Visual C++

简明教程

计 算 机 教 程

青苹果电子图书系列

Visual C++ 简明教程

邓　力　　编著
张　晋　　审校

丛 书 前 言

计算机软件技术日新月异，编程软件种类繁多且各具特色。众多的计算机爱好者都希望能成为电脑编程的行家里手，如何才能向“卡乃基殿堂”迈出正确的第一步？

为满足各级 Windows 程序开发人员、大专院校相关专业的师生和业余编程爱好者学习和使用各种流行编程软件的需求，编者在搜集了不同层次读者意见的基础上，经过仔细探讨，精心策划了本套丛书，目的在于引导读者学习二十一世纪真正的编程工具！

本丛书名为“流行编程软件简明教程丛书”，主要包括：

《C++ Builder 简明教程》

《Delphi 简明教程》

《Visual C++ 简明教程》

《Visual Basic 简明教程》

《Visual FoxPro 简明教程》

《Quick BASIC 简明教程》

《C 语言简明教程》

本套丛书针对当前最流行的开发工具，通过浅显易懂、活泼生动的语言，采用逐步引导的方式将读者带入电脑编程的殿堂，使读者轻松掌握编程的基础知识，为进一步深造打下坚实基础。生动友好的例子帮助读者了解、掌握编程技巧，同时还大大激发了读者的兴趣。

本套丛书既不是面面俱到的“功能大全”，也不是单一查询函数的“用户手册”，而是独具特色的操作和编程指导书。丛书注重对软件的主要功能和新增特性进行介绍，结合实例和项目，讲解软件的主要功能、主要原理。

我们希望，本套丛书对于读者能“抛砖引玉、触类旁通”，指引读者踏上通往编程高手之路。

总的来说，本套丛书有以下几个最优越的特点：

- ✧ “精”——精选软件，精选作者，精选图书，是值得用户收藏的精品
- ✧ “准”——紧跟软件版本更新，出版时间准，图书出版市场定位准
- ✧ “简”——教程风格简单朴素，要让读者学习起来感觉简单实用
- ✧ “明”——丛书内容十分明晰，让读者读完之后能够真正明白

我们相信：

如果你是一位初学编程的读者，本套丛书将带领你入门！

如果你是一位对编程有些了解的业余爱好者，本套丛书为你指明前进的方向！

如果你是一位软件编程专业人员或计算机行家里手，本套丛书将为你的程序锦上添花！

编 者

前　　言

Visual C++系列产品是开发 Windows 应用程序的有力工具，而 Visual C++ 6.0 则是 Microsoft 公司目前发布的最新版本。与 Visaul C++的以前各版本相比较，Visual C++ 6.0 有重大的改进，功能更加强大，使用起来更加方便。

本书简明扼要的介绍了 Visual C++ 6.0 的使用，通过丰富的例子程序向读者介绍如何使用 MFC 开发 Windows 应用程序。全书结构安排如下：

第一章和第二章是本书的基础部分，主要介绍了 Visual C++ 6.0 的一些基本知识和如何快速地建立一个 MFC 应用程序，并介绍了 MFC 应用程序的基本运行过程。

第三章详细介绍了 MFC 应用程序框架的消息映射机制。

第四章介绍了有关对话框和控件的知识。这包括如何使用有模式对话框，如何使用各种常用的控件，最后稍稍介绍了通用对话框和无模式对话框的一些基本知识。

第五章集中介绍了有关 Windows 屏幕绘图的话题。这包括使用设备环境和 GDI 对象工作等内容，最后引导用户建立了一个屏幕保护例子程序。

考虑到读者已经有了一定的基础，第六章着重谈谈有关 Windows 设备打印方面的话题。

第七章重点介绍了 MFC 应用程序的文档/视图结构。本章的内容比较多，包括如何建立单文档类型、多文档类型和多窗口的应用程序等内容。

第八章是对第四章的必要补充，着重介绍了更多改进用户程序界面的技术，教会读者学习使用更多的控件。

第九章介绍了有关数据库的知识，这包括使用 ODBC、DAO、ADO 和 OLE DB 等技术，读者应该注意区分这些技术各自的特色。

第十章简要介绍了有关 Internet 的话题，主要是介绍如何在客户端使用 WinInet 类访问 Internet 服务器，最后还引导用户建立了一个类似于 IE5.0 的 Web 浏览器。

本书的特点在于简明扼要，原理讲解结合必要的实例，这些例子程序都是笔者根据自己学习和使用 Visual C++中的体会精心挑选的，基本上都是针对程序员在开发过程中需要最迫切、使用频率最高的内容特意定制的，可以说比较贴切地符合了初级和中级程序员的需求。另外，本书中所有例子程序的代码都经过了严格的调试和测试，读者只要跟着书中给出的步骤往下做，最终一定能够圆满地完成程序。

由于时间作者水平有限，书中肯定存在一些不足之处，恳请读者指正。

编　者

内 容 提 要

Visual C++是 Windows 平台上使用最广泛的可视化开发环境，功能非常强大。本书共十章，简明扼要地介绍了如何使用 Visual C++开发 Windows 应用程序，主要内容包括：MFC 的消息映射机制，有关对话框和控件的知识，如何使用 Windows 图形界面绘图，了解文档/视图结构，有关数据库开发的知识和有关 Internet 的应用。

全书内容简洁明了，使用简明的语言并辅之以实例，介绍使用 Visual C++进行程序设计的各种方法和常用技巧。本书中的程序都是笔者在实践中总结提炼出来的，具有很强的实用性。

本书适合各个级别的用户使用，主要作为 Visual C++课程的培训教材，也可作为专业程序人员的参考资料，对广大学习编程的爱好者也是一本很好的学习指南。

目　　录

第一章　Visual C++ 6.0 概述

“真正的程序员用 Visual C++!”

Visual C++是当今最被广泛使用的可视化编程环境，为我们提供了一种方便、快捷的 Windows 应用程序开发工具。它使用了 Microsoft Windows 图形用户界面的许多先进特性和设计思想，采用了弹性可重复利用的完整的面向对象程序语言（Object-Oriented Language）、当今世界上最快的编辑器、最为领先的数据库技术。

Visual C++ 6.0 是 Visual C++系列的最新版本，是为 Windows 98 或 Windows 2000 等 32 位操作系统开发应用程序用的编程工具，功能强大，界面友好，操作方便。本章为全书的第一章，主要是对 Visual C++ 6.0 做一些简要的说明，介绍 Visual C++ 6.0 的基本特性。读者在阅读完本章后，将对 Visual C++ 6.0 有个大概的了解。

在本书以下的叙述中，为了简单起见，将 Visual C++ 6.0 简称为 VC6.0。

1.1　VC 6.0 的新特性

与 Visual C++系列以前的版本相比而言，VC6.0 提供了许多新的特性，大致可以分为以下几方面。

1.1.1　编辑器方面的新特性

VC6.0 在集成开发环境的编辑器做了一些改进，以方便用户快速准确地编辑代码和资源，这些新特性主要包括：

- 自动完成语句功能

在用户编辑代码时，编辑器根据光标当前位置判断作用的类或对象，在一个下拉列表中显示相应的内容，如类的成员、函数原型、标识符定义等等。自动完成语句功能减轻了用户在输入长的类名或成员对象名时的繁琐工作，方便了用户的使用。

- 快速宏录制

用户可以将集成开发环境中特定的连续操作定义为宏，在需要再次使用类似操作的时候只需调用已录制好的宏即可。

- 支持 IE5 新控件的资源编辑器

新版本的资源编辑器支持 IE5 提供的四种新控件，在资源编辑器中可以方便地将这些新控件添加到工具栏或者对话框中。

1.1.2　编译器、连接器和调试器方面的改进

VC6.0 在集成开发环境的编译器、连接器和调试器方面也做了大的改进，其目的就是使得用户的应用程序运行起来更快、更稳定，调试起来更方便。

- 编译器方面的改进

包括新添加的“_assume”关键字、增强对内联函数的控制、新增并更新了警告、加强了运行时刻的错误检测等内容。

- 连接器方面的改进

包括延迟加载外部支持、增添了新的连接选项和修正了外部函数的接口以减小文件尺寸等内容。

- 调试器方面的改进

包括改善了反汇编输出、改善了指针对象的显示、支持进程中的远程过程调用等内容。

1.1.3 MFC 类库的增强

VC6.0 在 MFC 类库方面做了许多改进，这包括一系列新的或改进过的类、对新技术的支持等等。这些改进大大增强了 MFC 应用程序的功能，主要包括以下方面的内容：

- Active 文档容器

与 OLE 容器相区别，Active 文档容器能够一次激活整个文档，文档占据了容器应用程序的整个框架窗口。通过在 AppWizard 中选择合适的选项，很容易就能够使得用户开发的应用程序成为 Active 文档容器。

- 动态 HTML 控件和 CHtmlView 类

动态 HTML 控件就是指 WebBrowser 控件，提供了浏览 HTML 页面的能力。CHtmlView 类封装了该控件，并能够在应用程序的视图中显示 HTML 页面。

- 支持 IE5 提供的新控件

IE5 提供了四种类型的新控件，MFC 为这些新类型的控件提供了类的支持，这包括 CComboBoxEx 类、CDateTimeCtrl 类、CIPAddressCtrl 类和 CMonthCalCtrl 等类，分别支持对应类型的控件。

- 新提供的类和改进的类

MFC 提供了许多新的类，除了上述四个类之外，还包括 COleDBRecordView 类、COleDocObjectItem 类、CPropertySheetEx 类、CPropertyPageEx 类、CReBar 类和 CReBarCtrl 类等；改进过的类就更多了，典型的有 CString 类、CMenu 类、CToolBar 类和 CListCtrl 类等等。

- 新的全局函数

MFC 还提供了新的全局函数 AfxCheckError()和 AfxDumpStack()，这都是为用户调试程序服务的。用户可以在程序的任何位置调用这些全局函数。

- 性能调整

静态链接的 MFC 库的尺寸减小了。

1.1.4 实用工具程序

VC6.0 中提供了一些新的实用工具程序，同时也对原有的一部分工具程序做了改进。

- 组件管理器

组件管理器帮助用户管理和共享软件工具或者可重用组件，其目的就是为了更好地支持代码或资源的重用与管理、协调开发工具的使用和管理软件小组的共同开发活动等等。

- HTML 帮助

HTML 帮助是一种新格式的帮助系统，用户可以在 Web 页中集成基于 HTML 格式的上下文相关帮助。HTML 帮助系统比传统 Windows 帮助系统提供了更多的功能，VC6.0 附带的 MSDN 就是 HTML 帮助系统的典范。

- 安装程序开发工具

为交流而开发的软件必须提供一整套的安装程序，InstallShield 就是用户书写安装程序的好帮手。随 VC6.0 提供的 InstallShield 6.0 是该工具程序的最新版本，也是为 VC6.0 度身定制的。

- 控件测试容器

新版本的控件测试容器更加稳定，并且支持 OCX96 新特性。控件测试容器还允许用户使用 VBScript 语言书写测试脚本程序。

1.1.5 向导方面的改进

在使用 Visual C++系列开发应用程序的过程中，用户在很多情况下都需要向导的帮助，VC6.0 在向导方面做了许多改进。

- 改进的 AppWizard

改进的 AppWizard 为用户定制 MFC 应用程序提供了更多的选择。新的 AppWizard 支持不带文档/视图结构的应用程序，可以为应用程序提供 Active 文档容器或者服务器功能，支持 ReBar 控件和网络浏览器风格的应用程序等。

- 支持 OLE DB

在 AppWizard 中选择数据库支持时，除了选择 DAO 和 ODBC 外，现在还可以选择 OLE DB，这可通过使

用 MFC 提供的 OLE DB 模板实现。

■ 为帮助文件定制编译规则

在建立应用程序的帮助文件时，AppWizard 用定制编译规则的方法取代了原来使用的 MakeHelp.bat，这使得帮助文件的编译只有在需要时才会进行。

■ 删除成员函数

在 VC6.0 中，用户可以通过 WizardBar 或者 ClassView 删除类的不再需要的成员函数，这虽然只是一个很小的功能，但是很实用。

■ 新的项目类型

定制的 AppWizard 现在可以支持静态链接库、控制台应用程序和 ActiveX 控件等多种新的项目类型。

■ 为非 MFC 程序提供的向导

VC6.0 为许多非 MFC 程序项目也提供了向导支持，这包括为 Win32 应用程序、动态链接库、控制台程序提供的向导等等。

提示：

本节中只是简单地介绍了 VC6.0 的一些新的特性，有关新特性的更多信息，请参阅 VC6.0 联机手册中的“What's new”部分的内容。

1.2 集成环境及基本操作

Microsoft Develop Studio 集成开发环境功能强大，但同时其窗口也比较复杂，包含有多个子窗口和工具栏，而且子窗口又包含了若干标签和选项卡，有必要对集成开发环境的基本使用做一些介绍。

集成开发环境的基本外观如图 1-1 所示。

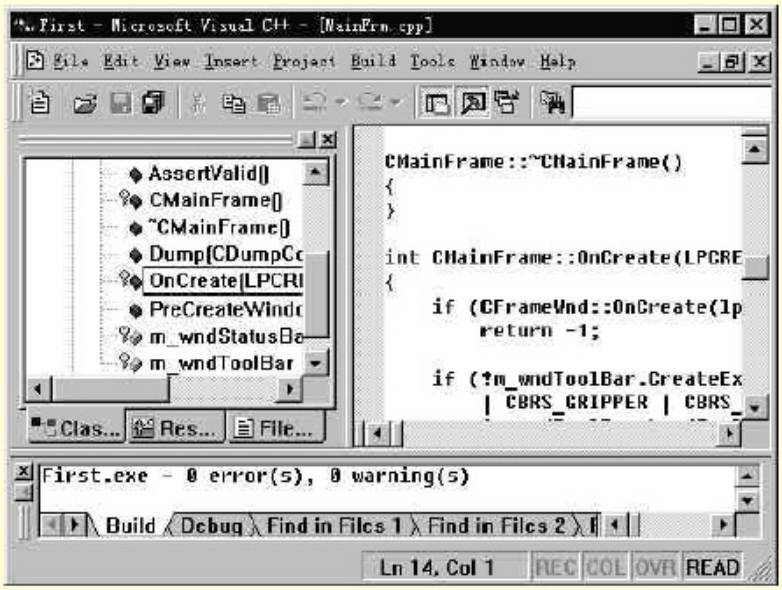

图 1-1 集成开发环境的基本外观

1.2.1 正文窗口及其操作

各种程序文件、资源文件和文档文件都可以在正文窗口中打开并相应地进行编辑。用户可以同时打开多个正文窗口（这就是所谓的多文档界面），每个窗口都可以通过其系统菜单或右上角的按钮进行最大化、最小化、复原、关闭、移动和改变尺寸等操作，并且可以通过在 Window 菜单中选定特定窗口或通过鼠标激活到想切换的窗口。

同其他 Windows 应用程序类似，Window 菜单下的 New window 命令将打开当前文档的另一窗口；Split 命

令将当前窗口分成四个小窗口；Close 和 Close all 命令将关闭正文窗口。Next 命令将切换到下一个正文窗口；Previous 命令将切换到上一个正文窗口。Cascade 命令将层叠排列各正文窗口；Tile horizontally 和 Tile vertically 命令将水平平铺或垂直平铺各正文窗口。

也可以通过选择 Window 菜单下的 Windows 命令弹出对话框来操纵正文窗口，如图 1-2 所示。

图 1-2 Windows 对话框

在 Windows 对话框中，左边的列表框列出了当前所有打开的正文窗口。选中一个正文窗口，单击 Activate 按钮将使得选中的正文窗口成为当前窗口；单击 OK 按钮将退出本对话框；单击 Save 按钮将储存选中的正文窗口的内容；单击 Close Window 将关闭该正文窗口。如果选中了多个正文窗口，则 Tile Horizontally 和 Tile Vertically 按钮变为有效，单击可以水平平铺或垂直平铺选中的窗口。

这个对话框看起来与菜单命令功能重复，在当前打开的正文窗口少的时候确实如此。然而如果打开的窗口比较多的话，比如说超过 10 个，这时 Windows 对话框就显得比较方便了，不过同时打开超过 10 个窗口的机会似乎并不多。

1.2.2 项目工作台窗口及其操作

项目工作台窗口在正文窗口的左边。在项目工作台窗口中，一般有三个标签：Classview（类视图）、Resourceview（资源视图）和 Fileview（文件视图）。单击某个标签就会显示相应的视图。

在 Classview 中，显示了当前工作空间（Workspace）的项目，以及项目中的类和类的成员变量、成员函数。就象在资源管理器中一样，Classview 中的对象可以一层一层地展开，直至最底层的对象。如果在 Classview 中双击类名，将在正文窗口中打开定义了该类的头文件；如果双击变量名，将在正文窗口中打开定义该变量的头文件，并高亮显示在正文窗口的中间；如果双击函数名，则将在正文窗口中打开该函数的声明，并高亮显示。利用这一特点，可以快速地浏览类的组成并跳到合适的地方。另外，用鼠标右键单击某个对象，将弹出一个快捷菜单，显示了该对象可以执行的命令，这也是很方便的。

Resourceview 显示了当前项目所用到的资源状况，如菜单、工具栏、对话框等等。在 Resourceview 中，同样可以展开，双击某个对象将在正文窗口打开该对象。

Fileview 几乎就是一个缩微版的资源管理器，不过它只显示当前项目所用到的文件。利用它可以快速打开指定文件。

注意：

✍ 与 Visual C++ 5.0 不同，这里并没有 InfoViewer，Microsoft 把所有的帮助信息都集成到了另外一个可单独运行的程序——MSDN 中。如果用户在安装 Visual C++ 6.0 的时候同时安装了 MSDN，那么用户应该到 MSDN 中查询帮助信息。

1.2.3 输出窗口及其操作

输出窗口在集成开发环境的底部，包括了 Build（编译）、Debug（调试）、Find in files 1（文件查询结果 1）、Find in files 2（文件查询结果 2）、Results（结果）和 SQL Debugging（SQL 调试）等标签。Build 标签中显示了编译和连接的进度和错误信息。Debug 标签中显示了调试信息。输出窗口的操作方式类似于项目工作台窗口。

1.3 菜单功能介绍

集成开发环境的菜单提供各种各样的命令，在开发应用程序的过程中将大量地使用菜单命令。因此，先简单地介绍一下各个菜单的基本功能是很有必要的。下面将逐一介绍集成开发环境的菜单。

1.3.1 File 菜单

File 菜单中的命令主要执行新建、打开或关闭文件或工作空间，另外还有打印和打印设置，最近使用过的文件和工作空间列表等。如图 1-3 所示。

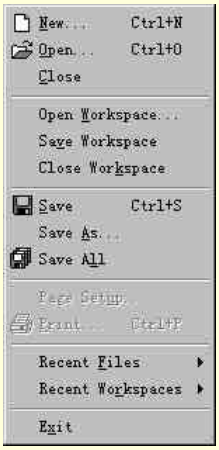

图 1-3 File 菜单

File 菜单命令的快捷键和功能的简要介绍如表 1-1 所示。

表 1-1 File 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
New	Ctrl+N	新建文件、项目、工作空间或其他文档
Open	Ctrl+O	打开一个已存在的文件

续 表

菜单命令	快捷键	功能说明
Close	——	关闭当前文件
Open Workspace	——	打开已存在的工作空间
Save Workspace	——	储存当前工作空间
Close Workspace	——	关闭当前工作空间
Save	Ctrl+S	储存当前文件
Save As	——	用新的文件名储存当前文件
Save All	——	储存所有文件
Page Setup	——	设置文件的打印页面
Print	Ctrl+P	打印文件的全部或一部分
Recent Files	——	最近使用过的文件列表
Recent Workspace	——	最近使用过的工作空间列表
Exit	——	退出集成开发环境

1.3.2 Edit 菜单

Edit 菜单中的命令主要有撤消、恢复上一步的操作，剪切、复制、粘贴和删除选定区域，查找与替换字符串，定位与标签，断点设置等普通命令。

Microsoft Vusaul Studio 6.0 版还增添了一些新的功能，如最下方的四个命令。用户如果使用过 Visual C++ 的以前的版本，是否在输入一个名称较长或拥有好几个参数的函数时遇到过困难，经常要在帮助文档和编辑窗口间进行切换。现在好了，在用户进行输入的时候，Visual Studio 可以弹出一个关于当前对象的说明，如对象的各个成员、函数原型、参数类型等等，甚至可以帮助你完成要输入的字符。看来在 Visual Studio 6.0 中，Tooltips 的功能是大大地增强了。Edit 菜单如图 1-4 所示：

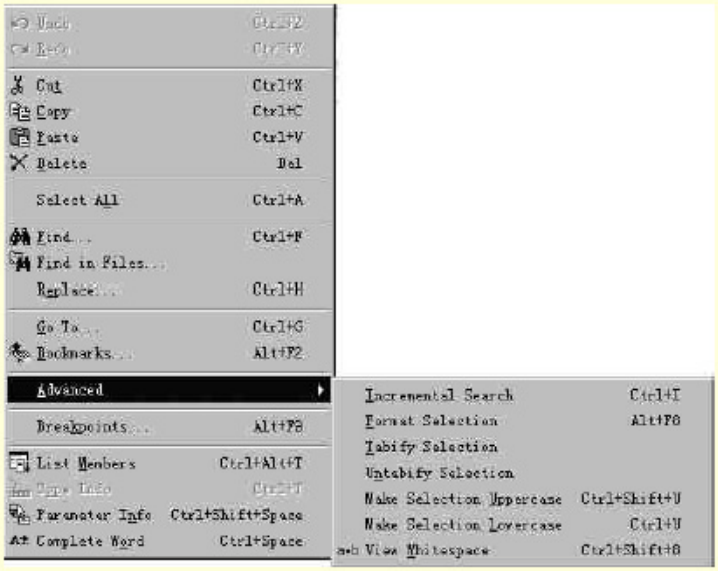

图 1-4 Edit 菜单

各命令的快捷键和功能说明如表 1-2 所示。

表 1-2 Edit 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
Undo	Ctrl+Z	撤消上一次操作
Redo	Ctrl+Y	恢复上一次撤消的操作
Cut	Ctrl+X	将选中的内容剪切至剪贴板
Copy	Ctrl+C	将选中的内容复制至剪贴板
Paste	Ctrl+V	从剪贴板中粘贴内容至光标处
Delete	Del	删除选中的内容
Select All	Ctrl+A	选中所有的内容
Find	Ctrl+F	在当前打开的正文窗口中查找字符串
Find in files	——	在当前项目的所有具有指定扩展名的文件中查找字符串
Replace	Ctrl+H	查找指定字符串并用新的字符串代替
Go To	Ctrl+G	跳转到指定的行、标签或地址等
Bookmarks	Alt+F2	将当前位置制作标签
Advanced\Incremental Search	Ctrl+I	直接对键盘输入的字符串进行查找
Advanced\Format Selection	Alt+F8	按格式进行选择
Advanced\Tabify Selection	——	对选中内容按制表位置定位
Advanced\Untabify Selection	——	取消定位
Advanced\Make Selection Uppercase	Ctrl+Shift+U	将选中的所有字符转换为大写
Advanced\Make Selection Lowercase	Ctrl+U	将选中的所有字符转换为小写
Advanced\View Whitespace	Ctrl+Shift+8	显示空格、制表位等空白字符
Breakpoints	Alt+F9	设置或清除程序的断点
List Members	Ctrl+Alt+T	列出光标左边的类或结构的有效的成员变量和成员函数

Type Info	Ctrl+T	显示标识符的完整的声明
Parameter Info	Ctrl+Shift+Space	显示光标所在处函数的完整声明，包括参数
Complete Word	Ctrl+Space	由编辑器自动完成待输入的单词

1.3.3 View 菜单

View 菜单如图 1-5 所示。

图 1-5 View 菜单

View 菜单中的命令主要功能是显示/隐藏指定的子窗口，满屏编辑模式切换等等。各命令的快捷键和功能说明如表 1-3 所示。

表 1-3 View 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
ClassWizard	Ctrl+W	编辑应用程序中的类
Resource Symbols	——	浏览和编辑资源文件中的符号
Resource Includes	——	编辑资源文件及其符号预定义文件
Full Screen	——	在窗口的正常模式和全屏模式之间切换
Workspace	Alt+0	显示项目工作台窗口
Output	Alt+2	显示输出窗口
Debug Windows\Watch	Alt+3	显示调试观察窗口
Debug Windows\Call Stack	Alt+7	显示调试堆栈调用窗口
Debug Windows\Memory	Alt+6	显示调试内存窗口
Debug Windows\Variables	Alt+4	显示调试变量窗口
Debug Windows\Registers	Alt+5	显示调试寄存器窗口
Refresh	——	刷新显示的窗口
Properties	Alt+Enter	显示当前对象的属性窗口

1.3.4 Insert 菜单

Insert 菜单如图 1-6 所示。

图 1-6 Insert 菜单

Insert 菜单中的命令主要用于在当前项目中插入资源、文件或组件。各命令的快捷键和功能说明如表 1-4 所示。

表 1-4 Insert 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
New Class	——	插入一个新的类
New Form	——	插入新的框架

Resource	Ctrl+R	插入任一类型的新资源
Resource Copy	——	为已存在的资源创建副本
Files as Text	——	以文本方式插入文件
New ATL Object	——	插入新的 ATL 对象

1.3.5 Project 菜单

Project 菜单中的如图 1-7 所示。

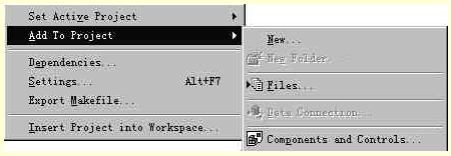

图 1-7 Project 菜单

Project 菜单中的命令主要用于对当前工作空间的项目进行操作，如设置活动项目，向项目中添加内容，改变项目设置等。各命令的快捷键和功能说明如表 1-5 所示。

表 1-5 Project 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
Set Active Project	——	设置当前活动项目
Add to Project\New	——	向当前项目添加新的预定义可支持的文件
Add to Project\New Folder	——	向当前项目添加新的文件夹
Add to Project\Files	——	向当前项目添加以存在的文件
Add to Project\Data Connection	——	向当前项目添加数据连接
Add to Project\Componets and Controls	——	向当前项目添加组件与控件
Dependencies	——	设置当前项目的依赖条件
Settings	Alt+F7	改变当前项目的设置选项
Export Makefile	——	输出当前项目的工程文件
Insert Project int Workspace	——	将项目添加至工作空间中

1.3.6 Build 菜单

Build 菜单如图 1-8 所示。Build 菜单中的命令包括编译代码、连接目标文件、调试和运行应用程序，以及设置编译、调试应用程序的选项。另外，如果在调试状态下运行程序，Build 菜单将被 Debug 菜单代替。

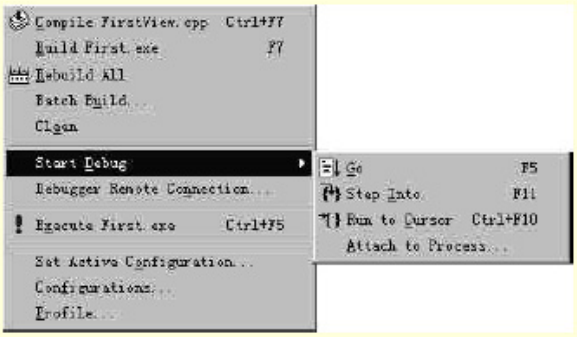

图 1-8 Build 菜单

Build 菜单的各命令的快捷键和功能说明如表 1-6 所示。

表 1-6 Build 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
Compile	Ctrl+F7	编译源文件为目标代码
Build	F7	编译源文件并连接目标代码为可执行程序
ReBuild All	——	重新编连所有文件
Batch Build	——	成批编连，一次编连多个项目文件

Clean	——	
Start Debug\Go	F5	从当前位置或程序开头执行至断点处或程序尾
Start Debug\Step Into	F11	执行下一条语句并跟踪进入遇到的函数
Start Debug\Run to Cursor	Ctrl+F10	执行至光标所在语句
Start Debug\Attach to Process	——	
Debugger Remote Connection	——	调试器远程连接以实现远程调试
Execute	Ctrl+F5	执行已建立的应用程序
Set Active Configuration	——	选择项目的当前编连配置
Configurations	——	设置项目的编连配置
Profile	——	

1.3.7 Tools 菜单

Tools 菜单如图 1-9 所示。

图 1-9 Tools 菜单

Tools 菜单中的命令主要用于选择或定制集成开发环境中的一些使用开发工具，并提供了定制菜单、工具栏等界面外观的命令（Customize）和宏操作命令。其基本命令的快捷键和功能说明如表 1-7 所示。

表 1-7 Tools 菜单

菜单命令	快捷键	功能说明
Source Browser	Alt+F12	浏览对选定对象或前后文的查询
Close Source Browser File	——	关闭浏览信息的文件
Customize	——	定制工具栏、菜单等界面元素
Options	——	修改集成开发环境的各项环境设置
Macro	——	显示宏操作对话框
Record Quick Macro	Ctrl+Shift+R	记录快捷宏
Play Quick Macro	Ctrl+Shift+P	使用已记录好的快捷宏

另外，在 Tools 菜单的中部有几个命令未加介绍。这些将一些实用程序和集成开发环境联系起来，选择这些命令将在集成开发环境中使用这些实用程序。随用户使用的不同版本或在安装时的不同选择，这里可能会有所不同。

1.3.8 Window 菜单

Window 菜单如图 1-10 所示。命令主要用于窗口的操作，这在前面已经介绍过，此处仅列表简单说明，如表 1-8 所示。

表 1-8 Window 菜单

菜单命令	快捷键	功能说明
New Window	——	打开一个新的工作窗口
Split	——	切分工作窗口
Docking View	——	
Close	——	关闭当前工作窗口

Close All	——	关闭所有打开的工作窗口
Next	——	激活下一个工作窗口
Previous	——	激活上一个工作窗口
Cascade	——	层叠排列打开的工作窗口
Tile Horizontally	——	水平平铺打开的工作窗口
Tile Vertically	——	垂直平铺打开的工作窗口
Windows	——	弹出 windows 对话框

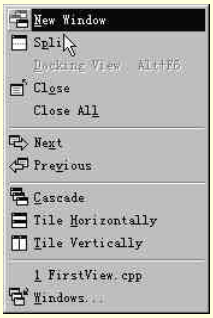

图 1-10 Window 菜单

1.3.9 Help 菜单

Help 菜单如图 1-11 所示。

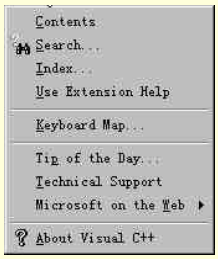

图 1-11 Help 菜单

Help 菜单是获取帮助信息最主要和最有效的途径。表 1-9 列出了各命令的功能。

表 1-9 Help 菜单

菜单命令	快捷键	功能说明
Contents	——	以目录方式列出所有帮助信息
Search	——	用查询方式获取帮助信息
Index	——	以索引方式显示所有帮助信息
Use Extension Help	——	使用扩展帮助信息
Keyboard Map	——	显示所有的键盘指令（包括快捷键）
Tip of the Day	——	显示每日提示信息
Technical Support	——	技术支持
Microsoft on the Web	——	连接至 Internet 上的 Microsoft 公司的有关专题
About Visual C++	——	显示版权信息

值得一提的是，选择 Help 菜单的前三项将激活 MSDN（Microsoft Develop Network）应用程序，所有的操作将转移至 MSDN 应用程序中进行，而不是象以前的版本那样也集成在开发环境中。下面是 MSDN 应用程序的外观，如图 1-12 所示。在 MSDN 中的操作比较直观，使用上与以前版本或其他应用程序的帮助是类似的。使用时有困难的用户可以参考 MSDN 的帮助。

图 1-12 MSDN 应用程序窗口

1.3.10 Debug 菜单

在调试用户的应用程序的时候，Debug 菜单将代替 Build 菜单，如图 1-13 所示。

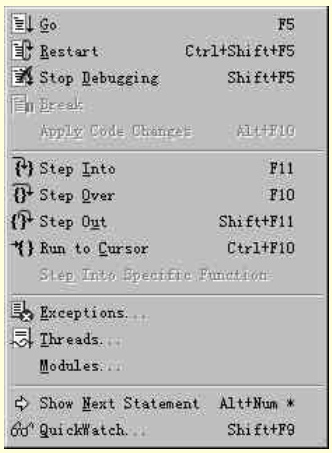

图 1-13 Debug 菜单

Debug 菜单中的命令主要用于操纵应用程序调试的进程，各命令的快捷键和功能说明列在表 1-10 中。

表 1-10 Debug 菜单命令的快捷键和功能说明

菜单命令	快捷键	功能说明
Go	F5	从当前位置起执行程序至断点处或程序尾
Restart	Ctrl+Shift+F5	重新从头开始运行程序
Stop Debugging	Shift+F5	强制停止程序的调试
Break	——	强制暂停程序的调试
Apply Code Changes	Alt+F10	按修改过的代码运行
Step Into	F11	执行下一条语句并跟踪进入遇到的函数
Step Over	F10	执行下一条语句但不进入遇到的函数
Step Out	Shift+F11	执行后面的语句直至推出当前函数
Run to Cursor	Ctrl+F10	执行到光标所在语句或程序尾
Step Into Specific Function	——	跟踪进入指定的函数
Exceptions	——	指定需要跟踪的例外
Threads	——	显示当前正在执行的所有线程
Modules	——	显示当前正在执行的所有模块
Show Next Statement	Alt+Num *	显示下一步将执行的语句
QuickWatch	Shift+F9	快速显示指定变量的值

以上较详细地介绍了 Visual Studio 6.0 集成开发环境的菜单，如果用户觉得还是不太熟悉也无须担忧，在使用 Visual C++ 6.0 进行应用程序开发的时候会逐渐熟悉的。

1.4 工具栏的使用

虽然只使用菜单命令就可以完成所有的工作，菜单命令的快捷键又可以提高菜单命令的操作效率，但由于选择菜单命令须多次鼠标点击动作，记忆多个快捷键又比较困难，所以用起来略有不便。而工具栏是一种图形化的界面，既直观又快捷，熟练使用工具栏将使得工作效率大大提高。

在 Visual C++ 6.0 的集成开发环境中，提供了丰富的工具栏供用户选用。第一次启动 Visual C++ 6.0 时，屏幕上显示缺省的标准工具栏，包括的按钮有：NewText File（新建文本文件）、Open（打开文件）、Save（存储文件）等常用的按钮。除了缺省工具栏外，还显示了与当前打开的编辑器有关的工具栏。

另外，工具栏上的某些按钮可能在一定的条件下才被激活，可以使用。因此，用户看到的工具栏上的按钮有些可能呈现为灰色不可使用状态。

和其他 Windows 应用程序一样，用鼠标点击工具栏上的按钮将执行相应的命令。一般情况下，工具栏上的每一个按钮均对应于菜单中的一条命令，点击按钮和选择对应命令执行的是同一操作。只在极少的情况下，工具栏上的按钮没有对应的菜单命令，但这种情况似乎不太规范，也很少见到。

如果将鼠标指针在某个按钮上停留片刻，Visual C++的集成开发环境将在光标附近弹出一个提示（Tooltips），简要说明该按钮的作用。这在我们记不清按钮作用的时候是很有用的。

利用鼠标可以很容易地调整工具栏和菜单栏的位置。将鼠标指针定位在工具栏的最左端（如图 1-14 所示）按下鼠标左键，不要松开，将工具栏拖曳至合适位置再松开左键，则工具栏将泊定在当前位置。

如果在主框架窗口的中部松开鼠标左键，工具栏将显示为一个小的工具窗口（如图 1-15 所示）。

利用鼠标的拖曳，用户可以将工具栏停泊在任何位置，甚至也可以对菜单栏进行拖曳，试一试，停在主框架窗口中部的菜单栏是什么样子？

另外，在工具栏或菜单栏的任意位置双击鼠标也将会使相应的工具栏或菜单栏作为小窗口停泊在主框架窗口中。

图 1-14 定位鼠标指针在工具栏的左端

图 1-15 工具窗口

1.5 文本编辑器的使用

一个完整的应用程序包括源代码文件和资源文件。在 Vusual C++ 6.0 的集成开发环境中，源代码文件的编辑在文本编辑器中完成，资源文件的编辑则在资源编辑器中完成。文本编辑器用来维护、编辑和打印源代码文件，资源编辑器用来创建和编辑对话框、菜单和工具栏等资源。本小节中将介绍文本编辑器的使用，其他编辑器将在后续小节中进行介绍。

文本编辑器的作用是编辑源文件。其主要功能有：

- 源文件的管理，包括创建、打开、保存和打印等功能。
- 在源文件中的定位，包括设置标签、跳转等功能。
- 对源文件的编辑，可以使用菜单命令、快捷键和鼠标拖曳功能进行编辑。
- 查找与替换字符串。

下面将对文本编辑器的功能逐一进行介绍。

1.5.1 文件的管理

在 File 菜单中选择 New 命令，将弹出 New 对话框；在 New 对话框中选择 Files 标签，就可以创建新文件了。这时的对话框如图 1-16 所示。

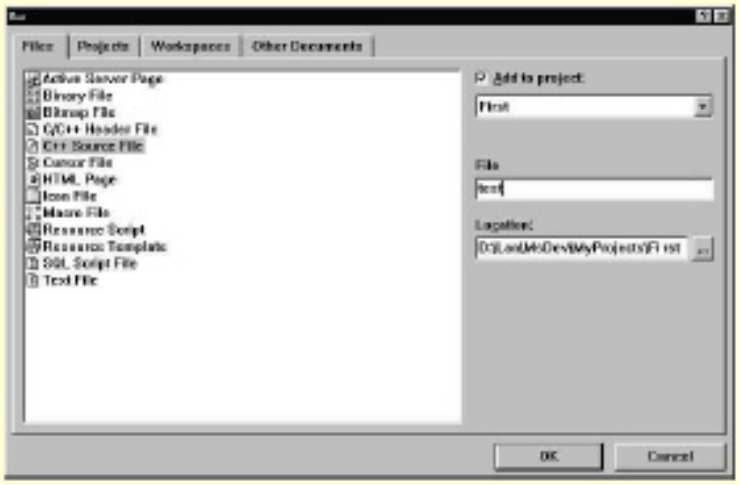

图 1-16 创建一个新文件

在 Files 标签下列出了所有可用的文件类型，包括：

- Active Server Page：活动服务器页面，选中该项后将创建一个 HTML 格式的文件。
- Binary File：二进制文件，该项将创建一个空的原始数据文件，数据用二进制或十六进制数表示。
- Bitmap File：位图文件，该项将创建新的位图文件，并打开位图资源编辑器。
- C/C++ Header File：C/C++头文件。一般在头文件中进行类、变量或函数的定义。
- C++ Source File：C++源文件。一般在源文件中进行函数的具体声明。
- Cursor File：光标文件，该项将创建新的光标文件，同时打开光标资源编辑器。
- HTML Page：HTML 页面，该项将创建新的 WEB 页面文件，并已提供了基本的框架，用户在适当的位置添加代码即可。
- Icon File：光标文件，该项将创建新的图标文件，同时打开图标资源编辑器。
- Macro File：宏文件，该项将创建新的宏文件。
- Resource Script：资源描述文件，该项将创建新的资源描述文件，可将新的资源添加至此描述文件中。
- Resource Template：资源模板，该项将创建新的资源模板文件，可将某些资源添加至资源模板文件保存起来，在以后的开发中即可重用这些资源。
- SQL Script File：SQL 脚本文件，该项将创建新的 SQL 脚本文件。SQL 脚本文件可以执行一些数据库功能，而无须编程。
- Text File：文本文件。

如果要将新文件添加到项目中，选中 Add to project 检查框，并选择相应的项目名称。在 File 框中输入新建文件的文件名，在 Location 框中输入文件的存储路径。单击 OK 按钮即可创建新文件。

如果用户需要打开已存在的文件，选择 File 菜单下的 Open 命令。如果用户需要保存正在编辑的文件，选择 Save 命令即可。如果需要用另外的文件名保存，选择 Save as 命令。如果需要打印文件，选择 Print 命令。具体操作与其他 Windows 应用程序中类似，此处不在赘述。

实际上，除了 Bitmap File、Cursor File、Icon File、Resource Script 和 Resource Template 等资源类的文件使用相应特定的编辑器外，其他类型文件的编辑均使用的是文本编辑器，最多也只是对不同的类型有一些小小的差异，基本使用方法是相同的。而用户使用文本编辑器大部分时间是编辑头文件或源文件，下面为叙述的方便起见，将以源文件的编辑为例进行介绍。

1.5.2 在文件中定位

在编辑一个较长的文件时，仅凭记忆和手工查找要迅速跳转到某一位置是不方便的。利用 Edit 菜单下的

Bookmarks 命令在适当位置加上一些标记，再进行跳转就方便的多了。而 Go to 命令则提供了更普通的跳转方法。

使用 Bookmarks 命令的具体操作为：

（1）将光标移到需要做标记的位置上。

（2）选择 Edit 菜单下的 Bookmarks 命令，将弹出 Bookmark 对话框。

（3）在 Name 文本框中输入标记名称。

（4）单击 Add 按钮，将本次标记添加到标记列表中。

（5）单击 Close 按钮，关闭对话框。

在添加了标记之后，如果要跳转到某个标记处，先打开 Bookmark 对话框，从列表中选择该标记，然后单击 Go to 按钮即可。如果要删除某个标记，先选中该标记，单击 Delete 按钮即可。

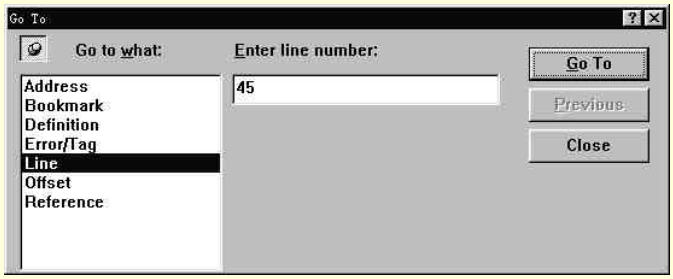

图 1-17 Go to 对话框

Go to 命令具有更强的功能，可以按指定的标记、行、符号定义等多种类型跳转到相应位置。选择 Edit 菜单下的 Go to 命令将弹出 Go to 对话框，如图 1-17 所示。

Go to 对话框左边的列表框中列出了跳转类型：

- Address：按地址跳转。本方式在调试时才有效，可以跳转到任何有效的内存地址处进行编辑。当然，这需要用户对直接编辑内存中的内容有相当的把握。
- Bookmark：按标记跳转。将跳转到指定标志处。
- Definition：跳转到指定标识符的定义处。
- Error/Tag：跳转到下一个错误位置。
- Line：跳转到指定行。
- Offset：按偏移量跳转。本方式同样在调试时才有效。
- Reference：跳转到指定标识符的参考信息处。本方式只在生成了浏览信息时有效。

以上介绍了利用 Bookmark 和 Go to 命令进行文件中的定位。如果用户正在编辑 C++头文件或源文件，还可以利用 WizardBar 工具栏进行定位。WizardBar 工具栏如图 1-18 所示。

图 1-18 WizardBar 工具栏

WizardBar 工具栏从左到右有三个下拉列表框，分别为类选择、成员选择和函数选择列表。如果需要跳转到某个函数，只须在类列表中选择函数所属的类，在成员选择列表中选择列出所有类成员（All class members），在函数列表中选择相应的函数，则该函数的完整的声明将显示在文本窗口中并高亮显示。

1.5.3 对文件进行编辑

文本编辑器提供了四种编辑方式：使用菜单命令、使用工具栏、使用快捷键和使用鼠标拖曳功能。这四种编辑方式是完全等效的，也无须进行方式切换，一切均取决于用户的习惯。如果用户使用过其他的文本编辑应用程序，如 Windows 附件中的写字板，以及 Microsoft 公司的排版软件 Word 等，那么这里的操作是相同的。下面对四种方式略作介绍。

- 使用菜单命令：实现文本编辑的各命令均在 Edit 菜单中。Cut 命令将所选定的内容剪切至剪贴板，Copy 命令将所选定的内容复制到剪贴板中，Paste 命令将从剪贴板中复制内容至当前位置，Undo 命令将撤

消上一步的操作，Redo 命令将恢复 Undo 命令撤消的操作。

- 使用工具栏：在标准工具栏中，对应于 Edit 菜单中的各命令都有相应的按钮。单击按钮与选择相应的命令作用相同。
- 使用快捷键：与工具栏类似，上述各命令均有对应的快捷键。对于使用集成开发环境比较熟悉的用户来说，使用快捷键可能是最快捷的编辑方式。
- 使用鼠标拖曳功能：使用鼠标的拖曳功能进行编辑也是很方便的，主要用于移动和复制文件内容。具体说来，先选中待移动的内容，用鼠标拖至目的位置在松开鼠标左键，就可将选中的内容移动到新的位置。如果在拖曳的过程中保持按下 Ctrl 键，则为复制选定内容。

1.5.4 查找与替换

查找与替换也是很重要的文件操作，Visual C++ 6.0 的集成开发环境对此也提供了支持，在 Edit 菜单下可以找到 Find 命令、Find in Files 命令和 Replace 命令。以下一一加以说明。

（1）选择 Find 命令将弹出 Find 对话框，如图 1-19 所示。

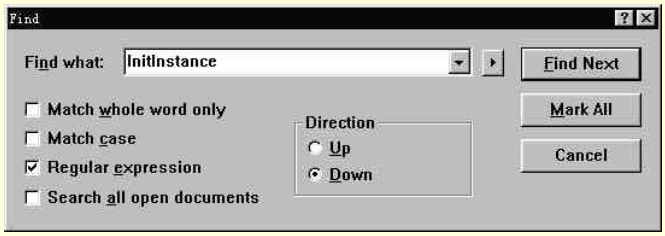

图 1-19 Find 对话框

在 Find what 文本框中输入带查找的字符串或正规表达式，或者单击右侧的下拉箭头从以前查找过的字符串和正规表达式列表中进行选择，单击 Find Next 按钮即开始寻找匹配的字符串。单击 Mark All 按钮将对所有找到的字符串进行标记，单击 Cancel 按钮放弃查找。

在 Find 对话框中还有一些选项，下面作一介绍：

- Match whole word only：如果选中了本选项，则将输入的字符串作为整体进行查找。即使某个字符串与输入的字符串在字符上一致，但含有多余的空格、制表符或其他标点符号，也视为不相匹配。
- Match case：如果选中本选项，则只有与输入的字符串大小写一致的字符串才认为相匹配。清楚本选项，将忽略大小写。
- Regular expression：如果选中本选项，在字符串查找中，可以使用正规表达式。稍后将介绍如何使用正规表达式进行查找。
- Search all open documents：如果选中本选项，将在所有已打开的文件中进行查找，否则只查找当前使用的文件。
- Direction（Up/Down）：指定查找的方向。Up 表示从当前光标所在位置起向文件头方向进行查找；Down 表示从当前光标位置起向文件尾方向进行查找。

注意到在 Find what 文本框的右边还有一个向右箭头的扩展按钮，这是用来输入正规表达式的。使用过 MS-DOS 的用户常常会利用“*”和“？”来进行文件列表，这里“*”和“？”称为通配符。在 Visual C++ 6.0 的集成开发环境中，为了增强查找功能，也引入了类似的通配符。所谓正规表达式，就是指采用通配符来匹配文件中的文件而进行查找。正规表达式的写法列于表 1-11 中。

用户可以直接直接在 Find what 文本框中输入正规表达式，也可以通过单击扩展按钮选择正规表达式的描述，文本编辑器会帮助用户输入正规表达式。

表 1-11 正规表达式

语 法	说 明
.	匹配任意字符。如 ma.可以匹配 ma、map 等匹配。
[]	匹配括号内字符集中的任一字符。如 m[abc]p 可匹配 map、mbp、mcp 等。
[^]	匹配除了括号内字符集之外的任一字符。如 m[^abc]p 可匹配 mdp、mep、mfp 等
^	从一行的行首开始进行匹配。

$	在一行的行尾进行匹配。
\~	匹配除\~符号后字符外的任一字符。
\!	与\!前后的字符相匹配。
*	*符号前的字符可以重复出现，也可不出现。
+	+符号前的字符可以重新出现，但至少出现一次。
\{\}	组合一个次级表达式。
\:a	任意单个字母字符或数字字符。等价于[a-z A-Z 0-9]。
\:c	任意单个字母字符。等价于[a-z A-Z]。
\:d	任意单个数字字符。等价于[0-9]。
\:h	任意十六进制数字字符。等价于[0-9 a-f A-F]。
\:n	任意无符号数。
\:z	任意整数。
\:I	任意 C/C++标识符。
\:w	任意单词或字母字符串。
\:q	任意被引用的字符串。

（2）在 Edit 菜单中选择 Find in Files 命令将弹出 Find In Files 对话框，如图 1-20 所示。

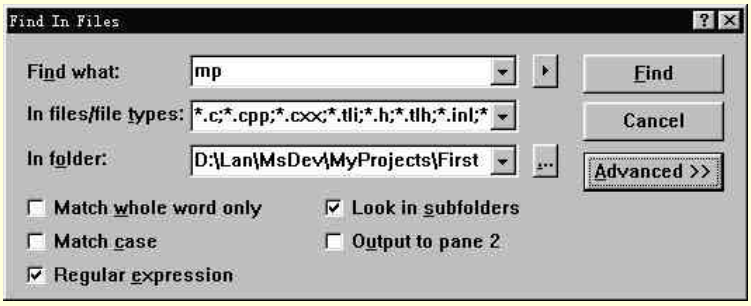

图 1-20 In Files 对话框

在 Find In Files 对话框中与在 Find 对话框中的操作是类似的。在 Find In Files 文本框中输入待查找的字符串或正规表达式。在 In files/fils types 列表框中选择要进行查找的文件类型。在 In folder 框中输入进行查找的文件夹，也可以通过单击右边的扩展按钮进行浏览选择。有两个选项是 Find 对话框所没有的，下面作一说明：

- Look in subfolders：选中本选项，将对选定的文件夹及其子文件夹中的文件进行查找，否则将忽略字文件夹中的文件。
- Output to pane 2：选中本选项，将查找的结果显示在输出窗口的 Find in Files 2 标签中，否则显示在 Find in Files 1 中。

（3）在 Edit 菜单中选择 Replace 命令将弹出 Replace 对话框，如图 1-21 示。

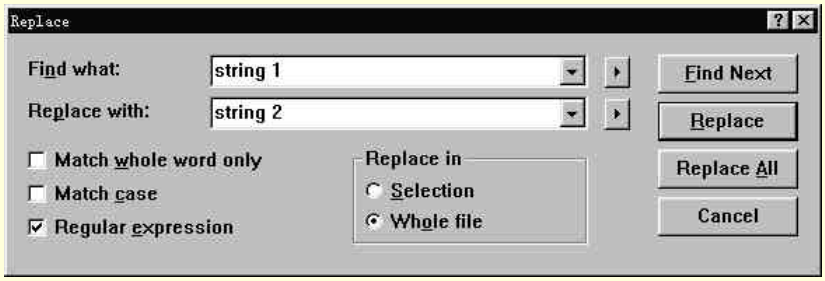

图 1-21 Replace 对话框

可以看到，Replace 对话框与 Find 对话框也是类似的。因此这里只略作介绍。

在 Find what 框中输入待查找的字符串或正规表达式，在 Replace with 框中输入替换用的字符串或正规表达式。单击 Replace 按钮将替换当前找到字符串，单击 Replace All 按钮将替换所有的字符串。

如果在打开 Replace 对话框之前选定了一个区域，则可以选择在选中区域（Selection）中进行替换，或是在整个文件（Whole file）中进行替换操作。

1.6 资源与标识符

资源是为 Windows 应用程序所用的一系列的预定义的数据，应用程序通过标识符对资源进行操作。

1.6.1 资源与资源编辑器

Windows 应用程序一般都包含有众多的图形元素，例如光标、位图、对话框等，在 Windows 环境下每一个这样的元素都是作为一种可装入应用程序的资源来存放。所谓资源，就是指用户可从中获得某种信息或进行某种操作的界面元素。资源本身仅仅是一种视觉上的表现，只显示信息或接受操作，但如何显示信息和接受操作之后如何动作不是资源所定义的，而是由应用程序本身来决定的。也就是说，如果没有给资源指定任何动作，那么在用户对资源进行操作将不会有任何事情发生。

在 Visual C++应用程序中，资源与源代码是分离的。这样处理资源与源代码的关系是有道理的。一方面，多个应用程序可以引用同一资源定义；另一方面，可以在不影响源代码的情况下编辑资源，或同时开发资源与源代码，缩短应用程序的开发时间。

Windows 环境下的资源主要有以下几类：

- 加速键（Accelerator）：一系列按键组合，一般与菜单命令相连，作为选择菜单命令的快速方法，被应用程序用来引发一个动作。
- 工具条（Toolbar）：包含多个按钮的组合，也被用来作为快速选择菜单命令的方法。
- 光标（Cursor）：32*32 像素的位图，指示鼠标的当前位置。
- 对话框（Dialog）：包含多种控件的窗口，与用户完成交互功能。
- 图标（Icon）：代表最小化窗口的位图。
- 菜单（Menu）：以可视的方式提供了对应用程序功能的选择。
- 字符串列表（String Table）：包含一系列的格式化文本。
- 版本信息（Version）：定义应用程序的版本，包含一系列格式化文本。

这些所有类型的资源都可以由 Visual C++提供的资源编辑器进行可视的编辑。相应于不同类型的资源，Visual C++提供了不同种类的编辑器，如对话框编辑器、菜单编辑器、工具条编辑器等等，这些编辑器的具体的使用方法将在介绍有关内容的同时加以介绍。

选择 Insert 菜单的 Resource 命令，就可以向应用程序添加新的资源，Insert Resource 对话框如图 1-22 所示：

图 1-22 Insert Resource 对话框

在 Resource Type 中选择合适的类型，单击 New 按钮就可以插入新的资源了。

1.6.2 标识符

在 Windows 应用程序中，通过标识符来引用资源或用户自定义的其他对象。实际上，标识符是映射了特定整数值的一个字符串，我们将该整数值称为标识符的编号，应用程序真正使用的编号，字符串只是便于程序员区分和使用标识符。

Visual C++提供了 Resource Symbols 对话框来管理项目的标识符。选择 View 菜单下的 Resource Symbols 命令就可以打开 Resource Symbols 对话框，如图 1-23 所示：

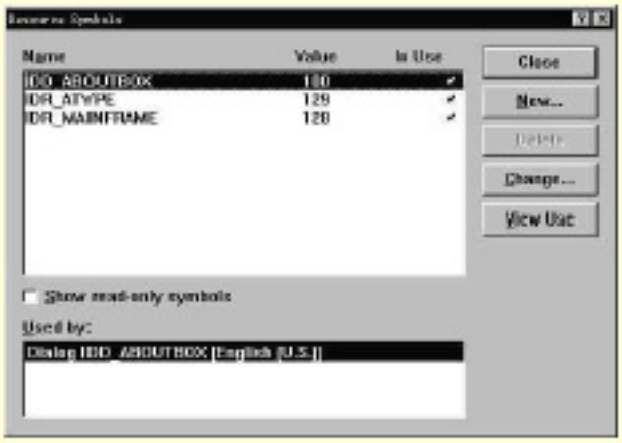

图 1-23 Resource Symbols 对话框

该对话框显示了本项目所包含的所有标识符，注意 In Use 列，如果某标识符对应的位置没有核对标记，则说明该标识符虽然被定义了，但没有任何资源和它相联系。下方的 Used By 框中列出了与选中标识符相联系的资源。单击 View Use 按钮可以查看与该标识符相联系的资源。

利用 Resource Symbols 对话框，可以方便地新建标识符，改变未使用的标识符的名称及编号，或者删除未使用的标识符。

建立新的标识符，只需单击 New 按钮，在弹出的 New Symbol 对话框中填写相应的名称与编号即可，如果不指定编号，系统将提供一个缺省的编号。

1.7 本章小结

本章是全书的开始，介绍了一些有关 VC6.0 的基本知识，如 VC6.0 的版本和新特性等等。Visual C++ 6.0 使用的是友好的操作界面——Develop Studio 集成开发环境，本章还主要介绍集成开发环境的外观、菜单条、工具条等的基本操作。

在开始下一章的学习之前，编者想就以下几个方面作额外说明。

说明：

1. 既然您已经决定要使用 VC6.0 开发程序，那么就请抛除一切旧的编程观念，包括使用 Visual Basic、C++ Builder、Delphi 等的编程思路，因为使用 Visual C++远远比它们要复杂，当然工作量也更大些。
2. 在使用 C/C++编程时，所有的工作都需要用户来完成，但是 Visual C++（或者说 MFC）扭转了这一局面，但是还有相当的工作需要用户自己动手，所以用户应该至少是一个熟练的 C 语言用户。
3. MFC 是以类的形式组织和提供代码的，在 C++语言中就有类的概念，所以用户应该对此有一些基本的了解，至少对 C++语言（类）有一点概念。

从下一章开始，我们就将跨入 MFC 的大门了，准备正式进入 Visual C++编程世界。

第二章 第一个 VC 应用程序

Visual C++提供了一系列的向导（Wizard），与 MFC 基础类库和 ATL 模板库配合使用，为用户生成应用程序的起始框架，或为用户的应用程序提供额外的功能。在这一系列的向导中，使用最多的应该是应用程序向导（AppWizard），该向导一步一步地引导用户进行应用程序的定制，最后为用户生成一整套源代码文件和资源文件，编译通过后就是一个具备了基本功能的起始应用程序。

2.1 快速建立 MFC 应用

许多 C 语言或者 C++语言的教材都将“ Hello，World !”程序作为入门的例子程序，本书同样也选用这个程序来作为读者学习 VC6.0 的第一个例子。通过这个简单而又有效的小程序，读者将了解到如何使用 VC6.0 的 AppWizard 快速建立一个可以实际运行的 MFC 应用程序。

对一般用户而言，使用 VC6.0 开发的应用程序大部分都是 Windows 应用程序。读者完全可以只使用基本的 C 语言开发出完美的 Windows 应用程序来，但是这可能需要读者付出大量的精力。实际上，使用 VC6.0 进行开发，更多的是使用其提供的 MFC 基础类库（Microsoft Foundation Class），在 MFC 的基础上完成自己的工作。

MFC 是微软公司提供的一套基于 C++语言的类库，随 Visual C++系列的发布不断更新和升级。MFC 是专为设计 Windows 应用程序而建立的“ 应用程序框架 ”，提供了大量的经过严格调试的代码，方便用户管理菜单、对话框、窗口等常见的 Windows 对象；实现基本的输入、输出功能；管理数据对象等等。

另外，MFC 还提供了大量的类和代码，简化用户对一些难点问题的处理，如 ActiveX 技术、数据库管理和 Internet 技术等。用户需要做的工作就是在 MFC 的“ 应用程序框架 ”上添加适当的代码，以完成自己特定的任务。

MFC 大大简化了用户的开发工作，缩短了用户开发所需的时间，同时 MFC 并没有降低程序开发的自由性。使用 MFC 将大大方便用户的工作。

下面将逐步演示如何使用 AppWizard 建立一个马上就可以运行的 MFC 应用程序——“ Hello，World !”。

2.1.1 建立新的应用程序项目

和大多数的 Windows 应用程序一样，在 VC6.0 的集成开发环境的“ File ”菜单下有“ New ”命令，用来建立各种新的内容。选择“ File ”菜单下的“ New ”命令，弹出“ New ”对话框，如图 2-1 所示。

为了建立一个新的 MFC 应用程序，选择“ Projects ”标签，并选中“ MFC AppWizard （.exe）”项，在“ Project name ”框中输入新项目的名称“ Hello ”，在“ Location ”框中输入保存本项目文件的路径，或者单击右边的扩展按钮选择相应的路径，并且选中“ Create new workspace ”选项，并选择“ Win32 ”平台。

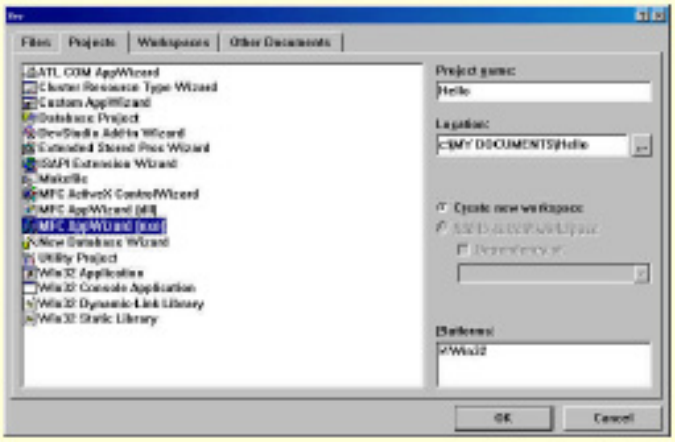

图 2-1 “New”对话框

单击“OK”按钮，AppWizard 将在指定的路径下建立新的项目，并弹出“MFC AppWizard – Step 1”对话框。

1．选择应用程序界面的类型

在“MFC AppWizard – Step 1”对话框中，主要是选择应用程序界面的类型和应用程序资源的语言种类，如图 2-2 所示。

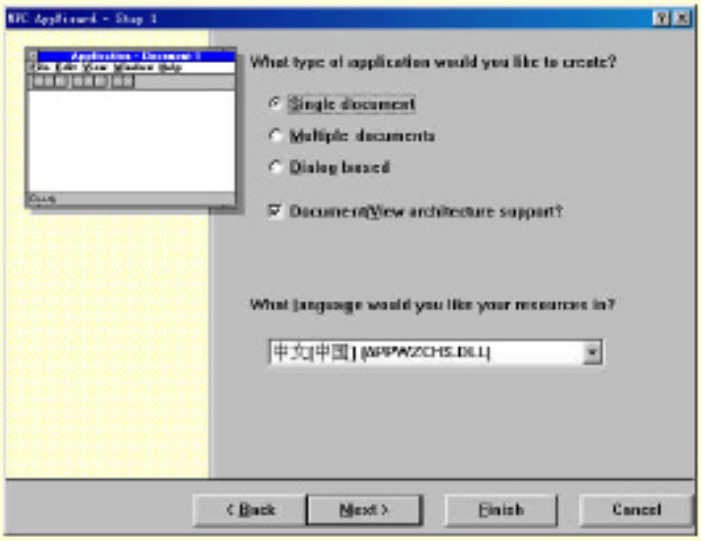

图 2-2 “MFC AppWizard – Step 1”对话框

在“Step 1”对话框中，有三种应用程序类型可供选择：

- 单文档界面（SDI：Single Document Interface）应用程序：在程序的运行过程中，每次只能打开一个文档。如果选择程序“File”菜单下的“New”命令或“Open”命令，程序首先将关闭当前正在使用的文档，然后才执行建立新文档或打开旧文档的操作。Windows 系统附件中的记事本就是一个典型的单文档界面应用程序。
- 多文档界面（MDI：Multiple Document Interface）应用程序：在程序运行时，允许同时打开两个或两个以上的文档。程序一般拥有 Window 菜单以调整各文档窗口之间的位置关系。Office 系列中的 Word 组件就是一个典型的多文档界面应用程序。
- 对话框形式的应用程序：程序的主要界面表现为一个对话框，可以使用对话框编辑器方便地进行设计。

作为一个简单的例子程序，“Hello，World！”程序只需要单文档界面。

在“Step 1”对话框中还可以选择本应用程序是否需要文档/视图结构的支持。文档/视图结构是 MFC 提供的方便用户进行文件存取、数据显示的一种程序框架，在后面的章节中将详细进行讨论。一般可以这样认为，如果程序需要存储和读取磁盘文件，一般就需要选中文档/视图结构支持。“Hello，World！”程序需要文档/视图结构支持。

在“ Step 1 ”对话框中的最后一项是选择本应用程序所使用资源的语言类型，如果选择英语，那么应用程序中的菜单、对话框等各种资源中的文本将是英语；如果选择中文，那么应用程序将具有中文的界面。用户可以根据自己的需要选择合适的语言类型。这里选择英语作为“ Hello，World !”程序的资源的语言类型。为了方便起见，本书中的例子程序全部采用英语作为资源类型。

完成 Step 1 的操作后，单击“ Next ”按钮进入下一步。

2．选择应用程序的数据库支持类型

在“ MFC AppWizard – Step 2 ”对话框中，主要是选择应用程序的数据库支持类型，如图 2-3 所示。

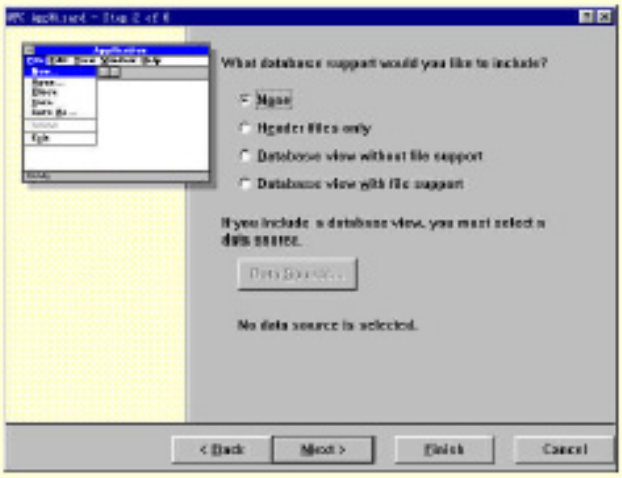

图 2-3 “ MFC AppWizard – Step 2 ”对话框

可以看到，在“ Step 2 ”对话框中提供了四种数据库支持类型供用户选择：

- 如果本应用程序不需要任何形式的数据库支持，选择“ None ”；
- 如果本应用程序需要访问数据库，但不需要派生自己的视图类，选择“ Header files only ”；
- 如果本应用程序需要派生自己的视图类来显示数据库中的记录，但不需要附加的文件存取功能，选择“ Database view without file support ”；
- 如果本应用程序既需要派生自己的视图类来显示数据库中的记录，又需要附加的文件存取功能，则选择“ Database view with file support ”；

如果选择了任何一种形式的数据库支持，“ Data Source ”按钮将变为有效，用户可以由此选择本应用程序将要处理的数据库的名称。

“ Hello，World !”程序不需要任何数据库功能，因此选择“ None ”。

完成 Step 2 的操作后，单击“ Next ”按钮进入下一步。

3．选择应用程序的 ActiveX 技术支持类型

在“ MFC AppWizard – Step 3 ”对话框中，主要是选择应用程序在 ActiveX 技术方面的支持类型，如图 2-4 所示。

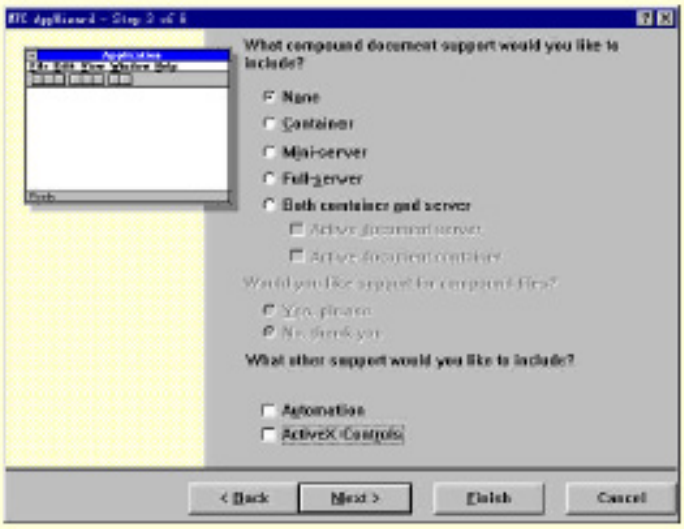

图 2-4 “MFC AppWizard – Step 3”对话框

这里所谓 ActiveX 技术，是 OLE 技术的发展，这里提供了五种选择：

- 如果不需要任何形式的 ActiveX 技术支持，选择“None”；
- 如果本应用程序需要作为一个 ActiveX 容器，即能够包含外来 ActiveX 对象，选择“Container”；
- 如果本应用程序需要作为一个 ActiveX 服务器，即自己的内容可以作为 ActiveX 对象嵌入其他程序中，但不需要独立运行，选择“Mini-server”；
- 如果本应用程序既需要作为一个 ActiveX 服务器，又需要独立运行，选择“Full-server”；
- 如果本应用程序既需要作为一个 ActiveX 容器，又需要作为一个 ActiveX 服务器，选择“Both container and server”。

如果选择了应用程序支持 ActiveX 技术——可以作为容器或服务器，还可以选择是否支持 Active 文档和复合文件。另外，即使本应用程序既不是容器也不是服务器，还可以选择是否支持 Automation 和 ActiveX 控件。

有关 ActiveX 技术的知识，将在第七章中讨论。

“Hello，World!”程序不需要支持任何形式的 ActiveX 技术，所以在“Step 3”对话框中选择“None”，并清除所有已经选中的选项。

完成 Step 3 的操作后，单击“Next”按钮进入下一步的操作。

4. 选择应用程序的界面风格

在“MFC AppWizard – Step 4”中，用户可以根据自己的需要对应用程序的界面进行定制，如图 2-5 所示。

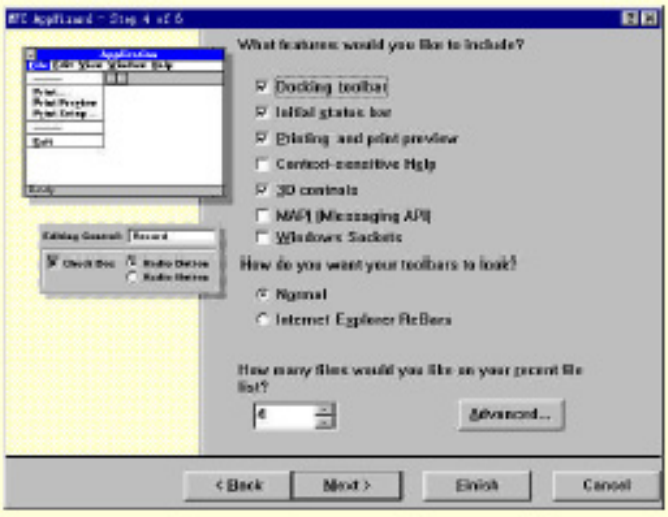

图 2-5 “MFC AppWizard – Step 4”对话框

在“Step 4”对话框中，有一些选项可供用户选择：

- 浮动工具栏（Docking toolbar）：AppWizard 可以为应用程序提供一个缺省的工具栏，包含一系列常用的按钮，如新建、打开和保存文件等。用户可以按自己的要求添加或删除按钮，或改变工具栏的外观。
- 初始状态栏（Initial status bar）：AppWizard 可以为应用程序提供一个缺省的状态栏，包含一系列常用的指示器，如指示 Caps Lock、Num Lock 的状态。用户同样可以按自己的要求定制状态栏。
- 打印和打印预览（Printing and print preview）：AppWizard 将在应用程序中插入一部分代码，完成基本的打印和打印预览任务，用户可以在此基础上进行完善。
- 上下文相关帮助（Context-sensitive Help）：AppWizard 提供相应的代码和资源，为用户建立一个基本的帮助系统，具体的内容需要由用户自己完成。
- 3D 控件（3D controls）：比平面的控件更有立体感，更美观。
- 消息收发支持（MAPI）：AppWizard 提供相应的代码，使得应用程序可以通过 MAPI 发送电子邮件、传真或其他消息。
- 插口支持（Windows Sockets）：应用程序可以通过 HTTP、FTP 等协议访问互联网。

用户还可以选择应用程序的工具栏的风格是普通风格还是 IE4 风格，这是 VC6.0 中新增加的选项。在 Step 4 中有一个“ Advanced ”按钮，单击该按钮将弹出“ Advanced Options ”对话框，在该对话框中可以修改应用程序的文档模板信息字符串，或者更改应用程序窗口的风格，这里不作详细介绍。

对“ Hello，World !”程序而言，保持缺省的选项就行了。

完成 Step 4 的操作后，单击“ Next ”按钮进入下一步操作。

5．选择应用程序的其他选项

在“ MFC AppWizard – Step 5 ”对话框中，只有三个选项，如图 2-6 所示。

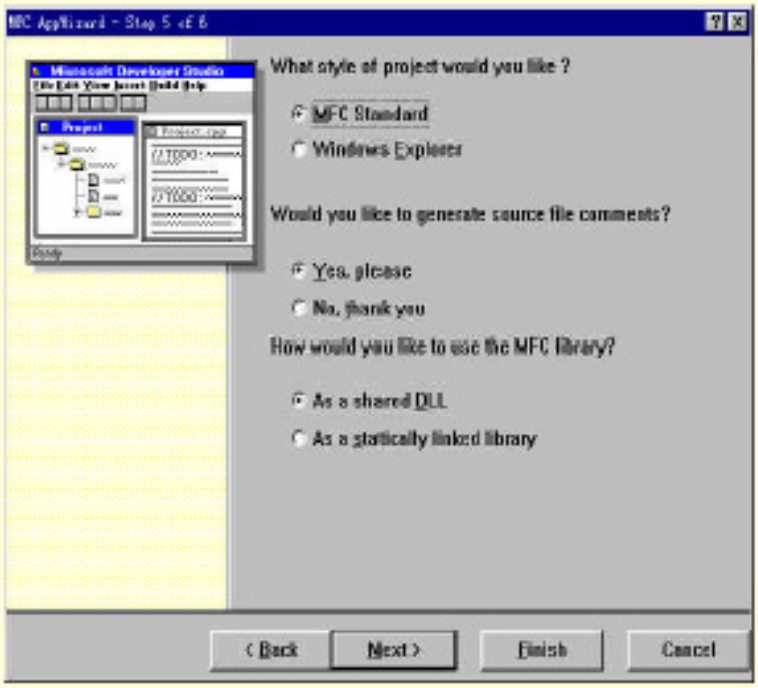

图 2-6 “ MFC AppWizard – Step 5 ”对话框

Step 5 的三个问题都很简单，但是不同的选择将对最终生成的应用程序的性能有很大的影响。

- 第一个问题是询问应用程序的风格，用户可以在 MFC 标准风格和 Windows Explorer 风格中作出选择，一般都选择 MFC 标准风格。
- 第二个问题是询问是否在源代码中插入相应的注释，答案当然是需要。如果源代码中没有注释，阅读和维护这些代码将是一项令人烦恼的工作。
- 第三个问题是询问以何种形式将 MFC 类库连接到应用程序中。静态连接将导致可执行文件所占空间增大，但是安装起来比较方便；动态连接可以减小可执行文件的尺寸，增加程序运行的速度，但安装时需要将用到的 dll 文件拷贝到目的系统中，比较麻烦。一般情况下都选择动态连接方式。

对“Hello，World!”程序而言，保持缺省的选项就行了。

完成 Step 5 的操作后，单击“Next”按钮进入下一步的操作。

6. 确认类名和文件名

在“MFC AppWizard – Step 6”对话框中列出了本应用程序所用到的所有类的类名和保存这些类的文件的文件名，如图 2-7 所示。

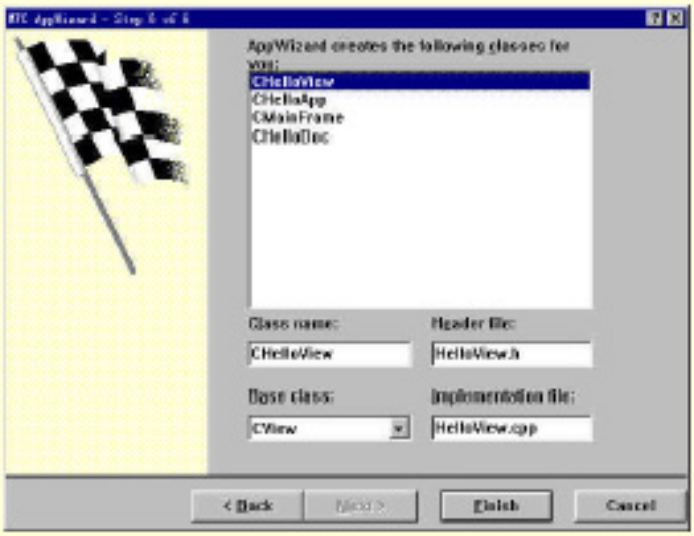

图 2-7 “MFC AppWizard – Step 6”对话框

用户可以检查列出的所有信息，或者按照自己的要求进行修改。在这一步中，最重要的是为视图类选择不同的基类，以实现不同的功能。

对“Hello，World!”程序而言，无需做任何修改。单击“Finish”按钮完成应用程序的设置，将弹出“New Project Information”对话框显示新项目的有关信息，如图 2-8 所示。

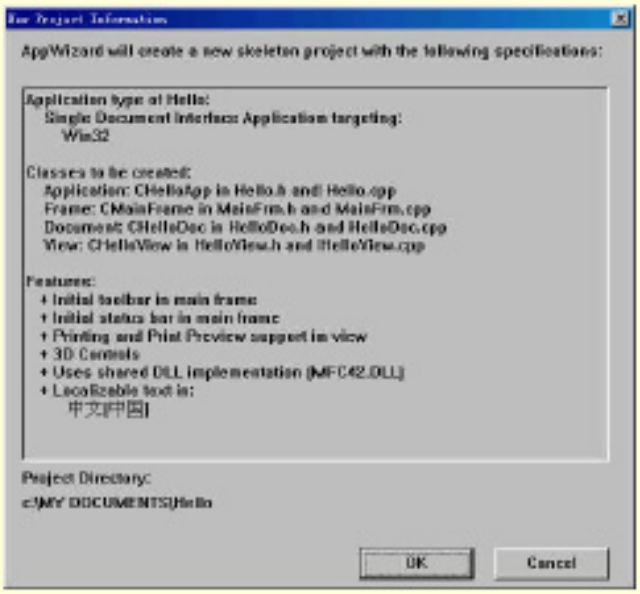

图 2-8 “New Project Information”对话框

用户可以检查“New Project Information”对话框中列出的本项目的所有设置是否符合自己的需要，如果是，单击“OK”按钮，AppWizard 将为用户建立相应的代码和资源；如果不满足要求，单击“Cancel”按钮可以重新进行设置。

2.1.2 为程序添加代码

AppWizard 已经为用户产生了大量的代码，用户可以编译和运行该应用程序，以检查生成的代码。运行中的程序如图 2-9 所示。

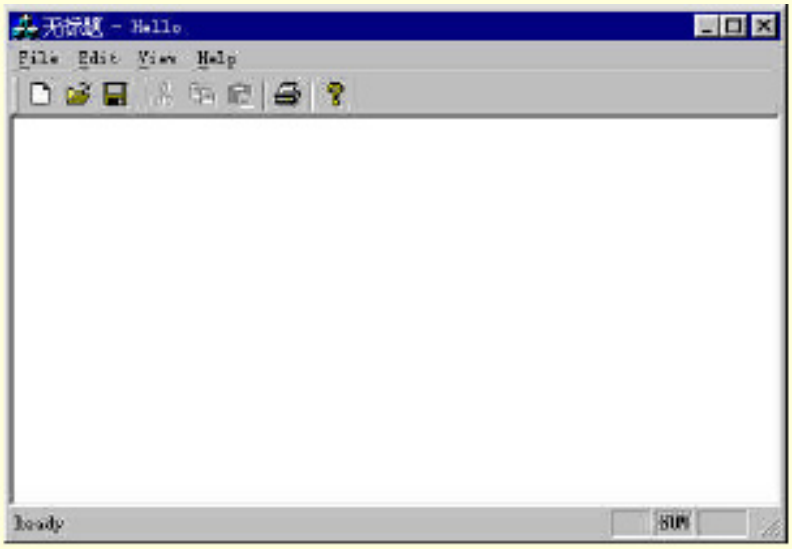

图 2-9 运行中的应用程序

可以看到，AppWizard 为用户生成了一个相当完善的“标准的”Windows 应用程序，该程序窗口拥有自己的菜单、工具栏和状态栏，可以移动位置或缩放大小，可以说，一般的 Windows 应用程序应该有的功能，都已经有了。但是，这个应用程序不能完成任何任务，不能响应键盘或鼠标输入，也不能显示任何有用的信息，还需要用户进行进一步的完善。在这个框架应用程序的基础上，用户可以专注于自己的特殊需要，完成特定的任务。

由于 AppWizard 为用户生成的框架应用程序什么也不能干，为了显示“Hello，World !”字符串，需要用户添加一些代码，具体的操作步骤如下：

1. 为 CHelloDoc 类添加成员

在 MFC 应用程序的文档/视图结构中，文档类用来完成应用程序数据的存取工作，视图类完成显示数据和与使用者交互的工作。在“Hello，World !”程序中，应用程序的数据很简单，就是一个字符串。在 MFC 中，提供了 CString 类来处理字符串，可以说，CString 类提供的成员函数几近完美地解决了所有的字符串操作。

为了在“Hello，World !”程序的视图中显示字符串，需要在 CHelloDoc 类中添加一个 CString 类型的成员变量 m_szMessage 以保存字符串信息；为了视图类能够方便地获得文档类中保存的字符串信息，CHelloDoc 类需要提供一个成员函数 GetMessage()。下面介绍如何向 CHelloDoc 类添加成员变量和成员函数。

在项目工作台的 ClassView 中，将光标定位于 CHelloDoc 类，单击鼠标右键，弹出一个快捷菜单，该快捷菜单是 CHelloDoc 类当前可以执行的命令集合，如图 2-10 所示。

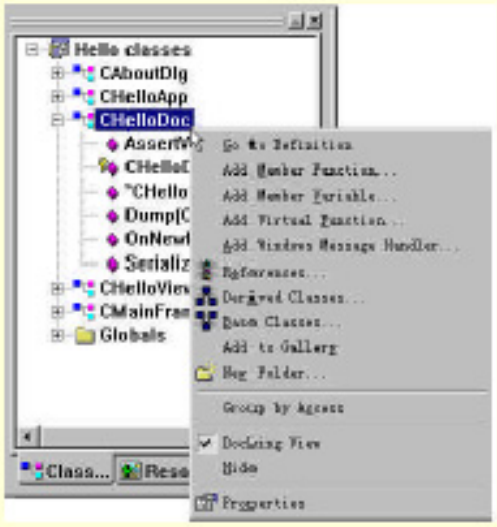

图 2-10 类的快捷菜单

在快捷菜单中选择“Add Member Variable（添加成员变量）”命令，弹出“Add Member Variable”对话框，如图 2-11 所示。

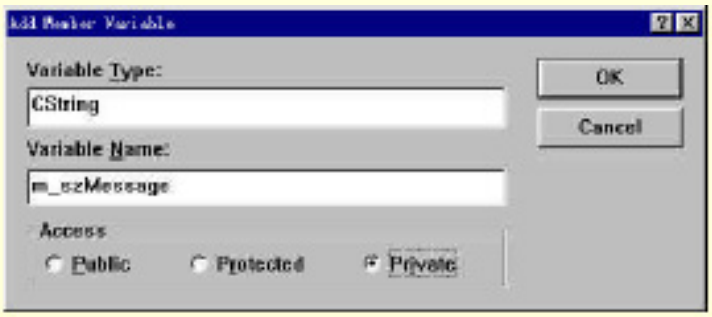

图 2-11 “Add Member Variable”对话框

在 Add Member Variable 对话框中，输入变量类型 CString 和变量名称 m_szMessage，选择变量的存取级别为“Private（私有）”。单击“OK”按钮即完成了向 CHelloDoc 类中添加成员变量的工作。

需要说明的是，如果将类的成员变量或成员函数的存取级别设置为“Private”，则只有类本身的成员或其友员（用“friend”关键字修饰的函数或类）才能够访问该成员，除此之外的任何情况下都不能访问该成员。这样就可以保护类的私有数据不会被随意改变，这是面向对象的设计方法的一条重要原则。

类似地，在快捷菜单中选择“Add Member Function（添加成员函数）”命令，弹出“Add Member Function”对话框，如图 2-12 所示。

