

# 前言

Visual C++是开发运行于Windows 95和Windows NT环境下的Win32应用程序的可视化编程工具中最重要的成员之一，它为软件开发人员提供了完整的编辑、编译和调试工具和建立于Win32 API (Application Programming Interface) 基础上的MFC类库 (Microsoft Foundation Class Library)，从而有效的缩短了Windows应用程序的开发周期。Windows操作系统本身大部分是使用C/C++语言写成的，而Visual C++正是使用C/C++语言的Win32应用程序集成开发环境，因此，使用Visual C++来进行Windows应用程序的开发便有着得天独厚的优势，学习和掌握Visual C++，也自然地就成为了广大程序设计和开发人员的迫切需要。

然而，尽管Visual C++使用了C/C++语言，由于Windows应用程序和过去所熟悉的DOS程序在运行机制上的不同，以及可视化编程工具和常规的程序开发语言在使用上的差异，再加上Visual C++本身的博大精深，以至于不少由普通的C或C++语言转移到Visual C++开发环境的程序开发人员感到不适应。本书以具有C/C++语言基础的读者为对象，深入浅出地阐述使用Visual C++进行程序开发所需要的基本知识和技巧。

全书共分为十三章，涵盖了三个部分的内容：

第一部分包括第一章到第三章。这一部分介绍了使用Visual C++进行Windows程序开发的必备知识，其中

第一章：“Visual C++简介”介绍了Visual C++ 5.0的新增特性，Microsoft Developer Studio集成开发环境的使用和定制，以及如何从Visual C++ 5.0的在线文档中获得所需的资料和信息。通过这一章，读者可以对Visual C++及其所使用的集成开发环境有一个大致的了解，以为使用Visual C++开发应用程序打下必要的基础。

第二章：“面向对象编程与C++语言”试图分析和介绍一些在使用C++语言进行面向对象的程序开发的过程中所常见的问题，而不是对C++语言的完整的讲述。在这一章中，我们旨在通过指出一些可能引起程序出错的语言问题，来避免读者的编程的过程中犯同样的错误。

第三章：“Win32应用程序设计”讲述了使用SDK和API进行32位Windows应用程序设计的基本知识，这些知识是正确理解MFC的工作方式的重要基础。

第二部分包括第四章到第八章。这一部分介绍了如何使用Visual C++来设计Windows应用程序的用户界面，其中

第四章：“基于对话框的应用程序”以基于对话框的应用程序为例讲述MFC应用程序框架、应用程序类以及应用程序的消息映射和对话框类等关于MFC的基本概念。

第五章：“响应用户命令”讲述一般的菜单命令、工具条按钮命令和快捷键命令的响应和处理。此外，我们在本章也附带讲述了两种Windows控件——滑块控件和进度条的消息处理。

第六章：“使用Windows标准控件”讲述了几种标准的Windows控件的使用。我们并没有在这一章中涵盖了所有的Windows标准控件，而只是选取了其中一些具有代表性的来讲述，并希望通过它们来阐述使用Windows标准控件的一般方法。

第七章：“使用ActiveX控件”讲述如何使用新的ActiveX控件来增强应用程序的功能。作为示例，我们使用了功能强大的多媒体控件ActiveMovie来完成了一个可以打开多种媒体文件的视频播放器。

第八章：“文档/视结构”讲述了如何在基于文档的应用程序中使用MFC的文档/视结构，这种结构通过将文档中数据的维护与用户的接口相分离，从而使得程序的结构更加合理，更便于维护，同时也便于实现一些有用的特性，如同一文档的多个视图的同步更新等。

第三部分包括第九章到第十三章，在这一部分中，我们挑选了Windows下的一些编程课题来进行了一些一般性的讨论。其中

第九章：“图形设备接口”讲述了如何通过Windows的设备上下文来使用画笔、刷子及字体等多种绘图对象来进行图形的绘制和输出。在讲述的过程中，我们还介绍了一些很有用的编程技巧，如使用路径和剪辑区域来绘制特殊效果等。

第十章：“MFC通用类”介绍了MFC通用数据类型，它们包括集合类、字符串类和日期及时间类等。这些通用类使用MFC应用程序的数据管理和使用更加方便。

第十一章：“异常处理和诊断”介绍了如何处理程序中出现的异常事件和使用MFC的诊断服务来检查程序中的错误，这些方法对于创建健壮的和无错的应用程序十分必要。

第十二章：“多线程”介绍了32位编程中线程和进程的概念，以及如

何充分使用Win32环境下的多任务功能。

第十三章：“动态链接库”介绍了动态链接库的创建和使用，以及如何使用动态链接库来扩展MFC等。

熟悉MFC类库的内容和Win32 API中的有关函数是快速高效地进行Win32程序设计的必要条件，然而一般情况下我们不可以记住数千个函数的功能和用法，由于本书篇幅有限，在每一章中也不可能面面俱到。立足于初中级Visual C++编程人员的实际需要，作者在本书中努力介绍一些实用的编程技巧，指出一些大多数人在编程时可能犯的错误，而不是全面的概括性的讲述MFC和Windows程序设计（当然，本书中出于完整性的考虑，仍有少量的这一类的概括性文字）。这本书不是供查阅各个函数和类方法的使用的参考手册，也不是一本C++的教科书，它完全立足于Visual C++的使用者，力图将他们引入编程的乐趣中来。“大而全”不是本书的目的，“少而精”才是我们所力图追求的。我们希望通过本书的学习，不仅能够让一位对Visual C++少有所知的C++程序开发人员学会熟练的使用Visual C++进行Win32应用程序的开发，而且对于初中级Visual C++程序员，也能够从本书中找到一些值得一看的内容。因此，本书可能会对MFC中的一些内容略去不谈，但有时候可能会因为需要实现一些有趣的特性而深入到MFC的内部或者绕过MFC而直接使用Windows API，这些都体现我们在实际编程中所遇到的真实情况——应用程序的需求有可能多种多样，其实现方式也不可能是千篇一律。完整地介绍Visual C++的方方面面不可能同时也不应该是本书的宗旨，本书的任务在于教会读者如何在Visual C++中使用以“我”为中心的方式来开发Windows应用程序。

在本书中使用的示例代码都进行了精心的选择，细心的进行了编写和调试，这也导致了本书的完稿时间一拖再拖。这里，我要感谢本书的编辑，是他一次又一次的容忍我使用相同的理由来推迟交稿的时间。此外，还需要感谢和我一同具有我正在使用的这台计算机的几位伙伴，是他们我才得以以“独占”方式使用所有的系统资源来完成本书。

必须感谢两位朋友的帮忙，否则这本书即使推迟交稿也不可能完成。在最后的紧要关头，刘斌编写了本书的第十章、第十一章和第十二章，赵仕健编写了本书的第五章和第八章，此外，他还替我修改了第四章，并增加了一些很有用的内容，可惜的是，他的一些优秀的工作成果，毁于一次意外事故中，在那次意外中，位于硬盘的数据，一夜之间便消失得无影无踪了。

尽管我们细致的对书中的每一行程序进行了反复的调试，但是仍有可

能在最后的一刻还包含着被未被虑及的问题。此外，在将Developer Studio中的过程粘贴到Word文档的过程中，也有可能出现不该有的笔误（确切的说是敲错了键）和疏漏。我们努力避免发生这种情况，但是，即使它的概率为零仍有可能发生，这就需要读者来批评指正了。

作者

一九九八年七月

# 目 录

## 前言

## 第一章 Visual C++ 简介

- 第一节 Visual C++和MFC的历史 \*
- 第二节 Visual C++ 5.0的版本及新特性 \*
- 第三节 Visual C++ 5.0的新特性 \*
- 第四节 Developer Studio的使用 \*
- 第五节 获得帮助 \*
- 第六节 自定义Developer Studio \*

## 第二章 面向对象编程与C++语言

- 第一节 面向对象的编程技术 \*
- 第二节 类的声明和定义 \*
- 第三节 类的继承 \*
- 第四节 多态与虚函数 \*
- 第五节 ClassView和WizardBar \*

## 第三章 Win32应用程序设计

- 第一节 事件驱动的应用程序 \*
- 第二节 Win32 API和SDK \*
- 第三节 使用SDK编写Windows应用程序 \*
- 第四节 32位编程的特点 \*

## 第四章 基于对话框的应用程序

第一节 使用AppWizard生成应用程序框架 \*

第二节 应用程序类 \*

第三节 MFC应用程序的消息循环 \*

第四节 对话框类 \*

第五节 小结 \*

## 第五章 响应用户命令

第一节 菜单消息响应 \*

第二节 工具条 \*

第三节 快捷键消息响应 \*

第四节 滑块控件消息响应 \*

第五节 进度条消息响应 \*

第六节 上下控件消息响应 \*

## 第六章 使用Windows标准控件

第一节 使用对话框编辑器和ClassWizard \*

第二节 所有窗口类的基类：CWnd \*

第三节 按钮 \*

第四节 静态控件 \*

第五节 文本编辑控件 \*

第六节 列表框控件 \*

第七节 组合框 \*

第八节 滚动条控件 \*

## 第七章 使用ActiveX控件

第一节 什么是ActiveX控件 \*

第二节 使用ActiveXMovie控件的视频播放器 \*

## 第八章 文档/视结构

第一节 文档/视结构概述 \*

第二节 使用AppWizard创建框架应用程序 \*

第三节 生成文档 \*

第四节 生成视 \*

第五节 视类 \*

第六节 同一文档的多个视 \*

第七节 添加对多文档类型的支持 \*

## 第九章 图形设备接口

第一节 设备上下文 \*

第二节 画笔对象 \*

第三节 刷子对象 \*

第四节 字体对象 \*

第五节 映射模式 \*

## 第十章 MFC通用类

第一节 数组类 \*

第二节 列表类 \*

第三节 映射类 \*

第四节 字符串类 \*

第五节 日期和时间类 \*

## 第十一章 异常处理和诊断

第一节 处理C++异常 \*

第二节 MFC异常 \*

第三节 诊断服务 \*

## 第十二章 多线程

第一节 创建线程 \*

第二节 线程间通信 \*

第三节 线程同步 \*

## 第十三章 动态链接库

第一节 概述 \*

第二节 创建和使用动态链接库 \*

第三节 使用动态链接库扩展MFC \*

## 附表1 MFC类库层次表

## 附表2 ASCII码表 (0~127)

## 附录3 虚拟键码



# 第一章 Visual C++ 简介

只要提到在Windows 95和Windows NT下进行32位的应用程序开发，就不能不提到Visual C++。相比其它的编程工具而言，Visual C++在提供可视化的编程方法的同时，也适用于编写直接对系统进行底层操作的程序，其生成代码的质量，也要优于其它的很多开发工具。随Visual C++所提供的Microsoft基础类库(Microsoft Foundation Class Library，简称为MFC)，对Windows 95/NT所用的Win32应用程序接口(Win32 Application Programming Interface)进行了十分彻底的封装，这使得可以使用完全的面向对象的方法来进行Windows 95/NT应用程序的开发，从而大量的节省了应用程序的开发周期，降低了开发成本，也使得Windows程序员从大量的复杂劳动中解救出来，相信随着对Visual C++了解的逐步深入，你会亲自感受到这一点。Visual C++使Windows编程不再深奥和晦涩，而是一件有意义并且有趣的事情，而且，你并没有因为获得这种方便而牺牲应用程序的性能。

在本章中，我们将讲述：

- Visual C++和MFC历史
- Visual C++ 5.0的不同版本和它们的区别
- Visual C++ 5.0的新特性
- Visual C++ 5.0集成开发环境的使用

## 第一节 Visual C++和MFC的历史

Visual C++的核心是Microsoft基础类库，即通常所说的MFC。尽管使用Visual C++进行编程并不一定要使用MFC，使用MFC也不一定就要使用Visual C++，Borland C++的新版本也提供了对MFC的支持，然而事实上，在很多情况下，我们提到Visual C++时指的就是MFC，而提到MFC时指的也就是Visual C++。因此，当你看到关于Visual C++或是MFC的资料时，要知道，在绝大多数情况下，它们都是指同一样东西。

MFC相当彻底的封装了Win32软件开发工具包(Software Development Kit，即通常所说的SDK)中的结构、功能，它为编程者提供了一个应用程序框架，这个应用程序框架为编程者完成了很多Windows编程中的例行性工作，如管理窗口、菜单和对话框，执行基本的输入和输

出、使用集合类来保存数据对象等等，并且，MFC使得在程序中使用很多过去很专业、很复杂的编程课题，如ActiveX、OLE、本地数据库和开放式数据库互联(Open Database Connectivity，简称为ODBC)、Windows套接字和Internet应用程序设计等，以及其它的应用程序界面特性，如属性页(也叫标签对话框)、打印和打印预览、浮动的和可定制的工具条变得更加的容易。

早在1989年，Microsoft的程序员们开始试图将C++和面向对象的编程概念应用于Windows编程中，以编写出一个可以使Windows编程更加简便的应用程序框架。他们把这个应用程序框架叫做AFX (AFX这个词来源于Application Framework，但奇怪的是这个词组中并没有包含“X”这个字母)。直到今天，AFX小组早已不存在了，AFX这个名称也于1994年初不再使用，但在Visual C++和MFC中，AFX的影子却随处可见，很多全局函数、结构和宏的标识符都被加上了AFX的前缀。

最初的AFX版本在经过一年的艰苦之后诞生，却未能被大多数Windows程序员所接受。AFX的确是经过了精心的规划和编码，并且，它也提供了对Windows API的高度抽象，建立了全新的面向对象的AFX API，但最要命的是AFX API库根本不兼容于现有的Windows API。由此导致的最严重后果是大量的SDK代码无法移植，而程序员将学习两种完全不同的编程方法。

AFX不得不重新做所有的一切，他们所创建的新的应用程序框架是一套扩展的C++类，它封装和映射了Windows API，这就是MFC的前身。过去的AFX小组也变成了MFC小组。最终，MFC的第一个公开版本于1992年3月随Microsoft C/C++ 7.0 (而不是Visual C++ 1.0)一起推出。那时距Windows 3.1发布尚有好几个月。在MFC 1.0中还没有文档/视结构，但有类CObject和CArchive。在12个月之后，MFC 2.0随Microsoft新的编程工具Visual C++ 1.0一道出炉。与MFC 1.0一样，MFC 2.0仍是16位的，因为32位的Windows NT 3.1直到1993年7月才问世。在MFC 2.0中，增加了对文档/视结构、OLE 1.0、Windows 3.1公用对话框的支持和消息映射等。在Windows NT 3.1面世一个月以后，Microsoft推出了32版本的Visual C++和MFC 2.1，它实际上是MFC 2.0的Win32接口。

最后一个16位的Visual C++编译器是1993年12月推出的Visual C++ 1.5，直到今天，一些为Windows 3.1编写16位应用程序的程序员还在使用这个版本。1994年9月，32位的MFC 3.0伴随着Visual C++ 2.0的一道面市，在今天的计算机图书市场上，还有着的大量的关于Visual C++ 2.0和MFC 3.0的图书出售，因此，你可以想象得出Visual C++ 2.0所取得的成功和它所产生的影响。并不象你预想的那样，在

Visual C++ 5.0中包括的MFC版本不是MFC 5.0，而是MFC 4.21。发展到今天，MFC已发展成一个稳定和涵盖极广的C++类库，为成千上万的Win32程序员所使用。MFC库是可扩展的，它和Windows技术的最新发展到目前为止始终是同步的。并且，MFC类库使用了标准的Windows命名约定和编码格式，所以有经验的Windows SDK程序员很容易过渡到MFC。MFC结合了Windows SDK编程概念和面向对象的程序设计技术，从而具有极大灵活性和易用性。

## 第二节 Visual C++ 5.0的版本及新特性

Visual C++ 5.0是Microsoft于1997年4月推出的最新的Visual C++编译器，它包括三个版本。各个版本之间的区别如表1.1所示：

### 第三节 Visual C++ 5.0的新特性

如果你没有使用过Visual C++ 4.x，或者虽然使用过Visual C++ 4.x，但对它还不是很熟悉，那么你可以跳过这一节的内容，继续阅读本书的其它内容。在这一节里列举了Visual C++ 5.0中新增的所有特性，这些内容对熟练掌握了Visual C++ 4.x的程序员来说是很 useful 的，他们可以通过阅读本节了解到Visual C++ 5.0所做的改进，从而知道自己应该补充的是哪一方面的内容。而对于Visual C++ 的初学者和刚入门者来说，了解这些内容就不是那么有必要了，尤其是当你急切地想进入Visual C++ 5.0的编程实践中去的时候。

表1.1 Visual C++ 5.0的不同版本

版本	特点
学习版 (Learning Edition)	除了代码优化、剖析程序(一种分析程度的运行时行为的开发工具)和到MFC库的静态链接外，Visual C++ 5.0学习版提供了专业版的其它所有功能。学习版的价格要比专业版本低很多，这是为了使希望使用Visual C++ 5.0来学习C++语言的个人也可以负担得起。但你不可以使用Visual C++ 5.0学习版来开发供发布的软件，其授权协议明确禁止这种做法。
专业版 (Professional Edition)	Visual C++ 5.0可用来开发Win32应用程序、服务和控件。在这些应用程序、服务和控件中可使用由操作系统提供的图形用户界面或控制台API。
企业版 (Enterprise Edition)	可用来开发和调试为Internet或企业内网(intranet)设计的客户-服务器应用程序。在Visual C++ 5.0企业版还包括了开发和调试SQL数据库应用程序和简化小组开发的开发工具。

下面我们分版本来讲述Visual C++ 5.0相对于上一个版本所新增加的内容：

(1) 专业版

在Visual C++ 5.0专业版中包括如下的新增特性：

C++语言

- 新增下列C++关键字：bool、explicit、false、mutable、true和typename。
- 允许使用\_\_declspec来声明指定的存储类属性是应用于某一类型还是某一类型的一个变量。

编译器、链接器和NMAKE

- 编译器添加了对COM的支持，从而简化了使用COM对象的C++客户的开发。为了演示如何使用该特性来支持COM，新增了如表1.2所示的示例程序。

表1.2 为演示编译器对COM的支持而新增的示例程序

示例程序	演示内容
ACDUAL	MFC应用程序中双界面的支持
INPROC	进程内自动化服务器
MFCCALC	使用自动化服务器实现的一个简单计算器
COMEXCEL	单独运行的自动化客户程序。该程序创建一个新的Microsoft Excel电子表格，并生成饼图。
COMIDE	单独运行的自动化客户程序。该程序自动操纵Microsoft Developer Studio

续表1.2

示例程序	演示内容
COMMAIL	单独运行的自动化客户程序。该程序为Microsoft Exchange 4.0自动操纵了MAPI
COMMAP	不同COM接口入口映射宏的使用

LABRADOR	ATL的使用。该程序实现了一个没有用户界面的EXE服务器
FRETHREAD	使用编译器的COM支持编写多线程客户程序和自由线程服务器
ALLINONE	MAC、STL、ATL和COM的使用

- 使用/O1选项编译生成的代码大小将比Visual C++ 4.2版小5%到10%。
- 链接器使用了/FIXED选项来创建更小的供发行的应用程序。因此，在使用剖析程序时，由于需要重定位信息，链接器必须使用/PROFILE和/FIXED:NO选项。这同样适用于其它如BoundsChecker或Purify之类的链接后(post-link)工具。
- 新增的/EH编译选项可以更有效的控制C++异常处理。C++同步异常处理允许编译器生成更小的代码，因此它是Visual C++ 5.0新的默认C++异常处理模式。
- 对用来控制代码优化所面向的编译器的编译器选项/G3、/G4、/G5、/G6和/GB作了修改。
- 将/GX编译器选项映射为/EHsc。
- 允许使用链接器选项/PDBTYPE指定包括调试信息的程序数据库(PDB)。该选项可以节省磁盘空间并加快链接。
- 在NMAKE中支持批处理规则。

## AppWizard

- 新的AppWizard可以自动管理基于对话框的应用程序中的对话框类。只需要简单的创建一个基于对话框的应用程序，并选择对自动化的支持，就可以象早期版本的AppWizard一样，得到一个支持基本自动化的基于对话框的应用程序。通过单独的代理类，对话框类也可以通过自动化导出。你可以添加方法和属性来导出对话框中的元素。
- 定制的AppWizard可以改变工程创建时的设定。例如，你可以在目标创建之后调整编译器、链接器和查看设定或者添加定制创建步骤。

## MFC

- asynchronous (URL) moniker允许应用程序异步的下载文件和控件属性，以便在任务完成后为其它进程释放系统资源。
- 可以在Web浏览器(如Internet Explorer 3.0)或支持ActiveX文档的OLE容器(如Microsoft Office Binder)的整个客户区显示活动的文档。
- Win32 Internet API (WinInet)使Internet成为任意应用程序的一个完整部分并简化了Internet服务，如FTP、HTTP和gopher的访问。
- 增加了对DAO 3.5的支持。
- 增加了对ODBC 3.0的支持，并对MFC ODBC类作了几个重要的修改。
- COleDateTime成员函数SetDate、SetDateTime、SetTime的返回值从BOOL改变为int。每一个成员函数当COleDateTime对象被正确设置时返回0，否则返回1。该返回值基于DateTimeStatus枚举类型。
- 新增示例程序IMAGE。该程序生成一个可以异步下载数据的ActiveX控件。

## Active Template Library (ATL) 2.1

- ATL 2.1版支持创建既小又快的ActiveX控件。

## C Runtime Library

- 新增的函数\_itoa、\_i64toa和\_ui64toa将数据转换为一个以null结尾的字符串。所对应的宽位字符版本为\_itow、\_i64tow和\_ui64tow为\_itoa、\_i64toa和\_ui64toa。
- 改善了下列的通用浮点超越函数的性能：pow、sqrt、log、log10、sin、cos、tan、asin、acos、atan。
- 改善了内存移动和内存拷贝函数的性能。

## ANSI标准C++库

- Visual C++的标准库遵从1996年9月24日公布的ANSI C++ (X3J16) 工作单——ANSI Doc No. X3J16/96-0178 WG21/N0996。该标准于1996年7月在Stockholm会议上制定。

## OLE DB

- OLE DB是一组OLE接口，它使应用程序可以以统一的方式访问保存在不同信息源中的数据。这些接口支持适合于数据源的大量数据库功能性，并允许数据源共享其数据。所配套的OLE DB软件开发工具包所提供的一组软件部件、工具和文档可以在开发OLE DB客户和提供程序提供帮助。

## ERRLOOK工具

- ERRLOOK工具可以使用系统错误的值来检索相应的错误消息，其中包括OLE HRESULT。错误值可以通过包括拖放、编辑命令等的多种方法给出。由ERRLOOK所返回的错误消息文本可以复制并粘贴到其它应用程序中。

除了以上新增特性之外，在Visual C++ 5.0光盘上的\DEVSTUDIO\VC\SAMPLES目录下还包括了一些新增的示例程序。

## (2) 企业版

除包括专业版中的所有特性外，Visual C++ 5.0企业版还包括下列特性以支持企业级应用程序的开发：

- Microsoft Transaction Server (Microsoft事务服务器)用于创建基于事务的应用程序。
- Visual Database Tools (可视化数据库工具)提供了数据库和SQL查询的图形化设计。
- 调试数据库连接时具有更好的性能，该进程将比过去快上很多。
- 扩展的SQL数据类型支持使你可以方便的将本地变量变为除text和image外的所有SQL数据类型。这种变换包括money类型和datetime类型。可以在本地变量和NULL值之间相互变换。还可以方便的查看包括text和image在内的所有SQL数据类型。
- 在光盘上的\DEVSTUDIO\VC\SAMPLES目录下包括了特定于企业版的新增示例程序。

### (3) 学习版

Visual C++ 5.0学习版包括了学习C/C++和使用MFC、OLE、ODBC、DAO、ActiveX和COM的各种工具，但不包括下面的特性：

- 到MFC的静态链接  
使用Visual C++ 5.0学习版编写的应用程序只能在运行时链接到MFC动态链接库。
- 代码优化  
不能使用/O选项来生成更小和更快的代码。
- 程序剖析  
不能使用剖析程序来分析程序代码中的某一部分是否可以从性能改进中获益。
- RemoteData控件及其它的数据绑定控件

除了上面的内容外，Visual C++ 5.0学习版包括专业版中的其它新特性。

### (4) 集成开发环境

Microsoft Developer Studio用于Visual J++ 1.1、Visual InterDev、Visual C++ 5.0和MSDN。新的Developer Studio包括以下的新特性：

- 自动化和宏  
可以使用Visual Basic脚本来自动操纵例行的和重复的任务。可以将Visual Studio及其组件当作对象来操纵，还可以使用Developer Studio对象模型创建集成的附加程序。
- ClassView  
使用文件夹来组织C++和Java中的类，包括使用MFC、ATL创建或自定义的新类。
- 可定制的工具条和菜单
- 连接到正在运行的程序并对其进行调试，还可以使用宏语言来自自动操作调试器。
- 可以在Developer Studio中查看Internet上的World Wide Web



页。

- 可以在一个工作空间中包括多个不同类型的工程  
工作空间文件使用扩展名.dsw来代替过去的扩展名.mdp，工程文件使用扩展名.dsp来代替过去的扩展名.mak。
- 改进的资源编辑器  
在Visual C++中，可以使用WizardBar来将代码与程序中的可视元素挂钩。  
快捷键、二进制、对话框和字符串编辑器支持定位至快捷键、ASCII字符串、十六进制字节串、控件ID和标签及指定字符串的Find命令  
更方便的一次修改多个项(可以快捷键、对话框、菜单和字符串)的属性。
- 改进的文本编辑器  
可以使用正确的句法颜色设置来显示无扩展名的头文件。  
可以定制选定页边距的颜色来更好的区分同一源代码窗口中的控件和文本区域。  
Find in Files命令支持两个单独的窗格。
- 上下文相关的What ' s This帮助
- 改进了的WizardBar  
可用于Visual J++。
- 新增的向导  
添加了集成到Visual J++和Visual InterDev中的新增向导。

## 第四节 Developer Studio的使用

在一整套的Visual Studio 97中，Visual C++ 5.0、Visual J++ 1.1和Visual InterDev都使用同一个开发环境，称作Developer Studio。你将在Developer Studio中创建所开发的应用程序的源文件、各种资源文件及其它文档。这些文件以工作空间和工程的形式进行组织。Developer Studio中一次只能打开一个工作空间，在同一个工作空间中可以包括多个工程，一般来说，每一个工程你所开发的一个应用程序。这些工程相互之间可以具有某个联系和从属关系，也可以彼此完全独立。此外，这些工程还可以是不同类型的，比如说，你可以在一个已经包括有一个Visual C++工程的工作空间中添加一个Visual J++或是Visual InterDev工程。工程中除了包括了应用程序

所用到的源文件、资源文件外，还可以包括其它类型的文件，如应用程序的规格说明书、流程图、开发日程等等。对于那些由ActiveX部件(如Microsoft Word等)所创建的ActiveX文档，可以在Developer Studio中直接打开。而对于那些与其它类型的应用程序相关联的文档，你也可以通过Developer Studio在独立的窗口的打开。

Developer Studio所包括的内容是很丰富的。本节只讲述一些基本的概念和用法。这些用法是进行下一步所不可缺少的。对一些特殊的用法，我们将在本书后面的章节中需要用于再作说明。

由于本书假定你已经是一个熟练的Windows 95/NT用户，因此，安装Visual C++的过程对于你来说应该是一个相对很容易的事，所以我们不再在这个问题上浪费时间。下面的过程中，我们假定你已在你的计算机中安装上了Visual C++ 5.0，在一般情况下，这同时也安装了Developer Studio，并以它作为Visual C++ 5.0的集成开发环境。

这时，从你的开始菜单中运行Visual C++ 5.0，屏幕显示应该如图1.1所示。图1.1还标注了Developer Studio中的几个常提到的部件的名称，以使得，当以后我们提到这些名词时，你可以知道它们所指的是Developer Studio中的哪一部分。

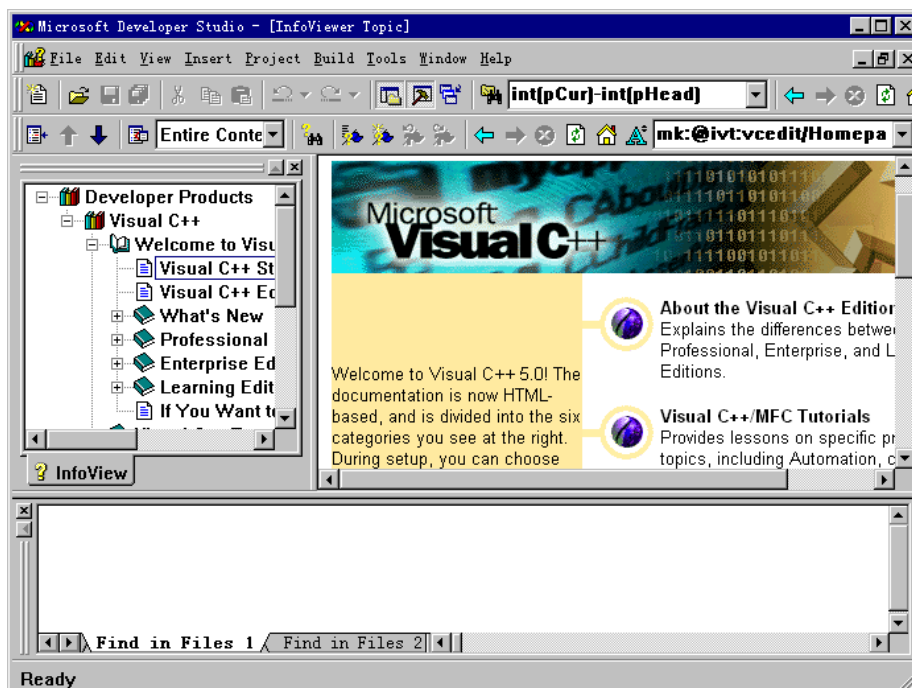


图1.1 Microsoft Developer Studio - Visual C++ 5.0的集成开发环境

当然，你所看到的内容和工具条等也许会有少许的不同，这取决于你的设置。下面我们来讲述其中的主要部分。我们看到，在Developer Studio，整个窗口被分成了若干个部分，需要注意的事，随着设置的

不同，或者所安装的软件包的不同，或者是处于开发的不同阶段(典型地，在输入源代码和调试程序的两个不同阶段)，你所见到的Developer Studio组件和相互之间的位置也会不一样。

前面说到过，在Developer Studio中，我们是以工作空间(workspace)和工程(project)来组织文件和进行工作的。工作空间位于这个结构的最顶层，因此，我们首先需要创建一个工作空间。创建工作空间通常有两种方法：

第一种方法是显式的创建一个空白的工作空间，然而向工作空间中添加工程。这时，我们从Developer Studio的File菜单下选择New...命令，这时出现如图1.2所示的对话框。

这时我们在Workspace name处键入工作空间的名字，这里假设为WorkSpc，则Developer Studio将在Location所指定的目录下创建名为WorkSpc的子目录(当在Workspace name处键入完工作空间名后，可以在Location处修改这个默认设置)，然后以WorkSpc.dsw的文件名将该工作空间保存到这个目录下。

第二种创建工作空间的办法是直接创建一个工程。创建一个新的工程同样是选择File菜单下的New...命令，在类似于图1.2的对话框中单击Project选项卡，如图1.3所示。然后在该对话框中选择Create new workspace单选钮(这是Developer Studio的默认选项)。注意在图1.3所示的对话框中，Project name和Location的意义和图1.2中的Workspace name和Location的意义类似。这样，在创建工程时，Developer Studio将创建一个同名的工作空间。然后将所创建的工程添加到该工作空间中。

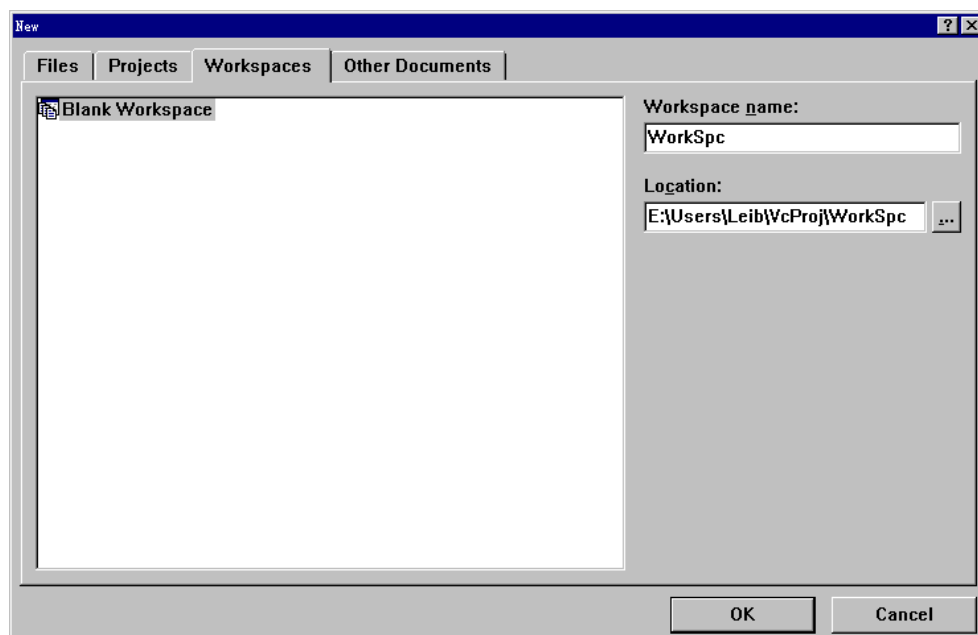


图1.2 创建空白工作空间

如果仅在工作空间中包括一个工程的话，使用第二种方式显然是很合理的，并且，也要比使用第一种方式创建空白工作空间，然后再在空白工作空间中添加工程的方法要简洁和方便。在今后的很多情况下，我们都将使用第二种方式来创建工程和包括工程的工作空间。但并不是说第一种方式就没有用处了。事实上，在第二种方法中，Developer Studio将工作空间和工程保存到Location所指定的同一个目录下，这对于单个工程的工作空间是合理的。但如果你希望在工作空间中包括多于一个的工程的话，你也许希望在保存工作空间的目录下新建子目录来保存这些工程，因为这样更有条理，更利于文件的管理。这时，我们就需要使用第一种方式来创建空白工作空间，然后再在这个工作空间中新建和添加工程。

在工作空间中新建工程的方法和上面的第二种方式几乎一样。只不过这时我们应该在图1.3所示的对话框中选择Add to current workspace（在图1.3中，这个单选钮是灰的，这是因为当前并没有打开的工作空间的缘故）。要注意这时Location处的目录名是基于当前工作空间所在的目录的。单击OK后，Developer Studio根据在Project name处所键入的工程名以.dsp的扩展名来保存该工程文件。

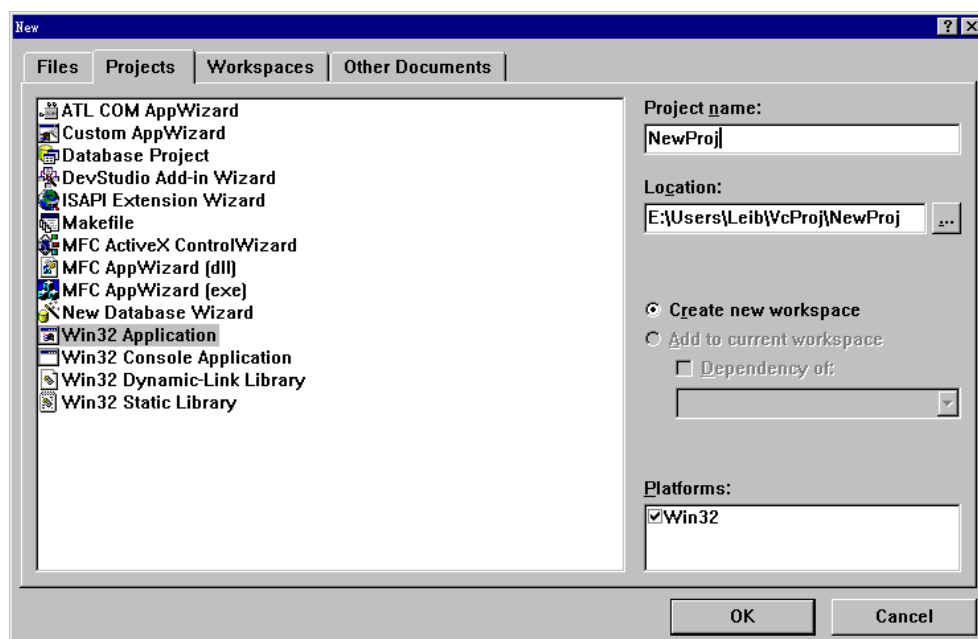


图1.3 创建一个新的工程

- 注意：
- 本节不讨论那些由Developer Studio和所集成的开发包(如Visual C++、Visual J++和Visual InterDev)创建和管理的文档，如源代码文件、资源文件等。对于使用这些文件的方法，我们将在本书

的其它章节中在需要用到时候进行讲述。

除了向工程中添加由Developer Studio和相应的开发包(如Visual C++、Visual J++和Visual InterDev)管理的文件外，还可以添加其它类型的文档，这些文档包括分成两类，一类由ActiveX部件创建和维护，另一类由其它的软件创建和维护。Developer Studio在编辑这些文档时的行为是不同的。

对于由ActiveX部件(最常见的ActiveX部件有Microsoft Word和Excel等，但是，这里所指的ActiveX部件并不限于Microsoft的产品，其它任何符合ActiveX部件标准的应用程序都是ActiveX部件)创建的文档，你可以在Developer Studio窗口内部打开并编辑它们，这时，由该部件提供的菜单项融合进了Developer Studio原有的菜单项，由该部件所提供的工具条取代了Developer Studio原有的工具条。并且，所打开的文档显示于原有的InfoViewer Topic窗口所在位置(如图1.4所示，在这幅图中，我们向工程Project1中添加了一个新建的Microsoft Word文档，并在Developer Studio内部打开并编辑该文档)。这样，你无需离开Developer Studio就可以查看和修改这些文档，这就是ActiveX技术所带来的巨大方便之处。

向工程中新建这类文档只需在图1.2和图1.3所示的对话框中选择Other Document选项卡，然后指定新建的文档的类型，并给出文档文件名即可(对于向工程中添加的文档，必须指定文件名，如果只是在Developer Studio中编辑该文档，则不受此限)。如果是工程中添加已有的文档，则必须保证这些文件的扩展名与文档的类型相符合，因为Developer Studio是根据相应的文件扩展名来判断文档的类型和寻找创建和维护该文档的ActiveX部件的。如果添加由其它非ActiveX部件的软件创建和维护的文档，也必须遵从这个约定。对于非ActiveX部件的软件创建和维护的文档，在Developer Studio选择打开时，Developer Studio将在另一个单独的窗口中打开该文档以供用户进行编辑。

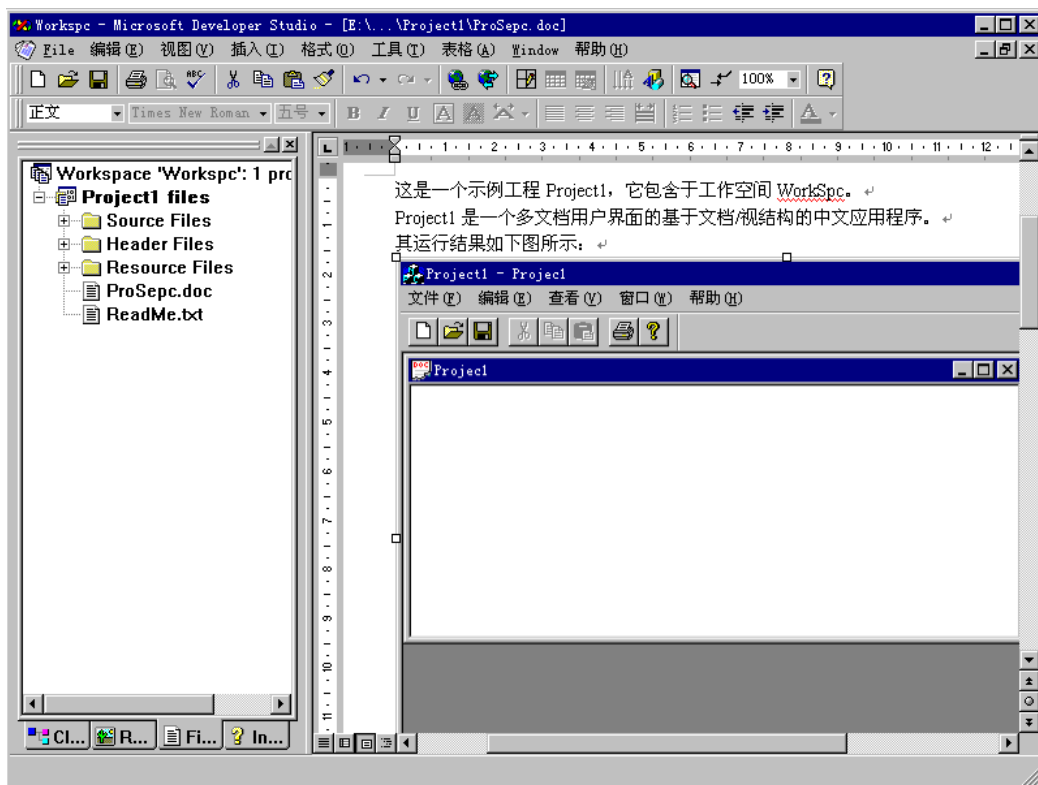


图1.4 向工程中添加并编辑Word文档

在Developer Studio中的另一个重要的部分是Workspace窗口(请参见图1.1),对于Visual C++这个窗口一般包括四个选项卡(随着是否打开工程及打开工程的种类的不同,你将看到的选项卡也许会少于四个,请对比图1.1和图1.4):ClassView、ResourceView、FileView和InfoView,分别用来查看工程的类信息、资源信息、文件信息和在线帮助文档。不论是否存在当前打开工程,InfoView总是存在的,并且,可以通过InfoView得到当前所有安装的开发工具包的在线文档。比如你当前正在开发Visual C++程序,但你一样可以查阅Visual J++的在线文档。

Output窗口用来显示各种输出信息,如编译和链接信息、调试信息以及查找信息等。这些内容将在我们具体用到时再作讲述。

## 第五节 获得帮助

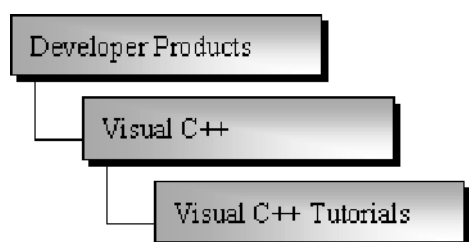
随着应用程序越来越复杂,程序员所需掌握的知识与越来越多,然而,在很多情况下,我们几乎没有可能把所有的知识都记到大脑里。就以Visual C++ 5.0为例,各种在线文档加到一起,足足有100多兆。因此,我们无法想象,如果没有在线文档的帮助,开发应用程序的过程如何进行。对于一个开发工具来说,在线文档做到是否完善,是否易于使用,成为衡量一个开发工具是好是坏的一个重要标准。值得庆幸的是,Visual C++ 5.0在这方面堪称表率,其在线帮助文档覆

盖从最基本的C语言语法到最深奥和最新的各种编程技术，几乎无所不有；并且，从全面而详尽的参考资料，到系统的编程技术，再到各种各样的示例程序和教程，足以满足各个不同层次的编程者的需要。然而，我们却常常听到这样的报怨：“Visual C++ 5.0的在线文档又多又杂，我都快弄清头绪了。”的确是这样，Visual C++ 5.0的在线文档在某些方面有些类似于Windows 95/NT的注册表，在如此庞杂的文档中寻找某一个特定主题的资料，对于编程者，尤其是则接触Visual C++的初学者来说，并非是一件很容易的事。因此，如何才能最有效的利用Visual C++的联机帮助，无疑是一个很值得探讨的课题。这也是本书的一大特色。在本书中，我们对所讲述到的每一个内容，都在该章节的末尾给出了在Visual C++ 5.0在线文档中的相关节点和可以进一步参考的内容。这样，通过阅读本书，你不但可以获得本书中已讲述的这些知识，还可以通过本书所给的线索，从Visual C++的档案库中提取出所需要的各种文档，这些文档加起来，会是本书篇幅的很多倍。可以这样说，从在线文档中快速地寻找到所需的各种资料的技能，和你从本书中学到的编程技术方面的知识同等的重要，因为任何一本几百页的书都不可能完整的包括Visual C++的所有内容，而几乎每一个实用的应用程序都会使用到一些特殊的编程，获得这些特殊的编程所需要资料的唯一来源则是联机帮助中所包括的各种技术文档。

因此，在学习使用Visual C++进行应用程序设计之前，先学习一下如何从Visual C++的集成开发环境Developer Studio中获得帮助是很有必要的。

### 1.5.1 使用InfoView

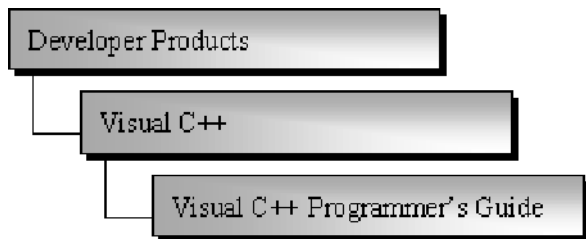
前面已经提到，在Workspace窗口的InfoView选项卡窗格中包括了Visual C++中每一份在线文档的树状结点列表。很多情况下我们正是从InfoView入手，来一步一步地查找到所需要的各种资料的。因此，我们有必要简单地了解InfoView中的结点的组织形式。由于InfoView中的树状结点的结构相当复杂，我们这里仅列举一些编程者经常会光顾的节点。



Visual C++ Tutorials包括了六个示例教程：Scribble、Scribble



OLE Server、Container、AutoClik、Circle控件、Enroll和DAOEnroll，这些示例分别讲述了MDI文档/视应用程序、OLE服务器和容器、自动化、ActiveX控件和数据库访问等MFC编程的相关知识。其中的一些典型的实现方法具有相当的参考价值，因此，如果你有足够的时间，并且愿意阅读这些英文教程的话，相信是会有所收获的。



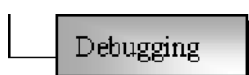
Visual C++ Programmer's Guide (Visual C++程序员参考)是最常访问的节点之一。这里提供了大多数的编程任务所需的知识。其中以下的几个子节点是最有用的：



Adding User Interface Features (添加用户界面特性)提供了完成你的应用程序界面设计所需的完整的参考。其内容涉及：窗口对象、对话框、属性页、控件、ActiveX控件、控件条(包括工具条、状态条和对话条)、工具提示、OLE (用户界面)、文档/视、剪贴板、菜单、资源编辑器、打印和打印预览、上下文相关的帮助等。



Adding Program Functionality (添加程序功能性)提供了实现特定的程序功能性所需的知识，这些知识包括：内联汇编器、调用协议、类、C++模板、Win32编程、内存管理、多线程、MFC、MFC的基础基类CObject、字符串、多字节字符集(MBCS)、Unicode编程、集合类、日期和时间数据、异常处理、文件处理、串行化、消息和命令、代码重用、动态链接库、数据库、编译器COM支持、OLE、ActiveX、自动化及远程自动化、Windows套接字、Internet支持、MAPI支持等。



每一程序员，那怕是高手和编程天才，都难免在程序包括这样或是那样的错误。如果想知道如何调试你的应用程序最有效率，最能充分发挥Visual C++和Developer Studio所提供的调试功能，那么，你需要



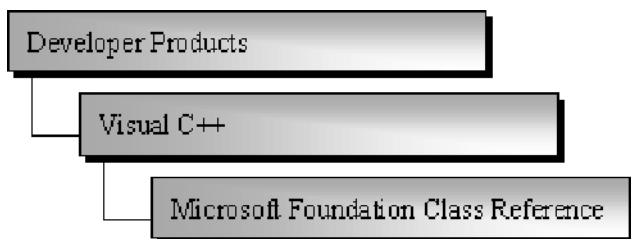
访问Visual C++ Programmer ' s Guide下的子节点 : Debugging (调试)。



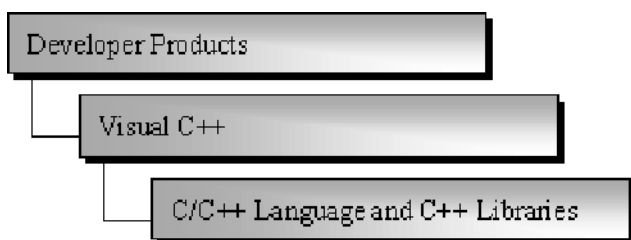
你知道了C或C++的所有内容了吗？似乎很难有人敢于给以肯定的答案。那么，你就有可能需要访问这个节点了。该节点提供了对C或C++语言的快速参考，这些说明的文字既简明，又能说明问题。当你偶尔忘了某个C/C++语言问题时，不妨垂询InfoView中的这个节点。



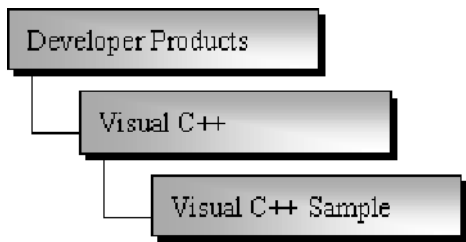
Run-Time Library Reference (运行库参考)提供了Microsoft运行库中的所有函数的参考信息。这些例程运行于Windows 95和Windows NT环境下。其中的Run-Time Routines by Category给出的库函数的分类索引，很方便查找实现某种功能所需的库函数。而Alphabetic Function Reference以字母为序给出了每一个库函数的参考信息。



Microsoft Foundation Class Reference (Microsoft基础类库参考)提供了MFC的完整的参考。除了包括所有的MFC类、全局函数、全局变量和宏外，该节点下还包括了MFC的一些技术资料。



可以把C/C++ Language and C++ Libraries (C/C++语言和C++库)当作前面的Language Quick Reference的补充，该节点对C/C++语言作了更深入和更详尽的阐述。



Visual C++ Sample (Visual C++示例)节点包括了所有的示例程序，可以通过该节点得到关于示例程序的简要说明，拷贝示例程序的代码，以及运行示例程序。但是，如果你在机器上安装了Microsoft Internet Explorer 4.0，那么，拷贝示例程序的功能不能正常使用。

#### Platform, SDK, and DDK Documentation

Platform, SDK, and DDK Documentation (平台，SDK和DDK文档)包括以下几个部分：

- Platform SDK (包括Win32 SDK中的所有文档资料)
- ActiveX SDK
- DAO SDK 3.5
- DirectX SDK
- OLE DB Programmers Reference (OLE DB程序员参考大全)

尽管这些东西看上去非常高深，但相信只要你使用Visual C++一段时间，你就会遇上一些需要查阅上面这些技术文档的问题。

### 1.5.2 使用上下文相关的帮助

最常用的一种方法是通过上下文相关的帮助快速的获得所需的信息。打开上下文相关的帮助的快捷键是F1。上下文相关的帮助可以用于多种场合：

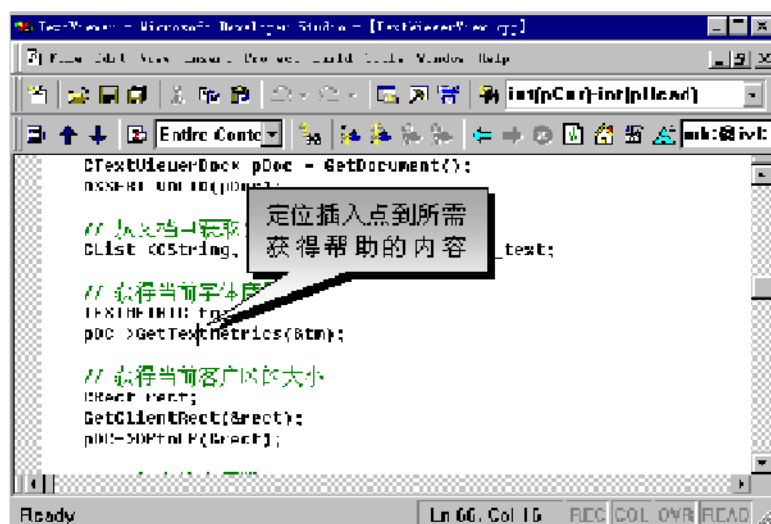


图1.5 从代码编辑器的获取上下文相关的帮助（步骤之一）

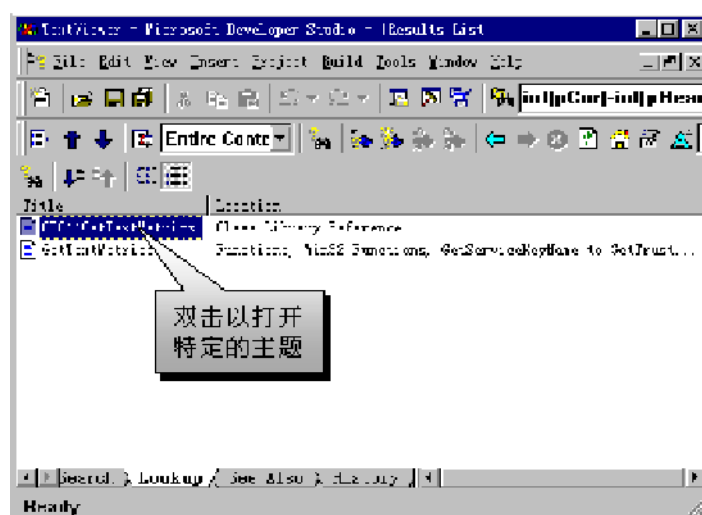


图1.6 从代码编辑器中获取上下文相关的帮助（步骤之二）

最常见的情况是从代码编辑器窗口可以获取与关键字、函数和类的相关的帮助，其步骤如下：

1. 将当前插入符定位所需获取帮助的关键字、函数或类名及类成员名，按下快捷键F1。
2. 当与指定的关键字、函数或类及成员相关的帮助条目仅有一条时，Developer Studio直接在InfoViewer Topic窗口的打开该主题，否则，将在Results List窗口中列举相符合的所有主题，双击其中Title栏下的某一项以打开相应主题。这个过程如图1.5和图1.6所示。

另外还可以在Output窗口或是在对话框中获得上下文相关的帮助，方法也是类似的。在对话框中还可以使用所谓的“ What ’ s This ” 按

钮，如图1.7所示。

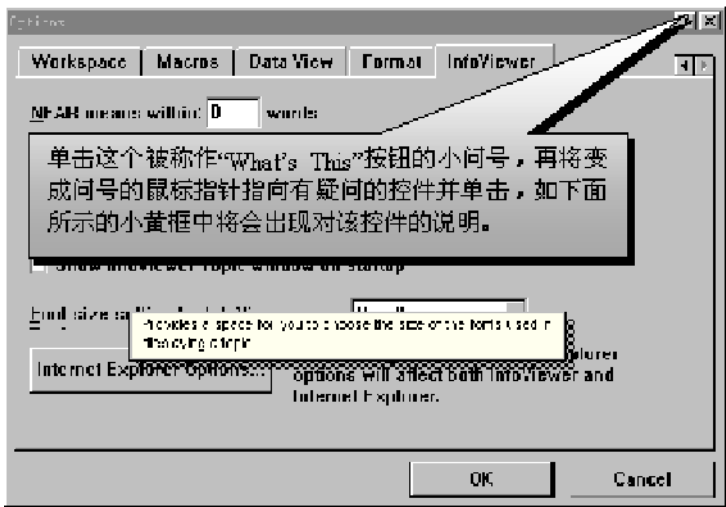


图1.7 使用 “What ’ s This ” 按钮来获取与对话框中的元素有关的信息

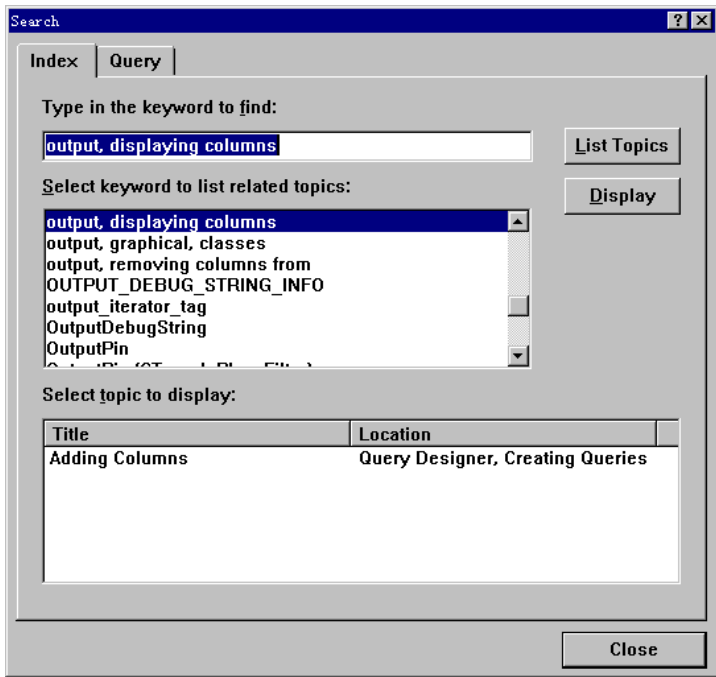



图1.8 使用Help菜单中的Search命令来查询在线文档


除了上面所说的这些方法之外，你还可以在Help菜单中选择Search命令或者单击工具条上的按钮来打开如图1.8所示的查询对话框。该对话框分成两个选项卡：选项卡Index以索引方式来查询在线文档，文档中的每一个主题都与一个或多个索引关键字相联系；反之，一个索引关键字也可能与多个文档主题相联系。如果你不知道你所需要的资料在InfoView窗格的内容列表中的节点位置，那么使用Index方式进行查询是一个很好的主意。另一个选项卡称作Query（查询）选项卡。该选项卡允许你指定查询字符串，然后从在线文档中查找匹配的所有文档，并且，你还可以指定多种不同的查找方式和范围。一般来

说，通过Query得到的结果要比使用Index的庞大得多，并且，由于Query还可以对文本，而不仅仅是标题进行匹配，因此，也许你会得到一些事实上和你所需要的资料无关的结果。这使得使用Query没有使用Index那么方便和有效。然而，Query方式也有其先进之处，在Query选项卡的Type in the word(s) to find处可以使用含逻辑运算符的查询表达式，可以使用的逻辑运行符包括AND、OR、NOT和NEAR，其中，AND、OR、NOT分别还可以简写为“&”、“|”、“!”。四个逻辑运算符的含义和示例如表1.3所示。

表1.3 在Query方式中使用逻辑运算符

运算符	示例	含义
AND	printf AND scanf	同时包括printf和scanf的匹配项
OR	printf OR scanf	包括printf或scanf的匹配项
NOT	printf NOT scanf	包括printf，但不包括scanf的匹配项
NEAR	printf NEAR scanf	在scanf周围8个字内包括printf。用于NEAR运算符的匹配范围可以在通过Tools菜单的Option命令在InfoViewer选项卡中进行设置

### 1.5.3 Developer Studio与Web

除了用来查看在线文档的窗口外，InfoViewer窗口还可以作为一个World Wide Web浏览器使用。事实上，Visual C++ 5.0的在线文档就是一系列的超文本文档。为了验证这一点，你可以在InfoViewer Topic窗口中打开一个在线文档主题，然后把工具条上的Current URL组合框(这个组合框看上去是这样的：)，如果你不能肯定哪一个组合框是Current URL的话，只需要将鼠标指针指向某一个组合框，稍待片刻，就会出现相应的工具提示。如果你能看到的工具条上的所有的组合框都不是Current URL组合框的话，你需要检查一下是否显示了InfoViewer工具条)中的内容复制到剪贴板，然后粘贴到浏览器Internet Explorer浏览器的地址框中，你可以发现浏览器也可以正常的打开该在线文档主题。如图1.9所示。反之，我们也可以在Current URL组合框中直接键入某一个Internet URL，从

而在InfoViewer Topic窗口中直接打开Internet Web页。在如图1.10所示的例子中，我们在Microsoft Developer Studio中打开了Microsoft的Visual C++技术支持主页http://www.microsoft.com/visualc。这种与网络的完整的无缝集成是Developer Studio的一大特点，并且在新的Windows应用程序中也越来越流行，从Microsoft新的操作系统Windows 98和Windows NT 5.0中我们可以很明显的看出这种趋势。相比Visual C++的前几个版本而言，Visual C++ 5.0为编写与此类似的应用程序提供更多和更好的支持，这些新添加的支持使得使用Visual C++ 5.0来编写这一类应用程序更加的方便的快捷。

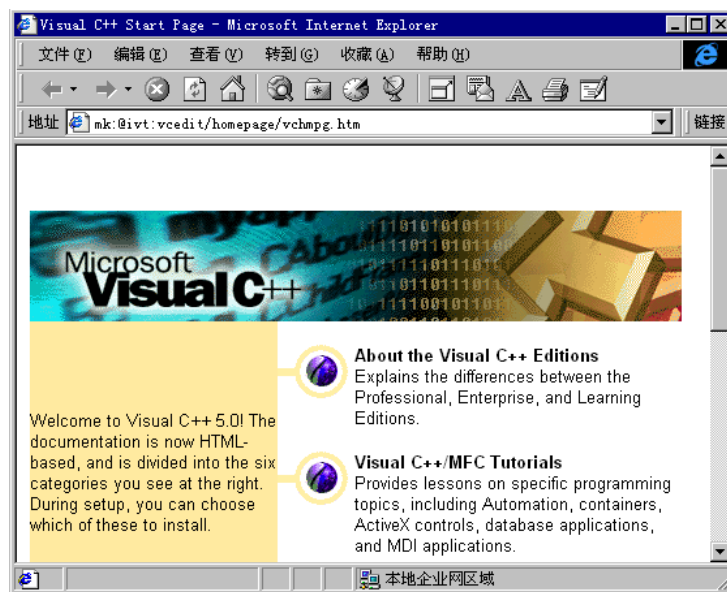


图1.9 在Internet Explorer浏览器打开Visual C++的在线文档

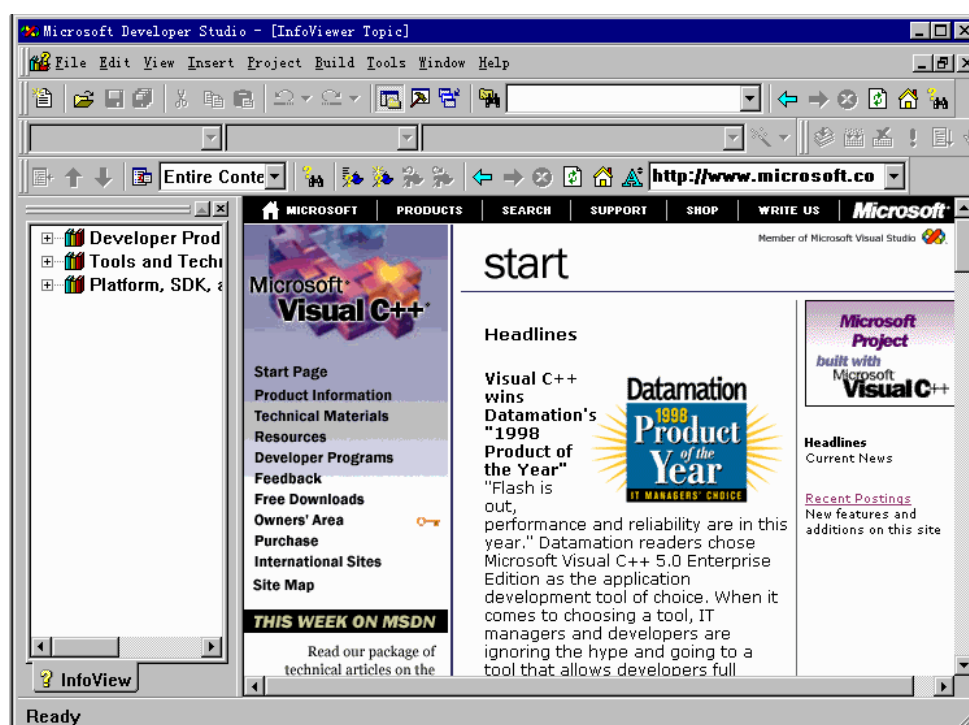


图1. 10 在Microsoft Developer Studio中打开Internet网页

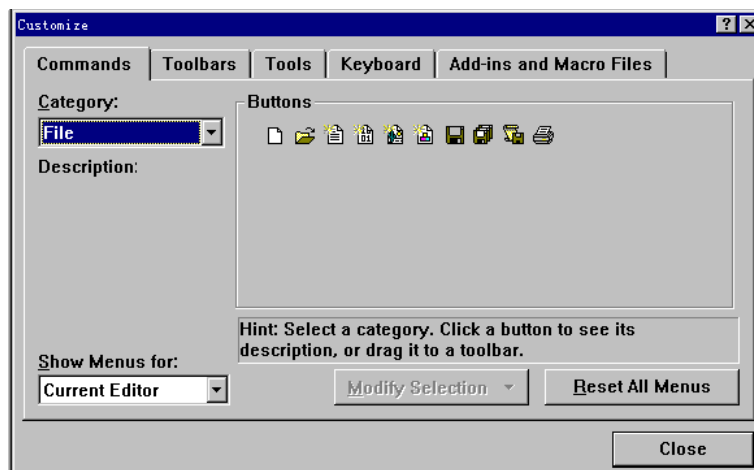


图1.11 自定义菜单项和对话框

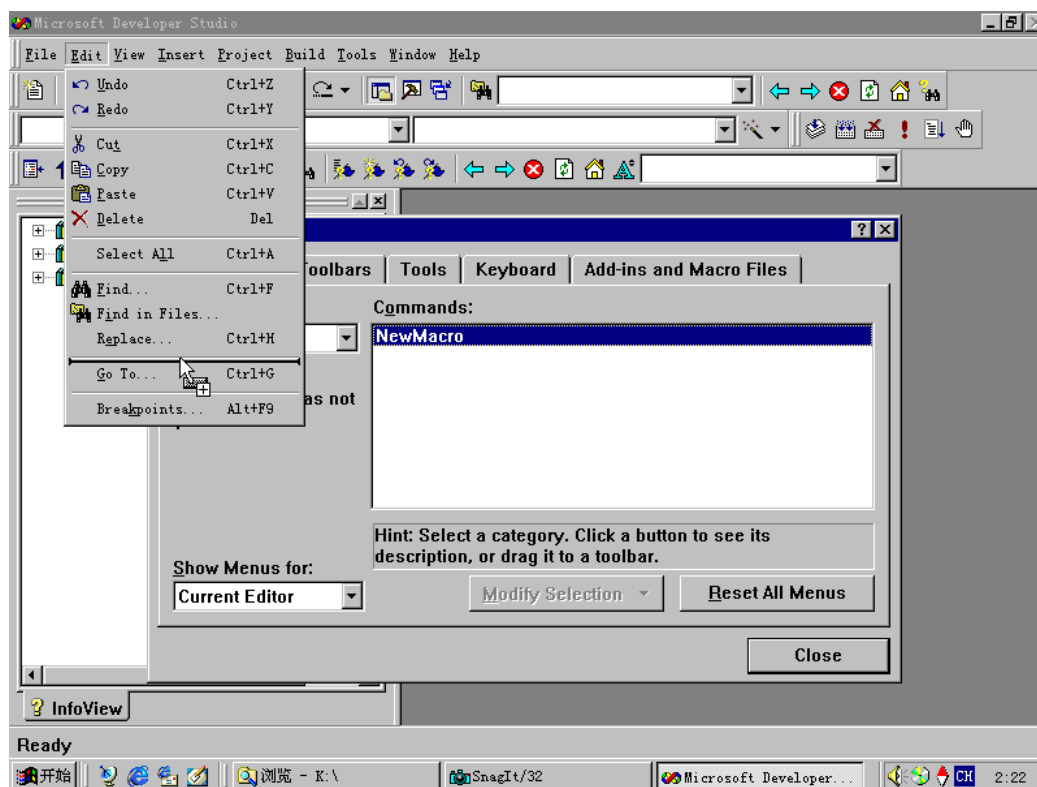


图1.12 使用拖放添加菜单项

## 第六节 自定义Developer Studio

通过对Microsoft Developer Studio进行各种自定义设置，可以更有效使用好这个开发工具。可以进行自定义的功能包括：

- 自定义工具条和菜单
- 自定义快捷键

- 使用宏

下面我们分别讲述以上几个方面的内容：

### 1.6.1 自定义工具条和菜单

在Tool菜单中选择Customize命令，单击Command选项卡，在如图1.11所示的对话框中修改Developer Studio的菜单命令的默认设置。单击Toolbars选项卡可以修改工具栏的默认设置。

这里我们假定已经编写了一个宏NewMacro，下面的示例将为宏NewMacro添加相应的菜单项和工具栏按钮。

在Category下拉列表框中选择Macro，然后将宏NewMacro直接用鼠标拖放到Developer Studio的菜单条中的适当位置。对Visual Studio 97而言，菜单项可以包括图标、文字或两者兼而有之。除了将菜单项放到一个顶级菜单项（如File、Edit等）下外，也可以直接将菜单项作为顶级菜单项。以上过程示于图1.12。

类似的方法可以用来从菜单条中删除一个菜单项。过程非常之简单，只需要把它们从菜单条中“拉”回到图1.11所示的对话框中即可。

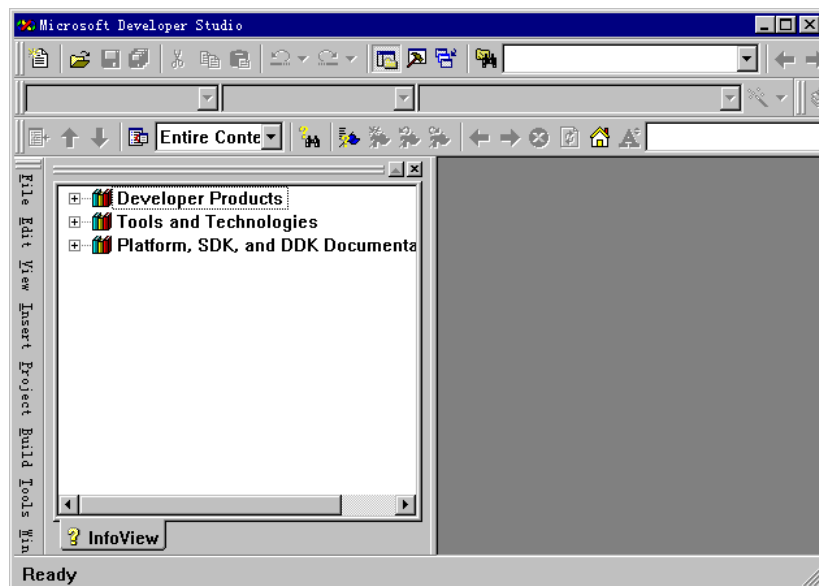


图1.13 像改变工具栏的停靠位置一样改变菜单条的停靠位置

以上的方法即简单又好用，以至于你随时都有可以把Developer Studio的菜单条改得面目全非，即使是非常有经验的用户，也有可能不小心把某一个非常之重要的菜单项拉到图1.11的窗口中。在这种情况下，如果你知道该菜单项所对应的命令，可以从Category下拉列表框中找到它所属的那一类命令，将其拉回原处即可。但是，如果你不



清楚这些命令是什么的话，就只有最后一招可以使用了，这就是放弃你所做的所有修改，把整个菜单条恢复成它原来的样子就可以了。这倒是非常之简单，只需要在图1.11所示的窗口中单击Reset All Menus按钮即可。

改变工具栏按钮的方法与此相仿。其原因非常之简单，在Microsoft Developer Studio中，菜单条事实上也是一种工具栏，若不信，你可以试一试将它拉到框架窗口的其它部分。这是完全可以的，如图1.13所示。

此外，在工具栏按钮上单击鼠标右键，将会弹出如图1.14所示的上下文菜单，从中可以选择相应的命令来改变按钮的外观，包括按钮图标及文字等。

### 1.6.2 自定义快捷键

在图1.11的对话框中单击Keyboard选项卡可以很轻松的为指定的编辑器和视的特定命令。这时，先在如图1.15所示的对话框中的Editor的下拉列表框中选择所指定的编辑器或视，然后，与自定义菜单项或工具栏的过程相似，在Category中选择命令分类，然后再选择欲设置快捷键的集合，在Enter new shortcut框中按下新的快捷键，再单击左上角的Assign按钮，即可将所按的新快捷键与相应的命令相关联，在该快捷方式所属的编辑器或视内按下这些快捷键时，相应的命令会被调用。



图1. 14 在按钮图标上右击鼠标弹出的上下文菜单

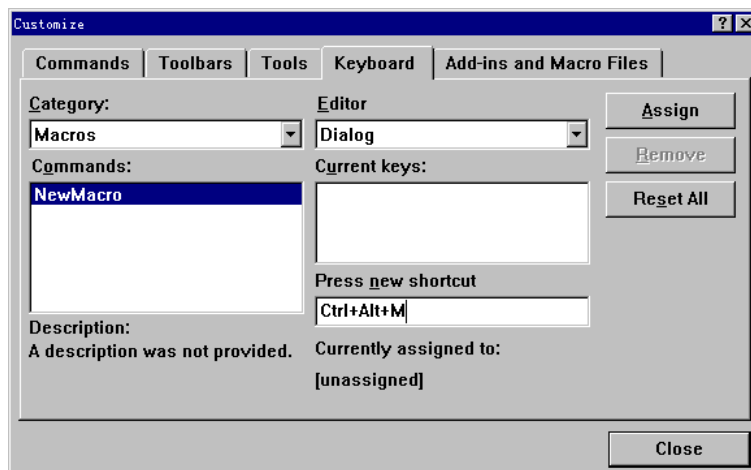


图1. 15 设置新的快捷键

### 1.6.3 使用宏

宏是Microsoft Developer Studio的新版本所提供的强大的定制工具。与通常概念中的宏不同，在Microsoft Developer Studio中，宏不再是一系列简单命令的记录和回放，而是使用了一种完整的编程语言——VBScript，VBScript使用了与Visual Basic相似的语法。整个Microsoft Developer Studio在VBScript中被看作是一个具有复杂结构的分层对象。通过访问调用这些对象的属性和方法，可以在最大程度上控制Microsoft Developer Studio的行为。

简单的宏可以通过录制来生成。下面的过程讲述了录制宏的整个过程：

1. 选择Tools菜单中的Macro命令，直至弹出如图1.16所示的对话框。

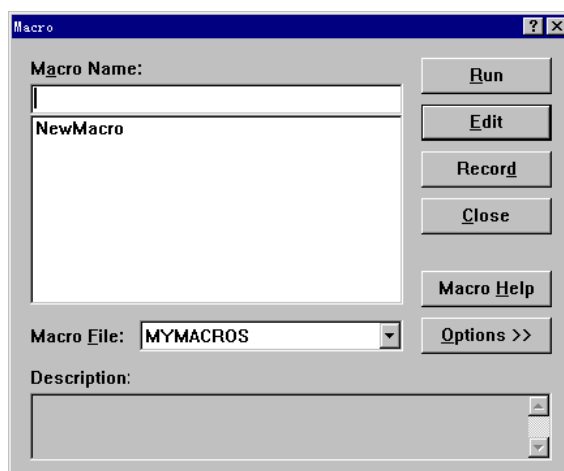


图1.16 准备生成宏

2. 单击Record按钮，然后输入宏的名称。这时，你可以看到多了两



个工具栏按钮：然后，按顺序执行一系列的操作，中间可以单击按钮来暂停录制，以跳过那些不需要录制到宏中的内容。最后，单击停止宏的录制。

下面是一次宏录制所生成的代码，可以根据VBScript语言的规则修改这些代码以适应你的特殊需要，当然，从头手动的键入这些代码也是完全可以的，但这往往要比修改通过宏录制的代码的方法来得慢。

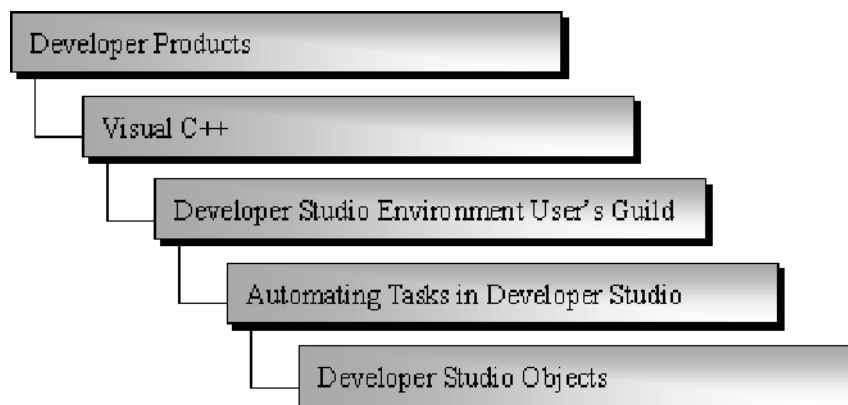
```
Sub NewMacro2()  
  
'DESCRIPTION: A description was not provided.  
  
'Begin Recording  
  
ActiveWindow.Close dsSaveChangesPrompt  
  
Documents.Open "F:\Users\Leib\VcProj\nlitera\nlitera.cpp", "Text"  
  
Documents.Open "F:\Users\Leib\VcProj\nlitera\main.cpp", "Text"  
  
Windows("main.cpp").Active = True  
  
ExecuteCommand "ConfigurationSelectTool"  
  
ExecuteCommand "ConfigurationSelectTool"  
  
ExecuteCommand "Build"  
  
Documents.SaveAll True  
  
'End Recording  
  
End Sub
```

这里我们将不对上述代码的内容进行讲解和说明，因为就VBScript语言而言，其内容就足以写成一本书了，再加上Microsoft Developer Studio本身提供的对象结构及其使用本身就非常的复杂，讨论这些内容已超过了我们将要在本书中讲述的范畴。关于VBScript可以参考Visual C++联机文档中的节点：



而关于Microsoft Developer Studio本身的对象结构则可以参考下面

的联机文档节点：



在Developer Studio Environment User's Guild节点的其它子节点处还有许多与Microsoft Developer Studio相关的参考资料，这里就不再一一尽述，需要时可以从该节点逐级展开至所需内容。

## 第二章 面向对象编程与C++语言

面向对象的编程是当前程序设计中的热门话题，在这方面已有很多的书籍和其它文献可供参考，而且，全面的讲述面向对象的编程技术也不是本书的任务，在这里，本章仅仅介绍使用Visual C++进行面向对象编程中所用到的一些关键概念，这些概念是进一步学习Visual C++所必须的。除了介绍面向对象编程和C++语言的基本概念外，本章还介绍了在使用C++的类时所需注意的一些问题。

另一个需要注意的问题是，尽管本章会介绍面向对象编程和C++语言的基本概念，但是，我们仍然假定阅读本书的读者具有C++语言的基础知识，至少，需要使用过C++中的类和对象，并且对它们有一定的了解。本章也会介绍类和对象的概念以及如何使用它们，但是，假设你在阅读这些内容之前，对它们还是一无所知，那么，我们建议你最好先看一下一些专门介绍面向对象编程和C++语言的入门知识，本书不需要你非常深入的了解这些内容，因为我们假定你懂得C++语言，但不假定你精通C++语言，本章正是写给那些虽然能够使用C++语言中的类和对象进行编程，但是对一些概念的认识还不够清晰的C++初学者。如果你已经对C++语言以及它在Visual C++中的实现非常之熟悉，那么，你也可以暂时跳过本章的内容，继续阅读本书的其它章节。

本章所要讲述的内容包括：

- 面向对象的编程的编程技术
- 对象及其特点
- 使用C++中的类
- 从已有的类中派生新的类
- 多态及其在C++中的实现
- 类模板

### 第一节 面向对象的编程技术

本节阐述面向对象的程序设计方法中的若干重要概念、方法和思想。在刚开始的时候，也许你对这些概念还不大能搞得清楚，但是，随着对C++的使用和了解的深入，当你再回过头来仔细推敲这些概念，就

会发现对它们有了新的认识。

传统的“结构化程序设计”方法是由荷兰学者Dijkstra在70年代提出的，它把面向机器代码的程序抽象为三种基本程序结构：顺序结构、选择结构和重复结构，并提出了一系列的设计原则，如自上而下、逐步求精、模板化编程等。根据这些原则，按照程序所需实现的功能，自上而下层层展开。上层是定义算法的模块，最下层是实现算法的模块。按照这样的规范构成的模块是高度功能性的，有很强的内聚力。但各模块的数据处于实现功能的从属地位，因此，各模块与数据间的相关性就较差，无论把数据分放在各个模块里还是作为全局量放在总控模块中，模块之间都有很大的耦合力。在Windows程序中，多个模块是并发执行的，这样，这种耦合力就会极易导致程序系统出现混乱。

而且，在传统的语言中，程序是由传递参数的程序和函数的集合组成，每个过程处理它的参数，并可能返回某个值，这种程序是以过程为中心的。在传统的面向过程的程序设计中，程序员必须基于过程来组织模块，这必然会导致程序的结构与应用领域中结构差异很大。

现行程序设计方法的另一个大的缺点是围绕着各种软件系统中关键结构的作用域与可见性。许多重要的函数或过程的实现主要地取决于关键的数据结构。如果一个或多个这样的数据结构发生了变化，这种变化将涉及到许多方面，许多函数和过程必须重写。有时几个关键的数据结构发生变化，将导致整个软件系统的结构崩溃。随着软件规模和复杂性的增长，这种缺陷日益明显。当程序达到一定规模后，为了修改一个小的错误，常可能引出多个大的错误，究其原因，问题就出在传统的程序设计方式上。

而当前的软件应用领域已从传统的科学计算和事务处理扩展到了其它的很多方面，如人工智能、计算机辅助设计和辅助制造等，所需处理的数据也已从简单的数字和字符串发展为记录在各种介质上并且有多种格式的多媒体数据，如数字、正文、图形、声音和影像等。数据量和数据类型的空前激增导致了程序的规模和复杂性均接近或达到了用结构化程序设计方法无法管理的程度。

为了最大限度的得用已有的资源和减少程序开发的工作量，需要有一种比传统的过程式结构化程序设计方法抽象能力更强的新方法，面向对象的程序设计方法正是在这种背景下诞生的。

### 2.1.1 面向对象的程序设计

面向对象的程序的最根本的目的就是使程序员更好的理解和管理庞大而复杂的程序，它在结构化程序设计的基础上完成进一步的抽象。这种在设计方法上更高层次的抽象正是为了适应目前软件开发的特点。

最早的“面向对象”设计语言是Smalltalk语言，该语言是1972年美国Xerox公司Palo Alto研究中心为快速处理各种信息而在Alto个人机上研制的软件，1983年正式发行了Smalltalk 2.0版。Smalltalk 80被认为是最纯的面向对象的语言，该语言只有一种数据类型—对象。Smalltalk未能推广，Alto机上的软件也从未出售过，但它所开创的面向对象程序设计方法将结构化程序设计的抽象层次又增高了一层，对程序设计方法产生的影响是巨大的。继Smalltalk之后，出现了许多的面向对象的程序设计语言，如Eiffel、Clos、C++和Objective-C等，其它的非面向对象的程序设计语言，如Pascal、Basic等也引进了新的面向对象的机制，产生了新的Object Pascal、Visual Basic等面向对象的变种。

使用面向对象的程序设计方法绝非是要摒弃现有的结构化程序设计方法，相反，它是在充分吸收结构化程序设计优点的基础上，引进了一些新的、强有力的概念，从而开创了程序设计工作的新天地。面向对象的程序设计方法把可重复使用性视为软件开发的中心问题，通过装配可重用的部件来生产软件，而不是像目前编程所用的那样，通过调用函数库中的函数来实现。这里要注意，我们是基于应用程序这一个层次来阐述这些问题是，事实上，在对象内部的实现上，我们常常使用过程式的结构化程序设计方法，也常常调用C/C++函数库中的很多有用的函数，然而从程序的总体结构上说，它是由一系列对象构成的，对象之间能够以某种方式进行通信和协作，从而实现程序的具体功能。

面向对象的程序设计中最基本的概念是对象，一般意义上的对象指的是一个实体的实例，在这个实体中包括了特定的数据和对这些数据进行操作的函数。对于面向对象的程序设计，一个对象具有状态(state)、行为(behavior)和标识(identity)。对象的状态包括它的属性和这些属性的当前值。对象的行为包括可以进行的操作以及所伴随的状态的变化。对象的标识用来区别于其它的对象。而一个COM对象的行为由它所支持的接口来定义。我们通常不显式的指定COM对象的状态，而被其接口所包含。使用IUnknown::QueryInterface在接口之间进行移动的能力定义了COM对象的标识。

对象的核心概念就是通常所说的“封装性”(encapsulation)、“继承性”(inheritance)和“多态性”(polymorphism)，下面我们分别阐述其具体含义。

### 2.1.2 封装

按照面向对象编程原定义，所谓的封装性是指隐藏类(class)为支持和实施抽象所作的内部工作的过程。类的接口是公有的，它定义了一个类所能完成的功能，而这些接口的实现是私有的或受保护的，它定义了类完成这些功能所作的具体操作。对于使用这些类的编程者来说，只需要知道类所能完成的功能，而不需要知道这些功能具体是如何实现。拿我们所常的手表作为例子，在使用手表时，我们只需要知道手表所能完成的功能和如何使用手表来完成这些功能，这些内容相当于对象的接口。我们不需要知道在手表的内部，这些功能是如何实现的，因此，对于手表来说，无论手表使用的是一般的机械摆，还是石英振荡器，只要它们的使用方法是完全一样的，用户就没有必要知道这个不同。另一个例子的集成电路芯片，我们只需要知道该芯片的每一个引脚的电气参数和功能，而不必知道这些功能在芯片的内部是如何实现的，就可以使用该芯片来组装电路。如果仅就封装性而言，这里的手表和集成电路芯片就相当于面向对象的编程中的对象。下面我们再举一个例子，众所周知，Win32操作平台目前包括Windows 95和Windows NT，而事实上，Windows 95和Windows NT使用了完全不同的内核，对于很多同样的操作，在Windows 95和Windows NT两个操作系统下的实现方式是不同，但两个操作系统都提供了同样的Win32应用程序编程接口(API)，这样，我们所编写的应用程序只需要和Win32应用程序编程接口打交道，而没有必要知道具体的每一个Win32 API函数在操作系统中是如何实现的。换一种说法就是，Windows 95和Windows NT封装了Win32 API函数的具体实现。

一个类在定义数据的同时也定义了对这些数据的操作，这些操作称作方法(method)。按照面向对象的定义，方法就是对对象中的数据的访问。在C++中，对对象中的数据的访问是通过公有成员函数来进行的，这些公有成员函数可以在对象的外部进行调用，它们提供了对象的外部接口。而对于这些接口的内部的实现在对象的外部的不可见的，这些实现包括了类内部所使用的数据结构和支持公有方法的实现的私有成员函数，通常，这些数据成员和成员函数是私有的，它们只能为类中成员函数所访问，而不能从类的外部进行访问。

访问一个方法的过程称为向这个对象发送一个消息(message)，对象的工作是靠消息来激发的，对象之间也是通过消息发生联系的，即请求其它对象做什么或响应其它对象的请求是通过发送或接收消息来实现的。

- 注意：



- 这里的所说消息和在Windows编程中所常说的消息是两个不同术语，尽管在某些方面两者的确有很多相象之处。

封装可避免许多维护性问题。如果一个基本数据类型的结构被修改了，例如一个链表修改成了一个数组，除类中的访问该数据的方法的代码外，软件系统的其余部分是不受影响的，因为基本数据在外部是不可见的，只能通过公有方法的接口与基本数据发生联系，改变一个类的实现，丝毫不影响使用这个类的程序员，从而大大的减少了应用程序出错的可能性。

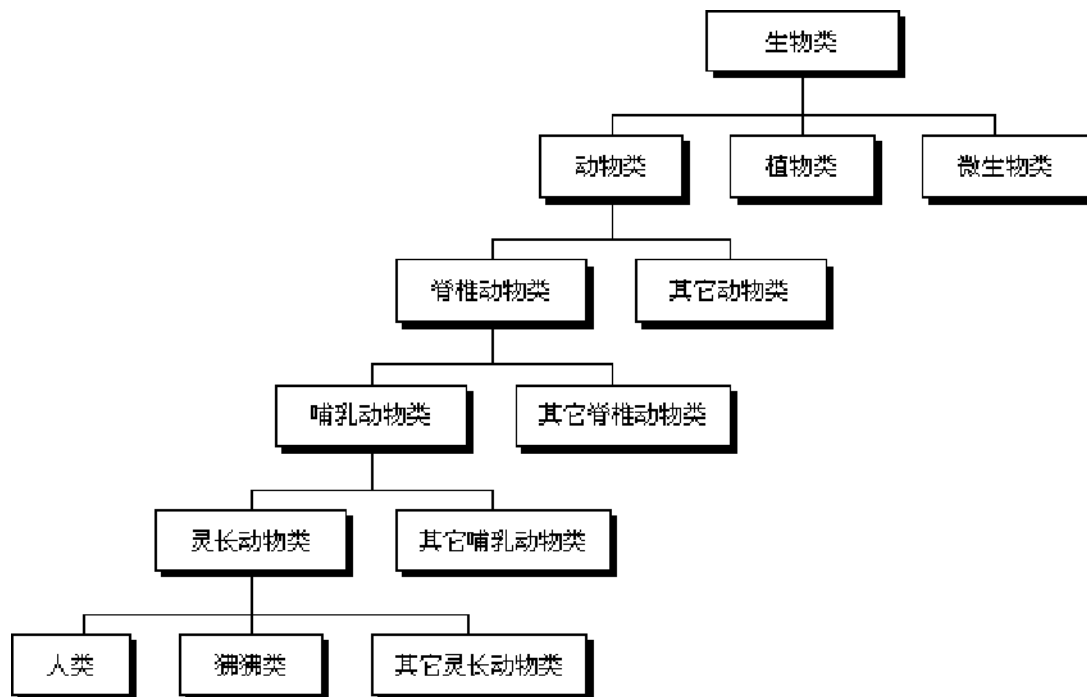


图2.1 自然世界中的继承关系

### 2.1.3 继承

类支持层次机制，因此我们可以借用可重用性部件来很容易的从一个或多个已有类出发，来生产各种更符合我们要求的新类。假设我们从类A出发来派生新的类B，那么我们称类A为类B的基类(base class)，类B为类A的派生类(derived class)，类B继承了类A中的各种行为和状态，并可添加自己的成员变量和成员函数。

我们先来看一个例子，图2.1给出了自然世界中的生物的一种继承层次图，最高层次的生物类代表了层次结构中最一般的概念，较低层次的类表示由上一层的类(即其基类)所派生的特殊的概念。如上面的继承关系，动物类从其基类—生物类中继承生物类的所有属性和行为，并且定义了动物类所特有的属性和行为，类似的，脊椎动物类从动物类那儿继承了所有的属性和行为，并且定义了自身特有的属性和行

为；……，人类从灵长动物类那儿继承了所有的属性和行为，并且定义了人类所特有的属性和行为。

之所以举上面的例子是为了将程序空间和现实生活空间来进行对比，结果说明一点，类的继承使得我们可以以一种自然的方式来模拟生活空间中的对象的层次结构，也就是说，我们可以以一种符合正常思维逻辑的自然的方式来思考和组织应用程序的结构，然后，可以将这个结构几乎不作修改或者只需作少量的修改地用面向对象的编程来表达，从而大大的缩短了软件系统的开发周期。

下面我们举一个现实编程中的例子，考虑MFC (Microsoft Foundation Class Library, Microsoft基础类库)中的CEdit类，它封装了Windows中的编辑框控件，图2.2显示了CEdit类的继承结构。

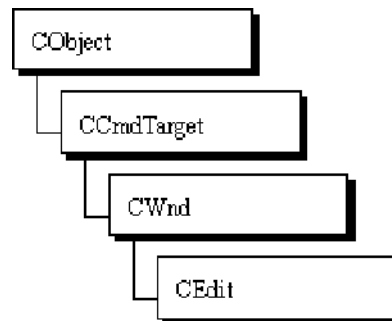


图2.2 类CEdit在MFC中的继承层次

在图2.2中，类CObject是所有的MFC类的根(根是一个术语，它指在继承层次中处于最顶层的类，根是所有继承层次中的类的最终基类)，在类CObject中提供了功能有：串行化支持、运行库信息、对象诊断输出以及与集合类的兼容等。类CCmdTarget从类CObject直接派生，它是Microsoft基础类库的消息映射结构的基类，消息映射将命令和消息传递给所编写的处理成员函数，这里，命令指来自菜单项、命令按钮和加速键的消息。类CWnd提供了MFC中所有窗口类的基本功能性，它封装了Windows中的窗口句柄hWnd。类CEdit从CWnd直接派生，它提供了对Windows编辑控件的特定支持。

我们看到，类CEdit本身仅提供了特定于编辑控件的38个成员函数，但是，你可以通过类CEdit进行调用的成员函数却多达300多个，事实上，这些成员函数中的绝大部分由其基类所提供，其中CWnd就为CEdit提供了多达304个成员函数，由于CEdit类继承了其基类的数据和方法，因此，可以通过CEdit类调用CWnd类中提供的方法来实现对标准Windows窗口的操作。

继承机制所带来的最大优势在于使软件系统非常的易于扩充，程序员

不仅可以直接的使用各种已有的类，还可以从这些类方便的派生出新的类，新的类继承了基类所包括的所有接口和功能，因此只需要定义和实现与基类所提供的功能中不同的那一部分，这大大的降低了软件开发的复杂性和费用，因此面向对象的编程方式非常之适合于进行大型软件系统的开发。

降低软件开发的复杂性的意义不仅在于它可以有效的降低软件开发的费用，而且还使得在软件系统中出错的可能性大为减少。由于类有着清晰的继承层次，因此，我们可以很快的定义出错的代码所处的位置，因此能够很快的修正程序中出现的問題。

继承机制还使得我们可以将与现实生活空间相一致的思维方式应用于程序空间，即我们可以在程序设计时使用直观的思维方式设计程序中所使用的对象的层次结构，然后，直接将此结构映射到面向对象的程序空间，而不需要做任何修改或仅需要作少量修改就可以使用面向对象的程序设计方法来实现该结构。这时，编写程序的过程更类似于“搭积木”，我们可以从很多途径来获得程序所需使用的各种对象，然后，将这些对象以一定的层次结构组合起来，从而实现程序的逻辑结构。

#### 2.1.4 多态和虚函数

在讲述多态之前我们先来看一个问题。仍以前面的图2.2为例，假定我们已经定义了一个指向哺乳动物类的实例对象的指针，如下所示：

```
CMammal *pMammal
```

然后我们定义了一个人类的实例对象和一个狒狒类的实例对象，如下所示：

```
CHuman Human;
```

```
CBaboon Baboon;
```

然后，我们可以将指针pMammal指向Human对象，在C++语言中是可以这样做的：

```
pMammal=&Human;
```

也可以将指针指向Human对象：

```
pMammal=&Baboon;
```

考查上面的两种情况，我们假定在哺乳动物类、人类和狒狒类中都定

义了一个Eat（吃）方法，很显然，当我们使用下面的代码来调用人类对象和狒狒类对象的Eat方法时不会遇到什么问题：

```
Human.Eat();
```

```
Baboon.Eat();
```

但是，现在来考虑下面的代码：

```
pMammal->Eat();
```

当pMammal指向不同的对象时，上面的代码将发生什么样的结果。很显然，当pMammal指向一个哺乳动物类的实例对象时，上面的代码将调用哺乳动物类的Eat方法。但是，当pMammal指向一个人类的实例对象时，上面的代码是调用人类的Eat方法呢，还是仍然调用哺乳动物类的Eat方法？我们期望的是前面一种情况，这就是类和对象的多态性。我们期望，当pMammal指向不同的实例对象时，编译器将根据实例对象的类型调用正确的Eat方法。在C++中，类的多态性是通过虚函数来实现的。就上面的例子来说，我们将Eat方法定义为一个公有的虚函数，这样，当pMammal指针指向一个人类的实例对象时，编译器调用的就是人类的Eat方法，当pMammal指针指向一个狒狒类的实例对象时，编译器调用的就是狒狒类的实例对象，从而实现了运行时的多态。

在C++中，多态定义为不同函数的同一接口。从这个定义出发，函数和操作符的重载也属于多态。

考虑下面定义的两个函数：

```
int print(char*);
```

```
int print(int);
```

上面的两个不同函数使用同样的函数名，在C++中，这称为函数的重载。这时，若将一个字符指针传递给print函数，如下所示：

```
char *sz="Hello World!";
```

```
print(sz);
```

这时，编译器调用的是int print(char\*)，如果将一个整型变量传递给print函数，如下所示：

```
int i=0;
```

```
print(i);
```

则编译器调用的是`int print(int)`。这种根据所传递的参数的不同而调用不同函数的情形也称作多态。

下面我们来看运算符重载的例子，在下面的过程中，我们为矩阵类重载了运算符“+”：

```
CMatrix operator +(CMatrix, CMatrix);
```

然后使用下面的代码：

```
int a=1, b=1, c;
```

```
CMatrix A(3, 3, 0), B(3, 3, 1), C(3, 3);
```

```
c=a+b;
```

```
C=A+B;
```

在上面的例子中，我们定义了三个整型变量和三个类`CMatrix`的实例变量，`A`、`B`和`C`均为 $3 \times 3$ 的矩阵，其中`A`中全部元素均置为0，`B`的全部元素均置为1。然后将“+”运算符作用来整型变量和类`CMatrix`的实例变量。这时，编译器将表达式`c=a+b`翻译为

```
c=operator +(a, b);
```

而将表达式`C=A+B`翻译为

```
C=operator +(A, B)
```

然后，根据传递参数的不同，调用不同的`operator +`函数。

与类和对象的多态不同，对于函数和运算符的多态，是在编译过程中完成的，因此我们将它们称为编译时多态，与此相对，将类和对象的多态称为运行时多态。

封装性、继承性和多态性是面向对象编程的三大特征，则开始的时候，你也许对它们还没有非常清晰的概念，但这没有什么关系，当你使用了一段时间的C++语言，然后再回过头来看这些概念时，你就会发现对它们有了更深入的认识和了解。事实上，许多的C++程序正是在使用C++语言进行编程已有相当一段时间时才对这三个概念有了正确而清晰的认识的。

- 注意：

- 在前面的过程中，我们多次混用了对象和类这两个术语，事实上，它们之间是有差别的。在C++中，类是一种数据类型，它是一组相同类型的对象的抽象。在类中定义了这一类对象的统一的接口。而对象在C++中是指某一数据类型的一个实例，如下面的代码所示：

- `Class CBalloon;`
- `CBalloon Balloon1;`
- `CBalloon Balloon2;`
- `CBalloon Balloon3;`

在上面的代码中，我们声明了一个类CBalloon，这是所有狒狒对象的抽象，然后，我们使用类来创建了三个不同的实例对象Balloon1、Balloon2和Balloon3。

然而，尽管类和对象是两个意义不同的概念，但是，它们在C++语言中是紧密相关的，这使得很多的文档(包括Visual C++的联机文档)在有些时候也不加区分的混用这两个术语。读者根据相应的上下文，应该能够很方便的辨明它们具体所指的内容究竟是类还是类的实例对象。

在随后的小节中，我们将讲述如何在C++中实现上面所说的面向对象的编程概念。

## 第二节 类的声明和定义

本节讲述在声明和定义类及类中的成员变量和成员函数所需注意的问题。在本章中，我们使用了传统的C/C++编程风格，即每一个程序都以一个名为main的主函数为入口，过去常见的MS-DOS应用程序即是这样的。众所周知，Visual C++ 5.0不支持16位MS-DOS编程，但这并不意味着我们必须放弃这种编程模式。事实上，我们仍然可以在Visual C++ 5.0中编译和调试在本章中出现的所有程序，方法是使用Win32 Console Application (Win32控制台应用程序)来编译它们，Win32控制台API提供了字符方式的应用程序的编程接口，并且，我们仍可以使用过去所熟知的以main函数为入口的编程方式。除此之外，Win32平台为控制台应用程序提供了虚拟的一个标准输入和输出设备(由于Windows程序共享屏幕和其它所有设备，因此不存在在DOS中的那种标准输入和输出设备)，这样，我们可以使用标准的C语言输入输出函数scanf和printf，还可以使用标准的C++语言输入和输出流cin和

cout。关于控制台应用程序的内容，可以参考本书的相关章节。

### 2.2.1 类及其成员变量和成员函数的声明和定义

在C++中，存在三种类类型：类、结构和联合，它们分别使用三个关键字来声明和定义类：class、struct和union。表2.1给出了不同类类型之间的差别。

表2.1 三种不同类型的类：类、结构和联合

	类	结构	联合
声明和定义时使用的关键字	class	struct	union
默认的成员访问权限	私有	公有	公有
使用限制	无	无	同时只能使用一个成员

由于联合的特殊性，我们将在后面的内容中讲述它。下面给出一个使用class来定义类的示例：

```
class CCircle
{
public:
    unsigned Radius;
    CPoint Center;
};
```

下面我们使用关键字struct来定义类CPoint：

```
struct CPoint
{
    unsigned x;
    unsigned y;
};
```

- 注意：
- 请不要忘了在类定义的第二个大括号“}”之后加上一个分号，否则编译时会出现一堆莫名其妙的错，并且没有一个错误会告诉你是定义类的时候少加了一个分号(也许它会告诉你别的什么地方少了一个分号，但事实上那里什么错误也没有)。也许有人认为这并非是一个值得强调的问题，然而事实上绝大多数初学者(甚至包括一些有经验的程序员)都会由于疏忽大意而犯了这样的错误，并且在修正它的时候花费了大量的时间却依然没有发现所有的问题实际上只是由一个小分号造成的。

细心的读者会注意到，在类CCircle的定义中我们使用了public关键字，而在类CPoint的定义中却没有使用。这是因为在使用struct定义的类中，成员的默认访问权限为公有，因此不需要使用public关键字，而对于使用class关键字来定义的类，由于其成员的默认访问权限为私有，因此必须使用关键字public来将成员Radius和Center的访问权限设置为公有。一般来说，一个类中总是包括了一定数量的公有成员，没有公有成员类由于没有提供任何接口，事实上没有什么用。

对于同一个类，可以使用关键字class来定义，也可以使用关键字struct来定义。对于上面的例子，我们将类CCircle改用关键字struct来定义，而将类CPoint改用关键字class来定义如下：

```
struct CCircle
{
    unsigned Radius;
    CPoint Center;
};

class CPoint
{
public:
    unsigned x,y;
};
```

现在我们来考虑一下，如果在同一个文件中定义类CCircle和类



CPoint，并采用如上面的代码，会遇到什么问题。我们看到，在类CCircle中定义了类CPoint的实例对象Center，然而，在定义实例对象Center的时候，我们还没有给出类CPoint的定义，这时，在编译时就会出现错误。因此，我们应该采用下面的顺序：

```
struct CPoint
{
    unsigned x;
    unsigned y;
};

class CCircle
{
public:
    unsigned Radius;
    CPoint Center;
};
```

以确保在使用类CPoint这前已经对它进行了定义。如果我们在类CPoint中添加一个成员函数IsInCircle来判断一个点是否在圆内，如下面的代码所示：

```
struct CPoint
{
    unsigned x;
    unsigned y;
    bool IsInCircle(CCircle *Circle)
    {
        return ( ((x-Circle->Center.x)*(x-Circle->Center.x)
        +(y-Circle->Center.y)*(y-Circle->Center.y))
        <=Circle->Radius*Circle->Radius );
    }
};
```

```
};
```

这真是一件很不幸的事情。为什么这样说呢？请看上面的代码，我们在IsInCircle成员函数中又使用到了类CCircle，而按照我们在前面所给出的代码，类CCircle是在类CPoint之后进行定义的，那么，将类CPoint放到类CCircle之后定义行不行呢？答案是不行的，不要忘了我们还在类CCircle中定义了一个类型为CPoint的数据成员Center。

使用类的声明可以解决上面所遇到的两难问题。为了在定义类CCircle之前先引用类CCircle的类名，我们可以在类CPoint之前添加如下的代码：

```
Class CCircle;
```

- 注意：

- 不能在末尾加上一对大括号，因为这样的话，上面的代码就会被当作一个类的定义，这时，在同一文件作用域中就会出现了一类的两个定义，而这在C++中是不允许的，一个类可以有多个声明，但是它只能有一个定义，对于函数也是这样。

上面的代码和类的定义的差别在于它没有一对大括号“{}”，这样，我们就可以在定义类CCircle先定义指向一个类CCircle的对象的指针和引用。但是，这时CPoint的代码不能像上面那样写，这是因为在成员函数IsInCircle中我们使用了类CCircle中定义的数据成员，而在此之前编译器只知道用以使用一个名为CCircle的类，但是不知道这个类中包括了什么样的成员，因此上面的代码在编译时通不过。

如果解决所遇到的新问题呢？回过头来看类CPoint的定义，我们在定义类的同时定义了成员函数IsInCircle，联想到C语言中的函数，我们可以很自然的想到，可不可以先声明成员函数IsInCircle，然后再定义它呢？如果可以这样的话，我们就可以在类CPoint的定义中先声明成员函数IsInCircle，然后在定义了类CCircle之后再添加成员IsInCircle的定义。

事实上我们正是基于这个想法来解决问题的。我们的确可以先在类CPoint的定义中声明成员函数IsInCircle，然后在后面添加它的定义，完整的代码如下：

```
#include <iostream.h>
```

```
class CCircle;
```

```

struct CPoint
{
    unsigned x;
    unsigned y;
    bool IsInCircle(CCircle *Circle);
};

class CCircle
{
public:
    unsigned Radius;
    CPoint Center;
};

bool CPoint::IsInCircle(CCircle *Circle)
{
    return ( ((x-Circle->Center.x)*(x-Circle->Center.x)
+(y-Circle->Center.y)*(y-Circle->Center.y))
<=Circle->Radius*Circle->Radius );
}

void main()
{
    CPoint Center;
    CPoint Point;
    CCircle Circle;
    Center.x=0;
    Center.y=0;
    Point.x=2;

```

```

Point.y=3;

Circle.Radius=3;

Circle.Center=Center;

if (Point.IsInCircle(&Circle))

cout<<"The point is in the circle."<<endl;

else

cout<<"The Point is not in the circle."<<endl;

}

```

我们先看一下函数IsInCircle的定义和一般的函数的定义有什么不同。我们看到，在函数名IsInCircle前使用了作用域限定符“::”，即出现在函数定义处的完整的函数名为CCircle::IsInCircle，这说明了函数IsInCircle是类CCircle的一个成员。如果仅使用函数名IsInCircle，则编译器不会把它当作类CCircle的成员，而是将它当作一个具有文件作用域的全局函数。

同在类定义内部定义的成员函数一样，在函数外定义的成员函数一样可以直接的访问类中的数据成员和调用其它的成员函数，而无论它们本身是私有的还是公有的。但是，在类定义内部定义的成员函数和在类定义外部定义的成员函数还是有差别的。对于在类定义内部定义的成员函数，编译器总是将它作为一个内联函数来进行编译，无论你是否使用了inline关键字。而对于在类定义外部定义的成员函数，除非你显式使用了inline关键字，编译不会将它作为一个内联函数来编译。但如果加上了inline关键字，则编译器以同样的方式对待在类定义外部定义的成员函数。一般来说，对于比较短少的成员函数，在可能情况下，我们大多在类定义内部定义它们，而对于代码量比较大的成员函数，则几乎都是在类定义的外部来定义它们，否则，代码的可读性会变得很糟糕。

- 注意：
- 请仔细分析下面的代码：

get i成员函数更好：

```

○ int geti()

○ {

```

```
○ return i;  
  
○ }
```

但是，对于一个结构庞大的对象，基于运行效率的考虑，我们可以不得不返回指向成员的指针，因为当C++为了返回对象时对这么一个庞大的对象的成员进行逐一拷贝会需要大量的时间。当然，我们可以这样修改get i函数：

```
○ const int* get i()  
  
○ {  
  
○ return &i;  
  
○ }
```

但是用户使用下面的代码却是合法的：

```
○ int *i=(int*)myCls.get i();
```

我们没有办法强制使用类MyClass的编程者不这样做，尽管我们建设他们不要这样做，但是，编程者应该知道由此可能导致的后果。而我们也没有更好的办法来避免出现这样的情况。即使我们愿意牺牲拷贝对象所耗费的时间，在一些情况下，编程者仍然可以将函数的返回值强制为一个引用，这时，由于C++编译所进行的优化，我们仍然可能通过该引用来修改类中的私有成员的值。

## 2.2.2 成员函数和this指针

类的成员函数可以是静态的，也可以是非静态的。静态的成员函数和静态的成员变量以及它们之间的关系我们将在下一小节中讲述，对于非静态成员函数，由于它可以使用对象中的非静态数据，C++为类的每一个实例对象维护了不同的非静态数据成员，这样，我们很自然的想到一个问题，类的成员函数是如何区分不同的实例对象的数据成员的呢？事实上，对于每一个类的非静态成员函数，都有一个隐含的this指针，该指针指向调用该成员函数的实例对象。

当成员函数使用对象中的非静态成员(无论是成员变量还是成员函数)，它事实上是通过this指针来调用该成员的。请看下面的代码：

```
class MyClass
{
public:
    void seti(int newi)
    {
        i=newi;
    }
private:
    int i;
};
```

对编译器而言，成员函数seti的定义事实上如下面的代码所示：

```
void seti(int newi)
{
    this->i=newi;
}
```

对于以不同方式定义的成员函数，this指针具有不同的类型，在上面的seti函数中，this指针的类型为MyClass\* const，这表明，this指针是一个常指针，程序中可能通过this指针来修改类中的成员的值，但不可以修改this指针本身的值，也就是说，不可以对this指针重新赋值，以使它指向另一个对象。

假设我们在类MyClass定义了另一个成员函数geti，如下面的代码所示：

```
int geti() const
{
    return i;
```

```
}
```

在上面的代码中，const关键字表明函数get i不会修改调用该成员函数的实例对象中的成员的值。对于编译器而言，上面的成员函数get i是这样的：

```
int get i() const  
  
{  
  
return this->i;  
  
}
```

由于使用了const关键字，则this指针的类型为const MyClass\* const，这表明不但不能修改this指针本身，也不能通过this指针修改对象中的成员。举一个例子，如果我们按如下方式定义set i函数：

```
void set i(int new i) const  
  
{  
  
i=new i;  
  
}
```

上面的代码将会导致编译错误“l-value specifies const object”。

一般情况下我们不需要显式地使用this指针。下面的代码显示了this指针的一个典型应用：

```
#include <windows.h>  
  
#include <stdio.h>  
  
#include <iostream.h>  
  
class MyPosition;  
  
class MyScreen  
  
{  
  
public:  
  
void PrintMyPosition(MyPosition* pMyPosition);
```

```

};

class MyPosition
{
public:
    int x,y;

    void DisplayMyPosition(MyScreen* pMyScreen);
};

void MyScreen::PrintMyPosition(MyPosition* pMyPosition)
{
    HANDLE hConsoleOutput=GetStdHandle(STD_OUTPUT_HANDLE);

    COORD dwCursorPosition={0, 24};

    char sz[80];

    sprintf(sz, "My position is (%d,%d).", pMyPosition->x, pMyPosition->y);

    DWORD cWritten;

    WriteConsoleOutputCharacter(hConsoleOutput, sz, lstrlen(sz),
    dwCursorPosition, &cWritten);

    dwCursorPosition.X=pMyPosition->x;
    dwCursorPosition.Y=pMyPosition->y;

    SetConsoleCursorPosition(hConsoleOutput,dwCursorPosition);
}

void MyPosition::DisplayMyPosition(MyScreen* pMyScreen)
{
    pMyScreen->PrintMyPosition(this);
}

void main()
{

```



```
MyPosition mypos;

MyScreen mysrn;

cout<<"Enter a position (e.g. 4 10): ";

int x,y;

cin>>x>>y;

mypos.x=x;

mypos.y=y;

mypos.DisplayMyPosition(&mysrn);

}
```

请注意类MyPosition的DisplayMyPosition成员函数，它使用一个指向MyScreen对象的指针作为参数。在DisplayMyPosition成员函数的实现中，调用了MyScreen对象的PrintMyPosition成员函数，PrintMyPosition成员函数需要一个指向MyPosition对象的指针作为其参数，这里，我们希望将指向调用DisplayMyPosition成员函数的对象本身的指针作为参数进行传递，这时就必须使用this指针。将指向对象自身的指针作为参数传递给其它函数，在实际编程中是一个很常用的技巧，在这些场合，this指针得到了广泛的运用，在以后编程的过程中，我们会经常看到这样的用法。

除了示范this指针的用法以外，上面的代码还示范了如何在控制台窗口中定位输出字符串的位置。运行上面的程序，它首先要求用户输入一个坐标值，然后，在屏幕的最底行(这里假定用户使用80×25的控制台窗口大小)，最后，将下一次显示输出的位置定位于用户所输入的坐标值所指定的点。整个程序运行的结果如图所示。

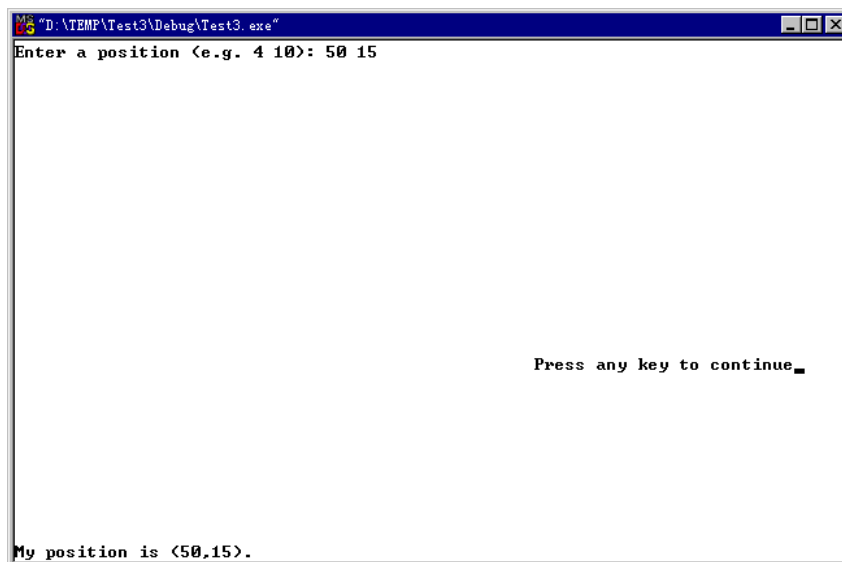


图2.3 在控制台窗口中进行定位

- 注意：
- 上面的程序代码不能在MS-DOS环境下进行编译和链接，因为我们在程序中显式的调用了Win32控制台API函数(如 `SetConsoleCursorPosition` 和 `WriteConsoleOutputCharacter` 等)，它们属于Win32应用程序接口的一部分。为了使用这些函数，我们在程序的最开始包含了头文件 `windows.h`。因此，上面的程序尽管具有和普通的MS-DOS环境下的C/C++程序代码相一致的结构，但它是一个Windows应用程序，只能运行于32位Windows环境。我们可以从上面的函数中看出Win32控制台应用程序和传统的MS-DOS字符模式应用程序之间的本质区别。

### 2.2.3 静态成员

前面说过，无论是类中的成员变量还是成员函数，都可以声明为静态的。类的静态成员在工作起来和非静态成员有很大的差别，因此我们在这里用单独的一个小节来讲述它们。

先看一个静态数据成员的例子。

```
#include <iostream.h>
```

```
class MyClass
```

```
{
```

```

public:

static int i;

};

int MyClass::i;

void main()

{

MyClass cls1;

MyClass cls2;

cls1.i=1;

cout<<"cls1.i="<<cls1.i<<" cls2.i="<<cls2.i<<endl;

cls2.i=2;

cout<<"cls1.i="<<cls1.i<<" cls2.i="<<cls2.i<<endl;

}

```

上面的示例程序的运行结果如下：

```
cls1.i=1 cls2.i=1
```

```
cls1.i=2 cls2.i=2
```

为什么会出现这样的结果呢？这就是静态数据成员和非静态数据成员的一个极大的不同之处。对于同一类的所有实例对象，C++只为它们维护了一份静态变量的拷贝，通过每一个对象对这些静态变量进行操作，事实上都是对同一个数据区域进行操作。这就可以解释上面的运行结果了，无论我们使用表达式cls1.i还是表达式cls2.i，它们都是对同一个i进行操作，很自然，使用表达式cls1.i修改了静态成员i的值，表达式cls2.i的值会立即反映所作的修改。

为了更清晰的说明问题，使用下面的代码替换前面的main主函数：

```

void main()

{

MyClass cls1;

```

```
MyClass cls2;

cout<<"Address of cls1.i is "<<&cls1.i<<endl;

cout<<"Address of cls2.i is "<<&cls2.i<<endl;

}
```

运行修改后的程序，得到如下的结果：

```
Address of cls1.i is 0x0041A0B0
```

```
Address of cls2.i is 0x0041A0B0
```

这下子就真象大白了。无论是cls1.i，还是cls2.i，它们都对应于同一块内存区域。那么，使用cls1.i还是cls2.i还有什么意义呢？的确是没有什么意思。更能够说明问题的使用方法是使用MyClass::i来代替cls1.i和cls2.i这种容易令人混淆的表达式。事实上，即使你使用了如下格式的表达式

```
objectname.staticmember;
```

位于成员选择运算符“.”前面的部分也永远不会被求值。考虑如下的代码：

```
GetMyClass().i=1;
```

这里我们假设函数GetMyClass的返回值为一个类型为MyClass的实例对象，由于编译器不对GetMyClass()进行求值，事实上函数GetMyClass永远也不会被调用，位于函数中的任何代码都不会被执行。

现在回过去看前面的例子，我们注意到，在编译该示例程序的时候，出现了两条警告信息：

```
'cls1' : unreferenced local variable
```

```
'cls2' : unreferenced local variable
```

由于编译器不对表达式cls1.i和cls2.i中运算符“.”前面的部分进行求值，而是把它们看作MyClass::i，这样，对象cls1和cls2实际上没有使用，因此编译器给出上面的两条错误信息。

这里我们还可能得出一条结论，即是说，即使没有定义类的任何实例对象，也可以使用类中声明的静态数据成员。这个问题可以这样解释，类中的静态成员事实上不是类的实例对象的一部分，

它们是单独的对象，尽管我们在类中声明了类的静态成员，但它们事实上具有文件作用域，因此我们必须在类定义的外部定义该成员变量。这就是为什么要在上面的程序代码文件中添加语句

```
int MyClass::i;
```

的原因。

不同于非静态数据成员，可以在定义类的静态成员的时候同时进行初始化，对于上面的静态成员*i*，使用下面的定义语句是合法的：

```
int MyClass::i=0;
```

下面我们来讲述类中的静态成员函数。前面已经讲过，不像非静态成员函数那样，静态成员函数没有隐含的*this*指针，因此，它们只能访问类中的静态数据成员、枚举和嵌套类型。而且，同静态数据成员一样，调用静态成员函数不需要有类的实例对象。看下面的例子：

```
#include <iostream.h>

class MyClass
{
public:
    static int Count()
    {
        return nCount;
    }

    static void Add(int NewElement)
    {
        nCount++;
    }

private:
    static int nCount;
```

```

};

int MyClass::nCount=0;

void main()

{

cout<<"Count="<<MyClass::Count()<<endl;

for (int i=1; i<=5; i++)

{

MyClass::Add(i);

}

cout<<"Count="<<MyClass::Count()<<endl;

}

```

我们没有定义一个类MyClass的实例对象，但仍可以使用其中的所有公有的静态数据成员和静态成员函数。上面的示例程序的运行结果如下：

Count=0

Count=5

同非静态成员函数相比，静态成员函数的使用受到如下的限制：

1. 不能在静态成员函数中直接访问类中的非静态成员；
2. 不能将静态成员函数定义为虚函数(关于虚函数将在后面的内容中讲到)。

○ 注意：

- 可以仅在类中声明其静态成员函数，然后在类的外部添加其定义，但这并不是必须的。可以在类中声明静态成员函数的同时定义它们。
- 同静态数据成员一样，如果使用成员选择运算符“.”和“->”来调用类的静态成员函数，表达式中运算符“.”和“->”前面的表达式不会被求值，如果在这一部分进行了函数调用，则这些函数永远也不会真正的被调用。

## 2.2.4 联合

前面说到，联合实际上也是一种类型的类。但与使用class关键字和struct关键字定义的类不同，联合中的所有数据成员共享同一块内存区域。这就是说，当你修改了联合中的一个数据成员的值，那么，其中所有数据成员的值都会因此而受到影响，请看下面的例子：

```
#include <iostream.h>

#include <iomanip.h>

union Numeric

{

short iValue;

long lValue;

};

void main()

{

Numeric num;

num.lValue=1000000000;

cout<<"num.lValue=0x"<<setiosflags(ios::uppercase)<<hex<<num.lValue<<endl;

cout<<"num.iValue=0x"<<setiosflags(ios::uppercase)<<hex<<num.iValue<<endl;

num.iValue=10000;

cout<<"num.lValue=0x"<<setiosflags(ios::uppercase)<<hex<<num.lValue<<endl;

cout<<"num.iValue=0x"<<setiosflags(ios::uppercase)<<hex<<num.iValue<<endl;

}
```

上面的示例程序的运行结果如下：

```
num.lValue=0x3B9ACA00

num.iValue=0xCA00
```

```
num.lValue=0x3B9A2710
```

```
num.iValue=0x2710
```

为了便于说明问题，我们使用十六进制格式来表示iValue和lValue成员的值，我们看到，如果在程序中修改了lValue的值，那么相应的iValue的值也会发生改变，反之亦然。

使用下面的代码替换上面的main主函数，我们不难得出答案：

```
void main()
{
    Numeric num;

    cout<<"Address of iValue: "<<&num.iValue<<endl;
    cout<<"Address of lValue: "<<&num.lValue<<endl;
}
```

运行得如下的结果：

```
Address of iValue: 0x0012FF7C
```

```
Address of lValue: 0x0012FF7C
```

该结果证实了我们在前面所说的内容：联合中的所有成员共享同一块内存区域。

同使用关键字class和struct定义的类一样，在联合中也可以定义成员函数，这里限于篇幅，就不再举例说明了。

但要注意，使用联合存在如下的限制：

1. 联合中不可以包括虚函数，因为联合不可以使用其它的类、结构和联合作为其基类，也不可以作为其它的类、结构和联合的基类。
2. 联合中不可以包括以下数据成员：
  - (1) 具有构造函数和析构函数的类；
  - (2) 具有自定义赋值运算符的类；
  - (3) 静态数据成员。



## 2.2.5 构造函数和析构函数

前面我们不只一次提到构造函数和析构函数，本节中对这两个概念进行进一步的阐述。

构造函数在创建类的实例对象时被调用，以对类中的数据进行初始化，在2.2.1节最后的示例代码中我们给出了一个使用构造函数来初始化类中的数据成员的例子。

在对象创建的时候，构造函数依次完成以下操作：

1. 如果类是从一个虚基类派生，则构造函数初始化对象的虚拟基指针；
2. 按声明的顺序调用基类及成员的构造函数；
3. 如果在类中声明了虚函数，或者类从基类中继承了虚函数，则初始化对象的虚函数指针，虚函数指针指向类的虚函数表，从而保证在虚函数调用时能够进行正确的绑定；
4. 运行包括在构造函数中的代码。

