦除的像素改变为当前前景色，这要取决于当前的资源类型。前面提到过，在 Colors 调色板的左上角，图标和光标的背景是不透明的颜色。要改变擦除块的大小，可以单击工具栏底部选择框中的一个尺寸图标

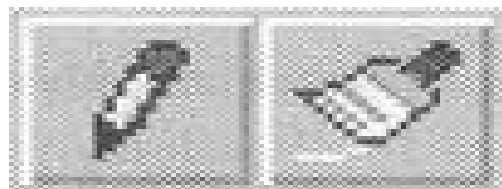


Fill (填充) 工具将颜色的像素改变为当前的前景或背景色。单击 Fill 按钮，然后单击要改变的颜色中的任何地方。若使用鼠标左键，则填充前景色；使用鼠标右键则填充背景色。编辑器将更改此颜色的所有连续像素，来填充颜色。像素必须水平或垂直接触，以对角方式接触的像素被认为是不连续的

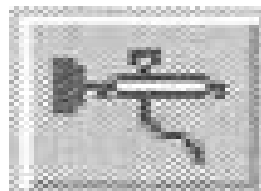
续表



要改变所选择图形的大小，可以使用 Magnify（放大）工具，并选择一个值 1，2，6 或 8。放大值指定对应于实际图形中一个像素的实际像素数。如果放大的图形有网格，则放大值确定的是每个网格的宽度和高度

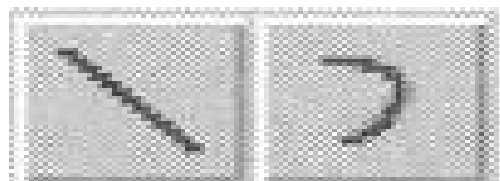


单击左边的图片按钮，可以手绘图，然后，拖动鼠标按钮，以留下像素的轨迹，颜色是前景色或背景色。Pencil（铅笔）工具只画出一个像素宽的细线。使用 Brush（刷子）可以画出粗线，在工具栏底部的选择框中可以选择刷子的宽度



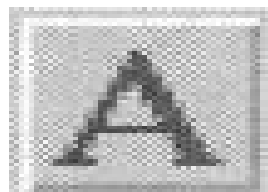
Air Brush（喷雾）工具可以画出随机的颜色图案，模拟在画布上轻微喷射的效果。在工具栏的选择框中，可以选择喷射的密度和尺寸

续表

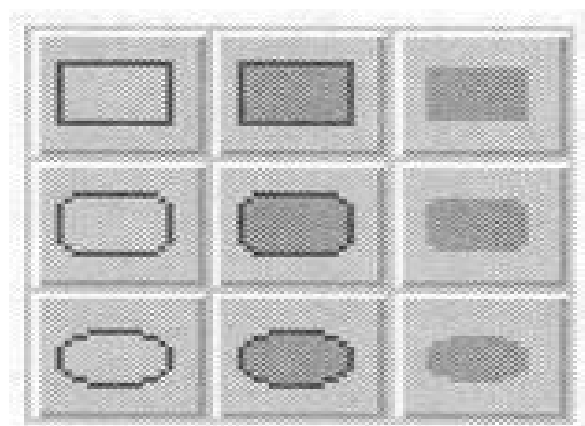


画线工具可以画直线，还可以画曲线，曲线从初始位置开始弹性弯曲。单击要开始画线的位置，拖动光标到线的终点，然后释放鼠标键来确定此线。Curve 工具还额外需要一个步骤：在释放鼠标键设置终点后，移动光标来设置曲线的曲率。在曲线以自己需要的方式成形后，双击便可以确定曲线（双击的鼠标键必须与拖动光标鼠标键相同）。如果在画线的过程中改变了主意，可以按 `Esc`，或者单击另一个鼠标键，来重新开始

续表



Text（文本）工具显示一个小窗口，可以从中输入在图形中出现的文字，所用的颜色是当前的前景色。如前所述，在透明选择框中选择文字的透明性，然后，在选择完 Text 工具时，按 Esc 键



这些工具可以画矩形、圆角矩形和椭圆形，这些图形可以是实心的，也可以是空心的。选择一个工具，然后，以前景色或背景色来画出形状，方法是在图形上，从左上角向右下角拖动鼠标光标。按下 Shift 键可以强制画出某种形状，使矩形画为正方形，或者将椭圆画为圆形。在拖动光标的过程中，如果要取消操作，可以按 Esc，或者单击另一个鼠标键

位图

在为一个新的位图打开图形编辑器时，它提供一个边长为 48 像素的正方形的干净区域。然而，位图不要求一定是正方形，它们可以是边长不大于 2048 个像素的任何尺寸的矩形。要调整工作区域的大小，拖动工作区域边界上一个调整大小的手柄，在拖动手柄的同时，注意一下编辑器状态栏里的新尺寸。也可以通过单

击 View（查看）菜单上的 Properties 打开 Bitmap Properties（位图属性）对话框，从中输入想得到的大小。

这里是一个 C 程序如何显示位图资源的例子。此代码段假设位图最初是保存在文件 Res\Bitmap.bmp 里，并在程序的 RC 文件里，通过 BitmapDemo 名进行标识，LoadBitmap 函数用和名字相同的字符串加载资源。

```
// Resource declaration in the RC file
```

```
BitmapDemo      BITMAP      “res\\bitmap.bmp”
```

```
// In the C source file
```

```
HBITMAP      hbm;          // Declare a globa handle for the bitmap
```

```
.
```

```
.
```

```
.
```

```
// Load the bitmap in WinMain or the InitInstance procedure
```

```
static char szAppName [ ] = “BitmapDemo”;
```

```
hbm = LoadBitmap( hInstance, szAppName);
```

```
.
```

```
.
```

```
.
```

```
// Display the bitmap in the window procedure
```

```
case WM_PAINT:
```

```
    hdc = BeginPaint( hwnd, &ps )'
```

```
    hdcMemory = CreateCompatibleDC( hdc );
```

```
    GetObject ( hbm, sizeof (BITAMP),&bm);
```

```
    SelectObject( hdcMemory, hbm);
```

```
    BitBlt( hdc, x, y, bm.bmWidth, bm.bmHeight,
```

```
        hdcMemory, 0, 0, SRCCOPY )'
```

```
    DeleteDC( hdcMemory);
```

```
EndPaint( hwnd,&ps);
```

```
break;
```

在 MFC 程序里,显示这个位图的步骤是相似的。首先,用资源初始化一个 CBitmap 对象：

```
bitmap.LoadBitmap(szAppName);
```

然后,在窗口的 OnDraw 函数里显示这个位图：

```
BITMAP    bm;
```

```
CDC        dcMemory;
```

```
bitmap.GetObject( sizeof (BITMAP), &bm );
```

```
dcMemory.CreateCompatibleDC( Pdc );
```

```
dcMemory.SelectObject( &bitmap );
```

```
PDc->BitBlt ( x, y, bm.bmWidth, bm.bmHeight, &dcMemory, 0, 0, SRCCOPY );
```

在结束这一节中有关位图的说明之前,让我们再次看一下,上一次讨论的有关位

图透明性的题目。我们已经知道，位图的背景色是不透明的，但是，如果这个位图的背景色是白色，而且，这个位图将显示在一个背景色也是白色的窗口里，那么，这个位图的正方形形状会被掩盖。如果给出透明背景色的设想，那么，只有不是白色的前景色可以显示出来。当时，如果窗口的背景色不是白色的，将是什么样的呢？在那种情况下，在先前代码段里显示的位图将暴露出它的正方形外形，这可能不是你想要的效果。

应用程序窗口一般采用标识为 `COLOR_WINDOW` 的系统窗口颜色，默认状态下，它是白色的。然而，一个程序可以通过调用 `SetSysColors` 来改变系统窗口的颜色，或者，用户可以在 Windows Control Panel 里的 Display（显示）部分改变颜色（如果想试一试，那么，在 Display Properties（显示属性）对话框，从 Appearance（外观）选项卡上的 Item 组合框里选择 Window，从 Color 下拉列表里选择一个新的颜色，然后单击 OK（确定））。通过使用 `LoadImage` API 函数，而不是 `LoadBitmap` 函数，来装载位图，并像下面这样指定 `LR_LOADTRANSPARENT` 标记，程序可以保证位图的背景色总是同 `COLOR_WINDOW` 颜色一致：

```
hbm=LoadImage( hInstance,szAppName,IMAGE_BITMAP,  
               0,0,LR_LOADTRANSPARENT);
```

这个函数查看图像里的第一个像素的颜色，它处于矩形位图的左上角，推测起来，应该是背景的一部分。然后，`LoadImage` 函数用当前的 `COLOR_WINDOW` 颜色来代替位图颜色表里相应的条目。这样，在这个位图里组成背景的所有像素

都用默认的窗口颜色显示。唯一的警告是，如果位图有 256 多种颜色，那么 `LR_LOADTRANSPARENT` 将不起作用。

我们还没有完全使用 `LoadImage` 函数。正如我们将在下一节中看到的那样，这个函数也可以装载图标。但是现在，让我们看一下位图怎样成为工具栏。

工具栏

在 Visual C++ 中创建工具栏，只不过是设计包含工具栏按钮图形的位图。这个位图作为一个 BMP 文件保存，并在项目的 RC 文件里用一个标识符名字引用它：

```
IDR_TOOLBAR      BITMAP      "res\\toolbar.bmp"
```

标识符的名字和文件名都由你决定。DiskPie1 用 `IDR_MAINFRAME` 命名它的工具栏，同程序的其他资源保持一致，如前所述，允许调用到 `CSingleDocTemplate` 构造器。

工具栏位图是一系列覆盖工具栏按钮的图形，每一个按钮都有一个图形。默认状态下，每一个图形都是 16 个像素宽，15 个像素高，它适合标准的 24 个像素、宽 22 个像素高的工具栏按钮。通过以典型的 Windows 方式拖动编辑器工作空间的边界，可以设置一个图形的尺寸是大一点，还是小一点，是宽一点，还是窄一点。由于你不能使一个工具栏里按钮具有不同的尺寸，所以，新尺寸将应用于所

有的工具栏图形。当保存工作时，Visual C++自动指定 RC 文件里工具栏按钮的新尺寸。

如果 AppWizard 为你生成项目，那么，它在项目的 Res 文件夹中创建一个叫做 Toolbar.bmp 的文件。这个文件包含对应于命令 New, Open, Save, Cut, Copy, Paste, Print 和 Help 的工具栏按钮的图形。图 4-13 是在 Toolbar.bmp 里放大的位图和最终的工具栏。

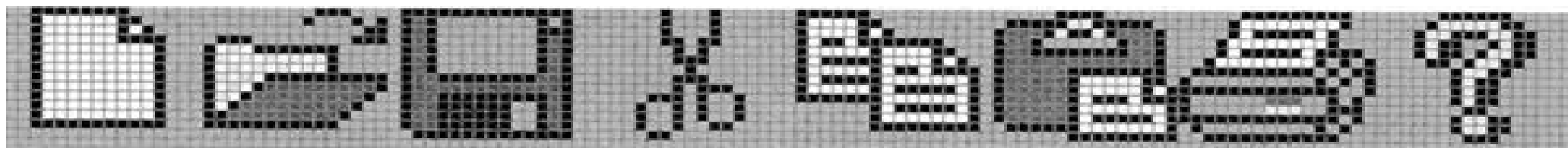


图 4-13 AppWizard 生成的默认工具栏

在下一段代码中，显示了 AppWizard 项目中 RC 文件的创建工具栏的脚本。工具栏脚本里的每一个按钮，都具有对应于本章“菜单和加速键”一节脚本里列出的菜单命令的相同标识符。

```
IDR_MAINFRAME          BITMAP          MOVEABLE PURE
    "res\\Toolbar.bmp"
```

```
IDR_MAINFRAME TOOLBAR DISCARDABLE    16, 15
```

BEGIN

BUTTON ID_FILE_NEW

BUTTON ID_FILE_OPEN

BUTTON ID_FILE_SAVE

SEPARATOR

BUTTON ID_EDIT_CUT

BUTTON ID_EDIT_COPY

BUTTON ID_EDIT_PASTE

SEPARATOR

BUTTON ID_FILE_PRINT

BUTTON ID_APP_ABOUT

END

脚本里的 BITMAP 语句指向保存位图的项目的 Toolbar.bmp 文件。TOOLBAR 语句用 IDR_MAINFRAME 值标识工具栏资源，并为每个按钮指定尺寸为 16×15 个像素的图形。SEPARATOR 语句使得邻近按钮之间有些空隙，它是通过 BUTTON 语句定义的。

MFC 程序通过调用 CToolBar::Create 来创建工具栏。当函数返回时，它创建的工具栏只不过是一个空的子窗口。下一步是调用 CToolBar::LoadToolBar 函数，来读

取工具栏按钮数据，装载工具栏位图，并画出这些按钮，所有这些都在一步内完成。

窗口为每一个在工具栏脚本里的 BUTTON 语句提供一个空按钮，并在每个按钮上画出一个对应的位图图形区。例如，要装载在上面脚本里定义的工具栏，程序可以声明一个 CToolBar 对象，叫做 m_toolbar，并在资源里包含这些行：

```
m_toolbar.Create( this );                                // Create the toolbar window
```

```
m_toolbar.LoadToolBar ( IDR_MAINFRAME ); // Load the bitmap images
```

在 Visual C++ 里，有两种方法可以从头开始创建一个工具栏。第一个方法是我们

很熟悉的：在一个打开的项目里，单击 Insert 菜单上的 Resource，并双击在 Resource Type 列表里的 Toolbar。这将启动图形编辑器，以便对工具栏进行改变，在图 4-14 里，图形编辑器显示了三个分割的窗口。底部的两个窗口显示了你正在工作的当前工具栏按钮的实际尺寸，以及放大尺寸的视图，上部的窗口显示了一个工具栏的完整视图。开始在一个按钮上工作时，按钮图形自动显示在工具栏视图里，在你编辑时，实时改变按钮显示。当完成了一个按钮时，单击上面窗口里空的新项目按钮，为下一个按钮刷新工作区域。你可以在工具栏里拖动它，以改变按钮的位置，或把它完全拉离工具栏，以便删除按钮。要在如图 4-14 里所示的按钮之间添加分割空隙，向左或向右拖动一个按钮到大约半个按钮宽度就可以了。你可以用同样的方法去掉这个空隙。

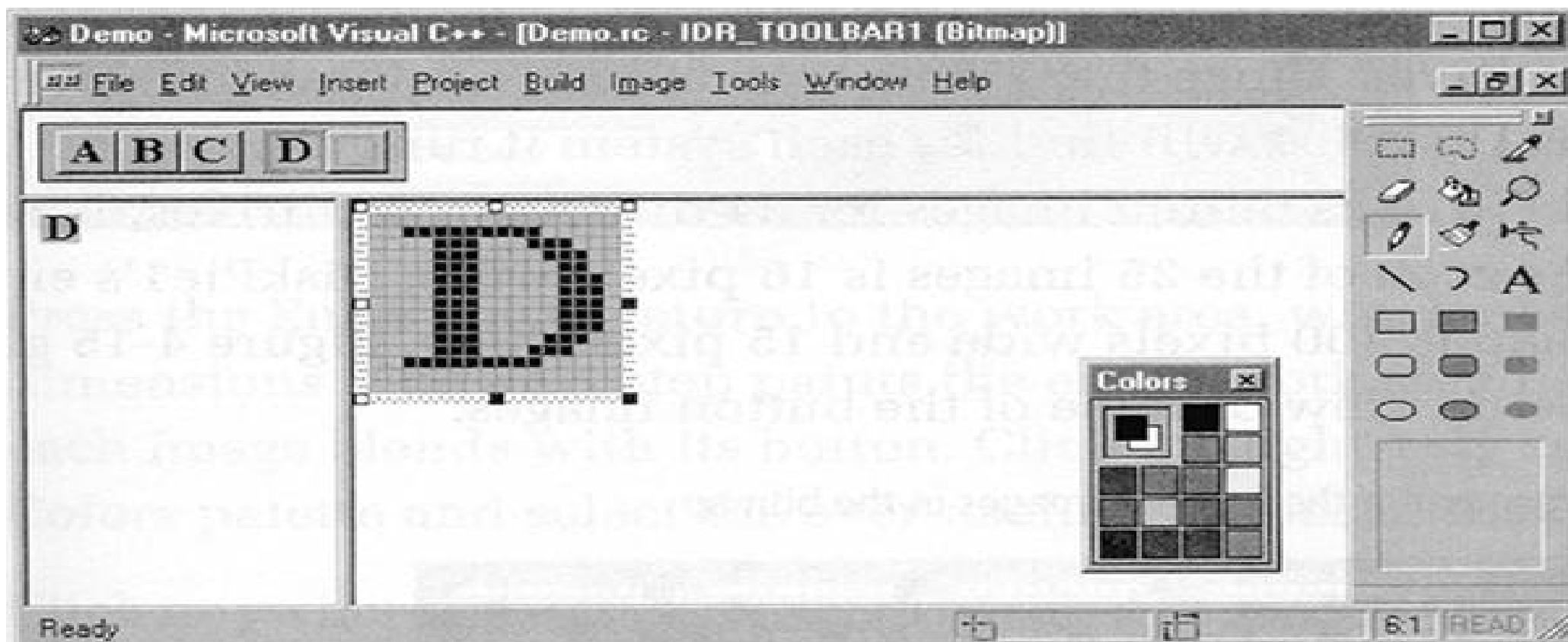


图 4-14 在 Visual C++图形编辑器中创建工具栏

保存工具栏时，不要担心空的项目按钮会一起保存。在编辑器写到 RC 文件里的工具栏脚本里，不会包括空的项目按钮。当在一个程序的主工具栏中工作时，应当给每一个按钮同菜单项对应的相同标识符，例如 `ID_FILE_NEW` 或 `ID_FILE_PRINT`。在 `Toolbar Button Properties`（工具栏按钮属性）对话框里输入

按钮的标识符，通过在工作区域里的任何地方双击鼠标，或通过从 View（查看）菜单上选择 Properties，来显示上述对话框。

在创建工具栏的第二种方法中，要求先设计位图图形，然后，将得到的结果转化为工具栏。对于 DiskPie1，这种方法更加方便些。

为 DiskPie1 创建工具栏

DiskPie1 是与众不同的，它在运行时决定需要的工具栏按钮，以及它们对应的图形的个数。在 DiskPie1.rc 文件里，工具栏脚本只有一个入口：

```
IDR_MAINFRAME    TOOLBAR 16, 15
```

```
BEGIN
```

```
        BUTTON        IDM_MEMORY
```

```
END
```

在程序启动时，便添加了为磁盘驱动器显示使用状况饼图的按钮。例如，如果 DiskPie1 找到了四个磁盘驱动器，分别指定为 C，D，P 和 R，那么，在它创建工具栏时，它指定五个按钮：一个对应于 Memory 命令，其余四个对应于四个磁

盘驱动器。由于在 DiskPie1 运行之前，没有办法知道系统中运行着多少个磁盘驱动器，所以，DiskPie1 的工具栏位图有 24 个不同磁盘驱动器的按钮图形，它们分别以 C 到 Z 标记。由于这 25 个图形都是 16 个像素宽，所以，DiskPie1 完整的工具栏位图是 400 个像素宽、15 个像素高。在图 4-15 中，可以看到一些按钮图形的特写视图。

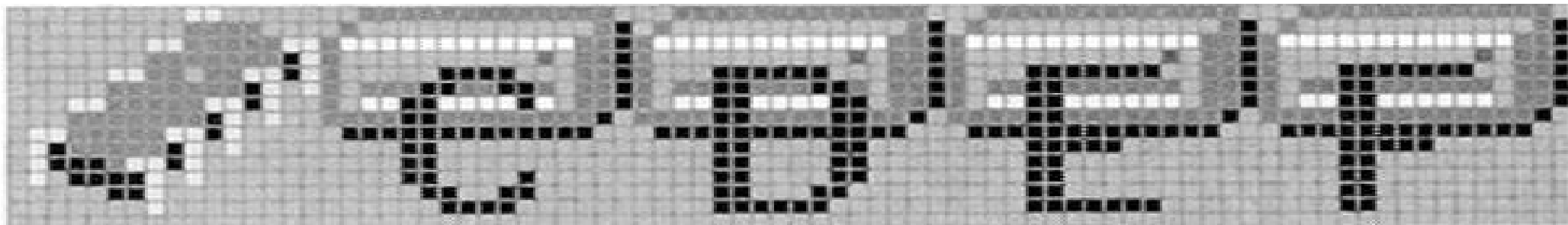
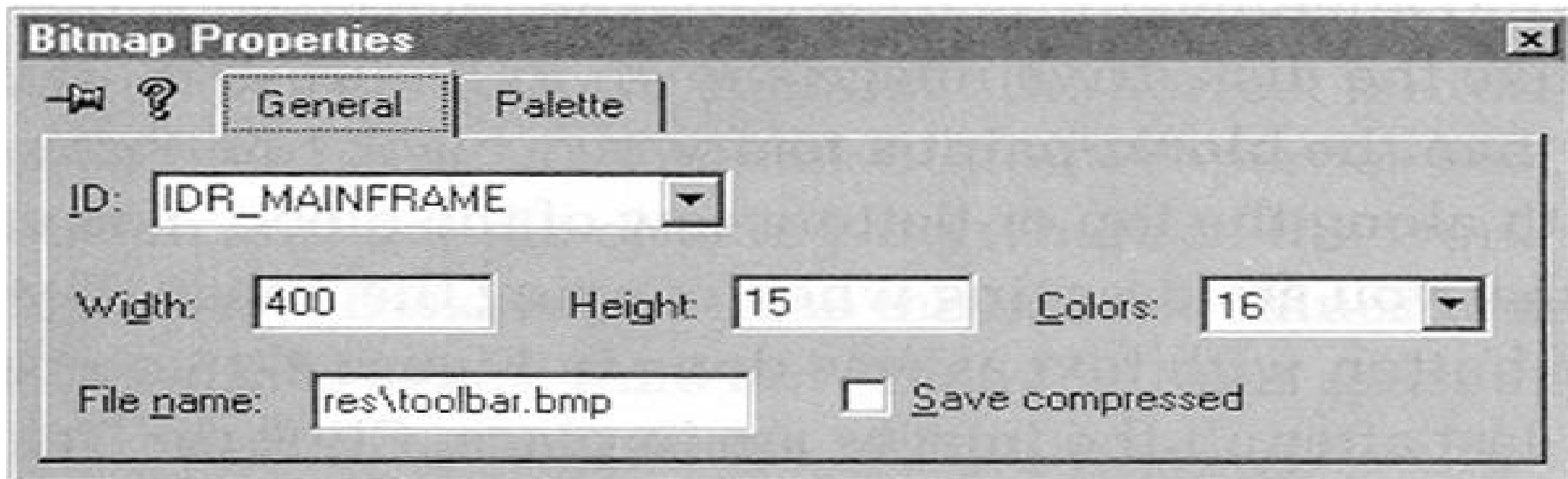


图 4-15 DiskPie1 的工具栏位图

可以看到，生成位图并不是十分困难，只用了大约 20 分钟的时间。这个秘密就是，告诉图形编辑器你正在创建一个位图，而不是一个工具栏。第一步在下列两种情况下是一样的：通过单击 Insert 菜单上的 Resource，来打开图形编辑器。为了用重复的图形加宽如图 4-15 所示的位图，从资源类型列表里选择 Bitmap，而不是 Toolbar。这将可以在一个连续的条中操作按钮图形，而单独操作每个按钮。在图形编辑器里，将普通位图转换为工具栏是很容易的，设计它就是做这个的。

这里是使用重复图形生成加宽的位图的步骤。双击编辑器工作区域的空白区域的任何地方，打开 Bitmap Properties 对话框。输入工具栏标识符，对于 DiskPie1，

是 IDR_MAINFRAME，并给 BMP 文件起一个文件名。通过增加按钮的数目，来增加一个工具栏按钮的宽度，并输入这个数字作为位图的宽度。位图的高度是一个按钮的高度。对于 DiskPie1，对话框看起来像这样：



按下 Enter 键返回工作区域，现在它有了新的尺寸。下一步，用亮灰色画出整个工作区域，以便每一个图形能和它的按钮相融合。在 Color 调色板里单击亮灰色框，并选择 Graphics 工具栏里的 Fill 工具。



单击工作区域格子中的任何地方，来用亮灰色画整个区域。现在，在工作区域里的一个 16×15 的块里，画第一个按钮的图形（或是任何的按钮尺寸）。一旦已经画出第一个图形，就可以通过单击 Rectangle Selection 工具来复制它，并在图形的 16×15 方块上拖动光标。是用鼠标左键，还是用右键拖动，取决于你是移动这个位图，还是拷贝它。如果想移动图形到位图工作区域的其他地方，那么使用鼠标左键。当你释放鼠标按钮时，在图形的周围出现一个选择框，此时可以拖动它来重新定位选择区域。按下 Ctrl 键拖动选择框，便移动了被选择图形的一个副本，而不是图形本身，但是，有一种更好的方法来复制图形。

要生成一个图形的副本，需单击 Rectangle Selection（选择）工具，并用按下鼠标右键（而不是鼠标左键）来选择图形。释放鼠标按钮时，被选择图形的一个副

本将跟随鼠标光标而移动。将副本定位在工作区域的任何地方，并单击，将它放置在选择的位置上。单击左键将放下图形的一个副本，单击右键将放下图形的遮罩，其中，前景像素转化为当前图形的背景颜色，图形里的背景像素被当作透明的小洞。你可以用这种方法生成任何数量的副本。完成操作时，按下 ESC 键，或选择另一个工具来返回一般编辑模式。

你将发现图形横越整个 16×15 方块，像图 4-16 里的磁盘驱动器图形一样，那么定位它将更加容易。如果图形比方块窄，那么在方块的顶部或底部画一条 16 个像素宽的临时黑线。然后，当你拷贝图形时，就可以准确看到正在拖动什么。如果你打算使用文本标记每一个按钮，像图 4-15 里所做的那样，那么，最后，在所有的图形都到位后，将添加标记。单击 Text 工具，输入文字，并拖动文字图形到指定位置。在做完每一个文字之后，按下 ESC 键取消 Text 工具。

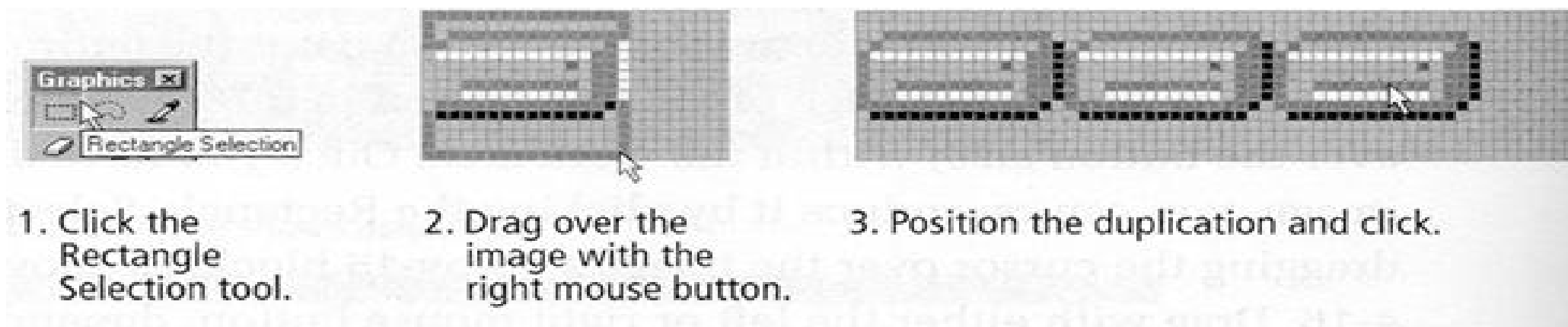


图 4-16 复制选择的图形

完成了位图的设计时，单击 Image 菜单上的 Toolbar Editor，来转化位图到一个工具栏上，在 New Toolbar Resource（新工具栏资源）对话框里，接受建议的 16×15 的按钮尺寸。你可以使用相同的 Toolbar Editor 命令，在工具栏和位图之间来回转换。单击 File（文件）菜单上的 Save（保存）命令，将新的工具栏脚本写到 RC 文件里，并将工具栏位图保存成在前面 Bitmap Properties 对话框里命名的 BMP 文件。Visual C++ 为新工具栏脚本中的按钮指定默认的标识符值，现在，在 DiskPie1.rc 文件里，它们看起来像下面这样：

```
IDR_MAINFRAME TOOLBAR DISCARDABLE 16, 15
```

```
BEGIN
```

```
BUTTON        ID_BUTTON40030
```

```
BUTTON        ID_BUTTON40031
```

```
BUTTON        ID_BUTTON40032
```

```
.
```

```
.
```

```
.
```

```
END
```

按钮标识符最终必须改成同对应菜单命令的标识符相同的标识符，即 `IDM_MEMORY`，`IDM_DISK_C`，`IDM_DISK_D` 等等。这将保证单击一个工具栏按钮有同选择相应菜单命令相同的效果。我们应该使用 `Toolbar Button Properties` 对话框，在图形编辑器里具有指定的正确按钮标识符，调用这个对话框的方法是选择一个按钮，并双击工作区域的任何地方。但是，这没有什么必要。当 `DiskPie1` 确定运行时必须在工具栏上显示多少按钮时，`DiskPie1` 给按钮指定正确的值。在后面，我们将编辑工具栏脚本，用在这节开头引用的简短的脚本来代替它，同时也从 `Resource.h` 文件中移去为按钮标识符定义的多余的 `#define` 语句。

图 标

图标是特殊的位图，设计用来用图形方式表示程序或文档。一般情况下，图标指定给一个框架窗口，以便在窗口的标题栏里显示图形，当图标指定给程序的主窗口时，图标资源叫做程序图标或应用程序图标。本节集中讲述怎样创建应用程序图标，它是最普遍使用的一种图标资源。但是，不管图标是表示主窗口，还是屏幕上的另一个对象，图标不过就是图标，它是在 Visual C++图形编辑器里创建的，而且创建的方式相同。

一个图标资源包含多个图形，它们经常是同样设计的不同尺寸的图形。例如，Microsoft 建议，Windows 程序提供给它图标资源的三个图形，每一个图形具有不同的尺寸：

- 一个边长为 16 个像素的正方形，16 色图形，Windows 将在程序的标题栏、任务栏和用小图标显示的目录列表里显示它。
- 边长为 32 个像素的正方形，16 色图形，在 About（关于）对话框这样的对话框窗口里使用，或在桌面上表示程序，或在用大图标显示的目录列表里显示。
- 边长为 48 个像素的 256 色图形，在系统 Display Properties 对话框里，若选中 Effects（效果）选项卡里的 Use Large Icon（使用大图标）选项

时，它在 Windows 98 里代替 32×32 图标（要打开这个对话框，在桌面空白区域按下鼠标右键，并选择 Properties。Windows 95 要求安装 Microsoft Plus! 包，在这种情况下，Use Large Icons 选项在对话框的 Plus! 选项卡里）。

你可以在 Explorer（资源管理器）窗口里看到大的和小的应用程序图标的例子，或在 Visual C++ 环境里通过打开 Save As（另存为）或 Open（打开）对话框看到。在目录列表窗口的空白区域右击，并从上下文菜单的 View 命令中选择 Large Icons（大图标）。同一个菜单里的 Small Icons（小图标）命令显示 16×16 的图形。

在 Windows 编程中，应当使用具有独特魅力的图标，但这不是必须的。如果一个程序在它的资源里根本不包括任何图标，它仍可以使用在 Winuser.h 文件里的标准系统图标，像 IDI_APPLICATION 和 IDI_WINLOGO。当系统图标处于 16×16 像素尺寸状态下，它们看起来像下面这样：



IDI_APPLICATION



IDI_WINLOGO

在一个图标资源里封装的三个图形，可以将可执行文件的大小增加大约 5000 个字节。如果它看起来太大，便可以用一个单独的 16×16 或 32×32 图形创建一个图标，当用其他尺寸显示图标时，Windows 按适当的比例显示。然而，你可能对这个结果感到失望，因为曲线和斜线将产生边界粗糙的效果。

下一节显示怎样用 Visual C++ 图形编辑器创建一个图标，但是，首先让我们看一下程序怎样加载图标资源。通常，C 程序在它创建主程序窗口时加载应用程序图标。如果图标资源只包含一个图形尺寸，那么程序可以调用 LoadIcon API 函数（在旧的 Windows 版本里使用的方法）。但是，要从相同的图标资源里加载多个图形，这个程序应该使用 LoadImage 函数代替前面的函数。由于使用旧版本的 WNDCLASS 结构的 RegisterClass 函数只接受一个图标句柄，所以，程序也必须调用指向 WNDCLASSEX 结构的 RegisterClassEx 函数，为窗口类设定大的和小的图标。这里是一个加载两个图标图形的代码段，一个是边长为 16 个像素的正方形，另一个是边长为 32 个像素的正方形：

```
// Declare the icon in the .RC file
```

```
IconDemo            ICON                “res\\appicon.ico”
```

```
//In WinMain, initialize the WNDCLASSEX structure with image handles
```

```
static char          szAppName[]= “IconDemo”;
```

```
WNDCLASSEX wndclass;
```

```
wndclass.cbSize = sizeof (wndclass);
```

```
wndclass.hIcon      = LoadImage( nhInstance, szAppName,IMAGE_ICON,
```

```
                                32,32 LR_DEFAULTCOLOR );
```

```
wndclass.hIconSm = LoadImage(hInstance,szAppName,IMAGE_ICON,
```

```
                                16,16, LR_DEFAULTCOLOR );
```

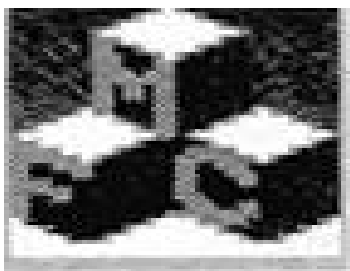
```
.
```

```
.
```

```
.
```

```
RegisterClassEx( &wndclass );
```

MFC 程序不需要担心任何上面这种情况。AppWizard 为项目提供两个图标，一个图标提供应用程序图标的功能，另一个表示应用程序创建的文档。RC 文件用 IDR_MAINFRAME 和 IDR_project_TYPE 标识图标资源，这里的 project 代表项目名。AppWizard 保存图标，如下所示，在项目的 Res 文件夹里以 project.ico 和 projectDoc.ico 表示：



project.ico



projectDoc.ico

AppWizard 自动生成正确加载同程序的其他资源在一起的应用程序图标的代码。要使用你设计的图标来代替常规的 AppWizard 图标，需关闭项目的工作空间，并从 File（文件）菜单里选择 New。双击在 File 选项卡里的 Icon File（图标文件）来打开图形编辑器，设计这个新图标，并把它保存在项目的 Res 文件夹里，重新覆盖已经存在的 project.ico 和 projectDoc.ico 文件。

如果你已经能够不使用 AppWizard 的帮助来写 MFC 程序，那么，加载应用程序的图标仍然是很简单的。如果主窗口类是来自 CFrameWnd，那么，使用在 MFC 的 Afres.h 文件里定义的 AFX_IDI_STD_FRAME 值来标识 ICO 文件。例如，在项目的 RC 文件中，在名为 AppIcon.ico 文件里保存的图标资源按如下方式进行标识：

```
#include "afxres.h"
```

```
.
```

```
.
```

```
.
```

```
AFX_IDI_STD_FRAME    ICON appicon.ico
```

如果窗口类来自 CMDIFrameWnd，那么使用这一行代替上面这一行。

```
AFX_IDI_STD_MDIFRAME ICON appicon.ico
```

如果图标文件同时包含一个大的和一个小的图形，那么 MFC 加载这些图形，并把它们同窗口框架正确关联，以便小图形显示在标题栏上，大图形显示在 About（关于）对话框里。如果你浏览 MFC 源代码，来学习有关这些工作的更多信息，那么你将看到 CWinApp::LoadIcon 并不像前面描述的那样调用 ::LoadImage。相

反，它调用使用 RT_GROUP_ICON 值的 ::FindResource 函数，来从图标资源里加载所有的图形，然后搜索图形资源，来查找最符合所需要尺寸的图形。CWinApp::LoadIcon 检索图形的方法是，使用 ::FindResource 函数返回的实例值来调用 ::LoadIcon 函数。

为 DiskPie1 创建图标

创建 DiskPie1 应用程序图标的过程是，在打开的项目里单击 Visual C++ Insert 菜单上的 Resource 命令，然后，双击 Resource Type 列表里的 Icon。图形编辑器默认打开一个边长为 32 个像素的正方形 16 色工作区域，Windows 在里面调用大的或标准的图标尺寸。编辑器在工作区域上面的 Device 组合框中，显示当前的标准 (32×32) 尺寸。组合框的下拉列表中只包含一个尺寸，这意味着，你当前正在操作的图标有一个图形，并且是边长为 32 个像素的正方形。

Visual C++ 图形编辑器可以创建有任何图形数目的图标资源，每一个都有不同的尺寸，或含有不同的颜色。要看其他可用的尺寸，可以按下 Insert 键、选择 Image 菜单的 New Device Image，或者单击 New Device Image 按钮：



上述方法都会打开图 4-17 里的 New Icon Image 对话框，在这里，显示了可用的图形尺寸，但是还没有与图标关联。如果你在列表里没有看到想要的尺寸，可单击 Custom 按钮，并指定一个新的图形尺寸。

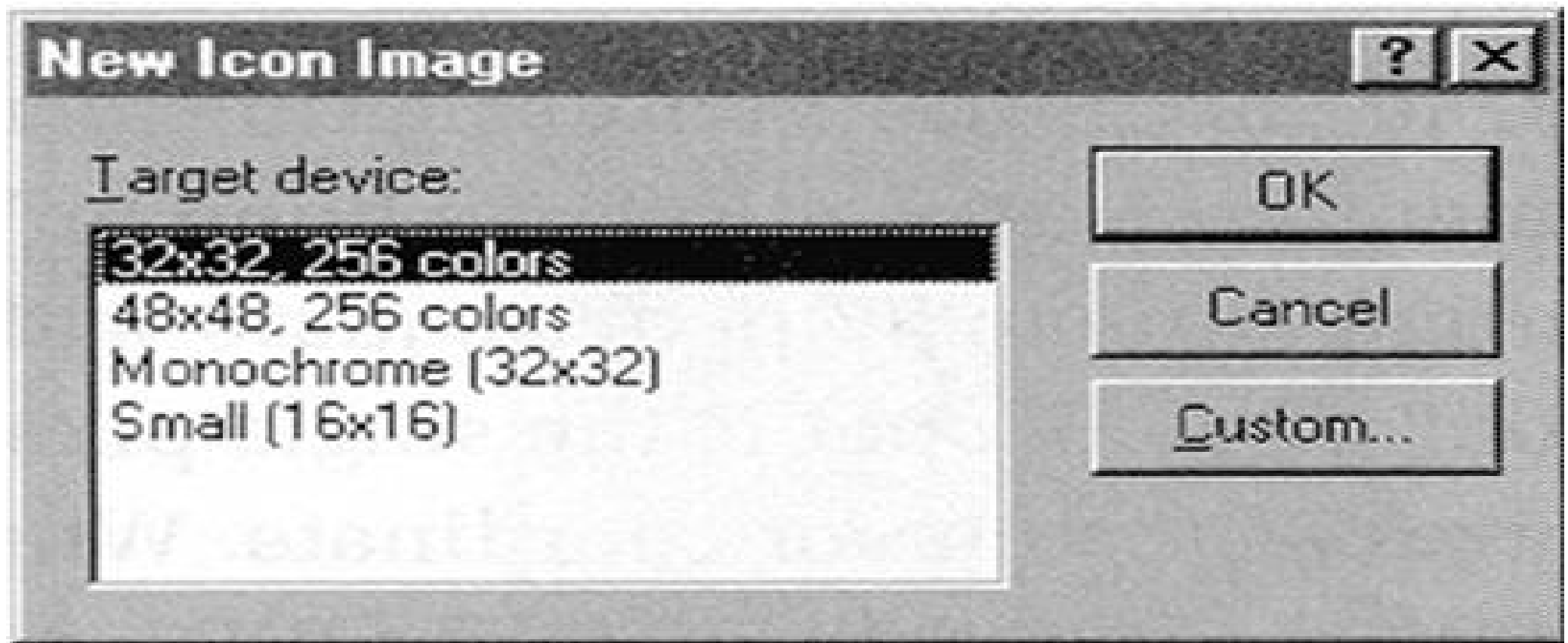
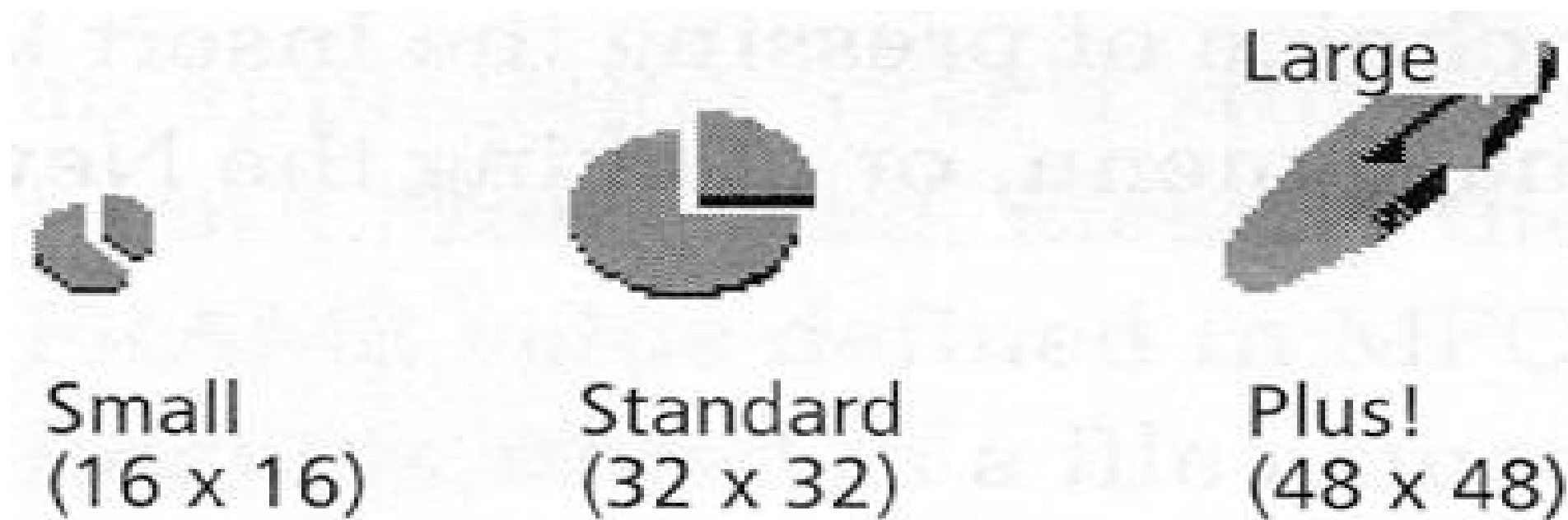


图 4-17 为图标选择新的图形

在 New Icon Image 对话框中，提供了在一个图标资源里包含多个图形的方法。在你画完这个标准的 32×32 大小的图形之后，从对话框里选择另一个尺寸，并重新开始。如果你想切换回原来的 32×32 大小的图形，在 Device 组合框里单击下拉箭头，并从打开的列表里选择 Standard(32×32)。当你从 New Icon Image 对话

框里选择了新的图形尺寸时，这个条目从对话框的列表里消失，并传递给组合框列表。换句话说，Device 组合框列出当前在图标里的图形尺寸，而 New Icon Image 对话框显示可添加到图标的尺寸。要从图标里移去当前图形，单击 Image 菜单上的 Delete Device Image（删除设备图形）。

DiskPie1 图标有三个图形尺寸，范围从边长为 16 个像素到 48 个像素的正方形：



一般情况下，程序图标里的图形是相同的图片，但是尺寸不同。我给出了三种不同的设计图形，来清楚地显示 Windows 从 DiskPie1 的资源数据里提取正确的图形，而不是按比例显示图形。双击编辑器工作区域的任何地方打开 Icon Properties

对话框，并把 IDR_MAINFRAME 标识符指定给资源，并保存工作。

鼠标光标

就 DiskPie1 这个程序来说，不需要设计一个定制的鼠标光标，但是，让我们花一些时间，来看一下 Visual C++是怎样做这些工作的。鼠标光标是一个边长为 32 个像素的单色位图，它有透明的背景和一个“热点”。热点是位图里的一个单独的像素，Windows 把它当做鼠标的坐标。例如，当一个程序接收到一个 WM_MOUSEMOVE 或一个 WM_LBUTTONDOWN 消息时，鼠标光标的 x 和 y 坐标保存在消息的 lParam 值里，表明这个在光标热点下面的像素坐标：

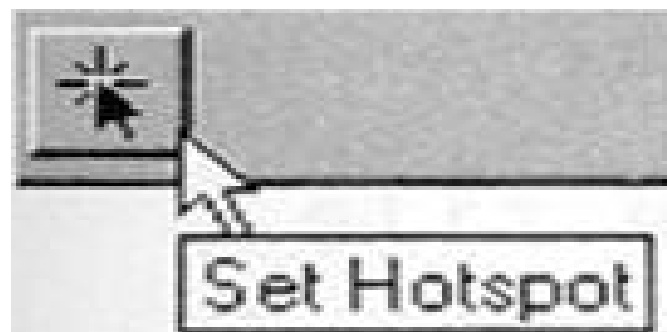
```
case WM_LBUTTONDOWN:
```

```
    x= LOWORD(lParam); //x-coordinate of mouse click
```

```
    y= HIWORD(lParam); //y-coordinate of mouse click
```

要创建一个鼠标光标，或者从 File 菜单里选择 New（新建）命令，然后双击在 File 选项卡里的 Cursor File（光标文件），或者，对于一个已经存在的项目，从 Insert 菜单里选择 Resource 命令。双击列表里的 Cursor，可以提供一个空白的方块，你可以在上面设计新光标。如果要从一个标准的 Windows 光标图形开始设计，那么，展开 Cursor 标题，并从 IDC_NODROP，IDC_POINTER 或

IDC_POINTER_COPY 里选择。编辑器显示一个边长为 32 个像素的工作区域，你将在上面画光标。如果你已经有了想使用的图形，但它是另一种形式，例如 32 × 32 的位图，那么，首先在图形编辑器里打开位图，并按下 Ctrl+C 将图形拷贝到剪贴板。然后打开一个新的光标，并按下 Ctrl+V 来粘贴位图到光标工作区域。位图里的颜色被转化为黑色或透明色，这取决于它们的亮度。如果你不喜欢这个结果，那么按下 Del 键删除光标图形。



当图形编辑器加载一个光标图形时，一个 Set Hotspot（设置热点）按钮显示在工作区域窗口的上面。你可以在图 4-18 里看到这个用坐标（0，13）标记的按钮，将热点放在图中啮齿动物的鼻子上。单击 Set Hotspot（设置热点）按钮，然后单击图形表格里想设置光标热点的点。完成时，从 File（文件）菜单里选择 Save，Visual C++ 将把这个图形保存成一个 CUR 文件，像下面这样添加定义到项目的 RC 文件里：

```
IDC_CURSOR1          COUSOR DISCARDABLE          "res\\cursor1.cur"
```

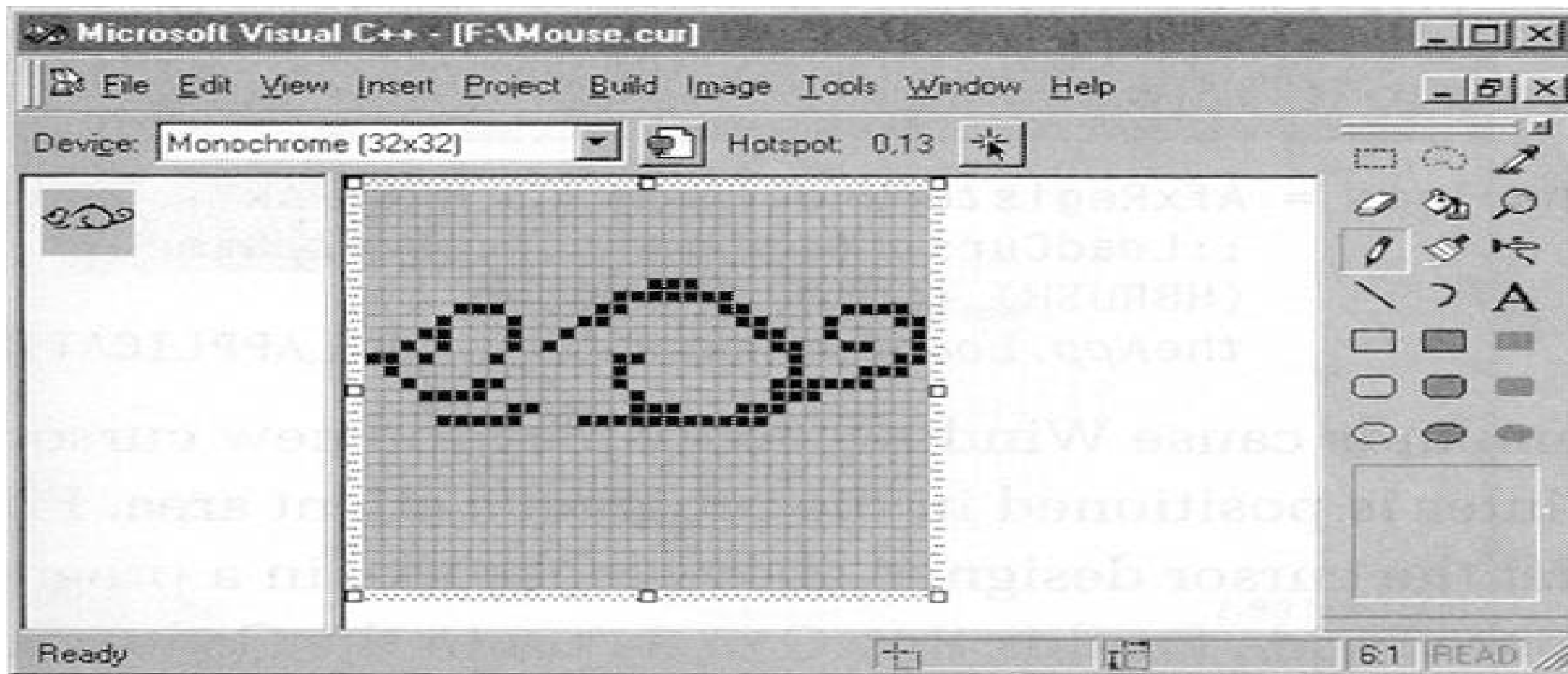


图 4-18 在 Visual C++ 图形编辑器中创建定制的鼠标光标

设置新的光标作为程序的默认光标，需要调用 `LoadCursor` API 函数。在一个 C 程序里，当为窗口初始化 `WNDCLASS` 或 `WNDCLASSEX` 结构时，这是典型的做法。这里是设置新的 `IDC_CURSOR1` 资源的方式：

```
WNDCLASSEX      wndclass;

wndclass.hCursor = LoadCursor( hInstance,

                               MAKEINTRESOURCE (IDC_CURSOR1) );

.

.

.

RegisterClassEx( &wndclass);
```

对于通过字符串名标识的光标资源，字符串是名字，而不是值，方法几乎和下面一样：

//Resource declaration in the RC file

```
MouseDemo      CURSOR              “mouse.cur”
```

//Load and set resource during initialization in the C source file

```
static char      szAppName[ ] = “MouseDemo”
```

```
WNDCLASSEX wndclass;
```

```
wndclass.hCursor = LoadCursor( hInstance, szAppName );
```

你可以在一个 MFC 程序里使用 `AfxRegisterWndClass` 全局函数做相同的事。下面的例子为 `AfxRegisterWndClass` 函数的最后一个参数使用一个常规图标，但是，真正的应用程序应该提供它自己的图标句柄：

```
CString wndclass = AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW,  
                                         ::LoadCursor( hInstance,szAppName ),  
                                         (HBRUSH) (COLOR_WINDOW + ),  
                                         theApp.LoadStandardIcon( IDI_APPLICATION ) );
```

利用这些命令，无论鼠标指针位于程序用户区域的任何地方，Windows 都会显示新的鼠标。图 4-19 显示了上面设计的光标在一个程序的外观（你可以从随书所附 CD 里的 `Code\Chaper.04\Mouse` 子文件夹里找到这个很小的 C 程序源代码）。

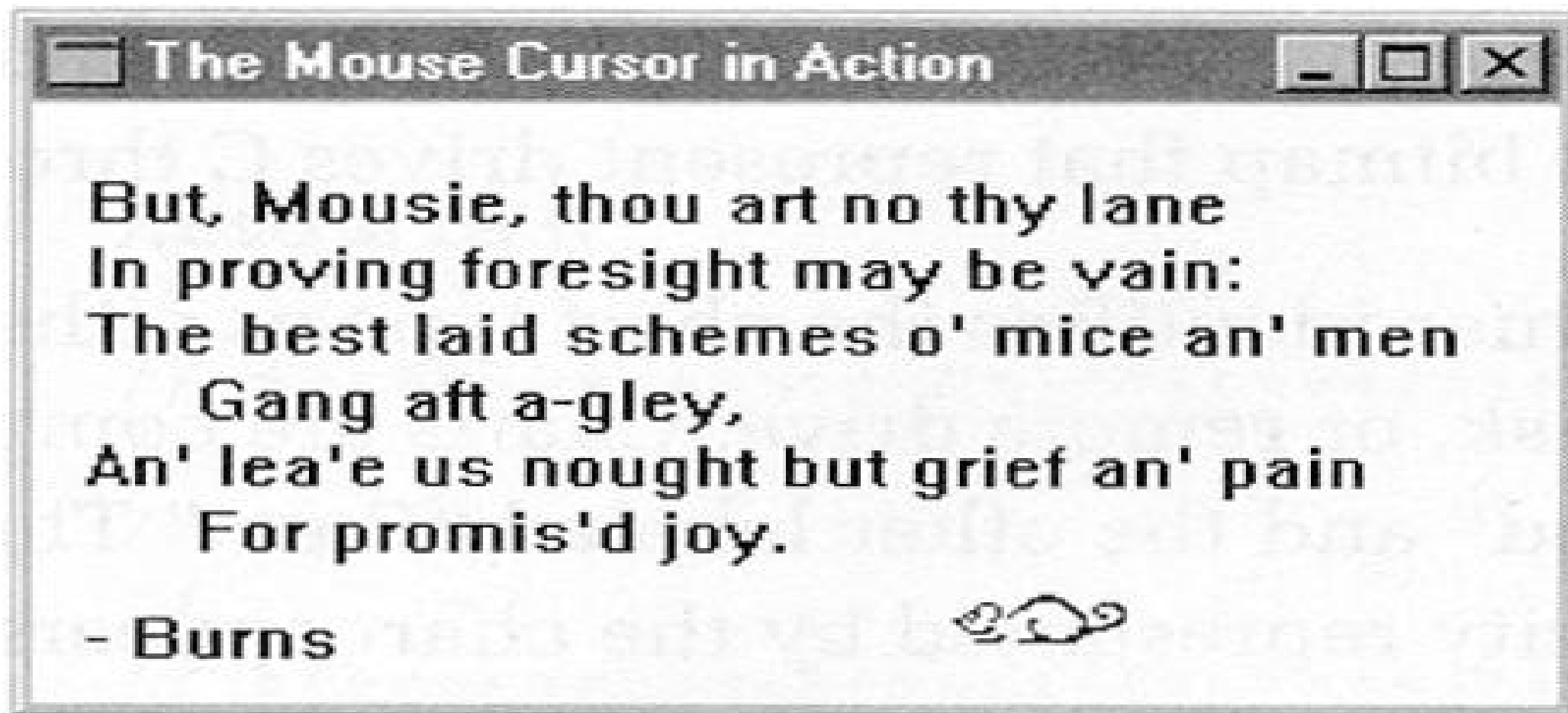


图 4-19 新的鼠标光标在程序中出现

光标就讲这些吧，我们再来谈谈 DiskPie1 程序。

向 DiskPie1 添加代码

现在，DiskPie1 有菜单、加速键、工具栏、状态栏和图标，这些是需要的所有资源。现在，可以添加代码使它执行。如果你在本章里依照 DiskPie1 的生成过程进行，那么，你将可以识别在图 4-20 里的用户界面元素。应用程序图标、菜单、工具栏和状态栏是在前面几节里用 Visual C++ 资源编辑器设计的。

在创建主窗口的过程中，DiskPie1 将找到所有连接的固定磁盘，包括远程驱动器和 RAM 驱动器，每找到一个它，都要做下面这些工作：

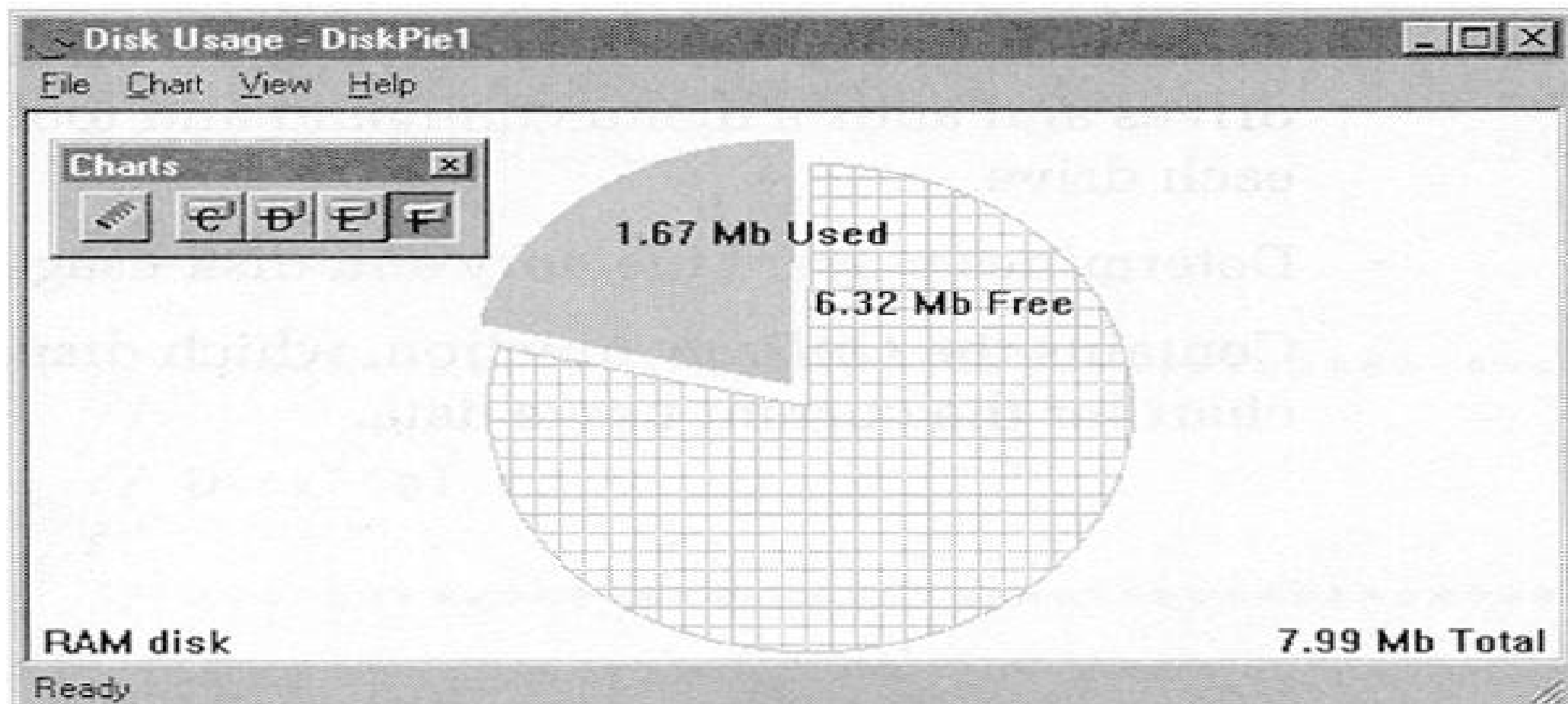


图 4-20 DiskPie1 程序，显示 8MB RAM 磁盘上的可用空间

- 向 Chart 菜单添加一个叫做 Disk x 的命令，这里 x 是磁盘的指定字母。
- 在工具栏里添加一个按钮，从表示驱动器 C 到 Z 的工具栏位图中，从

24 个图像中选择按钮的图形。

左下角的文本指出饼图的类型，或者是内存，或者是本地固定的驱动器、RAM 驱动器或远程驱动器。饼图由两部分组成，分别标记为“Used”和“Free”。内存的总数量或磁盘的容量通过右下角的图表表示。

虽然 DiskPie1 是一个相当小的程序，但是，它使用通用的 C++ 方式进行编译，将类分开放在分开的资源文件里。完整的程序列在后面，在随书附带的 CD 里的 Code\Chapter.04\DiskPie1 子文件夹里，可以读取这些程序。如果你已经跟随本章里的步骤，为新的 DiskPie1 项目创建了资源，那么，你可能想完成这个项目，并为这个程序建立你自己的版本。在这种情况下，拷贝 H 和 CPP 资源文件到你的项目文件夹，并使用 Project（项目）菜单里的 Add To Project 命令添加 CPP 文件。此代码要求有一个 About（关于）对话框，所以，使用文本编辑器从 CD 上的 DiskPie1.rc 文件里拷贝和粘贴脚本到你自己的 RC 文件。要记住，需除掉工具栏脚本，使它像“为 DiskPie1 创建工具栏”一节中给出的那样。

下面的目录描述了 DiskPie1 源文件，可以帮助你理解在列表 4-1 中的代码。

源文件	说明
DiskPie	InitInstance 函数，显示 About 对话
MainFrm1	创建主窗口，确定可用的驱动器，并为每个驱动器添加菜单命令和工具栏按钮
DiskDoc	确定当前内存和磁盘的使用状况
DiskView	包含 OnDraw 函数，它为当前的使用状况数据显示饼图

列表 4-1 DiskaPie1 源文件

```
Resource.h

// *****

//

// Resource.h

//

// *****

#define IDD_ABOUTBOX                100

#define IDR_MAINFRAME               101

#define IDM_MEMORY                  130

#define IDM_DISK_C                  131

#define IDM_DISK_D                  132
```

#define IDM_DISK_E	133
#define IDM_DISK_F	134
#define IDM_DISK_G	135
#define IDM_DISK_H	136
#define IDM_DISK_I	137
#define IDM_DISK_J	138
#define IDM_DISK_K	139
#define IDM_DISK_L	140
#define IDM_DISK_M	141
#define IDM_DISK_N	142
#define IDM_DISK_O	143
#define IDM_DISK_P	144

```
#define IDM_DISK_Q          145

#define IDM_DISK_R          146

#define IDM_DISK_S          147

#define IDM_DISK_T          148

#define IDM_DISK_U          149

#define IDM_DISK_V          150

#define IDM_DISK_W          151

#define IDM_DISK_X          152

#define IDM_DISK_Y          153

#define IDM_DISK_Z          154
```

DiskPie1.rc

```
// *****
```

```
//

// DiskPie1.rc

//

// *****

#include "resource.h"

#include "afxres.h"

IDR_MAINFRAME    ICON        "res\\diskpie1.ico"

IDR_MAINFRAME    BITMAP      "res\\toolbar.bmp"

IDR_MAINFRAME MENU

BEGIN

    POPUP "&File"

    BEGIN
```

```
        MENUITEM "E&xit",          ID_APP_EXIT

END

POPUP "&Chart"

BEGIN

        MENUITEM "&Memory\tCtrl+M",  IDM_MEMORY

        MENUITEM SEPARATOR

END

POPUP "&View"

BEGIN

        MENUITEM "&Toolbar",        ID_VIEW_TOOLBAR

        MENUITEM "&Status Bar",    ID_VIEW_STATUS_BAR

END
```

```
POPUP "&Help"
```

```
BEGIN
```

```
    MENUITEM "&About DiskPie1...",  ID_APP_ABOUT
```

```
END
```

```
END
```

```
IDR_MAINFRAME ACCELERATORS
```

```
BEGIN
```

```
    "M",          IDM_MEMORY,          VIRTKEY, CONTROL, NOINVERT
```

```
    "C",          IDM_DISK_C,          VIRTKEY
```

```
    "D",          IDM_DISK_D,          VIRTKEY
```

```
    "E",          IDM_DISK_E,          VIRTKEY
```


"F",	IDM_DISK_F,	VIRTKEY
"G",	IDM_DISK_G,	VIRTKEY
"H",	IDM_DISK_H,	VIRTKEY
"I",	IDM_DISK_I,	VIRTKEY
"J",	IDM_DISK_J,	VIRTKEY
"K",	IDM_DISK_K,	VIRTKEY
"L",	IDM_DISK_L,	VIRTKEY
"M",	IDM_DISK_M,	VIRTKEY
"N",	IDM_DISK_N,	VIRTKEY
"O",	IDM_DISK_O,	VIRTKEY
"P",	IDM_DISK_P,	VIRTKEY
"Q",	IDM_DISK_Q,	VIRTKEY

"R",	IDM_DISK_R,	VIRTKEY
"S",	IDM_DISK_S,	VIRTKEY
"T",	IDM_DISK_T,	VIRTKEY
"U",	IDM_DISK_U,	VIRTKEY
"V",	IDM_DISK_V,	VIRTKEY
"W",	IDM_DISK_W,	VIRTKEY
"X",	IDM_DISK_X,	VIRTKEY
"Y",	IDM_DISK_Y,	VIRTKEY
"Z",	IDM_DISK_Z,	VIRTKEY

END

IDR_MAINFRAME TOOLBAR 16, 15

BEGIN

```
        BUTTON            IDM_MEMORY

END

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 240, 65

STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU

CAPTION "About DiskPie1"

FONT 8, "MS Sans Serif"

BEGIN

ICON                IDR_MAINFRAME,IDC_STATIC,10,22,20,20

LTEXT                "DiskPie1 Version 1.0",IDC_STATIC,45,10,115,8

LTEXT                ""Microsoft Visual C++ Programmer's Guide"",IDC_STATIC,45,26,140,8
```

```
LTEXT          "Copyright \251 1998, Beck Zaratian",IDC_STATIC,45,42,115,8
```

```
DEFPUSHBUTTON  "OK",IDOK,195,10,35,40,WS_GROUP
```

```
END
```

```
STRINGTABLE
```

```
BEGIN
```

```
IDR_MAINFRAME          "DiskPie1\nDisk Usage\n\n\n\n\n\n\n"
```

```
IDM_MEMORY              "Memory usage\nMemory"
```

```
IDM_DISK_C              "Usage for drive C\nDrive C"
```

```
IDM_DISK_D              "Usage for drive D\nDrive D"
```

```
IDM_DISK_E              "Usage for drive E\nDrive E"
```

```
IDM_DISK_F              "Usage for drive F\nDrive F"
```

```
IDM_DISK_G              "Usage for drive G\nDrive G"
```

IDM_DISK_H	"Usage for drive H\nDrive H"
IDM_DISK_I	"Usage for drive I\nDrive I"
IDM_DISK_J	"Usage for drive J\nDrive J"
IDM_DISK_K	"Usage for drive K\nDrive K"
IDM_DISK_L	"Usage for drive L\nDrive L"
IDM_DISK_M	"Usage for drive M\nDrive M"
IDM_DISK_N	"Usage for drive N\nDrive N"
IDM_DISK_O	"Usage for drive O\nDrive O"
IDM_DISK_P	"Usage for drive P\nDrive P"
IDM_DISK_Q	"Usage for drive Q\nDrive Q"
IDM_DISK_R	"Usage for drive R\nDrive R"
IDM_DISK_S	"Usage for drive S\nDrive S"

IDM_DISK_T	"Usage for drive T\nDrive T"
IDM_DISK_U	"Usage for drive U\nDrive U"
IDM_DISK_V	"Usage for drive V\nDrive V"
IDM_DISK_W	"Usage for drive W\nDrive W"
IDM_DISK_X	"Usage for drive X\nDrive X"
IDM_DISK_Y	"Usage for drive Y\nDrive Y"
IDM_DISK_Z	"Usage for drive Z\nDrive Z"
AFX_IDS_IDLEMESSAGE	"Ready"
AFX_IDS_SCSIZE	"Change the window size"
AFX_IDS_SCMOVE	"Change the window position"
AFX_IDS_SCMINIMIZE	"Reduce the window to an icon"
AFX_IDS_SCMAXIMIZE	"Enlarge the window to full size"

AFX_IDS_SCCLOSE

"Close the DiskPie1 application"

AFX_IDS_SCRESTORE

"Restore the window to normal size"

END

DiskPie1.h

```
// *****
```

```
//
```

```
// DiskPie.h
```

```
//
```

```
// *****
```

```
class CDiskPieApp : public CWinApp
```

```
{
```

```
public:
```

```
virtual BOOL InitInstance();

afx_msg void OnAppAbout();

DECLARE_MESSAGE_MAP()

};
```

DiskPie.cpp

```
// *****

//

// DiskPie.cpp

//

// *****

#define VC_EXTRALEAN

#include <afxwin.h>
```



```
#include <afxext.h>

#include "resource.h"

#include "DiskPie.h"

#include "DiskDoc.h"

#include "MainFrm1.h"

#include "DiskView.h"

CDiskPieApp theApp;

BEGIN_MESSAGE_MAP (CDiskPieApp, CWinApp)

    ON_COMMAND (ID_APP_ABOUT, OnAppAbout)

END_MESSAGE_MAP ()

BOOL CDiskPieApp::InitInstance()

{
```

```
CSingleDocTemplate* pDocTemplate;

pDocTemplate = new CSingleDocTemplate( IDR_MAINFRAME,

                                     RUNTIME_CLASS (CDiskDoc),

                                     RUNTIME_CLASS (CMainFrame),

                                     RUNTIME_CLASS (CDiskView));

AddDocTemplate(pDocTemplate);

CCommandLineInfo cmdInfo;

if (!ProcessShellCommand(cmdInfo))

    return FALSE;

return TRUE;

}
```

```
class CAboutDlg : public CDialog
```

```
{  
  
public:  
  
    CAboutDlg();  
  
    enum { IDD = IDD_ABOUTBOX };  
  
};  
  
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)  
  
{  
  
}  
  
void CDiskPieApp::OnAppAbout()  
  
{  
  
    CAboutDlg aboutDlg;  
  
    aboutDlg.DoModal();  
}
```

```
}
```

DiskDoc.h

```
// *****
```

```
//
```

```
// DiskDoc.h
```

```
//
```

```
// *****
```

```
class CDiskDoc : public CDocument
```

```
{
```

```
    DECLARE_DYNCREATE (CDiskDoc)
```

```
public:
```

```
    static int      iDriveType[24];
```

```
static      DWORD dwTotal, dwFree;

static int   iChartType;

static UINT  nCurrent;

int         GetDriveCount();

void        GetMemoryUsage();

void        GetDiskUsage( UINT nID );

};
```

DiskDoc.cpp

```
// *****

//

// DiskDoc.cpp

//
```

```
// *****

#define VC_EXTRALEAN

#include <afxwin.h>

#include "resource.h"

#include "DiskPie.h"

#include "DiskDoc.h"

#define    PIE_MEMORY        0;

IMPLEMENT_DYNCREATE (CDiskDoc, CDocument)

int          CDiskDoc::iDriveType[24],

int          CDiskDoc::iChartType = PIE_MEMORY;

UINT         CDiskDoc::nCurrent = IDM_MEMORY;

DWORD        CDiskDoc::dwTotal, CDiskDoc::dwFree;
```

```
void CDiskDoc::GetMemoryUsage ()
```

```
{
```

```
    MEMORYSTATUS ms;
```

```
    ::GlobalMemoryStatus( &ms );
```

```
    dwTotal= ms.dwTotalPhys;
```

```
    dwFree = ms.dwAvailPhys;
```

```
    iChartType = PIE_MEMORY;
```

```
    nCurrent    = IDM_MEMORY;
```

```
}
```

```
void CDiskDoc::GetDiskUsage( UINT nID )
```

```
{
```

```
    char          szDrive[] = "x:\\\\0";
```

```
DWORD    dwSectsPerClust, dwBytesPerSect, dwFreeClusts, dwTotalClusts;
```

```
DWORD    dwBytesPerClust;
```

```
GetDriveCount();
```

```
if (iDriveType[nID - IDM_DISK_C] == DRIVE_FIXED    ||
```

```
    iDriveType[nID - IDM_DISK_C] == DRIVE_REMOTE    ||
```

```
    iDriveType[nID - IDM_DISK_C] == DRIVE_RAMDISK)
```

```
{
```

```
    szDrive[0] = (char) (nID - IDM_DISK_C) + 'C';
```

```
    if (::GetDiskFreeSpace( szDrive, &dwSectsPerClust, &dwBytesPerSect,
```

```
        &dwFreeClusts, &dwTotalClusts ))
```

```
{
```

```
    dwBytesPerClust = dwSectsPerClust * dwBytesPerSect;
```



```
dwTotal          = dwBytesPerClust * dwTotalClusts;
```

```
dwFree           = dwBytesPerClust * dwFreeClusts;
```

```
iChartType = iDriveType[nID - IDM_DISK_C];
```

```
nCurrent      = nID;
```

```
}
```

```
}
```

```
}
```

```
int CDiskDoc::GetDriveCount()
```

```
{
```

```
int    i, cDrives = 0;
```

```
char szDrive[] = "x:\\\\0";
```

```
for (szDrive[0]='C'; szDrive[0] <= 'Z'; szDrive[0]++)
```

```
{  
  
    i = (int) (szDrive[0] - 'C');  
  
    iDriveType[i] = ::GetDriveType( szDrive );  
  
    if (iDriveType[i] == DRIVE_FIXED    ||  
  
        iDriveType[i] == DRIVE_REMOTE  ||  
  
        iDriveType[i] == DRIVE_RAMDISK)  
    {  
  
        cDrives++;  
  
    }  
  
}  
  
return cDrives;  
  
}
```

DiskView.h

```
// *****
```

```
//
```

```
// DiskView.h
```

```
//
```

```
// *****
```

```
class CDiskView : public CView
```

```
{
```

```
    DECLARE_DYNCREATE (CDiskView)
```

```
private:
```

```
static    COLORREF rgbColor[2];

static    CString   strType[];

void      GetLabel( CString* str, double e, PCSTR strTail );
```

protected:

```
virtual   void      OnDraw( CDC* pDC );

afx_msg void      OnMemoryUpdate( CCmdUI* pCmdUI );

afx_msg void      OnDiskUpdate( CCmdUI* pCmdUI );

DECLARE_MESSAGE_MAP ()
```

```
};
```

DiskView.cpp

```
// *****
```

```
//
```

```
// DiskView.cpp

//

// *****

#define VC_EXTRALEAN

#include <afxwin.h>

#include "resource.h"

#include "DiskView.h"

#include "DiskDoc.h"

#include "math.h"

#define PI                3.141592654

#define RADIUS            900

#define SLICE_OFFSET      12
```

```
#define MEM_COLOR      RGB (0, 255, 255)    // Cyan

#define DISK_COLORRGB (0, 255, 0)          // Green

COLORREF CDiskView::rgbColor[2] = { MEM_COLOR, DISK_COLOR };

CString  CDiskView::strType[]      = { " Memory",

                                         " Fixed disk",

                                         " Remote drive",

                                         " RAM disk" };

IMPLEMENT_DYNCREATE (CDiskView, CView)

BEGIN_MESSAGE_MAP (CDiskView, CView)

    ON_UPDATE_COMMAND_UI (IDM_MEMORY, OnMemoryUpdate)

    ON_UPDATE_COMMAND_UI_RANGE (IDM_DISK_C, IDM_DISK_Z, OnDiskUpdate)

END_MESSAGE_MAP()
```

```
////////////////////////////////////
```

```
// Paint the pie chart
```

```
void CDiskView::OnDraw( CDC* pDC )
```

```
{
```

```
    Cpen        pen;
```

```
    CBrush      brush;
```

```
    CRect       rect;
```

```
    CString     str;
```

```
    int         x, y, i, iColor;
```

```
    double      dUseSweep, dFreeSweep;
```

```
    // Color is cyan for memory chart, green for drive charts
```

```
    iColor = (CDiskDoc::nCurrent == IDM_MEMORY) ? 0 : 1;
```

```
// Set coordinate system so that origin is at center of client area
```

```
GetClientRect( rect );
```

```
pDC->SetMapMode( MM_ISOTROPIC );
```

```
pDC->SetWindowExt( RADIUS+100, RADIUS+100 );
```

```
pDC->SetViewportExt( rect.right/2, -rect.bottom/2 );
```

```
pDC->SetViewportOrg( rect.right/2, rect.bottom/2 );
```

```
// Create solid brush of current color to paint "Used" pie slice
```

```
pen.CreatePen( PS_SOLID, 1, rgbColor[iColor] );
```

```
brush.CreateSolidBrush( rgbColor[iColor] );
```

```
pDC->SelectObject( &pen );
```

```
pDC->SelectObject( &brush );
```

```
// Sweep angles in radians for "Free" and "Used" pie slices
```



```
dFreeSweep = (double) (PI*2 * CDiskDoc::dwFree/CDiskDoc::dwTotal);

dUseSweep   = (double) PI*2 - dFreeSweep;

// Paint "Used" slice first, sweeping counter-clockwise from due north

x = -(int) (sin( dUseSweep ) * RADIUS);

y =   (int) (cos( dUseSweep ) * RADIUS);

pDC->Pie( -RADIUS, RADIUS, RADIUS, -RADIUS, 0, RADIUS, x, y );

// Create hatched brush of current color to paint "Free" pie slice

pDC->SelectStockObject( WHITE_BRUSH );

pDC->SelectStockObject( BLACK_PEN );

brush.DeleteObject();

pen.DeleteObject();

pen.CreatePen( PS_SOLID, 1, rgbColor[iColor] );
```

```
brush.CreateHatchBrush( HS_CROSS, rgbColor[iColor] );

pDC->SelectObject( &pen );

pDC->SelectObject( &brush );

// Compute new center for "Free" slice, slightly offset from

// original center, then paint the "Free" slice

x = (int) (sin( PI - dUseSweep/2 ) * SLICE_OFFSET);

y = -(int) (cos( PI - dUseSweep/2 ) * SLICE_OFFSET);

pDC->OffsetWindowOrg( x, y );

pDC->OffsetViewportOrg( x, y );

x = (int) (sin( dFreeSweep ) * RADIUS);

y = (int) (cos( dFreeSweep ) * RADIUS);

if(abs (x) > 4)
```

```
pDC->Pie( -RADIUS, RADIUS, RADIUS, -RADIUS, x, y, 0, RADIUS );

pDC->SelectStockObject( BLACK_PEN );

pDC->SelectStockObject( WHITE_BRUSH );

pDC->SetBkMode( TRANSPARENT );

pen.DeleteObject();

brush.DeleteObject();

// Label "Free" slice

GetLabel( &str, CDiskDoc::dwFree, "Free" );

pDC->TextOut( 10, RADIUS/2, str );

// Label "Used" slice, making sure it doesn't occur near the "Free" label

x = -(int) (sin( dUseSweep/2 ) * RADIUS);

y = (int) (cos( dUseSweep/2 ) * RADIUS);
```

```
if ( y > 0    &&    (y - RADIUS/2) < 25)

{

    x = -(RADIUS - 10);

    y = 0;

}

GetLabel( &str, (CDiskDoc::dwTotal - CDiskDoc::dwFree), "Used" );

pDC->TextOut( x, y, str );


// Restore original mapping mode so we can use DrawText function

pDC->SetMapMode( MM_TEXT );

pDC->SetWindowExt( rect.right, rect.bottom );

pDC->SetViewportExt( rect.right, rect.bottom );

pDC->SetViewportOrg( 0, 0 );
```

```
pDC->SetWindowOrg( 0, 0 );

// Write "Total" at bottom right corner of window


GetLabel( &str, CDiskDoc::dwTotal, "Total " );

pDC->DrawText( str, rect, DT_SINGLELINE | DT_BOTTOM | DT_RIGHT );

// Write device type at bottom left corner of window

i = 0;

if (CDiskDoc::iChartType == DRIVE_FIXED)

    i = 1;

if (CDiskDoc::iChartType == DRIVE_REMOTE)

    i = 2;

if (CDiskDoc::iChartType == DRIVE_RAMDISK)
```

```
        i = 3;

        pDC->DrawText( strType[i], rect, DT_SINGLELINE | DT_BOTTOM | DT_LEFT );

    }

void CDiskView::GetLabel( CString* str, double d, PCSTR strTail )

{

    charch = 'K';                                // 'K' for kilobytes

    d /= 1024;

    if (d > 1024)                                // If amount is greater than

    {                                              // 1024 kilobytes, divide

        d /= 1024;                                // again by 1024 to convert

        ch = 'M';                                // to megabytes

    }
```

```
        str->Format( "%.2f %c%s%s", d, ch, "b ", strTail );
    }

    //////////////////////////////////////

// Make sure menu check marks and toolbar buttons are in sync

void CDiskView::OnMemoryUpdate( CCmdUI* pCmdUI )
{
    pCmdUI->SetCheck( CDiskDoc::nCurrent == IDM_MEMORY );
}

void CDiskView::OnDiskUpdate( CCmdUI* pCmdUI )
{
    pCmdUI->SetCheck( CDiskDoc::nCurrent == pCmdUI->m_nID );
}
```

MainFrm1.h

```
// *****  
  
//  
  
// MainFrm1.h  
  
//  
  
// *****  
  
class CMainFrame : public CFrameWnd  
{  
  
    DECLARE_DYNCREATE (CMainFrame)  
  
private:  
  
    CToolBar toolbar;  
  
    CStatusBar  statusbar;
```



```
CDiskDocdiskdoc;
```

protected:

```
afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
```

```
afx_msg void OnMemory ();
```

```
afx_msg void OnDisk( UINT nID );
```

```
afx_msg void OnSetFocus( CWnd* );
```

```
DECLARE_MESSAGE_MAP()
```

```
};
```

MainFrm1.cpp

```
// *****
```

```
//
```

```
// MainFrm1.cpp
```

```
//
```

```
// *****
```

```
#define VC_EXTRALEAN
```

```
#include <afxwin.h>
```

```
#include <afxext.h>
```

```
#include <afxcmn.h>
```

```
#include "resource.h"
```

```
#include "DiskPie.h"
```

```
#include "DiskDoc.h"
```

```
#include "MainFrm1.h"
```

```
#include "DiskView.h"
```

```
IMPLEMENT_DYNCREATE (CMainFrame, CFrameWnd)
```

```
BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

    ON_WM_CREATE ()

    ON_WM_SETFOCUS ()

    ON_COMMAND (IDM_MEMORY, OnMemory)

    ON_COMMAND_RANGE (IDM_DISK_C, IDM_DISK_Z, OnDisk)

END_MESSAGE_MAP ()

////////////////////////////////////

// Create main window, toolbar, and status bar

int CMainFrame::OnCreate( LPCTSTR lpCreateStruct )

{

    int          i, j;

    char          szMenu[] = "Disk &x\\tx\\0";
```

```
CMenu*    pmenu;

static const UINT indicator    = ID_SEPARATOR;

static const UINT nButtons[] = { IDM_MEMORY, ID_SEPARATOR,

    IDM_DISK_C, IDM_DISK_D, IDM_DISK_E, IDM_DISK_F,

    IDM_DISK_G, IDM_DISK_H, IDM_DISK_I, IDM_DISK_J,

    IDM_DISK_K, IDM_DISK_L, IDM_DISK_M, IDM_DISK_N,

    IDM_DISK_O, IDM_DISK_P, IDM_DISK_Q, IDM_DISK_R,

    IDM_DISK_S, IDM_DISK_T, IDM_DISK_U, IDM_DISK_V,

    IDM_DISK_W, IDM_DISK_X, IDM_DISK_Y, IDM_DISK_Z };

if (CFrameWnd::OnCreate( lpCreateStruct ) == -1)

    return -1;
```

```
statusbar.Create( this );

statusbar.SetIndicators( &indicator, 1 );

toolbar.Create( this );

toolbar.SetWindowText( "Charts" );

toolbar.LoadToolBar( IDR_MAINFRAME );

toolbar.SetBarStyle( toolbar.GetBarStyle() |

                    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC );

toolbar.SetButtons( nButtons, diskdoc.GetDriveCount() + 2 );

toolbar.SetButtonStyle( 0, TBBS_CHECKGROUP );

pmenu = GetMenu()->GetSubMenu( 1 );

for (i=0, j=2; i < 24; i++)

{
```

```
if(CDiskDoc::iDriveType[i] == DRIVE_FIXED    ||

    CDiskDoc::iDriveType[i] == DRIVE_REMOTE  ||

    CDiskDoc::iDriveType[i] == DRIVE_RAMDISK)

{

    szMenu[6] = 'C' + (char) i;

    szMenu[8] = 'C' + (char) i;

    pmenu->InsertMenu( 0xFFFF, MF_BYPOSITION, IDM_DISK_C+i, szMenu );

    toolbar.SetButtonInfo( j++, IDM_DISK_C+i, TBBS_CHECKGROUP, i+1 );

}

}

toolbar.EnableDocking( CBR_S_ALIGN_ANY );

EnableDocking( CBR_S_ALIGN_ANY );
```

```
        DockControlBar( &toolbar );

        return 0;

}

////////////////////////////////////

// Respond to menu/toolbar/accelerator commands

void CMainFrame::OnMemory ()

{

    diskdoc.GetMemoryUsage ();

    Invalidate ();

}

void CMainFrame::OnDisk( UINT nID )

{
```

```
        diskdoc.GetDiskUsage( nID );

        Invalidate ();

    }

    //////////////////////////////////////

    // When focus regained, refresh display in case data has changed

void CMainFrame::OnSetFocus( CWnd* )

{

    if(CDiskDoc::nCurrent == IDM_MEMORY)

        diskdoc.GetMemoryUsage();

    else

        diskdoc.GetDiskUsage( CDiskDoc::nCurrent );

}
```


如果从程序的逻辑流来看，那么，你将发现，最重要的步骤在 `MainFrm1.cpp` 模块里。不管什么时候，只要 Windows 发送一条 `WM_SETFOCUS` 消息，通知获得输入焦点的主窗口，便将调用 `CMainFrame::OnSetFocus` 函数。当 `DiskPie1` 首先启动时，或无论何时，只要用户从其他程序切换到 `DiskPie1` 时，将发生这个事件。这样，这个函数提供两种用途的服务。它使 `CMainFrame::OnCreate` 免于在程序开始启动时调用 `DiskDoc::GetMemoryUsage` 来初始化数据，并且，它也保证当用户运行其他程序，或删除一个文件时，当 `DiskPie1` 重新获得焦点时，自动重画当前的饼图，来反映新的状况。

通过给 `CMainFrame::OnCreate` 函数提供一个连接的驱动器列表，函数添加 `Chart` 菜单命令，例如为每个连接的驱动器添加 `DISK C` 和 `DISK D` 这样的命令。它也为每一个驱动器添加一个工具栏按钮，根据驱动器指定的字母选择工具栏位图的合适部分。例如，驱动器 `D` 的按钮是用位图的 16×15 这部分画的，其中包含这个磁盘驱动器图像和字母 `D`（含有 25 个位图块的完整工具栏位图显示在图 4-15 中）。

用户可以请求内存状况饼图或磁盘驱动器的饼图，你可以通过下面步骤完成：

- 从 `Chart` 菜单选择一个命令。
- 单击工具栏按钮。

- 要查看内存，按下 Ctrl+M，或要查看磁盘驱动器，按下 Ctrl 和从 C 到 Z 对应于驱动器的字母。

这些事件是通过 CMainFrame::OnMemory 和 CMainFrame::OnDisk 函数处理的，它通过类的消息映射接收控制：

```
BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

.

.

.

    ON_COMMAND (idm_memory, OnMemory)

    ON_COMMAND_RANGE (IDM_DISK_C, IDM_DISK_Z, OnDisk)

END_MESSAGE_MAP ()
```

你可能回想起，在本章前面创建 DiskPie1 加速键时的情形，我们必须保证从 IDM_DISK_C 到 IDM_DISK_Z 的标识符值是连续排列的。上面的消息映射显示了为什么要这样。因为标识符有连续的值，程序可以使用 MFC 的 ON_COMMAND_RANGE 宏来为任何磁盘驱动器发送请求给 OnDisk 函数。根据

作为参数传递的磁盘驱动器标识符，OnDisk 决定用户要求用饼图显示的磁盘。OnMemory 不要求 ON_COMMAND_RANGE，因为只有一个内存饼图。

决定内存和磁盘使用状况的机制是在 DiskDoc.cpp 模块里使用两个函数。这两个函数 CDiskDoc::GetMemoryUsage 和 CDiskDoc::GetDiskUsage 使用了相似的逻辑结构。它们通过调用 GlobalMemoryStatus 和 GetDiskFreeSpace API 函数，从系统中找到它们要求的信息，然后，使用这个信息来决定 dwTotal 和 dwFree 内存变量的值，它包含当前饼图的总字节数和未使用的字节数。变量中的数据不管是表示内存，还是磁盘空间，对于画图程序来说，都没有什么区别。在决定了当前的使用状况数据之后，OnMemory 和 OnDisk 函数调用 Invalidate 来强制显示一个新的饼图。

现在，应该介绍 DiskView.cpp 文件里 CDiskView::OnDraw 函数了。这个函数使用了公共的 dwTotal 和 dwFree 值，来决定饼图的两个分块所占用的角度：

```
//Sweep angles in radians for “Free”and “Used” pie slices
```

```
dFreeSweep = (double) (PI*2 * CDiskDoc::dwFree/CDiskDoc::dwTotal);
```

```
dUseSweep = (double) PI*2 - dFreeSweep;
```

OnDraw 先画出 “Used（使用的）”的那一部分，通过 dUseSweep 角度，来决定从 12 点位置反时针转过的角度。“Free（空闲）”块是剩余的那部分。OnDraw 为

每一块关联一个标签，并显示这个饼图。如果需要画另一个饼图，就必须重新开始整个过程。

一个饼图快速提供了目前的使用状况。当 `DiskPie1` 获得焦点时，刷新饼图的唯一方便的方法是按下对应于这个饼图的加速键。在 Windows 这样的抢占式多任务系统里，内存使用状况的变化特别快，每一微秒都在发生变化。如果你在 Visual C++ 中，在后台编译一个项目时，连续按下 `Ctrl+M` 键，那么，你可以看到快速分配和释放物理内存的效果。添加 `OnTimer` 函数可以定期更新显示。

更严重的是，`DiskPie1` 无法解决驱动器连接的动态变化问题。我们已经看到，`CMainFrame::OnCreate` 函数怎样添加菜单命令，以及如何为连接到系统上的磁盘驱动器添加工具栏按钮，包括任何通过网络提供的远程驱动器。然后，在程序的生存期内，菜单和工具栏保持不变，即使用户后来可能添加或卸载一个网络驱动器，或由于服务器端出现问题，而使一个连接消失。然而，这并不会致命，`DiskPie1` 仍然正常工作，因为 `CDiskDoc::GetDiskUsage` 总是在显示一个饼图之前列举驱动器。增强程序将添加重新检查刷新显示时驱动器的连接状态的逻辑，从而根据情况添加或移去菜单命令和工具栏按钮。

再谈未结合命令

前面的章节描述了怎样通过指定键盘组合键和工具栏按钮打开命令，来设置名为

WordUpperCase 和 WordLowerCase 的命令，将它们与文本编辑命令结合。Visual C++ 环境也提供许多为图形编辑器设计的有用的未结合命令，你可以完全使用第 3 章“文本编辑器”中描述的过程。要看到图形编辑器的未结合和结合命令的列表，单击在 Help（帮助）菜单上的 Keyboard Map（键盘映射），并从 Help Keyboard（帮助键盘）对话框的组合框里选择 Image。图 4-21 显示了一个相似的列表。

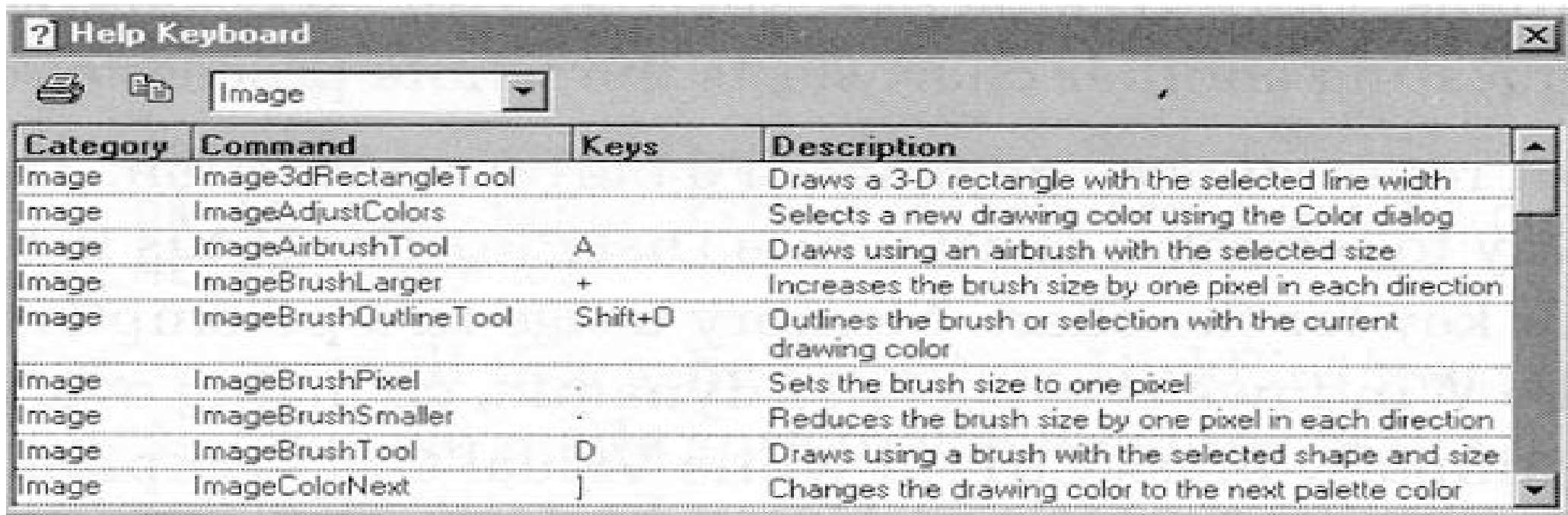


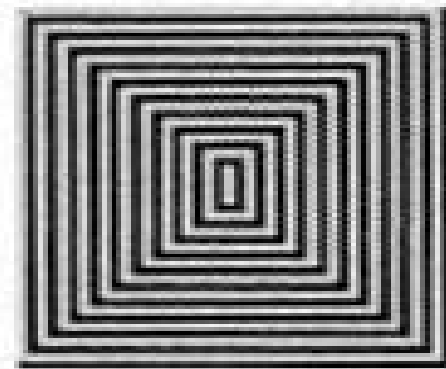
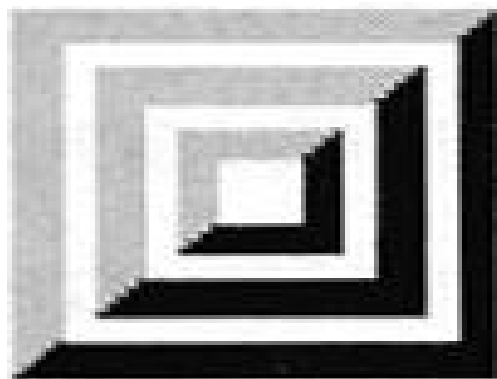
图 4-21 图形编辑器的命令，通过从 Help 菜单中选择 Keyboard Map 来显示

列表里的许多 Image 命令已经有了键盘指定。例如，图 4-21 显示了按下 A 键打开 Airbrush（喷雾器）工具，按下 + 或 - 键增加或减少其尺寸。但是，其他命令在为它们指定按键或创建组合键后，才可以使用。为了证明这点，本节显示了怎样为列表里的第一个名为 Image3dRectangleTool 的命令指定一个组合键，它是图形编辑器上 Rectangle（矩形）工具的一个变种。

从 Tool 菜单上选择 Customize，并单击 Keyboard 选项卡。从 Category 和 Editor

组合框里选择 Image，它将告诉 Visual C++，只在图形编辑器中使用新的键盘命令（也叫做图像编辑器）。从命令列表里选择 Image3dRectangleTool，单击 New Shortcut Key（新的快捷键）文本框，然后为命令输入一个组合键，例如 Shift+Ctrl+3。对话框显示说明目前没有分配的组合键，所以，我们不必担心是否从其他命令取走 Shift+Ctrl+3。单击 Assign 按钮，然后单击 Close（关闭）按钮来除去对话框。

Image3dRectangleTool 命令是为位图设计的，所以，要在运行中看到它，通过选择 File 菜单里的 New 打开图形编辑器，并双击 Files 选项卡里的 Bitmap File（位图文件）。按下 Shift+Ctrl+3 组合键时，或者按下指定给这个命令的任何其他组合键时，图形光标变成 Rectangle（矩形）工具使用的十字形。但是，当画一个矩形时，新的光标会稍微不同的效果。当你从左上角拖动光标到右下角时，上边和左边用目前的前景色，而右边和底部用背景色显示。通过用鼠标右键拖动来修改颜色。使用正确的颜色组合，嵌套的矩形显示一个三维外观，可以快速生成下面这样的图标：



在图形编辑器里工作时，Image3dRectangleTool 只是许多有用的键盘命令中的一个。例如，当设计自画的按钮时，相似的 ImageWindowRectangle 命令快速画出按钮处于正常状态、按下或非活动状态下的位图图像。在列表中浏览，并看一下你喜欢的命令。如果想为命令创建工具栏按钮，可查看第 3 章的“未结合命令”一节。

修整资源数据

由于有不使用的或效率低下的资源数据，Windows 程序经常是过度负载，那些程序使程序的文件尺寸增大，更糟糕的是，它占用内存。即使从来都没有使用的资源，也可以进入内存，但只是闲在那里，占用有效的共享资源。你可以解决这个问题，所需要付出的努力也不大。这一节研究一些减小程序资源数据大小的技术，并通过对 DiskPie1 程序做一些修改，来证明我们常常可以修整一个程序里

资源数据的有效数量。

要记住的第一件事是，你不必保留 AppWizard 提供给你的每一件东西。字符串是罪魁祸首，因为它们太长了，或者根本不需要它们。Win32 资源字符串保存在一个程序文件里，作为 Unicode 字符串，这意味着，你删除的每一个字符保存在两个字节里，而不是一个字节。如果你从通过 AppWizard 生成的 RC 文件里移去一个菜单命令，那么，你要确定移去和它在一起的提示字符串，并且不要忘记对应的加速键。

AppWizard 添加提示字符串到 RC 文件里，它描述了一个程序的系统菜单：

STRINGTABLE

BEGIN

AFX_IDS_SCSIZE	"Change the window size"
AFX_IDS_SCMOVE	"Change the window position"
AFX_IDS_SCMINIMIZE	"Reduce the window to an icon"
AFX_IDS_SCMAXIMIZE	"Enlarge the window to full size"
AFX_IDS_SCCLOSE	"Close the active window"

```
AFX_IDS_SCRESTORE    "Restore the window to normal size"
```

```
.  
.  
.
```

```
END
```

MFC 库文件包含这些相同的字符串，所以，如果你的程序使用 MFC 动态链接库，那么，你可以安全地从 RC 文件里删除这些字符串，而不用改变程序的行为（然而，如果你的程序是面向国际市场的，它有可能运行在配置不同使用语言系统的机器上，其他因素可能影响你是否使用 MFC 库文件提供的字符串的决定。看一下第 2 章开始的有关海外市场和动态链接到 MFC 的讨论）。通过释放一些资源字符串，动态链接到 MFC，而不是静态链接，将明显减少可执行文件的尺寸，因为 MFC 代码是在 DLL 里的，而不是在调用的程序里。当然，只有在两个或两个以上的程序同时使用 MFC 库运行时，用户才会获益。

我在前面的章节里提到，如果一个应用程序只提供一个边长为 32 个像素的正方形图标，Windows 将自动将图形按需要成比例转换成 16×16 或 48×48 的正方形。如果图标图形只包含直线和矩形，比例变换一般不会改变图形。对于不受比例变换影响的图形，你可能考虑只包含一个 32×32 的图形，而不是两个或三个

不同尺寸的图形。

你可以通过使它们的颜色数最小，来进一步减少图标和位图要求的空间。例如，默认状态下，图形编辑器的 New Icon Image（新图标图像）对话框（显示在图 4-17 上）为边长为 48 个像素的正方形图标图像设置 256 种颜色。但是，许多图标图形，像 DiskPie1 的图标，只包含几种颜色。为一个图形指定 16 种颜色，一个像素只使用 4 位；而不是使用 256 色的图形，每个像素占 8 位。通过减半 48×48 图形占用的空间，来减少图标文件的大小，这是最简单的步骤。在保存时，同时保存一个包含在图标里的颜色表，将增加 2K 字节大小的资源数据。要在图形编辑器里创建一个 16 色的 48×48 的图标图形，按下 Insert 键，打开 New Icon Image 对话框，单击对话框里的 Custom 按钮，并填写适当的控件框。

DiskPie1 使用了大量的资源空间，来保存它那巨大的工具栏位图，正如图 4-15 所示的那样。在位图里有 25 个图形，其中 24 个是副本，只有在重复的磁盘图形上显示的字母是不同的。在这种情况下，可以把它存成一个图形，并提供任何数目的工具栏按钮的服务。再增加一点附加的代码，它可以提供工具栏图形中其他 23 个图形的作用。这就大量节省了程序的尺寸。下面说明怎样实现它。

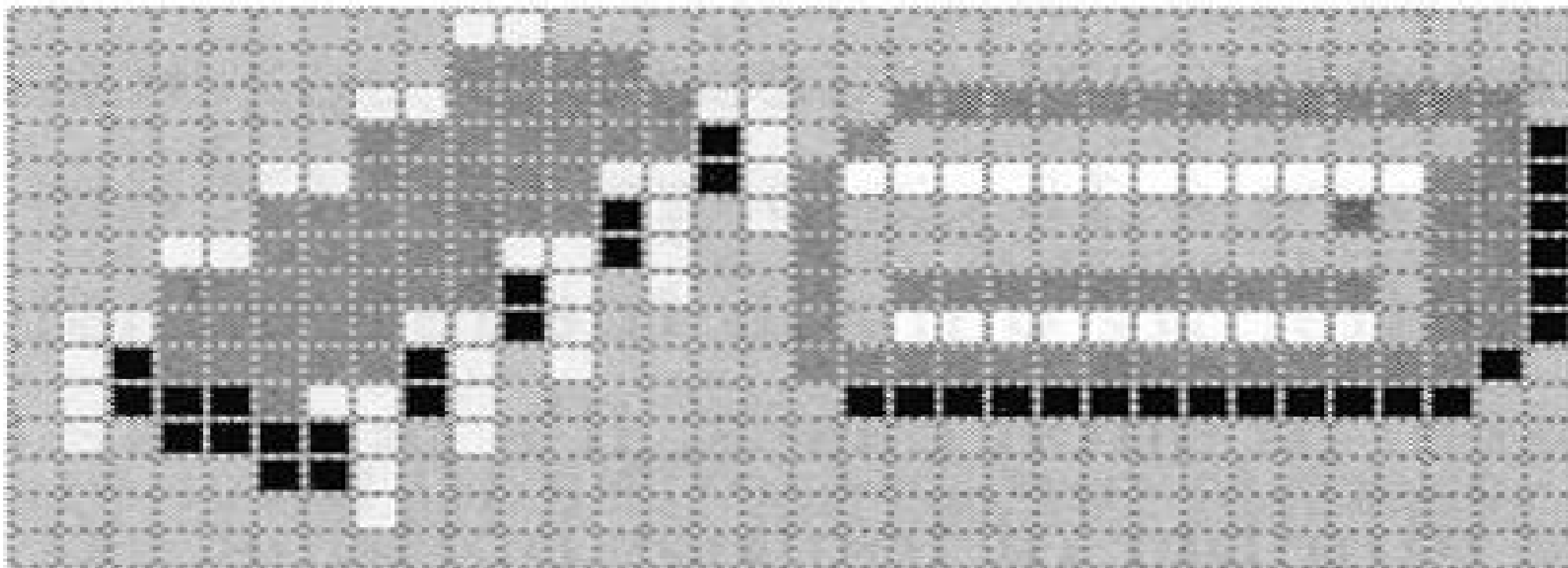
程序 DiskPie2

DiskPie1.exe 的发行版本文件大小是 24579 字节。通过去掉其资源数据中的许多无用数据，新的 DiskPie2 程序大小减小到 17920 字节，在没有明显改动程序的

同时，减少了几乎三分之一的大小。DiskPie2 用四种技术来减少它的资源数据的大小：

- 通过接收一个“Ready（就绪）”消息，这个程序没有在它的资源数据里包含提示字符串，而通过 MFC 库文件提供提示字符串，DiskPie2 在它的状态栏里为所有的命令显示一般性的提示信息，但那些在 Chart 菜单里选择的命令除外。
- 由于没有提示字符串，DiskPie2 必须在需要的时候生成工具提示。生成工具提示的代码比工具提示字符串本身占用更小的空间，导致程序大小整体降低。
- 48×48 图标图像有一个 16 色表，而不是 256 色表。
- DiskPie2 工具提示位图只有两个图像，而不是像 DiskPie1 那样要求的 25 个图像。

这两个工具栏位图的视图显示了 DiskPie2 的工具栏按钮所使用的图像：



这些图像复制了 DiskPie1 中工具栏的前两个图形，只是第二个图形没有标记，并使它适于表示任何磁盘驱动器。运行时，DiskPie2 给按钮的磁盘驱动器图形添加像“C:”和“D:”这样的名称。你可以在图 4-22 里看到这个结果。

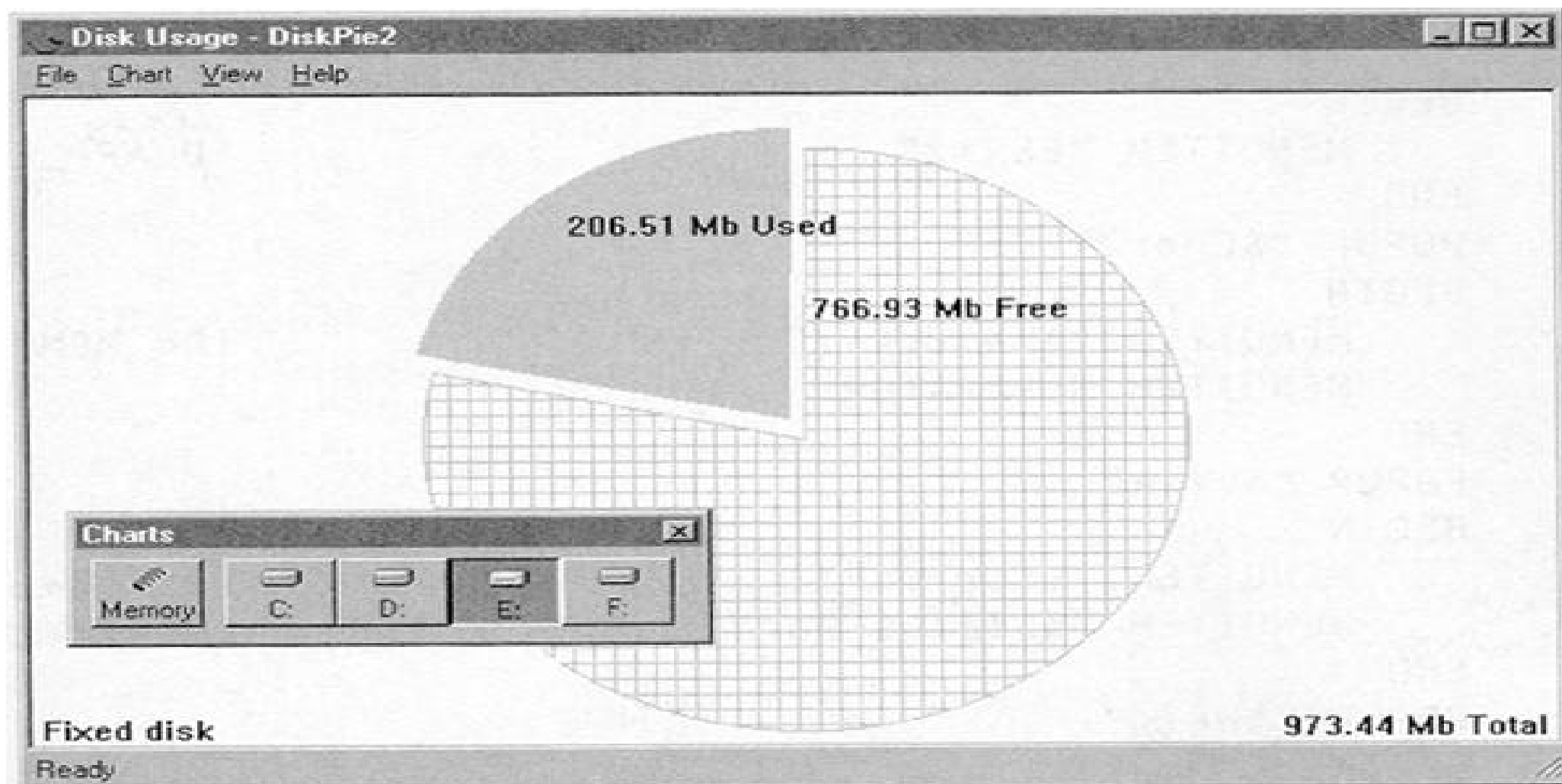


图 4-22 DiskPie2 程序

新的 DiskPie2 程序要求改变 DiskPie1 的三个源文件，来创建叫做 DiskPie2.rc，

MainFrm2.h 和 MainFrm.cpp 文件的升级版本。列表 4-2 显示了修订的代码，在 DiskPie1.rc 里去掉了不需要的提示字符串，在 MainFrm2.cpp 里接收为标记工具栏按钮构造工具提示文本而添加的指令。

DiskPie2.rc

```
// *****  
  
//  
  
// DiskPie2.rc  
  
//  
  
// *****  
  
#include "resource.h"  
  
#include "afxres.h"
```

```
IDR_MAINFRAME    ICON"res\\DiskPie2.ico"
```

```
IDR_MAINFRAME    BITMAP    "res\\toolbar.bmp"
```

```
IDR_MAINFRAME MENU
```

```
BEGIN
```

```
    POPUP "&File"
```

```
    BEGIN
```

```
        MENUITEM "E&xit",                ID_APP_EXIT
```

```
    END
```

```
    POPUP "&Chart"
```

```
    BEGIN
```

```
        MENUITEM "&Memory\tCtrl+M",      IDM_MEMORY
```

```
        MENUITEM SEPARATOR
```



```
END
```

```
POPUP "&View"
```

```
BEGIN
```

```
    MENUITEM "&Toolbar",          ID_VIEW_TOOLBAR
```

```
    MENUITEM "&Status Bar",       ID_VIEW_STATUS_BAR
```

```
END
```

```
POPUP "&Help"
```

```
BEGIN
```

```
    MENUITEM "&About DiskPie2...", ID_APP_ABOUT
```

```
END
```

```
END
```

```
IDR_MAINFRAME ACCELERATORS
```

BEGIN

"M",	IDM_MEMORY,	VIRTKEY, CONTROL
"C",	IDM_DISK_C,	VIRTKEY
"D",	IDM_DISK_D,	VIRTKEY
"E",	IDM_DISK_E,	VIRTKEY
"F",	IDM_DISK_F,	VIRTKEY
"G",	IDM_DISK_G,	VIRTKEY
"H",	IDM_DISK_H,	VIRTKEY
"I",	IDM_DISK_I,	VIRTKEY
"J",	IDM_DISK_J,	VIRTKEY
"K",	IDM_DISK_K,	VIRTKEY
"L",	IDM_DISK_L,	VIRTKEY

"M",	IDM_DISK_M,	VIRTKEY
"N",	IDM_DISK_N,	VIRTKEY
"O",	IDM_DISK_O,	VIRTKEY
"P",	IDM_DISK_P,	VIRTKEY
"Q",	IDM_DISK_Q,	VIRTKEY
"R",	IDM_DISK_R,	VIRTKEY
"S",	IDM_DISK_S,	VIRTKEY
"T",	IDM_DISK_T,	VIRTKEY
"U",	IDM_DISK_U,	VIRTKEY
"V",	IDM_DISK_V,	VIRTKEY
"W",	IDM_DISK_W,	VIRTKEY
"X",	IDM_DISK_X,	VIRTKEY

"Y",	IDM_DISK_Y,	VIRTKEY

"Z",	IDM_DISK_Z,	VIRTKEY

END

IDR_MAINFRAME TOOLBAR 16, 15

BEGIN

 BUTTON IDM_MEMORY

END

IDD_ABOUTBOX DIALOG 0, 0, 240, 65

STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU

CAPTION "About DiskPie2"

FONT 8, "MS Sans Serif"

BEGIN

```
        ICON                IDR_MAINFRAME,IDC_STATIC,10,22,20,20

        LTEXT               "DiskPie2 Version 1.0",IDC_STATIC,45,10,115,8

        LTEXT               ""Microsoft Visual C++ Programmer's Guide"",
                                IDC_STATIC,45,26,140,8

        LTEXT               "Copyright \251 1997, Beck Zaratian",IDC_STATIC,45,42,115,8

        DEFPUSHBUTTON       "OK",IDOK,195,10,35,40,WS_GROUP
```

```
END
```

```
STRINGTABLE
```

```
BEGIN
```

```
        IDR_MAINFRAME       "DiskPie2\nDisk Usage\n\n\n\n\n\n\n"
```

```
        AFX_IDS_IDLEMESSAGE "Ready"
```

```
END
```

MainFrm2.h

```
// *****  
  
//  
  
// MainFrm2.h  
  
//  
  
// *****  
  
class CMainFrame : public CFrameWnd  
{  
  
    DECLARE_DYNCREATE (CMainFrame)  
  
private:  
  
    CToolBar toolbar;  
  
    CStatusBar  statusbar;
```

```
CDiskDocdiskdoc;
```

protected:

```
afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
```

```
afx_msg void OnNewChart( UINT nID );
```

```
afx_msg void OnSetFocus( CWnd* );
```

```
afx_msg BOOL OnTooltip( UINT id, NMHDR* pNMHDR, LRESULT* );
```

```
DECLARE_MESSAGE_MAP()
```

```
};
```

MainFrm2.cpp

```
// *****
```

```
//
```

```
// MainFrm2.cpp
```

```
//
```

```
// *****
```

```
#define VC_EXTRALEAN
```

```
#include <afxwin.h>
```

```
#include <afxext.h>
```

```
#include <afxcmn.h>
```

```
#include "resource.h"
```

```
#include "DiskPie.h"
```

```
#include "DiskDoc.h"
```

```
#include "MainFrm2.h"
```

```
#include "DiskView.h"
```

```
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
```



```
BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

    ON_WM_CREATE ()

    ON_WM_SETFOCUS ()

    ON_COMMAND_RANGE (IDM_MEMORY, IDM_DISK_Z, OnNewChart)

    ON_NOTIFY_EX( TTN_NEEDTEXT, 0, OnTooltip )

END_MESSAGE_MAP ()

////////////////////////////////////

// Create main window, toolbar, and status bar

int CMainFrame::OnCreate( LPCREATESTRUCT lpCreateStruct )

{

    int      i, j;

    char      szDisk[] = "x:\0";
```

```
char        szMenu[] = "Disk &x\tx\0";

CMenu*      pmenu;


static const UINT indicator = ID_SEPARATOR;

if (CFrameWnd::OnCreate( lpCreateStruct ) == -1)

    return -1;

statusbar.Create( this );

statusbar.SetIndicators( &indicator, 1 );

toolbar.Create( this );

toolbar.SetWindowText( "Charts" );

toolbar.LoadToolBar( IDR_MAINFRAME );

toolbar.SetBarStyle( toolbar.GetBarStyle() |
```

```
        CBRSTOOLTIPS | CBRSTFLYBY | CBRSTSIZE_DYNAMIC );

toolbar.SetButtons( NULL, diskdoc.GetDriveCount() + 2 );

toolbar.SetButtonInfo( 0, IDM_MEMORY, TBBS_CHECKGROUP, 0 );

toolbar.SetButtonText( 0, "Memory" );

// The second "button" is a separator

toolbar.SetButtonInfo( 1, ID_SEPARATOR,

        TBBS_SEPARATOR | TBBS_CHECKGROUP, 0 );

pmenu = GetMenu()->GetSubMenu( 1 );

for (i=0, j=2; i < 24; i++)

{

    if(CDiskDoc::iDriveType[i] == DRIVE_FIXED    ||

        CDiskDoc::iDriveType[i] == DRIVE_REMOTE ||
```

```
        CDiskDoc::iDriveType[i] == DRIVE_RAMDISK)

{

    szDisk[0] = 'C' + (char) i;

    szMenu[6] = 'C' + (char) i;

    szMenu[8] = 'C' + (char) i;

    pmenu->InsertMenu( 0xFFFF, MF_BYPOSITION, IDM_DISK_C+i, szMenu );

    toolbar.SetButtonInfo( j, IDM_DISK_C+i, TBBS_CHECKGROUP, 1 );

    toolbar.SetButtonText( j++, szDisk );

}

}

toolbar.SetSizes( CSize( 45, 40 ), CSize( 16, 15 ) );
```

```
        toolbar.EnableDocking( CBRSS_ALIGN_ANY );

        EnableDocking( CBRSS_ALIGN_ANY );

        DockControlBar( &toolbar );

        return 0;

}

/////////////////////////////////////////////////////////////////

// Respond to menu/toolbar/accelerator commands

void CMainFrame::OnNewChart( UINT nID )

{

    CDiskDoc::nCurrent = nID;

    OnSetFocus( NULL );

    Invalidate();
```

```
        toolbar.Invalidate();                // Necessary when bar is floating
    }

////////////////////////////////////

// When focus regained, refresh display in case data has changed

void CMainFrame::OnSetFocus( CWnd* )

{

    if (CDiskDoc::nCurrent == IDM_MEMORY)

        diskdoc.GetMemoryUsage();

    else

        diskdoc.GetDiskUsage( CDiskDoc::nCurrent );

}

////////////////////////////////////
```

```
// When TTN_NEEDTEXT notification received. Generate tooltip text

BOOL CMainFrame::OnTooltip( UINT id, NMHDR* pNMHDR, LRESULT* )

{

    static char    szMemTip[]    = "Memory usage chart\0";

    static char    szDiskTip[] = "Usage chart for drive x\0";

    TOOLTIPTEXT* pTTT = (TOOLTIPTEXT*) pNMHDR;

    UINT nID    = pNMHDR->idFrom;

    if (nID == IDM_MEMORY)

        pTTT->lpszText = szMemTip;

    else

    {

        szDiskTip[22] = (char) (nID - IDM_DISK_C) + 'C';

        pTTT->lpszText = szDiskTip;
    }
}
```

```
    }  
  
    return TRUE;  
}
```

除了文件大小的差别外，DiskPie1 和 DiskPie2 之间唯一明显的差别在于它们工具栏的外观。由于 DiskPie2 为它的工具栏按钮使用单一的磁盘驱动器图像，所以，程序必须在运行时用文字标记它们。CMainFrame::OnCreate 函数在 For 循环里处理这个任务，这个循环运行通过每一个连接的驱动器。

```
for (i=0, j=2; i < 24; i++)  
{  
    .  
    .  
    .  
  
    pmenu->InsertMenu(    0xFFFF,    MF_BYPOSITION,    IDM_DISK_C+i,  
szMenu );  
  
    toolbar.SetButtonInfo( j, IDM_DISK_C+i, TBBS_CHECKGROUP, 1 );  
}
```



```
        toolbar.SetButtonText( j++, szDisk );  
  
}  
  
toolbar.SetSizes(Csize(45,40),Csize(16,15));
```

对 CToolBar::SetButtonInfo 的调用与在 DiskPie1 里的指令相同，不同的是最后的参数，它是在工具栏位图里按钮图形的下标，以 0 开始。DiskPie1 使用循环计数器从 24 个可用的图形中选择按钮的图形。DiskPie2 只处理一个磁盘驱动器图形。但是，为了不使每一个图标看起来都相似，代码也调用 CToolBar::SetButtonText 添加驱动器名称。添加的文本要求每一个按钮都有更多的空间，所以，代码通过调用 CToolBar::SetSizes 来增大按钮的尺寸到 45×40 ，而不是 DiskPie1 的 24×22 像素。

按要求出现的工具提示和提示字符串

由于在 DiskPie2 的资源数据中没有提示字符串，所以，它在运行时，根据需要来配置其工具提示。当系统要显示一个工具提示窗口时，它通过一个 TTN_NEEDTEXT 通知消息来通知程序。DiskPie2 在它的 CMainFrame::OnTooltip 函数里处理它。这个消息提供了一个指向 TOOLTIPTEXT 结构的指针，它是 NMHDR 结构的扩展形式：

```
typedef struct
{
    NMHDR        hdr;

    LPTSTR        lpzText;

    char          szText[80];

    HINSTANCE     hinst;

    UINT          uFlags;

}TOOLTIPTEXT, FAR *LPTOOLTIPTEXT;
```

hdr.idFrom 的值标识光标停留处的工具栏按钮，通过这个值，OnTooltip 决定合适的工具提示，lpzText 指向新的文本字符串，并返回一个 TRUE 值。函数在它的两个静态的字符串里保留它的工具提示：

```
static char szMemTip[]    = "Memory usage chart\0";

static char szDiskTip[]   = "Usage chart for drive x\0";
```

hdr.idFrom 里的 IDM_MEMORY 的值表明，第一个工具栏按钮需要工具提示文本。在这种情况下，代码将 lpstrText 的结构成员指向字符串 szMemTip。如果 hdr.idFrom 标识从 IDM_DISK_C 到 IDM_DISK_Z 中的一个驱动器状况按钮，OnTooltip 将 lpstrText 指向 szDiskTip 字符串，并将其中的字符 x 所处的位置用 C 到 Z 的合适的驱动器字母代替。这样，由于这些工具提示大多只差一个字符，所以，DiskPie2 的工具提示很容易创建。

通过相似的技术，就可以写 DiskPie2，使它按要求生成它缺少的状态栏提示信息。状态栏提示信息需要进行大量的测试，来决定光标是位于一个工具栏按钮上，还是在菜单命令上。在空闲时间中，MFC 框架自动完成这些工作，通过程序的 CCmdUI 对象来连续更新状态栏。随时产生滚动提示信息的秘密只是提供足够的字符串资源，来利用框架的碰撞测试，然后，建立一个超越 CStatusBar::OnSet 函数的完整的提示字符串。例如，DiskPie2 将在程序的 RC 文件里使用驱动器分配字母作为最小的提示字符串：

```
STRINGTABLE
```

```
BEGIN
```

```
IDM_MEMORY      "Memory usage"
```

```
IDM_DISK_C      " C "
```

```
IDM_DISK_D      " D "
```

```
.
```

```
.
```

```
.
```

```
IDM_DISK_X      " X "
```

```
IDM_DISK_Y      " Y "
```

```
IDM_DISK_Z      " Z "
```

```
END
```

要钩住 OnSetText 函数，你必须声明一个来自 CStatusBar 的类，并使用它在 CMainFrame 来创建状态栏：

```
class CStatusHook : public CStatusBar
```

```
{
```

```
protected;

afx_msg LRESULT OnSetText( WPARAM wParam, LPARAM lParam );

DECLARE_MESSAGE_MAP()

};

class CMainFrame : public CFrameWnd

{

    DECLARE_DYNCREATE (CMainFrame)

private:

    CStatusHook          statusbar;
```

当系统要在状态栏里显示一个简单的提示字符串时，一个消息映射和 OnSetText 处理函数捕获发送的 WM_SETTEXT 消息。处理函数通过将这个单独的字符提示插入一个冗长的文本字符串里，来生成一个完整的提示字符串，将 lParam 重新指向构造的字符串，并传递控制到低级的 OnSetText 函数。现在，状态栏中不是只显示单独的字母，像“ C ”，而是更加有用的字符串，如“ Disk C usage chart ”。下面是处理函数建立这样一个提示字符串的方式：

```
BEGIN_MESSAGE_MAP (CStatusHook,CStatusBar)
```

```
    ON_MESSAGE( WM_SETTEXT,OnSetText )
```

```
END_MESSAGE_MAP ()
```

```
LRESULT CStatusHook::OnSetText( WPARAM wParam, LPARAM lParam )
```

```
{
```

```
    // Constructed string must be static
```

```
    static char szDisk[] = " Disk x usage chart\0 " ;
```

```
    //If display is onl 1 character, convert it to full string
```

```
    if (lstrlen( (LPCTSTR) lParam )==1)
```

```
{  
  
    szDisk[5] = *((LPCTSTR) lParam);    //Set drive letter  
  
    lParam    =(LPARAM) szDisk;        //Repoint lParam  
  
}  
  
return CStatusBar::OnSetText( wParam, lParam );  
  
}
```

只有在实现代码比它所代替的资源数据占用更少的空间时，在运行时生成提示字符串才物有所值。不管怎样，当提示字符串必须反映某些动态的程序状态时，例如用户输入的不能预料的只读字符串资源，这个技术有时是很有用的。

第 5 章 对话框和控件

我们经常同时介绍对话框和控件，这是因为它们经常同时出现。控件是具有特殊用途的子窗口(例如，按钮、复选框或进度指示器)，而对话框是在其客户区内包含了一个或多个控件的父窗口。对话框和控件之间的联系非常密切，以至于我们干脆就将其整个组合叫做“对话”。

许多对话(例如 About 对话框)的作用是向用户传递信息，而其他对话则用于询问输入信息，为键入文件名或单击按钮做出选择提供一个方便的地方。如果此时你想起了工具栏，那么你就对了，工具栏就是对话的一种类型。像工具栏一样，大多数对话都是组成程序资源的用户界面元素的组成部分。

然而，有些对话框则不是程序资源数据的组成。像消息框这样的对话，以及所谓的通用对话是由系统提供的，它们可以通过像 MessageBox 和 GetOpenFileName 这样的 API 函数或像 CFileDialog 这样的 MFC 类来调用。你甚至可以在运行时使用 DialogIndirect API 函数为一个程序设计和创建对话，这个 API 函数可以把结构作为输入，而不是来自该程序的资源区域的数据。这样的对话严格说来属于编程问题，而不是你在 Visual C++ 中创建的资源，因此，你在这里找不到有关它们的任何讨论。

本章将接着前一章讨论资源数据的另一方面。首先，我们将看一看在项目的 RC 文件中定义对话的资源脚本，然后进入 Visual C++ 对话编辑器，在那里，可以像单击一样轻松地进行对话设计。本章将用几个例程来说明编辑器的功能，其中有一个例子显示了如何创建属性表(也多选项卡对话)。

对话脚本

向程序中添加对话是件很容易的事情。每个对话都在资源数据区中做为一系列命令而存在，而这些命令是由该项目的 RC 文件中的脚本编译而来的。它们用来指定诸如对话窗口大小、标题栏中的标题以及控件的放置之类的详细资料。要修改对话，通常仅仅需要在 RC 文件中编辑脚本。

为了举一个例子，让我们来看一看第 4 章的 Resource，在那里，AppWizard 将在 Help 菜单上生成一个命令，并为 About（关于）框生成所有必需的源代码，以及具有当前在 Windows 程序中越来越普遍的其他特征。AppWizard 还在项目的 RC 文件中编写用来定义对话及控件外观的脚本。下面是该脚本的另一种表示：

```
IDD_ABOUTBOX DIALOG DISCARDABLE 0, 0, 217, 55
```

```
CAPTION "About Demo"
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
ICON IDR_MAINFRAME, IDC_STATIC, 11, 17, 20, 20
```

```
LTEXT "Demo Version 1.0",
```

```
IDC_STATIC, 40, 10, 119, 8, SS_NOPREFIX
```

```
LTEXT "Copyright (C) 1998", IDC_STATIC, 40, 25, 119, 8
```

```
DEFPUSHBUTTON "OK", IDOK, 178, 7, 32, 14, WS_GROUP
```

```
END
```

脚本的第一行用符号 `IDD_ABOUTBOX` 来标识资源,该符号在项目的 `Resource.h` 文件中定义。`DISCARDABLE` 关键字后面跟着的四个数字,用来指定对话的大小。前两个数字(0, 0)用来设定对话窗口左上角的原点坐标。脚本中的所有其他坐标都相对于原点,正值的 `X` 是向右,正值的 `Y` 是向下。后两个数用来确定对话窗口的尺寸,在本例中,窗口的宽度是 217 个单元,高度是 55 个单元。

对话单元不是像素，而且，在 X 和 Y 方向上代表的距离也不一定相同。对话单元的尺寸取决于用于对话文字的字体，而对话字体在默认状态下为系统字体(在上面的脚本中，可选的 FONT 指令将对话字体指定为 8 磅 MS Sans Serif)。水平 X 方向的一个对话单元等于用于该对话字体的 1/4 平均字符宽。在垂直 Y 方向，一个对话单元是 1/8 字符高。因为在 Windows 采用的字体测量系统里面，将字符高度作为字体的大小，因此，对于一个使用了 8 磅字体的对话来说，一个垂直对话单元就是 1 磅，或 1/12 英寸。

尽管与可视化控件相比，完成控件的布局要复杂些，但将对话单元与字体大小结合起来，就意味着一个对话框将在不同的屏幕分辨率下保持同一尺寸。它还可以确保当对话的字体改变时，控件之间的相对位置保持不变，虽然大一点儿或小一点儿的字体仅仅是导致对话窗口及其内容在尺寸上扩大或缩小。在运行时，在对话窗口之中放置控件的程序应该首先通过调用 GetDialogBaseUnits API 函数来确定该对话的“基本单元”。基本单元定义了当前屏幕分辨率下的对话单元与像素之间的关系。MapDialogRect 函数可以自动实现会话，将对话单元中的坐标翻译成相等数量的屏幕像素。MSDN 在线帮助十分详细地介绍了这两个 API 函数。

对话脚本中的 STYLE 伪指令指定了影响对话外观的各种标志，例如，为对话提供标题栏的 WS_CAPTION 标志。DS_MODALFRAME 标志与对话的模态样式无关，它仅仅取决于程序创建对话的方式。模态样式的意思是，在对话可见期间，只有属于其他程序的窗口才可以接收输入焦点，因此，用户在运行负责该对话的程序(或至少是线程)之前必须先关闭它。例如，About（关于）框就是模态对话，

它可以阻塞程序执行，直至被放弃。非模态对话则不太明显，它允许用户不必关闭对话，就可以在程序内切换到另一个窗口。Visual C++文本编辑器的 Find 命令是非模态对话的好例子。将要在本章稍后介绍的名为 Color 的范例程序，说明了如何使用 Visual C++来创建非模态样式的对话。

尽管名字叫做 WS_SYSMENU，但这个标志只不过是 在对话标题栏的最右边放置了一个 Close 按钮而已，因为子窗口控件是不会有真正的系统菜单的。可选的 CAPTION 指令指定了将要在标题栏中出现的文字。

对话的控件定义语句被括在 BEGIN 和 END 语句之间。每条定义都包含有与脚本第一行一样的相同类型的四数字系列。在那里，前两个数字给出了控件左上角相对于对话原点的 X 和 Y 坐标，后两个数字用于指定控件窗口的宽度和高度。再重复一遍，所有的坐标和尺寸都是以对话单元表示的。

这个对话脚本为 About（关于）框提供了一个标签为 OK 的推压式按钮，以及三个静态控件，其中一个控件包含有程序图标的一个副本。我们将在本章后面的范例代码中遇到这些控件及其他控件。如果想获得包括有普遍控件在内的一个完整控件列表，请访问 MSDN 在线帮助，或查阅一些参考书，例如 Charles Petzold 编著的 Programming Windows 95（Windows 95 编程）。

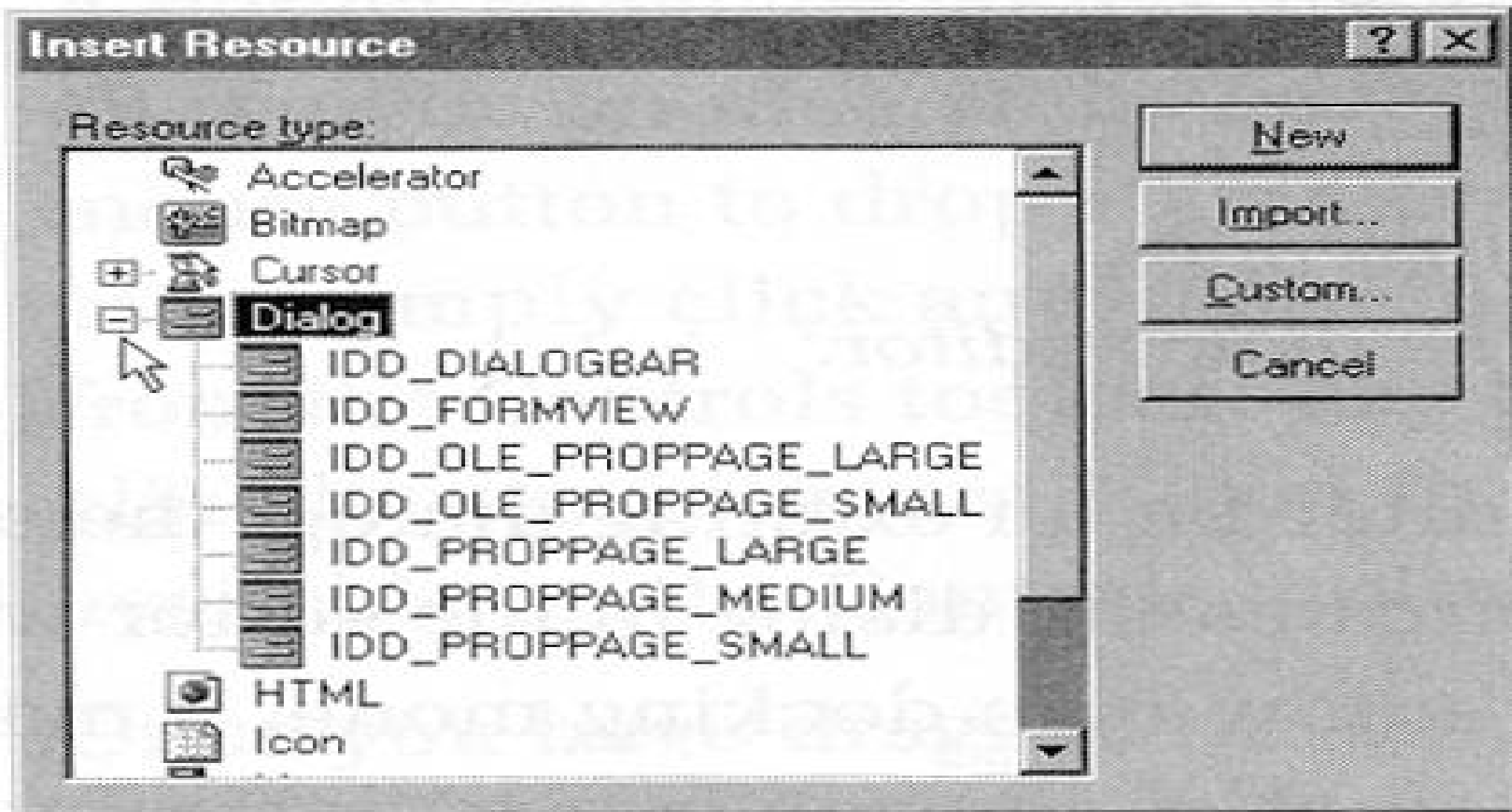
你可以对 AppWizard 提供的 About 对话进行改进。如果没有别的改进的话，就在版权行中添上你的名字。让我们首先来看一看 Visual C++对话编辑器，然后使用这个编辑器向 AppWizard 创建的对话框中添加一些其他东西。

对话编辑器

在以前进行 Windows 编程的时候，开发者们设计对话框的过程大概是这样的，它们不得不在 RC 文件中编写脚本，然后，再编译和运行程序，才能看到对话框实际的样子。这个不断摸索反复试验的过程，通常需要反复多次，才能正确地放置控件，并使其工作。但是，那个时代已经结束了。就是这个资源编辑器而不是其他任何东西，使得 Visual C++ 变得“形象”了，一旦你已经开始用对话编辑器来设计对话，那么就再也不用回到旧方法了。你不仅可以通过单击几下鼠标，把专业质量的对话放在一起，当你创建它的时候，看着它在屏幕上形成表单，而且还可以在编辑器中对对话的工作模式进行测试。控件看上去不对吗？想改变助记键吗？利用这个编辑器，修改工作会很轻松愉快。当你完成之后，也就知道了对话的确切工作方式以及它们在运行程序中的样子。正如我们将要在下一章中看到的，对话编辑器与 ClassWizard 结合得也很好，后者可以自动生成代码，来初始化和检索来自对话控件的数据。

至于其他资源，你可能会发现，通过使用文本编辑器编辑 RC 文件，以便更容易地对现有的对话脚本做出小小的修改。这一点儿都没有错。但对大多数修改来说，特别是当你创建一个新对话的时候，对话编辑器应该是你最好的选择。与其他的 Visual C++ 资源编辑器类似，可以用两种方法之一来启动对话编辑器，究竟使用哪一种方法取决于你是想创建一个新对话框，还是想继续使用已经在项目的 RC 文件中存在的那个对话框。如果想从头开始创建一个新对话框，请单击 Insert

（插入）菜单上的 Resource 来打开 Insert Resource（插入资源）框，然后从列表
中选择 Dialog：



展开 Insert Resource 框中的 Dialog 条目，将展现出代表特殊对话形状的标识符，例如具有工具栏尺寸的 IDD_DIALOGBAR。如果想创建一个带有默认的 OK 和

Cancel 按钮的通用对话，请双击 Dialog 而不展开它。

如果想继续在现有的对话上工作，请打开项目。并且单击 Workspace（工作空间）窗口 ResourceView（资源视图）窗格中列出的对话标识符。如果你想在非当前项目的一个项目中编辑对话，请单击 File（文件）菜单，并打开另一个项目的 RC 文件。该操作将打开其他项目的资源列表，你可以从中选取需要的对话框。不过，你不能将该资源存入当前的项目，原因是 Visual C++ 不能将来自 RC 文件的对话脚本合并到另一个对话脚本之中。文本编辑器为将另一个项目的对话复制到当前的项目提供了最现实的办法。可以按照第 4 章中介绍的方法在文本编辑器中打开其他项目的 RC 文件，选择和复制所需要的脚本，并将其粘贴进当前的 RC 文件。另外，还要在项目 Resource.h 文件中定义必要的标识符。然后，你就可以打开当前项目中的资源，并对其进行正常的编辑了。

图 5-1 显示了用于 Demo 这个虚构的 AppWizard 项目的 Workspace（工作空间）窗口，在里面，单击 ADD_ABOUTBOX 标识符，可以启动对话编辑器，并加载程序的 About（关于）框对话。

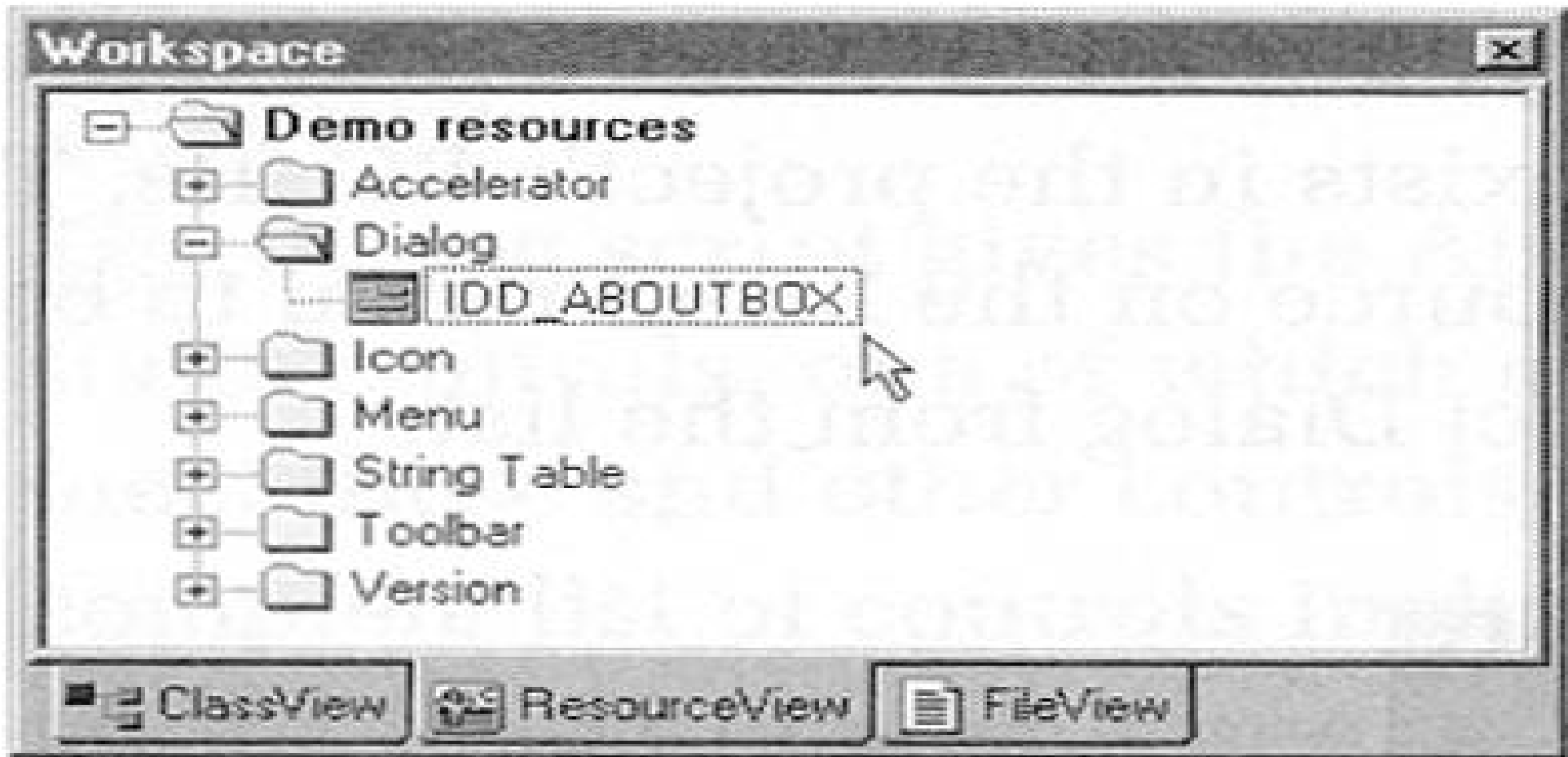


图 5-1 在 Workspace 窗口中，双击一个对话标识符，将启动对话编辑器

为现有的对话启动对话编辑器的时候，该编辑器将读取 RC 文件的脚本，并且在编辑器工作区中复制这个对话(如果 Workspace 或 Output (输出) 窗口处于连接模式之中，它可能会覆盖掉对话编辑器窗口。如果是这样的话，请右击每个覆盖

的窗口，并从上下文菜单中选择 Hide（隐藏）命令)。图 5-2 显示了已加载到编辑器中准备修改的 Demo 的 About 对话框。

尽管这样看上去已经足够真实了，但图 5-2 中的 About（关于）框只不过是一个没有运行的演示图片而已。单击对话工作区中的按钮或编辑框，可以选中控件，但不激活它。图 5-2 中所显示的规则指导是可选的，但是，你可能通过从 Layout（布局）菜单中选择 Guide Settings(指南设置)来将它们关闭。这两个叫做 Controls 和 Dialog 的工具栏需要详细解释一下。

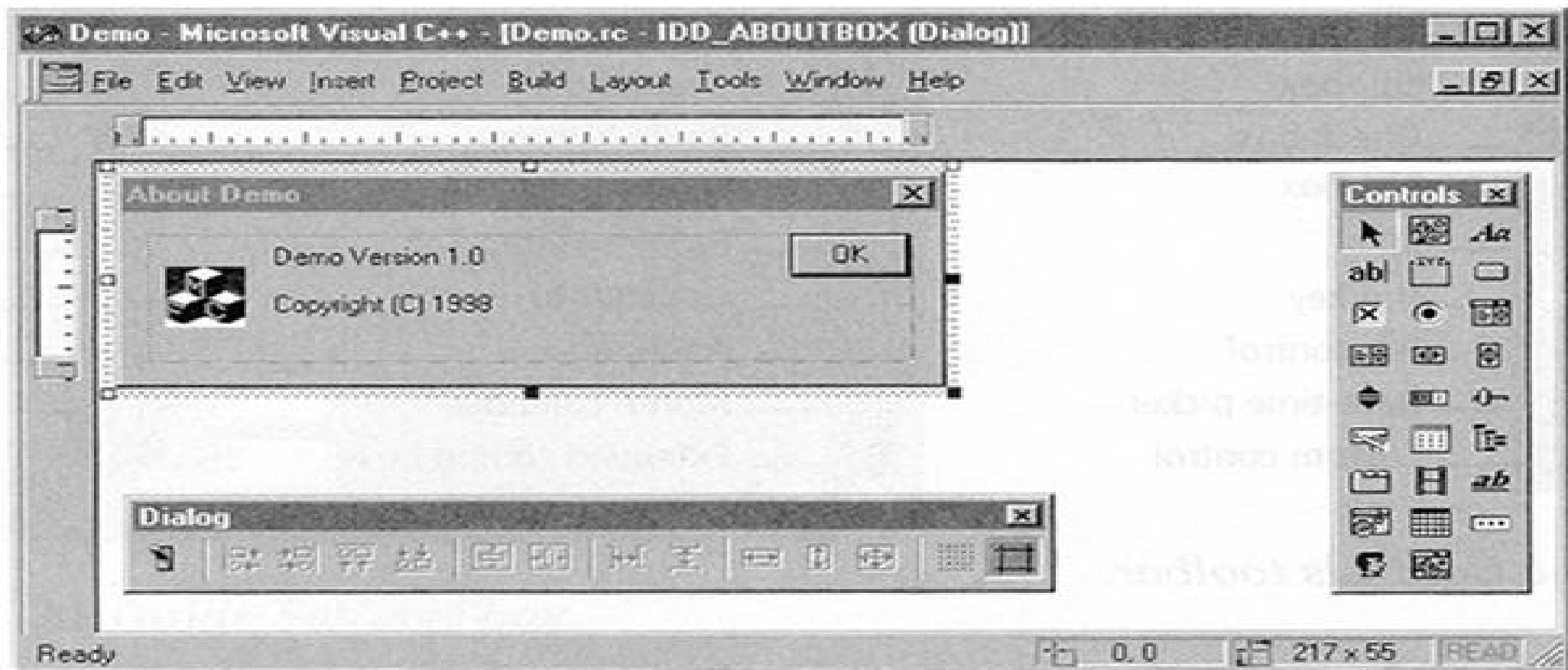


图 5-2 对话编辑器

Control（控件）工具栏

利用 Control（控件）工具栏，你可以单击访问对话框中放置的控件。单击代

表你想要的工具栏按钮，并且，将控件从工具栏拖入到对话框中的位置。在你拖动的时候，一个点划线的矩形显示了控件窗口的轮廓，这样，在你释放鼠标按钮放置控件之前，就可以对控件的尺寸和位置一目了然。另一种方法是，只需在从 Controls（控件）工具栏选中一个按钮，然后，在对话框的任何地方单击即可。控件窗口将在单击位置的中心地方出现。图 5-3 标识了 Controls（控件）工具栏上可以提供的控件类型。

一旦已经把自己想要的控件从 Controls（控件）工具栏拖进了对话区，那么，下面就是在自己喜欢的位置安排控件了。Dialog（对话）工具栏可以为你提供帮助，但是，首先我们必须来讨论一下在对话中选中控件窗口。

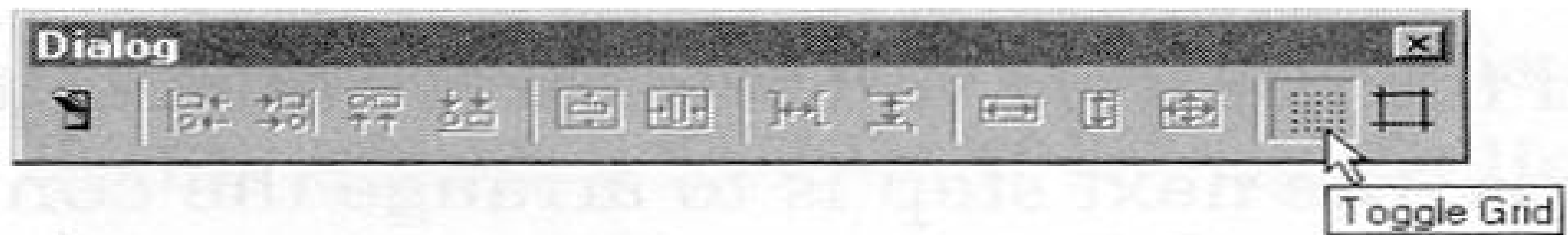


图 5-3 Controls 工具栏

选择和安排控件

将控件窗口拖入对话框的时候，在该控件周围包围着一个有阴影的矩形，这就表示该控件已被选中了。阴影的选择矩形有八个尺寸控制柄，它们是放置在矩形四角和四边的小正方形。像在 Windows 中的典型操作一样，你可以通过用鼠标指针来拖动一个尺寸控制柄，来调整选中的控件，或者如果想更加精确地进行调整的话，还可以通过按箭头键的同时按下 Shift 键来完成。每按一次，控件的尺寸改变一个单位。你还可以通过在工作内的任何地方而不是在控件窗口上单击，来选中并调整对话框本身。

当你将控件窗口拖入对话的时候，没有必要对其进行仔细定位，因为移动已经就位的控件是很容易的。单击对话工作区中的控件来选中控件窗口，然后使用鼠标拖动或使用箭头键移动它。为了更好地对齐，可单击 Dialog（对话）工具栏上的 Toggle Grid（开关网格）按钮打开网格。



当可以在对话窗口中看到网格的时候，控件仅从一个网格行移向另一个网格行，这个特征就是大家熟知的“与网格对齐”。在默认状态下，水平和垂直网格间距是 5 个对话单位，但你可以如图 5-4 所示的 Grid Settings（网格设置）对话框中改变这个对话框。

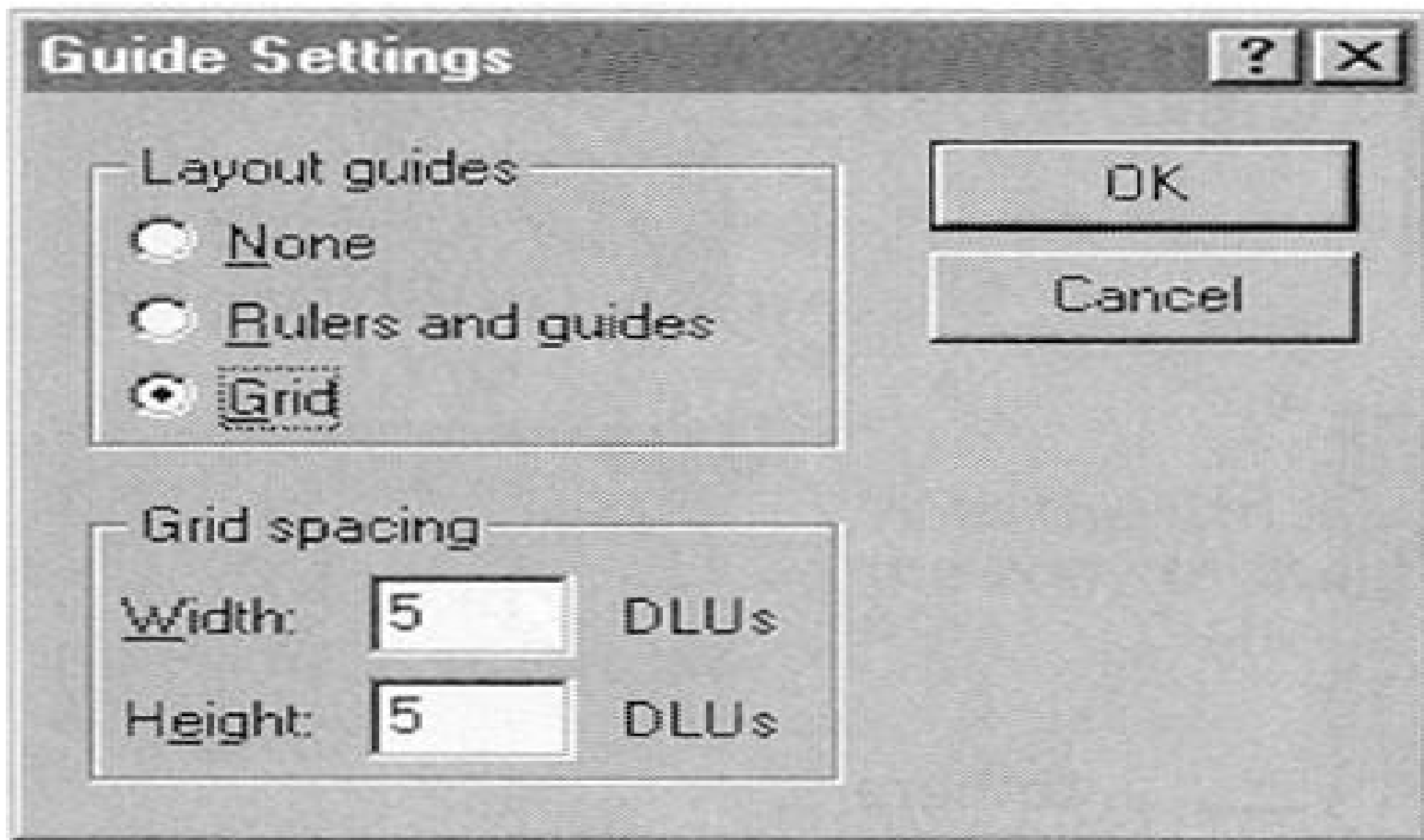


图 5.4 Guide Setting 框

如果与网格对齐特征阻碍你按照自己的方式来安排控件，那么只需将其关闭即可。如果你后来又打开了，编辑器不影响对话框中已经存在的控件窗口的放置。如果想临时抑制一下此特征，请在拖动控件的时候按 Alt 键。

每次将几个控件作为一组而不是单独一个进行移动或调整往往更加方便。对话编辑器为同时选中几个控件提供了两种方法。第一种方法是在按住 Shift 键的同时依次单击你想选中的控件。第二种方法最适于成组安排的控件。单击 Controls(控件)工具栏上的 Selection(选择)工具，并将虚线矩形拖到控件上面来选中它们。图 5-5 说明了这个过程。

如果你想取消对所选组中某个控件的选中，请在按住 Shift 键的同时单击该控件。还可以用同样的方法向组中添加控件。选中了多个控件之后，在选中组中，除了一个控件之外，除有控件的尺寸控制柄全都显示为空的，以表明它们处于非激活状态。剩下的那个具有实尺寸控制柄的控件被认为是选中组中的支配控件，编辑器可以从中确定作为整体的选中组的调整或对齐方式。例如，图 5-5 中的 Check1 控件就是选中的组中的三个控件的支配控件，因为它是唯一具有实尺寸控制柄的控件。在按下 Ctrl 键的同时，单击选中组中的另一个控件，可以使其成为新的支配控件。

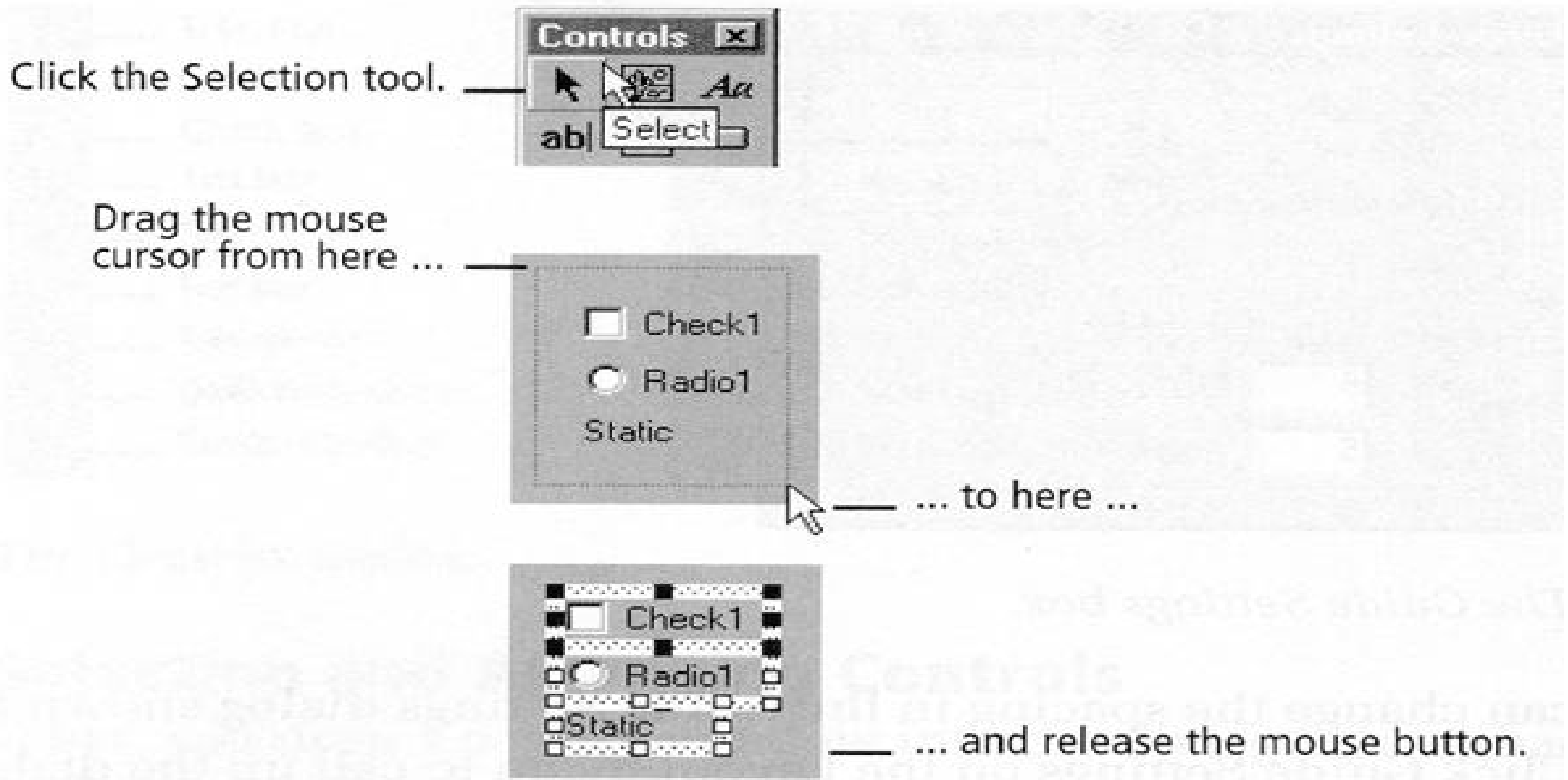


图 5-5 同时选中几个控件

仅有实尺寸控制柄处于活动状态；如果尺寸控制柄是空的，那么该控件就不能在

这个方向上进行调整。组合框控件仅仅在两边上具有活动的尺寸控制柄。这是因为通常看不到的控件下拉区域边是控件窗口的一部分，在建立窗口尺寸的时候，必须将其考虑进来。图 5-6 说明了如何改变组合框控件下拉区域的尺寸。

你可以创建控件的一个副本，并通过在按下 Ctrl 键的时候拖动控件窗口，来将其放在工作区中，如图 5-7 所示。该操作将创建一个新窗口，除了具有自己的标识符值外，它就是原始窗口的“克隆”体。

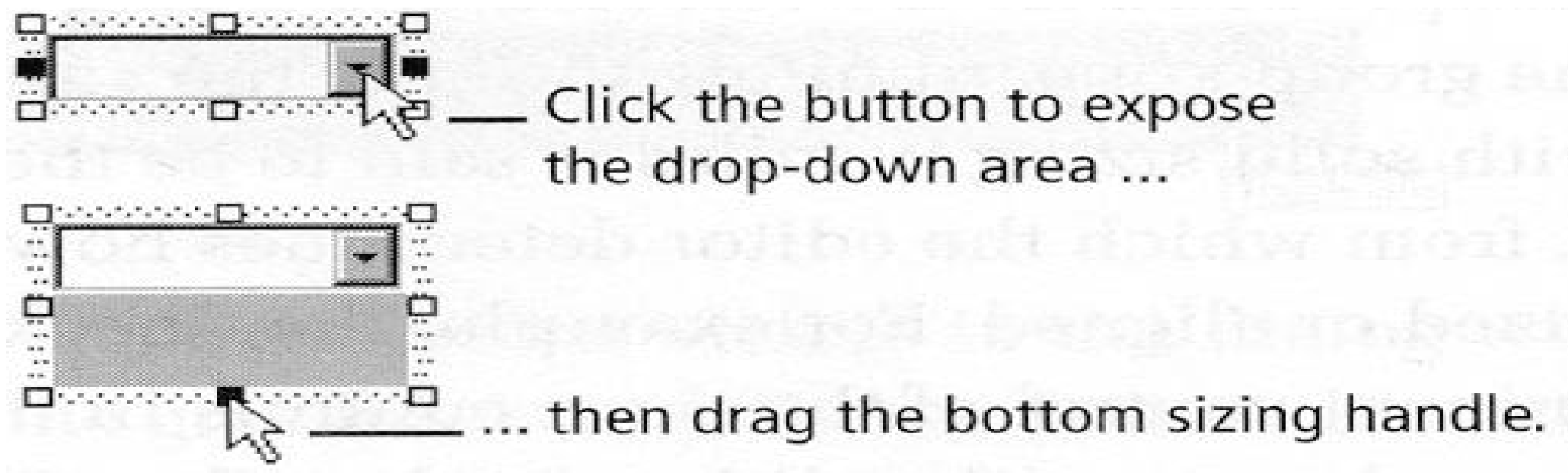
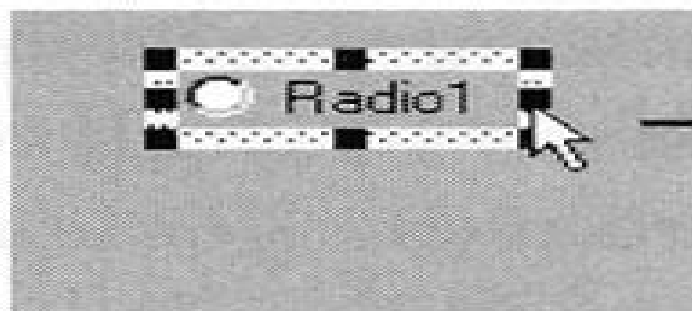
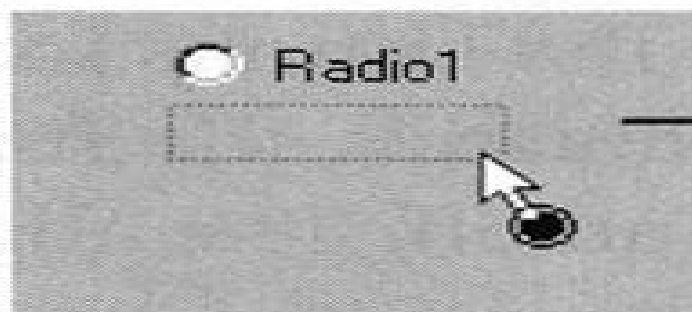


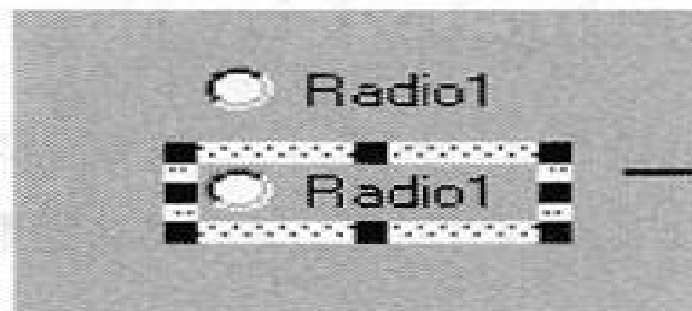
图 5-6 改变组合框的尺寸



Click a control to select it.



With the Ctrl key pressed, drag the copy to the desired location.



Release the mouse button.

图 5-7 在对话编辑器中复制控件窗口

Dialog（对话）工具栏

既然你已经知道了如何选中一组控件，那么，图 5-8 所示的 Dialog（对话）工具栏就更有意义了。在对话中四处拖动控件是找到合适位置的好办法，但是，如果要精确对齐控件窗口，你应该使用 Dialog（对话）工具栏上的工具。利用它们，你可以在对话框内部对齐了的行和列中放置控件窗口，为对话提供整齐的和专业的外观。这个工具栏上还有一个测试模式开关，它允许你将新的对话用于测试驱动程序，来看看它在实际中的外观和行为。

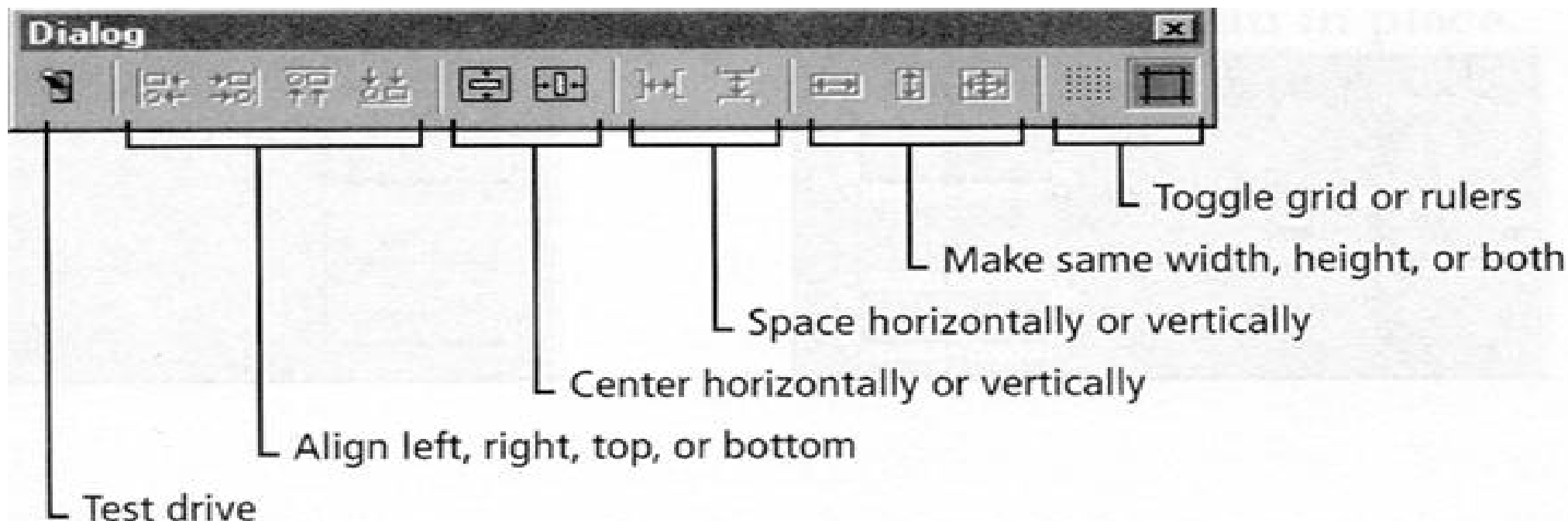


图 5-8 Dialog（对话）工具栏

所有的 Dialog（对话）工具栏按钮在 Layout（布局）菜单上都具有功能相同的命令，因此，如果你不想看见工具栏的话，还可以把它关闭。如果想显示或隐藏工具栏，请从 Tools（工具）菜单中选择 Customize（自定义），单击 Toolbars（工具栏）选项卡，然后单击列表中的 Dialog（对话）复选框。

正如你在图 5-8 中所看到的，工具栏将按钮安排在五个逻辑组中，它们分别用于对齐、居中、间隔、调整控件的尺寸，以及打开和关闭网格和标尺指南。在图

5-8 中呈现为灰色的对齐、间隔和尺寸调整工具，仅仅当对话框中选中了两个或多个控件时才被启用。

下面几节将阐述 Dialog（对话）工具的几个效果。通常，为了将一组控件放到所需要的位置，需要采用几个工具。因此，你必须考虑一下应用工具的顺序。不过，即使做错了也没关系。如果一个工具的效果不合适，只需单击 Edit（编辑）菜单上的 Undo（撤消）即可。

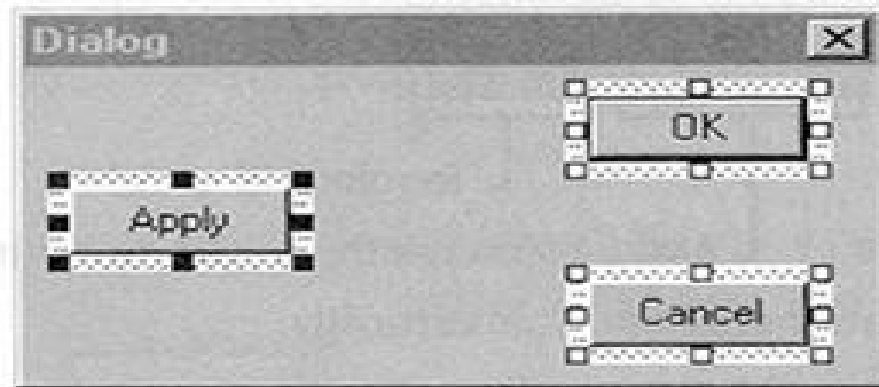
对齐工具

对齐工具的作用是将所选组的控件与支配控件对齐。例如，单击 Align Left（左对齐）按钮，将改变选中控件的 X 坐标，以便与支配控件的 X 坐标相匹配，而不影响 Y 坐标。

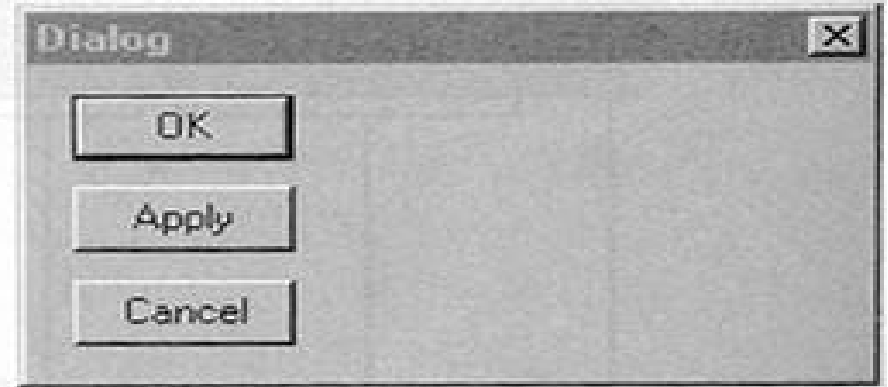
Click the Align Left tool ...



... to change this ...



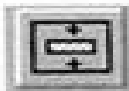
... to this.