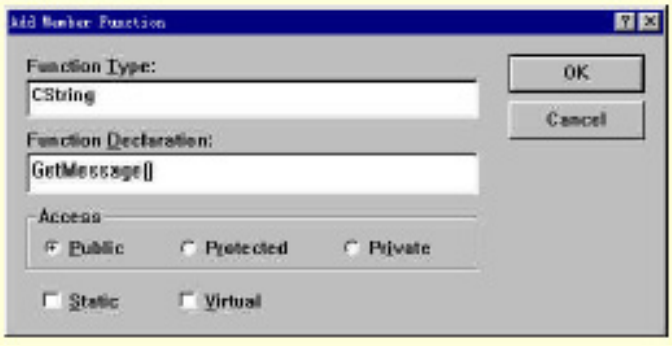

图 2-12 “Add Member Function”对话框

在“Add Member Function”对话框中，输入函数返回值类型 CString 和函数的完整声明 GetMessage()，选择函数的存取级别为“Public（公有）”。注意在输入函数的声明时，如果该函数是带参数的，则同时需要输入

参数的完整列表。当然，由于 GetMessage()函数无需任何参数，这里就不用输入了。

如果将类的成员的存取级别设置为“ Public ”，则该成员可以在任何函数中被访问。另外一种存取级别是“ Protected（保护）”，被设置为“ Protected ”的成员可以被类本身的成员、友员和派生类访问。

一般说来，类的私有数据都应设置为“ Private ”或“ Protected ”，类与外界的接口函数被设置为“ Public ”，这样既能够保护类的私有数据，又可以保证与其他对象间的数据交换。这是开发较大型的应用程序时保证对象数据的完整性和有效性时常用的一种做法。

添加的成员变量和成员函数立刻显示在 ClassView 的类及其成员的列表中。下面的工作是向有关的函数中添加合适的代码。

2．向 CHelloDoc 类的成员函数中添加代码

正如其名称所描述的那样，GetMessage()函数的功能就是提供其他类获取文档内部数据结构的接口。其实 GetMessage()函数相当简单，只需一行代码。

首先需要定位到 GetMessage()函数中，WizardBar 为用户在代码中跳转提供了方便的途径。操作方法如图 2-13 所示。

图 2-13　WizardBar 的操作

在 WizardBar 的类名下拉框中选择 CHelloDoc 类，在成员下拉框中选择 GetMessage()函数，则 HelloDoc.cpp 文件将作为当前文档被打开，光标定位于 GetMesssage()函数处。清单 2-1 中列出了完成后的 GetMessage()函数的代码：

程序清单 2-1　CHelloDoc::GetMessage()

```
CString CHelloDoc::GetMessage()
{
//返回文档类的内部数据结构
return m_szMessage;
}
```

在 GetMessage()函数中，以代码显示的内容是需要用户手工输入的，而“ // ”后面在中文注释的部分是由笔者添加的。当然，读者在实际的练习中，可以直接跳过输入注释的工作而只输入英文程序代码。

另外一个需要添加代码的函数是 OnNewDocument()。很显然，OnNewDocument()函数将在创建新的文档对象时被调用，因此在该函数中进行文档数据结构的初始化工作是合适的。在“ Hello，World !”应用程序中，需要在 OnNewDocument()函数中给 m_szMessage 成员赋值。清单 2-2 中是完整的 OnNewDocument()函数。

程序清单 2-2　CHelloDoc::OnNewDocument()

```
BOOL CHelloDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
        return FALSE;
//  初始化文档类的内部数据结构
m_szMessage="Hello,World!";
return TRUE;
}
```

注意：

- 在 AppWizard 自动生成的代码中，首先调用了基类版本的 OnNewDocument()函数进行一些必要的初始化工作。在以后用户将不断看到这种情况出现。

现在已经完成了向 CHelloDoc 类中添加代码的工作。

这里用户可能会对 GetMessage()函数的作用产生一点疑惑，如果仅仅是为了返回文档类的内部数据——m_szMessage 字符串，是否有必要专门添加一个函数，可否直接将 m_szMessage 成员的存取级别设置为“ Public ”，以供其他对象访问？应该说，确实可以直接将 m_szMessage 设置为“ Public ”，这完全可以满足要求并且更简单。但是，这样做不符合面向对象、封装对象内部数据的原则。在做一般的小型程序的时候，为方便可以走一些捷径，但在开发较大型的程序时，必须严格遵守开发原则。

3．向 CHelloView::OnDraw()函数添加代码

前面已经提到过，MFC 应用程序的文档类负责数据的存取，而视图类负责显示数据和交互工作。因此，显示“ Hello，World！”字符串的工作应该在 CHelloView 类的某个成员函数中完成。这个函数就是 OnDraw()，其代码负责在屏幕上“ 描画 ”程序的数据。

清单 2-3 中列出了完成后的 OnDraw()函数：

程序清单 2-3　CHelloView::OnDraw()

```
void CHelloView::OnDraw(CDC* pDC)
{
CHelloDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// 从文档类中获取数据
CString s=pDoc->GetMessage();
// 在适当的位置显示字符串
pDC->TextOut(50,50,s);
}
```

为了视图类对象可以存取文档类对象中的数据，一方面需要文档类提供合适的成员函数，另一方面也需要能够在视图类中方便地获取文档类对象（或对象的指针）。在 MFC 的文档/视图结构中，视图类提供了 GetDocument()函数以获取与视图有关的文档类对象的指针，通过该指针即可调用文档类的成员。

在 CHelloView::OnDraw()函数中，AppWizard 为用户产生的代码中就已经调用了 GetDocument()函数，为用户提供了 pDoc 指针。实际上，用户可以在任何需要使用文档类对象的地方调用 GetDocument()函数。

在获得文档中保存的数据后（由临时生成的 CString 对象 s 保存），TextOut()函数在指定的位置将字符串显示出来。这里 TextOut()函数是 CDC 类——设备环境类——的一个成员成员，函数的功能就是在指定位置输出指定字符串。

在 Windows 系统中，屏幕和打印机等输出设备由设备环境类（CDC 类）进行了封装，该类中包含了所有这些输出设备的信息，并提供了大量的成员函数供用户使用，如画直线、画简单曲线和输出文本等。设备环境提供了 Windows 环境下的设备无关性，用户无需考虑具体的输出对象，只要在“ 设备环境 ”上能够正确地绘图就可以了，其他与硬件有关的工作由 Windows 系统完成。在后面的两章中，将介绍有关设备环境的详细内容。

用户可以再次编译和运行“ Hello，World！”程序，结果如图 2-14 所示。

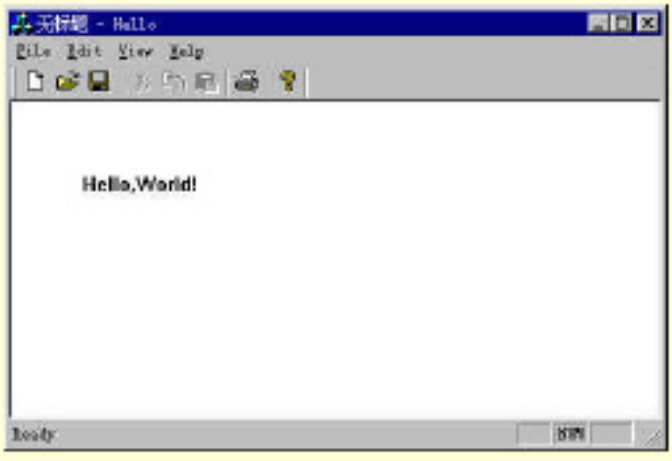

图 2-14　运行中的“ Hello，World！ ”程序

可以看到，在应用程序窗口的客户区中，正确显示了“Hello，World!”字符串，这表明已经成功地完成了“Hello，World!”程序。

在本节中，用户按照 AppWizard 的提示，一步一步地生成了“Hello，World!”程序的框架，然后添加了一些必要的代码，最终生成了一个正确执行的“Hello，World!”程序。在建立“Hello，World!”程序的过程中，用户可能已经感觉到，使用 AppWizard 建立应用程序是十分方便快捷的，不用自己书写任何代码，AppWizard 就已经按用户的定制要求建立了一个可以正常运行的起始应用程序。

在下一节中，将对 AppWizard 为用户生成的代码做一些解释，用户将在此基础上进一步了解 MFC 应用程序的结构和运行。

2.2 分析 MFC 程序结构

在上一节中，介绍了如何使用 AppWizard 建立“Hello，World!”程序，在本节中将有重点地介绍该程序的代码。只有理解了 MFC 应用程序的基本结构和运行过程，才能按照自己的要求在合适的位置添加合适的代码，完成特定的功能。

AppWizard 为“Hello，World!”程序生成了五个类，这些类的类名都是基于应用程序名而产生的：

- CHelloApp：应用程序类
- CMainFrame：主框架窗口类
- CHelloDoc：文档类
- CHelloView：视图类
- CAboutDlg：版本信息对话框类

下面对这几个类一一加以说明。如果用户注意一下 FileView 中的文件列表，就会发现每个类都有一个头文件和一个源代码文件与之对应。AppWizard 将类及类的成员的声明放在头文件中，而将类及其成员的具体定义放在源文件中，这样做是为了维护代码的方便。

2.2.1 CHelloApp 应用程序类

虽然绝大多数的 Windows 应用程序都拥有一个主窗口，但主框架窗口类并不是应用程序的基础，应用程序类才是整个应用程序的基础。在某些情况下，应用程序完全可能不需要主窗口。

1．CHelloApp 类的头文件

在 CHelloApp 类的头文件中，有一些代码（就是那些看起来比较“奇怪”的代码）是为了编译和调试应用程序的方便而由 AppWizard 添加的。这里只讨论对程序员有较大关系的代码，清单 2-4 中列出了“干净”的 CHelloApp 类的头文件。

程序清单 2-4 Hello.h

```
class CHelloApp : public CWinApp
{
public:
CHelloApp();

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CHelloApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL
```

```
// Implementation
//{{AFX_MSG(CHelloApp)
afx_msg void OnAppAbout();
    // NOTE - the ClassWizard will add and remove member functions here.
    //    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

在 Hello.h **文件中，代码行**

```
class CHelloApp : public CWinApp
```

表明 CHelloApp **类由** CWinApp **类派生。在** MFC **中，**CWinApp **类封装了** Windows **应用程序对象，并掌握了应用程序的初始化、运行和终止工作。每个应用程序有而且只能有一个** CWinApp **类（或其派生类）对象。**

接着，在 Hello.h **文件中，声明了** CHelloApp **类的构造函数** CHelloApp()**。在** Hello.cpp **文件中可以看到，**CHelloApp()**为空函数，什么工作也没有完成。**

注意在接下来的一段代码，夹在一大堆注释代码中，有这么两行：

```
public:
    virtual BOOL InitInstance();
```

这表明 InitInstance()**函数是一个公有的虚函数。在该函数中执行了应用程序的初始化工作，在稍后将对此进行重点讨论。**

其后声明的 OnAppAbout()**函数是用来处理应用程序中“** Help **”菜单下的“** About... **”命令的，这里不多作解释。**

最后的宏 DECLARE_MESSAGE_MAP()**使得** CHelloApp **类可以处理消息映射。**

在 Hello.h **文件中，有一些注释看起来是这样的：**

```
//{{AFX_...
    //Some codes...
//}}AFX_...
```

这些以 AFX_**开头的注释对是** ClassWizard **添加并管理的，用户不要对其进行任何修改，即使是注释也不要删除它们。**

2．CHelloApp **类的源文件**

在 Hello.cpp **文件中，除去开头的一段预编译代码，有** CHelloApp **类的三个成员函数的完整实现和一段消息映射。以下是清单** 2-5 **中的这段消息映射代码：**

程序清单 2-5　CHelloApp **类的消息映射**

```
BEGIN_MESSAGE_MAP(CHelloApp, CWinApp)
//{{AFX_MSG_MAP(CHelloApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
    //    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

在这段代码中，前后的 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 宏与 Hello.h 文件中的 DECLARE_MESSAGE_MAP 宏协同工作，共同管理 CHelloApp 类的消息映射。

注意这样的一行：

ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)

ON_COMMAND 宏的作用是将对标识符为 ID_FILE_NEW 的资源（包括“ File ”菜单下的“ New ”命令和工具栏上的“ New ”按钮）的命令映射到 OnFileNew()函数处理。其他的几个 ON_COMMAND 宏的作用类似。

上文中提到的几个宏共同负责完成应用程序的消息映射工作，有关知识将在第三章中详细介绍。

在 Hello.cpp 文件中，接下来是构造函数 CHelloApp()，不包含任何有意义的代码。

由于初始化工作在 InitInstance()函数中进行，因此需要对该函数作较仔细的介绍，其代码如清单 2-6 所示。

程序清单 2-6 CHelloApp::InitInstance()

```
BOOL CHelloApp::InitInstance()
{
// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.
#ifdef _AFXDLL
Enable3dControls();                 // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif

// Change the registry key under which our settings are stored.
// TODO: You should modify this string to be something appropriate
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings();  // Load standard INI file options (including MRU)

// Register the application's document templates.  Document templates
// serve as the connection between documents, frame windows and views.
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CHelloDoc),
        RUNTIME_CLASS(CMainFrame),          // main SDI frame window
        RUNTIME_CLASS(CHelloView));
AddDocTemplate(pDocTemplate);

// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
        return FALSE;
```

```
// The one and only window has been initialized, so show and update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();

return TRUE;
}
```

在 InitInstance()函数的初始化过程中，首先是根据 MFC 的连接情况（动态连接或静态连接，用户在 AppWizard 的第五步中选定）调用不同的函数以支持 3D 控件。接着完成程序的注册工作，SetRegistryKey()函数指明了本应用程序在系统注册表中的键名，LoadStdProfileSettings()完成标准的初始化文件的读取工作，其中包括了本应用程序最近使用过的文件的列表。

接下来的一段代码是这样的：

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CHelloDoc),
        RUNTIME_CLASS(CMainFrame),          // main SDI frame window
        RUNTIME_CLASS(CHelloView));
AddDocTemplate(pDocTemplate);
```

这段代码完成一项十分重要的工作：文档模板的设置。根据用户在 AppWizard 的第一步中的选择，首先定义了一个单文档模板对象，然后通过这个单文档模板将应用程序的资源（由标识符 IDR_MAINFRAME 确定）、文档类 CHelloDoc、主框架窗口类 CMainFrame 和视图类 CHelloView 联系在一起，最后通过调用 AddDocTemplate()函数将这一单文档模板对象加入到应用程序维护的文档模板列表中。

接下来的 ParseCommandInfo()和 ProcessShellCommand()函数完成分配和处理命令行信息的工作。在 DOS 方式下，如果要执行某应用程序，需要输入一长串的字符命令。

在 Windows 系统中，同样可以通过键入字符命令来执行应用程序，如输入：notepad.exe readme.txt，系统将执行记事本程序，并打开 readme.txt 文件。ParseCommandInfo()和 ProcessShellCommand()函数就是用来识别命令行信息并相应地启动应用程序的。

InitInstance()函数最后的工作就是显示和更新应用程序的窗口。

如果 InitInstance()函数中所有的初始化工作均正确地进行，函数将返回 TRUE。

在执行完初始化工作后，应用程序将进入消息循环，接收和处理系统发送来的各种消息，直到应用程序被关闭。

这里有必要解释一下应用程序的初始化问题。在 C++语言中，类的构造函数通常都是类的成员变量进行初始化的地方。然而在 CWinApp 类中，可以看到，构造函数中并没有执行任何初始化工作，而是由 InitInstance()函数完成的，实际上，在 CWinApp()函数中也是可以完成初始化任务的。但是，CWinApp()与 InitInstance()函数有一些区别。

Windows 系统允许同时运行同一应用程序的多个拷贝。从面向对象的角度来说，应用程序是某个特定的类，多个拷贝则是该类的多个实例，也就是说，Windows 系统允许类的多个实例同时存在。在进行初始化的时候，就需要考虑到这种情况。MFC 是这样处理的，在 CWinApp()函数中进行类的初始化，在 InitInstance()函数中进行实例的初始化。

Hello.cpp 文件的最后一段代码是 CAboutDlg 类的声明与定义，与 CHelloApp 类类似，这里不多做分析。

2.2.2 CMainFrame 主框架窗口类

在介绍 CMainFrame 类之前，有必要先解释一下应用程序的主框架窗口、子框架窗口和视图的概念。图 2-15 很清楚地揭示了这些“术语”间的区别与联系。

从图中可以看出，主框架窗口是整个应用程序的主窗口，子框架窗口占据了主框架窗口客户区的部分或者

全部，而视图则占据了子框架窗口的客户区的部分或者全部。在 MFC 的文档/视图结构下，用户的大部分显示工作都是在视图中完成的。图 2-15 是一个多文档界面应用程序的例子，对于单文档界面的应用程序，由于没有子框架窗口，可以认为是视图充满了主框架窗口的整个客户区。

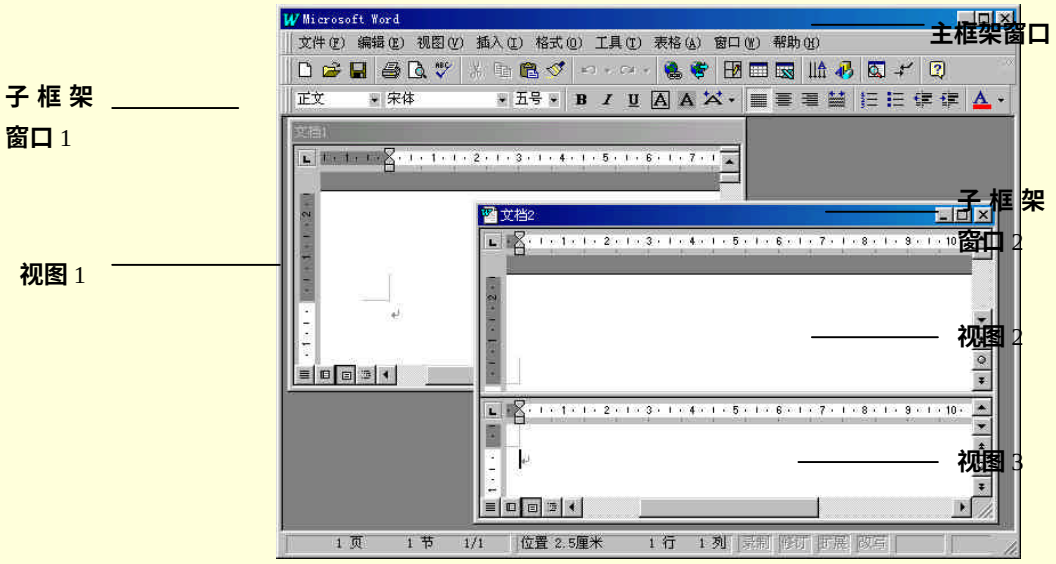

图 2-15　主框架窗口、子框架窗口与视图

下面来具体介绍 CMainFrame 类的头文件和源文件。

1．CMainFrame 类的头文件

清单 2-7 中列出了 MainFrm.h 文件中比较重要的部分。

程序清单 2-7　MainFrm.h

```
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
CMainFrame();
DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CMainFrame)
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CMainFrame();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
```

```
#endif

protected:    // control bar embedded members
CStatusBar    m_wndStatusBar;
CToolBar      m_wndToolBar;
// Generated message map functions
protected:
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
      // NOTE - the ClassWizard will add and remove member functions here.
      //       DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

在 MainFrm.h **文件中，值得注意的是** DECLARE_DYNCREATE **宏，该宏与** MainFrm.h **文件中的** IMPLEMENT_DYNCREATE **宏共同工作。使用了这一对宏的类的对象可以在程序运行过程中被动态创建。**

接下来声明了虚函数 PreCreateWindow()**，在** MainFrm.cpp **文件中可以看到，该函数仅仅调用了其基类版本，但用户可以用自己的代码覆盖该函数。**PreCreateWindow()**函数在窗口创建之前被调用，因此可以在该函数中添加代码以改变窗口的风格。**

注意这一段代码：

```
protected:    // control bar embedded members
      CStatusBar    m_wndStatusBar;
      CToolBar      m_wndToolBar;
```

这段代码声明了一个 CStatusBar **对象和一个** CToolBar **对象，实际上，这就是主框架窗口所拥有的状态栏和工具栏对象。从这里也可以看出，所谓“程序”的状态栏和工具栏实际上是主框架窗口的子窗口。**

MainFrm.h **文件的最后一段声明了几个消息处理函数。**

2．CMainFrame **类的源文件**

在 MainFrm.cpp **文件中，需要注意的只有** CMainFrame::OnCreate()**函数。清单** 2-8 **中列出了该函数的代码，并加上了一些注释。**

程序清单 2-8　CMainFrame::OnCreate()

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
// 创建主窗口
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
      return -1;

// 创建工具栏
if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
                  WS_VISIBLE | CBRS_TOP    | CBRS_GRIPPER |
                  CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
      TRACE0("Failed to create toolbar\n");
      return -1;          // fail to create
```

```
}

// 创建状态栏
if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
{
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
}

// TODO: Delete these three lines if you don't want the toolbar to
//    be dockable
// 泊定工具栏
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);

return 0;
}
```

在 OnCreate()**函数中，以代码显示的注释表明了每小段代码的作用。具体的函数说明将在以后的章节中加以介绍。**

2.2.3 CHelloDoc 文档类

文档类中保存着应用程序的数据，该类负责正确地从磁盘文件中读取数据和向磁盘文件中写入数据，维护应用程序的数据结构正常工作。CHelloDoc **类从** CDocument **类派生。**

1．CHelloDoc 类的头文件

清单 2-9 **中列出了** HelloDoc.h **文件的代码。**

程序清单 2-9　HelloDoc.h

```
class CHelloDoc : public CDocument
{
protected: // create from serialization only
CHelloDoc();
DECLARE_DYNCREATE(CHelloDoc)
// Attributes
public:
// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CHelloDoc)
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
```

```
//}}AFX_VIRTUAL

// Implementation
public:
CString GetMessage();
virtual ~CHelloDoc();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif

protected:
// Generated message map functions
protected:
//{{AFX_MSG(CHelloDoc)
        // NOTE - the ClassWizard will add and remove member functions here.
        //       DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
private:
CString m_szMessage;
};
```

CHelloDoc 类的头文件中，声明了两个很重要的函数：OnNewDocument()和 Serialize()。这两个函数都是虚函数，OnNewDocument()函数在创建新文档的时候被调用，Serailize()函数在应用程序与磁盘文件交换数据时被调用。

注意 HelloDoc.h 文件中以代码标识的代码，这并不是用户手工添加的，但是，这些代码是用户在 2.1.8 小节中使用对话框向 CHelloDoc 类添加成员变量 m_szMessage 和成员函数 GetMessage()时由 AppWizard 自动添加的。实际上，用户完全可以直接在 HelloDoc.h 文件中添加类似的代码，不需要通过“ Add member ... ”对话框。

2．CHelloDoc 类的源文件

在 HelloDoc.cpp 文件中，GetMessage()函数是用户添加的，OnNewDocument()函数也由用户修改过，在 2.1.8 小节中已经对这两个函数作过介绍，这里不再赘述。

值得注意的是 Serialize()函数，该函数的功能就是储存和加载数据，AppWizard 为用户生成的 Serialize()函数的代码列于清单 2-10 中。

程序清单 2-10　CHelloDoc::Serialize()

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

在 Serialize()函数中，两行注释很清楚地指明了添加储存数据和加载数据的代码的位置。MFC 将应用程序添加数据和加载数据的过程称为串行化，在第六章中将讨论有关文档串行化的内容。

2.2.4 CHelloView 视图类

在 MFC 应用程序的文档/视图结构中，视图类的作用是以图形或文本的方式显示应用程序的数据，并接受应用程序用户对文档的操作。可以说，使用 MFC 建立 Windows 应用程序的很大一部分的工作就是完善自己的文档类和视图类，使之符合特定的需求。

1．CHelloView 类的头文件

在 HelloView.h 文件中，声明了一批虚函数，实际上，CHelloView 类的基类——CView 类提供了更多的虚函数供用户在适当的时候进行覆盖。

清单 2-11 中列出了 CHelloView 类的头文件。

程序清单 2-11 HelloView.h

```
class CHelloView : public CView
{
protected: // create from serialization only
CHelloView();
DECLARE_DYNCREATE(CHelloView)

// Attributes
public:
CHelloDoc* GetDocument();
// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CHelloView)
public:
virtual void OnDraw(CDC* pDC);    // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
//}}AFX_VIRTUAL

// Implementation
public:
virtual ~CHelloView();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif

protected:
```

```
// Generated message map functions
protected:
//{{AFX_MSG(CHelloView)
    // NOTE - the ClassWizard will add and remove member functions here.
    //    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

在这些虚函数中，有些是与视图类如何显示数据有关的，如 OnDraw()函数；有些是与对窗口本身的操作有关的，如 PreCreateWindow()函数；有些是与打印有关的，如 OnBeginPrint()函数等等。

2．CHelloView 类的源文件

在 HelloView.cpp 文件中，PreCreateWindow()函数和 OnDraw()函数已经在前面作过介绍，应该注意的三个与打印和打印预览有关的函数，AppWizard 为这三个函数生成了一个空壳，如清单 2-12 所示。

程序清单 2-12　HelloView.cpp 中与打印有关的部分

```
BOOL CHelloView::OnPreparePrinting(CPrintInfo* pInfo)
{
// default preparation
return DoPreparePrinting(pInfo);
}

void CHelloView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
// TODO: add extra initialization before printing
}

void CHelloView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
// TODO: add cleanup after printing
}
```

这些函数的函数名和注释很清楚地指明了各函数的作用，OnPreparePrinting()函数在开始打印之前，准备打印的过程中被调用，在该函数中可以对打印对话框进行初始化，可以规定待打印文档的长度等；OnBeginPrinting()函数在开始打印时被调用，可以在该函数中分配打印过程中需要用到的 GDI 资源等；OnEndPrinting()函数则在打印过程结束时被调用，主要是负责清除已分配的 GDI 资源等收尾工作。在第六章中也讨论到了一些有关打印和打印预览的内容。

现在已经较为详细地介绍了 AppWizard 为用户生成的类的头文件和源文件，介绍了各类在应用程序中的基本作用。在本书的后续章节中，用户将逐渐更清楚地理解 MFC 应用程序中各类的作用。

2.2.5　理解其他文件

如果用户打开项目工作台中的“ FileView ”标签，并稍微浏览一下所列出的内容的话，很容易发现“ Hello，World !”应用程序还包含其他的一些文件。下面简单介绍这些文件的作用。

除了 CHelloApp、CMainFrame、CHelloDoc、CHelloView 等四个类的头文件和源文件以外，“ Hello，World !”程序还包括另外的两个代码文件：StdAfx.h 和 StdAfx.cpp。这两个文件用来产生一个预编译的头文件和一个预编译的目标文件，AppWizard 会自动管理这两个文件，一般情况下，用户无需对这两个文件进行任何操作或修改。

另外，Resource.h 和 Hello.rc 文件中记录了本应用程序所用到的所有资源的信息，这包括应用程序的菜单、

工具栏、“About”对话框等各种资源。在 Resource Files 项目下包含的一些文件，一般都是本应用程序所用到的一些位图信息。这些文件都保存在“res”子目录下。

最后，还有一些文件（后缀为 dsp、dsw、opt、clw 等的文件）保存了本应用程序或者说明本项目的一些信息。这些都由 AppWizard 进行管理，用户不必自行处理。

2.3 MFC 程序的运行过程

用户如果曾经直接使用 Windows API 函数编写过 Windows 应用程序，一定还记得 WinMain()函数和用来处理消息循环的回调函数中长长的 select...case...语句。这种应用程序的运行过程是比较容易理解的：WinMain()函数是应用程序的入口，在该函数中完成程序的初始化工作，然后进入消息循环，具体的消息由回调函数中的 select...case...语句处理，直至应用程序结束运行。

但是，在 MFC 应用程序中，正如上一节中浏览“Hello，World！”应用程序的代码时所看到的，并没有出现 WinMain()函数和 select...case...语句。MFC 采用了另外的一种方式，应用程序的初始化是在 InitInstance()函数中完成的，操作系统向应用程序发送的消息则由消息映射宏映射到特定的函数进行处理。

但如果进一步深入地研究 MFC 的源代码，可以发现 WinMain()函数还是存在的，但是 MFC 对此进行了封装，使得用户不需要直接处理 WinMain()函数。在应用程序启动的时候，应用程序框架将调用 WinMain()函数。该函数执行一些标准的初始化过程，如注册窗口类等；然后该函数将调用应用程序对象的成员函数，初始化应用程序并进入应用程序的消息循环。

在初始化应用程序对象的时候，WinMain()函数将调用应用程序类的成员函数 InitApplication()和 InitInstance()；接着调用应用程序类的 Run()进入消息循环；在应用程序终止的时候，WinMain()调用应用程序类的 ExitInstance()函数退出应用程序。图 2-16 清楚地显示了这一过程。

在图 2-16 中，标明的 WinMain()函数和 Run()函数是应用程序框架提供了标准的处理函数，无需用户更改；而 InitInstance()函数和 ExitInstance()函数需要用户进行覆盖和修改。

在应用程序的执行过程中，除了启动时的初始化过程和退出时的清理过程外，大部分的时间都在执行应用程序对象的 Run()函数，即处于消息循环中，因此，在一般情况下，处理应用程序的消息就是用户开发 MFC 应用程序最重要的任务。

在下一章中，我们将介绍 MFC 的消息机制，用户将更清楚地理解应用程序的执行过程。

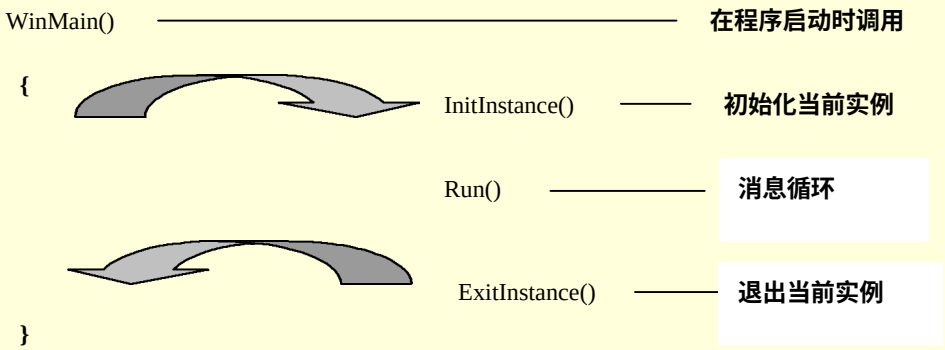

图 2-16 MFC 应用程序的运行过程

2.4 本章小结

在本章中，通过使用 AppWizard 建立“Hello，World！”示例程序，介绍了 AppWizard 的各步骤中一些选项的含义；并通过向“Hello，World！”示例程序中添加代码，展示了开发 MFC 应用程序的基本步骤；最后，通过介绍“Hello，World！”示例程序的代码，使读者基本了解了 MFC 应用程序的基本结构和运行过程。

在本章的基础上，本书将逐步向读者介绍使用 MFC 进行开发的多个方面，读者将逐渐了解到 MFC 所提供的强大的功能和便捷的开发途径。

第三章 MFC 消息与命令

Windows 应用程序与 MS-DOS 程序一个很大的不同点就在于其驱动方式：MD-DOS 程序使用顺序执行的模式，如果要取得用户的输入，则需要用巨大的循环来不断地检测键盘或鼠标，或其他的输入设备；在 Windows 应用程序中，系统用消息来向应用程序传递信息，程序本身不需要去检测输入设备的动作，只要对系统传递过来的消息给予正确的处理就行了。

传统的 Windows 应用程序和 MFC 应用程序也还有一些不同：在传统的 Windows 应用程序中，使用一个巨大的 switch 语句来接收和处理消息；在 MFC 应用程序中，使用消息映射将特定的消息映射到相应的类的成员函数进行处理。

MFC 的消息映射机制使得大部分的 C++对象都可以处理消息，如应用程序框架、文档、视图等都可以接收和处理系统传递过来的消息。用户可以对一条消息或一系列消息、菜单命令、控件动作等进行消息映射。消息映射也提供了更新用户界面对象的途径，如菜单命令或工具条按钮的激活与取消，使得用户界面对象的状态与当前的内容相一致。

在 MFC 应用程序中，菜单命令、工具条按钮和加速键被称为“用户界面对象”，菜单命令提供了应用程序的全部或绝大部分的功能，而工具条按钮和加速键则提供了快速选择菜单命令的方法。因此，处理菜单命令和更新用户界面对象的状态是应用程序的基本任务。MFC 提供了 CMenu 类使得用户可以方便地使用和管理应用程序的菜单，同时还提供了菜单编辑器、工具条编辑器和加速键编辑器（都是资源编辑器的组件），使得用户可以“可视”方式来编辑程序的用户界面对象。

3.1 消息驱动机制

所有的 Windows 应用程序都是使用“消息驱动”机制。对所有的事件，如鼠标按钮被按下、键盘被敲击或窗口被移动等，Windows 系统都向相应的窗口发送消息通知事件的发生。MFC 框架应用程序与其他任何 Windows 程序一样，也使用消息驱动机制，但是 MFC 框架提供了一些方法，使得消息的处理更容易。

消息可以细分为三类：

（1）Windows 消息：主要包括以 WM_开头的消息，但 WM_COMMAND 除外。Windows 消息由窗口类或视图类进行处理。比如，如果用户改变了窗口的大小，则系统向应用程序发送 WM_SIZE 消息。

（2）控件通知消息：包括从控件或其他子窗口发送给其父窗口的 WM_COMMAND 消息。比如说，如果用户改变了文本框控件中的文本，该控件将向其父窗口发送包含着 EN_CHANGE 控件通知的 WM_COMMAND 消息。框架对控件通知消息的分配与其他以 WM_开头的 Windows 消息一样，但 BN_CLICKED 是个例外，该控件通知消息的分配与下述命令消息一样。

（3）命令消息：包括来自于用户界面对象的 WM_COMMAND 消息。框架对命令消息的分配与其他消息不太一样，命令消息可以被分配到更多的对象进行处理。

在 MFC 应用程序中，是看不见用来分配消息的 switch 语句的，实际上，该语句存在于 CWinApp 类的成员函数 Run()中。但是对用户而言，并不需要在 Run()函数里做工作，MFC 提供了消息映射来进行处理。在 MFC 应用程序中，每个可以接收 Windows 消息或命令的框架类都拥有自己的“消息映射”。应用程序框架使用消息映射将消息或命令与相应的处理函数联系起来，从 CCmdTarget 类派生出来的所有类都可以拥有消息映射，这包括应用程序类、主框架窗口类、文档类和视图类等等。

可以处理 Windows 消息和控件通知消息的是窗口类：从 CWnd 类派生。这包括 CFrameWnd 类、CMDIFrameWnd 类、CMDIChildWnd 类、CView 类和 CDialog 类，或者是用户自己从上述类中派生出来的子类。命令消息可以被更多类型的对象处理，这包括文档类、文档模板类、应用程序类、窗口类和视图类等。从感觉

上说，命令消息对哪个对象起了作用，就应该由该对象来处理该命令消息。

为了使用户对 MFC 应用程序的消息驱动机制有一初步的了解，便于更深入地介绍 MFC 的消息机制，先通过一个简单的例子程序介绍一下在 MFC 应用程序中是如何进行消息的映射的。

3.2 使用菜单工作

在 Windows 应用程序中，菜单的使用是很频繁的，处理来自菜单的命令消息是程序的一大任务，这里就先通过一个使用菜单命令的程序进行说明。

首先需要建立一个新的项目。在 File 菜单中选择 New 命令，在弹出的 New 对话框中选择 Projects 标签，选择程序类型为 MFC AppWizard(exe)，输入项目名为 TestMsg，单击 OK 按钮进入项目设置：

（1）选择单文档界面，选中文档/视图结构支持，并选择英语为资源语言类型。

（2）不需支持任何数据库功能，选择 None。

（3）不需支持任何复合文档，故选择 None；不要选中 Automation 与 ActiveX Controls 选项。

（4）保持缺省选项。

（5）保持缺省选择。

（6）保持缺省状况，单击 Finish 完成项目设置。

用户可以试着编译并运行现在的程序以检查程序的基本功能。

下面的目标是向 TestMsg 程序中添加一个下拉菜单 Message，并包括不同的菜单项，选择某个菜单命令将在程序窗口中输出相应的字符串表明选择了该命令。已经完成的程序的 Message 菜单如图 3-1 所示，请读者注意程序中已经下拉的 Message 菜单项。

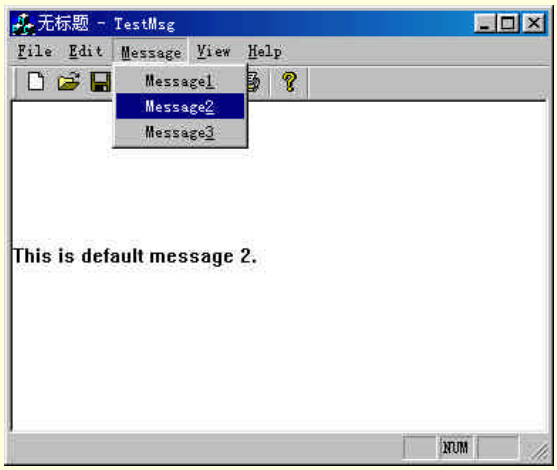

图 3-1 已经完成的程序的 Message 菜单

实现菜单工作可以分为以下几步：

（1）向程序中添加菜单资源

（2）映射菜单命令到相应的处理函数

（3）在处理函数中添加代码

下面将按照这一步骤引导用户实现最终的目标：

1. 修改菜单资源

在 ResourceView 中列出了本程序所拥有的所有资源，如图 3-2 所示。

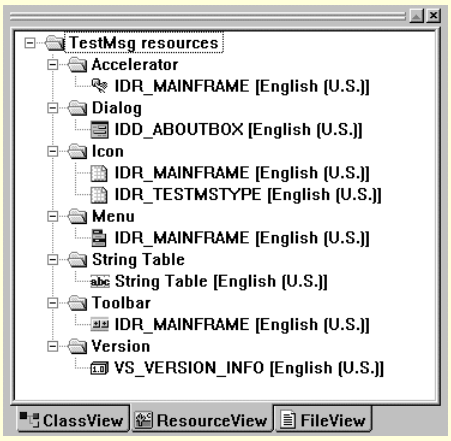

图 3-2 展开的 Resource View

修改菜单资源在菜单编辑器中完成。在菜单编辑器中，用户可以创建标准的菜单和菜单命令，可以将菜单或菜单命令从某处移动到另一位置，还可以同时编辑几个菜单的属性。在菜单编辑器中的菜单看起来和程序运行时的菜单几乎一模一样。

注意 Resource View 中 Menu 类型下的内容，这表明现在的项目中只有一项菜单资源，该菜单资源的标识符为 IDR_MAINFRAME。在标识符上双击鼠标，将自动在正文窗口中打开菜单资源编辑器，同时该菜单也显示在菜单资源编辑器中等待编辑。如图 3-3 所示。

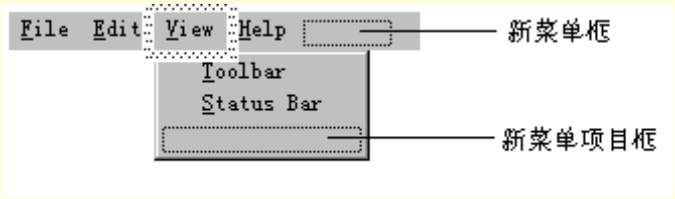

图 3-3 显示在菜单编辑器中的菜单资源

如图 3-3 所示的两个虚线框，在菜单条中的是新菜单框，如果要添加新的一组菜单，则在新菜单框中添加菜单的名称；在下拉菜单中的是新菜单项目框，在其中添加菜单的新项目的名称。在 View 菜单上有一个白色的虚框，表明当前的编辑对象。使用光标键可以移动该虚框到适当的位置；或者直接用鼠标点击欲编辑的菜单或菜单项，该虚框将同时移动到相应的位置。

TestMsg 程序需要在 Edit 菜单和 View 菜单之间添加新的 Message 菜单。为此，使用光标键将白色虚框移动到 View 菜单上，或直接单击 View 菜单。然后按下 Insert 键，这是新菜单框移到了 Edit 菜单和 View 菜单之间，如图 3-4 所示：

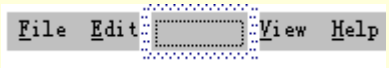

图 3-4 在两菜单之间添加新菜单

白色的虚框也包围在新菜单框周围，表明当前的编辑对象就是该新菜单。按 Enter 键或双击白色虚框，将显示 Menu Item Properties 对话框，如图 3-5 所示：

图 3-5 Menu Item Properties 对话框

在 Caption 框中输入&Message 作为菜单名，注意到输入的同时在菜单编辑器中的菜单也随着动态变化。输入菜单名时，在字母 M 的前面还有一个&符号，该符号的作用是在显示菜单时在其后面的字符的下面加上一短划，并将该字符与 Alt 键一起作为快速打开菜单的方式。如按 Alt+F 将打开 File 菜单，按 Alt+E 将打开 Edit 菜单等等，这里按 Alt+M 将打开 Message 菜单。输入完成后，按 Enter 键或在对话框之外的任意位置单击鼠标都将关闭该对话框。现在显示在菜单编辑器中的菜单如图 3-6 所示：

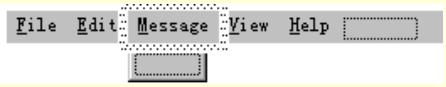

图 3-6 已添加 Message 菜单

接下来的工作是向 Message 菜单下添加菜单项 Default。与添加 Message 菜单的操作类似，先将白色虚框移动到新菜单项目框上，按 Enter 键或双击白色虚框将弹出 Menu Item Properties 对话框，现在要向 Message 菜单中添加菜单项 Message1。如图 3-7 所示：

图 3-7 Menu Item Properties 对话框

就像图 3-7 所示的那样，在 Caption 框中输入菜单项的名称 Message&1，在 ID 框中输入菜单项的标识符 ID_MESSAGE_MESSAGE1，在 Prompt 框中输入菜单项的提示信息 Show default message 1。按 Enter 键关闭 Menu Item Properties 对话框，完成添加菜单项的工作。

原则上，用户可以为自己的菜单项起任意名称的标识符，但是为了维护和修改程序的方便，一般使用 ID+菜单名+菜单项名的形式作为菜单项的标识符，所有的字母都大写，这是一个好的编程习惯。

在 Menu Item Properties 对话框中，还有一些选项，下面对这些选项做一些解释。

在 General 标签页面下：

- Separator：选中本选项表明将当前菜单项作为分隔条使用。
- Checked：选中本选项表明在最初显示菜单的时候在本菜单项前加核对记号。
- Pop-up：选中本选项表明将当前菜单项作为下拉菜单。
- Grayed：选中本选项表明在最初显示菜单的时候本菜单项呈灰色且无效。
- Inactive：如果 Grayed 选项被选中，本选项将自动被选中；否则选中本选项表明在最初显示菜单的时候本菜单项无效。
- Help：选中本选项将使得本菜单项在菜单条中右对齐。本选项对菜单才有意义，对菜单项没有作用。

另外，在 Extend Styles 标签页面下：Right-to-left order and alignment：选中本选项将使得菜单从右至左显示。在目前的 TestMsg 程序中，不需要修改菜单或菜单项的这些选项，保持缺省值即可。

在 Message 菜单下，还需要添加两个菜单项，表 3-1 中列出了菜单项的名称、标识符和提示信息。

表 3-1 Message 菜单的菜单项

名 称	标识符	提示信息
Message&1	ID_MESSAGE_MESSAGE1	Show default message 1.
Message&2	ID_MESSAGE_MESSAGE2	Show default message 2.
Message&3	ID_MESSAGE_MESSAGE3	Show default message 3.

添加完菜单资源后，可以编译和运行 TestMsg 程序，可以看到，在 TestMsg 程序的菜单条中确实出现了 Message 菜单，但其中的菜单项呈现灰色无效状态，不能选择，这是因为还没有给这些菜单项指定相应的消息处理函数。如图 3-8 所示：

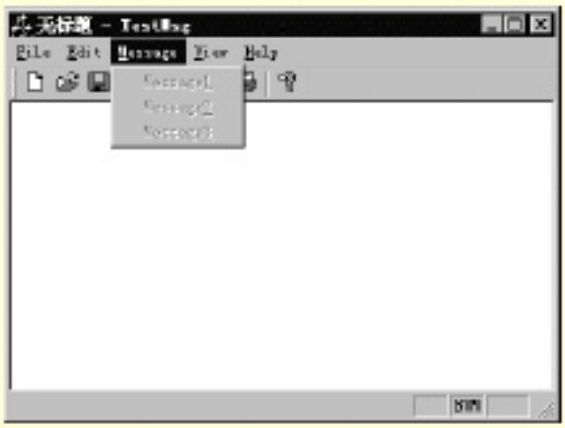

图 3-8 未建立消息映射的菜单项

下面就进行这一工作。在 ClassWizard 或 WizardBar 中都可以将菜单项与一定的函数之间建立映射。

2. 进行消息映射

一般使用 ClassWizard 进行消息映射，在 View 菜单中选择 ClassWizard，或者按 Ctrl+W 加速键可以打开 ClassWizard 对话框。如图 3-9 所示：

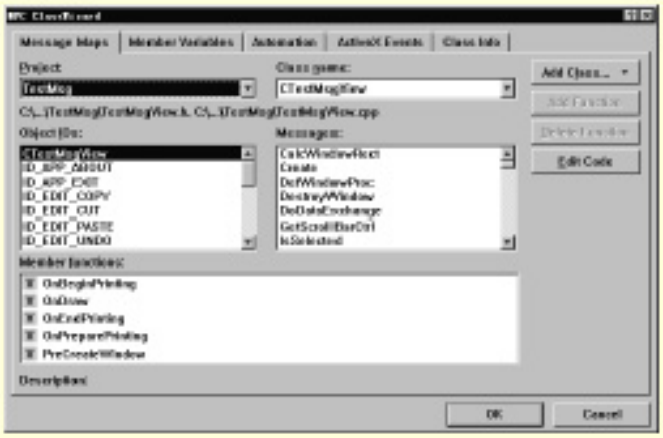

图 3-9 ClassWizard 对话框

ClassWizard 对话框中，在 Project 列表中选择待处理的项目，在 Class name 列表中选择接收消息的类。在 Object IDs 列表中显示了所有能产生消息的对象的标识符，包括菜单项、对话框控件等，列表的第一项一般是该类的类名；在 Messages 列表中显示了 Object IDs 列表中所选对象能产生的消息；在 Member functions 列表中列出了当前类已有的成员函数。

现在对 Message 菜单下的 Message1 菜单项产生的消息进行映射。在 Class name 列表中选择 CTestMsgView 类接收此条消息，并在 Object IDs 列表中选择该菜单项的标识符 ID_MESSAGE_MESSAGE1，则 Messages 列表中显示了两项内容，如图 3-10 所示：

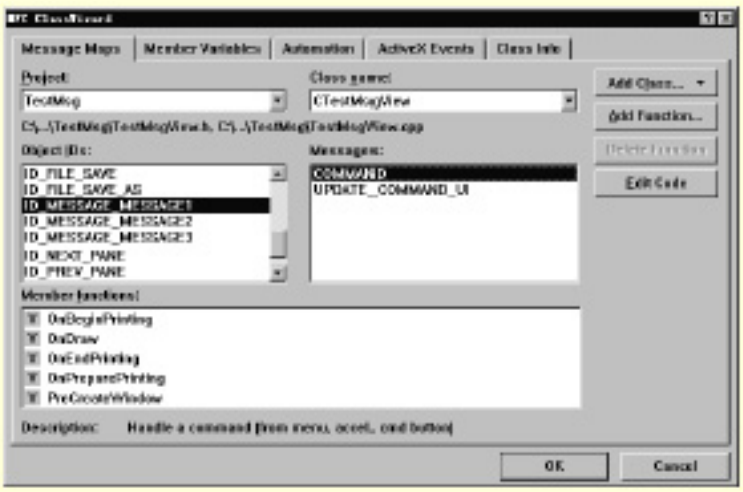

图 3-10 对 ID_MESSAGE_MESSAGE1 的命令进行映射

可以看到，在 Message 列表中有两条消息供选择，选择 COMMAND，此时 Add Functions 按钮变为有效，单击该按钮将弹出 Add Member Function 对话框询问用来处理所选消息的函数的名称，一般都接受 ClassWizard 自动产生的函数名，如图 3-11 所示：

图 3-11 Add Member Function 对话框

单击 OK 按钮，ClassWizard 就为 Message 菜单的 Message1 的 COMMAND 消息产生了消息映射，并为消息处理函数 OnMessageMessage1()产生了框架代码。

按上述方法，对 Message 菜单的其余菜单项也进行消息映射。

ClassWizard 所做的消息映射反映在 CTestMsgView 的源文件中，程序清单 3-1 中是文件中的消息映射的入口：

程序清单 3-1 消息映射入口

```
BEGIN_MESSAGE_MAP(CTestMsgView, CView)
//{{AFX_MSG_MAP(CTestMsgView)
ON_COMMAND(ID_MESSAGE_MESSAGE1, OnMessageMessage1)
ON_COMMAND(ID_MESSAGE_MESSAGE2, OnMessageMessage2)
ON_COMMAND(ID_MESSAGE_MESSAGE3, OnMessageMessage3)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
```

程序说明：

在 END_MESSAGE_MAP() 函数中，处于注释对“ //{{AFX_MSG_MAP(CTestMsgView) ”和“ //}}AFX_MSG_MAP ”中的部分是对 TestMsg 程序的 Message 菜单下 Default 菜单项的消息映射的入口。宏 ON_COMMAND()指明哪个函数对发自用户界面对象（通常是菜单项或工具条按钮）的消息进行处理，其第一

个参数指明为菜单项的标识符，第二个参数是用来处理命令消息的函数。只要该菜单项被选取，相应的消息处理函数就被调用，这种用宏将用户界面对象与相应的消息处理函数联系在一起的做法，比使用巨大 switch 语句来进行分配要简单的多。实际上，MFC 提供了好几个类似于 ON_COMMAND 的宏来进行各种分配工作。

在 CTestMsgView 源文件的末尾是 ClassWizard 产生的消息处理函数的框架代码，如程序清单 3-2 所示：

程序清单 3-2 消息处理函数的框架

```
void CTestMsgView::OnMessageMessage1()
{
// TODO: Add your command handler code here
}
void CTestMsgView::OnMessageMessage2()
{
// TODO: Add your command handler code here
}
void CTestMsgView::OnMessageMessage3()
{
// TODO: Add your command handler code here
}
```

下面向消息响应函数中添加代码，以实现程序的功能。

3. 添加代码

TestMsg 程序中消息处理函数要完成的工作是很简单的，只是在应用程序的窗口中显示一行字符串，用以表明选择的菜单项。

在 CTestMsgView::OnMessageMessage1()函数中，添加一小段代码，完成的函数代码列于程序清单 3-3 中：

程序清单 3-3 CTestMsgView::OnMessageMessage1()

```
void CTestMsgView::OnMessageMessage1()
{
// TODO: Add your command handler code here
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 1.");
}
```

在 OnMessageMessage1()函数中，首先创建了一个设备环境对象 dc，然后利用该设备环境对象的文本输出函数 TextOut()函数在窗口的左上角显示字符串“ This is default message ”，表明选择了 Message 菜单的 Message1 菜单项。

提示：

✍ 有关设备环境的概念和文本输出的内容将在第五章中详细介绍。

类似地，完成其他两个消息响应函数。完成的代码列于程序清单 3-4 中：

程序清单 3-4 OnMessageMessage2()和 OnMessageMessage3()

```
void CTestMsgView::OnMessageMessage2()
{
// TODO: Add your command handler code here
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 2.");
}
```

```
void CTestMsgView::OnMessageMessage3()
{
// TODO: Add your command handler code here
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 3.");
}
```

至此 TestMsg 程序的菜单已经能够正常工作，用户可以编译和运行该程序，检验菜单命令的作用。

TestMsg 程序本身是很简单的，但向 TestMsg 程序中添加菜单资源、进行消息映射和在消息处理函数中添加代码这几个步骤与所有具有菜单的 MFC 应用程序是一样的，TestMsg 程序已经足够说明在应用程序中使用菜单的全过程。

实际上，应用程序的菜单一般都随着程序的状态而有不同的变化，如某个菜单项的激活与无效、核对标记的添加与去除等，这正是下一节所要讨论的内容。

3.3 更新菜单状态

通常，菜单项都有不止一个状态。比如说，根据程序运行的情况，如果某个菜单项的功能不能被提供，那么，通常让该菜单项呈现灰色使之不能被选择；同样的理由，菜单项可能被添加核对记号，或者清除核对记号。

在 TestMsg 程序中，也可以具有这些功能。下面就来介绍如何在 TestMsg 程序中实现更新菜单状态的功能。

当应用程序的菜单被点中的时候，该菜单的每个菜单项都需要决定其显示状态。对菜单命令对象的 UPDATE_COMMAND_UI 消息进行处理可以为菜单项提供更新显示的消息。

在 3.1 节中对 ID_MESSAGE_MESSAGE1 产生的消息进行映射的时候，只处理了 COMMAND 消息，还有 UPDATE_COMMAND_UI 消息没有处理。如果要根据程序的运行情况动态更新菜单的状态，就需要处理 UPDATE_COMMAND_UI 消息。

在 ClassWizard 中添加 Message 菜单的三个菜单项的 UPDATE_COMMAND_UI 消息的处理函数，并接受 ClassWizard 提供的函数名。ClassWizard 用 OnUpdate***()表明是更新用户界面对象状态的函数。在 CTestMsgView 的源文件中会增加对以上消息的映射入口，现在的消息映射入口如程序清单 3-5 所示：

程序清单 3-5 消息映射入口

```
BEGIN_MESSAGE_MAP(CTestMsgView, CView)
//{{AFX_MSG_MAP(CTestMsgView)
ON_COMMAND(ID_MESSAGE_MESSAGE1, OnMessageMessage1)
ON_COMMAND(ID_MESSAGE_MESSAGE2, OnMessageMessage2)
ON_COMMAND(ID_MESSAGE_MESSAGE3, OnMessageMessage3)
ON_UPDATE_COMMAND_UI(ID_MESSAGE_MESSAGE1, OnUpdateMessageMessage1)
ON_UPDATE_COMMAND_UI(ID_MESSAGE_MESSAGE2, OnUpdateMessageMessage2)
ON_UPDATE_COMMAND_UI(ID_MESSAGE_MESSAGE3, OnUpdateMessageMessage3)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

程序说明：

在这一段消息映射入口中，ON_UPDATE_COMMAND_UI 宏将用户界面对象更新消息和相应的消息处理函数联系在一起。

ClassWizard 还产生了相应的函数的框架代码，如程序清单 3-6 所示：

程序清单 3-6　消息处理函数的框架

```
void CTestMsgView::OnUpdateMessageMessage1(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
}
void CTestMsgView::OnUpdateMessageMessage2(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
}
void CTestMsgView::OnUpdateMessageMessage3(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
}
```

OnUpdate***()函数的参数 CCmdUI* pCmdUI，这是一个指向 CCmdUI 对象的指针，该指针即代表了需要更新的用户界面对象。CCmdUI 类提供了一些成员函数来更新相应的用户界面对象，这些函数列于表 3-2 中。

表 3-2　CCmdUI 类的成员函数

成员函数	说　明
Enable()	激活相应的用户界面对象或使其无效
SetCheck()	为用户界面对象添加核对标记或取消该核对标记
SetRadio()	为一组用户界面对象添加或取消核对标记
SetText()	设置用户界面对象的文本

在 TestMsg 程序中，使用 SetCheck()函数在 Message 菜单的菜单项前添加核对标记。SetCheck()函数的原型如下：

```
virtual void SetCheck( int nCheck = 1 );
```

参数 nCheck 如果取 0，则取消用户界面对象前的核对标记；如果取非 0 值，则在用户界面对象前添加核对标记。

考虑 Message 菜单下的 Message1 菜单项，如果当前的状态是没有核对标记，选择该菜单项一方面意味着在应用程序窗口中输出对应的字符串，另一方面意味着在该菜单项前添加核对标记，同时取消 Message2 菜单项或 Message3 菜单项前的核对标记，因为同一时刻应该只有一个 MessageX 是作用的。

为了记录各菜单项的状态，需要向程序中添加几个变量。这里在 CTestMsgView 类中增加三个成员变量，列于表 3-3 中：

表 3-3　向 CTestMsgView 类添加成员变量

变量名	类　型	访问级别
m_bMessage1	BOOL	protected
m_bMessage2	BOOL	protected
m_bMessage3	BOOL	protected

这三个成员变量是互斥的，即任一时刻只能有一个变量为 true。比如说，当用户选择了菜单项 Message1 的时候，就需要将 m_bMessage1 变量设置为 true，其余的 m_bMessage2 和 m_bMessage3 设置为 false 选择 Message2 菜单项或 Message3 菜单项时类似。为此需要在 OnMessageX()函数中添加相应的代码，同时在 CTestMsgView 的构造函数中需要初始化这些标志变量。程序清单 3-7 中是已经修改过的函数的代码：

程序清单 3-7　设置标志变量

```
CTestMsgView::CTestMsgView()
{
```

```
// TODO: add construction code here
m_bMessage1=false;
m_bMessage2=false;
m_bMessage3=false;
}

void CTestMsgView::OnMessageMessage1()
{
// TODO: Add your command handler code here
m_bMessage1=true;
m_bMessage2=false;
m_bMessage3=false;
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 1.");
}

void CTestMsgView::OnMessageMessage2()
{
// TODO: Add your command handler code here
m_bMessage1=false;
m_bMessage2=true;
m_bMessage3=false;
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 2.");
}

void CTestMsgView::OnMessageMessage3()
{
// TODO: Add your command handler code here
m_bMessage1=false;
m_bMessage2=false;
m_bMessage3=true;
CClientDC dc(this);
dc.TextOut(0,100,"This is default message 3.");
}
```

在 OnUpdateMessageX()**函数中，只需要调用** SetCheck()**函数即可完成工作，程序清单** 3-8 **中是完成后的代码：**

程序清单 3-8　OnUpdateMessageX()

```
void CTestMsgView::OnUpdateMessageMessage1(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bMessage1);
}
void CTestMsgView::OnUpdateMessageMessage2(CCmdUI* pCmdUI)
```