构造函数名和类名相同，并且不具有任何返回类型，即使是void也不可以。可以在构造函数中使用return语句，但不可以通过构造函数返回任何类型的值。C++允许定义多个使用不同参数列表的构造函数。这里有两种特殊的构造函数，缺省构造函数和拷贝构造函数。缺省构造函数不带任何类型的参数，而拷贝构造函数带有一个单个参数，该参数的类型为一个对相同类的实例对象的引用。

如果编程者在类中没有显式的定义一个缺省构造函数，则编译器会为类生成一个缺省构造函数，如果编程者在类中没有显式的定义一个拷贝构造函数，则编译器会为类生成一个拷贝构造函数。由编译器生成的构造函数者被当作是类的公有成员函数。

编译器提供的缺省构造函数创建对象，并初始化虚拟基表和虚拟函数表，如果基类和成员提供了可用的构造函数，缺省构造函数还将调用基类和成员的构造函数，其顺序如前所述。除此之外，由编译器生成的缺省构造函数不进行其它的操作。

编译器提供的拷贝构造函数创建新的对象，并将当作参数传递给拷贝构造函数的对象中的成员逐个的拷贝到新的对象中。如果基

类或成员提供了相应的构造函数，则调用这些构造函数，否则执行按位拷贝。

作为一个实用的例子，下面的示例代码给出了一个简单的矩阵类，并为矩阵类定义了最基本的运算，包括矩阵加法、减法、乘法以及矩阵的转置运算，除此之外，该类还提供了另一些有用的功能，如矩阵的流式输出等。在示例代码中使用了友元函数和操作符重载的概念，如果对这些概念不够熟悉，可以先略过它们，在本章的后面内容中将有关于操作符重载和友元函数的相关内容。

```
class matrix
{
public:
    matrix(int rsize,int csize,float init=0);
    matrix(matrix& A);
    ~matrix();

    friend matrix operator+(matrix& A,matrix& B);
    friend matrix operator-(matrix& A,matrix& B);
    friend matrix operator*(matrix& A,matrix& B);
    friend ostream& operator<<(ostream& out,matrix& A);
    matrix& operator=(matrix& A);
    float& operator()(int midx,int nidx);
    matrix transpose();

protected:
    float *elem;

    int m;
    int n;
};

ostream& operator<<(ostream& out,matrix& A)
```

```

{
    int width=out.width();
    for(int i=1;i<=A.m;i++)
    {
        for(int j=1;j<=A.n;j++)
            out<<setw(width)<<A(i,j);
        out<<endl;
    }
    return out;
}

matrix::matrix(int row,int col,float init)
{
    m=row;
    n=col;
    elem=new float[row*col];
    for(int i=0;i<row*col;i++)
        *(elem+i)=init;
}

matrix::~~matrix()
{
    delete []elem;
}

matrix operator+(matrix& A,matrix& B)
{
    matrix result(A.m,A.n);
    for(int i=1;i<=result.m;i++)

```

```

    for(int j=1;j<=result.n;j++)
    result(i,j)=A(i,j)+B(i,j);
    return result;
}

matrix operator-(matrix& A,matrix& B)
{
    matrix result(A.m,A.n);
    for(int i=1;i<=result.m;i++)
    for(int j=1;j<=result.n;j++)
    result(i,j)=A(i,j)-B(i,j);
    return result;
}

matrix operator*(matrix& A,matrix& B)
{
    matrix result(A.m,B.n,0);
    for(int i=1;i<=result.m;i++)
    for(int j=1;j<=result.n;j++)
    for(int k=1;k<=A.n;k++)
    result(i,j)+=A(i,k)*B(k,j);
    return result;
}

matrix& matrix::operator=(matrix& A)
{
    m=A.m;
    n=A.n;
    elem=new float[m*n];

```

```

for(int i=1;i<=m;i++)
for(int j=1;j<=n;j++)
operator()(i,j)=A(i,j);
return *this;
}

matrix::matrix(matrix& A)
{
m=A.m;
n=A.n;
elem=new float[m*n];
for(int i=1;i<=m;i++)
for(int j=1;j<=n;j++)
operator()(i,j)=A(i,j);
}

matrix matrix::transpose()
{
matrix result(n,m);
for(int i=1;i<=n;i++)
for(int j=1;j<=m;j++)
result(i,j)=operator()(j,i);
return result;
}

float& matrix::operator()(int row,int col)
{
static float err=.0;
if (row>0 && col>0 && row<=m && col<=n)

```

```

return *(elem+(row-1)*n+col-1);

else

return err;

}

```

一个构造函数可以带有任意类型的参数表，对于可以不使用任何参数的构造函数，编译器将它当作一个缺省构造函数，对于可以使用一个对同类型的实例对象的引用作为参数的构造函数，编译器将它当作一个拷贝构造函数。在上面的例子中，由于函数

```
matrix(int rsize=0,int csize=0,float init=0);
```

可以不使用任何参数，因此可以将它作为类的缺省构造函数，由于函数

```
matrix(matrix& A);
```

使用了一个对matrix实例对象的引用作为参数，因此它被当作一个拷贝构造函数。

我们提供自定义的缺省构造函数和拷贝构造函数的唯一目的是为了进行特殊的初始化操作。考虑类matrix，其缺省FONT SIZE=3>Win32 Release版本，这样会产生很多的假象，从而掩盖了程序中确实存在的问题。

解决这个问题的办法是提供自定义的拷贝构造函数。试将下面的代码段放到类MyClass的定义中：

```

MyClass(MyClass& cls)

{

p=new float;

*p=*(cls.p);

}

```

这时再以Win32 Debug编译和运行程序，就不会出现如图2.4所示的错误了，而且，由下面的运行结果可知，两个实例对象cls1和cls2的成员指针p也指向了不同的内存地址。

Address of cls1.p: 0x00430D20

Address of cls2.p: 0x00430D50

现在回到类`matrix`，请观察它的拷贝构造函数`matrix::matrix(matrix& A)`，在其中使用了与上面的类`MyClass`的拷贝构造函数`MyClass::MyClass(MyClass& cls)`相类似的技巧。

下面我们来考虑一样在那些情况下需要调用拷贝构造函数。

第一种情况如前面的`MyClass`的示例代码，我们以主函数中使用了下面的代码：

```
MyClass ch2=ch1;
```

上面的代码导致编译器调用类`MyClass`的拷贝构造函数，如果用户没有在类`MyClass`中显式定义拷贝构造函数`MyClass::MyClass(MyClass& cls)`，则使用编译器缺省生成的拷贝构造函数。

第二种情况是使用下面的代码：

```
MyClass ch2(ch1);
```

这种情况和第一种情况几乎一样。

第三种情况最为复杂。在某些情况下，编译器需要创建一些临时对象，考虑前面定义的类`matrix`，由于我们在类中定义了运算符“`*`”，因此，在程序中可以出现这样的表达式：

`A*B*C`

这里我们假定`A`、`B`和`C`是类`matrix`的三个实例对象，且它们满足矩阵乘法运算的要求，即`A.n=B.m`，`B.n=C.m`。这时，编译器先调用函数`operator*`计算`(A*B)`，然后，由于表达式中没有指定一个对象用来保存`A*B`的计算结果，因此，编译器创建一个临时对象，我们这时使用`temporary`来指代这个临时对象，然后使用拷贝构造函数将函数`operator*`的返回值拷贝到对象`temporary`中，最后，再调用函数`operator*`计算`temporary*B`的值。这里，尽管我们没有显式的调用拷贝构造函数，但拷贝构造函数还是被调用了多次，如果使用由编译器提供了拷贝构造函数，那么，在析构函数`matrix::~~matrix()`中释放使用`new`运算符动态分配给成员`elem`的内存时就会出现指针挂起的问题。

因此，如果在类中使用了成员指针，一般来说，我们需要给出自定义的拷贝构造函数，以保证不会出现指针挂起的问题。

关于构造函数最后简单的说一点，这就是我们可以在构造函数中使用自定义的参数表。以类matrix的构造函数为例，我们可以以下面的三种方式调用其构造函数：

```
// 创建一个 0× 0 的矩阵，事实上没有多大的意义
```

```
matrix M()
```

```
// 创建一个 2× 2 的方阵，由于 init 取默认值，方阵的所有元素被初始化为 0
```

```
matrix M(2,2)
```

```
// 创建一个 2× 3 的矩阵，并将所有元素初始化为 1
```

```
matrix M(2,3,1)
```

下面现在我们简单介绍一下析构函数，析构函数在释放对象时被调用，对象的释放发生在以下的几种情况下：

1. 使用运算符new分配的对象被显式的使用运算符delete删除；
2. 一个具有块作用域的本地(自动)对象超出其作用域；
3. 临时对象的生存期结束；
4. 程序结束运行；
5. 使用完全限定名显式调用对象的析构函数；

析构函数的函数为类名前加波浪号“~”，如前面的例子，类matrix的析构函数被命名为~matrix，同构造函数一样，析构函数不可以有任何返回值，哪怕是void也不可以。不同的是，析构函数不得带有任何参数，但是，析构函数可以被声明为虚函数，这样，即使不知道对象的确切类型，也可以通过析构函数完全的释放对象。

我们常常的析构函数中释放在构造函数中所动态分配的内存，例如前面的类matrix的示例，我们在析构函数matrix::~~matrix中释放了在构造函数中使用运算符new为指针elem分配的内存。在Windows应用程序中，我们还常常需要在类的析构函数中释放对象所占用的其它系统资源。

## 2.2.6 友元



在上一节的类matrix中出现了一个新的关键字friend，在一个类的定义中对一个函数使用friend关键字说明了该函数是类的一个友元函数。友元函数不属于为类的成员，因此类matrix中的友元函数operator+的定义是

```
matrix operator+(matrix& A,matrix& B)

{

...

}
```

而不是

```
matrix matrix::operator+(matrix& A,matrix& B)

{

...

}
```

友元函数虽然不是类的成员，但却有访问类的受保护成员和私有成员的权限，这就是friend关键字的作用。在一些情况下这是必须的，事实上，我们在类matrix定义的大多数友元运算符函数都可以定义为类的成员函数，但运算符函数operator<<是一个例外，因为定义为成员函数的二元运算符函数只能带一个参数，该参数的类型指定了位于运算符右边的操作数的类型，而位于运算符左边的操作数必须是类的一个实例对象，而对于运算符“<<”，由于左边的操作数是一个对类ostream的实例对象的引用，因此它不可以被重载为类的成员函数，而我们又需要在函数operator<<中访问类matrix的实例对象A的受保护成员m和n，因此，我们必须将函数operator<<声明为类matrix的友元函数，这样，我们既可以使用类ostream的对象作为运算符“<<”的左操作数，又可以在运算符函数operator<<中访问类matrix的受保护成员。从这个角度来说，友元函数的确为我们带来了很大的便利之处。但我们要提醒的是，不要滥用友元函数，因为如果大量的使用友元函数，那么当初把这些成员的访问权限设置为受保护或私有也就失去了应有的意义了。一般来说，我们只将friend关键字应用于运算符重载函数以及一些和类关系极为密切的类和函数上。

除了将具有文件作用域的全局函数指定为的类的友元外，还可以

将其它某个类的成员函数指定为类的友元，请看下面的例子：

```
#include <iostream.h>

class A;

class B
{
public:
    int& geti();

    B();

    B(B&);

    ~B();

    B& operator=(B&);

private:
    A *pA;

    friend class A;

};

class A
{
public:
    void addout(B& b)
    {
        cout<<"The address of A in B is "<<b.pA<<endl;
    }

private:
    int i;

    friend int& B::geti();

};
```

```

void main()

{

B b;

A a;

b.get i ()=1;

a.addout(b);

cout<<"b.pA->i="<<b.get i ()<<endl;

}

B::B()

{

pA=new A;

}

B::B(B& b)

{

```

LANG="ZH-CN" SIZE=3>视结构，在需要频繁的访问文档和视中的成员时，使文档类和视类互为友元要更为方便。

○ 注意：

○ 友元关系不可以被继承。假设类A是类B的友元，而类C从类B派生，如果我们没有在类C中显式地使用下面的语句：

○ friend class A;

那么，尽管类A是类B的友元，但这种关系不会被继承到类C，也就是说，类C和类A没有友元关系，类A的成员函数不可以直接访问类C的受保护成员和私有成员。

○ 不存在“友元的友元”这种关系。假设类A是类B的友元，而类B是类C的友元，即是说类B的成员函数可以访问类C的受保护成员和私有成员，而类A的成员函数可以访问类B的受保护成员和私有成员；但是，类A的成员函数不可以直接访问类C的受保护成员和私有成员，即是说友元关系不存在传递性。

## 2.2.7 运算符重载

由于C++中的函数重载的实现相对比较简单，而且易于理解，因此本章不再讲述C++中的函数重载。这里使用一个小节的内容来简单的讲述C++的运算符重载。

由于类也是一种数据类型，因此很自然的，我们希望能够像使用C++基本数据类型一样，使用运算符来实现以类作为操作数的运算，如前一节中的例子，对于类matrix的实例对象，我们希望使用下面的这些表达式来直观的进行矩阵的运算：

A+B+C;

A\*B\*C;

A-B\*C;

而不是去调用相应的函数。再比如对于字符串a和字符串b，我们希望使用表达式a+b来进行字符串的连接，就象在Visual Basic和Delphi中那样，而不是使用C语言的标准库函数strcat（尽管我们所重载的运算符函数内部仍可能是使用strcat来实现的），因为前一种方法很显然要直观得多。

表2.3 允许重载的运算符

运算符	名称	类型
,	逗号运算符	二元运算符
!	逻辑否(NOT)运算符	一元运算符
!=	不等于运算符	二元运算符
%	取模运算符	二元运算符
%	取模/赋值运算符	二元运算符
&	按位和(AND)运算符	二元运算符

&	取地址运算符	一元运算符
&&	逻辑和(AND)运算符	二元运算符
&=	按位和/赋值运算符	二元运算符
()	函数调用运算符	—
*	乘法运算符	二元运算符
*	指针间接引用运算符	一元运算符
*=	乘法/赋值运算符	二元运算符
+	加法运算符	二元运算符
++	递增运算符(注)	一元运算符
+=	加法/赋值运算符	二元运算符
-	减法运算符	二元运算符
-	一元负号	一元运算符
--	递减运算符(注)	一元运算符
-=	减法/赋值运算符	二元运算符

续表2.3

运算符	名称	类型
->	成员选择运算符	二元运算符
->*	指针成员选择运算符	二元运算符

/	除法运算符	二元运算符	预处理符号
/=	除法/赋值运算符	二元运算符	
<	小于运算符	二元运算符	
<<	左移运算符	二元运算符	
<<=	左移赋值运算符	二元运算符	
<=	小于等于运算符	二元运算符	
=	赋值运算符	二元运算符	
==	等于运算符	二元运算符	
>	大于运算符	二元运算符	
>=	大于等于运算符	二元运算符	
>>	右移运算符	二元运算符	
>>=	右移/赋值运算符	二元运算符	
[]	数据下标运算符	—	
^	异或(Exclusive OR)运算符	二元运算符	
^=	异或(Exclusive OR)/赋值运算符	二元运算符	
	按位或(Inclusive OR)运算符	二元运算符	
=	按位或(Inclusive OR)/赋值运算符	二元运算符	

运算符的重载是通过对运算符函数的重载来实现的，对于每一个运算符@，在C++中都对应于一个运算符函数operator@，其中的符号“@”表示表2.3中所列出的运算符，例如运算符“+”所对应的运算符函数为operator+，而数组下标运算符“[]”所对应的运算符函数为operator[]。运算符函数的一般原型为：

```
type operator@(arglist);
```

其中type为运算结果的类型，arglist为操作数列表。大多数情况下，运算符函数可以重载为类的成员函数，也可以重载为全局函数。在两种不同情况，同一运算符的重载形式所对应的参数表略有差别。以前面的类matrix的加法运算符为例，我们将加法运算符函数重载为全局函数，并把它作为类matrix的友元，以便它可以直接的访问类matrix的受保护成员，其定义如下：

```
matrix operator+(matrix& A, matrix& B);
```

上面的运算将类matrix的实例对象A和B进行相加，然后返回一个类型为matrix的运算结果。

如果上面的运算符函数被重载为类的成员函数，那么它的定义应该是这样的：

```
matrix matrix::operator+(matrix& B);
```

其中参数B代表“+”右边的操作数，而“+”左边的操作数总是重载该运算符函数的类的一个实例对象或其引用，在成员函数operator+的内部，通过隐含的this指针来对它进行引用。考虑下面的表达式

```
C=A+B;
```

如果运算符函数operator+被重载为友元全局函数

```
matrix operator+(matrix& A, matrix& B);
```

则上面的表达式被编译器解释为

```
C=operator+(A, B);
```

如果operator+被重载为类matrix的成员函数

```
matrix matrix::operator+(matrix& B);
```

则该表达式被解释为

```
C=A.operator+(B);
```

初看起来，无论把操作符operator+重载为友元全局函数还是成员函数，都可以实现同样的功能。的确，在很多情况下是这样的，但是有些情况下，我们只能选择其中之一。考虑矩阵的数乘运算，我们可能希望使用下面的表达式

```
B=3*A;
```

从数学意义上说，上面的表达式将3乘以A中的每一个元素，然后将结果赋值给B。假设operator\*的定义如下：

```
matrix matrix::operator*(matrix& A);
```

即是说我们将operator\*函数定义为类的成员函数，这时，编译器如何解释上面的代码呢？下面的解释方法是行不通的：

```
B=3.operator*(A);
```

因为3不是类matrix的一个实例对象，而且，编译器在这种情况下并不会对左边的操作数作任何类型转换，也就是说，即使你为类matrix定义了一个构造函数

```
matrix::matrix(int);
```

编译器仍然不会将前面的表达式解释为

```
B=matrix(3).operator*(A);
```

因此，将运算符函数定义为类的成员函数是不可能实现我们的要求的，这时，我们需要将函数operator\*定义为全局函数，并且，将它作为类matrix的友元，如下所示：

```
matrix operator*(int k, matrix& A);
```

这时，编译器将前面的表达式解释为

```
B=operator*(3, A);
```

由于存在合适的函数原型，因此编译器将调用上面所定义的函数operator\*来进行运算，并将结果赋予B。

上面的叙述容易给人一种感觉，即是说将运算符函数定义为友元函数要比将它们定义为类的成员函数好得多。事实上很多情况下也是这样，然而，并不是所有的函数都能够被定义为友元函数，



以下的函数只能被定义为类的成员函数：

`operator=`

`operator()`

`operator[]`

`operator->`

○ 注意：

○ 函数`operator=`只能定义为类的成员函数，但是其它的二元重合赋值运算符，如`??`、`??`、`??`和`??`等却不受此限，请看下面的代码：

两个函数中C++中是不同的重载形式。

由编译器自动生成的运算符函数`operator`所进行的默认操作是将两个对象中的数据进行按成员拷贝，有一点需要强调的是，对于其中的指针成员，拷贝的是指针本身，而不是指针所指向的内容。如果在类中使用了指针成员，这是一个必须注意的问题，一般来说，在这种情况下，我们必须提供自定义的拷贝构造函数和以`type&`为参数的赋值运算符重载函数，否则很容易引起指针挂起的问题。

表2.5总结了不同运算符的重载方法。比较特殊的是递增运算符`++`和递减运算符`--`，特殊的原因是它们有两种不同的形式，即前缀形式和后缀形式。如果区别运算符“`++`”和“`--`”的两种不同形式呢？我们为此作如下的约定，对于前缀形式的递增/递减运算符，以和一般的一元运算符同样的方式将它们重载为

`type& type::operator++()`

表2.5 不同运算符的重载方法小结

运算符	以友元函数方式进行重载	以成员函数方式进行重载
一元运算符@ (不包括递增运	<code>type operator@ (arg)</code>	<code>type operator@()</code>  表达式 <code>A@</code> 或 <code>@A</code> 等价于

算符++和递减运算符??)	表达式A@或@A等价于operator@(A)	A.operator@()
二元运算符@	type      operator@ (arg1, arg2)  表达式A@B等价于 operator@(A, B)	type operator@(arg)  表达式A@B等价于A.operator@ (B)
赋值运算符=	—	type& operator=(arg)  表达式A=B等价于A.operator= (B)
函数调用运算符()		type operator()(arg, ...)  表达式A(arg, ...)等价于 A.operator()(arg, ...)  注意：  1. 函数调用运算符被当作一个二元运算符，然而函数调用运算符函数的参数表却可以拥有多个参数。  2. 函数调用运算符作用于一个类的实例对象，而不是一个函数名。  关于函数调用运算符可以参见前面的类matrix的实现。假设A是类matrix的一个实例对象，则表达式A(1,2)返回矩阵A中第一行第二列的元素。
下标运算符[]		type operator[](arg)  表达式A[arg]等价于 A.operator[](arg)  注意：  除了可以为整数以外，下标运算符函数的参数arg还可以为任何类型，比如，你可以创建一个以字符串为下标的数据列表。
成员函数运算符->		type operator->(arg)

		表达式A->arg等价于 A.operator->(arg)  注意：  可以重载成员选择运算符“ -> ”，但不可以重载另一个成员选择运算符“ . ”。
--	--	--

type& operator++(type&)

或

type type::operator--()

type operator--(type&)

要注意的是，如果使用将operator++和operator--重载为全局友元函数，则参数要使用引用类型，这是因为一般来说，运算符“++”和“--”都需要修改操作符本身。

对于后缀形式的递增/递减运算符，我们约定使用下面的方式来进行重载：

type& type::operator++(int)

type& operator++(type&, int)

或

type type::operator--(int)

type operator--(type&, int)

也就是说，我们使用一个额外的整型参数来表明所需调用的是后缀形式的递增/递减运算符。这样，表达式

++A

对于编译器等价于

A.operator++()

或

operator++(A)

而表达式

`A++`

对于编译器等价于

`A.operator++(0)`

或

`operator++(A, 0)`

运算符“`--`”与此类似。

这样，编译器就能有效的区分前缀形式的递增/递减运算符和后缀形式的递增/递减运算符，对于Visual C++而言，传递给后缀形式的递增/递减运算符函数的整型参数为0，事实上，我们可以显式的调用后缀形式的递增/递减运算符函数，如

`A.operator++(3);`

或

`operator++(A, 3);`

这时，所传递的整型参数可以不是0，而且，后缀形式的运算符函数`operator++`也的确可以使用这个参数，这时，“`++`”运算符看起来有点像一个二元运算符，但是，要记住，后缀形式的运算符函数`operator++`只是一个约定，对于C++来说，无论使用的是前缀形式还是后缀形式，递增/递减运算符都是一个一元运算符，下面的表达式在Visual C++中是通不过的：

`A++3;`

不要想当然的将它解释为

`A.operator++(3);`

或

`operator++(A, 3);`

■ 注意：

■ 由表2.3可以知道，用于动态分配内存的运算符`new`和

delete也可以被重载，但是，C++对这两个运算符的默认实现非常之好，而且，由于Win32平台的32位平坦内存管理机制，在系统物理内存不足时会自动使用磁盘交换文件，因此，在绝大多数情况下，我们不需要、也不应该重载new和delete运算符，而且，不正确或者说不完善的重载new和delete有可能在动态分配内存时带来难以预料的严重后果。也鉴于这个原因，本书中不再讲述重载new和delete运算符的细节，若读者在程序中的确需要重载它们的话，请参考Visual C++的联机文档或其它的C++文献。

- 重载运算符时应该遵从惯例，比如说，我们可以重载运算符“+”来连接两个字符串，这是合乎我们的日常思维方式的。同样，C++允许我们使用运算符“-”来连接两个字符串，但是，这样的重载方式不会给编程带来方便，它们只会混淆程序员的思维，因此是应该避免的。

### 第三节 类的继承

类的派生(derivation)和继承(inheritance)是面向对象程序设计中的一个很重要的概念。通过这种被称作继承的机制，可以从已有的类派生出新的类，新的类包括了已有类的全部接口(interface)，而且还可以包括特有的新的接口，从而实现我们在2.1.3小节中提出的继承结构。本节讲述使用C++编程语言来实现继承机制的具体的语法细节。

在C++中，若我们从类A派生出新的类B，那么，称类A为类B的基类(base class)，而类B为类A的派生类(derived class)

#### 2.3.1 单一派生

所谓的单一派生是指一个派生类只能有一个直接基类。什么叫直接基类呢？假设我们从类A派生出类B，然后又从类B派生出类C，那么，类B就是类C的直接基类，与之相对比，我们称类A为类C的间接基类。这一关系可以用图2.5表示。

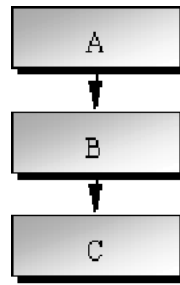


图2. 5 单一继承结构

下面的代码实现了上面的结构：

```
class A
{
    // 在此添加类A的实现
    // ...
};

class B:public A
{
    // 在此添加类 B 特有的实现
    // ...
};
```

由上面的代码可以看出，类的单一派生使用如下的语法：

```
class class-name : access-specifier base-class
{
    // 派生类的新增成员
};
```

其中class-name是派生类的类名，access-specifier指定了基类的继承方式，可以使用的关键字包括public（如前面的例子所示）、private和protected，它们的作用我们将在下面作说明，base-class是基类的类名。

同指定类中的成员的访问权限相仿，我们可以使用访问限定关键字指定派生类对基类成员的访问权限。对于使用不同方

式(public、private和protected)派生的类，对于基类中以不同方式(public、private和protected)定义的成员的访问限制是不同的。我们在前面讲述类中成员的访问限制时已在前面的表2.2中已经给出了不同派生方式不区别，为了叙述方便，这里将与本节有关的内容重新列于表2.6。

表2.6 不同派生方式得到的派生类对基类成员的访问权限

基类成员所使用的关键字	在派生类中基类的继承方式	派生类对基类成员访问权限
public (公有成员)	public	相当于使用了public关键字
	protected	相当于使用了protected关键字
	private	相当于使用了private关键字
protected (受保护成员)	public	相当于使用了protected关键字
	protected	相当于使用了protected关键字
	private	相当于使用了private关键字
private (私有成员)	public	不可访问
	protected	不可访问
	private	不可访问

从表2.6可以看出，无论使用哪种方式生成的派生类，其成员函数都可以访问基类中除了以private关键进行限定的以外的其它成员。但是，对于以public方式生成的派生类，在基类中使用public和protected关键字限定的成员，在派生类中仍相当于使用了public和protected关键字；对于以protected方式生成的派生类，在基类中使用public和protected关键字限定的成员，在派生类中都相当于使用了protected关键字；而对于以private方式生成的派生类，它们都相当于使用了private关键字。

■ 注意：

- 无论使用的是哪种继承方式，在基类中以private关键字进行限定的成员在派生类中都是不可以访问的，这和以protected关键字定义的成员有着很大的区别，因此，如果希望成员能够为派生类所访问，而同时又不希望被类外部的其它函数直接访问，那么应该使用的访问限定符是protected，而不是private。

### 2.3.2 多重继承

前面说过，当一个派生类只有一个基类时，我们称这种继承方式为单一继承。事实上，在C++中，我们还允许一个派生类具有多于一个的基类，这种派生方式被称为多重继承(multiple inheritance)。多重继承是单一继承的自然扩展，两者既有同一性，又有特殊性。图2.6给出了一个多重继承的示例。在这个示例中，类Software由两个类OperatingSystem和Application派生，而类ComputerSystem又由类Software和另一个类Hardware派生。

实现该示例的C++语言代码如下：

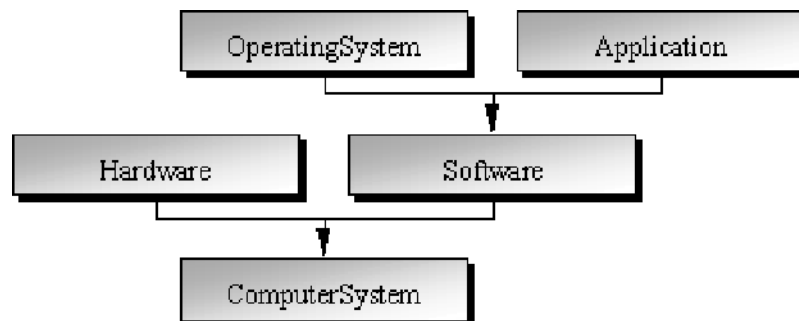


图2. 6 多重继承结构

```
#include <string.h>

#include <iostream.h>

#include <iomanip.h>

class OperatingSystem
{
private:
char OsName[80];
```



```

public:
    const char *GetOsName() const
    {
        return OsName;
    }

    OperatingSystem()
    {
    }

    OperatingSystem(char *OsName)
    {
        strcpy(OperatingSystem::OsName, OsName);
    }

};

class Application
{
private:
    char *(AppName[80]);
    char n;
public:
    const char *GetAppName(int i) const
    {
        return AppName[i];
    }

    int AddAppName(char *AppName)
    {
        char *NewAppName=new char[80];

```

```

strcpy(NewAppName,AppName);

Application::AppName[++n]=NewAppName;

return n;

}

int GetAppCount()

{

return n;

}

Application()

{

n=0;

}

~Application()

{

for (int i=1;i<=n;i++)

delete AppName[i];

}

};

class Software : public OperatingSystem, public Application

{

public:

Software(char *OsName)

: OperatingSystem(OsName),Application()

{

}

};

```

```

class Hardware
{
private:
char CpuType[80];
public:
void SetCpuType(char* CpuType)
{
strcpy(Hardware::CpuType,CpuType);
}
char *GetCpuType()
{
return CpuType;
}
Hardware()
{
}
};

class ComputerSystem : public Hardware, public Software
{
public:
ComputerSystem(char *CpuType,char *OsName)
: Hardware(),Software(OsName)
{
SetCpuType(CpuType);
}
};

```

```

void main()

{

ComputerSystem MyComputer("AMD K6-200","Windows NT 4.0");

cout<<"My computer's CPU: "<<MyComputer.GetCpuType()<<endl;

cout<<"Operating System: "<<MyComputer.GetOsName()<<endl;

MyComputer.AddAppName("Microsoft Visual C++");

MyComputer.AddAppName("Microsoft Word 97");

cout<<MyComputer.GetAppCount()<<" applications:"<<endl;

for (int i=1;i<=MyComputer.GetAppCount();i++)

cout<<"\t"<<MyComputer.GetAppName(i)<<endl;

}

```

声明使用了多重继承结构的类的语法如下：

```

class class-name : access-specifier1 base-class1, access-specifier2
base-class2, ...

{

// 派生类的新增成员

}

```

多个直接基类之间以逗号作为分隔符。

■ 注意：

- 在构造类class-name的对象时，编译器将按声明类时使用的基类的顺序(即按base-class1，base-class2，...的顺序)依次构造所有直接基类的子对象，然后构造派生类；而在释放类class-name的对象恰恰相反。因此，如果类class-name的构造不依赖其基类子对象的构造的先后次序，那么我们可以使用任意的顺序来给出类class-name的所有直接基类。反之，我们必须考虑在class-name的声明中基类的给出顺序。

我们看到类Software的构造函数使用了下面的语法格式：

```
Software::Software(char *OsName) : OperatingSystem(OsName), Application  
(  
  
{  
  
}
```

上面的定义表明在构造类Software的对象之前，先以OsName作为参数构造类型为OperatingSystem的子对象，以空参数构造类型为Application的子对象，然后再构造类Software。类ComputerSystem的构造函数也具有类似的结构。

现在我们来查看一种特殊情况，请观察下面的代码：

```
#include <iostream.h>  
  
#include <iomanip.h>  
  
class A  
  
{  
  
public:  
  
int a;  
  
};  
  
class B : public A  
  
{  
  
public:  
  
void SetA(int i)  
  
{  
  
a=i;  
  
}  
  
int GetA()  
  
{  
  
return a;  
  
}
```

```

};

class C : public A
{
public:
void SetA(int i)
{
a=i;
}

int GetA()
{
return a;
}
};

class D : public B, public C
{
};

void main()
{
D d;

d.B::SetA(3);
d.C::SetA(1);

cout<<d.B::GetA()<<endl;
cout<<d.C::GetA()<<endl;
}

```

■ 注意：

- 上面的类D可以导致成员名的二义性，这是因为在类B和

类C中包含了同名成员函数GetA和SetA。关于二义性的详细讨论将在本节的后面部分进行。

在上面的例子中，类B和类C都从类A派生，而类D从类B和类C通过多重继承机制派生。这样，类D中就包括了两个不同的类型为A的子对象，上面的程序的运行结果如下：

3

1

上面的结果表明，我们分别通过类B和类C的成员函数修改从基类继承的成员a的内容时，它们是分别对两个不同的子对象进行的。很自然的想到，如果我们在main函数中使用如d.a之类的表达式将会导致二义性，因为编译器不知道应该访问的是来自类B的成员a呢，还是来自类C的成员a。

在很多情况下，我们并不需要两个类A的子对象，这时，可以将A声明为虚基类，从而保证在多重派生时，只有一个类A的子对象存在。声明派生类时通过在基类的前面加上virtual关键字可以将一个基类声明为虚基类。为了便于对比，我们给出修改后的代码如下：

```
#include <iostream.h>

#include <iomanip.h>

class A

{

public:

    int a;

};

class B : virtual public A

{

public:

    void SetA(int i)

{
```

```
a=i;

}

int GetA()

{

return a;

}

};

class C : virtual public A

{

public:

void SetA(int i)

{

a=i;

}

int GetA()

{

return a;

}

};

class D : public B, public C

{

};

void main()

{

D d;

d.B::SetA(3);
```



```

d.C::SetA(1);

cout<<d.B::GetA()<<endl;

cout<<d.C::GetA()<<endl;

}

```

上面的程序的运行结果如下：

```

1
1

```

我们看到，无论是通过类B还是类C的子对象来修改成员a的值，它们实际上都是对同一个子对象进行操作。这时，如果使用d.a的表达式是不会导致二义性的，因为在类D的实例对象中只存在一个类型A的子对象。

相比非虚基类而言，虚基类将占用更小的内存空间，但是，如上所述，所有对于虚基类子对象的操作都是对同一个子对象进行的，这在某些情况下正是我们所不期望的。另外，使用虚基类将会给程序带来额外的开销，虽然就一般情况而言，这种开销对程序的运行效率所产生的影响非常之小。

由于在使用多重继承时，派生类可以以不同的途径来从基类中继承成员名，因此就有可能出现二义性。如下面的例子：

```

#include <iostream.h>

#include <iomanip.h>

class A
{
public:
    int a()
    {
        return 1;
    }
};

```

```

class B
{
public:
float a()
{
return float(1.2345);
}
};

class C : public A, public B
{
};

void main()
{
C c;

cout<<c.a()<<endl;
}

```

由于在类A和类B中都包括了名为a的公有成员函数，而类C由类A和类B通过多重继承机制派生，这样，如果我们使用了c.a()这样的表达式，编译器无法知道我们想调用的究竟是由类A继承过来的函数a呢，还是从类B继承过来的函数a，这样就导致了二义性。在上面的这种情况下，编译器将生成一个错误。

然而我们可以使用作用域限定符“::”来明确的告诉编译器调用的函数a是从类A继承的还是从类B继承的，在前面讲述虚基类时我们已经给出了一个这样的例子。按下面的方法来修改main函数的代码：

```

void main()
{
C c;

```

```
cout<<c.A::a()<<endl;

cout<<c.B::a()<<endl;

}
```

编译器能够正确的编译上面的代码，而且，程序的运行结果也是完全正确的。然而，尽管可以使用作用域限定符来解决二义性的问题，但是，我们仍然不建议这样做，在使用多重继承机制时，最好还是保证所有的成员都不存在二义性问题。

- 注意：

- 下面的代码仍然会导致二义性：

- #include <iostream.h>

- #include <iomanip.h>

- 

- class A

- {

- private:

- int a()

- {

- return 1;

- }

- };

- 

- class B

- {

- public:

- float a()

```

■ {

■ return float(1.2345);

■ }

■ };

■

■ class C : public A, public B

■ {

■ };

■

■ void main()

■ {

■ C c;

■ cout<<c.a()<<endl;

■ }

```

并不因为在类C中无法访问类A中的成员函数a（因为该成员函数的访问权限被限定为private）而使得编译器调用从类B继承的成员函数a，编译器在编译上面的代码时仍将给出二义性的错误。

■ 下面的代码不会导致二义性：

```

■ #include <iostream.h>

■ #include <iomanip.h>

■

■ class A

■ {

■ public:

```

```
■ int a()

■ {

■ return 1;

■ }

■ };

■

■ class B : virtual public A

■ {

■ public:

■ float a()

■ {

■ return float(1.2345);

■ }

■ };

■

■ class C : virtual public A

■ {

■ };

■

■ class D : public B, public C

■ {

■ };

■

■ void main()
```

```

■ {

■ D d;

■ cout<<d.a()<<endl;

■ }

```

如果同一个成员名在两个具有继承关系的类中进行了定义，那么，在派生类中所定义的成员名具有支配地位。在出现二义性时，如果存在具有支配地位的成员名，那么编译器将使用这一成员，而不是给出错误信息。以上面的代码为例，在类A和类B中都定义了具有相同参数的成员函数a，这样，尽管类D中可以以两种方式来解释成员函数名a——即来自类B的成员函数a和来自类C的成员函数a，但是，按照刚才所说的规则，类B的成员名a相比类A的成员名a（即是类C中的成员名a）处于支配地位，这样，编译器将调用类B的成员函数a，而不产生二义性的错误。

## 第四节 多态与虚函数

先考虑下面的代码：

```

#include <iostream.h>

#include <iomanip.h>

class Creature
{
public:
char *KindOf()
{
return "Creature";
}
};

class Animal : public Creature

```

```
{  
public:  
char *KindOf()  
{  
    return "Animal";  
}  
};  
  
class Fish : public Creature  
{  
public:  
char *KindOf()  
{  
    return "Fish";  
}  
};  
  
void main()  
{  
    Animal animal;  
    Fish fish;  
    Creature *pCreature;  
    Animal *pAnimal=&animal;  
    Fish *pFish=&fish;  
    pCreature=pAnimal;  
    cout<<"pAnimal ->KindOf(): "<<pAnimal->KindOf()<<endl;  
    cout<<"pCreature->KindOf(): "<<pCreature->KindOf()<<endl;  
    pCreature=pFish;
```

```
cout<<"pFish->KindOf(): "<<pFish->KindOf()<<endl;  
cout<<"pCreature->KindOf(): "<<pCreature->KindOf()<<endl;  
}
```

程序的输出结果如下：

```
pAnimal->KindOf(): Animal  
pCreature->KindOf(): Creature  
pFish->KindOf(): Fish  
pCreature->KindOf(): Creature
```

我们看到，无论pCreature指向的对象是什么，使用表达式pCreature->KindOf()调用的总是在类pCreature中所定义的成员函数KindOf。在这种情况下，我们更希望当pCreature指向类Animal的实例对象，该表达式调用的是类Animal中定义的成员函数KindOf；而当pCreature指向类Fish的实例对象时，该表达式调用的是类Fish中定义的成员函数KindOf，这也正是面向对象程序中的多态性的要求。

### 2.4.1 虚函数

以C++中，我们是通过将一个函数定义成虚函数来实现运行时的多态的。如果一个函数被定义为虚函数，那么，即使是使用指向基类对象的指针来调用该成员函数，C++也能保证所调用的是正确的特定于实际对象的成员函数。作为对比，我们使用虚函数的概念重写上面的程序代码如下：

```
#include <iostream.h>  
  
#include <iomanip.h>  
  
class Creature  
{  
public:  
virtual char *KindOf()  
{  
return "Creature";  
}}
```



```
}  
};  
  
class Animal : public Creature  
{  
public:  
    char *KindOf()  
    {  
        return "Animal";  
    }  
};  
  
class Fish : public Creature  
{  
public:  
    char *KindOf()  
    {  
        return "Fish";  
    }  
};  
  
void main()  
{  
    Animal animal;  
    Fish fish;  
    Creature *pCreature;  
    Animal *pAnimal=&animal;  
    Fish *pFish=&fish;  
    pCreature=pAnimal;
```

```

cout<<"pAnimal->KindOf(): "<<pAnimal->KindOf()<<endl;

cout<<"pCreature->KindOf(): "<<pCreature->KindOf()<<endl;

pCreature=pFish;

cout<<"pFish->KindOf(): "<<pFish->KindOf()<<endl;

cout<<"pCreature->KindOf(): "<<pCreature->KindOf()<<endl;

}

```

将一个成员函数声明为虚成员函数的方法是在基类中声明成员函数时使用virtual关键字。一旦一个函数在基类中第一次声明时使用了virtual关键字，那么，当派生类重载该成员函数是，无论是否使用了virtual关键字，该成员函数都将被看作一个虚函数，也就是说，虚函数的重载函数仍是虚函数。

## 2.4.2 纯虚函数与抽象类

一个虚函数可以被声明为纯虚函数，方法是在函数的声明之后使用纯虚函数标识符“=0”，下面的代码在类Creature中将虚函数KindOf声明为纯虚函数：

```

class Creature

{

public:

virtual char *KindOf()=0;

};

char *Creature::KindOf()

{

return "Creature";

}

```

使用下面的格式也是可以的：

```

class Creature

{

```

```
public:

virtual char *KindOf()=0

{

return "Creature";

}

};
```

两种不同格式之间的区别已以本章的前面作了说明。

包括了至少一个纯虚函数的类被看作抽象类，一个抽象类不可以用来创建对象，这只能用来为派生类提供了一个接口规范，派生类中必须重载基类中的纯虚函数，否则它仍将被看作一个抽象类。

如果要直接调用抽象类中定义的纯虚函数，必须使用完全限定名，如上面的示例，要想直接调用抽象类Creature中定义的纯虚函数，应该使用下面的格式：

```
cout<<pCreature->Creature::KindOf()<<endl;
```

上面的代码同时还给出了一种绕过虚函数机制的方法，即使使用带有作用域限定符的完全限定函数名。

抽象类具有下面的一些限制：

- 不可以用来创建对象
- 不可以作为函数返回值的类型
- 不可以作为函数参数的类型
- 不可以用来进行显式类型转换

而且，如果在抽象类的构造函数中调用了纯虚函数，那么，其结果是不确定的。还有，由于抽象类的析构函数可以被声明为纯虚函数，这时，我们应该至少提供该析构函数的一个实现。一个很好的实现方式是在抽象类中提供一个默认的析构函数，该析构函数保证至少有析构函数的一个实现存在。如下面的例子所示：

```
class classname  
  
{  
  
    // 其它成员  
  
public:  
  
    ~classname()=0  
  
    {  
  
        // 在此添加析构函数的代码  
  
    }  
  
};
```

由于派生类的析构函数不可能和抽象类的析构函数同名，因此，提供一个默认的析构函数的实现是完全必要的。这也是纯虚析构函数和其它纯虚成员函数的一个最大的不同之处。一般情况下，抽象类的析构函数是有派生类的实现对象释放时由派生类的析构函数隐含的所调用的。

## 第五节 ClassView和WizardBar

Visual C++在Microsoft Developer Studio中提供了几个工具来使用得创建和组织类非常的方便。这些工具包括ClassView、WizardBar和ClassWizard。其中ClassWizard只能用于基于MFC应用程序，因此我们将在以后的部分中讲述它的使用，在本节中，只涉及了ClassView和WizardBar中的部分内容。

### 2.5.1 使用ClassView

使用ClassView可以完成以下工作：

- 向工程中添加新类
- 向类中添加成员函数和成员变量
- 重载基类中的虚函数
- 添加消息处理程序
- 跳转到类或成员的定义

- 跳转到类或成员的引用
- 显示派生类或基类的图形
- 将类添加到Gallery中以便以后重用
- 在成员函数中设置断点
- 使用访问类型组织类的成员
- 向COM接口中添加属性和方法
- 向ActiveX控件添加事件
- 在工程空间中创建新的文件夹
- 察看类及其成员的属性



图2. 7 工程ClassViewDemo的ClassView窗格

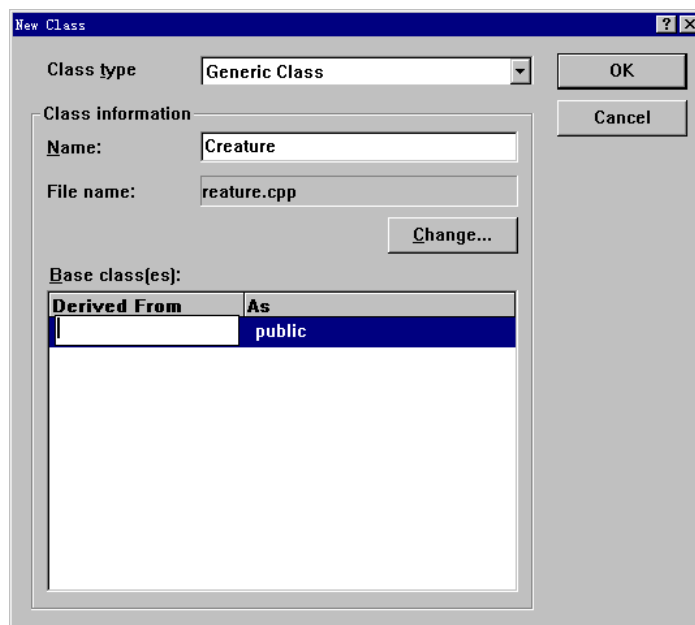


图2.8 新建类的对话框

为了进行下面的示例，首先创建一个空的Win32 Console Application工程，这里，我们假定工程名为ClassViewDemo，当然，你也可以使用其它的工程名。

在Workspace窗口中打开ClassView窗格，如图2.7所示。下面我们使用ClassView向工程中添加一个类Creature。方法如下：

1. 右击ClassViewDemo classes节点，选择New Class命令，出现如图2.8的对话框。
2. 在Name处输入新的类名，确认在Class Type中选择了Generic Class，对于Win32 Console Application，这也是唯一可用的选项。
3. 如果需要，可以在下面的Base class(es)中选择类Creature的基类和继承方式。这里，我们不需要为类Creature指定基类。
4. 单击File name处的Change按钮改变类的头文件和实现文件为creature.h和creature.cpp。
5. 最后，单击OK完成。

这时，Visual C++自动生成了下面的两个文件：

creature.h:

```
// creature.h: interface for the Creature class.

//

////////////////////////////////////////////////////////////////

#ifndef AFX_CREATURE_H__90147538_FA97_11D1_BBF0_0000B4810A31__INCLUDED_
#define AFX_CREATURE_H__90147538_FA97_11D1_BBF0_0000B4810A31__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```

class Creature

{

public:

Creature();

virtual ~Creature();

};

#endif // !defined
(AFX_CREATURE_H__90147538_FA97_11D1_BBF0_0000B4810A31__INCLUDED_)

```

**creature.cpp:**

```

// creature.cpp: implementation of the Creature class.

//

////////////////////////////////////////////////////////////////

#include ""

#include "creature.h"

////////////////////////////////////////////////////////////////

// Construction/Destruction

////////////////////////////////////////////////////////////////

Creature::Creature()

{

}

Creature::~~Creature()

{

}

```

这里，我们需要删掉creature.cpp文件中的

```
#include ""
```

一行。

在上面的过程中，还自动生成的类Creature的构造函数和析构函数的框架，并且，其析构函数还被声明为虚函数。

下面的步骤添加类Creature的成员函数KindOf：

1. 在ClassView中右击类Creature，选择Add Member Function，弹出如图所示的对话框。



图2.9 添加成员函数

2. 在Function Type处输入char \*。然后在对话框下方选择Virtual复选框，这导致Static复选框不可用，因为同一个成员函数不可能既是虚函数，又是静态函数，这时，在Function Declaration处自动添加上了virtual关键字。确认在Access处选择了Public，然后，在Function Declaration处输入函数名KindOf，单击OK完成。

双击ClassView中的KindOf，Microsoft Developer Studio将打开文件creature.cpp，将将插入点定位到函数KindOf的定义处。在此添加下面的代码：

```
char* Creature::KindOf()
{
    return "Creature";
}
```

下面以类似的方式生成类Animal，使其基类为Creature，继承方式为public。并将其头文件改为animal.h，实现文件改为animal.cpp。同样，在animal.cpp中删除

```
#include ""
```

一行。



对于MFC应用程序，还可以使用ClassView来重载基类中的虚函数，这将在本书后面的章节中讲述。而对于非MFC应用程序，我们仍可以使用上面的添加成员函数的方法来重载基类中的虚函数，只不过这时必须手动的给出虚函数的返回类型和参数列表。

使用上面的方法来生成为Animal的重载虚函数KindOf，其代码已在上一节中给出。

然后再新添加一个C++ Source File——main.cpp，并在其中定义程序的主函数main()。

由于我们目前还没有接触到MFC编程，因此，现在还不能讲解ClassView的某此用法。在本书后面的章节中，我们将会用到这些功能的时候讲解它们。

## 2.5.2 使用WizardBar

WizardBar是Microsoft Developer Studio中的一个工具条，如图2.10所示。它提供了对ClassView和ClassWizard（关于ClassWizard的使用将在后面的章节中讲述）中的命令的快速访问。



图2.10 WizardBar

WizardBar工具条中的显示的内容是与当前上下文相关联的，也就是说，随着当前上下文的改变，WizardBar的显示也会改变。图2.10是在ClassView中双击类Animal的构造函数之后的WizardBar。

可以使用WizardBar来完成下面的这些操作：

- 跳转到函数的定义(实现)处
- 跳转到函数的声明处
- 跳转到类的定义处
- 添加窗口消息处理函数
- 添加虚函数

- 添加成员函数
- 添加新类
- 跳转到文件中的下一个函数
- 跳转到文件中的前一个函数
- 打开包含文件
- 获得关于WizardBar的帮助

WizardBar中最左边的下拉列表框被称作类列表(Class List)，它列举了当前工程中所有可用的类；中间的下拉列表框被称作过滤器和成员列表(Filter和Member List)，对于Visual C++工程，可以通过过滤器和成员列表指定在WizardBar中列出的内容，即是列出所有的类成员，还是与特定的资源ID相关联的成员；最右边的部分被称作WizardBar行为控件，包括两个部分，一个按钮和位于按钮右边的下箭头。单击下箭头可以出现如图2.11所示的下拉菜单，菜单中各项的意义非常之直观，这里不再赘述。

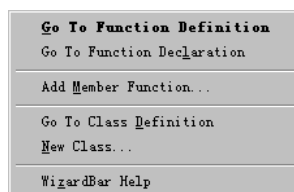


图2. 11 WizardBar行为菜单

注意在图2.11中被以粗体显示的菜单项，这些项被称为WizardBar默认行为。在以下几种情况下，WizardBar的默认行为被执行：

- 单击了下箭头左边的图标按钮
- 在WizardBar的下拉列表中按下了回车键
- 在对话框编辑器中双击了某个控件

当不同的WizardBar的下拉列表框获得输入焦点时，WizardBar采取不同的默认行为，如表所示。

表2. 7 不同的下拉列表框具有输入焦点时WizardBar的不同默认行为

获得输入焦点的下拉列表框	WizardBar采用的默认行为
类列表	跳转到选定的类中以字母排序的第一个成员。如果类中没有实现任何函数和方法，则弹出一个对话框为类创建新的成员函数和方法。对于C++全局类(global classes)，该对话框提示创建新的类。
过滤器和成员列表	如果位于一个资源ID上，默认行为为跳转到以字母排序的第一个消息的处理函数处。如果没有创建任何消息处理函数，则弹出一个对话框以创建新的消息处理函数。
成员列表	默认行为为跳转到选定的成员的定义处。

为了更快的访问WizardBar提供的功能，可以为WizardBar类列表、过滤器列表、成员列表和行为按钮设置快捷键，当按下所设置的快捷键时，WizardBar中的相应元素获得输入焦点；也可以为在下拉列表中给出的其它WizardBar行为设置相应的快捷键，当这些快捷键被按下时，相应的行为将被执行。为WizardBar设置快捷键的方法如下：

1. 选择Tools菜单下的Customize命令，打开如图2.12所示的对话框。
2. 在Category下拉列表框中选择WizardBar。然后，在Commands列表框中选择需要设置快捷键的命令。这时，在Description处就会出现该命令的相应的简单的英文解释，因此，我们这里无需再对全部的命令给出详细的说明，其中对应于WizardBar类列表、过滤器列表、成员列表和行为按钮的命令分别为WBActionButtonActive、WBClassComboActive、WBFilterComboActive、WBMemberComboActive。
3. 然后单击Press new shortcut，并按下新的快捷键，如果新的快捷键可用，则它将出现在Press new shortcut文本框中，单击Assign按钮，即可将新的快捷键与指定的命令相关联。

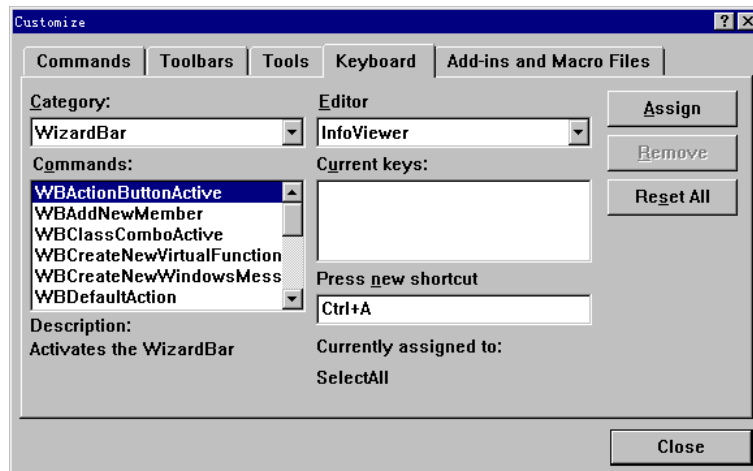


图2. 12 为WizardBar设置快捷键

- 注意：
- 不可以使用那些已被系统保留的快捷键，如ESC、F1、Ctrl+Alt+Del等。
- 如果所设置的快捷键已被其它命令使用，则在Press new shortcut下方的Currently assigned to处将给出与该快捷键相关联的当前命令，如果这时按下Assign按钮，则该快捷键将与新设定的命令相关联，与原命令的关联将自动失效。单击Reset All可以恢复Microsoft Developer Studio原有的默认设置。

## 第三章 Win32应用程序设计

在过去，进行Windows程序设计是一件痛苦异常的事情，原因是那时候还没有现在的这些设计精美的应用程序开发工具。在今天，一个对Windows程序运行的内部机制几乎一无所知的初入门者，只需要通过不到一天的学习，也可以使用如Visual Basic之类的程序开发工具创建出功能完整的Windows应用程序。这在几年前还是一件不可思议的事，因为即使是一个熟练掌握C语言的程序员，在当时差不多需要半年的学习才可以较全面的掌握Windows的编程技术，而且，与在DOS环境下编程相比，急剧膨胀的代码大大增加了程序调试的困难，从而使得编写一个出色的Windows应用程序要比编写一个出色的DOS需要考虑多得多的东西。

在Microsoft的另一种易学易用的编程工具Visual Basic中，从某种角度说，Windows程序不是编出来的，而是由程序员画出来的。但是要想知道，一个出色的Windows的应用程序并不仅在于在屏幕上绘出程序的各个窗口和在窗口中恰当的安排每一个控件。对于具有一定基础的程序员而言，更重要的内容在于知道Windows和Windows应用程序的运行机制，以及它们之间以何种方式来进行通信，然而，明确自己在编写Windows时所需做的工作是哪一些。换句话说，我们需要透过Windows漂亮的图形用户界面，认清在底层所发生的每一件事情。然而，这并非是一件容易的事。虽然，使用MFC和AppWizard，我们仍可能只需要回答几个简单的问题和添加少数的几条代码就能够生成功能完整的Windows应用程序。但是记住，没有一个成功的商业软件是使用这样的方式生成的。同时，也只有深入的理解了MFC应用程序框架的运行机制，才可能用好和用活这一工具，才能达到熟悉掌握Visual C++的境界。

尽管说MFC应用程序框架提供的是面向对象的Windows编程接口，这和传统的使用C语言和SDK来进行的Windows应用程序设计有着很大的不同，但是从底层来说，其中的大部分功能仍是通过调用最基本的Win32 API来实现的。其中最重要的一点是，Windows应用程序的运行机制仍然没有改变，它们仍然是通过消息来和操作系统，进而和用户进行交互的事件驱动的应用程序。MFC对这一切进行了比较彻底的封装，它们隐藏在你所看不见的背面。即使你对这一切一无所知，你仍可以在Visual C++中使用MFC来进行程序设计。但是，经验表明，理解这一切的最好的方式是回过头去，看一看这些内容在SDK编写的应用程序是如何实现的，然后，再看一看在MFC中是如何把它们一层一层的与程序员隔离开的。

因此，在本章中介绍相对已“过时”的Win32 SDK编程，并非是说以后也使用SDK来编写应用程序，而在于让你通过它们更深入的从MFC的内部了解MFC，并且，对于某些术语和概念的说明和澄清，也有助于你以后理解很多的东西。如果你一开始对这些东西不感兴趣，那么，你可以先暂时跳过此章，继续阅读本书的其它部分。当你对于MFC中的某些问题感到不解，或者想知道MFC究竟是如何工作的时候，再回过头来补充这些知识，也是完全可以的。

本章包括以下的内容：

- Windows应用程序的消息处理
- Win32 API和SDK
- WinMain函数
- 窗口和窗口过程
- 32位编程的特点

## 第一节 事件驱动的应用程序

类似的话已在很多书籍中说过了无数遍，以至于每一个正在或试图进行Windows编程的人都耳熟能详：Windows应用程序是事件驱动(或称作消息驱动)的应用程序。Windows是一个多任务的操作系统，也就是说，在同一时刻，在Windows中有着多个应用程序的实例正在运行，比如说这时我正在打开字处理软件Word来编写这本书的书稿，同时，还打开了Visual C++的集成开发环境Microsoft Developer Studio来调试书中的示例程序，而且，后台还在放着歌曲。在这样的一个操作系统中，不可能像过去的DOS那样，由一个应用程序来享用所有的系统资源，这些资源是由Windows统一管理的。那么，特定的应用程序如何获得用户输入的信息呢？事实上，Windows时刻监视着用户的一举一动，并分析用户的动作与哪一个应用程序相关，然后，将用户的动作以消息的形式发送给该应用程序，应用程序时刻等待着消息的到来，一旦发现它的消息队列中有未处理的消息，就获取并分析该消息，最后，应用程序根据消息所包含的内容采取适当的动作来响应用户所作的操作。举一个例子来说明上面的问题，假设我们编了一个程序，该程序有一个File菜单，那么，在运行该应用程序的时候，如果用户单击了File菜单，这个动作将被Windows（而不是应用程序本身！）所捕获，Windows经过分析得知这个动作应该由上面所说的那个应用程序去处理，既然是这样，Windows就发送了个叫做WM\_COMMAND

的消息给应用程序，该消息所包含的信息告诉应用程序：“用户单击了File菜单”，应用程序得知这一消息之后，采取相应的动作来响应它，这个过程称为消息处理。Windows为每一个应用程序(确切地说是每一个线程)维护了相应的消息队列，应用程序的任务就是不停的从它的消息队列中获取消息，分析消息和处理消息，直到一条接到叫做WM\_QUIT消息为止，这个过程通常是由一种叫做消息循环的程序结构来实现的。

Windows所能向应用程序发送的消息多达数百种，但是，对于一般的应用程序来说，只是其中的一部分有意义，举一个例子，如果你的应用程序只使用鼠标，那么如WM\_KEYUP、WM\_KEYDOWN和WM\_CHAR等消息就没有任何意义，也就是说，应用程序中事实上不需要处理这些事件，对于这些事件，只需要交给Windows作默认的处理即可。因此，在应用程序中，我们需要处理的事件只是所有事件中的一小部分。

图3.1给出了一般Windows应用程序的执行流程。

因此，从某种角度上来看，Windows应用程序是由一系列的消息处理代码来实现的。这和传统的过程式编程方法很不一样，编程者只能够预测用户所利用应用程序用户界面对象所进行的操作以及为这些操作编写处理代码，却不可以这些操作在什么时候发生或者是以什么顺序来发生，也就是说，我们不可能知道什么消息会在什么时候以什么顺序来临。

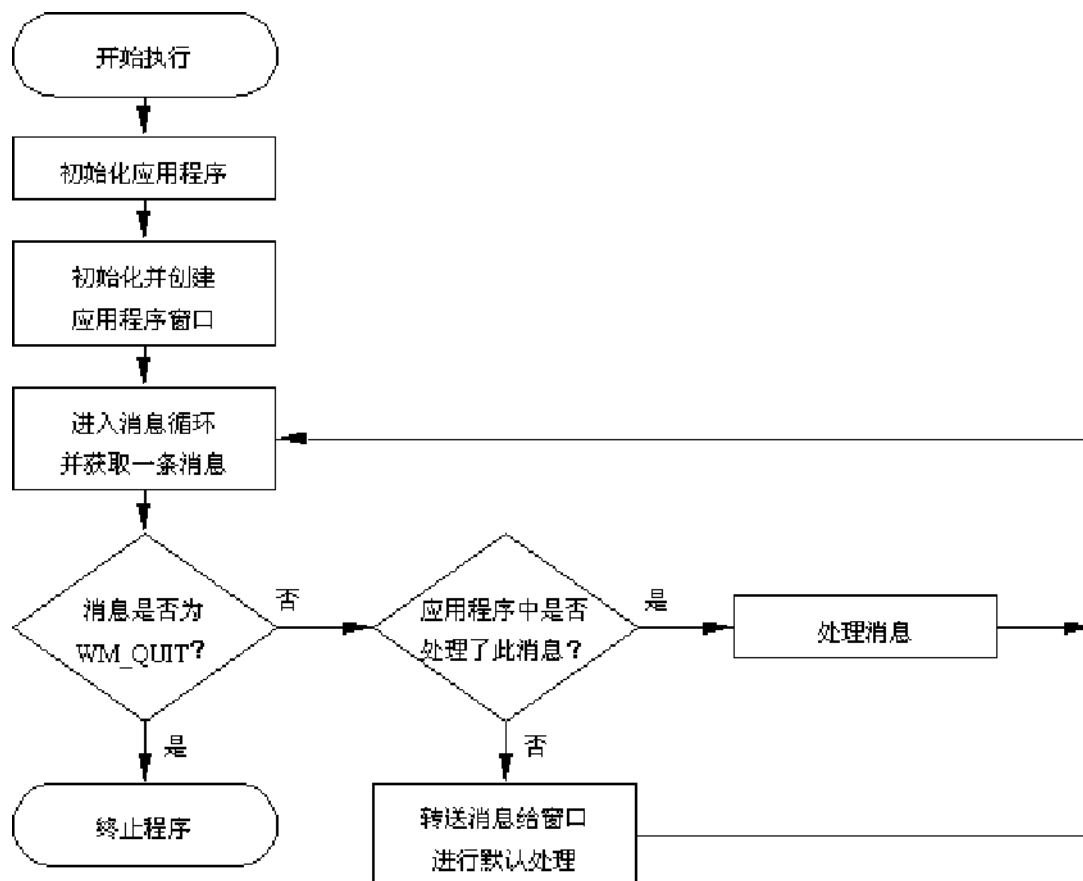


图3. 1 Windows应用程序的基本流程

Windows程序在处理消息时使用了一种叫做回调函数(callback function)的特殊函数。回调函数由应用程序定义,但是,在应用程序中并没有调用回调函数的代码,回调函数是供操作系统或者其子系统调用的,这种调用通常发生在某一事件发生,或者在窗口或字体被枚举时。典型的回调函数有窗口过程、对话框过程和钩子函数。其中的窗口过程和对话框过程将在本章后面的内容中讲述。

## 第二节 Win32 API和SDK

说到Windows编程,就不能不谈到Windows API (Windows Application Programming Interface, Windows应用程序编程接口),它是所有Windows应用程序的根本之所在。简单的说,API就是一系列的例程,应用程序通过调用这些例程来请求操作系统完成一些低级服务。在Windows这样的图形用户界面中,应用程序的窗口、图标、菜单和对话框等就是由API来管理和维护的。

Windows API具有两种基本类型:Win16 API和Win32 API。两者在很多方面非常相像,但是Win32 API除了几乎包括了Win16 API中的所有内容以外,还包括很多的其它内容。Windows API依靠三个主要的核心组件提供Windows的大部分函数,在Win16和Win32中,它们具有不



同的名称，如表3.1所示。

表3. 1 Win16和Win32的核心组件

Win16 API	Win32 API	说明
USER.EXE	USER32.DLL	负责窗口的管理，包括消息、菜单、光标、通信、计时器和其它与控制窗口显示
GDI.EXE	GDI32.DLL	提供图形设备接口，管理用户界面和图形绘制，包括Windows元文件、位图、设备描述表和字体等
KRNL386.EXE	KERNEL32.DLL	处理存储器低层功能、任务和资源管理等Windows核心服务

虽然Win16 API组件带有.EXE的扩展名，但是它们事实都是动态链接库(.DLL)，不能单独运行。其它一些非核心的Windows API由其它组件所提供的DLL来实现，这些组件包括通用对话框、打印、文件压缩、版本控制以及多媒体支持等。

Windows SDK (Windows Software Development Kit, Windows软件开发工具包)和Windows API紧密联系，它是一套帮助C语言程序员创建Windows应用程序的工具，在Windows SDK中包括了以下几个组成部分：

- 大量的在线帮助，这些帮助描述了Windows编程所可能用到的函数、消息、结构、宏及其它资源
- 各种编程工具，如对话框编辑器及图象编辑器等
- Windows库及头文件
- 使用C语言编写的示例程序

该工具包的最新版本就是我们正在使用的Win32 SDK，在安装Visual C++的同时，Win32 SDK也安装到你的计算机上了。尽管MFC提供了对Win32 API的比较完整的封装，但是，在某些情况下，我们更倾向于直接调用Win32 API，因为这有时候可以获得更高的效率，并且有着更大的自由度。而且，使用MFC编写的新风格的Windows应用程序的工作方式基本上与使用SDK编写的同一程序一样，它们往往有着很多的共同之处，只是使用MFC更加的方便，因为它隐藏了大量的复杂性。

前面提到过，面向对象的编程方式是当前最流行的程序设计方法，但是，Win32 API本身却是基于C语言的过程式编程的，SDK和MFC的最主要的不同之处也就是以C与C++之间的差别，使用MFC进行Windows应用程序设计需要面向对象的编程思想和方法，好在我们已经在前面这此进行了大量的铺垫。

### 第三节 使用SDK编写Windows应用程序

传统的DOS程序以main函数作为进入程序的初始入口点，在Windows应用程序中，main函数被WinMain函数取而代之，WinMain函数的原型如下：

```
int WINAPI WinMain (HINSTANCE hInstance, // 当前实例句柄
HINSTANCE hPrevInstance, // 前一实例句柄
LPSTR lpCmdLine, // 指向命令行参数的指针
int nCmdShow) // 窗口的显示状态
```

这里出现了一个新的名词“句柄”(handle)，所谓的句柄是一个标识对象的变量，或者是一个对操作系统资源的间接引用。

在上面的函数原型中，我们看到了一些“奇怪”的数据类型，如前面的HINSTANCE和LPSTR等，事实上，很多这样的数据类型只是一些基本数据类型的别名，表3.2列出了一些在Windows编程中常用的基本数据类型的别名，表3.3列出了常用的预定义句柄，它们的类型均为void\*，即一个32位指针。

表3. 2 Windows基本数据类型

Windows中所用的数据类型	对应的基本数据类型	说明
BOOL	int	布尔值
BSTR	unsigned short *	32位字符指针
BYTE	unsigned char	8位无符号整数
COLORREF	unsigned long	用作颜色值的32位值
DWORD	unsigned long	32位无符号整数，段地址和相关的偏移地址

LONG	long	32位带符号整数
LPARAM	long	作为参数传递给窗口过程或回调函数的32位值
LPCSTR	const char *	指向字符串常量的32位指针
LPSTR	char *	指向字符串的32位指针
LPCTSTR	const char * (注1)	指向可移植的Unicode和DBCS字符串常量的32位指针
LPTSTR	char *(注1)	指向可移植为Unicode和DBCS字符串的32位指针
LPVOID	void *	指向未定义类型的32位指针
LRESULT	long	来自窗口过程或回调函数的32位返回值
UINT	unsigned int	32位无符号整数
WNDPROC	long (__stdcall *) (void *, unsigned int, unsigned int, long)(注2)	指向窗口过程的32位指针
WORD	unsigned short	16位无符号整数
LPARAM	unsigned int	当作参数传递给窗口过程或回调函数的32位值

注1: 这是在DBCS版本下的情况, 在Unicode版本下LPCTSTR和LPTSTR将代表其它的数据类型。

注2: 事实上, WNDPROC被定义为LRESULT (CALLBACK\*)(HWND, UINT, WPARAM, LPARAM), 这个定义最终被编译器解释为long (\_\_stdcall\*)(void \*, unsigned int, unsigned int, long)。

表3. 3 Windows公用句柄类型

句柄类型	说明
HBITMAP	保存位图信息的内存域的句柄
HBRUSH	画刷句柄
HCTR	子窗口控件句柄
HCURSOR	鼠标光标句柄

HDC	设备描述表句柄
HDLG	对话框句柄
HFONT	字体句柄
HICON	图标句柄
HINSTANCE	应用程序的实例句柄
HMENU	菜单句柄
HMODULE	模块句柄
HPALETTE	颜色调色板句柄
HPEN	在设备上画图时用于指明线型的笔的句柄
HRGN	剪贴区域句柄
HTASK	独立于已执行任务的句柄
HWND	窗口句柄

查看Win32 SDK文档或者浏览Windows头文件(如windef.h、ctype.h以及winnt.h等)可以获得关于其它数据类型的定义，这些定义往往使用了#define和typedef等关键字。

这里解释什么是应用程序的一个实例(instance)。最简单的理解可以用下面的例子来说明：比如说已经在Windows中打开了一个“写字板”(可以在“开始”菜单中的“程序|附件”下面找到它的快捷方式)，现在你需要从另一篇文章里复制一部分内容到你正在写的这篇文章中，那么，你可以再打开一个“写字板”(注意写字板不是一个多文档应用程序，不能像在Word中那样打开多个不同的文件)，然后从该写字板中复制文件的内容到在前一个写字板内打开的文章中。这里，我们多次运行了同一个应用程序，在这个例子中，我们将所打开的两个写字板叫做该应用程序的两个实例。对于实例的更精确(当然也要比上面的例子要更难懂得多)的定义，在Win32 SDK中是这样给出的：实例就是类中一特定对象类型的一个实例化对象(instantiation)，如一个特定的进程或线程，在多任务操作系统中，一个实例指所加载的应用程序或动态链接库的一份拷贝。刚开始时我们也许看不懂这一定义，不过没有关系，慢慢的就理解了。

- 注意：

- 尽管在前面给出的WinMain函数的原型中包括了一个名为hPrevInstance的HINSTANCE类型的参数，按照其字面上的意义，它所传递的是应用程序的前一个实例的句柄，但是，在Win32平台下，该参数的值总是为NULL，而不管是否有当前应用程序的实例在运行。在过去的Windows 3.x环境下编程，我们常常使用下面的代码来检查应用程序是否已有一个实例在运行：

- if (!hPrevInstance)
- {
- // 在此添加没有应用程序实例在运行时的所需执行的代码。
- // 对于大多数应用程序，我们常在这里注册窗口类。
- }

然而，在Win32操作系统——Windows 95、Windows NT以及其后续版本中，上面的if条件体中的代码总会被执行，因为hPrevInstance总是为NULL，因此!hPrevInstance恒为真。

之所以这样，是因为在Win32环境下，每一个应用程序的实例都有自己独立的地址空间，因此，它们之间互相独立，互不干涉。但是，对于一些应用程序，只需要而且只应该有一个实例在运行。什么情况下会是这样呢？假设我们编写了一个应用程序，在默认情况下，该应用程序将在后台运行，通过按下程序所定义的某一个热键，应用程序将被激活。对于这样的应用程序，在同一时刻只应该有一个实例在运行。另外，像Windows NT下在任务管理器，在同一时刻也只能有一个实例在运行。

使用下面的技巧可以保证在同一时刻只有应用程序的一个实例：

- #include "windows.h"
- 
- #define VK\_X 0x58
- 
- int WINAPI WinMain (HINSTANCE hInstance,

- HINSTANCE hPrevInstance,
- LPSTR lpCmdLine,
- int nCmdShow)
- {
- if (!CreateMutex(NULL, TRUE, "No Previous Instance!"))
- {
- MessageBox(NULL, "创建Mutex失败!", "NoPrev", MB\_OK|MB\_SYSTEMMODAL);
- return FALSE;
- }
- if (GetLastError() == ERROR\_ALREADY\_EXISTS)
- {
- MessageBox(NULL, "已有NoPrev的一个实例在运行，当前实例将被终止!",
- "NoPrev", MB\_OK|MB\_SYSTEMMODAL);
- return FALSE;
- }
- if (!RegisterHotKey(NULL, 0x0001, MOD\_CONTROL|MOD\_SHIFT, VK\_X))
- {
- MessageBox(NULL, "注册热键Ctrl+Shift+X失败!",
- "NoPrev", MB\_OK|MB\_SYSTEMMODAL);
- return FALSE;
- }
- MessageBox(NULL, "NoPrev已启动!\n\n按下热键Ctrl+Shift+X将终止NoPrev.",
- "NoPrev", MB\_OK|MB\_SYSTEMMODAL);

- MSG msg;
- while (GetMessage(&msg,NULL,0,0))
- {
- switch (msg.message)
- {
- case WM\_HOTKEY:
- if (int(msg.wParam)==0x0001)
- if (MessageBox(NULL,"终止NoPrev?",
- "NoPrev",MB\_YESNO|MB\_SYSTEMMODAL)==IDYES)
- return TRUE;
- }
- }
- return TRUE;
- }

上面的代码是一个功能完整的Windows应用程序，其中用到了一些到目前为止我们还未讲述到的内容。程序定义了热键Ctrl+Shift+X，当按下该热键时将终止该程序。由于程序中没有包括任何窗口，因此这是唯一的一种正常终止应用程序的方法。当程序NoPrev正在后台运行时，如果用户按下了组合键Ctrl+Shift+X，Windows将向程序主线程的消息队列中发送一条称为WM\_HOTKEY的消息，当程序收到这条消息时，即弹出了消息框询问是否终止NoPrev。上面的说明将有助于你理解以上代码，但是我们目前对止并不做要求。这里，只需要注意下面的代码：

- if (!CreateMutex(NULL,TRUE,"No Previous Instance!"))
- {

- `MessageBox(NULL, "创建Mutex失败!", "NoPrev", MB_OK|MB_SYSTEMMODAL);`
- `return FALSE;`
- `}`
- `if (GetLastError()==ERROR_ALREADY_EXISTS)`
- `{`
- `MessageBox(NULL, "已有NoPrev的一个实例在运行, 当前实例将被终止!",`
- `"NoPrev", MB_OK|MB_SYSTEMMODAL);`
- `return FALSE;`
- `}`

在上面的代码中，我们先调用CreateMutex创建一个名为“ No Previous Instance ”的命名互斥对象(named mutex object)，如果该对象名已存在(注意这时函数CreateMutex仍返回真值TRUE)，则随后调用的GetLastError函数将返回ERROR\_ALREADY\_EXISTS，由此得知已有一个应用程序的实例正在运行。从而弹出消息框提醒用户，然后终止应用程序的当前实例。

在上面的WinMain函数原型中的另一个奇怪的标识符为WINAPI，这是一个在windef.h头文件中定义的宏，在当前版本Win32 SDK中，WINAPI被定义为FAR PASCAL，因此，使用FAR PASCAL同使用WINAPI具有同样的效果，但是，我们强烈建议你使用WINAPI来代替以前常用的FAR PASCAL，因为Microsoft不保证FAR PASCAL能够在将来的Windows版本中正常工作。在目前情况下，和FAR PASCAL等价的标识符还有CALLBACK（用在如窗口过程或对话框过程之类的回调函数前）和APIENTRY等。它们对编译器而言都是一回事，最终将被解释为\_\_stdcall。在Windows环境下编程，会遇到很多这样的情况，注意不要混淆它们。

一般情况下，我们应该在WinMain函数中完成下面的操作：

1. 注册窗口类；
2. 创建应用程序主窗口；



### 3. 进入应用程序消息循环。

接下来我们将依次讨论这些内容。

在Windows应用程序中，每一个窗口都必须从属于一个窗口类，窗口类定义了窗口所具有的属性，如它的样式、图标、鼠标指针、菜单名称及窗口过程名等。在注册窗口类前，我们先创建一个类型为WNDCLASS的结构，然后在该结构对象中填入窗口类的信息，最后将它传递给函数RegisterClass，整个过程如下面的代码所示：

```
WNDCLASS wc;

// 填充窗口类信息

wc.style=CS_HREDRAW|CS_VREDRAW;

wc.lpfnWndProc=WndProc;

wc.cbClsExtra=0;

wc.cbWndExtra=0;

wc.hInstance=hInstance;

wc.hIcon=LoadIcon(NULL, IDI_APPLICATION);

wc.hCursor=LoadCursor(NULL, IDC_ARROW);

wc.hbrBackground=GetStockObject(WHITE_BRUSH);

wc.lpszMenuName=NULL;

wc.lpszClassName="SdkDemo1";

// 注册窗口类

RegisterClass(&wc);
```

下面解释一下结构WNDCLASS中各成员的含义：

**style:** 指定窗口样式。该样式可以为一系列屏蔽位的按位或，在前面的例子中，CS\_HREDRAW表示当窗口用户区宽度改变时重绘整个窗口，而CS\_VREDRAW则表示当窗口用户区高度改变时重绘整个窗口。对于其它的窗口样式，请参阅SDK中关于WNDCLASS的联机文档。（顺便说一句，请注意该成员的大小写，它是小写的style，而

不是Style。)

**lpfnWndProc:** 指向窗口过程的指针。关于窗口过程我们将以后面的内容中讲述。在前面的例子中，我们使用名为WndProc的窗口过程。

**cbClsExtra:** 指定在窗口类结构之后分配的附加字节数。操作系统将这些字节初始化为0。

**cbWndExtra:** 指定在窗口实例之后分配的附加字节数。操作系统将这些字节初始化为0。如果应用程序使用WNDCLASS结构注册一个使用资源文件中的CLASS指令创建的对话框，那么cbWndExtra必须被设置为DLGWINDOWEXTRA。

**hInstance :** 标识该类的窗口过程所属的实例。

**hIcon :** 标识类图标。该成员必须为一个图标资源的句柄。如果该成员为NULL，则应用程序必须在用户最小化应用程序窗口时绘制图标。

**hCursor :** 标识类鼠标指针。该成员必须为一个光标资源的句柄，如果该成员为NULL，当鼠标移进应用程序窗口时应用程序必须显式指定指针形状。

**hbrBackground:** 标识类背景刷子。该成员可以作为一个用来绘制背景的画刷句柄，或者为标准系统颜色值之一。

**lpszMenuName:** 指向一个以NULL结尾的字符串的指针，该字符串指定了类菜单的资源名称。如果在资源名称为的菜单为一个整数所标识，则可以使用MAKEINTRESOURCE宏将其转换为一个字符串；如果该成员为NULL，则属于该类的窗口无默认菜单。

**lpszClassName:** 指向一个以NULL结尾的字符串或为一个原子。如果该参数为一个原子，那么它必须是一个使用GlobalAddAtom函数创建的全局原子；如果为一个字符串，该字符串器将成员窗口类名。

- 注意：
  - 这里多次提到窗口类这一名词，但是它和前面常说的C++类没有任何联系。窗口类只表示了窗口的类型，它完全不是面向对象意义上的类，因为它不支持面向对象技术中的继承及多态等。

在使用RegisterClass注册窗口类成功之后，即可以使用该窗口类创建并显示应用程序的窗口。这个过程如下面的代码所示：

```
// 创建应用程序主窗口

hWnd=CreateWindow ("SdkDemo1", // 窗口类名

"第一个Win32 SDK应用程序", // 窗口标题

WS_OVERLAPPEDWINDOW, // 窗口样式

CW_USEDEFAULT, // 初始化 x 坐标

CW_USEDEFAULT, // 初始化 y 坐标

CW_USEDEFAULT, // 初始化窗口宽度

CW_USEDEFAULT, // 初始化窗口高度

NULL, // 父窗口句柄

NULL, // 窗口菜单句柄

hInstance, // 程序实例句柄

NULL); // 创建参数

// 显示窗口

ShowWindow(hWnd,SW_SHOW);

// 更新主窗口客户区

UpdateWindow(hWnd);
```

由于上述代码均加上了详尽的注释，这里仅作一些简单的说明和强调。CreateWindow函数的原型是这样的：

```
HWND CreateWindow(LPCTSTR lpClassName, // 指向已注册的类名

LPCTSTR lpWindowName, // 指向窗口名称

DWORD dwStyle, // 窗口样式
```

```
int x, // 窗口的水平位置

int y, // 窗口的垂直位置

int nWidth, // 窗口宽度

int nHeight, // 窗口高度

HWND hWndParent, // 父窗口或所有者窗口句柄

HMENU hMenu, // 菜单句柄或子窗口标识符

HANDLE hInstance, // 应用程序实例句柄

LPVOID lpParam, // 指向窗口创建数据的指针

);
```

在前面的示例中，我们对x、y、nWidth和nHeight参数都传递了同一个值CW\_USEDEFAULT，表示使用系统默认的窗口位置和大小，该常量仅对于重叠式窗口(即在dwStyle样式中指定了WS\_OVERLAPPEDWINDOW，另一个常量WS\_TILEDWINDOW有着相同的值)有效。对于CreateWindows函数的其它内容，比如关于dwStyle参数所用常量的详细参考请自行参阅Win32 SDK中的文档。

- 注意：

- 尽管Windows 95是一个32位的操作系统，但是，其中也保留了很多16位的特征，比如说，在Windows 95环境下，系统最多只可以有16384个窗口句柄。而在Windows NT下则无此限。然而，事实上，对于一般的桌面个人机系统来说，我们几乎不可能超过这个限制。

创建窗口完成之后，ShowWindows显示该窗口，第二个参数SW\_SHOW表示在当前位置以当前大小激活并显示由第一个参数标识的窗口。然后，函数UpdateWindows向窗口发送一条WM\_PAINT消息，以通知窗口更新其客户区。需要注意的是，由UpdateWindows发送的WM\_PAINT消息将直接发送到窗口过程(在上面的例子中是WndProc函数)，而不是发送到进程的消息队列，因此，尽管这时应用程序的主消息循环尚未启动，但是窗口过程仍可接收到该WM\_PAINT消息并更新其用户区。

在完成上面的步骤之后，进入应用程序的主消息循环。一般情况下，主消息循环具有下面的格式：

```
while (GetMessage(&msg, NULL, 0, 0))
```

```
{  
TranslateMessage(&msg);  
DispatchMessage(&msg);  
}
```

主消息循环由对三个API函数的调用和一个while结构组成。其中 GetMessage从调用线程的消息队列中获取消息，并将消息放到由第一个参数指定的消息结构中。如果指定了第二个参数，则 GetMessage获取属于该参数指定的窗口句柄所标识的窗口的消息，如果该参数为 NULL，则 GetMessage获取属于调用线程及属于该线程的所有窗口的消息。最后两个参数指定了 GetMessage所获取消息的范围，如果两个参数均为0，则 GetMessage检索并获取所有可以得到的消息。

在上面的代码中，变量msg是一个类型为MSG的结构对象，该结构体的定义如下：

```
typedef struct tagMSG { // msg  
  
    HWND hwnd;  
  
    UINT message;  
  
    WPARAM wParam;  
  
    LPARAM lParam;  
  
    DWORD time;  
  
    POINT pt; } MSG;
```

下面解释各成员的含义：

hwnd：标识获得该消息的窗口进程的窗口句柄。

message：指定消息值。

wParam：其含义特定于具体的消息类型。

lParam：其含义特定于具体的消息类型。

time：指定消息发送时的时间。

pt：以屏幕坐标表示的消息发送时的鼠标指针的位置。

在while循环体中的TranslateMessage函数将虚拟按键消息翻译为字符消息，然后将消息发送到调用线程的消息队列，在下一次调用GetMessage函数或PeekMessage函数时，该字符消息将被获取。TranslateMessage函数将WM\_KEYDOWN和WM\_KEYUP虚拟按键组合翻译为WM\_CHAR和WM\_DEADCHAR，将WM\_SYSKEYDOWN和WM\_SYSKEYUP虚拟按键组合翻译为WM\_SYSCHAR和WM\_SYSREADCHAR。需要注意的一点是，仅当相应的虚拟按键组合能够被翻译为所对应的ASCII字符时，TranslateMessage才发送相应的WM\_CHAR消息。

如果一个字符消息被发送到调用线程的消息队列，则TranslateMessage返回非零值，否则返回零值。

- 注意：
- 与在Windows 95操作系统下不同，在Windows NT下，TranslateMessage对于功能键和光标箭头键也返回一个非零值。

然后，函数DispatchMessage将属于某一窗口的消息发送该窗口的窗口过程。这个窗口由MSG结构中的hwnd成员所标识的。函数的返回值为窗口过程的返回值，但是，我们一般不使用这个返回值。这里要注意的是，并不一定是所有属于某一个窗口的消息都发送给窗口的窗口过程，比如对于WM\_TIMER消息，如果其lParam参数不为NULL的话，由该参数所指定的函数将被调用，而不是窗口过程。

如果GetMessage从消息队列中得到一个WM\_QUIT消息，则它将返回一个假值，从而退出消息循环，WM\_QUIT消息的wParam参数指定了由PostQuitMessage函数给出的退出码，一般情况下，WinMain函数返回同一值。

下面我们来看一下程序主窗口的窗口过程WndProc。窗口过程名是可以由用户自行定义，然后在注册窗口类时在WNDCLASS结构中指定。但是，一般来说，程序都把窗口过程命令为WndProc来类似的名称，如MainWndProc等，并不是一定要这样做，但是这样明显的有利于阅读，因此也是我们推荐的做法。窗口过程具有如下的原型：

```
LRESULT WINAPI WndProc(HWND,UINT,WPARAM,LPARAM);
```

或

```
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
```

对于编译器而言，两种书写形式都是一样的，它们都等价于

```
long __stdcall WndProc(void *,unsigned int,unsigned int,long)
```

窗口过程使用了四个参数，在它被调用时(再强调一点，一般情况下，窗口过程是由操作系统调用，而不是由应用程序调用的，这就是我们为什么将它们称为回调函数的道理)，这四个参数对应于所发送消息结构的前四个成员。下面给出了一个窗口过程的例子：

```
// WndProc 主窗口过程
```

```
LRESULT WINAPI WndProc (HWND hWnd,
```

```
UINT msg,
```

```
WPARAM wParam,
```

```
LPARAM lParam)
```

```
{
```

```
HDC hdc;
```

```
RECT rc;
```

```
HPEN hPen,hPenOld;
```

```
HBRUSH hBrush,hBrushOld;
```

```
switch (msg)
```

```
{
```

```
case WM_PAINT:
```

```
hdc=GetDC(hWnd);
```

```
GetClientRect(hWnd,&rc);
```

```
hPen=CreatePen(PS_SOLID,0,RGB(0,0,0));
```

```
hBrush=CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,0));
```

```
hPenOld=SelectObject(hdc,hPen);
```

```
hBrushOld=SelectObject(hdc,hBrush);
```

```
Ellipse(hdc,rc.left,rc.top,rc.right,rc.bottom);
```

```
SelectObject(hdc,hPenOld);
```

```
SelectObject(hdc,hBrushOld);
```

```

ReleaseDC(hWnd, hdc);

break;

case WM_DESTROY:

PostQuitMessage(0);

break;

default:

break;

}

return DefWindowProc(hWnd, msg, wParam, lParam);

}

```

在该窗口过程中，我们处理了最基本两条消息。

第一条消息是WM\_PAINT，当窗口客户区的全部或一部分需要重绘时，系统向该窗口发送该消息。在前面的过程中我们已经提到过，在使用ShowWindow函数显示窗口之后，通常随即调用函数UpdateWindow，该函数直接向窗口过程发送一个WM\_PAINT消息，以通知窗口绘制其客户区。在该消息的处理函数中，我们先使用GetDC获得窗口的设备句柄，关于设备句柄本书后面将要专门涉及，这里我们只需知道它是用来调用各种绘图方法的。然后调用GetClientRect获得当前窗口的客户区矩形。接着调用CreatePen创建一个黑色画笔，调用CreateHatchBrush创建一个45度交叉线的填充画刷，并且SelectObject函数将它们选入设备描述表中，原有的画笔和画刷被保存到hPenOld和hBrushOld中，以便以后恢复。完成以上步骤之后，调用Ellipse函数以当前客户区大小绘制一个椭圆。最后，再一次调用SelectObject函数恢复原有的画笔和画刷，并调用ReleaseDC释放设备描述表句柄。在这个消息的处理代码中，我们涉及到了一些新的概念、数据类型和API函数，然而本章并不着意于讲述这些内容，读者也不必深究它们，这些代码只是为了完整该示例程序才使用的。对于窗口来说，除了客户区以外的其它内容将由系统进行重绘，这些内容包括窗口标题条、边框、菜单条、工具条以及其它控件，如果包含了它们的话。这种重绘往往发生在覆盖于窗口上方的其它窗口被移走，或者是窗口被移动或改变大小时。因此，对于大多数窗口过程来说，WM\_PAINT消息是必须处理的。

另一个对于绝大多数窗口过程都必须处理的消息是WM\_DESTROY，当窗



口被撤消时(比如用户从窗口的系统菜单中选择了“关闭”，或者单击了右边的小叉，对于这些事件，Windows的默认处理是调用DestroyWindow函数撤销相应的窗口)，将会接收到该消息。由于本程序仅在一个窗口，因此在这种情况下应该终止应用程序的执行，因此我们调用了PostQuitMessage函数，该函数向线程的消息队列中放入一个WM\_QUIT消息，传递给PostQuitMessage函数的参数将成为WM\_QUIT消息的wParam参数，在上面的例子中，该值为0。

对于其它情况，在上面的示例程序中我们没有必要进行处理，Windows专门为此提供了一个默认的窗口过程，称为DefWindowProc，我们只需要以WndProc的参数原封不动的调用默认窗口过程DefWindowProc，并将其返回值作为WndProc的返回值即可。

将上面讲述的所有内容综合起来，我们就已经使用Win32 SDK完成了一个功能简单，但是结构完整的Win32应用程序了。对于使用Win32 SDK编写的实用的Win32应用程序，它们的结构与此相比要复杂得多，在这些情况下，应用程序也许不仅仅包括一个窗口，而对应的窗口过程中的switch结构一般也会是一个异常膨胀的嵌套式switch结构。如此庞大的消息处理过程大大增加了程序调试和维护的难度，使用MFC则有可能在很多程度上减轻这种负担，这便是MFC为广大程序员所乐于接受，以至今天成为实际上的工业标准的原因。但是，不管它如何复杂，归根到底，一般情况下，它仍然具有和我们的这个功能简单的Win32应用程序一样或类似的结构。为了读者阅读和分析方便，我们把这个程序的完整代码给出如下：

```
#include <windows.h>

// 函数原型

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);

LRESULT WINAPI WndProc(HWND, UINT, WPARAM, LPARAM);

// WinMain 函数

int WINAPI WinMain (HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow)
{
```

```
HWND hWnd; // 主窗口句柄

MSG msg; // 窗口消息

WNDCLASS wc; // 窗口类

if (!hPrevInstance)

{

    // 填充窗口类信息

    wc.style=CS_HREDRAW|CS_VREDRAW;

    wc.lpfnWndProc=WndProc;

    wc.cbClsExtra=0;

    wc.cbWndExtra=0;

    wc.hInstance=hInstance;

    wc.hIcon=LoadIcon(NULL, IDI_APPLICATION);

    wc.hCursor=LoadCursor(NULL, IDC_ARROW);

    wc.hbrBackground=GetStockObject(WHITE_BRUSH);

    wc.lpszMenuName=NULL;

    wc.lpszClassName="SdkDemo1";

    // 注册窗口类

    RegisterClass(&wc);

}

// 创建应用程序主窗口

hWnd=CreateWindow ("SdkDemo1", // 窗口类名

"第一个Win32 SDK应用程序", // 窗口标题

WS_OVERLAPPEDWINDOW, // 窗口样式

CW_USEDEFAULT, // 初始化 x 坐标

CW_USEDEFAULT, // 初始化 y 坐标

CW_USEDEFAULT, // 初始化窗口宽度
```

```

CW_USEDEFAULT, // 初始化窗口高度

NULL, // 父窗口句柄

NULL, // 窗口菜单句柄

hInstance, // 程序实例句柄

NULL); // 创建参数

// 显示窗口

ShowWindow(hWnd, SW_SHOW);

// 更新主窗口客户区

UpdateWindow(hWnd);

// 开始消息循环

while (GetMessage(&msg, NULL, 0, 0))

{

    TranslateMessage(&msg);

    DispatchMessage(&msg);

}

return msg.wParam;

}

// WndProc 主窗口过程

LRESULT WINAPI WndProc (HWND hWnd,

UINT msg,

WPARAM wParam,

LPARAM lParam)

{

    HDC hdc;

    RECT rc;

    HPEN hPen, hPenOld;

```

```

HBRUSH hBrush,hBrushOld;

switch (msg)
{
case WM_PAINT:
hdc=GetDC(hWnd);

GetClientRect(hWnd,&rc);

hPen=CreatePen(PS_SOLID,0,RGB(0,0,0));

hBrush=CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,0));

hPenOld=SelectObject(hdc,hPen);

hBrushOld=SelectObject(hdc,hBrush);

Ellipse(hdc,rc.left,rc.top,rc.right,rc.bottom);

SelectObject(hdc,hPenOld);

SelectObject(hdc,hBrushOld);

ReleaseDC(hWnd,hdc);

break;

case WM_DESTROY:

PostQuitMessage(0);

break;

default:

break;

}

return DefWindowProc(hWnd,msg,wParam,lParam);

}

```

该示例代码中的所有内容都已在前面做了完整的讲解，这里我们简单的说一下如何在Microsoft Developer Studio中编译该示例程序。请按下面的步骤进行：

1. 选择File菜单下的New命令，新建一个Win32 Application工程，

这里我们假设对该工程命名为SdkDemo1，而事实上这完全取决于你的意愿。这个过程已经在本书的第一章中作为介绍，这里就不再重复说明了。

2. 选择Project菜单下的Add To Project|New...命令，向工程中添加一个C++ Source File (C++源文件)，可以将该文件命名为winmain.cpp，不需要键入扩展名，Microsoft Developer Studio在创建文件时会自动加上.cpp的后缀名。这个过程也已经在第一章中作过介绍。阅读过该章内容的读者不应感到陌生。然后在Wordspace窗口的FileView中双击文件名winmain.cpp（在依赖于你在前面过程中的设定），输入下面的源代码即可。

如果已将源代码输入为C++源文件(以.cpp为后缀名的文件)，则可以使用Project|Add To Project|Files...将其添加到工程中。

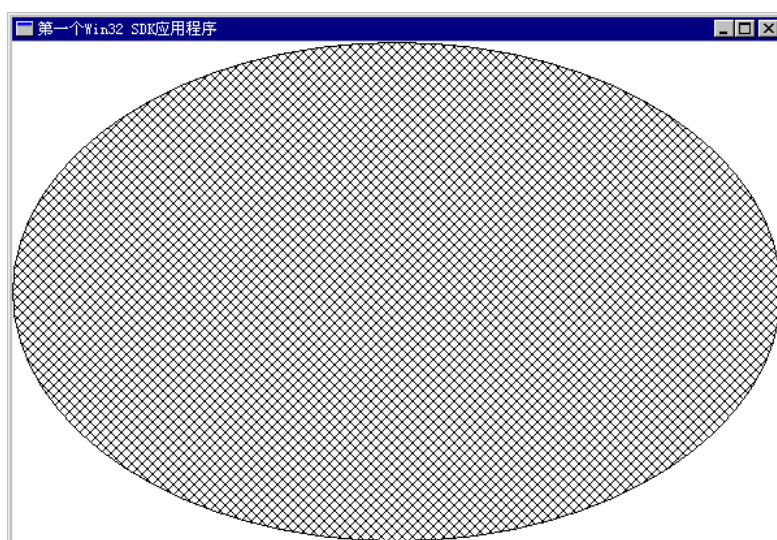



图3.2 示例程序SdkDemo1的运行结果

3. 单击Build菜单下的Build SdkDemo1.exe或Build All或按下快捷键F7（如果未对该快捷键做过自定义操作的话）或单击Build或Build Minibar工具条上的按钮，编译并创建可执行文件SdkDemo1.exe，运行该可执行文件(从Developer Studio中或资源管理器均可)，将得到如图3.2所示的结果。

前面已经不只一次说到过，使用这种方式编写的应用程序使用调试和维护的难度很大。这个问题是使用直接使用SDK编程的固有总是。但是，我们还是有办法可以使得该程序的结构更紧凑和更集中一些，从而改善代码的可读性，也使得它更接近于使用SDK编写的真正的Win32应用程序。

通过分析应用程序，我们发现，在上面的程序代码中，WinMain函数的代码显得有些过分臃肿，解决总是的办法就是将这些代码分离为单个的函数，这样，我们就可以得到更实用的基本SDK应用程序框架，当然，相对于MFC所提供的应用程序框架来说，我们的这个应用程序框架几乎不值一提，但是，它的确是要比前面的示例程序好多了。

经过修改的代码如下：

```
#include <windows.h>

// 函数原型

int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int);

LRESULT WINAPI WndProc(HWND,UINT,WPARAM,LPARAM);

BOOL InitApplication(HINSTANCE);

BOOL InitInstance(HINSTANCE,int);

// WinMain 函数

int WINAPI WinMain (HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow)
{
    if (!hPrevInstance)
    if (!InitApplication(hInstance))
    return FALSE;

    if (!InitInstance(hInstance,SW_SHOW))
    return FALSE;

    MSG msg; // 窗口消息

    // 开始消息循环

    while (GetMessage(&msg,NULL,0,0))
    {
```

```

TranslateMessage(&msg);
DispatchMessage(&msg);
}

return msg.wParam;
}

// WndProc 主窗口过程
LRESULT WINAPI WndProc (HWND hWnd,
UINT msg,
WPARAM wParam,
LPARAM lParam)
{
HDC hdc;
RECT rc;
HPEN hPen,hPenOld;
HBRUSH hBrush,hBrushOld;

switch (msg)
{
case WM_PAINT:
hdc=GetDC(hWnd);
GetClientRect(hWnd,&rc);
hPen=CreatePen(PS_SOLID,0,RGB(0,0,0));
hBrush=CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,0));
hPenOld=SelectObject(hdc,hPen);
hBrushOld=SelectObject(hdc,hBrush);
Ellipse(hdc,rc.left,rc.top,rc.right,rc.bottom);
SelectObject(hdc,hPenOld);

```

```

SelectObject(hdc,hBrushOld);

ReleaseDC(hWnd,hdc);

break;

case WM_DESTROY:

PostQuitMessage(0);

break;

default:

break;

}

return DefWindowProc(hWnd,msg,wParam,lParam);

}

BOOL InitApplication(HINSTANCE hInstance)

{

WNDCLASS wc; // 窗口类

// 填充窗口类信息

wc.style=CS_HREDRAW|CS_VREDRAW;

wc.lpfnWndProc=WndProc;

wc.cbClsExtra=0;

wc.cbWndExtra=0;

wc.hInstance=hInstance;

wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);

wc.hCursor=LoadCursor(NULL,IDC_ARROW);

wc.hbrBackground=GetStockObject(WHITE_BRUSH);

wc.lpszMenuName=NULL;

wc.lpszClassName="SdkDemo2";

// 注册窗口类

```



```

return RegisterClass(&wc);

}

BOOL InitInstance(HINSTANCE hInstance,int nCmdShow)

{
HWND hWnd; // 主窗口句柄

// 创建应用程序主窗口

hWnd=CreateWindow ("SdkDemo2", // 窗口类名

"经过修改的第一个Win32 SDK应用程序", // 窗口标题

WS_OVERLAPPEDWINDOW, // 窗口样式

CW_USEDEFAULT, // 初始化 x 坐标

CW_USEDEFAULT, // 初始化 y 坐标

CW_USEDEFAULT, // 初始化窗口宽度

CW_USEDEFAULT, // 初始化窗口高度

NULL, // 父窗口句柄

NULL, // 窗口菜单句柄

hInstance, // 程序实例句柄

NULL); // 创建参数

if (!hWnd)

return FALSE;

// 显示窗口

ShowWindow(hWnd,SW_SHOW);

// 更新主窗口客户区

UpdateWindow(hWnd);

return TRUE;

}

```

由于上面的代码只是将前面的代码的结构作了一下调整，并没有引入

新的API函数和其它编程内容，因此为了节省篇幅，我们这里对该代码不再进行讲解，至于其与SdkDemo1的代码相比的优越性，由读者自己将两段代码对比后得出。

## 第四节 32位编程的特点

本节假定用户是刚接触32位Windows编程的新手，那么，有必要将一些相关的概念术语弄清楚，同时，也要把Windows 95、Windows NT和16位的Windows 3.x相区别开来。这些最重要的概念包括进程和线程的管理以及新的32位平坦内存模式。

在介绍32位内存管理之前，我们有必要介绍一下进程和线程这两个术语。

进程是装入内存中正在执行的应用程序，进程包括私有的虚拟地址空间、代码、数据及其它操作系统资源，如文件、管道以及对该进程可见的同步对象等。进程包括了一个或多个在进程上下文内运行的线程。

线程是操作系统分配CPU时间的基本实体。线程可以执行应用程序代码的任何部分，包括当前正在被其它线程执行的那些。同一进程的所有线程共享同样的虚拟地址空间、全局变量和操作系统资源。

在一个应用程序中，可以包括一个或多个进程，每个进程由一个或多个线程构成。

线程通过“休眠”(sleeping，暂停所有执行并等待)的方法，来做到与进程中的其它线程所同步。在线程休眠前，必须告诉Windows，该线程将等待某一事件的发生。当该事件发生时，Windows发给线程一个唤醒调用，线程继续执行。也就是说，线程与事件一起被同步，除此之外，也可以由特殊的同步对象来进行线程的同步。这些同步对象包括：

- **互斥** 不受控制的或随意的线程访问在多线程应用程序中可能会引起很大的问题。这里所说的互斥是一小须代码，它时刻采取对共享数据的独占控制以执行代码。互斥常被应用于多进程的同步数据存取。
- **信号量** 信号量与互斥相似，但是互斥只允许在同一时刻一个线程访问它的数据，而信号量允许多个线程在同一时刻访问它的数据。Win32不知道哪一个线程拥有信号量，它只保证信号量使用的资源量。

- 临界区 临界区对象也和互斥相似，但它仅被属于单个进程的线程使用。临界区对象提供非常有效的同步模式，同互斥一样，每次在同一时间内只有一个线程可以访问临界区对象。
- 事件 事件对象用于许多实例中去通知休眠的线程所等待的事件已经发生，事件告诉线程何时去执行某一个给定的任务，并可以使多线程流平滑。

将所有的这些同步对象应用于控制数据访问使得线程同步成为可能，否则，如果一个线程改变了另一个线程正在读的数据，将有可能导致很大的麻烦。

在Win32环境下，每个运行的在进程内的线程还可以为它自己的特定线程数据分配内存，通过Win32提供的线程本地存储(TLS)API，应用程序可以建立动态的特定线程数据，在运行时这些数据联系在一起。

本书将在专门的章节中讨论线程和进程的问题。

下面我们来看在32位应用程序地址空间中的内存分配和内存管理。

常见的内存分配可以划分为两类：帧分配(frame allocation)和堆分配(heap allocation)。两者的主要区别在于帧分配通常和实际的内存块打交道，而堆分配在一般情况下则使用指向内存块的指针，并且，帧对象在超过其作用域时会被自动的删除，而程序员必须显式的删除在堆上分配的对象。

在帧上分配内存的这种说法来源于“堆栈帧”(stack frame)这个名词，堆栈帧在每当函数被调用时创建，它是一块用来暂时保存函数参数以及在函数中定义的局部变量的内存区域。帧变量通常被称作自动变量，这是因为编译器自动为它们分配所需的内存。

帧分配有两个主要特征，首先，当我们定义一个局部变量是，编译器将在堆栈帧上分配足够的空间来保存整个变量，对于很大的数组和其它数据结构也是这样；其次，当超过其作用域时，帧变量将被自动的删除。下面举一个帧分配的例子：

```
int Func(int Argu1,int Argu2) // 编译器将在堆栈帧上为函数参数变量分配空间
{
    // 在堆栈上创建局部对象

    char szDatum[256][256];
```

...

```
// 超过作用域时将自动删除在堆栈上分配的对象  
}
```

对于局部函数变量，其作用域转变在函数退出时发生，但如果使用了嵌套的花括号，则帧变量的作用域将有可能比函数作用域小。自动删除这些帧变量非常之重要。对于简单的基本数据类型(如整型或字节变量)、数组或数据结构，自动删除只是简单的回收被这些这是所占用的内存。由于这些变量已超出其作用域，它们将再也不可以被访问。对于C++对象，自动删除的过程要稍稍复杂一些。当一个对象被定义为一个帧变量时，其构造函数在定义对象变量时被自动的调用，当对象超出其作用域时，在对象所占用的内存被释放前，其析构函数先被自动的调用。这种对构造函数和析构函数的调用看起来非常的简便，但我们必须对它们倍加小心，尤其是对析构函数，不正确的构造和释放对象将可能对内存管理带来严重的问题。在本书的第二章中我们曾经历过其中的一种——指针挂起。

在帧上分配对象的最大的优越性在于这些对象将会被自动的删除，也就是说，当你在帧上分配对象之后，不必担心它们会导致内存漏损(memory leak)。但是，帧分配也有其不方便之处，首先，在帧上分配的变量不可以超出其作用域，其中，帧空间往往是有限的，因此，在很多情况下，我们更倾向于使用下面接着要讲述的堆分配来代替这里所讲述的帧分配来为那些庞大的数据结构或对象分配内存。

堆是为程序所保留的用于内存分配的区域，它与程序代码和堆栈相隔离。在通常情况下，C程序使用函数malloc和free来分配和释放堆内存。调试版本(Debug version)的MFC提供了改良版本的C++内建运算符new和delete用于在堆内存中分配和释放对象。

使用new和delete代替malloc和free可以从类库提供的增强的内存管理调试支持中得到好处，这在检测内存漏损时非常之有用。而当你使用MFC的发行版本(Release version)来创建应用程序时，MFC的发行版本并没有使用这种改良版本的new和delete操作符，取而代之的是一种更为有效的分配和释放内存的方法。

与帧分配不同，在堆上可分配的对象所占用的内存的总量只受限于系统可有的所有虚拟内存空间。

以下的示例代码对比了上面讨论的内存分配方法在为数组、数据结构和对象分配内存时的用法：

## 使用栈分配为数组分配内存：

```
{  
const int BUFF_SIZE = 128;  
// 在栈上分配数组空间  
char myCharArray[BUFF_SIZE];  
int myIntArray[BUFF_SIZE];  
// 所分配的空间在超出作用域时自动回收  
}
```

## 使用堆分配为数组分配内存：

```
const int BUFF_SIZE = 128;  
// 在堆上分配数组空间  
char* myCharArray = new char[BUFF_SIZE];  
int* myIntArray = new int[BUFF_SIZE];  
...  
delete [] myCharArray;  
delete [] myIntArray;
```

## 使用栈分配为结构分配内存：

```
struct MyStructType { int topScore;};  
void SomeFunc(void)  
{  
// 栈分配  
MyStructType myStruct;  
// 使用该结构  
myStruct.topScore = 297;  
// 在超出作用域时结构所占用的内存被自动回收  
}
```

## 使用堆分配为结构分配内存：

```
// 堆分配

MyStructType* myStruct = new MyStructType;

// 通过指针使用该结构

myStruct->topScore = 297;

delete myStruct;
```

## 使用栈分配为对象分配内存：

```
{

CMyClass myClass; // 构造函数被自动调用

myClass.SomeMemberFunction(); // 使用该对象

}
```

## 使用堆分配为对象分配内存：

```
// 自动调用构造函数

CMyClass *myClass=new CMyClass;

myClass->SomeMemberFunction(); // 使用该对象

delete myClass; // 在使用delete的过程中调用析构函数
```

### • 注意：

- 一定要记住一个事实，在堆上分配的内存一定要记得释放，对于使用运算符new分配的内存，应当使用delete运算符来释放；而使用malloc函数分配的内存应当使用free函数来释放。不应当对同一内存块交叉使用运算符new、delete和函数malloc、free（即使用delete运算符释放由malloc函数分配的内存，或使用free函数释放由new运算符根本的内存），否则在MFC的调试版本下将会导致内存冲突。

对固定大小的内存块，使用运算符new和delete要比使用标准C库函数malloc和free方便。但有时候我们需要使用可变大小的内存块，这时，我们必须使用标准的C库函数malloc、realloc和free。下面的示例代码创建了一个可变大小的数组：

```

#include <malloc.h>

#include <stdio.h>

#define UPPER_BOUND 128

void main()

{

int *iArray=(int *)malloc(sizeof(int));

for (int i=0;i<UPPER_BOUND;i++)

{

iArray=(int *)realloc(iArray,(i+1)*sizeof(int));

printf(" %08x", (int)iArray);

iArray[i]=i+1;

}

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

{

printf("%5d", iArray[i]);

}

free(iArray);

return;

}

```

观察下面的运行结果：

```

00410770 00410780 00410790 004107a0 004107a0 004107c0 004107e0 00410800
00410800 00410830 00410860 00410890 00410890 004108d0 00410910 00410950
00410950 004109a0 004109f0 00410a40 00410a40 00410aa0 00410b00 00410b60
00410b60 00410bd0 00410c40 00410cb0 00410cb0 00410d30 00410db0 00410e30
00410e30 00410ec0 00410f50 00410760 00410760 00410800 004108a0 00410940
00410940 004109f0 00410aa0 00410b50 00410b50 00410c10 00410cd0 00410d90

```

00410d90 00410e60 00410f30 00410760 00410760 00410840 00410920 00410a00  
00410a00 00410af0 00410be0 00410cd0 00410cd0 00410dd0 00410ed0 00410760  
00410760 00410870 00410980 00410a90 00410a90 00410bb0 00410cd0 00410df0  
00410df0 00410760 00410890 004109c0 004109c0 00410b00 00410c40 00410d80  
00410d80 00410760 004108b0 00410a00 00410a00 00410b60 00410cc0 00410e20  
00410e20 00410760 004108d0 00410a40 00410a40 00410bc0 00410d40 00410760  
00410760 004108f0 00410a80 00410c10 00410c10 00410db0 00410760 00410900  
00410900 00410ab0 00410c60 00410e10 00410e10 00410760 00410920 00410ae0  
00410ae0 00410cb0 00410760 00410930 00351d98 00351d98 00351d98 00351d98  
00351d98 00351d98 00351d98 00351d98 00351d98 00351d98 00351d98 00351d98  
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32  
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48  
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64  
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80  
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96  
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112  
113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128

上面的结果说明两个问题：一是在使用realloc改变所分配内存单元的大小时，realloc可能会移动该内存块到新的位置。二是在realloc改变和移动内存块时，该内存块中的数据也同时会被保留。所以上面的模式可以满足一些特殊场合的需要。但是，请记住一点，随着内存块的增大，realloc的运行效率会极大的下降，当上面的，因此，除非必要，我们不要使用这种方式，虽然它灵活方便，但这往往是以牺牲效率为代价的。

最后重新强调一点，即不要使用realloc改变由new运算符分配的内存块的大小。前面已经说过，这将会在MFC的调试版本中导致内存冲突。

下面我们简单的介绍Win32内存管理模式。



在Microsoft Win32应用程序编程接口中，每一个进程都有自己多达4GB的虚拟地址空间。内存中低位的2GB（从0x00到0x7FFFFFFF）可以为用户所用，高位的2GB（从0x80000000到0xFFFFFFFF）为内核所保留。进程所使用的虚拟地址并不代码对象在内存中实际的物理地址。事实上，内核为每一个进程维护了一个页映射，页映射是一个用于将虚拟地址转换为对应的物理地址的内部数据结构。

每一个进程的虚拟地址空间都要比所有进程可用的物理内存RAM（随机存取存储器）的总合大得多。为了增加物理存储的大小，内核使用磁盘作为额外的存储空间。对于所有正在执行的进程来说，总的存储空间的量是物理内存RAM和磁盘上可以为页面文件所用的自由空间的总合，这里页面文件指用来增加物理存储空间的磁盘文件。每个进程物理存储空间和虚拟地址（也称为逻辑地址）空间以页的形式来组织，页是一种内存单元，其大小依赖于宿主计算机的类型。对于x86计算机来说，宿主页大小为4KB，但我们不能假定对所有运行Windows操作系统的计算机，其页大小均为4KB。

为了使内存管理具有最大的灵活性，内核可以将物理内存中的页移入或移出磁盘上的页面文件。当一个页被移入物理内存时，内核更新受到影响的进程的页映射。将内核需要物理内存空间时，它将物理内存中最近最少使用的页移入页面文件。对于应用程序来说，内核对物理内存的管理是完全透明的，应用程序只对它自己的虚拟地址空间进行操作。

在进程虚拟地址空间中的页可以具在表所列的状态之一：

表3. 4 虚拟地址空间中页的不同状态

状态	说明
空闲(Free)	空闲页是当前不可用，但可以被占用或保留的页。
保留 (Reserved)	保留页是进程的虚拟地址空间中为将来使用所保留的页。进程不可以存取保留页，并且也没有为保留页分配物理存储。保留页保留虚拟地址中的一段以使得它们不可以被随后的其它分配操作(如malloc和LocalAlloc之类的函数等)所使用。一个进程可以使用VirtualAlloc函数在其地址空间中保留页面，然后再占用这些保留页。最后使用VirtualFree函数释放它们。
占用 (Committed)	已被占用的页是那些已分配了物理存储(在内存中或磁盘上)的页。占用页可以被禁止进行存取，或允许只读存取，或允许读写存取。进程可以使用VirtualAlloc函数分配占用页。GlobalAlloc和LocalAlloc函数分配允许读

	写存取占用页。占用页可以使用VirtualFree函数进行释放，函数VirtualFree释放页的存储空间，并将其状态改变为保留。
--	---

进程可以使用函数GlobalAlloc和LocalAlloc来分配内存。在Win32 API的32位线性环境中，本地堆和全局堆并没有区别，因此，使用这两个函数来分配内存对象也没有任何区别。

由GlobalAlloc和LocalAlloc函数分配的内存对象位于私有的占用页中，这些页允许进行读写存取。私有内存不可以为其它进程所访问。与在Windows 3.x中不同，使用带有GMEM\_DDESHARE标志的GlobalAlloc函数分配的内存事实上并没有被全局共享。保留该标志位仅是为了向前兼容和为一些应用程序增强动态数据交换(DDE, dynamic data exchange)的性能而使用。应用程序如果因其它目的需要共享内存，那么必须使用文件映射对象。多个进程可以通过映射同一个文件映射对象的视来提供命名共享内存。我们在这里将不讨论文件映射和共享内存的问题。

通过使用函数GlobalAlloc和LocalAlloc，可以分配能够表示为32位的任意大小的内存块，所受的唯一限制是可用的物理内存，包括在磁盘上的页面文件中的存储空间。这些函数，和其它操作全局和本局内存对象的全局和本地函数一起被包含在Win32 API中，以和Windows的16位版本相兼容。但是，从16位分段内存模式到32位虚拟内存模式的转变将使得一些函数和一些选项变得不必要甚至没有意义。比如说，现在不再有近指针和远指针的区别，因为无论在本地还是在全局进行分配都将返回32位虚拟地址。

函数GlobalAlloc和LocalAlloc都可以分配固定或可移动的内存对象。可移动对象也可以被标记为可丢弃的(discardable)。在早期的Windows版本中，可移动的内存对象对于内存管理非常之重要，它们允许系统在必要时压缩堆以为其它内存分配提供可用空间。通过使用虚拟内存，系统能够通过移动物理内存页来管理内存，而不影响使用这些页的进程的虚拟地址。当系统移动一个物理内存页时，它简单的将进程的虚拟页映射到新的物理页的位置。可移动内存存在分配可丢弃内存仍然有用。当系统需要额外的物理存储时，它使用一种称作“最近最少使用”的算法来释放非锁定的可丢弃内存。可丢弃内存可以用于那些不是经常需要和易于重新创建的数据。

当分配固定内存对象时，GlobalAlloc和LocalAlloc返回32位指针，调用线程可以立即使用该指针来进行内存存取。对于可移动内存，返回值为一个句柄。为了得到一个指向可移动内存的指针，调用线程可

以使用GlobalLock和LocalLock函数。这些函数锁定内存使得它不能够被移动或丢弃，除非使用函数GlobalReAlloc或LocalReAlloc对内存对象进行重新分配。已锁定内存对象的内存块保持锁定状态，直至锁定计数减到0，这时该内存块可以被移动或丢弃。

由GlobalAlloc和LocalAlloc所分配的内存的实际大小可能大于所要求的大小。为了得到已分配的实际内存数，可以使用函数GlobalSize和LocalSize。如果总分配量大于所要求的量，进程则可以使用所有的这些量。]

函数GlobalReAlloc和LocalReAlloc以字节为单位改变由GlobalAlloc和LocalAlloc函数分配内存对象的大小或其属性。内存对象的大小可以增大，也可以减小。

函数GlobalFree和LocalFree用于释放由GlobalAlloc、LocalAlloc、GlobalReAlloc或LocalReAlloc分配的内存。

其它的全局和本地函数包括GlobalDiscard、LocalDiscard、GlobalFlags、LocalFlags、GlobalHandle和LocalHandle。GlobalDiscard和LocalDiscard用于丢弃指定的可丢弃内存对象，但不使其句柄无效。该句柄可能通过函数GlobalReAlloc或LocalReAlloc与新分配的内存块相关联。函数GlobalFlags或LocalFlags返回关于指定内存对象的信息。这些住处包括对象的锁定计数以及对象是否可丢弃或是否已被丢弃。函数GlobalHandle或LocalHandle返回与指定指针相关联的内存对象的句柄。

Win32进程可以完全的使用标准的C库函数malloc、free等来操作内存。在Windows的早期版本中使用这些函数，将可能带来问题隐患，但是使用Win32 API的应用程序中则不会。举例来说，使用malloc分配固定指针将不能使用可移动内存的优点。由于系统可以通过移动物理内存页来自由的管理内存，而不影响虚拟地址，因此内存管理将不再成为问题。类似的，远指针和近指针之间不再有差别。因此，除非你希望使用可丢弃内存，否则完全可以将标准的C库函数用于内存管理。

Win32 API提供了一系列的虚拟内存函数来操作或决定虚拟地址空间中的页的状态。许多应用程序使用标准的分配函数GlobalAlloc、LocalAlloc、malloc等就可以满足其需要。然而，虚拟内存函数提供了一些这些标准分配函数所不具有的功能，它们可以进行下面的这些操作：

- 保留进程虚拟地址空间中的一段 保留地址空间并不为它们分配物理存储，而只是防止其它分配操作使用这段空间。它并不影响其它进程的虚拟地址空间。保留页防止了对物理存储的不必要的浪费，然而它允许进程为可能增长的动态数据结构保留一段地址空间，进程可以在需要的时候为这些空间分配物理存储。
- 占用进程虚拟地址空间中的保留页的一部分，以使得物理存储(无论是RAM还是磁盘空间)只对正在进行分配的进程可用。
- 指定允许读写存取、只读存取或不允许存取的占用页区域。这和标准的分配函数总是分配允许读写存取的页不同。
- 释放一段保留页，使得调用线程在随后的分配操作中可以使用这段虚拟地址。
- 取消对一段页的占用，释放它们的物理存储，使它们可以其它进程在随后的分配中使用。
- 在物理内存RAM中锁定一个或多个占用页，以免系统将这些页交换到页面文件中。
- 获得关于调用线程或指定线程的虚拟地址空间中一段页的信息。
- 改变调用线程或指定线程的虚拟地址空间中指定占用页段的存取保护。

虚拟内存函数对内存页进行操作。函数使用当前计算机的页大小来对指定的大小和地址进行舍入。

可以使用函数GetSystemInfo来获得当前计算机的页大小。

函数VirtualAlloc完成以下操作：

- 保留一个或多个自由页。
- 占用一个或多个保留页。
- 保留并占用一个或多个自由页。

你可以指定所保留或占用的页的起始地址，或者让系统来决定。函数将指定的地址舍入到合适的页边界。保留页是不可访问的，但占用页可以使用标志位PAGE\_READWRITE、PAGE\_READONLY和PAGE\_NOACCESS来

分配。当页被占用时，从页面文件中分配存储空间，每个页仅在第一次试图对其进行读写操作时被初始化并加载到物理内存中。可以用一般的指针引用来访问由VirtualAlloc函数占用的页。

函数VirtualFree完成下面的操作：

- 解除对一个或多个页的占用，改变其状态为保留。解除对页的占用释放与之相关的物理存储，使其为其它进程可用。任何占用页块都可以被解除占用。
- 释放一个或多个保留页块，改变其状态为自由。释放页块使这段保留空间可以为进程分配空间使用。保留页只能通过释放由函数VirtualAlloc最初保留的整个块来释放。
- 同时解除对一个或多个占用页的占用并释放它们，将其状态改变为自由。指定的块必须包括由VirtualAlloc最初保留的整个块，而且这些页的当前状态必须为占用。

函数VirtualLock允许进程将一个或多个占用页锁定在物理内存RAM中，防止系统将它们交换到页面文件中。这保证了一些要求苛刻的数据可以不通过磁盘访问来存取。将一个页锁定入内存是很危险的，因为它限制了系统管理内存的能力。由于可能会将可执行代码交换到页面文件中，可执行程序使用VirtualLock将有可能降低系统性能。函数VirtualUnlock解除VirtualLock对内存的锁定。

函数VirtualQuery和VirtualQueryEx返回关于以进程地址空间中某一指定地址开始的一段连续内存区域的信息。VirtualQuery返回关于调用线程内存的信息。VirtualQueryEx返回指定进程内存的信息，这通常用来支持调试程序，这些程序常常需要知道关于被调试进程的信息。页区域以相对于指定地址最接近的页边界为界。一般来说，它通过具有下述属性的后续页来进行扩展：

所有页具有相同的状态，或为占用，或为保留，或为自由。

如果初始页不为自由，区域中的所有页都属于通过调用VirtualAlloc保留的同一个最初页分配。

所有页的存取保护相同，或为PAGE\_READONLY，或为PAGE\_READWRITE，或为PAGE\_NOACCESS。

函数VirtualProtect允许进程修改进程地址空间中任意占用页的存取保护。举例来说，一个进程可以分配读写页来保存易受影响的数据，

然后将存取改变为只读或禁止访问，以避免无意中被重写。典型的，VirtualProtect用于使用VirtualAlloc分配的页，但事实下，它也可以用于通过其它分配函数占用的页。然而，VirtualProtect改变整个页的保护状态，而由其它函数返回的指针并非总是指向页边界。函数VirtualProtectEx类似于VirtualProtect，但函数VirtualProtectEx可以改变指定进程的内存的保护状态。改变这些内存的保护状态在调试程序访问被调试进程的内存的非常有用。