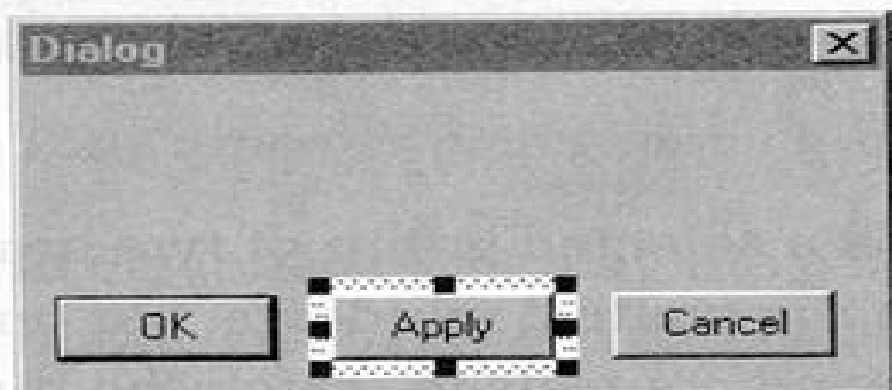
居中工具

居中工具可以作用于单个选中控件或一组控件，将选中部分放在对话客户区的水平或垂直中心。

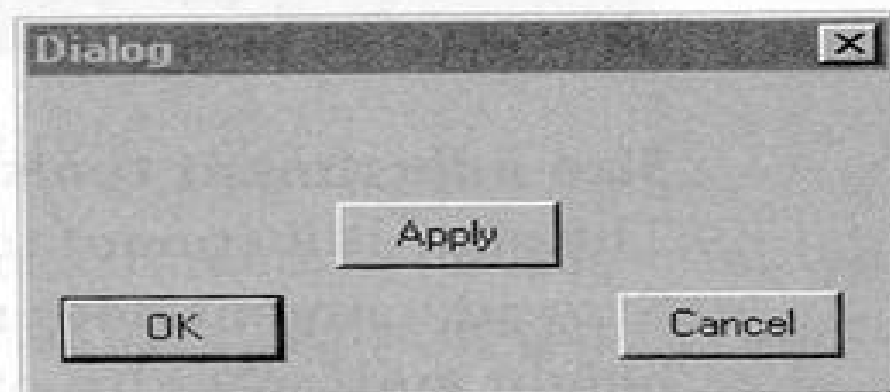
Click the Center Vertical tool ...



... to change this ...



... to this.



间隔工具

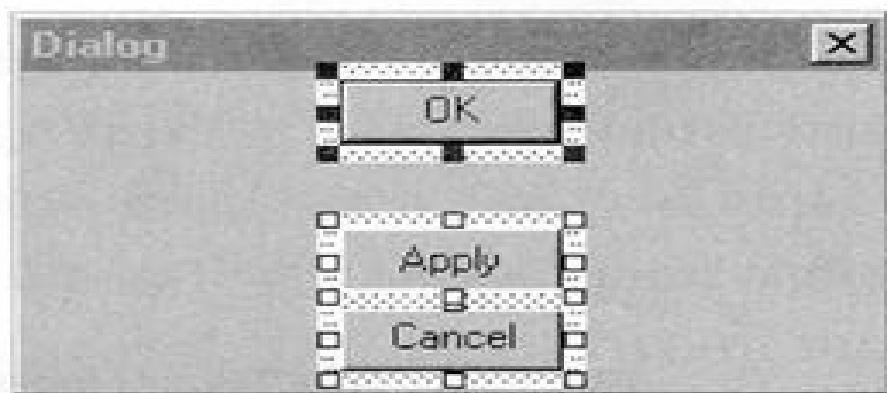
间隔工具仅仅在包含有三个或更多控件的选中组上才能工作。在 Dialog（对话）工具中，它们是独一无二的，因为它不能区分哪一个选中控件是支配控件。水平间隔将改变选中组中除了最左边和最右边控件之外的 X 坐标，使水平方向的控件均匀排列。垂直间隔工具完成 Y 坐标方向上相同的工作，使控件在垂直方向上均匀分布。在下列中，垂直间隔工具(编辑器为它取了个容易引起混淆的名字，Space Down（向下间隔）)调整的仅仅是 Apply（应用）按钮，OK 和 Cancel 按钮

依旧在原来的位置不变。

Click the Space Down tool ...



... to change this ...



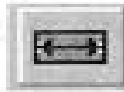
... to this.



尺寸调整工具

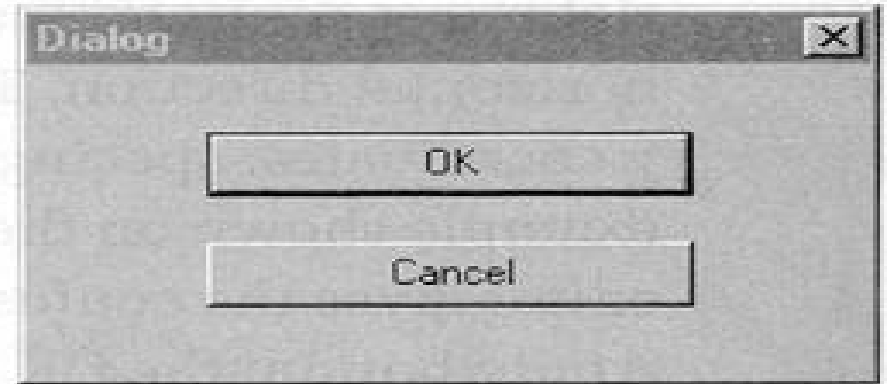
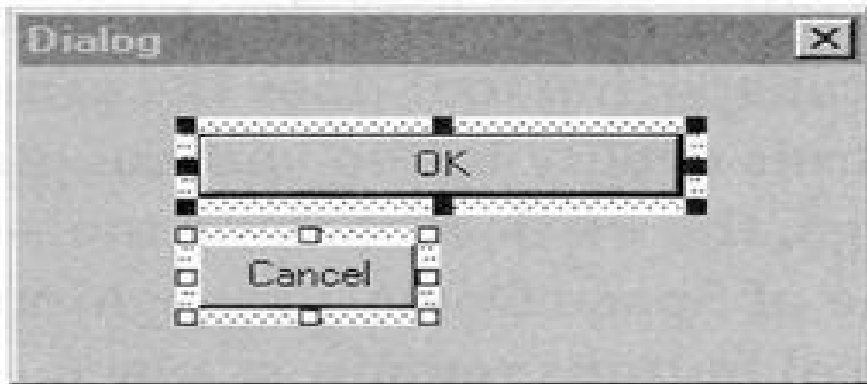
尺寸调整工具作用于包含有两个或更多控件的选中组。尺寸调整并不移动控件窗口，而仅仅改变选中的控件的高度和宽度，以便与选中部分中支配控件的尺寸相配。

Click the Make Same Width tool ...



... to change this ...

... to this.



控件属性

对话工作区中的每个控件都有一个 Properties（属性）对话框，你可以在里面为控件指定标识符和值，键入标签，设置适合于控件窗口的样式标志。如果想展现控件的 Properties（属性）框，请右击工作区中的控件窗口，并从弹出的上下文菜单上选择 Properties。你还可以选中一个控件，并单击 View（查看）菜单上的 Properties 命令。每个控件类型的 Properties（属性）对话在外观和内容上略有不同。图 5-9 显示了一个复选框控件的 Properties（属性）框。



图 5-9 一个复选框控件的 Properties（属性）对话

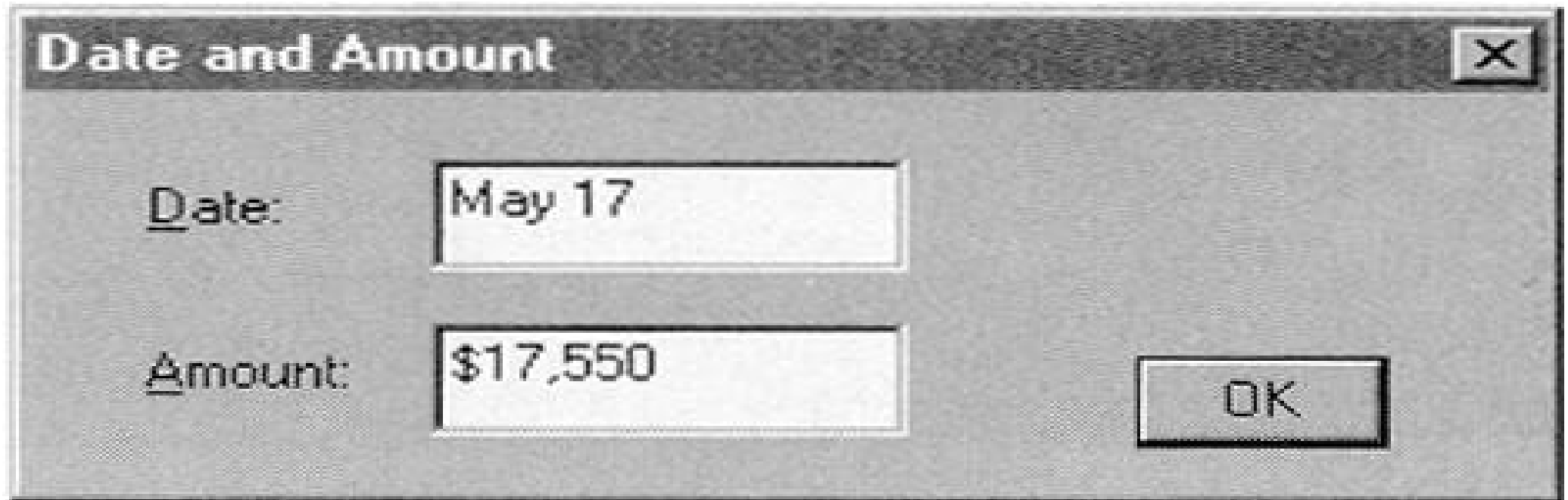
在第 4 章中，你可能已经习惯于在 Visual C++ 图形编辑器中双击一个项目来打开合适的 Properties（属性）框了，但该项技术在对话编辑器中则不具有相同的效果。双击对话工作区中的一个控件窗口可以导致三个结果之一，这主要取决于环境和控件类型。

- 如果控件接受用户输入，双击控件窗口，将调用一个题为 Add Function（添加函数）的对话，它允许你为这个控件快速添加存根处理程序函数。

- 双击静态控件或对话框，将在文本编辑器中打开对话类的源文件，并在构造器中放置插入符。
- 如果项目没有 ClassWizard 数据库 CLW 文件（将在下章介绍），那么双击非静态控件的一个控件将没有任何效果。这种行为不同于版本 5，在版本 5 中，没有 CLW 文件时，双击将显示该控件的 Properties（属性）框。

如果控件接受用户输入，例如复选框或滑块控件，那么，当你键入控件标题的时候，将分配一个独一无二的助记键。利用助记键，用户通过按下键盘上的一个键，可以将输入焦点移到控件上。前一章介绍了用于菜单命令的助记键，用于标签的助记键也没有什么差别。只需要在任何字符前加上“与”符号(&)，将字符作为控件的助记键标识即可。例如图 5-9 中的复选框就具有标题“&Red”，在里面，助记键是字母“R”。

对于滑块控件或像编辑框这样的文字条目控件，应在静态控件标签中为其分配助记键，下面是一个简单的例子：



在这种情况下，按 **Alt+D** 将把焦点放在 **Date** 编辑框中，按下 **Alt+A**，将把焦点放在 **Amount** 编辑框中(当两个编辑框都没有焦点时，**Alt** 键是可选的)。为了使助记键工作，每个编辑框必须按照按 **Tab** 键的移动顺序排列它的静态控件标签。

按 **Tab** 键的移动顺序

助记键并不是为某个控件分配输入焦点的唯一办法。用户可以用鼠标单击控件，或通过反复按 **Tab** 键跳到某一控件。每当系统检测到 **Tab** 键的时候，它就把输入焦点移动到 **Tab** 键移动顺序的下一个控件。

按 Tab 键的移动顺序可以由 RC 文件对话脚本的控件语句顺序中推出。例如，当在下面脚本中定义的对话出现的时候，IDC_COMBO1 组合框控件就具有了输入焦点。按下 Tab 键，将输入焦点从 IDC_COMBO1 控件按显示的顺序移到另一个控件。当 ID_CANCEL 按钮具有了焦点之后，再次按下 Tab 键，将把焦点循环回 IDC_COMBO1 的开始处。

```
IDD_DEMO_DLG DIALOG 0, 0, 247, 65
```

```
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION
```

```
CAPTION " Demo Dialog "
```

```
FONT 8, " MS Sans Serif "
```

```
BEGIN
```

```
COMBOBOX IDC_COMB01,15,15,159,80,WS_VSCROLL |
```

```
WS_TABSTOP | CBS_DROPDOWN|CBS_AUTOHSCROLL
```

```
CONTROL " and ",IDC_RADIO1, " Button ",BS_AUTORADIOBUTTON |
```

```

        WS_TABSTOP,14,39,25,10

CONTROL        " or ",IDC_RADIO2," Button",BS_AUTORADIOBUTTON |

        WS_TABSTOP,44,39,20,10

COMBOBOX        IDC_COMBO2,69,39,106,80,WS_VSCROLL | WS_GROUP |

        WS_TABSTOP | CBS_DROPDOWN | CBS_AUTOSCROLL

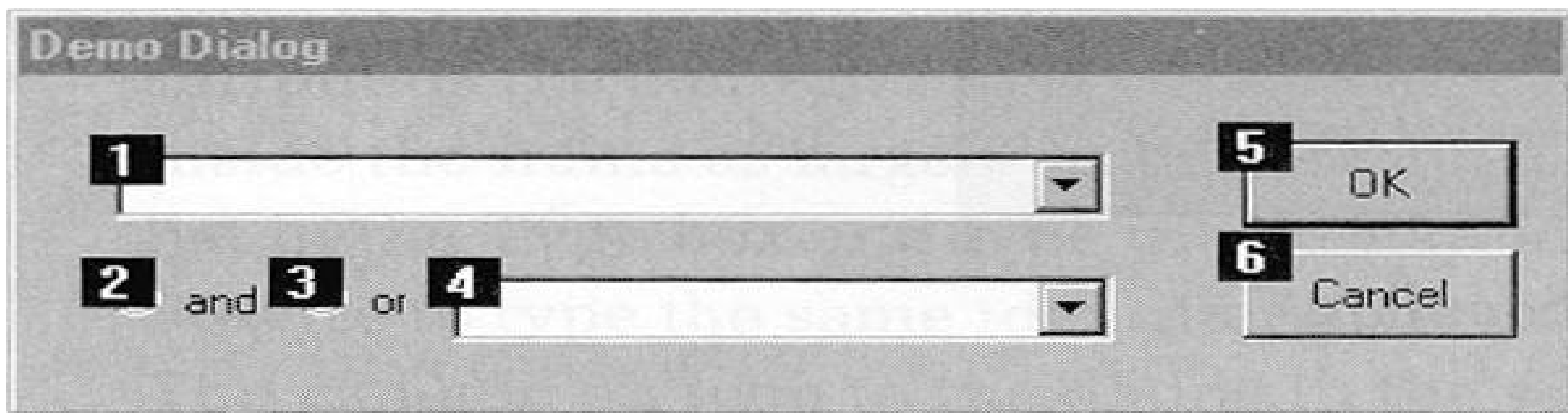
DEFPUSHBUTTON  " OK ",IDOK,196,12,44,17,WS_GROUP

PUSHBUTTON      " Cancel",ID_CANCEL,196,34,44,17,WS_GROUP

END

```

对话编辑器可以将你与 RC 文件中的详细信息隔离开来，但是，你仍然需要留心询问用户输入对话中 Tab 键的移动顺序。检查顺序通常是在单击 Dialog(对话) 工具栏上的测试开关之前所做的最后一件事情。从 Layout(布局) 菜单选择 Tab Order(Tab 次序)，可以显示当前对话的 Tab 键移动顺序，该顺序是作为序号码在对话控件附近出现的。下面是在前面的脚本中定义的用于对话的工作区：



如果想修改 Tab 键的移动顺序，请单击以第一个控件开头的序列中的每个控件，也就是当对话首次出现时，你想让其具有输入焦点的那个控件。如果现有的顺序仅仅对于某一点来说是正确的，那么，你可能会发现，只改变错误的是很容易的。在单击具有最高的正确 Tab 键的移动顺序号的控件窗口的同时按下 Ctrl 键，然后释放 Ctrl 键，并继续单击所要求顺序中的控件，直到顺序正确为止。例如，如果想改变控件 4 到 6 的顺序，请在按下 Ctrl 键的时候单击控件 3，然后依次单击你想让其拥有制表号 4，5 和 6 的控件。按 Enter 键设置顺序并返回到编辑模式。

对于成组出现的控件，例如单选按钮和复选框，改正 Tab 键的移动顺序特别重要。第 6 章将主要介绍 Tab 键的移动顺序，以及它在使一组单选按钮互斥(在一组按钮中设置一个将自动清除其他按钮)中所起的作用。

例 1：修改 About 对话

这里是演示对话编辑器的一些功能的简单例子。假定你想修正一下前一章中为虚构的 Demo 项目所引入的 About(关于)框。图 5-10 显示了一种可能的修改结果，它也是用于虚构程序 SpifDemo 的 About(关于)框，它制作得更加精心。

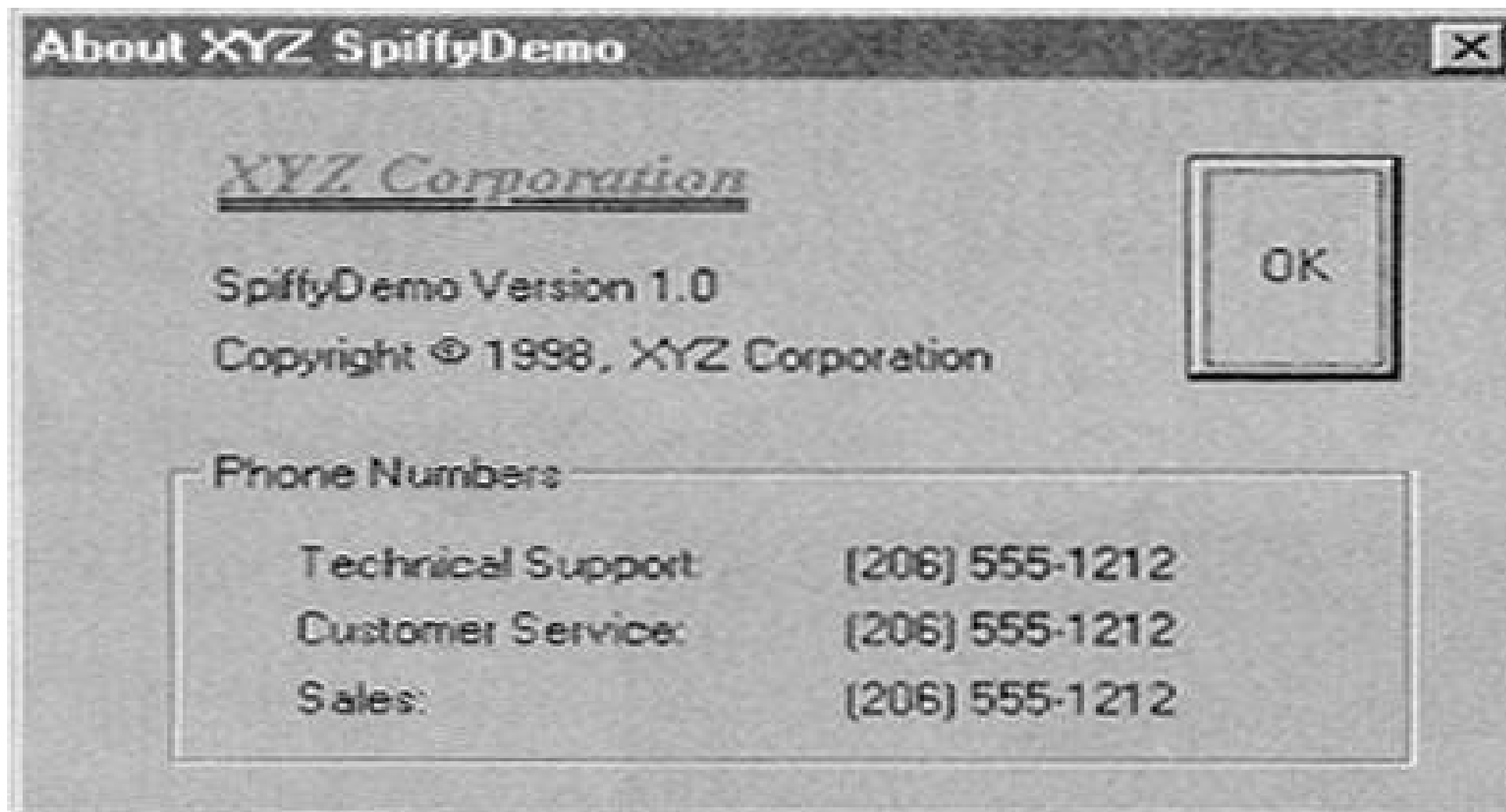


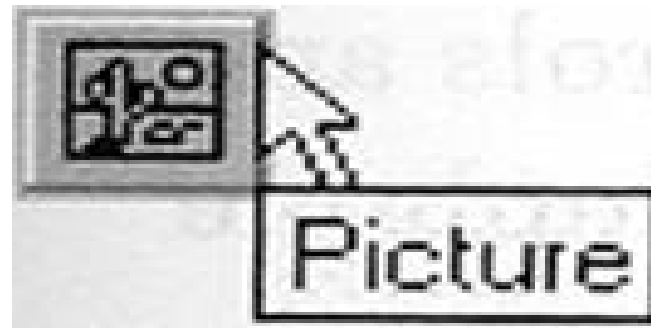
图 5-10 修改后的 About (关于) 框

位图标志事先在 Visual C++ 图形编辑器中创建，并保存在名为 XYZCorp.bmp 的

文件之中。除此之外，在对话编辑器中所做的改动只用几个步骤即可完成，所花的时间也就 5 分钟。如果你想试着自己修改 About（关于）框，那么，可以以一个名为 Demo 的免费分布的 AppWizard 项目作为开始。然后按如下步骤进行操作。

1. **创建标志**：单击 Insert（插入）菜单上的 Resource（资源），并选中 Bitmap（位图），来启动 Visual C++ 图形编辑器。设计标志位图，从 View（查看）菜单选择 Properties，来为新的位图资源提供标识符，并指定文件名。完成之后单击 File（文件）菜单上的 Save（保存）。
2. **启动对话编辑器并加载 About（关于）框**：在 Workspace（工作空间）窗口的 ResourceView 窗格中，双击 IDD_ABOUTBOX 来启动对话编辑器。
3. **调整对话的尺寸**：通过单击没有被控件占据的灰色对话框里面的任何地方，来选中对话窗口，并拖动底部尺寸调整控制柄来扩大窗口。
4. **改变标题**：在对话窗口处于选中状态的时候，单击 View（查看）菜单上的 Properties 命令，并在 Dialog Properties（对话属性）框中重新编写标题。
5. **添加标志**：在对话工作区中选中 MFC 图标，通过按 Del 键将其删除，然后用在步骤 1 中创建的代表公司标志的新位图来代替它。如果想向

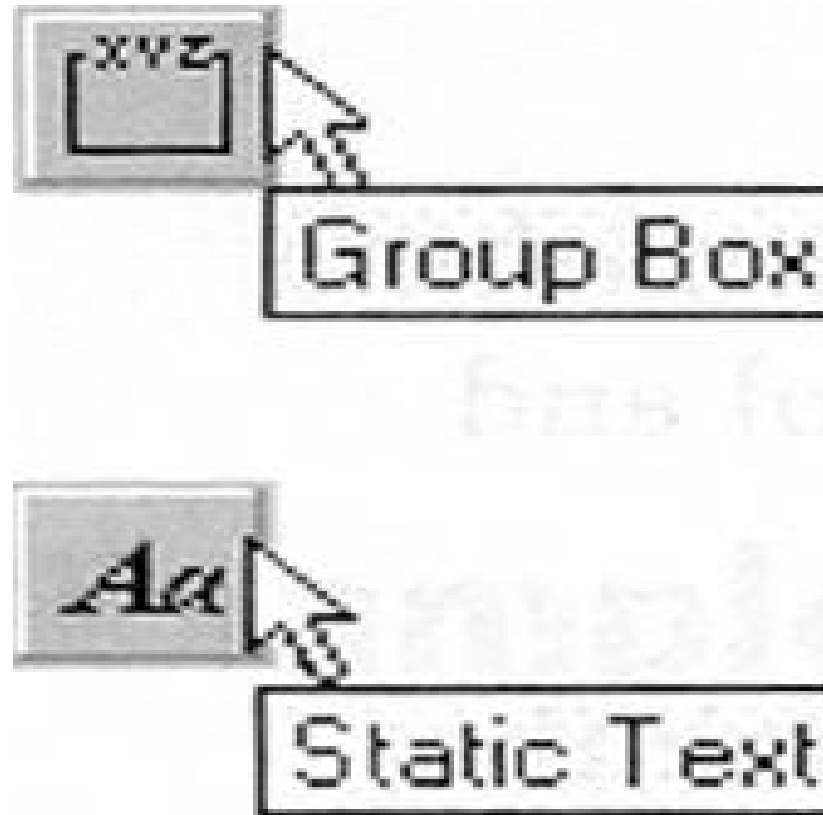
对话添加一个位图，请单击 Controls（控件）工具栏上的 Picture（图片）按钮，并将控件拖到对话中的任何地方（位置不需要十分精确）。在图形控件处于选中状态的时候，使用 Properties（属性）命令可以调用代表新图形控件的 Picture Properties（图片属性）框（如果想选中像图片这样的空控件，请单击该控件周围的框架，原因是编辑器不能将框架里面的单击识别为以控件作为目标）。从 Type（类型）框的列表中选择 Bitmap（位图），并在 Image（图像）框中键入位图标识符。一定要键入与在步骤 1 给定的相同的标识符。你可以按照同样的方式向对话中添加图标，只是应选择 Type（类型）框中的 Icon（图标），而不是 Bitmap（位图）。



6. 编辑对话文字：对话中每两行文字就是一个静态控件。为每个静态控件调用 Text Properties（属性）框，并修改 Caption（标题）框中的文字。在上例中，将“Demo”改为“SpitfyDemo”，版权行扩展为包括 XYZ Corporation。在原来文字中用 \251 代替“(C)”版权符号，\251 是 AXSI

字符的八进制代码(如果想获得你想用相同方式添加到文字控件的其他 ANSI 字符列表, 请参考附录 A)。

7. **添加电话号码** :该步骤需要用 Controls(控件)工具栏提供的 Group Box (组框) 工具和 Static Text (静态文本) 工具。首先, 单击 Group Box (组框) 按钮, 并将它的图像移动到对话框的下半空白区域。在需要的时候将其扩大, 然后再次单击 Properties(属性)命令打开 Group Box Properties (组框属性) 对话, 并将标题改为“Phone Numbers”, 下一步, 单击 Static text (静态文本) 按钮, 并将其拖到对话中的组合框里面。在静态控件窗口仍处于选中状态的时候, 通过在按住 Ctrl 键的同时短距离地拖动控件, 可以创建它的多个副本。将这些副本拖到独立的行上, 一个接一个地放置。在这个阶段, 对齐并不是很重要的, 因为以后可以更加仔细地进行。

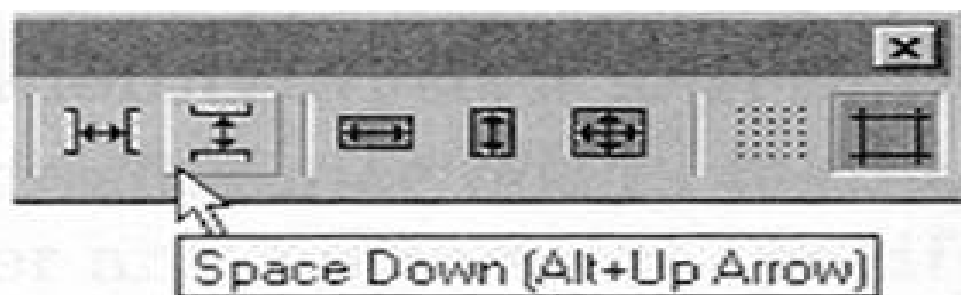


在默认状态下，静态控件包含有单个单词“Static”。为这三个静态控件中的每一个打开 Text Properties（属性）对话，并在 Caption（标题）框中用新文字和电话号码代替“Static”。单击位于 Properties 对话左上角的按钉按钮，可以阻止它在两个选择之间消失。

\t 制表符在三个静态控件中为电话号码留空，并将它们对齐，例如“Sales:\t\t\t(206)555-1212”。不过，在 Caption 栏中键入\t 字符将只为

静态控件扩展一个单空格，因此，\t 字符后面的文字可能不会在控件窗口中出现。如果看到它，你可以通过拖动控件的右尺寸调整控制柄来手工加长每个静态控件。

8. 对齐控件：打开网格，并使用鼠标拖动控件就位。三个带有电话号码的文字控件应当被对齐，而且被均匀排列。在你所希望的地方设置第一行，然后选中全部三行，并且单击 Dialog（对话）工具栏上的 Align Left（左对齐）和 Space Down（向下间隔）按钮。



如果想使对话的外观均衡，在上面的例子中，还可以扩大 OK 按钮。只需单击该按钮便可选中它，然后根据需要拖动尺寸调整控制柄即可。

9. 测试：单击 Dialog（对话）工具栏上的测试驱动开关，可以看看完成以后的作品是什么样子。如果想返回编辑模式，请单击对话的 OK 按钮，或按 ESC。

保存自己的工作之后，对话编辑器将用修改后的 About（关于）框所对应的新脚本覆盖文件中原来的脚本。

```
IDD_ABOUTBOX DIALOG DISCARDABLE 0, 0, 217,129
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

```
CAPTION "About XYZ SpiffyDemo"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
CONTROL                                IDB_XYZCORP,IDC_STATIC,"Static",
```

```
SS_BITMAP,17,16,85,10
```

```
LTEXT                                "SpiffyDemo Version 1.0",
```

```
IDC_STATIC,21,33,119,8,SS_NOPREFIX
```


续表

LTEXT	"Copyright \251 1998,XYZ Corporation", IDC_STATIC,21,45,119,8
DEFPUSHBUTTON	"OK ",IDOK,166,15,32,40,WS_GROUP
GROUPBOX	"Phone Numbers",IDC_STATIC,17,65,181,56
LTEXT	"Technical Support:\t(206) 555-1212", IDC_STATIC,33,82,140,8
LTEXT	"Customer Service:\t(206) 555-1212", IDC_STATIC,33,94,142,8
LTEXT	"Sales:\t\t\t(206) 555-1212", IDC_STATIC,33,106,141,8

END

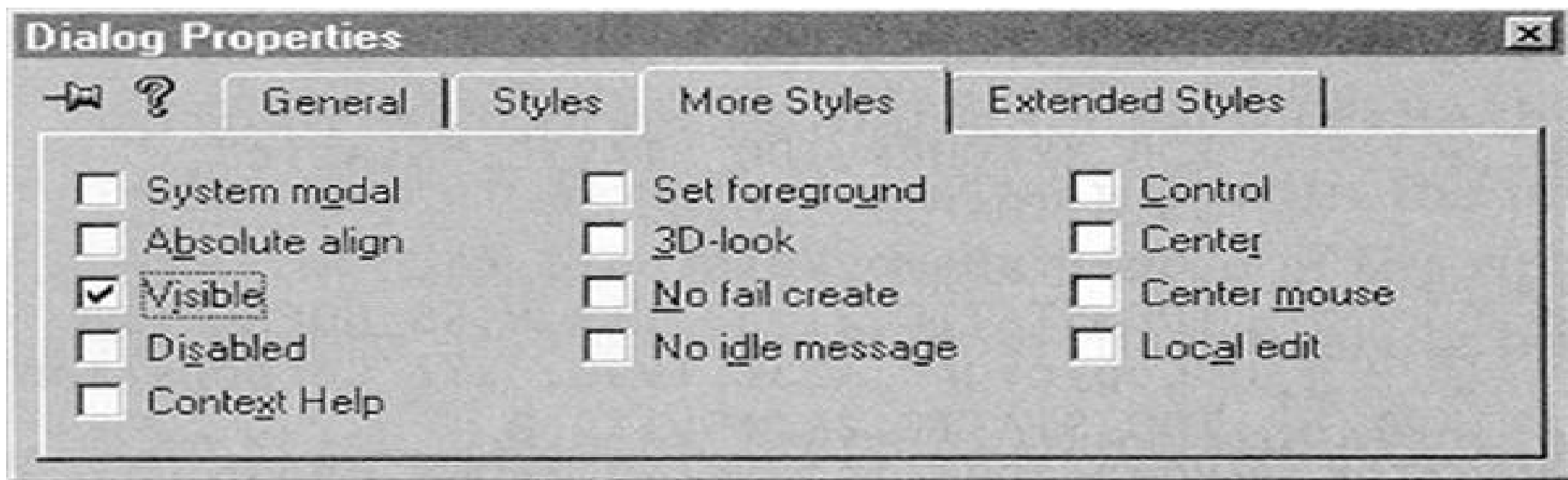
该脚本的第一个 CONTROL 语句引用 IDB_XYZCORP 位图，该位图包含步骤 1 中用图形编辑器创建的公司标志。资源编辑器知道到哪里去找 IOB_XYZCORP 位图，因为图形编辑器已在 RC 文件中其他地方记录了文件名。

```
IDB_XYZCORP      BIPMAP MOVEABLE PORE      "yes\\xyzcorp.bmp"
```

例 2：简单的非模态对话框

About 是模态对话框，直到单击 OK 关闭对话之后，它才允许用户继续在程序中工作。相反，非模态对话框就允许用户在同一程序中切换到另一个窗口，并且继续工作。而对话仍在屏幕上保留，直到被退出。

对话是模态的，还是非模态的，仅仅取决于程序在运行时间创建该对话的方式。你可以用相同的方法设计模态和非模态对话框。只有一点例外：非模态对话框必须具有样式标志 WS_VISIBLE。该标志可以这样设定，在对话工作区中右击，并从上下文菜单中选择 Properties 来调用 Dialog Properties（对话属性）框。然后，打开 More Style（其他风格）选项卡中的 Visible（可见性）复选框：



该操作将把 `WS_VISIBLE` 标志添加到 RC 文件中该对话脚本的 `STYLE` 语句。

C 程序通过调用 `CreateDialog` API 函数或者它的一个变种 (例如, `CreateDialogIndirect`) 来创建一个非模态对话：

```
hDig=CreateDialog(hInst,MAKEINTRESOURCE(IDD_DIALOG1),  
                hwnd,DlgProc);
```

在本例中，`hInst` 是程序的实例句柄，`IDD_DIALOG1` 是 RC 文件中代表该对话脚本的标识符，`hwnd` 是对话拥有者的句柄，`DlgProc` 是指向运行该脚本及接收它的

消息的过程的指针。

使用 MFC 的 C++ 程序用 `CDialog::Create` 函数来创建非模态对话。下面几行假定 `CMyDlg` 类是由 MFC 的 `CDialog` 派生出来的：

```
CMyDly*pDlg=new Cmydlg;
```

```
Ddlg Create(IDD_DIALOG,this);
```

`CDialog::Create` 和 `CDlglog::Domodel` 之间有一个重要的差异。因为后一个函数创建的是模态对话。直到对话关闭，它才会返回。而 `Create` 可以立即返回，在非模态对话保留在屏幕上的时候，允许程序继续运行。对话在程序关闭它之前，将一直存在：

```
delete pDlg;
```

在本节展示的 `Color` 程序中，显示了带有三个滑块控件(也叫轨迹条)的非模态对话，这三个滑块可以调整主窗口背景颜色的红、绿、蓝组成。当你用鼠标或箭头键移动滑块的时候，主窗口的颜色将根据新的颜色组成而变化，颜色值可能在 0 到 255 之间变化。如果视频硬件足够的话，从理论上讲，你可以通过移动滑动栏来显示 $16777216(256^3)$ 种不同颜色中的任一种。图 5-11 显示了带有非模态的 `Color` 对话的程序窗口。

Color 是一个 MFC 程序，但不是用 AppWizard 创建的(下节将显示如何在 AppWizard 应用程序中创建新对话)。Color 项目开始是选中 New 对话 Properties 选项卡中的 Win32 Application 图标。键入 Color 作为项目名，单击 OK。通过单击 Insert (插入) 菜单上的 Resource，并双击资源类型列表中的 Dialog 来启动对话编辑器。新的对话是以出现默认的 OK 和 Cancel 按钮作为开始的；在工作区中选中这些按钮，并按 Delete 来删除它们。

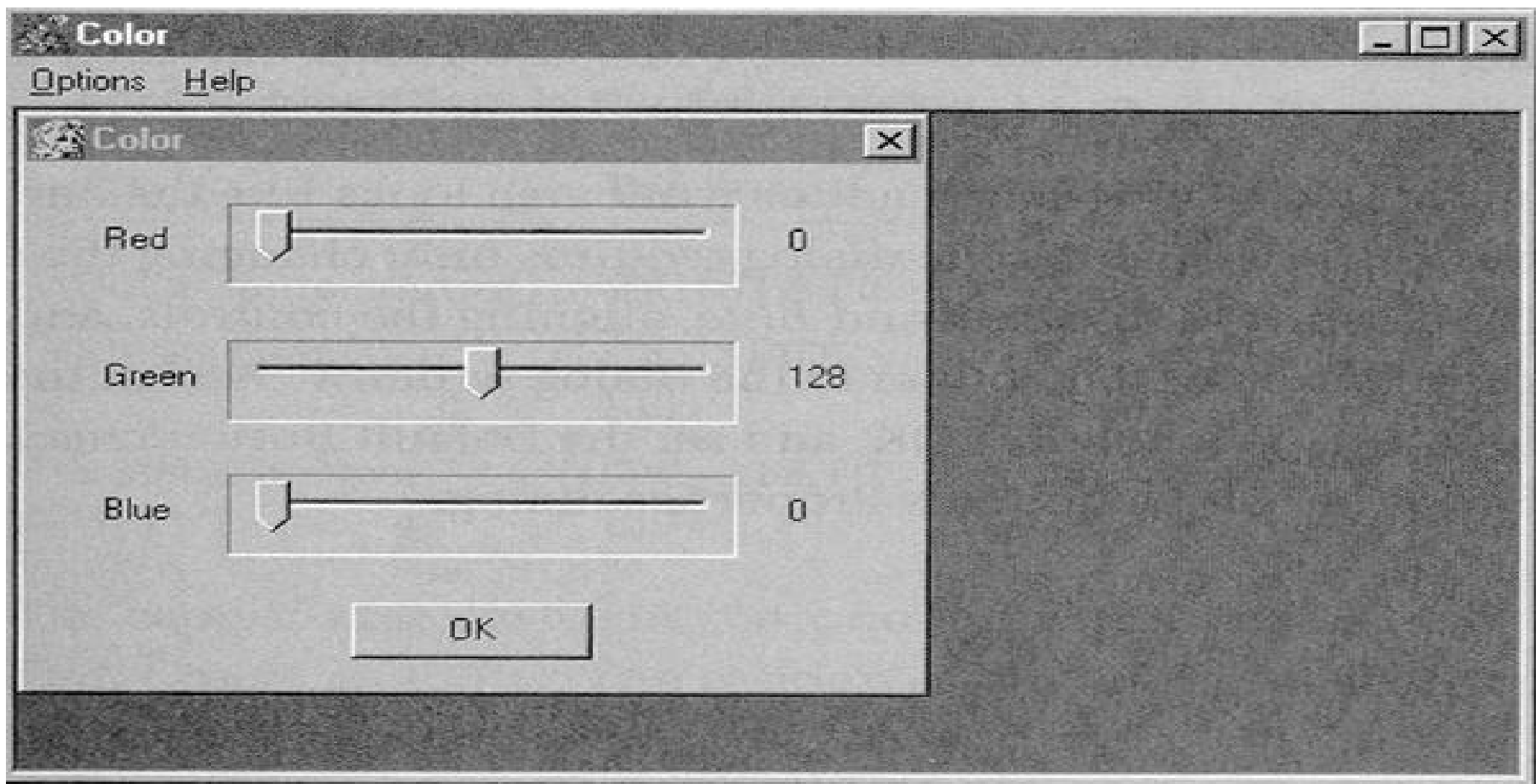
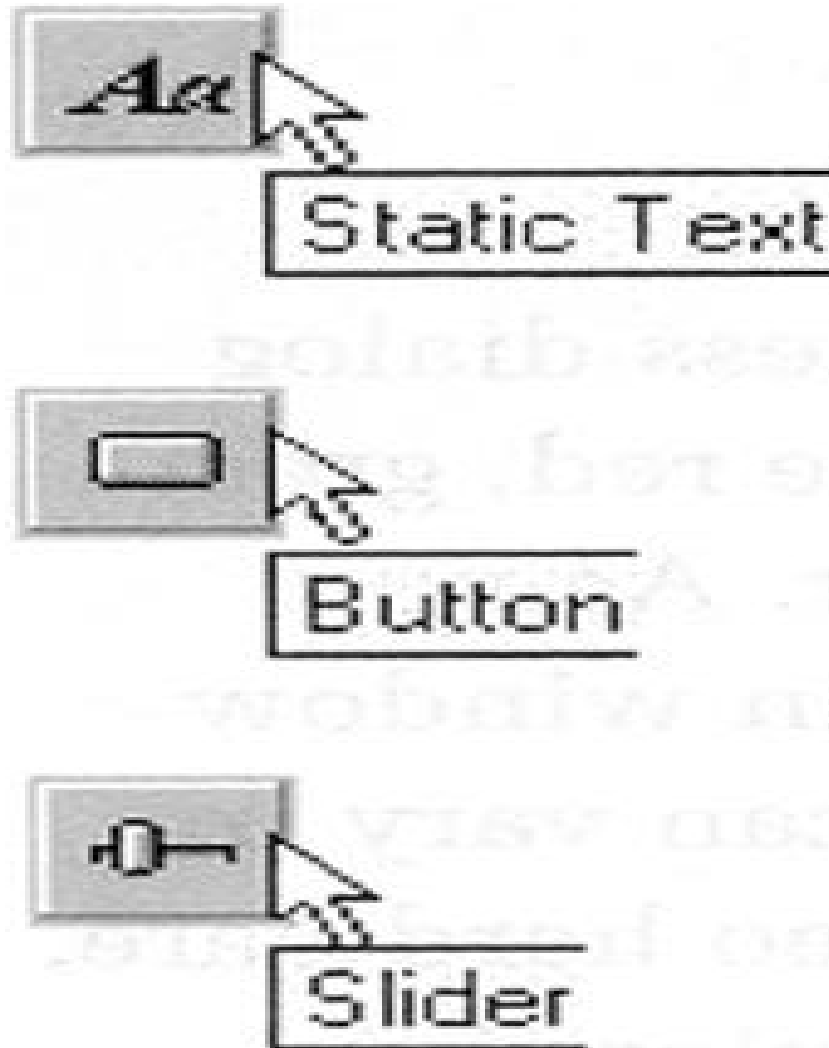


图 5-11 Color 程序

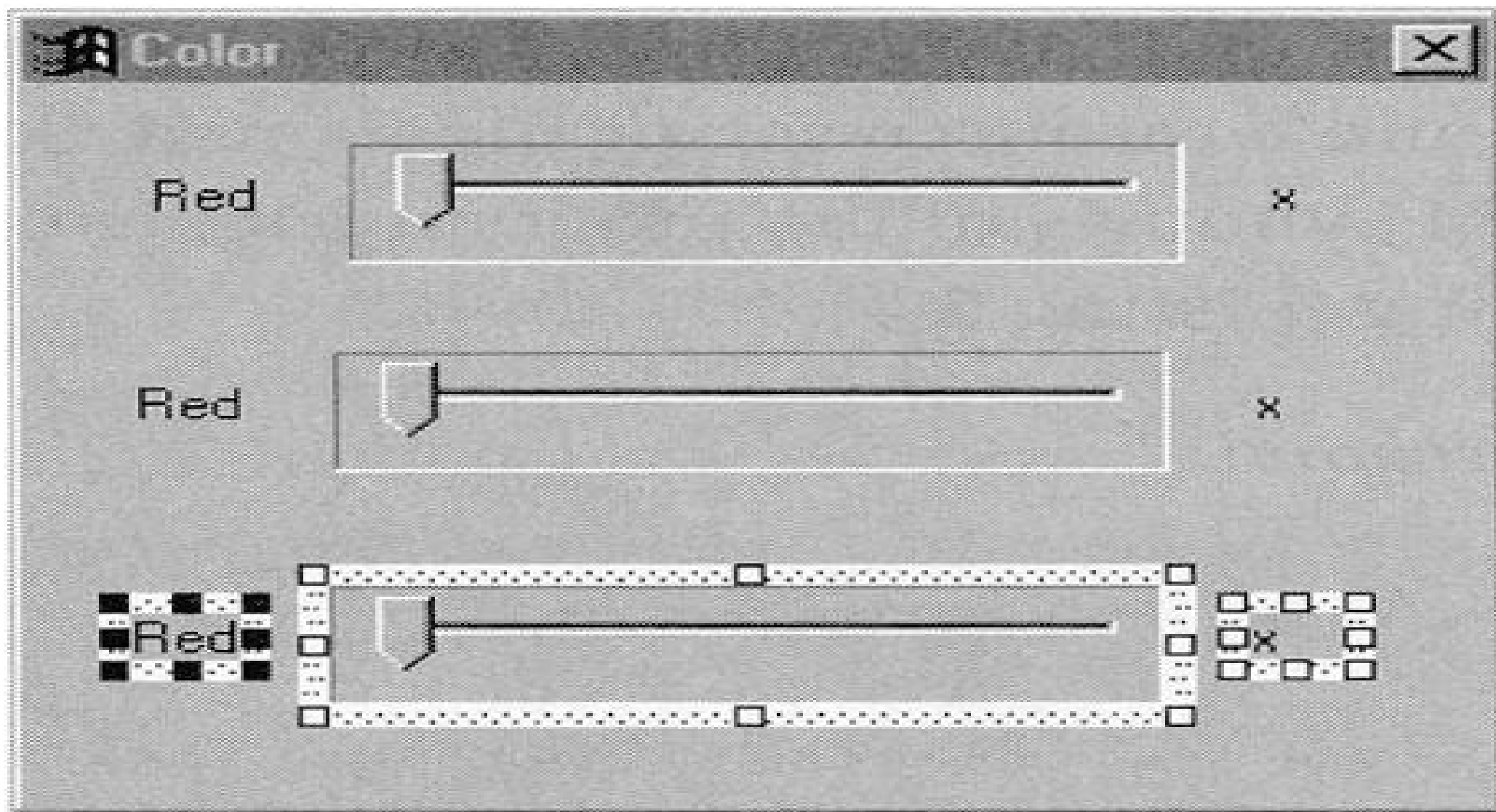


Color 对话框仅仅包含三个不同的类型，它们是由来自编辑器的 Controls（控件）工具栏的 Static Text（静态文本）、Button（按钮）和 Slider（滑块）工具创建的。

滑块栏在对话中形成了三条分界线，在图 5-11 中分别标为 Red, Green 和 Blue。为了创建 Red(上)线，首先需要在两个静态文本控件之间放置一个滑块控件，然后所有的控件都要对齐，并留出间隔。将文本控件的标题初始化为 Red，并在 Text Properties 对话框中键入 X。X 的作用是充当 Color 在运行期间用滑块位置的新值进行覆盖的占位符。为滑块控件分配一个静态边缘的边框和一个具有指针的按钮。这些样式可以在 Slider Properties（滑块属性）对话框中进行设置，可以通过右击对话框中的滑块控件窗口，并选择 Properties 来设置：

1. 在 Style（样式）选项卡中，在拇指样式的 Point（指向）框中选中 Bottom/Right（底部/右边）。
2. 在 Extender Style（扩展样式）选项卡中，单击 Static Edge（静态边缘）复选框。

Green 和 Blue 线的创建方法与 Red 线完全相同，通过将 Red 的三个控件做为一组选中来复制，然后按照前一章中介绍的方法，在按下 Ctrl 的同时拖动该组控件。这时候，对话编辑器工作区看上去与下图类似，完成该对话仅仅需要将第二个和第三个标题分别改为 Green 和 Blue，将控件对齐，并在对话框的底部添加 OK 按钮。为该按钮分配一个标识符 IDOK，并设置该按钮的 Properties（属性）对话框 Styles 选项卡中的 Default Button（默认按钮）复选框。



在复制选定了的控件的时候，对话编辑器将自动为新的控件分配不同的标识符符号，以确保该对话中的每个控件都被独一无二地标识。新的符号与原来的符号具

有相同的名字，只是后面添加了数字。例如，第一行的 Red 滑块控件将分配一个名为 IDC_SLIDE_RED 的标识符。当你通过复制 Red 线来创建 Green 和 Blue 线的时候，编辑器将为新控件分配名为 IDC_SLIDE_RED1 和 IDC_SLIDE_RED2 的标识符(在 Color 程序中，以后将在 Properties（属性）框中把每个控件的标识符改为 IDC_SLIDE_GREEN 的 IDC_SLIDE_BLUE)。在保存这个对话的时候，编辑器将在 Resource.h 文件中为新的标识符写下 #define 语句。结果总结如下：

Color 对话中的控件标识符

	上行	中行	下行
Color 静态控件	IDC_STATIC	IDC_STATIC	IDC_STATIC
滑块控件	IDC_SLIDE_RED	IDC_SLIDE_GREEN	IDC_SLIDE_BLUE
X 静态控件	IDC_STATIC_RED	IDC_STATIC_GREEN	IDC_STATIC_BLUE

Color 静态控件(最左面的控件)的标题应该是 Red，Green 和 Blue。最右边的控件的标题是所有三条线的 X。

Color 的源代码仅需要一个主文件，两个名为 Resource.h 和 Color.h 的头文件，以及一个定义了对话的名为 Color.re 的资源脚本文件。源程序列表如下(见列表 5-1)：

列表 5-1：Color 程序的源代码

```
Resource.h

// *****

//

// Resource.h

//

// *****

#define IDR_MAINFRAME 128

#define IDD_ABOUTBOX 100

#define IDD_COLOR_DIALOG 101
```

```
#define IDC_STATIC_RED          102

#define IDC_STATIC_GREEN        103

#define IDC_STATIC_BLUE         104

#define IDC_SLIDE_RED           105

#define IDC_SLIDE_GREEN         106

#define IDC_SLIDE_BLUE          107

#define IDM_COLOR                200
```

Color.rc

```
// *****

//

// Color.rc

//
```

```
// *****

#include "resource.h"

#include "afxres.h"

AFX_IDI_STD_FRAME      ICON      "res\\Color.ico"

IDR_MAINFRAME MENU

BEGIN

    POPUP "&Options"

    BEGIN

        MENUITEM "&Color",      IDM_COLOR

        MENUITEM SEPARATOR

        MENUITEM "E&xit",      ID_APP_EXIT

    END

END
```

```
POPUP "&Help"
```

```
BEGIN
```

```
    MENUITEM "&About Color...",  ID_APP_ABOUT
```

```
END
```

```
END
```

```
IDD_ABOUTBOX DIALOG 0, 0, 240, 65
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

```
CAPTION "About Color"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
    ICON
```

```
        AFX_IDI_STD_FRAME,IDC_STATIC,10,22,20,20
```

```
    LTEXT
```

```
        "Color Version 1.0",IDC_STATIC,45,10,115,8
```

```
LTEXT        ""Microsoft Visual C++ Programmer's Guide"" ,
```

```
IDC_STATIC,45,26,140,8
```

```
LTEXT        "Copyright \251 1998, Beck Zaratian",
```

```
IDC_STATIC,45,42,115,8
```

```
DEFPUSHBUTTON "OK",IDOK,195,10,35,40,WS_GROUP
```

```
END
```

```
IDD_COLOR_DIALOG DIALOGEX 0, 0, 186, 132
```

```
STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
```

```
CAPTION "Color"
```

```
FONT 8, "MS Sans Serif"
```

BEGIN

LTEXT	"Red",IDC_STATIC,16,15,14,8
LTEXT	"Green",IDC_STATIC,16,49,20,8
LTEXT	"Blue",IDC_STATIC,16,83,14,8
CONTROL	"Slider1",IDC_SLIDE_RED,"msctls_trackbar32", TBS_NOTICKS WS_TABSTOP,42,10,106,21, WS_EX_STATICEDGE
CONTROL	"Slider1",IDC_SLIDE_GREEN,"msctls_trackbar32", TBS_NOTICKS WS_TABSTOP,42,44,106,21, WS_EX_STATICEDGE
CONTROL	"Slider1",IDC_SLIDE_BLUE,"msctls_trackbar32", TBS_NOTICKS WS_TABSTOP,42,78,106,21,


```
WS_EX_STATICEDGE

LTEXT      "x",IDC_STATIC_RED,158,15,12,8

LTEXT      "x",IDC_STATIC_GREEN,158,49,12,8

LTEXT      "x",IDC_STATIC_BLUE,158,83,12,8

DEFPUSHBUTTON      "OK",IDOK,68,111,50,14
```

```
END
```

Color.h

```
// *****

//

// Color.h

//

// *****
```

```
#include "Resource.h"

class CColorApp : public CWinApp

{

public:

    virtual BOOL InitInstance();

};

////////////////////////////////////

// CColorDlg dialog

class CColorDlg : public CDialog

{

public:

    int        nColor[3];
```

```
    BOOL    bCreate;
```

```
    CColorDlg ();
```

```
protected:
```

```
    virtual BOOL OnInitDialog ();
```

```
    virtual void OnOK ();
```

```
    afx_msg void OnCancel ();
```

```
    afx_msg void OnHScroll( UINT nCode, UINT nPos, CScrollBar* pScroll );
```

```
    DECLARE_MESSAGE_MAP()
```

```
};
```

```
////////////////////////////////////
```

```
// CMainFrame
```

```
class CMainFrame : public CFrameWnd
```

```
{  
  
public:  
  
    CColorDlg*    pColorDlg;  
  
    CMainFrame();  
  
    ~CMainFrame();  
  
protected:  
  
    afx_msg void OnAbout();  
  
    afx_msg BOOL OnEraseBkgnd( CDC* pDC );  
  
    afx_msg void OnColor();  
  
    DECLARE_MESSAGE_MAP()  
  
};
```

```
////////////////////////////////////
```

```
// CAboutDlg dialog
```

```
class CAboutDlg : public CDialog
```

```
{
```

```
public:
```

```
    CAboutDlg();
```

```
};
```

```
CAboutDlg::CAboutDlg() : CDialog( IDD_ABOUTBOX )
```

```
{
```

```
}
```

Color.cpp

```
// *****
```

```
//
```

```
// Color.cpp

//

// *****

#define VC_EXTRALEAN

#include <afxwin.h>

#include <afxcmn.h>

#include "Color.h"

CColorApp theApp;

BOOL CColorApp::InitInstance()

{

    m_pMainWnd = new CMainFrame;

    m_pMainWnd->ShowWindow( m_nCmdShow );
```

```
m_pMainWnd->UpdateWindow ();

return TRUE;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// CMainFrame

BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

    ON_COMMAND (IDM_COLOR, OnColor)

    ON_COMMAND (ID_APP_ABOUT, OnAbout)

    ON_WM_ERASEBKGD ()

END_MESSAGE_MAP ()

CMainFrame::CMainFrame ()
```

```
{

    pColorDlg = new CColorDlg;

    Create(NULL, "Color", WS_OVERLAPPEDWINDOW, rectDefault, NULL,

        MAKEINTRESOURCE (IDR_MAINFRAME) );

}

CMainFrame::~CMainFrame ()

{

    delete pColorDlg;

}

void CMainFrame::OnColor()

{

    if (pColorDlg->bCreate)        // If the dialog already exists,
```



```
        pColorDlg->SetFocus();    // give it focus

else                                // Otherwise, create it

{

    if (pColorDlg->Create( IDD_COLOR_DIALOG, this ))

        pColorDlg->bCreate = TRUE;

}

}

BOOL CMainFrame::OnEraseBkgnd( CDC* pDC )

{

    Cbrush        brush;

    CRect         rect;

    COLORREF      rgbBackGnd = RGB ((BYTE) pColorDlg->nColor[0],
```

```
        (BYTE) pColorDlg->nColor[1],  
        (BYTE) pColorDlg->nColor[2]);
```

```
GetClientRect( &rect );
```

```
brush.CreateSolidBrush( rgbBackGnd );
```

```
pDC->FillRect( rect, &brush );
```

```
return TRUE;
```

```
}
```

```
void CMainFrame::OnAbout()
```

```
{
```

```
    CAboutDlg aboutDlg;
```

```
    aboutDlg.DoModal();
```

```
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// CColorDlg dialog
```

```
BEGIN_MESSAGE_MAP (CColorDlg, CDialog)
```

```
    ON_WM_HSCROLL ()
```

```
END_MESSAGE_MAP ()
```

```
CColorDlg::CColorDlg ()
```

```
{
```

```
    nColor[0] = 0;          // Initial color is
```

```
    nColor[1] = 128;        // medium-intensity green
```

```
    nColor[2] = 0;
```

```
    bCreate    = FALSE;    // Haven't created the dialog yet
```

```
}
```

```
BOOL CColorDlg::OnInitDialog ()

{

    CSliderCtrl*    pSlide[3];

    CString        szColorValue;

    int            i;

    for (i=0; i < 3; i++)

    {

        pSlide[i] = (CSliderCtrl*) GetDlgItem( IDC_SLIDE_RED + i );

        pSlide[i]->SetRange( 0, 255 );

        pSlide[i]->SetPos( nColor[i] );

        szColorValue.Format( "%d", nColor[i] );

        SetDlgItemText( i + IDC_STATIC_RED, szColorValue );
    }
}
```

```
    }

    SetIcon( AfxGetApp()->LoadIcon( AFX_IDI_STD_FRAME ), FALSE );

    return TRUE;
}

void CColorDlg::OnHScroll( UINT nCode, UINT nPos, CScrollBar* pScroll )
{
    CSliderCtrl*   pSlide = (CSliderCtrl*) pScroll;

    CString        szColorValue;

    int            i = pSlide->GetDlgCtrlID () - IDC_SLIDE_RED;

    nColor[i] = pSlide->GetPos ();           // Slider position is

    szColorValue.Format( "%d", nColor[i] ); //    component 0-255

    // Write current component value in the "x" static control
```

```
        SetDlgItemText( i + IDC_STATIC_RED, szColorValue );

        AfxGetMainWnd()->Invalidate ();

}

void CColorDlg::OnOK ()

{

    OnCancel ();        // Close dialog when OK Clicked

}

// Close on (1) OK button; (2) Esc key; or (3) dialog's Close command

void CColorDlg::OnCancel ()

{

    if (DestroyWindow ())

        ((CMainFrame*) AfxGetMainWnd())->pColorDlg->bCreate = FALSE;
```

```
}
```

程序的所有主要工作是由两个类来实现的，它们分别是用于主窗口的 CMainFrame，以及一个名为 CColorDlg 的 CDialog 派生类，它的作用是用来驱动非模态对话框。CColorDlg 构造器初始化一个代表主窗口的背景颜色值的数组，它以中等密度的绿色开始。既然窗口背景颜色的红、绿和蓝成份可从 0 变到 255，那么，OnInitDialog 函数可以把每个滑块滚动范围设为从 0 到 255。这就意味着，该范围内的一个整数可代表任何一个时刻的拇指按钮的位置，可以很方便地确定颜色成分的当前值。

当用户移动滑块拇指的时候，CColorDlg::OnHScroll 函数就获得控制权，并指出哪个滑块正在通过调用 GetDlgCtrlID 被移动。该函数然后调用 GetPos 来获得新的拇指位置，将该值写到 nColor 数组和对话框中的静态文字控件之中，并且调用 Invalidate 函数。

对 Invalidate 的调用将导致操作系统向主窗口发送一条 WH_ERASEBKGDND 消息，告诉该窗口为自己重新上色。该消息为像 Color 这样的应用程序提供了一个为自己填上背景颜色的机会。大多数应用程序都忽略 WM_ERASEBKGDND 消息，在这些情形中，Windows 将使用程序在创建窗口时所指定的默认背景颜色不为窗口上色(通常是白的)。Color.cpp 在它的 CMainFrame::OnEraseBKGDnd 函数中截住了 WM_ERASEBKGDND 消息，该函数在存储在 nColor 数组中的红、绿和蓝颜色组成的当前值的基础之上，声明了 COLORREF 对象：

```
COLORREF rgbBackGnd = RGB ((BYTE) pColorDog->nColor[0],  
  
                           (BYTE) pColorDog->nColor[1],  
  
                           (BYTE) pColorDog->nColor[2],
```

OnEraseBkGnd 然后从 rgbBackGnd 中创建画刷，并使用它为窗口上色，并且返回一个值 TRUE。TRUE 返回值用来告诉操作系统，它不应该用默认的颜色清除窗口，因为应用程序已经重新上了背景颜色。

所有这些都是为了解释为什么程序要在创建主窗口之前采取特殊的步骤来创建非模态对话框：

```
CMainFrame::CMainFrame ()  
  
{  
  
    pColorDlg = new CColorDlg;  
  
    Create( NULL, "Color", WS_OVERLAPPEDWINDOW, rectDefault, NULL,  
  
           MAKEINTRESOURCE (IDR_MAINFRAME) );  
  
}
```


在主窗口出现之后，操作系统立即发送 WM_ERASEBKGDND 消息，因此，首先创建 CColorDlg 对象，可以确保 OnEraseDKGnd 函数读到 nColor 数组中有效的颜色值。

例 3：向 AppWizard 程序添加对话

如果你愿意在 AppWizard 的帮助下开始项目，那么，前一节中的 Color 程序看上去就与此没有关系了。实际上，它是有关的。向程序添加新对话将调用相同的步骤，不管是否由 AppWizard 创建这个程序。但 AppWizard 将生成精确表单的代码，它与 Color.cpp 的紧凑样式相匹配可能会有点儿困难。为显示在向 AppWizard 程序中添加新对话中引用了什么，本节将演示使用 AppWizard 范例程序 MFcTree 的必要步骤。

图 5-12 所显示的 MfcTree 对话并不稀奇，仅仅是一些 MFC 类和一个 OK 按钮的树状视图。但它却可以很好地用来演示要说明的问题，在本章稍后和第 13 章，我们还要对其进行改进。

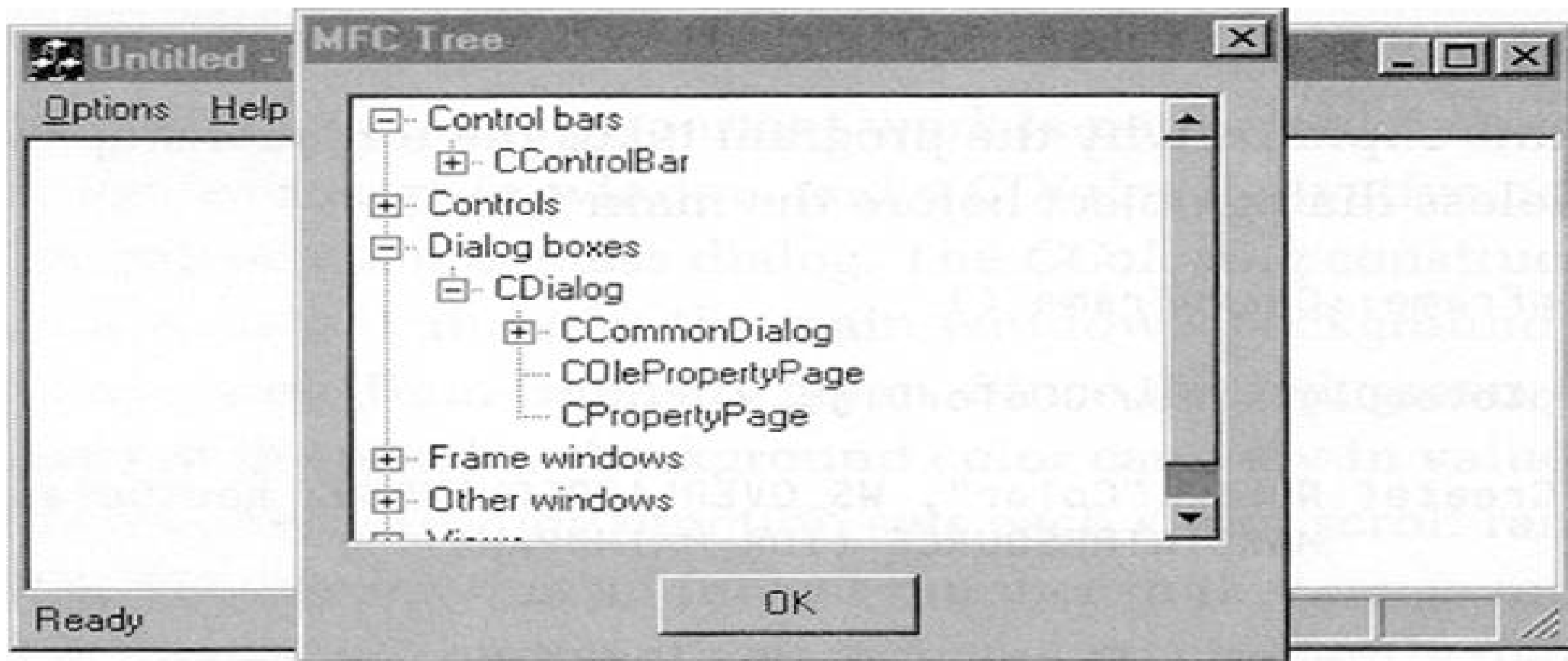


图 5-12 MfcTree 程序

该程序的创建过程可分为五步：

1. 运行 AppWizard 创建 MfcTree 项目。
2. 在对话编辑器中创建对话资源。

3. 向项目添加新的对话类的资源文件。
4. 修改菜单。
5. 向项目资源文件添加必要的源代码。

下面，我们来一步一步地进行。

步骤 1：运行 AppWizard 来创建 MfcTree 项目

在 File（文件）菜单上单击 New，并在 New 对话框中选中 Projects 选项卡。选择 MFC AppWizard (exe) 图标，并将新项目命名为 MfcTree，然后单击 OK 按钮运行 AppWizard。除了如下说明的例外选项外，其他均接受 AppWizard 的默认设置。这些例外是，在步骤 1 中选中 Single Document Interface（单文档界面），在步骤 4 中清除掉用于停靠工具栏和打印支持的复选框。如果想在配套 CD 上保持短文件名，那么默认的 MfcTreeDoc 和 MfcTreeView 文件名应改为 MfcDoc 和 MfcView，但这个步骤完全是可选的。

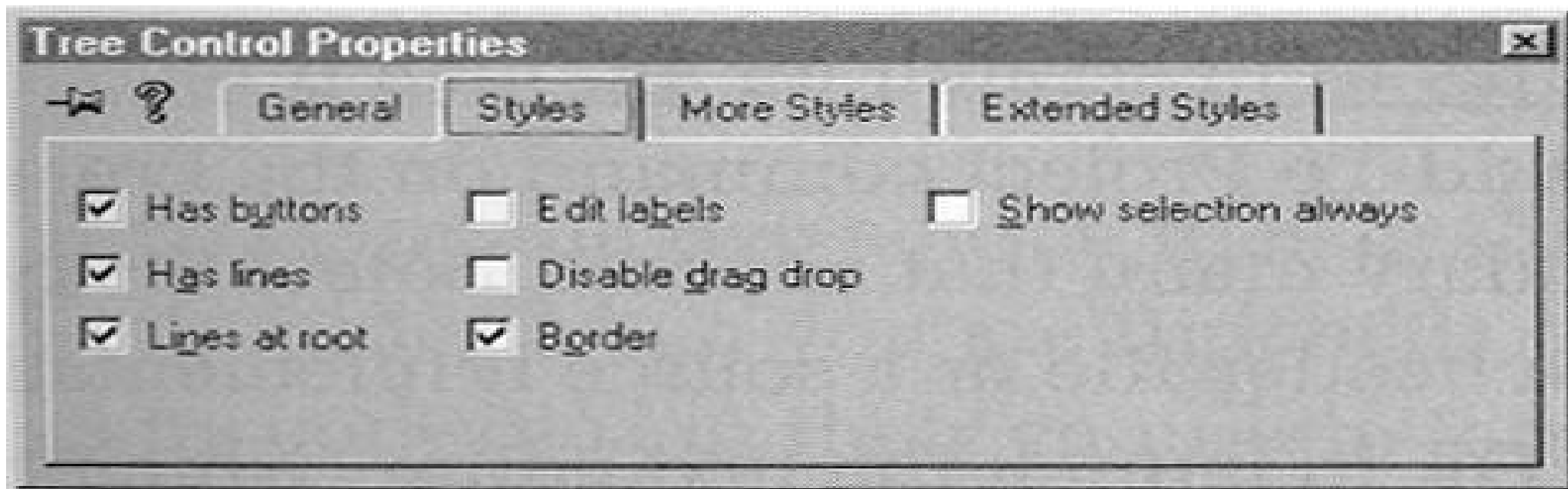
步骤 2：创建 MfcTree 对话

既然 MfcTree 是一个打开的项目，那么，我们就可以设计它的对话资源。通过单

击 Insert（插入）菜单上的 Resource，并双击资源类型列表中的 Dialog 来启动资源对话框。在工作区中选中默认的 Cancel 按钮，并按 Del 键来删除该按钮。在对话工作区内任何地方右击，并从上下文菜单中选择 Properties 来引用 Dialog Properties（属性）框。将对话标题改为 MfcTree，将对话标识符改为 IDC_MFC_DIALOG。

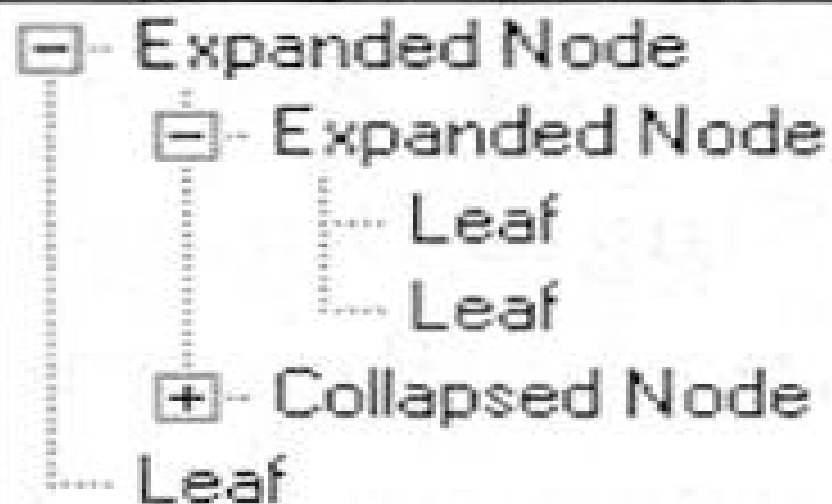


下一步是添加树状视图控件，从 Controls（控件）工具栏中，将 Tree Control（树状控件）工具拖入对话工作区，并根据需要调整对话工作区和控件的大小。单击 View（查看）菜单上的 Properties 来调用 Tree Control Properties（树状控件属性）框，并将标识符符号改为 IDC_MFC_TREE。在 Style 选项卡中，单击 Has Buttons（有按钮），Has Lines（有线）和 Lines At Root（根位置的线）复选框，如下图所示：



这些样式改变将向列表视图添加小的加号或减号图标，当单击这些图标的时候，将折叠或展开列表层次中的各级。使用本章前面介绍的水平间隔工具将 OK 按钮移到对话和中心两个控件的底部。现在，对话在编辑工作区中的样子如图所示：

MFC Tree



OK

从 File (文件) 菜单选择 Save, 将新对话脚本保存到 MfcTree.rc 文件。Visual C++ 自动向 Resource 文件添加代表新标识符的 #define 语句。通过单击 File (文件) 菜单上的 Close 命令关闭对话编辑器。

步骤 3：为 CMfcDlg 对话类添加源文件

我们现在已经拥有了一个新的对话资源，但是，需要从 CDialog 派生的一个类才能运行它。在下一章中，我们将会看到如何使用 ClassWizard 来生成主体的对话类，并自动将它的文件添加到项目中，但目前，我们只能使用文本编辑器来完成相同的工作。为包含新的 CMfcDlg 对话类(列表 5-2)而创建源文件 CMfcDlg.h 和 MfcDlg.cpp。现在我们只需要这些很少的文件，就能将 CMfcDlg 类添加到项目。我们将在步骤 5 中将源代码添加到文件。

如果想从头开始为一个像 MfcTree 这样的开放项目创建新的源文件，请将箭头停在 Project (项目) 菜单的 Add To Project (添加到项目) 命令上，然后，单击随后出现的次级菜单上的 New。选择你想创建的文件，并输入文件名。单击 OK 启动文本编辑器，编辑器中将出现一个空的文档窗口。

列表 5-2：CMfcDlg 类的主体源文件

MfcDlg.h

/// *****

//

```
// MfcDlg.h
```

//

// *****

```
class CMfcDlg : public CDialog
```

{

public:

```
CMfcDlg( CWnd* pParent = NULL );
```

protected:


```
virtual BOOL OnInitDialog();  
  
};
```

MfcDlg.cpp

```
// *****  
  
//  
  
// MfcDlg.cpp  
  
//  
  
// *****  
  
#include "stdafx.h"  
  
#include "MfcTree.h"  
  
#include "MfcDlg.h"  
  
CMfcDlg::CMfcDlg( CWnd* pParent ) :
```

```
CDialog( IDD_MFC_DIALOG, pParent )
```

```
{
```

```
}
```

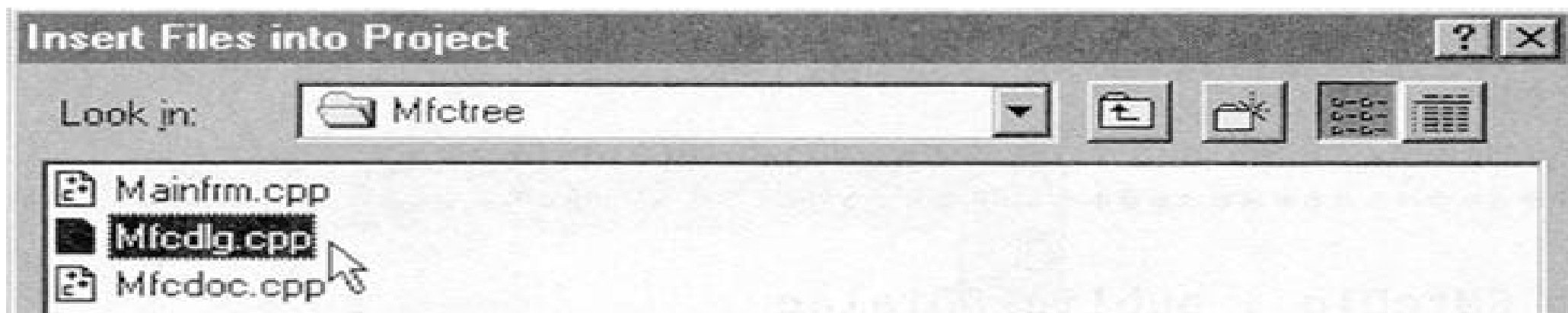
```
BOOL CMfcDlg::OnInitDialog()
```

```
{
```

```
    return TRUE;
```

```
}
```

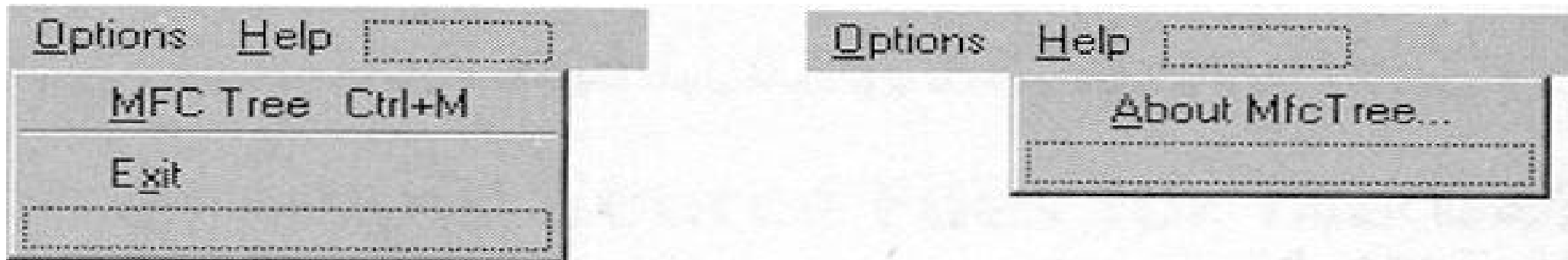
如果你在文本编辑器中键入了两个源文件，完成之后，一定要把它们保存到项目文件夹。另一种方法是，你可以从配套 CD 上的 Chapter.os\MfcTree 文件夹中复制这两个文件的完整形式。复制文件需要你将它们手工添加到项目。在弹出的次级菜单上选择 Add To Project，并单击 Files，然后双击文件列表中的 MfcDlg.cpp 文件，将它添加到项目之中。



不必用相同的方法来添加 `MfcDlg.h` 文件，因为 Visual C++ 将头文件识别为 `MfcDlg.cpp` 的相关文件。

步骤 4：修改菜单

`MfcTree` 需要一个菜单命令来调用对话，但是，不需要 AppWizard 放在菜单上的其他命令。使用第 4 章中介绍的 Visual C++ 菜单编辑器将 `MfcTree` 的菜单修改成如下的样子。



在菜单编辑器中，双击 Workspace（工作空间）窗口的 ResourceView 窗格中的 IDR_MAINFRAME 菜单标识符可以打开资源。在菜单编辑器中，双击 MFC Tree 菜单来显示 Menu Item Properties（菜单项属性）框。键入 IDM_OPTION_MFC 作为项目的标识符，然后按 ENTER 键。

步骤 5：添加需要的源代码

如果按照步骤 3 中所列出的代码键入了 MfcDlg.cpp 文件，该文件仅仅包含一个类构造器和一个叫做 OnInitDialog 的存根函数。OnInitDialog 函数需要用下面显示的带阴影的代码，来初始化树形视图控件。正如你所看到的，它可以用许多指令来初始化树形视图。如果你通过自己创建 MfcTree 来顺着这些步骤进行操作，那么没有必要键入所有的行。只需键入头一组，或者从配套 CD 的 Chapter.os\MfcTree 文件夹中的 MfcDlg.cpp 源文件中进行剪切和粘贴即可。

OnInitDialog 函数用来初始化一个名为 **pTree** 的指针，该指针指向的是作为 **IDC_MFe_TRUE** 标识的对话框的树形窗口控件。该函数然后重复调用 **CtreeCtrl::InsertItem** 函数来向控制添加一个包含有从 **CWnd** 派生的 **MFe** 类的层次列表。项目被插入到层次中比它的父层低一级的列表之中，第二个参数或者是以前为父层级调用 **InsertItem** 所返回的 **MTREEITEM** 值，或者是值 **TVI_ROOT**(如果没有父层的话)。 **TV1_SORT** 标志指示树形视窗控件将该项目按字母顺序排序：

```
BOOL CMfcDlg::OnInitDialog()
```

```
{
```

```
HTREEITEM    hRoot, hLevel1, hLevel2, hLevel3, hLevel4, hLevel5;

CTreeCtrl*    pTree = (CTreeCtrl*) GetDlgItem( IDC_MFC_TREE );

hRoot  = pTree->InsertItem( "Frame windows", TVI_ROOT, TVI_SORT );

hLevel1 = pTree->InsertItem( "CFrameWnd", hRoot );

hLevel2 = pTree->InsertItem( "CMDIChildWnd", hLevel1 );

hLevel2 = pTree->InsertItem( "CMDIFrameWnd", hLevel1 );

hLevel2 = pTree->InsertItem( "COlePFrameWnd", hLevel1 );
```

```
hRoot    = pTree->InsertItem( "Control bars", TVI_ROOT, TVI_SORT );

hLevel1 = pTree->InsertItem( "CControlBar", hRoot );

hLevel2 = pTree->InsertItem( "CDialogBar", hLevel1 );

hLevel2 = pTree->InsertItem( "COleResizeBar", hLevel1 );

hLevel2 = pTree->InsertItem( "CStatusBar", hLevel1 );

hLevel2 = pTree->InsertItem( "CToolBar", hLevel1 );

hRoot    = pTree->InsertItem( "Other windows", TVI_ROOT, TVI_SORT );

hLevel1 = pTree->InsertItem( "CPropertySheet", hRoot );

hLevel1 = pTree->InsertItem( "CSplitterWnd", hRoot );

hRoot    = pTree->InsertItem( "Dialog boxes", TVI_ROOT, TVI_SORT );

hLevel1 = pTree->InsertItem( "CDialog", hRoot );

hLevel2 = pTree->InsertItem( "CCommonDialog", hLevel1 );
```

```
hLevel3 = pTree->InsertItem( "CColorDialog", hLevel2 );

hLevel3 = pTree->InsertItem( "CFileDialog", hLevel2 );

hLevel3 = pTree->InsertItem( "CFindReplaceDialog", hLevel2 );

hLevel3 = pTree->InsertItem( "CFontDialog", hLevel2 );

hLevel3 = pTree->InsertItem( "COleDialog", hLevel2 );

hLevel4 = pTree->InsertItem( "COleBusyDialog", hLevel3 );

hLevel4 = pTree->InsertItem( "COleChangeIconDialog", hLevel3 );

hLevel4 = pTree->InsertItem( "COleChangeSourceDialog", hLevel3 );

hLevel4 = pTree->InsertItem( "COleConvertDialog", hLevel3 );

hLevel4 = pTree->InsertItem( "COleInsertDialog", hLevel3 );

hLevel4 = pTree->InsertItem( "COleLinksDialog", hLevel3 );

hLevel5 = pTree->InsertItem( "COleUpdateDialog", hLevel4 );
```

```
hLevel4 = pTree->InsertItem( "COlepasteSpecialDialog", hLevel3 );
```

```
hLevel4 = pTree->InsertItem( "COlePropertiesDialog", hLevel3 );
```

```
hLevel3 = pTree->InsertItem( "CPageSetupDialog", hLevel2 );
```

```
hLevel3 = pTree->InsertItem( "CPrintDialog", hLevel2 );
```

```
hLevel2 = pTree->InsertItem( "COlePropertyPage", hLevel1 );
```

```
hLevel2 = pTree->InsertItem( "CPropertyPage", hLevel1 );
```

```
hRoot    = pTree->InsertItem( "Views", TVI_ROOT, TVI_SORT );
```

```
hLevel1 = pTree->InsertItem( "CView", hRoot );
```

```
hLevel2 = pTree->InsertItem( "CCtrlView", hLevel1 );
```

```
hLevel3 = pTree->InsertItem( "CEditView", hLevel2 );
```

```
hLevel3 = pTree->InsertItem( "CListView", hLevel2 );
```

```
hLevel3 = pTree->InsertItem( "CRichEditView", hLevel2 );
```



```
hLevel3 = pTree->InsertItem( "CTreeView", hLevel2 );

hLevel2 = pTree->InsertItem( "CScrollView", hLevel1 );

hLevel3 = pTree->InsertItem( "CFormView", hLevel2 );

hLevel4 = pTree->InsertItem( "CDaoRecordView", hLevel3 );

hLevel4 = pTree->InsertItem( "CRecordView", hLevel3 );

hRoot   = pTree->InsertItem( "Controls", TVI_ROOT, TVI_SORT );

hLevel1 = pTree->InsertItem( "CAnimateCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CButton", hRoot );

hLevel2 = pTree->InsertItem( "CBitmapButton", hLevel1 );

hLevel1 = pTree->InsertItem( "CComboBox", hRoot );

hLevel1 = pTree->InsertItem( "CEdit", hRoot );

hLevel1 = pTree->InsertItem( "CHeaderCtrl", hRoot );
```

```
hLevel1 = pTree->InsertItem( "CHotKeyCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CListBox", hRoot );

hLevel2 = pTree->InsertItem( "CCheckListBox", hLevel1 );

hLevel2 = pTree->InsertItem( "CDragListBox", hLevel1 );

hLevel1 = pTree->InsertItem( "CListCtrl", hRoot );

hLevel1 = pTree->InsertItem( "COleCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CProgressCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CRichEditCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CScrollBar", hRoot );

hLevel1 = pTree->InsertItem( "CSliderCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CSpinButtonCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CStatic", hRoot );
```

```
hLevel1 = pTree->InsertItem( "CStatusBarCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CTabCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CToolbarCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CToolTipCtrl", hRoot );

hLevel1 = pTree->InsertItem( "CTreeCtrl", hRoot );


return TRUE;

}
```

尽管上面的代码目前可以工作，但是，这是十分不够的。第 13 章中所引入的 MfcTree3 程序显示了一个更加清楚的方法，来实现对 InsertItem 的一系列调用。

下一步，在文本编辑器中编辑文件 MfcTree.cpp 和 MfcTree.h，并添加一个叫做 OnMfcTree 的函数。当用户单击了程序的 Options 菜单上的 MFC TREE 命令的时候，OnMfcTree 处理函数就获得了控制。如果想添加 OnMfcTree，请通过在 Workspace(工作空间)窗口的 FileViews 窗格中双击其文件名，来打开 MfcTree.cpp 文档。在靠近源代码开始处插入如下代码：

```
#include "MfcDlg.h"
```

另外，编辑 MfcTree.cpp 中的消息映射，使它成为下面的样子：

```
BEGIN_MESSAGE_MAP(CMfcTreeApp, CWinApp)
```

```
//{{AFX_MSG_MAP(CMfcTreeApp)
```

```
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
```

```
ON_COMMAND(IDM_OPTIONS_MFC, OnMfcTree)
```

```
//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
```

将 OnMfcTree 函数添加到 MfcTree.cpp：

```
////////////////////////////////////
```

```
// CMfcTreeApp message handlers
```

```
void CMfcTreeApp::OnMfcTree()
```

```
{  
  
    CMfcDlg mfcDlg;  
  
    mfcDlg.DoModal();  
  
}
```

并在 MfcTree.h 文件中声明它：

```
// Implementation
```

```
//{{AFX_MSG(CMfcTreeApp)
```

```
afx_msg void OnAppAbout();
```

```
afx_msg void OnMfcTree();
```

```
//}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()
```

```
};
```

现在就准备创建 MfcTree 了。单击 Build 菜单上的 Set Active Configuration (设置有效配置) 命令, 来选中 Win32 Release 目标, 然后就可以创建和测试这个应用程序。

基于对话的应用程序

MfcTree 的所有操作都在对话中而不是在主窗口中进行。如果程序将主窗口完全隐藏起来, 仅仅显示对话, 那么将更加方便, 这可以省掉用户单击菜单命令来调用窗口的麻烦。以前, 你曾经看到过这样的基于对话的应用程序, 即 Windows 提供的 Character Map (字符映射表), Calculator (计算器) 和 Phone Dialer (电话拨号程序) 实用程序, 都是用来显示一个程序如何有效地通过代替主窗口的单个对话来与用户进行有效联系的例子。在本节中, 我们将看到如何用 Visual C++ 编写基于对话的应用程序, 并创建一个范例程序来演示这项技术。

用 C 编写的基于对话的应用程序不初始化 WNDCLASS 或 WNDCLASSEX 结构, 不调用 RegisterClass 来注册窗口类。不调用 CreateWindow 来创建主窗口。它不调用 ShowWindow 和 UpdateWindow, 甚至没有调用 GetMessage 和 DispatchMessage 的消息循环。它不需要这些东西, 与用户的所有交互都在对话中发生, Windows 可以处理它, 程序仅仅需要创建类似于下面的对话即可:

```
int WINAPI WinMain( HINSTANCE hInst, HINSTANCE hInstPrev,
```

```
LPSTR szCmdline, int nCmdShow)

{

    DialogBox(hInst, MAKEINTRESOURCE (IDD_DIALOG),

              NULL,DlgProc);

    return( 0 );

}
```

在这个代码段中，IDD_DIALOG 值用来标识该程序的 RC 文件中的对话脚本。DlgProc 是一个指针，它指向的是接收像 WM_INITDIALOG 和 WM_COMMAND 这样的系统消息的过程。

MFC 程序没有可见的 WinMain 函数或消息，因此，有关对话界面的代码数量减少了许多。但 MFC 提供了 AppWizard 的优点，它可以为基于对话的应用程序生成模板形式的代码。

例 4：MfcTree 的基于对话的版本

AppWizard 为基于对话的应用程序所编写的代码比它为一般的应用程序所生成

的代码更加清晰，更加易于理解。在 AppWizard 的 Step1(如图 5-13 所示)中单击 Dialog Based 单选按钮，将导致 AppWizard 为两个类创建源文件，一个为应用程序，另一个为对话对象。

AppWizard 为仅包含一个 OK 按钮、一个 Cancel 按钮以及一个带有警告“Place dialog controls here (将对话控件放在这里)”的静态控件的对话窗口生成资源脚本。如果你在 AppWizard 的步骤 2 中单击了 Content-Sensitive Help (与内容相关的帮助) 复选框的话。该对话还包括一个 Help 按钮 (如果想了解有关的详细信息，请参阅第 2 章)。这种方法的思路是在创建项目之后，为你运行对话编辑器，并将应用程序需要的控件添加到对话中。

在 Visual C++ 版本 6.0 中，当 AppWizard 完成之后，对话编辑器将自动启动。在以前的版本中，你必须自己通过双击 ResourceView 窗口中的对话标识符才能启动。

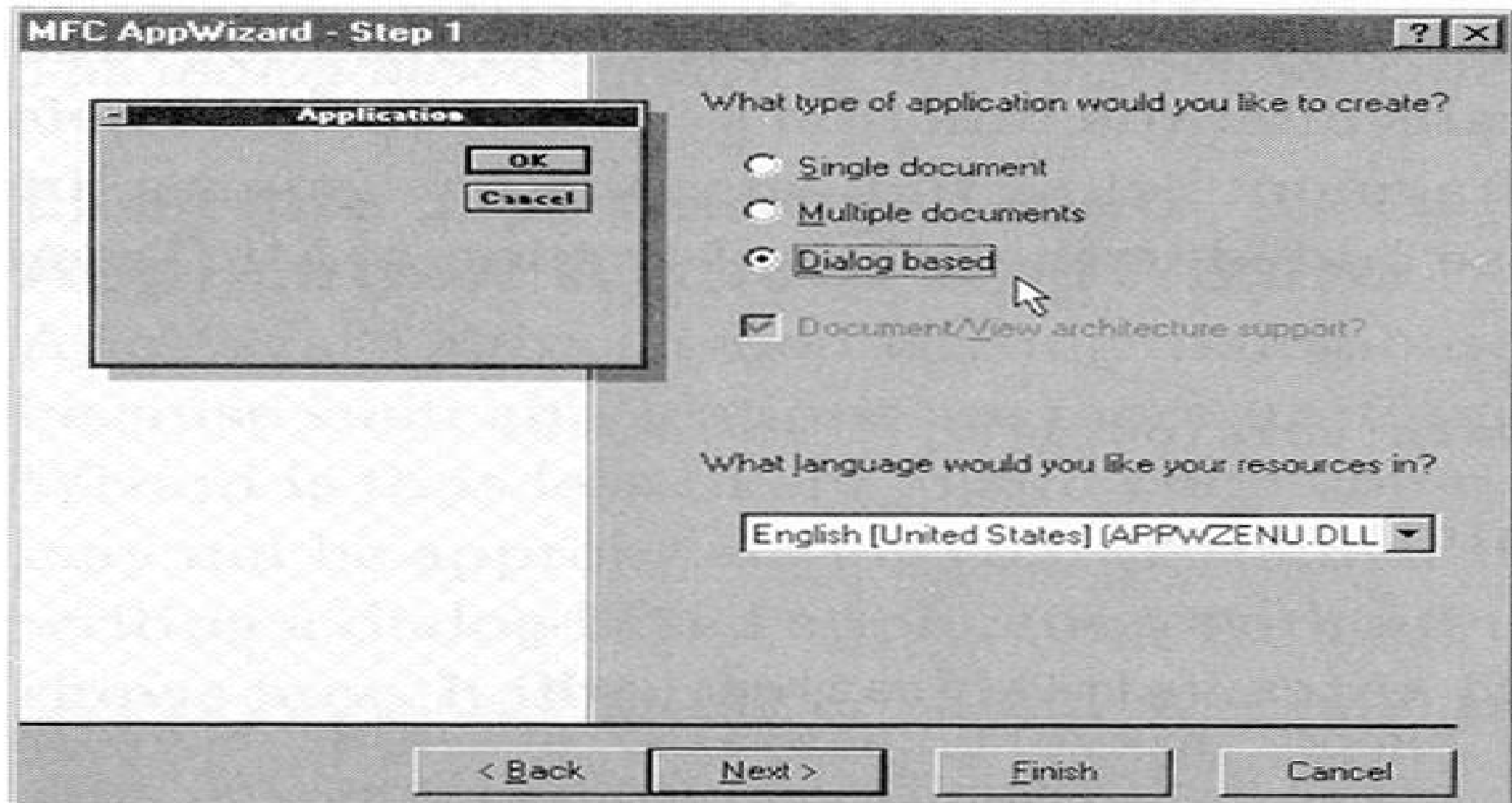


图 5-13 在 AppWizard 中创建基于对话的应用程序

用 AppWizard 创建 MfcTree 的基于对话的版本只需三个步骤,与前一节中的五个步骤对比如下：

步 骤	主 窗 口 版 本	基 于 对 话 的 版 本
1 运 行 AppWizard 创 建 项 目	√	√
2 创 建 或 修 改 对 话	√	√
3 插 入 新 对 话 类 的 源 文 件	√	
4 修 改 菜 单	√	
5 添 加 所 必 需 的 源 代 码	√	√

为对话创建新类的步骤 3 和修改主菜单的步骤 4,在 MfcTree 基于对话的版本中是不需要的。AppWizard 为对话类自动生成主体源文件,清除主窗口也取消了对菜单的需要。甚至 AppWizard 所生成的源代码也比较易于修改,因为你仅仅需要编辑 CMfDlg::OnInitDialog 函数,在该函数的 TODO 行后添加 InsertItem 调用和以前一样,下面代码段中的灰色阴影表示是新的源程序行,它们与以前介绍的添加到 MfcTree 的原始版本中的代码是一样的：

```
BOOL CMfDlg::OnInitDialog()
```

```
{  
  
    Cdialog::OnInitDialog();  
  
    .  
  
    .  
  
    .  
  
    // TODO: Add extra initialization here
```

```
HTREEITEM    hRoot, hLevel1, hLevel2, hLevel3, hLevel4, hLevel5;  
  
CTreeCtrl*    pTree = (CTreeCtrl*) GetDlgItem( IDC_MFC_TREE );  
  
hRoot    = pTree->InsertItem( "Frame windows", TVI_ROOT, TVI_SORT );  
  
hLevel1 = pTree->InsertItem( "CFrameWnd", hRoot );  
  
hLevel2 = pTree->InsertItem( "CMDIChildWnd", hLevel1 );  
  
.  
  
.  
  
.
```

```
//Add the rest of the source code here
```

```
}
```

图 5-14 显示了 MfcTree，这是它基于对话的版本。从本质上讲，除了主窗口已被清除，对话标题栏现在包含应用程序图标和系统菜单，并且程序更加易于创建和使用之外，它与图 5-12 中所显示的原始版本是同一个程序。如果你想自己试试这个新版本，可以在配套 CD 上 Chapteros\MfcTree2 文件类中找到它。

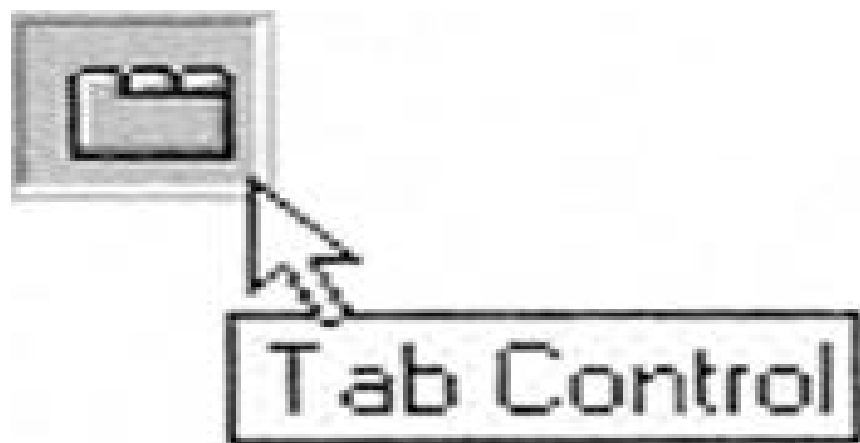


图 5-14 MfcTree 程序基于对话的版本

例 5：不用 AppWizard 实现的基于对话的应用程序

在创建基于对话的应用程序的时候，AppWizard 起了很大的作用，因为这样的应用程序在方法上很一致，因此 AppWizard 添加你不想要的代码的可能性就很小。但 AppWizard 可能仍然满足不了你的特殊需要，如果你对不用 AppWizard 编写基于对话的应用程序感兴趣，那么本节将告诉你如何去做。它用名为 DirList1 的范例程序来进行说明，该程序的作用是在对话中显示一个目录列表。但是为了更加有趣，DirList1 展现给用户的对话是一个属性表，通常叫做多选项卡的对话框。

属性表可以向用户展示大量的信息，这就很好地解决了对话框过分拥挤的问题，但从本质上讲，属性表也仅仅是一个对话，或者说是一系列叫做页面的对话，这些页面一个盖着另一个，你可以像设计其他对话一样用对话编辑器设计属性表的每页。RC 文件中一个页面的属性脚本看上去与普通的对话是一样的，只是属性表的所有页面都具有相同的大小。当程序调用 PropertySheet API 函数创建 MFC CPropertySheet 对象的时候，Windows 将属性表页面做为对话来显示。



偶然情况下，你可能会遇到对话编辑器的 Controls(控件)工具栏中的 Tab Control (选项卡控件) 的工具。这个控件显示一组可以在对话中充当属性表的分选项卡的页面，但并不将你的对话转化为属性表。

图 5-15 显示了 DirList1 属性对话的第一个和第二个页面，标签分别为 Location 和 Date。

Date 选项卡利用新的 Date-Time Picker (日期-时间选取器) 控件，允许用户通过弹出的日历选择日期，尽管 Date 和 Size 页面可以正常工作，但是，该页面的存在仅仅是为了说明的目的，对程序的这个版本并没有影响(第 7 章中将要介绍的 DirList2 项目，利用了 Date 和 Size 页)。DirList1 中的重要活动在 Location(位置)页中发生，它在一个列表框控件中显示一个目录列表。DirList1 通过调用 CWnd::DlgDirList 函数来用目录列表填充这个列表框。

```
DlgDirListDir,IDC_LIST,IDC_DIRPATH,DOL_ALL);
```

参数 pDir 指向以 null(空)结束的字符串，该字符串包含了目录路径。常数 IDC_LIST 用来标识显示目录列表框控件，IDC_DIRPATH 用来标识一个静态控件，DLgDirList 把路径字符串写到这个控件里面。DLL_ALL 是一个在源代码中定义的常数，它将标志 DDL_DRIVE，DDL_DIRECTORT 和 DDL_HIDDEN 组合了起来。这些标志告诉 DlgDirList，在目录列表中包括驱动器、子目录和隐藏的文件。单击列表中的驱动器或子目录，将改变路径，并刷新列表。利用 Location 选项卡中标签为 Up 1 Directory Level（向上一级目录）的大按钮，用户可以回到上一级目录。

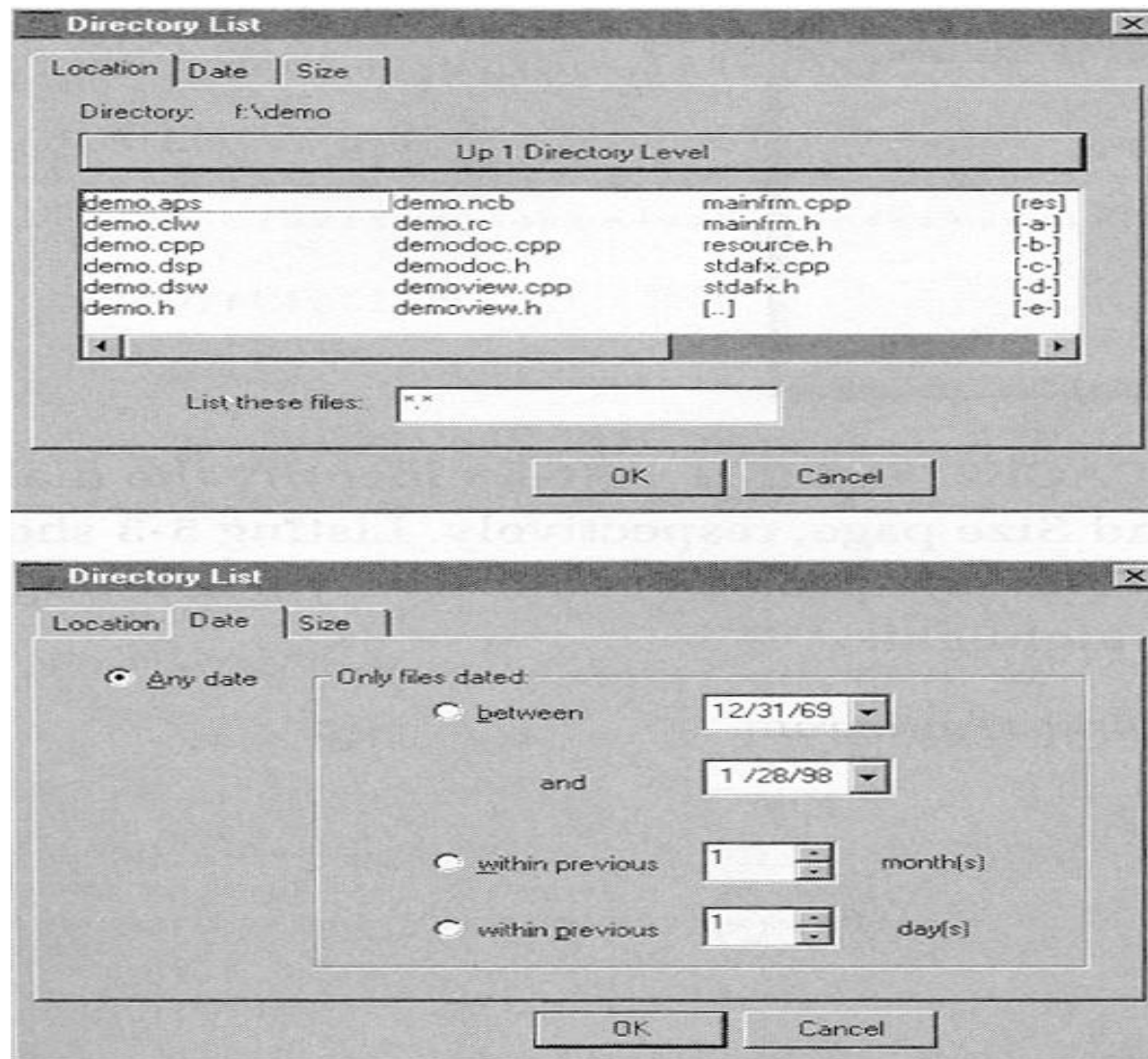
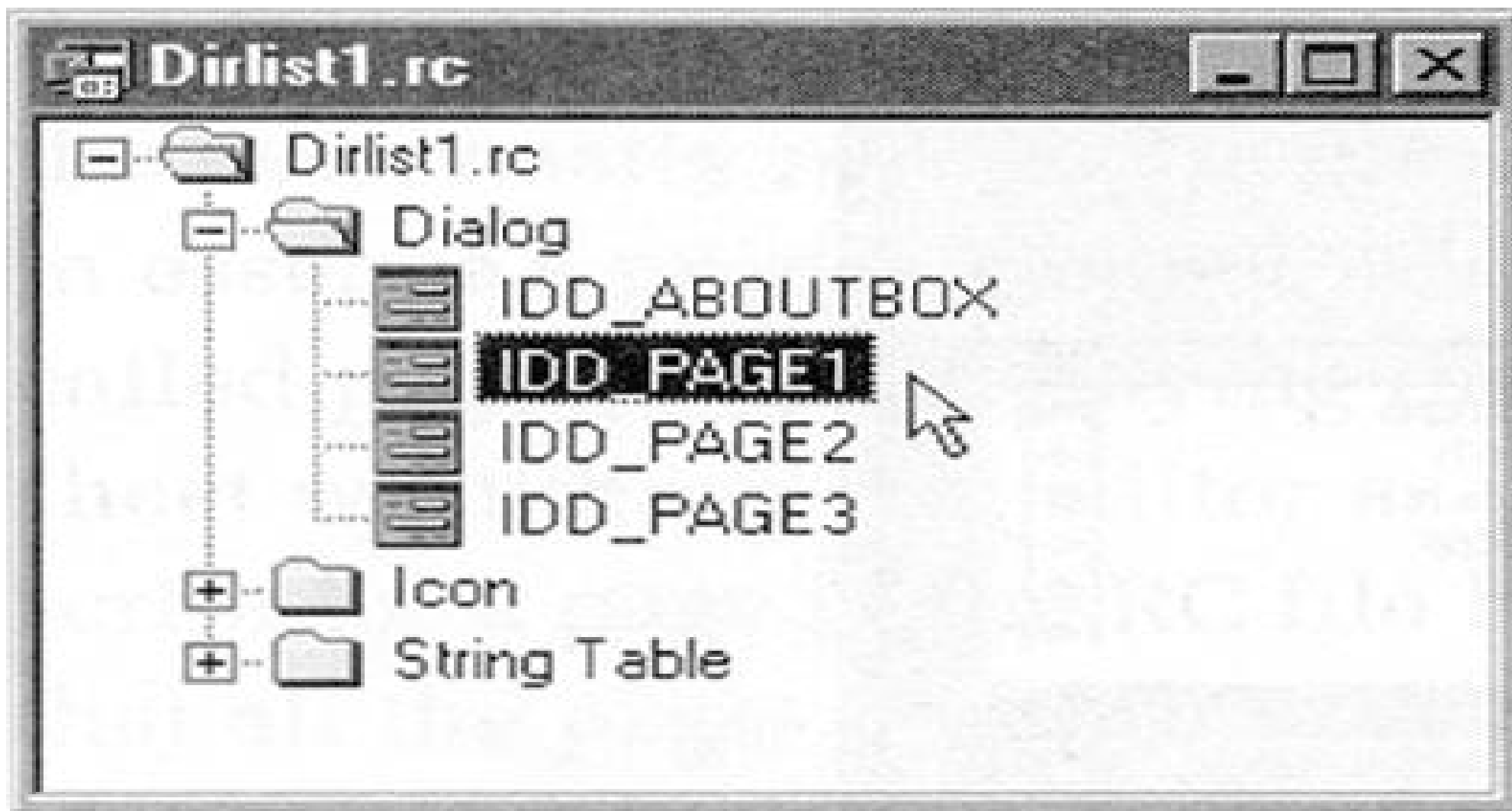


图 5-15 DirList1 程序的 Location 和 Date 选项卡

DirList1 中的属性表对话仅用到了五种控件类型：推压按钮、单选按钮、列表框、编辑控件和微调框。如果你对这些控件的安装方式感兴趣，请打开 DirList.rc 文件，并且通过双击下图所示的一个对话标识符来启动对话编辑器。



IDD_PAGE1, IDD_PAGE2 和 IDD_PAGE3 分别用来标识对话的 Location 页, Date 页和 Size 页。列表 5-3 显示了 DirList1 项目的源代码。

列表 5-3 DirList1 的源程序文件

```
Resource.h

// *****

//

// Resource.h

//

// *****

#define IDD_ABOUTBOX          100

#define IDD_DIRLIST           101

#define IDI_DRIVE              102

#define IDI_FOLDER             103

#define IDI_FILE               104
```

#define IDI_APPICON	105
#define IDC_DIRPATH	110
#define IDC_LIST	111
#define IDC_BUTTON1	120
#define IDC_EDIT1	121
#define IDC_EDIT2	122
#define IDC_RADIO1	123
#define IDC_RADIO2	124
#define IDC_RADIO3	125
#define IDC_RADIO4	126
#define IDC_SPIN1	127
#define IDC_SPIN2	128

```
#define IDC_DATETIME1          130

#define IDC_DATETIME2          131

#define IDD_PAGE1              1001

#define IDD_PAGE2              1002

#define IDD_PAGE3              1003
```

DirList1.rc

```
// *****

//

// DirList1.rc

//

// *****

#include "resource.h"
```

```
#include "afxres.h"
```

```
IDI_APPICON        ICON        "res\\dirlist.ico"
```

```
STRINGTABLE
```

```
BEGIN
```

```
    IDD_ABOUTBOX        "&About DirList..."
```

```
END
```

```
IDD_ABOUTBOX DIALOG 0, 0, 240, 65
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
```

```
CAPTION "About Directory List"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
    ICON
```

```
        IDI_APPICON, IDC_STATIC, 10, 22, 20, 20
```

```
LTEXT                "Directory List Version 1.0",IDC_STATIC,45,10,115,8

LTEXT                ""Microsoft Visual C++ Programmer's Guide"",

IDC_STATIC,45,26,140,8

LTEXT                "Copyright \251 1998, Beck Zaratian",

IDC_STATIC,45,42,115,8

DEFPUSHBUTTON        "OK",IDOK,195,10,35,40,WS_GROUP
```

```
END
```

```
////////////////////////////////////
```

```
//
```

```
// Property page 1 - "Location"
```

```
//
```

```
IDD_PAGE1 DIALOG 0, 0, 282, 135
```


STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION

CAPTION "Location"

FONT 8, "MS Sans Serif"

BEGIN

LTEXT "Directory:",IDC_STATIC,10,5,35,10

LTEXT "",IDC_DIRPATH,50,5,220,10

PUSHBUTTON "Up 1 Directory Level",IDC_BUTTON1,10,17,260,15

LISTBOX IDC_LIST,9,38,260,73,
LBS_STANDARD | LBS_MULTICOLUMN | WS_HSCROLL

LTEXT "List these files:",IDC_STATIC,37,115,47,10

EDITTEXT IDC_EDIT1,91,113,100,15,ES_AUTOHSCROLL

END

```
////////////////////////////////////
```

```
//
```

```
// Property page 2 - "Date"
```

```
//
```

```
IDD_PAGE2 DIALOG 0, 0, 282, 135
```

```
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION
```

```
CAPTION "Date"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

```
CONTROL
```

```
"&Any date",IDC_RADIO1,"Button",
```

```
BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP,
```

```
15,10,40,10
```

CONTROL	"&between",IDC_RADIO2,"Button", BS_AUTORADIOBUTTON WS_TABSTOP,100,23,43,10
CONTROL	"&within previous",IDC_RADIO3,"Button", BS_AUTORADIOBUTTON WS_TABSTOP,100,80,63,10
CONTROL	"within &previous",IDC_RADIO4,"Button", BS_AUTORADIOBUTTON WS_TABSTOP,100,105,63,10
GROUPBOX	"Only files dated:",IDC_STATIC,70,10,200,115,WS_GROUP
CONTROL	"DateTimePicker1",IDC_DATETIME1,"SysDateTimePick32",DTS_RIGHTALIGN WS_TABSTOP,170,20,50,15
LTEXT	"and",IDC_STATIC,128,50,13,8,NOT WS_GROUP
CONTROL	"DateTimePicker1",IDC_DATETIME2,"SysDateTimePick32",DTS_RIGHTALIGN WS_TABSTOP,170,45,50,15
EDITTEXT	IDC_EDIT1,170,76,35,16,ES_NUMBER

```
CONTROL          "Spin3",IDC_SPIN1,"msctls_updown32",

                  UDS_SETBUDDYINT | UDS_ALIGNRIGHT | UDS_AUTOBUDDY |
                  UDS_ARROWKEYS,205,75,11,14

LTEXT            "month(s)",IDC_STATIC,216,80,40,11,NOT WS_GROUP

EDITTEXT        IDC_EDIT2,170,101,35,16,ES_NUMBER

CONTROL          "Spin3",IDC_SPIN2,"msctls_updown32",

                  UDS_SETBUDDYINT |UDS_ALIGNRIGHT | UDS_AUTOBUDDY |
                  UDS_ARROWKEYS,205,100,11,14

LTEXT            "day(s)",IDC_STATIC,220,105,40,11,NOT WS_GROUP
```

```
END
```

```
////////////////////////////////////
```

```
//
```

```
// Property page 3 - "Size"
```

```
//
```

```
IDD_PAGE3 DIALOG 0, 0, 282, 135
```

```
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION
```

```
CAPTION "Size"
```

```
FONT 8, "MS Sans Serif"
```

```
BEGIN
```

CONTROL	"&Any size",IDC_RADIO1,"Button", BS_AUTORADIOBUTTON WS_GROUP WS_TABSTOP,15,20,42,10
CONTROL	"&Only files between:",IDC_RADIO2,"Button", BS_AUTORADIOBUTTON,15,40,75,10
EDITTEXT	IDC_EDIT1,50,60,48,16,ES_NUMBER WS_GROUP
CONTROL	"Spin3",IDC_SPIN1,"msctls_updown32",

	UDS_SETBUDDYINT UDS_ALIGNRIGHT UDS_AUTOBUDDY UDS_ARROWKEYS,98,62,11,14
LTEXT	"and",IDC_STATIC,111,65,13,8,NOT WS_GROUP
EDITTEXT	IDC_EDIT2,140,60,48,16,ES_NUMBER
CONTROL	"Spin3",IDC_SPIN2,"msctls_updown32", UDS_SETBUDDYINT UDS_ALIGNRIGHT UDS_AUTOBUDDY UDS_ARROWKEYS,188,61,11,14
LTEXT	"kilobytes",IDC_STATIC,201,65,31,9,NOT WS_GROUP

END

DirList1.h

```
// *****
```

```
//
```

```
// DirList1.h
```

```
//
```

```
// *****
```

```
class CDirListApp : public CWinApp
```

```
{
```

```
public:
```

```
    BOOL      InitInstance ();
```

```
};
```

```
class CAboutDlg : public CDialog
```

```
{
```

```
public:
```

```
    CAboutDlg();
```

```
};
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// CPage1 property page
```

```
class CPage1 : public CPropertyPage
```

```
{
```

```
private:
```

```
    BOOL            bEditFocus;
```

```
    CString         strDirectory, strFilter;
```

```
    void            GetCurrentDirectory ();
```

```
    void            ShowList ();
```

```
public:
```

```
    CPage1 () : CPropertyPage( IDD_PAGE1 ) {}
```


protected:

```
virtual void      DoDataExchange( CDataExchange* pDX );
```

```
virtual BOOL      OnInitDialog ();
```

```
afx_msg void      OnEditGainFocus ();
```

```
afx_msg void      OnEditLoseFocus ();
```

```
afx_msg void      OnUp1Level ();
```

```
afx_msg void      OnSelChange ();
```

```
DECLARE_MESSAGE_MAP ()
```

```
};
```

```
////////////////////////////////////
```

```
// CPage2 property page
```

```
class CPage2 : public CPropertyPage
```

```
{  
  
public:  
  
    int          nAnyDate;  
  
    Ctime        timeMin, timeMax;  
  
    int          PrevDays;  
  
    int          PrevMonths;  
  
    CPage2 () : CPropertyPage( IDD_PAGE2 ) {}
```

```
protected:  
  
    virtual void    DoDataExchange( CDataExchange* pDX );  
  
    virtual BOOL    OnInitDialog ();  
  
};
```

```
////////////////////////////////////
```

```
// CPage3 property page

class CPage3 : public CPropertyPage
{
public:

    int          nAnySize;

    DWORD       MinSize;

    DWORD       MaxSize;


    CPage3 () : CPropertyPage( IDD_PAGE3 ) {}

protected:

    virtual void      DoDataExchange( CDataExchange* pDX );

    virtual BOOL      OnInitDialog ();
```

```
};

////////////////////////////////////

// CListSheet

class CListSheet : public CPropertySheet

{

public:

    CPage1    page1;

    CPage2    page2;

    CPage3    page3;

    CListSheet( LPCTSTR szCaption );

protected:

    virtual BOOL OnInitDialog();
```

```
afx_msg void OnSysCommand( UINT nID, LPARAM lParam );

DECLARE_MESSAGE_MAP()

};
```

DirList1.cpp

```
// *****

//

// DirList1.cpp

//

// *****

#include "afxwin.h"

#include "afxdlgs.h"

#include "afxcmn.h"
```

```
#include "resource.h"

#include "dirlist1.h"

#define DDL_ALL      DDL_DIRECTORY | DDL_DRIVES | DDL_HIDDEN

CDirListApp  DirListApp;

BOOL CDirListApp::InitInstance ()

{

    CListSheet sh( "Directory List" );    // Create property sheet object

    sh.DoModal ();                        // and display dialog

    return FALSE;                         // Return FALSE to exit DirList

}

CAboutDlg::CAboutDlg() : CDialog( IDD_ABOUTBOX )
```

```
{  
  
}  
  
////////////////////////////////////  
  
// CListSheet property sheet  
  
CListSheet::CListSheet( LPCTSTR szCaption ) : CPropertySheet( szCaption )  
  
{  
  
    AddPage( &page1 );  
  
    AddPage( &page2 );  
  
    AddPage( &page3 );  
  
}  
  
BEGIN_MESSAGE_MAP (CListSheet, CPropertySheet)  
  
    ON_WM_SYSCOMMAND()
```

```
END_MESSAGE_MAP()
```

```
BOOL CListSheet::OnInitDialog()
```

```
{
```

```
    CPropertySheet::OnInitDialog();
```

```
    // Add "About..." command to system menu
```

```
    CMenu* pSysMenu = GetSystemMenu( FALSE );
```

```
    CString str;
```

```
    str.LoadString( IDD_ABOUTBOX );
```

```
    pSysMenu->AppendMenu( MF_SEPARATOR );
```

```
    pSysMenu->AppendMenu( MF_STRING, IDD_ABOUTBOX, str );
```

```
    // Remove Apply button, since it's not needed
```

```
    CButton* button = (CButton *) GetDlgItem( ID_APPLY_NOW );
```



```
button->DestroyWindow();
```

```
// Set the 16-by-16 image (see closing remarks in Chapter 5)
```

```
HICON hIcon = (HICON) ::LoadImage( DirListApp.m_hInstance,
```

```
                                MAKEINTRESOURCE( IDI_APPICON ),
```

```
                                IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR );
```

```
SetIcon( hIcon, FALSE );
```

```
return TRUE;
```

```
}
```

```
void CListSheet::OnSysCommand( UINT nID, LPARAM lParam )
```

```
{
```

```
    if (nID == IDD_ABOUTBOX)
```

```
{
```

```
        CAboutDlg dlgAbout;
```

```
        dlgAbout.DoModal();
```

```
    }
```

```
else
```

```
{
```

```
    CPropertySheet::OnSysCommand( nID, lParam );
```

```
}
```

```
}
```

```
////////////////////////////////////
```

```
// CPage1 property page
```

```
BOOL CPage1::OnInitDialog ()
```

```
{
```

```
bEditFocus = FALSE;
```

```
strFilter = "*.*";
```

```
GetCurrentDirectory ();
```

```
ShowList ();
```

```
return CDialog::OnInitDialog ();
```

```
}
```

```
void CPage1::GetCurrentDirectory ()
```

```
{
```

```
PTSTR pDir = new char[MAX_PATH];
```

```
::GetCurrentDirectory( MAX_PATH, pDir );
```

```
strDirectory = pDir;
```

```
if(strDirectory.Right( 1 ) != "\\")
```

```
        strDirectory += "\\";                // Append backslash

        delete [] pDir;

}

void CPage1::ShowList ()

{

    PTSTR pDir = new char[MAX_PATH];

    lstrcpy( pDir, strDirectory );

    lstrcat( pDir, strFilter );

    DlgDirList( pDir, IDC_LIST, IDC_DIRPATH, DDL_ALL );

    delete [] pDir;

}

BEGIN_MESSAGE_MAP (CPage1, CPropertyPage)
```

```
ON_EN_SETFOCUS (IDC_EDIT1, OnEditGainFocus)
```

```
ON_EN_KILLFOCUS (IDC_EDIT1, OnEditLoseFocus)
```

```
ON_BN_CLICKED(IDC_BUTTON1, OnUp1Level)
```

```
ON_LBN_SELCHANGE (IDC_LIST, OnSelChange)
```

```
END_MESSAGE_MAP ()
```

```
void CPage1::OnEditGainFocus ()
```

```
{
```

```
    bEditFocus = TRUE;
```

```
}
```

```
void CPage1::OnEditLoseFocus ()
```

```
{
```

```
    bEditFocus = FALSE;
```

```
        GetDlgItemText( IDC_EDIT1, strFilter );

        ShowList ();

}

void CPage1::OnUp1Level ()

{

    // If Enter pressed in edit control, refresh list

    if (bEditFocus)

    {

        GetDlgItemText( IDC_EDIT1, strFilter );

        ShowList ();

    }

    // Else go up one directory level
```

```
else

{

    // When strDirectory == "d:\", we're already at root

    if (strDirectory.Right( 2 ) != ":\\")

    {

        // Remove '\' at end of string

        strDirectory.GetBufferSetLength( strDirectory.GetLength() - 1 );

        strDirectory.ReleaseBuffer ();

        // Find last '\' and truncate strDirectory string

        int cLastSlash = strDirectory.ReverseFind( '\\' );

        if (cLastSlash != -1)

        {
```

```
        strDirectory.GetBufferSetLength( cLastSlash + 1 );

        strDirectory.ReleaseBuffer ();

        ShowList ();

    }

}

}

}

}

}

void CPage1::OnSelChange ()

{

    char        szItem[MAX_PATH];

    char        *pItem = szItem;

    int         i;
```



```
i = SendDlgItemMessage( IDC_LIST, LB_GETCURSEL, 0, 0 );

SendDlgItemMessage( IDC_LIST, LB_GETTEXT, i, (LPARAM) szItem );

// We're interested only in drives [-d-] or subdirectories [subdir]

if (szItem[0] == '[')

{

    if (lstrcmp( szItem, "[..]" ))        // Ignore parent directory "[..]"

    {

        pItem++;                          // Skip past first bracket '['

        // If drive, change "[-d-]" to "d:"

        if (pItem[0] == '-' && pItem[2] == '-')

        {

            pItem++;                        // Skip past first hyphen '-'
```

```
        pItem[1] = ':';                // Overwrite second '-' with ':'

        pItem[2] = '\0';                // Truncate string

        strDirectory = pItem;           // New directory
    }

    // If subdir, change "[subdir]" to "subdir"

    else

    {

        i = lstrlen( szItem );

        pItem[i-2] = '\0';              // Overwrite ']' to truncate

        strDirectory += pItem;           // Append new subdirectory
    }

    strDirectory += "\\";                // Append backslash
```

```
        ShowList ();                // Refresh list
```

```
    }
```

```
}
```

```
}
```

```
void CPage1::DoDataExchange( CDataExchange* pDX )
```

```
{
```

```
    CPropertyPage::DoDataExchange( pDX );
```

```
    DDX_Text( pDX, IDC_EDIT1, strFilter );
```

```
    DDV_MaxChars( pDX, strFilter, 128 );
```

```
}
```

```
////////////////////////////////////
```

```
// CPage2 property page

BOOL CPage2::OnInitDialog ()

{

    CSpinButtonCtrl* spin;

    // Initialize variables

    nAnyDate    = 0;

    PrevDays    = 1;

    PrevMonths  = 1;

    timeMin     = 0;

    timeMax     = CTime::GetCurrentTime();

    // Set limits of spin buttons

    spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN1 );
```

```
spin->SetRange( 1, 100 );                                // Within x months

spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN2 );

spin->SetRange( 1, 365 );                                  // Within x days

return CDialog::OnInitDialog ();

}

void CPage2::DoDataExchange( CDataExchange* pDX )

{

    CPropertyPage::DoDataExchange( pDX );

    DDX_DateTimeCtrl( pDX, IDC_DATETIME1, timeMin);

    DDX_DateTimeCtrl( pDX, IDC_DATETIME2, timeMax);

    DDX_Radio( pDX, IDC_RADIO1, nAnyDate );

    DDX_Text( pDX, IDC_EDIT1, PrevMonths );
```

```
        DDX_Text(    pDX, IDC_EDIT2, PrevDays );

}

////////////////////////////////////

// CPage3 property page

BOOL CPage3::OnInitDialog ()

{

    CSpinButtonCtrl* spin;

    // Initialize variables

    nAnySize = 0;

    MinSize   = 0;

    MaxSize   = 100;

    // Set limits of spin buttons
```

```
spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN1 );

spin->SetRange( 0, 9999 );                                // Min size

spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN2 );

spin->SetRange( 1, 9999 );                                // Max size

return CDialog::OnInitDialog ();

}

void CPage3::DoDataExchange( CDataExchange* pDX )

{

    CPropertyPage::DoDataExchange( pDX );

    DDX_Radio( pDX, IDC_RADIO1, nAnySize );

    DDX_Text( pDX, IDC_EDIT1, MinSize );

    DDX_Text( pDX, IDC_EDIT2, MaxSize );
```

```
}
```

该程序的 `InitInstance` 函数首先创建一个 `ClistSheet` 对象，该对象是从 MFC 的 `CPropertySheet` 类中派生出来的。`DoModel` 成员函数用来显示属性表：

```
CListSheet sh( "Directory List" ); // Make propsheet object
```

```
sh.DoModal (); // and display dialog
```

当该对象的 `OnInitDialog` 函数获得控制的时候，它向系统菜单添加一个调用该程序的 `About` 框的命令。然后删除不需要的 `Apply` 按钮，这个按钮是框架把它放在对话框中的。`OnInitDialog` 函数还演示了基于对话框的应用程序是如何设置两个图标图像的，一个 16×16 像素的小图像作为对话框标题栏上的应用程序图标， 32×32 像素的大图像将出现在 `About`（关于）框中。两个图像都在项目的 `DirList.ico` 文件之中。MFC 框架在创建应用程序窗口时将自动抽取大图像，因此，`DirList1` 必须通过调用第 4 章中介绍的 `Load Image API` 函数来自己调用小图像：

```
// Set the 16-by-16 icon image
```

```
HICON hIcon = (HICON) ::LoadImage( DirLsitApp.m_hInstance,
```

```
MAKEINTRESOURCE( IDI_APPICON ),
```



```
IMAGE_ICON,16,16,LR_DEFAULTCOLOR );
```

```
SetIcon( hIcon, FALSE );
```

在这种情况下，调用 LoadImage 是必要的，因为 LoadIcon 函数只能正确识别 32 × 32(或更多)像素的图像，一个尺寸与 SM_CXICON 和 SM_CYICON 系统计量值相匹配的图像。向 SetIcon 函数传递一个参数值 FALSE 可能把小的 16 × 16 图像设置为应用程序图标。

就其本身而言，ClistSheet 对象是一个空的对话框窗口。ClistSheet 构造器通过调用 AddPage 成员函数来添加三个属性页。AddPage 将指针指向 CPropertyPage 对象，该对象是通过向 CPropertyPage 基本初始化程序传递对话框标识符而创建的。例如，下面是 DirList1 为属性表的第一页声明 CPropertyPage 对象的代码：

```
public:
```

```
    CPage1 ():CPropertyPage(IDD_PAGE1) {}
```

AddPage 函数负责将对话框资源转换成带有选项卡的属性页，该选项卡上包含有对话框标题，例如 Location 和 Size。

当准备显示第一页的时候，它的 CPage1::OnInitDialog 函数调用 GetCurrentDirectory API 函数，向当前路径添加 *.* 过滤器，并且调用 DlgDirList，

将当前的目录列表写到列表框之中。StrFilter 字符串包含有用户可以通过编辑控件改变的过滤器。程序最有趣的部分在 CPage1::OnSelChange 函数中，当用户选中了列表框中一个项目的时候，该函数便得到控制权。

如果用户选中了一个文件名，函数将忽略这个选择。如果选中的项目是磁盘驱动器，那么驱动器目标将代替 StrDirectory 中的路径，然后成为字符串 d:\，其中 d 是所选中的驱动器符号。在列表框中选定了个子目录，将把子目录名添加到 StrDirectory 中的路径。当路径改变的时候，OnSelChange 将调用 ShowList 来显示新路径的目录列表。OnSelChange 将按照 CWnd::DlgDirList 在列表框中写该文件名、驱动器和子目录的方式来在它们之间进行区分。驱动器符号包含在方括号和边字符之中，例如 [-a-] 或 [-c-]。子目录包括在没有连字符的方括号之中。文件名没有括号或连字符。

虽然调用 DlgDirList 添加目录列表具有一定的方便性，但该技术还存在着两个严重的缺陷。首先，目录列表不吸引人，并没有达到当前的 Windows 标准。其次，DlgDirList 只有在 Windows NT 下才能识别长文件名。出于这两方面的考虑，我们不得不完全放弃列表框，而来试一试其他东西。当我们创建了一个从 MFC 的 CListView 中派生出来的新类，并将其添加到 Visual C++ 中叫做 Gallery 的组件集时，我们再来做这件事。

但是，目前有另一个问题比 Gallery 更加迫切。本章中已完成的一些工作可能就变得不必要了，或者至少是这样，它可以得到简化。你可能会记起，在创建

MFCTree 项目时，我们为 Mfclagh 和 MfcDlg.cpp 源文件编写了存根函数，并且将项目手工插入到一个消息映射之中。这些任务可以在 ClassWizard 中很容易地进行处理。ClassWizard 是对话框编辑器的天然伙伴；使用编辑器来设计对话框，使 ClassWizard 可以生成运行该对话框的源代码。下一章将显示如何去做。

第三部分 编程帮助工具

第 6 章 ClassWizard（类向导）

用 AppWizard 创建一个项目之后，你便拥有了 ClassWizard 这个 Visual C++ “程序员助手”选项。从广义上讲，ClassWizard 与类的关系和 AppWizard 与应用程序的关系是相同的。ClassWizard 通过生成实现 CPP 文件，以及一个带有合适存根函数的头文件，来帮助你开始编写新类。用实际的代码填充这些函数则是你的责任。

ClassWizard 在四个区域提供帮助，它可以为如下目的生成代码：

- 从接收消息或管理控件窗口的许多 MFC 类中的一个派生出来的新类。
- 处理消息的成员函数。
- OLE/ActiveX 方法、属性和事件触发。
- 用于输入到对话控件中的数据交换和验证函数。

ClassWizard 识别和支持 MFC 基类，它们用某种方法与用户进行交互。除了少数例外的类（例如 CRecordSet 和 CHttpServer）之外，基类都是由 CCmdTarget 派

生而来，它们能够对消息作出反应，并能够管理对话框中的控件。大约有 50 个多个 MFC 基类，你可以使用 ClassWizard 从这样的类中创建一个派生类；参考本书附录 B，可以得到一个完整列表。

访问 ClassWizard

不能从一个空的项目中访问 ClassWizard。项目中至少应有一个 RC 文件，即使这个 RC 文件是空的。一旦 RC 文件被附加到项目之中（AppWizard 可以自动完成这项工作），你就可以通过从 View 菜单中选择 ClassWizard 命令来调用 ClassWizard 对话了，如图 6-1 所示。

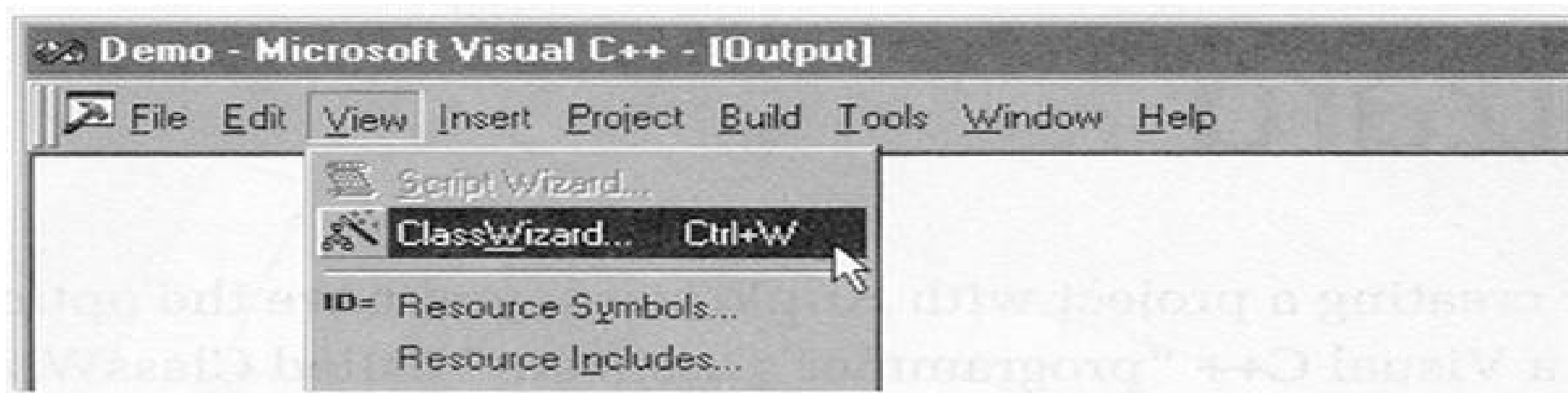
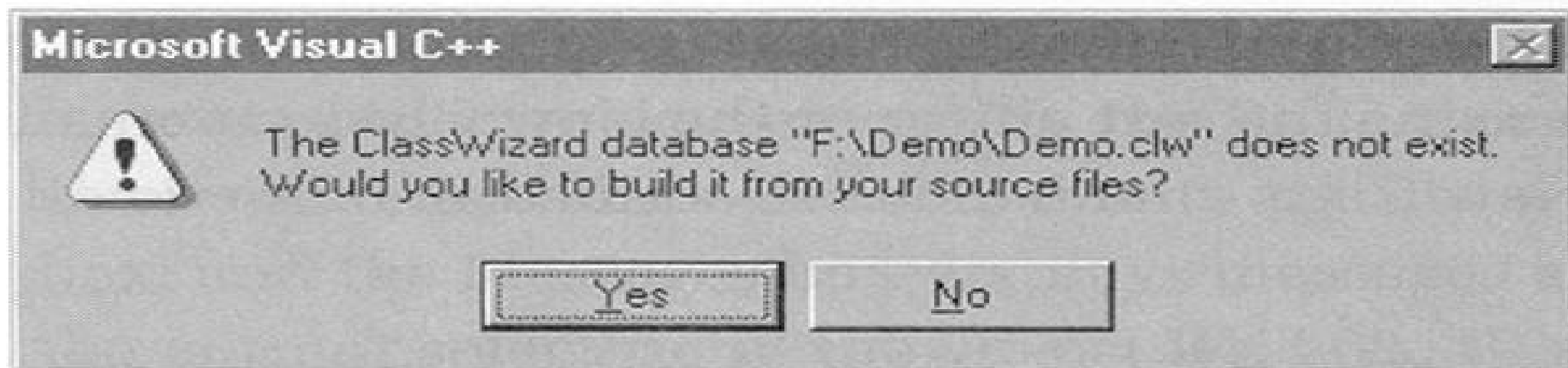


图 6-1 View 菜单中的 ClassWizard

关于 ClassWizard，有两点不大明显。首先，它的服务完全是可选的。你在开发自己项目的过程中，如果自己愿意的话，从头至尾，都可以不处理 ClassWizard。其次，你可以使用 ClassWizard 向 MFC 项目添加新类，即使该项目不是由 AppWizard 创建的。ClassWizard 编译项目的类的数据库，并将它存储在与项目同名且扩展名为 CLW 的一个文件之中。如果你一直都是使用 ClassWizard 为项目创建新类，那么 CLW 文件可以保持最新。不过，在 Visual C++ 中，你与 ClassWizard 之间并不是全有或全无的关系，你可以自由地编写新类，或者从当前项目之外的其他源文件中复制代码。在一个类不是以 ClassWizard 为起源的情况下，有一个简单的方法可以更新 CLW 数据库。在你使用了 Project 菜单上的 Add To Project 命令向项目添加新类源文件之后，请删除 CLW 文件，并再次调用

ClassWizard。Visual C++将检测到数据库不存在了，并提供重新创建它的服务：



当你单击 Yes 接受创建新数据库的服务之后，将出现一个标题为 Select Source Files 的对话，其中带有一个 Visual C++创建类数据库将要读取的实现文件和文件夹列表。如果你已经将全部的新源文件都插入到了项目之中，那么列表应该已经是完整的了，因此除单击 OK 外，你不需要做任何事情。进程指示器主要用来指示 Visual C++正在创建数据库文件，在此之后，将出现 ClassWizard 对话框。

ClassWizard对话

图 6-2 显示了主 ClassWizard 对话框。我之所以叫它为“主”对话，是因为 ClassWizard 可以在 20 多种不同的对话中显示自己，究竟在哪一个对话中显示需

取决于环境。图 6-2 中的对话充当着到 ClassWizard 主入口的角色，将其命名为 MFC ClassWizard，以便于让你记住它所处理的仅仅是 MFC 类。ClassWizard 不能帮助你创建从除了支持 MFC 类之外的任何类中派生的类；对于带有任何其他基的类，你必须使用文本编辑器或 Insert 菜单上的 New Class 命令从零开始自己编写代码。在本章稍后将详细介绍 New Class 命令。

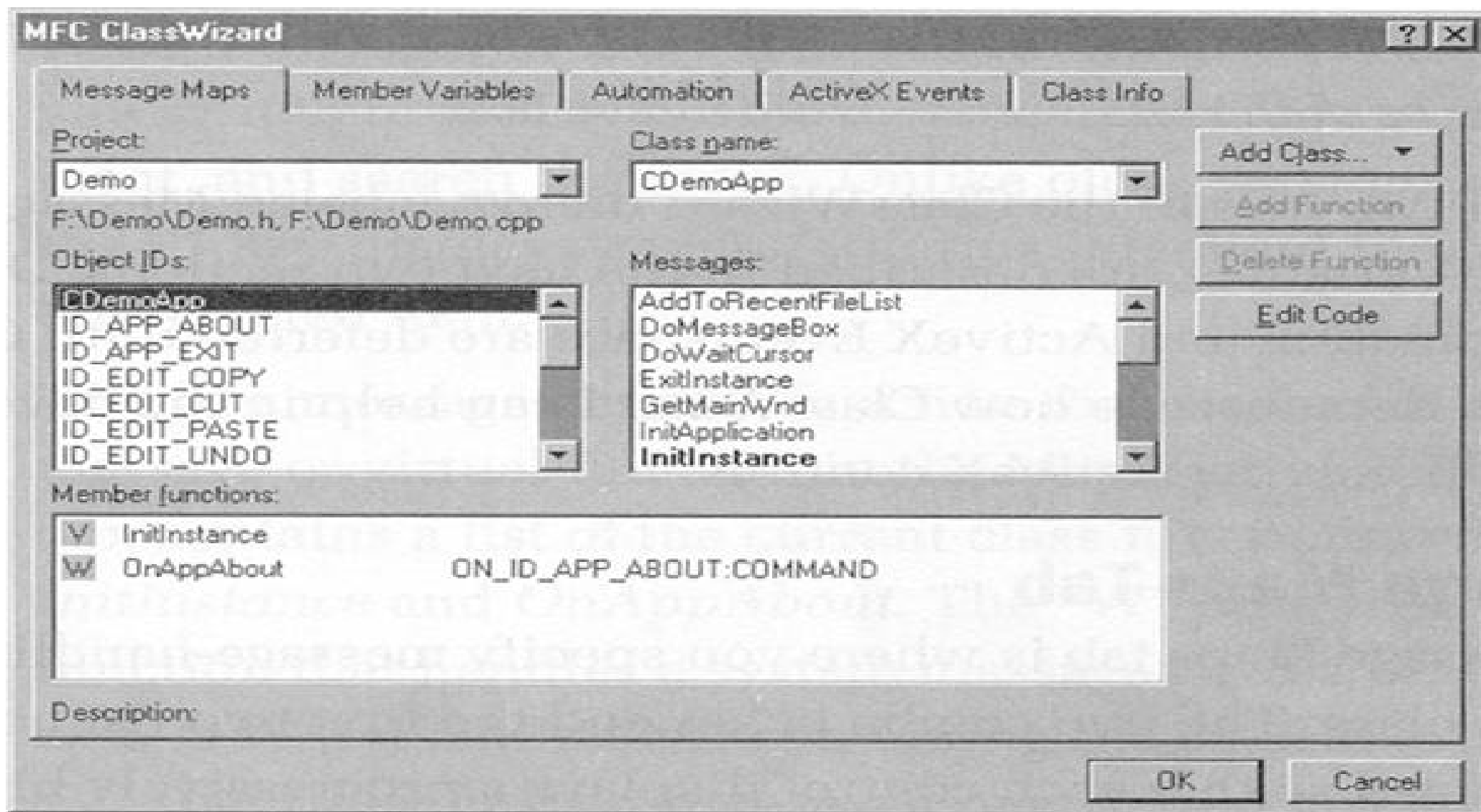


图 6-2 MFC ClassWizard 对话

MFC ClassWizard 对话的五个选项卡具有不同的用途，对于任何一个特定的类，都不是完全需要这五个选项卡。表 6-1 可以帮助你确定自己需要哪个（或哪几个）选项卡，这主要取决于你想为自己的类做哪些工作。

表 6-1 MFC ClassWizard 对话框的五个选项卡

选项卡	用途
Message Maps(消息映射)	添加或删除处理消息的成员函数
Member Variables (成员变量)	添加或删除附加到使用控件的类上的成员变量。一般地说，这些是从 CDialog，CPropertyPage，CRecordView 或 CDaoRecordView 中派生的对话类
Automation	向支持 Automation 的类（例如 ActiveX 控件类）添加属性或方法

续表

ActiveX Events

为触发事件（通常是实现 ActiveX 控件的类）添加支持。在开发接收触发事件的容器应用程序时，不用该选项卡

Class Info

有关项目的类的其他信息

在 ClassWizard 对话框中，标为 Message Maps 和 Member Variables 的头两个选项卡将在下两节介绍。Automation 和 ActiveX Event 选项卡的讨论推迟到第 8 章和第 9 章，这两章将阐述 ClassWizard 如何在包含 ActiveX 控件的项目开发中起帮助作用。

Message Maps（消息映射）选项卡

Message Maps（消息映射）选项卡是你为自己的类指定消息处理函数的地方。该选项卡中的两个组合框和前两个列表框是这样安排的，每个控件都显示逐渐高一级的控件详细资料。换句话说，控件的内容取决于前面控件中的选中部分。Class Name（类名）组合框列出了 Project 组合框中选中的项目的类；Object IDs 框中显示的是与 Class Name 框中选中的类相关联的标识符，Message 框中显示的是 Object IDs 框中当前选中部分的消息和其他信息。表 6-2 显示了 Object IDs 框中

选中的项目和 Message 框内容之间的关系。

表 6-2 在 Message Map 选项卡中，Object Ids 框中的选中部分决定了 Message 框的内容

Object IDS 框中选中部分	Message 框的内容
类名	WM_消息和处理消息的类虚拟函数
菜单命令标识符	用于菜单命令消息的 ON_COMMAD 和 ON_COMMAND_UI 宏
控件标识符	控件通知消息。反应的消息用等于 (=) 前缀标记

当你在 Message 框中选中一条消息或虚拟函数的时候，将在 MFC ClassWizard 对话的底部出现选中项目的简要说明。如果想了解有关选中项目的详细信息，可以像第一章中介绍的那样切换到 MSDN 在线帮助，然后再搜索索引。与 Visual C++ 旧版本不同，按下 F1 键将显示有关 Message 框本身的一般信息，而不是有关选中项目的信息。

如果想向选定的类添加消息处理程序函数，请双击 Message 框中的消息或虚拟函

数。Member Functions 框包含有当前类函数的一个列表,在图 6-2 中是 InitInstance 和 OnAppAbout。“W”标识符将 OnAppAbout 作为一个处理带有 WM_前缀的系统消息的函数来标识,在本例中,是包含有 ID_APP_ABOUT 菜单值的 WM_COMMAND。“V”用来将 InitInstance 作为一个超越虚拟函数来标识。

对于添加到类的每个消息处理程序函数,ClassWizard 对该类的源文件做三处修改:

- 向头文件添向函数声明。
- 向 CPP 实现文件添加带有骨干代码的函数定义。
- 将代表该函数的条目添加到类的消息映射。

Member Variables (成员变量) 选项卡

Member Variables (成员变量) 选项卡与使用控件的类相关,这些类中的大部分是从 CDialog, CPropertyPage, CFormView, CRecordView 或 CDaoRecordView 中派生出来的。从上述五个 MFC 类中的一个派生出来的类叫做对话类,因为它需要一个代表对话资源的标识符。Member Variables (成员变量) 选项卡是指定从类对话中的控件中接收数据的成员变量的地方。

如果想向对话类添加成员变量，请打开 Member Variables（成员变量）选项卡，并从 Class Name（类名）框中选择类。在 Control IDs 框中，选择你想向其上附加新变量的控件的标识符，然后单击 Add Variable 按钮打开如图 6-3 所示的 Add Member Variable 对话。

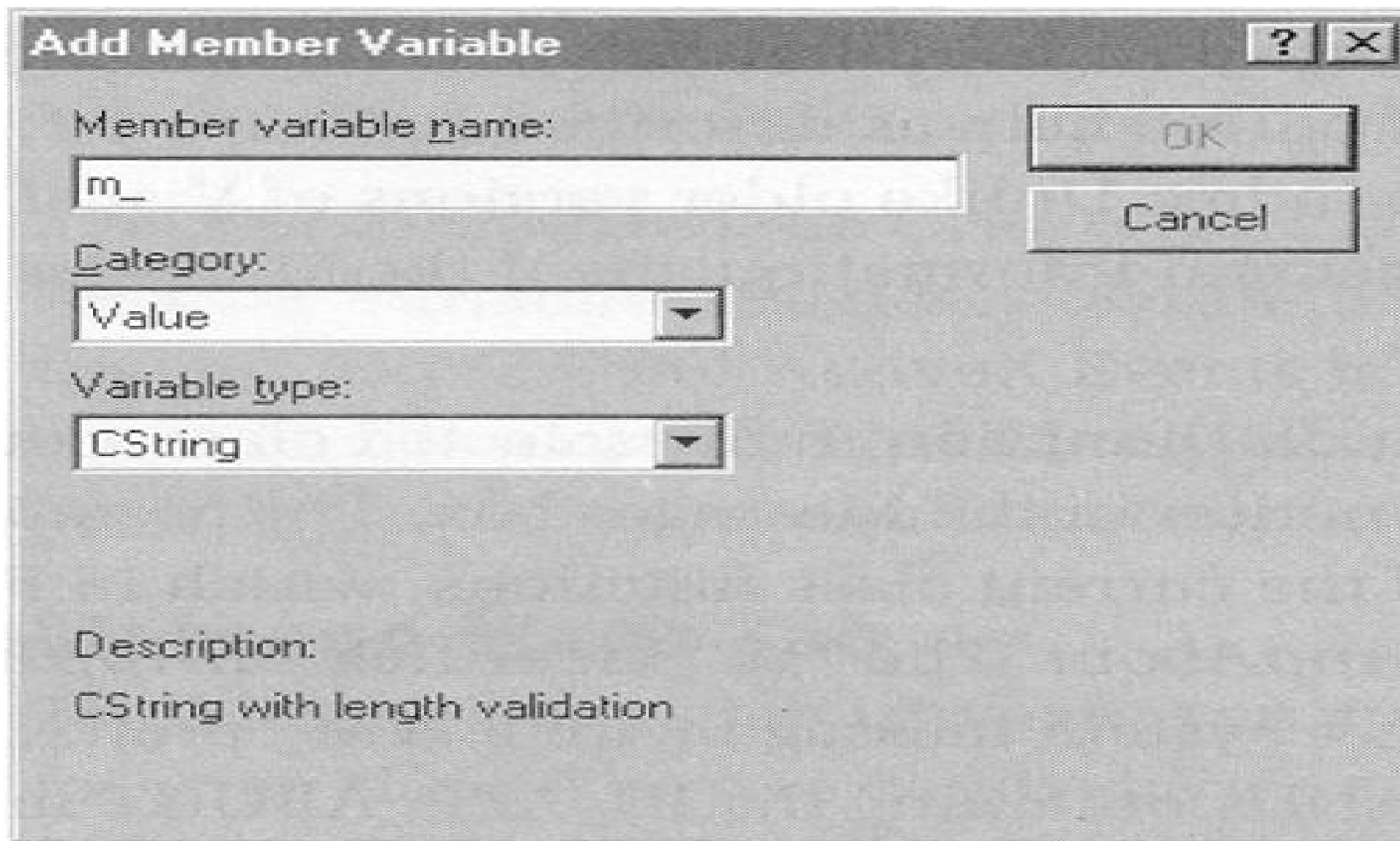


图 6-3 从 Member Variables (成员变量) 选项卡中调用的 Add Member Variable 对话框

在可选的“M”前缀后面键入变量名，然后根据控件类型，在 Category 框中选择 Value 或 Control。Value 设置的意思是指该变量包含有这个控件的数据，例如用户向编辑框键入的文字或数字值。Control 设置的意思是该变量代表控件本身。作为一个例子，可以把成员变量考虑成一个复选框控件。选定 Value 类别可以使该成员变量成为 Boolean 型，能够包含有用来指示该复选框状态的值 TRUE 或 FALSE。然而，选择 Control 类别，将使变量成为一个代表复选框控件的 CButton 对象。如果你想使用该变量来确定用户是打开还是关闭了这个复选框，那么请选择 Value 类别。如果你想使用该变量在运行时以某种方式改变复选框，例如通过调用 CButton:SetWindowText 来改变标题，或通过调用 CButton::SetCheck 设置复选框的状态，那么请选择 Control 类别。

表 6-3 列出了可用的标准控件的变量类型。当你阅读该表时，请记住如下几点：

- 编辑框特别适合将数据传递给许多不同类型的变量。表中提到的“数字”数据类型是一个包罗万象的术语，它包括 BYTE, short, int, UNIT, long, DWORD, float 和 double。
- ClassWizard 不列出由 IDC_STATIC 值标识的静态控件。如果想把静态控件与变量关联起来，应首先为该控件分配一个除了 IDC_STATIC 之外的值。
- 在一组单选按钮中，仅有该组中第一个按钮的标识符出现在 Control IDs 框中。当我们稍后讨论与小组的第一个按钮相关联的单个变量是

如何代表整个小组的时候，你就会清楚为什么不包括小组中其他按钮的原因。

- 正如表中所指示的，Push 按钮控件不允许 Value 类别的变量，因为这些控件不接受来自用户的数据。

表 6-3 用于标准控件的 Value 和 Control 类别的变量类型

控 件	数 据 类 型	
	Value 类别	Control 类别
复 选 框	BOOL	CButton
复 选 框 （ 三 态 ）	Int	CButton
组 合 框	CString 或 int	CComboBox

续表

编辑框	CString, BOOL, 数字, 或 COleDateTime COleCurrency	CEdit
列表框	Cstring 或 int	CListBox
压入式按钮		CButton
单选按钮	Int	CButton
滚动条	int	CScrollBar
静态文本	CString	CStatic

通过使用 ClassWizard 向对话类添加成员变量, 你可以利用 ClassWizard 所提供的高效特征: 为对话数据交换和对话数据验证自动生成源代码, 也就是人们所熟知的 DDX/DDV。数据交换和验证仅仅应用于为之选定了 Value 类别的成员变量。也就是说, 具有在表 6-3 中间那列列出的类型的成员变量。

对话数据交换负责数据在控件中的进出。当对话首次出现的时候, 每个控件窗口自动用相应的成员变量的值进行初始化。当用户通过单击 OK 按钮, 或通过按

Enter 键关闭对话的时候，该控件无论是包含哪一个值或文本，都将被复制回该变量。对话数据验证可以确保值落在规定的限制之内。交换和验证机制都是由MFC 框架通过表 6-4 和 6-5 中所列出的函数集提供的。每个函数都有一个前缀 DDX_或 DDV_，来把它作为数据交换或数据验证函数来标识。

表 6-4 常见的对话数据交换函数

交 换 函 数	获 得 / 设 置 这 种 类 型 的 数 据 ...	... 用 于 该 种 类 型 控 件
DDX_CBIndex	int	Combo box
DDX_CBString	CString	Combo box
DDX_CBStringExact	CString	Combo box
DDX_Check	int	Check box
DDX_DateTimeCtrl	Ctime	Date-time picker
DDX_LBIndex	int	List box

续表

DDX_LBString	CString	List box
DDX_LBStringExact	CString	List box
DDX_MonthCalCtrl	Ctime	Month calendar
DDX_Radio	int	Radio button
DDX_Scroll	int	Scroll bar
DDX_Text	CString or numerical (BYTE, short,int,UINT,long,etc.)	Edit control

对话数据交换 (DDX)

MFC 提供了大量的对话数据交换函数，它们可以在对话类中的控件成员变量之间移动数据。除了表 6-4 中所列的常见函数之外，还有用于记录集数据和由 ActiveX 控件返回的数据的特殊交换函数。DDX_Control 函数可以为几种不同类型的控件传输数据，例如 Animate 和 IP Address。如果想了解有关数据交换函数

的更为详细的信息，请参考 Visual C++在线帮助。

表 6-4 中列出的 DDX_Radio 函数值得讨论一下。在所有的数据交换函数之中，该函数是独一无二的，它只应用于一组控件，而不是一个控件。DDX_Radio 返回一个 int 值用来指示用户打开了组中的哪个按钮：0 代表组中的第一个按钮，1 代表第二个按钮，以此类推。值 -1 的意思是小组中的所有按钮都是清除的。你可以调用 DDX_Radio 来确定单个单选按钮的状态（假如它是组中的唯一一个按钮）。在这种情况下，返回值 0 的意思是按钮是打开的，值 -1 意为按钮是关闭的。建立一个单选按钮通常在对话编辑器中进行，我们一会儿就能看到。

对话数据验证（DDV）

表 6-5 列出了八个对话数据验证函数，它们仅仅应用于接受用于从键盘输入数据的控件成员变量。换言之，就是编辑控件和组合框。

表 6-5 对话数据交换函数

验证函数	验证 ...
DDV_MinMaxByte	指定限制范围内的一个 BYTE 值
DDV_MinMaxInt	指定限制范围内的一个 int 值
DDV_MinMaxUInt	指定限制范围内的一个 UNIT 值
DDV_MinMaxLong	指定限制范围内的一个 long 值
DDV_MinMaxDWord	指定限制范围内的一个 DWORD 值
DDV_MinMaxFloat	指定限制范围内的一个 float 值
DDV_MinMaxDouble	指定限制范围内的一个 double 值
DDV_MaxChars	CString 字符串的长度不能超过指定的最大长度

当你为编辑控件或组合框添加成员变量，然后在 Member Variable 选项卡中的 Control IDs 框中选择控件的时候，两个提示之一将会出现在该选项卡的底部。究

竟出现哪一个提示取决于变量所具有的是数字数据，还是文字数据；在任一种情况下，都要输入用于验证的变量限制值：

Minimum/maximum limits of a number	Maximum string length
Description: int with validation	Description: CString with length validation
Minimum Value:	Maximum Characters:
Maximum Value:	

除了一个对话数据验证函数之外，所有函数都监视数值数据，以确保由用户输入的值落在指定的上下限之间。例外是 `DDV_MaxChars` 函数，它用来验证键入到编辑控件或组合框中的字符数不超过给定的最大值。与表 6-4 中列出的交换函数不同，验证函数仅仅在对话关闭的时候起作用，而不在它刚出现的时候起作用。如果输入到一个控件的值落到了指定的限制之外，那么，该控件的验证函数将显示一个消息框，以通知用户出了问题。当消息框被关闭的时候，出问题的控件将具有一个焦点，提示用户重新输入数据。除非所有的数据验证函数都满足了，否则，用户不能够通过单击 OK 来关闭对话。

DoDataExchange 函数

ClassWizard 把叫做 `DoDataExchange` 的成员函数添加到对话类的 CPP 文件。这个

函数包含对话所需要的对数据交换和验证函数的所有调用。例如，对于接受从 1 到 99 的整数的单个编辑控件，ClassWizard 可以编写不同的 DoDataExchange 函数：

```
void CDemoDlg::DoDataExchange(CDataExchange* pDX)

{

    CDialog::DoDataExchange(pDX);

   //{{AFX_DATA_MAP(CDemoDlg)

    DDX_MinMaxInt(pDX,m_nEditVal, 1, 99);

   //}}AFX_DATA_MAP

}
```

DoDataExchange 是一个完整的函数，并不是你必须进一步添加的存根代码。既然在对话开始和结束的时候，框架都要调用 DoDataExchange，那么，有时向它添加初始化和清除代码是很方便的，否则你通常会忘记该函数的存在。如果你在 Member Variables（成员变量）选项卡删除了一个变量，那么 ClassWizard 将会在 DoDataExchange 函数中删除对该变量任何数据的交换和验证调用。

ClassWizard 和 AppWizard 为项目中的每个对话编写 DoDataExchange 函数，即使

对话没有接受用户输入的控件也是如此。例如，由 AppWizard 生成的 About 框类具有一个 DoDataExchange 函数，即使该对话没有任何控件（一个图标，两行静态文字和一个压入式按钮）可以从用户处接收数据。如果你想让一个控件仅仅显示数据，而不允许用户改变值，那么应该删除 ClassWizard 添加到 DoDataExchange 的控件的任何 DDX/DDV 调用。另一个选项是完全删除 DoDataExchange 函数，我们将要在本章稍后看到另一个例子。

在创建对话的时候，在设计过程的早期，你应该考虑一下对话的类是如何在对话数据交换与验证融合在一起的。在第 5 章中介绍过，对话编辑器允许你通过在控件的 Properties 框中打开或关闭合适的复选框来设计每个控件的属性。当为使用对话数据交换和验证的控件设计属性的时候，请记住以下几点：

- 对于只接受简单的十进制（而不是十六进制）的整数的编辑控件，在 Edit Properties（编辑属性）对话的 Styles（风格）选项卡中设置 Number（数字）复选框。这将给这个控件分配一个样式标志 ES_NUMBER，使它忽略除从 0 到 9 之外的任何字符，其中也包括逗号和句号。
- 如果组合框或列表框只接受自由数字数据，那么，请访问该控件的 Properties 对话，并关闭它的排序选项。对于组合框和一般的列表框，请清除 Styles（风格）选项卡中的 Sort Property（排序属性）复选框；对于列表控件，确保 Style（风格）选项卡中的 Sort Selection（排序选择）为 None。因为这些控件将根据数字的 ASCII 值，而不是数字值，

对数字进行排序，仅仅当条目具有固定的数字号码，并且包含有前导 0（例如 001 和 099）时，它们才能正确地排出一个号码列表。

- 当你创建一组单选按钮的时候，请为组中的每个按钮设置 `Auto` 属性。这将使得这个单选按钮成为互斥的，以便单击一个按钮将自动清除组中其他所有的按钮。
- 仅仅为组中的第一个单选按钮设置 `Group` 属性，确保组中其他所有的按钮按照 `Tab` 键顺序跟在第一个按钮之后。如果这样做失败了，那么当用户按箭头键时，将禁止对话将焦点从一个单选按钮移动到下一个单选按钮。另外，还要为紧跟在组中最后的单选按钮之后的那个控件设置 `Group` 属性。这表示这是前一组的末尾和另一组的开始。在调试器中运行该程序将会告诉你，何时一组单选按钮不能由 `Group` 属性来正确地定界，因为当对话出现的时候，调试器将在 `Output` 窗口中显示这条消息。

Warning: skipping non-radio button in group.（警告：跳过组中的非单选按钮）

向项目添加类

MFC ClassWizard 的所有五个选项卡都具有 `Add Class`（添加类）按钮，因为你可以从对话的任何一个选项卡中开始一个新类。单击了 `Add Class`（添加类）按钮

之后，选中如下两个选项之一：

- **New（新建）**：创建包含有由 ClassWizard 生成的骨干代码的 CPP 文件和 H 文件。
- **From a type library（来自类型库）**：为基于 OLE 类库的一个类创建源代码。这个类型库可以是一个单独的文件（通常具有 TLB 扩展名），或者它可以作为资源数据包含在像动态链接库这样的程序内部。动态链接库一般具有如下扩展名：OLB（对象库）、OCX（ActiveX 控件）或 DLL。例如，选择第 8 章中介绍的任何一个 ActiveX 控件，都将从 OCX 文件的类型库信息中生成的项目中添加类源代码。在你定位并选中了所需要的类型库文件之后，在类列表中选择一类，这个类列表是 ClassWizard 从库中的信息收集保存的，并在 Confirm Classes(确认) 对话框中显示。如果你希望为当前的项目重命名选中的类，那么请在 Class Name 控件中输入新的名字。你还可以指定 ClassWizard 所创建的 CPP 和 H 文件的名称。

两个选项都可以为新类自动更新 CLW 数据库。如果想通过从另一个项目中复制一个现有的类来创建一个新类，那么可以采用三种不同的方法。借用现有类的最直接方法是将源代码复制到你的项目文件夹，并使用 Add To Project（添加到项目中）命令，将 CPP 文件添加到项目。在弹出式次级菜单中单击 Files，然后双击文件列表中该类的 CPP 文件。正如前一章中所提及的，不必要以相同的方法

添加头文件，原因是 Visual C++ 是将头文件做为从属文件来识别。第二种方法需从 Project 菜单中 Insert Project Into Workspace（插入项目到工作空间）命令来插入现有的项目。然后，你就可以自由地从插入的项目中借用源文件了。第三种方法是通过 Gallery，我们将在第 7 章中进行讨论。

你可能不是从类型库中添加一个现有的类，而是更经常地选择 New 选项，来让 ClassWizard 启动你想开发的新的派生类。单击 New 将显示如图 6-4 所示的 New Class 对话框，在这个对话框中，你可以为你的类输入一个名字，然后从 ClassWizard 支持的 MFC 类中选择它的基。

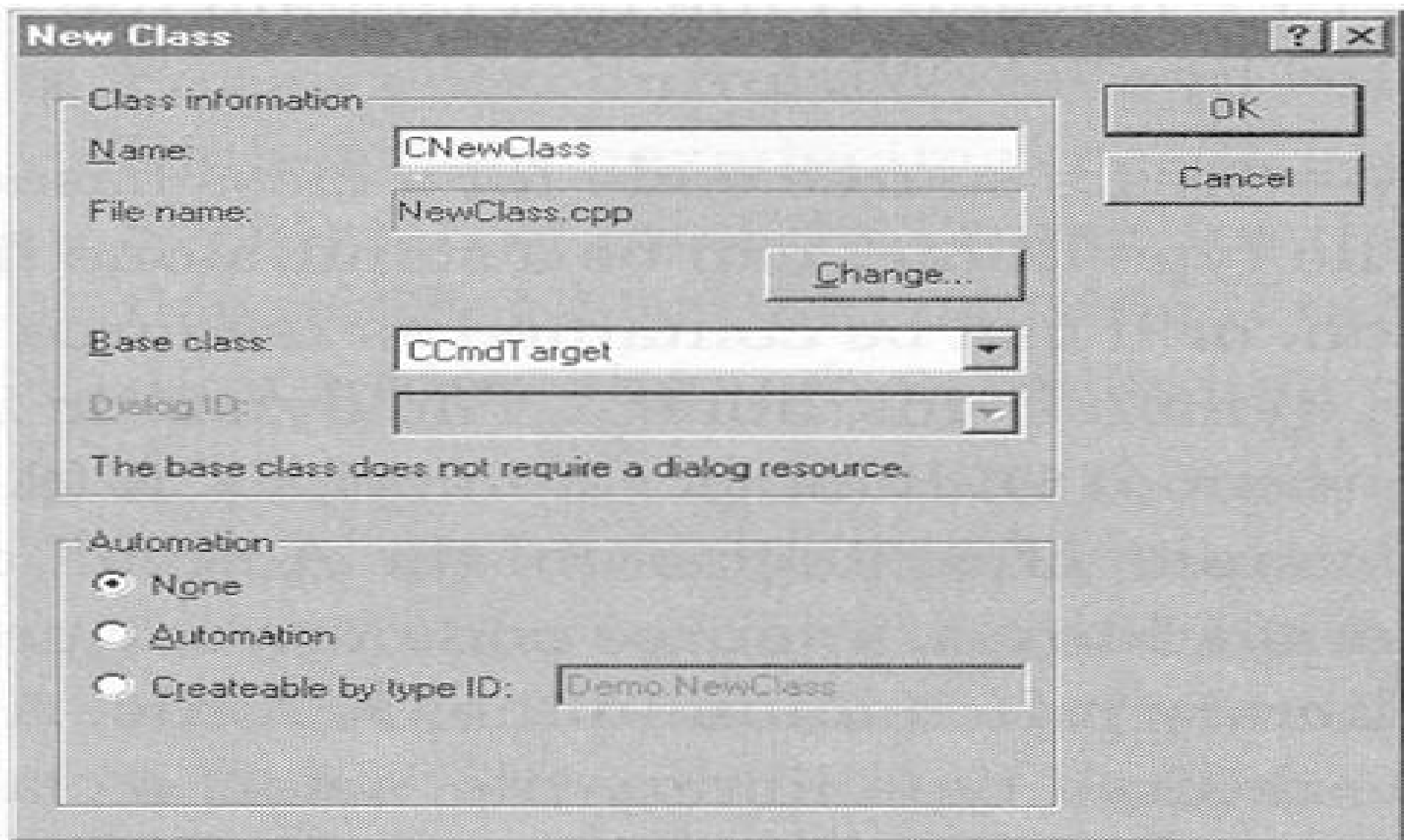


图 6-4 从 ClassWizard 中调用的 New Class 对话

为了命名类源文件，ClassWizard 向你所指定的类名添加 CPP 和 H 文件扩展，去掉任何 C 前缀，如果你想采用其他名称，请单击 Change 按钮。

Dialog ID 控件是否在 New Class 对话框中处于活动状态取决于所选中的基类。当你作为基类从五个 MFC 对话类(CDialog, CProperty Page, CFormView, CRecordView 或 CDaoRecordView)中进行选择的时候，Dialog ID 控件处于激活状态，提示与新类相关联的对话的资源标识符。创建基于对话的类的最好方法是，首先在对话编辑器中设计和保存对话资源，然后，在编辑器仍然处于激活状态且有输入焦点的时候，访问 ClassWizard，来为对话创建新类。在本章稍后的例子中，我们将这样做。

Automation 组框中单选按钮的外观取决于选中的基类是否支持 Automation。当你选中像 CHttpFilter 这样的基类的时候，将出现一条十分详细的信息，通知你 Automation 不是一个选项。不过，对于可察觉 Automation 的基类，Automation 单选按钮是启用的。打开该按钮，将指导 ClassWizard 编写代码来生成源文件，从而使得这个新类成为一个可编程的对象，而且对于像 Microsoft Excel 这样的 Automation 客户应用程序来说是可见的。本书附录 B 标识了那些 MFC 类可察觉 Automation。如果你想了解有关 Automation 的更多信息，由 David Kruglinski 编著的 Inside Visual C++中，专门有一章来讨论这一主题。

单击 Createable By Type ID (通过类型 ID 可创建) 单选按钮将生成代码，这些代码允许其他的 ActiveX 应用程序创建你的新类的 Automation 对象。ClassWizard

自动将项目名和类名组合起来，以形成编辑控件中所显示的类型标识符，以及一个用来帮助确保标识符为独一无二的方案。这个类型标识符也就是大家所熟知的程序设计标识符，它可以被 ActiveX 客户应用程序用来指定对象。例如，一个 Excel 宏可以创建一个 CNewClass 对象：

```
Set DemoObj=CreatObject("Demo.NewClass")
```

添加非 MFC 类

ClassWizard 对话不是向项目添加类的唯一方法。单击 Insert 菜单上的 New Class 命令，将显示如图 6-5 所示的对话，它与图 6-4 中的对话共享相同的名字，而且看起来十分相似。唯一的区别是，在对话的顶部多了个 Class Type 框，它是用来指示 New Class 命令的双重特性的。

对于类的类型，默认选择是 MFC Class，这将导致对话的行为与图 6-4 中的 New Class 对话一致。选择 Base Class 控件中的一个基本 MFC 类，并且像以前一样设置 Automation 单选按钮。如果想创建一个基于表单的新类，请在 Class Type 框中选择 Form Class，然后选择 CFormView，CDialog，CRecordView 或 CDaoRecordView 的一个基类。对于类的类型，第三个方案是 Generic Class，它将导致 Visual C++ 为不是从 MFC 中派生的类生成存根代码。Visual C++ 仅仅借用来自 ClassWizard 的技术，如果你愿意看到 New Class 命令只是作为 ClassWizard 中的另一部分，当然也可以。但是一定要切记，图 6-4 所示的 New Class 对话的变种仅仅可用于从 MFC 派生而来的类，对于它，你总是可以应用 ClassWizard

的全部功能。用 New Class 命令创建常规的非 MFC 的类将生成存根代码，但同时也舍弃了 ClassWizard 的其他功能。

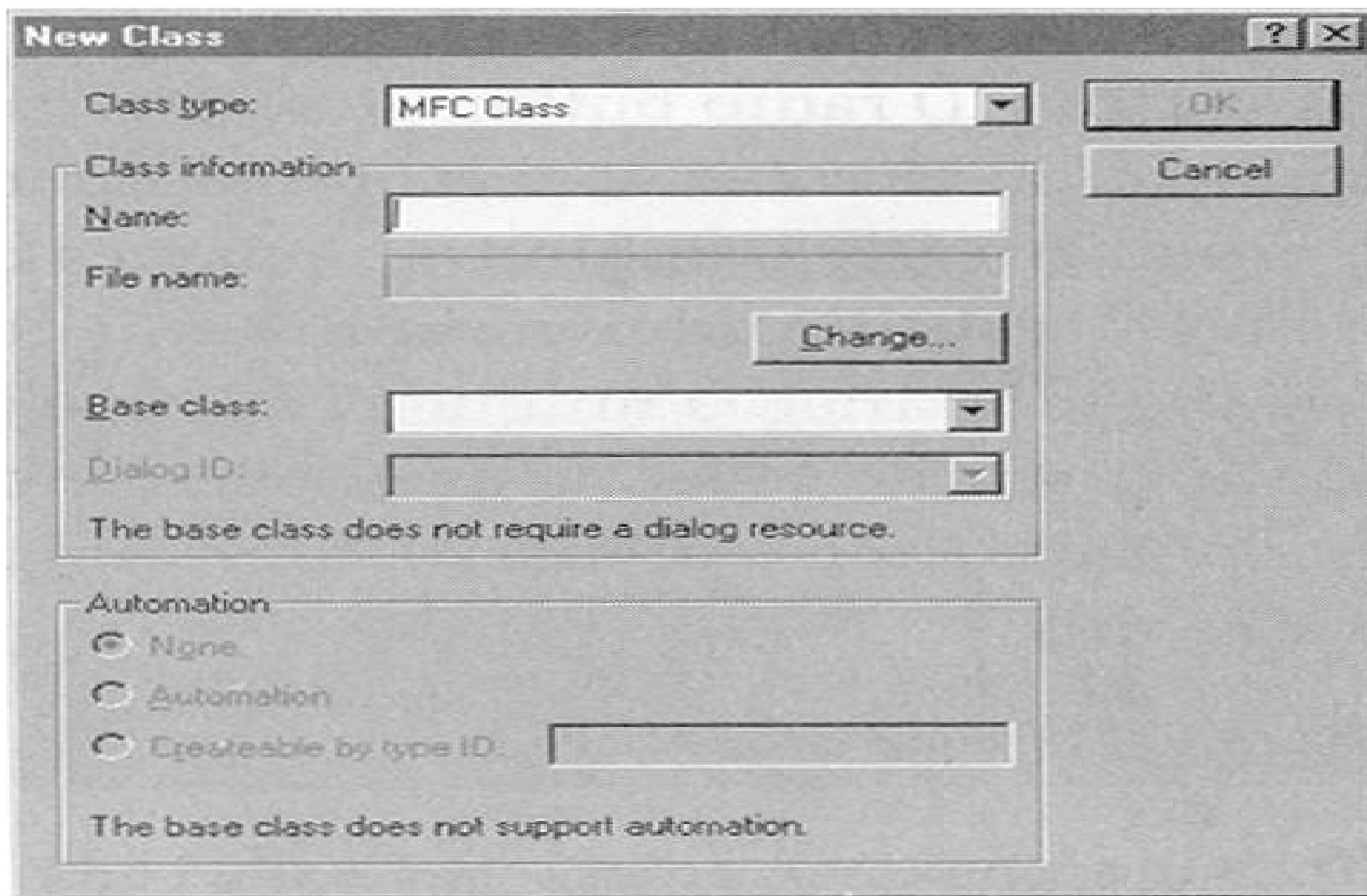


图 6-5 通过从 Insert 菜单 New Class 命令进行访问的 New Class 对话

对话中的 Generic Class 设置启用一个列表框，你可以从中为新类选择一个基。生成的源代码仅仅由用于类构造器和析构器的存根函数组成，这些函数包含在一个为该类命名的 CPP 文件和 H 文件之中。例如，这是 New Class 命令为从 CBaseClass 中派生出来的类写到头文件中的声明。