```
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bMessage2);
}
void CTestMsgView::OnUpdateMessageMessage3(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bMessage3);
}
```

添加完上述代码后，用户可以再次编译和运行 TestMsg 程序，可以看到：

一旦用户选择了 Message 菜单的菜单项，程序一方面在客户区中显示相应的字符串，另一方面对相应的菜单项加上了核对标记。

运行中的程序如图 3-12 所示：

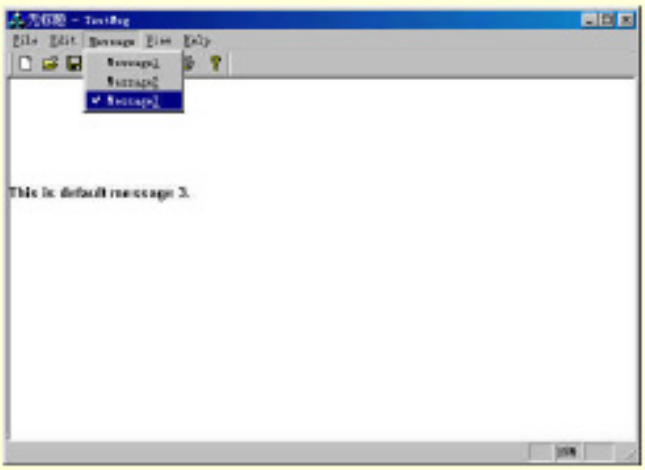

图 3-12 运行中 TestMsg 的程序

Enable()函数的使用与 SetCheck()函数的使用是类似的，用户可以自己进行试验。

现在用户已经清楚如何使用菜单驱动程序工作、如何更新菜单项的状态，下面将介绍如果使用用户界面对象的另外两种：工具条按钮和加速键。

3.4 使用工具条

在 Windows 应用程序中，工具条的使用也是很普遍的，使用工具条的按钮可以直观、方便地执行程序功能，实现与选择菜单同样的效果。

在利用 AppWizard 生成应用程序时，如果在步骤 4 中选择了 Toobar 选项，生成的程序就拥有一个缺省的工具条，如图 3-13 所示：

图 3-13 缺省工具条

在缺省的工具条中，提供了常用的文件和编辑命令。下面将介绍如何对程序的缺省工具条进行修改，使之符合用户自己的需要。

注意：

- 与菜单的使用相似，如果要使用工具条，首先需要修改工具条资源。资源编辑器的组件之一是工

具条编辑器，用它可完成修改工具条资源的工作。

在 ResourceView 中同样显示了程序已经拥有的工具条资源（参见图 3-2），其标识符为 IDR_MAINFRAME。双击该标识符可以打开工具条编辑器，并同时在正文窗口中显示对应的工具条供编辑，如图 3-14 所示：

图 3-14 工具条编辑器

工具条编辑器简直就是一个小型的画笔程序，在绘图工具条中提供了许多绘图工具，使用这些工具可以方便地修改工具条资源。绘图工具的简要说明如图 3-15 所示：

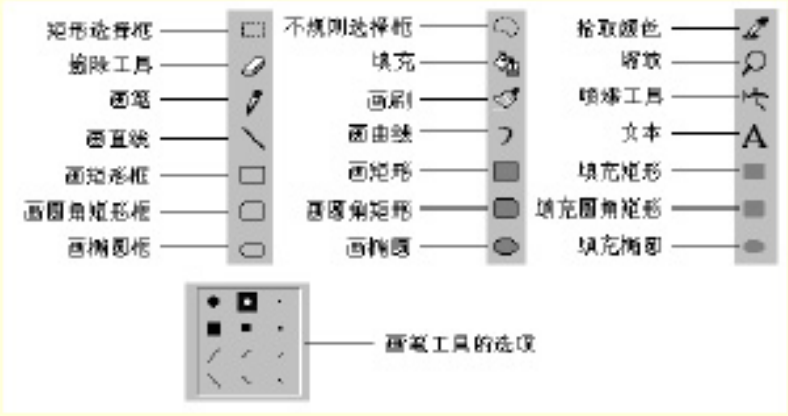

图 3-15 绘图工具说明

在工具条编辑器处于活动状态时，在 Visual C++的集成开发环境的菜单栏中也新增了 Image 菜单，如图 3-16 所示：

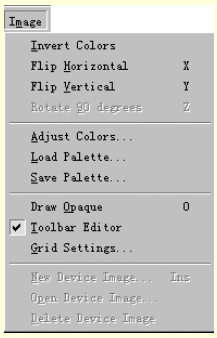

图 3-16 Image 菜单

Image 菜单中各菜单项的说明列于表 3-4 中：

表 3-4 Image 菜单项说明

菜单命令	说 明
Invert Colors	反转颜色
Flip Horizontal	水平翻转
Flip Vertical	垂直翻转
Rotate 90 degrees	旋转 90 度
Adjust Colors	调整颜色
Load Palette	加载调色板
Save Palette	存储调色板
Draw Opaque	不透明方式绘图
Toolbar Editor	工具条编辑器
Grid Settings	网格设置
New Device Image	插入新图象
Open Device Image	打开原有图象
Delete Device Image	删除图象

在 TestMsg 程序中，缺省的工具条提供的按钮都不需要，删除它们的方法很简单：将按钮拖曳出工具条的范围即删除了该按钮。试图建立的新的 TestMsg 程序的工具条如图 3-17 所示：

图 3-17 试图建立的新工具条

在删除完所有缺省的工具条按钮后，新工具条按钮框自动成为当前编辑按钮。利用工具条编辑器的文本工具可以很容易地在按钮上绘上字符“ 1 ”，这就完成了一个新的按钮。注意到一旦用户对新按钮框做了任何修改，在工具条预览中立即反映出该变化，并且在按钮的右边出现新按钮框。按创建按钮“ 1 ”的方法，创建按钮“ 2 ”和“ 3 ”。 用户可能需要更漂亮的按钮来达到美观的效果，但 TestMsg 程序用这样的按钮已经足够说明工具条的使用了。

重要的在于指定按钮的标识符。双击按钮将打开该按钮的属性对话框，图 3-18 中所示为按钮“ 1 ”的属性对话框：

图 3-18 按钮的属性对话框

在 ID 框中输入按钮“ 1 ”的标识符，或者通过下拉箭头打开已存在的标识符列表选择相应的标识符，这里输入的是 ID_MESSAGE_MESSAGE1，注意该标识符与 Message 菜单下的 Message1 菜单项的标识符相同。这是因为，在 MFC 应用程序中，如果将工具条按钮指定为与某个菜单项具有相同的标识符，就意味着该按钮与该菜单项共享相同的消息处理函数，也就是说，点击按钮与选择菜单项的作用相同，这也正是工具条的作用所在：提供快速选择菜单项的功能。输入完标识符后，将输入焦点移动到 Prompt 框中，此时该框中已经自动出现已经在菜单资源中定义过的提示信息：“ Show default message1 ”，现在在已有的字符串后再加上“ \nMessage 1 ”。在程序运行的时候，“ \n ”前的字符串作为提示信息显示在状态栏中，而“ \n ”后的字符串则作为工具条按钮的提示（Tooltip）显示。这样在指定工具条按钮的提示信息的同时就很容易地实现了按钮提示这一功能。

对按钮“ 2 ”与按钮“ 3 ”做类似的属性指定，ID 分别为 ID_MESSAGE_MESSAGE2 和 ID_MESSAGE_MESSAGE3，提示信息分别为“ Show default message 2.\nMessage 2 ”和“ Show default message 3.\nMessage 3 ”。这样就将工具条上三个按钮与 Message 菜单中的三个菜单项联系在了一起。

现在用户可以再次编译和运行 TestMsg 程序，以检验刚才所做的修改。图 3-19 中显示了运行中的 TestMsg 程序：

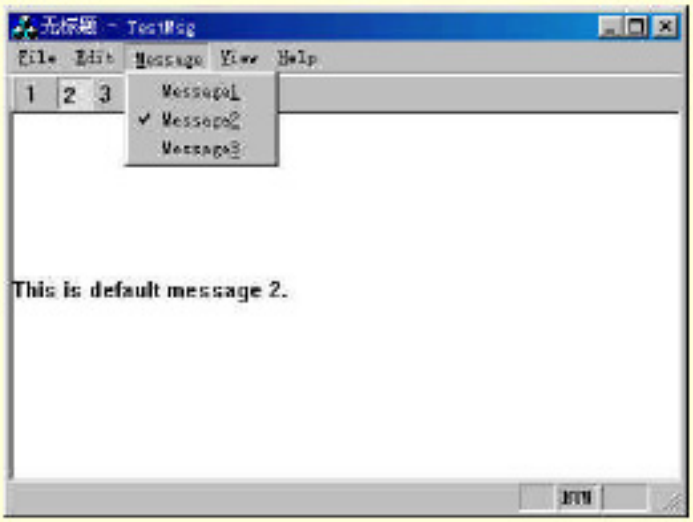

图 3-19 运行中的 TestMsg 程序

注意：

1. 上图中，Message 菜单下的 Message1 菜单项前有核对标记，同时工具条中的按钮“1”处于下陷状态，这是 SetCheck()函数的作用：对菜单项加核对标记，对工具条按钮则以下陷方式显示。
2. 用户还可以实验以下将鼠标在工具条按钮上稍做停留，观察出现的按钮的提示信息。

3.5 使用加速键

同工具条一样，加速键提供了快速选择菜单命令的另一种方式。在 TestMsg 程序中，也计划使用加速键。使用加速键要修改两类资源：菜单资源和加速键资源。

修改菜单资源主要是在菜单项的后面加上相应的加速键提示，图 3-20 显示了未加加速键提示的 Message 菜单和加了加速键提示的 Message 菜单的提示：

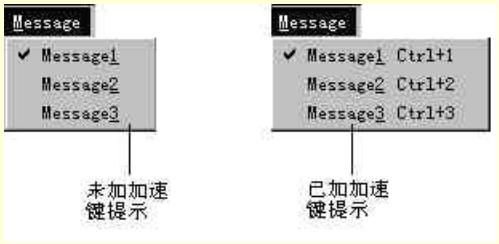

图 3-20 前后菜单的对比

要在菜单项的后面加上相应的加速键的提示，只需要在菜单项的 Caption 字符串后添加\t+加速键组合即可。具体来说，对 Message1 菜单项，需要将 Caption 字符串由“Message&1”改为“Message&1\tCtrl+1”，其中的“\t”表示在显示菜单时将该符号的字符串靠右对齐。类似地，修改 Message2 和 Message3 菜单项。

仅仅修改菜单项的显示是不够的，真正的使加速键工作还需要修改加速键资源。在 ResourceView 的 Accelerator 下列出了本程序已经拥有的加速键资源 ID_MAINFRAME，双击该标识符打开加速键资源编辑器，如图 3-21 所示：

ID_EDIT_COPY	Ctrl + C	VIRTKEY
ID_FILE_NEW	Ctrl + N	VIRTKEY
ID_FILE_OPEN	Ctrl + O	VIRTKEY
ID_FILE_PRINT	Ctrl + P	VIRTKEY
ID_FILE_SAVE	Ctrl + S	VIRTKEY
ID_EDIT_PASTE	Ctrl + V	VIRTKEY
ID_EDIT_UNDO	Alt + VK_BACK	VIRTKEY
ID_EDIT_CUT	Shift + VK_DELETE	VIRTKEY
ID_NEXT_PANE	VK_F6	VIRTKEY
ID_PREV_PANE	Shift + VK_F6	VIRTKEY
ID_EDIT_COPY	Ctrl + VK_INSERT	VIRTKEY
ID_EDIT_PASTE	Shift + VK_INSERT	VIRTKEY
ID_EDIT_CUT	Ctrl + X	VIRTKEY
ID_EDIT_UNDO	Ctrl + Z	VIRTKEY

图 3-21 加速键资源编辑器

在加速键资源编辑器中，列出本程序已经拥有的加速键资源，如 Ctrl+C、Ctrl+X、Ctrl+V 等熟悉的编辑命令的加速键。最下端的虚框是新加速键框，双击该框将弹出加速键属性对话框，如图 3-22 所示：

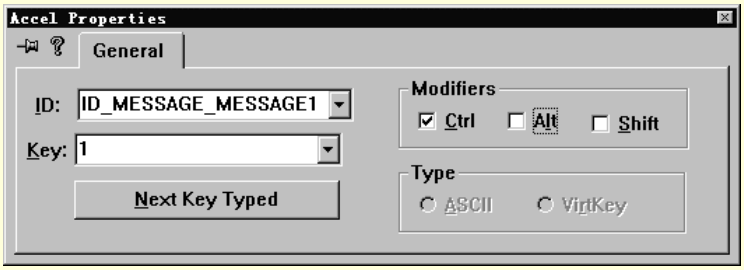

图 3-22 加速键属性对话框

在 ID 框内输入新的加速键的标识符，这里输入“ ID_MESSAGE_MESSAGE1 ”，与工具条按钮同样的道理，如果加速键使用与某个菜单项相同的标识符，则加速键与菜单项使用相同的消息处理函数。在 Key 框中输入基本键，这里输入“ 1 ”；在 Modifiers 下选择修饰键，这里保持缺省状态，即选中“ Ctrl ”作为修饰键。这意味着为 ID_MESSAGE_MESSAGE1 设置的加速键为 Ctrl+1。类似地为 ID_MESSAGE_MESSAGE2 设置加速键 Ctrl+2，为 ID_MESSAGE_MESSAGE3 设置加速键 Ctrl+3。

原则上，可以使用 Ctrl、Alt 和 Shift 键中的任一键或任一组合作为修饰键，但是由于系统缺省地将 Alt 键用来与字符组合形成打开菜单的加速键，因此，一般不单独使用 Alt 键作为修饰键，但可以使用 Ctrl+Alt 或 Alt+Shift 组合作为加速键的修饰键。

在完成加速键资源的编辑后，用户可以重新编译和运行 TestMsg 程序，以检验加速键的作用。

以上完成了完整的 TestMsg 程序，在 TestMsg 程序中，使用了菜单、工具条和加速键，也由此向用户介绍了用户界面对象的基本知识。在以后的实践中，还将不断地涉及到这些内容。

3.6 消息的传递

在本章第一节中，简单介绍了 MFC 中的消息映射机制，然而，在 MFC 应用程序中，消息的传递途径是怎样的呢？弄清楚这个问题，对于决定由程序的哪个对象来处理特定的消息是很重要的。

Windows 消息通常都是传递给主框架窗口，但是命令消息将传递给其他对象。应用程序框架按一定的次序在命令目标对象中传递命令消息，这些对象中的某一个应该拥有对该消息的处理函数，每个命令目标对象都检查其消息映射是否能处理传递过来的命令消息。这里所谓的命令目标对象，就是指那些可以处理命令消息的对象。

不同的命令目标对象在不同的时刻检查其消息映射。通常情况下，收到命令消息的某个类会先将该命令消息传递给其他一些特定的类，使那些拥有最先处理该命令消息的机会。如果那些对象都没有处理该命令消息，

该类将检查其自身的消息映射。如果它自身也没有对该命令消息进行处理，它将该消息传递给更多的命令目标。表 3-5 显示了每个类是怎样建立这一次序的。

表 3-5 命令传递途径

收到消息的类	传递途径
MDI 框架窗口（CMDIFrameWnd）	当前活动的 MDI 子窗口类 本框架窗口 应用程序（CWinApp 对象）
文档框架窗口（CFrameWnd,CMDIChildWnd）	当前活动视图 本框架窗口 应用程序（CWinApp 对象）
视图	本视图 与视图相联的文档
文档	本文档 与文档相联的文档模板
对话框	本对话框 对话框的父窗口 应用程序（CWinApp 对象）

在上表中，如果右边列中涉及到其他对象，比如说文档，应从左边列中查看相应的部分来跟踪命令消息的传递途径。举个例子，如果视图类接收到命令消息，首先查看它自己的消息映射，如果没有相应的处理函数，便将该命令消息传递给与之相联的文档，文档类则首先查看自己的消息映射，然后将消息传递给相关的文档模板，如此继续下去。

在消息的传递过程中，如果某个类拥有对该消息的处理函数，则消息传递至此终止，由该类来处理消息。

对某一条特定的消息而言，选择哪个类来处理该消息也是一个很重要的问题。一般说来，该消息对哪个类影响最大，就由该类负责处理该消息；如果某消息影响多个视图或框架窗口共享，则有文档类来处理；如果没有共享的需要，则由视图或文档来处理。当然，这并不是绝对适用的标准，具体到某个应用程序中，应该由具体的情况来决定如何选择处理消息的类，这需要用户在使用 MFC 编程的时候自行体会。

3.7 本章小结

本章中介绍了 MFC 应用程序的消息驱动机制和程序运行中消息的传递过程，并通过 TestMsg 程序详细介绍了用户界面对象（包括菜单、工具条和加速键）的使用，希望读者熟练掌握以下的几个知识点：

- 理解消息驱动机制和消息的传递途径
- 学习使用菜单工作和更新菜单状态
- 使用工具条工作
- 使用加速键工作

最后需要说明的是，有关鼠标和键盘消息在第五章、第六章中结合具体例子有详细介绍，下面让我们开始下一章的学习。

第四章 对话框和控件的使用

在 Windows 应用程序中，对话框是应用程序接收用户数据最主要的渠道之一。使用对话框，用户可以完成打开文件、选择字体、改变应用程序的参数、向应用程序提供数据等多种功能，这其中的某些功能可能是其他方法无法实现或难以实现的。当然，在 Visual C++的集成开发环境中，也有许多地方用到了对话框。一般地，在 Windows 应用程序中，如果某个菜单项后带有三个点的省略号（如“ File ”菜单下的“ New ”命令），选择该菜单项将弹出一个对话框。

在应用程序的对话框中，通常都包含了一系列的控件。每个控件通常都是一个小的窗口，能够完成一些基本的交互任务。除了在对话框中使用控件外，用户可以在任何需要的时候在其他窗口中创建和使用控件。

在本章中，将结合实例介绍有关对话框与控件的内容。

4.1 有模式对话框

在 Windows 应用程序中，对话框可以分为两类：

- 有模式对话框（Modal Dialog）：对话框始终位于应用程序的最顶层，在对话框被关闭之前，用户不能选择应用程序的其他功能。应用程序的大部分对话框都是有模式对话框。
- 无模式对话框（Modaless Dialog）：在对话框被关闭之前，用户可以选择应用程序的其他功能。如“ Word ”字处理软件中的“ Find ”对话框就是一个典型的无模式对话框。

在本节中，将通过“ CompuInfo ”例子程序，向用户介绍如何在应用程序中使用有模式对话框。这包括建立对话框资源、建立对话框类、设置对话框类的成员变量和通过代码调用对话框、与对话框交换数据等内容。

4.1.1 建立新项目

本节我们计划建立的项目为“ CompuInfo ”，此程序将拥有一个“ Computer Information ”对话框，用户可以在对话框中输入计算机的有关信息，在单击“ OK ”按钮关闭该对话框后，程序将在视图的客户区中显示用户在对话框中输入的信息。

在“ File ”菜单下选择“ New ”命令，建立新的 MFC 应用程序项目：CompuInfo。在 AppWizard 中，在步骤一中选择“ Single Document（单文档界面）”的程序类型，选择“ 英语（美国）”为应用程序资源的语言类型；在步骤二中可以保持缺省设置；在步骤三中清除“ ActiveX Controls ”选项；在步骤四、五、六中保持缺省设置。

最后单击 AppWizard 的“ Finish ”按钮完成应用程序的设置，并生成“ CompuInfo ”程序的框架代码。

4.1.2 建立对话框资源

在 MFC 应用程序中，一个对话框包含了两方面的内容：

- 一个指定了对话框中各控件及其相对位置的对话框模板资源。

在应用程序的运行过程中需要显示该对话框的时候，Windows 系统将根据该资源建立对话框窗口并显示出来。在模板资源中需要定义对话框的各种特征，包括对话框的大小、位置、风格，以及对话框中各控件的类型和相对位置。在 Visual C++集成开发环境中，用户可以使用对话框编辑器方便地建立应用程序的对话框资源。当然，如果有必要，用户也可以在内存中建立对话框模板资源。

- 一个对话框类，通常都从 CDialog 类派生。

该类提供了应用程序中操作该对话框的接口。在用户成功地建立了对话框资源后，ClassWizard 可以让用户方便地根据对话框资源建立一个相应的类，并为对话框添加成员变量。这些成员变量大多与对话框内控件有关，

通过这些成员变量，一方面可以控制相应控件的状态，另一方面可以与应用程序的其他类进行数据交换。

因此，在应用程序中使用对话框就要完成两方面的任务：建立对话框资源和建立对话框类。本小节中将介绍使用对话框编辑器建立对话框资源的方法。

1．建立一个新的对话框资源

使用 VC6.0 提供的对话框编辑器，可以调整对话框的大小和位置；可以从控件工具箱中选择不同类型的控件并添加到对话框资源中；可以设置控件的属性和相对位置；还可以模拟应用程序运行时显示对话框。

在用户完成对话框资源的编辑后，该对话框资源将被保存在应用程序的资源文件中，用户可以在需要的时候再次进行修改和编辑。

在应用程序的项目工作台中可以浏览应用程序已经拥有的所有资源，AppWizard 为“CompuInfo”程序生成的缺省的对话框是“About CompuInfo”对话框，命令 ID 为 IDD_ABOUTBOX。如果用户在“CompuInfo”程序的“About”菜单下选择“About CompuInfo”命令，将弹出该对话框，如图 4-1 所示。

图 4-1 “About CompuInfo”对话框

用户可以对该对话框进行修改，以显示有关本应用程序的一些信息，如版权信息、开发人员列表等等。

在“CompuInfo”应用程序中，主要的目的是建立一个新的“Computer Info”对话框。为此在 VC6.0 集成开发环境的“Insert”菜单下选择“Resource”命令，弹出“Insert Resource”对话框。在“Insert Resource”对话框中选择“Dialog”类型，单击“New”按钮建立一个新的对话框资源，如图 4-2 所示。

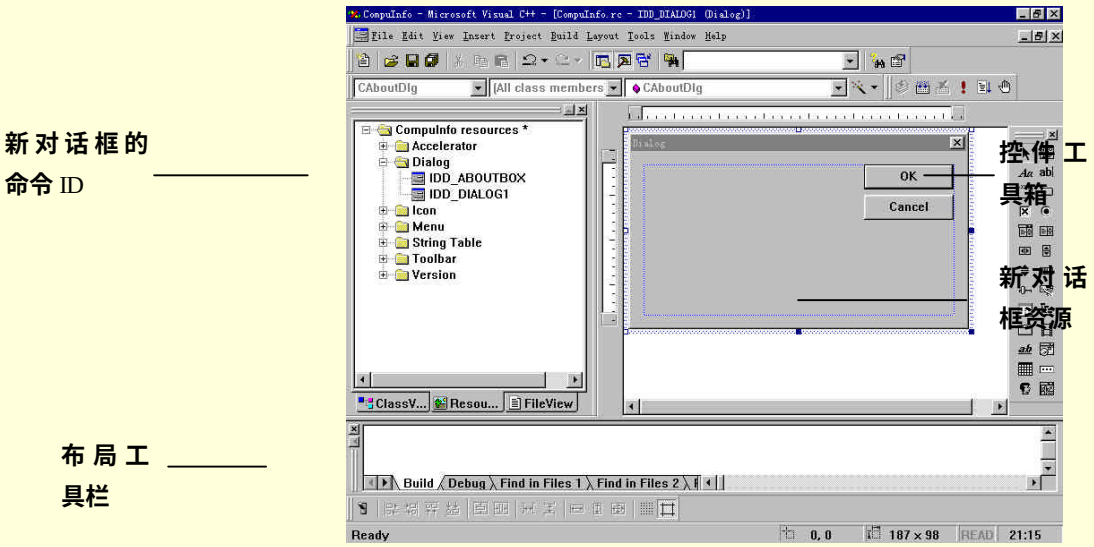

图 4-2 新的对话框资源

在图 4-2 中可以看到，新的对话框资源出现在 VC6.0 的集成开发环境的工作区中，同时显示的还有控件工具箱、布局工具栏和‘Layout’菜单。控件工具箱中列出了当前可以使用的控件的类型，而布局工具栏与‘Layout’菜单相对应，提供了一些用来对控件进行对齐、均布和尺寸分配等功能。这些都为用户在满足应用程序功能的前提下对对话框进行美化提供了方便。

在向对话框添加控件之前，需要设置新对话框的属性。在新对话框的空白处单击鼠标右键（注意不要在控件上单击），在弹出的快捷菜单中选择“Properties”命令，打开“Dialog Properties”对话框，如图 4-3 所示。

图 4-3 “ Dialog Properties ”对话框

在“ Dialog Properties ”对话框中，需要将新对话框的命令 ID 修改为 IDD_COMPUINFO,将“ Caption ”框中的字符串改为“ Computer Information ”，该字符串将在对话框的标题栏中作为标题文本而出现。

除了需要修改的内容外，“ Dialog Properties ”对话框中还提供了许多选项，用户通过选择这些选项就可以很方便地获得一些重要功能，或许这其中的某项功能正好是用户所需要的，而如果由用户自己来实现可能是一件很困难的事情。这些选项对本书来说完全没有必要作详细、全面的介绍，用户可以从联机帮助中获得详细的解释。

2. 认识各种控件

在控件工具箱中用户可以看到许多种类型的控件，这些控件在 Windows 应用程序中得到了广泛的应用。这里对这些控件做一简单介绍：

- 静态文本（Static Text）：用来在指定的位置显示特定的字符串，一般用来标识附近另一控件的内容。显示在静态文本控件中的字符串一般不再改变，但是在需要的时候，也可以通过调用相应的函数进行设置。MFC 提供了 CStatic 类支持静态控件。
- 编辑框（Edit Box）：用来接收用户输入的字符串。通过选择编辑框的选项，编辑框可以接收字符串、数字、密码等等；编辑框还可以设置成接收多行字符串的模式；可以自动进行大小写转换。编辑框可能向其父窗口发送多种控件通知，如果用户需要，可以对这些控件通知进行处理。MFC 提供了 CEdit 类支持编辑框控件。
- 成组框（Group Box）：用来包围具有逻辑关系的一组控件，在这些控件的周围加上边界和标题。但需要注意，成组框仅仅是在视觉效果上对控件进行“ 成组 ”，真正的“ 成组 ”工作还需要另外的一些工作。
- 按钮（Button）：用来接收用户的命令，应用程序在接收到用户命令后，通常需要进行一些后台的工作。按钮可以响应单击或双击动作，在按钮接收到鼠标动作后，向其父窗口发送相应的控件通知，用户可以对这些控件通知进行消息映射，从而进行相应的处理。在一个对话框中，可以定义一个缺省按钮，这只要选中按钮属性中的“ Default ”选项。如果在对话框活动的时候按下了 Enter 键，则等同于单击了缺省按钮。MFC 提供了 CBotton 类支持按钮控件。
- 复选框（Check Box）：用来显示某种可能的选择，该项选择是独立的，用户可以选中或取消该选项。在选项被选中的时候核选标记出现，选项被取消时核选标记消失。在 MFC 中由 CButton 类对复选框进行支持，用户可以通过 SetCheck()函数和 GetCheck()函数设置或获取复选框当前的状态。
- 单选按钮（Radio Button）：用来选择某种可能的选择，与复选框不同，该选项不是独立的。一般的情况是，几个单选按钮组成一组，同组中的单选按钮可以有也只能有一个按钮被选中。MFC 同样使用 CButton 类对单选按钮控件进行支持，SetCheck()函数和 GetCheck()函数对单选按钮也是适用的。
- 列表框（List Box）：用来选择一系列的可能选择，用户通过滚动条可以在这些选择中浏览。在列表框中，可以进行单项选择，也可以进行多项选择，这取决于用户在控件属性对话框中的设置。MFC 提供了 CListBox 类对列表框控件进行支持。
- 组合框（Combo Box）：是列表框和编辑框的组合，用户除了可以在列表中对已经存在的选项进行选择外，还可以输入新的选择。MFC 提供了 CComboBox 类对组合框控件进行支持。
- 滚动条（Scroll Bar）：这包括水平滚动条和垂直滚动条，除了在视觉效果上的方向不同外，水平滚动条在被滚动时发送 WM_HSCROLL 消息，而垂直滚动条在被滚动时发送 WM_VSCROLL 消息。MFC 提供了 CScrollBar 进行支持。

- 微调按钮（Spin Button）：包括一对紧靠在一起的上下箭头，使用微调按钮可以增大或者减小某个特定的数值。微调按钮往往都需要一个“伙伴”控件，这通常都是一个编辑框。当微调按钮的向上箭头被单击时，编辑框中的数字就增大；反之则减小。MFC 提供了 CSpinButtonCtrl 类进行支持。
- 进度条（Progress）：在进行一项需要占用较长时间的操作时用来反应当前的进度。当操作的进度不断前进时，进度条就用特定颜色填充进度条框。用户可以设置进度条的范围和当前位置。MFC 提供了 CProgressCtrl 类进行支持。
- 滑块控件（Slider）：通常用来在程序中接受一系列离散的数值。用户可以设置滑块控件的取值范围，并可以为控件加上刻度标记以显示特定位置的含义。MFC 提供了 CSliderCtrl 类进行支持。
- 热键控件（Hot Key）：热键控件看起来就像一个编辑框，但是在热键控件中能够立刻反应用户刚刚按下的键组合，这在设置程序的热键时特别有用。当然，热键控件只是在“视觉”上显示了按键组合，设置热键的工作还需要用户添加代码完成。MFC 提供了 CHotKey 类进行支持。
- 列表控件（List Control）：按一定的排列顺序显示一系列带图标的字符串。列表控件提供了四种显示模式：大图标、小图标、列表和详细信息。用户可以向列表控件中添加新的项，也可以控制列表控件的显示模式。MFC 提供了 CListCtrl 类进行支持。
- 树形控件（Tree Control）：用来显示一系列项目的层次关系，最典型的例子就是显示磁盘上的文件与文件夹。如果有子项目的话，单击树形控件中的项目可以展开或者收缩其子项目。MFC 提供了 CTreeCtrl 类进行支持。
- 属性表控件（Tab Control）：属性表控件用来包含大量的控件，可以满足用户显示或者获取大量数据的要求。每个属性表又分为好几个属性页，这些属性页由各自的标签进行区分，这些属性页中都可以包容其他控件。在显示属性表的时候，一次只能够显示一个属性页的全部内容，同时显示其他属性页的标签，用户通过单击标签打开相应的属性页。MFC 提供了 CTabCtrl 类进行支持。
- 动画控件（Animation）：用来播放一段 AVI 格式的视频剪辑。用户可以控制视频剪辑的播放、停止和定位，但也仅限于这些功能。动画控件甚至不能播放音频剪辑，如果用户需要更高层次的视频或音频支持，请选用 MCIWnd 控件。MFC 提供了 CAnimateCtrl 类对动画控件进行支持。
- 高级编辑框（Rich Edit）：很显然，高级编辑框是编辑框控件功能的扩展。在高级编辑框中，除了简单地输入和编辑字符串外，用户还可以为字符或段落指定特定的格式，用户甚至还可以向高级编辑框中插入 OLE 项。高级编辑框基本上实现了一个带格式的文本编辑器的功能，而只需要用户添加少量的接口。MFC 提供了 CRichEditCtrl 类进行支持。
- 日历控件（Month Calender）：这是 VC6.0 新提供的控件类型。日历控件看起来与一个真正的日历很相似，操作起来也是类似的。这样就很直观地为用户提供了观察和显示当前日期的途径。MFC 提供了 CMonthCalCtrl 类进行支持。
- 日期/时间选择器（Date Time Picker）：这也是 VC6.0 新提供的控件类型。日期/时间选择器向用户提供了一种直观的选择日期和时间的方法。日期/时间选择器在外观上类似于一个组合框，但是当用户单击下拉箭头时就会展开一个日历控件供用户进行选择，而一旦用户做出了选择，日期/时间选择器会自动显示新的日期/时间。MFC 提供了 CDateTimeCtrl 类进行支持。
- IP 地址控件（IP Address）：这也是 VC6.0 新提供的控件类型。很显然，IP 地址控件用来输入和编辑 IP 地址。该控件外观类似于一个编辑框，但是可以自动对输入的字符按三个一组进行区分和加间隔圆点。IP 地址控件为开发支持 Internet 技术的程序提供了方便。MFC 提供了 CIPAddressCtrl 类进行支持。
- 扩展组合框（Extended Combo Box）：这也是 VC6.0 新提供的控件类型。扩展组合框在普通组合框的基础上还支持图像列表。也就是说，可以在组合框中显示特定的图标表示相应的选择，而不仅仅是显示文本。MFC 提供了 CComboBoxEx 类进行支持。

这里列出了十几种控件类型，在本书中不可能一一介绍。本节中通过“CompuInfo”例子程序介绍了静态文本、按钮、编辑框、列表框、成组框、单选按钮、组合框和核选按钮等类型的控件，在下一节中还将介绍列表控件和树形控件。其他类型控件的使用方法可以参考联机手册中的有关内容或相关书籍。

3．向对话框中添加控件

在新的“Computer Information”对话框中，已经有了两个按钮：“OK”按钮和“Cancel”按钮，这是缺省

生成的。大多数的对话框都需要两个类似的按钮。

现在来向“Computer Information”对话框添加一些新的控件。添加控件的方法很简单，先在控件工具箱中选中控件的类型，然后在对话框的空白处用鼠标进行拖拉就可以了；或者在选择控件类型后，在对话框的空白处单击，一个缺省大小的控件就会出现在对话框的中央位置。最后完成的对话框如图 4-4 所示。

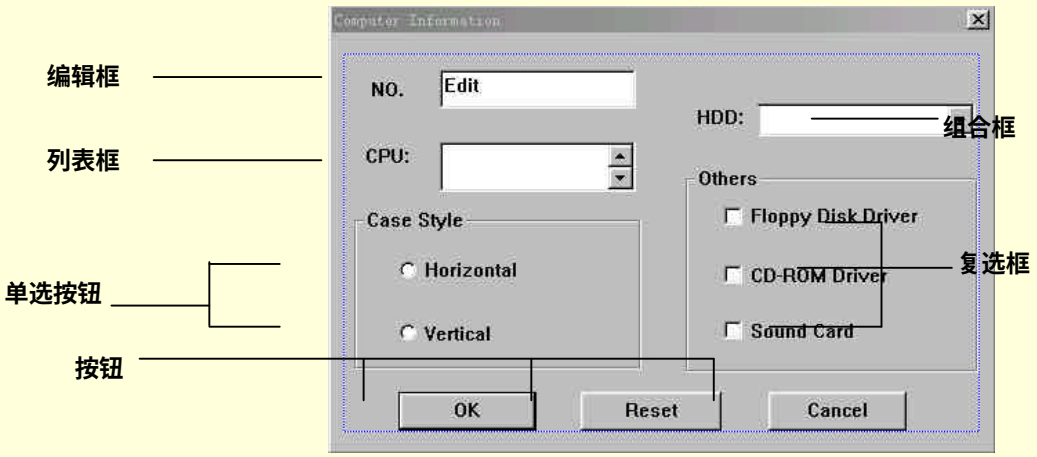

图 4-4 完成后的“Computer Information”对话框

在图 4-4 中，除了静态文本控件和成组框控件外，其他类型的控件都加上了说明，用户在添加控件的时候注意不要弄错了类型。

在加入组合框控件的时候，有一点需要注意：在拖拉出组合框编辑部分的大小后，还需要拖拉出下拉窗口的大小，如图 4-5 所示。方法是单击组合框的下拉箭头，此时出现一个带有缩放句柄的矩形框，拖拉矩形框底边的中点为蓝色，是可以拖动的，用鼠标进行拖拉就可以了。

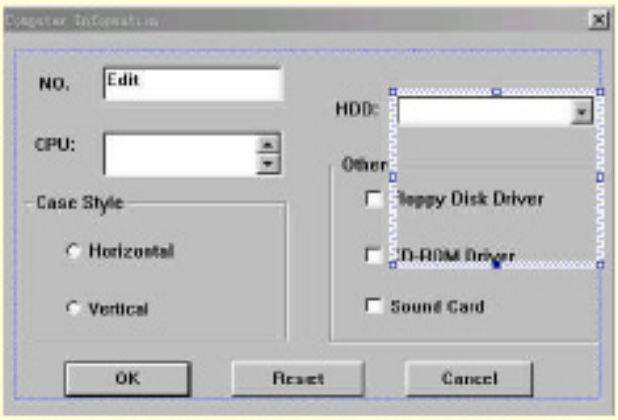

图 4-5 设置组合框的下拉窗口的大小

在添加控件的过程中，可以使用布局工具栏对对话框中的控件进行定位与布局。

在添加控件的工作完成后，对话框还仅仅是在视觉上满足了要求，还需要设置对话框中各控件的属性和跳转顺序。

4. 设置控件的属性

各种不同类型的控件提供了不同的选项供用户选择。这里不打算一一描述这些选项的意义和用法，只对某些重要的，在“CompuInfo”程序中需要用到的选项加以说明。表 4-1 中列出了各控件的需要设置的属性。

表 4-1 设置控件的属性

控件类型	命令 ID	标题文本	需要修改的属性
静态文本	IDC_STATIC	No.	无
编辑框	IDC_EDT_NUMBER	无	无
静态文本	IDC_STATIC	CPU:	无
列表框	IDC_LST_CPU	无	取消“Sort”选项

成组框	IDC_STATIC	Case Style	无
单选按钮	IDC_RAD_HOR	Horizontal	选中“Group”选项
单选按钮	IDC_RAD_VER	Vertical	无
静态文本	IDC_STATIC	HDD:	无
组合框	IDC_CMB_HDD	无	取消“Sort”选项
成组框	IDC_STATIC	Others	无
复选框	IDC_CHK_FDD	Floppy Disk Driver	无
复选框	IDC_CHK_CDROM	CD-ROM Driver	无
复选框	IDC_CHK_SND	Sound Card	无
按钮	IDOK	OK	无
按钮	IDC_BTN_RESET	Reset	无
按钮	IDCANCEL	Cancel	无

在表 4-1 中，在选择命令 ID 的时候，使用了“IDC_”+“控件类型”+“控件含义”的格式，通过这种方式组成的命令 ID，很容易就能知道其代表的含义。当然，用户完全可以随便命名，但是，有规则的命名习惯在开发大型应用程序的时候是很重要的。

在表 4-1 中只列出了控件的需要修改的属性，其他控件接受缺省的属性设置即可。

为单选按钮 IDC_RAD_HOR 选中“Group”选项是有目的的。前面已经提到，成组框只能在视觉效果上将控件组合在一起；在程序运行的时候，需要通过某种途径确定单选按钮的“组”。

MFC 是这样做的，按跳转顺序在对话框所拥有的多个单选按钮中进行搜寻，如果某个单选按钮设置了“Group”属性，则认为从该单选按钮开始了一个新的单选按钮组，直到遇到下一个设置了“Group”属性的按钮为止。因此，“Group”属性是一个相当重要的标志。

5. 设置控件的跳转顺序

所谓控件的跳转顺序，就是当用户使用“Tab”键移动当前输入焦点的时候，各控件接收焦点的顺序。

在设计对话框的时候，控件添加的顺序也就是控件的跳转顺序，当然，可以重新设置各控件的跳转顺序。通常情况下，设置跳转顺序的一个重要原则就是按照从上到下，从左到右的方向进行编号。

对话框中的每个控件都有一个跳转顺序的编号，但是并不是每个控件都能够拥有输入焦点，只有在控件的属性对话框中选中了“Tab stop”选项的控件才能够拥有输入焦点。比如说，静态文本控件就不能接收输入焦点。该控件在接收到输入焦点的时候，马上将焦点传递给下一个控件。

注意：

> 因此，在设置跳转顺序的时候，另外一个重要原则就是，静态控件的跳转顺序应该正好位于与其相关的控件之前。

在对话框处于打开状态时，选择“Layout”菜单下的“Tab Order”命令，对话框各控件的左上角出现一个数字，该数字代表了该控件在当前跳转顺序中的编号。按需要的跳转顺序单击各控件，可以看到编号相应改变。在完成设置后，单击对话框的空白处或按“Enter”键结束设置跳转顺序的工作。对于“Computer Information”对话框，正确的跳转顺序如图 4-6 所示。

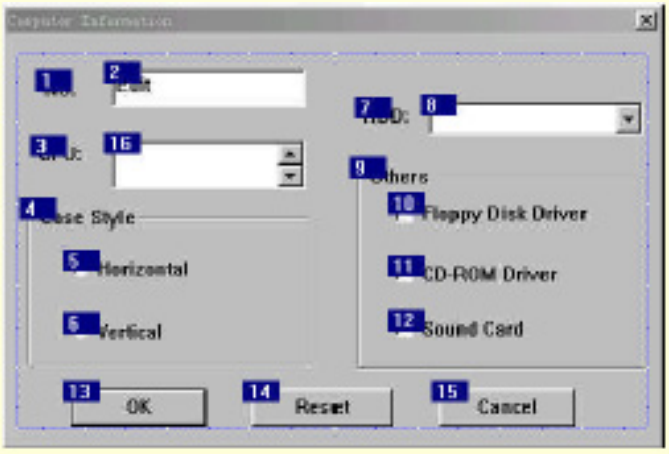

图 4-6 设置控件的跳转顺序

4.1.3 建立对话框类

对用户自己设计的对话框，通常都有一个特定的类与之对应，该类从 CDialog 类派生。ClassWizard 可以帮助用户方便地完成建立对话框类和向对话框类添加成员变量的工作。

1．建立新的对话框类

在对话框资源处于打开状态时，打开 ClassWizard 对话框，ClassWizard 检测到新的对话框的存在，并弹出“ Adding a class ”对话框询问用户是否生成新的对话框类。“ Adding a class ”对话框如图 4-7 所示。

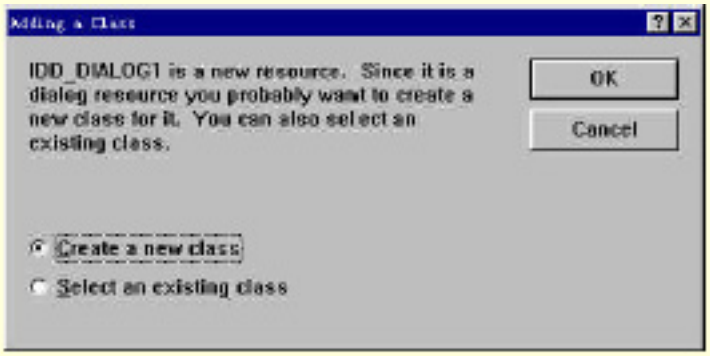

图 4-7 “ Adding a class ”对话框

用户可以选择创建新的对话框类或者选择一个已经存在的类。在大多数情况下，当然是选择建立新的对话框类。ClassWizard 弹出“ New Class ”对话框供用户填写类名等信息。“ New Class ”对话框如图 4-8 所示。

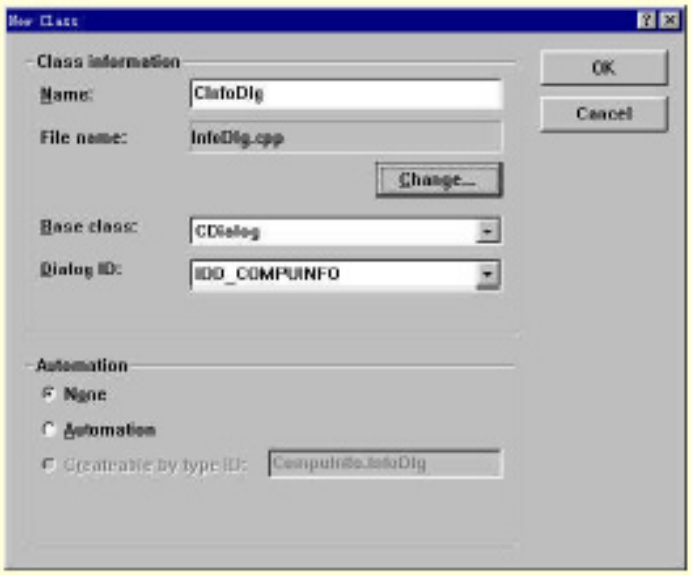

图 4-8 “ New Class ”对话框

在“ New Class ”对话框中，输入新的对话框类的类名“ CInfoDlg ”，这时在“ File name ”框中出现该类的头文件和源代码文件的文件名。如果用户对缺省的文件名不满意，可以单击“ Change ”按钮进行更改；选择新类的基类为 CDialog，这是所有对话框的基类；在“ Dialog ID ”框中选择对话框资源的命令 ID。单击“ OK ”按钮完成设置工作，ClassWizard 将会为用户生成新类的头文件和源代码文件。在 ClassWizard 的类列表中也出现了新增加的类。

在应用程序中，通过建立 CInfoDlg 类的对象，就可以使用先前建立的对话框了。但是在使用对话框之前，为了对话框能够方便地与其他对象交换数据，需要向对话框类增添一些成员变量。

2．添加对话框类的成员变量

如果应用程序的用户在应用程序的对话框中输入了一些信息，对话框对象必须能够接收这些信息并进一步将这些信息与应用程序的其他对象进行交换。由于对话框的控件反应了用户输入的信息，因此对话框对象首先需要从控件中收集用户输入的信息，用一套数据结构保存起来，然后再用这套数据结构与其他对象进行交换。

MFC 采用为对话框类添加与控件有关的成员变量来完成收集和交换信息的任务。对于某种特定类型的控件，有相应类型的成员变量与之进行交互。举个例子，对于一个编辑框，可以添加一个 CString 类型的成员变量与其关联，在对话框打开和关闭的时候，该变量的值就代表了编辑框中的内容。

当然，作为一个一般的类，也可以为对话框类添加其他的一些成员变量，这些变量可能和对话框中的控件没有任何关联。

ClassWizard 为用户在对话框类中添加与控件相联系的成员变量提供了方便。打开 ClassWizard，选取“Member Variables”标签，在类名框中选择 CInfoDlg 类，如图 4-9 所示。

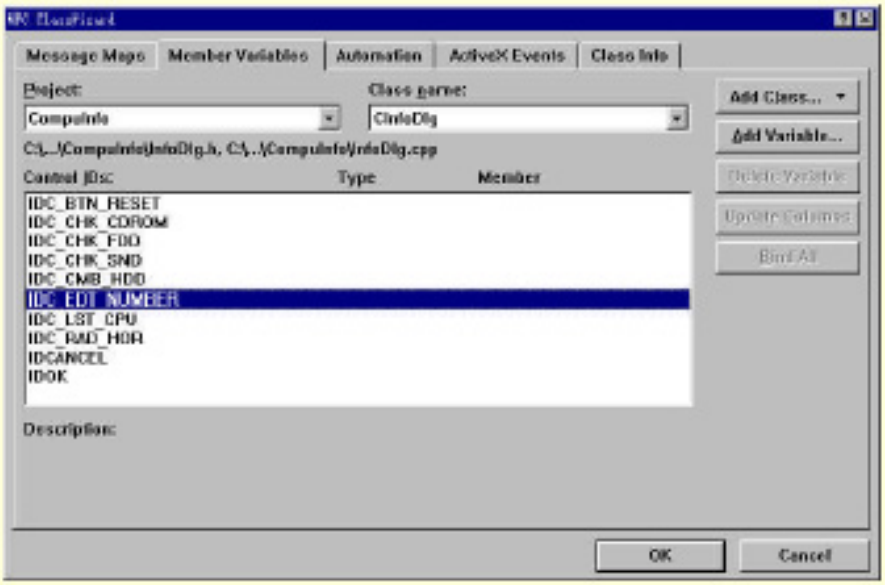

图 4-9 使用 ClassWizard 为对话框类添加成员变量

在 ClassWizard 的 Object IDs 列表中选择合适的控件的命令 ID，如上图中选择了 ID_EDT_NUMBER，单击“Add Variable”按钮，弹出“Add Member Variable”对话框，如图 4-10 所示。

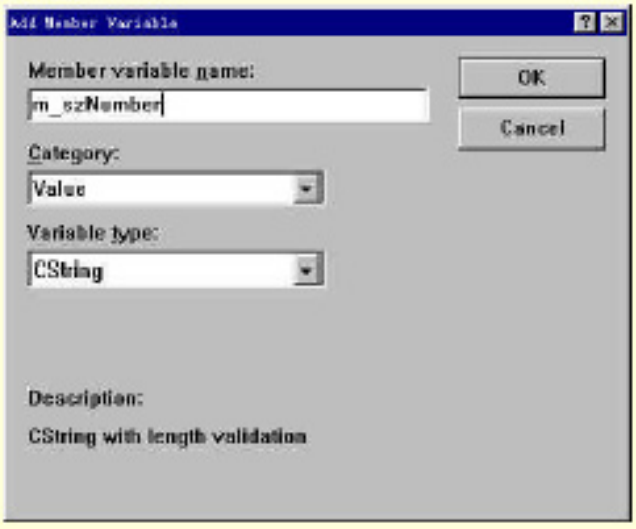

图 4-10 “Add Member Variable”对话框

在“Add Member Variable”对话框中，由于在前面选中的 IDC_EDT_NUMBER 是一个编辑框，因此缺省的变量类型为 CString。输入变量名“m_szNumber”，单击“OK”按钮确认添加该成员变量。

用户可能已经注意到，在“Add Member Variable”对话框中，可以选择添加变量的种类，是“值（Value）”类型或者“控件（Control）”类型。上面加入的 m_szNumber 变量就是一个“值（Value）”类型的变量，或者说是普通类型的变量。如果用户选择了添加“控件”类型的变量，就可以为该控件添加一个对应类型的变量，在程序的运行过程中，可以通过该变量对该控件进行控制。

在 CInfoDlg 类中，需要添加的成员变量列于表 4-2 中。

表 4-2 CInfoDlg 类的成员变量

控 件	变量名	变量类型
IDC_CHK_CDROM	m_bCDROM	BOOL
IDC_CHK_FDD	m_bFDD	BOOL
IDC_CHK_SND	m_bSND	BOOL
IDC_CMB_HDD	m_szHDD	CString
IDC_CMB_HDD	m_CmbHDD	CComboBox
IDC_EDT_NUMBER	m_szNumber	CString
IDC_LST_CPU	m_szCPU	CString
IDC_LST_CPU	m_LstCPU	CListBox
IDC_RAD_HOR	m_nCaseIndex	int

在为编辑框 IDC_EDT_NUMBER 添加关联的 m_szNumber 变量后，ClassWizard 会向用户询问该编辑框中可能输入的字符数目的上限，这里输入“ 20 ”，也就是说，在该编辑框中最多可以输入 20 个字符。如图 4-11 所示。

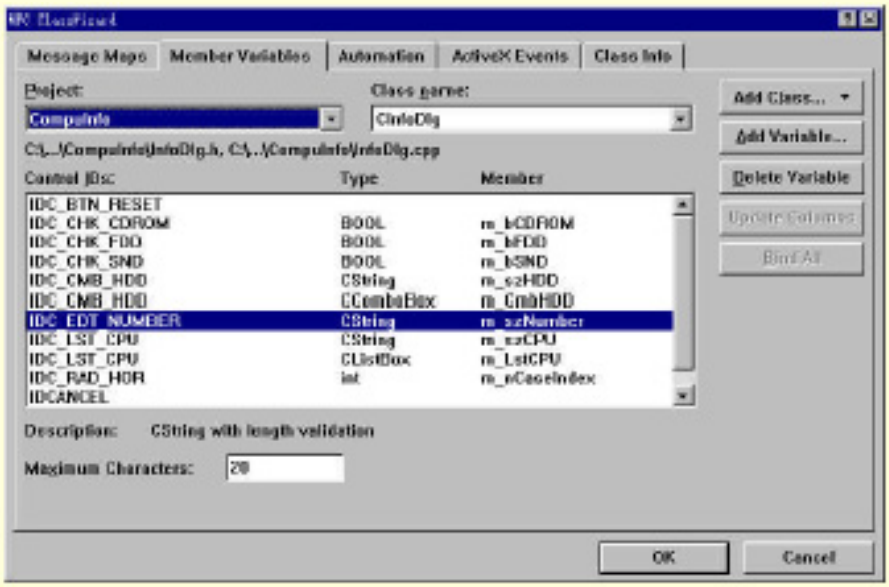

图 4-11 限制编辑框可能输入的字符的数目

对与组合框 IDC_CMB_HDD 关联的 m_szHDD 变量，设置最多可能字符数目为 10。

除了 CString 类型的变量的字符数目进行限制外，还可以对数值类型变量的取值范围进行限制。当然，也可以保持输入框为空白，这样就可以忽略这些限制。

这些变量的类型与控件类型的关系是很容易理解的，比如说，编辑框中输入的字符串用一个 CString 类型的变量来保存是再恰当不过了。比较难以理解一点的是与单选按钮相关联的 m_nCaseIndex 变量，该变量被声明为 int 类型，代表了单选按钮组中当前被选中按钮的序号，该序号从 0 开始计算。

这些变量将被对话框对象用来与应用程序的其他对象交换数据，在对话框控件与对话框对象之间交换数据，以及检查交换的数据是否有效。这被称为 DDX（Dialog Data Exchange：对话框数据交换）与 DDV（Dialog Data Validation：对话框数据确认）。

3. 了解 ClassWizard 所做的幕后工作

在向 CInfoDlg 类添加成员变量时，ClassWizard 作了大量的幕后工作，用户有必要弄清除这些在幕后发生的事情。在完成整个例子程序后，用户将会对此有一个全面、清楚的认识。

ClassWizard 所做的第一件事情当然是在头文件 InfoDlg.h 中添加的成员变量的声明，如程序清单 4-1 所示。

程序清单 4-1 成员变量的声明

```
// Dialog Data
//{{AFX_DATA(CInfoDlg)
```

```
enum { IDD = IDD_COMPUINFO };
CListBox   m_LstCPU;
CComboBox      m_CmbHDD;
BOOL       m_bCDROM;
BOOL       m_bFDD;
BOOL       m_bSND;
CString    m_szNumber;
int        m_nCaseIndex;
CString    m_szCPU;
CString    m_szHDD;
//}}AFX_DATA
```

注意到 CInfoDlg **类的这些成员变量都被注释对**//{{AFX_DATA……//}}AFX_DATA **所包围，这标明这些变量是被** DDX **所使用的。用户最好不要手工编辑这些变量，如果确实需要修改这些变量，可以在** ClassWizard **中先进行删除，再重新建立成员变量。**

ClassWizard **所做的第二件事情就是在** CInfoDlg **类的构造函数中进行成员变量的初始化工作，如程序清单** 4-2 **所示。**

程序清单 4-2　CInfoDlg **类的构造函数**

```
CInfoDlg::CInfoDlg(CWnd* pParent /*=NULL*/): CDialog(CInfoDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CInfoDlg)
m_bCDROM = FALSE;
m_bFDD = FALSE;
m_bSND = FALSE;
m_szNumber = _T("");
m_nCaseIndex = -1;
m_szCPU = _T("");
m_szHDD = _T("");
//}}AFX_DATA_INIT
}
```

同样，这些初始化代码被特定的注释对包围起来，用户无需手工修改。注意成员变量 m_nCaseIndex **被初始化为“** -1 **”，这表示没有任何单选按钮被选中。**

ClassWizard **所做的第三件事情就是在** CInfoDlg::DoDataExchagne()**函数中以函数调用的形式将对话框的控件与成员变量确实地关联起来，如程序清单** 4-3 **所示。**

程序清单 4-3　CInfoDlg::DoDataExchange()

```
void CInfoDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CInfoDlg)
DDX_Control(pDX, IDC_LST_CPU, m_LstCPU);
DDX_Control(pDX, IDC_CMB_HDD, m_CmbHDD);
DDX_Check(pDX, IDC_CHK_CDROM, m_bCDROM);
DDX_Check(pDX, IDC_CHK_FDD, m_bFDD);
DDX_Check(pDX, IDC_CHK_SND, m_bSND);
```

```
DDX_Text(pDX, IDC_EDT_NUMBER, m_szNumber);
DDV_MaxChars(pDX, m_szNumber, 20);
DDX_Radio(pDX, IDC_RAD_HOR, m_nCaseIndex);
DDX_LBString(pDX, IDC_LST_CPU, m_szCPU);
DDX_CBString(pDX, IDC_CMB_HDD, m_szHDD);
DDV_MaxChars(pDX, m_szHDD, 10);
//}}AFX_DATA_MAP
}
```

这些函数均以“DDX_”或“DDV_”开头。“DDX_”类型的函数负责完成从控件到变量或者从变量到控件的数据交换工作，“DDV_”类型的函数负责检查用于交换的数据是否合法，而且“DDV_”类型的函数必须紧跟在相应的“DDX_”类型函数的后面。“DDX_”和“DDV_”函数的配合使用，使得对话框数据的交换和确认工作变得十分容易。

用户自己并不需要直接调用 DoDataExchange()函数，该函数将在需要交换数据的时候由 UpdateData()函数调用，UpdateData()函数在调用的时候决定了数据传送的方向。一般说来，在对话框被显示的时候，数据从成员变量传送到控件；在对话框被关闭的时候则反之。当然，用户可以在需要的时候调用 UpdateData()函数，并通过参数决定数据传送的方向。在“CompuInfo”程序中，用户将会清楚地看到 UpdateDate()函数的使用。

4.1.4 使用对话框

现在用户最迫切的愿望可能就是显示出对话框，以检查自己的成果了。现在就来实现这一愿望。

1．建立新的菜单项和消息处理函数

在大多数情况下，对话框都是由菜单命令所引出的，即在程序的某个菜单命令的处理函数中弹出对话框。因此首先需要向 CompuInfo 程序添加一个新的菜单项，并生成该菜单命令的消息处理函数。

可以在 CompuInfo 程序的“View”菜单下添加一个新的菜单项，完成后的“View”菜单如图 4-12 所示。

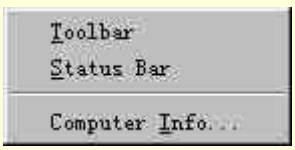

图 4-12 新的“View”菜单

修改程序菜单的方法已经在上一章中介绍过，这里不再重复。新菜单项的命令 ID 为：ID_VIEW_INFO，提示信息为“Input the infomation of user's computer”。使用 ClassWizard 完成该菜单项命令的消息处理函数，由 CCompuInfoDoc 类接收该命令，并接收缺省的消息处理函数名：OnViewInfo()。

2．添加代码以显示对话框

现在来完成 CCompuInfoDoc::OnViewInfo()函数。显示对话框需要两步：生成一个对话框对象，然后调用该对象的 DoModal()函数显示对话框。因此 OnViewInfo()函数中的代码应该如程序清单 4-4 所示。

程序清单 4-4 CCompuInfoDoc::OnViewInfo()

```
void CCompuInfoDoc::OnViewInfo()
{
// TODO: Add your command handler code here
// 生成对话框对象
CInfoDlg dlg;
// 显示对话框
if(dlg.DoModal()==IDOK)
{
}
```

```
}
```

在 OnViewInfo()函数中，调用了 DoModal()函数来显示对话框，这意味着对话框窗口将是有模式的，在该对话框窗口被关闭之前不能进行应用程序的其他工作。DoModal()函数的返回值代表了关闭的按钮：如果用户按下“ OK ”按钮关闭对话框，返回值就是 IDOK；如果用户按下“ Cancel ”按钮对话框，返回值就是 IDCANCEL。

在大多数情况下，只有用户选择“ OK ”按钮关闭对话框时才需要对对话框中的数据做进一步的处理，因此，根据 DoModal()函数的返回值就可以判断是否需要处理对话框的数据，这是很方便的。

在重新编译并运行 CompuInfo 程序之前，还需要在 CompuInfoDoc.cpp 文件的顶部加上这样一行：

```
#include "InfoDlg.h"
```

这样编译器才能够识别新的 CInfoDlg 类。

现在用户可以编译并运行“ CompuInfo ”程序了。选择“ View ”菜单下的“ Computer Info ”菜单项，检查“ Computer Information ”对话框。如图 4-13 所示。

用户可以发现，对话框中的各控件都能够正常地工作，而且对话框各控件的状态也反应了 CInfoDlg 类的构造函数对各成员变量进行初始化的结果，这些成员变量的值最终由与之相关联的控件表现出来。

但是令人遗憾的是，到目前为止此对话框还只是一个简单的框架，没有程序代码给予支持，因此用户也无法在其中进行选择，例如：组合框中的内容为空，无法选择；列表框中的内容由 VC 自动添加，并不是我们需要的；另外，添加的“ Reset ”按钮也没有起到其作用：按照对话框成员变量的值重新设置各控件的内容。

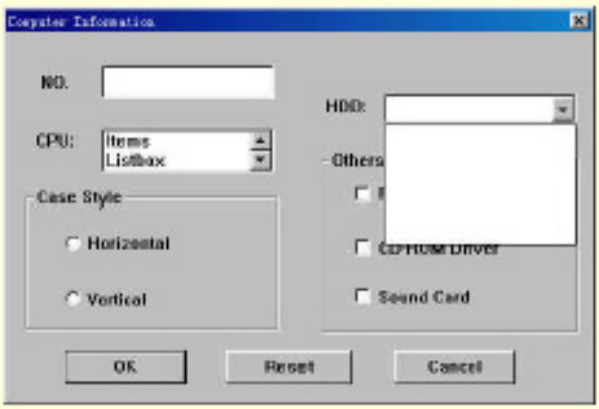

图 4-13 Computer Information 对话框

因此，用户还应该进一步完善对话框的工作。为了让列表框和组合框中出现可能的选择，必须在对话框窗口被显示之前对列表框和组合框添加数据。为了“ Reset ”按钮能够真正起到“ 重置控件内容 ”的作用，必须对“ Reset ”按钮的控件通知进行处理。

3. 初始化对话框的控件

初始化对话框中控件的工作需要在对话框窗口以及其控件已经被创建，但对话框窗口被显示之前进行，也就在这个时候，对话框对象的 OnInitDialog()函数被调用，因此，应该在 OnInitDialog()函数中完成控件的初始化工作。

OnInitDialog()函数是对话框窗口的 WM_INITDIALOG 消息的处理函数，因此首先应该生成该消息处理函数。使用 ClassWizard 完成这一工作，如图 4-14 所示。

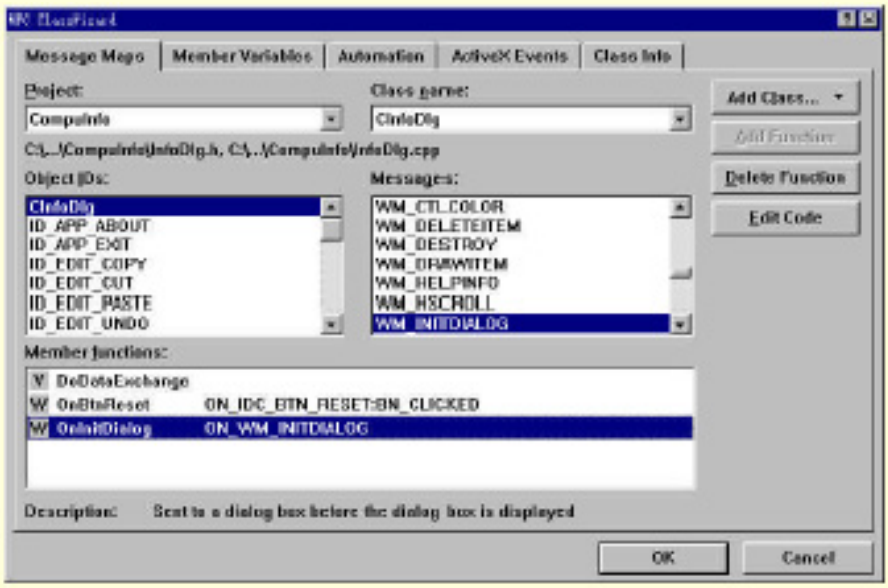

图 4-14 添加 OnInitDialog()函数

在“ Message Maps ”标签下，先选择类名：CInfoDlg；在“ Object IDs ”列表中不要选择用户界面元素的命令 ID，而是选择类名，在“ Message ”列表框中就会列出当前可以使用的各种消息，选择“ WM_INITDIALOG ”消息，单击“ Add Function ”按钮就可以生成相应的消息处理函数。

用户或许还记得，在前面为 CInfoDlg 类添加成员变量的时候，对列表框控件和组合框控件除了添加了一个 CStirng 类型的“ 普通 ”类型的变量外，还添加了一个对应类型的“ 控件 ”类型的变量，通过这些“ 控件 ”类型的变量，就可以对控件进行操作了。

在 OnInitDialog()函数中，就需要使用这些“ 控件 ”类型的变量。先来完成组合框的初始化工作，其实最主要的工作就是添加一些可能的选项。CComboBox 类提供了 AddString()函数可以完成这一任务，因此在 OnInitDialog()函数中需要加上这样的一段代码：

```
// 为组合框添加选项
m_CmbHDD.AddString("1.2G");
m_CmbHDD.AddString("1.7G");
m_CmbHDD.AddString("2.1G");
m_CmbHDD.AddString("3.2G");
m_CmbHDD.AddString("6.4G");
m_CmbHDD.AddString("10G");
```

这里只列出了六项，如果用户愿意，列出多少项都可以。AddString()函数将参数字符串加入到组合框的列表中，由于在设置组合框属性的时候，取消了“ Sort ”选项，因此，添加字符串的顺序就是组合框列表中的顺序。如果设置了“ Sort ”选项，组合框将自动进行重新排列。

类似地可以完成向列表框中添加选项的工作，但这里使用一种稍稍有些不同的方式。这里不再罗列许多个 AddString()函数，而是预先定义好一个字符串数组，然后采用一个循环将这些字符串添加到列表框控件中。

当然，这样做并不仅仅只是耍一个花招。实际上，在真正的应用程序中，需要向列表中添加的字符串个数往往都是动态的，字符串的内容也是可变的，因此，经常采用循环来完成添加字符串的工作。

因此，最终的 OnInitDialog()函数如程序清单 4-5 所示。

程序清单 4-5 CInfoDlg::OnInitDialog()

```
// 预定义列表框的字符串数组
static CString CPUType[]=
        {"AMD K5",       "Intel Pentium",  "Cyrix MII",
         "AMD K6",       "Intel Pentium Pro",    "AMD K6-2",
```

```
        "Intel Celeron",  "Intel Pentium II",      "Intel Pentium III"};

BOOL CInfoDlg::OnInitDialog()
{
CDialog::OnInitDialog();
// TODO: Add extra initialization here
// 为列表框添加选项
for(int i=0;i<=8;i++)
{
      m_LstCPU.AddString(CPUType[i]);
}
m_LstCPU.SelectString(-1,m_szCPU);

// 为组合框添加选项
m_CmbHDD.AddString("1.2G");
m_CmbHDD.AddString("1.7G");
m_CmbHDD.AddString("2.1G");
m_CmbHDD.AddString("3.2G");
m_CmbHDD.AddString("6.4G");
m_CmbHDD.AddString("10G");

return TRUE;
}
```

注意字符串数组 CPUType **的定义被放在了** OnInitDialog()**函数之外，并加上了“** static **”关键字，这表明该数组只在本文件范围内有效，而不是作为一个全局变量存在。**

4．重置控件的内容

在对话框显示出来后，用户可能会改变某些控件的内容，如修改编辑框中的字符串、改变列表框中的选择等等。但是，即使控件上显示的内容发生了改变，对话框的成员变量中保持的内容并没有改变。

“ Reset **”按钮的作用就是重新设置控件中显示的内容，使得其与对话框对象的各成员变量中保存的内容一致，也就是说，主动调用** UpdateData()**函数，再次由对话框的成员变量向对话框的控件发送数据。**

UpdateData()**函数需要一个** BOOL **类型的参数以决定数据传送的方向。如果参数为** TRUE**，则从控件向成员变量发送数据；如果参数为** FALSE**，则从成员变量向控件发送数据。**

在对话框窗口显示出来的时候，会自动调用 UpdateData（FALSE）**函数；而在对话框被关闭的时候，会自动调用** UpdateData（TRUE）**函数，这样就完成了对话框的成员变量和控件之间的数据交换。实际上，**UpdateData()**函数是调用了** DoDataExchange()**函数完成数据传递任务的。**

对于 CompuInfo **程序的“** Computer Information **”对话框中的“** Reset **”按钮，只要在其消息处理函数中调用** UpdateData（FALSE）**就可以了。**

仍然使用 ClassWizard **对“** Reset **”按钮发送的控件通知进行映射。在“** Object IDs **”列表中选择** IDC_BTN_RESET**，在“** Message **”列表中选择** BN_CLICKED**，单击“** Add Function **”按钮，就生成了对“** Reset **”按钮的单击动作的消息相应函数（参见图** 4-14**）。**

用户现在可以领会到，无论是窗口消息，还是界面命令或者控件通知，使用 ClassWizard **进行消息映射的工作都是类似的。**

程序清单 4-6 **中列出了** OnBtnReset()**函数的代码。**

程序清单 4-6　CInfoDlg::OnBtnReset()

```
void CInfoDlg::OnBtnReset()
{
// TODO: Add your control notification handler code here
// 重置各控件显示的内容
UpdateData(FALSE);
}
```

到现在为止，“Computer Information”对话框本身需要完成的任务已经全部实现。但是，对话框的数据还只是在控件和成员变量之间进行交换，更重要的任务是需要把数据传送到应用程序的其他对象中去。

4.1.5 与其他对象交换数据

在 CompuInfo 程序中，并不需要一个很复杂的例子来演示对话框对象与其他对象间的数据交换，重要的是向用户演示如何实现数据的交换，以使用户能够理解数据的传送途径。

CompuInfo 程序的计划是，在显示对话框窗口之前，对话框对象从文档对象获取数据，并据此初始化控件显示的内容；而一旦用户按“OK”按钮关闭了对话框，对话框对象将向文档对象传递数据，最后将用户作出的选择显示在视图的客户区中。

1．添加文档类的成员变量

由于需要记录的内容相同，因此文档类的成员变量与对话框类的成员变量基本相同，只是没有“控件”类型的成员变量而已。因此文档类需要添加的成员变量包括：

```
public:
    int m_nCaseIndex;
    BOOL m_bSND;
    BOOL m_bCDROM;
    BOOL m_bFDD;
    CString m_szHDD;
    CString m_szCPU;
    CString m_szNumber;
```

注意这些变量的存取级别全都是“Public”，这样在程序中可以比较方便地修改这些变量的取值。但是，正如在第二章中所指出，这样做是不符合“数据封装”的要求的。对于比较大的程序，应该严格保证数据结构的私有性。

当然，在使用这些成员变量之前，应该进行初始化。初始化工作在 OnNewDocument()函数中完成。程序清单 4-7 中是 OnNewDocument()函数的完整代码。

程序清单 4-7　CCompuInfoDoc::OnNewDocument()

```
BOOL CCompuInfoDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
    return FALSE;
// 初始化成员变量
m_szNumber="10000123";
m_szCPU="Intel Celeron";
m_szHDD="2.1G";
m_bFDD=TRUE;
m_bCDROM=FALSE;
m_bSND=FALSE;
m_nCaseIndex=0;
```

```
return TRUE;
}
```

2．实现数据交换

前面已经说过，在显示对话框之前和关闭对话框之后，对话框对象都要与文档类对象交换数据。实现起来很简单，在显示对话框之前，让文档对象的成员变量给对话框对象的成员变量赋值；在关闭对话框之后，让对话框对象的成员变量给文档对象的成员变量赋值就可以了。

这一系列的赋值操作都在 CCompuInfoDoc::OnViewInfo()函数中完成，其代码如程序清单 4-8 所示。

程序清单 4-8　CCompuInfoDoc::OnViewInfo()

```
void CCompuInfoDoc::OnViewInfo()
{
// TODO: Add your command handler code here
// 生成对话框对象
CInfoDlg dlg;
// 向对话框的成员变量传送数据
dlg.m_szNumber=m_szNumber;
dlg.m_szCPU=m_szCPU;
dlg.m_szHDD=m_szHDD;
dlg.m_bFDD=m_bFDD;
dlg.m_bCDROM=m_bCDROM;
dlg.m_bSND=m_bSND;
dlg.m_nCaseIndex=m_nCaseIndex;

// 显示对话框
if(dlg.DoModal()==IDOK)
{
    //从对话框的成员变量接收数据
    m_szNumber=dlg.m_szNumber;
    m_szCPU=dlg.m_szCPU;
    m_szHDD=dlg.m_szHDD;
    m_bFDD=dlg.m_bFDD;
    m_bCDROM=dlg.m_bCDROM;
    m_bSND=dlg.m_bSND;
    m_nCaseIndex=dlg.m_nCaseIndex;
}

// 更新视图中的显示
UpdateAllViews(NULL);
}
```

在 OnViewInfo()函数的代码中，很容易看到对话框对象和文档对象交换数据的操作。在完成数据的交换后，调用了 UpdateAllViews()函数通知视图类对象文档中保存的内容已经改变，需要重新输出。

3．在视图中输出数据

在视图客户区中输出文档的内容是很正常的事情，需要向 CCompuInfoView::OnDraw()函数中添加一些代码，如程序清单 4-9 所示。

程序清单 4-9 CCompuInfoView::OnDraw()

```
void CCompuInfoView::OnDraw(CDC* pDC)
{
CCompuInfoDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
pDC->TextOut(20,0,"My Computer");

pDC->TextOut(0,20,"Computer No. "+pDoc->m_szNumber);
pDC->TextOut(0,40,"CPU Type: "+pDoc->m_szCPU);
pDC->TextOut(0,60,"Hard Disk Driver: "+pDoc->m_szHDD);

switch(pDoc->m_nCaseIndex)
{
case 0:
    pDC->TextOut(0,80,"Case Type: Horizontal Case");
    break;
case 1:
    pDC->TextOut(0,80,"Case Type: Vertical Case");
    break;
}

pDC->TextOut(0,100,"Other:");
int y=120;
if(pDoc->m_bFDD)
{
    pDC->TextOut(20,y,"Floppy Disk Driver");
    y+=20;
}
if(pDoc->m_bCDROM)
{
    pDC->TextOut(20,y,"CD-ROM Driver");
    y+=20;
}
if(pDoc->m_bSND)
{
    pDC->TextOut(20,y,"Sound Card");
}
}
```

OnDraw()**函数中的代码是如此简单，相信用户自己可以看懂，因此这里不多加解释了。**

4.1.6 检查和运行程序

在完成“ CompuInfo **”程序的所有设计工作后，就可以试着运行和检查它了。当然，在编译运行时，读者可以按照下面的要求检查最后生成的程序，同时回顾一下本例子程序中所介绍的知识。**

1．运行和检查程序

用户现在可以编译和运行“CompuInfo”程序，可以按照下面的要求检查程序运行的内容：

- 检查对话框窗口显示时，控件的内容是否与文档中的内容一致。
- 检查单击“OK”按钮关闭对话框窗口时，文档的内容是否随着用户在对话框中的选择而改变。
- 试着在编辑框中输入多于 20 个字符的字符串，检查程序的反应。
- 单击“Reset”按钮，检查重置功能是否有效。

下面我们仅仅演示本程序最基本的应用，看看是否正确。按 Ctrl+F9 键运行程序，如图 4-15 所示为“CompuInfo”程序运行时的初始画面。

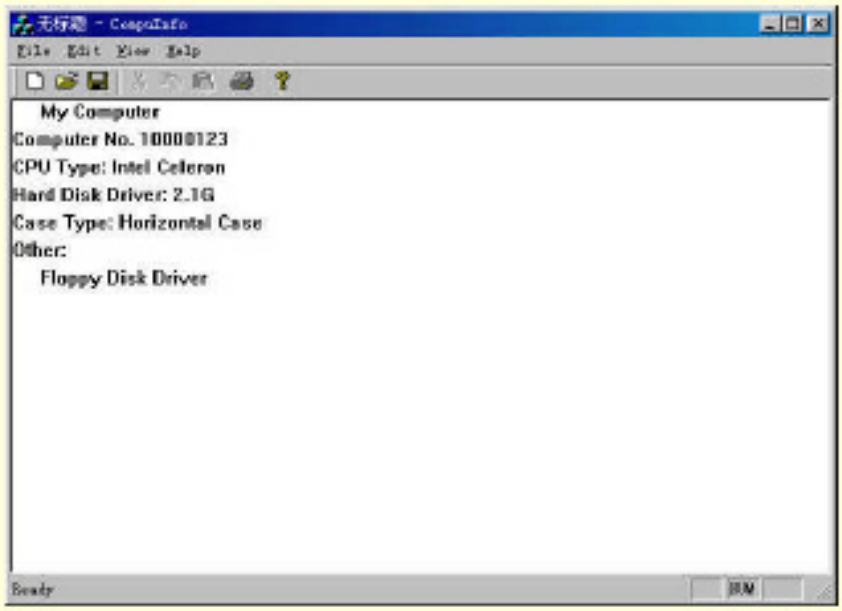

图 4-15　程序运行初始画面

然后选择 View 菜单下的 Computer Info 菜单项，打开如图 4-16 所示的对话框。

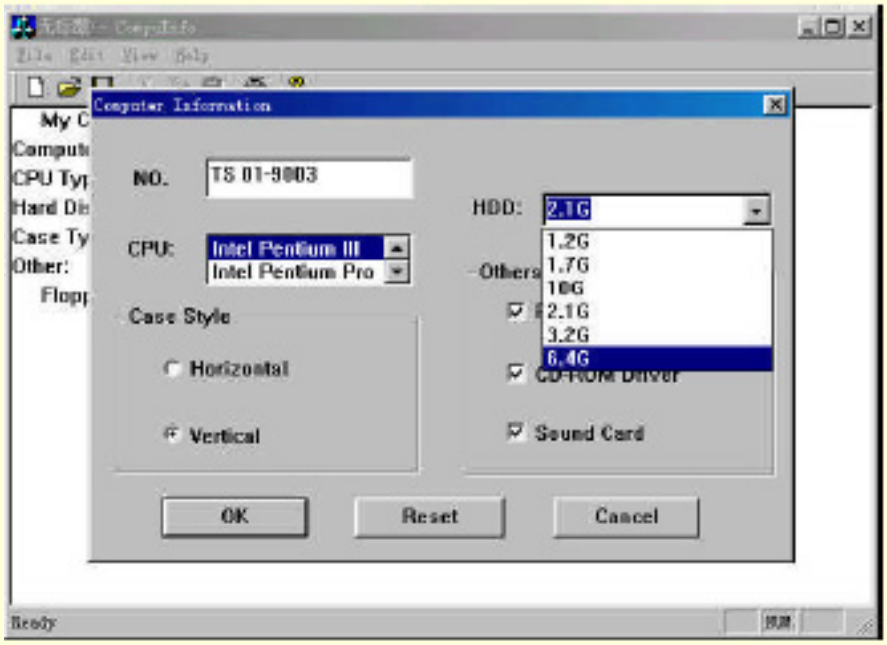

图 4-16　Computer Information 对话框

按照如图 4-16 所示的信息输入，单击“OK”按钮，回到主程序界面，在文档窗口中显示出我们在对话框中输入的数据信息，如图 4-17 所示。

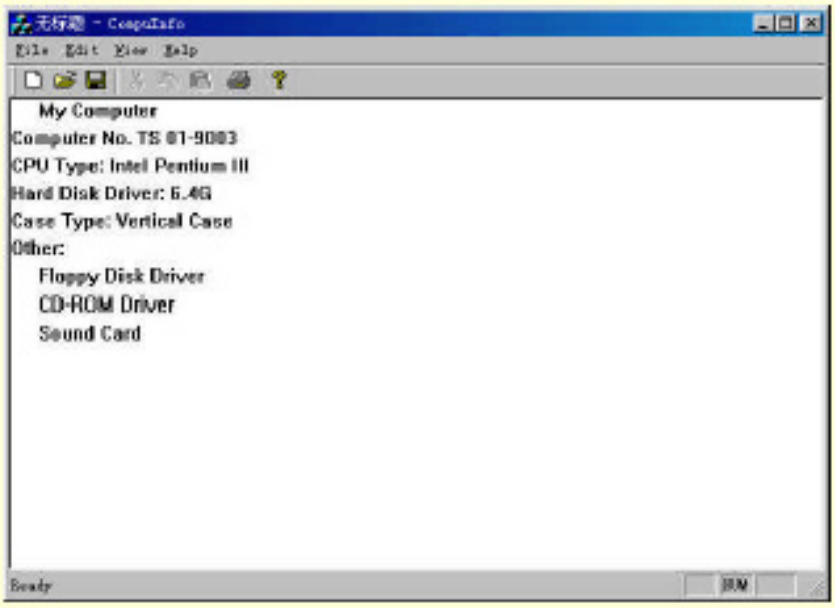

图 4-17 对话框输入的提交结果

2. 程序中的数据传递过程

在“CompuInfo”程序中，用户输入的信息最终反映到视图的客户区中，而在这一传递过程中，经历了多次数据交换过程，图 4-18 清楚地显示了数据交换的全过程。

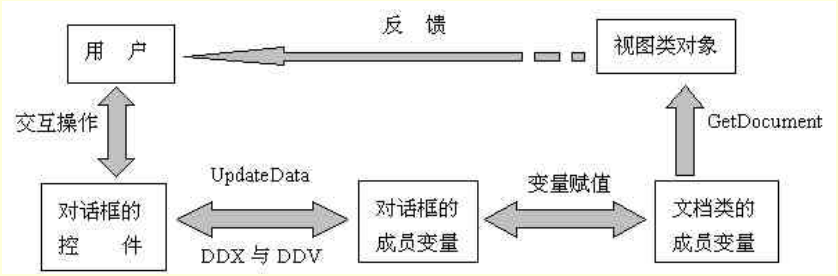

图 4-18 程序中的数据交换过程

在图 4-18 中，有两个比较重要的数据传递环节，一是对话框的控件与成员变量之间的数据交换，这由 UpdateData()函数控制 DDX 和 DDV 完成；另外一个就是对话框与文档类的成员变量间的数据交换，这需要在显示对话框前和关闭对话框后做一些变量赋值的工作。理解了这两个环节，也就容易理解整个程序的数据交换过程了。

当然，在“CompuInfo”程序中还提到了其他一些知识，如建立对话框资源、建立对话框类、向对话框类添加成员变量、如何实现有模式对话框等等，用户可以自行体会，或浏览 VC6.0 的联机手册获取更多的信息。

4.2 无模式对话框

在本章前面的两节中，主要介绍了有模式对话框和一些常用的控件。既然本章介绍有关对话框的内容，就不应该忽略了另外两个重要的内容：无模式对话框（Modeless Dialog）和通用对话框（Common Dialog）。

前面已经提到，在无模式对话框处于激活状态的同时，用户还可以选择应用程序的其他功能；而有模式的对话框则不行，必须首先关闭打开的对话框才能继续操作。

无模式对话框的典型是 Find ..菜单。 在对话框处于活动状态的同时，用户还可以在应用的其它地方中工作。因此，相对于有模式对话框，它的编程更加复杂，也对程序员提出了更多的要求：

- 应用不会等待用户在处理数据之前先关闭无模态对话框。这样，在对话框被激活时，必须更新应用中的数据。当在对话框以外的地方改变了信息时，应该更新对话框。

■ 在对话框所属的窗口有效时，对话框也应该可见。

下面我们将用一个小的例子来说明如何建立无模式的对话框。

4.2.1 建立 Exp1 项目

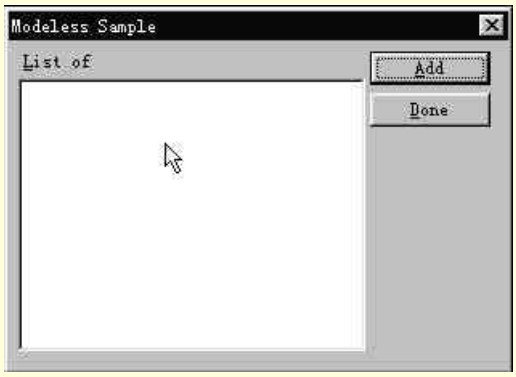

图 4-19 主对话框

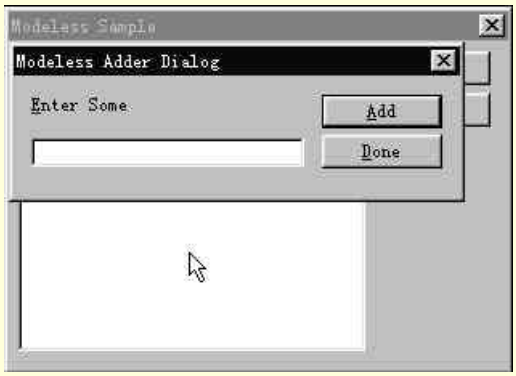

图 4-20 打开的无模式对话框

Exp1 是一个非常简单的基于对话框的应用程序。它在主对话框中管理一个列表框，并提供一个无模态的对话框，让你向主对话框中的列表框中加入字符串。

在应用开始时，出现一个空的列表框。你可以单击 Add 按钮来打开一个无模态的对话框。在这个无模态的对话框打开期间，你可以将焦点带回主对话框。当无模态的对话框打开时，主对话框中的 Add 按钮被禁止使用，这样用户就不能创建无模式对话框的多于一个的实例。

具体的应用程序外观如图 4-19 和 4-20 所示。

按以下步骤建立项目 Exp1：

（1）首先在 File 菜单中单击 New，单击 Project 标签，选择 MFC AppWizard(exe)。在 Project Name 框中键入 Exp1。

（2）单击 OK 进入下一个对话框。选择基于对话框的应用，并在 Language 下拉框选取英语。

（3）单击 Next 进入下一对话框，选择 None。

（4）单击 Next 进入下一对话框，清除 ActiveX Control 前面的复选标志。

（5）单击 Next 进入下一对话框，清除除了“3D Controls”外全部的复选标志。

（6）单击 Next 进入下一对话框，分别选取“MFC Project ”，“Yes,please” ，“As a share DLL”项。

（7）单击 Next，Appwizard 在其中列出了所有将要产生的类及文件信息。单击 Finish。

（8）显示了将要产生的新项目的一些信息。单击 OK，产生项目 Exp1。

通过这一系列操作，我们产生了一个基于对话框的项目。其中的一些缺省类名和选项如图 4-21 所示。

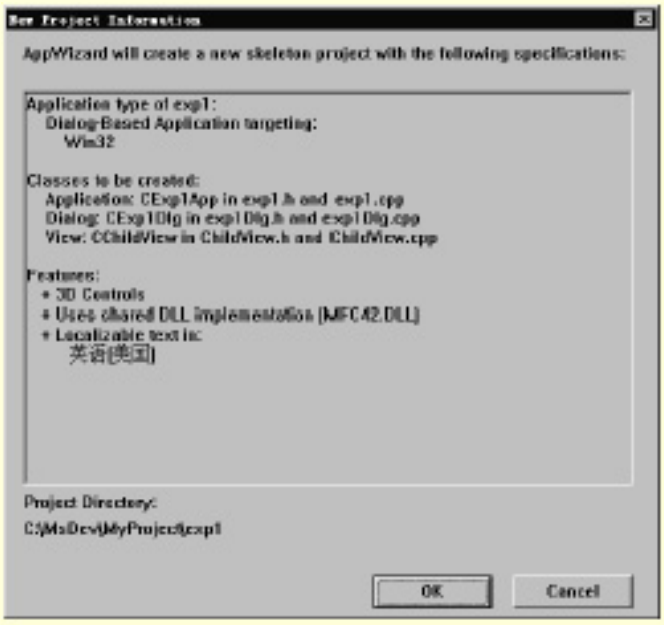

图 4-21　Exp1 项目的一些信息

4.2.2 修改项目资源

现在我们将使用资源编辑器来编辑项目资源，加入我们需要的主对话框和无模式对话框。

1．加入主对话框

按以下步骤加入主对话框：

（1）打开项目 Exp1 **的资源文件，加入一个新的对话框。**

（2）设定它的属性选项。边界为 Dialog Frame**，风格为** Popup**。另外选中** Visible **，**Titlebar**，**System Menu **几个复选框。**

（3）加入标题为 List of Strings**：的静态文本控件。**

（4）加入列表框控件。指定其标号为 IDC_LIST**。选中** Visible **、**Tab stop**、**Border**、** Sort **、**Notify**、**Vertical scroll**、** No intergral height **几个复选框。在列表** Selection **中选择** single**，在** Owner draw **中选择** No**。**

（5）加入两个按钮。使两者都具有 Visible **和** Tab Stop **的属性。一个的标题为**&Add**，标号为** IDOK**，并选择** Default button **属性，指定它为缺省按钮。另一个标题为**&Done**，标号为** IDCANCEL**。**

在按钮的标题中输入“ & **”字符，以方便用户的操作。在运行时刻，“** & **”字符后面的字符下面将出现一个下划线。用户在按下** Alt **键的同时按下相应的标有下划线的字母所在的键就可以将焦点跳转到本按钮上来。当然，所设计的跳转字符应该在整个对话框内是唯一的。，如图** 4-22 **所示，在** Add **按钮的** A **下面和** Done **按钮的** D **下面，都出现了一个下划线，给读者一个跳至该按钮的快捷方法。**

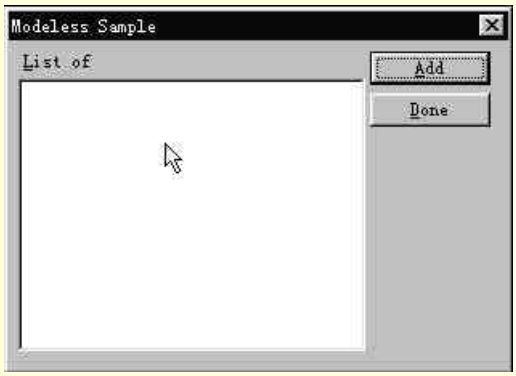

图 4-22　特殊的跳转字符

2．加入无模式对话框

按以下步骤加入无模式对话框：

（1）向项目中加入一个对话框，该对话框就是后面要用到的的无模式对话框。

（2）指定对话框的标号为 IDD_ADD_ME，标题为 Modeless Adder Dialog。

（3）设定它的属性选项。边界为 Dialog Frame，风格为 Popup。另外选中 Visible ，Titlebar，System Menu 几个复选框。

（4）加入标题为 Enter some text：的静态文本控件。

（5）加入编辑框控件。指定其标号为 IDC_NEWTEXT。选中 Visible 、Tab stop、Border、 Auto HScroll 几个复选框。在列表 AlignText 中选择 Left。

（6）加入两个按钮。使两者都具有 Visible 和 Tab Stop 的属性。一个的标题为&Add，标号为 IDOK，并选择 Default button 属性，指定它为缺省按钮。另一个标题为&Done，标号为 IDCANCEL。

4.2.3 为对话框指定新类和加入成员函数

现在我们要为新产生的两个对话框分别指定新类，并加入各自的成员函数，这样程序才能正常地工作。

1．为主对话框指定新类和加入成员函数

按以下步骤操作：

（1）在资源浏览器中选中主对话框。打开 ClassWizard， 为主对话框产生一个 CMainDialog 的新类。

（2）在 ClassWizard 的 Message Map 中，选中 CMainDialog 类。选择控件 ID 为 IDOK，双击 Message 中的 BN_CLICKED，产生名为 ONOK 的函数，使两者对应。

2．为无模式对话框指定新类和加入成员函数

按以下步骤操作：

（1）在资源浏览器中选中标号为 IDD_ADDME 对话框。打开 ClassWizard， 为主对话框产生一个 CAddDialog 的新类。

（2）在 ClassWizard 的 Message Map 中，选中 CMainDlg 类。

（3）选择控件 ID 为 IDOK，双击 Message 中的 BN_CLICKED，产生名为 OnOK 的函数，使两者对应。

（4）选择控件 ID 为 IDCANCEL，双击 Message 中的 BN_CLICKED，产生名为 OnCancel 的函数，使两者对应。

（5）选择控件 ID 为 ，双击 Message 中的 PostNcDestroy，产生名为 PostNcDestroy 的函数，使两者对应。

4.2.4 代码规整

在前面我们已经用开发环境生成了应用程序框架，下面我们将修改其中的代码，使程序能够正常的运行。

1．编辑 AddDialog.h 文件

请在 CAddDialog 类声明中加入如下的发灰代码：

```
protected:
CWnd* m_pParent;
int m_nID;
```

同时，还要加入如下的函数声明：

```
public:
CAddDialog(CWnd* pParent);      // standard constructor
BOOL Create();
```

简要对上面的代码进行说明：

- 成员变量 m_pParent 是一个指向窗口的指针，将用于存储指向主对话框的指针。
- 成员变量 m_nI 用于存储创建本对话框时要用到的对话框资源模板的标号。
- 成员函数 Create()将用于创建对话框的窗口。

2．编辑 AddDialog.cpp 文件

（1）修改 CAddDialog 类的构造函数，键入下面的代码：

程序清单 4-10 CAddDialog 函数代码

```
CAddDialog::CAddDialog(CWnd* pParent)
: CDialog(CAddDialog::IDD, pParent)
{
//{{AFX_DATA_INIT(CAddDialog)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
m_pParent= pParent;
m_nID= CAddDialog::IDD;
}
```

这个构造函数将一个指向父窗口的指针保存在成员变量 m_pParent 中，将创建对话框对象的对话框模板资源的标号存入成员变量 m_nID 中 。

（2）修改 OnOK()函数

键入下面的代码：

程序清单 4-11 OnOK 函数代码

```
void CAddDialog::OnOK()
{
CEdit* pEdit = (CEdit*) GetDlgItem(IDC_NEWTEXT);
CListBox* pList = (CListBox*) (m_pParent->GetDlgItem(IDC_LIST));

if (pList != NULL && pEdit != NULL)
   {
   CString str;
   pEdit->GetWindowText(str);
   pList->AddString(str);
   }
}
```

删掉其中的“ CDialog::OnOK(); ”，使得函数的内容与上面所示相同。

对 OnOK()函数的说明：

OnOK()函数是一个对话框类的成员函数。

- 当用户按下 OK 按钮（标号为 IDOK 的按钮）时调用本函数。
- 覆盖本函数来执行 OK 按钮的动作。如果对话框包含了自动的数据交换和有效性的检验，基类的缺省实现函数将检查对话框中数据的有效性，并将数据存入程序中对应的数据变量中。
- 如果在无模式的对话框中使用 OK 按钮，你必须覆盖基类的 OnOK 函数，并在新的 OnOK 函数中调用 DestroyWindow 来关闭对话框窗口。因为基类的 OnOK 函数将调用用于关闭有模式窗口的 EndDialog 函数，它会使对话框不可见但不会删掉它。

对函数 GetWindowText（）的说明：

- 函数 GetWindowText()是 CWnd 类的一个成员函数，其声明如下：void GetWindowText(CString& rString) const;
- 函数的调用将会把窗口的标题（如果有的话）拷进字符串对象 rString 中。当窗口对象是一个控件时，函数会把控件内的文本拷入字符串对象 rString 中。

对函数 AddString（）的说明：

- 函数 AddString（）是 ClistBox 类的一个成员函数，其声明如下：int AddString（ LPCTSTR lpszItem);
- 参数 lpszItem 为一个指向以空格结束的字符串的长指针。
- 函数的调用将 lpszItem 指向的字符串加入列表框中。若列表框为 LBS_SQRT 风格的，则函数加入字符串，然后对列表框中的条目进行排序；否则，将字符串加入到列表框的最后。

在 OnOK()函数中，首先用前面曾经提到的 GetDlgItem()取得指向子对话框的编辑框控件的指针和主对话框的列表框控件的指针。然后将编辑框控件中的字符串加入到列表框控件中。

（3）修改 OnCancel()函数

键入下面的代码：

程序清单 4-12　OnCancel 函数代码

```
void CAddDialog::OnCancel()
{
((CMainDialog*)m_pParent)->BoxDone();
DestroyWindow();
}
```

然后删除对基类的 OnCancel()函数的使用。

函数 OnCancel()的说明：

- 函数 OnCancel()是一个 CWnd 类的成员函数。当用户在对话框中单击 Cancel 按钮或者按下 ESC 键时，程序的主框架调用该成员函数。
- 一般在其中写入用于退出对话框的代码。
- 缺省的 OnCancel()函数（即基类的函数）会简单的调用用于关闭有模式对话框的 EndDialog()函数来关闭对话框。这将使对话框的窗口不可见，但不会去掉它。如果在无模式的对话框中使用，你必须编写新的 OnCancel()来覆盖缺省函数，并在函数体内调用 DestroyWindow

函数 DestroyWindow()的说明：

- 它是一个 CWnd 类的成员函数，用于关闭与 CWnd 对象对应的窗口。它还可以用于关闭用 Create()函数创建的无模式的对话框。
- DestroyWindow()在关闭成功时返回 TRUE ，否则关闭失败返回 FALSE。
- 它首先向窗口发送适当的消息使其无效并且移去输入的焦点。然后向窗口发送 WM_DESTROY 和 WM_NCDESTROY 的消息。但是，它并不删除 CWnd 对象。因此，函数 DestroyWindow()中应该被写入做一些清除工作的代码。
- 如果该窗口是一些窗口的父窗口，则在删掉该窗口时这些子窗口也被自动的删除。函数将首先删除这些子窗口，然后删除它本身。

在本例中用 CAddDialog::OnCancel()覆盖了基类的 OnCancel()函数。函数 CAddDialog::OnCancel 首先调用主对话框类的 BoxDone()函数，然后用 DestroyWindow()来关闭子对话框。

（4）手工加入 CAddDialog::Create()函数，键入下面的代码：

程序清单 4-13　Create 函数代码

```
BOOL CAddDialog::Create()
{
return CDialog::Create(m_nID, m_pParent);
}
```

函数 Create()的说明：

- 它是 CDialog 类的成员函数，有两种调用形式，其函数声明如下：BOOL Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);和 BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);
- 参数 lpszTemplateName 为字符串指针，指向对话框资源模板的名字；而参数 nIDTemplate 是对话框资源模板的标号；参数的指针，缺省时为空指针，表示它属于主框架。
- 函数 Create()使用一个对话框模板资源来创建属于 pParentWnd 指向的父窗口的无模式对话框。当对话框仍然保留在屏幕上时，控件已经立刻返回了。

用 CAddDialog::Create()函数来重载基类的 Create()函数。函数用第二种方式调用了基类的 Create()函数。

（5）编辑 CAddDialog::PostNcDestroy()函数，键入下面的代码：

程序清单 4-14 PostNcDestroy 函数代码

```
void CAddDialog::PostNcDestroy()
{
delete this;
}
```

删除函数体中的 CDialog::PostNcDestroy()语句。

对 this 指针的说明：

- this 指针是一个用于类、结构、联合三种数据类型的指针，只能在成员函数内被使用。它指向调用成员函数的对象。静态成员函数没有 this 指针。
- 当非静态的成员函数被某对象调用时，对象的地址作为隐含的参数被传递给该函数。对象的地址在成员函数内部可以从这个 this 指针中得到。
- 在指向类的成员时使用 this 指针是合法的，但是却不是必需的。

注意：

- 在新的 C++中，修改 this 指针被认为是非法的。

对函数 PostNcDestroy()的说明：

- 函数 PostNcDestroy()是 CWnd 类的一个成员函数。
- 在窗口被删除后，缺省的 OnNcDestroy 函数将会调用函数 PostNcDestroy()。派生类可以用这个函数来作一些自定义的清除工作，例如删掉 this 指针。OnNcDestroy 函数在前面曾经被提到，它会被缺省的 Destroy 函数调用。

（6）加入 CMainDialog 类的声明

由于在文件中调用了 CMainDialog 类的成员函数，我们需要在程序的前面加入下面的头文件包含语句：

```
#include "MainDialog.h"
```

这样本文件就能够找到 CmainDialog 类的声明。

3．编辑 MainDialog.h 文件

请在 CMainDialog 类声明中加入如下代码：

```
protected:
CAddDialog* m_pModeless;
```

同时，还要加入如下的函数声明：

```
public:
CMainDlg(CWnd* pParent = NULL); // standard constructor

void BoxDone();
```

简要说明如下：

- 成员变量 m_pModeless，它是一个指向属于主对话框的无模式对话框的指针。
- 成员函数 BoxDone()。它在子对话框正常退出后重置指向子对话框的指针，并使主对话框的 Add 按钮恢复有效。

由于在定义中使用了 CAddDialog 类，我们需要在程序的前面加入下面的头文件包含语句：

```
#include "AddDialog.h"
```

这样系统就能够自动识别 CAddDialog 类。

4．编辑 MainDialog.cpp 文件

（1）编辑函数 CMainDialog::CMainDialog，加入下面代码：

程序清单 4-15 CMainDialog 函数代码

```
CMainDialog::CMainDialog(CWnd* pParent /*=NULL*/)(CWnd* pParent /*=NULL*/): CDialog(CMainDialog::IDD, pParent)
{
//{{AFX_DATA_INIT(CMainDlg)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
m_pModeless = NULL;
}
```

在该构造函数中，用一个控制针来初始化指针成员变量 m_pModeless**（因为这时候尚未创建子对话框的对象）。**

（2）编辑函数 CMainDialog::OnOK()**，键入下面的代码：**

程序清单 4-16　OnOK **函数代码**

```
void CmainDialog::OnOK()
{
// if our modeless child isn't already up, create it and display it
// otherwise, just set focus to it

if (m_pModeless == NULL)
    {
    m_pModeless = new CAddDialog(this);
    if (m_pModeless->Create() == TRUE)
       GetDlgItem(IDOK)->EnableWindow(FALSE);
    }
else
    m_pModeless->SetActiveWindow();
}
```

然后删除“ CDialog::OnOK(); **”的语句。**

我们编写了自己的 OnOk **函数，它在用户按下** Add **按钮时被调用。如果无模式的对话框尚未产生，创建一个并且显示它。然后使** Add **按钮失效，以免产生多于一个的子对话框。若已经存在一个无模式的子对话框，使它获得当前的焦点。**

new **操作符通常被放在某个类的构造函数的前面 ；或者放在类名的前面，其实质也是调用构造函数。该操作符用于动态的产生某类的一个对象。它从被称为空闲内存的程序内存区内为对象分配相应的内存空间，并将指向该对象的指针返回。值得注意的是，对于由** new **操作符产生的对象，当不再使用时必须使用** delete **操作符来释放地址内存空间。否则，分配的内存将无法收回。**

EnableWindow()**函数是** CWnd **类的一个成员函数，其原型如下：**

BOOL EnableWindow(BOOL bEnable = TRUE);

- **变量** bEnable **是一个布尔变量，指定当前的窗口状态。**FALSE **表示窗口失效，**TRUE **表示窗口有效。缺省值为** TRUE**。**
- **返回值也是一个布尔变量，记录函数调用前的窗口状态。**
- **函数用来使窗口有效或失效。当窗口失效时，所有对它进行的鼠标及键盘操作都被忽略，同时它的所有子窗口也失效。另外，要激活一个窗口也必须首先使它有效。**
- **函数也被用于使对话框中的控件失效或有效。控件在失效时，无法获得输入的焦点，使得用户无法访问它。**

函数 SetActiveWindow()**是** CWnd **类的一个成员函数，其原型如下：**CWnd* SetActiveWindow( );

- **函数用于使调用该函数的窗口对象成为活动窗口。**
- **返回值为一个窗口类的指针，它指向函数调用前处于活动状态的窗口对象。该指针是暂时的，不能存**

储以供以后使用。

值得注意的是，SetActiveWindow()函数应该被谨慎地使用，因为它使应用程序能够随机地指定活动窗口和输入焦点。正常情况下，应由 Windows 来管理所有的激活行为。

（3）加入函数 CMainDialog::BoxDone()，键入下面的代码：

程序清单 4-17　BoxDone 函数代码

```
void CMainDialog::BoxDone()
{
// this function is called by the modeless dialog as it terminates
// just reset our pushbutton to be enabled again.
// I _don't_ delete the MFC CDialog object because the dialog's own
// code will do that.

m_pModeless = NULL;
// don't delete m_pModeless; !
GetDlgItem(IDOK)->EnableWindow();
}
```

在子对话框类 CAddDialog 的成员函数 OnCancel 中调用了本函数。当用户关闭子对话框时，函数重置指向子对话框的成员变量 m_pModeless，然后使主对话框中的 Add 按钮重新有效。

（4）加入 CAddDialog 类的声明

由于在文件中调用了 CAddDialog 类的成员函数，我们需要在程序的前面加入下面的头文件包含语句：

```
#include "AddDialog.h"
```

这样本文件就能够找到 CAddDialog 类的声明。

5．编辑 Exp1.cpp 文件

（1）修改 InitInstance()函数：

程序清单 4-18　缺省产生的 InitInstance 函数代码

```
BOOL CExp1App::InitInstance()
{
// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

#ifdef _AFXDLL
Enable3dControls();          // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic();    // Call this when linking to MFC statically
#endif

CExp1Dlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
```

```
// TODO: Place code here to handle when the dialog is
//  dismissed with OK
}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
//  dismissed with Cancel
}
// Since the dialog has been closed, return FALSE so that we exit the
//  application, rather than start the application's message pump.
return FALSE;
}
```

将语句 CExp1Dlg dlg;用 CMainDialog dlg 语句代替;

原来的语句将创建一个由系统指定的对话框。我们通过修改，使程序在初始化的时候创建我们定制的主对话框 CMainDialog。原有的系统创建的主对话框类 CExp1Dlg 将不起作用。

实际上，如果我们开始直接修改系统创建的主对话框资源，使它与现在的 IDD_MAIN_DIALOG 相同；并且对 CExp1Dlg 类的代码作与 CMainDialog 同样的改动。那麼在这儿就无须对 Exp1.cpp 文件和 Exp1.h 文件作改动，可以得到完全相同的结果。

（2）修改头文件的包含语句

上一步删去了有关 CExp1Dlg 的语句，加入了对 CMainDialog 类的调用。因而我们相应地用语句