## 第四章 基于对话框的应用程序

从本章我们将学习如何使用Visual C++进行真正意义上的Windows应用程序设计。对于编程者来说，最简单的实用程序应该是本章所要讲述的基于对话框的应用程序。当然，我们已在前面的章节中介绍了基于包括用户区的一般窗口的最简单的Windows程序，相比本章将要讲述的程序来说，它们还要更简单一些。但是要记住，在前面讲述的这些应用程序中，我们没有实现任何实际的功能。事实上，在这些应用程序中，实现一些哪怕是很简单的功能也需要很多的代码量。举一个例子，如果我们需要在用户区绘制图形或文本的话，就需要为Windows消息WM\_PAINT编写消息处理函数。一个实用的Windows应用程序的WM\_PAINT的消息处理过程常常会很庞大和很复杂。而对于在这章将要讲述的基于对话框的应用程序来说，整个应用程序都是由一个或多个对话框(dialog box)组成，对话框只是作为其它一些行为标准化了的窗口(我们叫它们控件(control))的容器，其行为也是标准化了的。这样，Windows就可以为这些行为标准化了的窗口实现很多默认的操作，而不需要我们进行更多的干预。比如对于一个文本框，我们只需要预先设定文本框的文字及其它一些属性，Windows就可以知道在不同的情况下这个文本框应该进行什么样的操作，因为它的行为，如编辑和选定文本等，都已经是标准化了的。虽然各种各样的控件各有各的特点，但是，对于一类控件来说，它们的行为在某种意义是一致的。(从这个意义上来说，这是面向对象程序设计的很大程度上的优点所在。)

使用MFC编写的对话框应用程序和使用SDK编写的对话框应用程序在结构上有着很大的不同。对于使用Visual C++的程序员来说，我们有充分的理由使用MFC来编写这些应用程序，因此，我们在这里将不讲述如何使用SDK编写的基于对话框的应用程序。

本章涉及的内容包括：

- 使用AppWizard生成基于对话框的框架应用程序
- 由CWinApp派生的应用程序类
- MFC的消息映射机制及其实现
- 对话框及由CDialog派生的对话框类

### 第一节 使用AppWizard生成应用程序框架

在编写MFC应用程序时，我们通常使用AppWizard来生成应用程序框架，然后再在此框架的基础上来添加特定于应用程序的功能的实现。在计算机术语中，Wizard通常被译作向导(AppWizard即应用程序向导)，它是这样的一种程序：你只需要回答一系列的与你所需完成的操作有关的问题，Wizard就会自动的完成其余的步骤，而这些步骤如果通过手工来完成的话，将可能会耗费相当长的时间和精力（但同时我们也要向你指出，AppWizard所能够帮助你完成的，也只是一个应用程序的框架。它所建立的，在绝大部分是我们在Windows 下程序设计中所需要完成的例行化的工作）。在Visual C++中，一种被称作AppWizard的向导，通过向你询问一些关于所需编写的应用程序各项特性的问题，即可按照程序员的要求生成相应的框架文件，这些框架文件本身就构成了一个完整的Windows应用程序，它实现了绝大多数同类型的Windows所共有的一些特性和功能。

- 注意：

- 我们还想说明的一点是：AppWizard能够帮助我们建立起一个应用程序的框架，但绝大多数的应用程序的代码还需要我们亲自编写。我们还从来没有看到仅仅通过AppWizard的代码就生成了一个成功的程序。明白这一点是很重要的：AppWizard所做的，只不过是我们在程序设计过程中所需要的最没有创意的那一部分事情。

真正应该引起我们更多的重视的，是Visual C++的可视化的界面生成。要知道，在可视化编程出现以前，为了一个简单的框架，程序员不得不重复干一些相当烦琐的工作：仅仅是为了安排好一个对话框中的控件的位置，他们就不得不先在稿纸上试着画布局图，写入程序，运行，再修改，再运行。。。你可以想见，这是一件多么烦琐乏味的工作！

最后我们想提醒你，如果你不需要一个标准的Windows 程序界面或者不需要AppWizard所提供的文档/视结构，使用AppWizard并不一定是一个明智的选择。



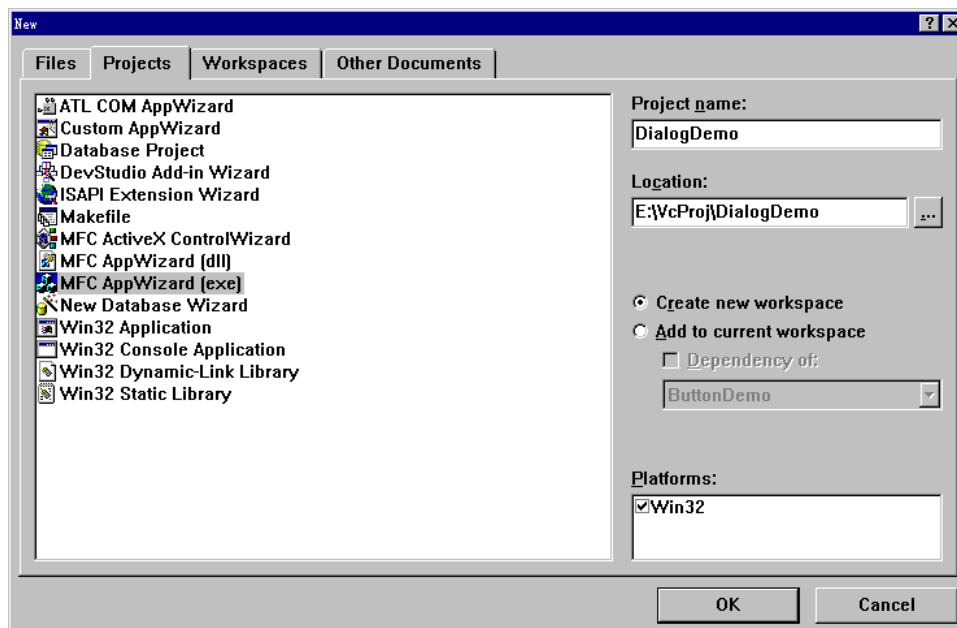


图4. 1 新建工程

使用AppWizard是一件简单和愉快的事情，并且，明白AppWizard所能完成的和所不能完成的内容会在编程时少走很多的弯路，因此，我们将在下面的过程中详细的讲述如何使用AppWizard创建基于对话框的应用程序：

1. 选择文件菜单下的New命令（出于排版方面的一些考虑，我们将参考图形进行了一定程度的处理），如图4.1所示。
2. 从New对话框中选择Project选项卡。在Project name处输入工程名，一般来说，工程的命名在一定程度上是任意的，这里我们假定工程名为DialogDemo，在Location处输入保存工程的文件夹。然后在左边的列表中确信选择了MFC AppWizard (exe)，在Platform列表中确信选择了Win32。完成之后单击OK进入下一步。
3. 随后弹出如图4.2所示的对话框。在该对话框中选择应用程序类型为Dialog based，即基于对话框的应用程序。还可以在下面的下拉列表框中选择应用程序资源所使用的语言。这里我们选择了简体中文，即“中文[中国](APPWZCHS.DLL)”。这样，AppWizard为应用程序自动创建的所有资源都将是中文的。单击Next进入下一步。

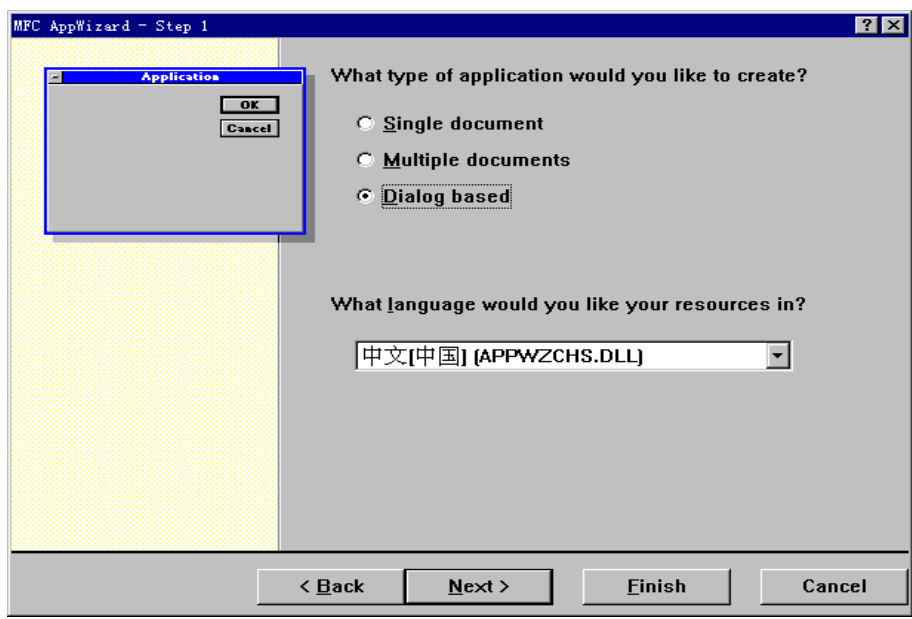


图4. 2 使用AppWizard创建基于对话框的应用程序：第一步

- 注意：
- 在默认情况下，安装Visual C++时并没有安装支持远东语言的动态链接库，这些语言包括简体中文、繁体中文、日文和韩文等，对这些语言的支持需要相应的双字节的操作系统。因此，在如图4.2所示的MFC AppWizard对话框中将不会看到这些语言的选择项。为了添加对这些语言的支持，我们必须手动的将它们添加Visual C++的安装目录下。这些语言的支持文件在Visual C++安装光盘上的\DevStudio\SharedIDE\Bin\IDE目录下，不同的语言所对应的动态链接库的.DLL文件名如表4.1所示。

表4. 1 不同的远东语言所对应的支持文件

语言	支持该语言的动态链接库
中文(简体)	APPWZCHS.DLL
中文(繁体)	APPWZCHT.DLL
日文	APPWZJPN.DLL
韩文	APPWZKOR.DLL

如果在你的AppWizard中还没有添加对上面的这些语言的支持文件，可以将它们从光盘上的  
\\DevStudio\\SharedIDE\\Bin\\IDE目录中将它们复制到对应的Visual C++安装目录下，举个例子说，如果你的Visual C++被安装到D:\\Program Files\\DevStudio目录下，对应的目录将是D:\\Program Files\\Devstudio\\SharedIDE \\Bin\\IDE。然后再在操作系统中安装对应的代码页。在很多情况下，我们还需要重新启动Developer Studio或操作系统。

- 如果应用程序使用MFC的动态链接，还必须有相对应的MFC资源动态链接库的本地化版本，它们位于Windows系统目录下，且具有MFC40LOC.DLL的文件名。可以将Visual C++安装光盘上的MFC\\include\\L.XXX\\MFC40XXX.DLL 目录下的对应DLL文件复制到Windows的系统目录下，然后将其改名为MFC40LOC.DLL。详细的内容可以参考帮助中的Microsoft Foundation Class Reference\\MFC Technical Notes节点下的文章TN056和TN057。
- 如果你使用的是Visual C++的专业版或企业版，还可以在应用程序中使用MFC的静态链接。这时，需要在MFC\\[src|include]\\L.XXX\\\*.rc目录下有正确的本地化的MFC资源文件。这些文件可以在Visual Studio的第一章安装光盘上的对应目录中找到。

4. 在如图4.3所示的对话框中为应用程序选择合适的特性。该对话框中各选项的含义如下：

About box : 如果选择了该选项，AppWizard将为一个被称为“关于”对话框的消息框生成代码，该消息框用来显示应用程序的版本号和版权信息等。绝大多数的Windows应用程序都具有一个关于对话框，图4.4是Microsoft Word 95的“关于”对话框。在默认情况下，使用AppWizard创建的应用程序，都具有一个“关于”对话框。

Context-sensitive help : 该选项决定是否让AppWizard为应用程序创建上下文相关的帮助文件。需要注意的是，对帮助的支持需要帮助编译器(help compiler)，如果在你的Visual C++中没有安装帮助编译器的话，可以重新运行Setup程序来安装它。

- 3D controls : 决定是否在应用程序中使用具有三维阴影的用户界面。使用了三维外观的应用程序界面看起来象那些在Windows的早期版本(如Windows 3.x)下运行的程序的外观。默认情况下,使用AppWizard创建的应用程序都具有三维外观。
- Automation : 决定应用程序是否可以操作由其它程序实现的对象,也就是说是否可以将程序作为自动化客户(Automation client)。

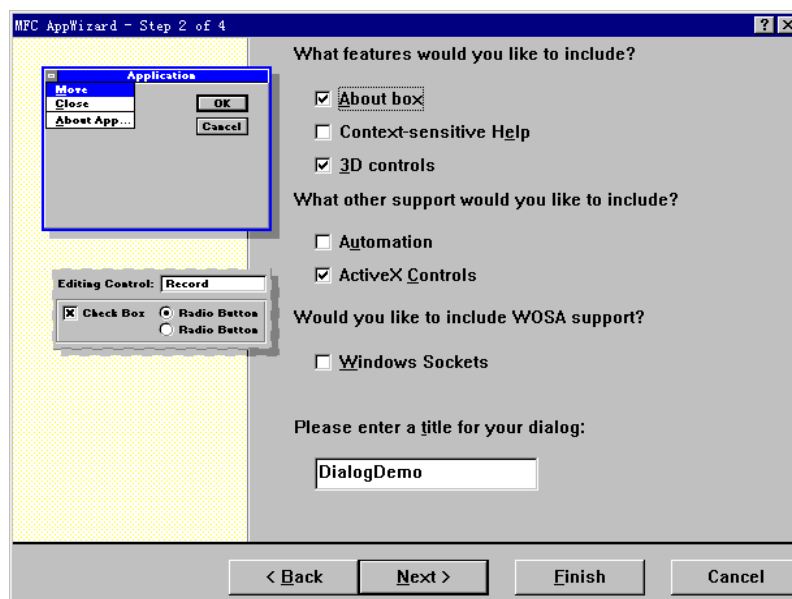


图4. 3 使用AppWizard创建基于对话框的应用程序: 第二步



图4. 4 应用程序中的“关于”对话框

- ActiveX 决定你的应用程序是否使用ActiveX控件。

- controls : 如果在创建应用程序框架时没有选择该选项，则必须在InitInstance成员函数中添加对AfxEnableControlContainer的调用来向工程中插入ActiveX控件。在默认情况下，使用AppWizard创建的应用程序可以使用ActiveX控件。
- Windows sockets : 该选项决定应用程序是否支持Windows套接字。Windows套接字允许应用程序之间通过基于TCP/IP的网络进行通信。

在图4.3所示对话框的最下部可以输入对话框所用的标题。在默认情况下AppWizard将工程名作为对话框的标题。

设置完成后单击Next继续。

5. 在如图4.5所示的对话框中决定是否让AppWizard为所创建的源文件添加上注释和使用哪一种链接方式。可以使用两种方式链接到MFC，一种方式是使用动态链接库DLL，即选择As a shared DLL；第二种方式是像过去所常用的方式那样，使用MFC的静态链接，即选择As a statically linked library。两种方式各有优缺点，使用DLL可以有效的应用程序执行文件的大小，但在运行时必须保证应用程序能够找到所需的动态链接库。使用静态链接的应用程序不需要额外的动态链接库的支持，而且可以在性能上获得少许的提高，但是，使用静态链接的应用程序的可执行文件可能会大上很多倍。

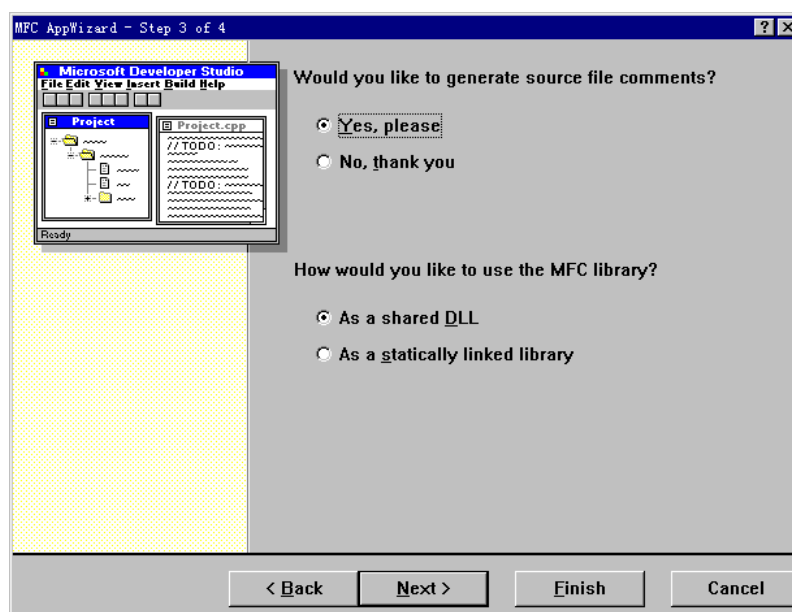


图4. 5 使用AppWizard创建基于对话框的应用程序：第三步

6. 在图4.6所示的对话框中更改AppWizard创建的类的类名、基类及实现该类的头文件和实现文件。要注意的是，并不是所有的类的所有项都可以更改，如果某一项所对应的文本框中的内容的显示变为灰色，则表示该项内容不可以修改。如图4.6所示的对话框，我们不可能更改类CDialogDemoApp的基类、头文件和实现文件，但是我们却可以更改它的类名。修改完成之后，单击Finish。这时，AppWizard弹出如图4.7所示的对话框，该对话框给出了对你在前几步中所做的设置的总结，如果一切正确无误的话，单击OK接受，这里AppWizard根据你的设定定制应用程序的各个代码文件和资源文件，这些文件将在本章的后面部分中进行讲述。反之，如果在前面的某一步中的设置有误，则可以单击Cancel，然后重新运行AppWizard。

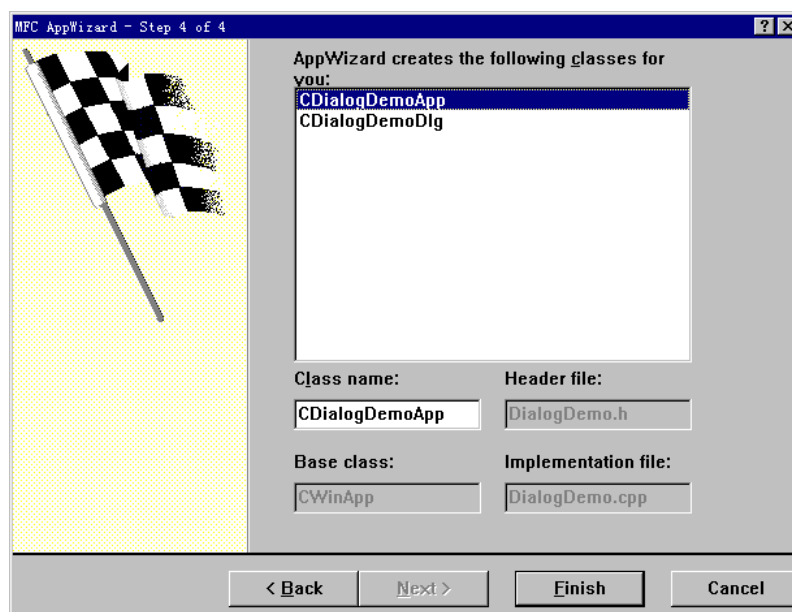


图4. 6 使用AppWizard创建基于对话框的应用程序：第四步

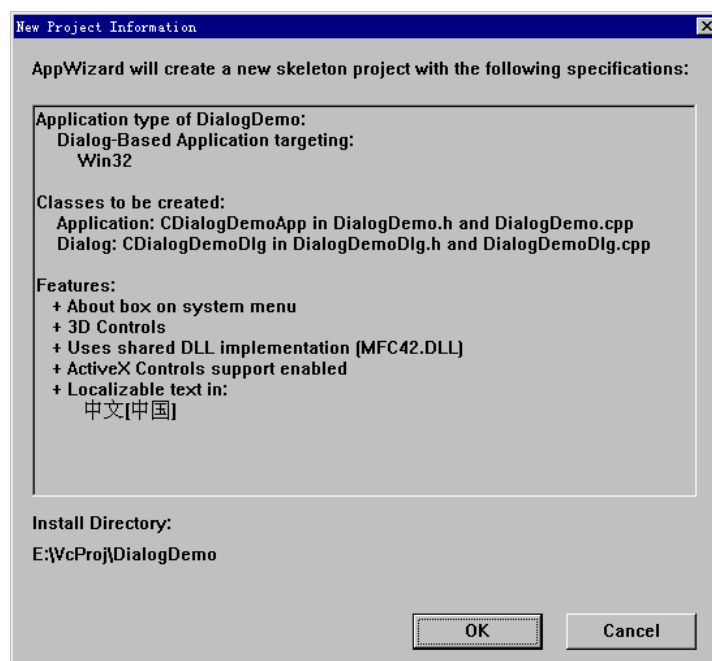


图4. 7

- 注意：
- 在单击了Finish按钮之前，你可以使用Back按钮和Next按钮在AppWizard的各个步骤之间进行切换，并更改不合适的设置，但是，一旦单击了Finish之后，除了直接手动的修改源文件或是推翻重来以外，一切行为都是不可逆的。因此，在使用AppWizard之后还需添加某些特性往往需要很多额外的工作，因此，如果是编写新的应用程序的话，我们建议在创建应用程序之前先进行完整的构思，以便于在使用AppWizard时充分发挥其作用，争取让它完成更多的工作，以减轻编写Windows应用程序的复杂程度。

## 第二节 应用程序类

在创建应用程序框架时，AppWizard同时生成了一个说明文件ReadMe.txt。该说明文件是英文的，为了便于读者阅读，我们在下面给出的工程DialogDemo的说明文件ReadMe.txt已被翻译为中文。基于同样的考虑，在本书中给出示例程序时，对于所有的程序注释，我们一律使用中文；对于由AppWizard等生成的程序注释，也一律翻译为中文。注意对于不同的工程，其说明文件的内容也是不同的，但它们都具有大致相同的结构。

工程DialogDemo的说明文件ReadMe.txt：

```
=====
MICROSOFT FOUNDATION CLASS LIBRARY : DialogDemo
=====
```

AppWizard 已为您创建了应用程序 DialogDemo。 该应用程序不仅演示了 Microsoft Foundation Class 的基本使用，而且还可以作为您编写应用程序的起点。

本文件包括了对构成应用程序 DialogDemo 的每一个文件的总结。

DialogDemo.h

该文件是应用程序的主要头文件。它包括了其它工程特定的头文件(包括 Resource.h)和声明 CDialogDemoApp 应用程序类。

DialogDemo.cpp

该文件是工程的主要应用程序源文件，它包括了应用程序类 CDialogDemoApp 的实现。

DialogDemo.rc

该文件是程序所使用的所有 Microsoft Windows 资源的列表。它包括保存在目录 RES 下的图标、位图和光标。该文件可以在 Microsoft Developer Studio 中直接编辑。

res\DialogDemo.ico

应用程序图标所使用的图标文件。该图标被包括在文件 DialogDemo.rc 中。

res\DialogDemo.rc2

该文件包括的资源不被 Microsoft Developer Studio 编辑。你可以将不可以被资源编辑器编辑的资源放入此文件。

DialogDemo.clw

ClassWizard 使用该文件所包括的消息来编辑已有的类或添加新类。ClassWizard 还使用该文件来保存在创建和编辑消息映射和对话框数据映射以及创建成员函数原型时所需的信息。

//

AppWizard 创建了一个对话框类：

DialogDemoDlg.h, DialogDemoDlg.cpp - 对话框

这些文件包括了类 CDialogDemoDlg。该类定义了应用程序主对话框的行为。对话框的模板包括在 DialogDemo.rc 中，可以使用 Microsoft Developer Studio 来编辑该文件。

//

其它标准文件：

StdAfx.h, StdAfx.cpp

这些文件用于创建预编译头文件(PCH)，该文件名为 DialogDemo.pch，预编译类型



文件名为 StdAfx.obj。

Resource.h

定义新资源 ID 的标准头文件。Microsoft Developer Studio 读取和更新该文件。

//

其它注意事项：

AppWizard 使用 "TODO:" 来指出需要添加或定制源代码的地方。

如果你的应用程序使用了 MFC 的共享动态链接库，并且你的应用程序使用了与操作系统当前语言不同的语言，那么你需要将相应的本地化资源文件 MFC40xxx.DLL 从 Microsoft Visual C++ CD-ROM 上复制到 system 或 system32 目录下，并将其改名为 MFCLOC.DLL。 ("xxx"代表指定语言的缩写，如 MFC40DEU.DLL 包括了已被翻译为德文的资源。)如果你没有进行这个步骤，应用程序的一些用户界面元素仍将保持为操作系统的语言。

//

下面我们从头文件StdAfx.h入手，来分析该应用程序。头文件StdAfx.h的列表清单如下：

```
// stdafx.h : 本包含文件包含了标准系统包含文件，以及经常使用的工程特定的包含
```

```
// 文件，在很多情况下，我们并不需要修改这些文件。
```

```
//
```

```
#if !defined(AFX_STDAFX_H__7ABABF8C_0C8C_11D2_BC21_0000B4810A31__INCLUDED_)
```

```
#define AFX_STDAFX_H__7ABABF8C_0C8C_11D2_BC21_0000B4810A31__INCLUDED_
```

```
#if _MSC_VER >= 1000
```

```
#pragma once
```

```
#endif // _MSC_VER >= 1000
```

```
#define VC_EXTRALEAN // 从 Windows 头文件中排除很少用到的那一部分
```

```
#include <afxwin.h> // MFC 核心和标准部件
```

```
#include <afxext.h> // MFC 扩展
```

```
#include <afxdisp.h> // MFC OLE 自动化类
```

```

#ifdef _AFX_NO_AFXCMN_SUPPORT

#include <afxcmn.h> // MFC 对 Windows 公用控件的支持

#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}

// Microsoft Developer Studio 将在上一行之最接近的地方添加附加的声明

#endif // !defined(AFX_STDAFX_H__7ABABF8C_0C8C_11D2_BC21_0000B4810A31__INCLUDED_)

与头文件相对应的实现文件为StdAfx.cpp文件，该文件只有一行代码，如下面的清单所示：

// stdafx.cpp：包含标准包含文件的源文件

// DialogDemo.pch 为预编译头文件

// stdafx.obj 中包含了预编译类型信息

#include "stdafx.h"

```

StdAfx.h文件是Visual C++工程的预编译头文件，将一些常用的并且很少需要修改的头文件放入StdAfx.h中可以有效的提高Visual C++的编译速度。

在StdAfx.h头文件中定义的标识符VC\_EXTRALEAN将从Windows包含文件中排除了一些不常用的头文件，从而提高应用程序的编译速度，但是，如果应用程序中需要使用在被排除的这些头文件中声明的函数，必须额外的添加这些对这些头文件的包含，否则将会导致编译出错。

对其它包含头文件的说明请参见StdAfx.h头文件清单中的注释。

在头文件Resource.h中包含了对程序和资源中所用到的常量的定义，其清单如下：

```

//{{NO_DEPENDENCIES}}

// Microsoft Visual C++ 生成包含文件.

// 由 DIALOGDEMO.RC 使用

//

#define IDR_MAINFRAME 128

#define IDM_ABOUTBOX 0x0010

```

```

#define IDD_ABOUTBOX 100

#define IDS_ABOUTBOX 101

#define IDD_DIALOGDEMO_DIALOG 102

// 新对象的下一个默认值

//

#ifdef APSTUDIO_INVOKED

#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE 129

#define _APS_NEXT_COMMAND_VALUE 32771

#define _APS_NEXT_CONTROL_VALUE 1000

#define _APS_NEXT_SYMED_VALUE 101

#endif

#endif

```

下面我们再来分析一下头文件DialogDemo.h：

```

// DialogDemo.h：应用程序 DIALOGDEMO 的主要头文件

//

#ifndef AFX_DIALOGDEMO_H__7ABABF88_0C8C_11D2_BC21_0000B4810A31__INCLUDED_

#define AFX_DIALOGDEMO_H__7ABABF88_0C8C_11D2_BC21_0000B4810A31__INCLUDED_

#if _MSC_VER >= 1000

#pragma once

#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__

#error include 'stdafx.h' before including this file for PCH

#endif

#include "resource.h" // 主要符号

////////////////////////////////////

```

```

// CDialogDemoApp:

// 在 DialogDemo.cpp 中包括了该类的实现

//

class CDialogDemoApp : public CWinApp

{

public:

CDialogDemoApp();

// 重载

// ClassWizard 生成的虚函数重载

//{{AFX_VIRTUAL(CDialogDemoApp)

public:

virtual BOOL InitInstance();

//}}AFX_VIRTUAL

// 实现

//{{AFX_MSG(CDialogDemoApp)

// 注意 - ClassWizard 将在这里添加或删除成员函数

// 不要编辑你在这里所看到的这些生成代码块！

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

////////////////////////////////////

//{{AFX_INSERT_LOCATION}}

// Microsoft Developer Studio 将在紧接着上一行之前的地方插入附加的声明。

#endif // !defined
(AFX_DIALOGDEMO_H__7ABABF88_0C8C_11D2_BC21_0000B4810A31__INCLUDED_)

```

在头文件DialogDemo.h中定义了应用程序类CDialogDemoApp。在MFC

中，应用程序类封装了一个Windows应用程序的初始化、运行和终止。每一个MFC应用程序都必须包括一个从CWinApp派生的应用程序类，在应用程序DialogDemo中，这个类就是CDialogDemoApp。这个对象在窗口被创建之前进行构造。类CWinApp从类CWinThread派生，它代表了应用程序的主执行线程，正如我们在前面的章节中所讲述的那样，一个应用程序可以有多个执行线程。在MFC的最近看到版本中，已为类CWinThread的成员函数InitInstance、Run、ExitInstance和OnIdle提供了默认的实现。由于我们在讨论CWinApp时更侧重于它作为应用程序对象所扮演的角色，而不是作为主线程，因此，我们在讨论前面提到的四个成员函数的时候，可以把它们想象为在类CWinApp定义的成员函数，即立足于应用程序对象的角度来看待和分析它们的行为，而不是立足于一个单独的线程的角度。

应用程序类构成了应用程序的主执行线程。使用Win32 API函数可以创建新的执行线程。这些线程仍可以使用MFC库，详细的信息请参考本书关于进程和线程的有关章节。

同其它的任何Windows应用一样，框架应用程序仍然具有一个WinMain函数。但是，在由AppWizard生成的应用程序框架中，我们却找不到对WinMain函数的声明或定义。在MFC应用程序中，WinMain函数是由类库提供的，它在应用程序启动时被调用。WinMain函数执行如注册窗口类之类的标准服务。接着，它调用应用程序对象的成员函数来初始化并且运行应用程序。通过重载WinMain函数所调用的类CWinApp的成员函数可以自定义WinMain函数。

WinMain函数通过调用应用程序对象的InitApplication和InitInstance成员函数来初始化应用程序，通过调用Run成员函数运行应用程序的消息循环，最后在程序结束时调用程序程序的ExitInstance成员函数。其中Run函数一般由MFC提供，而InitApplication、InitInstance和ExitInstance一般需要程序员创建或进行重载。

- 注意：
- 由于在Win32环境下，每一个应用程序的实例都是独立的，它们有着自己的虚拟地址空间，因此，在目前版本的MFC中，成员函数CWinApp::InitApplication已被废弃，原来在InitApplication中进行的初始化操作应该移到InitInstance中进行。

由AppWizard生成的基于对话框的框架应用程序提供了对InitInstance的默认重载，也正是在该成员函数中提供了基于对话框

的应用程序的特点。在MFC中，无论是基于对话框的应用程序，还是基于文档/视结构的应用程序，它们的应用程序对象都是从CWinApp派生而来的，它们之间的功能的巨大差异，往往就是通过对类CWinApp的成员函数进行不同的重载来体现的。

在每一个同一应用程序的实例被启动时，WinMain函数调用一次InitInstance成员函数。原有的InitApplication不再有意义，每一个应用程序的实例都被认为是独立的，对应用程序的初始化同对实例的初始化没有本质区别。对于InitInstance成员函数，我们这时只需要知道它在当应用程序进行初始化的时候由WinMain函数调用。关于基于对话框的应用程序的InitInstance成员函数的典型实现，我们将在本章后面的内容中专门讲述。

类CDialogDemoApp的实现包含在文件DialogDemo.cpp中，其清单如下：

```
// DialogDemo.cpp : 为应用程序定义类的行为

//

#include "stdafx.h"

#include "DialogDemo.h"

#include "DialogDemoDlg.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CDialogDemoApp

BEGIN_MESSAGE_MAP(CDialogDemoApp, CWinApp)

//{{AFX_MSG_MAP(CDialogDemoApp)

// 注意 - ClassWizard 将在此添加或删除映射宏。

// 不要删除你在这里看到的这些生成代码块！
```

```

//}}AFX_MSG

ON_COMMAND(ID_HELP, CWinApp::OnHelp)

END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////

// CDialogDemoApp 构造

CDialogDemoApp::CDialogDemoApp()

{
    // TODO: 在这里添加构造代码 ,
    // 将所有重要的初始化放入 InitInstance
}

/////////////////////////////////////////////////////////////////

// 唯一的 CDialogDemoApp 对象

CDialogDemoApp theApp;

/////////////////////////////////////////////////////////////////

// CDialogDemoApp 初始化

BOOL CDialogDemoApp::InitInstance()

{
    AfxEnableControlContainer();

    // 标准初始化

    // 如果你不需要使用这些特性，并且希望减小最终可执行文件的大小，你可以删除
    // 下面的特定的初始化过程中不需要的部分。

#ifdef _AFXDLL

    Enable3dControls(); // 当通过共享 DLL 使用 MFC 时调用

#else

    Enable3dControlsStatic(); // 当通过静态链接到 MFC 时调用

#endif
}

```

```

CDialogDemoDlg dlg;

m_pMainWnd = &dlg;

int nResponse = dlg.DoModal();

if (nResponse == IDOK)
{
    // TODO: 在这里添加当使用 OK 关闭对话框时的处理代码
}

else if (nResponse == IDCANCEL)
{
    // TODO: 在这里添加当使用 Cancel 关闭对话框时的处理代码
}

// 由于对话框已被关闭，返回 FALSE 并退出应用程序，而不需要启动应用程序
// 消息泵。

return FALSE;

}

```

上面的源代码为类CDialogDemoApp提供了一个空的构造函数和一个对InitInstance的默认重载。我们把讨论的重点放在InitInstance成员函数上。在InitInstance成员函数的一开始，先调用MFC全局函数AfxEnableControlContainer，该函数为应用程序提供了对OLE控件(新的术语称作ActiveX控件)的支持。

接着，InitInstance成员函数调用类CWinApp的成员函数Enable3dControls或Enable3dControlsStatic以允许对话框和窗口可以使用具有三维外观的控件。这两个成员函数加载CTL3D32.DLL并且注册应用程序。Enable3dControls和Enable3dControlsStatic的区别在于一个在链接到MFC动态链接库时使用，而另一个在使用MFC的静态链接时使用。

MFC自动为以下窗口类提供3D控件效果：

- CDialog
- CDialogBar



- CFormView
- CPropertyPage
- CPropertySheet
- CControlBar
- CToolBar

如果你所希望得到3D效果的控件属于以上类型之一的話，你只需调用Enable3dControls或Enable3dControlsStatic即可。反之则必须直接调用相应的CTL3D32 API函数。

然后在InitInstance中定义了类型为CDialogDemoDlg的对象dlg，然后将其指针赋予类型为CWnd的成员变量m\_pMainWnd。成员变量m\_pMainWnd用来保存指向线程主窗口对象的指针，当由m\_pMainWnd引用的窗口被关闭时，该线程由MFC自动终止。当应用程序的主线程被终止时，该应用程序相应的也被终止。如果该成员的值为NULL，则应用程序的CWinApp对象的主窗口被用来判断线程何时终止。成员m\_pMainWnd具有共有访问权限。对于工作者线程而言，该数据成员的值从其父线程继承。

接着InitInstance调用了对象dlg的成员函数DoModal，该成员函数以模态方式调用对话框并在结束时返回对话框的结果。在对话框激活时，该成员函数处理所有与用户的交互，也就是说，对于模态对话框，用户不可以在对话框关闭之前与其它窗口进行交互。

如果用户单击了对话框中的OK或Cancel按钮，相应的消息处理函数，如OnOK或OnCancel被调用以试图关闭对话框。OnOK成员函数的默认行为为验证和更新对话框数据并以结果IDOK关闭对话框，OnCancel的默认行为为以结果IDCANCEL关闭对话框并不更机关报所有对话框数据。通过重载这些消息处理函数可以改变它们的行为。

在DoModal返回时，对话框将被关闭，理所当然的，基于该对话框的应用程序也应该被关闭，因此在InitInstance的最后使用了语句

```
return FALSE;
```

### 第三节 MFC应用程序的消息循环

上面的对应用程序的类的定义和声明还包括了MFC的消息循环

(message loop)，下面我们来详细的描述MFC应用程序的消息循环。框架应用程序处理Windows消息的方式同其它Windows应用程序是类似的，只不过它提供了一些方法来使得这个过程更加的方便，更加的易于维护和更好的包装。

- 注意：
- 为便于读者理解，在此我们给出一个MFC对对话框控制的支持的树图(如图4.8所示)。

在类CWinApp的Run成员函数中的消息循环获取各种事件所产生的排队消息(queued message)，应用程序消息循环的框架实现是将它们分发到合适的窗口。

在MFC中，每一个单独的消息都由一个专门的函数进行处理，这种称作消息处理函数(message-handler function或message handler或handler)的专门函数以类的成员函数的方式进行定义。处理命令消息的函数还常被称作命令处理函数(command handler)。

Windows应用程序是消息驱动的，因此编写消息处理函数就成了编写框架应用程序的工作中的一个很大的组成部分。

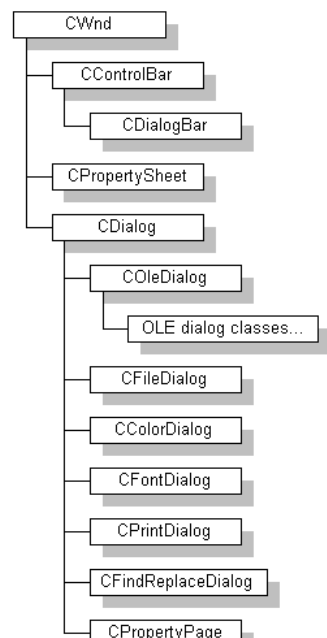


图4. 8 MFC中对话框控件类的树状结构图

每一个有能力获取消息或命令的框架类都有它自己的消息映射(message map)，框架使用消息映射来建立消息和命令到它们的处理函数之间的链接。所有从类CCmdTarget派生的类都可以有它自己的消息映射。尽管我们常常区别消息和命令两个术语，但这里所说的消息

映射同时对它们进行处理。

下面我们来看一下消息是如何发送和获取的。

很多消息来自于用户和程序之间的交互：当用户使用鼠标单击了菜单项或工具条按钮或按下了快捷键时，即产生了命令。同样由用户产生的Windows消息还可以来自移动窗口或改变窗口的大小。此外，当程序启动或终止、窗口获得或失去焦点等等事件发生时，相应的Windows消息也将被发送。控件作为一种特殊形式的窗口，相应的控件通知消息也在类似的情形下产生。

Run成员函数用来获取消息并将它们发送到合适的窗口，很多的命令消息被发送到应用程序的主窗口，由类库预先定义的WindowProc函数获得这些消息，然后根据所获得的消息的类型来以不同的方式对它们进行处理。

最初接受消息的必须是一个窗口对象。Windows消息通常直接由该窗口对象进行处理。而命令消息一般由应用程序的主框架窗口开始，按照命令路径描述的命令目标链进行处理。

当一个命令目标链获得消息或命令时，它将搜索它的消息映射以寻找匹配项。如果该消息的一个处理函数存在，该处理函数将被调用。

与命令不同，对于标准Windows消息，它们并不经过命令目标链，通常由该消息的目标窗口进行处理，这个目标窗口可能是主框架窗口，也可能是一个MDI子窗口、一个标准控件、对话框、视或其它形式的子窗口。

在运行时，每一个Windows窗口都与一个窗口对象建立关联，该窗口对象由直接的或间接的由类CWnd派生，并且有着它自己的消息映射和处理函数。框架使用这个消息映射来将到来的消息映射到它们的处理函数。

对于命令，我们所需做的只是建立命令到它们的处理函数之间的链接，通常使用ClassWizard来完成这一步工作，然后编写绝大多数的命令处理程序。

Windows消息通常发送到主框架窗口，但是命令消息将可能传送到其它的对象，框架通过一个命令目标对象的标准顺序来传递命令，这些命令目标对象至少有一个可能(并不是一定)包含该命令的处理函数。每一个命令目标对象检查它的消息映射以查看是否有一个命令处理函数来处理到来的消息。

不同的命令目标类在不同的时候检查它们自己的消息映射。典型的，一个类将命令传送给其它一些类，这使得这些类具有最先处理该消息的机会。如果这些类中没有一个处理了该命令，最初的那个类将检查它的消息映射，然后，如果它也没有提供相应的处理函数，它可能会将该命令传送给更多的命令目标。表4.2描述了构成这一顺序的结构。通常的顺序是先传送给当前激活子命令目标对象，再传送给自身，最后传送给其它的命令目标。

表4.2 标准命令传送路径

该类型的对象获取某一命令时.....	它将按下面的顺序给自身和其它命令目标对象处理该命令的机会
MDI 框架窗口 (CMDIFrameWnd)	<ol style="list-style-type: none"> <li>1. 当前激活 CMDIChildWnd</li> <li>2. 当前框架窗口自身</li> <li>3. 应用程序(CWinApp对象)</li> </ol>
文档框架窗口 (CFrameWnd, CMDIChildWnd)	<ol style="list-style-type: none"> <li>1. 当前激活视</li> <li>2. 当前框架窗口</li> <li>3. 应用程序(CWinApp对象)</li> </ol>
视	<ol style="list-style-type: none"> <li>1. 当前视</li> <li>2. 与视相关联的文档</li> </ol>
文档	<ol style="list-style-type: none"> <li>1. 当前文档</li> <li>2. 与文档相关联的文档模板</li> </ol>
对话框	<ol style="list-style-type: none"> <li>1. 当前对话框</li> <li>2. 拥有当前对话框的窗口</li> <li>3. 应用程序(CWinApp对象)</li> </ol>

上面的过程看起来很复杂，并且添加了程序的开销，但是相比处理程序对命令的响应来说，传送命令的开销要小得多，因为仅当用户与一个用户界面对象进行交互时框架才生成相应的命令。

当使用AppWizard创建新的框架应用程序(skeleton application)时，AppWizard就已经为它所创建的每一个命令目标类编写了相应的消息映射。在这些消息映射中，有一些已经添加了对某些消息和预定义命令的处理，而其它一些只是为了下一步添加处理函数的占位符。

类的消息映射位于该类的.CPP文件中，我们通常使用ClassWizard为每一个类将要处理的消息和命令添加入口。一个典型的消息映射具有如下的结构，它来自文件DialogDemo.cpp：

```
BEGIN_MESSAGE_MAP(CDialogDemoApp, CWinApp)

//{{AFX_MSG_MAP(CDialogDemoApp)

// 注意 - ClassWizard 将在此添加或删除映射宏。

// 不要删除你在这里看到的这些生成代码块！

//}}AFX_MSG

ON_COMMAND(ID_HELP, CWinApp::OnHelp)

END_MESSAGE_MAP()
```

上面的消息映射包括一系列的宏。消息映射位于两个宏——BEGIN\_MESSAGE\_MAP和END\_MESSAGE\_MAP之间，其它的宏，如ON\_COMMAND构成了消息映射的内容。

- 注意：
- 在消息映射宏的后面没有分号。

消息映射还包括了下面形式的注释：

```
//{{AFX_MSG_MAP(CDialogDemoApp)

//}}AFX_MSG_MAP
```

在两行注释之间包括了消息映射入口，但不要求所有的消息映射入口都在这两行注释之间。当使用ClassWizard编写入口时，它将使用这些特殊的注释。所有由ClassWizard生成的注释都位于这两行注释之间。

- 注意：
- 除非你确实不想在程序中再使用ClassWizard，不要更改//{{AFX\_MSG\_MAP和//}}AFX\_MSG\_MAP记号，这是ClassWizard进行程序相关数据库管理的特殊标记。

当使用ClassWizard创建新类时，相应的消息映射将由ClassWizard自动生成。而且，在前面的示例代码中，我们还看到了不要随意修改由ClassWizard生成的消息映射项的警告。但是，对于那些有经验的程

序员，使用源代码编辑器来手动的创建消息映射也是完全可行的。

我们注意到前面的BEGIN\_MESSAGE\_MAP具有下面的格式，它具有两个参数：CDialogDemoApp和CWinApp。

```
BEGIN_MESSAGE_MAP(CDialogDemoApp, CWinApp)
```

第一个参数CDialogDemoApp表示消息映射所属的类，第二个参数CWinApp表示CDialogDemoApp的直接基类，这向我们暗示了一点，这就是说，如果框架在类CDialogDemoApp中没有找到某一特定消息或命令的映射入口，它将按照类的继承结构依次查找该与该消息或命令相匹配的入口。如果按照这种方式还是未能找到一个匹配的映射项，对于命令，框架将将它传送给下一个命令目标，对于标准Windows消息，框架将将它传递给一个合适的默认窗口过程。为了加速消息映射匹配的速度，框架使用了一种类似于磁盘缓存的机制，它保存了与最近匹配项有关的信息，以便在获取相同的消息时可以很快的找到与消息相匹配的消息映射。事实上，消息映射同使用虚函数相比，在某些方面要更为有效。

下面我们讨论一下消息处理函数的声明。消息处理函数的声明需要遵从一些规则与协议，这些规则和协议因消息所属的种类不同而有所不同。

在类CWnd中定义了标准的Windows消息处理函数，这些消息以前缀WM\_开头。相应的消息处理函数的命名基于消息的名称。举例来说，消息WM\_PAINT的处理函数在CWnd中被声明为

```
afx_msg void OnPaint();
```

关键字afx\_msg使得上面的函数看起来和其它的CWnd成员函数有所不同，然而实际上，经过预处理之后，afx\_msg将被空白所代替，也就是说，除了可以使程序员很清楚的知道哪一些函数是消息处理函数，而哪一些是一般的成员函数。（实际上，该关键字为Microsoft公司为今后所作的保留字。）消息处理函数只通过消息映射来实现，而消息映射仅依赖于几个标准的预处理宏。

如果需要重载在基类中定义的消息处理函数，只需简单的使用ClassWizard在派生类中定义一个具有相同原型的函数，并且为它添加消息映射入口。关于如果使用ClassWizard重载消息处理函数的示例我们将会在本书后面的内容中遇到，这里就不再赘述。

在一些情况下，重载以后的处理函数应该在适当的地方调用基类的被

重载函数以使得基类和Windows可以处理这些消息。而在什么地方调用基类的被重载函数依环境而定。有时候我们需要根据一些条件来决定是否需要调用基类的被重载函数，而在另外的一些场合可能恰恰相反，我们需要基类的处理函数的返回值来决定是否或如何执行自己的处理函数代码。

- 注意：

- 有些时候我们可能会想到在将传递给消息处理函数的参数再传递给基类的处理函数时修改它们。比如说，我们有可能想到通过修改OnChar处理函数的nChar参数来在用户输入时屏蔽掉一些字符。但是这样做是不安全的，如果一定需要这样做，我们应该使用类CWnd的成员函数SendMessage，而不是直接修改传递给消息处理函数的参数。

按照惯例，我们在消息处理函数名的前面都使用了前缀“On”。并且消息处理函数可能带有几个参数，也可能一个参数也没有。一些消息处理函数可以返回值，而另一些可能被声明为void。可以在Class Library Reference中找到以WM\_开头的消息的默认处理函数，它们都是类CWnd的成员函数，并且具有前缀“On”。这些成员函数在类CWnd中的声明均带有前缀afx\_msg。

对于命令或控件通知消息，MFC并未提供默认的处理函数。因此，我们需要根据命名约定来自己命名或编写这些消息处理函数。当将命令或控件通知映射到处理函数时，ClassWizard根据命令ID或控件通知代码建议处理函数名。

举个例子来说，按照命名约定，响应File菜单下的Open命令的消息处理函数将被命名为

```
afx_msg void OnFileOpen();
```

对于一些很常见的用户界面元素，在框架中已为它们预定义了一些命令ID，比如与File菜单下的Open命令对应的命令ID为ID\_FILE\_OPEN，这些预定义ID可以在文件AFXRES.H中找到。下面是所支持的最重要的一些命令的列表：

- File菜单命令：New、Open、Close、Save、Save As、Page Setup、Print Setup、Print、Print Preview、Exit以及最近使用的文件
- Edit菜单命令：Clear、Clear All、Copy、Cut、Find、Paste、

Repeat、Replace、Select All、Undo以及 Redo.

- View菜单命令：Toolbar以及Status Bar.
- Window菜单命令：New、Arrange、Cascade、Tile Horizontal、Tile Vertical以及Split.
- Help菜单命令：Index、Using Help以及About.
- OLE命令(位于Edit菜单)：Insert New Object、Edit Links、Paste Link、Paste Special以及typename Object (谓词命令).

再举一个例子，按照命名约定的建议，对一个标签为Default的按钮控件的BN\_CLICKED通知消息处理函数将被命名为：

```
afx_msg void OnClickedDefault();
```

这样，我们就可以将IDC\_DEFAULT的ID与一个命令相关联，这样该命令等价于应用程序指定的用户界面对象。

以上讲到的两类消息(命令和控件通知消息)的处理函数都不带任何参数，同时也不返回任何值。

在上面的例子中，每一个处理函数都对应了单个命令ID或控件ID。然而，在MFC的消息映射机制中，我们还可能将单个处理函数对应一个命令ID或控件ID范围，但是，ClassWizard不支持命令ID或控件ID范围的处理函数的映射，因此我们必须手动的添加消息映射入口。由于到目前为止所讲述的内容还不足以提供了一个完整的示例，因此我们将在本书后面的内容中给出以上内容的示例程序。

在本节的最后解释一下宏DECLARE\_MESSAGE\_MAP。该宏一般出现的支持消息映射的类定义的尾部。前面已经说过，每一个从CCommandTarget派生的类都可以提供了一个消息映射来处理消息，这时我们需要在类声明中使用宏DECLARE\_MESSAGE\_MAP，然后在定义该类的成员函数的.CPP文件中使用宏BEGIN\_MESSAGE\_MAP，然后为每一个消息处理函数添加宏入口，最后使用宏END\_MESSAGE\_MAP。

- 注意：
- 如果你在DECLARE\_MESSAGE\_MAP之后声明了任何成员，必须为它们重新指定新的访问类型(public、private或protected)。为了避免出现错误，我们一般都在类声明的最底部使用宏DECLARE\_MESSAGE\_MAP。



## 第四节 对话框类

本节讲述应用程序DialogDemo的对话框类CDialogDemoDlg。由于DialogDemo是一个基于对话框的应用程序，因此该对话框也将是应用程序的主窗口。

下面我们先给出头文件DialogDemoDlg.h的清单：

```
// DialogDemoDlg.h : 头文件

//

#ifndef __AFX_DIALOGDEMODLG_H__7ABABF8A_0C8C_11D2_BC21_0000B4810A31__INCLUDED_
#define __AFX_DIALOGDEMODLG_H__7ABABF8A_0C8C_11D2_BC21_0000B4810A31__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

/////////////////////////////////////////////////////////////////

// CDialogDemoDlg 对话框

class CDialogDemoDlg : public CDialog
{
// 构造

public:
CDialogDemoDlg(CWnd* pParent = NULL); // 标准构造函数

// Dialog Data

//{{AFX_DATA(CDialogDemoDlg)
enum { IDD = IDD_DIALOGDEMO_DIALOG };
// 注意：ClassWizard 将在此添加数据成员

}}AFX_DATA

// 由 ClassWizard 生成的虚函数重载

//{{AFX_VIRTUAL(CDialogDemoDlg)
```

```

protected:

virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV 支持

//}}AFX_VIRTUAL

// 实现

protected:

HICON m_hIcon;

// 生成的消息映射函数

//{{AFX_MSG(CDialogDemoDlg)

virtual BOOL OnInitDialog();

afx_msg void OnSysCommand(UINT nID, LPARAM lParam);

afx_msg void OnPaint();

afx_msg HCURSOR OnQueryDragIcon();

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

//{{AFX_INSERT_LOCATION}}

// Microsoft Developer Studio 将在紧贴前一行前面添加附加的声明

#endif

// !defined(AFX_DIALOGDEMODLG_H__7ABABF8A_0C8C_11D2_BC21_0000B4810A31__INCLUDED_)

```

ClassWizard和AppWizard在生成代码时向源代码文件中插入了一些特定的格式化注释分隔符，以标明那些ClassWizard可以写入的地方。在前几节的内容中我们已经看到了一些这样的注释分隔符，如AFX\_MSG和AFX\_MSG\_MAP等。在头文件DialogDemoDlg中出现了一些新的分隔符，如AFX\_DATA用来标记在头文件中为对话框数据交换(dialog data exchange, DDX)所声明的成员变量的开始和结束。

AFX\_VIRTUAL用来标记在头文件中由ClassWizard生成和管理的虚函数的开始和结束。在.CPP文件中没有对应的块。

下一个看到的函数是DoDataExchange。框架调用该函数来交换和验证

对话框数据。该函数从来不直接调用，而总是在成员函数UpdateData中调用。成员函数UpdateData用户初始化对话框控件或从对话框中获取数据。

如果需要利用框架的自动数据交换和验证，则需要在从类CDialog派生应用程序特定的对话框时得载该成员函数。ClassWizard会为你创建一个该成员函数的重载版本，在该重载版本的DoDataExchange函数中包括了所期望的对话框数据交换(DDX)和有效性验证(DDV)的全局函数的“数据映射”。

DoDataExchange函数的重载版本将在后面的内容中给出和进行更为详细的分析。

成员变量m\_hIcon包括了应用程序主对话框的图标句柄。

下面介绍重载的几个消息映射函数。

第一个介绍的是OnInitDialog成员函数。该成员函数为消息WM\_INITDIALOG的处理函数。当成员函数Create、CreateIndirect或DoModal被调用时该消息被发送至对话框，但此时对话框尚未显示于屏幕上。

一般情况下我们重载该成员函数以进行一些对话框的初始化。在OnInitDialog成员函数的重载版本中，我们应该先调用基类的OnInitDialog成员函数，但无需理会其返回值。在正常情况下，OnInitDialog成员函数的重载版本返回TRUE。

不需要为该成员函数添加一个消息映射入口。这是因为Windows通过一个标准全局对话框函数来调用OnInitDialog函数，该对话框函数对所有的MFC对话框都是一样的，这种调用并不通过消息映射来完成。

表4.3 OnSysCommand成员函数的nID参数

参数nID的值	含义
SC_CLOSE	关闭CWnd对象
SC_HOTKEY	激活与应用程序指定的热键相关联的CWnd对象。 lParam参数的低位字节标识了待激活窗口的HWND。
SC_HSCROLL	垂直滚动
SC_KEYMENU	通过按键检索菜单

SC_MAXIMIZE (或SC_ZOOM)	最大化CWnd对象
SC_MINIMIZE (或SC_ICON)	最小化CWnd对象
SC_MOUSEMENU	通过鼠标单击检索菜单
SC_MOVE	移动CWnd对象
SC_NEXTWINDOW	移到下一窗口
SC_PREVWINDOW	移到上一窗口
SC_RESTORE	恢复窗口到通常位置和大小
SC_SCREENSAVE	执行由SYSTEM.INI文件的[boot]段指定的 屏幕保护程序
SC_SIZE	改变窗口的大小
SC_TASKLIST	运行或激活Windows任务管理程序 (Windows Task Manager)

- 注意：

- OnInitDialog成员函数的返回值将影响应用程序设置对话框中控件的输入焦点的方式。如果OnInitDialog返回非零值，窗口设置输入焦点为对话框中的第一个控件。如果已显式的将输入焦点设置为对话框中的某个控件，那么应该返回0。

接下来声明的是成员函数OnSysCommand，它是WM\_SYSCOMMAND消息的处理函数。当用户从控制菜单选择了某一个命令或单击了最大化或最小化按钮。

该函数的原型如下：

```
afx_msg void OnSysCommand( UINT nID, LPARAM lParam );
```

第一个参数nID指定了系统命令要求的类型，它可以为表4.3所示值之一。如果通过鼠标选择控制菜单命令，则参数lParam包含了鼠标的当前坐标。低位字代表x坐标，而高位字代表y坐标。

在默认情况下，OnSysCommand执行表4.3所示的预定义行为。

在WM\_SYSCOMMAND消息中，参数nID的低4位由Windows内部使用，当应用程序测试nID的值时，必须使用按位与(bitwise-AND)操作符组合值0xFFF0和nID来得到正确的结果。

应用程序可以在任何时候通过传递一个WM\_SYSCOMMAND消息给成员函数OnSysCommand来执行任何系统命令。

已被定义为用来选择控制菜单项的加速键(快捷键)消息将解释为OnSysCommand的调用，其它快捷键将翻译为WM\_COMMAND消息。

- 注意：
- 控制菜单项可以通过成员函数GetSystemMenu、AppendMenu、InsertMenu以及ModifyMenu来进行修改。修改了控制菜单的应用程序必须处理WM\_SYSCOMMAND消息，所有未被应用程序处理的消息必须传递给成员函数OnSysCommand。由应用程序添加的命令值必须由应用程序处理，而不能传递给函数OnSysCommand。
- 该成员函数由框架所调用，以允许应用程序处理Windows消息。传递给函数的消息代表了当获得消息时由框架获得的参数。如果你调用了基类的实现，该实现将使用消息最初传递的参数，而不是由函数提供的参数。

成员函数OnPaint是消息WM\_PAINT的处理函数，它当窗口客户区需要重绘时被调用。

成员函数OnQueryDragIcon对应于消息WM\_QUERYDRAGICON，它也是由框架在最小化窗口而没有为类定义一个图标时调用。系统通过对该函数的调用来获得当用户拖动被最小化的窗口时用来显示的光标。

如果应用程序返回了一个图标名光标的句柄，系统将其转换了黑白的。该句柄必须标识一个与显示驱动程序分辨率相兼容的单色光标或图标。应用程序可以调用CWinApp::LoadCursor或CWinApp::LoadIcon成员函数来从它的可执行文件中的资源内加载一个光标或图标并获得其句柄。

下面给出类CDialogDemoDlg的实现文件DialogDemoDlg.cpp的清单：

```
// DialogDemoDlg.cpp : 实现文件
```

```
//
```

```
#include "stdafx.h"
```

```

#include "DialogDemo.h"

#include "DialogDemoDlg.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CAboutDlg 用于 App About 的对话框

class CAboutDlg : public CDialog

{

public:

CAboutDlg();

// Dialog Data

//{{AFX_DATA(CAboutDlg)

enum { IDD = IDD_ABOUTBOX };

//}}AFX_DATA

// 由 ClassWizard 生成的虚函数重载

//{{AFX_VIRTUAL(CAboutDlg)

protected:

virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV 支持

//}}AFX_VIRTUAL

// 实现

protected:

//{{AFX_MSG(CAboutDlg)

//}}AFX_MSG

```

```

DECLARE_MESSAGE_MAP()

};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
//{{AFX_DATA_INIT(CAboutDlg)

//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)

//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)

// 没有消息处理函数

//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////

// CDialogDemoDlg 对话框

CDialogDemoDlg::CDialogDemoDlg(CWnd* pParent /*=NULL*/)
: CDialog(CDialogDemoDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CDialogDemoDlg)

// 注意：ClassWizard 将在此添加成员初始化

//}}AFX_DATA_INIT

```

```

// 注意在 Win32 中 LoadIcon 不需要一个相应的 DestroyIcon

m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

}

void CDialogDemoDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDialogDemoDlg)
    // 注意: ClassWizard 将在此添加 DDX 和 DDV 调用
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDialogDemoDlg, CDialog)
    //{{AFX_MSG_MAP(CDialogDemoDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////

// CDialogDemoDlg 消息处理函数

BOOL CDialogDemoDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // 添加 "About..." 菜单项到系统菜单

    // IDM_ABOUTBOX 必须在系统命令范围之内

    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);

    ASSERT(IDM_ABOUTBOX < 0xF000);

```



```

CMenu* pSysMenu = GetSystemMenu(FALSE);

if (pSysMenu != NULL)
{
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
}

// 该当前对话框设置图标。当应用程序主窗口不为对话框时由框架自动完成该任务
SetIcon(m_hIcon, TRUE); // 设置大图标
SetIcon(m_hIcon, FALSE); // 设置小图标

// TODO: 在此添加额外初始化

return TRUE; // 除非将焦点设置为某一控件，否则返回 TRUE
}

void CDialogDemoDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFFF) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {

```

```
CDialog::OnSysCommand(nID, lParam);
```

```
}
```

```
}
```

// 如果你向对话框添加了一个最小化按钮，你将需要下面的代码来绘制图标。

// 对于使用文件/视模型的 MFC 应用程序，这一步由框架自动完成。

```
void CDialogDemoDlg::OnPaint()
```

```
{
```

```
if (IsIconic())
```

```
{
```

```
CPaintDC dc(this); // 重绘设备上下文
```

```
SendMessage(WM_ICONERASEBKGND, (LPARAM) dc.GetSafeHdc(), 0);
```

// 在窗户区矩形中居中图标

```
int cxIcon = GetSystemMetrics(SM_CXICON);
```

```
int cyIcon = GetSystemMetrics(SM_CYICON);
```

```
CRect rect;
```

```
GetClientRect(&rect);
```

```
int x = (rect.Width() - cxIcon + 1) / 2;
```

```
int y = (rect.Height() - cyIcon + 1) / 2;
```

// 绘制图标

```
dc.DrawIcon(x, y, m_hIcon);
```

```
}
```

```
else
```

```
{
```

```
CDialog::OnPaint();
```

```
}
```

```
}
```

// 在用户拖动被最小化的窗口时系统调用该函数来获得用于显示的光标。

```
HCURSOR CDialogDemoDlg::OnQueryDragIcon()  
{  
    return (HCURSOR) m_hIcon;  
}
```

在实现文件DialogDemoDlg.cpp中包括了类CAboutDlg的定义，该类封装了应用程序的“关于”对话框。由于类CAboutDlg与类CDialogDemoDlg相比要简单得多，所以我们在此不再讲述。下面我们主要分几个部分来讲述类CDialogDemoDlg。

### (1) 构造函数

在类CDialogDemoDlg的构造函数中，注释分隔符AFX\_DATA\_INIT用于在对话框类的构造函数中标记对话框数据交换(DDX)成员变量初始化的开始和结束。接着使用MFC全局函数AfxGetApp获得指向应用程序对象的指针，再通过该指针调用成员函数LoadIcon加载图标，并将其句柄放入成员变量m\_hIcon。

类CDialogDemoDlg的成员函数DoDataExchange只是简单的调用基类的对应成员函数。在成员函数DoDataExchange内，注释分隔符AFX\_DATA\_MAP用于标记对话框数据交换函数调用的开始和结束。

- 注意：
- 尽管在接下来的一章中，我们将要再次对菜单的应用作系统的介绍，但考虑到程序说明的完整性，在这里，我们仍然先对其作简明的介绍。

### (2) 消息处理成员函数OnInitDialog

消息处理函数OnInitDialog首先调用基类的对应成员函数。接着的两个ASSERT宏断言IDM\_ABOUTBOX处于系统命令范围内。ASSERT宏以一个布尔表达式为参数，然后对该表达式求值，如果结果为0（假），ASSERT输出一个诊断信息，然后中断程序，如果结果为非0，宏ASSERT什么也不做。

由宏ASSERT输入的诊断信息具下面的格式：

assertion failed in file <文件名> in line <行号>

在这里文件名代表了出现ASSERT错的源代码文件，行号指出了源代码文件中出错的行。

在MFC的发行版本(Release version)中，宏ASSERT并不对表达式进行求值，从而也不会中断程序的执行。与宏ASSERT不同，宏VERIFY在MFC中调试版本和发行版本中都将对表达式进行求值。

接下来，由类CWnd的成员函数GetSystemMenu返回控制菜单的一个拷贝。如果传递给该函数的参数为假，函数GetSystemMenu返回正在使用的控制菜单的一个拷贝的句柄。该拷贝同控制菜单完全相同，但它可以修改。如果传递的参数为真，函数GetSystemMenu重置控制菜单为默认值，如果原有控制菜单经过修改，该改动将丢失。在这种情况下，函数GetSystemMenu的返回值是未定义的。可以通过由成员函数GetSystemMenu返回的指针使用CMenu::AppendMenu、CMenu::InsertMenu或CMenu::ModifyMenu函数来改变控制菜单。

控制菜单最初包括由不同ID值，如SC\_CLOSE、SC\_MOVE和SC\_SIZE等标识的项。这些控制菜单中的项产生WM\_SYSCOMMAND消息。所有预定义的控制菜单具有大于0xF000的ID值，因此，如果应用程序向控制菜单中添加了新的项，这些项所使用的ID值应该小于0xF000。

Windows可以自动的决定是否使标准控制菜单中的项变灰。类CWnd可以通过响应WM\_INITMENU来执行检查或变灰，该消息在所有菜单显示之前发送。

表4. 4 成员函数AppendMenu的nFlags参数

值	含义
MF_CHECKED	在菜单项前面放置一个默认的选中标记。当应用程序提供了选中标记位图时，则使用表示选中状态的位图。
MF_UNCHECKED	清除菜单项前面的选中标记。如果应用程序提供了选中标记位图，则使用表示不选中状态的位图。
MF_DISABLED	禁止某项被选择，但不使其变灰。
MF_ENABLED	允许某项被选择。
MF_GRAYED	禁止某项被选择，并使其变灰。

MF_MENUBARBREAK	对于静态菜单，将项放到新的一行，对于弹出菜单，将项放到新的一列。新的弹出菜单列和旧的列使用垂直分隔线隔开。
MF_MENUBREAK	将项放到新的一行，或弹出菜单的新的一列。列之间没有任何分隔线。
MF_OWNERDRAW	指定项为一个自绘制项。当菜单第一次显示时，拥有该菜单的窗口将收到消息WM_MEASUREITEM，该消息包括了菜单项的高度和宽度。而每次当需要更新菜单项的显示外观时，将发送消息。该选对顶层菜单项无效。
WM_POPUP	指定该菜单项具有一个相关联的弹出菜单。ID参数指定与该项相关的弹出菜单句柄。该标志用来添加顶层弹出菜单或将一弹出菜单添加到另一弹出菜单。
WM_SEPARATOR	绘制一个水平分隔线。仅用于弹出式菜单。该行不可以变灰，不可以被禁止，也不可以被加亮，并且其它参数被忽略。
MF_STRING	指定菜单项为一字符串。

如果由函数GetSystemMenu返回的指针不为空，则通过一个CString对象strAboutMenu从资源文件中加载字符串IDS\_ABOUTBOX。如果成功的话(通过检验strAboutMenu是否为空来判断)，将该菜单项添加到控制菜单中。

添加菜单项使用类CMenu的成员函数AppendMenu。该函数具有如下的原型：

```
BOOL AppendMenu( UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL );
```

```
BOOL AppendMenu( UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp );
```

第一个参数nFlags指定了在新添加菜单时关于新增菜单项的状态的信息。它可以为表4.4所列的一个或多个值。需要注意的事是表4.4中所列的某些标志是互斥的，也就是说这其中的一些标志不可以同时使用。关于这方面的详细信息请参考Visual C++中关于成员函数AppendMenu的联机文档。

- 注意：
- 在程序设计中（尤其是在涉及到图形编程时！），图形化的菜单是不是对用户具有更大的吸引力？要达到这一点，在菜单编程中使用位图参数就大功告成了！是不是相当简单？

如果nFlags参数被设置为MF\_POPUP，参数nIDNewItem为一个弹出菜单的句柄；如果nFlags参数被设置为MF\_SEPARATOR，该参数被忽略；对于其它情况，它为新增菜单项的命令ID。

随着参数nFlags的不同，参数lpszNewItem具有不同的解释，如表4.5所示。

表4.5 nFlags参数不同时对lpszNewItem参数的不同解释

NFlags的值	对lpszNewItem的解释
MF_OWNERDRAW	包括一个由应用程序提供的32位值，应用程序使用该值来维护与该菜单项相关联的附加数据。该32位值在应用程序处理消息WM_MEASUREITEM和WM_DRAWITEM时可用，它被保存在这些消息提供的结构的itemData成员中。
MF_STRING	包括指向以null结束的字符串的指针。
MF_SEPARATOR	lpszNewItem参数被忽略。

在第二种格式的成员函数AppendMenu中，参数pBmp指向一个CBitmap对象，该对象将被用作菜单项。

函数SetIcon用来将句柄设置为某一特定图标，该图标由第一个参数标识。第二个参数指定和图标的大小，如果该参数为真，表示图标为32像素× 32像素大小；如果为假，表示图标为16像素× 16像素大小。

最后，由于我们没有将输入焦点设置为某一个特定的控件，因此消息处理函数OnInitDialog返回真值TRUE。

### (3) 消息处理成员函数OnSysCommand

类CDialogDemoDlg的OnSysCommand成员函数非常之简单，首先它检查用户选择的命令是否IDM\_ABOUTBOX（注意在if语句中使用了表达式nID & 0xFFF0），如果是的话，声明一个类CAboutDlg的对象，调用该

对象的DoModal成员函数以模态方式显示该对话框；否则，函数OnSysCommand调用基类的成员函数OnSysCommand。前面已经提到过，对于应用程序自己添加的控制菜单项(如这里的IDM\_ABOUTBOX)，不要调用基类提供的成员函数OnSysCommand进行默认处理。

#### (4) 消息处理成员函数OnPaint

首先，类CWnd的成员函数IsIconic用来判断一个窗口是否被最小化。如果是，函数IsIconic返回真值(非零值)；反之，返回假值(零)。如果IsIconic返回真，则以当前this指针(它指向当前CDialogDemoDlg对象)为参数构造类CPaintDC的对象dc。类CPaintDC封装了Windows应用程序重绘时所使用的设备描述表。然后，成员函数SendMessage向窗口发送一条WM\_ICONERASEBKGND消息，该条消息表示在重画被最小化的窗口的图标之前需要对图标背景进行填充。而类CDC的成员函数GetSafeHdc返回该类的成员函数m\_hDC，即相应的输出设备上下文，该设备上下文作为WM\_ICONERASEBKGND的wParam参数被发送。

接着成员函数OnPaint分别以SM\_CXICON和SM\_CYICON有参数调用Win32 API函数GetSystemMetrics (注意它并不是类CDialogDemoDlg及其基类，典型的如CWnd的成员函数)，从而得到以像素为单位的图标默认宽度和高度，图标大小的典型值为32× 32，但我们不可以在应用程序中作此假设，因为它依赖于所安装的显示硬件，并可能随用户对系统设置的改变而改变。

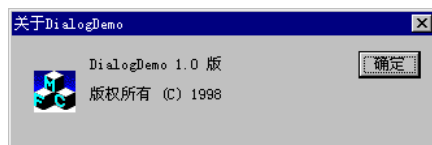
然后类CWnd的成员函数GetClientRect将当前客户区矩形的度量放入第一个参数所指向的CRect对象中。类CRect封装了由左上角和右下角所确定的一个矩形。其成员函数Width和Height分别返回所确定矩形的宽和高。然后通过计算得到使图标居中的坐标。最后调用类CDC的成员函数DrawIcon绘制由m\_hIcon所标识的图标。

如果当前窗口并未被最小化，类CDialogDemoDlg的OnPaint成员函数调用基类提供的相应成员函数。

#### (5) 消息处理函数OnQueryDragIcon

消息处理函数OnQueryDragIcon只是简单的返回句柄m\_hIcon，并将其类型强制转换为HCURSOR。

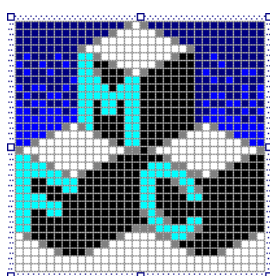
程序所使用的资源如图4.9所示。



a. 类CAboutDlg所对应的对话框



b. 类CDialogDemoDlg所对应的对话框



c. 对话框CDialogDemoDlg所用的图标

图4. 9 在应用程序DialogDemo中所用到的资源

现在即可编译并运行应用程序DialogDemo，其结果如图4.10。通过单击控制菜单中的项“关于DialogDemo”，可以打开如图4.9中a所示的对话框。而无论单击“确定”还是“取消”，应用程序DialogDemo都将被关闭。

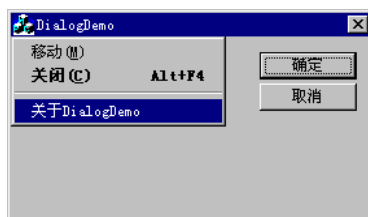


图4. 10 应用程序DialogDemo的运行结果

## 第五节 小结

本章详细的分析了由AppWizard生成的一个基于对话框的应用程序，这是进行下一步对控件的学习的基础。同时，我们还借用该程序说明了使用MFC编写的应用程序的结构，涉及到了以下的一些内容：

- 从CWinApp派生的应用程序类



- MFC应用程序的消息映射机制和方法
- 从CDialog派生的对话框类(这里CDialog又从CWnd派生)

我们在这些内容上花费了大量的篇幅，但就算是这样还是未能将其完全的阐述清楚。它们是一些比较复杂，但并不神秘的东西，然而一旦理解了这些东西，那么你对MFC的理解立刻就会上升一个层次，从而在实际编程的过程中获益更多。

在本章，我们对代码进行几乎是逐行逐字的讲解和说明，这是考虑到初学者对于MFC的编程方式比较陌生的情况。在后面几章的内容中，我们对于一些关系不是很紧密的内容不再作详细的讲述，你需要随时查阅Visual C++的联机文档，好在里面有数不清的资源可以利用。但是应该记住一点，我们建议你弄懂示例程序中的每一行所完成或实现的功能，只有这样，才能充分利用这些经过实践证明是可行的编程方法和技巧，从而在比较短的时期内很快的提高自己的编程水平和能力。

## 第五章 响应用户命令

我们很容易想见，在Windows 95程序设计中，一个很重要的方面就是对各种消息的响应。而在这其中，各种输入命令的响应又几乎占据着最重要的地位。就象我们在前面的章节中所介绍的那样，窗口可以说是Microsoft Windows的最重要的用户界面对象，而第二重要的用户界面对象就得算是菜单了，用户选择可用命令的一个最常用，也是最重要的手段就是菜单。这一点，通过查看Windows API中菜单所支持的庞大的功能也可以更直观的看出。当然，我们也可以发现，在一些情况下，我们可以有更快捷，有时也更有效的命令输入方式，就是使用加速键或者工具条。当然，出于对界面介绍的完整性考虑，我们在本章的末尾，也会简单的介绍滑块控件，进度条以及一些上下控件的基础知识。

即使是Microsoft Windows的临时用户也知道，在应用程序的主窗口的顶部会出现一个菜单条，在Windows 95中，Microsoft还提示使用对象的上下文菜单。用户所需要做的，简单到只需要轻轻单击一下鼠标右键就可以了。（在后面的程序设计中，我们会看到，要是你觉得使用双击右键更有意思的话，作出这种改变几乎没有什麼更多的工作。）同时，我们也会向你展示实现图符菜单的简单方法。在该节的最后，我们还会介绍如何对系统菜单进行操作。

加速器是这样的一个按键，程序负责将这一按键解释成一个命令，从用户程序的角度看，在菜单选择与加速器按键的选择并没有什麼差别，这是通过Windows为二者生成相同的消息决定的。对于程序员来说，将加速键显示在相应菜单的右侧是一个良好的习惯，它为熟练用户提供了进行更方便选择提供了一种可能性。但是从程序设计的角度看二者却是分别定义的---菜单是用菜单资源定义的，而加速键是用加速键资源定义的。

工具条是第三种命令输入机制。工具条是一个带有按钮的窗口，它使用户发出命令的动作节省到在相应按钮上按一下鼠标即可。当然，由于工具条本身也要占据屏幕空间，因此，放在工具条上的命令应该是最常用的。而且，出于对用户的尊重，在用户不希望使用它的时候，应该能将工具条隐去。AppWizard自动创建的工具条提供这种能力，但我们会对此作出更详细的解释。

在本章的最后，我们会对一些在Windows下常用的控件，比如滑块控件，进度条以及上下控件的一些基础知识。总得来说，我们准备在本章中介绍如下知识：

- 菜单消息响应
- 快捷键消息响应
- 工具条消息响应
- 对上下控件、进度条、以及轨道消息响应

## 第一节 菜单消息响应

在程序接口中，我们最常用的选择方式就是用菜单进行选择。而对于从程序员来说，我们所要做的很大一部分工作，就是对程序的输入进行响应。毋庸置疑，菜单可以有多种实现的方式。从用户的使用角度来说，当然希望可以有多种多样的选择。我们常常建立下拉菜单，也可以建立一些必要的图符菜单，当然在必要的时候，我们业应该建立一种使用上更便捷的上下文菜单。而在很多情况下，我们希望对菜单进行动态的操作 在许多时候，这甚至是一种很重要的事情，对在我们下面的章节中也会有所提及。

从下面开始，我们将按以下的顺序对菜单消息的响应进行讲解。

让我们先看看怎样在资源编辑器中实现菜单。

首先新建一个基于单文档界面的程序，这只需要在AppWizard新建程序的第一步(如图5.1)中在Application Type选择时选择Single document即可。

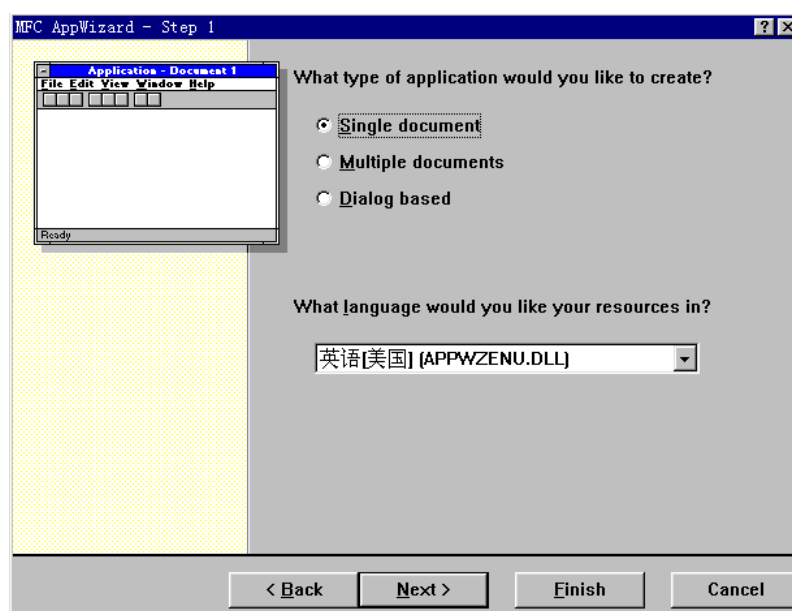


图5. 1 建立单文档界面程序：Step1

我们设该应用程序的名称为Menu.同时，由于是首次建立不是基于对话框的程序，我们简单的说一下建立的过程。(基于对话框的程序AppWizard只需要四步，而基于文档的程序需要六步。)

在第二步(如图5.2所示)中，我们将设定数据库支持，由于我们现在建立的只是简单的单文档程序，我们选择不需要任何数据库支持。

在第三步(如图5.3所示)中，我们将设定生成的标准程序中的文档支持。AppWizard提供有容器类(Container)与服务类(Server)应用，我们保持缺省设置。(不需要容器类与服务类支持，但保留ActiveX Control控件支持。)

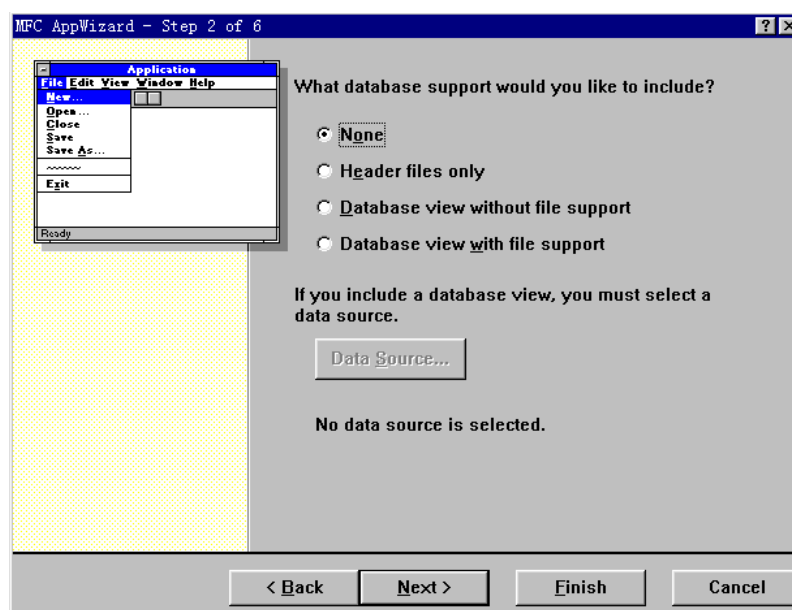


图5. 2 建立单文档界面程序Step 2

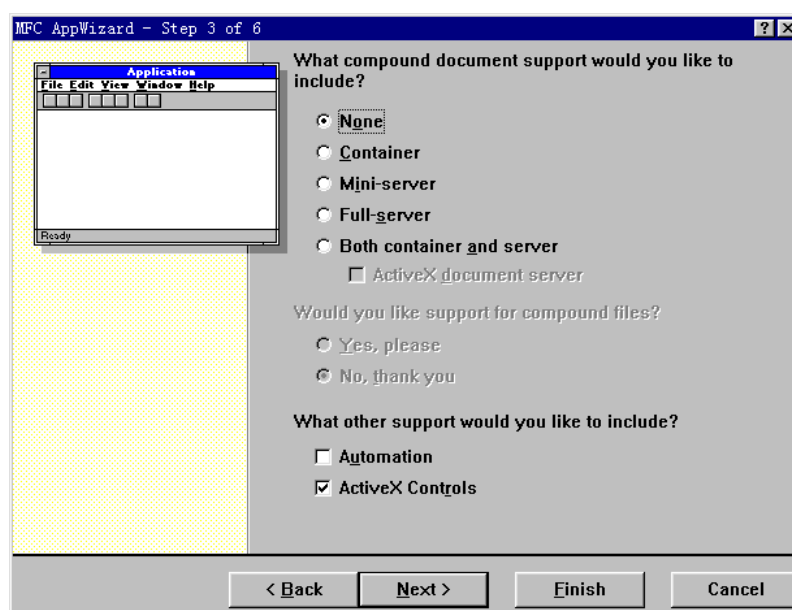


图5. 3 建立单文档界面程序Step 3

在第四步(如图5.4)中,我们去除掉打印预览及打印支持。但我们保持工具条,状态条,三维控制支持。对刚使用的文件列表数设置为四。

在第五步(如图5.5)中,我们将设定是否设置提示及怎样使用MFC库。在刚开始时,我们设定需要提示,同时,动态链接MFC库。

在最后一步中(如图5.6)中,我们将设定AppWizard将要为我们生成的类。

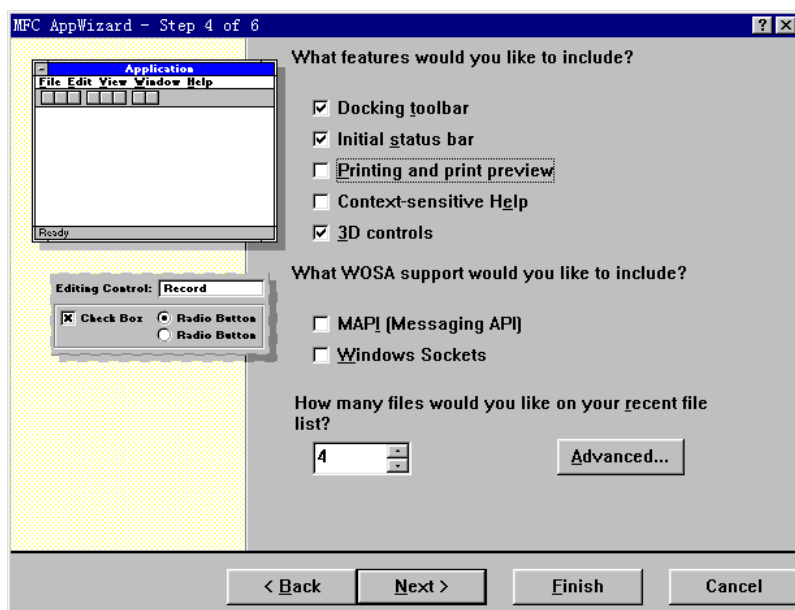


图5. 4 建立单文档界面程序Step 4

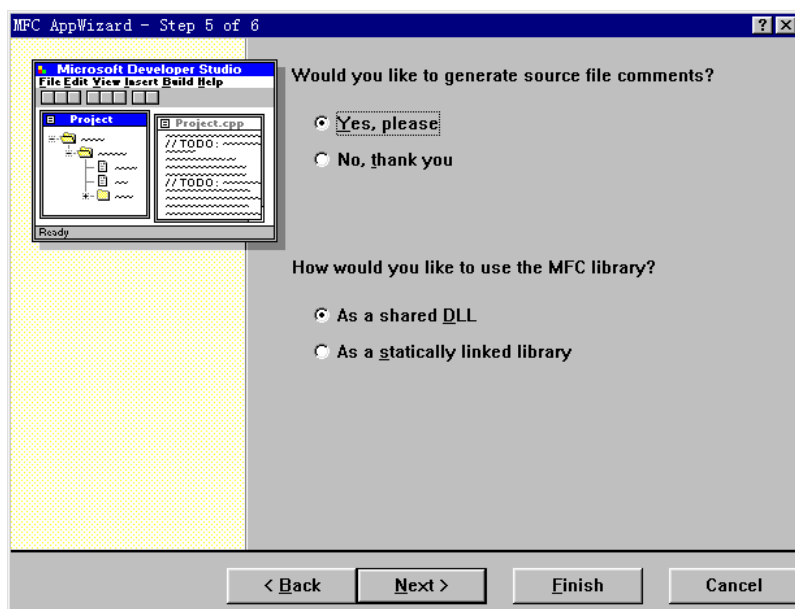


图5. 5 建立单文档界面程序Step 5

这样,我们就建立起一个具有Document /View结构的简单的应用程序。

在进行进一步的编程以前，我们希望你能对照在第四章中对基本框架的解释，对产生的其它文件(MenuDoc和MenuView)我们将在第六章中作详细的解释，现在读者要是理解得不太清楚，可以先跳过去这一部分。在作进一步的编程之前，建议读者先将AppWizard生成的程序编译，运行，看看AppWizard都为我们实现了什麼功能。

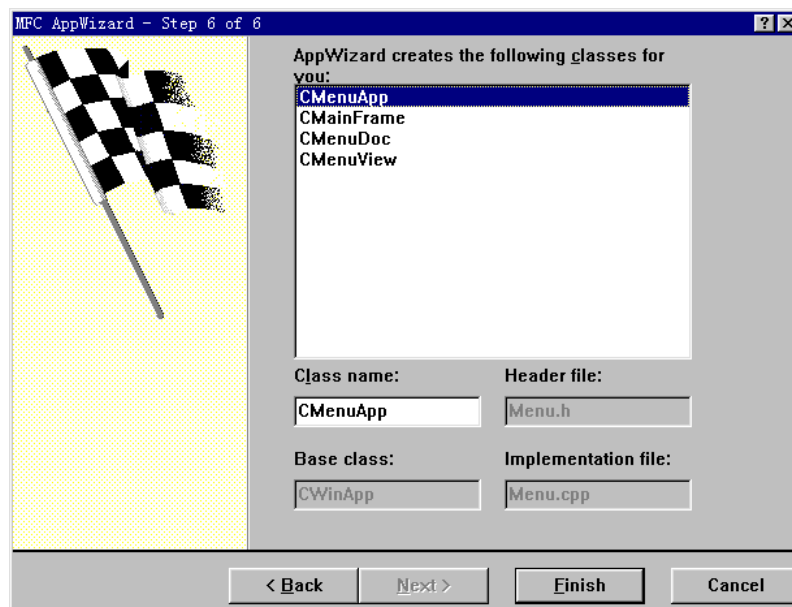


图5. 6 建立单文档界面程序Step 6

下面我们对程序的菜单作一些修改。

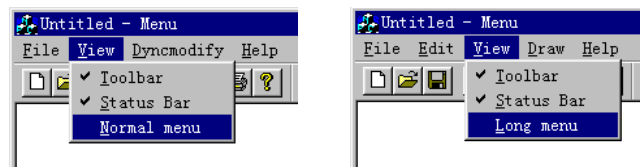


图5. 7 菜单资源IDR\_MAINFRAME0

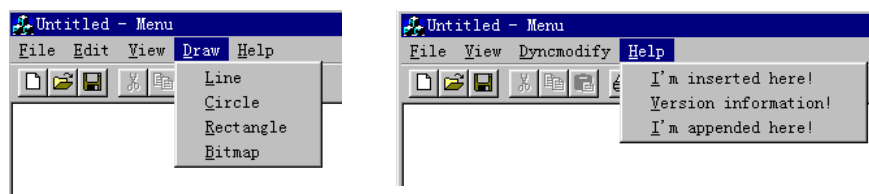
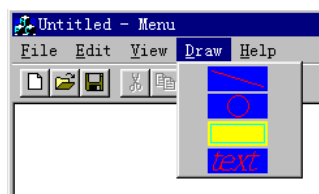


图5. 8 菜单资源IDR\_MAINFRAME1



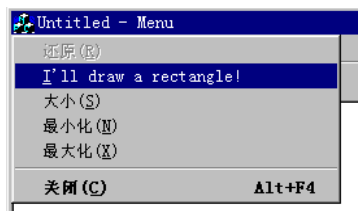


图5. 9 系统菜单

如图5.7所示,为我们的程序运行菜单的一个画面.该菜单的属性为 IDR\_MAINFRAME0

而下面的两个图片为我们的应用程序运行时的另外一组菜单.该菜单的属性为IDR\_MAINFRAME1.

在程序中,我们的改变了Help和Draw菜单,如图5.8所示.

最后,我们改变了系统菜单,并改变了其中的一个选项的功能.如图5.9所示.

作为补充,下面我们看一下程序所实现的绘图以及系统菜单被改变后实现的功能(如图5.10所示)。

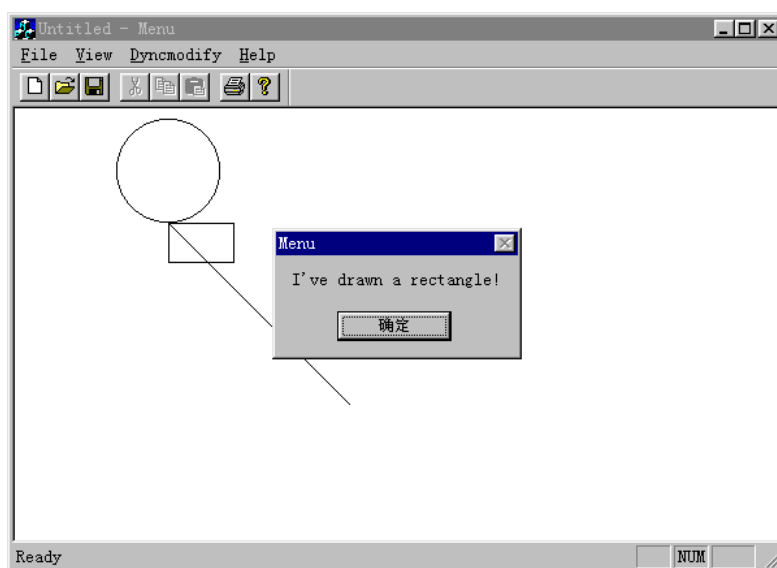


图5. 10 程序的一个运行画面（系统菜单中新增菜单被选择）

我们认为,通过第四章中基于对话框的程序以后,读者应该已经掌握了实现菜单的修改的方法,但为了下面描述的方便,我们在此列出各菜单项的属性.

- 注意:

- 下面的代码摘自菜单完成后的文件Menu.rc中(该文件以文本形式保存,可以用写字板或者记事板打开.为了更清楚地理解下面的代码,我们粗略的讲解一下菜单资源的语法:

关键字POPUP表明其后的BEGIN与END之间的为其弹出式子菜单的内容。

关键字MENUITEM标识一具体的子菜单内容。MENUITEM后所跟的第一部分为 菜单标题,其最后一部分则为标识该菜单项的ID号。需要注意的是,具有子菜单的弹出式菜单项是没有标识的ID号的。

- 我们程序的最初的菜单,并不包含程序执行后结果的一些菜单,最初的菜单中仅仅包含前四幅图中的菜单.

- IDR\_MAINFRAME0 MENU PRELOAD DISCARDABLE
- BEGIN
- POPUP "&File"
- BEGIN
- MENUITEM "&New\tCtrl+N", ID\_FILE\_NEW
- MENUITEM "&Open...\tCtrl+O", ID\_FILE\_OPEN
- MENUITEM "&Save\tCtrl+S", ID\_FILE\_SAVE
- MENUITEM "Save &As...", ID\_FILE\_SAVE\_AS
- MENUITEM SEPARATOR
- MENUITEM "&Print...\tCtrl+P", ID\_FILE\_PRINT
- MENUITEM "Print Pre&view", ID\_FILE\_PRINT\_PREVIEW
- MENUITEM "P&rint Setup...", ID\_FILE\_PRINT\_SETUP
- MENUITEM SEPARATOR
- MENUITEM "Recent File", ID\_FILE\_MRU\_FILE1, GRAYED
- MENUITEM SEPARATOR



- MENUITEM "E&xit", ID\_APP\_EXIT
- END
- POPUP "&Edit"
- BEGIN
- MENUITEM "&Undo\tCtrl+Z", ID\_EDIT\_UNDO
- MENUITEM SEPARATOR
- MENUITEM "Cu&t\tCtrl+X", ID\_EDIT\_CUT
- MENUITEM "&Copy\tCtrl+C", ID\_EDIT\_COPY
- MENUITEM "&Paste\tCtrl+V", ID\_EDIT\_PASTE
- END
- POPUP "&View"
- BEGIN
- MENUITEM "&Toolbar", ID\_VIEW\_TOOLBAR
- MENUITEM "&Status Bar", ID\_VIEW\_STATUS\_BAR
- MENUITEM "&Long menu", ID\_VIEW\_LONGMENU
- END
- POPUP "&Draw"
- BEGIN
- MENUITEM "&Line", ID\_DRAW\_LINE
- MENUITEM "&Circle", ID\_DRAW\_CIRCLE
- MENUITEM "&Rectangle", ID\_DRAW\_RECTANGLE
- MENUITEM "&Bitmap", ID\_CHANGE
- END

- POPUP "&Help"
- BEGIN
- MENUITEM "&About Menu...", ID\_APP\_ABOUT
- END
- END
- 
- IDR\_MAINFRAME1 MENU DISCARDABLE
- BEGIN
- POPUP "&File"
- BEGIN
- MENUITEM "&New\tCtrl+N", ID\_FILE\_NEW
- MENUITEM "&Open...\tCtrl+O", ID\_FILE\_OPEN
- MENUITEM "&Save\tCtrl+S", ID\_FILE\_SAVE
- MENUITEM "Save &As...", ID\_FILE\_SAVE\_AS
- MENUITEM SEPARATOR
- MENUITEM "&Print...\tCtrl+P", ID\_FILE\_PRINT
- MENUITEM "Print Pre&view", ID\_FILE\_PRINT\_PREVIEW
- MENUITEM "P&rint Setup...", ID\_FILE\_PRINT\_SETUP
- MENUITEM SEPARATOR
- MENUITEM "Recent File", ID\_FILE\_MRU\_FILE1, GRAYED
- MENUITEM SEPARATOR
- MENUITEM "E&xit", ID\_APP\_EXIT
- END

- POPUP "&View"
- BEGIN
- MENUITEM "&Toolbar", ID\_VIEW\_TOOLBAR
- MENUITEM "&Status Bar", ID\_VIEW\_STATUS\_BAR
- MENUITEM "&Normal menu", ID\_VIEW\_NORMALMENU
- END
- POPUP "&Dyncmodify"
- BEGIN
- MENUITEM "&Insert a menuitem", ID\_DYNCMODIFY\_INSERT
- MENUITEM "&Delete a menuitem", ID\_DYNCMODIFY\_DELETE
- MENUITEM "&Append a menuitem", ID\_DYNCMODIFY\_APPEND
- MENUITEM "&Modify help", ID\_DYNCMODIFY\_MODIFY
- MENUITEM "M&odify system menu", ID\_DYNCMODIFY\_MODIFYSYSTEMMENU
- MENUITEM "&Reset system menu", ID\_DYNCMODIFY\_RESET
- END
- POPUP "&Help"
- BEGIN
- MENUITEM "&About Menu...", ID\_APP\_ABOUT
- END
- END
- 
- IDR\_CONTEXTMENU MENU DISCARDABLE
- BEGIN

- POPUP "dummy"
- BEGIN
- MENUITEM "&Line", ID\_DRAW\_LINE
- MENUITEM "&Circle", ID\_DRAW\_CIRCLE
- MENUITEM "&Rectangle", ID\_DRAW\_RECTANGLE
- END
- END

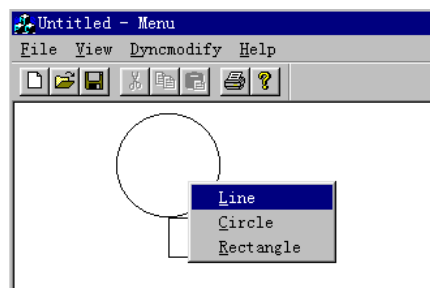


图5. 11 上下文菜单的一个运行画面

为了建立弹出式菜单，在上述代码中，我们发现还有一个菜单 IDR\_CONTEXTMENU没有提及。它在该文件中与其它菜单的表现形式是一样的。下图为弹出式菜单的一个运行画面。

下面我们看看怎样改变菜单项的缺省名称。如图5.12所示，在 WorkSpace中于 IDR - MAINFRAME上右击鼠标，在弹出式菜单中选择属性项（Property），在接下来的属性设置对话框中在ID设置区输入后者从下拉列表选择所需的菜单ID号即可。依此处理其它各菜单项。

- 注意：
- 菜单项ID号的设置务请同我们给出的保持一致，由于ID号是辨识菜单消息的唯一手段，此处的差别将会导致你在下面的学习过程中将不得不跟我们给出的代码有一些差异。

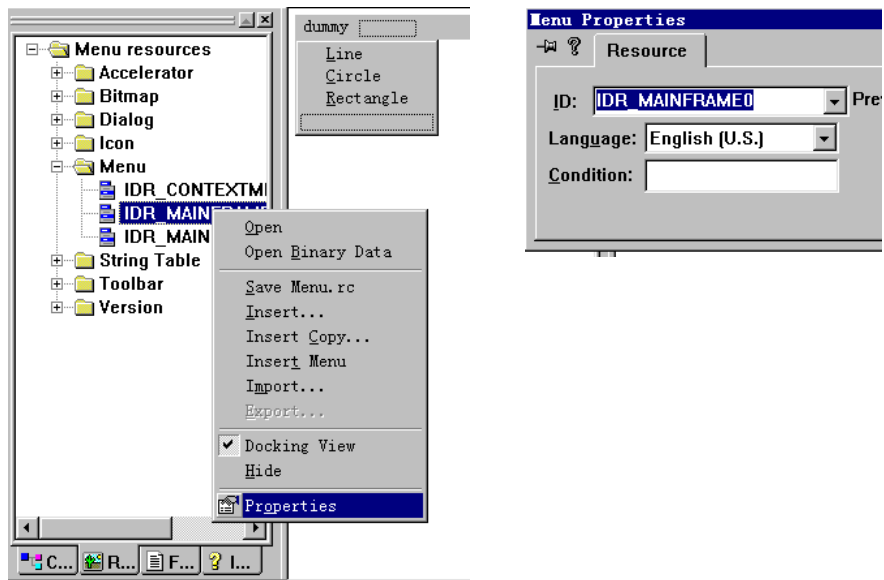


图5. 12 改变菜单项ID值

现在，试着运行一下我们的程序，会发现它并没有能比最初的AppWizard建立的标准程序多做些什么。这是必然的，毕竟，我们还没有编写一行让它动起来的代码吗！不过不要紧，我们接下来所要做的，就是这项工作。（不用担心IDR\_MAINFRAME0对IDR\_MAINFRAME的替代会有什么不良结果，实际上，ClassWizard监测到这些，并已在程序代码中作了相应的变动以反应这一变化。）

首先，由于对菜单的动态改变在菜单IDR\_MAINFRAME1的菜单项中实现，因此，我们所要做的第一步工作就是实现菜单IDR\_MAINFRAME1对菜单IDR\_MAINFRAME0的动态替换。这可以用CWnd类的一个成员函数SetMenu实现。函数SetMenu的原型为：

```
BOOL SetMenu(CMenu * pMenu);
```

其中pMenu为一指向欲替换原有菜单的一CMenu类的对象的指针，函数的返回值仅仅在菜单没有发生改变时为0。

- 注意：
- 如果你以空值NULL作为指向CMenu的指针，那你的程序的菜单就算消失了。除非在程序中以适当方式告诉用户重新得到菜单的方法，用户很难再得到菜单了。不过，在一些情形下，实现菜单的消隐/显示可以实现对屏幕显示空间的有效管理。

尽管通过第四章的学习，你已经掌握了一些在程序中添加消息响应函数的方法，但基于文档/视结构的成员函数的添加还是有自己的很大程度上的不同。在我们已经生成的程序中，添加对菜单消息的响应的

函数是有一些自己的特殊性的。下面我们结合图示顺序讲解。

1. 在菜单项上或源代码编辑区中右击鼠标，或选择View/ClassWizard...菜单项，打开ClassWizard对话框。
2. 在Class name:选项中，确认为CMainFrame，在Object IDs:中选择ID\_VIEW\_LONGMENU,在Message:选项中选择COMMAND，如图5.13。

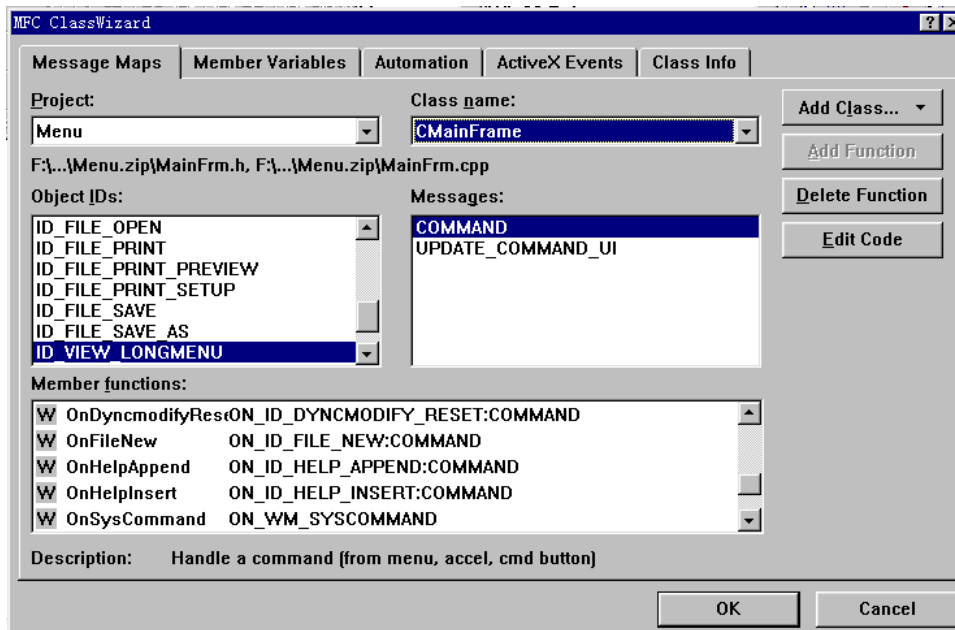


图5. 13 为菜单项增加消息响应函数

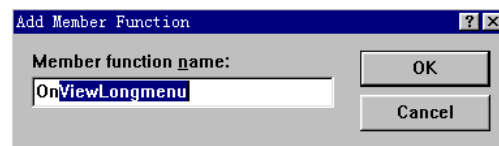


图5. 14 选择响应函数名称

3. 单击Add Class...为响应函数命名（如图5.14所示），一般说，接受系统提供的缺省名就足够了。
4. 单击OK后进入源代码编辑区进行编辑。

如下所示，为我们为函数OnViewLongmenu所加的代码：

```
void CMainFrame::OnViewLongmenu()
{
    // TODO: Add your command handler code here

    SetMenu(&hLongMenu);
}
```

```
}
```

理解这段代码并不困难，但参数hLongMenu从何而来？这是我们在程序中创建的一个标识菜单IDR\_MAINFRAME1的CMenu类的一个对象。在文件MainFrm.cpp的下示函数中我们对其进行了初始化(这两处代码均是手工加入的)：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // ...

    hLongMenu.LoadMenu(IDR_MAINFRAME1);

    // ...

}
```

而其定义在头文件MainFrm.h中：

```
class CMainFrame : public CFrameWnd
{
    // ...

    CMenu hLongMenu;

    // ...

}
```

对View菜单的前三个菜单项的响应涉及到GDI绘图的一些知识，读者如果暂时弄不清楚，可以留待看过相应章节后再回头来理解。下面我们看一看对菜单的动态操作。

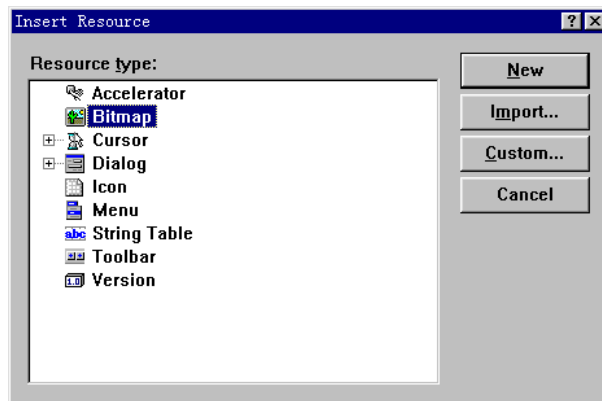


图5. 15 在程序中增加位图资源

在前面的第四章的那个例程中，我们对这类实现作了些简要的介绍，在这里，我们结合本章的程序作一些更深入的介绍。

首先，对菜单的动态操作的几个函数，其最后一个参数都可以用一个位图指针替代。本章例程中Draw菜单下的图符菜单即由此实现。当然，要使用位图作为菜单项，首先必须创建位图资源。

在Visual C++菜单中选择Insert/Resource...,再在弹出的对话框中选择Bitmap，如图5.15所示。

如果有现成的位图，可以选择Import...，当然，在很多情况下，我们可以选New直接在资源编辑器中对它们进行创建、编辑。位图编辑器的使用很容易掌握，在此不再赘述。图形可以参考前面的插图。下面的代码摘自Menu.rc，以便读者在指定位图属性时作参考。

```
IDB_LINE BITMAP DISCARDABLE "res\\line.bmp"

IDB_CIRCLE BITMAP DISCARDABLE "res\\bitmap1.bmp"

IDB_RECTANGLE BITMAP DISCARDABLE "res\\rectangle.bmp"

IDB_TEXT BITMAP DISCARDABLE "res\\bmp00001.bmp"
```

同时，当然也应该注意映射的关系。只要与欲执行该功能的字符菜单项的ID号一致，那么，它们所实现的功能就是一样的。下面的几行代码摘自MainFrm.cpp

```
void CMainFrame::OnChange()

{

// TODO: Add your command handler code here

if(!bBitmap)
```



```

{
bBitmap=TRUE;

CMenu *pMenu=GetMenu();

CMenu *pSubMenu=pMenu->GetSubMenu(3);

pSubMenu->ModifyMenu(ID_DRAW_LINE,MF_BYCOMMAND,ID_DRAW_LINE,&hLine);

pSubMenu->ModifyMenu(ID_DRAW_CIRCLE,MF_BYCOMMAND,ID_DRAW_CIRCLE,&hCircle);

pSubMenu->ModifyMenu(ID_DRAW_RECTANGLE,

MF_BYCOMMAND,ID_DRAW_RECTANGLE,&hRectangle);

pSubMenu->ModifyMenu(ID_CHANGE,MF_BYCOMMAND,ID_CHANGE,&hText);

}

else

{

bBitmap=FALSE;

CMenu *pMenu=GetMenu();

CMenu *pSubMenu=pMenu->GetSubMenu(3);

pSubMenu->ModifyMenu(ID_DRAW_LINE,MF_BYCOMMAND,ID_DRAW_LINE,"&Line");

pSubMenu->ModifyMenu(ID_DRAW_CIRCLE,MF_BYCOMMAND,ID_DRAW_CIRCLE,"&Circle");

pSubMenu->ModifyMenu(ID_DRAW_RECTANGLE,

MF_BYCOMMAND,ID_DRAW_RECTANGLE,"&Rectangle");

pSubMenu->ModifyMenu(ID_CHANGE,MF_BYCOMMAND,ID_CHANGE,"&Bitmap");

}

}

```

而下面这段程序完成了位图资源的装入与初始化：

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{

// ...

```

```

hLine.LoadBitmap(IDB_LINE);

hCircle.LoadBitmap(IDB_CIRCLE);

hRectangle.LoadBitmap(IDB_RECTANGLE);

hText.LoadBitmap(IDB_TEXT);

// ...

}

```

工作间，此时，系统检测到资源已经在系统外部发生变化，询问是否重组数据库，选择是，我们所需要的就已经全部准备完毕。下面给出这一变化前后文件Menu.rc的变化情况。

```

//改变以前：

POPUP "&Help"

BEGIN

MENUITEM "&About Menu...", ID_APP_ABOUT

MENUITEM "&I ' m Menu inserted", ID_HELP_INSERT

//上面这行代码我们在实现了消息映射后将将其删除

END

```

```

//改变以后：

POPUP "&Help"

BEGIN

MENUITEM "&About Menu...", ID_APP_ABOUT

END

```

通过以上的比较我们不难发现，使用第二种方式进行修改是比较方便的。

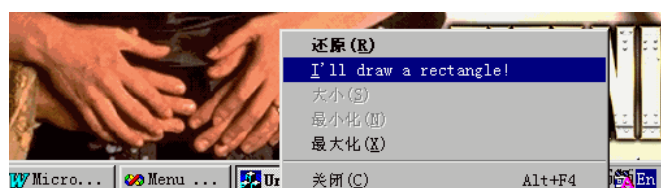


图5. 16 系统菜单改变后的任务条

下面我们讲一下有关系统菜单的消息响应问题。如图5.16，在我们改变系统菜单后，当在任务栏右单击鼠标时，弹出的系统菜单具有了与原来不同的形式。

顺便提一下，如果我们不希望用户在程序外部直接退出程序，改变系统菜单就是这个方法的一个途径。（如果将系统菜单设为NULL，用户就只有进入应用程序进行选择了。当然，除非这是必要的，否则，不要强迫用户改变他们的操作习惯。）

首先，我们必须取得系统菜单，这可以通过调用CWnd类的一个成员函数GetSystemMenu)达到。此函数的原型为：

```
CMenu *GetSystemMenu(BOOL bRevert) const;
```

在以TRUE作为bRevert的值进行调用时，恢复原系统菜单设置。在更多的情况下，我们以FALSE进行调用，此时我们可以对系统菜单进肋  
僮螯5 牵 孟 煊??/FONT>OnSysCommand函数，而且，此消息响应函数是系统设定的，不允许程序员另行指定。

- 注意：

- 我们可以通过系统提供的Spy++来查看系统菜单的消息发送及响应状况。至于详细的情况，在我们随后给出了详细的解释。与该消息相应的代码，我们将用具有底纹的文字标出。

在本章的最后，我们不妨来看看怎样在程序中实现右击鼠标时出现的上下文菜单，与前面我们所遇到的其他弹出式菜单在CmainFrame中实现不同，上下文菜单要在类CmenuView中实现。从本质上来说，实现上下文菜单时发送的消息与标准的菜单消息并没有什么两样，它们都发送ON\_COMMAND消息。但是，由于上下文菜单的特殊性，我们首先必须保证我们能接收到这样的消息。同时，由于上下文菜单中没有顶层菜单，我们对它们的跟踪有一定的特殊性。我们采用CMenu类的一个成员函数TrackPopupMenu来实现。该函数的原型为：

```
BOOL TrackPopupMenu(UINT nFlags, int x, int y, CWnd* pWnd, LPCRECT lpRect = NULL )
```

该函数中，参数nFlags指明屏幕位置与鼠标位置风格，x,y为一依赖于前面所指定风格的上下文菜单的屏幕位置坐标，pWnd为一指向拥有该菜单的窗口指针，而lpRect指明用户不会丢失掉上下文菜单的矩形域。参数nFlags可以取以下值：

屏幕位置风格：TPM\_CENTERALIGN, TPM\_LEFTALIGN, TPM\_RIGHTALIGN

鼠标位置风格：TPM\_LEFTBUTTON，TPM\_RIGHTBUTTON

至于说它们的取值是可以互相取“或”的，而它们的实际效果，我们希望读者通过实践具体看看。下面的程序段摘自例程中文件CMenuView.cpp中：

```
void CMenuView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CMenu ContextMenu;

    if(!ContextMenu.LoadMenu(IDR_CONTEXTMENU))
        AfxThrowResourceException();

    CMenu *pPopupMenu=ContextMenu.GetSubMenu(0);

    ASSERT(pPopupMenu!=NULL);

    ClientToScreen(&point);

    pPopupMenu->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON,
        point.x,point.y,AfxGetMainWnd());

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

下面我们对这一段代码作简要分析：

程序中先装入该菜单（使用LoadMenu装入），接下来由于上下文菜单的定位使用屏幕坐标，所以使用了一个CWnd类的成员函数ClientToScreen，然后使用了TrackPopupMenu将菜单以上下文方式显示，最后，程序调用CView类的通用代码处理一些平常信息。

- 注意：

- 由于该函数在类CMenuView中实现，在使用ClassWizard时务请注意基类的选择。如图5.17所示（如果错选了CMaimFrame类，程序最后所调用的处理例程为CWnd类的处理函数，上下文菜单的显示就成了问题）。

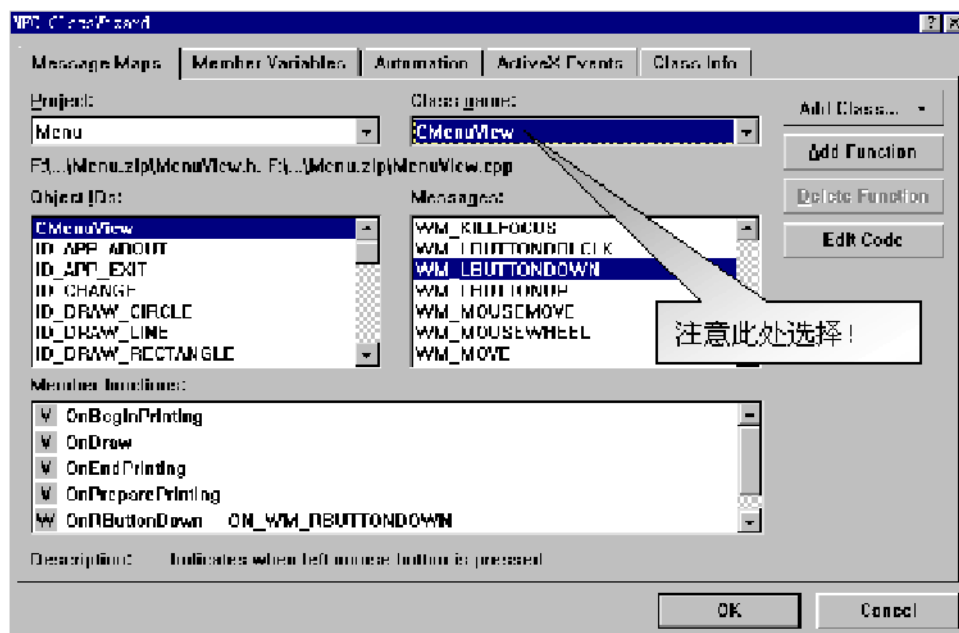


图5. 17 选择上下文菜单实现的基类

在本章的最后，我们给出一部分关键性代码，其中手工加入的部分以具有底纹的文字给出（基于版面方面的考虑，我们对一些重复性的代码略去了）。

- 注意：
- 由于菜单项的选择标记及使能实现较简单，此处省去了该部分。但实现时需调用消息响应WM\_INITMENU。
- 为了节省篇幅，在程序中我们没有在窗口变化后进行重绘，这部分的详细内容请参见GDI绘图一节。

```
// MainFrm.cpp：类CMainFrame 的实现

//

#include "stdafx.h"

// ...

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)

//{{AFX_MSG_MAP(CMainFrame)

ON_WM_CREATE()

ON_COMMAND(ID_VIEW_NORMALMENU, OnViewNormalMenu)
```

```

ON_COMMAND(ID_DRAW_LINE, OnDrawLine)

ON_COMMAND(ID_DRAW_CIRCLE, OnDrawCircle)

ON_COMMAND(ID_DRAW_RECTANGLE, OnDrawRectangle)

ON_COMMAND(ID_DYNCMODIFY_INSERT, OnDyncmodifyInsert)

ON_COMMAND(ID_HELP_INSERT, OnHelpInsert)

ON_COMMAND(ID_APP_EXIT, OnAppExit)

ON_COMMAND(ID_DYNCMODIFY_APPEND, OnDyncmodifyAppend)

ON_COMMAND(ID_HELP_APPEND, OnHelpAppend)

ON_COMMAND(ID_DYNCMODIFY_DELETE, OnDyncmodifyDelete)

ON_COMMAND(ID_DYNCMODIFY_MODIFY, OnDyncmodifyModify)

ON_COMMAND(ID_DYNCMODIFY_MODIFYSYSTEMMENU, OnDyncmodifyModifysystemmenu)

ON_COMMAND(ID_DYNCMODIFY_RESET, OnDyncmodifyReset)

ON_COMMAND(ID_CHANGE, OnChange)

ON_WM_SYSCOMMAND()

//此处为系统菜单消息响应声明，改变系统菜单后的消息处理必须声明该响应

ON_COMMAND(ID_FILE_NEW, OnFileNew)

ON_COMMAND(ID_VIEW_LONGMENU, OnViewLongmenu)

//}}AFX_MSG_MAP

END_MESSAGE_MAP( )

//以上函数映射部分仅供读者选用菜单ID时参考。但是，

//将这部分代码与实现系统菜单时的消息映射作一对比是有意义的。

static UINT indicators[] =

{

    ID_SEPARATOR, // status line indicator

    ID_INDICATOR_CAPS,

    ID_INDICATOR_NUM,

```

```

ID_INDICATOR_SCROLL,

};

//此处为状态栏上的分配，具体我们在后面会有提及。

////////////////////////////////////

// CMainFrame construction/destruction

CMainFrame::CMainFrame()

{
// TODO: add member initialization code here

bInsert=TRUE;//防止菜单被重复插入的控制变量

bAppend=TRUE;//防止菜单被重复加入的控制变量

bDelete=TRUE;//防止菜单被重复删除的控制变量

bModify=TRUE;//防止菜单被重复修改的控制变量

bBitmap=FALSE;//标识是否使用了图符菜单

}

//在CMainFrame类的构造函数中，我们加入了一部分支持菜单动态

//处理控制的BOOL型变量，它们的定义在函数CMainFrame.h中

// ...

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

{
// ...

// 提示：以下部分代码用于控制工具条提示及工具条本身实现，

//如果不需要可以将下面两行代码移去

m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |

CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

//提示：下面三行代码控制工具条是否可以移动，如果你并不

//希望工具条可以在屏幕内移动，可以删去它们。

```

```

m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);

EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(&m_wndToolBar);

hLongMenu.LoadMenu(IDR_MAINFRAME1);

hNormalMenu.LoadMenu(IDR_MAINFRAME0);

hLine.LoadBitmap(IDB_LINE);

hCircle.LoadBitmap(IDB_CIRCLE);

hRectangle.LoadBitmap(IDB_RECTANGLE);

hText.LoadBitmap(IDB_TEXT);

//以上几行代码装入资源（菜单，位图），但由于上下文菜单
//处理机制不一样，其资源并不在此处装入。

return 0;

}

//以上省略代码用于调试，在程序的发行版中将被移去

// ...

////////////////////////////////////

// CMainFrame message handlers

void CMainFrame::OnViewLongmenu()

{

// TODO: Add your command handler code here

SetMenu(&hLongMenu);

hLongMenu.Detach();

//以上部分代码用于实现至长菜单的切换

}

void CMainFrame::OnViewNormalmenu()

{

```



```

// TODO: Add your command handler code here

SetMenu(&hNormalMenu);

hNormalMenu.Detach();

//以上部分代码用于实现至普通菜单的切换

}

void CMainFrame::OnDrawLine()

{
// TODO: Add your command handler code here

CClientDC ClientDC(this);//创建一客户区设备对象

ClientDC.MoveTo(120,120);//移动对象起点至（120，120）

ClientDC.LineTo(260,260);//自（120，120）至（260，260）画直线

//以上部分代码用于在程序主窗口中作一起点

//为（120，120），终点为（260，260）的直线

}

void CMainFrame::OnDrawCircle()

{
// TODO: Add your command handler code here

CClientDC ClientDC(this);

ClientDC.Arc(80,40,160,120,80,80,80,80);

//以上代码部分用于在屏幕内画圆，基本同上，只是最后作图调用函数Arc（）

}

void CMainFrame::OnDrawRectangle()

{
// TODO: Add your command handler code here

CClientDC ClientDC(this);

ClientDC.MoveTo(120,120);

```

```

ClientDC.LineTo(170,120);

ClientDC.LineTo(170,150);

ClientDC.LineTo(120,150);

ClientDC.LineTo(120,120);

//基本解释同上，使用设备对象画矩形

}

void CMainFrame::OnDyncmodifyInsert()

{

// TODO: Add your command handler code here

if(bInsert)//控制重复加入变量为真时允许加入

{

CMenu *pMenu=GetMenu( );//取得程序主菜单指针

CMenu *pSubMenu=pMenu->GetSubMenu(3);

//函数GetSubMenu()用于从0基取得菜单项指针

pSubMenu->InsertMenu( ID_APP_ABOUT,

MF_BYCOMMAND, ID_HELP_INSERT, "&I'm inserted here!");

bInsert=FALSE;

}

//以上部分代码用于在HELP菜单中About....项前

//加入一新菜单 " I ' m inserted here! "

}

void CMainFrame::OnHelpInsert()

{

// TODO: Add your command handler code here

MessageBox("I'm inserted in the help menu!");

//显示一标准消息框，当然，此处对加入菜单的响应较简单，

```

```

//仅仅是为了说明问题而已,在后面的函数实现中我们遵循同样的原则,
//读者如果有兴趣,尽可以以自己编写的函数响应代替之
//以上为新加入菜单的响应函数,请读者
//注意前面所讲的加入该响应函数的方法
}

void CMainFrame::OnAppExit()
{
// TODO: Add your command handler code here
MessageBox("Thank you for using this programe!");
PostMessage(WM_CLOSE); //发送程序退出消息

//我们对程序退出进行控制,由于程序退出总会产生对该函数的调用,
//在这里,我们力图使我们的程序显得有礼貌些,这一部分读者也可以
//改写之以实现自己所期望的功能
}

void CMainFrame::OnDyncmodifyAppend()
{
// TODO: Add your command handler code here
if(bAppend)//动态加菜单项至菜单末尾BOOL型控制变量
{
CMenu *pMenu=GetMenu();
CMenu *pSubMenu=pMenu->GetSubMenu(3);
pSubMenu->AppendMenu(MF_STRING, ID_HELP_APPEND, "&I'm appended here!");
bAppend=FALSE;
}

//以上部分代码在Help菜单的最末尾加入一菜单项 " I ' m appended here! "
}

```

```

void CMainFrame::OnHelpAppend()
{
// TODO: Add your command handler code here
MessageBox("I'm appended in the help menu!");
//菜单项 " I ' m appended here! " 的消息响应函数
}

void CMainFrame::OnDyncmodifyDelete()
{
// TODO: Add your command handler code here
if(bDelete)//动态删除菜单项BOOL型控制变量
{
CMenu *pMenu=GetMenu();
CMenu *pSubMenu=pMenu->GetSubMenu(0);
pSubMenu->DeleteMenu(5,MF_BYPOSITION);
//这里我们采用用位置对菜单项定位，注意菜单项是0基的
//注意：菜单项分隔线也是菜单项
bDelete=FALSE;
}
//以上部分代码删除File中第六个菜单项
}

void CMainFrame::OnDyncmodifyModify()
{
// TODO: Add your command handler code here
if(bModify)//动态修改菜单项BOOL型控制变量
{
CMenu *pMenu=GetMenu();

```

```

CMenu *pSubMenu=pMenu->GetSubMenu(3);

pSubMenu->ModifyMenu(ID_APP_ABOUT,
MF_BYCOMMAND,ID_APP_ABOUT,"&Version information!");

bModify=FALSE;

}

//动态的改变Help菜单中显示程序版本信息对话框的菜单文本
//(这种改变后菜单项是不是更能较贴切的反应菜单项真实内容)?