```
class CDerivedClass : public CBaseClass  
  
{  
  
public:  
  
    CDerivedClass();  
  
    virtual ~CDerivedClass();  
  
};
```

除了 New Class 命令之外，Visual C++ 还提供了 ClassWizard 的其他分支。我们将在下面看到，WizardBar 作为到达 ClassWizard 的侧门，它通常比主要入口更加方便。

WizardBar (向导栏)

WizardBar 是一个可停靠的工具栏，当你在文本和对话编辑器中四处移动的时候，它可以记住插字符的位置或当前选中的部分。无论你当前正在处理的是哪一类，该栏都会连续不断地调整它的外观。例如，打开用于 CMainFrame 类的实现文件将自动为该类初始化 WizardBar，提供一种方便的方法迅速找到成员函数的定义和声明。和其他任何 Visual C++ 工具栏一样，你可以通过单击 Tools 菜单上的 Customize 命令，并在 ToolsBar 选项中选中合适的复选框来将 WizardBar 打开或关闭。

图 6-6 显示了 WizardBar 的类型视图。

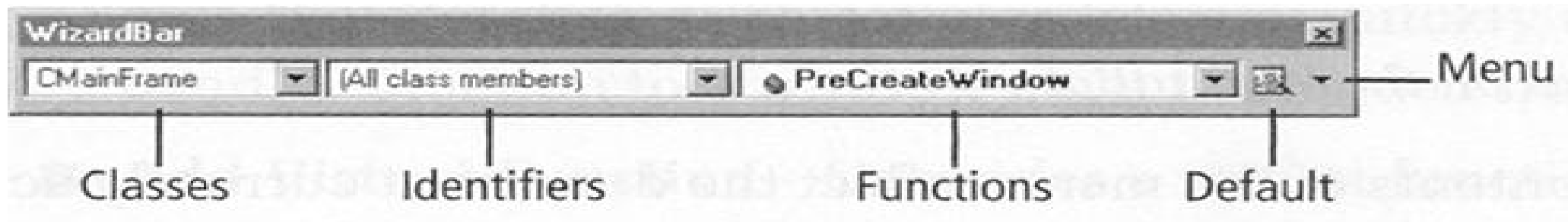


图 6-6 WizardBar

WizardBar 的三个组合框用于封装在 MFC ClassWizard 对话的 Message Maps (消息映射) 选项卡中显示的信息, ClassWizard 中所做的任何改变都可以在 WizardBar 中连续不断地反映出来。表 6-6 说明了 WizardBar 的框和按钮。

WizardBar 最右边的箭头按钮显示的是图 6-7 中所示的选项的下拉菜单。

表 6-6 WizardBar 上的控件

向导控件	说明
Classes (类)	显示编辑器中当前处于打开状态的类名, 并且提供一个项目中所有类的下拉列表。当既没有文本编辑器也没有对话编辑器具有焦点的时候, 框中条目是灰色的
Identifiers (标识符)	列出当前所使用的符号标识符(当前类的名字出现在 WizardBar 的 Classes 框中)
Functions (函数)	包含用于当前类的虚拟函数和 CCmdTarget 的过程名

续表

Default (默认)

单击位于 WizardBar 最右端的“棍棒”图标，将执行两行 WizardBar 菜单的默认命令。默认命令取决于当前的支持以及在 WizardBar 中选中的部分。例如，如果类实现文件在文本编辑器中处于打开状态，并且插字符在函数块之内，那么默认的操作将是 Go To Function Declaration (转向函数声明)，即打开类的头文件，并定位在函数原型中的插字符位置，然后，默认的操作将变为 Go To Function Definition (转向函数定义)，将插字符发回到实现 CPP 文件，默认操作总是在 WizardBar 菜单上列出的第一个命令，如图 6-7 所示

Menu (菜单)

显示可用选项的 WizardBar 菜单，将在下面列出的项目符号列表中详细说明

菜单的内容反映了当前处于活动状态的文档，因此，当源文档在文本编辑器中处于打开状态的时候，像 Go To Next Function 这样的命令才可用。在菜单命令的以下说明中，词“当前”指的是 WizardBar 中的设置。例如，当前类是在 WizardBar 的 Classes 框中显示的类，如图 6-6 所示。

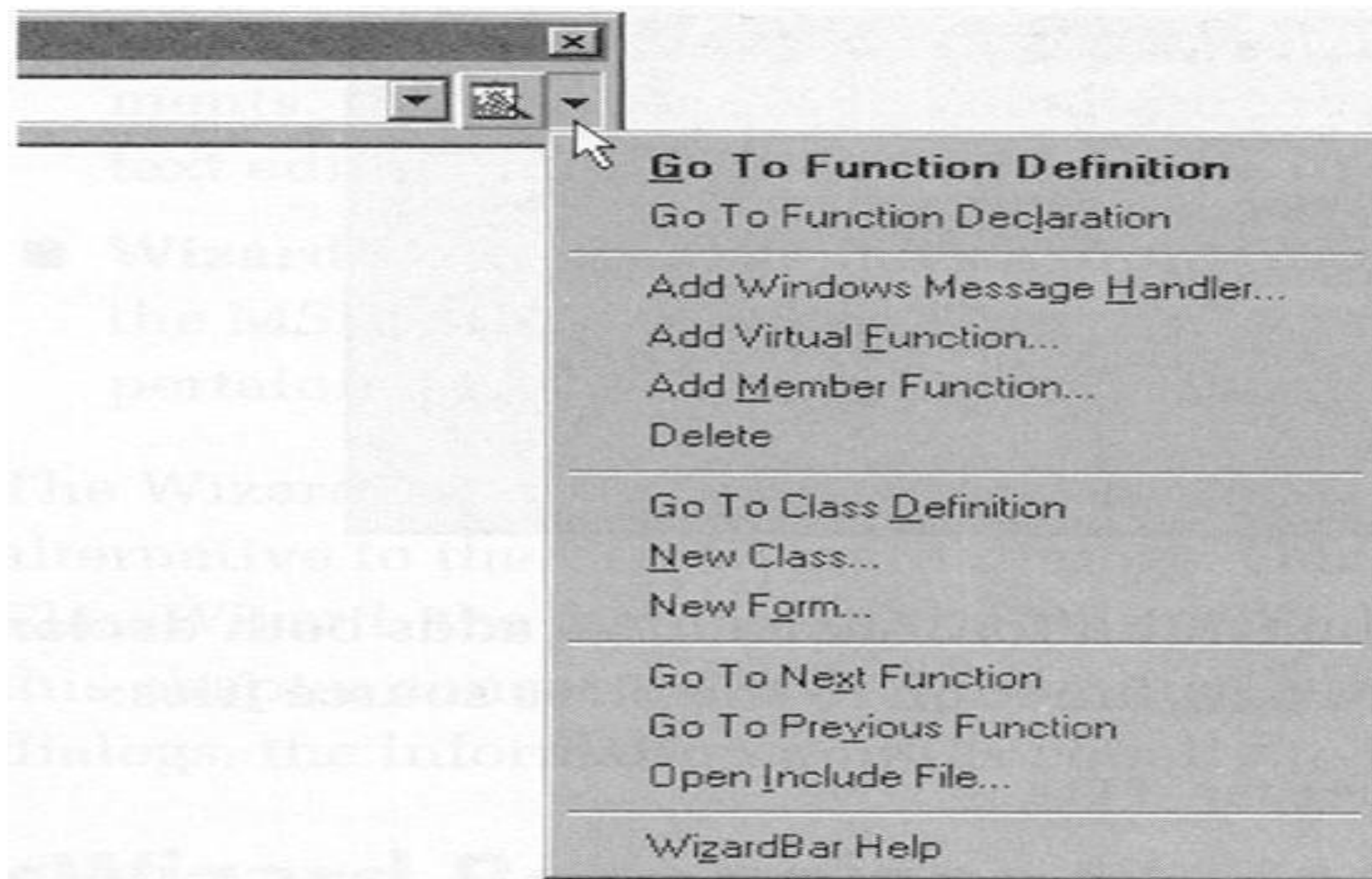
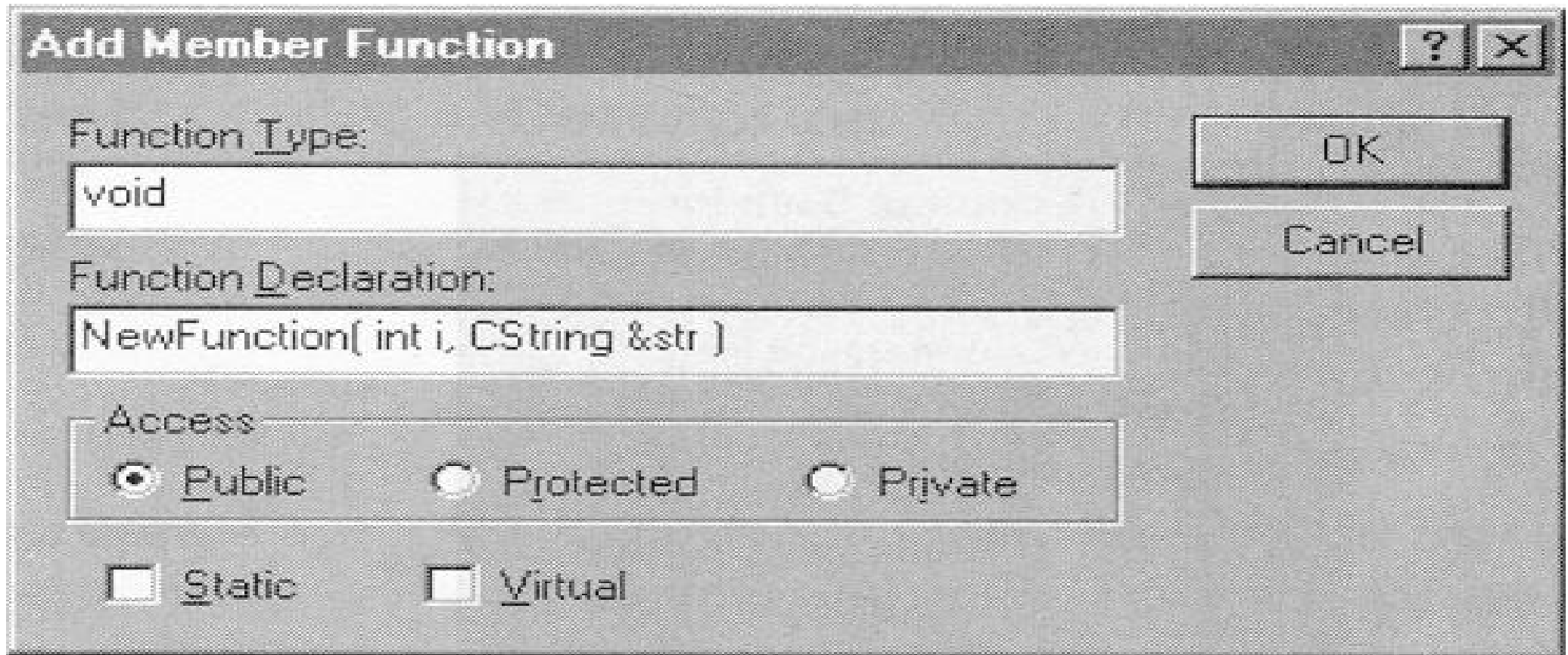


图 6-7 通过单击 WizardBar 上的箭头按钮显示的 WizardBar 菜单

- **Go To Function Definition (转向定义)**：如果必要的话，打开源 CPP 文件，并在当前函数的第一行放置插字符，在 WizardBar 的 Functions 框中标识。
- **Go To Function Declaration (转向函数声明)**：在当前函数的原型处放置插字符。
- **Add Windows Message Handler (添加窗口消息处理函数)**：调用 New Windows Message Handler (新建窗口消息处理程序) 对话。在该对话中，可以快速从 CWnd 开始，向随后的类中添加存根消息处理程序函数。
- **Add Virtual Function (添加虚拟函数)**：超越基类的虚拟函数。该命令显示两个列表，一个包含有可以为超越提供的虚拟函数，另外一个显示已被当前类超越的那些函数。这些列表提供了与 MFC ClassWizard 对话中的 Messages 框相同的信息，但更加易于使用，且更加便于浏览。
- **Add Member Function (添加成员函数)**：向当前类添加存根成员函数。像下图如示那样，输入该函数的返回类型、声明和访问标签。



Visual C++将这个新函数的声明和定义代码都添加到类源文件。

```
//In the class header file
```

```
public:
```

```
void NewFunction(int I, CString &str);
```

```
// In the class CPP implementation file
```

```
void CMainFrame::NewFunction(int I, CString &str)
```

```
{
```

```
}
```

- **Delete (删除)：**从类的来源中删除当前函数。在查询确认之后，Visual C++把该函数的声明和注释从实现代码中删除。编辑器的 Undo 命令不能恢复已被删除的函数。
- **Go To Class Definition (转到类)：**在当前类的构造器函数的实现代码处放置插字符。
- **New Class (新建类)：**与 Insert 菜单上的 New Class 命令是等价的。
- **New Form (新建表单)：**显示 New Form 对话，该对话可用于生成代表从 CFormView, Cdialog, CRecordView 或 CDaoRecordView 中派生的基于表单的新类的源代码。该操作与从 New Class 对话中选中 Form

Class 类型产生的结果相同。

- Go To Next/Previous Function (转向下一个/前一个函数)：在实现文件中，将插字符向前或向后发送到下一个函数定义。在头文件中，将插字符发送到邻近的函数声明。
- Open Include File (打开包括文件)：扫描当前文档查找 #include 语句，然后展示一个带有包括文件的列表。如果想在文本编辑器中打开一个文件，请在列表中双击它的文件名。
- WizardBar Help (向导栏帮助)：在 MSDN Help 窗口中显示一个题为“OverView: WizardBar”的话题。从那里，你可以跳到与 WizardBar 有关的其他话题。

WizardBar 提供了到 ClassWizard 的快速访问，是 ClassWizard 对话的替代方法，使用起来很方便。一旦你知道了 ClassWizard 的输入与输出，WizardBar 的功能看上去就很熟悉了。尽管本章将重点放在通过各种对话框来访问 ClassWizard 之上，该信息同样可以应用到 WizardBar 工具。

ClassWizard 如何识别类

CLW 数据库文件是以 ASCII 文本形式存在的，因此你可能对使用文本编辑器阅

读它比较感兴趣。数据库将项目中的每个类进行分类，并记录该类的基和源文件。对话、菜单和加速键这样的资源数据，也用它们的标识符在文件中运行分类。

为了构造 CLW 数据库文件，Visual C++ 扫描附加在该项目之上的每个源文件，寻找特殊的注释行。在前面的 AppWizard 程序中，你已经看到了这些注释；它们以 {/或//} 开头，以起到标记声明、消息映射条目和其他与类成员相关的代码的作用，除了为 CLW 数据库中包括的东西标识类信息之外，注释没有其他的作用。如果你没有使用 ClassWizard 写一个类，而后来又想使用 ClassWizard 向该类添加其他的函数或数据，那么你必须包括注释行。否则，注释就不是必要的。

每个注释限定符都包含表 6-7 中列出的十三个关键字中的一个。大多数关键字是成对使用的，一个关键字用来标记类的头文件中的一条声明，而它的对应物（带有 _MAP 后缀）标记的是 CPP 文件中消息映射中的相应条目。例如，这里是 Visual C++ 如何识别名为 CDemoApp 中消息处理程序函数的例子，其中 CDemoApp 是从 MFC 的 CWinApp 中派生出来的。在 CPP 实现文件中，特殊的 AFX_MSG_MAP 注释将代表处理程序函数 CDemoApp::OnAppAbout 的消息映射条目括了起来，使得该条目可以被 ClassWizard 所识别。

```
BEGIN_MESSAGE_MAP(CDemoApp, CWinApp)
```

```
//{{AFX_MSG_MAP(CDemoApp)
```

```
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
```

```
//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
```

表 6-7 ClassWizard 所需要的注释关键字

关键字	文件	说明
AFX_DATA	H	用于对话数据交换的成员变量声明
AFX_DATA_INIT	CPP	对话类的构造器中对话数据交换变量的初始化
AFX_DATA_MAP	CPP	对话类的 DoDataExchange 函数中的对话数据交换调用
AFX_DISP	H	自动声明
AFX_DISP_MAP	CPP	自动映射
AFX_EVENT	H	ActiveX 事件声明
AFX_EVENT_MAP	CPP	ActiveX 事件映射

续表

AFX_FIELD_INIT	CPP	初始化记录集类的构造器中的记录 字段交换成员变量
AFX_FIELD_MAP	CPP	记录类的 DoFieldExchange 成员函数 中的记录字段交换调用
AFX_MSG	H	在消息映射中出现的函数变量
AFX_MSG_MAP	CPP	消息映射条目
AFX_VIRTUAL	H	虚拟函数超越的声明

ClassWizard 还需要读取该函数的原型，因此头文件中的 OnAppAbout 声明相应的 AFX_MSG 注释进行标记：

public:

```
//{{AFX_MSG(CDemoApp)
```

```
afx_msg void OnAppAbout();
```



```
//}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()
```

有了这条记录在 CLW 数据库中的信息，ClassWizard 就可以知道，CDemoApp 类包含有一个名为 OnAppAbout 的成员函数，当程序接收到包含在 WM_COMMAND 消息中的 ID_APP_ABOUT 标识符的时候，便调用该成员函数。函数中的 afx_msg 前缀是为了 ClassWizard 才包括进来的，否则，它是不需要的。ClassWizard 和 AppWizard 可以在生成源代码的时候自动添加正确的限定符注释。但是，如果你想在不是用 AppWizard 开始的项目中，使用 ClassWizard 的话，表 6-7 显示了如何转换现有的源文件，使它们可以被 ClassWizard 所识别。该表说明了由注释关键字所标记的函数和声明的类型，并且指示了关键字在哪个源文件中使用。

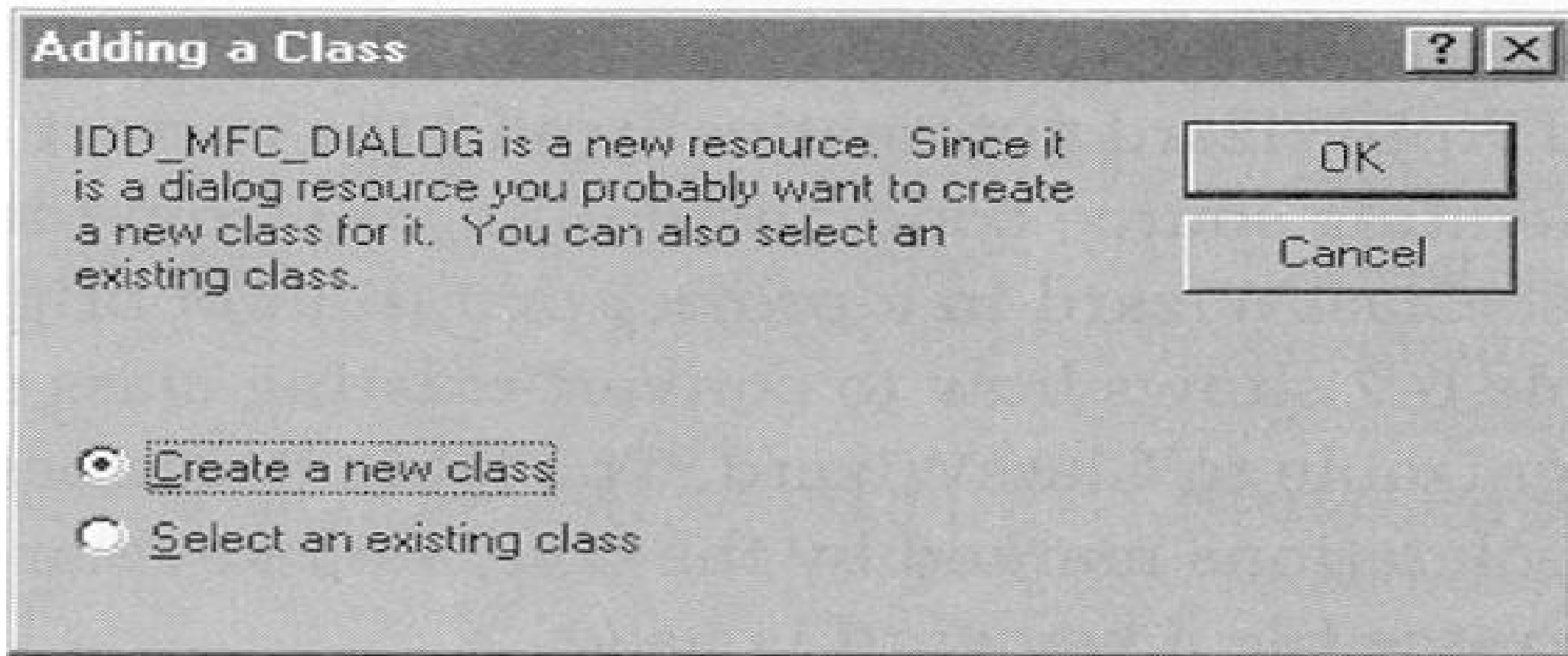
用 ClassWizard 创建对话类

在为新的 MFC 程序设计了新的对话资源之后，你必须还提供从 CDialog(或者其他基于对话的类)中派生的类，来显示对话和响应它的消息。通常情况下，在用对话编辑器完成之后，ClassWizard 是你的下一个逻辑步骤。例如，当我们在第 5 章中编写 MfcTree 程序的时候，就曾经用过 ClassWizard。MfcTree 显示的是一个包含有一些 MFC 类的树状列表的简单对话。在第 5 章中，我们编写了一个叫做

CMfcDlg 的类来运行这个对话,首先使用文本编辑器来编写主要的实现文件和头文件,名字分别为 MfcDlg.cpp 和 MfcDlg.h,然后通过从 Insert 菜单选择 Add To Properties 命令来将这些文件添加到项目中。这两个步骤都可以由 ClassWizard 运行处理。

做为例子说明,让我们再次创建主要的 MfcDlg.cpp 和 MfcDlg.h 文件。这次使用 ClassWizard,而不是在文本编辑器中从零开始编写文件。为了清楚地了解 ClassWizard 自动完成的工作,你可能想回顾一下第 5 章题为“为 CMfcDlg 对话类添加源文件”一节的内容。该节介绍了创建 MfcTree.exe 所使用的五个步骤中的第三步。

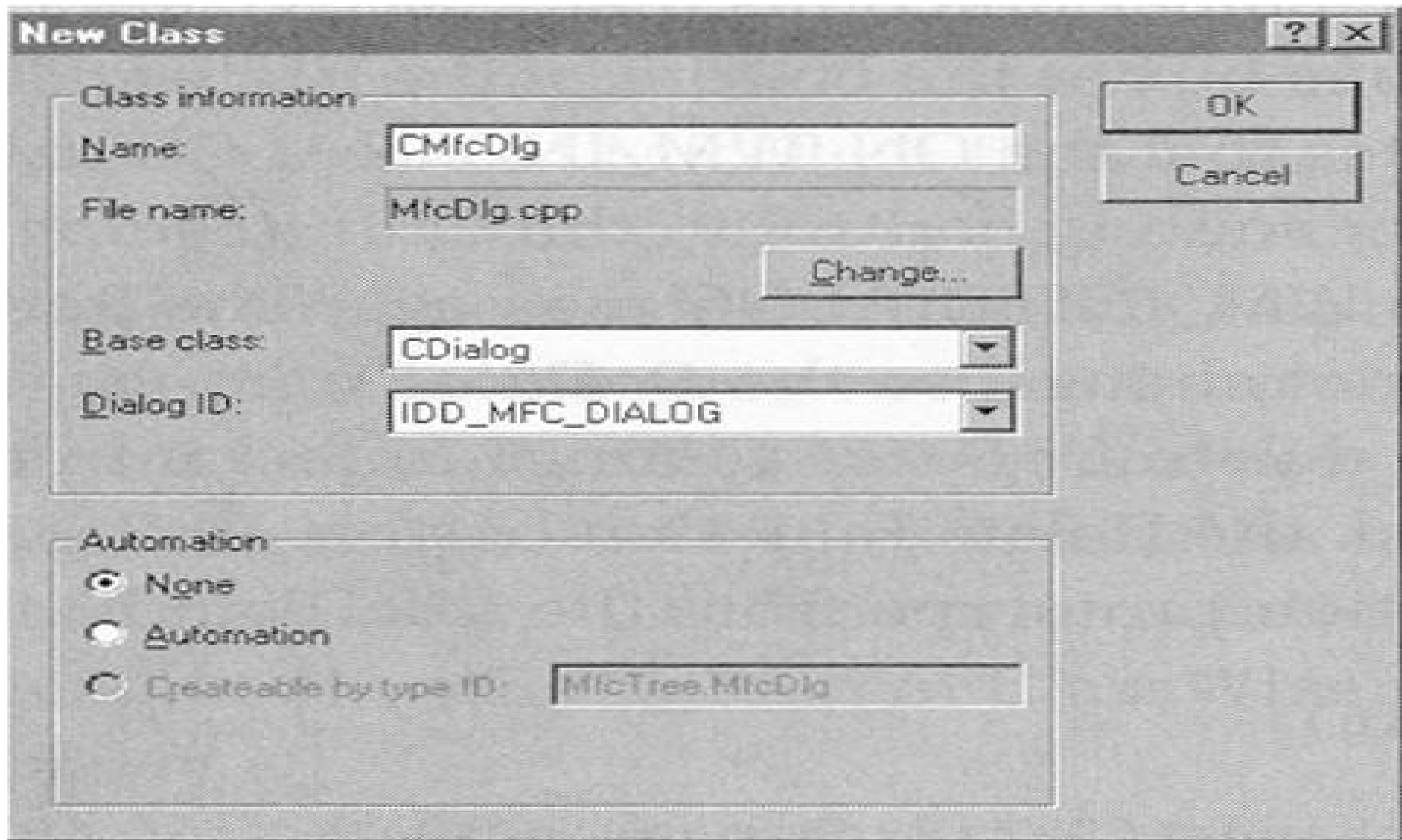
在开始这个练习的时候,假设 MfcTree 的对话刚刚创建,并且保存在对话编辑器之中。在 IDD_MFC_DIALOG 资源仍在对话编辑器中处于打开状态的时候,请单击 View 菜单上的 ClassWizard。ClassWizard 将检测新的对话资源,并且询问你是否愿意为它创建一个新类。



我们可以向已经存在源代码的类中添加类。例如，你在对话编辑器中创建和保存资源之前，可能已经编写了对话类。在这种情况下，你应该单击 Select An Existing Class（选择现有类）单选按钮，将对话附加到类上。并且阻止 ClassWizard 生成

新的 CPP 和 H 源文件。不过，对于本例，既然 CMfcDlg 类还没有存在，那么，Create A New Class（创建新类）单选按钮便是正确的选择。

单击 OK 按钮打开相似的 New Class 对话。在 Name 框中输入 CMfcDlg，将为新类取一个名字。跳过 Change 按钮，接受 ClassWizard 为源文件建议的 MFcdlg.cpp 和 MfcDlg.h 默认名字。因为它知道我们正在为这个新对话创建一个类，所以 ClassWizard 已经选中了 CDialog 作为基类。它还用对话编辑器写向 MfcTree.rc 文件的对话标识符 IDD_MFC_DIALOG，来填充 Dialog ID 框，图 6-8 显示了 New Class 对话框的样子。



6-8 在 ClassWizard 的 New Class 对话框中创建新 CMfcDlg 类

单击 OK 按钮来关闭 New Class 对话，并再现如图 6-9 所示的 MFC ClassWizard 对话，这就是你为新的 CMfcDlg 类生成主体代码的地方。CMfcDlg 仅仅需要 OnInitDialog 函数，它刚好在 IDD_MFC_DIALOG 对话出现之前获得控制权。在 Object IDs 框中，选中 CMfcDlg，然后双击 Messages 栏中的 WM_INITDIALOG 来添加 OnInitDialog 函数。

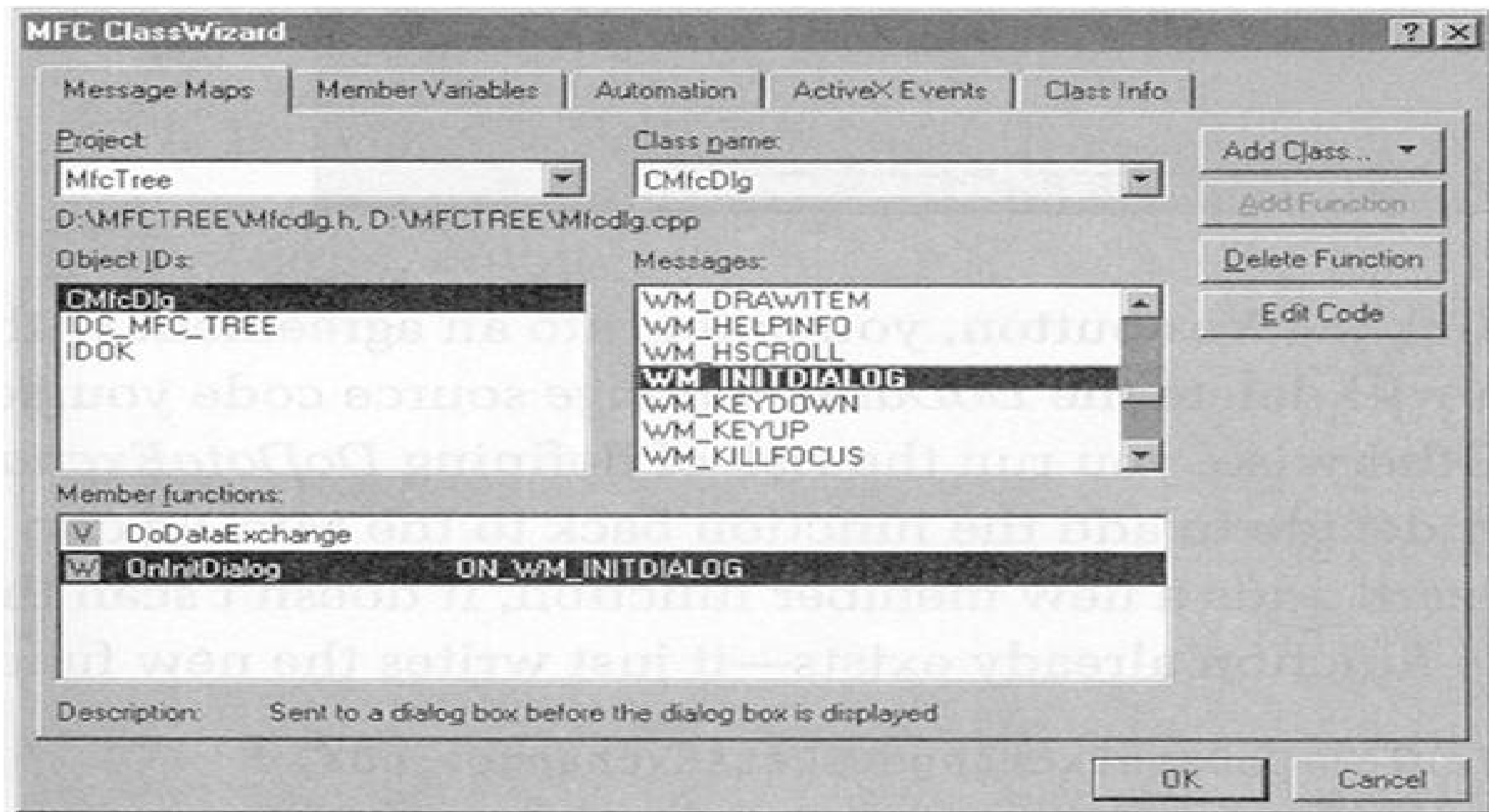


图 6-9 向 CMfcDlg 类添加 OnInitDialog 成员函数

ClassWizard 创建的两个成员函数的名字出现在对话底部的 Member Functions 框中。框中选中的项目用来指示当程序接收到 WM_INITDIALOG 消息的时候，ON_WM_INITDIALOG 消息映射宏负责对 OnInitDialog 函数的日常控制。CMfcDlg 树状视图中的项目保持不变，因此，也就不需要 DoDataExchange 函数。如果你想删除 DoDataExchange，请在 Member Functions 框中选中它，并单击 Delete Function 按钮。不过，这样做只能从 MfcDlg.h 头文件中删除函数原型：

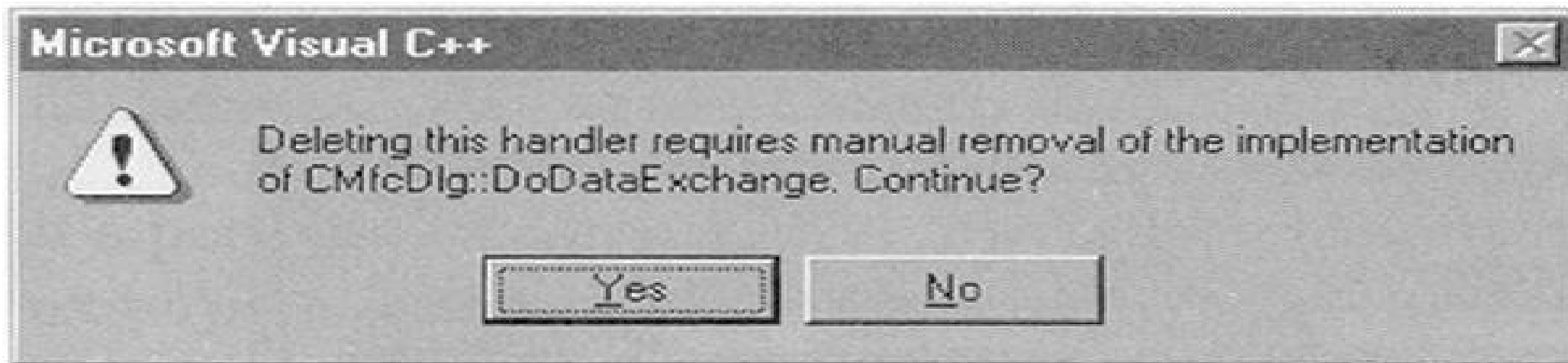
```
//{{AFX_VIRTUAL(CMfcDlg)
```

```
protected;
```

```
virtual void DoDataExchange(CDataExchange* pDX); //DDX/DDV support
```

```
//}}AFX_VIRTUAL
```

DoDataExchange 源代码仍旧在 MfcDlg.cpp 文件中存在。原因是 ClassWizard 仅仅删除了注释限定符之内的代码，而从不删除实现代码。一条消息可以提醒你注意这个事实：



如果单击 Yes 按钮，就是同意将在文本编辑器中删除 DoDataExchange 源代码。如果你后来想将函数添回到 MfcDlg.cpp 文件，便会有两次定义 DoDataExchange 的危险。当 ClassWizard 添加一个新的成员函数的时候，它应首先扫描文件，看看函数是否已经存在，它只是编写新的函数外壳：

```
void CMfcDlg::DoDataExchange(CDataExchange* pDX)
{
    Cdialog::DaDataExchange(pDX);
}
```

如果删除 DoDataExchange 的原始实现失败，将会导致编译错误，因为编译器不能接受同一函数的两个定义。

在退出 MFC ClassWizard 对话的时候，ClassWizard 将向项目添加新的 MfcDlg.cpp 和 MfcDlg.h 文件，这两个文件都包含有助于 CMfcDlg 类的存根函数。一旦你向第 5 章中介绍的那样，在类的 OnInitDialog 函数中添加了代码，这些新文件在功能上与从头开始编写的文件是等价的，而且，更加易于创建。如果你愿意比较新的源代码，可以在配套光盘的 Chapter.05\MfcTree 文件夹中找到原来的 MfcDlg 源文件。在 ClassWizard 的帮助下生成的相应的 MfcDlg.cpp 和 MfcDlg.h 文件位于 Chapter.06\MfcTree 文件夹中。

第 7 章 Gallery（组件集）

Gallery 是一类工具箱，Microsoft 在它里面放置了一类叫做组件的“罐装代码”，你可以把它们添加到自己的项目中。还可以使用 Gallery 来保存你以后想再使用的任何属于自己的类。Gallery 保存三种主要类型的组件：源代码，对话资源和 ActiveX 控件。通过 Gallery，你可以为了寻找某个组件而浏览自己的硬盘或整个网络，然后单击一下鼠标，将它插入到你的项目中。因为组件可以存储在任何地方，所以 Gallery 既可以作为个人代码存储库，存储组件供你个人使用，又可以作为全局性的代码存储库，允许开发网络链向它来共享通用组件集。

Gallery 提供的预先包装好的组件存储在由 Visual C++ 安装程序创建的两个文件夹中。一个文件用于源代码组件，另一个文件夹用于 ActiveX 控件。Visual C++ Components 文件夹包含有到动态链接库的快捷方式，它可以自动向项目添加源代码和资源。第二个文件夹名字叫做 Registered ActiveX Controls，它包含有到在你的系统上注册的所有 ActiveX 控件的快捷方式链接。有些注册的控件是由 Visual C++ 经过许可后提供的，因此，你可以在自己的应用程序之中重新分布它们。有关 ActiveX 控件有很多地方值得一提，它们中的大部分不直接附属于 Gallery，因此，直到本书第四部分，才能涉及这方面的话题。本章把重点放在如何访问 Gallery 以及创建你自己的可重用的对象集上。

对于预包装的组件来说，Visual C++不是唯一可提供的资源。许多销售商以源代码和二进制两种形式提供组件工具，并且在商业期刊和 Internet 上做广告。预包装的组件比你自己创建的更加方便，因为它们可以自动实现向项目添加组件的全部过程。通过操作 Visual C++环境加载和运行的可执行库，这些专业组件可以通过在必要的时候添加函数和 #include 语句（通常很少或干脆就没有为你留下编程工作的地方）来插入图形资源，并重写现有的项目文件。预包装组件通常提供有它们自己的在线帮助。

Gallery 在如图 7-1 所示的 Components And Controls Gallery 对话框中显示它的组件。当一个 C++项目在环境中处于打开状态的时候，可以通过将光标暂时停留在 Project 菜单上的 Add To Project（添加到项目）命令上来访问 Gallery 对话框。该操作将显示一个二级下拉式菜单，你可以从中选择 Components And Control（组件和控件）命令。当你从列表中选中一个组件时，如果该对话框的 More Info 按钮是启用的，那么该组件便可以通过在线帮助来描述自己。如果想把一个 Gallery 组件插入到项目中，请选中它的图标，并单击 Insert 按钮。

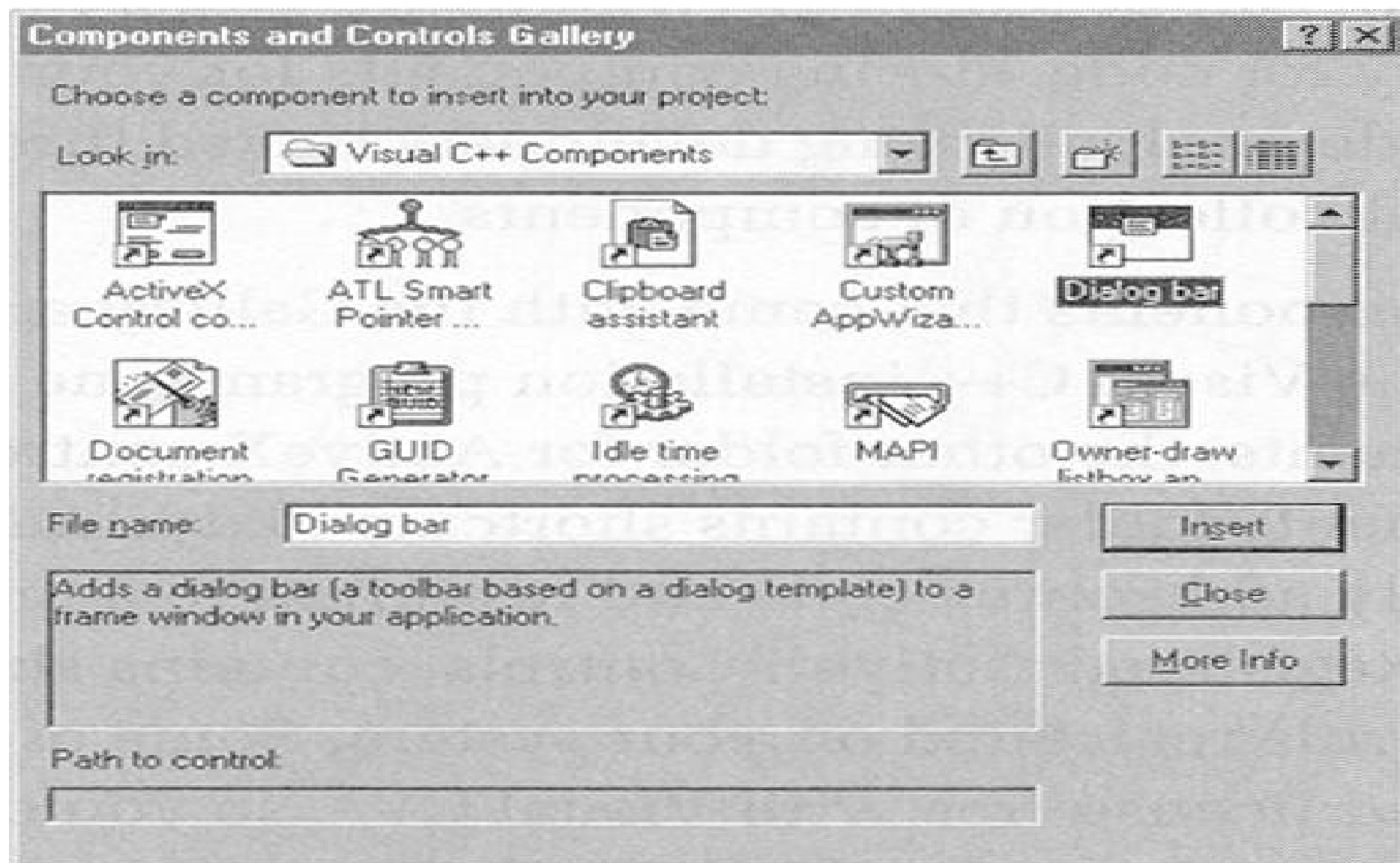


图 7-1 Gallery 的 Visual C++ 组件文件夹

在 Visual C++ 版本 4 中，如果你总是用 AppWizard 创建项目，Gallery(就是所熟知的 Component Gallery)将会变得很拥挤。每当 AppWizard 执行的时候，它都自动向 Component Gallery 添加新类，并且安装所有的项目类(例如 CMainFrame 和 CAboutDlg)作为源代码组件。在默认状态下，ClassWizard 还向项目分类添加你所创建的任何新类。清除 Component Gallery 中不想要的附件的责任就交给用户了，尽管许多程序员只是忽略展开的组件列表，或没有注意到 AppWizard 和 ClassWizard 在幕后做了些什么。整个 Component Gallery 数据库存储在一个叫做 Gallery.dat 的单个文件之中，因此没有方便的方法可以与其他开发者一起共享组件。

尽管版本 6 中 Gallery 的函数没有改变，但它的方法已经得到了很大改进。Gallery.dat 不再存在，它的数据库已被存在于磁盘或网络上任何地方的单个组件文件所取代。AppWizard 和 ClassWizard 都不再向 Gallery 添加类。正如我们将在下一章中所看到的，ActiveX 控件可以成为 Gallery 数据库的一部分，而无需你的许可，但是，对于其他组件类型则不可以。你必须采用特殊的步骤才能把组件添加到 Gallery，其中包括你想从版本 4 中继承的任何组件。下一节将解释如何向 Gallery 添加新组件，现在，让我们来看一看 Visual C++ 中提供的预包装组件。

例子：添加一个属性表

第 5 章介绍了如何使用编辑器来创建属性表对话，并将其融入一个程序。Gallery

提供了一个叫做 Property Sheet 的组件，利用它，工作起来就会更加容易。本节和下一节将阐述 Property Sheet 以及 Visual C++ 提供的一些其他组件，首先显示如何向叫做 Gadgets 的项目添加属性表，然后用从 Gallery 中借用的更多代码来充实和完善程序。Gadget 作为一个典型的 AppWizard 项目启动，然后，随着你不断地添入组件，它会变得越来越高级。下面是建立项目并添加第一个组件的步骤：

1. 从 File 菜单选择 New 对话，在 Projects 选项卡中选择 AppWizard(exe) 图标。键入 Gadgets 作为项目名并单击 OK。
2. 当 AppWizard 在屏幕上依次出现各个步骤时，在第一步中选中 Single Document 单选按钮，并且在第四步中解除对状态栏、停靠工具栏和打印支持的选择。在其他步骤中接受默认的设置。
3. 当 AppWizard 完成，而且 Visual C++ 打开项目的时候，从 Project 菜单中选择 Add To Project, 并单击位于次级菜单上的 Components And Controls 命令，来显示 Components And Controls Gallery 对话。
4. 在对话中打开 Visual C++ Components 文件夹，并双击 Property Sheet 组件图标(为了找到该图标，你可能不得不从头至尾滚动列表。Gallery 出现一个消息框，询问你是否确实想插入这个组件。
5. Visual C++ 加载并执行这个组件的可执行库，它将显示一个对话，询

问与你想创建的属性表类型有关的各种问题，只需单击 Next 按钮，接受所有的默认设置即可。完成之后，关闭 Gallery 对话。

你将会发现，Property Sheet 组件已经为新的属性表类的源文件自动放进了 Gadgets 文件夹。该组件还修改了 Gadget 项目中四个现有的文件，Resource.h 文件包含有属性表类所需要的定义，Gadgets.rc 文件已用代表两个属性页的脚本进行了更新，MainFrm.h 和 MainFrm.cpp 文件包含有用于新的名为 OnProperties 的函数的新代码。

OnProperties 函数中的“to do”注释解释了你必须将该函数连接到消息处理程序，这样，当用户单击菜单命令的时候，属性表才能显示出来。该操作仅需要两步：

- 1.在 MainFrm.cpp 中，将代表该函数的条目添加到消息映射。新的映射条目在这里显示为阴影，它必须插在 ClassWizard 注释限定符之间：

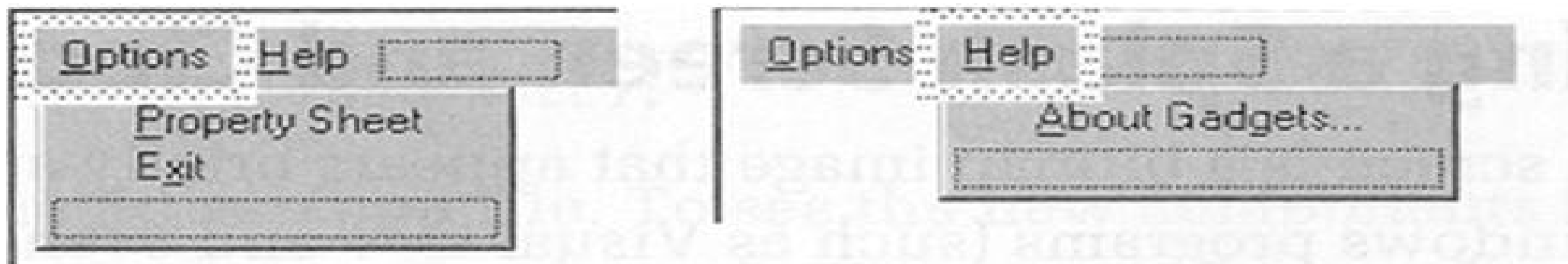
```
//{{AFX_MSG_MAP(CMainFrame)

ON_COMMAND(IDM_PROPSHEET, OnProperties)

//}}AFX_MSG_MAP
```

- 2.在 Workspace 窗口的 ResourceView 窗格中，展开 Menu 文件夹，并双击 IDR_MAINFRAME 菜单编辑器。编辑该菜单系统，使它如下图所

示：



当你向 Option 菜单添加 Property Sheet 命令的时候,一定要在 Menu Item Properties (菜单项属性) 对话框中给它一个 IDM_PROPSHEET 的标识符值。这与第一步中添加到消息映射中的阴影行所使用的标识符是相同的。

使用 Build 的工具栏, 可以将活动项目配置改为 Win32 Release, 创建 Gagents.exe, 并运行它。从程序的 Option 菜单中选择 Property Sheet 命令, 将显示如图 7-2 所示的属性表。在对话的这两页中, 准备使用对话编辑器向其中添加控件。从开始到完成, 整个操作仅需要几分钟。你可以轻轻松松地创建属性表, 这是前所未有的。

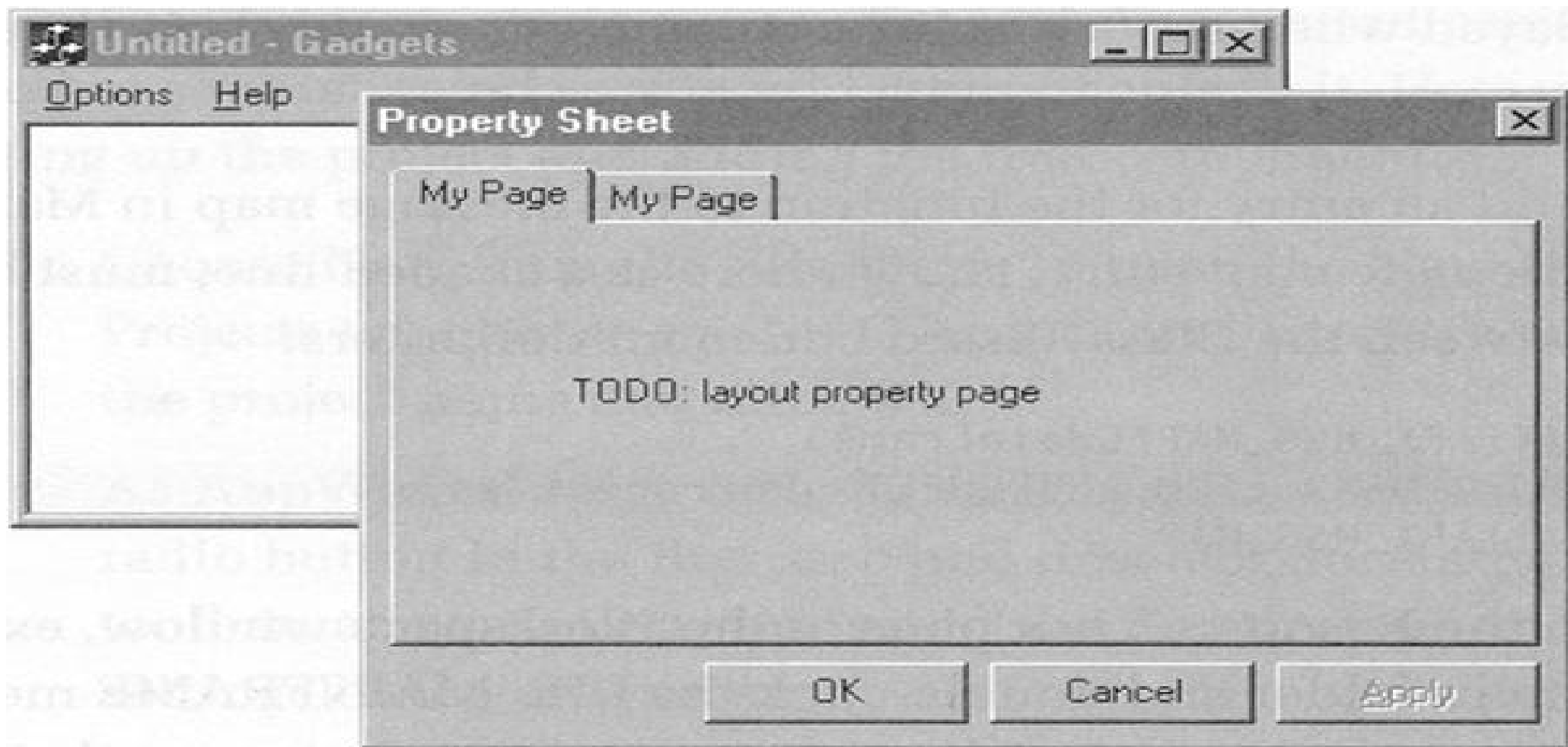


图 7-2 显示有属性表的 Gadgets 程序

例子：添加初始屏幕和时钟

初始屏幕是位图图像，它在程序启动的时候出现一会儿。在加载文件和实现其他初始化操作的时候，大的 Windows 程序 (例如 Visual C++，甚至是 Windows 自己) 通常要在显示一个主窗口之前出现一个初始屏幕。这不仅仅可以在程序忙的时候为用户提供一些吸引人的东西，而且还可以起到表达系统响应的效果。Gadget

程序十分小，因此，它不需要初始屏幕，但这也没关系。让我们添加一个带有时钟的状态栏。你可能不会想相信，这非常简单。

我们所需要的组件叫做 Splash Screen 和 Status Bar,在 Gallery 对话的 Visual C++ Components 文件夹中，它们是用这些大图标标识的。



通过双击组件图标可以将每个组件添到打开的 Gadgetes 项目之中。Status Bar 组件运行了一个对话，让你选择新状态栏是否显示日期和时间。如果想在状态栏中包括时间，请在第二步中单击 Use System Default(使用系统默认值) 单选按钮(对于其他步骤接受默认设置)。在关闭 Gallery 时，你将会发现创建在最右边带有小时钟状态栏的代码已添加到 MainFrm.cpp 文件之中。你只需将阴影行添向 MainFrm.cpp 文件的消息映射中即可：

```
//{{AFX_MSG_MAP(CMainFrame)

ON_COMMAND( IDM_PROPSHEET, OnProperties )

ON_WM_CREATE ()

//}}AFX_MSG_MAP
```

Splash Screen 组件提供了一个名为 Splsh16.bmp 的文件，它包含一个位图图像。

对于你自己设计的初始屏幕，在图形编辑器中创建一个新的 16 色图像，并将其保存在相同的文件名之下，覆盖掉原来的 Splash16.bmp 文件。初始屏幕默认的持续时间是 3/4 秒 (750 毫秒)，但是，你可以通过在 Splash.cpp 文件中修改如下代码来改变它：

```
SetTimer(1,750,Null);
```

如果想看到有效的新组件，请重建 Gadgets 程序，并运行它。图 7-3 显示了新程序的样子。

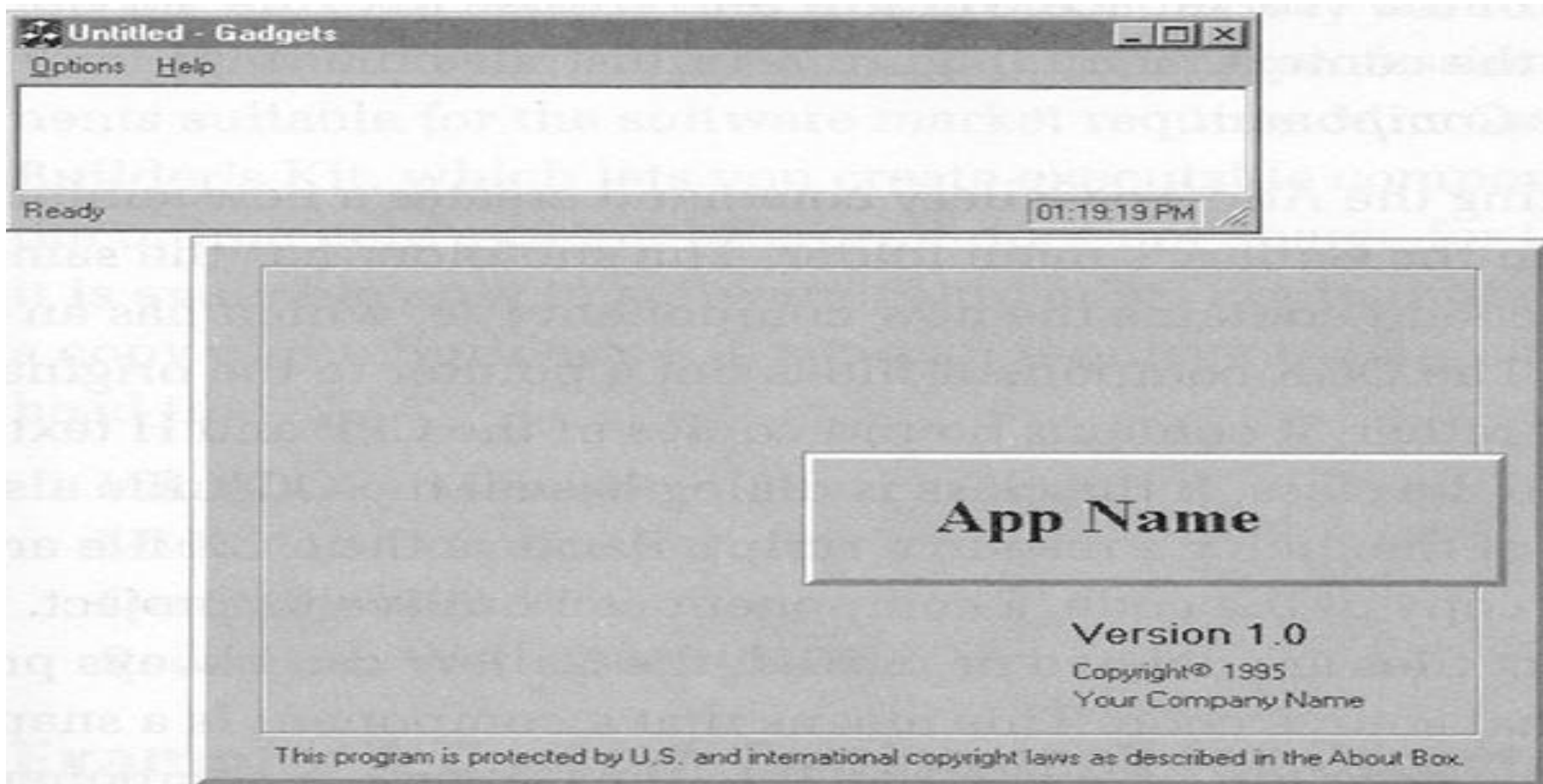


图 7-3 带有新的初始屏幕、状态栏和时钟的 Gadgets 程序

创建自定义的组件

Gallery 是无限可扩展的。你可以通过创建自定义的组件（例如对话框或新类），将它们添加到新的集合中。对于类来说，自定义的组件由源代码组成，通常是单个的 CPP 实现文件和 H 头文件。如果这个类是从 MFC 对话类(CDialog, CFormView, CPropertyPage, CRecordView 或 CDaoReardView)中派生出来的，那么，该组件还包括用于该对话的资源数据。

应当把你的工作作为一个新组件来保存，理由如下。首先，Gallery 可以做为存储可重用源代码的仓库来使用。当你为一个项目开发一个类，并将其作为一个组件来保存的时候，你不必为了将该类添加到新项目而搜索磁盘文件来查找源代码。创建新组件的第二个原因是通过网络链接的开发者的特殊需要，他们现在可以向中心组件“银行”中添加组件，或从中调用组件。在开发新类之前，你可以检查一下 Gallery，看看是否有人已经为你做了这项工作。

如果想向 Gallery 添加一个组件，请在 Workspace 窗口的窗格中右击它的类，并从上下文菜单中选择 Add To Gallery（添加到组件集）命令。图 7-4 说明了为名为 CNewComponent 的类的添加步骤。

如果必要的话，单击 Add To Gallery 命令将创建一个新的文件夹。该子文件夹具有与项目相同的名字，并且包含新的具有 OGX 扩展名的组件文件。OGX 组件文件不是指向原始类源代码的指针；它包含有定义该类的 CPP 和 H 文本文件的绑

定副本。如果该类是基于对话的，那么 OGX 文件便存档了代码对话资源脚本的一个副本。因为 OGX 文件存档了代码的完整副本，所以，一个组件可以在它的项目消失后依旧存在。即使项目被删除或移动了，Gallery 总是可以产生原始的资源代码。这就意味着，在将组件添加到 Gallery 时代码的快照便是组件。组件不是与源代码一起自动更新的；如果想更新组件，你必须再次向 Gallery 添加该类。

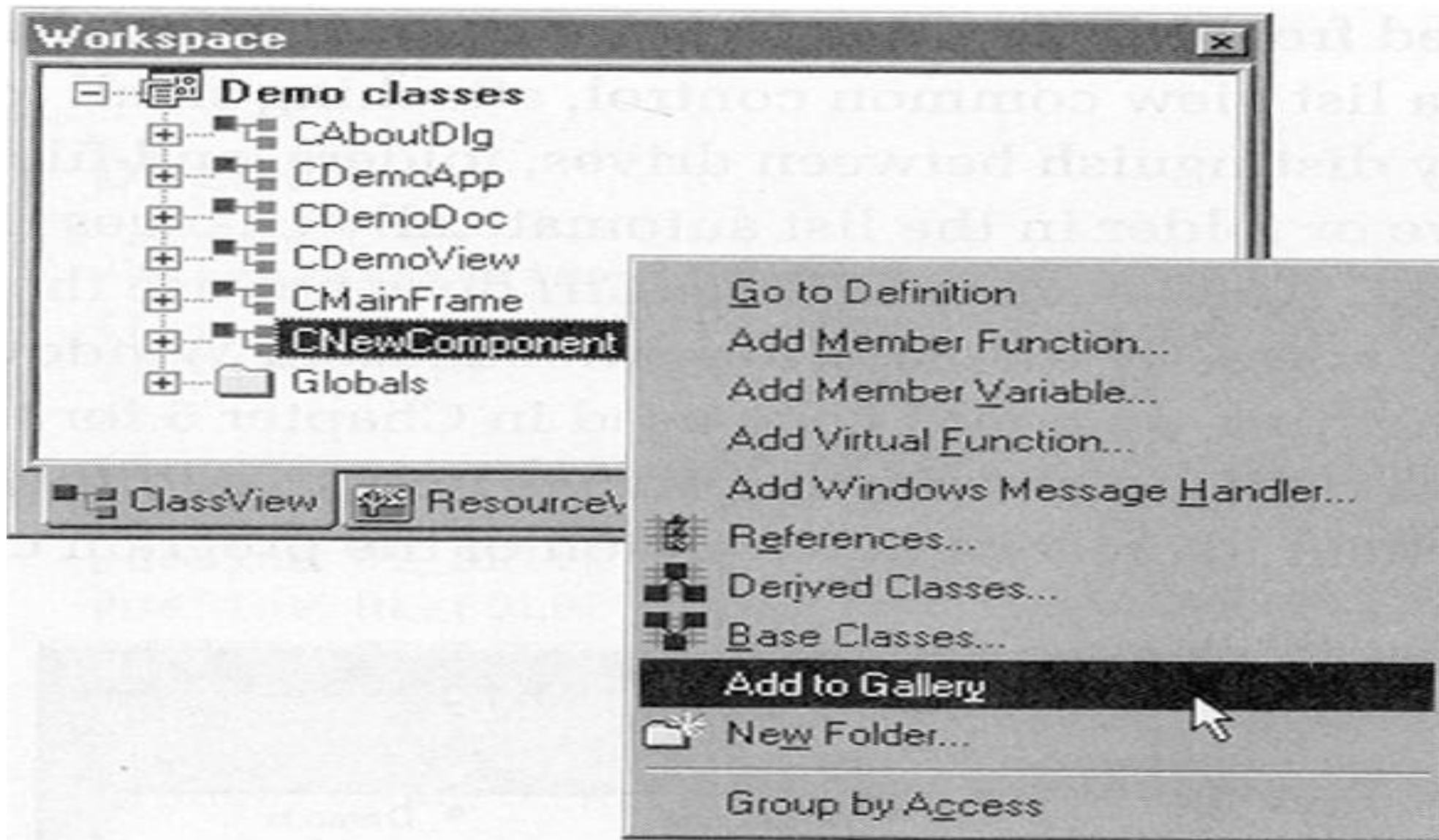


图 7-4 向 Gallery 添加一个类

你所创建的自定义组件有一定的局限。一是对于自定义组件，Gallery 对话框中的 More Info 按钮是非活动的，因为没有直接的方法添加在线帮助，来解释该组件是如何工作的。自定义组件也不能像 Property Sheet 和 Splash Screen 组件所做的那样，自动修改项目中现有的源文件。要创建适合于软件市场的专业质量的组件，需要 Microsoft 的 Component Builder's Kit，利用它，可以创建可使用在线帮助系统的可执行组件。Microsoft 不再为该工具箱收取费用，但目前只能提供给软件公司，不能提供给个人。如想申请 Builder's Kit 的副本，请在你的公司的信头上向如下地址发送一个请求。

Visual C++ Manager

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052-6399

例子：用于目录列表的自定义组件

本节说明创建自定义组件的过程，首先规划出一个名为 CDirListCtrl 的新类，然后将它添加到 Gallery。CDirListCtrl 是从 MFC 的 CListCtrl 类中派生的，它在列表视图通用控件中显示目录列表，并附加上帮助你迅速区分驱动器、文件夹或文

件的小图标（见图 7-5）。单击列表中的驱动器或文件夹将自动改变路径并刷新列表。因为 CDirListCtrl 不使用 DlgDirList API 函数，所以，它可以在 Windows 95 中正确显示长文件名。如果这与我们在第 5 章中为 DirList1 程序所使用的一些东西相像，那么你就对了。下一节将显示为了创建叫做 DirList2 的程序的新版本而添加 DirListCtrl 自定义组件。

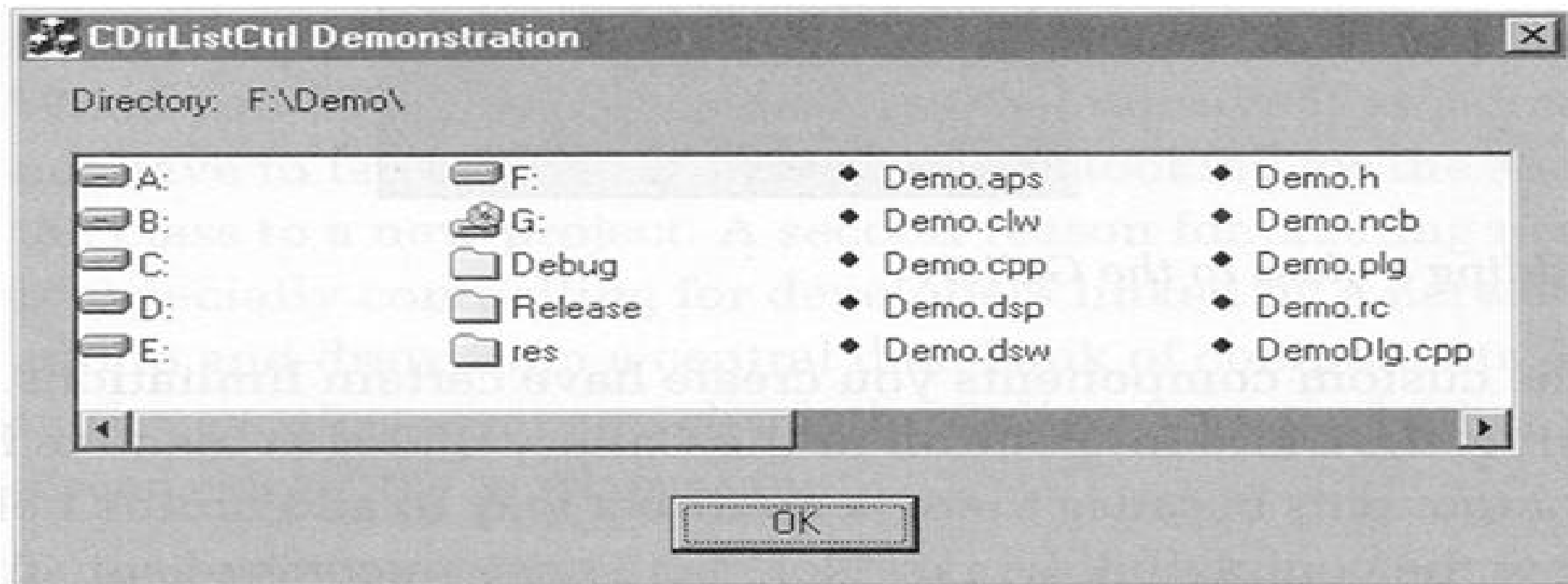


图 7-5 典型对话框中的 CDirListCtrl 列表视图控件

列表 7-1 显示了 CDirListCtrl 类的源代码，包含在本书配套光盘 Code\Chapter.07\DirCtrl 文件夹中的 DirCtrl.h 和 DirCtrl.cpp 文件中。CD.CDirlistCtrl 包含五个公共成员函数，其中有三个在 DirListCtrl 头文件中声明。在其余两个之中，Create 用于建立列表视图控件，ShowList 用于刷新该控件中的目录列表。显示的目录路径存储在私有的 StrDirectory 字符串之中。改变列表是两步过程，在该过程中，CDirListCtrl 对象的创建者首先调用 SetPath 函数来改变目录路径，然后调用

ShowList 来显示新目录。

表 7-1 CDirListCtrl 类的源文件

```
DirCtrl.h

// *****

//

// DirCtrl.h

//

// *****

#define DL_DRIVE          1

#define DL_FOLDER        2

#define DL_FILE           4

#define DL_ALL            7
```

```
typedef      BOOL (CALLBACK *PCALLBACK)( PWIN32_FIND_DATA );
```

```
class CDirListCtrl : public CListCtrl
```

```
{
```

```
private:
```

```
    int                iListFlags, idcPath;
```

```
    CString            strDirectory, strFilter;
```

```
    CListBox            listDummy;
```

```
    CImageList          imageList;
```

```
    CDialog*            pDialog;
```

```
    BOOL                FindFiles( DWORD dwFlags );
```

```
    void                GetCurrentDirectory ();
```

```
    PCALLBACK           pCallBack;
```

public:

```
    BOOL        ShowList( CString& Filter );

    void        GetPath( CString& strPath ) { strPath = strDirectory; }

    void        SetPath( CString& strPath ) { strDirectory = strPath; }

    void        SetCallBack( PCALLBACK pCB ) { pCallBack = pCB; }

    void        Create( CDialog* pDlg, LPRECT prect, int idcControl,

                        int idcStatic, int iFlags=DL_ALL, int idiIcon=0 );
```

protected:

```
    BOOL        GetDirectoryList( int iType );

    afx_msg void OnSelChange( NMHDR* pnumhdr, LRESULT* pResult );

    DECLARE_MESSAGE_MAP ()
```

```
};
```

DirCtrl.cpp

```
// *****
```

```
//
```

```
// DirCtrl.cpp
```

```
//
```

```
// *****
```

```
#include "afxcmn.h"
```

```
#include "dirctrl.h"
```

```
#define IDC_DUMMY          48888
```

```
void CDirListCtrl::Create( CDialog* pDlg, LPRECT prect, int idcControl,  
  
                           int idcStatic, int iFlags, int idiIcon )
```



```
{

    pDialog    = pDlg;                // Save ptr to dialog object

    idcPath    = idcStatic;           // Save ID of static control

    pCallBack = NULL;                // Assume no call-back

    // Create DirListCtrl control

    if (!CreateEx( WS_EX_CLIENTEDGE, WC_LISTVIEW, NULL,

        LVS_LIST | LVS_SINGLESEL | LVS_ALIGNLEFT | WS_VISIBLE | WS_CHILD,

        WS_VISIBLE | WS_CHILD, prect->left, prect->top,

        prect->left, prect->top, prect->right, prect->bottom,

        pDialog->m_hWnd, (HMENU) idcControl ))

    {

        MessageBox( "Failed to create DirListCtrl" );
    }
}
```

```
        return;

    }

    // Associate "file", "folder", and "drive" icons with DirListCtrl control

    imageList.Create( 16, 16, TRUE, 1, 0 );

    if (idiIcon)

    {

        imageList.Add( AfxGetApp()->LoadIcon( idiIcon ) );           // Floppy

        imageList.Add( AfxGetApp()->LoadIcon( idiIcon + 1 ) );       // Hard

        imageList.Add( AfxGetApp()->LoadIcon( idiIcon + 2 ) );       // CD-ROM

        imageList.Add( AfxGetApp()->LoadIcon( idiIcon + 3 ) );       // Folder

        imageList.Add( AfxGetApp()->LoadIcon( idiIcon + 4 ) );       // File

    }
```

```
SetImageList( &imageList, LVSIL_SMALL );

// Create dummy list box

CRect rectDummy;

rectDummy.SetRectEmpty ();

listDummy.Create( WS_CHILD | LBS_SORT, rectDummy, pDialog, IDC_DUMMY );

// Start with current directory

GetCurrentDirectory ();

iListFlags = iFlags;

}

BOOL CDirListCtrl::ShowList( CString& Filter )

{

    static int    iType[3] = { DL_DRIVE, DL_FOLDER, DL_FILE };
}
```

```
BOOL          bRetCode = FALSE;

int           i, j, n, nIcon, nItem = 0;

UINT          uType;

PSTR          pItem      = new char[MAX_PATH];

char          szRoot[] = {"x:\\\\0"};

if (Filter.IsEmpty ())

    strFilter = " *.*";

else

{

    strFilter = Filter;

    strFilter.TrimLeft ();           // Remove any leading

    if (strFilter.GetAt( 0 ) == '\\') // white spaces or
```

```
        {                                // backslash

            strFilter.SetAt( 0, ' ' );

            strFilter.TrimLeft ();

        }

    }

    // Update static control and empty the list

    pDialog->SetDlgItemText( idcPath, strDirectory );

    DeleteAllItems ();

    // Display directory list in this order:  drives, folders, files

    for (n=0, j=DL_DRIVE; n < 3; n++, j <= 1)

    {

        if ((iListFlags & j))
```

```
{

    if (GetDirectoryList( iType[n] ))

    {

        bRetCode = TRUE;

        // Copy list from dummy box to list view control

        i = 0;

        while (listDummy.SendMessage( LB_GETTEXT, i++,

                                        (LPARAM) pItem ) != LB_ERR)

        {

            if (lstrcmp( pItem, "." ) == 0 &&

                lstrcmp( pItem, ".." ))

            {
```

```
// Determine which icon the listed item

// should have:floppy, hard disk, etc.

switch (j)

{

    case DL_DRIVE:

        szRoot[0] = *pItem;

        nIcon = 1;

        uType = ::GetDriveType( szRoot );

        if (uType == DRIVE_REMOVABLE)

            nIcon = 0;

        if (uType == DRIVE_CDROM)

            nIcon = 2;
```

```
        break;

    case DL_FOLDER:

        nIcon = 3;

        break;

    default:

        nIcon = 4;

    }

    InsertItem( nItem, pItem, nIcon );

    SetItemData( nItem++, iType[n] );

}

}

}
```



```
    }

}

delete [] pItem;                // Clean up

return bRetCode;

}

BEGIN_MESSAGE_MAP (CDirListCtrl, CListCtrl)

    ON_NOTIFY_REFLECT (LVN_ITEMCHANGED, OnSelChange)

END_MESSAGE_MAP ()

void CDirListCtrl::OnSelChange( NMHDR* pnumhdr, LRESULT* pResult )

{

    NM_LISTVIEW* pnmlv = (NM_LISTVIEW*) pnumhdr;

    // Get item selected in list view
```

```
if ((pnmlv->uNewState) & LVIS_FOCUSED)

{

    // Is it a drive, folder, or file?

    int iType = GetItemData( pnmlv->iItem );

    if (iType != DL_FILE)                // Ignore it if it's a file

    {

        CString strItem = GetItemText( pnmlv->iItem, 0 );

        // If drive selected, replace directory

        if (iType == DL_DRIVE)

            strDirectory = strItem;        // Directory = "d:"

        // If folder selected, append it to current path

    else
```

```
        strDirectory += strItem;           // Append string "subdir"

strDirectory += "\\";                      // In either case, append '\'

if (!ShowList( strFilter ))                // New directory listing
{

    // If error, reset to current directory

    GetCurrentDirectory ();

    strFilter.Empty ();

    ShowList( strFilter );

}

}

}

*pResult = 0;
```

```
}

BOOL CDirListCtrl::GetDirectoryList( int iType )

{

    DWORD      dwDrives;

    char        szDrive[] = "A:";

    BOOL        bRet;

    // Empty the dummy list box

    listDummy.SendMessage( LB_RESETCONTENT, 0, 0 );

    switch (iType)

    {

        case DL_DRIVE:

            dwDrives = ::GetLogicalDrives ();
```

```
    for (; dwDrives && szDrive[0] <= 'Z'; ++szDrive[0])

    {

        if (dwDrives & 1)

            listDummy.SendMessage( LB_ADDSTRING, 0,

                                    (LPARAM) szDrive );

        dwDrives >>= 1;

    }

    bRet = TRUE;

break;

case DL_FOLDER:

    bRet = FindFiles( FILE_ATTRIBUTE_DIRECTORY );

break;
```

```
        case DL_FILE:

            bRet = FindFiles( 0L );

            break;

    }

    return bRet;

}

BOOL CDirListCtrl::FindFiles( DWORD dwFlags )

{

    WIN32_FIND_DATA fd;

    HANDLE                hFind;

    CString                str = " *.*";

    BOOL                   bOkay;
```

```
if (dwFlags == 0)

    str = strFilter;

hFind = ::FindFirstFile( strDirectory + str, &fd );

if (hFind == INVALID_HANDLE_VALUE)

    return FALSE;


while (TRUE)

{

    if (dwFlags == (fd.dwFileAttributes &

        FILE_ATTRIBUTE_DIRECTORY))

    {

        bOkay = (pCallBack) ? pCallBack( &fd ) : TRUE;
```

```
    if (bOkay)
```

```
        listDummy.SendMessage( LB_ADDSTRING, 0, (LPARAM) fd.cFileName );
```

```
    }
```

```
    if (!::FindNextFile( hFind, &fd ))
```

```
        break;
```

```
}
```

```
::FindClose( hFind );
```

```
return TRUE;
```

```
}
```

```
void CDirListCtrl::GetCurrentDirectory ()
```

```
{
```

```
    PTSTR pDir = new char[MAX_PATH];
```



```
    ::GetCurrentDirectory( MAX_PATH, pDir );

    strDirectory = pDir;

    if (strDirectory.Right( 1 ) != "\\")

        strDirectory += "\\";                // Append backslash

    delete [ ] pDir;

}
```

仅仅为了创建控件而调用一次的 Create 函数，在 DirCtrl 头文件中，其原型为：

```
void Create( CDialog* pDlg, LPRECT prect, int idcControl,

            int idcStatic, int iFlags=DL_ALL, int idiIcon=0 );
```

表 7-1 解释了该函数的六个参数。

调用者可以在它的资源代码中最多提供五个图标图像，每个 16 像素形成一个正方形，它们代表了在列表视图控件中显示的各种驱动器类型和文件名。图 7-5 显示了可能的图标图像，它们包括软盘驱动器、硬盘、CD-ROM 驱动器、文件、文件夹以及代表文件的小钻石形状。Create 函数通过调用 CImageList::Add 函数，

可以将每个图标附属到控件，由 `idIcon` 参数标识以软盘驱动器图标作为开始。因此 `idIcon+1` 代表硬盘图像的标识符；`idIcon+2` 是代表 CD-ROM 驱动器图像的标识符；等等。图标是可选的，也可以不使用它。

表 7-1 `Create` 函数的参数

参 数	说 明
<code>pDlg</code>	用于包含对话框的 <code>this</code> 指针
<code>prect</code>	指向包含在对话窗口内列表视图控制的维数的 <code>RECT</code> 结构
<code>idcControl</code>	列表视图控件的标识符
<code>idcStatic</code>	对话中静态控件的标识符。 <code>ShowList</code> 函数将当前的目录路径写到 <code>idcStatic</code> 控件

续表

iFlags	在 DirCtrl.h 中定义的位标志，由它来确定目录所列出的内容。值可以是 DL_DRIVE, DL_FOLDER 和 DL_FILE 的任意组合，默认的 DL_ALL 值包括列表中所有的三种类型
idIcon	用于在目录列表中显示的五个图标中的第一个标识符值

Create 函数在退出之前调用 CListBox::Create，来建立一个名为 listDummy 的不可见的列表框控件。listDummy 列表框从来都不显示，它仅仅作为组成目录列表的文件名的中间存储仓库。FindFiles 函数在目录中定位的每个文件夹的名字被添加到 listDummy 列表框。既然该列表框是用 LBS_SORT 标志创建的，那么，当它接收到每个文件和文件夹名字的时候，它会自动将它的字符集进行分类。当 ShowList 函数从列表框中展开字符串的时候，该列表框将该按字母顺序一个接一个地传送字符。

SetCallback 内嵌函数用于取得一个可选的回调例程的地址。如果 PCallback 地址为非空，那么，FindFiles 函数就假定回调例程存在，并在将文件名添加到 listDummy 列表框之前，为每个文件调用它：

```
bOkey = (pCallBack) ? pCallBack(&fd):TRUE;
```

```
if(bOkey)
```

```
listDummy.SendMessage(LB_ADDSTRING, 0, (LPARAM))fd.cFileName;
```

通过从回调例程中返回 TRUE 或 FALSE， CDirListCtrl 对象的创建者可以接受和拒绝任何文件。在本章稍后，我们将看到 DirList2 程序是如何利用该项功能来为目录列表中所显示的文件打开窗口的。

创建 CDirListCtrl 类

在作为组件添加 CDirListCtrl 之前，我们需要一个临时项目来包含源文件。这是因为 Workspace 窗口的 ClassView 窗格只能在打开的项目中才能访问。因此，当你想添加到 Gallery 的类作为不属于项目的源代码存在时，创建一个临时类是很有必要的。一旦该类已被添加到 Gallery，你就可以删除这个临时项目了。项目名称无关紧要。

你可以用两种方法创建一个临时项目来包含现有的类。第一种方法依赖于 AppWizard。在 File 菜单上单击 New，在 Projects 选项卡中选择 MFC AppWizard (exe) 图标，键入项目名(下面所概括的步骤假定项目名为 DirCtrl)。单击 AppWizard 中的 Finish 按钮来接受所有的默认设置，这并不重要的原因是，我们将抛弃所有的生成文件。

为包含组件类而创建临时项目的第二种方法不依赖于 AppWizard。这样一个项目仅仅需要一个与项目同名的存根 RC 文件。甚至这个 RC 文件可以是空的。如果想创建这个项目，请在 New 对话中的 Projects 选项卡中选中 Win32 Application 图标(而不是 AppWizard 图标)，给项目取一个名字(例如 DirCtrl)，然后单击 OK。在该向导的下一步，接受指定空项目的默认设置，单击 Finish 按钮。下一步，使用文本编辑器来创建一个空的 RC 文件，并将其保存到 DirCtrl 项目文件夹。在新项目处于打开状态的时候，从 Project 菜单选择 Add To Project 命令，并单击次级菜单上的 Files。在 Insert Files Into Project 对话框中双击存根 RC 文件，将其添加到项目，这就是启用 ClassWizard 所需要做的所有工作，现在可以通过 View 菜单上的 ClassWizard 命令访问 ClassWizard 了。当调用 ClassWizard 的时候，Visual C++ 将检测到用于该项目的 CLW 文件不存在，并提供它以创建一个新文件。你必须单击 Yes，并在随后出现的 Select Source Files (选择源文件) 对话框中单击 OK 来继续 MFC ClassWizard 对话。

无论使用哪一种方法来创建新项目，都要单击 MFC ClassWizard 对话中的 Add Class (添加类) 按钮，然后选择 New，来调用如图 6-4 所示的 New Class 的对话。键入 CDirListCtrl 作为类名，选择 CListCtrl 作为基类。单击 Change 按钮，将所生成的源文件重命名为 DirCtrl.h 和 DirCtrl.CPP。当你关闭 ClassWizard 对话的时候，Visual C++ 将编写用于新 CDirListCtrl 类的存根代码，并且将文件自动添加到项目中。

第二步是为新类提供源代码。将 DirCtrl.cpp 和 DirCtrl.h 文件从本书配套光盘的

Code\Chapter.07\Dirctrl 文件夹复制到 DirCtrl 项目文件夹，覆盖掉 ClassWizard 刚刚创建的存根文件。还将文件 DirCtrl.rco, Floppy.ico, HardDisk.ico, CD-ROM.ico, Folder.ico 和 File.ico 复制到项目文件夹，以后将需要这些文件。

创建 CDirListCtrl 组件

CDirListCtrl 目前是一个正在开发的类，还不是已完成的和可以使用的组件。展现项目的 ClassView 窗格，并单击类列表中的 CDirListCtrl，然后从如图 7-4 所示的上下文菜单中选择 Add To Gallery 命令。Visual C++ 把 DirCtrl.cpp 和 DirCtrl.h 组成一个名为 DirListCtrl.ogx 的单一文件，并将这个 OGX 文件存储在名为 DirCtrl 的新 Gallery 文件夹中。下面几段将详细介绍这个新的 Gallery 文件夹，因此，不要将它与原来的 DirCtrl 项目文件夹弄混了。

打开 Gallery 中的 DirCtrl 文件夹看看这个新组件，它是以大图标视图的形式出现的。



它需要做一些额外的工作，但你可以将它这个组件的外观再装饰一下。第一步是将 O G X 文件移出 DirCtrl 文件夹，并且用快捷方式文件来代替它。右击新的 DirListCtrl.ogx 图标，并从上下文菜单中选择 Cut 命令。通过右击大列表框的任何空白区域，并选择 New 命令，在 Gallery 对话的文件夹中创建一个新的子文件夹，为新文件夹取一个像 O G X Files 这样的一般性的名字，然后，将 DirListCtrl.ogx 文件粘贴进这个新文件夹。右击 DirListCtrl.ogx 图标，从上下文菜单中选择 Create Shortcut（创建快捷方式）命令，然后剪切该快捷方式文件，并将其粘贴回原来的 DirCtrl 文件夹。

我们就要完成了。这时候，原来的 O G X 文件已被移到了一个叫做 O G X Files 的文件夹中，并且为 DirCtrl 文件夹中的一个快捷方式所取代。最后一步是改进 DirCtrl 文件夹中快捷方式图标的外观。通过右击该图标，并选择 Rename，为快捷方式图标提供一个更具说明性的名字(例如 Directory List)。右击 Directory List

图标，选择 Properties,然后单击 Properties 对话 Shortcut 选项卡中的 Change Icon（更改图标）按钮。输入（或浏览）到你以前曾经从配套光盘中复制的 DirCtrl.ico 文件的路径。当你单击 OK 关闭 Properties 对话的时候，组件现在在 DirCtrl 文件中就应该像这个样子。



我们只做了很少的工作，就使组件的外观比我们开始的时候要**好**。如果你肯付出努力，还可以把所有的新组件文件都放入 OGX Files 文件夹，按照相同的方法用快捷方式来代替它们。

不幸的是，Visual C++没有提供简单的方法来将说明附加到类组件上，在 OGX 文件和它的快捷方式中均不可以。只能用文件名来向其他开发者提供有关组件目的的**必要信息**。创建一个可以描述自己的组件，需要前面提到的 Component Builder's Kit。图 7-1 显示了当 Dialog 栏组件被选中的时候，第一条说明是如何出现在 Gallery 之中的。尽管已超出了本章的范围,但使用 Builder's Kit 创建 Directory

List 组件, 允许我们在 DIR 图标在对话中被选中的时候, 附加上一条类似的说明:

Display a sorted directory listing in a list view control.

Complete with icons representing drives, folders, and files.

退出 Gallery 之后, 你可以通过在 Gallery 对话中显示 DirCtrl 文件夹, 并双击代表 Directory List 组件的快捷方式, 来将 CDirListCtrl 类添加向任何打开的项目。既然组件已被安装到 Gallery 之中, DirCtrl 项目就已经实现了它的目的, 不再需要它了。DirListCtrl.ogx 包含有类源代码, 因此你可以安全地关闭这个项目, 并且删除原来的 DirCtrl.cpp 和 DirCtrl.h 文件, 以及 DirCtrl 项目文件夹中的其他文件。不过, 不要删除以前从光盘中复制的 ICO 文件。Directory List 快捷方式链接指向包含有代表组件的 DIR 图标的 DirCtrl.ico 文件。当你将新 CDirListCtrl 类插入项目的时候, 它可能需要部分或全部保留文件, 例如 HardDisk.ico 和 Folder.ico。下节将用一个名为 DirList2 的范例程序来说明 Directory List 组件。

例子:DirList2程序

你可能会回想起第 5 章中介绍的 DirList1 程序, 它使用了 DlgDirList API 函数来在列表框控件中显示目录列表。该列表不仅在外观上很平淡, 而且难以阅读, 因为 DlgDirList 不能在 Windows 95 中识别长文件名, 所以, 它只能在 Windows NT 中显示长文件名。这里所展示的 DirList2 程序通过融入新的 DireCtory List 组件

克服了这些不足之处。

如图 7-6 所示的 Dirlist2 程序与 Dirlist1 一样，是基于对话的，它使用属性表与用户进行交互。除了 Directory List 控件所提供的吸引人的目录列表之外，DirList2 还利用了属性表的 Date 和 Size 选项卡，允许用户按照文件大小或日期对列表进行过滤。例如，程序可以只显示在最后一日中创建容量为 5 到 10KB 之间的文件。使用前面所介绍的 CDirListCtrl 回调功能可以实现其他过滤。

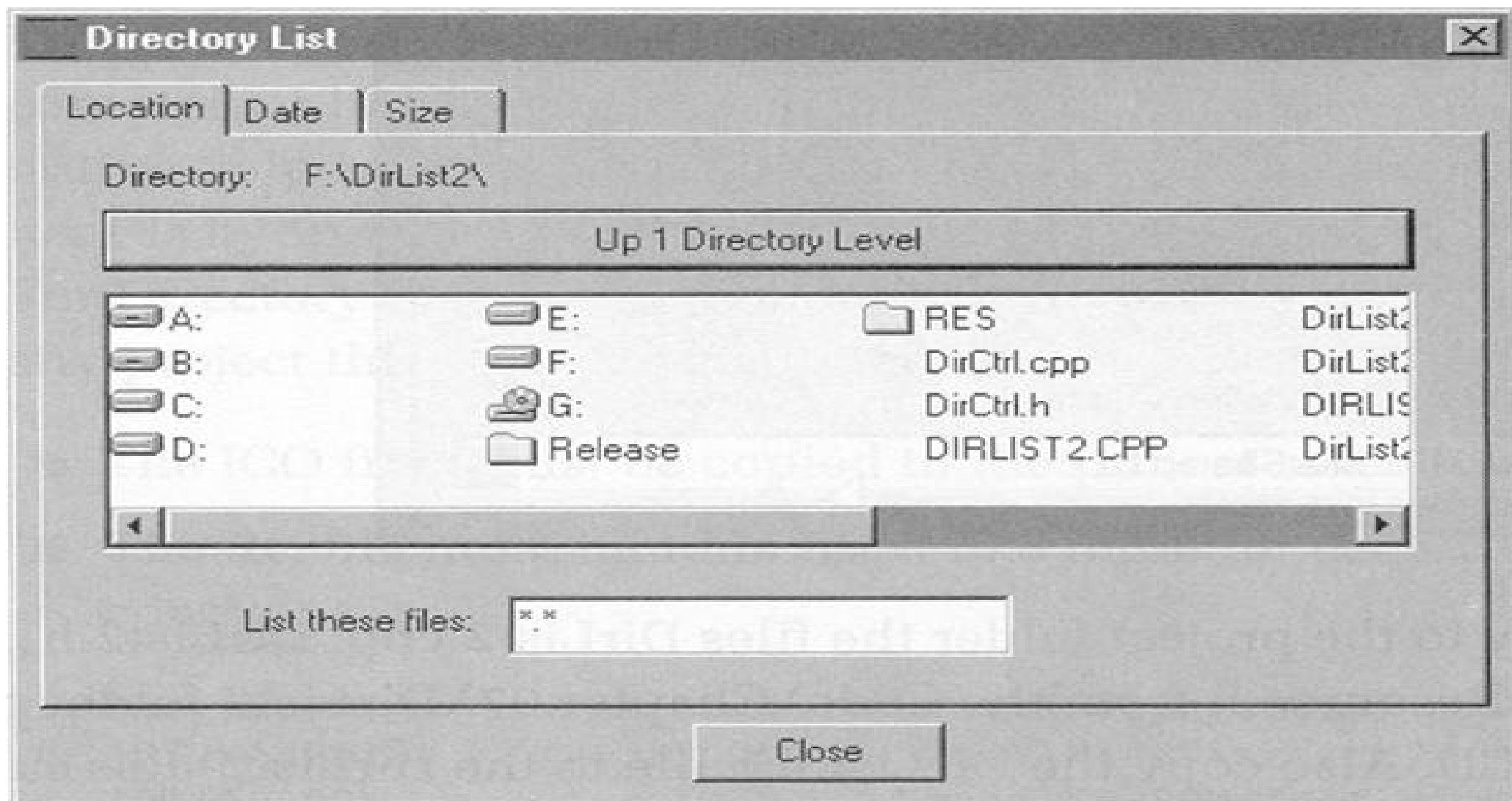
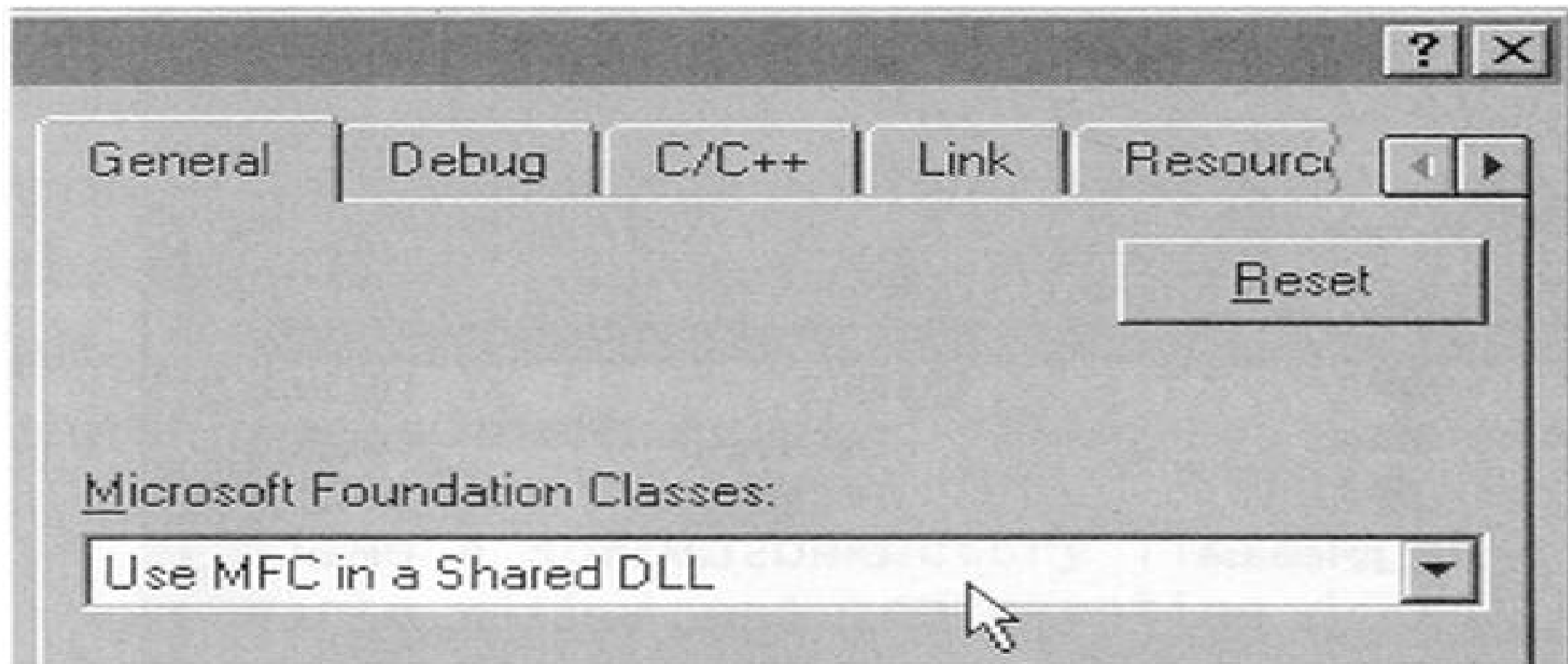


图 7-6 DirList2 程序

为使我们添加的内容比较少，DirList2 程序使用了与 DirList1 程序相同的 Resource.h 文件和相同的 RC 文件。这些文件的内容在第 5 章中已经列出。如果你愿意自己独立创建 DirList2 程序，并且不执行 Setup 程序来从配套光盘上安装范例项目；请单击 File 菜单上的 New，并选中 Projects 选项卡。因为 DirList2 不是 AppWizard 程序，所以，请单击 Win32 Application 来创建该项目。在该向导的对话出现时，输入项目名，并接受默认的设置。在 Visual C++ 创建了空项目之后，单击 Project 菜单上的 Settings。在 Project Settings 对话的 General 选项卡中，选择选项，Use MFC In A Shared DLL（在共享的 DLL 中使用 MFC）。



将文件 `DirList2.cpp`, `DirList2.h`, `DirList2.rc` 和 `Resource.h` 从本书配套光盘的 `Code\Chapter.07\DirList2` 文件夹复制到项目文件夹。还要把 `DirList.ico` 文件复制到 `DirList2\Res` 子文件夹, 以便 `DirList2` 有一个应用程序图标。通过从 `Project` 菜单选择 `Add To Project`, 然后从次级菜单中选择 `Files` 命令, 来将 `DirList2.cpp` 和 `DirList2.rc` 文件附加到项目上。

因为 DirList2 程序利用了新的 CDirListCtrl 类，所以还需要几个文件。当然，这是练习的整个目的：既然类已经安装在 Gallery 之中了，那么，添加用于 CDirListCtrl 的源文件就是一件很容易的工作了。通过再次从 Project 菜单中选择 Add To Project，并且单击 Components And Controls，可以打开 Gallery 对话。在 Gallery 的 DirCtrl 文件夹中，双击代表新 Directory List 组件的图标。源文件 DirCtrl.cpp 和 Dirctrl.h 自动从 DirListCtrl.ogx 组件文件中展开并添加到项目中。

不幸的是，Gallery 不能够提供 CDirListCtrl 在目录列表中显示图标所需要的 ICO 文件。除了通过使用 Microsoft 的 Bulder's Kit 创建该组件之外，没有办法将 ICO 文件与 OGX 文件中的源代码捆绑起来，因此，必须在一步中将所有必要的文件传送给项目。组件的用户必须手工复制 ICO 文件，DirList2 只需要驱动器和文件夹图标，它们包含在位于以前创建的 DirCtrl 项目文件夹的 Floppy.ico, HardDisk.ico, CD-ROM.ico 和 Folder.ico 文件之中。请将这四个文件复制到 DirList2\Res 文件夹。

这是额外的步骤，它必须为 Directory List 这样的自定义组件进行存档。既然自定义组件不能在 Gallery 对话中提供在线帮助，那么，组件提供文档的最好办法就是通过在 CPP 文件的开头包括一块注释。对于 Directory List，注释应该满足使用该组件的任何项目的四个需要：

- ICO 文件必须被复制到项目的 Res 的文件夹。
- RC 文件必须包括在控件中用于每个图标的链接。

IDI_FLOPPY	ICON	"res\\Floppy.ico"
IDI_HARDDISK	ICON	"res\\HardDisk.ico"
IDI_CD_ROM	ICON	"res\\CD-ROM.ico"
IDI_FOLDER	ICON	"res\\Folder.ico"
IDI_FILE	ICON	"res\\File"

- 标识符 IDI_FLOPPY,IDI_HARDDISK,IDI_CD-ROM,IDI_FOLDER 和 IDI_FILE 必须在项目的 Resource.h 文件中定义。IDI_FLOPPY 可以是任何值；其余的标识符值必须依给定的顺序依次增加。
- 所有使用类的源文件必须包含如下一行。

```
# include "dirctrl.h"
```

DirList2 的操作基本上与 DirList1 相同，除了 CPage1 类包括了一个叫作 CheckDateSize 的新成员函数之外，CheckDateSize 是一个通过调用 CDirListCtrl::SetCallBack 进行注册的回调函数。正如在源代码的注解中所说明的那样，CDirListCtrl::findFiles 为它想添加到目录列表的每个文件名调用这个回调函数，为该回调函数分配一个指向 WIN32_FIND_FILE 结构的指针，该结构包含

有与该文件有关的信息。CheckDateSize 用于决定文件确认用户在 Size 和 Date 页中所设置的过滤器，并返回一个值 TRUE 或 FALSE，允许或不允许该文件。

修改后的源代码列在下面。没有包括 DirList.rc 和 Resoure.h 文件，原因是它们与第 5 章中的只有细微的差别，只是融入了代表图标资源 IDI_FLOPPY, IDI_HARDISK, IDI_CD-ROM 和 IDI_FOLDER 的几行，因为程序在没有目标标记的列表中留下了文件名，所以没有定义 IDI_FILE。你可以在本书配套光盘的 Code\Chapter.07\DirList2 子文件夹中找到所有的文件。

列表 7-2 用于 DirList1 程序的源文件

DirList2.h

```
// *****  
  
//  
  
// DirList2.h  
  
//  
  
// *****
```



```
class CDirListApp : public CWinApp
{

public:

    BOOL InitInstance ();

};
```

```
class CAboutDlg : public CDialog
{

public:

    CAboutDlg();

};
```

```
class CPage2;                // Forward reference
```

```
class CPage3;
```

```
////////////////////////////////////
```

```
// CPage1 property page
```

```
class CPage1 : public CPropertyPage
```

```
{
```

```
private:
```

```
    BOOL                bEditChange;
```

```
    CDirListCtrl        dirlist;
```

```
    CString             strFilter, strOldFilter;
```

```
static CPage2*      pDate;
```

```
static CPage3*      pSize;
```

```
static BOOL CALLBACK    CheckDateSize( PWIN32_FIND_DATA pfd );
```

```
public:
```

```
CPage1 () : CPropertyPage( IDD_PAGE1 ) {}
```

```
protected:
```

```
virtual void          DoDataExchange( CDataExchange* pDX );
```

```
virtual BOOL          OnInitDialog ();
```

```
virtual BOOL          OnSetActive ();
```

```
afx_msg void      OnUp1Level ();

afx_msg void      OnEditGainFocus ();

afx_msg void      OnEditChanging ();

afx_msg void      OnEditLoseFocus ();

DECLARE_MESSAGE_MAP ()

};

////////////////////////////////////

// CPage2 property page

class CPage2 : public CPropertyPage

{
```

```
public:
```

```
    int          nAnyDate, PrevDays, PrevMonths;
```

```
    Ctime        timeMin, timeMax;
```

```
    CPage2 () : CPropertyPage( IDD_PAGE2 ) {}
```

```
protected:
```

```
    virtual void DoDataExchange( CDataExchange* pDX );
```

```
    virtual BOOL OnInitDialog ();
```

```
};
```

```
////////////////////////////////////
```

```
// CPage3 property page
```

```
class CPage3 : public CPropertyPage
```

```
{
```

```
public:
```

```
    int          nAnySize;
```

```
    DWORD       MinSize, MaxSize;
```

```
    CPage3 () : CPropertyPage( IDD_PAGE3 ) {}
```

```
protected:
```

```
    virtual void DoDataExchange( CDataExchange* pDX );
```

```
virtual BOOL    OnInitDialog ();

};

////////////////////////////////////

// CListSheet

class CListSheet : public CPropertySheet
{

public:

    CPage1    page1;

    CPage2    page2;

    CPage3    page3;
```

```
CListSheet( LPCTSTR szCaption );
```

protected:

```
virtual BOOL OnInitDialog();
```

```
afx_msg void OnSysCommand( UINT nID, LPARAM lParam );
```

```
DECLARE_MESSAGE_MAP()
```

```
};
```

DisList2.cpp

```
// *****
```

```
//
```

```
// DirList2.cpp
```



```
//
```

```
// *****
```

```
#include "afxwin.h"
```

```
#include "afxdlgs.h"
```

```
#include "afxcmn.h"
```

```
#include "resource.h"
```

```
#include "dirctrl.h"
```

```
#include "dirlist2.h"
```

```
CDirListApp DirListApp;
```

```
BOOL CDirListApp::InitInstance ()  
  
{  
  
    CListSheet sh( "Directory List" );           // Create property sheet object  
  
    sh.DoModal ();                               // and display dialog  
  
  
    return FALSE;                               // Return FALSE to exit DirList2  
  
}
```

```
CAboutDlg::CAboutDlg() : CDialog( IDD_ABOUTBOX )  
  
{  
  
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// CListSheet property sheet
```

```
CListSheet::CListSheet( LPCTSTR szCaption ) :
```

```
    CPropertySheet( szCaption )
```

```
{
```

```
    AddPage( &page1 );
```

```
    AddPage( &page2 );
```

```
    AddPage( &page3 );
```

```
}
```

```
BEGIN_MESSAGE_MAP (CListSheet, CPropertySheet)
```

```
ON_WM_SYSCOMMAND()
```

```
END_MESSAGE_MAP()
```

```
BOOL CListSheet::OnInitDialog()
```

```
{
```

```
    CButton* button;
```

```
    CPropertySheet::OnInitDialog();
```

```
    // Add "About..." menu item to system menu
```

```
    CMenu*    pSysMenu = GetSystemMenu( FALSE );
```

```
    CString str;
```

```
str.LoadString( IDD_ABOUTBOX );

pSysMenu->AppendMenu( MF_SEPARATOR );

pSysMenu->AppendMenu( MF_STRING, IDD_ABOUTBOX, str );


// Remove Apply and Cancel buttons and rename OK button

button = (CButton *) GetDlgItem( ID_APPLY_NOW );

button->DestroyWindow();

button = (CButton *) GetDlgItem( IDCANCEL );

button->DestroyWindow();

button = (CButton *) GetDlgItem( IDOK );

button->SetWindowText( "Close" );
```

```
// Set the 16-by-16 image (see closing remarks in Chapter 5)

HICON hIcon = (HICON) ::LoadImage( DirListApp.m_hInstance,

                                   MAKEINTRESOURCE( IDI_APPICON ),

                                   IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR );

SetIcon( hIcon, FALSE );


// Activate OnInitDialog for each page to init member variables

SetActivePage( 2 );

SetActivePage( 1 );

SetActivePage( 0 );


return TRUE;
```

```
}
```

```
void CListSheet::OnSysCommand( UINT nID, LPARAM lParam )
```

```
{
```

```
    if (nID == IDD_ABOUTBOX)
```

```
    {
```

```
        CAboutDlg dlgAbout;
```

```
        dlgAbout.DoModal();
```

```
    }
```

```
    else
```

```
    {
```

```
        CPropertySheet::OnSysCommand( nID, lParam );
```

```
    }  
  
}  
  
////////////////////////////////////  
  
// CPage1 property page  
  
CPage2* CPage1::pDate;           // Static ptrs to Date and  
  
CPage3* CPage1::pSize;           //      Size property pages  
  
  
  
BOOL CPage1::OnInitDialog ()  
{  
  
    RECT rect = { 15, 60, 390, 102 };      // Dimensions of DirCtrl ctrl
```



```
strFilter    = " *.*";
```

```
bEditChange = FALSE;
```

```
pDate       = (CPage2*) ((CListSheet*) GetParent())->GetPage( 1 );
```

```
pSize       = (CPage3*) ((CListSheet*) GetParent())->GetPage( 2 );
```

```
dirlist.Create( this, &rect, IDC_DIRCTRL, IDC_DIRPATH, DL_ALL, IDI_FLOPPY );
```

```
dirlist.SetCallBack( &CheckDateSize );
```

```
return CDialog::OnInitDialog ();
```

```
}
```

```
BOOL CPage1::OnSetActive ()
```

```
{  
  
    dirlist.ShowList( strFilter );  
  
    return CPropertyPage::OnSetActive ();  
  
}
```

```
BEGIN_MESSAGE_MAP ( CPage1, CPropertyPage )  
  
    ON_EN_SETFOCUS      ( IDC_EDIT1, OnEditGainFocus )  
  
    ON_EN_CHANGE        ( IDC_EDIT1, OnEditChanging )  
  
    ON_EN_KILLFOCUS     ( IDC_EDIT1, OnEditLoseFocus )  
  
    ON_BN_CLICKED       ( IDC_BUTTON1, OnUp1Level )  
  
END_MESSAGE_MAP ()
```

```
// These three functions simply ensure that when the user types in another  
  
// filter string (such as "*.txt"), the directory listing is automatically  
  
// updated to reflect the change.  
  
// -----
```

```
void CPage1::OnEditGainFocus ()  
  
{  
  
    GetDlgItemText( IDC_EDIT1, strOldFilter );  
  
}
```

```
void CPage1::OnEditChanging ()  
  
{
```

```
        bEditChange = TRUE;

}

void CPage1::OnEditLoseFocus ()

{

    if (bEditChange)

    {

        bEditChange = FALSE;

        UpdateData( TRUE );

        // If user enters a new filter string, update the directory list

        if (strFilter != strOldFilter)
```

```
    {  
  
        dirlist.ShowList( strFilter );  
  
        strOldFilter = strFilter;  
  
    }  
  
}  
  
}
```

```
void CPage1::OnUp1Level ()
```

```
{
```

```
    CString strPath;
```

```
    // If Enter pressed in edit control, refresh list
```

```
if (GetFocus() == GetDlgItem( IDC_EDIT1 ))

    OnEditLoseFocus ();


// Else go up one directory level

else

{

    dirlist.GetPath( strPath );


    // When strPath == "d:\", we're already at root

    if (strPath.Right( 2 ) != ":\\")

    {

        // Remove '\' at end of string
```

```
strPath.GetBufferSetLength( strPath.GetLength() - 1 );
```

```
strPath.ReleaseBuffer ();
```

```
// Find last '\' and truncate strPath string
```

```
int cLastSlash = strPath.ReverseFind( '\\' );
```

```
if (cLastSlash != -1)
```

```
{
```

```
    strPath.GetBufferSetLength( cLastSlash + 1 );
```

```
    strPath.ReleaseBuffer ();
```

```
    dirlist.SetPath( strPath );
```

```
    dirlist.ShowList( strFilter );
```

```
}
```

```
    }

}

}

// Each time it finds a file that is a candidate for inclusion in the
// directory list, CDirListCtrl::FindFiles calls this static callback
// -----

BOOL CPage1::CheckDateSize( PWIN32_FIND_DATA pfd )
{

    // Accept all subdirectories regardless of date/size criteria

    if (pfd->dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
```



```
    return TRUE;

// If options set in Date page, filter files by date criteria

if (pDate->nAnyDate)

{

    CTime    timeFile( pfd->ftLastWriteTime );

    if (pDate->nAnyDate == 1)

    {

        // Reject file not dated within specified min/max period

        if (timeFile <= pDate->timeMin    || timeFile > pDate->timeMax)

            return FALSE;
    }
}
```

```
}

else

{

    CTime      timeNow = CTime::GetCurrentTime();

    CTimeSpan timeAge( timeNow.GetTime() - timeFile.GetTime() );


    // Reject file older than specified number of months

    if (pDate->nAnyDate == 2    &&    pDate->PrevMonths)

        if (timeAge.GetDays()/30 >= pDate->PrevMonths)

            return FALSE;


    // Reject file older than specified number of days
```

```
        if (pDate->nAnyDate == 3    &&    pDate->PrevDays)

            if (timeAge.GetDays() >= pDate->PrevDays)

                return FALSE;

    }

}

// If options set in Size page, filter files by size criteria

if (pSize->nAnySize)

{

    if (pSize->MinSize    &&    pfd->nFileSizeLow < pSize->MinSize*1024)

        return FALSE;

}
```

```
    if (pSize->MaxSize  &&  pfd->nFileSizeLow > pSize->MaxSize*1024)
```

```
        return FALSE;
```

```
    }
```

```
    return TRUE;
```

```
}
```

```
void CPage1::DoDataExchange( CDataExchange* pDX )
```

```
{
```

```
    CPropertyPage::DoDataExchange( pDX );
```

```
    DDX_Text( pDX, IDC_EDIT1, strFilter );
```

```
    DDV_MaxChars( pDX, strFilter, 128 );
```

```
}
```

```
////////////////////////////////////////////////////////////////
```

```
// CPage2 property page
```

```
BOOL CPage2::OnInitDialog ()
```

```
{
```

```
    CSpinButtonCtrl* spin;
```

```
    // Initialize variables
```

```
    nAnyDate      = 0;
```

```
    PrevDays      = 1;
```

```
PrevMonths    = 1;

timeMin        = 0;

timeMax        = CTime::GetCurrentTime();


// Set limits of spin buttons

spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN1 );

spin->SetRange( 1, 100 );                // Within x months

spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN2 );

spin->SetRange( 1, 365 );                // Within x days


return CDialog::OnInitDialog ();

}
```



```
// CPage3 property page
```

```
BOOL CPage3::OnInitDialog ()
```

```
{
```

```
    CSpinButtonCtrl* spin;
```

```
    // Initialize variables
```

```
    nAnySize = 0;
```

```
    MinSize   = 0;
```

```
    MaxSize   = 100;
```

```
    // Set limits of spin buttons
```



```
spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN1 );

spin->SetRange( 0, 9999 );                                // Min size

spin = (CSpinButtonCtrl *) GetDlgItem( IDC_SPIN2 );

spin->SetRange( 1, 9999 );                                // Max size


return CDialog::OnInitDialog ();

}
```

```
void CPage3::DoDataExchange( CDataExchange* pDX )

{

    CPropertyPage::DoDataExchange( pDX );

    DDX_Radio( pDX, IDC_RADIO1, nAnySize );
```

```
DDX_Text(  pDX, IDC_EDIT1, MinSize );
```

```
DDX_Text(  pDX, IDC_EDIT2, MaxSize );
```

```
}
```

第四部分 ActiveX 控件

第 8 章 使用 ActiveX 控件

ActiveX 控件是可执行组件，设计它们的目的是将其嵌入窗口或 Web 页中，来实现一些完备的功能。对于用户来讲，它们与我们在前一章中遇到的普通的 Windows 控件十分相像，这些普通控件是通过对话框编辑器或 Gallery 添加到程序的。但与普通控件不同的是，ActiveX 控件在 Web 页上或在对话框中是相同的，它使开发者可以同时接触两个明显不同的市场。

如果你对 ActiveX 控件感兴趣(作为一个开发者，你理应如此)，那么，你可能就是因为想使用它们或想编写它们。Visual C++可以帮助你实现这两个目的。本章所涉及的是这一主题的前半部分，介绍如何在叫做包容器的客户应用程序中使用 ActiveX 控件。第 9 章和第 10 章将涉及后半部分，介绍编写 ActiveX 控件的两种不同方法。本章将展现有关 ActiveX 控件的介绍性信息，因此，如果你愿意了解有关这一主题的入门知识的话，应该首先阅读这一章。

为了使讨论的长度适中，这些章中的范例程序使用了 MFC。MFC 框架管理了几百个 ActiveX 编程的详细资料，它使得编写包容器或 ActiveX 控件不再比 Windows 中其他编程项目更为困难。编写使用现有的 ActiveX 控件的 MFC 包容器应用程序通常需要很少的或者根本就不需要什么基础知识。尽管在编写

ActiveX 控件时使用 MFC 有一些争论(完全是因为大小的问题，我们将在下一章讨论)，但是，当应用到包容器的时候，这些争论就不那么有效了。MFC 将客户/服务器之间的交互过程包装得非常完全，以致于如果没有 MFC 类库或类似支持的帮助，就难以编写一个包容器应用程序。不过，如果你不愿意使用 MFC，Active Template Library(ATL)提供了另一种方案。Visual C++ 包括一个叫做 AtlCon 的范例项目，它演示了如何使用 ATL 编写包容器应用程序。源文件位于文件夹 MSDN\Samples\Vc98\ATL\AtlCon 之中。第 10 章为包容器应用程序的开发仔细讨论了 ATL 所提供的类型支持。

尽管本章和随后两章所讨论的是 ActiveX 控件的需求和内部操作，对于这样一个大主题，它们所介绍的也仅仅是基础知识。本章重点介绍一些方法，应用这些方法，Visual C++ 将使程序员在处理 ActiveX 控件时更加得心应手，无论你正在编写的是一个控件，还是使用该控件的包容器应用程序。如果想了解本领域内可能对 Internet 编程来说十分重要的内容，请参考专业资料，例如 Kraig Brockschmidt 的 Inside OLE（OLE 内幕）第二版。你可以在 MSDN 在线帮助中找到完整的文本。

背景知识简介

ActiveX 这个名字是新的，但是，它的技术是成熟的。ActiveX 控件形成了 Microsoft 的 ActiveX 技术的唯一组成部分，它们是建立在 Component Object Model(组件对

象模型, COM)和 OLE 的基础之上。OLE 用于代表 Object Linking and Embedding (对象链接与嵌入), 但是, 因为对象嵌入很早就是 OLE 能力中的很小的一部分, 所以 Microsoft 已经抛弃了这个缩略语。目前, OLE 已经有了新的含义, 不再具有版本号了。它已经从为特定目的而创建的一门技术, 转化到了一种作为其他技术(其中包括 ActiveX)基础的一般结构。OLE 定义了为创建和连接包括叫做 OLE 自定义控件在内的多种程序组件定义了一个标准的蓝图。至少, 它们过去被称作 OLE 控件, Microsoft 现在叫它们为 ActiveX 控件。

那么, 什么是 OLE/ActiveX 控件呢?简单的答案是 OLE/ActiveX 控件是一个动态链接库, 它是作为基于 COM 的服务器进行操作的, 并且可以嵌入在包容器宿主应用程序之中。详细的答案就是本章的内容。在深入介绍它的工作方式之前, 让我们来回顾一下 OLE 控件的历史。

吸引 Windows 开发者注意的第一种类型的组件软件可能是 Visual Basic Extension 模型的自定义控件。自定义控件就是人们所熟悉的 VBX, 它是为添加到控件名之后的三字母扩展名而命名的。VBX 结构允许开发者向 Visual Basic 程序创建有效的和可重使用的附加物, 它们可以作为自己组件放置在窗口之中, 在 Visual Basic 中叫做表单。VBX 控件的优点有三方面:

- VBX 能够实现虚拟显示并与用户交互。
- Visual Basic 应用程序能够通过叫做方法的由 VBX 导出的函数来编程 VBX。

- 作为一个动态链接库，VBX 控件可以以二进制的形式代替源代码重新使用。

正如我们将要看到的，ActiveX 控件可以提供相同的优点。

VBX 控件还允许程序员对从 Visual Basic 中继承下来的一些局限性进行补偿。例如，既然 VBX 控件通常是用 C 或汇编语言编写的，那么，它们就可以使用 Visual Basic 中没有的指针来帮助应用程序进行大量的指针操作，例如散列和分类。VBX 模型的一个问题是，它不是为向其他语言 and 平台完美地进行移植而设计的。例如，一个 C++ 程序员不能很容易地创建一个 VBX 派生物，因为 VBX 不是由类表示的。更进一步来说，VBX 模型是一个捆绑在 Intel 处理器分段结构之上的 16 位标准。不过，VBX 控件活跃的市场已经证明了，组件软件可以在 Windows 开发中充当很重要的角色。

OLE 控件标准是设计用来满足下一层次的要求的，它将 VBX 类型控件的优点带给了具有 Win32 编程功能的所有语言。这些语言包括 Access Basic; Visual Basic for Applications(VBA), WordBasic 和 Visual Basic Scripting(VBScript)。

考虑到 VBX 取名的方法，OLE 控件通常被叫作 OCX，这是因为 OCX 扩展名称通常添加到文件名的后面。有了 OCX 和 VBX 共有的约定，就足以说明其中一个是如何从另一种中演化的。例如，Microsoft 从 VBX 术语中借用了三个接口类型，它们定义了 OLE 控件和它的客户、包容器应用程序之间的通信。

- **Methods (方法)**：OLE 控件展现给包容器应用程序的函数，允许客户调用该控件。
- **Properties(属性)**：控件和包容器内部的公共数据，它可以用来向对方描述自己。在启动期间，一个控件可以读取该包容器的属性，并且调整它的初始化过程，以便它与包容器的外观和特性相配。当控件处于活动状态的时候，包容器可以读取该控件的属性，来了解它当前的状态，如果控件允许的话，还可以重写属性，以改变控件的行为。
- **Events(事件)**：控件发送给包容器的通知，本章稍后将详细介绍这个问题。一个事件通知是通过调用包容器中的一个函数来发生的，也就是众所周知的“点燃”事件。

就在人们已经注意到 16 位 VBX 的局限的时候，OLE(那些日子是叫 OLE2)已逐渐趋于成熟，从逻辑上来说，它已成为以 OLE 控件形式存在的 VBX 的接任者，现在叫做 ActiveX 控件。

控件包容器

ActiveX 控件是服务器，包容器应用程序是客户。ActiveX 控件最好从客户端实现，因此本节一开始是讨论包容器是如何通过现有的 ActiveX 控件扩展其能力，

我们通过几个例子和实验来进行演示。幸运的是，现成的样本非常多，我们可以从中进行选择。Visual C++和 Internet Explorer 随带一个免费特许的 ActiveX 控件集，其中的一部分在图 8-1 的 Gallery 对话中列出。如果想打开 Gallery 对话，请从 Project（项目）菜单中选择 Add To Project（添加到项目）命令，并单击次级菜单上的 Components And Controls（组件和控件）。然后双击 Registered ActiveX Controls 文件夹来显示控件列表。

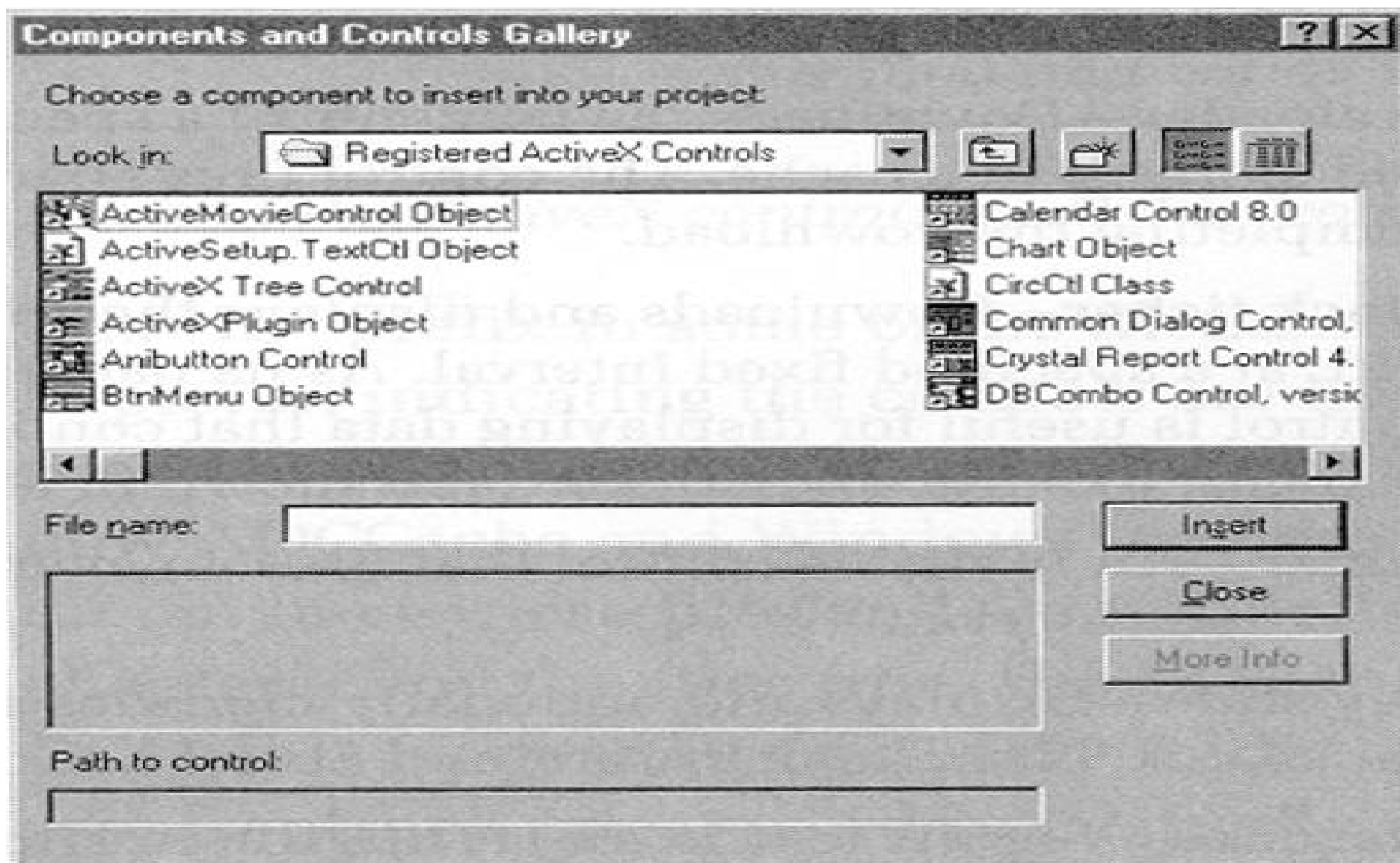


图 8-1 Gallery 中的 ActiveX 控件

表 8-1 列出了 Microsoft 提供的免费特许的 ActiveX 控件。如果在 Gallery 中选了一个控件图标，More Info（其他信息）按钮是启用的，那么，这就意味着该控件可以通过在线帮助来描述自己。单击 More Info 按钮可以查看该控件的资料。

表 8-1 一些可以从 Microsoft 中提供的 ActiveX 控件

文件 名	说 明
AniBtn32.ocx	Animated button（动画按钮）：使用位图或元文件来创建一个带有变化的图像的按钮
BtnMenu.ocx	Menu（菜单）：显示一个按钮和一个弹出式菜单，如图 8-2 所示。
DBGrid32.ocx	Grid（网格）：以标准网格样式显示单元格的电子表控件。用户可以选定单元格(与老的 Grid32 控件不同)，并直接将数据输入单元格。还可以由包容器通过编程来填充单元格，或为了自动更新而捆绑到记录集
IELabel.ocx	label（标签）：以一定角度或沿指定的曲线显示文字

续表

IEMenu.ocx	Pop-up menu（弹出式菜单）：显示一个弹出式菜单，如图 8-2 所示
IEPopWnd.ocx	Pop-up window（弹出窗口）：在弹出式窗口中显示 M T M L 文档
IEPrld.ocx	Preloader（预加载器）：下载指定 URL 的内容，并将它存储在高速缓存中。完成下载之后，该控件将触发一个事件
IEStock.ocx	stock ticker（股市自动接收机）：以指定的固定周期下载和显示 URL 的内容。正如它的名字所暗示的，该控件对于显示连续变化的数据时特别有用，如图 8-2 所示
IETimer.ocx	Timer（计时器）：一个不可见的控件，以一定的周期触发一个事件
keySta32.ocx	Key（键）状态：显示并有选择地修改 Cap Lock, Num Lock, Insert 和 Scroll Lock 键的状态

续表

Marquee.ocx	Marquee (大 屏 幕) : 水 平 或 垂 直 方 向 滚 动 HTML 文 件 中 的 文 字 , 可 以 被 配 置 成 改 变 滚 动 的 数 量 和 延 迟 时 间
MCI32.ocx	Multimedia (多 媒 体) : 管 理 Media Control Interface(MCI, 媒 体 控 制 接 口) 设 备 的 多 媒 体 文 件 的 录 音 和 播 放 。 该 控 件 可 以 显 示 一 套 用 于 将 MCI 命 令 传 向 设 备 的 推 压 式 按 钮 , 这 些 设 备 包 括 音 频 板 、 MIDI 序 列 发 生 器 、 CD-ROM 驱 动 器 、 音 频 CD 播 放 器 、 视 频 光 盘 播 放 器 以 及 视 频 磁 带 录 音 机 和 播 放 器 。 该 控 件 还 支 持 Video for Windows AVI 文 件 的 播 放
MSCal.ocx	Calendar (日 历) : 在 屏 幕 上 显 示 的 日 历 , 用 户 可 以 从 中 选 择 日 期 。

续表

MSChart.ocx	Chart (图表): 一个高级的图表控件, 它接收数字数据, 然后显示几种图表类型之一, 包括线、条和栏形图表。该控件的绘制以二维或三维形式显示, 见图 8-2 和图 8-3
MSComm32.ocx	Comm (通信): 为串行通信提供支持, 处理发送到串行口和从串行口接收的数据传输
MSMask32.ocx	Masked edit (掩码编辑): 一个增强的编辑控件, 它可以确保输入符合预先定义好的格式。例如, 一个 “##:##??” 的约束将输入限制为时间格式。例如 “11:18 AM”
PicCLP32.ocx	Picture clip (图片剪辑): 显示位图中剪下的矩形区域, 可以根据指定的行数和列数将位图划分为网格

在表 8-1 中, 一些文件名的 “IE” 前缀代表 Internet Explorer, 用来表明该控件是那个程序所包括的。OCX 文件可以位于系统之上的任何地方, 但通常都放在 Windows\OCCache 和 Windows\System 子文件夹中。如果由于某种原因, 你没有了这些文件, 并且想沿着本章的演示继续下去, 请从本书配套光盘的 OCX 文件

夹中将这此文件复制到你的 OCache, System 或 System32 文件夹中。不过，不要假定这个小小的样本就代表了 ActiveX 控件中的最新词汇。市场上每天都有新控件出现，你可以在自己的应用程序中免费使用它们的演示版。如果你想浏览 Internet 寻找一些控件，如下两个地址可以提供免费下载，并且提供了到应用程序开发者感兴趣的 Web 站点的链接：

<http://www.mcirosoft.com/com/>

<http://www.actrvex.com>

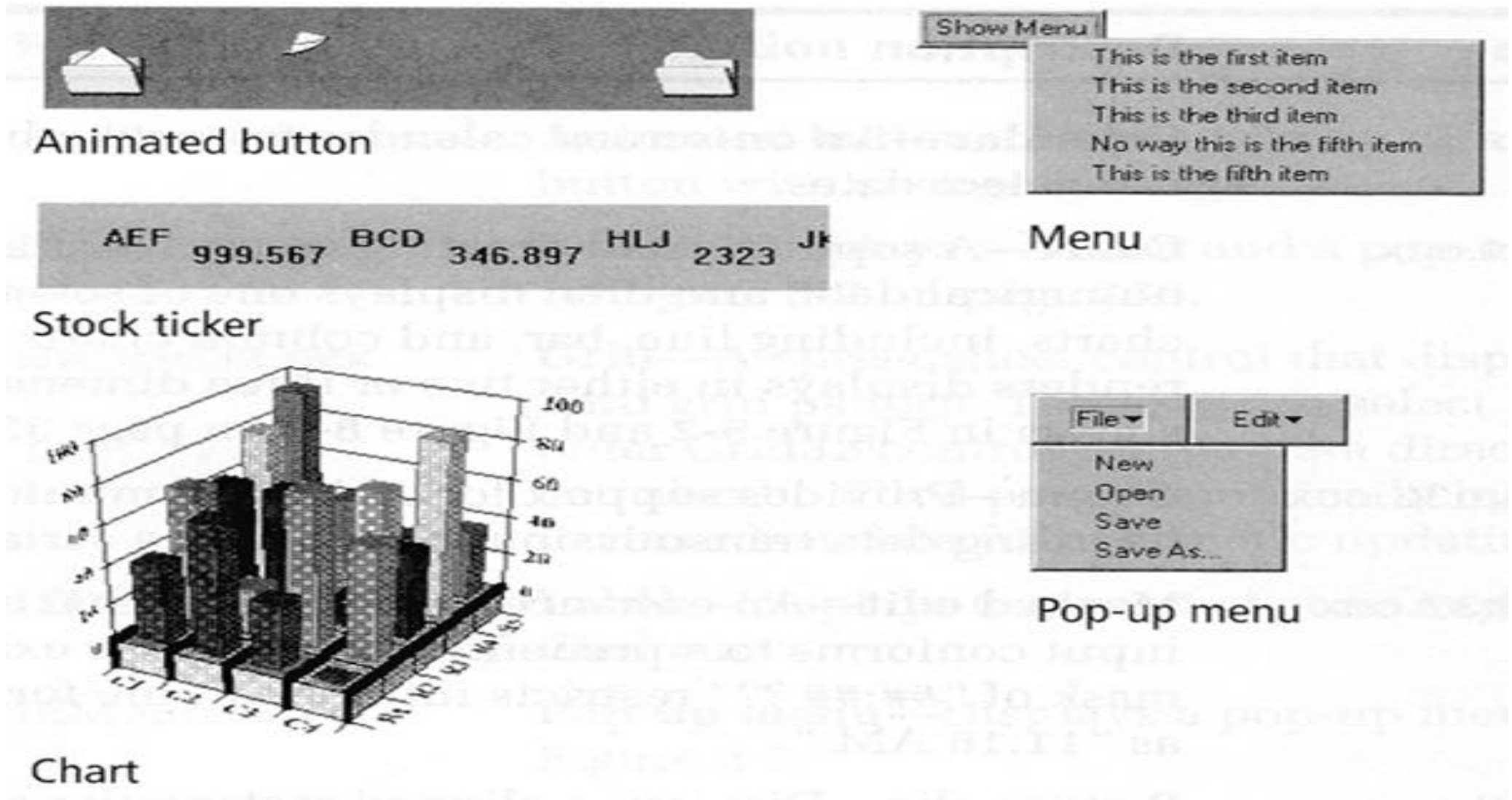


图 8-2 几个在包容器上出现的 Microsoft ActiveX 控件

当你从配套光盘或另一个来源向自己的硬盘复制控件文件的时候，请使用在 vc98\Bin 子文件夹中找到的 RegSvr32.exe 实用程序对其进行注册。RegSvr32 调用该控件的自注册函数，该函数将有关这个控件的信息写入系统注册表。在注册控件之前，包容器应用程序一般是没有办法来为嵌入而定位它的。单击 Start 按钮，并从 Run 对话框中执行 Regsvr32,在命令行中指定一个 ocx 文件：

```
regsvr32\windows\occache\anibtm32.ocx
```

如果你的系统 PATH 语句没有包括 VC98\Bin 文件夹，那么，请在键入 regsvr32 的时候指定正确的路径。如果想解除对 ActiveX 控件的注册(也就是说，从注册表中删除它的条目)，请按照相同的方式再运行一遍 RegSvr32,但是要在文件名前包括开关 “ /u ”。解除对控件的注册并不从磁盘上删除它的 ocx 文件。

你还可以通过单击 Tools（工具）菜单上的 Register Control（注册表控件）命令来从 Visual C++ 中运行 Regsvr32。不过，在默认状态下，该命令假定你想在结构之下注册一个控件，因此，它仅仅注册项目目标文件。第 13 章将解释如何修改像 Register Control（注册表控件）这样的工具，以便你可以将任何文件指定为命令行参数，而不仅仅是当前项目中一个文件。

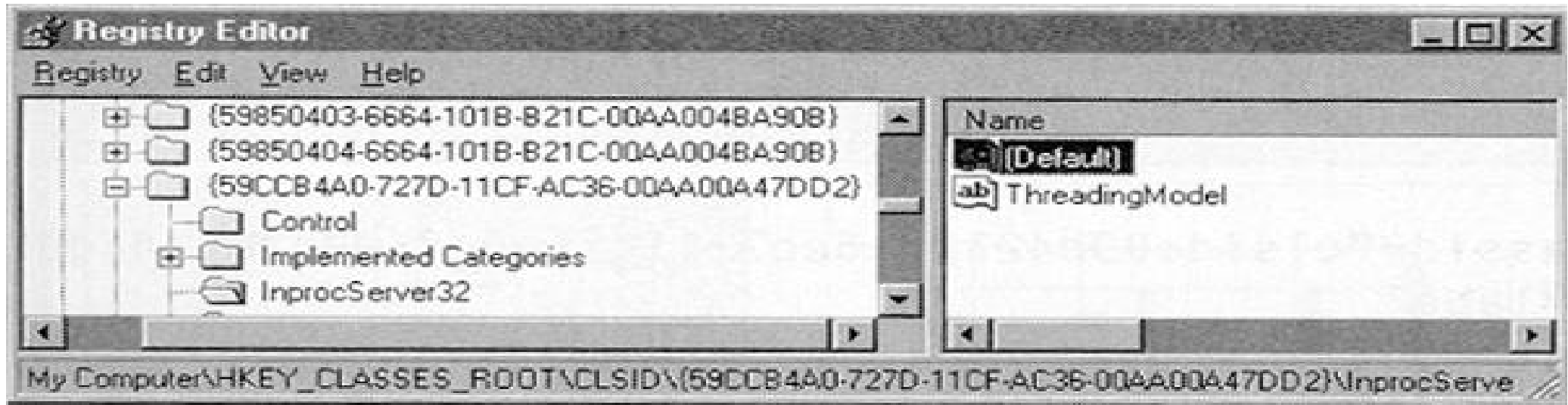
向 Web 页添加 ActiveX 控件

在向项目插入 ActiveX 控件之前，你可能想先看一看控件。你只需一个文本编辑

器和一个支持 ActiveX 的浏览器即可，一个用于创建 HTML 文档，另一个用于查看它。HTML 代表 Hypertext Markup Language(超文本标记语言)，它定义了一个简单的约定，以创建在各种书和文章中说明良好的 Web 页。仅仅通过几分钟的学习，你就可以了解到有关 HTML 的许多知识。Visual C++ 文本编辑器可以编辑 HTML 文件，在这方面，它的能力有限，但可以在显示窗口中为标记和其他文档元素加上颜色。

为了在 HTML 文档中使用 ActiveX 控件，你必须首先找到该控件的 32 个数字的类标符号码。我们将在下一章中更加详细地介绍 CLSID，现在，你只需知道如何查看该号码即可。注册表编辑器为找到控件的 CLSID 提供了一种方法。单击 Start 按钮，并在 Run 对话框中键入 regedit 或 regedit32，究竟键入哪个，需取决于系统是 Windows 95 还是 Windows NT。单击注册表编辑器 Edit 菜单上的 Find 命令，然后键入该控件的文件名。

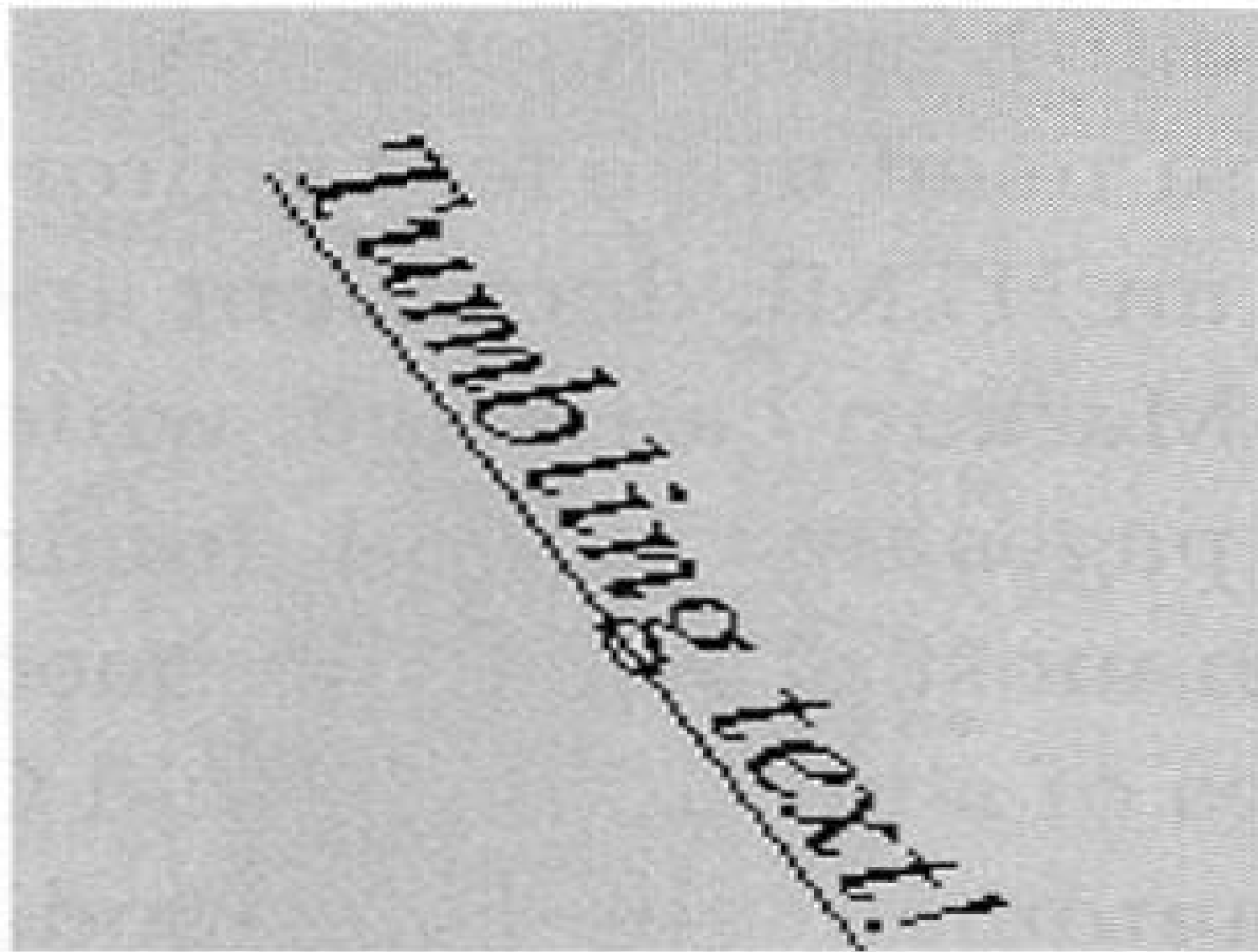
例如，在注册表编辑器中，搜索 ietimer.ocx，将找到注册表中的如下层次：



位于窗口底部的 32 位数字的号码代表 Timer ActiveX 控件的 CLSID。用相同的方式搜索 ielabel.ocx 将显示代表 Label ActiveX 控件的 CLSID：

99b42120-6ec7-11cf-abc7-00aa00a47dd2

应用手头的这两个号码,你可以编写一个简单的 HTML 文档,使用 Timer 和 Label 控件来显示文本,它们不停地翻滚,永无休止地从彩色框底部弹出：



如果想看动画，请使用 Internet Explorer 这样的浏览器，或任何可察觉 ActiveX

的创作工具，来打开本书配套光盘 Code\chapter.08 文件夹的 Tumble.htm 文档。在 Internet Explorer 3.0 及以后版本中，单击 Open 命令，并找到该文档，然后双击打开它。列表 8-1 显示了 Tumble.htm 文档的内容。

列表 8-1 Tumble.htm 文档

```
<OBJECT  
  
    classid="clsid:59ccb4a0-727d-11cf-ac36-00aa00a47dd2"  
  
    id=timer1  
  
>  
  
<PARAM NAME="Interval" value="100">  
  
<PARAM NAME="Enabled" value="TRUE">  
  
</OBJECT>  
  
<OBJECT
```

```
classid="clsid:99b42120-6ec7-11cf-a6c7-00aa00a47dd2"
```

```
id=label
```

```
width=150
```

```
height=150
```

```
>
```

```
<PARAM NAME="Angle" value="0">
```

```
<PARAM NAME="Alignment" value="7">
```

```
<PARAM NAME="BackStyle" value="1">
```

```
<PARAM NAME="BackColor" value="255">
```

```
<PARAM NAME="FontItalic" value="-1">
```

```
<PARAM NAME="FontUnderline" value="-1">
```

```
<PARAM NAME="Caption" value="Tumbling text!">
```

```
<PARAM NAME="FontName" value="Times New Roman">
```

```
<PARAM NAME="FontSize" value="18">
```

```
</OBJECT>
```

```
<SCRIPT LANGUAGE="VBSCRIPT">
```

```
sub timer1_timer
```

```
    label.Angle = (label.Angle + 5) mod 360
```

```
end sub
```

```
</SCRIPT>
```

Test Container实用程序

Visual C++提供了一个名为 Test Container 的工具，正如它的名字所暗示的那样，利用它，你不用创建自己的包容器应用程序，便可加载和试验已注册的 ActiveX 控件。单击 Tools（工具）菜单上的 ActiveX Control Test Container（ActiveX 控件

测试包容器) 命令, 来启动 Test Container, 如图 8-3 所示, 它带有两个典型的 ActiveX 控件, 分别叫做 Button Menu 和 Microsoft Chart, 在配套光盘中均有提供。该程序的可执行文件是 Tstcon32.exe, 在 Common\Tools 子文件夹中。

如果想在 Test Container 中加载控件, 请从 Edit (编辑) 菜单中选择 Insert New Control (插入新控件) 命令, 或者单击工具栏上的 New Control (新建控件) 按钮, 然后从 Insert Control (新建控件) 对话框中显示的列表中选择所需要的控件, 在 Text Container (文本包容器) 窗口中, 一个控件可以首次仅仅作为小框出现; 如果是这样的话, 请通过拖动一角来调整控件。控件的初始尺寸取决于该控件从包容器中申请的启动尺寸(如果有的话)。下一章中的一个范例项目演示了用 MFC 编写的 ActiveX 控件是如何调用 `COleControl::SetInitialSize` 函数来建立它的启动尺寸的。如图 8-3 所演示的帮样, 几个控件可以在 Test Container 中同时运行; 在每个控件窗口所圈定的矩形框内单击, 可以从活动控件中进行选择。当一个控件被选中之后, 尺寸控制柄将出现在它的边框之上, Test Container 工具栏上 Invoke Methods and Properties (调用方法和属性) 工具成为可以启用的。

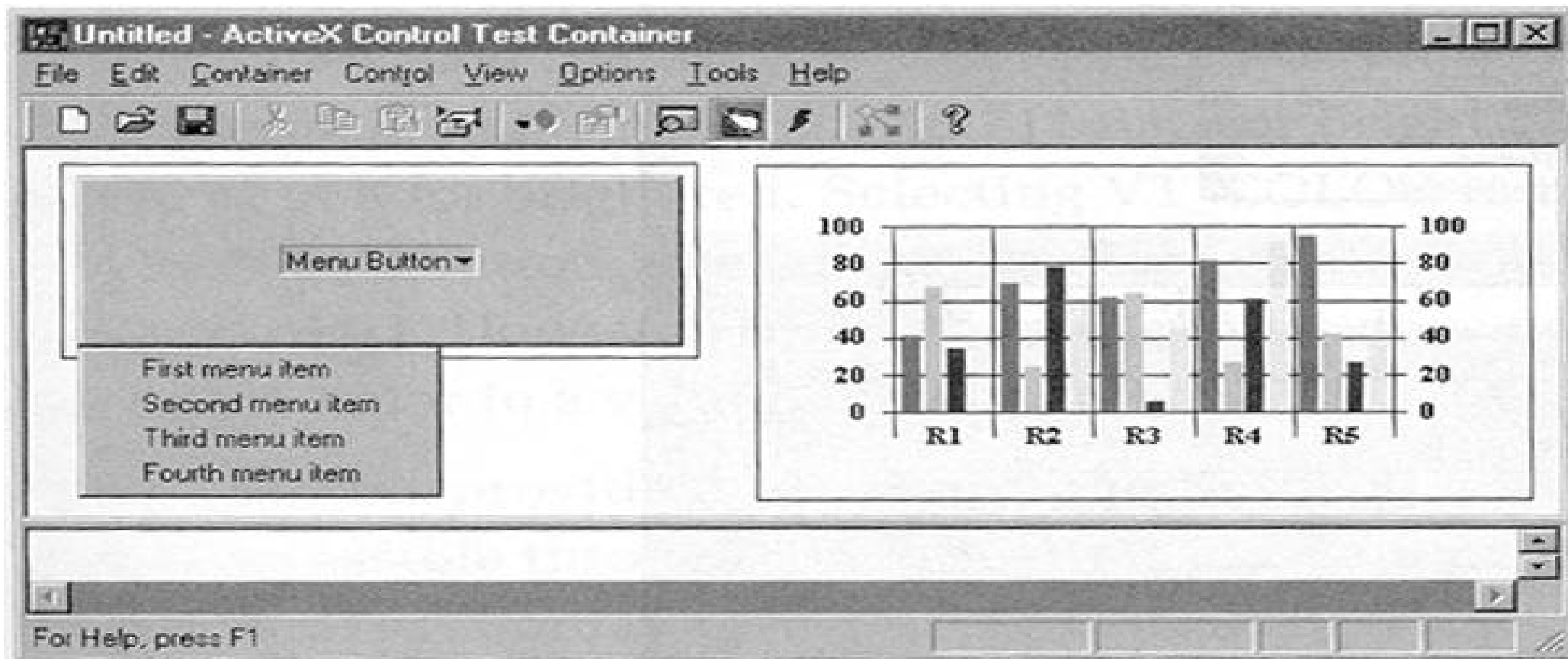
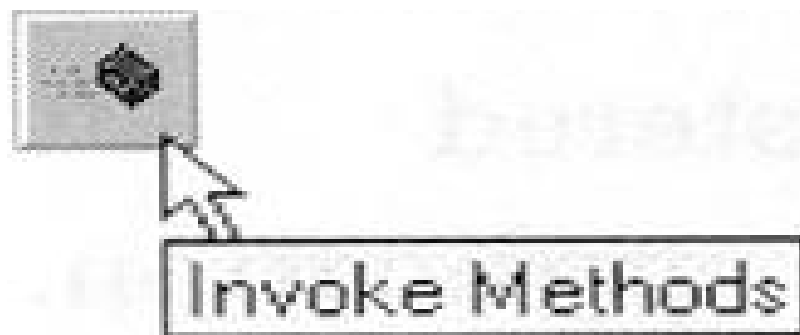


图 8-3 通过 Tools（工具）菜单调用的 Test Container 实用程序



被选中的 ActiveX 控件可以通过它的方法函数编程。单击 Invoke Methods (调用方法), 或从 Control 菜单中选择相应的命令, 来打开如图 8-4 所示的 Invoke Methods 对话。Method Name (方法名) 组合框的下拉式列表将所有控件方法归类, 大致可分为普通方法和属性方法两类。普通方法在下拉式列表中被标为 Method。属性方法被标为 PropGet 或 PropPut, 究竟标记为哪一个, 主要取决于它们所对应的是检索属性值的“获取”方法, 还是写属性值的“放置”方法。例如图 8-4 显示了 Button Menu 控件导出了两种方法类型, 允许包容器通过获取和放置方法读或写该控件的 Caption 属性(就是出现在按钮上的文字)。

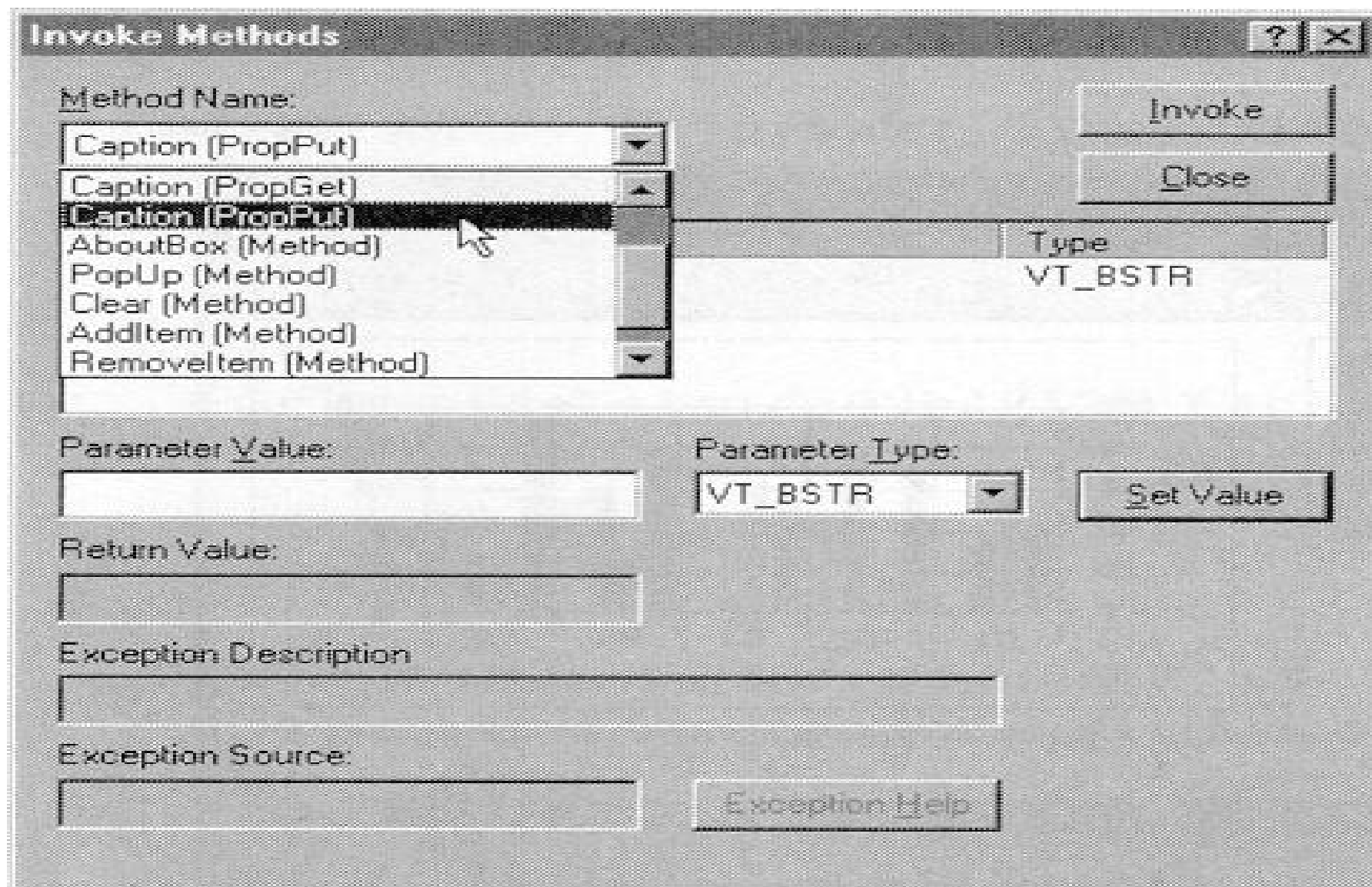
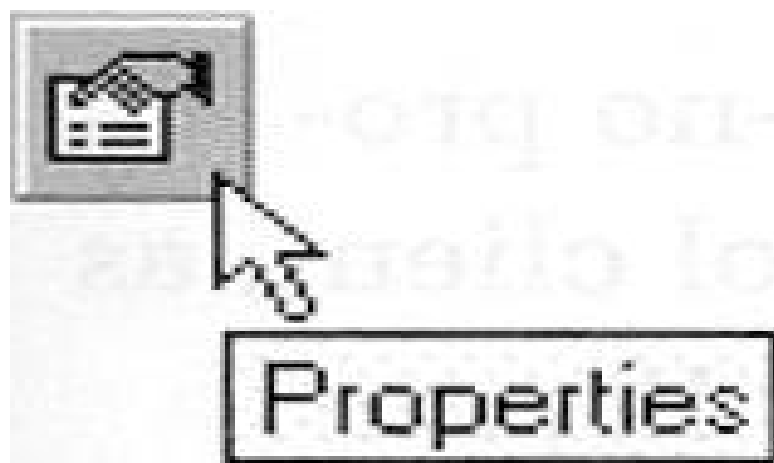


图 8-4 通过 Test Container 的 Invoke Methods (调用方法) 对话对控件编程

为了向 Button Menu 控件的弹出式菜单中添加一个项目，包容器调用 AddItem 方法。我们可以使用 Invoke Methods（调用方法）对话框做相同的事情。在 Method Name 框中选择 AddItem,然后在标签为 Parameter Value（参数值）的编辑框中键入所需求的文字。当你单击 Invoke 按钮的时候，Test Container 将调用 AddItem 方法,将文字添加到控件的菜单命令列表之中。为了看到新命令,可以关闭 Invoke Methods（调用方法）对话，并单击 Button Menu 控件。

当为一个整数类型的属性选中放置方法的时候，你必须还在 Parameter Type（参数类型）框中做出像 VT_14 这样的选择。通过先调用相应的获取方法，并记下返回值，或者在 Parameter Type（参数类型）框中选择 VT_UNKNOWN，可以确定正确的参数类型。如果放置方法有多个参数，请依次在 Parameters 列表框中选中每个变量，在键入值后单击 Set Value（设置值）按钮。当所有的值都在 Value 列中正确出现之后，单击 Invoke 来将参数传递给方法。

像 BackColor 这样的颜色属性是 24 位 COLORREF 值，它可以表示为 VT_14 整数类型。COLORREF 值的三个字节对应于整个颜色的红、绿和蓝组件，请参阅第 5 章中 Color 范例项目。尽管 COLORREF 值作为像 0xFF 这样的十六进制数来表达更为容易，Invoke Method 对话仅仅能识别以十进制格式键入的值。如果想在对话中输入一个新颜色值，可键入 16, 711, 680 来代表深蓝，65, 280 代表深绿，255 代表深红。在 Parameter Type（参数类型）框中选择 VT_COLOR，将启用标签为 Choose Color（选择颜色）的按钮显示范例颜色的种类。不过，该选项目前不能将选择的颜色正确转换成有效的方法参数。



许多控件提供它们自己的属性表, Test Container(测试包容器)可以通过 Properties 工具按钮访问它们。单击工具将导致 Test Container 向控件发布一条 OLEIVERB_PROPERTIES 命令, 告诉它显示属性表(如果有的话)。双击控件的边框还可以调用该命令, 其功能与从 Text Container 的 Edit (编辑) 菜单中选择 Properties 一样。

通过可移动的分隔条, 可以把 Test Container 窗口分成两个水平的窗格。底部视窗一般显示所选中控件触发事件的实时记录。记录也叫做事件日志, 可以重新确定路线, 方法是从 Test Container 的 Options (选项) 菜单中选择 Logging (记录) 命令。在 ActiveX 控件开发期间, 利用事件日志, 可以节省开发人员的时间, 让你快速测试控件事件是否已被正确触发。在下一章, 当我们测试范例 ActiveX 控件时, 我们还会看到事件日志的功能。

向对话框添加 ActiveX 控件

尽管从 CWnd 中派生出来的任何类都可以嵌入一个 ActiveX 控件，MFC 还需为对话框容器进行优化。这是很好的，因为 ActiveX 控件与普通控件一样，通常也是在对话框中出现的。这种优化是在 Visual C++ 中反映出来的，它提供了许多特征，可以帮助你为第 6 章介绍的基于对话的类(就是 CDialog, CPropertyPage, CFormView, CRecordView 或 CDaoRecordView)使用的 ActiveX 控件创建容器应用程序。向对话添加已经注册的类仅需单击几下鼠标即可，先是在 Gallery 中，然后是在 Visual C++ 对话编辑器中。

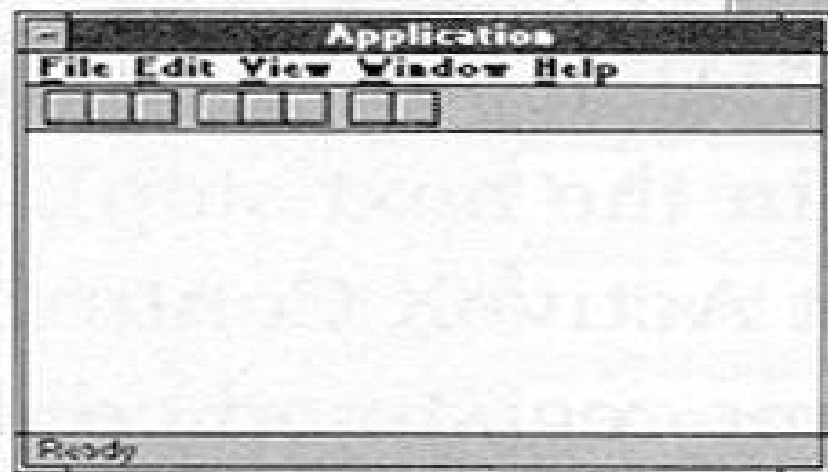
在某些方面，对话编辑器为 ActiveX 控件比 Test Container 实用程序提供了更为方便的测试区。其中之一便是，即使控件没有提供它自己的属性表，对话编辑器也将显示一个 Properties (属性) 框。例如，如果你双击了 Test Container 窗口中 Button Menu 控件的边框，将会出现一个消息框，说“Property Page are not supported (属性页未支持)”。但是，如果你在对话编辑器中为相同的命令调用了 Properties (属性) 命令，那么编辑器将向它的一般的 Properties (属性) 框添加一个标签为 All (所有) 的颜色选项卡，它列出了该控件的属性，并允许你进行编辑。All 选项卡比在 Test Container 实用程序中引用方法所包含的过程提供了到控件属性的更为方便的访问。

下面查看对话框中正在工作的 ActiveX 控件的方法：无需任何编程。对话编辑器自己作为控件客户，在这里，我们以 Animate Button ActiveX 控件为例进行演示。

如果你愿意在这些步骤中用完整的项目练习，请打开本书配套光盘上的 Code\Chapter.08\AniButtn 文件夹中的 Dewo.dsw 文件。运行 Demo.exe，并单击 Help 菜单上的 About 命令。

步骤 1：创建一个空项目

使用 AppWizard 创建一个空项目，为它取一个喜欢的名字，并接受所有的默认设置。在默认状态下，AppWizard 通过在步骤 3 中打开 ActiveX Controls 复选框，来使得每个应用程序都成为 ActiveX 控件容器：



What compound document support would you like to include?

- ☒ None
- ☐ Container
- ☐ Mini-server
- ☐ Full-server
- ☐ Both container and server
 - ☐ Active document server
 - ☐ Active document container

Would you like support for compound files?

- ☐ Yes, please
- ☒ No, thank you

What other support would you like to include?

- ☐ Automation
- ☒ ActiveX Controls

< Back

Next >

Finish

Cancel

选择 ActiveX Controls 选项，将运行大量的其他代码，来支持控件包容方式，但是，由框架来管理每件东西。在表面上，该选项只是把这行添加到应用程序类 InitInstance 函数：