```
#include MainDialog.h
```

来代替语句

```
 #include exp1Dlg.h
```

现在程序已经全部完成了。试着编译它，并运行程序。可以看出，程序的运行完全符合要求。

4.2.5 进一步理解 exp1

在代码规整时，我们已经对各个语句和函数的含义分别作了解释。下面我们将结合整个程序的理解来帮助读者进一步理解无模式对话框的编制。

在应用开始时，出现一个空的列表框。你可以单击 Add 按钮来打开一个无模态的对话框。在这个无模态的对话框打开期间，你可以将焦点带回主对话框。当无模态的对话框打开时，主对话框中的 Add 按钮被禁止使用，这样用户就不能创建无模式对话框的多于一个的实例。

主对话框类 CMainDlg 管理一个指向无模式对话框的指针。这样做是为了方便，在无模式的对话框对象被创建后，它完全能够自理而不需要主对话框的管理。

单击主对话框中的 Add 按钮将会调用 Create 函数而不是 DoModal 函数来创建无模式对话框。正是因为调用 Create 函数才使得产生的对话框为无模式的。当对话框被删除时，调用 DestroyWindow 函数。因为缺省时基类 CDialog 的成员函数 OnOk 和 OnCancel 将会调用 EndDialog 函数，所以你必须确保你的无模式对话框没有调用那些函数。相反的，你应该重载它们来调用 DestroyWindow 函数。

通常，当你创建有模式对话框时，你将在 DoModal 函数返回后手工删除它。在显示你的无模式对话框时，既然你不能等待 Create 函数返回，就需要用一些其他的机制来删除与窗口相关的 C++ 对象。在本例中，我们使用一个十分简单的机制。在消息 PostNcDestroy 对应的处理函数中删除调用本函数的对象（即当前的无模式对话框），该函数在 Destroy 函数已经删除了对话框的窗口后被调用。

这个无模式对话框通过两种不同的方式同它的父对话框通讯。首先，当用户按下 Done 按钮时，无模式对话框的编辑控件中的字符串被加入到有模式对话框的列表框中。其次，当用户通过各种途径关闭窗口时，无模式对话框调用有模式对话框的 BoxDone 函数。该函数简单地重置指向指向有模式对话框的指针，然后再次使 Add 按钮有效。

4.3 通用对话框

在 Windows 应用程序中，有很多情况下需要使用类似的对话框，如文件的打开和保存、选择程序使用的字体、选择特定的颜色等等。为了避免程序员在这些问题上耗费不必要的精力，MFC 提供了通用对话框类，封装了这些经常使用的对话框。

通用对话框类包括：

- CFileDialog 类：封装了文件操作对话框；
- CFontDialog 类：封装了字体选择对话框；
- CColorDialog 类：封装了颜色选择对话框；
- CPageSetupDialog 类：封装了页面设置对话框；
- CPrintDialog 类：封装了打印对话框；
- CFindReplaceDialog 类：封装了查找/替换对话框；
- COleDialog 类：包括更多的派生类，封装了 OLE 操作。

使用这些通用对话框是很简单的，只要生成一个相应类的对话框对象，调用 DoModal()函数显示该对话框，然后从对话框对象接收数据就可以了。这些通用对话框类都提供了许多成员函数供用户获得对话框对象的数据。

在本书后面的有关章节中，用户将会看到文件操作对话框、字体对话框、颜色对话框等通用对话框的使用。

4.4 本章小结

本章的主要内容是向用户介绍在 MFC 中应用程序中如何使用对话框和控件，这是通过建立一个完整的实例程序实现的。

“CompuInfo”程序虽然比较简单，但是该程序是用户使用对话框和控件的一个很好的起点。在“CompuInfo”程序中，详细介绍了如何建立对话框资源、如何建立对话框类、如何向对话框类中添加与控件相关联的成员变量、DDX 与 DDV、响应控件通知、与程序的其他对象交换数据等知识。可以说，“CompuInfo”程序为用户在以后的开发中使用对话框和控件建立了一个坚实的基础。

最后，本章还简要的介绍了无模式对话框和通用对话框的一些知识。

在本章所介绍的内容的基础上，用户应该可以比较自如地使用对话框和控件了，对于本书中没有介绍到的一些控件，用户可以在联机手册中得到详细信息。

第五章 图形绘制和输出

自从 Windows 操作系统诞生以来，凭借其漂亮的图形界面和友好的用户接口（当然不仅仅是凭借这两点），很快就击败了采用冷冰冰的字符界面的 DOS 操作系统，虽然后者也曾经是 Microsoft 公司的成功产品。Windows 的图形界面也给程序员丰富的想象力提供了发挥的空间，程序员可以使用各种样式的图形和文本来完成程序，即便同样是使用文本，Windows 也是“画”上去的。

当然，程序员的发挥是受到一定限制的。在 Windows 应用程序中，一般都是在窗口的客户区中使用图形和文本，以显示应用程序的数据。当然，使用对话框和控件同样也能够将程序的数据显示给用户。但是对于一个具有“窗口”的应用程序来说，不在窗口中显示点什么似乎总是一种缺憾。因此，在窗口客户区中按特定的方式绘制图形或显示文本就成为了开发 Windows 应用程序时的一项重要工作。

在 MFC 应用程序中，广泛采用了文档/视图结构。文档中保存了应用程序的数据，视图则负责将文档的数据显示出来，因此，在 MFC 应用程序中，大多数的显示工作都是在应用程序的视图类中完成的。本章中将向用户介绍如何在应用程序中使用图形或文本显示数据。

5.1 在文档窗口中绘图

对 Windows 程序而言，将程序运行的结果在文档窗口上显示出来是最自然的一种选择，本节中就将通过一个例子程序向用户介绍如何在 MFC 应用程序中实现绘图。用户将会看到，在文档窗口上绘图并不困难，基本上就像使用画笔在画布上作画那样简单。但在开始建立例子程序之前，必须对一些基本的概念进行一些解释。

5.1.1 理解设备环境

如果稍微考虑一下实际作画的过程就可以发现，有两样东西是画家所必须的：画布与画笔。在 Windows 程序中“绘图”也是类似的，用户必须有一块电子画布，一支电子画笔，然后才能开始绘图。

在 Windows 程序中，所谓的“设备环境（DC：Device Context）”就是这样的一块电子画布。值得注意的是，设备环境并不仅仅只是用户的显示器屏幕，也有可能是用户的打印机，或者是其他的输出设备。用户在绘图的时候，不需要考虑面对的是显示器设备环境或者打印机设备环境，只要保证了在设备环境上的正确绘图，Windows 系统会通过具体物理设备的驱动程序，将用户需要的图形显现出来，这就是 Windows 系统的设备无关性。

开始绘图的另外一个条件就是用户需要有一支电子画笔。在 Windows 系统中，电子画笔被称为“GDI 对象（GDI Object）”。除了画笔之外，GDI 对象还包括画刷、字体、位图和调色板等。因此，前面简单地说电子画笔可能是不准确的，应该说，GDI 对象就是那些可以用来在设备环境上绘图的工具。

在 Windows 程序中，一个设备环境只能拥有一种画笔、一种画刷和一种字体，也就是说，一个设备环境不能同时拥有同一类型的多个 GDI 对象。因此，如果用户需要使用一支粗一点的画笔，就需要先创建这样的一支画笔，然后将它选进设备环境代替原来的画笔，然后使用这支新的画笔。这就是在 Windows 应用程序中使用 GDI 对象的一般过程。

1．CDC 类

在 MFC 中，CDC 类实现了对设备环境的封装。所有的绘图操作都必须通过一个 CDC 类（或其派生类）的对象来完成。CDC 类提供了许多函数，可以完成各种与设备环境有关的操作，如选择 GDI 对象、使用颜色与调色板、绘制图形、显示字体、设置设备环境的属性和坐标转换等等。

在 MFC 基础类库中，CDC 类还有几个派生类，包括 CPaintDC 类、CClientDC 类、CWindowDC 类和

CMetaFileDC 类。

在 CDC 类的派生类中，CPaintDC 类只在响应 WM_PAINT 消息的函数中使用，大多数情况下都是 OnPaint()函数。当应用程序的窗口出于某种原因需要更新时，系统就会向应用程序发送 WM_PAINT 消息，从而调用 OnPaint()函数。但是用户不要试图去截获视图类的 WM_PAINT 消息或重载 OnPaint()函数，而应该在 OnDraw()函数中完成绘图任务。这是因为 CView 类已经完成了对 WM_PAINT 消息的捕获，并在 OnPaint()函数中完成了特定了工作。

OnPaint()函数在最后调用了 OnDraw()函数，并且向 OnDraw()传递了一个 CDC 类对象的指针，用户可以通过该指针完成各种绘图操作。这里需要提醒用户的是，并不是一定要在 OnDraw()函数完成绘图操作，在其他函数中同样是可以的，本节的例子程序就绕过了 OnDraw()函数，在其他的消息处理函数中完成了绘图操作。

CClientDC 类可能是使用最多的 CDC 的派生类，正如其类名所指出，CClientDC 类代表了应用程序窗口的客户区。因此，所有使用 CClientDC 类对象完成的绘图操作都位于窗口的客户区内。

CWindowDC 类封装了与整个窗口有关的设备环境，包括窗口的客户区和非客户区；CMetaFileDC 类封装了与 Windows 元文件有关的绘图操作。用户可以通过联机手册获得这两个派生类的详细信息。

前面已经指出，在使用设备环境之前，必须构造一个 CDC 类（或其派生类）的对象。在 MFC 应用程序中，大部分的绘图操作是针对应用程序窗口客户区的，因此，大多数情况下都使用了 CClientDC 类的对象。

有两种方法可以建立一个 CClientDC 对象，最简单的是这样：

```
CClientDC dc(this)
```

这样的一条语句在程序的堆栈上建立了一个 CClientDC 对象，该对象与当前使用的窗口的客户区相关联。在本节的例子程序中，都是使用这种方式建立 CClientDC 对象的。

另外一种建立 CClientDC 对象是调用 GetDC()函数，该函数的返回值虽然是一个 CDC 类对象的指针，但是实际该对象所代表的是窗口的客户区。需要注意的是，在 CDC 类对象使用结束后，需要调用 ReleaseDC()函数释放设备环境。

2．CGdiObject 类

在 MFC 中，CGdiObject 类封装了 GDI 对象。但是，用户几乎从来不需要直接使用 CGdiObject 类，而是使用该类的派生类。CGdiOjbect 类的派生类包括：CPen 类、CBrush 类、CFont 类、CBitmap 类、CPalette 类和 CRgn 类。

CPen 类封装了 GDI 画笔对象，在 Windows 中，画笔用来绘制各种线条，包括直线、曲线以及各种封闭图形的边框；CBrush 类封装了 GDI 画刷对象，画刷用来填充矩形、椭圆等各种封闭图形的内部区域；CFont 类封装了 GDI 字体对象，字体用来显示文本。

CBitmap 类封装了 GDI 位图对象，并提供了相应的成员函数对位图进行操作，在本书的第四章的“ NewCtrl ”程序中已经使用过了 CBitmap 类；CPalette 类封装了 Windows 调色板，调色板用来在用户需要的颜色和系统能够提供的颜色之间进行协调；CRgn 类封装了 GDI 区域，区域是窗口中特定的一块，通常用来指定操作的范围，以免干扰其他不需要修改的区域。

在本节的例子程序中，将介绍画笔对象和画刷对象的使用方法。在下一章的例子程序中，介绍了一些使用字体对象进行文本输出的知识。

在上面的叙述中，简要介绍了在 MFC 应用程序中实现屏幕输出的 CDC 类和 CGdiObject 类（及其派生类）的一些基础知识。在本节的例子程序中，将向用户演示使用 CDC 类和 CGdiObject 类对象的具体方法。

5.1.2 建立新的项目

在本节中使用的“ Draw ”例子程序，将向用户介绍的内容包括：如何建立和使用设备环境对象、如何创建和使用不同的画笔和画刷、如何响应鼠标消息等等。在“ Draw ”程序中，OnDraw()函数没有承担任何绘图工作，所有的绘图工作都是在相应的消息响应函数中完成的。用户只要理解了设备环境的工作过程，自然也就能够在 OnDraw()函数中进行绘图工作。

在“ Draw ”程序中，用户可以作出相当多的绘图方式选择：

- 选择绘图功能：绘直线、矩形或者是随手画；
- 选择画笔的线宽：1 个像素宽、2 个像素宽或者 5 个像素宽；

- 选择画笔的线型：实线、虚线或者点线；
- 选择画刷的类型：空画刷、白色画刷、垂直条纹画刷或者是斜条纹的画刷；
- 选择画笔和画刷使用的颜色。

此外，“Draw”程序还提供了图形拉伸的功能。本节将向用户展示如何逐步实现这些功能。

建立“Draw”程序的工作与前面的例子程序没有什么不同。在“File”菜单下选择“New”命令，建立新的 MFC 应用程序项目“Draw”。在 AppWizard 中，在步骤一中选择“单文档界面”，选择“英语（美国）”为应用程序资源的语言类型；在步骤二中接受缺省设置；在步骤三中清除“ActiveX Controls”选项；在步骤四中清除“Print and print preview”选项；在步骤五、六中保持缺省设置。最后，单击“Finish”按钮完成应用程序的设置，并生成“Draw”程序的框架代码。

5.1.3 实现绘图功能

对于“Draw”程序，首要的任务是实现计划中的三种绘图功能：画直线、画矩形和随手画。用户或许期待着能够绘制更多类型的图形，如椭圆、多边形等等，但是作为一个例子程序而言，实现这三种绘图功能已经足够了。用户如果真正理解了实现这些功能的过程，其他的绘图功能也不是什么困难的事情，只要调用合适的函数就可以了。

1. 选择绘图功能

在“Draw”程序中，每次只能选择和使用一种绘图功能，因此必须提供给用户在绘图功能中进行切换的方法。“Draw”程序通过菜单提供了选择绘图功能的方法。为了让程序能够识别当前使用的功能，“Draw”程序采用了一种比较“笨”的方法，就是对每一种绘图功能相应地设置了一个标志变量，在程序运行的过程中检测相应的变量是否为“TRUE”就可以了。这有些类似于本书第三章“AlignMode”程序中选择水平对齐方式的处理方法，虽有重复累赘之嫌，对于这种小的例子程序还是适用的。

“Draw”程序需要在“Edit”菜单和“View”菜单之间增加一个新的下拉菜单“Draw”，在该下拉菜单中提供了菜单项供用户选择当前的绘图功能，最后程序运行时的“Draw”菜单如图 5-1 所示。

图 5-1 “Draw”菜单

表 5-1 中列出了各菜单项的需要设置的属性。

表 5-1 “Draw”菜单中的菜单项

命令 ID	标题文本	提示信息
ID_DRAW_LINE	Line	Draw a line.
ID_DRAW_RECT	Rectangle	Draw a rectangle.
ID_DRAW_SKETCH	Sketch	Sketch as you want.

相应地，在 CDrawView 类中需要添加三个 BOOL 类型的成员变量，如下所示：

```
protected:
    BOOL m_bLine;          // 绘直线功能标志
    BOOL m_bRect;          // 绘矩形功能标志
    BOOL m_bSketch;        // 随手画功能标志
```

用户需要使用 ClassWizard 对“Draw”菜单下各菜单项的 COMMAND 消息和 UPDATE_COMMAND_UI 消息生成消息处理函数，在这些消息处理函数中，需要对上述标志变量进行相应的设置。程序清单 5-1 中列出了这些函数的代码，这其中还包括 CDrawView 类的构造函数，在构造函数中对上述标志变量进行了初始化。

程序清单 5-1 设置标志变量

```
CDrawView::CDrawView()
```

```
{
// TODO: add construction code here
m_bLine=TRUE;
m_bRect=FALSE;
m_bSketch=FALSE;
}

void CDrawView::OnDrawLine()
{
// TODO: Add your command handler code here
m_bLine=TRUE;
m_bRect=FALSE;
m_bSketch=FALSE;
}

void CDrawView::OnDrawRect()
{
// TODO: Add your command handler code here
m_bLine=FALSE;
m_bRect=TRUE;
m_bSketch=FALSE;
}

void CDrawView::OnDrawSketch()
{
// TODO: Add your command handler code her
m_bLine=FALSE;
m_bRect=FALSE;
m_bSketch=TRUE;
}

void CDrawView::OnUpdateDrawLine(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bLine);
}

void CDrawView::OnUpdateDrawRect(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bRect);
}

void CDrawView::OnUpdateDrawSketch(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
```

```
pCmdUI->SetCheck(m_bSketch);
}
```

在程序清单 5-1 中的代码都很简单，因此没有添加任何注释，用户应该可以看懂。这些函数只是完成了设置各标志变量的任务，真正的绘图功能的实现还需要更多的努力。

2．实现绘直线与绘矩形功能

在很多绘图程序中，最简单的如 Windows 系统附带的“ PaintBrush ”程序，都是使用鼠标进行操作的。在这些程序中，只要选择了相应的功能，通过鼠标的“ 拖曳 ”就可以完成任务。“ Draw ”程序也将使用鼠标完成绘图操作。

用户已经知道，一条直线需要两点才能确定，在 MFC 应用程序中也不例外。设想一下具体的操作就可以发现，按下鼠标左键的位置应该是直线的起点，松开鼠标左键的位置则是直线的终点。因此，需要响应鼠标的 WM_LBUTTONDOWN 和 WM_LBUTTONUP 消息，并在相应的消息处理函数中记录直线的起点和终点，完成画线任务。

为了记录直线的起点和终点，还需要向 CDrawView 类添加两个成员变量，如下所示：

```
protected:
    CPoint m_LastPoint;          //上一点的位置
    CPoint m_CurPoint;           //当前点的位置
```

用户可能已经注意到这两个成员变量的名称和注释，并没有使用“ 起点 ”、“ 终点 ”之类的名称，这是因为这两个变量还将被其他的绘图功能所使用，而其他的功能中不一定有“ 起点 ”和“ 终点 ”这样的说法。CPoint 类的两个成员 cx 和 cy 保存了点在 x 方向和 y 方向上的坐标。

用户可以使用 ClassWizard 或者 WizardBar 生成对鼠标消息的处理函数，由 CDrawView 类接收鼠标消息，并接受缺省的函数名：OnLButtonDown()和 OnLButtonUp()。程序清单 5-2 中列出了这两个函数的代码。

程序清单 5-2　OnLButtonDown()和 OnLButtonUp()

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 记录上一点的位置
m_LastPoint=point;
CView::OnLButtonDown(nFlags, point);
}

void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 绘直线
if(m_bLine)
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}
```

```
// 绘矩形
if(m_bRect)
{
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
}
CView::OnLButtonUp(nFlags, point);
}
```

OnLButtonDown()的参数有两个，nFlags 参数中包含了一些当前输入设备的信息，这里不需要；point 参数中保存了 WM_LBUTTONDOWN 事件发生时鼠标的位置，这也正是需要保存的直线的“起点”。

OnLButtonUp()函数中，首先建立了一个客户区设备环境对象，注意传递给 CClientDC 对象构造函数的参数“this”。在 C++语言中，“this”关键字只能在类、结构或联合中使用，代表的就是调用该成员函数的对象。在这里，OnLButtonUp()函数是 CDrawView 类的一个成员函数，因此，“this”代表的是一个 CDrawView 类对象，即应用程序的视图。因此，最终建立的设备环境就是视图的客户区的设备环境，所有的绘图操作都显示在视图的客户区中。

在 OnLButtonUp()函数中，接着保存了直线的“终点”。在对当前选中的绘图功能进行判断后，调用了 CClientDC 类的 MoveTo()函数将设备环境的当前位置移动到直线的“起点”，然后调用 LineTo()函数从“起点”向“终点”画了一条直线。这样就实现了绘直线的功能。

绘矩形的代码是类似的，只不过 Rectangle()函数可以直接接受两个点的坐标作为参数，因此不需要进行当前点的移动而已。

3. 实现随手画功能

与实现绘直线和绘矩形相比，实现随手画功能要复杂一点。实际上，随手画过程就是用一小段一小段的直线跟踪鼠标的移动过程。试设想一下具体的操作过程：按下鼠标左键，随手画过程开始；在鼠标的移动过程中始终按下鼠标左键保持随手画过程；松开鼠标左键，结束随手画过程。

因此，实现随手画同样需要响应鼠标的 WM_LBUTTONDOWN 和 WM_LBUTTONUP 消息，同时还要响应 WM_MOUSEMOVE 消息。注意并不是鼠标每移动一个像素的距离系统就会发送一次 WM_MOUSEMOVE 消息，而是鼠标每移动一小段距离才发送一次 WM_MOUSEMOVE 消息。

对随手画功能来说，OnLButtonDown()函数中只要记录下随手画开始的位置就可以了，因此现有的代码已经足够；但是在 OnLButtonUp()函数和 OnMouseMove()函数中都需要增加一些代码，如程序清单 5-3 所示。

程序清单 5-3 OnLButtonUp()和 OnMouseMove()

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;
……
 //此处代码略
// 随手画
if(m_bSketch)
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}
```

```
CView::OnLButtonUp(nFlags, point);
}

void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 随手画
if(m_bSketch && ((nFlags & MK_LBUTTON)!=0) )
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
    m_LastPoint=m_CurPoint;
}
CView::OnMouseMove(nFlags, point);
}
```

在 OnMouseMove()函数中，判断随手画的条件除了标志变量 m_bSketch 为“ TRUE ”外，还需要判断鼠标的左键是否被按下。OnMouseMove()函数的参数 nFlags 中就记载了有关的信息，该参数可能取的标志位如表 5-2 所示。

表 5-2 nFlags 参数

可能的标志	说 明
MK_CONTROL	键盘的“ Ctrl ”键被按下
MK_LBUTTON	鼠标的左键被按下
MK_MBUTTON	鼠标的中键被按下
MK_RBUTTON	鼠标的右键被按下
MK_SHIFT	键盘的“ Shift ”键被按下

在 OnMouseMove()函数中，使用了按位与操作符“ & ”判断 nFlags 参数的取值。一旦判定当前确实在使用随手画功能，就从“ 上一点 ”向“ 当前点 ”画一小段直线，然后再将“ 当前点 ”的值赋予“ 上一点 ”，这是一种普遍的做法。

现在用户已经初步完成了“ Draw ”程序中绘直线、绘矩形和随手画的功能，不妨编译并运行一下该程序，检验程序的基本功能。如图 5-2 所示。

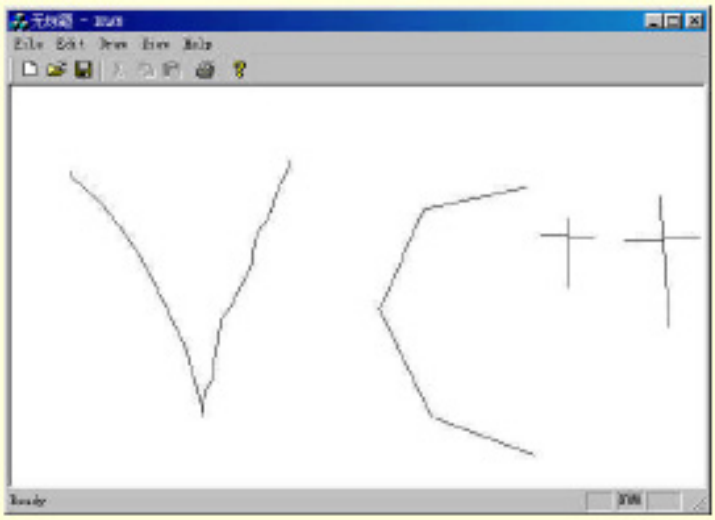

图 5-2 简单的绘画效果

5.1.4 画笔和画刷

在前面的绘图过程中，使用了系统缺省的画笔和画刷，用户完全可以按自己的要求定制画笔和画刷，以达到满意的效果。

对一支画笔而言，有三个方面的效果可以进行选择：画笔的宽度、画笔的风格和画笔的颜色。Windows 系统已经为用户提供了一些画笔风格，表 5-3 中列出了其中比较常用的一些画笔风格。

表 5-3 常用的画笔风格

风 格	说 明
PS_SOLID	建立一支实心画笔
PS_DASH	建立一支虚线画笔，只在画笔宽度不大于 1 时有效
PS_DOT	建立一支点线画笔，只在画笔宽度不大于 1 时有效
PS_DASHDOT	建立一支点划线画笔，只在画笔宽度不大于 1 时有效
PS_USERSTYLE	画笔的风格由用户定制
PS_ENDCAP_ROUND	画笔的末端为圆形
PS_JOIN_ROUND	线的交点为圆形

表 5-3 中的说明已经表明，某些风格的画笔没有宽度限制，而另外一些风格的画笔只在画笔宽度不大于 1 时才有效，这需要用户在建立新的画笔时有所考虑。

表 5-3 中只列出了几种比较常用的画笔风格，用户可以从联机手册中得到更详细的信息。在“Draw”程序，将使用表 5-3 中的前面几种风格：PS_SOLID、PS_DASH 和 PS_DASHDOT 风格的画笔。

对于画刷而言，情况似乎要复杂一些，因为画刷分为两种：实心的画刷和带阴影的画刷。实心的画刷只需要选择画刷的颜色即可，而带阴影的画刷需要选择画刷阴影的风格和画刷使用的颜色。表 5-4 中列出了常用的画刷阴影风格。

表 5-4 常用的画刷阴影风格

阴影风格	说 明
HS_BDIAGONAL	阴影线为+45°斜线
HS_CROSS	阴影线为交叉的水平线和垂直线
HS_DIAGCROSS	阴影线为斜向交叉的直线
HS_FDIAGONAL	阴影线为-45°斜线
HS_HORIZONTAL	阴影线为水平直线
HS_VERTICAL	阴影线为垂直斜线

为简单起见，“Draw”程序中只选择了有代表性的几种画刷风格。

1. 设置画笔和画刷

与选择程序的绘图功能类似，“Draw”程序通过菜单提供了选择画笔宽度和风格的途径。为此需要新建

“ Options ”菜单，提供各种可能的选项。完成后的“ Options ”菜单如图 5-3 所示。

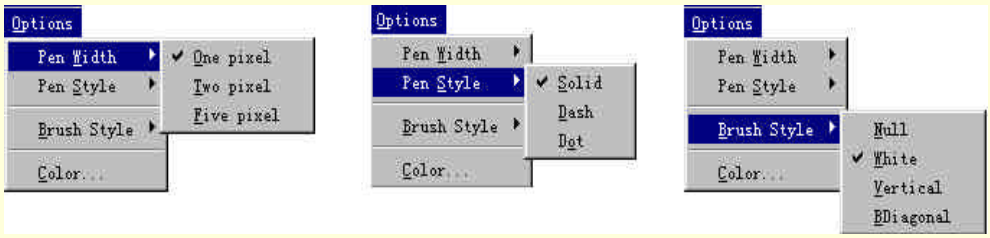

图 5-3 “ Options ”菜单的各级子菜单

从图 5-2 中可以看出，“ Options ”菜单中还包含了一些子下拉菜单：“ Pen Width ”子菜单中可以选择画笔的宽度，“ Pen Style ”子菜单中可以选择画笔的风格，“ Brush Style ”子菜单中可以选择画刷的风格。本步骤中将实现用户界面上的“ 选择 ”，在下一步骤中将介绍如何真正地实现在绘图使用特定宽度和风格的画笔和画刷。

另外，选择“ Color ”菜单项将弹出颜色对话框，供用户选择画笔和画刷使用的颜色。这也是下一步骤中将要介绍的内容。

表 5-5 中列出了各菜单项的有关信息。

表 5-5 “ Options ”菜单中的菜单项

命令 ID	标题文本	提示信息
ID_PENWIDTH_ONE	One pixel	Select a one-pixel wide pen.
ID_PENWIDTH_TWO	Two pixels	Select a two-pixels wide pen.
ID_PENWIDTH_FIVE	Five pixels	Select a five-pixel wide pen.
ID_PEN_SOLID	Solid	Select a solid pen.
ID_PEN_DASH	Dash	Select a dash pen.
ID_PEN_DOT	Dot	Select a dot pen.
ID_BRUSH_NULL	Null	Select a null brush.
ID_BRUSH_WHITE	White	Select a white brush.
ID_BRUSH_VERTICAL	Vertical	Select a brush with vertical hatch.
ID_BRUSH_BDIAGONAL	BDiagonal	Select a downward diagonal brush.
ID_OPTION_COLOR	Color...	Select color.

同样地，在 CDrawView 类必须设置一些变量来记录画笔的宽度、画笔的风格和画刷的风格，还必须对菜单的显示状态进行更新。为此在 CDrawView 类中添加了新的变量，如下所示：

```
protected:
    UINT m_nPenStyle;
    UINT m_nBrushStyle;
    UINT m_nPenWidth;
```

用户需要使用 ClassWizard 生成“ Options ”菜单下各菜单项命令的处理函数，以及更新显示命令的处理函数。程序清单 5-4 中列出了这些函数的代码。也包括了 CDrawView 类的构造函数，其中对新添加的变量实现了初始化。

程序清单 5-4 设置选项变量

```
CDrawView::CDrawView()
{
……//其余代码略
m_nPenWidth=1;
m_nPenStyle=PS_SOLID;
m_nBrushStyle=WHITE_BRUSH;
}

void CDrawView::OnPenwidthOne()
```

```
{
// TODO: Add your command handler code here
m_nPenWidth=1;
}

void CDrawView::OnPenwidthTwo()
{
// TODO: Add your command handler code here
m_nPenWidth=2;
}

void CDrawView::OnPenwidthFive()
{
// TODO: Add your command handler code here
m_nPenWidth=5;
}

void CDrawView::OnUpdatePenwidthOne(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nPenWidth==1);
}

void CDrawView::OnUpdatePenwidthTwo(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nPenWidth==2);
}

void CDrawView::OnUpdatePenwidthFive(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nPenWidth==5);
}

void CDrawView::OnPenSolid()
{
// TODO: Add your command handler code here
m_nPenStyle=PS_SOLID;
}

void CDrawView::OnPenDash()
{
// TODO: Add your command handler code here
m_nPenStyle=PS_DASH;
}
```

```
void CDrawView::OnPenDot()
{
// TODO: Add your command handler code here
m_nPenStyle=PS_DOT;
}

void CDrawView::OnUpdatePenSolid(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(m_nPenWidth==1);
pCmdUI->SetCheck(m_nPenStyle==PS_SOLID);
}

void CDrawView::OnUpdatePenDash(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(m_nPenWidth==1);
pCmdUI->SetCheck(m_nPenStyle==PS_DASH);
}

void CDrawView::OnUpdatePenDot(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(m_nPenWidth==1);
pCmdUI->SetCheck(m_nPenStyle==PS_DOT);
}

void CDrawView::OnBrushNull()
{
// TODO: Add your command handler code here
m_nBrushStyle=NULL_BRUSH;
}

void CDrawView::OnBrushWihte()
{
// TODO: Add your command handler code here
m_nBrushStyle=WHITE_BRUSH;
}

void CDrawView::OnBrushVertical()
{
// TODO: Add your command handler code here
m_nBrushStyle=HS_VERTICAL;
}
void CDrawView::OnBrushBdiagonal()
{
```

```
// TODO: Add your command handler code here
m_nBrushStyle=HS_BDIAGONAL;
}
void CDrawView::OnUpdateBrushNull(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nBrushStyle==NULL_BRUSH);
}

void CDrawView::OnUpdateBrushWihte(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nBrushStyle==WHITE_BRUSH);
}

void CDrawView::OnUpdateBrushVertical(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nBrushStyle==HS_VERTICAL);
}

void CDrawView::OnUpdateBrushBdiagonal(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nBrushStyle==HS_BDIAGONAL);
}
```

程序清单 5-4 **中的代码保证用户在“** Options **”菜单中作出选择后，画笔或画刷的风格立即改变，并相应地更新菜单的显示状态。**

2．**创建和使用新的画笔**

在 Windows **应用程序中，“画笔”用来画线，包括直线、各种简单曲线，以及封闭图形的边框等。因此在“** Draw **”程序中，新风格的画笔对于绘直线、绘矩形和随手画都是有用的。**

创建一支新的画笔并不困难，通常都是这样进行创建的：

```
CPen myPen;
myPen.CreatePen(PS_SOLID,1,RGB(0,0,255));
```

这种创建方法分为两步：首先构造了一支新的画笔，然后调用 CreatePen()**函数对新的画笔进行了初始化。**CreatePen()**函数的三个参数依次指定了新画笔的风格、宽度和颜色。**

在指定画笔颜色的时候，使用了 RGB **宏。**RGB **宏构造一种新的颜色，该宏的三个参数分别为新颜色的红、绿、蓝三原色的比重值，取值范围为** 0~255**。比如说，**RGB(255,0,0)**构造出纯红色，**RGB(255,255,255)**构造出白色，**RGB(0,0,0)**则构造出黑色。**

因此上面所创建的是一支实心的、 1 **个像素宽的蓝色画笔。**

创建画笔更简单的方法是一步构造：

```
CPen myPen(PS_SOLID,1,RGB(0,0,255))
```

得到的结果是相同的。

在创建了新的画笔之后，需要将新画笔选进当前的设备环境中才能使用。这由 CDC **类的** SelectObject()**函数完成。**SelectObject()**函数有好几种版本，分别对应于不同类型的** GDI **对象，如画笔、画刷、字体等等。**

SelectObject()**函数返回设备环境原来使用的** GDI **对象，通常情况下都要将该对象保存起来，在使用新的对**

象结束后，再将原 GDI 对象选进设备环境。这样做的目的是尽量恢复原设备环境的情形，用户在使用自己定制的 GDI 对象时一定要这样做。

因此使用新的画笔的过程通常都是这样的：

```
CClientDC dc(this);
CPen myPen;
myPen.CreatePen(PS_SOLID,1,RGB(0,0,255));
CPen* pOldPen=dc.SelectObject(&myPen);
//...Do some drawing
dc.SelectObject(pOldPen);
```

在“Draw”程序中也不例外，程序清单 5-5 中列出了 OnLButtonUp()函数和 OnMouseMove()函数的代码，正是这两个函数完成了绘图任务，因此需要定制新的画笔。

程序清单 5-5　OnLButtonUp()和 OnMouseMove()

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 创建新画笔
CPen myPen;
if(m_nPenWidth==1)
{
    myPen.CreatePen(m_nPenStyle,1,RGB(0,0,0));
}
else
{
    myPen.CreatePen(PS_SOLID,m_nPenWidth, RGB(0,0,0));
}
CPen* pOldPen=dc.SelectObject(&myPen);

// 绘直线
if(m_bLine)
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}

// 绘矩形
if(m_bRect)
{
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
}

// 随手画
```

```
if(m_bSketch)
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}

// 恢复原画笔
dc.SelectObject(pOldPen);
CView::OnLButtonUp(nFlags, point);
}

void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 创建新画笔
CPen myPen;
if(m_nPenWidth==1)
{
    myPen.CreatePen(m_nPenStyle,1, RGB(0,0,0));
}
else
{
    myPen.CreatePen(PS_SOLID,m_nPenWidth, RGB(0,0,0));
}
CPen* pOldPen=dc.SelectObject(&myPen);

// 随手画
if(m_bSketch && ((nFlags & MK_LBUTTON)!=0) )
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
    m_LastPoint=m_CurPoint;
}

// 恢复原画笔
dc.SelectObject(pOldPen);
CView::OnMouseMove(nFlags, point);
}
```

在程序清单 5-5 中，用户已经确实体会到了使用定制 GDI 对象的过程，使用画笔是这样，使用其他 GDI 对象也没有什么大的不同。

前面已经提到，某些特定风格的画笔只在画笔宽度为 1 时有效，因此在创建新画笔的时候需要加以区分：如果画笔宽度是 1，则创建一支符合风格要求的画笔；否则只是简单地创建一支指定宽度的实心画笔。如果用

户回过头看一看程序清单 5-4 中的 OnUpdatePenXXX()函数，就会看到只有画笔宽度为 1 时，画笔风格菜单项才被设置为有效，也就是说，只有画笔宽度为 1 时，才能选择画笔的风格。

3. 创建和使用新的画刷

在 Windows 应用程序中，所有需要“填充”的地方都需要使用画刷。在“Draw”程序中，只有绘矩形时才需要对内部进行填充，这是才需要使用画刷，因此只有 OnLButtonUp()函数需要进行修改。程序清单 5-6 中列出了这一小段代码。

程序清单 5-6 OnLButtonUp()中定制新的画刷

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;
……
// 绘矩形
if(m_bRect)
{
    // 使用系统画刷
    if(m_nBrushStyle==NULL_BRUSH || m_nBrushStyle==WHITE_BRUSH)
    {
    dc.SelectStockObject(m_nBrushStyle);
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
    }
    // 使用定制画刷
    else
    {
    CBrush myBrush;
    myBrush.CreateHatchBrush(m_nBrushStyle,RGB(0,0,0));
    CBrush* pOldBrush=dc.SelectObject(&myBrush);
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
    dc.SelectObject(pOldBrush);
    }
}
……
CView::OnLButtonUp(nFlags, point);
}
```

在 OnLButtonUp()函数中，对新画刷的使用也做了区分：使用系统画刷和使用定制画刷。这是因为 NULL_BRUSH 和 WHITE_BRUSH 是系统已经预定义的，可以直接使用。除了这两种特定的画刷外，还有一些可以直接使用的系统 GDI 对象，表 5-6 中列出了比较常用的一些。

表 5-6 常用的系统 GDI 对象

GDI 对象	说 明
NULL_BRUSH	空画刷
WHITE_BRUSH	白色实心画刷
BLACK_BRUSH	黑色实心画刷

NULL_PEN	空画笔
WHITE_PEN	白色画笔
BLACK_PEN	黑色画笔
SYSTEM_FONT	当前系统字体
ANSI_VAR_FONT	ANSI 可变字体
DEFAULT_PALETTE	系统缺省调色板

与使用用户自己定制的 GDI 对象不同，使用系统 GDI 对象由 SelectStockObject()函数调用，而且一般也不需要恢复设备环境原先的设置。

这里有必要解释一下空画刷和其他画刷的区别。使用其他画刷时，封闭区域的内部将由特定的颜色或阴影进行填充，因此，如果在填充区域内的其他图案将被覆盖；而使用空画刷时，区域内部不填充，因此可以保留原来的图案。图 5-4 清楚地表明了使用空画刷和白色画刷的区别。

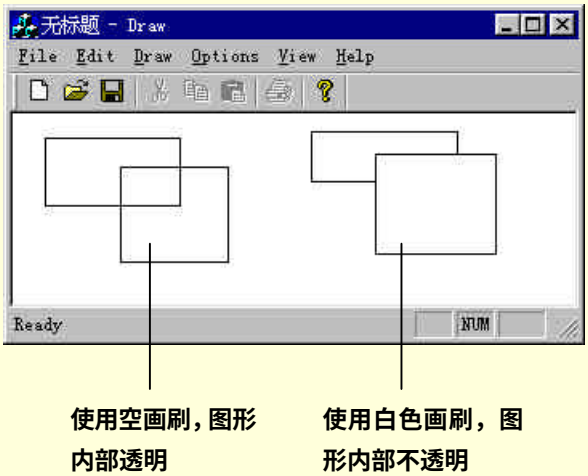

图 5-4 使用空画刷与白色画刷的区别

4. 选择特定的颜色

在前面创建新的画笔和画刷的时候，使用的是固定的颜色：RGB(0,0,0)，即黑色。实际上，在创建画笔和画刷时是可以指定颜色的。下面就来实现这一目标。

在第三章中简要介绍通用对话框时已经提到，MFC 用 CColorDialog 类封装了颜色选择对话框，使用该对话框完成颜色选择是很方便的。

MFC 使用 COLORREF 类型保存颜色信息，实际上是一个 32 位的值，但用户直接使用该类型可能不太方便，因此提供了 RGB 宏由三原色分量合成 COLORREF 值。

在“Draw”程序的 CDrawView 类中需要添加一个新的变量保存用户已经选择的颜色值，该变量的完整声明为：

```
protected:
    COLORREF m_CurColor;
```

在 CDrawView 类的构造函数中需要对该变量进行初始化，添加下面一行就可以了：

```
    m_CurColor=RGB(0,0,0);
```

在 OnLButtonUp()函数和 OnMouseMove()函数中创建新的画笔和画刷时，需要用成员变量 m_CurColor 替代 RGB(0,0,0)。这里列出详细的程序代码，用户可以按照下面的代码要求自行进行替换即可。

程序清单 5-7 CDrawView::OnOptionColor()

```
void CDRAWView::OnLButtonUp(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
```

```
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 创建新画笔
CPen myPen;
if(m_nPenWidth==1)
{
    myPen.CreatePen(m_nPenStyle,1,m_CurColor);
}
else
{
    myPen.CreatePen(PS_SOLID,m_nPenWidth, m_CurColor);
}
CPen* pOldPen=dc.SelectObject(&myPen);

// 绘直线
if(m_bLine)
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}

// 绘矩形
if(m_bRect)
{
    // 使用系统画刷
    if(m_nBrushStyle==NULL_BRUSH || m_nBrushStyle==WHITE_BRUSH)
    {
    dc.SelectStockObject(m_nBrushStyle);
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
    }
    // 使用定制画刷
    else
    {
    CBrush myBrush;
    myBrush.CreateHatchBrush(m_nBrushStyle,m_CurColor);
    CBrush* pOldBrush=dc.SelectObject(&myBrush);
    dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
    dc.SelectObject(pOldBrush);
    }
}

// 随手画
if(m_bSketch)
```

```
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
}

// 恢复原画笔
dc.SelectObject(pOldPen);

CView::OnLButtonUp(nFlags, point);
}

void CDRAWView::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default

// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 创建新画笔
CPen myPen;
if(m_nPenWidth==1)
{
    myPen.CreatePen(m_nPenStyle,1, m_CurColor);
}
else
{
    myPen.CreatePen(PS_SOLID,m_nPenWidth, m_CurColor);
}
CPen* pOldPen=dc.SelectObject(&myPen);

// 随手画
if(m_bSketch && ((nFlags & MK_LBUTTON)!=0) )
{
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
    m_LastPoint=m_CurPoint;
}

// 恢复原画笔
dc.SelectObject(pOldPen);

CView::OnMouseMove(nFlags, point);
}
```

重要的工作是实现颜色选择对话框，并能够从该对话框中接受用户选择的颜色。实际上使用颜色选择对话

框与使用一般的有模式对话框没有什么大的不同，并且 CColorDialog 类已经提供了一些成员函数供用户从对话框中获取信息。

下面是 OnOptionColor()函数的代码。

```
void CDrawView::OnOptionColor()
{
// TODO: Add your command handler code here
// 使用颜色选择对话框
CColorDialog dlg;
if(dlg.DoModal()==IDOK)
{
    m_CurColor=dlg.GetColor();
}
}
```

用户已经看到，OnOptionColor()函数的代码是很简单的，但是这样简单的代码已经足够完成选择颜色的任务了，这是因为 MFC 已经为用户做了大量的工作。

至此“ Draw ”程序已经能够实现用户选择不同画笔和画刷的任务了，用户可以编译并运行程序，检查程序的各项功能。如图 5-5 所示为在绘制矩形时，设置了不同笔刷和填充图案以及画笔颜色的绘画效果。

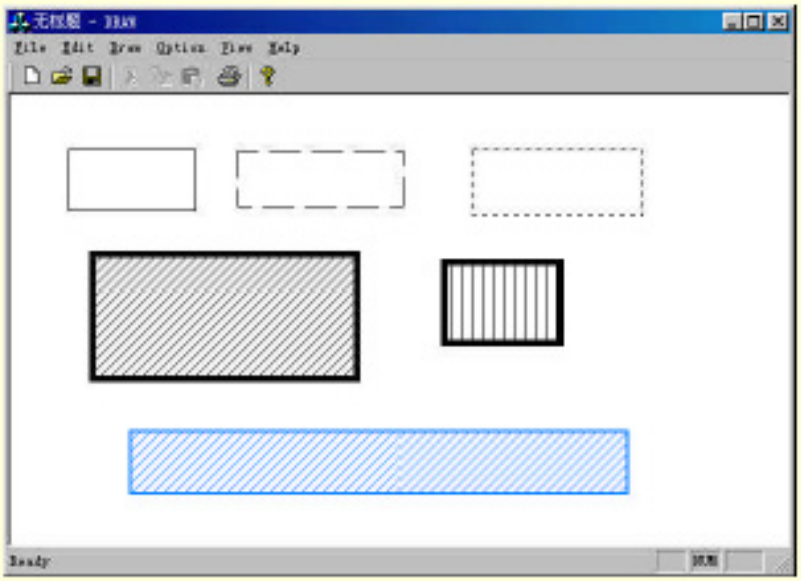

图 5-5 程序运行的效果

5.1.5 实现图形拉伸

在作为商业产品出售的绘图程序中，使用鼠标拖曳出直线或矩形时，一般都有相应的图形随鼠标的移动而不断改变大小与位置，这就是图形拉伸功能。在“ Draw ”程序中，也可以实现这一功能。

以直线的拉伸为例，实现过程的基本思路是这样的：从直线的起点开始，当用户移动鼠标时始终从直线的起点向当前的鼠标位置画一条线，但是在画新的直线之前需要擦除上一次画的直线。显示给用户的效果就是图形在不断被拉伸。矩形的拉伸也是类似的，只不过是不断地擦除矩形和绘制新的矩形而已。

擦除原有图案的最好的方法莫过于用相反的颜色在原来的位置上重新绘制一次该图案，但是考虑到“ Draw ”程序中可以选择各种不同的颜色，拉伸的图形可能经过了好几种不同颜色的区域，为了不影响已经存在的各种图案，需要做一点额外的工作。

用户可以通过控制画笔颜色和当前显示颜色的光栅操作方式以避免影响原有的图案。CDC 类提供了 GetROP2()函数和 SetROP2()函数获取或设置当前的光栅操作模式。用户可以选择的光栅操作模式达 16 种之多，在联机手册的关于 CDC::SetROP2()函数的解释中有详细的信息。这里只要简单地使用 R2_NOTXORPEN 模式就

可以了。

为了记录上一次鼠标的位置，需要向 CDrawView 类中添加一个新的成员变量：

```
protected:
    CPoint m_OldPoint;
```

擦除“旧”图形和绘制“新”图形的工作是在 OnMouseMove()函数中完成的，因此该函数中需要做重大的修改。另外，在鼠标左键被按下的时候，应该记录当前点作为第一次的“旧”直线的终点。

程序清单 5-8 中列出了 OnLButtonDown()函数和 OnMouseMove()函数的代码。

程序清单 5-8　OnLButtonDown()和 OnMouseMove()

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 记录上一点的位置
m_LastPoint=point;
m_OldPoint=point;
CView::OnLButtonDown(nFlags, point);
}

void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// 建立客户区设备环境对象
CClientDC dc(this);
// 记录当前点的位置
m_CurPoint=point;

// 绘直线
if(m_bLine && ((nFlags & MK_LBUTTON)!=0))
{
    // 创建新画笔
    CPen myPen;
    CPen* pOldPen;
    myPen.CreatePen(PS_DASHDOT,1,RGB(192,192,192));
    pOldPen=dc.SelectObject(&myPen);

    // 设置光栅操作模式
    int nDrawMode;
    nDrawMode=dc.GetROP2();
    dc.SetROP2(R2_NOTXORPEN);

    // 擦除与重画
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_OldPoint);
    dc.MoveTo(m_LastPoint);
    dc.LineTo(m_CurPoint);
```

```
        m_OldPoint=point;

        // 恢复原设置
        dc.SetROP2(nDrawMode);
        dc.SelectObject(pOldPen);
}

// 绘矩形
if(m_bRect && ((nFlags & MK_LBUTTON)!=0))
{
        // 创建新画笔
        CPen myPen;
        CPen* pOldPen;
        myPen.CreatePen(PS_DASHDOT,1,RGB(192,192,192));
        pOldPen=dc.SelectObject(&myPen);

        // 设置光栅操作模式
        int nDrawMode;
        nDrawMode=dc.GetROP2();
        dc.SetROP2(R2_NOTXORPEN);

        // 擦除与重画
        dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_OldPoint.x,m_OldPoint.y);
        dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
        m_OldPoint=point;

        // 恢复原设置
        dc.SetROP2(nDrawMode);
        dc.SelectObject(pOldPen);
}

// 随手画
if(m_bSketch && ((nFlags & MK_LBUTTON)!=0) )
{
        // 创建新画笔
        CPen myPen;
        CPen* pOldPen;
        if(m_nPenWidth==1)
        {
                myPen.CreatePen(m_nPenStyle,1,m_CurColor);
        }
        else
        {
                myPen.CreatePen(PS_SOLID,m_nPenWidth,m_CurColor);
        }
        pOldPen=dc.SelectObject(&myPen);
```

```
        // 画线
        dc.MoveTo(m_LastPoint);
        dc.LineTo(m_CurPoint);
        m_LastPoint=m_CurPoint;

        dc.SelectObject(pOldPen);
    }

    CView::OnMouseMove(nFlags, point);
    }
```

在 OnMouseMove()函数中，用户可以看到，在使用了新的光栅操作模式后，立即恢复了原设置，这同使用新的 GDI 对象时的原因是一样的。另外，在绘直线和绘矩形的代码中，建立了一支新的画笔——灰色的点划线画笔——来绘制拉伸时的图形，而只在最后鼠标左键被释放时才真正绘出需要的图案。

现在用户已经完成了“ Draw ”程序的最后一项功能，应该运行一下该程序，用户可以检查的功能有：

- 检查“ Draw ”程序提供的三种绘图功能：绘直线、绘矩形和随手画；
- 为画笔选择不同的宽度，再次检验各种绘图功能；
- 选择不同的画笔风格，检验各种绘图功能；
- 选择不同的画刷风格，检验绘矩形功能时矩形内部的填充情况；
- 选择不同的颜色，检验各种绘图功能；
- 检查图形拉伸变化时，原有的图案是否受到了影响。

前面的几项功能我们已经检查过，现在我们检查程序提供的“ 拉伸 ”功能。简单的，可以绘制一个带填充的矩形，如图 5-6 所示为其正在绘制过程中鼠标拖动时的情形，可以看见明显的拉伸痕迹；如图 5-7 所示为鼠标松开后的效果，完全达到程序功能效果。

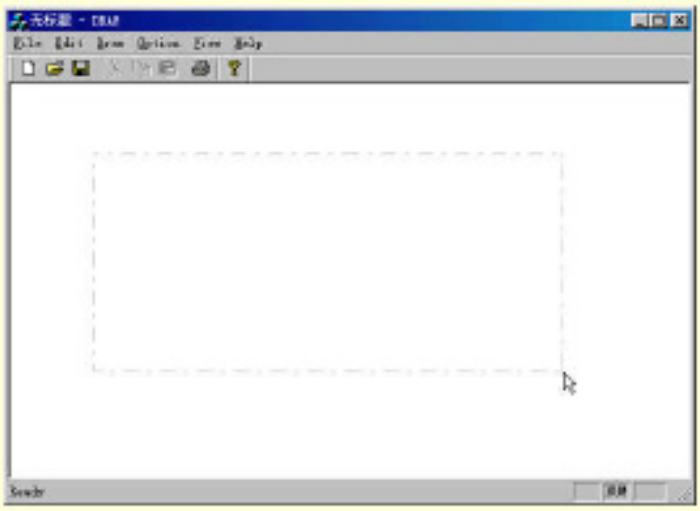

图 5-6 “ Project Settings ”对话框

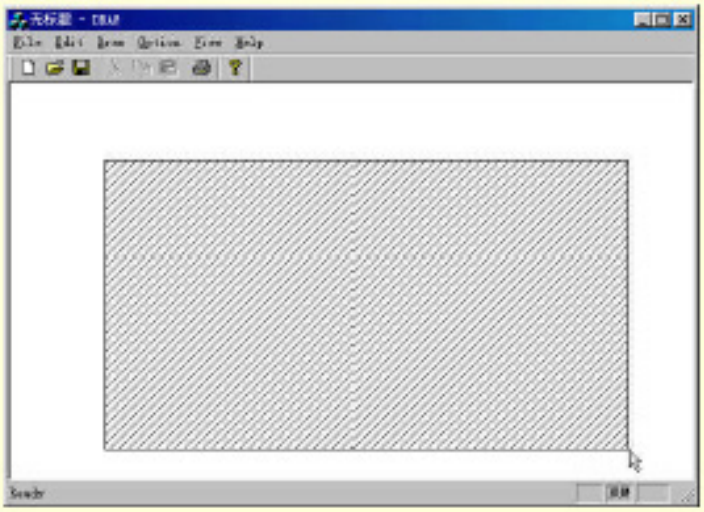

图 5-7 “Project Settings”对话框

“Draw”程序介绍了 MFC 应用程序中一些基本的绘图功能，这些都与 CDC 类——设备环境类——有很大的关系。实际上，CDC 类提供的与绘图有关的功能远不止这么一些，用户可以从联机手册中学习到更全面的内容。

5.2 在屏幕上绘图

这里我们所讲的在屏幕上绘图，专指类似屏幕保护程序的 Windows 系统绘图。用户应该知道，从 Windows 95 开始，系统提供了屏幕保护程序。当系统有比较长的一段时间没有接收到用户的键盘或鼠标输入时，就会调用屏幕保护程序。屏幕保护程序的作用一是避免屏幕的荧光物质在过长时间段内显示静止图像时被灼伤，二是当用户暂时离开计算机时，为用户的敏感数据提供保护。

现在已经出现了各种各样的屏幕保护程序，甚至出现了专门为用户提供屏幕保护程序的服务商，但是另一方面，利用 MFC，用户自己也可以做一个屏幕保护程序。在开始介绍之前，有必要向用户解释一下有关屏幕保护程序的基本知识。

5.2.1 屏保的概念

本节中将介绍如何建立一个简单的屏幕保护程序，说其简单，是因为程序的功能简单，只是不断地在屏幕上的随机位置画圆圈而已；但是该程序是一个真正的屏幕保护程序，具有屏幕保护程序的一切特征，用户在此基础上可以建立更漂亮的屏幕保护程序。

“标准”的实现屏幕保护的方法是直接使用 API 函数，用户需要对一些特定的消息作出处理，然后需要连接 SCRNSAVE 库。如果用户使用 SDK 进行 Win32 程序开发，那么理解直接使用 API 建立屏幕保护程序的步骤并不困难。但本书是介绍使用 MFC 基础类库的，因此，本节中介绍如何使用 MFC 基础类库建立一个真正可运行的屏幕保护程序。

实际上，使用 MFC 建立一个屏幕保护程序并不比建立一个普通的应用程序困难多少。屏幕保护程序的特殊支出在于两方面：首先，屏幕保护程序的后缀名必须是“.scr”；其次，系统在调用屏幕保护程序的时候附带有特定的参数，程序必须能够分析这些参数并作出相应的反应。

系统调用屏幕保护程序主要有两种方式：显示屏幕保护程序的设置对话框或者开始运行屏幕保护程序。系统通过向屏幕保护程序传递一个命令行参数而进行区分。表 5-7 中列出了系统发送的参数及其说明。

表 5-7 系统向屏幕保护程序发送的参数

参 数	说 明
/s,-s 或 s	以屏幕保护方式启动程序
/c,-c 或 c	以当前活动窗口为父窗口，显示设置对话框
无参数	显示设置对话框，且没有父窗口

对于使用“c”参数或无参数的情况，如果在设置对话框中没有什么特别的操作，可以当作为一种情况来进行处理。在本节的例子程序中就是这样做的。

有了这些基本的知识后，就可以开始建立用户自己的屏幕保护程序了。

5.2.2 建立新的项目

因为本屏幕保护程序的作用就是不断地在屏幕上画圆圈，因此将本项目的名称取为“Circle”是很合适的。建立的“Circle”程序仍然是一个 MFC 应用程序，只是在 AppWizard 中不需要选择众多的选项，而且还需要对 AppWizard 生成的框架代码做一番大的修改。

1．建立“Circle”项目的框架

在 AppWizard 的第一步中，选择程序为单文档界面，选择资源语言类型为“英语（美国）”，最重要的是清除“文档/视图结构”选项；在第二、三、四步中，清除所有的选择或将选项设置为“None”；在第五步和第六步中，仍然接受缺省选择。最后单击“Finish”按钮生成“Circle”程序的框架代码。

2．删除不需要的代码

在项目工作台的“ClassView”中，用户可以看到，“Circle”程序有四个类：CAboutDlg 类、CChildView 类、CMainFrame 类和 CCircleApp 类。实际上为建立一个屏幕保护程序只需要 CCircleApp 类就可以了，但因为 CAboutDlg 类的代码与 CCircleApp 类的代码在同一文件中，删除起来有点麻烦，因此将 CAboutDlg 类也留下，需要删除的是 CMainFrame 类和 CChildView 类。

为了删除 CMainFrame 类和 CChildView 类，打开项目工作台的“FileView”，展开“Circle”项目所包含的文件，删除下列文件：

- MainFrm.h
- MainFrm.cpp
- ChildView.h
- ChildView.cpp

再返回到“ClassView”，就可以看到 CMainFrame 类和 CChildView 类确实被删除了。

3．修改程序的后缀

屏幕保护程序是一类特殊的可执行程序，因此需要将“Circle”程序的后缀名由“.exe”修改为“.scr”。

在 VC6.0 的集成开发环境中选择“Project”菜单下的“Settings”命令，打开“Project Settings”对话框，如图 5-4 所示。

在“Project Settings”对话框中，选择“Link”标签，在“Output file name”框中修改“Circle”程序的输出文件名。图 5-8 中已用粗椭圆框标出需要改动的部分。

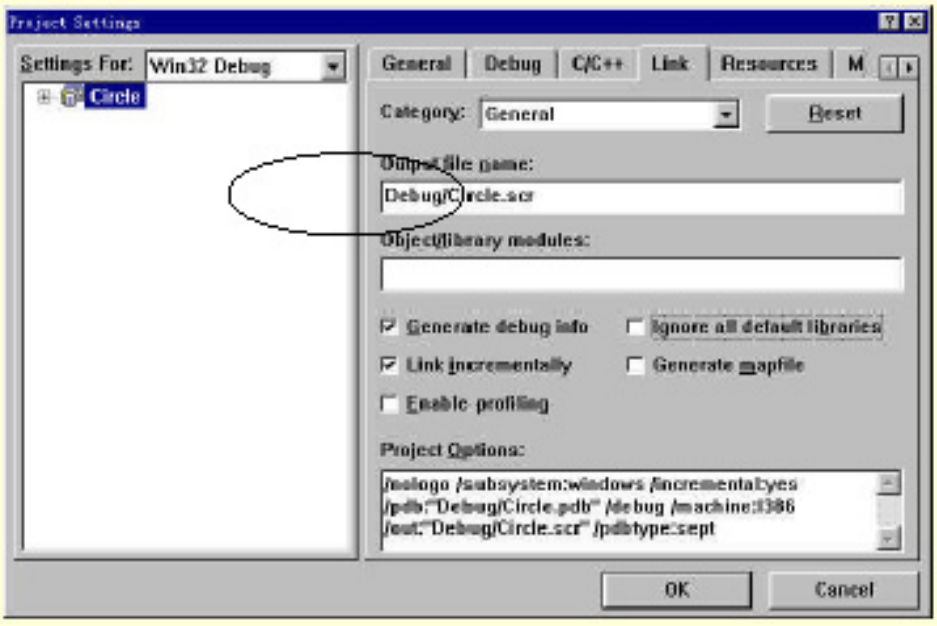

图 5-8 “Project Settings”对话框

5.2.3 修改 InitInstance()函数

在“Circle”程序中，用户几乎需要重建 InitInstance()函数的所有代码，本小节中将详细向用户介绍向 InitInstance()函数添加代码的过程。

首先打开 Circle.cpp 文件，删除开头的这样一行：

```
#include “MainFrm.h”
```

因为 CMainFrame 类已经被删除，所有与 CMainFrame 类有关的代码都不再需要。

1．使用系统注册表

同样地，在 InitInstance()函数中也需要删去所有与 CMainFrame 类有关的代码和注释，并稍稍修改 SetRegistryKey()函数。最后“干净”的 InitInstance()函数如程序清单 5-9 所示。

程序清单 5-9 修改后的 InitInstance()

```
BOOL CCircleApp::InitInstance()
{
// 设置应用程序在系统注册表中的键名
SetRegistryKey(_T("VC6.0 Examples and Technique "));
return TRUE;
}
```

在本书第二章中介绍应用程序类的时候，曾经稍稍提及了 SetRegistryKey()函数，该函数的功能就是指定本应用程序在系统注册表中的键名。在“Circle”程序中，需要多次使用系统注册表保存和提取信息，因此需要详细一点地介绍有关系统注册表的内容。

在 Windows 3.x 系统中，应用程序使用“.ini”文件保存重要的初始化信息。在 Windows 95 等 32 位系统中，“.ini”文件仍然可以使用，但是系统提供了另外一条保存信息的途径：系统注册表。用户可以通过运行 Windows 自带的“Regedit.exe”程序查看系统注册表。

MFC 应用程序也将自己的初始化信息保存在系统注册表中，SetRegistryKey()函数就指定了保存信息的位置。应用程序的信息在系统注册表中的位置一般为：

```
HKEY_CURRENT_USER\Software\<companyname>\<applicationname>\<sectionname>\<valuename>
```

SetRegistryKey()函数指定的是键名，一般都使用公司的名称<company name>，但本书建立的只是例子程序，因此简单地使用了"VC6.0 Examples and Technique"。“_T”宏的作用是将字符串转化为 Unicode 格式，这是为了程序国际化的需要。对一个简单的例子程序而言，用户完全可以不使用“_T”宏。

在“Circle”程序最终被完整地建立并运行通过后，<application name>就是程序名称“Circle”。另外，条目名（section name）和值名（value name）在对系统注册表进行写入操作时指定。表 5-8 中列出了 CWinApp 类的与系统注册表操作有关的成员函数，在“Circle”程序中将多次使用到这些函数。

表 5-8 注册表操作函数

函 数	说 明
SetRegistryKey()	指定本应用程序在系统注册表中的键名
GetProfileInt()	从系统注册表指定节的指定条目读取一个整数
GetProfileString()	从系统注册表指定节的指定条目读取一个字符串
WriteProfileInt()	向系统注册表指定节的指定条目写入一个整数
WriteProfileString()	向系统注册表指定节的指定条目写入一个字符串

在 InitInstance()函数中，只调用了 SetRegistryKey()函数指定程序的键名，在程序的其他部分，将调用另外一些注册表操作函数完成数据的保存与获取工作。

2．分析调用程序的参数

在前面已经介绍了系统在调用屏幕保护程序的时候会向程序传递不同的参数，因此，在程序中需要分析这些参数并做出相应的反应。这些工作应该在 InitInstance()函数中完成。

首先列出 InitInstance()函数的代码，如程序清单 5-10 所示。

程序清单 5-10 CCircleApp::InitInstance()

```
BOOL CCircleApp::InitInstance()
{
// 设置应用程序在系统注册表中的键名
SetRegistryKey(_T("VC6.0 Examples and Technique"));

// 分析调用程序的参数
if (__argc == 1 || MatchOption(__argv[1], _T("c")))
{
    // 显示设置对话框
    DoConfig();
}
else if (MatchOption(__argv[1], _T("s")))
{
    // 建立屏幕保护窗口
    CDrawWnd* pWnd=new CDrawWnd;
    CRect rect(0,0,::GetSystemMetrics(SM_CXSCREEN),
        ::GetSystemMetrics(SM_CYSCREEN));
// 注册窗口类
    LPCTSTR lpszClassName=
        AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW,
            ::LoadCursor(AfxGetResourceHandle(),
                MAKEINTRESOURCE(IDC_NULLCURSOR)));
    // 建立窗口对象
    pWnd->CreateEx(WS_EX_TOPMOST,lpszClassName,
        _T(""),WS_VISIBLE|WS_POPUP,rect,NULL,0,NULL);
    m_pMainWnd=pWnd;

    return TRUE;
}
return FALSE;
}
```

在 InitInstance()函数中，如果判断出系统传递的参数为“ c ”或者没有参数，则调用 DoConfig()函数显示设置对话框；如果判断出参数为“ s”，则建立一个 CDrawWnd 类的对象，并将该对象作为程序的主窗口。

研究一下 InitInstance()函数中的 return 语句就可以发现，程序在完成设置对话框的操作后直接退出了程序，而只有在建立了程序主窗口的情况下才进入程序的消息循环。这样做是完全符合程序的要求的。

在 InitInstance()函数中，有两件工作是留在后面完成的，这就是：

- DoConfig()函数：该函数的主要作用是显示屏幕保护程序的设置对话框，并完成程序在系统注册表中读取和写入程序设置的工作。用户需要建立一个设置对话框，一个相应的对话框类，并完成对话框的数据交换工作。
- CDrawWnd 类：该类就是屏幕保护程序真正运行时显示的窗口，用户需要完成 CDrawWnd 类的许多消息映射。

另外，在 AfxRegisterWndClass()函数中使用了资源 IDC_NULLCURSOR，从名称就可以看出，这是一个空的光标。用户需要向“ Circle ”程序中插入一个新的光标资源，并将该资源的命令 ID 修改为“ IDC_NULLCURSOR ”。用户不需要对该光标做任何修改，因为屏幕保护程序在运行的过程中需要的正是一个空的光标。

在 InitInstance()函数中还使用了 MatchOption()辅助函数，该函数用来对系统向“ Circle ”程序传递的参数进

行判断，程序清单 5-11 中是该函数的代码。

程序清单 5-11 CCircleApp::MatchOption()

```
BOOL CCircleApp::MatchOption(LPTSTR lpsz, LPTSTR lpszOption)
{

if (lpsz[0] == '-' || lpsz[0] == '/')
        lpsz++;
if (lstrcmpi(lpsz, lpszOption) == 0)
        return TRUE;
return FALSE;
}
```

在 InitInstance()函数中还调用了 AfxRegisterWndClass()函数，该函数用来对窗口类进行注册。在前面的各例子程序中，用户并没有看到任何对 AfxRegisterWndClass()函数的调用，这是因为 MFC 已经自动完成了框架窗口、视图等窗口类的注册工作。而“Circle”程序中的 CDrawWnd 类将由用户自己建立，因此需要手工完成注册过程。

分析一下 CDrawWnd 类对象的建立过程：

```
pWnd->CreateEx(WS_EX_TOPMOST,lpszClassName,
                _T(""),WS_VISIBLE|WS_POPUP,rect,NULL,0,NULL);
```

对照 CWnd::CreateEx()函数的说明可以看出，WS_EX_TOPMOST 指定了建立的窗口将处于系统所有窗口的最顶层；在初始化 rect 参数时使用了整个屏幕的最大宽度和高度，因此建立的窗口将覆盖整个屏幕。这两点保证了屏幕保护窗口的特性。

对于 InitInstance()函数中的具体代码，用户可以仔细分析，以理解新窗口对象的建立过程。

5.2.4 完成设置对话框

完成设置对话框的任务包括建立对话框资源、建立对话框类和在 DoConfig()函数中调用对话框等，下面来逐步完成这些任务。

1．建立对话框资源

用户需要向“Circle”程序中添加一个新的对话框，将对话框的命令 ID 指定为 IDD_CONFIG，对话框的标题文本为“Config”即可，并向对话框中添加必须的控件，如图 5-9 所示。

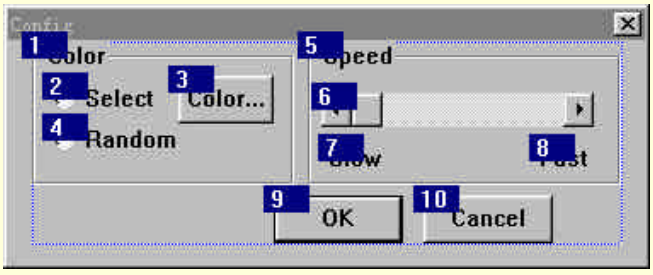

图 5-9 “Config”对话框

表 5-9 中列出了“Config”对话框中各控件的信息。

表 5-9 “Config”对话框的控件

跳转顺序	命令 ID	标题文本	需要修改的属性
1	IDC_STATIC	Color	无
2	IDC_RAD_SINGLECOLOR	Select	选中“Group”
3	IDC_BTN_COLOR	Color...	无
4	IDC_RAD_RANDOMCOLOR	Random	无

5	IDC_STATIC	Speed	无
6	IDC_SCR_SPEED	无	无
7	IDC_STATIC	Slow	无
8	IDC_STATIC	Fast	无
9	IDOK	OK	无
10	IDCANCEL	Cancel	无

2. 建立对话框类

在建立了对话框资源后，需要建立对话框类。选择新的对话框类的类名为 CConfigDlg，并为 CConfigDlg 添加一些成员变量，如表 5-10 所示。

表 5-10 CConfigDlg 类的成员变量

控件 ID	变量类型	变量名称
IDC_ RAD_SINGLECOLOR	int	m_nColorMode
IDC_SCR_SPEED	int	m_nSpeed
IDC_SCR_SPEED	CScrollBar	m_ScrSpeed

除了表 5-10 中的这些与控件相关联的成员变量外，用户还需要手工添加三个成员变量：

```
public:
int m_nColorRed;
int m_nColorGreen;
int m_nColorBlue;
```

3. 初始化对话框

在对话框显示之前，需要进行一些初始化工作。与前面使用有模式对话框一样，在 OnInitDialog()函数中完成对话框的初始化工作，程序清单 5-12 中列出了该函数的代码。

程序清单 5-12 CConfigDlg::OnInitDialog()

```
BOOL CConfigDlg::OnInitDialog()
{
CDialog::OnInitDialog();
// TODO: Add extra initialization here
// 将对话框置于中央位置
CenterWindow();
// 设置滚动条的特性
m_ScrSpeed.SetScrollRange(0,100);
m_ScrSpeed.SetScrollPos(m_nSpeed);
      return TRUE;
}
```

4. 调用颜色对话框

在“Config”对话框中，“Color”按钮用来供用户选择特定的颜色。用户需要响应 IDC_BTN_COLOR 按钮的 BN_CLICKED 通知，程序清单 5-13 中列出了 OnBtnColor()函数的代码。

程序清单 5-13 CConfigDlg::OnBtnColor()

```
void CConfigDlg::OnBtnColor()
{
// TODO: Add your control notification handler code here
CColorDialog dlg;
if(dlg.DoModal()==IDOK)
```

```
{
    // 返回选定颜色的各分量值
    m_nColorRed=GetRValue(dlg.GetColor());
    m_nColorGreen=GetGValue(dlg.GetColor());
    m_nColorBlue=GetBValue(dlg.GetColor());
}
}
```

5．处理滚动条消息

在“Config”对话框中使用了一个滚动条，为了能够正确地操作滚动条，用户需要对滚动条发出的WM_HSCROLL消息进行处理。程序清单5-14中列出了OnHScroll()函数的代码。

程序清单5-14　CConfigDlg::OnHScroll()

```
void CConfigDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
// TODO: Add your message handler code here and/or call default
int nCurPos=pScrollBar->GetScrollPos();

switch(nSBCode)
{
case SB_LEFT:           // 滚动至最左端
    nCurPos=0;
    break;
case SB_LINELEFT:       // 向左滚动一小步
    nCurPos--;
    break;
case SB_LINERIGHT:      // 向右滚动一小步
    nCurPos++;
    break;
case SB_PAGELEFT:       // 向左滚动一大步
    nCurPos-=10;
    break;
case SB_PAGERIGHT:          // 向右滚动一大步
    nCurPos+=10;
    break;
case SB_RIGHT:          // 滚动至最右端
    nCurPos=100;
    break;
case SB_THUMBPOSITION:  // 滚动至指定位置
case SB_THUMBTRACK:     // 自由拖动
    nCurPos=nPos;
}

// 控制取值范围
if(nCurPos<0)
    nCurPos=0;
if(nCurPos>100)
```

```
        nCurPos=100;
    pScrollBar->SetScrollPos(nCurPos);
    // 返回滚动条的位置
    m_nSpeed=nCurPos;
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
```

在 OnHScroll()函数中，用户也许已经感觉到滚动条的控制似乎有些奇怪：首先获取滚动条的当前位置，然后经过一定的处理后又重新设置滚动条的位置。其实这是很正常的事情，比如当滚动条需要向左滚动一大步时，滚动条本身并不知道需要移动多少距离，这都需要用户的直接控制。

6. 完成 DoConfig()函数

在 CCircleApp::DoConfig()函数中，完成显示对话框和交换数据的工作，程序清单 5-15 中列出了 CCircleApp::DoConfig()函数的完整代码。

程序清单 5-15 CCircleApp::DoConfig()

```
void CCircleApp::DoConfig()
{
// 构造对话框对象
CConfigDlg dlg;

// 从系统注册表中获取数据并初始化对话框的成员
dlg.m_nColorMode=GetProfileInt(_T("Simple Screen Saver"),
                    _T("ColorMode"),0);
dlg.m_nColorRed=GetProfileInt(_T("Simple Screen Saver"),
                    _T("ColorRed"),255);
dlg.m_nColorGreen=GetProfileInt(_T("Simple Screen Saver"),
                    _T("ColorGreen"),0);
dlg.m_nColorBlue=GetProfileInt(_T("Simple Screen Saver"),
                    _T("ColorBlue"),0);
dlg.m_nSpeed=GetProfileInt(_T("Simple Screen Saver"),
                    _T("Speed"),0);

if(dlg.DoModal()==IDOK)
{
    // 从设置对话框返回数据并写入系统注册表
    WriteProfileInt(_T("Simple Screen Saver"),
            _T("ColorMode"),dlg.m_nColorMode);
    WriteProfileInt(_T("Simple Screen Saver"),
                _T("ColorRed"),dlg.m_nColorRed);
    WriteProfileInt(_T("Simple Screen Saver"),
            _T("ColorGreen"),dlg.m_nColorGreen);
    WriteProfileInt(_T("Simple Screen Saver"),
            _T("ColorBlue"),dlg.m_nColorBlue);
    WriteProfileInt(_T("Simple Screen Saver"),
            _T("Speed"),dlg.m_nSpeed);
}
}
```

当然，在 Circle.cpp 文件的顶部要加上对“ ConfigDlg.h ”文件的包含。

建立和使用对话框的工作是用户所熟悉的，而且在代码中已经为用户加上了一些注释进行说明，因此这里也不需要多做解释。

5.2.5 完成窗口类

完成窗口类的工作并不是太困难，用户需要做的工作主要是从 CWnd 派生一个新的窗口类和控制窗口的动作，如擦除背景、不断地画圆圈和终止运行等等。

1．派生窗口类

使用 ClassWizard 可以很方便地完成从 CWnd 类派生一个新窗口类的工作。在 ClassWizard 中选择“ Add Class ”按钮，并在弹出菜单中选择“ New ”命令，就出现了“ New Class ”对话框。如图 5-10 所示。

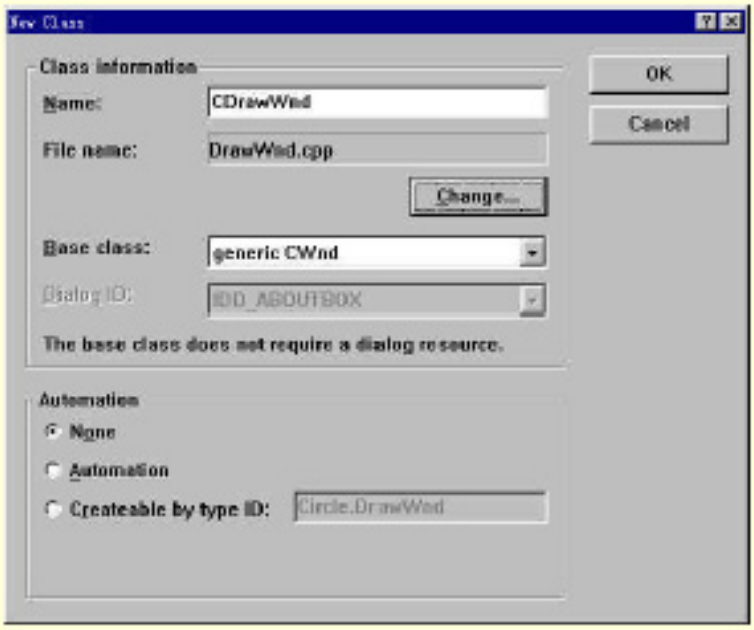

图 5-10 “ New Class ”对话框

在“ New Class ”对话框中，选择新类的基类为“ generic CWnd ”，并输入新类的名称“ CDrawWnd ”，单击“ OK ”按钮就生成了新的窗口类。ClassWizard 为用户提供了基本的代码框架，用户需要在此基础上添加自己的代码。

为了在 CDrawWnd 类中保存有关绘图的信息，需要添加一些成员变量，如下所示：

```
public:
    int m_nColorRed;
    int m_nColorGreen;
    int m_nColorBlue;
    int m_nColorMode;
    int m_nSpeed;
    CPoint m_ptLast;
```

其中前五个成员变量的含义与 CConfigDlg 类是类似的，而 m_ptLast 成员将用来控制屏幕保护程序的关闭。

2．初始化窗口类的成员

对 CDrawWnd 类的成员变量而言，可以在类的构造函数中进行初始化，也可以在 OnCreate()函数中进行初始化，这里选择了 OnCreate()函数。程序清单 5-16 中列出了 OnCreate()函数的完整代码。

程序清单 5-16 CDrawWnd::OnCreate()

```
int CDrawWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

// TODO: Add your specialized creation code here
// 从系统注册表中获取数据并初始化成员变量
m_nColorMode= AfxGetApp()->GetProfileInt(
                    _T("Simple Screen Saver"),_T("ColorMode"),0);
m_nColorRed= AfxGetApp()->GetProfileInt(
                    _T("Simple Screen Saver"),_T("ColorRed"),255);
m_nColorGreen= AfxGetApp()->GetProfileInt(
                    _T("Simple Screen Saver"),_T("ColorGreen"),0);
m_nColorBlue= AfxGetApp()->GetProfileInt(
                    _T("Simple Screen Saver"),_T("ColorBlue"),0);
m_nSpeed= AfxGetApp()->GetProfileInt(
                    _T("Simple Screen Saver"),_T("Speed"),0);


// 设置计时器
SetTimer(1,10+500-5*m_nSpeed,NULL);
// 设置 m_ptLast 为一个不可能的取值
m_ptLast=CPoint(-1,-1);
// 设置随机数发生器的“种子”
srand(1);

return 0;
}
```

在 OnCreate()函数中，调用了 srand()函数为程序的随机数发生器设置了一个“种子”，因为在程序中将需要在随机的位置显示随机大小的圆圈，必须先设置一个“种子”，随机数发生器才能正常工作。

3. 关闭屏幕保护窗口

当系统检测到用户敲击了键盘、移动了鼠标或者按下了鼠标的某个键时，就需要关闭屏幕保护程序。也就是说，在 WM_LBUTTONDOWN、WM_MBUTTONDOWN、WM_RBUTTONDOWN、WM_KEYDOWN 和 WM_MOUSEMOVE 等消息的处理函数中，需要以某种方式终止屏幕保护程序的运行。

另外，当程序窗口接收到 WM_ACTIVE 消息，并且该消息的参数指定窗口应该非终止活动状态时，也需要终止屏幕保护程序的运行。

程序清单 5-17 中列出了这些负责“关闭”程序的函数的代码。

程序清单 5-17　关闭屏幕保护程序

```
void CDrawWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
PostMessage(WM_CLOSE);
CWnd::OnLButtonDown(nFlags, point);
}

void CDrawWnd::OnMButtonDown(UINT nFlags, CPoint point)
```

```
{
// TODO: Add your message handler code here and/or call default
PostMessage(WM_CLOSE);
CWnd::OnMButtonDown(nFlags, point);
}

void CDrawWnd::OnRButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
PostMessage(WM_CLOSE);
CWnd::OnRButtonDown(nFlags, point);
}

void CDrawWnd::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
PostMessage(WM_CLOSE);
CWnd::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CDrawWnd::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
if (m_ptLast == CPoint(-1,-1))
        m_ptLast = point;
else if (m_ptLast != point)
        PostMessage(WM_CLOSE);
CWnd::OnMouseMove(nFlags, point);
}
void CDrawWnd::OnActivate(UINT nState, CWnd* pWndOther,
                BOOL bMinimized)
{
CWnd::OnActivate(nState, pWndOther, bMinimized);

// TODO: Add your message handler code here
if(nState==WA_INACTIVE)
        PostMessage(WM_CLOSE);
}
```

在 OnMouseMove()函数和 OnActivate()函数中，首先需要进行一定的判断，然后才能关闭程序。

关闭程序的任务是由 PostMessage()函数实现的，该函数将一条 WM_CLOSE 消息放在程序的消息循环中。当程序接收到 WM_CLOSE 消息时，就关闭程序的主窗口，终止程序的运行。

4. 清除程序的遗留问题

在关闭程序的过程中，还有两件事情需要处理，这就是清除计时器和删除窗口对象。在本书前面的叙述中已经强调过设置了计时器之后一定要计时清除，这里也不例外。在 CDrawWnd::OnDestroy()函数中可以完成这一任务。

另外，由于程序的主窗口对象是由“ new ”关键字进行内存空间分配的，因此在程序终止时需要删除该对

象以释放内存空间，这可以在 CDrawWnd::PostNcDestroy()函数中完成。

程序清单 5-18 中列出了这两个函数的代码。

程序清单 5-18 OnDestroy()和 PostNcDestroy()

```
void CDrawWnd::OnDestroy()
{
CWnd::OnDestroy();
// 清除计时器
KillTimer(1);
}

void CDrawWnd::PostNcDestroy()
{
// TODO: Add your specialized code here and/or call the base class
// 删除主窗口对象
delete this;
CWnd::PostNcDestroy();
}
```

5. 实现屏幕绘图

显而易见，当窗口对象每收到一条 WM_TIMER 消息，就应该执行一次绘图任务，因此需要在 CDrawWnd::OnTimer()函数中添加绘图代码。但是在开始绘图之前，一般都需要擦除背景，可以在 OnPaint()函数中实现这一任务。程序清单 5-19 中列出了这两个函数的代码。

程序清单 5-19 OnPaint()和 OnTimer()

```
void CDrawWnd::OnPaint()
{
CPaintDC dc(this); // device context for painting

// TODO: Add your message handler code here
// 清除窗口背景
CBrush myBrush;
myBrush.CreateSolidBrush(m_CurColor);
CRect rect;
GetClientRect(&rect);
dc.FillRect(&rect,&myBrush);
// Do not call CWnd::OnPaint() for painting messages
}

void CDrawWnd::OnTimer(UINT nIDEvent)
{
// TODO: Add your message handler code here and/or call default
if(nIDEvent==1)
{
    // 随机产生颜色
    if(m_nColorMode==1)
```

```
    {
        m_nColorRed=(int) 255.0*rand()/RAND_MAX;
        m_nColorGreen=(int) 255.0*rand()/RAND_MAX;
        m_nColorBlue=(int) 255.0*rand()/RAND_MAX;
    }

    // 建立新的画笔
    CPen myPen;
    int nPenWidth=(int) 10.0*rand()/RAND_MAX;
    myPen.CreatePen(PS_SOLID,nPenWidth,
        RGB(m_nColorRed,m_nColorGreen,m_nColorBlue));

    // 选择画笔和画刷
    CClientDC dc(this);
    CPen* pOldPen=dc.SelectObject(&myPen);
    dc.SelectStockObject(BLACK_BRUSH);

    // 随机产生圆心位置和半径
    int xCenter=::GetSystemMetrics(SM_CXSCREEN)*rand()/RAND_MAX;
    int yCenter=::GetSystemMetrics(SM_CYSCREEN)*rand()/RAND_MAX;
    int nRadius=(int) 100.0*rand()/RAND_MAX;

    // 画圆圈
    dc.Ellipse(xCenter-nRadius,yCenter-nRadius,
        xCenter+nRadius,yCenter+nRadius);
    dc.SelectObject(pOldPen);
}

CWnd::OnTimer(nIDEvent);
}
```

OnPaint()函数和 OnTimer()函数中的绘图代码都是很简单的，在添加了足够注释的情况下，用户应该可以看懂这些代码。

现在应该是用户激动的时候了，编译一下“ Circle ”程序，然后在项目的“ Debug ”或者“ Release ”目录下找到“ Circle.scr ”程序，将该程序拷贝到 Windows 系统的“ System ”目录下，然后打开“ Control Panel ”，双击“ Display ”图标，在“ Screen saver ”标签下选择“ Circle ”程序作为当前的屏幕保护程序。

用户可以单击“ Config ”按钮或“ Preview ”按钮进行参数设置或者直接预览屏幕保护程序，甚至可以放下鼠标与键盘，静静地等上一小段时间（这取决于用户的设置），欣赏屏幕保护程序的运行，如图 5-11 所示。

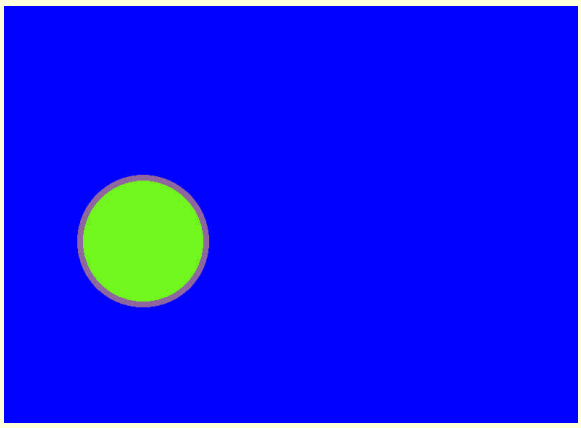

图 5-11 屏保运行结果

至此，用户自己的屏幕保护程序已经建立成功了。“ Circle ”程序是简单的，但是在该程序的基础上，再加上一些努力，用户就可以实现更漂亮的屏幕保护程序了。

5.3 本章小结

在本章中，通过两个例子程序介绍了在 MFC 应用程序中进行屏幕绘图的基础知识。

“ Draw ”程序是一个简单的但是很实用的例子程序，通过该例子程序，介绍了设备环境和 GID 对象的概念，并通过实现一些简单的图形绘制功能，介绍了在其他函数中构造和使用设备环境对象的方法，并介绍了如何创建和使用新的画笔和画刷，以及设置画笔和屏幕的光栅操作模式等内容。这些知识为用户使用 MFC 的设备环境对象打下了一个坚实的基础。

“ Circle ”程序看起来很困难，但真正实现起来却并非用户所料想的那么艰难。该程序综合运用了本书前面各章中介绍的知识，用于绘图的代码并不是很复杂。本书试图通过“ Circle ”程序，向用户展示一下非文档/视图结构的 MFC 应用程序是怎样工作的，希望加深用户对 MFC 应用程序结构的理解。

本章并没有详细地介绍设备环境各方面的操作，但是在本书的几乎所有的例子程序中都涉及到了设备环境的使用。在下一章中，用户将学习到有关使用字体和显示文本的一些知识，用户还可以接触到打印和打印预览，这也是图形界面程序的一个很重要的方面。用户可以从 VC6.0 联机手册中获得更多更详细的说明，或者可以直接从联机手册提供的例子程序中进行学习。

第六章 MFC 打印技术的应用

在 Windows 应用程序中实现打印功能在以前并不是一件容易的事情。尽管 Windows 系统已经提供了与具体的打印设备无关的设备环境供程序员使用，但是还有许多需要考虑的东西。

MFC 应用程序框架使得程序员能够比较容易地实现打印功能，它通过 CView 类支持文档的打印和打印预览功能。本章，我们将主要学习以下方面的内容：

- 基本打印功能
- 设置映射模式
- 正确实现打印分页
- MFC 打印进程中各函数的调用顺序

6.1 基本打印与打印功能

在 MFC 程序中实现基本的打印与打印预览功能是很简单地，只要在利用 AppWizard 生成应用程序的步骤 4 中选择了 Print and print preview 选项，生成的程序就自动提供了基本的打印与打印预览功能。

下面建立一个名为 TestPrint 的例子程序。利用 AppWizard 新建一个 MFC AppWizard(exe)类型的项目，项目名为 TestPrint，在以后的各步骤中按以下操作：

（1）选择单文档界面，选中文档/视图结构支持，并选择英语为资源语言类型。

（2）不需支持任何数据库功能，选择 None。

（3）不需支持任何复合文档，故选择 None；不要选中 Automation 与 ActiveX Controls 选项。

（4）保持缺省选项，注意一定要选中 Print and print preview 选项。

（5）保持缺省选择。

（6）保持缺省状况，单击 Finish 完成项目设置。

这样生成的 TestPrint 程序就具有了基本的打印与打印预览功能。为了验证程序的功能，需要对程序代码稍做修改。

在 CTestPrintView 类的 OnDraw()函数中添加一行代码，画一个矩形。程序清单 6-1 中是完整的 OnDraw()函数的代码：

程序清单 6-1 OnDraw()

```
void CTestPrintView::OnDraw(CDC* pDC)
{
CTestPrintDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
pDC->Rectangle(0,0,300,100);
}
```