}

void CMainFrame::OnDyncmodifyModifysystemmenu()
{
// TODO: Add your command handler code here

CMenu *pSystemMenu=GetSystemMenu(FALSE);

//以FALSE调用系统菜单，以便对其进行修改，
//但请注意其消息响应函数的特殊性

pSystemMenu->ModifyMenu(1,MF_BYPOSITION,ID_FILE_NEW,"&I'll draw a rectangle!");

//改变系统菜单中“移动”菜单项，以实现在程序窗口中画一矩形

//以上代码修改了系统菜单

}

void CMainFrame::OnDyncmodifyReset()
{
// TODO: Add your command handler code here

CMenu *pSystemMenu=GetSystemMenu(TRUE);

//以上部分以TRUE为参数调用系统菜单，实现了系统菜单的复原

}

void CMainFrame::OnChange()
{

```

```

// TODO: Add your command handler code here

if(!bBitmap)//控制使用何种菜单的变量
{
    bBitmap=TRUE;

    CMenu *pMenu=GetMenu( );

    CMenu *pSubMenu=pMenu->GetSubMenu(3);

    pSubMenu->ModifyMenu(ID_DRAW_LINE,MF_BYCOMMAND,ID_DRAW_LINE,&hLine);

    pSubMenu->ModifyMenu(ID_DRAW_CIRCLE,MF_BYCOMMAND,ID_DRAW_CIRCLE,&hCircle);

    pSubMenu->ModifyMenu(ID_DRAW_RECTANGLE,
        MF_BYCOMMAND,ID_DRAW_RECTANGLE,&hRectangle);

    pSubMenu->ModifyMenu(ID_CHANGE,MF_BYCOMMAND,ID_CHANGE,&hText);

}

//以上部分分别以位图对象指针作ModifyMenu( )函数的最后一个参数,

//将菜单换为了图符菜单

else
{
    bBitmap=FALSE;

    CMenu *pMenu=GetMenu();

    CMenu *pSubMenu=pMenu->GetSubMenu(3);

    pSubMenu->ModifyMenu(ID_DRAW_LINE,MF_BYCOMMAND,ID_DRAW_LINE,"&Line");

    pSubMenu->ModifyMenu(ID_DRAW_CIRCLE,MF_BYCOMMAND,ID_DRAW_CIRCLE,"&Circle");

    pSubMenu->ModifyMenu(ID_DRAW_RECTANGLE,
        MF_BYCOMMAND,ID_DRAW_RECTANGLE,"&Rectangle");

    pSubMenu->ModifyMenu(ID_CHANGE,MF_BYCOMMAND,ID_CHANGE,"&Bitmap");

}

//以上部分分别以相应文本对象指针作ModifyMenu( )函数的最后一个参数,

```

```
//将菜单换为了文本菜单

//以上部分代码实现Draw的图符菜单与文本菜单的切换
}

//注意下面这一部分代码，系统菜单的消息响应总是调用函数
//OnSysCommand( )，同时注意该函数的映射接口的特殊性
//系统菜单发送的消息为WM_SYSCOMMAND,而普通的菜单
//发送消息WM_COMMAND，这一点通过Spy++工具可以很
//明显的看出

void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID&0xFFF0)==ID_FILE_NEW)
    //在消息WM_SYSCOMMAND中,参数nID的低四位为
    //Windows系统内部使用，在应用程序使用其值时，
    //应与0xFFF0求与后使用方能得到正确结果。

    {
        CMainFrame::OnFileNew();
    }

    //此处产生正常调用，在应用程序没有涉及的消息映射，
    //应调用系统标准调用处理

    else
    {
        CFrameWnd::OnSysCommand(nID, lParam);
    }

    //以上部分代码改变了系统缺省菜单的行为，当然，读者对
    //系统菜单的改变可能更具创造性
```

```

}

void CMainFrame::OnFileNew()

{
// TODO: Add your command handler code here

CMainFrame::OnDrawRectangle();

MessageBox("I've drawn a rectangle!");

//系统菜单中的一个消息响应函数，它的编写没有什么特殊之处

}

//以下为函数MenuView.cpp的代码

// MenuView.cpp :类CMenuView的实现

//

#include "stdafx.h"

// ...

////////////////////////////////////

// CMenuView

IMPLEMENT_DYNCREATE(CMenuView, CView)

BEGIN_MESSAGE_MAP(CMenuView, CView)

//{{AFX_MSG_MAP(CMenuView)

ON_WM_RBUTTONDOWN()

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

//以上为消息响应入口

////////////////////////////////////

// CMenuView construction/destruction

CMenuView::CMenuView()

{

```



```

// TODO: add construction code here

//CMenuView类构造函数，在文档/视一章中我们将对此作详细的解释
}

CMenuView::~CMenuView()

{
//CMenuView类析构函数，在文档/视一章中我们将对此作详细的解释
}

BOOL CMenuView::PreCreateWindow(CREATESTRUCT& cs)

{
// TODO: Modify the Window class or styles here by modifying
// the CREATESTRUCT cs
//在此处可以改变窗口的实现
return CView::PreCreateWindow(cs);
}

////////////////////////////////////

// CMenuView drawing

void CMenuView::OnDraw(CDC* pDC)

{
CMenuDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

//文档绘图调用函数

// TODO: add draw code for native data here
}

// CMenuView diagnostics

// ...

//以上部分为调试代码时用，在程序的发行版中将被移去

```

```

CMenuDoc* CMenuView::GetDocument( ) // 该函数非调试时为内联函数
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMenuDoc)));
    return (CMenuDoc*)m_pDocument;
}

//获得文档指针

#endif // _DEBUG

////////////////////////////////////

// 以下为CMenuView 类消息处理函数

//下面的代码处理了右击鼠标时上下文菜单的处理，当然，你也
//可以改在别的鼠标事件时使用上下文菜单

void CMenuView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CMenu contextMenu;

    if(!ContextMenu.LoadMenu(IDR_CONTEXTMENU))
        AfxThrowResourceException();

    //装入上下文菜单资源，装入失败时调用异常处理函数

    CMenu *pPopupMenu=contextMenu.GetSubMenu(0);

    ASSERT(pPopupMenu!=NULL); //使用断言，方便调试

    ClientToScreen(&point); //转换客户区坐标至屏幕坐标

    pPopupMenu->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON,
    point.x,point.y,AfxGetMainWnd()); //将菜单作为上下文菜单显示

    CView::OnRButtonDown(nFlags, point);
}

//以上部分为上下文菜单的处理函数

```

```
//以下部分代码摘自MainFrm.h中
```

```
// ...
```

```
// Attributes
```

```
public:
```

```
BOOL bInsert;
```

```
BOOL bAppend;
```

```
BOOL bDelete;
```

```
BOOL bModify;
```

```
BOOL bBitmap;
```

```
//以上部分代码为控制变量
```

```
CMenu hLongMenu;
```

```
CMenu hNormalMenu;
```

```
//以上部分为菜单资源对象
```

```
// ...
```

## 第二节 工具条

首先让我们从MFC控制条谈起。

MFC的工具条类CToolBar是几种可创建用来接收某些命令输入并向用户显示状态消息的类中的一类。用户可以用工具条来立即访问程序命令。工具条是直接可以看到的，而不是象菜单那样需要一层一层的深入，或象键盘那样需要记忆。由于它占用屏幕空间，因此一定要确保你的工具条包括的是最经常使用的命令。大型程序通常有多个工具条来为不同的用户任务服务。即使你的程序只有一个工具条，也要使用户在感觉它碍事时能将其藏起来。

从编程的角度看，工具条是一个显示一系列位图按钮的子窗口。一旦创建了工具条并使其可见，就不能再忽略它，因为它将和菜单与加速键一样生成WM\_COMMAND消息。但应使工具条的命令ID与菜单和加速键的命令ID同步。

在这一节中，将介绍几个有关工具条的问题。我们将首先看看工具条在MFC层次结构中的位置，然后我们再介绍有关动态创建和修改工具

条的若干细节，最后，我们将说明工具条定位与消隐的一些技术并给出一个相应的较小的例程。我们计划按下列顺序讲解：

- MFC控制条
- 创建工具条
- 显示和隐藏工具条

从图5.18中可见工具条类的基本类为CControlBar，而该基本类是由CWnd类派生的。掌握这个继承关系对我们来说是很有用的。例如，由于所有的控制条都是有CWnd类派生的，所有的控制条都连接到一个Windows API窗口。因此，CWnd的所有功能——创建、移动、显示和隐藏窗口——在用控制条工作是都是可用的。

CToolBar类有几个兄弟类，包括CStatusBar和COleResizerBar，CDialogBar。当发生请求时，AppWizard通过创建一个CStatusBar对象来创建一个状态条。正如我们在介绍菜单时所提到的，状态条位于帧窗口的底部并显示有关菜单选择的帮助消息。当用户在各按钮上移动鼠标时，状态条也显示关于不同工具条按钮的详细帮助信息。

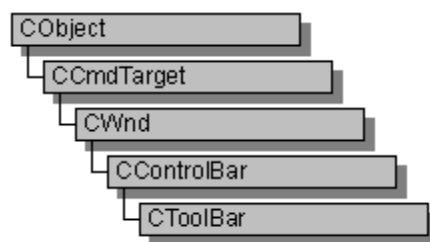


图5.18 类CToolBar继承关系图

CDialogBar类创建对话条，它位于工具条和对话框之间。对话条能够象对话框那样起控制作用，但它象一个工具条那样位于帧窗口中。我们对此不打算作详细介绍，有兴趣的读者可以参考有关书籍。

MFC控制条COleResizerBar可以改变在为OLE对象的大小。

MFC的工具条（和对话条）都是可迁移的。如允许这一特性，用户就可将一个工具条移到一个具有不同边界的帧窗口中。但它将根据这一新窗口的边界而更新以表明它所迁移的位置。此外，工具条还可被移出帧边界并被放到空闲的调色板上。

MFC工具条也支持工具便笺，工具便笺可帮助用户理解单独一个位图按钮的作用。如允许这一特性，用户就可通过将鼠标光标移到一个工

具条按钮上（并不单击按钮）来调用工具便笺支持。过一会儿以后，就会在工具条按钮上方的一个小的正文窗口中显示工具便笺——一个简略的字或短语。

为在你的MFC程序中使用工具条，必须协调使用位图资源，包括工具条自身以及帧窗口等几项。为帮助用户对此的理解。在后面我们将介绍如何创建自己的工具条。

下面我们创建一个工具条。AppWizard自动生成一个工具条。但它的某些特性，比如象显示和隐藏工具条，则被隐藏在现有MFC类中。为了能更详细地介绍工具条，我们将创建另一个菜单。在此之后，我们将说明两个工具条如何共处于同一个程序中。这样，你就将看到创建一个附加的工具条是多么容易。

创建工具条分五步。首先要创建按钮映象的位图。其次要建立一个将按钮映射到你的程序命令码的数组。第三要编写创建工具条并将其适当初始化的程序。一旦工具条窗口被创建，第四步和第五步是将按钮映象和命令ID与工具条相连接。一旦这些基本步骤完成后，就可采取其它步骤来进一步改善工具条。下面我们分别讲解。

1. 创建按钮映象位图。应将该位图作为位图资源存储起来。如图5.19，打开资源编辑器，将按钮映象的位图创建到一行中。每个映象的缺省大小是16×15像素。如果你希望使用别的大小，那么你就需要调用函数CToolBar::SetSize通知应用程序相应的改变。该函数的相关信息可以通过系统的帮助信息获得，在此不再赘述。

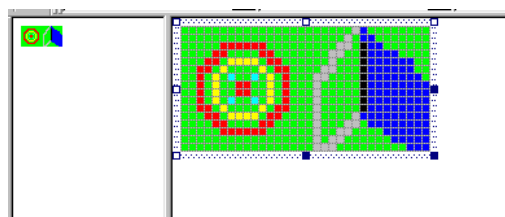


图5.19 建立工具条位图资源

2. 定义命令码数组。映射按钮映象到命令ID号。下面的代码摘自例程，无符号整形数组（UINT）为上图所示工具条按钮的对应数组。

3. 创建并初始化工具条对象。与其它创建过程类似，采用两步法生成：先声明对象，再调用初始化函数CToolBar::Create生成。该函数原型为：

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle=WS_CHILD| WS_VISIBLE | CBRS_TOP,
UINT nID = AFX_IDW_TOOLBAR );
```

该函数中参数pParentWnd为指向工具条所在父窗口的指针（别忘了工具条本身也是窗口！），dwStyle为工具条的风格说明，其取值如表5.1所示：

该函数最后一个参数表示工具条子窗口的ID号。

如下述代码生成一工具条：

```
d_pToolBar=new CToolBar( );  
  
d_pToolBar->Create( this,WS_CHILD|CBRS_TOP,0x9100);
```

表5. 1 工具条风格

标志	简单描述
WS_VISIBLE	使工具条窗口初始可见
CBRS_BOTTOM	初始时将工具条放到窗口底部
CBRS_FLYBY	鼠标光标在按钮上暂停时，显示命令描述
CBRS_NOALIGN	防止控制条在其父窗口改变大小时被复位
CBRS_TOOLTIPS	鼠标光标在按钮上暂停时，显示工具便笺
CBRS_TOP	初始时将工具条放在窗口底部

4. 将储存在位图资源中的按钮映象与程序的工具条相连接。这可以通过函数调用CToolBar:LoadBitmap达到。该函数原型为：

```
BOOL LoadBitmap( LPCTSTR lpszResourceName );  
  
BOOL LoadBitmap( UINT nIDResource );
```

该函数提供的重载两个不同版本，使我们既可以通过指向位图资源的指针调用，也可以通过该位图的ID号调用达到。

如下段代码：

```
d_pToolBar->LoadBitmap(IDR_TOOLS);
```

5. 作为该过程的最后一步，我们需要将按钮与命令ID联系起来。这可以用一个在第二步中创建的数组的指针来调用函数

CToolBar::SetButtons达到，该函数的原型为：

```
BOOL SetButtons( const UINT* lpIDArray, int nIDCount );
```

其中参数lpIDArray为指向一命令ID数组的指针。当然，为了使用不含按钮的工具条，该参数可能为NULL，参数nIDCount为前一数组中的元素个数。

一般来说，我们可以这样编写我们的程序：

```
d_pToolBar->SetButtons(buttons,sizeof(buttons)/sizeof(UINT);
```

到此为止，我们所创建的工具条已经实现。当然，我们可能仍然需要以别的方式改善工具条的操作。对于一般要求来说，最常用的操作也许就是移动和显隐切换了。

我们先谈其移动。在缺省状况下，一个CToolBar工具条只能被应用程序所移动。但也可以使用户能够将工具条移到帧的另一部分。为此，需向工具条及帧窗口发送消息。这可通过调用CToolBar::EnableDocking和CFrame::EnableDocking实现。二函数原型均如下：

```
void EnableDocking( DWORD dwStyle );
```

其中参数dwStyle为工具条风格，对CToolBar其取值可如下：

而对于CFrame，风格值CBRS\_FLOAT\_MULTI不可用。

下面这段代码几乎也是标准的：

```
d_pToolBar->EnableDocking(CBRS_ALIGN_ANY);  
  
EnableDocking(CBRS_ALIGN_ANY);
```

表5. 2 工具条停靠风格

风格	含义
CBRS_ALIGN_TOP	工具条可在客户区顶端移动
CBRS_ALIGN_BOTTOM	工具条可在客户区底端移动
CBRS_ALIGN_LEFT	工具条可在客户区左端移动
CBRS_ALIGN_RIGHT	工具条可在客户区右端移动

CBRS_ALIGN_ANY	工具条可在客户区任意位置移动
CBRS_FLOAT_MULTI	允许在一单边窗口内存在多个可移动控制条

用户也可以将工具条移动或定位。或者在程序控制下，通过调用 `CFrameWnd::DockControlBar` 来移动以及调用 `CFrameWnd::FloatControlBar` 来定位一工具条。它们的原型及参数如下所示：

```
void DockControlBar( CControlBar * pBar, UINT nDockBarID = 0, LPCRECT lpRect = NULL );
```

其中 `pBar` 为指向欲移动的工具条的指针，`nDockBarID` 决定框架窗口哪一边可移动。它为0时，则控制条可任意移动，它取值 `AFX_IDW_DOCKBAR_TOP`，`AFX_IDW_DOCKBAR_BOTTOM`，`AFX_IDW_DOCKBAR_LEFT`，`AFX_IDW_DOCKBAR_RIGHT` 时，分别表示控制条可移动至框架窗口的顶端，底端，左端及右端，该函数最后一个参数标识控制条可放置的框架非客户区的屏幕坐标。

第二个函数的原型及参数读者不妨从系统帮助文件中查到。

例如，我们可以这样编写代码：

```
d_pToolBar->EnableDocking(CBRS_ALIGN_ANY);
```

```
EnableDocking(CBRS_ALIGN_ANY);
```

接下来我们再谈谈工具条的显隐控制。由于工具条是一个窗口，它的显隐可以通过其父类 `CWnd` 的成员函数实现。

在我们要改变工具条状态前知道当时工具条的状态有时是至关重要的，工具条的可视性可以通过函数 `CWnd::GetStyle` 拣取。该函数不带参数，其原型为：

```
DWORD GetStyle( ) const;
```

函数返回当前控制条风格值。

而通过对函数 `CWnd::SetStyle` 的调用可以改变某些窗口风格，但该函数不能改变 `WS_VISIBLE`，这意味着我们不得不通过调用其基类的成员函数 `ShowWindow` 来实现：将参数 `SW_HIDE` 传给函数以使工具条不可见或传递 `SW_SHOWNORMAL` 使工具条再次显示。函数 `SetStyle` 的原型及参数读者不妨自己从系统的帮助文件中查到。



一旦在程序中改变了工具条，就必须将这一改变通知帧窗口，为此需再次计算控制条的位置，这可以通过调用函数 `CFrameWnd::RecalcLayout` 来实现。该函数为一不带参数的函数，读者在程序中可以简单地调用即可。例如程序中下列代码

```
RecalcLayout( );
```

即可将该工具条的变化通知到程序窗口中。

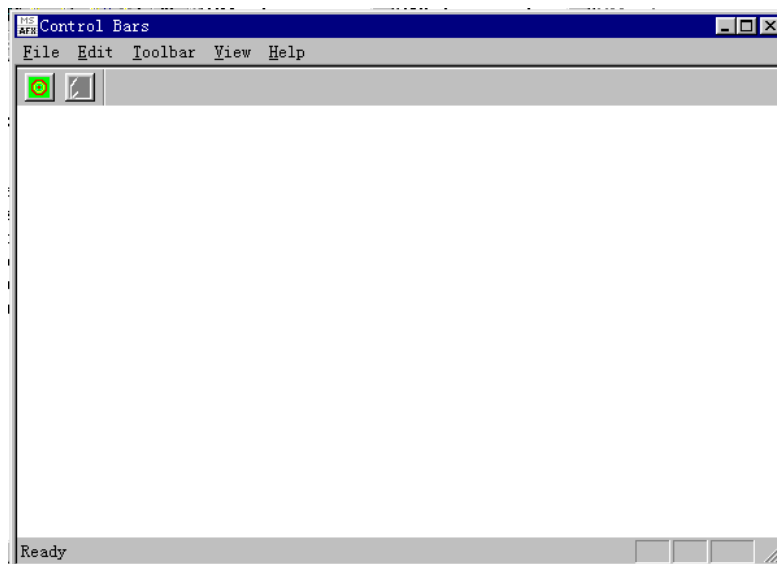


图5. 20 程序初始画面

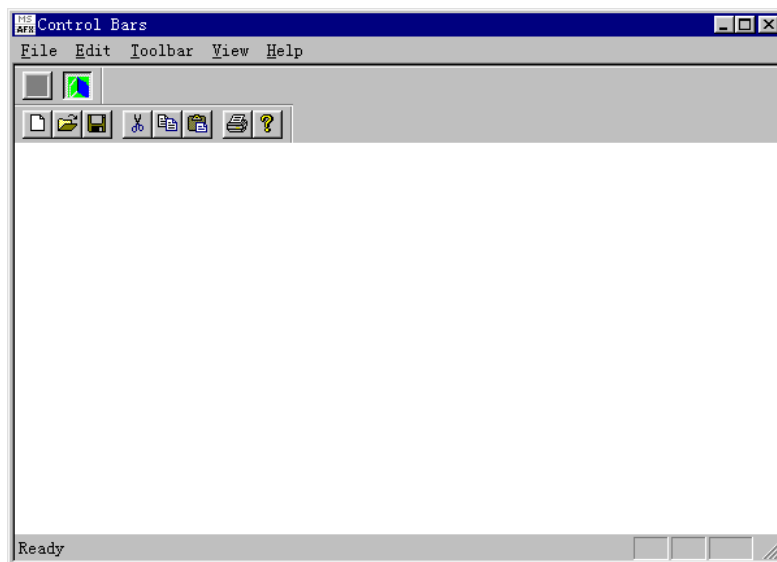


图5. 21 两个工具条并存

- 注意：
- 限于篇幅原因，本节内容给出一个源程序代码，但较不完整。读者如果感兴趣，可以自己试着编写完全。但需要注意的是，在编

辑工具条时，应该将工具条的各个按钮作为位图储存。否则，在程序进行编译时，会出现ID号未定义错。（Undeclared Identifier error）。下面给出程序的一些关键性代码。而且由于前文对各函数讲得较细致，对下面的源代码我们给出的解释较简单。同时，考虑到只有文件mainframe.cpp中改动较多，下面的源代码仅为该文件。但我们给出了几幅运行画面（图5.20到图5.22）供比较。



图5.22 隐去自编工具条

```
// mainfrm.cpp : 类DMainFrame的实现  
  
//  
  
// ...  
  
#include "stdafx.h"  
  
// ...  
  
//以上为程序包含文件部分。  
  
IMPLEMENT_DYNCREATE(DMainFrame, CFrameWnd)  
  
BEGIN_MESSAGE_MAP(DMainFrame, CFrameWnd)  
  
//{{AFX_MSG_MAP(DMainFrame)  
  
ON_WM_CREATE()  
  
ON_COMMAND(ID_TOOLBAR_CREATE, OnToolBarCreate)  
  
ON_COMMAND(ID_TOOLBAR_SHOW, OnToolBarShow)
```

```

ON_UPDATE_COMMAND_UI(ID_TOOLBAR_CREATE, OnUpdateToolBarCreate)

ON_UPDATE_COMMAND_UI(ID_TOOLBAR_SHOW, OnUpdateToolBarShow)

ON_COMMAND(ID_EDIT_COPY, OnEditCopy)

ON_COMMAND(ID_EDIT_CUT, OnEditCut)

ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)

ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)

ON_COMMAND(ID_FILE_NEW, OnFileNew)

ON_COMMAND(ID_FILE_OPEN, OnFileOpen)

ON_COMMAND(ID_FILE_SAVE, OnFileSave)

ON_COMMAND(ID_FILE_PRINT, OnFilePrint)

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

//以上部分为消息响应,由于要在程序中改变图标选中状态,我们处理了消息
ON_UPDATE_COMMAND,

//该消息在每一个发送该消息之前发送,因此,通过处理该消息,我们就可以在用户看到该图标
出

//现以前改变该图标表现.

////////////////////////////////////

// arrays of IDs used to initialize control bars

// toolbar buttons - IDs are command buttons

static UINT BASED_CODE buttons[] =

{

// same order as in the bitmap 'bitmap1.bmp'

ID_TOOLBAR_CREATE,

ID_SEPARATOR,

ID_TOOLBAR_SHOW

//以上部分为工具条上按钮对应情况, ID_SEPARATOR在两个相邻按钮间加一分隔线

```

```

};

// toolbar buttons - IDs are command buttons

static UINT BASED_CODE Toolbar2Buttons[] =

{
// same order as in the bitmap 'toolbar.bmp'

ID_FILE_NEW,

ID_FILE_OPEN,

ID_FILE_SAVE,

ID_SEPARATOR,

ID_EDIT_CUT,

ID_EDIT_COPY,

ID_EDIT_PASTE,

ID_SEPARATOR,

ID_FILE_PRINT,

ID_APP_ABOUT,

//这一部分同上,注意,工具条按钮间分隔线最好在编程时确定.同时,一定要注意按钮的对应
//情况.

};

static UINT BASED_CODE indicators[] =

{

ID_SEPARATOR, // status line indicator

ID_INDICATOR_CAPS,

ID_INDICATOR_NUM,

ID_INDICATOR_SCRL,

//状态条空间分配

};

////////////////////////////////////

```

```

// DMainFrame construction/destruction

DMainFrame::DMainFrame()

{
d_pToolBar2 = 0;
d_bToolBarVisible = FALSE;
//在此处我们为工具条初始化加入代码,新建工具条初始时不可见
//以上二变量在文件mainfrm.h中定义:
//public:
// CToolBar * d_pToolBar2; // Pointer for dynamic toolbar.
// BOOL d_bToolBarVisible; // Flag for toolbar visibility.
}

DMainFrame::~DMainFrame()

{
//mainframe类析构函数
}

int DMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
return -1;
//以下代码创建一工具条
if (!m_wndToolBar.Create(this) ||
!m_wndToolBar.LoadBitmap(IDR_MAINFRAME) ||
!m_wndToolBar.SetButtons(buttons,
sizeof(buttons)/sizeof(UINT)))
{
TRACE0("Failed to create toolbar\n");
}
}

```

```

return -1; // fail to create
}

//以下代码创建一可移动工具条,如程序不希望如此,可删除下列代码
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);

EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(&m_wndToolBar);

//此下代码使能工具提示,如不需要,可移去
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY);

//以下代码创建状态条
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
}

return 0;

//以上部分为系统对给出的工具条,状态条的初始化,读者不妨对其仔细研究一下.
}

////////////////////////////////////

// DMainFrame diagnostics

// ...

//其中删节部分调试程序时有用

// ...

////////////////////////////////////

```

```

// DMainFrame message handlers

//以下为消息处理函数

// 菜单项ToolBar|Create消息处理,用于创建工具条

void DMainFrame::OnToolBarCreate()

{

// 仅在工具条不存在时断言成立

ASSERT(d_pToolBar2 == 0);

// 创建工具条对象,由于正文中已作较多解释,此处从简

d_pToolBar2 = new CToolBar();

d_pToolBar2->Create(this, WS_CHILD | CBRS_TOP |

CBRS_TOOLTIPS | CBRS_FLYBY,

0x9100);

//获得位图,并与工具条相联系

d_pToolBar2->LoadBitmap(IDR_TOOLS);

d_pToolBar2->SetButtons(ToolBar2Buttons,

sizeof(ToolBar2Buttons)/sizeof(UINT));

//使能工具条移动

d_pToolBar2->EnableDocking(CBRS_ALIGN_ANY);

EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(d_pToolBar2);

}

//-----

// 选项ToolBar|Show消息处理

void DMainFrame::OnToolBarShow()

{

ASSERT(d_pToolBar2 != 0);

```

```

//查询工具条当前状态

BOOL bVisible = (d_pToolBar2->GetStyle() & WS_VISIBLE);

//显隐切换

int nShow = (bVisible) ? SW_HIDE : SW_SHOWNORMAL;

d_pToolBar2->ShowWindow(nShow);

//发送工具条变化通知

RecalcLayout();

//记录工具条显示状态以留作后用

d_bToolBarVisible = (!bVisible);

}

//-----

//处理消息 Toolbar|Create ON_COMMAND_UPDATE_UI,对菜单项 Toolbar|Create作使能检查

void DMainFrame::OnUpdateToolBarCreate(CCmdUI* pCmdUI)

{

pCmdUI->Enable(d_pToolBar2 == 0);

}

//-----

//处理消息 Toolbar|Show ON_COMMAND_UPDATE_UI,对菜单项 Toolbar|Create作选择检查

void DMainFrame::OnUpdateToolBarShow(CCmdUI* pCmdUI)

{

pCmdUI->Enable(d_pToolBar2 != 0);

int nCheck = (d_bToolBarVisible) ? 1 : 0;

pCmdUI->SetCheck(nCheck);

}

//-----

// 消息Edit|Copy选择处理,为了简化程序,对各选项我们的处理是仅仅显示一对话

```



//框显示该菜单项已被选择而已

```
void DMainFrame::OnEditCopy()
```

```
{
```

```
AfxMessageBox(_T("Edit|Copy command selected."));
```

```
}
```

```
//-----
```

```
// WM_COMMAND handler for Edit|Cut.
```

```
void DMainFrame::OnEditCut()
```

```
{
```

```
AfxMessageBox(_T("Edit|Cut command selected."));
```

```
}
```

```
//-----
```

```
// WM_COMMAND handler for Edit|Paste.
```

```
void DMainFrame::OnEditPaste()
```

```
{
```

```
AfxMessageBox(_T("Edit|Paste command selected."));
```

```
}
```

```
//-----
```

```
// WM_COMMAND handler for Edit|Undo.
```

```
void DMainFrame::OnEditUndo()
```

```
{
```

```
AfxMessageBox(_T("Edit|Undo command selected."));
```

```
}
```

```
//-----
```

```
// WM_COMMAND handler for File|New.
```

```
void DMainFrame::OnFileNew()
```

```

{
AfxMessageBox(_T("File|New command selected.));
}

//-----

// WM_COMMAND handler for File|Open.
void DMainFrame::OnFileOpen()
{
AfxMessageBox(_T("File|Open... command selected.));
}

//-----

// WM_COMMAND handler for File|Save.
void DMainFrame::OnFileSave()
{
AfxMessageBox(_T("File|Save command selected.));
}

//-----

// WM_COMMAND handler for File|Save.
void DMainFrame::OnFilePrint()
{
AfxMessageBox(_T("File|Print... command selected.));
}

```

### 第三节 快捷键消息响应

键盘加速键有时也叫键盘捷径,它使用户可用键盘发出命令。Windows API提供了加速键表资源,用以保存加速键定义集合。AppWizard生成一单文档或多文档应用程序时,提供一个加速键表当通过调用 CFrameWnd::LoadFrame来初始化一个帧(CFrameWnd)窗口时,则加速键表被自动连接到该帧窗口。

但有几点我们必须提出来说一下。首先,在Microsoft研制Windows时,并没有考虑到加速键系统。毕竟在当时看来,Windows所支持的,最主要的应该是鼠标和菜单。毫无疑问,这是当时许多的GUI系统(包括著名的Star和Apple的Macintosh)得以成功的巨大原因。但出于Microsoft对第三方厂家软件的一贯支持,Windows系统中还是加入了加速键支持部分。在今天看来,Microsoft的这一步,实际上具有重大的意义,这至少可以从两个方面加以说明:并不能强迫所有的计算机都以鼠标作为输入手段,也并不是在所有的情形下,鼠标输入都较键盘输入为快,而且,对一些已经习惯了键盘输入的用户来说,使用鼠标输入即使只是一个很容易的转变,可毕竟也还是一个转变,用户需要对此有一个适应的过程。

同时也应该指出的时,Microsoft在<<Windows95用户界面设计指南>>中特别提出,为增强Windows95应用程序的一致性,键盘命令“不应该是进行特定操作的唯一方法。”实际上,指南中将加速键称为“捷径”多少说明了Microsoft的倾向:它应该也只能是其它命令输入机制的一种补充而不是替代。它只应该是以鼠标作为主要输入手段的Windows系统的一种平衡。实际上,我们需要注意的一点就是,我们所建立的加速键,要么是使用菜单项进行逐项选择较麻烦,或者是我们系统中使用频率相当高的部分,同时,加速键要尽量简单,否则加速键就失去了其本来的意义。遵循这些简单的原则,我们就能开发出用户界面简捷的应用程序。

在本节中,为了理解的清晰,我们将首先简单的介绍一下键盘输入的处理,然后,我们将进行加速键的处理的介绍。我们将尽量介绍清楚键的使用规则及其运行机制,最后,作为一个小小的补充,我们将介绍如何动态地安装一个加速键表。

下面,我们首先简单地介绍一下键盘输入。

在<<Windows 95用户界面设计指南>>中介绍了四种键盘输入:正文键,访问键,模式键以及快捷键。正文键指的是可打印字符:大小写字母,数字,标点符号及其它符号。用户期望用字符和数字键来产生可打印的字符和数字。而访问键(记忆键)则我们在菜单中经常提到的与Alt键一起组合按下时,可以访问弹出式菜单项或对话框控制。由于可能与加速键间产生冲突,我们一般应避免将加速键定义为Alt键加上一个字母或数字键。模式键指的是封锁键与修改键。它们改变其它键或输入设备的动作。从IBM PC机继承下来的标准Windows兼容键盘有三个封锁键: Caps Lock, Num Lock 和 Scroll Lock。而修改键 Shift, Ctrl和Alt使可用于定义的加速键成数倍的增长。任何在其它几类中定义的键都可以作为快捷键(包括功能键F1至F12,光标移动

键Home, 向上箭头, 向下箭头等)。用这些 键与修改键结合, 就能够产生许多加速键。当然, 我想, 使用104键键盘的用户一定也会注意到键盘上的Windows键及Application键, 其中Windows键为一附加的修改键, 为操作系统保留, 而Application键则在程序中用于引导出上下文菜单。

接下来我们看一看定义加速键的方法。这包含两个步骤。

首先, 我们需要创建一个加速键资源。

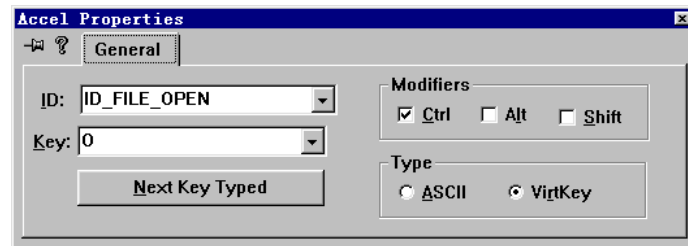


图5. 23 加速键表编辑器

图5.23为系统显示的编辑加速键表时的特性对话框。其本身包含三列：ID，键以及类型。命令ID标识了当用户按下一个加速键时通过WM\_COMMAND消息所传送的命令码。我们已经知道，系统调用消息映射进行识别是通过消息的ID号辨识的，因此，如果两条消息的ID号相同，它们所调用的消息处理函数就是相同的。形象一些地说，如果我们的程序中有菜单项与加速键所对应的ID号均为ID\_TITLE\_HELLO，那么，该加速键与该菜单项所完成的动作就是一样的。键域则标识了加速键所对应的键盘键。对于数字键或字母键，只需要简单的键入即可，而对于非打印字符，则需要使用虚拟码。当然，我们觉

得在弄不清楚虚拟键的具体使用时，一个最省属的办法还是单击Next Key Typed，然后直接指定我们所需要的键（此时系统会自动帮助我们采集消息并填入个域中）。如果不是使用上面的方法，我们所还需要做的最后一件事就是填写修改键域。按照Windows的建议，我们应首先考虑使用Ctrl键，然后再使用其它键。虽然使用Type域中ASCII选项以便区分所指定的键的大小写是可能的，然而，绝大多数的用户会对如此严格要求的键厌烦的，进一步的后果当然就是放弃该程序。所以，使用VirtKey几乎总是合适的选择。



图5. 24 加速键提示设计

绝大多数时候我们需要为加速键创建一个菜单提示。这一步在前面的菜单设计中我们实际上已经遇到过，我们一般是在具有加速键的菜单项右侧显示的。如图5.24中，在Caption域中，

Say &Hello\tCtrl+H

的前一部分指示菜单项，“\t”后面的部分则指定了该菜单项的加速键是Ctrl+H。

- 注意
- 初学者往往认为在菜单项中作上述设定以后就同时指定了相应的加速键，而不用加速键编辑器进行指定。那么，菜单项中所指定的加速键实际上是不开用的。

对没有相应的菜单项与之相对应的加速键，我们应通过一些途径（例如放在帮助文件中）告诉用户。（除非你本来就没打算让用户知道。）

- 我们还需要再重新提一下加速键定义的原则(参考<<Windows95用户界面设计指南>>)：

尽可能指派用户容易操作的单键

使得修改 - 字母 - 键组合不是特别敏感（避免混乱）。

对不使用Shift键的键或键组合动作的扩展和补充动作最好使用Shift键组合。例如，如果使用Alt+Tab从上到下选择控件焦点，那么使用Shift+Alt+Tab键组合就是一个很不错的安排。

对作用范围较大的动作使用Ctrl键组合。例如，在大家熟悉的记事本中，使用Home来移动到客户编辑区的光标所在行开始

处，而使用Ctrl+Home键从编辑区的任一位置移动到文本的开始处。

避免使用Alt键组合，因为它容易与标准的键盘对菜单和控制的访问项冲突。另外，一些Alt的组合键已经留给Windows系统调用。

尽量指定用户熟悉的组合键。对Windows程序中，几乎很少有程序指定除Ctrl+C别的键来作为“复制”命令的加速键 - 虽然并不是不可以这样。

应在可能的情况下让用户能够修改程序中的快捷键。（利用热键控件几乎总是一个可行的选择。）

通常总是使用Esc键来终止一个正在处理的功能或操作。

作为参考，我们给出一部分Microsoft Windows所使用的标准键盘命令于表5.3中。

表5.3 Windows系统所使用的标准键盘命令

范 畴	键 入	描 述
系统命令	Ctrl+Alt+Del	Windows95/NT注册/注销
	Alt+Tab	选择下一应用程序
	Alt+Esc	选择下一活动的应用程序
	Ctrl+Esc	显示Windows95开始菜单
	Alt+Space	打开Windows95系统菜单
	PrtScr	屏幕快照至剪贴板
	Alt+PrtScr	当前活动快照至剪贴板
	Alt	激活/撤消应用程序菜单条
	Alt+Enter	封锁DOS for Windows全屏正文模式
	Esc	取消当前模式或操作
	Enter	对话框中缺省项选择
	Tab	对话框中选择下一控制

应用程序模式	Alt+F4	关闭当前活动的最顶层窗口
	Application键	显示一上下文菜单（104键盘中）
	Shift+F10	显示一上下文菜单
	F1	调用系统帮助
	Shift+F1	调用上下文帮助
文件命令	Ctrl+N	File New
	Ctrl+O	File Open
	Ctrl+P	File Print
	Ctrl+S	File Save
剪贴板命令	Ctrl+Z	Edit Undo
	Alt+BackSpace	Edit Undo(向Windows3.x兼容)
	Ctrl+X	Edit Cut

续表5.3

剪贴板命令	Shift+Del	Edit Cut(向Windows3.x兼容)
	Ctrl+C	Edit Copy
	Ctrl+Ins	Edit Copy(向Windows3.x兼容)
	Ctrl+V	Edit Paste
	Shift+Ins	Edit Paste(向Windows3.x兼容)
MDI命令	Ctrl+F4	关闭当前活动文档窗口
	Ctrl+F6	激活下一文档窗口
	Shift+Ctrl+F6	激活前一文档窗口

在本节的最后，我们简单的谈一下多重加速键表的问题。在所有加速键函数中，Win32 API只有六个加速键函数。（MFC中没有封装加速

键函数就是因为原因的加速键函数太少)。其中函数LoadAccelerator用于从加速键资源中创建一个常驻RAM的加速键表，而函数TranslateAccelerator测试一个特定的键盘消息是否与一个加速键表命令入口相对应。限于篇幅的原因，在这里我们仅仅简单的作一介绍，更详细的信息可以从Visual C++的帮助文件中获得。顺便说一下，可以调用函数PreCreateWindow来在窗口显示以前装入一个加速键表装入内存。此后我们可以用对函数PreTranslateMessage的调用保证加速键表在适当的时候被调用。例如：

```
BOOL DWindowingClass::PreTranslateMessage(MSG *pMsg)

{

if(d_hAccel==NULL)

return FALSE;//我们不处理

return ::TranslateAccelerator(m_hWnd,d_hAccel,pMsg);

}
```

作为对本章中键盘消息响应的结束，我们给出一个有趣的例程，读者不妨运行之，看看是否很有意思。

```
// echofill.h : main header file for the ECHOFILL application

//

#ifdef __AFXWIN_H__

#error include 'stdafx.h' before including this file for PCH

#endif

#include "resource.h" // main symbols

////////////////////////////////////

// DApp:

// See echofill.cpp for the implementation of this class

//

class DApp : public CWinApp

{

public:
```



```

DApp();

// Overrides

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(DApp)

public:

virtual BOOL InitInstance();

//}}AFX_VIRTUAL

// Implementation

//{{AFX_MSG(DApp)

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()

};

////////////////////////////////////

//{{NO_DEPENDENCIES}}

// Microsoft Visual C++ generated include file.
// Used by ECHOFill.RC

//

#define IDR_MAINFRAME 128

#define IDD_ABOUTBOX 100

// Next default values for new objects

//

#ifdef APSTUDIO_INVOKED

#ifdef APSTUDIO_READONLY_SYMBOLS

```

```

#define _APS_3D_CONTROLS 1

#define _APS_NEXT_RESOURCE_VALUE 130

#define _APS_NEXT_CONTROL_VALUE 1000

#define _APS_NEXT_SYMED_VALUE 101

#define _APS_NEXT_COMMAND_VALUE 32771

#endif

#endif

// echofill.cpp : Defines the class behaviors for the application.

//

#include "stdafx.h"

#include "echofill.h"

#include "mainfrm.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// DApp

BEGIN_MESSAGE_MAP(DApp, CWinApp)

//{{AFX_MSG_MAP(DApp)

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)

END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////

// DApp construction

DApp::DApp()

{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////

// The one and only DApp object

DApp theApp;

/////////////////////////////////////////////////////////////////

// DApp initialization

BOOL DApp::InitInstance()

{
    // Step 1: Allocate C++ window object.

    DMainFrame * pFrame;

    pFrame = new DMainFrame();

    // Step 2: Initialize window object.

    pFrame->LoadFrame(IDR_MAINFRAME);

    // Make window visible

    pFrame->ShowWindow(m_nCmdShow);

    // Assign frame as application's main window

    m_pMainWnd = pFrame;

    return TRUE;

```

```

}

/////////////////////////////////////////////////////////////////

// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

    // Dialog Data
   //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

   //{{AFX_MSG(CAboutDlg)
    // No message handlers
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{

```

```

CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)

//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)

//{{AFX_MSG_MAP(CAboutDlg)

// No message handlers

//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void DApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////

// DApp commands

// mainfrm.cpp : implementation of the DMainFrame class
//

#include "stdafx.h"
#include "echofill.h"
#include "mainfrm.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

```

```

/////////////////////////////////////////////////////////////////

// DMainFrame

IMPLEMENT_DYNCREATE(DMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(DMainFrame, CFrameWnd)

//{{AFX_MSG_MAP(DMainFrame)

ON_WM_PAINT()

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////

// DMainFrame construction/destruction

DMainFrame::DMainFrame()

{

ASSERT (MAX_COLORS == 8); // Table fill assumes this many colors.

crBackground[0] = g_crGray;

crBackground[1] = g_crYellow;

crBackground[2] = g_crRed;

crBackground[3] = g_crBlue;

crBackground[4] = g_crGreen;

crBackground[5] = g_crWhite;

crBackground[6] = g_crBlack;

crBackground[7] = g_crCyan;

iNextColor = 0;

lpszClassName = _T("Afx:No:Redraw:Bits");

}

DMainFrame::~DMainFrame()

{

```

```

}

//-----

// PreCreateWindow -- Called before Windows API window is created.

BOOL DMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // Register our own window class to avoid the default
    // class style of CS_HREDRAW | CS_VREDRAW;
    HICON hIcon = ::LoadIcon(AfxGetResourceHandle(),
    MAKEINTRESOURCE(IDR_MAINFRAME));
    HCURSOR hCursor = ::LoadCursor((HINSTANCE)0, IDC_ARROW);
    LPCTSTR lpszClassName =
    AfxRegisterWndClass(0, // Class style.
    hCursor, // Mouse cursor.
    (HBRUSH)COLOR_WINDOW+1, // Background color.
    hIcon); // Icon.
    // Pass class name on to creation code.
    cs.lpszClass = lpszClassName;
    return CFrameWnd::PreCreateWindow(cs);
}

////////////////////////////////////

// DMainFrame diagnostics
#ifdef _DEBUG
void DMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

```

```

void DMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////

// DMainFrame message handlers

void DMainFrame::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // Query size of window's client area.
    CRect rClient;
    GetClientRect(&rClient);

    // Select next color for background
    ASSERT(iNextColor >= 0 && iNextColor < MAX_COLORS);
    dc.SetBkColor(crBackground[iNextColor]);

    // Fill client area with background color.
    dc.ExtTextOut(0, 0, ETO_OPAQUE, &rClient, 0, 0, 0);

    // Increment color index and wrap to start of range.
    iNextColor++;

    if (iNextColor >= MAX_COLORS) iNextColor = 0;
}

//Microsoft Visual C++ generated resource script.
//

#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

```



```
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
"resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
#include "\"afxres.h\""\r\n"
"\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
#include "\"res\\echofill.rc2\"" // non-Microsoft Visual C++ edited resources\r\n"
"\r\n"
```

```
"#define _AFX_NO_SPLITTER_RESOURCES\r\n"

"#define _AFX_NO_OLE_RESOURCES\r\n"

"#define _AFX_NO_TRACKER_RESOURCES\r\n"

"#define _AFX_NO_PROPERTY_RESOURCES\r\n"

"#include "afxres.rc" \011// Standard components\r\n"

"\0"

END

////////////////////////////////////

#endif // APSTUDIO_INVOKED

////////////////////////////////////

//

// Icon

//

IDR_MAINFRAME ICON DISCARDABLE "res\\echofill.ico"

////////////////////////////////////

//

// Menu

//

IDR_MAINFRAME MENU PRELOAD DISCARDABLE

BEGIN

POPUP "&File"

BEGIN

MENUITEM "E&xit", ID_APP_EXIT

END

POPUP "&Help"
```

BEGIN

MENUITEM "&About echofill...", ID\_APP\_ABOUT

END

END

//

//

// Dialog

//

IDD\_ABOUTBOX DIALOG DISCARDABLE 34, 22, 217, 55

STYLE DS\_MODALFRAME | WS\_POPUP | WS\_CAPTION | WS\_SYSMENU

CAPTION "About echofill"

FONT 8, "MS Sans Serif"

BEGIN

ICON IDR\_MAINFRAME, IDC\_STATIC, 11, 17, 20, 20

LTEXT "echofill Version 1.0", IDC\_STATIC, 40, 10, 119, 8

LTEXT "Copyright \251 1995", IDC\_STATIC, 40, 25, 119, 8

DEFPUSHBUTTON "OK", IDOK, 176, 6, 32, 14, WS\_GROUP

END

//

//

// Version

//

VS\_VERSION\_INFO VERSIONINFO

FILEVERSION 1,0,0,1

```
PRODUCTVERSION 1,0,0,1

FILEFLAGSMASK 0x3fL

#ifdef _DEBUG

FILEFLAGS 0x1L

#else

FILEFLAGS 0x0L

#endif

FILEOS 0x4L

FILETYPE 0x1L

FILESUBTYPE 0x0L

BEGIN

BLOCK "StringFileInfo"

BEGIN

BLOCK "040904B0"

BEGIN

VALUE "CompanyName", "\0"

VALUE "FileDescription", "ECHO FILL MFC Application\0"

VALUE "FileVersion", "1, 0, 0, 1\0"

VALUE "InternalName", "ECHO FILL\0"

VALUE "LegalCopyright", "Copyright \251 1995\0"

VALUE "LegalTrademarks", "\0"

VALUE "OriginalFilename", "ECHO FILL.EXE\0"

VALUE "ProductName", "ECHO FILL Application\0"

VALUE "ProductVersion", "1, 0, 0, 1\0"

END

END
```

```

BLOCK "VarFileInfo"

BEGIN

VALUE "Translation", 0x409, 1200

END

END

////////////////////////////////////

//

// String Table

//

STRINGTABLE PRELOAD DISCARDABLE

BEGIN

IDR_MAINFRAME "EchoFill\n\nEchofi\n\n\nEchofill.Document\nEchofi Document"

END

STRINGTABLE PRELOAD DISCARDABLE

BEGIN

AFX_IDS_APP_TITLE "EchoFill"

END

#ifndef APSTUDIO_INVOKED

////////////////////////////////////

//

// Generated from the TEXTINCLUDE 3 resource.

//

#include "res\echofill.rc2" // non-Microsoft Visual C++ edited resources

#define _AFX_NO_SPLITTER_RESOURCES

#define _AFX_NO_OLE_RESOURCES

#define _AFX_NO_TRACKER_RESOURCES

```

```
#define _AFX_NO_PROPERTY_RESOURCES

#include "afxres.rc" // Standard components

////////////////////////////////////

#endif // not APSTUDIO_INVOKED
```

## 第四节 滑块控件消息响应

滑块控件 (Slider, 有时也称为TrackBar) 是这样一种控件, 通过它我们可以较直观地设置程序要求的用户输入, 而在另一些情况下, 则可以由我们控制用户输入的取值范围, 步长等相关信息。在作进一步介绍之前, 让我们先看看滑块控件的外观以及本节的一个例程的运行画面。

在Visual C++工具栏中, 滑块控件图标如图5.25所示。单击此图标后拖动至目标位置

即可。需要注意的是如图5.27滑块控件的风格设置。如图5.26例程的运行画面, 其风格设置为: Auto\_Ticks (自动刻度), Enable Selection (选择使能)。扩展风格为Transparent (透明)。在接下来的讲解中, 我们将具体说明各风格值的设置的具体作用。

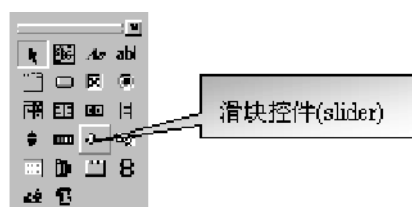


图5. 25 滑块控件图标

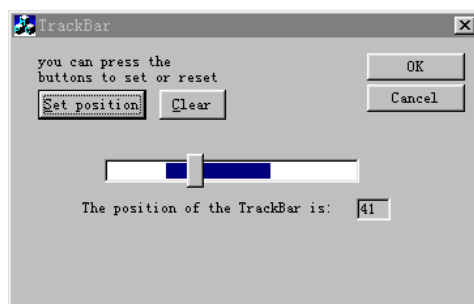


图5. 26 设置滑块控件范围后运行一画面

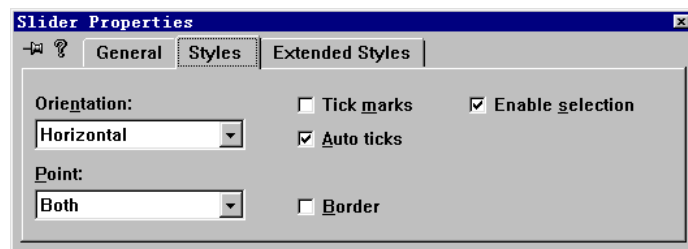


图5. 27 滑块控件风格设置

如图5.27所示，在设计中，我们可以设置滑块控件的诸多属性，而这些属性对于不同的应用程序，所展示的效果是不同的。在Styles选项卡中，域Orientation中方向的设置，不但使其具有不同的外观，也影响了我们在程序中所要处理的消息的不同：选择Horizontal时，我们要处理消息WM\_HSCROLL，而当我们选择Vertical时，我们所要处理的消息则变成了WM\_VSCROLL。当然，它们的处理过程从实际看来，也并没有本质上的差别。域Point的选择则使滑块控件滑块具有不同的外观。剩下的几个选项则更多的是一些决定滑块控件外观及工作形式上的一些区别。选择Tick marks滑块控件上下（或左右）两端具有的刻度；选择Auto ticks时，滑块控件外侧中央具有刻度（但此时Tick marks必须被选中，否则还是没有，比如例程中就设定了该风格，但由于没有设定Tick marks，程序没有刻度），Border选项在滑块控件四周画一边框，而Enable selection时，使我们可以指定可以在程序中进行选择的范围（在我们的例程中，就指定了该风格，从而使我们可以指定该滑块控件的可选范围为一较小的值）。Extended Styles选项卡大致与前面所讲的相近，但选项Accept Files, No parent Notify则主要与滑块控件的消息发送有关。

下面我们结合例程来看一看几个有关于滑块控件的函数的作用。

- 注意：
- 由于单一滑块控件发送的消息主要是WM\_HSCROLL或WM\_VSCROLL，因此我们所要处理的消息也就只有该消息。
- 上下控件、进度条与滑块控件的几个函数的形式大致相同，同时它们也具有大致相识的功能。因此掌握了有关滑块控件的几个函数基本上也就掌握了它们的相关函数。

在此我们将程序中较关键的一段代码列于本节末尾，同时，我们在正文中结合滑块控件的函数处理作介绍，因此，在文末的代码中，我们不再给出注释。但我们注意将需手工加入的代码特意作出了标记，读者可以对照查阅。

首先，我们需要创建滑块控件。在对话框程序中，滑块控件可以在资源编辑器中可视地生成（我们的程序中即是这样创建的），但在并非基于对话的程序中，我们则可能需要在程序中动态地完成。这时我们应该调用滑块控件的生成函数CSliderCtrl::Create来完成：

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

该函数中参数dwStyle标识了该滑块控件的风格值，在上面可视地创建滑块控件的过程中，我们实际上以作了部分介绍。它们可能的取值为下列风格（或其组合）：

TBS_HORZ：	将滑块控件方向设为水平方向（缺省设置：如不指定，系统认为这就是轨道条的设置）。
TBS_VERT：	将滑块控件方向设为竖直方向。
TBS_AUTOTICKS：	程序所生成的滑块控件在其取值范围内对每一个增长标有小刻度。这些小刻度在程序调用SetRange时自动生成。但设置此风格后在你的程序中除非调用了函数ClearTicks，否则你对函数SetTic，SetTicFreq是无效的。
TBS_NOTICKS：	生成的滑块控件隐去小刻度现。
TBS_BOTTOM：	仅在水平滑块控件的底端出现刻度（同TBS_TOP同时使用则可以在轨道条的上下皆出现刻度。）
TBS_TOP：	仅在水平滑块控件的顶端出现刻度（同TBS_BOTTOM同时使用则可以在轨道条的上下皆出现刻度。）
TBS_RIGHT：	仅在竖直滑块控件的右端出现刻度（同TBS_LEFT同时使用则可以在滑块控件的左右皆出现



	刻度。 )
TBS_RIGHT :	仅在竖直滑块控件的左端出现刻度 ( 同TBS_RIGHT同时使用则可以在滑块控件的左右皆出现刻度。 )
TBS_BOTH :	水平/竖直滑块控件的上下/左右皆出现刻度。
TBS_ENABLESELRANGE :	显示一可选范围。

该函数的第二个参数指明滑块控件控制的大小和位置 ( 通过该矩形确定 )。参数pParentWnd指明该滑块控件所属的父窗口 ( 一定非空 ! 但是一般为一对话框 )。该函数的最后一个参数nID则明确指出该滑块控件的ID号。

- 注意 :
- 滑块控件的生成也应该使用一般的两步法 : 首先生成一对象 , 系统调用该对象的构造函数CSliderCtrl , 然后调用函数Create将滑块控件与一CSliderCtrl联系起来。

接下来我们所要涉及的就是一些有关滑块控件控制操作的成员函数。

与滑块控件位置操作有关的函数主要有三个 :

SetRange : 设定滑块控件的变化范围。

void SetRange( int nMin, int nMax, BOOL bRedraw = FALSE ) : 函数前两个

参数指定滑块控件变化范围的最小 , 最大值。最后一个参数则通知系统滑块控件在执行该函数后是否需要重绘。当其为TRUE时应用程序重绘滑块控件 , 为FALSE时则不重绘。

SetTicFreq : 设置滑块控件的一个刻度所对应的步长 ( 系统缺省时为一 )。

void SetTicFreq( int nFreq ) : 函数的唯一参数nFreq指定滑块控件的每一小刻度所对应的变化。( 在一些情形下 , 缺省的为一的步长可能相当不合适。此时我们应调用该函数进行重新设定。但最好不

要将每一步长设定的太大或太小。 )

SetPos : 指定滑块控件滑块的位置。

void SetPos( int nPos ) : 参数nPos指定滑块控件滑块的新位置并将滑块移至该位置。

SetSelection : 指定滑块控件中当前可选的范围 ( 在当前滑块控件中指定。 )

void SetSelection( int nMin, int nMax ) : 二参数指定变化范围的最小, 最大值。如果需要在设置时便显示选择的范围, 应该在调用该函数后立即调用函数Invalidate对滑块控件进行重绘。否则, 滑块控件在设定选择范围后选择范围并不在滑块控件中显示。(但在窗口重绘时该范围将被指示出。)

ClearSelection : 清除选择范围。

void ClearSel( BOOL bRedraw = FALSE ) : bRedraw指明在清除选择范围后滑块控件是否重绘。当其为TRUE时选择范围的失去立即显示出来, 为FALSE时则选择范围指示只能在窗口进行重绘时被清除。

在本节的最后, 我们讲一下滑块控件的消息处理。在前面的介绍中, 我们已经提到, 对于滑块控件我们需要处理的消息很少。通过Spy++监视我们可以看出的确如此。下面, 我们就来看看消息WM\_HSCROLL ( 对水平滑块控件, 对竖直滑块控件消息应为WM\_VSCROLL, 但由于它们的处理近似, 在下面的介绍中我们仅以水平滑块控件为例进行讲解 ) 的处理。

首先我们必须辨别清楚消息WM\_HSCROLL是否为我们所期望的滑块控件发出。在例程中, 我们使用了下面的代码进行识别:

```
// ...  
  
CSliderCtrl *pSlider=(CSliderCtrl *)pScrollBar;  
  
if(pSlider==&m_TrackBar)  
  
// ...
```

在这段代码中, 我们首先对程序发送的消息进行转换, 这在第一条语句中完成。它将该消息转换为工具条消息。然后, 在紧跟着的一条语句中, 我们判断该滑块控件消息究竟是那一个滑块控件发出的。(毕竟, 在该窗口中, 很有可能不止一个滑块控件!)

接下来，我们就可以对该消息进行分类处理了。在我们的程序中，为了简单起见，对每一个消息，我们简单的调用宏TRACE将用户的选择输出。在用户的程序中，可能需要对它们进行别的更有意义的处理，在这里就不详述了。至于对该控制的各个消息的具体含义，读者结合其名称自不难理解。

- 注意：
- 宏TRACE有些类似于C语言的标准格式化输出函数printf，它能一次输出不超过256个字符（超过此限制时导致一个ASSERT被发出）。利用它，我们可以很方便地对程序进行调试。但需要注意的是，该宏仅仅在程序调试状态下起作用，在程序的发行版中，该宏被展开为空。

```
// TrackBarDlg.cpp : implementation file
//

#include "stdafx.h"

// ....

//以上部分为AppWizard生成的标准代码，从略

// ...

BEGIN_MESSAGE_MAP(CTrackBarDlg, CDialog)

//{{AFX_MSG_MAP(CTrackBarDlg)

ON_WM_SYSCOMMAND()

ON_WM_PAINT()

ON_WM_QUERYDRAGICON()

ON_WM_HSCROLL()

ON_BN_CLICKED(IDC_CLEAR_BUTTON, OnClearButton)

ON_BN_CLICKED(IDC_SET_BUTTON, OnSetButton)

//以上三处为我们通过ClassWizard加入的消息响应

//}}AFX_MSG_MAP

END_MESSAGE_MAP()
```

```

// CTrackBarDlg message handlers

BOOL CTrackBarDlg::OnInitDialog()

{
CDialog::OnInitDialog();

// Add "About..." menu item to system menu.

// IDM_ABOUTBOX must be in the system command range.
ASSERT((IDM_ABOUTBOX & 0xFFFF0) == IDM_ABOUTBOX);
ASSERT(IDM_ABOUTBOX < 0xF000);

// ...

SetIcon(m_hIcon, FALSE); // Set small icon

// TODO: Add extra initialization here

m_TrackBar.SetRange(10,100);

m_TrackBar.SetTicFreq(10);

m_TrackBar.SetPos(10);

char szBuffer[81];

sprintf(szBuffer,"10");

m_EditControl.SetWindowText(szBuffer);

//此处为编辑框控制处理，在学习完第四章后，读者应该已经能理解上述代码

return TRUE; // return TRUE unless you set the focus to a control
//Exception:OCX Property Pages should return FALSE
}

//下面为一系统消息处理范例

void CTrackBarDlg::OnSysCommand(UINT nID, LPARAM lParam)

{

if ((nID & 0xFFFF0) == IDM_ABOUTBOX)

{

```

```

CAboutDlg dlgAbout;

dlgAbout.DoModal();

}

else

{

CDialog::OnSysCommand(nID, lParam);

}

}

// If you add a minimize button to your dialog, you will need the code below
// to draw the icon. For MFC applications using the document/view model,
// this is automatically done for you by the framework.

void CTrackBarDlg::OnPaint()

{

if (IsIconic())

{

// ....

}

else

{

// ...

}

//假如在滑块控件中没有在设置范围后立即重绘，建议读者仔细看一下上面代码

}

// The system calls this to obtain the cursor to display while the user drags
// the minimized window.

HCURSOR CTrackBarDlg::OnQueryDragIcon()

```

```

{
return (HCURSOR) m_hIcon;
}

void CTrackBarDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
// TODO: Add your message handler code here and/or call default
//the trackbar sends an WM_HSCROLL(or WM_VSCROLL if the trackbar is vertical)
//message telling the position of the slider has changed!
//this iagnostic message shows what parameters are passed to our OnHScroll()
//handler

char szBuffer[81];
CSliderCtrl *pSlider=(CSliderCtrl *)pScrollBar;
if(pSlider==&m_TrackBar)
{
switch(nSBCode)
{
case TB_BOTTOM:TRACE("TB_BOTTOM \n");break;
case TB_TOP:TRACE("TB_TOP\n");break;
case TB_ENDTRACK:TRACE("TB_ENDTRACK \n");break;
case TB_LINEDOWN:TRACE("TB_LINEDOWN \n");break;
case TB_LINEUP:TRACE("TB_LINEUP \n");break;
case TB_PAGEDOWN:TRACE("TB_PAGEDOWN \n");break;
case TB_PAGEUP:TRACE("TB_PAGEUP \n");break;
case TB_THUMBPOSITION:TRACE("TB_THUMBPOSITION \n");break;
case TB_THUMBTRACK:TRACE("TB_THUMBTRACK \n");break;
default:

```

```

TRACE("default:(error) \n");

break;

}

sprintf(szBuffer,"%d",pSlider->GetPos());

m_EditControl.SetWindowText(szBuffer);

}

else

{///It's some other scrollbar in the programe!

}

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}

void CTrackBarDlg::OnClearButton()

{

// TODO: Add your control notification handler code here

m_TrackBar.ClearSel(TRUE);

}

void CTrackBarDlg::OnSetButton()

{

// TODO: Add your control notification handler code here

m_TrackBar.SetSelection(30,70);

m_TrackBar.Invalidate();

}

```

## 第五节 进度条消息响应

进度条（ProgressBar）是应用程序中另一种常见的控件。它一般被用于在程序中完成较大的任务时，粗略地指示该任务的进展。这是一种在Windows 95中新增加的控件类型，因此，类CProgressCtrl只有在Windows 95或Windows NT 3.51及其以后的版本下可用。进度条体

现了Windows 95程序界面中更贴近人性化的一部分。它具有一个通常情况下充满显示的矩形框，并用一种系统的高亮色指示当前任务的完成情况。它具有一个代表整个任务完成情况的指定的范围，同时，它也具有一个确定的代表任务已经完成情况的当前位置。窗口过程调用进度条的范围及当前位置确定任务当时完成情况的百分比。当然，如果在需要的情况下，可将该值进行显示。由于位置值使用无符号整数，进度条的最高可能范围为65535。图5.28给出了在Visual C++中的资源编辑器中的进度条的图标。同时，由于进度条与滑块控件之间的相识性（在前面我们已经提到过，它们具有相同的基类，读者不妨回头看看本章第三节开始时的介绍。），我们在本节中将尽量简单地分析有关一些常用的进度条的相关成员函数的用法，最后，我们将给出一个简单的综合了上一节有关滑块控件以及本节的进度条的用法，给出一个稍微综合的例程，程序的一个运行画面见图5.29。



图5. 28 资源编辑器中的进度条图标

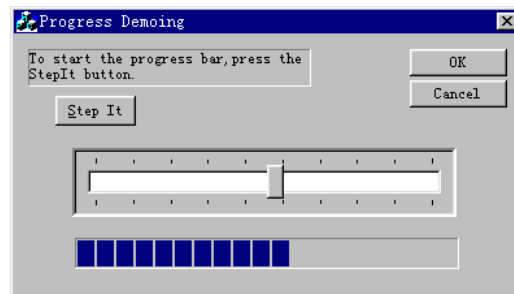


图5. 29 例程的一个运行画面

下面我们首先谈谈在Visual C++的资源编辑器中进度条的可视实现。进度条与滑块控件的实现相近。生成的方法也类似。但进度条的样式（Style）选项卡的内容远较滑块控件为少，只有边框（Border）一项可选。而其扩展风格(Extended Styles)的用法与含义与进度条相近，读者可参见前文所介绍内容，此处从略。

与滑块控件相似，进度条具有几个相类似的成员函数供选择操作。在这里，我们仅仅列出各函数，其具体用法我们结合实例在例程中作对照分析。

设定范围：`void SetRange( int nLower, int nUpper );`



设定步长：`int SetStep( int nStep );`其返回值为进度条先前步长

设定当前位置：`int SetPos( int nPos );`其返回值为进度条当前发生位置改变时的位置

前进一个步长并即时刷新窗口：`int StepIt;`它与SetStep函数的一个重要区别是其

前进值确定。函数返回值为进度条发生步进前位置。

与前文所述类似，在非基于对话框的程序中使用进度条时，我们需要先创建该对象，这可以通过对类CProgressCtrl成员函数Create的调用来完成：

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

参数意义与滑块控件所具有的参数的意义相近，读者不妨参见上节中所讲的内容。

- 注意：

- 在下面的例程中，我们基本上简单地综合了滑块控件与进度条的内容，但出于篇幅的原因，我们仅仅列出了其中添加较多的文件ProgressDlg.cpp。

```
// ProgressDlg.cpp :实现文件
//
#include "stdafx.h"
// ...
//以上部分多为各程序中相同的部分，从略
// ...
BEGIN_MESSAGE_MAP(CProgressDlg, CDialog)
//{{AFX_MSG_MAP(CProgressDlg)
ON_WM_SYSCOMMAND()
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
```

```

ON_BN_CLICKED(IDC_STEPIT, OnStepit)

ON_WM_HSCROLL()

//滑块控件所需要处理的唯一一个消息。

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

////////////////////////////////////

// CProgressDlg message handlers

BOOL CProgressDlg::OnInitDialog()

{
CDialog::OnInitDialog();

// Add "About..." menu item to system menu.

// IDM_ABOUTBOX must be in the system command range.

ASSERT((IDM_ABOUTBOX & 0xFFFF0) == IDM_ABOUTBOX);

ASSERT(IDM_ABOUTBOX < 0xF000);

CMenu* pSysMenu = GetSystemMenu(FALSE);

if (pSysMenu != NULL)

{
CString strAboutMenu;

strAboutMenu.LoadString(IDS_ABOUTBOX);

if (!strAboutMenu.IsEmpty())

{
pSysMenu->AppendMenu(MF_SEPARATOR);

pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);

}

}

// Set the icon for this dialog. The framework does this automatically

```

```

// when the application's main window is not a dialog

SetIcon(m_hIcon, TRUE); // Set big icon

SetIcon(m_hIcon, FALSE); // Set small icon

// TODO: Add extra initialization here

m_TrackBar.SetRange(10,100);

m_TrackBar.SetTicFreq(10);

m_TrackBar.SetPos(15);

//以上部分设定滑块控件的一些重要信息。它们的顺序是无关紧要的。

//但先设定范围，再设定刻度频率，最后设定滑块控件当前位置总是一个较好的习惯。

m_ProgressBar.SetRange(10,100); //设定进度条范围为10至100

m_ProgressBar.SetStep(1); //设定函数StepIt ( ) 调用时，每前进一个步长的长度为1

m_ProgressBar.SetPos(10); //设定进度条当前位置为10，在程序中，进度条开始时的位置

//一般为其开始位置，但程序根据需要进行变动也无妨

//以上部分设定进度条的一些重要信息。它们的顺序也是无关紧要的。

//但先设定范围，再设定步长，最后设定进度条当前位置是一个较好的习惯。

return TRUE; // return TRUE unless you set the focus to a control
}

void CProgressDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFFF) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {

```

```

CDialog::OnSysCommand(nID, lParam);

}

// If you add a minimize button to your dialog, you will need the code below
// to draw the icon. For MFC applications using the document/view model,
// this is automatically done for you by the framework.

void CProgressDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle

        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);

        CRect rect;

        GetClientRect(&rect);

        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon

        dc.DrawIcon(x, y, m_hIcon);
    }

    else
    {
        CDialog::OnPaint();
    }
}

```

```

//程序据此函数进行窗口的重绘
}

// The system calls this to obtain the cursor to display while the user drags
// the minimized window.
HCURSOR CProgressDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

void CProgressDlg::OnStepIt()
{
    // TODO: Add your control notification handler code here
    m_ProgressBar.SetStep(1);
    //设定函数StepIt ( ) 的步长，一般来说，使用步长宜适中。
    //在设定的步长大于刻度所代表的长度时，需执行几次函数StepIt ( ) 进度条的滑块
    //才会前进一个刻度距离。而当其大于一个刻度所代表的长度时，每执行一次函数
    //StepIt ( ) 滑块就会前进不止一个刻度距离。但每个刻度只有在被完全跨越时滑块
    //在外观上才前进一个刻度。
    m_ProgressBar.StepIt();
    //若函数执行时滑块已经到达其可能达到的最大位置，则执行后滑块重新达到其
    //最小位置，在例程中读者不妨实际执行之以得到一直观印象。
    //按钮Step It的消息处理函数。
}

void CProgressDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    CSliderCtrl *pSlider=(CSliderCtrl *)pScrollBar;

```

```

if(pSlider==&m_TrackBar)

m_ProgressBar.SetPos(pSlider->GetPos());

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}

```

//滑块控件需要处理的唯一消息，从这里我们可以看出，在例程中，我们根据滑块控件的位置

//来设定进度条的当前位置。换句话说，我们让滑块控件与进度条在程序中具有了相同的意义。

## 第六节 上下控件消息响应

与其它的标准控件相比，上下控件（Spin）是一种更常用于对输入进行在一定范围内进行精确定位的控件。与滑块控件不同的是，滑块控件的定位是连续的，而上下控件的定位则在很大程度上可以被视为是离散的。滑块控件一般多用于在很大范围内进行快速的选择，而上下控件则多用在选择范围较小而对精度要求较苛刻的场合。同时，使用上下控件也可以将程序要求的用户输入精度明显地表示出来。图5.30为Visual C++的资源编辑器中的上下控件图标。

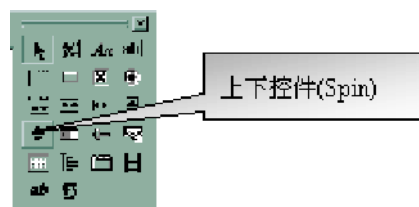


图5.30 资源编辑器中的上下控件图标

由于上下控件在程序控制中与滑块控件的诸多相识性，在本章中，我们将结合两者进行比较，同时考虑到上下控件在控制方式上的特殊性，我们会对其特性也进行简单的介绍。

首先，创建一个上下控件与创建其它控件在可视阶段并没有什么特殊之处。在基于对话框的程序中，将上下控件直接放到对话框中即可。在并非基于对话框的程序中，我们使用其生成函数Create即可：

```

BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );

```

函数的几个参数很容易理解。参数dwStyle指定上下控件的风格，rect则指定其大小和位置，参数pParentWnd为指向该控件的父窗口的指针，在对话框程序中，简单地设为NULL在很多场合就足够了，函数最后一个参数nID则指定控件的ID号。

但是，对于上下控件的风格（Style）的设置则对程序控制具有很大的影响。随参数dwStyle取值的变化，上下控件的行为具有相当大的区别。下面我们结合资源编辑器中的可视实现对其作讲解。

如图5.31中，上下控件的风格选项卡具有七个不同的域。

- 注意：
- 下面的讲解中各选项后的括号中的值为对应该选项的风格值设定。

域Orientation决定上下控件的显示方式 同时决定其处理的消息的不同。选择Vertical时，控件按上下方式显示，而当选择Horizontal（UDS\_HORZ）时，它按水平方式显示。域Alignment则决定上下控件的关联窗口的布置方式。它可以为下列值之一：

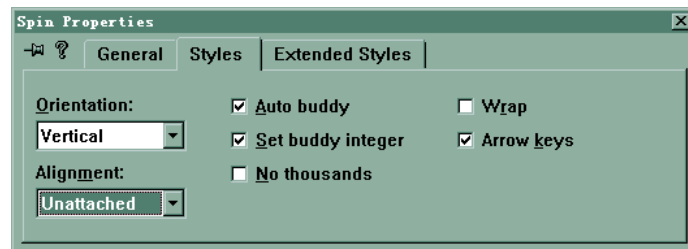


图5. 31 上下控件的风格设定

Unattached（default）：此为域缺省值，表明上下控件与其关联控件之间没有什么特定的要求。

Left（UDS\_ALIGNLEFT）：此时上下控件的关联控件的位置被指定为在上下控件的右边，同时，其关联控件的大小被调整到与其关联的上下控件等高。

Right（UDS\_ALIGNRIGHT）：此时上下控件的关联控件的位置被指定为在上下控件的左边，同时，其关联控件的大小被调整到与其关联的上下控件等高。

选项Auto buddy（UDS\_AUTOBUDDY）对程序控制影响较大。当该选项被选择时，系统自动设定该上下控件的关联窗口，同时，自动在该关联窗口中显示其所联系的上下控件位置值。（因此，该关联窗口一般为一编辑框或静态文本控件。）也就是说，设定该值后，上下控件的关联窗口的行为管理是自动的，在指定了上下控件的关联窗口后，就不再需要对其行为控制添加另外的代码。而当没有设定该风格时，我们必须为该控件的行为编写独立的代码：要考虑编辑框中值的显示，

要考虑刷新静态文本控件的内容。我们认为，如果仅仅是简单的利用上下控件的位置信息，一般设为自动处理方式就相当合适了。当然，如果程序员希望在其中处理一些额外的动作，就不得不动动脑筋考虑一下其实现方法了。

选项Set buddy integer ( UDS\_SETBUDDYINT ) 则决定了上下控件位置改变时的消息发送。设置该选项时，上下控件的位置改变是，向其关联窗口发送消息WM\_SETTEXT，同时，上下控件的位置以十进制或十六进制格式发出。

选项No thousands(UDS\_NOTHOUSANDS)被设置时，在数的每三位中，不插入一个起分隔作用的千位分隔符。

选项Wrap ( UDS\_WRAP ) 决定当上下控件的选择位置超过极限位置时的处理方式：当该选项被设置时，当上下控件的位置达到其最大或最小位置后，用户试图再增大或减小该值时，上下控件的值变为最小或最大。否则，上下控件的值只是简单的保持不变。

选项Arrow keys ( UDS\_ARROWKEYS ) 被设置时，当按下向上或向下方箭头时，控制增加或减小其位置。

上下控件的扩展风格的设定与其它控件的设定没有什么大的改变，这里不再详述。

下面，我们看一下上下控件的关联窗口的指定及其行为管理的实现。

首先，我们看看当上下控件的Auto buddy风格被设定的情况下的管理。

我们先指定上下控件的关联窗口。这需要用到函数SetBuddy：

```
CWnd* SetBuddy( CWnd* pWndBuddy );
```

该函数的唯一参数为一指向上下控件的关联窗口的指针，其返回值为指向

该上下控件原来的关联窗口的指针。

- 注意：
- 在程序中，一般应先指定上下控件的变动范围SetRange：
- void SetRange( int nLower, int nUpper );



该函数的两个参数分别指定了上下控件的最小及最大变动值。

以后我们不再需要为上下控件的关联窗口的行为编写任何代码。系统会为我们作好所有的规范化的工作：上下控件的位置改变时，其变化直接反应在与其关联的窗口中。该窗口中会以十进制或十六进制数的形式来反应上下控件此时的位置信息。

当我们不设定该风格的时候，情况发生了变化。范围及关联窗口的设定同上文没有什么差别。但现在，当上下控件的位置发生变化时，虽然上下控件会发送消息反应这一变化，但其关联窗口的行为需要我们编写代码来实现。当上下控件发生的位置与其前一位置不一样时，它发送消息ON\_WM\_HSCROLL ( ON\_WM\_VSCROLL ) 来反应这一变化。在程序接收到该消息后，经过适当的处理，就可以自己对该消息作出反应了（当然，如果你愿意，你也可以什么事也不做）。

下面为一段关于上下控件的关联窗口控制的代码。

```
// ...

m_Spin2.SetRange(0,(sizeof(szColors)/sizeof(szColors[0]))-1);

m_Spin2.SetPos(0);

m_Spin2.SetBuddy(&m_Buddy2);

m_Spin2.SetWindowText(szColors[0]);

// ...

void CSpinBoxDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default

    CSpinButtonCtrl *pSpin=(CSpinButtonCtrl *)pScrollBar;

    if(pSpin==&m_Spin2)

        m_Buddy2.SetWindowText(szColors[pSpin->GetPos()]);

    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

第一段代码设定了上下控件的范围，关联窗口，位置，并初始化了编

辑框中的显示。

第二段中的函数处理了消息ON\_WM\_HSCROLL，并利用上下控件的位置对其关联窗口的显示进行了改变。语句

```
CSpinButtonCtrl *pSpin=(CSpinButtonCtrl *)pScrollBar;
```

将该消息的最后一个参数进行转换，以便确定是否该消息是我们所关心的上下控件发出的。接下来用语句

```
if(pSpin==&m_Spin2)
```

进行判断。在确知该消息是由特定的上下控件发出的后，语句

```
m_Buddy2.SetWindowText(szColors[pSpin->GetPos()]);
```

对编辑框中的文本进行了更新。先用上下控件的成员函数GetPos取得其位置，然后用函数SetWindowText重新设定编辑框中的文本（szColors[ ]为预先设定的一枚枚举数组）。

- 注意：
- 以上设定范围、关联窗口、位置（还有没有在上文提到的加速设置）动作，都有相应的获取函数：例如GetBuddy函数对应于SetBuddy等。读者有兴趣可以参考系统的技术支持部分。

## 第六章 使用Windows标准控件

我们在前面曾提到过，控件是一些行为标准化了的窗口，一般用于对话框或其它窗口中充当与用户交互的元素。在Visual C++中，可以使用的控件分成三类：

### (1) Windows标准控件

Windows标准控件由Windows操作系统提供，在Windows 95中还提供了一些新增的控件。所有这些控件对象都是可编程的，我们可以使用Visual C++提供的对话框编辑器把它们添加到对话框中。Microsoft基础类库(MFC)提供了封装这些控件的类，它们列于表6.1。

表6.1 Windows标准控件

控件	MFC类	描述
动画	CAnimateCtrl	显示连续的AVI视频剪辑
按钮	CButton	用来产生某种行为的按钮，以及复选框、单选钮和组框
组合框	CComboBox	编辑框和列表框的组合
编辑框	CEdit	用于键入文本
标题头	CHeaderCtrl	位于某一行文本之上的按钮，可用来控制显示文件的宽度
热键	CHotKeyCtrl	用于通过按下某一组合键来很快的执行某些常用的操作
图象列表	CImageList	一系列图象(典型情况下是一系列图标或位图)的集合。图象列表本身不是一种控件，它常常是和其它控件一起工作，为其它控件提供所用的图象列表
列表	CListCtrl	显示文本及其图标列表的窗口
列表框	CListBox	包括一系列字符串的列表
进度	CProgressCtrl	用于在一较长操作中提示用户所完成的进度
多格式文本编辑	CRichEditCtrl	提供可设置字符和段落格式的文本编辑的窗口

滚动条	CScrollBar	为对话框提供控件形式的滚动条
滑块	CSliderCtrl	包括一个有可选标记的滑块的窗口
旋转按钮	CSpinButtonCtrl	提供一对可用于增减某个值的箭头
静态文本	CStatic	常用于为其它控件提供标签
状态条	CStatusBarCtrl	用于显示状态信息的窗口，同MFC类CStatusBar类似

续表6.1

控件	MFC类	描述
选项卡	CTabCtrl	在选项卡对话框或属性页中提供具有类似笔记本中使用的分隔标签的外观的选项卡
工具条	CToolBarCtrl	具有一系列命令生成按钮的窗口，同MFC类CToolBar类似
工具提示	CToolTipCtrl	一个小的弹出式窗口，用于提供对工具条按钮或其它控件功能的简单描述
树	CTreeCtrl	用于显示一系列的项的继承结构

前面提到过，在MFC中，类CWnd是所有窗口类的基类，很自然的，它也是所有控件类的基类。Windows标准控件在以下环境下提供：

- Windows 95
- Windows NT 3.51及以后版本
- Win32s 1.3
- 注意：
- Visual C++ 4.2及以后版本不再支持Win32s。

## (2) ActiveX控件

ActiveX控件可用于对话框中，也可用于HTML文档中。这种控件过去被称为OLE控件。本书将在专门的章节中来讲述关于ActiveX控件的知识。这里仅指出ActiveX控件使用了与标准控件完全不同的接口和实现方法。

### (3) 其它MFC控件类

除了Windows标准控件和自己编写的或者来自于第三方软件开发商的ActiveX控件以外，MFC还提供了另外三种控件，它们由下面的三个类进行封装：

- 类CBitmapButton用于创建以位图作为标签的按钮，位图按钮最多可以包括四个位图图片，分别代表按钮的四种不同状态。
- 类CCheckBox用于创建选择列表框，这种列表框中的每一项前面有一个复选框，以决定该项是否被选中。
- 类CDragListBox用于创建一种特殊的列表框，这种列表框允许用户移动列表项。

在本章我们仅讲述第一类控件，即Windows标准控件。所涉及的内容包括各个控件的使用及相应的技巧。

## 第一节 使用对话框编辑器和ClassWizard

对于大多数Windows标准控件，我们一般都使用对话框编辑器来将它们添加到对话框中。

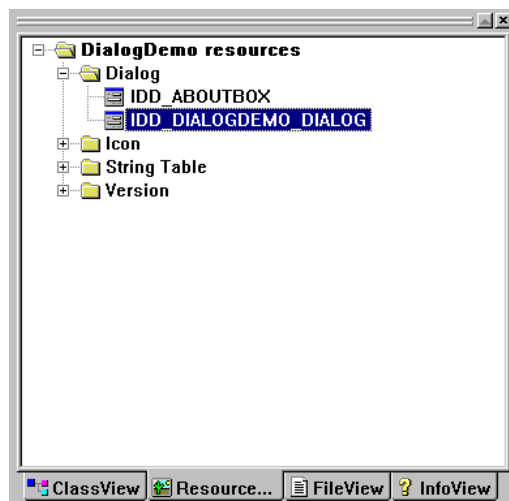


图6. 1 在ResourceView中选择对话框  
IDD\_DIALOGDEMO\_DIALOG



图6. 2 控件的Properties对话框



图6. 3 对话框编辑器的Controls工具窗口

在下面的过程中，我们将一个编辑框控件添加到在第四章创建的基于对话框的MFC框架应用程序的主对话框窗口中。

1. 首先，在Workspace窗口的ResourceView选项内双击DialogDemo resources\Dialog节点下的IDD\_DIALOGDEMO\_DIALOG图标。上面的操作如图所示。
2. 用鼠标选中标有“要做.....”的静态文本控件。右击鼠标，从上下文菜单中选择Properties，打开如图6.2所示的对话框，在Caption文本框中输入新的控件文本：“在下面的文本框中输入一些字符”，然后将静态文本控件拖动到对话框的左上角。
3. 从Controls工具窗口(如图6.3所示，如果在你的资源编辑器中看不到该工具窗口，可以在工具条上右击鼠标，从上下文菜单中选择Controls)中选择编辑控件图标 **ab**，在对话框中绘制一个编辑框控件，如图6.4所示。

在该编辑框控件的Properties窗口的General选项卡中输入其ID为IDC\_EDIT。然后在Styles选项卡下将Multiline复选框划上勾，并消除Auto HScroll复选框前的勾。

4. 右击该编辑框控件，从上下文菜单中选择ClassWizard命令，打开ClassWizard对话框，该对话框看起来如图6.5所示。

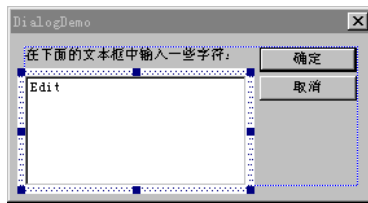


图6. 4 向对话框中添加一个编辑框控件

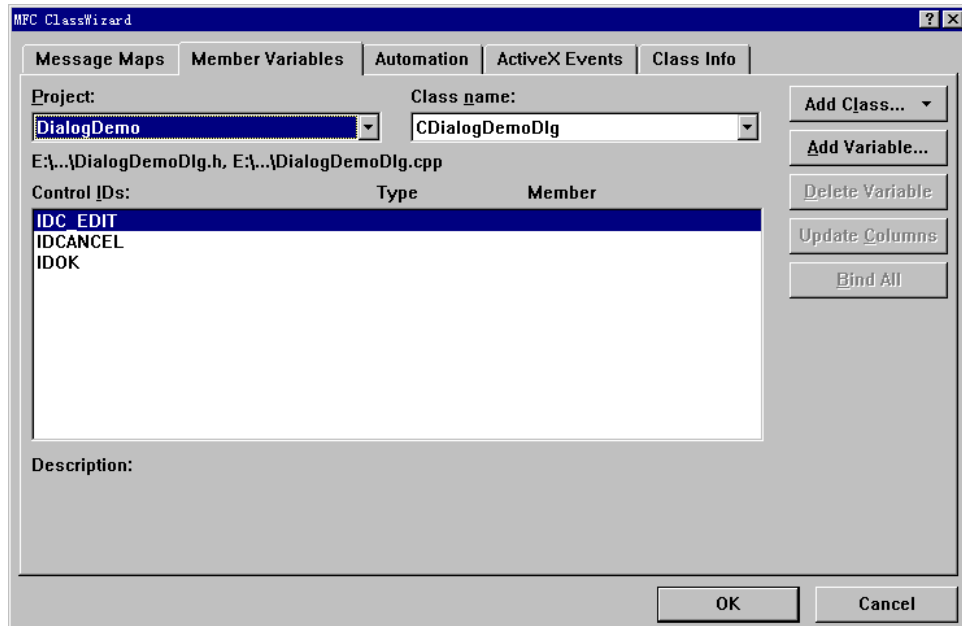


图6. 5 ClassWizard对话框

单击Member Variables选项卡，确信在Project处选择了DialogDemo，在Class name处选择了CDialogDemoDlg。现在我们为刚才添加的编辑框控件IDC\_EDIT添加一个数据映射入口。在Control IDs处选择IDC\_EDIT，单击右边的Add Variable按钮。打开如图6.6所示的对话框。

在Member variable name处链接变量名m\_strEdit（这里m表示该变量为类CDialogDemoDlg的一个成员变量，str表明其类型为字符串，即类CString），在Category下拉列表中选择Value（另一种选择是Control，两种选择的不同将在后面的内容中讲述），在Variable type下拉列表中选择CString（还有其它很多数据类型可供选择，但由于这里编辑框中的内容为一字符串，因此CString是最恰当的选择）。单击OK关闭对话框。

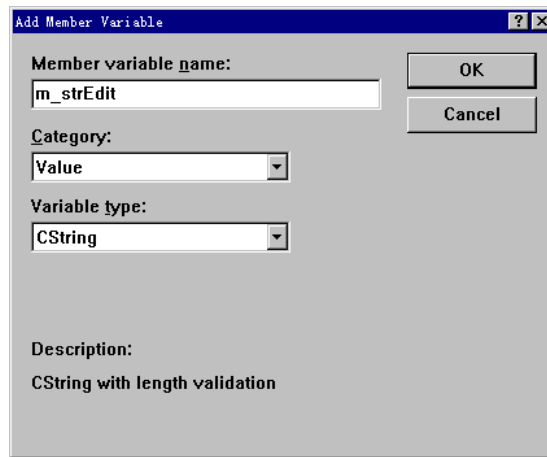


图6. 6 为控件映射添加成员变量

5. 检查一下现在的ClassWizard对话框(图6.7)与图6.5相比有何不同。在图6.7所示的对话框中下方的Maximum characters文本框中输入50。由字面意思可以很容易猜出其含义，即将编辑框IDC\_EDIT中可能的最长字符串的大小限制为50。单击OK关闭对话框。

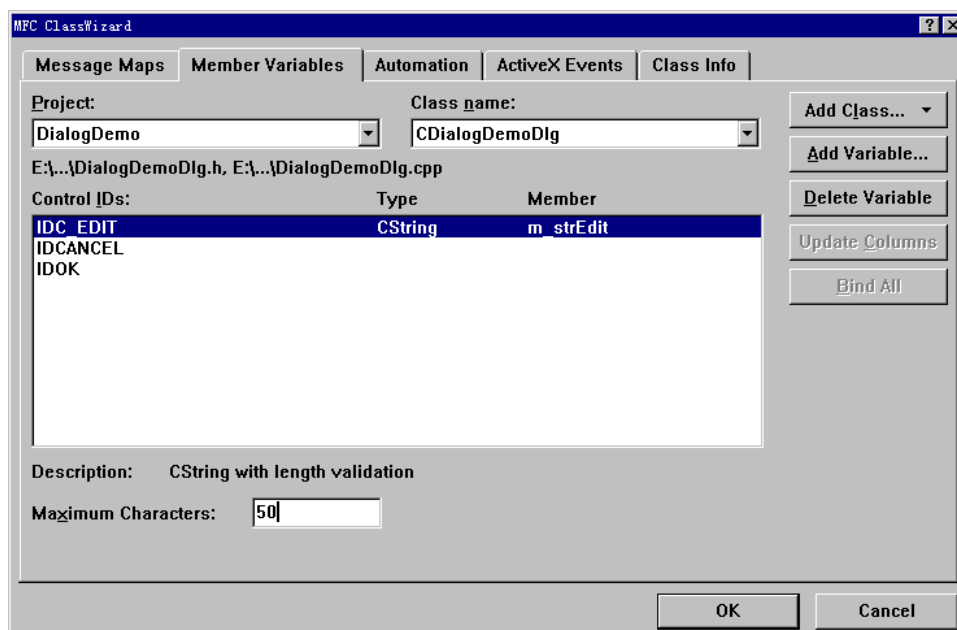


图6. 7 使用ClassWizard设置数据验证方案

6. 从Workspace窗口的ClassView中双击类CDialogDemoDlg的OnInitDialog成员函数，使用下面的代码来代替位于语句

```
return TRUE;
```

前的// TODO注释：

```
m_strEdit="您好！请在这里输入一些字符串。";
```

```
UpdateData(FALSE);
```



7. 在ClassView中双击类CDialogDemoApp的InitInstance成员函数，使用下面的代码来找替位于选择支

```
if (nResponse == IDOK)
```

下的//TODO注释：

```
AfxMessageBox(dlg.m_strEdit);
```

然后将同一成员函数中的下面的代码行删掉(或注释掉)：

```
m_pMainWnd = &dlg;
```

8. 编译并运行该应用程序。显示如图6.8所示的对话框。

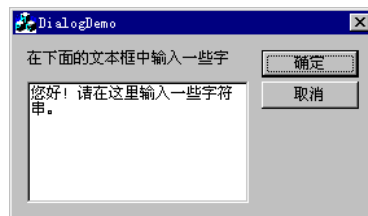


图6.8 示例程序DialogDemo的运行结果

在图6.8所示的文本框中输入一些字符，单击“确定”。随即弹出如图6.9所示的消息框。该消息框复述了用户在图6.8所示的对话框中的输入。我们还发现，在图6.8所示的对话框中，当输入字符串达到一定的长度之后，我们不可以再输入更多的字符，这是我们在前面设置了Maximum characters为50的结果。

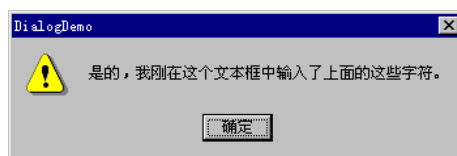


图6.9 以消息框的形式反馈输入的字符串

下面我们来看在上面的步骤中都完成了什么。首先我们使用资源编辑器向对话框模板中添加这些标准控件，这一步的概念很清晰，因此并不难理解。

然后，我们打开了所绘制的编辑框的Properties (属性)对话框。先将其控件ID设置为IDC\_EDIT。这时如果打开头文件Resource.h，就会发现宏IDC\_EDIT被定义为常量1001。不过，事实上在很多情况下我们并不需要关心每一控件的ID的具体值，而只需要记住相应的助记符。对于这里的编辑框控件，我们只需要记住IDC\_EDIT即可，而不需要关

心它等于1001。接着，我们在Styles选项卡中设置了Multiline属性，同时清除了Auto HScroll属性，两者共同作用使得编辑框IDC\_EDIT支持多行文本，并且如文本行的长度超过编辑框宽度时自动回行。

下面的步骤是最重要的一步，我们动用了功能强大的工具ClassWizard。首先，我们将编辑框与一个CString对象相关联，这使用了一种被称为Dialog Data Exchange (DDX)的机制。在这种机制中，我们先在处理函数OnInitDialog或对话框类的构造函数中对对话框对象的成员变量进行初始化，在对话框显示之前，框架的DDX机制将成员变量的值传递给对话框中的控件。这个过程在成员函数DoModal或Create被调用的过程中发生。类CDialog中对OnInitDialog成员函数的默认实现调用了类CWnd成员函数UpdateData来初始化对话框中的控件。这时我们就可以看到前面的第6步还可在具有下面的几种变通方案：

## 1. 将代码行

```
m_strEdit="您好！请在这里输入一些字符串。";
```

移到对基类的OnInitDialog成员函数的调用之前，即位于下面的代码之前：

```
CDialog::OnInitDialog();
```

## 2. 将代码

```
m_strEdit="您好！请在这里输入一些字符串。";
```

移到类CDialogDemoDlg的构造函数中。

对于上面的两种方法，与前面第6步中使用的方法相比，我们没有必要调用类CWnd的成员函数UpdateData。因为该函数在类CDialog的成员函数OnInitDialog中将被调用。

这三种方法之间并没有明确的优劣之分，在很多情况下，它们分别适用于不同的场合。

这里我们说一下成员函数UpdateData。该函数带有一个布尔类型的参数，如果该参数为FALSE，函数UpdateData将成员变量的值传递给对话框的变量；而如果该参数为TRUE，函数UpdateData将进行相反的过程。

如果用户单击了对话框中ID为IDOK的按钮，或者以TRUE为参数调用函数UpdateData，DDX机制从控件中将值传递到成员变量，同时对话框数据验证(dialog data validation, DDV)机制根据设定的验证规则验证所有数据项。

在数据交换的过程中，成员函数UpdateData先创建一个CDataExchange对象，然后调用对话框对类CDialog成员函数DoDataExchange的重载版本。该CDataExchange对象将作为成员函数DoDataExchange的一个参数，该参数定义了数据交换的上下文。

在DoDataExchange中，我们为每一个数据成员指定了一个对DDX函数的调用。每一个函数定义了基于由成员函数UpdateData所提供的CDataExchange参数所确定的上下文而进行的双向数据交换。

下面的代码摘自实现文件DialogDemo.cpp中对函数DoDataExchange的定义：

```
void CDialogDemoDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDialogDemoDlg)
    DDX_Text(pDX, IDC_EDIT, m_strEdit);
    DDV_MaxChars(pDX, m_strEdit, 50);
    //}}AFX_DATA_MAP
}
```

在两行注释//{{AFX\_DATA\_MAP和//}}AFX\_DATA\_MAP之间的代码部分称作数据映射。函数DDX\_Text使用CString对象m\_strEdit与ID为IDC\_EDIT的编辑框控件相关联。函数DDV\_MaxChars设置与编辑框控件IDC\_EDIT相关联CString对象m\_strEdit的最大长度为50。

需要注意的是，如果用户在模式对话框中单击了“取消”(Cancel)按钮，DoModal函数将返回值IDCANCEL，在这种情况下，在对话框和对话框对象之前的数据交换不会发生。

由于这个原因，如果DoModal函数返回了值IDOK，我们可以使用下面的代码来复述用户在对话框中所输入的值：

```
AfxMessageBox(dlg.m_strEdit);
```

- 注意：
- 在前面的第7步中有一个乍看起来有一些费解的过程，这就是我们为什么要将下面的代码从函数OnInitDialog中删除：
- `m_pMainWnd = &dlg;`

这基于下面的一个事实：

类CWinThread的数据成员m\_pMainWnd有一个有用的特征，如果由该成员所引用的窗口被关闭的话，MFC库将自动的终止CWinThread对象所代表的线程。这样，如果我们将指向dlg的指针赋予了成员变量m\_pMainWnd，那么，无论我们单击了“确认”还是“取消”，应用程序的主线程都将被自动终止，之后的代码当然不会得到执行。而在本示例中，我们希望在对话框被关闭后程序继续运行(即弹出一个消息重述用户所输入的内容)，因此不应该将dlg对象的指针赋予成员变量m\_pMainWnd，从而需要将前面的代码从函数OnInitDialog中删除。

## 第二节 所有窗口类的基类：CWnd

在MFC中类CWnd是一个很重要的类，它封装了Windows窗口句柄HWND。在Windows编程中，窗口句柄唯一的标识了一个窗口。然而，尽管类CWnd的对象和窗口句柄之间有着如此紧密的联系，但两者并不是等同的概念。CWnd对象通过类CWnd的构造函数和析构函数创建和销毁，而Windows窗口是Windows内部的一种数据结构，在类CWnd中，它通过Create成员函数创建，通过其析构函数销毁。除此之外，成员函数DestroyWindow可以销毁Windows窗口，而不需要销毁CWnd对象。

传统的Windows应用程序中，消息是通过一个称作窗口过程(window procedure，通常具有WndProc之类的函数名)的回调函数来处理的。这种方式在MFC中仍然使用，但为CWnd类及其消息映射所隐藏。在类CWnd中，Windows通知消息会被自动的通过消息映射传递到类CWnd中合适的OnMessage成员函数(这里OnMessage是指这些函数具有的以On为前缀的函数名，如OnPaint和前面接触到的OnInitDialog等)进行处理。通常我们都在类CWnd的派生类中重载需要处理的特定消息所对应的OnMessage成员函数。除了直接从CWnd派生新的窗口类以外，我们

更倾向于从MFC中定义的其它类，如CFrameWnd、CMIDFrameWnd、CMDIChileWnd、CView和CDialog以及CButton之类的控件类派生新的窗口类。在MFC中定义的这些类本身也是从CWnd派生的。

通常我们使用两个步骤来创建一个窗口：首先，调用类CWnd的构造函数来构造一个CWnd对象，然后调用其成员函数Create来创建窗口并将该窗口与所创建的CWnd对象相关联。

当用户终止该窗口时，销毁与之相关联的CWnd对象，或者调用CWnd对象的成员函数DestroyWindow删除窗口并销毁其数据结构。

大多数以HWND为参数的Win32 API函数都已作为类CWnd的成员函数进行了封装，事实上，很多时候我们通过类CWnd的派生类调用的成员函数并不是由派生类本身所提供的，而是在类CWnd中进行定义的。下面我们分类给出在CWnd类中定义的各类成员函数。完整而详尽的说明每一个成员函数在本书中是不现实的，这里我们仅给出对每一个成员函数的简短说明，以便读者在编程时能够很快的查找到所需的函数，这时再去查找有关于该函数的详细的说明就不是一件困难的事了。

1. 类CWnd的数据成员(表6.2)：

表6. 2 类CWnd的数据成员

数据成员	描述
m_hWnd	与该CWnd对象相关联的Windows窗口句柄(HWND)

2. 构造函数/析构函数(表6.3)

表6. 3 类CWnd的构造函数和析构函数

成员函数	获得图标句柄
SetIcon	设置句柄为一指定图标
GetWindowContextHelpId	获得帮助上下文标识符
SetWindowContextHelpId	设置帮助上下文标识符
ModifyStyle	修改当前窗口样式

ModifyStyleEx	修改当前窗口的扩展样式
---------------	-------------

## 5. 窗口大小和位置函数(表6.6)

表6.6 类CWnd的窗口大小和位置成员函数

成员函数	描述
GetWindowPlacement	获得显示状态和窗口的正常、最小化和最大化位置
SetWindowPlacement	设置显示状态和窗口的正常、最小化和最大化位置
GetWindowRgn	获得窗口的窗口区域的拷贝
SetWindowRgn	设置窗口区域
IsIconic	判断窗口是否被最小化(图标化)
IsZoomed	判断窗口是否被最大化
MoveWindow	改变窗口的位置和度量
SetWindowPos	改变子窗口、弹出式窗口或顶层窗口的大小、位置和顺序
ArrangeIconicWindows	排列所有最小化的子窗口
BringWindowToTop	将CWnd对象放到覆盖窗口栈的顶部
GetWindowRect	获得CWnd对象的屏幕坐标
GetClientRect	获得CWnd对象客户区的度量

## 6. 窗口访问函数

表6.7 类CWnd的窗口访问成员函数

成员函数	描述
ChildWindowFromPoint	判断包含指定点的子窗口
FindWindow	返回由其窗口名称和窗口类标识的窗口的句柄
GetNextWindow	返回窗口管理器列表中的下一个(或上一个)窗口

GetOwner	返回指向CWnd对象的所有者的指针
----------	-------------------

续表6.7

成员函数	描述
SetOwner	改变CWnd对象的所有者
GetTopWindow	返回属于CWnd对象的第一个子窗口
GetWindow	返回与当前窗口有指定关系的窗口
GetLastActivePopup	判断由CWnd对象所有的弹出窗口中最近激活的窗口
IsChild	判断CWnd对象是否为一个子窗口
GetParent	如果存在的话，获得CWnd对象的父窗口
GetSafeOwner	获得给定窗口的安全的所有者
SetParent	改变父窗口
WindowFromPoint	标识包括给定点的窗口
GetDlgItem	从指定的对话框获得标准符为指定ID的控件
GetDlgCtrlID	如果CWnd为一子窗口，返回其ID值
SetDlgCtrlID	当CWnd对象为一子窗口(不仅指对话框中的控件)时，为其指定控件ID或窗口ID
GetDescendantWindow	检查所有下级窗口(descendant window)并返回具有指定ID的窗口
GetParentFrame	获得CWnd对象的父框架窗口
SendMessageToDescendants	发送一条消息到窗口的所有下级窗口
GetTopLevelParent	获得窗口的顶层父窗口
GetTopLevelOwner	获得窗口的顶层所有者窗口
GetParentOwner	返回指向子窗口的父窗口的指针
GetTopLevelFrame	获得窗口的顶层框架窗口

UpdateDialogControls	用来更新对话框按钮或其它控件的状态
UpdateData	初始化对话框或从对话框中获取数据
CenterWindow	相对于父窗口使窗口居中

## 7. 更新和绘制函数(表6.8)

表6.8 类CWnd的更新和绘制函数

成员函数	描述
BeginPaint	为重绘操作准备CWnd对象
EndPaint	标记重绘操作的结束

续表6.8

成员函数	描述
Print	在指定的设备上下文中绘制当前窗口
PrintClient	在指定的设备上下文中(通常是打印机)绘制所有窗口
LockWindowUpdate	禁止或重新允许绘制指定的窗口
UnlockWindowUpdate	解除CWnd::LockWindowUpdate对窗口的锁定
GetDC	获得客户区的显示上下文
GetDCEx	获得客户区的显示上下文，并在绘制过程中允许裁剪
RedrawWindow	在客户区中更新指定的矩形或区域
GetWindowDC	获得整个窗口的显示上下文，包括标题条，菜单和滚动条
ReleaseDC	释放客户区或窗口设备上下文，并使其可为其它程序所使用
UpdateWindow	更新客户区
SetRedraw	决定在CWnd对象中的改变是否被重绘



GetUpdateRect	获得完全覆盖CWnd对象的更新区域的最小矩形坐标
GetUpdateRgn	获得CWnd对象的更新区域
Inval idate	使用整个客户区无效
Inval idateRect	通过将给定矩形添加到当前更新区域来使包括在给定矩形内的客户区无效
Inval idateRgn	通过将给定区域添加到当前更新区域来使包括在给定区域内的客户区无效
Val idateRect	通过将给定矩形从当前更新区域中移出来使包括在给定矩形内的客户区有效
Val idateRgn	通过将给定区域从当前更新区域中移出来使包括在给定区域内的客户区有效
ShowWindow	显示或隐藏窗口
IsWindowVisible	判断窗口是否可见
ShowOwnedPopups	显示或隐藏窗口拥有的所有弹出式窗口
EnableScrol lBar	允许或禁止滚动条上的一个或两个箭头

### 8. 坐标映射函数(表6.9)

表6. 9 类CWnd的坐标映射函数

成员函数	描述
MapWindowPoints	从CWnd对象的坐标空间映射一系列点到另一窗口的坐标空间

续表6.9

成员函数	描述
ClientToScreen	转换给定点的客户坐标或显示矩形到屏幕坐标
ScreenToClient	转换给定点的屏幕坐标或显示矩形到客户坐标

### 9. 窗口文本函数(表6.10)

表6. 10 类CWnd的窗口文本函数

--	--

成员函数	描述
SetWindowText	设置窗口文本或标题条(如果有的话)为指定文本
GetWindowText	获得窗口文本或标题条
GetWindowTextLength	返回窗口文本或标题条的长度
SetFont	设置当前字体
GetFont	获得当前字体

## 10. 滚动函数(表6.11)

表6.11 类CWnd的滚动成员函数

成员函数	描述
GetScrollPos	获得滚动框的当前位置
GetScrollRange	拷贝给定滚动框中滚动块的当前最大和最小位置
ScrollWindow	滚动客户区的内容
ScrollWindowEx	滚动客户区内容。与ScrollWindowEx类似，但具有一些附加特性
GetScrollInfo	获得关于某一滚动条的由SCROLLINFO结构维护的信息
GetScrollLimit	获得滚动条的限制
SetScrollInfo	设置关于滚动条的信息
SetScrollPos	设置滚动条的当前位置，并在指定的情况下重绘滚动条以反映新的位置
SetScrollRange	设置给定滚动条的最小和最大位置值
ShowScrollBar	显示或隐藏滚动条
EnableScrollBarCtrl	允许或禁止兄弟滚动条控件
GetScrollBarCtrl	返回兄弟滚动条控件

RepositionBars	在客户区中对控件条重定位
----------------	--------------

### 11. 拖放函数(表6.12)

表6. 12 类CWnd的拖放成员函数

成员函数	描述
DragAcceptFiles	使窗口可以接受文件拖放

### 12. 插入符函数(表6.13)

表6. 13 类CWnd的插入符成员函数

成员函数	描述
CreateCaret	新的插入符形状，并获得该插入符的所有权
CreateSolidCaret	创建方块形状的插入符，并获得该插入符的所有权
CreateGrayCaret	创建变灰方块形状的插入符，并获得该插入符的所有权
GetCaretPos	获得插入符当前位置的客户坐标
SetCaretPos	移动插入符到指定的位置
HideCaret	隐藏插入符
ShowCaret	在插入符的当前位置显示插入符

### 13. 对话框项函数(表6.14)

表6. 14 类CWnd的对话框项函数

成员函数	描述
CheckDlgButton	在按钮控件前放置选中标记或清除按钮控件的选中标记
CheckRadioButton	选中指定的单选钮并清除指定给中其它所有单选钮的选中标记

GetCheckedRadioButton	返回一组按钮中当前选中单选钮的ID
DlgDirList	使用文件或目录列表填充一列表框
DlgDirListComboBox	使用文件或目录列表填充一组合框的列表框
DlgDirSelect	从一列表框中获得当前选择
DlgDirSelectComboBox	从一组合框的列表框中获得当前选择
GetDlgItemInt	将给定对话框中某一控件的文本转换为一个整数值
GetDlgItemText	获得与某一控件相关联的标题或文本
GetNextDlgGroupItem	查找同一组中的下一个(或前一个)控件

续表6.14

成员函数	描述
GetNextDlgTabItem	查找在指定控件之前(或之后)的第一个具有WS_TABSTOP样式的控件
IsDlgButtonChecked	判断一个按钮控件是否选中
IsDialogMessage	判断一个给定消息是否影响非模态对话框，如果是，处理该消息
SendDlgItemMessage	向指定的控件发送一条消息
SetDlgItemInt	使某一控件的文本为某一给定整数值
SetDlgItemText	设置指定对话框中某一控件的标题或文本
SubclassDlgItem	将一个Windows控件与CWnd对象相关联，并使其通过CWnd对象的消息映射传递消息
ExecuteDlgInit	初始化对话框资源
RunModalLoop	为一模态窗口获取、翻译或发送消息

ContinueModal	使一窗口继续保持模态
EndModalLoop	结束某一窗口的模态状态

## 14. 数据绑定函数(表6.15)

表6. 15 类CWnd的数据绑定成员函数

成员函数	描述
BindDefaultProperty	将调用对象的默认简单绑定属性(该属性在类型库中标记)绑定至相关联的数据源控件的游标
BindProperty	将数据绑定控件的游标绑定属性绑定至数据源控件，并使用MFC绑定管理器注册绑定关系
GetDSCCursor	获得指向由数据源控件的数据源、用户名、密码和SQL属性定义的底层游标的指针

## 15. 菜单函数(表6.16)

表6. 16 类CWnd的菜单成员函数

成员函数	描述
GetMenu	获得指向指定菜单的指针
SetMenu	设置菜单为指定的菜单
DrawMenuBar	重绘菜单条
GetSystemMenu	允许应用程序访问控制菜单以进行复制和修改

续表6.16

成员函数	描述
HiLiteMenuItem	加亮顶层菜单项或移去顶层菜单项的加亮显示

## 16. 工具提示函数(表6.17)

表6. 17 类CWnd的工具提示函数

成员函数	描述
EnableToolTip	允许工具提示控件
CancelToolTip	禁止工具提示控件
FilterToolTipMessage	获得对话框中与某一控件相关联的标题或文本
OnToolHitTest	判断一个点是否在指定工具的绑定矩形内，并获得该工具的信息

## 17. 计时器函数(表6.18)

表6. 18 类CWnd的计时器成员函数

成员函数	描述
SetTimer	安装系统计时器，计时器触发时发送WM_TIMER消息
KillTimer	消除系统计时器

## 18. 提示函数(表6.19)

表6. 19 类CWnd的提示成员函数

成员函数	描述
FlashWindow	闪烁窗口一次
MessageBox	创建并显示一个包括应用程序提供的消息和标题的窗口

## 19. 窗口消息函数(表6.20)

表6. 20 类CWnd的窗口消息成员函数

成员函数	描述
GetCurrentMessage	返回窗口正在处理的消息的指针。仅当在一个OnMessage消息处理函数中调用该成员函数。
Default	调用默认窗口过程，该过程提供对所有应用程序未处理的消息的默认处理

PreTranslateMessage	由CWinApp使用，在窗口消息被发送到TranslateMessage和DispatchMessage之前对其进行过滤
---------------------	--

续表6.20

成员函数	描述
SendMessage	将一条消息发送到CWnd对象，直至该对象处理该消息之后才返回
PostMessage	将一条消息放入程序的消息队列，不等待窗口处理该消息就立即返回
SendNotifyMessage	将指定消息发送到窗口，并尽可能快的返回，这依赖于调用线程如何创建窗口

20. 剪贴板函数(表6.21)

表6. 21 类CWnd的剪贴板函数

成员函数	描述
ChangeClipboardChain	从剪贴板查看器链中移去CWnd对象
SetClipboardViewer	添到CWnd对象到窗口链，这些窗口当剪贴板内容改变时会收到通知
OpenClipboard	打开剪贴板。其它程序仅当Windows CloseClipboard函数被调用时才可以更改剪贴板
GetClipboardOwner	获得剪贴板的当前拥有者的指针
GetOpenClipboardWindow	获得指向当前打开剪贴板的窗口的指针
GetClipboardViewer	获得指向剪贴板查看器链中第一个窗口的指针

21. OLE控件函数(表6.22)

表6. 22 类CWnd的OLE控件函数

成员函数	描述

SetProperty	设置OLE控件属性
OnAmbientProperty	实现环境属性值
GetControlUnknown	获得指向一未知OLE控件的指针
GetProperty	获得一OLE控件的属性
InvokeHelper	调用OLE控件方法或属性

## 22. 可重载函数(表6.23)

表6. 23 类CWnd的可重载成员函数

成员函数	描述
WindowProc	为CWnd对象提供一个窗口过程。默认的窗口过程通过消息映射发送消息
DefWindowProc	调用默认窗口过程，该过程提供应用程序未处理的所有窗口消息的默认处理
PostNcDestroy	在窗口被销毁后由OnNcDestroy函数调用
OnNotify	由框架调用以通知父窗口某一事件在某一控件中发生或者该控件需要信息
OnChildNotify	由父窗口调用以给通知控件一个响应控件通知的机会
DoDataExchange	用于对话框数据交换和验证。由UpdateData调用

其余函数包括对各种窗口消息的消息处理函数，这些函数为数众多，这里我们限于篇幅不再一一介绍。类CWnd中定义的消息处理函数几乎都具有一致的命名方式，其格式为前缀On再加上相应的消息名，如WM\_PAINT消息的处理函数在类CWnd中被命名为OnPaint。因此，只需要知道所需处理的消息，就可以很快的推知该消息的处理函数名。

### 第三节 按钮

在本节中要讲述的实际包括四种控件：下压按钮、单选钮、复选框和



组框，它们之间无论在外观还是在使用上都有较大的差异。在MFC中之所以使用一个类CButton来封装这四种不同控件纯粹出于历史的原因。这使得一些使用过Visual Basic之类的编程工具的程序员可能会有一点混淆，但相信只需要很短的时间就可以习惯这一点转变。

下面我们分别讲述这四种按钮控件：

### 6.3.1 下压按钮

在基于对话框的应用程序中，下压按钮是最常见的控件之一，如图6.10所示。

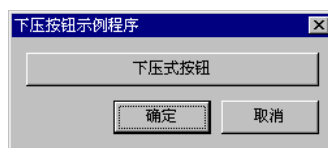



图6. 10 下压按钮

下面的步骤讲述如何向对话框中添加下压按钮控件。

1. 在ResourceView中双击需要添加下压按钮控件的对话框模板，Developer Studio将在资源编辑器中打开该对话框模板。如图6.11所示。
2. 在图6.3所示的控件工具窗口中选择图标，直接使用鼠标在对话框中绘制出一个下压按钮。
3. 右击所绘制的下压按钮，选择Properties命令打开其属性对话框，设置下压按钮的各项属性。下面详细描述这些属性的含义：

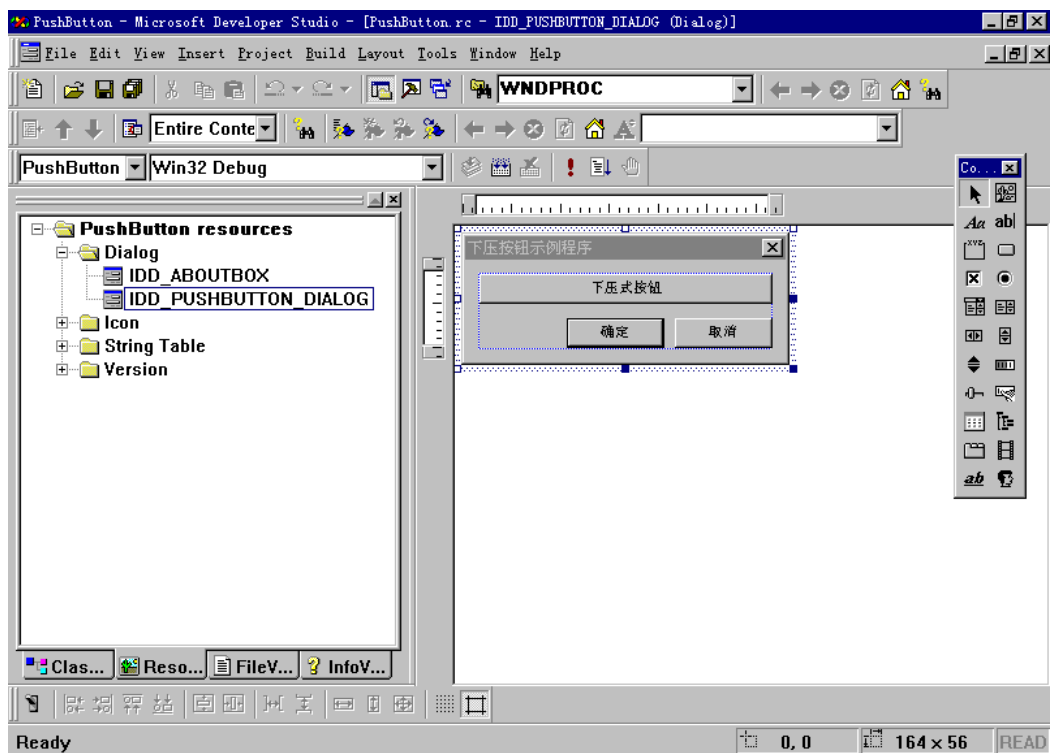


图6. 11 在资源编辑器中打开一对话框模板



图6. 12 在对话框中绘制下压按钮控件

### 一般属性：

- ID：** 在头文件中定义的符号。类型：符号、整数或用引号括起来的字符串
- Caption：** 控件标签文本。如果在标题中的某个字母前加上了“&”符号，该字母在显示时将被加上下划线，相应的“&”符不会被显示。在运行直接按下加有下划线的字母同单击按钮具有同样的效果。默认情况下，资源编辑器对按钮标题的命名依赖于控件的类型，如Button1、Button2等。
- Visible：** 决定当应用程序第一次运行时控件是否可见。类型：布尔值 默认值为真
- Disabled：** 决定当对话框创建时该控件是否显示为禁止状态。类型：布尔值 默认值为假

- Group :** 指定一组控件中的第一个控件。在同组控件中用户可以使用箭头键在控件之间移动。以tab\_order为序，在该控件之后的所有该属性值为False的控件将被视为同一组控件，直到遇上Group属性标记为True的控件为止。类型：布尔值 默认值为假
- Tabstop :** 决定用户是否可以使用TAB键来定位到该控件。类型：布尔值 默认值为假
- HelpID :** 为控件指定一个帮助标识符。该标识符基于相应的资源标识符。类型：布尔值 默认值为假

## 样式 :

- Default button :** 该属性为真时，控件将作为对话框中的默认按钮，默认按钮在对话框第一次显示时具有粗的黑边，用户在对话框中按下ENTER键相当于单击该按钮。一个对话框中只允许有一个默认按钮。类型：布尔值 默认值为假
- Owner draw :** 创建一个自绘按钮。使用自绘按钮可以定制按钮的外观。使用自绘按钮需要重载下面的两个函数或其中之一：CWnd::OnDrawItem和CButton::OnDraw。
- Icon :** 在按钮显示时使用一个图标来代替文本。类型：布尔值 默认值为假
- 该按钮样式为Windows 95中新引入的按钮样式
- Bitmap :** 在按钮显示时使用位图来代替文本。类型：布尔值 默认值为假
- 该样式为Windows 95中新引入的样式
- Multi-line :** 当按钮文本太长时使用多行回绕的方式进行显示。类型：布尔值 默认值为假

Notify :	按钮控件被单击或双击时通知父窗口。 类型：布尔值 默认值为真
Flat :	使用平面外观代替按钮默认的三维外观。类型：布尔值 默认值为假
Horizontal alignment :	设置按钮标题文本的对齐方式(左对齐、右对齐、居中对齐或使用默认位置)
Vertical alignment :	设置按钮标题文本的对齐方式(向上对齐、向下对齐、居中对齐或使用默认位置)

## 扩展样式

Client edge :	使按钮看起来有下凹的感觉。类型：布尔值 默认值为假
Static edge :	在按钮边缘创建边框。类型：布尔值 默认值为假
Modal frame :	提供一个三维框架
Transparent :	使控件透明。位于透明窗口下面的窗口不会被该窗口所覆盖。具有透明样式的窗口仅当所有底层兄弟窗口完成更新之后才会收到WM_PAINT消息。类型：布尔值 默认值为假
Accept files :	是否接受文件拖放。如果在控件上放下文件时，控件将接收到WM_DROPFILES消息。类型：布尔值 默认值为假
No parent notify :	指定子窗口不向父窗口发送WM_PARENTNOTIFY消息。类型：布尔值 默认值为假
Right aligned text :	指定文本为右对齐。类型：布尔值 默认值为假
Right-to-left reading	使用从右向左的阅读方式来显示文本。主要用于希伯来语系和阿拉伯语

order :                    等。类型：布尔值 默认值为假

- 技巧：

- 如果需要在控件的标题文本中使用“&”符，可以使用双写的“&”符，如按钮文本“&File && Directory”在显示时将成为



- 如果需要在控件标题中使用多行文本，可以将按钮控件的 Multiline 属性设置为真，然后在需要换行的地方使用转义字符“\n”或“\r”。在 Multiline 属性值为真的情况下，如果文本行的宽度超过了控件的宽度，即使没有使用换行转义字符，文本也将会在合适的地方进行折行处理。但要注意，其它一些转义字符序列，如“\t”等不被控件所支持。

我们一般只处理按钮控件一种通知消息：BN\_CLICKED，该消息表示用户单击了该按钮控件。按钮控件的另外一种通知消息是 BN\_DOUBLECLICKED，它表示用户双击了按钮控件，但是一般情况下我们不需要处理下压按钮的双击事件。

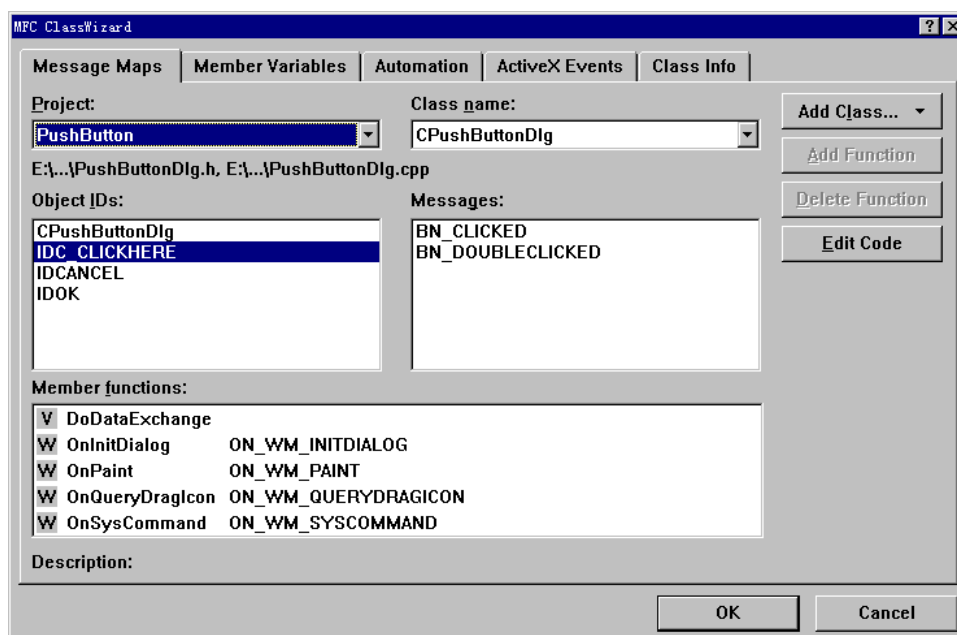


图6. 13 ClassWizard对话框：Message Maps选项卡

下面我们介绍如何为下压按钮的单击事件添加消息处理函数和消息映射，这里我们假设所添加的下压按钮ID为IDC\_CLICKHERE，标题文本为“单击这里(&C)”，其余属性使用默认设置。

第一种方法如下：

1. 在资源编辑器右击按钮IDC\_CLICKHERE，选择“ClassWizard”，打开如图6.13所示的窗口，单击Message Maps选项卡。

确信在Project处选择的工程为当前工程，Class name处为当前对话框模板所对应的类。Object IDs列表框中给出了当前对话框类中的所有对象标识符，从中选择IDC\_CLICKHERE，即我们刚才添加的下压按钮，这里，在右边的Message列表框中给出了当前对象的消息，这里即BN\_CLICKED和BN\_DOUBLECLICKED，从中选择BN\_CLICKED（它代表了按钮的单击事件），然后单击右边的Add Function按钮(注意：Add Function按钮仅当已选择了某一消息时才会出现)。

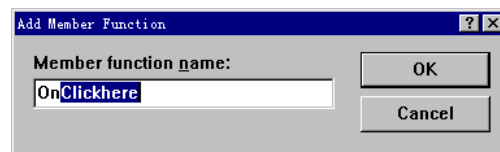


图6.14 决定是否需要更改命令处理函数名

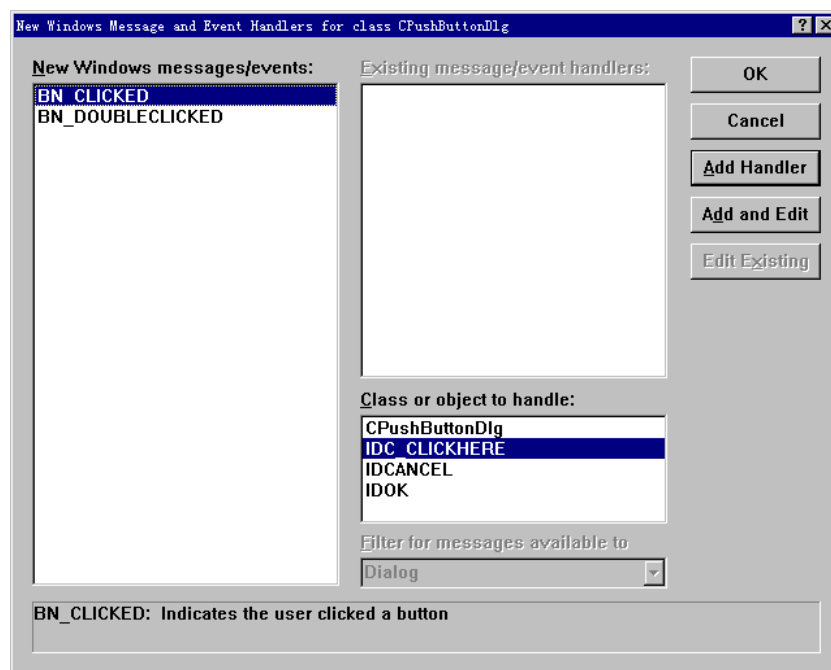


图6.15 为控件通知消息添加处理函数

2. 在随后出现的对话框(如图6.14所示)中选择是否需要更改命令处理函数的函数名。ClassWizard的默认函数名遵从于下面的命令协议：

前缀On + 控件ID中除去IDC\_前缀的剩余部分

这里我们接受默认的命令处理函数名OnClickhere。

3. 新添加的命令处理函数OnClickhere已经出现在图6.13所示的对话框中的下面的Member functions部分。同时，Edit Code按钮获得输入焦点。单击该按钮，ClassWizard将在Developer Studio的代码编辑器窗口中打开函数OnClickhere，并高亮度显示下面的// TODO注释：