```
AfxEnableControlContainer();
```

将这行添加到 StdAfx.h 文件：

```
#include<afxdisp.h>      //MFC OLE automation classes.
```

如果你有一个 MFC 项目想转化为控件包容器，请使用文本编辑器来手工实现上述改变。如果想得到相同的结果，还可以将位于 Gallery 的 Visual C++ Components 文件夹的 ActiveX Control Containment 组件添加到项目中。

步骤 2：插入 ActiveX 控件

当 AppWizard 完成创建项目之后，请从 Project(项目)菜单中选择 Add To Project (添加到项目) 命令，然后单击次级菜单上的 Components And Controls 命令来显示 Gallery 对话框。在 Registry ActiveX Controls 文件夹中选择 Anibutton Control 图标，并单击 Insert 按钮。接受 Confirm Classes (确认类) 对话框中的默认设置，然后退出 Gallery。

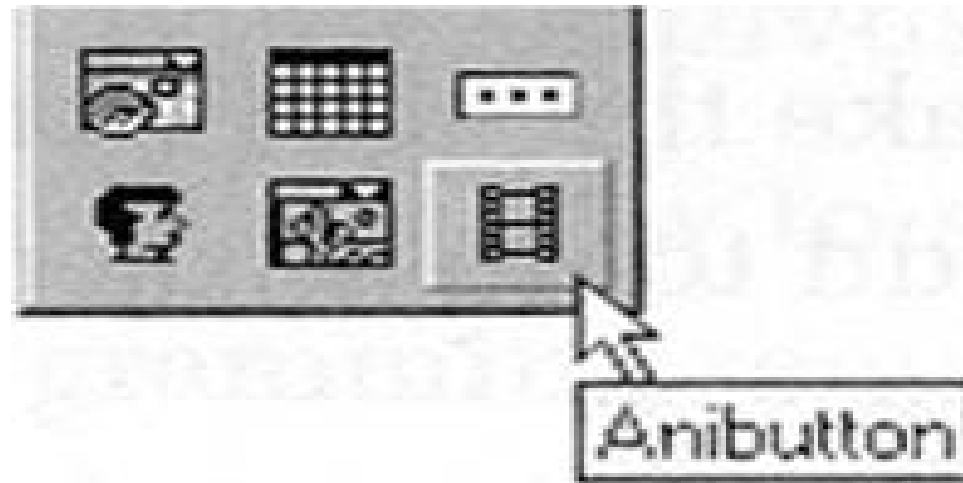
因为你还可以从对话框编辑器中向项目添加 ActiveX 控件，所以，进入 Gallery 不

是必需的。当对话编辑器的工作区域出现(在下一步中介绍)的时候，请右击工作区中的任何地方，并从上下文菜单中选择 Insert ActiveX Control。该操作将打开与 Gallery 对话框中显示的相同的注册控件列表。只需单击列表中的一个控件，便可以将它添加到对话框中去。

步骤 3：向对话框添加控件并初始化

从技术上讲，对话框容器不是 ActiveX 控件的父窗口，而仅提供一个站点(COM 这样称呼它)，这个词不应该从字面上来理解。站点的作用是嵌入式对象及其容器之间的中间人，在这种情况下，是处理 ActiveX 控件和对话框之间的通信。任何对话框都可以为实现演示的目的而做到过一点，甚至是项目的 About（关于）框也是如此。我们甚至不需要为使用新控件而创建项目。我们所需要的只是一个站点，以及对话框编辑器中模拟的 About（关于）框。

双击 Workspace（工作空间）窗口 ResourceView 窗格中的 IDD_ABOUTBOX，来启动对话框编辑器，并加载 About（关于）框。当编辑器的窗口出现的时候，它的控件窗口就具有了一个新按钮，用来代表插入的 Anibutton 控件。



这个工具并不是在工具栏上永久存在，它是仅仅为这个项目而使用的。如果想向另一个项目添加 Anibutton 组件，你必须再返回步骤 2 去插入一个控件。至于使控件进入对话工作区，没有特殊的处理要求。只需像你对任何控件工具操作的那样，把它拖入对话框，然后右击对话中选中的控件，并且单击上下文菜单上的 Properties，来调用该控件的 Properties(属性)对话。图 8-5 显示了 Anibutton Control Properties 对话的 Control 选项卡，这就是我们将开始初始化控件的地方。

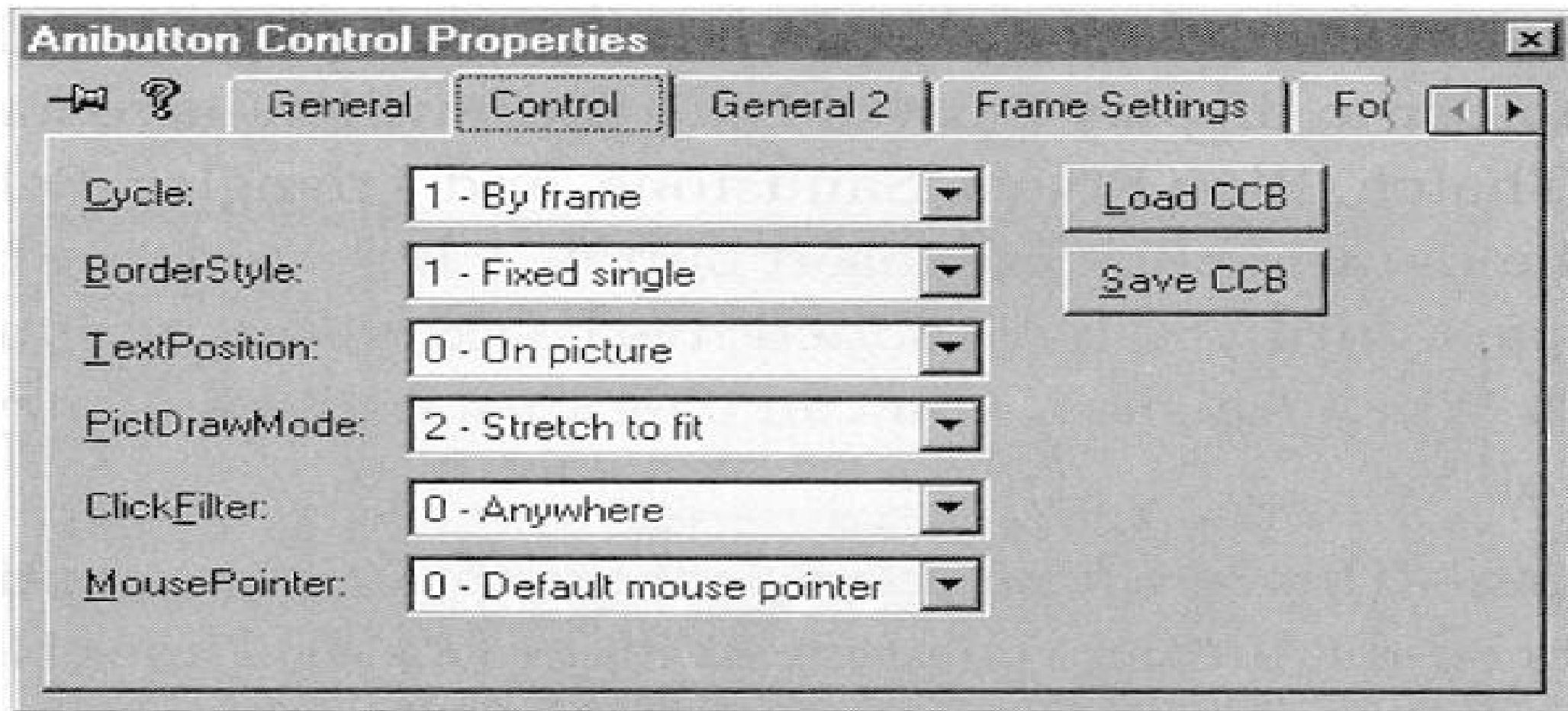


图 8-5 Anibuton Control Properties 对话框的 Control 选项卡

注意 :如果注册表中的某个条目丢失了,Visual C++ 将显示一条消息,说 Animated Button 控件需要一个设计时间许可证。如果你按照这里列出的步骤操作时出现了这条消息,这就表明,你要么装了只具有 USER 权限的 Visual C++, 或者是注

册表已经崩溃了。重新安装 Visual C++ 好象是唯一的解决方案。如果想了解这个问题，以及其他 ActiveX 控件的列表，请访问这个 Knowledge Base 站点：

<http://support.microsoft.com/support/kb/articles/Q155/0/59.asp>

如下的列表将引导你完成演示程序所需要的初始化设置。这些设置是在 Properties（属性）对话的五个选项卡中完成的。

- **Control（控件）选项卡**：在 Control 选项卡中单击组合框，并选中如图 8-5 所示的条目。
- **General 2（常规 2）选项卡**：如果想指定在控件窗口中显示的数字，请在 Caption（标题）框中键入 Click Here!。将复选框设置为 HideFocusBox，这可以防止在控件具有焦点的时候点线矩形出现在标题文字的周围。
- **Frame Setting（帧设置）选项卡**：Anibuton 控件可以保持大量充当按钮图像的位图。对于我们的演示，任何位图均可行，其中也包括 Windows 文件夹中的系统墙纸图像文件。单击 Load 按钮，并找到 Windows 文件夹来显示 BMP 文件列表，这些文件具有像 Black Thatch, BlueRivets, Sandstone 以及 Triangles 这样的名字。从列表中选中一个文件，并单击 Insert 按钮。再次单击 Load 按钮，并重复上述过程，直至已经向控件添加了五六个不同的位图。完成之后，你可以通过移动滚

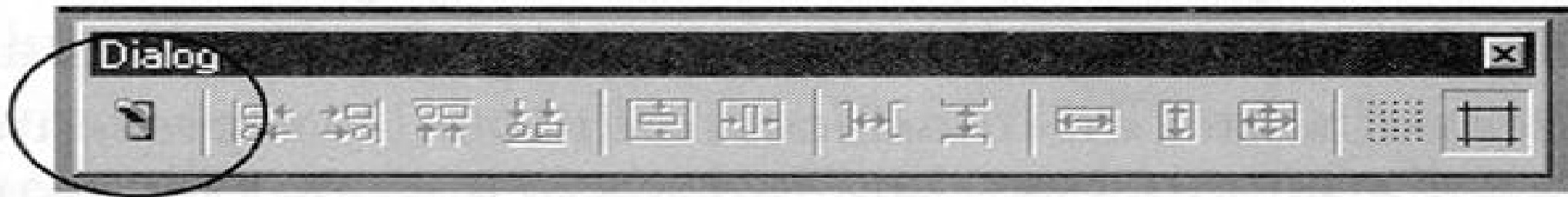
动条来检查每个位图条目。

- **Fonts（字体）选项卡**：这是你为按钮上出现的标题选中字体的地方。图 8-6 中的字体是 Times New Roman，斜体，磅值为 32。
- **Colors（颜色）选项卡**：因为位图图像是填充控件窗口的，所以，背景颜色无关紧要。通过从 Property Name 文本框中选定 ForeColor，并单击白色块来设置标题文字的前景颜色。

在本章前面的章节中，已经提到了，并不是所有的 ActiveX 控件都提供自己的属性表，但 Anibuton 控件就是这样。上面列出的属性页是在 AniBtn32.ocx 可执行文件中包含的资源，对话编辑器选取并添加自己的 General 和 All 选项卡，以形成如图 8-5 所示的完整的 Properties（属性）对话。这就意味着，你无需与两个对话进行交互，这两个对话一个由控件提供，而另一个由对话编辑器提供。

步骤 4：测试控件

通过拖动尺寸控制柄在对话工作区内扩大控件，然后，在对话框的中心重新定位控件窗口。打开 Dialog（对话）工具栏上的编辑器测试模式开关：



在新 ActiveX 控件窗口内部单击几次，循环显示位图图像，其中一个如图 8-6 所示。单击对话的 OK 按钮来返回到编辑模式。



图 8-6 典型对话中的 Anibutton ActiveX 控件

既然我们对 ActiveX 控件可以实现的许多形式已经有了一定的了解，那么，让我

们来分析其中的一个，看看它是如何工作的。

在容器和 ActiveX 控件之间进行通信

ActiveX 控件服务器可以非常有效地与一个客户进程关联起来。一个 ActiveX 控件通常在作为动态链接库工作，这就意味着该控件在客户进程的地址空间执行，但这并非一定如此。由于这个原因，ActiveX 控件通常称为进行中的服务器。容器程序在加载普通 DLL 的时候，并不是通过调用 LoadLibrary API 函数来加载这个 ActiveX 控件。相反，它调用 CoCreateInstance 来申请 Component Object Model 框架的运行时间服务来加载库，并在客户和控件服务器之间建立初始通信点。这个通信点叫做接口。容器所调用的一个接口，一般在图中用一个小圈来表示，如图 8-7 所示。在服务器中，实现对正确函数的调用。请注意该图，一旦初始接口就位之后，所有对象就将开始彼此之间的对话，图中没有包括 COM。

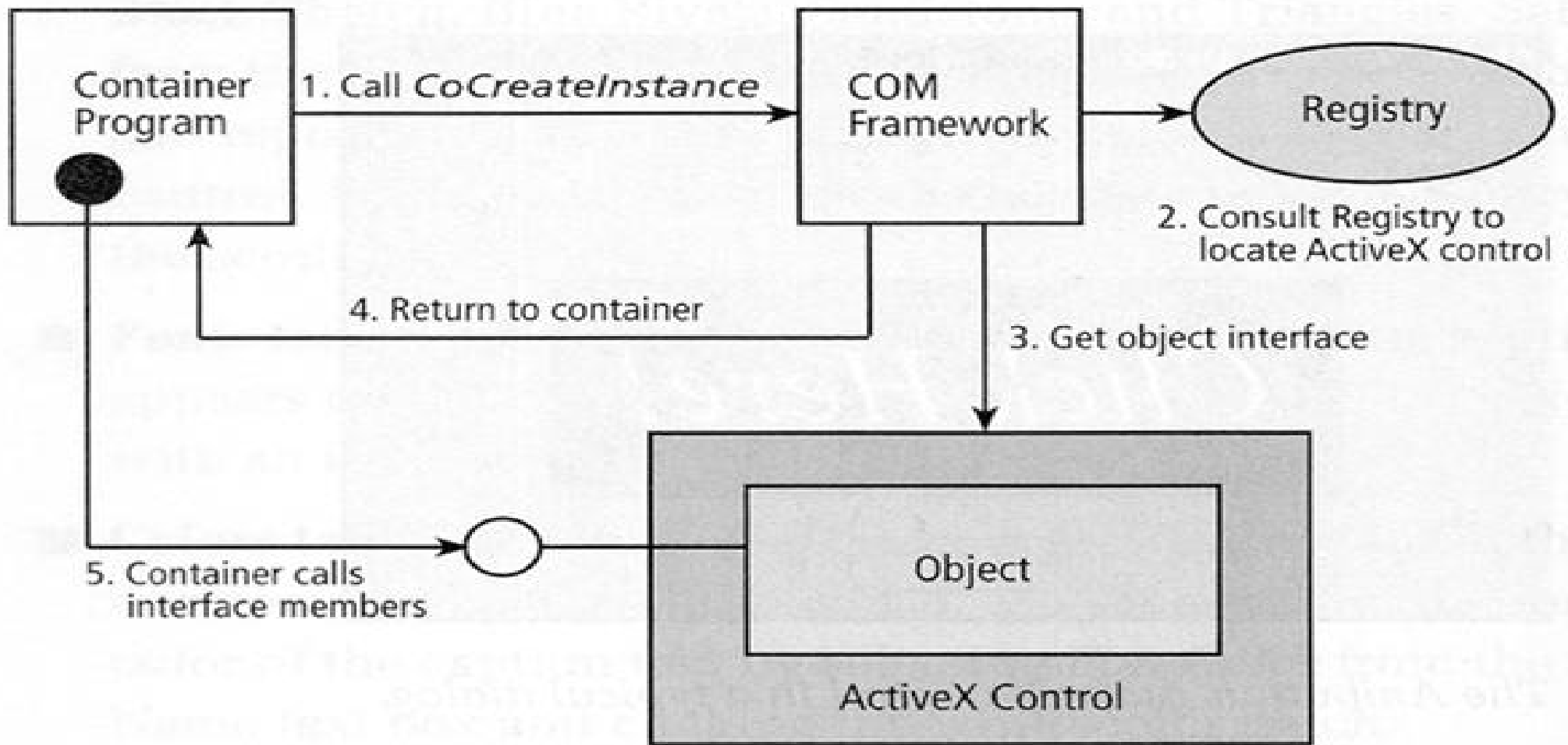


图 8-7 将 Activex 控件连向包容器

每个接口都是一个指向 Activex 控件导出的函数的指针数组。这个数组一般叫做 V 表，因为它十分像指向虚拟函数的指针的 C++ 表。因为仅有接口的单一间接步

骤处于客户和运行的服务器之间，所以，调用 ActiveX 控件实际上和调用一个普通的动态链接库一样迅速。

并不是所有的 COM 服务器都处于运行状态，服务器 EXE 应用程序在它自己的地址空间中运行，或者是与客户相同的机器，或者是通过网络附加到另一台机器上。在这种情况下，客户和服务器都是通过处理边界进行分隔的，并不能直接进行通信。对于进程之外的服务器，COM 为了处理通信而加载两个动态链接库。第一个库叫做代理，被映射到服务器空间。当客户调用代理的接口的时候，代理将函数参数打包，通过远程过程调用(RPC)将它们发送到 Stub。Stub 将包中的信息转换回参数列表，并调用服务器中的目标函数。从服务器回到客户的通信的是同一个路径。通过代理和存根库连接客户和服务器的过程叫做编组。正如你所希望的那样，编组比客户和 ActiveX 控件之间更加直接的交互更慢，因为运行的服务器并不依赖远程过程调用来与客户进行通信，除非通信是线程之间的，否则也不需要编组。第 10 章将详细讨论线程内的编组。

通信在 ActiveX 控件和它的包容器间的两个方向运行，因此，包容器应用程序必须提供它自己的一套接口，才能从控件处接收呼叫。Microsoft 发布了指定包容器应该支持的最小接口集的指导方针。该指导方针在在线帮助中有说明，可以通过 MSDN Library 窗口的 Index 选项卡进行访问。从 Visual C++ 的 Help 菜单选择 Index 命令，然后键入 required interface 来定位该标题的文章。

通过支持这些接口，包容器应用程序可以确保与遵守该指导方针的任何 ActiveX

控件融在一起。表 8-2 介绍了包容器为了遵守 OLE/ActiveX 技术规范而应该支持的八个接口。

表 8-2 中的头三个接口提供复合文档包容器，而不是控件包容器。用 MFC 编写包容器程序，可以使你不必担心接口支持的详细情况。正如本章以前曾经介绍的，在 AppWizard 中，为一个包容器项目选择 ActiveX 控件支持，将向源代码添加一个对框架的 `AfxEnableControlContainer` 函数的调用。该函数用于建立表 8-2 中列出的所有接口。一旦这些接口就位之后，ActiveX 控件和它的包容器之间的通信就可以通过事件、方法和属性来发生了。

事件

尽管 ActiveX 控件是完备的，它仍然可以通过触发事件使包容器应用程序了解到控件内部的行为。对于某个控件触发的事件，控件开发者便认为包容器应用程序想知道有关控件的事情。例如，控件可以为了响应控件窗口内的鼠标单击，或者在控件有焦点的时候，向包容器传递任何收集到的键盘输入来触发一个事件。一个触发事件可能发送信号，来表明类似于定位 URL、下载数据或分类列表这样的任务已经完成。我们可以把事件触发和普通控件向它的父窗口发送诸如 `CBN_DROPDOWN` 或 `BN_DOUBLECLICKED` 之类的通知消息做一个类比，我们可以看到，除了 ActiveX 控件是通过调用包容器中的一个函数，而不是通过发送消息来触发事件的之外，两者是很相似的。

表 8-2 包容器应该支持的遵守 OLE/ActiveX 技术规范的接口

接口	说明
<i>IOleClientSite</i>	嵌入式对象用来查询包容器有关客户站点的大小以及用户接口的特性。IOleClientSite 接口还提供了诸如 RequestNewObject Layout 函数之类的服务，控件可以通过这个函数来为它的站点申请新的大小
<i>IAdviseSink</i>	由对象使用，用来将对象数据中的改变通知给包容器
<i>IOleInPlaceSite</i>	用于管理包容器和对象站点之间的交互
<i>IOleControlSite</i>	为嵌入式 ActiveX 控件提供各种服务。例如，Translate Accelerator 函数用来请求包容器处理指定的键击，OnFocus 函数用来告诉控件它是否具有输入焦点

续表

<i>IOleInPlaceFrame</i>	ActiveX 控件用来统治像组成菜单这样的资源的显示
<i>IOleContainer</i>	允许控件强制它的包容器保持在运行状态，或访问其他嵌入在同一文档或窗口中的其他控件
<i>IErrorInfo</i>	支持双重接口包容器所需要的(请参阅第 10 章)
<i>IDispatch</i>	控件用来访问包容器的外部层性，并调用该包容器的事件处理程序函数。该包容器为属性和事件实现独立的 Idispatch 接口

接收触发事件的包容器中的函数是回调类型的函数。如果包容器应用程序想被通知一个特殊的控件事件，它必须提供一个函数(也就是众所周知的事件处理函数或事件实现函数)来接收调用。包容器在一个 IDispatch V 表(也就是事件接收端)中存储到它的事件处理程序的指针列表。这个事件接收端将每个事件与它自己的处理程序函数连接起来。这个包容器应用程序不必要为该控件触发的每个事件提供处理程序函数，也不是每个 ActiveX 控件都触发事件。

OLE/ActiveX 标准预定义了大量的库存事件，它们将控件窗口中所发生的事情通

知给容器。例如，当鼠标在控件窗口内部单击的时候，为了将这个库存事件通知给容器，一个使用 MFC 的控件可以通过 EVENT_STOCK_CLICK 宏建立 Click 事件：

```
BEGIN_EVENT_MAP(CDemoCtrl, COleControl)
```

```
    //{{AFX_EVENT_MAP(CDemoCtrl)
```

```
    EVENT_STOCK_CLICK()
```

```
    //}}AFX_EVENT_MAP
```

```
END_EVENT_MAP()
```

因为由框架来负责感知鼠标移动和触发事件，所以，该控件不需要其他代码。如果容器想知道鼠标单击在控件窗口中是何时发生的，它可以为 click 事件提供一个处理程序函数，可以在匹配事件接收端映射中引用它。

```
BEGIN_EVENTSINK_MAP(CDemoContainer, Cdialog)
```

```
    //{{AFX_EVENTSINK_MAP(CDemoContainer)
```

```
    ON_EVENT(CDemoContainer, IDC_CTRL, DISPID_CLICK,
```

```
OnClick,VTS_NONE)
```

```
//}}AFX_EVENTSINK_MAP
```

```
END_EVENTSINK_MAP()
```

在上面的代码段中，ON_EVENT 宏的参数可能需要解释一下。CDemoContainer 是该容器的类，它是从 CDialog 中派生的。常数 IDC_CTRL 用来标识类对话框中的控件。DISPID_CLICK 是 Click 事件的调度标识符(简称为 dispid)。用于库存事件的调度标识符在 OleCtl.h 文件中定义，每个标识符都具有一个 DISPID_ 前缀。所有的非库存事件都叫自定义事件，OLE 可以为它分配一个正的调度标识符，而为库存事件保留负的标识符。该宏的第四个参数指向用于处理事件的容器成员函数。本例中，该成员函数的名字是 OnClick。VTS_NONE 参数用于指定 Click 事件没有参数。

表 8-3 列出了由 OLE/ActiveX 技术规范定义的库存事件的函数原型。除了 Error 之外，所有库存事件者仅仅在 ActiveX 控件具有输入焦点时发生。事件原型好像可以暗示单一函数的存在，事实上，通常至少包括三个函数，如图 8-8 所示。在较低层次上，一个控件通过调用容器的 IDispatch::Invoke 方法，并传递给它适于该事件的参数来触发一个事件。但是，在更高的层次上，存在着两个其他的函数，一个位于 ActiveX 控件中用包装对 IDispatch::Invoke 的调用，另一个位于容器中用于处理调用。两个函数共享同一个参数表，而在效果上是单一的函数，隐藏了发生在它们之间的低级的 IDispatch 行为。函数名是任意的。MFC 通过向

事件名前添加前缀 Fire，来形成触发函数的名字，例如 FireClick 函数触发的是 Click 事件。

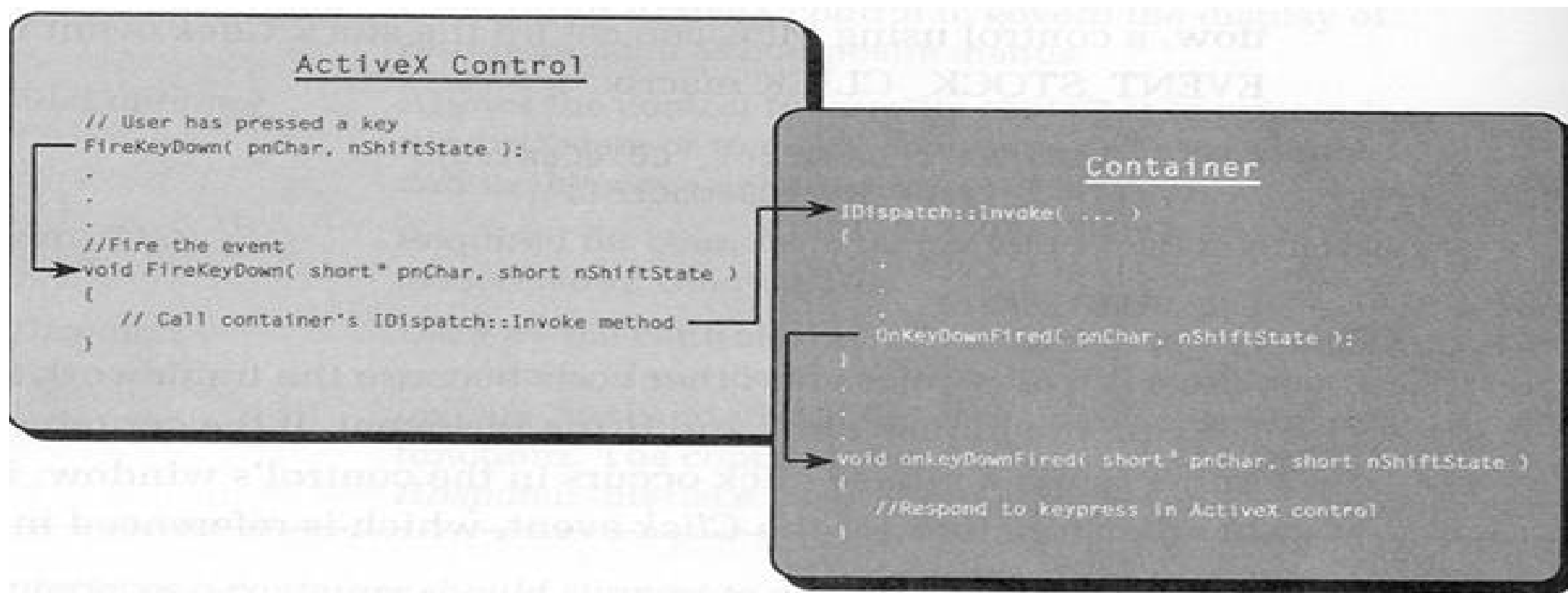


图 8-8 触发一个典型的事件

表 8-3 由 OLE/ActiveX 定义的库存事件

事件原型	当时触发事件
<code>void FireClick()</code>	在控件窗口中单击了任何鼠标按钮(左, 中或右), <code>MouseDown</code> 和 <code>MouseUp</code> 库存事件在 <code>Click</code> 之前触发
<code>void FireDbClick()</code>	在控件窗口中双击了任何鼠标按钮
<code>void FireError(SCODE scode, LPCSTR lpszDescription, UNIT nHelpID=0)</code>	控件检测到错误
<code>void FireKeyDown(short* pnChar, short nShiftState)</code>	控件接收到 <code>WM_SYSKEYDOWN</code> 或 <code>WM_KEYDOWN</code> 消息
<code>void FireKeyPress(short* pnChar)</code>	控件接收到 <code>WM_CHAR</code> 消息

续表

<code>void FireKeyUp(short* pcChar, Short nShiftState)</code>	控件接收到 WM_SYSKEYUP 或 MW_KEYUP 消息
<code>void FireMouseDown(short nButton, short nShiftState, float x, float y)</code>	按下任何鼠标按钮(左、中或 右) , 生成一条 WM_xBUTTONDOWN 消息
<code>void FireMouseMove(short nButton, short nShiftState, float x, float y)</code>	控件接收到 WM_MOUSEMOVE 消息
<code>void FireMouseUp(short nButton, short nShiftState, float x, float y)</code>	任何鼠标按钮被释放, 生成一个 WM_xBUTTONUP 消息

方法

方法与事件处理函数刚好相反。事件处理函数位于包容器之中，并由控件调用，方法则位于控件之中，由包容器调用。包容器可以调用方法来了解条件，或申请控件完成一些操作。

OLE/ActiveX 预定义了三个库存方法，分别叫做 DoClick, Refresh 和 ActiveX，它们中没有一个具有参数或返回值。Doclick 导致控件触发它的 Click 库存事件(如果支持它的话)，Refresh 方法告诉控件使它的窗口失效，并自己重绘，AboutBox 告诉该控件去显示一个信息丰富的对话框。ActiveX 控件导入的任何其他方法都叫做自定义方法，它们是由控件的作者设计的。对于包容器而言，方法是作为由动态链接库导出的普通函数出现的，带有一个可选的参数表，最多可带有 15 个参数，并返回任何类型的值。

属性

属性是包含在包容器和控件内部并可以互相展示的公共数据。OLE/ActiveX 定义了四类属性，分别叫做库存的、自定义的、环境的和扩展的。库存和自定义属性属于控件，环境和扩展属性属于包容器。

库存和自定义属性

库存属性用于指定由 ActiveX 标准定义的典型控件特性，例如控件的前景和背景颜色，在窗口中显示的文字，以及用于文字的字体。自定义属性是控件设计者想展现给容器的所有其他数据。容器通过调用 MFC 中作为 Get 和 Set 方法的函数来读写控件的属性，这些方法是由该控件导出的。在使用 Test Container 的 Invoke Methods 对话的时候，我们曾经遇到过相同的 get/put 属性方法。不同之处是名称方面的差异，WFC 以前缀 Get 和 Set 作为方法名的开头，而 COM 术语使用 get 和 put(小写形式)。一般说来，每个属性都有一个对应的 Get/Set 方法对，但控件可以不为它导出 Set 方法，来阻止容器改变控件属性。第 6 章将演示这是如何实现的。

表 8-4 显示了控件中的库存属性和容器为读取该属性而调用的函数之间的链接。对于在表中第三列列出的每个获取方法，都存在一个具有与其相匹配的名字对应的 Set 方法。Set 方法没有返回值，只有一个类型与 Get 方法的返回值相同的参数。GetAppearance 和 SetAppearance 的原型说明了 Get/set 函数的样式：

```
short GetAppearance( )      //Returns a property of type short
```

```
void SetAppearance( short n ) //Passes a property of type short
```

表 8-4 由 OLE/ActiveX 定义的库存控件属性

属 性	控件中调度映射项	由 包 容 器 调 用 的 Get 函数
<i>Appearance</i>	DISP_STOCKPROP_APPEARANCE	short GetAppearance()
<i>BackColor</i>	DISP_STOCKPROP_BACKCOLOR	OLE_COLOR GetBackColor()
<i>BorderStyle</i>	DISP_STOCKPROP_BORDERSTYLE	short GetBorderStyle()
<i>Caption</i>	DISP_STOCKPROP_CAPTION	BSTR GetText()
<i>Enabled</i>	DISP_STOCKPROP_ENABLED	BOOL GetEnabled()
<i>Font</i>	DISP_STOCKPROP_FONT	LPFONTDISP GetFont()

续表

<i>ForeColor</i>	DISP_STOCKPROP_FORECOLOR	OLE_COLOR GetForeColor()
<i>hWnd</i>	DISP_STOCKPROP_HWND	OLE_HANDLE GetHwnd()
<i>Text</i>	DISP_STOCKPROP_TEXT	BSTR GetText()
<i>ReadyState</i>	DISP_STOCKPROP_READYSTATE	long GetReadyState()

环境和扩展属性

环境和扩展属性属于客户站点，不能被控件改变。扩展属性是与嵌入的控件相关的数据，但由包容器进行实现和管理。环境属性用于描述包容器本身。例如，它的当前背景颜色或字体。通过阅读它的包容器的环境属性，控件可以对它的外观和行为进行剪裁，以与包容器相匹配。控件通过调用带有用于所要求属性的调度标识符的 `COleControl::GetAmbientProperty` 函数来查询环境属性：

`LPFONTDISP fontdisp;`

```
GetAmbientProperty(DISPID_AMBIENT_FONT,VT_FONT,&fontdisp);
```

对于由 OLE/ActiveX 技术规范预定义的标准环境属性,一个控件可以更加方便地调用由 COleControl 提供的相关帮助器函数,例如 AmbientFont:

```
LPFONTDISP fontdisp=AmbientFont();
```

表 8-5 列出了包容器所支持的标准环境属性。ActiveX 控件通过调用使用表的第二列中列出的调度标识符之一的 GetAmbientProperty,或者通过调用第三列中等价的帮助器函数,来确定环境属性的值。如果你使用 AppWizard 来创建包容器应用程序,那么,对标准环境属性的支持就已经在内部创建了,而不需进行特殊的操作。调用 SetFont 或 SetTextColor 在包容器中设置字体或前景颜色,将自动在对话中为站点设置环境属性。当 ActiveX 控件调用 AmbientFont 或 AmbientForeColor 函数的时候,它接收到用于当前对话的环境数据。

编写包容器应用程序

包容器开发者是如何事先知道一个 ActiveX 控件所提供的事件、方法和属性,以及你的包容器应用程序应包含哪些处理程序函数呢? Gallery 和 ClassWizard 可以为你完成这项工作。在你的包容器程序中,使用现有的控件取决于单个的许可证协议(这是下一章将要涉及的主题),但是,你可以从 Gallery 中选择一个 ActiveX

控件，并将它作为任何其他组件添加到你的项目中。Visual C++自动扫描注册表，来定位在系统上注册的所有控件，因此，向 Gallery 添加 ActiveX 控件仅仅是注册它的事情。

表 8-5 标准的包容器环境属性

属 性	调 度 标 识 符	控 件 调 用 的 函 数
<i>BackColor</i>	DISPID_AMBIENT_BACKC OLOR	OLE_COLOR AmbientBackColor()
<i>DisplayName</i>	DISPID_AMBIENT_DISPLA YNAME	CString AmbientDisplayName()
<i>Font</i>	DISPID_AMBIENT_FONT	LPFONTDISP AmbientFont()
<i>ForeColor</i>	DISPID_AMBIENT_FORECO LOR	OLE_COLOR AmbientForeColor()
<i>LocaleID</i>	DISPID_AMBIENT_LOCALE ID	LCID AmbientLocaleID()
<i>ScaleUnits</i>	DISPID_AMBIENT_SCALEU NITS	CString AmbientScaleUnits()

续表

<i>ShowGrabHandles</i>	DISPID_AMBIENT_SHOWGRABHANDLES	BOOL AmbientShowGrabHandles()
<i>ShowHatching</i>	DISPID_AMBIENT_SHOWHATCHING	BOOL AmbientShowHatching()
<i>TextAlign</i>	DISPID_AMBIENT_TEXTALIGN	short AmbientTextAlign()
<i>UIDead</i>	DISPID_AMBIENT_UIDEAD	BOOL AmbientUIDead()
<i>UserMode</i>	DISPID_AMBIENT_USERMODE	BOOL AmbientUserMode()

当 Gallery 在你的包容器项目中放置 ActiveX 控件的时候，它将检查该控件的可执行图像中包含的类型库，以获得由该控件所导出的事件、方法和属类的一个列表。从这个信息中，Gallery 可以创建一个完整的包装类，该类中包含有 Get/Set 属性函数和方法调用，包容器可以通过它们获得到控件数据的访问。如果想在控件中获得或设置属性(例如背景颜色)，包容器将调用包装类中的一个函数：

```
OLE_COLOR CDemoCtrl::GetBackColor()

{

    OLE_COLOR result;

    GetProperty(DISPID_BACKCOLOR,VT_I4, (void*)&result);

    return result;

}
```

```
void CDemoCtrl::SetBackColor(OLE_COLOR propVal)

{

    SetProperty(DISPID_BACKCOLOR,VT_I4,propVal);

}
```

既然事件处理程序属于容器的类，通常由 CDialog 这样的类中派生出来，那么，Gallery 就不为事件处理程序函数添加源代码。在控件被添加到对话之中，这项

工作留给 ClassWizard 完成。

最后，举例来说明这个过程。本节创建了一个叫做 Hour 的简单容器应用程序，它使用本书配套光盘中包括的一个 ActiveX 控件。该控件与以前在 Tumber.htm 文档中使用的是同一个 IETimer.ocx 定时器控件。在 Gallery 的 Registered ActiveX Control 文件夹中名称 Timer Object 下列出的控件之中，你可以找到这个 IETimer 控件。这个文件夹中的控件列表可能还包括另一个定时器 ActiveX 控件，它是由名为 Time Control 的样本项目中创建的 (Time Control 的源文件位于文件夹 MSDN\Samples\VC98\MFC\Controls\Time 之中)。两个定时器控件导出相同的方法，并实现相同的功能，因此，Hour 项目使用哪一个都无所谓。

与 Anibutton 和 Calendar 这样的 ActiveX 控件不同的是，Timer Object 不是一个可见的控件。在容器内，它不作为窗口显示，而只是以指定的周期触发一次事件，为包含程序起到定时器的作用。Hour 程序使用定时器事件来管理如图 8-9 所示的三个进程指示器。这个进程控件以分秒和十分之一秒的形式来显示剩余时间。Hour 程序的名称来源于这种事实：当 Minutes 进程控件满 60 分钟之后，三个控件便会复位。

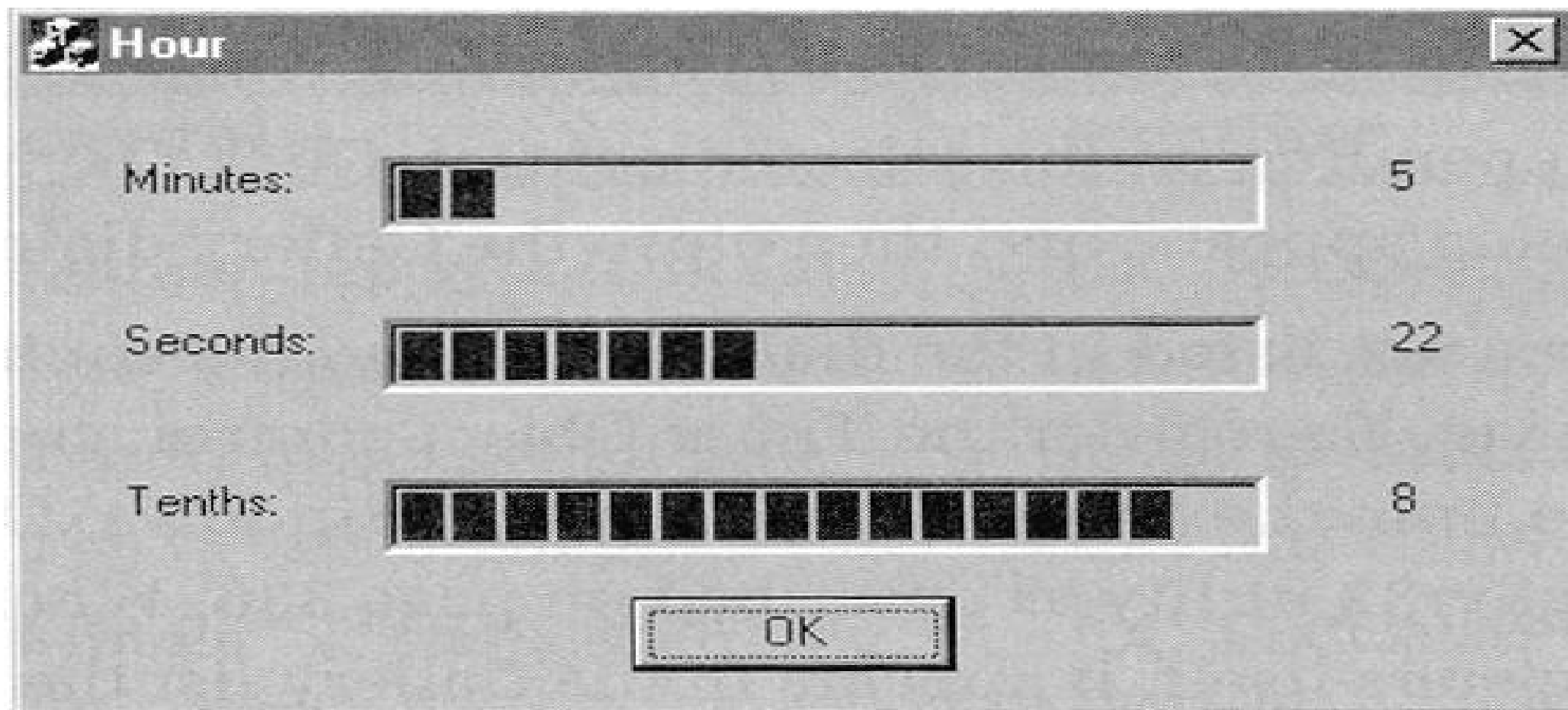


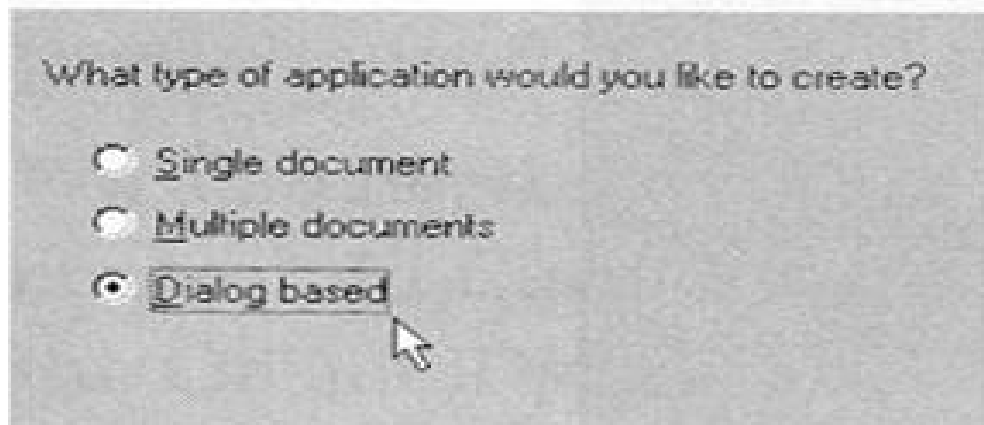
图 8-9 Hour 程序

从开始到结束创建 Hour 项目只需五步。

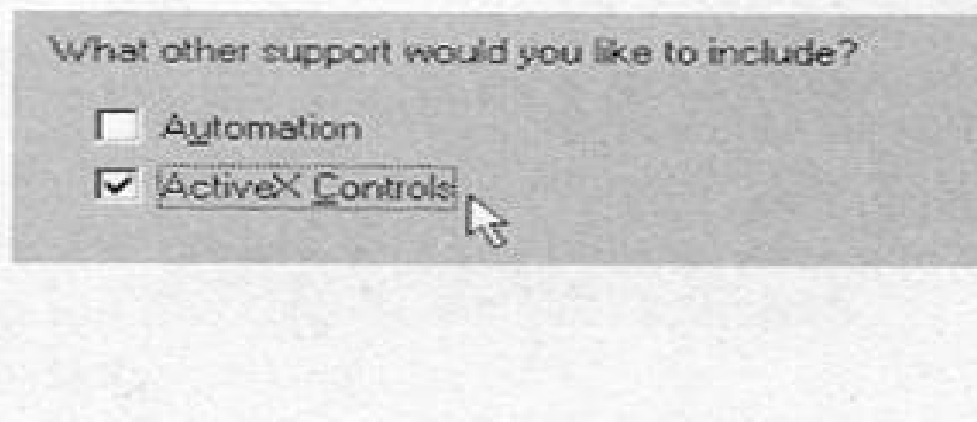
步骤 1：用 AppWizard 创建 Hour 项目

从环境的 File 菜单选择 New，在 Projects 选项卡中选中 MFC AppWizard(exe)图标，键入 Hour 作为项目名。Hour 是基于对话的应用程序，因此，单击 AppWizard 步骤 1 中的 Dialog Based（基于的对话）单选按钮，并确保步骤 2 中的 ActiveX Controls 复选框已经打开：

AppWizard's Step 1



AppWizard's Step 2



单击 Finish 按钮创建该项目。

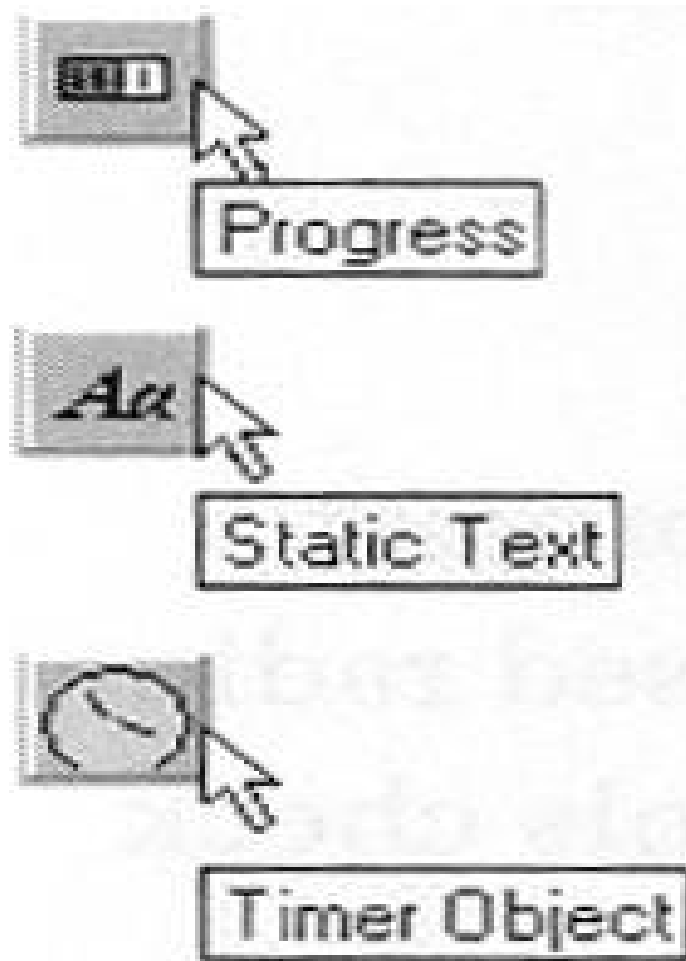
步骤 2：将 Timer Object 控件插入项目

现在，你应该很熟悉这一步骤了。使用 Project 菜单上的 Add To Project 命令来打

开 Gallery。并显示如图 8-1 所示的 ActiveX 控件列表。水平滚动并选择 Timer Object 或 Timer Control 图标，然后单击 Insert 按钮。该操作将把 CIETimer 或 CTimerCtrl 类的源代码添加到 Hour 项目，究竟是哪一个类取决于所选定的控件。当 Confirm Classes 对话框出现的时候单击 OK，然后关闭 Gallery 对话框。

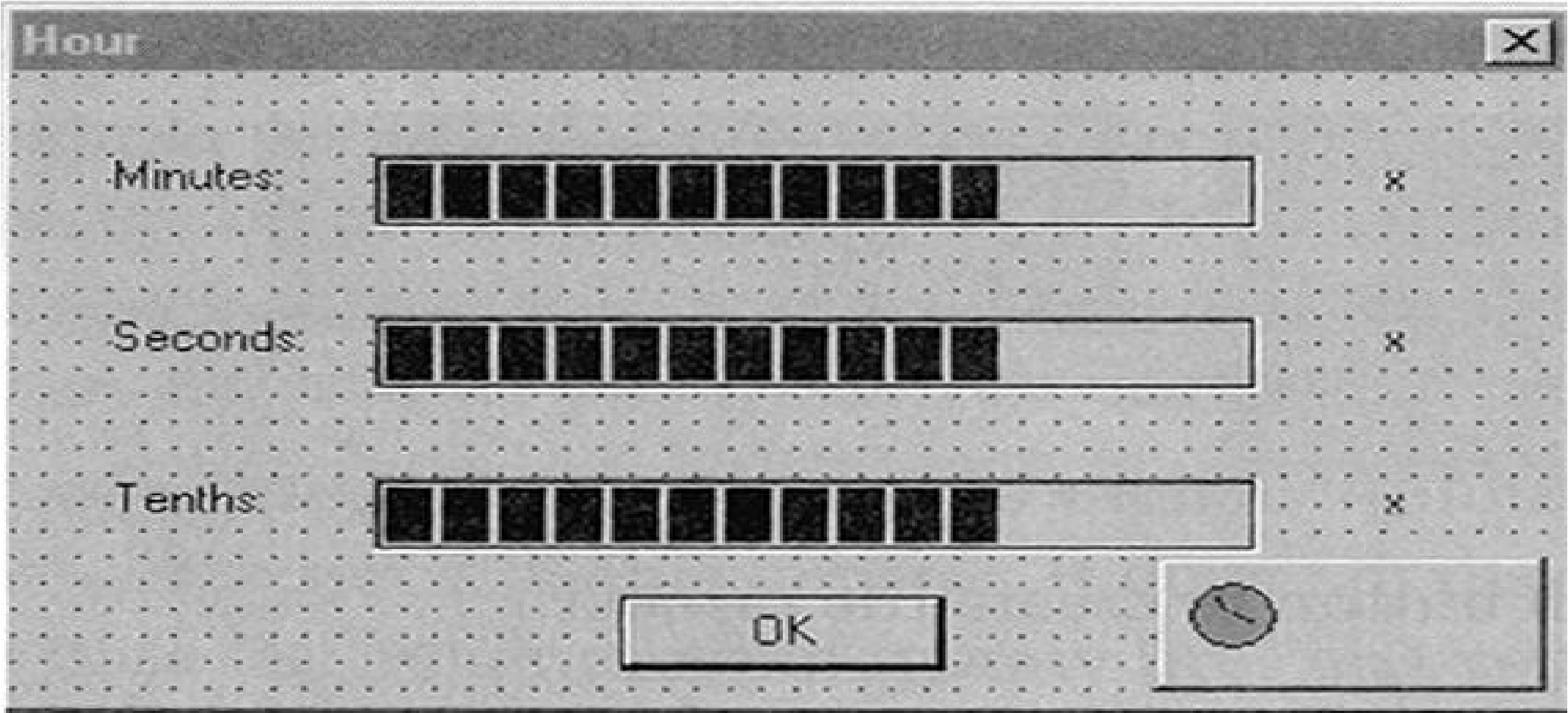
如果 Timer Object 没有在对话框显示中列出，那么，该控件就还没有注册。如果想注册 Timer Object 控件，请将 IETimer.ocx 文件从配套光盘中复制到 Windows\OCCache 文件夹，并运行 RegSvr32 实用程序。当控件成功注册了自己之后，当你下次打开对话框时，它将在 Gallery 列表中出现。

步骤 3：在 Hour 对话框中放置 Timer Object 控件



在 Visual C++ 的早期版本中，你必须双击 ResourceView 窗格中的 IDD_HOUR_DIALOG 标识符，来启动对话编辑器，并加载主对话。通过选中对话工作区中的“to do”静态文字控件和 Cancel 按钮，并按 Del 键来删除它们。将 Progress, Static Text 和 Timer Object 工具从 Controls 工具栏拖到工作区之上，

并对其进行排列，使它看上去与下图类似：



由于 Timer Object ActiveX 控件在程序运行的时候没有创建自己的窗口，所以，你把它放在对话中的什么地方都可以。展现每个控件的 Properties 框，并键入屏幕图像上显示的标题，以及在表 8-6 中第二列列出的标识符。

表 8-6 Hour 程序中的控件标识符

控 件	标 识 符	变 量 名
分钟进度指示器	IDC_PROGRESS_MIN	progMin
秒钟进度指示器	IDC_PROGRESS_SEC	progSec
十分之一秒进度指示器	IDC_PROGRESS_MIN	progTen
顶端“x”静态控件	IDC_MINUTES	strMin
中间“x”静态控件	IDC_SECONDS	strSec
底部“x”静态控件	IDC_TENTHS	strTen
时间控件	IDC_TIMER1	time

应用程序类 CHourDlg 需要代表每个对话控件的成员变量，你可以通过 ClassWizard 来添加它。在对话编辑器仍处于活动状态的时候，单击 View 菜单上的 ClassWizard 命令来调用第 6 章中介绍的 MFC ClassWizard。在 Member Variables（成员变量）选项卡中，选中 Control IDs 框中的每个新类，并单击 Add Variable（添加变量）按钮，来显示 Add Member Variable（添加成员变量）对话。在标

签为 Member Variable Name（成员变量名）的文本框中，键入表 8-6 中第三列列出的控件变量。图 8-10 显示了最后的结果。

我们还需要一个函数来处理由 Timer Object 控件触发的事件。在 ClassWizard 的 Message Maps（消息映射）选项卡中，选中 IDC_TIMER1，并从 Messages 框中选中 Timer，然后单击从 Object IDs 框中的 Add Function（添加函数）按钮。ClassWizard 添为名为 OnTimerTimer1 的事件处理程序函数添加存根代码。



“E”前缀用来将 OnTimeTimer1 指定为一个事件处理程序函数。单击 OK 来关闭 ClassWizard 对话。

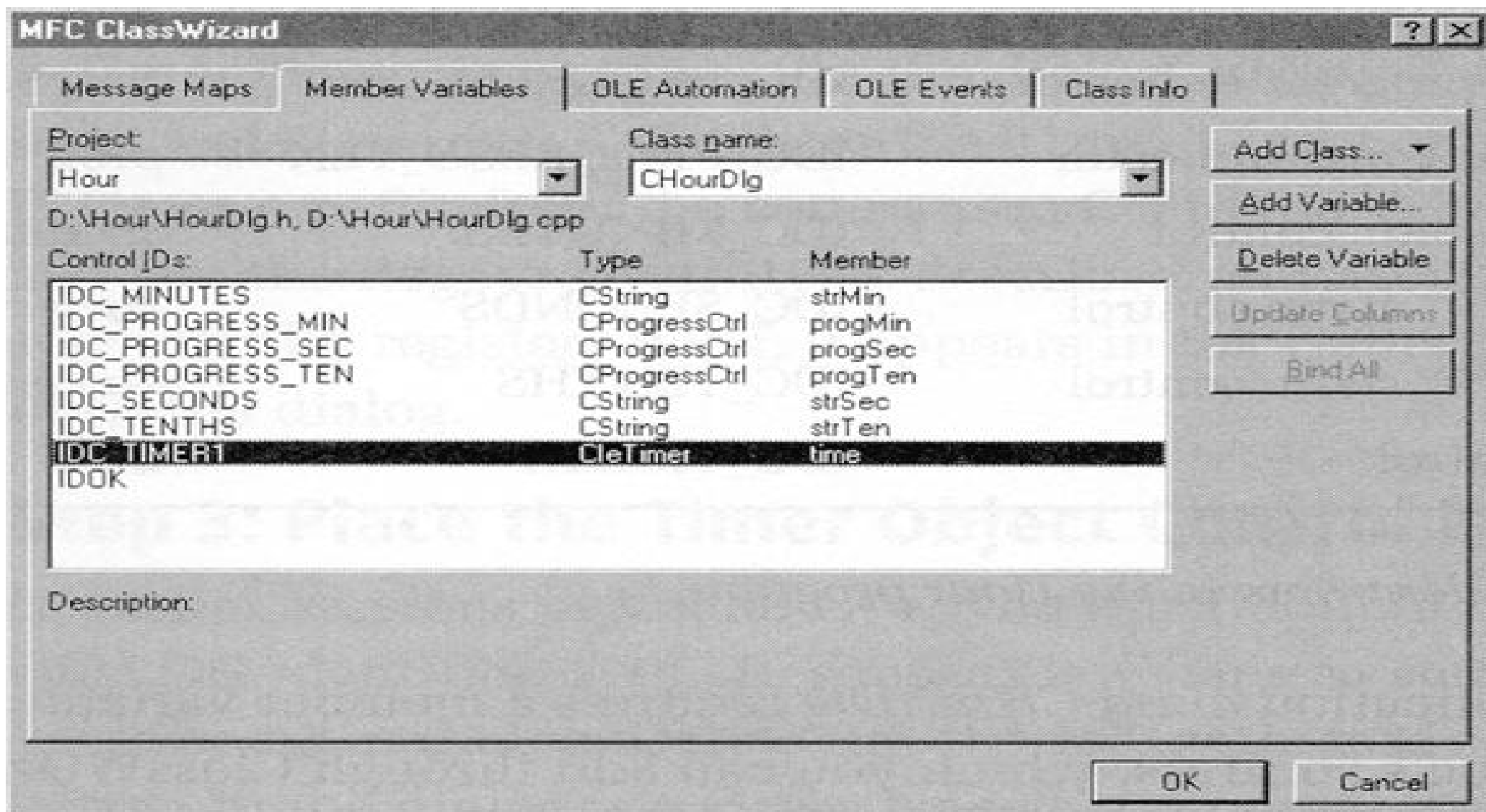
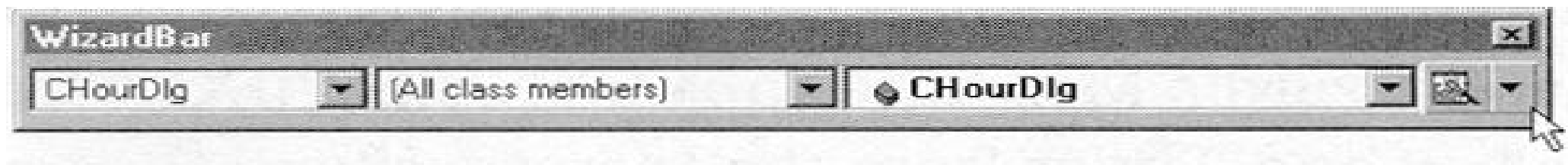


图 8-10 向 CHourDlg 类添加成员函数

步骤 4：向 Hour.cpp 和 Hour.h 文件添加代码

如果想查看一下 ClassWizard 已经添加到 HourDlg.h 头文件的变量和函数声明，请单击位于 WizardBar 最右端的箭头按钮：



并从下拉式菜单中选择 Go To Class Definition（转向类定义）。Visual C++ 将在文本编辑器中自动打开 HourDlg.h，并将插入符定位在 CHourDlg 声明的开始处，在这里添加新控件变量。

```
// Dialog Data
```

```
//{{AFX_DATA(CHourDlg)
```

```
enum { IDD = IDD_HOUR_DIALOG };
```

```
CProgressCtrl    progTen;
```

```
CProgressCtrl    progSec;
```

```
CProgressCtrl    progMin;
```

```
CString          strMin;
```

```
CString          strSec;
```

```
CString          strTen;
```

```
    CIETimer      time;
```

```
//}}AFX_DATA
```

ClassWizard 还可以为 OnTimerTimer1 事件处理程序函数添加原型：

```
afx_msg void OnTimerTimer1();
```

```
DECLARE_EVENTSINK_MAP()
```

CHourDlg 类声明仅需两行：

```
class CHourDlg : public CDialog
```

```
{
```

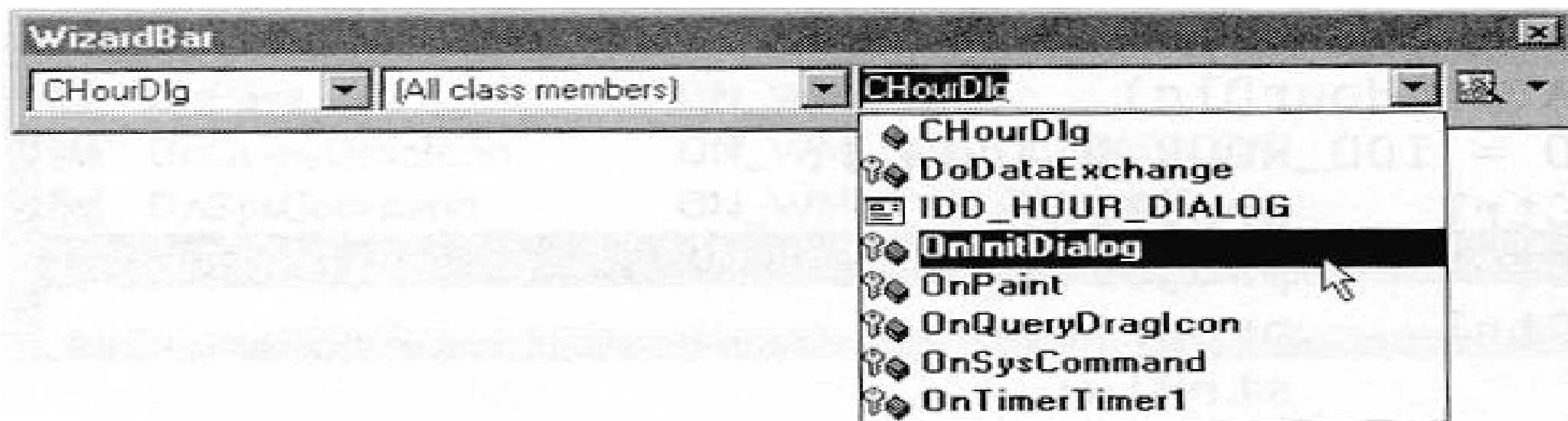


```
private:  
  
    int        iMin, iSec;
```

像以前一样，阴影部分表示你必须在文本编辑器中键入的附加代码。

变量 `iMin` 和 `iSec` 用于保持已经过去的分和秒的计数情况，它们被写到对话框中与进程指示器相邻的静态控件之中。对于消逝的十分之一秒，不需要类似的计数器，因为 `IDC_PROGRESS_TEN` 进程指示器的位置在由 `Timer Object` 控件触发的每个事件之前。过一会，当我们向事件处理程序添加代码的时候，这一切就会变得很清楚。

对源代码的最后修改是在 `CHourDlg::OnInitDialog` 函数中进行的。在 `WizardBar` 的 `Members` 框内任何地方单击，来显示成员函数的下拉列表，并从如图所示列表中选择 `OnInitDialog`。



Visual C++在文本编辑器中打开 HourDlg.cpp 源文件，并且自动将插字符放在 OnInitDialog 定义的开始处。在该函数的“to do”行添加阴影文字，如下所示：

```
// TODO: Add extra initialization here
```

```
time.SetInterval( 100 );           // Set timer interval to 1/10 second
```

```
progMin.SetRange( 0, 59 );         // Set ranges for progress indicators
```

```
progSec.SetRange( 0, 59 );         // Seconds: 1-60
```

```
progTen.SetRange( 0, 9 );           // Tenths:   1-10

progMin.SetStep( 1 );               // Set step intervals for prog indicators

progSec.SetStep( 1 );

progTen.SetStep( 1 );

iMin = iSec = 0;                    // Initialize tallies
```

这些指令用于初始化进程指示器控件。指令：

```
time.SetInterval(100); //Set timer interval to 1/10 second
```

调用 Time Object 中的一个方法，告诉控件开始每隔 100 毫秒触发一次事件。

使用 WizardBar 向下到 CHourDlg::OnTimerTimer1 函数，并添加如下阴影行：

```
void CHourDlg::OnTimerTimer1()
```

```
{
```

```
// TODO: Add your control notification handler code here
```

```
int i = progTen.StepIt ();
```

```
if (++i == 10)
```

```
{
```

```
if (++iSec == 60)
```

```
{
```

```
    if (++iMin == 60)
```

```
    {
```

```
        iMin = 0;
```

```
        progMin.SetPos( 0 );

    }

    else

        progMin.StepIt ();


    iSec = 0;

    progSec.SetPos( 0 );

    strMin.Format( "%d", iMin );

    SetDlgItemText( IDC_MINUTES, strMin );

}

else

    progSec.StepIt ();
```

```
i = 0;

progTen.SetPos( 0 );

strSec.Format( "%d", iSec );

SetDlgItemText( IDC_SECONDS, strSec );

}

strTen.Format( "%d", i );

SetDlgItemText( IDC_TENTHS, strTen );

}
```

每隔十分之一秒，OnTimerTimer1 实现函数接收到该控件的触发事件，并将 IDC_PROGRESS_TEN 进程指示器提前一步。当 Tenths 进程指示器达到最大值之后，该指示器被复位到零，Seconds 指示器加 1。按照同样的方式，在 60 秒过去之后，Seconds 指示器被复位到零，Minutes 指示器加 1。当一个小时过去之后，

整个过程再重新开始。

步骤 5：创建和测试项目

在 Build 工具栏上选中 Win32 Release 配置，然后创建 Hour.exe 程序的发行版本。单击 Build 菜单上的 Execute 命令来运行完成后的程序。请注意，Hour 运行得有点儿慢，它是依靠系统定时器资源的一个典型的 Win32 程序。尽管你可以把它作为一个煮蛋定时器使用，但 Timer Object 控件并不适合于需要精确定时的应用程序。

下一章将介绍名为 Game 的另一个包容器项目，它与 Hour 十分相似。不同之处是 Game 使用你自己编写的而不是由 Microsoft 提供的自定义 ActiveX 控件。

在没有对话编辑器的情况下工作

利用 Visual C++ 对话编辑器，就可以很容易地向对话框添加控件。但是，当想在窗口而不是对话中放置一个控件时，可能就要花些时间了。不存在任何技术上的障碍(只是因为常规控件可以在框架化的窗口中出现，因此 ActiveX 控件也能)，但你必须走在对话编辑器服务的前面。本节将解释如何向非对话窗口添加 ActiveX 控件，通过将 Button Menu 控件放入应用程序的客户区来演示这点。

这里所介绍的 Button 项目仅需要键入一些东西即可；否则，你可以在配套光盘上找到所有的源文件。请注意，该项目没有使用对话编辑器，因此，在开发期间从来都不用例子来说明 Button Menu 控件。本节中介绍的技术可以克服一些控件在设计阶段所存在的创建错误。任何 ActiveX 控件都可以按照这些步骤放在窗口之中，应用程序将正确地进行编译。但是，这里可以进行真正的许可保护，因为未经许可的应用程序不能在运行时实例化一个受保护的控件。下一章当我们讨论有关许可的问题时，就会清楚这一点了。

步骤 1：创建 Button 项目

运行 AppWizard 开始项目，在 AppWizard 的第一屏中选中 Single Document（单文档）选项。在 AppWizard 完成之后，打开 Gallery 中的 Registered ActiveX Controls 文件夹，并双击标签为 BtnMenu Object 的图标。接受提出的新控件类名 Cpmenu，然后关闭 Gallery。

步骤 2：向 CButtonView 类添加控件

我们的目的是在应用程序的主窗口视图内放置 Button Menu 控件，因此，开始在 Button View 源文件中进行编码。首先，向位于 ButtonView.h 头文件的 CButtonView 类声明中添加 Cpmenu 对象：


```
#include "pmenu.h"
```

```
#define IDC_BTNMENU 1001
```

```
class CButtonView : public CView
```

```
{
```

```
private:
```

```
    Cpmenu    btnmenu;
```

```
    .
```

```
    .
```

```
    .
```

```
}
```

下一步是编写初始化 btnmenu 对象的代码。这项工作最好在 CButtonView::OnInitialUpdate 函数中完成，以确保该应用程序仅仅在视图刚出现的时候创建控件。ClassWiard 可以生成该函数的初始代码；只需在 ClassWizard

的 Message Maps 选项卡中的 Class Name 框中选中 CButtonView，然后双击 Messages 框中的 OnInitialUpdate 即可。通过 Edit Code 按钮退出 ClassWizard，该操作将在文本编辑器中自动打开 ButtonView.cpp 源文档，并将插字符定位在新 OnInitialUpdate 函数。向该函数添加如下所示的初始化代码：

```
void CButtonView::OnInitialUpdate()
```

```
{
```

```
    CView::OnInitialUpdate();
```

```
    // TODO: Add your specialized code here and/or call the base class
```

```
    COleVariant v( 1L );
```

```
    CRect rect( 30, 30, 250, 120 );
```

```
    btnmenu.Create( NULL, WS_VISIBLE, rect, this, IDC_BTNMENU );
```

```
    btnmenu.SetCaption( "Click Here" );
```

```
    btnmenu.Invalidate();
```

```
    btnmenu.AddItem( "Menu item #1", v );
```

```
v = 2L;  
  
btnmenu.AddItem( "Menu item #2", v );  
  
v = 3L;  
  
btnmenu.AddItem( "Menu item #3", v );  
  
v = 4L;  
  
btnmenu.AddItem( "Menu item #4", v );  
  
}
```

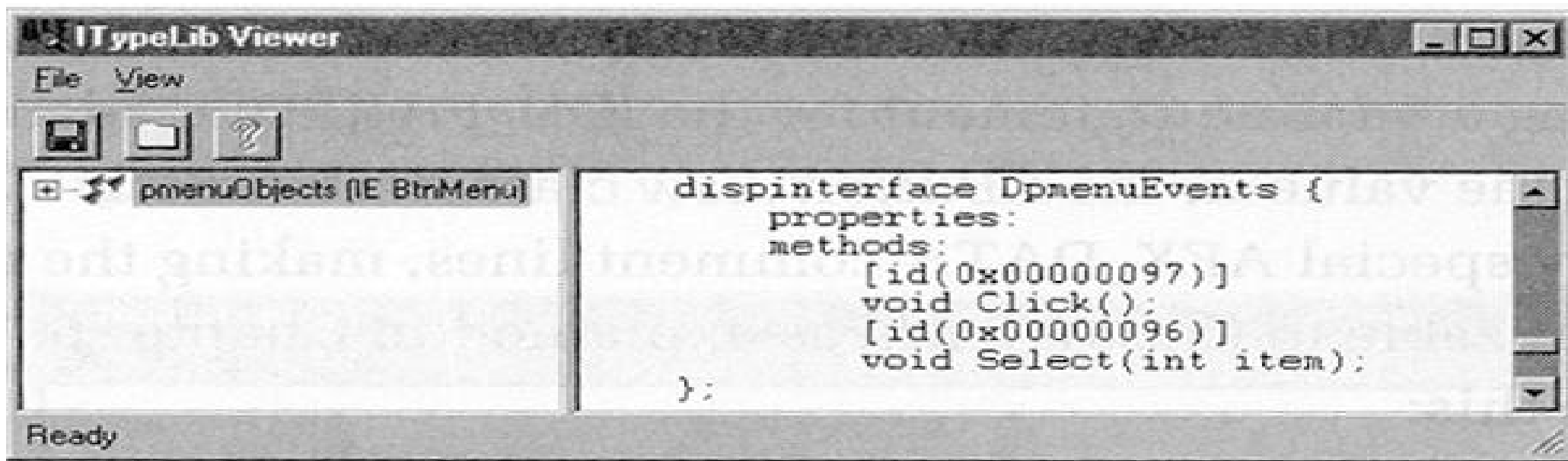
Cpmenu 是从 Cwnd 派生出来的，它提供了 Create 函数的两个版本，这两个版本在 Pmenu.h 文件中都有原型。出于简化的考虑，这里所列出的代码段使用了 CRect 对象将该控件在主窗口中的尺寸和位置固定下来。Cpmenu::AddItem 函数用于向该按钮的弹出式菜单添加命令字符串，根据函数的第二个参数所给定的 VARIANT 值，来组织菜单中的命令。本例中的 OnInitialUpdate 函数，只是为了保持 VARINT 值而创建一个 ColeVariant 对象，以及为了插入一个有代表性的菜单命令列表而四次调用 AddItems。

如果你这时编译并运行 Button 应用程序，那么，它将在主窗口中正确地显示

Button Menu 控件。尽管单击该控件可以调用它的弹出式菜单，但是，当选中并单击菜单中的命令的时候，应用程序本身并不能做出响应。这是因为我们还没有添加函数来处理 Button Menu 控件在用户交互期间触发的事件。那是下一步的工作。

步骤 3：处理事件

我们现在仍然不知道 Button Menu 控件触发的是什么事​​件。该信息被作为类型库资源的一部分存储在控件的 OCX 文件之中。但是，ClassWizard 却不能读取这个数据，原因是它对控件或 Cpmenu 类一无所知。Visual C++ 提供了一个实用程序，你可以应用它来探究一个控件的类型库，来了解该控件可以触发的事件，以及处理程序函数所需要的参数列表。该程序的名字叫 OleView.exe，是通过单击 Tools 菜单上的 OLE/COM Object Viewer 调用的。利用 Object Viewer 的 File 菜单上的 View TypeLib 命令，你可以打开 BtnMenu.ocx 文件，并显示它的类型库。它表明 Button Menu 仅能触发两个事件，分别叫做 Click 和 Select。库脚本显示了包容器应用程序如何必须声明处理程序函数，以便正确地接收事件触发：



这是我们为事件编写处理程序函数所需要的所有信息。但是对于提供了许多不同事件的控件来说，浏览类型库还有更容易的方法，即将该控件的存在简单地通知给 ClassWizard。然后，ClassWizard 可以承担从控件读取类型信息的所有工作，为存根处理程序函数生成代码，并向事件接收端映射添加它们的条目。所需要的仅仅是项目类数据库的一点消息，以及向 CButtonView 类声明添加一点儿东西。步骤如下：

首先，在文本编辑器中打开 Button.clw 文件，并为一个对话框添加新的资源条目：

ResourceCount=3

```
Resource1=IDR_MAINFRAME
```

```
Resource2=IDD_ABOUTBOX
```

```
Resource3=IDD_FAKEDLG
```

在第一行中一定要把资源数目增加到 3。在文件的底部添加新对话资源的一个声明。

```
[DLG:IDD_FAKEDLG]
```

```
Type=1
```

```
Class=CButtonView
```

```
ControlCount=1
```

```
Control1=IDC_BTNMENU,{52dfae60-cebf-11cf-a3a9-00a0c9034920},1342177280
```

在项目中没有这样的对话(及标识符名),但是,ClassWizard 并不需要知道这一点。该条目将告诉 ClassWizard,一个标识为 IDD_FAKEDLG 的对话资源属于 CButtonView 类;并且包含有由 CLSID 号进行标识的 Button Menu 控件。

下一步，返回到 ButtonView.h 文件，并修正我们以前添加的代码，为 IDD_FAKE DLG 标识符添上 #define 语句，并在 CButtonView 类声明中声明该值。新条目是用特殊的 AFX_DATA 注释进行标记的，使得对话框标识符对于第 6 章中解释的标识符来说是可以识别的。结果应该与下面类似：

```
#include "pmenu.h"
```

```
#define IDC_BTNMENU 1001
```

```
#define IDD_FAKE DLG 1002
```

```
class CButtonView : public CView
```

```
{
```

```
private:
```

```
    Cpmenu    btnmenu;
```

```
    //{AFX_DATA(CButtonView)
```

```
    enum { IDD = IDD_FAKE DLG };
```

```
//}}AFX_DATA
```

保存该文件。现在,当你调用 ClassWizard 的时候,如果 ClassName 中 CButtonView 类被选中的话, IDC_BTNMENU 标识符将在 Message Maps 选项卡中对象列表的底部显示出来。选中 IDC_BTNMENU 条目来显示 Messages 框中的两个事件,然后双击 Select 事件来添加名为 OnSelectBtnmenu 的处理程序函数。无论何时用户单击了控件弹出式菜单中的一个命令,该处理程序函数前将赢得控制权。

在 Member Functions(成员函数)框中选定了新处理程序函数之后,单击 Edit Code (编辑代码)按钮,来重新打开 ButtonView.cpp 文件。将代码添加到 OnSelectBtnmenu 函数,无论何时,该控件触发了它的 Select 事件,该函数都将做出反应:

```
void CButtonView::OnSelectBtnmenu(long item)
```

```
{
```

```
    // TODO: Add your control notification handler code here
```

```
    CString str;
```



```
str.Format( "You selected item #%1i\t", item );  
  
MessageBox( str, "Button application", MB_ICONINFORMATION );  
  
}
```

创建 Button 程序并运行它。每次你单击如图 8-11 所示控件菜单中的一个项目时，你都应该能看到一个消息框。

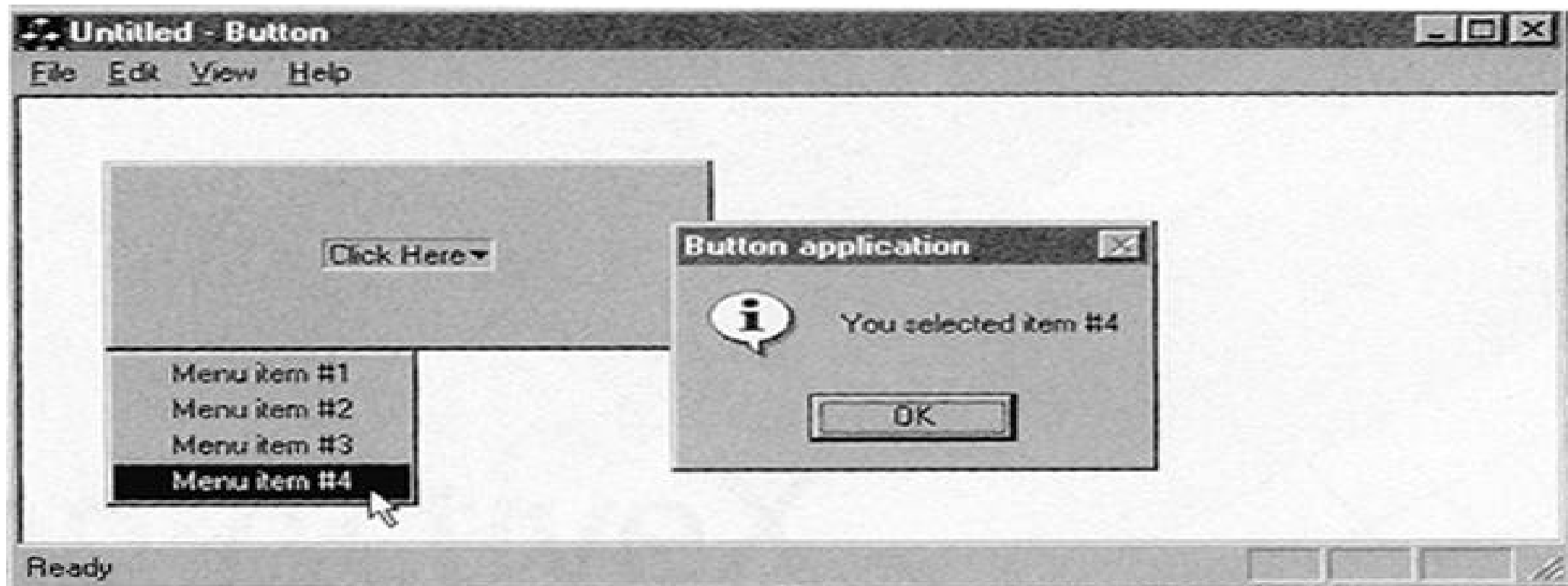


图 8-11 在框架窗口中放置的 ActiveX 控件

第 9 章 使用 MFC 编写 ActiveX 控件

第 8 章演示了在使用 MFC 的情况下，创建 ActiveX 控件并不特别需要理解 OLE 和 COM 基础结构。很显然，在编写 ActiveX 控件时亦是如此。MFC 可以处理诸多的细节，你不必关心它太多复杂的基础信息，便可以编写控件。如果你决定不用 MFC 来编写自己的 ActiveX 控件(这个想法的理由很充足)，那么，你对项目的实现就更加有信心了。根据你的方法和控件的复杂性，你可能需要仔细地了解 ActiveX 和 Component Object Model 的原理。

本章继续前一章的内容。它从服务器而不是客户的角度来检查 ActiveX 控件，介绍了 Visual C++ 帮助想编写而不只是使用 ActiveX 控件的开发者的一些方法。Visual C++ 提供了三个不同的工具来帮助建立 ActiveX 控件项目：

- 对 ActiveX 控件的 MFC 支持。
- BaseCtl 框架。
- Active Template Library。

用于创建 ActiveX 控件的 Visual C++ 工具

MFC 提供了创建 ActiveX 控件的最容易的方法。以前在 Microsoft Developer Network 上提供的 Control Development kit 已经作为一套工具并入 Visual C++ 中，该套工具包括已经重命名了的 MFC Active Control Wizard：Test Container 和 Control Wizard。正如本章稍后用例子所演示的那样，Control Wizard 将生成包含有原始代码的源文件，该代码使用 MFC 来管理几乎所有的 COM 详细资料。生成的源代码可以处理串行方式，显示控件的属性表，并为程序员和用户提供其他许多方便之处。你仅仅需要添加必要的代码来画出该控件，对用户的输入做出反应和触发事件。

使用 MFC 编写 ActiveX 控件可以使 OCX 文件相当小，但是，这是一种误导，因为该控件永远依靠两个大文件的存在而存在。这是 MFC 的缺点。如果控件的 OCX 文件在用户的计算机上不可用，该用户的浏览器程序必须下载这个文件，才能显示它，因此，小的可执行文件尺寸对于用于 Web 页的控件来说就特别重要。一个 ActiveX 控件不能与 MFC 静态链接起来，这就意味着，如果用户硬盘上不存在 MFC 库 DLL 文件的正确版本，那么该文件必须随控件一起传输。更为糟糕的是，MFC 需要 C 运行时间库，因此 Msrcrt.dll 文件也可能需要下载。当用户首先遇到一个显示依靠 MFC 的 ActiveX 控件的 Web 页的时候，将自动传输库文件。尽管对于用户通过快速网络连向的内部 Web 站点来说，这是很合理的，但是对于 Internet 来说却是不现实的。既然 MFC 的大小大约为 1 兆字节，因此，使通

过快速快速调制解调器下载该文件，也要花上几分钟的时间。

BaseCtrl 框架也就是人们所熟知的 ActiveX Controls 框架，是 MFC 的一个简便的替代方案。尽管 BaseCtrl 为开发者提供的支持比 MFC 少得多，但它却提供了更大的灵活性。从 BaseCtrl 中创建的 ActiveX 控件既不需要 MFC，也不需要 C 运行时间库。因为框架仅仅提供了最少的代码，因此该控件比它的 MFC 同伴占用更少的内存。但是 BaseCtrl 的优点是要付出代价的，使用 BaseCtrl 需要对你的组件做更多的工作，并且要对 COM 和 ActiveX 的原理有更深入的理解。例如，你必须十分适应持久的接口，例如 Istream，IPersistPropertyBag 和 IPersistStream。框架通过三个主类提供了核心功能，这三个主类的名字分别为 CAutomationObject，ColeControl 和 CPropertyPage。

BaseCtrl 过去常常需要 Visual Basic 4.0 来开始一个项目，但现在不需要了。你现在可以运行一个由 Visual C++ 提供的 NMake 实用程序来开始一个项目，第一次生成库文件，再次为新类生成存根源文件。在 BaseCtrl 提供的 ReadMe.txt 文件中解释了这个过程。在安装 Visual C++ 时，如果你申请包括范例代码，那么 ReadMe.txt 文件和几个范例 BaseCtrl 项目的源代码将位于文件夹 MSDN\Samples\VC98\SDK\COM\AativeXC\BaseCtrl 之中。如果想得到 BaseCtrl 的最近更新版本，请从如下站点下载 ActiveX 开发工具箱：

<http://www.microsoft.com/intdev/sdk>

目前并不常使用 BaseCtrl，原因是 Active Template Library(ATL)提供了一种创建小

ActiveX 控件的高级工具。尽管结果 OCX 文件比依赖于 MFC 的类似控件要大，但通过 ATL 创建的控件通常占用更少的内存，原因是它不需要其他辅助文件的存在。除了为你实现许多标准接口的智能模板代码库之外，ATL 还提供了一个可以生成初始源文件的向导，这就大大简化了 ActiveX 控件项目的早期阶段。与用 ControlWizard 和 MFC 创建 ActiveX 控件相比，用 ATL 创建相同的控件需要更多的工作，但对于创建用 Internet 上使用的 ActiveX 控件来讲，ATL 是一个很好的选择。

你可能听说过 ATL 不适合于编写 ActiveX 控件，但自从该库的 2.0 发行之后，这种说法就不再正确了。不过，ATL 不是一个简单的主题，直到下一章才能涉及到更加详细的讨论。本章将介绍创建 ActiveX 控件的 MFC 方法。MFC 提供了试验 ActiveX 编程的最好方法，并提供了充足的信息。除了可以为你节省大量的编码之外，使用 MFC 编写 ActiveX 控件，还可以提供许多有帮助的 Visual C++ 特征，例如 ClassWizard 和 ControlWizard。正如我们将要在下面看到的那样，ControlWizard 为 ActiveX 控件项目提供了一个很好的起点。

ControlWizard

应用与 AppWizard 为 MFC 应用程序创建项目相同的方法，ControlWizard 可以为 ActiveX 控件创建一个项目。在 ControlWizard 帮助下创建的控件使用 MFC，因此也就具有前一节所介绍的优点和缺点。ControlWizard 是 AppWizard 的自定义的形式，ControlWizard 项目以和普通 AppWizard 项目相同的方式开始。从 File

菜单中选择 New 命令打开 New 对话,在 Projects 选项卡中单击代表 MFC ActiveX ControlWizard 的图标。图 9-1 说明了这些步骤。

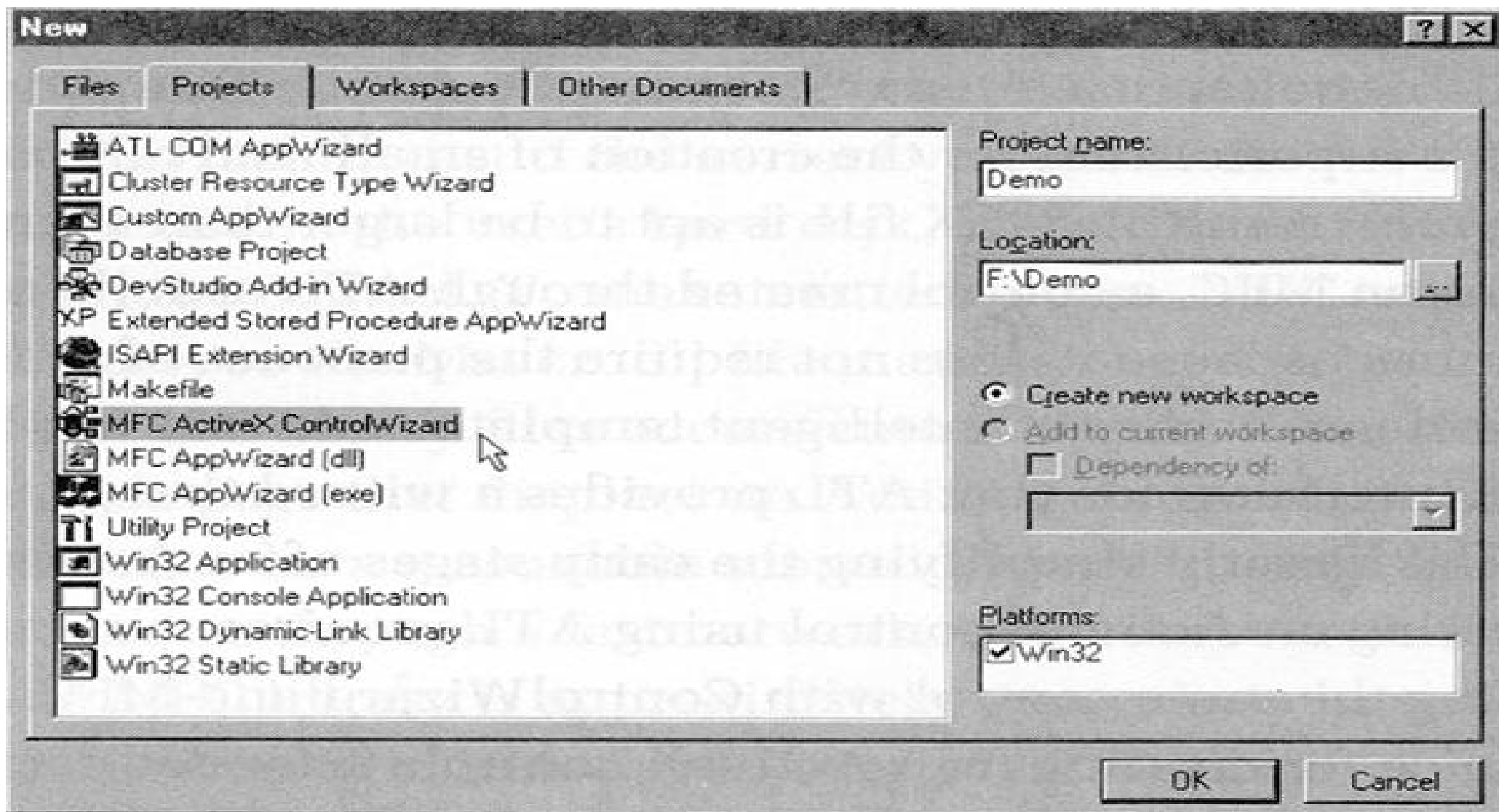


图 9-1 用 ControlWizard 开始一个 ActiveX 控件项目

在创建项目之前，ControlWizard 将引导完成两个步骤。本节将检查这个向导所提供的选项，并讨论一个选项何时对于你的 ActiveX 控件来说是合适的，以及它为什么合适。

图 9-2 显示了 ControlWizard 的打开屏幕，它首先询问你项目中控件的数量。与 VBX 自定义的控件类似，一个 OCX 文件可以包含不止一个 ActiveX 控件组件。请在对话顶部的文本框中指定你想要的控件数量。以后，在项目开发期间，你还可以添加控件。使用 ControlWizard 屏上的下一个选项，可以通过许可来限制控件的使用。尽管许可支持选项在默认状态下是关闭的，对于一般用途的许多控件，应该具有简短描述的许可保护。

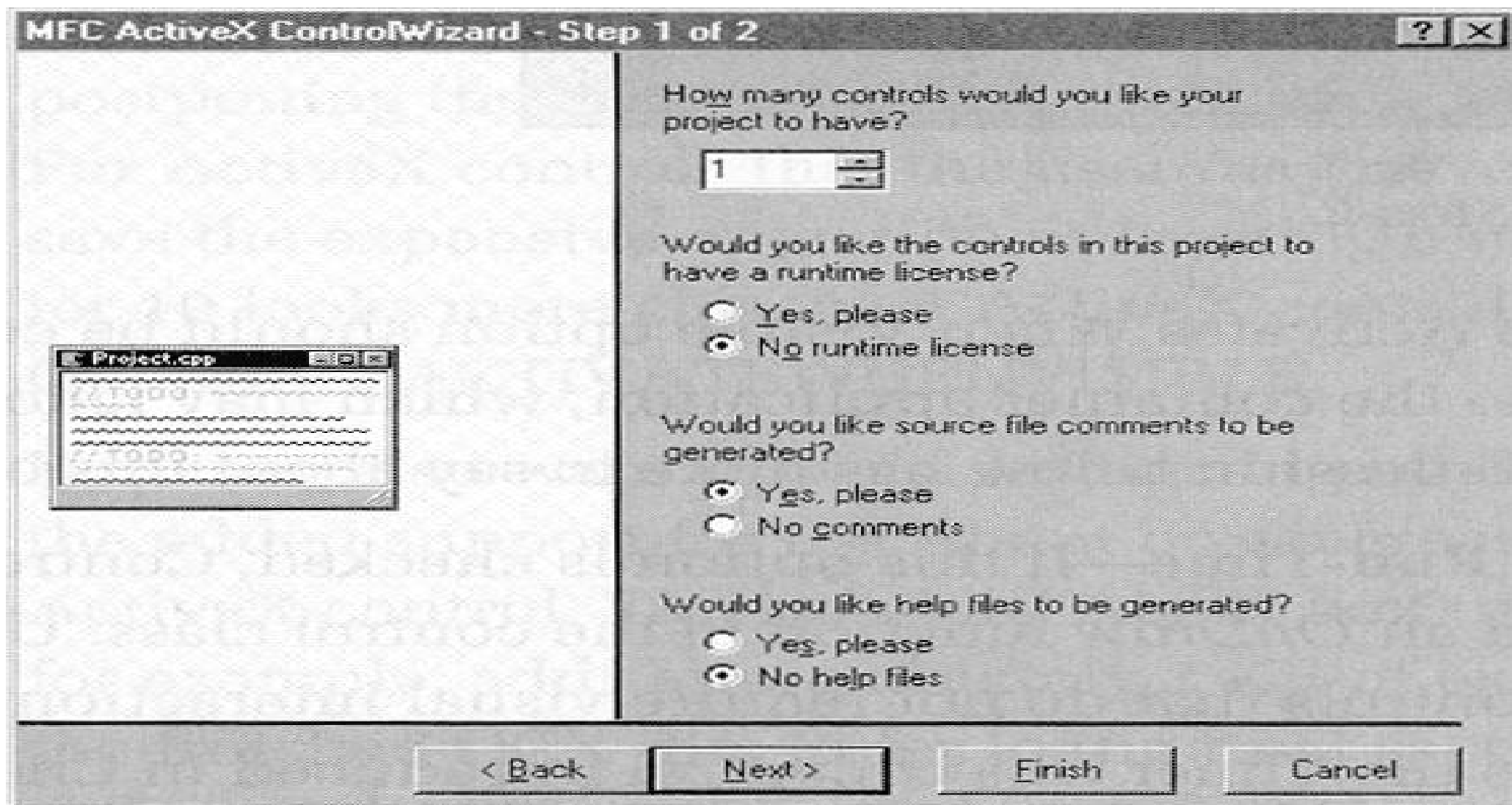


图 9-2 ControlWizard 的步骤 1

步骤 1 中的最后一个选项用于指导 ControlWizard 为控件生成帮助文件。对项目帮助文件的申请，所添加的帮助支持种类与从 AppWizard 中得到的相同。如果想获得所生成的帮助文件的说明，请参阅第 2 章开始部分的讨论。

ControlWizard 的第二屏(图 9-3)展现了决定控件如何与包容器进行交互的选项。第二步中的选项需要解释一下，因此，下面详细对它们说明。

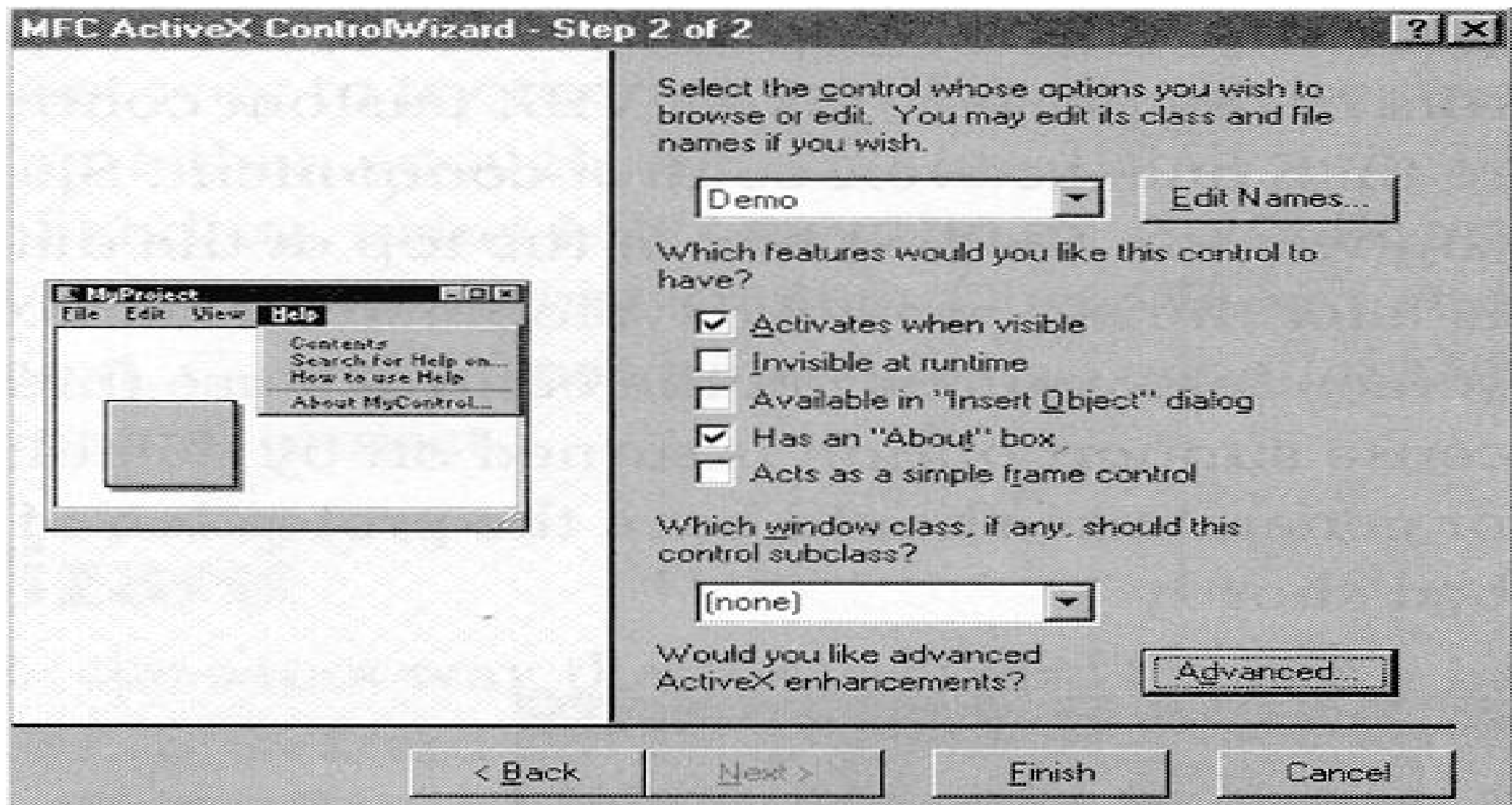


图 9-3 ControlWizard 的步骤 2

- **Activates when Visible (可见时激活)**：决定当控件可见时，容器是否应自动激活它。对于 ActiveX 控件，通常需要立即激活，尽管选中 Activates When Visible 选项仅仅作作为容器应用程序的线索，它可能忽略申请。如下的讨论有很多是与此选项有关的。
- **Invisible At Run-Time (运行时不可见)**：如果选中了该选项，ControlWizard 将不把 OnDraw 函数添向控件类。可以为那些不需要与用户进行交互的控件使用该选项，例如第 8 章中介绍的 Timer Object 控件。
- **Available In Insert Object Dialog (在插入对象对话框中可用)**：与从 Test Container 工具到 Microsoft Office 应用程序的许多容器应用程序提供的 Insert Object 对话框(或等价物)相关。不选中该选项，将通知容器，它不应该在容器的 Insert Object 对话框中包括这个新控件。
- **Has An About Box (有一个 About 框)**：ControlWizard 将为 About 方法生成源代码，为常用 About 框生成资源。
- **Acts As A Simple Frame Control (作为一个简单的框架控件)**：为 IsimpleFrame- Site 接口添加支持。该选项将设置控件，来充当框架，用来包围容器窗口中其他 ActiveX 控件，将控件分组，并允许它们一起移动。并不是所有的容器都支持简单的框架。

- **Window Subclassing (窗口细分类)**: 通过对诸如编辑框或进程指示器之类的普通 Windows 控件进行细分类, 来建立控件项目。

在默认状态下, ControlWizard 仅仅打开上面列出的两个选项 `Activates when Visible` 和 `Has An About Box`。取消选中 `Activate when Visible` 复选框, 将清除掉该控件的 `OLEMISC_ACTIVATEWHENVISIBLE` 状态标志, 该标志是控件通过 MFC 的 `AfxOLERegisterControlClass` 全局函数放置在它的注册表数据之中的。嵌入控件的容器可以通过调用控件的 `IOleObject::GetMiscStatus` 方法来确定该标志的状态。清楚的标志将向容器传递信号, 当控件可见时, 它应该保持非激活状态, 这样, 将延迟对控件窗口的创建, 直至用户需要它。对于 ActiveX 控件, 这将节省创建窗口这一不必要的操作。第 10 章将详细讨论 ActiveX 控件状态标志, 例如 `OLEMISC_ACTIVATEWHUENVISIBLE`。

你可能还想考虑关闭 `About Box` 选项, 因为对 `About` 框添加的支持将增大完成之后的 ActiveX 控件的容量。例如由 ControlWizard 生成的标准 `About` 框将向完成之后的 OCX 文件添加大约 2KB 的额外代码和资源文件。如果想了解有关最小化资源数据的其他方法, 请参考第 4 章的最后一节, 这对 ActiveX 控件来说是特别重要的。

如果选中了 `Available In Insert Object Dialog` 复选框, 那么, 该控件的自我注册过程将向控件的 CLSID 注册表层次添加名为 `Insertable` 的 Registry 键。`Insertable` 键将通知容器 ActiveX 控件可以充当被动嵌入的对象。这样, 容器就能够通过

OLE Documents 接口创建 ActiveX 控件的对象。这些用 IOle 前缀标识的接口包括 IOleCache, IOleClientSite, IOleContainer, IOleInplaceObject 和 IOleInPlaceSite。可以在包容器文档中嵌入对象的应用程序为寻找具有 Insertable 键的对象而扫描注册表，并在标准的 Object 对话框中列出了一个对象列表。例如，为了在 Microsoft Word 中看到这个列表，请单击 Word 的 Insert 菜单上的 Object 命令。仅仅当 Available In Insert Object Dialog 复选框打开的时候，使用 ControlWizard 创建的 ActiveX 控件才能出现在列表之中。不支持包容器文档的应用程序将忽略 Insertable 键。例如，我们在第 8 章中所遇到的 Test Container 实用程序中的 New Control，将显示一个所有注册控件的列表，而不管它是否标记为可插入的。

如果你想让 ActiveX 控件对标准或普通 Windows 控件进行细分，那么请单击图 9-3 中屏幕底部显示的框。该下拉窗口将显示一个列表，其范围从按钮到树状视图的 16 个 Windows 控件。从该列表中选中一条目，将导致 ControlWizard 为对选中的 Windows 控件进行细分的 ActiveX 控件生成源代码。使用该选项可以生成只有某个 Windows 控件的特性的 ActiveX 控件，但是，为了添加所需要的效果，你可能还要进行修改。

在 ControlWizard 的步骤 2 中单击 Advanced 按钮，来打开如图 9-4 所示的 Advanced ActiveX Features 对话框。该对话框提供了用来设置或清除由 ColeControl::ControlFlags 枚举集定义的位标志，该枚举集描述了控件被激活时的行为特性。设置任何复选框都将导致 ControlWizard 添加超越 ColeControl::GetControlFlags 方法的代码，该方法的作用是将 ControlFlags 设置通知给包容器。为了在现有的控件项目中添加

超越代码，请设置特定的位标志，例如下面的 WindowlessActivate:

```
DWORD CDemoCtrl::GetControlFlags()

{

    return COleControl::GetControlFlags() | windowlessActivate;

}
```

在 Advanced ActiveX Features 对话框中的复选框标签看上去很简洁，但是，略加解释之后是可以很容易理解它的。下面的列表说明了标志的含义。如果想了解有关 ControlFlags 设置，以及它们是如何影响到控件动作的更为详细的信息，请查阅题目为“ActiveX Controls::Optimization”的帮助文章，通过索引条目 Optimizing ActiveX Controls 可以找到它。

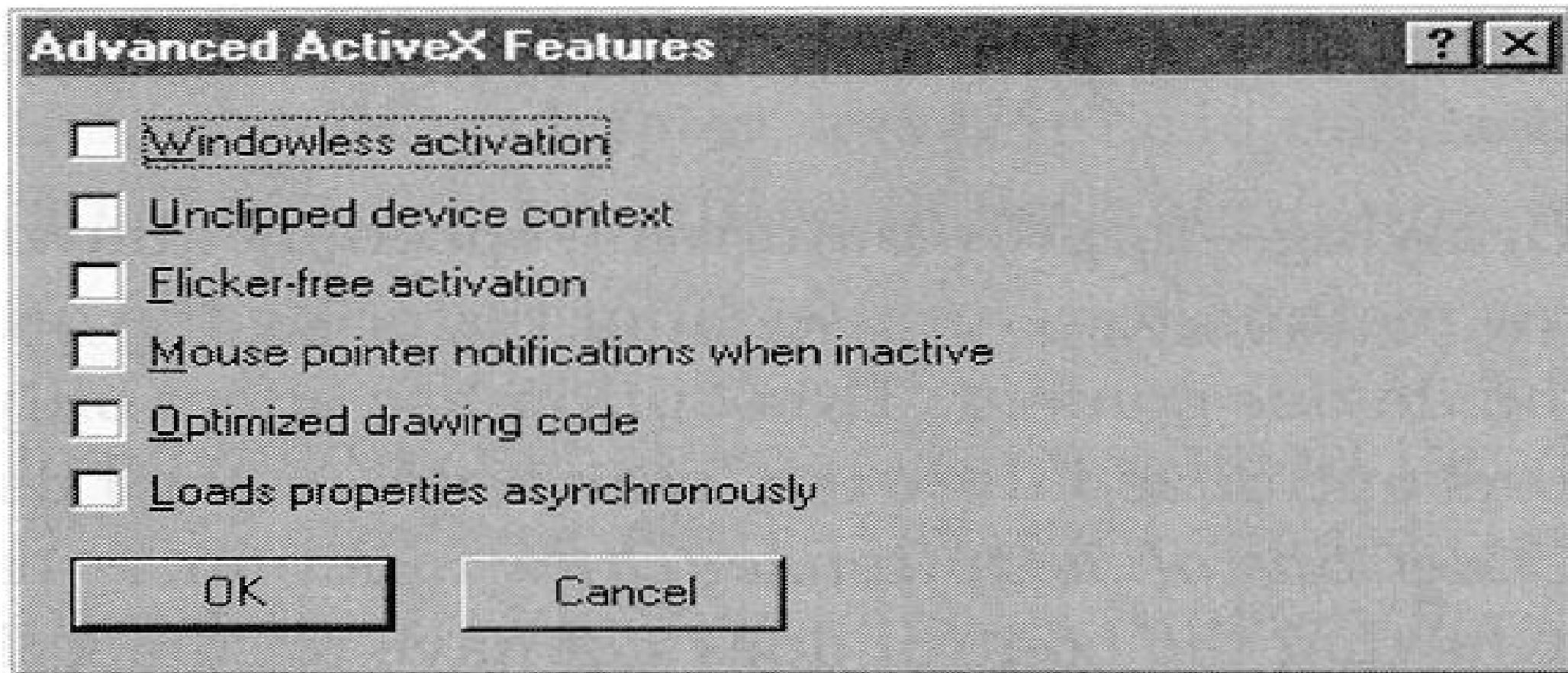


图 9-4 通过单击 ControlWizard 的 Advanced 按钮调用 Advanced ActiveX Features 对话

- **Windowless Activation(无窗口激活)**:通知包容器,控件在激活时不创建自己的窗口。下面的讨论有许多是关于无窗口激活的。
- **Unclipped Device Context (不剪辑设备上下文)**:申请不要剪辑控件

的显示，这样处理会更快。不过，控件必须确保不在站点边界之外显示。

- **Flicker-Free Activation (激活时不闪烁)**：要求包容器要在控件切换状态时不要使控件窗口无效。该操作将阻止控件在成为活动或不活动的时候重绘自己，这样也就清除了可能发生的轻微的闪烁。该选项仅仅适合于不管状态如何而以同一种方式绘制自己的控件。
- **Mouse Pointer Notifications (鼠标指针通知)**：要求包容器在控件不处于活动状态对 ActiveX 控件发送鼠标消息。如果包容器满足了要求，那么，非活动的控件将继续接收与控件窗口上鼠标活动有关的 WM_SETCURSOR 和 WM_MOUSEMOVE 消息。选中该选项将启用 IPointerInactive 接口，包容器将把这些属于控件的消息委托给这个接口。IPointerInactive 接口为控件窗口调整每条消息的鼠标坐标，并且通过控件的消息映射分派消息。通过这项功能，控件甚至可以在非活动状态实现拖放目标的功能。
- **Optimized Drawing (优化的绘制)**：通过允许控件的 OnDraw 方法在返回时不恢复用于设备上下文的原始 GDI 对象，来提高绘制的速度。仅仅当包容器支持优化的绘制时，该选项才起作用，而包容器是否支持优化的绘制，由控件通过调用 COleControl::IsOptimizedDraw 函数来确定。返回值 TRUE 意味着该控件不必在完成绘制之后不必选中诸如

钢笔和刷子之类的原始 GDI 对象，便可回到设备上下文之中。

- **Load Properties Asynchronously（异步加载属性）**：该选项可以提高 ActiveX 控件的响应性，而且这种控件需要大量属性数据。异步加载将允许控件在 Web 页上尽可能地成为活动状态，即使在浏览器在后台连续不断地通过调制解调器加载该控件的数据时也是如此。例如，该控件这样便可以立即播放音频或视频片断，而无需等待完整的数据集。不过，控件必须不对尚未到达的数据进行操作。异步加载为控件增添了负担，因此，应该仅仅为可以从中受益的控件使用该选项。

Windowless Activation 选项与前面介绍的 Invisible At Run-Time 标记不是同一件事情。无窗口激活仅仅意味着控件不提供它自己的窗口。通过不创建窗口，控件优化了创建的速度，同时稍微降低了一点儿可执行文件的大小。如果包容器支持无窗口对象的话，该控件可以自由使用包容器的窗口服务。这种支持需要 IOleInPlaceObjectWindowless 接口来将用户输入消息反应到无窗口控件。通过覆盖包容器的窗口，无窗口控件可以带有真正透明的背景，对于显示自己矩形窗口的普通 ActiveX 控件来说，达到这种效果是不可能的。不过，使用第 4 章中介绍的使背景透明的想法，一个窗口控件通常可以将它自己的窗口颜色与包容器的背景颜色相匹配，来模拟透明背景。正如在上一章中所提到的那样，控件可以通过调用 `COleControl::AmbientBackColor` 来确定包容器周围的当前颜色：

```
OLE_COLOR ContainerBkGrnd = AmbientBackColor();
```

并非所有的容器都支持 ColeControl 的 Ambient 函数。在调用诸如 AmbientBackColor 之类的函数之后。一个 ActiveX 控件应该检查有效返回值。

许可

放置在流行 Web 上的 Active 控件不久便可以在全世界的计算机上出现，在数以千计的浏览器上查看。这种简单重复使用 ActiveX 控件的能力可能是该项技术最吸引人的特征和最大的优点。不过，这样的话，便引起了有关程序员的知识产权的潜在问题。为了更清楚地认识这一问题，请考虑一下 ActiveX 控件是如何传过 Author、Webmaster 和 User 这迥然不同的三方面的。

收取一定的费用，Author 便允许 Webmaster 在 Web 页上安装 ActiveX 控件。User 通过 Internet 访问 Webmaster 的站点，结果，ActiveX 控件将从 Webmaster 的计算机复制到 User 的计算机；在 User 的浏览器程序中出现。到此为止，每件事情都如期进行，控件的使用像 Author 所期望的那样。但是，如果不采取一些安全措施的话，拥有这个 ActiveX 控件的其他程序员就可以在自己的应用程序中使用它。许多开发者不愿意在未经授权的情况下，他们的产品用这种方式被重用，特别是在可以得益的商品应用程序之中使用。

要防止未经授权使用 ActiveX 控件，最常用的措施是包含一个许可证。许可证不仅将我们的范例中的 Author 标识为作为控件版权的所有者，而且还可以阻止没

有从 Author 处收到许可证的开发者随便重用控件。OLE/ActiveX 控件标准的设计考虑到了许可证。该标准定义了 IClassFactory2 接口，包容器可以通过该接口来创建控件对象的一个实例，并证明它自己已获得了使用该控件的许可证。仅仅当包容器满足了有效许可证存在的控件的时候，才能完成控件对象的创建。

许可证已成为 ActiveX 控件普遍采用的方法，因此，详细检查一下 ControlWizard 的许可证方案还是十分值得的。在这个主题上花费时间的另一个原因是，Visual C++ 在线文档中的许可证说明不十分清楚，这主要是因为当可能涉及到几个包容器的时候，这个文档将说“the container（这个包容器）”。记住，任何程序都是可以创建 ActiveX 控件的一个实例，并为之提供站点的包容器，这一点是很重要的。第 8 章演示了可以在不同环境下嵌入 ActiveX 控件的几个不同的包容器：

- Internet Explorer 或另一个可察知 Active 的浏览器，它通过由 HTML 文档中的 OBJECT 标记指定的类标识符来定位控件。
- Visual C++ 对话编辑器，当它被放入开发环境下的一个对话中时，创建控件的一个实例。
- Hour 程序这样的包容器应用程序，它可以在运行时嵌入一个 ActiveX 控件。

许可证方案可以阻止包容器应用程序未经授权便使用控件，但是，阻止哪一个包容器呢？在我们的方案中，当 User 连同显示 Author 的控件的 HTML 指令一起下

载该控件的时候，User 的浏览器必须能够自由运行该控件，而不必使用许可证。访问应该仅仅限制到列表中的其他两种类型的包容器：也就是开发程序(像 Visual C++)和以及它们所创建的包容器应用程序(像 Hour)。

当 Webmaster 决定开发使用 Author 的 ActiveX 控件的时候。应该考虑一连串的事件。为了创建应用程序，Webmaster 运行一个 Windows 开发程序，例如 Visual C++ 或 Visual Basic。这个程序设计要求应用程序在对话框中显示控件，因此，Webmaster 使用对话编辑器本身就是包容器创建控件的一个实例，并在该对话中显示它。这个时候，叫做设计阶段，许可证问题便出现了。在创建控件实例的过程中，开发程序将调用该控件的 `IClassFactory2::CreateInstanceLic` 方法，其中的参数为 `NULL`，控件通过仅仅在确认许可证存在之后返回到接口的一个指针来响应该函数(过一会儿，我们将看到是如何响应的)。

既然 Webmaster 被授权可以在应用程序中使用该控件，那么许可证验证将获得成功，对话编辑器可以创建该控件的一个实例。Webmaster 完成应用程序的开发，并将可执行文件的副本出售给 User。作为软件包的一部分，Webmaster 提供一个安装程序，用来在 User 的硬盘上放置 Author 的 ActiveX 控件的副本，并注册它。尽管 User 从来没有与 Author 一起签订过许可证协议，新的包容器应用程序将在运行时成功地创建控件对象的一个实例(这个处理过程过一会再解释)。这叫做许可证验证的运行阶段。

现在，让我们来考虑一下当 User(恰好又是程序员)试图创建嵌有 Author 控件的另

一个包容器应用程序的时候，会有什么事情像以前一样发生。User 的开发程序调用控件的 `IClassFactory2::CreateInstanceLic` 方法，但这次，控件检测到 User 没有拥有许可证，因此也就不允许这一创建企图。有决心的 User 可以在没有对话编辑器的帮助下开发应用程序，但是，完成之后的应用程序不再能够创建该控件的一个实例。未经授权的控件中的许可证验证代码既在设计时使用，又在运行时使用。

ControlWizard 添向控件项目的代码可以实现类似刚才介绍的许可证方案。下一节将解释该方案是如何工作的。

ControlWizard 许可证支持

如图 9-2 所示，ControlWizard 的初始屏幕提供了对许可证协议的支持。如果你为新控件要求运行时间许可证，那么，ControlWizard 将生成额外的源代码和文本文件，它们一起可以保证使你的控件将只被经过授权的人使用。文本文件是带有 LIC 扩展名的文档，包含如下文字。

Copyright (c) 1998 author

Warning: This product is licensed to you pursuant to the terms of the

license agreement included with the original software, and is
protected by copyright law and international treaties. Unauthorized
reproduction or distribution may result in severe civil and criminal
penalties, and will be prosecuted to the maximum extent possible under
the law.

在许可证的第一行中，单词 `author` 代表你的用户名或公司名。像这样一个由 `ControlWizard` 实现的基于文件的许可证方案，当包容器应用程序在设计时添加该控件的时候，需要 `LIC` 文档与控件 `OCX` 文件存在于同一目录之中。由于这个原因，`Author` 必须向 `Webmaster` 分发这个 `LIC` 文本文件，以便 `Webmaster` 可以创建使用该控件的包容器应用程序。但许可证的期限阻止了 `Webmaster` 将该文档重新分发给别人。用户不需要 `LIC` 文件便可运行 `Webmaster` 的应用程序，并或在浏览 `ControlWizard` 在主项目文件夹中放置 `LIC` 文件的主副本。当 `Visual C++` 创建这个 `OCX` 文件的时候，它将这个 `LIC` 文件从主文件夹复制到 `OCX` 文件驻留的 `Release` 或 `Debug` 文件夹，因此，最终一个项目可能有两三个许可证文件的副本。在创建 `OCX` 文件之前，应当制订文件主副本的更改方案，以便所有的副本保持最新。

`ControlWizard` 添加到项目的源代码由两个函数组成，一个叫做 `GetLicenseKey`,

它可以从该控件的 OCX 文件中检索到一个独一无二的密码或关键字，另一个函数叫做 VerifyUserLicense，它为许可证文本文件的存在检查用户磁盘的指定位置。一个名为 License 什么都不做的项目，演示了 ControlWizard 是如何把这两个函数添加到控件的类源代码之中的。你没有必要自己创建 License 项目，因为不在这这里开发控件。它的作用仅仅是使得如下讨论更加容易，同时用来阐述 ControlWizard 的许可证方案。

下面是这两个函数的源代码清单，它取自该项目的 LicenseCtl.cpp 实现文件(第三行上的 _s2LicString 文字根据系统不同而有所不同)。

```
static const TCHAR BASED_CODE _szLicFileName[] = _T("License.lic");
```

```
static const WCHAR BASED_CODE _szLicString[] =
```

```
    L"Copyright (c) 1998 Witzend Software";
```

```
////////////////////////////////////
```

```
// CLicenseCtrl::CLicenseCtrlFactory::VerifyUserLicense -
```

```
// Checks for existence of a user license
```

```
BOOL CLicenseCtrl::CLicenseCtrlFactory::VerifyUserLicense()
```

```
{
```

```
    return AfxVerifyLicFile(AfxGetInstanceHandle(), _szLicFileName,
```

```
        _szLicString);
```

```
}
```

```
////////////////////////////////////
```

```
// CLicenseCtrl::CLicenseCtrlFactory::GetLicenseKey -
```

```
// Returns a runtime licensing key
```

```
BOOL CLicenseCtrl::CLicenseCtrlFactory::GetLicenseKey(DWORD dwReserved,
```

```
    BSTR FAR* pbstrKey)
```

```
{
```

```
    if (pbstrKey == NULL)
```

```
        return FALSE;
```

```
    *pbstrKey = SysAllocString(_szLicString);
```

```
return (*pbstrKey != NULL);  
  
}
```

在设计时，当开发程序企图将一个 ActiveX 控件插入到包容器项目时，也就是当对话编辑器向正在开发中的对话添加控件的时候，VerifyUserLicense 和 GetLicenseKey 函数均被调用。当开发完成，而且包容器应用程序已经建立并执行的时候，它在运行时创建控件实例的企图，会导致了再调用该控件的 GetLicenseKey 函数。让我们分别考查一下这两种情形，首先看一看在设计时控件是如何为开发程序验证许可证的存在。

设计时间许可证验证

通过调用控件的 IClassFactory2::CreateInstanceLic 方法，开发程序事实上是在说：“如果有效许可证存在，就创建控件的一个新实例，并返回一个指向该实例接口的指针”。该控件的工作便是验证许可证的存在。当 CreateInstanceLic 被调用的时候，框架将调用路由到控件的 VerifyUserLicense 函数，该函数用来确认 LIC 许可证文件与控件 OCX 文件存在于同一目录之中，而且，该文件的第一行与 _szLicString 参数的内容相匹配。_szLicString 参数就是人们所熟知的许可证关键字。

如果 LIC 文件存在，并且包含正确的许可证关键字，那么，VerifyUserLicense 将返回值 TRUE，允许开发程序为正在开发之中的包容器应用程序创建 ActiveX 控件的实例。开发程序下一步调用控件的 IclassFactory2::RelestLickey 方法。这个调用以 GetLicenseKey 结束，该函数是 ControlWizard 向控件源代码添加的两个函数中的第二个。GetLicenseKey 将返回控件的 _szLicenseKey 关键字的一个副本，开发程序将它嵌入在包容器应用程序的可执行文件之中。当我们谈到运行时间许可证的时候，我们还将看到为什么包容器需要它自己的关键字副本。

如果控件没有在与 OCX 文件相同的目录中找到 LIC 文件，或者，如果该许可证文件的第一行已经被改变了，那么 VerifyUserLicense 将返回 FALSE，在这种情况下，开发程序将显示一条解释该问题的错误信息。例如，下面是当 LicenseLic 文件被改变或重命名之后，Visual C++ 是如何处理这种情形的(如果你想自己试一试，一定要在包含 OCX 文件的子文件夹中改变 LicenseLic 文件，因为在控件创建之后，改变项目文件夹中的主副本是没有任何效果的)。假定你正在开发一个叫做 DemoContainer 的包容器应用程序，并且希望把 License ActiveX 控件添加到 DemoContainer 的 About 框，请在对话编辑器中加载 About 框资源，并在工作区内右击来显示该编辑器的上下文菜单。从上下文菜单中选择 Insert ActiveX Control 命令，然后从图 9-5 所示的列表中选择 License Control。

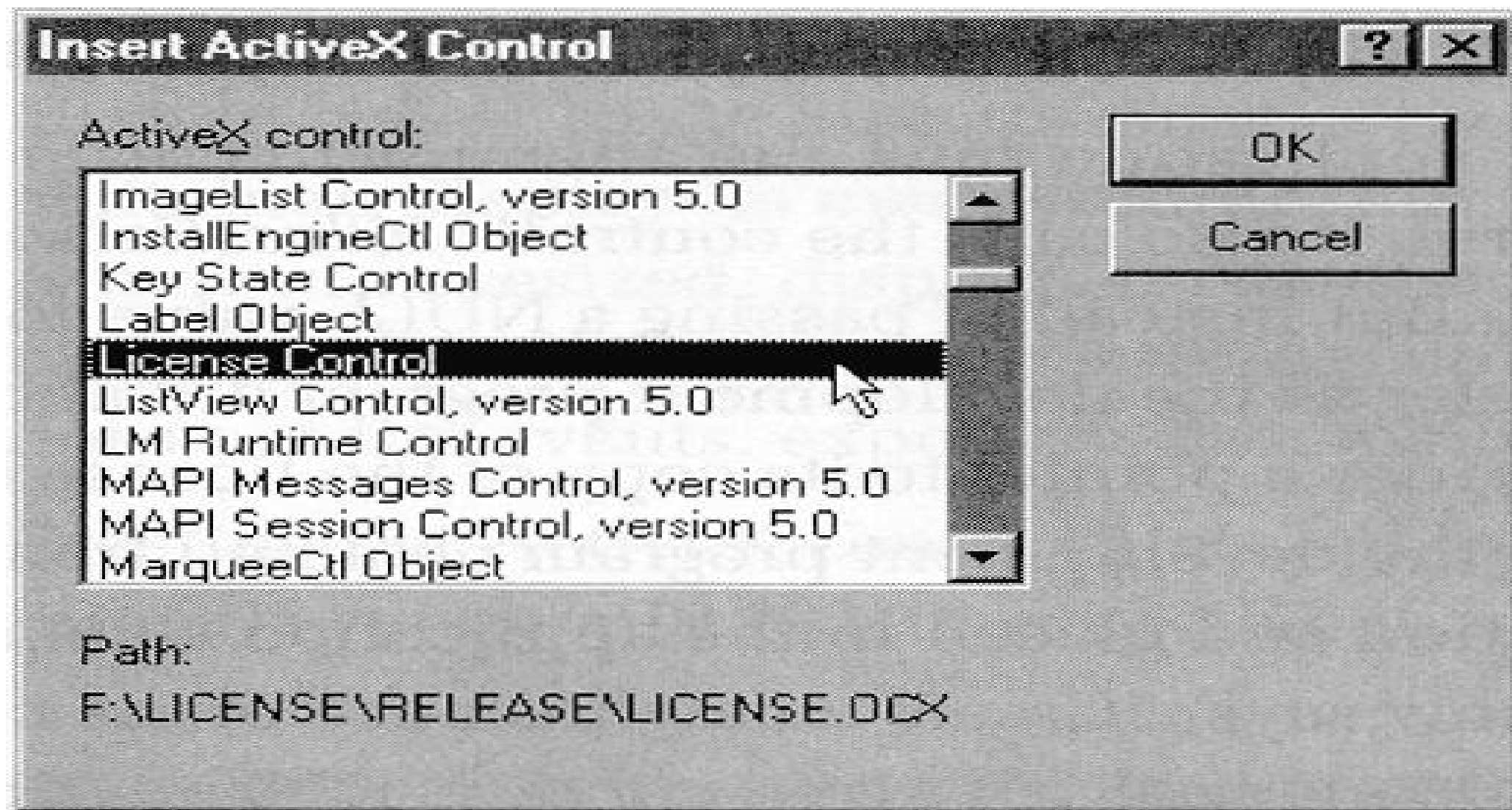


图 9-5 向容器项目插入 License 控件

对话框编辑器企图创建 License 控件的一个实例，这导致了一系列的函数调用。编辑器首先调用控件的 `IClassFactory2::CreateInstanceLic` 方法，该方法依次调用控件的 `VerifyUserLicense` 函数，该函数调用框架的 `AfxVerifyLicFile` 函数。这个 MFC 函数读取由 `szLicenseName` 字符串(这里是 `LicenseLic`)来标识的文件，如果该文件存在的话，将它的第一行与包含在 `_szLicString` 中的许可证关键字进行比较。如果文件不存在，或第一行与许可证关键字不匹配，`AfxVerifyLicFile` 将返回值 `FALSE`，来禁止对象创建。结果会出现 Visual C++ 的一条错误信息，来解释为何失败：



运行时间许可证

如果许可证文件存在，并且，VerifyUserLicense 函数返回了值 TRUE，那么开发程序将成功地创建控件实例，并将控件类的源文件插入到包容器项目之中。直到包容器应用程序被创建并成为可执行程序之后，才发生下一次有效许可证的验证。

当包容器应用程序运行，并试图创建 ActiveX 控件的一个实例的时候，它还调用该控件的 IClassFactory2:: CreateInstanceLic 方法。但是与开发程序在设计时间为第四个参数传递的 NULL 值不同，应用程序提供了一个指向它的许可证关键字副本的指针。这个字符串与开发程序从控件的 GetLicensekey 函数获得并放在应用程序的数据之中的字符中是相同的。对 CreateInstanceLic 的调用即：“创建控

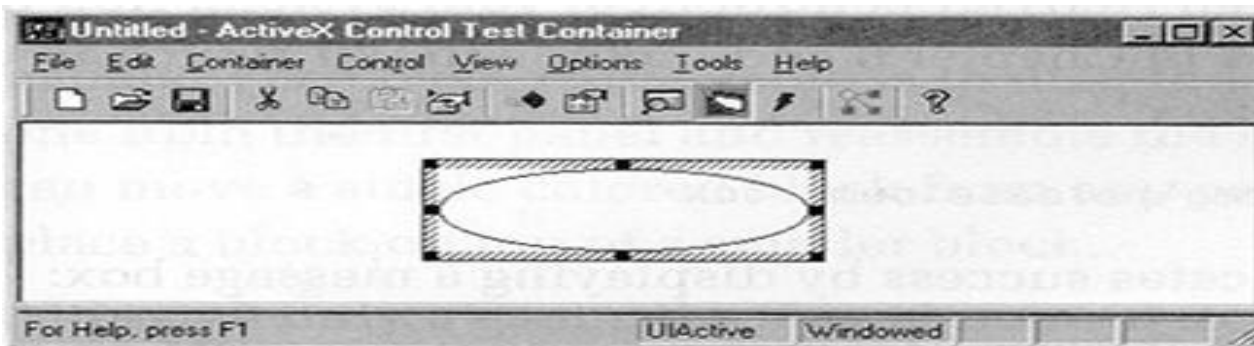
件的一个新实例，这里是证明我是一个经过授权的包容器的证据。”控件将 `_szlicString` 中它自己的许可证关键字副本与包容器递交的副本进行比较，验证匹配，并允许创建操作。在这种情况下，框架不调用 `VerifyUserLicense` 函数，这就是当应用程序在 `User` 的机器上运行时，为什么不需要 `LIC` 文件的原因。仅仅是 `Webmaster` 机器上的开发程序而不是完成后的包容器应用程序过需要许可证。

为了获得更多的保护，改变许可证 `LIC` 文件的第一行，使文字更具体些。但是要
对源代码中的 `_szLicString` 字符数组做出相同的改变，否则，`VerifyUserLicense` 将识别不了这个文本文件。你还可能考虑修改这里介绍的基于文件的许可证方案来依靠系统注册表中而不是 `LIC` 文本文件的关键字。这将需要为搜索注册表简单地重写一下 `VerifyUserLicense` 函数，而且，还要假定已经安装了程序，来为授权的许可证注册关键字。

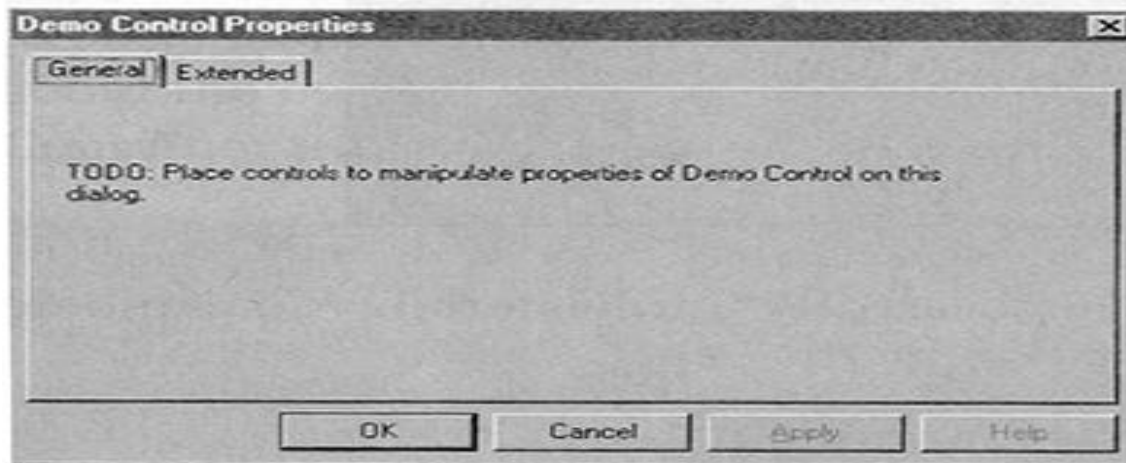
例 1：什么都不做的 ActiveX 控件

在开始 `ActiveX` 控件的开发之前，我们应该清楚地了解 `ControlWizard` 对项目的巨大贡献。最好的办法是运行 `ControlWizard`，并从生成的源文件中创建一个什么都不做的控件。正如我们将要在下一节中所看到的，创建一个有用的 `ActiveX` 控件需要做一些工作，并且需要进行一定数量的讨论，所有这些都使得在开始编写代码之前，`ControlWizard` 已经为工作控件生成了所有必要的源文件这一事实不那么明朗了。留给开发者的任务通常只是装饰一下现有的控件，其工作已不如

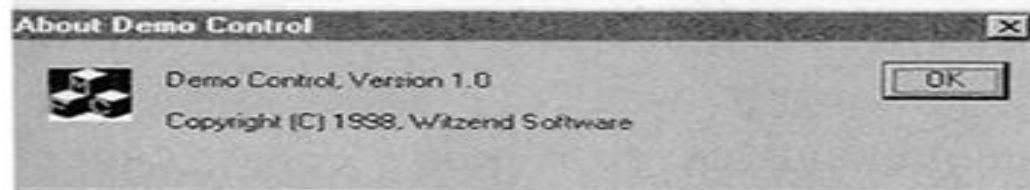
创建 ActiveX 控件那样多了。从一开始, ControlWizard 创建就可以在一个包容器中运行, 显示它自己的窗口, 当它的窗口被移动或调整之后, 做出合适的反应, 显示 About 框, 并显示它的属性表的模型。不过, 该控件所执行的没有一件是有用的工作, 因为它没有触发事件, 导出自定义方法或包含属性。



ActiveX control in
the Test Container



Property sheet

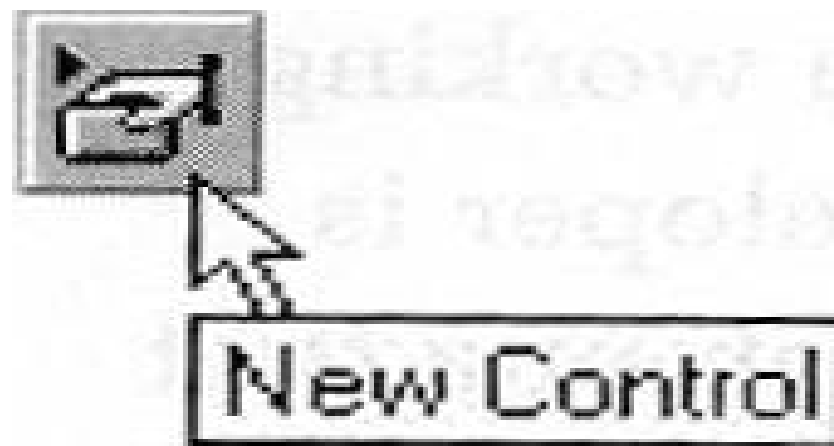


About box

图 9-6 由 ControlWizard 创建的默认 ActiveX 控件

创建如图 9-6 所示的 ActiveX 控件不需要任何编程。如果你愿意进行实验，可以像以前解释的那样通过运行 MFC ControlWizard 来创建一个空项目，将项目的配置改为 Win32 Release，从所生成的源文件中创建控件。像任何 ActiveX 控件一样，结果可能只通过像第 8 章中介绍的 Test Container 实用程序这样的执行容器来加载。从 Tools 菜单中启动 Test Container，调用它的 New Control 工具，为新的 ActiveX 控件搜索列表。该控件与项目具有相同的名字，作为 Demo 控件或某些类似的东西出现在列表之中。将该控件插入 Test Container，然后通过双击该控件的窗口边框或通过单击 Test Container 的 Properties 按钮，来打开如图 9-6 所示的默认属性表。如果想显示控件的 About 框，请选中 Invoke Method 工具，并单击 Invoke 按钮。完成实验之后，退出 Test Container 并关闭项目。注册表中的控件条目可以永远保持，而不受损害，但在从系统中删除一个 ActiveX 控件之前，最好是清理一下注册表。运行第 8 章中介绍的 Regsvr32 实用程序，并包括 /u 开关来取消对该控件的注册：

```
regsvr32/u \demo\release\demo.ocx
```



该实用程序通过显示一个消息框来指示成功：



当控件被成功地取消注册之后，就可以删除项目文件了。

一个名为 `ColeControl::OnDraw` 的过载成员函数用于绘制如图 9-6 所示的控件窗口。与由 `ControlWizard` 所生成的一样，`OnDraw` 函数在白色矩形内部显示一个椭圆：

```
void CDemoCtrl::OnDraw(  
  
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)  
  
{  
  
    pdc->FillRect(rcBounds,  
  
        CBrush::FromHandle(HBRUSH)GetStockObject(WHITE_BRUSH));  
  
    pdc->Ellipse(rcBounds);  
  
}
```

当你开发使用 `Control Wizard` 创建的 `ActiveX` 控件的时候，这个函数是你开始的首选位置之一。我们使用一个范例函数，演示了如何重写 `OnDraw` 函数，来显示

一个含义更加丰富的控件窗口。

例 2：Tower ActiveX 控件

本节将对前一节的内容进行扩展，展示了一个简单的项目，用来说明如何从 ControlWizard 开始，来开发一个有用的 ActiveX 控件。我已经将其命名为 Tower，因为它是汉诺塔（Tower of Hanoi）难题的变种，这个难题由十九世纪法国数学家 Edouard Lucas 提出。图 9-7 显示了 Tower 控件出现在对话中的情形，它显示为分为三层的矩形窗口。游戏的目的是一个接一个地从第一个窗格中拖动几个新块，并将其重新安装在第三个窗格之中。你可以从任何一个窗格向另一个窗格移动单个新块，但不能把大块放在小块之上。

尽管仅仅是个游戏，但是 Tower 控件显示了一个典型 ActiveX 控件的所有标志。Tower 包含库存和自定义属性，可以导出方法，以及触发事件，来让容器应用程序知道游戏的当前状态。在本章稍后，我们将使用 Text Container 实用程序来在控件发生时监视它们的状态。

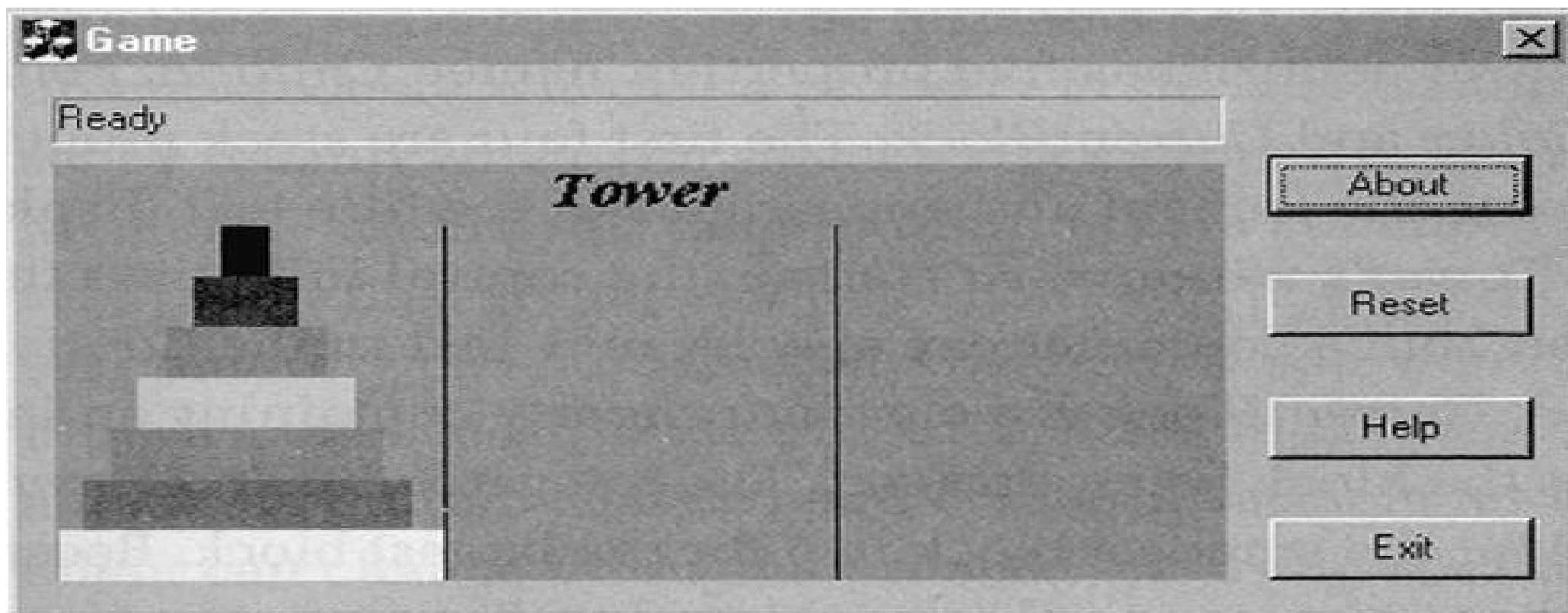


图 9-7 嵌入典型的对话中的 Tower ActiveX 控件

如果你愿意自己创建 Tower ActiveX 控件，可按如下步骤进行，其中解释了操作。不过，讨论不是专门针对 Tower 项目的，而且同时也搜索了你在建立自己的 ActiveX 控件项目时可能想考虑的一些替代方法。第一步是运行 ControlWizard 创建项目，后四步是使用 ClassWizard 向 Tower 项目添加属性、方法、事件和消息处理程序函数。第六步是为控件创建一个简单的属性页。在 ClassWizard 的存

根代码就位之后，第七步是展示如何为了运行第八步中创建和测试的控件而用其他代码来充实程序。图 9-7 中显示的 Game 程序是一个简单的包容器，编写它的目的只是为了演示 Tower。它是在 AppWizard 的帮助下创建的基于对话的应用程序，与前一章中介绍的 Hour 程序有些类似。因为我们已经研究过了这种类型的程序，所以，本节仅仅简单提及一下 Game。你将在本书配套光盘的 Code\Chapter.04 文件夹中找到 Tower 和 Game 包容器程序的源代码。

步骤 1：创建 Tower 项目

Tower 项目是通过 ActiveX ControlWizard 诞生的。通过单击 File 菜单上的 New 命令激活 ControlWizard，然后单击 Projects 选项卡，并选择如图 9-1 所示的 MFC ActiveX ControlWizard 图标。键入 Tower 作为项目名，并接受 ControlWizard 第二步中的默认设置。

步骤 2：添加属性

Tower 控件有五个属性，名字分别为 Caption, Font, ForeColor, BackColor 和 CurrentBlock。头四个用于确定在 Tower 窗口的顶部显示的标题文字和外观的库存属性。在启动的时候，控件将标题文字初始化为“Tower”。但是，如果需要的话，包容器可以在 Caption 属性中指定新文字。CurrentBlock 是包含有代表被拖动块的整数值的 Cursor 属性。CurrentBlock 中整数值的范围是从代表最小块的 0

到代表最大块的 6。因为包容器没有理由改变这个值，所以 Tower 将 Currentblack 作为只读属性来保持。

在像 Tower 这样的 MFC ActiveX 控件中指定属性，需要各种宏的精确放置和取名，因此，这项工作最好留给 ClassWizard 去做。在 ClassWizard 的 Automation 选项卡中，单击 Add Property（添加属性）按钮来打开 Add Property 对话框(图 9-8)，然后单击 External Name（外部名称）框中的箭头来显示一个库存属性列表。通过从列表中选中它，来向项目添加库存属性。如果想指定 Custom 属性，请键入库存名字列表之中没有的任何外部名字。

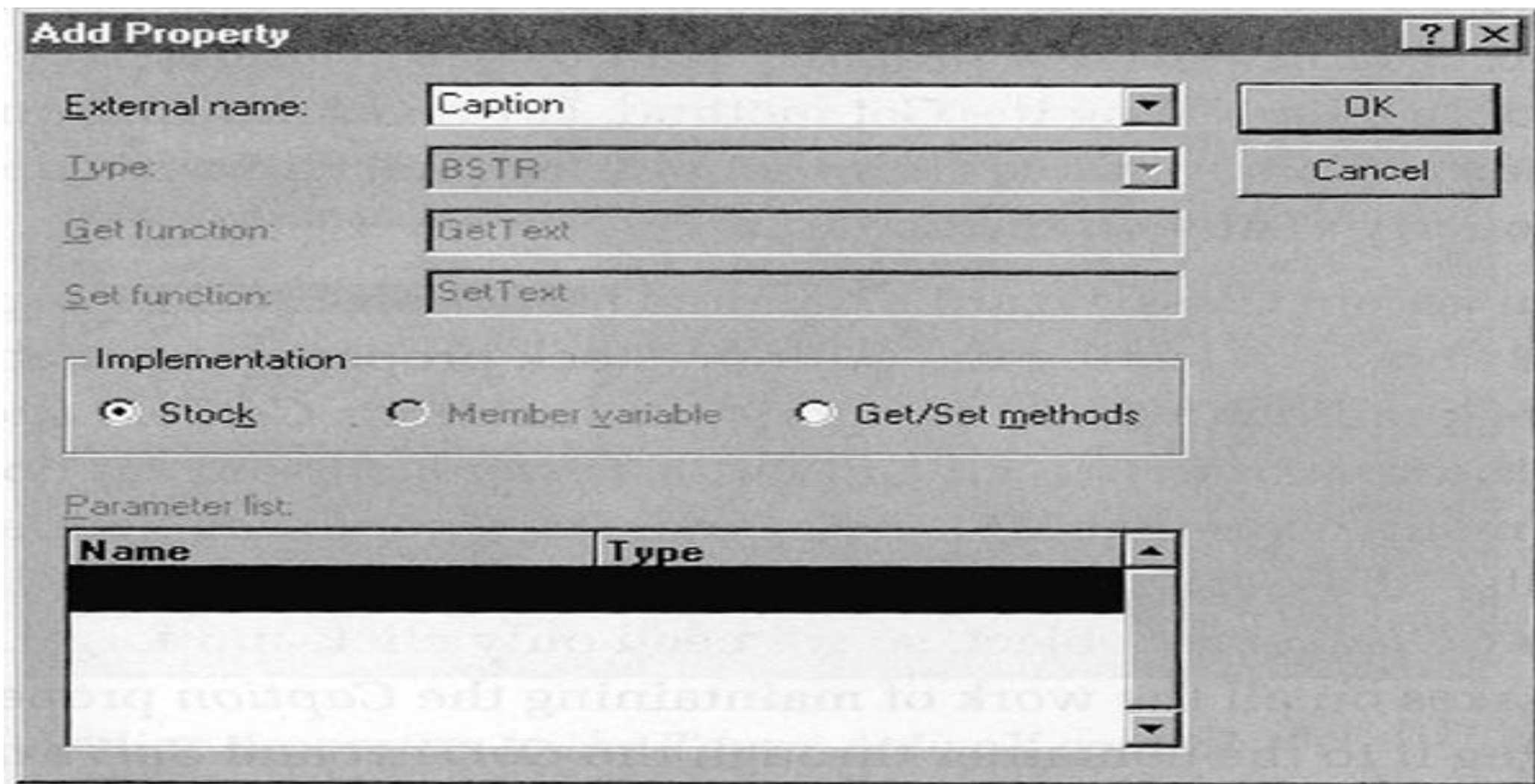


图 9-8 从 Automation 选项卡中调用的 ClassWizard 的 Add Property 对话

利用 Implementation（实现）组框中的 Stock 单选按钮，可以明确地指定一个属

性是库存的，还是自定义的。但是，当你选中库存属性时，该按钮是默认设置的。标签为 Member Variable 和 Get/Set Methods 的单选按钮仅用于自定义属性，为控件将自定义属性展现给包容器应用程序提供了两个选择。如果你想使客户到该属性的访问不受限制，请在 Member Variable 单选按钮处进行设置。ClassWizard 将为客户可以通过属性页改变的属性创建一个变量，并且还可以在包容器改变了属性时，产生一个简单的通知来让控件知道。如果你的控件不需要这个通知，可以清除文本框，以阻止函数的生成，这样可以减少开销量。

为了对包容器如何(或何时)可以读或写自定义属性进行最大限度的控制，请单击 Get/Set Methods 单选按钮。该操作将指示 ClassWizard 为一对包容器可以为了读或写属性而调用的方法生成源代码。如果想检索属性的当前值，包容器可以调用相应的获取方法；如果想设置新值，包容器可以调用 Set 方法。你可以通过省略 Set 方法将属性设为只读，或通过省略 Get 方法，而将属性设为只读。不过，打开 Get/Set Methods 单选按钮至少需要定义一种方法，来使得该属性对于包容器来说是可见的。

我们可以接受 ClassWizard 为 Tower 的库存属性所做的默认设置。如图 9-8 所示。选中 Caption 库存属性将自动打开 Stock 单选按钮，并导出函数 GetText 和 SetText，以允许包容器读和写 Tower 的当前标题。这是 MFC 帮助进行 ActiveX 控件轻松编程的几种方法之一：GetText 和 SetText 方法是框架的 COleControl 对象的成员，因此，我们只需要单击它，随后，便可以忽略它们。框架承担了维护 Caption 属性，并通过 GetText 和 SetText 函数导出到容器的所有工作。这两个函

数从 Text 库存属性中获取它们的名字，在该属性中，Caption 仅仅是一个别名)。单击 OK 按钮将 Caption 属性添加到 Tower 控件，然后，重复相同的步骤来添加 Font, ForeFont 和 BackColor 库存属性。

如果仅仅添加 Tower 的自定义属性，请第五次进入 Add Property 对话框，并且键入 CurrentBlock 作为外部名字，为它分配 short 类型。不是接受建议的通知函数 OnCurrentBlockChanged，而是单击 Get/Set Methods 单选按钮。ClassWizard 将自动分配名为 GetCurrentBlock 和 SetCurrentBlock 的函数。但是，既然 CurrentBlockCustom 属性对于包容器来说应呈现为只读，因此应清除 Set Function 框，以便不再将 SetCurrentBlock 添加到控件类。嵌入了 Tower 的包容器应用程序现在已没有办法改变 CurrentBlock，尽管它可以在任何时候调用 GetCurrentBlock，查询控件，找出属性的当前值。

单击 Add Property 对话中的 OK 按钮，将导致 ClassWizard 在 TowerCtl.cpp 文件中为 GetCurrentBlock 函数编写存根代码。图 9-9 显示了在指定了五个属性之后的 ClassWizard 对话的样子。靠近外部名字的“S”和“C”代码用来指示该属性是库存的，还是自定义的。

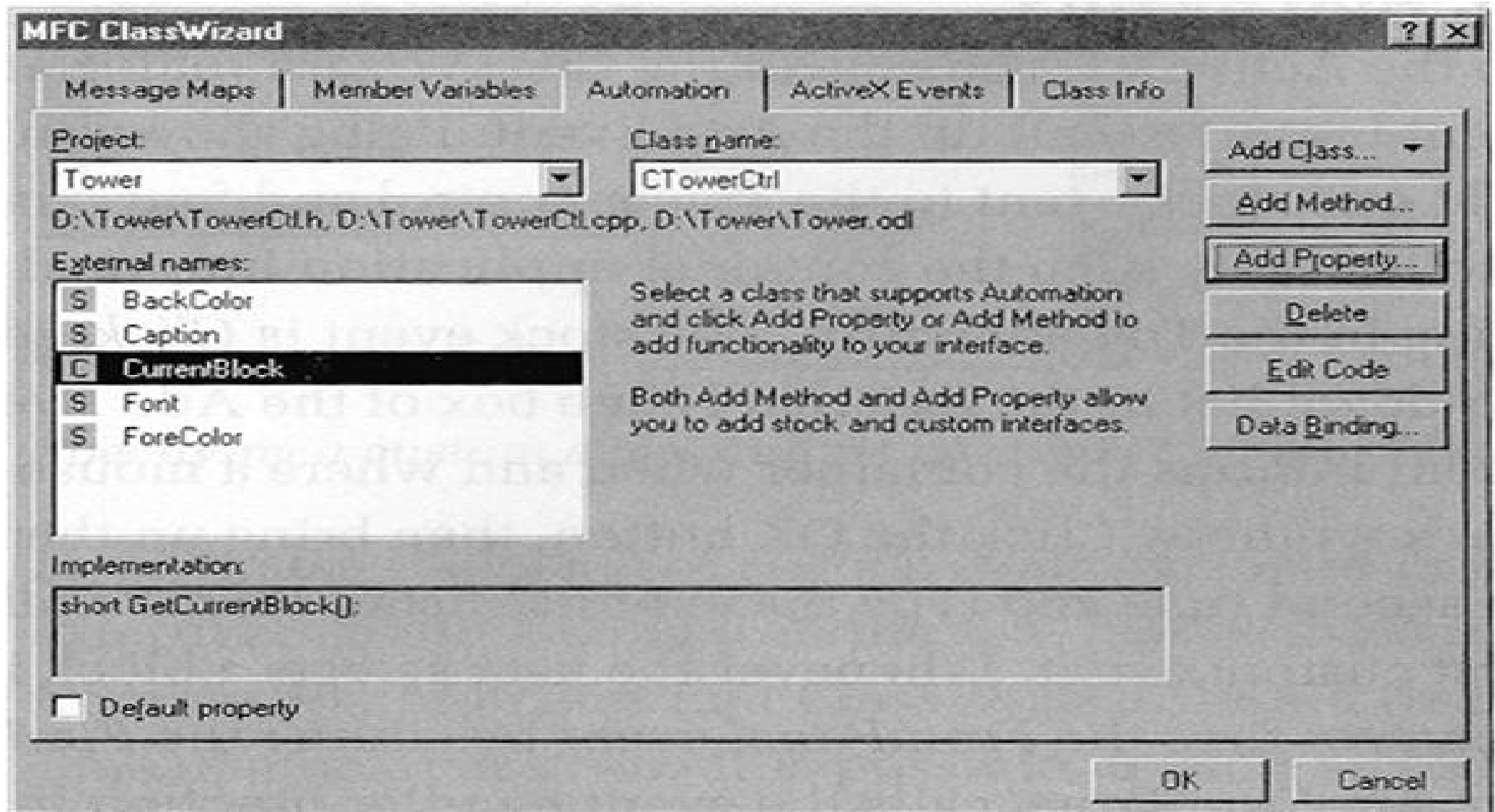


图 9-9 向 Tower ActiveX 控件添加属性

步骤 3：添加方法

从技术上讲，我们已经向 Tower 项目添加了一种方法。在前一节中生成的 GetCurrentBlock 函数就是一种方法。也就是说，它是包容器可以通过接口调用的导出函数。在这里，我们将添加另一个名为 Reset 的方法。Reset 函数为包容器指示控件去重新启动游戏提供了一种方法。图 9-7 中的 Game 程序演示了应用程序是如何使用该特征的，当用户单击对话框的 Reset 按钮时，将调用 Tower 的 Reset 方法。

还是在 ClassWizard 的 Automation 选项卡中，单击 Add Method 按钮，并键入 Reset 作为外部名字。选择 Void 为返回类型，因为 Reset 不返回值。Reset 也没有参数，但如果有的话，将通过双击的标签为 Parameter List(参数列表)的框的 Name 列，并键入函数参数来指定，每行一个参数。在下节中，当我们添加 Tower 的事件时，还要做类似的工作。

单击 OK 返回 ClassWizard 对话框，该对话框现在在外部名字列表中列出了 Reset 方法。“M”前缀将 Reset 作为一种方法来进行标识。

步骤 4：添加事件

下面介绍 ClassWizard 对话框的 ActiveX Events (ActiveX 事件) 选项卡，并单击

Add Event 按钮，来打开如图 9-10 所示的 Add Event 对话框。我们还要向 Tower 控件添加一个库存事件和四个 Custom 事件，它们共同使包容器应用程序能够了解 Tower 控件中所发生的事情。库存事件是 Click，可以从 Add Event 对话框 External Name 框中的下拉式列表中选中它。Click 事件用于通知包容器 Tower 窗口什么时候以及在哪里发生了鼠标单击。单击 OK 按钮，然后第二次打开 Add Event 对话框，并为该控件的第一个 Custom 事件键入外部名字 FromPanel。无论何时用户选中面板中的一块，Tower 控件都将通过调用 FireFromPanel 函数来触发 FromPanel 事件。FireFromPanel 调用客户中的事件处理程序函数，向它传递一个从 0 到 2 的单一参数，这一参数是用来标识从中拖出块的面板。通过双击标签为 Parameter List 的列表框中的蓝色区域展现新条目框来指定函数的参数。在新条目框中，键入代表参数名的 nPanel，并接受默认的类型 short，如图 9-10 所示。

下一个 Custom 事件叫做 ToPanel，它与 FromPanel 类似，有一个类型为 short 的单一参数 nPanel。当用户放下块的时候，Tower 触发 ToPanel 事件，通过 nPanel 参数表明三个面板中哪一个接受了放置操作。第三个自定义事件叫做 Error，当用户企图在小块上面放置大块的时候，Tower 将触发它，来通知包容器这是无效的移动。Error 没有参数表，因此，只需在 External Name 框中键入它，并单击 OK，返回到主 ClassWizard 对话框即可。

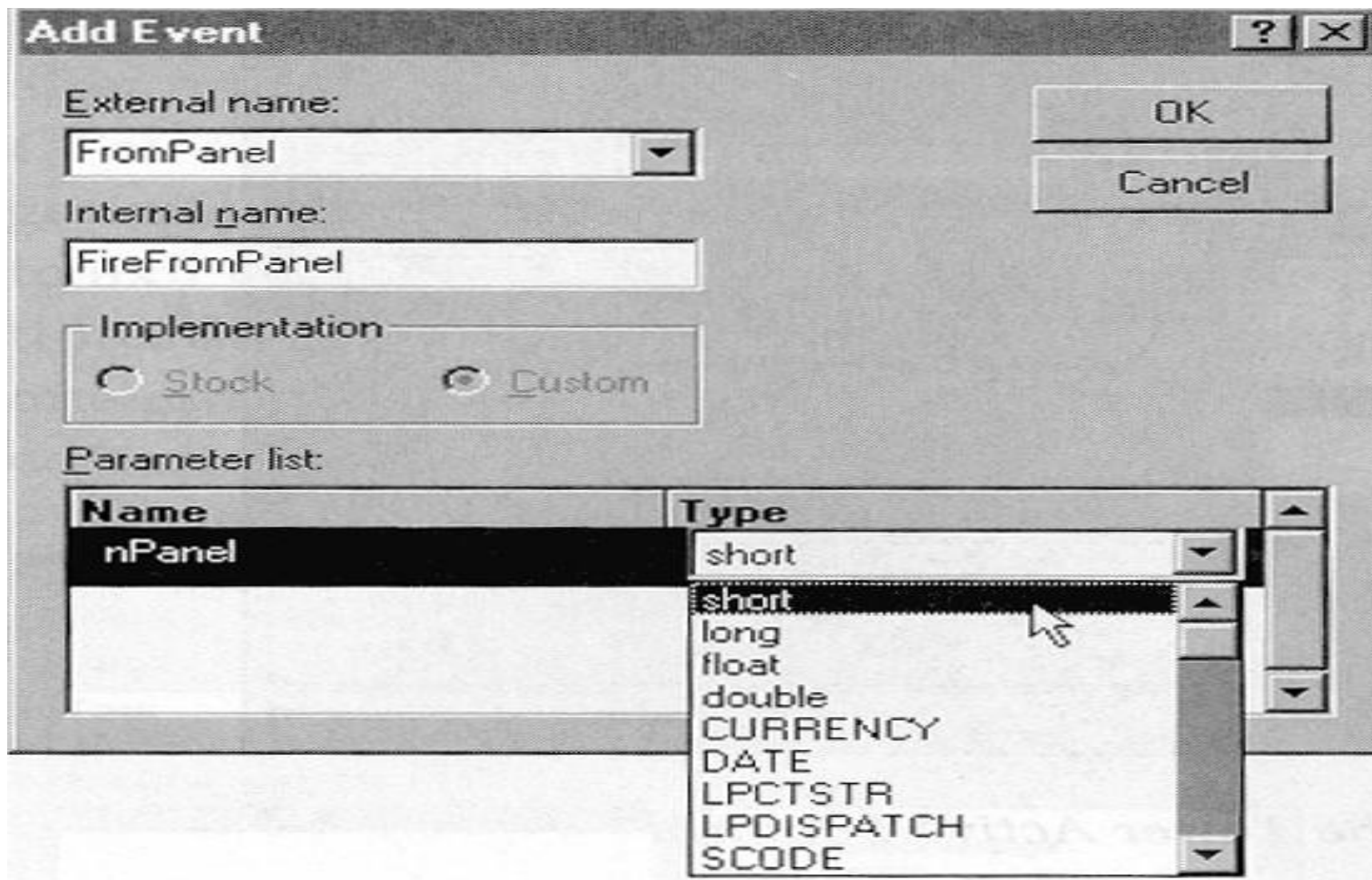


图 9-10 为 Tower ActiveX 控件指定自定义事件及其参数

Error 是包括在外部名字下拉式列表之中的库存事件的名字。在 External Name 框中键入它，而不是从列中选中它，表明 classWizard 应该把这个事件视为自定义的。这就演示了如何可以使用库存事件名以及它的调度标识符用于自定义事件。因为 Error 库存事件至少需要七个参数，因此，通过使用自定义事件，而不是库存事件，可以简化用于控件及其容器的代码。

第四个也就是最后一个自定义事件是 Winner，它用来通知容器，用户已成功地将最后一块移入了控件的第三个面板，从而赢得了这场游戏。与自定义 Error 事件类似，Winner 没有参数表。

图 9-11 显示了在添加了 Tower 的五个事件之后对话的 ActiveX Events 选项卡的样子。

正如在上一章中所提及的，容器不需要为 ActiveX 控件触发的所有事件提供处理程序函数。例如，Game 容器程序将忽略 Tower 的 Click 库存事件，而仅仅处理自定义事件，来在事件发生的时候更新状态窗口。控件设计者必须预料到容器可能需要的信息类型，并且提供事件来传达该信息，同时允许同一个容器忽略某些事件。

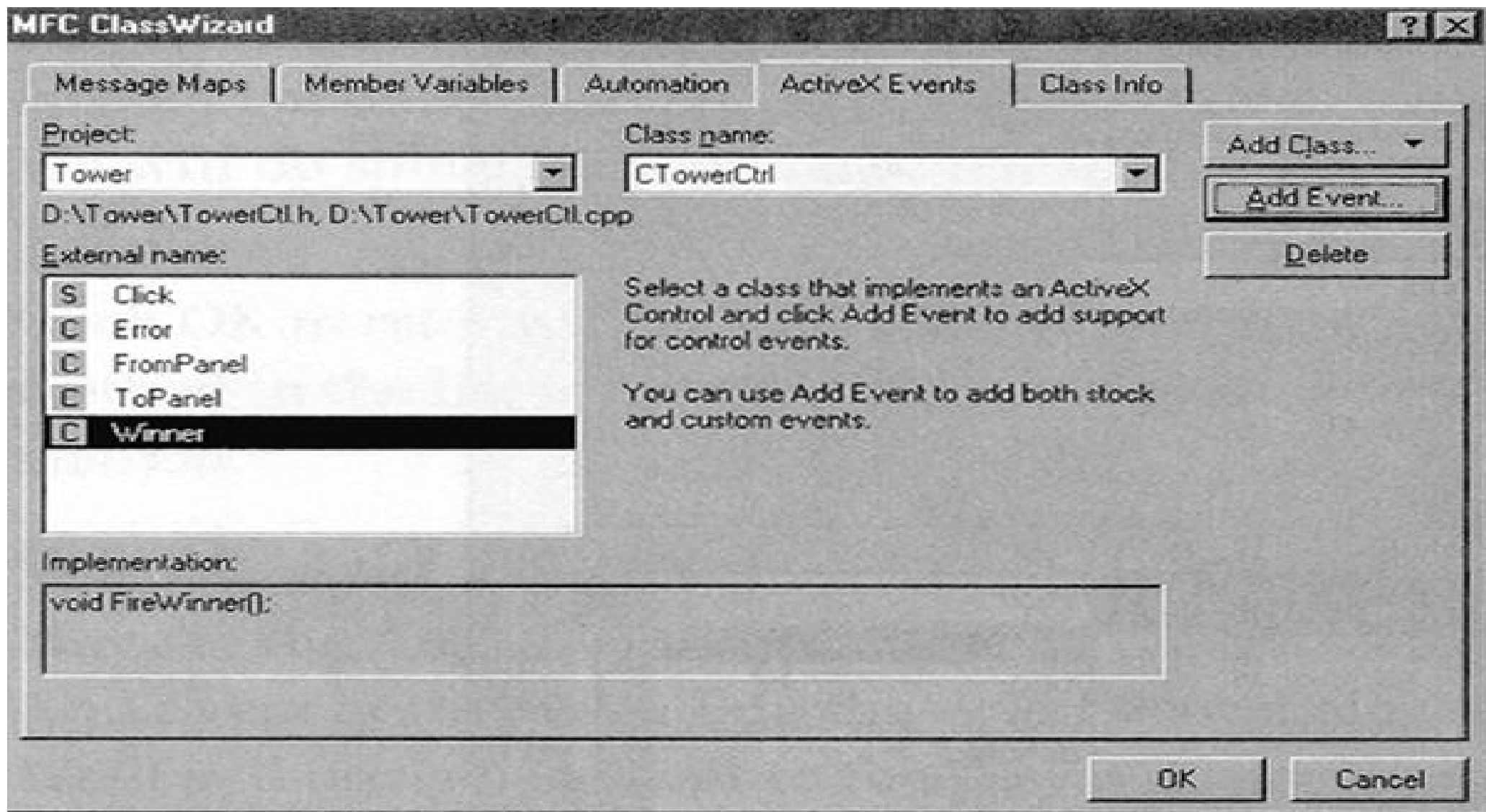


图 9-11 向 Tower ActiveX 控件添加事件

步骤 5：添加消息处理程序函数

在 Tower 中，用户使用拖放方式在面板中移动块。当用户按下左鼠标按钮的时候，箭头将变为十字形状，从而提供反馈，来指示拖动操作起作用了。当鼠标按钮被释放的时候，系统将把鼠标恢复为原来的箭头形状。在本练习的步骤 6 中编写代码的时候，我们将详细说明处理的过程，但目前，我们仅仅需要使用 ClassWizard 来为鼠标消息创建存根处理程序函数。

在 ClassWizard 的 Message Map 选项卡中，从 Messages 框中选中 WM_LBUTTONDOWN 消息，并单击 Add Function 按钮来创建 On_ButtonDown 处理程序函数。为 WM_LBUTTONUP 消息做相同的操作，接受默认的函数名 OnButtonUp。对于第三个消息处理程序函数，从 Messages 框选中 PreCreateWindow，并单击 Add Function 按钮。这将生成 CWnd 虚拟函数的超越存根，为 Tower 最后几分钟的初始化提供一个方便的位置。

图 9-12 显示了在添加需要的函数之后 ClassWizard 的 Message Maps 选项卡的外观 (Tower Ctrl 类的其他成员函数，例如 OnDraw 和 OnResetState，已在前面由 ControlWizard 生成)。完成之后，单击 OK 按钮退出 ClassWizard。我们还必须编写代码来填充 ControlWizard 和 ClassWizard 已经添加到项目中的所有存根代码，但是，还有一个更艰巨的任务，需要 Visual C++ 对话编辑器的服务来完成。

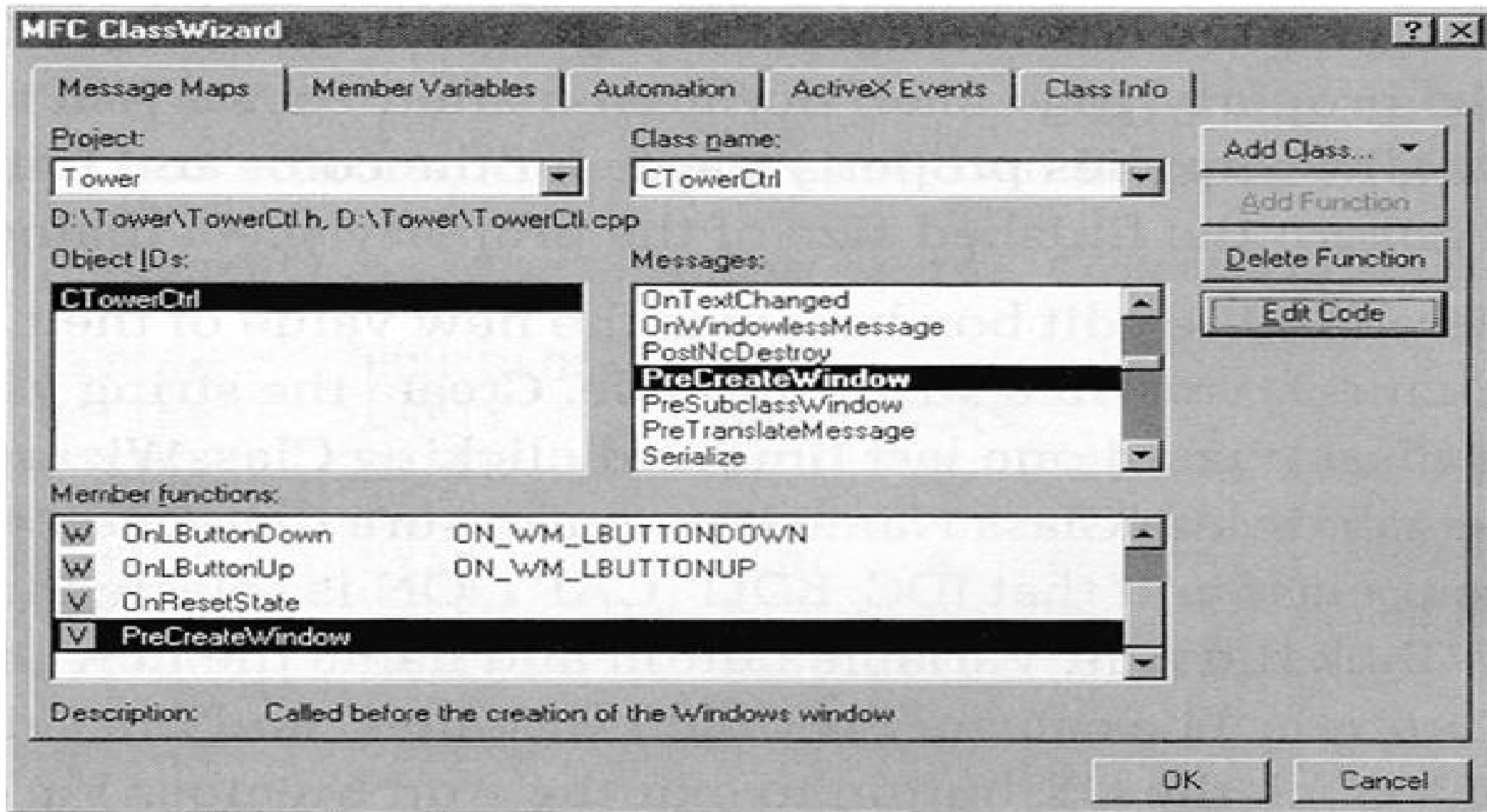
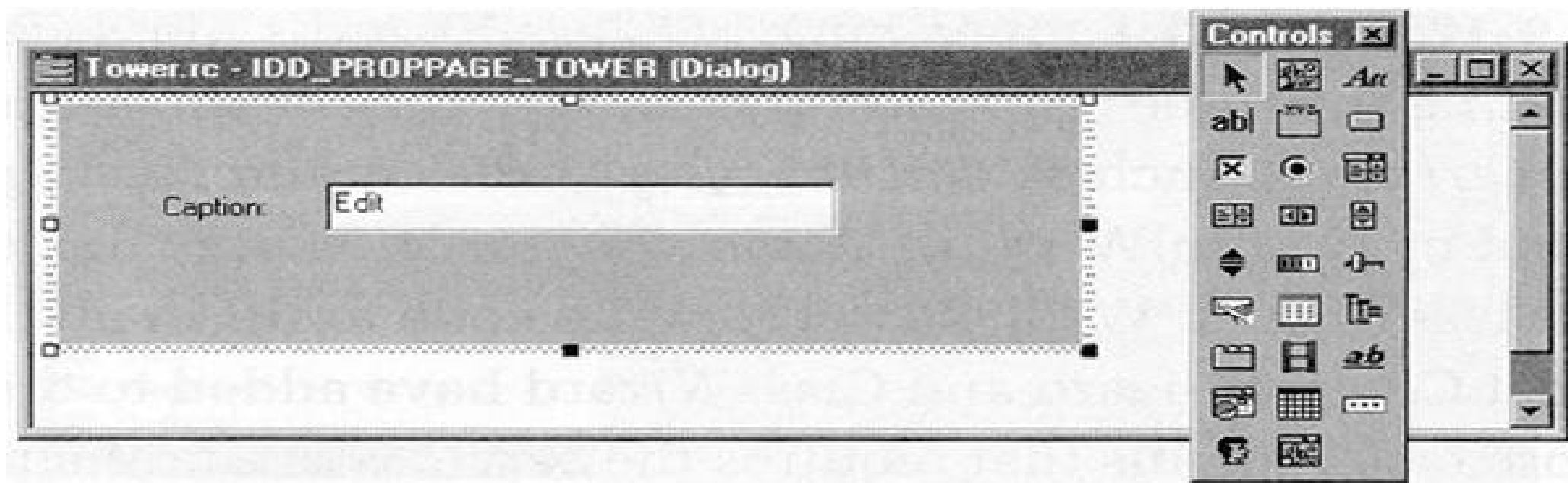


图 9-12 向 Tower ActiveX 控件添加消息处理程序函数

步骤 6：创建属性表

我们以前曾经看到，ControlWizard 向项目添加一个普通的属性页资源(见图 9-6)。本节将解释如何修正该资源，它最终将扩展进入一个可以使用的属性表，该属性表允许用户在设计时查看和改变 Tower 的属性。第一步是在对话编辑器中的修改普通的属性页。通过双击 Workspace 窗口 ResourceView 窗格中的 IDD_PROPPAGE_TOWER 标识符来加载资源。选中对话工作区中的“to do”静态文字控件，并删除它，用如下所示的静态标签和编辑框来代替它。



选中对话工作区中的编辑框，单击 View 菜单上的 Properties，给该框一个标识符

值 `IDC_EDIT_CAPTION`。属性页本身的大小无关紧要。我们一会儿将要看到，MFC 为属性页提供颜色和字体库存属性，它们决定了完成之后的属性表对话的大小。

在编辑框中输入的文字成为 `Caption` 属性的新值，它被存储在一个字符串变量之中。通过最后一次进入 `ClassWizard`，并单击 `ClassWizard` 的 `Member Variables` 选项卡，来创建该字符串变量。在 `Class Name` 框中，确定 `CTowerPropPage` 是当前类，并且在 `Control IDs` 框中已经选中了 `IDC_EDIT_CAPTION`。单击 `Add Variable` 按钮，并将新成员变量命名为 `strCaption`。类别应该是 `Value`，变量类型应该是“`CString`”。单击 `OK` 按钮退出 `Add Member Variable` 对话。最好是通过对话数据验证(请参阅第 6 章)，来限制基于文本的属性(例如 `strCaption`)的长度。通过在 `Member Variable` 选项卡的顶部文本框中键入一个值，可以设置字符串限制：

Description: CString with length validation

Maximum Characters: 25

单击 OK 按钮退出 ClassWizard。用户现在已经可以在设计时通过调用属性页并键入新字符串来改变 Tower 的标题了。在下一节中，我们将看到这些工作是如何完成的。

步骤 7：添加源代码

从 ControlWizard 和 ClassWizard 之间，已经生成了 500 多行源代码。但是，如果我们现在就创建 Tower 控件的话，它看上去还是有点像前面所介绍的什么都没做的 Demo 控件。该练习的第七步将向 TowerCtl.cpp 和 TowerCtl.h 文件添加源代码，充实在前面步骤中添加的存根函数。本节最后将对 TowerPpg.cpp 文件和 Tower.rc 脚本文件进行微小的修改。

TowerCtl.h

TowerCtl.h 头文件只需稍微改动一下。在文本编辑器中加载该文件，并像下面那样对其进行修正，添加阴影行。

```
// TowerCtl.h : Declaration of the CTowerCtrl ActiveX Control class.
```

```
#define      NUM_BLOCKS      7

#define      EMPTY           NUM_BLOCKS

#define      BLACK           RGB(  0,  0,  0 )

#define      BLUE           RGB(  0,  0, 255 )

#define      CYAN           RGB(  0, 255, 255 )


#define      GREEN          RGB(  0, 255,  0 )

#define      MAGENTA        RGB( 255,  0, 255 )
```

```
#define RED RGB( 255, 0, 0 )
```

```
#define YELLOW RGB( 255, 255, 0 )
```

```
////////////////////////////////////
```

```
// CTowerCtrl : See TowerCtl.cpp for implementation.
```

```
class CTowerCtrl : public COleControl
```

```
{
```

```
    DECLARE_DYNCREATE(CTowerCtrl)
```

```
private:
```

```
    short nPanel[3][NUM_BLOCKS]; // Panel contents
```

```
    short nBlockNdx, nFromPanel; // nPanel index of moved block
```

```
    BOOL bMoving; // Flag is set when dragging
```

```
    COLORREF color[NUM_BLOCKS]; // Block colors
```

```
HCURSOR      hCrossHairs;           // Dragging cursor

public:

    short      GetPanel( int i );

// Constructor

public:

    CTowerCtrl();

.
.
.
```

以后，当我们讨论实现代码的时候，将遇到 CTowerCtrl 类的六个成员变量。表 9-1 提供了这些变量的简要说明。

表 9-1 CTowerCtrl 类的成员变量

变量	说明
<i>nPanel</i>	一个 3×7 的数值，其作用是反映任何时刻三个面板的内容。数组元素值 0 到 6 意味着该位置被一个彩色块所占据，0 代表最小的块，6 代表最大的块。元素值为 7 意味着该位置是空闲的。例如，当块整齐地堆在第一个面板的时候，nPanel 数组应该是这样的 <pre> nPanel[0][]={0,1,2,3,4,5,6,}; //Panel 1 nPanel[1][]={7,7,7,7,7,7,7,}; //Panel 2 nPanel[]2[]={7,7,7,7,7,7,7,}; //Panel 3 </pre>
<i>nBlockNdx</i>	正在拖动的块的辅助数组索引
<i>nFormPanel</i>	正在拖动的块的主数组索引
<i>bMoving</i>	当用户拖动块时被设置为 TRUE 的 Boolean 值

续表

<i>Color</i>	包含有块的颜色值的数组。颜色是按照块尺寸递增的顺序安排的，最小块的颜色放在数组的第一个值中
<i>hCrossHairs</i>	系统十字箭头的一个句柄。当用户拖块一个块时，该光标应为十字形状

TowerCtl.cpp

下一个是 TowerCtl.cpp 类实现文件。通过单击 WizardBar 上的棍棒图标打开这个文件，并滚动到在这里显示的属性页映射。按照阴影行中指示的那样做出修改，为颜色和字体属性添加由 MFC 框架提供的属性页：

```
////////////////////////////////////
```

```
// Property pages
```

```
BEGIN_PROPPAGEIDS(CTowerCtrl, 3)
```

```
PROPPAGEID(CTowerPropPage::guid)
```

```
PROPPAGEID( CLSID_CColorPropPage )
```

```
PROPPAGEID( CLSID_CFontPropPage )
```

```
END_PROPPAGEIDS(CTowerCtrl)
```

在映射第一行的 `BEGIN_PROPPAGEIDS` 宏中，将页数设置为 3，这是十分重要的。如果你以后在属性表中添加或删除了页，应该对页数进行调整，以反映出改变。通过使用对话框编辑器创建其他的资源，并且将每个页的新条目插入属性页映射，可以向控件的属性表添加更多的自定义页面。这个过程很少会用到，因此，直到本章最后一节，才讨论附加属性页的问题。

图 9-13 显示了完成之后的 Tower 属性表的样子。该表的顺序是前三页(标签为 `Caption`、`Colors` 和 `Fonts`)对应于属性页映射中三个条目的顺序。剩下的修改影响到后半个 `TowerCtrl.cpp` 实现文件，这里列出的文件以类构造器开头。该清单被分为几部分，每部分之后跟着一段注释，用来解释阴影行中所添加的代码的目的。

```
////////////////////////////////////
```

```
// CTowerCtrl::CTowerCtrl - Constructor
```

CTowerCtrl::CTowerCtrl()

$$\{$$

```
InitializeIIDs(&IID_DTower, &IID_DTowerEvents);
```

```
color[0] = BLACK; // Initialize block colors
```

```
color[1] = BLUE;
```

```
color[2] = C Y A N;
```

```
color[3] = GREEN;
```

```
color[4] = MAGENTA;
```

```
color[5] = RED;
```

```
color[6] = YELLOW;
```

```
Reset(); // Initialize panels
```

```
SetInitialSize( 200, 75 );           // Control window size
```

```
}
```

```
////////////////////////////////////
```

```
// CTowerCtrl::~~CTowerCtrl - Destructor
```

```
CTowerCtrl::~~CTowerCtrl()
```

```
{
```

```
}
```

类构造器用块颜色初始化 color 数组，并调用 Reset 方法来初始化 nPanel 数组。对 COleControl::SetInitialSize 函数的调用为 Tower 控件分配了一个 200×75 像素的默认窗口尺寸。在创建站点时，许多包容器程序超越控件的初始尺寸，因此，对 ActiveX 控件来说，调用 SetInitialSize 通常是没有用的。当你在 Test Container 实用程序中打开控件的时候，该函数的目的就很明显了，它接受控件为自己所建立的任何一个初始尺寸。正如我们在第 8 章中所看到的那样，有些 ActiveX 控件

(例如 Button Menu 控件)在 Test Container 中是作为小方框出现的，为了展现控件窗口，用户必须对其进行调整。对 Tower 并不发生这种情形，因为它通过 SetInitialSize 为自己设置默认窗口尺寸。

```
// CTowerCtrl::OnDraw - Drawing function
```

```
void CTowerCtrl::OnDraw(
```

```
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
```

```
{
```

```
    RECT                rect;
```

```
    TEXTMETRIC          tm;
```

```
    Cpen                 pen;
```

```
    CPen*                pOldPen;
```

```
    CBrush               brush;
```

```
    CBrush*              pOldBrush;
```

```
COLORREF          colorBack = TranslateColor( GetBackColor() );

int               i, j, k, yCaption, iPanelWidth, iPanelHeight;


// Paint control background

brush.CreateSolidBrush( colorBack );

pdc->FillRect( rcBounds, &brush );


pdc->SetBkMode( TRANSPARENT );

pdc->SetTextColor( TranslateColor( GetForeColor() ) );

SelectStockFont( pdc );


// Display caption
```

```
::CopyRect( &rect, rcBounds );

pdc->DrawText( InternalGetText(), -1, &rect, DT_CENTER | DT_TOP );


pdc->GetTextMetrics( &tm );                                // Compute height

yCaption          = tm.tmHeight + tm.tmExternalLeading;      // of Caption area,

iPanelWidth       = rcBounds.Width()/3;                      // width and height

iPanelHeight = rcBounds.Height() - yCaption;                  // of a panel


// Draw column dividers

pen.CreatePen( PS_SOLID, 1, TranslateColor( GetForecolor() ) );

pOldPen = pdc->SelectObject( &pen );

pdc->MoveTo( rcBounds.left+iPanelWidth,    rcBounds.top+yCaption );
```

```
    pdc->LineTo( rcBounds.left+iPanelWidth,    rcBounds.bottom );

    pdc->MoveTo( rcBounds.left+iPanelWidth*2, rcBounds.top+yCaption );

    pdc->LineTo( rcBounds.left+iPanelWidth*2, rcBounds.bottom );


// Save current brush

pOldBrush = (CBrush*) pdc->SelectStockObject( NULL_BRUSH );


// Outer loop: for each panel...

for (i=0; i < 3; i++)

{

    rect.top      = rcBounds.top + yCaption;

    rect.bottom = rect.top + iPanelHeight/NUM_BLOCKS;
```

```
// Inner loop: for each colored block in panel...

for (j=0; j < NUM_BLOCKS; j++)

{

    if (nPanel[i][j] != EMPTY)

    {

        // Determine left and right edges of colored block

        k = NUM_BLOCKS - 1 - nPanel[i][j];

        rect.left  = rcBounds.left + iPanelWidth*i +

                    (iPanelWidth*k)/(2*NUM_BLOCKS) + 1;

        rect.right = rect.left +

                    iPanelWidth*(nPanel[i][j]+1)/NUM_BLOCKS - 1;
```

```
        // Fill rectangle with block's color

        brush.CreateSolidBrush( color[nPanel[i][j]] );

        pdc->SelectObject( &brush );

        pdc->FillRect( &rect, &brush );

    }

    rect.top      = rect.bottom;

    rect.bottom += iPanelHeight/NUM_BLOCKS;

}

}
```

```
pdc->SelectObject( pOldPen );  
  
pdc->SelectObject( pOldBrush );  
  
}
```

下一步，需在该类的 `OnDraw` 函数中修改源代码，无论何时，`Tower` 的窗口失效时都将执行这个函数。我们以前曾经看到 `ControlWizard` 编写 `OnDraw` 的简单版本。在白色矩形中显示普通的椭圆。在这里，我们来修改一下这个函数，在 `Tower` 的三个面板中绘制彩块的当前排列情况。无论何时用户移动了一个块，窗口都将重画自己，以反映出改变。

`OnDraw` 首先用控件的 `BackColor` 属性的当前值来画窗口背景。该函数从 `ColeControl::GetBackColor` 函数中检索到属性，通过 `ColeControl::TranslateColor` 将它转换为 `COLORREF` 值，并创建一个刷子，用它来填充窗口矩形(`rcBounds` 参数提供了 `Tower` 窗口相对于包容器窗口原点的坐标)。类似地，该函数使用 `ForeColor` 属性的当前值来设置设备上下文的文字颜色，并使用 `Font` 属性设置当前的字体。`OnDraw` 然后编写包含在 `Caption` 属性之中的字符串，将文字在 `Power` 窗口的顶部居中显示：

```
pdc->DrawText(InternalGetText(),-1,&rect,DT_CENTER | DT_TDP);
```

从 `Tower` 窗口的高度中减去标题文字的高度便是面板的高度，这个值被存储在

iPanelHeight 之中。每个彩块的高度(厚度)是面板高度的七分之一，因此，七个块堆起来正好从面板底部堆到顶部。每个面板的宽度是整个窗口宽度的三分之一。有了这些尺寸之后，OnDraw 就开始准备在面板中显示彩块了。

每个块的当前位置被存储在一个 3×7 的 nPanel 数组之中。使用为每个面板重复的外循环和为每个块重复的内循环，该函数可以遍历所出现的 21 个槽中的每一个，并在每一步读取 nPanel 数组的一个元素。EMPTY 元素值意味着该槽没有包含一个块。如果一个元素具有从 0 到 6 的值，OnDraw 将使用 Color 数组中相应的颜色为槽中的块上色。一个名为 rect 的 RECT 结构用于保持当前块的坐标。

```
////////////////////////////////////
```

```
// CTowerCtrl::DoPropExchange - Persistence support
```

```
void CTowerCtrl::DoPropExchange(CPropExchange* pPX)
```

```
{
```

```
    ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
```

```
    COleControl::DoPropExchange(pPX);
```

```
}
```



```
////////////////////////////////////
```

```
// CTowerCtrl::OnResetState - Reset control to default state
```

```
void CTowerCtrl::OnResetState()
```

```
{
```

```
    COleControl::OnResetState();    // Resets defaults found in DoPropExchange
```

```
}
```

```
////////////////////////////////////
```

```
// CTowerCtrl::AboutBox - Display an "About" box to the user
```

```
void CTowerCtrl::AboutBox()

{

    CDialog dlgAbout(IDD_ABOUTBOX_TOWER);

    dlgAbout.DoModal();

}
```

尽管 Tower 没有改变类的 DoPropExchange 函数，但是，简单考虑一下该函数还是值得的。属性交换允许 ActiveX 控件保存嵌入物之间的自定义属性。例如，每当它启动的时候，Tower 控件总是初始化游戏，并在第一个面板中安装彩块堆。通过属性交换，我们可以增强 Tower，以便在关闭的时候保存被中断了的游戏，并且在下次包容器嵌入该控件的时候，重新创建同一个块位置。在运行之间保存和恢复的属性具有持久性。

由框架管理的库存属性自动具有持久性。为了使自定义属性具有持久性，应向 DoPropExchange 函数添加合适的属性交换函数，当该控件被加载以及当它再次中止时，该函数便可执行。属性交换函数是用 PX_前缀后面跟着由函数系列化的数据类型来进行标识的。例如，PX_Bool, PX_Font 和 PX_String 函数使得 Boolean、字体和 CString 属性具有持久性。如果想获得这些及其他属性交换函数的说明，请参考在线帮助。

```
////////////////////////////////////
```

```
// CTowerCtrl message handlers
```

```
short CTowerCtrl::GetCurrentBlock()
```

```
{
```

```
    return nPanel[nFromPanel][nBlockNdx];
```

```
}
```

GetCurrentBlock 函数是在第二步中建立的方法，在此，包容器想了解哪一个块正在移动(尽管 ClassWizard 向代码中添加标题，但 GetCurrentBlock 不是消息处理函数)。可以在任何时候调用这个函数，但是，如果包容器对 GetCurrentBlock 所提供的信息感兴趣的话，它可能将调用该函数来响应 FromPanel 事件，这个事件宣布一个块正在被移动。GetCurrentBlock 的 EMPTY 的返回值意味着用户当前没有拖动块。可以回想一下第二步过程，因为包容器没有理由改变属性，所以，Tower 不为 CurrentBlock 属性导进相应的 Set 方法。

```
void CTowerCtrl::Reset()
```

```
{
```

```
int i;

for (i=0; i < NUM_BLOCKS; i++)          // Initialize panel array
{
    nPanel[0][i] = i;                    // Panel 0 = 0,1,2,3,4,5,6
    nPanel[1][i] = EMPTY;                // Panel 1 = 7,7,7,7,7,7,7
    nPanel[2][i] = EMPTY;                // Panel 2 = 7,7,7,7,7,7,7
}

nBlockNdx    = 0;                        // Ndx of block being moved

nFromPanel = 0;

InvalidateControl();
```

```
}
```

利用 Reset 方法，包容器应用程序可以重新开始游戏。例如，当用户单击如图 9-7 所示的 Reset 按钮的时候，Game 程序将调用 Reset 方法。TowerCtrl 类构造器还调用 Reset，以在启动时初始化 nPanel 数组，在第一个面板中堆起七个块，并将其他两个面板中的所有位置标记为空。Reset 调用 COleControl::InvalidateControl 函数来触发对 OnDraw 的一次调用，刷新控件窗口。为了使自己失效，基于 COleControl 的 ActiveX 控件应调用 InvalidateControl 函数，而不是 Invalidate API 函数。

```
BOOL CTowerCtrl::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
```

```
    SetText( "Tower" );                                // Default caption
```

```
    hCrossHairs = ::LoadCursor( NULL, IDC_CROSS );      // Dragging cursor
```

```
    return COleControl::PreCreateWindow(cs);
```

```
}
```

Tower 通过监视左鼠标按钮来模拟拖和放操作。当用户在 Tower 窗口之中按下鼠

标按钮的时候，控件将把箭头改为系统十字形状，以为用户提供简单而形象的反馈，表明拖操作起作用了。当容器第一次嵌入 Tower 控件时调用的 PreCreateWindow 函数将加载十字箭头，并将该句柄存储在 hCrosHairs 之中。该函数还调用 ColeControl::SetText 来初始化 Caption 属性。

```
void CTowerCtrl::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
```

```
    short  i=0;

    nFromPanel = GetPanel( point.x );           // Panel from which block is taken

    while (nPanel[nFromPanel][i] == EMPTY  &&  i < NUM_BLOCKS)

        i++;                                   // i=ndx of panel's smallest block

    if (i < NUM_BLOCKS)                        // Does panel have a block in it?