现在请编译并运行 TestPrint 程序，图 6-1 显示了程序的运行结果，在窗口的客户区画了一个长为 300 单位、高为 100 单位的矩形。

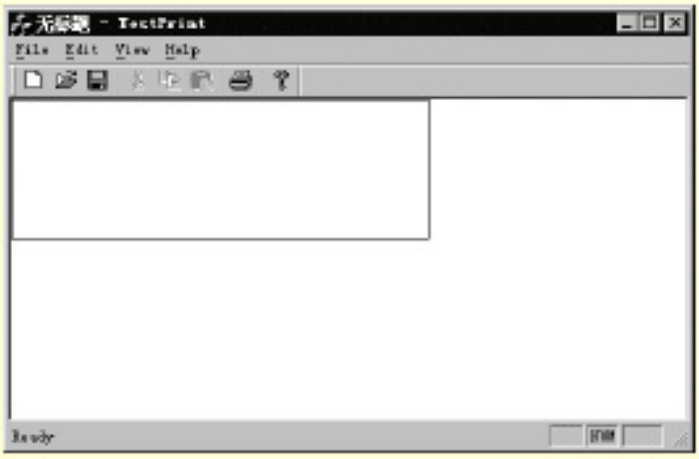

图 6-1 在窗口客户区中画矩形

在 TestPrint 程序的菜单系统中，在 File 菜单下有三个与打印有关的菜单项：Print（打印）、Print Preview（打印预览）和 Print Setup（打印设置）。选择 Print Preview 命令将打开打印预览窗口，如图 6-2 所示：

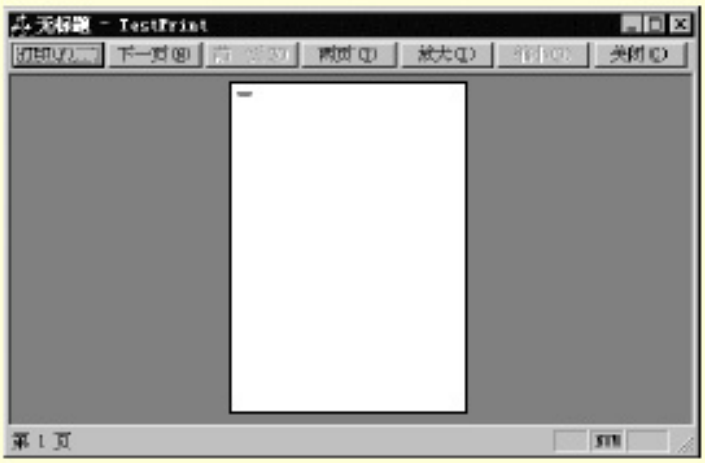

图 6-2 打印预览窗口（缩小至整页）

在打印预览窗口中显示了当前页——第 1 页，在页面的左上角确实有一个矩形。这是图形被缩小至整页时的情形。图 6-3 是放大至 100%时的打印预览情形：

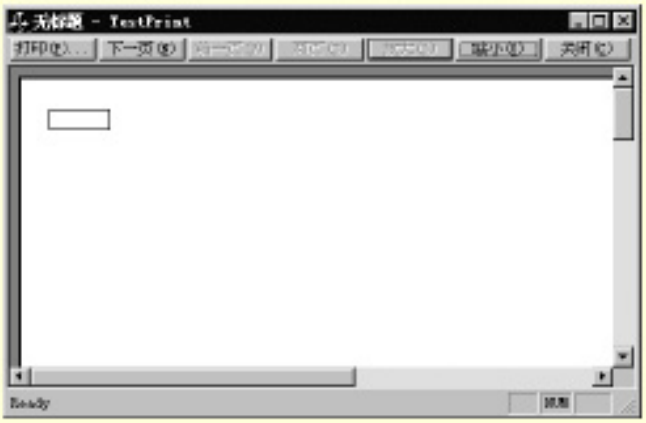

图 6-3 打印预览（100%）

如果用户愿意，可以选择 File 菜单下的 Print 命令在打印机上输出该页。选择 Print 命令将弹出打印对话框，如图 6-4 所示：

图 6-4 典型的打印对话框

打印对话框的具体形式取决与用户的系统，但打印对话框的基本功能都是相似的。在打印对话框中，可以选择打印机，可以选择打印范围，可以选择打印份数等等。确定所有的打印选项后，单击“确定”按钮（也可能是 OK 按钮或其他形式）就会在打印机上输出所绘矩形。输出的打印页与打印预览中的页面看起来应该没什么区别。

选择 File 菜单下的 Print Setup 命令将弹出打印设置对话框，如图 6-5 所示：

图 6-5 典型的打印设置对话框

同打印对话框一样，打印设置对话框的具体表现形式取决与用户的系统。在打印设置对话框中主要是选择打印纸张以及打印方向。在图 6-2 的打印预览窗口中，使用的是缺省的纸张，A4 大小，纵向打印。为了做一比较，这里选择 A5 大小的纸张，横向打印。并单击“确定”按钮关闭对话框。

在 File 菜单中再次选择 Print Preview 命令，观察改变了纸张的大小和打印方向后的打印预览结果，如图 6-6 所示：

图 6-6 打印预览（100%）（A5 幅面，横向打印）

比较图 6-2 和图 6-6 的预览效果，可以看出，改变了纸张大小和打印方向后，预览的效果确实有很大的变化。

现在可以看到，对于 TestPrint 程序目前的版本而言，用户不用做任何额外的努力，就能够让应用程序拥有这些很基本但又很重要的打印功能。TestPrint 程序可以设置打印选项，可以进行打印预览，自然也可以在打印机上真正地输出。

当然，TestPrint 程序的打印功能还有一些不尽人意之处，但对程序员来说，利用 AppWizard 生成的应用程序自动就拥有这些打印功能，这已经是一个很不错的起点了。在下面各节中，将逐步完善 TestPrint 程序的打印功能，同时介绍在 MFC 程序中实现打印功能的方法。

6.2 改变映射模式

TestPrint 程序第一个令人不满意的地方就是矩形的大小没有被保持：仔细比较图 6-1、图 6-3 和图 6-6 中的矩形，显示在屏幕上的矩形很明显地要比打印预览窗口中显示的或打印机上打印出的矩形大出许多。这是第一个需要改进的地方。

注意 OnDraw()函数中调用的 Rectangle()函数，如果传递给 OnDraw()函数的参数 pDC 代表的是显示设备环境，该函数就在屏幕上画一个长 300 单位、高 100 单位的矩形；如果 pDC 代表的是打印设备环境，该函数就在打印机上画一个长 300 单位、高 100 单位的矩形。问题就在于 TestPrint 程序现在的版本采用的是缺省的映射模式 MM_TEXT，在这种映射模式下，显示设备的一个单位对应于打印机一个的单位，但是打印机的一个单位比显示设备的一个单位要小得多，因此即使使用同样的单位数，在打印机上输出的矩形也要比在屏幕上输出的矩形小得多。

如果要求显示在屏幕上的矩形与输出在打印机上的矩形同样大小，只需要改变绘图时的映射模式。表 6-1 列出了可以使用的映射模式：

表 6-1 映射模式

映射模式	单 位	X 轴	Y 轴
MM_HIENGLISH	0.001 英寸	向右为正	向上为正
MM_HIMETRIC	0.01 毫米	向右为正	向上为正
MM_ISOTROPIC	用户自定义	用户自定义	用户自定义
MM_LOENGLISH	0.01 英寸	向右为正	向上为正
MM_LOMETRIC	0.1 毫米	向右为正	向上为正
MM_TEXT	设备象素	向右为正	向下为正
MM_TWIPS	1/1440 英寸	向右为正	向上为正

使用 MM_HIENGLISH、MM_HIMETRIC 等映射模式是与设备无关的，也就是说，在屏幕上显示为 1 英寸长度，在打印机上输出也是 1 英寸长度。使用这样的映射模式就不会出现屏幕上显示与打印机上输出不一致的情况了。

设置绘图映射模式应该使用 CDC 类的 SetMapMode()函数，该函数原型如下：

```
virtual int SetMapMode( int nMapMode );
```

参数 nMapMode 的取值就是表 6-1 中所列的映射模式。与 SetMapMode()函数对应的有一个 GetMapMode()函数，该函数返回当前的映射模式。

设置绘图的映射模式一般在 OnPrepareDC()函数中进行，OnPrepareDC()函数是 CDC 类的一个虚函数，利用 ClassWizard 对它进行重载，如图 6-7 所示：

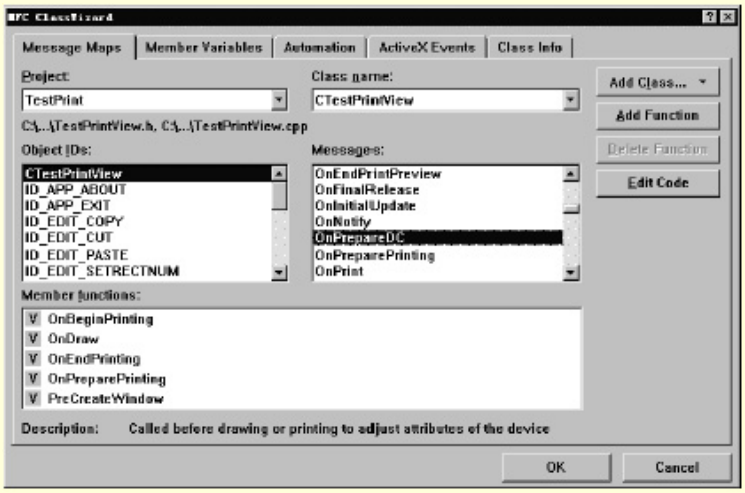

图 6-7 重载 OnPrepareDC()虚函数

从 OnPrepareDC()函数的函数名就可以看出，该函数用于使用 DC 对象前的准备工作，对在屏幕上显示而言，在 OnPrepareDC()函数进行映射模式的设置是合适的。在后面将看到 OnPrepareDC()函数在打印中的重要作用。

需要在 OnPrepareDC()函数中增加一行设置映射模式的代码，程序清单 6-2 中列出了修改后的 OnPrepareDC()函数：

程序清单 6-2 OnPrepareDC()

```
void CTestPrintView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
// TODO: Add your specialized code here and/or call the base class
pDC->SetMpaMode(MM_LOENGLISH);
CView::OnPrepareDC(pDC, pInfo);
}
```

由于在 OnPrepareDC()函数中改变了映射模式，而且 MM_LOENGLISH 模式与 MM_TEXT 模式的 Y 轴的正向是相反的，因此也需要修改 OnDraw()函数中的代码，程序清单 6-3 中是修改后的 OnDraw()函数：

程序清单 6-3 CTestPrintView::OnDraw()

```
void CTestPrintView::OnDraw(CDC* pDC)
{
CTestPrintDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
pDC->Rectangle(0,0,300,-100);
}
```

在 OnDraw()函数中仍然调用 Rectangle()函数画矩形。注意传递给 Rectangle()函数的参数，矩形的右下角的 Y 坐标是负值。这是因为在 MM_LOENGLISH 映射方式下，Y 坐标向上为正，而坐标系的原点任在应用程序窗口客户区的左上角。按 MM_LOENGLISH 映射方式的单位设置，Rectangle()函数画了一个 3 英寸长、1 英寸高的矩形。图 6-8 显示了运行中的程序：

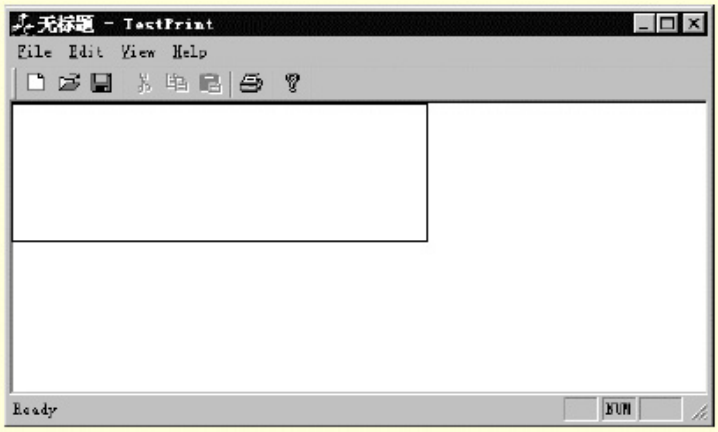

图 6-8 运行中的程序（MM_LOENGLISH 模式）

再次进行打印预览，图 6-9 显示了打印预览窗口：

图 6-9 打印预览（100%）（MM_LOENGLISH 模式）

比较图 6-8 和图 6-9 中两矩形的大小，确实是同样大小。这表明使用 MM_LOENGLISH 模式保持了矩形的大小和相对位置关系。实际上，除了 MM_TEXT 映射模式以外，其他的映射模式都可以保持图形的大小和相对位置关系。

到现在为止，都只是需要在打印预览窗口中显示一页或在打印机上输出一页，这已经比较正确地完成了，但如果要打印多页，还需要做相当的努力。

6.3 打印多页

打印多页比打印单页要困难一些。在打印单页时，根本无须考虑换页的问题。然而，打印多页的情形是经常遇到的。

在 TestPrint 程序的上一个版本中，只画了一个矩形，打印机的一页已经足够打印该矩形了。为了能够演示打印多页的方法，需要输出多个矩形，实现多页。

6.3.1 设置矩形的数目

为了记录矩形的数目，在文档类 CTestPrintDoc 中添加一个变量 m_nRectNum，类型为 int，访问级别为 public。如图 6-10 所示：

图 6-10 添加成员变量 m_nRectNum

注意：

✍ 这里为方便起见，将文档类的成员变量设置为 public，这是不太符合对象封装的要求的，严格说来，应该将该变量设置为保护状态，并使用 GetXXX()和 SetXXX()函数来获取和设置该变量的值。

在 CTestPrintDoc 类的构造函数中对该变量进行初始化，程序清单 6-4 中是 CTestPrintDoc 类的构造函数：

程序清单 6-4 CTestPrintDoc()

```
CTestPrintDoc::CTestPrintDoc()
{
// TODO: add one-time construction code here
m_nRectNum=1;
}
```

为了在程序运行的过程中能改变该变量的值，使用一个简单的对话框。如图 6-11 所示为完成的对话框：

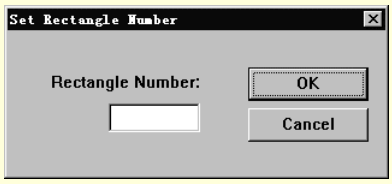

图 6-11 Set Rectangle Number 对话框

其中文本框控件的标识符指定为 IDC_RECTNUM。打开 ClassWizard，为该对话框建立一个新类 CSetRectNumDlg，并为 IDC_RECTNUM 控件指定一个关联变量 m_nSetRectNum，类型为 int，取值范围从 0 到 100。如图 6-12 所示：

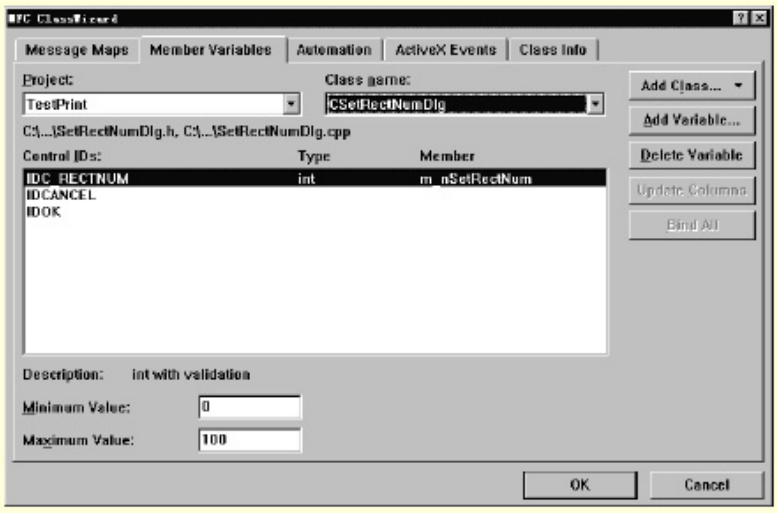

图 6-12 添加关联变量 m_nSetRectNum

为了显示 Set Rectangle Number 对话框，在 TestPrint 程序的 Edit 菜单中增加一项：Set Rectangle Number，指定标识符为 ID_EDIT_SETRECTNUM。使用 ClassWizard 让 CTestPrintDoc 类处理该菜单命令。程序清单 6-5 列出了需要向消息处理函数添加的代码：

程序清单 6-5 OnEditSetrectnum()

```
void CTestPrintDoc::OnEditSetrectnum()
{
// TODO: Add your command handler code here
CSetRectNumDlg dlg;
dlg.m_nSetRectNum=m_nRectNum;
if(dlg.DoModal()==IDOK)
    m_nRectNum=dlg.m_nSetRectNum;
UpdateAllViews(NULL);
}
```

程序说明：

一定要在 CTestPrintDoc 类源文件的顶部加上以下一行：

```
#include "SetRectNumDlg.h"
```

否则在编译时将因为找不到 CSetRectNumDlg 类的定义而出错。这样通过在 Edit 菜单中选择 Set Rectangle Number 命令，在弹出的 Set Rectangle Number 对话框中就可以设置矩形的数目了，并调用 UpdateAllView()函数及时进行重画。

在 CTestPrintView 类的 OnDraw()函数中也要做相应的修改。程序清单 6-6 列出了修改后的 OnDraw()函数的代码：

程序清单 6-6 OnDraw()

```
void CTestPrintView::OnDraw(CDC* pDC)
{
CTestPrintDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
for(int i=0;i<pDoc->m_nRectNum;i++)
{
    CString s;
    s.Format("%d",i);
    pDC->Rectangle(0,-100*i,300,-(100*i+80));
    pDC->TextOut(10,-(100*i+20),s);
}
}
```

在 OnDraw()函数中利用一个循环画矩形，注意传递给 Rectangle()函数的参数，这意味着每隔一英寸位置就画一个矩形，矩形长为 3 英寸，高为 0.8 英寸。通过字符串对象 s 还在矩形的左上角还标明了该矩形的编号，该编号从 0 开始。

再次编译并运行 TestPrint 程序，在 Edit 菜单下选择 Set Rectangle Number 命令，在 Set Rectangle Number 对话框中将矩形的个数设置为 11，设置完成后，程序立即在窗口的客户区中显示 11 个矩形，当然，受屏幕大小限制，不可能将这 11 个矩形完全显示出来。但可以通过打印预览来进行观察。如图 6-13 所示是打印预览窗口。

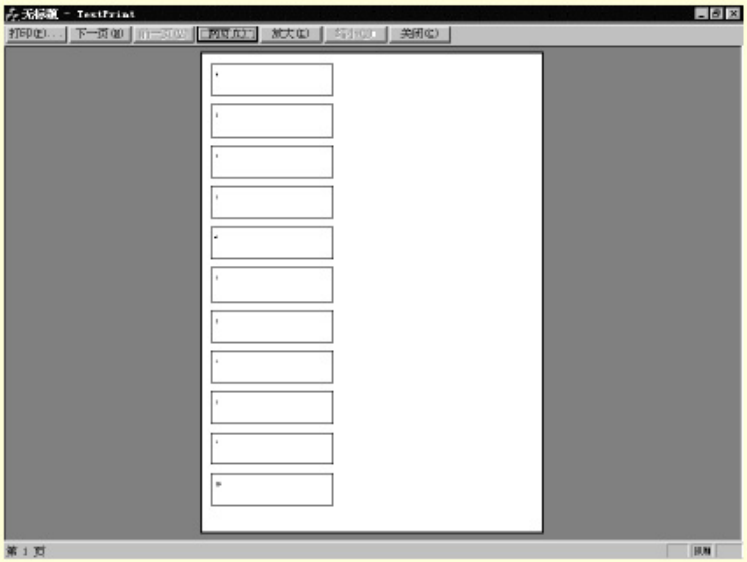

图 6-13 11 个矩形时的打印预览

在打印预览窗口中选择“一页”模式，从打印预览窗口中可以看出，11 个矩形仍然没有超出一页范围，打印预览能够在一页中正确地显示这 11 个矩形。如果超过 11 个矩形将是什么情况呢？这正是以下要讨论的内容。

6.3.2 设置页数

如果程序所画的矩形的个数较多，很明显是不可能在一页中打印完的。因此需要对打印任务进行分页。对文档正确分页后，一页一页地完成打印任务。

分页可以在 OnPreparePrinting()函数或 OnBeginPrinting()函数中完成，这两个函数都是 CDC 类的虚函数，要完成特定的任务必须加以重载。如果事先已经知道文档的页数，可以在 OnPreparePrinting()函数中进行分页；如果需要知道有关打印机的信息，比如纸张的大小，才能决定文档的页数，可以在 OnBeginPrinting()函数中进行分页。

OnPreparePrinting()函数在文档被打印或打印预览前由应用程序框架调用。如果用户的程序需要实现打印或打印预览功能，必须重载本函数。在 TestPrint 程序中，由于在利用 AppWizard 生成应用程序的步骤四中选择了 Print and print preview 选项，程序已经自动地重载了该函数。程序清单 6-7 中是缺省的 OnPreparePrinting()函数的代码：

程序清单 6-7 OnPreparePrinting()

```
BOOL CTestPrintView::OnPreparePrinting(CPrintInfo* pInfo)
{
// default preparation
return DoPreparePrinting(pInfo);
}
```

在 OnPreparePrinting()函数的缺省版本中，只是简单地调用了 CDC 类的另一个函数 DoPreparePrinting()。注意传递给 OnPreparePrinting()函数的参数，pInfo 是一个指向 CPrintInfo 对象的指针，CPrintInfo 对象中保存着关于本次打印或打印预览工作的信息。表 6-2 列出了 CPrintInfo 类的成员：

表 6-2 CPrintInfo 类的成员

成 员	说 明
SetMaxPage()	设置文档的最大页码
SetMinPage()	设置文档的最小页码

GetMaxPage()	获取文档的最大页码
GetMinPage()	获取文档的最小页码
GetFromPage()	获取用户选择打印的第一页的页码
GetToPage()	获取用户选择打印的最后一页的页码
m_bDirect	标志文档是否直接被打印（跳过打印对话框）
m_bPreview	标志文档是否在打印预览状态
m_bContinuePrinting	标志文档是否被继续打印
m_nCurPage	保存当前打印或预览的页码
m_nNumPreviewPages	保存当前预览的页数（1 或 2）
m_rectDraw	保存定义当前页有效区域的矩形
m_strPageDesc	保存页码格式字符串
m_pPD	保存指向打印作业的 CPrintDialog 的指针

一旦用户选择了打印或打印预览命令，应用程序框架就建立一个 CPrintInfo 对象，该对象在命令结束时被销毁。CPrintInfo 类既保存了有关整个文档的信息，如文档的打印范围，也保存了打印工作的当前状态，如当前打印页的页码。还有信息保存在相关联的 CPrintDialog 中（m_pPD 指针代表），主要是用户在打印对话框中输入的参数。

在整个打印进程中，CPrintInfo 对象在应用程序框架和视图之间传递，用于在两者之间交换信息。比如说，框架通过 CPrintInfo 对象的 m_nCurPage 成员通知视图类当前打印页的页码，视图类接收该页码值并实际地将该页打印出来。

DoPreparePrinting()函数显示打印对话框并创建一个打印机设备环境对象。如果用户希望在显示打印对话框之前初始化该对话框，可以将初始化值传递给参数 pInfo 的成员。比如说，如果已经知道文档的长度，在调用 DoPreparePrinting()函数之前将其传递给 pInfo 的成员函数 SetMaxPage()，该值将显示在打印对话框的范围选择框中。

DoPreparePrinting()函数对打印和打印预览有不同的处理，由 pInfo 参数的 m_bPreview 成员区分。如果是打印文档，则显示打印对话框，并用 pInfo 参数传递过来的值先对打印对话框进行初始化；一旦打印对话框被关闭，该函数就根据用户在打印对话框中的设置创建一个打印设备环境，并通过 pInfo 参数将该设备环境返回。创建的打印设备环境将用于后续的打印文档工作。如果是打印预览文档，DoPreparePrinting()函数根据当前的打印机设置创建一个打印设备环境，该设备环境将被用于在预览中模拟打印。

OnBeginPrinting()函数在打印或打印预览工作开始是被调用，此时 OnPreparePrinting()函数已经返回。程序清单 6-8 中是缺省生成的 OnBeginPrinting()函数，缺省版本没有完成任何功能，需要注意的是传递给 OnBeginPrinting()函数的参数。

程序清单 6-8 OnBeginPrinting()

```
void CTestPrintView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
// TODO: add extra initialization before printing
}
```

由于 DoPreparePrinting()已经被调用而且返回了一个打印设备环境对象，该对象以指针的形式传递给 OnBeginPrinting()函数。因此，在 OnBeginPrinting()函数中可以进行与设备环境有关的初始化工作。一方面是创建打印过程中需要的各种 GDI 资源，另一方面是设置打印的页数。前面已经提到，有时候只有知道了打印纸张的长度，才能决定打印的页数，这种情况下就无法在 OnPreparePrinting()函数中决定打印页数，因为在 OnPreparePrinting()函数被调用时，打印设备环境还没有生成，而在 OnBeginPrinting()函数中正好可以确定需要打印的页数。

在 OnPreparePrinting()函数或 OnBeginPrinting()函数中调用参数 pInfo 的成员函数 SetMaxPage()就可以设置打印页数。

对 TestMsg 程序来说，由于所绘矩形的数目是可变的，而且决定打印页数也与打印纸张的大小有关，因此，在 OnPreparePrinting()函数中是无法设置打印文档的页数的，只能在 OnBeginPrinting()函数中进行设置。程序清单 6-9 中是完成的 OnBeginPrinting()函数：

程序清单 6-9 OnBeginPrinting()

```
void CTestPrintView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
// TODO: add extra initialization before printing
int pageHeight=pDC->GetDeviceCaps(VERTRES);
int logicY=pDC->GetDeviceCaps(LOGPIXELSY);
int rectHeight=(int) 1*logicY;
int pageNum=(GetDocument()->m_nRectNum*rectHeight)/pageHeight+1;
pInfo->SetMaxPage(pageNum);
}
```

在向 OnBeginPrinting()函数添加代码之前，必须去掉参数两边的注释，之所以加上这样的注释，是因为缺省的 OnBeginPrinting()函数并没有使用这两个参数，不注释掉则编译无法通过。现在需要在代码中使用这两个参数，因此需要去掉注释。

计算文档的页数其实很简单，就是用文档的长度除以页的长度。由于文档长度很可能不会是页长度的整数倍，因此在后面加 1 以保证剩余的部分也能正确地打印。

为了获得页的长度，使用了 CDC 类的 GetDeviceCaps()函数，该函数的原型如下：

```
int GetDeviceCaps( int nIndex ) const;
```

根据参数 nIndex 的不同的取值，GetDeviceCaps()函数可以返回设备的不同的特性。表 6-3 中列出了一些常用的特性：

表 6-3 nIndex 参数

取 值	说 明
DRIVERVERSION	设备驱动程序的版本号
HORZSIZE	设备的宽度（毫米为单位）
VERTSIZE	设备的高度（毫米为单位）
HORZRES	设备的宽度（点数）
VERTRES	设备的高度（点数）
LOGPIXELSX	设备宽度方向上每英寸的点数
LOGPIXELSY	设备高度方向上每英寸的点数

在 OnBeginPrinting()函数中，首先传递 VERTRES 参数至 GetDeviceCaps()函数获得打印纸每页高度方向上的点数，接着计算每个矩形在高度方向上的点数（矩形高度（英寸为单位）*高度方向上每英寸的点数）。这样就可以通过高度方向上“总的点数”除以“每页的点数”得到需要的页数。

如果用户在打印设置对话框中改变了纸张的大小，或打印的方向，或打印机的分辨率，GetDeviceCaps()函数始终返回相应的正确的值，这就保证了始终都能正确地进行打印页数设置。

现在再来编译和运行 TestPrint 程序。这次让程序画 20 个矩形，图 6-14 中是打印预览的“两页”模式：

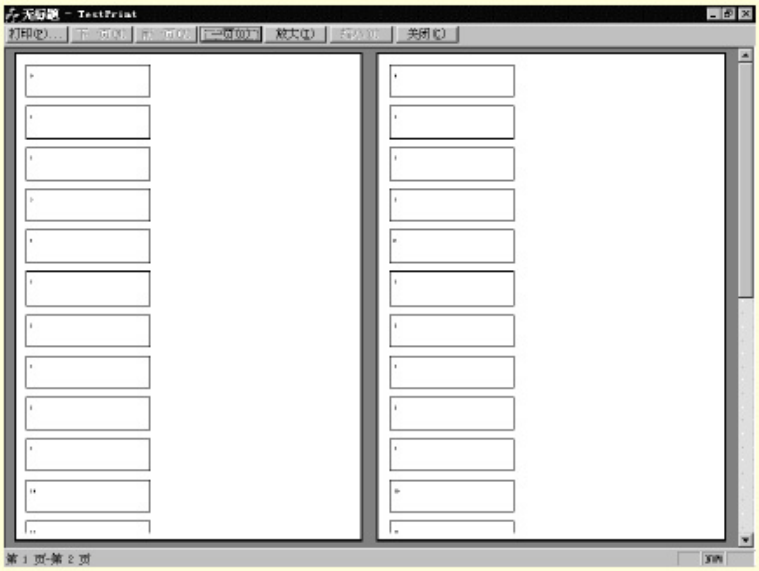

图 6-14　现在 20 个矩形的打印预览还不正确

可以看出，图 6-14 中显然还存在一些不足。虽然已经正确地进行了分页（20 个矩形需要 2 页来打印），但是第 2 页应该接着第 1 页继续打印，而不是象图 6-14 中的那样再次从文档的开始处打印。

6.3.3　设置每页的起点

造成图 6-14 中打印不正确的原因是没有设置打印页每页的起点，结果不管对于打印哪一页，都是从文档的头部开始打印。要正确地让打印继续下去，就必须在开始打印每一页之前，设置打印的起点。

设置打印起点的工作一般在 OnPrepareDC()函数中进行。在 6.2 节中设置映射模式就是在 OnPrepareDC()函数中进行的。

注意传递给 OnPrepareDC()函数的两个参数，pDC 可能指向的是显示设备环境，也可能指向的是打印设备环境。pInfo 参数指向一个 CPrintInfo 对象，如果 pDC 指向显示设备环境，pInfo 参数为 NULL，因为没有任何与打印有关的信息；如果 pDC 指向打印设备环境，pInfo 参数中就保存了当前打印任务的信息，特别地，m_nCurPage 成员指出了当前将要被打印的页。

如果是为显示而使用，应用程序框架在调用 OnDraw()函数之前调用 OnPrepareDC()函数以调整显示属性；如果是为打印或打印预览而使用，应用程序框架在调用 OnPrint()函数打印每一页之前调用 OnPrepareDC()函数。因此在 OnPrepareDC()函数设置打印页的起点是合适的。程序清单 6-10 中是完成了的 OnPrepareDC()函数：

程序清单 6-10　OnPrepareDC()函数

```
void CTestPrintView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
// TODO: Add your specialized code here and/or call the base class
pDC->SetMapMode(MM_LOENGLISH);
if(pDC->IsPrinting())
{
    int pageHeight=pDC->GetDeviceCaps(VERTRES);
    int y=pageHeight*(pInfo->m_nCurPage-1);
    pDC->SetViewportOrg(0,-y);
}
CView::OnPrepareDC(pDC, pInfo);
}
```

在 OnPrepareDC()函数中，在设置了设备的映射模式后，首先使用 CDC 类的 IsPrinting()函数判断是否是因

为打印或打印预览工作而调用本函数。如果是，则进行起点设置；如果不是，不影响在屏幕上的显示。计算当前页的起点是简单的，只需获得每页的高度，已经完成的打印高度就是当前页的起点。需要注意的是SetViewportOrg()函数，该函数的原型如下：

```
virtual CPoint SetViewportOrg( int x, int y );
```

在显示大文档的时候，窗口是无法完全显示整个文档的，但可以通过调用 SetViewportOrg()函数移动视口来观察整个文档。在打印的时候也是如此，大文档无法在一页中完成打印，但可以每次从一定的位置开始打印，以完成打印工作。

现在用户可以再次编译和运行 TestPrint 程序，仍然是画 20 个矩形，观察现在版本的打印预览功能，如图 6-15 所示：

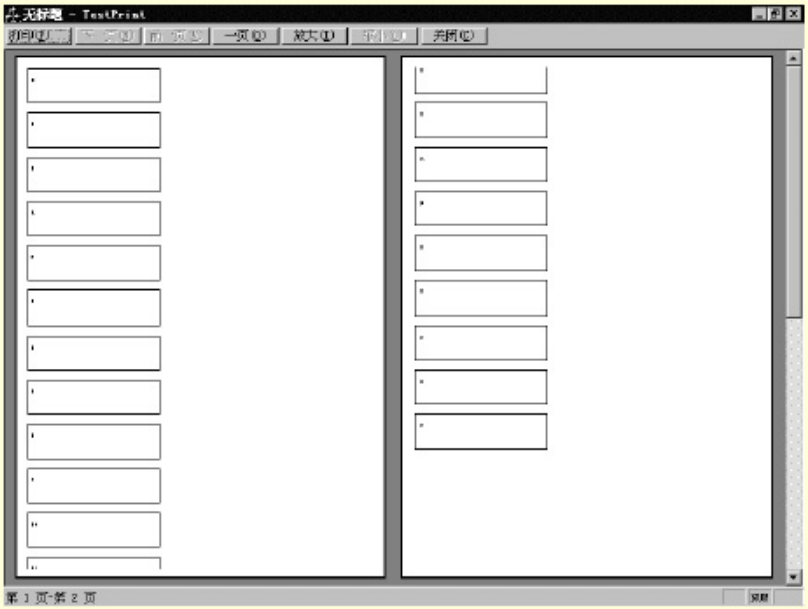

图 6-15　现在 20 个矩形可以正确地打印预览

6.4　MFC 的打印进程

在前面各节中，通过 TestPrint 程序介绍了在 MFC 应用程序中实现打印与打印预览功能的基本知识。在本节中，对 MFC 的打印做一总结性的介绍，以使用户对 MFC 应用程序中的打印功能的实现有更清晰的了解。

在 MFC 应用程序中，由于 Windows 系统提供的设备无关性，视图类 OnDraw()函数中的所有绘图函数对显示设备和打印设备都同样有效。对打印预览而言，只不过是将打印输出模拟到屏幕罢了。

在设计一个具有打印功能的 MFC 应用程序时，用户需要做的工作就是：

- 通知应用程序框架文档有多少页。
- 在需要打印一页文档的时候，正确地绘制文档的相应部分。
- 分配和释放打印时使用的字体资源或其他 GDI 资源。
- 如果需要，在打印特定页前通知应用程序框架改变打印模式，比如说，改变打印的方向。

应用程序框架会在适当的时候调用适当的函数来完成打印功能。CView 类定义了好几个在打印时将被应用程序框架调用的成员函数。在用户的应用程序中，通过在视图类中重载这些函数而在应用程序框架和视图类之间建立联系。

表 6-4 中列出了这些函数：

表 6-4 CView 类的打印函数

函 数	说 明
OnPreparePrinting()	设置打印对话框的初始值，特别是文档的长度
OnBeginPrinting()	分配字体或其他 GDI 资源
OnPrepareDC()	调整给定页的设备环境属性，或是在打印过程中分页
OnPrint()	打印给定页
OnEndPrinting()	释放 GDI 资源

当然用户也可以在其他的函数中进行与打印有关的工作，但上述函数驱动打印工作正确完成。图 6-16 显示了打印过程的步骤，并显示了 CView 类的各成员函数是在何时被调用的。

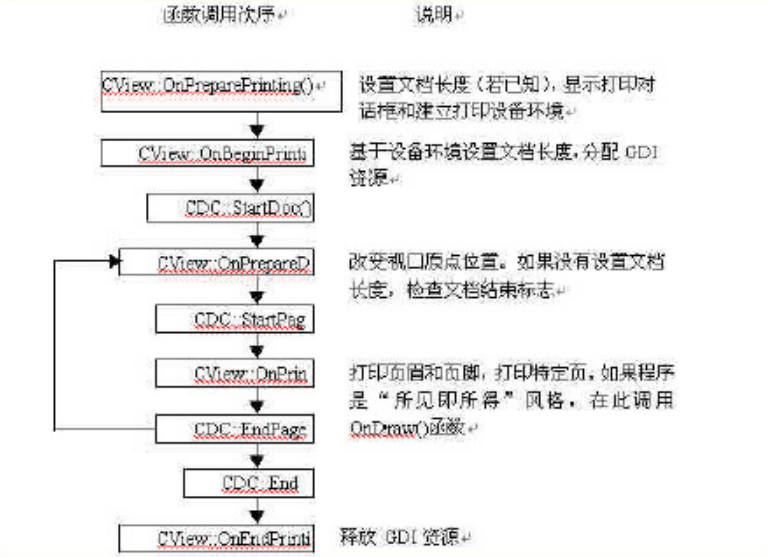

图 6-16 MFC 打印过程

前面已经通过 TestPrint 程序介绍过打印分页的一些基本知识，这里结合图 6-16 再做更深入地说明。

在打印工作开始时，应用程序框架调用视图类的 OnPreparePrinting()函数，并传递一个指向 CPrintInfo 对象的指针。AppWizard 为 OnPreparePrinting()函数生成一个缺省的版本，在该版本中仅调用了视图类的另一个函数 DoPreparePrinting()。DoPreparePrinting()函数显示打印对话框并创建一个打印设备环境。

这时候应用程序还不知道文档究竟有多少页，它使用缺省的值，1 为第一页，0xFFFF 为最后一页。如果用户此时已经知道文档的长度，重载 OnPreparePrinting()函数并在调用 DoPreparePrinting()函数之前调用 pInfo 的 SetMaxPage()成员函数。这使得用户可以设置文档的长度。

DoPreparePrinting()函数显示打印对话框。当该函数返回时，CPrintInfo 对象中保存了用户设定的值。如果用户只希望打印一定范围内的页，那么应该在打印对话框中设定打印的起始页和结束页。应用程序通过调用 CPrintInfo 的 GetFromPage()函数和 GetToPage()函数获得设定值。如果用户没有设定打印范围，应用程序调用 CPrintInfo 的 GetMinPage()函数和 GetMaxPage()函数，并打印整个文档。

对将要打印的每一页，应用程序框架都会调用视图类的两个成员函数：OnPrepareDC()函数和 OnPrint()函数，并传递给这两个函数两个参数：一是指向 CDC 对象的指针，一是指向 CPrintInfo 对象的指针。当应用程序框架每次调用这两个函数的时候，通过 CPrintInfo 对象的 m_nCurPage 成员传递过来的值是不同的。应用程序框架通过这种方式告知视图类哪一页将被打印。

OnPrepareDC()函数也用于显示设备环境，该函数在真正执行绘图任务前调整设备环境的属性。在打印任务中，OnPrepareDC()函数的作用也是类似的。但存在着两点差别：一是此时 CDC 对象代表的是打印设备环境，而非显示设备环境；一是 CPrintInfo 对象作为第二个参数被传递过来（在显示任务时该参数为 NULL）。重载 OnPrepareDC()函数可以调整与打印页有关的设备环境的属性。比如说，可以改变视口位置或剪贴区域已保证打印文档的相应部分。

OnPrint()函数完成真正的打印一页的工作，在 OnPrint()函数中又调用了 OnDraw()函数。重载 OnPrint()函数可以进行只对打印才有用的修饰工作。比如说，可以在 OnPrint()函数进行页眉与页脚的打印，然后调用 OnDraw()

函数完成打印和显示都需要的工作。

图 6-16 清楚地展现了 MFC 应用程序打印过程的步骤及各有关函数的调用顺序。如果用户需要实现更完美的打印和打印预览功能，还需要在图 6-16 的基础上做更多的努力。

6.5 本章小结

MFC 通过 CView 类实现了对打印和打印预览功能的支持，简单地覆盖 OnDraw()函数就可以实现最基本的打印功能。OnDraw()函数既可以在屏幕上绘图，也可以在一个真实的打印机设备环境上绘图，还可以在一个模拟打印机的设备环境上绘图。在基本打印功能的基础上，用户可以添加自己的代码以实现较为复杂的打印功能。当然，用户如果要达到完美的输出效果，肯定需要付出较多的努力。

第七章 使用文档/视图结构

MFC 框架中最容易被程序员和用户两者都见到的部分就是文档和视图。你在利用应用框架编写程序时的大部分工作是编写你的视图类和文档类。因此，我们有必要详细了解文档/视图结构。

文档/视图结构是基于 MFC 库的应用的一个重要特性。它将的实质就是将数据本身与用户对数据的观察和操作分离开来。所有的数据变化都在文档类中进行管理，同时它为视图对它的访问提供一个接口。视图调用两者间的接口来响应用户的操作，进行数据的修改，并且不断更新对文档的显示。这样就允许对同一数据可以有多个视图。了解这一特性对于我们的应用开发有着极为重要的意义。文档与视图的关系可用图 7-1 表示。

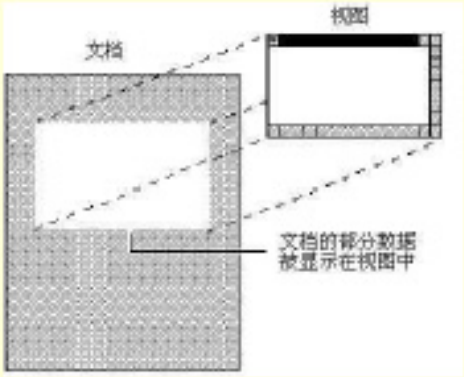

图 7-1 文档与视图的关系

文档，关联的视图以及包含视图的主框架窗口由一个文档模板创建。文档模板有责任创建和管理一个文档类的所有文档对象。

我们将结合单文档以及多文档的应用来详细阐述这些内容。

7.1 分析文档/视图结构

我们首先分析由 AppWizard 自动创建的一个 SDI 应用程序，了解有关文档和视图的一些最基本的东西。

7.1.1 建立一个应用程序

首先我们利用 AppWizard 建立一个基于单文档应用的项目 SingleDoc1。具体的操作步骤如下：

（1）首先在 File 菜单中单击 New，单击 Project 标签，选择 MFC AppWizard(exe)。在 Project Name 框中键入 SingleDoc1。

（2）单击 OK 进入下一个对话框。选择基于单文档的应用，并在 Language 下拉框选取英语。注意有一个 Document view architechture support？（支持文档/视图结构吗？）的复选框，单击它前面的方框，使其中出现一个勾的记号，表示选中了该项。

（3）单击 Next 进入下一对话框，选择 None。

（4）单击 Next 进入下一对话框，清除 ActiveX Control 前面的复选标志。

（5）单击 Next 进入下一对话框，选择系统的缺省设置。

（6）单击 Next 进入下一对话框，分别选取“ MFC Project ”,“ Yes,please ” ,“ As a share DLL ”项。

（7）单击 Next，Appwizard 在其中列出了所有将要产生的类及文件信息。单击 Finish。

（8）出现一个消息框显示有关项目的信息。单击 OK，产生项目 SingleDoc1 。

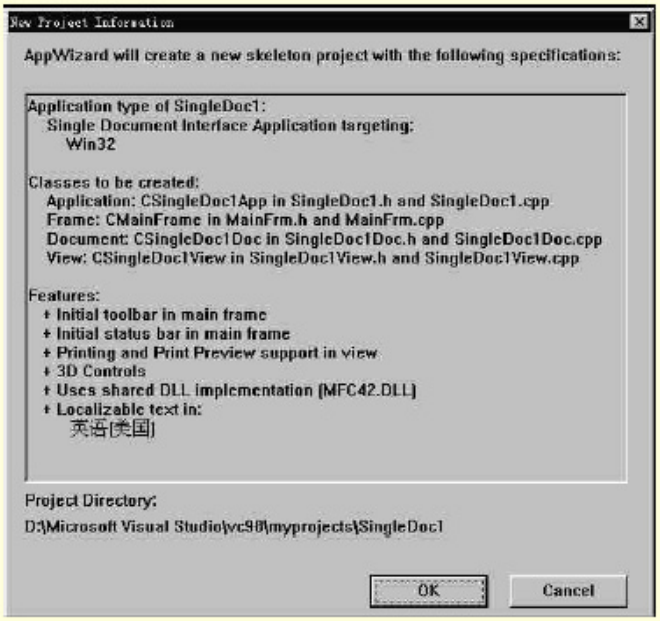

图 7-2 SingleDoc1 项目的信息

这样，我们就建立了一个基于单文档应用的项目。项目的有关信息如图 7-2 所示。接下来我们将结合 AppWizard 产生的代码讲述一些基本的知识。

7.1.2 程序运行的流程

前面我们曾经简要介绍过由 AppWizard 产生的各种代码的含义。但是没有介绍整个程序的运转流程。了解这一点能够帮助您对 Windows 编程的深入理解。

当应用程序开始运行时，它会首先查找程序中的静态和全程变量。其中最重要的就是 CWinApp 类所创建的实例。由 AppWizard 创建的代码如下所示：

```
// The one and only CSingleDoc1App object
CSingleDoc1App theApp;
```

该代码首先通过初始化数据段来建立全局变量，以及建立一些 MFC 内部使用的对象，然后执行 CWinApp 类的构造函数。

当 CWinApp 类的构造函数执行时，应用实例并没有进入正常的运行。所以不要在这些构造函数中做重要的操作，但是你可以在这里初始化变量。如果你在这里进行 CWnd 类的操作，可能出现错误，因为实际的 CWnd 类对象尚未创建完毕。

在所有的静态对象的构造工作完成以后，运行时间库就会调用 WinMain 函数。该函数使我们想起 Dos 下的程序中的 main 函数。的确，两者有许多的类似之处。它是每一个 Windows 应用程序都必须具备的。我们无法在 AppWizard 产生的源代码当中找到它，因为它被隐藏在应用框架内部了。

WinMain 函数进行应用的初始化。它首先调用 CWinApp 类的 InitApplication 函数和 InitInstance 函数。InitInstance 函数会完成一系列的初始化工作，其中最重要的是完成主框架窗口的构造和显示工作。

在工作完成后，它会调用 CWinApp 类的 Run 函数，缺省状态下为 CWinThread::Run()，来进入应用程序的消息循环。这时，它开始等待消息并对接收到的消息调用进行分类，然后将它传给相应的消息处理函数相应的处理。

当它接收到表示程序终止的 WM_QUIT 消息时，MFC 会调用 CWinApp 类的 ExitInstance 函数。随后调用静态对象的析构函数，删除包括 CWinApp 对象在内的全局和静态对象。最后将控制权交还给操作系统。

7.1.3 框架窗口类

在 AppWizard 产生的文件中，我们可以找到 MainFrm.cpp 和 MainFrm.h 文件。在其中定义了一个 CMainFrame

的类。

这个 CMainFrame 定义了程序的主框架窗口。它是由框架窗口类 CFrameWnd 派生出来的，而 CFrameWnd 又是由窗口类 CWnd 派生出来的。对于单文档应用来讲，主框架窗口定义了一切的应用界面的特性。它是程序与用户之间交换信息的主要界面。

当程序的一个新的实例运行时，系统根据文档模板生成新的对象，这些对象包括一个新的 CMainFrame 对象、一个新的 CSingleDoc1Doc 对象、一个新的 CSingleDoc1View 对象。屏幕上出现的是标准框架窗口，接着生成文档，接着生成视图。

由 AppWizard 产生的主框架窗口包含了菜单项、工具栏、状态栏和一个用户区。视图窗被作为子窗口附着在框架窗口上，覆盖了框架窗口的用户区。用户主要在用户区中工作，但是用户可以选择框架窗口的菜单命令对视图窗产生影响。

当应用程序初始化主框架窗口时，会向主框架窗口发送一个 WM_CREATE 的消息，要求创建主框架窗口。然后主框架窗口调用 OnCreate 函数来处理该消息。在 OnCreate 函数中，主要对主框架窗口进行一些初始化。在调用了该函数后，系统再调用 ShowWindow 函数，就可以将主框架窗口显示在屏幕上。值得指出的是，本函数必须在主框架窗口对象创建以后才能使用。

下面列出了 AppWizard 缺省产生的 OnCreate 函数：

程序清单 7-1　OnCreate 函数代码

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
if (!m_wndToolBar.CreateEx(this,TBSTYLE_FLAT,WS_CHILD | WS_VISIBLE
|CBRS_TOP|CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY|
CBRS_SIZE_DYNAMIC)||!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
}
if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
{
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
}
// TODO: Delete these three lines if you don't want the toolbar to
//    be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;
}
```

可以看出，在函数中，创建了工具条、状态条。框架窗口负责为程序处理菜单、状态条和工具条。而这些菜单命令都已经被连到了对应的函数上，因此，用户不用在去处理各个菜单，只要将精力集中在文档和视图上

就可以了。

另外，我们这里介绍的是 SDI 的框架窗口，将来我们还会讲到 MDI 的框架窗口，它和 SDI 的框架窗口有较大的不同，容我们以后再介绍。

7.1.4 文档模板

文档模板被用于指明程序的一个新的实例、一个框架窗口、一个文档、以及一个视图之间的某种联系。它通常在 InitInstance 函数中被创建。然后由它来创建文档对象和主框架窗口。

在用 AppWizard 产生的 SDI 程序的代码中，已经创建了一个单文档模板并将它加入到应用程序中，如下所示：

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CSingleDoc1Doc),
        RUNTIME_CLASS(CMainFrame),          // main SDI frame window
        RUNTIME_CLASS(CSingleDoc1View));
AddDocTemplate(pDocTemplate);
```

我们对该段代码解释如下：

（1）声明了一个指向 CSingleDocTemplate 的指针，使用 new 在堆中产生模块对象。只要应用程序存在模块对象，就应保留在内存中，即使在调用 AddDocTemplate 后不直接使用指针。

（2）用四个参数调用构造函数，包括资源 ID、文档、框架和视图类的类信息结构，后三个参数由 RUNTIME_CLASS 宏提供。

（3）调用 AddDocTemplate，用 MFC 指定模板，并将该模板放入一个由系统保持的可用文件模板列表中。

构造函数函数的声明如下：

```
CSingleDocTemplate( UINT nIDResource, CRuntimeClass* pDocClass,
CRuntimeClass* pFrameClass, CRuntimeClass* pViewClass );
```

其中：

- nIDResource：用于该文档类型的资源标识符
- pDocClass：指向文档类的指针
- pFrameClass：框架类的指针
- pViewClass：视图类的指针

函数动态的生成一个文档模板的对象。

在加入了单文档模板对象后，我们就在 CSingleDoc1Doc、CMainFrame、CSingleDoc1View 之间建立起了某种联系。你的应用程序将使用由 MFC 为产生新文档和视图提供的缺省功能。这些工作你不必过问，只要让 MFC 去做就行了。

7.1.5 文档类

文档就是指一组数据的集合。用户常常使用 File 菜单中的 Open 命令来打开它，用 File 菜单中的 Save 命令来保存它。在 MFC 中，由文档类 CDocument 来实现文档操作。派生文档类一般用来完成对文档对象数据的读和写工作。

AppWizard 为本项目生成了一个文档类 CSingleDoc1Doc，它派生于 MFC 的文档类 CDocument。它在 SingleDoc1Doc.h 中被如下声明：

程序清单 7-2 SingleDoc1Doc.h 的部分代码

```
// SingleDoc1Doc.h : interface of the CSingleDoc1Doc class
class CSingleDoc1Doc : public CDocument
{
```

```
protected: // create from serialization only
CSingleDoc1Doc();
DECLARE_DYNCREATE(CSingleDoc1Doc)
// Attributes
public:
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CSingleDoc1Doc)
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CSingleDoc1Doc();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
CString m_string;
// Generated message map functions
protected:
//{{AFX_MSG(CSingleDoc1Doc)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

现在它只包含了一个类定义的基本框架。注意宏 DECLARE_DYNCREATE **的包含，它使得该类可以动态的生成。要在文档中存储数据，只需要简单地向类的定义中加入数据成员。**

7.1.6 视图类

视图就是附属于某个文档的一个窗口，它扮演着用户和文档之间交换信息的媒体的角色。视图在屏幕上显示一个文档，并且解释用户的输入来对文档进行相应的操作。视图也显示图象以用于打印和打印预览。

它在 MFC **中由** CView **类来支持。该类通常表示一个依附于框架窗口并占据它的用户区的子窗口。根据自己的需要，我们也可以选择一些由** CView **类派生出来的类作为基类，它们都有着自己的特性。例如，使用** CScrollView **类可以使视图支持滚动条；使用** CFormView **支持表格风格；使用** CEditView **能够方便地编辑文本。**

AppWizard **为本项目生成了一个文档类** CSingleDoc1Doc**，它派生于** MFC **的文档类** CDocument**。它在** SingleDoc1View.h **中被如下声明：**

程序清单 7-3 SingleDoc1View.h

```
// SingleDoc1View.h : interface of the CSingleDoc1View class
//
```

```
////////////////////////////////////////////////////////////////////////
class CSingleDoc1View : public CView
{
protected: // create from serialization only
CSingleDoc1View();
DECLARE_DYNCREATE(CSingleDoc1View)
// Attributes
public:
CSingleDoc1Doc* GetDocument();
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CSingleDoc1View)
public:
virtual void OnDraw(CDC* pDC);  // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CSingleDoc1View();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
//{{AFX_MSG(CSingleDoc1View)
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in SingleDoc1View.cpp
inline CSingleDoc1Doc* CSingleDoc1View::GetDocument()
  { return (CSingleDoc1Doc*)m_pDocument; }
#endif
```

现在它只包含了一个类定义的基本框架。我们可以看到，其中定义了两个很重要的函数：OnDraw **和** GetDocument**。前者用于重绘窗口，后者用于获得指向视图对应的文档的指针。我们将在后面详细介绍这两个函数。实际上我们将来要做的工作主要是向视图中加入各种有用的函数。**

7.1.7 程序员的任务

前面我们已经对于 AppWizard 产生的程序框架有了一个大致的了解。我们也可以试着运行一下程序看看它能做些什么。总结起来，程序框架能够完成以下任务：

- 自动创建文档模板、视图、文档和框架窗口。
- 初始化框架窗口，构建菜单，并将每个菜单项和相应的函数（函数可能还是空的，需要程序员去加入代码）连接起来。创建状态条和工具条。
- 在程序刚运行时自动的打开一个文档和一个视图窗口。

简单的说，它为我们生成了一个标准的框架，省去了我们的不少麻烦。

因此，在开发应用时我们只需将精力集中在我们的任务上，主要包括以下内容：

- 为文档加入数据成员，定制一些重要的函数使菜单能够进行实质性的操作。
- 为视图编写一些成员函数，来支持用户对文档的修改，并正确的显示文档。

至此我们大致介绍了文档/视图结构的一些基本概念和我们需要研究的重点。下一节我们将结合一个单文档编辑器来讲述如何实现文档/视图结构。

7.2 单文档应用

7.2.1 单文档与多文档

常用的文档应用程序分为单文档应用和多文档的应用。单文档应用在任一时刻只能处理一个文档和显示该文档的一个视图窗；而多文档应用可以处理几个不同的文档。由于处理方式不同，其相应的程序编制也不太一样。

我们将首先从比较简单的单文档程序入手，针对上一节建立的程序框架加入一些代码，以实现一定的功能。我们要做的就是在视图中键入字符，并在屏幕上显示出来。

7.2.2 在文档中加入数据变量

我们需要在文档中加入一个数据变量。这样，当用户在视图中键入字符时，我们就可以保存用户输入的字符。

我们将加入一个 CString 类的数据成员来存储字符。

CString 类是 MFC 中的一个类，它包含一个可变长度的字符序列。它主要用于字符串的操作。由于提供了串联、替换、比较等运算符，再加上简化的内存管理，使得 CString 类比普通的字符串要好用的多。

在浏览窗口中单击 File View 标签，打开文件浏览窗口。单击以展开 Single Doc1 Files 图标，展开 Header Files 图标，并双击图标 SingleDoc1Doc.h，这时 SingleDoc1Doc.h 头文件被打开。从中找出文档类 CSingleDoc1Doc 的声明如下：

程序清单 7-4 文档类 CSingleDoc1Doc 的声明

```
class CSingleDoc1Doc : public CDocument
{
protected: // create from serialization only
CSingleDoc1Doc();
DECLARE_DYNCREATE(CSingleDoc1Doc)

// Attributes
public:

// Operations
```

```
public:
……
```

在其中加入一个数据成员 m_TextString，键入下面发灰的代码：

```
// Attributes
public:
CString m_TextString;
```

现在文档类 CSingleDoc1Doc 具有了一个公有的数据成员。

7.2.3 在视图中处理键盘输入

接下来我们讨论如何接收用户的输入。根据文档/视图结构的要求，我们应该在视图中处理用户的输入。

在 Windows 的窗口环境下，屏幕上可能有很多窗口，但是只有一个窗口处于活动状态，可以接受当前的键盘输入。我们称为具有当前的焦点。当用户使用键盘进行输入的时候，一串键盘消息会发送给具有当前焦点的窗口。

键击消息有 5 个，如下所示：

- WM_KEYDOWN：非系统键被按下。非系统键就是指在按下某键的同时没有按着 Alt 键。
- WM_KEYUP：非系统键被释放
- WM_SYSKEYDOWN：系统键被按下
- WM_SYSKEYUP：系统键被释放
- WM_CHAR：当 WM_KEYDOWN 的消息被转换成键的代码后发送，它包含了被按下的键的代码

由于我们的输入内容是文本，因此不需要处理系统按键。这样，我们就有两种选择：处理 WM_KEYDOWN 或者处理 WM_CHAR。我们将对两者进行比较。

窗口类的成员函数 OnKeyDown 用于处理 WM_KEYDOWN 消息，其原型如下：

```
afx_msg void OnKeyDown( UINT nChar, UINT nRepCnt, UINT nFlags );
```

- nChar：按键的虚拟键编码
- nRepCnt：按键的重复次数
- nFlags：按键的一系列标志位

由于虚拟键编码不能直接在变量中使用，我们需要重载该函数，对消息传入的参数进行一些处理来获得用户输入的字母。

另一个成员函数 OnChar 用于处理 WM_CHAR 消息，其原型如下：

```
afx_msg void OnChar( UINT nChar, UINT nRepCnt, UINT nFlags );
```

与 OnKeyDown 函数不同的是，该函数的第一个参数 nChar 是输入键的 ASCII 编码，这正符合我们的使用要求，避免了在 OnKeyDown 函数中需要进行的烦琐的编码处理和转换。因此就输入文本而言，对 WM_CHAR 消息进行处理是一个很好的选择。

现在我们需要重新编写自己的 OnChar 函数，来将输入的字符存入文档之中。

按照如下步骤来向视图中加入一个 OnChar 函数：

（1）单击 View 菜单项，选择 ClassWizard 菜单。

（2）在 ClassWizard 对话框中，选择 Class name 为 CSingleDoc1View，选择 Object IDs 为 CSingleDoc1View，双击 Messages 列表中的 WM_CHAR，并选择 OK。

（3）下面的 Member Functions 列表中新出现了一项 OnChar 函数。

（4）单击 Edit Code 按钮，对 OnChar 函数的代码进行编辑。

7.2.4 使用视图类的 GetDocument 函数

现在我们开始编辑 OnChar 函数的代码。为了将输入的字符存入文档，我们需要得到指向该文档的指针。下面我们就来论述这一问题。

对每个视图对象而言，只有一个文档对象与之相联系。它的成员函数 GetDocument 允许应用在视图中得到与其相联系的文挡。

GetDocument 的函数声明如下：

```
CDocument* GetDocument( ) const;
```

函数的返回值为一个指向 CDocument 对象的指针，它指向与调用本函数的视图对象相联系的文档对象。

当 AppWizard 产生 CView 类的派生类时，它同时也创建一个 GetDocument 函数来覆盖基类的 GetDocument 函数。它返回的是指向派生文档类的指针，而不是指向 CDocument 类的指针。

新生成的 GetDocument 函数如下：

```
CSingleDoc1Doc* CSingleDoc1View::GetDocument() // non-debug version is inline
{
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSingleDoc1Doc)));
return (CSingleDoc1Doc*)m_pDocument;
}
```

第一句中的 ASSERT 是一个用于查看输出结果的诊断宏，我们将在后面学习它。它对程序的运行没有实质性的影响。m_pDocument 是一个成员变量，它存储了指向文档对象的指针，是属于指向 CDocument 类的指针。而函数将它转换成了指向派生文档类的指针。这样，就不用我们再去进行数据类型的强制转换。

7.2.5 将用户输入的字符存入文档

首先查看一下由 ClassWizard 自动产生的 OnChar 函数，代码如下：

```
void CSingleDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default

CView::OnChar(nChar, nRepCnt, nFlags);
}
```

现在该函数只是简单地调用了基类的 OnChar 函数函数。下面我们要对它进行修改，使它能够将输入的字符存入文档的数据成员 m_TextString 中。

CString 类的加法运算符可以把单个的字符加入到字符串的尾部。这样就能把用户键入的字符连续的存入 m_TextString 中。

修改 OnChar 函数如下，加入发灰的代码：

程序清单 7-5 OnChar 函数代码

```
void CSingleDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CSingleDoc1Doc* pDoc=GetDocument();
pDoc->m_TextString+=nChar;
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

现在键入的字符已经存入文档中。下一步我们需要将它显示在屏幕上。这也是视图应该完成的任务。

7.2.6 使用设备描述表显示文本

现在需要将字符显示在屏幕上。

在 Windows 的数据结构中，有一种设备描述表 CDC 类。它包含了关于设备的绘制属性的信息。例如显示和打印。所有的绘画绘制要求都通过设备描述表来完成。它集成了用于画线，图形和文本的 Windows 的 API 函数。用户通过设备描述表来进行绘制，不需要了解具体的设备。设备描述表还能用于绘制图形到屏幕、打印机和位图文件。

在这当中，由 CDC 类派生出的 CClientDC 类集成了窗口用户区对应的设备描述表的功能。它允许在窗口

用户区显示文本和图形。

由于我们要在窗口用户区中显示文本，我们就需要创建一个 CClientDC 类的对象。该类的构造函数如下：

CClientDC(CWnd* pWnd);

函数的参数 pWnd 是一个指向窗口对象的指针。函数将为该指针指向的窗口的用户区创建一个设备描述表。

要在窗口中显示文本，可调用 CDC 的成员函数 TextOut，它的定义如下：

BOOL TextOut(int x, int y, const CString& str);

- x：文本起点的 X 坐标
- y：文本起点的 Y 坐标
- str：要显示的 CString 对象

函数将指定的字符串在用户区的指定位置开始显示。

下面，修改 OnChar 函数，加入下面发灰的代码：

程序清单 7-6 OnChar 函数

```
void CSingleDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CSingleDoc1Doc* pDoc=GetDocument();
pDoc->m_TextString+=nChar;
CClientDC dc(this);
dc.TextOut(0,0,pDoc->m_TextString );
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

其中，this 是一个 C++的关键字，表示指向当前对象的内部指针。在本例中，它指向 CSingleDoc1View 的对象。

我们首先为视图创建一个附属的设备描述表，然后在里面显示出文本。

现在程序已经能够正常运行了。当键入字符串时，它能够正确地显示在屏幕上。程序的运行如图 7-3 所示。

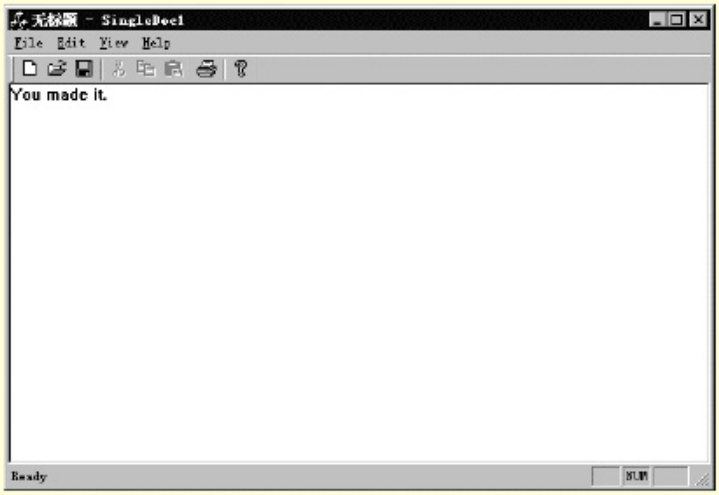

图 7-3 SingleDoc1 程序的运行

但是我们发现，现在的程序与我们的使用习惯还有较大的不同。首先，它应该显示一个插入符指示当前输入文本的位置。其次，它还应该支持用鼠标单击改变插入符的位置，以便调整输入文本出现的位置。

7.2.7 处理 WM_CREATE 消息

下面我们要在程序中加入插入符。在视图窗口刚显示时，该插入符应该显示在屏幕左上角的位置。这就需要在视图窗口显示之前对视图窗口进行初始化。

我们可以在视图中重新定义处理处理 WM_CREATE 消息的 OnCreate 函数。

WM_CREATE 消息是 Windows 发给视图的第一个消息。当应用框架调用 Create 函数时该消息被发送。因此这时窗口创建尚未完成，窗口还不可见。因此，在控制函数 OnCreate 内部，不能调用那些依赖于窗口处于完全激活状态的函数。但是，该函数非常适合做一些窗口对象显示之前的准备工作。

利用 ClassWizard，加入视图类的成员函数 OnCreate 函数。步骤如下：

（1）单击 View 菜单项，选择 ClassWizard 菜单。

（2）在 ClassWizard 对话框中，选择 Class name 为 CSingleDoc1View，选择 Object IDs 为 CSingleDoc1View，双击 Messages 列表中的 WM_CREATE，并选择 OK。

（3）下面的 Member Functions 列表中新出现了一项 OnCreate 函数。

（4）单击 Edit Code 按钮，对 OnCreate 函数的代码进行编辑。

现在我们查看一下由 ClassWizard 自动生成的 OnCreate 函数的框架代码：

程序清单 7-7　ClassWizard 自动生成的 OnCreate 函数

```
int CSingleDoc1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;
    // TODO: Add your specialized creation code here
    return 0;
}
```

现在它只是调用了基类的 OnCreate 函数。

7.2.8　在屏幕上显示插入符

下面我们将对视图类的 OnCreate 函数进行修改，使得窗口刚出现时，就在屏幕左上角显示插入符。

在类 CWnd 中提供了一些成员函数对插入符进行处理。由于 CView 类是由 CWnd 派生出来的，因此我们可以在视图中使用这些函数。主要有以下几个：

- CreateCaret：创建一个插入符
- CreateSolidCaret：创建一个实心的插入符
- DestroyCaret：删除插入符
- HideCaret：隐藏插入符
- SetCaretPos：设置插入符到指定位置
- ShowCaret：显示插入符

为了显示插入符，首先我们应该用 CreateSolidCaret 函数生成一个插入符。该函数的定义如下：

```
void CreateSolidCaret( int nWidth, int nHeight );
```

其中 nWidth 指定了插入符的宽度，nHeight 指定了插入符的高度。

本函数会生成一个新的插入符，同时它会删除系统中原来的插入符（不管它是由哪个窗口创建的），从而保持系统中始终只存在一个插入符。因此，我们只有在窗口具有当前焦点的时候才创建插入符。新生成的插入符缺省地被隐藏，我们需要调用 ShowCaret 函数来显示它。

插入符的宽度和高度通常与系统的字符大小有关。我们希望生成的插入符与字符等高，宽度为它的八分之一。在 MFC 中定义了一个 TEXTMETRIC 的结构，它保存着系统使用的字体特征：

程序清单 7-8　TEXTMETRIC 结构定义

```
typedef struct tagTEXTMETRIC {    /* tm */
int    tmHeight;
int    tmAscent;
```

```
int    tmDescent;
int    tmInternalLeading;
int    tmExternalLeading;
int    tmAveCharWidth;
int    tmMaxCharWidth;
int    tmWeight;
BYTE tmItalic;
BYTE tmUnderlined;
BYTE tmStruckOut;
BYTE tmFirstChar;
BYTE tmLastChar;
BYTE tmDefaultChar;
BYTE tmBreakChar;
BYTE tmPitchAndFamily;
BYTE tmCharSet;
int tmOverhang;
int tmDigitizedAspectX;
int tmDigitizedAspectY;
} TEXTMETRIC;
```

其中成员 tmAveCharWidth **定义了字符的宽度，**tmHeight **定义了字符的高度。**

我们可以调用设备描述表 CDC **的成员函数** GetTextMetric **来得到该结构变量。函数的定义如下：**

BOOL GetTextMetrics(LPTEXTMETRIC lpMetrics) const;

lpMetrics **为一个指向** TEXTMETRIC **的结构的指针。函数将把当前的字体特征保存到该指针指向的变量中。**

在 OnCreate **函数中加入如下发灰的语句来创建光标：**

程序清单 7-9　OnCreate()**函数**

```
int CSingleDoc1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
return -1;
// TODO: Add your specialized creation code here
CClientDC dc(this);
TEXTMETRIC tm1;
dc. GetTextMetrics(&tm1);
CreateSolidCaret(tm1. tmAveCharWidth/8, tm1.tmHeight);
return 0;
}
```

接下来我们需要指定插入符的位置，然后显示插入符。函数 SetCaretPos **的定义如下所示：**

static void PASCAL SetCaretPos(POINT point);

point **是一个** POINT **的结构变量，它定义了插入符的位置坐标。它的定义如下：**

```
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT;
```

POINT **结构包含了两个成员。**x **指定了点的横坐标，**y **指定了点的纵坐标。**

由于插入符需要在屏幕上随着文本的输入不断的后移，我们就需要不断的指定光标的位置。因此，在视图

中加入一个 POINT 类型的数据成员 m_CaretPos 来记录光标的位置是十分必要的。

修改 SingleDoc1View.h 中类 CSingleDoc1View 的声明如下：

程序清单 7-10　类 CSingleDoc1View 的声明

```
class CSingleDoc1View : public CView
{
protected:
// create from serialization only
CSingleDoc1View();
DECLARE_DYNCREATE(CSingleDoc1View)

// Attributes
public:
CSingleDoc1Doc* GetDocument();
POINT m_CaretPos;
……
}
```

在 OnCreate 函数中加入如下发灰的语句来在屏幕的左上角(0,0)处显示前面创建的光标：

程序清单 7-11　OnCreate 函数

```
int CSingleDoc1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
// TODO: Add your specialized creation code here

CClientDC dc(this);
TEXTMETRIC tm1;
dc. GetTextMetrics(&tm1);
CreateSolidCaret(tm1. tmAveCharWidth/8,tm1. tmHeight);
m_CaretPos.x=0;
m_CaretPos.y=0;
SetCaretPos(m_CaretPos);
ShowCaret();
return 0;
}
```

现在运行程序，当视图窗口出现时，一个闪烁的插入符也出现在屏幕的左上角。下面我们将改进 OnChar 函数使插入符随着字符输入而移动。

7.2.9　移动插入符

按照插入符的定义，它指示了当前字符的输入位置。因此，当输入一个字符后，它应该随即移到整个字符串的末尾，指明用户的下一次输入将显示在这里。

我们采用如下的方法。每次获得输入后，在 OnChar 函数中，首先隐藏插入符。向字符串中插入新的字符，然后将新的字符串显示在屏幕上。再计算出新的字符串的长度，将插入符的当前位置设置在新的字符串的末尾。

最后显示该字符串。

由于 Windows 的大多数字体对于不同的字符，显示时使用的宽度也不同。因此不能简单地用字符个数乘以每个字符的宽度。但是，我们可以使用设备描述表的成员函数 GetTextExtent，它的定义如下：

```
CSize GetTextExtent( const CString& str ) const;
```

str 是一个字符串对象的指针。函数计算在当前的字体下字符串中包含的一行文本显示时的高和宽。这些信息被存储在一个 CSize 类的变量中返回。

CSize 类有两个公有成员变量 cx 和 cy，分别代表着宽和高。它们使用的是相对的坐标和位置。我们可以在其他函数中直接获取这两个成员变量。

在隐藏插入符时，使用前面提到的 HideCaret 函数。它的使用很简单，没有参数，也没有返回值。

在 OnChar 函数中加入下面发灰的代码：

程序清单 7-12 OnChar 函数

```
void CSingleDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
HideCaret();
CSingleDoc1Doc* pDoc=GetDocument();
pDoc->m_TextString+=nChar;
CClientDC dc(this);
dc.TextOut(0,0,pDoc->m_TextString);
CSize str_size=dc.GetTextExtent(pDoc->m_TextString);
m_CaretPos.x=str_size.cx;
SetCaretPos(m_CaretPos);
ShowCaret();
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

现在程序可以使插入符随用户的输入移动了。如图 7-4 所示。

图 7-4 插入符随着输入移动

7.2.10 用 DeleteContents 函数进行数据清除

在只能使用一个文档的单文档应用中，一旦用户创建或者打开另一个文档，当前的这个文档对象就要被重新使用，这时系统会调用 DeleteContents 函数来清除当前文档对象中的一些数据。因此，你需要覆盖本函数来删除文档中的数据。对文档的数据成员的一些必要的清除工作都可以放在本函数中进行。在本章中，我们有一

个数据成员 m_TextString 需要清除。

利用 ClassWizard，加入视图类的成员函数 DeleteContents 函数。步骤如下：

（1）单击 View 菜单项，选择 ClassWizard 菜单。

（2）在 ClassWizard 对话框中，选择 Class name 为 CSingleDoc1View，选择 Object IDs 为 CSingleDoc1View，双击 Messages 列表中的 DeleteContents，并选择 OK。

（3）下面的 Member Functions 列表中新出现了一项 DeleteContents 函数。

（4）单击 Edit Code 按钮，对 DeleteContents 函数的代码进行编辑。

现在我们查看一下由 ClassWizard 自动生成的 DeleteContents 函数的框架代码：

程序清单 7-13 ClassWizard 自动生成的 DeleteContents 函数

```
void CSingleDoc1Doc::DeleteContents()
{
// TODO: Add your specialized code here and/or call the base class

CDocument::DeleteContents();
}
```

AppWizard 产生的 DeleteContents 函数只是简单的调用基类的函数，什么实际的工作也没有做。

在 DeleteContents 函数中加入如下的代码：

程序清单 7-14 DeleteContents 函数

```
void CSingleDoc1Doc::DeleteContents()
{
// TODO: Add your specialized code here and/or call the base class

m_TextString.Empty();
CDocument::DeleteContents();
}
```

Empty 是 CString 类的一个成员函数，调用它会使字符串对象被清空。函数的调用很简单，没有参数。

7.2.11 用 OnNewDocument 函数进行初始化

菜单中的 File/New 命令会调用本函数。缺省的实现会调用 DeleteContents 成员函数来确保文档对象是空的，然后将该新文档标记为空的。目前由 AppWizard 产生的 DeleteContents 函数如下：

程序清单 7-15 AppWizard 产生的 DeleteContents 函数

```
BOOL CSingleDoc1Doc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
        return FALSE;
// TODO: add reinitialization code here

// (SDI documents will reuse this document)
return TRUE;
}
```

它简单地调用了基类的函数。

如果用户在单文档应用中选择 File/New 命令，因为应用只允许存在一个文档对象，应用框架会使用本函数来对现存的文档对象重新初始化，而不是创建一个新的文档对象。在这种情况下，用户应该覆盖本函数来为新的文档初始化数据结构，注意调用基类的 OnNewDocument 。

不要把初始化的代码放在文档对象的构造函数中，因为在整个 SDI 应用程序运行期间，构造函数只被调用一次。这样，当用户选择 File/New 命令重新初始化文档时，无法调用构造函数来进行初始化。

如果是在多文档应用中，应用框架会产生一个新的文档对象，并用本函数对它进行初始化。由于每次都会调用文档类的构造函数，因此可以将初始化的代码放在构造函数中，而不对本函数做任何改动。

本例中的文档不需要进行初始化。

7.2.12 用鼠标定位插入符

Windows 应用程序的重要特征是可以用鼠标完成大部分的操作。因此，在输入中我们也应该支持用户用鼠标进行插入符定位。当用户在视图窗口中单击某处时，插入符应该定位到该位置。

前面已经介绍了常用的鼠标消息。对应于鼠标的单击，Windows 会向视图类发送 WM_LBUTTONDOWN 的消息。因此，我们只要对该消息的处理函数 OnLButtonDown 进行重新的定义，在其中加入插入符定位的代码就可以了。

利用 ClassWizard，加入视图类的成员函数 OnLButtonDown 函数。步骤如下：

（1）单击 View 菜单项，选择 ClassWizard 菜单。

（2）在 ClassWizard 对话框中，选择 Class name 为 CSingleDoc1View，选择 Object IDs 为 CSingleDoc1View，双击 Messages 列表中的 WM_LBUTTONDOWN，并选择 OK。

（3）下面的 Member Functions 列表中新出现了一项函数 OnLButtonDown。

（4）单击 Edit Code 按钮，对函数 OnLButtonDown 的代码进行编辑。

现在我们查看一下由 ClassWizard 自动生成的 OnLButtonDown 函数的框架代码：

程序清单 7-16 ClassWizard 自动生成的 OnLButtonDown 函数

```
void CSingleDoc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default

CView::OnLButtonDown(nFlags, point);
}
```

其中，nFlags 为一个无符号整数，表明鼠标按钮的状态。

参数 point 是一个 CPoint 类的对象，存储了鼠标左键单击的屏幕位置的坐标。CPoint 与前面我们提到的 POINT 结构类似，有两个公用成员变量 x 和 y，分别存储屏幕位置的横坐标和纵坐标。该 CPoint 类定义了赋值运算符，使得可以将一个对象的值用赋值运算符直接赋给另一个对象。

当用户单击鼠标左键后，表明下面输入的文本将从一个新的位置开始显示。因此，我们需要定义一个新的成员变量 m_BeginPoint 来记录显示文本的起始位置。

修改 SingleDoc1View.h 中类 CSingleDoc1View 的声明如下：

程序清单 7-17 类 CSingleDoc1View 的声明

```
class CSingleDoc1View : public CView
{
protected:
// create from serialization only
CSingleDoc1View();
DECLARE_DYNCREATE(CSingleDoc1View)
```

```
// Attributes
public:
CSingleDoc1Doc* GetDocument();
POINT m_CaretPos;
CPoint m_BeginPoint;
……..
```

同时，由于从新位置开始显示的字符串不再包括在鼠标定位插入符以前键入的字符串。我们需要将字符串对象 m_TextString 清空，然后从 m_BeginPoint 定义的位置开始显示后面输入的字符。

利用 CString 类提供的成员函数可以将该字符串对象清空为一个空字符串。该函数的使用很简单，没有参数，也没有返回值。

修改 OnLButtonDown 函数如下：

程序清单 7-18　OnLButtonDown 函数

```
void CSingleDoc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default

HideCaret();
m_BeginPoint=point;
CSingleDoc1Doc* pDoc=GetDocument();
pDoc->m_TextString.Empty();
m_BeginPoint=point;
m_CaretPos=point;
SetCaretPos(m_BeginPoint);
ShowCaret();
CView::OnLButtonDown(nFlags, point);
}
```

函数首先隐藏插入符，然后将插入符定位到新的位置，最后在新的位置重新显示插入符。值得一提的是，对于 CPoint 的对象 m_BeginPoint，我们直接使用赋值运算符将 point 的值传给了它。

相应地，我们也需要对 OnChar 函数加以修改。当用户用鼠标进行插入符的定位后，它需要从新的位置 m_BeginPoint 开始显示字符串。并且，当用户键入字符后的插入符定位也要加以改变，用 m_BeginPoint 加上当前字符串的显示长度来得到新的插入符位置。修改后 OnChar 函数的代码如下：

程序清单 7-19　OnChar 函数

```
void CSingleDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default

HideCaret();
CSingleDoc1Doc* pDoc=GetDocument();
pDoc->m_TextString+=nChar;
CClientDC dc(this);
dc.TextOut(m_BeginPoint.x,m_BeginPoint.y,pDoc->m_TextString);
CSize str_size=dc.GetTextExtent(pDoc->m_TextString);
m_CaretPos.x=m_BeginPoint.x+str_size.cx;
```

```
SetCaretPos(m_CaretPos);
ShowCaret();
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

另外，我们还需要在 OnCreate **函数中对** m_BeginPoint **进行初始化，修改** OnCreate **函数的代码如下：**

程序清单 7-20　OnCreate **函数**

```
int CSingleDoc1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
return -1;