```
// TODO: Add your control notification handler code here
```

我们使用下面的代码来替换上面的// TODO注释：

```
MessageBox
```

```
("您刚才单击了按钮 IDC_CLICKHERE, 因此相应的命令处理函数 OnClickhere 被调用!");
```

第二种方法：

1. 在资源编辑器中右击按钮IDC\_CLICKHERE，选择Events命令，打开如图6.15所示的对话框：

2. 在Class or object to handle列表框中选择IDC\_CLICKHERE，然后在New Windows messages/events列表框中选择BN\_CLICKED，单击右边的Add and Edit，余下的步骤同第一种方法的第2步开始相同。

这时编译并运行上面的程序，单击标签为“单击这里”的下压按钮，弹出如图所示的消息框。



图6. 16 程序PushButton的运行结果

下面我们来看相应的消息映射。

首先，在类CPushButtonDlg的定义中添加了消息处理函数OnClickhere的原型：

```
afx_msg void OnClickhere();
```

函数OnClickhere的声明被放进了两行注释分隔符//{{AFX\_MSG (CPushButtonDlg)和//}}AFX\_MSG之间。前面我们提到过，ClassWizard将由它定义的消息处理函数的声明放入这两行注释分隔符之间。

下面我们来看相应的消息映射入口。它位于实现文件

PushButtonDlg.cpp中的两个宏BEGIN\_MESSAGE\_MAP和END\_MESSAGE\_MAP之间：

```
ON_BN_CLICKED(IDC_CLICKHERE, OnClickhere)
```

其中第一个参数IDC\_CLICKHERE为控件的标识符，第二个参数OnClickhere为相应的消息处理函数。

一旦弄清楚了由ClassWizard添加这些代码，我们就可以手动的添加命令消息处理函数的消息映射。但是，从上面的过程中我们可以很明显的看出一点，使用ClassWizard来完成这一点要简单得多。

下面我们介绍与下压按钮控件有关的几个技巧：

### (1) 在运行过程中改变下压按钮的标题文本

有时候我们需要在程序运行的过程中改变按钮的标题文本。典型的，我们可能需要根据用户所输入的数据来决定按钮上应该写些什么。我们到前面去看一下表6.14，看一看有什么成员函数可以完成这种功能。

很好，类CWnd的成员函数SetDlgItemText可以由窗口或对话框所有的控件的标题文本。其原型如下：

```
void SetDlgItemText( int nID, LPCTSTR lpszString );
```

其中nID为控件标识符(ID)，lpszString为控件的新标题文本。

成员函数SetDlgItemText事实上是向控件发送一条WM\_SETTEXT消息，该消息的wParam参数必须为0，而lParam为指向窗口标题文本字符串的指针。

因此，SetDlgItemText等价于下面的函数调用：

```
CWnd::SendDlgItemMessage(nID, WM_SETTEXT, 0, LPARAM(lpszString));
```

或

```
::SendDlgItemMessage(GetSafeHwnd(), nID, WM_SETTEXT, 0, LPARAM(lpszString));
```

比如说，我们用以将下面的代码添加到OnClickhere中对MessageBox的调用之后：

```
SetDlgItemText(IDC_CLICKHERE, "此按钮已被单击过.");
```



## (2) 使用按钮无效(或有效)

假设我们在上面的例子中希望用户只能单击按钮IDC\_CLICKHERE一次。那么，按钮IDC\_CLICKHERE被单击一次之后应该变灰，以禁止用户再次单击它。这可以通过下面的步骤来实现：

首先调用对话框对象的成员函数GetDlgItem（该成员函数在类CWnd中定义），该成员函数获得一个指向对话框中的控件的CWnd指针，然后再通过该指针调用控件对象的成员函数EnableWindow（该成员函数在类CWnd中定义）。该成员函数允许或禁止调用它的CWnd对象对应窗口。整个过程可以使用一行语句来实现，如下所示：

```
GetDlgItem(IDC_CLICKHERE)->EnableWindow(FALSE);
```

其中GetDlgItem函数以控件的ID为参数，返回值的类型为CWnd\*，如果需要通过该指针调用在类CButton所定义的成员函数，可以使用强制类型转换。EnableWindow以一个布尔值为参数，该参数为真时表示允许该窗口接受鼠标和键盘输入，为假时禁止该窗口接受鼠标和键盘输入。这里再一次强调，控件本身也是一种窗口。

将上面的代码放到命令处理函数OnClickhere的最后，这样，在单击一次按钮“单击这里”之后，对话框如图6.17所示。

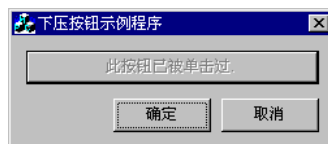


图6.17 处于禁止状态的按钮控件

此外，如果使用了ClassWizard为按钮建立了对话框的成员变量的数据映射，则可以通过对话框中的成员变量直接操纵控件。在本例中，如果我们已将下压按钮映射为类型为CButton的成员变量m\_bnClickhere，则可以通过下面成员函数调用设置按钮的允许状态：

```
m_bnClickhere->EnableWindow(FALSE);
```

## (3) 使按钮获得输入焦点

具有输入焦点的窗口将会得到所有的键盘输入消息。我们可以通过类CWnd的成员函数GetFocus来使对话框中的控件获得输入焦点。

试将下面的代码加到消息处理函数OnInitDialog的return语句前：

```
m_bnClickhere.SetFocus();
```

或

```
GetDlgItem(IDC_CLICKHERE)->SetFocus();
```

编译并运行程序。非常奇怪，输入焦点并没有被设置到下压按钮“单击这里”上。依然是按钮“确定”拥有当前输入焦点。

请注意这样的事实：

- 注意：
- 如果在消息处理成员函数OnInitDialog中将输入焦点设置到指定的控件，则函数应该返回FALSE，这是因为如果WM\_INITDIALOG消息的处理函数返回真值，Windows会将输入焦点设置为对话框中的第一个控件。因此，如果在该处理函数中设置了控件的输入焦点，WM\_INITDIALOG消息的处理函数应该返回假值。

将下面的代码

```
return TRUE;
```

修改为

```
return FALSE;
```

这时再编译并运行程序，则输入焦点将被正常地设置到下压按钮“单击这里”上。这时按下空格键相当于在按钮“单击这里”上单击鼠标左键。

#### (4) 使用图形代替文本

在一些应用程序，尤其是一些多媒体应用程序中，我们希望按钮的外观看起来更加的美观，比如说我们希望使用多变的图形代替单调乏味的纯文本。对于一般的按钮控件，我们可以使用两种方法来在按钮中使用图形来代替文本。

第一种方法是使用图标来代替文本。下面的示例说明了这种用法：

1. 使用资源编辑器或其它工作编辑一个图标资源，其ID为IDI\_CLICKHERE，图案如图6.18所示。
2. 在希望使用图标图案的按钮控件的Properties属性框在Styles选

项卡中设置Icon属性为真。并按图6.19修改对话框及其中控件的大小。

3. 在类CPushButtonDlg的消息处理成员函数OnInitDialog中添加下面的代码。这些代码应该在对基类的OnInitDialog成员函数的调用之后。

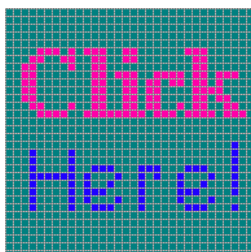


图6. 18 图标IDI\_CLICKHERE



图6. 19 为使用图标按钮修改对话框中控件的大小

```
HICON hIcon=AfxGetApp()->LoadIcon(IDI_CLICKHERE);  
m_bnClickhere.SetIcon(hIcon);
```

编译该应用程序，运行结果如图6.20所示。



图6. 20 在按钮中使用图标的示例

这时单击按钮Click Here，图标图案会有向右和向下下压的效果。

第二种方法是使用位图来代替文本。步骤如下：



图6. 21 位图资源IDB\_CLICKHERE

1. 向工程资源中添加如图6.21的位图资源，其ID为IDB\_CLICKHERE。
2. 在希望使用位图图案的按钮控件的Properties属性框在Styles选项卡中设置Bitmap属性为真。我们注意到Icon属性和Bitmap属性是互斥的，即选择一属性的同时也清除了另一属性。并按图6.19修改对话框及其中控件的大小。同时参考最终运行结果(如图6.22)修改对话框及其按钮的大小。
3. 在OnInitDialog成员函数中添加如下的代码：

```
HBITMAP hBitmap=LoadBitmap(AfxGetApp()->m_hInstance,MAKEINTRESOURCE  
(IDB_CLICKHERE));
```

```
m_bnClickhere.SetBitmap(hBitmap);
```

在上面的代码中，我们使用Win32 API函数LoadBitmap（注意它不是类CWinApp的成员函数）来加载位图资源IDB\_CLICKHERE，从而获得位图句柄hBitmap，最后以该句柄为参数调用类CButton的成员函数SetBitmap。

编译并运行上面的程序，得到如图6.22所示的运行结果。



图6.22 在按钮中使用位图的示例

- 注意：
- 在上面的示例程序中我们使用了真彩色(24位)的位图。这样的位图不可以使用资源编辑器来进行编辑。上面的位图是使用其它专门的图形工具来进行编辑并保存到文件Clickhere.bmp中的。请按下面的步骤将该文件添加为工程的资源：
  1. 选择Insert菜单下的Resource命令，打开如图所示的对话框。

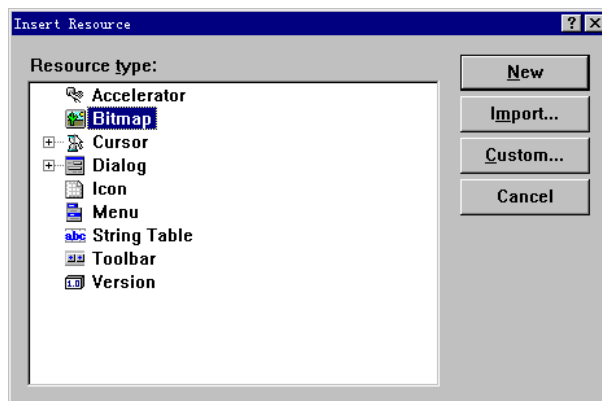


图6. 23 插入新的资源

2. 从中单击Import按钮，从位图文件Clickhere.bmp中输入资源。注意在文件类型中选择“ All files (\*.\*) ”。

Developer Studio将弹出如图6.24所示的警告对话框。该对话框表明位图资源已被正确添加。但由于使用了多于256色的颜色数，因此该资源不可以在资源编辑器中打开。

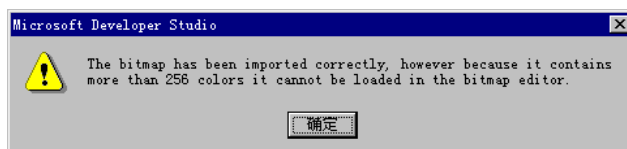


图6. 24 试图添加使用了多于256色的位图资源时的警告消息框

3. 按正常的方法将所添加的位图资源的ID修改为IDB\_CLICKHERE。必要时重新编辑资源文件或工程。

### 6.3.2 位图按钮

位图按钮是由MFC提供的几种附加控件之一。在前一节的过程中，我们可以使用一个位图来代替文本作为下压按钮的标签。而在位图按钮中，我们可以使用多达四个位图来分别代表按钮处于四种不同的状态(凸起、按下、获得焦点或被禁止)下的显示。而且，使用位图按钮还可以去除掉令人讨厌的按钮黑边。而使用位图按钮并不复杂，但是相比起标准的按钮控件(它由Windows自身所提供)而言有一些特殊。下面的过程描述了位图按钮的使用，它们在MFC中使用类CBitmapButton封装。

1. 使用AppWizard创建新的基于对话框的MFC工程CBitmapButton。

2. 使用资源编辑器绘制一个标准按钮，将其ID设为IDC\_CLICKHERE，标题文本设为CLICKHERE，然后在Styles选项卡中将Owner draw属性

设置为真。

### 3. 向工程中添加四个位图资源。

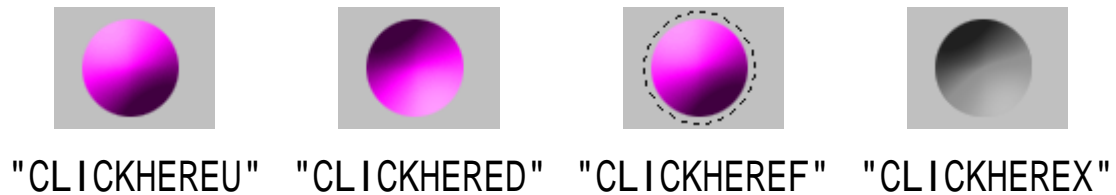


图6. 25 位图按钮IDC\_CLICKHERE所使用的四个位图资源

所添加的四个位图资源的ID的设置取决于在第一步中的标题文本的设置：按钮未按下去时使用的位图添加了后缀"U"；按钮按下去时使用的位图添加了后缀"D"；按钮拥有焦点时使用的位图添加了后缀"F"；按钮被禁止时使用的位图添加了后缀"X"。需要注意的是，由于这些位图资源的ID为字符串，因此在使用属性对话框设置其ID时一定要加了双引号，否则资源编辑器会将该ID值看作代表一个整型量的符号。

4. 在对话框类CBitmapButtonDlg(这里我们沿用上一节中的示例程序)中添加类型为CBitmapButton的新的成员变量m\_bnClickhere。

5. 在OnInitDialog成员函数中的return语句前添加下面的代码：

```
m_bnClickhere.AutoLoad(IDC_CLICKHERE, this);

CRect rect1,rect2;

CButton *pClickhere=(CButton*)GetDlgItem(IDC_CLICKHERE);

GetClientRect(&rect1);

pClickhere->GetWindowRect(&rect2);

ScreenToClient(&rect2);

pClickhere->MoveWindow(rect2.left,(rect1.Height()-rect2.Height())/2,
rect2.Width(),rect2.Height());
```

其中第一个参数IDC\_CLICKHERE是位图按钮的资源ID，第二个参数为指向该位图按钮的父窗口的CWnd对象的指针，这里即类CBitmapButtonDlg的this指针。类CBitmapButton的成员函数AutoLoad完成以下几步工作：

(1) 将按钮与CBitmapButton对象相关联；

(2) 自动加载按钮所使用的位图，条件是这些位图资源满足步骤2中的命名约定；

(3) 自动改变控件的大小以适合所加载的位图资源。

接下来的几行代码将位图按钮在对话框中进行垂直居中。首先类CWnd的成员函数GetClientRect返回了对话框的客户区矩形，接着，类CWnd的成员函数GetWindowRect返回了控件IDC\_CLICKHERE的窗口矩形，然后使用类CWnd的成员函数ScreenToClient将rect2由屏幕坐标转换为对话框的客户坐标，这是因为类CWnd的成员函数MoveWindow在移动子窗口时将使用父窗口的客户区坐标，而不是使用屏幕坐标。

6. 按图6.26添加下面的下压按钮IDC\_DISABLE，将其标题设置为“禁止使用(&X)”。

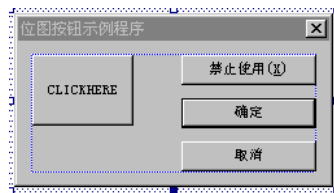


图6. 26 位图按钮示例程序对话框的设计

7. 将所有下压按钮的Tab stop属性(位于General选项卡中)设置为真。并按图6.26调整各控件的大小位置。其中按钮CLICKHERE的大小的无关紧要的，我们只需要保证对话框左边是否有足够的空间来显示按钮所使用的位图即可。

8. 为按钮IDC\_DISABLE的BN\_CLICKED命令编写下面的命令处理函数：

```
void CBitmapButtonDlg::OnDisable()
{
    CButton *pClickhere=(CButton*)GetDlgItem(IDC_CLICKHERE);
    static int blsEnabled=pClickhere->IsWindowEnabled();
    if (blsEnabled)
    {
        pClickhere->EnableWindow(FALSE);
        SetDlgItemText(IDC_DISABLE,"允许使用(&E)");
    }
}
```



```

else
{
pClickhere->EnableWindow(TRUE);

SetDlgItemText(IDC_DISABLE,"禁止使用(&X)");

}

bIsEnabled=!bIsEnabled;

}

```

上面的代码实现两个功能，即当位图按钮的状态为允许时，单击按钮 IDC\_DISABLE 将其状态设置为不允许；在相反的状态下，单击按钮 IDC\_DISABLE 将其状态设置为允许。由于实现该过程的代码比较简单，因此我们在这里不作详细的讲述。

编译并运行上面的示例程序，其结果如图6.27所示。

反复单击位图按钮和禁止使用按钮，以观察位图按钮在不同状态下的外观的改变。还可以使用TAB键改变按钮的输入焦点，以观察位图按钮获得输入焦点和失去输入焦点时的不同外观。

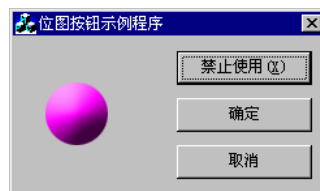


图6. 27 位图按钮示例程序

### 6.3.3 组框

组框也是一种按钮控件。它常常用来在视觉上将控件(典型情况下是一系列的单选钮和复选框)进行分组，从而使对话框中的各个控件看起来比较有条理。



图6. 28 组框(Group box)控件

相对于其它控件来说，组框的使用非常之简单。这里我们需要强调的



是，组框仅仅是在视觉上将控件进行分组，事实上控件在编程上的分组依赖于其Group属性的设置。

组框也可以发送BN\_CLICKED和BN\_DOUBLECLICKED命令消息。但是在般情况下我们都不对这些命令作响应。此外，组框也可以设置Icon或Bitmap属性(注意它们之间的互斥的)，即我们可以使用图标或位图来代替默认情况下的文本。但是在绝大多数情况下，我们仅使用纯文本来作为组框的标题。

与前面讲述的下压按钮类似，我们同样可以使用SetDlgItemText成员函数来设置组框控件的标题文本。此外，我们还可以使用GetDlgItem来获得与组框控件相关联的CWnd对象的指针，然后通过该指针调用成员函数SetWindowText来实现同样的功能。由于在程序中常常不需要频繁的操纵组框控件，因此大多数情况下我们不需要为组框控件进行成员变量的映射，但这种方法是完全可以的。

对于如何将控件进行分组的方法在讲述单选钮和复选框时再作介绍。

#### 6.3.4 单选钮

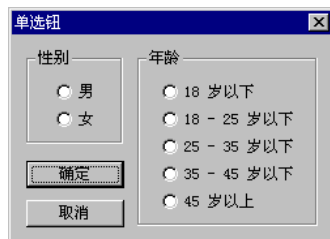


图6. 29 单选钮示例程序

单选钮用来表示一系列的互斥选项，这些互斥选项常常被分成若干个组。下面的示例程序说明了单选钮的使用。



1. 创建新的基于对话框的MFC应用程序，将工程名设置为RadioBut ton。
2. 按图6.29绘制应用程序的主对话框。其中在Control工具箱中单选钮对应的图标是，组框控件对应的图标是.
3. 单击Layout菜单下的Tab Order命令，按图6.30的顺序单击各控件以设置控件的TAB键顺序(Tab Order)。
4. 确信所有控件的Group属性都被设置为假。分别单击组框“性别”和组框“年龄”，将其Group属性设置为真。



图6. 30 设置控件的Tab Order

以Tab Order为序，从Group属性为真的控件开始(包括该控件)，到下一个Group属性的真的控件结束(不包括该控件)，所有的这些控件将组成一个组。对于单选钮，同一组内同时只能有一个处于被选中的状态。当其中一个控件被置于选中状态时，同组的其它单选钮应该清除其选中状态。对于由资源编辑器生成的单选钮控件，在默认情况由Windows自动处理同组控件之间的互斥关系。

下面我们简述一下特定于单选钮的一些属性及其含义，这些属性被列于Styles选项卡内：

- Auto： 在具有Auto属性的情况下，当用户单击了同一组的某个单选钮时，其余单选钮的选中属性被自动清除。当在一组单选钮中使用Dialog Data Exchange时，该属性必须被设置为True。类型：布尔值 默认值：真
- Left text： 将单选钮的标题文本显示于圆形标记的左边。类型：布尔值 默认值：假
- Push-like： 使一个复选框、三态复选框或单选项具有类似于下压按钮的外观和行为。该按钮在选中时显示为凸起，在不被选中时显示为凹下(参见图。类型：布尔值 默认值：假
- Notify： 决定在默认情况下当单选钮被单击或双击时向父窗口发送通知消息。类型：布尔值 默认值：真



图6. 31 具有Push-like样式的单选钮

5. 将性别框内的两个控件的ID按从上到下的顺序设置为IDC\_SEX1和IDC\_SEX2；将年龄框内的两个控件的ID按从上到下的顺序设置为IDC\_AGE1、IDC\_AGE2、IDC\_AGE3、IDC\_AGE4和IDC\_AGE5。

在程序运行时可以调用CButton的成员函数SetCheck设置单选钮的选中状态。该成员函数带有一个类型为整型的参数，该参数为0表示清除选中按钮的选中状态，参数为1表示设置选中按钮的选中状态。

- 注意：

- 如果我们在程序中调用SetCheck设置同一组中某一单选钮的为选中状态，并不意味着同时清除同一组中其它单选钮的选中状态。以前面创建的工程来举例，请看下面的两行代码：

- ((CButton\*)GetDlgItem(IDC\_AGE1))->SetCheck(1);

- ((CButton\*)GetDlgItem(IDC\_AGE5))->SetCheck(1);

上面的代码将导致年龄组中的第一个按钮和第五个按钮在对话框第一次显示时同时处于选中状态。这是应该避免的。因此，如果我们通过代码改变了单选钮的选中状态，一定要记得同时清除同组的其它单选钮的选中状态。

对于单个的单选钮，我们可以调用类CButton的成员函数GetCheck，该函数的返回值为0、1或2，分别代表按钮处理未选中状态、选中状态或中间状态(对三态复选框而言)。但是，对于对话框中的单选钮而言，我们更感兴趣于同一组单选钮中哪一个被选中，因此，调用类CWnd的成员函数GetCheckedRadioButton要更为方便。该成员函数原型如下：

```
int GetCheckedRadioButton( int nIDFirstButton, int nIDLastButton );
```

第一个参数nIDFirstButton是同一组中的第一个单选钮控件的ID，nIDLastButton是同一组中最后一个单选钮控件的ID。成员函数GetCheckedRadioButton返回指定组中第一个所选中的单选钮(在正常情况下仅应当有一个单钮被选中)的ID，如果没有按钮被选中，则返回0。

这里需要注意的是，成员函数GetCheckedRadioButton被没有要求两个参数nIDFirstButton和nIDLastButton所指定的控件一定位于同一组中。

- 注意：
- 若干个单选钮是否属于同一组是以其Tab顺序来排定的，而GetCheckedRadioButton函数是以ID顺序来检查按钮的选定状态的。因此，如果传递给函数GetCheckedRadioButton的第一个参数的值大于第二个参数的值时，其返回值总是为0，而事实上，由这两个参数指定的单选钮的TAB顺序可能恰恰相反。因此，一般情况下我们应该尽量保证同一组单选钮的资源ID是连续递增的。通常这些资源ID是在头文件Resource.h中定义的。如果你同一组的单选钮不是一次创建的，那么它们的资源ID可能不是连续递增的，甚至可能是相反的。这时我们可以手动的修改资源头文件中的宏定义，以保证如GetCheckedRadioButton之类的成员函数得到正确的结果。

同时，这也说明一点，即使用GetCheck一个一个控件的检查各单选钮的选中状态要安全得多。

下面我们来完成应用程序RadioButton。

首先，使用ClassWizard重载类CDialog的OnOK成员函数，方法是重载ID为IDOK的按钮的BN\_CLICKED命令处理函数。由ClassWizard生成的默认重载形式如下：

```
void CRadioBoxDlg::OnOK()
{
    // TODO: 在此添加附加的验证

    CDialog::OnOK();
}
```

这里特定的代码来替代前面的// TODO注释后得到如下的程序代码：

```
void CRadioBoxDlg::OnOK()
{
    // 暂时隐藏主对话框

    ShowWindow(SW_HIDE);

    UINT nSex=GetCheckedRadioButton(IDC_SEX1, IDC_SEX2); // 获得性别选择

    UINT nAge=GetCheckedRadioButton(IDC_AGE1, IDC_AGE5); // 获得年龄选择
```

```
CString msg="性别: "; // 保存输出消息字符串
```

```
// 根据用户的选择生成消息串
```

```
// 添加性别信息
```

```
switch (nSex)
```

```
{
```

```
case IDC_SEX1:
```

```
msg+="男\n";
```

```
break;
```

```
case IDC_SEX2:
```

```
msg+="女\n";
```

```
break;
```

```
default:
```

```
break;
```

```
}
```

```
// 添加年龄信息
```

```
msg+="年龄: ";
```

```
switch (nAge)
```

```
{
```

```
case IDC_AGE1:
```

```
msg+="18 岁以下";
```

```
break;
```

```
case IDC_AGE2:
```

```
msg+="18 - 25 岁";
```

```
break;
```

```
case IDC_AGE3:
```

```
msg+="25 - 35 岁";
```

```

break;

case IDC_AGE4:

msg+="35 - 45 岁";

break;

case IDC_AGE5:

msg+="45 岁以上";

break;

default:

break;

}

msg+="\n\n以上数据是否正确?";

// 显示输入消息框询问用户所输入的信息是否正确

if (MessageBox(msg,NULL,MB_YESNO|MB_ICONQUESTION)==IDNO)

{

// 当用户回答“否”时重新显示对话框以供使用户可以更改所作的选择

ShowWindow(SW_SHOW);

return;

}

// 否则退出应用程序

CDialog::OnOK();

}

```

以上应用程序的运行结果如图6.32所示：

按如图6.32所示进行选择，单击确定弹出如图6.33所示的对话框。

下一节中我们将讲述复选框的使用。

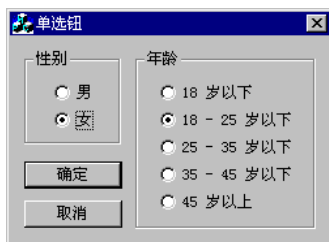


图6. 32 单选钮示例程序的运行结果



图6. 33 单击“确定”之后的确认消息框

### 6.3.5 复选框

复选框与单选钮很相象，不同之处在于在同一组控件中，通常使用复选框来代表多重选择，即选项不是互斥的。从外观上来说，复选框所使用的选中标记是一个方框和方框里面的小叉，而不是单选钮所使用的小圆圈和里面的小点。

对于编程者来说，复选框和单选钮非常相似。我们通过SetCheck成员函数来设置某一复选框的选中状态，通过GetCheck成员函数来获取某一复选框的选中状态。一般来说，对于复选框，由于其选项不是互斥的，我们一般不通过GetCheckedRadioButton之类的函数来获得处于选中状态的按钮。

以下特定于复选框的样式可以在Properties对话框的Styles属性页中进行设置：

- Auto： 对于Auto属性为真的复选框，在单击时将自动在“选中”和“不选中”之间进行切换。如果在一组复选框中使用了Dialog Data Exchange，则必须将该属性设置为真。类型：布尔值 默认值：真
- Tri-state： 创建三态复选框。除了处于“选中”和“不选中”状态外，三态复选框还可以处于变灰状态。通常，态复选框的变灰状态表示其选中状态不确定。在很多软件的安装程序中，变灰往往表示仅选中该组件中的一部分。

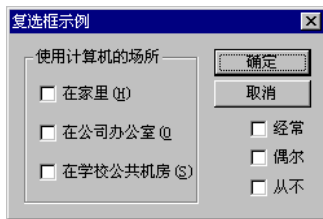


图6. 34 工程CheckBox的主对话框的设计

下面的应用程序举例说明了复选框的使用。


1. 使用默认选项创建一个基于对话框的MFC工程，设置工程名为CheckBox。
2. 按图6.34绘制对话框中的各个复选框(在Control工具箱中复选框所对应的图标为  )，并按表6.24设置各复选框的样式和属性。

表6. 24 工程CheckBox中各控件的属性设置

控件	ID	标题文本	其它
复选框	IDC_PLACE1	在家里(&H)	Auto属性和Tri-state属性均为真
	IDC_PLACE2	在公司办公室(&O)	
	IDC_PLACE3	在学校公共机房(&S)	
	IDC_OFTEN	经常	Auto属性为假，Tri-state属性为真
	IDC_SELDOM	偶尔	
	IDC_NEVER	从不	
组框	IDC_STATIC	使用计算机的场所	

3. 使用下面的代码替换类CCheckBoxDlg的成员函数OnInitDialog中的// TODO注释：

```
((CButton*)GetDlgItem(IDC_OFTEN))->SetCheck(1);
((CButton*)GetDlgItem(IDC_SELDOM))->SetCheck(2);
((CButton*)GetDlgItem(IDC_NEVER))->SetCheck(0);
```

由于三个复选框IDC\_OFTEN、IDC\_SELDOM、IDC\_NEVER的Auto属性值为



假，因此当用户单击这三个复选框时其状态不会发生改变。它们在本示例程序中起了图例的作用。

4. 在类CCheckBoxDlg中重载类CDialog的成员函数OnOK如下(关于对命令处理成员函数OnOK的重载我们已经在前一小节中作了讲述)：

```
void CCheckBoxDlg::OnOK()
{
    // 定义和初始化所用的变量

    CString strMsg, // 消息字符串
    strMsgA[3]; // 分别对应于三种不同时间频度的消息字符串
    int iCount[3]; // 对应于每种时间频度的情况计数

    // 初始化各变量

    iCount[0]=iCount[1]=iCount[2]=0;
    strMsgA[0]="从不在";
    strMsgA[1]="经常在";
    strMsgA[2]="偶尔在";

    int i; // 用着循环变量或中间变量

    // 检查各复选框的选中状态，并根据用户的选择生成对应于三种不同时间
    // 频度的消息字符串

    // 检查复选框 IDC_PLACE1

    i=( (CButton*)GetDlgItem(IDC_PLACE1) )->GetCheck();
    if ( (iCount[i]++)==0 )
        strMsgA[i]+="家里";
    else
        strMsgA[i]+="、家里";

    // 检查复选框 IDC_PLACE2

    i=( (CButton*)GetDlgItem(IDC_PLACE2) )->GetCheck();
    if ( (iCount[i]++)==0 )
```

```

strMsgA[i]+="公司办公室";

else

strMsgA[i]+="、公司办公室";

// 检查复选框 IDC_PLACE3

i=( (CButton*)GetDlgItem(IDC_PLACE3) )->GetCheck();

if ( (iCount[i]++)==0 )

strMsgA[i]+="学校开放机房";

else

strMsgA[i]+="、学校开放机房";

// 为了符合汉语的语气转折，判断是否需要在“从不.....”分句前添加转折

// 连词“但”。如果用户对三种情况的选择都是“从不”，那么这个“但”

// 字是不应该要的。

if ( !(iCount[1]==0 && iCount[2]==0) )

strMsgA[0]=CString("但")+strMsgA[0];

// 如果用户对三种情况的选择都不属于某种时间频度，那么该时间频度所对应

// 的消息字符串应该为空。否则，在该分句的末尾加了字符串“使用计算机，”。

for (i=0;i<3;i++)

{

if ( iCount[i]==0 )

strMsgA[i]="";

else

strMsgA[i]+="使用计算机，";

}

// 生成最终显示的消息字符串

strMsg=CString("您")+strMsgA[1]+strMsgA[2]+strMsgA[0];

// 处理消息字符串的标点

```

```

strMsg=strMsg.Left( strMsg.GetLength()-2 )+"。 ";

// 弹出消息框询问用户所输入的数据是否正确

if ( MessageBox( strMsg,"确认",MB_YESNO|MB_ICONQUESTION )==IDNO )

{

// 如果用户选择“否”，则重新输入数据

return;

}

// 调用基类的 OnOK 成员函数，并关闭对话框

CDialog::OnOK();

}

```

上面的代码都加上了详细的注释，而且所用的函数也都是我们所熟知的，这里我们就不再重复讲述了。

到目前为止，我们已经讲述完了Windows标准控件中的按钮类控件：下压按钮、组框、单选钮和复选框。此外，我们还介绍了位图按钮，一般来说我们并不把它归入Windows标准控件中，而认为它是由MFC提供的少数几个控件之一。而位图按钮事实上是具有Owner draw属性的自绘制下压按钮，MFC类CBitmapButton封装了其内部实现的复杂性，而以简单的接口提供给程序员。

作为本节的结束，我们来讨论一样如何改变按钮标题文本的字体属性。在Developer Studio的资源编辑器中，我们可以统一的修改同一对话框中所有按钮的标题文本的字体属性。方法是打开对话框本身的属性(Properties)对话框，在General选项卡中单击Font按钮，从弹出Select Dialog Font对话框中选择对话框所用的字体。



图6. 35 设置对话框的字体属性

通过上面的方法设置的字体对整个对话框中所有的控件都有效。如果需要设置单个控件的字体，我们必须通过编写代码来实现。下面的示

例程序ButtonFont演示了如何单独更改某个控件的字体。



图6. 36 示例程序ButtonFont的主对话框

按图6.36绘制应用程序主对话框中所有的各按钮控件。其中标签为“我爱你”的按钮ID为IDC\_LOVE，标签为“改变字体”的按钮ID为IDC\_CHANGEFONT。

在类CButtonFontDlg中添加类型为CFont的私有成员变量m\_Font。

为按钮IDC\_CHANGEFONT的BN\_CLICKED事件编写下面的处理函数：

```
void CButtonFontDlg::OnChangeFont()
{
    // 获取按钮 IDC_LOVE 的当前所用字体
    LOGFONT lf;
    GetDlgItem(IDC_LOVE)->GetFont()->GetLogFont(&lf);
    // 使用按钮的当前字体初始化字体对话框
    CFontDialog dlgFontDlg(&lf);
    // 显示字体选择对话框
    if (dlgFontDlg.DoModal()==IDOK)
    {
        // 如果用户在字体选择对话框中单击了“确定”按钮，
        // 则使用
        dlgFontDlg.GetCurrentFont(&lf);
        m_Font.DeleteObject();
        m_Font.CreateFontIndirect(&lf);
        GetDlgItem(IDC_LOVE)->SetFont(&m_Font);
    }
}
```

```
}  
  
}
```

编译并运行程序ButtonFont，单击“改变字体”按钮，在随后弹出的字体选择对话框中设置字体并单击“确定”按钮。对话框的显示可能如图6.37所示。



图6. 37 示例程序ButtonFont的运行结果

- 注意：
- 在示例程序中，如果不定义类CButtonFontDlg的成员变量m\_Font，命令处理函数OnChangeFont可以应该这样编写：
- void CButtonFontDlg::OnChangeFont()  
  
• {  
  
• // 获取按钮 IDC\_LOVE 的当前所用字体  
  
• LOGFONT lf;  
  
• GetDlgItem(IDC\_LOVE)->GetFont()->GetLogFont(&lf);  
  
•  
• // 使用按钮的当前字体初始化字体对话框  
  
• CFontDialog dlgFontDlg(&lf);  
  
•  
• // 显示字体选择对话框  
  
• if (dlgFontDlg.DoModal()==IDOK)  
  
• {  
  
• // 如果用户在字体选择对话框中单击了“确定”按钮，  
  
• // 则将按钮 IDC\_LOVE 的标题文本字体设置为所选定的字体。

- static CFont font;
- dlgFontDlg.GetCurrentFont(&lf);
- font.DeleteObject();
- font.CreateFontIndirect(&lf);
- GetDlgItem(IDC\_LOVE)->SetFont(&font);
- }
- }

按下面的方式编写命令处理函数OnChangeFont不会得到正确的结果：

- void CButtonFontDlg::OnChangeFont()
- {
- ...
- if (dlgFontDlg.DoModal()==IDOK)
- {
- CFont font;
- dlgFontDlg.GetCurrentFont(&lf);
- font.DeleteObject();
- font.CreateFontIndirect(&lf);
- GetDlgItem(IDC\_LOVE)->SetFont(&font);
- }
- }

之所以会出现这种情况与用来设置字体的CFont变量的存活期有关。

## 第四节 静态控件

静态控件一般用来显示静态的文本、图标、位图或图元文件，它不能用来接受用户的输入，也很少用来显示输出，而在更多的情况下用作那些没有固定的标题文本属性的控件(如文本编辑控件、列表框等)的标签，或者用来进行控件的分组，或者用来显示一些提示性文本。

MFC类CStatic封装了标准的Windows静态控件。下面的示例程序StaticDemo演示了静态控件的使用。

1. 使用AppWizard创建一个基于对话框的MFC应用程序，设置其工程名为StaticDemo。
2. 按如图6.38绘制主对话框中的控件。其中标签为“静态控件”的静态控件ID为IDC\_STATIC。需要注意的是，由资源管理器添加的静态控件在默认情况下其ID均为IDC\_STATIC，因此，如果需要在程序中区分和操纵各个不同的静态控件，一般情况下我们都需要更改新添加的静态控件的ID值。这里我们将静态控件的ID值设置为IDC\_STATICDEMO。



图6. 38 示例程序StaticDemo的主对话框

以下属性和样式没有在本章前面的内容中涉及，它们可以适用于静态控件。可以通过静态控件的Properties属性对话框的Styles选项卡进行这些属性或样式的设置。

- |                     |   |
|---------------------|---|
| Align text :        | 决定静态文本控件中文本的横向对齐方式。可供选择的值为Left (向左对齐)、Center (居中对齐)和Right (向右对齐)。默认值 : Left |
| Center Vertically : | 在静态文本控件中将文本进行垂直居中。类型 : 布尔值 默认值 : 假  |
| No prefix :         | 不将控件文本中的“&”符解释为助记字  |

符。在默认情况下，“&”符号在显示时会被去掉，取而代之的是紧接“&”符之后的字符被以加下划线的格式进行显示。我们早在前面说过，通过双写“&”符可以在控件文本中显示出实际的“&”符，但是，对于一些特殊的场合，如使用静态文本控件来显示文件名的时候，将No prefix属性设置为“真”要更方便。

- No wrap : 以左对齐的方式来显示文本，并且不进行文本的自动回行。超出控件右边界的文本将被裁去。需要注意的是，这时即使使用转义字符序列“\n”也不可以强制控件文本进行换行。类型：布尔值 默认值：假
- Simple : 禁止设置Text Align属性和No Wrap样式。在该属性为真的情况下，静态文本控件中的文本不会被自动回行，也不会被剪裁。类型：布尔值 默认值：假
- Notify : 决定控件在被单击时是否通知父窗口。类型：布尔值 默认值：假
- Sunken : 使用静态文本控件看上去有下凹的感觉。类型：布尔值 默认值：假
- Border : 为文本控件创建边框。类型：布尔值 默认值：假

4. 静态控件一般不用于输入，但是如果它的Notify属性设置为真，则当用户单击静态控件时，静态控件将向父窗口发送通知消息。但是，我们不可以使用前面所讲述的方法(即使用ClassWizard或从上下文菜单中选择Events命令)来为静态控件添加消息处理函数。而要以手动的方式来实现这一点。下面我们结合示例StaticDemo来说明如何为静态控件添加单击事件的命令处理程序。在进行下面的步骤之前，请确认静态控件IDC\_STATICDEMO的Notify属性值为真。

在类CStaticDemoDlg的定义处添加下面的命令处理函数声明：

```
afx_msg void OnStaticDemo();
```



最好把成员函数OnStaticDemo的声明与其它命令处理函数的声明放在一起，但不要放到//{{AFX\_MSG和//}}AFX\_MSG之间。

然后，打开类CStaticDemoDlg的实现文件StaticDemoDlg.cpp，在宏

```
BEGIN_MESSAGE_MAP(CStaticDemoDlg, CDialog)
```

和宏

```
BEGIN_MESSAGE_MAP
```

之间添加如下的消息映射入口：

```
ON_BN_CLICKED(IDC_STATICDEMO, OnStaticDemo)
```

同样，不要把手动添加的消息映射入口项放到注释//{{AFX\_MSG\_MAP和//}}AFX\_MSG\_MAP之间。

手动添加成员函数OnStaticDemo或OnDoubleClickedStaticDemo的实现代码：

```
void CStaticDemoDlg::OnStaticDemo()
{
    MessageBox("您刚才单击了“静态控件”！");
}
```

编译上面的示例程序，单击“静态控件”，命令处理函数OnStaticDemo将被调用，从而弹出相应的消息框。

下面我们来看一下如果在静态控件中使用图标和位图。



图6. 39 使用图标代替静态控件中的文本

首先介绍使用图标代替文本的例子，方法如下：

假设对话框类为CStaticDemoDlg，所需使用图标的静态控件ID为IDC\_STATICDEMO，相应的图标的ID为IDR\_MAINFRAME，则可用下面的

代码代替类CStaticDemoDlg的成员函数OnInitDialog中的// TODO注释：

```
// 获得指向静态控件的指针
```

```
CStatic *pStaticDemo=(CStatic*)GetDlgItem(IDC_STATICDEMO);
```

```
// 加载图标
```

```
HICON hIcon=AfxGetApp()->LoadIcon(IDR_MAINFRAME);
```

```
// 设置静态控件的样式以使得可以使用图标，并使图标显示时居中
```

```
pStaticDemo->ModifyStyle(0xF,SS_ICON|SS_CENTERIMAGE);
```

```
// 设置静态控件图标
```

```
pStaticDemo->SetIcon(hIcon);
```

运行该程序，显示如图6.39。

接着我们来看如何使用位图代替文本，方法如下：

假设所用位图的资源ID为IDB\_STATICDEMO，其余设置如上。用以下代码来代替成员函数OnInitDialog中的// TODO注释：

```
// 获得指向静态控件的指针
```

```
CStatic *pStaticDemo=(CStatic*)GetDlgItem(IDC_STATICDEMO);
```

```
// 获得位图句柄
```

```
HBITMAP hBitmap=::LoadBitmap(AfxGetApp()->m_hInstance,
```

```
MAKEINTRESOURCE(IDB_STATICDEMO));
```

```
// 设置静态控件的样式以使得可以使用位图，并使位图在显示时居中
```

```
pStaticDemo->ModifyStyle(0xF,SS_BITMAP|SS_CENTERIMAGE);
```

```
// 设置静态控件显示时使用的位图
```

```
pStaticDemo->SetBitmap(hBitmap);
```


编译并运行该程序，对话框显示如图6.40所示。



图6. 40 使用位图代替文本的静态控件

- 注意：
- 在使用位图的例子中，传递给ModifyStyle的第一个参数的值绝对不可以为0，否则将得不到正常的运行结果。

## 第五节 文本编辑控件

静态文本控件只能用来显示文本，而不可以用来输入文本。如果需要提供输入文本的功能应该使用文本编辑控件。文本编辑控件在Control工具箱中对应的图标为 。对于文本编辑控件，除了我们在前面所涉及的一些外，还可以设置以下的一些属性样式：

Align text :	决定当Multiline属性为真时文本的对齐方式。默认值为：Left
Multi- line :	创建一个多行文本编辑控件。当一个多行文本编辑控件具有输入焦点时，如果用户按下了ENTER键，以默认情况下的行为是选择对话框中的默认命令按钮，而不是向文本编辑控件中插入新行。将AutoHScroll属性或Want return属性设置为真可以将用户按下的ENTER键解释为插入新行，而不是选择默认命令按钮。

在选择了AutoHScroll属性时，如果插入点超过了控件的右边界，多行文本编辑控件自动进行水平滚动。用户可以使用ENTER键来开始新行。

如果没有选择AutoHScroll属性，多行文本编辑控件将视需要将文本进行自动折行。而仅当Want return属性为真时，用

户才可以使用ENTER键来开始新行。

多行文本编辑控件也可以拥有自己的滚动条。具有滚动条的编辑控件处理自己的滚动条消息，而不具有滚动条的编辑控件也可以由父窗口发送的滚动条消息。

类型：布尔值 默认值：假

Number : 用户不能输入非数字字符。类型：布尔值  
默认值：假

Horizontal  
scroll : 为多行控件提供水平滚动条。类型：布尔  
值 默认值：假

Auto  
HScroll : 当用户输入的字符超过了编辑框的右边界  
时自动水平向右滚动文本。类型：布尔值  
默认值：真

Vertical  
scroll : 为多行控件提供垂直滚动条。类型：布尔  
值 默认值：假

Auto  
VScroll : 在多行控件中，当用户在最后一行按下  
ENTER键时自动向上滚动文本

Password : 当用户键入时将所有字符显示为星号  
(\*)。该属性对于多行控件不可用。类  
型：布尔值 默认值：假

No        hide  
selection : 改变当编辑框失去和重新获得焦点时文本  
的显示方式。如果该属性为真，在编辑框  
中选中的文本在任何时候都显示为选中状  
态(即反白状态)。类型：布尔值        默认  
值：假

OEM  
convert : 将键入的文本从Windows字符集转换为OEM  
字符集，再转换回Windows字符集。该操  
作确认应用程序在调用AnsiToOem函数将  
编辑框中的字符串转换为OEM字符串时进  
行正确的字符转换，因此该样式对于包括  
文件名的编辑控件特别有用。类型：布尔  
值 默认值：假

Want return :	指定当用户在多行编辑控件中按下ENTER键时插入一个回车符，否则用户按下ENTER将被解释为选择了对话框中的默认命令按钮。该样式对于单行编辑框控件没有任何影响。类型：布尔值 默认值：假
Border :	在编辑框边缘创建边框。类型：布尔值 默认值：真
Uppercase :	将用户在编辑框中输入的字符转换为大写。类型：布尔值 默认值：假
Lowercase :	将用户在编辑框中输入的字符转换为小写。
	类型：布尔值 默认值：假
Read-only :	防止用户编辑和更改编辑框中的文本。类型：布尔值 默认值：假

相比我们在前面所讲述的几个类CButton、CBitmapButton和CStatic而言，封装标准编辑控件的MFC类CEdit要复杂得多。表给出了在类CEdit中定义的成员函数：

表6. 25 类CEdit中定义的成员函数

成员函数	描述
CEdit	构造CEdit控件对象
Create	创建Windows编辑控件，并将其与CEdit对象相关联
GetSel	获得编辑控件中当前选择的开始和结束字符的位置
ReplaceSel	使用特定的文本来替换编辑控件中的当前选择
SetSel	设置编辑控件中所选定的字符范围
Clear	删除编辑控件中当前选定的字符
Copy	使用CF_TEXT格式将编辑控件中当前选定的文本复制到剪贴板

Cut	删除当前选定的字符，并将所删除的字符复制到剪贴板
Paste	将剪贴板中格式为CF_TEXT的数据 (如果有的话)插入到编辑框中的当前位置。
Undo	撤销最后一次编辑操作
CanUndo	决定编辑控件的操作是否可以被撤销
EmptyUndoBuffer	重置编辑控件的undo标志
GetModify	判断编辑控件中的内容是否被修改过
SetModify	设置或清除编辑控件中的修改标志
SetReadOnly	设置编辑控件的只读状态
GetPasswordChar	当用户输入文本时获得编辑控件中显示的密码字符
SetPasswordChar	设置或移去当用户输入文本时编辑控件中显示的密码字符
GetFirstVisibleLine	获得编辑控件中最上面的可见行
LineLength	获得编辑控件中一行的长度
LineScroll	滚动多行编辑控件中的文本
LineFromChar	获得包含指定索引字符的行的行号
GetRect	获得编辑控件的格式矩形
LimitText	限制用户可以在编辑控件中输入的文本的长度
GetLineCount	获得多行编辑控件中行的数目
GetLine	获得编辑控件中的一行文本
LineIndex	获得多行编辑控件中一行的字符索引
FmtLines	在多行编辑控件中设置是否包含软换行符的开关

续表6.25

成员函数	描述
SetTabStops	在多行编辑控件中设置制表位
SetRect	设置多行文本编辑控件的格式矩形，并更新控件
SetRectNP	设置多行文本编辑控件的格式矩形，但不重绘控件窗口
GetHandle	获得为多行编辑控件分配的内存的句柄
SetHandle	设置供多行编辑控件使用的本地内存句柄
GetMargins	获得当前CEdit对象的左右页边距
SetMargins	设置当前CEdit对象的左右页边距
GetLimitText	获得当前CEdit对象可以包括的最大文本量
SetLimitText	设置当前CEdit对象可以包括的最大文本量
CharFromPos	获得最接近于指定位置的行和字符的索引
PosFromChar	获得指定字符索引的左上角的坐标

上面的成员函数涵盖了编辑控件在使用中的很多方面，可以满足我们在很多情况下的绝大部分需要。这里要注意的是，一些CWnd中定义的成员函数也是很重要的，比如说我们常用CWnd的成员函数GetWindowText和SetWindowText来获取和设置编辑控件的文本，使用成员函数GetFont和SetFont来获取和设置编辑控件显示文本时所使用的字体。

编辑控件可以向父窗口发送的通知消息也要比前面讲述的几种控件多。这些消息有：

ON_EN_CHANGE：	编辑控件不能按选定需要分配足够的内存
ON_EN_ERRSPACE：	
ON_EN_HSCROLL：	用户单击了编辑控件中的水平滚动条。父窗口在屏幕更新前获得此消息
ON_EN_KILLFOCUS：	编辑控件失去输入焦点
ON_EN_MAXTEXT：	当前插入内容超过了编辑

控件中的指定的字符数，该插入内容已被裁剪。如果控件没有设置ES\_AUTOHSCROLL样式，那么在插入的字符超出了编辑控件的宽度也发送该通知消息。同样，如果控件没有指定ES\_AUTOVSCROLL样式，该通知也以插入操作导致总行数超过编辑控件的高度时发送。

ON\_EN\_SETFOCUS :

编辑按钮获得输入焦点

ON\_EN\_UPDATE :

控件已对文本作了格式化，但尚未显示文本。通常可以处理该消息以决定是否需要对窗口的大小作改变等。

ON\_EN\_VSCROLL :

用户单击了编辑控件的垂直滚动条。父窗口在屏幕更新前收到该消息。

示例程序EditDemo演示了编辑控件的一般使用方法。按如下步骤创建该工程：

1. 使用AppWizard创建基于对话框的工程EditDemo。
2. 向工程中添加菜单资源IDR\_MAINMENU，该菜单资源包括两个顶层菜单项“文件(&F)”和“编辑(&E)”，“文件(&F)”下包括如图6.41所示的菜单命令。各菜单命令(不包括具有Separator样式的菜单项)的资源ID依次为ID\_FILE\_NEW和ID\_FILE\_EXIT。“编辑(&E)”菜单下包括如图6.42所示菜单命令。各菜单命令的资源ID依次为ID\_EDIT\_UNDO、ID\_EDIT\_CUT、ID\_EDIT\_COPY、ID\_EDIT\_PASTE、ID\_EDIT\_DEL、ID\_EDIT\_SELECTALL和ID\_EDIT\_SETFONT。

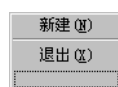



图6. 41 “文件”菜单下的菜单命令





图6. 42 “编辑”菜单下的菜单命令

3. 按图6.43在应用程序的主对话框上绘制编辑框(对应于Control工具箱中的图标为 ), 设置其ID为IDC\_EDIT, 并将其Multiline属性、Auto VScroll属性和Want return属性设置为真, 同时将Auto HScroll属性设置为假。这里, 编辑框IDC\_EDIT在大小和位置并不重要, 我们将在程序中对其进行调整。

4. 删除原有的“确定”按钮和“取消”按钮。接着打开对话框本身的属性对话框, 从Menu下拉列表框中选择IDR\_MAINMENU。



图6. 43 设置主对话框的Menu属性

5. 在资源管理器中打开菜单资源IDR\_MAINMENU, 如图6.44所示。在任一菜单项上单击鼠标右键, 选择命令ClassWizard。这时ClassWizard将弹出如图6.45所示的对话框, 单击Cancel。在Object IDs处选择ID\_FILE\_EXIT, 在Messages处选择COMMAND, 单击And function按钮并接受默认的处理函数名OnFileExit, 在函数OnFileExit中调用类CDialog的成员函数OnCancel, 如下面的代码所示:

```
void CEditDemoDlg::OnFileExit()
{
    // 调用基类成员函数 OnCancel 终止对话框
    OnCancel();
}
```

按同样的方法为ID\_FILE\_NEW的COMMAND命令添加处理函数OnFileNew如下:

```

void CEditDemoDlg::OnFileNew()
{
    // 将编辑控件中的文本初始化为零，
    // 并清除其撤销缓冲区。

    CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);

    pEdit->SetWindowText("");

    pEdit->EmptyUndoBuffer();

}

```

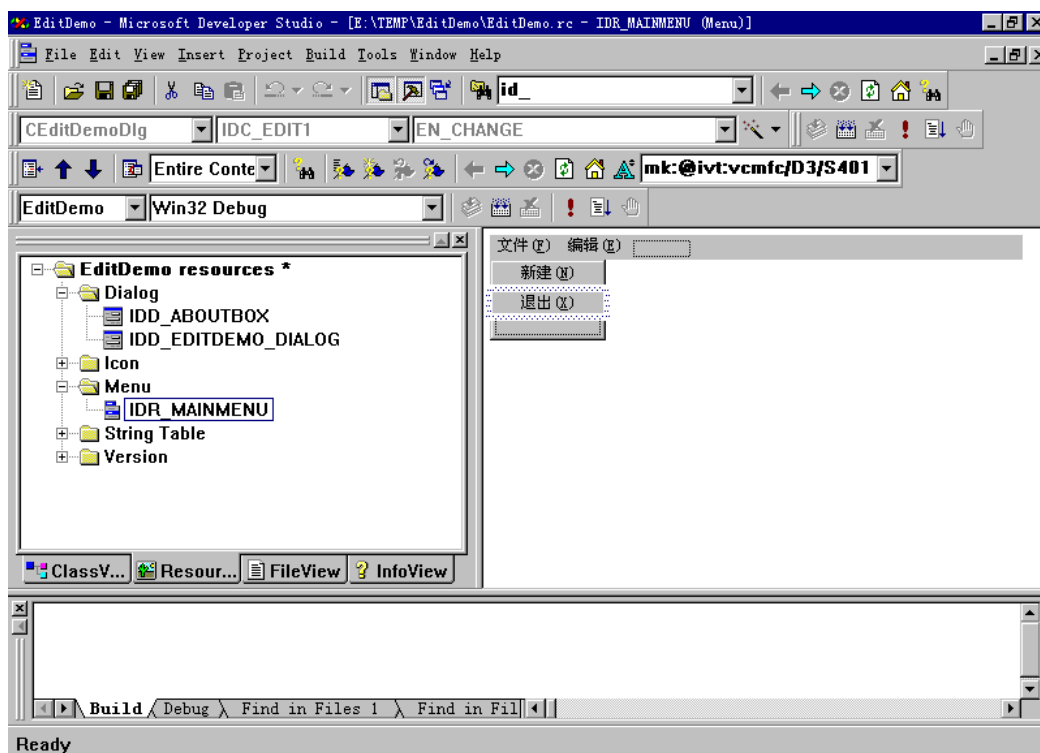


图6. 44 在Developer Studio的资源编辑器中打开菜单资源IDR\_MAINMENU

为ID\_EDIT\_UNDO的COMMAND命令添加处理函数OnEditUndo如下：

```

void CEditDemoDlg::OnEditUndo()
{
    // 直接调用类 CEdit 的成员函数 Undo

    CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);

    pEdit->Undo();

}

```

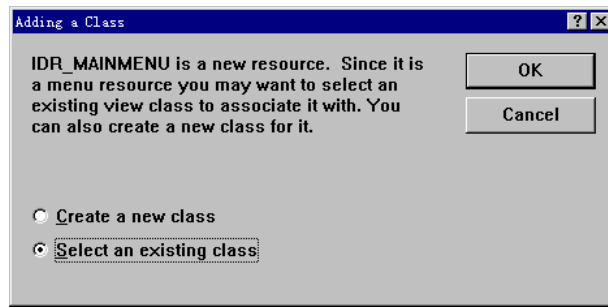


图6. 45 询问是否将菜单IDR\_MAINMENU与某一视类相关联

为ID\_EDIT\_CUT的COMMAND命令添加处理函数OnEditCut如下：

```
void CEditDemoDlg::OnEditCut()
{
    // 直接调用类 CEdit 的成员函数 Cut
    ((CEdit*)GetDlgItem(IDC_EDIT))->Cut();
}
```

为ID\_EDIT\_COPY的COMMAND命令添加处理函数OnEditCopy如下：

```
void CEditDemoDlg::OnEditCopy()
{
    // 直接调用类 CEdit 的成员函数 Copy
    ((CEdit*)GetDlgItem(IDC_EDIT))->Copy();
}
```

为ID\_EDIT\_PASTE的COMMAND命令添加处理函数OnEditPaste如下：

```
void CEditDemoDlg::OnEditPaste()
{
    // 直接调用类 CEdit 的成员函数 Paste
    ((CEdit*)GetDlgItem(IDC_EDIT))->Paste();
}
```

为ID\_EDIT\_DEL的COMMAND命令添加处理函数OnEditDel如下：

```
void CEditDemoDlg::OnEditDel()
```

```

{
// 直接调用类 CEdit 的成员函数 Clear
((CEdit*)GetDlgItem(IDC_EDIT))->Clear();
}

```

为ID\_EDIT\_SELECT的COMMAND命令添加处理函数OnEditSelectall如下：

```

void CEditDemoDlg::OnEditSelectall()
{
int nStart,nEnd;
// 设置选定字符的开始
nStart=0;
// 设置选定字符的结尾。函数 GetWindowTextLength 返回编辑控件中文本的长度
nEnd=((CEdit*)GetDlgItem(IDC_EDIT))->GetWindowTextLength();
// 以 nStart 和 nEnd 为参数调用类 CEdit 的成员函数 SetSel
((CEdit*)GetDlgItem(IDC_EDIT))->SetSel(nStart,nSel);
}

```

为ID\_EDIT\_SETFONT的COMMAND命令添加处理函数OnEditSetFont如下：

```

void CEditDemoDlg::OnEditSetFont()
{
LOGFONT lf;
static CFont font;
// 获得编辑框原来使用的字体信息，并使用该信息初始化字体对话框
CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);
pEdit->GetFont()->GetLogFont(&lf);
CFontDialog dlg(&lf);
// 弹出字体对话框以供用户选择新的字体，

```

// 并在用户确认的情况下更改编辑控件所使用的字体。

```
if (dlg.DoModal()==IDOK)
{
    dlg.GetCurrentFont(&lf);

    font.DeleteObject();

    font.CreateFontIndirect(&lf);

    pEdit->SetFont(&font);
}
}
```

在成员函数OnEditSetFont中所使用的方法和技巧已在第三节的末尾讲述如何为按钮控件设置字体时进行了介绍。因此对于函数OnEditSetFont我们不进行详细的注解。

6. 考虑下面的情况：如果当前没有可供撤消的操作，“编辑”菜单下的“撤消”应该处于不可用(变灰)状态；同样的，如果当前编辑控件中没有选定任何文本，那么“剪贴”、“复制”以及“删除”命令也应该不可用；如果当前剪贴板中没有任何文本数据，“粘贴”命令应该不可用。我们通过为消息WM\_INITMENUPOPUP添加消息处理函数来设置各菜单命令的可用状态。该消息在用户单击某菜单之后在菜单项弹出之前发送。

对于类CEditDemoDlg，我们不能使用ClassWizard来为消息WM\_INITMENUPOPUP添加消息处理函数，但事实上，对话框也可以接收到消息WM\_INITMENUPOPUP。这里，我们可以手动来添加相应的消息映射项。

第一步是在类CEditDemoDlg的定义中添加消息处理函数

```
afx_msg void OnInitMenuPopup( CMenu* pPopupMenu, UINT nIndex, BOOL bSysMenu );
```

可以把该处理函数的声明添加到由ClassWizard生成的消息处理函数的后面。由ClassWizard生成的消息处理函数位于两行注释标记//{{AFX\_MSG和//}}AFX\_MSG之间。同我们在此之前强调过的一样，不要将OnInitMenuPopup的声明添加到两行注释之间。以后如果再遇到与此相似的情况，我们将不再强调。

接着添加相应的消息映射入口，在类CEditDemoDlg的实现文件EditEemoDlg.cpp中找到宏BEGIN\_MESSAGE\_MAP(CEditDemoDlg, CDialog)，在它之后，宏END\_MESSAGE\_MAP之前添加下面的宏代码：

```
ON_WM_INITMENUPOPUP()
```

我们仍应将上面的代码添加到注释标记//{{AFX\_MSG\_MAP和//}}AFX\_MSG\_MAP之外。同样的，以后如果再遇到这种情况我们将不再强调。

最后添加函数OnInitMenuPopup的定义：

```
void CEditDemoDlg::OnInitMenuPopup( CMenu* pPopupMenu, UINT nIndex, BOOL  
bSysMenu )
```

```
{
```

```
CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);
```

```
// 当用户单击的是窗口的控制菜单时 bSysMenu 参数为真，否则为假
```

```
if (!bSysMenu)
```

```
{
```

```
// 检查编辑控件是否有可撤消的操作
```

```
if (pEdit->CanUndo())
```

```
{
```

```
pPopupMenu->EnableMenuItem( ID_EDIT_UNDO, MF_ENABLED);
```

```
}
```

```
else
```

```
{
```

```
pPopupMenu->EnableMenuItem( ID_EDIT_UNDO, MF_GRAYED);
```

```
}
```

```
// 检查编辑控件中是否有选定的文本
```

```
int nStart,nEnd;
```

```
pEdit->GetSel(nStart,nEnd);
```

```
if (nStart==nEnd)
```

```

{
pPopupMenu->EnableMenuItem(ID_EDIT_CUT,MF_GRAYED);
pPopupMenu->EnableMenuItem(ID_EDIT_COPY,MF_GRAYED);
pPopupMenu->EnableMenuItem(ID_EDIT_DEL,MF_GRAYED);
}

else

{
pPopupMenu->EnableMenuItem(ID_EDIT_CUT,MF_ENABLED);
pPopupMenu->EnableMenuItem(ID_EDIT_COPY,MF_ENABLED);
pPopupMenu->EnableMenuItem(ID_EDIT_DEL,MF_ENABLED);
}

// 检查剪贴板中是否有文本格式的数据可供粘贴
// 该过程通过调用 Win32 API 函数 IsClipboardFormatAvailable 来实现
if (IsClipboardFormatAvailable(CF_TEXT))
{
pPopupMenu->EnableMenuItem(ID_EDIT_PASTE,MF_ENABLED);
}

else

{
pPopupMenu->EnableMenuItem(ID_EDIT_PASTE,MF_GRAYED);
}

}

}

```

7. 最后我们希望一点，就是说用户可以改变对话框的大小，而且当用户改变对话框的大小时，编辑框自动的改变其大小以适应父窗口大小的变化。方法是为WM\_SIZE添加消息处理函数。在进行这一步操作之前，打开对话框的Dialog Properties对话框，在Styles选项卡中

将其Border属性设置为Resizing (即可以改变大小)，同时将Maximize box属性值设置为真。然后，使用ClassWizard为消息WM\_SIZE添加消息处理函数OnSize，其定义如下：

```
void CEditDemoDlg::OnSize(UINT nType, int cx, int cy)
{
    // 调用基类的 OnSize 成员函数
    CDialog::OnSize(nType, cx, cy);

    CRect rect;

    // 获得父窗口的客户区矩形
    GetClientRect(&rect);

    CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);

    if (pEdit)
    {
        // 改变编辑控件的大小以适应父窗口大小的改变
        pEdit->MoveWindow(&rect);
    }
}
```

由于OnSize会在对话框第一次显示时被调用，因此使用if语句检查pEdit是否为NULL是必要的。出于同样的目的，我们还需要使用下面的代码来替换成员函数OnInitDialog中的// TODO注释：

```
CRect rect;

GetClientRect(&rect);

CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT);

if (pEdit)
{
    pEdit->MoveWindow(&rect);
}
```



它在对话框第一次显示时完成与上面的OnSize成员函数同样的操作。以保证在第一次显示对话框时编辑框控件以正确的大小进行显示。

编译并运行上面的程序(如图6.46),并测试其各项功能是否正常。



图6. 46 示例程序Edit Demo的运行结果

## 第六节 列表框控件

列表框控件通常用来列出一系列可供用户从中进行选择的项，这些项一般来说都在字符串的形式给出，但也可以采用其它的形式，如图形等。列表框可以只允许单一选择，也就是说用户同时只能选择所有列表项中的一项；除此之外，列表框也可以是多项选择的，用户可以在多项选择列表框中选择多于一项的列表项。当用户选择了某项时，该项被反白显示，同时列表框向父窗口发送一条通知消息。MFC类CListBox封装了Windows标准列表框控件，其成员函数(参见表6.26)提供了对标准列表框的绝大多数操作。

表6. 26 在类CListBox中定义的成员函数

成员函数	描述
AddString	向列表框中添加字符串
CharToItem	为不包含字符串的自绘制列表框提供对 WM_CHAR 的定制处理
CListBox	构造一个 CListBox 对象
CompareItem	由框架调用以决定新添加的项在有序自绘制列表框中的位置
Create	创建一个 Windows 列表框控件，并将它与 CListBox 对象相关联

DeleteItem	当用户从自绘制列表框中删除一项时由框架调用
DeleteString	从列表框中删除字符串
Dir	从当前目录向列表框中添加文件名
DrawItem	当自绘列表框的可视部分改变时由框架调用
FindString	在列表框中查询指定的字符串
FindStringExact	查找与指定字符串相匹配的第一个列表框字符串
GetAnchorIndex	返回列表框中当前“锚点”项的基于零的索引

续表6.26

成员函数	描述
GetCaretIndex	在多重选择列表框中获得当前拥有焦点矩形的项的索引
GetCount	返回列表框中字符串的数目
GetCurSel	返回列表框中当前选择字符串的基于零的索引值
GetHorizontalExtent	以像素为单位返回列表框横向可滚动的宽度
GetItemData	返回下列表框项相关联的32位值
GetItemDataPtr	返回指向列表框项的指针
GetItemHeight	决定列表框中项的高度
GetLocale	获得列表框使用的区域标识符
GetSel	返回列表框项的选定状态
GetSelItems	返回当前选定字符串的索引
GetSelCount	在多重选择列表框中获得当前选定字符串的数目
GetText	拷贝列表框项到缓冲区
GetTextLen	以字节为单位返回列表框项的长度
GetTopIndex	返回列表框中第一个可视项的索引


InitStorage	为列表框项和字符串预先分配内存
InsertString	在列表框中的指定位置插入一个字符串
ItemFromPoint	返回与指定点最接近的列表框项的索引
MeasureItem	当自绘列表框创建时由框架调用以获得列表框的尺寸
ResetContent	从列表框中清除所有的项
SelectString	从单项选择列表框中查找并选定一个字符串
SetItemRange	在多重选择列表框中选中某一范围的字符串或清除某一范围的字符串的选定状态
SetAnchorIndex	在多重选择列表框的设置扩展选定的起点(“锚点”项)
SetCaretIndex	在多重选择列表框中设置当前拥有焦点矩形的项的索引
SetColumnWidth	设置多列列表框的列宽
SetCurSel	在列表框中选定一字符串
SetHorizontalExtent	以像素为单位设置列表框横向可滚动的宽度
SetItemHeight	设置列表框中项的高度
SetItemRect	返回列表框项当前显示的边界矩形
SetLocale	为列表框指定区域标识符

续表6.26

成员函数	描述
SetSel	在多重选择列表框中选定一列表框项或清除某一列表框项的选定状态
SetTabStops	设置列表框的制表位
SetTopIndex	设置列表框中第一个可视项的基于零的索引
VKeyToItem	为具有LBS_WANTKEYBOARDINPUT样式的列表框提供定制的WM_KEYDOWN消息处理

以下是列表框可能向父窗口发送的通知消息及其说明：

ON_LBN_DLBCLK：	用户双击了列表框中的字符串。仅当列表框具有LBS_NOTIFY样式时会发送该通知消息
ON_LBN_ERRSPACE：	列表框不能按需要分配足够的内存
ON_LBN_KILLFOCUS：	列表框失去输入焦点
ON_LBN_SELCANCEL：	列表框中的当前选择被取消。仅当列表框具有LBS_NOTIFY样式时才会发送该通知消息
ON_LBN_SELCHANGE：	列表框中的选择将被更新。需要注意的是，当使用成员函数CListBox::SetCurSel时不会发送该通知消息，同时，该消息也仅当列表框具有LBS_NOTIFY样式才会发送。对于多重选择列表框，当用户按下光标键时，即使所选择的内容没有改变，也会发送LBN_SELCHANGE通知消息。
ON_LBN_SETFOCUS：	列表框获得输入焦点
ON_WM_CHARTOITEM：	不包括字符串的列表框收到WM_CHAR消息
ON_WM_VKEYTOITEM：	具有LBS_WANTKEYBOARDINPUT样式的列表框接收到WM_KEYDOWN消息

在资源编辑器中，对应于列表框的Control工具箱图标为。在绘制列表框的同时可以在Properties属性对话框中指定其属性。除了在前面几节中所讲述的以外，我们还可以为列表框设置以下的属性，这些属性可以在Styles选项卡中设置。

Selection： 决定列表框的选择方式。可以设置的值如下：

Single：用户同时只能选择列表框中

的一项

Multiple：用户可以同时选择多于一个的列表框项，但不可以从开始项扩展选定内容。在鼠标单击时可以使用SHIFT键和CTRL键选定和取消选定，同时选定项不一定需要连续。单击或双击未选定项时将选定该项；单击或双击已选定项时将取消对该项的选定。

Extended：用户可以通过拖动来扩展选定内容。用户可以鼠标和SHIFT键和CTRL键进行选定或取消选定，选择成组的项或不连续的项。

默认值为Single。

Owner  
draw：

控制列表框的自绘特性。可以设置的值如下：

No：关闭自绘制样式，列表框中包含的内容为字符串。

Fixed：指定列表框的所有者负责绘制其内容，并且列表框中的项具有相同的高度。

Variable：指定列表框的所有者负责绘制其内容，并且列表框的项具有不同的高度。

当列表框创建时CWnd::OnMeasureItem将被调用；当列表框的可视部分改变时CWnd::OnDrawItem将被调用。

默认值为No。

Has  
strings：

指定自绘制列表框包括由字符串组成的项。列表框为字符串维护内存和指针，因此应用程序可以使用LB\_GETTEXT消息来获得特定项的文本。在默认情况下，除了自绘制按钮以外，所有的列表框都具有该项

	属性。由应用程序创建的自绘制列表框可以具有或不具有该样式。
	该样式仅当自绘制属性被设置为Fixed或Variable时可用。如果自绘制属性被设置为No，列表框在默认情况下包括字符串。
	类型：布尔值 默认值为假
Sort :	以字母为序对列表框内容进行排序。
	类型：布尔值 默认值为真
Notify :	如果列表项被单击或双击时通知父窗口。
	类型：布尔值 默认值为真
Multi-colum :	指定多列列表框，多列列表框可以在水平方向上进行滚动。消息 LB_SETCOLUMNWIDTH用来设置列宽。
	类型：布尔值 默认值为假
Horizontal scroll :	创建具有水平滚动条的列表框。类型：布尔值 默认值为假
Vertical scroll :	创建具有垂直滚动条的列表框。类型：布尔值 默认值为真
No redraw :	指定当发生改变时列表框外观不进行更新。可以通过发送WM_SETREDRAW消息或调用CWnd::SetRedraw函数改变该属性。
	类型：布尔值 默认值为假
Use tabstops :	允许列表框在绘制字符串辨认和扩展制表符。默认的制表位为32个对话框单位(DLU)。类型：布尔值 默认值为假
Want key input :	指定当用户有按键动作并且列表框具有输入焦点时列表框的所有者收到WM_VKEYTOITEM和WM_CHARTOITEM消息，以允许应用程序在使用键盘输入时进行特定的处理。如果列表框具有了Has Strings

样式，列表框将接收到WM\_VKEYTOITEM消息；如果列表框不具有WM\_CHARTOITEM消息，则列表框将接收到WM\_CHARTOITEM消息。

类型：布尔值 默认值为假

Disable no scroll：当列表框不具有足够多的项时显示不可用的滚动条。如果不使用该属性，在这种情况下将不使用滚动条。类型：布尔值 默认值为假

No integral height：设置对话框的大小严格等于创建对话框时由应用程序指定的大小。一般情况下，Windows改变列表框的大小以使得它不会只显示某一项的一部分，即列表框客户区的高度为项高的整数倍。

类型：布尔值 默认值为真

下面的示例程序演示了列表框控件的使用。

- 1. 使用AppWizard创建名为ListBoxDemo的基于对话框的MFC应用程序工程。
- 2. 按图6.47设计应用程序的主对话框。各控件的属性值如表6.27所示。



图6. 47 应用程序ListBoxDemo的主对话框

- 3. 单击Insert菜单下的Resource命令，插入ID为IDD\_INPUT的对话框，按图6.48添加对话框的各个控件。

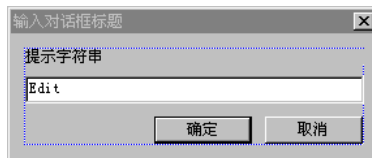


图6. 48 应用程序ListBoxDemo的IDD\_INPUT对话框

各控件的属性如表6.28所示。

4. 为对话框IDD\_INPUT创建新的对话框类CInputDialog。方法是在资源编辑器中打开对话框IDD\_INPUT，此时按下Ctrl+W键打开ClassWizard，由于尚没有类与对话框IDD\_INPUT相关联，因此ClassWizard将弹出如图6.49所示的对话框，询问是否为对话框创建新的类。选择Create a new class（这也是默认选项），单击OK，弹出如图6.50所示的New class对话框。在Name处输入CInputDialog，其余采用默认设置，单击OK即为对话框IDD\_INPUT创建了新类CInputDialog。这时就可以使用ClassWizard的Member Variables选项卡为对话框进行如表6.28所示的成员变量映射了。

表6. 27 应用程序ListBoxDemo主对话框各控件的属性设置

控件类型	资源ID	控件标题	其他
列表框	IDC_LISTSELECTABLE		位于图6.47左边的列表框，其Selection属性为Extended。对应的DDX变量映射(使用ClassWizard的Member Variables选项卡进行设置)为CListBox类型变量m_IsSelectable。
	IDC_LISTSELECTED		位于图6.47右边的列表框，其Selection属性为Extended。对应的DDX变量映射为CListBox类型变量m_IsSelected。
静态控件	(无需更改)	待选择的文件	
		已选择的文件	



下压按钮	IDC_BTNCHANGEDIR	<- 改变目录 (&H)	
	IDC_BTNADD	添加到 (&A) ->	
	IDC_BTNDL	删除 (&D) <-	
	IDC_BTNCLEAR	全部清除 (&L) <-	

表6. 28 对话框IDD\_INPUT各控件的属性设置

控件类型	资源ID	控件标题	其他
静态控件	IDC_PROMPT	提示字符串	对应的DDX变量映射为CString类型成员变量m_strPrompt
编辑框	IDC_INPUT		对应的DDX变量映射为CString类型成员变量m_strInput

接着为类添加类型为CString的保护成员变量m\_strTitle。然而在类CInputDialog中添加成员函数GetInput的声明：

```
CString GetInput(LPCTSTR lpszTitle="输入",
LPCTSTR lpszPrompt="请在下面的文本框中输入字符串：");
```

该函数显示对话框IDD\_INPUT（如图6.51所示），并返回用户在对话框中输入的字符串，如果用户单击了输入对话框的“取消”按钮，则函数返回空字符串，参数lpszTitle为输入对话框的标题，lpszPrompt为输入对话框的提示字符串。其实现如下：

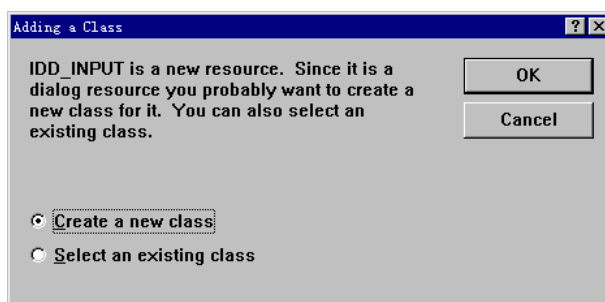


图6. 49 询问是否为对话框IDD\_INPUT创建新类

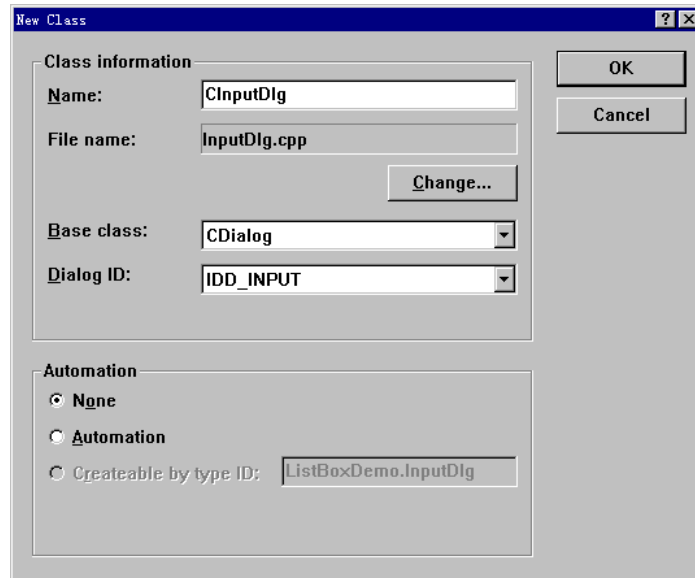


图6. 50 为对话框IDD\_INPUT创建新类

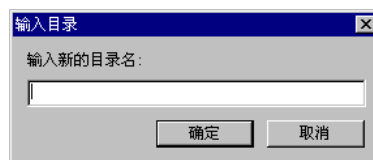


图6. 51 输入对话框

```
CString CInputDialog::GetInput(LPCTSTR lpszTitle, LPCTSTR lpszPrompt)
{
    // 设置标题字符串和提示字符串
    m_strTitle=lpszTitle;
    m_strPrompt=lpszPrompt;

    // 显示输入对话框并返回用户输入的字符串
    if (DoModal()==IDOK)
    {
        return m_strInput;
    }
    else
    {

```

```

return CString("");
}
}

```

## 为类CInputDialog重载OnInitDialog成员函数

```

BOOL CInputDialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: 在这里添加额外的初始化代码

    SetWindowText(m_strTitle);

    GetDlgItem(IDC_INPUT)->SetFocus();

    // 由于为控件 IDC_INPUT 设置了输入焦点，因此函数 OnInitDialog 应该返回 FALSE

    return FALSE;
}

```

成员函数OnInitDialog的重载版本设置输入对话框的标题文本和提示字符串。

5. 用下面的代码替代类CListBoxDemoDlg的OnInitDialog成员函数中的// TODO注释：

```

m_IsSelectable.ResetContent();

m_IsSelectable.Dir(0x17, "*. *");

```

上面的代码先调用成员函数ResetContent清除列表框IDC\_LISTSELECTABLE中的所有项，再调用成员函数Dir使用当前目录下的文件名来填充该列表框。第一个参数0x17是文件类型屏蔽位，它等于0x01|0x02|0x04|0x10，它包括了所有常规属性文件、只读文件、系统文件和目录名，第二个参数为所显示的文件名，在参数中可以使用通配符。

为按钮IDC\_BTNCHANGEDIR的BN\_CLICKED命令添加下面的处理函数OnBtnChangeDir：

```

void CListBoxDemoDlg::OnBtnChangeDir()

```

```

{
CInputDialog dlg;

CString str=dlg.GetInput("输入目录","输入新的目录名:");

if (str!=" " && str.Left(1)!="\\")

{

str+="\\";

}

if (str!="")

{

m_lsSelectable.ResetContent();

int iResult=m_lsSelectable.Dir(0x17,str+"*.*");

if (iResult==LB_ERR)

{

MessageBox("添加文件名出错!");

}

else if (iResult==LB_ERRSPACE)

{

MessageBox("无法为列表框分配足够的内存!");

}

}

}

```

上面的代码首先定义一个类型为CInputDialog的成员变量，然后调用其成员函数GetInput（我们已在前面讨论过该成员函数）获得用户输入的列表目录名，如果用户输入的目录名不为空字符串，则调用类CListBox的成员函数将指定目录下的文件名添加到列表框IDC\_LISTSELECTABLE中，如果添加失败，则弹出相应的出错信息。

为按钮IDC\_BTNADD的BN\_CLICKED命令添加下面的处理函数OnBtnAdd：

```

void CListBoxDemoDlg::OnBtnAdd()
{
    CString str;
    for (int i=0; i<m_IsSelectable.GetCount(); i++)
    {
        if (m_IsSelectable.GetSel(i))
        {
            m_IsSelectable.GetText(i, str);
            m_IsSelected.AddString(str);
        }
    }
}

```

其中，类CListBox的成员函数GetCount返回了列表框中项的数目，然后使用GetSel成员函数获得每一项的选定状态，这里要注意列表框中项的索引是基于零的。如果该项的已被选定(即GetSel成员函数返回真值)，则使用GetText成员函数获得该项的文本，并将它放到CString类型的变量str中，接着，调用类CListBox中定义的成员函数AddString将字符串str添加到列表框IDC\_LISTSELECTED中。

为按钮IDC\_BTNDL的BN\_CLICKED命令添加如下的处理函数OnBtnDel：

```

void CListBoxDemoDlg::OnBtnDel()
{
    for (int i=m_IsSelected.GetCount()-1; i>=-1; i--)
    {
        if (m_IsSelected.GetSel(i))
        {
            m_IsSelected.DeleteString(i);
        }
    }
}

```

```
}
```

上面的代码从最末一项开始，检查列表框IDC\_LISTSELECTED中每一项的选定状态，如果发现该项被选定，则将它从列表框中删除。从列表框中删除一项使用类CListBox的成员函数DeleteString，其参数为所删除项的索引值。

- 注意：
- 我们在上面的代码中使用的for循环为
- for (int i=m\_IsSelected.GetCount()-1; i>=0; i--)
- {
- ...
- }

而不是

- for (int i=0; i<m\_IsSelected.GetCount(); i++)
- {
- ...
- }

这是因为成员函数DeleteString的使用将导致所删除项之后的所有项的索引值发生改变，这里，如果所删除的项的下一项仍被选定的话，该项将不会被删除。与此相反，删除一项并不会导致此项之前的项的索引值发生改变，因此，从最末一项开始进行检查是可行的。

按钮IDC\_BTNCLEAR的BN\_CLICKED命令的处理成员函数OnBtnClear具有最简单的结构，它直接调用类CListBox的成员函数ResetContent删除列表框IDC\_LISTSELECTED中的所有项。

```
void CListBoxDemoDlg::OnBtnClear()
{
    m_IsSelected.ResetContent();
}
```

}




图6. 52 示例程序ListBoxDemo的运行结果

编译并运行上面的示例程序，其运行结果如图6.52所示。单击“改变目录”按钮，输入一个新的目录名，查看左边列表框中项的改变情况。从左边列表框中选定若干项，单击“添加到”，将所选定的项添加到右边列表框(注意列表框中可以包括相同字符串的项)。再从右边列表框中选定若干项，验证按钮“删除”和“全部清除”是否正常工作。

## 第七节 组合框

组合框(combo box)可以看作是一个编辑框或静态文本框与一个列表框的组合，组合框的名称也正是由此而来。当前选定的项将显示在组合框的编辑框或静态文本框中。如果组合框具有下拉列表(drop-down list)样式，则用户可以在编辑框中键入列表框中某一项的首字母，在列表框可见时，与该首字母相匹配的最近的项将被加亮显示。

组合框对应于Controls工具箱内的按钮为。在绘制组合框的同时可以使用控件的Properties对话框设置控件的各种属性样式。一些样式已在前面的几节中作了介绍，因此这里不再重复，下面给出一些在前面的内容中没有进行说明的样式及其含义：

Type：指定组合框的类型。可以使用的类型如下：

Simple：创建包括编辑框控件和列表框的简单组合框，其中编辑框控件用来接受用户的输入。

Dropdown：创建下拉组合框。该类型

与简单组合框类似。但仅当用户单击了编辑框控件部分右边的下拉箭头时组合框的列表框部分才被显示。

Drop List：该类型类似于下拉样式(drop-down)，只是使用静态文本项代替编辑框控件来显示列表框中的当前选择。

默认值为DropDown。

Uppercase：将选择域或列表中的所有文本转换为大写。

类型：布尔值 默认值为假

Lowercase：将选择域或列表中的所有文本转换为小写。

类型：布尔值 默认值为假

与列表框不同的是，在绘制组合框的同时可以预先为组合框添加一些可选项，方法是单击Properties对话框中的Data选项卡(如图所示)，直接在Enter listbox items处键入组合框中的可选项，每一行为一个选项，使用Ctrl+Enter键开始新的一行。在运行时这些选项将出现在组合框的列表框中。

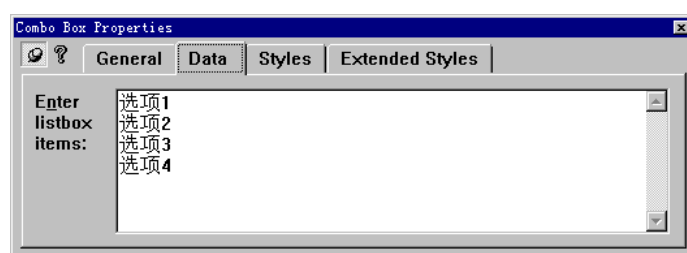


图6. 53 为组合框预置选项

MFC类CComboBox封装了Windows标准组合框，其成员函数提供了对组合框控件的常见操作的实现。表给出了对在类CListBox中定义的成员函数的描述。

表6. 29 在类CListBox中定义的成员函数

成员函数	描述
------	----



CComboBox	构造一个CComboBox对象
Create	创建一个组合框并将它与CComboBox对象相关联
InitStorage	为组合框的列表框部分的项和字符串预先分配内存块
GetCount	获得组合框中列表框项的数目
GetCurSel	如果存在的话，返回组合框中列表框的当前选定项的索引
SetCurSel	选择组合框中列表框内的一条字符串
GetEditSel	获得组合框中编辑控件的当前选定的起始和终止字符位置
SetEditSel	在组合框的编辑控件中选定字符
SetItemData	设置与组合框中指定项相关联的32位值
SetItemDataPtr	将与组合框中指定项相关联的32位值设置为指定的void指针
GetItemData	获得由应用程序提供的与指定组合框项相关联的32位值
GetItemDataPtr	以void指针的形式返回由应用程序提供的与指定组合框项相关联的32位值
GetTopIndex	返回组合框中列表框部分的第一个可视项的索引
SetTopIndex	在组合框中的列表框部分的顶部显示指定索引对应的项
SetHorizontalExtent	以像素为单位指定组合框的列表框部分可以横向滚动的宽度
GetHorizontalExtent	以像素为单位获得组合框中列表框部分可以横向滚动的宽度
SetDroppedWidth	为组合框的下拉列表框部分设置最小允许宽度
GetDroppedWidth	获得组合框的下拉列表框部分的最小允许宽度
Clear	如果存在的话，删除编辑控件中当前选定的内容

Copy	如果存在的话，将当前选定以CF_TEXT格式复制到剪贴板
Cut	如果存在的话，删除编辑控件中当前选定的内容，并将其以CF_TEXT格式复制到剪贴板
Paste	当剪贴板包括CF_TEXT格式的数据时，从剪贴板复制数据到编辑控件的当前插入位置
LimitText	设置用户可以在组合框的编辑控件中输入的文本的长度限制
SetItemHeight	设置组合框中列表项的高度或编辑控件(或静态文本控件)部分的高度
GetItemHeight	获得组合框中列表项的高度
GetLBText	从组合框中的列表框获取字符串

续表6.29

成员函数	描述
GetLBTextLen	获得组合框的列表框中某一字符串的长度
ShowDropDown	对于具有CBS_DROPDOWN或CBS_DROPDOWNLIST属性的组合框，显示或隐藏其列表框
GetDroppedControlRect	获得下拉组合框的可视(下拉)列表框的屏幕坐标
GetDroppedState	判断下拉组合框的列表框是否可见(处理下拉状态)
SetExtendedUI	对于具有CBS_DROPDOWN或CBS_DROPDOWNLIST样式的组合框，选择默认用户界面或扩展用户界面
GetExtendedUI	判断组合框具有默认用户界面还是扩展用户界面
GetLocale	获得组合框的区域标识符
SetLocale	设置组合框的区域标识符
AddString	向组合框的列表框添加一字符串，对于具有CBS_SORT样式的组合框，新增加的字符串将被排序并插入到合适的位置，否则将被添加到列表框框的末尾

DeleteString	从组合框的列表框中删除字符串
InsertString	向组合框的列表框中插入一字符串
ResetContent	清除组合框的列表框和编辑控件中的所有内容
Dir	添加文件名列表到组合框的列表框中
FindString	在组合框的列表框中查找包括指定前缀的第一个字符串
FindStringExact	在组合框的列表框中查找与指定字符串匹配的字符串
SelectString	在组合框的列表框中查找字符串，如果找到的话，在列表框中选择该字符串，并将字符串复制到编辑控件中
DrawItem	当一个自绘制组合框的可视部分改变时由框架调用
MeasureItem	在创建自绘制组合框时，由框架调用以判断组合框的尺寸
CompareItem	当将一新项插入到排序的自绘制框中时由框架调用以判断项的相对位置
DeleteItem	当一列表项被从自绘制组合框中删除时由框架调用

下面的示例程序演示了自绘制组合框的使用。

1. 使用AppWizard创建名为ComboDemo的基于对话框的工程，按图6.54添加工程的主对话框( IDD\_COMBODEMO\_DIALOG)中的各个控件。每个控件的属性如表6.30所示。
2. 在ClassView中用鼠标右击ComboDemo classes，选择New Class命令。上面的操作将弹出如图6.55所示的对话框，确认在Class type下拉列表框[注] 中选择了MFC Class。然后在Name处输入新的类名CClrComboBox，在Base class下拉列表框中选择CComboBox。如果需要修改新类的头文件或实现文件的文件名，可以单击Change按钮，这里，我们接受默认的文件名ClrComboBox.cpp和ClrComboBox.h。

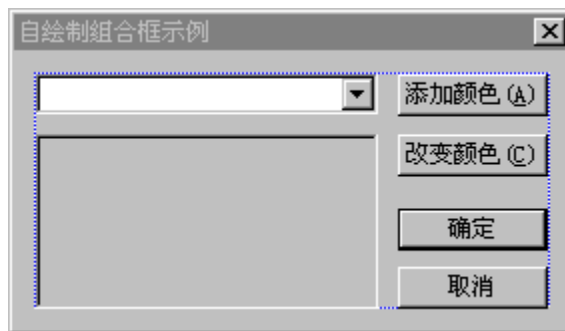


图6. 54 工程ComboDemo的主对话框

表6. 30 对话框IDD\_COMBODEMO\_DIALOG的控件属性设置

控件类型	ID	属性值
组合框	IDC_CLRCOMBO	Type : Dropdown Owner draw : Fixed Sort : 真 Vertical scroll : 真 Has string : 假
下压按钮	IDC_ADDCLR	Caption : 添加颜色(&A)
	IDC_CHGCLR	Caption : 改变颜色(&C)
静态控件	IDC_STATICCLR	Caption属性值为空

3. 使用ClassWizard的Message Map选项卡在类CClrComboBox中重载基类的MeasureItem成员函数，其重载版本的代码如下：

```
void CClrComboBox::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
// 由于组合框具有 CBS_OWNERDRAWFIXED 样式，因此以 0 为参数调用成员函数
// GetItemHeight 获得每一项的固定高度
lpMeasureItemStruct->itemHeight=GetItemHeight(0);
```

}

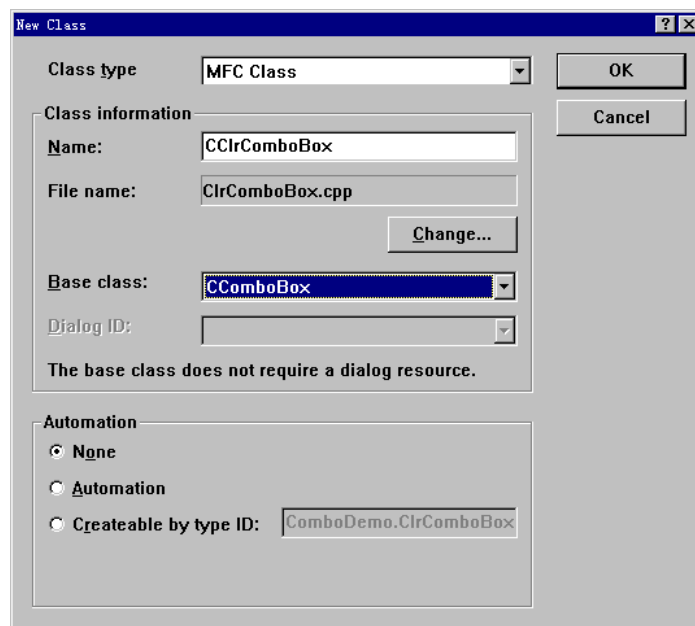


图6. 55 从CComboBox派生新类CClrComboBox

函数MeasureItem在自绘制样式的组合框创建时由框架调用。该函数将每一项的高度放入MEASUREITEMSTRUCT结构的成员中。如果对话框以CBS\_OWNERDRAWVARIABLE样式创建，框架将为列表框中的每一项调用一次该成员函数，否则，该成员函数只被调用一次。

接着，在CClrComboBox的重载基类的DrawItem成员函数，其代码如下：

```
void CClrComboBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC* pDC=CDC::FromHandle(lpDrawItemStruct->hDC);
    COLORREF cr=(COLORREF)lpDrawItemStruct->itemData;

    // 注意到在出错的情况下，GetCurSel 和 GetItemData 返回 CB_ERR，而常量
    // CB_ERR 被定义为 -1，这时不应把它视为一种系统颜色。

    if (cr==CB_ERR)
    cr=GetSysColor(COLOR_WINDOW);

    if (lpDrawItemStruct->itemAction & ODA_DRAWENTIRE)
    {
```

```

// 需要重绘整个项

// 以该项所对应的颜色填充整个项

CBrush br(cr);

pDC->FillRect(&lpDrawItemStruct->rcItem, &br);

// 反色居中显示该颜色的 RGB 组成

CString str;

str.Format("R: %d G: %d B: %d", GetRValue(cr), GetGValue(cr), GetBValue(cr));

CSize size;

size=pDC->GetTextExtent(str);

CRect rect=lpDrawItemStruct->rcItem;

COLORREF tcr;

tcr=~cr & 0x0FFFFFFF; // 获得背景色的反色，不能简单的使用 ~cr

pDC->SetTextColor(tcr);

pDC->SetBkColor(cr);

pDC->TextOut(rect.left+(rect.Width()-size.cx)/2,

rect.top+(rect.Height()-size.cy)/2, str);

}

if ((lpDrawItemStruct->itemState & ODS_SELECTED) &&

(lpDrawItemStruct->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))

{

// 选中状态由未选中变为选中，其边框被加亮显示

COLORREF crHilite=~cr & 0x0FFFFFFF;

CBrush br(crHilite);

pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);

}

if (!(lpDrawItemStruct->itemState & ODS_SELECTED) &&

```

```

(lpDrawItemStruct->itemAction & ODA_SELECT))

{

// 选中状态由选中变为非选中，清除其边框的加亮显示

CBrush br(cr);

pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);

}

}

```

对于自绘制组合框来说，成员函数DrawItem是需要重载的一个很重要的成员函数。该函数在自绘制组合框的可视部分发生改变时由框架调用。在默认情况下，该成员函数不做任何操作。其参数lpDrawItemStruct所指向的DRAWITEMSTRUCT结构包括了重绘制所需要的各种信息，如所需重绘的项、其设备上下文以及所执行的重绘行为等。在该成员函数终止前，应用程序应该恢复由该DRAWITEMSTRUCT结构所提供的为该显示上下文所选定图形设备接口。

由表6.30可知在本示例程序中所使用的自绘组合框中的可选项是有序的，而它们都是一些颜色值，框架如何知道当一个新的颜色值被添加到组合框的列表框中时，它应该处于哪个颜色值之前，哪个颜色值之后呢？这时通过调用成员函数CompareItem成员函数来实现的。如果在创建组合框时指定了LBS\_SORT样式，则必须重载该成员函数以提供足够的理由来帮助框架对新添加入组合框的列表框中的颜色项进行排序。这里，我们首先根据颜色亮度的大小来对颜色进行排序，对于亮度相同的颜色，我们依次以从蓝色到红色的优先级来判定其相对位置。这个操作是以下面的代码来实现的：

```

int CClrComboBox::CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct)

{

// TODO: 添加判断指定项的排序顺序的代码

// 当项 1 在项 2 之前时返回 -1

// 当项 1 和项 2 顺序相同时返回 0

// 当项 1 在项 2 之后时返回 1

// 获得项 1 和项 2 的颜色值

COLORREF cr1 = (COLORREF)lpCompareItemStruct->itemData1;

```

```

COLORREF cr2 = (COLORREF)lpCompareItemStruct->itemData2;

if (cr1 == cr2)
{
// 项 1 和项 2 具有相同的颜色

return 0;
}

// 进行亮度比较，亮度低的排列顺序在前

int intensity1 = GetRValue(cr1) + GetGValue(cr1) + GetBValue(cr1);
int intensity2 = GetRValue(cr2) + GetGValue(cr2) + GetBValue(cr2);
if (intensity1 < intensity2)

return -1;

else if (intensity1 > intensity2)

return 1;

// 如果亮度相同，按颜色进行排序，(蓝色最前，红色最后)

if (GetBValue(cr1) > GetBValue(cr2))

return -1;

else if (GetGValue(cr1) > GetGValue(cr2))

return -1;

else if (GetRValue(cr1) > GetRValue(cr2))

return -1;

else

return 1;
}

```

上面的代码同时也说明了CompareItem成员函数的不同返回值所代表的不同含义。这里要注意的是，由于同亮度的不同颜色给人的眼睛的亮度感觉是不一样的(这好比人耳对声音的高频段和低频段的听觉灵敏度要比对中频段的听觉灵敏度要小一样)，上面的排序结果给人感



觉并不象我们所想象的那样，是通过颜色的亮度来进行的。但我们没有必要在这个问题上过分的纠缠而浪费时间。另外我们解释一下，为什么要使用`~cr & 0x00FFFFFF`来代替`~cr`。很多人会认为直接将颜色值按位取反就可以得到其对比色，但事实不是这样的。这是因为如果32位颜色值的高位字节不为零的话，该颜色值将不被当作一个RGB颜色值，而使用某个32位值与`0x00FFFFFF`按位与恰可以使其高位字节为零，而其它位则不变。

到目前为止我们完成了自绘制组合框对应的类`CClrComboBox`的设计，下面我们来看如何在程序中将类`CClrComboBox`的对象与对话框模板中的现存组合框相关联，这是使用函数`SubclassDlgItem`来实现的。首先在类`CComboDemoDlg`中添加一个类型为`CClrComboBox`的成员变量`m_clrCombo`，其访问限制在本工程中是不重要的，可以将它设置为`protected`。然后，在类`CComboDemoDlg`的`OnInitDialog`成员函数中的// TODO注释之后添加下面的一行代码，这行代码将对象`m_clrCombo`与ID为`IDC_CLRCOMBO`相关联，第一个参数为控件的父窗口的指针。

```
m_clrCombo.SubclassDlgItem(IDC_CLRCOMBO, this);
```

这样就可以通过`CWnd`的消息映射机制和消息传递路径在类`CClrComboBox`中处理`IDC_CLRCOMBO`中事件了。比如当组合框中的项需要重绘时，在`CClrComboBox`中定义的`DrawItem`成员函数将被调用，在正确的绘制组合框中的内容。

4. 这里，我们在初始时没有为组合框添加任何选择项，用户可以单击如图6.54的对话框中所示的“添加颜色”按钮向组合框的列表框中添加新颜色项。单击该按钮首先将弹出一个颜色选择对话框，用户如果从颜色选择对话框中选择了一种具体的颜色，该颜色将被添加到组合框的列表框中以供选择。基于这个要求，我们为按钮`IDC_ADDCLR`的`BN_CLICKED`事件添加如下的命令处理成员函数`OnAddClr`：

```
void CComboDemoDlg::OnAddClr()
{
    CColorDialog dlg(0, 0, this);

    int iRes=dlg.DoModal();

    if (iRes==IDOK)
    {
```

```

COLORREF cr=dlg.GetColor();

m_clrCombo.AddString( (LPCTSTR)cr );

}

else

{

}

}

```

虽然使用颜色对话框在本书的前面内容中没有讲述过，但即使是对初学者而言，上面的代码也是非常之简单的，我们这里就不过多的作讲解了。

下面来看如何为“改变颜色”按钮(IDC\_CHGCLR)添加单击命令处理成员函数。我们希望用户在单击该按钮时，改变静态文本控件IDC\_CLRSTATIC的颜色以反映用户所选择的颜色。如果改变静态文本控件的颜色呢？我们这里使用了将自绘制静态文本控件的办法。这并不是最简单的方法。最简单的方法是处理对话框的WM\_CTLCOLOR消息，该消息在控件将要被重绘前发送给该控件的父窗口。这里我们舍近而求远，主要是为了附带讲述一下自绘制静态文本控件的用法，它和自绘制组合框的用法存在一些区别。但是，在资源编辑器中我们不能设置一个静态文本控件的自绘制样式，不过这并不意味着将不能使用自绘制静态控件。方法并不复杂。首先，为静态文本控件添加自绘制样式，将下面的代码添加到类CComboDemoDlg的OnInitDialog成员函数中的// TODO注释之后：

```

GetDlgItem(IDC_CLRSTATIC)->ModifyStyle(0, SS_OWNERDRAW);

```

上面的代码将静态文本控件修改为具有SS\_OWNERDRAW样式的自绘制静态控件。下面我们来看如何在需要的时候重新绘制静态文本控件IDC\_CLRSTATIC，方法是在类CComboDemoDlg中为消息WM\_DRAWITEM添加处理函数OnDrawItem，而使用ClassWizard很容易办到这一点。重载版本的OnDrawItem成员函数的定义如下：

```

void CComboDemoDlg::OnDrawItem(int nIDCtl, LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDialog::OnDrawItem(nIDCtl, lpDrawItemStruct);

    if (nIDCtl==IDC_CLRSTATIC)

```

```

{
CDC *pDC=CDC::FromHandle(lpDrawItemStruct->hDC);

CBrush br(m_crClrStatic);

CRect rc=lpDrawItemStruct->rcItem;

pDC->FillRect(&rc, &br);

}

}

```

要使上面的代码正常工作，我们还需要在类CComboDemoDlg中定义类型为COLORREF的成员变量m\_crClrStatic，该成员变量保存了静态文本控件应该具有的颜色。我们注意到了在重载版本的OnDrawItem成员变量调用了基类的OnDrawItem成员函数，这是必要的，不要忘记在我们的对话框中还有一个自绘制组合框，基类的OnDrawItem成员函数调用自绘制组合框所对应的CComboBox的派生类的DrawItem成员函数来绘制组合框中的各个项。

下面来看下压按钮IDC\_CHGCLR的BN\_CLICKED事件的处理函数OnChgClr，其代码如下：

```

void CComboDemoDlg::OnChgClr()

{

int nSel=m_clrCombo.GetCurSel();

COLORREF cr=(COLORREF)m_clrCombo.GetItemData(nSel);

if (cr!=-1)

{

DRAWITEMSTRUCT drawItemStruct;

drawItemStruct.CtlID=IDC_CLRSTATIC;

drawItemStruct.hwndItem=GetDlgItem(IDC_CLRSTATIC)->GetSafeHwnd();

drawItemStruct.hDC=:GetDC(drawItemStruct.hwndItem);

GetDlgItem(IDC_CLRSTATIC)->GetClientRect(&(drawItemStruct.rcItem));

m_crClrStatic=cr;

```

```
OnDrawItem(IDC_CLRSTATIC, &drawItemStruct);

}

}
```

该成员函数将当前选中的颜色值(如果不为CB\_ERR的话)放入成员变量m\_crClrStatic中，然后构造一个DRAWITEMSTRUCT结构变量，并对它进行必要的初始化，最后调用该结构对象调用OnDrawItem成员函数重绘对话框。以后在需要重绘时，OnDrawItem成员函数会由框架自动调用。

这时即可编辑并运行上面的示例程序了，其运行结果如图6.56所示。单击“添加颜色”向组合框的列表框中添加几种颜色选项，再来调试程序的各项功能是否正常。还可以不同的窗口之前进行切换和相互覆盖或移开，以观察自绘制组合框和自绘制静态文本控件是否正确的绘制了自身。



图6. 56 示例程序ComboDemo的运行结果

相比较标准的组合框而言，自绘制组合框要复杂得多，我们得自己考虑很多特殊的问题，但是如示例程序所示，它的确可以实现一些很有趣的特性，因此在很多程序中得到广泛的使用。而掌握了自绘控件的使用，就可以使你所编写的应用程序界面更加的缤纷多彩，但是，要注意一切事物的使用都有一个“度”，不适宜的将应用程序的用户界面做得过分的“花哩呼哨”，很多时候只会适得其反。

## 第八节 滚动条控件



滚动条(如图6.57所示)本身也可以作为一种控件，通常我们使用这种控件来进行如定位之类的操作。滚动条控件分为水平滚动条和垂直滚动条两种，它们对应于Controls工具箱中的图标分别为和.



图6. 57 滚动条控件

对于滚动条控件，可以在Properties对话框的Styles选项卡内设置的

属性只有一种：即Align属性，该属性可以为三种值之一：None、Top/Left和Bottom/Right。其中，Top/Left表示将滚动条控件的左上边与由CreateWindowEx函数的参数定义的矩形的左上边对齐，而Bottom/Right则表示以右下边进行对齐。该属性的默认值为None，即不进行任何对齐操作。

Windows标准滚动条的行为由MFC类CScrollBar封装。表中列出了在类CScrollBar中定义的成员函数及其说明。

表6. 31 在类CScrollBar中定义的成员函数

成员函数	描述
CScrollBar	构造一个CScrollBar对象
Create	创建一个Windows滚动条，并将它与CScrollBar对象相关联
GetScrollPos	获得滚动条的当前位置
SetScrollPos	设置滚动条的当前位置
GetScrollRange	获得给定滚动条的当前最大和最小位置
SetScrollRange	设置给定滚动条的当前最大和最小位置
ShowScrollBar	显示或隐藏滚动条
EnableScrollBar	允许或禁止滚动条上的一个或两个箭头
SetScrollInfo	设置关于滚动条的信息
GetScrollInfo	获得滚动条的信息
GetScrollLimit	获得滚动条的限制

当用户单击了滚动条时，父窗口将收到WM\_HSCROLL或WM\_VSCROLL消息，在CWnd类的定义了处理该消息的成员函数为OnHScroll和OnVScroll。成员函数OnHScroll的原型如下：

```
afx_msg void OnHScroll( UINT nSBCode, UINT nPos, CScrollBar* pScrollBar );
```

第一个参数nSBCode指定如下之一的滚动条代码，这些代码代表用户所作的滚动请求：

SB\_LEFT :                      向左滚动较远距离

SB_ENDSCROLL :	结束滚动
SB_LINELEFT :	向左滚动
SB_LINERIGHT :	向右滚动
SB_PAGELEFT :	向左滚动一页
SB_PAGERIGHT :	向右滚动一页
SB_RIGHT :	向右滚动较远距离
SB_THUMBPOSITION :	滚动到绝对位置。当前位置由nPos参数指定
SB_THUMBTRACK :	拖动滚动条到指定的位置。当前位置由nPos参数指定

通常，SB\_THUMBTRACK滚动条代码由应用程序使用，以便在滚动条被拖动时给以反馈。如果应用程序滚动了由滚动条控制的内容，它必须使用SetScrollPos来重置滚动条的位置。

传递给函数OnHScroll的参数反映了当收到消息时由框架获得的值，如果在重载版本的函数中调用了基类的实现，该实现将使用最初由消息传递的参数，而不是向函数提供的参数。

消息WM\_VSCROLL的处理函数OnVScroll与OnHScroll类似，我们这里就不再重复讲述了。下面我们来看一个例子：

1. 创建一个名为ScrollDemo的基于对话框的MFC工程，按图设置对话框的各控件。其中水平滚动条控件的ID为IDC\_SCROLL，编辑框控件的ID为IDC\_CURPOS。



图6. 58 示例程序ScrollDemo的主对话框的设计

2. 使用ClassWizard为编辑框控件IDC\_CURPOS映射类型为int的成员变量m\_iCurPos，并设置其最大值为100，最小值为-100。
3. 使用ClassWizard在类CScrollDemoDlg中为消息WM\_HSCROLL添加处理函数OnHScroll，其代码如下：

```
void CScrollDemoDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // 获得原有的滚动条位置

    int iPos=pScrollBar->GetScrollPos();

    // 根据不同的拖动方式设置新的滚动条位置

    switch (nSBCode)
    {
        // 向右滚动一行

        case SB_LINERIGHT:

            iPos+=1;

            break;

        // 向左滚动一行

        case SB_LINELEFT:

            iPos-=1;

            break;

        // 向右滚动一页

        case SB_PAGERIGHT:

            iPos+=10;

            break;

        // 向左滚动一页

        case SB_PAGELEFT:

            iPos-=10;

            break;

        // 直接拖动滚动块

        case SB_THUMBTRACK:

            iPos=nPos;
```

```

break;

default:

break;

}

// 滚动条的最大位置不超过 100, 最小位置不小于 -100

if (iPos<-100) iPos=-100;

if (iPos>100) iPos=100;

// 必须手动的更新滚动条的当前位置

pScrollBar->SetScrollPos(iPos);

// 在编辑框中显示滚动条的当前位置

SetDlgItemInt(IDC_CURPOS, iPos);

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}

```

上面的代码暗示了一点，这就是在被拖动时，滚动条不会自动更新其位置，我们必须自己在程序中做到这一点，即通过分析不同的滚动方式来改变并设置新的滚动条位置，上面的代码演示了这一过程。

编译上面的程序代码，我们发现滚动条不能正常工作！这是因为在默认情况下，滚动条的滚动范围为从0到0。这时，我们根本不可能对滚动条进行有意义的操作。因此，我们需要将下面的代码添加到OnInitDialog成员函数：

```

CScrollBar *pScroll=(CScrollBar*)GetDlgItem(IDC_SCROLL);

pScroll->SetScrollRange(-100, 100);

pScroll->SetScrollPos(0);

SetDlgItemInt(IDC_CURPOS, 0);

```

上面的代码设定了滚动条的滚动范围和默认的滚动条位置，然后，将当前滚动条位置显示在编辑控件IDC\_CURPOS中。

4. 最后我们来实现一个功能，这就是我们希望当编辑控件中的文本发生改变时，滚动条上的滑块的位置也相应的变化。要实现这一点，



使用ClassWizard为控件IDC\_CURPOS的通知消息EN\_CHANGE添加消息处理函数OnChangeCurPos。

```
void CScrollDemoDlg::OnChangeCurPos()
{
    CString str;

    GetDlgItemText(IDC_CURPOS, str);

    str.TrimLeft();
    str.TrimRight();

    int iPos=0;

    if (str!="-" && str!="")
    {
        if (!UpdateData())
        {
            return;
        }

        iPos=m_iCurPos;
    }

    CScrollBar *pScroll=(CScrollBar*)GetDlgItem(IDC_SCROLL);

    pScroll->SetScrollPos(iPos);
}
```

由于需要检验用户输入数据的有效性，上面的代码比较长。首先，如果用户只输入一个负号“？”或刚将原有的数据删除，此时不应该报错。这里我们可以将滚动条的位置设置为0。由于用户可能在所输入的数据之前或之后插入一些空格，这种情况下我们也不应该报错，因此，我们使用了一些额外的代码来避免了这种情况。最后，我们使用了UpdateData函数来使用控件IDC\_CURPOS的值更新成员变量m\_iCurPos，这样的目的是便于使用MFC提供的对话框数据检验机制。但有个不好的地方是，如果用户输入的数据有错，出现的报错消息是英文的。如果我们需要的是一个完全中文化的软件，这不能不算是一

个瑕疵，这时，我们应该编写自己的数据检验代码。但是在本示例程序中，并不需要这样要求，这里使用MFC的对话框数据检验机制是很有效的。回到程序代码中去，如果用户在编辑控件中输入的值有效的话，使用这个值去更新滚动条的当前位置，这是通过类CScrollbar的成员函数SetScrollPos来实现的。

其它的一些控件，如CSliderCtrl类所封装的滑块控件等，与滚动条控件的使用有很大的共通之处，读者完全可以根据本章中所讲述的内容通过举一反三来用于其它的情况。

- 注意：

- 由于篇幅有限，在本章中我们不打算介绍更多的Windows控件。事实上，Windows控件的使用是有规律可循的。只需要弄清楚几种控件的用法，以及MFC在处理控件时的机制，就很容易借助Visual C++所提供的丰富的联机文档来学习其它控件的使用。本章中所介绍的控件，还只是所有控件中很小的一个部分，而且，即使是对所介绍的几种控件的讲述也不是面面俱到的。我们的目的不再于详尽的罗列各种控件的使用方法，而在于起到一种“抛砖引玉”的作用。

## 第七章 使用ActiveX控件

Windows本身已经提供了很多的控件，我们已经在本书前面的章节中对这些控件作了一些介绍。但是，应用程序用户的需求是各种各样的，而且，程序员本身的创造力也不应该因此而受到制约。然而，经历过的人都会深深的体会到，仅仅凭借自己的力量，要想完成一个完善的功能强大的应用程序并非易事——不是不可以，只是非常之的艰难。绝大多数的优秀的应用程序，都凝结了很多优秀的程序员的天才的创造力和辛勤的劳动。因此，在很多时候，我们不得不对自己能不能够使用Visual C++写出一个既有强大的实用功能，又具有美观的用户界面的应用程序表示怀疑。很不幸，坦率的说，要自己从底层写起，一个这样的应用程序常常会埋葬无数的时间。但是，当我们从一个更大的范围来观察这一情况的时候，我们发现，事实上，就很多编程课题而言，无数的程序员在做的仅仅是一些重复的劳动。大量优秀的人才浪费在为同一个目的编写功能相同的软件上。这启发了我们，如果能够制定一套规则，程序员们在此规则的基础上开发各种各样的功能组件，这些功能组件可以方便的用于多个应用程序。ActiveX技术就是这样的一种技术。基于ActiveX技术的为数众多的软件组件都提供了满足某个规范的一系列编程接口，应用程序可以通过该编程接口使用由软件组件提供的各种功能，而无需知道这些功能在具体的软件组件里是如何实现的。这种方式也是我们早在本书一开始的时候所提到的面向对象的程序设计的一个主要特点。

ActiveX技术本身是一种非常复杂的技术，尽管有很多的书藉在讲述这一内容时故意淡化这一点。要想通过很短的篇幅以很通俗的语言阐明这个技术本身几乎是一件不可能的事。从本书的写作意图和篇幅来考虑，我们不打算深入的讨论ActiveX本身和如何创建基于ActiveX技术的各种软件组件，而把关心的焦点放到如何应用ActiveX控件本身上。

本章的焦点放到两个问题上：

- 什么是ActiveX控件
- 如何在应用程序中使用ActiveX控件

### 第一节 什么是ActiveX控件

什么是ActiveX控件？这个问题本身也并不容易说得清楚。在下面的内容中，我们将侧重的于控件使用者，而不是控件开发者的角度来说明这个问题。

ActiveX控件过去被称作OLE控件，其开发基于通常对象模型(Common Object Model, COM)，它可以嵌入对话框或其它的ActiveX控件容器，如Internet Explorer和Visual Basic应用程序中使用。ActiveX控件取代了过去的16位的Visual Basic控件(VBX)。

更专业一点说，ActiveX控件是这样的一个基于COM的对象：它可以自己的窗口内绘制自身，可以用户的如单击鼠标或按下键盘之类的操作事件，此外是最重要的一点，使用ActiveX控件的应用程序可以通过该控件所包括的一系列的属性和方法(合起来称作接口)来操作该控件的行为。

不要为ActiveX控件本身这个名词所误导，ActiveX控件并不仅仅限于与用户的可视交互，它还可以用于其它用途，如访问数据库、监视数据等。ActiveX控件所能提供的功能要远远超过自定义控件的能力。这些能力包括很多新鲜而有趣的特性，比如将控件本身所提供的菜单嵌入到容器的菜单中等。

ActiveX控件一般以.OCX文件的形式提供，并在系统中进行注册。

对于ActiveX控件这一复杂的课题，我们现在只打算给一个最简单的概念。下一步我们将通过一个示例来说明ActiveX控件在编程中的使用。事实上，使用实例来说话往往对初学者往往要更具效果。

## 第二节 使用ActiveXMovie控件的视频播放器

在下面的过程中，我们将通过使用ActiveMovie控件创建一个视频播放器，该播放器支持多种文件格式。按下面的步骤来创建示例程序VideoPlayer：

1. 使用AppWizard创建一个基于对话框的MFC工程。所有步骤均使用AppWizard给出的默认设置。在这种情况下，应用程序自动提供了对ActiveX控件的支持。紧接着删除应用应用程序主对话框中的所有控件，包括“确定”和“取消”按钮。
2. 在主对话框中单击鼠标右键，选择Insert ActiveX Control命令。随后弹出如图7.1所示的对话框。

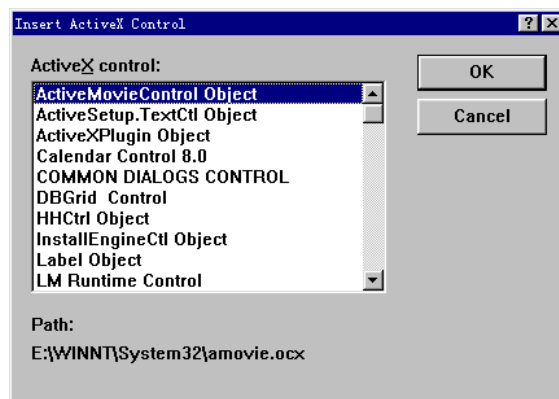


图7. 1 向对话框中插入ActiveX控件

3. 在图7.1的列表框中选择ActiveMovieControl Object，并单击OK按钮。

- 注意：
- 如果在你在如图7.1所示的列表框中找不到项ActiveMovieControl，则说明你的计算机系统中没有安装ActiveMovie控件，或者ActiveMovie控件没有在你的系统中进行正确的注册。这时，你需要安装ActiveMovie控件才可以继续创建示例程序VideoPlayer。可以有多个途径得到ActiveMovie控件。你可以从Internet Explorer 4.0软件包中获得该控件，也可以从Microsoft获得该控件的单独发布版本。

4. 这时，回到应用程序VideoPlayer的主对话框，右击新添加的控件，选择ClassWizard为新添加的控件映射一个成员变量(下面的步骤将说明这个成员变量的类型)。

5. 在添加成员变量映射的过程中，ClassWizard将会弹出如图7.2所示的对话框，以询问是否为ActiveMovieControl Object创建一个类以封装对该控件提供的接口的调用。在该对话框中单击确定。



图7. 2 询问是否创建一个C++类以封装ActiveMovieControl对象

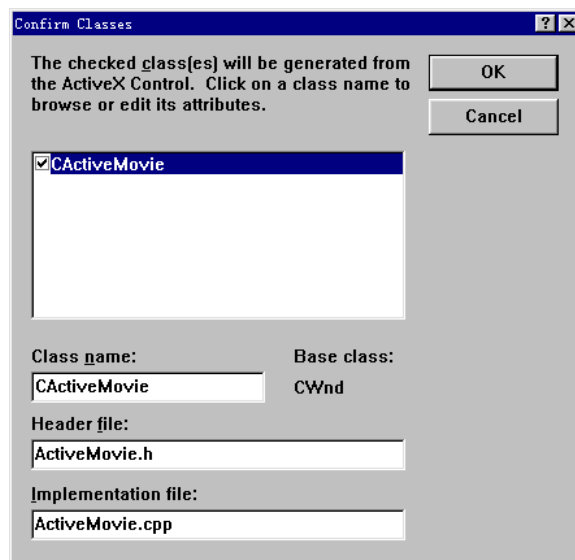



图7. 3 为ActiveMovie控件创建新类CActiveMovie

在如图7.3所示的对话框中为ActiveMovieControl创建新的类CActiveMovie。

上面的步骤也可以使用其它的途径来完成。首先，在Project菜单的Add To Project子菜单下选择Components and Controls命令。该命令打开如图7.4所示的对话框，在该对话框中双击Registered ActiveX Controls，并从中选择ActiveMovieControl Object。单击Insert按钮，也弹出如图7.3所示的对话框。按第五步中所讲述的内容完成类CActiveMovie。这时，在Controls工具箱中将会多一个按钮，单击该按钮，即可像添加一般的标准控件那样添加ActiveMovie控件。

相比较而言，后一种方法要更为直观一些。但两种方法的最后结果都是一致的，具体到每一个编程者来说，选择哪一种完成是任意的。但很明显，如果需要一次向对话本事添加多于一个的同一ActiveX控件，使用后面的方法要省事得多。

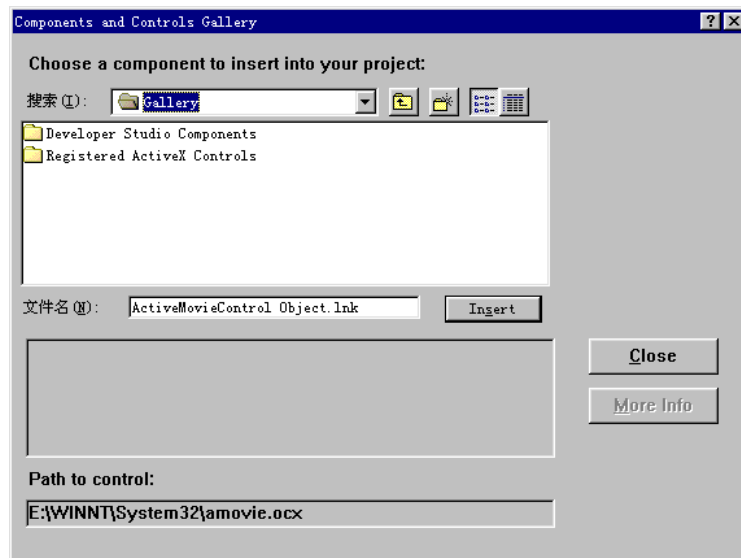


图7. 4 从Components and Controls Gallery中向工程添加ActiveX控件

在完成这一步骤之后，为新添加的ActiveMovie控件(我们设置其ID为IDC\_AMOVIE)映射类型为CActiveMovie的成员变量m\_amovie。

6. 现在我们来大致的浏览一下类CActiveMovie的定义。该定义保存在头文件ActiveMovie.h中。