    {

        bMoving      = TRUE;                  // If so, block is now moving

        nBlockNdx = i;                        // Save ndx of the block
```

```
        ::SetCursor( hCrossHairs );           // Change cursor to indicate drag

        FireFromPanel( nFromPanel );          // Tell container the panel number

    }
```

```
COleControl::OnLButtonDown(nFlags, point);
```

```
}
```

```
short CTowerCtrl::GetPanel( int x )
```

```
{
```

```
    short  i=0;
```

```
    RECT rect;
```

```
    GetClientRect( &rect );                // Control window
```

```
    if (x > rect.right/3)                    // Hit test:
```

```
        i = 1;                                // i = 0 for first panel

    if (x > rect.right*2/3)                    // = 1 for second panel

        i = 2;                                // = 2 for third panel

    return i;                                  // Return panel number
}
```

当用户在 Tower 窗口中的某处按下鼠标左按钮的时候，OnLButtonDown 函数将处理产生的 WM_LBUTTONDOWN 消息。该函数首先检查 point 参数中的单击坐标，然后确定单击发生在哪个面板之中。如果面板为空，单击将被忽略。否则，OnLButtonDown 将把箭头改为十字形状，并触发 FromPanel 事件，来将正在移动块的消息通知包容器。

因为仅有面板中最小的块才能被移动，因此，OnLButtonDown 只需确定单击发生在哪个面板之中，而不必确定发生在哪个块上。尽管这可以大大简化帮助器 GetPanel 函数中测试的次数，它同时还具有为一个块启动拖动操作的作用，即使单击没有精确地落在该块上面。

```
void CTowerCtrl::OnLButtonUp(UINT nFlags, CPoint point)
```



```
{
```

```
    short  i=0, nToPanel;
```

```
    nToPanel = GetPanel( point.x );  // Panel in which block is dropped
```

```
    if (bMoving  &&  nToPanel != nFromPanel)
```

```
    {
```

```
        while (nPanel[nToPanel][i] == EMPTY  &&  i < NUM_BLOCKS-1)
```

```
            i++;                      // i=ndx of panel's smallest block
```

```
            // Is dragged block smaller than smallest block in panel?
```

```
            if (nPanel[nFromPanel][nBlockNdx] < nPanel[nToPanel][i])
```

```
            {
```

```
                if (nPanel[nToPanel][i] != EMPTY)
```

```
                    --i;
```



```
        FireError ();                                // tell container

    }

    bMoving = FALSE;                                // Not moving now

    COleControl::OnLButtonUp(nFlags, point);

}
```

当用户在面板中释放左鼠标按钮来放下块时，OnButtonUp 函数接收控制。该函数比它的同伴 OnLButtonDown 有更多的工作要做。除了确定放置操作发生哪一个面板中之外，OnLButtonUp 还必须确定该面板中的块均不小于所放置的块。如果有的话，OnButtonUp 将触发 Error 事件通知包容器，说用户企图进行非法放置。如果放操作是合法的，OnLButtonUp 将触发 ToPanel 事件，来宣布放操作的结束。如果块正在被拖入第三个面板最上面的槽中，那么游戏结束，OnLButtonUp 将触发 Winner 事件。

释放鼠标按钮很方便，但也有副作用，即在 OnLButtonUp 没有采取任何动作的情况下，光标便恢复到以前的形状。因为 Tower 不处理 WM_SETCURSOR 消息，所以，当用户释放鼠标按钮的时候，系统将自动恢复原来的窗口光标。

TowerPpg.cpp

在步骤 6 中，我们通过添加一个允许用户重写 Caption 属性的文本框，来修改 Tower 的普通属性。Tower 在 StrCaption 变量中存储该文本框的内容，该变量是一个用 ClassWizard 创建的 CString 对象。在 setCaption 字符串和 Caption 库存属性之间需要一个链接，以便当用户在 Tower 的属性表中改变 strCaption 的时候，控件可以把改变转发给埋藏在框架之中的库存属性。

可以通过 DDP_前缀来识别 MFC 的属性数据传输函数的目的。Tower 所需要的函数是 DPP_Text 函数，它可以将文字以字符串变量(strCaption)中复制到字符属性(Caption)。如果想添加 DDP_Text 调用，请在文本编辑器中打开 TowerPpg.cpp 实现文件，并插入如下所示的阴影行：

```
void CTowerPropPage::DoDataExchange(CDataExchange* pDX)
```

```
{
```

```
//{{AFX_DATA_MAP(CTowerPropPage)
```

```
DDP_Text(pDX, IDC_EDIT_CAPTION, strCaption, _T("Caption"));
```

```
DDX_Text(pDX, IDC_EDIT_CAPTION, strCaption);
```

```
DDV_MaxChars(pDX, strCaption, 25);

//}}AFX_DATA_MAP

DDP_PostProcessing(pDX);

}
```

Tower.rc

在默认状态下，ControlWizard 将普通属性页标为 General，将该标签作为字符串资源存储在 Tower.rc 脚本文件之中。如果想改变该选项卡标签，请在文本编辑器中打开 Tower.rc 文件，并在如下所示的阴影行中将“General”改为“Caption”：

```
STRINGTABLE DISCARDABLE
```

```
BEGIN
```

```
IDS_TOWER                "Tower Control"
```

```
IDS_TOWER_PPG            "Tower Property Page"
```

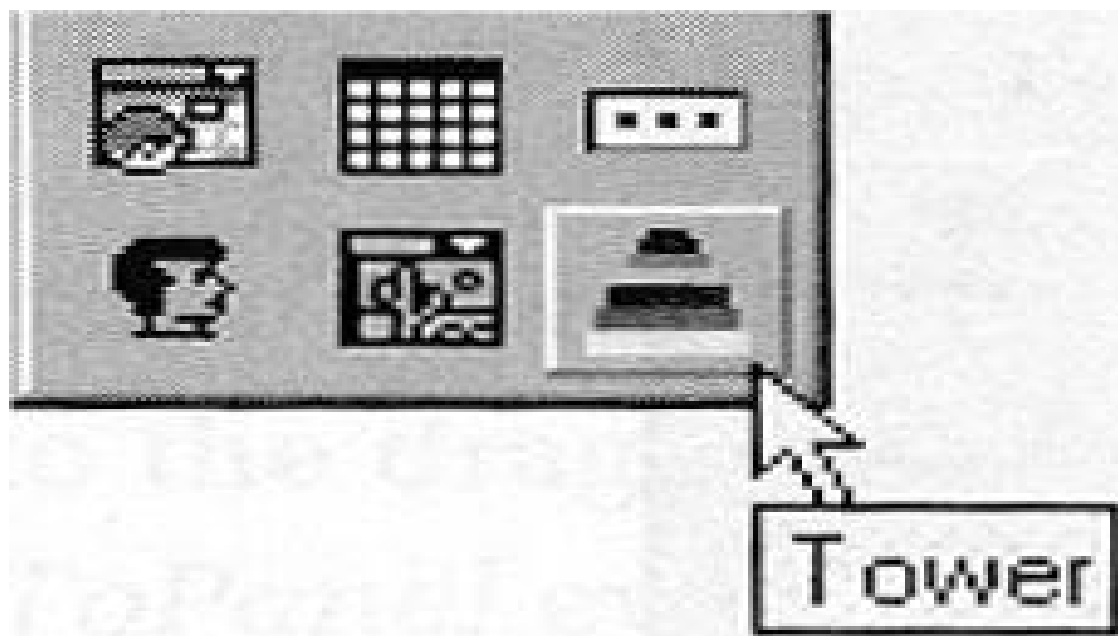
```
IDS_TOWER_PPG_CAPTION    "Caption"
```

END

你还可以使用第 4 章中介绍的 Visual C++ 字符串编辑器实现相同的修改。图 9-13 显示了修改后的结果。

步骤 8：创建和测试 Tower ActiveX 控件

如果你正在从源代码中创建 Tower 控件，首先应将文件 Tower.ico 和 TowerCtl.bmp 从 Code\chapter.09\Tower 文件夹复制到项目文件夹。第一个文件提供了在控件的 About 框中出现的独一无二的图标资源。第二个文件包含有一个位图，包容器可以在嵌入控件时从这个位图中创建一个个性化的工具按钮。正如我们在前一章中所看到的那样，Visual C++ 自己利用该控件的位图。当一个 ActiveX 控件通过 Gallery 添加到项目的时候，对话编辑器将从该控件的资源文件中展开位图，并使用这个图像在 Controls 工具栏上画新的工具按钮：



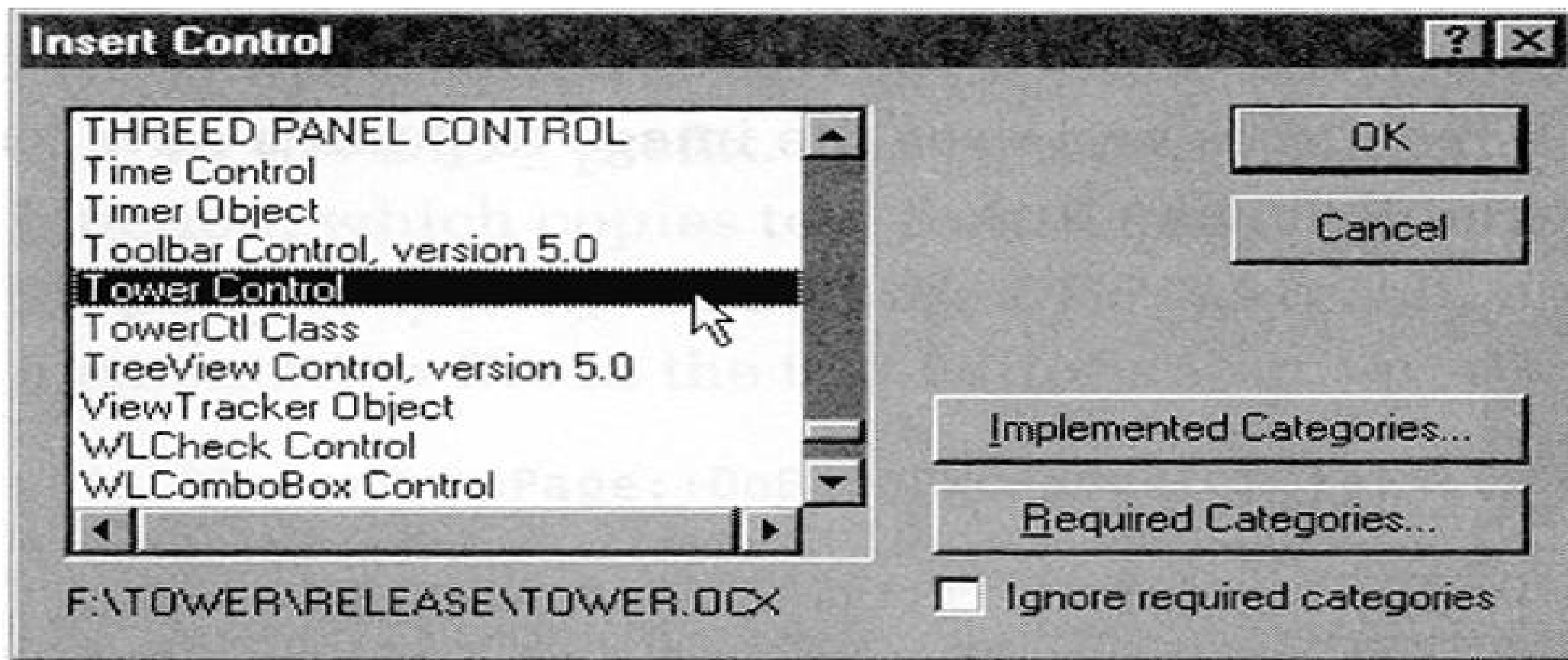
对于这种目的，一个图标一般来说太大了，因此，ActiveX 标准建议，在控件的资源数据中包括 16×16 位图图像。提供这种资源的控件通过类的注册表数据中的 `ToolboxBitmap32` 键来宣告自己。在下一章中，我们还要遇到这个键。

将项目配置设为 Win32 Release，并且从 Build 菜单选择 Build 命令。当源码成功编译和链接之后，Visual C++ 将自动注册这个控件。如果你想不将 Tower 作为项目创建便拿它做试验，那么，在使用它之前你必须注册该控件。如果想注册 Tower，首先应将 Tower.ocx 文件复制到硬盘（如果必要的话），然后运行 RegSvr32 实用程序：

```
regsvr32 path\tower.ocx
```

其中， path 代表 Tower.ocx 文件在硬盘上的位置。

本书配套光盘上的 Game 程序为试验新的 Tower 控件提供了最方便的方法。Game 具有用于显示 Tower 的 About 框、复位游戏以及列出游戏规则的按钮。程序还可以连续不断地显示游戏状态，这是通过监视 Tower 的 FromPanel, ToPanel, Error 和 Winner 事件来实现的。Test Container 实用程序提供了另一种方法来用 Tower 控件做试验，并且能够比 Game 程序展现该控件更多的内部工作过程。正确注册了控件之后，运行 Test Container，单击 New Control 按钮，从如下的列表中选中 Tower 控件：



Insert Control 对话框响应键盘条目，让你通过按控件名的第一个字母来快速滚动列表。例如，按下 T 可以立即把选择条放在 Tower 控件条目上。

通过双击控件边框，可以试一试 Tower 的属性表。MFC 框架已经添加了属性页，你可以在里面修改用于在 Tower 窗口中显示标题的控件和字体。图 9-13 显示了在控件的属性表中展现的 Font 页。在对话的其他三个选项卡中，Caption 选项卡

用于显示在步骤 6 中修改的属性页。当你为标题键入新文字的时候，变化立即在控件的窗口中反映出来。

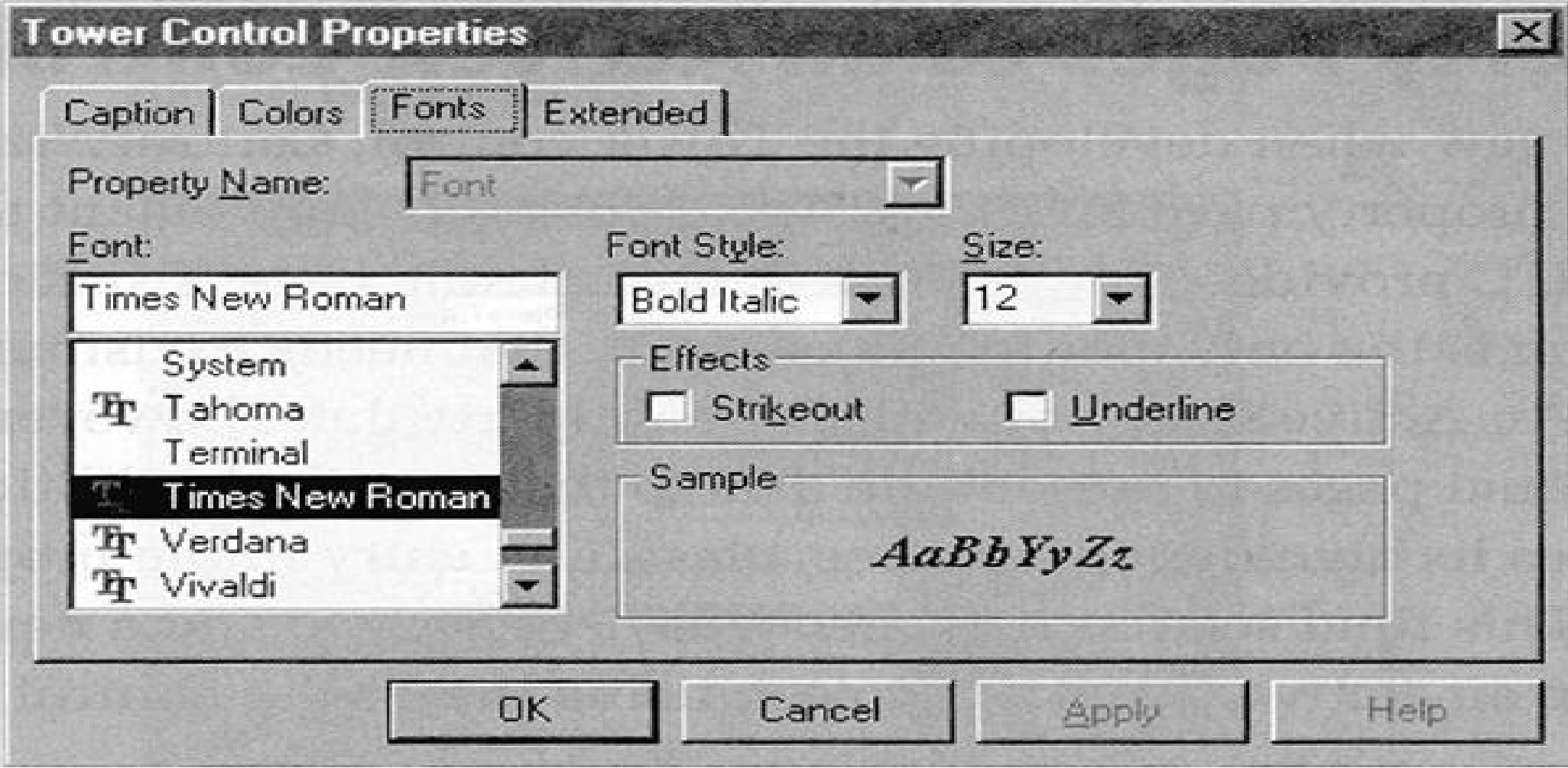


图 9-13 Tower Control Properties 属性表

Test Container 可以显示 Tower 事件的实时记录，当你调试一个 ActiveX 控件的时候，这是非常宝贵的。向上移动分隔栏，以为事件日志展现更多的空间，然后从 Tower 的第一个面板中拖动一个块，并把它放到另一个面板之中。作为对拖放操作的响应，Tower 触发 ClickFromPanel 和 ToPanel 事件，当它们发生的时候，在事件日志中均有记录。如图 9-14 所示，日志中的每个条目均以下一个用来指示事件在哪个控件发生的标签作为开头。既然 Test Container 可以嵌入多个控件，那么，利用标签，在事件很多时，日志条目仍可以保持直观。

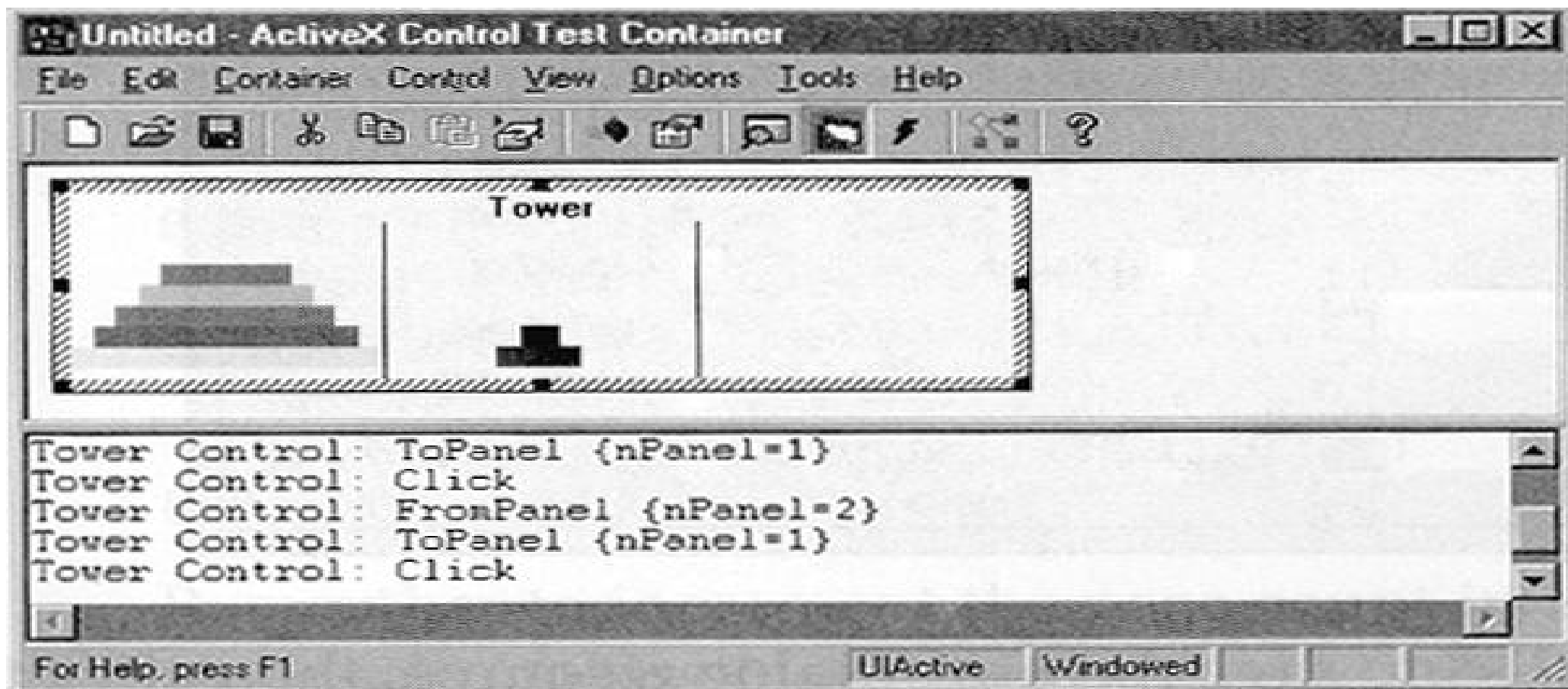
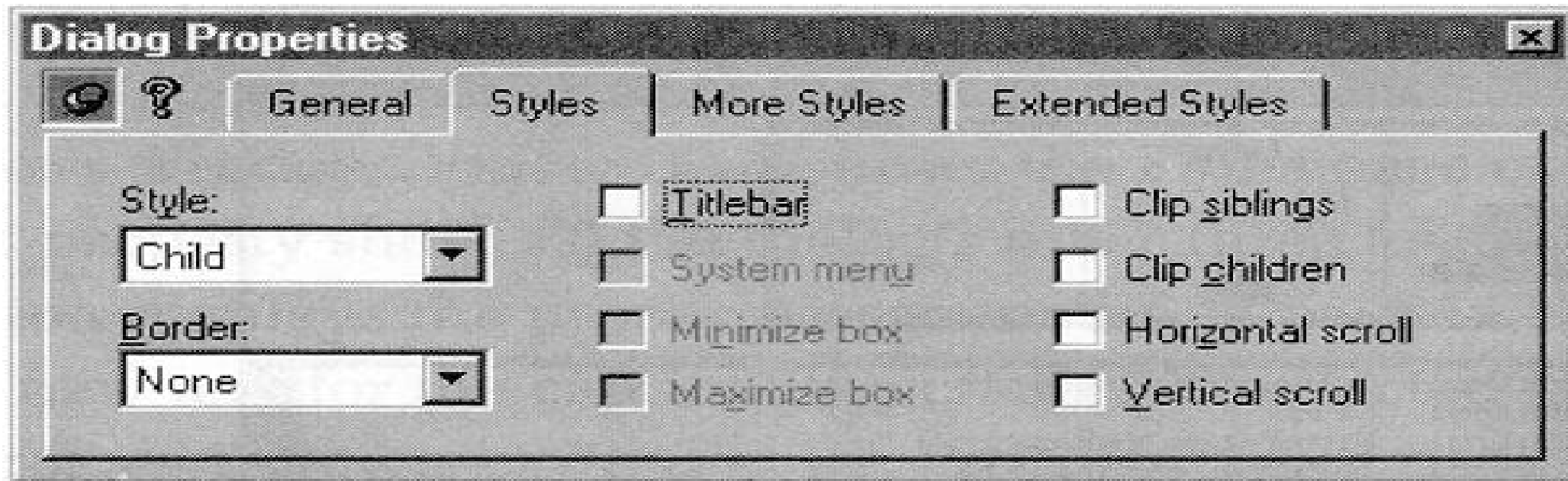


图 9-14 在 Test Container 中监视事件

向 ActiveX 控件项目添加属性页

正如我们在开发 Tower 控件时所看到的, ControlWizard 可以为 ActiveX 控件生成一个单一属性页, 来补充 MFC 提供的库存页。对于 Tower 项目, 一页就足够了, 原因是 Caption 是需要自定义属性页的唯一一个可修改的属性。不过, 带有多个属性数据的 ActiveX 控件可能需要其他的页来将数据展现给用户。每个附加页都需要在属性页映射中具有它自己的对话资源类和条目。最后一节列出了向 ActiveX 控件项目添加新属性页的必要步骤, 使用 Tower 控件作为例子。

1. 在项目处于打开状态的时候, 单击 Insert 菜单上的 Resource 命令, 并双击列表中的 Dialog 来启动对话编辑器。你还可以展开对话资源列表, 并选中 IDD_OLE_PROPPAGE_LARGE。和以前一样, 新属性页的大小并不重要。按照你所希望的那样设计新属性页, 然后在工作区中选中对话框, 并单击 View 菜单上的 Properties 命令, 来展现 Dialog Properties 框。在 Styles 选项卡中, 关闭 Titlebar 复选框, 将对话样式设为 Child, 禁用边框。如果你选中了 IDD_OLE_PROPPAGE_LARGE, 来启动对话编辑器, 这些设置就已经为你做好了。Styles 选项卡应该是下图的样子:



2. 按 Ctrl+S 来保存新对话资源，然后单击 View 菜单中的 ClassWizard 命令。当询问你是否愿意为对话资源创建一个新类的时候，单击 OK 按钮表示接受。为新类键入一个名字，选择 COlePropertyPage 作为基类：

New Class



Class information

Name:

CTowerPropPage2

File name:

TowerPPG2.cpp

Change...

Base class:

CObjectPropertyPage

Dialog ID:

CMainFrameWnd

CObjectDocument

CObjectLinkingDoc

CObjectPropertyPage

CObjectServerDoc

CObjectServerItem

CPrintDialog

CProgressCtrl

Automation

☒ None

☐ Automation

☐ Createable by type ID: Tower.TowerPropPage2

The base class does not support automation.

OK

Cancel

3. 为新页添加所需要的所有成员变量，然后退出 ClassWizard。在 TowerCtl.cpp 实现文件中，为 ClassWizard 刚刚创建的属性类头文件添加一条 #include 语句，如下：

```
# include "TowerPPG2.h".
```

正确的文件名出现在 New Class 对话的 File Name 框中，在 TowerCtl.cpp 文件中，还要为新页的属性页映射添加一个条目。对于 Tower 控件，添加部分应该是这个样子：

```
BEGIN_PROPPAGEIDS(CTowerCtrl, 4)

    PROPPAGEID(CTowerPropPage::guid)

    PROPPAGEID(CTowerPropPage2::guid)

    PROPPAGEID( CLSID_CColorPropPage )

    PROPPAGEID( CLSID_CFontPropPage )

END_PROPPAGEIDS(CTowerCtrl)
```

记住，要在映射第一行中的 BEGIN_PROPPAGEIDS 宏中增加页数。新页数是 4。

4. 在文本编辑器或字符串编辑器中打开项目的 RC 文件，并添加两个字符串资源。第一个字符串用来标识系统注册表中已注册的属性页，第二个字符串用来保持在属性表对话框中出现的选项卡标签：

```
STRINGTABLE DISCARDABLE
```

```
BEGIN
```

```
IDS_TOWER                "Tower Control"
```

```
IDS_TOWER_PPG            "Tower Property Page"
```

```
IDS_TOWER_PPG2           "Tower Property Page 2"
```

```
IDS_TOWER_PPG_CAPTION    "Caption"
```

```
IDS_TOWER_PPG_NEWPAGE    "New Page"
```

```
END
```

5. 如果你在上一步中使用文本编辑器来创建新字符串资源，那么请向 Resource.h 文件为 IDS_TOWER_PPG2 和 IDS_TOWER_PPG_NEWPAGE 常数添加定义：

```
#define IDS_TOWER_PPG2    300

#define IDS_TOWER_PPG_NEWPAGE  301
```

如果你在上一步中使用的是字符串编辑器，那么添加这些行就不是必要的了，因为 Visual C++ 在你保存字符串资源的时候会自动写下定义。

6. 在文本编辑器中，打开 ClassWizard 为新属性页类创建的实现 CPP 文件。搜索这里的源代码，在每个阴影行中，用字符串标识符代替 0 参数。

```
BOOL CTowerPropPage2::CTowerPropPageFactory
    ::UpdateRegistry(BOOL bRegister)

{

    if (bRegister)

        return AfxOleRegisterPropertyPageClass(

            AfxGetInstanceHandle(),

            m_clsid, IDS_TOWER_PPG2);
```

```

else

    return AfxOleUnregisterClass(m_clsid, NULL);

}

.

.

.

CTowerPropPage2::CTowerPropPage2():

    COlePropertyPage(IDD,IDS_TOWER_PPG_NEWPage)

{

.

.

.

```

在做出这些改变之后，重建 Tower ActiveX 控件可以向控件资源添加新属性页。这里是新属性页的一个例子，这主要取决于你在本练习的第一

步 中 的 设 计。

第 10 章 用 ATL 编写 ActiveX 控件

第 9 章演示了使用 MFC 轻松地编写 ActiveX 控件，但需要指出的是，其方便性的代价是，可执行部分过于庞大。对于 ActiveX 控件来说，它适于小文件，特别是在万维网上应用程序时，这点非常重要。幸运的是，Visual C++ 在 MFC 之外还提供了其他工具。这一章就是要讨论一个非常流行的 MFC 替代工具，叫做 Active Template Library（活动模板库）。

Active Template Library 通常简称为 ATL，它提供了一套范围很广的 C++ 类模板（注：C++ 类模板也叫做参数化的类型，它是一种高级的宏格式，编译器根据传递到模板的参数，扩展为正常的类定义。模板与使用 #define 语句创建的宏不同，它扩展为普通的类型，使它们成为类型安全的类。编译器可以监视程序模板化的类的使用，就像它对普通的类一样，而且，可以正确识别出任何不一致类型的情况），用于开发可嵌入在 COM 应用程序中的服务器目标。在 ATL 的第 3 版中，ATL 辅助编写各种类型的 COM 目标，甚至可帮助生成包容器程序；在这一章里，主要讨论怎样使用 ATL 开发 ActiveX 控件。这里的目的是继续说明第 9 章所讨论的问题，说明用 MFC 之外的方法来开发制作 ActiveX 控件。

MFC 简化了 ActiveX 控件的开发，但不可避免地会出现大量的 MFC 库 DLL，这

对于制作网页是不合适的。为了解决这个问题，Microsoft 已经增强了 ATL，其方法是为 ActiveX 控件添加特殊的支持。当前，选择开发 ActiveX 控件项目所使用的工具一般要遵守的原则是：

- 在开发如第 9 章所述的 Game 程序的一般包容器应用程序时，考虑使用 MFC。当包容器本身同 MFC 有较强的联系时，MFC 作为开发工具是很有吸引力的，因为控件不负责加载库。也就是说，它早已做出了补偿。
- 应用在 Internet 的 ActiveX 控件则使用 ATL。

这一章将比前两章更详细地讨论 COM，还详细地介绍使用 ATL 编写复杂的 ActiveX 控件，而不把更多的精力放在 COM 上。ATL 不像 MFC，它让开发者离 COM 的表面更近，而不是在它下面。

ATL 和包容器应用程序

尽管 ATL 不是本章的主题，但它对包容器编程的支持值得说两句。ATL 在 COM 编程中的作用比较突出，但它倾向于对服务器的开发，而不是客户。库提供两类模板，分别叫做 CComPtr 和 CComQIPtr，这两类模板证明对编写客户端代码非常有用。这些模板生成灵活的接口指针，可设计来保证客户正确地释放控制接

口，即使有异常错误打断了正常的运行流程。

首先看问题，再看解决方案。包容器用控件的 QueryInterface 方法去调用控件的 QueryInterface 方法，来请求控件提供的接口指针。唯一的限制是当完成使用接口时，包容器就需要调用接口的 Release 方法，通知控件这个接口已不需要了。不能执行 COM 这一基本规则，将导致控件目标在系统的内存池中保存，甚至在包容器终止之后仍然存在。使用一个代码段可以演示包容器如何需要指向控件接口的指针，以 IOleObject 为例：

```
void Function1(IUnknown* pUnk)

{

    IOleObject *pOleObj;

    pUnk->QueryInterface(IID_OleObject,(PVOID*) &pOleObj );

    .

    .                                     // Use the IOleObject interface

    .

    pOleObj->Release();                  //When finished,release it
```

```
}
```

如果在最后一行调用 `IOleObject::Release` 让该接口终止前，有一个异常错误让程序结束的话，这段编码就会出现麻烦。如果把 `pOleObj` 当作智能指针就会解决这一潜在问题，因为这一指针的破坏程序调用该接口的 `Release` 方法，当 `IOleObject` 在使用中时，即使程序突然中止，它仍能够运行。

```
void Function1(IUnknown* pUnk)

{

    CComPtr<IOleObject> pOleObj;

    pUnk-> QueryInterface(&pOleObj );

    .

    .                                     // Use the IOleObject interface

    .

    .                                     // Release called automatically

}
```

两个程序段的明显区别是，修改过的版本在停止使用 `IOleObject` 时，没有明确调用 `IOleObject::Release`。当 `pOleObj` 脱离作用域时，因为 `Function1` 正常返回或出

现错误，指针的破坏程序就承担起释放接口的任务。

把 pOleObj 当作智能指针还有另外的优点。现在，检索接口指针 pUnk 将更简单，原因是 CComPtr 提供其本身的 QueryInterface 函数，从 pOleObj 中推出接口的标识符。通过确保标识符 (IID_OleObject) 和目标指针 (pOleObj) 指向同一接口 (IOleObject)，它使编码更方便，也增加了安全性。可避免犯以下这样可笑的错误：

```
pUnk-> QueryInterface( IID_ThisObject, (PVOID*) &pThatObject);
```

CComPtr::QueryInterface 能够通过应用程序 __uuidof 运算符让接口标识符不指向指针 (UUID 为 universally unique identifier(通用唯一标识符)的缩写)。在编译时确定标识符，__uuidof 取消了多余的编码同定义标识符的程序间的链接。总的效果是减少了运行大小。例如，在前面 Function1 的第一个版本中，可以用以下两行的任意一行 __uuidof 代替调用 QueryInterface。

```
pUnk-> QueryInterface(__uuidof(IOleObject),(PVOID*)&pOleObj);
```

```
pUnk-> QueryInterface(__uuidof(pOleObj), (PVOID*)&pOleObj);
```

IUnknown 是客户最终很少需要的接口，但它是客户能肯定 ActiveX 控件可支持的唯一接口。获得一个接口可分两个步骤：首先调用程序得到一个 IUnknown 指针（在分段代码中的 pUnk），然后调用 IUnknown::QueryInterface 检索实际需要的接口指针。ATL 提供另一个智能指针把这两个步骤合二为一。CComQIPtr 模板

在构造器包含对 `Iunknown::QueryInterface` 的调用。以下是一个 ATL 的 `CComQIPtr` 模板的简要列表，显示构造器怎样得到所要的接口指针和破坏程序在以后如何释放它。

```
template <class T, const IID* piid = &_uuidof(T)>

class CComQIPtr

{

public:

    T* p;

    .

    .

    .

    CComQIPtr(Iunknown* lp)

    {

        p = NULL;
```

```
if (lp != NULL)
```

```
    lp->QueryInterface( * piid, (void **) &p );
```

```
}
```

```
~ CComQIPtr()
```

```
{
```

```
    if (p)
```

```
        p->Release();
```

```
}
```

```
    .
```

```
    .
```

```
    .
```

```
}
```

CComQIPtr 模板为调用程序提供了一个更为简洁的方法从控件得到一个接口指

针。如果控件不支持所需接口，指针类的 `p` 成员为 `NULL`。

```
CComQIPtr <IOleObject> pOleObj;
```

```
pOleObj = pUnk;
```

```
if (pOleObj.p)
```

```
{
```

```
    .
```

```
    .
```

```
        // Use the IOleObject interface
```

```
    .
```

```
}
```

调用程序可使用 `CComQIPtr` 为除了 `IUnknown` 之外的任何接口生成指针。要为 `IUnknown` 接口生成指针可使用 `CComPtr`。

`CComQIPtr` 和 `CComPtr` 模板受到 Standard Template Library（标准模板库）中的 `auto_ptr` 智能指针类的启发，它确保了通过 `new` 分配的目标返回到程序的空闲空间。尽管模板主要对使用 ActiveX 控件的客户应用程序有好处，它们也可很容易地为控件本身服务，特别是控件聚集或包含另一个控件时。本章稍后会介绍一个

例子，来说明 ActiveX 控件如何利用智能指针。

ATL 对客户编程的支持是非常被动的，它只包括项目中的源代码文件。这些文件可通过其 Atl 前缀识别，存储在 VC98\ATL\Include 文件夹中。例如，一个使用 CComQIPtr 和 CcomPtr 的包容器项目访问 MSDN 的 Samples\VC98\ATL 文件夹中的 AtlCon 项目。ATL 对开发类似 ActiveX 控件的服务程序的帮助远不止这些，本章其余部分将集中介绍 ATL 的这一方面。

ATL 和 ActiveX 控件

从技术上说，ActiveX 标准对控件的能力要求不高。作为 ActiveX 控件运行，一个组件只需要实现 Iunknown，可嵌入，而且可以自我注册。其他诸如属性、方法和事件均是可选的。由于过于微小，而且是隐性的，一个 COM 目标在学术之外没有讨论的价值。实际上，一个 ActiveX 控件维持一组数据；激发事件；并支持足够的接口，让用户应用程序和它交互。ActiveX Test Container（ActiveX 测试包容器）功能对于 ActiveX 控件来说就像是石蕊试纸。本章开发的两个控件项目能够成功地在 Test Container 中运行，因为它们插入了在表 10-1 描述的接口。所列的接口是 ActiveX 控件为同公布的 Microsoft 指南相兼容所必须提供的支持。比较表 10-1 和表 8-2，其中列出了要嵌入 ActiveX 控件，一个包容器必须支持的接口。

如果希望 ATL 很容易地创建一个只有几 K 大小的 ActiveX 控件，一定会很失望。库提供了几种方法来减少运行量，但实现表 10-1 所列的接口会加入大量的编码。创建短小编码的唯一方法就是直接使用 COM 编程，而不借助于 ATL 和其他支持库（Visual C++ 的 Samples\VC98\ATL 文件夹中有一个叫做 Minimal 的 ATL 项目范例，创建了一个小 COM 服务器程序只有 5600 个字节。Minimal 服务器缺少几个基本特征，并不是为了创建真正的 ActiveX 控件）。与 MFC 一样，ATL 带来了方便，也带来了大量冗长无用的编码。做一些额外的工作，就可以得到一个简单的无窗口的 ActiveX 控件，例如只有 40KB 大小或更小。本章稍后会介绍几种帮助减少用 ATL 控件大小的方法。

表 10-1 ActiveX 控件必须支持的接口，以同指南相兼容

接口	说明
<i>IOleObject</i>	与控件的客户站点通信所需要的，事件除外。事件通过 IConnectionPointContainer 接口进行处理，在下面将说明
<i>IOleInPlaceObject</i>	通过在适当位置激活的控件来实现，并提供它们自己的用户接口。需要 IOleObject 的支持
<i>IOleInPlaceActiveObject</i>	只有在控件提供用户接口，而且支持 IOleInPlaceObject 的情况下才需要
<i>IDataObject</i>	以某种方式传送数据到容器的控件需要它，如共享内存或文件。IDataObject 提供 COM 的 Uniform Data Transfer（统一数据传输）的手段，这是一种协议，用于为任何数据类型的交换建立规则
<i>IViewObject2</i>	通过显示窗口的可见控件来实现

续表

<i>IDisPatch</i>	具有定制方法或客户可以通过 <code>IDispatch::Invoke</code> 来访问的控件需要它
<i>IConnectionPointContainer</i>	启动事件的控件需要它。这个接口为客户枚举控件目标可以启动的事件
<i>IConnectionPoint</i>	支持 <code>IConnectionPointContainer</code> 的控件需要
<i>IProvideClassInfo</i>	由包含类型库信息的控件实现，这意味着大多数 ActiveX 控件。通过其 <code>GetClassInfo</code> 方法，接口提供了到 <code>ITypeInfo</code> 实现的指针，从中，客户可以抽取控件的类型信息。类似的

续表

	<code>IProvideClassInf</code>	接口是一个扩展接口，它扩充了 <code>GetGUID</code> 方法，这样，客户便可以为控件的默认事件来检索到标识符的指针
<code>IPersistStorage</code>		可以保存和从包容器提供的 <code>IStorage</code> 实例加载的控件需要它
<code>IClassFactory</code>		实例化一个要求的类目标，将返回给它一个指针。此目标是通过在系统注册表中注册的类标识符来进行标识
<code>IClassFactory2</code>		与 <code>IClassFactory</code> 相同，但是，它还增加了许可证的支持（参见第 9 章的“许可证”一节）

另一方面，当考虑到与控件有关的大量 MFC 库 DLL 时，用 ATL 编写的 ActiveX 控件比用 MFC 编写的相应的 ActiveX 控件要小得多。用 ATL 库构造的 ActiveX 控件可使用 MFC，但这样用 ATL 就没有意义。如果计划用 MFC 编写控件，可以首选 MFC ControlWizard，只有在创建智能指针类 `CComQIPtr` 和 `CcomPtr` 才会使用 ATL。使用 ATL 的主要好处是，它所创建的组件不需要 MFC 和 C 运行库。

这样的组件可放在 Internet 上独立存在，而不用依靠用户机器上的支持文件。

与 MFC 一样，ATL 为控件包含的每一个目标创建单一的代表性的类。这些类来源于目标支持的所有接口，这是 MFC 通过嵌套类玩的一个小把戏。ATL 可通过使用多样继承更灵活地达到同样的效果，这些继承是几个基类发展出来新的类，新类从每个类继承了成员数据和函数。基类列表叫做继承列表，看起来像以下的类声明：

```
class CMyClass:  
  
public CClass1,  
  
public CClass2,  
  
public IInterface1,  
  
public IInterface2,  
  
.  
  
.  
  
.  
  
{
```

ATL 为开发 ActiveX 控件提供最重要的服务是许多库的实现代码，这些代码用于控件通常支持的许多 COM 接口。以类模板的形式，有了库代码，就不用再编写程序来支持通用的接口了。ATL 为列在表 10-1 中的每一个接口提供了模板化的实现代码，并以接口名加上 Impl 后缀来命名该模板。例如，IQuickActiveImpl 模板为 IQuickActive 方法提供如 SetExtent 和 GetExtent 这样的代码。与 COM 要求的那样，一个支持接口的所有方法都列出来，但不一定使用。许多方法只叫做 ATLTRACE_NOTIMPL 宏，为 Visual C++ Output（输出）窗口的 Debug 选项卡编写很少的记录信息，并把 E_NOTIMPL 返回给调用程序。如果想让 ActiveX 控件提供这样的方法，则必须自己添加代码。

除了代码库外，ATL 还提供 AppWizard，以开始创建一个服务器项目，还有一个向导，用来帮助生成 ActiveX 控件所需的类代码。你会感觉到，使用 ATL 并不比使用 MFC 困难，ATL 需要编程人员比使用 MFC 时多注意项目细节，并处理更多的 COM 问题。在展示 ATL 的演示项目前，先停顿一下，了解一下后面会遇到的有关 ATL 3 个方面的背景信息：接口映射，目标映射和线程模型。

接口映射

接口映射是大多数 QueryInterface 函数的通用模式。COM 目标的所有接口都支持这一函数，让包容器应用程序在任何接口中调用 QueryInterface，并接收其他任何接口的指针。如果控件不支持所需的接口，它返回的值为 E_NOINTERFACE。

以下程序段演示了一个典型的 QueryInterface 函数在不使用 ATL 时看起来会是什么样。参数 riid 指调用程序所需的接口标识符，参数 ppvObject 接收接口指针：

```
STDMETHODIMP CMyClass::QueryInterface( REFIID riid,PVOID *ppvObject )

{

    switch( riid)

    {

        case IID_UNKNOWN:

        case IID_IInterface1:

            *ppvObject = (IInterface1*) this;

            break;

        case IID_IInterface2:

            *ppvObject = (IInterface2* this;
```

```
break;
```

```
case IID_IInterface3:
```

```
*ppvObject = (IInterface3*) this;
```

```
break;
```

```
default:
```

```
*ppvObject = 0;
```

```
return E_NOINTERFACE;
```

```
}
```

```
(IUnknown*) *ppvObject->AddRef();
```

```
return S_OK;
```

```
}
```

这类重复的模式很容易用宏来实现，了解宏，便可对接口映射的物理分布有清楚的认识。当用户调用类目标的 `QueryInterface` 方法时，接口映射通过 ATL 的 `CComObjectRootBase::InternalQueryInterface` 函数来确定调用的路线，在 `AtlCom.h` 文件中，如下处理：

```
static HRESULT WINAPI InternalQueryInterface(PVOID pThis,  
  
const_ATL_INTMAP_ENTRY *pEntries, REFIID riid, PVOID *ppvObject)
```

同以前一样，`riid` 确认接口，`ppvObject` 接收它的指针。该函数的第二个参数 `pEntries` 指向接口映射的开始，它包含 `_ATL_INTMAP_ENTRY` 数组结构，数组中的每一个结构包含一个接口标识符，一个 `DWORD` 变量，还有一个函数指针：

```
struct _ALT_INTMAP_ENTRY  
  
{  
  
    const IID*                piid;                //Interface identifier  
  
    DWORD                    dw;                    //Offset  
  
    _ATL_CREATORARGFUNC*    pFunc;                Function pointer  
  
};
```

pFunc 的值决定 InternalQueryInterface 如何解释 dw 的值。如果 pFunc 是 ATL_SIMPLEMAPENTRY(定义为 1), dw 则包含类目标的偏移值, 允许 InternalQueryInterface 完成类似以下的调用:

```
*ppvObject=pThis+pEntries->dw;
```

```
*ppvObject->AddRef();
```

pEntries 已经增加值, 以指向数组中接口 _ATL_INTMAP_ENTRY 结构。如果 pFunc 的值大于 1, 则 InternalQueryInterface 调用函数, 把 dw 作为参数。

```
pentries->pFunc( pThis, riid, ppvObject, pEntries->dw);
```

所调用的函数负责编写所需的指向 *ppvObject 的接口指针。

pfunc 成员也可 NULL。这个特殊值为数组中最后的 _ATL_INTMAP_ENTRY 结构所保留, 作为接口映射的结尾标记。你没有必要过分担心这一部分, ATL 通过特殊的宏来确保最后结构。另外, 在安排映射数组的 _ATL_INTMAP_ENTRY 结构方面没有限制。以下是前面所引述的 CmyClass 的接口映射。注意, 一个空的 _ATL_INTMAP_ENTRY 结构是如何结束数组的:

```
{&IID_Interface1, 0, 1}
```

```
{&IID_Interface 2, 4, 1}
```

```
{&IID_Interface 3, 8, 1}
```

```
{0, 0, 0 }
```

在使用 ATL 的代码中，接口映射由一系列的 COM_INTERFACE 宏构成，每个宏扩展为一个 ATL_INTMAP_ENTRY 结构。ATL 提供 17 种不同的 COM_INTERFACE 宏，通过其后缀 _ENTRY 和 _TEAR_OFF 来表明宏要处理的接口类型。宏的名称由 COM_INTERFACE 加上后缀字符串构成。表 10-2 将描述 COM_INTERFACE 宏，并解释什么时候使用它们。为简明起见，第一列只列宏后缀名。要想详细了解宏，请参阅在线帮助“COM_INTERFACE_ENTRY Macros”。

表 10-2 ATL 的 COM_INTERFACE 宏

COM_INTERFACE 宏	说明
_ENTRY	暴露衍生类的接口
_ENTRY_IID	与 COM_INTERFACE_ENTRY 相同，但是，压指定接口的标识符
_ENTRY2	对于从两个或多个双接口衍生的类，要解析其接口的不明确性，应当提供到 Idispach 接口的指针。双接口在本章的后面介绍
_ENTRY2_IID	与 COM_INTERFACE_ENTRY2 相同，但是，它指定接口的标识符
_ENTRY_IMPL	是 COM_INTERFACE_ENTRY 的另一种方法

续表

<code>_ENTRY_IMPL_IID</code>	与 <code>COM_INTERFACE_ENTRY_IMPL</code> 相同，但是，它指定接口的标识符。此宏和 <code>COM_INTERFACE_ENTRY_IMPL</code> 在 ATL 版本 3 中是过时的，要使用 <code>COM_INTERFACE_ENTRY</code>
<code>_ENTRY_FUNC</code>	在 ATL 处理 <code>QueryInterface</code> 时，指定获得控件的挂钩函数。挂钩函数可以忽略此进程，方法是返回 <code>E_NOINTERFACE</code> ，这样，便可以隐藏 ATL 要返回的接口
<code>_ENTRY_FUNC_BLIND</code>	与 <code>COM_INTERFACE_ENTRY_FUNC</code> 相同，只是要查询引起挂钩函数调用的所有接口

续表

<code>_ENTRY_TEAR_OFF</code>	为快用 (tear-off) 接口声明 COM 映射项, 这种接口只有在客户通过 <code>QueryInterface</code> 请求时, 才进行实例化。这种接口只在需要时才占用内存, 这使它更适合于 <code>ISupportErrorInfo</code> 这样的接口, 因为在控件的生存期, 这样的接口也可以不使用。其缺点是, 它比常规接口的开销要稍大些 实现这种接口的类必须从 <code>CComTerzaOffObjectBase</code> 衍生出来, 而且具有自己的 COM 映射
<code>_ENTRY_CACHED_TEAR_OFF</code>	与 <code>COM_INTERFACE_ENTRY_TEAR_OFF</code> 宏相同, 不同的是, 在第一个实例后, 保存此接口数据。如果快用接口是实例化的, 则有效地缓存可以将它转换为常规接口
<code>_ENTRY_AGGREGATE</code>	为聚合目标提供的接口声明 COM 映射。此宏查询转发到聚合目标的 <code>IUnknown</code> 接口的接口标识符。本章的后面将讨论有关聚合的问题

续表

<code>_ENTRY_AGGREGATE_BLIND</code>	与 <code>COM_INTERFACE_ENTRY_AGGREGATE</code> 相同，不同的是，所有查询都转发到指定的 <code>Iunknown</code> 接口
<code>_ENTRY_AUTO- AGGREGATE</code>	如果提供 <code>IUnknown</code> 指针，则与 <code>COM_INTERFACE_ENTRY_AUTOAGGREGATE</code> 相同，否则，此宏自动插件由给定的类标识符指出的聚合
<code>__ENTRY_AUTO- AGGREGATE_BLIND</code>	与 <code>COM_INTERFACE_ENTRY_AUTOAGGREGATE</code> 相同，除非提供了 <code>IUnknown</code> 指针，在这样的情况下，所有查询都转发到 <code>Iunknown</code> 接口，如果 <code>Iunknown</code> 指针没有提供，则宏创建由给定的类标识符指出的聚合

续表

<code>_ENTRY_CHAIN</code>	允许继续处理指定的基本类的 COM 映射。基本类必须在当前类的继承性列表中出现。也就是说，它必须是当前类的基础。 <code>COM_INTERFACE_ENTRY_CHAIN</code> 不能是 COM 映射中的第一项
<code>_ENTRY_BREAK</code>	在请求指定的接口时，调用 <code>DebugBreak</code> 。使用此宏来触发调试器断点（调试器在第 11 章中介绍）
<code>_ENTRY_NOINTERFACE</code>	在查询指定的接口时，返回 <code>E_NOINTERFACE</code> ，并结束 COM 映射处理。此宏禁用此接口，防止它被 COM 映射后的任何 <code>COM_INTERFACE</code> 宏进行处理

ATL 指作为 COM 映射的源代码的宏系列，这一名称常常和宏所创建的接口映射混用。在 ATL 中，COM 映射以调用 `BEGIN_COM_MAP` 宏开始，以 `END_COM_MAP` 宏结束。在它们之间有一系列的 `COM_INTERFACE` 宏，都有类支持的接口。

```
BEGIN_COM_MAP(CMyClass)

    COM_INTERFACE_ENTRY(IMyClass)

    COM_INTERFACE_ENTRY(IDispatch)

    COM_INTERFACE_ENTRY2(IPersist, IpersistStreamInit)

    .

    .

    .

END_COM_MAP()
```

映射从 MFC 的标准消息映射中借鉴了许多方法（MFC 提供自己的宏来创建接口映射，这一点在第 9 章没有叙述。在 MFC 中，接口映射以 BEGIN_INTERFACE_MAP 宏开始，以 END_INTERFACE_MAP 宏结束）。COM 映射的结构顺序并不重要，列表中的第一个接口必须使用简单的映射项，例如 COM_INTERFACE_ENTRY 或其他 COM_INTERFACE 宏，用 ATL_SIMPLE_MAPENTRY 的 pFunc 值扩展至 ATL_INTMAP_ENTRY 结构。这个需求阻止了 ATL 使用映射中的第一个接口来响应目标的 IUnknown 接口的请求。

目 标 映 射

一个 ActiveX 控件可能包含几个目标，每个目标都通过一个类来表示，提供自己的接口映射。图 10-1 显示了控件、控件所包含的目标和目标实现的接口之间的层

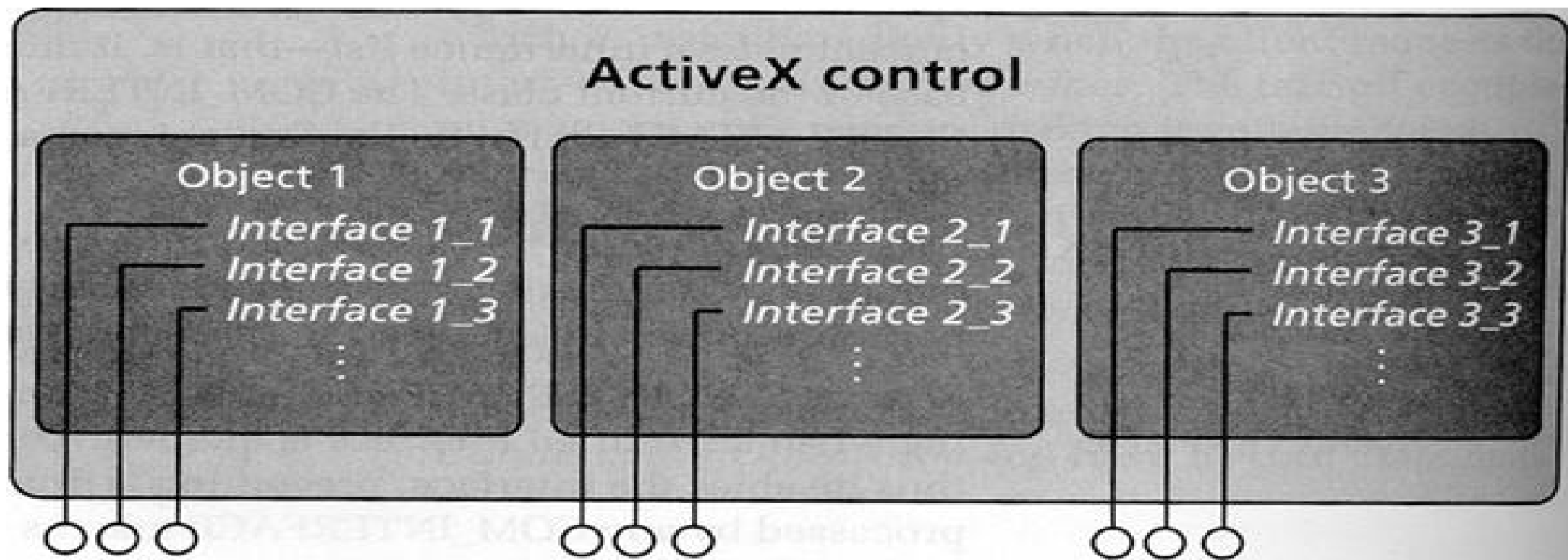


图 10-1 ActiveX 控件的元素

次关系。同接口映射的概念一样，目标映射记录了一个控件的目标和目标相关的类标识符（CLSID）。ATL 把目标映射安排成一个 `_ATL_OBJMAP_ENTRY` 结构数组，每一个定义一系列求助功能。以下是一个简单的 `_ATL_OBJMAP_ENTRY` 结构数组：

```
struct _ATL_OBJMAP_ENTRY
```



```
{  
  
    HRESULT UpdateRegistry( BOOL bRegister );  
  
    HRESULT GetClassObject( void* pv, REFIID riid,PVOID* ppv);  
  
    HRESULT CreateInstance( void* pv, REFIID riid,PVOID* ppv);  
  
    LPCTSTR GetObjectDescription();  
  
    HRESULT RevokeClassObject();  
  
    HRESULT RegisterClassObject( DWORD dwClsContext,DWORD dwFlags);  
  
};
```

当客户第一次请求一个类目标时，便调用 `GetClassObject` 函数，生成一个目标实例，向调用程序提供了一个指针，来指向所需要的 `IClassFactory` 或 `IClassFactory2` 接口。函数在目标映射结构中存储类工厂（class factory）指针，让下一个调用类目标新实例的需求尽快完成。因为类目标在栈和堆上进行实例化，而不是在控件的静态数据中，所以，用 ATL 构造的 ActiveX 控件不需要同 C 运行库进行链接。不使用 C 运行库和静态构造器，可以保证最终控件的可执行尺寸很少，不用对链接的 C 库进行额外的初始化。

线程模型

ATL 支持四种线程模型，分别名为单一（single）、公寓（apartment）、自由（free）和两种（both）。线程模型描述了一个控件实现的线程安全类型和程度，尽管对于任何客户应用程序，无论线程安排是怎么样的，都可安全地进入任何线程模型生成的控件。如果客户的线程与任何线程模型都不兼容，则 COM 便会在二者之间实现自身，以保证线程的安全通信。有了这个保证，为自己的控件选择哪种线程模型就会很简单了。对于每一种选择，都有其各自的优缺点，通常，性能与代码的大小是相对的，效率与简化是相对的。ATL 添加到项目中的代码支持用户选择的线程，所以，你只需要保证自己编写的代码也与所选择的模型相符。在本节中，将解释 4 种模型之间的不同之处，以帮助你决定哪一种最适合你的项目。

从客户的角度来看，很容易想象线程是怎样的。开始，我们来看看应用程序于容器的线程模型，然后，检查每种线程模型对应用程序嵌入的 ActiveX 控件的影响如何。线程模型并不难理解，但是，其中的规则却有些不好理解。在本章的后面，有一个示例项目，它应用这里讨论的一些理论，并说明一种控件在不同的线程模型下是如何起作用的。

单一线程

单一线程模型是四种线程中最简单的，因为它不需要任何控件来防卫其数据的模拟使用，即使是静态数据。这种模型允许客户创建一个控件项目的任何数目的实

例，但是，它限制在一个线程中，所有客户都访问这些实例。单一线程不限制客户可以运行的线程数目，而且，确实，ActiveX 控件应当假定其许多客户都是多线程的。这种模型只是指示对目标接口的所有调用都从客户的线程中产生，这个线程是第一次调用 CoInitializeEx 来初始化 COM 框架的线程。由于 ActiveX 控件不需要耗费其他精力来保证线程对其接口或其他数据的访问是安全的，所以，单一的线程是在四种线程模型中尺寸最小的。只要客户符合此模型的限制条件，在客户和服务端之间进行通信就是直接而快速的。

但是，考虑一下，在客户使用相同的 ActiveX 控件时，在两个线程之间会发生什么。线程 A，它不是客户的主线程，通过 CoInitializeEX 来初始化 COM，然后调用 CoCreateInstance，来创建控件目标的一个实例。线程 B 随后需要此控件的服务，与线程 A 一样，它创建一个新的线程实例。但是，在这一次，COM 不像线程 A 调用 CoCreateInstance 那样，返回直接指向控件代码的接口指针。而是，返回的指针引用一个在线程 B 上运行的不可见的代理服务目标。在客户使用指针从

线程 B 来调用一种方法时，代理服务便会发送一条消息给线程 A 上运行的存根目标。存根依次直接调用到相同线程上的 ActiveX 控件，然后，通过另一条消息传递返回值给代理服务。消息的交换在线程之间进行，而且保证 ActiveX 控件总是在线程 A 的环境中执行，也就是第一次初始化 COM 的线程。

从一个线程中重新确定一个调用的路线，并在另一个线程中完成它，这便是线程间调度，从概念上来说，它与第 8 章讨论的进程间调度类似。主要的不同是代理服务和执行线程间调度的存根目标是由 COM 设置的隐藏目标，而不是在单独的进程中安装的动态链接库。窗口之间的消息传送与远程过程调用类似，它在客户进程内的代理服务和进程外的存根之间来回传递。线程之间的调度通常要比进程之间的要快，但是，与直接调用目标相比，交换消息和切换线程的环境还会大大减慢对目标实例的访问速度。每次客户从线程 B 调用控件的一种方法时，为了调度到对线程 A 的调用，还要重复相同的繁琐过程。最糟糕的是，线程 A 可能会非常忙，没有在其消息循环中，这样，代理服务提出的消息就必须进行等待，直到其抽取和路由到存根。对线程 A 生成的客户的所有方法调用，都直接到控件中，而且不进行调度，这使得线程 A 具有了特殊的权限，而其他使用此控件的线程都没有授予此权限。只有在客户从不是第一个通过 CoInitializeEx 用 COM 注册自身的线程调用控件时，单一线程模型才会得到性能上的损失。

公寓线程

公寓线程模型为了消除线程间的调度，做了很多的工作。在这种安排方法下，所有的客户线程都享有特权，而且能够直接与 ActiveX 控件的实例交互，而不需要首先通过代理服务和存根服务。换句话说，每个公寓线程都像是前面讨论的线程 A，没有一个线程像线程 B 那样。需要控件服务的每个线程都首先调用 CoInitializeEx，与以前一样，然后调用 CoCreateInstance，来创建一个控件实例。在此公寓线程中，返回的接口指针直接指向接口实例化目标中的 V 表，而且不

进入代理服务。每个线程都具有自己的目标实例，只要线程只访问它创建的实例，就不会有调度发生。

有时，人们会产生混淆，这是因为有两类公寓线程。ATL 所说的“公寓”模型更像是单一线程的公寓模型，通常缩写为 STA。这种模型在前面已经介绍了：每个线程都有一个目标实例，每个线程都调用自己的实例。多线程的公寓模型，即 MTA，就是 ATL 称之为自由模型的公寓模型，稍候我们将介绍这种模型。公寓是一个抽象的概念，它与现实世界进程和线程的处理方法的关系不大，所以，不要浪费很多时间来试图将它可视化。可以这样来看，一个进程就像是一个建筑物，其中，线程表示单独的房间。这种模型将一个目标的单一实例与可以直接安全调用到那个实例的一个或多个线程进行折衷。

尽管我们通常要讲到 STA 或 MTA 客户应用程序，模型的指定可以更精确地应用到应用程序中的线程，因为进程可以包含 STA 和 MTA 线程。在客户线程调用 CoInitializeEx 时，它将传递一个值，这个值指定此线程是为哪种公寓模型而设计的。COINIT_APARTMENTTHREADED 的值在 STA 模型下注册线程，也就是单一线程公寓的方法。COINIT_MULTITHREADED 的值注册线程作为多线程公寓的一部分。一个进程可以包含任何数目的单一线程公寓，但只能包含一个多线程的公寓，在多线程公寓中，任意数目的线程都可以属于它。在公寓内部调用线程时，从目标检索的接口指针不进行调度，但是，在使用其他公寓中的线程时，总是要进行调度。在谈到单一线程公寓模型时，公寓和线程这两个词经常互换使用，不会引起混乱，但是，在实际应用程序中，在应用到多线程公寓模型中时，

这就不对了，下面将讨论这个问题。

自由线程

自由线程是多线程公寓模型的另一个名称。在多线程公寓中的线程可以安全地共享目标的单一实例所提供的接口指针，而不管创建实例的线程是什么。COM 在此方法中不起作用，而且，在公寓的限制内，不发生调度调用。然而，与在单一线程模型中一样的是，在线程调用到在另一个公寓中实例化的目标时，还需要进行调度。

这似乎是足够了，但是，自由线程将重大的责任都放到了开发者身上。选择支持自由线程模型意味着，在任何时间，要编写任意数目的线程安全访问的代码。下一节将介绍每个线程对 ActiveX 控件施加影响的一些需求。

为 ActiveX 控件选择线程模型

理解 COM 的客户端口的线程模型，便可以选择最适于 ActiveX 控件的线程模型。幸运的是，服务器中的线程涉及到更到的代码，而且，理论概念更少。你只需要为自己的控件选择一种线程模型，并根据情况来编写代码，保证 COM 将通过调度来解决任何不匹配的问题。客户线程与 STA 或 MTA 模型是一致的，但是，ActiveX 控件采用了前面列出的四种模型之一：单一（非线程），公寓（STA），自由（MTA）或两种。两种线程模型意味着 STA 和 MTA，也就是说，必须写控件来适应客户线程的两种可能模型，而不需要 COM 来调度其交互作用。

我们已经看到了，在调用 `CoInitializeEx` 时，客户线程如何对 COM 识别其模型，但是，ActiveX 控件的主线程是接收客户调用的线程，它并不需要注册自身。这是很有意义的，因为毕竟我们讲过，只有一个线程沿着程序逻辑流从客户到 ActiveX 控件，而且再回来，而不通过调度程序进行传递。控件通过其系统注册表中的条目来来识别其线程模型，这些条目可能是 `apartment`(公寓)，`free` (自由) 或 `both` (两种)。如果这个条目不存在，COM 就假定控件符合单一线程模型。本章中以后的项目演示了 ActiveX 控件如何在注册表中指定其线程模型。

学完了线程模型这些难以理解的内容后，我们至少可以处理一种非常重要的问题了：ActiveX 控件如何提前知道客户使用的哪种线程模型？答案相当简单：它并不知道，而且，它也不需要知道。你为控件选择的线程模型告诉 COM 客户设计用来处理的客户线程的种类，而不需要进行调度。COM 识别何时发生调度，并且，只有在发生任何调度时，才透明地设置调度。例如，实例化控件的 STA 客户线程标记 `apartment` (公寓)，直接与控件实例进行交互。然而，在 MTA 线程实例化控件时，COM 必须调度所有的调用，以保证客户和控件在相同的线程上运行。在注册公寓线程模型时，控件已经通知 COM，每个实例只可以适应单一线程上的调用。调度可以保证所发生的事情。表 10-3 概括了在什么条件下 COM 调度客户及其嵌入的控件之间的调用。

表 10-3 COM 在客户和控件之间调度的条件

此模型的客户线程	访问此模型的 ActiveX 控件	需要调度吗？
STA	单一	第一个初始化 COM 的线程不需要，所有其他线程都需要
MTA	单一	需要
STA	公寓	不需要
MTA	公寓	需要
STA	自由	需要
MTA	自由	不需要
STA	两种	不需要
MTA	两种	不需要

哪种线程模型最适于控件取决于你提前使用的客户应用程序的种类，以及你要承担多少额外的工作，以保证线程的访问安全。为了实现线程安全，单一的非线程模型根本不需要额外的代码，因为只有一个客户线程可以直接访问控件的所有实例。另一方面，自由线程模型最适于这样的情况：你提前知道 MTA 应用程序将独占使用控件。然而，很少能够得到这样的保证，除非是你自己编写客户程序。现在，由于大量的客户应用程序都符合 STA 线程，所以，对于设计用来服务于许多不同的客户的 ActiveX 控件来说，公寓模型通常是最佳选择。Internet Explorer 和 Netscape Navigator 是 STA 应用程序，使用 MFC 的包容器程序也是，另外，还有使用 Visual Basic 5 及以后版本编写的程序也是。Microsoft Transaction Server 也符合 STA 的规则，所以，控件如果支持 MTS，则应当使用公寓模型。

线程安全性是在公寓模型下易于编程的，它仅需在安全保护下不同时写静态数据，通常通过关键区域或一些类似的机制。如果你要确保快速访问控件而不管客户机的线程模型，选择两种（both）模型。顺从 STA 和 MTA 访问而不配置额外工作，特别是运行在多线程下的控件。要研究一下 MTA 线程的复杂性，细读 David Platt 在 Microsoft Systems Journal 第 12 卷第 8 页的文章即可。在目录标签的期刊条目下的 MSDN 在线帮助中可找到此文章。

示例 1：Pulse ActiveX 控件

在本节中提出的 Pulse 控件说明了如何使用 ATL 来生成简单的 ActiveX 控件。

Pulse 是选择性的，它从库中获取。这里的目的不仅仅是演示 ATL，还要给出减少控件大小的方法，而且不修改 ATL 源代码，或者，采取直接的 COM 指令。通过结果，你应当理解在使用 ATL 创建时，一个 ActiveX 控件的尺寸可以达到多小。

Pulse 只是一个计时器，在编程时，它以一定的间隔来启动事件，这样，它的作用就很像是我们在建立 Hour 程序时使用的 IETimer 控件。与 IETimer 一样的是，Pulse 是完全自给自足的，它既不使用 MFC，也不使用 C 运行库。但是，在 37KB，Pulse 没有 IETimer.ocx 的一半大小，而且还提供相同的服务。Pulse 这样小的一个主要原因是，如果不显示窗口的话，它运转起来是不可见的。

Pulse 控件包含一个有名为 CPulseCtl 的类控制的单一目标。除了实现表 10-1 中列出的接口外，此目标还提供了这些控件元素：

- 包含脉冲间隔的适当的变量，以毫秒为单位。
- 允许包容器启动和结束事件启动的方法。
- 每次间隔过去时通知包容器的事件。

在这里介绍了 10 步，来演示 Pulse 项目如何获取格式。本讨论采用常规的方法来使用 ATL 创建 ActiveX 控件项目，而且，并不专门针对不可见的控件，如 Pulse。这些不要使你感觉到我们在这里所做的事情有多么困难。使用 ATL 创建简单的

ActiveX 控件非常简单。

步骤 1：运行 ATL COM AppWizard

使用 ATL 创建的 ActiveX 控件项目开始总是使用 ATL 的向导，ATL COM AppWizard。单击 File（文件）菜单上的 New（新建）命令，选择 ATL COM AppWizard，如图 10-2 所示，并输入项目名 Pulse。

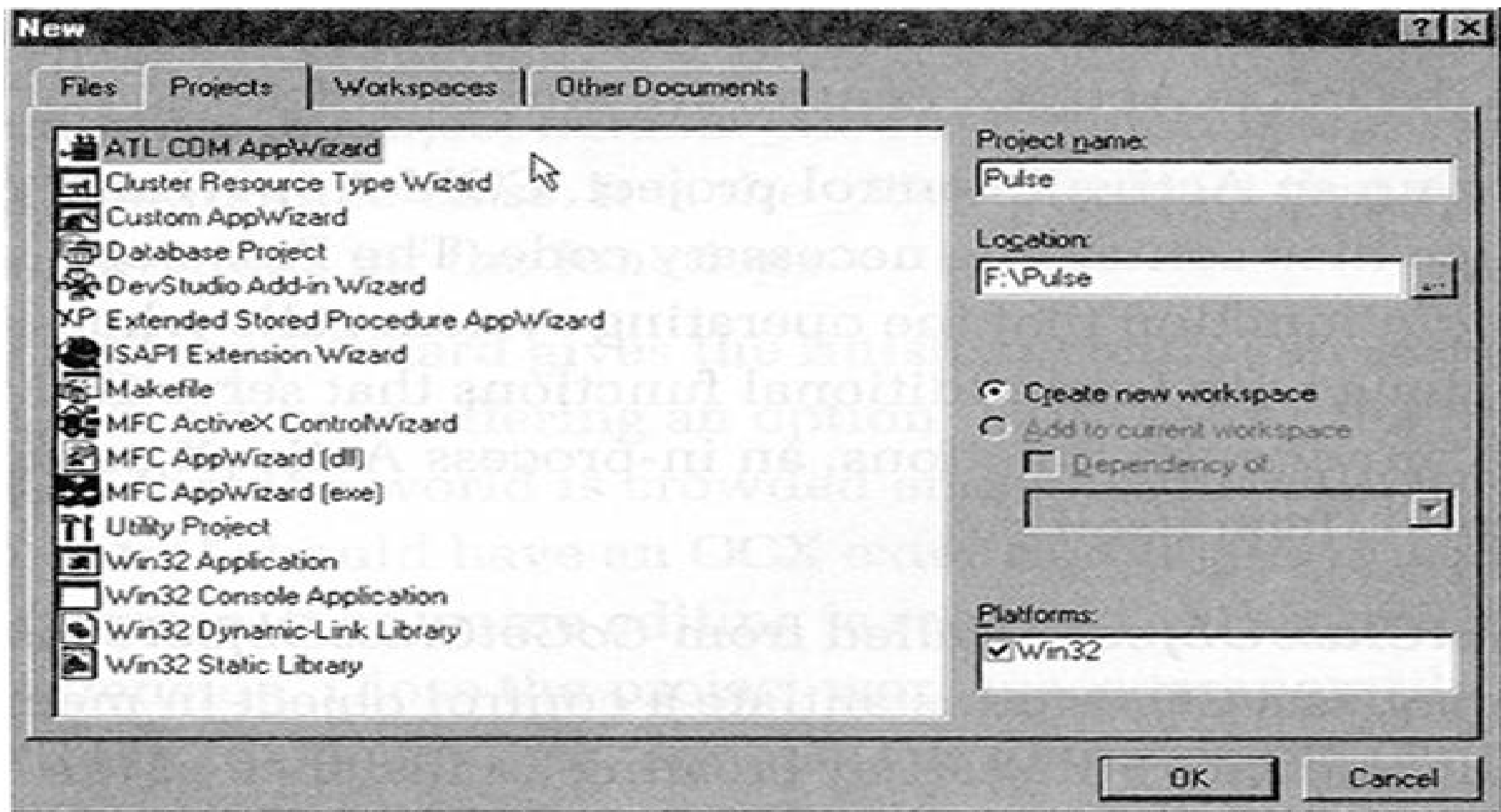


图 10-2 启动 ATL COM AppWizard

在出现一个对话框列出控件选项时（图 10-3），单击 Finish（完成）按钮，来接受默认设置。这些设置指定新的控件作为不使用 MFC 的动态链接库来运行。对于标签为 Allow Merging Of Proxy/Stub Code（允许代理和存根代码的合并），它只对 DLL 服务器类型可用。选择这个选项告诉 COM AppWizard，来设置项目来链接调度代码到控件的可执行映像中，产生一个单一的 DLL 文件，其中包含控件及其代理/存根指令。清除选项告诉 COM AppWizard 来编写调度代码到单独的名为 DllData.c 文件，这样，可以减少最终可执行文件的整个大小。在本练习的第 5 步中，我们将更多地讨论控件的代理和存根代码。

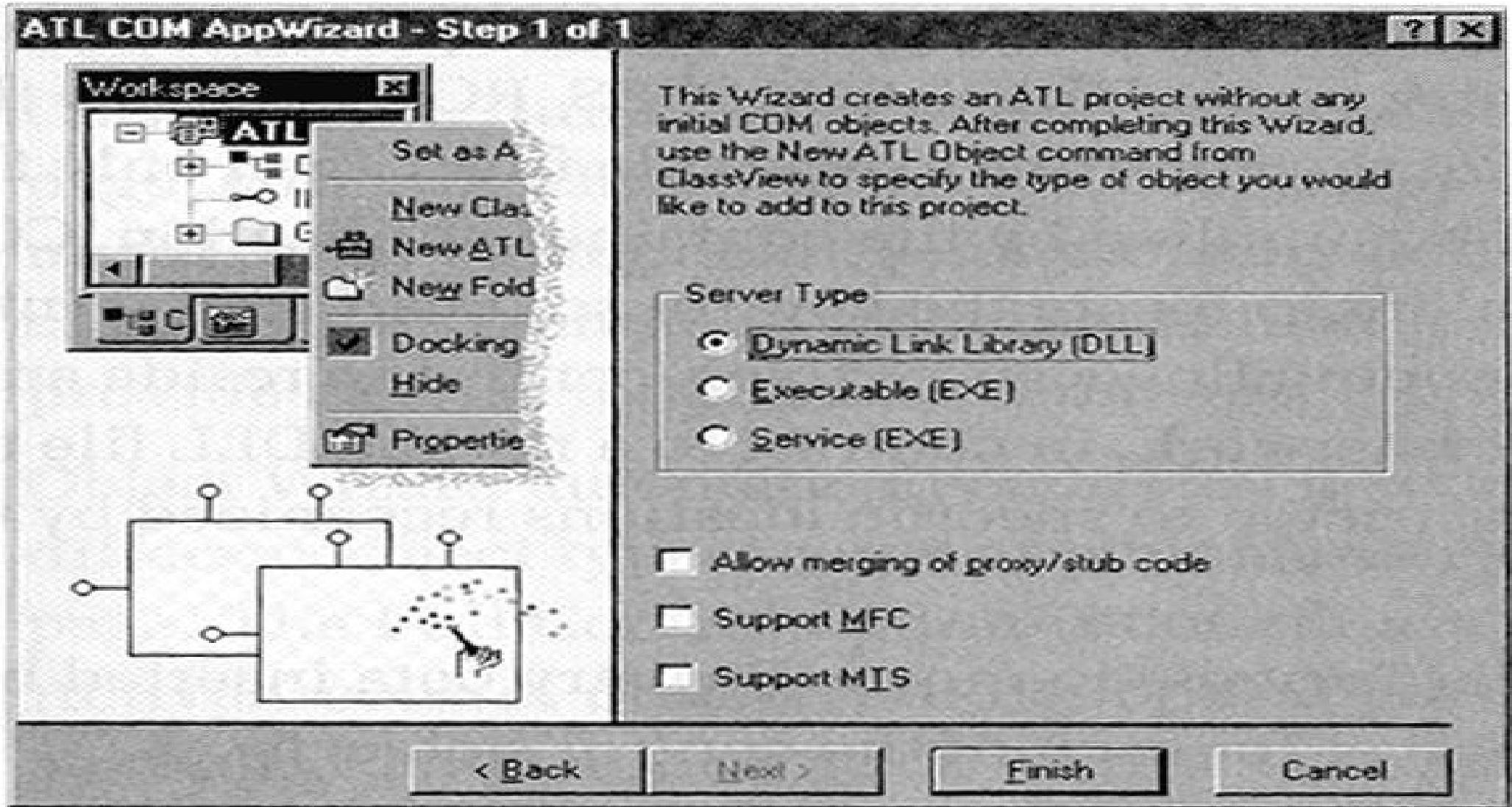


图 10-3 在 COM AppWizard 中选择一个项目

COM 和 ATL 都引用进程中的 ActiveX 控件作为动态链接库。在第 9 章中，我们已经看到，ActiveX 确实是一个动态链接库，但是，具有特殊的格式。在随后的步骤中，你可以在源代码中看到，要记住，“DLL”引用我们正在创建的 Pulse ActiveX 控件。

在设置 ActiveX 控件项目中，COM AppWizard 生成了几个源文件，其中包含必要的代码。Pulse.cpp 文件实现 DllMain 函数，在第一次加载库时，操作系统便调用这个函数，还有附加的四个服务于 COM 的函数。由于 COM 不提供这些函数，所以，Pulse 这样的进程内 ActiveX 控件就必须自己导出它们：

- DllGetClassObject：在客户第一次请求 COM 在内存中实例化控件目标时，从 CoGetClassObject 调用。此函数通过创建类工厂（class factory）的实例，并返回到 IClassFactory 或 IClassFactory 接口的指针，来处理这个调用。
- DllCanUnloadNow：在客户使用控件服务器完成时，从 CoFreeUnusedLibraries 中调用。此函数通知调用程序是否 ActiveX 控件管理的任何目标还处于服务状态，这是通过控件未决目标的内部引用计数来指出的。如果对于所有的接口这个计数值为 0，则 DllCanUnloadNow 函数便返回 S_OK，以允许 COM 从内存中卸载控件。

- **DllRegisterServer**：实现有关 ActiveX 控件的信息到系统注册表中。在控件的注册表数据中最重要的是 HKEY_CLASSES_ROOT\CLSID 下的项，它指定控件的可执行文件的位置。容器应用程序只给出控件的类标识符，COM 扫描 CLSID 文件夹，来定位控件，这样，它就可以将它加载到内存中。此安排允许 ActiveX 控件驻留在硬盘或网络上的任何地方，与普通的 DLL 文件通常必须要限制在由操作系统可以识别的特定位置不同。
- **DllUnregisterServer**：对每个控件目标，都删除由 DllRegisterServer 实现的注册表数据。

这四个函数是简短的包装器（wrapper），每个只有一行。它们都调用到一个名为 `_Module` 的目标中，这便是 ATL 的 `CComModule` 类的实例。它是 `_Module` 目标，为函数提供主要的实现。

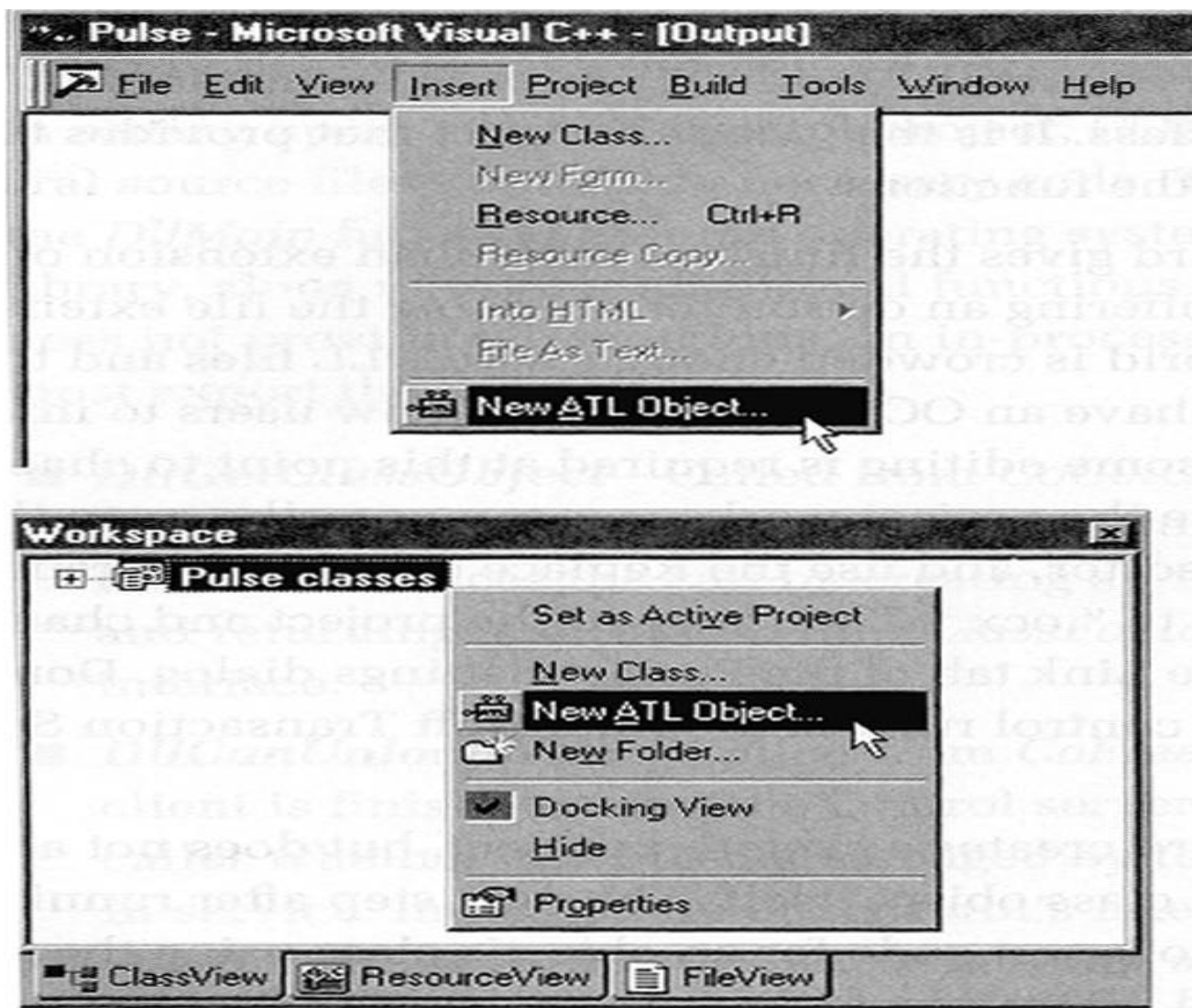
COM AppWizard 提供完成的控件，具有 DLL 扩展名，而不是 OCX 扩展名，它不提供选择文件扩展名的选项。你可能会感觉到 DLL 文件太多了，ActiveX 控件应当具有 OCX 扩展名，以允许用户推断出文件的目的。如果是这样的话，此时就需要进行一些编辑工作，来更改文件的扩展名。暂时关闭项目的工作控件，在文本编辑器中打开 `Pulse.dsp` 文件，并使用 `Replace` 命令，来重新命名所有的 “.dll” 为 “.ocx”。然后重新打开项目，并在 Project Settings（项目设置）对话框的 Link（链接）选项卡中更改

文件的扩展名。如果控件在 Microsoft Transaction Server 环境中运行的话，就不要这样做。

COM AppWizard 创建一个项目梗概，但是，不要为 ActiveX 类目标本身添加源代码。在运行 COM AppWizard 后，下一步是为目标的类实现代码，这要使用 ATL 的第二个向导，名为 ATL Object Wizard (ATL 目标向导)。

步骤 2：运行 ATL Object Wizard

ATL 的 Object Wizard 生成一个类的声明，以及为目标类实现函数的存根，使它与 MFC 的 ClassWizard 有些一致之处。运行 Object Wizard，如图 10-4 所示，方法是从 Insert (实现) 菜单中选择 New ATL Object (新建 ATL 目标) 命令，或者是在 ClassView 窗格中右击项目名，并从上下文菜单中选择这个命令。WizardBar 菜单也提供了到相同命令的访问。



10-4 调用 ATL Wizard 支持的常用方法

对于 17 种不同的目标类型，Object Wizard 自动添加 ATL 代码到项目中，其中的一些在表 10-4 中列出。在开发 ActiveX 控件项目时，通常选择 Full Control（完全控制）选项，但是，此选项假定控件将显示一个窗口，并以属性表来支持这种方法，而且还有一个表达图标。对于小的不可见的控件。例如 Pulse，通常，从那里选择一种更简单的组件类型，并建立它，而不是以后再删除不必要的代码，这样更合适。Pulse 项目不需要许多启动程序所使用的代码，所以，在向导对话框的左窗格中选择 Objects，并双击标签为 Simple Object（简单目标）的图标，如图 10-5 所示。

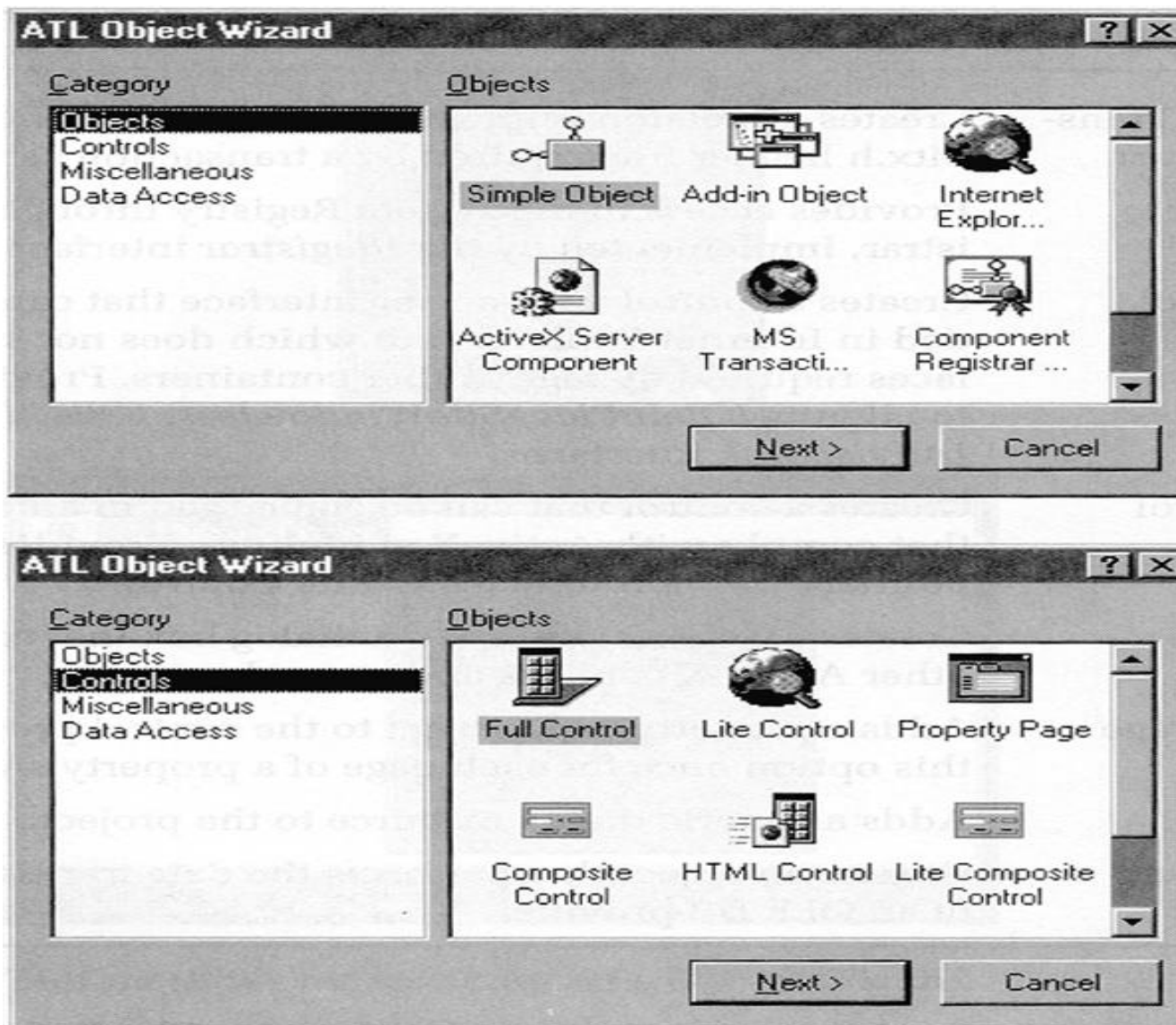


图 10-5 ATL Object Wizard 对话

表 10-4 ATL Object Wizard 支持的常用目标类型

目标类型	说明
Simple Object (简单目标)	创建一个主干 COM 目标。此接口支持必须手工添加
Add-in Object (附加目标)	创建一个简单的附加目标，它连接到 Developer Studio 的 IApplication 接口。第 13 章更详细地解释了 Developer Studio
Internet Explorer Object (Internet Explorer 目标)	创建一个 COM 目标，它支持 Internet Explorer 所期望的接口，但是，没有用户界面的附加支持
ActiveX Server Component (ActiveX 服务器组件)	为 OnStartPage 和 OnEndPage 添加支持，具有 ASP 接口的指针，例如 Irequest, Iresponse 和 Iserver

续表

MicrosoftTransactionServer (Microsoft 事务服务器)	创建一个概略的实现文件，它包括事务服务器所需要的 Mtx.h 头文件
Component Registrar (组件记录器)	通过 ATL 的 Registrar 来访问系统注册表，通过 Iregistrar 接口来实现
Lite Control (真正的控件)	使用可以嵌入到 Internet Explorer 中的用户界面来创建一个控件，但是，它并不支持由许多其他包容器所需要的接口。提供指针到客户的 IOleInPlaceWindowless, IOleClientSite 和 Iadvise
	Sink 接口
Full Control (完全控件)	创建一个控件，可以嵌入到所有符合 ActiveX 规则

续表

	的 7 中。提供相同的指针作为 Lite Control 到站点接口
Composite Control (混合控件)	创建一个类似对话框的控件, 它可以包含其他 ActiveX 控件和常规控件
Property Page (属性页面)	添加属性页面目标到控件项目中。只为每个属性页面选择一次此选项
Dialog (对话)	对目标添加一个常规对话资源
Provider (提供者)	创建一个目标, 来执行 OLE DB 提供者的数据转换服务

接下来, Object Wizard 显示 Properties 对话 (参见图 10-6), 它查询控件的特性。在 Names 选项卡中, 键入短名 PulseCtl, 从中, Object Wizard 使用适当的条目填充其他编辑框。Class 编辑框包含 CpulseCtl 类的名称, 它可以实现控件的目标。此类非常重要, 原因是它从 Pulse 控件支持的所有接口继承而来。

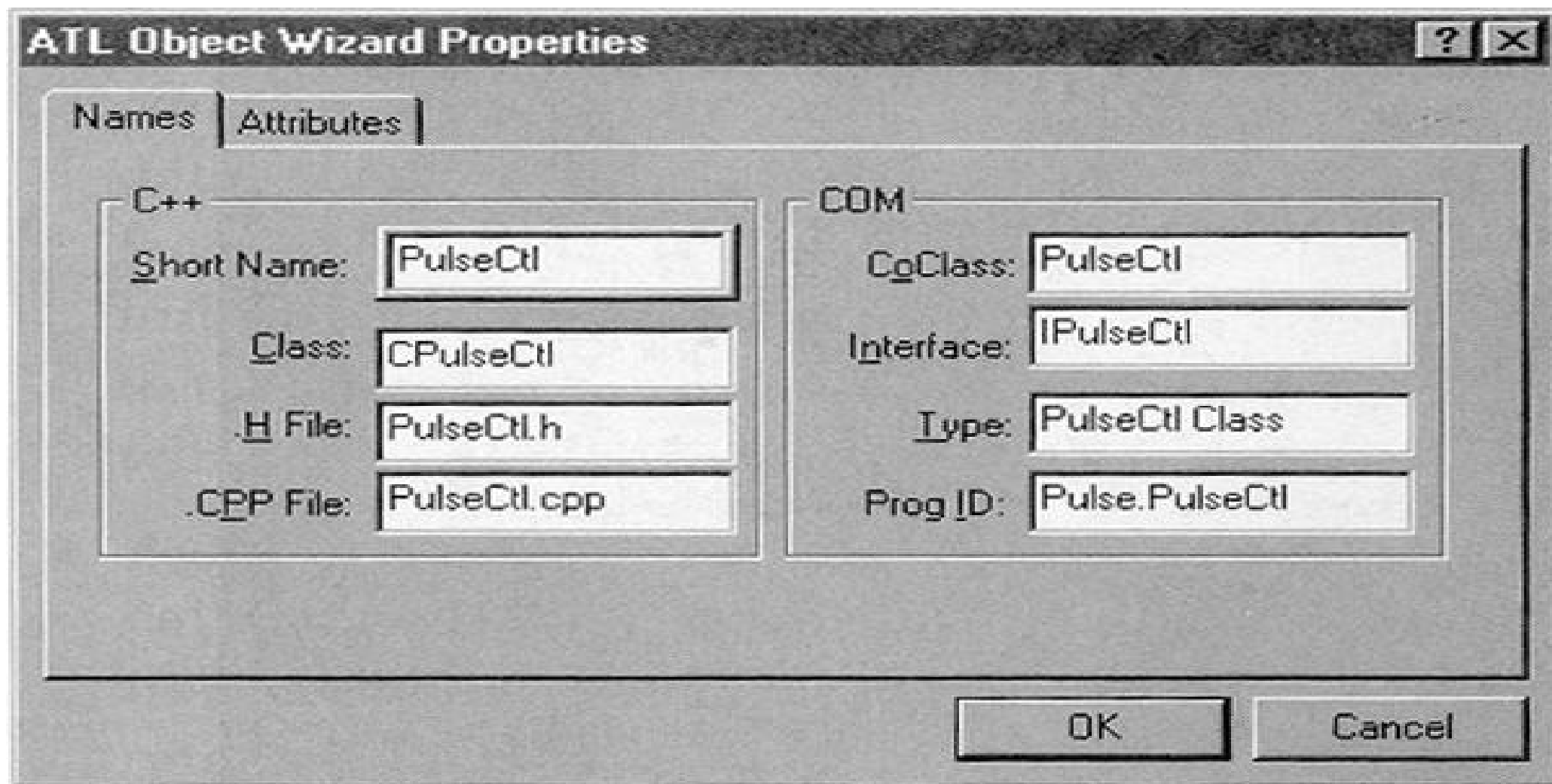


图 10-6 为 ATL Object Wizard Properties 对话中的对象指定名称

Object Wizard 编写 CPulseCtl 类源代码到指出的 H 和 CPP 文件。CoClass 编辑框

中容纳字控件的组件类的名称，它作为类型库，等效于对象类。CPulseCtl 类和 PulseCtl 组件类引用相同的目标，但是，定义对象的位置有所不同。CPulseCtl 引用到实现此对象的 C++ 源代码，而 PulseCtl 组件类引用到此对象在控件的类型库中的定义。Interface 框显示接口的名称，在这个接口中，控件将其定制的方法和属性展示出来。标有 Type and ProgID 的编辑框容纳字符串，它描述了 CPulseCtl 对象，及其编程的标识符。

如图 10-6 所示，一些名称作为系统注册表中的条目，由 Pulse 的 DllRegisterServer 函数放置在那里。例如，编程的标识符 PulseCtl.PulseCtl 和 PulseCtl.PulseCtl.1 分别成为 VersionIndependentProgID 和 ProgID 注册表键下面的字符串。在这两个标识符之间，唯一的不同是，在此情况下，后者包含控件的版本号 1。人们可以识别这两个字符串，它们是控件的标识符的另一种形式，它提供了一种手段，使容器应用程序可以请求控件对象，而不必使用 CLSID 字符串。例如，MFC 的 CWnd::CreateControl 函数按如下的方法接收编程的标识符：

```
CreateControl( "PulseCtl.PulseCtl.1", NULL, 0 &rect,  
  
              pParentWnd, IDC_PULSECTL);
```

类似地，Visual Basic 或 VBA 容器可以传递标识符到 CreateObject 函数：

```
Dim PulseCtl As Object
```

```
PulseCtl = CreateObject (“PulseCtl.PulseCtl.1”)
```

客户仅为控件控制变成的标识符，它可以确定对应的类标识符，方法是调用 OLE 函数 CLSIDFromProgID。函数 ProgIDFromCLSID 执行相反的转换。

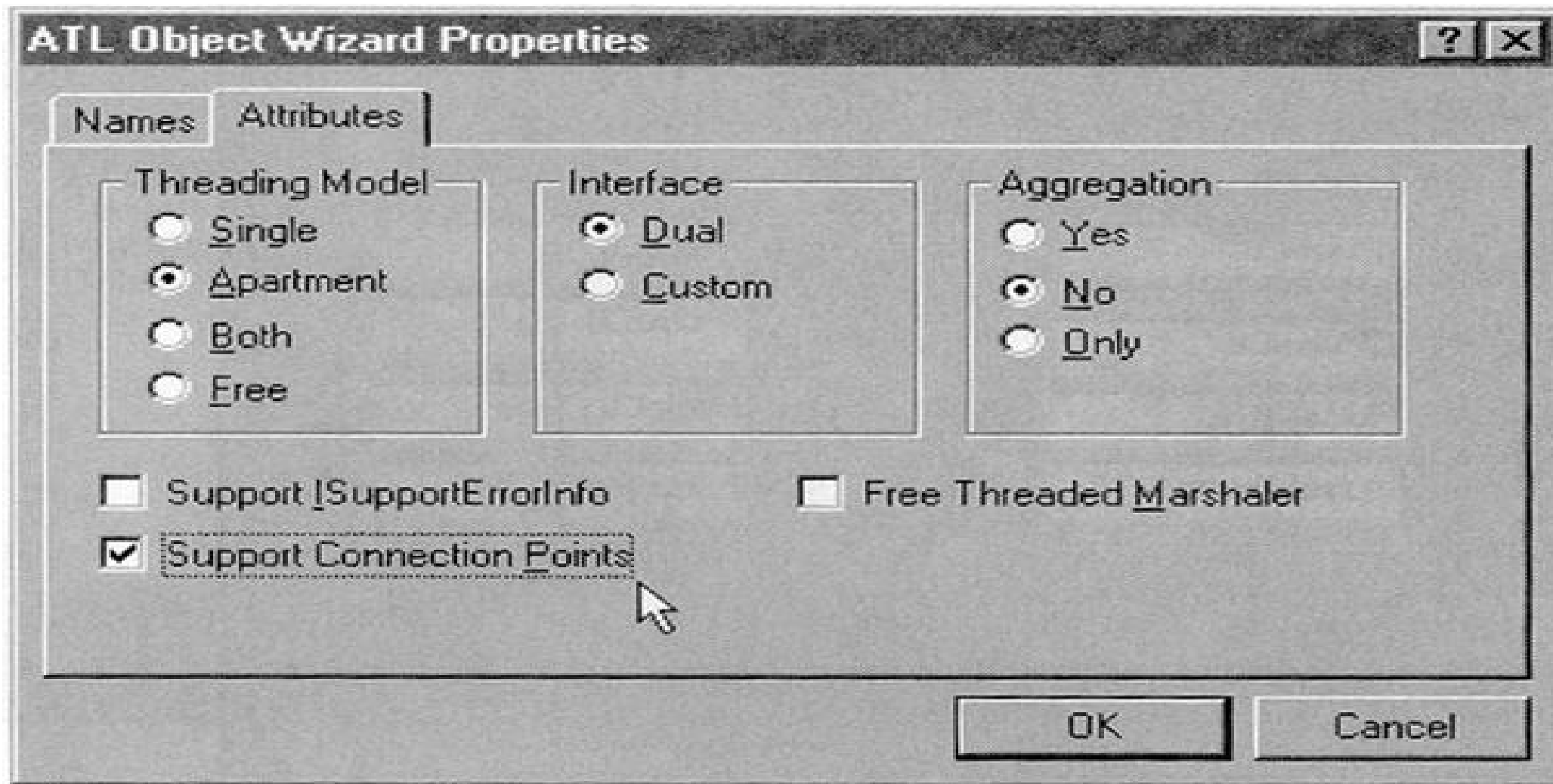


图 10-7 ATL Object Wizard Properties 对话框

在 Attributes（属性）选项卡中，选择标有 Support Connection Points（支持连接

点)的复选框,如图 10-7 所示。对于诸如启动事件的 Pulse 这样的 ActiveX 来说,连接点是必要的。此选项卡还显示单选按钮,来确定控件的线程模式、接口类型,以及是否它支持聚合特征。我们已经查看了 ATL 支持的线程模型,但是,双接口和聚合特征也应当详细说明。

- 双接口:在第 8 章中,我们已经看到,Microsoft 已经扩展了 OLE 的设计,以支持定制的控件,使这样的组件作为 VBX 组件的 32 位方式。然而,由于这种方式传递函数参数,所以,版本 4 之前的 Visual Basic 不能直接调用控件方法。为了解决这个问题,Microsoft 在 OLE 中合并了 Idispatch 接口,以提供必要的链接。现在,Idispatch 通常作为客户脚本编程的通信管道,例如 VBScript 和 JavaScript,使它们可以通过 Idispatch::Invoke 函数的服务,来调用控件的方法,它可以转换参数,并返回值到客户的本机数据类型,或者,返回客户本机数据类型的值。尽管这种解决方案可行,但是,额外的转换和间接调用 Invoke,将会减慢包容器与控件之间的交互。

对于 Web 页面,以及以前的 Visual Basic 客户,Idispatch 是一个必要的折衷方法。然而,在 C++这样的语言中编写的包容器应用程序可以直接访问控件的方法,其方法是通过指针进行调用。Idispatch::Invoke 及其耗时的转换都不再需要了。为了适应支持除了 Variant 数据类型的客户,OLE 支持双接口技术。双接口方法函数要么通过 Idispatch::Invoke 间接进行调用,要么通过调用到方法的指针来直接访

问，这样，不管包容器应用程序是使用什么语言编写的，所有包容器应用程序都可以得到适当的服务。

- 聚合：通过聚合，另一个对象可以在其调用程序中出现，以具有 Pulse 控件加上第二个对象提供的新服务的能力。聚合对象通常叫做包含或外部对象，它嵌入 Pulse 控件的方式与包容器应用程序的相同。然后，外部控件选择性地从其客户中 IQueryInterface 请求，为 IPulseCtl 接口将请求传递给 Pulse。外部对象的客户接收需要的指针，并调用到 Pulse 中，它不知道现在是 Pulse，而不是聚合它的对象正在提供服务。

在 Obejct Wizard Properties 对话框中，聚合选择不确定控件是否聚合另一个对象，是否控件本身是聚合的。

如果你的控件不打算服务于以前的 Visual Basic 应用程序，或者，在 Web 页或 Active Server Page 上进行脚本编程，在此对话框中选择 Custom（定制）单选按钮。这样做的好处是，它可以稍微减少最终控件的大小，因为每个实现的接口都没有合并 Idispatch 的四个方法。尽管具有客户接口的控件可以在 HTML 文档中出现，但是，它并不能通过脚本进行编程。脚本解释器必须在控件对象上找到 Idispatch，或者，它不能访问此控件。

对于 Pulse 控件，接收默认的设置，来启用公寓模型线程，并添加双接口支持，但是，没有聚合特征。在深思熟虑之后，可以通过在 Aggregation（聚合）组中选择 No 单选按钮，来使 Pulse 成为非聚合的。这便对 CPulseCtl 类定义添加了一

行，如下所示：

```
DECLARE_NOT_AGGREGATABLE(CPulseCtl)
```

尽管聚合通常被认为是 ActiveX 控件的一个有用的特性，但是支持聚合使控件的执行空间增加了大约 2KB。由于这本书的一个目的是说明如何用 ATL 创建控件，从而减小控件的大小，因而放弃聚合似乎是一种合理的折衷方案。

这个折衷方案并不像表面上看起来的那么严重，因为聚合并不是 ActiveX 控件在使用另外一种技术时所必须使用的唯一技术。任何原因都不能阻止一个控件像一个客户一样动作，而且可以嵌入在 Pulse 控件中。这项技术被称为包容，包容控件提供了运行 Pulse 自己的方法函数，并通过它自己的类库公布这种方法。当一个客户应用程序要求某项 Pulse 服务时，包容控件（我们称之为 Outer（外部对象））将这个调用传递给 Pulse。当 Pulse 启动这个事件后，控件回到 Outer（外部对象）处理程序函数，它将通过启动它的客户应用程序来传递事件。当使用聚合时，客户应用程序并不知道另一个控件正在提供这项服务。但与聚合不同的是，包容器减少了 Pulse 和客户应用程序之间的通信，因为 Outer（外部对象）控件必须作为二者之间的中介而存在。

不要选中标有“Free Threaded Marshaler（自由线程调度器）”的复选框。这个选项将把自由线程调度器这个对象加入到项目中，自由线程调度器说明如下。

自由线程调度器

在 ATL Object Wizard（对象向导）中选中“Free Threaded Marshaler（自由线程调度器）”复选框，将产生对 COM 的 CoCreateFreeThreadedMarshaler 函数的调用。

```
Hresult FinalConstruct()  
  
{  
  
    return CoCreateFreeThreadedMarshaler(  
  
        GetControllingUnknown(), &m_pUnkMarshaler.p);  
  
}
```

这个函数将创建一个称为自由线程调度器的对象，它将与控件聚合并监督调度操作。它的主要目的是，当一个包容器应用程序使用嵌入到使用两个线程模型的某个 ActiveX 控件中的单线程公寓模型时，来提高性能。为了理解自由线程调度器如何改善这样一个控件的性能，应仔细研究一个典型的运行两个 STA 线程的客户，在这样的客户中，线程 B 必须调用线程 A 所拥有的对象实例。线程 A 首先调用 CoMarshalInterThreadInterfaceInStream，返回时收到一个指向流的指针（流只是一个数据的集合）。线程 A 将流指针传递给线程 B，线程 B 利用流指针调用 CoGetInterfaceAndReleaseStream，收到一个指向代表理想实例的代理的指针，线

程 B 现在可以安全地调用代理，来对对象接口的方法进行访问，即使实例在不同的公寓中创立。

如果一个同时使用两个线程模型的 ActiveX 控件可以安全地处理从任何公寓来的直接调用，甚至来自于不同的 STA 公寓，通过一个调度器整理来自客户的线程 B 的调用所浪费的时间是不必要的，因为客户可以直接正确（而且更加有效）地从线程 B 中调用对象实例。客户不能安全地假定这个选项存在，所以必须通过 CoGetInterfaceAndReleaseStream 应用一个代理。但是，因为这个控件是用于对不同 STA 线程上同时进行安全的访问，它通过自由线程调度器调用 CoCreateFreeThreadedMarshaler 以进行它自身的用户调度。这个对象就像一个存根一样，将一个指向接口的指针拷贝到流中，而这个接口正是客户通过 CoGetInterfaceAndReleaseStream 所申请的。结果就是线程 B 得到了它的接口指针，而不用切换线程，或者遍历 COM 调度器的路径。客户无法了解这个差别，除非因为调用实例是直接访问方法，而不是通过代理码进行，使得调用实例更快一些。可是一个采用这项技术的控件必须确保对象可以处理同时调用。

单击 OK 按钮可以关闭 ATL Object Wizard（ATL 对象向导）对话框，此时向导将产生三个文件：

- PulseCtl.h 和 PulseCtl.cpp：新 CpulseCtl 类的定义和执行码。
- PulseCtl.rgs：包括控件的注册信息脚本的文本文件。RGS 文件中的注册脚本是包含在控件执行文件资源数据中的一部分，DllRegisterServer

函数将其读出，并安装在 Registry（注册表）中。Object Wizard（对象向导）在项目的 RC 文件中加入一行，来引用注册脚本：

```
IDR_PULSECTL      REGISTRY DISCARDABLE      "PulseCtl.rgs"
```

并在 Resource.h 中为 IDR_PULSECTL 常量插入一个 #define 语句。

Object Wizard（对象向导）也在 StdAfx.cpp 和 StdAfx.h 文件中插入了几个 #include 语句。这些语句将带给项目所必须的 ATL 源文件，例如 AtlImpl.cpp，AtlCtl.cpp 和 AtlWin.cpp。尽管 StdAfx 文件的名称看起来与 MFC 项目类似，它们只在选择了步骤 1 中的 MFC 选项后才引用 MFC 头文件。

对于加入到控件中的每一个对象，Object Wizard（对象向导）在控件对象映射中加入一个入口。Pulse 仅包含一个 CPulseCtl 对象，所以在向导结束后，Pulse.cpp 文件中的对象映射看起来是这样的：

```
BEGIN_OBJECT_MAP(ObjectMap)

    OBJECT_ENTRY(CLSID_PulseCtl,CpulseCtl)

END_OBJECT_MAP()
```

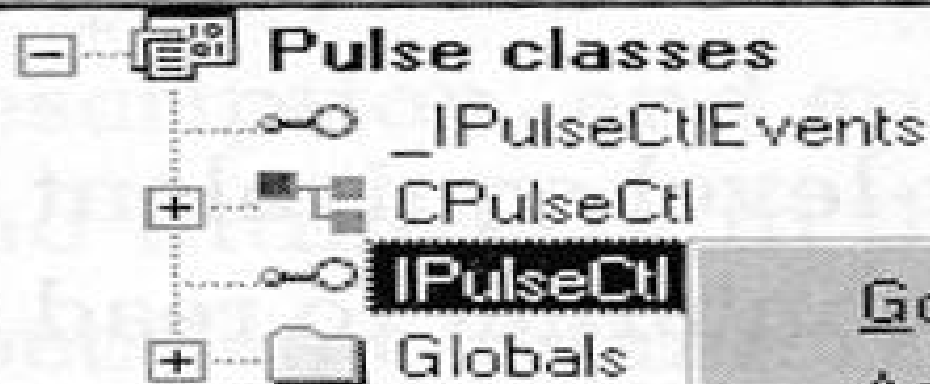
最后，ATL Object Wizard（对象向导）对项目的 IDL 文件进行必要的修改，控件

的类库就是从 IDL 文件中产生的。IDL 代表接口描述语言，而 IDL 文件作为 Microsoft IDL 编译工具 MIDL 的输入。当增加 Pulse 的事件函数时，就能看见这个项目的 IDL 文件的详细资料。

步骤 3：加入 nInterval 属性

Pulse 有唯一的一个用户属性，称为 nInterval，它包括以毫秒为单位的控件启动事件的速度。为了在接口上加入这个属性，在 ClassView 窗口中展开 Pulse 类列表，用鼠标右键单击 IPulseCtl 的入口，然后选择 Add Property（加入属性）命令：

Workspace



Go to Definition

Add Method...

Add Property...



New Folder...



Docking View

Hide



Properties

 ClassView

 ResourceView

 FileView

这将激活 Add Property To Interface（为接口加入属性）对话框，如图 10-8 所示，它与第 9 章中所遇到的类向导的 Add Property（加入属性）对话框所要求的信息是一样的。

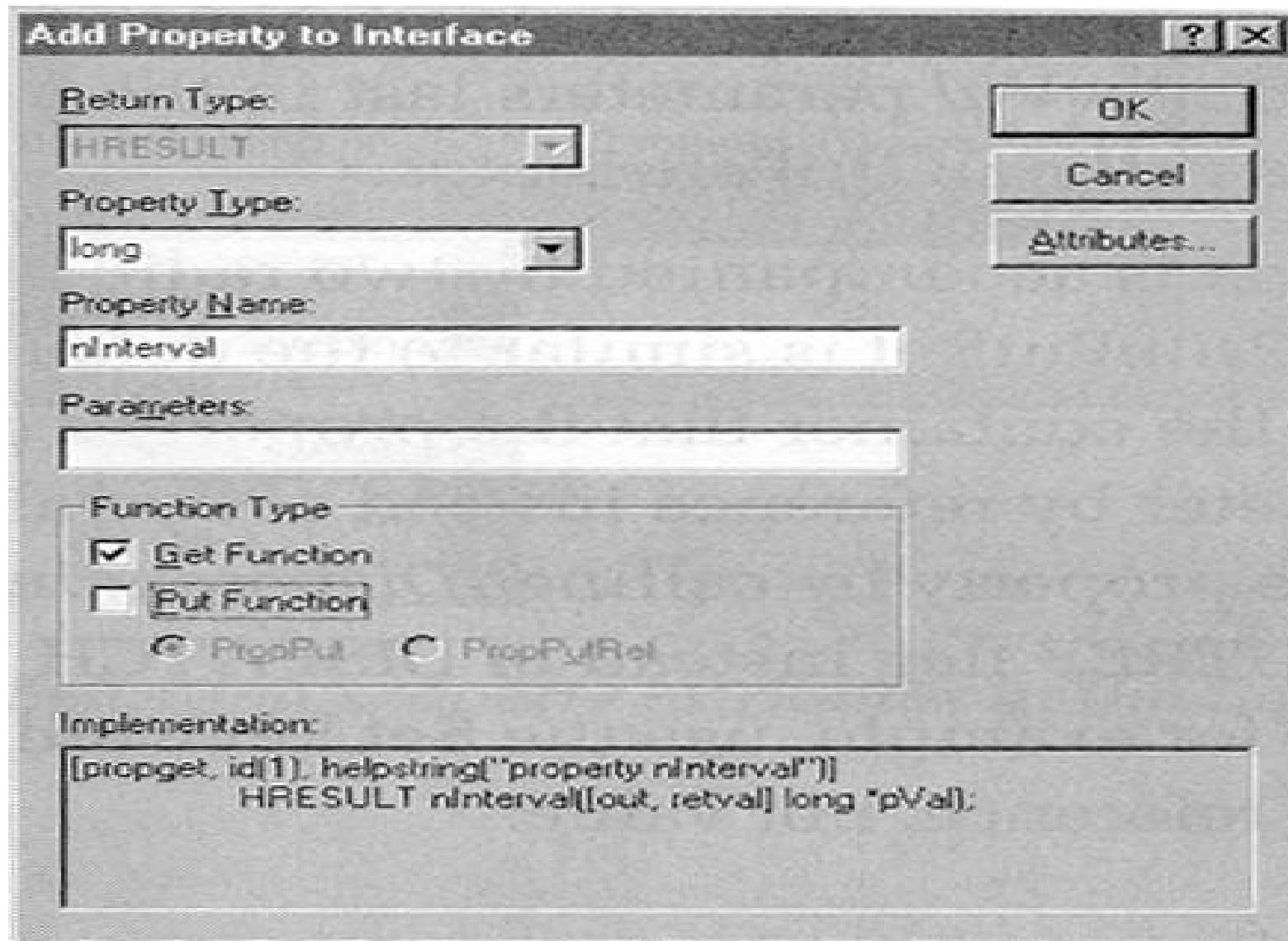


图 10-8 对 Pulse ActiveX 控件增加 nInterval 用户属性

键入 `nInterval` 作为 `Pulse` 属性的名字，给定它的属性类型为 `Long`（这个属性为 `short` 时将简单一些，但是，那将限制最大时间间隔为一分钟多一点）。属性的获取/放置方法的默认 IDL 类型出现在对话框的底部：

```
[propget, ... ]
```

```
HRESULT nInterval ( [out, retval ] long *pVal);
```

```
[propput,... ]
```

```
HRESULT nInterval ([in] long newVal);
```

COM 的获取/放置方法与 MFC 的获取/设置函数是低级等价的，它为包容器提供了读写控件属性数据的方法。当 MIDL 编译器编译项目的 IDL 文件时，它将通过在属性名字前加入 `get_` 和 `put_` 来定义接口的两个方法。获取函数的 `pVal` 参数指向属性的当前值；而放置函数的 `newVal` 参数包含了一个取代旧属性值的新属性值。如果控件的获取/放置方法需要一个更大的扩展参数列表，在对话框参数框中增加变量数。对话框将任何增加的变量插入到 `pVal` 和 `newVal` 参数之前，并使它们可以被获取和放置方法公用。如果不想让两个函数具有相同的列表，必须在稍后用文本编辑器手工编辑 IDL 文件。可以自由改变 IDL 文件中参数的顺序或者名字，但是，在参数列表的最后，必须留下 `pVal` 和 `newVal` 参数，以及它们的 `[out,retval]` 和 `[in]` 属性。

Put Function（放置函数）复选框与两个标有 PropPut 和 ProPutRef 的单选钮是同时出现的。PropPutRef 与默认的 PropPut 选项是类似的，除了它告诉包容器，属性的放置函数靠引用而不是靠值来得到 newVal 参数。在这种情况下，包容器将通过调用 IDispatch::Invoke 设置属性，并用 DISPATCH_PROPERTYPUTREF 标志取代 DISPATCH_PROPERTYPUT。例如对于 VB 的用户，用 set 关键字来表示一个属性通过引用而不是通过值进行赋值：

```
Set PulseCtl.nInterval=x
```

清除图 10-8 中的 Put Function（放置函数）复选框，目的是防止 MIDL 编译器为 nInterval 属性产生一个放置方法入口。不能允许一个包容器改变 nInterval 属性，清除对话框内的 Put Function（放置函数）复选框将使变量是只读的。控件的另一种方法为客户提供了一个更加符合逻辑的设置 nInterval 属性的方法。

对话框的 Attributes（属性）按钮打开了一个对话框，其中可以选择属性特征，例如调度标识符，一个可选的描述和特殊标志。所选择的属性将出现在 IDL 文件中，并最终出现在类型库中，在那里，它们向将来使用的包容器描述属性。单击图 10-9 中的组合框，将显示出一张可用的属性设置列表，如表 10-5 所示。要牢记，属性的确定要引用属性的获取/放置函数，而不要引用属性变量自身。在 COM 中，字的属性通常被用作获取/放置方法的速记形式，它可以让一个客户访问属性。

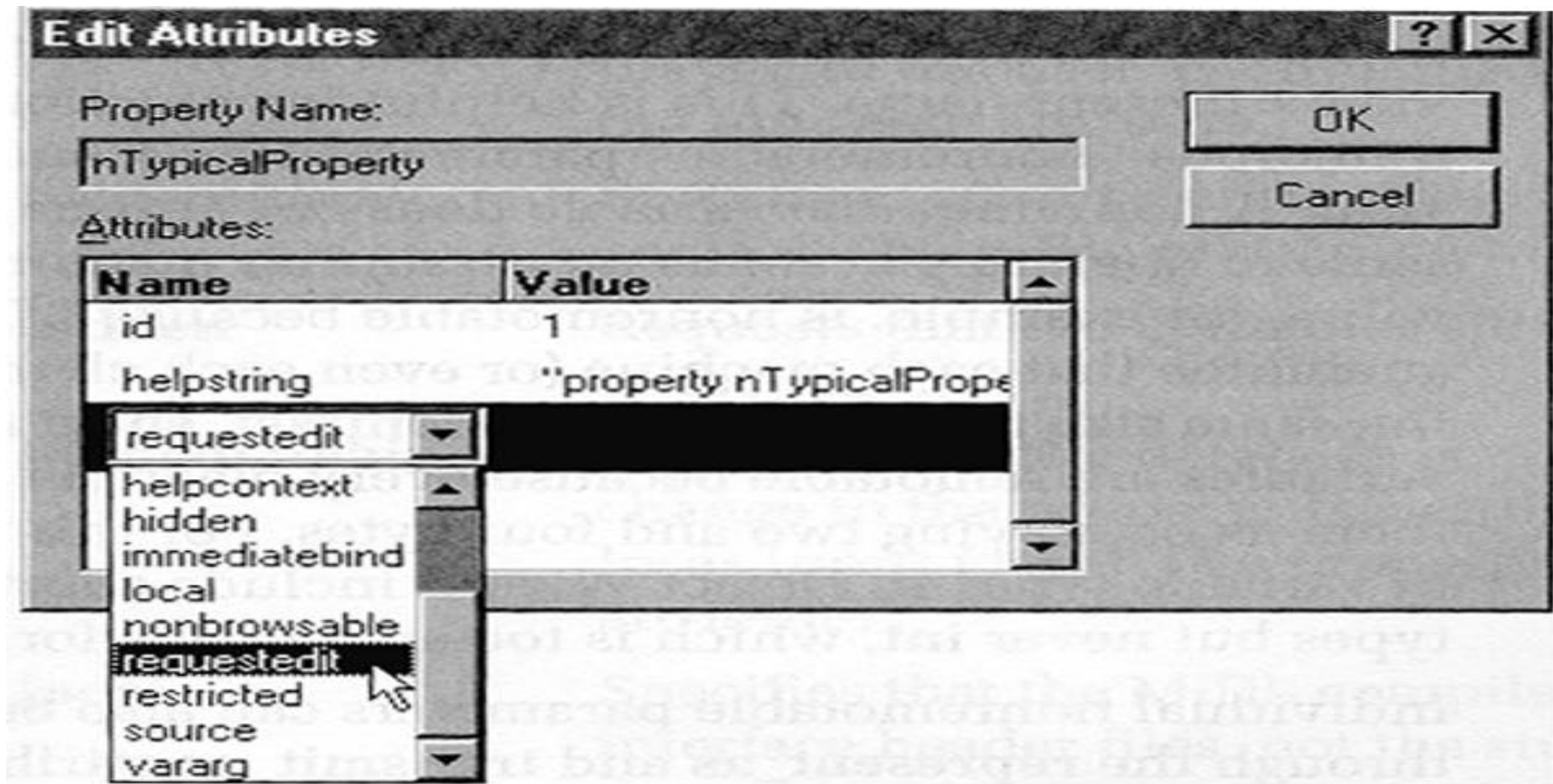


图 10-9 在 Edit Attributes (编辑属性) 对话框中选择属性

表 10-5 属性

属性	描述
id	为属性的获取或者放置方法指定一个调度标识符（DISPID）
helpstring	指定一个短文本串，用以描述属性。包容器可以通过控件的 <code>TypeInfo::GetDocumentation</code> 方法提取这个串。尽管这个帮助文本作为控件类信息的一部分储存在控件内部，在 Edit Attribute（编辑属性）对话框内删除 helpstring 入口，并不能减小控件的执行映像的大小

续表

bindable

用组合数据使属性与数据库中的特定域联系在一起。它的意思是，任何时候当属性值改变时，控件通知数据库，并要求所联系的记录域进行放置，以反映新的值。为得到关于这个非常有用的概念的进一步信息，请参考标题为“ActiveX Controls: Using Data Binding in an ActiveX Control (ActiveX 控件：在一个 ActiveX 控件中使用组合数据)”的 MSDN 文章。可以用复选框设置 Search Titles Only (仅按标题搜索) 在 MSDN 的 Search (搜索) 选项卡中搜索这篇文章

续表

call_as

使客户可以用不同的名字访问属性的获取/放置函数。这对于有大量的“nonremotable（非远程）”参数类型的函数是非常有用的，例如 int 或者 void。一个非远程的变量对于不同机器上的操作系统来说并不是完全相同的。例如，因为 COM 并不保证每一台机器（或者每一个客户）为变量分配相同的空间，所以一个 int 值并不是远程的。相反，由于每一台机器认为 short 和 long 变量占据 2 和 4 个字节，所以它们是远程的。由于这个原因，在 Object Wizard（对象向导）中的变量类型列表包括 short 和 long 类型，而没有 int，这对于 COM 是非常不明确的。

单个非远程参数也可以通过 represent_as 和 transmit_as 属性进行规定。当时对于使用几个非远程类型变量的获取/放置函数来说，call_as 比 represent_as 和 transmit_as 更为方便和有效。对属性函数规定一个 call_as 属性，意味着控件可以一步进行所有必要的转换，而不是用几步完成，每一步进行一个变量的转换。

用非远程函数参数，对于像 ActiveX 控件这样的进程内

续表

服务器的效率是非常低的，应为它强迫通过代理/存根代码对调用 `call_as` 函数进行调度。必须为处理非远程参数的包容器的代理 DLL 提

供一个对话例程，而且为控件中的存根提供另一个例程，以接受调用。在一个 `ActiveX` 控件中的所有函数参数全部使用远程类型，将确保从包容器可以不经过调度而直接访问控件

`defaultbind`

指明可绑定的最代表控件对象的属性。只有控件的一个属性可以具有 `defaultbind` 标志，而且它必须具有 `bindable` 标志。`Defaultbind` 标志允许一个包容器绑定在整个控件对象上，而不是单个属性

`defaulcollelem`

允许用 `Visual Basic for Applications` 编写的客户直接访问属性的获取/放置函数

`displaybind`

指示包容器绑定的属性必须显示给用户。这个属性同样必须具有 `bindable` 标志

续表

helpcontext	规定一个 32 位数字,以识别控件的属于某个属性的帮助文件中的信息
hidden	要求容器不向用户显示属性
immediatebind	要求在放置属性后,立即通知数据库,而不是等到控件失去输入焦点之后。这个属性同样必须具有 bindable 标志
local	规定 MIDL 编译器必须仅仅产生接口头文件,而不产生存根代码。local 标志与类似 Pulse 这样的过程内 ActiveX 控件没有关系
nonbrowsable	要求容器不要包含容器属性浏览器中的属性

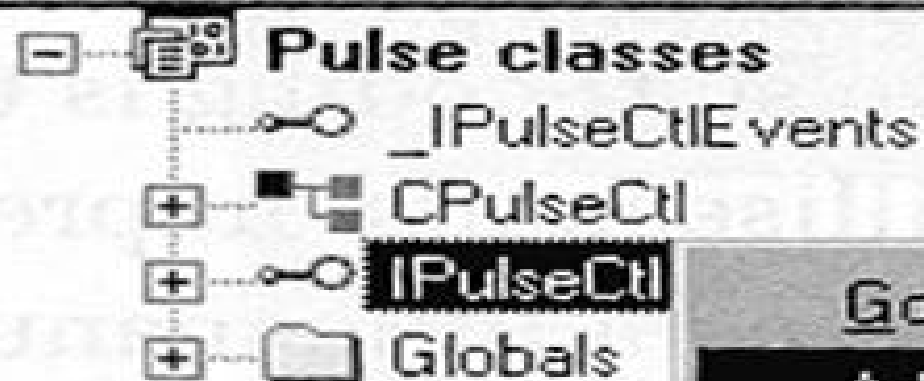
续表

requestededit	表示在改变属性值之前，控件将向包容器查询是否允许改变。通过 <code>IPropertyNotifySink::OnRequestEdit</code> 函数可以申请许可，这个函数将通知包容器属性将被改变，而且对象正在申请允许，以继续进行。从 <code>OnRequestEdit</code> 的返回值 <code>S_FALSE</code> 将拒绝这个申请；而返回值 <code>S_OK</code> 将允许改变属性值。在收到 <code>S_OK</code> 后，如果属性也具有 <code>bindable</code> 标志，则控件必须调用 <code>IPropertyNotifySink::OnChanged</code>
restricted	规定属性的获取/放置方法不能从宏中调用
source	说明获取/放置函数返回一个对象或者 <code>Variant</code> 作为某事件源的变化。 <code>source</code> 很少用在属性中，但是我们将在 <code>Pulse</code> 的接口列表中使用它
vararg	说明这个属性的获取/放置方法可以接收数目不同的参数。这个方法的一个参数必须是一个可靠的 <code>Variant</code> 类型数组，其中包含了每一个未规定的参数的默认值

步骤 4：加入方法

关闭 Add Property To Interface（为接口加入属性）对话框，将在 IDL 文件中写入属性信息。下一步是加入三个名为 StartPulse, EndPulse 和 _OnTimer 的方法。通过调用 StartPulse 方法，包容器告诉 Pulse 开始节拍性的事件启动过程。这个函数参数以毫秒为单位，规定了包容器接收事件通知的事件间隔。在控件停止激活后，包容器调用 EndPulse 停止事件启动过程，这样将不占用 CPU 时间。_OnTimer 函数由 CTimer 类使用，在步骤 6 中进行说明。对于每一种方法，像以前一样用鼠标右键单击 ClassView 窗格中的 IPulseCtl 接口，但这一次选择 Add Method（加入方法）命令：

Workspace



Go to Definition

Add Method...

Add Property...



New Folder...



Docking View

Hide



Properties



ClassView



ResourceView



FileView

StartPulse 的 nRate 参数是 nInterval 属性的新值。这解释了在前面的步骤中，我们决定一个客户不需要访问一个 put_nInterval 函数，因为调用 StartPulse 可以达到同样的目的。在图 10-10 中所示的 Add Method To Interface（为接口加入方法）对话框中键入函数和参数名称，然后单击 OK 关闭对话框。重复以上步骤，以加入 EndPulse 和 _OnTimer 方法。EndPulse 和 _OnTimer 都没有参数，所以使参数框空着。关闭 Add Method To Interface（为接口加入方法）对话框，此时 Visual C++ 将把对应 3 种方法的适当代码加入到 Pulse.idl 文件中。它还将为方法写出存根函数，加入到 Pulse.cpp 文件中，我们将在为控件加入一个事件函数后，对其进行编辑。

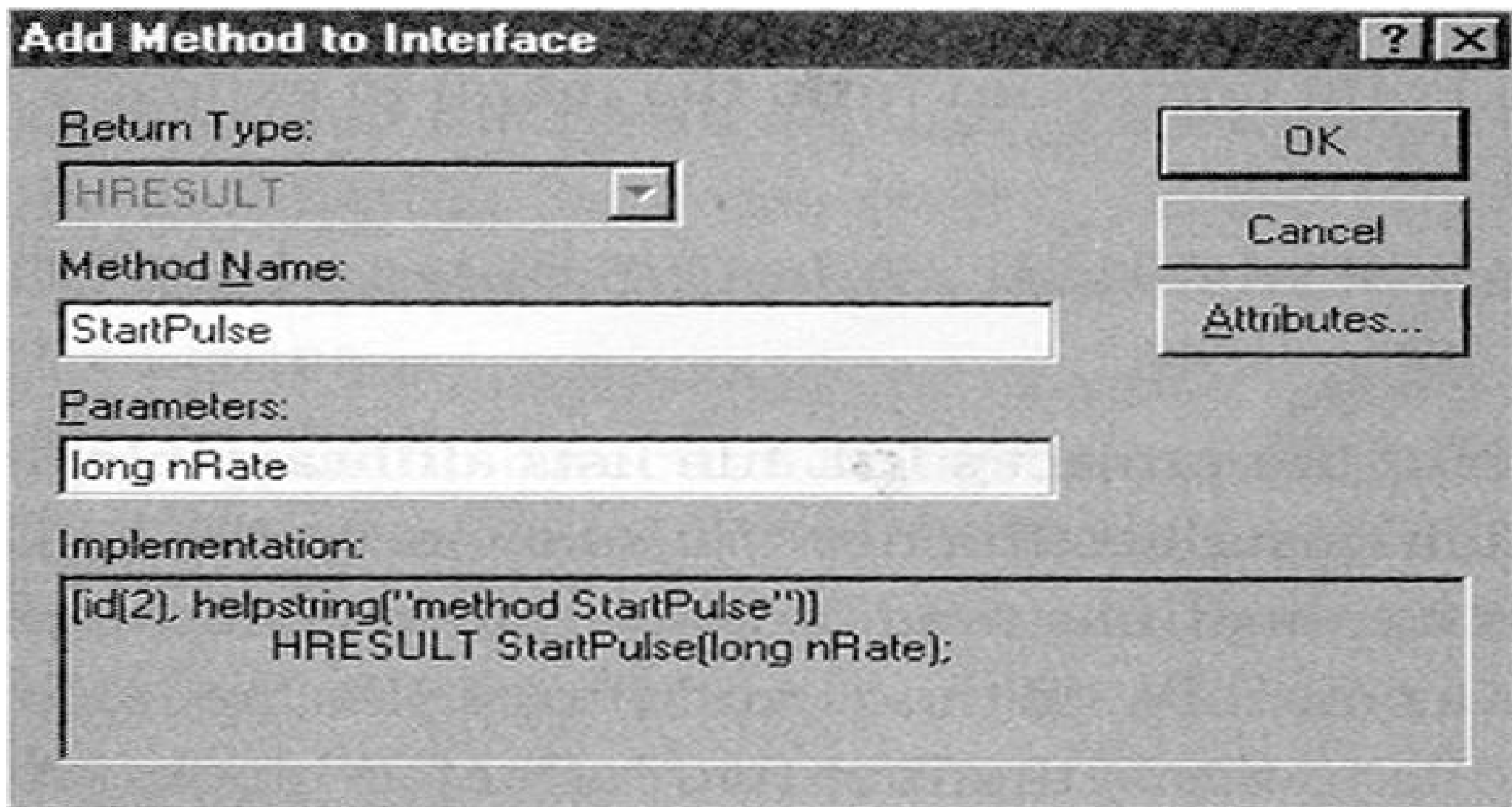


图 10-10 Add Method to Interface 对话框

步骤 5：加入 Pulse 事件

在我们开始编码之前，为项目添加的最后一个部分是 Pulse 事件，它将在每一个时间间隔到来时启动。在 ATL 以前的版本中，在控件中增加一个事件需要一点手动操作，包括产生一个事件接口的 GUID 标识符，编辑 IDL 文件和运行一个称为 ATL Proxy Generator 的工具。但随着库的第三版的发布，这个过程变得更加简单和友好。现在为控件增加事件和增加方法和属性一样容易，仅仅需要一点时间对项目的 IDL 文件进行编译。

在我们将讨论发生在屏幕之后的事情之后，下面给出了产生新事件代码过程中所需要的步骤列表。图 10-11 说明了 4 个步骤。

1. 用鼠标右键单击 ClassView 窗格中的 `_IPulseCtlEvents` 入口，并从菜单中选择 `Add Method`（加入方法）命令。在 `Add Method To Interface`（为接口加入方法）对话框中，选择 `void` 返回类型，将事件函数命名为 `Pulse`，然后单击 `OK` 关闭对话框。
2. 找到 Workspace 窗口中的 FileView 窗格，用鼠标右键单击 `Pulse.idl` 入口，然后选择 `Compile`（编译）`Pulse.idl` 命令。这将启动 MIDL 编译器，它将产生类型库文件 `Pulse.tlb`，并加入到项目中。
3. 当 MIDL 编译器结束工作后，切换回 ClassView 窗格，然后用鼠标右

键单击 CPulseCtl 入口。从菜单中选择 Implement Connection Point (实现连接点) 命令, 以显示同名字的对话框。

4. 设置标有 _IpulseCtlEvents 的复选框, 并单击 OK 关闭对话框。

项目的 IDL 文件底部一半的地方列出了名为 PULSELib 的 library 块, 它将向 MIDL 编译器描述新 Pulse 事件。当为一个控件的方法和属性搜索类型库时, 某些包容器应用程序仅仅读出 library 块, 所以通常要小心地编辑 IDL 文件, 以移动最前的两个缩进代码块到 library 块。第一个块由方括号 [] 组成, 包括 IPulseCtl 接口的标志;

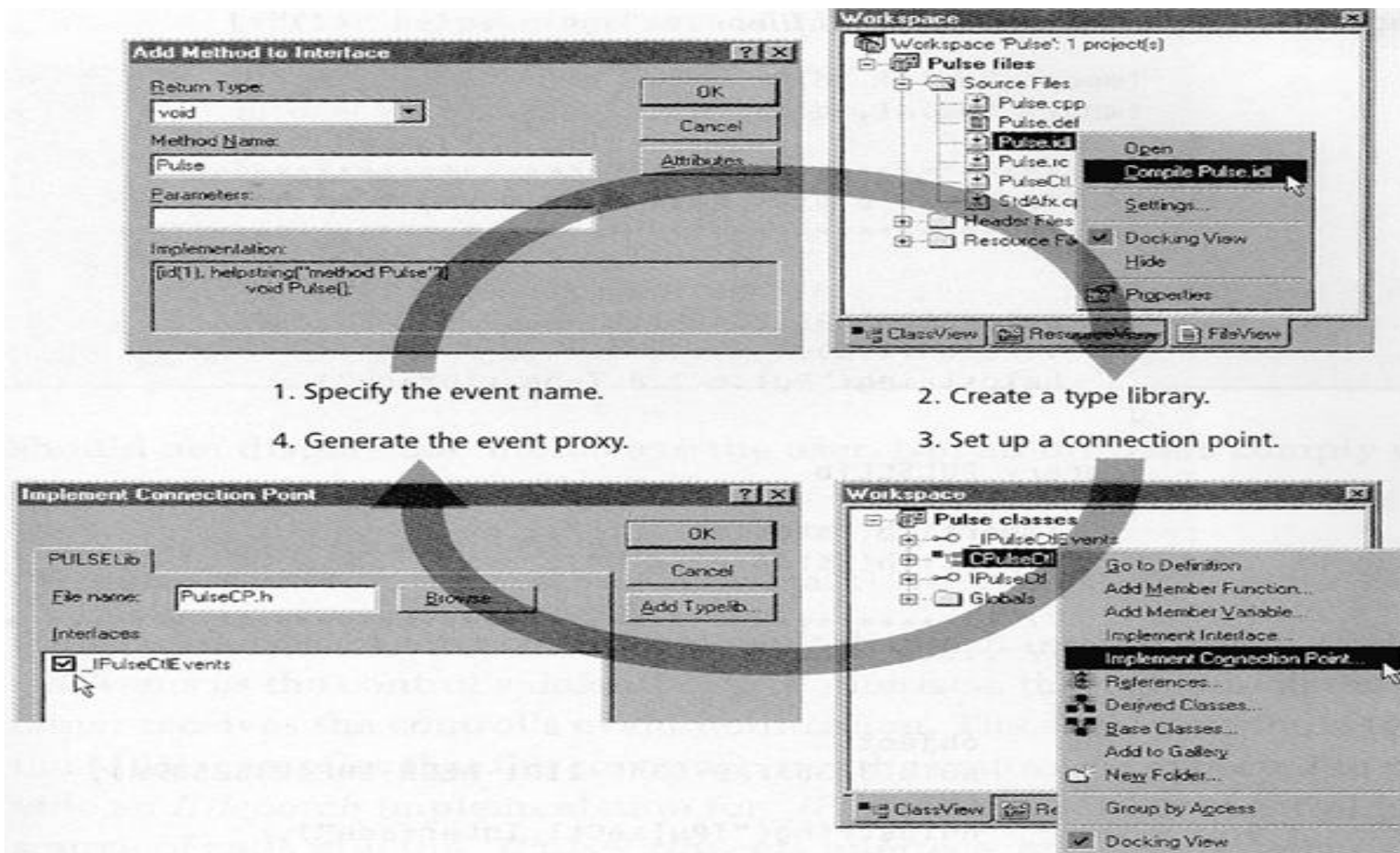


图 10-11 为 ATL ActiveX 项目加入一个事件

第二个块用花括号括起来，包含有接口的方法列表。列表 10-1 表示了在新安排代码之后应该有的结果。如果按照这些步骤建立 Pulse 项目，你自己的 IDL 文件的 GUID 数目与列表中所显示的将不会相同。

importlib 指令从 OLE 类型库 StdOle32.tlb 和 StdOle2.tlb 中带入预编译类型信息，这两个类型库通常位于 Windows 系统或者 System32 目录中。尽管并非必要，接口名字 _IPulseCtlEvents 仍然以下划线开始。这个约定将通知接口浏览器，_IPulseCtlEvents 接口属于控件，而且浏览器不能将接口显示给用户。

列表 10-1 改正后的 Pulse.idl 文件

```
import "oidl.idl";

import "ocidl.idl";

// *****

// First two blocks have been moved from here...