// TODO: Add your specialized creation code here

CClientDC dc(this);
TEXTMETRIC tm1;
dc.GetTextMetrics(&tm1);
CreateSolidCaret(tm1.tmAveCharWidth/8,tm1.tmHeight);
m_CaretPos.x=0;
m_CaretPos.y=0;
m_BeginPoint=0;
m_BeginPoint=0;
SetCaretPos(m_CaretPos);
ShowCaret();
return 0;
}
```

现在再运行程序。先在视图窗口中进行输入，然后在其它位置上单击鼠标并继续进行输入，程序的运行如图 7-5 **所示。**

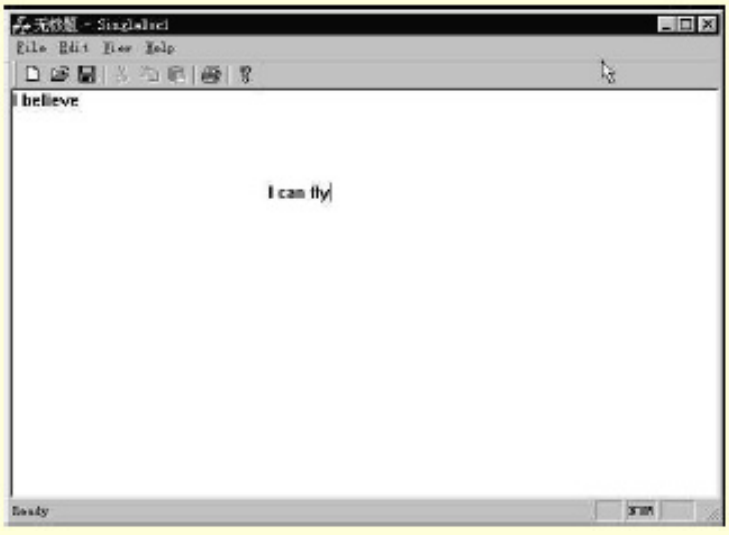

图 7-5　**用鼠标定位插入符**

到现在为止，这个 SDI **应用程序就全部完成了。下一节我们将结合一个多文档的编辑器的开发，讲述多文档的应用。**

7.3 多文档应用

在前一节中我们建立了一个单文档的应用。它能够进行一些简单的文本输入。在这一节里，我们将建立一个新的项目 MultiDoc1。它是一个基于多文本的应用。该项目与前一节中建立的文本编辑器有很多相似之处，当然也有许多不同。最大的不同之处是 MultiDoc1 可以支持打开多个文档。

7.3.1 建立一个多文档的应用

首先我们利用 AppWizard 建立一个基于多文档应用的项目 MultiDoc1。具体的操作步骤如下：

（1）首先在 File 菜单中单击 New，单击 Project 标签，选择 MFC AppWizard(exe)。在 Project Name 框中键入 MultiDoc1。

（2）单击 OK 进入下一个对话框。选择基于多文档的应用，并在 Language 下拉框选取英语。

注意：

✍ 有一个 Document view architechture support？（支持文档/视图结构吗？）的复选框，单击它前面的方框，使其中出现一个勾的记号，表示选中了该项。

（1）单击 Next 进入下一对话框，选择 None。

（2）单击 Next 进入下一对话框，清除 ActiveX Control 前面的复选标志。

（3）单击 Next 进入下一对话框，不要改动系统的缺省设置。

（4）单击 Next 进入下一对话框，分别选取“ MFC Project ”,“ Yes，please ” ,“ As a share DLL ”项。

（5）单击 Next，Appwizard 在其中列出了所有将要产生的类及文件信息。单击 Finish。

这样，我们就建立了一个基于多文档应用的项目。项目的有关信息如图 7-6 所示。

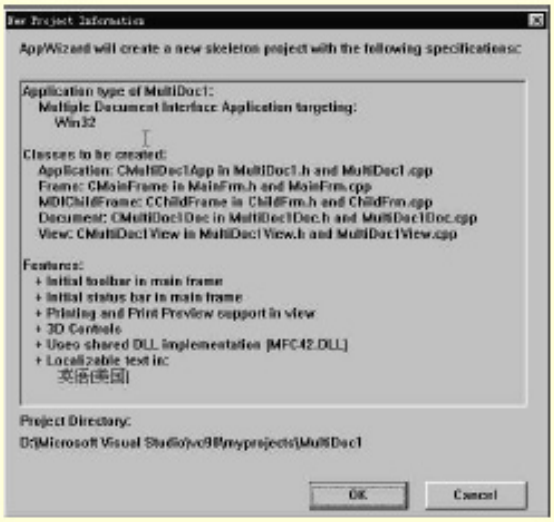

图 7-6 MultiDoc1 的项目信息

显示了将要产生的新项目的一些信息。单击 OK，产生项目 MultiDoc1。

7.3.2 分析 AppWizard 产生的 MDI 框架程序

我们首先运行一下现在的 MDI 框架程序。每当选择一次 File/New 命令，程序都会打开一个新的文档窗口，而不是象在 SDI 中只是清除了当前的文档显示。程序的运行如图 7-7 所示。

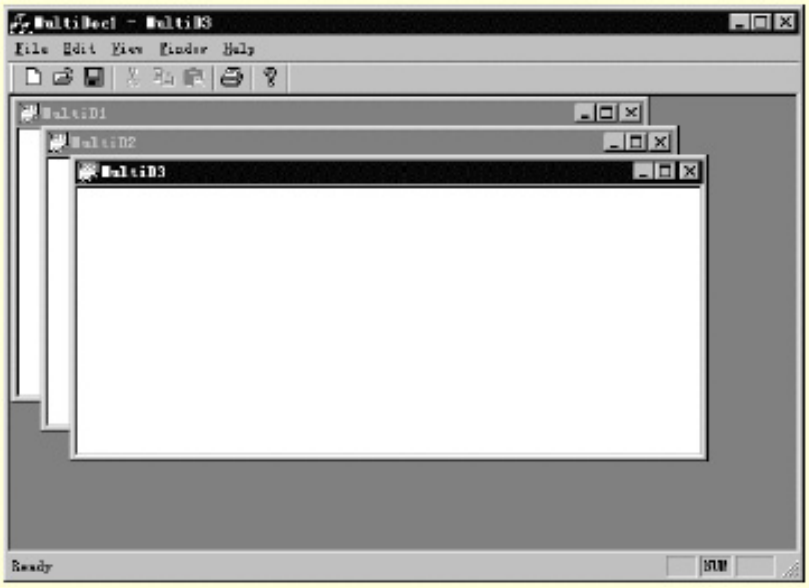

图 7-7 由 AppWizard 生成的 MultiDoc1 程序

下面我们分析一下 AppWizard 自动产生的 MDI 项目的框架，看看与在单文档应用时有哪些不同。

1. 文档模板

在 CMultiDoc1App 类的 InitInstance 函数中（MultiDoc1.cpp），如下定义了多文档应用的文档模板：

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CmultiDocTemplate
(
    IDR_MULTIDTYPE,
    RUNTIME_CLASS(CMultiDoc1Doc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CMultiDoc1View));
AddDocTemplate(pDocTemplate);
}
```

就像在第一节见到的 CSingleDocTemplate 一样，多文档模板类 CMultiDocTemplate 也允许使用多种文档类型，但是它允许同时存在多个文档对象，这是 MDI 应用的最重要的特征。

该文档模板的作用是将一个框架子窗口、一个文档类和一个视图类联系在一起。这个视图类依附于框架子窗口。由于 CChildFrame 已经给传给了 CMultiDocTemplate 类的构造函数，应用框架可以动态的创建应用框架子窗口 CChildFrame 的对象。

另外，MDI 应用可以通过多次调用 AddDocTemplate 函数来支持多个文档模板，每个模板可以指定不同的文档类、视图类和 MDI 子框架窗口类的组合。当用户从 File 菜单中选择 New 时，应用框架会显示一个列表框，让用户选择一个模板。而在 SDI 应用中不支持多个文档模板，因为在整个应用期间，文档、视图和框架均只被创建一次。

2. 框架窗口及子窗口

在 SDI 程序中，主窗口 CMainFrame 由基类框架窗口 CFrameWnd 派生出来。由于它的文档模板将视图、文档、主框架窗口联系在一起，每个视图直接属于主框架窗口。整个应用只允许存在一个主框架窗口类和一个主框架窗口对象，因此不允许在程序运行时动态的创建文档模板，进而创建新的文档。所以它只能打开一个文档。

而在 MDI 程序中，存在两种框架窗口：主框架窗口 CMainFrame 和子框架窗口 CChildFrame。主框架窗口与前面 SDI 应用中的主框架窗口相同，而子框架窗口则属于主框架窗口。子框架窗口没有菜单和工具栏，只有一个用户区可以容纳视图窗口。

主窗口 CMainFrame 是由类 CMDIFrameWnd 派生出来的，如下所示：

```
class CMainFrame : public CMDIFrameWnd
```

```
{
DECLARE_DYNAMIC(CMainFrame)
public:
CMainFrame();
.....
```

它在 InitInstance **函数中被创建如下：**

```
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
m_pMainWnd = pMainFrame;
```

对于该主窗口，它可以拥有多个框架子窗口类。而每个框架子窗口类 CChildFrame **和一个文档类、一个视图类通过文档模板联系在一起。视图和文档只属于框架子窗口，并不依附于主框架窗口类。因此，尽管程序中只允许有一个一个主框架窗口类和一个主框架窗口对象，但是却可以存在多个框架子窗口类。因此，只要在运行时动态的创建新的文档模板，就可以产生多个框架子窗口，从而允许存在多个文档对象。正是这种将主框架窗口与视图窗口分离的办法使得它可以支持多个文档。**

尽管程序中存在多个框架子窗口，但是任意时刻，只有一个框架子窗口处于活动状态。用户对于主框架窗口中的菜单和工具条的命令和操作被传给当前活动的框架子窗口。主窗口的标题条中显示的是活动窗口中的文档文件名。

3. **函数** OnFileNew

这个函数对应着菜单中的 File/New **命令，用于创建空文档。对于** MDI **应用来说，每次打开程序时都会自动的打开一个新文档。因此，**InitInstance **函数也要调用** OnFileNew **函数。**

在文档类的文件 MultiDoc1Doc.cpp **中，包含了函数** OnNewDocument **的代码，其代码如下：**

程序清单 7-21　**函数** OnNewDocument **的代码**

```
BOOL CMultiDoc1Doc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
    return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
return TRUE;
}
```

目前它只调用了基类的函数。该函数会调用 DeleteContents **成员函数来确保文档对象是空的，然后将该新文档标记为空的。覆盖本函数来为新的文档初始化数据结构，注意调用基类的** OnNewDocument **。**

如果用户在单文档应用中选择 File/New **命令，应用框架会使用本函数来对现存的文档对象重新初始化，而不是创建一个新的文档对象。在单文档的应用中，必须将初始化的代码放在本函数中而不是放在文档类的构造函数中，因为构造函数只被调用一次。**

如果是在多文档应用中，应用框架会产生一个新的文档对象，并用本函数对它进行初始化。由于每次都会调用文档类的构造函数，因此可以将初始化的代码放在构造函数中。

7.3.3 **增强文本编辑器的功能**

在 SingleDoc1 **中，我们曾经介绍了一个简单的文本编辑器。现在，我们将它稍加修改，加入一些新的功能，使它能够识别回车键并相应的进行换行。由于我们希望继续用** TextOut **函数来直接输出字符串对象，而该函数不能对回车符进行正确的显示，因此我们需要用一种新的文档结构。**

我们用一个字符串对象数组来保存输入文本，每个字符串对象保存一行，并用一个整型变量存储当前的行数。

在文档类 CMultiDoc1Doc（MultiDoc1Doc.h）中加入两个成员变量如下：

程序清单 7-22 MultiDoc1Doc.h 的部分代码

```
const MaxLines=50;

class CMultiDoc1Doc : public CDocument
{
protected: // create from serialization only
CMultiDoc1Doc();
DECLARE_DYNCREATE(CMultiDoc1Doc)

// Attributes
public:
CString m_TextString[MaxLines];
int m_LineCount;
}
```

其中字符串对象数组最多可容纳 50 个字符串。换句话说，我们的文档可容纳用户输入 50 行文本。

在 CMultiDoc1Doc 的构造函数中为 m_LineCount 赋初值为 0：

程序清单 7-23 CMultiDoc1Doc 函数代码

```
CMultiDoc1Doc::CMultiDoc1Doc()
{
// TODO: add one-time construction code here
m_LineCount=0;
}
```

现在可以对 OnChar 函数进行修改了。当用户键入的是回车键，只需要简单的将 m_LineCount 加 1 就可以了。否则，对于 m_TextString[m_LineCount]进行操作，将输入的字符加入到该字符串中，然后将它显示出来，程序的代码如下：

程序清单 7-24 OnChar 函数

```
void CMultiDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CMultiDoc1Doc* pDoc=GetDocument();
if(nChar== ' \r ' )
pDoc->m_LineCount++;
else
{
    pDoc->m_TextString[pDoc->m_LineCount]+=nChar;
    TEXTMETRIC tm;
    CClientDC dc（this）;
    dc.GetTextMetrics(&tm);
```

```
        dc.TextOut(0,pDoc->m_LineCount*tm.tmHeight,pDoc->m_TextString[pDoc->m_LineCount]);
    }
    CView::OnChar(nChar, nRepCnt, nFlags);
    }
```

其中“\r”是回车符的转义字符，其 ASCII 码值与回车符相同。用到的其它函数在前面都介绍过，如果不懂的话，参见第二节中的内容。

7.3.4 设置文档的修改标志

Windows 的许多应用都会跟踪用户对文档的修改。当用户试图关闭文档或者退出程序时，应用就会弹出一个消息框，询问用户是否需要将文档保存。这样可以避免用户疏忽而引起的数据丢失。我们也希望在程序中加入这种功能。

类库应用程序提供了一个 CDocument 类的成员变量 m_bModified 来支持这种功能。它是一个布尔变量，取值为 TRUE 表示文档被修改过，取值为 FALSE 表示文档未被修改。

当用户试图关闭文档或者退出程序时，它自动检查该布尔变量 m_bModified。如果文档被修改过，系统会弹出对话框来要求用户确认，否则，直接退出。

因此，我们只要在文档被修改后对该成员变量 m_bModified 进行相应的操纵就可以实现这种功能，其他部分系统会自动完成。

我们可以通过 CDocument 的成员函数 SetModifiedFlag 和 IsModified 来访问成员变量 m_bModified。

函数 SetModifiedFlag 的声明如下：

```
void SetModifiedFlag( BOOL bModified = TRUE );
```

bModified 就是一个 BOOL 变量，它的值被赋给 m_bModified。

当你对文档作了任何修改时，需要调用本函数。缺省时用参数 TRUE 来调用，表示文档被修改过。如果用 FALSE 调用，则表示要清除修改标志位。

函数 IsModified 用于确定文档在上次存盘后是否被修改过。如果被修改过，返回值为非零值；否则，返回值为零。

由于我们对文档的修改都是在 OnChar 函数中进行，因此我们应该在每次修改文档后设置修改标志，对 OnChar 函数修改如下：

程序清单 7-25 OnChar 函数

```
void CMultiDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CMultiDoc1Doc* pDoc=GetDocument();
if(nChar=='\r')
        pDoc->m_LineCount++;
else
{
        pDoc->m_TextString[pDoc->m_LineCount]+=nChar;
        TEXTMETRIC tm1;
        CClientDC dc(this);
        dc.GetTextMetrics(&tm1);
        dc.TextOut(0,pDoc->m_LineCount*tm1.tmHeight,
pDoc->m_TextString[pDoc->m_LineCount]);
}
pDoc->SetModifiedFlag(TRUE);
CView::OnChar(nChar, nRepCnt, nFlags);
```

}

现在程序可以监测数据的改动了。现在运行程序。在其中一个子窗口中输入一些字符，试着不存盘退出，系统会弹出一个对话框询问是否保存该信息，如图 7-8 所示：

图 7-8 询问是否存盘的对话框

7.3.5 修改视图类的 OnDraw 函数

现在的应用程序可以同时打开多个文档，这样，也就存在多个视图窗。当我们在文档之间切换时，就会有视图窗重新弹出或者露出原来被遮盖的部分，或者用户改变了窗口尺寸。这时系统会自动地调用 OnDraw 函数重新绘制窗口。

OnDraw 成员函数是视图类的一个虚成员函数。每当视图窗需要重新绘制时，应用程序都会调用 OnDraw 函数。该函数没有缺省的实现，我们必须自己编写该函数以便窗口能被正确的显示。

如果应用改变了窗口的数据，也应该调用该函数，这种调用由用户自己处理。我们可以调用视图的成员函数 Invalidate 来通知系统，从而触发对 OnDraw 函数的调用。

Invalidate 是视图从 CWnd 继承的函数。调用该函数会使窗口的整个用户区无效。这样当下一个 WM_PAINT 消息发生时，这个用户区被重画。我们可以调用 CWnd 类的成员函数 UpdateWindow 来向某窗口发送 WM_PAINT 消息的消息。

现在系统已经生成了如下的 OnDraw 函数：

程序清单 7-26 系统生成的 OnDraw 函数

```
void CMultiDoc1View::OnDraw(CDC* pDC)
{
CMultiDoc1Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
}
```

可以看出，现在的 OnDraw 函数并未作任何实质性的工作。

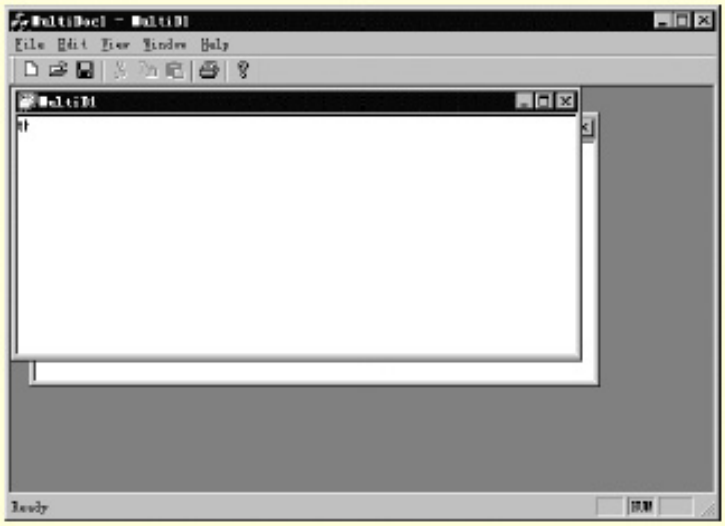

图 7-9 窗口的不正确显示

因此，当某个视图窗口被遮住后重新绘制时，原来被显示的输入字符就不再显示。我们可以作一个演示。

运行现在的程序。打开两个子窗口 MultiD1 和 MultiD2，在其中一个窗口中 MultiD1 中键入“ this is a test.”。然后将焦点移至另一个子窗口 MultiD2，并使它遮住 MultiD1 的一部分。再将焦点移回 MultiD1。这时窗口的显示是不正确的，如图 7-9 所示。

我们必须在 OnDraw 函数中加入一些代码，使它能对当前的文档进行正确地显示。修改 OnDraw 函数如下：

程序清单 7-27 OnDraw 函数

```
void CMultiDoc1View::OnDraw(CDC* pDC)
{
CMultiDoc1Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

// TODO: add draw code for native data here
TEXTMETRIC tm1;
CClientDC dc(this);
dc.GetTextMetrics(&tm1);
for(int p=0;p<=pDoc->m_LineCount;p++)
dc.TextOut(0,p*tm1.tmHeight,pDoc->m_TextString[p]);
}
```

现在再做前面的实验，我们发现，窗口的显示已经正常了。

至此，我们的多文档功能已经被加入到程序中了。在下一节，我们将继续修改这个项目，在其中加入对多窗口的支持。

7.4 多窗口应用

在前面的程序 MultiDoc1 中，我们生成了一个支持多文档的编辑器。在这一节中，我们要介绍如何使用多视图窗。使用多视图窗有两种办法：

- 当用户选择 Window 菜单下的 New Window 菜单，打开一个新的视图窗，并在其中显示出当前文档。
- 利用切分窗口，让用户在同一窗口中处理同一文档的多个视图窗。
- 我们将在下面利用第一种方法程序 MultiDoc1 加以改进，加入对多视图窗功能的支持。

7.4.1 程序框架实现的功能

现在我们首先运行目前的程序 MultiDoc1。在视图窗中键入“ it is a test ”，然后选择 Window 菜单下的 New Window 菜单时，系统会弹出一个新的视图窗，其中显示了当前的文档“ it is a test ”，如图 7-10 所示。

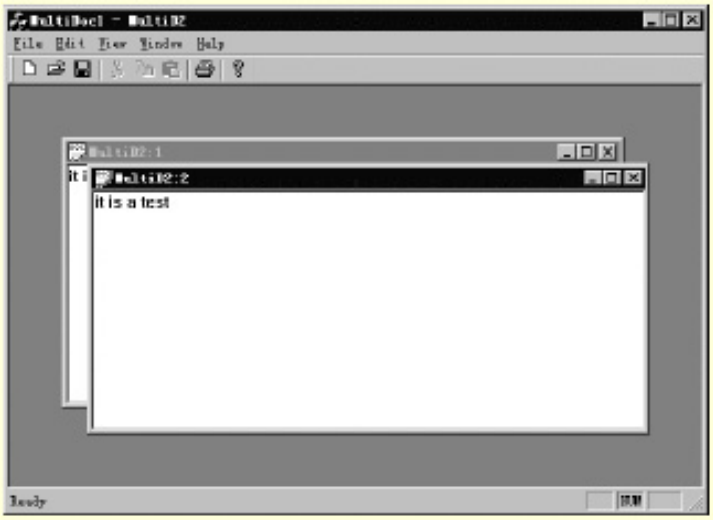

图 7-10 选择 New Window 菜单后弹出的新视图窗

可以看出，创建新视图窗和将该视图窗连接到当前文档的操作由系统自动完成。Window 菜单中的其他命令也由系统自动的支持。这包括 Cascade 、Tile 和 Arrange Icons 等菜单命令。

下面我们进行另一个尝试。选择 Window/Tile 将前面实验中的两个视图窗并列在视图窗中。在其中一个视图窗中键入字符，我们发现，另一个视图窗并没有随着字符的输入而改变它的显示内容。如图 7-11 所示。

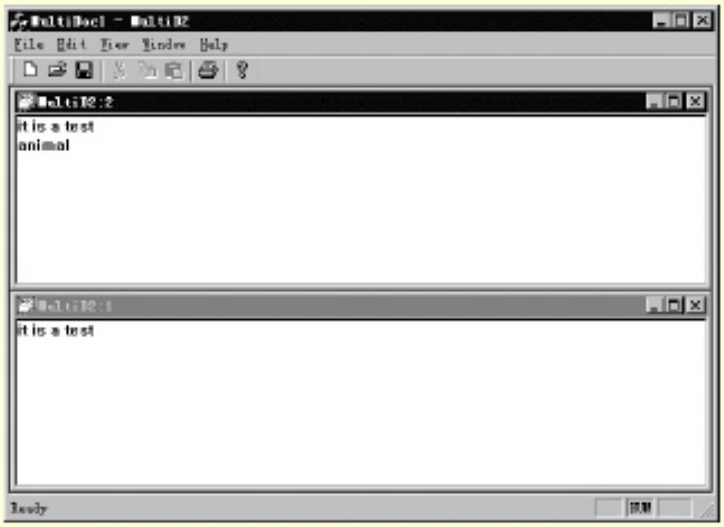

图 7-11 同一文档的视图窗显示不一致

由于这两个视图窗是属于同一个文档的。因此，它们对文档的显示应该相同。AppWizard 产生的程序没有对此功能进行支持，这正是我们在应用程序中应该做的。

7.4.2 使文档和视图保持一致

当应用程序在某个视图中改动了文档内容时，我们需要更新所有与文档相连的视图的显示。在 MFC 中，采用如下的机制使各个视图的显示同步。

当文档类的内容被改变时，调用文档类的成员函数 UpdateAllViews 来给所有与它相连的视图类发送文档数据已被改变的消息。如果文档内容的改变是在某个视图内进行的，我们假定该视图自己已经进行了更新，这样就只需向其他的视图发送消息。

这个消息会导致所有被通知的视图类调用自己的 OnUpdate 函数，对文档发来的消息进行相应的处理。因此，我们要做的工作主要有两个：

（1）在文档数据被改变的地方（通常在视图内或文档内）调用 UpdateAllViews 函数。

（2）为视图编写 OnUpdate 成员函数，取得新的文档数据，并将它显示出来。

7.4.3 在 OnChar 函数中加入 UpdateAllViews 函数

下面我们首先做第一个工作，在文档数据被改变的地方调用 UpdateAllViews 函数。

UpdateAllViews 函数的定义如下：

```
void UpdateAllViews(
CView* pSender,
LPARAM lHint = 0L,
CObject* pHint = NULL );
```

说明：

- pSender：一个指向视图类的指针，它指向修改文档的视图
- lHint：包含一些修改的信息
- pHint：指向存储着修改信息的 CObject 对象的指针

如果该函数在文档类的成员函数中被调用，则 pSender 参数为 NULL，这时函数通知与文档相关的所有视图类。如果函数在视图类的成员函数中被调用，则 pSender 参数为指向该视图对象的指针。关于数据变动的有关信息被通过后两个参数发送。

在本程序中，所有的文档数据修改都是在 OnChar 函数中进行的，因此我们将修改 OnChar 函数，向其中加入 UpdateAllViews 函数。

修改后的 OnChar 函数如下所示：

程序清单 7-28 OnChar 函数

```
void CMultiDoc1View::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CMultiDoc1Doc* pDoc=GetDocument();
if(nChar=='\r')
        pDoc->m_LineCount++;
else
{
        pDoc->m_TextString[pDoc->m_LineCount]+=nChar;
        TEXTMETRIC tm1;
        CClientDC dc(this);
        dc.GetTextMetrics(&tm1);
        dc.TextOut(0,pDoc->m_LineCount*tm1.tmHeight,
       pDoc->m_TextString[pDoc->m_LineCount]);
}
pDoc->SetModifiedFlag(TRUE);
pDoc->UpdateAllViews(this);
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

7.4.4 修改视图类的 OnUpdate 成员函数

现在我们进行第二个步骤：为视图编写 OnUpdate 成员函数，取得新的文档数据，并将它显示出来。

OnUpdate 也是视图类的一个虚成员函数。当应用程序修改了视图对应的文档时，该文档通常调用 UpdateAllViews 函数，从而导致应用框架调用该函数。视图类的 OnUpdate 函数通常用于响应文档中的更新数据显示的要求。包括访问文档，读取文档的新数据，然后更新视图的成员和显示，从而反映文档的变化。可以在 OnUpdate 函数中调用 Invalidate 函数使视图的某一区域无效，触发 OnDraw 函数，用更新后的数据重新绘制窗口。

函数的定义如下：

virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);

- pSenderz：指定不用更新的视图，如果为 NULL，则更新所有视图。这是由 UpdateAllViews 函数传给本函数的参数。
- lHint：包含修改的信息
- pHint：指向存储修改信息的对象的指针

缺省的 OnUpdate 会使窗口的整个用户区无效。这样当下一个 WM_PAINT 消息发生时，这个用户区被重画。在 OnUpdate 函数中通常只需指定需要重画的区域，而不应该直接作任何实际的绘制工作。

利用 ClassWizard，加入视图类的成员函数 OnUpdate 函数。步骤如下：

（1）单击 View 菜单项，选择 ClassWizard 菜单。

（2）在 ClassWizard 对话框中，选择 Class name 为 CSingleDoc1View，选择 Object IDs 为 CSingleDoc1View，双击 Messages 列表中的 OnUpdate，并选择 OK。

（3）下面的 Member Functions 列表中新出现了一项函数 OnUpdate。

（4）单击 Edit Code 按钮，对函数 OnUpdate 的代码进行编辑。

现在我们查看一下由 ClassWizard 自动生成的 OnUpdate 函数,现在它是空的。我们向其中加入代码如下：

程序清单 7-29 OnUpdate 函数

```
void CMultiDoc1View::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
// TODO: Add your specialized code here and/or call the base class
if(this!=pSender)
Invalidate();
}
```

InValidate 函数是一个 CWnd 类的成员函数，其定义如下：

void Invalidate(BOOL bErase = TRUE);

该函数使得窗口的整个用户区无效。这样当下一个 WM_PAINT 消息来到时该窗口被重画。参数 bErase 为布尔变量，缺省值为 TRUE ，这时系统会擦去用户区的当前背景。现在程序能够使文档的各个视图显示一致了。

运行程序，打开同一文档的两个视图窗，并在其中一个视图中键入文本，我们发现，另一个视图窗也更新了自己的显示，如图 7-12 所示。

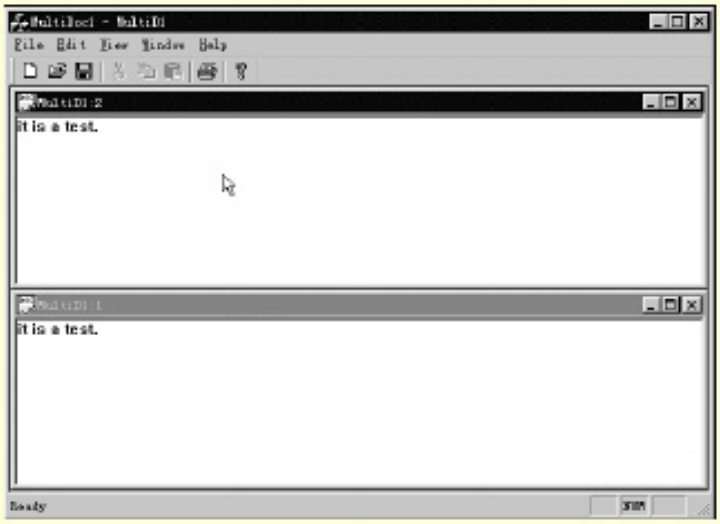

图 7-12 同一文本在两个视图中的同步显示

7.4.5 视图类的 OnInitialUpdate 函数

当视图第一次被分配给一个文档以后，但是视图窗口还没有显示之前，应用框架会调用该函数。换句话说，当应用被启动时，或当用户选择 File/New 命令和 File/Open 命令时，都会调用 OnInitialUpdate 函数。基类函数的缺省实现调用 OnUpdate 函数。如果我们要在派生视图类中重新设计 OnInitialUpdate 函数，一定要在其中调用基类的 OnInitialUpdate 函数，或者是派生类的 OnUpdate 函数。

我们可以覆盖该函数来执行对视图对象做一些要求文档信息的初始化工作。例如，如果你的应用有固定大小的文档，你可以使用该函数来根据文档的大小初始化视图的滚动范围。如果你的程序支持可变大小的文档，每当文档变化时可以用函数 OnUpdate 来更新滚动范围。

当应用程序被启动时，应用框架会调用 OnCreate 函数，然后立刻调用 OnInitialUpdate 函数。在 OnCreate 函数中也可以完成一部分初始化工作。但是，OnCreate 函数只能被调用一次，而 OnInitialUpdate 函数可以被调用很多次。这就是两者的区别。

在本程序中，我们使用缺省的基类函数就够了。

到现在为止，多文本编辑器已经编写完成了。我们可以同时打开多个文本进行编辑，也可以对一个文本打开多个视图窗口。试着运行这个程序，并测试我们叙述的各项功能。虽然它的功能还不多，但是已经包括了文档/视图结构提供的各项基本的功能了。

7.5 本章小结

本章论述的重点是 MFC 的一种重要的编程模式——文档/视图结构。

文档/视图结构的实质就是将数据从数据的显示和用户对数据的操作中分离出来。在这种模式下，一个文档对象读和写数据并保存数据。文档同时为现存的数据提供一个访问的界面给视图。而单独的视图对象负责数据的显示，将数据放入一个窗口供用户选择和操作。视图对象从文档中获取显示数据，并将任何数据的变化通知文档。

采用文档/视图结构使我们能够很容易支持多个视图窗、多个文档类型、分离窗口以及许多其它有用的用户界面特性。它的核心内容是四个类：

- CDocument：文档类
- CView ：视图类
- CFrameWnd ：框架窗口类
- CDocTemplate：文档模板类

我们分别介绍了四个类的定义和功能。文档类用于存储数据，视图类用于显示数据和接收用户输入，框架

窗口类用于包含窗口和提供必要的用户界面（如菜单，工具条和状态条），文档模板类则将定义了文档类、视图类和框架窗口之间的联系，并且直接创建框架窗口和视图窗口。

我们提供了两个例子。一个为单文档应用，一个为多文档的应用。两者的最大不同就是前者在程序运行期间只允许打开一个文档对象和一个视图窗对象，而后者可以打开多个文档对象和视图窗对象。

第一个例子为单文档的编辑器。我们在其中介绍了应用框架在执行应用程序时的大致流程，剖析了由 AppWizard 自动生成的单文档程序中对上述四个类是如何使用的。然后介绍了键盘消息的处理。我们向视图中加入了数据成员，然后用视图类的成员函数 GetDocument 的到了指向文档的指针，以便于对文档进行操作。

第二个例子为多文档的编辑器。我们主要处理了文档的所有视图显示不一致的问题。这可以通过处理视图类函数 OnUpdate 和文档类函数 UpdateAllViews 来解决。重新编写了 OnDraw 函数来处理视图的重画。

两个例子的主要程序清单都被给出，用户如果有不懂的地方可以参见清单。

第八章　改进应用程序界面

在“消息与命令”一章中，初步介绍了在 MFC 应用程序中如何使用菜单和工具栏进行工作。实际上，有关改进程序界面方面的知识还有很多。本章中详细介绍如何改进应用程序的界面，使之具有更好的用户交互性。

8.1　控制条类

在 Windows 应用程序中，工具条、状态条或对话框条等是经常可见的，为用户提供了快速选择程序功能的能力。在 MFC 基本类库中，工具条、状态条和对话框条都是控制条的子类。MFC 对控制条提供了很好的支持，在下面的几节中，我们将比较详细地介绍控制条的有关知识。

8.1.1　控制条

控制条类（CControlBar）是工具条类（CToolBar）、状态条类（CStatusBar）和对话框条类（CDialogBar），以及 Visual C++ 6.0 中新增的集合条类（CReBar）的父类，同时，CControlBar 类由 CWnd 类派生，图 8-1 显示了有关 CControlBar 的继承关系。

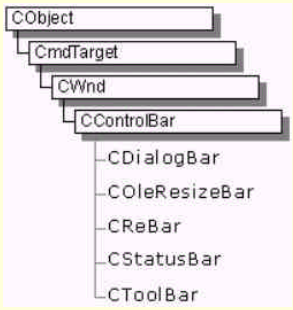

图 8-1　CControlBar 类的继承关系

由于 CControlBar 类是 CWnd 类的子类，因此，控制条通常都是一个停泊在主框架窗口四周的小窗口。控制条可以包含基于 hWnd 句柄的控件，这些控件都是 Windows 窗口，可以产生和响应 Windows 消息；控制条也可以包含非基于 hWnd 句柄的项，这些项通常有应用程序类或主框架窗口类来管理。比如说，列表框和文本框就是基于 hWnd 句柄的控件，而状态条面板和图象按钮则是非基于 hWnd 句柄的项。

CControlBar 类为其子类提供了基本的功能，如在父窗口中的合适位置停泊等。由于控制条通常都是其父框架窗口的一个子窗口，因此，它是视图或 MDI 子窗口的“同胞”。控制条对象使用其父框架窗口的客户区域的信息来定位其自身，因此，其父框架窗口的客户区域将随着控制条的停泊位置的改变而改变。

8.1.2　工具条

工具条是经常使用的一种控制条，通常包含一系列的位图按钮，点击按钮等价与从菜单中选择相应的命令。工具条通常都停泊在框架窗口的顶部，但实际上，MFC 的工具条可以停泊在框架窗口的任何靠边的位置，也可以浮动于自己的小框架窗口中，甚至可以浮动于应用程序的窗口中，可以改变其大小，可以用鼠标拖动到任何位置。图 8-2 和图 8-3 分别显示了工具条停泊在窗口顶部和以浮动窗口显示时的情形。

图 8-2 工具条停泊在窗口顶部

图 8-3 工具条以浮动窗口显示

用户通过 CToolBar 类管理自己的工具条，但是从 MFC4.0 开始，CToolBar 类被重新定义为使用工具条控件（CToolBarCtrl 类）来建立。工具条控件被 Windows 95 和 Windows NT 3.51 或其更高的版本所支持。由于利用了操作系统级的支持，这一重新定义使得 MFC 可以用较少的代码来管理工具条，同时也增强了工具条的功能。用户可以 CToolBar 类的成员函数来操作工具条，或者通过获得一个 CToolBarCtrl 的指针而直接对工具条进行操作。

8.1.3 状态条

状态条是应用程序中经常使用的另外一种控制条，通常包含多个面板，用做文本输出或指示器。状态条一般都停泊在框架窗口的底部，而且也无须改变其位置。图 8-4 是利用 AppWizard 生成的应用程序中的缺省的状态条。

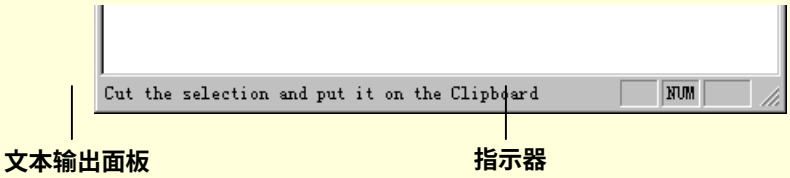

图 8-4 状态条停泊在窗口的底部

与工具条类似，状态条（CStatusBar）也被重新定义为使用状态条控件（CStatusBarCtrl）来建立。

8.1.4 对话框条

对话框条是一种基于对话框资源的控制条，其作用类似于一个无模式对话框，在对话框条中可以包含任何 Windows 控件，甚至可以设置这些控件的制表顺序。图 8-5 显示了一个典型的对话框条，这就是 Visual C++集成开发环境的 WizardBar。

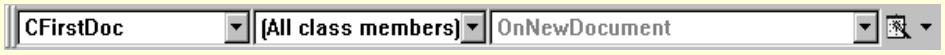

图 8-5 典型的对话框条

8.1.5 集合条

集合条是 Visual C++ 6.0 中新增的，也是一种控制条。集合条中可以包含多种子窗口类型，通常都是一些控件，如按钮、文本编辑框等。集合条能将其子窗口在特定的位图背景上显示出来。使用集合条可以创建漂亮的程序界面。图 8-6 中显示了一个典型的集合条，这就是 Internet Explorer 4.0 的界面。

图 8-6 典型的集合条

集合条（CReBar）也使用了集合条控件（CReBarCtrl）来建立。

在使用控制条的时候，很少直接使用 CControlBar 类或直接从 CControlBar 类派生子类，更多的是使用其子类或从其子类派生自己的类。

以上简要介绍了控制条类极其子类的一些基本的知识，从下一节开始具体介绍如何使用这些控制条。对于

对话框条，我们仅介绍其创建方法，其使用方法可以参照无模式对话框；重点介绍的是工具条、状态条和集合条。

8.2 工具条和状态条

在“消息与命令”一章中，介绍了如何初步地使用工具条，仅仅介绍了如何修改工具条资源，如何将工具条按钮与菜单命令联系起来，如何改变工具条按钮的状态等等，而且使用的是 AppWizard 为用户创建的缺省的工具条。在本节中，将详细解释应用程序框架是如何建立工具条的，如何创建新的定制的工具条，如何控制工具条的显示与隐藏等知识，以及如何使用状态条，如何向状态条中添加新的面板，如何控制状态条的显示等。

8.2.1 缺省的工具条与状态条

在利用 AppWizard 生成应用程序的时候，只要在步骤 4 选中了 Docking toolbar 和 Initial status bar 选项，AppWizard 为用户生成的应用程序就拥有一个缺省的工具条和状态条。

1．建立新的项目

为了便于说明，需要建立一个例子程序 TestBars。利用 AppWizard 生成该应用程序框架。在创建过程中，按以下步骤操作：

（1）选择单文档界面，选中文档/视图结构支持，同时选择资源类型为英语。

（2）由于本程序不需要支持任何数据库技术，选择 None。

（3）由于本程序不需要支持任何形式的复合文档，选择 None；同时本程序也不需要包含 ActiveX 控件，故清除缺省对 ActiveX controls 的选中。

（4）本程序不涉及有关打印的知识，故清除 Print and print preview 选项；但需要缺省的工具条和状态条，故选中 Docking toolbar 和 Initial status bar 选项；工具条风格选择 Normal；其余接受缺省选择即可。

（5）接受缺省选择。

（6）单击 Finish 按钮完成设置，在新项目信息对话框中列出了本 TestBars 程序的有关已设定的信息，如图 8-7 所示。

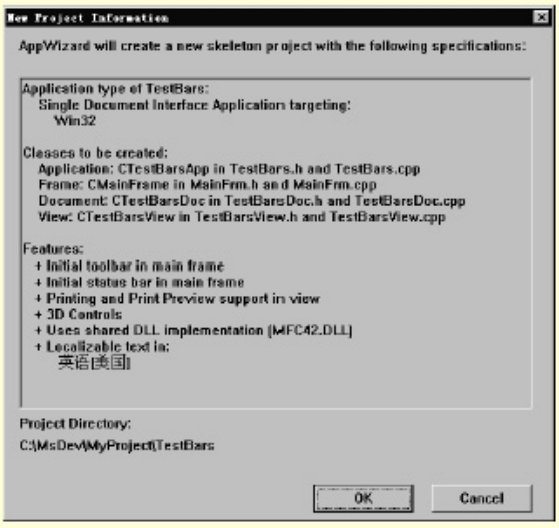

图 8-7 新项目信息

建立项目后，用户可以立即编译和运行 TestBars 应用程序，现在的程序版本就已经具有了缺省的工具条和状态条，如图 8-8 所示。

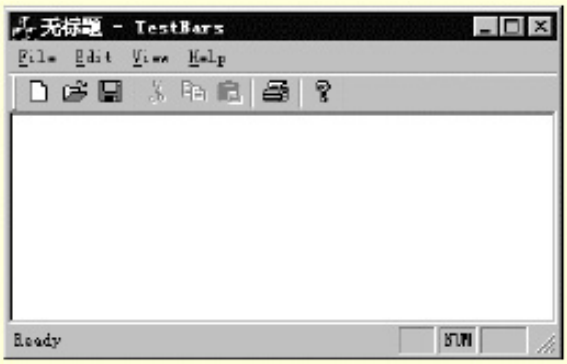

图 8-8 TestBars 程序已经拥有缺省的工具条和状态条

在“消息与命令”一章中讲述工具条的基本使用时提到，首先工具条有一个指定其按钮形式的资源，该资源实际上是一幅位图。但工具条是如何由资源创建为工具条对象并显示在窗口的顶部的呢？下面就来解释工具条的创建过程。

2. 工具条的建立

每个工具条都对应着一个工具条对象，一般说来，工具条都隶属于其父窗口，这里就是主框架窗口，因此，工具条对象应该是主框架窗口的成员。状态条也是如此。在主框架窗口类的定义中，确实可以发现对工具条对象和状态条对象的定义，如程序清单 8-1 所示。

程序清单 8-1 CMainFrame 类中定义了工具条对象和状态条对象

```
class CMainFrame : public CFrameWnd
{
……
protected:  // control bar embedded members
CStatusBar  m_wndStatusBar;
CToolBar    m_wndToolBar;
……
};
```

在 CMainFrame 类中，定义了一个 CStatusBar 类对象 m_wndStatusBar，一个 CToolBar 类对象 m_wndToolBar。仅仅定义了工具条对象是不够的，还需要初始化该对象。由于工具条是主框架窗口的内嵌对象，因此 MFC 在创建主框架窗口的时候就创建了工具条对象，同样地，在销毁主框架窗口的时候就销毁了工具条。状态条的情况是类似的。

工具条的创建在 CMainFrame 类的 OnCreate()函数中进行。程序清单 8-2 中列出了 OnCreate()函数的代码。

程序清单 8-2 OnCreate()函数

```
 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP|CBRS_GRIPPER|
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||  !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
    TRACE0("Failed to create toolbar\n");
    return -1;      // fail to create
}
```

```
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1;          // fail to create
}

// TODO: Delete these three lines if you don't want the toolbar to
//   be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);

return 0;
}
```

创建工具条对象的代码作为一个 if **语句的条件，以检查创建过程是否正确进行。如果不进行正确性检测的话，可以将创建过程分解为调用两个函数，如下所示。**

```
m_wndToolBar.CreateEx(this,TBSTYLE_FLAT,WS_CHILD|WS_VISIBLE | CBRS_TOP| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ；
m_wndToolBar.LoadToolBar(IDR_MAINFRAME)；
```

首先调用 CToolBar **的** CreateEx()**函数创建一个** Windows **工具条并将其与** m_wndToolBar **对象关联在一起。**CreateEx()**函数的原型如下：**

```
BOOL CreateEx(CWnd* pParentWnd, DWORD dwCtrlStyle = TBSTYLE_FLAT, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP, CRect rcBorders = CRect(0, 0, 0, 0), UINT nID = AFX_IDW_TOOLBAR);
```

注意 CreateEx()**函数的参数。**pParentWnd **是一个指向父窗口的指针，这里使用了关键字** this**，即将当前对象，也就是主框架窗口作为本工具条的父窗口。**

参数 dwCtrlStyle **指定在建立内嵌** CToolBarCtrl **对象时所用的风格。缺省的风格是** TBSTYLE_FLAT**。**

参数 dwStyle **指定工具条的风格。**dwCtrlStyle **参数与** dwStyle **参数可取的值比较多，而且相互关联，因此本书中只对使用到的风格略作说明，对于其他可选风格的详细信息，用户在使用时请查阅** Visual C++**的联机手册。**

参数 rcBorders **指定工具条窗口边框的宽度。缺省值为** CRect(0,0,0,0)**，即工具条窗口没有边框。**

参数 nID **指定工具条的标识符。**

AppWizard **创建的缺省工具条使用的风格的说明列于表** 8-1 **中。**

表 8-1　**缺省工具条使用到的风格**

所用风格	说　明
TBSTYLE_FLAT	创建一个平面工具条
WS_CHILD	创建一个子窗口
WS_VISIBLE	创建的工具条初始即可见
CBRS_TOP	工具条位于父窗口的顶部
CBRS_GRIPPER	工具条具有控制棒
CBRS_TOOLTIPS	工具条按钮具有提示信息（鼠标经过时显示在一个小窗口中）
CBRS_FLYBY	工具条按钮具有提示信息（显示在状态条的左部）
CBRS_SIZE_DYNAMIC	工具条的大小动态可调

除了 CreateEx()**函数之外，**CToolBar **类还有另外的一个函数** Create()**也可以创建工具条对象，其原型如下：**

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP, UINT nID = AFX_IDW_TOOLBAR );
```

其参数的意义大致同 CreateEx()函数。

在创建了工具条对象后，就要将其与相应的工具条资源联系起来，这项工作由 CToolBar 类的 LoadToobar()函数完成。LoadToobar()函数的原型如下：

```
BOOL LoadToolBar( UINT nIDResource );
```

参数 nIDResource 指定所用资源的标识符。

在显示工具条之前还有一项工作，就是确定工具条的停泊位置。与此有关的代码是：

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
```

确定工具条的停泊位置需要三个步骤：

（1）确定工具条可以停泊的位置，也就是说，确定工具条本身希望停泊在主框架窗口的位置。至于能否停泊在指定的位置，还需要主框架窗口的支持。由 CToolBar 类的 EnableDocking()函数进行设置，这里使用 CBRS_ALIGN_ANY 作为参数说明工具条期望停泊的位置可以是窗口的任一边。

（2）确定主框架窗口中可以停泊工具条的位置。由 CWnd 类的 EnableDocking()函数指定，这里使用 CBRS_ALIGN_ANY 作为参数说明主框架窗口允许工具条停泊在其窗口中的任一边。

（3）停泊工具条在指定位置，由 CWnd 类的 DockControlBar()函数完成。

至此已经完成了工具条的对象创建、资源关联和位置确定等所有的准备工作，这样在程序显示主框架窗口的时候，工具条也就被显示出来。

3．状态条的建立

状态条的建立过程与工具条是类似的。在缺省的情况下，状态条的建立由以下代码完成：

```
m_wndStatusBar.Create(this);
m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT));
```

首先是建立状态条对象。CStatus 类的 Create()函数的原型如下：

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, UINT nID = AFX_IDW_STATUS_BAR );
```

其参数的意义与 CToolBar 类的 Create()函数类似。需要注意的是建立状态条对象后如何设置状态条指示器，这由 CStatusBar 类的 SetIndicators()函数完成，该函数的原型如下：

```
BOOL SetIndicators( const UINT* lpIDArray, int nIDCount );
```

参数 lpIDArray 是一个指向指示器标识符数组的指针，参数 nIDCount 是数组元素的数目。在缺省状态下，AppWizard 建立的标识符数组具有四个元素，在 CMainFrame 类的源文件的头部可以找到如下定义：

```
static UINT indicators[] =
{
ID_SEPARATOR,           // status line indicator
ID_INDICATOR_CAPS,
ID_INDICATOR_NUM,
ID_INDICATOR_SCRL,
};
```

数组 indicators 包含的四个元素都是预先定义好的标识符。

由于状态条一般都停泊在主框架窗口的底部，因此对状态条而言，没有确定停泊位置的问题。

现在用户应该清楚，缺省的工具条和状态条是如何被创建出来的。在下面的部分，将通过建立自己的工具条与向状态条中添加指示器，更深入地介绍工具条与状态条的操作。

8.2.2 创建自己的工具条

在比较大的 Windows 应用程序中，往往都使用了多个工具条，因此，在这样的情况下，仅仅使用缺省的工具条是不够的。其实，创建自己的工具条也不是一项困难的工作，只要用户已经真正理解缺省工具条的创建过

程，实现自己的工具条就是很自然的过程。

我们计划在 TestBars 程序中加入一个新的绘图工具条，包含一些基本的绘图功能。

1．创建工具条资源

首要的任务是创建一个工具条资源。在开发环境的 Insert 菜单中选择 Resource 命令，弹出 Insert Resource 对话框，如图 8-9 所示。

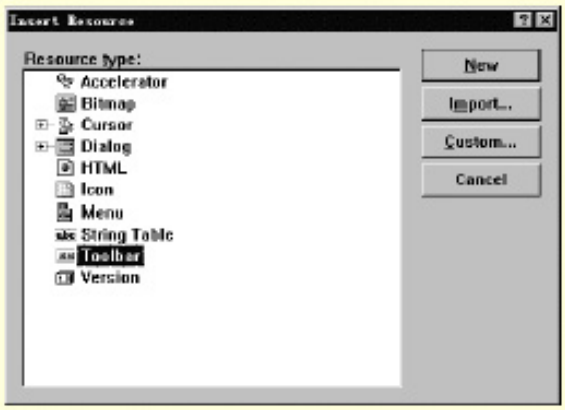

图 8-9 Insert Resource 对话框

在对话框中选择 Toolbar 类型，并单击 New 按钮建立新的工具条资源。使用工具条编辑器建立工具条按钮，完成后的工具条如图 8-10 所示。

图 8-10 完成的新工具条

打开工具条的属性对话框，修改工具条的标识符为 IDR_DRAWTOOLS，如图 8-11 所示。

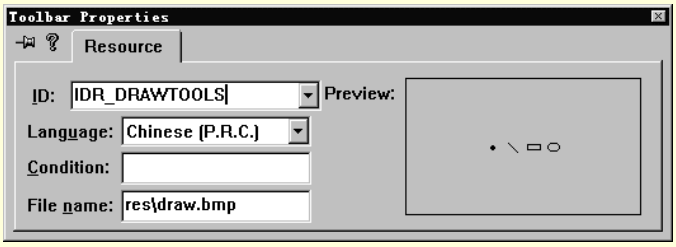

图 8-11 工具条属性对话框

为建立的新按钮指定标识符及提示信息，各按钮的标识符列于表 8-2 中。

表 8-2 按钮标识符

按 钮	标识符	提示信息
点	ID_DRAW_POINT	Draw a point\nPoint
直线	ID_DRAW_LINE	Draw a line\nLine
矩形	ID_DRAW_RECTANGLE	Draw a rectangle\nRectangle
椭圆	ID_DRAW_ELLIPSE	Draw a ellipse\nEllipse

至此已经完成了新工具条的资源创建工作。下面的任务是添加代码以完成工具条的创建工作。

2．创建工具条

为了建立一个新的工具条对象，需要向主框架窗口添加一个新的 CToolBar 类的成员 m_wndDrawTools。可以直接在 MainFrame.h 中进行修改，定位到 AppWizard 定义缺省工具条和状态条的位置，修改为：

```
protected:    // control bar embedded members
    CStatusBar    m_wndStatusBar;
    CToolBar      m_wndToolBar;
    CToolBar m_wndDrawTools;
```

即添加了新的 CToolBar 类的成员 m_wndDrawTools。在主框架窗口的 OnCreate()函数中需要完成常见对象

和关联资源以及工具条的泊定工作。我们几乎可以完全照搬缺省工具条的做法。程序清单 8-3 中列出修改完成后的 OnCreate()函数的代码。

程序清单 8-3 CMainFrame::OnCreate()

```
 int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
}
//Create drawtools toolbar
m_wndDrawTools.Create(this);
m_wndDrawTools.LoadToolBar(IDR_DRAWTOOLS);

if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
{
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
}

// TODO: Delete these three lines if you don't want the toolbar to
//    be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);

//Docking drawtools toolbar
m_wndDrawTools.EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndDrawTools);

return 0;
}
```

为了比较旧式工具条与新的平面式工具条的不同效果，我们使用了 Create()函数来建立新的工具条。其余的工作与缺省工具条的做法是类似的。

现在请用户编译和运行 TestBars 程序，观察程序的运行结果，如图 8-12 所示。

图 8-12 运行中的 TestBars 程序

注意比较缺省工具条和新建工具条的外观，很容易就可以看出，缺省工具条由于使用了 TBSTYLE_FLAT 风格，工具条上的按钮在不拥有鼠标焦点时是平的，只有鼠标指向按钮的时候才稍稍突起，而新建的工具条上的按钮一直处于突起状态，这是因为新建工具条采用的是旧式的工具条风格。

由于我们没有响应按钮的任何动作，因此按钮呈灰色无效状态，但这已足以说明新的工具条已经被正确地创建出来了。如果用户愿意，可以在工具条的 Create()函数（或使用 CreateEx()函数）中指定不同的风格，观察工具条的外观效果。

3. 响应按钮动作

现在我们的目标是让新工具条的按钮能确实地动作起来，而不是仅仅停留在无效状态。当然，按照“绘图与文本”一章中的做法，可以为每个用户界面对象（这里是工具条按钮）设置一个标志变量，然后对每个用户界面对象的命令消息和更新显示消息都加上消息处理函数。在本章，我们将采用另外一种更简单的办法。

考虑到对 DrawTools 工具条上的按钮，每一时刻只可能有一个按钮处于活动状态，即程序功能由该按钮确定，是画点，画线，还是画矩形等等，而且，也只有这一个按钮处于选中状态，表现为被按下。另外，从操作来看，这些按钮的作用是类似的，处理这些按钮的消息的方法也是类似的，甚至更新这些按钮的显示状态的方法也是类似的。由此可以想到，能否将这些按钮的命令消息都集中在一个响应函数中进行处理。很幸运，答案是能。

一种可能的办法是将所有按钮产生的消息都用 ON_COMMAND 宏映射到同一函数中，每个按钮命令有一个消息映射入口。具体到 TestBars 程序来说，如果假定由 OnDrawTools()函数来处理这些按钮的命令，那么可以用这样的方法：在使用 ClassWizard 进行消息映射的时候，将 ID_DRAW_XXX 的 COMMAND 消息都映射到同一函数 OnDrawTools()上去，或者直接在相应的源文件中修改。这里我们选择了主框架窗口接受这些按钮的消息，因此在修改之后，MainFrame.cpp 中的消息映射入口应该像程序清单 8-4 中所示的那样。

程序清单 8-4 多个 ON_COMMAND 宏映射到同一处理函数

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
ON_COMMAND(ID_DRAW_ELLIPSE, OnDrawTools)
ON_COMMAND(ID_DRAW_POINT, OnDrawTools)
ON_COMMAND(ID_DRAW_LINE, OnDrawTools)
ON_COMMAND(ID_DRAW_RECTANGLE, OnDrawTools)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

然后就可以在 OnDrawTools()函数中针对不同的按钮选择进行不同的处理，这样就避免了产生一大串的消息处理函数。

另外一种更好的办法就是使用 ON_COMMAND_RANGE 宏来进行消息映射。ON_COMMAND_RANGE 宏

用来将一定范围内的命令映射到一个消息处理函数中，该宏被定义为：

ON_COMMAND_RANGE(id1, id2, memberFxn)

其中参数 id1 指定了范围内编号最小的标识符，id2 指定了范围内编号最大的标识符，memberFxn 指定了用来处理消息的函数。使用 ON_COMMAND_RANGE 宏将编号界于 id1 与 id2 之间的所有标识符对应的用户界面对象的命令消息都送至 memberFxn 函数处理。

对 TestBars 程序而言，图 8-13 显示了按钮的标识符及对应的编号。

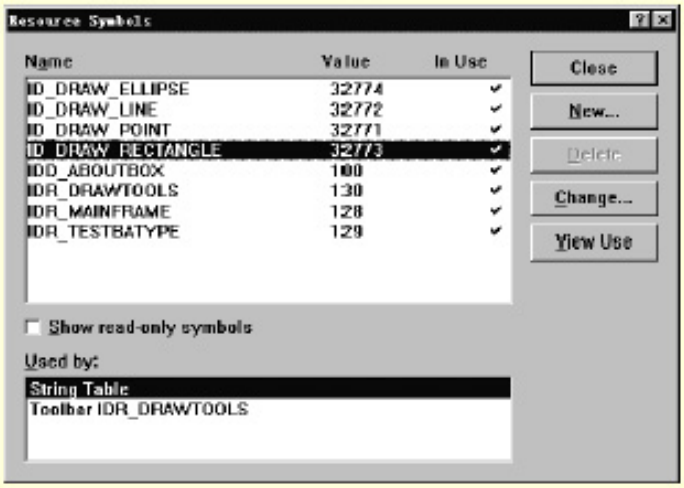

图 8-13 TestBars 程序所用到的标识符

可以看到，在几个按钮中，ID_DRAW_POINT 标识符对应的编号最小为 35771，ID_DRAW_ELLIPSE 标识符对应的编号最大为 32774。因此，如果按以下方式使用 ON_COMMAND_RANGE 宏：

ON_COMMAND-RANGE(ID_DRAW_POINT,ID_DRAW_ELLIPSE,OnDrawTools)

这意味着从 ID_DRAW_POINT 到 ID_DRAW_ELLIPSE 的命令消息都送至 OnDrawTools()函数进行处理。

让人觉得有些遗憾的是，在 ClassWizard 中进行消息映射只能是产生 ON_COMMAND 宏，因此我们必须手工添加 ON_COMMAND_RANGE 宏。

定位到 MainFrame.cpp 文件中的消息映射部分，添加这样一行：

ON_COMMAND-RANGE(ID_DRAW_POINT,ID_DRAW_ELLIPSE,OnDrawTools)

注意一定要将 ON_COMMAND_RANGE 宏添加在//{{AFX_MSG_MAP 和//}}AFX_MSG_MAP 注释对的外面，因为该对注释里面的内容是由 ClassWizard 生成并管理的，我们不能破坏 ClassWizard 的正常工作。添加完成后的代码应该如程序清单 8-5 所示。

程序清单 8-5 手工添加 ON_COMMAND_RANGE 宏

```
 BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
//}}AFX_MSG_MAP
ON_COMMAND_RANGE(ID_DRAW_POINT,ID_DRAW_ELLIPSE,OnDrawTools)
END_MESSAGE_MAP()
```

由于同样的原因，ON_COMMAND_RANGE 宏的消息处理函数的声明与定义也必须手工添加。ON_COMMAND_RANGE 宏的消息处理函数可以有一个参数 nID，该参数为 UINT 型，指定了当前产生命令的标识符。因此对 TestBars 程序而言，添加 OnDrawTools()函数的对话框应该如图 8-14 所示。

图 8-14 添加 OnDrawTools()函数

以上介绍了 ON_COMMAND_RANGE 宏，对于用户界面对象的更新显示的消息，MFC 同样提供了 ON_UPDATE_COMMAND_UI_RANGE 宏供用户使用。该宏的使用方法是类似的，其定义如下：

```
ON_UPDATE_COMMAND_UI_RANGE( id1, id2, memberFxn )
```

其参数的意义同 ON_COMMAND_RANGE 宏。ON_UPDATE_COMMAND_UI_RANGE 宏的消息处理函数可以拥有一个 CCmdUI 类的指针，该指针代表了需要显示更新的用户界面对象。

ON_UPDATE_COMMAND_UI_RANGE 宏及其消息处理函数也是必须手工添加的。对 TestBars 程序，程序清单 8-6 中是添加了 ON_UPDATE_COMMAND_UI_RANGE 宏的消息映射入口部分。

程序清单 8-6 手工添加 ON_UPDATE_COMMAND_UI_RANGE 宏

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
//}}AFX_MSG_MAP
ON_COMMAND_RANGE(ID_DRAW_POINT,ID_DRAW_ELLIPSE,OnDrawTools)
ON_UPDATE_COMMAND_UI_RANGE(ID_DRAW_POINT,ID_DRAW_ELLIPSE,OnUpdateDrawTools)
END_MESSAGE_MAP()
```

添加 OnUpdateDrawTools()函数的对话框应该类似于图 8-15。

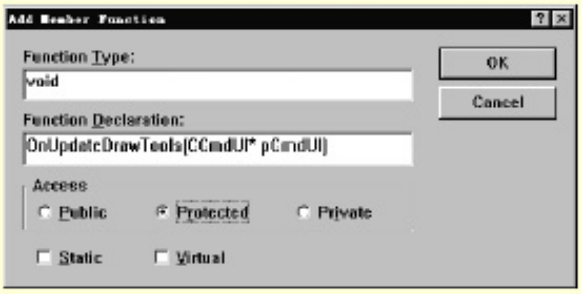

图 8-15 添加 OnUpdateDrawTools()函数

现在已经正确地对一组用户界面对象的消息进行了映射，为了使程序真正地工作起来，还必须添加一些代码。

为了记录当前被选择的按钮，必须向主框架窗口类添加一个成员变量。如图 8-16 所示。

图 8-16 添加 m_nCurID 成员变量

在 CMainFrame 类的构造函数中，必须初始化该成员变量；在 OnDrawTools()函数中，必须根据当前选择的按钮改变 m_nCurID 的值；在 OnUpdateDrawTools()函数中，必须及时更新各按钮的显示。程序清单 8-7 中列出了完成后的代码。

程序清单 8-7 CMainFrame()、OnDrawTools()和 OnUpdateDrawTools()

```
CMainFrame::CMainFrame()
{
// TODO: add member initialization code here
m_nCurID=ID_DRAW_POINT;
}
void CMainFrame::OnUpdateDrawTools(CCmdUI *pCmdUI)
{
pCmdUI->SetCheck(m_nCurID==pCmdUI->m_nID);
}
void CMainFrame::OnDrawTools(UINT nID)
{
m_nCurID=nID;
}
```

这些函数中的代码都很简单，用户完全可以自己分析。

现在的 TestBars 程序的工具条部分已经基本完成了。用户可以编译和运行现在的版本，运行的结果如图 8-17 所示。

由于本章的目的并不在于讲述如何进行绘图工作，因此到现在为止已经清楚地展示了创建一个新的工具条的过程。如果用户有兴趣，可以自己添加代码响应鼠标消息而完成绘图功能。

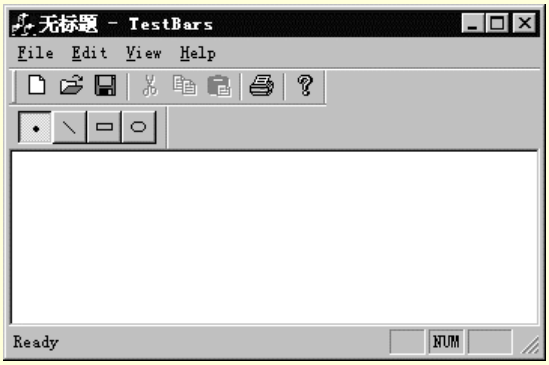

图 8-17 运行中的 TestBars 程序

4. 显示/隐藏工具条

对于工具条而言，最后一项尚未完成的工作就是控制工具条的显示和隐藏了。因为工具条本身实际是一个小窗口，因此，使用 CWnd 类的 ShowWindow()函数是有效的，该函数的原型如下：

```
BOOL ShowWindow( int nCmdShow );
```

该函数根据参数 nCmdShow 的不同取值设置窗口的显示状态。表 8-3 中列出了参数 nCmdShow 可能的取值:

表 8-3 nCmdShow 参数

取 值	说 明
SW_HIDE	隐藏窗口并激活其他窗口
SW_MINIMIZE	最小化窗口
SW_RESTORE	激活并显示窗口至原来的位置和大小
SW_SHOW	激活并显示窗口
SW_SHOWMAXIMIZED	激活窗口并以最大化方式显示
SW_SHOWMINIMIZED	激活窗口并以最小化方式显示
SW_SHOWMINNOACTIVE	将窗口作为图标显示
SW_SHOWNA	按当前状态显示窗口
SW_SHOWNOACTIVATE	按最近一次使用时的位置和大小显示窗口
SW_SHOWNORMAL	激活和显示窗口

在控制工具条时，只需要简单地控制工具条是显示还是隐藏。在上述参数的取值中还隐藏着一条规则：如果参数 nCmdShow 取值为 0，则隐藏窗口；如果参数 nCmdShow 取值非 0，则显示窗口。在 TestBars 程序中，使用这一规则来控制工具条的显示与隐藏是很方便的。

一般说来，都在应用程序的 View 菜单中控制工具条、状态条等的状态。为此我们需要向 View 菜单中添加一条命令项，完成后的菜单如图 8-18 所示。

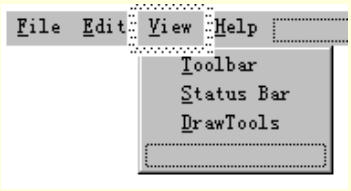

图 8-18 在 View 菜单中添加 DrawTools 项

新添加的菜单项的属性如图 8-19 所示。

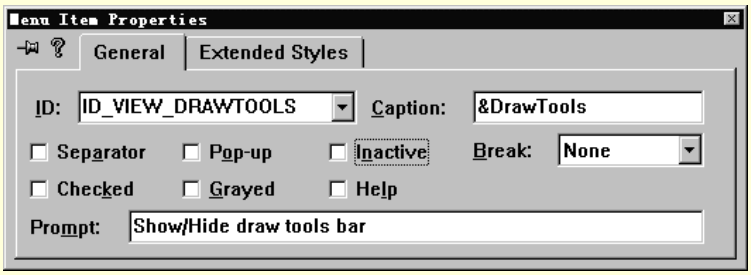

图 8-19 菜单项 DrawTools 的属性

使用 ClassWizard 为 ID_VIEW_DRAWTOOLS 消息建立处理函数，对 COMMAND 消息，处理函数为 OnViewDrawtools()；对 UPDATE_COMMAND_UI 消息，处理函数为 OnUpdateViewDrawtools()。

在 OnViewDrawtools()函数中，完成工具条的显示与隐藏工作，程序清单 8-8 列出了完成后的 OnViewDrawtools()函数。

程序清单 8-8 OnViewDrawtools()

```
void CMainFrame::OnViewDrawtools()
{
// TODO: Add your command handler code here
m_wndDrawTools.ShowWindow((m_wndDrawTools.GetStyle()&WS_VISIBLE)==0);
```

```
RecalcLayout();
}
```

在 OnViewDrawtools()函数中，使用了 ShowWindow()函数控制工具条的显示与隐藏。需要注意的是 ShowWindow()函数的参数：

(m_wndDrawTools.GetStyle()&WS_VISIBLE)==0

首先由 CToolBar 类的成员函数 GetStyle()获得工具条当前的状态，然后使用&（与）操作符检验工具条当前状态是否具有 WS_VISIBLE 风格：如果具有 WS_VISIBLE 风格，则&操作的结果非 0，整个参数为 false，因此将工具条设置为不可见；如果不具有 WS_VISIBLE 风格，则&操作的结果为 0，整个参数为 true，因此将工具条设置为可见。这样就完成了工具条的显示与隐藏工作。

RecalcLayout()函数用于在主框架窗口被改变了大小或显示/隐藏了工具条等情况下调用，该函数重新计算框架窗口中各部分的尺寸。

在 OnUpdateViewDrawtools()函数，完成在菜单项前添加或清除核对标记的工作，程序清单 8-9 中列出了该函数的代码。

程序清单 8-9　OnUpdateViewDrawtools()

```
void CMainFrame::OnUpdateViewDrawtools(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
int nCheck=m_wndDrawTools.GetStyle() & WS_VISIBLE;
pCmdUI->SetCheck(nCheck);
}
```

用户可以自行分析该函数中函数的调用与参数的传递过程。

到此为止我们已经彻底完成了 TestBars 的工具条部分的工作，从分析应用成许缺省的工具条的建立开始，逐步演示了创建自己的工具条并使之工作，控制工具条的显示/隐藏等过程。用户可以编译和运行现在的程序的版本，观察工具条的操作。

5．MFC 中的消息映射宏

在响应工具条按钮动作的函数中，我们使用了 ON_COMMAND_RANGE 宏和 ON_UPDATE_COMMAND_UI_RANGE 宏以达到减少代码的目的。下面我们对 MFC 基础类库中有关消息映射的宏略做介绍。

迄今为止，介绍过的用于消息映射的宏共有七个：DECLARE_MESSAGE_MAP 宏、BEGIN_MESSAGE_MAP 宏、END_MESSAGE_MAP 宏、ON_COMMAND 宏、ON_UPDATE _COMMAND_UI 宏、ON_UPDATE_COMMAND_UI_RANGE 宏以及 ON_COMMAND _RANGE 宏。实际上，MFC 提供的用于消息映射的宏共有十几个，大致可以分为以下几类：

（1）消息映射定义和划分宏：

- DECLARE_MESSAGE_MAP：使用在类的头文件中，声明该类拥有消息映射。
- BEGIN_MESSAGE_MAP：与 END_MESSAGE_MAP 配对使用，用于类的源文件中，开始该类的消息映射。
- END_MESSAGE_MAP：用于类的源文件中，结束该类的消息映射。

（2）消息映射宏（用于处理单个消息）：

- ON_COMMAND：指定用来处理命令消息的函数。
- ON_CONTROL：指定用来处理特定的控件通知消息的函数。
- ON_MESSAGE：指定用来处理用户自定义消息的函数。
- ON_OLECMD：指定用来处理 OLE 命令消息的函数。
- ON_REGISTERED_MESSAGE：指定用来处理注册过的用户自定义消息的函数。
- ON_REGISTERED_THREAD_MESSAGE：如果拥有一个 CWinThread 类，指定用来处理注册过的用户自定义消息的函数。
- ON_THREAD_MESSAGE：如果拥有一个 CWinThread 类，指定用来处理用户自定义消息的类。

- ON_UPDATE_COMMAND_UI：指定用来处理特定的用户界面更新命令消息。

（3）消息映射宏（用于处理一组消息）：

- ON_COMMAND_RANGE：指定用来处理特定的一组命令消息的函数。
- ON_UPDATE_COMMAND_UI_RANGE：指定用来处理特定的一组用户界面更新命令消息。
- ON_CONTROL_RANGE：指定用来处理特定的一组控件通知消息的函数。

这些宏的具体使用场合及详细的使用方法，请用户参阅 Visual C++的联机文档。

8.2.3 向状态条中添加指示器

前面已经介绍了缺省的状态条创建的全过程，但是缺省的状态条往往不能满足用户的特殊要求，需要对缺省的状态条进行修改。在本小节中，将介绍如何向状态条中添加自己的指示器。

我们计划在 TestBars 程序缺省状态条的基础上，添加两个指示器用来指示鼠标的当前位置，一个指示器显示鼠标当前位置的 X 坐标，另一个指示器则显示 Y 坐标。这一功能在许多绘图程序中都是有的，可以方便程序使用者的工作。完成后的程序运行时如图 8-20 所示，注意观察状态条上两个新的指示器。

图 8-20 新的状态条

向状态条中添加指示器和向工具条中添加按钮是不同的。下面介绍的方法是通用的。

1．添加新的指示器

首先是为计划添加的指示器定义命令标识符。选择 View 菜单下的 Resource Symbols 命令，打开 Resource Symbols 对话框。在对话框中单击 New 按钮，建立新的标识符。两个指示器需要两个标识符 ID_COORDINATE_X 和 ID_COORDINATE_Y，在新建标识符的时候，可以接受缺省的编号，或自己指定编号，但自己指定编号则要避免发生不必要的重复。图 8-21 是添加完成后的 Resource Symbols 对话框，最上端的两行就是新建的标识符。

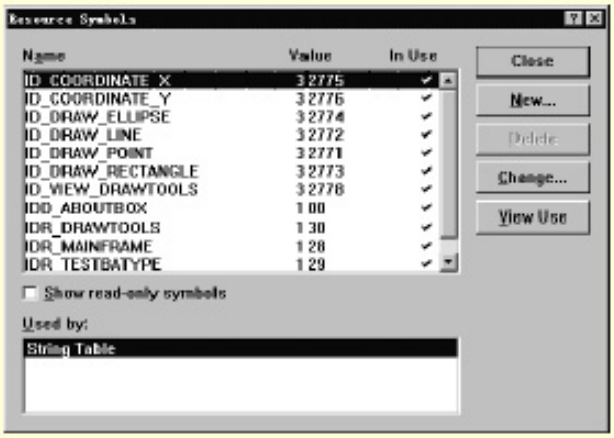

图 8-21 Resource Symbols 对话框

在定义了指示器的标识符后，接下来的工作就是为指示器指定缺省的字符串资源。如果不指定字符串资源，编译的时候将可能会出错。在项目工作台的 ResourceView 中，双击 String Table 下的项，打开字符串资源编辑器。在 Insert 菜单下选择 New String 命令，打开 String Properties 对话框，如图 8-22 所示。

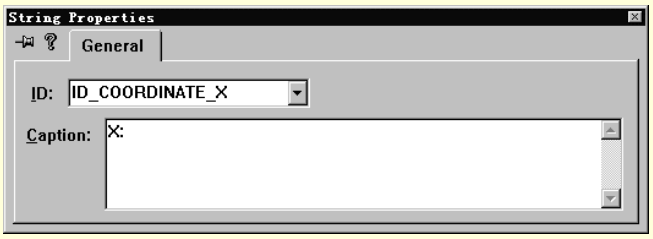

图 8-22 String Properties 对话框

在对话框的 ID 列表中选择刚才新建立的 ID_COORDINATE_X，在 Caption 框中输入缺省的字符串，这里输入“X：”；对 ID_COORDINATE_Y，输入“Y：”。这样就为指示器建立了缺省的字符串。

然后将新的指示器的标识符添加到状态条指示器标识符数组中，该数组存在于 MainFrm.cpp 文件的开始部分，对数组 indicators[]的元素进行手工修改，修改后的 indicators[]数组如下：

```
static UINT indicators[] =
{
ID_SEPARATOR,           // status line indicator
ID_COORDINATE_X,
ID_COORDINATE_Y,
ID_INDICATOR_CAPS,
ID_INDICATOR_NUM,
ID_INDICATOR_SCRL,
};
```

其中的 ID_COORDINATE_X 和 ID_COORDINATE_Y 就是新增加的。

现在用户可以编译和运行 TestBars 程序，运行中的程序如图 8-23 所示。

图 8-23 在状态条中出现了两个新的指示器

从图 8-23 中可以明显地看出，在状态条中出现了两个新的指示器，指示器的外观是凹下的，宽度正好足够显示指示器标识符对应的缺省字符串。

当然，现在的指示器仅仅是显示出来了，要能够指示鼠标的位置，还有许多工作要做。

2．让新的指示器工作起来

比较图 8-23 和图 8-20 中的新的指示器，可以发现两图中指示器的外观效果是不一样的，图 8-23 中的指示器显示为凹下状态，而图 8-20 中的指示器显示为凸起状态。在主框架窗口的 OnCreate()函数中建立的状态条之后，可以对状态条指示器进行设置。对状态条的指示器进行设置使用 CStatusBar 类的 SetPaneInfo()函数，该函数的原型如下：

```
void SetPaneInfo( int nIndex, UINT nID, UINT nStyle, int cxWidth )
```

该函数的参数有四个。nIndex 参数指明进行设置的面板，其取值就代表了在状态条中该面板出现的位置，

最左边的面板的 nIndex 为 0；nID 参数指明为该面板设置的新的标识符；nStyle 指定设置的风格，表 8-4 列出了可以使用的风格；cxWidth 指定面板的宽度。

表 8-4 nStyle 参数

风 格	说 明
SBPS_NOBORDERS	面板周围无 3D 边框
SBPS_POPOUT	面板显示为凸起状态
SBPS_DISABLED	在面板中不显示字符串
SBPS_STRETCH	扩展该面板至所有无用空间
SBPS_NORMAL	面板为普通模式

在 TestBars 程序的状态条中，新增加的两个指示器需要显示鼠标当前位置的坐标，因此必须将指示器的宽度值设置得大一些，同时，需要设置指示器风格为 SBPS_POPOUT。在 CMainFrame 类的 OnCreate()函数的末尾加上下面这两行代码（在 return 语句之前）：

```
m_wndStatusBar.SetPaneInfo(1,ID_COORDINATE_X,SBPS_POPOUT,50);
m_wndStatusBar.SetPaneInfo(2,ID_COORDINATE_Y,SBPS_POPOUT,50);
```

这就完成了指示器的设置工作。下面的任务是添加适当的代码使指示器真正能显示鼠标的当前位置。

显然，为了能够跟踪鼠标的移动，必须不断地更新指示器中显示的文本。当然，更新指示器中显示的文本可以有很多种办法，但推荐的方法是在状态条面板的更新显示消息的处理函数中使用 CCmdUI 类的成员函数 SetText()。

稍微有些遗憾的是，我们无法利用 ClassWizard 为状态条面板的更新消息消息添加处理函数，因此，我们必须手工添加消息映射的入口。

一般说来，鼠标当前位置的坐标都是相对于视图的原点而言的。因此，应该由视图来处理状态条面板的更新显示消息。为此在 CTestBarsView 类的消息映射入口中添加这样的两行：

```
ON_UPDATE_COMMAND_UI(ID_COORDINATE_X,OnUpdateCoorX)
ON_UPDATE_COMMAND_UI(ID_COORDINATE_Y,OnUpdateCoorY)
```

当然，完全可以用 ON_UPDATE_COMMAND_UI_RANGE 宏来进行映射。程序清单 8-10 显示了 TestBarsView.cpp 文件中有关消息映射的部分。

程序清单 8-10 消息映射入口

```
BEGIN_MESSAGE_MAP(CTestBarsView, CView)
//{{AFX_MSG_MAP(CTestBarsView)
ON_WM_MOUSEMOVE()
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
ON_UPDATE_COMMAND_UI(ID_COORDINATE_X,OnUpdateCoorX)
ON_UPDATE_COMMAND_UI(ID_COORDINATE_Y,OnUpdateCoorY)
END_MESSAGE_MAP()
```

最后的两行就是应该手工添加的部分。

我们还需要把相应的处理函数 OnUpdateCoorX()和 OnUpdateCoorY()真正地建立起来。首先要建立这两个函数的框架，利用 Add Member Function 对话框建立框架代码。图 8-24 显示了建立 OnUpdateCoorX()函数时的情形。

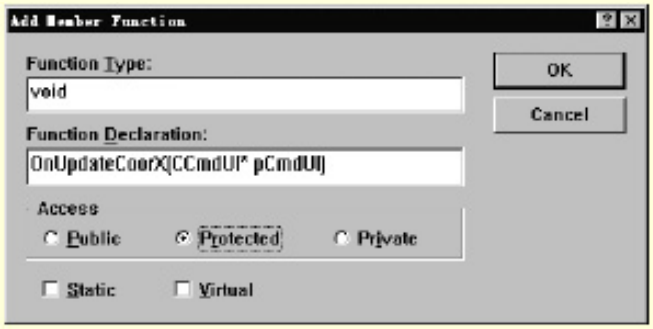

图 8-24 添加 OnUpdateCoorX()函数

添加 OnUpdateCoorY()函数的过程是类似的。

为了记录鼠标的当前位置，需要在 CTestBarsView 类添加一个成员变量 m_CurPos，类型为 CPoint，访问级别为 protected。

在 CTestBarsView 的构造函数中要初始化 m_CurPos。同时，鼠标一旦移动，就需要更新 m_CurPos 的值，使得记录在 m_CurPos 中的值始终代表了鼠标的当前位置。使用 ClassWizard 添加对 WM_MOUSEMOVE 消息的处理函数，并在该函数中对 m_CurPos 的值进行设置。程序清单 8-11 中列出了 CTestBarsView()函数和 OnMouseMove()函数的代码。

程序清单 8-11 CTestBarsView()函数和 OnMouseMove()函数

```
CTestBarsView::CTestBarsView()
{
// TODO: add construction code here
m_CurPos=CPoint(0,0);
}

void CTestBarsView::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
m_CurPos=point;
CView::OnMouseMove(nFlags, point);
}
```

在 OnUpdateCoorX()函数和 OnUpdateCoorY()函数中真正地更新指示器中显示的文本。程序清单 8-12 中列出了完成后的 OnUpdateCoorX()函数和 OnUpdateCoorY()函数的代码。

程序清单 8-12 OnUpdateCoorX()函数和 OnUpdateCoorY()函数

```
void CTestBarsView::OnUpdateCoorX(CCmdUI *pCmdUI)
{
CString s;
s.Format("X:%d",m_CurPos.x);
pCmdUI->SetText(s);
}

void CTestBarsView::OnUpdateCoorY(CCmdUI *pCmdUI)
{
CString s;
s.Format("Y:%d",m_CurPos.y);
```

```
pCmdUI->SetText(s);
}
```

代码其实很简单。由于 m_CurPos 变量中已经保存了鼠标当前位置的坐标，因此只需对将要显示在指示器中的文本稍加格式化即可。

至此我们已经完成了 TestBars 的状态条部分。用户现在编译和运行程序，就可以看到如图 8-20 所示那样的效果了。

本节中介绍了普通的工具条，或者说是老样子的工具条，和状态条的使用方法。实际上，工具条和状态条还有更多的使用技巧，用户可以参阅 Visual C++的联机文档并自己进行实验。在后面的几节中，还将介绍到另外的一些控制条的使用方法。

8.3 对话框条

对话框条通过控制条的方式提供了无模式对话框的功能，它最大的优点就在于它可以包含所有类型的 Windows 控件，而不仅仅是按钮。与工具条不一样的是，对话框条通过对话框资源来建立，用户可以使用对话框编辑器来编辑该对话框资源。

在使用上，对话框条包含的所有控件发生的动作消息都被发送给其父窗口，其行为和操作完全类似与一个无模式对话框。

本节中仅仅介绍如何在 TestBars 程序中创建一个对话框条。

单击 Insert 菜单的 Resource 命令，弹出 Insert Resource 对话框，如图 8-25 所示。

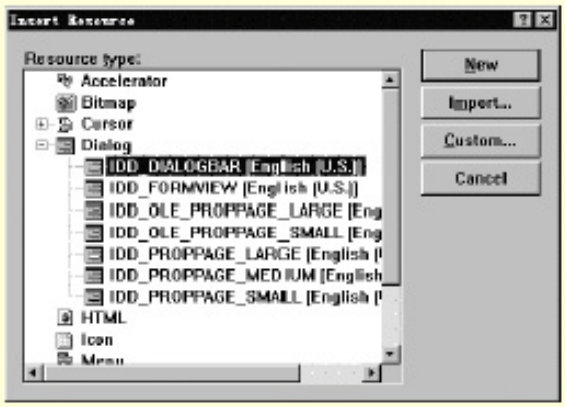

图 8-25 Insert Resource 对话框

在 Insert Resource 对话框中展开 Dialog 项，注意到 Dialog 下的第一个子项的标识符为 IDD_DIALOGBAR，这表明该项代表着缺省的对话框条资源。选中 IDD_DIALOGBAR，单击 New 按钮，则向 TestBars 程序的资源中添加了一个缺省的对话框条，如图 8-26 所示。

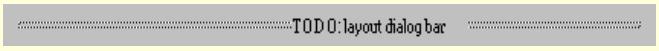

图 8-26 缺省的对话框条

当然，缺省的对话框条是不能满足 TestBars 程序的要求的。修改缺省的对话框条，完成后如图 8-27 所示。

图 8-27 修改后的对话框条

将对话框条的标示号改为 IDD_TEXTTOOLS，在 CMainFrame 类的 OnCreate()函数中将使用该标识符创建对话框条对象。在 OnCreate()函数的末尾添加如下一行语句：