```
#if !defined(AFX_ACTIVEMOVIE_H__9B0F9FA0_1F04_11D2_9717_0000B4810A31__INCLUDED_)
#define AFX_ACTIVEMOVIE_H__9B0F9FA0_1F04_11D2_9717_0000B4810A31__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

// 注意：不要修改你在这里看到的文件的内容。如果该类是由 Visual C++ 所创建的话，
// 您在这里所作的修改将会被覆盖

////////////////////////////////////

// CActiveMovie 封装类

class CActiveMovie : public CWnd
{
protected:
DECLARE_DYNCREATE(CActiveMovie)

public:
```

```

CLSID const& GetClsid()
{
static CLSID const clsid
= { 0x5589fa1, 0xc356, 0x11ce, { 0xbf, 0x1, 0x0, 0xaa, 0x0, 0x55, 0x59, 0x5a } };
return clsid;
}

virtual BOOL Create(LPCTSTR IpszClassName,
LPCTSTR IpszWindowName, DWORD dwStyle,
const RECT& rect,
CWnd* pParentWnd, UINT nID,
CCreateContext* pContext = NULL)
{ return CreateControl(GetClsid(), IpszWindowName, dwStyle, rect, pParentWnd,
nID); }

BOOL Create(LPCTSTR IpszWindowName, DWORD dwStyle,
const RECT& rect, CWnd* pParentWnd, UINT nID,
CFile* pPersist = NULL, BOOL bStorage = FALSE,
BSTR bstrLicKey = NULL)
{ return CreateControl(GetClsid(), IpszWindowName, dwStyle, rect, pParentWnd,
nID,
pPersist, bStorage, bstrLicKey); }

// Attributes

public:

// Operations

public:

void AboutBox();

void Run();

void Pause();

```



```
void Stop();

long GetImageSourceWidth();

long GetImageSourceHeight();

CString GetAuthor();

CString GetTitle();

CString GetCopyright();

CString GetDescription();

CString GetRating();

CString GetFileName();

void SetFileName(LPCTSTR lpszNewValue);

double GetDuration();

double GetCurrentPosition();

void SetCurrentPosition(double newValue);

long GetPlayCount();

void SetPlayCount(long nNewValue);

double GetSelectionStart();

void SetSelectionStart(double newValue);

double GetSelectionEnd();

void SetSelectionEnd(double newValue);

long GetCurrentState();

double GetRate();

void SetRate(double newValue);

long GetVolume();

void SetVolume(long nNewValue);

long GetBalance();

void SetBalance(long nNewValue);
```

```
BOOL GetEnableContextMenu();  
void SetEnableContextMenu(BOOL bNewValue);  
BOOL GetShowDisplay();  
void SetShowDisplay(BOOL bNewValue);  
BOOL GetShowControls();  
void SetShowControls(BOOL bNewValue);  
BOOL GetShowPositionControls();  
void SetShowPositionControls(BOOL bNewValue);  
BOOL GetShowSelectionControls();  
void SetShowSelectionControls(BOOL bNewValue);  
BOOL GetShowTracker();  
void SetShowTracker(BOOL bNewValue);  
BOOL GetEnablePositionControls();  
void SetEnablePositionControls(BOOL bNewValue);  
BOOL GetEnableSelectionControls();  
void SetEnableSelectionControls(BOOL bNewValue);  
BOOL GetEnableTracker();  
void SetEnableTracker(BOOL bNewValue);  
BOOL GetAllowHideDisplay();  
void SetAllowHideDisplay(BOOL bNewValue);  
BOOL GetAllowHideControls();  
void SetAllowHideControls(BOOL bNewValue);  
long GetDisplayMode();  
void SetDisplayMode(long nNewValue);  
BOOL GetAllowChangeDisplayMode();  
void SetAllowChangeDisplayMode(BOOL bNewValue);
```

```
LPUNKNOWN GetFilterGraph();

void SetFilterGraph(LPUNKNOWN newValue);

LPDISPATCH GetFilterGraphDispatch();

unsigned long GetDisplayForeColor();

void SetDisplayForeColor(unsigned long newValue);

unsigned long GetDisplayBackColor();

void SetDisplayBackColor(unsigned long newValue);

long GetMovieWindowSize();

void SetMovieWindowSize(long nNewValue);

BOOL GetFullScreenMode();

void SetFullScreenMode(BOOL bNewValue);

BOOL GetAutoStart();

void SetAutoStart(BOOL bNewValue);

BOOL GetAutoRewind();

void SetAutoRewind(BOOL bNewValue);

long GetHWnd();

long GetAppearance();

void SetAppearance(long nNewValue);

long GetBorderStyle();

void SetBorderStyle(long nNewValue);

BOOL GetEnabled();

void SetEnabled(BOOL bNewValue);

BOOL IsSoundCardEnabled();

long GetReadyState();

};

//{{AFX_INSERT_LOCATION}}
```

```
#endif // !defined
(AFX_ACTIVEMOVIE_H__9B0F9FA0_1F04_11D2_9717_0000B4810A31__INCLUDED_)
```

尽管我们可以由此得知由该ActiveX控件所提供的各个接口属性和方法的参数和返回值，但是，这些信息并不足以正确的使用该ActiveX控件。一般来说，由第三方开发商提供的ActiveX控件都附带了对所提供的控件的各个接口属性和方法的说明及其与使用该控件进行程序设计所需的信息。

## 7. 按表创建应用程序的菜单资源IDR\_MENU。

接着使用属性对话框将应用程序主对话框的菜单资源设置为IDR\_MENU。

8. 设计用于音量调节的对话框，该对话框如图7.5所示，其资源ID为IDD\_VOLUME。设计完成之后使用ClassWizard为该对话框创建新的类CVolumeDlg，并为滑块控件映射类型为CSliderCtrl的成员变量m\_slid。

根据下面的代码在完成类CVolumeDlg，该类提供了外部编程接口SetVolume，该公有成员函数使用一个指向CActiveMovie对象的指针作为其参数，所进行的音量调节作用于该控件。

这里需要注意的是，拖动或点击滑块控件时，向父窗口发送的消息是WM\_HSCROLL，该消息的处理函数OnHScroll，传递给该处理函数的第二个参数的类型为CScrollBar\*，我们需要使用强调类型转换将其转换为CSliderCtrl\*，以便能正确的调用由CSliderCtrl对象所提供的各种成员函数。

表7. 1 应用程序VideoPlayer使用的菜单资源IDR\_MENU

顶层菜单项	子菜单项	资源ID
文件(&F)	打开(&O)	ID_FILEOPEN
	关闭(&C)	ID_FILECLOSE
	具有Separator样式的菜单分隔符	
	退出(&X)	ID_FILEEXIT
	开始(&S)	ID_PLAYSTART

播放(&P)	暂停(&P)	ID_PLAYPAUSE
	停止(&T)	ID_PLAYSTOP
视频(&V)	原始大小(&O)	ID_VIDEO1X
	原始大小的2倍(&T)	ID_VIDEO2X
音频(&A)	调节音量(&V)	ID_AUDEOVOLUME
	调节左右声道平衡(&B)	ID_AUDEOBALANCE
帮助(&H)	关于 视频播放器(&A)	ID_HELPABOUT

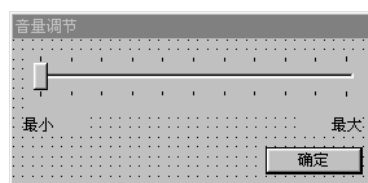


图7. 5 用于调节音量的对话框：IDD\_VOLUME

以下是类CVolumeDlg的代码清单。为了节省篇幅和便于读者阅读和理解，我们删除了一些由AppWizard生成的代码和注释。

```
// VolumeDlg.h : 头文件
//

class CActiveMovie;

////////////////////////////////////

// CVolumeDlg 对话框

class CVolumeDlg : public CDialog
{
// 构造

public:

void SetVolume(CActiveMovie* pAmovie);

CVolumeDlg(CWnd* pParent = NULL); // 标准构造函数

//{{AFX_DATA(CVolumeDlg)
```



```

CVolumeDlg::CVolumeDlg(CWnd* pParent /*=NULL*/)
: CDialog(CVolumeDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CVolumeDlg)
// 注意：ClassWizard 将在这里添加对成员的初始化代码
//}}AFX_DATA_INIT
}

void CVolumeDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CVolumeDlg)
DDX_Control(pDX, IDC_SLIDER1, m_sld);
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CVolumeDlg, CDialog)
//{{AFX_MSG_MAP(CVolumeDlg)
ON_WM_HSCROLL()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////

// CVolumeDlg 消息处理函数

BOOL CVolumeDlg::OnInitDialog()
{
CDialog::OnInitDialog();

// 设置音量滑块的最小值和最大值、标度和当前位置等
m_sld.SetRange(-10000, 0);

```

```

m_sld.SetTicFreq(1000);

m_sld.SetLineSize(200);

m_sld.SetPageSize(1000);

// 以 ActiveMovie 控件的当前音量作为音量滑块的当前位置

m_sld.SetPos(m_pAmovie->GetVolume());

return TRUE;

}

// 提供给类外部的使用者的编程接口方法

void CVolumeDlg::SetVolume(CActiveMovie * pAmovie)

{

m_pAmovie=pAmovie;

// 如果未加载任何媒体文件，则音量调节不可用。ActiveMovie 控件的 CurrentState 属
// 性返回控件的当前状态，-1 表示未加载任何文件，此时弹出出错提示信息

if (m_pAmovie->GetCurrentState() != -1)

{

DoModal();

}

else

{

MessageBox("音频设备尚未加载，请先打开一个媒体文件。");

}

}

// 在用户拖动或点击滑块控件时，将所作的改变立即作用于 ActiveMovie 控件

void CVolumeDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)

{

CSliderCtrl *pSlider=(CSliderCtrl *)pScrollBar;

```



```
int nVolume=pSlider->GetPos();

m_pAmovie->SetVolume(nVolume);

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}
```

8. 设计用于调节左右声道平衡的对话框。该对话框如图7.6所示，其资源ID为IDD\_BALANCE。由ClassWizard创建的新类CBalanceDlg封装了该对话框。同前面的对话框IDD\_VOLUME相类似，CBalanceDlg定义了公有成员函数SetBalance以供类外部的使用者调用，该函数同样使用一个指向CActiveMovie对象的指针作为其参数。



图7. 6 用于调节左右声道平衡的对话框：  
IDD\_BALANCE

根据下面提供的代码来完成类CBalanceDlg。

```
// BalanceDlg.h : 头文件

//

class CActiveMovie;

////////////////////////////////////

// CBalanceDlg 对话框

class CBalanceDlg : public CDialog

{

// 构造

public:

void SetBalance(CActiveMovie* pAmovie);

CBalanceDlg(CWnd* pParent = NULL); // 标准构造函数

// 对话框数据

//{{AFX_DATA(CBalanceDlg)
```

[illegible]

```

CBalanceDlg::CBalanceDlg(CWnd* pParent /*=NULL*/)
: CDialog(CBalanceDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CBalanceDlg)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
}

void CBalanceDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CBalanceDlg)
DDX_Control(pDX, IDC_SLIDER1, m_sld);
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CBalanceDlg, CDialog)
//{{AFX_MSG_MAP(CBalanceDlg)
ON_WM_HSCROLL()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////

// CBalanceDlg 消息处理函数

BOOL CBalanceDlg::OnInitDialog()
{

```

```

CDialog::OnInitDialog();

// 初始化音量调节滑块的最大值和最小值、标度以及当前位置等
m_sld.SetRange(-10000, 10000);

m_sld.SetTicFreq(2000);

m_sld.SetLineSize(500);

m_sld.SetPageSize(2000);

m_sld.SetPos(m_pAmovie->GetBalance());

return TRUE;

}

// 提供给类外部的使用者的接口方法

void CBalanceDlg::SetBalance(CActiveMovie * pAmovie)

{

m_pAmovie=pAmovie;

// 如果当前 ActiveMovie 未加载任何文件，则音量调节不可用并弹出出错信息

// ActiveMovie 控件的 CurrentState 属性返回了当前控件的状态，-1 表示未加载任何文件

if (m_pAmovie->GetCurrentState() != -1)

{

DoModal();

}

else

{

MessageBox("音频设备尚未加载，请先打开一个媒体文件。");

}

}

// 当用户在拖动或点击滑块时，将用户的改变立即作用于 ActiveMovie 控件

void CBalanceDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)

```

```

{
CSliderCtrl *pSlider=(CSliderCtrl *)pScrollBar;

// 得到滑块控件的当前位置

int nVolume=pSlider->GetPos();

// 设置 ActiveMovie 控件的当前的声道平衡设置

m_pAmovie->SetBalance(nVolume);

CDialog::OnHScroll(nSBCode, nPos, pScrollBar);

}

```

9. 按下面所给的代码来修改类CVideoPlayerDlg，在注意的是，这里的大部分菜单处理的声明是使用ClassWizard添加的：

```

// VideoPlayerDlg.h : 头文件

//

//{{AFX_INCLUDES()
#include "activemovie.h"
//}}AFX_INCLUDES

////////////////////////////////////

// CVideoPlayerDlg 对话框

class CVideoPlayerDlg : public CDialog
{
// 构造

public:

CVideoPlayerDlg(CWnd* pParent = NULL); // 标准构造函数

// 对话框数据

//{{AFX_DATA(CVideoPlayerDlg)
enum { IDD = IDD_VIDEOPLAYER_DIALOG };

CActiveMovie m_amovie;

```

```

//}}AFX_DATA

//{{AFX_VIRTUAL(CVideoPlayerDlg)

protected:

virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV 支持

//}}AFX_VIRTUAL

// 实现

protected:

void MoveMovieWindow();

HICON m_hIcon;

//{{AFX_MSG(CVideoPlayerDlg)

virtual BOOL OnInitDialog();

afx_msg void OnSysCommand(UINT nID, LPARAM lParam);

afx_msg void OnPaint();

afx_msg HCURSOR OnQueryDragIcon();

afx_msg void OnFileOpen();

afx_msg void OnOpenCompleteAmovie();

afx_msg void OnReadyStateChangeAmovie(long ReadyState);

afx_msg void OnFileClose();

afx_msg void OnStateChangeAmovie(long oldState, long newState);

afx_msg void OnDisplayModeChangeAmovie();

afx_msg void OnFileExit();

afx_msg void OnPlayStart();

afx_msg void OnPlayStop();

afx_msg void OnAudioVolume();

afx_msg void OnErrorAmovie(short SCode, LPCTSTR Description,
LPCTSTR Source, BOOL FAR* CancelDisplay);

```

```

afx_msg void OnAudeoBalance();

afx_msg void OnVideo1x();

afx_msg void OnVideo2x();

afx_msg void OnPlayPause();

afx_msg void OnHelpAbout();

DECLARE_EVENTSINK_MAP()

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

// VideoPlayerDlg.cpp : 实现文件

//

#include "stdafx.h"

#include "VideoPlayer.h"

#include "VideoPlayerDlg.h"

#include "VolumnDlg.h"

#include "BalanceDlg.h"

// 在此省略了定义和实现类 CAboutDlg 的代码

// ...

////////////////////////////////////

// CVideoPlayerDlg 对话框

CVideoPlayerDlg::CVideoPlayerDlg(CWnd* pParent /*=NULL*/)
: CDialog(CVideoPlayerDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CVideoPlayerDlg)

//}}AFX_DATA_INIT

// 注意：LoadIcon 函数并不需要的后面调用 Win32 中的 DestroyIcon

```

```

m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

}

void CVideoPlayerDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CVideoPlayerDlg)
    DDX_Control(pDX, IDC_AMOVIE, m_amovie);
    //}}AFX_DATA_MAP
}

// 类 CVideoPlayerDlg 的消息映射
BEGIN_MESSAGE_MAP(CVideoPlayerDlg, CDialog)
    //{{AFX_MSG_MAP(CVideoPlayerDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_COMMAND(ID_FILEOPEN, OnFileOpen)
    ON_COMMAND(ID_FILECLOSE, OnFileClose)
    ON_COMMAND(ID_FILEEXIT, OnFileExit)
    ON_COMMAND(ID_PLAYSTART, OnPlayStart)
    ON_COMMAND(ID_PLAYSTOP, OnPlayStop)
    ON_COMMAND(ID_AUDEOVOLUME, OnAudeoVolume)
    ON_COMMAND(ID_AUDEOBALANCE, OnAudeoBalance)
    ON_COMMAND(ID_VIDEO1X, OnVideo1x)
    ON_COMMAND(ID_VIDEO2X, OnVideo2x)
    ON_COMMAND(ID_PLAYPAUSE, OnPlayPause)
    ON_COMMAND(ID_HELPABOUT, OnHelpAbout)

```



```

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

////////////////////////////////////

// CVideoPlayerDlg 消息处理函数

BOOL CVideoPlayerDlg::OnInitDialog()

{
CDialog::OnInitDialog();

// 为对话框设置图标。对于不使用对话框作为主窗口的应用程序，这一步骤由框架自动完成

SetIcon(m_hIcon, TRUE); // 设置大图标

SetIcon(m_hIcon, FALSE); // 设置小图标

// 设置 ActiveMovie 控件的初始文件为空

m_amovie.SetFileName("");

// 改变窗口的大小以适应 ActiveMovie 控件的大小

MoveMovieWindow();

return TRUE;

}

// 如果向对话框中添加到最小化按钮，您将需要编写代码来绘制相应的图标。对于使用了
// 文档/视结构的应用程序，这一操作由框架自动完成

void CVideoPlayerDlg::OnPaint()

{

if (IsIconic())

{

CPaintDC dc(this); // 绘制设备上下文

SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

// 在客户区矩形内居中图标

int cxIcon = GetSystemMetrics(SM_CXICON);

```

```

int cylcon = GetSystemMetrics(SM_CYICON);

CRect rect;

GetClientRect(&rect);

int x = (rect.Width() - cxlcon + 1) / 2;

int y = (rect.Height() - cylcon + 1) / 2;

// 绘制图标

dc.DrawIcon(x, y, m_hlcon);

}

else

{

CDialog::OnPaint();

}

}

// 当用户拖动最小化了的窗口时，系统调用该函数来获得显示使用的光标

HCURSOR CVideoPlayerDlg::OnQueryDragIcon()

{

return (HCURSOR) m_hlcon;

}

// 打开媒体文件

void CVideoPlayerDlg::OnFileOpen()

{

CFileDialog dlg(TRUE, NULL, NULL, 0, "所有文件|*.*", NULL);

if (dlg.DoModal()==IDOK)

{

m_amovie.SetFileName(dlg.GetPathName());

}

}

```

```

}

// ActiveMovie 控件 IDC_AMOVIE 的消息映射
BEGIN_EVENTSINK_MAP(CVideoPlayerDlg, CDialog)

//{{AFX_EVENTSINK_MAP(CVideoPlayerDlg)
ON_EVENT(CVideoPlayerDlg, IDC_AMOVIE, 50 /* OpenComplete */,
OnOpenCompleteAmovie, VTS_NONE)
ON_EVENT(CVideoPlayerDlg, IDC_AMOVIE, -609 /* ReadyStateChange */,
OnReadyStateChangeAmovie, VTS_I4)
ON_EVENT(CVideoPlayerDlg, IDC_AMOVIE, 1 /* StateChange */,
OnStateChangeAmovie, VTS_I4 VTS_I4)
ON_EVENT(CVideoPlayerDlg, IDC_AMOVIE, 51 /* DisplayModeChange */,
OnDisplayModeChangeAmovie, VTS_NONE)
ON_EVENT(CVideoPlayerDlg, IDC_AMOVIE, 999 /* Error */,
OnErrorAmovie, VTS_I2 VTS_BSTR VTS_BSTR VTS_PBOOL)
//}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

// 当 ActiveMovie 控件完成媒体文件的加载时自动播放该文件
void CVideoPlayerDlg::OnOpenCompleteAmovie()
{
    m_amovie.Run();
}

// 当 ActiveMovie 控件的准备状态发生改变时，在需要的情况下改变对话框的大小以适应
// ActiveMovie 控件的大小
void CVideoPlayerDlg::OnReadyStateChangeAmovie(long ReadyState)
{
    MoveMovieWindow();
}

```

```

}

// 关闭媒体文件，即简单的将 ActiveMovie 控件的 FileName 属性值设置为空

void CVideoPlayerDlg::OnFileClose()

{
    m_amovie.SetFileName("");
}

// 当 ActiveMovie 控件的状态发生改变时，在需要的情况下改变对话框的大小以适应
// ActiveMovie 控件的大小

void CVideoPlayerDlg::OnStateChangeAmovie(long oldState, long newState)

{
    MoveMovieWindow();
}

// 改变对话框在大小以适应 ActiveMovie 控件的大小

void CVideoPlayerDlg::MoveMovieWindow()

{
    CRect rc1, rc2, rc3;

    // 得到 ActiveMovie 控件的大小

    m_amovie.GetWindowRect(rc1);

    // 保证对话框客户区的宽不小于 300 像素，高不小于 225 像素

    if (rc1.Width()<300 || rc1.Height()<225)

    {
        rc1.right=rc1.left+300;

        rc1.bottom=rc1.top+225;
    }

    // 获得对话框的大小

    GetWindowRect(rc2);

```

```

// 获得对话框客户区的大小

GetClientRect(rc3);

// 改变对话框的大小以适应 ActiveMovie 控件的大小

MoveWindow(rc2.left, rc2.top,

rc2.Width()-rc3.Width()+rc1.Width(),

rc2.Height()-rc3.Height()+rc1.Height());

// 获得 ActiveMovie 控件的大小

m_amovie.GetWindowRect(rc1);

GetClientRect(rc3);

// 使 ActiveMovie 控件在对话框的客户区居中

m_amovie.MoveWindow((rc3.Width()-rc1.Width())/2,

(rc3.Height()-rc1.Height())/2, rc1.Width(), rc1.Height());

}

// 当 ActiveMovie 控件的显示模式发生改变时，改变对话框的大小以适应这个改变

void CVideoPlayerDlg::OnDisplayModeChangeAmovie()

{

MoveMovieWindow();

}

// 关闭应用程序

void CVideoPlayerDlg::OnFileExit()

{

OnCancel();

}

// 开始播放所选定的媒体文件，由 ActiveMovie 控件的 Run 方法来实现

void CVideoPlayerDlg::OnPlayStart()

{

```

```

m_amovie.Run();

}

// 停止播放正在播放的媒体文件，由 ActiveMovie 控件的 Stop 方法来实现

void CVideoPlayerDlg::OnPlayStop()

{

m_amovie.Stop();

}

// 调节音量

void CVideoPlayerDlg::OnAudeoVolumn()

{

CVolumeDlg dlgVolumn;

dlgVolumn.SetVolumn(&m_amovie);

}

// 处理 ActiveMovie 的出错事件，这里我们只是简单的弹出一个消息以告诉用户

// 出错的代码和描述

void CVideoPlayerDlg::OnErrorAmovie(short SCode, LPCTSTR Description,

LPCTSTR Source, BOOL FAR* CancelDisplay)

{

CString str;

str.Format("出现错误[%d]：\n\n%s", SCode, Description);

MessageBox(str);

*CancelDisplay=TRUE;

}

// 调节左右声道的平衡

void CVideoPlayerDlg::OnAudeoBalance()

{

```

```
CBalanceDlg dlgBalance;

dlgBalance.SetBalance(&m_amovie);

}

// 设置视频窗口为视频文件的原始大小
void CVideoPlayerDlg::OnVideo1x()

{
m_amovie.Pause();
m_amovie.SetMovieWindowSize(0);
m_amovie.Run();
}

// 设置视频窗口为视频文件的两倍大小
void CVideoPlayerDlg::OnVideo2x()

{
m_amovie.Pause();
m_amovie.SetMovieWindowSize(1);
m_amovie.Run();
}

// 暂停媒体文件的播放，由 ActiveMovie 控件的 Pause 方法来实现
void CVideoPlayerDlg::OnPlayPause()

{
m_amovie.Pause();
}

// 显示关于对话框
void CVideoPlayerDlg::OnHelpAbout()

{
```

```
CAboutDlg dlgAbout;  
  
dlgAbout.DoModal();  
  
}
```

9. 最后补充一点，在应用程序VideoPlayer的运行过程中，我们可能不希望用户通过其它的方式来更改ActiveMovie控件的状态。因为这有可能会破坏整个程序用户界面的一致性。举一个例子，在正常情况下，用户可以在ActiveMovie控件上右击鼠标来改变ActiveMovie控件的一些显示元素，如面板和控件条等。这些操作都会改变ActiveMovie控件的大小发生改变，而且在这种情况下，容纳该控件的对话框的大小不会同时发生改变，这样就会破坏应用程序的外观。因此，我们有必要禁止用户这样做。

在资源编辑器中打开应用程序的主对话框，再打开ActiveMovie控件的属性页对话框。在General选项卡中选择Disabled；在“重放”选项卡中选择“播放次数”单选钮，并输入播放次数为1；在“控件”选项卡中清除“显示面板”和“控制面板”两个复选框。

现在就可以编译和生成该应用程序了。图7.7为应用程序VideoPlayer播放VCD文件时的效果。

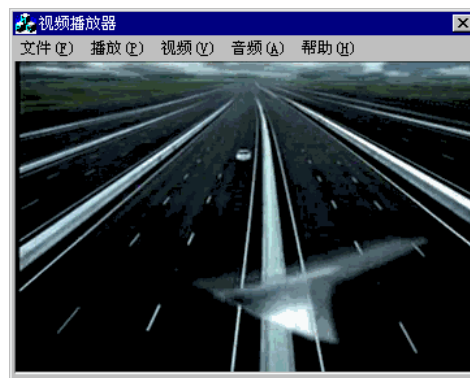


图7.7 应用程序VideoPlayer播放VCD文件时的运行效果

- 技巧：
- 从上面的示例程序可以看出，ActiveMovie控件提供了非常完善的音频和视频媒体文件的回放功能。它能够支持多种文件格式，从最常见的WAV文件和AVI文件到使用MPEG压缩格式的VCD视频文件，都可以正常的进行播放。因此，若正在编写的应用程序需要提供多媒体支持，那么使用ActiveMovie是一个很好的主意。事实上，很多优秀的多媒体应用程序，其内部的多媒体回放就是使用ActiveMovie来实现的。只要精心的设计应用程序的用户界面，我



们可以把一切“隐藏”得滴水不漏。而且，在Windows 95/98和Windows NT的最新版本中，ActiveMovie控件已作为操作系统的一部分来提供。即使用户的系统中没有安装ActiveMovie控件，Microsoft的许可协议也允许在你的应用程序的发行包中发布ActiveMovie的运行时文件。充分的使用各种优秀的第三方ActiveX控件，可使不费力的使用你的应用程序增色不少，这也就是ActiveX控件本身的一个魅力之所在。

## 第八章 文档/视结构

文档/视结构是在Visual C++中使用MFC开发基于文档的应用程序的基本框架，在这个框架中，数据的维护及其显示是分别由两个不同，但又彼此紧密相关的对象——文档和视负责的。文档/视结构在很多场合与传统的编程方式相比要更有利于这一类应用程序的编写。

本章介绍Visual C++中的文档/视结构。这是一个相对比较复杂的课题，所涉及的内容也比较广泛，具有来说有以下一些：

- 文档/视结构以及这种结构以编程带来的便利之处
- 是否使用文档/视结构的考虑
- 使用AppWizard创建基于文档/视结构的框架应用程序
- 使用文档类
  - 在文档类的成员变量中保存文档数据
  - 串行化文档数据
  - 在文档类中处理命令消息
- 使用视类
  - 从文档类中获取数据
  - 在视中显示数据
  - 处理用户输入的信息
  - 更新文档的所有视
  - 视的滚动和缩放
- 多视与多文档
- 打印和打印预览

视结构非常适合于编写这些应用程序。

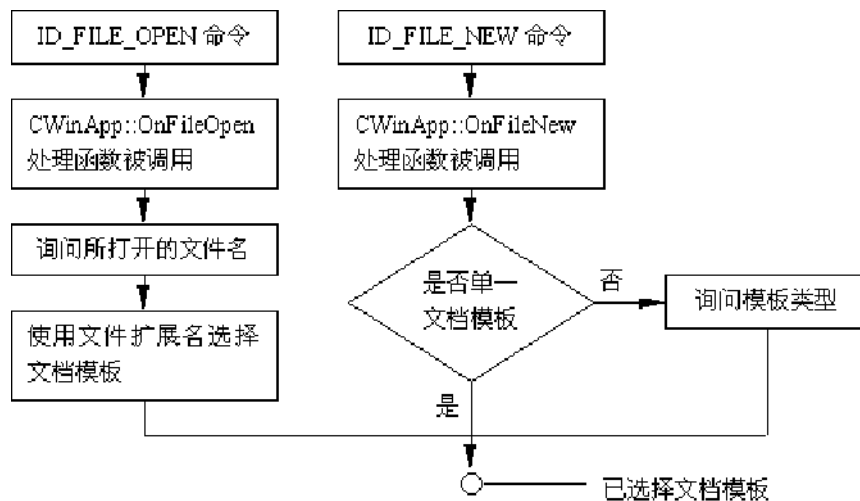


图8.2 文档的创建

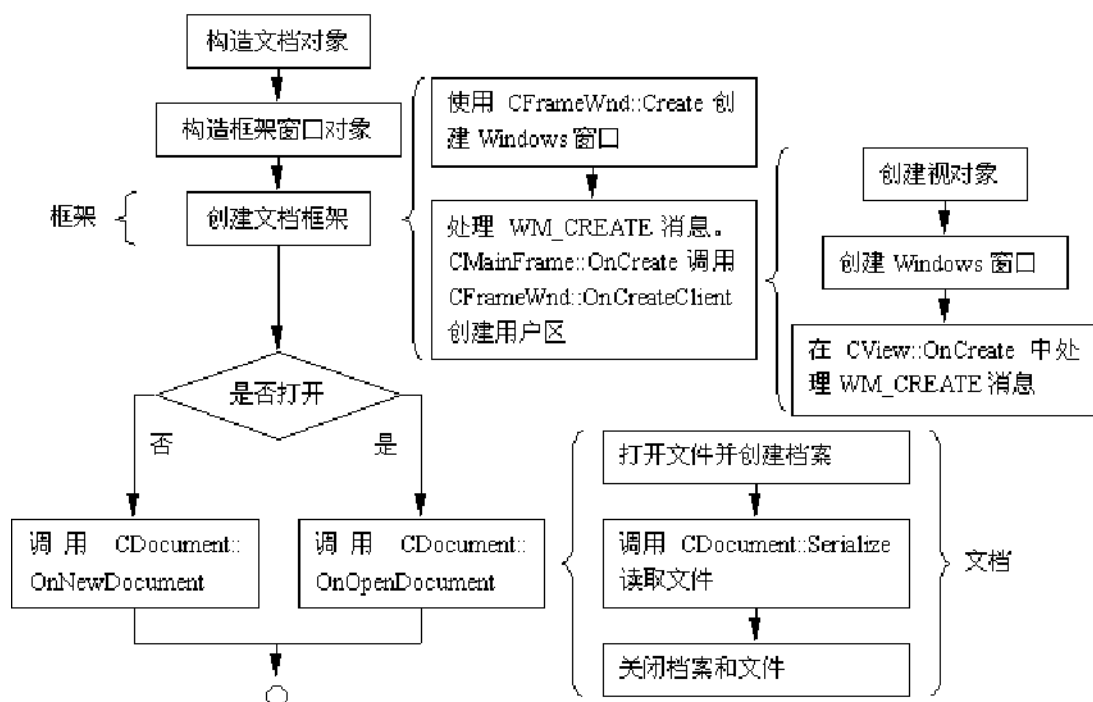


图8.3 框架窗口的创建

文档/视结构尽管有很多的优势，但是，在一些很特殊的情况下，我们仍有可能不需要或者说不应该使用文档/视结构。

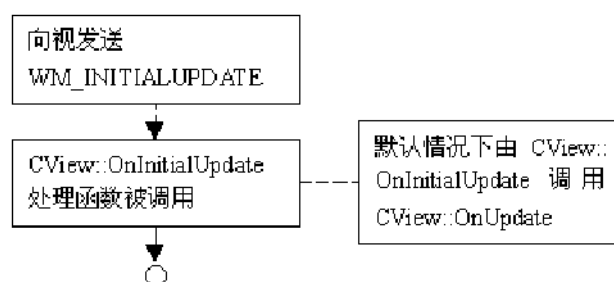


图8.4 视的创建

最典型的一种情况是，如果我们需要移植一些过去编写的Windows应用程序，而在这些程序中，数据的管理和显示是合在一起，要使用文档/视结构来改写它们将会是一件消耗大量人力和物力的工作。对于这些应用程序，最有效的方法是不使用文档/视结构。

在某些情况下，我们既不使用文档，也不使用视，而是在框架窗口中管理和显示数据，这时，我们应该修改应用程序类的InitInstance成员函数，创建自己的框架窗口。这种方式需要的工作量最大，并且要对框架有相当深入的了解，然而，它提供了彻底避开文档/视结构所带来的微小额外开销的方法。

文档对象所产生的微小额外开销来源于文档类本身以及它的基类CCmdTarget和CObject。并且，文档还需要额外的时间来创建文档对象、相关的视对象、框架窗口以及文档模板对象。

- 注意：
- 使用文档对象所带来的额外开销是很微小的。因此，在绝大多数场合，我们都应该选择文档/视结构来编写基于文档的MFC应用程序，而没有必要去考虑因为这种额外开销所带来的性能问题。

## 第二节 使用AppWizard创建框架应用程序

单击Microsoft Developer Studio中的菜单项File|New...，在Project选项卡中选择MFC AppWizard(EXE)，并在Project类和CChildFrame类外，AppWizard所生成的类的名称是基于您所设定的工程名称的，而类的头文件和实现文件的文件名则是基于类名的。通常，我们只能为AppWizard所生成的视类指定另外的基类。可以使用的基类通常包括CEditView、CFormView、CListView、CRichEditView、CScrollView、CTreeView和CView。默认情况下，AppWizard使用CView作为应用程序的视类的基类。事实上，其它的视类也都是CView的派生类。使用其它的视类方便的可以实现某些而不用我们编写额外的代码。关于这些视类的使用请参阅本章的“8.4 CView的派生类”一节。

单击Finish，AppWizard将根据您在上面的步骤中所作的选择为应用程序生成所需的框架文件。

在Introduc应用程序中，我们在所有的步骤中均使用AppWizard的默认设置。

使用AppWizard创建单文档界面的应用程序和上面的过程几乎是完全

一样的。主要的差别在于：AppWizard在创建单文档界面的应用程序时，不生成CChildFrame类，并且，单文档界面应用程序的CMainFrame类的基类为CFrameWnd，而多文档界面应用程序的CMainFrame类的基类为CMDIFrameWnd。

由AppWizard生成的应用程序框架已经是一个可以运行的完整的应用程序了。单击菜单项Build|Execute `Introduc.exe`或按快捷键Ctrl+F5，Visual C++编译并运行由AppWizard所生成的框架应用程序。该应用程序运行结果如图8.5所示。

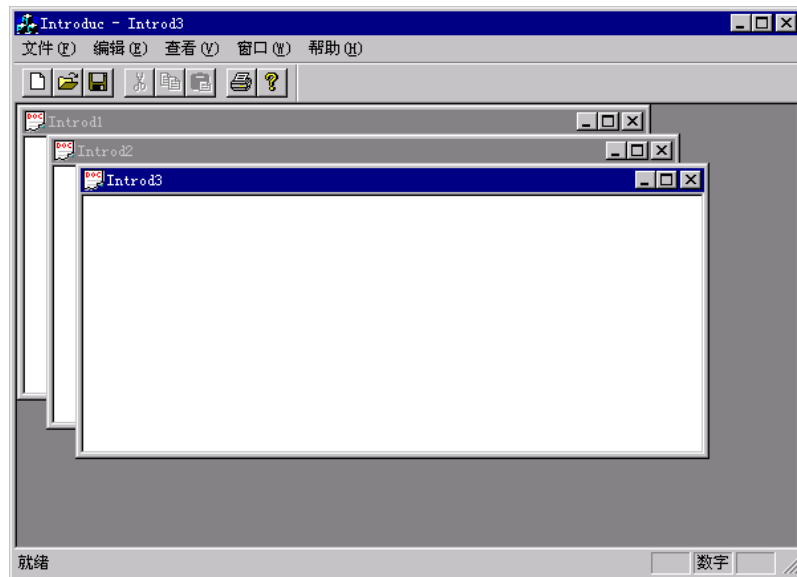


图8.5 使用AppWizard创建的框架应用程序

这个应用程序框架可以运行，并且看起来好象具有完全的功能，但要记住它仅仅是一个框架，具体的功能仍需要我们去添加。AppWizard所生成的代码只是为我们完成了一些标准的功能。如维护文件菜单下的最近文件列表，进行新建窗口、窗口的层叠和平铺以及重排图标等。对于所有提供这些命令的应用程序，其实现方法几乎是完全一样的。在过去，我们不得不编写在每一个需要实现这些功能的应用程序时一行一行地添加相同的代码，现在，AppWizard为我们完成了这些既枯燥乏味又容易出错的工作，从而将程序员从大量的重复性劳动中解放出来，使他们可以有时间去从事更有意义的功能，把更多的精力放到完成应用程序所具有的特定的功能上。

另外，我们还会发现，上面的框架应用程序还可以响应如“打开”、“保存”等命令，工具条上的相应的按钮也可以使用。但是，这些命令本质上什么都没有做，AppWizard只是创建了这些命令实现的框架，程序员需要根据应用程序的特定需求去添加相应的操作。从本章后面几节的讲述来看，添加这些操作的复杂程度比过去小

了很多。

可以使用Workspace窗口的ClassView查看AppWizard所生成的类和类中的成员函数，从中我们可以看到AppWizard在Introduc应用程序的视类和文档类中所重载的基类函数。在后面的章节中，我们需要修改这些重载函数和添加新的成员函数来为特定的应用程序实现所需的功能。

### 第三节 生成文档

在文档/视结构中，文档的任务通常是对数据进行管理和维护。我们通常将数据保存在文档类的成员变量中。视可以直接或间接的访问文档类中的这些成员变量，并通过这种方式来显示和更新数据。关于使用文档类的成员变量来保存数据的详细介绍请参阅“8.3.2 把数据保存到成员变量中”和“8.3.3 使用集合类管理数据”。文档还负责将数据保存到永久存储介质中。常见的情况是将数据保存到磁盘文件或数据库中。在Visual C++的与文档/视结构相关的文档中，我们称这个过程叫串行化(serialize)。MFC类库为数据的串行化提供了默认的支持，我们只需要在此基础中稍加修改就可以为自定义的文档类提供串行化支持。在“8.3.4 数据的串行化”一节中讲述了实现一般的串行化过程的方法和步骤。对象的串行化需要考虑一些额外的问题，这在“8.3.5 串行化对象”中讲述。文档类还可以处理命令消息，这里所谓的命令消息是指来自如菜单、工具栏按钮和加速键的WM\_COMMAND通知消息。与Windows消息和控件通知消息不同，命令消息可以被多种对象处理，这些对象除了窗口和视外，还可以是文档、文档模板或应用程序本身。除了WM\_COMMAND外，文档不能处理其它的Windows消息。

#### 8.3.1 概述

所有的文档类都以CDocument类为其基类。CDocument类提供了文档类所需要的最基本的功能实现。更重要的是，CDocument类为文档对象以及文档和其它对象(如视对象、应用程序对象以及框架窗口等)交互的实现提供了一个框架。我们所做的工作基本上是在这个已有框架的基础上，添加与特定应用程序相关的实现。

从CDocument类派生自己的文档类所需的典型步骤为：

1. 为每一个文档类型从CDocument类(当然也可以是其它CDocument类的派生类)派生一个相应的文档类。

2. 为文档类添加成员变量。这些成员变量用来保存文档的数据，其它对象(如与文档相关联的视)直接或间接的访问这些成员变量来读取或更新文档的数据。

3. 重载Serialize成员函数，实现文档数据的串行化。

如果您的应用程序只使用一种文档类型，那么，在创建应用程序工程时，AppWizard已为我们完成了一部分工作。典型地，AppWizard为应用程序框架生成一个CDocument类的派生类，在默认情况下该类的命名依赖于工程的名称。然后，AppWizard在该文档类中重载了基类的几个成员函数，包括OnNewDocument和Serialize等。但是，AppWizard在这些重载函数中只是简单地调用基类的相应函数，您需要根据自己的应用程序的需要来修改它们。

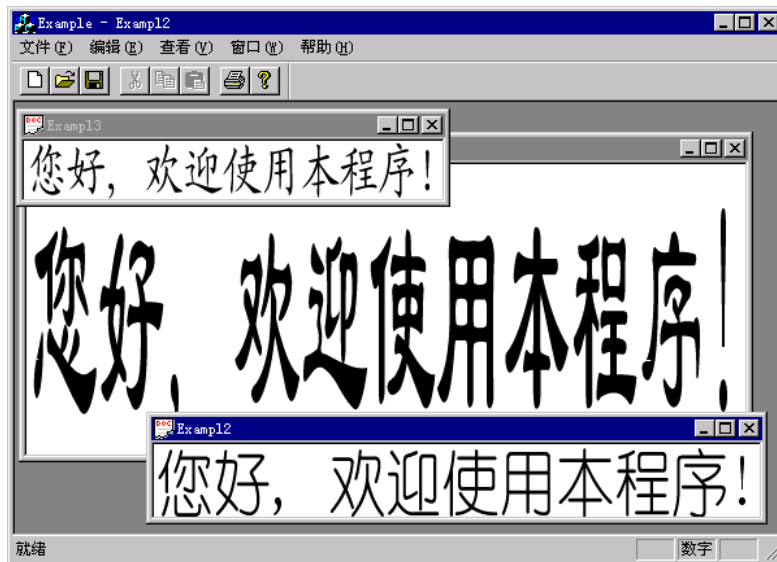


图8.6 示例程序Example的运行结果

### 8.3.2 把文档数据保存到成员变量中

在使用文档/视结构的应用程序中，我们通常使用文档类的成员变量来保存文档的数据。并使其它的对象(如与文档相关联的视)可以访问这些成员变量，从而实现了文档和其它对象(主要是视)的相互搭配使用。

下面我们来看一个简单的例子。该示例程序运行时如图8.6所示。

1. 首先创建一个MFC AppWizard (exe)工程，并取名为Example。如果需要了解如何使用AppWizard创建一个基于文档/视结构的多文档界面应用程序框架，请参阅“8.2.1使用AppWizard创建使用文档/视结构的应用程序所需的步骤”一节中的讲述。

2. 在Workspace窗口的ClassView选项卡中展开Example classes，可以看到AppWizard为Example程序生成的所有类。右击CExampleDoc类，单击Add Member Variable...，在Variable Type框中输入成员变量的类型CString，在Variable Declaration框中输入成员变量名m\_str。由于我们希望其它类的对象可以访问该成员变量，因此在Access框中选择其访问类型为Public。单击“OK”，Visual C++将该成员变量的定义添加类的定义中。

再按照与上面的过程相同的方法，在类CExampleDoc中添加类型为LOGFONT的公有成员变量m\_lf。

我们也可以手动地将成员变量添加到类CExampleDoc的定义中。如下面的步骤所示：

2'. 在Workspace窗口中的FileView选项卡中展开Example files|Header Files。双击ExampleDoc.h，Visual C++将在代码编辑窗口中打开文件ExampleDoc.h，这个文件包括了Example应用程序中的文档类CExampleDoc的定义。

在其中的

```
// Attributes
```

```
public:
```

之后手工地输入

```
CString m_str;
```

```
LOGFONT m_lf;
```

您也许已经发现，在类CExampleDoc的定义中包括多个public块。事实上，您可以把这些定义都放到同一个public块中，在定义中包括多个public块只是为了区别开不同用途的公有成员。例如，在上面所示的代码中，我们将成员变量m\_str的定义放到Attributes块内；而如果您是使用Add Member Variable...添加变量的话，Visual C++是将它添加到Implementation块中。这只是为了便于程序的阅读和维护，对于编译器而言，您将公有成员的定义放到哪一个public块中其结果都是一样的，对于私有成员和保护成员也是一样。

3. 为了测试该程序中，我们在CExampleDoc的OnNewDocument成员变量中为公有成员m\_str赋以初值“您好，欢迎使用本程序！”，并弹出一个字体对话框让用户为该字符串选定字体。方法是使用下面的代



码替换OnNewDocument的实现代码中的// TODO注释：

```
m_str="您好，欢迎使用本程序!";

CFontDialog dlg;

dlg.GetCurrentFont(&m_lf);

// 将用户选定的字体信息填充到LOGFONT类型的结构m_lf中，以供视类使用

if(dlg.DoModal()==IDOK)

dlg.GetCurrentFont(&m_lf);
```

4. 再重复一下，在MFC应用程序中，文档类是和视类一起协作以完成应用程序功能的。下面我们将为Example程序的视类CExampleView类的OnDraw成员函数添加一些代码，以将文档类中的m\_str成员变量的内容显示到视的框架窗口中。关于视类的内容是在本章的“8.4 生成视”一节中讲述的。在本节，为了使应用程序完整并且能够运行，以反映我们对文档类所进行的一些操作，书中给出一些用于视类的代码，并且，为了使章节的行文连贯和有重点，我们并不详细的讲解这些代码。如果您还不是很了解视类的话，大可不必去在意这些代码究竟都是怎样工作的，以及为什么要这样书写这些代码，把本书继续看下去，这些代码都不会成其为问题。但若您现在很想了解这些内容，那也不妨跳过去浏览一下“8.4 生成视”以及“图形设备接口”一章。

这里我们用下面的代码来替换类CExampleView的OnDraw成员函数。

```
// 获取当前客户区的大小

CRect rectClient;

GetClientRect(rectClient);

CSize sizeClient=rectClient.Size();

// 从文件中读取数据

CString str=pDoc->m_str;

LOGFONT lf=pDoc->m_lf;

// 使字体充满整个客户区

lf.lfHeight=sizeClient.cy;

lf.lfWidth=long(sizeClient.cx/str.GetLength());
```

```

// 用当前字体信息生成CFont对象
CFont *pFont=new CFont;
pFont->CreateFontIndirect(&l f);
// 改变当前所用字体，并保存旧字体
CFont *pOldFont=pDC->SelectObject(pFont);
// 用新选定的字体绘制字符串 "您好，欢迎使用本程序!"
CSize sizeTextExtent=pDC->GetTextExtent(str);
pDC->TextOut((sizeClient.cx-sizeTextExtent.cx)/2,
(sizeClient.cy-sizeTextExtent.cy)/2,
str);
// 恢复系统默认字体
pDC->SelectObject(pOldFont);

```

在上面的示例程序中，我们在文档类中定义了两个公有的成员变量 `m_str` 和 `m_l f`，然后在视类的 `OnDraw` 成员函数中访问了这两个成员变量，通过这些变量从文档中获取所要显示的字符串和所使用的字体。

很多时候我们常使用的是另外一种方法，即把成员变量定义为私有或保护成员，然后添加存取该成员变量的函数。如下面的代码所示：

```

protected:
CString m_str;
public:
CString GetStr()
{
CString *pStr=new CString;
*pStr=m_str;
return *pStr;
}

```

这样的好处是可以防止数据成员被从类外部修改，从而维护了类中数

据的安全。

- 注意：
- 不象很多资料上所说的那样，在GetStr函数中使用下面的代码，并不能保证类中的保护成员绝对不会被从外部修改：

- CString GetStr()
- {
- return m\_str;
- }

这时，如果类的使用者在外部使用了如下面的语句所示的强制类型转换，

- CString& str=(CString&)pDoc->GetStr()
- str="字符串将被修改!"

则可以通过引用str来修改类中的保护性成员m\_str的值，读者可以编写程序来自行验证这一点。

但如果不使用强制类型转换将pDoc->GetStr转换为CString&并把str定义为CString（而不是CString&）的话，修改str并不会改变类中的保护成员m\_str的值。

- 使用以const关键字修饰的指针来返回指向私有或保护性数据成员的指针也并不是总是安全的。例如，若GetStr函数的定义如下：

- const CString\* GetStr()
- {
- return (const CString\*)&m\_str;
- }

那么，在类的外部，使用者同样可以使用强制类型转换被声明为const的指针，从而修改私有成员m\_str，如下面的代码所示：

- `CString *pStr=(CString *)pDoc->GetStr();`
- `*pStr="字符串将被修改!";`
- 虽然我们不应该过多地使用这种强制类型转换，但是，在编写类的时候，还是要尽可能的避免可能出现这些不希望发生的事情，以免用户对类的不正当的使用导致某些意外的问题，如使类中的数据不再可用等。如果调用者对自己的所做不是很清楚的话，这很可能导致程序出错，并且难以被检查出来，因为很多的编程者（尤其是初学者）一般不容易想到问题的根源出在类的内部，尽管这种问题是由于错误的使用类造成的。

与添加读取文档数据的公有函数相似，我们还可以添加设置文档数据的公有成员函数，如下面的代码所示：

```
public:

int SetStr(const CString& NewStr)

{

m_str=NewStr;

}
```

使用公有成员函数来存取类中的数据的一个最大的好处在于可以验证用户所传递的数据的有效性。从而避免用户把一个非法的数据赋给类的成员变量，这有可能在后面的使用中导致意外的问题。

### 8.3.3 串行化数据

在Visual C++术语中，我们把对象的保存到永久介质中或从永久介质中读取对象称作串行化。串行化的基本观点是每一个对象都应该能够将自身的当前数据保存到永久介质中，这些数据一般由其成员变量所提供；在需要的时候，对象还应该能够从永久介质中读出所保存的数据，并用这些数据来重建该对象。在本节中，我们只讨论最基本串行化操作，在“8.3.5 串行化对象”一节中描述了串行化对象和生成可串行化对象所需要的附加信息。

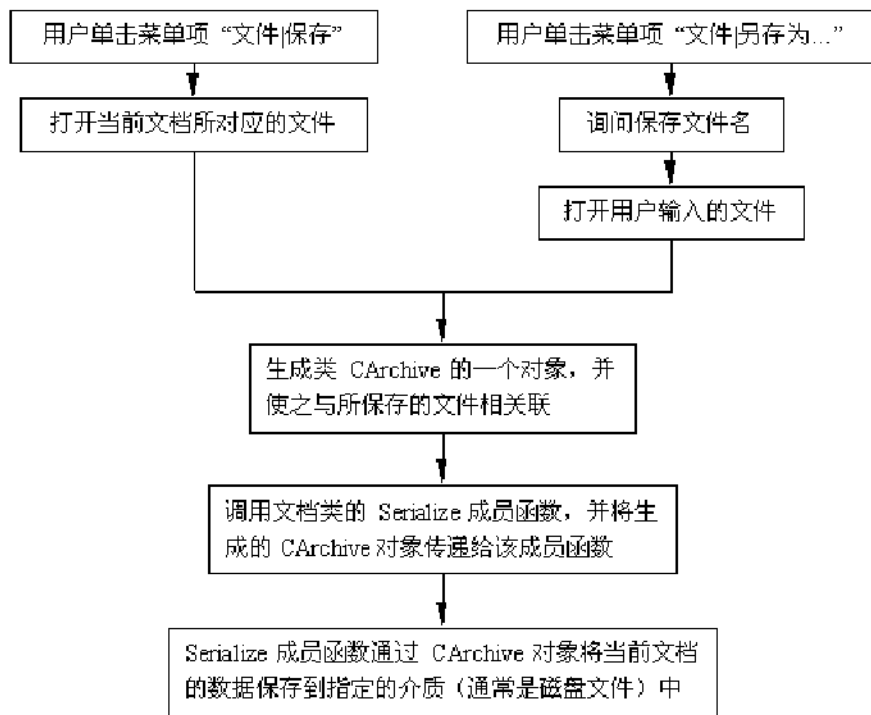


图8.7 保存文件时的串行化过程

在文档/视结构实现中，串行化一般是重载文档对象的Serialize成员函数。在AppWizard创建应用程序框架的时候，生成了一个Serialize重载函数的框架，如下面的代码所示：

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

图8.7描述了保存文档时的串行化过程。

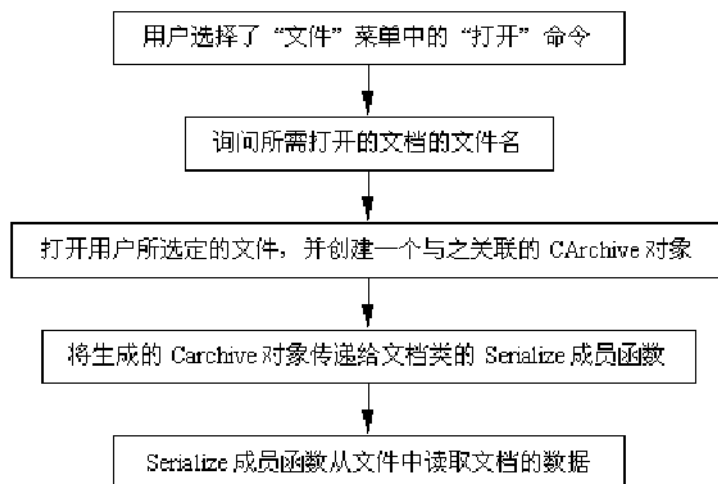


图8.8 打开文档时的串行化过程

图8.8描述了打开文档时的串行化过程。

由图8.7和图8.8可以看出，无论是保存文档或是打开文档，应用程序都是通过调用文档类的Serialize成员函数来完成串行化操作的。因此，在大多数情况下，我们都通过重载Serialize成员函数来实现文档的串行化。Serialize成员函数带有一个CArchive类型的参数，这是一个与所打开的文件相关联的对象。一般情况下，我们总是使用CArchive对象来保存和打开文档。

一个CArchive对象和一个标准的C++输入/输出流相类似，可以使用C++析取运算符>>和插入运算符<<将一个对象串行化到与CArchive对象的存储介质中，两者最大的差别在于标准的C++输入/输出流用来存取格式化的ASCII字符串，而CArchive对象用来存取非冗余的二进制对象，这种格式更为有效，并且应用也更加广泛。

在创建CArchive对象之前必须创建一个CFile对象，并将该CFile对象与CArchive对象相关联。在应用程序框架对文档/视结构的实现中，这一步工作已由框架在调用Serialize成员函数之前完成，我们无需再做。

CArchive对象是单向的，也就是说，同一个CArchive对象只能用于保存或读取两者之一，不能通过同一个CArchive对象既进行文档的保存，又进行文档的读取。在框架创建CArchive对象时，已根据用户选择的是“保存”（“另存为”）还是“打开”来设置了CArchive对象的类型，我们可以使用CArchive类的成员函数IsStoring来检索当前CArchive对象的类型，从而得知用户所期望的操作是保存还是读取，从而执行不同的操作。

下面，我们为示例程序Example完成保存及打开文档的实现。在此之前，我们先在 “编辑” 菜单中添加一条改变文档的显示文件的内容，用户可以将对文档显示文本所做的修改保存到一个磁盘文件中，需要的时候再从磁盘文件中打开并恢复该文档。



图8.9 示例程序Example的 “编辑” 菜单

1. 单击Workspace窗口中的ResourceView选项卡，展开Example resources|Menu，双击IDR\_EXAMPLTYPE，为 “编辑” 菜单添加一个分隔条两个菜单项 “改变显示文本” 和 “改变字体” (如图8.9所示)，并设置新添加的菜单项的ID为ID\_EDIT\_CHANGETEXT和ID\_EDIT\_CHANGEFONT。

2. 为示例程序添加如图8.10所示的对话框。然后使用ClassWizard为对话框生成CDialog类的派生类CInputDialog，并为其中的Edit Box控件 (其ID为IDC\_EDIT1)添加相关联的成员变量m\_input，其类型为CString。



图8.10 “改变显示文本” 对话框

3. 在类中添加公有成员函数GetInput，其类型为CString，参数为空，定义如下：

```
CString CInputDialog::GetInput()
{
    if (DoModal()==IDOK)
        return m_input;
    else
        return "";
}
```

4. 使用ClassWizard在类CExampleDoc中为菜单项 “编辑|改变显示文本” (ID为ID\_EDIT\_CHANGETEXT)添加处理函数OnEditChangeText，其代码如下：

```
void CExampleDoc::OnEditChangeText()
{
    CInputDialog inputDlg;
    CString str=inputDlg.GetInput();
    if (str!="")
    {
        m_str=str;
        UpdateAllViews(NULL);
    }
}
```

为菜单项 “编辑|改变字体” (ID为ID\_EDIT\_CHANGEFONT)添加处理函数OnEditChangeFont，其代码如下：

```
void CExampleDoc::OnEditFont()
{
    CFontDialog dlg;
    dlg.GetCurrentFont(&m_lf);
    if(dlg.DoModal()==IDOK)
    {
        dlg.GetCurrentFont(&m_lf);
        UpdateAllViews(NULL);
    }
}
```

现在编译并运行示例程序，即可以通过单击 “编辑” 菜单下的 “改变显示文本” 和 “改变字体” 来修改文档的显示文本和字体了。为了把这些修改保存到磁盘文件中，并在需要时可以打开所保存



的磁盘文件读取文档，我们重载CExampleDoc类的Serialize函数来完成串行化。重载后的Serialize函数的代码如下：

```
void CExampleDoc::Serialize(CArchive& ar)
```

```
{
```

```
if (ar.IsStoring())
```

```
{
```

```
ar<<m_str;
```

```
ar<<m_lf.lfHeight
```

```
<<m_lf.lfWidth
```

```
<<m_lf.lfEscapement
```

```
<<m_lf.lfOrientation
```

```
<<m_lf.lfWeight
```

```
<<m_lf.lfItalic
```

```
<<m_lf.lfUnderline
```

```
<<m_lf.lfStrikeOut
```

```
<<m_lf.lfCharSet
```

```
<<m_lf.lfOutPrecision
```

```
<<m_lf.lfClipPrecision
```

```
<<m_lf.lfQuality
```

```
<<m_lf.lfPitchAndFamily
```

```
<<CString(m_lf.lfFaceName);
```

```
}
```

```
else
```

```
{
```

```
ar>>m_str;
```

```
CString lfFaceName;
```

```

ar>>m_lf.lfHeight

>>m_lf.lfWidth

>>m_lf.lfEscapement

>>m_lf.lfOrientation

>>m_lf.lfWeight

>>m_lf.lfItalic

>>m_lf.lfUnderline

>>m_lf.lfStrikeOut

>>m_lf.lfCharSet

>>m_lf.lfOutPrecision

>>m_lf.lfClipPrecision

>>m_lf.lfQuality

>>m_lf.lfPitchAndFamily

>>lfFaceName;

strcpy(m_lf.lfFaceName,lfFaceName);

}

}

```

在上面的代码中，我们之所以可以使用如

```
ar<<m_lf.lfHeight<<m_lf.lfWidth...
```

和

```
ar>>m_lf.lfHeight>>m_lf.lfWidth...
```

之类的代码，是因为表达式`ar>>var`和`ar<<var`的结构均为`ar`。

注意到在串行化LOGFONT结构类型的变量`m_lf`时，我们依次对它的每一个成员进行串行化，这是因为对于LOGFONT结构类型的变量，不能使用`<<`和`>>`运算符对其串行化。同样地，我们不能使用`<<`和`>>`运算符串行化`m_lf.lfFaceName`，因为它的类型为`char[32]`。对于`m_lf.lfFaceName`，我们在保存时先将其转换为一个CString对象，在

读取时先将它读入一个CString对象中，然后再将该CString对象赋予m\_lf.lfFaceName。

表8.1给出了可以使用<<和>>运算符进行串行化的数据类型。

表8.1 可以使用<<和>>运算符进行串行化的数据类型

CObject*	SIZE and CSize	float
WORD	CString	POINT and CPoint
DWORD	BYTE	RECT and CRect
double	LONG	CTime and CTimeSpan
int	COleCurrency	COleVariant
COleDateTime	COleDateTimeSpan	

对于除表8.1所示的数据以外的其它数据，我们需要使用另外的方法来实现串行化，如前面的示例所示，关于串行化的更深入的内容请参见本章的后续内容。

#### 8.3.4 使用集合类管理数据

在上面的代码中我们使用单个的成员变量来保存类中的数据。这样，如果文档中有多少数据，就需要定义多少成员变量。然而，在某些情况下，如编写一个管理文本文件的文档类，我们事先很难得知一个文本文件的大小，从而很难在类中定义合适的成员变量来保存它们。为了适应这些场合的需要，MFC类库中提供了一些称作集合的类，使用集合类可以很容易的满足前面所说的要求。

- 注意：
- 在本书中使用专门的章节来讲述MFC中的集合类，这里我们侧重于介绍集合类在文档对象中的使用。在叙述中对一些相关知识的讲述仅仅是为了便于读者更好的理解其中的示例程序来安排的。关于集合类的更为系统的讲解请参阅其它有关章节。

MFC中的集合类分为三种：数组（array）、列表（list）和映射（map）。

数组和通常在C或C++中使用的数组一样，是一个有序的元素序列，其中的每一个元素在连续的整数进行索引，不同的是MFC中的数组类可

以自动的增长以使它可以容纳新添加的元素。如果不需要添加新的元素，那么访问数组中的元素和使用标准的C/C++数组一样的快。

列表也是一个有序的元素序列，但其中的元素并不以一个整数作为其索引，取代的方法是每一个元素（如果它不是列表的开头和结尾的话）都保存了指向前一个元素和后一个元素的指针。列表本身的有序的，但列表中的元素自身的存储则不一定是有序的，从元素中间插入一个元素事实上只需修改前一个元素的指向后一个元素的指针和后一个元素的指向前一个元素的指针，删除一个元素的过程也是类似的，这和数组是不同的，在数组中，要添加或删除元素必须移动数组中位于指定元素之后的所有元素。因此，在需要大量的进行元素的插入或删除时，使用列表要比使用数组快很多。

映射也被称作字典（dictionary）。映射的最大的特点的是它将其中的元素和一个唯一的键值相关联。键和数组中的整数索引相类似，可以使用键来检索映射中的元素，这弥补了列表中的元素不能进行索引的缺点。

表8.2总结了三种集合类的特点。

表8.2 不同集合类的特点

集合类型	是否有序	是否有索引	效率		是否允许重复元素
			插入元素	查找指定的元素	
数组	是	使用整数作为索引	慢	慢	是
列表	是	否	快	慢	是
映射	否	使用键作为索引	快	快	允许重复的值，但不允许重复的键

由MFC提供的集合类分为两种类型：即基于C++模板的集合类和不基于C++模板的集合类。不基于模板的集合类从MFC 1.0开始提供，现在还保留它们的主要原因是为了向后兼容，便于过去编写的程序进行移植。如果您是从头开始使用集合类的话，则应该从新提供的基于模板的集合类中生成类型安全的集合类。

MFC提供的新的基于模板的集合类对于数组、列表和映射的实现如表8.3所示。

表8.3 基于模板的MFC集合类

集合内容	数组	列表	映射
任意类型的对象	CArray	Clist	CMap
指向任意对象的指针	CTypedPtrArray	CTypedPtrList	CTypedPtrMap

表8.4所示的不基于模板的MFC集合类仍可以使用。

您可以根据上面所给的特点来为您的应用程序选定的合适的集合类。下面我们将举例来说明各个集合类的基本使用方法。

表8.4 非模板集合类

数组	列表	映射
CObArray	CObList	CMapPtrToWord
CByteArray	CPtrList	CMapPtrToPtr
CDWordArray	CStringList	CMapStringToOb
CPtrArray		CMapStringToPtr
CStringArray		CMapStringToString
CWordArray		CMapWordToOb
CUIntArray		CMapWordToPtr

### (1) 使用CArray模板创建数组集合

正如同前面所说，很多时候我们使用集合类来保存文档中的数据。在下面的例子中，我们在文档中使用集合类来创建一个功能非常简单的文本文件查看程序。这个实用程序的功能还很单一，这里我们仅是用它来作为说明集合类的使用的示例。

首先我们需要为这个示例创建一个MDI工程，可以使用AppWizard来完成很多模式化的东西，这里我们假定读者已经对如何使用AppWizard创建起始工程非常地熟悉，所以不再重复的讲述这个过程。如果您需

要帮助的话，可以看一看 “ 8.2使用AppWizard创建框架应用程序 ” 和 “ 8.3生成文档 ” 中的示例。在下面的讲述中，我们假定您所创建的工程名为TextViewer，相应的文档类和视类为CTextDoc和CTextView。

刚开始的时候我们考虑使用CArray模板来创建该应用程序，这也是最直观的想法。

要在程序中使用MFC模板类，需要添加头文件afxtempl.h，由于在整个过程中我们都 没有必要去修改这个头文件(并且，我们也不建议您去修改这个头文件)，所以我们可以把它添加工程TextViewer的预编译头文件StdAfx.h中。这样做还可以获得额外的好处，就是没有必要对每一个包含了StdAfx.h头文件的源文件都添加

```
#include <afxtempl.h>
```

一行。由于由AppWizard和ClassWizard生成的文件大多添加了下面的代码：

```
#include <StdAfx.h>
```

因此，我们就可以避免为每一个需要使用模板的源文件都手动的添加对afxtempl.h的包含。

完成这一步之后，在CTextDoc的定义中添加下面的代码：

```
public:
```

```
CArray <CString, CString&> m_text;
```

从CArray模板生成指定类型的元素的数组的语法为

```
CArray <TYPE, ARG_TYPE> myArray;
```

其中使用到了两个参数，第一个参数TYPE为数组类所存储的元素类型，可以指定的元素类型包括：基本C++数据类型、C++结构和类以及其它的用户自定义类型。第二个参数指定在函数参数传递中的使用的数据类型，对于结构和类类型的元素，我们一般都把ARG\_TYPE参数设置为对TYPE参数指定的数据的引用，如上面的例子。把函数参数指定为对数据的引用可以生成更有效的代码，这对于在集合类中使用大的类对象非常有意义。

在本示例中使用向集合类中添加元素是在文档的串行化时进行的。按下面的代码重载类CTextDoc的Serialize成员函数：

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

```
// CTextDoc 串行化
```

```
void CTextDoc::Serialize(CArchive& ar)
```

```
{
```

```
CString str;
```

```
if (ar.IsStoring())
```

```
{
```

```
for (int i=0; i<m_text.GetSize(); i++)
```

```
{
```

```
str=m_text.GetAt(i);
```

```
ar.WriteString(str);
```

```
}
```

```
}
```

```
else
```

```
{
```

```
while (ar.ReadString(str))
```

```
{
```

```
m_text.Add(str);
```

```
}
```

```
}
```

```
}
```

上面的代码同时还说明了遍历数组集合类中的所有元素的方法。其结构如下：

```
for (int i=0; i<myArray.GetSize(); i++)
```

```
{
```

```
TYPE element=myArray.GetAt(i);
```

```
...  
}
```

CArray模板类的成员函数GetSize返回数组的大小，而另一个成员函数GetUpperBound返回当前数组的上界，由于数组索引是基于0的，因此GetUpperBound的返回值要比GetSize的返回值小1。如果GetUpperBound返回-1，则说明当前数组中没有任何元素，在同样的条件下，GetSize返回0。成员函数GetAt的类型为TYPE（请参阅前面所给的定义CArray模板类的语法），返回值为指定的整数索引所对应的元素，其参数必须为一个不小于0和不大于当前数组上界(由GetUpperBound函数返回)的整数，如果向GetAt成员函数传递一个负数或一个大于当前数组上界的整数将导致一个断言失败（failed assertion）。由于GetAt函数的返回值为TYPE，因此它不能够作为一个左值，即是说，您不可以使用下面的代码来改变指定的元素：

```
myArray.GetAt(i)=myType;
```

完成这种操作可以使用ElementAt成员函数，ElementAt函数的使用方法和GetAt完全一样，但由于它返回一个对数组元素的引用(其类型为TYPE&)，因此，把Element成员函数的返回值作为一个左值是合法的，下面的代码可以正常工作，不会导致错误：

```
myArray.Element(i)=myType;
```

使用CArray模板类中重载的运算符[]可以简化上面的操作，而且，由于operator []的返回值既可以是TYPE，也可以是TYPE&，因此，无论是获得指定元素的值，还是把operator []的返回值当作一个左值使用都是合法的。由于使用[]运算符的用法和标准的C/C++数组下标运算符一样，所以可以像使用标准的C/C++数组一个使用从CArray模板生成的数组集合类。需要注意的问题是，仅当使用MFC的调试版本时，CArray模板类的成员函数才会对下标越界进行检查，因此从根本上说，保证数组集合类的下标操作及传递给其它成员函数的索引值不会越界仍是编程者的职责，这和使用标准的C/C++数组时的情况是相似的。

现在回到类CTextDoc的成员函数Serialize中来，我们需要注意它的串行化操作使用了CArchive类的成员函数ReadString和WriteString，而没有使用一般的“<<”和“>>”运算符。这是由于一般情况下，存储在文件文件中的数据是以字符序列的形式排列的，而不是CString对象序列的形式，因此，在读取时使用“>>”运算符将得不到正确的结果；出于同样的原因，如果是使用“<<”



运算符将字符串作为CString对象串行化到文件中去，那么，所生成的文件并不是通常意义上的文本文件，您可以通过修改CTextDoc的Serialize成员函数来证实这一点。在类CArchive的实现中，并没有为字符串重载“<<”和“>>”运算符函数，所以不能使用“<<”和“>>”运算符来串行化字符串，同时，也没有可用的Serialize函数。CArchive类的成员函数ReadString从档案中读取一行字符串，其原型如下：

```
Bool ReadString(CString& rString);
```

```
LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
```

在每二种格式的ReadString函数中，最大读取字符数被限制为nMax-1，而使用对CString的引用作为参数的ReadString函数则无此限(事实上，使用CString受限于当前物理存储空间和CString允许的最大字符串长度INT\_MAX (2,147,483,647，即 $2^{31}-1$ )。当ReadString函数遇到回车换行组合时即停止读取，并将已读取的字符放到缓冲区中。ReadString函数总是删除最后的回车换行组合并在字符串的末尾添加上'\0'。第一种重载形式的ReadString在成功时返回TRUE，失败(如遇到文件尾)时返回FALSE；而第二种重载形式在成功时返回指向文本数据缓冲区的指针，失败时返回NULL。WriteString成员函数将字符串写入档案中，由于在读取时删除了最后的回车换行组合，因此，在写入字符串时要将少掉的回车换行组合添加进去，如前面的代码所示。不同于Write成员函数，WriteString成员函数不会将表示字符串结尾的'\0'写进档案中去。对于文本格式的档案来说，使用ReadString和WriteString是合适的，但对于二进制的文件，应该使用Read和Write成员函数。如果我们现在编写的程序不是一个文本文件查看程序，而是一个十六进制文件传储程序的话，使用Read和Write来替代ReadString和WriteString是必要的(如后面的示例程序FileDump所示)。

如下面的代码所示的对CTextView类的OnDraw成员函数的重载显示文本文件的内容：

```
////////////////////////////////////  
  
// CTextView drawing  
  
void CTextView::OnDraw(CDC* pDC)  
  
// 获得视所属文档对象的指针  
  
CTextDoc* pDoc = GetDocument();
```

```

ASSERT_VALID(pDoc);

// 从文档中获得文本内容

CArray <CString, CString>& text=pDoc->m_text;

// 获得当前字体度量

TEXTMETRIC tm;

pDC->GetTextMetrics(&tm);

// 获得当前客户区的大小

CRect rect;

GetClientRect(&rect);

pDC->DPtoLP(&rect);

// 每行平均字符数

int cpl=rect.Width()/tm.tmAveCharWidth;

// 每行字符高度

int h=int(tm.tmHeight*1.5);

int cl=0; // 所显示的总行数，用来计算文本的输出位置

for (int i=0; i<text.GetSize(); i++)

{

CString str=text[i];

// 指针pTail指向待显示字符串尾，pHead指向当前显示字符串首，

// pCur用来移动当前显示字符串尾

LPCTSTR pHead=str.LockBuffer();

LPCTSTR pTail=LPCTSTR(str)+str.GetLength();

LPCTSTR pCur=pHead;

// 下面的代码实现了文本在窗口内折行的处理

do

{

```

```

// 按每行平均字符长度预置指针pCur
while(pCur-pHead<cpl && pCur<pTail)
pCur=_tcsinc(pCur);
if (pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx<rect.Width())
{
// 当前显示字符串的输出度量小于客户区宽度时,
// 递增当前字符指针至输出度量适合客户区宽度为止
while(pCur<pTail &&
pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx<rect.Width())
{
pCur=_tcsinc(pCur);
}
while(pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx>rect.Width())
{
pCur=_tcsdec(pHead, pCur);
}
// 在客户区绘制文本
pDC->TabbedTextOut(0, (cl++)*h,
pHead, int(pCur-pHead), 0, NULL, 0);
}
else
{
// 当前显示字符串的输出度量大于客户区宽度时,

```

```

// 递减当前字符指针到输出度量适合窗口区宽度为止
while(pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx>rect.Width())
{
pCur=_tcsdec(pHead,pCur);
}
// 在客户区绘制文本
pDC->TabbedTextOut(0, (cl++)*h,
pHead, int(pCur-pHead), 0, NULL, 0);
}
pHead=pCur;
}while(pCur<pTail);
str.UnlockBuffer();
}
}

```

上面的代码支持文本在窗口客户区内的自动回行，并且，对于双字节文本，如中文、日文等能够很好的工作，即是说，它不会把一个双字节字符分成两半，显示于不同的行。为了达到这个目的，需要在预编译头文件StdAfx.h的开头添加下面的几行：

```

#ifdef _UNICODE
#undef _UNICODE
#endif
#ifndef _MBCS
#define _MBCS
#endif

```

上面的代码确认编译器按双字符支持的方式来编译该应用程序。需要注意的是，前面所给的OnDraw函数对双字节字符集是不完善的，甚至对基本的西文字符集也不完善，这种不完善主要体现在两个方

面：一是上面的代码不能进行双字符的行禁则处理，一些不能出现的行首的标点符号，如“。”等仍将有可能出现于行首，同样，一些不能出现在行尾的标点符号，如“《”等仍将有可能出现于行尾；一是它不能保证换行时西文单词的完整性，即是说一个完整的西文单词有可以被分在两行显示，比如“The”可以被分为“Th”和“e”显示在一行的末尾和下一行的开头。考虑这问题会显著地添加代码的复杂程度，因此上面的OnDraw函数中我们没有进行这样的处理。另外，对于较长的文本文件，使用上面的OnDraw函数的效率是很低的，对于当前不可见的文本区域，它也进行不必要输出，这大大降低了程度在重绘客户区时的响应速度。另外，因为没有添加滚动支持，我们目前还不可以看到超出了当前客户区限制的文本内容。由于我们仅在于说明使用集合类来管理文档的数据，因此我们也没有必要完善该应用程序的显示，否则就有喧宾夺主之嫌了。对于该应用程序的进一步的改进，您可以查看本书的其它章节，这些章节对每一个特定主题都有相当详细和较深入的介绍和讨论。

## (2) 使用CList模板创建列表集合

使用CList模板创建列表集合类的方法和使用CArray模板非常相似。我们同样需要两个参数TYPE和ARG\_TYPE，其含义仍如前面的讲述过的一样。如可以使用下面的语句更改示例TextViewer中的类CTextDoc的成员m\_text的定义：

```
CList <CString, CString&> m_text;
```

前面已经提到过，列表并没有一个象数组那样的索引，因此，必须修改CTextView中的OnDraw成员函数以及CTextDoc中的Serialize成员函数来正确的使用和遍历列表中的所有元素。

修改过的Serialize成员函数如下：

```
////////////////////////////////////  
  
// CTextDoc serialization  
  
void CTextDoc::Serialize(CArchive& ar)  
{  
    CString str;  
    if (ar.IsStoring())  
{
```

```

POSITION pos=m_text.GetHeadPosition();

while (pos!=NULL)

{

str=m_text.GetNext(pos);

ar.WriteString(str+CString("\r\n"));

}

}

else

{

while (ar.ReadString(str))

{

m_text.AddTail(str);

}

}

}

```

由于不存在索引的概念，列表使用类型为POSITION的32位值来标识列表中的数据的位置。在上面的代码中，CList的成员函数GetHeadPosition返回列表中第一个元素的位置。成员函数GetNext以一个对POSITION变量的引用为参数，除了返回指定位置的元素外，GetNext还将传递给它的POSITION变量引用的值置为所返回元素的下一个元素的位置，如果该引用的值被置为NULL，则说明已遍历至列表尾。

使用列表时添加元素的方法也与数组有所不同。CList类模板中并没有名为Add的成员函数，取而代之的是两个成员函数AddHead和AddTail。它们分别把指定的元素添加到列表的首部和尾部，并返回所添加元素在列表中的位置，这是一个POSITION类型的值。另外，成员函数InsertBefore和InsertAfter分别将新的元素插入到指定位置之前或之后。前面已经说过，由于列表相对于数组具有特殊的结构，在列表中插入或删除元素并不需要移动插入点之后的元素，因此在列表中插入和删除元素要比对数组进行同样的操作快上很多，尤其是对于较大的集合对象。

按下面的清单修改CTextView类的OnDraw成员函数。

```
void CTextViewView::OnDraw(CDC* pDC)
{
    CTextViewerDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CList<CString, CString*>& text=pDoc->m_text;
    TEXTMETRIC tm;
    pDC->GetTextMetrics(&tm);
    CRect rect;
    GetClientRect(&rect);
    pDC->DPtoLP(&rect);
    int cpl=rect.Width()/tm.tmAveCharWidth;
    int h=int(tm.tmHeight*1.5);
    int cl=0;
    POSITION pos=text.GetHeadPosition();
    while (pos!=NULL)
    {
        CString str=text.GetText(pos);
        LPCTSTR pHead=str.LockBuffer();
        LPCTSTR pTail=LPCTSTR(str)+str.GetLength();
        LPCTSTR pCur=pHead;
        do
        {
            while(pCur-pHead<cpl && pCur<pTail)
            pCur=_tcsinc(pCur);
            if (pDC->GetTabbedTextExtent(pHead,
```

```

int(pCur-pHead), 0, NULL).cx<rect.Width())
{
while(pCur<pTail &&
pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx<rect.Width())
{
pCur=_tcsinc(pCur);
}
while(pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx>rect.Width())
{
pCur=_tcsdec(pHead, pCur);
}
pDC->TabbedTextOut(0, (cl++)*h, pHead,
int(pCur-pHead), 0, NULL, 0);
}
else
{
while(pDC->GetTabbedTextExtent(pHead,
int(pCur-pHead), 0, NULL).cx>rect.Width())
{
pCur=_tcsdec(pHead,pCur);
}
pDC->TabbedTextOut(0, (cl++)*h, pHead,
int(pCur-pHead), 0, NULL, 0);
}

```



```

pHead=pCur;

}while(pCur<pTail);

str.UnlockBuffer();

}

}

```

### 8.3.5 串行化对象

使用MFC对象的一个很大的优点是它可以对它自身进行串行。前面我们曾经介绍过，这些对象可以把自身写入磁盘，但是程序员也许想建立自己的类并对其自身进行串行（与文档无关，这有助于说明串行处理的过程）。

对象可以放在一个对象表中以记录它，MFC库中的类CObList支持这样的表，表对象用来维护其它对象的表。下面介绍如何使用这样的表。具体的讲是要把对象放在以类CObList为父类的类中。建立了这样一个表后就可以自动地将它传到磁盘上，以后再将它读出来，以这种方式串行的对象必须是由基类CObject派生出来的。

通过表8.5我们可以看看CObject的成员函数和数据。

我们首先需要声明类的构造函数，除此以外，还需要在类的定义中包括宏DECLARE\_SERIAL，指明要重载函数Serialize，指明把那一个类的的数据送到磁盘上以及传送的顺序。函数Serialize负责把数据传到磁盘上的过程，该函数是以表中的单个对象而定义的，而不是为整个表定义的。如果要串行一个对象，就要为这些对象定义该函数。另外，还要注意宏IMPLEMENT\_SERIAL的使用，MFC库就是在该宏中增加串行实际所需的函数。宏IMPLEMENT\_SERIAL用于表及表元素。还要注意指明各类的基类。

表8. 5 在类CObject中定义的成员函数和数据

成员	含义
AssertValid	对象是否合法
CObject	复制构造函数
CObject	缺省的构造函数
Dump	诊断对象

GetRuntimeClass	取CRuntimeClass结构
IsKindOf	测试与指定类的关系
IsSerailizable	测试对象是否可串行
operator=	赋值
operator delete	删除
operator new	创建
Serialize	从档案装入或存入档案
~CObject	析构函数
DECLARE_DYNAMIC(宏)	对访问运行时类信息授权
DECLARE_SERIAL(宏)	可串行
IMPLEMENT_DYNAMIC(宏)	访问运行时类信息
IMPLEMENT_SERIAL(宏)	实现串行
RUNTIME_CLASS(宏)	提取CRuntimeClass结构

## 第四节 生成视

生成视的第一个步骤是要给它加上处理键输入的能力，因而对类CkeyView进行自定义使它能处理击键。键击消息只有五个（发往具有当前焦点的窗口），如下所示：

WM_KEYDOWN	键已按下
WM_SYSKEYDOWN	系统键被按下
WM_KEYUP	键已释放
WM_SYSKEYUP	系统键已

## 释放

WM\_CHAR

## 转换键

当某一键被按下时产生WM\_KEYDOWN消息，而当某一键被释放时产生WM\_KEYUP消息。另外，Windows对于系统按键加以区分（这些键对应于Windows命令，通常是与<Alt>键的组合，其中包括切换活动窗口的ALT ESC）这些消息是WM\_SYSKEYDOWN和WM\_SYSKEYUP，但本书目前不使用系统键盘消息。

MFC库中的视图函数OnKeyDown和OnKeyUp按下列方式处理WM\_KEYDOWN和WM\_KEYUP消息：

```
void OnKeyDown(UINT nChar,UINT nRepCnt,UINT nFlags);
```

```
void OnKeyUp(UINT nChar,UINT nRepCnt,UINT nFlags);
```

当程序员表明要处理WM\_KEYDOWN和WM\_KEYUP消息时,Visual C++ 将生成这些函数的框架。

如果键刚被按下,例如WM\_KEYDOWN消息,则迁移状态位为0;如果键刚被释放(WM\_KEYUP消息),则迁移状态位为1。如果该键在此之前被释放过,则前一键位为0;如果该键在此之前是被按下的,则前一键位为1。如果ALT键被按下,则描述表代码位为1;在一般情况下,该位对应于WM\_KEYDOWN和WM\_KEYUP为0,对于系统消息为1。另外,键击是按下或释放IBM增强型键盘(PS/2所用)上的附加键之一的结果,则扩充标志位为1。注意,程序一般不使用扩充键。

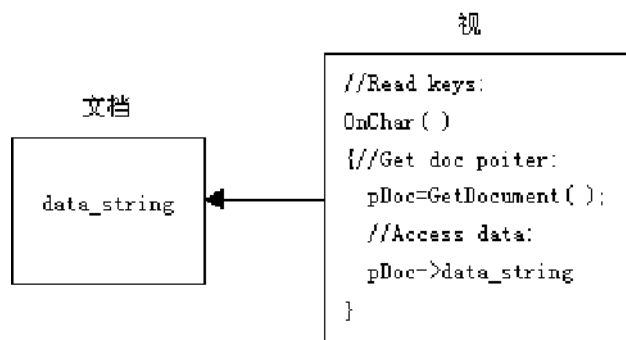


图8. 11 OnChar的处理原理

OEM (OEM代表原设备制造商(Original Equipment Manufacturer)), 扫描码存有由键盘产生的某个键的扫描码。对于每个键击或合法的键击组合(如Shift-a), 键盘都将产生一个唯一的扫描码。而重复计数值是持续按键不放这个动作的表征。如果用户按下某一个键并产生该键

的自动重复，则这个字段存有重复操作的次数。对于连续按键不放的动作，Windows通常不会产生单独的WM\_KEYDOWN或WM\_SYSKEYDOWN消息，否则这些消息会清掉消息队列。Windows将这些消息串到一起，并把一个非零值赋给重复计数参数nRepCnt。

处理键盘消息的一个更有效的途径是处理消息WM\_CHAR，即在类CView中建立OnChar，下面的图8.11显示了这种处理的原理：

- 注意：
- 下面的例程我们假定你建立了一个简单的单文档应用程序。同时，为了同今后的讲解方便，我们假设该程序名为Key。

如图8.12，我们在我们的程序中加入了类CView的派生类CKeyView，从而建立起我们的视的处理。

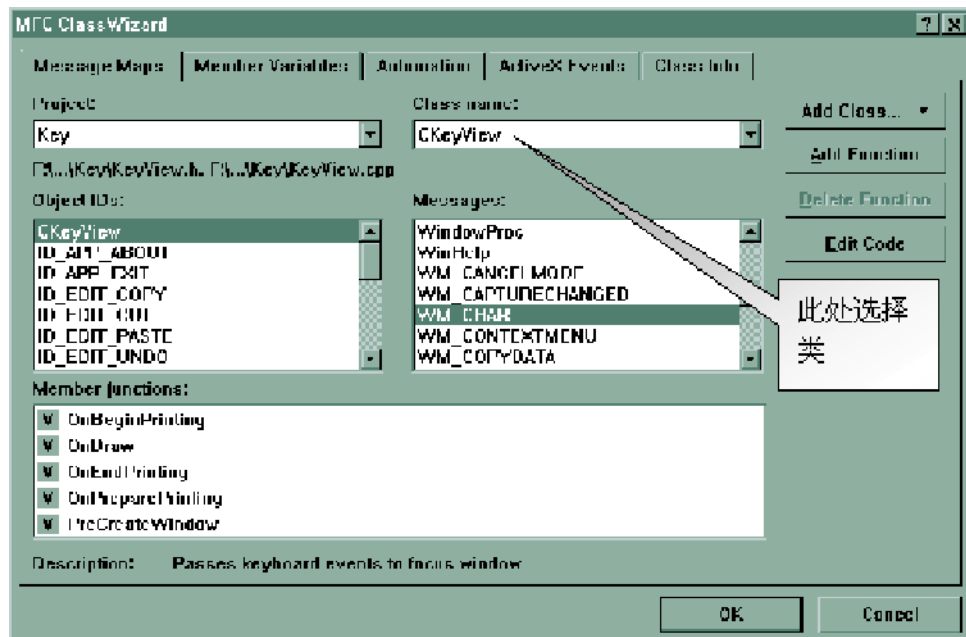


图8.12 建立一个视

在CKeyView内生成函数OnChar( )的框架。当ClassWizard框消失后，打开文件keyview.cpp查看各成员函数的定义，可以在最下面（当然指你还没有为其它消息建立响应时）找到OnChar的定义：

```
void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

```
}
```

- 注意：
- ClassWizard已经生成了调用基类的函数OnChar ( ) ( 如 CView::OnChar ( ) 的代码，如果不准备处理消息，或者希望进行一些预处理，通常最好调用基类的该函数的重载版本。

App Wizard在keyview.cpp已经用MFC的宏BEGIN\_MESSAGE\_MAP完成WM\_CHAR到OnChar的连接，如下所示：

```
IMPLEMENT_DYNCREATE(CKeyVies,CView)

//前面的宏对于文档模板的使用是必要的,它声明了需要使用文档模板的函数

BEGIN_MESSAGE_MAP(CKeyView, CView)

//{{AFX_MSG_MAP(CKeyView)

ON_WM_CHAR() //此处生成了消息的特定连接

//}}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( )
```

- 注意：
- 该消息映射是系统内定的，换句话说，该消息的处理函数是系统固定调用的，这在上一步中选择消息处理函数时也可以看出来：虽然生成了处理该消息的函数，但该函数的处理函数名已经不允许读者改动。同时，由于为系统消息，该消息的处理并不调用宏ON\_COMMAND来进行映射。

消息映射宏将Windows消息连接到MFC库中的ON x x x函数。在上面的程序中，宏ON\_WMCHAR建立了OnChar，这个宏是形式为ON\_WM\_x x x的预定义的消息映象宏之一，其中WM\_x x x对应于想要截获的Windows消息，最终生成的函数就是On x x x。

另外，OnChar的声明也已经加到类CKeyView的声明中，如下所示（摘

自keyview.h)，其中afx\_msg项目前是MFC库中的一个空串，afx代表应用程序框架。

```
class CKeyView : public CView
{
protected: // create from serialization only
CKeyView();
DECLARE_DYNCREATE(CKeyView)
// ...
// Generated message map functions
protected:
//{{AFX_MSG(CKeyView)
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

现在，所必需的一切东西都已建立完毕，因而当某个键被按下时，就可以调用函数CKeyView::OnChar（）。如：

```
void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
CKeyDoc *pDoc=GetDocument();
// ...
CView::OnChar(nChar, nRepCnt, nFlags);
}
```

指针pDoc指向文档，因此可以用pDoc->data\_string访问保存nChar的数据串。

现在实现对数据的采集工作，已经变的相当轻松。只需要利用MFC库

中的类的现成功能即可，直接将该字符串相加。将nChar中的字符加到pDoc->data\_string上，只需要下面一行：

```
void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    CKeyDoc *pDoc=GetDocument();
    pDoc->data_string+=nChar;
    // ...
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

下面接下来的一步是把这个数据显示在屏幕上（这是视图的任务），此时最简单的办法是用类CClientDC来生成一个对应于视图中的用户区的新的设备描述表，使用该设备描述表要比使用别的更一般的设备描述表容易。为此，需要把一个指向该对象的指针传给CClientDC 的构造函数，这可以通过关键字this完成。即：

```
void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CKeyDoc *pDoc=GetDocument();
    pDoc->data_string+=nChar;
    // ...
    CView::OnChar(nChar, nRepCnt, nFlags);
    CClientDC dc(this);
    // ...
}
```

现在已经有有了一个附加在视图上的设备描述表，可以象以前一样使用TextOut（）在它里面打印，打印pDoc->data\_string的代码如下：

```
void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
```

```

// TODO: Add your message handler code here and/or call default

CKeyDoc *pDoc=GetDocument();

pDoc->data_string+=nChar;

// ...

CView::OnChar(nChar, nRepCnt, nFlags);

CClientDC dc(this);

dc.TextOut(0,0,pDoc->data_string,Pdoc->data_string.GetLength ( ) );

// ...

}

```

以上即为OnChar的全部代码。现在当键入串时，它将显示在屏幕上，读取键击的程序取得了成功。图8.13为一个运行画面。

- 注意：
- 别忘了在CKeyDoc.h中加入数据data\_string的声明：
- CString data\_string;

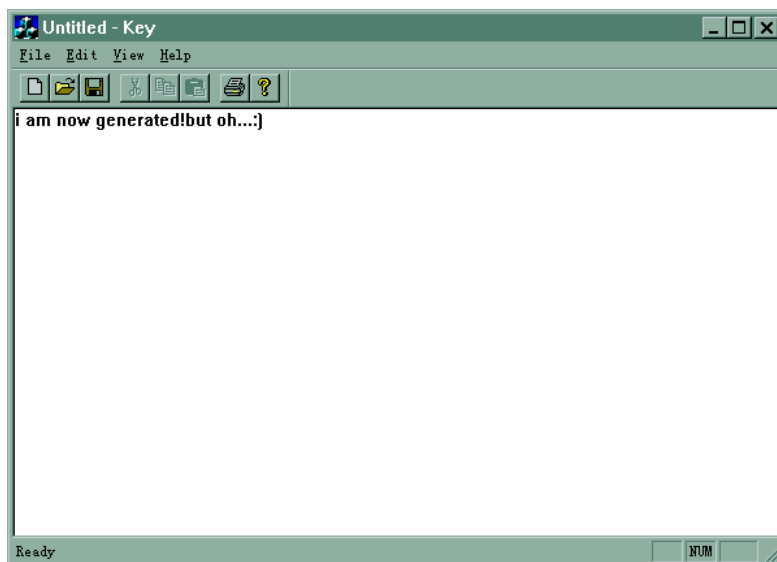


图8.13 生成一个视

- 注意：
- 在程序中，很明显的还有很大的缺陷：



1. 虽然我们的程序是为单文档服务的，在多文档程序中，还需要在一个文档发生改变时通知其所有视图更新自己的数据，则可以通过下列的函数调用 `UpdateAllViews`来完成。当然，对于我们现在的单文档程序还没有涉及到这一点。
2. 在视窗中的输入在窗口失去再重新得到焦点时，数据不能被正确显示。这应该在窗口重绘函数 `OnDraw`中考虑。
3. 如果按通常的字处理程序来看，该程序至少还需要一个适宜的插入符。这点我们在以后的程序中会作出事例。
4. 视的滚动及缩放处理我们放到多文档界面中再作详细解释。下面，我们列出将上述问题加以解决后的函数 `OnChar`和重绘函数 `OnDraw`:

```
• void CKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
• {
• // TODO: Add your message handler code here and/or call default
• CKeyDoc *pDoc=GetDocument();
• pDoc->data_string+=nChar;
• CView::OnChar(nChar, nRepCnt, nFlags);
• CClientDC dc(this);
• dc.TextOut(0,0,pDoc->data_string,pDoc->data_string.GetLength());
• pDoc->UpdateAllViews(this,0l,0);
• }
•
• void CKeyView::OnDraw(CDC* pDC)
• {
• CKeyDoc* pDoc = GetDocument();
• ASSERT_VALID(pDoc);
```

- 
- // TODO: add draw code for native data here
- pDC->TextOut(0,0,pDoc->data\_string,pDoc->data\_string.GetLength());
- }

在本节的末尾，我们给出一个较长一些的例子。它使用了有关视的一些典型方法，能打开VC程序的源代码，同时，其中也涉及到一些文件操作，但由于我们使用的是标准的文件处理对话框，应该不会对读者的阅读带来很大困难。但即使暂时不能全部看懂也没有关系，在本章的随后的几节里，我们会作尽量详细的解释。同时，我们也只是在其中简单的使用了视图的滚动处理。更详细的解释，我们认为留到多文档程序中讲解会更容易接受。

下面给出程序的一部分运行画面（图8.14到图8.16）。

程序源代码（由于篇幅原因，对程序代码没有作详细解释）：

```
// filelist.h : main header file for the FILELIST application

//

#ifdef __AFXWIN_H__

#error include 'stdafx.h' before including this file for PCH

#endif

#include "resource.h" // main symbols

////////////////////////////////////

// DApp:

// See filelist.cpp for the implementation of this class

//
```



图8.14 打开文件

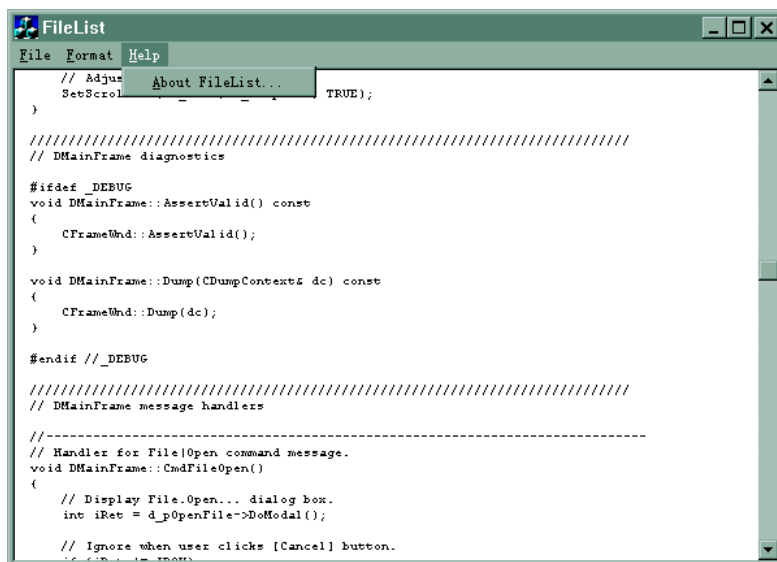


图8.15 对程序选择字体

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

    // Overrides

    // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(DApp)
public:
```

```
virtual BOOL InitInstance();

//}}AFX_VIRTUAL
```

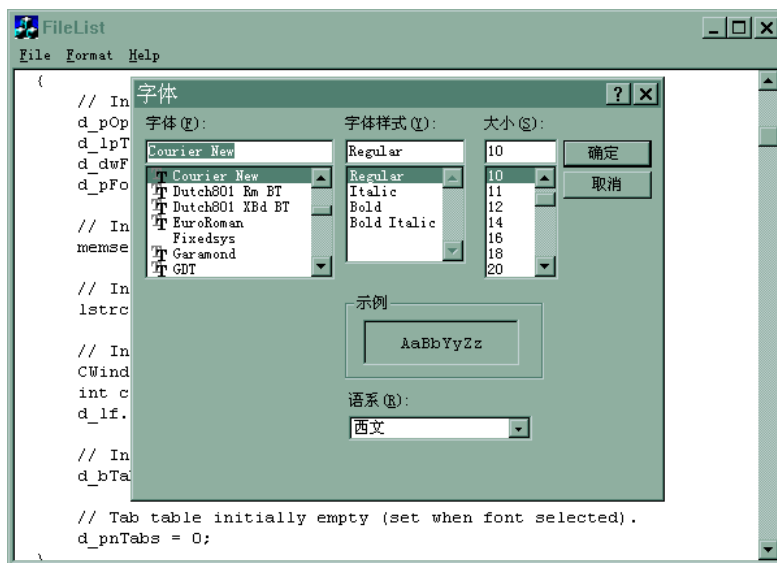


图8.16 字体已被改变

```
// Implementation

//{{AFX_MSG(DApp)

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()

};

////////////////////////////////////

// filelist.cpp: This program opens and displays text files.

//

#include "stdafx.h"

#include "filelist.h"

#include "mainfrm.h"

#ifdef _DEBUG
```

[illegible]

```

// DApp initialization

BOOL DApp::InitInstance()

{
// Step 1: Allocate C++ window object.

DMainFrame * pFrame;

pFrame = new DMainFrame();

// Step 2: Initialize window object.

pFrame->LoadFrame(IDR_MAINFRAME);

// Make window visible

pFrame->ShowWindow(m_nCmdShow);

// Assign frame as application's main window

m_pMainWnd = pFrame;

return TRUE;

}

////////////////////////////////////

// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog

{

public:

CAboutDlg();

// Dialog Data

//{{AFX_DATA(CAboutDlg)

enum { IDD = IDD_ABOUTBOX };

//}}AFX_DATA

// Implementation

protected:

```

```

virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

//{{AFX_MSG(CAboutDlg)

// No message handlers

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
//{{AFX_DATA_INIT(CAboutDlg)

//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)

//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)

// No message handlers

//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog

void DApp::OnAppAbout()
{
CAboutDlg aboutDlg;

```

```

aboutDlg.DoModal();

}

/////////////////////////////////////////////////////////////////

// DApp commands

// mainfrm.h : interface of the DMainFrame class

//

/////////////////////////////////////////////////////////////////

#include <afxtempl.h> // Connect to CArray template.

// Structure to track lines in text file.

class STRING

{
public:
    LPTSTR pText;

    int ccLen;

};

// This declaration uses the CArray template

// to provide a dynamic array of STRING records.

class StringArray : public CArray <STRING, STRING &>

{
public:

    StringArray() {}

    STRING& operator=(STRING & s)

    {m_pData->pText = s.pText;

    m_pData->ccLen = s.ccLen;

    return *m_pData;}

};

```



```

const int g_nTabCount=25;

const int g_nTabStop=4;

class DMainFrame : public CFrameWnd
{
public:

DMainFrame();

protected: // create from serialization only
DECLARE_DYNCREATE(DMainFrame)

// Attributes

public:

BOOL d_bTabs; // Respect tabs or not.

CFileDialog * d_pOpenFile; // Ptr to OpenFile dialog.

CFontDialog * d_pSetFont; // Ptr to Font Picker dialog.

CFont * d_pFont; // Ptr to CFont object.

COLORREF d_crForeground; // Foreground text color.

COLORREF d_crBackground; // Background text color.

LOGFONT d_lf; // Current logical font.

LPINT d_pnTabs; // Array of tab settings.

LPTSTR d_lpTextBuffer; // Ptr to text buffer.

DWORD d_dwFileLength; // File length.

StringArray d_saTextInfo; // Array of string info.

int d_cLines; // Count of lines in text.

int d_cyLineHeight; // Height of one line of text.

int d_clHeight; // Window height line count.

int d_iTopLine; // Index of top-most line.

int d_cyClient; // Window client area height.

```

```

int d_cxLeftMargin; // Margin to left of text.

// Operations

public:

void BuildStringArray();

BOOL CreateNewFont();

void ResetScrollValues();

// Overrides

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(DMainFrame)
//}}AFX_VIRTUAL

// Implementation

public:

virtual ~DMainFrame();

#ifdef _DEBUG

virtual void AssertValid() const;

virtual void Dump(CDumpContext& dc) const;

#endif

// Generated message map functions

protected:

//{{AFX_MSG(DMainFrame)
afx_msg void CmdFileOpen();
afx_msg void CmdFormatFont();
afx_msg void CmdFormatTabs();
afx_msg void UpdFormatTabs(CCmdUI* pCmdUI);
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnPaint();

```

```

afx_msg void OnSize(UINT nType, int cx, int cy);

afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);

afx_msg void OnWinIniChange(LPCTSTR lpszSection);

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

/////////////////////////////////////////////////////////////////

// mainfrm.cpp : implementation of the DMainFrame class

//

#include "stdafx.h"

#include "filelist.h"

#include "mainfrm.h"

#ifdef _DEBUG

#undef THIS_FILE

static char BASED_CODE THIS_FILE[] = __FILE__;

#endif

/////////////////////////////////////////////////////////////////

// DMainFrame

IMPLEMENT_DYNCREATE(DMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(DMainFrame, CFrameWnd)

//{{AFX_MSG_MAP(DMainFrame)

ON_COMMAND(ID_FILE_OPEN, CmdFileOpen)

ON_COMMAND(ID_FORMAT_FONT, CmdFormatFont)

ON_COMMAND(ID_FORMAT_TABS, CmdFormatTabs)

ON_UPDATE_COMMAND_UI(ID_FORMAT_TABS, UpdFormatTabs)

ON_WM_CREATE()

```

```

ON_WM_PAINT()

ON_WM_SIZE()

ON_WM_VSCROLL()

ON_WM_WININICHANGE()

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////

// DMainFrame construction/destruction

DMainFrame::DMainFrame()

{

// Init all data members to a known value.

d_pOpenFile = 0;

d_lpTextBuffer = 0;

d_dwFileLength = 0;

d_pFont = 0;

// Init desired font.

memset (&d_lf, 0, sizeof(LOGFONT));

// Initial font face name is Courier New.

lstrcpy (d_lf.lfFaceName, _T("Courier New"));

// Initial font size is 10 pt.

CWindowDC dc(NULL);

int cyPixels = dc.GetDeviceCaps(LOGPIXELSY);

d_lf.lfHeight = (-1) * MulDiv(10, cyPixels, 72);

// Initial tab setting is OFF.

d_bTabs = TRUE;

// Tab table initially empty (set when font selected).

```

```

d_pnTabs = 0;

}

DMainFrame::~DMainFrame()

{
    if (d_pFont) delete d_pFont;
    if (d_pnTabs) delete [] d_pnTabs;
    if (d_pOpenFile) delete d_pOpenFile;
    if (d_pSetFont) delete d_pSetFont;
}

////////////////////////////////////

// DMainFrame helper functions.

//-----

// BuildStringArray -- Parses text file to create a CString array.

void DMainFrame::BuildStringArray()

{
    // Scan buffer to calculate line count.

    LPTSTR lpNext = d_lpTextBuffer;
    LPTSTR lpEnd = d_lpTextBuffer + d_dwFileLength - 1;
    *lpEnd = '\n';

    for (d_cLines = 0; lpNext < lpEnd; d_cLines++, lpNext++)
    {
        // Search for next <CR> character.

        lpNext = strchr(lpNext, '\n');

        // Discontinue if NULL encountered.

        if (lpNext == NULL) break;
    }
}

```

```

// Set array size.

d_saTextInfo.SetSize(d_cLines);

// Scan buffer to build array of pointers & sizes.

STRING string;

lpNext = d_lpTextBuffer;

for (int iLine = 0; iLine < d_cLines; iLine++)

{
// Char count for current line.

string.ccLen = 0;

string.pText = lpNext;

// Loop to end-of-line.

while ((*lpNext != '\n') && (*lpNext != '\r'))

{

lpNext++; // Check next char.

string.ccLen++; // Increment length counter.

}

// Enter value in array.

d_saTextInfo[iLine] = string;

// Skip over <CR> <LF>

lpNext += (2 * sizeof(char));

}

}

//-----

// CreateNewFont -- Creates new CFont for use in drawing text.

BOOL DMainFrame::CreateNewFont()

{

```

```

// Delete any previous font.

if (d_pFont)

delete d_pFont;

// Create a new font

d_pFont = new CFont();

if (!d_pFont) return FALSE;

d_pFont->CreateFontIndirect(&d_lf);

// Calculate font height

CClientDC dc(this);

TEXTMETRIC tm;

dc.SelectObject(d_pFont);

dc.GetTextMetrics(&tm);

d_cyLineHeight = tm.tmHeight + tm.tmExternalLeading;

// Calculate left margin.

d_cxLeftMargin = tm.tmAveCharWidth * 2;

// Rebuild tab setting table.

if (d_pnTabs) delete [] d_pnTabs;

d_pnTabs = new INT[g_nTabCount];

for (int i=0; i<g_nTabCount; i++)

{

d_pnTabs[i] = d_cxLeftMargin + (i * g_nTabStop * tm.tmAveCharWidth);

}

return TRUE;

}

//-----

// ResetScrollValues -- Adjust scrolling for window size changes.

```

```

void DMainFrame::ResetScrollValues()
{
    // Set count of lines in window height.
    d_clHeight = d_cyClient / d_cyLineHeight;
    // Hide scroll bars when not needed.
    if (d_cLines <= d_clHeight)
    {
        SetScrollRange(SB_VERT, 0, 0, TRUE);
        d_iTopLine = 0;
        return;
    }
    // Adjust scroll range for new window size.
    SetScrollRange (SB_VERT, 0, d_cLines - d_clHeight, TRUE);
    // Adjust scroll thumb position.
    SetScrollPos(SB_VERT, d_iTopLine, TRUE);
}

/////////////////////////////////////////////////////////////////

// DMainFrame diagnostics
#ifdef _DEBUG
void DMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void DMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

```



```

}

#ifdef _DEBUG

/////////////////////////////////////////////////////////////////

// DMainFrame message handlers

//-----

// Handler for File|Open command message.

void DMainFrame::CmdFileOpen()
{
    // Display File.Open... dialog box.

    int iRet = d_pOpenFile->DoModal();

    // Ignore when user clicks [Cancel] button.

    if (iRet != IDOK)

        return;

    // Fetch fully-qualified file name.

    CString csFile = d_pOpenFile->GetPathName();

    // Open file in read-only mode.

    CFile cfData;

    if (!cfData.Open(csFile, CFile::modeRead))

    {

        AfxMessageBox(_T("Error Opening File"));

        return;

    }

    // Free previous buffer contents

    if (d_lpTextBuffer)

    {

        free(d_lpTextBuffer);

```

```

d_lpTextBuffer=0;

}

// Calculate file size & allocate buffer.

// Pad to avoid bad address reference.

d_dwFileLength = cfData.GetLength() + sizeof(char);

d_lpTextBuffer = (LPTSTR)malloc (d_dwFileLength);

if (!d_lpTextBuffer)

{

AfxMessageBox(_T("Cannot Allocate Memory For File"));

return;

< // Loop through client area coordinates to draw.

int cyLine = 0;

while (iLine < cLastLine)

{

// Draw a line of text.

LPTSTR lp = d_saTextInfo[iLine].pText;

int cc = d_saTextInfo[iLine].ccLen;

// Drawing function depends on 'respect tabs' setting.

if (d_bTabs && d_pnTabs != 0)

{

dc.TabbedTextOut(d_cxLeftMargin, cyLine, lp, cc,

g_nTabCount, d_pnTabs, 0);

}

else

{

dc.TextOut(d_cxLeftMargin, cyLine, lp, cc);

```

```

}

// Increment various counts.

cyLine += d_cyLineHeight;

iLine++;

}

}

//-----

// WM_SIZE message handler.

void DMainFrame::OnSize(UINT nType, int cx, int cy)
{
// Notify base class of new window size.

CFrameWnd::OnSize(nType, cx, cy);

// Save client area height.

d_cyClient = cy;

// Recalculate scroll bar info.

ResetScrollValues();

}

//-----

// WM_VSCROLL message handler.

void DMainFrame::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
// Temporary "new line" value.

int iTop = d_iTopLine;

// Based on particular scroll button clicked, modify top line index.

switch(nSBCode)
{

```

```
case SB_BOTTOM:
    iTop = d_cLines - d_clHeight;
    break;

case SB_ENDSCROLL:
    break;

case SB_LINEDOWN:
    iTop++;
    break;

case SB_LINEUP:
    iTop--;
    break;

case SB_PAGEDOWN:
    iTop += d_clHeight;
    break;

case SB_PAGEUP:
    iTop -= d_clHeight;
    break;

case SB_THUMBPOSITION:
    iTop = nPos;
    break;

case SB_THUMBTRACK:
    iTop = nPos;
    break;

case SB_TOP:
    iTop = 0;
    break;
```

```

}

// Check range of new index;

iTop = max (iTop, 0);

iTop = min (iTop, d_cLines - d_clHeight);

// If no change, ignore.

if (iTop == d_iTopLine) return;

// Pixels to scroll = (lines to scroll) * height-per-line.

int cyScroll = (d_iTopLine - iTop) * d_cyLineHeight;

// Define new top-line value.

d_iTopLine = iTop;

// Scroll pixels.

ScrollWindow(0, cyScroll);

// Adjust scroll thumb position.

SetScrollPos(SB_VERT, d_iTopLine, TRUE);

}

//-----

// WM_WININICHANGE message handler.

void DMainFrame::OnWinIniChange(LPCTSTR lpszSection)

{

CFrameWnd::OnWinIniChange(lpszSection);

// Get new background & foreground colors in case these have changed.

d_crForeground = GetSysColor(COLOR_WINDOWTEXT);

d_crBackground = GetSysColor(COLOR_WINDOW);

// Force redraw of window.

Invalidate();

}

```

```
//Microsoft Developer Studio generated resource script.

//

#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////

//

// Generated from the TEXTINCLUDE 2 resource.

//

#include "afxres.h"

////////////////////////////////////

#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////

// Chinese (P.R.C.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)

#ifdef _WIN32

LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED

#pragma code_page(936)

#endif // _WIN32

#ifdef APSTUDIO_INVOKED

////////////////////////////////////

//

// TEXTINCLUDE

//

1 TEXTINCLUDE DISCARDABLE

BEGIN

"resource.h\0"
```

////////////////////////////////////

//

// Menu

//

IDR\_MAINFRAME MENU PRELOAD DISCARDABLE

BEGIN

POPUP "&amp;File"

BEGIN

```
MENUITEM "&Open...", ID_FILE_OPEN
```

MENU ITEM SEPARATOR

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

END

POPUP "&amp;Format"

BEGIN

```
MENUITEM "&Font...", ID_FORMAT_FONT
```

MENU ITEM SEPARATOR

```
MENUITEM "&Expand Tabs", ID_FORMAT_TABS
```

END

POPUP "&amp;Help"

BEGIN

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

END

END

////////////////////////////////////

//

```
// Dialog
```

//



```
IDD_ABOUTBOX DIALOG DISCARDABLE 34, 22, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About filelist"
FONT 8, "MS Sans Serif"
BEGIN
LTEXT "filelist Version 1.0", IDC_STATIC, 40, 10, 119, 8
LTEXT "Copyright ", IDC_STATIC, 40, 25, 119, 8
DEFPUSHBUTTON "OK", IDOK, 176, 6, 32, 14, WS_GROUP
CONTROL "", IDC_STATIC, "Static", SS_BLACKFRAME, 8, 8, 22, 19
ICON IDR_MAINFRAME, IDC_STATIC, 8, 8, 21, 20
END

#ifdef _MAC
////////////////////////////////////
//
// Version
//
VS_VERSION_INFO VERSIONINFO
FILEVERSION 1,0,0,1
PRODUCTVERSION 1,0,0,1
FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
FILEFLAGS 0x1L
#else
FILEFLAGS 0x0L
#endif
FILEOS 0x4L
```

FILETYPE 0x1L

FILESUBTYPE 0x0L

BEGIN

BLOCK "StringFileInfo"

BEGIN

BLOCK "040904b0"

BEGIN

```
VALUE "CompanyName", "\0"
```

```
VALUE "FileDescription", "FILELIST MFC Application\0"
```

```
VALUE "FileVersion", "1, 0, 0, 1\0"
```

```
VALUE "InternalName", "FILELIST\0"
```

```
VALUE "LegalCopyright", "Copyright \0"
```

```
VALUE "OriginalFilename", "FILELIST.EXE\0"
```

```
VALUE "ProductName", "FILELIST Application\0"
```

```
VALUE "ProductVersion", "1, 0, 0, 1\0"
```

END

END

BLOCK "VarFileInfo"

BEGIN

VALUE "Translation", 0x409, 1200

END

END

```
#endif // !_MAC
```

////////////////////////////////////

//

```
// String Table
```

```

//

STRINGTABLE PRELOAD DISCARDABLE

BEGIN

IDR_MAINFRAME "FileList\n\nFileli\n\n\nFilelist.Document\nFileli Document"

END

STRINGTABLE PRELOAD DISCARDABLE

BEGIN

AFX_IDS_APP_TITLE "FileList"

END

#endif // Chinese (P.R.C.) resources

////////////////////////////////////

#ifndef APSTUDIO_INVOKED

////////////////////////////////////

//

// Generated from the TEXTINCLUDE 3 resource.

//

#include "res\\filelist.rc2" // non-Microsoft Visual C++ edited resources

#define _AFX_NO_SPLITTER_RESOURCES

#define _AFX_NO_OLE_RESOURCES

#define _AFX_NO_TRACKER_RESOURCES

#define _AFX_NO_PROPERTY_RESOURCES

#include "afxres.rc" // Standard components

////////////////////////////////////

#endif // not APSTUDIO_INVOKED

//{{NO_DEPENDENCIES}}

// Microsoft Visual C++ generated include file.

```

```

// Used by filelist.rc

//

#define IDD_ABOUTBOX 100

#define IDR_MAINFRAME 128

#define ID_FORMAT_FONT 32771

#define ID_FORMAT_TABS 32772

// Next default values for new objects

//

#ifdef APSTUDIO_INVOKED

#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_3D_CONTROLS 1

#define _APS_NEXT_RESOURCE_VALUE 130

#define _APS_NEXT_COMMAND_VALUE 32773

#define _APS_NEXT_CONTROL_VALUE 1000

#define _APS_NEXT_SYMED_VALUE 101

#endif

#endif

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#include <afxwin.h> // MFC core and standard components

#include <afxext.h> // MFC extensions

// stdafx.cpp : source file that includes just the standard includes
// filelist.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

```

```
#include "stdafx.h"
```

## 第五节 视类

在本章的前面部分，我们已经涉及到了一些视类及其派生类的用法，在本章的后面几节中，我们还结合例程讲解了一些常用的类，由此，在本节中，我们计划仅就各类作一简明的介绍，至于各类的具体应用，我们认为，通过例程来了解，熟悉各类的使用，远较简单地讲解原理更容易掌握。但我们也需要提醒读者的是，由于我们的篇幅原因，我们的讲解只可能涉及其中的最主要的部分，至于读者如果希望更深入地对系统的体系结构作了解的话，参考系统的随机帮助文件是一个相当重要，也相当方便的途径。

在这一节中，我们会按照各类在MFC中的层次讲解，由于面向对象语言的自身特性，父类所具有的特征，其派生出来的子类一般也具有该特性，因此，掌握各类的派生关系，往往是深刻理解其行为的一个捷径。同时，我们也应该看到，由于各类与其父类的差别，它们又具有各式各样的自身的特性，而把握这些特性，又成为进一步理解其独特特征的必由之路。下面，我们分类简单地讲解一下视类及各视类派生类。

首先，我们看看处于这些派生类基类位置的CView类在整个系统类体系中的位置。

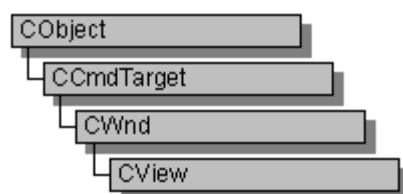


图8.17 类CView的继承示意图

由图8.17我们可以看出，类CWnd为类CView的直接基类。从这里，我们至少可以得到的一个最基本的概念就是，视类的对象具有窗口的一些基本特性。视类CView是一个用户定义视类的常用基类。一个视类与一个文档相连接，在文档与用户之间起了一个桥梁的作用。视类，文档，窗口之间的关系有对象CDocTemplate刻画。当用户新开一个窗口或将一个窗口进行分割时，程序框架就构造一个与其文档相联系的视类。一个视只能对应于一个文档，但一个文档可以拥有多个视，因此，就使一个文档，多个视图的程序可以轻易地实现。在本章的稍后几节中，读者会发现几个这方面的例程，读者可以看看它们的实现是多么的轻松。视的主要功能在于显示和修改文档的数据，但它对文档

数据的存储没有什么支持。

一个视类可以直接地接受文档的数据，也可以通过成员函数的调用来实现。

当一个类的数据发生变化时，该文档所关联的视类通常通过调用函数 `CDocument::UpdateAllViews` 来作出响应。这个函数是维护数据正确显示的常用手段。

视类 `CView` 具有为数不多的成员函数。但这些成员函数中的很多个是有着重要的意义的。函数 `OnBeginPrinting` 初始化打印，`OnEndPrinting` 结束一个打印任务，而 `OnEndPrintPreview` 则更是关于打印预览操作的一个重要函数，而函数 `OnPreparePrinting` 则负责了对打印以及打印预览的全面支持。

另外，视类中的两个成员函数 `OnDraw`（该函数在屏幕发生变化或因为焦点的变易需要重绘时调用，没有该函数，就不可能在程序的切换后保证屏幕的正确显示），`OnUpdate` 则对维护程序的正确显示负有重要的责任。（当一个文档发生变化时，绝大多数情况下总是希望其各个视图中的数据能得到同步的正确显示的。）函数 `UpdateAllViews` 则是实现单文档多视图程序不可缺少的手段。（在一个文档的任一视发生变化时，通过该类实现各视图的正确显示。）

接下来我们看看视类各派生类的继承及成员函数情况。但是，在这里，我们不再准备对各派生类作详细介绍。但是，我们会尽量简捷地告诉读者，怎样从系统的庞大帮助数据库中，查到自己所需要的信息。

### (1) 类 `CEditView`

其继承关系如图8.18。注意，该类的直接基类不是 `CView` 类，而是类 `CCtrlView`。

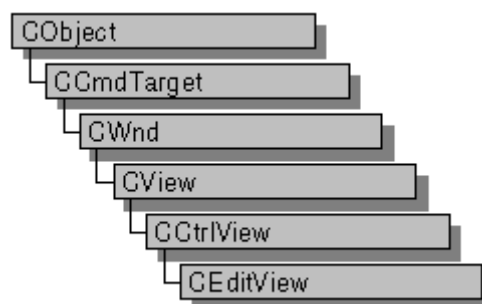


图8.18 类 `CEditView` 的继承图解

类CEditView主要被设计来支持类似编辑控件所要实现的功能，通过打印，查找/替换的支持。它们拥有自己的内存，可以在程序中被任意正确地使用。我们常见的文本操作，基本上都是由该类支持实现的。

## (2) 类CRichEditView

类CRichEditView的继承层次如图8.19。

该类主要提供Rich文本操作的支持（Rich文本是既可以为文本，也可以为图形的一种特殊格式文本。同时，在很多场合下，它们也充当热字的角色。）

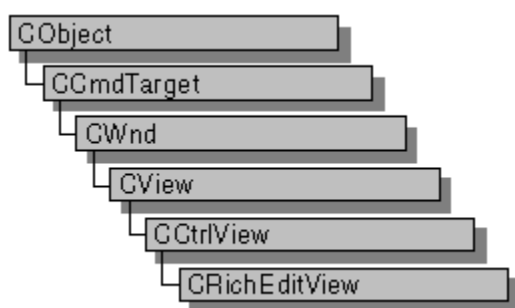


图8.19 类CRichEditView的继承图解

## (3) 类CTreeView

类CTreeView的继承层次如图8.20。

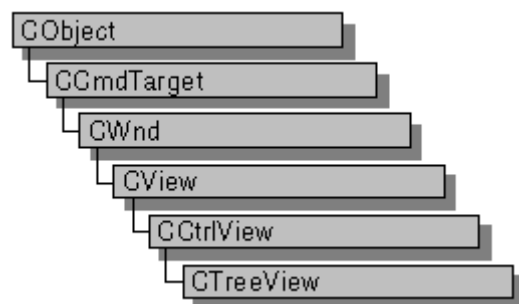


图8.20 类CTreeView的继承图解

该类主要提供一些树型控件所实现的功能的支持。它使一种数据的显示方式可以更富于变化。

## (4) 类ListView

类CListView的继承层次如图8.21。

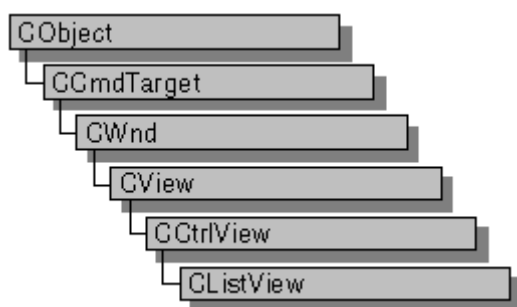


图8.21 类CListView的继承图解

该类与类CTreeView一样，更多的好处在于提供了一种简捷地实现数据的不同显示的途径。为数据的组织提供多种手段。

### (5) 类CScrollView

类CScrollView的继承层次如图8.22。

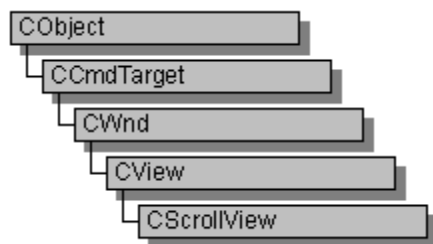


图8.22 类CScrollView的继承图解

该类也是一个比较重要的类，但它主要提供视图的滚动显示。同时，需要注意的是，该类的直接基类是类CView，这决定了其动作特点的特殊性。具体细节请参考系统帮助。

在本节的最后，我们简要地谈一下怎样在系统庞大的帮助文件中快速准确地找到自己所需要的主题。

Visual C++提供的帮助支持数据库不可谓不大，但往往在这个庞大的支持面前，我们多少感到一些无从下手。下面，我们结合程序设计时最可能碰到的一些关于类及函数的查询。

如图8.23，我们以查询有关打印支持操作的类CPrintDialog的情况的查询。首先，工作区键入CPrintDialog然后按F1，系统搜索其数据库后给出了帮助信息。单击之即可进入具体的帮助主题了。



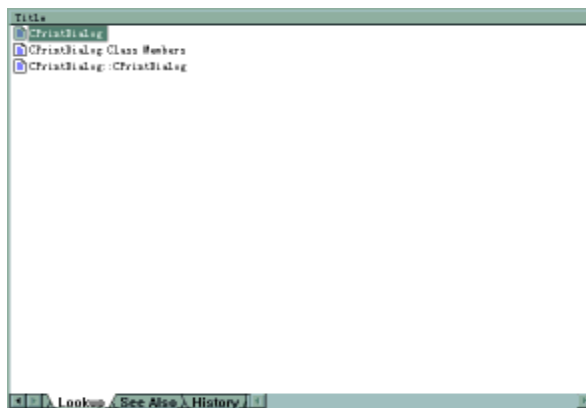


图8.23 查询结果显示

## 第六节 同一文档的多个视

启动Visual C++下的App Wizard 可以生成多文档程序，这只需要在第一步中选择MDI应用程序类型即可。从生成的MDI程序中，我们可以发现MDI程序与SDI程序的一个最主要的区别就是主窗口是由基类CMDIFrameWnd而不象在SDI程序中由CFrameWnd派生出来的。

```
class CMainFrame : public CMDIFrameWnd
{
    DECLARE_DYNAMIC(CMainFrame)

public:
    CMainFrame();
    // ...
}
```

在最初的文档模板中（文档被打开时被激活）只支持主窗口。这可以由下面的一段初始化代码中看出(摘自CMDIApp.cpp)。

```
BOOL CMdiApp::InitInstance()
{
    AfxEnableControlContainer();

    // 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
```

```

// ...