// *****
```



```
//      ...to here:
```

```
[
```

```
    object,
```

```
    uuid(3B365FA9-C3AE-11D1-BEC9-E0F4E352507A),
```

```
    dual,
```

```
    helpstring("IPulseCtl Interface"),
```

```
    pointer_default(unique)
```

```
]
```

```
interface IPulseCtl : IDispatch
```

```
{
```

```
    [propget, id(1), helpstring("property nInterval")] HRESULT nInterval([out, retval] long
```



```

        *pVal);

[id(2), helpstring("method StartPulse")] HRESULT StartPulse(long nRate);

[id(3), helpstring("method EndPulse")] HRESULT EndPulse();

[id(4), helpstring("method _OnTimer")] HRESULT _OnTimer();

};

// *****

[

    uuid(3B365FAA-C3AE-11D1-BEC9-E0F4E352507A),

    helpstring("_IPulseCtlEvents Interface")

]

dispinterface _IPulseCtlEvents

{

```

```
    properties:

    methods:

    [id(1), helpstring("method Pulse")] void Pulse();

};

[

    uuid(8C9BABDD-BCE5-11D1-BEC9-D43CA8CB2F51),

    helpstring("PulseCtl Class")

]

coclass PulseCtl

{

    [default] interface IPulseCtl;
```

```
[default, source] dispinterface _IPulseCtlEvents;  
  
};  
  
};
```

然而，并不是所有的浏览器都遵守这个约定。

```
[default] interface IPulseCtrl;
```

```
[default, source] dispinterface _IPulseCtlEvents;
```

下面这几行规定 `IPulseCtrl` 为控件的默认调度接口，而 `_IPulseCtlEvents` 作为控件的默认源接口，通过这些包容器接收控件的事件通知信息。`source` 告诉 MIDL 编译器，包容器而不是控件，将为 `_IPulseCtlEvents` 提供 `IDispatch` 实现。这个控件是 `_IPulseCtlEvents` 接口中调用的源，而且包容器的事件 `IDispatch` 是事件的接收器。

下面的这一行通过内部名字规定了控件的事件，而且将事件与唯一的由 `id` 关键字规定的调度标识符值联系在一起：

```
[id(1), helpstring("method pulse")] void Pulse();
```

这个控件并不直接调用 `Pulse` 函数来启动事件，而是调用一个名为 `Fire_Pulse` 的由 ATL 加入到项目中的封装函数。这个封装函数称为代理，依次调用包装容器的 `IDispatch::Invoke` 方法，提供调度标识符号码，以识别启动的事件。以上就是一个代理如何得到它的名字的过程。`Fire_Pulse` 函数作为包装容器事件处理程序函数的替身或者代理，当启动一个事件时，作为控件调用的现场，而不用担心调用最终如何到达客户的细节。代理函数是单一代理类的成员，单一代理类作为导出 `CPulseCtl` 的基类，允许控件在 `CPulseCtl` 实现的任何地方启动事件。代理类的角色与 COM 所设置的用于调度线程间调用的代理对象是不同的。类和调度对象通常被简称为“`proxy`”，所以有时候很容易混淆它们。但它们不是同一件事。

在执行 `Implement Connection Point`（实现连接点）命令时，我们在项目内增加了 `IConnectionPoint` 接口的实现，通过这个接口包装容器可以决定 `ActiveX` 控件支持哪一个连接点。这个命令检查控件的类型库文件，从中提取接口名字 `_IPulseCtlEvents` 和 `IPulseCtl` 以创立代理类。一个 `ActiveX` 控件通常在控件的可执行文件中作为原始数据包含它自己的类型库，并允许类型浏览器如 `ClassWizard` 来访问数据。然而，在控件的开发阶段，类型库以 `TLB` 文件的形式独立存在。

ATL 的 `IConnectionPointImpl` 模板为新 `CProxy_IPulseCtlEvents` 代理类定义了一个基准。这个类的代码驻留在 `PulseCP.h` 文件中，以子函数的形式包含了 `Pulse` 的单个事件代理：

```
template <class T>

class CProxy_IPulseCtlEvents : public IConnectionPointImpl<T,

    &DIID__IPulseCtlEvents,

    CComDynamicUnkArray>

{

    //Warning this class may be recreated by the wizard.

public:

    VOID Fire_Pulse()

    {

        T* pT = static_cast<T*>(this);

        int nConnectionIndex;

        int nConnections = m_vec.GetSize();
```

```
        for (nConnectionIndex = 0; nConnectionIndex < nConnections;
nConnectionIndex++)

    {

        pT->Lock();

        CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);

        pT->Unlock();

        IDispatch* pDispatch = reinterpret_cast<IDispatch*>(sp.p);

        if (pDispatch != NULL)

        {

            DISPPARAMS disp = { NULL, NULL, 0, 0 };

            pDispatch->Invoke(0x1, IID_NULL, LOCALE_USER_DEFAULT,
```

```
DISPATCH_METHOD, &disp, NULL, NULL, NULL);

        }

    }

}

};
```

注意，Fire_Pulse 代理函数的类型是 void。事件与方法不同，它从不返回值。

通过它的基类，CProxy_IPulseCtlEvents 为由第二个模板参数识别的接口实现了连接点。在我们的例子中，接口是_IPulseCtlEvents，而它的调度接口标识符是 DIID_IPulseCtlEvents，由 Pulse_i.c 文件进行定义。第三个模板参数规定了处理连接的 ATL 类。这个默认的 CComDynamicUnkArray 类允许的连接数目不限；而可选的 CComUnkArray 类只允许数目固定的连接。

步骤 6：加入 CTimer 类

Pulse 可以捕获 WM_TIMER 消息来激活它的事件启动过程，但是那样将需要创

立一个窗口来接收消息。窗口将消耗系统资源，需要时间来建立和关闭，而且，向 Pulse 这样的流型 ActiveX 控件应尽可能避免创立窗口。幸运的是，Visual C++ 中所包含的 AtlButton 例子项目包含了一个简单的安全线程 CTimer 类，这个类正是我们所需要的。由于代码不是由 ATL 向导产生的，列表 10-2 中列出了它的全部内容。

列表 10-2 Timer.h 文件

```
// Timer.h : Declaration of the CTimer class (borrowed from  
  
//          Samples\VC98\ATL\AtlButto\AtlButto.h file)  
  
template <class Derived, class T, const IID* piid>  
  
class CTimer  
{  
  
public:  
  
    CTimer() { m_bTimerOn = FALSE; } // Timer is OFF
```



```
HRESULT TimerOn( DWORD dwTimerInterval )      // Arm the timer

{

    Derived* pDerived = ((Derived*) this);

    m_dwTimerInterval = dwTimerInterval;

    if (m_bTimerOn)    // If already on, just change interval

        return S_OK;

    m_bTimerOn= TRUE;

    m_dwTimerInterval = dwTimerInterval;

    m_pStream = NULL;
```

```
HRESULT hRes = CoMarshalInterThreadInterfaceInStream(  
    *piid, (T*)pDerived, &m_pStream );
```

```
// Create thread and pass the thread proc the this ptr
```

```
m_hThread = CreateThread(NULL, 0, &_Apartament,  
    (PVOID) this, 0, &m_dwThreadId);
```

```
return S_OK;
```

```
}
```

```
void TimerOff()                // Disable the timer
```

```
{
```

```
    if (m_bTimerOn)
```

```
{  
  
    m_bTimerOn = FALSE;  
  
    AtlWaitWithMessageLoop( m_hThread );  
  
}  
  
}
```

// Implementation

private:

```
static DWORD WINAPI _Apartment( PVOID pv )  
  
{  
  
    CTimer<Derived, T, piid>* pThis = (CTimer<Derived, T, piid>*) pv;  
  
    pThis->Apartment();  
  
}
```

```
return 0;
```

```
}
```

```
DWORD Apartment()
```

```
{
```

```
    CoInitialize(NULL);
```

```
    HRESULT hRes;
```

```
    m_spT.Release();
```

```
    if (m_pStream)
```

```
    {  
        hRes = CoGetInterfaceAndReleaseStream(  
            m_pStream, *piid, (PVOID*) &m_spT );
```

```
    }  
}
```

```
// Main timer loop that periodically calls _OnTimer

while(m_bTimerOn)

{

    Sleep( m_dwTimerInterval );

    if (!m_bTimerOn)

        break;

    m_spT->_OnTimer();

}


m_spT.Release();           // When TimerOff function sets

CoUninitialize();         // m_bTimerOn = FALSE, unregister
```

```
        return 0; // and quit
```

```
    }
```

```
public:
```

```
    DWORD    m_dwTimerInterval;
```

```
    BOOL     m_bTimerOn;
```

```
private:
```

```
    HANDLE m_hThread;
```

```
    DWORD  m_dwThreadId;
```

```
    LPSTREAM m_pStream;
```

```
    CComPtr<T> m_spT;
```

```
};
```

通过列表，可以看出 CTimer 分为几步，其原因还不是很立即就可以看出来。类通过它的 TimerOn 函数封装了计时器，这个函数将创立一个新线程，以调用 Sleep 函数，以在需要时进入休眠。当线程被唤醒时，它调用我们在步骤 4 中加入的 _OnTimer 方法，以表明间隔已经结束。线程然后再次回到休眠，这个过程在一个循环中不断重复，直至 TimerOff 函数被调用。这个类是非常有趣的，因为它执行自己的线程之间的用户调度，确保 _OnTimer 函数不是从新线程受到调用，而是从启动计时器的原始线程受到调用。在我们的例子中，这是 Pulse 控件的主线程，从这里 _OnTimer 可以安全的启动 Pulse 事件。调度回到主线程对于像 Pulse 一样的公寓线程控件是必要的一步，因为 COM 的规则指出一个对象必须在客户的公寓内启动事件。也就是说，在与使对象实例化的同一个客户 STA 线程上。

如果作为新项目创立 Pulse，简单地从公司 CD 上拷贝 Timer.h 文件到自己的项目文件夹中。这个文件是一个头文件，所以没有必要用 Add To Project（加入到项目中）命令将其加入到项目中。

步骤 7：编辑 PulseCtl.h 文件

在步骤 2 中，选择最简单的可以从 ATL Object Wizard（对象向导）中得到的对象类型，这将设置 CPulseCtl 类，以便仅仅继承 3 个基类和 2 个接口。一个真正的 ActiveX 控件必须实现比这些更多的接口，所以，在这一步，我们将扩展类的继

承列表，以包含一个典型包容器希望一个控件所实现的附加接口。

在文本编辑器打开 PulseCtl.h 文件，然后加入下面灰色所示的两个 #include 语句，一个用于我们在前面的步骤中创立的 CTimer 类，而另一个用于引入项目所需要的附加接口实现：

```
#include "resource.h"    //main symbols
```

```
#include "PulseCP.h"
```

```
#include "timer.h"
```

```
#include <atlctl.h>
```

然后，在 CPulseCtl 继承列表中增加下列内容。这些行的实际顺序并不重要，但是，注意列表中的所有入口都有一个逗号结尾，最后一个例外。

```
class ATL_NO_VTABLE CPulseCtl :
```

```
    public CComObjectRootEx<CComSingleThreadModel>,
```

```
    public CComCoClass<CPulseCtl, &CLSID_PulseCtl>,
```

```
    public IConnectionPointContainerImpl<CPulseCtl>,
```



```
public IDispatchImpl<IPulseCtl, &IID_IPulseCtl, &LIBID_PULSELib>,

public CTimer<CPulseCtl, IPulseCtl, &IID_IPulseCtl>,

public IObjectWithSiteImpl<CPulseCtl>,

public CComControl<CPulseCtl>,

public IPersistStreamInitImpl<CPulseCtl>,

public IOleControlImpl<CPulseCtl>,

public IOleObjectImpl<CPulseCtl>,

public IViewObjectExImpl<CPulseCtl>,

public IOleInPlaceObjectWindowlessImpl<CPulseCtl>,

public IPersistStorageImpl<CPulseCtl>,

public IProvideClassInfo2Impl<&CLSID_PulseCtl,

                                &DIID__IPulseCtlEvents, &LIBID_PULSELib>,
```

```
public CProxy_IPulseCtlEvents< CPulseCtl >

{
```

尽管 Pulse 控件是不可见的（而不是无窗口），然而，CpulseCtl 类必须从 IOleInPlaceObjectWindowlessImpl 中导出（复习第 9 章内容，无窗口控件取决于显示服务的控件）。这是 ATL 对于像 Pulse 这样的不可视控件不进行优化的例子。ATL 没有提供我们的控件所需要的 IOleInPlaceObject 接口的单独实现，而是仅仅提供了 IOleInPlaceObjectWindowless，这是对 IOleInPlaceObject 的扩展，这将增加对窗口信息的支持和拖放操作。Pulse 并不需要这些额外的方法，但是必须包含它们，以实现 IOleInPlaceObject。

按照类继承列表，为 FinalRelease 子函数加入原型，这个子函数在卸载 Pulse 控件时，被 ATL 的 CComObject 调用：

```
public:
```

```
HRESULT FinalRelease();
```

```
CPulseCtl()
```

```
{
```

```
}
```

注意 CPulseCtl 并没有声明一个类析构函数。这是因为析构函数在从 CPulseCtl 导出的 ATL 基类中是不可视的，所以类并不能确实地假定它的析构函数将永远不会被调用。一个使用 ATL 的 ActiveX 控件类执行在 FinalRelease 函数中的任何必要的清除功能，这将在对象实例被破坏之前被调用。它的必然结果是一个控件必须将它的初始化任务限定在 FinalConstruct 函数中。本章后面出现的第 2 个项目说明了 FinalConstruct 的用法。为了与 CPulseCtl 继承性列表内的接口模板匹配，必须为类的 COM 映射加入相应的入口，如下所示：

```
BEGIN_COM_MAP(CPulseCtl)
```

```
    COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
```

```
    COM_INTERFACE_ENTRY(IPulseCtl)
```

```
    COM_INTERFACE_ENTRY(IDispatch)
```

```
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
```

COM_INTERFACE_ENTRY(IObjectWithSite)

COM_INTERFACE_ENTRY(IViewObjectEx)

COM_INTERFACE_ENTRY(IViewObject2)

COM_INTERFACE_ENTRY(IViewObject)

COM_INTERFACE_ENTRY(IOleInPlaceObjectWindowless)

COM_INTERFACE_ENTRY(IOleInPlaceObject)

COM_INTERFACE_ENTRY2(IOleWindow, IOleInPlaceObjectWindowless)

COM_INTERFACE_ENTRY(IOleControl)

COM_INTERFACE_ENTRY(IOleObject)

COM_INTERFACE_ENTRY(IPersistStreamInit)

COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)

COM_INTERFACE_ENTRY(IPersistStorage)

```
COM_INTERFACE_ENTRY(IProvideClassInfo)
```

```
COM_INTERFACE_ENTRY(IProvideClassInfo2)
```

```
END_COM_MAP()
```

IProvideClassInfo2 是新的 OLE 接口，替换了旧的 IProvideClassInfo。一些包容器不认识 IProvideClassInfo2，所以映射包含了 IProvideClassInfo 的入口。因为 IProvideClassInfo2 代表旧接口，所以加入的内容不会增加 Pulse 的代码大小。IProvideClassInfo 和 IProvideClassInfo2 支持 Pulse 的事件启动。两个接口都提供了 GetClassInfo 方法，它将提供属于 PulseCtl 组件类对象的类型信息。这个类型信息来自控件的类型库，将告诉客户如何为 Pulse 事件建立它的处理程序函数。

除了建立 Uniform Data Transfer（统一数据传送）的核心之外，IDataObject 接口提供了向客户通知数据变化的方法。一个实现 IAdviseSink 接口的客户调用控件的 IDataObject::Advise 方法在控件的数据被改变时，开始接收通知。这并不适用于 Pulse，而且控件可以在不支持 IDataObject 的情况下正常工作。然而，一些客户，例如 Test Container 实用程序要求接口建立通知连接。

确保 COM 映射包含 IConnectionPointContainer 的入口：

```
COM_INTERFACE_ENTRY_IMPL(IConnctionPointContainer)
```

如果没有在步骤 2 中的 Object Wizard(对象向导)中选择 Support Connection Points (支持连接点) 选项, 在 COM 映射中键入入口, 并在类继承性列表中增加 IConnectionPointContainerImpl。一个包容器通过它的 IConnectionPointContainer 接口来查询对象, 以了解对象支持哪些接口, 在我们的例子中是事件接口 _IpulseCtlEvents。包容器的事件接收器与对象的连接和断开将通过对象的 IConnectionPoint 接口进行。COM 映射不需要单独的 IConnectionPoint 接口, 因为 IConnectionPointContainer 提供了 FindConnectionPoint 方法, 它将向 ATL 的代表 _IPulseCtlEvents 的 IConnctionPoint 返回一个指针。图 10-12 说明了包容器如何将它的事件接收器挂接在 Pulse 的事件函数上的步骤。

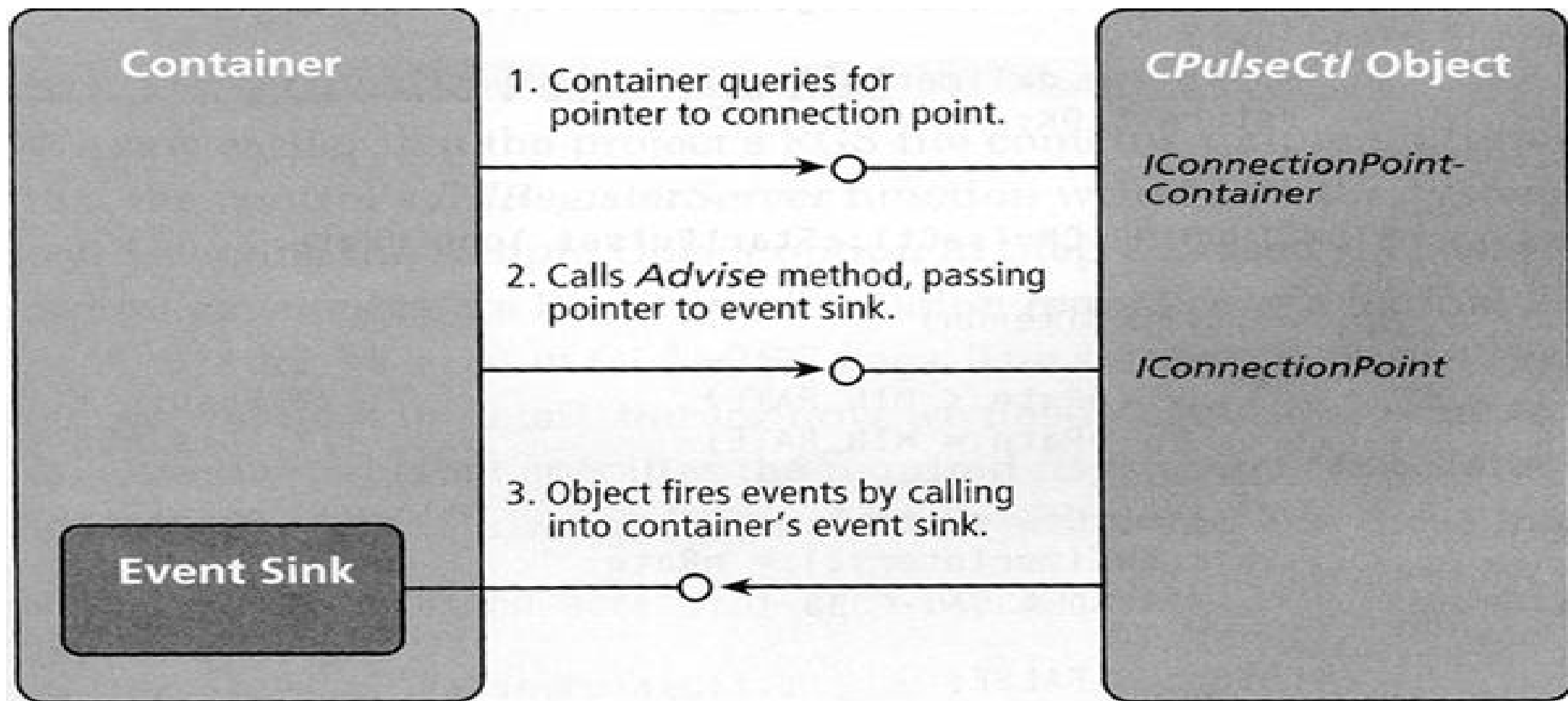


图 10-12 包容器如何建立连接以接收事件启动消息

`FindCorrelationPoint` 和它的姊妹方法 `EnumCorrelationPoints` 读入一个数组，这个数组被称为连接点映射，它包含了控件提供的每一个连接点的接口标识符列表。

Pulse 仅仅支持_IPulseCtlEvents 的一个连接点，这将在 COM 映射后面的连接点映射中规定：

```
BEGIN_CONNECTION_POINT_MAP(CPulseCtl)

    CONNECTION_POINT_ENTRY(DIID__IPulseCtlEvents)

END_CONNECTION_POINT_MAP()
```

在连接点映射下面，加入如下所示的属性映射。这个映射是空的，因为 Pulse 不支持属性工作表，但是，加入到 CPulseCtl 类中的一些接口实现要求映射存在：

```
BEGIN_PROPERTY_MAP(CPulseCtl)

END_PROPERTY_MAP()
```

步 骤 8 ： 编 辑 PulseCtl.cpp 文 件

在 PulseCtl.cpp 文件中加入下面灰色部分所示的指令，以完成存根函数：

```
////////////////////////////////////

// CPulseCtl
```



```
#define    MIN_RATE    10                // Minimum firing rate
```

```
STDMETHODIMP CPulseCtl::get_nInterval( long *pVal )
```

```
{
```

```
    *pVal = m_dwTimerInterval;
```

```
    return S_OK;
```

```
}
```

```
STDMETHODIMP CPulseCtl::StartPulse( long nRate )
```

```
{
```

```
    if (!m_bTimerOn)
```

```
{  
  
    if (nRate < MIN_RATE)                // Ensure firing rate  
  
        nRate = MIN_RATE;                //    is not too low
```

```
    TimerOn( nRate );                    // Start the timer  
  
    m_dwTimerInterval = nRate;  
  
}  
  
return S_OK;  
  
}  
  
return S_FALSE;  
  
}
```

```
STDMETHODIMP CPulseCtl::EndPulse()
```

```
{
```

```
    TimerOff();
```

```
    m_dwTimerInterval = 0;
```

```
    return S_OK;
```

```
}
```

```
STDMETHODIMP CPulseCtl::_OnTimer()
```

```
{
```

```
    Fire_Pulse();
```

```
    return S_OK;
```

```
}
```

```
HRESULT CPulseCtl::FinalRelease()

{

    return EndPulse();

}
```

get_nInterval 方法通知调用者当前的计时器间隔，它被存储在 CTimer::m_dwTimerInterval 成员变量中。因为我们在步骤 2 中规定 nInterval 属性为只读，所以没有匹配的放置函数。StartPulse 和 EndPulse 实现控件的两个方法，启动和停止计时器。当包容器在结束控件后，没有调用 EndPulse 时，FinalRelease 函数调用 EndPulse，以确保计时器的工作线程在控件结束前是正确的。

步骤 9：编辑 Pulse.rgs 文件

早些时候，我们了解到项目的 RGS 文件中包含脚本信息，这是由控件的

DllRegisterServer 函数写入到系统 Registry(注册表)内的。在步骤 2 中选择 Simple Object (简单对象) 选项将是 ATL Object Wizard (对象向导) 丢失某些一般 ActiveX 控件需要的 Registry (注册表) 信息, 例如 OLEMISC 标志列表。稍后的附加内容详细讨论了标志, 但现在我们仅仅需要在 Pulse 的注册数据上增加一个值, 以规定在 MiscStatus Registry 键内所需要的标志。在文本编辑器中打开 Pulse.rgs 文件, 并增加下列内容:

```
ForceRemove {8C9BABDD-BCE5-11D1-BEC9-D43CA8CB2F51} = s 'PulseCtl Class'
```

```
{
```

```
    ProgID = s 'Pulse.PulseCtl.1'
```

```
    VersionIndependentProgID = s 'Pulse.PulseCtl'
```

```
    ForceRemove 'Programmable'
```

```
    InprocServer32 = s '%MODULE%'
```

```
{
```

```
        val ThreadingModel = s 'Apartment'
```

```
}
```

```
ForceRemove 'Control'
```

```
'MiscStatus' = s '0'
```

```
{
```

```
    '1' = s '148624'
```

```
}
```

```
'TypeLib' = s '{3B365F9D-C3AE-11D1-BEC9-E0F4E352507A}'
```

```
}
```

控件入口将 Pulse 作为一个可嵌入的 ActiveX 控件。MiscStatus 的值 148624 代表 5 个标志的组合值：OLEMISC_SETCLIENTSITEFIRST, OLEMISC_INSIDEOUT, OLEMISC_CANTLINKINSIDE, OLEMISC_INVISIBLEATRUNTIME 和 OLEMISC_NOUIACTIVATE，这些都在稍候的附加内容上有说明。

一般地，ATL Object Wizard（对象向导）为控件项目增加了一个默认位图，但对于像 Pulse 这样的简单对象类型不是这样。如果想增加一个小位图，以增强 Pulse 控件的功能，下面将告诉你处理方法。我们已经看见一些包容器，例如 Visual C++ 对话框编辑器，如何向用户显示代表一个控件的工具按钮上的一幅位图。包容器

从 ActiveX 控件的自身资源数据中提取位图图像，图像由系统 Registry（注册表）内的控件的 ToolboxBitmap32 键识别。为了插入 ToolboxBitmap32 键，在 Pulse.rgs 文件中连同其他加入的部分一起加入下面的一行：

```
ForceRemove 'ToolboxBitmap32' = s '%MODULE%', 1'
```

一个控件的位图是完全可选的，但是，为那些控件加入一个专业的方法，是市场所需要的。位图的大小是 16×15 像素，所以一幅 16 色位图占据了空间资源数据段的 512 个字节。从 Insert（插入）菜单中选择 Resource（资源）命令，选择 Bitmap（位图），然后按下 Alt+Enter，以打开 Bitmap Propertiers（位图属性）对话框。改变工作区域的尺寸到 16×15 ，并指定 ToolboxBitmap32 键的资源标识符的值与 ForceRemove 语句中给定的值一样。在我们的例子中，这个值是 1：



OLEMISC 标志

一个 ActiveX 控件注册了一组 OLEMISC 位标志，并作为一个 32 位值存储在系统 Registry（注册表）中。标志包括控件的信息，通知包容器控件的特性，运行首选项和功能。将这个信息公布在 Registry（注册表）中，意味着将来的包容器不需要首先将控件嵌入，以了解它的要求和功能。如果一个控件需要包容器无法提供的服务，包容器可以在不浪费加载控件所需的时间和资源的情况下决定匹配是不合适的。

为决定控件的 OLEMISC 标志的设置，包容器可以调用 `IOleObject::GetMiscStatus` 函数。这个调用并不会将控件载入，因为 OLE 为这个函数提供了默认的实现，

即从 Registry (注册表) 中读出控件的 OLEMISC 标志。在涉及到 OLE 实时库时, 可以用另外的方法, 即容器自己可以从 Registry (注册表) 中直接读出控件的标志, 访问控件的 CLSID 入口下的 \MiscStatus\1 文件夹。

ATL Object Wizard (对象向导) 仅仅为 ActiveX 控件在 RGS 文件中产生 MiscStatus 值, 而对于像 Pulse 这样的简单对象则不是这样的。一般可视的 ActiveX 控件, 例如, MiscStatus 的值为 131,473, 代表下面的 5 个 OLEMISC 位标志:

- OLEMISC_RECOMPOSEONRESIZE——调整大小时, 控件将放置它的显示, 而不是仅仅是比例缩放。因而, 当调整大小时, 容器必须激活对象, 并用新窗口大小尺寸调用控件的 IOleObject::SetExtent 方法。
- OLEMISC_CANTLINKINSIDE——如果对象被拷贝到剪贴板上, 容器必须禁止用户激活与剪贴板连接的对象。
- OLEMISC_INSIDEOUT——ActiveX 控件所需要的。这个标志告诉容器, 控件对象可以在容器窗口内的适当位置被激活, 而容器不必显示任何特殊的用户接口元素, 例如菜单和工具栏, 以使用户与控件相互作用。适当位置激活, 也被称为可视化编辑, 要求容器支持 IOleInPlaceSite 接口。

- `OLEMISC_ACTIVATABLEWHENVISIBLE`——当对象可视时，对象想要被激活。
- `OLEMISC_SETCLIENTSITEFIRST`——仅仅在使用 ActiveX 控件时使用，这个标志指出，控件优先使用 `IOleObject::SetClientSite` 作为它的初始化函数，甚至在调用 `IPersistStreamInit::InitNew` 或者 `IPersistStorage::InitNew`，以从磁盘上提取控件的属性数据之前就是如此。这将允许控件在从永久存储加载信息之前，访问包容器的环境属性。注意 `OleCreate`, `OleCreateFromData`, `OleCreateFromFile`, `OleLoad` 的当前实现和默认处理程序的当前实现并不理解这个值。希望兑现这个标志的控件包容器必须在当前实现这些函数自己的版本，目的是为控件建立正确的初始化顺序。
- 像 `Pulse` 这样的不可视控件一般为 `OLEMISC_ACTIVATABLEWHENVISIBLE` 设置两个标志：
- `OLEMISC_INVISIBLEATRUNTIME`——通知包容器，控件没有用户接口，而且不需要显示在屏幕上。
- `OLEMISC_NOUIACTIVATE`-控件不需要共享用户接口元素，例如菜单，而且运行时不需要输入焦点。

如果控件有 Object Wizard（对象向导）设置的标志所无法识别的其他实时要求，

在建立项目之前，对 RGS 文件中的 MiscStatus 值进行适当的放置。为了得到标志和它们的位值的完整列表，参考 Visual C++在线帮助中的文章“OLEMISC”的内容。

步骤 10：建立和测试 Pulse ActiveX控件

最后的一步将监督完成 Pulse 项目，并提出如何检查新控件的方法。首先我们选择一个建立的目标。ATL COM AppWizard 按照 4 种公布的配置建立一个 ATL 项目：

配置

预处理定义

释放 MinSize

_ATL_DLL

释放 MinDependency

_ATL_STATIC_REGISTRY

Unicode 释放 MinSize

_UNICODE, _ATL_DLL

Unicode 释 放
MinDependency

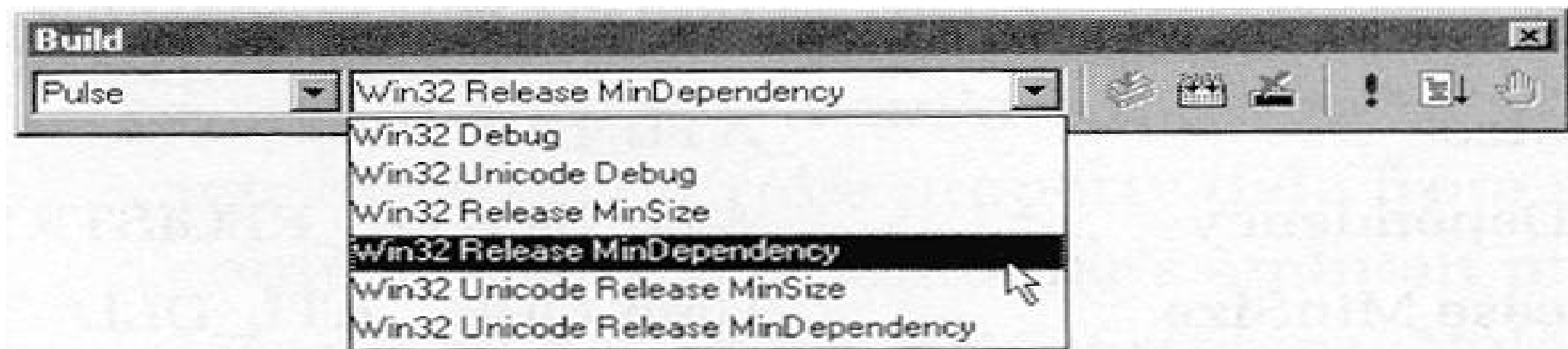
_UNIDOCCE,
_ATL_STATIC_REGISTRY

对于 ANSI 和 Unicode 控件项目，MinSize 和 MinDependency 目标为 ATL 服务提供一种选择机会，即控件依赖于一个辅助的实时文件，或者将它所需要的全部代码合并到它自己的可执行文件内。这是在减小文件大小和减小运行时相关性之间进行选择，非常类似在 MFC 项目中，必须在与 MFC 库静态或者动态链接之间进行选择。

MinSize 配置将控件动态链接在 Atl.dll 上，这是 Visual C++ 安装在 Windows\System 文件夹下的一个 54KB 的库文件，从而减小控件的大小。当控件运行时，它为控件所要求的服务函数而调用 Atl.dll。这种安排的结果是，当有几个链接在 Atl.dll 上的 ActiveX 控件一起运行时，将使内存的利用效率最高。在这

种情况下，MinSize 目标也可以在通过网络或者 Internet 传送控件时，减小下载时间，这是因为控件的组合文件大小被减小了——即使考虑加上 Atl.dll 也是如此，Atl.dll 将与控件一起传送。

MinDependency 配置将设置控件项目，使编译器扩展类模板到整个类，而不是调用 Atl.dll 的存根函数。在这种配置下，控件自身包括它需要的接口实现代码，就像 ATL 服务被静态链接一样（这仅仅是一个类比，因为对于 ATL 没有像 MFC 那样的静态库 LIB 文件）。产生的 ActiveX 控件并不依赖于 Atl.dll 文件，它用较大的执行映像换来了独立性。MinDependency 目标最好用于那些不会与其他 ATL 控件一起运行的单个的控件。因为 Pulse 符合这样的条件，所以在 Build 工具栏中选择 Release MinDependency 配置：



从配套光盘上安装的项目文件将目标名字简化为 MinSize 和 MinDep，这样可以使文件夹名字小于 8 个字母。如果没有比在系统注册表中减小它们所占的空间更重要的原因，你最好为你自己的 ATL 项目选择短目标名字。为了修正 COM AppWizard 所设置的目标名字，暂时关闭工作空间，然后在文本编辑器中打开项目的 DSP 文件。用 Replace（替换）命令，用“MinDep”替换所有存在的“Release MinDependency”和“ReleaseMinDependency”。按照类似的步骤，缩短 MinSize 目标的名字。将 DSP 文件存盘，并重新打开项目。你就可以看见在 Build（建立）工具栏中列出的新目标名字。

至今为止，我们还没有讨论编译器优化，那是第 12 章的内容，但是，Visual C++ 优化器可以增加执行速度，或者减小代码大小。后者通常是 ActiveX 控件的最佳选择，所以 COM AppWizard 为你预先选择了小代码空间优化设置。如果这不是你需要的，改变 Project Setting（项目设置）对话框的 C++ 选项卡的选项。在你

选择目标配置之后，单击 Build（建立）菜单上的 Build Pulse.dll 命令，编译和链接 Pulse 控件。如果所有编译和链接步骤成功结束，Visual C++ 将运行 RegSvr32.exe 来注册控件，并在 Output（输出）窗口中显示结果：

```
Registering ActiveX Control...
```

```
Regv32: DllRegisterServer in .\MinDep\Pulse.ocx succeeded
```

注意：如果你的 ActiveX 控件需要 C 实时服务，首先除去阻止链接到运行库的预处理定义。在 Project Setting（项目设置）对话框的 C++ 选项卡内，删除标有 Preprocessor Definitions（预处理定义）的框内的常量 _ATL_MIN_CRT。

列表 10-3 所列出的 Tumble2.htm 文档展示了新控件。如果你按照这里所描述的步骤创立 Pulse 控件，则它的一个类标识符与配套光盘上的 Pulse.ocx 控件所用的不同。在这种情况下，你必须在 Tumble2.htm 文档中放置对 Pulse 对象的 classid 语句：

```
<OBJECT
```

```
classid="clsid:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx"
```

```
id=pulse1
```

>

项目的 IDL 文件提供了你所需要的新类标识符。从 Pulse.idl 文件中最后一行的 uuid 语句将标识符拷贝到剪贴板内，然后将其粘贴在 Tumble2.htm 文档内，以改变 classid 的值。尽管 Tumble2 使用不同的计时器，它的动画显示与原始 Tumble 文档相同。

```
<HTML>

<HEAD>

<TITLE>Tumbling Text Example 2 (Chapter 10)</TITLE>

</HEAD>

<BODY>


<OBJECT

classid="clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"

id=label
```



```
width=150
```

```
height=150
```

```
>
```

```
<PARAM NAME="Angle" value="0">
```

```
<PARAM NAME="Alignment" value="7">
```

```
<PARAM NAME="BackStyle" value="1">
```

```
<PARAM NAME="BackColor" value="255">
```

```
<PARAM NAME="FontBold" value="-1">
```

```
<PARAM NAME="Caption" value="Click Here">
```

```
<PARAM NAME="FontName" value="Times New Roman">
```

```
<PARAM NAME="FontSize" value="18">
```

```
</OBJECT>
```

```
<SCRIPT LANGUAGE="VBSCRIPT">
```

```
sub    label_Click
```

```
    label.Caption    = "Tumbling text!"
```

```
    label.FontBold    = "0"
```

```
    label.FontUnderline = "-1"
```

```
    label.FontItalic= "-1"
```

```
    pulse1.StartPulse( 100 )
```

```
end sub
```

```
sub    pulse1_Pulse
```

```
    label.Angle = (label.Angle + 5) mod 360
```

```
end sub
```

</SCRIPT>

</BODY>

</HTML>

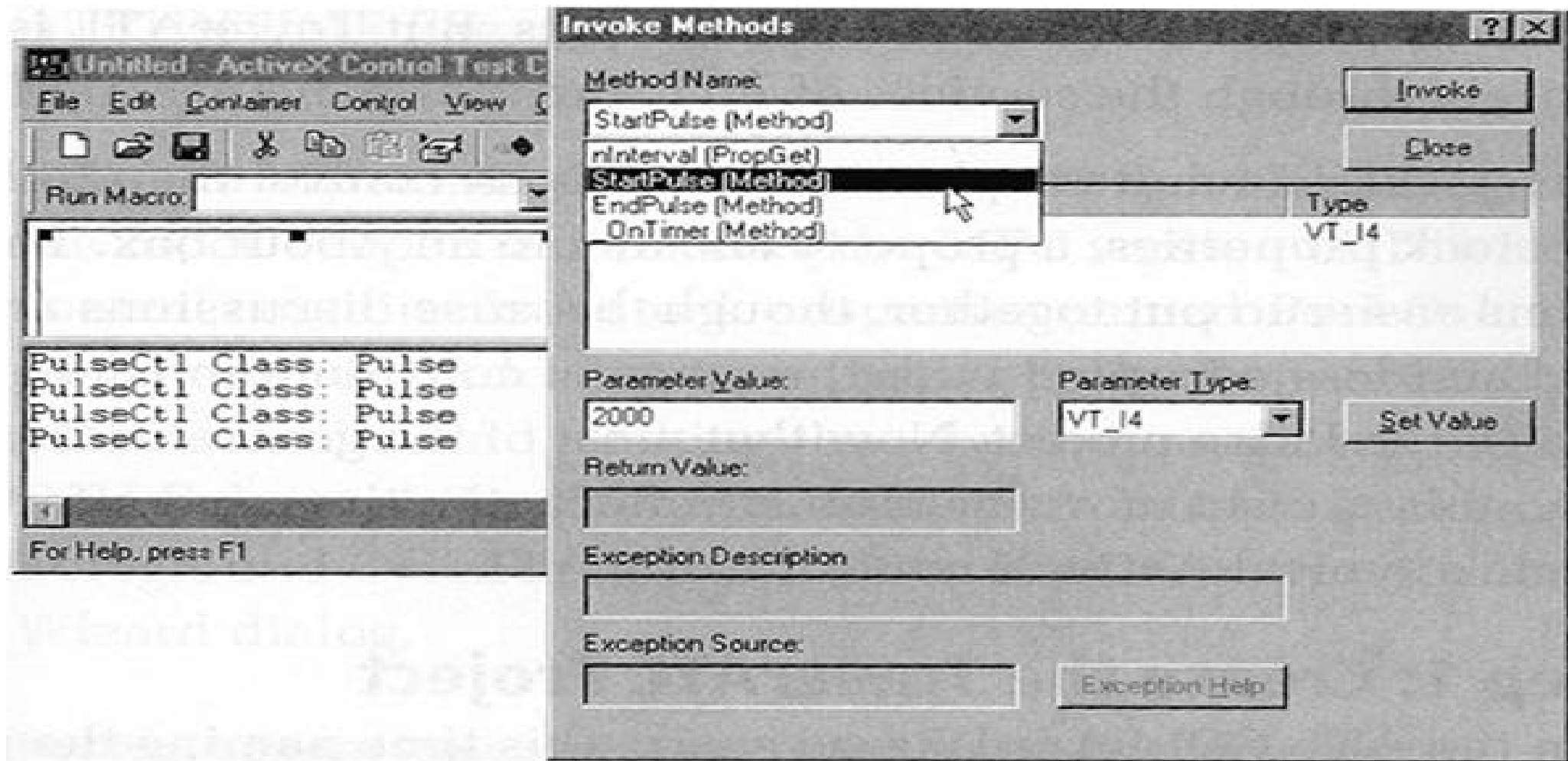


图 10-13 在 Test Container 中测试 Pulse ActiveX 控件

Test Container（测试包容器）实用程序也可以成功地嵌入我们的新控件，就像图 10-13 中说明的那样。启动 Test Container（测试包容器），然后选择 New Control

(新控件)工具,并双击列表中的 PulseCtl 类入口,加载 Pulse 控件(列表是按照阿拉伯字母排列的;在键盘上按下 P,将自动滚动到以 P 打头的第一个控件处)。一旦控件被加载到 Test Container(测试包容器)内,单击 Invoke Methods(激活方法)工具,按照图 10-13 所示,在 Method Name Box(方法名称框)中选择 StartPulse 方法,将 nRate 参数设置为 2000,然后单击 Invoke(激活)按钮。这将使控件每 2 秒启动它的 Pulse 事件,Test Container(测试包容器)将为事件日志增加一个新入口而检查这一点。激活 EndPulse 方法结束启动。

如果想了解 Pulse.ocx 如何被嵌入到一个普通的包容器中应用程序中,运行配套光盘上文件夹 Code\Chapter.10\Hour2。Hour2 使用 Pulse 控件作为它的计时器,而不是 IETimer 控件,除了这一点之外,Hour2 与第 8 章 Hour 程序相同。

例子 2: TowerATL ActiveX 控件

Pulse 项目证明 ATL 可以帮助你创立比利用 MFC 产生的控件更小的 ActiveX 控件,但是,对 ATL 和 MFC 的精确比较,只有通过利用两种工具建立同样的控件才能进行。这正是我们在这一章所讲述的内容。正如标题所指出的那样,TowerATL 控件复制第 9 章的 Tower 项目,输出相同的属性、方法和事件。但是,TowerATL 是完全利用 ATL 的服务创立的,没有用到 MFC。

TowerATL 是一个比 Pulse 更复杂的 ActiveX 控件，包含有栈属性、属性工作表和一个 About 框。尽管如此，这个项目看起来比较容易，因为讨论更加紧凑，而且我们在 Pulse 项目中所注意过的部分占据较小篇幅。现在已经排除了大多数的不相关的东西，这个项目更加精确地反映了用 ATL 创立一个典型 ActiveX 控件所需要的时间和努力。

步骤 1：创立 TowerATL 项目

再次运行 ATL COM AppWizard，这一次将项目命名为 TowerATL。单击 Finish（结束）按钮接受默认设置，这将是 TowerATL 控件成为一个不需要 MFC 的动态链接库。当 COM AppWizard 结束后，从 Insert（插入）菜单中选择 New ATL Object（新 ATL 对象），启动 ATL Object Wizard（对象向导）。选择对话框的第一个列表内的控件，然后双击标有 Full Control（完整控件）的图标。TowerATL 支持比 Pulse 更多的接口，我们将需要像 IQuickActivate 和 ISpecifyPropertyPages 这样的接口，它们由 Full Control（完整控件）选项加入到源代码中。

注意，在对话框中，ATL 将控件的属性页作为一个单独的对象。在结束这个项目之前，返回到 Object Wizard（对象向导），以增加一个属性页对象。在 Properties（属性）对话框的 Names（名称）选项卡中键入 TowerCtl 作为短名字，接受其他框内的默认名字。在 Attributes（属性）选项卡内，与我们在 Pulse 项目中一样，选择 Support Connection Points（支持连接点）复选框。接受选项卡内的其他默认

设置，启用公寓线程、双接口和聚合。

Miscellaneous（杂项）选项卡包括设置各种 OLEMISC 标志的开关。例如，选择 Acts Like Button（按钮功能）复选框，设置 OLEMISC_ACTSLIKEBUTTON 标志，这将告诉容器，控件将对鼠标单击作出响应，并通常作为一个按钮动作。Acts Like Label（选项卡功能）复选框设置 OLEMISC_ACTSLIKELABEL 标志，这将通知容器，这个控件作为按选项卡顺序紧跟其后的控件的选项卡使用，就是在容器的 RC 脚本内列出的下一个控件。Invisible At Runtime（运行时不可见）复选框将打开 OLEMISC_INVISIBLEATRUNTIME 标志，这适合像 Pulse 这样的在激活时仍然不可视的控件。

跳过 Miscellaneous 选项卡，接受它的默认设置，并进入 Stock Properties（栈属性）选项卡。与 Tower 的 MFC 版本类似，TowerATL 控件包括 4 个栈属性，名字为 BackColor, ForeColor, Caption 和 Font。在列表中选择每一个属性，并单击 > 按钮，就可以在 Supported（支持）框内增加这些属性，如图 10-14 所示（BackColor 和 ForeColor 属性在列表中标为 Background Color 和 Foreground Color）。单击 OK 按钮，关闭 ATL Object Wizard（对象向导）对话框。

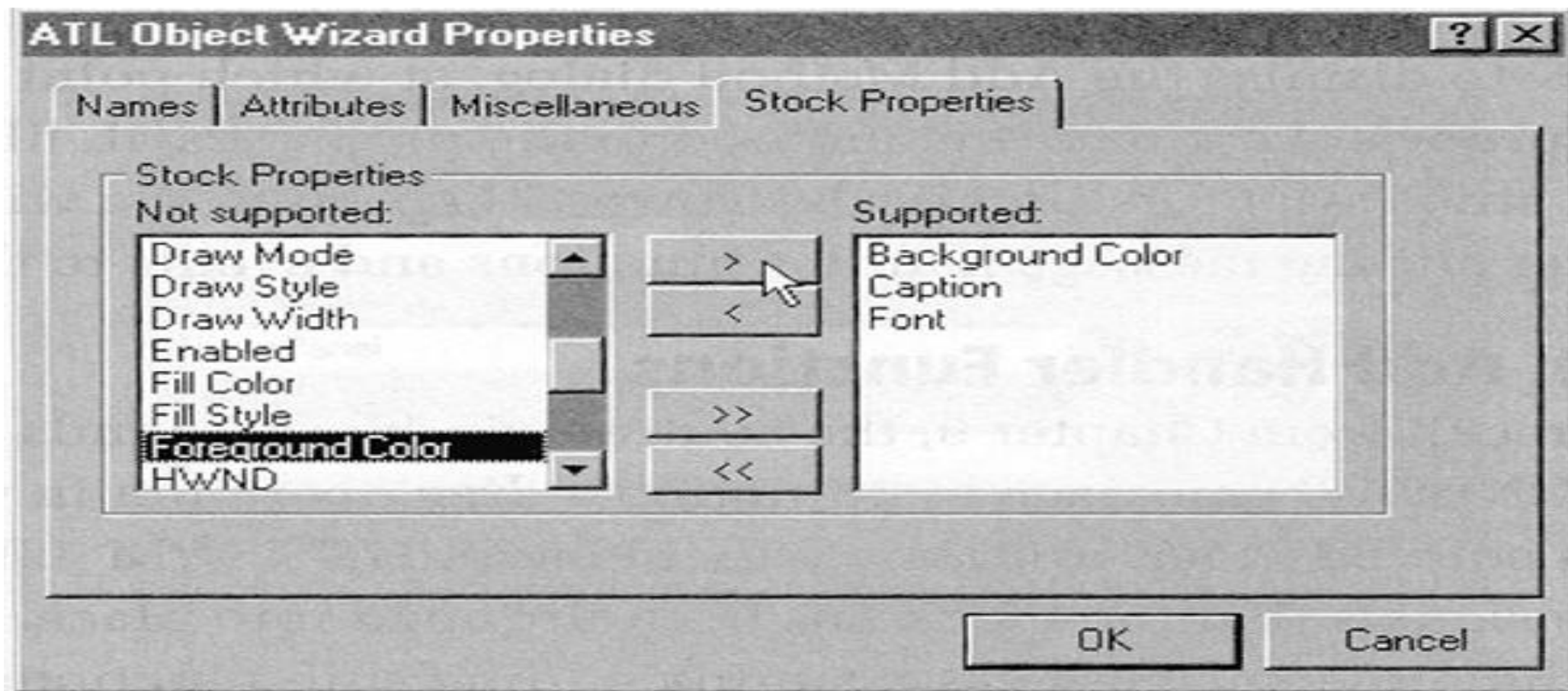


图 10-14 在 ATL Object Wizard (对象向导) 内选择栈属性

步骤 2：加入 nCurrentBlock 用户属性

与 4 个栈属性一起，TowerATL 具有一个称为 nCurrentBlock 的用户属性，这个属

性保留了一个数，用于识别正在被拖动的块。从技术上讲，它是由通过 `get_nCurrentBlock` 方法显露这个属性的对象所继承的 `ITowerCtl` 接口。为了给 `ITowerCtl` 接口加入这个用户属性，在 `ClassView` 窗口中展开 `TowerATL` 类列表，用鼠标右键单击 `ITowerCtl` 入口，然后选择 `Add Property`（加入属性）命令，以激活 `Add Property To Interface`（为接口加入属性）对话框。选择 `short` 作为属性类型，并键入 `nCurrentBlock` 作为属性的名字。就像我们在第 9 章中所做的一样，使 `nCurrentBlock` 属性具有只读特性，以阻止一个容器应用程序改变它的值。清除标有 `Put Function`（放置函数）的单选钮，以告诉 `Object Wizard`（对象向导）不要为这个属性产生放置方法，然后关闭这个对话框。

步骤 3：加入 `Reset` 方法

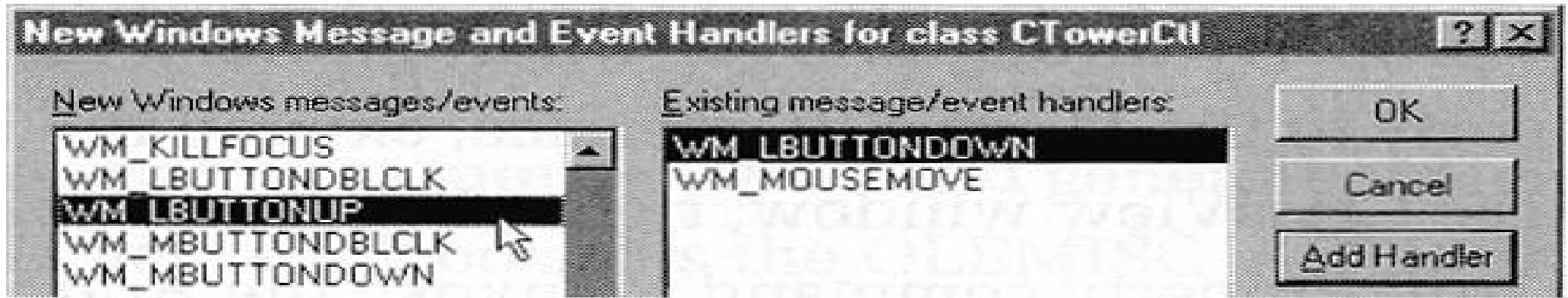
`TowerATL` 导出了一个称为 `Reset` 的方法函数，容器应用程序利用这个函数告诉控件再次启动程序，重新启动左面板中的所有块。再次用鼠标右键单击 `ClassView` 中的 `ITowerCtl` 接口，选择 `Add Method`（加入方法）命令，然后在 `Add Method To Interface`（为接口加入方法）对话框中键入 `Reset` 作为函数名字。这个方法没有参数，所以使参数框保持空白。

单击 `OK` 关闭 `Add Method`（加入方法）对话框，此时 `Visual C++` 将在这个项目的 `IDL` 文件中为这个方法加入适当的代码，并在 `TowerCtl.cpp` 文件中写入一个存根 `Reset` 函数。在控件中加入消息处理程序和事件之后，我们将编辑这些代码。

步骤 4：加入处理程序函数

如果你回忆第 9 章的内容，控件窗口将对 `WM_LBUTTONDOWN` 消息作出响应，方法是初始化一个拖动操作，在这个操作中，用户将一个彩色块从一个面板移动到另一个。`WM_LBUTTONDOWN` 消息表示这个块已经被拖动到位。项目的第 4 步非常简单，只需要加入存根函数以处理两条消息。

在 `ClassView` 窗格中，用鼠标右键单击 `CTowerCtl` 入口，并从菜单中选择 `Add Window Message Handler`（加入窗口消息处理程序）。双击列表中的 `WM_MOUSEMOVE`，`WM_LBUTTONDOWN` 和 `WM_LBUTTONUP`，以将它们加入到右边的框中，然后关闭对话框。



步骤 5：加入事件

与原来的 Tower 控件一样, TowerATL 启动 5 个事件, 名称分别为 Click, FromPanel, ToPanel, Error 和 Winner 它们共同保证使包容器了解对控件所发生的事件。在项目中加入这些事件的步骤与我们对 Pulse 控件的操作是一样的：

1. 在 ClassView 窗格中用鼠标右键单击 `_ITowerCtlEvents` 入口, 并从关联菜单中选择 Add Methods (加入方法) 命令, 然后在对话框中选择 void 返回类型, 并键入一个函数名字。对所有 5 个事件函数重复这些操作。只有 FromPanel 和 ToPanel 带有参数, 分别为 `nFromPanel` 和 `nToPanel`, 这两个参数的类型都是 `short`。

Add Method to Interface

Return Type:
void

Method Name:
FromPanel

Parameters:
short nFromPanel

Implementation:
[id(2), helpstring("method FromPanel")]
void FromPanel(short nFromPanel);

OK
Cancel
Attributes...

2. 在 Workspace 窗口的 FileView 窗格中，用鼠标右键单击 TowerATL.idl 的入口，并选择 Compile（编译）命令，以创建这个项目的类型库文

件。忽略来自 MIDL 编译器中的警告信息“ interface does not conform to [oleautomation] attribute ”。这个警告信息是因为编译器认为 pFont 参数——一个指向 IFontDisp 接口的指针——不是一个兼容 Automation 的类型。但是因为 IFontDisp 来自与 IDispatch，这是一个合法的 Automation 接口，所以这条警告信息是不正确的。

3. 在 ClassView 窗格中，用鼠标右键单击 CTowerCtl，并从菜单中选择 Implement Connection Point（实现连接点）命令。
4. 设置标有 _ITowerCtlEvents 的复选框，然后单击 OK 关闭对话框。

在这里，我们为项目加入事件的步骤与我们在前面章节中用 MFC 开发 Tower 控件时所用的步骤是类似的，尽管看起来可能不是这样。在这些操作之后，第 9 章中的 ControlWizard 将创建一个 ODL 文件（与 IDL 类似），其中包含了这些指令：

```
//{{AFX_OLD_EVENT(CTowerCtrl)
```

```
[id(DISPID_CLICK)] void Click ();
```

```
[id(1)] void FromPanel (short nPanel);
```

```
[id(2)] void ToPanel (short nPanel);
```

```
[id(DISPID_ERROREVENT)] void Error() ;
```

```
[id(3)] void Winner();
```

```
//}}AFX_ODL_EVENT
```

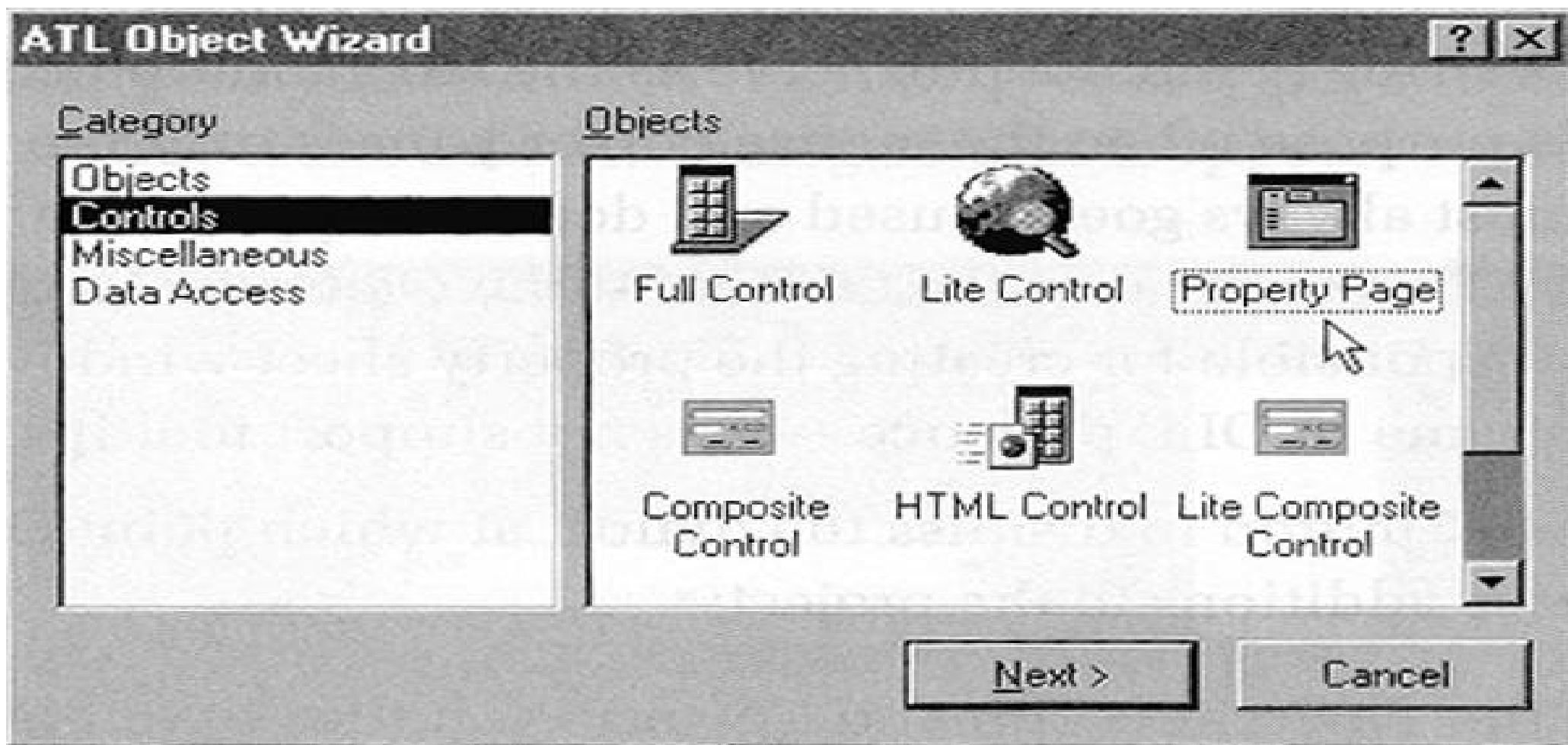
从这些信息中，ClassWizard 产生像 FireClick 和 FireWinner 这样的代理函数，它们将调用 MFC 的 COleControl::FireEvent 函数。然后 FireEvent 将调用包装器的 IDispatch::Invoke 方法，这些与 ATL 产生的代理函数完全一样。

步骤 6：加入属性工作表

第 9 章的 Tower 控件提供了一张属性工作表，这允许用户改变控件的标题，字体和颜色。图 9-13 显示了属性工作表的外观。在这一节，我们将为 TowerATL 创建一个类似的属性工作表。

ATL 将控件的属性工作表的每一页设置为一个独立的对象，通过从 IPropertyPage 导出的一个类来实现。系统的 Msstkprp.dll 库为标有 Font（字体）和 Colors（颜色）的属性页提供默认的实现，这将允许用户改变 Font, BackColor 和 ForeColor 属性。这些属性将依次决定显示在 TowerAtl 窗口顶部的标题的字体和颜色。为提供访问 Caption 属性的方法，它其中包含了控件标题的文本，我们必须像以前对原来的 Tower 控件的操作一样，多加入一个标为 Caption 的属性页。在控

件中加入一个属性页需要 ATL 的 Object Wizard（对象向导）提供服务，为你想加入的每一页运行 Object Wizard（对象向导）。为加入 TowerATL 的新 Caption 页，在 Object Wizard（对象向导）的左框中选择 Controls（控件），并双击 Property Page（属性页）图标：



当出现 ATL Object Wizard Properties (ATL 对象向导属性) 对话框时, 键入 TowerPPG 作为对象的短名字:

ATL Object Wizard Properties [?] [X]

Names | Attributes | Strings

C++

Short Name: TowerPPG

Class: CTowerPPG

.H File: TowerPPG.h

.CPP File: TowerPPG.cpp

COM

CoClass: TowerPPG

Interface: ITowerPPG

Type: TowerPPG Class

Prog ID: TowerPPG.TowerPP

OK Cancel

像以前一样，向导将帮助我们用推荐的名字填充其他框。Interface（接口）框是灰色的，因为一个属性页对象不需要用户接口。接受 Attributes（属性）选项卡

内的默认设置，并选择 Strings（字符串）选项卡。键入 &Caption 作为页的标题，在标有 Doc String（Doc 字符串）的框内键入 Caption property。TowerATL 不会提供帮助文件，所以删除第 3 个编辑框内的文本，并保持空白。

Object Wizard（对象向导）将标题和文档字符串写入项目的 RC 文件，在这里，它们变成控件的字符串资源数据的一部分。标题规定了出现在新属性页的选项卡上的内容。文档字符串被用作选项卡的工具提示文本，它将在鼠标光标移动暂停在选项卡上时，描述这一页的作用，但是这—个字符串并不经常使用，并且不会作为工具提示或者其他的东西出现。这是因为负责创建属性工作表窗口的 OLE 运行 OleCreatePropertyFrame 函数，在 OLE 术语中被称为属性帧，并不支持工具提示。

单击 OK 按钮，关闭对话框，此时 Object Wizard（对象向导）将在项目中加入这些：

- 加入 TowerPPG.cpp 和 TowerPPG.h 文件，以实现新 CTowerPPG 类。
- 在 TowerATL.rc 文件中为标题和工具提示写入字符串资源：

```
IDS_TITLETowerPPG    "Caption"
```

```
IDS_DOCSTRINGTowerPPG    "Caption property"
```

- 为新属性页对象在 TowerATL.idl 文件中追加一个标识符和 coclass 入口。

```
[  
  
    uuid(05D2BAA4-C471-11D1-BEC9-FB1AF66FCC79),  
  
    helpstring("TowerPPG Class")  
  
]  
  
coclass TowerPPG  
  
{  
  
    interface IUnknown;  
  
};
```

- 加入 Towerppg.rgs 文件，为新对象提供注册表脚本。
- 在 TowerATL.cpp 文件中的控件对象映射中为页插入一个入口。

```
BEGIN_OBJECT_MAP(ObjectMap)
```

.

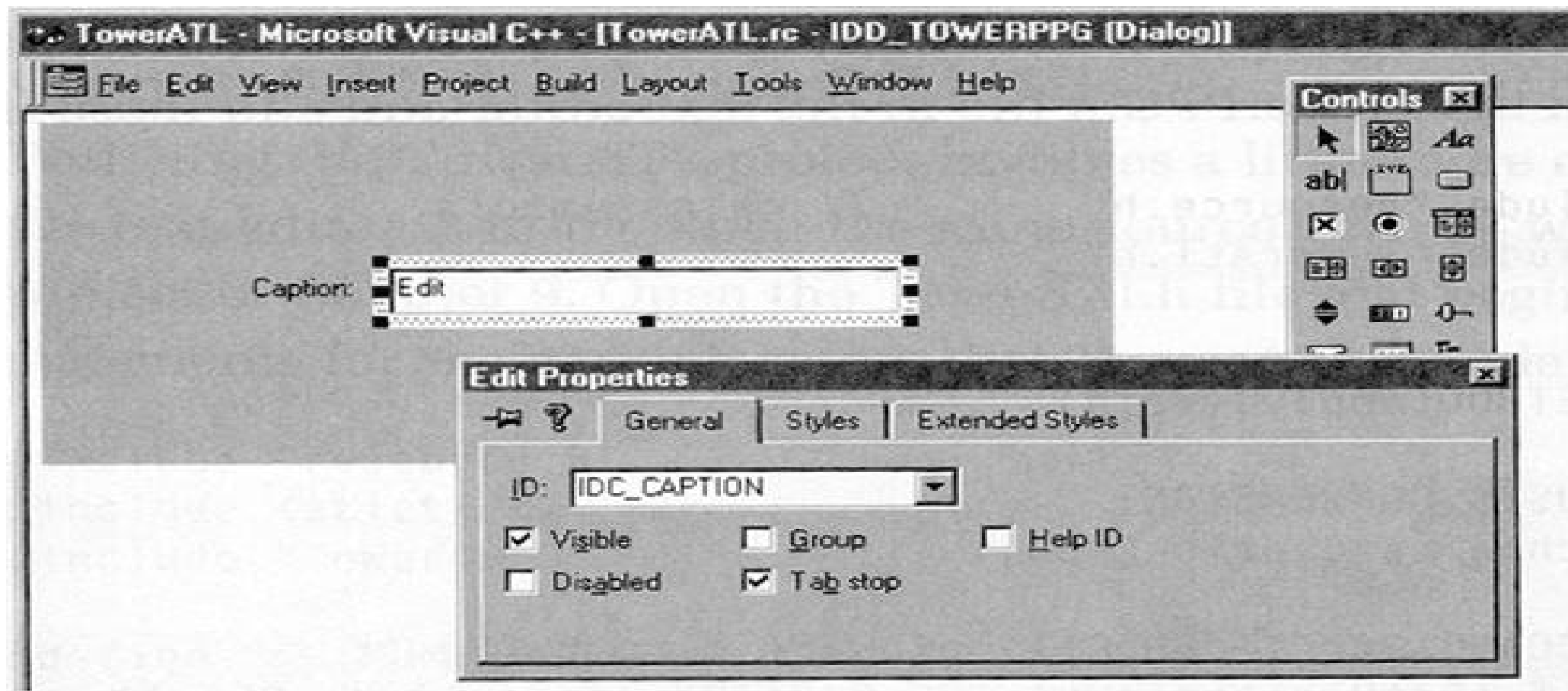
.

.

OBJECT_ENTRY(CLSID_TowerPPG, CTowerPPG)

END_OBJECT_MAP()

当 Object Wizard（对象向导）完成这些操作后，将自动出现对话框编辑器，这表示它已经准备好让你开始设计新属性页。用静态和编辑控件工具编辑对话框，使对话框看起来就像这样：



你自己的属性页的精确布置并不重要，但是，要像图中那样为编辑框指定一个 IDC_CAPTION 的标识符（为激活 Properties 对话框，选择编辑框，并单击 Edit（编辑）菜单上的 Properties（属性））。存盘，并关闭对话框编辑器。

此时 TowerATL 控件已接近完成了。剩下的最重要的工作是在 ATL 产生的骨架源文件中加入代码。这是下一步的事情。

步骤 7：编辑 Towerppg.h 文件

TowerPPG.h 文件包含对应 CTowerPPG 类的代码，这个类用于处理控件的 Caption 属性页对象。我们仅仅需要在类上加入一些指令，以在用户键入新 Caption 字符串时监视页的编辑框和调用 put_Caption 方法。用鼠标右键单击 ClassView 窗格中的 CTowerPPG 类，并选择 Add Windows Message Handler（加入窗口消息处理程序）命令。当 New Windows Message（新窗口消息）对话框出现时，在标有 Class Or Object To Handle（处理的类或对象）的小框内选择 IDC_CAPTION，并在列表中双击 EN_CHANGE。接受 OnChangeCaption 这个名字作为函数名字，并关闭 New Windows Message（新窗口消息）对话框。

当用户在我们前面所加入的属性页的编辑框中键入字符时，将运行 OnChangeCaption 函数，这样产生修正 Caption 属性的意图。这个函数仅仅通过传递一个值 TRUE 到 SetDirty 子函数，使对话框的 Apply（应用）按钮可用。单击属性工作表对话框中的 OK 或者 Apply（应用）按钮将运行 Apply 函数，它需要额外的代码。在文本编辑器中打开 TowerPPG.h 文件，并加入这些行：

```
#include "resource.h"    // main symbols
```

```
#include "TowerAtl.h"
```

.

.

```
STDMETHOD(Apply)(void)
```

```
{
```

```
    USES_CONVERSION;
```

```
    char    szCaption[256];
```

```
        CComQIPtr<ITowerCtl> pTower( m_ppUnk[0] );
```

```
    if (GetDlgItemText( IDC_CAPTION, szCaption, 256 ))
```

```
        pTower->put_Caption( A2BSTR( szCaption ) );
```

```
        m_bDirty = FALSE;
```

```
    return S_OK;
```

```
}
```

```
LRESULT OnChangeCaption(WORD wNotifyCode, WORD wID,
```

```
HWND hWndCtl, BOOL& bHandled)
```

```
{
```

```
SetDirty( TRUE ); // Enable Apply button
```

```
return 0;
```

```
}
```

Apply 函数调用 GetDlgItemText 将新标题字符串从编辑框拷贝到 szCaption 缓冲区中，然后调用控件的 put_caption 方法按照新字符串放置 Caption 属性。put_Caption 方法需要一个 BSTR 参数，所以代码用 ATL 宏 A2BSTR 来将 szCaption 中的 ANSI 字符串转换为 BSTR 类型。正如 Apply 函数所证明的那样，在激活像 A2BSTR 这样的转换宏之前，必须包含 USES_CONVERSION 宏。这样做将消除转换所带来的编译器错误。

这段代码也证明一个 ActiveX 控件如何利用 CComQIPtr 灵巧指针模板类。因为

put_Caption 是 CTowerCtl 的一个子函数，而不是 CTowerPPG 的，所以这个函数必须首先通过 QueryInterface 接收一个指向 ITowerCtl 的指针。将 pTower 作为一个灵巧指针，将确保 ITowerCtl::Release 在 GetDlgItemTest 或者 put_Caption 不大可能出现错误的事件中被调用。

步骤 8：编辑 TowerCtl.h 文件

完成 TowerATL 项目需要用文本编辑器进行较多的编辑工作，为 Tower 加入与我们在第 9 章中所加入的源代码一样的部分。打开 TowerCtl.h 文件，并找到 TowerATL 中彩色显示的以 #define 语句开头的部分：

```
#include "resource.h"                //main symbols
```

```
#include <atlctl.h>
```

```
#include "TowerCP.h"
```

```
#define                NUM_BLOCKS                7
```

续表

#define	EMPTY	NUM_BLOCKS
#define	BLACK	RGB(0, 0, 0)
#define	BLUE	RGB(0, 0, 255)
#define	CYAN	RGB(0, 255, 255)
#define	GREEN	RGB(0, 255, 0)
#define	MAGENTA	RGB(255, 0, 255)
#define	RED	RGB(255, 0, 0)
#define	YELLOW	RGB(255, 255, 0)
#define	WHITE	RGB(255, 255, 255)
#define	GRAY	RGB(128, 128, 128)

然后，在 `CTowerCtl` 类中为我们在原来的 `Tower` 项目中使用的相同的子变量加入声明：

```
class ATL_NO_VTABLE CTowerCtl :
```

```
·
```

```
·
```

```
·
```

```
{
```

```
private:
```

```
    short    nPanel[3][NUM_BLOCKS];           // Panel contents
```

```
    short    nBlockNdx, nFromPanel;           // nPanel index of moved block
```

```
    BOOL     bMoving;                         // Flag is set when dragging
```

```
    COLORREF  color[NUM_BLOCKS];             // Block colors
```

```
    HCURSOR#hArrow, hCrossHairs;             // Cursor handles
```

```
    int#iLeft, iWidth, iHeight;               // Window dimensions
```

```
    short    GetPanel( int i );
```

```
public:
```

```
    HRESULT#FinalConstruct();
```

```
    CTowerCtl()
```

```
{
```

```
}
```

一个像 CTowerCtl 这样的使用数据的对象必须覆盖 CComObject-RootEx::FinalConstruct 子函数，就像这里说明的一样。与 MFC 的有用的 OnInitDialog 函数非常类似，在 ATL 完成设置对象之后，对象被激活之前，将调用 FinalConstruct。正是在这里，而不是在其他的类操作时，控件必须执行它自己的大部分初始化工作。在 Pulse 控件中已经被说明的匹配的 FinalRelease 函数允许控件执行任何必要的清除工作。

更改属性映射，以建立 Caption, Color 和 Font 在控件的属性工作表中出现的顺序：

```
BEGIN_PROP_MAP(CTowerCtl)
```

```
    PROP_PAGE( CLSID_TowerPPG )
```

```
PROP_PAGE( CLSID_StockColorPage )
```

```
PROP_ENTRY( "Font", DISPID_FONT, CLSID_StockFontPage )
```

```
END_PROP_MAP()
```

这个消息映射包含在步骤 4 中加入的控件的 3 个消息处理程序的入口：

```
BEGIN_MSG_MAP(CTowerCtl)
```

```
MESSAGE_HANDLER(WM_MOUSEMOVE, OnMouseMove)
```

```
MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
```

```
MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUp)
```

```
·
```

```
·
```

```
·
```

```
END_MSG_MAP()
```

当用户按下和松开左鼠标按钮，在面板之间拖动彩色块时，OnLButtonDown 和

OnLButtonUp 函数将进行控制。当鼠标移动时，OnMouseMove 进行控制，它的作用仅仅是当用户拖动一个块时，保证光标保持交叉形状。

所有带有 MESSAGE_HANDLER 宏的在消息映射中列出的函数必须具有相同的参数列表：

```
LRESULT MessageHandler(UINT uMsg, WPARAM wParam,
```

```
    LPARAM lParam, BOOL &bHandled);
```

最前面的 3 个参数是标准的消息参数。例如，OnLButtonDown 处理程序，接收 uMsg 参数中的值 WM_LBUTTONDOWN，wParam 中的当前键状态和 lParam 中的鼠标光标的坐标。在调用处理程序函数之前，MESSAGE_HANDLER 宏将 bHandled 标志设定为 TRUE。一个不能完全满足消息要求的处理程序在返回之前要清除这个标志：

```
*bHandled = FALSE
```

否则，MESSAGE_HANDLER 宏产生的代码将立即返回，向操作系统指出消息已经被完全处理了。

一个 ATL 消息映射与在 MFC 中的映射具有一样的形式。ATL 通过向 WM_COMMAND 和 WM_NOTIFY 消息提供 COMMAND_HANDLER 和 NOTIFY_HANDLER 宏，而进一步与 MFC 匹配。为了用一个函数处理一组不同

的消息，用一个 RANGE 宏替代：MESSAGE_RANGE_HANDLER, COMMAND_RANGE_HANDLER 或者 NOTIFY_RANGE_HANDLER。与 MFC 中的 ON_COMMAND_RANGE 和 ON_NOTIFY_RANGE 类似，这些宏为所有在 规定值范围内的消息规定一个处理程序函数。Visual C++ 将函数 OnDraw, OnMouseMove, OnMouseDown 和 OnButtonUp 定义为 CTowerCtl 类声明中的内联函数。为了在同一个地方实现代码，并帮助与原来的 Tower 源文件进行比较，将这 4 个函数移动到 TowerCtl.cpp 文件中，我们将在下面讨论它。当然，剪切和粘贴是完全可选的；如果你按照本文创立这个项目，并喜欢将这些函数放在头文件中，只需要按照下一节所讲的内容编辑它们的代码。

在关闭 TowerCtl.h 文件之前，有必要进行最终的检查。在类声明的结尾，一些版本的 Visual C++ 写下了一个名为 m_spFont 的子变量。这个名字必须换为 m_pFont，以防止栈的 Font 页的 ATL 实现产生错误：

```
CComPtr<IFontDisp>m_pFont
```

步骤 9：编辑 TowerCtl.cpp 文件

TowerCtl.cpp 的源代码与我们为原来的 Tower 项目的相同文件加入的代码非常相似。事实上，它们相似的程度，使得第 9 章中的代码仍然适用于这个控件的新版本。因为这个文件需要扩充，列表 10-4 列出了完整的编辑后的版本。

控件的 MFC 和 ATL 版本之间的一个重要区别是 TowerCtl.cpp 中如何进行初始化。Tower 控件使用类构造器和 PreCreateWindow 虚拟函数来初始化它的数据，而现在 TowerATL 使用 ATL 的 FinalConstruct 函数。

按排初始化工作仍然是相同的；只是位置发生了变化。

列表 10-4 TowerCtl.cpp 文件

```
// TowerCtl.cpp : Implementation of CTowerCtl
```

```
#include "stdafx.h"
```

```
#include "TowerALT.h"
```

```
#include "TowerCtl.h"
```

```
#include "TowerBox.h"      // About box dialog
```



```
// CTowerCtl
```

```
HRESULT CTowerCtl::FinalConstruct()
```

```
{
```

```
    color[0]          =BLACK;          // Initialize block colors
```

```
    color[1]          =BLUE;
```

```
    color[2]          =CYAN;
```

```
    color[3]          =GREEN;
```

```
    color[4]          =MAGENTA;
```

```
    color[5]          =RED;
```

```
    color[6]          =YELLOW;
```

```
    m_clrBackColor = GRAY;          //Default background color,
```

```
m_clrForeColor = WHITE;          // foreground color,

m_bstrCaption = "TowerATL";      // and caption

m_bAutoSize   = FALSE;          //Control can be resized

m_sizeExtent.cx      =7000;      //Init HIMETRIC size for

m_sizeExtent.cy      =2500;      //    4 x 1.5 width/height ratio

Reset();                  //Initialize panels


// Cursors for normal (arrow) and dragging (crosshairs)

hArrow      = LoadCursor( NULL, IDC_ARROW);

hCrossHairs= LoadCursor( NULL,IDC_CROSS);

return S_OK;
```

```
}
```

```
HRESULT CTowerCtl::OnDraw( ATL_DRAWINFO& di)
{
    RECT& rc=*(RECT*)di.prcBounds;

    RECT rect;

    TEXTMETRIC tm;

    HPEN hPen,hPenOld;

    HBRUSH hBrush, hBrushOld;

    HFONT hFont, hFontOld;

    int l,j,k,yCaption;

    USES_CONVERSION;
```

```
//Paint control background

hBrush    = CreateSolidBrush( m_clrBackColor);

hBrushOld=(HBRUSH) SelectObject(di.hdcDraw,hBrush);

FillRect(di.hdcDraw,&rc,hBrush);

// Set caption color and font

SetBkMode(di.hdcDraw,TRANSPARENT);

SetTextColor(di.hdcDraw,m_clrForeColor);

if (m_pFont)

{

    CComQIPtr<Ifont> pFont(m_pFont);

    pFont->get_hFont(&hFont);

    hFontOld=(HFONT) SelectObject( di.hdcDraw,hFont);
```

```
}  
  
else  
  
{  
  
    hFont    = (HFONT) GetStockObject( SYSTEM_FONT);  
  
    hFontOld=NULL;  
  
}
```

```
// Display caption
```

```
CopyRect( &rect, &rc);
```

```
DrawText( di.hdcDraw,OLE2A( m_bstrCaption ), -1,
```

```
&rect, DT_CENTER | DT_TOP );
```

```
// Compute height of Caption area

GetTextMetrics( di.hdcDraw,&tm);

yCaption = tm.tmHeight + tm.tmExternalLeading;


// Compute width and height of a panel

iLeft      = rc.left;

iWidth     = (rc.right - rc.left)1/3;

iHeight    = rc.bottom - rc.top - yCaption;


// Draw column dividers

hPen       = CreatePen( PS_SOLID, 1, m_clrForeColor );
```

```
hPenOld    = (HPEN) SelectObject( di.hdcDraw, &hPen);

MoveToEx(di.hdcDraw,rc.left+iWidth,    rc.top+yCaption, 0 );

LineTo(    di.hdcDraw,rc.left+iWidth,    rc.bottom );

MoveToEx(di.hdcDraw, rc.left+iWidth*2,  rc.top+yCaption, 0 );

LineTo(    di.hdcDraw,rc.left+iWidth*2,  rc.bottom);

// Outer loop: for each panel

for (i=0 i < 3 i++)

{

    rect.top      = rc.top + yCaption;

    rect.bottom = rect.top + iHeight/NUM_BLOCKS;


    // Inner loop: for each colored block in panel...
```

```
for (j=0; j < NUM_BLOCKS; j++)  
  
{  
  
    if (nPanel [I] [j] != EMPTY)  
  
    {  
  
        // Determine left and right edges of block  
  
        k= NUM_BLOCKS - 1 - nPanel [I] [j] ;  
  
        rect.left = rc.left + iWidth*I +  
  
            (iWidth*K) / (2*NUM_BLOCKS) + 1;  
  
        rect.right = rect.left +  
  
            iWidth*(nPanel[I][j]+1)/NUM_BLOCKS - 1;  
  
  
        // Fill rectangle with block's color
```



```
hBrush = CreateSolidBrush( color[nPanel[I][j]] );
```

```
SelectObject( di.hdcDRAW, &rect,hBrush );
```

```
}
```

```
rect.top      = rect.bottom;
```

```
rect.bottom += iHeight/NUM_BLOCKS;
```

```
}
```

```
}
```

```
SelectObject(di.hdcDraw, &hPenOld );
```

```
SelectObject( di.hdcDraw, &hBrushOld );
```

```
if (hFontOld)
```

```
SelectObject( di.hdcDraw, &hFontOld );
```

```
DeleteObject(hPen);
```

```
DeleteObject(hBrush);
```

```
return S_OK;
```

```
}
```

```
STDMETHODIMP CTowerCtl::get_nCurrentBlock( short *pVal_
```

```
{
```

```
*pVal = nPanel[nFromPanel][nBlockNdx];
```

```
return S_OK;
```

```
}
```

```
STDMETHODIMP CTowerCtl::Reset()
```

```
{

    int I;

    for (I=0; I < NUM_BLOCKS; I++)    // Initialize panel array

    {

        nPanel [0][i] = I;                // Panel 0 = 0,1,2,3,4,5,6

        nPanel[1][i] = EMPTY;            // Panel 1= 7,7,7,7,7,7,7

        nPanel[2][i] = EMPTY;            // Panel 2= 7,7,7,7,7,7,7

    }

    nBlockNdx    =0;                    // Ndx of block being moved

    nFrom Panel =0;
```

```
FireViewChange();
```

```
return S_OK
```

```
}
```

```
////////////////////////////////////////////////////////////////
```

```
// CTowerCtrl message handlers
```

```
LRESULT Ctowerctl::OnLButtonDown( UINT uMsg, WPARAM wParam,
```

```
                                LPARAM lParam, BOOL& bHandled)
```

```
{
```

```
    short i = 0;
```

```
    int    x = LOWORD( lParam);
```

```
nFromPanel = GetPanel( x );

while (nPanel[nFromPanel][I] == EMPTY && i < NUM_BLOCKS)

    i++;                                // i =ndx of smallest block


if (I < NUM_BLOCKS)                    // Does panel have a block?
{

    bMoving      = TRUE;                // If so, block is moving

    nBlockNdx    = I;                  // Save ndx of the block

    Fire+FromPanel( nFromPanel);      // Tell container panel #


    // change cursor to crosshairs while dragging

    SetCursor( hCrossHairs);
```

```
}

return 0;

}

LRESULT CTowerCtl::OnLButtonUp(UINT uMsg, WPARAM wParam,

                                LPARAM lParam, BOOL& bHandled)

{

    short i = 0, nToPanel;

    int    x= LOWORD( lParam );

    nToPanel = GetPanel( x);    // Panel where block is dropped


    if (bMoving && nToPanel != nFormPanel)

    {
```

```

while (nPanel[nToPanel][i]==EMPTY && I < NUM_BLOCKS-1)

    i ++;                // I - ndx of panel's smallest block


// Is dragged block smaller than smallest block in panel?

If (nPanel [nFromPanel][nBlockNdx]< nPanel [nToPanel][I])

{

    if (nPanel [nToPanel][I] != EMPTY)

        --I;

    nPanel[nToPanel][I] = nPanel[nFromPanel][nBlockNdx];

    nPanel[nFromPanel][nBlockNdx] = EMPTY;

    Fire_ToPanel( nToPanel );           // Tell container

    if (i== 0    && nToPanel == 2)      // If we've filled

```

```
        {                                // the third panel,

            Fire_Winner ();                // fire Winner event

            Reset ();                      // and reset game

        }

        FireViewChange();

    }

    else                                // If invalid drop,

        Fire_Error ();                  // tell container

}

// Restore original arrow cursor

SetCursor( hArrow);

bMoving = FALSE;                       // Not moving now
```



```
    return 0;
```

```
}
```

```
short CTowerCtrl :: GetPanel ( int x )
```

```
{
```

```
    short I = 0;
```

```
    x -= iLeft;                //Convert x to window coords
```

```
    if (x > iWidth)            //Hit test:
```

```
        i = 1;                //I = 0 for first panel
```

```
    if (x > iWidth*2)          //    =1 for second panel
```

```
        i = 2;                //    = 2 for third panel
```

```
    return i;                  // Return panel number
```

```
}
```

```
LRESULT CTowerCtl::OnMouseMove( UNIT uMsg, WPARAM wParam,
                                LPARAM lParam, BOOL& bHandled)
{
    if (bMoving)                // While dragging, keep
        SetCursor( hCrossHairs );    // crosshairs cursor

    return 0;
}
```

步骤 10：加入 About 框

除了属性页外，ATL Object Wizard（对象向导）也可将一个对话资源作为一个独立的对象加入到控件项目中。这一节将通过为 TowerATL 项目创建一个 About 框，从而解释如何在 ActiveX 控件中合并对话框对象。许多开发者不愿意在 ActiveX 控件中加入 About 框，这是可以理解的，因为即使一个最简单的对话框，也将在 OCX 文件中增加至少 2KB 资源数据。但是，一个 About 框对于你自己的控件可能需要的任何类型的对话框而言，是一个非常好的说明。更重要的是，项

目设计迫使我们尽可能地接近原来的 Tower 控件，原来的 Tower 控件中包括由 MFC ControlWizard 支持的自己的 About 框。

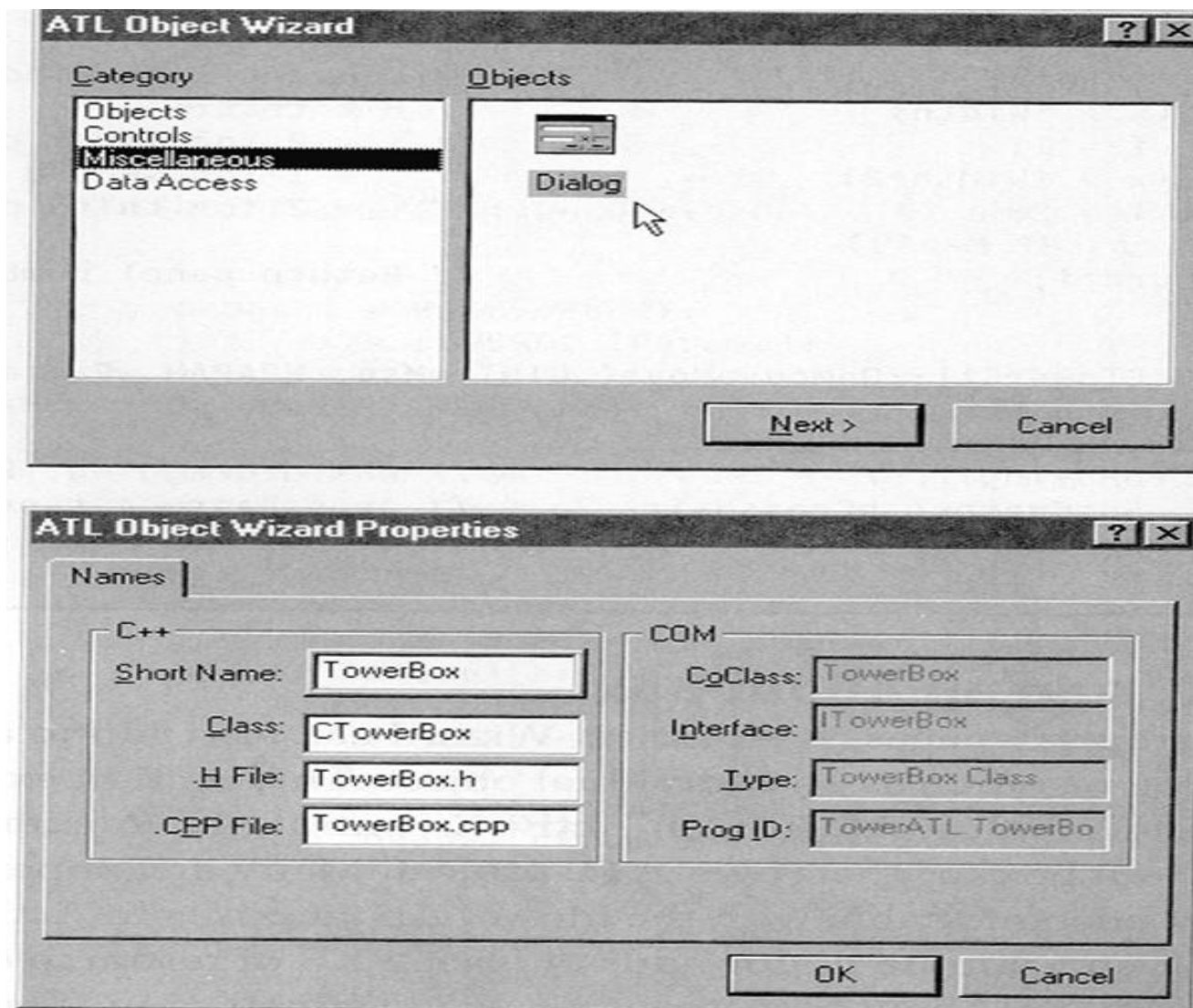


图 10-15 为 ATL ActiveX 控件项目加入对话框资源

这一节的内容是自我包容的，与练习的其他部分保持独立性。它将在同一个位置说明包含一个对话框资源的所有必须的步骤，所以如果你希望不影响项目的其余部分，你可用忽略这一节的内容。为了加入 TowerATL 的 About 框，再次允许 ATL Object Wizard（对象向导），这一次，在左框中选择 Miscellaneous，并双击 Dialog（对话框）图标。按照图 10-15 命名对象 TowerBox，并单击 OK，关闭对话框。

Object Wizard(对象向导)为类 CTowerBox 产生源代码，它被包含在 TowerBox.cpp 和 TowerBox.h 文件中。如果你想使用在另一个项目中的一个 About 框资源，只需要将对话框脚本合并到 TowerATL.rc 中，并在 Resource.h 文件中加入必要的 #define 语句。如果不是这样，在对话框编辑器中设计 About 框，在你关闭 Object Wizard（对象向导）时，对话框编辑器将自动启动。下面是对话框可能的样子：

About TowerATL



TowerATL ActiveX control

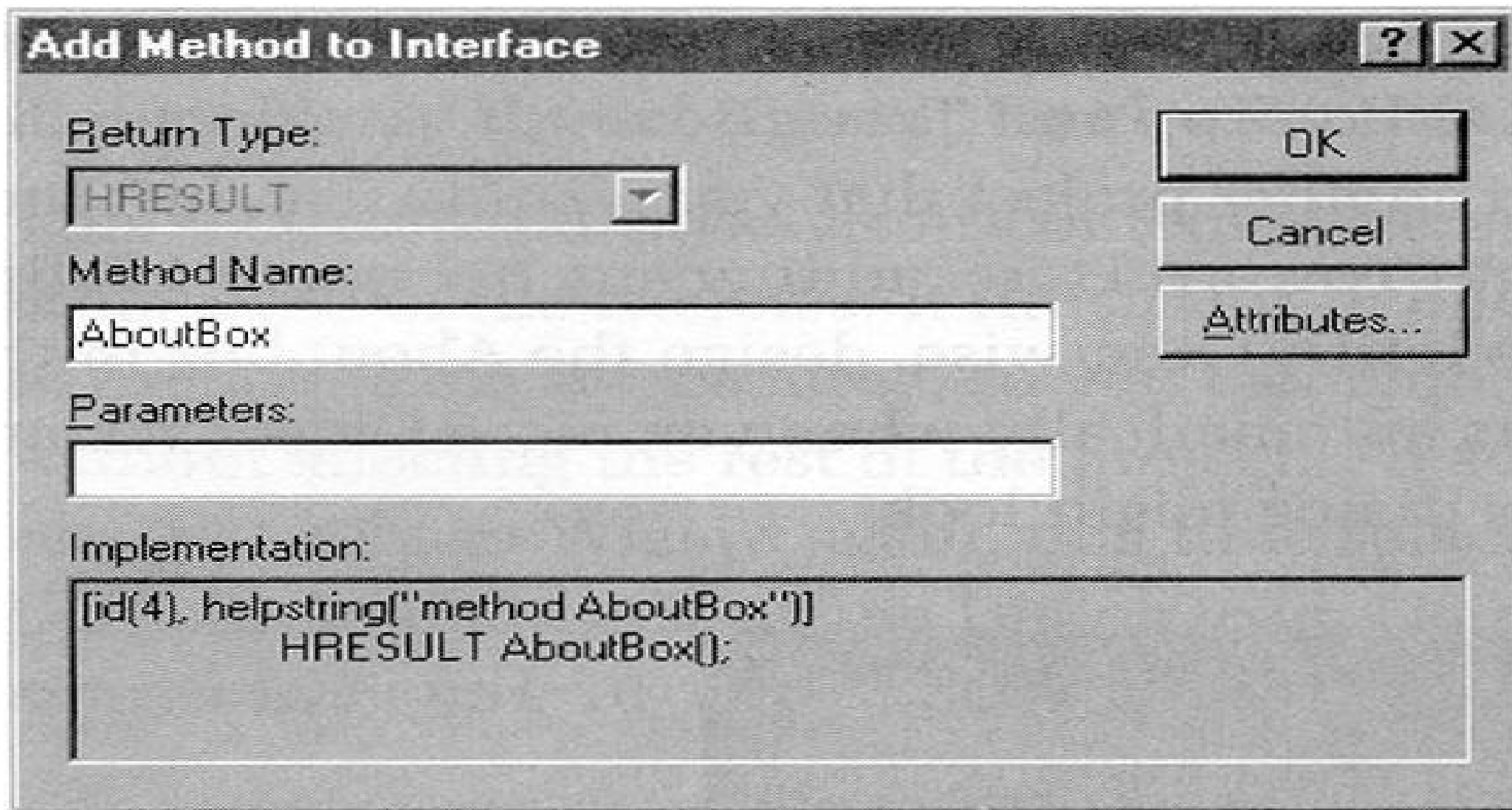
"Visual C++ Programmer's Guide"

Current thread:

OK

这个对话框借用了项目的位图，所以这种装饰图并不代表资源控件的奢侈浪费。首先将 TowerCtl.bmp 文件从 Code\Chapter.10\TowerATL 文件夹内拷贝到你的项目中，覆盖有 ATL Object Wizard（对象向导）提供的相同名字的 BMP 文件。用 Picture（图像）工具将图像放在对话框编辑器内，然后弹出 Properties（属性）对话框，并在 Type（类型）框内选择 Bitmap（位图），在 Image（图像）框内选择标识符。最后一行的编辑框是可选的，仅仅为了方便显示控件正在其上运行的线程的标识符。这将使我们可用在独立公寓中运行的时候，确认一个像 TowerATL 这样的公寓线程控件的动作。给编辑框分配一个 IDC_THREAD_ID 标识符，并在 Properties（属性）对话框内设置 Read Only（只读）复选框。

在完成设计 About 框后，将对话框资源存盘，然后在 ClassView 窗格中用鼠标右键单击 ITowerCtl，并选择 Add Method（加入方法）命令。如下所示键入 AboutBox 作为新方法的名字。与 Reset 方法类似，AboutBox 没有参数。



在文本编辑器中打开 TowerBox.h 文件，并在 OnInitDialog 函数中加入下面灰色显示的指令。在 About 对话框出现之前，代码将从系统中提取线程标识符，并将

其写入到 IDC_THREAD_ID 编辑框中：

```
LRESULT OnInitDialog(UINT uMsg, WPARAM wParam,
                    LPARAM lParam, BOOL& bHandled)
{
    SetDlgItemInt( IDC_THREAD_ID,
                  (UINT) ::GetCurrentThreadId(), FALSE );

    return 1;  // Let the system set the focus
}
```

然后重新打开 TowerCtl.cpp 文件，并在新 AboutBox 函数中加入这些行：

```
STDMETHODIMP CTowerCtl::AboutBox()
{
    CTowerBox dlgAbout;
```

```
dlgAbout.DoModal();
```

```
return S_OK;
```

```
}
```

当一个包容器应用程序调用控件的 `AboutBox` 方法时，这个函数将创建一个 `CTowerBox` 对象，并激活这个对话框。下一节中的 `Game2` 程序证明了一个应用程序如何通知 `TowerATL` 显示它的 `About` 框。

步骤 11：建立和测试 `TowerATL` `ActiveX` 控件

选择 `Release MinDependency`（或 `MinDep`）配置，并建立 `TowerATL` 控件，在 `Visual C++` 成功编译、链接和注册这个控件之后，你可以在任何 `ActiveX` 软件包容器应用程序中测试完成的产品。图 10-16 显示了在 `Test Container`（测试包容器）实用程序中带有属性工作表的 `TowerATL` 的样子。在将 `TowerATL` 加入到 `Test Container`（测试包容器）窗口之后，单击 `Properties`（属性）工具，以显示控件的属性工作表。

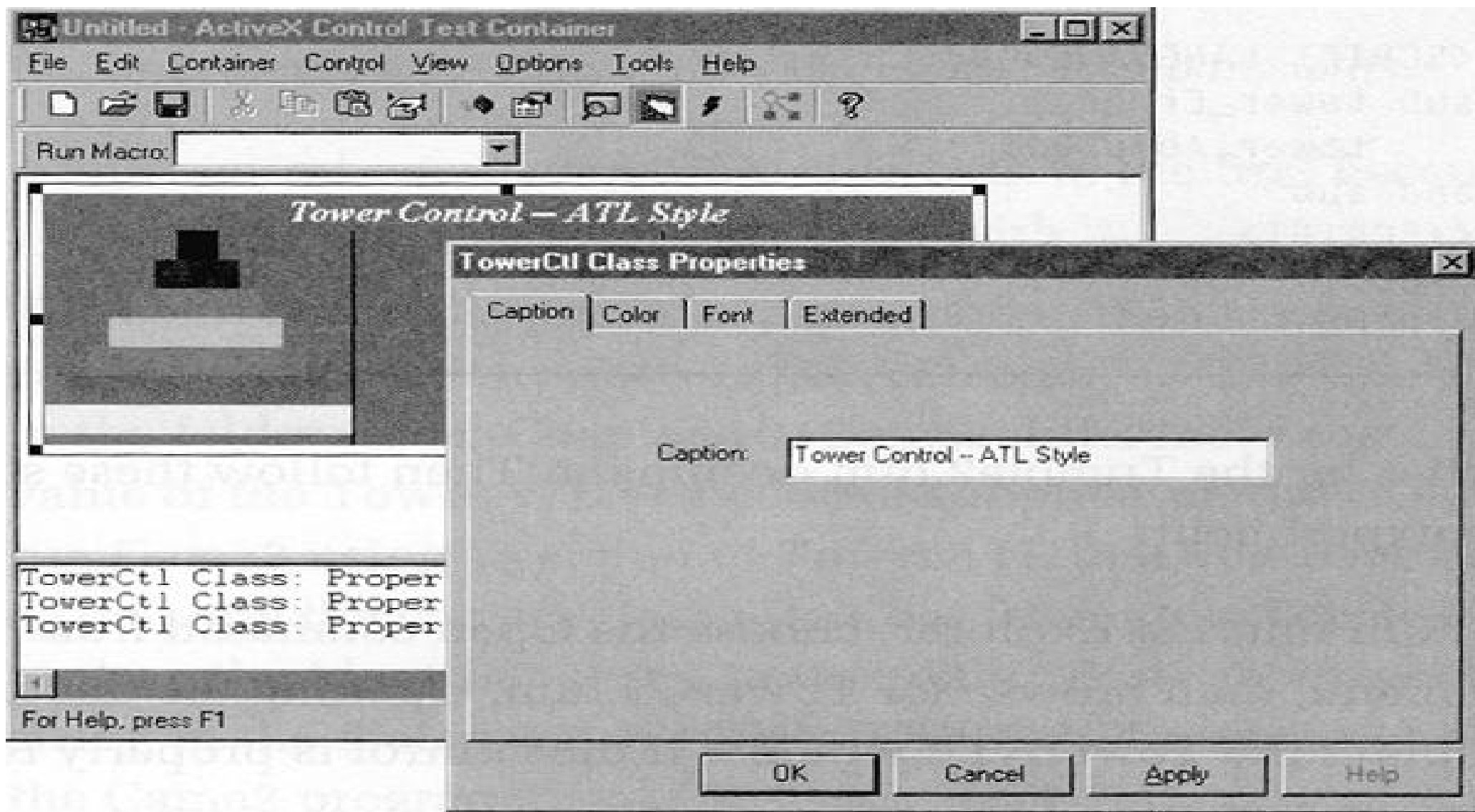


图 10-16 在 Test Container (测试包容器) 内显示的 TowerATL 属性工作表

为了说明,可以在 Test Container 中加载另一个 TowerATL 控件的实例,并用 Invoke Methods (激活方法) 工具激活每一个实例的 About 框。你将会发现两个实例中显示在 About 框中的线程标识符是相同的,这证明 Test Container (测试包容器) 在一个线程上创建所有的控件实例。另一方面,Internet Explorer 在一个独立的线程上运行每一个窗口,我们可以用显示在列表 10-5 中的简单的 TowerCtl.htm 文档证明这一点。在配套光盘的 Code\Chapter.10\TowerAtl 文件夹下找到这个文档。如果你创建了你自己的版本的 TowerATL,首先用控件的类标识符放置这个文档的 classid 入口,从项目的 IDL 文件中拷贝字符串,这就和我们前面对 Tumble2.htm 文档的操作一样。然后按照这些步骤进行试验:

列表 10-5 TowerCtl.htm 文件

```
<HTML>

<HEAD>

<TITLE>ATL 3.0 test page for object TowerCtl</TITLE>

</HEAD>

<BODY>
```

```
<OBJECT
```

```
    classid="clsid:3B365FBA-C3AE-11D1-BEC9-E0F4E352507A"
```

```
    id=tower
```

```
>
```

```
</OBJECT>
```

```
<SCRIPT LANGUAGE="VBSCRIPT">
```

```
sub      tower_Error
```

```
    .tower AboutBox()
```

```
end sub
```

```
</SCRIPT>
```

</BODY>

</HTML>

- 运行 Internet Explorer, 从 File (文件) 菜单中选择 Open (打开) 命令, 然后浏览, 并找到 TowerCtl.htm。如果控件已经正确的注册在你的系统中, 打开这个文档, 将显示 TowerATL 窗口。
- 从同样的菜单中选择 New (新建), 打开一个显示控件的复制窗口。
- 在这两个窗口中的每一个窗口中, 拖动一个彩色块到一个小一些的块, 这将启动控件的 Error 事件。在事件启动的任何时候, 文档的脚本将激活控件的 About 框。

比较显示在每一个 About 框中的线程标识符 (图 10-17), 可以发现 Internet Explorer 在独立的 STA 公寓中运行每一个控件实例。因为 TowerATL 采用公寓线程模型, 所以并不需要在任两个实例之间进行线程间调度, 这对于像 Internet Explorer 这样的 STA 客户是理想的。但是我们为 TowerATL 选择了单一非线程模型, 因为 COM 将被强迫调度所有与它有关的交互作用, 所以控件的第二个实例将比第一个实例运行得慢很多。

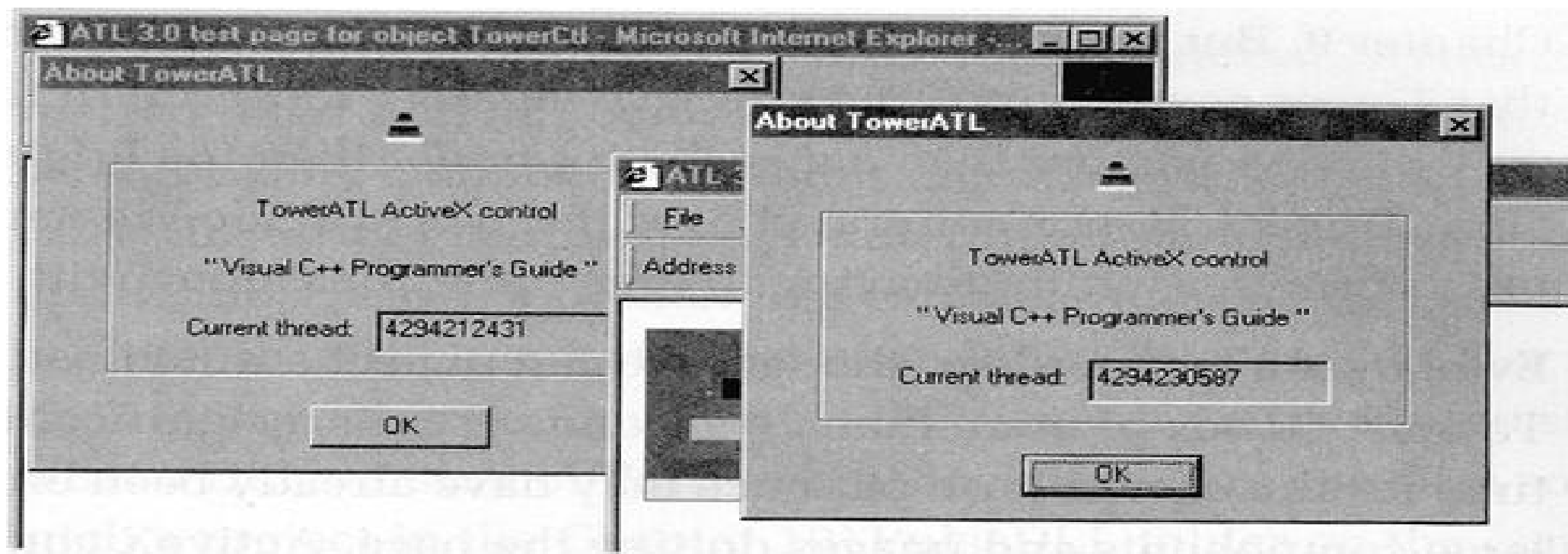


图 10-17 在分离的 STA 公寓中运行 TowerATL 控件

配套光盘提供了一个名为 Game2 的程序，它是专门为显示新 TowerATL 控件而设计的。Game2.rc 文件位于 Code\Chapter.10\Game2 文件夹中，引用了 CD 上提供的 TowerATL.ocx 控件的类标识符的值。如果你想要用你自己创立的 TowerATL 版本运行 Game2，仍需要从你的 TowerCtl.idl 文件中拷贝正确的类标识符，来替换 Game2.rc 中的标识符字符串。在重建 Game2 程序之前，在对话框脚本的 IDC_TOWERATL 语句中粘贴新类标识符字符串：

WS_TABSTOP,5,20,225,100

与第 9 章的 Tower 项目相比，TowerATL 需要在 MFC 为我们屏蔽的细节上更加努力 and 更加注意。所以从这一点说，值得总结我们在用 ATL 代替 MFC 控件时的得失。损失很容易发现：更多的工作量。然而，作为补偿，我们得到了一个小得多的可执行映像，尽管最开始你可能不这么认为。新 TowerATL.ocx 有 70KB，这是我们在第 9 章中创建的 Tower.ocx 文件大小的 3 倍。但是如果你考虑到 Tower.ocx 需要的实时库，控件的 ATL 版本的实际需要的存储器比原来的 MFC 版本小 1 兆多。TowerATL.ocx 可以在 World Wide Web 上传送，可以在一个用户的浏览器中运行，而不需要与它一起拖动其他文件。

可是，即使通过一个较快的调制解调器，我们可预料 TowerATL.ocx 的下载时间将超过 20s。这对于用户来说是一个明显的时间投资，而用户的耐心可能已经被

其他巨大的组件和点缀页面的图像所消耗了。Internet 上的 ActiveX 控件面对唯一的障碍是用户事先通常对要求并不明确，并且可能不愿意在这方面多花时间。可以理解的是，急躁经常可以战胜好奇心。为 Internet 编写一个成功的 ActiveX 控件的诀窍是使它在 Web 页上看起来足够小，并在好奇心被击败之前准备运行。

现在 ATL 提供了编写小 ActiveX 控件，缩短直接 COM 编程的最佳途径。库代表了大小和劳动量之间，以及可能和实际之间的折衷。因为 ATL 用优秀的结果回报了合理的开发努力，所以这个折衷是成功的。

对比组件模型

在结束这本处理 ActiveX 控件的书之前，让我们后退一步，从一个安全的距离来考虑这个主题。当着手一个组件项目时，一个开发者不能仅仅衡量像 ATL 这样的支持技术的正面和负面因素，而必须首先考虑是否使用 ActiveX 控件。像其他软件技术一样，ActiveX 控件对于某些情况是合适的，但并不适用于其他情况。如果组件是作为一个嵌入应用程序单独计划的，并且永远不会在一个 Web 页上应用，那么 ActiveX 需要较高的成本。通过在 COM 和 ActiveX 服务库中拖动，即使是一个向 Pulse 这样的简单控件将在存储器中安装许多 KB 的代码，而且可能显著地降低调用应用程序的加载进程速度。

有另一种解决方案。将 Pulse 设计为一个一般的动态链接库，而不是一个 ActiveX

控件，这将以一个线程安全的形式保留软件和可用的组件，而不用花费 ActiveX 和 COM 那样的代价。客户应用程序向库在每一个要求的时间间隔结束时调用的回调函数返回一个指针，而不是建立一个处理程序函数来接收事件。既然在新外壳下，一个事件处理程序仅仅是一个回调，那么结果是一样的。如果你想了解一个 Pulse 的 DLL 版本的模样，你可用在配套光盘的 Chapter.10\PulseDLL 文件夹中找到这样的项目的源文件。Pulse 的动态链接库版本的最有趣的特点是它的大小：3KB，而 ActiveX 控件版本需要 37KB。而且 PulseDLL 不需要 COM。

这个简短的对比的目的仅仅是想说明这个问题的各个方面，提醒大家注意至少在某些情况下，低级动态链接库仍然在世界组件软件中占据一席之地。这并不是建议所有不用于 Internet 的组件必须用动态链接库编写。事实上远非如此，ActiveX 超越动态链接库而提供了显著的方便性，特别是在处理众所周知的“版本”问题时更是如此。动态链接库的吸引力由于它的第二版的发布而降低了，而这个问题对 ActiveX 控件的影响并没有那么大。例如，假设我们决定，如果它调用 timeSetEvent 和 timeKillEvent 应用程序编程接口函数，而不依赖于系统的 Sleep 计时器的行为，Pulse 将是一个更精确的计时器。放置后的版本将向新客户提供新对象和能力，而像 Hour2 这样的旧客户应用程序将像以前一样嵌入到新 Pulse 中，不需要改变任何东西。甚至用新版本覆盖旧 Pulse.ocx 文件，也不会影响客户应用程序，而不论它们是否利用扩展特性。在利用一个动态链接库的高版本时，这种稳定性是非常难以达到的。

第五部分 高级主题

第 11 章 调试器

完成设计和编码后，即开始调试程序，这是软件开发的第三步。一个 3000 行的程序，其编译可达到没有一个警告，然而在运行时却可能崩溃，更糟糕的是，运行只是偶尔崩溃。当程序不能顺利运行，而又不知道问题的症结所在时，就该使用调试器来监视此程序的运行了。

Visual C++ 调试器是整个产品中最具特色的一部分，省时省力，简单易用，它可以帮助找到在 Windows 软件开发过程中可能遇到的几乎一切故障。但调试程序就像科学一样，它需要技术，需要有清醒的头脑和良好的洞察力，调试器就像显微镜一样，因为它能扩大你的视野，但你必须知道所查找的症结。

动态链接库，包括 Active X 控件，对 Visual C++ 调试器来说，并不是特殊的情况。调试器轻松跨越两个项目之间的边界时，即意味着可以开始调试项目中的程序，然后，当此程序调用动态链接库中的外部函数时，继续进行调试，即使该库和它的源文件存在于另一项目或子项目中，反之亦然。在动态链接库的项目中，可以启动调试会话。在这种情况下，调试器自动运行所调用的应用程序，并且，当执行流执行了某一库函数时，就会将控制权返回给用户。

调试器能够处理多线程和 ActiveX 应用程序，并且有能力在调试程序的计算机上运行。在以后的章节中，我们将会看到这些特殊的情况。首先，让我们先熟悉一下调试器。

调试与发行

Visual C++ 中的程序能产生两种类型的执行代码，称为调试与发行版本，或称之为“目标”版本。调试版本是软件开发和检测修改程序的部分，发行版本是最终的结果，将发行给客户。调试版本较之发行版本要大，运行起来要慢一些。编译器在目标文件中填满了符号信息，这些符号信息记录了编译器知道的函数名、程序中的变量名和标识的内存地址。通过读取源文件和包含在执行文件中的符号信息，调试器能将源代码中的每条流线同相应的可执行映像中的二进制指令联系起来。调试器运行可执行文件，但必须使用源代码来显示程序的进程。

发行版本含有的仅仅是编译器优化的可执行指令，没有符号信息。如果你需要，可以在调试器中执行一个发行版本。调试器将会提醒你文件没有符号数据。同时，可以执行程序中的调试版本而无须调试器。这具有实际效果，因为 Visual C++ 以即时（just-in-time）调试而闻名，在以后的章节中将证明这一点。当运行没有调试器的程序的调试版本时，Windows 加载程序将会忽略文件中的一些外加的符号信息，从而允许程序正常运行。如果程序产生了错误，而且系统的异常处理导致控制权返回 Visual C++，则继续执行调试器。当程序停止时，调试器显示产生

错误的指令，并且显示程序停止当前的数据值。对查找在程序检测期间跟踪错误来说，以及寻找那些看上去似乎是随机的，但却是程序运行中的祸根的错误来说，这种高级特征尤其有用。

顺便说一下，如果想保留对你来说比较重要的内容，处理源代码时，就应该处理程序中的调试版本。含有符号信息的程序文件要比工程逆转容易得多，原因是该文件含有程序中的所有变量名和函数名。它取代了匿名的反汇编语句，例如：

```
004017ae      push      ebp
```

```
004017af      mov       ebp,esp
```

调试器包括从符号数据中读取的函数名：

```
Myclass::InitInstance:
```

```
004017ae      push      ebp
```

```
004017af      mov       ebp,esp
```

虽然源代码现在没有什么用处了，但是，现在任何人都可以识别名为 `InitInstance` 的函数序码。

调试器的使用

当你调试一个有小毛病的程序时，这种程序有时也称之为被调试程序（debuggee），首先调试器开始运行，接着开始执行你想要调试的程序，当所运行的程序到达所选定的指令或者改变某一特定的变量时，调试器允许你重新获得控制权。当此程序被挂起时，这将会给你一个检查当前数据值的机会，也能确保控制流沿着一条所期望的路径进行处理。

调试器能够提供大量的信息，这使之看上去更为复杂。如果你是编程新手，不要被调试器所吓倒，很快，你就会了解每个程序员所学得的知识。一个典型的调试操作由以下几步组成：首先，确定出现问题的那一段程序，然后给这段程序的第一个指令加上标记，启动调试器，运行程序，直到控制到达你事先在带有疑问的程序段所设置的标记。当调试器停止程序的执行时，可以通过单步操作指令，以核实每步的影响效果。

那么，调试器怎样知道什么时候中断程序呢？实际上，并不是调试器中断程序，而是程序自己自行中断，当程序运行到你先前设置的标志时便进行中断，这种标记为断点。

断点

在 Windows 中，调试器与所运行的程序之间的关系是独一无二的。不可能再找到两个程序在运行时联系得如此紧密。调试器和程序不能够同时运行在一种其他常规应用程序能同时运行的多任务环境下。当程序运行时，调试器停止工作，当执行程序触发断点时，它会重新获得控制权。

调试器允许你设置两种不同类型的断点，一种断点是基于代码所在的位置，另一种是基于程序数据。位置断点是一种与源代码相联系的特殊指令的标志，类似于文本编辑器中的书签。在所有想彻底弄清楚错误怎样产生的含有可疑代码的程序段中，你设置一个位置断点。当执行程序试图运行所标记的指令时，它就会停止或者“中断”(在接下来的章节中我们将会弄清楚这是怎样进行的)。

数据断点取决于数据，而不是代码。当你怀疑某一变量在程序中的某一地方被错误地改变但又不知道在什么位置时，就可以使用数据断点。数据断点可以告诉调试器，当变量改变或变为某一不确定的值时，就中断执行。例如，当指针重新赋值或变量 x 的值超过了 500 时。

当位置或数据断点被触发时，控制权返回到调试器。调试器更改显示目前变量值的窗口，以及发生中断的源代码所在的程序段。现在，可以单步执行代码，每次一条指令，以便弄清楚变量是怎样变化的，以及程序的执行状况。

断点返回控制权给调试器的方式

为了阐述断点是怎样工作的，本节讲述调试器设置断点来中断程序和断点触发时重新获得控制权的步骤。讨论以 Intel 和兼容处理器为主，但在其他处理器中的过程与这里所概述的过程类似。在接下来的讨论中，“程序”仅指你的应用程序，不会是调试器。这可以避免使用“被调试的程序”这样绕口的短语。并且，“Debuggee（被调试的程序）”听起来太像 debugger（调试器）。

当设置一个位置断点，或者单步操作从一个 C/C++ 指令到另一个指令时，在执行程序代码块中断的地方，调试器会覆写一个简单的字节。它存储字节的原始值，然后在它所在的位置写上 0xCC。处理器将 0xCC 解释成 INT 3 指令，它将会提醒处理器去执行相应的中断 3 系统处理程序，它与 Intel 调用的中断 3 这个断点不会重合。

写完 INT 3 指令后，调试器通过调用 WaitForDebugEvent API 函数进入睡眠状态，系统将控制权返回给该程序，直到它运行到 INT 3 指令后才会恢复正常。在执行这个指令时，处理器将 CS 和 EIP 寄存器的当前值写到堆栈上，并调用系统的中断 3 处理程序，这就是将控制权返回给调试器的过程。内核从 WaitForDebugEvent 返回，唤醒调试器，然后更改它的窗口，并等待你的指令（参见后面的 CS 和 EIP 寄存器说明）。

此时，程序将冻结，由于它无法接收到 CPU 时间，当你与调试器交互时，程序

不能继续执行通过断点。一般来说，在程序中，调试器直接控制程序中的某一线程，以便别的线程继续接收 CPU 时间，在以后的章节中，将会说明怎样将线程挂起，而不是进行调试。

CS 和 EIP 寄存器说明

处理器含有少量的芯片存储区，即寄存器，它可以暂时将所需要处理的指令寄存起来。Intel 处理器含有 CS 和 EIP 两个寄存器，在处理过程中，它们不断指向下一个指令。这里简单描述了寄存器中的指针当产生中断后怎样引导处理器返回程序。

CS 代表代码段，在 32 位编程中也可以说是代码选择器，因为寄存器容纳指向程序码基址的值，这是操作系统赋予的，EIP（扩展命令指针）寄存器容纳一个 32 位偏移量，它指向处理器将要执行的另一个指令的代码区域。当它响应某一中断（诸如 INT 3 指令）时，Intel 处理器在跳到为中断服务的处理程序函数之前，首先将 CS 和 EIP 寄存器的内容写到堆栈上。当处理程序完成时，原始的 CS 和 EIP 值从堆栈中弹出，告诉处理器下一步该干什么。也就是说，在离开中断服务时，返回将要执行的程序指令。同 CS 和 EIP 一样，含有当前处理器标记的另一种寄存器也从堆栈中恢复，以便原始程序恢复执行它所不知道的（而且没有被影响的）中断。

发生中断的情况很多，这主要源于执行程序中的 INT 指令和硬件故障。例如，按下键盘上的一个键，即触发中断。当松开此键时，又会产生另一种中断。系统

计时器以每秒 18 次的速度不时地中断处理器，将控制权返回给操作系统，以便它能直接将处理器重新引导到时间片中的下一线程。中断服务很快，但断点中断是一个例外。操作系统和调试器保留控制权，直到恢复调试。此时，系统退出中断 3 处理程序，并且允许处理器继续运行正在调试的程序。

当恢复运行程序时，调试器将被 0xCC 值覆写的字节重置，并恢复设置断点处的原始指令。因为堆栈中的 EIP 值现在指向临时的 INT 3 指令之后的字节，调试器减少此值，以便能够重新指向原始指令。接着调试器通过调用 ContinueDebugEvent API 函数回到睡眠状态。操作系统将寄存器恢复到原始数据，并执行中断 3 处理程序中的 IRET 指令（中断返回）。处理器从堆栈中弹出 CS，并改变 EIP 值，继续恢复执行被中断的程序，就像什么也没有发生一样。

在所有运行的 Windows 程序实例均共享含有程序码的同一内存段时，情况更为复杂。由于调试器将 INT 3 指令写成程序代码，你可能希望程序实例在调试器之外运行，并且也能触发断点，并进入调试器，但这是不可能发生的。Windows 提供一种著名的复制写入机制，能将这种情形处理得相当好。当调试器调用系统的 WriteProcessMemory 函数，并将 INT 3 指令写入一页程序码中时，Windows 分配一块可写的内存，将代码页复制到新块中，重新映射该页的虚拟内存地址，利用复制的页面来完成调试器的写操作。另外一些在调试器外部运行的程序实例可从原始码中继续运行，并且不会遇到 INT 3 指令。

调试器采用不同的方法来捕获与程序数据中的变量相关联的数据断点，对用来改

变量的每个指令中设置 INT 代码是不切实际的，所以，调试器必须单步通过程序中的每条指令，每次都检查这些变量。如果变量没有变化，调试器就会执行下一条指令。可以想像，不间断的中断与检查可能会很大程度地降低被调试程序的执行速度，由于上千种中断均能够发生，所以，在利用数据断点来调试时，可能会程序挂起。

一些处理器所提供的特殊调试寄存器就在芯片上，可以使调试器脱离开这些冗长繁杂的任务，一个 Intel 处理器拥有 8 个调试寄存器，虽然它可以监视 4 种数据断点的最大值，因为仅仅是 4 个调试寄存器（DR0 到 DR3）中的第一个，并容纳内存地址。当你为变量设置数据断点时，调试器将变量的地址写入到处理器的调试寄存器中。接着使用 1 位标记来编写调试控制寄存器，以指示处理器去监视内存写指令。当处理器运行程序时，它会不断地检查每条指令，以弄清楚是否它已写到调试寄存器所指向的内存，如果是这样，处理器会产生中断 1，称为跟踪中断或调试中断，通过中断处理程序将控制权返回给操作系统。当系统从 WaitForDebugEvent 返回时，调试器从 DR6 状态寄存器读取信息，以确定四个断点中究竟是哪个触发了断点。调试器随之更新其窗口，以提醒你变量已经被改变。

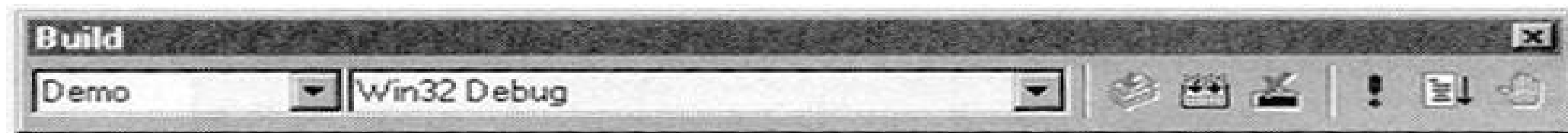
值得一提的是，无论什么时候，当程序存取一个变量时，也就是说，当它从变量中读取或者给它赋值时，Intel 处理器都能立即发生中断。而且，Visual C++ 调试器不会利用这种选项，因为通常情况下，与进行调试时的赋值变量的程序指令相比，读取变量的程序指令更缺少意义。用处理器的调试寄存器来监视位置断点和

数据断点，并用 INT 3 指令来保存覆写代码的额外工作，这也是可能的。但是，调试寄存器是珍贵的资源，因此，Visual C++ 调试器通过调用中断 3 来监视位置断点，并且保留使用数据断点的调试寄存器。

通过使用调试寄存器，调试器将检查数据断点的重担转给处理器，有效地消除了性能障碍，否则，将会导致每条指令都单步执行来监视数据的变化。然而，在断点包括条件时使用数据断点，例如变量 x 是否超出了某个值，数据断点也可能影响程序的运行速度。指定断点的一个变量在一个单循环的过程中可以改变很多次。变量每变化一次，处理器就会中断，控制权返回调试器，然后估价条件。永远或很少得到真值的条件表达式可能会因此降低程序的运行速度，后面的章节中将会谈到这种情况。在某些情况下，调试器不会使用调试寄存器来监视数据断点，而是依赖于速度更慢的单步调试法。

建立调试版本

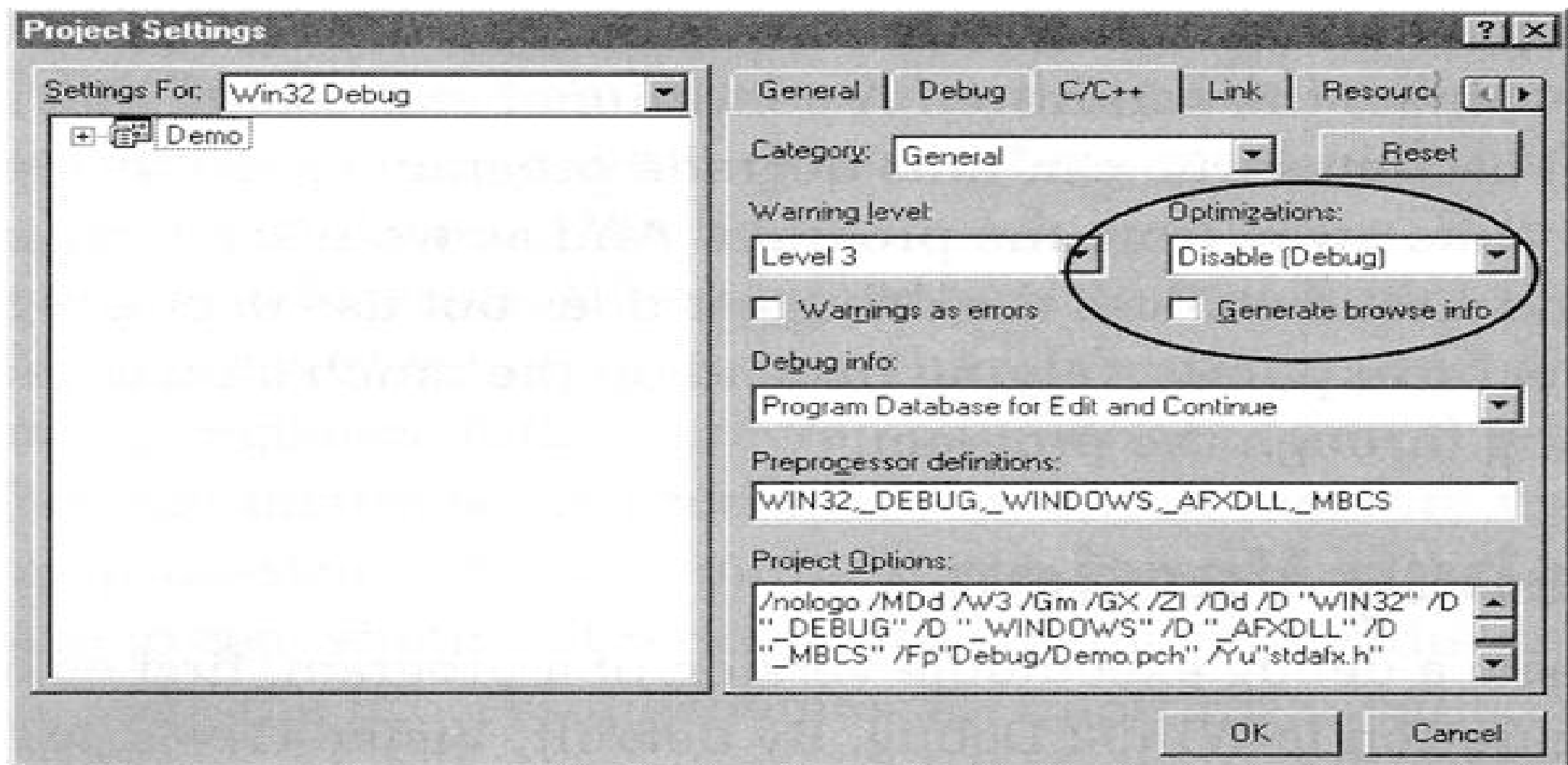
为了创建一个程序的调试执行版本，首先应该保证配置是 Win32 Debug。默认情况下，当你创建一个新项目时，Visual C++ 即设置配置为 Win32 Debug，在 Build 工具栏中显示当前的配置：



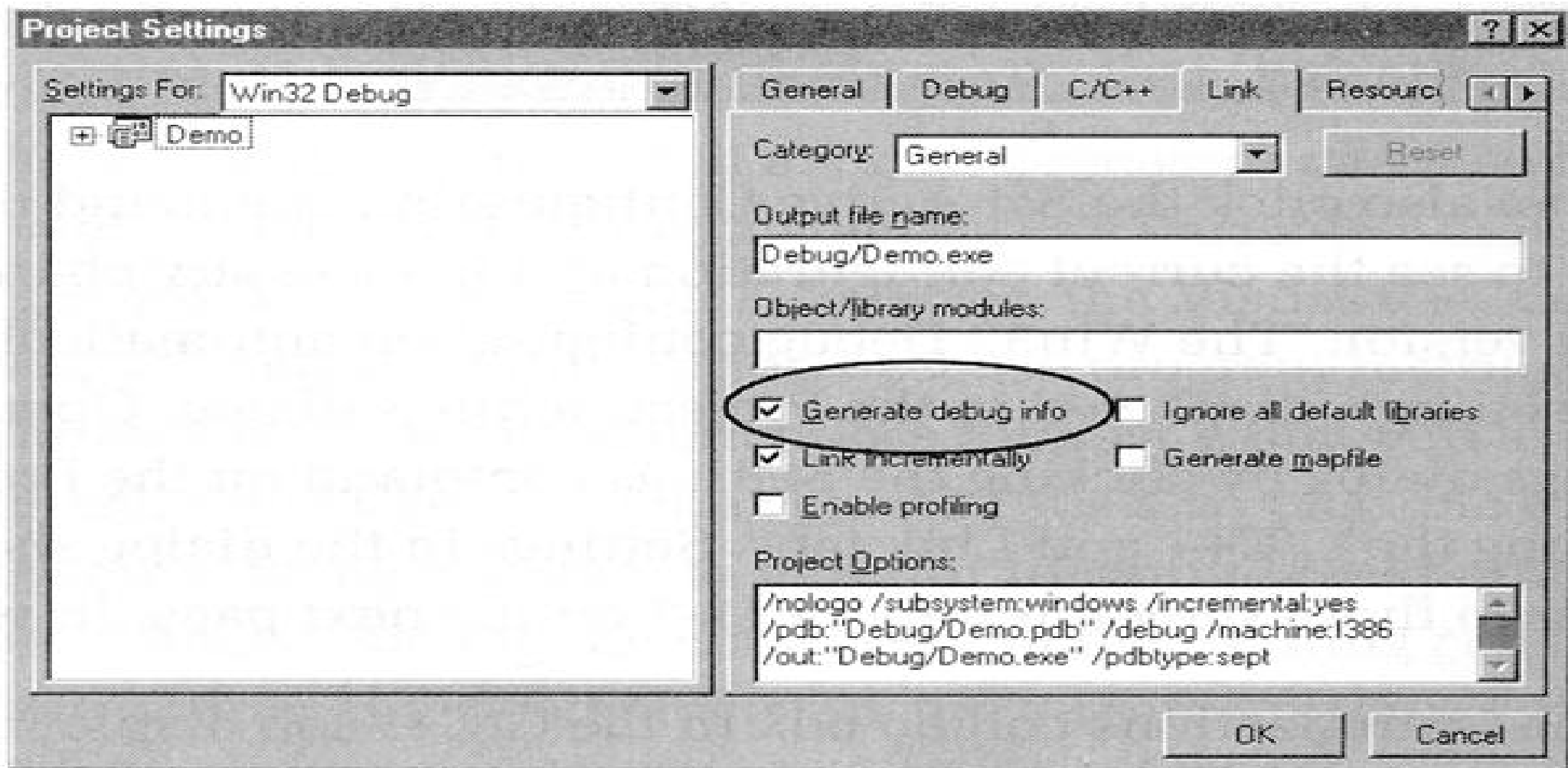
也可以在 Build（建立）菜单中单击 Set Active Configuration（设置活动配置）命令，以查看当前的配置，看是否有必要将它更改为调试版本。Win32 Debug 配置自动改变为程序设置，并在 Project Settings（项目设置）对话框中显示出来。通过单击 Project（项目）菜单中的 Settings（设置）命令，打开 Project Settings（项目设置）对话框，并调出 C/C++ 和 Link 选项卡，对话框中的设置应和图 11-1 中所示类似：

- 在 C/C++ 选项卡上的 Optimizations（优化）组合框显示 Disable（Debug）（禁用（调试））选项。
- 在 Link 选项卡中，复选标志显示在 Generate Debug Info（生成调试信息）复选框中。

有了这些设置，就可以顺利的建立项目了，其结果是一个针对调试器含有符号信息的程序调试版本。



C/C++ 选项卡中的设置



Link 选项卡中的设置

图 11-1 在 Project Settings 对话框中设置调试信息

调试器界面

像 Visual C++ 编辑器一样，调试器仅适用于 Developer Studio 环境。调试程序要求项目是开放式的，并且，你应该创建一个程序的调试执行版本。

文本编辑器是开始调试的好地方，打开一个或更多个程序源文件，找出程序运行时需中断执行的行，单击此行所在的位置，将在此处设置插入标志，然后按 F9 设置一个位置断点。编辑器在所选行的左边放置一个小的红色八角形，来作为此行的标记，以表示中断。如果禁用选择边界，如第 3 章所述，编辑器会将整行用红色高亮显示。为了取消位置断点，可在要设置位置断点的所在行的任一位置设置插入标志，并重新按 F9 来消除断点。

如果想使用鼠标而不是键盘，可以在此行单击鼠标右键，即可设置或消除断点。如图 11-2 的上下文菜单中所示。从此菜单中，可选择 Insert/Remove Breakpoint（插入/删除断点）命令来清除或设置断点。此菜单也提供一个 Disable Breakpoint（禁用断点）命令，允许关闭断点而不去消除它。

虽然有点不太方便，但也可以通过 Breakpoint（断点）对话框来设置一个位置断点，此对话框仅提供设置数据断点的方式，和两个其他变量，称为条件断点和消息断点。



图 11-2 从上下文菜单中选择 Insert/Remove Breakpoint (插入/删除断点) 命令

Breakpoint（断点）对话框

图 11-3 所示为 Breakpoint 对话框，按下 Ctrl+B 键或在 Edit（编辑）菜单中单击 Breakpoint 命令，对话框中的三个选项卡可以设置位置断点、数据断点、条件断点和消息断点，后面介绍这四种断点类型。

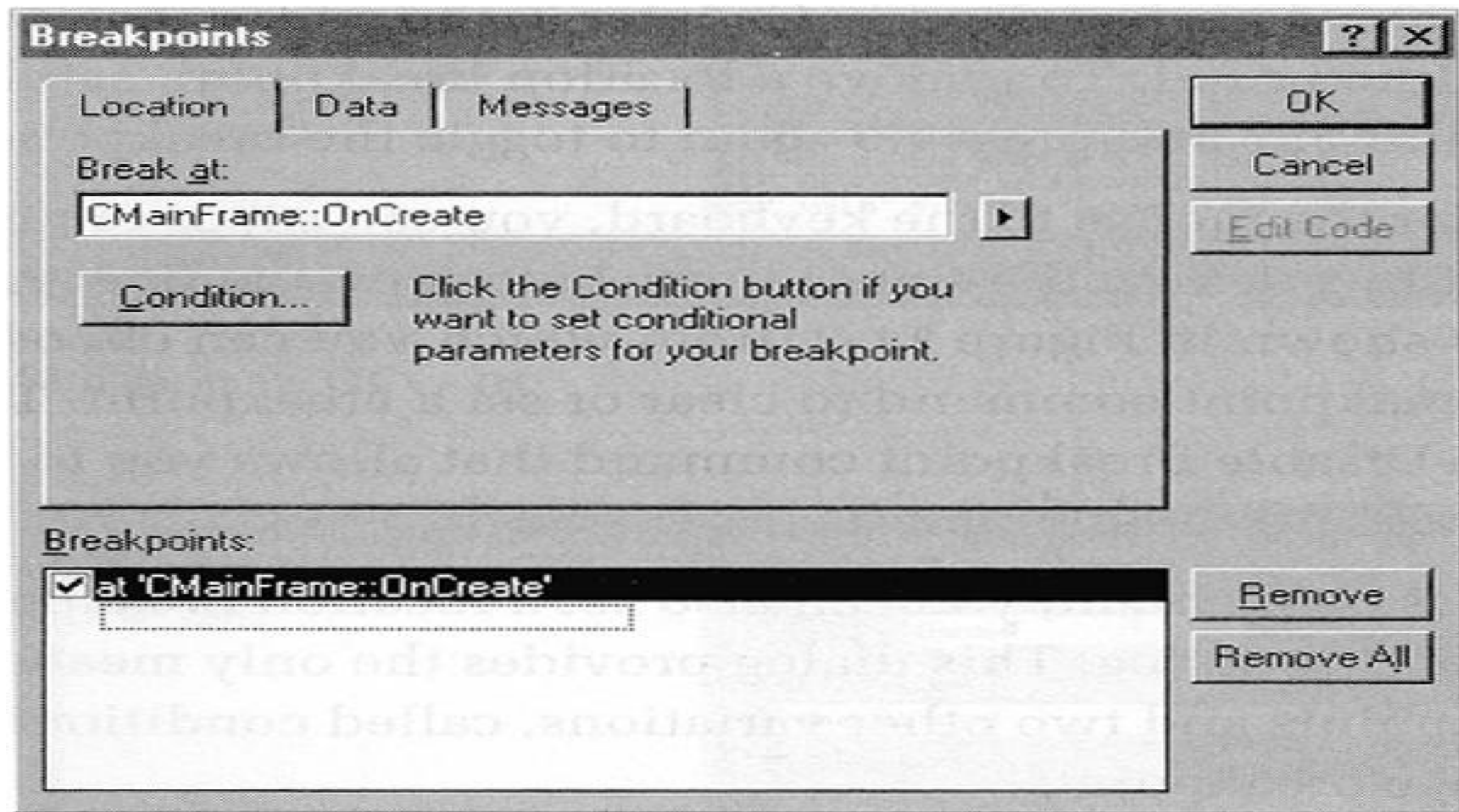


图 11-3 Breakpoint (断点) 对话框

位置断点

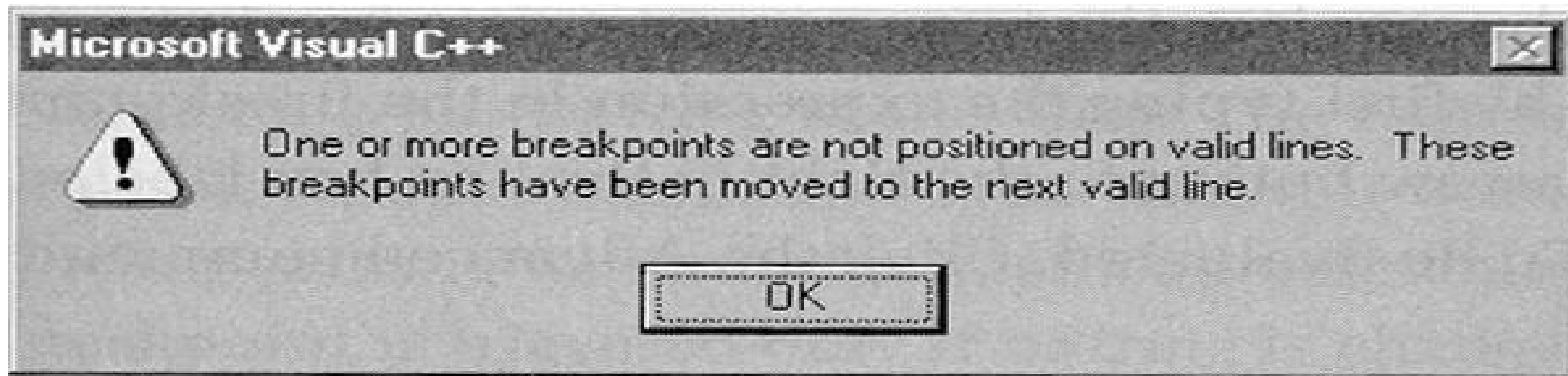
在 Breakpoint（断点）对话框中设置位置断点比按 F9 键或从编译器上下文菜单中选择一条命令要麻烦得多。但是，对话框对位置断点来说，提供了几种增强功能，这些被证明是非常有用的。你可以在 Break At（断点位置）控制处键入函数名，以便在函数的第一行处设置一个位置断点；键入标号的名字在标志行处设置一个断点（在这二者之间有一点细微的差别，因为函数名仅仅是一个标号）。在 Break At 控制中的字母必须与函数名或标号相符合，并且，C++ 函数名必须包括类名和作用域解析运算符。因此，OnCreate 条目不会指定有效的断点位置，而 CMainFrame::OnCreate 却可以做到这一点。

在 Breakpoint 对话框中键入函数或标号名，以便在源文件中设置一个有效的断点，但它不能给文本编辑器提供行号。因为文本编辑器要求行号来显示断点符号，所以，它不能在文档窗口中标记此行，或者，给予其他已有位置断点的标志行一些别的可视线索。在 Breakpoint 对话框底部列出的断点，只能用来确认新断点的存在，断点符号的停止标志仅仅在调试器处于运行状态时的源文件中显示出来。

为了在一个特殊行中设置位置断点，在紧接着的 Break At 控制的行号处键入一个句点。对于当前行，也就是说，在文本编辑器中含有插入记号的所在行，单击控件右边的小箭头按钮，并选择所给的行号，这个按钮能够引出 Advanced Breakpoint（高级断点）对话框，你可以据此区分源文件中的函数或标号，而不

是根据文本编辑器中显示的内容。

位置断点有类似于文本编辑器书签的特征，并且，Visual C++使用相同的逻辑实现断点和书签。除了标记特殊行以外，位置断点就像命名的标签一样，它会保持在文档中的固定位置，除非你消除它。如果在 Developer Studio 环境之外编辑了一个文档，断点和书签就会均显示在另一行中。在调试器处于运行状态的环境下编辑文档时，也能与此位置之外的位置断点发生冲突，因为位置断点与行号相联系，随着文档在编辑过程中变大或缩小，新的语句可能会进入标有断点的行号中。如果新行不含有效的程序指令，Visual C++在你下一次启动调试器时会提供警告：



当出现这种消息时，应该使用文本编辑器检查源代码，消除不适当的断点，在其原来的位置上重新设置新的断点。如果你经常在 Visual C++环境之外编辑源代

码，或者在调试器处于运行状态时进行一些变化，以期能不断地了解一些信息，对有效地解决重置断点之类的问题是比较棘手的，尤其是你在另一个编译器中做了某些变换时，就像书签、断点是指针，而不是嵌入源代码文档中的字符一样（仅使编译器出错），因此，改变该环境外的文本将不可避免地有重置指针目标的危险。

然而，重置问题并不会影响使用 Breakpoint 对话框时与符号或函数名相联系的位置断点，无论文档内容怎么改变，这些位置断点对其标志来说总是保持固定，因为它每运行一次，调试器就会搜寻标志的源代码，用断点给每一个标志行作上标记，这经常是在使用 Breakpoint 对话框时设置位置断点的原因，远没有按 F9 设置方便。在以后的章节中，我们将使用对话的方式在一个调试例子中来设置位置断点。

数据断点

Breakpoint 对话框仅仅提供了一种设置数据断点的方法，当变量的值有所改变，或者，当条件表达式变成真时，数据断点即被触发。如果已经使用了微软的旧式 CodeView 调试器，你可能会将数据断点作为 CodeView 中的跟踪点。单击 Breakpoint 对话框中的 Data（数据）选项卡，键入你想利用调试器进行监视的变量或表达式的名称（参见图 11-4），进入一个标准的 C/C++ 条件表达式，例如 `i==100` 或 `nCount>25`。

当调试器处于运行状态时，可以为不在作用域中的变量设置数据断点，方法是先

在 Breakpoint(断点)对话框中键入表达式,然后单击标有 Enter The Expression To Be Evaluated (输入要求值的表达式) 文本框右边的箭头按钮。单击弹出的 Advanced (高级) 命令,并输入所要求的文本信息,使调试器跟踪进入作用域的变量。

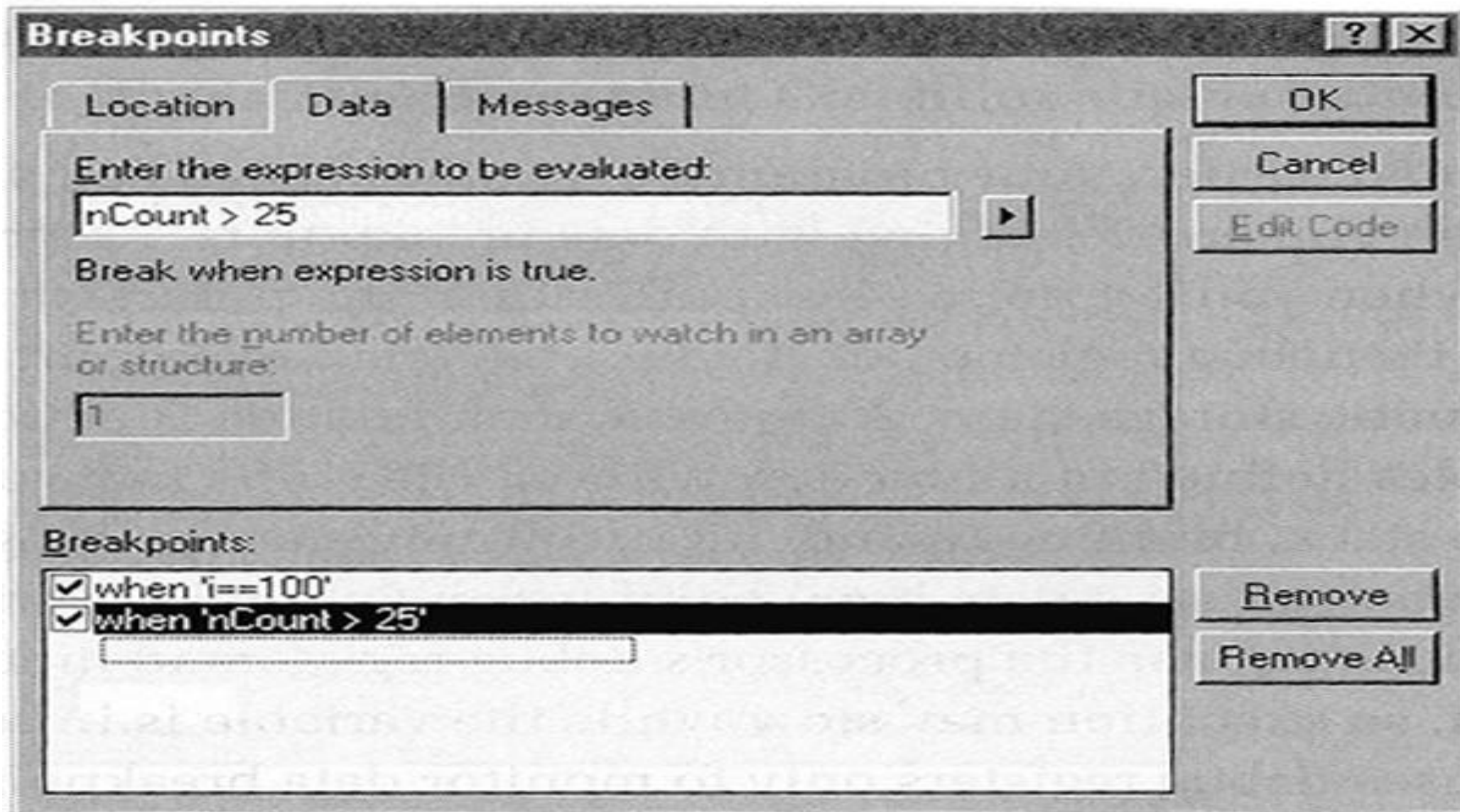


图 11-4 在 Breakpoint (断点) 对话框中输入数据断点

调试器能够监视指针所标识的变量范围，如数组或结构名，以提供表达式中指针的重引用状况。例如，在 Enter The Expression（输入表达式）文本框中键入数组名 `iArray`，就不可能按你所期望的那样为数组的第一个元素设置断点，必须通过键入 `iArray[0]` 来重引用数组指针。为了监视数组的其他元素，需在 Enter The Number Of Elements To Watch 的小控件中设置元素数。值得注意的是，这是元素数，而不是字节数。例如，如果 `iArray` 含有整数，在第一个控件中键入 `iArray[0]` 和在第二个控件中键入数字 10，如果变化发生在数组的前 40 个字节（从 `iArray[9]` 到 `iArray[0]` 的整数），就会引起程序中断。

类似地，为了监视变量 `pString` 所指向的一串字符，在 Enter The Expression 控件中键入 `*pString`。在较小的控件中，键入你想利用调试器监视的字节数，键入不带星号的 `pString` 重引用运算符意味着，只有在 `pString` 改变为指向别的地方时，再触发断点。在这种情况下，调试器所监视的是 `pString` 本身，而不是它所指向的数组的内容。

以前提到过，当用数据断点调试时，程序的执行速度将会明显减慢。当设置的断点比处理器中的调试寄存器所能容纳的断点数要多，或者用自动存储类为变量设置了数据断点时，程序运行速度也会降下来。自动数据包括函数自变量以及在没有 `static` 关键字的函数中所定义的变量。这类数据存在于堆栈中，当程序运行时就不再存在。尽管你可以为自动变量设置数据断点，但调试器也不会利用处理器的调试寄存器来监视断点，因此，当变量在作用域内时，执行速度就会下降。调试器利用调试寄存器监视的仅仅是存在于程序数据段中的静态变量的数据断

点，而不是堆栈中的局部数据。

由于数据断点会使程序的执行速度大大降低，可能会使你认为程序已被挂起了。使用数据断点时，需要耐心。如果你认为程序确实因为某些原因停止了，在 Visual C++ Debug（调试）菜单中单击 Break（中断）命令，这就会中断程序，并将控制权返回给调试器。

条件断点

条件断点是位置断点的扩展方式，在源代码指令中设置断点与设置位置断点的方式相同。但是，当控制到达带标记的指令时，如果指定的条件是真，调试器就会响应条件断点。条件断点在同一指令执行数百次的循环中用处不大，设置在循环中的位置断点在每次重复过程中都会停止执行，这是我们所不希望的。条件断点中断指令仅在某些情况下发生，例如当循环次数达到 100 时。

在 Breakpoint（断点）对话框的 Location（位置）选项卡中设置条件断点，在定义想利用断点标记的源代码后，单击图 11-3 中所示的 Condition（条件）按钮，显示 Breakpoint Condition 对话框：

Breakpoint Condition



Enter the expression to be evaluated:

iCount == 100

OK

Cancel

Break when expression is true.

Enter the number of elements to watch in
an array or structure:

1

Enter the number of times to skip before
stopping:

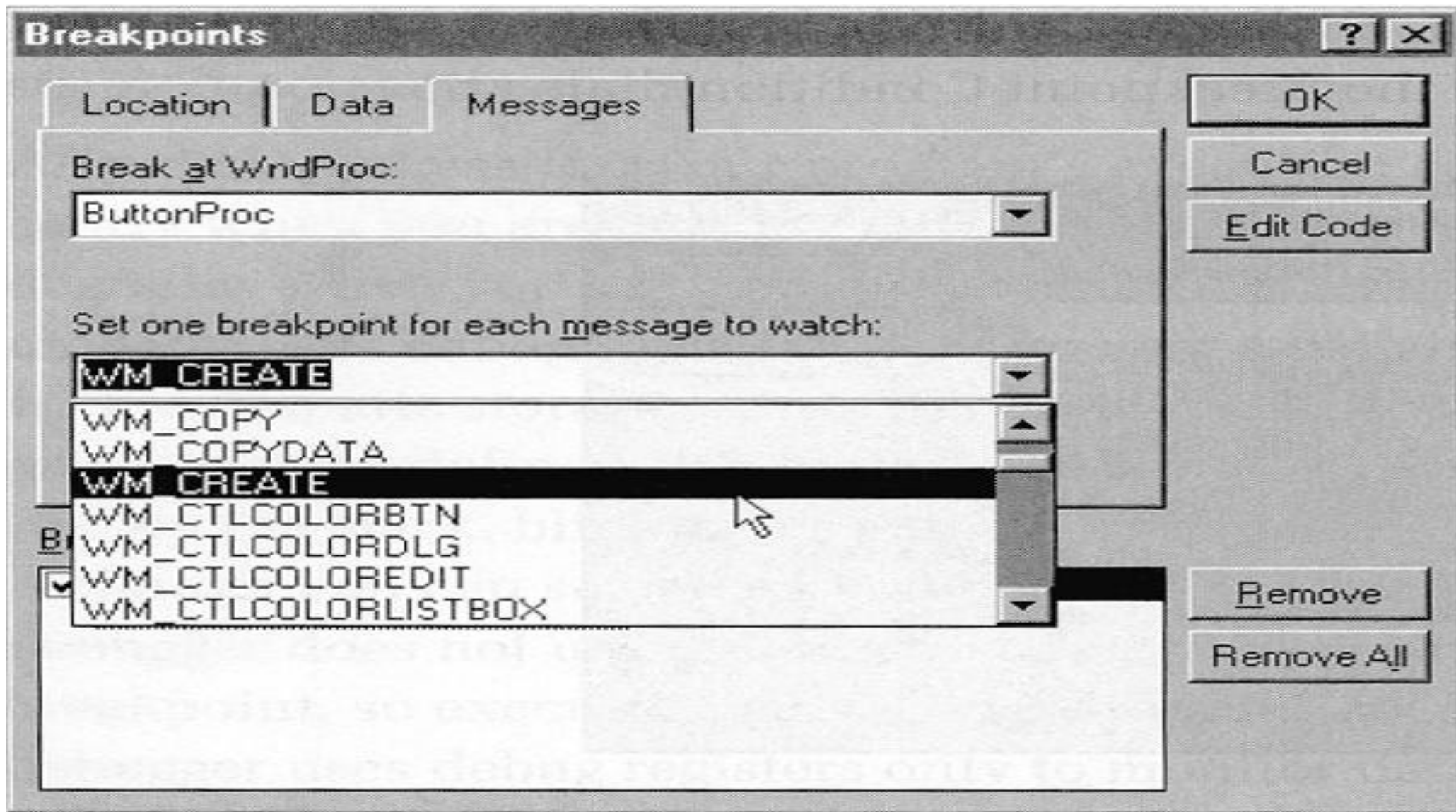


图 11-5 在 Breakpoint (断点) 对话框中设置消息断点

当操作系统调用 ButtonProc 过程时，它会在 msg 参数中传递诸如 WM_COMMAND 或 WM_CREATE 的消息值，以通知程序为何被调用。当 ButtonProc 接收到一条特定的消息时，为了中断运行，单击 Breakpoint（断点）对话框中的 Message（消息）选项卡，并在 Break At WndProc 控件中键入 ButtonProc，单击第二个组合框中的箭头，显示出消息标识符的下拉列表，并选择一条消息，诸如在图 11-5 中所示的 WM_CREATE 消息。当运行调试器中的程序时，在 Windows 调用带有 WM_CREATE 消息的过程时，在 ButtonProc 的第一行处将中断执行。

可以看到，消息断点就是条件断点的特殊形式，在 Location（位置）选项卡中可以获得相同的结果，方法是对具有如下条件的 ButtonProc 标号设置位置断点：

```
msg == WM_CREATE
```

运行调试器

一旦确定了程序停止的条件和位置，就作好了运行的准备。现在，是文本编辑器而不是调试器处于运行状态。执行程序的调试版本，只要启动调试器，接着运行程序。

从 Build（建立）菜单中选择 Start Debug（开始调试）命令，将出现四个名字为 Go（转到）、Step Into（开始内部单步调试）、Run To Cursor（运行到光标处）和

Attach To Process（与进程联系）的选项，如图 11-6 所示。在源代码中设置了至少一个断点时，即可以使用 Go 命令。调试器运行程序时，当程序中的执行流到达位置断点或触发数据断点时，就会停下。顾名思义，Step Into 命令的意思是：进入程序，并且在第一条指令处终止。Windows 程序的第一条指令是 WinMain 函数的开始处，或者，对于 MFC 程序，是 _tWinMain 函数的开始处。在另外一种情况下，调试器打开源模块，对 _tWinMain 来说，是位于 MFC 文件夹中的 Appmodul.cpp 文件，并显示在源文件的窗口中。

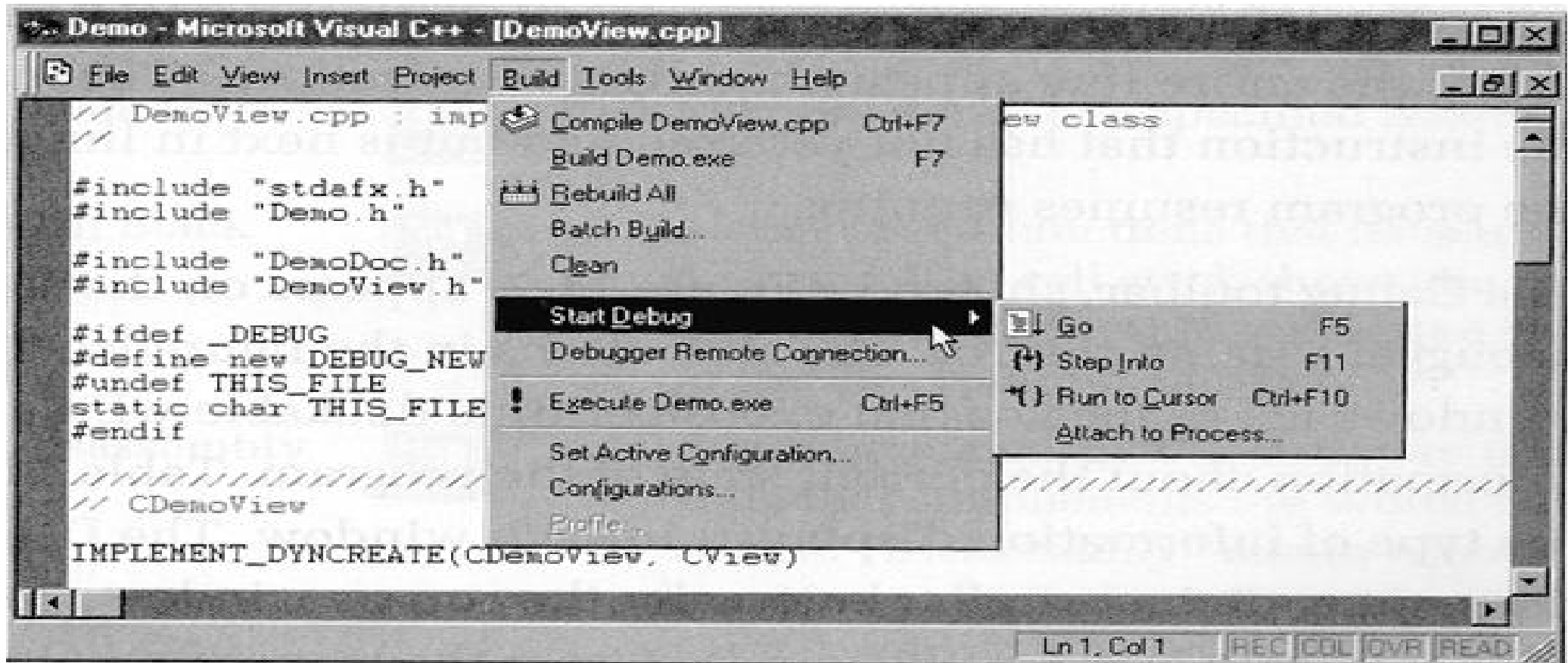


图 11-6 从 Build（建立）菜单中启动调试器

Run To Cursor 命令在插入标记符号的源代码行处终止执行，如果在文本编辑器中没有源文件打开，Run To Cursor 命令将会失去作用。否则，它将提供较方便的方式进入程序中，而无须设置一个断点，如果程序流在到达插入记号之前触发了断点，在断点处，执行程序就会停下，而不是在带有插入记号的行。为了继续

运行程序，在目标行处重新设置插入记号，再次单击 Run To Cursor。利用 Attach To Process（与进程关联）命令，可启动调试器，并且可将它与目前运行的程序联系起来。调试器通过 DeugActiveProcess API 函数来完成这种功能，在在线帮助中有所描述。

调试器为 Start Debug（开始调试）中的前三个子命令提供了快捷键，因此，可以不打开 Build（建立）菜单来开始调试。快捷键 F5 的功能是 Go（转到），F11 快捷键的功能是 Step Into（开始内部单步调试）。Ctrl+F10 快捷键的功能是 Run To Cursor。

调试器窗口

调试的程序在某一断点处停止时，调试器更新带有目前程序运行状态消息的窗口。也许调试器窗口中最重要的是源程序窗口，它能显示程序所停之处的源代码。称为指令指针的小黄色箭头出现在中断指令左边的选择边界处（如果选择边界禁用，整行将会以黄色高亮显示）。这种标志表明程序恢复运行时，将要执行而还没有执行的下一行。

如图 11-7 所示，当调试器重新获取控件时，Debug(调试器)工具栏出现在屏幕上。图中标有 Debugger Windows 的六个按钮充当触发器，显示或隐藏含有目前程序状态消息的窗口。表 11-1 描述了显示在每个窗口中的消息类型。Debug(调试器)工具栏对源程序窗口来说不含有类似的按钮，因为调试器仅仅从文本编辑器中借

用窗口。当你需要查看普通文档时，打开或关闭源程序窗口即可。

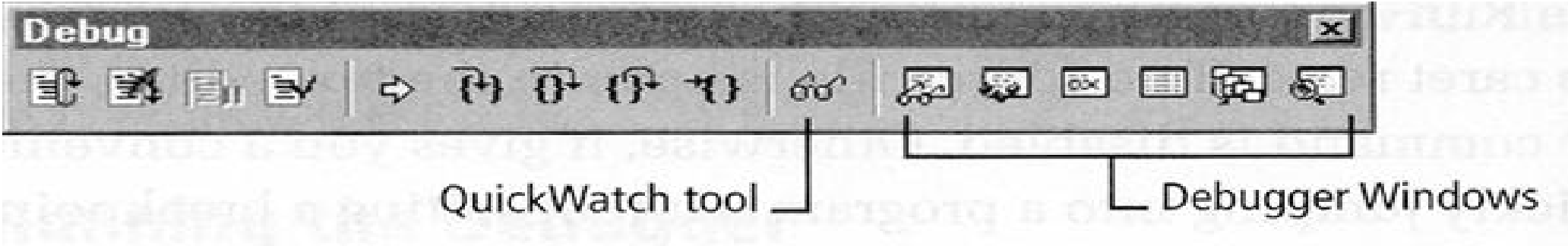
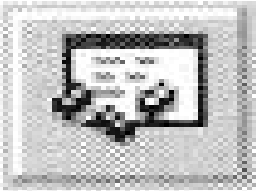


图 11-7 触发调试器窗口开关的工具按钮

表 11-1 被 Debug(调试器)工具栏中的按钮激活的六个调试器窗口中的消息

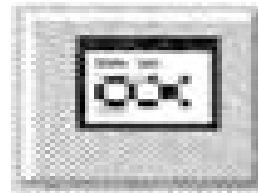
窗 口	按 钮	说 明
Watch(观 察)		通过调试器跟踪的目前变量和表达式的值。在 Watch 窗口中，指定当程序被挂起时，要了解哪些变量的当前值
Variables（变量）		在访问或靠近中断位置处的当前变量值，Variables（变量）窗口具有以下三个选项卡： Auto（自动）：显示变量和函数的返回值

续表

Locals (局部)：显示当前函数的局部变量

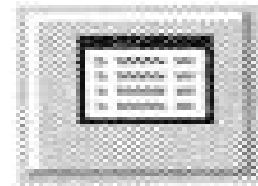
This (当前)：在一个 C++ 程序中，标识指针目前所指向的目标

Registers (寄存器)



CPU 寄存器的当前内容

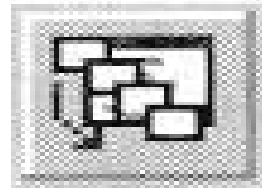
Memory (内存)



特定地址的内存转储

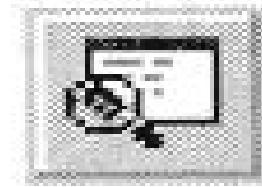
续表

Call Stack (调用堆栈)



没有返回的被调用函数的列表，Call Stack (调用堆栈) 显示通过嵌套函数调用断点位置的执行路径

Disassembly (反汇编)



在屏幕上，为了补充源程序窗口，其中显示编译代码的汇编语言转换，“Disassembly (反汇编)”意味着将程序中的机器码转化为相当的汇编指令

Watch 窗口提供了特定变量的概况，显示出当程序挂起时这些变量的当前值。Watch (观察) 窗口的变量与中断程序流没有关系，因此，不要将观察到的变量与数据断点中所设置的变量混淆。为了在 Watch 窗口增加一个变量，在窗口中双击带点的新项目框，并键入变量名。

图 11-7 中所示的 QuickWatch (快速查看) 工具栏，对没有在 Watch 窗口中添加的变量，提供了一种查看的方法。为了方便起见，可以简单地在源程序窗口中将鼠标光标指向变量名来查询其当前值。这将在含有当前值的工具提示窗口中显示出来：

```
if (bCreateFlag && iID > 1)
{
    bCreateFlag = 1
}
```

对诸如结构元素和类成员的一些变量，对调试器来说，可能不能确切地辨别这些变量。在这种情况下，必须首先选择目标和变量名以及相关的点运算符（同在 MyClass.Member 中的一样），然后将鼠标光标停在源程序窗口的选择项上。

在 Watch 窗口中，可以看到作用域中的变量，而不管它们是在程序中的什么地方获取的。Variables 窗口将集中在运行的冻结点上。在程序挂起之前，最终运行的指令（可能还有前一两条指令）所引用的所有变量，将出现在 Variables 窗口中。你也可以通过在 Variables 窗口中双击变量，并键入新值，来改变它的值。

Registers（寄存器）窗口一般仅当 Disassembly（反汇编）窗口处于运行状态时使用，当程序被挂起时，它显示处理器寄存器的状态。在表 11-2 中所描述的 Intel 处理器标记就是位标记，以及双击 Registers（寄存器）窗口中的标记所触发的值。表中的符号栏显示出现在 Registers（寄存器）窗口中的标记符号。

调试器窗口能够存储许多信息，但在一般情况下，不必在同一时间内查看它们。窗口还具有显示源程序窗口中的源代码的屏幕空间，一般情况下，总可以看到它们，以便你知道当时正处于程序中的什么地方。

表 11-2 Intel 处理器标记

标 记 名	符 号	说 明
Overflow (溢出)	OV	当一个整数指令所产生的结果太小或者太大，不能填入目标寄存器或内存地址中时，便设置此标记
Direction (方向)	UP	决定重复字符串和比较指令的方向，例如 MOVS (Move String) 和 CMPS (Compare String)
Enable interrupt (启用中断)	EI	当清除它时，处理器忽略硬件中断，如键盘活动

续表

Sign (正负号)	PL	含有一条算术指令的高序位值 (不清楚为什么 Visual C++ 设计者选择字母 PL 来代表符号标志, 不要将它与处理器的 I/O Privilege Level 标志相混淆)
Zero (零)	ZR	当一条算术指令的结果是零时设置
Auxiliary carry (附加进位)	AC	在一条算术指令后, 含有 AL 寄存器的四个低序位 (也叫做半字节) 的进位
Parity (奇偶性)	PE	当一条算术指令结果的二进制值是偶数个 1 位时设置
Carry (进位)	CY	类似于 Overflow 标记, 但指示无符号的溢出, 可以通过 STC (Set Carry) 和 CLC (Clear Carry) 指令来控制

如第一章所提到的, 调试器使用环境 Output (输出) 窗口来显示来自 OutputDebugString 函数或 afxDump 类的的数据。Output (输出) 窗口也能显示线程终止代码、第一次异常通知, 并调出信息。在 Debug 工具栏中, 没有控制显

示 Output 窗口的按钮，因为此窗口属于 Developer Studio 环境，而不是调试器。通过单击标准工具栏上的 Output 按钮，将显示或隐藏 Output 窗口。

只有真正使用调试器窗口，才能掌握它。在以后的章节中，将会给出一个例子来描述调试器窗口怎样调试进程。

单步调试程序

当利用调试器跟踪一个程序的错误时，应当确定你认为有问题的代码段，并在此段或刚好在此段之前设置一个位置断点。当调试器在断点处将程序挂起时，你可以单步调试程序，每次执行一条指令，检查变化时的变量状况。

如图 11-8 所示，Debug 工具栏拥有一组四个按钮，可以让你单步调试被挂起的程序。可以通过箭头粗略地了解 Step 工具；现在我们知道，按钮上的图像表达了它的作用，在如下次序所显示的按钮中，分别激活 Step Into（开始内部单步调试），Step Over（整体单步调试），Step Out（完成内部单步调试）和 Run To Cursor（运行到光标处）命令，在这之前，我们已经讨论了 Run To Cursor（运行到光标处）命令，其他的三条命令则需要更多的解释。

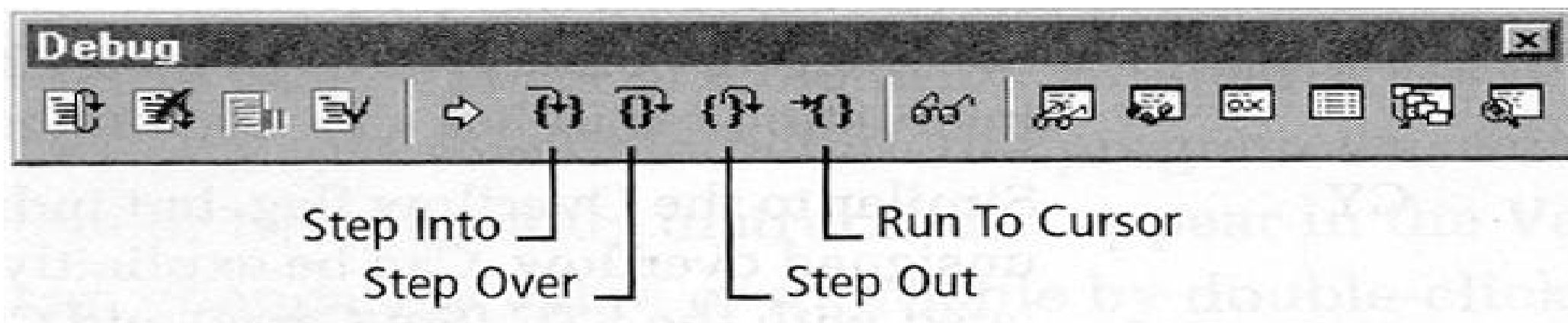


图 11-8 Debug(调试器)工具栏中的四个 Step (单步调试) 工具

利用 Step Into 和 Step Over 命令 (或者它们等同的快捷键 F11 和 F10), 可以单步调试程序。选择 Step Into 或 Step Over 命令时, 调试器可以让程序恢复运行, 但仅仅执行一条指令。当指令完成时, 调试器重新将执行程序挂起。你可能会想, 是什么组成了一条指令, 因为在高级语言, 像 C++ 中, 一条简单的命令可以转化成 12 条处理器级别的机器码指令。它取决于是否启用了 Disassembly (反汇编) 视图, 如果启用了, 则单步操作的指令仅仅是当前的机器码指令, 移动箭头将指向反汇编列表中的另一条指令。如果没有启用反汇编视图, Step Into 或 Step Over 命令执行当前的源程序窗口中的 C/C++ 指令, 处理由一组机器码构成的指令。

当 Step Into 和 Step Over 命令作为调用函数指令使用时, 你会对它们有更深的认识。思考一下, 当调试器在如下的 if 语句中终止程序运行时会发生什么:

```
if (Function1(hdc, Function2(msg)))
```

```
    x=3;
```

```
else
```

```
    y=100;
```

Step Over 命令所执行的如其名字所指的一样，处理整个 if 语句，包括调用 Function1 和 Function2。程序在 x=3 或 y=100 语句处会再次终止。Step Into 命令处理这种情况稍有不同。当你在 if 语句中单击 Step Into 按钮时，调试器进入 Function2，并在第一条指令处停下。如果此时检查 Call Stack（调用堆栈）窗口，将会在此列表的顶部看到 Function2，并在下部看到你刚离开的函数名。

此时，Step Out 命令就有用了，这个命令执行当前函数的其余部分，接着，在所调用函数之后的下一条语句处停下来。换句话说，所调用的 Step Into 和 Step Out 命令一起作用于一个函数时，它有着与 Step Over 相同的作用。而且，如果指令在所给的例子中含有不止一个函数，事情将会更复杂。在条件语句中单击 Step Into 按钮，并在 Function2 的第一条指令处终止时，继续单击 Step Out 命令，指令指针的箭头保持指向 if 语句。这是因为 Function2 已经完成运行，但 Function1 还没有被调用，再次激活 Step Into 指向 Function1 的第一条指令。如果单击 Step Out 命令返回 Function1，指令箭头仍然指向 if 语句，因为 if 语句检测自身还没有执行。

利用反汇编的代码视图，可以对所发生的一切更清楚，带阴影的行指示 C 源程序语句，紧跟的是等价的反汇编指令（反汇编行仅仅用来阐述 if 语句中的内部事件链，并不意味着 Disassembly（反汇编）窗口是可视的）。在代码序列的开始处，黄色的指针箭头指向 if 语句：

```
if (Function1(hdc, Function2( msg )))
```

← Instruction pointer arrow is here.

```
mov     eax, dword ptr [msg]
```

```
push    eax
```

```
mov     ecx, dword ptr [this]
```

```
call    @ILT+30(CMyClass::Function2) (0040101e)
```

```
push    eax
```

← First Step-Into-Step-Out processes

```
mov     eax, dword ptr [hdc]
```

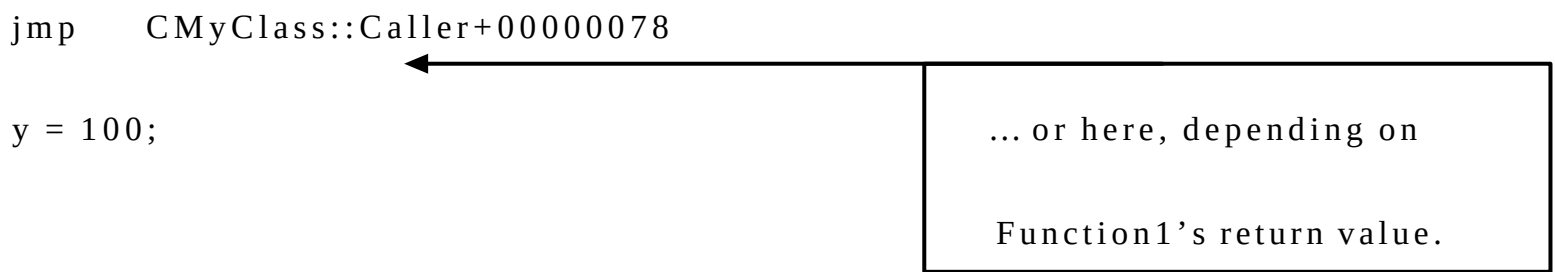
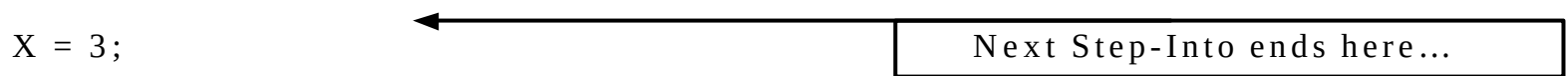
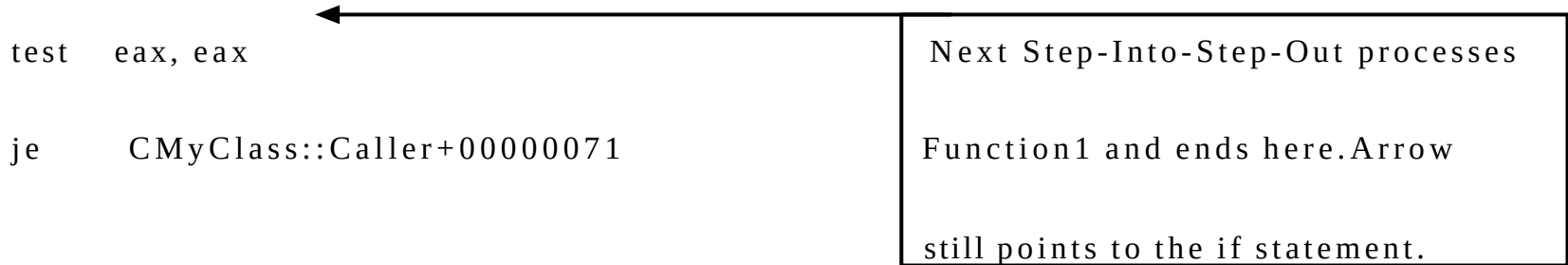
Function2 and ends here.Arrow

```
push    eax
```

still points to the if statement.