```
m_wndTextTools.Create(this,IDD_TEXTTOOLS,CBRS_TOP|CBRS_GRIPPER,IDD_TEXTTOOLS);
```

CDialogBar 类的 Create 函数的原型如下所示：

BOOL Create(CWnd* pParentWnd, UINT nIDTemplate, UINT nStyle, UINT nID);

参数 pParentWnd 指定对话框条的父窗口；参数 nIDTemplate 指定对话框条资源的标识符；参数 nStyle 指定对话框条的风格；参数 nID 指定对话框条的标识符。

这样就完成了向 TestBars 程序中添加一个对话框条的任务。用户可以编译并运行 TestBars 程序，运行中的程序如图 8-28 所示。

注意图 8-28 中的新的工具条和对话框条。当然，现在的 TestBars 程序的对话框条只是正确地显示出来了，并不能真正的实现交互功能。要实现具体的程序功能需要对对话框条中的控件动作加入处理函数，这完全类似于无模式对话框中控件的处理，对此我们不再赘述。有一点要指出的是，对对话框条并不需要建立新的类，对话框条控件的动作完全由对话框条的父窗口——通常都是应用程序的主框架窗口来进行处理。

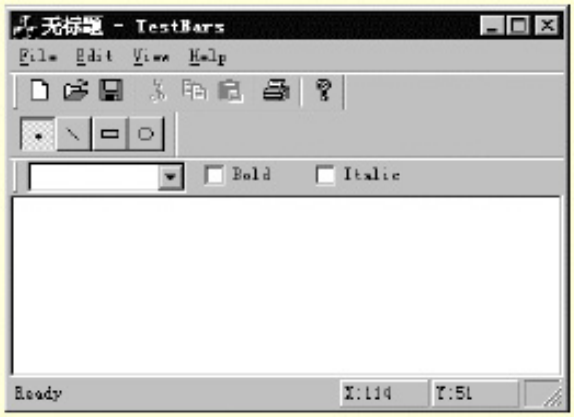

图 8-28 运行中的 TestBars 程序（具有新的工具条和对话框条）

8.4 集合条

集合条是 Visual C++ 6.0 才提供的新型的控制条。集合条中通常都包含着一个或几个段（Band），每个段通常都具有控制棒（Gripper Bar）、背景位图（Background Bitmap）、标志文本（Label Text）和子窗口（Child Window），但一个段中只能有一个子窗口。图 8-29 中解释了集合条的各组成部分。

图 8-29 集合条的各组成部分

集合条和集合条控件是紧密联系在一起的。实际上，CReBar 类仅仅完成集合条对象的创建工作，更多的操作是由 CReBarCtrl 类完成的。下面我们通过自己的实践来体会创建集合条的过程。

在真正的应用程序中，往往都是先构造好工具条或对话框条等，然后将这些建立好的元素插入集合条中，作为集合条的段来使用。当然，可以用作集合条的段的元素可以是工具条、对话框条、各种控件等等，只要是 CWnd 类的派生对象即可。

我们计划建立新的例子 AdvBar，在 AdvBar 程序中新建一个工具条，并使用一个集合条，将缺省的工具条和新建的工具条作为该集合条的两个段。

8.4.1 建立 AdvBar 程序框架

首先建立例子 AdvBar 程序的框架代码。使用 AppWizard 建立该程序的步骤如下：

（1）选择单文档界面，选中文档/视图结构支持，同时选择资源类型为英语。

（2）由于本程序不需要支持任何数据库技术，选择 None。

（3）由于本程序不需要支持任何形式的复合文档，选择 None；同时本程序也不需要包含 ActiveX 控件，故清除缺省对 ActiveX controls 的选中。

（4）本程序不涉及有关打印的知识，故清除 Print and print preview 选项；但需要缺省的工具条和状态条，故选中 Docking toolbar 和 Initial status bar 选项；工具条风格选择 Normal；其余接受缺省选择即可。

（5）接受缺省选择。

（6）最后，单击 Finish 按钮完成 AdvBar 的设置并产生框架代码。

8.4.2 建立新的工具条

在 TestBars 程序中也曾经创建过工具条，但那时创建的工具条是旧式的。用户如果使用过 Microsoft 公司的浏览器 Internet Explorer 4.0 的话，肯定还记得其中的新式的工具条：在鼠标没有指向工具条按钮的时候，按钮颜色比较灰暗；如果鼠标指向工具条的某一按钮，该按钮除了凸起显示外，颜色也变得比较鲜艳；另外，某些按钮的右边还有一个向下的箭头，单击该箭头将弹出更多的选择项。在 AdvBar 程序中，我们将创建一个与 IE4.0 类似的工具条，如图 8-30 所示，这需要付出较多的努力。

图 8-30 新的工具条

1．修改程序资源

先修改程序的菜单资源，在 View 菜单和 Help 菜单中添加新的 CD 菜单，如图 8-31 所示。

图 8-31 新的 CD 菜单

我们计划是增加一组菜单命令和工具条按钮，以控制 CD 的播放，但由于本章讨论的重点并不在于如何实现 CD 的播放控制，因此 AdvBar 程序只是提供了所需的界面。各菜单项的标识符和提示信息列于表 8-5 中。

表 8-5 新工具条按钮标识符和提示信息

标识符	提示信息
IDS_PLAY	Play current track\nPlay
IDS_FIRST	Go to the first track\nFirst
IDS_PREVIOUS	Go to the previous track\nPrevious
IDS_STOP	Stop play\nStop
IDS_NEXT	Go to the next track\nNext
IDS_LAST	Go to the last track\nLast
IDS_PAUSE	Pause play\nPause
IDS_TRACK	Select track\nTrack

接下来的任务是创建工具条资源。由于工具条的活动状态与非活动状态的显示是不同的，因此我们需要对工具条的两种状态创建相应的资源，仍然使用工具条编辑器创建工具条资源。先创建非活动状态的工具条的资源，如图 8-32 所示。

图 8-32 非活动状态的工具条

注意非活动状态的工具条按钮的背景色为浅灰色，前景色为深灰色。工具条按钮的标识符选择相应的菜单项的标识符即可。（记下保存非活动状态工具条的位图文件的文件名：toolbar1.bmp）。

然后创建活动状态的工具条，如图 8-33 所示。

图 8-33 活动状态的工具条

注意活动状态的工具条按钮的背景色为浅灰色，前景色为黄色。（遗憾的是这本书只能印刷成黑白两色，所以无法观察到颜色的区别。）特别需要注意的是对应按钮上图象的位置一定要一一对应，否则可能在运行时工具条的显示将不正确。（记下保存活动状态工具条的位图文件的文件名：toolbar2.bmp）。

最后，由于在建立工具条的时候需要的是图象列表，因此必须将保存工具条的位图载入为位图（Bitmap）资源。选择 Insert 菜单的 Resource 菜单项，在 Insert Resource 对话框中，选择 Bitmap 类型，然后单击 Import 按钮，在 res 子目录下找到 toolbar1.bmp 文件和 toolbar2.bmp 文件，完成载入过程。并将 toolbar1.bmp 的标识符改为 IDB_COOLBMP，将 toolbar2.bmp 的标识符改为 IDB_HOTBMP。由于在后面的工作中不再需要工具条资源，因此可以删除刚建立的两个工具条资源。

实际上，上面的工作就是利用工具条编辑器建立了两个位图资源。之所以要用工具条编辑器，是因为建立的位图资源是用作工具条对象的图象列表，要求每幅图象的大小一致，而工具条编辑器建立的每个按钮的图象都是 16X15 象素（缺省值），正好能满足要求。用户完全可以直接使用位图编辑器编辑所需的位图资源，但可能操作起来要麻烦一些。

以上完成了 AdvBar 程序的资源修改工作，下面的任务就是添加代码，完成工具条的创建工作。

2．建立工具条

首先仍然是向 CMainFrame 类添加新的成员，可以直接在 CMainFrame 类的头文件中的合适位置进行修改，修改后的一段代码如下所示：

```
protected:    // control bar embedded members
CStatusBar    m_wndStatusBar;
CToolBar      m_wndToolBar;
CToolBar   m_wndPlayBar;
CReBar           m_wndReBar;
```

这里新增了一个 CToolBar 类对象 m_wndPlayBar 和一个 CReBar 类对象 m_wndReBar，该 CReBar 类对象将在后面建立集合条时使用。

创建新的工具条的工作自然应该在 CMainFrame 类的 OnCreate()函数中完成，我们可以直接在 OnCreate()函数中添加合适的代码。但为了清楚起见，为 CMainFrame 类增加一个成员函数 CreatePlayBar()，用于创建新工具条的工作，该函数类型为 void，访问级别为 protected，不需要任何参数。在 OnCreate()函数中调用 CreatePlayBar()函数完成新工具条的创建工作。

创建新工具条的第一步就是创建工具条对象本身。这里使用 CToolBar 类的 OnCreateEx()函数，如下所示：

```
m_wndPlayBar.CreateEx(this);
```

CreateEx()函数的用法在 8.2 节中已有介绍，此处不再赘述。

在例子程序 TestBars 中建立工具条对象后，使用了 CToolBar 类的 LoadToolBar()函数直接将工具条对象与工具条资源联系起来，从而完成了工具条的创建。但是在 AdvBar 程序中，为了达到“Cool”和“Hot”两种不同的效果，我们使用另外的一种方法。

在 8.1 节中提到过，CToolBar 中包含了 CToolBarCtrl，可以通过操作 CToolBarCtrl 来操作 CToolBar。要达到“Cool”和“Hot”两种不同的效果，就必须使用 CToolBarCtrl 类的成员函数，我们可以通过 CToolBar 类的 GetToolBarCtrl()函数获得一个指向 CToolBarCtrl 对象的指针。

CToolBarCtrl 类的 SetImageList()函数和 SetHotImageList()函数设置工具条非活动状态时的图象和活动状态时的图象。两函数的原型如下：

```
CImageList* SetImageList( CImageList* pImageList );
CImageList* SetHotImageList( CImageList* pImageList );
```

参数 pImageList 为指向图象列表的指针。为了适合此要求，先构造一 CImageList 对象，将该对象与相应的

图象列表联系起来，就可以使用该列表了，因此设置工具条图象的一段代码如下：

```
CImageList img;
img.Create(IDB_COOLBMP,16,0,RGB(255,255,255));
m_wndPlayBar.GetToolBarCtrl().SetImageList(&img);
img.Detach();
img.Create(IDB_HOTBMP,16,0,RGB(255,255,255));
m_wndPlayBar.GetToolBarCtrl().SetHotImageList(&img);
img.Detach();
```

接下来设置新工具条的按钮的数目和标识符，这由 CToolBar 类的 SetButtons()函数完成，注意下面这一小段代码：

```
UINT btn[]={
        IDS_PLAY,
        IDS_FIRST,
        IDS_PREVIOUS,
        IDS_STOP,
        IDS_NEXT,
        IDS_LAST,
        IDS_PAUSE,
        IDS_TRACK};
m_wndPlayBar.SetButtons(btn,8);
```

首先构造了 btn 数组用于保存新工具条上个按钮的命令标识符，注意标识符的顺序与按钮的顺序是对应的。SetButtons()函数的原型如下：

```
BOOL SetButtons( const UINT* lpIDArray, int nIDCount );
```

参数 lpIDArray 为一指向标识符数组的指针，这里就是 btn 数组；参数 nIDCount 指定工具条中按钮的数目，这里是 8 个按钮。

然后指定工具条中各按钮的信息，主要是指定每个按钮所用的标识符、风格和位图，以及指定按钮下方的字符串。由于在我们在构造 btn 数组的时候注意了标识符的顺序与按钮的顺序对应，因此可以使用一个循环来完成这一工作。这一段代码如下所示：

```
for(int i=0;i<=7;i++)
{
CString s;
s.LoadString(btn[i]);
int n=s.GetLength()-s.Find('\n');
s=s.Right(n);
m_wndPlayBar.SetButtonInfo(i,btn[i],TBSTYLE_BUTTON,i);
m_wndPlayBar.SetButtonText(i,s);
}
m_wndPlayBar.SetButtonInfo(7,btn[7],TBSTYLE_BUTTON|TBSTYLE_DROPDOWN,7);
```

函数 SetButtonInfo()的原型如下：

```
void SetButtonInfo( int nIndex, UINT nID, UINT nStyle, int iImage );
```

参数 nIndex 指定进行设置按钮的编号，按照按钮在工具条上从左到右的顺序，编号从 0 开始；nID 指定该按钮使用的标识符；nStyle 指定按钮的风格；iImage 指定按钮使用的位图的编号。从该函数的定义可以看出，我们在创建按钮位图资源，或构造标识符数组的时候，不一定需要位图、标识符等的顺序与按钮的顺序一致，只要在使用 SetButtonInfo()函数的时候能够正确地指定就行了。

函数 SetButtonText()的原型如下：

```
BOOL SetButtonText( int nIndex, LPCTSTR lpszText );
```

参数 nIndex 同样指定设置按钮的编号；参数 lpszText 指定按钮使用的字符串。在获得 lpszText 参数的取值时，我们用了一个小小的技巧。我们构造了一个 CString 对象，并最终让该 CString 对象的值取为对应标识符代表的字符串的“ \n ”字符后面的部分，再将其传递给 SetButtonText()函数。

注意图 8-30 中最后一个按钮的右边有一个向下的箭头按钮，这意味着单击该箭头按钮将弹出更多的选项。为达到这个效果，在最后特别用一行指定了 Track 按钮的风格：

```
m_wndPlayBar.SetButtonInfo(7,btn[7],TBSTYLE_BUTTON|TBSTYLE_DROPDOWN,7);
```

风格 TBSTYLE_DROPDOWN 说明在该按钮的后面附加一个下拉箭头按钮。然而，仅仅是为按钮指定了下拉箭头按钮还不够，工具条本身也需要支持下拉箭头按钮，这使用 CToolBarCtrl 类的 SetExtendedStyle()函数指定：

```
m_wndPlayBar.GetToolBarCtrl().SetExtendedStyle(TBSTYLE_EX_DRAWDDARROWS)
```

风格 TBSTYLE_EX_DRAWDDARROWS 说明工具条支持下拉箭头按钮。只有工具条与按钮两方面都支持下拉箭头按钮，该风格才真正能实现。

建立新工具条最后一步的工作是设置工具条的尺寸，使用了下面这样一小段代码：

```
CRect rcBar;
m_wndPlayBar.GetItemRect(0,&rcBar);
m_wndPlayBar.SetSizes(rcBar.Size(),CSize(16,15));
```

函数 SetSizes()的原型为：

```
void SetSizes( SIZE sizeButton, SIZE sizeImage );
```

参数 sizeButton 指定按钮总的尺寸，这包括按钮位图、按钮下的文本以及按钮的边框；参数 sizeImage 指定按钮位图的尺寸。注意 sizeButton 比起 sizeImage 来，在高度方向上至少要大 6 个象素，在宽度方向上至少要大 7 个象素，否则程序将出错。

函数 GetItemRect()返回指定按钮的大小。在未指定按钮的尺寸前，这是按钮的缺省大小。

至此已经完成了新的工具条的创建，完整的 CreatePlayBar()函数如程序清单 8-13 所示。

程序清单 8-13　CMainFrame::CreatePlayBar()

```
void CMainFrame::CreatePlayBar()
{
CImageList img;
UINT btn[]={
        IDS_PLAY,
        IDS_FIRST,
        IDS_PREVIOUS,
        IDS_STOP,
        IDS_NEXT,
        IDS_LAST,
        IDS_PAUSE,
        IDS_TRACK};

//set the dedault image and hot image and bar style
m_wndPlayBar.CreateEx(this);

img.Create(IDB_COOLBMP,16,0,RGB(255,255,255));
m_wndPlayBar.GetToolBarCtrl().SetImageList(&img);
img.Detach();
img.Create(IDB_HOTBMP,16,0,RGB(255,255,255));
m_wndPlayBar.GetToolBarCtrl().SetHotImageList(&img);
```

```
img.Detach();
m_wndPlayBar.SetButtons(btn,8);

//set infomation of each button
for(int i=0;i<=7;i++)
{
    CString s;
    s.LoadString(btn[i]);
    int n=s.GetLength()-s.Find('\n');
    s=s.Right(n);
    m_wndPlayBar.SetButtonInfo(i,btn[i],TBSTYLE_BUTTON,i);
    m_wndPlayBar.SetButtonText(i,s);
}
m_wndPlayBar.SetButtonInfo(7,btn[7],TBSTYLE_BUTTON|TBSTYLE_DROPDOWN,7);
m_wndPlayBar.GetToolBarCtrl().SetExtendedStyle(TBSTYLE_EX_DRAWDDARROWS);

//set the size of the playbar
CRect rcBar;
m_wndPlayBar.GetItemRect(0,&rcBar);
m_wndPlayBar.SetSizes(rcBar.Size(),CSize(16,15));
}
```

为了使 CD 菜单的各项命令能够起作用，需要响应菜单项的 COMMAND 命令消息，使用 ClassWizard 进行消息映射工作，由 CAdvBarView 类接收菜单项命令消息，接受缺省提供的函数名。

在 CMainFrame 类的 OnCreate()函数中需要调用 CreatePlayBar()函数，就在 OnCreate()函数的 return 语句前加上一行：

```
CreatePlayBar();
```

现在用户可以编译和运行 AdvBar 程序，新的工具条应该能够正确地显示出来。当然，由于我们没有实际地添加代码驱动 CD 播放器工作，选择菜单命令或单击工具条按钮不会有任何实际效果。

8.4.3 建立集合条

建立集合条也应该在 CMainFrame 类的 OnCreate()函数中进行，同样我们另建一个成员函数来完成此项工作。函数名为 CreateReBar()，类型为 Void，访问级别为 protected。

在已经具有缺省工具条和新工具条的基础上，建立集合条的工作是很简单的。首先建立集合条对象，使用 CReBar 类的 Create()函数，该函数的原型如下：

```
BOOL Create( CWnd* pParentWnd, DWORD dwCtrlStyle = RBS_BANDBORDERS, DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | WS_CLIPCHILDREN | CBRS_TOP, UINT nID = AFX_IDW_REBAR );
```

参数 dwCtrlStyle 指定 CReBarCtrl 对象的风格，这里是 RBS_BANDBORDERS，这意味着在集合条的段间加上细的边框进行区分。其他参数的含义类似于 CToolBar 类的 CreateEx()函数。

在建立集合条对象后，需要将已存在的缺省工具条和新建的工具条作为集合条的段加入到集合条中。CReBar 类的 AddBar()函数完成这一工作：

```
BOOL AddBar( CWnd* pBar, COLORREF clrFore, COLORREF clrBack, LPCTSTR lpszText = NULL, DWORD dwStyle = RBBS_GRIPPERALWAYS );
```

参数 pBar 指向将作为子窗口加入到集合条中的 CWnd 对象；参数 clrFore 和参数 clrBack 分别指定段的前景色和背景色，可以使用 RGB 宏来构造适当的颜色；参数 lpszText 代表段的标志文本，也可以使用一个 CString 对象作为此参数；参数 dwStyle 指定段的风格。

因此，添加集合条的一段代码如下：

```
m_wndReBar.Create(this);
m_wndReBar.AddBar(&m_wndToolBar);
CString s="CD:";
m_wndReBar.AddBar(&m_wndPlayBar,RGB(255,0,0),RGB(192,192,192),s,RBBS_GRIPPERALWAYS|RBBS_BREAK);
```

建立集合条和向集合条中添加段都是简单的，重要的工作在于设置集合条中各段的各种信息，这些信息都保存在一个 REBARBANDINFO 结构中，表 8-6 列出了该结构的各元素名称及说明。

表 8-6 REBARBANDINFO 结构

元 素	类 型	说 明
cbSize	UINT	结构的大小，以字节为单位。
fMask	UINT	标志结构的有效元素
fStyle	UINT	指定段的风格
clrFore	COLORREF	指定段的前景色
clrBack	COLORREF	指定段的背景色
lpText	LPTSTR	指定段的标志文本
cch	UINT	指定标志文本的长度
iImage	int	指定段中显示的位图的编号
hwndChild	HWND	段的子窗口的句柄
cxMinChild	UINT	子窗口的最小宽度
cyMinChild	UINT	子窗口的最小高度
cx	UINT	段的宽度
hbmBack	HBITMAP	用作段背景的位图的句柄
wID	UINT	段的标识符
cyChild	UINT	段的初始高度
cyMaxChild	UINT	段的最大高度
cyIntergral	UINT	段改变大小的步长
cxIdeal	UINT	段的理想宽度
lParam	LPARAM	一个 32 位的程序指定的数值
cxHeader	UINT	段头部的宽度

结构 REBARBANDINFO 的成员很多，但并不是所有的成员都要进行设置的，一般只对我们感兴趣的成员进行设置，其他的成员取缺省值即可；如果对段的状态没有什么要求，甚至可以不设置该结构的任何成员，所有的参数都取缺省值。这里也不准备对所有的成员都给出详细的解释，用户可以参阅 Visual C++的联机文档以获得更多的信息。

使用一个 REBARBANDINFO 结构，首先要指定该结构的大小，这可以使用 sizeof 关键字完成。然后利用 fMask 成员指定需要设置的成员，比如说，如果 fMask 成员取 RBBIM_BACKGROUND|RBBIM_COLOR，则需要指定段的背景位图、前景色和背景色。

在 AdvBar 程序的集合条中，主要对集合条两个段的各种尺寸进行了设置，这一段代码如下所示：

```
//set info of the first band
CRect rcBar;
m_wndToolBar.GetItemRect(0,&rcBar);
rbbi.cxMinChild=rcBar.Width();
rbbi.cyMinChild=rcBar.Height();
rbbi.cx=rbbi.cxIdeal=rcBar.Width()*8;
m_wndReBar.GetReBarCtrl().SetBandInfo(0,&rbbi);

//set info of the second band
m_wndPlayBar.GetItemRect(0,&rcBar);
rbbi.cxMinChild=rcBar.Width();
rbbi.cyMinChild=rcBar.Height();
rbbi.cx=rbbi.cxIdeal=rcBar.Width()*8;
```

```
m_wndReBar.GetReBarCtrl().SetBandInfo(1,&rbbi);
```

这里我们将段的最小尺寸设置为一个按钮大小，段的理想宽度与段的总宽度一致。程序清单 8-14 **中是完整的** CreateReBar()**函数。**

程序清单 8-14　CMainFrame::CreateReBar()

```
void CMainFrame::CreateReBar()
{
//create the rebar
m_wndReBar.Create(this);

//add the default toolbar and playbar to the rebar
m_wndReBar.AddBar(&m_wndToolBar);
CString s="CD:";
m_wndReBar.AddBar(&m_wndPlayBar,RGB(255,0,0),RGB(192,192,192),s,RBBS_GRIPPERALWAYS|RBBS_BREAK);

//set up sizes of each band
REBARBANDINFO rbbi;
rbbi.cbSize=sizeof(rbbi);
rbbi.fMask=RBBIM_CHILDSIZE|RBBIM_IDEALSIZE|RBBIM_SIZE;

//set info of the first band
CRect rcBar;
m_wndToolBar.GetItemRect(0,&rcBar);
rbbi.cxMinChild=rcBar.Width();
rbbi.cyMinChild=rcBar.Height();
rbbi.cx=rbbi.cxIdeal=rcBar.Width()*8;
m_wndReBar.GetReBarCtrl().SetBandInfo(0,&rbbi);

//set info of the second band
m_wndPlayBar.GetItemRect(0,&rcBar);
rbbi.cxMinChild=rcBar.Width();
rbbi.cyMinChild=rcBar.Height();
rbbi.cx=rbbi.cxIdeal=rcBar.Width()*8;
m_wndReBar.GetReBarCtrl().SetBandInfo(1,&rbbi);
}
```

在 CMainFrame **类的** OnCreate()**函数中** return **语句前加入以下对** CreateReBar()**函数的调用：**

```
CreateReBar();
```

另外，由于 CReBar **类不支持** CControlBar **类提供的停泊功能，因此应该将** OnCreate()**函数中有关** m_wndToolBar **对象的停泊功能的语句删除或注释掉；同时去除** m_wndToolBar **对象的** CBRS_GRIPPER **风格。修改后的** OnCreate()**函数如程序清单** 8-15 **所示。**

程序清单 8-15　CMainFrame::OnCreate()

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
```

```
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
    }

    CreatePlayBar();
    CreateReBar();
    return 0;
    }
```

到现在为止，我们已经完成了 AdvBar **程序的新的工具条和集合条的创建工作，用户可以编译和运行** AdvBar **程序，并比较使用了集合条和未使用集合条的** AdvBar **程序在外观上有何不同。实际上，集合条的用处还有很多，这里不可能都加以详细介绍，有兴趣的用户可以参阅** Visual C++**的联机文档。**

8.5 本章小结

对 Windows **应用程序而言，具有一个优秀的程序界面并不是一件太困难的事，本章中主要就工具条方面的增强与改进做了一些介绍。**

最后需要说明的是，学习完本章后读者应该重点掌握以下内容：

- **控制条的基本知识**
- **创建和使用工具条**
- **定制状态条**
- **创建和使用使用集合条**

第九章　数据库的开发和使用

目前，数据库的研究和使用是计算机最活跃的领域之一。使用数据库管理数据有很明显的优点，数据冗余少，可以对数据的一致性和完整性统一控制，实现了数据共享，能快速查找等，因此数据库管理系统在不同的领域，不同的部门得到了广泛的应用。如今，很多小型应用程序都用到了数据库，大中型的应用程序则几乎就离不开数据库，所以数据库编程是 VC++编程的一种基本而且重要的技能。

注意：

- 本书重点讲解 VC 中对数据库的使用，涉及到数据库的基本概念仅有一小节内容，更多的知识请读者查阅相关书籍。

9.1　关系数据库模型

市场上的数据库产品绝大部分都是关系数据库管理系统（RDBMS），虽然基于新的数据模型（如对象数据模型）的数据库产品已经面市，但 RDBMS 在市场上的主导地位并没有改变，并将保持数年。

关系模型是 RDBMS 的基础，是 1970 年 IBM 公司的 E.F.Codd 在论文《一个通用关系式数据库系统的模型》中首先提出的。关系模型提供了优异的数据独立性和数据相容性，它由三个部分组成：

- 数据结构——模型中对象和类型的集合。
- 完整性规则——确保数据有效和正确的约束条件。
- 数据操作——对模型中的对象所允许的操作方式。

9.1.1　数据结构

一个关系就是一个二维的数据表格，每一列对应实体的一个属性，其中给出相应各实体的属性值，每一行形成一个由多种属性组成的多元组，与一个特定的实体相对应。表中所有元组必须是各不相同，列值必须是不可分割的，数组和集合都不能作为列值。表 9-1、表 9-2 和表 9-3 给出了关系数据库的一个示例，它们是学校数据库中的三张表：

表 9-1　教工表

TeacherID（教工编号）	Name（姓名）	Sex（性别）	BirthDate（出生日期）	BirthPlace（出生地）
T0001	张辉	男	1962．7	北京
T0002	李毅	男	1958．2	浙江
T0003	夏雪	女	1966．4	河北

表 9-2　学生表

StudentID（学生编号）	Name（姓名）	Sex（性别）	BirthDate（出生日期）	BirthPlace（出生地）	ClassID（班号）
S0001	王晓刚	男	75．8	北京	31
S0002	赵枫	女	75．4	北京	31
S0003	孙茵	女	74．10	天津	32

表 9-3 班级表

ClassID（班号）	DirectorID（班主任）	StudentNumber（人数）	AverageGrade（平均成绩）
31	T0001	31	84．2
32	T0003	32	85．1

列的值可以为空。空值一般用 null 表示，与空格或零完全不同。

在关系数据库中，关系（relation）对应于表（table），元组（tuple）对应于行（row），属性（attribute）对应于列（column）。

9.1.2 完整性规则

在关系数据库模型中，各行是互不相同的，即存在一个或多个列，其在每一行上的值都互不相同，所以可以用某一或某几列唯一标志一行，这样的列称为表的关键字，也称为码。所谓关键字，就是说当此值确定后，该行其余列的列值也就唯一确定了，而且关键字的真子集不再具有这样的性质。如果关键字多于一个，则选定其中的一个作为主关键字，其余的称为侯选关键字。在支持主关键字约束的 RDBMS 中，若插入行的主关键字不唯一，该行将被拒绝接受，并返回出错信息。在表 9-1 教工表中，可以选定教工编号作为主关键字，其它列不能作为关键字，因为不同行中的列值可能相同，如姓名相重是很常见的事情。

关系模型要求主关键字的任一属性不能为空，这就是实体完整性。因为主关键字用来标志表中的行，而带有空值的主关键字无法标志行。

关系模型中各表之间是有相互联系的，可以通过在一张表中包含另一张表的关键字来表示表之间的联系，这种被包含的其它表的关键字称为外部码。码和外部码提供了表示关系间联系的方法。

外部码必须为它所参照表的主关键字的某个值或为空，这就是关系模型的第二个完整性规则——参照完整性。

9.1.3 数据操作

关系数据库之所以发展如此之快，是因为关系模型简明，便于用户理解。同时，关系模型有着坚实的数学基础，操作表达能力很强大，这又给关系数据库的发展提供了保证条件。

关系代数的运算可分为两类，一类是特殊的关系运算，包括选择（Selection），投影（Projection），连接（Join）等；另一类是通常的集合运算，包括并（Union），差（Difference），交（Intersection）和笛卡儿积（Cartesian Product）等。

■ 选择（Selection）：

选择操作返回满足条件的表中行的子集，用于对表的横向行的选择。

设 R 是一个关系，F 是一个命题公式，则在关系 R 上的 F 选择是在 R 中选出满足 F 的所有元组组成的新关系，这个新关系是 R 的一个子集。

■ 投影（Projection）：

投影操作返回表中列的子集，用于对表的纵向列的投影

■ 连接（Join）：

连接操作用于从多个表中获取结果。连接的种类很多，最常用的是自然连接。自然连接是指将一个表的属性与另一个表中的属性进行匹配连接，两个表通常含有一个同名列。按同名列自然连接后的结果集中不包含不匹配的行。

■ 并（Union）：

设 R，S 为同类关系，则 R，S 的并是由属于 R 或 S，或同时属于 R 和 S 的元组构成的集合，记作 R∪S。

■ 差（Difference）：

设 R，S 为同类关系，则 R，S 的差是由属于 R 而不属于 S 的元组构成的集合，记作 R-S。

■ 交（Intersection）：

设 R，S 为同类关系，则 R，S 的交是由同时属于 R 和 S 的元组构成的集合，可以记作 R∩S。

■ 笛卡儿积（Cartesian Product）：

设 R 为 K1 元关系，S 为 K2 元关系，则 R，S 的笛卡儿积 R×S 是一个（K1+K2）元组构成的集合，其中元组的前 K1 个分量是 R 的一个元组，后 K2 个分量是 S 的一个元组。R×S 是所有满足这种条件的元组构成的集合。

9.1.4 SQL 语言

SQL 是基于关系代数的一种关系数据查询语言，是关系数据库领域中的主流语言，也是 ANSI 标准，各个数据库产品公司纷纷推出了各自的 SQL 软件或与 SQL 的接口。

SQL 能够完成定义数据库，录入数据建立数据库，提供用户查询，更新，维护，扩充等操作，并具有保障数据安全的操作，因此是一个功能极强的数据库语言。SQL 的语言简洁，为完成核心功能只用了 6 个动词，即 SELECT，CREATE，INSERT，UPDATE，DELETE，GRANT，完成数据库查询（QUERY），数据定义（DDL），数据操作（DML）和数据控制（DCL）功能，是一个一体化语言。

1．SQL 查询命令

数据库查询命令 SELECT 是 SQL 的核心，SQL 对数据库的操作十分灵活方便，原因就在于 SELECT 命令的语句成分丰富多样。SELECT 的命令格式如下：

```
SELECT 结果表
FROM 表或视图
[WHERE 条件表达式]
[GROUP BY 列名 1]
[HAVING 内部函数条件式]
[ORDER BY 列名列表 2]
```

其中，[ ]中的内容是可选的。

该 SELECT 语句的含义是：从“表或视图”中选取满足“条件表达式”的数据元素形成结果表，且选出的数据元素在结果表中要根据“列名列表 2”排序。GROUP 子句用于按给定“列名 1”对 COUNT（个数），AVG（平均值），SUM（总和），MAX（最大值），MIN（最小值）等统计函数的计算进行分组。HAVING 子句表示只有满足“内部函数表达式”的数据元素才输出。

在查询命令的条件表达式中有一些经常使用的谓词，我们简单介绍一下：

■ LIKE 谓词：

用于字符串的模式匹配，字符‘ _ ’匹配一个任意字符，字符‘ % ’匹配零个或多个任意字符。例如，查询姓李的教师的 SQL 语句为：

```
SELECT *
FROM Teacher
WHERE Name LIKE ‘李%’
```

■ IN 谓词：

判断列值是否在某一个集合中。例如，查询教师编号为 T0001，T0003 的教师的 SQL 语句为：

```
SELECT *
FROM Teacher
WHERE TeacherID IN ( ‘T0001’ ,‘T0003’ )
```

■ BETWEEN 谓词：

判断列值是否在两个值之间。例如，查询查询教师编号在 T0001，T0003 之间的教师的 SQL 语句为：

```
SELECT *
FROM Teacher
WHERE TeacherID BETWEEN ‘T0001’  AND ‘T0003’
```

2．SQL 数据定义语句（DDL）

SQL DDL 包含 CREATE，DROP，ALTER。

■ CREATE 命令

CREATE 命令定义表（TABLE），视图（VIEW）和索引（INDEX）。

定义表的命令格式为：

CREATE TABLE 表名 （列名 1，数据类型[NOT NULL [UNIQUE]] [，列名 2，数据类型[NOT NULL [UNIQUE]]]…）

表名是数据表的标识，在系统中必须唯一，表名后的括号中给出了这个表的全部列名以及它们的数据类型。

定义视图的命令格式为：

CREATE VIEW 视图名 [(列名 1 [，列名 2]…)] AS SELECT 语句

命令为所需建立的视图命名，视图中包含的列名由 SELECT 语句规定，视图名后的列名仅当对所选的列名需要换名时才需要。

定义索引的命令格式为：

CREATE INDEX 索引名 ON 表名（列名 1 [，列名 2]…) [ASC 或 DESC]

索引建立在一个或多个列上，缺省按索引码的升序创建，，还可以用 ASC 或 DESC 显示说明索引是升序还是降序的。

■ DROP 命令：

DROP 命令用于删除表，视图或索引。

删除数据表的命令格式为：

DROP TABLE 表名

删除视图的命令格式为：

DROP VIEW 视图名

删除索引的命令格式为：

DROP INDEX 索引名

■ ALTER 命令：

ALTER 命令用于修改数据表的结构。

向数据表添加一列的命令格式为：

ALTER TABLE 表名 ADD COLUMN 列名 列的数据类型

从数据表删除一列的命令格式为：

ALTER TABLE 表名 DROP COLUMN 列名

3．SQL 数据操作语句（DML）

SQL DML 有三个命令，INSERT，DELETE 和 UPDATE。

■ INSERT 命令：

用来在表中插入一个或一组记录。

插入一个记录的格式为：

INSERT INTO 表名 VALUES（ 行值列表）

要从已有的表中选取一组记录插入到另一张表中，用下面的格式：

INSERT INTO 表名 SELECT 语句

上面的命令把 SELECT 语句查询的结果插入到表中。

如果插入时有的列没有给出列值，那么这些列的值为空。

■ DELETE 命令：

用于从表中删除一个或多个记录，其格式为：

DELETE FROM 表名 [WHERE 条件表达式]

如果没有 WHERE 子句，则删除表中的所有记录，但没有删除表。有 WHERE 子句时，删除表中满足条件表达式的记录。

■ UPDATE 命令：

用于修改表中的记录，其格式为：

UPDATE 表名 SET 列名 1=表达式 1 [，列名 2=表达式 2]… [WHERE 条件表达式]

通常 UPDATE 命令应该包括 WHERE 子句以决定要修改哪些记录，如果没有 WHERE 子句，则表中所有的

记录都要被修改。

4．SQL 数据控制语句（DCL）

SQL 的数据控制功能包括存取控制和完整性控制。

为防止非法用户使用或破坏数据库，用 GRANT 和 REVOKE 语句对操作权限进行控制，为保证数据的完整性，用 ASSERT 语句进行完整性控制。

- GRANT 语句：

把表的操作权限授予用户，其格式为：

GRANT <操作权限表> ON 表名 TO 〈用户列表〉

- REVOKE 语句：

从用户处收回表的操作权限，其格式为：

REVOKE <操作权限表> ON 表名 FROM〈用户列表〉

- ASSERT 语句：

ASSERT 语句是完整性断言定义语句，它定义了数据在插入和更新时必须满足的条件，其格式为：

ASSERT 完整性断言 ON 表名

9.2 使用 ODBC

本节讲述了 ODBC 是什么，如何使用 VC++和 MFC ODBC 类建立工作在 ODBC 数据库上的应用程序，如何浏览、增加、删除和修改数据库中的记录，以及如何操作数据库中的多张数据表。

9.2.1 ODBC 概述

目前，关系数据库是数据库产品的主流，从小型数据库（如 Foxpro，Access 等）到大型数据库（如 Oracle，Sybase，Informix，DB2 等），用户可以根据实际需要方便地进行选择。但由于各种数据库间难以相互访问，人们迫切希望有一个一致的数据库操作接口，使数据库应用程序能够独立于数据库产品。正是在这种要求下，Microsoft 于 1991 年 1 月发布了 ODBC，并迅速得到了广大用户的认可，各主要数据库厂商纷纷也推出支持 ODBC 的驱动程序。事实证明 ODBC 作为 Microsoft 的重大成就，已经为 Windows 操作系统建立了一个开放的接口。

ODBC 允许应用程序在数据库管理系统（DBMS）中使用 SQL 语言访问数据。作为一个一致的数据库操作接口，ODBC 提供了一个允许单一应用程序去访问许多不同数据库管理系统的机制，这种机制允许应用程序开发者不把一个特定的 DBMS 作为目标去创建应用程序，随后用户能够增加数据库驱动程序模块，该数据库驱动程序把应用程序连接到用户选择的 DBMS 上。

在 ODBC 程序员手册中，ODBC 的定义是：

- ODBC 函数调用库，它允许应用程序连接到 DBMS，执行 SQL 语句以返回检索结果。
- SQL 语法，它是基于 X/Open 和 SQL 访问组（SAG）的 SQL CAE 说明书（1992）。

- 一个错误代码的标准集。
- 一个连接并记录在 DBMS 上的标准方法。
- 一个数据类型的标准表示。

从 ODBC 的最初提出到发展壮大的这段时间内，一直都是由 Microsoft 成功推动的，Microsoft 向业界保证要继续提供并发展 ODBC 的核心技术，因此无论现在还是可见的将来，Microsoft 的应用程序都将依赖 ODBC 而获得数据库的管理。

ODBC 支持如此众多的数据库，会不会很难使用？这是不会的。因为一个典型的应用程序调用的函数不会超过 20 个，如果使用 MFC 数据库类，从应用程序访问数据库的代码就更简单了。

1．ODBC 的工作原理

典型的 ODBC 结构如图 9-1 所示。

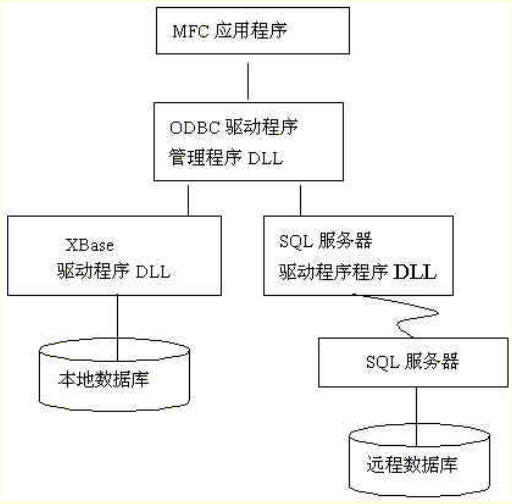

图 9-1 典型的 ODBC 结构

ODBC 通过使用数据库驱动程序（Driver）来提供数据库的独立性。驱动程序是用以支持 ODBC 函数调用的模块（通常是一个 DLL），ODBC 通过调用驱动程序所支持的函数来对数据库操作。驱动程序与具体的数据库有关，如果要访问不同类型的数据库，ODBC 就需要动态地连接到不同的驱动程序上。

ODBC 的另一个组件是驱动程序管理器（driver manager）。ODBC 驱动程序管理器(Administrator)位于 Windows 控制面板(ControlPanel)中的 32bitODBC 程序项中。驱动程序管理器包含在 ODBC32.DLL 中，能够连接到所有的 ODBC 应用程序中，它负责安装驱动程序，管理数据源，负责 ODBC 函数与 DLL 中函数的绑定，并帮助程序员跟踪 ODBC 的函数调用。在 ODBC 中，应用程序不能直接存取数据库，它必须通过管理器和数据库交换信息。ODBC 管理器负责将应用程序的 SQL 语句及其它信息传递给驱动程序，而驱动程序则负责将运行结果送回应用程序。应用程序，ODBC 管理器和数据库之间的关系仍然是如图 9-1 所示。

2．MFC ODBC 类

MFC ODBC 数据库类是为了简化数据库编程，使数据库开发者容易使用 ODBC，它的层次结构如图 9-2 所示。

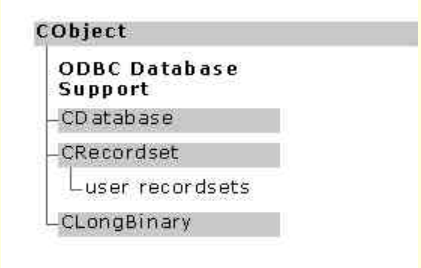

图 9-2 MFC ODBC 类的层次结构

在数据库类中，最主要的有 CDatabase，CRecordset 和 CRecordView。

类 CDatabase 代表与数据源的 ODBC 连接。它包含用于打开，关闭和配置连接的操作，用于执行不返回结果集的 SQL 语句，还可以控制与连接有关的 SQL 语句是否执行和执行的时间。基本的操作是：

- Open()——建立数据库对象与 ODBC 数据源间的连接。
- Close()——拆除数据库对象与 ODBC 数据源间的连接。
- IsOpen()——测试数据库对象是否与数据源连接了。
- ExecuteSQL()——执行不返回结果集的 SQL 语句，如 INSERT，UPDATE，DELETE 语句等。

类 CRecordset 代表一个对数据库的查询和从该查询返回的结果集。它包含一些设置查询条件的操作。用户很少需要从类 CDatabase 中派生出类，但却总是要从类 CRecordset 派生出一些类，以便同所使用的数据库中的表匹配。类 CRecordset 包含以下的操作：

- Open()——用于定义和执行当前查询，并打开结果集。
- Close()——关闭结果集，撤消查询。

- IsBOF()——测试是否滚动到第一个记录之前。
- IsEOF()——测试是否滚动到最后一个记录之后。
- MoveFirst()——移到第一个记录。
- MoveLast()——移到最后一个记录。
- MoveNext()——移到后一个记录。
- MovePrev()——移到前一个记录。
- Move()——相对当前位置移动指定个数的记录（可以向前也可以向后）。
- AddNew()——用于添加新记录，在添加新记录前调用，作用是创建一个新的空行和存储该行的缓冲区。填写新的记录后再调用 UpDate()完成添加新记录。
- Edit()——用于修改记录，在修改记录前调用，作用是创建存储该记录的缓冲区。修改记录后再调用 UpDate()完成修改。
- Delete()——删除当前行。
- UpDate()——把缓冲区中的值存储到数据库的当前记录中以完成添加和修改记录。
- Requery()——重新运行查询，更新结果集。

至于类 CRecordView，由这个类创建的对象取代了通常的视，提供显示和编辑当前记录的用户界面。在程序员创建新的数据库应用程序时，必须要在 CRecordView 的窗口内添加编辑控件，并且把这些编辑控件绑定到记录集中对应的数据库字段上。CRecordView 对象保持了与 CRecordset 对象的连接，可以在用户窗口控件和 CRecordset 之间传送数据。OnMove()函数处理数据的传送和记录集中的定位。

9.2.2 创建 ODBC 数据库应用程序

使用 VC++，程序员可以快速方便地创建 ODBC 数据库应用程序，其步骤是：

（1）建立 ODBC 数据源。

（2）用 AppWizard 创建数据库应用程序框架。

（3）根据需要的功能，向数据库应用程序框架中添加相应的代码。

在下面的小节中，我们将介绍创建和完善数据库应用程序 School 的方法，使应用程序能够对数据库表中的记录进行浏览，添加，删除，更新，排序等。

1．建立 ODBC 数据源

在创建数据库应用程序前，开发者必须先要注册建立应用程序要访问的作为 ODBC 数据源的数据库。注册建立 ODBC 数据源的操作步骤为：

（1）创建一个新文件夹 Database，使用 Microsoft Access 根据表 9-1，9-2 和 9-3 中的数据库结构在 Database 文件夹中创建数据库 School.mdb。以后我们要在将创建的 School 数据库应用程序中使用数据库 School.mdb 作为数据源。

（2）从 Windows 98 的启动菜单运行设置中的控制面板，在控制面板中，用鼠标双击“ODBC 数据源（32 位）”图标，就会出现“ODBC 数据源管理器”对话框，如图 9-3 所示。

图 9-3 ODBC 数据源管理器

（3）单击“添加”按钮，出现“创建新数据源”对话框，从驱动程序列表中选择 Microsoft Access Driver，然后单击“完成”按钮，如图 9-4 所示。现在 Microsoft Access Driver 成为与 School 程序所使用的数据源相关联的 ODBC 驱动程序。

（4）当“ODBC Microsoft Access 安装”对话框出现后，在“数据源名”文本框中输入 School，在“描述”文本框中输入 School Sample，如图 9-5 所示。数据源的名字是数据源的标识，不同的数据源应该有不同的名字；描述中的内容是对该数据源信息更详细的描述。

图 9-4 从 ODBC 驱动程序列表中选择与数据库类型相应的驱动程序

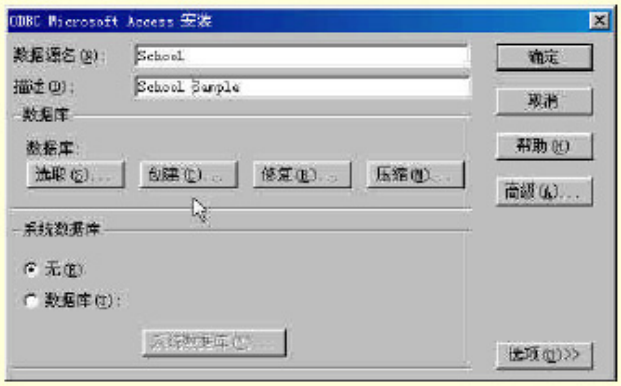

图 9-5 给数据源命名

（5）单击“选取”按钮，会出现“选定数据库”对话框，在该对话框中定位并选择数据库 School.mdb，如图 9-6 所示。

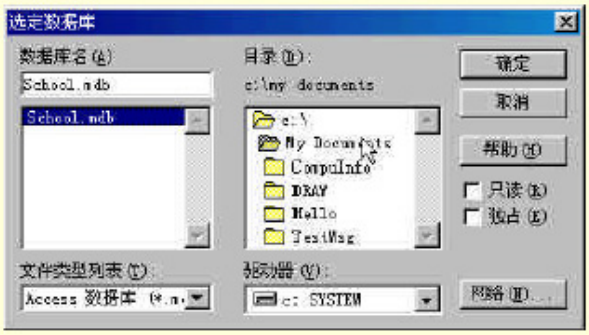

图 9-6 选择定位数据库文件

（6）单击 OK 按钮完成数据库选择，然后在“ODBC Microsoft Access 安装”对话框中单击“确定”OK 按钮，完成数据源的创建过程。

（7）最后单击“ODBC 数据源管理器”对话框中的“确定”按钮即可。

完成上述操作后，应用程序就可以通过 Microsoft Access ODBC 驱动程序访问数据库 School.mdb 了。

2. 创建数据库应用程序框架

创建了数据库 School.mdb 并注册建立数据源 School 后，就可以建立数据库应用程序 School 。创建的步骤如下，完成了这些步骤后，将会得到一个能浏览 School 数据库 Teacher 表的应用程序：

（1）从开发平台的菜单中选取 File 菜单项的 New 菜单命令，单击 Projects 标签。

（2）选择 MFC AppWizard (exe)程序类型，并在 Project name 编辑框中输入 School，如果要改变该项目所在的文件夹，在 Location 组合框中选择或输入该项目所在的文件夹，单击 OK 按钮，出现 Step 1 对话框。

（3）应用程序是一个简单的单文档界面应用程序，所以选择 Single Dcument，单击 Next 按钮。

（4）该应用程序是为了浏览数据库，不需要文件支持，选择 Database View Without File Support 选项，如图 9-7，AppWizard 将生成浏览数据库内容的类。再单击该对话框中的 Data Source 按钮，我们将把该应用程序与前面建立的数据源连接起来。

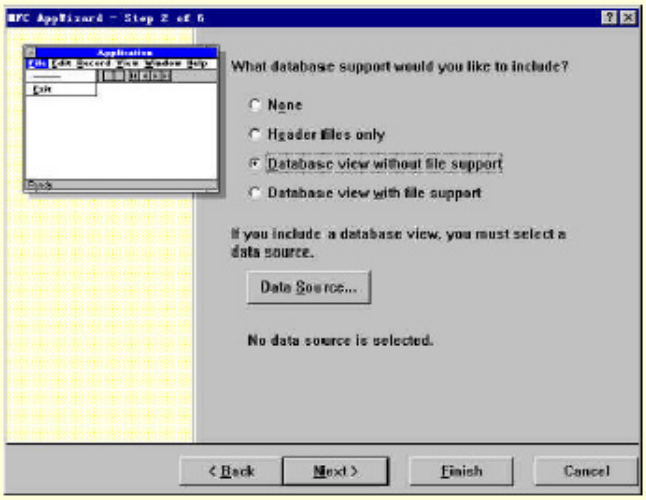

图 9-7 使用数据库视图，但没有文件支持

（5）单击 Data Source 按钮后，将会出现一个 Database Options 对话框，在此对话框中我们下拉 ODBC 列表，选择数据源 School，如图 9-8，单击 OK 按钮。

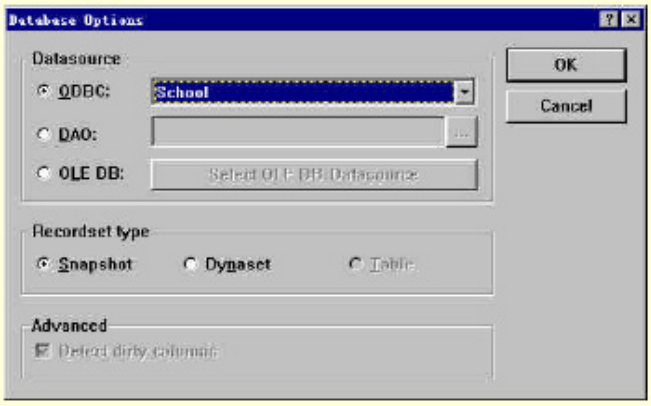

图 9-8 选择 School 为该应用程序连接的 ODBC 数据源

（6）在 Select Database Tables 对话框中，选择 Teacher 表，如图 9-9，单击 OK 按钮，现在，我们在数据源 School 中的 Teacher 表和应用程序之间建立了连接。然后，又出现 Step 2 对话框，单击 Next 按钮转到 Step 3。

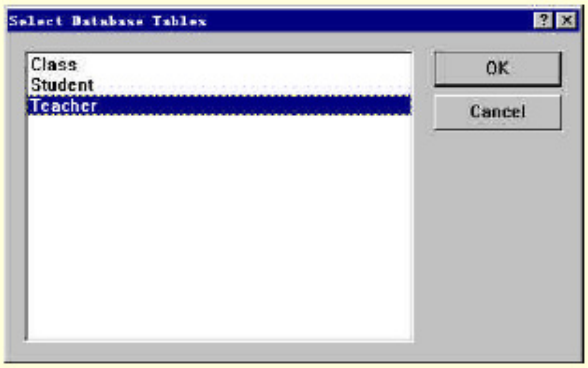

图 9-9 从数据源中选择要使用的表

（7）接受缺省值 No Compound Document Support，单击 Next 按钮。

（8）在 Step 4 对话框中，因为没有用到打印功能，我们可以选择关闭 Printing and Print Preview 选项，单击 Next 按钮。

（9）Step 5 中，我们接受缺省值，单击 Next 按钮。

（10）在 Step 6 中单击 Finish 按钮就完成了创建 School 应用程序框架所做的选择。随后出现 New Project Information 对话框，单击 OK 按钮，AppWizard 根据我们作出选择创建 School 应用程序框架。

创建完应用程序框架后，编译运行该程序，会出现图 9-10 所示的窗口。可以使用程序工具栏上的数据库控件浏览 Teacher 表的记录，工具栏的四个按钮分别表示移动到首条记录，移动到前一条记录，移动到下一条记录，移动到最后一条记录。

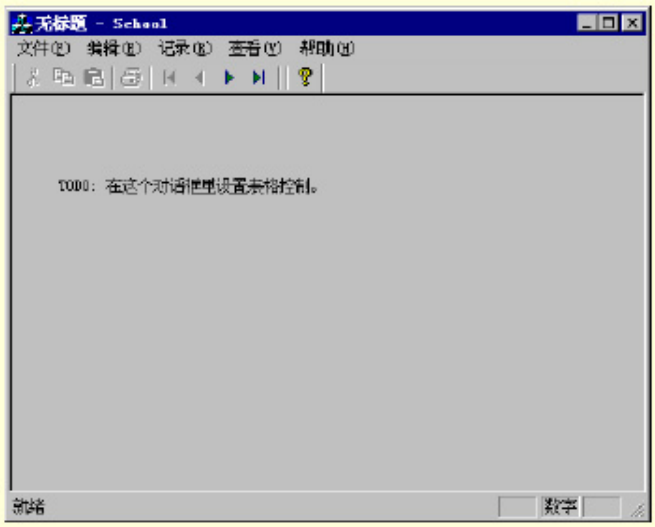

图 9-10 应用程序 School 框架的外观

注意：

> 但因为没有在控件和表中要显示的字段间建立联系，在屏幕上看不到任何信息。下一节中，我们将完成这一关联操作。

9.2.3 为数据库应用程序创建视图

上面创建的应用程序可以浏览 Teacher 表中的记录，但在屏幕上看不到显示信息，这是因为在视图中没有用于显示记录信息的控件，为看到表中的记录，必须要向视图添加用于显示信息的控件。添加控件分为两步：

（1）在资源编辑器中创建和编辑控件。

（2）用 ClassWizard 关联控件和对应的成员变量。

1. 设计新的视图

选择 Resource View 标签 Dialog 文件夹中的 IDD_SCHOOL_FORM，从对话框中删除 TODO 静态字符串，

再加入新的静态文本和编辑框，新的视图如图 9-11 所示。

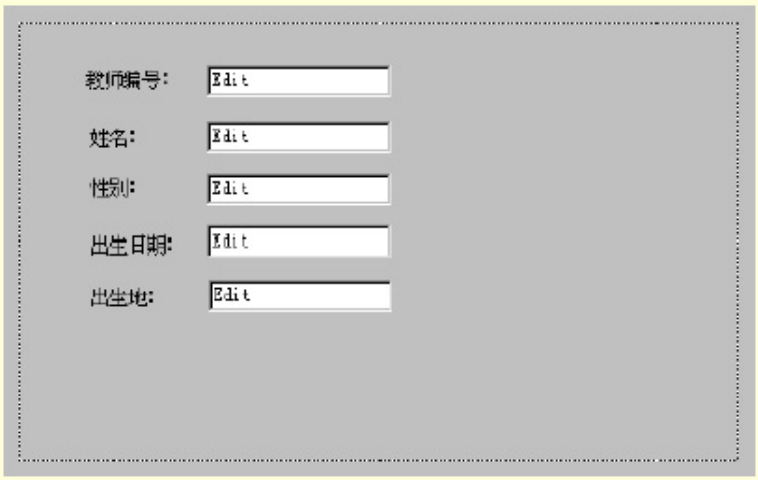

图 9-11　School View 的视图

这些编辑框的对象 ID 如表 9-4 所示。

表 9-4　编辑框的 ID

编辑框	标识符 ID
教师编号	IDC_TEACHERID
姓名	IDC_NAME
性别	IDC_SEX
出生日期	IDC_BIRTHDATE
出生地	IDC_BIRTHPLACE

2. 关联控件和成员变量

为关联编辑框和成员变量：

（1）选择 View 菜单下的 ClassWizard，单击 Member Variable 标签。

（2）选择 IDC_TEACHERID，单击 Add Variable 按钮出现 Add Member Variable 对话框。

（3）单击 Member Variable Name 下方的箭头，在下拉列表中选择 m_pSet->m_TeacherID，如图 9-12 所示。

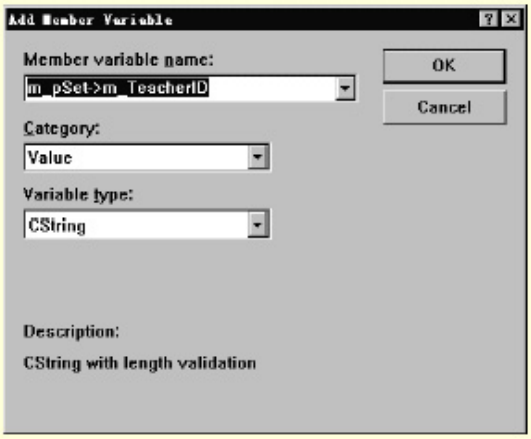

图 9-12　关联编辑框和成员变量

（4）按同样的方法将其它编辑框绑定到对应的成员变量上，结果如图 9-13 所示。

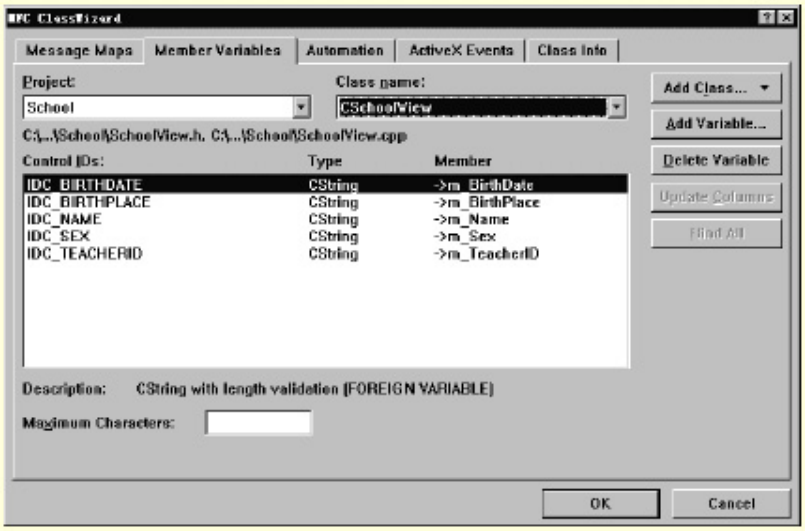

图 9-13　所有编辑框绑定到对应的成员变量上

（5）单击 ClassWizard 的 OK 按钮，完成修改。

3．创建和运行程序

用 Build 选项创建这个修改后的应用程序，再用 Execute 选项运行这个程序，新的程序应该如图 9-14 所示。

现在，我们就可以用数据库控件浏览数据表中的记录。这个程序同时还有修改记录的功能，在编辑框中修改列值后再滚动记录，修改后的值就传递到数据表中了。修改记录的功能是 AppWizard 生成的程序自动提供的，但要想添加记录，删除记录，对记录进行排序和筛选就得用户自己去添加代码了。

图 9-14　应用程序 School 的外观

9.2.4　应用程序是如何工作的

我们应该注意数据库应用程序 School 的两个类：CSchoolSet 类和 CSchoolView 类。

CSchoolSet 类是 CRecordset 类的派生类。CRecordset 类提供在数据表中滚动，添加，修改和删除记录的能力，通常我们需要为要使用的每一个数据表创建一个从 CRecordset 类派生出来的记录集类，然后可以通过这个记录集类的对象操作对应的数据表。CSchoolSet 类的定义和实现如程序清单 9-1，9-2 所示。

程序清单 9-1　CSchoolSet 类的定义

```
class CSchoolSet : public CRecordset
{
public:
```

```
CSchoolSet(CDatabase* pDatabase = NULL);
DECLARE_DYNAMIC(CSchoolSet)

// Field/Param Data
//{{AFX_FIELD(CSchoolSet, CRecordset)
CString     m_TeacherID;
CString     m_Name;
CString m_Sex;
CString     m_BirthDate;
CString     m_BirthPlace;
//}}AFX_FIELD

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CSchoolSet)
public:
virtual CString GetDefaultConnect();    // Default connection string
virtual CString GetDefaultSQL();  // default SQL for Recordset
virtual void DoFieldExchange(CFieldExchange* pFX);    // RFX support
//}}AFX_VIRTUAL

// Implementation
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif

};
```

程序清单 9-2　CSchoolSet 的实现

```
CSchoolSet::CSchoolSet(CDatabase* pdb)
: CRecordset(pdb)
{
//{{AFX_FIELD_INIT(CSchoolSet)
m_TeacherID = _T("");
m_Name = _T("");
m_Sex = _T("");
m_BirthDate =_T("");
m_BirthPlace = _T("");
m_nFields = 5;
//}}AFX_FIELD_INIT
m_nDefaultType = snapshot;
}

CString CSchoolSet::GetDefaultConnect()
```

```
{
return _T("ODBC;DSN=School");
}

CString CSchoolSet::GetDefaultSQL()
{
return _T("[Teacher]");
}

void CSchoolSet::DoFieldExchange(CFieldExchange* pFX)
{
//{{AFX_FIELD_MAP(CSchoolSet)
pFX->SetFieldType(CFieldExchange::outputColumn);
RFX_Text(pFX, _T("[TeacherID]"), m_TeacherID);
RFX_Text(pFX, _T("[Name]"), m_Name);
RFX_Text(pFX, _T("[Sex]"), m_Sex);
RFX_Text(pFX, _T("[BirthDate]"), m_BirthDate);
RFX_Text(pFX, _T("[BirthPlace]"), m_BirthPlace);
//}}AFX_FIELD_MAP
}

/////////////////////////////////////////////////////////////////////////////
// CSchoolSet diagnostics

#ifdef _DEBUG
void CSchoolSet::AssertValid() const
{
CRecordset::AssertValid();
}

void CSchoolSet::Dump(CDumpContext& dc) const
{
CRecordset::Dump(dc);
}
#endif //_DEBUG
```

在 AFX_FIELD **块中可以看到** CSchoolSet **类的成员变量，每一个成员变量对应数据表** School **的一列，这些成员变量就是与显示控件相关联的变量。**

CSchoolSet **类的构造函数主要完成成员变量的初始化。其中** m_nFields **代表相应数据表的字段的数目，**m_nDefaultType **代表记录集的类型，记录集有三种类型：**

- CRecordset::snapshot——**允许前滚和后滚，对记录集的修改直到重新打开或重新查询记录集时才生效，这是缺省设置。**
- CRecordset::dynaset——**允许前滚和后滚，对记录集的修改立即生效。**
- CRecordset::forwardOnly——**只允许前滚，记录集是只读的。**

DoFieldExchange()**函数类似于对话框中的数据传递函数，用于在记录集的成员变量和** ODBC **数据源之间传递数据。读取记录时，该函数使用记录字段交换宏** RFX **把数据从数据源中传递到记录集的成员变量中；添加和修改记录时，该函数把数据从记录集的成员变量传递到数据源中 。记录字段交换宏很多，如表** 9-5 **所示。**

表 9-5 记录字段交换宏 RFX

RFX 宏	功 能
RFX_Binary()	传递 CByteArray 类型的字节数组
RFX_Bool ()	传递布尔数据
RFX_Byte ()	传递字节数据
RFX_Date ()	使用 CTime 或 TIMESTAMP_STRUCT 传递时间和日期数据
RFX_Double()	传递双精度浮点数据
RFX_Int ()	传递整型数据
RFX_Long()	传递长整型数据
RFX_LongBinary()	通过 CLongBinary 类对象传递长二进制对象
RFX_Single()	传递浮点数据
RFX_Text ()	传递字符串数据

每个 RFX 宏有三个参数：传向 DoFieldExchange()函数的 CFieldExchange 指针 pFX，数据表的列名，存放列值的成员变量。

了解它们的含义后，在需要改变记录集的成员变量时可以直接修改源文件，而不必先删除该类，然后再用 ClassWizard 重新生成。

CSchoolSet 类还有一些虚函数：

GetDefaultConnect()函数返回数据源的名称，字符串中的“ODBC；”表示数据源是一个 ODBC 数据源，“DSN=School；”表示数据源的名称是“School”,其中“DSN”是指 Data Source Name（数据源名称）。

GetDefaultSQL()函数返回记录集使用的缺省 SQL 语句，通常是数据表的名称。

在需要的时候，我们可以修改 GetDefaultConnect()函数和 GetDefaultSQL()函数以改变它们的返回值，这将会在很多地方用到，如统计函数计算，多表连接。

CSchoolView 类是 CRecordView 类的派生类，它包含一个指向 CSchoolSet 对象的指针，该 CSchoolSet 类对象来自于 CSchoolDoc 文档类的成员变量。CRecordView 类是 CFormView 类的派生类，它增加了几个新函数，其中，OnGetRecordSet()函数年返回与该视图连接的 CRecordSet 类对象，IsOnFirstRecord()函数和 IsOnLastRecord()函数将分别在当前记录位于第一条记录和最后一条记录时返回，OnMove()函数在滚动数据记录时被调用。

程序运行后，将自动创建 CSchoolDoc 对象和 CSchoolView 对象，CSchoolView 对象的 OnInitialUpdate()函数会得到指向 CSchoolDoc 对象的 CSchoolSet 成员对象的指针，并打开 CSchoolSet 成员对象，把第一条记录传递到 CSchoolSet 对象的成员变量中，然后调用 CSchoolView 对象的 DoDataExchange()函数更新屏幕的输出。

OnMove()函数在滚动数据记录时被调用，它把当前记录传递到记录集的成员变量中，再调用 DoDataExchange()函数更新屏幕的输出。

9.2.5 遍历、添加、修改和删除记录

1．数据表的遍历

要遍历数据库，首先要用 CRecordSet 类的 Open()函数打开数据表。前面的 School 应用程序的数据表 Teacher 是由 AppWizard 自动生成的，并由 CDocument 和 CRecordView 类负责打开。如果用户用 ClassWizard 创建了新的记录集，那就需要自己打开记录集了。打开记录集后，就可以用 CRecordSet 类的成员函数遍历数据表，主要使用的成员函数为：IsBOF()，IsEOF()，MoveFirst()，MoveLast()，MoveNext()，MovePrev()，Move(long lRows)。其中，IsBOF()在移到了第一条记录之前或记录集中没有记录时返回值为真，IsEOF()在移到了最后一条记录之后或记录集中没有记录时返回值为真，它们用于滚动记录前后的判断。使用完记录集后，还要记住要关闭该记录集，否则它们将一直占用系统的资源。

程序清单 9-3 是遍历数据表的例程，可以计算数据表中记录的数目。

程序清单 9-3 遍历数据表的例程

```
CSchoolSetrstSchool;
rstSchool.Open();
int nRecordNumber=0;
while(!rstSchool.IsEOF())
```

```
{
nRecordNumber++;
rstSchool.MoveNext();
}
rstSchool.Close();
```

2．添加新记录

添加新记录的步骤为：

（1）用 CRecordSet **类的成员函数** AddNew()**进入添加模式。**AddNew()**创建了一个新的空记录。**

（2）填写该记录。

（3）调用 Update()**把添加的新记录传递到数据表中。**

程序清单 9-4 **是添加新记录的例程。**

程序清单 9-4　**添加新记录的例程**

```
rstSchool.Open();
if(!rstSchool.CanAppend())
{
rstSchool.Close();
return;
}
rstSchool.AddNew();
rstSchool.m_TeacherID=”T0008”;
rstSchool.m_Name=”赵盛”;
rstSchool.m_Sex=”男”;
rstSchool.m_BirthDate=”1967-7-5”;
rstSchool.m_BirthPlace=”江苏”;
if(!rstSchool.Update())
AfxMessageBox(“添加新记录失败！”)；
rstSchool.Close();
```

上面的例程在添加新记录前还判断是否允许向数据表中添加新记录，如果不允许，就关闭记录集然后返回。

3．修改记录

修改记录的步骤为：

（1）用 CRecordSet **类的成员函数** Edit()**进入修改模式。**

（2）修改当前记录。

（3）调用 Update()**把修改后的记录传递到数据表中。**

程序清单 9-5 **是修改记录的例程。**

程序清单 9-5　**修改记录的例程**

```
rstSchool.Open();
if(!rstSchool.CanUpdate())
{       rstSchool.Close();
return;
}
rstSchool.Edit();
rstSchool.m_TeacherID=”T0008”;
rstSchool.m_Name=”赵盛”;
```

```
rstSchool.m_Sex="男";
rstSchool.m_BirthDate="1967-7-5";
rstSchool.m_BirthPlace="江苏";
if(!rstSchool.Update())
AfxMessageBox("修改记录失败！")；
rstSchool.Close();
```

上面的例程在修改记录前还判断是否允许修改记录，如果不允许，就关闭记录集然后返回。

4. 删除记录

删除记录比添加和修改记录要简单，其步骤为：

（1）移动到待删除的记录。

（2）调用 Delete()函数。

与添加和修改记录相比，不必在调用 Delete()后再调用 Update()。

程序清单 9-6 是删除记录的例程。

程序清单 9-6　删除记录的例程

```
rstSchool.Delete();
rstSchool.MoveNext();
if(rstSchool.IsEOF())
rstSchool.MoveLast();
```

上面的例程在删除记录后将记录移动到下一条记录，如果移动到最后一条记录之后，则把当前位置设置在最后一条记录上。

5. 数据库异常

程序错误分为三种：语法错误，逻辑错误和运行时错误。MFC 函数可以识别出运行时错误并产生异常事件，于是程序转去执行处理异常的代码。使用异常时，程序不必对被调用函数的返回值进行条件检查，减少了额外的检查开销，于是程序更加清晰，有更高的运行效率。

异常处理分为两部分：一个 try 块和一个或多个 catch 块。catch 的参数是异常的类型。由于一个 try 块可能产生多种不同类型的异常，所以需要一个或多个 catch 块。程序清单 9-7 是异常处理的程序结构。

程序清单 9-7　异常处理的程序结构

```
try{
程序块
}
catch(异常类型 1){
异常处理代码 1
}
catch(异常类型 2){
异常处理代码 2
}
…
catch(异常类型 n){
异常处理代码 n
}
```

CRecordSet 类产生三种不同类型的异常：

- CDBException——数据库操作时出现错误。
- CMemoryException——内存分配时出现错误。

■ CFileException——文件存取时出现错误

其中 CDBException 出现的最多。例如处理记录删除时出现的异常,例程如程序清单 9-8 所示。

程序清单 9-8 记录删除异常的处理例程

```
try{
rstSchool.Delete();
}
catch(CDBException* e){
AfxMessageBox("记录删除出错！");
AfxMessageBox(e->m_strError);
}
```

6. 记录的筛选和排序

打开记录集时，缺省的情况是选中全部记录，并按照在数据库中存储的顺序返回。但在使用数据库中，最常见的操作是检索满足给定条件的记录，有时还需要将数据库中的全部或部分记录排序。如果使用 SQL 语言，检索和排序可以表示成 SELECT 操作，要是使用 ODBC，我们该怎样做呢？

用 CRecordSet 类可以做同样的事，具体的做法是在 m_strFilter 成员变量中设置筛选条件以实现 SELECT 语句中 WHERE 子句的功能，在 m_strSort 成员变量中设置排序字段以实现 SELECT 语句中 ORDER BY 子句的功能。打开记录集后，记录的组成和顺序就是确定的，为了对记录重新筛选和排序，m_strFilter 和 m_strSort 成员变量设置新值后，还必须重新打开记录集或调用 CRecordSet 类的 Requery()方法。通常重新查询比重新打开记录集要快。

对记录筛选的例程为：

```
rstSchool.m_strFilter="StudentID=' T0002' ";
rstSchool.Requery();
```

对记录排序的例程为：

```
rstSchool.m_strSort="StudentID";
rstSchool.Requery();
```

学习了记录的遍历，添加，修改，删除，排序和筛选后，下面我们用这些功能来完善应用程序 School，并在使用中加深对这些操作的理解，同时还能够学到一些编程技巧。

■ 增加记录的添加和删除功能

School 应用程序在由 AppWizard 生成后就具有记录修改功能，在编辑框中修改列值后滚动到其他记录上，修改过的记录就被传递到数据表中，所以此处就不用为记录的修改功能添加代码了。下面介绍如何增加记录的添加和删除功能：

（1）在资源编辑器里编辑 IDR_MAINFRAME 菜单，为“记录”菜单项增加两条新的菜单命令——“添加新记录”和“删除记录”，它们的 ID 分别为 ID_RECORD_ADD 和 ID_RECORD_DELETE。如图 9-15 所示。

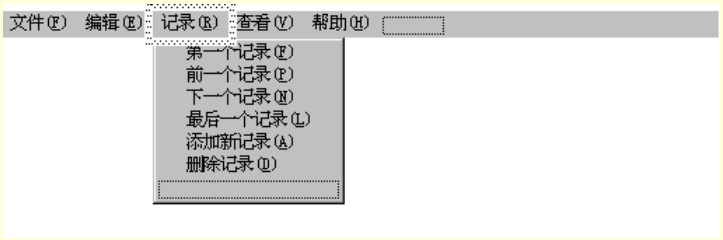

图 9-15 为 Record 菜单项增加两条新的菜单命令

（2）用 ClassWizard 为菜单命令——添加新记录 ID_RECORD_ADD 和删除记录 ID_RECORD_DELETE 填加命令响应函数。

（3）修改 CSchoolView 类的定义，在该类的 Attributes 属性部分添加代码：

```
protected:
```

```
BOOL      m_bAdd;
```

修改 CSchoolView **类的实现，在该类的构造函数中添加代码：**

```
m_bAdd=FALSE;
```

成员变量 m_bAdd **用于指示应用程序是否处于添加新记录的状态，在构造函数中初始化为** FALSE**。**

（4）编辑 OnRecordAdd()**函数，代码如程序清单** 9-9 **所示。**

程序清单 9-9　OnRecordAdd()**函数**

```
void CSchoolView::OnRecordAdd()
{
m_pSet->AddNew();
m_bAdd=TRUE;
UpdateData(FALSE);
}
```

代码中的 UpdateData(FALSE)**将空记录显示在屏幕上。**

（5）在 ClassView **中的** CSchoolView **上单击鼠标右键并选择** Add Virtual Function**，然后选择左侧列表中的** OnMove**，最后单击** Add and Edit **按钮来添加和编辑** OnMove()**函数，代码如程序清单** 9-10 **所示。**

程序清单 9-10　OnMove()**函数**

```
BOOL CSchoolView::OnMove(UINT nIDMoveCommand)
{
if(m_bAdd)
{
      m_bAdd=FALSE;
      UpdateData(TRUE);
      if(m_pSet->CanAppend())
            m_pSet->Update();
      m_pSet->Requery();
      UpdateData(FALSE);
      return TRUE;
}
return CRecordView::OnMove(nIDMoveCommand);
}
```

在添加新记录时，先用 UpdateData(TRUE)**从屏幕获取新值，然后更新记录，最后再用** UpdateData(FALSE) **显示。**

（6）编辑 OnRecordDelete()**函数，代码如程序清单** 9-11 **所示。**

程序清单 9-11　OnRecordDelete()**函数**

```
void CSchoolView::OnRecordDelete()
{
m_pSet->Delete();
m_pSet->MoveNext();
if(m_pSet->IsEOF())
{
      m_pSet->MoveLast();
}
```

```
if(m_pSet->IsBOF())
    m_pSet->SetFieldNull(NULL);
UpdateData(FALSE);
}
```

上面的例程在删除记录后将记录移动到下一条记录，如果移动到最后一条记录之后，则把当前位置设置在最后一条记录上，如果还移动到首条记录之前，意味着删除记录后记录集为空，则用 SetFieldNull(NULL)把记录的各字段置为空。

完成这些修改后，School 应用程序就具有了添加，修改和删除记录的功能了。

记录的添加要分为两步：填写空白记录和向数据库添加。所以记录的添加由两个函数 OnRecordAdd()和 OnMove()配合完成的。OnRecordAdd()函数利用 CRecordset 类的 AddNew()方法使数据库生成空白记录，进入记录添加状态，并且把成员变量 m_Add 设置为 TRUE 来表示程序处于记录添加状态，UpdateData(FALSE)是为了使视窗的显示为空白记录。OnMove()根据成员变量 m_Add 来判断应用程序是否处于记录添加状态，如果位于记录添加状态，就从屏幕上获取数据向数据库添加并更新显示，否则只是简单地移到其它记录上。

记录的删除比较简单，删除当前记录后移到下一个记录，但必须要处理两种特殊情况：删除最后一条记录和删除唯一的记录。如果删除最后一条记录，再向后移会移出最后一条记录，所以应用 MoveLast()方法把记录集定位到最后一条记录。如果删除唯一的记录，记录集为空，程序应该把当前记录的列值置为 NULL。

■ 增加记录的排序和筛选功能

下面我们再为 School 应用程序增加记录排序和筛选功能：

（1）在资源编辑器里编辑 IDR_MAINFRAME 菜单，增加排序(S)菜单项，为该菜单项增加两条菜单命令——按教师编号排序和按出生日期排序，将它们的 ID 分别设为 ID_SORT_STUDENTID 和 ID_SORT_BIRTHDATE。如图 9-16 所示。

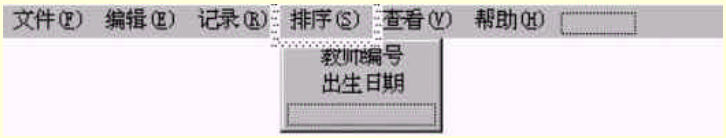

图 9-16 增加 Record 菜单项并为其增加两条菜单命令

（2）在资源编辑器里编辑 IDR_MAINFRAME 菜单，增加筛选(F)菜单项，为该菜单项增加两条菜单命令——按教师编号筛选和按姓名筛选，将它们的 ID 分别设为 ID_FILTER _STUDENTID 和 ID_FILTER_NAME。如图 9-17 所示。

图 9-17 增加 Filter 菜单项并为其增加两条菜单命令

（3）用资源编辑器为筛选参数对话框创建新的对话框资源。参数对话框的外观如图 9-18 所示，列表控件的 ID 为 ID_FLITERVALUE。

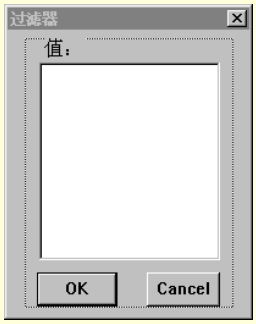

图 9-18 CFilterDlg 对话框类的资源

（4）双击对话框资源启动 ClassWizard，在 Adding a Class 对话框中选择 Create a New Class，然后单击 OK 按钮。

在 Create a New Class 对话框的 Name 文本框中输入 CFilterDlg。

单击 ClassWizard 的 Member Variables 标签，给列表控件 ID_FLITERVALUE 同时添加和关联到 CString 类型的成员变量 m_strFilterValue 和 CListBox 类型的成员变量 m_lstFilterValue 上，如图 9-19 所示。

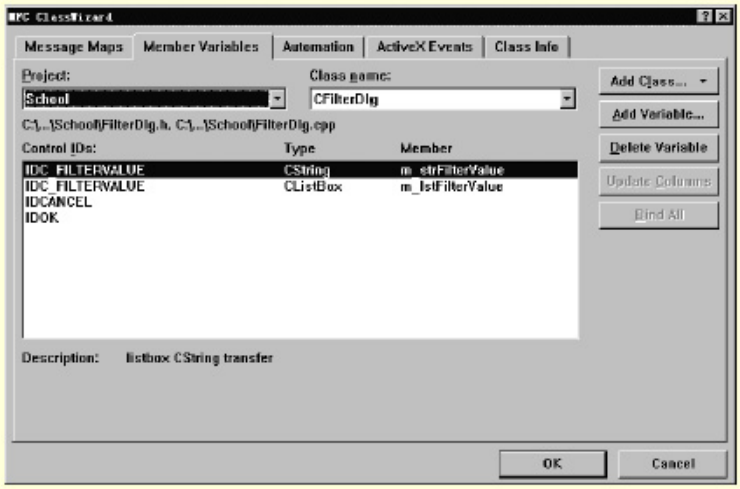

图 9-19 添加和关联成员变量

（5）给 CFilterDlg 对话框类增加成员变量，在类定义的开始部分添加代码：

```
public:
CString m_strField;
```

在该类实现的构造函数后部添加代码：

```
m_strField="TeacherID";
```

成员变量 m_strField 用于指示为哪个字段确定筛选值，在构造函数中初始化为教师编号 TeacherID 字段。

（6）重载 CFilterDlg 对话框类的 OnInitDialog()函数，按程序清单 9-12 编辑相应的代码。

程序清单 9-12 CFilterDlg 对话框类的 OnInitDialog()函数

```
BOOL CFilterDlg::OnInitDialog()
{
CDialog::OnInitDialog();

CSchoolSet rstSchool;
rstSchool.Open();
m_lstFilterValue.ResetContent();
if(m_strField=="TeacherID")
{
    while(!rstSchool.IsEOF())
    {
        m_lstFilterValue.AddString(rstSchool.m_TeacherID);
        rstSchool.MoveNext();
    }
}
if(m_strField=="Name")
{
    while(!rstSchool.IsEOF())
```

```
        {
            m_lstFilterValue.AddString(rstSchool.m_Name);
            rstSchool.MoveNext();
        }
    }
    rstSchool.Close();

    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
    }
```

(7)用 ClassWizard **为菜单命令** ID_SORT_STUDENTID**,**ID_SORT_BIRTHDATE**,**ID _FILTER_STUDENTID **和** ID_FILTER_NAME **添加命令响应函数** OnSortTeacherID()**,** OnSortBirthdate()**,** OnFilterTeacherID()**和** OnFilterName()**,按程序清单** 9-13 **编辑相应的代码。**

程序清单 9-13　**记录排序和筛选函数的代码**

```
void CSchoolView::OnSortTeacherID()
{
m_pSet->m_strSort="TeacherID";
m_pSet->Requery();
UpdateData(FALSE);
}

void CSchoolView::OnSortBirthdate()
{
m_pSet->m_strSort="BirthDate";
m_pSet->Requery();
UpdateData(FALSE);
}

void CSchoolView::OnFilterTeacherID()
{
CFilterDlg dlg;
dlg.m_strField="TeacherID";
if(dlg.DoModal()==IDOK)
{
    m_pSet->m_strFilter.Format("TeacherID='%s'",dlg.m_strFilterValue);
    m_pSet->Requery();
    UpdateData(FALSE);
    m_pSet->m_strFilter="";
}
}

void CSchoolView::OnFilterName()
{
CFilterDlg dlg;
```

```
dlg.m_strField="Name";
if(dlg.DoModal()==IDOK)
{
    m_pSet->m_strFilter.Format("Name='%s'",dlg.m_strFilterValue);
    m_pSet->Requery();
    UpdateData(FALSE);
    m_pSet->m_strFilter="";
}
}
```

9.2.6 统计函数和多表连接

1．统计函数和多表连接

在数据库的使用中，我们还经常要用到数据库的统计函数以获取数据库所有记录的统计结果。使用统计函数时，函数的运算是 DBMS 的 SQL 引擎完成的，只把结果返回给应用程序，所以执行速度很快。其中最常用的一些统计函数有：

- COUNT(*)——返回所有行的数目，空记录也包含在内。
- COUNT(表达式)——返回所有行的数目，表达式所指定的列的值为空的记录不包含在内。
- AVG(表达式)——计算平均值，表达式中的列的类型必须为数值型。
- SUM(表达式)——计算总和，表达式中的列的类型必须为数值型。
- MIN(表达式)——计算最小值，忽略空值。
- MAX(表达式)——计算最大值，忽略空值。

使用 MFC ODBC 类时，统计函数的使用方法为：

（1）创建新的记录集类。创建过程中，在 Database Options 对话框中去掉 Advanced 成组框中 Bind all columns 的选中标记。

（2）为记录集类中添加成员变量以存储统计结果。

（3）为成员变量提供相应的初始化代码和记录集字段数据传递宏。记录集字段数据传递宏的第二个参数中含有所需的统计函数。

新的记录集类中应只包含统计函数的计算，因此打开后只有一条记录。

例如，要统计 Teacher 表中记录的数目：

（1）用 ClassWizard 创建基于 CRecordset 的记录集类 CSummaryQuery。

（2）在记录集类 CSummaryQuery 中添加成员变量 m_RecordNumber。

（3）在记录集类 CSummaryQuery 的 DoFieldExchange()函数中添加记录集字段数据传递宏 RFX_Long(pFX, “ COUNT(*)”, m_RecordNumber);

2．多表的连接

在建立数据库时，我们不是将所有的数据都放在一张表中，否则就会出现大量的数据冗余，不仅浪费了存储空间，在操作数据时还会带来一大堆数据一致性问题。因此，所有规模大一些的数据库都要经过认真设计，建立多张数据表来存储数据，各张表之间存在着的联系，这种联系是通过主码和外码表示的。

因为数据存储在多张数据表中，对数据的查询就不可避免的要涉及多张数据表，也就要使用在关系数据库模型中介绍过的连接操作。连接操作通过两张表或多张表中的共同的列把这些表联系成一个整体。在 SQL 语句中，连接的表示为：

```
SELECT 列名列表
FROM 表或视图列表
WHERE 连接条件
```

例如，要确定 31 班的班主任姓名，就需要连接 Teacher 和 Class 两张数据表，相应的 SQL 语句为：

```
SELECT Teacher.Name
```

```
FROM Teacher, Class
WHERE Class.DirectorID = Teacher.TeacherID AND Class.ClassID=' 31'
```

在 VC 中，对 CRecordset 类进行一些修改后，我们也能够完成数据表的连接操作，方法为：

（1）使用 ClassWizard 建立基于 CRecordset 类的记录集类。创建过程中，在 Database Options 对话框中去掉 Advanced 成组框中 Bind all columns 的选中标记。在 Select Database Tables 对话框中选择连接中涉及到的所有数据表。