CMultiDocTemplate* pDocTemplate;

pDocTemplate = new CMultiDocTemplate(

    IDR_MDITYPE,

    RUNTIME_CLASS(CMdiDoc),

    RUNTIME_CLASS(CChildFrame), // custom MDI child frame

    RUNTIME_CLASS(CMdiView));

AddDocTemplate(pDocTemplate);

// ...

return TRUE;

}

```

每次打开一个新文档时都调用CDocument的函数OnNewDocument。该函数被定义如下：

```

class CMdiDoc : public CDocument

{

// ...

// Overrides

// ClassWizard generated virtual function overrides

//{{AFX_VIRTUAL(CMdiDoc)

public:

virtual BOOL OnNewDocument();

virtual void Serialize(CArchive& ar);

//}}AFX_VIRTUAL

// ...

}

```

当调用此函数时可以得到一个新的MDI子窗口，它由CDMIDChildWnd派生出来。这些子窗口保存着各种已打开的文档，所有的细节都是由

MFC库处理。而每个MDI子窗口都支持由类CView派生出的普通类型的视图。这意味着可以同以前一样不受约束地使用WM\_CHAR和WM\_MOUSEMOVE这类消息。

- 注意：
- 如果读者对这一部分的理解不太清楚，建议再看看6.5视类一节中有关视类层次结构的介绍。

在8.4生成视一节中的最后，我们曾经提到过，可以用调用函数UpdateAllViews来维护所需要的众多视图的同时协调工作。当任意一个视图修改了文档之后就可调用该函数，这样可确保除了第一个调用该函数进行修改的视图外所有视图均调用函数OnUpdate。在前面的那个例子中，我们就如上所述的调用了该类。

函数UpdateAllViews的原型为：

```
void UpdateAllViews( CView* pSender, LPARAM lHint = 0L, CObject* pHint = NULL );
```

其中LONG型参数lHint通常所包含的信息与视图对文档所做的修改有关；pHint总是指向一个对象，该对象有助于处理更新。为简单起见，只需将这些值赋为NULL。

但另一方面，我们还需要将函数OnUpdate加到视图类中，以处理由其它视图引发的更新，因此需要处理函数CView::OnUpdate：

```
virtual void OnUpdate( CView* pSender, LPARAM lHint, CObject* pHint );
```

其中参数pSender为一指向修改了文档的指针（当所有文档均需要更新时可以设置该参数为NULL），参数lHint内存放着有关这些修改的信息，函数最后一个参数pHint则为一存放该修改信息的内容的指针。

这样，当任意一个视图修改了文档时，它就调用pDoc->UpdateAllViews，该函数又调用与该文档相关联的其它所有视图中的OnUpdate来更新所有显示。由此，在系统中的所有视图就可以一起协调工作了。当在任意一个视图中键入任何内容时，在其它视图中也可以见到这些内容被显示出来。

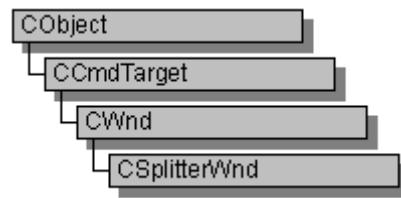


图8.24 类CSplitterWnd的层次

在程序中处理多文档，但同时，我们也可以通过分离窗口支持一个文档的多个视。分离窗口是这样一种窗口，它们在一个视窗中被分成几个相关联的部分，虽然它们在视窗中是分离的，但它们同样协同工作，任一窗口中所做的修改同样地反应到其它窗口中。

MFC中支持分离窗口的类是CSplitterWnd，该类的层次结构如图8.24。

因为CSplitterWnd类由CWnd类派生而来，它就可以使用它被授权使用的那些CWnd类的成员函数。

```

class CMdiSplitFrm : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CMdiSplitFrm)

protected:
    CMdiSplitFrm(); // protected constructor used by dynamic creation

    // add and remove member functions here.

    // ...

    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    DECLARE_INTERFACE_MAP()
}
  
```

- 注意：

- 在这里，涉及到一个创建新类的问题。如图，打开ClassWizard，选择Add Class..然后，我们键入想要生成类CMdiSplitFrm。并在Base class的下拉列表中选择splitter作为该类的基类。如图8.25所示。

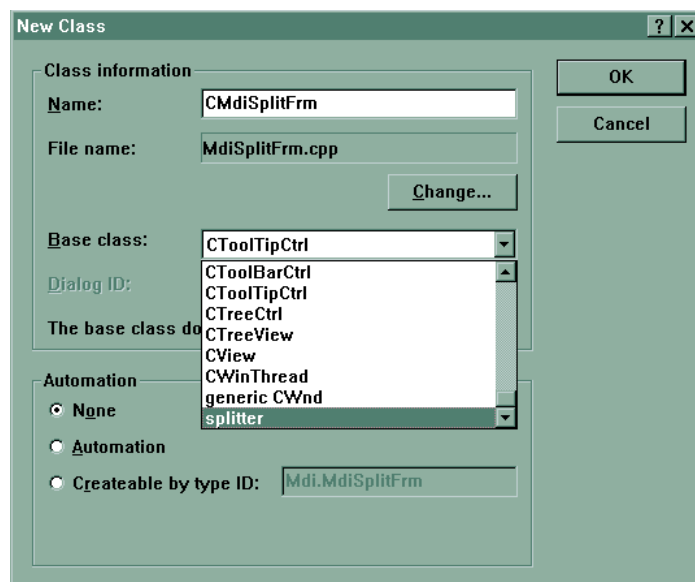


图8.25 利用Class Wizard加入一个新类

我们在下面的程序中使用了上面介绍的一些概念，我们的程序也可以良好的运行，但仍有许多工作要做。我们的程序使用前面提到的方法生成了一个支持窗口分割的MDI，但是，在参考我们的程序建立起你自己的应用程序之后，我们想，你一定会发现，我们的程序离一个好用的编辑器仍有很长的距离。不过，通过我们的逐步完善，我们完全可以将它做得更好。

下面我们给出一个程序运行的画面（如图8.26）。同时，我们也给出一部分关键的源代码，在需要自己添加的部分，我们会以具有底纹的文本标出，希望读者注意这些地方。

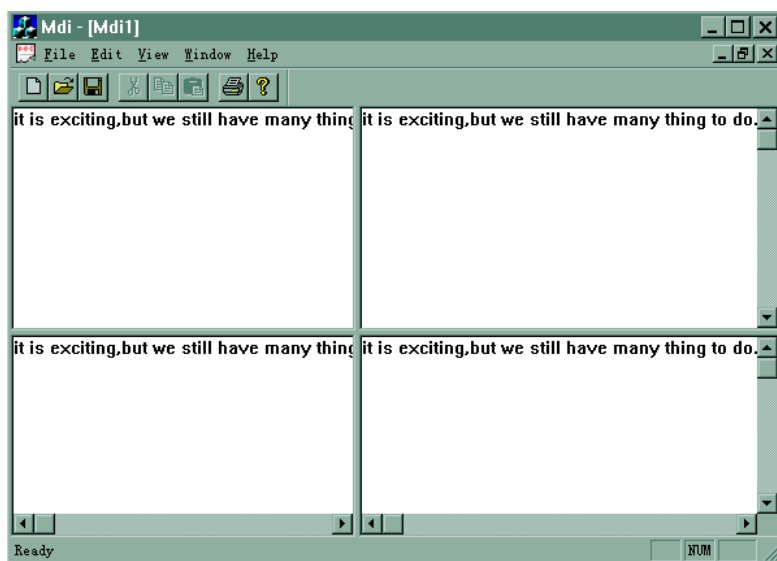


图8.26 分割窗口的一个运行画面

```
// MdiDoc.h : interface of the CMdiDoc class
```

```

#if !defined(AFX_MDIDOC_H__6A6E4F01_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)

// ...

class CMdiDoc : public CDocument
{
protected: // create from serialization only
CMdiDoc();

DECLARE_DYNCREATE(CMdiDoc)

CSize m_sizeDoc;

// Attributes

public:

CString data_string[100];

long line_number;

// Operations

public:

CSize GetDocSize(){return m_sizeDoc;}

// Overrides

// ...

protected:

protected:

//{{AFX_MSG(CMdiDoc)

// ...

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

// ...

#endif // !defined(AFX_MDIDOC_H__6A6E4F01_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)

```

```

// MdiDoc.cpp : implementation of the CMdiDoc class

#include "stdafx.h"

// CMdiDoc

// ...

IMPLEMENT_DYNCREATE(CMdiDoc, CDocument)

BEGIN_MESSAGE_MAP(CMdiDoc, CDocument)

//{{AFX_MSG_MAP(CMdiDoc)

// ...

//}}AFX_MSG_MAP

END_MESSAGE_MAP()

// CMdiDoc construction/destruction

CMdiDoc::CMdiDoc()

{
// TODO: add one-time construction code here

line_number=0;

m_sizeDoc=CSize(800,1000);

}

// ...

BOOL CMdiDoc::OnNewDocument()

{
if (!CDocument::OnNewDocument())

return FALSE;

// ...

return TRUE;

}

```

```

// CMdiDoc serialization

void CMdiDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar<<line_number;
        for(int loop_index=0;loop_index<line_number;loop_index++)
        {
            ar<<data_string[loop_index];
        }
    }
    else
    {
        ar>>line_number;
        for(int loop_index=0;loop_index<line_number;loop_index++)
        {
            ar>>data_string[loop_index];
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// CMdiDoc diagnostics

#endif // _DEBUG

// ...//debug function

// CMdiDoc commands

// MdiView.h : interface of the CMdiView class

```



```

#if !defined(AFX_MDVIEW_H__6A6E4F03_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)

// ...

class CMdiView : public CView
{
protected: // create from serialization only

// Operations

// ...

public:

// Overrides

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CMdiView)

public:

virtual void OnDraw(CDC* pDC); // overridden to draw this view

virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

virtual void OnInitialUpdate();

protected:

virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);

virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);

virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);

//}}AFX_VIRTUAL

// Implementation

public:

virtual ~CMdiView();

// ...

protected:

```

```

// Generated message map functions

protected:

//{{AFX_MSG(CMdiView)

afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

// ...

#endif // !defined
(AFX_MDIVIEW_H__6A6E4F03_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)

// MdiView.cpp : implementation of the CMdiView class

#include "stdafx.h"

// CMdiView

// ...

IMPLEMENT_DYNCREATE(CMdiView, CView)

BEGIN_MESSAGE_MAP(CMdiView, CView)

//{{AFX_MSG_MAP(CMdiView)

ON_WM_CHAR()

//}}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()

// ...

// CMdiView drawing

```

```

void CMdiView::OnDraw(CDC* pDC)
{
    CMdiDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    TEXTMETRIC tm;
    pDC->GetTextMetrics(&tm);
    int yval=0;
    for(int loop_index=0;loop_index<=pDoc->line_number;loop_index++)
    {
        pDC->TextOut(0,yval,pDoc->data_string[loop_index],
        pDoc->data_string[loop_index].GetLength());
        yval+=tm.tmHeight;
    }
    // TODO: add draw code for native data here
}

/////////////////////////////////////////////////////////////////

// CMdiView printing
// CMdiView diagnostics
// ...

#ifdef _DEBUG
void CMdiView::AssertValid() const
{
    CView::AssertValid();
}
// ...

// CMdiView message handlers

```

```

void CMdiView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default

    CMdiDoc *pDoc=GetDocument();

    CClientDC dc(this);

    OnPrepareDC(&dc);

    if(nChar=='\r'){
        pDoc->line_number++;
    }

    else{
        pDoc->data_string[pDoc->line_number]+=nChar;

        TEXTMETRIC tm;

        dc.GetTextMetrics(&tm);

        dc.TextOut(0,(int)pDoc->line_number*tm.tmHeight,
        pDoc->data_string[pDoc->line_number],
        pDoc->data_string[pDoc->line_number].GetLength());
    }

    pDoc->UpdateAllViews(this,0l,NULL);

    pDoc->SetModifiedFlag();

    CView::OnChar(nChar, nRepCnt, nFlags);
}

void CMdiView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // TODO: Add your specialized code here and/or call the base class

    Invalidate();
}

```

```

void CMdiView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    // TODO: Add your specialized code here and/or call the base class
}

// Mdi.h : main header file for the MDI application

#if !defined(AFX_MDI_H__6A6E4EF9_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)
// ...

////////////////////////////////////

// CMdiApp:

// See Mdi.cpp for the implementation of this class
//

class CMdiApp : public CWinApp
{
public:
    CMdiApp();

    // Overrides

    // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CMdiApp)

public:
    virtual BOOL InitInstance();

    }}AFX_VIRTUAL

    // ...

    DECLARE_MESSAGE_MAP()

};

// ...

```

```
#endif // !defined(AFX_MDI_H__6A6E4EF9_1FC8_11D2_BC8B_E95B8191F13C__INCLUDED_)

// Mdi.cpp : Defines the class behaviors for the application.

#include "stdafx.h"

#include "Mdi.h"

#include "MdiSplitFrm.h"

#include "MainFrm.h"

// ...

// CMdiApp

// ...

// CMdiApp construction

// ...

// The one and only CMdiApp object

CMdiApp theApp;

// CMdiApp initialization

// ...

CMultiDocTemplate* pDocTemplate;

pDocTemplate = new CMultiDocTemplate(

    IDR_MDITYPE,

    RUNTIME_CLASS(CMdiDoc),

    RUNTIME_CLASS(CMdiSplitFrm), // custom MDI splitter frame

    RUNTIME_CLASS(CMdiView));

AddDocTemplate(pDocTemplate);

// ...

return TRUE;

}
```

```
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
// ...
};

// ...

////////////////////////////////////

// CMdiApp commands

// 尽管类CSplitter是我们这节的重点，但由于在该类中我们并没有添加什么新的东西，而
// 有关该类的基础知识我们在讲解中已经作了解释，此处不再列出。
```

## 第七节 添加对多文档类型的支持

在本章前面的部分，我们已经比较详细地对多文档类型支持的一些问题。在本节中，我们只计划再简单地作一回顾，在本节的最后部分，我们则简单地谈谈程序对打印的支持手段。

在程序中加入多文档的支持，从技术上来说，只是需要将窗口的直接基类从单文档界面的CFrameWnd换成多文档程序的CMDIFrameWnd就可以了。但是，由于多文档程序的特殊性，也出现了一些需要不同处理的问题。对每个文档来说，它们所关联的视类（或其派生类）总是仅仅对应于唯一的文档，因此，对各个文档的数据的管理实际上与单文档程序没有什么区别，但是，由于多个文档的出现，对它们的管理就成为主要的问题。

我们不妨先将问题简化。每一个文档可以有多个视，但每个视只能对应于一个确定的文档。因此，多文档程序需要解决的问题仅仅是多个文档的数据管理方法的问题（对视图的管理我们在前面一节中已经作出了比较详细的解释）。对多文档程序来说，最初的文档模板只支持主窗口，但每次打开一个新文档时都调用CDocument的函数OnNewDocument，建立一个有CMDIChildWnd派生的新的MDI子窗口，这些窗口中保存着各种已打开的文档，所有的细节都由MFC库处理。例如，当用户单击File菜单下的New菜单项时，将生成另外一个文档并打开其窗口。但是，每个文档中初始时都是一片空白，因此还需加上

能对不同的窗口事件作出反应的代码。每个MDI子窗口都支持由类CView派生的普通类型的视窗，这就意味着可以同以前一样不受约束地使用各类消息。而对各类消息的单独处理，读者不妨参考对应各章的讲解。

作为文档/视结构的一个组成部分，在本章我们并没有对打印和打印预览这一部分内容进行讲述。由于篇幅的有限，在本书中我们将不单独讲述如何添加和完善应用程序的打印支持。这里仅指出一点：在我们建立非基于对话框的程序时，如果我们选择了打印支持，那么，系统已经为我们完成了绝大部分的工作。但如果我们没有在最初的设计中选择打印支持，那么，我们就需要考虑对它自己实现了。MFC提供了标准打印对话框CPrintDialog来简化打印操作。读者如果有这方面的需要，请参考系统的帮助文件。



## 第九章 图形设备接口

事实上，图形设备接口(Graphics Device Interface, GDI)是指这样的一个可执行程序，它处理来自Windows应用程序的图形函数调用，然后把这些调用传递给合适的设备驱动程序，由设备驱动程序来执行与硬件相关的函数并产生最后的输出结果。GDI可以看作是一个应用程序与输出设备之间的中介，一方面，GDI向应用程序提供了一个设备无关的编程环境，另一方面，它又以设备相关的格式和具体的设备打交道。

经常同图形设备接口相提并论的另一个概念是设备上下文(Device Context, DC)。设备上下文是一种Windows数据结构，它包括了与一个设备(如显示器或打印机)的绘制属性相关的信息。所有的绘制操作通过一个设备上下文对象进行，该对象封装了实现绘制线条、形状和文本的Windows API函数。设备上下文可以用来向屏幕、打印机和图元文件输出结果。

在Windows应用程序中，我们通常在绘制之前调用BeginPaint函数，然后在设备上下文中进行一系列的绘制操作，最后调用EndPaint函数结束绘制。MFC类CPaintDC封装了这一过程。在构造CPaintDC对象的同时，其构造函数自动调用BeginPaint函数；在销毁CPaintDC对象的同时，其析构函数自动调用EndPaint函数。因此前面所讲述的过程可以对应于下面的三个步骤：构造一个CDC对象，进行绘制操作，销毁该CDC对象。在基于文档/视结构的应用程序框架中，这个过程被进一步的简化。回忆前几章中讲述的内容，我们一般在视类的OnDraw成员函数中处理有关重绘的操作。通过OnPrepareDC成员函数，框架自动的向OnDraw成员函数传递一个类型为CPaintDC的设备上下文对象。我们只需简单的通过该对象进行绘制，而不需要关心这一对象的构造和销毁。这一过程由框架自动的完成，而且，隐藏在背后的设备上下文在对OnDraw的调用返回时由框架进行释放。

除了上面的CPaintDC类外，MFC还提供了其它的一些封装不同设备上下文的类。如CClientDC类，它所封装的设备上下文仅代表了一个窗口的客户区。在CClientDC的构造函数中调用的不是BeginPaint函数，而是GetDC函数；相应的，ReleaseDC函数在类CClientDC的析构函数被自动调用。与此对应的还有另一个类CWindowDC，它所封装的设备上下文代表的是整个窗口，不仅包括其客户区，也同时包括窗口的边框及其它非客户区对象。

所有的设备上下文类中比较特殊的是类CMetaFileDC，通过

CMetaFileDC对象所进行的绘制操作不是对一个实在的设备来进行的，这些操作都被记录到一个Windows图元文件中。不象自动传递给OnDraw成员函数的CPaintDC对象，如果要在这种情况下使用CMetaFileDC对象的话，我们必须自己调用OnPrepareDC成员函数。

所有的这些设备上下文类都以类CDC作为其基类。

一般情况下，很多绘制操作都是在应用程序的视类的OnDraw成员函数中进行的，前面说到过，当视类窗口收到消息WM\_PAINT时，该消息对应的处理函数OnPaint被调用，该处理函数构造一个CPaintDC对象，并将指向该对象的指针传递给OnDraw成员函数。这里我们考虑这样一种情况，如果我们正在编写的是一个通过鼠标在屏幕上绘图的应用程序。这时我很显然需要为鼠标的移动消息添加消息处理函数，而且，我们希望用户在移动鼠标的过程中立即就可以看到所绘制的内容，而不是等到窗口收到WM\_PAINT消息(即发生重绘事件)才调用成员函数OnDraw绘制窗口。在这种情况下，我们更倾向于直接在鼠标消息处理函数中进行绘制，这时，就需要创建一个设备上下文对象，然后通过该对象调用一系列的绘制方法。

Windows本身是一个图形界面的操作系统，进行Windows程序设计随时都会同设备上下文打交道，甚至在本书前面的章节中的一些示例程序中我们也已经用到了设备上下文，只不过在当时我们回避了与设备上下文有关的很多复杂东西。本章的目的之一就是系统的讨论这些前面已经用到但没有加以阐述的概念和技巧，并补充一些尚未涉及的内容，这些内容包括：

- 使用设备上下文进行绘制
- 绘图对象
- 直线与曲线
- 填充形状
- 字体和文本
- 颜色
- 坐标空间及变换

这些概念往往是交织起来的，哪怕是一个很简单的绘制操作，往往都需要用到不只一个绘图对象，因此我们很难将它们人为的分割开来进

行讲述。在本章上，各节的标题只代表了本节的侧重点，而对于某一个概念的叙述或使用，则有可能分散在不只一个小节中。事实上，一个应用程序是一个整体，它常常需要很多个部件共同协调工作才可以正常工作。因此，出现这种情况是很自然的。

### 第一节 设备上下文

在MFC应用程序中，绘制操作通常涉及三类对象，一类是输出对象，亦即设备上下文对象，包括CDC及其派生类；一类是绘制工具对象，亦即前面所说的图形对象，如果CFont、CBrush和CPen等；另一类属于Windows编程中需要用到的基本数据类型，如CPoint、CSize和CRect等。

不同的设备上下文类封装了不同类型的设备上下文类对象，如表9.1所示。

除了设备上下文以外，在Windows中进行绘制通常还需要各种绘制工具，如用来绘制线条的笔、用来填充一个图形内部的刷子以及用来绘制文本的字体等。这些工具称作图形对象，它们由Windows系统提供，MFC提供的图形对象类对它们进行了封装，表9.2给出了这些图形对象以及与它们等价的Windows图形设备句柄类型。

表9. 1 MFC中的设备上下文类

设备上下文类	描述
CDC	所有设备上下文类的基类。可用来直接访问整个显示器或如打印机之类的非显示设备上下文。
CPaintDC	在窗口的OnPaint成员函数中使用的一种显示上下文。在其构造过程中自动调用BeginPaint，在其析构过程中自动调用EndPaint。
CClientDC	代表窗口的客户区的显示上下文。通常在需要直接在窗口客户区进行绘制时使用。
CWindowDC	代表整个窗口的显示上下文，包括客户区和非客户区。
CMetaFileDC	代表Windows图元文件的设备上下文。一个Windows图元文件包括一系列的图形设备接口命令，可以通过重放这些命令来创建图形。向CMetaFileDC对象进行的各种绘制操作可以被记录到一个图元文件中。

表9. 2 MFC中的Windows GDI对象类

图形对象类	等价的Windows图形设备句柄	描述
CBrush	HBRUSH	用来填充正在绘制的对象的内部
CPen	HPEN	用来绘制对象的边线
CFont	HFONT	用来绘制文本
CBitmap	HBITMAP	用来提供操作位图的接口
CPalette	HPALETTE	用作应用程序和色彩输出设备(如显示器)之间的接口

### 9.1.1 几个与图形绘制有关的简单数据类型

在讲述设备上下文和图形对象之前，我们来介绍几个常用的数据结构类。

#### (1) CPoint类

CPoint类封装了一个点的坐标。它事实上是从POINT结构派生而来的。结构POINT在Win32 SDK中定义。因此，CPoint也继承了POINT结构的数据成员x和y。CPoint对象可以用在任何使用POINT结构的场合。CPoint对象还可以和另一种简单数据类型CSize或SIZE结构相互进行转换。

CPoint类具有多种形式的构造函数：

```
CPoint( );
CPoint( int initX, int initY );
CPoint( POINT initPt );
CPoint( SIZE initSize );
CPoint( DWORD dwPoint );
```

当使用DWORD类型的值来构造CPoint对象时，其低位字将被赋值给

CPoint对象的成员x，高位字将被赋值给成员y。

CPoint的成员函数Offset可以设置点的偏移量，同时，在类中定义的一些运算符，如?、?、??和??等大大的简化了对点坐标的各种运算和比较。

## (2) CSize类

如果要表示距离以及相对位置，可以使用CSize对象。MFC类CSize事实上是从SIZE派生而来的，因此，CSize继承了SIZE结构的数据成员cx和cy。构造一个CSize对象与用对应的方法构造CPoint对象非常相似，因此我们不需讲述。同样，我们可以使用一个DWORD值来构造CSize对象，这时，其低位字被赋值给CSize对象的成员cx，高位字被赋值给成员cy。在类Size中定义了六个运算符：?、?、??、??、??和??。

## (3) CRect类

CRect类是编程时经常使用的几个简单数据结构之一，它从RECT结构派生，因此，CRect类继承了RECT结构的数据成员left、top、right和bottom。它们是CRect的公有成员。

一个CRect对象可以传递给任何以RECT结构或LPCRECT和LPRECTW指针为参数的函数。

- 注意：

- 在指定一个CRect对象时，一般情况下我们需要使它的左边界的坐标小于右边界的坐标和上边界的坐标小于下边界的坐标。我们称满足该条件的矩形为常态矩形。很多函数要求传递给它的CRect对象表示一个常态矩形，否则这些函数将有可能返回一个错误的结果。我们可以通过调用成员函数NormalizeRect来将一个非常态矩形转换为一个常态矩形。在程序中出现非常态矩形并不一定的程序员的疏忽大意。这里举一个例子，如果当前显示上下文的映射模式为MM\_LOENGLISH，将一个表示常态矩形的CRect对象传递给成员函数CDC::DPtoLP，将得到一个非常态矩形，该矩形的高度将成为一个负值。这是因为在MM\_LOENGLISH映射模式中，纵坐标的方向是向上的。

相比我们在前面所讲述的CPoint类和CSize类来说，类CRect要庞大得多。表列出了在类CRect中定义的成员函数。

表9. 3 类CRect的成员函数

成员函数	描述
Width	计算矩形的宽度
Height	计算矩形的高度
Size	计算矩形的大小

续表9.3

成员函数	描述
TopLeft	返回矩形的左上角
BottomRight	返回矩形的右下角
CenterPoint	返回矩形的中点
IsEmpty	判断矩形是否为空。空的矩形的宽和高都为0
IsNull	判断矩形的top、bottom、left和right成员变量是否全都为0
PtInRect	判断指定点是否在矩形内
SetRect	设置矩形的大小
SetRectEmpty	将矩形设置为空(所有坐标均为0)
CopyRect	从源矩形中拷贝维度到矩形中
EqualRect	判断两个矩形是否相等
InflateRect	扩大矩形的宽和高
DeflateRect	减小矩形的宽和高
NormalizeRect	使用矩形的宽和高标准化
OffsetRect	按指定的偏移量移动矩形
SubtractRect	从一个矩形中减去另一个矩形
IntersectRect	设置矩形为两个矩形的交
UnionRect	设置矩形为两个矩形的并
LPCRECT	转换CRect对象为LPCRECT

LPRECT	转换CRect对象为LPRECT
=	拷贝一个矩形的维度到CRect对象
==	判断两个矩形的维度是否相等
!=	判断两个矩形是否不等
+=	将指定的偏移量添加到CRect对象或扩展CRect对象
-=	从CRect对象中减去指定的偏移量或缩小CRect对象
&=	设置CRect对象为CRect对象和另一矩形的交
=	设置CRect对象为CRect对象和另一矩形的并
+	将指定的偏移量添加到CRect对象或扩展CRect对象，并返回一个CRect对象
-	从CRect对象减去指定的偏移量或缩小CRect对象，并返回一个CRect对象

续表9.3

成员函数	描述
&	返回CRect对象和另一矩形的共同部分
	返回CRect对象和另一矩形的并

### 9.1.2 显示设备上下文

对于在视类的OnDraw成员函数中使用设备上下文进行输出的这种情况，我们已经以前面讲述文档和视时给出了一些示例，因此这里就不再重复叙述，读者可以参考前面所讲述的内容。下面我们来看一下如何自己构造设备上下文，并通过该设备上下文来进行绘制。

在示例程序MulticlrCaption中，我们通过CWindowDC对象获得包括客户区和非客户区的显示设备上下文，然后将窗口的标题绘制为五彩的。

```
#include <afxwin.h>
```

```
#include <afxext.h>
```

```
// 派生自己的应用程序类
```

```

class CMyApp : public CWinApp
{
public:
virtual BOOL InitInstance();

};

// 应用程序主窗口类

class CMyWnd : public CFrameWnd
{
protected:
void PaintTitleBar(BOOL bActive);

// 声明主窗口的消息处理函数

afx_msg void OnNcPaint();
afx_msg BOOL OnNcActivate(BOOL bActive);
DECLARE_MESSAGE_MAP();

};

// 初始化应用程序的实例

BOOL CMyApp::InitInstance()
{
// 创建应用程序主窗口

CMyWnd *pWnd=new CMyWnd;

pWnd->Create(NULL, "具有五彩标题条的窗口");

// 显示应用程序主窗口，并更新客户区

pWnd->ShowWindow(SW_SHOW);

pWnd->UpdateWindow();

m_pMainWnd=pWnd;

return TRUE;

```



```

}

// 声明应用程序对象

CMyApp MyApp;

// 应用程序主窗口的消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CWnd)

ON_WM_NCPAINT()

ON_WM_NCACTIVATE()

END_MESSAGE_MAP()

// 绘制窗口的标题条，参数 bActive 代表窗口的当前激活状态

void CMyWnd::PaintTitleBar(BOOL bActive)

{

// 创建代表整个窗口的显示设备上下文对象

CWindowDC dc(this);

CRect rc;

// 获得窗口矩形及其宽度

GetWindowRect(rc);

UINT nWidth=rc.Width();

// 获得窗口边框的度量

UINT nXFrame=GetSystemMetrics(SM_CXSIZEFRAME);

UINT nYFrame=GetSystemMetrics(SM_CYSIZEFRAME);

// 获得窗口标题条的高度

UINT nYCaption=GetSystemMetrics(SM_CYCAPTION);

COLORREF cr;

if (bActive)

{

// 获得当窗口处于激活状态时其标题条的颜色

```

```

cr=GetSysColor(COLOR_ACTIVECAPTION);

// 按从红到绿，再到蓝的渐变规律绘制标题条

for (UINT j=nYFrame; j<=nYFrame+nYCaption; j++)

{
    for (UINT i=nXFrame; i<=nWidth/2; i++)

    {
        UINT nLen=nWidth/2-nXFrame+1;

        if (dc.GetPixel(i, j)==cr)

        {
            dc.SetPixelV(i, j,
                RGB(255-255*(i-nXFrame)/nLen, 255*(i-nXFrame)/nLen, 0));
        }
    }

    for (i=nWidth/2+1; i<=nWidth-nXFrame; i++)

    {
        UINT nLen=nWidth/2-nXFrame-1+1;

        if (dc.GetPixel(i, j)==cr)

        {
            dc.SetPixelV(i, j,
                RGB(0, 255-255*(i-nWidth/2-1)/nLen, 255*(i-nWidth/2-1)/nLen));
        }
    }
}

else

{

```

```

// 获得窗口处于非激活状态时的标题条颜色
cr=GetSysColor(COLOR_INACTIVECAPTION);

// 按从黑到灰，再到黑的渐变规律绘制标题条
for (UINT j=nYFrame; j<=nYFrame+nYCaption-1; j++)
{
    for (UINT i=nXFrame; i<=nWidth/2; i++)
    {
        UINT nLen=nWidth/2-nXFrame;
        if (dc.GetPixel(i, j)==cr)
        {
            dc.SetPixelV(i, j, RGB(192*(i-nXFrame)/nLen,
            192*(i-nXFrame)/nLen, 192*(i-nXFrame)/nLen));
        }
    }
    for (i=nWidth/2+1; i<=nWidth-nXFrame; i++)
    {
        UINT nLen=nWidth/2-nXFrame-1;
        if (dc.GetPixel(i, j)==cr)
        {
            dc.SetPixelV(i, j, RGB(192-192*(i-nWidth/2)/nLen,
            192-192*(i-nWidth/2)/nLen, 192-192*(i-nWidth/2)/nLen));
        }
    }
}
}
}
}
}
}

```

```
// 消息 WM_NCPAINT 的处理成员函数

void CMyWnd::OnNcPaint()

{
    CFrameWnd::OnNcPaint();

    // 检查当前窗口是否为激活窗口

    if (::GetActiveWindow()==GetSafeHwnd())

        PaintTitleBar(TRUE);

    else

        PaintTitleBar(FALSE);

}

// 消息 WM_NCACTIVATE 的处理成员函数

BOOL CMyWnd::OnNcActivate(BOOL bActive)

{
    CFrameWnd::OnNcActivate(bActive);

    // 根据不同的激活状态按不同的方式绘制窗口标题条

    if (bActive)

    {
        PaintTitleBar(TRUE);

        return FALSE;

    }

    else

    {
        PaintTitleBar(FALSE);

        return FALSE;

    }

}
```

下面补充说明一下应用程序MulticlrCaption的创建。由于该应用程序的结构比较简单，因此我们不打算使用AppWizard来创建框架应用程序。这里，我们先创建一个Win32 Application工程，然后添加一个C++ source file，在该源代码文件中输入上面的代码。这个过程已经在本书前面的章节中使用过，因此你应该能够很轻松的完成它。下面我们来分析这个应用程序。首先，我们在类CMyWnd中添加一个成员函数PaintTitleBar。该成员函数用来绘制窗口的标题条，其参数bActive给出了窗口的激活状态。如果当前窗口正处于激活状态，我们使用从红色到绿色再到蓝色的渐变颜色来绘制应用程序的标题条，如果当前窗口正处于非激活状态，我们使用从黑色到灰色再到黑色的渐变色来绘制标题条。

由于我们需要通过设备上下文在窗口的非客户区(这里指窗口的标题条区域)在进行绘制，所以我们选用了CWindowDC类。在类CWindowDC构造函数中自动调用了GetWindowDC函数，在其析构函数中自动调用了ReleaseDC函数。类CWindowDC的构造函数使用了一个指向CWnd对象的指针作为其参数，通过所创建的CWindowDC对象可以在窗口的非客户区进行图形输出。

在创建了类型为CWindowDC的设备上下文对象dc之后，我们调用API函数GetSystemMetrics来获得当前窗口的边框高度和宽度以及标题条的高度。这里我们指出一点，即这些度量值仅适用于具有常规样式的窗口，对于一些特殊的窗口可能不成立，如对于工具条窗口，其标题条高度要小得多。这是上面的应用程序的一个局限，但不可以对一个仅用来讲解CWindowDC对象的使用的示例过于苛求，否则我们就不得不花篇幅去介绍很多完善整个应用程序所需要的额外代码。这种对应用程序的简化的处理方法下面还会遇到。通过使用不同的参数调用GetSystemMetrics函数可以得到不同的系统度量。我们所使用的仅仅是这些度量值中的很少一部分。

如果窗口是处于激活状态，我们使用参数COLOR\_ACTIVECAPTION调用API函数GetSysColor得到当前系统颜色设置中激活状态条使用的颜色(我们不能假定用户的Windows窗口在激活时都使用标准的蓝色标题条，因此用户可以很方便的使用控制面板或通过右击桌面选择“属性”来更改这些设置)。然后，我们调用在类CWindowDC的基类CDC中定义的成员函数GetPixel来获得标题条上的每一点的颜色值，如果这一点的颜色值等于当前使用的激活标题条颜色的话，就调用函数SetPixelV将该点的颜色设置为新的值。这种方式不会不正确的擦除当前标题条上的标题文本、应用程序图标以及窗口右上角最大化、最小化和关闭按钮，但是，由于在新的Windows操作系统Windows 98以

及Windows NT 5.0中，标题条的颜色在默认情况下是渐变的，因此应用程序将不能正确工作。类CDC的成员函数GetPixel返回一个COLORREF值以指示位于指定坐标的点的颜色值。成员函数SetPixelV将位于指定坐标的点的颜色设置为新的值，另一个成员函数SetPixel可以完成同样的功能，并且更常用。但是，与成员函数SetPixel不同，SetPixelV不需要返回设置的实际颜色值，因此它要比SetPixel快。尽管如此，上面的程序仍只能作为一个示例程序出现，因为这种一个点一个点的描绘的方法实在是太慢，在作者的具有64M内存和K6/200的CPU的计算机上，更新一个常规大小的窗口的标题条的颜色需要大约0.3秒的时间，在这样长的时间段内，用户还是可以清楚的看到标题条一点一点绘制的过程。提高应用程序的绘制速度的一种方案是使用位图来内存中对位图进行变换和处理，然而再使用位图来更新标题条。由于位图的绘制速度要比一个点一个点的描快得多，从而有可能大幅度的提高标题条的重绘速度，但是这种算法要使用到我们在这里不打算深入讲述的一些概念和技巧，为了便于理解，我们还是采用了上面给出的算法。

下面要做的事是处理两个重要的非客户区消息，WM\_NCPAINT和WM\_NCACTIVATE。第一个消息WM\_NCPAINT在非客户区的全部或一部分需要重绘时由操作系统发送。如果试图给窗口绘制特殊的边框或标题条，处理这个消息是必要的。MFC在CWnd中定义了该消息的默认处理函数OnNcPaint，该成员函数对WM\_NCPAINT的默认处理绘制了窗口的正常边框。我们在类CMyWnd中重载了该成员函数。由于我们只是绘制了窗口的标题条，因此在此之前有必要调用一下基类中的默认实现绘制窗口的边框。然后，通过判断由API函数GetActiveWindow返回的窗口句柄(它代表了当前激活窗口)和当前窗口的句柄是否相等来以不同的参数调用PaintTitleBar成员函数来绘制窗口处于激活状态和非激活状态的标题条。

另一个必须考虑的事件是当窗口的激活状态发生改变时正确绘制窗口的标题条以反映窗口的新的激活状态。这时窗口将会收到WM\_NCACTIVATE消息，该消息所带的参数给出了窗口新的激活状态。MFC在CWnd中定义了该消息的默认处理函数OnNcActivate。我们在类CMyWnd中重载了该成员函数。该函数根据窗口新的激活状态调用了PaintTitleBar来绘制新的窗口标题条以反映激活状态的改变。

整个应用程序使用了典型的MFC的结构，代码也比较简单和清晰，这里我们就不多作的介绍了。你可以根据上面的源代码清单和本书前面章节中讲述的内容来完成该应用程序。

完成上面的步骤之后我们就可以编译并试运行该应用程序了。在编译

之前我们需要做一些额外的工作：

1. 单击Project菜单下的Settings命令或按下快捷键Alt+F7，打开如图9.1所示的工程设置对话框。

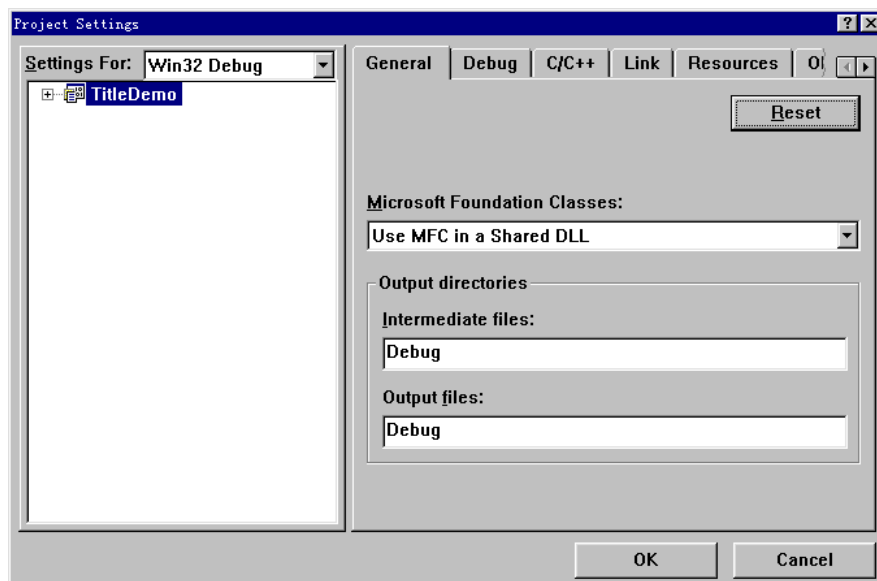


图9. 1 设置应用程序的工程属性

2. 在General选项卡中的Microsoft Foundation Classes下拉列表框中选择Use MFC in a Shared DLL或Use MFC in a Static Library (仅适用于Visual C++ 5.0的专业版和企业版)。你需要对应用程序的调试版本和发行版本各重复一次上面的设置过程。如果忽略此步设置的话，在链接应用程序的过程中会出现错误。

现在就可以编译并运行上面的程序了。其运行结果如图9.2所示。

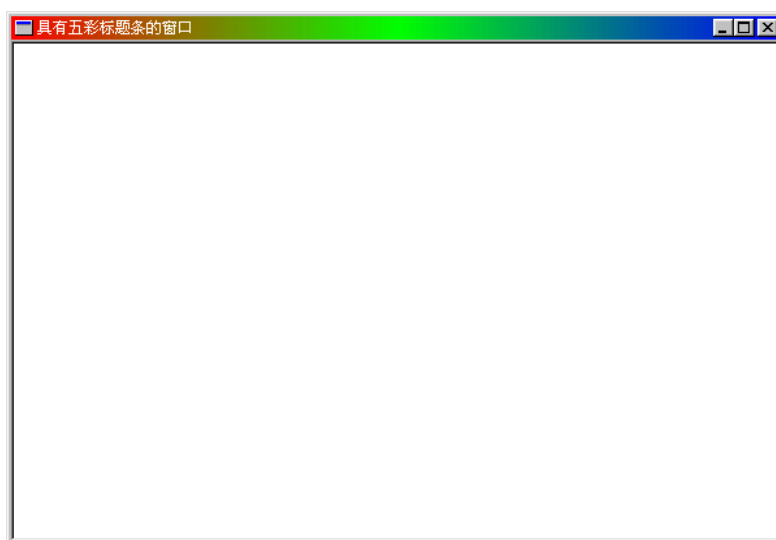


图9. 2 具有五彩标题条的窗口

上面的程序还有其它的一些局限性。比如当窗口处于非激活状态时，如果使用鼠标在窗口上移动其它窗口，窗口的标题条将会变成标准的灰色；还有，如果应用程序通过调用。要完善这些功能需要考虑更多的问题和处理更多的消息。这并不是本书在这里引入上面的示例程序的目的，我们只是为了演示一下CWindowDC对象的使用，而不是编写一个功能完善的应用程序。当然，你可以使用更好和更完善的方法来实现该应用程序并将它用于你的其它应用程序。一个特殊的标题条常常会给程序带来一些吸引人的东西，但是过分花哨的用户界面可能会使用户感到不适应甚至招至用户的反感。

使用CClientDC的应用程序与此类似，只不过我们通常在一些需要直接在窗口的客户区进行绘制的场合创建和使用该设置上下文对象。比如在一些使用鼠标绘图的应用程序中，当用户在客户区中单击鼠标时，我们通常需要直接在客户区中绘制出相应的图形，而不必等到WM\_PAINT消息发送。对于这样的应用程序，我们一般在鼠标的移动和单击事件的处理函数中创建类型为CClientDC的设置上下文对象，并通过该设备上下文对象进行绘制。如果使用了AppWizard来生成MFC应用程序的话，我们一般使用ClassWizard来完成添加这些消息处理成员函数和相应的消息映射项。

如前所述，在WM\_PAINT消息的处理函数中，我们一般不使用CClientDC对象，而应该使用CPaintDC对象。

在下面的小节中，我们将讲述MFC中的GDI绘图对象类的使用。

## 第二节 画笔对象

MFC类CPen封装了GDI中的画笔对象，画笔对象代表了进行绘制时所用的线条。我们一般通过两个步骤来创建画笔对象：首先构造一个CPen对象，再调用对象的CreatePen成员函数。成员函数CreatePen按指定的样式、宽度等属性创建一个逻辑画笔，然后将该画笔与CPen对象相关联。

### 9.2.1 创建画笔

成员函数CreatePen有两种形式。第一种形式的如下：

```
BOOL CreatePen( int nPenStyle, int nWidth, COLORREF crColor );
```

参数nPenStyle代表了画笔的样式，可以为下列值之一：



PS_SOLID :	创建一个实线画笔
PS_DASH :	创建一个虚线画笔。一个虚线画笔的宽度不能超过一个设备单位。
PS_DOT :	创建一个点线画笔。一个点划线画笔的宽度不能超过一个设备单位。
PS_DASHDOT :	创建一个点划线画笔。同样，这种样式的画笔宽度也不能超过一个设备单位。
PS_DASHDOTDOT :	创建一个双点划线画笔。这种样式的画笔宽度也不能超过一个设备单位
PS_NULL :	创建一个空画笔
PS_INSIDEFRAME :	对于那些指定一个边界矩形的GDI输出函数，具有这种样式的画笔将线条绘制到输出形状框架的内侧。而对于那些没有指定边界矩形的GDI输出函数，这种画笔的绘制区域则不受框架的限制。

参数nWidth以逻辑单位给出画笔的宽度。如果参数nWidth为零，则无论当前使用何种映射模式，所创建和画笔的宽度都为一个像素。参数crColor为画笔的颜色，这里可以使用RGB宏来生成合适的颜色值。

另一种形式的CreatePen函数使用如下的参数：

```
BOOL CreatePen( int nPenStyle, int nWidth, const LOGBRUSH* pLogBrush,
int nStyleCount = 0, const DWORD* lpStyle = NULL );
```

这种形式的CreatePen可以创建一个具有指定的宽度、样式和刷子属性的逻辑修饰(cosmetic)或几何(geometric)画笔。参数nPenStyle指定了画笔的样式，它可以为PS\_COSMETIC、PS\_GEOMETRIC或它们与一些附加属性的组合，详细的说明这里不进行说明，如果需要的话你可以参考联机文档中对构造函数CPen::CPen的说明。nWidth以逻辑单位指定画笔的宽度，如果nPenStyle参数包括了PS\_COSMETIC的话，参数nWidth必须为1。参数pLogBrush为指向一个LOGBRUSH结构的指针，该LOGBRUSH结构定义了画笔的刷子属性。最末两个参数nStyleCount和

lpStyle定义了画笔的每一划及它们之间的空白的长度。

画笔对象实际上也可以一步创建，这时所使用的构造函数也使用与函数CreatePen相一致的参数。以使用一步创建的方式创建画笔对象时，我们通过捕获一个异常是否发生来判断是否出错。

### 9.2.2 使用画笔在设备上下文中进行输出

一旦画笔对象创建成功之后，即可使用CDC类的成员函数SelectObject将其选入设备描述表中进行各种输出。

下面的示例程序PenDemo演示了画笔对象的使用。

创建工程PenDemo的方法同在9.1.2中引入的示例程序MulticlrCaption相同。您可以参照上一节的讲述来创建工程PenDemo。其代码清单如下：

```
#include <afxwin.h>

#include <afxext.h>

#include <time.h>

// 派生应用程序类

class CMyApp : public CWinApp

{

public:

virtual BOOL InitInstance();

};

// 派生窗口类

class CMyWnd : public CFrameWnd

{

protected:

// 声明消息处理函数

afx_msg void OnPaint();

DECLARE_MESSAGE_MAP();
```

```

};

// 初始化应用程序实例

BOOL CMyApp::InitInstance()

{
    // 创建应用程序的主窗口

    CMyWnd *pWnd=new CMyWnd;

    pWnd->Create(NULL, "各种画笔的示例");

    // 显示应用程序主窗口并刷新其客户区

    pWnd->ShowWindow(SW_SHOW);

    pWnd->UpdateWindow();

    // 在主窗口关闭时终止应用程序的执行线程

    m_pMainWnd=pWnd;

    return TRUE;
}

// 声明唯一的应用程序对象

CMyApp MyApp;

// 应用程序主窗口的消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CWnd)

    ON_WM_PAINT()

END_MESSAGE_MAP()

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{
    // 获得窗口的客户区设备上下文句柄

    CPaintDC dc(this);

    // 定义一个画笔数组

```

```
CPen pen[8];

// 创建实线画笔
pen[0].CreatePen(PS_SOLID, 10, RGB(255, 0, 0));

// 创建虚线画笔
pen[1].CreatePen(PS_DASH, 1, RGB(0, 255, 0));

// 创建点线画笔
pen[2].CreatePen(PS_DOT, 1, RGB(0, 0, 255));

// 创建点划线画笔
pen[3].CreatePen(PS_DASHDOT, 1, RGB(0, 255, 255));

// 创建双点划线画笔
pen[4].CreatePen(PS_DASHDOTDOT, 1, RGB(255, 0, 255));

// 创建空画笔
pen[5].CreatePen(PS_NULL, 1, RGB(255, 255, 0));

// 创建内侧实线画笔
pen[6].CreatePen(PS_INSIDEFRAME, 10, RGB(0, 0, 0));

// 创建具有刷子属性的几何画笔
LOGBRUSH lb;

lb.lbStyle=BS_HATCHED;

lb.lbColor=RGB(128, 128, 128);

lb.lbHatch=HS_DIAGCROSS;

pen[7].CreatePen(PS_GEOMETRIC, 10, &lb);

// 保存指向设备上下文原有画笔的指针
CPen *pOldPen;

// 以实线画笔绘制矩形
pOldPen=dc.SelectObject(&pen[0]);

dc.Rectangle(10, 10, 110, 110);
```

```
// 以虚线画笔绘制矩形
dc.SelectObject(&pen[1]);
dc.Rectangle(130, 10, 230, 110);
// 以点线画笔绘制矩形
dc.SelectObject(&pen[2]);
dc.Rectangle(250, 10, 350, 110);
// 以点划线画笔绘制矩形
dc.SelectObject(&pen[3]);
dc.Rectangle(370, 10, 470, 110);
// 以双点划线画笔绘制矩形
dc.SelectObject(&pen[4]);
dc.Rectangle(10, 130, 110, 230);
// 以空画笔绘制矩形，因此该矩形不会被显示出来
dc.SelectObject(&pen[5]);
dc.Rectangle(130, 130, 230, 230);
// 以内侧实线画笔绘制矩形，因此该矩形比使用实线画笔绘制的矩形看起来要小一些
dc.SelectObject(&pen[6]);
dc.Rectangle(250, 130, 350, 230);
// 以具有刷子属性的几何画笔绘制矩形
dc.SelectObject(&pen[7]);
dc.Rectangle(370, 130, 470, 230);
// 恢复设备上下文的原有画笔
dc.SelectObject(pOldPen);
// 删除所创建的画笔资源
for (int i=0; i<8; i++)
pen[i].DeleteObject();
```

}

上面的代码创建了八个画笔对象，每一个画笔对象对应了一种不同的画笔样式。使用这些不同的画笔样式进行输出的结果如图9.3所示。

注意第二排的第二个矩形，由于所使用的画笔具有PS\_NULL，所以实际上什么也没有绘制。再看最后一个矩形，具有PS\_GEOMETRIC样式的几何画笔可以具有刷子属性，这样，使用画笔绘制出来的线条就可以不只是单调实心图案或各种虚线和点划线了。

最后需要提醒的是，各种绘图工具对象是有限的系统资源，因此用过之后最好记得使用DelectObject释放为该GDI对象分配的系统资源。此外，在完成绘制工作之后，我们应该恢复设备描述表中的原有GDI对象。体现在上面的示例程序中，我们使用了一个CPen类型的指针来保存原有的GDI对象。类CDC的成员函数是将特定的GDI对象选入设备上下文的同时还返回了设备上下文中的原有对象，以便在以后进行恢复。

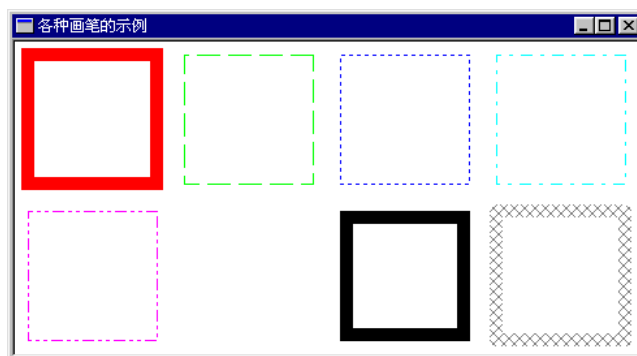


图9.3 不同画笔对象得到的不同输出结果

- 注意：
- 不能删除正被选入设备上下文中的Windows GDI对象。并且，删除Windows GDI对象也不等于删除相关联的C++对象。以上面的例子为例，调用CPen对象的DelectObject（该成员函数在基类CGdiObject中定义）只是删除与之相关联的Windows画笔对象所占用的系统资源，C++语义上的CPen对象并没有被删除。你还可以调用其成员函数CreatePen创建一个新的画笔对象并将它们与CPen对象相关联。在Windows编程中，我们所说的“对象”一词可能指实际的C++对象，也可能指各种Windows对象，在很多情况下两者之间有着密切的联系，但要注意它们并不是同义语。读者可以根据上下文来判断“对象”一词的真正含义。

### 9.2.3 图形输出函数

表9.4给出了在CDC在类中定义的一些图形输出函数。

表9.4 在类CDC中定义的图形输出函数

成员函数	描述
GetCurrentPosition	获得当前画笔位置的逻辑坐标
MoveTo	移动当前位置
LineTo	从当前位置向指定绘制一条不包括终点的直线
Arc	绘制一个椭圆弧
ArcTo	绘制一个椭圆弧，该函数与Arc类似，但当前位置会被更新
AngleArc	绘制一条线段和一个椭圆弧，并将当前位置移到椭圆弧的终点
GetArcDirection	返回设备上下文上当前的画弧方向

续表9.4

成员函数	描述
SetArcDirection	设置当前弧和矩形函数的绘图方向
PolyDraw	绘制一系列直线段和贝塞尔(Bézier)曲线段并更新当前位置
Polyline	绘制连接指定点的一系列直线段
PolyPolyline	绘制多系列的相连直线段，当前位置既不被使用也不被更新
PolylineTo	绘制一条或多条直线，并移动当前位置到最末一条线的终点
PolyBezier	绘制一条或多条贝塞尔曲线，当前位置既不被使用也不被更新
PolyBezierTo	绘制一条或多条贝塞尔曲线，并将当前位置移动到最后一条曲线的末端
FillRect	用指定的刷子填充给定的矩形
FrameRect	绘制矩形的边框

InvertRect	反转矩形的内容
DrawIcon	绘制一个图标
DrawDragRect	在矩形区域被拖动时擦除并重绘它
FillSolidRect	以原色填充一个矩形区域
Draw3dRect	填充一个三维矩形区域
DrawEdge	绘制矩形的边界
DrawFrameControl	绘制一个框架控件
DrawState	显示一幅图象并对图象应用表示其状态的可视效果
Chord	绘制一个“弓形”，一个“弓形”是由一个椭圆弧和一条线段所围成的区域
DrawFocusRect	绘制一个矩形以用来表示其焦点
成员函数	描述
Ellipse	绘制一个椭圆
Pie	绘制一个饼块
Polygon	绘制一个由多条线段连接而成的多边形
PolyPolygon	绘制一个或多个以当前多边形填充模式填充的多边形。这些多边形可能互不相交，也可能互相覆盖
Polyline	绘制一个包括一系列连接指定点的线段的多边形
Rectangle	使用当前笔和刷子绘制并填充一个矩形
RoundRect	使用当前笔和刷子绘制并填充一个圆角矩形

示例程序DrawingDemo演示了表9.4中的一些绘图函数的使用和输出效果。程序中涉及了较多的设备上下文输出函数，但由于这些函数的使用相对比较简单，因此我们仅给出示例程序DrawingDemo的代码清单，而不过多的分析各段程序代码。这些代码清单都有比较详细的注释，结构也非常清晰，很容易就可以看懂。同本章中前面的应用程序一样，示例程序DrawingDemo使用了MFC的应用程序框架，但我们没有使用AppWizard来生成它。程序中还涉及了一些我们目前还未作系统阐述的内容，在现阶段并不要求读者理解这些内容，尽管实际上它们



并不复杂。

```
#include <afxwin.h>

#include <afxext.h>

#include <math.h>

// 派生应用程序类

class CMyApp : public CWinApp

{

public:

virtual BOOL InitInstance();

};

// 派生窗口类

class CMyWnd : public CFrameWnd

{

protected:

// 声明消息处理函数

afx_msg void OnPaint();

DECLARE_MESSAGE_MAP();

};

// 初始化应用程序实例

BOOL CMyApp::InitInstance()

{

// 创建应用程序的主窗口

CMyWnd *pWnd=new CMyWnd;

pWnd->Create(NULL, "CDC 绘图函数示例");

// 显示应用程序主窗口并刷新其客户区

pWnd->ShowWindow(SW_SHOW);
```

```

pWnd->UpdateWindow();

// 在主窗口关闭时终止应用程序的执行线程

m_pMainWnd=pWnd;

return TRUE;

}

// 声明唯一的应用程序对象

CMyApp MyApp;

// 应用程序主窗口的消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CWnd)

ON_WM_PAINT()

END_MESSAGE_MAP()

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{

// 获得窗口的客户区设备上下文句柄

CPaintDC dc(this);

CPen pen1(PS_SOLID, 1, RGB(192, 192, 192)), pen2(PS_SOLID, 1, RGB(0, 0, 255)),
*pOldPen;

// 更改设备上下文所使用的当前字体，使之更适合于文本输出

LOGFONT lf;

dc.GetCurrentFont()->GetLogFont(&lf);

lf.lfHeight=-12;

lf.lfWidth=0;

strcpy(lf.lfFaceName, "宋体");

CFont font, *pOldFont;

font.CreateFontIndirect(&lf);

pOldFont=dc.SelectObject(&font);

```

```
// 使用函数 Arc 和 ArcTo 输入弧形
```

```
{  
pOldPen=dc.SelectObject(&pen1);  
dc.Rectangle(10, 10, 160, 110);  
dc.MoveTo(85, 60);  
dc.LineTo(160, 60);  
dc.MoveTo(85, 60);  
dc.LineTo(10, 10);  
dc.SelectObject(&pen2);  
dc.MoveTo(10,10);  
dc.ArcTo(10, 10, 160, 110, 160, 60, 10, 10);  
dc.Arc(10, 10, 160, 110, 10, 30, 160, 110);  
dc.TextOut(10, 115, "Arc & ArcTo");  
}
```

```
// 使用函数 PolyPolyline 输出多段折线
```

```
{  
dc.SelectObject(&pen1);  
dc.Rectangle(180, 10, 330, 110);  
CPoint pts[]={CPoint(190, 20), CPoint(200, 60), CPoint(270, 40), CPoint(210, 80),  
CPoint(250, 100), CPoint(300, 30), CPoint(310, 80), CPoint(320, 50)};  
DWORD pps[]={5, 3};  
dc.SelectObject(&pen2);  
dc.PolyPolyline(pts, pps, 2);  
CRect rc(200, 30, 310, 90);  
dc.TextOut(180, 115, "PolyPolyline");  
}
```

```
// 使用函数 DrawFocusRect 和 Draw3dRect 输出特殊样式的矩形

{
dc.SelectObject(&pen1);
dc.Rectangle(350, 10, 500, 110);
dc.SelectObject(&pen2);
dc.DrawFocusRect(CRect(370, 25, 480, 95));
dc.Draw3dRect(CRect(390, 40, 460, 80), RGB(192, 192, 192), RGB(64, 64, 64));
dc.SetBkColor(RGB(255, 255, 255));
dc.TextOut(350, 115, "Draw3dRect & DrawFocusRect");
}

// 使用 Pie 和 Chord 输出弓形和扇形

{
dc.SelectObject(&pen1);
dc.Rectangle(10, 140, 160, 240);
dc.SelectObject(&pen2);
dc.Ellipse(10, 140, 160, 240);
dc.Pie(20, 150, 150, 230, 160, 160, 10, 160);
dc.Chord(20, 150, 150, 230, 10, 220, 160, 220);
dc.TextOut(10, 245, "Ellipse, Pie & Chord");
}

// 使用 PolyDraw 输出贝塞尔曲线

{
dc.SelectObject(&pen1);
dc.Rectangle(180, 140, 330, 240);
dc.MoveTo(180, 140);
dc.LineTo(330, 160);
```

```

dc.MoveTo(330, 240);

dc.LineTo(180, 220);

dc.SelectObject(&pen2);

CPoint pts[]={CPoint(330, 160), CPoint(180, 220), CPoint(330, 240)};

BYTE typs[]={PT_BEZIERTO, PT_BEZIERTO, PT_BEZIERTO|PT_CLOSEFIGURE};

dc.MoveTo(180, 140);

dc.PolyDraw(pts, typs, 3);

dc.TextOut(180, 245, "PolyDraw");

}

// 使用 Polygon 输出多边形

{

dc.SelectObject(&pen1);

dc.Rectangle(350, 140, 500, 240);

dc.Ellipse(375, 140, 475, 240);

dc.SelectObject(&pen2);

CPoint pts1[5];

for (int i=0; i<=4; i++)

{

pts1[i].x=425+int(50.*cos(3.14159/2.5*i+3.14159/10.));

pts1[i].y=190-int(50.*sin(3.14159/2.5*i+3.14159/10.));

}

CPoint pts2[]={pts1[0], pts1[2], pts1[4], pts1[1], pts1[3]};

dc.Polygon(pts2, 5);

dc.TextOut(350, 245, "Polygon");

}

// 恢复设备上下文原有的 GDI 绘图对象

```

```

dc.SelectObject(pOldPen);

dc.SelectObject(pOldFont);

font.DeleteObject();

pen1.DeleteObject();

pen2.DeleteObject();

}

```

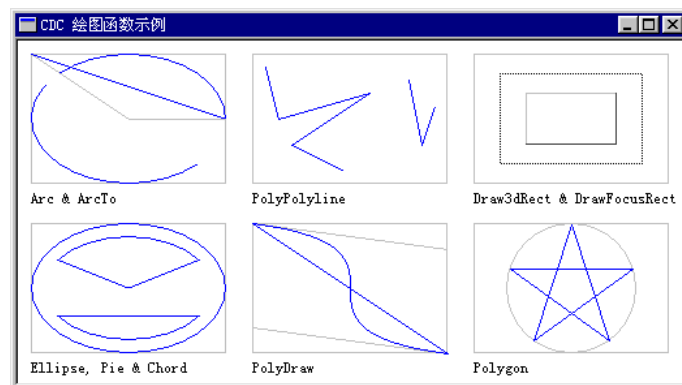


图9. 4 示例程序DrawingDemo的运行结果

程序DrawingDemo的运行结果如图9.4所示。

### 第三节 刷子对象

刷子对象用来在GDI输出时填充一个封闭图形的内部，它事实上定义一个8像素× 8像素大小的位图。在绘制时，Windows将多个这样的位图平铺起来填充封闭图形的内部。MFC类CBrush封装了标准的Windows刷子对象。在创建刷子时，我们通常先定义一个CBrush对象，然后调用CreateSolidBrush、CreateHatchBrush或者CreatePatternBrush之一来定义该刷子对象的属性。CBrush对象可以用作任何使用HBRUSH句柄的函数的参数。

以创建一个刷子之后，我们可以使用CDC类的成员函数SelectObject将它选入当前设备上下文作为当前绘图输出所使用的刷子。

成员函数CreateSolidBrush创建一个原色刷子，它仅带有一个类型为COLORREF的参数，该参数指定了刷子所使用的RGB颜色值。

成员函数CreateHatchBrush创建一个阴影刷子，其原型如下：

```

BOOL CreateHatchBrush( int nIndex, COLORREF crColor );

```

参数nIndex指定了刷子的样式，它可以为以下常量之一：

HS\_BDIAGONAL： 由左向右下斜45度的阴影线

HS\_CROSS： 水平和垂直的交叉线

HS\_DIAGCROSS： 45度的斜交叉线

HS\_FDIAGONAL： 由左向右上斜45度的阴影线

HS\_VERTICAL： 垂直阴影线

参数crColor指定和阴影线所使用的前景色。

成员函数CreatePatternBrush以一个指向CBitmap对象的指针为参数，它使用该CBitmap所代表的位图的左上角8像素× 8像素的区域来创建一个图案刷子。

- 注意：
- 一个刷子所使用的图案的大小总是8像素× 8像素大小。即使提供给成员函数CreatePatternBrush的位图大于这个大小，也仅有左上角的8像素× 8像素被使用。

示例程序BrushDemo演示了各种刷子的使用，在工程包括了两个位图资源IDB\_BRUSH1和IDB\_BRUSH2，分别如图9.5所示。

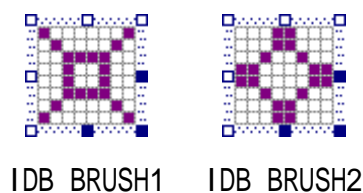


图9. 5 在工程BrushDemo中使用的位图资源

程序BrushDemo的代码清单如下：

```
#include <afxwin.h>

#include <afxext.h>

#include <math.h>
```

```
#define IDB_BRUSH1 101

#define IDB_BRUSH2 102

// 派生应用程序类

class CMyApp : public CWinApp

{

public:

virtual BOOL InitInstance();

};

// 派生窗口类

class CMyWnd : public CFrameWnd

{

protected:

// 声明消息处理函数

afx_msg void OnPaint();

DECLARE_MESSAGE_MAP();

};

// 初始化应用程序实例

BOOL CMyApp::InitInstance()

{

// 创建应用程序的主窗口

CMyWnd *pWnd=new CMyWnd;

pWnd->Create(NULL, "刷子示例");

// 显示应用程序主窗口并刷新其客户区

pWnd->ShowWindow(SW_SHOW);

pWnd->UpdateWindow();

// 在主窗口关闭时终止应用程序的执行线程
```



```

m_pMainWnd=pWnd;

return TRUE;

}

// 声明唯一的应用程序对象

CMyApp MyApp;

// 应用程序主窗口的消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CWnd)

ON_WM_PAINT()

END_MESSAGE_MAP()

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{

// 获得窗口的客户区设备上下文句柄

CPaintDC dc(this);

// 更改设备上下文所使用的当前字体，使之更适合于文本输出

LOGFONT lf;

dc.GetCurrentFont()->GetLogFont(&lf);

lf.lfHeight=-12;

lf.lfWidth=0;

strcpy(lf.lfFaceName, "宋体");

CFont font, *pOldFont;

font.CreateFontIndirect(&lf);

pOldFont=dc.SelectObject(&font);

// 创建一个原色刷子

{

CBrush br, *pOldBrush;

```

```
br.CreateSolidBrush(RGB(128, 0, 128));
pOldBrush=dc.SelectObject(&br);
dc.Rectangle(10, 10, 160, 110);
dc.SelectObject(pOldBrush);
br.DeleteObject();
dc.TextOut(10, 115, "原色刷子");
}

// 创建一个具有样式 HS_BDIAGONAL 的刷子
{
CBrush br, *pOldBrush;
br.CreateHatchBrush(HS_BDIAGONAL, RGB(128, 0, 128));
pOldBrush=dc.SelectObject(&br);
dc.Rectangle(180, 10, 330, 110);
dc.SelectObject(pOldBrush);
br.DeleteObject();
dc.TextOut(180, 115, "HS_BDIAGONAL");
}

// 创建一个具有样式 HS_CROSS 的刷子
{
CBrush br, *pOldBrush;
br.CreateHatchBrush(HS_CROSS, RGB(128, 0, 128));
pOldBrush=dc.SelectObject(&br);
dc.Rectangle(350, 10, 500, 110);
dc.SelectObject(pOldBrush);
br.DeleteObject();
dc.TextOut(350, 115, "HS_CROSS");
}
```

```

}

// 创建一个具有样式 HS_DIAGCROSS 的刷子

{

CBrush br, *pOldBrush;

br.CreateHatchBrush(HS_DIAGCROSS, RGB(128, 0, 128));

pOldBrush=dc.SelectObject(&br);

dc.Rectangle(520, 10, 670, 110);

dc.SelectObject(pOldBrush);

br.DeleteObject();

dc.TextOut(520, 115, "HS_DIAGCROSS");

}

// 创建一个具有样式 HS_FDIAGONAL 的刷子

{

CBrush br, *pOldBrush;

br.CreateHatchBrush(HS_FDIAGONAL, RGB(128, 0, 128));

pOldBrush=dc.SelectObject(&br);

dc.Rectangle(10, 140, 160, 240);

dc.SelectObject(pOldBrush);

br.DeleteObject();

dc.TextOut(10, 245, "HS_FDIAGONAL");

}

// 创建一个具有样式 HS_VERTICAL 的刷子

{

CBrush br, *pOldBrush;

br.CreateHatchBrush(HS_VERTICAL, RGB(128, 0, 128));

pOldBrush=dc.SelectObject(&br);

```

```

dc.Rectangle(180, 140, 330, 240);

dc.SelectObject(pOldBrush);

br.DeleteObject();

dc.TextOut(180, 245, "HS_VERTICAL");

}

// 创建一个使用位图图案的刷子

{

CBitmap bitmap;

bitmap.LoadBitmap(IDB_BRUSH1);

CBrush br, *pOldBrush;

br.CreatePatternBrush(&bitmap);

pOldBrush=dc.SelectObject(&br);

dc.Rectangle(350, 140, 500, 240);

dc.SelectObject(pOldBrush);

br.DeleteObject();

dc.TextOut(350, 245, "使用位图图案创建的刷子之一");

}

{

CBitmap bitmap;

bitmap.LoadBitmap(IDB_BRUSH2);

CBrush br, *pOldBrush;

br.CreatePatternBrush(&bitmap);

pOldBrush=dc.SelectObject(&br);

dc.Rectangle(520, 140, 670, 240);

dc.SelectObject(pOldBrush);

br.DeleteObject();

```

```

dc.TextOut(520, 245, "使用位图图案创建的刷子之二");
}

// 恢复设备上下文原有的 GDI 绘图对象
dc.SelectObject(pOldFont);
}

```

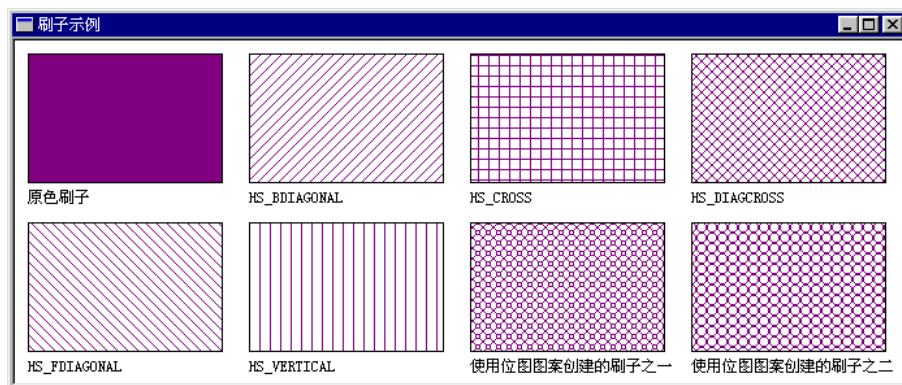


图9. 6 示例程序BrushDemo的运行结果

以前几节讲述的方法编译并链接上面的应用程序，运行结果如图9.6所示。

## 第四节 字体对象

MFC类CFont封装了Windows图形设备接口中的字体对象。字体对象决定的设备上下文中进行文本输出的字符样式。在使用字体对象的时候，我们一般先创建一个CFont对象，然后调用CreateFont、CreateFontIndirect、CreatePointFont及CreatePointFontIndirect之一的成员函数来对该字体对象进行初始化。

### 9.4.1 创建字体对象

创建字体对象的最方便的方法是使用CreatePointFont函数，CreatePointFont函数仅需三个参数，其原型如下：

```

BOOL CreatePointFont( int nPointSize, LPCTSTR lpszFaceName, CDC* pDC = NULL );

```

第一个参数nPointSize以十分之一磅为单位设置字体的大小，磅是印刷行业中的常用度量单位，1磅=1/72英寸 0.03528厘米。磅这个单位在涉及图形和文本输出的Windows应用程序中被大量的使用，因此我们应该熟知它和其它常用度量单位之间的换算关系。在后面的部分中我们还会讨论到在Windows编程中还会使用到的其它度量单位以及

它们之间的换算关系。

参数lpszFaceName指定了创建字体对象所使用的字体名，pDC指向一个设备上下文对象，函数CreatePointFont将以磅表示的字体大小转换为pDC所指向的设备上下文中相应的逻辑单位。如果指针pDC为空，函数CreatePointFont将字体大小以设备单位表示。

#### 9.4.2 LOGFONT结构

在Windows内部，字体是以一个名为LOGFONT的结构来表示的。结构LOGFONT的定义如下：

```
typedef struct tagLOGFONT { // If  
  
    LONG lfHeight;  
  
    LONG lfWidth;  
  
    LONG lfEscapement;  
  
    LONG lfOrientation;  
  
    LONG lfWeight;  
  
    BYTE lfItalic;  
  
    BYTE lfUnderline;  
  
    BYTE lfStrikeOut;  
  
    BYTE lfCharSet;  
  
    BYTE lfOutPrecision;  
  
    BYTE lfClipPrecision;  
  
    BYTE lfQuality;  
  
    BYTE lfPitchAndFamily;  
  
    TCHAR lfFaceName[LF_FACESIZE];  
  
} LOGFONT;
```

各成员的含义如下：

lfHeight :	以逻辑单位指定字体字符元 (character cell)或字符的高
------------	---------------------------------------

度。字符高度值为字符元高度值减去内部行距(internal-leading)值。当lfHeight大于0时，字体映射程序将该值转换为设备单位，并将它与可用字体的字符元高度进行匹配；当该参数为0时，字体映射程度将使用一个匹配的默认高度值；如果参数的值小于0，则将其转换为设备单位，并将其绝对值与可用字体的字符高度进行匹配。

对于任何一种情况，字体映射程度最终得到的字体高度值不会超过所指定的值。以MM\_TEXT映射模式下，字体高度值和磅值有如下的换算公式：

$$\text{lfHeight} = -\text{MulDiv}(\text{PointSize}, \text{GetDeviceCaps}(\text{hDC}, \text{LOGPIXELSY}), 72);$$

lfWidth：

以逻辑单位指定字体字符的平均宽度。如果lfWidth的值为0，则根据设备的纵横比从可用字体的数字转换纵横中选取最接近的匹配值，该值通过比较两者之间的差异的绝对值得出。

lfEscapement：

以十分之一度为单位指定每一行文本输出时相对于页面底端的角度。

ifOrientation：

以十分之一度为单位指定字符基线相对于页面底端的角度。

lfWeight：

指定字体重量。在Windows中，字体重量这个术语用来指代字体的粗细程度。lfWeight的范围为0到1000，正常情况下的字体重量为400，粗体为700。如果lfWeight为0，则使用默认的字体重量。

IfItalic :	当IfItalic为TRUE时使用斜体
IfUnderline :	当IfUnderline为TRUE时给字体添加下划线
IfStrikeOut :	当IfStrikeOut为TRUE时给字体添加删除线
IfCharSet :	指定字符集。可以使用下面的预定义值：

ANSI\_CHARSET

OEM\_CHARSET

SYMBOL\_CHARSET

UNICODE\_CHARSET

其中OEM字符集是与操作系统相关的。

IfOutPrecision :	指定输出精度。输出精度定义了输出与所要求的字体高度、宽度、字符方向等的接近程度。它可以为下面的值之一：
------------------	---

OUT\_CHARACTER\_PRECIS

OUT\_DEFAULT\_PRECIS

OUT\_STRING\_PRECIS

OUT\_STROKE\_PRECIS

IfClipPrecision :	指定剪辑精度。剪辑精度定义了当字符的一部分超过剪辑区域时对字符的剪辑方式，它可以为下列值之一：
-------------------	---

CLIP\_CHARACTER\_PRECIS

CLIP\_DEFAULT\_PRECIS



CLIP\_STROKE\_PRECIS

IfQuality :

定义输出质量。输出质量定义了图形设备接口在匹配逻辑字体属性到实际的物理字体的所使用的方式，它可以为下列值之一：

DEFAULT\_QUALITY (默认质量)

DRAFT\_QUALITY (草稿质量)

PROOF\_QUALITY (正稿质量)

IfPitchAndFamily :

指定字体的字符间距和族。最低两位指定字体的字符间距为以下值之一：

DEFAULT\_PITCH

FIXED\_PITCH

VARIABLE\_PITCH

第4到7位指定字体族为以下值之一：

FF\_DECORATIVE

FF\_DONTCARE

FF\_MODERN

FF\_ROMAN

FF\_SCRIPT

FF\_SWISS

这些值的具体含义可以参考Visual C++中关于结构LOGFONT的文档。

字符间距和字体族可以使用逻辑

或(OR)运算符来进行组合。

l fFaceName :

一个指定以NULL结尾的字符串的指针，它指定的所用的字体名。该字符串的长度不得超过32个字符，如果l fFaceName为NULL，图形设备接口将使用默认的字体名。

### 9.4.3 使用字体对象和枚举系统中的所有字体

示例程序FontDemo演示了LOGFONT结构和CFont对象的使用。此外，在该示例程序中，我们还演示了如何获得当前系统中已安装的所有可用字体，这些信息是通过API函数EnumFontFamilies和自定义的字体枚举回调函数EnumFontFamProc来得到，并放入程序主窗口内的列表框中的。

```
#include <afxwin.h>

#include <afxext.h>

#include <math.h>

#define DegToRnd(x) (x/180.*3.14159)

int WINAPI EnumFontFamProc(const LOGFONTA *lpLf,
const TEXTMETRICA *lpTm, unsigned long FontType, LPARAM lParam);

// 派生应用程序类

class CMyApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};

// 派生窗口类

class CMyWnd : public CFrameWnd
{
```

```

public:
    CListBox lst;

protected:
    // 声明消息处理函数

    afx_msg void OnPaint();

    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);

    DECLARE_MESSAGE_MAP();

};

// 初始化应用程序实例

BOOL CMyApp::InitInstance()

{
    // 创建应用程序的主窗口

    CMyWnd *pWnd=new CMyWnd;

    pWnd->Create(NULL, "字体示例");

    // 显示应用程序主窗口并刷新其客户区

    pWnd->ShowWindow(SW_SHOW);

    pWnd->UpdateWindow();

    // 在主窗口关闭时终止应用程序的执行线程

    m_pMainWnd=pWnd;

    return TRUE;

}

// 声明唯一的应用程序对象

CMyApp MyApp;

// 应用程序主窗口的消息映射

BEGIN_MESSAGE_MAP(CMyWnd, CWnd)

    ON_WM_CREATE()

```

```
ON_WM_PAINT()

END_MESSAGE_MAP()

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{
    // 获得窗口的客户区设备上下文句柄

    CPaintDC dc(this);

    // 设置字体输出背景为透明

    dc.SetBkMode(TRANSPARENT);

    // 获得设备上下文所使用的当前字体

    LOGFONT lf;

    dc.GetCurrentFont()->GetLogFont(&lf);

    CFont font1, font2;

    CFont *pOldFont; // 保存设备上下文最初使用的字体对象

    // 创建 font1 为 12 象素宋体

    lf.lfCharSet=134;

    lf.lfHeight=-12;

    lf.lfWidth=0;

    strcpy(lf.lfFaceName, "宋体");

    font1.CreateFontIndirect(&lf);

    // 更改当前字体为 20 象素的 Times New Roman , 并且向上倾斜 40 度

    lf.lfCharSet=0;

    strcpy(lf.lfFaceName, "Times New Roman");

    lf.lfEscapement=400;

    lf.lfHeight=-20;

    font2.CreateFontIndirect(&lf);
```

```
pOldFont=dc.SelectObject(&font2);

// 获得字符串 "lfEscapement= 400" 在输出时的宽度和高度
CSize sz=dc.GetTextExtent("lfEscapement= 400");

// 计算字符串合适的输出位置
dc.TextOut(10, 10+int(sz.cx*sin(DegToRad(40))), "lfEscapement= 400");

// 将字体输出方向更改为向下倾斜 40 度
lf.lfEscapement=-400;

dc.SelectObject(pOldFont);

font2.DeleteObject();

font2.CreateFontIndirect(&lf);

dc.SelectObject(&font2);

// 计算字符串合适的输出位置
dc.TextOut(290-int(sz.cx*cos(DegToRad(40))), 10, "lfEscapement=-400");

// 将字体输出方向更改为水平方向
lf.lfEscapement=0;

dc.SelectObject(pOldFont);

font2.DeleteObject();

font2.CreateFontIndirect(&lf);

dc.SelectObject(&font2);

// 对称于直线 x=150 输出字符串 "lfEscapement=0"
sz=dc.GetTextExtent("lfEscapement=0");

dc.TextOut(150-sz.cx/2, 110, "lfEscapement=0");

// 更改当前字体为最细的 50 象素大小的宋体
lf.lfCharSet=134;

strcpy(lf.lfFaceName, "宋体");

lf.lfEscapement=0;
```

```
lf.lfWeight=0;

lf.lfHeight=-50;

dc.SelectObject(pOldFont);

font2.DeleteObject();

font2.CreateFontIndirect(&lf);

dc.SelectObject(&font2);

// 输出一个 "细" 字

dc.TextOut(330, 10, "细");

// 在旁边使用 12 像素大小的宋体字给出当前字体的重量

dc.SelectObject(&font1);

dc.TextOut(395, 29, "lfWeight=0");

// 更改当前字体为最粗的 50 像素大小的字体

lf.lfWeight=1000;

dc.SelectObject(pOldFont);

font2.DeleteObject();

font2.CreateFontIndirect(&lf);

dc.SelectObject(&font2);

// 输出一个 "粗" 字

dc.TextOut(330, 80, "粗");

dc.SelectObject(&font1);

// 在旁边使用 12 像素大小的宋体字给出当前字体和重量

dc.TextOut(395, 99, "lfWeight=1000");

// 恢复设备上下文原有的 GDI 绘图对象

dc.SelectObject(pOldFont);

}

// 主窗口的 WM_CREATE 消息的处理函数
```

```

int CMyWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // 调用基类的默认消息处理函数

    int iResult=CWnd::OnCreate(lpCreateStruct);

    // 创建一个列表框，该列表框中给出了当前系统中安装的所有可用字体

    lst.Create(LBS_STANDARD | LBS_USETABSTOPS | WS_CHILD | WS_VISIBLE,
    CRect(10, 150, 480, 350), this, 1001);

    // 更改列表框所用的字体为 9 磅 (12 象素) 大小的宋体

    static CFont font;

    font.CreatePointFont(90, "宋体");

    lst.SetFont(&font);

    // 设置列表框的制表符位置为 200 个对话框单位

    lst.SetTabStops(200);

    // 枚举当前系统的所有可用字体，将指向列表框的 CListBox 对象的指针作为应用程序提供的

    // 参数传递给枚举字体回调函数 EnumFontFamProc

    EnumFontFamilies(::GetDC(GetSafeHwnd()), NULL, EnumFontFamProc, (LPARAM)&lst));

    return iResult;
}

// 枚举字体时的回调函数，该函数将系统的所有可用字体及其字符集添加到列表框中，
// 由应用程序提供的参数 lParam 提供了指向该列表框的 CListBox 对象的指针

int WINAPI EnumFontFamProc(const LOGFONTA * lpLf,
const TEXTMETRICA * lpTm, unsigned long FontType, LPARAM lParam)
{
    CListBox *pList=(CListBox*)lParam;

    CString str;

    // 将当前字体的字体名 (FaceName) 和字符集 (CharSet) 添加到列表框中

```

```

str.Format("FACENAME: %s \tCHARSET: %d", lpIf->lfFaceName, lpIf->lfCharSet);

pList->AddString(str);

return TRUE; // 返回 TRUE 以继续字体枚举的过程，返回 FALSE 将终止字体枚举的过程
}

```

我们先来看重绘消息处理函数OnPaint，一开始时，我们调用了CDC类的成员函数GetCurrentFont，该成员函数返回当前设备上下文所使用的字体，其返回值是一个指向CFont对象的指针，然后，我们通过该指针调用CFont类的成员函数GetLogFont，该成员函数将字体的信息填入到一个LOGFONT结构中。在下面的步骤中，我们通过修改该结构的成员来创建新的字体对象。首先，我们创建一个CFont对象font1，font1使用了12个像素大小的宋体字，在程序中它主要用来输出一些标识文本。这里，我们先在LOGFONT结构对象lf在相关成员中填入新的值，再以该结构对象为参数来调用CFont类的成员函数CreateFontIndirect创建相应的GDI字体对象。这里我们将lfCharSet成员修改为134，这个值可以通过本程序的运行结果得出，lfWidth成员修改为0，这样将使用默认的字符纵横比得到字符的宽度。

在下面过程中，我们按照类似的方法创建一大一小为20个像素的Times New Roman字体，与刚才不同的是，我们将结构对象lf的lfEscapement成员的值设置为400，这样，文本将以向上倾斜40度（lfEscapement的值的单位为1/10度）的角度进行输出。接着，我们将该字体对象通过CDC的成员函数SelectObject选入设备上下文中作为设备上下文的当前字体。CDC类的成员函数GetTextExtent可以在输出一个字符串之前得到该输出字符串的大小，以便于我们可以恰当的安排字符串的输出位置。需要注意的是，通过该成员函数得到的度量值不会受到我们在lfEscapement中设置的值的影响。在程序示例中，我们通过成员函数GetTextExtent得到字符串"lfEscapement= 400"在输出时的长度和宽度，然后根据所得的结果计算得出以40度角输出文本串的合适的起始位置，最后调用CDC类的成员函数TextOut以当前字体输出字符串"lfEscapement= 400"。

- 注意：
- 不管当前的lfEscapement值如何，函数TextOut总是以输出字符串的第一个字符的起始位置的坐标作为其前两个参数。
- 在Windows 95中，lfEscapement和lfOrientation总是具有相同的值，而在Windows NT中，两者在某些情况下可以不相同。



- 设置 `lfFaceName` 时应该使用库函数 `strcpy`，不要犯这样的错误：
- `lfFaceName="宋体";`

此外，如果为 `lfFaceName` 设置了新值，同时也应该将 `lfCharSet` 的值设置为相匹配的字符集。如果字符集与字体名不匹配，将会导致设置不起作用。

要使文本向下倾斜输出，我们只需简单的将 `lfEscapement` 设置为负值。如下面的代码所示：

```
// 将字体输出方向更改为向下倾斜 40 度

lf.lfEscapement=-400;

dc.SelectObject(pOldFont);

font2.DeleteObject();

font2.CreateFontIndirect(&lf);

dc.SelectObject(&font2);

// 计算字符串合适的输出位置

dc.TextOut(290-int(sz.cx*cos(DegToRad(40))), 10, "lfEscapement=-400");
```

上面的代码摘自应用程序 `FontDemo`。

- 注意：
- 在调用 `font2` 的 `CreateFontIndirect` 成员函数创建新的字体对象之前，应该先调用其成员函数 `DeleteObject` 删除该字体对象，而当一个 GDI 图形对象正为设备上下文所使用时，我们不能删除该图形对象，因此在前面的代码中，我们在删除在 `font2` 原有的字体对象之前先将设备上下文的字体对象进行复原。

接着在消息处理函数 `CMyWnd::OnPaint` 中，我们又将 `lfEscapement` 成员的值设置为 0，输出字符串 "`lfEscapement=0`" 以示对比。

在随后的代码中，我们演示了 `lfWeight` 成员的不同值对字体的笔划粗细的影响。我们先将 `lfWeight` 值设置为 0，以 50 象素的宋体字绘制了一个“细”字，然后再将 `lfWeight` 值设置为 1000，以同样大小和同种字体绘制了一个“粗”字。通过如图 9.7 的输出结果，我们看到字体笔划的粗细发生的明显的变化。

在OnPaint函数返回之前，不要忘记恢复设备上下文的原有字体对象，指向该对象的CFont指针在前面被保存到了名为pOldFont的指针变量中。我们仍然使用SelectObject将其选入当前设备上下文。

上面我们来看示例程序FontDemo的另一个主要的功能板块，即枚举当前系统中所安装的所有字符并将它添加到一个列表框中。

首先我们在窗口CMyWnd的WM\_CREATE消息的处理函数OnCreate中调用CListBox对象lst（该对象被定义为类CMyWnd的成员变量）的Create成员函数。在Create成员函数中，我们指定了列表框的样式包括了LBS\_USETABSTOPS，该样式允许在列表项中使用制表符，这些制表符在显示时会被扩展到指定的位置。

接着，我们将列表框所使用的字体设置为9磅大小的宋体字。这里我们调用的是CFont对象的CreatePointFont成员函数来创建字体。当需要创建的指定磅值大小的某种字体时，使用CreatePointFont成员函数要方便得多，因此该函数仅需要三个参数，并且，第三个参数在很多情况下可以省略。这样，我们就可以避开填写复杂的LOGFONT结构。

在改变列表框字体的同时，我们将列表框中的当前制表位设置为200个对话框单位，对话框单位是一种在控件和对话框使用的度量单位。每4个水平对话框单位等于以系统字体显示的字符的平均宽度，我们还将这个宽度称作对话框基本单位。对话框基本单位的具体量值可以通过API函数GetDialogBaseUnits得到，该函数返回值的低位字代表对话框水平基本单位，高位字代表对话框垂直基本单位。

紧接着我们调用了API函数EnumFontFamilies来枚举系统中的所有可用字体，该函数使用4个参数，第一个参数为枚举所使用设备上下文句柄，我们使用API函数GetDC来得到代表当前窗口的客户区；为了枚举系统中的所有字体，我们将第二个参数设置为NULL；第三个参数为枚举字体回调函数，这里为EnumFontFamProc，我们将在下面定义该回调函数；第四个参数为指向列表框的指针，这个参数将被传递给回调函数，由于我们需要将可用的字体添加到列表框中，因此我们可以很自然的将指向该列表框指针当前应用程序提供的参数传递给枚举字体回调函数。

枚举字体回调函数的原型在程序中被声明为

```
int WINAPI EnumFontFamProc(const LOGFONTA *lpIf,  
const TEXTMETRICA *lpTm, unsigned long FontType, LPARAM lParam);
```

- 注意：
- 在随Visual C++ 5.0提供的Platform SDK (即Win32 SDK)中所给出的EnumFontFamProc的原型为
- `int CALLBACK EnumFontFamProc( ENUMLOGFONT FAR *lpelf, // pointer to logical-font data`
- `NEWTEXTMETRIC FAR *lpntm, // pointer to physical-font data`
- `int FontType, // type of font`
- `LPARAM lParam // address of application-defined data`
- `);`

然而在本程序中使用上面所给的原型将会在编译时导致类型不匹配。正确的回调函数的原型应该如代码清单中所给的那样。

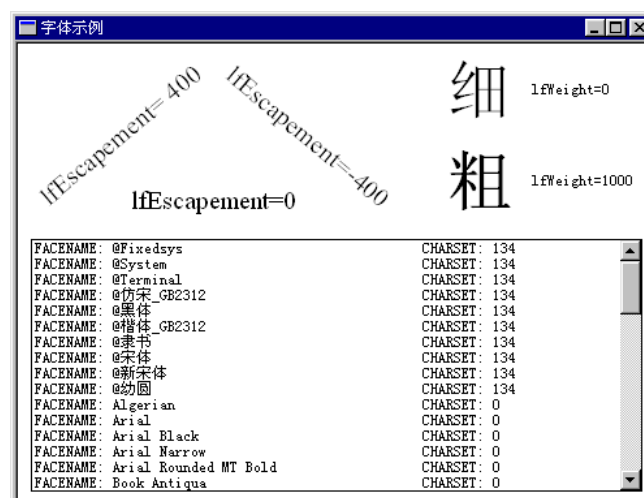


图9. 7 示例程序FontDemo的运行结果

在本程序中，传递给回调函数的第一个参数lpelf为该字体对应的LOGFONTA结构，最后一个参数为指向列表框的CListBox指针。第二个参数和第三个参数在本程序中没有使用。在本程序中，枚举字体回调函数的结构很简单，它只是将字体的字体名和相应的字符集格式化之后添加到列表框中。为了使枚举继续进行，回调函数应该返回真值，如果回调函数返回了FALSE，则枚举的过程将被终止。

示例程序FontDemo的运行结果如图9.7所示。

在类CDC中定义的字体和文本函数如表所示。

表9. 5 在类CDC中定义的字体和文本函数

成员函数	描述
TextOut	在指定位置以当前选定字体绘制字符串
ExtTextOut	在指定的矩形区域内使用当前选定字体绘制字符串
TabbedTextOut	以指定的位置绘制字符串，并按指定的制表符位置扩展字符串的制表符
DrawText	在指定的矩形区域内绘制格式化文本
GetTextExtent	使用当前字体中属性设备上下文中计算一行文本的宽度和高度
GetOutputTextExtent	在输出设备上下文中计算一字符串的宽度和高度
GetTabbedTextExtent	在属性设备上下文中计算一字符串的宽度和高度
GetOutputTabbedTextExtent	在输出设备上下文中计算一字符串的宽度和高度
GrayString	在指定位置绘制变灰的文本
GetTextAlign	获得文本对齐标志
SetTextAlign	设置文本对齐标志
GetTextFace	将当前字体的字体名拷贝到缓冲区
GetTextMetrics	从属性设备上下文中获得当前字体的度量值
GetOutputTextMetrics	从输出设备上下文中获得当前字体的度量值
SetTextJustification	在字符串的分隔字符处添加空白
GetTextCharacterExtra	获得字符间空白的当前设置
SetTextCharacterExtra	设置字符间空白的当前设置
GetFontData	从可缩放字体文件中获取字体信息。所获取的信息通过指定字体文件中的偏移量和返回信息的长度来确定

GetKerningPairs	在选定的设备上下文中获得当前选定字体的字距调整字符对
GetOutlineTextMetrics	获得TrueType字体的字体度量信息
GetGlyphOutline	返回当前字体的字符的轮廓曲线或位图
GetCharABCWidths	从当前字体中以逻辑单位返回给定范围的连续字符的宽度
GetCharWidth	从当前字体中返回给定范围的连续字符的相对宽度
GetOutputCharWidth	从输出设备上下文中的当前字体返回连续字符组中若干单个字符的宽度

续表9.5

成员函数	描述
SetMapperFlags	改变字体映射程序中从逻辑字符到物理字体的映射过程中所使用的算法
GetAspectRatioFilter	获得当前纵横比过滤器的设定

在一些应用程序(如字处理应用程序)中，我们一般需要由用户来指定所使用的字体。这时常使用的方法是弹出一个字体对话框，用户通过该字体对话框来设置应用程序所使用的字体。MFC类CFontDialog封装了标准的Windows字体对话框。在最简单的情况下，我们只需要声明一个类的实例对象CFontDialog，然后通过该对象调用类CFontDialog的成员函数DoModal，如果该成员函数返回IDOK，则通过成员函数GetCurrentFont将用户所选择的字体信息填入一个LOGFONT结构中，在下面的过程中即可通过该结构创建CFont对象。在很多情况下，我们需要为字体对话框设置一些初始值，一种很简单的方式在其构造函数中传递一个指向LOGFONT结构对象的指针。我们可以在创建CFontDialog对象之后，调用DoModal成员之前改变其类型为CHOOSEFONT的成员结构m\_cf的各成员的值来为字体对话框进行初始设置。

#### 9.4.4 创建特殊的字体效果

在一般的应用程序中，我们可以使用SetBkMode和SetBkColor来设置绘制文本所使用的颜色和模式，但是，这两个函数所设置的效果是很有限的。有时候我们可能希望得到一些特殊的文本输出效果。这时我

们就应该考虑其它特殊的实现方式。使用路径是其中的一种方法。下面我们讲述一些使用路径得到的特殊的字体效果。

### (1) 空心字

在开始一个路径前，我们先调用CDC类的成员函数BeginPath，然后调用一系列的输出函数，在完成绘制之后，我们可以调用CDC类的成员函数EndPath。在完成一个路径之后，我们可以调用StrokePath来绘制该路径。为了简单起见，我们仅给出应用程序的OnPaint成员函数如下：

```
// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{
    // 获得窗口的客户区设备上下文句柄

    CPaintDC dc(this);

    // 更改当前字体

    LOGFONT lf;

    dc.GetCurrentFont()->GetLogFont(&lf);

    CFont font;

    CFont *pOldFont; // 保存设备上下文最初使用的字体对象

    lf.lfCharSet=134;

    lf.lfHeight=-150;

    lf.lfWidth=0;

    strcpy(lf.lfFaceName, "隶书");

    font.CreateFontIndirect(&lf);

    pOldFont=dc.SelectObject(&font);

    dc.SetBkMode(TRANSPARENT);

    // 更改当前画笔

    CPen pen(PS_SOLID, 1, RGB(255, 0, 0));
```

```

CPen *pOldPen;

pOldPen=dc.SelectObject(&pen);

// 开始一个路径

dc.BeginPath();

dc.TextOut(10, 10, "空心字");

dc.EndPath();

// 绘制路径

dc.StrokePath();

// 恢复设备上下文的原有设置

dc.SelectObject(pOldFont);

dc.SelectObject(pOldPen);

}

```

上面的程序的运行结果如图9.8所示。

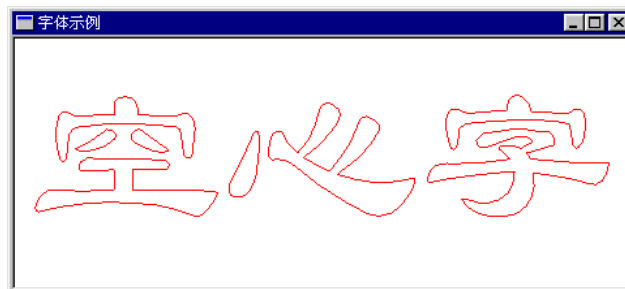


图9. 8 绘制空心字

函数FillPath可以使用当前刷子填充路径的内部。按下面的代码修改前面的OnPaint成员函数：

```

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{

// 获得窗口的客户区设备上下文句柄

CPaintDC dc(this);

// 更改当前字体

```

```
LOGFONT lf;  
dc.GetCurrentFont()->GetLogFont(&lf);  
CFont font, *pOldFont;  
lf.lfCharSet=134;  
lf.lfHeight=-150;  
lf.lfWidth=0;  
strcpy(lf.lfFaceName, "隶书");  
font.CreateFontIndirect(&lf);  
pOldFont=dc.SelectObject(&font);  
dc.SetBkMode(TRANSPARENT);  
// 更改当前画笔  
CPen pen(PS_SOLID, 1, RGB(255, 0, 0)), *pOldPen;  
pOldPen=dc.SelectObject(&pen);  
// 更改当前刷子  
CBrush br(HS_DIAGCROSS, RGB(0, 255, 255)), *pOldBrush;  
pOldBrush=dc.SelectObject(&br);  
// 开始一个路径  
dc.BeginPath();  
dc.TextOut(10, 10, "空心字");  
dc.EndPath();  
// 绘制路径  
dc.StrokeAndFillPath();
```



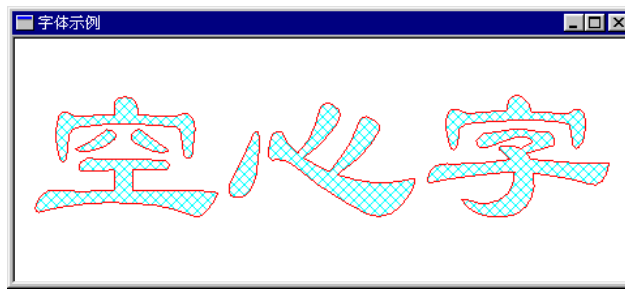


图9. 9 使用刷子填充空心字的内部

```
// 恢复设备上下文的原有设置  
dc.SelectObject(pOldFont);  
dc.SelectObject(pOldPen);  
dc.SelectObject(pOldBrush);  
}
```

上面的程序的运行结果如图9.9所示。

## (2) 渐变字

在完成一个路径之后，如前所述，我们可以调用CDC类的成员函数 FillPath、StrokePath或StrokeAndFillPath来绘制和填充路径。这只是最初级的技巧。更进一步，我们可以使用成员函数 SelectClipPath将路径选入当前剪辑区域，这样，所有的绘制操作都将只作用于这个剪辑区域。

使用此技巧可以绘制具有渐变颜色效果的字体。如下面的OnPaint成员函数所示：

```
// 应用程序主窗口的重绘函数  
void CMyWnd::OnPaint()  
{  
    // 获得窗口的客户区设备上下文句柄  
    CPaintDC dc(this);  
    // 更改当前字体  
    LOGFONT lf;  
    dc.GetCurrentFont()->GetLogFont(&lf);
```

```
CFont font, *pOldFont;

lf.lfCharSet=134;

lf.lfHeight=-150;

lf.lfWidth=0;

strcpy(lf.lfFaceName, "隶书");

font.CreateFontIndirect(&lf);

pOldFont=dc.SelectObject(&font);

dc.SetBkMode(TRANSPARENT);

// 更改当前画笔为空

CPen pen(PS_NULL, 1, RGB(255, 0, 0)), *pOldPen;

pOldPen=dc.SelectObject(&pen);

// 更改当前刷子

CBrush br(0), *pOldBrush;

pOldBrush=dc.SelectObject(&br);

// 开始一个路径

dc.BeginPath();

dc.TextOut(10, 10, "渐变字");

dc.EndPath();

// 绘制渐变效果

dc.SelectClipPath(RGN_COPY);

for (int i=255; i>0; i--)

{

    int iRadius=(600*i)/255;

    dc.SelectObject(pOldBrush);

    br.DeleteObject();

    br.CreateSolidBrush(RGB(255, i, 0));
```

```

dc.SelectObject(&br);

dc.Ellipse(-iRadius, -iRadius/3, iRadius, iRadius/3);

}

// 恢复设备上下文的原有设置

dc.SelectObject(pOldFont);

dc.SelectObject(pOldPen);

dc.SelectObject(pOldBrush);

}

```

在上面的示例中，我们以RGN\_COPY方式将路径选作当前剪辑区域，然后，在该剪辑区域上进行一系列的绘制操作，这些绘制操作以不同的颜色绘制了一系列的同心椭圆，这些同心椭圆有视觉上给用户以渐变的感觉。上面的程序的运行结果如图。

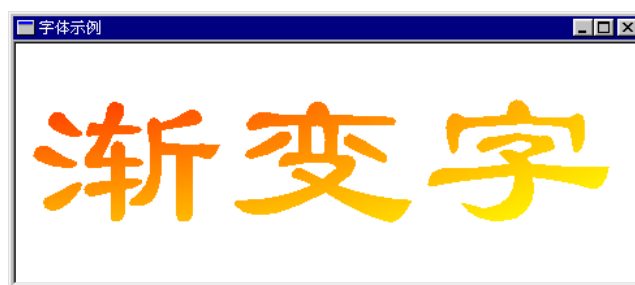


图9. 10 绘制渐变字

### (3) 使用不同的光栅模式创建特殊效果

光栅模式决定了画笔的颜色和屏幕上原有的点的颜色值在进行绘制时的组合方式。可以使用类CDC的成员函数SetROP2来设置设备上下文所使用的光栅绘制模式。该操作仅对光栅设备起作用。通过指定不同的光栅模式，我们可以制造一些特殊的效果，比如说反色字等。

下面的OnPaint处理函数演示了16种不同的光栅模式，需要注意的是，即使是同一种光栅模式，在当前画笔的颜色不同时所产生的结果也可能有很大差异。

```

// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()

{

```

```

// 获得窗口的客户区设备上下文句柄

CPaintDC dc(this);

// 更改当前字体

LOGFONT lf;

dc.GetCurrentFont()->GetLogFont(&lf);

CFont font1, font2, *pOldFont;

lf.lfCharSet=134;

lf.lfWidth=0;

lf.lfHeight=-12;

strcpy(lf.lfFaceName, "宋体");

font2.CreateFontIndirect(&lf);

lf.lfHeight=-50;

lf.lfWeight=1000;

strcpy(lf.lfFaceName, "黑体");

font1.CreateFontIndirect(&lf);

pOldFont=dc.SelectObject(&font1);

dc.SetBkMode(TRANSPARENT);

// 更改当前画笔为空

CPen pen1(PS_NULL, 1, RGB(0, 0, 0)), pen2(PS_SOLID, 1, RGB(0, 0, 0)), *pOldPen;

pOldPen=dc.SelectObject(&pen1);

// 更改当前刷子

CBrush br(RGB(0, 0, 255)), *pOldBrush;

pOldBrush=dc.SelectObject(&br);

// 绘制背景

for (int i=0; i<1000; i+=20)

{

```

```

for (int j=0; j<1000; j+=20)
{
dc.Rectangle(i, j, i+10, j+10);
dc.Rectangle(i+10, j+10, i+20, j+20);
}
}

dc.SelectObject(&pen2);

int nDrawModes[]={R2_BLACK, R2_WHITE, R2_NOP,
R2_NOT, R2_COPYPEN, R2_NOTCOPYPEN,
R2_MERGEENNOT, R2_MASKPENNOT, R2_MERGEENNOTPEN,
R2_MASKNOTPEN, R2_MERGEEN, R2_NOTMERGEEN,
R2_MASKPEN, R2_NOTMASKPEN, R2_XORPEN,
R2_NOTXORPEN};

char *szDrawModes[]={ "R2_BLACK", "R2_WHITE", "R2_NOP",
"R2_NOT", "R2_COPYPEN", "R2_NOTCOPYPEN",
"R2_MERGEENNOT", "R2_MASKPENNOT", "R2_MERGEENNOTPEN",
"R2_MASKNOTPEN", "R2_MERGEEN", "R2_NOTMERGEEN",
"R2_MASKPEN", "R2_NOTMASKPEN", "R2_XORPEN",
"R2_NOTXORPEN"};

for (i=0; i<16; i++)
{
// 更改当前光栅模式，绘制字体的前景
dc.SetBkMode(TRANSPARENT);
dc.SetROP2(nDrawModes[i]);
dc.BeginPath();
dc.SelectObject(&font1);

```

```

dc.TextOut((i%4)*200+10, (i/4)*80+10, "光栅字");

dc.EndPath();

dc.FillPath();

// 以 R2_BLACK 光栅模式绘制字体的轮廓

dc.SetROP2(R2_BLACK);

dc.BeginPath();

dc.TextOut((i%4)*200+10, (i/4)*80+10, "光栅字");

dc.EndPath();

dc.StrokePath();

// 显示所使用的光栅模式

dc.SelectObject(&font2);

dc.SetBkMode(OPAQUE);

dc.TextOut((i%4)*200+10, (i/4)*80+70, szDrawModes[i]);

}

// 恢复设备上下文的原有设置

dc.SelectObject(pOldFont);

dc.SelectObject(pOldPen);

dc.SelectObject(pOldBrush);

}

```

该程序的运行结果如图9.11所示。

这里再强调一下，除了R2\_BLACK，R2\_WHITE，R2\_NOP，R2\_NOT以外，其作光栅模式的效果都同当前画笔的颜色有关。比如在上面的示例中，如果当前画笔(刷子)的颜色值为背景方格相同，即都为蓝色，那么光栅模式R2\_MERGEPEENNOT和R2\_NOTXORPEN都将屏幕上的蓝色区域绘制为白色，白色区域绘制为蓝色。但是，如果当前画笔(刷子)的颜色不同蓝色，则得到的结果则可能不是这样。



图9. 11 使用不同的光栅模式的绘制效果

各种不同的光栅模式实际上是将背景颜色和画笔或刷子的颜色进行某种位运算来得到实际输出的颜色值，关于每一种光栅模式的具体含义可以参考Visual C++中关于函数CDC::SetROP2和SetROP2的联机文档。

## 第五节 映射模式

作为本章的末尾，我们来讲述一个以前一直没有涉及的问题。这就是在设备上下文中绘制时所使用的坐标系统。在前面的示例中，我们在绘图时使用了如图9.12所示的坐标系统。

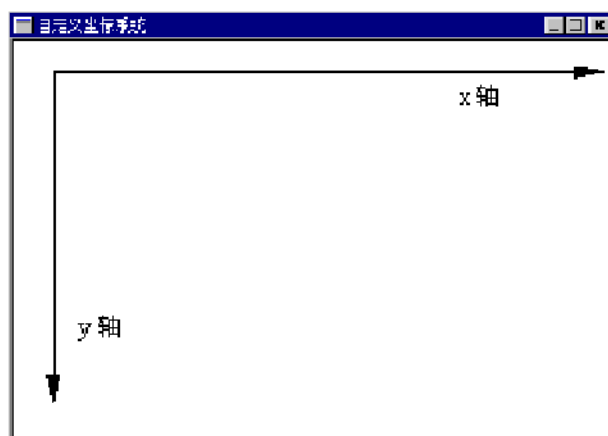


图9. 12 MM\_TEXT映射模式

这种映射模式称作MM\_TEXT映射模式。在这样的坐标系统中，向右的方向为x轴的正方向，向下的方向为y轴的正方向。无论是x方向还是y方向，每一个单位长度都代表设备上的一个像素。这种映射模式对于很多绘制操作是合理的，由于它以像素为单位，所以我们可以很直观的控制图形在屏幕上输出的大小。但是，其缺陷也是明显的。考虑不同的设备，如屏幕和打印机，它们所对应的像素的大小是不同的，因

此，在不同的设备上输出的相同像素大小的图形，其实际大小可能有很大的差异。

为了解决这个问题，Windows提供了设备无关的映射模式。这些模式使用了诸如1/1000英寸、1/100毫米之类的单位长度。在不同的设备上进行输出的时候，Windows自动将逻辑单位换算成对应的设备单位。这样，你在屏幕上输出的1厘米的正方形和在打印机上输出的1厘米的正方形，它们的实际大小是一样的，尽管由于设备在分辨率上的差异，它们所包括的像素的数目可能不一样。

### 9.5.1 预定义的映射模式

对当前映射模式的操作是在设备上下文中进行的，因此，MFC在CDC类中提供了成员函数SetMapMode来改变当前所使用的映射模式。

函数SetMapMode使用一个整型量作为其参数，该参数可以取以下的常量之一：

- |                 |  |
|-----------------|--|
| MM_ANISOTROPIC： | 使用自定义的映射模式，这种映射模式在x方向和y方向均使用自定义的单位长度，并且两个方向上的单位长度可以不一样。如果将映射模式设置为MM_ANISOTROPIC，需要调用CDC类的成员函数SetWindowExt和SetViewportExt来设置其单位长度、坐标轴的方向等 |
| MM_HIENGLISH：   | 以0.001英寸为逻辑单位长度、向右的方向为x轴正方向，向上的方向为y轴正方向  |
| MM_HIMETRIC：    | 以0.01毫米为逻辑单位长度、向右的方向为x轴正方向，向上的方向为y轴正方向   |
| MM_ISOTROPIC：   | 使用自定义的映射模式，这种映射模式在x方向和y方向上使用相同的单位长度。如果将映射模式设置为MM_ISOTROPIC，需要调用CDC类的成员函数SetWindowExt和SetViewportExt来设置其单位长                               |



	度、坐标轴的方向等
MM_LOENGLISH :	以0.01英寸为逻辑单位长度、向右的方向为x轴正方向，向上的方向为y轴正方向
MM_LOMETRIC :	以0.1毫米为逻辑单位长度、向右的方向为x轴正方向，向上的方向为y轴正方向
MM_TEXT :	以1设备像素为逻辑单位长度、向右的方向为x轴正方向，向下的方向为y轴正方向
MM_TWIPS :	以1/20磅(每一磅为1/72英寸)为逻辑单位长度、向右的方向为x正方向，向上的方向为y轴正方向

在上面的映射模式中，除了MM\_ANISOTROPIC、MM\_ISOTROPIC和MM\_TEXT以外，其它映射模式都使用向上的方向为y轴的正方向，而设备上下文的默认映射模式为MM\_TEXT，它在向下的方向为y轴的正方向，因此，如果我们在程序中将映射模式更改为其它的映射模式，需要注意应该随y轴的正方向的不同而更改图形输出函数所使用的坐标值的正负。

SetMapMode将映射模式设置为指定的映射模式，同时返回原有的映射模式。

下面我们来编写一个实用工具MappingConverter，该工具在不同的逻辑单位之间进行转换。需要注意的是，这种转换有两种不同的方式，即逻辑的还是物理的。如果当前映射模式为MM\_TEXT，这时一英寸所对应的像素值大小是一定的，我们称这种映射方式的逻辑的；如果当前映射模式为MM\_HIENGLISH、MM\_LOENGLISH等，则屏幕上的一英寸对应的像素值依赖于屏幕的实际分辨率，在这种模式下，800× 600的屏幕分辨率下一英寸所对应的像素值要比在640× 480的屏幕分辨率多。工具MappingConverter考虑到了这种差异，允许用户指定转换是基于逻辑英寸进行还是基于物理英寸进行。对于实际的转换过程，则通过调用CDC类的成员函数DPtoHIMETRIC来实现的，该成员函数将设备坐标值转换为相应的HIMETRIC度量。

由于使用的方便考虑，我们还在工具MappingConverter中添加了其它

几种度量单位的转换，这些度量单位包括：磅(point)、英寸(inch)、厘米(centimeter)、水平对话框单位(horizontal dialog units)和垂直对话框单位(vertical dialog units)。这里需要说明的是水平对话框单位和垂直对话框单位。这两种度量单位在对话框模板中用于对话框和控件的度量，此外，在编辑控件和列表框以及组合框控件设置制表位的函数SetTabStops也使用对话框单位。水平对话框单位等于当前系统字体的半角字符的平均宽度的1/4，而垂直对话框单位则等于当前系统字体的字符的高度的1/8。API函数GetDialogBaseUnits返回了当前系统所使用的对话框基本单位，由此可以导出当前使用的水平对话框单位和垂直对话框单位。但是要注意的是，实际使用的对话框单位依赖于当前对话框所使用的字体。CDialog类的成员函数MapDialogRect可以将一个以对话框单位表示的矩形转换为相应的屏幕像素单位。在工具MappingConverter，对话框单位是通过函数GetDialogBaseUnits的返回值计算得到的，也就是说，该单位是基于默认的系统字体，而不是对话框实际所选用的字体的。

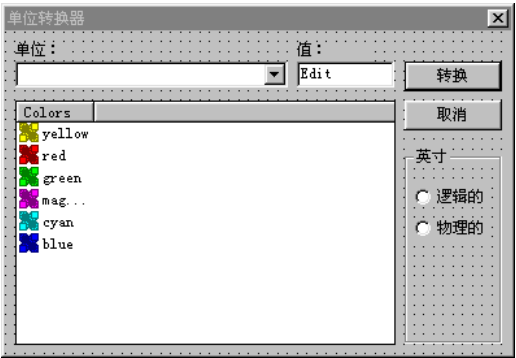


图9. 13 应用程序MappingConverter的主对话框

下面的步骤讲述了工具MappingConverter的创建。

1. 使用AppWizard创建一个基于对话框的MFC应用应用程序，按图修改应用程序的主对话框。

各主要控件的ID如表9.6所示。

表9. 6 控件属性列表

控件	ID	属性
“ 单位 ” 组合框	IDC_COMBO1	Type : Drop List Sort : 假

“ 值 ” 文本编辑框	IDC_EDIT1	
“ 转换 ” 按钮	IDOK	
列表控件	IDC_LIST1	View : Report Sort : None
“ 物理的 ” 单选钮	IDC_RADIO1	Group : 真
“ 逻辑的 ” 单选钮	IDC_RADIO2	

按图9.14设置各控件的Tab Order。

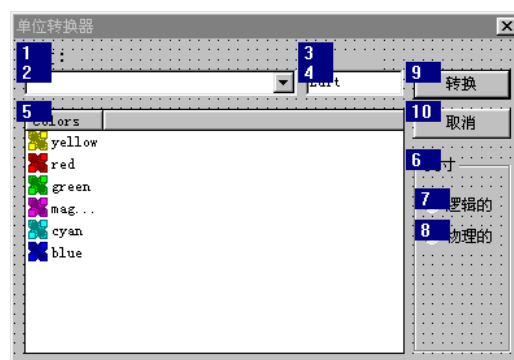


图9. 14 各控件的Tab Order

在组合框IDC\_COMBO1的属性对话框的Data选项卡内输入以下的列表项：

MM\_HIENGLISH (0.001英寸)

MM\_LOENGLISH (0.01英寸)

MM\_HIMETRIC (0.01毫米)

MM\_LOMETRIC (0.1毫米)

MM\_TEXT (象素)

MM\_TWIPS (缇)

磅

英寸

厘米

水平对话框单位（系统字体）

垂直对话框单位（系统字体）

注意各列表项的排列顺序，如果顺序出错，将导致在单位转换是进行不正确的换算，这也是将组合框的Sort属性设置为假的缘故。

按图9.15映射对话框的控件到类CMappingConverterDlg的成员变量。由于我们仅使用编辑框IDC\_EDIT1来输入数值，因此我们将它映射到类型为float的成员变量m\_fValue；同时，我们将单选按钮IDC\_RADIO1映射为类型为int的成员变量m\_nradio，当变量m\_nradio的值为0时表示单选钮“逻辑的”被选中，变量值为1时表示单选钮“物理的”被选中。

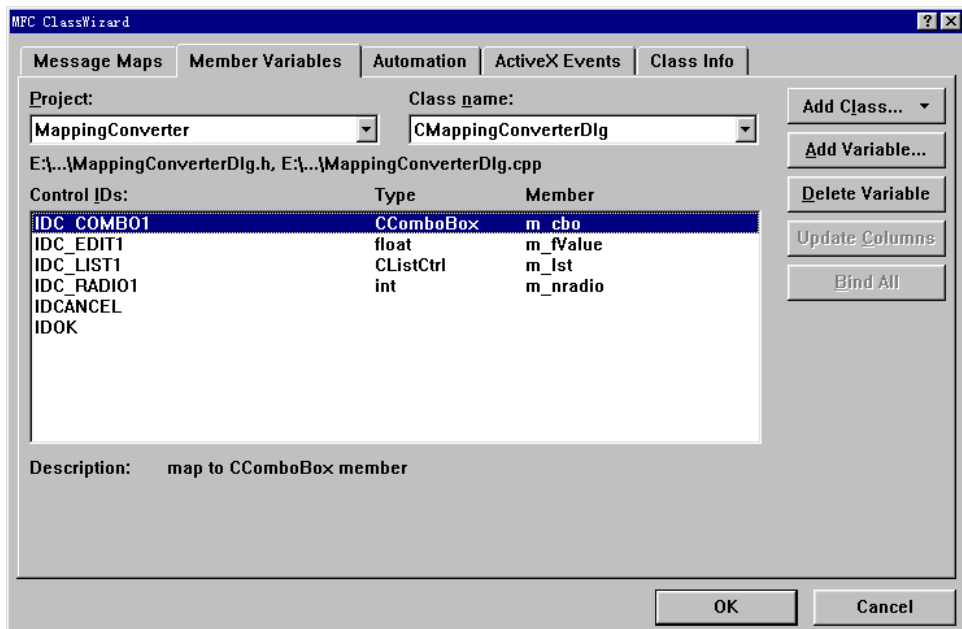


图9.15 映射成员变量到对话框控件

按下面的代码修改类CMappingConverterDlg的成员函数OnInitDialog：