```
mov     ecx, dword ptr [this]
```

```
call    @ILT+85(CMyClass::Function1) (00401055)
```



Windows 95 不允许进入系统 API 函数进行单步调试，如下所示：

```
::SlectObject (hdc,::GetStockObject(BLACK_PEN));
```

如果在这条指令中选择 Step Into, SlectObject 和 GetStockObject 在调试器停止运行之前继续执行下一语句。在这种情况下, Step Into 和 Step Over 具有相同的作用。

终止和重新启动调试器

如图 11-9 中所示, Restart (重新启动) 按钮可以取消执行命令, 并从头开始重新运行程序, 废弃诸如系统资源或内存之类的分配。这可以了结以前的状态, 而无须退出并重新启动调试器。单击 Stop Debugging (停止调试) 按钮, 可立即退出调试器, 一步取消调试器和程序。Break Execution 按钮同 Debug (调试) 菜单中的 Break (中断) 指令一样, 有着同样的作用, 终止程序的运行, 并将控制权返回给调试器。选择 Break Execution (中断执行) 按钮来终止无限循环中的程序。

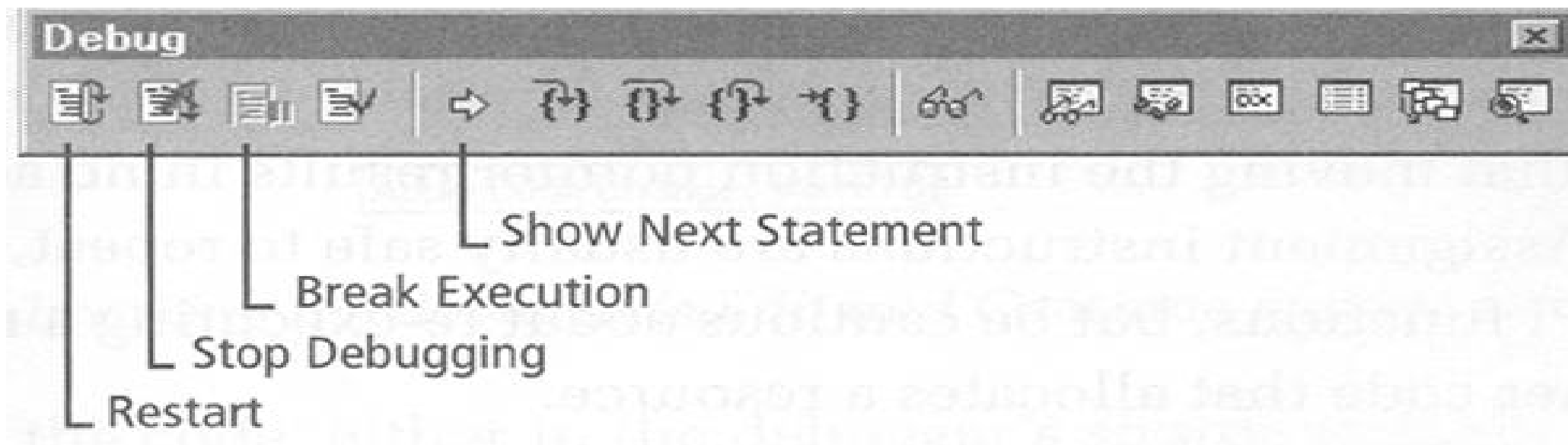
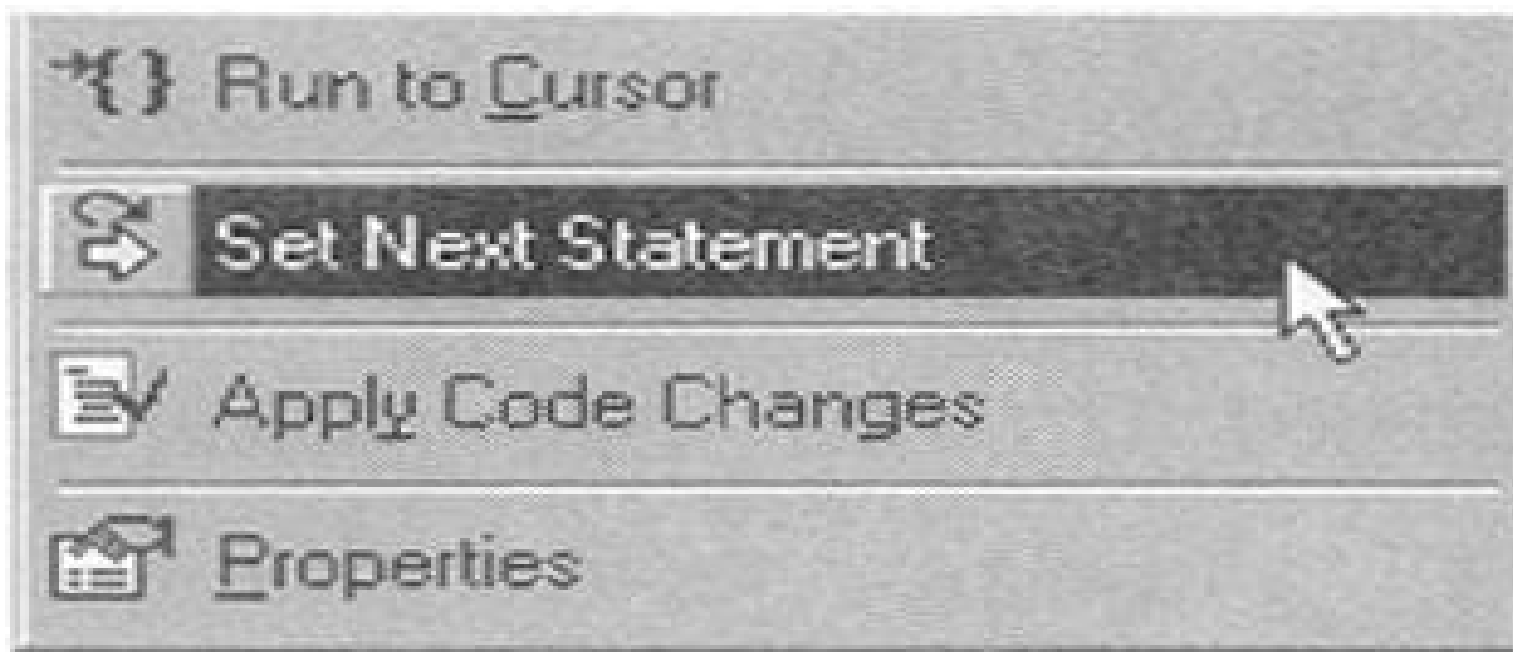


图 11-9 控制执行点的调试器工具

标有 Show Next Statement (显示下一语句) 的小黄色箭头可以很好地解决以前的问题, 当程序被挂起时跟踪错误, 经常要求查询其他部分的文档, 并且, 甚至会使你进入别的源文件中。单击 Debug(调试器)工具栏中的 Show Next Statement(显示下一语句) 箭头, 会马上带你返回到源程序窗口中的终止指令处。从这个工具图标中, 可以联想到出现在邻近终止指令选择边界中的指令指针箭头。

随机校正

在调试程序时发现了错误时，Visual C++在许多情况下允许你不停止调试器而继续进行校正。例如，一个变量含有一个错误的值，在 Variables（变量）窗口中，键入正确的值，很容易就可以继续执行程序。如果喜欢从一条新指令中恢复运行程序，右击源程序窗口需要的行，从上下文菜单中选择 Set Next Statement（设置下一条语句）命令：



这将会使指令指针重新再次指向单击过的行，允许你重做或跳过程序中的指令。而且，对你来说，这是一个负担，你要保证移动的指令指针方向不要错。重复赋

值指令通常来说是比较安全的，像许多调用的 API 函数一样，但是，在重新执行一条 new 指令或分配资源的其他代码的情况下，一定要当心。

在调试会话中键入一个新值，这是比较好的临时处理办法。它意味着，最终你将退出调试器，使用文本编辑器修整错误的源代码，并重新编译，得到正确的结果。但 Visual C++ 现在提供了一个更方便的方法，对那些运行过的步骤，你可以在调试器的源窗口修改出现的问题，而且这种修改是永久性的。在编辑完源程序后，重新恢复运行程序时，Visual C++ 首先编译已校正的代码，并用正确的版本替代被影响的机器指令。这种修改是永久性的，尽管你关闭了调试器，重新编译被校正的源代码，并再次启动调试器。

Edit and Continue（编辑和继续执行）特征是众所周知的，在调试程序时，总是使用它们。尽管你可以选择通过手工或者自动应用这些特征。默认情况下，当你选择 Go（转到）或 Step 命令中的一条命令，在修正源代码后，继续执行被中断的程序时，它们自动处于激活状态。如果你倾向于更多地控制调试会话中变换的代码是否重新编译，则显示 Options（选项）对话框中的 Debug 选项卡，清除标有 Debug Commands Invoke Edit And Continue（调试命令调用编辑和继续执行）的复选框。这并不能使编辑和继续执行特征失去作用，但仅能保证调试器在你没有允许时不会去调用它们。只要需要，就可以在调试器中重新编译校正的代码，方法是单击 Debug(调试器)工具栏（图 11-10）上的 Apply Code Changes（应用代码更改）按钮，或者从 Debug（调试）菜单或源程序窗口的上下文菜单中选择相应的命令。Apply Code Changes 命令仅当你对代码做某些更改时才会起作用，所

做的更改或者在调试器的源程序窗口中，或在 Visual C++ 环境之外的文本编辑器中。

重新编译代码后，执行点有时会有所改变，在这种情况下，调试器会用消息来通知你。如果指令指针不在它应该在的地方，应该用 `Set Next Statement` 命令重新进行设置。

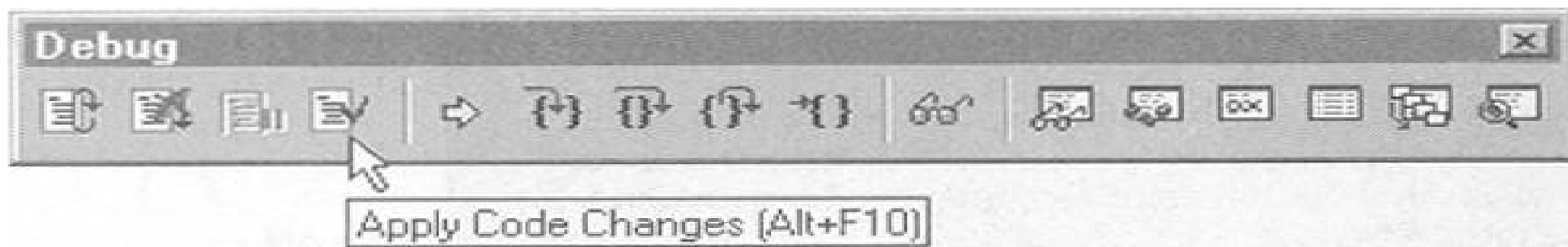


图 11-10 调用调试器的 Edit 和 Continue 特性

应当记住，`Edit and Continue` 命令有某些限制。这种特征不能识别源程序中不可能、不切实际的源代码更改，也不能识别在调试时进行编译不安全的代码更改，例如：

- 某些异常处理程序的改变。
- 函数的大量删除。

- 类和函数定义的变化。
- 静态函数的变化。
- 项目 RC 文档中的源程序数据的变化。

完成上述任何变化后，要通过 Edit and Continue 命令来恢复程序运行，调试器会在状态栏中显示一条错误消息来解释问题所在。你有权选择是使用以前的代码继续调试，还是关闭调试器，并重新编译校正后的代码。为了安全起见，Edit and Continue 命令可用来延迟对一个函数的更改，直到所调用的堆栈完全“展开”。如果你编辑在执行时被冻结的函数，并且，如果函数已被另一个函数调用，调试器在你试着恢复运行时，将显示下图所示的消息。

Edit and Continue



Cannot apply code changes to the following functions while they are on the call stack. Code changes will be available the next time these functions are called. Until the functions exit, they can be debugged only in the Disassembly window.

Function name	Thread	Call stack index
CShockWaveView::OnDraw(CD...	fffb611d	0

Reason:

New local variable requires construction or destruction

Do you still want to apply code changes?

Yes

No

这个对话框并不是一条错误消息，它仅仅提醒你，函数将要利用其初始代码来完成运行，并且，你的校正不会起任何作用，直到下一次函数执行相同的调用链。单击 Yes 按钮继续运行，单击 No 来取消 Edit and Continue 命令，并返回调试器。

断点编程

有时，做一些小的确认测验来通过程序，比在不同的位置设置断点要容易方便得多。ASSERT 宏一般用做这样的目的，如果产生错误，即终止运行，通过创建一个简单的名为 CHECK 的测试宏来编制自己的断点，以显示一条消息，如果测试失败，则终止执行。断点在宏中容易编制，利用在线汇编，可以简单地将其编码成一条 INT 3 指令。在以前的章节中，我们已经谈过，在处理器执行它时，这个指令会将控制权返回给调试器。

只有在一个条件失败时，CHECK 宏便触发 INT 3 断点。宏采用两个参数：一个表达式用来检测条件，还有一个用来解释错误的字符串指针。

```
#ifdef _DEBUG
```

```
    #define CHECK(b, s) \
```

```
        if (b) \
```

```
{ \
    ::MessageBox(NULL, s, "CHECK Error"), MB_ICONINFORMATION); \
    _asm int 3 \
}
```

`#else`

```
#define CHECK(b, s)
```

`#endif`

利用此处的宏，可以检测程序的逻辑性，如：

```
iRet = Function1();
```

```
CHECK(iRet !=1, "Bad return value from Function1");
```

如果检测失败，宏会显示错误消息，并在断点处终止程序的运行，重新设置指令指针。接着，你可以更正调试器中出现的问题，重新设置指令指针，再次通过 Edit and Continue 命令试着运行该指令。

例子：开发和调试 ShockWave 程序

本章中介绍的 ShockWave 程序提供了应用某些知识的机会。ShockWave 所能做的仅仅是显示任意色彩波动图案中的圆环。虽然它编译没有错误，但 ShockWave 不能正常运行，程序存在有两个错误，一个比较明显，另一个不太明显。使用调试器的检测工作就是让程序运行起来。

ShockWave 显示如何通过颜色层次来获得三维效果。你将需要配备一个 24 位色的视频适配器，并至少需 1Mb 的显存来观看 3-D 效果，但是，由于程序的目的是演示调试器的作用。在图 11-11 中，显示了成功调试后的程序状态。

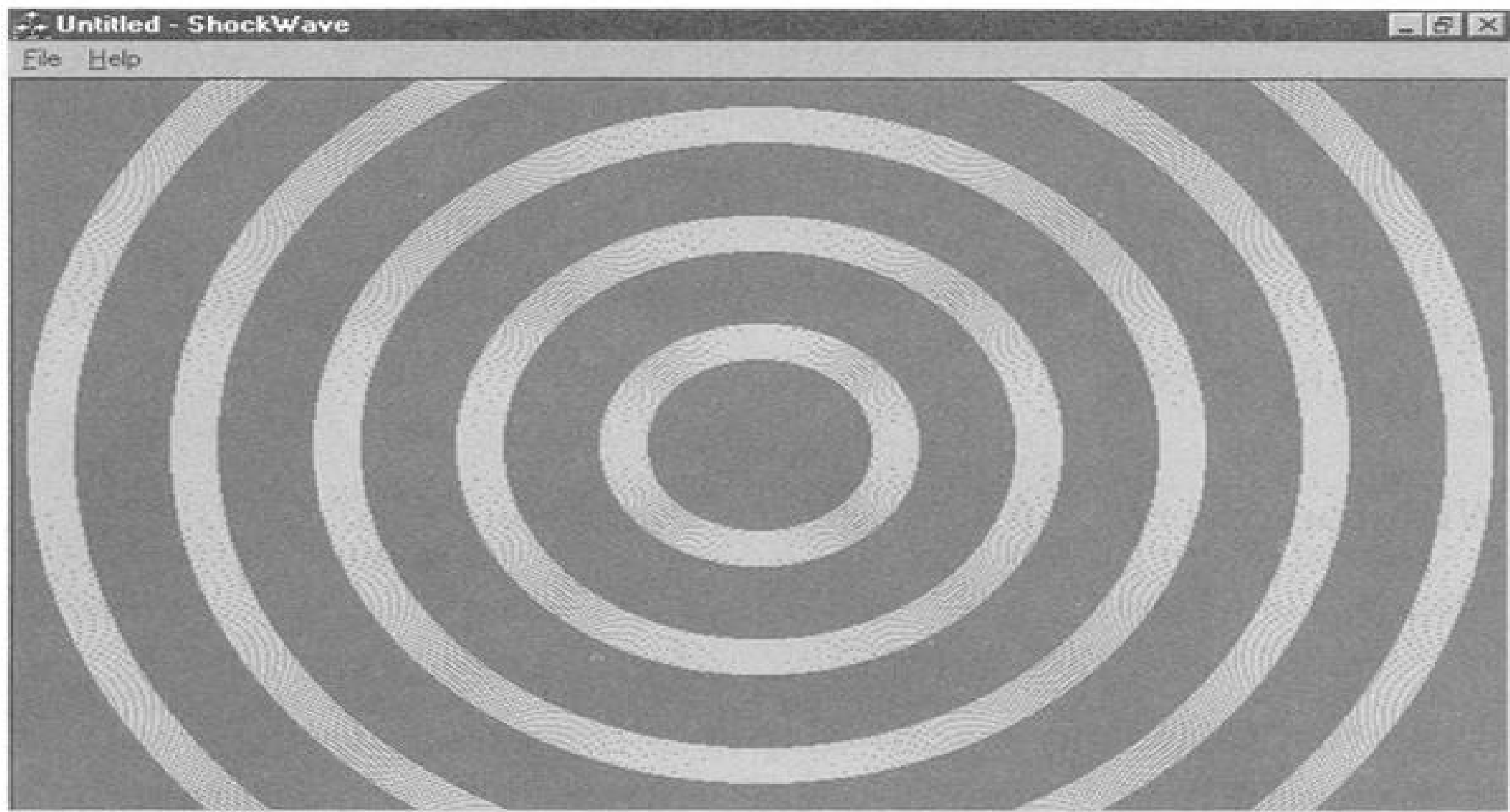


图 11-11 ShockWave 样例程序

开发 ShockWave

ShockWave 是一个 MFC 应用程序，是在 AppWizard 的帮助下创建的。你可以在从 CD 中的 Code\Chapter.11\ShockWav 的子文件夹中建立此项目。此子文件夹中的文档中含有错误的源代码，程序更改后的版本是 Shock_OK 文件夹，也可以通过接下来的 6 个步骤来开发 ShockWave 程序。程序非常简单，只要编辑它的视图类，因此，开发此程序是一个比较有趣的练习，它需要键入的代码并不多。

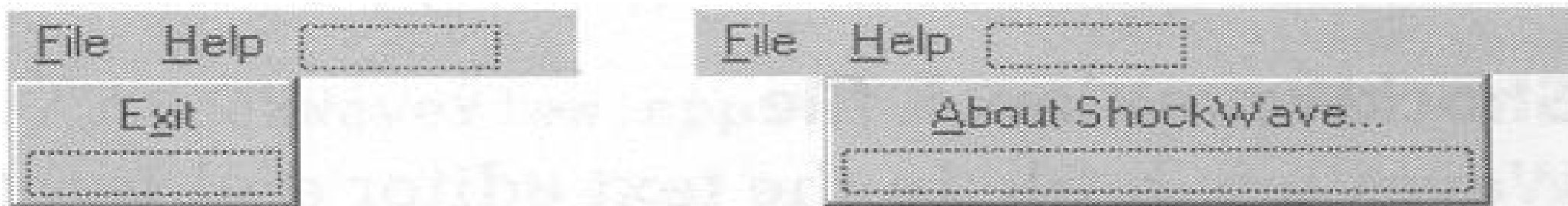
步骤 1：运行 AppWizard 来创建 ShockWave 项目

单击 Visual C++ File（文件）菜单中的 New(新建)命令，选择 Project（项目）选项卡中 MFC AppWizard（exe）图标，键入 ShockWave 作为项目名字。单击 OK 以运行 AppWizard，并创建新的 ShockWave 项目。当指定项目选项时，在 AppWizard 的步骤 1 中选择 Single Document(简单文档)按钮，并禁用的工具栏、初始状态栏和步骤 4 中的打印支持。

AppWizard 所创建的源文件名和 CD 上的略有不同，它严格控制字符的长度为 8 个字母或更少，以适应以前的文本编辑器，因为那些文本编辑器不能识别长文件名。尽管文件名不同，文件中的源代码却与这 6 个步骤中所描述的代码完全一样。

步骤 2：校正 ShockWave 的菜单

ShockWave 要求的仅仅是一个菜单命令，用以退出程序，因此，并不需要 AppWizard 添加到菜单资源中的其他命令。利用 Visual C++ 菜单编译器，校正 ShockWave 菜单，以便它仅有一个 File（文件）菜单和一个 Help（帮助）菜单。如下：



步骤 3：利用 ClassWizard 添加消息处理程序函数

ShockWave 调节波动图案的大小，以填充在客户窗口里，并适应窗口大小的变化。也可以在客户区域利用鼠标单击将新的波纹图案置于中心位置。为了响应这些事件，ShockWave 利用名字为 OnSize 和 OnLButtonDown 的处理函数，来跟踪 WM_SIZE 和 WM_LBUTTONDOWN 消息。

通过单击 View 菜单中的 ClassWizard 命令，来添加处理函数到 CShockWaveView 类。在 Message Maps（消息映射）选项卡中，设置 CShockWaveView 作为类名，并在 Message（消息）框中选择 WM_SIZE。单击 Add Function（添加函数）按

钮，以便自动创建 `OnSize` 函数，它用来处理 `WM_SIZE` 消息。对 `WM_LBUTTONDOWN` 消息做同样的工作，使之添加到 `OnLButtonDown` 函数中，如图 11-12 中所示。我们将简单地添加代码到处理函数中。在继续进行下一步操作之前，关闭 `ClassWizard` 对话框。

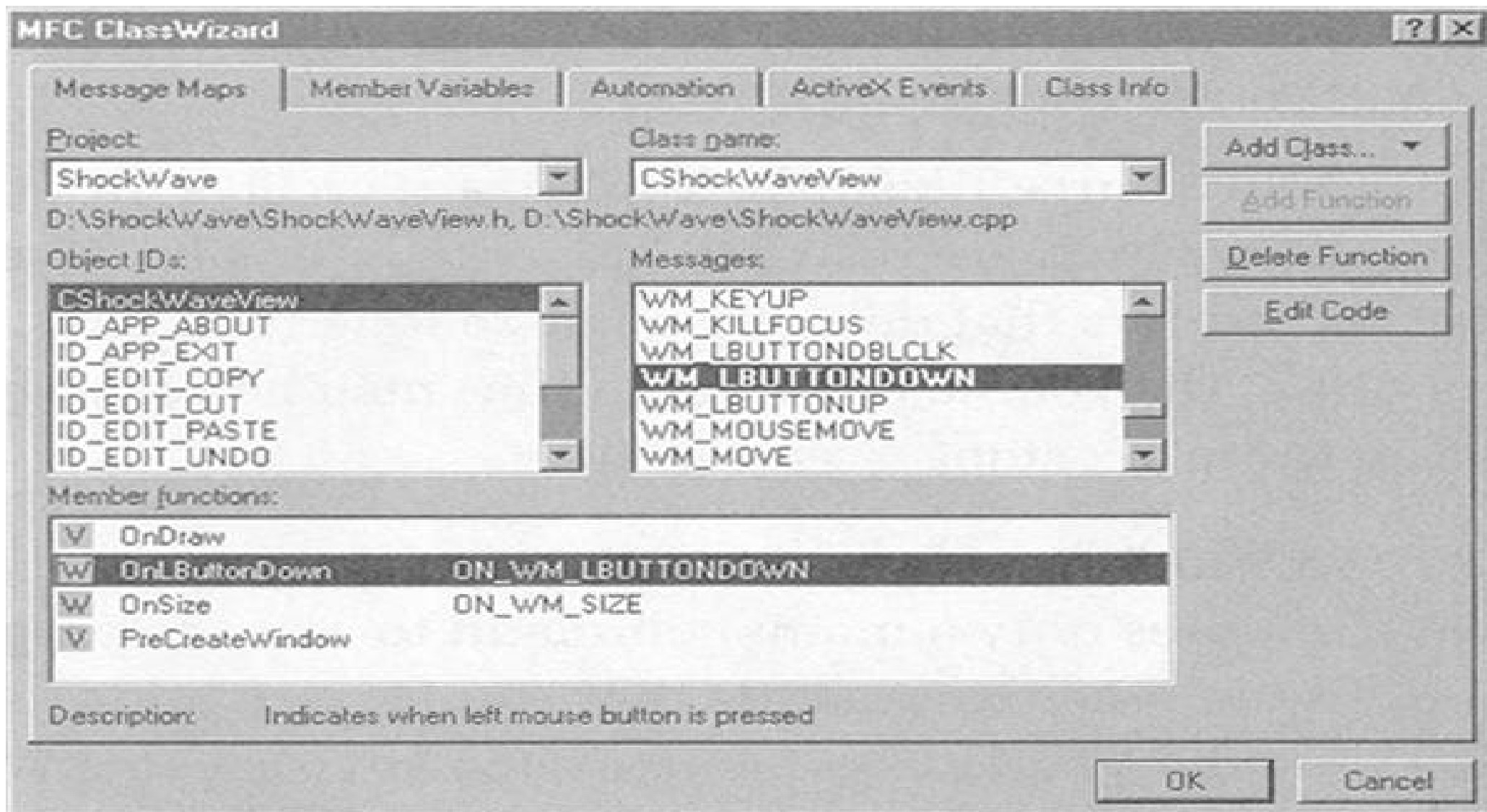


图 11-12 在 ClassWizard 中创建 OnSize 和 OnLButtonDown 处理函数

步骤 4：编辑 ShockWaveView.h 文档

打开文本编辑器中的 ShockWaveView.h 文件，并定位到 CShockWaveView 类声明处（在配套 CD 中，此文件名为 ShockVw.h），添加阴影代码如下：

```
// ShockWaveView.h : interface of the CShockWaveView class
```

```
//
```

```
////////////////////////////////////
```

```
#define    NUM_COLORS    6
```

```
#define    RED            0
```

```
#define    GREEN          1
```

```
#define    BLUE           2
```

```
#define    CYAN           3
```

```
#define    MAGENTA        4
```

```
#define GRAY 5
```

```
class CShockWaveView : public CView
```

```
{
```

```
private:
```

```
    CPoint center;
```

```
    CRect rectClient;
```

```
    COLORREF rgb[NUM_COLORS];
```

```
    int iColor;
```

```
protected: // create from serialization only
```

```
    CShockWaveView();
```

```
DECLARE_DYNCREATE(CShockWaveView)
```

```
.
```

```
.
```

```
.
```

步骤 5：编辑 ShockWaveView.cpp 文件

ShockWaveView.cpp 文件（或 CD 中的 ShockVw.cpp 文件）含有所有针对 CShockWaveView 类的实现的详细信息。它也要求一些附加信息，在带有阴影的行中有所显示。WizardBar 提供了在文本编辑器中打开文件的简单方法。因为文本编辑器显示了当前类的头文件，单击 WizardBar 中的棍棒图标，打开 CPP 文件。可以通过单击相同的工具立即返回到头文件。

接下来列出的程序显示了整个 ShockWaveView.cpp 文件中的源代码。在每条重要的函数之后增加了注释，以解释所发生的一切。

```
// ShockWave View.cpp : implementation of the CShockWaveView class
```

```
//
```



```
#include "stdafx.h"
```

```
#include "ShockWave.h"
```

```
#include "ShockWaveDoc.h"
```

```
#include "ShockWave View.h"
```

```
#include <math.h>
```

```
#ifdef _DEBUG
```

```
#define new DEBUG_NEW
```

```
#undef  THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
```

```
#endif
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
// CShockWaveView
```

```
IMPLEMENT_DYNCREATE(CShockWaveView, CView)
```

```
BEGIN_MESSAGE_MAP(CShockWaveView, CView)
```

```
   //{{AFX_MSG_MAP(CShockWaveView)
```

```
        ON_WM_SIZE()
```

```
        ON_WM_LBUTTONDOWN()
```

```
   //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
```

```
////////////////////////////////////////////////////////////////
```

```
// CShockWaveView construction/destruction
```

```
CShockWaveView::CShockWaveView()
```

```
{
```

```
    SYSTEMTIME st;
```

```
    ::GetSystemTime( &st );
```

```
    srand( (int) st.wMilliseconds );           // Seed the random number
```

```
    rgb[0] = RGB( 128,    0,    0 );           // Red
```

```
    rgb[1] = RGB(    0, 128,    0 );           // Green
```

```
    rgb[2] = RGB(    0,    0, 128 );           // Blue
```

```
    rgb[3] = RGB(    0, 128, 128 );           // Cyan

    rgb[4] = RGB( 128,    0, 128 );           // Magenta

    rgb[5] = RGB( 128, 128, 128 );           // Gray

    iColor = 1;                               // Start with Green
```

```
}
```

```
CShockWaveView::~CShockWaveView()
```

```
{
```

```
}
```

CShockWaveView 构造器用 6 种色彩的 COLOR_REF 值初始化 rgb 数组。当显示波动图案时，CShockWave 随机地选择其中的一个色彩。在变量 iColor 中，容纳 rgb 中决定的程序用来绘制波动图案的颜色种类的下标值。

构造器调用 Windows API 函数 `GetSystemTime` 来检索当前系统时间的 `ms` 组件。这个值的范围是 0 到 999，并为 `srand` 函数（C 运行时间随机数字发生器）提供一个便利的种子值。

```
BOOL CShockWaveView::PreCreateWindow(CREATESTRUCT& cs)
```

```
{
```

```
    HCURSOR hCur = ::LoadCursor( NULL, IDC_CROSS );
```

```
    cs.lpszClass = AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW, hCur, NULL );
```

```
    ::DeleteObject( hCur );
```

```
    return TRUE;
```

```
}
```

ShockWave 显示的鼠标光标是十字形，而不是一般的箭头形状。利用这种轻微的变化，用户可以更明确客户区域中的目标是什么，然后单击它。ShockWave 使用

单击坐标作为下一个波动图案的中心。为了改变窗口光标的形状，CShockWaveView 使 PreCreateWindow 函数无效。MFC 框架在它创建程序的主窗口之前，调用 PreCreateWindow 函数，将函数传递给 CREATESTRUCT 结构。此结构含有 MFC 计划使用窗口的设置。使 PreCreateWindow 函数无效给程序提供了一个机会，在 MFC 创建窗口之前更改窗口特征，诸如它的光标形状。

在 CShockWaveView 的实现中，PreCreateWindow 首先加载标准的 Windows 十字形光标，它拥有一个 IDC_CROSS 标识值。函数接着注册一个新的窗口类，给它指定一个十字形光标，以及具有 NULL 值的背景画笔。NULL 背景颜色标志着在操作系统重新调整窗口大小时不能重新绘制窗口背景。因为窗口颜色可随每一个新的波纹而随机变化，ShockWave 本身负有绘制背景的任务。

你可以重新调用第 5 章中的 Color 样例程序来绘制自己的背景。Color 程序不能像 ShockWave 那样调整窗口创建标志，但是，可以捕获 WM_ERASEBKGD 消息，以防止操作系统绘制窗口。以下的两个程序说明了两种不同的技术可以获得相同的结果：

```
////////////////////////////////////
```

```
// CShockWaveView drawing
```

```
#define PI          3.1415926
```

```
#define NUM_RINGS 5
```

```
void CShockWaveView::OnDraw(CDC* pDC)
```

```
{
```

```
    CPen      pen;
```

```
    CRect      rect;
```

```
    COLORREF   color;
```

```
    int        i, j, iPeriod;
```

```
    double     Angle;
```

```
    // Set up coordinate system for largest wave
```

```
i = min( rectClient.right, rectClient.bottom );    // i = diameter
```

```
pDC->SetMapMode( MM_ISOTROPIC );
```

```
pDC->SetWindowExt( i, i );
```

```
pDC->SetViewportExt( rectClient.right, -rectClient.bottom );
```

```
pDC->SetViewportOrg( center.x, center.y );
```

```
i = max( rectClient.right, rectClient.bottom )/2; // i = radius
```

```
rect.SetRect( -i, -i, i, i );
```

```
iPeriod = i/(2*NUM_RINGS);
```



```
// Two loops: loop 1 displays one wave per iteration

//          loop 2 draws one color gradation per iteration


for (j=0; j < NUM_RINGS; j++)

{

    for (Angle=0.0, i=1; i < iPeriod; i++)

    {

        Angle += PI/iPeriod;

        color  = 128 + (DWORD)(128.0 * sin( Angle ));

        if (color > 255)

            color = 255;
```

```
switch (iColor)

{

    case GREEN:

        color <<= 8;

        break;


    case BLUE:

        color <<= 16;

        break;


    case CYAN:

        color = RGB( 0, (int) color, (int) color );
```

```
break;
```

```
case M A G E N T A:
```

```
color = RGB( (int) color, 0, (int) color );
```

```
break;
```

```
case G R A Y:
```

```
color = RGB( (int) color, (int) color,
```

```
            (int) color );
```

```
break;
```

```
}
```

```
rect.InflateRect( -1, -1 );

pen.CreatePen( PS_SOLID, 1, color );

pDC->SelectObject( &pen );

pDC->Ellipse( rect );

pDC->SelectStockObject( BLACK_PEN );

pen.DeleteObject();

}

rect.InflateRect( -iPeriod, -iPeriod );

}
```

```
}
```

OnDraw 函数承担了显示波动图案的整个任务，在一系列小的同心圆中画出每道波纹。每个圆具有一个像素宽，从波纹的外部边缘开始，然后向中央作用（由于

这个原因，波动图案出现在屏幕上时，看上去似乎是向内扩展而不是向外扩展)。对每一个像素圆，代码只会在当前颜色的强度（或亮度）上有轻微的增加或减弱。强度的递减使波纹明显具有三维效果。

此函数在两个循环中画波动图案，一个循环嵌套在另一个里面。外部循环重复 5 次，一次重复即可画出一个完整的波纹。内部循环每重复一次，便画出一个像素宽度的圆，直到画完足够的圆，形成整个波纹。对每个圆，内循环创建一个新的设备上下文画笔，采用 iColor 确定的当前色彩，轻微调整颜色的强度。在每循环一次后，强度从中间亮度值 128 变化到最大值 255，在一个代码行中体现了这种变化：

```
color=128+(DWORD)(128.0*sin(Angle));
```

由于窗口背景的颜色强度值为 128，每个波纹似乎都是以正弦曲线从背景中产生。

```
////////////////////////////////////
```

```
// CShockWaveView message handlers
```

```
void CShockWaveView::OnSize(UINT nType, int cx, int cy)
```

```
{  
  
    CView::OnSize(nType, cx, cy);
```

```
    rectClient.SetRect( 0, 0, cx, cy );  
  
    center.x = cx/2;                // Center of shock wave  
  
    center.y = cy/2;
```

```
}
```

```
void CShockWaveView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{  
  
    CView::OnLButtonDown(nFlags, point);
```

```
center = point;  
  
iColor = rand();  
  
Invalidate( FALSE );
```

```
}
```

这个练习中的步骤 3 使用 ClassWizard 来为 WM_SIZE 和 WM_LBUTTONDOWN 消息添加处理程序函数。在这里，我们将代码添加到 ClassWizard 创建的存根函数中。

当窗口大小发生变化时，OnSize 处理程序将波纹图案置于客户窗口的中心，并在 rectClient 中记录新窗口的尺寸。OnDraw 函数利用这些尺寸，以保证波动图案填充这个窗口。OnLButtonDown 函数记录鼠标光标的坐标，它将确定下一波形的中心。函数也可从 rgb 数组的 6 个变量中随机选择一个新的颜色，并调用 Invalidate，以便窗口用新的波动图案重绘。

步骤 6 ：建立并运行 ShockWave.exe 程序

确定 Build 工具栏显示 Win32 Debug 作为当前的程序配置：



单击工具栏中的 Build 按钮（或者选择 Build（建立）菜单中的命令），以创建一个 ShockWave 的调试版本，然后单击 Execute（执行）命令，以便运行程序。

调试 ShockWave

第一次运行 ShockWave 时，第一个错误是显而易见的（但不是致命的）。客户窗口似乎是透明的，Visual C++ 中的工具栏和文本出现在 ShockWave 的窗口里。你可能已经明白了在哪里出的错，但是，让我们用调试器来调试程序，以弄清楚问题产生的原因，利用 Exit（退出）命令关闭有错误的 ShockWave 程序。

开始调试工作时，利用文本编辑器来寻找 ShockWaveView.ccp（或 ShockVw.ccp）文档。错误最有可能在视图类中产生，因为它含有要求大量变换的部分源代码。由于 ShockWave 窗口中存在一些错误，所以，应该怀疑开始的 PreCreateWindow 和 OnDraw 函数。这两个函数在这个练习的第 5 步已经校正。第 1 个函数设置窗口特性，第 2 个函数设置窗口内容。

单击 Edit（编辑）菜单中的 Breakpoint（断点）命令，在 Breakpoint（断点）对话框的 Location（位置）选项卡中键入 CShockWaveView::PreCreateWindow，按

Enter 键，并在同一地方键入 CShockWaveView:: OnDraw，这可以将位置断点设置在每个不可信函数的开始处，单击 OK 按钮返回编译器。

现在按 F5 启动调试器，硬盘的活动表示 Visual C++正在启动调试器，接着运行 ShockWave.exe 文件。调试器源程序窗口接着显示指令箭头指向 CShockWaveView:: PreCreateWindow 函数的第一行。ShockWave 程序在已设置的两个位置断点处的第一个位置停下来。Visual C++窗口现在看上去有点像图 11-13。

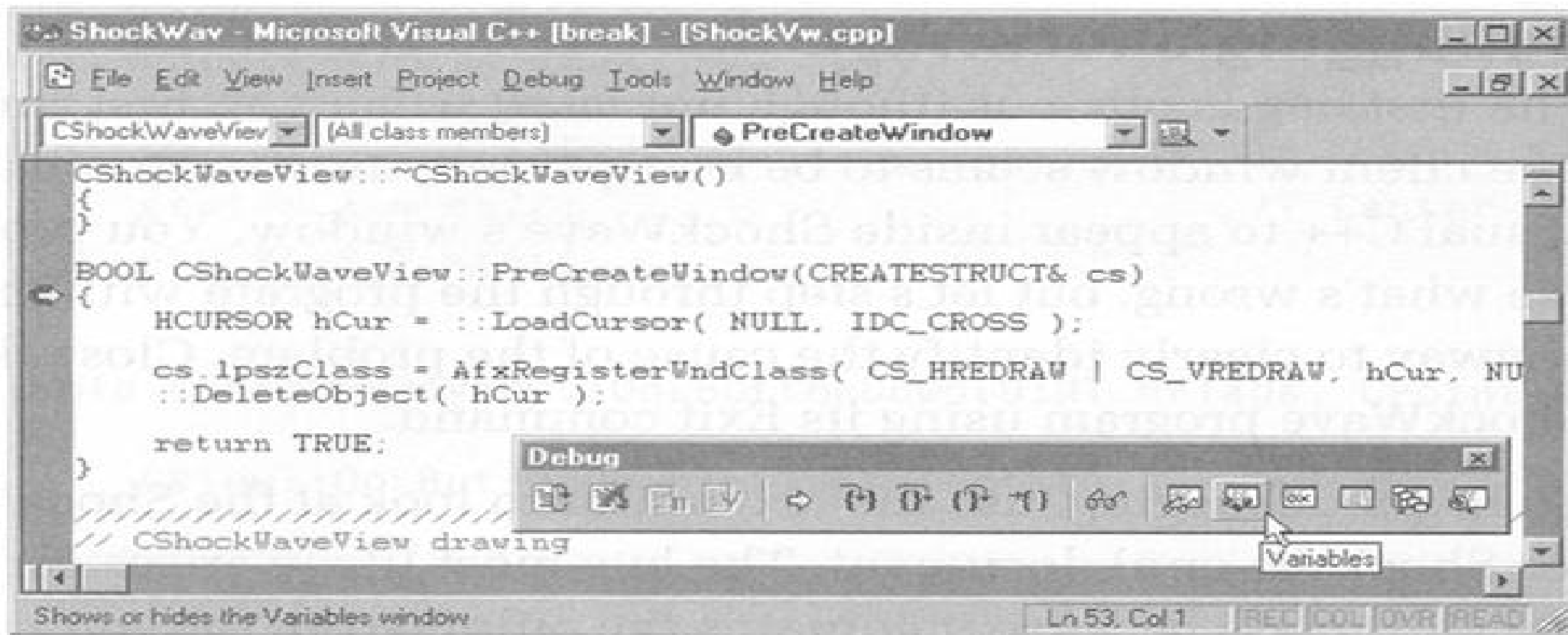


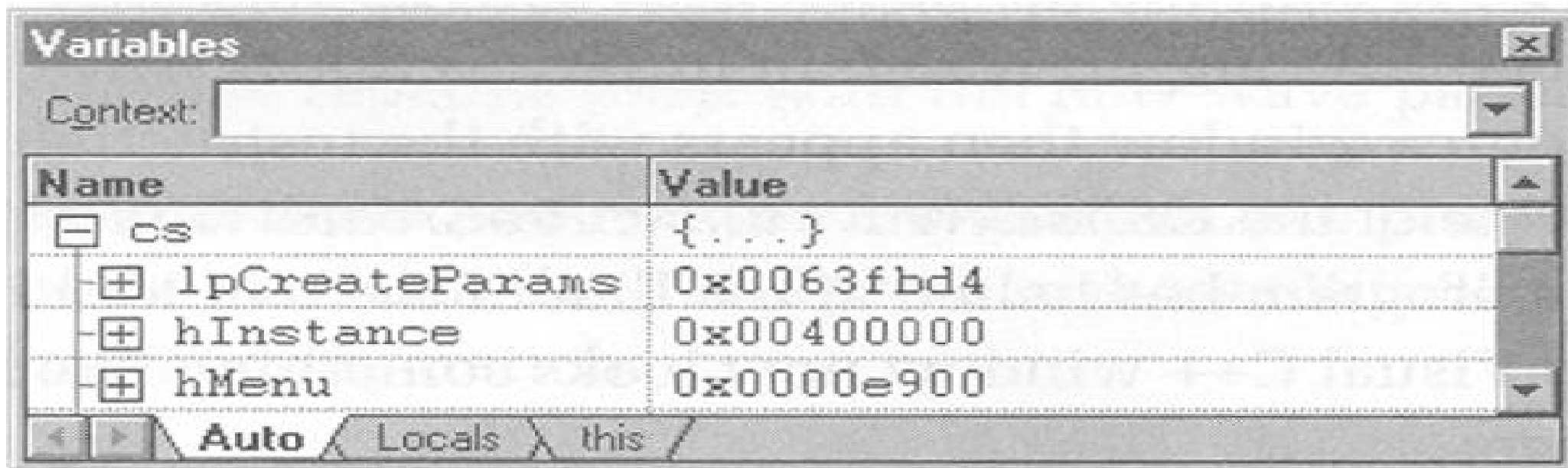
图 11-13 ShockWave 程序在 Visual C++ 调试器中的断点处停止

PreCreateWindow 函数内幕

单击图 11-13 所示的 Debug 工具栏中的 Variables（变量）工具，以显现 Variables（变量）窗口。Variables（变量）窗口列出了最后执行行中引用的变量，在此情况下，是通过函数序言来访问变量 cs。cs 变量指向 MFC 用来创建 ShockWave 窗

口的 CREATESTRUCT 结构。在 Variables（变量）窗口中临近 cs 名字的地方，单击小的加号（+）按钮，以扩大显示结构成员变量。

如果一些变量名太长，不能填充在栏目里，可以使 Variables（变量）窗口里的 Name 栏调宽一些。将鼠标光标置于两个栏目标签之间的分隔线上，刚好在 Value（值）标签的左边，并左右拖动分隔线，重新调整栏宽。双击分隔线将自动调整栏宽，以便在栏中容纳最长的变量名。



按 F10 键或者单击 Step Over（整体单步调试）工具按钮来执行函数序言代码。指针在 PreCreateWindow 函数的下一行处停止。

```
HCURSOR hCur=::LoadCursor(NULL, IDC_CROSS);
```

Variables（变量）窗口现在包括目前的 hCur 值，但是因为指令还没有执行，窗口中显示的值是没有意义的，整体单步执行此指令，赋予 hCur 返回给 LoadCursor API 函数的值。新值以红色出现，来指示最后一条指令已经改变 hCur 的值。颜色代码是一些调试器窗口的优秀特征，可以让你迅速弄清楚指令改变后的变量值。

按 F10 再次调用 `AfxRegisterWndClass`:

```
cs.lpszClass= AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW,hCur,NULL);
```

Variables (变量) 窗口显示, `cs.lpszClass` 现在指向一个有效类名。有时有点类似被 MFC 框架所指定的 `Afx:400000:3:13ce:0:0`。`cs.lpszClass` 的新值可以确定 `AfxRegisterWndClass` 已被正确运行。这不值得惊奇, 因为操作系统正确创建了 ShockWave 窗口, 类必须正确注册, 窗口绘制, 而不是窗口创建, 这是一个难题, 因此错误必定在 `OnDraw` 函数里。按下 F5 键, 或者从 Debug (调试) 菜单中选择 Go (转到) 来移向下一断点, 同时仔细观察屏幕。

OnDraw 函数内幕

继续运行, 直到程序流到达下一个早先在 `OnDraw` 函数里设置的断点, 为了理解程序的这一点, 可以看见 ShockWave 窗口闪烁, 然后消失, ShockWave 窗口仍然存在, 但在重新获得控制权时, 调试器窗口将会覆盖它。通过最小化 Visual C++ 可以展现 ShockWave 窗口, 注意 ShockWave 已完全处于不活动状态, 它甚至没有菜单栏。在此时, 控制权属于调试器。

返回到调试器, 并重复按 F10 键, 以便单步通过数据声明, 向下到此段代码:

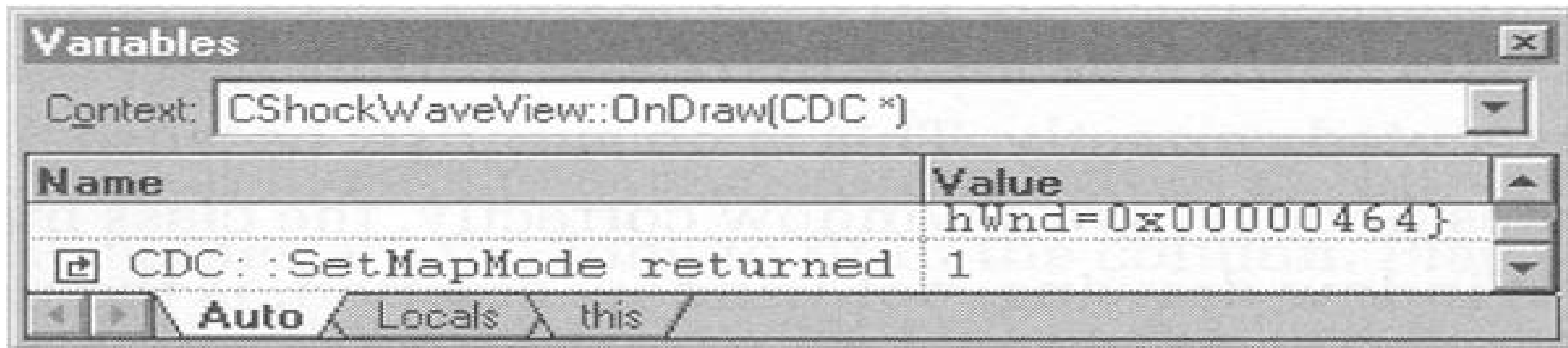
```
pDC->SetMapMode( MM_ISOTROPIC );
```

```
pDC->SetWindowExt( i, i );
```

```
pDC->SetViewportExt( rectClient.right, -rectClient.bottom );
```

```
pDC->SetViewportOrg( center.x, center.y );
```

第一行设置映射模式为 `MM_ISOTROPIC`，确保波形以圆形显示在屏幕上，而不是椭圆。接下来的两行设置窗口范围和覆盖整个 `ShockWave` 客户区域的视口。最后一行在窗口的中央设置视口原点。像许多 MFC 函数一样，这些 CDC 成员函数在成功时返回一个正值，或返回 `NULL` 以指出某一问题。这些函数失败是不大可能的，因此，它们不可能保证利用其他检查返回值的代码来使程序混乱。即使 `ShockWave` 没有存储函数返回值，也可以观察 `Variables`（变量）窗口里的这些值，以便确保函数正确运行。每次整体调试一个函数时，窗口会显示一个返回值，如下图所示：



如果一个函数返回错误代码，当你调试它时，调试器能将此返回值转换成所需要的有用消息。在 Watch（观察）窗口，双击 Name（名字）栏内带点的条目框，并键入 err,hr，在决定这个条目的值时，调试器调用 GetLastError API 函数，并将结果转换成有用的文本，诸如“The handle is invalid（此句柄无效）”。

Registers（寄存器）窗口也可提供另一种检查函数返回值的方法。在 Intel 处理器中，一个 Win32 函数在退出之前，将一个返回值放在 EAX 寄存器中（64 位返回值占据了 EDX:EAX 寄存器对）。为了检查函数的返回值，在单步进行整体函数调用后，立即浏览一下 Registers（寄存器）窗口中的 EAX，像 Variables（变量）窗口一样，Registers（寄存器）窗口用红色显示新值，以显示寄存器在最后一指令的变化。由于以上函数返回来的值都不是 0，因此，可以得知本段代码运行正确。在这点上，ShockWave 在绘制波纹图案的两个循环的开始做好准备。但是，有时也会出错，程序不应该画出圆的波纹，因为 ShockWave 客户区域的背景依然没

有重新绘制。

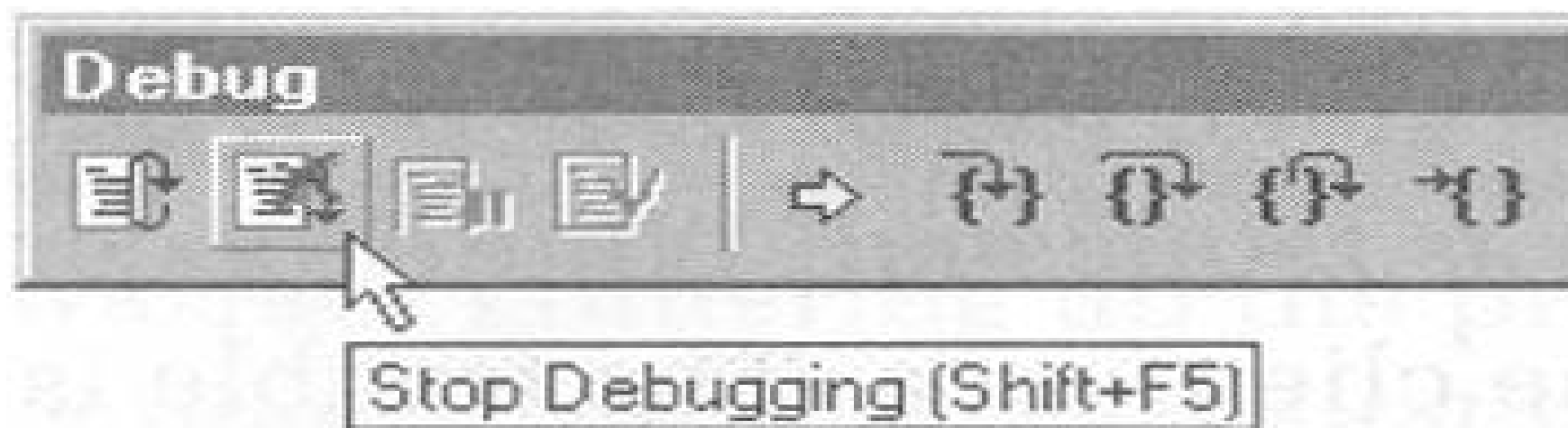
重新调用第 5 步中校正过的 `PreCreateWindow` 函数,指示 Windows 不要重新绘制 `ShockWave` 的背景。这也是在注册窗口类时, `NULL` 画笔值赋给 `AfxRegisterWndClass` 的目的。因此,按照要求,Windows 正确建立了窗口,而无须填充客户区域。问题在于,`ShockWave` 不这样做,必须有人来重新绘制窗口背景,如果不是系统,`ShockWave` 就必须自己去做。解决第一个错误的办法就是:`ShockWave` 需要在画出波动图案之前绘制窗口背景。

在这里,我们可以有选择地修正源代码,重新恢复执行 `Edit and Continue`。但是 `Call Stack` (调用堆栈) 窗口显示了,`OnDraw` 函数是几个嵌套函数行中的最后一个,这些调用操纵内核和 MFC 框架。这意味着所有的更改不会马上起作用,仅当函数下次执行时才有效。因此,可以终止调试器并返回编译器。

这将引出一个比较有趣的情况,你可以假定,继续 `ShockWave` 的运行是终止调试的一种方法,而且这种方法是经过三思的。可以利用 `Exit` (退出) 命令正常退出 `ShockWave`,并且调试器将会停止。返回到文本编辑器,好,试一试它,按下 `F5` 继续运行 `ShockWave`。

这样,你永远也不可能到达 `ShockWave` 菜单,因为每次按下 `F5` 键,在退后到 `CShockWaveView::OnDraw` 函数里的断点之前,`ShockWave` 窗口只是简短地出现一下。要弄清楚发生了什么并不困难。当 `ShockWave` 重新获得焦点时,它将出现在 `Visual C++` 的顶部以及屏幕上所有其他窗口之上。Windows 将 `WM_PAINT`

消息传送给 ShockWave，告诉它需要重新绘制。但在重新绘制时，框架需要调用



ShockWave 的 OnDraw 函数触发断点。调试器然后获得焦点，Visual C++ 就在 ShockWave 窗口上显示自己。每次你按下 F5 来继续执行 ShockWave 时，进程便重复无限的循环。可以通过在按下 F5 键之前删除或禁用断点来终止循环。但是，Debug（调试）工具栏中的 Stop Debug（停止调试）按钮（或 Debug（调试）菜单中与其等同的命令）可以提供更好的终止调试器的办法。

这个命令将会使你重新返回到编译器，离开所有的断点。

校正和重建 ShockWave

绘制窗口背景并不要求太多的代码，在文本编辑器中，添加如下带有灰色的程序行，以使 OnDraw 函数成为如下的形式：

```
void CShockWaveView::OnDraw(CDC* pDC)

{

    CPen        pen;

    CRect        rect;

    COLORREF     color;

    int          i, j, iPeriod;

    double       Angle;

    CBrush       brush;

    // Paint client area with current color

    brush.CreateSolidBrush( rgb[iColor] );

    pDC->FillRect( rectClient, &brush );

    pDC->SelectStockObject( NULL_BRUSH );
```

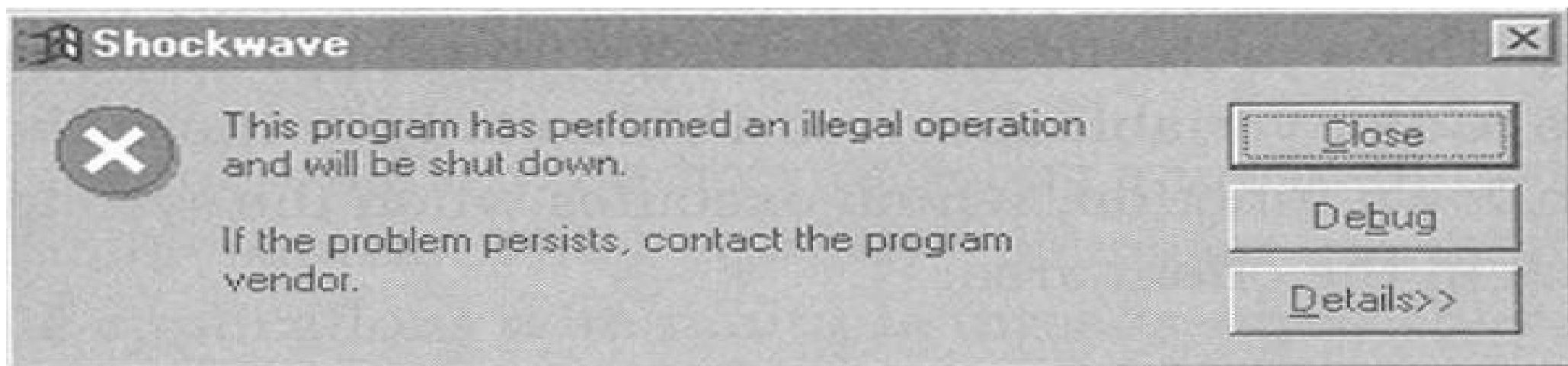
```
brush.DeleteObject();
```

```
.  
.br/>.br/.
```

灰色行给画笔分配当前被 iColor 指出的颜色，利用它绘制客户区域，接着，删除画刷，在此处使用新的代码，再建立 ShockWave 的调试版本，并使用 Build（建立）菜单的 Execute（执行）命令运行它。这次将会正确显示出来，其背景用中间色度的绿色来绘制。

第二个错误

ShockWave 还存在第二个错误，第二个错误比第一个错误更有趣，因为它说明了在调试器没有运行时，Visual C++ 怎样让你发现程序错误。为了弄清楚第二个错误，在 ShockWave 窗口中的任意地方单击鼠标。按照程序设定，这种行为不会清除窗口，用 6 个可用颜色中的一个来重绘窗口，在鼠标单击的坐标中心重绘波动图案。你可以多击几次鼠标，但最终 Windows 将会显示如下消息：



在某些地方，ShockWave 试图访问不属于它的内存，当这些在程序的发行版本中发生时，你将别无选择，只好单击 Close（关闭）按钮来终止程序，建立一个等同的调试版本，再次启动调试器，并重复错误的产生情况。但以上消息提供了另一种选择，如果你单击 Debug（调试）按钮，Windows 将会自动重新启动调试器，甚至即使在 Visual C++ 没有正确运行的情况下。你将会发现，自己在查看 ShockWave 程序，因为它在错误之后立即出现。没有必要猜测哪一行引起了保护错误，因为黄色的指令指针箭头正指向它。Microsoft 将此特征称作即时调试。

根据源程序窗口显示，程序在 CShockWaveView:: OnDraw 函数里的第一条指令处运行失败：

```
brush.CreateSolidBrush(rgb[iColor]);
```

这是我们刚刚添加的一行，它建立了一个画刷，以便 OnDraw 绘制 ShockWave 客户区域的背景。变量 iColor 中是 rgb 数组的下标，如下所示，它在 ShockWaveView.h 中已经声明了。

```
#define NUM_COLORS 6
```

```
.  
.br/>.br/.
```

```
COLORREF rgb[NUM_COLORS];
```

iColor 的当前值决定了用于背景画刷使用的颜色。在 Variables（变量）窗口中查看一下 iColor，它的值应该为 6 或更多，这意味着画刷目前的颜色是 rgb 数组中的第 iColor 个元素。

这里存在一个问题，给 iColor 赋予一个比 5 大的值，这意味着程序试图访问 rgb 数组中的一个元素，而这个元素并不存在。很明显，这是保护错误。我们已经找到了错误，但是不知道是什么原因引起的。iColor 变量所接收的值仅仅是 CShockWaveView:: OnLButtonDown 函数里的，当系统在客户区域检测鼠标时，它便开始运行。

```
void CShockWaveView::OnLButtonDown(UINT nFlags, CPoint point)

{

    CView::OnLButtonDown(nFlags, point);

    center = point;

    iColor = rand();

    Invalidate( FALSE );

}
```

此行：

```
iColor=rand ( )；
```

指定 iColor 为一个随机数，是从 rand 函数中得到的。这个 C 运行函数返回的值范围是从 0 到 RAND_MAX，Stdlib.h 头文件定义为 0x7FFF 或 32767。iColor 会以这样高的值结束就不奇怪了。我们需要保证 iColor 的值永远不会超过 rgb 数组中的元素数目，以便 OnDraw 函数只访问有效的颜色元素。可以利用下行语句，通过重置错误行来限制 iColor 的值。

```
iColor=rand() % NUM_COLORS;
```

这样就解决了第二个错误，方法是限制 iColor 的值为从 0 到 5 的范围。如果你重建了 ShockWave，并再次通过 Execute 命令执行了它，将会看到程序的运行将如愿以偿。

特殊的调试情况

Win32 程序可以完成许多任务，并且，在本章中所述的这些简单例子中，几乎不可能直接应用于你自己的程序中。这就是为什么我将尽量把重心集中在技术上，而不是细节上的原因，无论你的 Win32 应用程序多么复杂，或者多么不正常，也要有自信心，Visual C++ 调试器能给你提供帮助。

这里说明如何利用先进的 Win32 特征来调试程序。Visual C++ 调试器能截取异常情况。用多线程处理应用程序，调试 ActiveX 客户和服务端应用程序，所有这些都能快速完成。当被调试的控制程序在一台计算机上运行时，调试器还能在另一台计算机上运行。

调试异常

利用 C++ 异常处理机制，当意想不到的错误发生时，程序可以保留控制权。当函

数检测到一个错误时，它通过调用 `throw` 关键字来通知异常处理程序，异常处理程序利用 `catch` 来获得通知。如果没有捕获异常情况的处理程序，调试器会通知你没有捕获异常。C 程序也能执行结构化异常处理，它用 `_try` 和 `_except` 语句，而不是 `throw` 和 `catch` 语句。

在图 11-14 中显示了 Exceptions（异常）对话框，可以从中指定调试器应该怎样处理每种类型的异常情况。通过单击 Debug 菜单上的 Exceptions（异常）命令来调出此对话框。你可以为程序中可能发生的每种异常情况设置两个选项中的一项：Stop Always（永远停止）或 Stop If Not Handled（如果不处理便停止）。

如果为一种异常指定 Stop If Not Handled，当异常出现时，调试器将一条消息写到 Output（输出）窗口中，但不会终止程序，或用一个对话框来通知你，除非异常处理程序解决异常时失败。此时修改问题或检查源代码以弄清楚异常发生的位置就太晚了，因为程序已经出现异常，并正在运行异常处理程序。



图 11-14 Exceptions (异常) 对话框

如果为一种异常情况指定 Stop Always, 你会对异常情况的处理具有更多的控制权。当异常情况产生时, 调试器将会立即终止程序, 更新源程序窗口, 以显示错误指令, 并在异常情况处理函数获得控制权之前通知你。有些情况下, 可以在 Variables (变量) 窗口中更改一些错误的变量来处理异常情况。如果你接着按下 F5 来继续运行程序, 对话框将会询问你, 是否想忽略异常, 返回到程序异常处理函数。如果你修正了问题, 单击 No 按钮。否则, 单击 Yes 按钮, 以便将控制

权传递给异常处理程序。如果异常处理程序不能修正问题，调试器会终止程序，并再次通知，就像你已经选择了 Stop If Not Handled 一样。因为 Stop Always 选项使用处理器的调试寄存器，对没有调试寄存器的处理器，这个选项不合适。

显示在图 11-14 中的 Exceptions 列表框含有一个系统异常的默认列表。可以添加或从列表中删除异常，Visual C++ 在项目的 OPT 文件中能保存新的列表。对于不在列表中的异常，调试器的处理像对待 Stop If Not Handled 异常那样。每个异常有一个唯一的数字，系统异常利用 EXCEPTION 前缀在 Winbase.h 头文件中定义，诸如 EXCEPTION_ACCESS_VIOLATION。

为了添加新的异常到异常列表中，调用 Exceptions (异常) 对话框，并在 Number 控件中键入异常号，在 Name 控件中键入异常情况名。单击 Stop Always 或 Stop If Not Handled 按钮，接着单击 Add 按钮。为了删除异常情况，从 Exceptions 列表中选择它，并单击 Remove (删除) 按钮。如果你改变主意，并想恢复所有删除的系统异常，单击 Reset (重置)，如果为异常情况改变选项，诸如它的名字，单击 Change (改变) 按钮来使更改成为永久的。

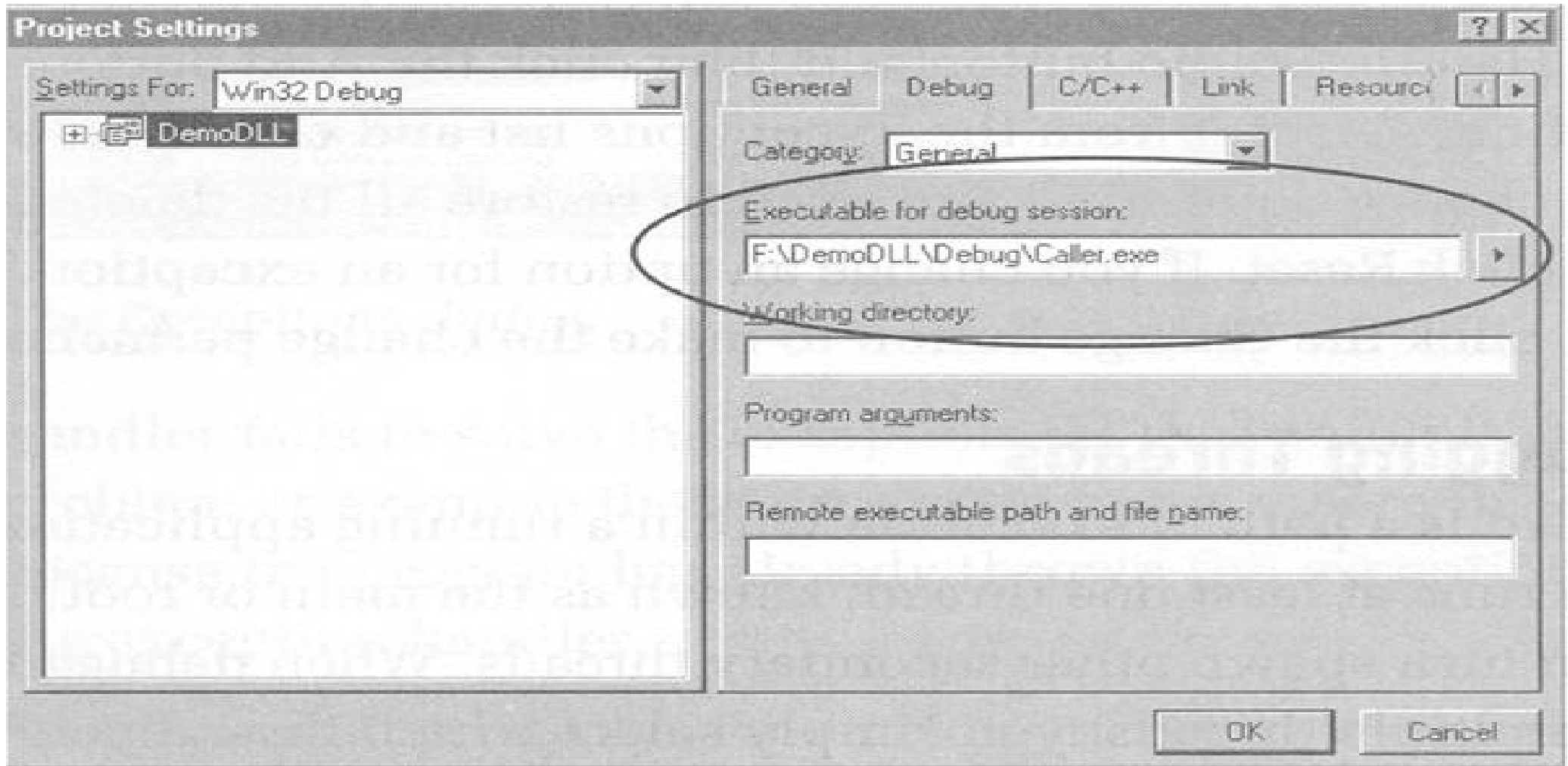
调试线程

在一个运行的应用程序中，一个线程就是一条运行路径。每个应用程序至少运行一个线程，即主线程或根线程，它也可能生成其他线程。当调试具有多个线程的程序时，你可以选择要调试哪个线程，并按照其执行流进行调试。

只有在调试器开始执行后，才可以选择一个要调试的线程。首先，在需要的地方设置一个断点，当执行程序在断点处停下来时，通过断点的所有线程均被挂起。单击 Debug 菜单的 Threads（线程）来调出 Threads 对话框，从线程列表选出所需要的线程，并单击 Set Focus（设置焦点）按钮。当你继续单步调试程序时，调试器沿着具有焦点的线程进行调试。为了防止别的线程运行同样的代码，在 Threads（线程）对话框中，将其他线程挂起。后面，你可以恢复被挂起的程序，方法是在同样的对话框中选择它，并单击 Resume（恢复）按钮。

调试动态链接库

在 Visual C++中调试动态链接库与调试普通的应用程序没有什么区别，不同的只是调试器启动库的调用程序，并没有加载 DLL 文件本身。当调用程序需要库时，操作系统才会照管加载的库。当控制到达库代码中的断点时，所运行的调用程序和 DLL 均被挂起。在调试动态链接库中，唯一额外的步骤就是标识所调用的应用程序，以便调试器能运行它。显示 Project Settings(项目设置)对话框中的 Debug（调试）选项卡，方法是选择 Project（项目）菜单中的 Settings（设置）命令，然后键入或浏览所调用应用程序的文件名和路径。



如果离开 Executable For Debug Session（调试会话的可执行程序）对话框，当你开始调试动态链接库时，调试器会提示文件名。在线帮助建议你在 Category（分类）框中选择 Additional DLL（附加的动态链接库），在 Local Name（局部名称）

栏中双击蓝色的条目框，并浏览想调试的动态链接库文件。而且，依靠所设置的路径，当你进行调试时，操作系统在定位 DLL 文件时仍然会出现错误。避免此类问题的方法是，忽略 Additional DLLs 设置，并在项目 Debug 文件夹中放置一个调用程序的可执行文件副本。将调用程序和动态链接库均放置于相同的文件夹中，以确保 Windows 总是能够调出动态链接库。

在库源代码中设置断点后，选择 Go（转到）命令，或者按下 F5 键，来启动调试器。如果被调用的程序是调试或发行的格式，这两种启动方式将没有多大区别。但是，在后一种情况下，Visual C++ 显示一条消息，告知你程序没有符号消息。因为 DLL 文件正在被调试，而不是被调用的程序。这种消息仅仅是一个手续，提醒你不能跟随后面的执行流进入被调用的程序。单击 OK 按钮开始调试工作。

调试 OLE/ActiveX 应用程序

除了诸如 ActiveX 控件一类的进程内的服务器外，COM 的工作就像是远程过程调用（RPC）一样，从一个应用程序到另一个应用程序。一般来说，正在调用的应用程序是客户，被调用的程序是服务器。如果你只想开发一个服务器或者一个客户，应该关注所支持的远程过程调用一端所发生的情况。在这种情况下，对于调试 OLE/ActiveX 应用程序没有特殊的地方。对客户而言，在访问服务器的调用中设置一个断点，运行调试器，当断点被激活时，确保参数能正确地初始化，然后整体单步调试此调用，检查所有的返回值。当调试一个服务器时，在处理程序

函数中设置一个断点，以便接收远程过程调用，并运行调试器来启动服务器。然后，转换到调试器，程序应当在断点处被中断。

如果开发的是同时工作的客户和服务端，利用 Visual C++ 调试器，可以对远程过程调用的两端能进行调试，即使你所开发的应用程序是一个独立项目，调试器所要求的仅仅是一个额外的步骤。在两个项目中，从 Tools(工具)菜单中选择 Options (选项)，然后单击 Debug(调试)选项卡，并启用 OLE RPC Debugging(OLE RPC 调试)复选框。所有这些均能实现，只是在 Windows NT 中，必须要有管理员的特权才能启用此复选框。

如本书的第 4 部分所述，ActiveX 控件充当动态链接库的服务器，在同一地址空间内利用控件来执行。调试 ActiveX 控件与调试普通的动态链接库没有很大的区别，一个 OLE/ActiveX 服务器作为一个应用程序运行，而不是作为一个动态链接库来运行，而且，在不同的地址上运行，而不是在客户上，通过 PRC 穿过进程边界相互通信。Visual C++ 处理这种情况的方法是，运行调试器的两个实例，一个对客户，一个对服务器。当调试一个远程过程调用的两端时，必须有两种要求，它们都没有任何限制。首先，如上所述，必须利用 OLE PRC Debugging 复选框，其次，服务器应用程序必须是局部的。也就是说，它必须在同一机器上作为客户运行。Visual C++ 调试器不能启动在不同机器上运行的远程服务器，通过网络为调试客户应用程序，需沿着该点调用服务器的运行路径进行。如果接着进入调用的内部单步调试中，Visual C++ 会启动另一个调试器实例，如果它可用，就会加载服务器源代码。接着，在服务器响应远程调用时，单步调试通过服务器。当服

务器从 RPC 返回时，控制权恢复调试器的第一个实例，并且，在调用后，又返回到了客户的下一指令。调试器的第 2 个实例直到将服务器终止下来它才会终止。所以，现在立即穿越 RPC 桥接，再次从客户进入服务器进行单步调试，不要等待启动新的调试器实例。

你也可以从服务器端开始调试，尽管你必须手工启动客户应用程序，在想终止服务器的地方设置一个断点，接着按下 F5 来启动调试器，并启动服务器。转换到客户应用程序，并调出所要求的程序。接着转换返回调试器，以继续调试服务器。如果你从 RPC 处理程序函数中退出单步调试，并进入客户应用程序。Visual C++ 启动调试器的另外一个实例，并将它与执行客户关联。新的调试器实例只有在退出客户应用程序时才会终止，你可以在远程过程调用的不同位置继续调试客户和服务器，而不用管在哪个应用程序里开始调试。

用两台计算机进行调试

在调试时，经常由于调试器竞争被调试程序的屏幕空间而引起问题，当被调试程序运行时，它在屏幕上显示了正常的输出值，但是，调试器必须使用屏幕与用户联系。基于 DOS 的调试器像 CodeView 对此类问题有一个有效的解决方法，因为调试器仅仅在文本模式中运行，程序员能够将系统连接一个独立的单色监视器，用来显示调试器的源程序窗口、寄存器，并观察变量。同时，被调试程序在 EGA 或 VGA 主系统监视器上正常显示。两个监视器会使桌面显得拥挤，但是，

调试起来会更加简单、有效。

这种解决方法在 Windows 下是不可能的，因为调试器不再在文本模式下运行，调试器和程序使用相同的显存来运行，像其他的 Windows 程序一样，两者必须在一个或更多个窗口中显示其输出值。这意味着，当运行的程序被中断，并且当调试器获得控制权时，调试器窗口很容易就会覆盖属于被调试程序的窗口。当我们在前一章中调试 ShockWave 程序时，就看到了这种现象。

像以往的 CodeView 一样，Visual C++ 调试器提供了一种解决办法，将冲突的显示分离开来，引导它们到各自的监视器中。但是，你不是使用一个单色监视器，而是需要一个额外的能够运行程序的计算机及其主机环境，无论是 Windows 95 还是 Windows NT (Power Macintosh 不再支持)，因为 Visual C++ 不再支持串行空调制解调器链接，所以，两个计算机必须通过网络链接起来。一个计算机作为主机，以显示调试器窗口，而另一台计算机，指定为目标计算机或远程计算机，用来显示被调试程序的输出值。Visual C++ 称这种安排为远程调试。

远程调试由三步组成：

1. 拷贝文件到远程计算机上。
2. 配置主机。
3. 配置远程计算机。

步骤 1：拷贝文件到远程计算机上

拷贝文件 Msvcrt.dll, Tin0t.dll, Dm.dll, Msvcp60.dll, Msdis110.dll 到远程计算机的 Windows 文件夹中, 如果被调试程序在 Windows NT 下运行, 还需要拷贝 PsAPI.dll 文件。这些文件操作调试器的远程监视器程序, 文件在 Common\MSDev98\Bin 和 VC98\Redist 的 Visual C++ 文件夹中的子文件夹中。

步骤 2：配置主机

配置主机能告知 Visual C++ 在哪儿找到你想调试的程序, 以及它所运行的远程计算机种类, 和两台计算机的连接类型。首先, 单击 Project(项目) 菜单中的 Settings (设置), 在 Project Settings (项目设置) 对话框的 Debug 选项卡中标有 Remote Executable Path And File Name (远程可执行路径和文件名) 的文本框中指定程序路径。这个路径可以视作从自调试器正在运行的主机上进行查看。在这个文本框中, 输入程序路径, 作为 Msvcmon.exe 从远程计算机上的所在位置进行查看的程序路径。

接着, 从 Build (建立) 菜单中选择 Debugger Remote Connection (调试器远程连接), 以显示 Remote Connection (远程连接) 对话框。选择 TCP/IP 作为远程计算机的连接类型, 然后单击 Remote Connection (远程连接) 对话框的 Settings (设置) 按钮。这将会显示另一个对话框, 可以查询通信设置, 包括远程计算机的密码。

步骤 3：配置远程计算机

在远程计算机上运行 `Msvcmon.exe` 来调试监视器程序，当 Visual C++ Debug Monitor（调试监视器）对话框出现时，单击 `Settings`(设置)按钮，并在接下来的步骤中键入相同的密码，单击 `OK` 按钮就退出了对话框，然后，在主机上开始正常运行调试器。

第 12 章 编译器优化

Microsoft Visual C++ 编译器将 C 和 C++ 源代码转换成机器码，对程序的调试版本来说，这种转换是真正进行的，在完成的可执行程序中，产生了一系列低级的机器指令，它们完全表示源程序的高级指令。发行版本给编译器提供了更大的工作范围，因为完全进行源程序转换是没有必要的，甚至并不是人们所希望的。当创建发行版本时，编译器有不同的任务，它产生一个最小的或最快的目标代码，而不给程序引进新的和不能预料的行为。

这章有两个目的，一个是使你熟悉 Visual C++ 优化代码和处理影响优化的不同情况的方法。了解一些内在的进程信息，有助于你（而不是优化器）更好地工作，从而使源代码可以通过优化来进行改进。第 2 个目的就是解释 Visual C++ 中的许多开关和选项，它们是用来约束优化进程的，以便你能够更精确地理解，在打开或关闭一个开关时，编译器的行为将会如何作用。

为了完成这两个目标，本章粗略地分成两部分，前半部分描述编译器优化的全貌，解释技术，并讨论它们的优缺点。第后半部分将前半部分的概述同 Project Settings 对话框中的一个特定的编译器开关联系起来。最后，将详细说明优化问题，在汇编级上检查一个优化代码的例子。

优化入门

在本章的讨论中，仔细区分了运行的速度和尺寸，有些读者可能奇怪，为什么这里会区分它们，不是越小的代码速度越快吗？我们感觉是这样，广告也是这样讲的，它保证产品是“轻巧而快速”或“小巧而灵活”。但实际上，在可执行代码的速度和大小之间，没有什么严格的界限，而且，提高质量的优化可能反过来影响其他方面。

这里存在三种代码优化级别，程序员和编译器都可以使用它们，最高级是算法级别，这属于程序员。例如，快速排序算法能很容易执行一种简单的插入排序，并使用二叉树方法搜寻一个查询表，这比简单地从头到尾扫描要快得多。不幸的是，快速算法要比简单算法使用更多的代码，并且，简单算法的方法要更直接了当。

最低级的优化，被称为窥视孔优化，它属于编译器。在这种级别上，编译器利用机器特定的方法来保存一个消息或时钟周期，当通过整个程序积累时，存储将会更有意义。窥视孔优化通常由于代码比较小或者速度较快而引起，但并非总是这样，例如，Intel 指令：

```
and dword ptr [iVar], 0
```

它有 3 个字节，比较小，但是，它要比下面的指令慢 3 倍。

```
mov    dword ptr [iVar], 0
```

两个指令均将 0 赋给整数 iVar，在 80486 和奔腾处理器上，如下指令：

```
push    1
```

```
pop     eax
```

它采用 3 个字节和 2 个时钟周期，尺寸将近一半，但是，速度只有如下等效指令速度的一半：

```
mov     eax, 1
```

中间级别的优化介于算法和窥视孔优化级别之间，它包括了传统的优化技术，像子表达式的消除、复制传播和循环启动，所有这些将在下一节说明。这种中间级别经常留给编译器，尽管程序员也可以随意介入，例如，程序员可能会注意到，两个独立的循环可以在一个循环中实现（称为循环阻塞技术），并根据情况重新编写代码，典型的循环如下：

```
for (i=0; i<10; i++)
```

```
    nArray1[i]=i;
```

```
for (j=0; j<10; j++)
```

```
nArray2[j]=j;
```

循环阻塞将循环组合为一个循环，在同一循环中，所做的工作并不相同，这样便节省了第 2 个循环的开销：

```
for (i=0; i<10; i++)
```

```
{    nArray1[i]=i;
```

```
        nArray2[i]=i;
```

```
}
```

Visual C++不能识别这样的循环阻塞，因此，如果没有人为的干涉，便将失去优化机会。

当决定是否优化速度或者大小时，应该记住，速度的节省几乎总是可以度量的，但并不是总被人们感觉到。一个较普遍的原因在于，计算机时钟的测量和人们所能辨别的速度之间有很大的差别。提高最终用户感觉不到的程序运行速度，无异于浪费精力。

一般来说，只有算法优化能使运行速度得以显著提高，较低级的优化通常不能节省几百万个时钟周期，而这是人们能感觉到速度的提高所需要的，除非是应用优

化的特定循环和或函数执行上百次。由于这个原因，在编程方面，人们已经发展了以源程序级别来编写有效的算法，并设置编译器优化尺寸，而不是优化速度。对于多任务操作系统，例如 Windows 系统，特别鼓励这种方式，在内存拥挤的条件下，对于具有较小内存映像的程序，其引发页面错误的机会就更少。在出现页面错误时，操作系统必须从磁盘上重新加载内存，这种操作的代价是很昂贵的。将它们与程序组合在一起，无论对速度的优化有多大，看上去，也似乎没有反应，而且反应较慢。

优化技术

Visual C++从优化技术中引用的许多技术已经被编译器使用了多年。表 12-1 列出了一些最重要的优化技术，Visual C++利用它们，并指出其目的是用来减小代码的尺寸，提高代码速度，还是二者兼而有之。因为这里涉及到许多变量，有时，要事先预测优化技术的全面影响是相当困难的。然而，表中所反应的仅仅是编译器的意图，而不是必要的结果。对特殊程序的最佳优化设置经常通过试错法来决定。

在这里，我们开始给出一系列简短的子程序段，来检查表 12-1 中所列的 14 种优化方法。每个子程序段描述了优化是怎样工作的，什么时候使用它，它的优缺点是什么。

处理器寄存器的使用

在以往的 C 程序编制中，较好的方法是利用 Register 关键字，使一或两个函数的局部变量成为寄存器变量。寄存器存储类反应了程序员的一个要求，即告诉编译器，如果有一个可用的寄存器变量，就不将它分配在堆栈中，而在处理器寄存器中保持局部变量。除了节省少量的堆栈空间外，在寄存器里保持变量，可以保证访问它的速度最快，因为处理器读写自己的寄存器要比读写内存快得多。在寄存器中管理一个变量（而不是在内存中）也能导致代码大小稍有下降。

表 12-1 Visual C++ 编译器优化技术

优化	减小大小	提高速度
处理器寄存器的使用	✓	✓
常量传播和复制传递	✓	✓
死码和死存储区的消除	✓	✓
普通子表达式的消除	✓	✓
循环优化	✓	✓
指令计划		✓
强度减少	✓	✓
内嵌扩展	✓	✓
字符串合并	✓	
禁用堆栈检查	✓	✓

续表		
结构指针的删除	✓	✓
堆栈覆盖	✓	✓
假定没有使用别名	✓	✓
函数级别链接		

今天，人们很少看到有人使用 register 变量，因为优化的编译器（像 Visual C++）能自动处理这项任务（实际上，Visual C++ 忽略了 Registry 关键字）。几乎所有的数据对象都可以作为寄存器对象，例如，全局和局部变量、常数值、结构元素和函数参数，包括按引用传递的参数指针。编译器扫描函数，来决定怎样使用它的数据，给每个变量分配一个分数，来代表在寄存器中存储变量的好处。当写出函数的目标代码时，编译器尽可能将得到最高分的变量置于寄存器中。结果是，在适当的情况下，提高了运行速度。

在计算机中，寄存器较少，并且，在决定什么时候使用寄存器来存储变量时，编译器必须作出明智的选择。优化的代码花费部分时间，以便在寄存器和内存之间改变数据。代码通过将它的内容写入变量的基本内存地址来释放寄存器。但是，优化编译器必须首先决定是否值得内存访问。释放寄存器，而后再再次用同一值调用，如果它使寄存器可以用于更短的代码段，这样做也可能并不值得。

常量传播和复制传播

代码优化中的指导原则就是寄存器比常量快，常量比内存要快。如果没有足够的可用寄存器来容纳一个代码段中的所有变量，下一个最佳选择方法就是用一个常量来代替表达式。编译器遇到常量传播时可以使用常量，在这种情况下，指定的常量值向前或通过代码传播。编译器通过替换表达式来优化代码，以求出一个常量值，例如，下面的行：

```
X = 255 ;
```

```
Y = X ;
```

用下面的式子表示会更好：

```
X=255 ;
```

```
Y=255 ;
```

通过用一个常量值重写第 2 行，编译器便节省了一个不必要的内存访问。尽管优化技术本身经常就叫做“传播（propagation）”，这个术语更准确地说明了优化即意味着修正。

复制传播类似于常量传播，在一系列的赋值中，一个值从一个变量传递到另一个变量时，便会发生复制传播。在这种情形中，中间的变量不使用值，而只是将它传递到下一个变量。在这个系列中，更有效的是，将值直接赋予最后的变量，并跳过其他变量。这里有一个例子，在此，删除复制传播，来给出一个无关紧要的语句，编译器作出了一个简单的替换。看看这些代码序列：

```
i= nParam;
```

```
Function(i);
```

```
i=j;
```

转变为：

```
i= nParam;
```

```
Function(nParam);
```

```
i=j;
```

在这里，nParam 值通过 i 来传播，以便变成 function 的参数，但是，既然 i 从来不使用 nParam 值，复制传播是没有必要的。编译器能较安全地将 nParam 替换为函数参数。因为这种优化，第 1 个赋值语句现在变成了无用的“死存储区”，在下一节将会讨论这一点。

死存储区和死码的消除

当我们看到前面的例子时，复制传播经常将中间赋值语句留作死存储区，程序可以将数据赋给一个变量，而无需从中读取它。当它识别出死存储区赋值时，优化编译器只是忽略这些指令，以便它不会变成目标映象中的一部分。例如，在编译器消除了死存储区后，原来此段中的三个指令被减少为两个指令。

```
Function (nParam)
```

```
i = j
```

在编译器扩充了复杂的宏后，便可以删除复制传播和死存储区了。

与死存储区相关的是死码。死码是程序运行时处理器不能到达的一条指令或一群指令，这些不可达到的代码经常是目前优化的副产品，由于编译器没有为死码或死存储区产生目标指令，因此，消除这些条件代表了最理想的优化。

公共子表达式的消除

当编译器识别到一系列表达式都影响相同的值时，它会计算一次子表达式，并替换系列中所有子表达式的结果。例如，考虑下段中的子表达式 $y * z$ ：

```
x=y*z;
```

```
w=y*z;
```

通过添加一个赋值语句，并用一个变量替代两个子表达式，编译器消除了两个多相操作中的一条。

```
temp=y*z;
```

```
x=temp;
```

```
w=temp;
```

根据环境以及子表达式是否经常出现，替换可以减少程序段中的代码大小。而且，消除公共的子表达式几乎总能提高运行速度。

循环优化

循环内部的优化通常是显著有效的，因为速度是通过循环的次数来加倍提高。在

以前程序段里所描述的优化仅仅当应用是在循环内的代码时才会更有效。但是，这里也有其他的优化技术，适用于针对循环的编译器。也许，最常见的循环优化技术就是不变式移动或提升，这意味着将代码从一个循环内移到循环外，“不变”是指，在通过所有的循环时，一个表达式仍保持常量值，这里有一个典型的例子，它是循环中的不变表达式：

```
for (i=0; i<10; i++)
```

```
    nArray[i]=x+y ;
```

通过从循环中移出不变表达式，编译器产生仅计算一次表达式的代码，而不是 10 次，在代码大小上没有明显的增大。

```
temp=x+y;
```

```
for(i=0; i<10; i++)
```

```
    nArray[i]=temp;
```

指令顺序安排

如果一条指令不依赖于其他指令的结果，高级处理器（像奔腾系列）能够在两个类似的流水线同时执行两条指令。独立性可能会导致称为流水线堆栈的条件。通

过使用指令时序安排，也就是指令定序，编译器通过重新安排机器指令的顺序，以阻止这样的独立性产生。例如，考虑标为 A，B，C 的三个指令：

```
add      ax,iShort      ;Instruction  A
```

```
movsx    ebx,ax          ; Instruction  B
```

```
xor      ecx,ecx         ; Instruction  C
```

指令 A 和 B 不能同时执行，因为指令 B 依赖于指令 A 的结果。也就是说，在运行指令 B 之前，处理器必须了解寄存器 AX 中的值和它的符号位状态。而且，指令 C 既不依赖于 A，也不依赖于 B，通过颠倒指令 B 和指令 C 的顺序，编译器便避免了潜在的拖延情况的发生，允许指令 A 和指令 C 同时执行：

```
add      ax,iShort      ; Instruction  A
```

```
xor      ecx,ecx         ; Instruction  C
```

```
movsx    ebx,ax          ; Instruction  B
```

指令顺序安排对代码的大小无多大的影响，并且，仅当程序在高级处理器上运行时才会发生作用。在此章的最后一节中，对指令顺序安排有较详细的说明。

降低强度

处理器在做加法和减法时比较快，在做乘法和除法时，相对要慢些。例如，奔腾处理器在一个时钟周期中可以添加两个 32 位寄存器，而在进行乘法时，需要 10 个周期，在做除法时，所需要的周期要超过 40 个。在优化时，Visual C++ 编译器要寻求减少算术复杂性的机会，也就是减少指令“强度”的机会，而不影响计算的结果。

例如，乘以或除以 2 的指令强度可以通过等效的移位操作来实现。假定 y 是一个无符号整数，对于如下操作：

```
y=y/16
```

编译器可以将它替换为：

```
y=y>>4;
```

这种替代产生了与原始指令同样的结果，因为除以 16 (2^4) 可以与右移 4 位有同样的效果。以其他方向移位也一样，因此，整数变量乘以 2^n ，就等效于左移 n 位。

在从汇编级别来观察时，这种优化要更有趣，这里给出了反汇编时原始指令的状态，每条机器指令使用的时间在注释中给出：

```
// Instructions for y = y/16
```

```
mov     ecx, 16                                ; 1 cycle on a Pentium
```

```
mov     eax dword ptr [y]                      ; 1 cycle
```

```
cdq                                           ; 3 cycles
```

```
idiv    ecx                                    ;46 cycles
```

```
mov     dword ptr [y], eax                    ; 1 cycle
```

```
                                           ;52 cycles total
```

强度减少可以利用一条简单的指令替代：

```
// Instruction for y=y>>4;
```

```
sar     dword ptr [y],4                      ;3 cycles total
```

在 Visual C++编译器里，这个例子颇具学术价值。用等效的移位指令来替代乘或除操作，其改进的效果相当明显，Visual C++即使在优化关闭时也会作出这样的替换。

内嵌扩展

为什么调用一个函数的行为会使程序执行流的进程减慢，原因有几个。因为处理器会跳到代码的新位置，处理器指令队列中保存的即将调用的指令就不再有用了。如果处理器不执行二分法预测（奔腾机也是一样），在这个队列满时，并且函数的第一个指令从内存中检索到时，它就会停下来。更糟糕的是，当函数的参数被推到堆栈上和处理器的 EIP 寄存器上时，此调用也许会产生一系列的内存写操作（对 EIP 寄存器的描述，参见第 11 章中有关调试器的附加信息）。当函数完成后，当返回地址从堆栈中弹出到 EIP 寄存器中时，处理器会再次停止，如果有必要的话，预取的队列会被填满，并且，EIP 指向的下一指令将从内存中读取。简而言之，调用函数和退出函数的代价都相当昂贵。

内嵌扩展可以解决这些问题，但有时会以增大代码大小为代价。在这个优化技术里，编译器将函数码插入程序的主体里，用一个函数自己的复制版本来代替函数调用。永远也不生成 CALL 机器指令，使处理器可以沿着指令的顺序路径进行，而不会转到其他地方。按照顺序的逻辑路径，处理器能够更精确地预取指令，当内嵌扩展发生在一个循环中时，节省的时间会更多。

看上去似乎很奇怪，但内嵌扩展经常可以降低程序的大小。内嵌扩展在应用到较小的函数时会更有效，尤其是对于参数是常量，或者按指针传递而不是按值来传递的参数。在这些情况下，编译器可以不需要整段代码，便将参数值写到堆栈上。内嵌函数能够节省程序段的起始和结束部分，并可以省却创建独立的堆栈结构。

内嵌也暴露了函数的副作用，例如，改变为全局变量，它们对编译器来说是不可见的。这可以进行更多的其他优化，可能无需内嵌扩展也行。

字符串合并

编译器能决定什么时候程序多次创建相同的字符串。字符串合并是一种优化技术，在这种技术中，编译器仅仅为第 1 个字符串分配数据空间，然后将指针重新指向第一个字符串的任何复制字符串。

堆栈结构指针的忽略

堆栈结构指针的忽略对 Intel 系统来说是一种优化，它能节省代码的起始和结尾代码，对拥有许多函数的程序来说，这种节省是可观的。如果没有堆栈结构指针的忽略，编译器会为每一个要求堆栈结构的函数产生序言代码。将处理器的 EBP 的寄存器指向堆栈结构的顶端，如下：

```
push    ebp                ;Save  EBP  register
```

```
mov     ebp, esp           ;Point to top of frame
```

```
sub     esp,local_space    ;Allocate  stack frame
```

当函数完成时，结尾代码消除堆栈结构：

```
mov esp,ebp ;Restore stack pointer
```

```
pop ebp ;Restore EBP register
```

利用这种方法，EBP 寄存器被堆栈结构指针所调用，自动内存类变量在堆栈结构中通过相对于 EBP 的偏移量而被引用，利用 EBP 作为结构指针是 Windows 老版本中遗留下来的，它被设计在 Intel 80286 处理器上运行。当结构指针的忽略起作用时，编译器根据 ESP 寄存器的相对堆栈数据，而不是 EBP 寄存器，来引用堆栈数据。函数的序言变成一个简单的指令，来调整 ESP 堆栈指针创建堆栈结构：

```
sub esp,local_space ;Allocate stack frame
```

结尾程序会去除堆栈结构，方法是将 local_space 添加到 ESP，更好的是，堆栈指针的忽略使 EBP 寄存器可以自由地用在其他优化上，结构指针忽略的弊端在于，要编码相对于 ESP 的内存指针，这比相对于 EBP 的相同指针多用一个字节。

禁用堆栈检查

Win32 里的堆栈检查与 16 位环境不一样。在 16 位 Windows 中，堆栈检查涉及到调用一个 C 运行函数，即堆栈探测器（stack probe）。在一个程序中，在每个函数的开始进行调用，堆栈探测器确定这个堆栈是否有足够的空间，来适应函数的自动存储要求。如果堆栈空间足够，探测器返回，并且函数继续运行。否则，探测器会通知开发人员函数不能执行，因为它超出了堆栈的限度。

Win32 应用程序不要求这种堆栈检查，因为有一种系统服务阻止了堆栈溢出。当一个程序（或线程）访问堆栈底端附近的内存时，操作系统假设堆栈空间已经不够用了，并且相应提高堆栈的大小。这将在程序的最深访问处和堆栈的底端之间产生更大的距离。尽管自动堆栈调整使老式的 16 位堆栈探测器过时，堆栈检查仍然可以应用在 Win32 应用程序里。为了了解这种目的，有必要检查系统怎样将内存添加到堆栈上。

对一个应用程序或单独的线程来说，堆栈空间在页面提交。页面大小取决于目标系统，对 Intel、MIPS 和 PowerPC 系统来说，一页有 4KB。仅当堆栈的访问结束，进入到保护页的区域后，操作系统才能识别堆栈溢出。保护页是堆栈中最后的提交页（Windows NT 中设置的保护页与 Windows 95 中略有不同）。当程序访问保护页中的堆栈内存时，系统将提交另一页，来增加堆栈的大小，此过程称为“堆栈的增长”。图 12-1 显示了堆栈是怎样通过被操作系统提交的页而增长的。

如图 12-1 所示，对一个应用程序来说，超出堆栈保护页，并试图进入保留的内存，这是可能发生的。这可能发生在函数为其局部变量分配更多的堆栈页面的情况下：

```
void BigLocal ( )
```

```
{
```

```
    char chArray [3*4096];           //Allocate 3 pages (12 kb) of stack
```

```
    chArray[12000] = -1;           //This assignment may fail  
  
    .  
    .  
    .  
  
}
```

在这个简单的说明中，chArray 假定 3 页(12KB)的堆栈。函数为自动数据分配空间，方法是按请求的 12KB 来减少处理器的堆栈指针 ESP，但是，这不可能提交更多的堆栈。如果分配给 chArray 的空间在靠近堆栈的底端进行，访问 chArray 中的高序元素可能超过堆栈的保护页，从而进入保留的内存。这将触发一种非法的访问操作，系统会通过终止应用程序来解决这种系统问题。

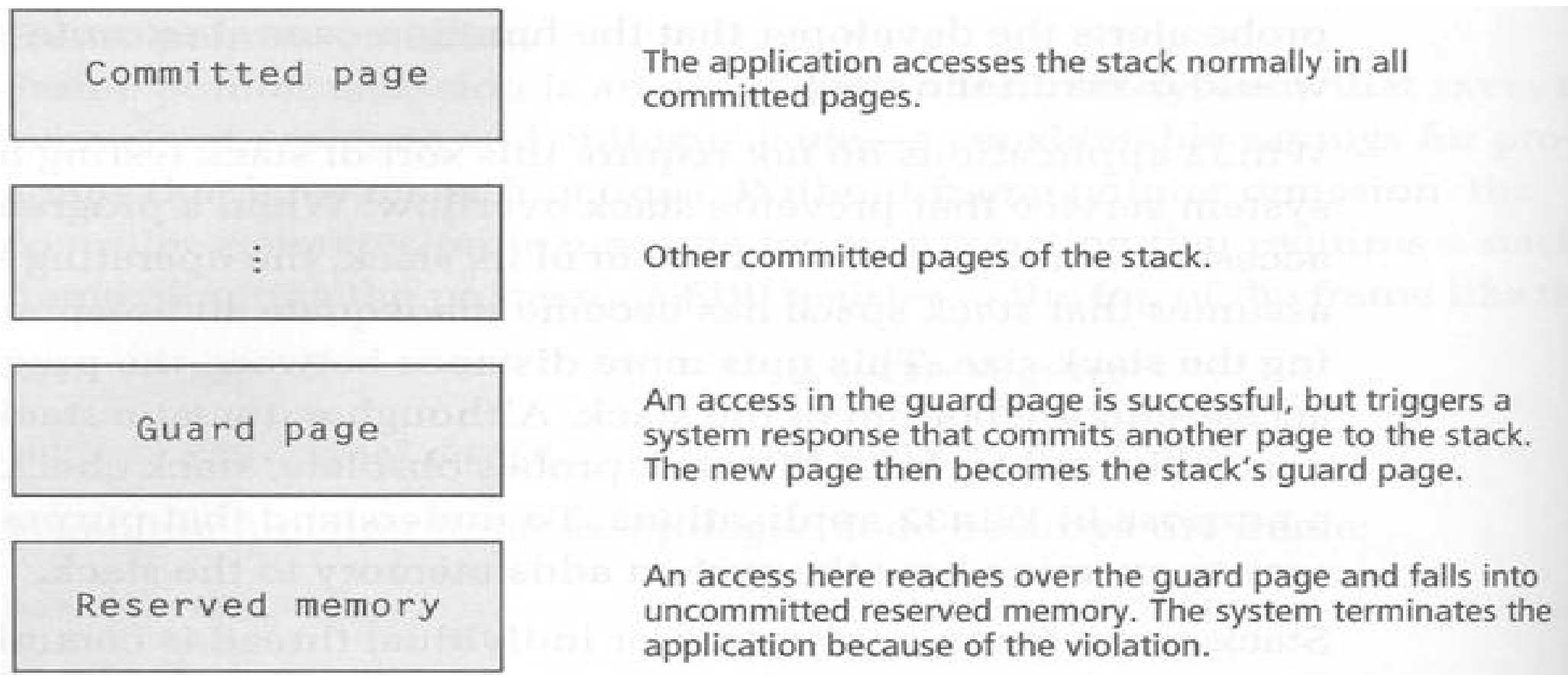


图 12-1 应用程序堆栈的增长

在 Win32 的堆栈检查中，可以阻止这种类型的方案。当启用堆栈检查时，编译器计算每个函数局部数据的总尺寸，带局部变量的函数消耗的堆栈页面要少一个，不能超出保护页，并因此不能要求堆栈检查。而且，带有多页自动数据的每个函数要优先于在 C 运行库中的堆栈检查调用。堆栈检查例程只是到达接下来

的堆栈页，也就是说，它进入一个循环，来读取堆栈上的字节，其增量为 4096 字节。循环在堆栈的顶端开始，并向下继续执行，直到堆栈检查例程到达足够多的页，来满足函数的堆栈要求。

图 12-1 显示了堆栈检查怎样解决 BigLocal 函数的问题，在 BigLocal 函数得到控制权之前，堆栈检查例程到达堆栈顶端下面的 1，2 和 3 页。假定 chArray 的分配在堆栈中的最后提交页中开始，第一次到达将访问保护页，系统的反应是提交另一页，使它成为新的保护页。在堆栈检查例程中，循环里的第 2 次重复到达新的保护页，使得系统提交下一页。此处理会第 3 次重复，在堆栈检查例程返回和 BigLocal 得到控制之前，给堆栈增加三页。现在，当 BigLocal 存取邻近 chArray 末尾时的一个元素时，存取陷入堆栈的提交页中，并且不会触发错误。

BigLocal 没有堆栈检查也能解决它自己的问题吗？绝对可以。例如，只要在 Array[12000]之前访问 chArray[4000]和 chArray[8000]，函数便可以照管提交要求的内存，并确保堆栈没有溢出。堆栈检查给程序添加了开销，禁用它可以节省代码，并且，对于具有很大的存储器要求的应用程序，可以增强其运行速度。只要在随后的页中存取堆栈数据，并从堆栈的顶部向底部工作，这些应用程序不要求进行堆栈检查。

堆栈覆盖

堆栈覆盖优化也许有好处，也许没有好处。这取决于堆栈适用的范围。通过使用堆栈覆盖，编译器重新使用堆栈空间，来保存没有覆盖的局部变量，这意味着，

在一个函数中，如果最后访问 X 在第一次访问 Y 之前，则 X 和 Y 均能安全地占据堆栈结构的同一位置。

通过最大限度的地减小所占据堆栈的深度，编译器降低程序运行时堆栈溢出的机会。尽管堆栈增大时的系统反应对用户来说是透明的，但操作会消耗不少时间，堆栈覆盖也能减少一个函数的运行大小，方法是缩短堆栈上的局部变量与堆栈结构顶端之间的距离。重新调用函数的结构指针指向堆栈结构的顶端。如果在一个堆栈中，局部变量所占据的位置距离堆栈结构指针少于 128 个字节，则编码每个变量的指针时，要比偏移量大于 128 字节时少用 3 个以下的字节。

```
mov     eax,[EBP+4]      ; This instruction is 3 bytes smaller
```

```
mov     eax,[EBP+256]    ; than this instruction
```

任何在堆栈上从覆盖变量所得到的好处都与环境有关，但是，堆栈覆盖没有任何代价。

假定没有别名

别名意味着使用多个名字引用一个内存对象。指针和联合给程序员提供了众多的机会来使用别名。下面给出了一个比较典型的例子，其中，c 和 *cptr 引用内存中的相同字节。

```
char c;
```

```
char *cptr = &c;
```

别名的使用限制了编译器进行某种优化的能力，如将变量设为寄存器变量。例如，在这种代码段中，编译器不能安全地存储寄存器中的变量，如果有这种可能性，程序随后将用 `cptr` 而不是 `c` 将一个新值写入内存。如果发生这样的情况，寄存器内的值将不再有效。编译器能经常较成功地跟踪 `cptr` 和 `c` 的使用，而且，不管别名的使用如何，在安全的情况下，它可以重新将 `C` 作为寄存器变量（`cptr` 变量在任何情况下均能被再次寄存）。

别名可以假定一种微妙的格式，使编译器不能识别，接下来的代码描述了两个变量 `ptr1` 和 `ptr2` 均指向同一数组的情况。然而，编译器不能识别这个别名，因为 `ptr2` 是从主函数作用域之外的另一个函数中获得它的值。

```
char chArray[5];                                // Global scope
```

```
main ()
```

```
{
```



```
char *ptr1 = chArray;           //ptr1 points to chArray

char *ptr2 = GetPointer();       //So does ptr2

}

char * GetPointer (void)

{

    return chArray;

}
```

当以保守的方式进行优化时，以上例子能够正确运行，因为编译器不能事先知道 ptr2 的值，它允许这种可能性存在，即 ptr1 和 ptr2 均是同一内存目标的别名。因此，它可能在涉及到其中任何一个指针时不进行任何优化。这种假设是比较安全的，但也可能会引起编译器错过合理优化的机会。

Visual C++编译器提供了一个 Assume No Aliasing（假定没有别名）开关来处理类似这样的情形。这种优化开关提示编译器变量没有隐藏的别名，与本例子中的 chArray 的情形一样。此开关给编译器提供了一种权限，去主动优化涉及指针的

代码，这些指针通过不可见的别名来释放。

相对于 Assume No Aliasing 选项，Visual C++ 还提供了不太主动的方式，名为 Assume Aliasing Across Function Calls（假定别名跨越函数调用）。这种优化开关告诉编译器，假定别名在代码中不存在，除非是跨越函数调用。此开关给予编译器的仅仅是限定的权限，来优化涉及到指针的代码，但是，这比根本没有权限要更好一些。本章的后一节将会讨论这两个优化开关经常是怎样应用于试错方法中的。

函数级别的链接

对一个函数而言，在它外面对它进行优化是可能的，可以通过内嵌方法，或者因为编译器已经知道了怎样用这种从不调用函数的方式来编译程序。而且，因为编译器无法决定是否有的源模块访问此函数，函数必须要被编译，并包含在目标映象中。仅当链接器能识别何时函数在一个程序中保持不引用时。如果编译器以压缩形式写下了一个未引用的函数，链接器将从所完成的可执行程序中删除它。

函数级别的链接确保在一个源模块里的所有函数都封装起来。也就是说，在目标代码中，通过一个 COMDAT 记录来识别。COMDAT 记录的格式是 COFF（Common Object File Format，通用目标文件格式），含有允许链接器去识别未引用函数，并从可执行映象中删除它们，这个过程叫做“传递 COMDAT 消除”。如果没有 COMDAT 记录，在链接后，映象中将保留未引用的函数，从而占据空间。

优化开关

现在，让我们从概述转向细则。本节主要是将目前所学的编译器优化同 Visual C++ 中控件优化进程的开关联系起来。开关在图 12-2 所示的 Project Settings 对话框中，可以使用 Project 菜单中的 Settings 命令来调用它。Project Settings 对话框具有许多开关和选项，它们可以影响建立的进程和最终可执行程序的效率。这节集中在对话框的 C/C++ 选项卡上，其中包含有用来管理编译器如何优化项目源文件的一切开关。

默认优化设置取决于建立的目标。在建立调试版本时，Visual C++ 会关闭优化，以确保可执行程序真正转换的是源程序。对于发行版本，默认情况下，编译器优化速度，甚至不惜增大代码尺寸。对许多项目来说，默认优化设置是可以接受的，并且，在本书中已经用实例说明了这一点，你可以较轻松地建立和开发一个无需进入 Project Settings 对话框的项目。我们将会明白，为什么需要人工调节项目的优化设置。

图 12-2 显示了含有一列项目源文件的 Project Settings 对话框的左半部分，它类似于 Workspace（工作空间）窗口里的 FileView 窗格。在设置优化开关之前，从列表顶端选择项目名或单独的文件。为了选择一组文件，当按下 Ctrl 键时，单击需要的文件名。文件列表中的选择指出优化开关将应用在哪个文件或文件组。选择项目名，使优化设置对于所有源文件是通用的。选择单独的模块，可以为最快速度而优化其中的一些，有些是为了最小的尺寸，还有一些是为了所选择的混合优

化标准。初始的建立目标在对话框的左上角显示，它取决于当前项目的有效配置。当设置编译器优化时，目标应该是 Win32 发行版本。在 Project Settings 对话框中选择目标并不能改变项目的有效配置。

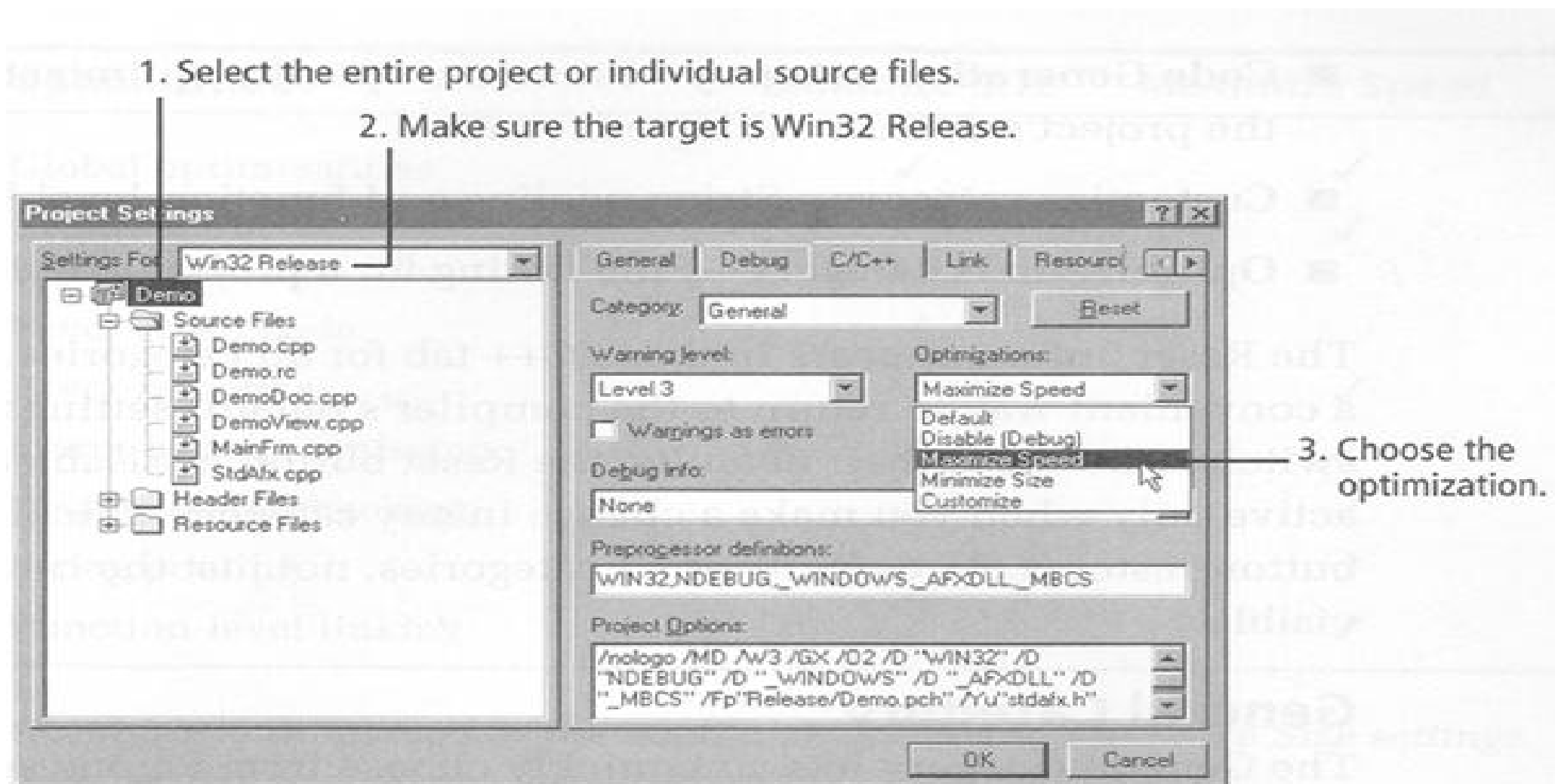


图 12-2 在 Project Settings 对话框中选择优化目标的快捷方式

为了更细致地进行优化，可在源代码的关键位置插入 `optimize`。这会为单独的函

数设置编译器优化开关，从而忽略目前的项目设置，你可以优化特定函数的运行速度，例如，当为代码的大小优化源模块的余下部分时，参考在线帮助，以寻求更多的关于 optimize 的信息。

Project Settings 对话框中的 C/C++ 选项卡的外观取决于对话框顶部的 Category 框中的当前选项。在这个框中已列出 8 种编译器分类，四种含有所有的与编译器优化相关的开关。

- **General category**（通用分类）：对于总的优化目标，这是很方便的选择，它没有针对单独的优化方法的微调使用的控件。
- **Code Generation category**（代码生成分类）：处理器特定的优化和项目的默认调用约定。
- **Customize category**（自定义分类）：字符串共用和函数级别的链接。
- **Optimizations category**（优化分类）：对项目优化进行微调。

在 C/C++ 选项卡中，所有分类都有 Reset（重新设置）按钮，而且，它是返回到编译器的默认设置的方便方式。当所有开关被设置为默认值时，Reset 按钮被禁用，只有在任意分类中作出某种变化时，它才会变成有效的。单击 Reset 按钮可以恢复所有分类的默认设置，而不只是可见的分类。

通用分类

在通用分类中，你可以快速从几种名为 Default（默认）、Disable（禁用）、Maximize Speed（最大化速度）、Minimize Size（最小化大小）和 Customize（自定义）（图 12-2）的优化设置中选择。因为 Disable 设置代表了完全抑制所有编译器优化的唯一方式，它用于进行调试。Default 设置清除了所有优化开关，包括 Disable 开关，这意味着，编译器还可执行一些提高速度的优化，稍候，我们将介绍其详情。Default 设置不是特别有效。只有在想手工控制字符串共用和函数级别链接优化的开关时，才选择 Customize 设置。这些开关出现在 Customize 分类中，这将在以后简单进行描述。

在 General 分类中，最重要的优化设置是 Minimum Size 和 Maximize Speed。这些开关可以作为大多数（不是所有的）优化技术中的快捷方式，允许你选择优化目标，而无须涉及细节。表 12-2 显示了 Maximize Speed 和 Minimize Size 设置启用的特定优化。

Maximize Speed 和 Minimize Size 设置很方便，但有点保守。如表 12-2 所示，两种设置均不能启用别名优化（Assume No Aliasing 优化是 Optimizations 分类的一部分）。虽然 Maximize Speed 设置包括字符串共用和函数级别链接的优化，这些优化技术一般仅仅改善代码的尺寸，而不是速度。

表 12-2 通过 Maximize Speed 和 Minimize Size 设置启用的优化

优化	最小尺寸	最快速度
Global optimizations (全局优化)	✓	✓
Generate intrinsic functions (产生内部函数)		✓
inline (内嵌)	✓	
Favor small code (支持小代码)		✓
Favor fast code (支持快速代码)	✓	✓
Frame pointer omission (结构指针的忽略)	✓	✓
	✓	✓
Disable stack checking (禁用堆栈检查)	✓	✓

续表

String pooling (字符串共用)

Function-level linking (函数级别
链接)

目前，我们还没有遇到表 12-2 中的前 4 项，因此，需要对它们稍许解释一下。对于本章前半部分介绍的某些编译器优化来说，Global optimizations 这个术语是包罗万象的，例如窥孔优化，处理器寄存器的使用，循环优化，强度降低，消除不需要的元素，像死存储区和死码。表 12-1 中所列的前 8 个编译器优化均在全局优化的范围中。

一般地，通过时间运行库提供的函数，在表 12-3 中列出的函数有其特定的格式。编译器编写内部的内嵌函数。也就是说，在选择 Maximize Speed 选项时，没有一个函数调用。放置内嵌内部函数能够提高程序的运行速度，但也能导致程序尺寸更大，这取决于程序使用内部函数的程度。例如，在 Intel 处理器上运行的应用程序中，strcpy 函数的内部形式占据了 41 个代码字节。调用函数的正常运行版本最多占用 18 个字节，其中包括指令，来传递堆栈上的两个字符串指针，以及随后的堆栈清除，它通过调用程序来处理。使用 strcpy 函数的运行版本，而不

是它的内在格式，这样可使随后大量使用此函数的应用程序代码大小大幅度降低。当仅应用于一两次调用时，这种节省似乎不太有用。

表 12-3 Visual C++ 中内在的运行函数

<code>_disable</code> <code>strcat</code>	<code>_lrotr</code>	<code>_strset</code>	<code>exp</code>	<code>memcpy</code>
<code>_enable</code> <code>strlen</code>	<code>_outp</code>	<code>abs</code>	<code>fabs</code>	<code>memcpy</code>
<code>_inp</code> <code>strcmp</code>	<code>_outpw</code>	<code>atan</code>	<code>labs</code>	<code>memset</code>
<code>_inpw</code> <code>strcpy</code>	<code>_rotl</code>	<code>atan2</code>	<code>log</code>	<code>sin</code>
<code>_lrotl</code> <code>tan</code>	<code>_rotr</code>	<code>cos</code>	<code>log10</code>	<code>sqrt</code>

除了针对表 12-3 中函数的内嵌调用，转向内部优化也可以加速 `acos`, `asin`, `cosh`, `fmod`, `pow`, `sinh` 和 `tanh` 库函数的调用。虽然这些函数并不是真正的内部函数，编译器通过写入代码来直接将函数参数放在浮点芯片中，而不是将它们置于堆栈

中，以此来优化它们的性能。结果是在函数中将会花费更少的时间，但代码的代价要提高些。

当编译器遇到某种通过优化来改善速度或大小的代码序列，但并不是两者同时优化时，Favor Small Code 和 Favor Fast Code 优化将影响编译器的决定。例如，考虑一条指令，用 71 乘以变量 x，因为编译器不能通过简单的移位来完成乘法计算，因此，当决定怎样将操作转换成机器代码时，它可以有两种有效的选择。如下所示，第一种选择要慢一些，但要求的代码更少。

```
// Instructions for x *= 71;

mov      eax, dword ptr [x]      ;          1 cycle      4 bytes

imul     eax,eax,71              ;          10 cycles     3 bytes

mov      dword ptr [x], eax      ;          1 cycles     4 bytes

                                ; Total :      12 cycles     11 bytes
```

第二种选择使用特定的 Intel 技巧，来避免昂贵的 IMUL 指令，结果是速度更快，但代码序列更长。

```
// Instructions for x *= 71;
```

mov	ecx, dword ptr [x]	;	1 cycle	4 bytes
lea	eax, dword ptr [ecx+ecx*8]	;	1 cycle	3 bytes
shl	eax, 3	;	1 cycle	3 bytes
sub	eax, ecx	;	1 cycle	2 bytes
mov	dword ptr [x], eax	;	1 cycle	4 bytes
		; Total:	5 cycles	16 bytes

比较这两个代码段的实际运行速度是比较困难的，因为使用时钟周期极少能区别整个事件。时钟周期所测定的仅仅是处理器在运行一个指令时所花费的时间，不是读指令和从内存读取数据到处理器时所花的时间。尽管就处理时间而言，第 2 种代码序列要比第 1 种代码序列快得多，严格来说，真正提高的速度可能要比这些数字所指示少些。因为第 2 种序列要更长一些，处理器必须花更多的时间来访问和解码代码的多余字节。这是一个比较少见的问题，而且，如果序列出现在循环内部，因为循环的第一次重复之后，处理器将从它的指令缓存内而不是内存内取出代码。

当试图决定两个序列中谁代表速度较快的代码时，要考虑到另一个能进一步影响结果的因素。第 2 个序列比第 1 个序列要使用更多的寄存器，而寄存器有利于帮

助优化另一部分代码。估计另一种优化方法的整体效果经常是相当困难的。一般来说，当对大小而不是速度作出假设时，可能更容易猜测些。

代码生成分类

在 Category 框中选择 Code Generation (代码生成)，可选择在图 12-3 中所示的选项，包括：

- 优化处理器类型
- 编译器假设的默认调用约定
- 应用程序所采用的运行库类型
- 结构成员的对齐

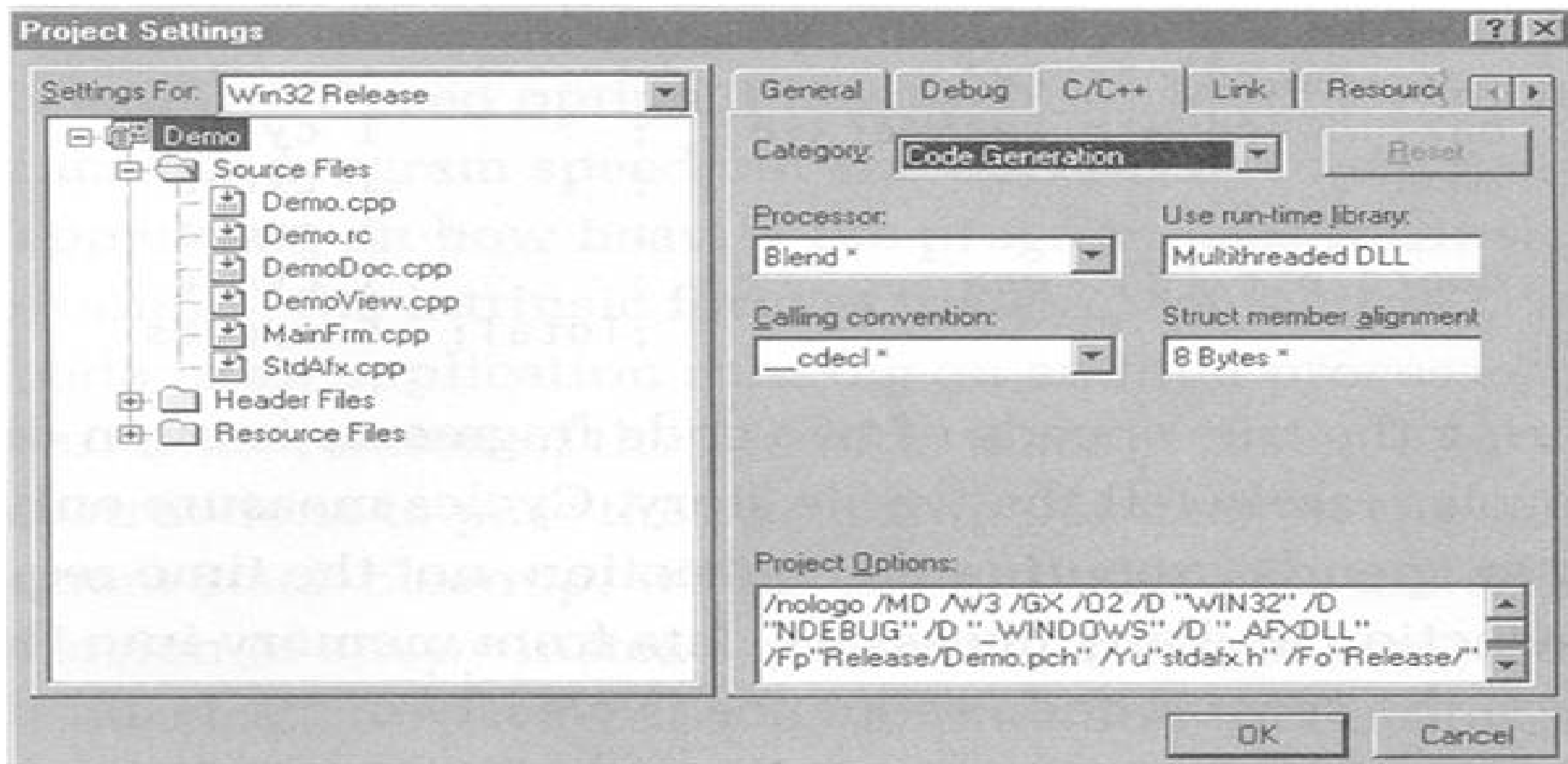


图 12-3 Code Generation 分类中的选项

处理器

在 Processor (处理器) 框中, 可以选择优化的处理器级别。默认设置为 Blend (混合), 它代表了一种折衷方案。在 Visual C++ 版本 4 中, Blend 设置使得编译器主要用于 Intel 80486 的优化, 为 80386 和奔腾处理器添加在 80486 上并不禁止执行的优化。在版本 5 和版本 6 中, Blend 设置主要针对奔腾机, 为低级的 80486 处理器添加所选择的优化。

不考虑处理器设置, 编译器所产生的仅仅是能被 80386 识别的机器指令。这能保证优化的程序即使是利用 Pentium 或 Pentium Pro 设置来编译, 也能运行在低级处理器上。实际上, Pentium 和 Blend 设置具有同样的作用。

调用约定

Calling Convention (调用约定) 框中的选择确定为项目或所选定源文件的默认调用约定。此设置仅仅指定了默认调用约定, 所有约定都明显地包括在超越默认设置的函数原型中。调用约定为调用程序和被调用的函数确立了规则, 指定参数以什么顺序压入堆栈中, 外部名字怎样给定, 当函数返回时谁将清除堆栈。

Visual C++ 能识别 `_cdecl`, `_fastcall` 和 `_stdcall` 调用约定, 这些均为 C 关键字命名, 以函数声明的顺序指定约定, 约定总结如下:

调用约定 序	参数顺	堆栈清除	名字
<code>_cdecl</code>	从右至左	调用程序	<code>_function</code>
<code>_fastcall</code> 左	从右至 左	调用 <code>@function@nnn</code>	函数
<code>_stdcall</code>	从右至左	调用 <code>_function@nnn</code>	函数

表中最右端栏目所描述的是函数名怎样出现在目标列表中，其中，`function` 出现在源程序中时，它代表函数名，`nnn` 表示参数列表的大小，以字节为单位。表中所总结的方案仅仅应用在 C 程序，以及 `extern "C"` 关键字的 C++ 函数中。如果没有关键字，C++ 便使用一种不同的设定系统。

`_cdecl` 设置指定了 C 调用约定，这种约定变量参数列表，原因是调用程序负有在函数返回后清除堆栈的义务。在一个函数调用后，清除堆栈只需要使用一个简单的机器指令，以便来重新设置堆栈指针。代码并不多，尤其是如果函数采用一个参数的情况下，此时，一字节的 `POP` 指令便可以重新设置堆栈指针。但是，当采用更多的函数调用时，清除堆栈的指令便会对程序少许增加一些开销。

`_fastcall` 约定改进了调用至少占用一个参数的 C 函数的速度。在这种约定中，前两个适当的函数参数值被传递给处理器寄存器。所有其他参数传递到函数中，方法是它们以从左到右的顺序压入堆栈。在 Intel 系统中，Visual C++ 利用 ECX 和 EDI 寄存器为 `_fastcall` 函数传递参数。尽管 ECX 和 EDI 寄存器自从 C 7 时代以来已经被 `_fastcall` 所应用，但微软公司不能保证将来的 Visual C++ 发行版本将会继续使用同一寄存器（这种警告仅适用于含有内嵌汇编代码的 `_fastcall` 函数）。除了提高运行速度外，`_fastcall` 约定通常会稍微减少代码大小，约定的唯一不利之处就是，它不允许变量参数列表。

`_stdcall` 约定通过 Windows API 来调用约定，当应用给有固定参数列表的函数时，`_stdcall` 类似于 `_fastcall` 约定，只是它不能向寄存器中传递参数。这个约定有利于减少程序代码大小，因为堆栈清除的义务属于被调用的函数，而不是调用程序。在 `_fastcall` 和 `_stdcall` 下，函数通过 RET（返回）指令可以有效地清除堆栈，而无需显式地调节 ESP 堆栈指针。`_stdcall` 约定也允许函数的变量参数列表，在这种情况下，调用以与 `_cdecl` 一样的方式实现，强迫调用程序清除堆栈。

运行库

选择正确的运行库，能减少应用程序的代码大小，尽管通常没有必要去调整默认设置。不要误解 Multithreaded DLL 和 Debug Multithreaded DLL 的设置，设置中的“DLL”指的是运行库，而不是项目，并且，它并不意味着运行时仅仅应用于创建动态链接库的项目。Multithreaded DLL 设置将项目与用于 Msvcrt.dll 的导入

库链接起来，这是一个可重新分布的动态链接库，它含有 C 运行库的版本，而且可以处理线程。动态或静态地与运行库链接，与动态或静态地与 MFC 库链接所需考虑的问题相同。静态链接使可执行文件变得更大。动态链接使得代码更小，但是，要求用已完成的应用程序来重新分配 Msvcrt.dll。

在表 12-4 中，总结了 Code Generation 分类中用来指定项目运行库的 Use Run-Time Library（使用运行库）框中的设置：

表 12-4 Use Run-Time Library 框中的设置，它确定程序如何与 C 运行库链接

设置	运行库	说明
Single-Threaded（单线程）	Libc.lib	与库的静态链接，单线程
Multithreaded（多线程）	Libcmt.lib	与库的静态链接，单线程
Multithreaded DLL	Msvcrt.dll	用于 Msvcrt.dll 的导入库
Debug Single-Threaded（单线程调试）	Libcd.lib	静态链接，单线程（调试版本）
Debug Multithread（多线程调试）	Libcmtd.lib	静态链接，多线程（调试版本）
Debug Multithreaded DLL（调试多线程 DLL）	Msvcrt.dll	用于 Msvcrt.dll 的导入库

结构对齐

在 Code Generation 分类中，最后的设置指定了结构和联合成元对齐的边界。在结构的第 1 个成员后，随后的每个成员落在内存边界上，这个边界是按成员的大小或按对齐的设置来决定的，取其中的最小值。设定结构对齐值为 1，可以保证在结构成员之间没有浪费的内存，这就是著名的压缩结构的技术。

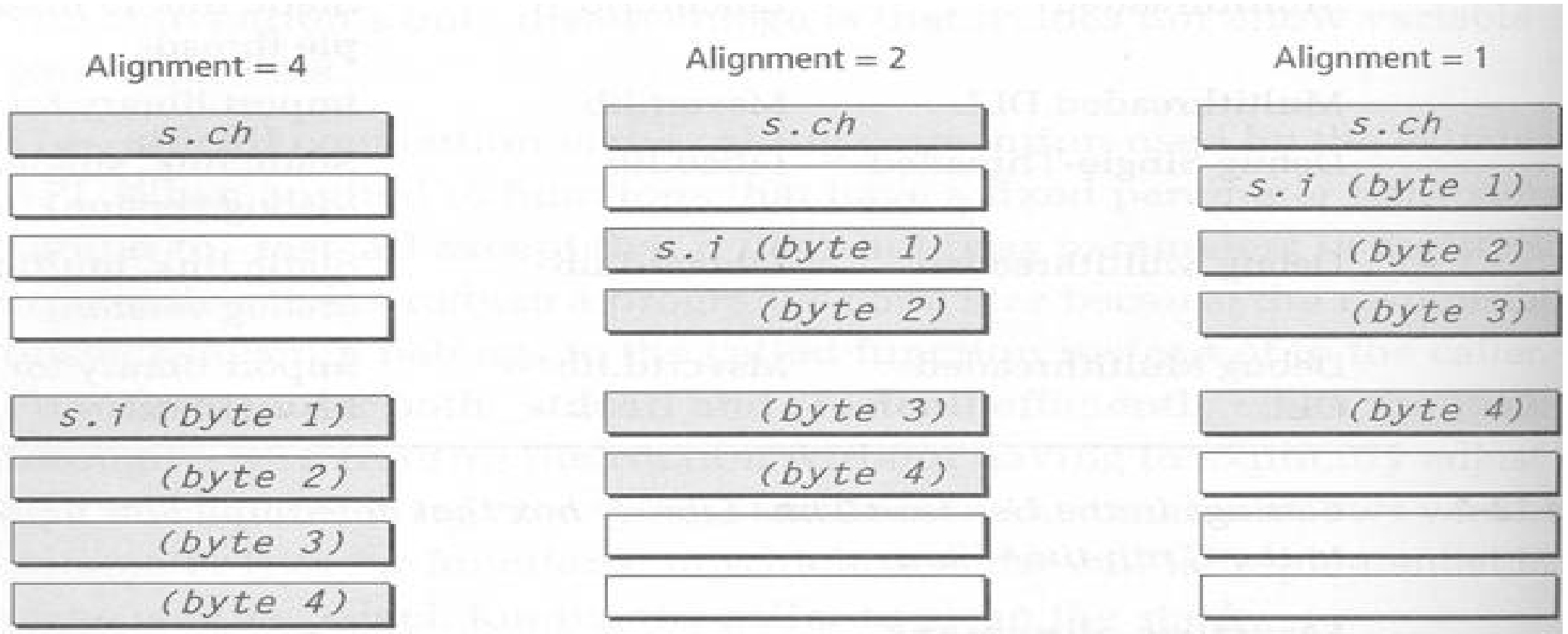
压缩能减少自动内存类结构的堆栈使用，或减少静态存储类的结构的程序大小。对压缩来说，要起作用，一个结构就必须有至少一个元素长度仅为 1 或 2 个字节，而且放置在一个更大的多字节成员之前，如一个整数。例如，考虑压缩对这个简单结构的影响。

```
struct s
{
    char ch;      //One-byte element

    int I;        //Four-byte element
}
```

对齐值为 4 或更多，将会在两个成员之间浪费 3 个字节，原因是编译器将成员 i

置于一个双字边界上。然而，对齐值 1 会将 i 压缩 ch 邻近内存的位置：



尽管压缩能够减少结构的大小，这种节省可能并不会使程序数据区域全面减少。它取决于结构里的成员的混合状况，以及内存中结构的数据目标。如果一个整数在结构之后出现在内存之中，例如，编译器在 `s.i` 之后，将整数与下一个双字边界对齐，因此，浪费的字节通过压缩结构便可以节省下来了。

结构压缩需要运行速度方面的代价，因为处理器当从内存中读取不对齐的数据时

便会有时间上的拖延。在 Intel 80486 和奔腾处理器上，均能够在在一个内存引用周期里取回一个 4 字节的整数，整数便安排在双字对齐边界上。如果整数的位置与它的优化边界有偏移，处理器在取数据时就必须等待 3 个附加的周期。如前所述，在存储整数 `s.i` 时，对齐设置 1 或 2 可以节省空间，但是，代价是当读写入整数时，访问时间要增加 4 倍。

对动态链接库和组件软件，诸如 ActiveX 控件，结构压缩也会导致出现一些小问题。如果库导出一个函数，这个函数采用结构作为参数，或者返回结构指针，调用应用程序和导出函数两者都必须对齐结构。看一下前面的图，将会使你确信，当调用应用程序使用 4 或更高的对齐设置来编译，试图将结构 `s` 传递给已经用对齐设置 1 或 2 编译的动态链接库时，便会产生问题。当发生这种情况时，调用程序和库对整数 `s.i` 找到的内存位置是不同的。

如果你的库将结构或指针传递给一个结构，应当更多地考虑压缩所带来的后果。Visual Basic 假定结构对齐在 WORD 边界上，因此，为了成功地共享一个结构，对齐设置 2 对库和 Visual Basic 调用程序来说是必须的。当在一个头文件中声明此结构时，也必须考虑使用 `pack` 关键字。接着，使头文件对于调用你的库来编写 C/C++ 应用程序的开发人员都可用。用相同的头文件编译库和调用程序来确保结构对齐的一致，而不用管在 Project Settings 对话框中的选项。

```
#pragma pack( push, PACK_S )           // Save current alignment setting
```

```
#pragma pack( 2)                       //Word-align the structure
```

```
struct s  
  
{  
  
    char ch;  
  
    int  I;  
  
};  
  
#pragma pack( pop, PACK_S )           //Restore original alignment
```

自定义分类

Customize 分类（图 12-4 中）控制启用函数级别的链接和重复字符串的消除（字符串共用）的优化。两种优化都是 Maximize Speed（最大速度）选项中的一个整体部分。如果在 General 分类中设置 Maximize Speed，标有 Enable Function_Level Linking（启用函数级链接）和 Eliminate Duplicate Strings（消除重复字符串）的复选框在 Customize 分类中便被禁用。这似乎也指出了那些优化同时也失去作用，但并不是这样，选择 Maximize Speed 将打开两种优化。为了使复选框发生作用，首先在 General 分类中选择 Minimize Size 或 Customize，如前面所述。

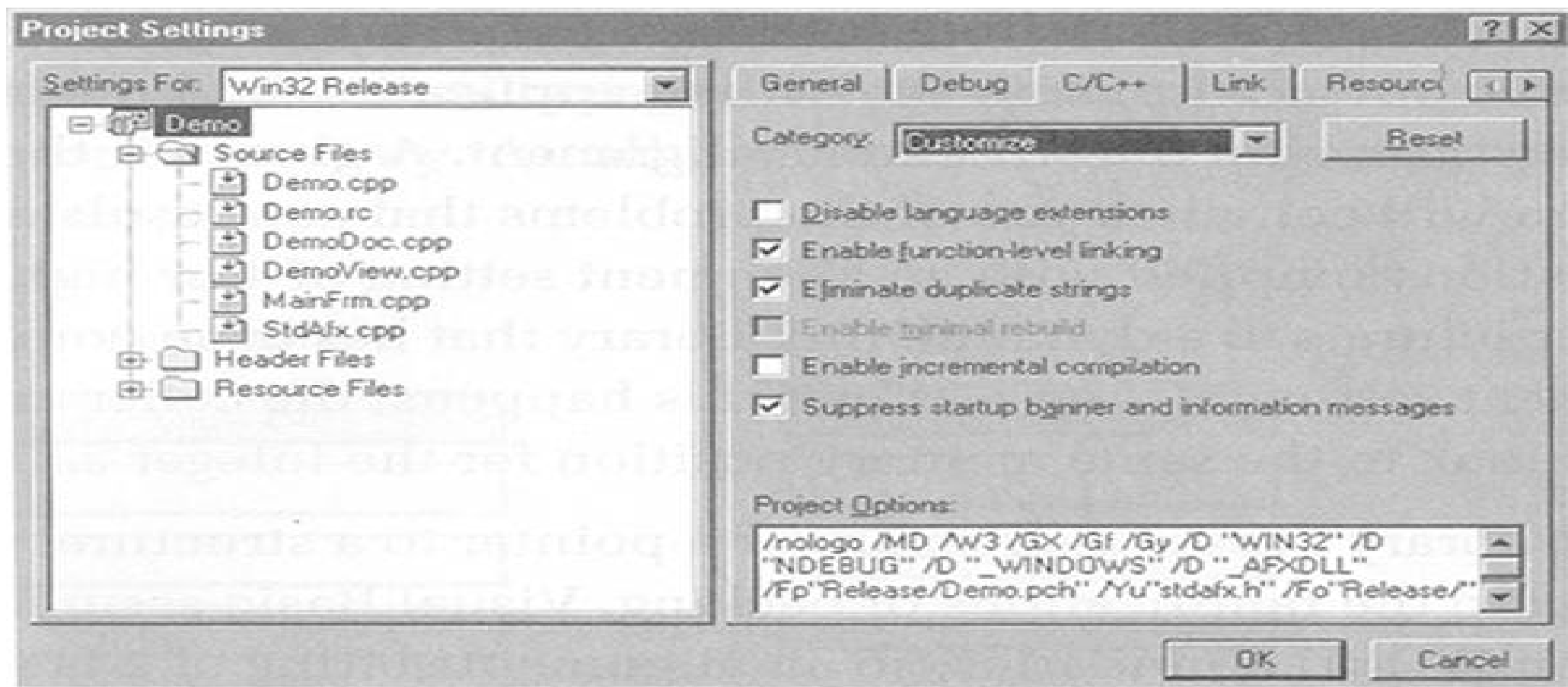


图 12-4 Customize 分类中的选项

函数级别的链接仅仅应用于封装的函数。也就是说，函数通过目标列表中的一个 COMDAT 记录向链接器进行标识。在 C/C++ 类声明中的内嵌成员函数是自动封装的，而别的成员函数不是这样。为了编译所有封装形式的函数，Enable Function-Level Linking（启用函数级链接）复选框必须打开（或者，如果选择

Maximize Speed 设置，则仍禁用)。

优化分类

Optimizations(优化)分类为用于项目中的优化类型提供更细微的控制，并且允许你指定编译器是否应该扩大函数内嵌。此分类显示 General 分类中所选择的同样的优化设定，这些设置必须进行自定义，来启用图 12-5 中所示的复选框，这将允许你从编译器优化列表中选择。Customize 设置为打开 Assume No Aliasing 优化提供了唯一的方法。

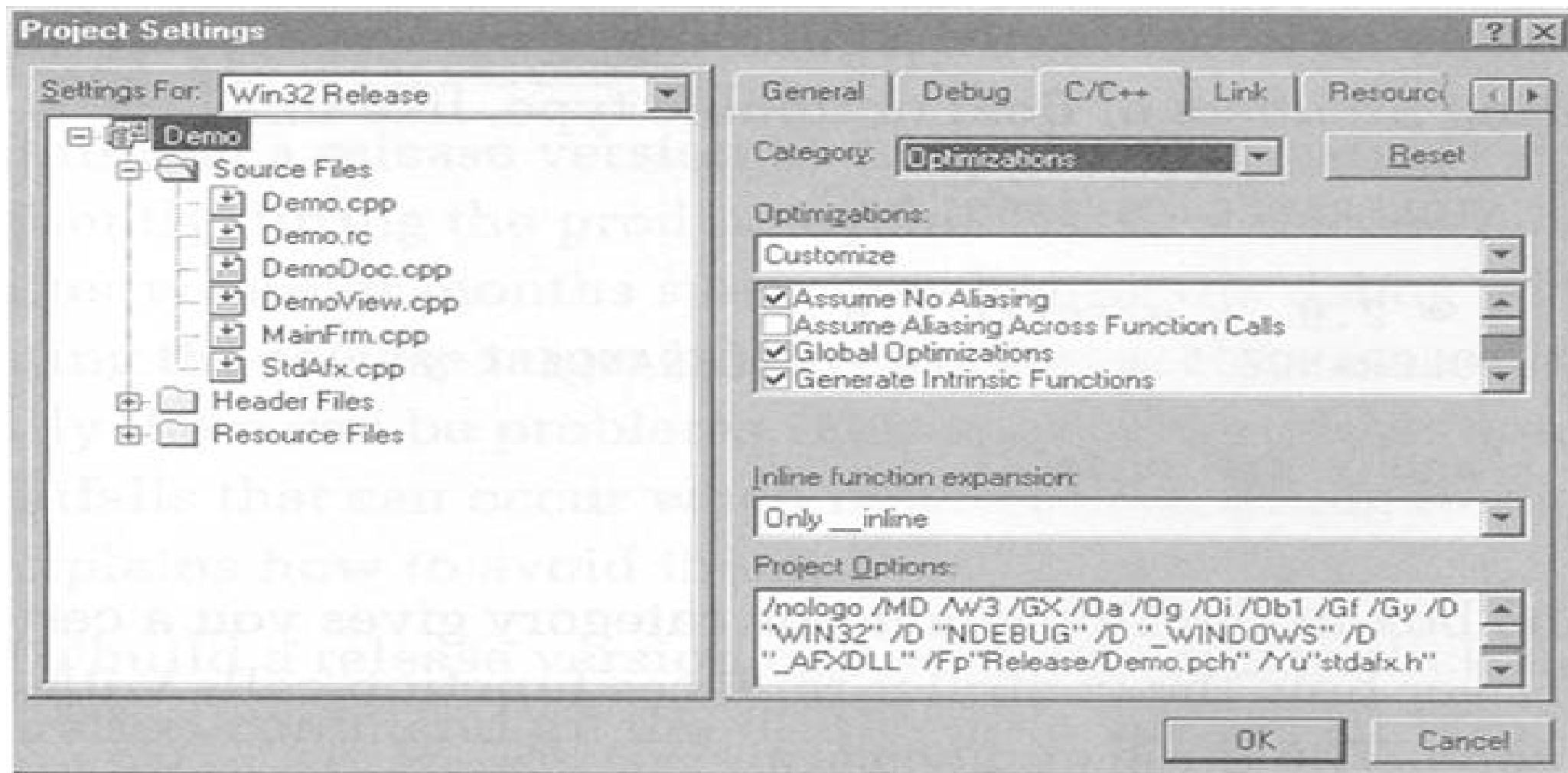


图 12-5 Optimizations 分类中的选项

在此列表中，最后的复选框开关标有 Full Optimization（完全优化），它打开一系

列优化，包括内嵌扩充、内在函数、支持快速代码、无堆栈检查以及全局优化。在此列表中，可能只需要解释标为 Improve Float Consistency（改进浮点一致性）的复选框。这个开关当被关闭时实际上是一种优化。它的代价是需要更多的代码和更慢的浮点操作，打开开关，会使编译器采取下列步骤来减少浮点舍入错误的机会。

- 在每个浮点操作之前添加指令，从内存中复制数据到浮点寄存器上。尽管这将会在很大程度上减慢操作，但是计算结果可以保证得到数据类型所能容纳的精确数据。
- 禁用执行浮点运算的运行函数内嵌的内在形式，在表 12-3 中已列出，程序使用标准的运行函数。
- 禁用这样的计算优化，它允许计算结果为浮点处理器的 80 位精度。

这些步骤把浮点计算的结果维持在 32 位或 64 位精度，有利于确保两个浮点数字能更准确地测试其是否相等。而且，即使打开了 Improve Float Consistency（改进浮点一致性）开关，它仍然是一个好主意，在比较 float 型和 double 型数字时，可以容许少量的偏差，例如：

```
#define TOLERANCE 0.00001
```

```
double x = 2.0, y = sqrt(4.0 );

if (x + TOLERANCE > Y && x - TOLERANCE < y)

{

    // x and y are equal

}
```

在 Optimizations 分类组合框中，提供了一定数量的控制，它们针对编译器怎样利用等价的内嵌代码来代替函数调用，三种选择如下：

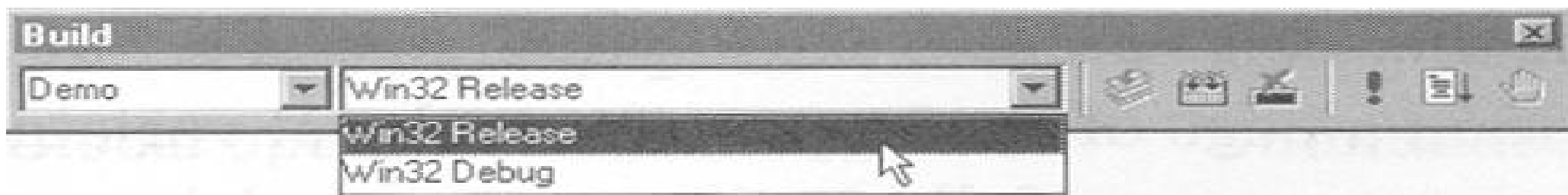
- **Only_inline (只有内嵌的) :** 仅仅替换标有 `_inline`, `inline` 或 `_forceinline` 关键字的目标函数，或者对于类声明中定义的类型成员函数。当优化速度时，编译器使用内嵌代码替代所有这样的函数调用。当 Visual C++ 优化尺寸时，就不完全相同了。如果 Favor Small Code 选项已经起作用，编译器不会扩大标有 `_inline` 或 `inline` 的太大函数。甚至当内嵌大范围使用时，这也会保证得到更好的优化结果。新的 `_forceinline` 关键字超出了编译器的能力，尽管不是在每种情况下均如此。例如，强制采用变量参数列表的函数的内嵌，或者是不能被 `inline recursion` 关键字识别的递归函数。

- **Any suitable (任何适用的)**：除了被 `Only_inline` 设置覆盖的函数外，这种选项也能替代函数的调用，那些函数是编译器认为太小，不足以进行内嵌的函数。微软公司没有为这样的函数说明编译器标准。
- **Disable (禁用)**：没有设置内嵌，即使对调用标有 `_inline` 或 `_forceinline` 的函数。

从调试到发行

在产品的生产周期中，需要不时地建立应用程序的发行版本。第一次发行版本的建立可能要等到编完调试版本以后的数周或数月后。一般来说，创建发行目标仅仅涉及到一个新的建立版本，但偶尔也存在其他一些问题。这节主要讨论当从调试转到发行时，可能产生的潜在缺陷，并解释如何避开它们。

为了建立程序的发行版本，在 Build 菜单中单击 Set Active Configuration (设置活动配置) 命令，并选择 Win32 Release，或者在 Build 工具栏中选择目标。



在 Project Settings 对话框中设置所希望的优化开关，然后单击 Build 命令。你可以注意到，编译程序的发行版本所花的时间，要比编译调试版本要多，这是因为编译器优化时需做更多的工作。

对一个应用程序来说，如果它在调试状态正确运行，在作为发行目标重新编译，根据优化器进行中间过程的处理时，很少会出问题。仅仅在别名的情况下，才可能出错，通常，这是很少见的。隐藏的别名可以在程序员陌生的代码中。如果一个调试应用程序在打开 Assume No Aliasing 选项时失败，则问题出自所存在的隐含别名。通过重新建立发行版本，而且关闭 Assume No Aliasing，可以很容易地检测到这个条件。如果应用程序仍然失败，就不应当责怪优化器。当应用程序从调试版本移到发行版本时，为何一个应用程序可能中断，原因有许多。

例如，ASSERTs 在发行模式中，编译器忽略在 ASSERT 宏中的代码。如果断言的代码调用一个函数，或执行一些在 ASSERT 之外的代码所要求的其他任务，这将会引起一些问题，考虑如下例子：

```
ASSERT ((ptr=GetPointer())!=NULL);
```

```
X=*PTR;
```

在调试时，这种代码能够正确运行。在发行版本里，代码可能会引起错误，因为 `ptr` 没有被初始化。解决方法就是在 `ASSERT` 之前调用 `GetPointer` 函数，或使用 `VERIFY` 宏代替 `ASSERT`。对于使用 `#ifdef _DEBUG` 加前缀的条件代码，也可能发生类似的问题。因为编译器不会预先在发行模式中定义 `_DEBUG`，条件块中的代码不必执行影响块外代码的行为。尽管这看上去很明显，但许多程序员经常犯简单的错误。

禁用堆栈检查也能引起一些问题，对于一个函数来说，它可能需要更多的堆栈空间页，用于其局部变量。尽管在程序的调试版本里，函数可以通过第一次调用一个堆栈检查例程来成功地运行，当优化时，如果禁用堆栈检查，则可能会失败。解决办法就是重写函数，以到达随后页面的堆栈内存，或者，插入一个 `check_stack` 程序，选择性地使堆栈检查对函数起作用。

在调试和发行模式之间，编译器行为可以用其他更微妙的方法来改变。例如，在调试版本里，`new` 运算符对内存分配添加保护字节。无意之中依赖于这些额外字节的程序也许在其发行版本中会运行失败。

函数参数可以用任何次序来求值，而且，必须保证参数顺序在程序的调试版本和发行版本中保持一致。接下来的例子在一种版本中可以正确运行，而在另一种版本中不能：

```
Function1( ptr=Getpointer()),*ptr);
```

隐藏的线程问题有时仅在程序的发行版本中出现，如常见的错误有，两个线程同时访问一个写静态变量的函数。在调试版本里，变量总是保持在内存里，因此，线程间的潜在冲突不会由于时间上的轻微差别而出现。在发行版本里，错误出现的机会会更多，因为变量能为函数运行的持续时间而重新寄存，这将使得一个线程覆盖其他结果更为可能。

一个优化过的程序可能会运行失败，原因可能是许多其他类型的源代码问题，它们中的一些如表 12-5 所示。为了跟踪某一问题，应当尽量单独打开这些优化，以确定错误在什么情形下会出现。表中的第 2 栏提供了一些建议，在检查代码时应该查询这些。

表 12-5 从代码优化中产生的典型源代码问题

优化	可能引起的问题
内嵌扩展	未初始化的局部变量
全局优化	未初始化的局部变量
产生固有的函数内嵌	未初始化的局部变量
改善浮点一致性	比较时依赖于确切的精度
结构指针的忽略	由于不正确的函数原型而引起堆栈破坏

在发行版本出问题时怀疑优化器，这可能是人类的本性。毕竟，编译器用我们不熟悉的方法来重写代码，但是，在 Visual C++ 编译器里，代码优化技术的可靠性相当高。微软公司在自己的主要产品里，例如 Windows 95, Windows NT 和 Microsoft Office，使用 C/C++ 语言编写它们，并进行优化，这体现了微软公司具有足够的自信。单凭这个事实，就应该减轻人们对优化不安全的担心。

基准 Visual C++

当 Visual C++ 4.0 版本处在 β 测试阶段时，微软公司请我去执行新产品的基准测试，并写一篇讨论基准结果和方法的文章。此测试将 Visual C++ 与三种合格的产品相比较，以查看对于相同的源代码，哪种编译器将产生最快的或最小的可执行代码。得到的结果将是每个编译器能力的可靠性量度，由此可以知道如何最好地优化源代码。

Visual C++ 在基准测试中做的较好，可以说是相当不错，实际上，这并不是本节的要点。然而，它可以使我们得知，要查看基准测试代码中的单个函数，将会涉及到复杂的数字计算。这种特殊的函数，代表了在基准测试中，结果将是各不相同的，它生动地演示了一些聪明的编译器优化可以获得的潜在好处。尽管所有 4 个编译器都进行设置，来为函数的最大速度进行优化，在此部分的测试中，Visual C++ 产生的可执行程序的运行速度要比占第 2 位的代码速度快 3 倍多，从反汇编代码中可以得知原因何在：

Microsoft Visual C++ 4

最接近的竞争对手

?Fn x9@@YAXXZ PROC NEAR

?Fn x9@@YAXXZ:

sub esp, 8



xor edx,edx

push ebx

push esi

push edi

\$L1220:

mov ebx, DWORD PTR ?X@@@3PAVTest@@A [edx]

push EBP

mov EBP, ESP



sub ESP, 038h

push EBX

push ESI

mov dwod ptr -038h[EBP], 5

mov dword ptr -034h[EBP], 8

续表

```
mov     edi, DWORD PTR ?X@@3PAVTest@@A [edx+4]
```

```
lea     edi,  DWORD PTR ?Y@@3PAVTest@@A [edx]
```

```
add     edx, 8 
```

```
lea     eax, DWORD PTR [ebx+ebx*4]
```

```
lea     ecx, DWORD PTR [edi*8]
```

```
sub     eax, ecx
```

```
mov     ecx, DWORD PTR ?Y@@3PAVTest@@A [edx-4]
```

```
add     eax, DWORD PTR [esi]
```

```
mov     DWORD PTR $T1455[esp+20],eax
```

```
lea     eax, DWORD PTR [ecx+ebx*8] 
```

```
mov     ESI,offset FLAT:?Y@@3QAVTest @@A
```

```
mov     EBX,offsetFLAT:? X@@3QAVTest @@A
```

```
mov     EDX, -034h[EBP]
```

```
mov     EAX, -038h[EBP]
```

```
mov     -020h[EBP],EAX
```

```
mov     -01ch[EBP],EDX
```

L352:

```
mov     EDX, 4[ESI]
```

```
mov     EAX, [ESI]
```

```
mov     -010h[EBP],EAX
```


续表

```
lea    ebx,DWORD PTR [edi+edi*4]
```

```
mov    edi,DWORD PTR $T1455[esp+20]
```

```
add    eax,ebx
```

```
mov    DWORD PTR [esi],edi
```

```
mov    DWORD PTR [esi],edi
```

```
mov    DWORD PTR [esi+4],eax
```

```
cmp    edx,8000
```

```
jl     SHORT $11220
```

```
pop    edi
```

```
pop    esi
```

```
mov    -0Ch[EBP],EDX
```

```
mov    EDX,4[EBX]
```

```
mov    EDX,[EBX]
```

```
mov    -018h[EBP],EAX
```

```
mov    -018h[EBP],EAX
```

```
mov    -014h[EBP],EDX
```

```
mov    EDX,-020h[EBP]
```

```
imul   EDX,-018h[EBP]
```

```
mov    EDX,-01Ch[EBP]
```

```
imul   EDX,-014h[EBP]
```

续 表

pop ebx

add esp, 8



ret 0

sub ECX, EDX

mov -030h[EBP], ECX

mov ECX, -020h[EBP]

imul ECX, -014h[EBP]

mov ECX, -01Ch[EBP]

imul ECX, -018h[EBP]

add ECX, EDX

mov -02Ch[EBP], ECX



mov EDX, -02Ch[EBP]

mov EAX, -030h[EBP]

续 表

mov	-8[EBP], EAX
mov	-4[EBP], EDX
mov	ECX, -010h[EBP]
add	ECX, -8[EBP]
mov	-028h[EBP], ECX
mov	ECX, -0Ch[EBP]
add	ECX, -4[EBP]
mov	-024h[EBP], ECX
mov	EDX, -024h[EBP]
mov	EAX, -028h[EBP]

续表

mov [ESI],EAX

mov 4[ESI],EDX

mov ECX, 8

add ESI, ECX

add EBX, ECX

cmp EBX,offset FLAT:??@3QAVTest@@A

jb L352

pop ESI

pop EBX

mov ESP, EBP

续表

pop EBP



ret

除了在大小上的明显不同外，在序言和结尾部分中，标有☞和✂的两个列表之间的差异也是显而易见的。通过结构指针的忽略进行优化时，Visual C++产生的代码分配 8 字节堆栈，并且，通过使用 ESP 寄存器的偏移量来访问函数的堆栈结构。其他的编译器，不能提供堆栈指针忽略作为优化技术，必须压入和弹出 EBP 寄存器，来使它更适于用作结构指针。而且，再进一步看，对于此函数来说，揭露堆栈结构甚至是没有必要的，而且，Visual C++失去了优化在列表中标记✂的代码的机会。虽然 EBP 寄存器因为结构指针的忽略现在是空闲的，代码仍然使用堆栈来支持中间计算。这将会浪费掉结构指针忽略所带来的好处：为了在其他优化中使用，便释放 EBP 寄存器。如果写成如下方式，标有✂的序列将会稍快，而且更小一些：

```
mov     ebp, eax                ;Store intermediate calculation

lea     eax,DWORD PTR [ecx+ebx*8] ;With value temporarily saved,

lea     ebx,DWORD PTR [edi+edi*4] ; we can use EAX
```

```
mov     edi, ebp
```

```
;Recover calculation in EDI
```

利用 EBP 来存储中间计算结果，在每次循环中，将可以节省两次内存访问。

Visual C++ 版本可以节省代码空间，并通过组合地址模式为一条指令，来赢得速度。这允许它为简单的算术计算使用 LEA（加载有效地址）指令，这是 Intel 处理器的一个显著特征。通过使用左移和加法运算，LEA 指令能操纵基本的和索引的寄存器，以便执行某种乘法操作，这比处理器的乘法指令 MUL 和 IMUL 指令要快得多。例如，这里有一个 LEA 指令，它能够乘以存在 EAX 寄存器里的值：

指令	说明
lea eax, [eax*2]	$EAX \times 2$
lea eax, [eax+eax*2]	$EAX \times 3$
lea eax, [eax*4]	$EAX \times 4$
lea eax, [eax+eax*4]	$EAX \times 5$
lea eax, [eax*8]	$EAX \times 8$
lea eax, [eax+eax*8]	$EAX \times 9$

尽管用这种方式使用 LEA 由 Intel 公司清楚地编制了文档，其他编译器恢复为 4 个 IMUL 指令，即使它们在一个重复了一千次的循环中发生，其代价是相当昂贵的。从其他编译器产生的循环长度也可以知道，需要使用约 39 个机器指令。这个 Visual C++ 版本的循环方式要求仅仅是 18 个指令，并且根本没有 IMUL 指令。

其他编译器不太在意其使用的寄存器，考虑到从标有 ρ 的代码里取出的序列，在这里，值被写入堆栈中，并且能立即再次获取：

```
mov     -02ch[EBP],ECX
```

```
mov     EDX,-02CH[EBP]
```

因为 ECX 寄存器已经被控制，所以，从内存中再次读取这个值是没有必要的，如果像以下这样，代码序列将会更小：

```
mov     -02CH[EBP],ECX           ;store the ECX value
```

```
mov     EDX,ECX                 ;Also copy it to EDX
```

两个编译器似乎要进行合理的尝试，以避免流水线通过正确的指令顺序时会停顿。人们不应该期望编译器能正确排序，因为它通过代码传递的事情太多了，导致建立时间长得无法让人接受。在搜寻优化指令顺序时，Visual C++编译器仅仅错误了一次，产生了标有⚡的 3 个指令序列：

```
lea     esi,  DWORD   PTR ?Y@@@3PAVTest@@A[edx]
```

```
add     edx,8
```

```
lea     eax,DWORD   PTR   [ebx+ebx*4]
```

第 2 条指令不能改变 EDX 寄存器，直到第 1 条指令已经完成读取它。改变第 2 和第 3 条指令的顺序，可以避免潜在的停止，确保序列中邻近的指令能同时运

行：

```
lea      esi, DWORD PTR ?Y@@3PAVTest@@A[edx]
```

```
lea      eax, DWORD PTR [ebx+ebx*4]
```

```
add      edx, 8
```

对版本 4 来说，这就够了。版本 5 对优化器进行了大量改进，对于个人计算机来说，Visual C++ 确实具有优化的编译器技术。目前的发行版本更进一步进行改进，但是注意力不在新技术上，而是更多地放在了调节编译器的能力上，以便为使代码更小和速度更快寻求机会。微软公司估计，Visual C++ 在版本 4 中，能够减少的可执行代码大小超过 10%，并且，再次编译本节中讨论过的代码，便可以知道，它的改进确实很显著。代码大小缩小约 16%，而且，现在，编译器将指出怎样完全废除堆栈结构，而不依赖于序言代码和结束代码：

```
?Fn x9@@YAXXZ PROC NEAR
```

```
push     esi
```

```
push     edi
```

```
xor      eax, eax
```

\$L1621:

mov edx, DWORD PTR ?X@@3PAVTest@@A[eax+4]

mov ecx, DWORD PTR ?X@@3PAVTest@@A[eax]

add eax, 8

lea edi, DWORD PTR [edx*8]

lea edx, DWORD PTR [edx+edx*4]

lea esi, DWORD PTR [ecx+ecx*4]

lea ecx, DWORD PTR [edx+ecx*8]

mov edx, DWORD PTR ?Y@@3PAVTest@@A[eax-8]

sub esi, edi

mov edi, DWORD PTR ?Y@@3PAVTest@@A[eax-4]

add edx, esi



```

add     ecx, edi

mov     DWORD PTR ?Y@@3PAVTest@@A[eax-8], edx

mov     DWORD PTR ?Y@@3PAVTest@@A[eax-4], ecx

cmp     eax, 8000

jl      SHORT $L1621

pop     edi

pop     esi

ret     0

```

标有代码&的程序段使我们可以更清楚优化编译器的工作。优化器用 3 个步骤写下本代码段，代码从内存中读取两个整数，将它们添加到寄存器的值中，然后，将它们的和写回到内存中，用下面 3 个指令来替代所有标有&的代码：