这样建立的记录集在 GetDefaultSQL()函数的返回值中包含了连接中的所有数据表。

（2）用 ClassWizard 为在查询结果中出现的列添加相应的成员变量。

（3）根据连接条件和筛选条件构造过滤字符串。

例如，使用 MFC ODBC 类来查询 31 班的班主任姓名的步骤为：

（1）使用 ClassWizard 为连接中的 Class 数据表建立基于 CRecordset 类的记录集类 CClassTeacherSet。

（2）用 ClassWizard 为 Teacher 表的 Name 列添加成员变量 m_Name。

（3）根据连接条件和筛选条件构造过滤字符串。

```
rstClassTeacher.m_strFilter = "Class.DirectorID = Teacher.TeacherID AND Class.ClassID=' 31' "
```

3. 为应用程序增加统计功能

（1）在资源编辑器里编辑 IDR_MAINFRAME 菜单，为 Record 菜单项增加菜单命令——统计结果，设置它的 ID 为 ID_RECORD_SUMMARY。

（2）接着就使用 ClassWizard 为菜单命令 ID_RECORD_SUMMARY 填加命令响应函数 OnRecordSummary()。

（3）用 ClassWizard 创建基于 CRecordset 的记录集类 CSummaryQuery。

创建过程中，在 Database Options 对话框中去掉 Advanced 成组框中 Bind all columns 的选中标记，如图 9-20 所示。在 Select Database Tables 对话框中选择 Teacher 表。

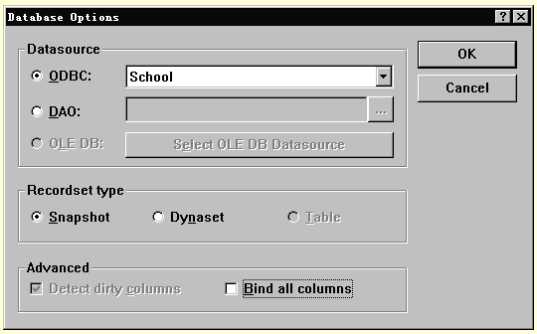

图 9-20 去掉 Bind all columns 的选中标记

（4）为记录集类 CSummaryQuery 添加成员变量 m_RecordNumber 和 m_Youngest。按程序清单 9-14 修改记录集类 CSummaryQuery：

程序清单 9-14 记录集类 CSummaryQuery 定义的修改

```
// Field/Param Data
//{{AFX_FIELD(CSummaryQuery, CRecordset)
long m_RecordNumber;
CString     m_Youngest;
//}}AFX_FIELD
```

（5）修改记录集类 CSummaryQuery 的构造函数,如程序清单 9-15 所示。

程序清单 9-15 记录集类 CSummaryQuery 构造函数的修改

```
CSummaryQuery::CSummaryQuery(CDatabase* pdb)
: CRecordset(pdb)
```

```
{
//{{AFX_FIELD_INIT(CSummaryQuery)
m_RecordNumber=0;
m_Youngest = _T("");
m_nFields = 2;
//}}AFX_FIELD_INIT
m_nDefaultType = snapshot;
}
```

（6）在记录集类 CSummaryQuery **的** DoFieldExchange()**函数中添加记录集字段数据传递宏，如程序清单** 9-16 **所示。**

程序清单 9-16　**添加记录集字段数据传递宏**

```
void CSummaryQuery::DoFieldExchange(CFieldExchange* pFX)
{
//{{AFX_FIELD_MAP(CSummaryQuery)
pFX->SetFieldType(CFieldExchange::outputColumn);
RFX_Long(pFX, _T("COUNT(*)"), m_RecordNumber);
RFX_Text(pFX, _T("MAX([BirthDate])"), m_Youngest);
//}}AFX_FIELD_MAP
}
```

在记录集字段数据传递函数的第二个参数中，我们可以看到出现了统计函数，此处用到了两个统计函数 COUNT（）**和** MAX（），**分别用于计算** Teacher **表中记录的数目和找到最年轻的教师的出生日期。**

（7）编辑 OnRecordSummary()**函数，如程序清单** 9-17 **所示。**

程序清单 9-17　OnRecordSummary()**函数**

```
void CSchoolView::OnRecordSummary()
{
CSummaryQuery rstSummaryQuery;
rstSummaryQuery.Open();
CString strPrompt;
strPrompt.Format("Teacher 表中的记录数目为：%ld \n\n 最年轻的教师出生于：%s",
rstSummaryQuery.m_RecordNumber,rstSummaryQuery.m_Youngest);
MessageBox(strPrompt,"Teacher 表的统计结果");
rstSummaryQuery.Close();
}
```

图 9-21 统计结果显示

完成这些修改后，编译运行 School 应用程序，单击 Record 菜单项的统计结果菜单命令，将会弹出一个消息框显示 Teacher 表的一些统计结果，如图 9-21 所示。

4. 为应用程序增加查询功能

（1）创建用于数据表连接的记录集 CClassTeacherSet 类

首先，使用 ClassWizard 建立基于 CRecordset 类的记录集类 CClassTeacherSet。

创建过程中，在 Database Options 对话框中去掉 Advanced 成组框中 Bind all columns 的选中标记。在 Select Database Tables 对话框中选择 Class 和 Teacher 两张表。

应该注意一下新建记录集类的 GetDefaultSQL()函数：

```
CString CClassTeacherSet::GetDefaultSQL()
{
return_T("Class,Teacher");
}
```

返回值中包含连接中的所有数据表：Teacher 和 Class 表。

查询结果中包含班主任的姓名，所以要用 ClassWizard 为 Teacher 表的 Name 列添加成员变量 m_Name。

（2）创建 CClassSet 记录集类

使用 ClassWizard 创建基于 CRecordset 类的 CClassSet 记录集类。创建过程中，在 Select Database Tables 对话框中选择 Class 数据表。我们在将要建立的 CDirectorDlg 对话框类中用到这个记录集类。

（3）建立 CDirectorDlg 对话框类

我们在该对话框中进行数据表的连接，查询和显示查询结果。

用资源编辑器为班主任对话框创建新的对话框资源 IDD_DIALOGDIRECTOR。对话框的外观如图 9-22 所示，编辑框的 ID 为 IDC_TEACHERNAME，列表控件的 ID 为 IDC_LISTCLASSID。

图 9-22 对话框资源 IDD_DIALOGDIRECTOR

双击对话框资源以启动 ClassWizard，在 Adding a Class 对话框中选择 Create a New Class，然后单击 OK 按钮。

在 Create a New Class 对话框的 Name 文本框中输入 CDirectorDlg，然后单击 OK 按钮。

单击 ClassWizard 的 Member Variables 标签，把编辑框 IDC_TEACHERNAME 关联到 CString 类型的成员变量 m_strTeacherName 上，把列表控件 IDC_LISTCLASSID 关联到 CList 类型的成员变量 m_lstClassID，如图 9-23 所示。

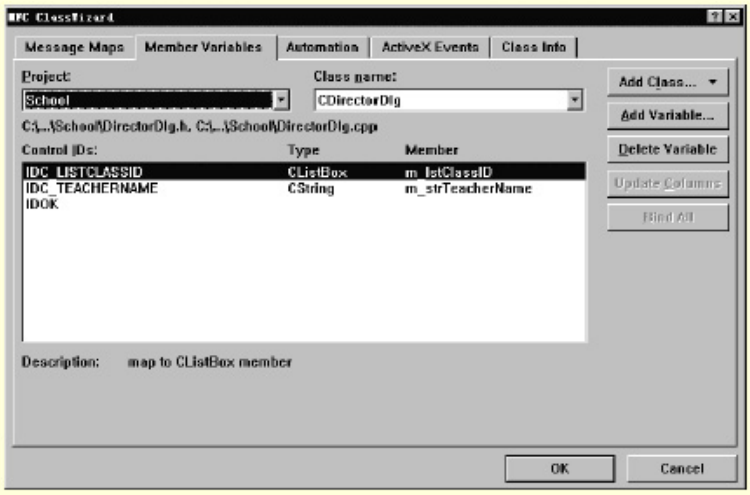

图 9-23　**为** CDirectorDlg **的控件添加成员变量**

为 CDirectorDlg **类添加和编辑** OnInitDialog ()**函数，如程序清单** 9-18 **所示。**

程序清单 9-18　OnInitDialog ()**函数**

```
BOOL CDirectorDlg::OnInitDialog()
{
CDialog::OnInitDialog();

CClassSet rstClass;
rstClass.Open();
m_lstClassID.ResetContent();
while(!rstClass.IsEOF())
{
        m_lstClassID.AddString(rstClass.m_ClassID);
        rstClass.MoveNext();
}
rstClass.Close();

return TRUE;
// return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE
}
```

这个函数根据 Class **数据表向列表框** IDC_LISTCLASSID **中添加班级编号选项。**

为 CDirectorDlg **类添加和编辑** OnSelchangeListclassid ()**函数，如程序清单** 9-19 **所示。**

程序清单 9-19　OnSelchangeListclassid ()**函数**

```
void CDirectorDlg::OnSelchangeListclassid()
{
CClassTeacherSet rstClassTeacher;
CString strClassID;
m_lstClassID.GetText(m_lstClassID.GetCurSel(),strClassID);
rstClassTeacher.m_strFilter.Format("Class.DirectorID=Teacher.TeacherIDANDClass.ClassID='%s'",strClassID);
rstClassTeacher.Open();
```

```
m_strTeacherName=rstClassTeacher.m_Name;
UpdateData(FALSE);
}
```

在 OnSelchangeListclassid ()函数中，根据连接条件和筛选条件构造过滤字符串 rstClassTeacher.m_strFilter ，查询后再将结果显示到屏幕上。

（4）添加菜单命令和响应函数

通过菜单命令，我们启动和运行 CDirectorDlg 对话框类的实例。

在资源编辑器里编辑 IDR_MAINFRAME 菜单，增加班主任菜单项，把该菜单项的 ID 设为 ID_DIRECTOR。

用 ClassWizard 为菜单命令 ID_ DIRECTOR 添加命令响应函数 OnDirector()。

编辑响应函数 OnDirector()，如程序清单 9-20 所示。

程序清单 9-20 OnDirector()函数

```
void CSchoolView::OnDirector()
{
CDirectorDlg dlg;
dlg.DoModal();
}
```

完成上述修改后，编译运行 School 应用程序，单击班主任菜单项将弹出一个查询各班班主任的对话框，在班级列表中选择班级后，教师姓名编辑框中就会显示选定班级的班主任姓名，如图 9-24 所示。

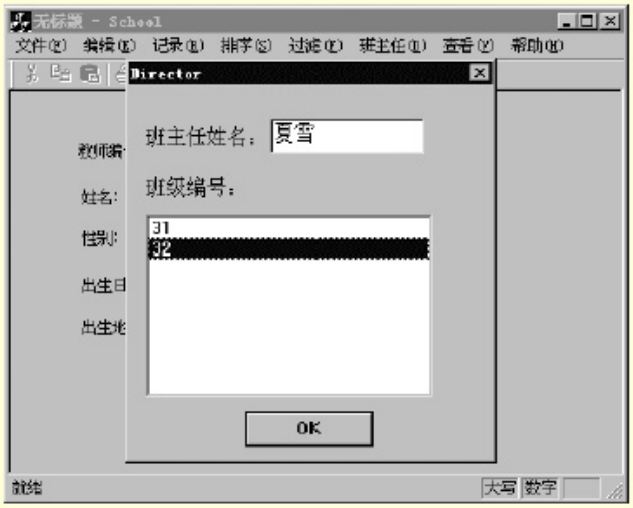

图 9-24 使用表的连接操作查询各班的班主任姓名

9.2.7 直接使用 SQL 语句

前面介绍了如何使用 CRecordset 类操作数据库，但有时直接使用 SQL 语句更有效。例如涉及多个记录的修改和删除，使用 CRecordset 类来执行就太慢了。一个较好的方法是使用 CDatabase 类的 ExecuteSQL()方法，这样的操作正好满足 ExecuteSQL()的特点——不能返回结果集，但相对于使用 CRecordset 类逐条记录进行编辑效率更高。

使用 ExecuteSQL()方法时，需要把一个包含 SQL 语句的字符串作为参数传递给 ExecuteSQL()方法。例如，把 32 班学生合并到 31 班，相应的操作如程序清单 9-21 所示。

程序清单 9-21 使用 ExecuteSQL()方法示例

```
CDatabase dbSchool;
```

```
dbSchool.Open( "School", FALSE, FALSE, "ODBC;" );
dbSchool.ExecuteSQL( “UPDATE Student SET ClassID=32 WHERE ClassID=32” );
dbSchool.Close();
```

9.2.8 使用 CDatabase 进行事务处理

由本章的 9.1.2 小节，我们知道数据库的完整性包括实体完整性和参照完整性。当数据库满足完整性规则时，又称数据库处于一致状态。为保证完整性，在插入和删除多张表的数据时，需要将某些操作作为一个整体来执行而不允许被打断，因此就出现了事务的概念。

事务允许把一些 SQL 语句作为一个整体来执行，如果事务中的一个 SQL 语句出错，整个事务都会失败，并回到事务执行前的状态。

事务具有原子性，一致性，隔离性和持久性。原子性是指事务作为一个操作单元，或者全部完成，或者全部不做。一致性是指事务将数据库从一个一致状态转换到另一个一致状态。隔离性是指一个正在执行的事务在结束前不会让其它事务看到自己的运行结果。持久性是指当事务完成后，它对数据库的操作结果是永久的，不能再被擦除。

使用 MFC ODBC 类时，为开始一个事务，应该调用 CDatabase::BeginTrans()，然后调用 CDatabase::ExecuteSQL()执行这个事务，或者使用从该 CDatabase 产生的 CRecordset 对象执行这个事务。事务的结束有两种：当所有的操作成功时，用 CDatabase::CommitTrans()提交事务；当发生错误时，用 CDatabase::RollBack()放弃事务，并回到事务执行前的状态。

程序清单 9-22 是使用 CDatabase 进行事务处理的一个例程，它向 School 数据库的 Teacher 表和 Class 表同时添加记录。

程序清单 9-22 事务处理例程

```
CDatabase *pDB;
pDB->Open( "School", FALSE, FALSE, "ODBC;" );
try{
if(pDB->BeginTrans())
    TRACE(“Transaction started\n”);
else
    TRACE(“Error in starting transaction.\n”);
pDB->ExecuteSQL(“INSERT INTO Teacher VALUES (‘ T0005’ , ‘ 顾云’ , ‘ 女’ , ‘ 1966.6’ ,‘ 广西’ )” );
pDB->ExecuteSQL(“INSERT INTO Class VALUES (‘ 33’ , ‘ T0005’ , ‘ 28’ , ’ 86.1’ )” );
if(pDB->CommitTrans())
    TRACE(“Transaction commited\n”);
else
    TRACE(“Error in committing transaction.\n”);
}
catch (CDBException *e)
{
TRACE(e->m_strError);
if(pDB->RollBack())
    TRACE (“Transaction rolled back\n”);
else
    TRACE(“Error in rolling back transaction.\n”);
}
pDB->Close();
delete pDB
```

9.3 使用 DAO

本节我们讲述讲述了 DAO 是什么，如何使用 VC++和 MFC 建立工作在 DAO 数据库上的应用程序。

9.3.1 DAO 概述

DAO 给出了一个通过编程创建和操作数据库的框架，是使用 Microsoft Jet 数据库引擎访问数据和数据结构的对象的有层次的集合，可以访问的数据源包括：

- Microsoft Jet (.MDB) 数据库。
- ODBC 数据源，使用 ODBC 驱动程序。
- 可安装的 ISAM 数据库，如数据库引擎可直接读取的 dBASE®, Paradox™和 Microsoft FoxPro 。

DAO 数据库使用的数据库引擎与 Microsoft Visual Basic®and Microsoft Access 相同，MFC DAO 类和其它应用程序框架类一起提供了访问 DAO 数据库的方便的方法。同时，MFC DAO 类也能访问具有 ODBC 驱动程序的 ODBC 数据库。

DAO 很大程度上是 ODBC 的超集，它包含 ODBC 类的大部分功能，并且又增加了很多新功能。ODBC 和 DAO 实现函数库的方式有所不同，ODBC 用一组 DLL 来实现，而 DAO 用 OLE 对象来实现。尽管 DAO 用 OLE 对象来实现，但 MFC DAO 类封装了所有细节，给用户提供了与 OLE 对象交互作用的数据和函数成员。

DAO 和 ODBC 都使用数据库类对象提供与数据库的连接，使用记录集对象保存结果集。DAO 中的数据库类为 CDaoDatabase，记录集类为 CDaoRecordset，它们包含了与 ODBC 类相似的成员变量和成员函数，同时还增加了很多新的成员变量和成员函数。使用 DAO 数据库的应用程序至少要用到 CDaoDatabase 对象和 CDaoRecordset 对象。

9.3.2 MFC DAO 类

MFC DAO 类的层次结构如图 9-25 所示：

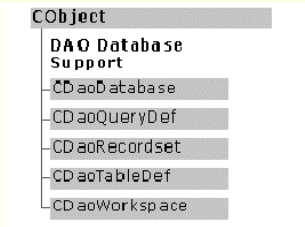

图 9-25 MFC DAO 类的层次结构

- CDaoWorkspace

在登录和退出这段时间中管理一个命名过的，有口令保护的数据库会话，大多数程序使用缺省工作空间。如果需要，可以创建工作空间对象，并同时运行多个会话。每个工作空间对象在它的数据库集合中可以包含多个打开的数据库对象。在 MFC 中，工作空间主要是事务管理器，在同一个事物空间中指定一个打开的数据库集合。

- CDaoDatabase

表示与数据库的连接，通过这个连接可以对数据库中的数据进行操作。在工作空间中同时可以有多个活动的 CDaoDatabase 对象，工作空间维护着打开的数据库对象的集合。

- CDaoRecordset

表示查询结果集。有三种类型：

表类型的记录集——表示一个基表，可用于从一张数据表中检查，增加，修改或删除记录。

动态集类型的记录集——是记录可以更新的查询结果。可以用来从底层数据库的一张或多张数据表中检查，修改或删除记录。动态集类型的记录集可以包含多张数据表中的字段。

快照类型的记录集——是记录集的静态拷贝，用于寻找数据或产生报表。可以包含一个数据库中多张表的

字段，但它们不能被更新。

- CdaoQueryDef：表示查询定义，通常保存在数据库中。
- CdaoTableDef：表示基表和绑缚表的定义

此外，还有两个没有在图 9-25 中列出的 MFC DAO 类：

- CdaoRecordView：提供显示和编辑数据记录的用户界面。
- CdaoException：表示由 DAO 类引发的异常

9.3.3 创建 DAO 数据库应用程序

DAO 很大程度上是 ODBC 的超集，它包含 ODBC 类的大部分功能，并且又增加了很多新功能。使用 AppWizard 创建 DAO 数据库应用程序同创建 ODBC 应用程序很相似。

在本章第二节，我们创建了使用 ODBC 类的数据库应用程序，现在我们用 DAO 类来实现相同的功能。

1．创建应用程序框架

创建程序的步骤如下，完成了这些步骤后，将会得到一个能浏览 School 数据库 Teacher 表的应用程序。

（1）从 Developer Studio 菜单栏中选择 File 菜单下的 New，单击 Project 标签。

（2）选择 MFC AppWizard (exe)并在文本框中输入 DAOSchool，单击 OK 按钮。

（3）应用程序是一个简单的单文档界面应用程序，所以选择 Single Document。

（4）该应用程序是为了浏览数据库，不需要文件支持，选择 Database View Without File Support 选项，AppWizard 将生成浏览数据库内容的类。再单击该对话框中的 Data Source 按钮，我们将选择该应用程序使用的的数据库。

（5）在 Database Options 对话框中，选中 DAO，单击标有“…”的按钮，在文件打开对话框中选择数据源 School，如图 9-26，打开该数据库文件后。

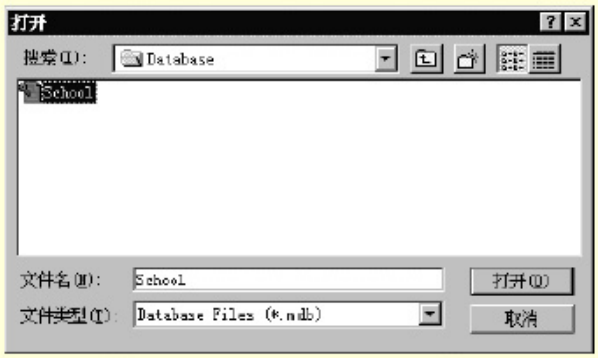

图 9-26 从数据源中选择要使用的表

然后如图 9-27 所示，单击 OK 按钮。

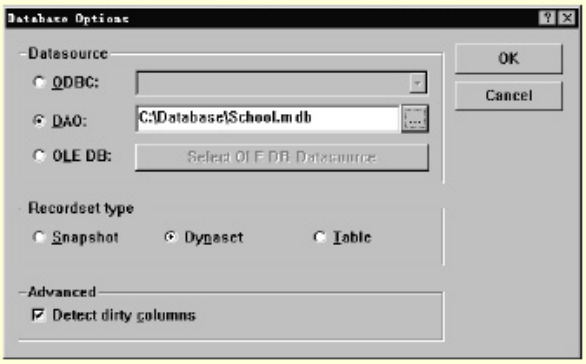

图 9-27 选择 School 为该应用程序连接的 ODBC 数据源

（6）在 Select Database Tables 对话框中，选择 Teacher 表，单击 OK 按钮，现在，我们在数据源 School 中的 Teacher 表和应用程序之间建立了连接。然后，Step 2 对话框出现，单击 Next 按钮转到 Step 3。

（7）接受缺省值 No Compound Document Support，单击 Next 按钮。

（8）在 Step 4 对话框中可以对程序的外观进行选择，因为不需要打印功能，选择关闭 Printing and Print Preview 选项，单击 Next 按钮。

（9）Step 5 中，我们接受缺省值，单击 Next 按钮。

（10）在 Step 6 中单击 Finish 按钮就完成了创建 School 应用程序所做的选择。随后出现 New Project Information 对话框，单击 OK 按钮，AppWizard 创建 School 应用程序。

创建完应用程序后，编译运行该程序，将出现图 9-28 所示的窗口，这与使用 ODBC 类的应用程序框架外观一致。可以用应用程序工具栏上的数据库控件从 Teacher 表的一个记录转到另一个记录，但因为没有在控件和表中要显示的字段间建立联系，在屏幕上看不到任何信息。在下一小节，我们将完成这一关联操作。

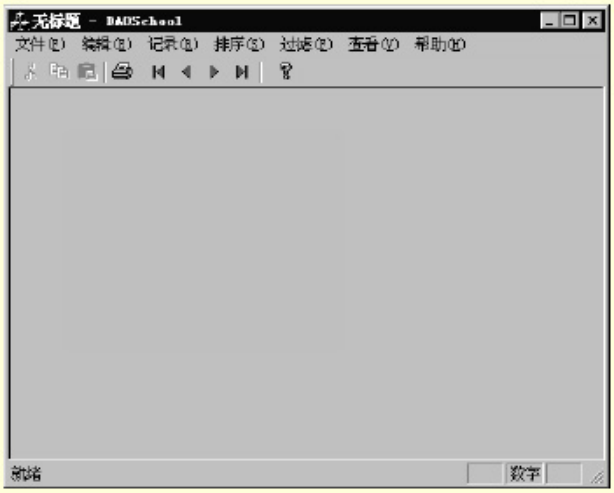

图 9-28 应用程序 DAOSchool 框架的外观

2．为数据库应用程序创建视图

为使上面创建的应用程序可以在屏幕上显示信息，必须要向视图中添加控件，可以分为两步：

（1）设计新的视图

选择 Resource View 标签 Dialog 文件夹中的 IDD_DAOSCHOOL_FORM，从对话框中删除 TODO 静态字符串，再加入新的静态文本和编辑框，新的视图如图 9-29。

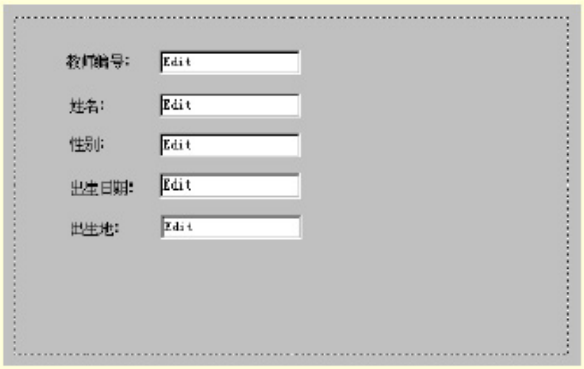

图 9-29 School View 的视图

这些编辑框的对象 ID 如表 9-6 所示。

表 9-6 编辑框的 ID

编辑框	标识符 ID
教师编号	IDC_TEACHERID
姓名	IDC_NAME
性别	IDC_SEX
出生日期	IDC_BIRTHDATE
出生地	IDC_BIRTHPLACE

（2）关联编辑框和成员变量

用 ClassWizard 为对话框的编辑框关联记录集的成员变量，结果如图 9-30 所示。

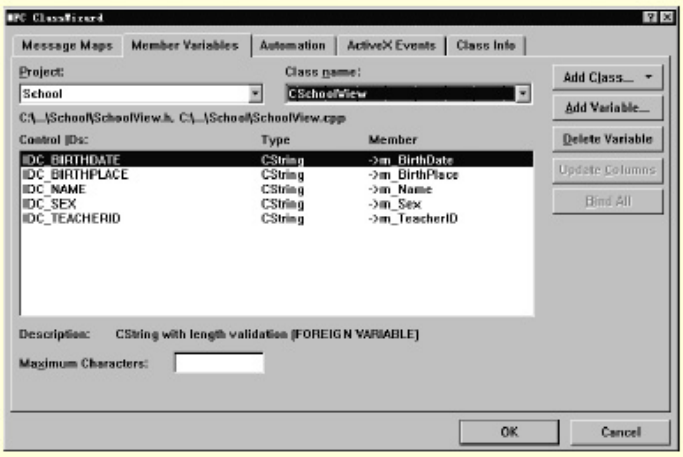

图 9-30　所有编辑框绑定到对应的成员变量上

3. 创建和运行程序

编译运行该应用程序，新的程序应该如图 9-31 所示。现在，我们就可以用数据库控件浏览数据表中的记录。这个程序同时还有修改记录的功能，在编辑框中修改列值后再滚动记录，修改后的值就传递到数据表中了。修改记录的功能是 AppWizard 生成的程序自动提供的，但要想添加记录，删除记录，对记录进行排序和筛选就得用户自己去添加代码了。

图 9-31　应用程序 DAOSchool 的外观

9.3.4　理解派生的记录集类

CDAOSchoolSet 类是 CDAORecordset 类的派生类，它提供在数据表中滚动、添加、更新和删除记录的能力，通常我们需要为要使用的每一个数据表创建一个从 CDAORecordset 类派生出来的类。CDAOSchoolSet 类的定义和实现如程序清单 9-23 和 9-24 所示。

程序清单 9-23　CDAOSchoolSet 类的定义

```
class CDAOSchoolSet : public CDaoRecordset
{
public:
CDAOSchoolSet(CDaoDatabase* pDatabase = NULL);
DECLARE_DYNAMIC(CDAOSchoolSet)

// Field/Param Data
//{{AFX_FIELD(CDAOSchoolSet, CDaoRecordset)
```

```
CString     m_TeacherID;
CString     m_Name;
CString     m_Sex;
CString     m_BirthDate;
CString     m_BirthPlace;
//}}AFX_FIELD

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDAOSchoolSet)
public:
virtual CString GetDefaultDBName();          // REVIEW:    Get a comment here
virtual CString GetDefaultSQL();  // default SQL for Recordset
virtual void DoFieldExchange(CDaoFieldExchange* pFX);       // RFX support
//}}AFX_VIRTUAL

// Implementation
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif

};
```

程序清单 9-24　CDAOSchoolSet 的实现

```
CDAOSchoolSet::CDAOSchoolSet(CDaoDatabase* pdb)
: CDaoRecordset(pdb)
{
//{{AFX_FIELD_INIT(CDAOSchoolSet)
m_TeacherID = _T("");
m_Name = _T("");
m_Sex = _T("");
m_BirthDate = _T("");
m_BirthPlace = _T("");
m_nFields = 5;
//}}AFX_FIELD_INIT
m_nDefaultType = dbOpenDynaset;
}

CString CDAOSchoolSet::GetDefaultDBName()
{
return _T("C:\\Database\\School.mdb");
}

CString CDAOSchoolSet::GetDefaultSQL()
```

```
{
return _T("[Teacher]");
}

void CDAOSchoolSet::DoFieldExchange(CDaoFieldExchange* pFX)
{
//{{AFX_FIELD_MAP(CDAOSchoolSet)
pFX->SetFieldType(CDaoFieldExchange::outputColumn);
DFX_Text(pFX, _T("[TeacherID]"), m_TeacherID);
DFX_Text(pFX, _T("[Name]"), m_Name);
DFX_Text(pFX, _T("[Sex]"), m_Sex);
DFX_Text(pFX, _T("[BirthDate]"), m_BirthDate);
DFX_Text(pFX, _T("[BirthPlace]"), m_BirthPlace);
//}}AFX_FIELD_MAP
}

/////////////////////////////////////////////////////////////////////////////
// CDAOSchoolSet diagnostics

#ifdef _DEBUG
void CDAOSchoolSet::AssertValid() const
{
CDaoRecordset::AssertValid();
}

void CDAOSchoolSet::Dump(CDumpContext& dc) const
{
CDaoRecordset::Dump(dc);
}
endif //_DEBUG
```

在 AFX_FIELD 块中可以看到 CDAOSchoolSet 类的成员变量，每一个变量对应数据表 Teacher 的一列。

CDAOSchoolSet 类的构造函数主要进行成员变量的初始化。其中 m_nFields 代表相应数据表的列的数目，m_nDefaultType 代表打开记录集的方式，缺省为动态集方式。

DoFieldExchange()函数使用记录字段交换宏 DFX 在数据源和记录集的成员变量间交换数据 。每个 DFX 宏有三个参数：传向 DoFieldExchange()函数的 CFieldExchange 指针 pFX，数据表的列名，存放列值的成员变量。

CDAOSchoolSet 类还有一些虚函数：

- GetDefaultConnect()函数返回数据源的路径。
- GetDefaultSQL()函数返回记录集使用的缺省 SQL 语句，通常是表的名称。

从 9.3.2 和 9.3.3 小节，我们可以看到由于 DAO 类很大程度上是 ODBC 类的超集，它包含 ODBC 类的大部分功能，因此将使用 ODBC 类的应用程序转换到使用 DAO 类的应用程序所需的改动是极少的。仿照 9.2.12 中遍历，添加，修改和删除记录的实现方法和 9.2.15 中统计函数和多表连接的实现方法，我们可以用这些功能来进一步完善 DaoSchool 应用程序。

9.4 其它数据库技术简介

9.4.1 OLE DB 技术

OLE DB 是通过 OLE 组件对象模型 (COM)访问数据的一组接口。OLE DB 接口为应用程序访问存储在各种信息源中的数据提供统一的访问接口，这些接口为数据源提供了足够的 DBMS 功能，以便数据共享。

OLE DB 还是一个数据库体系结构，能够忽略数据类型的差异在企业网中提供从大型主机到台式机的统一的数据集成。因为可以访问更多类型的数据，并且基于组件对象模型 (COM)，OLE DB 是比 ODBC 更通用更有效的访问数据的方法。

OLE DB 提供:

- 通过标准接口集合访问各种类型数据的能力。
- 用户的应用程序，控件，数据提供者和各种数据源（包括 DBMS 数据）间的无缝集成。
- 服务提供者的即插即用，如查询处理器和光标引擎。使用了 OLE DB 的查询处理器可以集成在数据访问环境中为各种应用程序和数据提供者服务。

微软的 OLE DB 简单提供者工具箱允许开发人员在简单数据源上使用核心的 OLE DB 功能，从而不需要花费时间从草图阶段开始建立完整的 OLE DB 提供者。使用这个工具箱，不仅可以节省开发时间，还可以在更高的层次上建立一个实现数据访问策略的基础。

用 OLE DB 编程是一个较高深的话题，有兴趣进一步深入的用户可以通过联机文档 OLE DB Programmer's Reference 来学习。

9.4.2 ADO 技术

ActiveX 数据对象(ADO)是 OLE DB 的使用者，它通过向提供者发出访问数据源的请求来获取所需的信息。在 ADO 中提供者的名称作为连接字符串的一部分或者作为连接对象的提供者属性，这个步骤使得连接依赖于简单提供者是否已经作为完全的 OLE DB 提供者注册过。

ADO 允许编写客户端应用程序，客户程序通过提供者去访问和操作数据源中的数据，特别适合于 OLE DB 提供者可以访问的数据。ADO 的主要优点是易于使用、高速、低内存费用并且只占用很少的磁盘空间。

使用了 ADO 工具箱后，就在更高的层次上建立了一个实现柔性数据访问策略的基础。例如，将 ADO 应用程序编程的易用性与使用简单提供者工具箱开发提供者的易用性结合，可以快速实现一个端到端的单层或多层应用程序，以用于公司、内部网、因特网或企业级的数据访问。

9.5 本章小结

Visual C++ 提供了多种访问数据库的方法：直接调用 DAO 和 ODBC 软件开发包(SDKs)中的 API 函数；选择使用 MFC，让 MFC DAO 类和 MFC ODBC 类简化各自 API 的使用；使用 OLE DB 或 ADO。由于 MFC 类库强大的封装作用，我们不必再深入了解应用编程接口 API 的大量函数的细节及其使用方法，AppWizard 能为我们生成一个很好的框架，在此基础上，再使用数据库类来实现所要求的功能，从而大大减轻了编程的强度。

ODBC 类可以处理大部分数据库的编程，但 DAO 更新，更适合访问用 Microsoft Access 创建的.MDB 数据库。一般来说，操作 Microsoft Jet (.MDB)数据库时使用 MFC DAO 类（DAO 也可以访问外部数据库，如 ODBC 数据源）。当不想使用 Microsoft Jet 数据库引擎，而是想使用 ODBC API 来实现与数据源之间完全的独立性，应该使用 MFC ODBC 类。

OLE DB 和 ADO 是访问数据库的新的手段，由于篇幅所限，不能详细介绍，想深入学习的用户可以通过联机文档来学习。

第十章 Internet 编程

Internet/Intranet 是一项飞速发展的技术，特别是现在，随着环球网（WWW——World Wide Web）的推广，Internet/Intranet 已经成为了当前最热门的话题之一。用户开发的程序或许就需要支持 Internet 访问，本章就来介绍 MFC 中与 Internet 有关的一些内容。

Microsoft 提供了大量的 API 函数以支持用户开发 Internet 服务器或者客户端应用程序。客户端的应用程序需要提供访问 Internet 的能力，而服务器端的应用程序需要支持 HTTP、FTP 或 Gopher 等类型的服务。

就个人计算机用户而言，更重要的是客户端的应用程序。因此在本章中主要介绍客户端应用程序的实现，如果需要了解如何开发服务器端应用程序，请参考有关专著，或者直接从 VC6.0 的联机帮助中获取信息。

直接使用 API 函数开发 Internet 程序是令人烦恼的，MFC 对其中的一部分进行了封装。使用 MFC 进行开发还可以进一步分为几种类型：

■ 使用 Windows 插口：Winsock 标准定义了一个 DLL 接口，MFC 使用 CAsyncSocket 和 CSocket 类对插口进行了封装。即使是使用 MFC 提供的类，直接通过插口实现 Internet 访问也是困难的。

■ 使用消息收发 API（MAPI——Message API）：使用 MAPI 可以方便地向其他应用程序发送电子邮件、语音邮件或者传真。AppWizard 为应用程序获得 MAPI 支持提供了捷径，用户只需要在 AppWizard 的第四步中选中“ Message API ”选项即可。

■ 使用 WinInet 类：MFC 建立了许多新的类免除用户学习插口编程之苦，这些新的 WinInet 类能够帮助用户建立客户端的应用程序。WinInet 类支持 HTTP、FTP 和 Gopher 等标准的协议。

■ 使用 ISAPI 类（ISAPI——Internet Server API）：ISAPI 类帮助用户扩展 HTTP 服务器的功能。ISAPI 可以用来创建 Internet 服务器扩展器和过滤器。用户可以通过使用 ISAPI Extension Wizard 建立扩展器或者过滤器的框架。

本章中主要介绍如何使用 MFC 提供的 WinInet 类完成客户端的应用程序。

10.1 使用 WinInet 类

使用 WinInet 类使得用户可以在相当高级的层次上实现对 Internet 的访问，根本不需要考虑有关插口和协议的详细内容，MFC 包办了大部分的工作。

WinInet 类只是一个总称，包括大约 10 个具体的类。其中最重要的是 CInternetSession 类，该类用来建立与某个 Internet 服务器的会话。只有首先建立起与其他服务器的会话，才能够进行下一步的操作。

其次是 CInternetConnection 类及其派生类 CHttpConnection、CFtpConnection 和 CGopherConnection 类等。这些类帮助用户管理与 Internet 服务器的连接，同时，还提供了大量的成员函数用来与相应的 HTTP、FTP 或 Gopher 服务器通信。

最后，CInternetFile 类及其派生类 CHttpFile、CGopherFile 实现了对远程系统上的文件的存取工作。而 CFindFile 类及其派生类 CFtpFindFile、CGopherFindFile 类实现了对本地系统及远程系统上文件的搜索和定位工作。

如果应用程序只是作为一个浏览器访问 Internet 上的资源，那么很简单，只要首先建立一个 CInternetSession 对象，即建立一个会话。然后就可以调用会话的 OpenURL()函数打开指定位置的资源，当然，用户必须提供正确的 URL。

如果应用程序需要进一步的操作，比如说，需要向 FTP 服务器上传文件，那么可以在建立 CinternetSession 会话后调用 GetFtpConnection()等函数建立一个连接。通过连接可以进行一些“ 高级 ”的操作。

本节中将建立的“Query”程序只作为一个浏览器使用，而且是一个很粗糙的浏览器，只能返回本次连接的一些基本信息。本例子程序的目的只在于说明如何建立与 Internet 服务器的会话与连接，建立连接之后进一步的完善工作用户可以自己完成。

10.1.1 建立应用程序框架

“Query”程序是基于对话框界面的，因此在 AppWizard 的第一步中选择“Dialog Based”程序类型，其他步骤中接受缺省设置即可。

1. 修改“Query”程序的界面

由于“Query”程序是一个基于对话框的应用程序，首先需要确定界面对话框资源。图 10-1 显示了最终完成的 IDD_QUERY_DIALOG 资源。

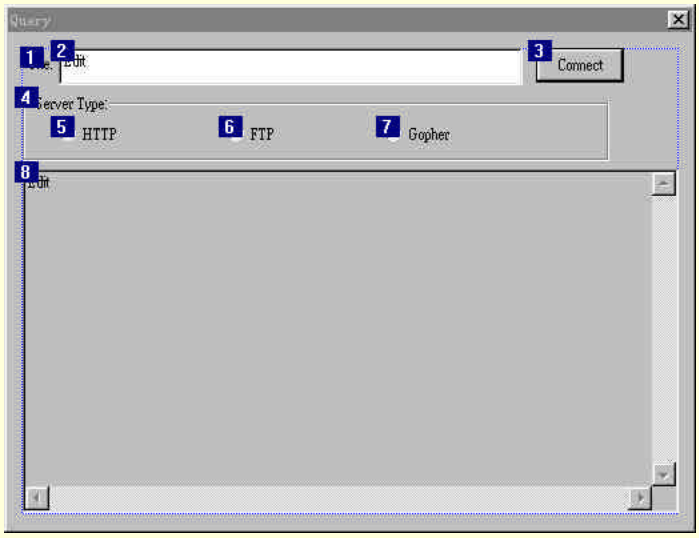

图 10-1 “Query”程序的主界面对话框

在 IDD_QUERY_DIALOG 对话框中，提供了一组单选按钮供用户选择服务类型，用户向编辑框中输入目标服务器的地址后，单击“Connect”按钮开始连接过程。连接的结果，无论成功与否，都显示在下方的编辑框中。

表 10-1 中列出了 IDD_QUERY_DIALOG 对话框中各控件的属性。

表 10-1 IDD_QUERY_DIALOG 对话框的控件

跳转顺序	控件类型	命令 ID	标题文本
1	静态文本	IDC_STATIC	Site:
2	编辑框	IDC_EDIT_SITE	无
3	按钮	IDC_BTN_CONNECT	Connect
4	成组框	IDC_STATIC	Server Type
5	单选按钮	IDC_RAD_HTTP	HTTP
6	单选按钮	IDC_RAD_FTP	FTP
7	单选按钮	IDC_RAD_GOPHER	Gopher
8	编辑框	IDC_EDIT_RESULT	无

在这些控件中，需要修改 IDC_RAD_HTTP 单选按钮和 IDC_EDIT_RESULT 编辑框的属性。对 IDC_RAD_HTTP 单选按钮，确定选中了“Group”选项；对 IDC_EDIT_RESULT 编辑框，选中“Read-only”、“Auto HScroll”、“Auto VScroll”和“MultiLine”等属性，该编辑框将用来显示连接的结果。

用户还必须向 CQueryDlg 类添加几个成员变量，表 10-2 中列出了这些成员变量。

表 10-2 CQueryDlg 类的成员变量

控件 ID	变量类型	变量名称
IDC_EDIT_SITE	CString	m_szSite
IDC_EDIT_RESULT	CString	m_szResult
IDC_RAD_HTTP	int	m_nType

2．响应按钮单击

最后，需要响应“Connect”按钮的 BN_CLICKED 通知，并按照程序清单 10-1 所示修改 OnBtnConnect()函数的代码。

程序清单 10-1　CQueryDlg::OnBtnConnect()

```
void CQueryDlg::OnBtnConnect()
{
// TODO: Add your control notification handler code here
UpdateData(TRUE);

switch(m_nType)
{
case 0:
    TryHttp();
    break;
case 1:
    TryFtp();
    break;
case 2:
    TryGopher();
    break;
}
}
```

在 OnBtnConncet()函数中，通过对当前选中的服务类型的判断，调用不同的辅助函数完成连接工作。其中 TryHTTP()、TryFTP()和 TryGopher()等辅助函数将在下一小节中完成。

10.1.2　与 Internet 连接

“Query”程序提供了三种连接 Internet 的方式：HTTP、FTP 和 Gopher，但所有的操作都必须以用户提供正确的 Internet 服务器地址为前提。当然，“Query”程序的目的只是演示如何与指定的 Internet 服务器进行连接，输出的结果也很简单：如果没有连接上，则显示“Sorry”；如果连接上了，除了显示“OK”之外，还附带了一些基本的信息。

1．HTTP 连接

如果用户已经输入了正确的 HTTP 站点的地址或者域名，并且选中“HTTP”服务类型，单击“Connect”按钮，那么程序运行的结果就完全由 TryHttp()函数决定。首先在程序清单 10-2 中列出了 TryHttp()函数的代码。

程序清单 10-2　CQueryDlg::TryHttp()

```
void CQueryDlg::TryHttp()
{
// 初始化数据
if(m_szSite.Left(7)!="http://")
{
    AfxMessageBox("HTTP site needed!");
    return;
}
m_szResult.Empty();
```

```
m_szResult+="Please wait while trying to open "+m_szSite+"\r\n";
UpdateData(FALSE);

// 尝试连接
CInternetSession session;
CHttpFile* pFile=NULL;
try
{
    pFile=(CHttpFile*) session.OpenURL(m_szSite);
}
catch(CInternetException* pEx)
{
    pFile=NULL;
    pEx->Delete();
}

// 读取文件
if(pFile)
{
    m_szResult+="OK! HTTP Site found! \r\n";
    m_szResult+="---------------------------------\r\n";
    CString szLine;
    while(pFile->ReadString(szLine))
    {
        m_szResult+=szLine+"\r\n";
    }
    pFile->Close();
    delete pFile;
}
else
{
    m_szResult+="Sorry! HTTP Site not found! \r\n";
}

// 关闭会话
session.Close();
UpdateData(FALSE);
}
```

在 TryHttp()**函数中用到了许多** MFC **提供的** WinInet **类，为了使编译器能够寻找到这些类的声明与定义，用户需要在** QueryDlg.cpp **文件的顶部添加如下一行：**#include "afxinet.h"

在 TryHttp()**函数中，首先构造了一个** CInternetSession **类的会话对象。在任何使用** WinInet **类开发的** Internet **客户端程序中，都必须首先构造一个会话对象，在对象的构造函数将完成内部数据结构的初始化并为应用程序后面的调用做好准备。**

为了创建与 HTTP **站点的一个连接，调用了会话对象的** OpenURL()**函数，该函数总是返回一个文件类型的指针，具体类型与指定站点提供的服务有关。**OpenURL()**函数支持四种协议：**

■ file:// **打开一个本地文件。函数创建一个** CStdioFile **对象并返回其指针。**

■ ftp:// 试图连接一个 FTP 站点，如果连接成功，返回一个指向 CInternetFile 对象的指针。

■ http:// 试图连接一个 HTTP 站点，如果连接成功，返回一个指向 CHttpFile 对象的指针。

■ gopher:// 试图连接一个 Gopher 站点，如果连接成功返回一个指向 CGopherFile 对象的指针。

由于在 TryHttp()函数的最开头确保了本函数中只处理 HTTP 连接，因此最终将 OpenURL()函数返回的文件指针强制转换为 CHttpFile 类型。

读取文件数据的工作由 ReadString()函数完成，该函数每次从文件中读取连续的字符串，当遇到新的行标志时便停止。如果遇到文件尾导致读取失败，则返回值为 FALSE。

如果用户想现在就尝试一下“Query”程序访问 HTTP 站点的功能，那么可以编译并运行“Query”程序，在“Site”编辑框中输入某个 HTTP 站点的网址，单击“Connect”按钮就可以等待结果出现了。等待的时间取决于用户的计算机与 Internet 网连接的速度。

可以输入一个 HTTP 站点的域名进行检验，比如说输入“http://www. tsinghua.edu.cn”，“Query”程序应该能够找到它，图 10-2 就是连接至该站点时的结果。

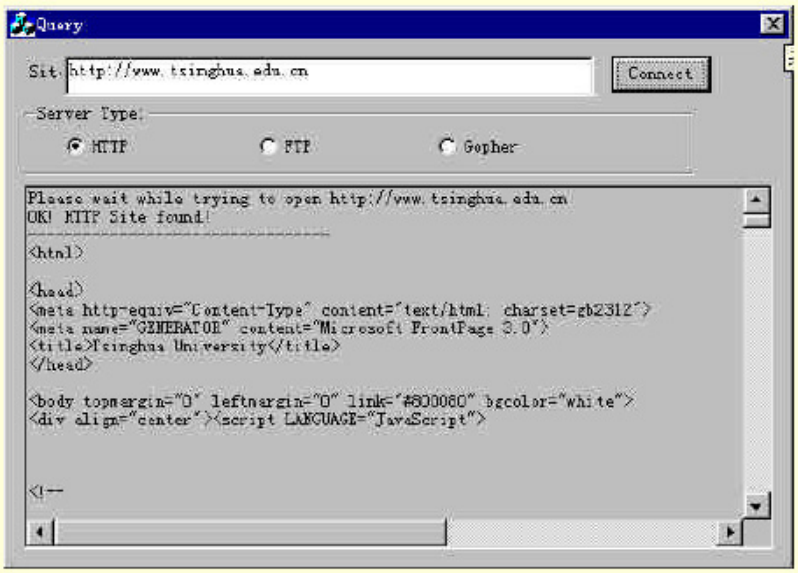

图 10-2 “Query”连接到 HTTP 站点

有点遗憾的是，“Query”程序只能以字符串的方式来显示网页的内容。但是这已经足够了，“Query”程序从来都不希望成为一个漂亮的浏览器。在本章的下一节中，将向用户介绍如何建立一个自己的浏览器程序，当然，这需要 MFC 提供类的支持。

2. FTP 连接

访问 FTP 站点的工作与访问 HTTP 站点是类似的，程序清单 10-3 中列出了 TryFTP()函数的代码。

程序清单 10-3 CQueryDlg::TryFtp()

```
void CQueryDlg::TryFtp()
{
// 初始化数据
if(m_szSite.Left(6)!="ftp://")
{
    AfxMessageBox("FTP site needed!");
    return;
}
m_szResult.Empty();
m_szResult+="Please wait while trying to open "+m_szSite+"\r\n";
UpdateData(FALSE);
```

```
// 尝试连接
CInternetSession session;
CFtpConnection* pFtp;
try
{

    pFtp=session.GetFtpConnection(m_szSite);
}
catch(CInternetException* pEx)
{
    pFtp=NULL;
    pEx->Delete();
}

// 读取 FTP 站点的当前目录
if(pFtp)
{
    m_szResult+="OK! FTP Site found! \r\n";
    m_szResult+="----------------------------------\r\n";
    CString szLine;
    pFtp->GetCurrentDirectory(szLine);
    m_szResult+="Current directory: "+ szLine +"\r\n";
    pFtp->Close();
    delete pFtp;
}
else
{
    m_szResult+=" Sorry! FTP Site not found! \r\n";
}

// 关闭会话
session.Close();
UpdateData(FALSE);
}
```

可以看到，TryFtp()函数和 TryHttp()函数的大体结构是类似的。不同的是在 TryFtp()函数中没有使用 OpenURL()函数以打开一个 FTP 站点，而是调用了 GetFtpConnection()函数建立一个 FTP 连接。最后，调用 GetCurrenDirectory()函数获取远程站点上的当前目录。

TryFtp()函数并没有做更多的事情，用户可以通过 CFindFile 类对象从当前目录开始进行搜索与定位，这又有些类似于本书第四章中的“ NewCtrl ”程序，不同的是“ Query ”程序操作的对象是远程系统，而“ NewCtrl ”程序只搜索本地系统上的文件。

3. Gopher 连接

访问 Gopher 站点的工作由 TryGopher()函数完成，程序清单 10-4 中是该函数的代码。

程序清单 10-4 CQueryDlg::TryGopher()

```
void CQueryDlg::TryGopher()
```

```
{
// 初始化数据
m_szResult.Empty();
m_szResult+="Please wait while trying to open "+m_szSite+"\r\n";
UpdateData(FALSE);

// 尝试连接
CInternetSession session;
CGopherConnection* pGopher;
try
{
    pGopher=session.GetGopherConnection(m_szSite);
}
catch(CInternetException* pEx)
{
    pGopher=NULL;
    pEx->Delete();
}
// 返回 Gopher 站点信息
if(pGopher)
{
    m_szResult+="OK! FTP Site found! \r\n";
    m_szResult+="----------------------------------\r\n";
    CString szLine;
    CGopherLocator locator=
            pGopher->CreateLocator(NULL,NULL,GOPHER_TYPE_DIRECTORY);
    szLine=locator;

    m_szResult+="Found locator: "+szLine+" \r\n";
    pGopher->Close();
    delete pGopher;
}
else
{
    m_szResult+=" Sorry! Gopher Site not found! \r\n";
}

// 关闭会话
session.Close();
UpdateData(FALSE);
}
```

TryGopher()函数的代码与 TryFtp()函数是完全类似的，读者可以自行分析。

现在读者已经完成了“ Query ”程序的所有代码，可以对程序的功能进行检验了。这里需要提醒读者的是，“ Query ”程序并没有对用户输入的 Internet 服务器的域名或地址进行转换与尝试，因此，如果用户输入的网址不存在或者是错误的，就无法得到正确的结果。

10.2 制作 Web 浏览器

互联网是目前最流行的技术之一，互联网上众多的站点提供了大量的信息，许多计算机用户已经视互联网是自己生活不可分割的一部分。Microsoft 公司提供的 Internet Explorer 和 Netscape 公司提供的 Navigator 成为了大部分 PC 用户浏览网上信息时首选的工具。

然而，利用 VC6.0 开发一个用户自己的 Web 浏览器也不是一项很困难的工作。使用 MFC 提供的 CHtmlView 类，再加上一些努力，很快就能够看到自己的丰硕成果。本节中就将通过“Browser”例子程序介绍如何创建自己的 Web 浏览器。

“Browser”程序的视图类从 CHtmlView 类派生，这使得“Browser”程序很容易就获得了浏览 Web 页面的能力。CHtmlView 类是 VC6.0 中新近提供的 MFC 类，通过封装 WebBrowser ActiveX 控件实现其功能，而只有安装了 Internet Explorer 4.0 的计算机才拥有该控件。因此，必须确保安装了 Internet Explorer 4.0，否则可能无法完成“Browser”程序。

10.2.1 建立应用程序框架

“Browser”程序首先是一个典型的单文档界面的应用程序，除了视图类从 CHtmlView 类派生之外，“Browser”程序还有一点点特殊，这就是工具栏的风格。本书前面各章中使用的例子程序，在 AppWizard 的第四步中都选择工具栏的风格为“Normal”，而“Browser”程序选择“Internet Explorer ReBars”。需要指出的是，这也是 VC6.0 中才提供的新选项。这将使得“Browser”程序看起来更像 Internet Explorer。

用户可以使用 AppWizard 建立“Browser”程序的框架，在单击“Finish”按钮之前，需要确定一下是否更改了视图类的基类和工具栏的风格，其他的选项可以自行设置。

1．WebBrowser 控件与 CHtmlView 类

在开始向“Browser”程序中添加代码之前，了解一下 WebBrowser 控件和 CHtmlView 类的基本知识是很有帮助的，因为“Browser”程序的功能将主要地通过这些对象实现。

WebBrowser 控件支持通过超级链接和 URL 对互联网上的资源进行访问。WebBrowser 控件中还保存了一份用户最近访问过的网址列表，这使得用户在浏览时可以在已经访问过的网址中前进或者后退。WebBrowser 控件直接管理用户在互联网上的浏览活动，管理用户的最近访问过的网址列表、最喜爱的网址列表和安全检测等等。同时，WebBrowser 控件还是一个 Active 文档容器和 ActiveX 控件容器。一句话，WebBrowser 控件几乎就是 Internet Explorer 4.0 的内核。

由于 CHtmlView 封装了 WebBrowser 控件，因此通过 CHtmlView 或者其派生类来实现 Web 页面浏览功能就更方便了。CHtmlView 提供了大量的成员函数帮助用户完成浏览功能，表 10-3 中列出了与浏览操作有关的一些成员函数，这只是 CHtmlView 类成员的一小部分。

表 10-3 CHtmlView 类的一部分成员函数

函数名称	函数功能说明
GoBack	浏览至网址列表中的上一项
GoForward	浏览至网址列表中的下一项
GoHome	浏览至指定的起始网址
GoSearch	浏览至指定的搜索网址
Refresh	刷新当前显示的信息
Stop	停止当前的下载操作
Navigate2	浏览至指定网址或打开本地文件
ExecWB	执行特殊动词

用户将在“Browser”程序中看到这些函数的使用。

2．CReBar 类——成组栏

CReBar 类也是 VC6.0 提供的新类。与 CToolBar、CStatusBar 和 CDialogBar 等类似，CReBar 类也是从 CControlBar 类派生，因此 CReBar 也是一种控制条。但是 CReBar 似乎有些特殊的地方，可以参看本章后面的图 10-6。该图中是“Browser”程序最终完成后的情形，可以看到，“Browser”程序拥有两个工具栏，同时这两

个工具栏又位于同一个成组栏中，因此确切地说，成组栏为其他的控制栏或控件提供了一个支撑环境。使用 CReBar 类，可以为自己的程序添加漂亮的工具栏。在“ Browser ”程序中就实现了这一点。

10.2.2 浏览 Web 页

用户或许现在就想来实现图 10-6 所示的漂亮的工具栏，但是这需要添加一大段代码。与此相比，实现浏览 Web 页的功能更重要，相对而言也要容易一些。因此这里先来介绍如何实现“ Browser ”程序浏览 Web 页的功能。

1. 打开本地 HTML 文件

对于一个简单的 Web 页浏览器而言，“ File ”菜单下的许多命令都是不需要的，可以将多余的菜单项都删除掉，只留下“ Open ”、“ Print ”和“ Exit ”就够了。

首先，如果在本地系统中已经保存了 HTML 文件，“ Browser ”程序应该可以将其打开，这需要对“ File ”菜单下“ Open ”菜单项发送的 COMMOND 消息作出反应。

在一般的文档/视图结构的应用程序中，选择‘ File ’菜单下的‘ Open ’菜单项将调用 CWinApp::OnFileOpen()，该函数显示一个“ Open ”对话框，在选择了将打开的文件之后，调用文档类的 Serialize()函数进行串行化，然后在视图中显示出来。

但是“ Browser ”程序不需要这么复杂，由于 CHtmlView::Navigate2()函数可以直接打开 HTML 文件，因此不需要经过文档类的串行化过程。可以使用 ClassWizard 生成对“ Open ”菜单项的 COMMOND 消息的处理函数，由 CBrowserView 类接收该消息。程序清单 10-5 中列出了 OnFileOpen()函数的代码。

程序清单 10-5 CBrowserView::OnFileOpen()

```
void CBrowserView::OnFileOpen()
{
// TODO: Add your command handler code here
CString str;
str="HTML Files|*.htm;*.html";

CFileDialog dlg(TRUE, NULL, NULL, OFN_HIDEREADONLY, str);
if(dlg.DoModal() == IDOK)
    Navigate2(dlg.GetPathName(), 0, NULL);
}
```

可以看到，OnFileOpen()函数中的代码并不是很复杂，关键在于对 Navigate2()函数的调用，该函数直接打开选中的 HTML 文件，并在视图中完成显示。这些都无须用户自己实现。

2. 通过 URL 进行访问

通过 URL 访问 HTML 页是经常使用的一种途径，URL（Uniform Resource Locator）标识了互联网上某个文档、图片或者其他类型文件的全路径，URL 表明了文件存取的协议和位置。“ Browser ”程序也应该提供通过 URL 访问 Web 页面的能力。

在最终完成的“ Browser ”程序中，可以在工具栏中输入网址，但是现在由于工具栏还没有建立，因此只能通过菜单弹出对话框输入了。

在“ Browser ”程序的 IDR_MAINFRAME 菜单资源中，在“ View ”菜单前添加一个新的下拉菜单“ Go ”，并向“ Go ”菜单中添加几个菜单项，表 10-4 中列出了这些菜单项的有关信息。

表 10-4 “ Go ”菜单的菜单项

命令 ID	标题文本	提示信息
ID_GO_ADDRESS	Address...	Go to the address you specified.\nAddress
ID_GO_BACK	Back	Goes back one step.\nBack

续 表

命令 ID	标题文本	提示信息
ID_GO_FORWARD	Forward	Goes forward one step.\nForward
ID_GO_START_PAGE	Start Page	Opens your start page.\nStart Page
ID_GO_SEARCH_THE_WEB	Search the Web	Opens your search page.\nSearch the Internet

在“Go”菜单中有五个菜单项，这里先来完成“Address”菜单项，其他几个的处理在稍后介绍。用户还需要建立一个新的对话框资源 IDD_ADDRESS，如图 10-3 所示。

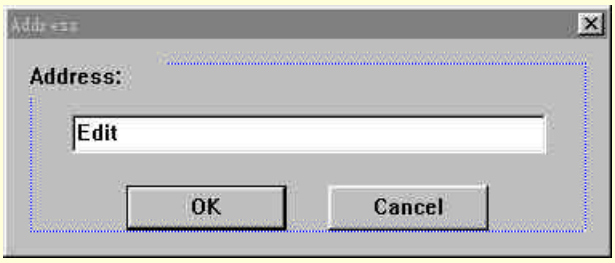

图 10-3 “Address”对话框

“Address”对话框很简单，只需要添加一个静态文本控件和一个编辑框即可。基于 IDD_ADDRESS 对话框资源，建立新的对话框类 CAddressDlg。在 CAddressDlg 类中添加一个与编辑框控件相关联的成员变量 m_szAddress，类型为 CString。

最后，通过 ClassWizard 对“Go”菜单下的“Address”菜单项命令添加消息处理函数，接受缺省的函数名：OnGoAddress()，由 CBrowserView 类接收该命令。程序清单 10-6 中是该函数的代码。

程序清单 10-6 CBrowserView::OnGoAddress()

```
void CBrowserView::OnGoAddress()
{
// TODO: Add your command handler code here
CAddressDlg dlg;
if(dlg.DoModal()==IDOK)
    Navigate2(dlg.m_szAddress,0,NULL);
}
```

在开始编译之前，在“BrowserView.cpp”文件的顶部添加以下头文件：

```
#include "AddressDlg.h"
```

现在可以检验“Browser”程序的功能了，既可以通过在“File”菜单下选择“Open”菜单项打开本地 HTML 文件，也可以通过选择“Go”菜单下的“Address”菜单项，在“Address”对话框中输入 URL 而访问远程系统上的 Web 页面。

这些属于“Browser”程序的基本功能，后面将完成其他的一些辅助功能，这些功能完全是“锦上添花”。

3．后退、前进、搜索与回到起始页

由于 CHtmlView 封装的 WebBrowser 控件完全管理了用户在互联网上的“冲浪”过程，因此实现后退、前进等功能十分简单，只要调用 CHtmlView 类中相应的成员函数就可以了。程序清单 10-7 中列出了有关函数的代码，可以看到，这些代码是相当枯燥的。

程序清单 10-7 后退、前进、搜索与回到起始页

```
void CBrowserView::OnGoBack()
{
// TODO: Add your command handler code here
GoBack();
```

```
}

void CBrowserView::OnGoForward()
{
// TODO: Add your command handler code here
GoForward();
}

void CBrowserView::OnGoStartPage()
{
// TODO: Add your command handler code here
GoHome();
}

void CBrowserView::OnGoSearchTheWeb()
{
// TODO: Add your command handler code here
GoSearch();
}
```

这些函数完成了对“Go”菜单下其他菜单项命令的处理，可以使用 ClassWizard 生成这些消息处理函数，然后完成代码。

4．刷新、停止和选择字体

为了实现刷新、停止和选择字体的功能，需要向“Browser”程序的“View”菜单下添加一些新的菜单项，图 10-4 中是最终完成后的“View”菜单。

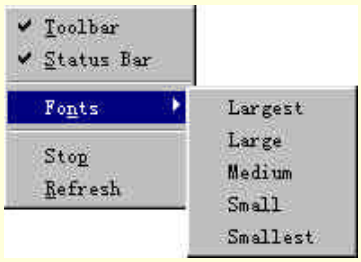

图 10-4 “View”菜单

表 10-5 中列出了这些菜单项的信息。

表 10-5 “View”菜单的菜单项

命令 ID	标题文本	提示信息
ID_VIEW_STOP	Stop	Stops opening a file.\nStop
ID_VIEW_FRESH	Refresh	Refreshes the contents of the current page.\nRefresh
ID_VIEW_FONTS_LARGEST	Largest	Selects largest font size.
ID_VIEW_FONTS_LARGE	Large	Selects large font size.
ID_VIEW_FONTS_MEDIUM	Medium	Selects medium font size.
ID_VIEW_FONTS_SMALL	Small	Selects small font size.
ID_VIEW_FONTS_SMALLEST	Smallest	Selects smallest font size.

最后，需要使用 ClassWizard 生成这些菜单项命令的消息处理函数，程序清单 10-8 中列出了这些函数的代码。

程序清单 10-8 刷新、停止和选择字体

```
void CBrowserView::OnViewRefresh()
{
Refresh();
}

void CBrowserView::OnViewStop()
{
Stop();
}

void CBrowserView::OnViewFontsLargest()
{
COleVariant vaZoomFactor(4l);
ExecWB(OLECMDID_ZOOM, OLECMDEXECOPT_DONTPROMPTUSER,
        &vaZoomFactor, NULL);
}

void CBrowserView::OnViewFontsLarge()
{
COleVariant vaZoomFactor(3l);
ExecWB(OLECMDID_ZOOM, OLECMDEXECOPT_DONTPROMPTUSER,
        &vaZoomFactor, NULL);
}

void CBrowserView::OnViewFontsMedium()
{
COleVariant vaZoomFactor(2l);
ExecWB(OLECMDID_ZOOM, OLECMDEXECOPT_DONTPROMPTUSER,
        &vaZoomFactor, NULL);
}

void CBrowserView::OnViewFontsSmall()
{
COleVariant vaZoomFactor(1l);
ExecWB(OLECMDID_ZOOM, OLECMDEXECOPT_DONTPROMPTUSER,
        &vaZoomFactor, NULL);
}

void CBrowserView::OnViewFontsSmallest()
{
COleVariant vaZoomFactor(0l);
ExecWB(OLECMDID_ZOOM, OLECMDEXECOPT_DONTPROMPTUSER,
        &vaZoomFactor, NULL);
}
```

在这些函数中，OnViewStop()**和** OnViewFresh()**是很简单的，只是调用了** CHtmlView **类提供的相应函数。需要说明的是** OnViewFontXXX()**函数(其中** XXX **代表某些特定字符串),以** OnViewFontLargest()**函数为例，**

首先构造了一个 COleVariant 类型的对象，注意传递给构造函数的参数是 4 加上小写的字母 l，不是数字 41，该参数指定了调整字体大小的比例，这只是一个理想的值，具体的实现有赖于用户系统中安装的字体。

调用 ExecWB()函数是完成字体选择的关键，该函数最终执行 WebBrowser 控件暴露的 Exec 方法。Exec 方法可以根据指定的不同参数执行特定的操作，可供选择的操作类型有 37 种之多，CHtmlView 对其中的一些操作进行了封装，如刷新、停止等，更多的操作可以通过调用 CHtmlView::ExecWB()函数实现。

现在已经完成了“ Browser ”程序的基本功能，可以对刚才添加的代码进行一番测试了。在 10.2.3 节中将介绍如何改善“ Browser ”程序的界面，具体地说，就是使用 CReBar 类实现漂亮的工具栏。

如果用户愿意增强本程序的功能，完全可以在“ Browser ”程序的基础上，充分利用 WebBrowser 控件暴露的属性、方法和事件，一定能够实现一个功能强大的 Web 浏览器。

10.2.3 改善程序的界面

在“ Browser ”程序中，改善程序的界面主要是指改善程序的工具栏。从图 10-6 可以看出，“ Browser ”程序的工具栏由两部分组成，一部分是真正的“ 工具 ”栏，包含了一些常用的浏览功能按钮，另外一部分实际上是一个组合框控件，用来输入网址。支撑这两部分的则是一个成组栏。用户现在或许还在对成组栏的作用有些疑惑，在完成程序的所有代码之后，就会清楚了。

1．修改 CMainFrame 类

用户已经知道，工具栏是应用程序主框架窗口的子窗口，因此改善程序的工具栏就意味着在 CMainFrame 类中做必要的修改。首先，需要修改 CMainFrame 类中工具栏对象的声明。

打开 CMainFrame 类的头文件就可以看到，在 CMainFrame 类中以成员变量的方式包含了四个控制条对象，如下所示：

```
protected:    // control bar embedded members
CStatusBar    m_wndStatusBar;
CToolBar      m_wndToolBar;
CReBar        m_wndReBar;
CDialogBar    m_wndDialogBar;
```

保留这四个对象中的前三个，而将最后一个替换为：

```
CComboBoxEx m_wndAddressBar;
```

建立这些控制条对象的工作在 CMainFrame::OnCreate()函数中完成，由于缺省的工具栏的创建不符合“ Browser ”程序的要求，因此首先删除 OnCreate()函数中所有与 m_wndStatusBar、m_wndDialogBar 和 m_wndReBar 对象有关的代码，然后再添加新的代码。

但是由于在“ Browser ”程序中需要添加的代码比较长，因此一种比较好的方法是在 CMainFrame 类中添加几个辅助函数，然后在 OnCreate()函数中调用这些辅助函数。程序清单 10-9 中列出了修改后的 OnCreate()函数的代码。

程序清单 10-9　CMainFrame::OnCreate()

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

// 创建状态栏
if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
{
        TRACE0("Failed to create status bar\n");
```

```
        return -1;          // fail to create
    }

    // 调用辅助函数创建工具栏
    CreateToolBar();
    CreateAddressBar();
    CreateReBar();
    return 0;
    }
```

在 OnCreate()函数中，调用了三个辅助函数 CreateToolBar()、CreateAddressBar()和 CreateReBar()，分别完成构造和初始化 m_wndToolBar、m_wndAddressBar 和 m_wndReBar 对象的工作，稍后将完成这些函数的具体代码。

2．完成 CreateToolBar()函数

“Browser”程序的工具栏是类似于 Internet Exploer 4.0 风格的，工具栏的按钮在两种状态之间切换：当按钮处于非活动状态时，按钮表面与工具栏表面平齐，而且表现为比较灰暗的颜色，这称为按钮的“冷”态；当鼠标指向按钮时，按钮表面突起，并且表现为比较明亮的颜色，这称为按钮的“热”态。

实现这种工具栏的关键在于为工具栏提供两套图标，分别在按钮处于“冷”态和处于“热”态时使用。图 10-5 中所示就是“Browser”程序使用的两套图标。

图 10-5　工具栏按钮的两套图标

遗憾的是，由于本书印刷时只能采用黑白两色，因此读者看到的图 10-5 中的两套图标可能没有什么太大的区别。但实际上这两套图标是很漂亮的，几乎完全类似于 Internet Explorer 4.0 中的图标。

每一套图标都被保存在一个位图文件中，可以使用 VC6.0 集成开发环境中提供的位图编辑器来建立这两套图标的位图。在图 10-5 中，每个图标都是 22X20 像素大小，背景色为紫色，用颜色分量来表示就是：RGB(255,0,255)。记住这个分量值，在后面的代码中将要用到。当然，完全可以采用其他颜色作为背景色。

建立了这两套图标的位图后，将对应位图的标识符分别设置为 IDB_COLDTOOLBAR 和 IDB_HOTTOOLBAR。在 CreateToolBar()函数中将根据指定的标识符访问这两套图标。

CreateToolBar()函数的代码比较长，如程序清单 10-10 所示。

程序清单 10-10　CMainFrame::CreateToolBar()

```
void CMainFrame::CreateToolBar()
{
CImageList img;
CString str;

// 创建工具栏对象
if (!m_wndToolBar.CreateEx(this))
{
    TRACE0("Failed to create toolbar\n");
    return;          // fail to create
}
```

```
// 设置工具栏的属性
m_wndToolBar.GetToolBarCtrl().SetButtonWidth(50, 150);
m_wndToolBar.GetToolBarCtrl().SetExtendedStyle(
                                TBSTYLE_EX_DRAWDDARROWS);

// 设置“冷”态和“热”态时的图标
img.Create(IDB_HOTTOOLBAR, 22, 0, RGB(255, 0, 255));
m_wndToolBar.GetToolBarCtrl().SetHotImageList(&img);
img.Detach();
img.Create(IDB_COLDTOOLBAR, 22, 0, RGB(255, 0, 255));
m_wndToolBar.GetToolBarCtrl().SetImageList(&img);
img.Detach();
m_wndToolBar.ModifyStyle(0, TBSTYLE_FLAT | TBSTYLE_TRANSPARENT);
m_wndToolBar.SetButtons(NULL, 8);

// 设置每个按钮的属性
m_wndToolBar.SetButtonInfo(0, ID_GO_BACK, TBSTYLE_BUTTON, 0);
str.Format("Back");
m_wndToolBar.SetButtonText(0, str);
m_wndToolBar.SetButtonInfo(1, ID_GO_FORWARD, TBSTYLE_BUTTON, 1);
str.Format("Forward");
m_wndToolBar.SetButtonText(1, str);
m_wndToolBar.SetButtonInfo(2, ID_VIEW_STOP, TBSTYLE_BUTTON, 2);
str.Format("Stop");
m_wndToolBar.SetButtonText(2, str);
m_wndToolBar.SetButtonInfo(3, ID_VIEW_REFRESH, TBSTYLE_BUTTON, 3);
str.Format("Fresh");
m_wndToolBar.SetButtonText(3, str);
m_wndToolBar.SetButtonInfo(4, ID_GO_START_PAGE, TBSTYLE_BUTTON, 4);
str.Format("Home");
m_wndToolBar.SetButtonText(4, str);
m_wndToolBar.SetButtonInfo(5, ID_GO_SEARCH_THE_WEB,
                        TBSTYLE_BUTTON, 5);
str.Format("Search");
m_wndToolBar.SetButtonText(5, str);
m_wndToolBar.SetButtonInfo(6, ID_FILE_PRINT, TBSTYLE_BUTTON, 6);
str.Format("Print");
m_wndToolBar.SetButtonText(6, str);
m_wndToolBar.SetButtonInfo(7, ID_FONT_DROPDOWN, TBSTYLE_BUTTON |
                        TBSTYLE_DROPDOWN, 7);
str.Format("Font");
m_wndToolBar.SetButtonText(7, str);

// 设置工具栏的大小
CRect rectToolBar;
m_wndToolBar.GetItemRect(0, &rectToolBar);
```

```
m_wndToolBar.SetSizes(rectToolBar.Size(), CSize(30,20));
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_FIXED);
}
```

在 CreateToolBar()函数中，创建工具栏对象部分的代码是很平常的。在设置工具栏属性的部分，调用了 GetToolBarCtrl()函数，读者或许会对此感到疑惑。实际上，MFC 的工具栏是通过一个工具栏控件实现的，GetTooBarCtrl()函数则可以获取对工具栏控件的直接控制。

接着，CreateTooBar()函数设置了工具栏按钮在“冷”态和“热”态时使用的图标列表，这是通过调用 SetImageList()和 SetHotImageList()函数实现的。注意在调用图像列表的 Create()函数时指定的参数，最后一个参数就是图标的背景色，这也是前面提醒用户需要注意的。该参数指定的颜色被用来产生一个颜色掩码，最终的效果就是图标中使用了该种颜色的区域是透明的。

最长的代码出现在设置按钮属性的部分，对工具栏上的每一个按钮，都调用了 SetButtonInfo()函数和 SetButtonText()函数设置按钮的属性，SetButtonInfo()函数指定按钮的命令 ID、风格和对应图标在图标列表中的索引，而 SetButtonText()函数指定按钮上的标识文本。

注意最后一个按钮的风格设置中包含了 TBSTYLE_DROPDOWN 标志，这表明该按钮将作为一个下拉按钮出现，在按钮的右边会添加一个下拉箭头，单击该下拉箭头会弹出更多的选择。该按钮的命令 ID 为 ID_FONT_DROPDOWN，用户需要为该 ID 指定一个编号，该 ID 可以不与任何资源相联系。

最后，设置了工具栏上各按钮的大小和按钮上图标的大小，并为工具栏增添了一些风格标志。

3. 完成 CreateAddressBar()函数

CreateAddressBar()函数可能是最简单的，其代码如程序清单 10-11 所示。

程序清单 10-11 CMainFrame::CreateAddressBar()

```
void CMainFrame::CreateAddressBar()
{
if (!m_wndAddressBar.Create(CBS_DROPDOWN | WS_CHILD,
    CRect(0, 0, 200, 120), this, AFX_IDW_TOOLBAR + 1))
{
    TRACE0("Failed to create combobox\n");
    return;          // fail to create
}
}
```

在 CreateAddressBar()函数中，m_wndAddressBar 成员的初始化函数 Create()指定了其命令 ID 为 AFX_IDW_TOOLBAR+1，该命令 ID 用来区分各不同的工具栏，在接收工具栏向主框架窗口发送的通知时将用到该命令 ID。在创建 m_wndToolBar 对象时，并没有指定工具栏的命令 ID，这时使用的是缺省的设置，即 AFX_IDW_TOOLBAR。

4. 完成 CreateReBar()函数

成组栏为上述工具栏提供了一个停泊的环境，程序清单 10-12 中是 CreateReBar()函数的完整代码。

程序清单 10-12 CMainFrame::CreateReBar()

```
void CMainFrame::CreateReBar()
{
// 创建 CReBar 对象
if (!m_wndReBar.Create(this))
{
    TRACE0("Failed to create rebar\n");
```

```
        return;          // fail to create
    }

    // 添加工具栏
    CString str;
    m_wndReBar.AddBar(&m_wndToolBar);
    str.Format("Address");
    m_wndReBar.AddBar(&m_wndAddressBar, str, NULL,
                      RBBS_FIXEDBMP | RBBS_BREAK);

    // 设置各工具栏的参数
    REBARBANDINFO rbbi;
    CRect rectToolBar;

    m_wndToolBar.GetItemRect(0, &rectToolBar);

    rbbi.cbSize = sizeof(rbbi);
    rbbi.fMask = RBBIM_CHILDSIZE | RBBIM_IDEALSIZE | RBBIM_SIZE;
    rbbi.cxMinChild = rectToolBar.Width();
    rbbi.cyMinChild = rectToolBar.Height();
    rbbi.cx = rbbi.cxIdeal = rectToolBar.Width() * 9;
    m_wndReBar.GetReBarCtrl().SetBandInfo(0, &rbbi);
    rbbi.cxMinChild = 0;

    CRect rectAddress;

    rbbi.fMask = RBBIM_CHILDSIZE | RBBIM_IDEALSIZE;
    m_wndAddressBar.GetEditCtrl()->GetWindowRect(&rectAddress);
    rbbi.cyMinChild = rectAddress.Height() + 10;
    rbbi.cxIdeal = 200;
    m_wndReBar.GetReBarCtrl().SetBandInfo(1, &rbbi);
    }
```

在 CreateReBar()函数中，首先创建了 CReBar 对象，然后将前面创建好的工具栏添加到 ReBar 中。最后设置了各子工具栏的参数。在设置子工具栏的参数时，填充了一个 REBARBANDINFO 结构，这里不准备罗列该结构的成员，可以从 VC6.0 的联机手册中了解到详细信息。

5．显示弹出菜单

现在可以试验一下，除了“ Font ”按钮不可选取之外，其他的按钮都已经能够正常地工作。现在就来实现“ Font ”按钮的功能。

在 CreateToolBar()函数中，指定了“ Font ”按钮的命令 ID 为 ID_FONT_DROPDOWN，由于没有对该命令 ID 提供消息处理函数，因此现在看到的“ Font ”按钮呈现灰色禁止状态。由于 ClassWizard 不支持 ID_FONT_DROPDOWN，因此必须手工向 CMainFrame 类添加消息映射入口。

```
ON_COMMAND(ID_FONT_DROPDOWN,DoNothing)
```

DoNothing()函数的目的仅仅是使“ Font ”按钮能够激活，而不需要完成任何操作。用户需要向 CMainFrame 类添加该辅助函数，并不需要添加任何代码。

重要的是在单击了“ Font ”按钮右边的下拉箭头应该弹出字体选择菜单，即“ View ”菜单下“ Font ”下拉

菜单的所有菜单项。在下拉箭头被按下的时候，会向其父窗口，即程序的主框架窗口发送 TBN_DROPDOWN 通知，因此可以通过响应该通知消息实现字体选择弹出菜单。

现在需要再次手工向 CMainFrame 类添加一条消息映射入口。

```
ON_NOTIFY(TBN_DROPDOWN, AFX_IDW_TOOLBAR, OnDropDown)
```

可以看到，ON_NOTIFY 宏的第二个参数指定了发出通知的工具栏的命令 ID，正如前面所指出的，AFX_IDW_TOOLBAR 代表了 m_wndToolBar 对象。

程序清单 10-13 中列出了 OnDropDown()函数的代码。

程序清单 10-13　CMainFrame::OnDropDown()

```
void CMainFrame::OnDropDown(NMHDR *pNotifyStruct, LRESULT *pResult)
{
// 转换坐标
NMTOOLBAR* pNMToolBar = (NMTOOLBAR*)pNotifyStruct;
CRect rect;
m_wndToolBar.GetToolBarCtrl().GetRect(pNMToolBar->iItem, &rect);
rect.top = rect.bottom;
::ClientToScreen(pNMToolBar->hdr.hwndFrom, &rect.TopLeft());

// 显示弹出菜单
if(pNMToolBar->iItem == ID_FONT_DROPDOWN)
{
    CMenu* pMenuBar=AfxGetMainWnd()->GetMenu();
    CMenu* pMenu=pMenuBar->GetSubMenu(1);
    CMenu* pPopup=pMenu->GetSubMenu(3);
    pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_LEFTBUTTON,
                           rect.left, rect.top + 1, AfxGetMainWnd());
}
*pResult = TBDDRET_DEFAULT;
}
```

在 OnDropDown()函数中，为了确定显示弹出菜单的位置，首先获取了“ Font ”按钮的位置并转换为屏幕坐标。然后，获取了“ Font ”下拉菜单的所有菜单项并显示出来。

6．使用网址栏

在“ Browser ”程序中，添加 m_wndAddressBar 组合框的目的就是方便用户输入目的网址，因此不妨将其称之为网址栏。

网址栏必须提供两项功能，一是在组合框中输入新的目的网址后按“ Enter ”键时，“ Browser ”程序应该马上开始指定网址的连接；二是从组合框提供的列表中选择一个网址时，“ Browser ”也应该开始新的连接。

为了完成这两方面的功能，再次向 CMainFrame 类中添加以下消息映射入口：

```
ON_CBN_SELENDOK(AFX_IDW_TOOLBAR + 1,OnNewAddress)
ON_COMMAND(IDOK, OnNewAddressEnter)
```

其中 OnNewAddress()和 OnNewAddressEnter()函数需要手工向 CMainFrame 类添加，程序清单 10-14 中列出了这两个函数的代码。

程序清单 10-14　OnNewAddress()和 OnNewAddressEnter()

```
void CMainFrame::OnNewAddress()
```

```
{
CString str;
m_wndAddressBar.GetLBText(m_wndAddressBar.GetCurSel(), str);
((CBrowserView*)GetActiveView())->Navigate2(str, 0, NULL);
}

void CMainFrame::OnNewAddressEnter()
{
CString str;
m_wndAddressBar.GetEditCtrl()->GetWindowText(str);
((CBrowserView*)GetActiveView())->Navigate2(str, 0, NULL);

COMBOBOXEXITEM item;
item.mask = CBEIF_TEXT;
item.iItem = -1;
item.pszText = (LPTSTR)(LPCTSTR)str;
m_wndAddressBar.InsertItem(&item);
}
```

OnNewAddress()和 OnNewAddressEnter()函数的代码是很简单的，读者可以自行分析。

10.2.4 检查应用程序功能

现在已经完成了“ Browser ”程序的所有代码，在 CHtmlView 类和 WebBrowser 控件的帮助下，很容易就实现了一个界面美观、功能强大的 Web 页浏览器。“ Browser ”程序可供检查的功能有：

- 通过“ File ”菜单下的“ Open ”命令打开本地 HTML 文件；
- 在网址栏中输入新的 HTTP 站点的地址，检查连接状况；
- 检查各种辅助功能，如后退、前进、刷新和停止等等。

图 10-6 中是最终使用“ Browser ”程序连接至 http://sohoo.com.cn 站点时的情形。

在“ Browser ”程序中，使用了 VC6.0 提供的新类 CHtmlView 类和 CReBar 类，但是这远不是 VC6.0 所提供的新特性的全部。读者在今后的使用中还可能会接触到更多的新的特性。

图 10-6 使用“ Browser ”程序连接到 http://sohoo.com.cn

10.3 本章小结

在本章中介绍了一些有关使用 MFC 访问 Internet/Intranet 网络的知识，但这些介绍是很粗浅的。本章的内容仅仅侧重于使用 MFC 提供的 WinInet 类完成各种操作，用户可以从其他专著中了解更多的信息。

“Query”是一个简单的演示程序，展示了如何在 MFC 应用程序建立与 HTTP、FTP 和 Gopher 服务器的连接。

“Browser”程序充分利用了 CHtmlView 类和 WebBrowser 控件而实现了一个漂亮的 Web 页浏览器，并且本浏览器的功能还相当完善。“Browser”程序中还介绍了有关 CReBar 类的一些内容。

需要说明的是，本章介绍了一些有关使用 MFC 访问 Internet/Intranet 网络的知识，但这些介绍是很粗浅的，这里限于作者水平有限，这里所讲解的内容最多只能充当用户开发 Internet 应用程序的一个基础，用户可以从其他方面获取开发 Internet 程序的详细资料。

自此，本书全部讲解完毕，希望读者学完本书后能对 Visual C++有相当的掌握程度。

“书山有路勤为径，学海无涯苦作舟”，希望读者能继续钻研下去，早日成为真正的程序员。