```

sub     esi,  edi

add     DWORD   PTR    ?Y@@3PAVTest@@A[eax-8],esi

```

```
add    DWORD   PTR    ?Y@@3PAVTest@@A[eax-4],ecx
```

修正的版本所呈现的是，读取操作是没有必要的，并且进一步减少函数的大小约 21%，尽管它不能提高运行速度。添加一个寄存器的值到内存中，比复制这个值要慢得多，但是，其补偿是指令数减少了，因为处理器读取的代码更少。修正后的函数虽然更小一些，但仍会同以往的 Visual C++ 编译器生成的一样，以同样的速度来运行。

如果你在自己的程序里对研究编译器优化的影响有兴趣，Visual C++ 可以产生汇编语言列表，像在本节中所列出的那些一样。在 Project Settings 对话框的 C/C++ 选项卡里，选择 Listing Files（列表文件）分类，然后，选择在 Listing Files Type（列表文件类型）框中所希望的汇编列表类型。

第 13 章 自定义 Visual C++

随着对 Visual C++ 的逐渐熟悉，你会迫切希望改变它的某些特性，以便更好地配合自己的工作风格。可以根据自己的诸多喜好，来改变环境的许多方面，从文本编辑器窗口的内容，到自定义工具栏的外观。我们可以用复杂的宏语言和附加库，来扩充工作环境的很多方面，给 Visual C++ 编制新的功能，而这些功能即便是设计者也不曾想到。本章阐述了自定义 Visual C++ 诸多方法中的几种，采用这些方法，能获得更富效率的工作环境。

大多数环境行为都具有记忆功能，这就意味着，一次只需调整一个设置。其后，这个调整的环境会在下次启动 Visual C++ 时变为默认的环境。例如，退出 Visual C++ 时，如果文本编辑器窗口处于全屏，则在下次运行该程序时，将再次自动出现全屏方式。某些自定义的设置不很明显，如果不知从何处查看，将很难找到它们。

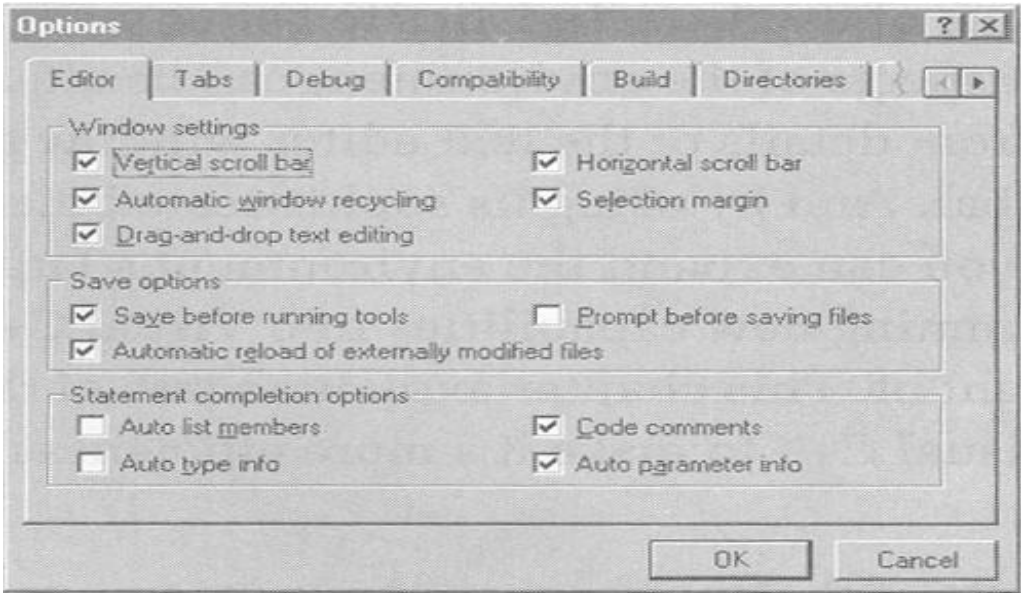
在 Tools（工具）菜单中，有 Options 和 Customize 两项指令，从中，可以直接访问给管理 Developer Studio 环境的行为指令的开关和选项。虽然不能从这些命令中访问所有的定制设置，但这还是一个相当不错的起点。

Options（选项）对话框

如表 13-1 所示，Options 命令显示许多选项卡，这个 Options 对话框中包含着范围广泛的设置集合，包括 Visual C++ 编辑器和调试器的行为及外观、文本编辑器中的制表位间隔和缩进、编译和链接期间中使用的文件和库的目录位置、不同窗口所用的字体类型和文本颜色。表 13-1 列举了一些对话框的选项。

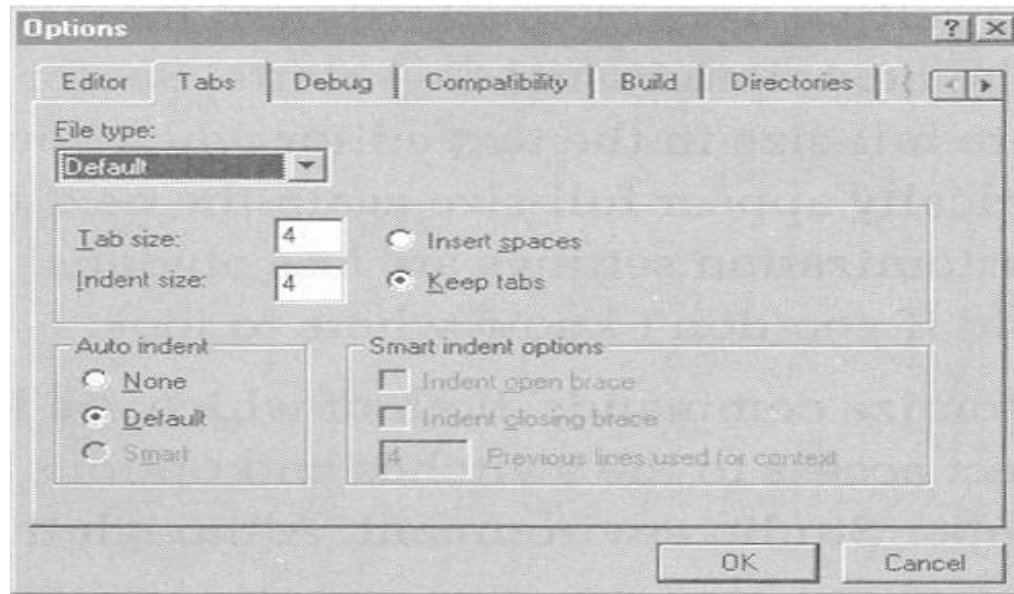
表 13-1 单击 Tools（工具）菜单中的 Options，在调出的 Options 对话框中进行设置

Options（选项）对话框中的选项卡	说明



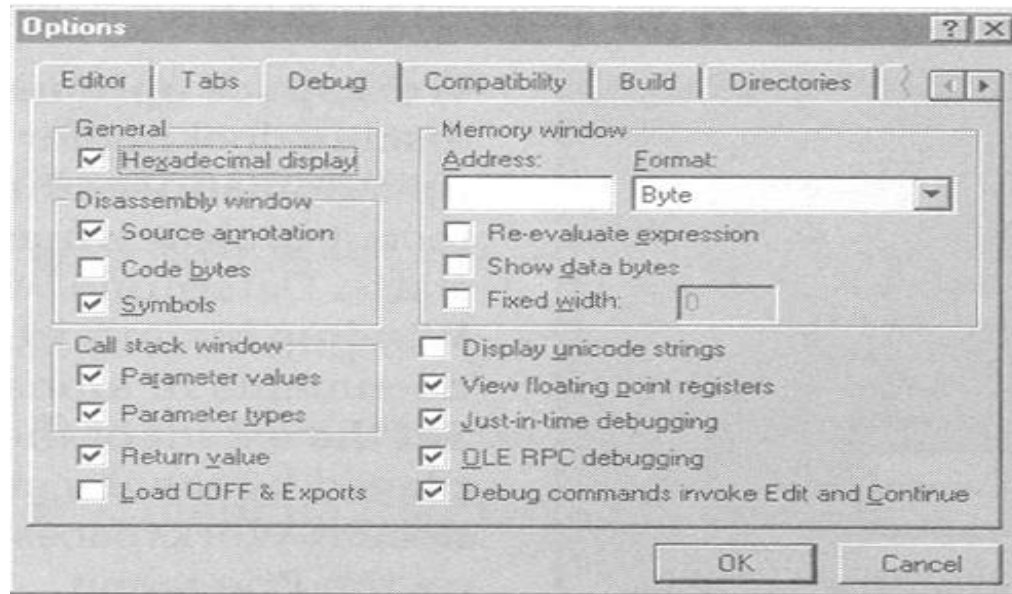
文本编辑器——启用文本编辑器里的滚动条和选择边界。选择影响编译器如何保存文件的设置，以及在文档窗口里是否自动出现语句的自动补全特征

续表



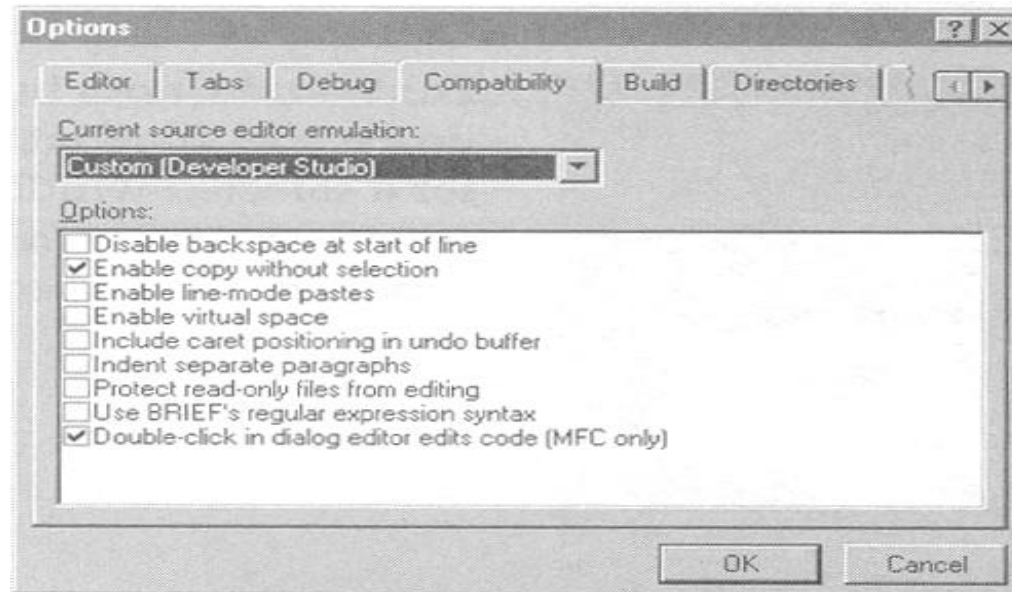
文本档里的制表位位置——在文本档里设置制表位宽度,以及源代码的自动缩进。确定编辑器是否应当接受制表位作为 ASCII 字符,或者,在你键入时,将它们转换为等量的间隔

续表



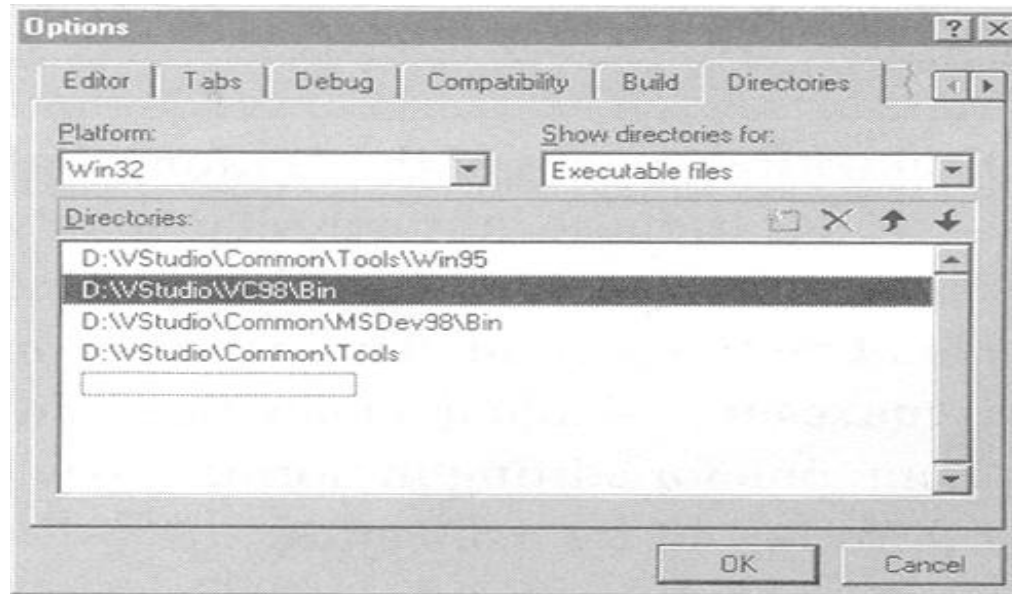
调式器设置——选择 16 进制或 10 进制显示，以及调试器里不同窗口的外型。指定显示在 Memory 窗口里的默认地址，启用即时调试，并在 Edit and Continue 特征的自动或手工应用程序之间选择

续表



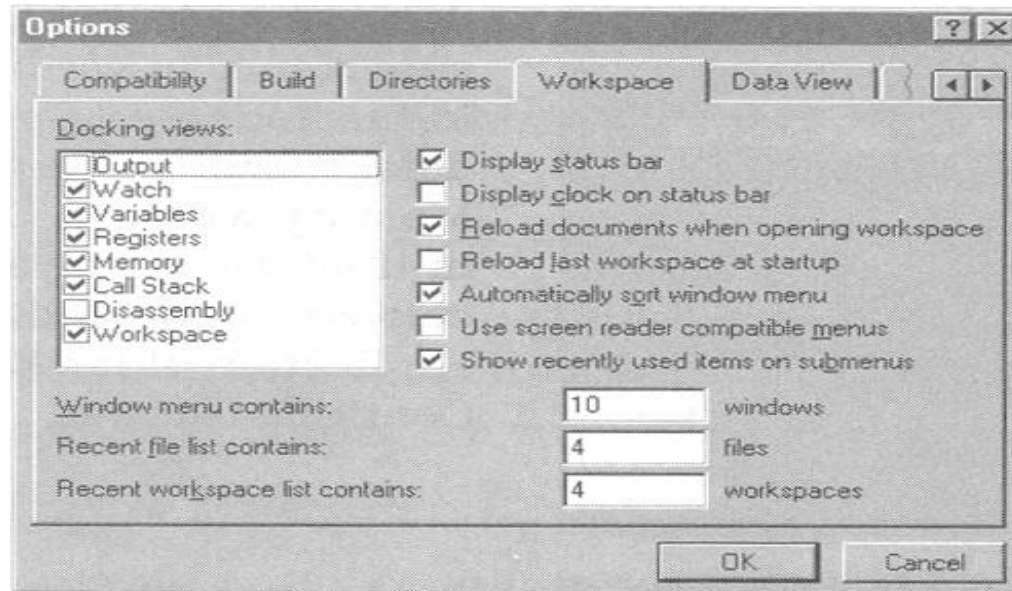
文本编辑器选项——选择
文本编辑器仿真
(Developer Studio, BRIEF,
Epsilon), 启用其他选项,
例如对话框编辑器里双击
的虚拟空间和行为

续表



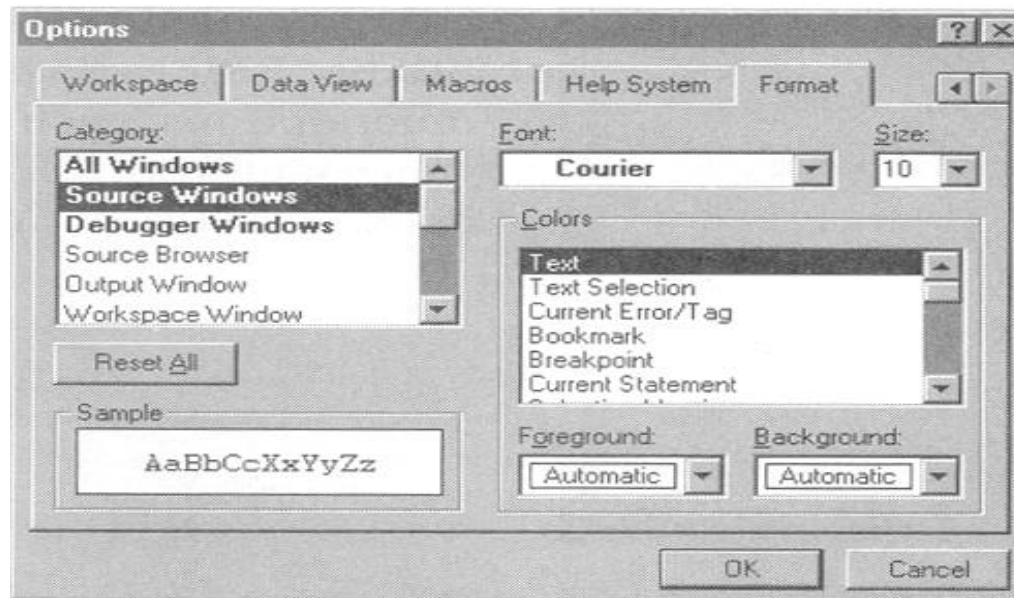
定位文件的目录——选择 Executable files(可执行文件)、Include files(包括文件)、Library files(库文件)或 Source files(源文件), 然后添加或删除文件路径, Visual C++按列出的顺序扫描每条路径来搜索文件

续表



工作空间选项——启用或禁用窗口的停靠，设置 Visual C++ 启动时大部分最近项目中自动加载的内容。决定 Window（窗口）菜单是否按字母顺序来排序文件。通过在 File（文件）菜单中的 Recent Files（最新文件）和 Recent Workspace（最新工作空间）命令，来设置显示的列表的范围

续表



文本窗口里的颜色与字体——为选择的窗口设定字体类型和大小。设定各种文本成员的颜色，诸如注释、关键字和 HTML 标记

Options 对话框中的 Editor, Tabs 和 Compatibility 选项卡仅应用于文本编辑。而且，这些选项卡中的大多数设置同时影响编辑器的标准和全屏幕显示，以及调试器的源文本窗口。另一些设置取决于当前的查看方式，以及调试器是否处于运行状态，这就为在环境中创建三个不同的屏幕布局创建了条件，这三个布局是：一个用于正常显示时的编辑，另一个全屏显示，第三个用于调试。

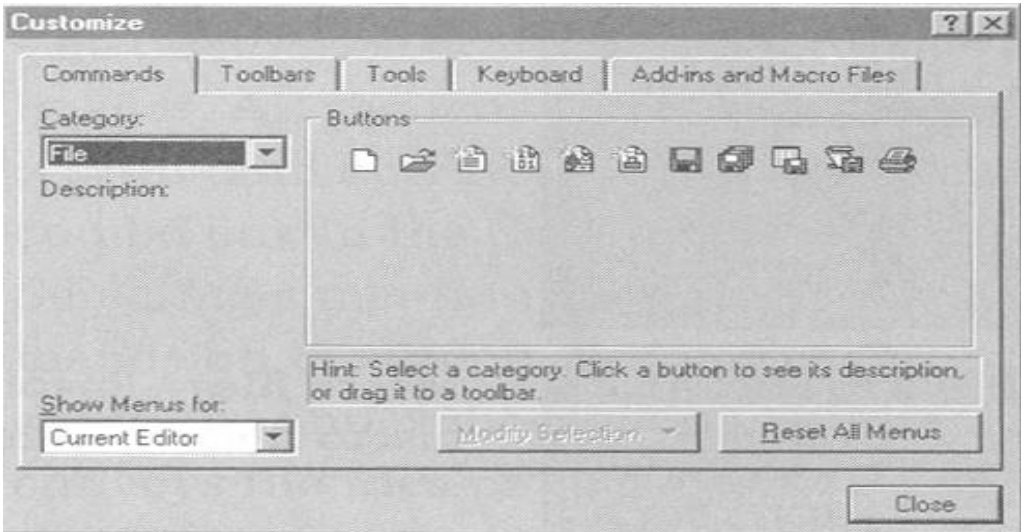
Customize（自定义）对话框

表 13-2 中是 Customize 命令所显示的对话框，它们可以做如下事情：

- 增加或删除菜单命令
- 给菜单命令增加图标
- 打开或关闭工具栏
- 增加或删除工具栏按钮
- 创建命名的敲键命令

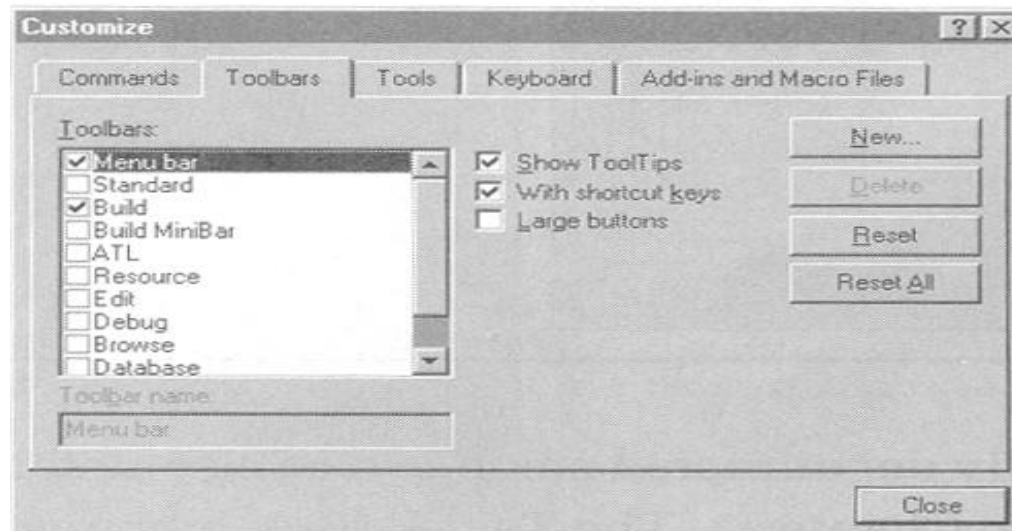
表 13-2 Customize（自定义）对话框中的设置，通过单击 Tools（工具）
菜单中的 Customize 来调出

Customize（自定义）对话框里的选项说明卡



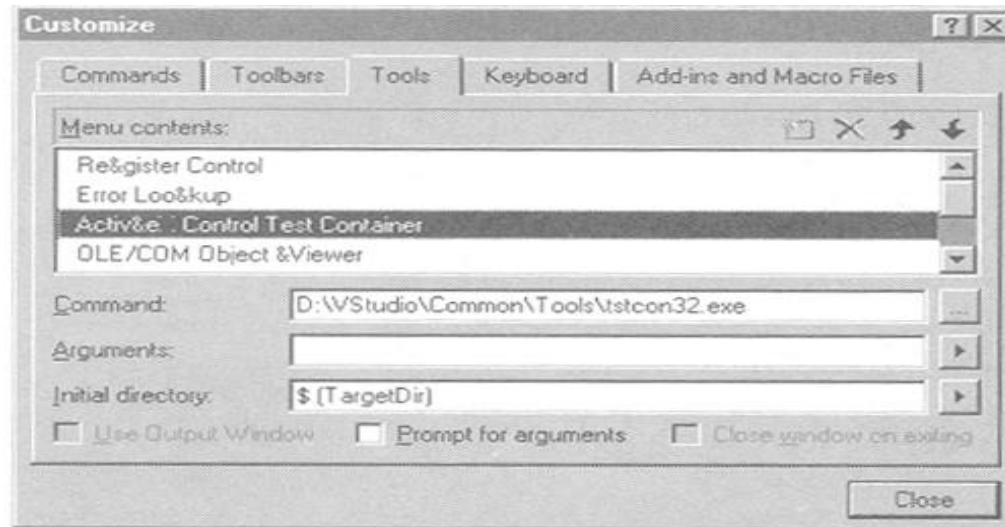
添加命令到菜单——更改一个菜单命令或恢复默认菜单

续表



工具栏——使工具栏可视或不可视，对工具栏按钮启用工具提示，在工具提示消息中显示快捷键，以及选择普通或较大的工具栏按钮

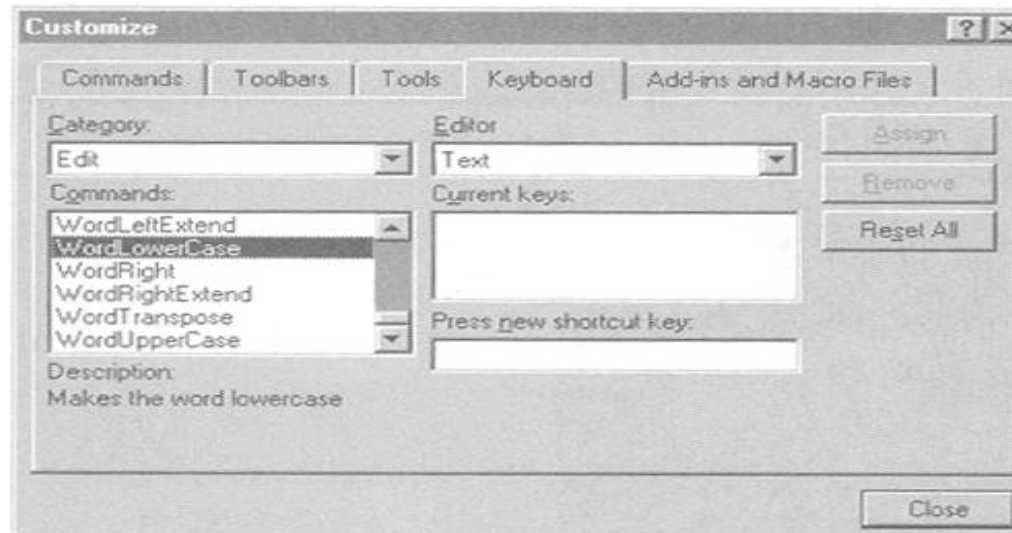
续表



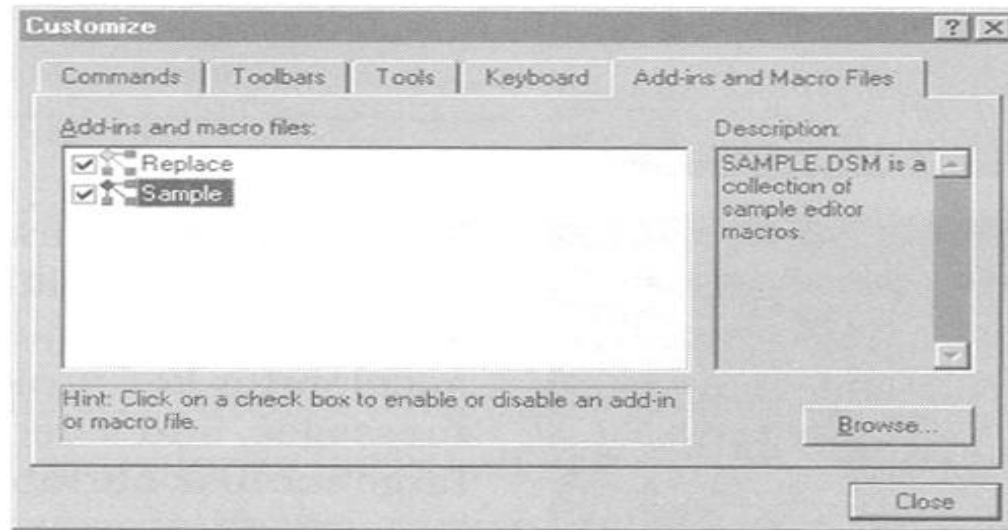
工具菜单命令——添加或删除用于实用程序的命令，指定路径、文件名，以及每个程序的命令行参数。指定实用程序的初始目录

续表

未结合的键盘命令——给
Visual C++命令指定新的敲
键



续表



宏文件——启用或禁用宏文件或附加实用程序

本书在其他几章中已简要地叙述了这些命令，例如，第 3 章中演示了如何为名为 WondUpperCase 和 WondlowerCase 的两个未结合的命令指定敲键。

Customize 对话框的 Commands 选项卡控制着环境菜单的内容，当屏幕上显示 Customize 对话框时，可显示菜单，但菜单中的每个单独命令并不处于有效状态。相反，这个环境作为一个菜单编辑器，可以在其中增加或删除菜单命令，改变命令顺序，以及为现存命令增加图标。

这儿有一个给 File 菜单上的 Page Setup 命令增加图标图像的例子。第一步是调出一个合适的图像，并存在剪贴板中。可以使用屏幕捕获程序，或者在 Visual C++

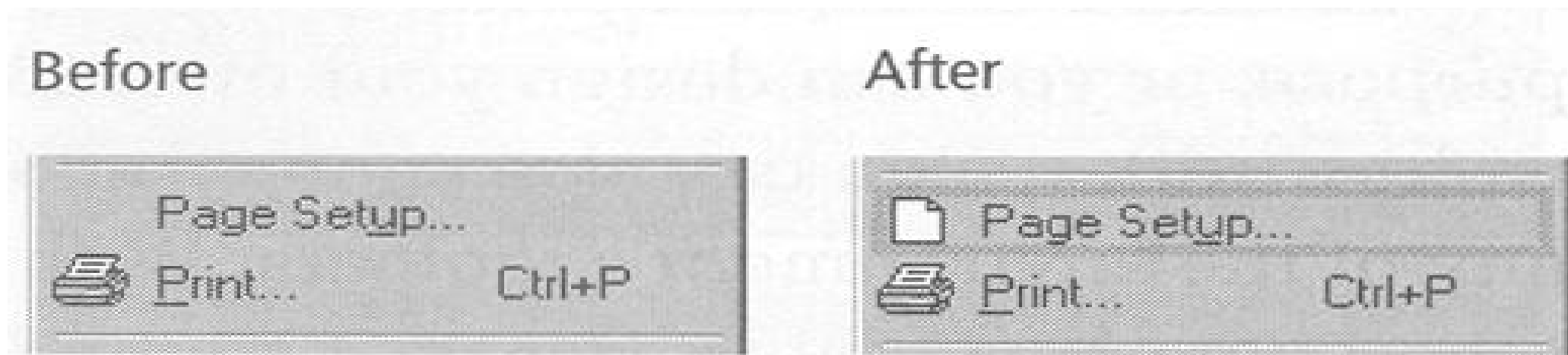
图形编辑器中，自己设计一个 16×16 的图像。当然，也可以从任何环境工具栏中捕获图标图像到剪贴板，使用 Customize 对话框的 Command 选项卡，右击工具栏按钮，显示上下文菜单，如图 13-1 所示，菜单上的 Copy Button Image（复制按钮图像）命令将按钮的图标图像复制到剪贴板。

Customize 对话框本身作为按钮图像的便利来源，能显示 Visual C++ 工具栏中的所有图像，并借用它们。首先在 Commands 选项卡中单击 Category 组合框，屏上显示一个下拉菜单，在表中选择一个菜单，为菜单命令显示一个小图标集。例如，从 Category 下拉表中选择 File，并右击表中的任何一个图标图像，而且，这个图标能够表达出页面设置的意思。所显示的上下文菜单与图 13-1 所显示的相同，从中，我们可选择 Copy Button Image（复制按钮图像）命令，将图像复制到剪贴板。



图 13-1 当 Customize（自定义）对话框可视时，右击菜单命令或工具栏按钮，将显示这个上下文菜单

只要剪贴板存有 16×16 图像，就可以在 File 菜单上将其转换成 Page Setup 命令。在 Commands 选项卡中，在菜单栏中单击 File 显示菜单，然后右击 Page Setup 命令显示上下文菜单，如图 13-1 所示。用 Paste Button Image（粘贴按钮图像）命令，将新图标图像放置在 File 菜单 Page Setup 命令的左侧：



如果想改变结果，有两种方法可撤消前面的工作。第一种是在 Customize 对话框中，单击 Reset All Menus（重置所有菜单），也就是将全部菜单命令全部恢复为初始状态，如果仅仅只想删除个别菜单命令中的图标图像，右击命令，并从上下文菜单中选出 Text Only（仅对文本）。另外，要使用这种方法，Customize 对话框必须显示出来。

用户可以为菜单工具栏本身增加图标图像，右击菜单的标题显示上下文菜单，与前面一样，选择 Paste Button Image（粘贴按钮图像）。下图就是一个定制的菜单

栏，在 Visual C++ 环境下，从不同位置中调出图标图像而创建的：



要给环境菜单增加新命令，仅需两步就能完成：

1. 当 Customize 对话框的 Commands 选项卡显示在屏上时，下拉菜单。
2. 从 Category 下拉列表中选出需要的工具组，并将工具图标或命令名称从对话框拖到菜单中，会有一个水平放置栏指出菜单的位置。

删除菜单上的新命令时，右击命令，并从弹出菜单中选击 Delete（删除），同时可为宏和未结合命令增加菜单项，从 Category 下拉列表中选出 All Commands（所有命令），并在表中定位命令名，然后，将命令名从表上移到菜单中的理想位置，图 13-2 显示了给 Edit 菜单增加 WordUpperCase 命令的过程。

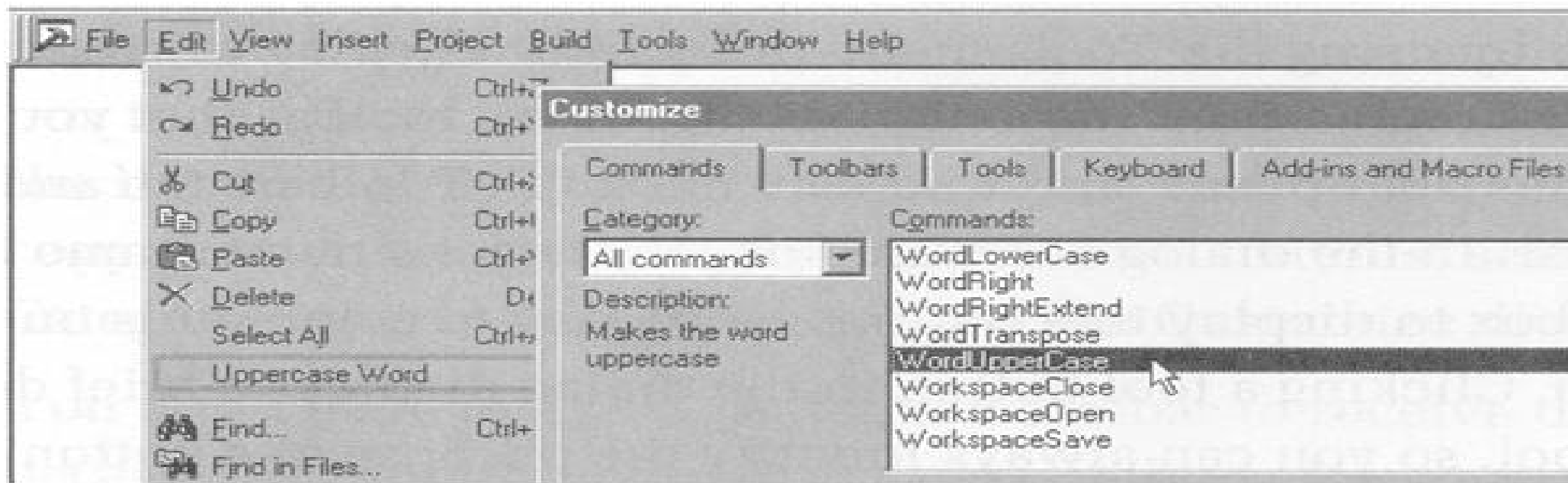


图 13-2 将一条新命令放在一个菜单里

工具栏

工具栏在屏幕里的显示取决于运行的编辑器，以及编辑器是否处于全屏状态。默认情况下，在全屏方式中时，工具栏仍处于不可见的状态；要使工具栏可视，先打开全屏查看，并按住 Alt+T 键下拉 Tools 菜单，单击 Customize 命令，以及 Toolbars 选项卡，然后选择表中所需工具栏旁边的复选框。用这种方法，也可以在全屏方式中显示菜单栏。因为是在全屏方式进行这种改变，工具栏的外观以及

在屏幕中的位置仅适合于全屏查看。按 `Esc` 键时，屏幕返回到正常的查看方式，工具栏也返回到原来位置，而且，它甚至可能是不可见的。

如果想要工具栏浮动，而不靠着另一个窗口或屏幕边缘，在工具栏中双击任何空白区。要使一个浮动工具栏返回到原来停靠的位置，双击工具的标题栏即可。在第 1 章中我们提到过 The Environment 工具栏，在拖动浮动工具栏时，按住 `Ctrl` 键，以防它停靠在另一个窗口。当拖动工具栏到停靠位置时，按住 `Shift` 键，可以在水平和垂直方向之间切换工具栏窗口。

Visual C++ 能轻而易举地改变工具栏的外观和内容，故能将工具栏按钮从一个工具栏复制到另一个工具栏上，并保存使用的工具所在的工具栏中。首先，显示 `Customize` 对话框，在这里，可用 `Tools` 菜单，或右击工具栏，从它的上下文菜单中选出 `Customize`。如果想要修改的工具栏没有显示在屏幕上，单击一下 `Toolbars` 选项卡，工具栏就会显示出来。在对话框的 `Commands` 按钮中，从 `Category` 框中选择一个菜单名，来显示图示所选菜单上属于命令的工具按钮。单击对话框中的工具按钮，来显示关于这个工具的简短描述，从而确定按钮的作用。为确保目标工具栏和 `Customize` 对话框不会在屏幕上重叠，可将想要的工具从对话框拖到工具栏中，只要用户乐意，也可在环境菜单栏中设置工具按钮。

如同菜单命令一样，当 `Customize` 对话框显示在屏幕上时，工具栏按钮不再起作用。这就要求用户拖动工具栏按钮，并将它放在相同工具栏的不同位置，或拖动这个按钮到不同的工具栏。当然，也可通过左右移动按钮，在两个按钮间插入约

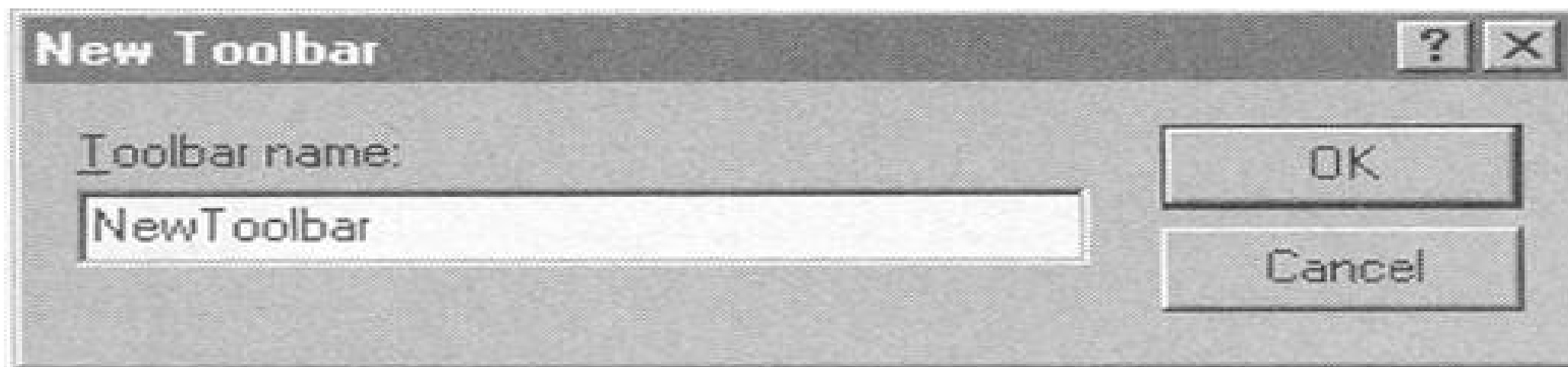
半个按钮宽度的空间。完成上述过程后，单击对话框中的 Close 按钮。实际上，在把一个工具从一个工具栏移到另一个工具栏中并不需要 Customize 对话框。显示工具栏，按住 Alt 键，将想要的工具按钮从一个工具栏移到另一个工具栏中，当然，也可以同时按住 Alt 和 Ctrl 键来复制按钮，而不是移动它。

自定义工具栏

根据个人的口味，如果某些较大的工具栏占据了屏幕较大的区域，我们会希望 Visual C++ 能提供一两个小工具栏，并能将它们放在屏幕角落里，里面只有确实需要的几个工具。要想实现上面的想法，我们可以自定义工具栏，而且这只需要很少的几步就能完成。本书第 3 章曾介绍过自定义工具栏的内容，并用 WordUpperCase 和 WordLowerCase 两个命令进行演示过。下面将前述内容进行补充说明。

首先，根据在何处使用工具栏，来选定标准或全屏的显示方式，然后在 Tools 菜单上单击 Worduppercase 一类的未结合命令，在 Category 框中选出 All Commands（所有命令）来显示命令名称表，在表中确定所要的命令，然后拖动表项到工具栏中。要复制预先规定的环境按钮时，在 Category 表中选出一个菜单，将 Customize 对话框的某个按钮移到工具栏中。

不必用预先确定的工具栏来接收这个新的工具按钮，原因是环境提供了两种方法来创建定制工具栏。第一种方法是在 Customize 对话框的 Toolbars 选项卡中单击 New 按钮，然后给新工具栏命名：



单击 OK 按钮时，Visual C++ 就会创建可存放前述按钮的空白工具栏。

与第一种方法比较起来，另一种方法更易于创建新工具栏，只须从 Customize 对话框中拖动一个命令，并将它放到屏幕上不会被工具栏覆盖住的任何区域即可。对于 WordUpperCase 一类的新命令来说，只要从 Category 列表中选出 All Commands，并从 Commands 表中拖出适当的表项。这个过程和前面将 Worduppercase 存放在菜单上一样。在 Button Appearance（按钮外观）对话框中选择一个按钮图标，增加适当的正文来标识这个按钮，单击 OK 按钮，Visual C++ 就会自动创建一个新工具栏来存放这个按钮；



为了复制已确定的环境命令,可在 Category 表中选出一个菜单名,并从 Customize 对话框中移出一个图标,放在屏幕上的空白区,如图 13-1 所示。这种方法的最大好处在于能同时复制图标图像和按钮正文。要显示新工具栏按钮的图像和正文,右击工具栏中的按钮,并从上下文菜单中选出 Image And Text(图像和文本)。

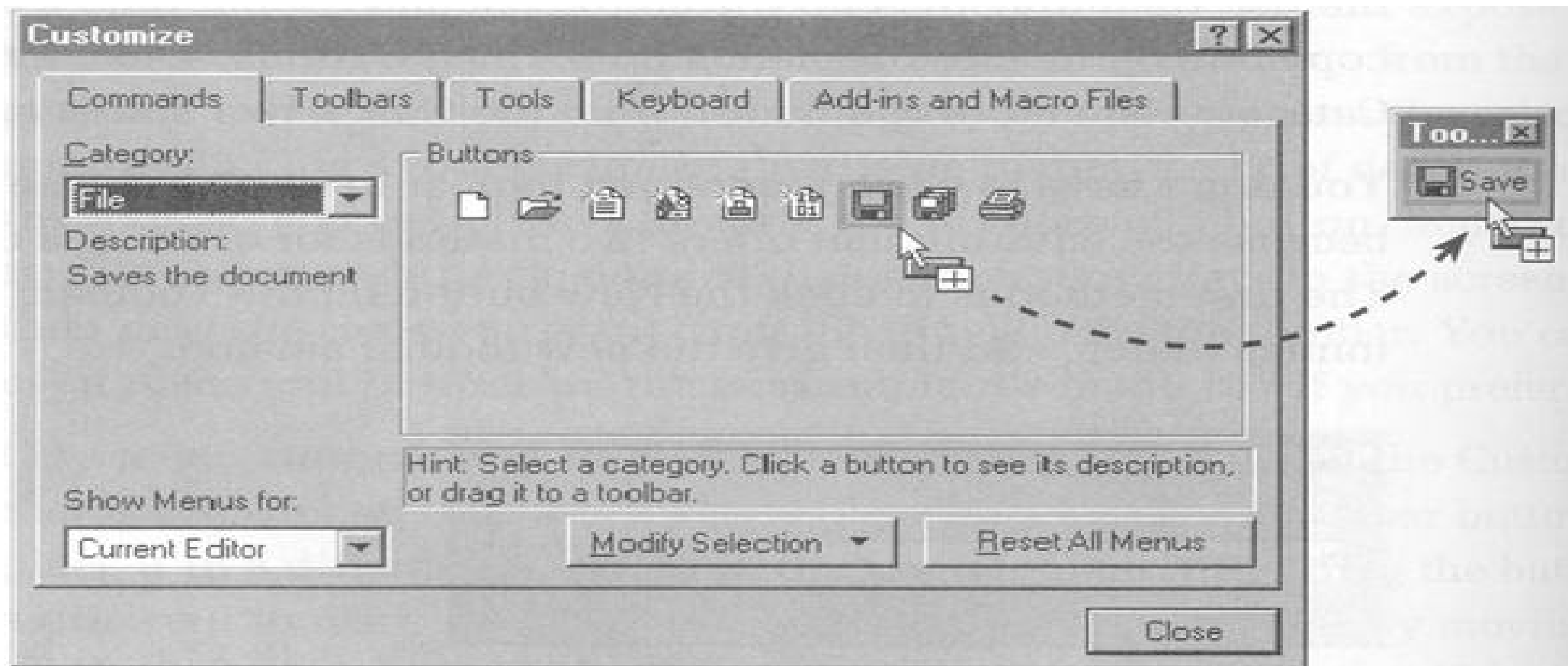


图 13-3 创建一个新的自定义工具栏

可以轻而易举地从任何工具栏中删除按钮，不管这些按钮是自定义的，还是预定义的。当屏幕上显示 Customize 对话框时，我们可以从工具栏中拖出一个按钮，并在屏上的空白处释放它。同时，也可为工具栏和任何按钮标题更名。给按钮标题更名时，先在它所在的工具栏中右击这个按钮，并从弹出菜单中选择 Button

Appearance（按钮外观）命令，然后，在 Button Appearance 对话框底部的编辑框中重新输入按钮标题，当然，也可在任何时候，通过剪贴板的剪切和粘贴操作来更换按钮图标。

给定制的工具栏更名时，有必要先浏览一下 Customize 对话框中的 Toolbars 选项卡，在这里，值得注意的是，用户只能给自定义的工具栏更名，而不能给系统预定义的 Developer Studio 工具栏更名。更名时，先从表中选出自定义的工具栏，并在 Toolbar Name（工具栏名称）框中输入新的名称，新的名称就会在工具栏的标题区域和 Customize 对话框的 Toolbar 表中显示出来，这样，我们就能以相同的方式打开和关闭其他工具栏。当新工具栏只有几个按钮时，应尽量名称简短，否则，标题栏中的空间会不够用。图 13-4 显示了在第 3 章中创建从 Toolbar1 到 Word Case 工具栏的更名过程。

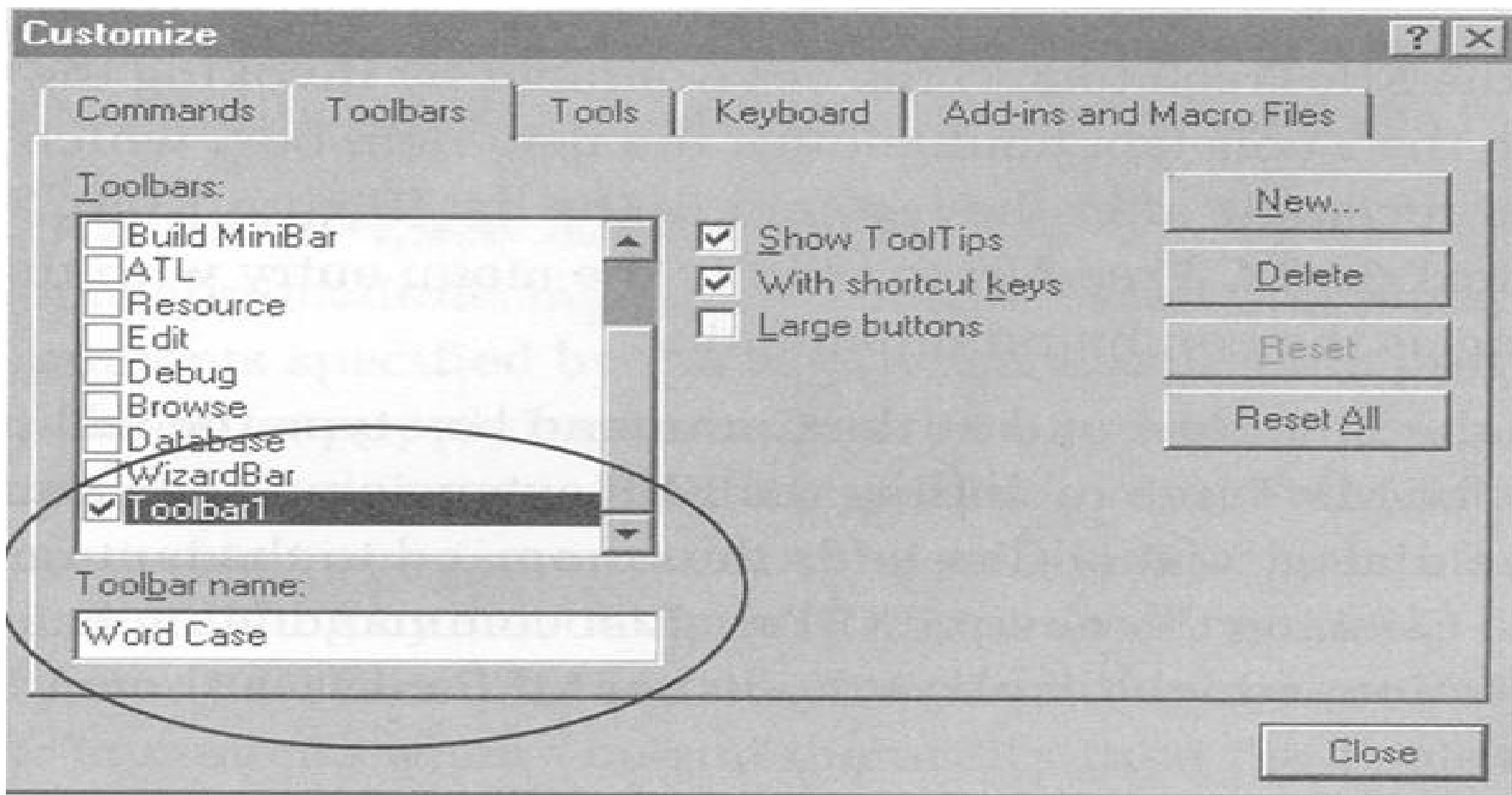


图 13-4 在 Customize (自定义) 对话框中重新命名一个工具栏

在表中选择一个定制工具栏，并启用 Delete 按钮，使它具有永久删除定制工具栏的功能。

添加命令到 Tools（工具）菜单

本书第 3 章中的最后一节，讲述了在 Developer Studio 环境下，添加命令到 Tools 菜单中，以便启动第三方文本编辑器。并讲述了不仅可为文本编辑器，也可为任何程序增加命令。本节更详细地论述了增加命令的过程，以及集成应用程序到环境中的可用选项。

首先，让我们在没有提示或专门处理工具的情况下，回顾一下给 Tools 菜单增加简单的实用程序的过程。MfcTrees3 应用程序列出了 MFC 类的层次，它是第 5 章中开发的 MfcTree2 对话框的更完整的版本。与先前的版本相比，MfcTree3 的源代码并无多大改变，但此程序现在将类名作为字符串资源来保存，而不是在对话框的 OnInitDialog 函数中，将它们硬编码到 CTreeCtrl::InsertItem 的长长的调用列表中。在配套光盘的 Chapter.13\MfcTree3 文件夹中，可以找到所有源文件，在下面的演示中，只需要将 CD 盘上的 MfcTree3.exe 程序文件复制到磁盘上的任何位置。

通过调用 Customize 对话框，可以添加 MfcTrees 工具到环境中。在 Tools 选项卡中，双击新项所在的框，它在列表最后的工具之后，显示为一个点划的矩形（如

图 13-5 所示)。输入菜单项文本 &MFC Tree List, 用字母 “ M ” 作为命令助记符, 来指定菜单项。

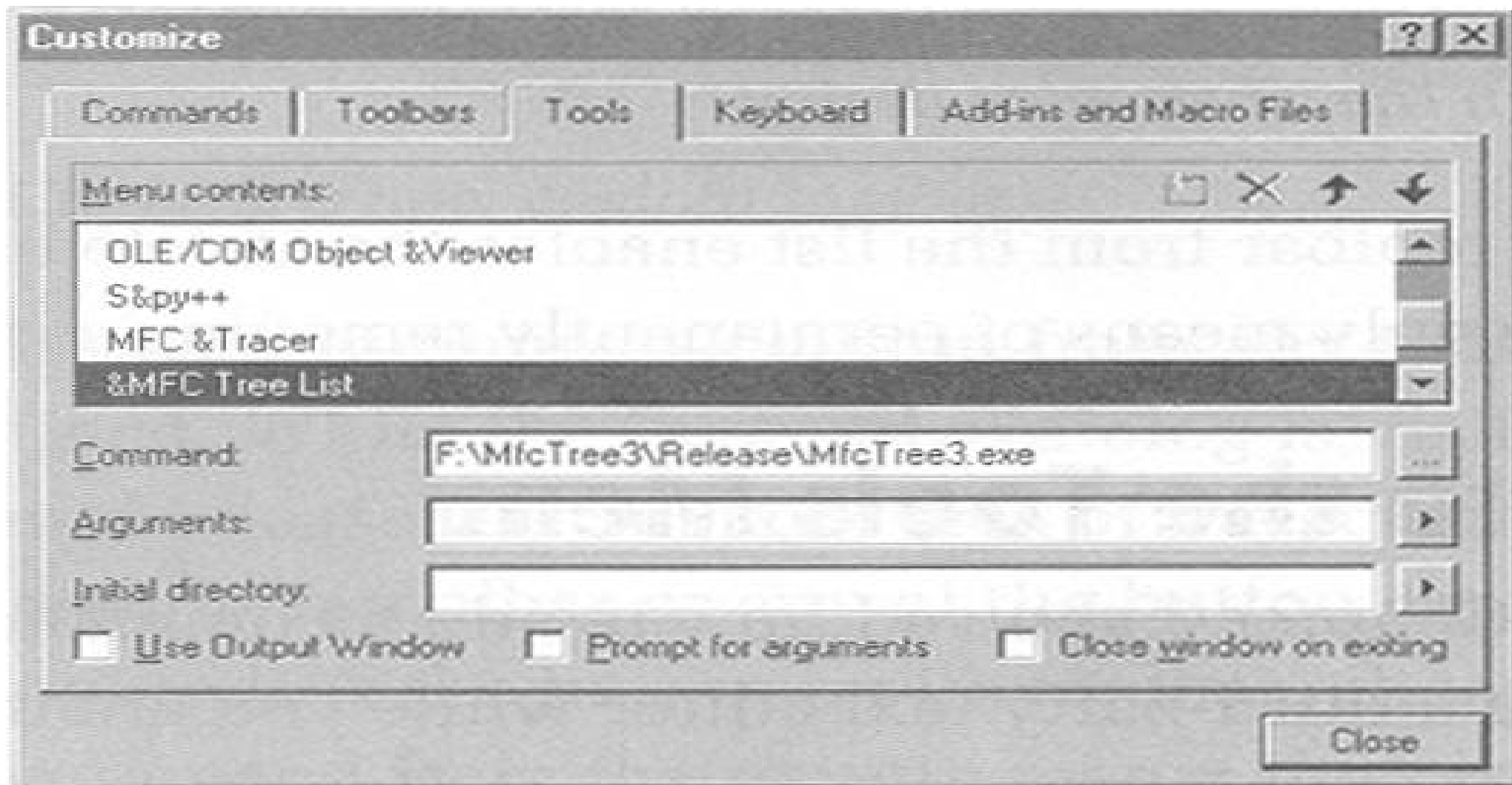


图 13-5 添加新的 MFC Tree List 命令到 Tools（工具）菜单

按下 Enter 键，在 Command 框中，为 MfcTree3 输入完整的路径和文件名，包括 EXE 扩展名。在关闭 Customize 对话框时，Visual C++ 将此命令添加到 Tools 菜单的底部。单击新的 MFC Tree List 命令便启动 MfcTree3 应用程序，它将显示 MFC 的类列表，如图 13-6 所示。

MfcTree3 可以很容易地添加到 Tools 菜单中，因为此程序不需要任何命令行参数，而且在一个单独的窗口即可显示其输出。下面将介绍的其他实用程序就不这么简单了。

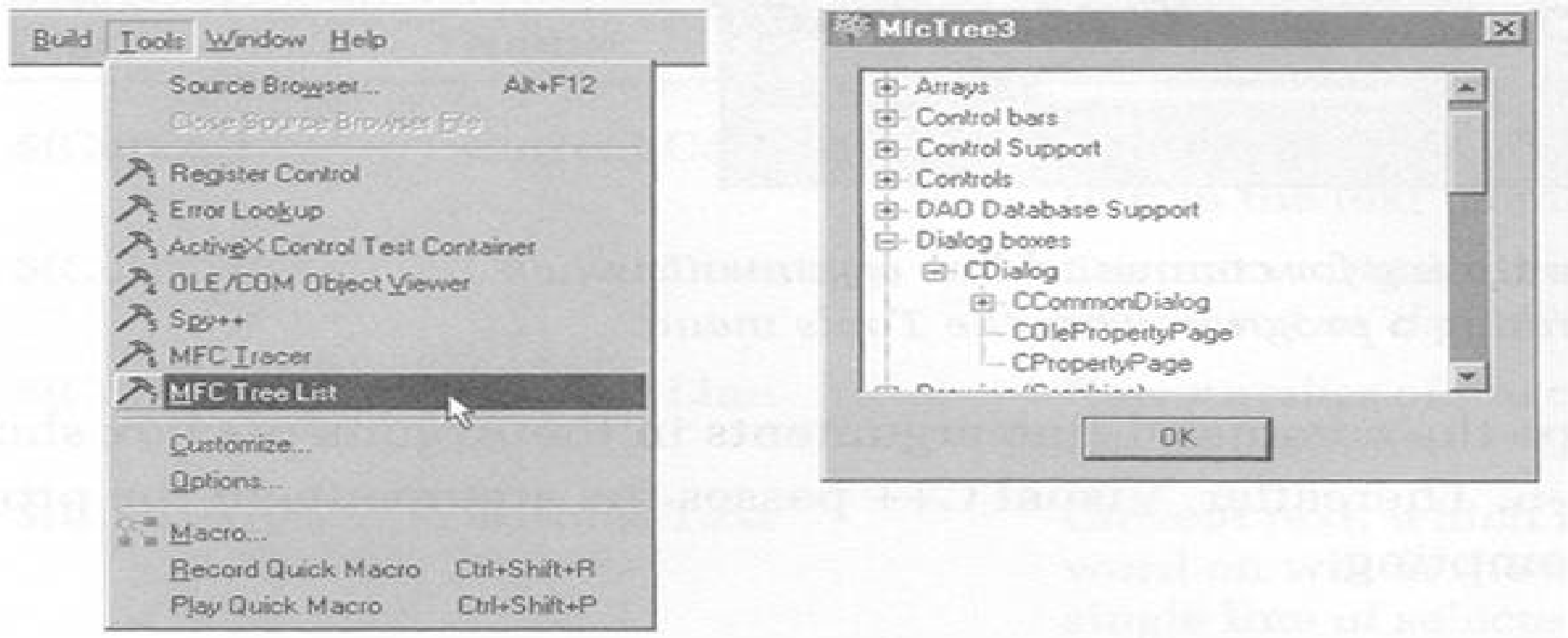


图 13-6 Tools (工具) 菜单和 MFC Tree3 应用程序的新的 MFC Tree List 命令

命令行参数

某些应用程序，尤其是基于控制台的程序，在程序运行时，需要使用用户指定的

命令栏参数。对从 Tools 菜单上启动使用命令行参数的程序，有两种方法。第一种是给 Visual C++ 配置每次运行程序时查询命令行参数的功能。可选用 Customize 对话框中的 Prompt For Arguments (提示参数) 复选框来设置这种配置，如图 13-5 所示。具有这种配置后，在每次从 Tools 菜单运行此实用程序时，此复选框会使 Visual C++ 提示参数，然后，通过命令行将参数传给程序。

我们看一下范例：假定用户想浏览一下 Prompt For Arguments 复选框打开时，添加 Windows Notepad 实用程序到 Tools 菜单中，将出现的情况。在 Tools 菜单中启动 Notepad，首先，显示一个提示，你可以在其中输入文件名，如图 13-7 所示。单击 OK 按钮，Notepad 启动，并自动装入指定的文件。

如果用户想要在每次启动应用程序时，都能接收到同一命令行参数，使用第二种方法来提供参数更为简捷。清除 Prompt For Arguments 复选框，在图 13-5 所示的 Arguments 框中输入命令行参数。此后，Visual C++ 会将参数传递给程序，而且不再提示。

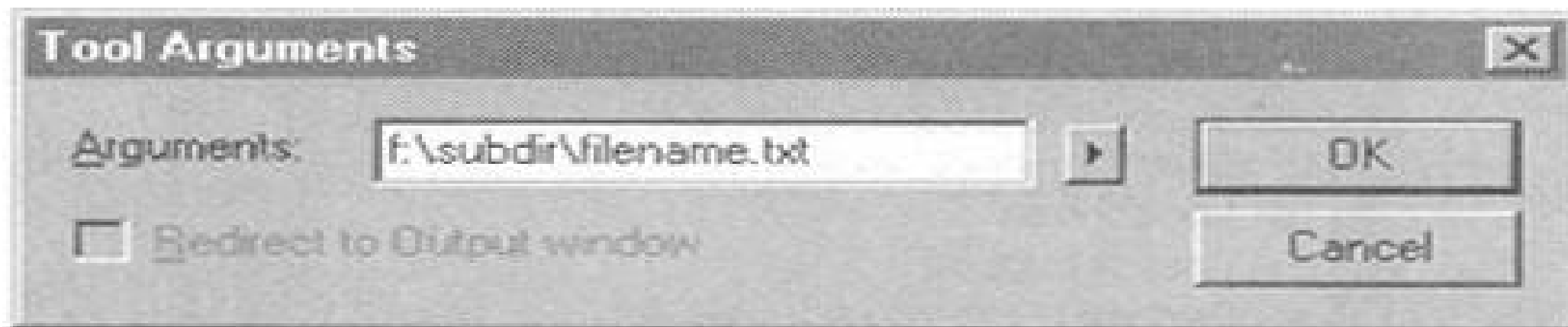


图 13-7 当从工具菜单运行程序时的命令行变量提示

参数宏

Visual C++具有参数宏这一较显著的特征，这样，给 Tools 程序传递命令更为容易一些。如表 13-3 所示。每一条宏命令都扩展成一个字符串，用来描述当前项目或文件的特点。

在说到参数宏命令的灵活性时，我们来看一个简单的例子，假定用户想用 Notepad 工具，使当前运行的文本编辑器中的文档处于常开状态，使用宏命令 `$(FilePath)` 要容易得多，且不用每次都提示文件名，即可达到上述目的。同时，这个宏命令也能扩展，对当前具有输入焦点文档使用完全的路径和文件名。如若没有文档焦点，宏将产生一个空字符串。要利用 Notepad 工具来使用宏命令，可删除 Customize

Arguments 对话框中的 Prompt For Arguments 复选框，并键入 \$(FilePath)，如下所示：

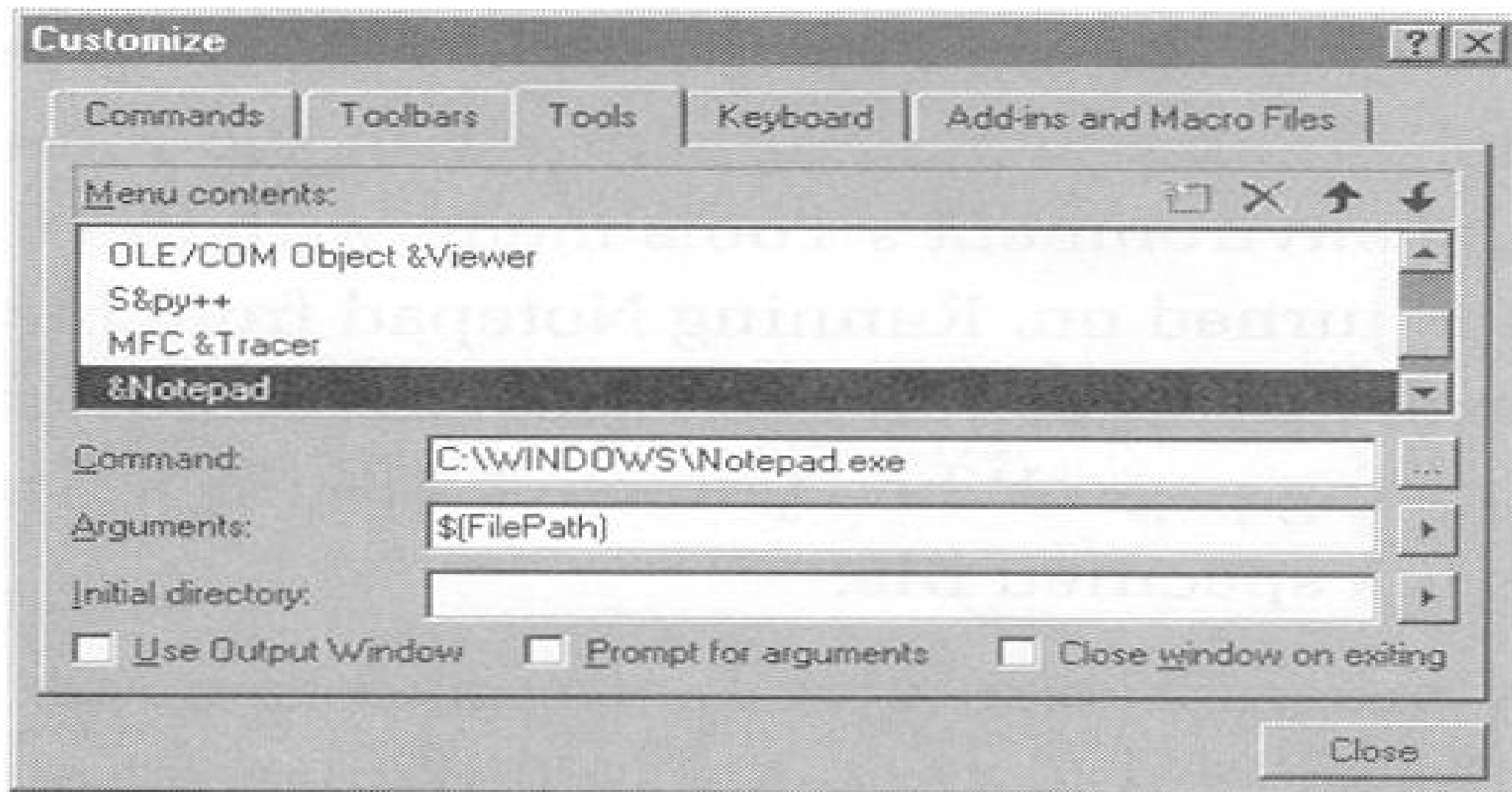


表 13-3 适用于 Tools 选项卡的参数宏

宏	名称	说明
\$(CurCol)	当前列	在文本窗口里，插入符位置的当前列数
\$(CurDir)	当前目录	当前工作目录，表示为 d:path\
\$(CurLine)	当前行	在文本窗口里，插入符位置的当前行数
\$(CurText)	当前文本	当前文本，插入符所停位置的词，或者所选文本的一行
\$(FileDir)	文件目录	活动窗口里的源文件目录，表示为 d:path\
\$(FileExt)	文件扩展名	活动窗口里的源文件扩展名
\$(FileName)	文件名	活动窗口里源文件的文件名
\$(FilePath)	文件路径	活动窗口里源文件的完整路径和文件名，表示为： d:path\filename

续表

\$(TargetArgs)	目标参数	传递给项目应用程序的命令行参数
\$(TargetDir)	目标目录	Debug 或 Release 子目录里包含的可执行项目的路径，表示为 d:\path\
	目标扩展名	可执行项目的文件名范围，例如 EXE 或 DLL
\$(TargetExt)	目标名	可执行项目的文件名（通常是项目名）
\$(TargetName)	目标路径	可执行项目的文件名的完整路径和文件名，表示为 d:\path\filename
\$(TargetPath)		含有项目文件的目录，表示为 d:\path\项目
	工作空间目录	项目名
\$(WkspDir)	工作空间名	
\$(WkspName)		

在文本编译器中编辑文档期间，启动此工具时，Notepad 会自动打开相同的文档。

表 13-3 中提供了 Tools 选项卡中可用的 15 种不同的参数宏命令。

在 Visual C++ 查询命令行参数时，可使用参数宏命令来完成，如图 13-7 所示。而且，在 Arguments（参数）和 Initial Directory（初始目录）文本框中，这种宏命令更为有用。我们在使用这些宏命令时，不必费力去记住它们，需要时只要单击一下文本框旁边的箭头按钮，就会显示一个菜单，其中是宏命令名称的完整列表。单击表中的任一宏命令，它就会出现在旁边的文本框中。宏名不区分大小写，例如，\$(FileDir)与\$(filedir)是相同的，宏所产生的路径字符串，诸如\$(FileDir)与\$(TangetDir)，以反斜线结束。

例子：Struct实用程序工具

参数宏命令能为 Developer Studio 环境与实用程序更紧密地集成到一起提供一种方式。通过自动为当前项目或文档提供合适的命令行命令，利用参数宏，可以为 Visual C++ 专门设计的 Tools 菜单创建附加的实用程序。

这里有一个例子，说明了工具程序是怎样响应在文本编辑器中的当前插入符位置，甚至显示 Developer Studio 环境内部的输出。Struct 实用程序描述了接收命令行参数的过程，而这些参数包含来自当前文本文档中的词。如果参数容纳此程序小数据库中表示的 Win32 API 结构的一个名称，Struct 就会在 Developer Studios 的 Output 窗口中显示结构的声明。\$(CurText)宏产生的命令行参数可以在文档中去选择，如果没有选择文本，该词就会在插入符位置。如果 Struct 不能识别所提

供的结构名称，屏幕上将会出现未知结构名的消息。

图 13-8 显示了在 Customize 对话框中怎样创建 Struct 程序，并使它成为 Tools 菜单上的一个工具的过程。Struct 可以访问到 Output 窗口，原因是 Use Output Window(使用输出窗口)复选框处于打开状态。这个复选框只对基于控制台的程序启用。设置此复选框，能使 Develop Studio 截取所有实用程序的标准输出，并在 Output 窗口中单独的选项卡里显示。通过选项卡中显示的工具名称，我们可以确定输出来源。

Struct 程序相当简单，仅能识别几个 Win32 API 结构，例如 RECT 和 POINT。Struct 是以 C 语言编写的，只不过是一个程序外壳，但是，它也能很容易地扩充，以包括附加的结构和其他信息，如函数原型以及消息参数。尽管具有某些局限性，这个程序还是能清楚地演示如何将这些工具和 Visual C++ 集成。安装和检测 Struct 实用程序时，可采取以下步骤：

1. 如果从配套 CD 盘上复制项目时，没有运行 Setup 程序，我们可复制 Struct.exe 程序文件复制到硬盘的适当位置。此程序装在 CD 的 Code\Chapter.13\Struct\Release 子文件夹中。
2. 在 Customize 对话框的 Tools 选项卡中，给 Tools 菜单，为 Struct 添加命令，如图 13-8 所示。在 Command 框中，输入 Struct.exe 的路径和文件名，包括 EXE 扩展名，在硬盘上指定文件夹到复制此程序的位置。在 Arguments 框中输入 \$(CurText)，或单击旁边的箭头按钮，并从表中

选择 Current Text (当前文本)。注意，在此过程中，必须打开对话框左下角的 Use Output Window 复选框。

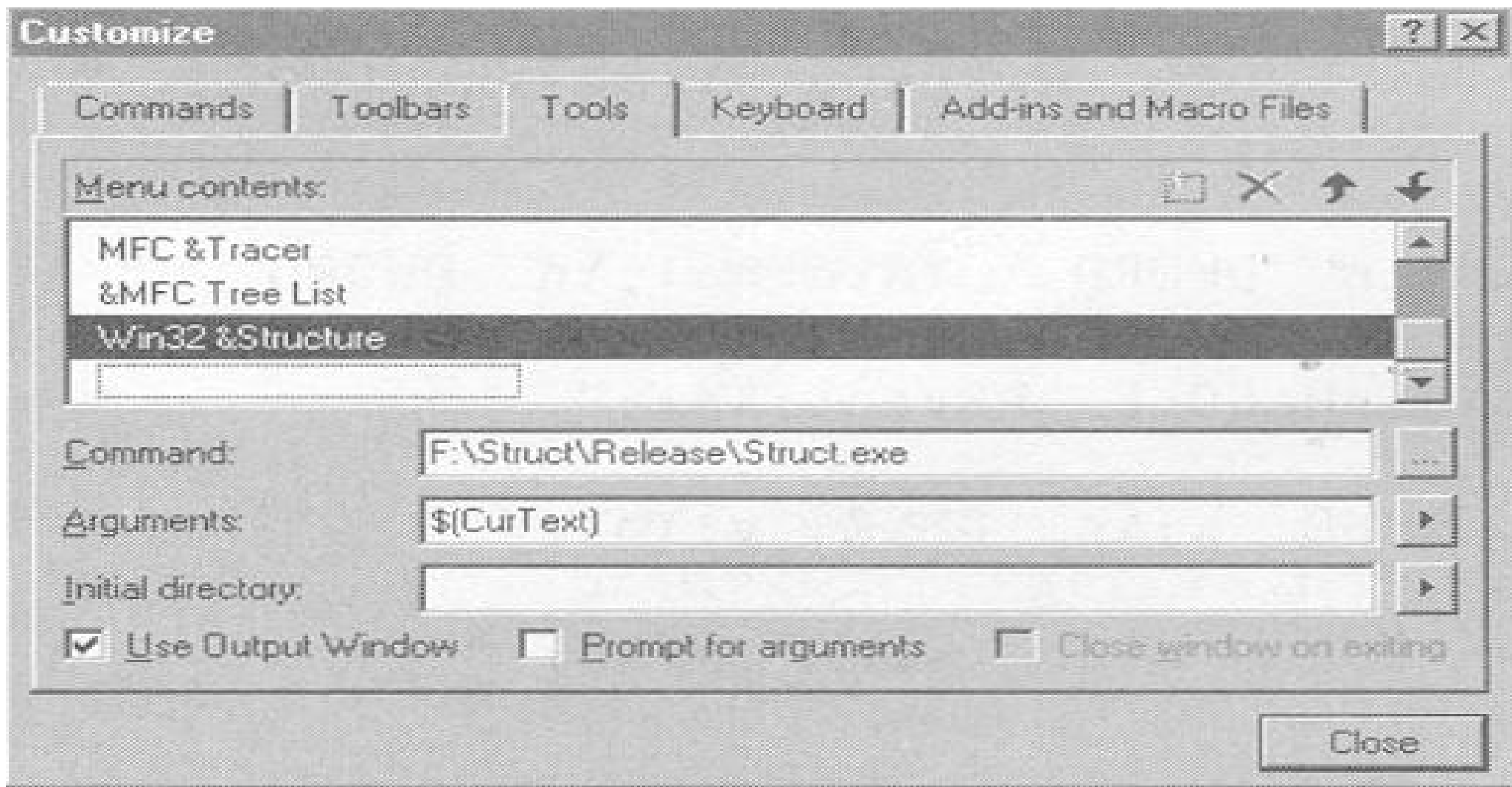


图 13-8 为 Struct 实用程序设置一个菜单命令

3.在文本编辑器中打开 Struet.c 源文件（列表 13-1），或用 File 菜单上的

New 命令，来创建一个新文本文档。在新建文档中输入一行，其中包含 Struct 所能识别的 API 结构名称，如下：

```
RECT RECTL POINT#POINTL#SIIE#POINIS#FILE#TIME#SYSTE#MTIME
```

4. 在文档中，将插入符放在函数名中的任一位置，在 Tools 菜单上单击 Win32 Structure 命令，图 13-9 显示了 Output 窗口中 Win32 Struct 选项卡的实用程序消息。

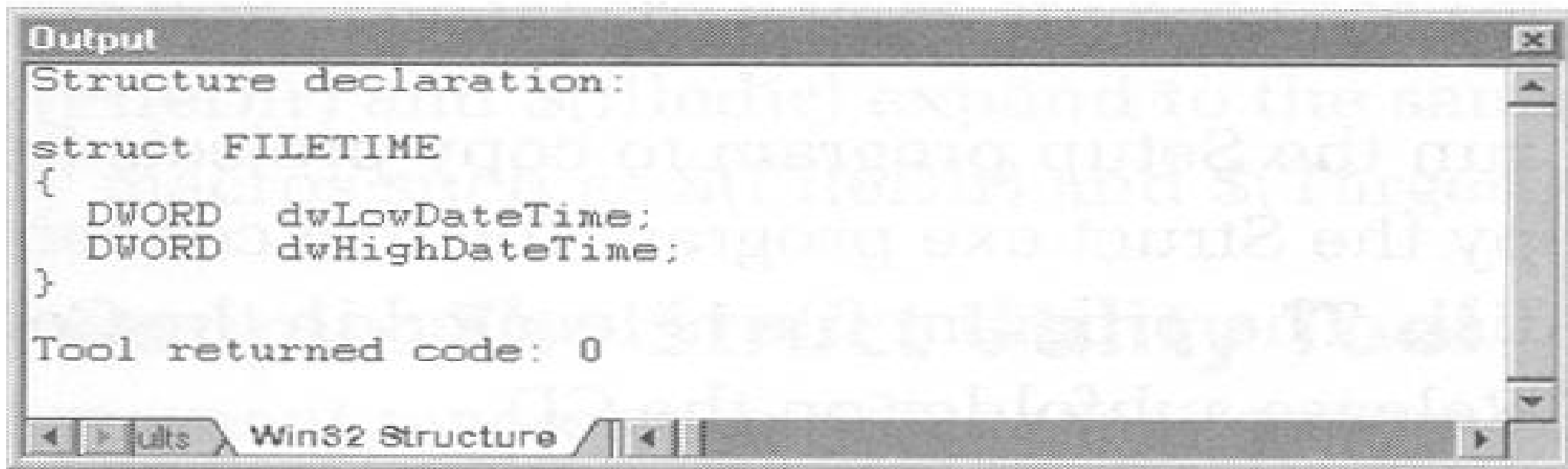


图 13-9 在 Output（输出）窗口里，Struct 实用程序的输出出现在自己的选项卡中

列表 13-1 Struct.c 源文件

```
// Struct.c           Displays Win32 structure declarations in Output window

//                   Copyright (c) 1998, Beck Zaratian


#include <stdio.h>

#include <string.h>


char *pszStruct[] =

{

    "FILETIME\n{\n    DWORD    dwLowDateTime;\n    DWORD    " \

    "dwHighDateTime;\n}",
```



```
"RECT\n{\n    LONG    left;\n    LONG    top;\n    LONG    right;\n\n    LONG    bottom;\n};",
```

```
"RECTL\n{\n    LONG    left;\n    LONG    top;\n    LONG    right;\n\n    LONG    bottom;\n};",
```

```
"SECURITY_ATTRIBUTES\n{\n    DWORD    nLength;\n    LPVOID    "\n\n    lpSecurityDescriptor;\n    BOOL    bInheritHandle;\n};",
```

```
"SIZE\n{\n    LONG    cx;\n    LONG    cy;\n};",
```

```
    "SYSTEMTIME\\n{\\n    WORD    wYear;\\n    WORD    wMonth;\\n    WORD    " \\n\n    "wDayOfWeek;\\n    WORD    wDay;\\n    WORD    wHour;\\n    WORD    " \\n\n    "wMinute;\\n    WORD    wSecond;\\n    WORD    wMilliseconds;\\n};"\n};
```

```
main( int argc, char *argv[], char *envp[] )\n{\n\n    char    *sz;\n\n    size_t    iLen;\n\n    int        i, iCount = sizeof(pszStruct)/sizeof(char*);\n\n\n    if (argc > 1)
```

```
{  
  
    for (i=0; i < iCount; i++)  
  
    {  
  
        // Determine length of structure name  
  
        sz      = pszStruct[i];  
  
        iLen = (size_t) (strchr( sz, '\n' ) - sz);  
  
  
        // If structure is in database, display declaration  
  
        if (strlen( argv[1] ) == iLen    &&  
            !strncmp( sz, argv[1], iLen ))  
  
        {  
  
            printf( "Structure declaration:\n\n" );
```

```
        printf( "struct %s\n\n", pszStruct[i] );

        break;

    }

}

if (i == iCount)

    printf( "Structure not recognized\n\n" );

}

return 0;

}
```

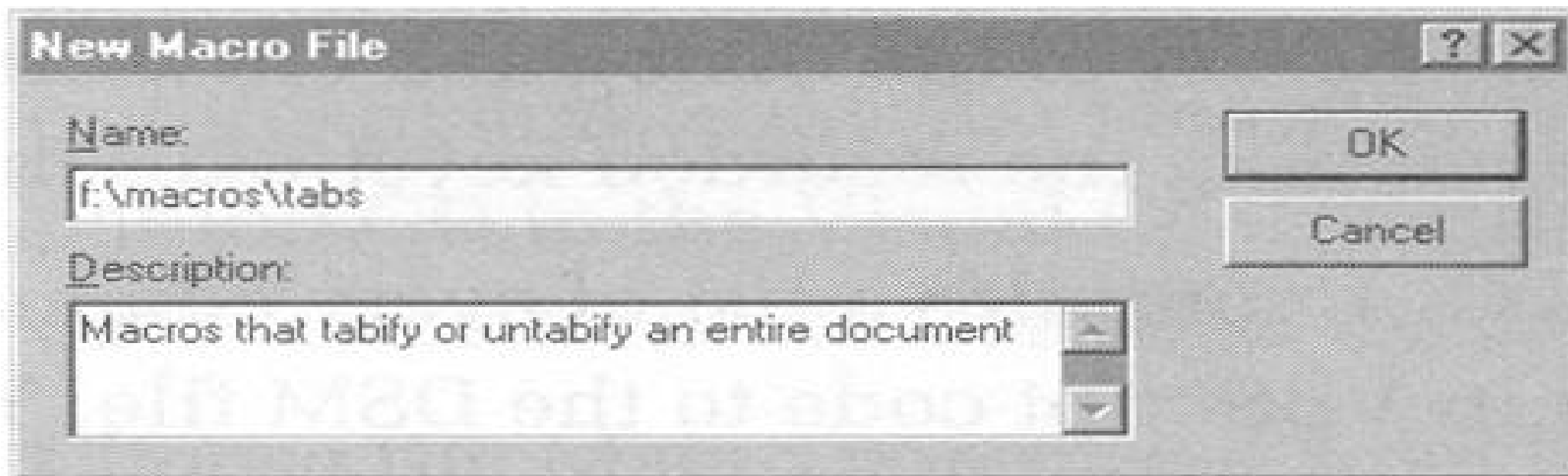
宏

Visual C++中有一种优秀的宏语言，其格式为 Visual Basic Scripting Edition，即著名的 VBScript。通过在文件中存储宏脚本时，这个环境能使不断收集与其他文件共享的宏命令。通过记录任务序列，或者是书写程序脚本，用户可创建一个宏。在这部分里，我们将看到，VBScript 提供了一个函数库，它可以运行宏来查询用户的输入，显示消息框，进行算术运算，控制字符串，并执行一些超出用户接口程序范围的其他任务。

VBScript 是 Microsoft Visual Basic for Application (VBA) 的子集，VBA 是 Microsoft Access 这样程序的编程语言。VBScript 设计成一种用 HTML 书写的 Web 文档的脚本编程语言，它为 HTML 页面提供了嵌入 ActiveX 控件和其他对象的方式。但是，它还可以作为常规的脚本编程语言，能解释针对应用程序的命令表，并通过 Automation 来执行这些命令。换句话说，VBScript 可以作为宏语言。在这种环境中，Visual C++ 能使用 VBScript。

一个宏意味着将一类命令封装为一个命令。用 VBScript 写的宏实际上是一个简单程序，在这种程序中，宏脚本作为源代码。执行一个宏，就等于执行了脚本中包含的所有命令。在第 3 章中，我们讲了怎样通过记录敲键和鼠标单击，来创建 VBScript 宏的过程。记录后，便为用户提供一个宏，它可以回放一系列记录的命令。一旦记录了一个宏，就需要手工使用这些命令。此后，无论在什么时候运行录制的宏，Developer Studio 都会自动重复相同的步骤。

第 3 章中的解除制表符转换宏是使用 Record Quick Macro（快速录制宏）命令创建的，这一方法尽管很方便，但也有其不足之处，即 Record Quick Macro 命令仅能创建一个临时宏，当下次使这个命令和记录另一个宏时，前一个宏将永远消失。相反，Tools 菜单中的 Macro 命令能记录永久保存的宏，并用一个单独的文件来保存几个相关的宏。为了说明这点，这里说明了怎样创建这个宏的永久性版本，并用相应的制表符转换宏来增强它。这个宏扩展了 TabifySelection 和 UntabifySelection 命令，来对整个文档加制表符和取消制表符，而不仅仅选择的文本。第一步是创建一个宏文件，接着为这些新宏增加脚本。在文件编辑器中打开一个文档，从 Tools 菜单中选取 Macro 命令，来打开 Macro 对话框，单击 Options 按钮，接着单击 New File（新文件）按钮，输入一个文件名和这个宏文件的说明，如下：



单击 OK 按钮，并在第一个宏的名下键入 UntabifyAll，单击 Record 按钮，在 Add Macro（添加宏）对话框中键入可选的说明文字，例如，Expands tabs into spaces。

单击 OK 按钮，主 Macro 对话框关闭，并返回到文本编辑器的文档中。现在，Record 工具栏显示在屏上，鼠标光标中包含着盒式磁带的图像，这就暗示着，现在，Visual C++正在记录着每个敲键和鼠标单击。同第 3 章所述的一样，我们可以通过同样的三步来创建宏；

- 1 在 Edit 菜单上，单击 Select All（全选）命令来选整个文档。

- 2 从 Edit 菜单中选 Advanced，并单击 Untabify Selection 命令。

3 按 Ctrl+Home 将插入符移到文档顶端。

在 Record 工具栏中单击 Stop Recording 按钮来结束录制，在此处，文本编辑器自动打开新的 VBScript 宏文件，该文件名为 Tabs.dsm，此时可以编辑宏（该文件的扩展名代表 Developer Studio 宏）。在 Window 菜单上单击文件名，就会返回到原来的文本文档中。

当再次显示 Macro 对话框时，Macro File（宏文件）组合框中就会出现 Tabs 文件名。按上文所述的相同步骤，可增加一个新的转换制表符宏，只是这次不单击 Options 和 New File 按钮，因为我们只想增加到宏文件，而不是去创建一个新文件。仅需键入 TabifyAll 来给第二个宏命名，单击 Record 按钮，输入新宏的说明，例如 Converts spaces into tabs。接着按以上步骤，创建第二个宏，这次，我们选用 Tabify Selection（将选择内容进行制表符转换）命令。此 Tabs 宏文件现在包含着两个宏，即在 Macro 对话框中列出的 UntabifyAll 和 TabifyAll。运行宏时，从对话框的列表中选出它，并单击 Run 按钮，其效果与人工重新键入录制的敲键一样。

在录制过程中，Visual C++ 将所执行的每个命令列表进行编译，并把它们用 VBScript 代码写到 DSM 文件中。在录制时，可以在任何编辑器中工作，甚至可以在编辑器之间来回转换。单击 Record 工具栏中的第二个按钮暂停录制，可以执行不包括在最终宏中的操作。再单击相同的按钮，则恢复录制。一般来说，在运行宏前，首先，应把插入符移到要回放所录制敲键的文档位置。这一步对

TabifyAll 和 UntabifyAll 宏来说，并不需要。

大多数的宏都是通过这种方法来完成录制一系列命令的任务，但是，可能出现这种状况：需要一个宏完成的任务却不能被录制。要创建这样一个宏，必须写一个脚本，并将它作为 DSM 文件存盘，下节将介绍如何做这一工作。

例子：用于栏目搜索和替换的宏

VBScript 包括一个编译器和一个运行库。VBScript 编译器解释宏中列出的命令，库提供了宏可以调用的函数。附录 C 中包含了关于 VBScript 的简短教程，它描述了语言元素和 VBScript 库函数。VBScript 十分简单，可以很快掌握，好在它的许多编程特点和 C 语言十分相似。

使用一个简单例子即可说明一些 VBScript 宏的特征。在第 3 章中讲述过，虽然在一个文档中可以选择一块文本，但文本编辑器对所选栏没有提供搜索和替换操作的方法。如果给文本栏作上标记，并从 Edit（编辑）菜单中选择 Replace（替换），Replace 对话框就会使 Selection 单选钮失效。对编辑器来说，在标记栏中替换文本，是非常有用的特征。用一个宏，我们能编程 Developer Studio 文本编辑器来做到这一点。

通过尽可能多地进行录制来创建宏，然后，使用文本编辑器，来给宏脚本文件增加一些不能录制的命令，这是十分方便的。在本例中，我们不录制任何东西，原

因是 Replace.dsm 宏文件已经从头开始编码了。要检测一下这个宏，将 CD 盘上的 Chapter.13 文件夹中的 Replace.dsm 复制到硬盘的 Visual C++ Macro 文件夹中。该文件夹的默认路径是 Common\MsDev98\Macros。

从 Tools 菜单中选 Macro 命令，单击 Options 按钮，然后单击 Loaded Files（加载的文件）按钮。这就会显示我们熟悉的 Customize 对话框，其中，新的 Replace.dsm 文件显示在 Macros 文件夹中的宏文件表中，打开表中的 Replace 复选框，关闭对话框，并再一次调用 Macro 命令。这个 Replace 宏文件马上显示在 Macro 对话框的下拉列表中，如图 13-10 所示，选择这个文件，在表中增加单独的 ColumnarReplace 宏。

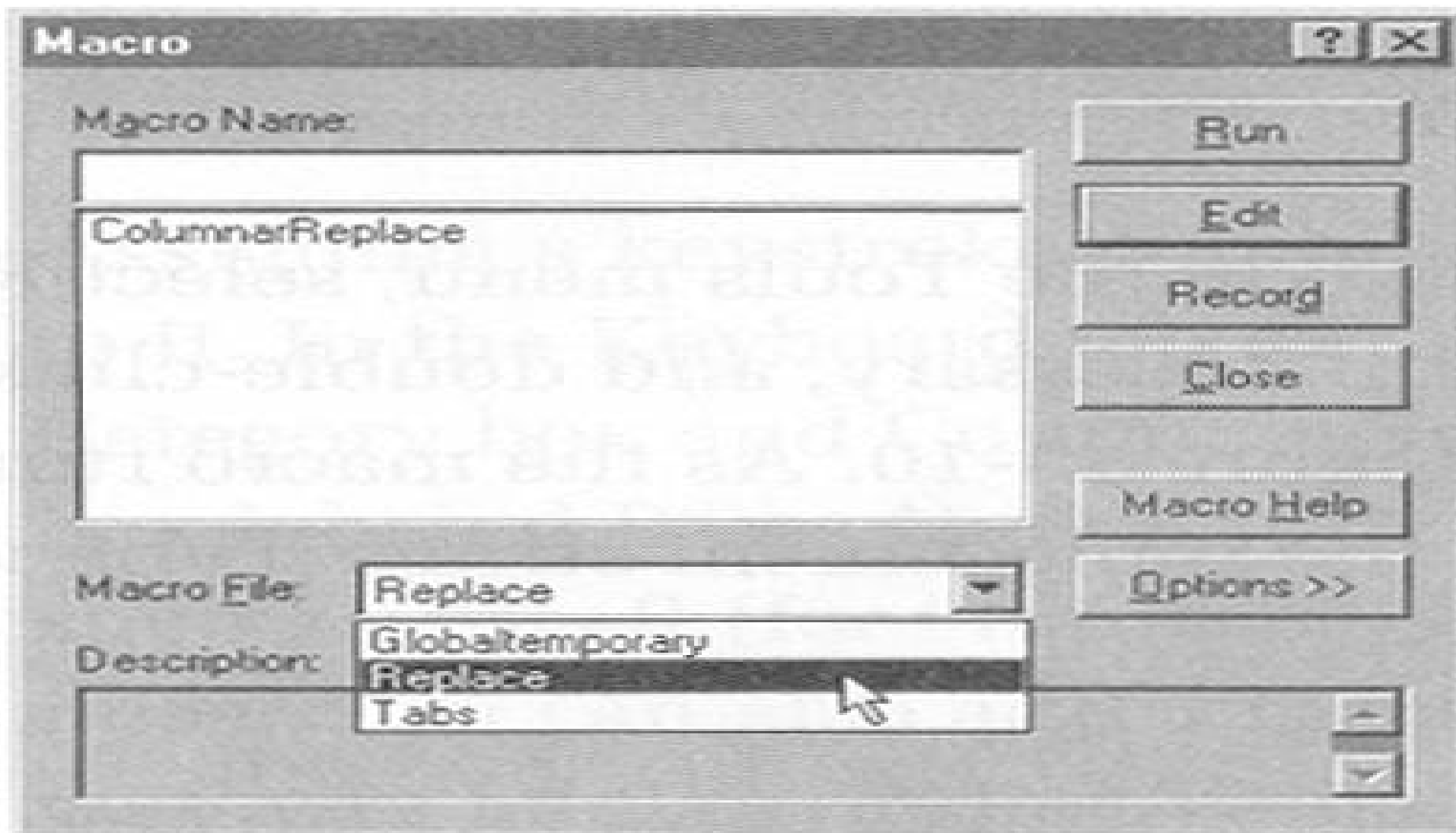
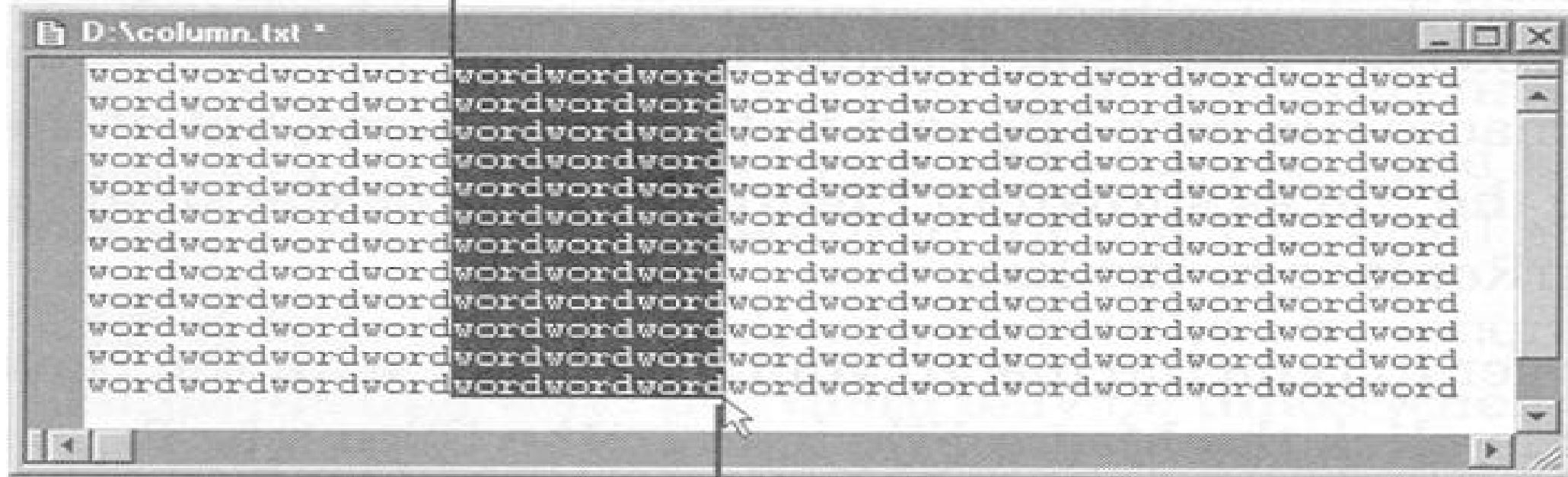


图 13-10 调用 ColumnarReplace 宏

如果想亲自键入这个宏，用 New 命令，并在 Files 选项卡中双击 Macro File（宏文件）图标。键入脚本，如列表 13-2 所示，接着，把该文件以 Replace.dsm 为名存在 Common\MsDev98\Macros 文件夹中。Developer Studio 仍不能识别这个新的宏文件，因此，必须按前一段所说的那样，打开它在 Customize 对话框中的复选框。

配套光盘的 Chapter.13 文件夹包括了名为 Column.txt 的文本文件，为演示 ColumnarReplace 宏提供了简单的测试基础（你可以使用所要的任何文本文档来试验）。为了改变文本中的一栏，拖动鼠标时按着 ALT 键，先选择一块文本，如图 13-11 所示。通常，可以从右下角向左上角拖动鼠标来标记一块，但是，ColumnarReplace 宏却假定鼠标光标从左上方向右下方拖动。

With the Alt key pressed, drag the mouse from here ...

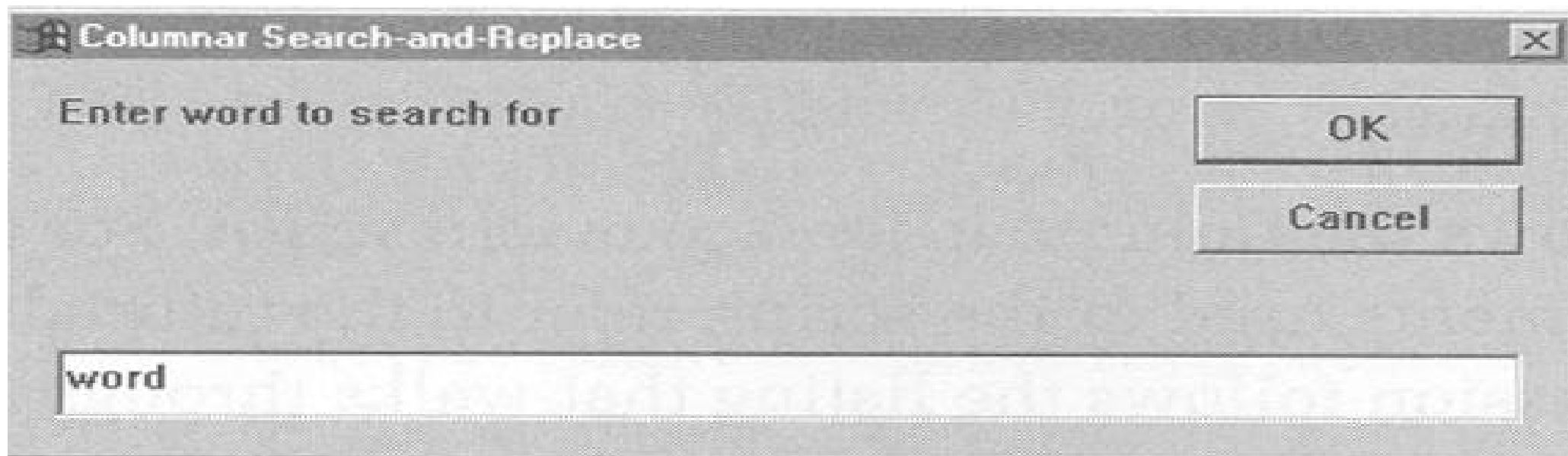


... and release here.

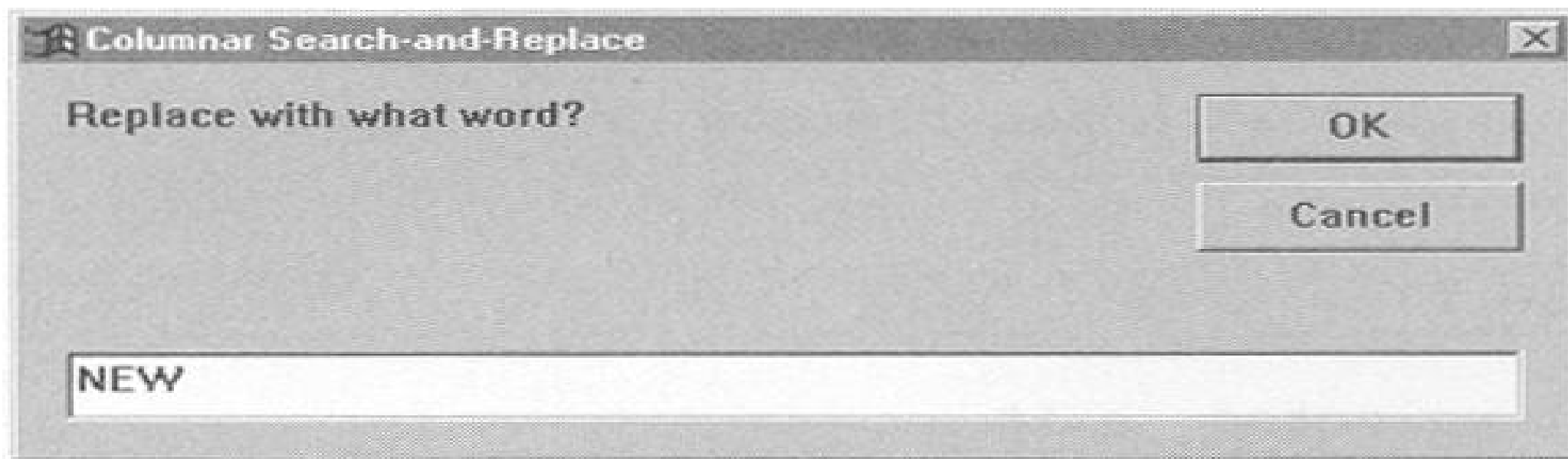
图 13-11 标记文本块

接下来，单击 Tools 菜单中的 Macro 命令，如果有必要的话，从 Macro File 下拉列表中选择 Replace，并双击框中的 ColumnarReplace，如图 13-10 所示。当这个

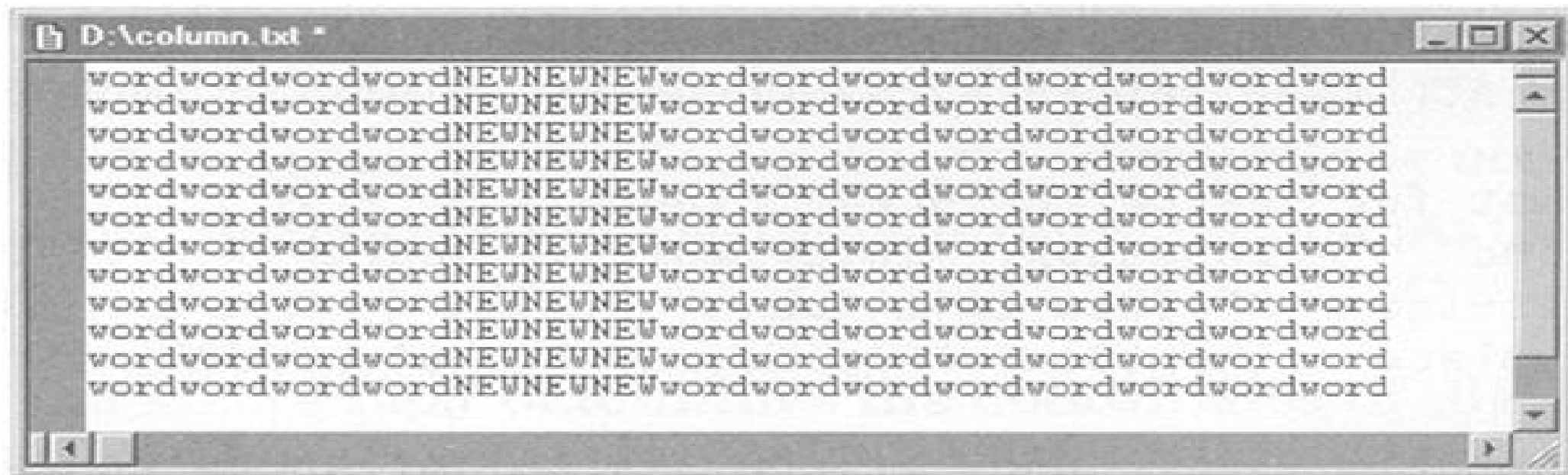
宏运行时，它查询搜索的字符串和替换的字符串。并在第一次查询时键入 Word 一词：



在第二个询问下键入替换文本，如 New：



在第二个询问下单击 Enter 键或者单击 OK，在选择块中，宏就会代替所有的“word”，其余的“word”词不变：



把一个敲键组合指定给宏,以便可以快速访问它。在 Customize 对话框的 Keyboard 选项卡中,从 Category 框选 Macros,并从 Commands 表中选 ColumnarReplace. 单击标有 Press New Shortcut Key (按新的快捷键)的框,并键入组合键,如 Alt+R (用于替换),能更好地提醒你宏的用途。你可以给工具栏或菜单增加宏命令,这与增加 Visual C++ 其他的命令一样。在 Customize 对话框的 Command 选项卡中,从 Category 列表中选择 Macros,然后,将宏名称从显示的列表中拖到工具栏或菜单上,如本章前部分所述。

列表 13-2 列出了 ColumnarReplace 宏脚本。对于脚本中使用的 VBScript 元素的解释,可参考附录 C 中的教程。此列表后面的简短讨论,概括了重要的代码段,

并描述了宏是如何工作的。

列表 13-2 ColumnarReplace 宏脚本

Macro: Columnar search-and-replace

```
Sub ColReplace ()
```

```
    If no text selected, exit
```

```
    If ActiveDocument.Selection="Then Exit Sub"
```

```
    strTitle="Columnar Search-and-Replace"
```

```
    Get column coordinates: (x1,y1) = upper-left corner
```

```
                                (x2,y2) = lower-right corner
```

```
    y2 = ActiveDocument.Selection.CurrentLine
```

```
x2 = ActiveDocument.Selection.Current Column
```

```
x1 = Int( x2 - InStrB(ActiveDocument.Selection,vbCR)/2 + .5)
```

```
y1=ActiveDocument Selection TopLine
```

Prompt for search/replace strings

```
strFind = InputBox( Enter word to search for, strTitle)
```

```
If strFind = "Then Exit Sub"
```

```
strReplace = InputBox("Replaece with what word?, strTitle )
```

Temporarily add a copy of the search string to the end of

the line. This prevents ReplaceText from cycling to the

top of the document after the final replacement.

```
ActiveDocument.Selection.EndOfLine
```

```
ActiveDocument.Selection = ActiveDocument.Selection + strFind
```

Start from top line of selection and work down

```
Do While y1 <= y2
```

```
    ActiveDocument.Selection.GoToLine y1
```

```
    ActiveDocument.Selection.StartOfLine
```

```
    Do While ActiveDocument.Selection.CurrentColumn < x1
```

```
        ActiveDocument.Selection
```

```
        CharRight dSMove,1
```

```
    Loop
```

```
    Do While ActiveDocument.Selection.CurrentColumn < x2
```

```
ActiveDocument.Selection.CharRight dsExtend, 1
```

```
Loop
```

```
ActiveDocument.Selection.ReplaceText strFind, strReplace
```

```
y1 = y1 + 1
```

```
Loop
```

```
Remove temporary string at end of line
```

```
ActiveDocument.Selection.EndOfLine
```

```
for i= 1 To Len( strFind)
```

```
ActiveDocument.Selection.BackSpace
```

```
Next
```

End Sub

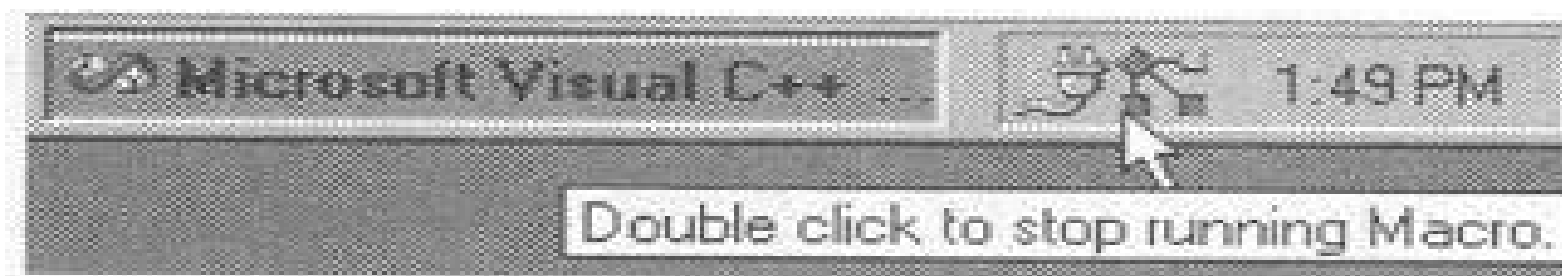
ActiveDocument.Selection 是 VBScript 属性,它包含当前文档所选的所有文本(“属性”是 Visual Basic 术语,指的是一个对象的值,用在这里与第 8 章中的意思相同。你还会记得,向上追溯,ActiveX 控件来源于 Visual Basic 自定义控件)。如果在文档中没有选择文本,则 ActiveDocument.Selection 中是空字符串。ColumnarReplace 通过检查 ActiveDocument.Selection 的内容,首先确定在文档中已选取了文本。如果这个字符串是空的,则宏终止,而不会采取任何动作。

紧接着,宏确定选择块的坐标,坐标指的是选择区的左上角和左下角的行列位置。属性 ActiveDocument.Selection.CurrentLine 和 ActiveDocument.Selection.Current Column 提供了插入符位置的行和列值,其坐标是选择块右下角的坐标 (x_2 , y_2)。这便解释了宏为什么需要你向下拖动鼠标光标,而不是向上拖动,来选择文本块。在宏开始时,插入符必须停在所选块的右下角,ActiveDocument.Selection.Topline 提供了所选块的第一行,并保存在 y_1 坐标中。对于文本块开始的栏,没有对应的属性。因此,宏依赖于这样一个事实,即 Developer Studio 插入一个返回字符,来标出选择栏中每行的结束位置,选择栏包含在 ActiveDocument.Selection 中。通过定位字符串的第一个返回字符(用 vbCR 常量表示),宏确定屏上列区域块的宽度,然后,用 x_2 减去区域块宽,算出区域块中的第一个列标 x_1 。

单独调用 Input Box 库函数,将向用户查询搜索字符串及其替换字符串。如果用

户单击 Cancel 按钮，或不指定搜索字符串，宏将停止。对于替换字符串没有类似的操作，因为宏允许用空值作为有效的替换项。如果替换字符串为空，宏只是用空来替换搜索字符串，也就是从文本中删掉字符串。在 InputBox 函数中，没有提供区分使用空替换字符串单击 OK 与单击 Cancel 之间的手段。因此，这个宏对于这两个操作的处理是一样的。

Developer Studio 提供取消运行宏的手段，当宏运行时，Developer Studio 宏图标显示在 Windows 任务栏右边的系统区中：



双击这个图标，显示结束宏的确认消息。因为在确认消息显示时，宏继续运行，所以，必须要尽快对此消息作出反应。

ColumnarReplace 中主循环的每一次重复，都能替换被选区的每一行文本。循环从区域顶端行标 ($y1$) 开始，一次下移一行，直到到达区域底部的行标 ($y2$) 处：

Do  e $y1 \leq y2$

```
ActiveDocument.Selection.ReplaceText strFind, strReplace
```

```
y1 = y1 + 1
```

```
Loop
```

第一个嵌套循环把插入符从行的始端移到区域块的第一列：

```
Find left edge of selected column
```

```
Do While ActiveDocument. Selection.CurrentColumn < x1
```

```
ActiveDocument.Selection.CharRight dsMove, 1
```

```
Loop
```

嵌套的第二个循环移动插入符，并一次通过块中的一个字符，并在运行时选择文本：

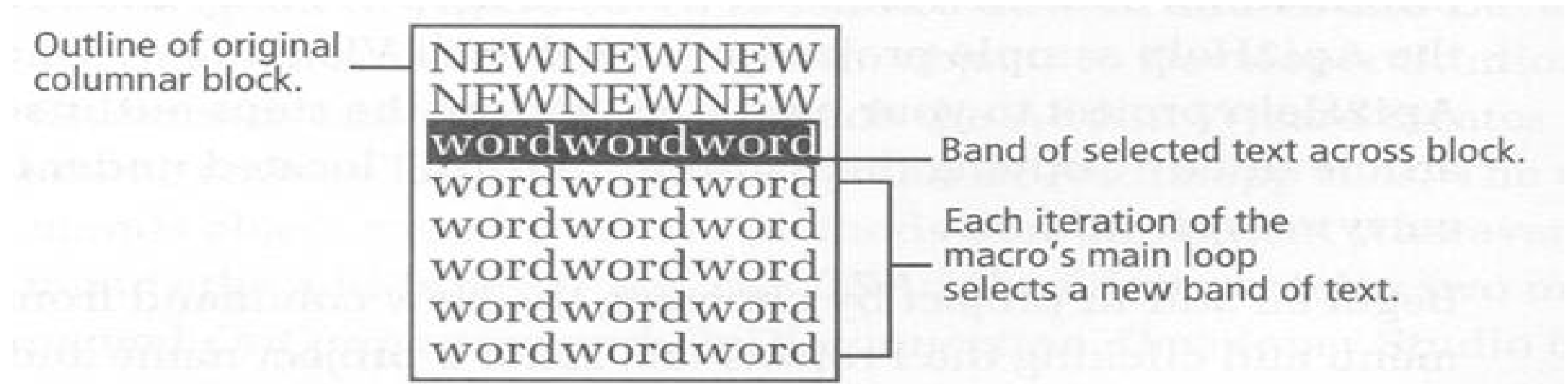
```
Select text across width of column
```

```
Do While ActiveDocument.Selection. CurrentColumn < x2
```

ActiveDocument.Selection.CharRight dsExtend, 1

Loop

如下显示的是选择的文本覆盖原来区域块的结果：



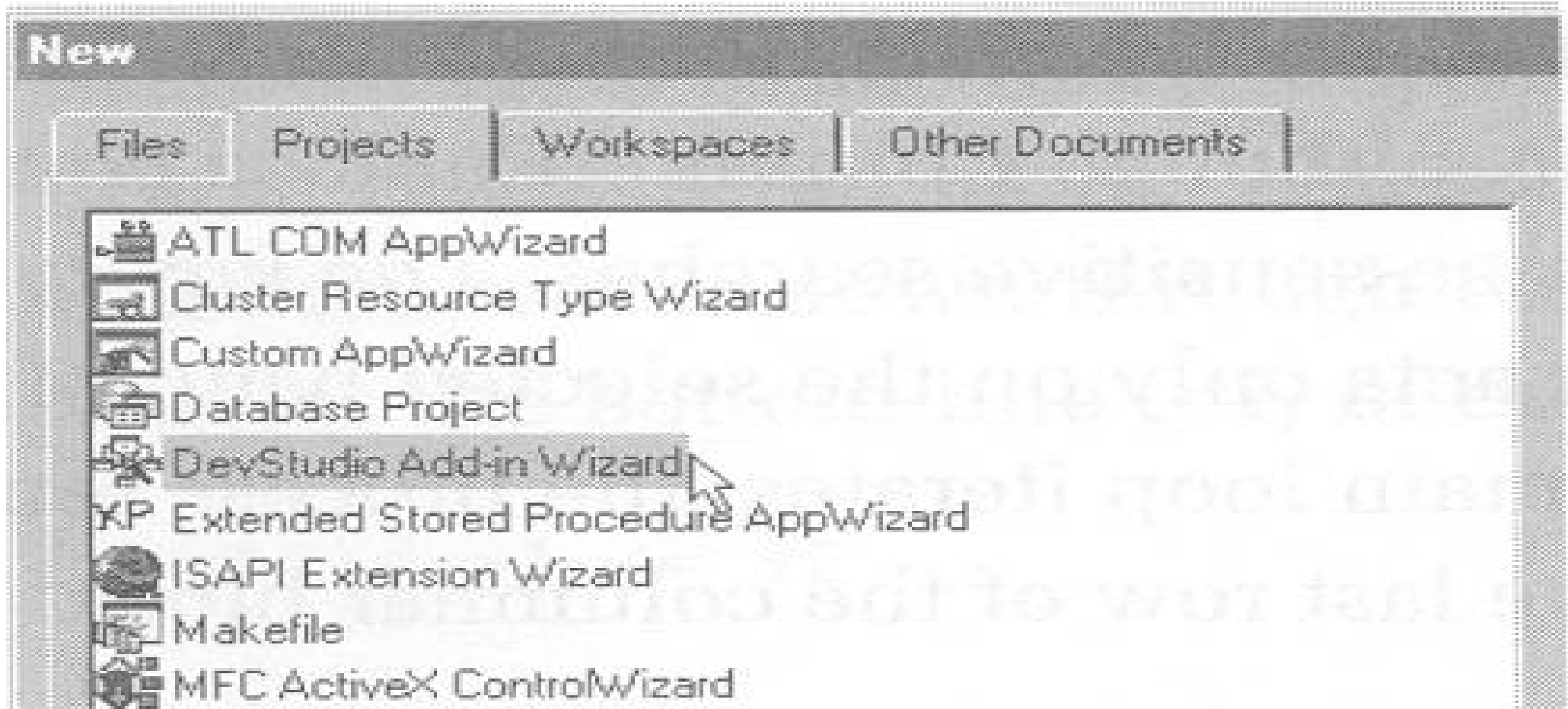
一旦在一行中已经选择了一部分文本，宏就调用 `ReplaceText`。这个 `Developer Studio` 方式使用替换字符串替换每个 `StrFind` 搜索字符串。这种搜索不区分大小写，尽管 `ColumnarReplace` 能轻而易举地进行修改，成为可以区分大小写的方法。`ReplaceText` 只作用于所选文本，不是在整行上，这是整个宏的关键。当主循环重复时，对于所选块最后一行的下一行，这一过程将重复进行。

Developer Studio 附加项

通过创建附加项，有助于增强 Developer Studio 的环境，附加项是 ActiveX 动态链接库，这是 Visual C++ 在启动时加载的，并可以响应用户的命令。附加项可以为环境提供集成的特征，但不能与宏相结合。由于它们可以访问全部 Windows API，附加项能完成以下任务，诸如文件的输入与输出、通信、打印、Internet 支持等所有想让程序做的工作。在使用 InputBox 和 MsgBox 函数显示的对话框中，宏能与用户联系，附加项包含有自己的资源数据，也能显示对话框和自己设计的属性表。附加项的不利之处在于创建时颇费时间和精力。

本节只粗略地描述了在 Developer Studio 中附加项是怎样工作和开始编写的，但由于有关附加项的主题相当深广，其中的许多对象、属性、方法和事件都必须进行说明。所以，详细讨论这个主题超出了本书的范围。更详细消息，可参阅 Visual C++ 的在线帮助，或研究 Visual C++ 提供的 Api2Help 项目的源代码。为了在硬盘上复制 Api2Help 项目，应当按题为 Copying the Sample Add-ins（复制样本附加项）的在线论文中所概括的步骤，它的位置在索引项 sample add-ins（样本附加项）下面。

要开始一个附加项项目，从 File（文件）菜单中选择 New 命令，并单击 Projects 选项卡。输入一个项目名，并双击 DevStudio Add-in Wizard 图标，来启动创建项目的向导，这个向导用一步即可创建项目：



Add-in Wizard 会自动设置项目，并生成源代码，对于大多数编写附加项的工作，这些源代码依赖于 MFC 和 ATL。对于不需要 MFC 的附加项项目，需用 ATL COM AppWizard 来开始项目，而不是 DevStudio Add-in Wizard。从 Insert 菜单中选 New ATL Object（新的 ATL 目标）命令，双击 ATL Object Wizard 对话框中的 Add-in

Object 图标。用 C 或 Visual Basic 也可写附加项，但这样的话，Wizards 所带来的便利和优势就都不能用了。

在编码和建立附加项的动态链接库后，紧接着，把 DLL 文件的信息通知给 Developer Studio，以便加载库，并调用它。调用 Customize 对话框，单击 Add-Ins And Macro Files（附加项和宏文件）选项卡，在浏览了 DLL 文件后，双击表中的该文件。也可将该文件复制到 Common/MsDev98/Add-in 文件夹中，这就没有必要再浏览了。在当前的 Developer Studio 会话中加载附加项库，打开 Customize 对话框中用于附加项文件的复选框。这个复选框处于打开状态会使 Visual C++ 自动将动态链接库文件在启动时自动加载。

图 13-12 说明了附加项和 Developer Studio 是如何相互作用的。一个附加项将两个

对象展现给 Developer Studio，并命名为 Commands 和 DSAddIn，Add-in Wizard 在 Commands.cpp 和 DSAddIn.cpp 文件中创建类的源代码。Commands 对象则容纳着完成附加项提供的全部命令的方法，而 DSAddIn 对象仅容纳名为 OnConnection 和 OnDisconnection 的两种方法。当加载（或连接）附加项时，Developer Studio 调用第一种方法。在卸载附加项时，则需要调用第二种方法。附加项开始执行时，OnConnection 方法获得控件，并调用 Developer Studios 的 AddCommand 函数。调用 AddCommand 给环境增加新的命令，并提供 Developer

Studio 需要呈现给用户的所有命令信息。新的命令通过动态链接库来提供服务，但对用户来说，这个命令与 Developer Studio 内部命令集中的任何命令一样。AddCommand 的参数可以用来指定信息，例如命令名称，在菜单上选择命令后，Developer Studio 的状态栏显示的文本、工具提示的文本和命令的工具栏按钮，以及通过附加项动态链接库导出的方法名称，在用户调用这个命令时，Developer Studio 应当调用该动态链接库。

Developer Studio 也能导出两种其他方法，使得附加项在用户使用时可以更快地访问命令。AddCommandBarButton 可指示 Developer Studio 为附加项导出的命令创建一个工具栏按钮，AddKeyBinding 则可为命令设置组合键。尽管将敲键存入系统中，可以节省用户指定键的麻烦，我们却不推荐调用 AddKeyBinding，因为它有可能会覆盖现存的敲键，给用户引起混淆。通常，在附加项开始后，最好是使附加项命令成为未结合的命令，让用户在 Customize 对话框中设置一个键。对当前键组合的列表来说，附加项无法查询 Developer Studio。

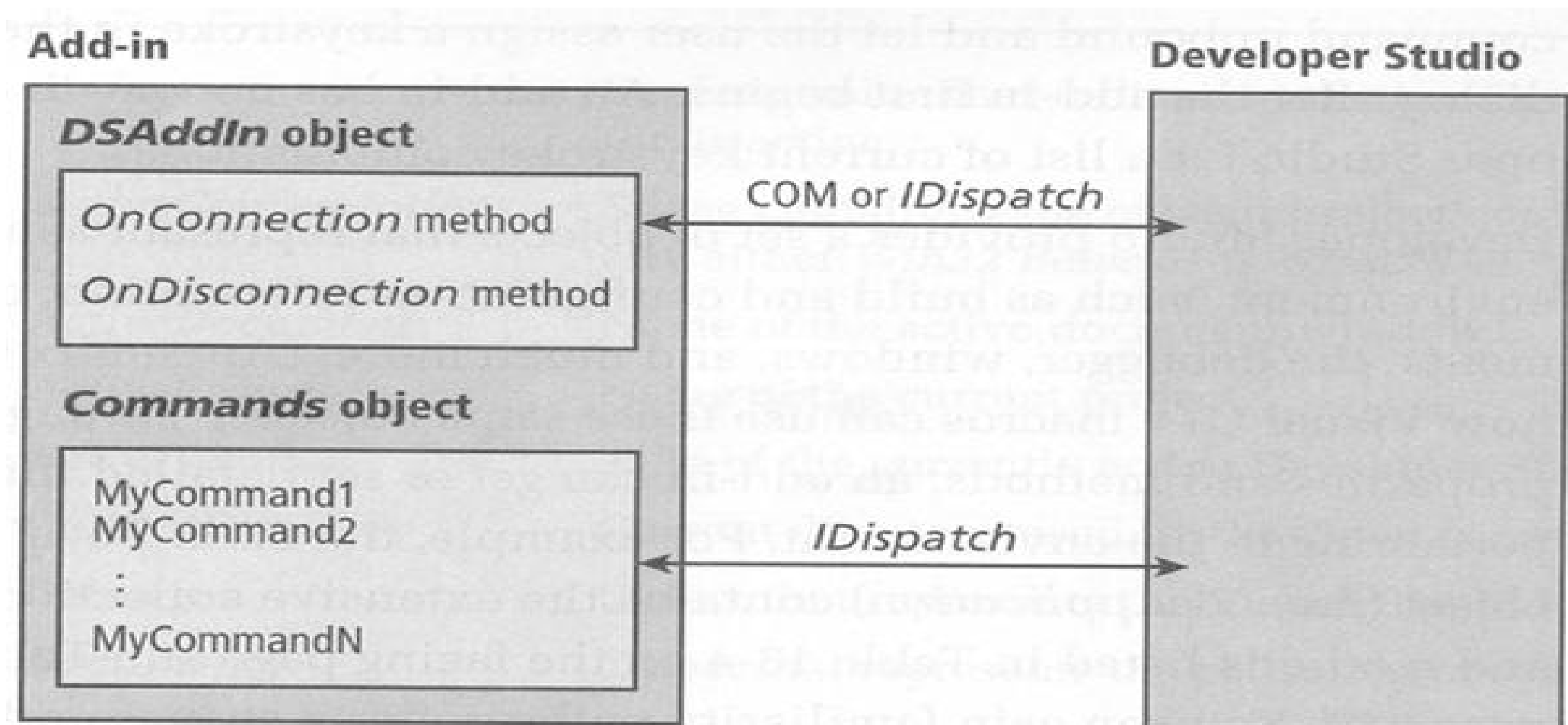


图 13-12 Developer Studio 怎样与附加的动态链接库联系

Developer Studio 提供了一系列代表环境方面的对象，例如建立和配置信息，打开文档、调试器、窗口等等（附录 C 解释了 Visual C++ 宏怎样使用这些对象的）。通过一个对象的属性和方法，附加项能获得或设置与环境相关的详细信息。例

如，在表 13-4 和表 13-5 中，列出了主要的 Developer Studio 对象（名为 Application）所包含的所有属性和方法的集合。在简单的宏脚本中显示出来，可获得熟悉的属性字符串和其他值。例如在一个宏中执行此行：

```
MsgBox(Application.Name+"version"+"Application.Version)
```

产生如下消息：



Application 和 Debugger 是启动事件的唯一 Developer Studio 对象，且 Debugger 仅能启动 BreakPointHit 事件，它通知附加项，在调试器已经触发断点（见第 11 章的“断点”部分，该节专门论述了调试器的断点）。Application 对象能启动 12 种事件，本书将它们列在表 13-6 中，Add-in Wizard（附加项向导）产生的 CCommands 类包含所有启动事件的外壳处理程序函数，要使附加项得知 Developer Studio 的当前状况，可增加实现代码，以便在 Commands.cpp 文件中选

择事件处理函数。

表 13-4 Application 对象的属性

属 性	说 明
Active	用来指示是否 Developer Studio 处于运行状态的布尔值
ActiveConfigurati on	含有当前项目配置的字符串，通常是 Win32 Release 或 Win32 Debug
ActiveDocument	活动文档窗口名
ActiveProject	当前项目名
ActiveWindow	当前活动的 Developer Studio 窗口标题
CurrentDirectory	Open 命令使用的当前目录
Debugger	代表 Visual C++调试器的对象
Documents	代表打开文档的对象
FullName	Developer Studio 可执行的路径和文件名，通常是 C:\Program Files\MicrosoftVisual Studio\Common\MsDev98\Bin\MsDev.exe

Height

Left 含有 Developer Studio 主窗口高度的 long 型值，以像素为单位

Name 含有主窗口左端的 long 型值

Parent 含有 Microsoft Developer Studio 文本的字符串

Path Application 的父对象

Projects Developer Studio 的可执行文件路径，通常是
C:\ProgramFiles\Microsoft Visual Studio\Common\MsDev98\Bin

TextEditor 当前工作空间中代表所有项目的集合对象

Top 代表 Visual C++ 文本编辑器的对象

VersionVisible 含有 Developer Studio 主窗口顶端边界的 y 坐标 long 型值

Width 含有当前 Developer Studio 版本（如 6.0）的字符串 决定
Developer Studio 主窗口是否可视的 Boolean 值

Windows	含有 Developer Studio 主窗口宽度的 long 型值

WindowsState	代表所有打开窗口的集合对象

	代表主窗口状态的 long 型值。值可能包括 dsWindowStateMaximized 和 DswindowStateMinimized 常量，用来指示 Developer Studio 是最大还是最小

表 13-5 Application 对象的方法

方法	说明
AddCommand	对 Visual C++ 添加通过附加项定义的命令
AddCommandBarButton	为附加项创建一个工具栏
AddKeybinding	给附加项命令指定一个键组合
Build	通过只处理变化的文件来创建项目
EnableModeless	在 Developer Studio 中使模态窗口有效或失效
ExecuteCommand	执行一条指定的命令或 VBScript 宏
ExecuteConfiguration	运行通过项目创建的程序
GetPackageExtension	对 Developer Studio 之外的其他对象提供访问

续表

PrintToOutputWindow	将一个字符串写到 Output（输出）窗口之外的 Macro 选项卡中（可参见附录 C 的例子）
Quit	提示用户保存文档, 关闭所有的文档窗口, 并关闭 Developer Studio
RebuildAll	执行 Developer Studio Rebuild All 命令
SetAddInInfo	提供关于附加项的信息

表 13-6 Application 对象的事件

应用程序事件	使用时间
BeforeApplicationShutDown	在 Developer Studio 关闭之前
BeforeBuildStart	用户选择 Developer Studio Build 命令之后，但在编辑开始之前
BeforeDocumentClose	在一个文档关闭之前，当事件发生时，文档仍打开着
BuildFinish	当创建完成时（成功或者不成功）
DocumentOpen	在一个文档打开之后
DocumentSave	在一个文档保存之后，当事件发生时，老文档已经被覆盖
NewDocument	当一个新文档被创建时，当事件发生时，文档处于打开状态
NewWorkspace	当一个新的工作空间被创建时
WindowActive	当一个窗口处于活动状态时。此事件用在编辑器里的文档窗口

续表

WindowDeactivate	和 Developer Studio 应用程序窗口里，例如调试器窗口
WorkspaceClose	在一个窗口失效或被关闭之后
WorkspaceOpen	在一个工作空间被关闭之后
	在一个工作空间被打开以后

第六部分 附 录

附录 A ASCII 和 ANSI 文件格式

Visual C++ 文本编辑器以 ANSI 格式保存文件，它是基于 Windows 的文本编辑器的一种首选格式，如 Windows 95 所带的 Notepad 实用程序。DOS 文本编辑器通常使用相似的 ASCII 格式。ANSI 和 ASCII 格式都是把从 0~256 的数字赋给相应的 256 个字符。

文本编辑器几乎总是以 ASCII 或 ANSI 格式保存文件，而不是以像 Word 处理器那样的专用格式。起初，缩写词 ASCII 指的是一种约定：它把从 32~127 的数字赋给相应的 96 个字符，包括数字、标点符号、大小写字母。这个数字很适合由 7 位组成的值的范围（因为 2 的 7 次方等于 128）。ASCII 约定留出 0~31 作为打印机控制代码，并保留每个字节的第 8 位用于奇偶校验。因为现在的内存不要求奇偶校验，所以，每个字节的第 8 位可以用于表示数据，即用一个字节可代表的字符数翻倍，达到 256。现在，我们所认为的 ASCII 字符集包括 0~31 的值，最初的 96 个 ASCII 字符，以及由 IBM 的最初 PC 设计者用额外的第 8 位添加的另外 128 个字符。这额外的 128 个字符，作为高位 ASCII 或者 IBM 字符集，如表 A-1 所示。

在高位 ASCII 集中的一些字符，尤其是画框字符，在首选 ANSI 标准的 Windows

这样的图形环境中不能正常工作。ASCII 和 ANSI 在前 128 个字符中，大多数是一致的，但对更高的字符，却指定了不同的数字。这就是使用基于 DOS 的文本编辑器中创建包括高位 ASCII 字符的文档，而在 Visual C++ 文本编辑器中这些字符就显示为另一种形式的原因。

从技术上来说，任何文件都由 ASCII（或 ANSI）字符组成。但在 ASCII 和 ANSI 格式中，每一个字符具有不同的面值。Z, é 和 ¼ 即为“Z,”，“é,”和“¼,”。字符只代表字符本身，并不代表代码、命令或别的东西。这项规则的唯一例外是制表符和回车符。制表符是一个代表一定数目空格的字符（ASCII 值为 9）。用于标识文本行结尾的回车符格式取决于编辑器和操作系统。在 DOS 和 Windows 环境中，一个回车符包括一对由 ASCII 值 13 和 10 组成的字符。ASCII 13 被称为回车，通知光标移到行的开始处。而 ASCII 10，被称为换行，指示移向下一行。这两个字符的顺序是很重要的；ASCII 13 必须在 ASCII 10 之前，否则，回车符通常就不会被识别。在 UNIX 操作系统中，换行独自作为行结尾标识，它暗示着一个新行和一个回车。Visual C++ 文本编辑器同时识别这两种回车符形式，即 ASCII 13-10 成对使用，或者是单独使用 ASCII 10。

表 A-1 高位 ASCII 集，值从 128~255

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	€	160	A0	á	192	C0	Ł	224	E0	α
129	81	ü	161	A1	í	193	C1	ł	225	E1	β
130	82	é	162	A2	ó	194	C2	Ť	226	E2	Γ
131	83	â	163	A3	ú	195	C3	ţ	227	E3	π
132	84	ä	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	à	165	A5	ñ	197	C5	†	229	E5	σ
134	86	ã	166	A6	•	198	C6	‡	230	E6	μ
135	87	ç	167	A7	•	199	C7	‡	231	E7	τ
136	88	ê	168	A8	¿	200	C8	Ł	232	E8	ξ
137	89	ë	169	A9	ƒ	201	C9	Ŧ	233	E9	θ
138	8A	è	170	AA	¬	202	CA	Ł	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	Ŧ	235	EB	δ
140	8C	î	172	AC	¼	204	CC	Ł	236	EC	∞
141	8D	ì	173	AD	↓	205	CD	=	237	ED	ϕ
142	8E	ñ	174	AE	«	206	CE	Ł	238	EE	€
143	8F	ñ	175	AF	»	207	CF	Ł	239	EF	π
144	90	é	176	B0		208	D0	„	240	F0	≡
145	91	•	177	B1	■	209	D1	Ť	241	F1	±
146	92	ff	178	B2	■	210	D2	π	242	F2	≥
147	93	õ	179	B3		211	D3	„	243	F3	≤
148	94	ö	180	B4	†	212	D4	Ł	244	F4	Γ
149	95	õ	181	B5	†	213	D5	Ŧ	245	F5	J
150	96	û	182	B6		214	D6	π	246	F6	÷
151	97	ù	183	B7	π	215	D7		247	F7	≈
152	98	ÿ	184	B8	Ŧ	216	D8	†	248	F8	*
153	99	ÿ	185	B9		217	D9	J	249	F9	.
154	9A	U	186	BA		218	DA	Ŧ	250	FA	.
155	9B	©	187	BB	Ŧ	219	DB	■	251	FB	√
156	9C	£	188	BC		220	DC	■	252	FC	π
157	9D	¥	189	BD	„	221	DD		253	FD	z
158	9E	£	190	BE	J	222	DE		254	FE	■
159	9F	f	191	BF	Ŧ	223	DF	■	255	FF	

表 A-2 列出了 US ANSI 集的全部 256 个字符，其中一些 Windows 不能显示。这些不能显示的 ANSI 字符通常在屏幕上显示为一个小框或空格，表 A-2 中所示的是像  一样常见的小框。这个表为每个字符列出八进制值，而不是十六进制值。知道一个字符的八进制值，可以让用户通过在一个反斜杠后键入字符的八进制值来在一个资源串中或静态文本框中加入该字符。例如，为了在一个串中加入 ANSI 字符 ½，只要输入 \275。注意，Windows 能够显示一些字符，如只能用 TrueType 字体的商标符号 ™。

表 A-2 ANSI 字符集，值从 0~255

Dec	Octal	Char	Dec	Octal	Char	Dec	Octal	Char	Dec	Octal	Char
0	000	□	20	024	□	40	050	(	60	074	<
1	001	□	21	025	□	41	051	)	61	075	=
2	002	□	22	026	□	42	052	*	62	076	>
3	003	□	23	027	□	43	053	+	63	077	?
4	004	□	24	030	□	44	054	,	64	100	@
5	005	□	25	031	□	45	055	-	65	101	A
6	006	□	26	032	□	46	056	.	66	102	B
7	007	□	27	033	□	47	057	/	67	103	C
8	010	□	28	034	□	48	060	0	68	104	D
9	011	□	29	035	□	49	061	1	69	105	E
10	012	□	30	036	□	50	062	2	70	106	F
11	013	□	31	037	□	51	063	3	71	107	G
12	014	□	32	040	□	52	064	4	72	110	H
13	015	□	33	041	!	53	065	5	73	111	I
14	016	□	34	042	"	54	066	6	74	112	J
15	017	□	35	043	#	55	067	7	75	113	K
16	020	□	36	044	\$	56	070	8	76	114	L
17	021	□	37	045	%	57	071	9	77	115	M
18	022	□	38	046	&	58	072	:	78	116	N
19	023	□	39	047	'	59	073	;	79	117	O

Dec	Octal	Char	Dec	Octal	Char	Dec	Octal	Char	Dec	Octal	Char
80	120	P	110	156	n	140 ^{TT}	214	Œ	170	252	ª
81	121	Q	111	157	o	141	215	□	171	253	«
82	122	R	112	160	p	142	216	□	172	254	¬
83	123	S	113	161	q	143	217	□	173	255	-
84	124	T	114	162	r	144	220	□	174	256	®
85	125	U	115	163	s	145	221	'	175	257	—
86	126	V	116	164	t	146	222	'	176	260	°
87	127	W	117	165	u	147 ^{TT}	223	"	177	261	±
88	130	X	118	166	v	148 ^{TT}	224	"	178	262	²
89	131	Y	119	167	w	149 ^{TT}	225	□	179	263	³
90	132	Z	120	170	x	150 ^{TT}	226	—	180	264	´
91	133	[	121	171	y	151 ^{TT}	227	—	181	265	µ
92	134	\	122	172	z	152 ^{TT}	230	~	182	266	¶
93	135	]	123	173	{	153 ^{TT}	231	™	183	267	·
94	136	^	124	174		154 ^{TT}	232	§	184	270	˘
95	137	_	125	175	}	155 ^{TT}	233	>	185	271	†
96	140	`	126	176	~	156 ^{TT}	234	œ	186	272	°
97	141	a	127	177	□	157	235	□	187	273	»
98	142	b	128	200	□	158	236	□	188	274	¼
99	143	c	129	201	□	159 ^{TT}	237	Ÿ	189	275	½
100	144	d	130 ^{TT}	202	,	160	240	□	190	276	¾
101	145	e	131 ^{TT}	203	f	161	241	i	191	277	ℓ
102	146	f	132 ^{TT}	204	"	162	242	€	192	300	À
103	147	g	133 ^{TT}	205	...	163	243	£	193	301	Á
104	150	h	134 ^{TT}	206	†	164	244	¤	194	302	Â
105	151	i	135 ^{TT}	207	‡	165	245	¥	195	303	Ã
106	152	j	136 ^{TT}	210	^	166	246	¡	196	304	Ä
107	153	k	137 ^{TT}	211	‰	167	247	§	197	305	Å
108	154	l	138 ^{TT}	212	Š	168	250	¨	198	306	Æ
109	155	m	139 ^{TT}	213	‹	169	251	©	199	307	Ç

(continued)

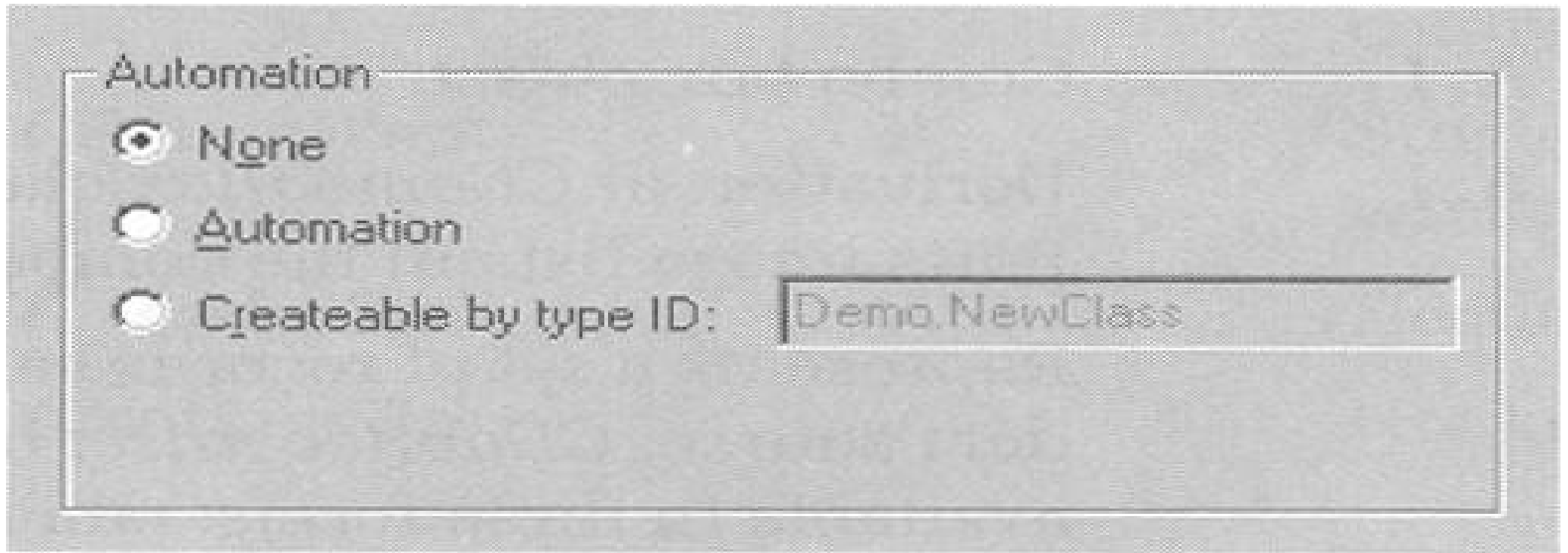
表示 Windows 不显示此字符

仅表示 TrueType 字体。

附录 B ClassWizard 支持的 MFC 类

正如第 6 章所讲的，ClassWizard 是一个被设计成帮助用户创建 MFC 派生类的工具。本附录列出并简单描述了 MFC 类，ClassWizard 能用这些 MFC 类为派生类生成初始代码。列表中包括这个版本 MFC 新的 5 个类：CComboBoxEx，CDateTimeCtrl，CHtmlView，CIPAddressCtrl，和 CMonthCalCtrl。

ClassWizard 会显示 New Class（新类）对话框，以提示输入新类的名字和基类，如图 6-4 所示。在对话框的 Automation 组框内的单选按钮决定新类是否支持 Automation（自动化）或提供一个类型标识符。



标识符是名称，更精确地说，它是编程标识符或 ProgID 的名字，通过它，客户程序能够使用 Automation 创建类的一个对象。MFC 宏 IMPLEMENT_OLECREATE 把程序设计标识符录制到系统注册表中，以作为该类类识别号的别名。

并不是所有的 MFC 类都支持 Automation 选项。在这种情况下，New Class 对话框中的单选按钮就被禁用。表 B-1 的支持列包括一个 A 或 I 代码，用于指示哪一个类支持 Automation 或类型标识。支持列中的 D 符号标识基于对话框的类，如

Cdialog，它要求对话框资源。

表 B-1 ClassWizard 识别的 MFC 类

MFC 类	支持 *	描述
CAnimateCtrl	A,I	自动化通用控件
CAsyncMonikerFile		在 ActiveX 控件中提供对异步标记的支持
CAsyncSocket		封装 Windows Sockets API，参看 CSocket
CButton	A,I	按钮控件对象
CCacheDatapathProperty		允许一个 ActiveX 控件异步传输属性数据和缓冲内存中的数据，参考 CDataPathProperty
CCmdTarget	A,I	所有能够接收和响应消息的对象的基类
CColorDialog		颜色选择的通用对话框，提供为显示系统定义的颜色列表

续表

CComboBox	A,I	组合框对象
CComboBoxEx	A,I	CcomboBox 类的派生类，用于支持在组合框控件中的图像列表
CDaoRecordset		代表选自数据源的记录集。CDaoRecordset 对象可用于三种格式：表类型记录集，动态集类型记录集和快照类型记录集
CDaoRecordView	D,A	提供表单视图，以在控件中显示数据库记录。表单视图是 CDaoRecordset 对象的一部分。参考 CFormView 和 CRecordView
CDataPathProperty		实现一个 ActiveX 控件属性，它能够异步加载其数据。这个类允许 ActiveX 控件在后台下载属性数据时被激活

续表

CDateTimeCtrl	A,I	封装新的日期/时间选取器控件（在第7章中的DirList2示例程序中演示过）
CDialog	D,A	用于包含控件窗口的对话框对象
CDocument	A	用于管理程序的数据的类
CDragListBox	A,I	Windows 列表框,允许用户把其中的项拖放到不同的位置
CEdit	A,I	用于文本输入的子窗口控件
CEditView	A	提供 Windows 编辑控件的功能。因为 CEditView 派生于 Cedit, 该对象可同文件和文件模板一同使用
CFileDialog		通用文件对话框, 提供 Open 和 Save As 对话框中的功能

续表

CFontDialog		通用字体对话框,用于显示当前已装入系统的字体列表
CFormView	D,A	包含对话框控件的窗口
CFrameWnd	A	SDI (单窗口界面) 框架窗口
CHeaderCtrl	A,I	标题通用控件
CHotKeyCtrl	A,I	热键通用控件
CHtmlView	A	实现 Web Browser 控件的视图类,能够访问本地或 Web 上的 HTML 文件。参考 Gallery 中的 Microsoft Web Browser ActiveX 控件
CHttpFilter		创建并处理超文传输协议过滤器对象,该对象用于过滤用于 HTTP 请求的服务器通知
CHttpSever		Internet Server API (ISAPI) 的包装类

续表

CIPAddressCtrl	A,I	IP 地址控件。类似于编辑框，该控件接收 Internet 协议格式的地址
CListBox	A,I	列表框对象
CListCtrl	A,I	列表视通用控件。示例可参考第 5 章中的 MfcTree 项目
CListView		简化 CListCtrl 的使用，添加了对文件和视图的支持
CMDIChildWnd	A	MDI（多文档界面）子框架窗口
CMiniFrameWnd	A	半高的框架窗口，主要用于浮动工具栏。一个小框架窗口没有最小化和最大化按钮，但其他都类似于正常的框架窗口
CMonthCalCtrl	A,I	月历控件，用于显示一个用户可选择日期的日历

续表

COleDocument	A	把一个文件看作为 CDocItem 对象的一个集合。包容器和服务器都需要这个结构,因为它们的文件必须能够包含 OLE 项
COleLinkingDoc	A	OLE 包容器文件的基类,这些文件支持对它们所包含项的链接
COleSeverDoc	A	OLE 服务器文件的基类
COleServerItem	A	为 OLE 项提供一个服务器界面
CPrintDialog		通用打印对话框,提供 Print 和 Print Setup 对话框中的功能
CProgressCtrl	A,I	通用进程指示器控件
CPropertyPage	D,A	代表属性表单中的一页。参考第 5 章中的 DirList1 示例

续表

CPropertySheet	A	属性表，也叫做多选项卡对话框。一个属性表由一个 CpropertySheet 对象和几个 CpropertyPage 对象组成
CRecordset		用于访问数据库表或查询的类
CRecordView	D,A	包含对话框控件的窗口
CRichEditCtrl	A,I	用户能够输入和编辑文本的窗口，提供字符和程序段格式，以及对嵌入 OLE 项的支持
CScrollBar	A,I	滚动条对象
CScrollView	A	可滚动的窗口，派生于 Cview
CSliderCtrl	A,I	提供包含一个滑块和可选的刻度线的窗口。关于如何使用 CSliderCtrl 的示例，参考第 5 章的 Color 项目

续表

CSocket		Windows Socket API 的包装类
CSpinButtonCtrl	A,I	提供箭头按钮, 用户可单击它, 以增加或减少某个控件中的一个值。参考第 7 章中 DirList 项目
CStatic	A,I	用于标识另一个控件或给用户提供消息的简单文本框
CStatusBarCtrl	A,I	提供一个层次窗口, 通常放于父窗口的底部, 用于显示关于应用程序的状态信息
CTabCtrl	A,I	允许应用程序在一个窗口或对话框的同一区域显示多个页面
CToolBarCtrl	A,I	工具栏通用控件
CToolTipCtrl	A,I	提供工具提示控件的功能, 它以一个弹出窗口的样子显示, 包含描述某个工具用途的一行文本

续表

CTreeCtrl	A,I	显示项的分层结构列表
CTreeView		简化 CTreeCtrl 的用法
CView	A	用于显示程序数据的类
CWinThread		代表一个应用程序中的一个线程
generic CWnd	A,I	自定义的窗口
splitter	A	包含一个 CSplitterWnd 类的 MDI 子窗口。用户能够把这个窗口分割成多个窗格

*支持符号：D=Dialog A=Automation I=Programmatic 标识符

附录 C VBScript 初步

Visual Basic Scripting 版本，即我们所熟知的 VBScript，是 Visual Basic for Applications (VBA) 的一个子集，而 VBA 又是 Visual Basic 的一种方言。通过与 Visual C++ 组合作为它的宏语言，最后 VBScript 给 Visual C++ 带来了强有力的宏能力。VBScript 被打包为一个编译器（更精确地说，是一个翻译器）和一个运行库。用 VBScript 编写的宏脚本能够重建一系列的 Developer Studio 命令，自动执行手工做的几乎全部的任务。而且，通过引用 VBScript 运行库，一个宏能够执行在 Visual C++ 中不可能做的许多其他任务，如在一个标准 Windows 消息框中显示消息和查询用户的输入。

Visual C++ 提供了两种创建宏的技术：录制和编程。大多数宏只需录制一序列的动作就能够创建了，这项技术不需要对 VBScript 的理解。在幕后，环境会自动创建一个 VBScript 宏文件，以复制录制的命令。但如第 13 章所演示的，在定制 Visual C++ 时，录制宏是有局限性的。一个录制的宏是“硬连接的”，能很好地适用于重复一定的动作集合，但对于响应宏运行时的不同环境却是不够的。宏要有适应能力就需要编程，这就要求有关 VBScript 的知识。幸运的是，VBScript 是很容易学会的，C/C++ 程序员会很快地赏识它的许多特性。用户已经知道了 VBScript 的许多内容，即使用户以前没有见这种语言。本附录添补了一些知识，

提供了针对 C/C++ 程序员的宏编程入门指南。为了学习 VBScript 更多的内容，尤其是它在文件中的应用，请参考关于这方面的大量书籍，或者访问 Microsoft 的 Web 网址：

<http://www.microsoft.com/vbscript>

本附录的第一部分描述了 VBScript 中的各种语言元素，如变量、程序流语句和过程。每项讨论都用注释的代码段举例说明。VBScript 中的注释以一个单引号开始，直到行末。VBScript 也承认 BASIC 中老式的 Rem 语句，但 Rem 已很少用了：

```
' This is a comment
```

```
Rem So is this
```

本附录的第二部分集中于 VBScript 库中的函数上，为每个函数提供了一个简短的描述，并常有示例代码来演示。

C.1 变量

VBScript 默认一种名为 Variant 的数据类型，根据上下文，它可以包含数字或文本消息。如果把数字数据赋给 VBScript 中的一个变量，这个变量就作为一种数

字数据类型，以匹配该数据。把文本数据赋给一个 Variant 变量，就把它变成了一个字符串。VBScript 提供了几个函数，用于转换内部数据类型。表 C-1 列出了一些 Variant 能够模拟的数据类型。

变量名应该表示出变量的内部子类型的含义。在给变量命名时，使用匈牙利符号或一种相似的约定，用以指示该变量的数据类型。像 bFlag, iNumber 和 strString 这样的名字，可以很容易地辨认出包含 BOOL, int 和文本数据的变量来。

表 C-1 用 Variant 模拟的数据类型

数据类型	说明
BOOL	或者为 TRUE（非零），或者为 FALSE（零）
BYTE	8 位的无符号整数，值从 0 到 255
int	16 位的有符号整数，值从 -32,768 到 32,767
long	32 位的有符号整数，值从 -2,147,483,648 到 2,147,483,647
float	单精度的浮点值，其负值变化范围从 -3.402823E38 到 -1.401298E-45，其正值变化范围从 1.401298E-45 到 3.402823E38

续表

<code>double</code>	双精度的浮点值，其负值变化范围从 -1.79769313486232E308 到 -4.9406564584124E-324，其正值变化范围从 4.9406564584124E-324 到 1.79769313486232E308
<code>Date</code>	表示从 100 年的 1 月 1 日到 9999 年的 12 月 31 日的日期和时间
<code>String</code>	BSTR 类型的字符串，长度可为约 20 亿个字符

变量名称必须遵守下列规则：

- 以一个希腊字母开头
- 不能包含句点
- 最大长度为 255 个字符
- 在它被声明的范围内必须是唯一的

除了数组外，不必在变量使用前声明它。如果用户喜欢在程序的开头声明变量，请使用 `Dim` 语句（`dimension` 的缩写），如下：

```
Dim x
```

```
x=3
```

```
y=x 'This is statement is legal ,even though y was not declared
```

为了易捉书写错误，用户可以通过包含 Option Explicit 语句，在一个宏中强制变量。有了这个语句，只有脚本用一个 Dim 语句预先声明了变量，使用该变量合法的。在宏的顶部放入 Option Explicit 语句，如下所示：

```
Option Explicit
```

```
Dim x
```

```
x=3 'Legal
```

```
y=x 'Not legal ,because y has not been previously declared
```

Const 关键字在 C 语言和 VBScript 中有相同的功能。对于要保存静态数据的变量，请在变量的初始赋值语句中用 Const。此后，VBScript 解释器不允许再对该变量赋值：

```
Const x=3
```

```
x=5 'This statement causes an error
```

数 组

宏必须专门用 Dim 语句声明数组：

```
Dim iArray(10) 'A one-dimensional array
```

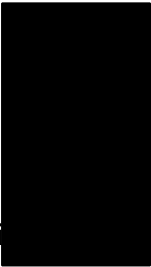
像 C 语言一样，VBScript 中的数组是基于零的。但与 C 中的声明不同的是，Dim 语句的数组下标指定最高元素下标，而不是元素的数目。因此，上面的声明实际上分配了 11 个元素，从 iArray(0)到 iArray(10)。一个数组最多可有 60 维，由 Dim 语句中的用逗号分隔的一系列下标指明：

```
Dim iArray(5,10) 'A two-dimensional array
```

```
Dim iArray(5,10,15) 'A three-dimensional array
```

没有下标列表的数组是动态的，这意味着，宏能够在运行时改变数组的大小。用 ReDim 语句重新设置一个动态数组的大小，如下：

```
Dim iArray() 'Define but don't allocate an array
```



```
ReDim iArray(100) 'Allocate space for 101 elements
```

```
ReDim Preserve iArray(50) 'Resize the array for 51 elements
```

ReDim 能够调整一个数组，使之变大或变小。在这个程序段中的数组 iArray 最初有 101 个元素，但后来减少到 51 个元素。Preserve 关键字用来确保变小的数组保留前 51 个元素的值，尽管在数组调整大小时，从元素 iArray(51)到元素 iArray(100)会丢失。在调整数组大小时，不用 Preserve 关键字，数组的原来值会被清除。VBScript 没有限制宏调整数组大小的次数。但是，如果代码用 Dim 语句声明了一个有下标的数组，该数组就不能用 ReDim 语句来调整大小。企图这样做会导致错误。

字符串

像在 C 中一样，可把双引号中的文本赋给字符串变量：

```
strName = "Johe Q.Public"
```


用英镑符号包围含有日期的字符串：

```
dateBirth = #07-04-76#
```

运算符

除一个以外，所有的 VBScript 运算符可分为三类：算术，比较，逻辑。当不同的运算符出现在同一个表达式中时，算术运算符拥有最高级，即 VBScript 在执行其他类的运算符前先进行算术操作，如加和乘。比较运算符居次，逻辑运算符最后。像在 C 语言中的一样，括号中的运算符比括号外的运算符先执行，而不管类的优先级如何。下面的两行语句演示了括号是如何改变 VBScript 执行表达式中的运算符顺序的：

```
x+y And z      'AND the variable z with the sum of  x+y
```

```
x+(y And z)    'Add to the variable x the result of y AND z
```

优先级次序也存在于算术和逻辑运算符中。比较运算符具有相同的优先级，按照在表达式中出现的顺序由左到右执行。表 C-2 列出了 VBScript 运算符，算术和逻辑运算符按照优先级从高到低放置。这样，表中的优先级按照从左到右和从上到下递减。

表 C-2 按递减顺序排列的 VBScript 运算符

算术运算符		比较运算符		逻辑运算符	
说明	符号	说明	符号	说明	符号
幂	^	等于	=	非	Not
负	-	不等于	<>	逻辑 AND	And
乘法	*	小于	<	逻辑 OR	Or
除法	/	大于	>	异或	Xor
整数除法	\	小于或等于	<=	等于	Eqv
算术求模	Mod	大于或等于	>=		
加法	+				
减法	-				

像 C 语言一样，乘除运算符有同样的优先级。加减运算符也是如此。VBScript 提供了一个字符串连接符 (&)，它不属于表 C-2 中所列的任何一类。它的优先

级位于算术和比较运算符之间。

控制程序流

宏通过条件分支语句和循环语句以及调用过程来控制程序流。VBScript 不识别标号，不存在 Goto 语句的等价语句，这意味着，宏不能无条件地从脚本中的一个地方跳到另一个地方。有一种称为 GoToLine 的方法，但 GoToLine 只用于在文件中移动符号，与正在执行的宏中控件程序流没有任何关系。

条件分支语句

If...Then...Else 语句在功能上等同于 C 中的 if...else 语句，只是 VBScript 不用在大括号来包围代码段：

```
If ActiveDocument.Selection="" Then str="No Selection"
```

```
Else str=ActiveDocument.Selection
```

包含两个或多个语句的条件语句必须用一个 End If 语句来结尾：

```
If x<y Then
```

```
iLine=ActiveDocument.Selection.CurrentLine
```

```
iCol =ActiveDocument.Selection.CurrentColumn
```

```
End If
```

当一个条件语句因为使用 If...Then...Else 进行简单的测试而变得太复杂时，VBScript 的 Select Case 命令提供了一个更好的办法。该命令提供了一种基于变量值的条件分支方法，十分简洁。它和 C 语言的 switch 语句非常相似。像在 C 语言中的一样，每一个分支由一个 case 语句和一个唯一值来标记。当一个 case 语句的值和 Select Case 命令中的值相同时，控件就会到达这个 case 语句：

```
Select Case i
```

```
Case 1
```

```
'Come here when i = 1
```

```
Case 2
```

```
'Come here when i=2
```

```
'And so forth
```

```
End Select
```

VBScript 不提供 break 关键字，所以，每个 case 语句都意味着前一个 case 块的结束。当一个块结束后，程序流继续执行 End Select 语句下面的一句，End Select 语句用于结束 Select Case 程序段。Case Else 语句执行和 C 中 default 关键字同样的功能，它用于标记当控制不经过其他任何 Case 语句时所要执行的代码段：

```
Select Case strColor
```

```
    Case    "red"           strHiLite="magenta"
```

```
    Case    "blue"         strHiLite="cyan"
```

```
    Case    "brown"        strHiLite="yellow"
```

```
    Case    Else           strHiLite="undefined"
```

```
End Select
```

循 环 语 句

VBScript 提供几种循环结构，这几种循环对 C/C++ 程序员来说是很熟悉的：

- Do While...Loop 或 Do...Loop While：当条件为真时重复运行
- Do Until...Loop 或 Do...Loop Until：一直运行到条件为真时
- For...Next：在一个循环计数器的控制下重复运行

Do 关键字作为一个循环块的开始，Loop 语句作为它的结束句。While 和 Until 关键字既能够出现在块顶部的 Do 语句中，又能够出现在块末端的 Loop 语句中。这取决于用户想让 VBScript 解释器在执行循环之前检查条件，还是在执行循环之后检查条件。下面的几个代码段说明了 VBScript 中循环语句的正确格式：

```
Do While x<y
```

```
    'Loop is not entered unless x is less than y
```

```
Loop
```

```
Do
```

```
    'Loop executes at least once and repeats only if i is less than j
```

```
Loop While i<j
```

Do Until x=10

'Loop is not entered if x is equal to 10

'When x attains a value of 10, the loop exits

Loop

Do

'Loop executes at least once and repeats only

'until I is not equal to j

Loop Until I<>j

默认情况下， For...Next 循环中的计数器是以 1 为步长的：

For i=1 To 10

‘This loop iterates 10 times, incrementing I from 1 to10

Next

在 For...Next 循环中，可以用 Step 关键字来为循环计数器指定一个不同的步长：

```
For I=1 to 10 Step 2
```

```
    'This loop iterates 5 times
```

Next

```
For i=10 To 2 Step -2
```

```
    'This loop also iterates 5 times
```

Next

上面的第一个 For...Next 循环重复运行了 5 次。它用数值 1 初始化计数器，然后在后面的每一次循环中增长，其值为 3，5，7，9。当最后一次通过循环时，i 的值为 9；当 i 达到 11 时退出循环。第二个 For...Next 循环也重复运行了 5 次，但是当循环进行时，i 不是增长，而是减少，因为 Step 值是负的。i 的初始值为 10，它在每一次循环时都减少，其值为 8，6，4，2。当退出循环时，i 的值

过程

VBScript 除了它自己的内部函数库外，还在宏脚本中识别两种类型的过程，Sub 和 Function。这两种类型都有参数，但只有 Function 返回值。除此以外，它们几乎没有什么区别。

每一个宏脚本都有一个主 Sub 过程，它的名字决定了出现在 Visual C++ Macro 对话框中的宏名。主 Sub 过程首先出现在宏脚本中，不带参数。宏的可调用过程按任意顺序排在主程序的后面。如下所示，每个 Sub 过程（包括第一个）都以 End Sub 语句结

```
Sub MacroName 'Main procedure
```

```
End Sub
```

```
Sub Subroutine1(arg1, arg2)
```

```
    'Code for first callable procedure goes here
```

```
End Sub
```

```
Sub Subroutine2(arg1, arg2)
```

```
    'Code for second callable procedure goes here
```

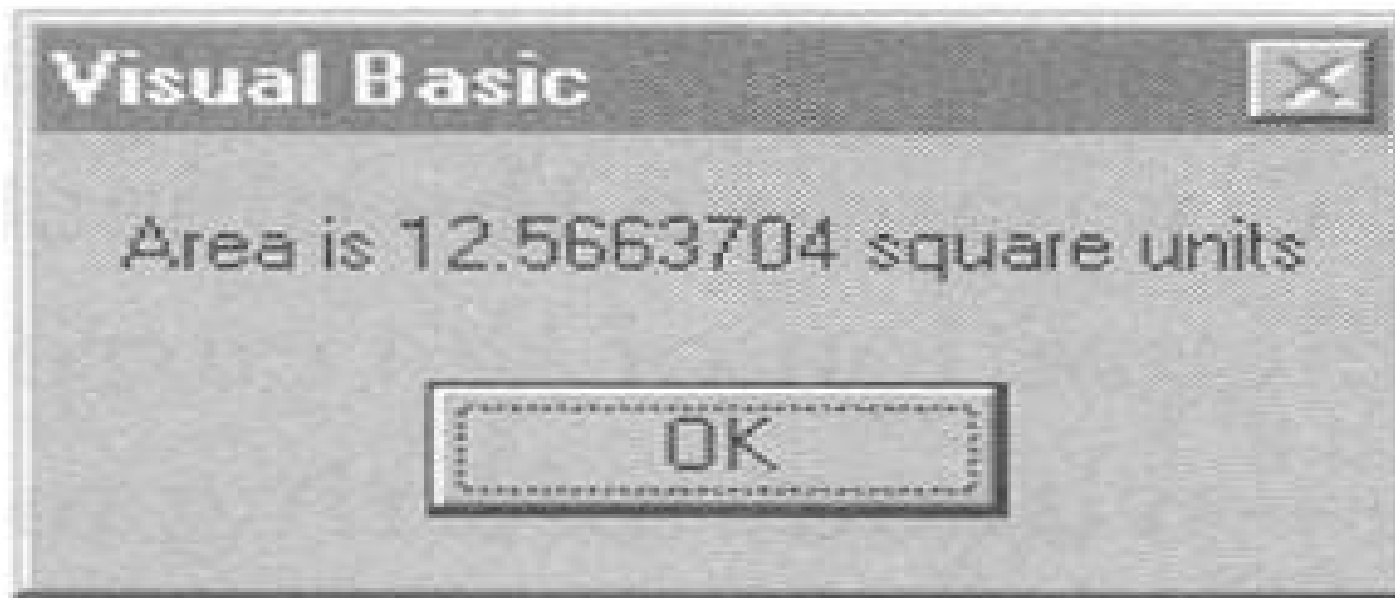
```
End Sub
```

一个可调用的 Sub 过程（即除了第一个以外的任何 Sub 过程）能够通过下面两种方式被调用：Call 语句，以及一个过程名作为单独的语句。这模仿了 C/C++ 中调用函数的方式。这两种方法的格式有些细微的差别。Call 语句要求把 Sub 过程的参数放在程序名后面的括号中。如果不用 Call 语句，程序名后的参数就可以不用括号，只用逗号分隔开来。所以下面这两句的作用是相同的：

```
Call AnySub(param1,param2)
```

```
AnySub param1,param2
```

因为 Function 过程返回值，所以，它能够像 C 函数一样作为一个值来使用。为了返回一个值，Function 过程必须包含与程序同名的变量。当退出过程时，解释器会返回一个值给调用程序。因为所有的值都属于 Variant 类型，解释器不对返回值进行类型检查。与 Sub 过程不同的是，Function 过程的参数列表总是出现在括号内。如果一个 Function 过程没有参数，它就必须包括一对空括号。下面是一个根据半径计算圆面积的简单例子：



VBScript 中参数是按值传送的，而不是按指针传送。不管是 Sub 过程，还是 Function 过程，都不能改变外部数据，除非是全局变量。VBScript 按级来区分作用域，所以局部变量被称为“过程级”。一个过程级变量只在被声明和过程结束这段时间内才是存在的。既然一个过程变量有局部使用范围，它就不能在下一次进入该过程时保留着它的值。全局作用域的变量是脚本级的，在宏的整个执行过程中一直存在。在过程外部声明的所有变量都有脚本级的范围，如下所示：

```
Dim iGlobal
```



'This variable has script-level scope

```
Sub Main()
```

```
Dim iLocal          'This variable has procedure-level scope
```

```
End Sub
```

一个过程最多可有 127 个变量；一个数组作为一个变量。

对象

Developer Studio 的不同方面对一个运行的宏来说，可看作是一个 17 个不同对象的集合。例如，调试器可以像 Visual C++ 编辑器、窗口等等一样被看作是一个对象。每个对象都有属性和方法，宏通过这些属性和方法来了解或调整对象的状态。主要的 Developer Studio 对象被称为 Application，它的属性、方法和事件列在表 12-4、12-5 和 12-6 中。在宏脚本中 Application 是默认对象，所以用户不必专门指定该对象，就可以使用 Application 的元素。例如，下面的这个命令显示了当前项目的活动配置（无论是 Win32 Release 还是 Win32 Debug），而不必引用 Application 对象：

```
MsgBox("The configuration is "&ActiveConfiguration)
```

为了让用户对对象有一些认识，这一节讨论 TextSelection 对象，它代表在文本编

辑器的文件窗口中所选的文本。一个宏通过用 `ActiveDocument.Selection` 属性来自定义当前文件中的所选文本。我们在第 13 章的 `ColumnarReplace` 宏中见过这个属性（请看列表 13-2）。表 C-3 和表 C-4 列出了 `TextSelection` 的属性和方法，它们可以被宏用来处理所选的文本，移动插入符号、滚动以及执行其他任务。在宏脚本中，每个属性和方法都必须用一个句号来与 `ActiveDocument.Selection` 属性关联。例如，下面的一个语句用 `Copy` 方法把所选的文本复制到剪贴板上：

```
ActiveDocument.Selection.Copy
```

表 C-3 `TextSelection` 属性

属 性	说 明
<code>BottomLine</code>	所选内容的最后一行的行号
<code>CurrentColumn</code>	插入符号目前所在的列号
<code>CurrentLine</code>	插入符号目前所在的行号

续表

Text 包含所选文本的字符串。如果所选的内容超过两行，该字符串除了最后一行外在每一行的末尾处放一个新行字符。因为 Text 属性是默认的，所以不必指定这个属性。因此下面的两个语句的作用是一样的：

```
str=ActiveDocument.Selection str=AvtiveDocument.Selection.Text
```

TopLine 所选内容第一行的行号

表 C-4 TextSelection 方法

方法	说明
Backspace	与按 Backspace 键的效果一样
Cancel	与按 Esc 键的效果一样
ChangeCase	改变所选内容的大小写，或者只把每个词的第一个字符变为大写

续表

CharLeft	把插入符号左移一定的距离
CharRight	把插入符号右移一定的距离
ClearBookmark	从当前行清除一个未命名的书签
ClearBookmarks	从文件中清除所有的未命名的书签
Copy	把所选文本复制到剪贴板上
Cut	把所选文本复制到剪贴板上，然后删除文件上的所选内容
Delete	从文件中删除所选内容
DeleteWhitespace	删除插入符号附近的所有空格和制表符。不必选择文本
DestructiveInsert	用新的文本替换所选文本

续表

EndOfLine	把插入符号移到当前行行末。参见 StartOfLine
EndOfDocument	把插入符号移到文件末尾。参见 StartOfDocument
FindText	在文件中查找自定义的文本；如果找到了，把插入符号放在所找到的文本的开始处
Go ToLine	把插入符号移到自定义行的开始处
Indent	在当前行添加一个缩进。这同把插入符号放在一行的开始处，然后按 Tab 键的效果相同。在使用 Indent 时要小心，因为这个函数删除所选文本
LineDown	把插入符号向下移一定的行数
LineUp	把插入符号向上移一定的行数
MoveTo	把插入符号移到指定的行和列

续表

NewLine	与按 Enter 键的效果一样
NextBookmark	把插入符号向前移到下一个已命名或未命名的书签。参见 PreviousBookmark
PageDown	与按 PgDn 键的效果一样
PageUp	与按 PgUp 键的效果一样
Paste	把当前剪贴板中的内容粘贴到文件的插入符号处。
PreviousBookmark	把插入符号移动到前一个未命名或已命名的书签处。参见 NextBookmark
ReplaceText	查找并替换文本。关于 ReplaceText 用法的示例，请参看第 13 章中所描述的 ColumnarReplace 宏
SelectAll	选择全部文件

续表

SelectLine	选择插入符号所在的行
SetBookmark	为插入符号所在的行设置一个未命名的书签
SmartFormat	根据当前的精确格式设置格式化所选文本
StartOfDocument	把插入符号移到文件的开始处。参见 EndOfDocument
StartOfLine	把插入符号移到当前行的开始处。参见 EndOfLine
Tabify	对选择内容前面加制表符。关于 Developer Studio 的 Tabify 命名的详细信息请看第 3 章
Unindent	把所选的全部行的缩进标尺取消。其效果同按 Shift+Tab 一样
Untabify	取消所选内容的制表符
WordLeft	把插入符号左移一定数目的词
WordRight	把插入符号右移一定数目的词

调试 VBScript 宏

通常，用户可通过在宏运行时显示其关键位置的变量的当前值，来有效地调试宏。可用 MsgBox 函数或者在第 1 章 Visual C++ OutPut 窗口的 Macro 选项卡中写入消息的 PrintToOutputWindow 方法（见第 1 章中的 The Environment）来显示调试字符串。MsgBox 函数中止宏，并要求用户输入，而 PrintToOutputWindow 方法并不中止宏。

下面的代码段产生图 C-1 所示的调试字符串：

```
x=3  
  
MsgBox("x=" & x)           ‘Display a message box  
  
PrintToOutputWindow("x=" & x) ‘Writes to Output window
```

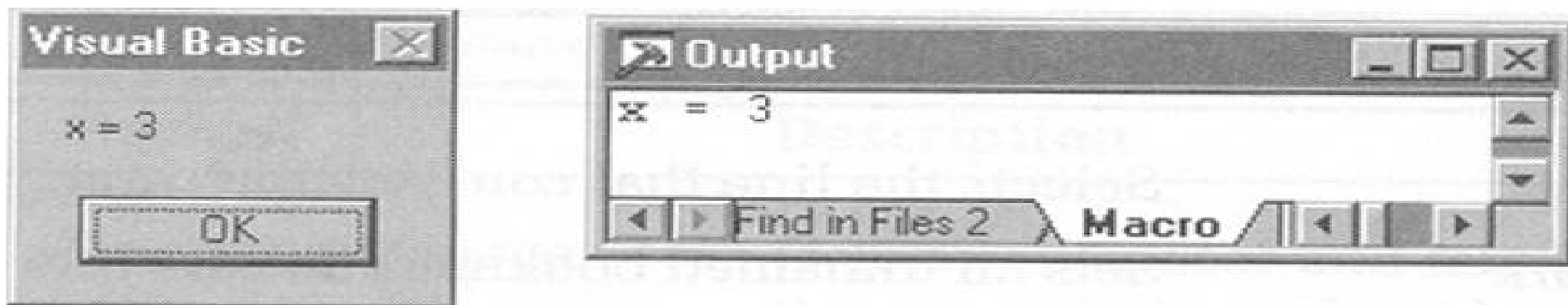


图 C-1 用 MsgBox 和 PrintToOutputWindow 显示调试字符串

当需要更强的调试能力时，有时可在 Visual Basic 中重建部分宏，它允许用户单步执行一个宏、设置断点、查看变量等等。Microsoft Script Debugger 中也可以这样做。尽管 Script Debugger 是为 Microsoft Internet Explorer 的 HTML 页中所嵌入的 VBScript 代码而设计的，它还是能够处理 Visual C++ 宏中的一些 VBScript 代码。

关于 Script Debugger 的详细消息，请看在线帮助中的 script debugging(脚本调试)主题。也可以从本附录开始处的 Microsoft 的 VBScript Web 网页下载一份。Script Debugger 软件包中包括文档说明。

库函数

本附录的其他部分将着重描述 Visual C++宏脚本中可用的 VBScript 库函数。表 C-5 把库函数分为几类，以便于用户决定哪一个函数适合于特定的编程要求。表 C-6 按字母顺序列出了这些函数，并提供了简单的描述和示例代码段。可以在表 C-5 中找到所需的函数，然后在表 C-6 中或者 Visual C++在线帮助中查看该函数的描述。这些库函数包含在 VBScript.dll 文件中，该文件通常在 Windows 的 System 或 System32 目录中。

根据惯例，宏脚本中的函数名字由大小写字符组成。但 VBScript 解释器不区分大小写，对函数名也完全不区分大小写。“C”前缀用于识别转换函数，如 CByte 和 CDate 函数。这些函数把值从一种类型强制转换成另一种类型，提供一种类似于 C/C++中的类型强制的机制。

表 C-5 VBScript 不同类型的库函数

数组，算术和三角函数					
Abs	Cos	Hex	Log	Round	Sqr
Array	Exp	Int	Oct	Sgn	Tan
Atn	Fix	LBound	Rnd	Sin	UBound
转换和变量类型函数					
CBool	CDate	CInt	CStr	IsEmpty	TypeName
CByte	CDBl	CLng	IsArray	IsNull	VarType
CCur	Chr	CSng	IsDate	IsNumeric	
日期和时间函数					
Date	DateValue		MonthName	TimeValue	

续 表

DateAdd	Day	Now	WeekDay
DateDiff	Hour	Second	WeekDayName
DatePart	Minute	Time	Year
DateSerial	Month	TimeSerial	

格 式 化 函 数

FormatCurrency	FormatNumber	FormatPercent	FormatDateTime

字 符 串 和 文 本 函 数

Asc	Join	Mid	ScriptEngine	StrComp
Filter	LCase	MsgBox	ScriptEngineBuildVersion	String
InputBox	Left	Replace	ScriptEngineMajorVersion	StrReverse

续表

InStr	Len	Right	Space	Trim
InstrRev	LTrim	RTrim	Split	UCase

表 C-6 VBScript 库函数

函 数	说 明
Abs	返回一个数的绝对值（无符号）。例如，Abs(-2)和 Abs(2)都返回值 2
Array	返回包含一个数组的 Variant 量。在下面的例子中，第一个语句创建一个变量 x。第二个语句把一个数组赋给变量 x，并初始化数组元素。后面的几个语句演示了所赋的值是如何放置在新的 x 数组中的： <pre>Dim x x= Array(10,20,30)</pre>

续表

a=x(0) ‘a=10

b=x(1) 'b=20

c=x(2) 'c=30

A S C

返回一个字符串的第一个字符的 ANSI 码。另一个类似的函数 AscB 返回一个字符串的第一个字节。相关的 AscW 函数返回一个字符串的第一个字符的字节和 Unicode 码，这样就避免了从 Unicode 到 ANSI 的转换

Atn

返回一个数的反正切值（以弧度为单位）。其结果的范围为从 $-\pi/2$ 到 $\pi/2$ 弧度。为了把度转化为弧度，在度数上乘以 $\pi/180$ 。Atn 是 Tan 的反三角函数，Tan 是以角度为参数，返回一个直角的两边的比值

C B o o l

返回一个表达式的 Boolean 值

CByte

返回一个表达式的字节值

续表

CCur

把一个表达式转换为货币子类型。CCur 函数提供了基于当前系统

上的国际设置的正确转换。使用这个函数能够确保宏为任何国家显示正确的货币值

CDate

把一个表达式转换为日期子类型：

```
str=mm&"-"&dd&"-"&yy
```

```
date=CDate(str)
```

CDBl

返回一个表达式的双精度值

续表

Chr	返回一个字符码相应的 ANSI 字符。从 0 到 127 的字符码同标准的 ASCII 码一样。例如，Chr(10)返回一个换行符。关于 ASCII 和 ANSI 字符码的讨论，请看附录 A，一个类似的函数 ChrB 用于字符串中的字节数据。ChrB 总是返回一个字节，而不是返回可能是一个或者两个字节的字符。ChrW 函数被提供给使用 Unicode 字符的 32 位平台
CInt	把一个表达式转换为整数子类型。CInt 不同于 Fix 和 Int 函数，这两个函数是截断，而不是对一个数值的小数部分取整。当小数部分正好为 0.5 时，CInt 函数总是把这个数取为最近的偶数。例如，0.5 被取为 0，1.5 被取为 2。
CLng	把一个表达式转换为长整数子类型。CLng 对一个数的小数部分取整。当小数部分正好为 0.5 时，CLng 函数总是把这个数取为最近的偶数
Cos	返回一个角度的余弦值：

续 表

	<code>cosx=Cos(x)</code>	'Where x is in radians
<code>CSng</code>	把一个表达式转换为单精度子类型	
<code>CStr</code>	把一个表达式转换为字符串子类型：	
	<code>x=111</code>	
	<code>str="The numeric value is "+CStr(x)</code>	
	<code>MsgBox(str)</code>	
	<code>str= "The Boolean value is "+CStr(CBool(x))</code>	
	<code>MsgBox(str)</code>	
<code>Date</code>	返回当前系统日期：	
	<code>MsgBox("The current date is "&Date)</code>	

续表

DateAdd

返回已经加入指定时间间隔的日期。DateAdd 函数不返回无效的日期，如 2 月 31 日。它计算闰年，承认每月的天数。它不承认 100 年 1 月 1 日之前或者 9999 年 12 月 31 日之后的日期。用下列特定

的字符串来指定时间间隔：

字符串	含义	字符串	含义
"d"	天	"q"	季度
"ww"	星期	"yyyy"	年
"m"	月	"h"	时
"w"	星期中的天	"m"	分钟
"y"	年中的天	"s"	秒

下面是 DateAdd 函数的几个例子：

续 表

```
str = DateAdd( d, -1, Date )
```

```
MsgBox(Yesterday was & str )
```

```
str = DateAdd( ww, -1, Date )
```

```
MsgBox( Last week was & str )
```

```
str = DateAdd( m, 1, Date)
```

```
MsgBox( Next month will be & str )
```

```
str = DateAdd( q, 1, Date )
```

```
MsgBox( Three months from now will be & str )
```

DateDiff

返回两个日期之间的时间间隔。请看 DateAdd 函数中用于指示时间间隔的字符串：

续 表

```
str=DateDiff("d",Date,#12-25#)
```

```
MsgBox("Only"&str&"days till Christmas")
```

DatePart

返回一给定日期的特定部分：

```
str = DatePart ("ww", Date )
```

```
MsgBox("This week is number" & str & "of the year" )
```

```
day1 = DatePart( "d", Date)
```

```
day2 = day1
```

```
Do While day2 >9
```

```
    day2 = day2 -10
```

```
Loop
```

```
suffix = "th"
```


续表

```
If (day1 = 1) Then suffix = "st"
```

```
If (day1 = 2) Then suffix = "nd"
```

```
If (day1 = 3) Then suffix = "rd"
```

```
MsgBox("Today is the" & day1 & suffix)
```

DateSerial

返回一个包含指定年、月和日的字符串。DateSerial 允许宏计算经过一个连续时间段后的绝对日期。例如，下面的 DateSerial 调用返回了从当前日期 100 天以后的日期：

续 表

```
iYear = DatePart( "yyyy", Date)
```

```
iMonth = DatePart( "m", Date)
```

```
iDay = DatePart( "d", Date)
```

```
str = DateSerial( iYear, iMonth, iDay + 100 )
```

```
MsgBox( "100 days from now will be" & str)
```

当 `year` 参数包含从 0 到 99 之间的值时，`DateSerial` 认为这个值表示从 1900~1999 之间的某一年。对于其他的 `year` 参数，要用完整的 4 位数表示，如 2010。

DateValue

返回包含一个格式化日期的字符串。`DateValue` 函数在根据时区设置辨认各种格式的日期时很有用。例如，在 `United States` 设置下，下面的语句都返回字符串 "12/31/99"：

续表

`DateValue( "December 31, 1999" )`

`DateValue( "Dec 31, 1999" )`

`DateValue( "12-31-99" )`

`DateValue( "12 31 1999" )`

Day

读入一个日期字符串，从中分离出该月中的天，即从 1~31

Exp

返回基于 e（自然对数的基）的指数。常量 e 的值约为 2.718282

Filter

从一个字符串数组中分离出包含指定子字符串的字符串，或者，分离出不包含指定子字符串的字符串。Filter 返回的数组只包含符合匹配规则的字符串。在下面的例子中，x 数组包含字符串 "string2"，因为它是 str 数组中唯一包含子字符串 "2" 的字符串：

`str(0) = "string0"`

`str(1) = "string1"`

续表

```
str(2) = "string2"
```

```
str(3) = "string3"
```

```
MsgBox( x(0) )
```

Fix

返回一个浮点数的整数部分。Int 和 Fix 函数都是截断小数部分而返回整数部分。以变量 x 为例，如果 x 为正，这两个函数的结果相同；如果 x 为负，Int 返回小于或者等于 x 的第一个负整数，而 Fix 返回大于或者等于 x 的第一个负整数。例如，如果 x 为 -5.6，那么 Int 返回 -6 而 Fix 返回 -5

FormatCurrency

用系统 Control Panel（控制面板）中的区域设置，以正确的货币格式返回一个表达式

FormatDateTime

以符合当前区域设置的日期或者时间格式返回一个表达式

续表

FormatNumber

返回字符串形式的数值。这个字符串的格式根据系统的区域设置，例如，超过 1000 的数值在美国是用逗号分开，而在欧洲是用句号

FormatPercent

返回以 % 字符为后缀的百分数（被 100 乘）格式的表达式

Hex

返回一个代表 16 进制数的字符串。宏能够通过`在数的前面加上前缀 &H`来表示一个 16 进制数。参考 `Oct`

Hour

返回一个 0~23 之间的数，以表示给定时间的小时数。例如，在晚上 6 点到 7 点之间，下面的代码返回 18：

```
hr=Hour(Time)
```

```
MsgBox("The current hour is "&hr)
```

参考函数 `Minute` 和 `Second`

续表

InputBox	显示一个带有指定提示消息的对话框，并返回用户输入的字符串。关于 InputBox 的示例，请看第 13 章中的 ColumnarReplace 宏
InStr	返回一个字符串中另一个字符串第一次出现的位置
InstrRev	同 InStr 函数类似，InstrRev 返回一个字符串中在另一个字符串的位置，不同的是它是从字符串的尾部开始
Int	参考函数 Fix 的说明
IsArray	返回指示给定变量是否为数组的一个 Boolean 值
IsDate	返回指示给定表达式是否能够转换为日期 的一个 Boolean 值
IsEmpty	返回指示给定变量是否已初始化的一个 Boolean 值。当变量未被初始化，或者被显式设为 Empty 时，IsEmpty 返回 True.；否则，这个函数返回 False。参考 IsNull

续表

IsNull	返回指示给定变量是否包含有效数据的一个 Boolean 值。在变量为 Null（不包含有效数据）时，IsNull 返回 True；否则，该函数返回 False。VBScript 承认 Null 和 Empty 之间的区别。Empty 意味着变量还没有被初始化。一个零长度的字符串，有时被看作是一个空字符串，但不是一个 Null 变量
IsNumeric	返回指示给定表达式是否为数字的一个 Boolean 值
Join	通过连接数组中字符串而形成一个新的字符串
LBound	为数组的一个维返回最小可用下标。默认值为 0
LCase	把一个字符串的所有字符都转换为小写字符。参考 Ucase 函数
Left	返回由一个字符串最左边的字符所组成的字符串。宏能够通过调用 Len 函数来确定一个字符串的长度。参考 Right 函数
Len	返回一个字符串中的字符数

续表

Log 返回一个数的自然对数（基数为 e）。数 x 基数为 n 的对数是 x 的自然对数和 n 的自然对数的比值：

$\log_n x = \text{Log}(x) / \text{Log}(n)$

LTrim 消除一个字符串的前导空格。参考 **RTrim** 和 **Trim**

Mid 从一个字符串分离出一个子字符串

Minute 返回从 0~59 之间的一个数以指示给定时间的分钟数。例如，在 6:47:53，下面的代码返回 47：

`min=Minute(Time)`

`MsgBox("The current minute is " & min)`

参考 **Hour** 和 **Second** 函数

续表

Month

返回从 1~12 之间的一个数，以指示给定年的月份：

```
Dim mnth(11)

mnth(0) = "January"

mnth(1) = "February"
```

```
mnth(11) = "December"
```

```
m = Month( Date )
```

```
MsgBox(The current month is "& mnth(m-1) )
```

MonthName

返回一个字符串，它包含由 1~12 之间的一个数所指定的月份。
通过使用 MonthName 函数，上面的例子可以像下面这样改写：

续 表

```
m= Month(Date)
```

```
MsgBox("The current month is"&MonthName(m))
```

MsgBox

显示一个带有 OK,Cancel,Abort,Retry,Ignore,Yes,No 按钮的标准 Windows 消息框。MsgBox 返回下列值以指示用户单击了哪一个按钮关闭消息框：

常 量	值	按 钮
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry

续 表

常 量	值	按 钮
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

MsgBox 也可以在消息框中显示一个 Help 按钮，当单击时，就会显示来自特定帮助文件的上下文相关帮助。用户也可以按 F1 键来查看与上下文相对应的帮助主题

Now

返回包含当前日期和时间的字符串，当前日期和时间的格式符合当

前区域设置。例如，对于 United States 设置，返回字符串的格式为 "12/31/99 3:33:57 PM"。参考 Time 和 Date 函数

续表

Oct	返回一个表示八进制数的字符串。在宏中，一个八进制数可用前缀 &O 表示。参考 Hex 函数
Replace	以一个字符串作为输入值，返回该字符串的一个新值，其中的子字符串已被另外的值所代替。参考表 C-4 所列的 ReplaceText 函数
Right	返回一个字符串最右边的字符所组成的字符串。宏通过调用 Len 函数来确定一个字符串的长度。参考 Left 函数
Rnd	返回一个小于 1 但大于或者等于 0 的随机数。为了产生一个上下界之间的一个随机整数，要采用下列方式： $\text{iRange} = \text{iUpperBound} - \text{iLowerBound} + 1$ $\text{Int}(\text{iRange} * \text{Rnd} + \text{iLowerBound})$ 使用 Randomize 语句产生根据系统时钟的 VBScript 随机数生成器

续表

	Randomize
	iRandom=Rnd
Round	把一个浮点数四舍五入到指定的精度
Rtrim	取消一个字符串的尾部的空格。参考 LTrim 和 Trim 函数
ScriptEngine	返回一个标识使用中的脚本语言的字符串。对于 Visual Basic Scripting Editon, 该函数返回“VBScript”
ScriptEngineBuildVersion	返回使用中的脚本引擎的建立版本号
ScriptEngineMajorVersion	返回使用中的脚本引擎的主版本号
Second	返回从 0~59 之间的一个整数, 以表示给定时间的秒成分。例如, 在 6:47:53, 下面的代码返回 53: sec=Second(Time)

续表

```
MsgBox("The current second is"&sec)
```

参考 Hour 和 Minute 函数

Sgn

确定给定数的符号。Sgn 返回一个值为 -1, 0 或者 1 的整数, 以指示这个数是小于零, 等于零, 还是大于零:

```
Dim str(2)
```

```
str(0)="less than zero"
```

```
str(1)="equal to zero"
```

```
str(2)="greater than zero"
```

```
i =InputBox("Enter a number")
```

```
MsgBox("The number is " & str(Sgn(i)+1))
```

续表

Sin

返回一个角度的正弦值

`sinx=Sin(x)` 'Where x is in radians

参考 Cos 和 Tan 函数

Space

返回一个包含指定数目空格的字符串

Split

根据一个字符串创建一个字符串数组,这个字符串的子字符串由指定的字符分隔。下面的例子用加号把一个字符串分成三个单独的字符串：

`x=Split("string1+string2+string3","+")`

`MsgBox(x(0))` 'Display "string1"

`MsgBox(x(1))` 'Display "string2"

`MsgBox(x(2))` 'Display "string3"

续表

Sqr

返回一个数的平方根

StrComp

比较两个字符串，并返回一个值，指示这两个字符串是否相同。

StrComp 的用法和语法同 C 动态库中的 strcmp 函数相似：

```
If StrComp(string1,string2)=0) Then
```

```
    MsgBox("Strings are equal")
```

```
End If
```

StrComp 返回下列值中的一个：

返回值	含义
-1	string1 小于 string2
0	string1 等于 string2

续表

	1	string1 大于 string2
StrReverse	颠倒给定字符串的字符顺序	
String	创建由单个字符组成的指定长度的字符串	
	<code>str=String(10,"a")</code>	<code>'str="aaaaaaaaaa"</code>
Tan	返回一个角度的正切值	
	<code>tanx=Tan(x)</code>	<code>'Where x is in radians</code>
	参考 Sin 和 Cos 函数	
Time	按照当前的区域设置格式返回包含当前系统时间的字符串。例如，在美国，所返回的字符串为 "3:33:57 PM"。当设置为欧洲时，同样的时间表示为 "15:33:57"。参考 Now 函数	

续表

TimeSerial

按照当前的区域设置格式返回一个包含给定小时、分钟、秒的字符串。当设置为美国时，下面的语句把 "6:47:53 PM" 赋给变量 x：

```
x=TimeSerial(18,47,53)
```

TimeValue

返回一个包含一个有格式的时间的字符串。像 DateValue 函数一样，TimeValue 承认各种各样取决于区域设置的格式。例如，当设置为美国时，下面的语句返回字符串 "6:47:53 PM"：

```
TimeValue("6:47:53PM")
```

```
TimeValue("18:47:53")
```

```
TimeValue("6:47:53 pm")
```

Trim

删除一个字符串首尾的空格。参考 RTrim 和 LTrim 函数

TypeName

以一个变量作为参数，返回指示变更子类型的字符串：

续表

返回字符串	变量子类型
"Byte"	字节值
"Integer"	整数值
"Long"	长整数值
"Single"	单精度浮点数
"Double"	双精度浮点数
"Currency"	货币字符串
"Decimal"	十进制数
"Date"	日期或时间字符串
"String"	字符串

续表

"Boolean"

布尔值

"Empty"

未初始化

"Null"

无有效数据

下面的代码演示了 `TypeName` 函数：

```
x=3
```

续表

<code>MsgBox(TypeName(x))</code>	‘Displays "integer"

`y="string"`

<code>MsgBox(TypeName(y))</code>	‘Displays "String"

`z=#12-31-99#`

<code>MsgBox(TypeName(z))</code>	‘Displays "Date"

<code>MsgBox(TypeName(w))</code>	‘Displays "Empty"

UBound

返回一个数组指定维的最大可用下标值。例如：

`Dim A(100,3,4)`

续表

<code>x=UBound(A,1)</code>	<code>'x=99</code>

<code>y=UBound(A,2)</code>	<code>'y=2</code>

<code>z=UBound(A,3)</code>	<code>'z=3</code>

UCase

把字符串中的所有字符都转换为大写。参考 Lcase 函数

VarType

以一个变量作为参数，返回下表中的一个整数，以指示变量的子类型。注意这个函数与 TypeName 函数的相似性

续 表

常 量	值	变 量 子 类 型
vbEmpty	0	未 初 始 化
vbNull	1	无 有 效 数 据
vbInteger	2	整 数
vbLong	3	长 整 数
vbSingle	4	单 精 度 浮 点 数
vbDouble	5	双 精 度 浮 点 数
vbCurrency	6	货 币 字 符 串
vbDate	7	日 期 或 时 间 字 符 串
vbString	8	字 符 串
vbVariant	11	Variant 类 型 的 数 组

续表

	VbBoolean	11	布尔值
	vbByte	17	字节值
	vbArray	8192	数组。VarType 返回数组值 加上数组中的变量子类型的 值的和
Weekday	返回从 1 到 7 之间的一个整数，表示一星期中的星期几，其中 1 表示星期天。参考 WeekDayName 函数的示例代码		
WeekDayName	返回一个字符串以指示一星期中的某一天：		
	<pre>d=WeekDay(Date)</pre>		
	<pre>MsgBox("Today is"&WeekDayName(d))</pre>		

续 表

Year

从一个给定的日期中以整数值返回其中的年数：

```
MsgBox("The year is "&Year(Date))
```