```
////////////////////////////////////  
  
// CMappingConverterDlg 消息处理函数  
  
BOOL CMappingConverterDlg::OnInitDialog()  
{  
  
    // 设置默认选中的单选钮为“逻辑的”  
  
    m_nradio=0;
```

```

CDialog::OnInitDialog();

// 在系统菜单中添加“关于”菜单项

// IDM_ABOUTBOX 必须在系统命令的范围内

ASSERT((IDM_ABOUTBOX & 0xFFFF0) == IDM_ABOUTBOX);
ASSERT(IDM_ABOUTBOX < 0xF000);

CMenu* pSysMenu = GetSystemMenu(FALSE);

if (pSysMenu != NULL)
{
    CString strAboutMenu;

    strAboutMenu.LoadString(IDS_ABOUTBOX);

    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);

        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
}

// 设置对话框的图标。当应用程序的主窗口不是对话框时框架会自动进行这个过程

SetIcon(m_hIcon, TRUE); // Set big icon

SetIcon(m_hIcon, FALSE); // Set small icon

// TODO: 添加额外的初始化代码

// 在列表控件中添加两列：“单位”和“值”

m_lst.InsertColumn(0, "单位", LVCFMT_LEFT, 180, 0);

m_lst.InsertColumn(1, "值", LVCFMT_LEFT, 100, 1);

// 初始化列表控件中的项所用的字符串数组

char *szItemTexts[]={ "MM_HIENGLISH (0.001英寸)",
    "MM_LOENGLISH (0.01英寸)",

```

```

"MM_HIMETRIC (0.01毫米)",
"MM_LOMETRIC (0.1毫米)",
"MM_TEXT (像素)",
"MM_TWIPS (缇)",
"磅",
"英寸",
"厘米",
"水平对话框单位 (系统字体)",
"垂直对话框单位 (系统字体)";
// 初始化列表控件中的项
for (int i=0; i<11; i++)
{
m_lst.InsertItem(i, szItemTexts[i]);
}
// 设置组合框控件中的当前选择项为 MM_HIENGLISH
m_cbo.SetCurSel(0);
return TRUE; // 除非将输入焦点设置为某一控件，否则消息处理函数
// OnInitDialog 应该返回真值
}

```

为按钮IDOK的BN\_CLICKED通知消息添加命令处理函数OnOK如下：

```

void CMappingConverterDlg::OnOK()
{
// 获得当前在组合框中的选择项
int nCurSel=m_cbo.GetCurSel();
// 如果选择项不为空...
if (nCurSel!=LB_ERR)

```

```

{
// 更新映射到控件的成员变量

UpdateData();

// 获得对话框的当前设备上下文，该设备上下文用于不同映射模式的单位转换

CClientDC dc(this);

if (m_nradio==0)

{
// 如果选择了“逻辑英寸”，则将映射模式设置为 MM_TEXT

dc.SetMapMode(MM_TEXT);

}

else

{
// 如果选择了“物理英寸”，则将映射模式设置为 MM_HIENGLISH

dc.SetMapMode(MM_HIENGLISH);

}

// 获得每英寸长度的像素值

CSize sz(10000, 10000);

dc.DPtoHIMETRIC(&sz);

double fPixelPerInch=10000./sz.cx*100*25.4;

// 获得用户需要进行转换的单位值

double fValue=m_fValue;

// 获取当前对话框基本单位，其高位字为垂直对话框基本单位，

// 低位字为水平对话框基本单位

long lDbUnits=GetDialogBaseUnits();

// nType 为 1 表示经过初步转换的单位为英寸，

// nType 为 2 表示经过初步转换的单位为像素

```

```
int nType;

// 根据用户所选择的当前单位对输入的值进行初步的转换

switch (nCurSel)

{

case 0: // MM_HIENGLISH

nType=1;

fValue/=1000;

break;

case 1: // MM_LOENGLISH

nType=1;

fValue/=100;

break;

case 2: // MM_HIMETRIC

nType=1;

fValue/=2540;

break;

case 3: // MM_LOMETRIC

nType=1;

fValue/=254;

break;

case 4: // MM_TEXT

nType=2;

break;

case 5: // MM_TWIPS

nType=1;

fValue/=1440;
```



```
break;

case 6: // 磅

nType=1;

fValue/=72;

break;

case 7: // 英寸

nType=1;

break;

case 8: // 厘米

nType=1;

fValue/=2.54;

break;

case 9: // 水平对话框单位

nType=2;

fValue=fValue/4.*LOWORD(IDbUnits);

break;

case 10: // 垂直对话框单位

nType=2;

fValue=fValue/8.*HIWORD(IDbUnits);

break;

}

// 列表控件中第二列的输出项对应的字符串

CString str[11];

// 根据初步转换得到的结果生成在列表控件中给出的转换后的值

switch (nType)

{
```

```
case 1: // 英寸

{
// 得到对应的像素值

double fPixel=fValue*fPixelPerInch;

str[0].Format("%g", fValue*1000);

str[1].Format("%g", fValue*100);

str[2].Format("%g", fValue*25.4*100);

str[3].Format("%g", fValue*25.4*10);

str[4].Format("%g", fPixel);

str[5].Format("%g", fValue*72.*20);

str[6].Format("%g", fValue*72);

str[7].Format("%g", fValue);

str[8].Format("%g", fValue*2.54);

str[9].Format("%g", fPixel*4./LOWORD(IDbUnits));

str[10].Format("%g", fPixel*8./HIWORD(IDbUnits));

break;

}

case 2: // 像素

{

// 得到对应的英寸值

double fInch=fValue/fPixelPerInch;

str[0].Format("%g", fInch*1000);

str[1].Format("%g", fInch*100);

str[2].Format("%g", fInch*25.4*100);

str[3].Format("%g", fInch*25.4*10);

str[4].Format("%g", fValue);
```

```

str[5].Format("%g", fInch*72*20);

str[6].Format("%g", fInch*72);

str[7].Format("%g", fInch);

str[8].Format("%g", fInch*2.54);

str[9].Format("%g", fValue*4./LOWORD(IDbUnits));

str[10].Format("%g", fValue*8./HIWORD(IDbUnits));

break;

}

}

// 将转换后的值写到列表控件中对应项的第二列中

for (int i=0; i<11; i++)

{

m_lst.SetItem(i, 1, LVIF_TEXT, str[i], 0, 0, 0, LPARAM(0));

}

}

}

```

编辑并运行应用程序MappingConverter，其结果如图9.16所示。



图9.16 示例程序MappingConverter的运行结果

## 9.5.2 自定义的坐标系统

在很多情况下使用系统预定义的映射系统非常的恰当，但并非总是这样。假设我们需要编写一个绘制函数图形的应用程序，把整个屏幕映

射为一个平面直角坐标系会更加的恰当。在本节的后面我们举了一个这样的例子。

前面提到过，使用下面的函数调用可以自定义设备上下文对象dc的坐标映射。

```
dc.SetMapMode(MM_ANISOTROPIC);
```

但是，在使用了上面的代码之后，还需要具体的指定自定义坐标系统在x方向和y方向上的单位长度、坐标轴正方向以及视口左上角的位置在坐标映射中所对应的坐标值等。这个操作是通过调用CDC类的成员函数SetWindowExt和SetViewportExt来实现的。

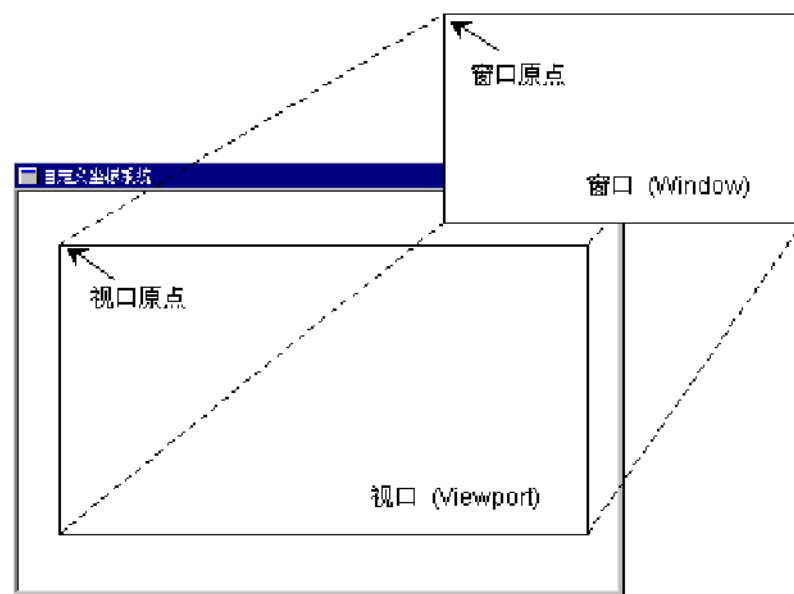


图9. 17 设备上下文中窗口和视口的关系

图9.17说明了在设备上下文中窗口(Window)和视口(ViewportExt)的关系。视口度量的设置以设备象素为单位，而窗口度量的设置则几乎的任意的。所谓的自定义坐标系统即是将由窗口定义的坐标系统映射到由视口定义的设备上下文中的区域。

- 注意：
- 上面的术语“窗口”并不是指Windows中的窗口对象，而是在设备上下文中进行坐标映射时使用的一个抽象概念。在本节中，具有上述意义的术语“窗口”有可能和通常意义中的Windows术语“窗口”混用，读者应根据具体的上下文明确其含义。

在调用SetViewportExt设置视口之前必须先调用SetWindowExt设置窗口。

函数SetWindowExt可以使用两个整型量作为其参数，也可以使用一个CSize对象作为其参数。它指定了窗口的横向度量和纵向度量。当所指定的横向度量为负值时，表示x轴的正方向向左；当所指定的纵向度量为负值时，表示y轴的正方向向上。

同时还可以使用函数SetWindowOrg来设置窗口左上角对应的坐标。这样就可以实现坐标平面原点的平移。

在设置了窗口之后，我们还需要调用SetViewportExt函数设置视口。

函数SetViewportExt使用了与SetWindowExt相类似的参数。它们指定了视口的横向度量和纵向度量。类似的，我们还可以使用函数SetViewportOrg来设置视口的左上角的屏幕坐标。这里需要注意的是，在函数SetViewportExt和SetViewportOrg中所使用的参数的量值是以设备象素为单位的。

上面所提到的这些函数都是CDC类的成员函数。

下面所给的示例程序FuncGraphy输出一个函数[示例中是 $\sin\left(\frac{1}{x^3}\right)$ ]的图象。为了方便起见，我们使用了自定义的坐标系统。在该坐标系统中，原点的窗口客户区中的中心，横坐标的范围为-31.006到31.006，在这个范围内恰好包括函数的一个完整的图象范围。其纵坐标的范围为-1.2到1.2。

为了节省篇幅，我们这里仅给出应用程序主窗口的OnPaint函数：

```
// 应用程序主窗口的重绘函数

void CMyWnd::OnPaint()
{
    // 获得窗口的客户区设备上下文句柄
    CPaintDC dc(this);

    // 设置映射模式为 MM_ANISOTROPIC
    dc.SetMapMode(MM_ANISOTROPIC);

    // 设置窗口左上角的坐标为
    dc.SetWindowOrg(-31006, 1200);

    // 设置窗口度量
```

```

dc.SetWindowExt(int(2000*31.006), -2400);

// 获得客户区矩形

CRect rc;

GetClientRect(rc);

// 设置视口左上角的坐标

dc.SetViewportOrg(0, 0);

// 设置视口度量

dc.SetViewportExt(rc.Width(), rc.Height());

// 创建蓝色实线画笔

CPen pen(PS_SOLID, 1, RGB(0, 0, 255)), *pOldPen;

pOldPen=dc.SelectObject(&pen);

// 创建蓝色斜线刷子

CBrush br(HS_BDIAGONAL, RGB(0, 0, 255)), *pOldBrush;

pOldBrush=dc.SelectObject(&br);

// 开始一个路径

dc.BeginPath();

dc.MoveTo(-31006, 0);

for (double x=-31.006; x<=0; x+=0.02)

{

dc.LineTo(int(1000*x), int(1000*sin(-pow(-x, 1./3))));

}

for (x=0; x<=31.006; x+=0.02)

{

dc.LineTo(int(1000*x), int(1000*sin(pow(x, 1./3))));

}

dc.LineTo(3142, 0);

```

```
dc.MoveTo(0, -1200);  
  
dc.LineTo(0, 1200);  
  
dc.EndPath();  
  
// 绘制路径并填充路径的内部  
  
dc.StrokeAndFillPath();  
  
// 恢复设备上下文原有的画笔和刷子  
  
dc.SelectObject(pOldPen);  
  
dc.SelectObject(pOldBrush);  
  
}
```

完成并编译上面的程序，其运行结果如图所示。

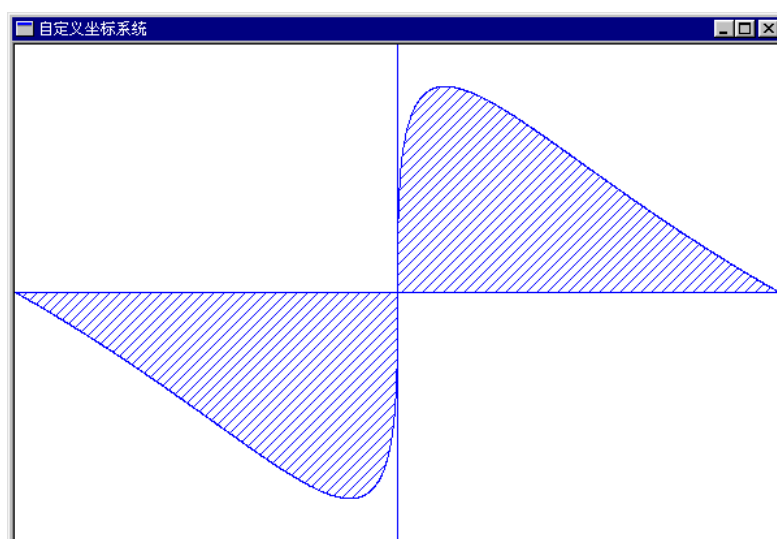


图9. 18 示例程序FuncGraphy - 使用自定义的坐标系

- 技巧：

- 下面我们讨论在应用程序FuncGraphy中所用到的一些技巧。首先，我们不是直接使用LineTo来一段一段的绘制曲线，而是先创建一个路径，然后再来填充该路径。这至少有两个好处：一是如果曲线需要使用如虚线等线型时，如果还一段一段的使用LineTo来绘制的话，事实上不可能得到虚线(读者可以自行验证这一点)，这时我们只能先创建一个路径，再使用StrokePath之类的函数来绘制出这个路径；第二，如果使用路径的话，我们还可以使用当前刷子来填充曲线的内部，如示例代码中所示。

- 在自定义坐标系时，一定要记住一点，这就是所有的图形输出函数在指定输出位置的坐标时都使用了整型参量。这说明一点，我们必须使坐标系统的单位长度具有一定的区分度。简单的说，坐标系统的单位长度应该小于一个设备像素的大小，否则你将不能通过该坐标系准确的定位到输出设备上的每一个设备像素。如果坐标系统的单位长度过大，一个很常见的结果就是导致图形输出发生失真和变形。读者可以使用上面的示例程序FuncGraphy直接验证这一点。一般来说，对于屏幕输出设备，我们只需要使横坐标和纵坐标的跨度在2000以上，就可以有效的避免这个问题。同样的，考虑示例程序FuncGraphy，在绘制输出图形时考虑合理的步长也是很重要的。步长大小将会导致程序进行过多的不必要运算，从而明显的降低曲线的绘制速度，而太大的步长又会导致曲线不够精确。



# 第十章 MFC通用类

MFC不仅提供了大量的用于编写图形用户界面的类，它也包含了许多通用类用于处理字符串、列表、数组、日期和时间，有了这些通用类，编写程序时就可以避免使用复杂的数据结构。例如，由于MFC的数组类能够自动的改变大小，我们在不知道数组维数的情况下就不必使用一个大数组，这样就可以节约内存，提高程序的运行速度。

本章主要涉及以下内容：

- 数组类
- 列表类
- 映射类
- 字符串类
- 日期和时间类

## 第一节 数组类

MFC的数组类使你可以创建和操作一个实际上可以处理各种数据类型的一维数组对象。除了MFC可以在运行时动态的增大和缩小数组对象外，这些数组对象非常象常规的数组。这也意味者在声明数组对象时不必关心它的维数。由于MFC可以动态的变大和变小，你不必考虑使用常规数组时出现的内存浪费。使用常规数组时你必须将其定义成能够容纳所有可能需要的元素，而不管这些元素是否真的被会使用。

MFC的数组类包含CByteArray、CDWordArray、CPtrArray、CUIIntArray、CWordArray和CStringArray。从这些类的名称可以看出，每一个类都被设计成能够处理一个特定的数据类型。例如，在本节例子中将要用到的CUIIntArray类是一个处理无符号整形数的数组类，而CObjArray类代表对象数组类。这些数组类几乎相同，仅仅的区别在于它们储存的数据类型不同。如果你学会使用其中的一种数组类的使用，你就学会了所有数组类的使用。

数组类有下列成员函数：

Add ( )

在数组的最后追加一个元素，可以根据需要增大数组大小。

ElementAt ( )

获得一个指向数组元素的指针

FreeExtra ( )

释放不用的数组内存

GetAt ( )

获取数组内指定位置处的值

GetSize ( )

获取数组中包含的元素个数

GetUpperBound ( )

获取数组的上界值。

InsertAt ( )

在数组的指定位置处插入一个元素，后面的元素的下标加1。

RemoveAll ( )

删除数组中所有的元素。

SetAt ( )

设定数组指定位置处的值。因为制革函数不会增加数组的大小，故这个下标此时一定要有效。

SetAtGrow ( )

设定数组的指定位置处的值，可以根据需要增大数组的大小。

SetSize ( )

设置数组的初始大小。

下面将介绍一个数组的程序，这个程序可以让你测试一下MFC的数组类。

首先，这个程序在View类中声明一个数组对象，如下：

```
CUIntArray array;
```

接着，在View类的构造函数中初始化数组，将其设置成包含十个元素，

```
array.SetSize(10, 5);
```

SetSize（）函数有两个参数，第一个参数是数组的初始大小，第二个参数是数组元素每次增加时增加的个数。

在设置完数组的大小之后，程序等待用户在窗口中单击鼠标左键或右键。如果用户这样做了，程序将显示一个合适的对话框并且处理输入到对话框中的数据。下面的代码是该程序的OnLButtonDown（）函数，用于处理用户单击右键的事件。

```
void CArrayView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
```

```
CArrayAddDlg dialog(this);
```

```
dialog.m_index = 0;
```

```
dialog.m_value = 0;
```

```
dialog.m_radio = 0;
```

```
int result = dialog.DoModal();
```

```
if (result == IDOK)
```

```
{
```

```
if (dialog.m_radio == 0)
```

```
array.SetAtGrow(dialog.m_index, dialog.m_value);
```

```
else if (dialog.m_radio == 1)
```

```
array.InsertAt(dialog.m_index, dialog.m_value, 1);
```

```
else
```

```
array.Add(dialog.m_value);
```

```
Invalidate();
```

```

}

CView::OnLButtonDown(nFlags, point);

}

```

这段代码开始先创建一个对话框对象并初始化。如果用户选择对话框的“确定”按钮，OnLButtonDown()函数将检查对话框的成员变量m\_radio。值为0表示第一个单选按钮（设置）被选中，值为1表示第二个单选按钮（插入）被选中，值为2表示第三个单选按钮（添加）被选中。

如果用户希望设置数组的元素，程序将调用SetAtGrow()函数，它需要两个参数，一个是数组元素的下标值，另一个是希望设置的值。它不象常规的SetAt()函数，用户必须使用当前有效的数组下标值。SetAtGrow()为了设置指定位置处的值将根据需要增大数组的大小。

当用户选择了“插入”单选按钮，程序将调用InsertAt()函数，它需要两个参数，一个是要插入的数组元素的下标，另一个是该数组元素的值。这将在指定位置处创建一个新的数组元素，并将把后面的元素往后推。最后当用户选择了“添加”按钮，程序将调用Add()函数，这将在数组的后面添加一个元素。对Invalidate()的调用将使程序重新显示数据。

下面的OnDraw()函数读取并显示数组。

```

void CArrayView::OnDraw(CDC* pDC)
{
    CArrayDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Get the current font's height.

    TEXTMETRIC textMetric;

    pDC->GetTextMetrics(&textMetric);

    int fontHeight = textMetric.tmHeight;

    // Get the size of the array.

    int count = array.GetSize();
}

```

```

int displayPos = 10;

// Display the array data.
for (int x=0; x<count; ++x)
{
    UINT value = array.GetAt(x);

    char s[81];

    wsprintf(s, "Element %d contains the value %u.", x, value);

    pDC->TextOut(10, displayPos, s);

    displayPos += fontHeight;
}
}

```

这里，程序首先获得当前字体的高度，然后程序通过调用GetSize（）函数获得数组元素的个数。最后，使用数组元素的个数控制一个for循环，调用GetAt（）函数获得当前下标处的数组元素的值。为了显示程序将数组元素的值转化成字符串。

程序的OnRButtonDown（）函数用来响应用户按下鼠标右键的事件，此函数用来处理删除数组元素的任务。下面是该函数的代码：

```

void CArrayView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CArrayRemoveDlg dialog(this);

    dialog.m_radio= 1;

    dialog.m_index= 0;

    int result = dialog.DoModal();

    if (result == IDOK)
    {
        if (dialog.m_radio==0)

```

```

array.RemoveAll();

else

array.RemoveAt(dialog.m_index);

Invalidate();

}

CView::OnRButtonDown(nFlags, point);

}

```

在这个函数中，当显示完对话框后，程序检查对话框的成员变量 `m_removeAll`。如果这个值为真意味着用户希望删除数组中所有的元素。这种情况下，程序调用数组类的成员函数 `RemoveAll()`。否则，程序将调用 `RemoveAt()` 删除指定位置处的数组元素。最后调用 `Invalidate()` 函数刷新数据显示。

下面将介绍对话框类 `CArrayAddDlg`，按照下面的步骤创建这个对话框类。

1. 创建如图10.1所示的对话框，3个单选按钮的ID分别为 `IDC_ADD0`，`IDC_ADD1`，`IDC_ADD2`。

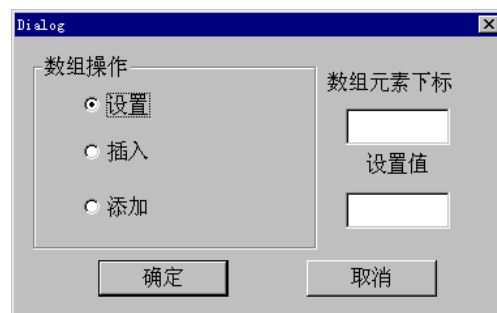


图10.1 添加元素对话框

2. 在ClassWizard中为两个文本框映射两个UINT类型的变量 `m_index`，`m_value`。
3. 给该对话框添加一个UINT类型的成员变量 `m_radio`。
4. 为 `WM_INITDIALOG` 添加函数 `OnInitDialog()`，在其中设置单选按钮的初始状态。

```

BOOL CArrayAddDlg::OnInitDialog()

```

```

{
CDialog::OnInitDialog();

// TODO: Add extra initialization here

((CButton*)GetDlgItem(IDC_ADD0))->SetCheck(1);

return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE
}

```

## 5. 添加三个函数用以响应用户单击单选按钮。

```

void CArrayAddDlg::OnAdd0()

{
// TODO: Add your control notification handler code here

GetDlgItem(IDC_INDEX)->EnableWindow(true);

}

void CArrayAddDlg::OnAdd1()

{
// TODO: Add your control notification handler code here

GetDlgItem(IDC_INDEX)->EnableWindow(true);

}

void CArrayAddDlg::OnAdd2()

{
// TODO: Add your control notification handler code here

GetDlgItem(IDC_INDEX)->EnableWindow(false);

}

```

为了确定用户选择了哪一个单选按钮，重载CDialog::OnOK()。

```

void CArrayAddDlg::OnOK()

{

```

```

// TODO: Add extra validation here

UINT nRadio=GetCheckedRadioButton(IDC_ADD0, IDC_ADD2);

switch(nRadio)
{
case IDC_ADD0:

m_radio=0;

break;

case IDC_ADD1:

m_radio=1;

break;

case IDC_ADD2:

m_radio=2;

break;

default:

break;

}

CDialog::OnOK();
}

```

按照下面的步骤创建另一个对话框类CRemoveDlg。

1. 创建如图10.2所示的对话框，两个单选按钮的ID分别是IDC\_REMOVE0和IDC\_REMOVE1。

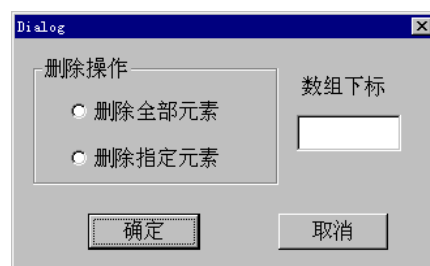


图10. 2 删除数组元素对话框



2. 在ClassWizard中为文本框映射一个UINT类型的变量m\_index。
3. 给该对话框添加一个UINT类型的成员变量m\_radio。
4. 为WM\_INITDIALOG添加函数OnInitDialog()，在其中设置单选按钮的初始状态。

```
BOOL CArrayRemoveDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here

    ((CButton*)GetDlgItem(IDC_REMOVE1))->SetCheck(1);

    return TRUE; // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
```

5. 添加两个函数用以响应用户单击单选按钮。

```
void CArrayRemoveDlg::OnRemove0()
{
    // TODO: Add your control notification handler code here

    GetDlgItem(IDC_INDEX)->EnableWindow(false);
}

void CArrayRemoveDlg::OnRemove1()
{
    // TODO: Add your control notification handler code here

    GetDlgItem(IDC_INDEX)->EnableWindow(true);
}
```

6. 为了确定用户选择了哪一个单选按钮，重载CDialog::OnOK()。

```
void CArrayRemoveDlg::OnOK()
{
```

```

// TODO: Add extra validation here

UINT nRadio=GetCheckedRadioButton(IDC_REMOVE0,IDC_REMOVE1);

switch(nRadio)

{

case IDC_REMOVE0:

m_radio=0;

break;

case IDC_REMOVE1:

m_radio=1;

break;

default:

break;

}

CDialog::OnOK();

}

```

现在编译并运行这个程序，首先在窗口中显示一个有十个元素的数组，如图10.3所示。单击左键弹出如图10.1所示的对话框，你可以选择三种数组操作：设置、插入和添加。单击右键弹出如图10.2所示的对话框，你可以选择两种删除操作：删除全部元素和删除指定元素。

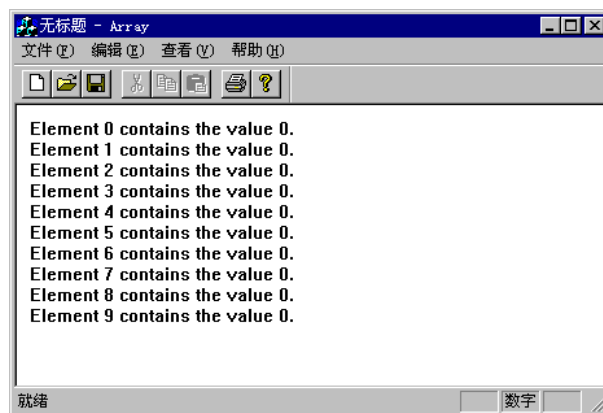


图10. 3 程序运行初始窗口

## 第二节 列表类

列表类象是有特殊功能的数组。列表的元素被称为节点。列表使用指针来连结它的节点。如果你希望快速的插入和删除数组元素，列表类是一个比较好的选择。但是在列表中查找一个元素要比在数组中慢，因为列表需要按照指针顺序从一个节点到另外一个节点。

通常我们称列表中第一个节点为列表的头，列表中最后一个节点是列表的尾。

列表类有以下成员函数：

`Clist ( )`

`Clist`类的构造函数，其中的参数指定分配内存的基本单元。

`GetHead( )`

获得列表的第一个元素的值。

`GetTail( )`

获得列表的最后一个元素的值。

`RemoveHead( )`

删除列表中第一个元素

`RemoveTail( )`

删除列表中最后一个元素。

`AddHead ( )`

在列表的头部添加一个节点，使这个节点成为列表的新的头。

`AddTail ( )`

在列表的尾部添加一个节点，使这个节点成为列表的新的尾。

`RemoveAll()`

删除节点中所有的元素。

GetHeadPosition( )

获得列表的头节点的位置。

GetTailPosition( )

获得列表中尾节点的位置。

GetNext( )

获得指定位置下一个节点处的值。

GetPrev( )

获得指定位置上一个节点处的值。

GetAt( )

获得指定位置处节点的值。

SetAt( )

设置指定位置处节点的值。

RemoveAt( )

删除指定位置处的节点。

InsertBefore( )

在指定位置的前面插入一个节点。

InsertAfter( )

在指定位置的后面插入一个节点。

Find( )

按照列表顺序搜索给定的对象指针，返回一个POSITION类型的量。

FindIndex( )

按照列表顺序搜索指定的下标。

GetCount( )

获得列表中包含的节点个数。

IsEmpty()

检查一个列表是否不含有任何节点。

下面的程序将允许用户添加和删除节点，按照以下步骤进行：

1. 使用MFC AppWizard创建一个单文档应用程序List。
2. 添加一个对话框类CAddStudentDlg，其对应的对话框如图10.4所示。



图10. 4 添加节点对话框

3. 为两个文本框映射两个变量m\_name，m\_score。
4. 添加一个对话框类CRemoveStudentDlg，其对应的对话框如图10.5所示。两个单选按钮的ID为IDC\_REMOVE0和IDC\_REMOVE1。



图10. 5 删除节点对话框

5. 为对话框类添加一个UINT类型的成员变量m\_radio。
6. 在OnInitDialog()函数中设置单选按钮的初始状态。

```
BOOL CRemoveStudentDlg::OnInitDialog()
```

```
{
```

```
CDialog::OnInitDialog();
```

```
// TODO: Add extra initialization here

((CButton*)GetDlgItem(IDC_REMOVE0))->SetCheck(1);

return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE

}
```

7. 为了知道用户选择了哪一个单选按钮，在OnOk()函数中添加下面的代码。

```
void CRemoveStudentDlg::OnOK()

{
// TODO: Add extra validation here

UINT nRadio=GetCheckedRadioButton(IDC_REMOVE0, IDC_REMOVE1);

switch(nRadio)

{

case IDC_REMOVE0:

m_radio=0;

break;

case IDC_REMOVE1:

m_radio=1;

break;

default:

break;

}

CDialog::OnOK();

}
```

8. 在ListView.h中CListView类的声明之前添加如下代码，用来定义一个结构体CStudent，包含两个变量m\_name和m\_score，分别用于存放学生的姓名和成绩。

```

struct CStudent
{
CString m_name;
int m_score;
};

```

9. 为CListView添加一个类型为CptrList的成员变量m\_list。

10. 在ListView.cpp中添加下列语句：

```

#include "AddStudentDlg.h"

#include "RemoveStudentDlg.h"

```

11. 当用户单击左键后，弹出如图10.4所示的对话框，可以添加一个节点。对应的OnLButtonDown（）函数代码如下：

```

void CMyListView::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default

CAddStudentDlg dialog;

dialog.m_name = "";

dialog.m_score = 0 ;

// Display the dialog box.

int result = dialog.DoModal();

if (result == IDOK)
{
// Create and initialize the new node.

CStudent* m_pStudent = new CStudent;

m_pStudent->m_name = dialog.m_name;

m_pStudent->m_score = dialog.m_score;

// Add the node to the list.

```

```

m_list.AddTail(m_pStudent);

// Repaint the window.

Invalidate();

}

CView::OnLButtonDown(nFlags, point);

}

```

12. 当用户单击右键后，弹出如图10.5所示的对话框，可以删除一个节点。对应的OnRButtonDown（）函数代码如下：

```

void CMyListView::OnRButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default

CRemoveStudentDlg dialog;

dialog.m_radio = 0;

// Display the dialog box.

int result = dialog.DoModal();

// If the user clicked the OK button...

if (result == IDOK)
{
CStudent* m_pStudent=new CStudent;

// Make sure the list isn't empty.

if (m_list.IsEmpty())

MessageBox("节点已经全部删除!");

else

{

// Remove the specified node.

if (dialog.m_radio == 0)

```



```

m_pStudent = (CStudent*)m_list.RemoveHead();

else

m_pStudent = (CStudent*)m_list.RemoveTail();

// Delete the node object and repaint the window.

delete m_pStudent;

Invalidate();

}

}

CView::OnRButtonDown(nFlags, point);

}

```

### 13. 最后设置OnDraw ( ) 函数用来响应Invalidate()。

```

void CMyListView::OnDraw(CDC* pDC)

{

CListDoc* pDoc = GetDocument();

ASSERT_VALID(pDoc);

// TODO: add draw code for native data here

TEXTMETRIC textMetric;

pDC->GetTextMetrics(&textMetric);

int fontHeight = textMetric.tmHeight;

// Initialize values used in the loop.

POSITION pos = m_list.GetHeadPosition();

int displayPosition = 10;

// Iterate over the list, displaying each node's values.

while (pos != NULL)

{

CStudent* m_pStudent = (CStudent*)m_list.GetNext(pos);

```

```

char s[81];

wsprintf(s, " 的成绩是 %d.",m_pStudent->m_score);

CString m_string=m_pStudent->m_name+s;

pDC->TextOut(10, displayPosition, m_string);

displayPosition += fontHeight;

}

}

```

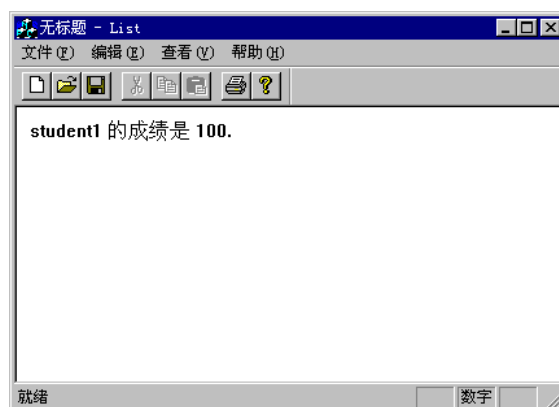


图10. 6 程序运行的初始窗口

#### 14. 最后在CListView的析构函数中删除数组中所有的节点。

```

CMyListView::~CMyListView()
{
while (!m_list.IsEmpty())
{
CStudent* m_pStudent = (CStudent*)m_list.RemoveHead();
delete m_pStudent;
}
}

```

现在编译并运行这个程序，当程序运行后，弹出如图10.6所示的窗口。单击左键，弹出如图10.4所示的对话框来添加节点。单击右键，弹出如图10.5所示的对话框来删除节点。如果在没有节点删除节点，将弹出如图10.7所示的对话框提示用户节点已经全部删除。



图10. 7 没有节点时删除节点弹出的消息框

### 第三节 映射类

这个类用于创建关键对象和数值对象联系的集合。你可以使用MFC的映射类创建查询表格。MFC的映射类包含CMapPtrToPtr, CMapPtrToWord, CMapStringToOb, CMapStringToPtr, CMapStringToString, CMapWordToOb, and CmapWordToPtr.在类的名称中第一个数据类型是关键字的数据类型, 第二个数据类型是对应的数值的数据类型。

映射类有下列成员函数：

Lookup()

查询映射到指定关键字的值。

SetAt()

向映射中插入一个元素, 如果指定的关键字存在, 替换掉原来的元素。

operator [ ]

向映射中插入一个元素, 其作用和SetAt()相同。

RemoveKey ( )

查询符合关键字的映射。如果发现, 则删除这个元素。

RemoveAll( )

删除映射中所有的元素。

GetStartPosition( )

获得映射中第一个元素的位置。映射中第一个元素是不预知的, 所以映射的第一个元素实际上没有特定的意义。一般将这个值传递给GetNextAssoc ( ) 函数。

GetNextAssoc ( )

获得映射中指定位置处下一个元素。

GetCount ( )

获得映射中元素的个数。

IsEmpty ( )

测试这个数组元素是否为空。

现在用一个查询程序来使你对映射类有一个更深入的了解。按照下面的步骤创建这个程序。

1. 创建一个单文档的应用程序Map。
2. 添加如图10.8所示的对话框，并生成基于这个对话框的类CLookupMapDlg,并为文本框添加一个变量m\_key。

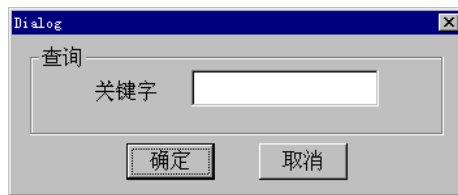


图10. 8 查询表格

3. 为CMapView添加一个CMapStringToString类型的成员变量m\_map。
4. 在CMapView的构造函数中添加下列代码初始化映射。

```
CMapView::CMapView()
{
    // TODO: add construction code here

    m_map.SetAt("red", "红色");
    m_map.SetAt("yellow", "黄色");
    m_map.SetAt("green", "绿色");
    m_map.SetAt("blue", "蓝色");
    m_map.SetAt("white", "白色");
}
```

```
m_map.SetAt("black", "黑色");  
}
```

5. 在MapView.cpp中添加下列语句：

```
#include "LookUpMapDlg.h"
```

6. 为WM\_LbuttonDown消息添加消息处理函数OnLButtonDown(), 代码如下：

```
void CMapView::OnLButtonDown(UINT nFlags, CPoint point)  
{  
    // TODO: Add your message handler code here and/or call default  
    CLookUpMapDlg dialog(this);  
    dialog.m_key = "";  
    // Display the dialog box.  
    int result = dialog.DoModal();  
    if (result == IDOK)  
    {  
        // Look for the requested value.  
        CString m_value;  
        BOOL m_bFound = m_map.Lookup(dialog.m_key, m_value);  
        if (m_bFound)  
            MessageBox(m_value);  
        else  
            MessageBox("未发现匹配字符串");  
    }  
    CView::OnLButtonDown(nFlags, point);  
}
```

7. 在OnDraw()函数中添加下列代码，在视图中显示映射中所有的元

素。

```
void CMapView::OnDraw(CDC* pDC)
{
    CMapDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here

    TEXTMETRIC textMetric;
    pDC->GetTextMetrics(&textMetric);
    int fontHeight = textMetric.tmHeight;
    int displayPosition = 10;
    POSITION pos =m_map.GetStartPosition();
    CString m_key;
    CString m_value;
    int m_index=m_map.GetCount();
    for(int i=0;i<m_index;i++)
    {
        m_map.GetNextAssoc(pos, m_key, m_value);
        CString m_str=m_key+"的意思是"+m_value;
        pDC->TextOut(10, displayPosition,m_str);
        displayPosition += fontHeight;
    }
}
```

现在编译并运行程序，首先出现如图10.9所示的主窗口，映射中每一个元素都显示在

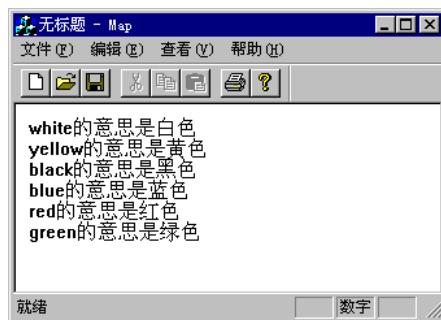


图10. 9 程序运行初始窗口

其中。单击左键弹出如图10.8所示的对话框，用户在其中输入要查询的关键字，选择“确定”按钮，如果映射中存在此关键字，将弹出如图10.10所示的对话框告知用户关键字对应的中文意思。如果映射中不存在此关键字，弹出如图10.11所示的对话框告知用户未发现匹配字符串。



图10. 10 查询结果显示对话框

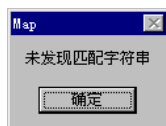


图10. 11 未发现匹配字符串

## 第四节 字符串类

MFC中的CString类使用C++操作字符串和使用Basic或Pascal一样便捷。CString类没有基类。一个CString类的对象由一个长度可变的字符序列组成。CString包含很多成员函数用来操作字符串。

CString主要包含以下成员函数：

CString()

CString类的构造函数，用来创建一个CString类的对象。

GetLength()

获取CString类的对象包含的字符串的长度（字节数）。

IsEmpty()

测试CString类的对象包含的字符串是否为空。

Empty()

使CString类的对象包含的字符串为空字符串。

GetAt()

获得字符串指定位置处的字符。

operator [ ]

获得字符串指定位置处的字符，作用和GetAt()一样。

SetAt()

设定字符串指定位置处的字符。

operator LPCTSTR

返回指向储存在CString类的对象内的字符的指针。

operator =

将一个新的值赋予CString对象。

operator +

将两个字符串合并成一个新的字符串。

operator +=

在一个字符串的后面再添加一个字符串。

Compare ( )

比较两个字符串。

CompareNoCase ( )

在忽略字符大小写的情况下比较两个字符串。

Mid()



从CString类对象包含的字符串中提取指定开头和结尾的字符串，类似于Basic中的MID\$函数。

Left()

获取字符串左边指定长度的字符串，类似于Basic中的LEFT\$函数。

Right()

获取字符串右边指定长度的字符串，类似于Basic中的RIGHT\$函数。

SpanIncluding()

从字符串中提取包含在指定字符数组内的字符的子串。

SpanExcluding()

从字符串中提取不包含在指定字符数组内的字符的子串。

MakeUpper()

将字符串中所有的字符全部转化成大写形式。

MakeLower()

将字符串中所有的字符全部转化成小写形式。

MakeReverse()

将字符串倒置。

Format()

象sprintf()函数一样格式化字符串。

TrimLeft()

删除字符串左边开头的空白字符。

TrimRight()

删除字符串右边结尾的空白字符。

FormatMessage()

格式化消息字符串。

Find()

在字符串中查找指定的字符或字符串。

ReverseFind()

返回字符串中最后一次和指定的字符匹配的字符的下标。

FindOneOf()

在字符串中查找第一个和指定的字符匹配的字符。

GetBuffer()

获得指向CString对象内字符的指针。

GetBufferSetLength()

获得指向CString对象内字符的指针，但是只能截取指定长度的字符。

ReleaseBuffer()

释放在缓冲区内由GetBuffer()函数返回的字符串。

LockBuffer()

复制字符串，并将其锁入缓冲区。

UnlockBuffer()

将调用LockBuffer()函数锁入缓冲区的字符串解锁。

LoadString()

从一个Windows资源加载一个已经存在的CString对象。

下面用几个例子来上读者体会一个CString类的好处。

示例1：连结字符串

代码如下：

```
CString m_str1="工作";  
CString m_str2="正常";  
CString m_str3=m_str1+m_str2;  
AfxMessageBox(m_str3);
```

运行结果如图10.12所示。



图10. 12 连结字符串

## 示例2：比较字符串

代码如下：

```
CString m_str1="a";  
CString m_str2="b";  
int result=m_str1.Compare(m_str2);  
if(result=0)  
AfxMessageBox("两者相同");  
else if(result>0)  
AfxMessageBox("m_str1大于m_str2");  
else  
AfxMessageBox("m_str1小于m_str2");
```

运行结果如图10.13所示。两个字符串比较大小时从第一个字母开始，按照对应的ASCII值比较。如果第一个字母相同，再比较下一个字母。依次往下直到比较出大小为止。



图10. 13 比较字符串

### 示例3：提取字符串

代码如下：

```
CString m_str1="aabcc";  
CString m_str2=m_str1.Left(1)+m_str1.Mid(2,1)+m_str1.Right(1);  
AfxMessageBox(m_str2);
```



图10. 14 提取字符串

### 示例4：查找字符串

代码如下：

```
CString m_str1="abcdef";  
CString m_str2="deq";  
int index=m_str1.Find(m_str2);  
if(index>=0)  
{  
char s[10];  
wsprintf(s,"匹配字符的下标为%d",index);  
MessageBox(s);  
}  
else  
MessageBox("没有匹配字符");
```

运行结果如图10.15所示。



图10. 15 查找字符串

## 示例5：变换字符串

代码如下：

```
CString m_str=" ABCabc ";  
m_str.TrimLeft();  
m_str.TrimRight();  
m_str.MakeUpper();  
MessageBox(m_str);
```



图10. 16 提取字符串

## 第五节 日期和时间类

MFC提供了两个日期和时间类CTime和CTimeSpan,分别代表相对时间和绝对时间。CTime是基于格林威治平均时间（GMT）的，本地的时间由环境变量TZ决定。CTimeSpan代表了时间间隔。

CTime类由下列成员函数：

CTime()

创建一个CTime对象。

GetCurrentTime()

由当前时间创建一个CTime对象。

GetTime()

由CTime对象返回一个time\_t变量。

GetYear()

获取CTime对象代表的年。

GetMonth ( )

获取CTime对象代表的月。

GetDay() 获取CTime对象代表的日期。

GetHour() 获取CTime对象代表的小时。

GetMinute() 获取CTime对象代表的分。

GetSecond() 获取CTime对象代表的秒。

GetDayOfWeek() 获取CTime对象代表的周日，1代表周日，2代表周一等等。

Format() 将字符串转换成一个基于本地时区的格式字符串。

FormatGmt() 将字符串转换成一个基于UTC（世界时）的格式字符串。

operator = 赋予新的时间。

operator + 增加CTime和CTimeSpan对象。

operator - 减小CTime和CTimeSpan对象。

operator += CTime对象加一个CTimeSpan对象。

operator -= CTime对象减一个CTimeSpan对象。

operator == 比较两个绝对时间是否相等。

operator != 比较两个绝对时间是否不相等。

operator < 比较两个绝对时间，是否前一个大于后一个。

operator > 比较两个绝对时间，是否前一个小于后一个。

operator >= 比较两个绝对时间，是否前一个大于等于后一个。

operator <= 比较两个绝对时间，是否前一个小于等于后一个。

下面用几个例子来看看如何使用CTime操作时间。

示例1：获取当前时间

代码如下：

```
CTime m_time=CTime::GetCurrentTime();  
CString s=m_time.Format("%A,%B,%d,%Y");  
CString m_strTime="当前时间是："+s;  
MessageBox(m_strTime);
```

运行结果如图10.17所示。有关Format()函数中的参数在下面的示例中介绍。

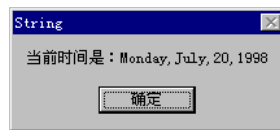


图10. 17 获取当前时间

示例2：由年、月、日得到对应的周日。

代码如下：

```
CTime m_time(1998,7,20,12,0,0);  
int weekday=m_time.GetDayOfWeek();  
switch(weekday)  
{  
case 1:  
    MessageBox("今天是周日");  
    break;  
case 2:  
    MessageBox("今天是周一");  
    break;  
case 3:  
    MessageBox("今天是周二");  
    break;  
case 4:
```

```
MessageBox("今天是周三");  
  
break;  
  
case 5:  
  
MessageBox("今天是周四");  
  
break;  
  
case 6:  
  
MessageBox("今天是周五");  
  
break;  
  
case 7:  
  
MessageBox("今天是周六");  
  
break;  
  
default:  
  
break;  
  
}
```

运行结果如图10.18，代码准确的给出了CTime对象代表周几。



图10. 18 获取CTime对象代表周几

**示例3：将CTime对象转换成格式字符串。**

通过使用Format()函数来实现CTime对象转换成格式字符串。所以在看这个例子之前，先看一下Format()函数的参数，由下列参数：

%a 简写的日期名，例如Sat代表Saturday。

%A 日期名，不简写。

%b 简写的月名，例如Mar代表March。

%B 月名，不简写。



%c 地区化的日期和时间。

%d 月中的天数，值在01到31之间。

%H 24小时格式的小时数，值在00到23之间。

%I 通常的12小时格式的小时数，值在01到12之间。

%j 年中的天数，值在001到366之间。

%m 月数，值在01到12之间。

%M 分钟数，值在00到59之间。

%p 由12小时格式的时钟指示的a.m./p.m.（上午/下午）。

%S 秒数，值在00到59之间。

%U 年中的周数，值在00到51之间，以周日为一周的第一天。

%W 周中的日期数，值在0到6之间，其中0为周日。

%W 年中的周数，值在00到51之间，以周一为一周的第一天。

%x 本地化的日期表示。

%X 本地化的时间表示。

%y 不带年代前缀的年数，值在00到99之间。

%Y 带年代前缀的年数。

%Z 简写的时区名称。

%Z 不简写的时区名称。

%% 百分号标志。

这个程序的代码如下：

```
CTime m_time=CTime::GetCurrentTime();  
CString s1=m_time.Format("%A,%B,%d,%Y");  
CString s2=m_time.Format("%U");
```

```
CString m_str="当前时间是："+s1+"\n"+"本周是今年的第"+s2+"周";  
MessageBox(m_str);
```

运行结果如图10.19所示。

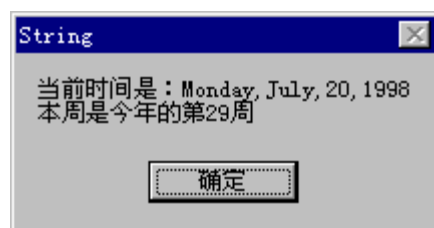


图10. 19 显示当前时间

CTimeSpan类由下列成员函数：

CTimeSpan() 构造一个CTimeSpan类对象。

GetDays() 获得CTimeSpan类对象中包含的完整的天数。

GetHours() 获得当天的小时数，值在-23到23之间。

GetTotalHours() 获得CTimeSpan类对象中包含的完整的小时数。

GetMinutes() 获得当前小时包含的分数，值在-59到59之间。

GetTotalMinutes() 获得CTimeSpan类对象中包含的完整的分数。

GetSeconds() 获得当前分钟包含的秒数，值在-59到59之间。

GetTotalSeconds() 获得CTimeSpan类对象中包含的完整的秒数。

Format() 将一个CTimeSpan对象转换成格式字符串。

operator = 赋予新的时间范围值。

operator + 增加或减小CTimeSpan对象。

operator - 增加或减小CTimeSpan对象。

operator <, <=, >, >=, ==, != 比较两个CTimeSpan对象的大小。

下面的程序将构造一个CTimeSpan对象，并获取其中的完整天数、小时数、分数和秒数，代码如下：

```

CTimeSpan m_timespan(3,4,5,6);

//构造一个CTimeSpan对象

LONG m_totalDays=m_timespan.GetDays();

//获得完整天数

LONG m_totalHours=m_timespan.GetTotalHours();

//获得完整小时数

LONG m_totalMinutes=m_timespan.GetTotalMinutes();

//获得完整分数

LONG m_totalSeconds=m_timespan.GetTotalSeconds();

//获得完整秒数

char s1[8],s2[8],s3[8],s4[8];

wsprintf(s1,"%ld",m_totalDays);

wsprintf(s2,"%ld",m_totalHours);

wsprintf(s3,"%ld",m_totalMinutes);

wsprintf(s4,"%ld",m_totalSeconds);

CString m_str="此时间范围包含：\n完整天数："+CString(s1)+

"\n完整小时数："+CString(s2)+"\n完整分数："+CString(s3)+

"\n完整秒数："+CString(s4);

MessageBox(m_str);

```



图10. 20 获取时间范围量包含的天、小时、分和秒数

这段代码主要让读者体会使用CTimeSpan处理时间范围量的便捷。运行结果如图10.20所示，消息框中显示了希望获得的信息。

# 第十一章 异常处理和诊断

编写程序时出一些错误是难免的，在C++中称在软件或硬件中发生的不期望或不需要的事件为异常（Exception）。MFC提供了两种异常处理机制：

- C++异常，在MFC 3.0和更高版本中可以使用
- MFC异常，在MFC 1.0和更高版本中可以使用

在程序出错误后需要调试程序，MFC提供了许多诊断服务，可以让用户轻松的调试程序，这些诊断服务大多以特定宏和全局函数形式出现。

本章将向读者介绍以下内容：

- 处理C++异常
- MFC异常
- 诊断服务

## 第一节 处理C++异常

C++使用try、catch、throw三个关键字来实现异常处理。使用C++的异常处理能够使你的程序从异常状态中恢复。这些异常由处于正常控制流之外的代码来处理。

- 注意：
- 新的32位的异常处理机制支持C和C++。但是，它并非为C++专门设计的。你应当保证你的代码非常适合于C++的异常处理，并且C++的异常处理是相当灵活的，它可以处理任何类型的异常。

异常处理机制允许程序对严重的和没有预料到的问题做出响应。一个异常块由下列三个部分组成：

- try块

标志你认为可能会出现异常的代码。

- catch块

紧跟着try块，里面包含了处理异常的代码。

- throw块

抛出一个异常，激活catch块中的相应的异常处理代码。

异常处理的机制比较简单，首先你将有可能出现问题的代码放在try块中，然后在catch块中放进用来处理异常的代码。如果在try块中的代码抛出一个异常，try块迅速退出执行，程序将转入catch块中执行相应的异常处理代码。

## 第二节 MFC异常

在MFC中CException类是所有异常类的基类，它是一个抽象类，你不能使用它的对象，只能创建它的派生类的对象。它有两个公用方法：GetErrorMessage（）和ReportError（），分别用于查找描述异常的信息和为用户显示一个错误信息的信息对话框。

CException类包含以下基类（如图11.1所示），并提供了THROW、THROW\_LAST、TRY、CATC、AND\_CATCH、END\_CATCH这些宏用来处理异常。

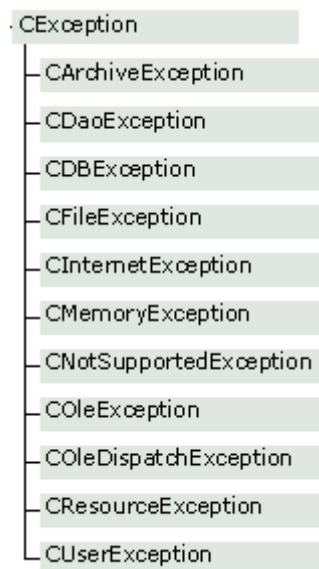


图11. 1 CException的基类

### (1) CArchiveException

一个CArchiveException用来描述序列化异常状态。它包含了一个数据成员m\_cause用来表现异常的原因。它可能为以下值：

none : 没有错误发生

generic : 未指定的错误

readOnly : 试图写入一个为载入而打开的文档

endOfFile : 当读取一个文档时到达文档尾部

writeOnly : 试图读取一个为存储而打开的文档

badIndex : 非法文件格式

badClass : 试图读一个对象到一个错误类型对象中

badSchema : 试图读一个对象, 它带有不同的类的版本

另外MFC还提供AfxThrowArchiveException ( ) 函数来抛出一个存档文件异常, 形式如下:

```
void AfxThrowArchiveException( int cause, LPCTSTR lpszArchiveName );
```

## (2) CDaoException

一个CDaoException类对象用来表示基于数据访问对象 ( DAO, data access objects ) 的MFC数据库类的异常。这个异常类包含三个决定异常原因的成员变量, 这三个成员变量是:

m\_scode: 描述与异常相关的SCODE代码

m\_pErrorInfo : 包含用于所有MFC的DAO类的所有异常的扩展错误代码

m\_nAfxDaoError : 包含DAO错误对象的指向CDaoErrorInfo结构的指针

这个类有下列两个操作:

GetErrorCount ( ) 获取在数据库引擎异常集合中的异常数目

GetErrorInfo ( ) 获取在异常集合中相似对象的错误信息

MFC中的AfxThrowDaoException ( ) 函数抛出一个CDaoException类型的异常, 使用如下形式:

```
void AFXAPI AfxThrowDaoException( int nAfxDaoError = NO_AFX_DAO_ERROR, SCODE  
scode = S_OK );
```

### (3) CDBException

一个CDBException类的对象表示MFC数据库中的一种异常情况。这个类包含两个用来决定异常原因的成员变量：

m\_nRetCode: 一个ODBC(Open Database Connectivity)返回的RETCODE类型的代码

m\_strError: 包含一个描述错误信息的字符串

m\_strStateNativeOrigin: 包含描述带有ODBC错误代码的错误的字符串

MFC中的AfxThrowDBException ( ) 函数抛出一个CDBException类型的异常，使用如下形式：

```
void AfxThrowDBException( RETCODE nRetCode, CDatabase* pdb, HSTMT hstmt );
```

### (3) CFileException

一个CFileException对象描述一个与文件相关的异常状态。这个类包含三个描述异常原因的成员变量：

m\_cause: 包含与错误原因对应的代码。

m\_lOsError: 包含相关的操作系统错误数

m\_strFileName: 包含出现例外的文件名

成员变量m\_cause可能为以下值：

none: 没有错误发生

generic: 未指定的错误

fileNotFound : 文件不能定位错误

badPath : 全部或部分路径无效

tooManyOpenFiles : 达到允许打开的文件数目

accessDenied : 不能访问文件

invalidFile : 试图访问一个无效的文件

removeCurrentDir : 删除正在操作中的目录

directoryFull : 目录个数已满

badSeek : 试图设置文件指针出错

hardIO : 硬件出错

sharingViolation : 共享出错

lockViolation : 试图锁定已锁定的区域

diskFull : 磁盘空间已满

endOfFile : 到达文件结尾

MFC中的AfxThrowFileException ( ) 函数抛出一个CFileException类型的异常，使用如下形式：

```
void AfxThrowFileException( int cause, LONG IOSError = -1, LPCTSTR lpszFileName = NULL );
```

#### (4) CInternetException

一个CInternetException对象代表一个和Internet操作有关的异常状态。它包含两个成员变量：

m\_dwError:表示导致异常的错误

m\_dwContext:和引起错误的操作有关的上下文变量

#### (5) CMemoryException

一个CMemoryException对象描述一个内存溢出异常。内存异常自动的被new操作符抛出。

MFC中的AfxThrowMemoryException ( ) 函数抛出一个CMemoryException类型的异常，使用如下形式：

```
void AfxThrowMemoryException( );
```

#### (6) CNotSupportedException

一个CNotSupportedException对象表示当不支持的特性被请求时发生的异常，没有其它必要或可能的限制。



MFC中的AfxThrowNotSupportedException ( )函数抛出一个CNotSupportedException类型的异常，使用如下形式：

```
void AfxThrowNotSupportedException( );
```

### (7) COleException

一个COleException对象表示和OLE操作有关的异常。它包含一个成员变量m\_sc来容纳异常原因的状态码。

MFC中的AfxThrowOleException ( )函数抛出一个COleException类型的异常，使用如下形式：

```
void AFXAPI AfxThrowOleException( SCODE sc );
```

```
void AFXAPI AfxThrowOleException( HRESULT hr );
```

### (8) COleDispatchException

一个COleDispatchException对象表现为OLE自动化的关键部分所特有的异常。它包含5个成员变量：

m\_wCode: IDispatch特有的错误代码

m\_strDescription: 一个描述性错误

m\_dwHelpContext: 用于错误的Help上下文ID

m\_strHelpFile: 使用m\_dwHelpContext的Help文件

m\_strSource: 产生异常的应用程序

MFC中的AfxThrowOleDispatchException ( )函数抛出一个COleDispatchException类型的异常，使用如下形式：

```
void AFXAPI AfxThrowOleDispatchException( WORD wCode, LPCSTR lpszDescription,  
UINT nHelpID = 0 );
```

```
void AFXAPI AfxThrowOleDispatchException( WORD wCode, UINT nDescriptionID,  
UINT nHelpID = -1 );
```

### (9) CResourceException

一个CResourceException对象表示当Windows不能定位或分配需要的

资源时抛出的异常。

MFC中的AfxThrowOleDispatchException( )函数抛出一个CResourceException类型的异常，使用如下形式：

```
void AfxThrowResourceException( );
```

### (10) CUserException

一个CUserException对象表示停止终端用户操作时抛出的异常。

MFC中的AfxThrowUserException( )函数抛出一个CUserException类型的异常，使用如下形式：

```
void AfxThrowUserException( );
```

## 第三节 诊断服务

上节讲述了使用异常处理来捕获程序中的错误，然而并非程序中所有的错误都是可以捕获的，还会出现很多无法预知的错误，这些错误需要在调试程序中发现并更正。MFC提供了许多诊断服务，供程序员调试程序使用。

MFC提供的用于诊断程序的宏有：ASSERT、ASSERT\_KINDOF、ASSERT\_VALID、DEBUG\_NEW、TRACE、TRACE0、TRACE1、TRACE2、TRACE3、VERIFY。本节将详细介绍ASSERT、VERIFY、TRACE。

### (1) ASSERT

这个宏的用法如下：

```
ASSERT( booleanExpression )
```

其中的参数booleanExpression是一个表达式或指针。

这个宏用来测试它的参数是否为真。如果参数不为真，这个宏就显示一个诊断信息对话框，并终止程序的运行。如果参数为真，它不做任何事情。诊断信息按照下面的形式显示：

```
Debug Assertion Failed!
```

```
Program:<Program Name>
```

```
File:<File Name>
```

Line:<num>

其中Program Name是程序的名称，File Names是出错的文件名，num是出问题的诊断语句所在的行数。

诊断信息对话框如图11.2所示。



图11. 2 诊断输出信息

值得注意的是，ASSERT仅在MFC的调试（Debug）版本中有效，在MFC的发布（Release）版本中，ASSERT语句不再有效，它不对参数进行真假检测。

下面的代码中ASSERT作用是检查一个指向用户自定义的类CMyClass的指针是否为空,代码如下：

```
CMyClass* m_pMyClass=new CMyClass;  
ASSERT(m_pMyClass);  
  
// .....
```

## (2) VERIFY

这个宏和ASSERT差不多，它的用法如下：

```
VERIFY( booleanExpression )
```

其中的参数booleanExpression是一个表达式或指针。

在MFC的调试版本中，VERIFY宏检测它的参数，如果参数不为真，弹出如图1.2所示的诊断信息对话框。如果参数为真，它不做任何事情。

在MFC的发布版本中，它仍对参数进行测试，但是当参数为假时，不弹出诊断信息对话框。

下面这段代码可以让用户对ASSERT和VERIFY两个宏之间的差别有更

深入的了解，这段代码是在MFC的发布版本中编译的，之所以选择发布版本是因为在这段代码中两个宏后面的参数都为假，使用发布版本编译可以忽略这些错误。但是ASSERT不检测参数的真假，而VERIFY检测参数的真假，所以只能弹出一个对话框。

在一个MFC的多文档应用程序Test中的OnDraw（）函数中ToDo语句后面添加下面的代码：

```
bool m_bValue=false;

ASSERT(m_bValue & ASSERTMessage());

VERIFY(m_bValue & VERIFYMessage());
```

为CTestView类添加两个成员函数ASSERTMessage()和VERIFYMessage()：

```
bool CTestView::ASSERTMessage()

{

MessageBox("经过ASSERT检验");

return true;

}

bool CTestView::VERIFYMessage()

{

MessageBox("经过VERIFY检验");

return true;

}
```

在MFC的发布版本中编译并运行该程序，弹出如图11.3所示的消息框，显示“经过VERIFY检验”，说明VERIFY检测了它的参数，而ASSERT没有检测其参数。



图11. 3 VERIFY示例

### (3) TRACE

TRACE宏的用法如下：

TRACE( exp )

其中的参数exp定义了一组数量可变的参数。

TRACE是一个在程序运行时跟踪变量数值的便捷的方法，它的用法和Printf完全相同。

- 注意：
- 使用TRACE一次最多可以显示512个字符，而且这个宏也只在MFC的调试版本中有效。

下面的例子使用TRACE宏在程序运行时跟踪变量m\_value的值。

代码如下：

```
int m_value=100;

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

{

m_value++;

TRACE("m_value = %d",m_value);

}
```

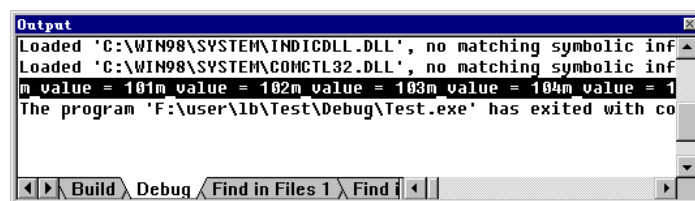


图11. 4 查看TRACE信息

以调试方式运行该应用程序的调试版本，即可从Developer Studio的Output窗口在检查到TRACE宏输出的诊断信息。

## 第十二章 多线程

当使用Windows 95或者其它现在比较流行的操作系统时，可以同时运行几个程序，这是大家都知道的。操作系统的这种能力称之为多任务处理。现今的许多操作系统也支持线程。一个应用程序能够创建几个线程。线程能够使你在多任务中进行多任务。一般的用户知道他能够在同一时刻运行多个程序，而编程者知道一个程序可以在同一时刻运行几个线程。在本章中，你将学会如何在你的程序中创建和管理线程。具体的说，包含以下内容：

- 创建线程
- 线程间通信
- 线程同步

### 第一节 创建线程

线程就是操作系统分配处理器时间的最基本单元。在一个多线程的应用程序中，每一个线程都有它自己的堆栈，并且可以独立的操作在同一程序中运行的其它线程。MFC支持两种线程类型：用户接口线程和工人线程。前者有自己的消息泵，可以处理用户接口的任务，而后者则不能，它是最常用的线程。

一个应用程序至少有一个线程，即程序的基本或主线程。你可以根据需要启动和停止其它附加线程，但是一旦主线程停止了，整个程序就被关闭了。只要程序还在运行，主线程就在运行。

为了使用MFC创建一个线程，你所做的就是编写一个你希望的和程序的其它部分同时运行的函数，然后调用AfxBeginThread（）来启动一个用以执行你的函数的线程。只要线程的函数在运行，线程就存活着，当线程函数结束时，线程就被销毁。

AfxBeginThread（）函数如下所示：

```
CWinThread* AfxBeginThread( AFX_THREADPROC pfnThreadProc, LPVOID pParam,  
int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0,  
DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );  
CWinThread* AfxBeginThread( CRuntimeClass* pThreadClass,
```

```
int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0,  
DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

第一种形式用于创建工人线程，第二种线程用于创建用户接口线程。

这两种形式的函数的返回值是新创建的线程对象的指针。

参数意义如下：

**pfnThreadProc**：指向工人线程的控制函数的指针，它不能是NULL。  
此控制函数必须声明成如下样式：

```
UINT MyControllingFunction( LPVOID pParam );
```

**pThreadClass**：从CWinThread派生的RUNTIME\_CLASS

**pParam**：传递给工人线程的控制函数的参数

**nPriority**：线程的期望的优先权。如果这个值为0，则新线程和创建线程具有同样的优先级。

**nStackSize**：以字节为单位定义了新线程的堆栈大小。如果这个值为0，则新线程和创建线程具有同样大小的堆栈。

**dwCreateFlags**：控制线程创建的附加标志。这个值可以是以下两个值中的一个：CREATE\_SUSPENDED和0。如果是标志是前者，以挂起数为1启动线程。只有在ResumeThread被调用时，这个线程才会被执行。如果标志为0，则在创建线程后立即执行线程。

**lpSecurityAttrs**：指向定义了线程安全属性的SECURITY\_ATTRIBUTES结构的指针。如果为NULL，则新线程和创建线程具有同样的安全属性。

线程可能具有下面的优先级别：

THREAD\_PRIORITY\_ABOVE\_NORMAL 比正常优先级高一个级别

THREAD\_PRIORITY\_BELOW\_NORMAL 比正常优先级低一个级别

THREAD\_PRIORITY\_HIGHEST 比正常优先级高两个级别

THREAD\_PRIORITY\_IDLE 基本优先级为1。对于  
REALTIME\_PRIORITY\_CLASS进程，优先级为16。

THREAD\_PRIORITY\_LOWEST 比正常优先级低两个级别

THREAD\_PRIORITY\_NORMAL 正常优先级别

THREAD\_PRIORITY\_TIME\_CRITICAL 基本优先级为15。对于  
REALTIME\_PRIORITY\_CLASS进程，优先级别是30。

一个线程的优先级决定了相对于其它正在运行的线程这个线程控制系统的时间。通常，线程的级别越高，它的运行时间也越长，这也正是THREAD\_PRIORITY\_TIME\_CRITICAL如此高的原因。

下面用一个简单的例子说明如何创建线程，按照下面的步骤进行：

使用MFC AppWizard生成一个单文档应用程序Thread。

使用资源编辑器编辑器给程序的IDR\_MAINFRAME菜单添加一个菜单“线程”。

在“线程”菜单中添加一个菜单项启动线程，其ID为ID\_STARTTHREAD。

在CThreadView类中添加消息映射函数OnStartthread()。

在OnStartthread()函数中添加如下代码：

```
void CThreadView::OnStartthread()
{
    // TODO: Add your command handler code here

    HWND hWnd = GetSafeHwnd();
    AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);
}
```

添加的代码将调用ThreadProc()，这个函数是新添加的线程的控制函数，所以还需要在程序中添加这个函数。

在ThreadView.cpp中OnStartthread()的上面添加函数ThreadProc()。

- 注意：



- 这个函数是一个全局函数，而并非是CThreadView类的成员函数，尽管它在CThreadView类的执行文件中。

在函数ThreadProc()中添加如下代码：

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);
    return 0;
}
```

这个线程实际上并没有作什么，它仅仅报告它被启动了。

在函数前面的两个冒号表明是在调用全局函数，对于Windows程序员来说，这通常称为API或SDK调用。

当你运行这个程序后，主窗口出现。选择“线程”菜单中的“启动线程”菜单选项，系统启动一个线程，并且显示一个消息框，如图12.1所示。



图12.1 线程启动消息框

## 第二节 线程间通信

通常，一个次要的线程为主线程执行一定的任务，这也暗示这在主线程和次要线程之间需要有一个联系的渠道。有几种方法可以完成这些联系任务：使用全局变量、使用CEvent类或者使用消息。本节将介绍这几种方法。

### (1) 使用全局变量通信

假定你需要你的程序能够停止线程。你需要一个告诉线程何时停止的方法。一种方法是建立一个全局变量，然后让线程监督这个全局变量是否为标志线程终止的值。为了实现这种方法，按照如下步骤修改前面创建的Thread程序。

1. 在“线程”菜单中添加菜单项“停止线程”，ID为

ID\_STOPTHREAD。

2. 为 “ 停止线程 ” 添加消息处理函数OnStopthread()。

3. 在ThreadView.cpp文件中添加一个全局变量threadController。添加方法是在ThreadView.cpp的最上面，在endif下面添加下面的语句：

```
volatile int threadController;
```

关键字volatile表示你希望这个变量可以被外面使用它的线程修改。

4. 修改OnStartthread()函数，代码如下所示：

```
void CThreadView::OnStartthread()
{
    // TODO: Add your command handler code here

    threadController = 1;

    HWND hWnd = GetSafeHwnd();

    AfxBeginThread(ThreadProc, hWnd, THREAD_PRIORITY_NORMAL);
}
```

现在你可能已经猜到threadController的值决定线程是否继续。

5. 在OnStopthread()函数中添加下列代码：

```
threadController = 0;
```

6. 修改ThreadProc()函数的代码，代码如下：

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    while (threadController == 1)
    {
        ;
    }
}
```

```
::MessageBox((HWND)param, "Thread stopped.", "Thread", MB_OK);

return 0;

}
```

现在线程首先显示一个消息框，告诉用户线程被启动了。然后通过一个while循环检查全局变量threadController，等待它的值变成0。尽管这个while循环微不足道，但是你在这里可以加上执行你希望的任务的代码。

现在编译并运行这个程序，选择“线程”菜单中的“启动线程”菜单项启动一个线程，这是弹出如图12.1所示的对话框。然后选择“线程”主菜单中的“停止菜单”菜单项，这时弹出如图12.2所示的对话框，告诉用户线程已经终止。



图12. 2 线程关闭消息框

## (2) 使用用户自定义消息通信

现在你有了一个简单的用于从主线程中联系附加线程的方法。反过来，如何从附加线程联系主线程呢？最简单的实现这种联系的方法是在程序中加入用户定义的Windows消息。

首先，要定义用户消息。这一步很容易，例如：

```
const WM_USERMSG = WM_USER + 100;
```

WM\_USER变量是由Windows定义的，它是第一个有效的用户消息数。因为你的程序的其它部分也会使用用户消息，故将新的用户消息WM\_USERMSG设置为WM\_USER+100。

在定义了用户消息之后，你应当在线程中调用::PostMessage()函数来向主线程发送你所需要的消息。一般按照下面的方式调用::PostMessage()函数：

```
::PostMessage((HWND)param, WM_USERMSG, 0, 0);
```

PostMessage()的四个参数分别是接收消息的窗口的句柄、消息的ID、消息的WPARAM和LPARAM参数。

将下面的语句加入到ThreadView.h中CThreadView类声明的上面。

```
const WM_THREADENDED = WM_USER + 100;
```

仍然是在此头文件中，在消息映射中加入下列语句，注意要加到AFX\_MSG的后面和DECLARE\_MESSAGE\_MAP的前面。

```
afx_msg LONG OnThreadended();
```

然后切回到ThreadView.cpp，在类的消息映射中加入下列语句，位置在}}AFX\_MSG\_MAP之后。

```
ON_MESSAGE(WM_THREADENDED, OnThreadended)
```

再用下面的语句更改ThreadProc()函数。

```
UINT ThreadProc(LPVOID param)
{
    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    while (threadController == 1)
    {
        ;
    }

    ::PostMessage((HWND)param, WM_THREADENDED, 0, 0);

    return 0;
}
```

在CThreadView中添加下面的成员函数。

```
LONG CThreadView::OnThreadended(WPARAM wParam, LPARAM lParam)
{
    AfxMessageBox("Thread ended.");

    return 0;
}
```



图12. 3 线程终止对话框

当你重新运行这个程序时，选择“线程”主菜单中的“启动线程”菜单选项，弹出一个消息框告诉你线程已经启动。为了结束这个线程，选择“线程”主菜单中的“停止菜单”菜单选项，这将弹出一个如图12.3所示的消息框告诉你线程已经停止。

### (3) 使用Event对象通信

一个比较复杂的在两个线程间通信的方法是使用Event对象，在MFC下也就是CEvent类对象。一个Event对象可以有两种状态：通信状态和非通信状态。线程监视着Event对象的状态，并由此在合适的时间执行它们的操作。

创建一个CEvent类的对象很简单，如下：

```
CEvent threadStart;
```

实际上，CEvent的构造函数形式如下：

```
CEvent( BOOL bInitiallyOwn = FALSE, BOOL bManualReset = FALSE,  
LPCTSTR lpszName = NULL, LPSECURITY_ATTRIBUTES lpsaAttribute = NULL );
```

4个参数含义如下：

**bInitiallyOwn** 布尔量。如果值是True，用于CMultiLock和CSingleLock对象的线程将被允许。如果值为False，所有希望访问资源的线程必须等待。缺省值为False。

**bManualReset** 布尔量。如果值为True，则Event对象是手动对象。如果值为False，则Event对象是自动对象。缺省值为True。

**lpszName** CEvent对象的名称。如果事件对象被多个进程使用时必须提供一个名称。缺省值为NULL。

**lpsaAttribute** CEvent对象的安全属性，与在Win32中的SECURITY\_ATTRIBUTES 相同。

尽管CEvent的构造函数有4个参数，但是经常不加任何参数的创建缺

省的对象。当CEvent对象被创建之后，它自动的处在未通信状态。为了使其处在通信状态，可以调用其成员函数SetEvent（），如下所示：

```
threadStart.SetEvent();
```

在执行完上述语句之后，threadStart将处在其通信状态。你的线程应当监视它，这样才能知道何时执行。线程是通过调用如下Windows API函数WaitForSingleObject()来监视CEvent对象的，形式如下：

```
::WaitForSingleObject(threadStart.m_hObject, INFINITE);
```

预定义的常量INFINITE告诉WaitForSingleObject()直到指定的CEvent对象处在通信状态时才返回。换句话说，如果你把WaitForSingleObject()放在线程的开头，系统将挂起线程直到CEvent对象处在通信状态。当主线程准备好后，你应当调用SetEvent()函数。

一旦线程不再挂起，它就可以运行了。但是，如果此时你还想和线程通信，线程必须监视下一个CEvent对象处在通信状态，故你需要再次调用WaitForSingleObject()函数，此时需要将等待时间设置为0，如下所示：

```
::WaitForSingleObject(threadend.m_hObject, 0);
```

在这种情况下，如果WaitForSingleObject()返回值为WAIT\_OBJECT\_0,则CEvent对象处在通信状态。否则，CEvent对象处在非通信状态。

下面的例子说明如何使用CEvent类在两个线程间通信。按照以下步骤进行：

1. 在ThreadView.cpp中#include "ThreadView.h"语句后面加上#include "afxmt.h"。
2. 在ThreadView.cpp中volatile int threadController语句后加上下列语句：

```
CEvent threadStart;
```

```
CEvent threadEnd;
```

删除语句volatile int threadController。

### 3. 用下面的代码更换ThreadProc()函数。

```
UINT ThreadProc(LPVOID param)
{
    ::WaitForSingleObject(threadStart.m_hObject, INFINITE);

    ::MessageBox((HWND)param, "Thread activated.", "Thread", MB_OK);

    BOOL keepRunning = TRUE;

    while (keepRunning)
    {
        int result = ::WaitForSingleObject(threadEnd.m_hObject, 0);

        if (result == WAIT_OBJECT_0)
            keepRunning = FALSE;
    }

    ::PostMessage((HWND)param, WM_THREADENDED, 0, 0);

    return 0;
}
```

### 4. 用下面的语句替换OnStartthread()函数中的内容。

```
threadStart.SetEvent();
```

### 5. 用下面的语句替换OnStopthread()函数中的内容。

```
threadEnd.SetEvent();
```

### 6. 使用ClassWizard为CthreadView处理WM\_CREATE消息的函数OnCreate(), 并在TODO后面添加代码。OnCreate()函数如下所示:

```
int CThreadView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
```

```
HWND hWnd = GetSafeHwnd();  
  
AfxBeginThread(ThreadProc, hWnd);  
  
return 0;  
  
}
```

编译并运行这个程序，新版本的程序运行起来和旧版本的程序一样，但是，新版本的程序为了实现在主线程和次要线程间通信，既使用了CEvent类，又使用了用户定义的Windows消息。

新版本的程序和旧版本的程序的一个大的不同在于次要线程在OnCreate()函数中被启动。然而由于线程函数的第一行即调用WaitForSingleObject()，所以此线程立即被挂起并且等待threadStart处于通信状态。

当threadStart处在通信状态时，新线程显示消息框，然后进入while循环。这个while循环继续执行直到threadEnd处在通信状态，然后线程向主线程发送一个WM\_THREADENDED消息并退出。因为此线程是在OnCreate()函数中被创建的，一旦结束，不会被重新启动。

### 第三节 线程同步

使用多线程可以带来一些非常有趣的问题。例如，如何防止两个线程在同一时间访问同一数据？例如，假设一个线程正在更新一个数据集，而同时另外一个线程正在读取数据集，结果如何？第二个线程将会读取到错误的数据，因为数据集中只有一部分元素被更新过。

保持在同一个进程内的线程工作协调一致称之为线程同步。Event对象实际上就是线程同步的一种形式。在本节中，你将会学到三种使你的多线程程序更安全的线程同步对象——critical section、互斥对象（mutex）、信号量（semaphore）。

#### (1) 使用Critical Section

Critical Section是一种保证在一个时间只有一个线程访问数据集的非常简单的方法。当你使用Critical Section，你给了线程一个它们必须共享的对象。任何拥有Critical Section对象的线程可以访问被保护起来的数据。其它线程必须等待直到第一个线程释放了Critical Section对象，此后其它线程可以按照顺序抢占Critical Section对



象，访问数据。

因为线程只有拥有Critical Section对象才能访问数据，而且在一个时刻只有一个线程可以拥有Critical Section对象，所以决不会出现一个时刻有多个线程访问数据。

为了在MFC程序中创建一个Critical Section对象，你应当创建CCriticalSection类的对象，如下所示：

```
CCriticalSection criticalSection
```

然后，当程序代码准备访问你保护的数据时，你应当调用CCriticalSection的成员函数Lock()，

```
criticalSection.Lock();
```

如果另外一个线程并没有拥有criticalSection，Lock()将criticalSection给调用它的线程。这个线程便能够访问受保护的数据，此后它调用CCriticalSection的成员函数Unlock()：

```
criticalSection.Unlock();
```

Unlock()释放了对criticalSection的拥有权，这样其它线程就可以占有它并访问受保护的数据。

最好的方法是将数据放在线程安全类中。当你这样做后，你不用担心在主线程中的线程同步，线程安全类会替你处理的。下面的类CCountArray便是一个线程安全类。

以下是COUNTARRAY.H，CcountArray的头文件。

```
#include "afxmt.h"
```

```
class CCountArray
```

```
{
```

```
private:
```

```
int array[10];
```

```
CCriticalSection criticalSection;
```

```
public:
```

```
CCountArray() {};
```

```

~CCountArray() {};

void SetArray(int value);

void GetArray(int dstArray[10]);

};

```

在该头文件中包含一个MFC的头文件afxmt.h，以使程序可以使用CCriticalSection类。在CCountArray类的声明中，头文件声明了一个十个元素的整形数组，这是CCriticalSection类的对象将要保护的数据，并且声明了一个CCriticalSection类的对象criticalSection。CCountArray类的公共成员函数包含构造和析构函数。后面两个成员函数用于访问数据。

下面是CCountArray类的执行文件。注意，在每一个成员函数中，CCountArray都在密切关注着CCriticalSection类的对象的状态。这也意味这任何调用这些成员函数的线程不必担心线程同步。例如，如果线程1调用了SetArray(),SetArray()所做的第一件事就是调用criticalSection.Lock(),这将把criticalSection给线程1，此后可以完成一个循环而不用担心被其它线程打断。如果线程2调用了SetArray()或GetArray(),criticalSection.Lock()语句将挂起线程2直到线程1完成循环，执行criticalSection.Unlock()语句将对criticalSection的拥有权释放。这时系统唤醒线程2，并将criticalSection给它。通过这种方式，所有线程必须安静的等待它们访问数据的机会到来。

下面是COUNTARRAY.CPP，CcountArray类的执行文件。

```

#include "stdafx.h"

#include "CountArray.h"

void CCountArray::SetArray(int value)

{

criticalSection.Lock();

for (int x=0; x<10; ++x)

array[x] = value;

criticalSection.Unlock();

}

```

```

void CCountArray::GetArray(int dstArray[10])
{
    criticalSection.Lock();
    for (int x=0; x<10; ++x)
        dstArray[x] = array[x];
    criticalSection.Unlock();
}

```

现在你有机会看到线程安全类是什么样式了，现在可以让这个类工作了。按照以下步骤修改前面的Thread程序来测试CCountArray。

1. 使用File/New菜单命令添加一个新的C++头文件CountArray.h，并将此头文件加入到Thread工程中，并在其中添加代码。
2. 再次使用File/New菜单命令添加一个新的C++资源文件CountArray.cpp，并在其中添加代码。
3. 在ThreadView.cpp文件中，在#include "afxmt.h"下面加上：

```
#include "CountArray.h"
```

4. 在ThreadView.cpp文件中，删除CEvent threadStart；和CEvent threadEnd语句，并加上下列语句：

```
CCountArray countArray;
```

5. 从消息映射中删除下列语句：

```
ON_MESSAGE(WM_THREADENDED, OnThreadended)
```

```
ON_COMMAND(ID_STOPTHREAD, OnStopthread)
```

```
ON_WM_CREATE()
```

6. 用下面两个函数更换ThreadProc()函数。

```
UINT WriteThreadProc(LPVOID param)
```

```

{
    for(int x=0; x<10; ++x)
    {

```

```

countArray.SetArray(x);
::Sleep(1000);
}

return 0;
}

UINT ReadThreadProc(LPVOID param)
{
    int array1[10];
    for (int x=0; x<20; ++x)
    {
        countArray.GetArray(array1);
        char str[50];
        str[0] = 0;
        for (int i=0; i<10; ++i)
        {
            int len = strlen(str);
            wsprintf(&str[len], "%d ", array1[i]);
        }
        ::MessageBox((HWND)param, str, "Read Thread", MB_OK);
    }
    return 0;
}

```

7. 用下面的语句替换OnStartthread()中的所有语句：

```

void CThreadView::OnStartthread()
{
    // TODO: Add your command handler code here
}

```

```

HWND hWnd = GetSafeHwnd();

AfxBeginThread(WriteThreadProc, hWnd);

AfxBeginThread(ReadThreadProc, hWnd);

}

```

8. 删除函数OnStopthread()、OnThreadended()、OnCreate()。

9. 在ThreadView.h删除下列语句：

```
const WM_THREADENDED = WM_USER + 100 ;
```

10. 在ThreadView.h中删除下面的语句：

```
afx_msg void OnStopthread();

afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);

afx_msg LONG OnThreadended();
```

11. 在资源编辑器中删除“线程”主菜单中的“停止线程”菜单选项。

现在编译并运行Thread程序的新版本。当运行这个程序后，主窗口首先出现。选择“线程”主菜单中的“启动线程”菜单选项，首先弹出如图12.4所示的消息框显示当前受保护的数组的值。每次当你单击消息框的“确定”按钮，一个新的消息框又出现。这种消息框要出现20次。在消息框中列出的数组的值取决于单击消息框的“确定”按钮以销毁消息框所花的时间，因为第一个线程每一秒更新一次数组中的数据。



图12. 4 读取数据对话框

如果没有线程同步，则在消息框中出现的数字将不会相同。

如果仔细检查源代码，你会发现第一个线程WriteThreadProc()在一个循环类十次调用CCountArray类的成员函数SetArray()。每一次SetArray()函数将criticalSection给这个线程，修改受保护的数组的值，然后又释放对criticalSection的所有权。注意函数Sleep()，它将这个线程挂起1000毫秒。

第二个线程ReadThreadProc()为了构造一个显示数组元素的字符串也在访问criticalSection。但是如果WriteThreadProc()正在用更新数组中元素的值，ReadThreadProc()必须等待。反之也是对的，即WriteThreadProc()不能够访问受保护的数据直到它重新从ReadThreadProc()得到对criticalSection的所有权。

如果你希望测试一下criticalSection是否在起作用，把SetArray()函数最后的criticalSection.Unlock()删除。重新编译并运行该程序，这次没有消息框出现。因为WriteThreadProc()完全占有了criticalSection，这将导致系统一致将ReadThreadProc()挂起，直到你退出这个程序。

## (2) 使用Mutex (互斥对象)

互斥对象有点象critical section，但有些复杂，因为它不仅允许同一程序的线程之间，而且允许不同程序的线程之间共享资源。尽管在不同程序之间的线程同步超出了本章的范围，但是你可以通过替换critical section获得使用互斥对象的经验。

下面是CCountArray2类的头文件。除了名称和互斥对象外，这个文件和原来的CCountArray完全相同。

```
#include "afxmt.h"

class CCountArray2
{
private:
    int array[10];

    CMutex mutex;

public:
    CCountArray2() {};

    ~CCountArray2() {};

    void SetArray(int value);

    void GetArray(int dstArray[10]);

};
```

下面是CCountArray2的执行文件，正如你所看到的，尽管互斥对象和critical section提供相同的服务，但是二者使用起来还是有很多不同的。

```
#include "stdafx.h"

#include "CountArray2.h"

void CCountArray2::SetArray(int value)
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();
    for (int x=0; x<10; ++x)
        array[x] = value;
}

void CCountArray2::GetArray(int dstArray[10])
{
    CSingleLock singleLock(&mutex);
    singleLock.Lock();
    for (int x=0; x<10; ++x)
        dstArray[x] = array[x];
}
```

为了访问一个互斥对象，你必须创建一个CSingleLock对象或一个CMultiLock对象，由它们来执行实际上的访问控制。CCountArray2类使用CSingleLock对象，因为这个类只处理单一的互斥对象。当代码准备操作受保护的资源，你应当创建一个CSingleLock对象，如下所示：

```
CSingleLock singleLock(&mutex);
```

这个构造函数的参数是一个指向你希望控制的线程同步对象的指针。接着为了获得对互斥对象的访问权限，你应当调用CSingleLock的成员函数Lock()：

```
singleLock.Lock();
```

如果互斥对象不被任何线程拥有，调用上述语句的线程将拥有该互斥对象。如果另外一个程序已经占有了互斥对象，系统将挂起调用上述语句的线程直到互斥对象被释放，此时被挂起的线程被唤醒并且占有互斥对象。

为了释放这个互斥对象，你应该调用CSingleLock对象的成员函数Unlock()。然而，如果你是在栈中创建的CSingleLock对象，就不必调用Unlock()。当函数结束后，该对象超出作用范围，这将使其析构函数执行。析构函数将释放互斥对象。

为了在Thread程序中测试新的类CCountArray2，向工程中添加新的CountArray2.h和CountArray2.cpp，并删除原来的CountArray.h和CountArray.cpp。最后在ThreadView.cpp中将有关CcountArray的代码全部换成CCountArray2，编译并运行程序，结果和使用critical section一样。

### (3) 使用信号量 ( Semaphore )

尽管在MFC程序中使用信号量和使用critical section和互斥对象相差不多，但是功能却不大相同。信号量允许多个线程同时访问资源，但必须是同一点。

当你创建了信号量。你应当告诉它同一时刻有多少线程访问它。这样，每次一个线程抢占资源，信号量减小它内部的计数器。当计数器为0时，不会再有其它线程被允许访问资源直到有释放了资源使计数器增加。

在创建信号量时应当设置计数器的初始值和最大值，如下所示：

```
CSemaphore Semaphore(2, 2);
```

因为在本节中你将使用信号量创建线程安全类，因此需要声明一个CSemaphore的指针作为成员变量，并且在类的构造函数中创建一个CSemaphore对象。

```
semaphore = new CSemaphore(2, 2);
```



一旦你创建了信号量对象，就开始计算资源访问。为了实现资源访问，首先应当创建一个CSingleLock对象，给它信号量的指针，如下：

```
CSingleLock singleLock(semaphore);
```

接着，为了减小信号量的计数器，应当调用CSingleLock的成员函数Lock()，

```
singleLock.Lock();
```

此时，信号量减小了内部的计数器。这个新的数目保持有效直到信号量被释放。通过调用CSingleLock的成员函数Unlock()，

```
singleLock.Unlock();
```

下面是一个新的类CSomeResource。CSomeResource用来说明信号量的用法。这个类只有一个成员变量即指向CSemaphore对象的指针。

下面是CSomeResource类的头文件SOMERESOURCE.H。

```
#include "afxmt.h"

class CSomeResource
{
private:
    CSemaphore* semaphore;

public:
    CSomeResource();
    ~CSomeResource();

    void UseResource();
};
```

下面是CSomeResource类的执行文件SOMERESOURCE.CPP。从中可以看出，CSemaphore对象在构造函数中被创建，在析构函数中被删除。UseResource()成员函数通过从信号量获得一个计数器来访问资源，然后当函数退出时释放信号量。

```
#include "stdafx.h"

#include "SomeResource.h"
```

```

CSomeResource::CSomeResource()
{
semaphore = new CSemaphore(2, 2);
}

CSomeResource::~~CSomeResource()
{
delete semaphore;
}

void CSomeResource::UseResource()
{
CSingleLock singleLock(semaphore);
singleLock.Lock();
Sleep(5000);
}

```

按照下面的步骤修改原来的Thread程序以测试CSomeResource对象。

1. 删除CCountArray类的头文件和执行文件。
2. 创建新的SomeResource.h和SomeResource.cpp。
3. 在新添加的两个文件中加入代码。
4. 用#include "SomeResource.h"语句换掉"CountArray2.h"。
5. 用CSomeResource someResource; 替换CCountArray2 countArray。
6. 用下面三个函数替换WriteThreadProc()和ReadThreadProc()。

```

UINT ThreadProc1(LPVOID param)
{
someResource.UseResource();
}

```

```

::MessageBox((HWND)param, "Thread 1 had access.",
"Thread 1", MB_OK);

return 0;

}

UINT ThreadProc2(LPVOID param)

{

someResource.UseResource();

::MessageBox((HWND)param, "Thread 2 had access.",
"Thread 2", MB_OK);

return 0;

}

UINT ThreadProc3(LPVOID param)

{

someResource.UseResource();

::MessageBox((HWND)param,
"Thread 3 had access.", "Thread 3", MB_OK);

return 0;

}

```

7. 使用下面的语句替换OnStartthread()中所有的代码。

```

void CThreadView::OnStartthread()

{

// TODO: Add your command handler code here

HWND hWnd = GetSafeHwnd();

AfxBeginThread(ThreadProc1, hWnd);

AfxBeginThread(ThreadProc2, hWnd);

AfxBeginThread(ThreadProc3, hWnd);

```

```
}
```

现在编译并运行程序。当主窗口出现后，选择“线程”主菜单中的“启动线程”菜单选项。过5秒钟后，弹出两个消息框告诉你线程1和线程2拥有访问资源的权限。再过5秒钟，第三个消息框弹出告诉你线程3也拥有了访问资源的权限。线程3之所以5秒钟以后才出现，如图12.5所示。因为线程1和线程2首先占有了资源的控制权。信号量被设置成同一时刻只能有两个线程访问资源，所以第三个线程必须等待直到前两个线程释放对信号量的所有权。



图12. 5 线程3拥有访问权限

# 第十三章 动态链接库

在Windows应用程序中使用动态链接库有很多的好处。最主要的一点说是它可以使得多个应用程序共享一段代码，从而可以大幅度的降低应用程序的资源开销，同时很缩小了应用程序的最终执行代码的大小。此外，通过使用动态链接库，我们可以把一些常规的例程独立出来，有效的避免了不必要的重复开发，并且，由于应用程序使用了动态链接的方式，还可以在不需重新改写甚至编译应用程序的基础上更新应用程序的某些组件。

本章介绍Visual C++ 5.0中的动态链接库的创建和使用。这些内容包括

- 为什么要使用DLL
- 不同的DLL类型之间的选择
- 在程序中创建和使用DLL
- 使用DLL扩展MFC

本章的重点在讲述Win32 DLL和基于MFC的常规DLL，对于MFC扩展DLL仅给了一个很简单的例子，更深入的资料请参阅Visual C++的联机文档。

## 第一节 概述

动态链接库（DLL，Dynamic-Link Library）也是一种可执行文件，只不过它不能象普通的EXE文件那样可以直接运行，而是用来为其它可执行文件（包括EXE文件和其它DLL）提供共享函数库。使用DLL的应用程序可以调用DLL中的导出函数（imported function），不过在应用程序本身的执行代码中并不包含这些函数的执行代码，它们经过编译和链接之后，独立的保存在DLL中。这是一种和过去常用的静态链接不同的方式，使用静态链接库（static link library）的应用程序从函数库得到所引用的函数的执行代码，然后把执行代码放进自身的执行文件中，这样，应用程序在运行时就可以不再需要静态函数库的支持。而使用DLL的应用程序只包括了用于从DLL中定位所引用的函数的信息，而没有函数具体实现，要等到程序运行时才从DLL中获得函数的实现代码。显然，使用了DLL的应用程序在运行时必须要有相应的DLL的支持。

DLL在Windows编程中得到了广泛的应用。Windows应用程序的基础：Windows API函数中的相当部分就是由一组DLL所提供的，这些DLL从安装Windows起就存在于系统中了。尽管在前面的几章中我们没有明确的提到，但事实上我们早就在使用DLL进行编程了，只不过，所使用的DLL都是现成的，并且所有调用DLL的操作都由Visual C++的编译和链接程序替我们完成了。

本章将详细的介绍如何创建自己的DLL和如何使用自己创建的DLL进行编程。

为什么要使用DLL呢？这当然是因为与传统的静态链接库相比，使用DLL有着更多的优势：

- 使用DLL提供了一种共享数据和代码的方便途径。并且，由于多个应用程序可以共享同一个DLL中的函数，因此，使用DLL可以显著的节省磁盘空间。尤其对于Windows应用程序，有很多的操作都是“标准化”了的，如果使用传统的静态链接，每一个需要完成这些操作的应用程序都必须在自己的执行文件中包括相同的执行代码，这不但使单个的应用程序变得更长，也浪费了磁盘空间。
- 由于相同的原因，多个应用程序还可以同时共享DLL在内存中的同一份拷贝，这样就有效的节省了应用程序所占用的内存资源，减少了频繁的内存交换，从而提高了应用程序的执行效率。
- 由于DLL是独立于可执行文件的，因此，如果需要向DLL中增加新的函数或是增强现有函数的功能，只要原有函数的参数和返回值等属性不变，那么，所有使用该DLL的原有应用程序都可以在升级后的DLL的支持下运行，而不需要重新编译。这就极大的方便了应用程序的升级和售后支持。
- DLL除了包括函数的执行代码以外，还可以只包括如图标、位图、字符串和对话框之类的资源，因此可以把应用程序所使用的资源独立出来做成DLL。对于一些常用的资源，把它们做到DLL中后，就可为多个应用程序所共享。
- 便于建立多语言的应用程序。我们可以把多语言应用程序中所使用的与语言相关的函数做到DLL中，只要不同语言的应用程序所调用的函数都具有相同的接口，这样就可以通过简单地更换DLL来实现多语言支持。

然而，使用DLL也有其不足之处。最典型的就是应用程序在运行时必

须要有相应的DLL的支持。另外，使用DLL也增大了程序运行的开销，好在这种额外的开销对于大多数应用程序的影响并不明显，我们也只是在某些对运行速度要求苛刻的特殊场合，才不得不考虑这一点。

Visual C++ 5.0支持多种DLL，包括：

- 非MFC DLL
- 静态链接到MFC的常规DLL
- 动态链接到MFC的常规DLL
- MFC扩展DLL

其中非MFC DLL (non-MFC DLL) 内部不使用MFC，调用非MFC DLL提供的导出函数的可执行程序可以使用MFC，也可以不使用MFC。一般来说，非MFC DLL的导出函数都使用标准的C接口 (standard C interface)。

其余三种DLL的内部都使用了MFC。顾名思义，静态链接到MFC的常规DLL (regular DLL statically linking to MFC) 和动态链接到MFC的常规DLL (regular DLL dynamically linking to MFC) 的区别在于一个使用的是MFC的静态链接库，而另一个使用的是MFC的DLL。这和一般的MFC应用程序的情况是很类似的。

MFC扩展DLL一般用来提供派生于MFC的可重用的类，以扩展已有的MFC类库的功能。MFC扩展DLL使用MFC的动态链接版本。只有使用MFC动态链接的可执行程序（无论是EXE还是DLL）才能访问MFC扩展DLL。MFC扩展DLL的另一个有用的功能是它可以在应用程序和它所加载的MFC扩展DLL之间传递MFC和MFC派生对象的指针。在其它情况下，这样做是可能导致问题的。

- 注意：
- 只有Visual C++ 5.0的专业版和企业版才支持到MFC的静态链接。
- 静态链接到MFC的常规DLL过去的USRDLL有着相同的特性，同样的，MFC扩展DLL和过去的AFXDLL有着相同的特性。在Visual C++ 5.0中已不再使用USRDLL和AFXDLL这两个术语。

如何选择所应该使用的DLL的类型呢？我们可以从以下几个方面来考虑：

- 相比使用了MFC的DLL而言，非MFC DLL显得更为短小精悍。因此，如果DLL不需要使用MFC，那么使用非MFC DLL是一个很好的选择，它将显著地节省磁盘和内存空间。同时，无论应用程序是否使用了MFC，都可以调用非MFC DLL中所导出的函数。
- 如果需要创建使用了MFC的DLL，并希望MFC和非MFC应用程序都能使用所创建的DLL，那么可以选择的范围包括静态链接到MFC的常规DLL和动态链接到MFC的常规DLL。动态链接到MFC的常规DLL比较短小，因此可以节省磁盘和内存，但是，在分发动态链接到MFC的常规DLL时，必须同时分发MFC的支持DLL，如MFCx0.DLL和MSVCRT.DLL等。而使用静态链接到MFC的常规DLL则不存在这种问题。
- 如果希望在DLL中实现从MFC派生的可重用的类，或者是希望在应用程序和DLL之间传递MFC的派生对象时，必须选择MFC扩展DLL。

## 第二节 创建和使用动态链接库

本节以非MFC DLL为例来讲解DLL的结构和导出方法，并介绍创建和使用DLL的方法和步骤。

### 13.2.1 DLL的结构和导出方式

DLL文件和EXE文件都属于可执行文件，不同的是DLL文件包括了一个导出表，导出表中给出了可以从DLL中导出的所有函数的名字。外部可执行程序只能访问包括在DLL的导出表中的函数，DLL中的其它函数是私有的，不能为外部可执行程序所访问。

可以使用Visual C++提供的DUMPBIN实用程序（可以在DevStudio\VC\bin目录下找到这个工具）来查看一个DLL文件的结构。举一个例子，如果需要查看DLL文件msgbox.dll（我们将在本小节的后续内容中创建该DLL）的导出表，可以在命令提示符下键入下面的命令：

```
>dumpbin /exports msgbox.dll
```

运行结果如下：

```
Microsoft (R) COFF Binary File Dumper Version 5.00.7022
```

```
Copyright (C) Microsoft Corp 1992-1997. All rights reserved.
```

```
Dump of file msgbox.dll
```



File Type: DLL

Section contains the following Exports for MSGBOX.dll

0 characteristics

351643C3 time date stamp Mon Mar 23 19:13:07 1998

0.00 version

1 ordinal base

1 number of functions

1 number of names

ordinal hint name

1 0 MsgBox (00001000)

Summary

7000 .data

1000 .idata

2000 .rdata

2000 .reloc

17000 .text

由上面的结果得知，msgbox.dll中仅包括了一个导出函数MsgBox()。

- 注意：

- 仅仅知道导出函数的名称并不足以从DLL中导出该函数。若在使用应用程序中使用显式链接（link explicitly），至少还应该知道导出函数的返回值的类型以及所传递给导出函数的参数的个数、顺序和类型；若使用隐含链接（link implicitly），必须有包括导出函数（或类）的定义的头文件（.H文件）和引入库（import library，.LIB文件），这些文件是由DLL的创建者所提供的。关于显式链接和隐含链接，将在本章的“13.2.2 链接应用程序到DLL”小节中讲述。

从DLL中导出函数有两种方法：

- 在创建DLL时使用模块定义（module DEFinition，.DEF）文件。

- 在定义函数时使用关键字\_\_declspec(dllexport)。

下面我们通过一个简单的例子来分别说明两种方法的使用。在这个例子中，我们将创建一个只包括一个函数MsgBox()的DLL，函数MsgBox()用来显示一个消息框，它和Win32 API函数MessageBox()的功能是一样的，只不过在函数MsgBox()中，不需要指定消息的父窗口，而且可以缺省其它所有的参数。

### (1) 使用模块定义文件

模块定义文件是一个文本文件，它包括了一系列的模块语句，这些语句用来描述DLL的各种属性，典型的，模块语句定义了DLL中所导出的函数的名称和顺序值。

在讲解模块定义文件之前，我们先创建一个Win32 Dynamic-Link Library工程。

1. 在Microsoft Developer Studio中选择File菜单下的New命令，在Projects选项卡中选择Win32 Dynamic-Link Library，并为工程取一个名字，如msgbox。单击OK后，Visual C++创建一个Win32 DLL的空白工程，必须手动的将所需要的文件添加到工程中。

2. 单击Project菜单下的Add To Project子菜单下的New命令，在Files选项卡中选择Text File，在File文本框中输入DEF文件名，如msgbox.def。

3. 双击Workspace窗口的FileView选项卡中的msgbox.def节点，在msgbox.def文件中输入下面的内容：

```
LIBRARY MSGBOX
```

```
DESCRIPTION "一个DLL的简单例子"
```

```
EXPORTS
```

```
MsgBox @1
```

在DEF文件中的第一条语句必须是LIBRARY语句，该语句表明该DEF文件属于一个DLL，在LIBRARY之后是DLL的名称，这个名称在链接时将放到DLL的引入库中。

EXPORTS语句下列出了DLL的所有导出函数以及它们的顺序值。函数的顺序值不是必须的，在指定导出函数的顺序值时，我们在函数名后跟

上一个@符号和一个数字，该数字即导出函数的顺序值。如果在DEF中指定了顺序值，它必须不小于1，且不大于DLL中所有导出函数的数目。

DESCRIPTION语句是可选的，它简单的说明了DLL的用途。

4. 下一步是向工程中添加一个头文件，它定义了DLL中的函数的返回值的类型和参数的个数、顺序和类型。

单击菜单项Project|Add To Project|New...，在Files选项卡下选择C/C++ Header File，在File文本框中指定头文件名，如msgbox.h（可以省略后缀名.h）。

在头文件中输入如下的内容：

```
#include <windows.h>

extern "C" int MsgBox(

// 消息框的文本

LPCTSTR lpText="虽然这个例子有一些幼稚，但它工作得非常的好！",

// 消息框的标题

LPCTSTR lpCaption="一个简单的例子",

// 消息框的样式

UINT uType=MB_OK);
```

请注意函数定义前的关键字extern "C"，这是由于我们使用了C++语言来开发DLL，为了使C语言模块能够访问该导出函数，我们应该使用C链接来代替C++链接。否则，C++编译器将使用C++的类型安全命名和调用协议，这在使用C调用该函数时就会遇上问题。在本例中并不需要考虑到这个问题，因为我们在开发DLL和应用程序时都是使用C++，但我们仍然强烈建议使用extern "C"，以保证在使用C编写的程序调用该DLL的导出函数不会遇上麻烦，在本章后面的内容中我们还会讨论到这个问题。

5. 下面要做的事是向工程中添加一个C++源文件，在该文件中实现函数MsgBox()。

仿照上面的过程，单击菜单项Project|Add To Project|New...，在Files选项卡下选择C++ Source File，在File文本框中指定源文件

名，如msgbox.cpp。

在msgbox.cpp文件中添加如下的代码：

```
#include "test1.h"

int MsgBox(LPCTSTR lpText,
LPCTSTR lpCaption,
UINT uType)
{
return MessageBox(NULL, lpText, lpCaption, uType);
}
```

编译该工程，在Debug目录下生成文件msgbox.lib和msgbox.dll。

在“13.2.2 链接应用程序到DLL”小节中将讲述如何使用在本节中所创建的DLL：msgbox.dll。

## (2) 使用关键字\_\_declspec(dllexport)

从DLL中导出文件的另一种方法是在定义函数时使用\_\_declspec(dllexport)关键字。这种方法不需要使用DEF文件。

仍使用前面的例子，在工程中删除msgbox.def文件，将msgbox.h文件修改如下：

```
#include <windows.h>

extern "C" __declspec(dllexport) int MsgBox(

// 消息框的文本

LPCTSTR lpText="虽然这个例子有一些幼稚，但它工作得非常的好！",

// 消息框的标题

LPCTSTR lpCaption="一个简单的例子",

// 消息框的样式

UINT uType=MB_OK);
```

msgbox.cpp文件并不需要做任何修改，重新编译该工程，在Debug目

录下仍生成两个文件msgbox.lib和msgbox.dll。

在下一小节“13.2.2 链接应用程序到DLL”中讲述了如何在应用程序中使用所创建的DLL：msgbox.dll的导出函数MsgBox()。

使用\_\_declspec(dllexport)从DLL中导出类的语法如下：

```
class __declspec(dllexport) CDemoClass
{
...
}
```

- 注意：
- 如果在使用\_\_declspec(dllexport)的同时指定了调用协议关键字，则必须将\_\_declspec(dllexport)关键字放在调用协议关键字的左边。如：
- `int __declspec(dllexport) __cdecl MyFunc();`
- 在32位版本的Visual C++中，\_\_declspec(dllexport)和\_\_declspec(dllimport)代替了16版本中使用的\_\_export关键字。因此，在将16位的DLL源代码移植到Win32平台时，需要把每一处\_\_export替换为\_\_declspec(dllexport)。

如何从这两种导出函数的方法中作出选择，可以从下面的几个方面考虑：

- 如果需要使用导出顺序值（export ordinal value），那么应该使用DEF文件来导出函数。只在使用DEF文件导出函数才能指定导出函数的顺序值。使用顺序值的一个好处是当向DLL中添加新的函数时，只要新的导出函数的顺序值大于原有的导出函数，就没有必要重新链接使用隐含链接的应用程序。相反，如果使用\_\_declspec(dllexport)来导出函数，如果向DLL中添加了新的函数，使用隐含链接的应用程序有可以需要重新编译和链接。
- 使用DEF文件来导出函数，可以创建具有NONAME属性的DLL。具有NONAME属性的DLL在导出表中仅包含了导出函数的顺序值，这种类型的DLL在包括有大量的导出函数时，其文件长度要小于通常的

DLL。

- 使用DEF文件从C++文件导出函数，应该在定义函数时使用extern "C"或者在DEF文件中指定导出函数的decorated name。否则，由于编译器所产生的decorated name是基于特定编译器的，链接到该DLL的应用程序也必须使用创建DLL的同一版本的Visual C++来编译和链接。
- 由于使用\_\_declspec(dllexport)关键字导出函数不需要编写DEF文件，因此，如果编写的DLL只供自己使用，使用\_\_declspec(dllexport)较为简单。
- 注意：
  - MFC本身使用了DEF文件从MFCx0.DLL中导出函数和类。

### 13.2.2 链接应用程序到DLL

同样，链接应用程序到DLL也有两种方法：

- 隐含链接
- 显式链接

隐含链接有时又称为静态加载。如果应用程序使用了隐含链接，操作系统在加载应用程序的同时加载应用程序所使用的DLL。显式链接有时又称为动态加载。使用动态加载的应用程序必须在代码中明确的加载所使用的DLL，并使用指针来调用DLL中的导出函数，在使用完毕之后，应用程序必须卸载所使用的DLL。

同一个DLL可以被应用程序隐含链接，也可以被显式链接，这取决于应用程序的目的和实现。

下面我们在分别讲述两种不同的链接方式之后再作对比。

#### (1) 使用隐含链接

在使用隐含链接除了需要相应的DLL文件外，还必须具备如下的条件：

- 一个包括导出的函数或C++类的头文件

- 一个输入库文件（.LIB文件）

通常情况下，我们需要从DLL的提供者那里得到上面的文件。输入库文件是在DLL文件被链接时由链接程序生成的。

在“13.2.1 DLL的结构和导出方式”中所创建的DLL：msgbox.dll所对应的头文件msgbox.h如下：

```
#include <windows.h>

extern "C" __declspec(dllimport) int MsgBox(

// 消息框的文本

LPCTSTR lpText="虽然这个例子有一些幼稚，但它工作得非常的好！",

// 消息框的标题

LPCTSTR lpCaption="一个简单的例子",

// 消息框的样式

UINT uType=MB_OK);
```

需要注意的是，这个msgbox.h文件和创建DLL时所使用msgbox.h是不同的，唯一的差别在于，创建DLL时的msgbox.h中使用的是\_\_declspec(dllexport)关键字，而供应用程序所使用的msgbox.h中使用的是\_\_declspec(dllimport)关键字。无论创建DLL时使用的是DEF文件还是\_\_declspec(dllexport)关键字，均可使用\_\_declspec(dllimport)关键字从DLL中引入函数。引入函数时也可以省略\_\_declspec(dllimport)关键字，但是使用它可以使编译器生成效率更高的代码。

- 注意：
  - 如果需要引入的是DLL中的公用数据和对象，则必须使用\_\_declspec(dllimport)关键字。

现在使用Microsoft Developer Studio创建一个Win32 Application工程，命名为tester。向工程中添加一个C++源文件，如tester.cpp。在tester.cpp文件中输入下面的代码：

```
#include "msgbox.h" // 应将msgbox.h文件拷贝到工程tester的目录下。

int WINAPI WinMain(HINSTANCE hInstance,
```

```

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nCmdShow)

{

return MsgBox();

}

```

在上面的代码中，MsgBox() 函数的所有参数都使用了缺省值。

• 注意：

- 在编译之前，将上一步生成的msgbox.lib文件拷贝到tester工程所在的目录下。然后单击菜单项Project|Settings...，将msgbox.lib添加到Link选项卡下的Object/Library modules文本框中。如果忽略这一步，链接时将会导致错误。完成之后创建该应用程序。

如果现在运行该程序，将出现如图13.1所示的对话框。

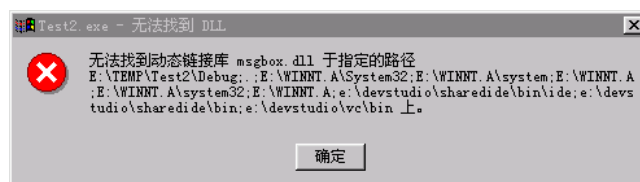


图13.1 未找到DLL时出现的错误

上面的对话框说明程序没有在指定的路径未找到所需要的DLL。

一般情况下，程序在运行时，系统将按如下的顺序查找程序所使用的动态链接库：

- 系统预安装的DLL，如KERNEL32.DLL和USER32.DLL等
- 当前目录
- Windows的系统的目录，如WINNT\system32
- Windows所在的目录，如WINNT
- 环境变量PATH中所指定的目录



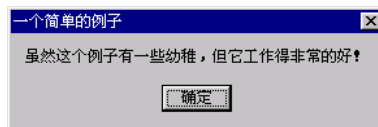


图13.2 tester应用程序的运行结果

如果Windows在上面的目录中未找到所需要的DLL，则弹出如图13.1所示的对话框。这里，我们把msgbox.dll文件拷贝到tester\Debug目录下，再运行应用程序，则出现如图13.2所示的对话框。

## (2) 使用显式链接

如果没有与DLL相关联的LIB文件，则必须使用显式链接。使用显式链接同样必须知道函数返回值的类型和所传递的参数个数、类型和顺序。与使用隐含链接不同的是，使用显式链接的应用程序在调用DLL中的导出函数前，必须使用LoadLibrary()函数加载DLL并得到一个模块句柄。然后使用该句柄调用GetProcAddress()函数获得所需调用的导出函数的指针，并通过该指针调用DLL中的导出函数，这种模式使用显式链接到DLL的应用程序不再需要相应的LIB文件。在使用完毕之后，还需调用FreeLibrary()函数释放加载的DLL。

下面我们使用显式链接的方式来实现前面的例子。

由于使用指针来调用DLL中的导出函数，所以本例中不再需要msgbox.h文件。

在tester.cpp中添加的代码如下所示：

```
#include <windows.h>

typedef int (CALLBACK* DLLFUNC)(

LPCTSTR lpText="虽然这个例子有一些简单，但它工作得非常的好！",

LPCTSTR lpCaption="一个简单的例子",

UINT=MB_OK);

int WINAPI WinMain(HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nCmdShow)

{
```

```

HINSTANCE hDLL;

DLLFUNC MsgBox;

hDLL = LoadLibrary("msgbox");

if (hDLL != NULL)

{

MsgBox =

(DLLFUNC)GetProcAddress(hDLL, "MsgBox");

return MsgBox();

}

}

```

LoadLibrary()函数的参数是所调用的DLL的名字，这个名字不是放入输入库文件中的名字，而是DLL的文件名。如果文件的扩展名为.DLL，则可以省略。

这个程序的运行结果同使用隐含链接的前一个程序一样，但它的内部实现是很不相同的。使用显式链接的应用程序加载时，所调用的DLL并不加载，只有当应用程序调用LoadLibrary()时系统才加载相应的DLL，并在应用程序调用FreeLibrary()时卸载该DLL。使用隐含链接的应用程序调用DLL中的导出函数时，方法同调用一般的函数一样，而使用显式链接的应用程序必须使用指针来调用。由于使用了指针，因此在编译时不能验证参数的合法性，通过指针使用不合法的参数来调用DLL中的导出函数将会导致不可预料的后果。

很明显，使用隐含链接的方式调用DLL中的导出函数要比使用显式链接方便得多。但在某些情况下我们必须使用显式链接。事实上，使用显式链接调用DLL提供了更大的灵活性。尤其在没有任何与DLL相对应的LIB文件时，我们只能使用显式链接来调用DLL中的导出函数，并且，只要我们使用函数名作参数来调用GetProcAddress()，在更新DLL时，就没有必要重新链接应用程序。另外，使用隐含链接的方式的应用程序加载DLL时如果发生错误（如DLL文件未找到或是DLL中的DllMain()函数初始化失败）时，应用程序将被终止，而使用显式链接的应用程序则可以使用如上面的例子中所给出的方法来避免出现这种情况（可以使用所创建的两个不同版本的tester程序来验证这一点）。

由于应用程序调用LoadLibrary()函数时才加载DLL，因此使用显式链接的应用程序的加载速度要比使用隐含链接的应用程序快。使用显式链接的另一个好处是，应用程序可以在运行时决定所加载的DLL。

但是要记住，由于使用了指针来传递应用程序的参数，因此编译器在编译时无法确认应用程序所传递的参数类型是否合法。传递不合法的参数给DLL中的导出函数是一件危险的事。在程序调试的过程中我们一定需要注意这一点。

### 第三节 使用动态链接库扩展MFC

我们还可以使用DLL来实现从MFC派生的一些可重用类，这种动态链接库一般称作MFC扩展动态链接库(MFC Extension DLL)。正如这个名称所暗示的那样，通过这种方式我们可以扩展MFC所包括的内容，使得使用MFC编程更加的方便。此外，如果需要在应用程序和DLL之间传递MFC或者由MFC派生的对象的指针的话，我们也必须使用MFC扩展DLL。

在本节中，我们使用MFC扩展DLL来创建一个输入通用对话框，如图13.3所示。该对话框很象Visual Basic中的InputBox函数所产生的对话框，使用过Visual Basic的程序员都有印象，函数InputBox非常之好用，这里，我们来使用动态链接库在Visual C++的MFC中也创建这么一个好用的类。



图13. 3 输入通用对话框

输入通用对话框由类CInputDialog封装，类CInputDialog提供了一个公有成员函数GetInput，该成员函数的原型如下：

```
CString GetInput(CString Title, CString Prompt)
```

第一个参数Title表示输入对话框的标题，在图13.3中为“输入”；第二个参数Prompt代表在输入对话框中显示的简短提示文本，在图13.3中为“请输入对话框的标题：”。函数的返回值为用户在对话框的文本框中输入的字符串。如果用户没有输入任何字符串或者单击了“取消”按钮，返回值为空串“”。

下面我们来介绍该对话框的创建和使用。首先讲述DLL工程ExtDllDemo的创建。该工程实现了类CInputDialog的导出。

1. 使用AppWizard创建一个MFC扩展DLL工程，将工程取名为ExtDllDemo。

2. 向工程中添加一个对话框资源IDD\_INPUT，按图13.3绘制对话框中的各控件。这些控件的资源ID如表所示。

表13.1 对话框资源IDD\_INPUT中的控件属性

控件	资源ID
提示文本 标签	IDC_PROMPT
输入文本 框	IDC_EDIT

3. 使用ClassWizard为对话框资源IDD\_INPUT创建新的对话框类CInputDlg，该类直接派生于CDialog。按下面的代码修改类CInputDlg的头文件和实现文件。

```
#if !defined(AFX_INPUTDLG_H__02DB98CF_1F76_11D2_971A_0000B4810A31__INCLUDED_)
#define AFX_INPUTDLG_H__02DB98CF_1F76_11D2_971A_0000B4810A31__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

// InputDlg.h : 头文件
//

// 以下对话框 ID 常量需要手动添加

#define IDC_EDIT 1000
#define IDC_PROMPT 1001
#define IDD_INPUT 129

////////////////////////////////////

// CInputDlg 对话框

class __declspec(dllexport) CInputDlg : public CDialog
{
```

```

// 构造

public:

CString GetInput(CString Title, CString Prompt);

CInputDialog(CWnd* pParent = NULL); // 标准构造函数

// 对话框数据

//{{AFX_DATA(CInputDialog)

enum { IDD = IDD_INPUT };

CString m_strTitle;

CString m_strPrompt;

CString m_strInput;

//}}AFX_DATA


// 重载

// 由 ClassWizard 生成的虚函数重载

//{{AFX_VIRTUAL(CInputDialog)

protected:

virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV 支持

//}}AFX_VIRTUAL


// 实现

protected:

// 生成的消息映射函数

//{{AFX_MSG(CInputDialog)

virtual BOOL OnInitDialog();

//}}AFX_MSG

DECLARE_MESSAGE_MAP()

};

```

```

//{{AFX_INSERT_LOCATION}}

// Microsoft Developer Studio 将在紧贴上一行之前的位置添加附加的声明

#endif // !defined
(AFX_INPUTDLG_H__02DB98CF_1F76_11D2_971A_0000B4810A31__INCLUDED_)

// InputDlg.cpp : 实现文件

//

#include "stdafx.h"

#include "InputDlg.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif

////////////////////////////////////

// CInputDialog dialog

CInputDialog::CInputDialog(CWnd* pParent /*=NULL*/)
: CDialog(CInputDialog::IDD, pParent)
{
//{{AFX_DATA_INIT(CInputDialog)
m_strInput = _T("");
//}}AFX_DATA_INIT
}

void CInputDialog::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);

```

```

//{{AFX_DATA_MAP(CInputDialog)
DDX_Text(pDX, IDC_EDIT, m_strInput);

//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CInputDialog, CDialog)

//{{AFX_MSG_MAP(CInputDialog)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////

// CInputDialog message handlers

CString CInputDialog::GetInput(CString Title, CString Prompt)
{
    CString str(""); // 注意：这里对 CString 对象 str 的初始化是必要，否则
    // 在后面的过程将会出错

    // 对标题条和提示文本的实际的更新将在消息处理函数 OnInitDialog 中进行
    m_strTitle=Title;
    m_strPrompt=Prompt;
    if (DoModal()==IDOK)
    {
        // 如果用户单击了确定，则返回所输入的字符串
        str=m_strInput;
    }

    return str;
}

BOOL CInputDialog::OnInitDialog()

```

```

{
CDialog::OnInitDialog();

// 使用用户指定的标题字符串

SetWindowText(m_strTitle);

// 设置提示文本

GetDlgItem(IDC_PROMPT)->SetWindowText(m_strPrompt);

// 将输入焦点设置为 IDC_EDIT 控件

GetDlgItem(IDC_EDIT)->SetFocus();

// 由于将输入焦点设置为 IDC_EDIT 控件，因此 OnInitDialog 成员函数应该返回假值

return FALSE;

}

```

在示例程序中，我们使用了\_\_declspec(dllexport)来导出类CInputDlg，最主要的原因是因为这种方法相对比较简单一些。

编译DLL工程ExtDIIDemo，在Debug目录下生成了动态链接库ExtDIIDemo.dll的导入库ExtDIIDemo.lib。

下面我们创建动态链接库ExtDIIDemo.dll的测试工程ExtDIITest。

1. 使用AppWizard创建基于对话框的EXE工程ExtDIITest，工程ExtDIITest使用了动态MFC链接(这是必须的)。将在前一步中生成的ExtDIIDemo.lib拷贝到工程ExtDIITest所在的目录下。完成这一步之后，从Project菜单下选择Settings命令，在工程的设置对话框中选择Link选项卡，在Object/Library Modules框中输入ExtDIIDemo.lib。

2. 在实现文件ExtDIIDemoDlg.cpp的最前面输入下面的代码：

```

////////////////////////////////////

// CInputDlg dialog

class __declspec(dllimport) CInputDlg : public CDialog
{
public:

```



```
CString GetInput(CString Title, CString Prompt);  
  
CInputDialog(CWnd* pParent = NULL);  
  
};
```

上面的代码在工程中定义了类CInputDialog，以便于在以后的代码中使用该类。这里需要注意的是，我们只需给出对我们有意义的那些成员的声明即可，而没有必须在上面的定义中给出完全的成员声明。

在OnInitDialog成员函数的// TODO注释下输入下面的代码。该代码在应用程序的主对话框弹出之前询问主对话框的标题文本。

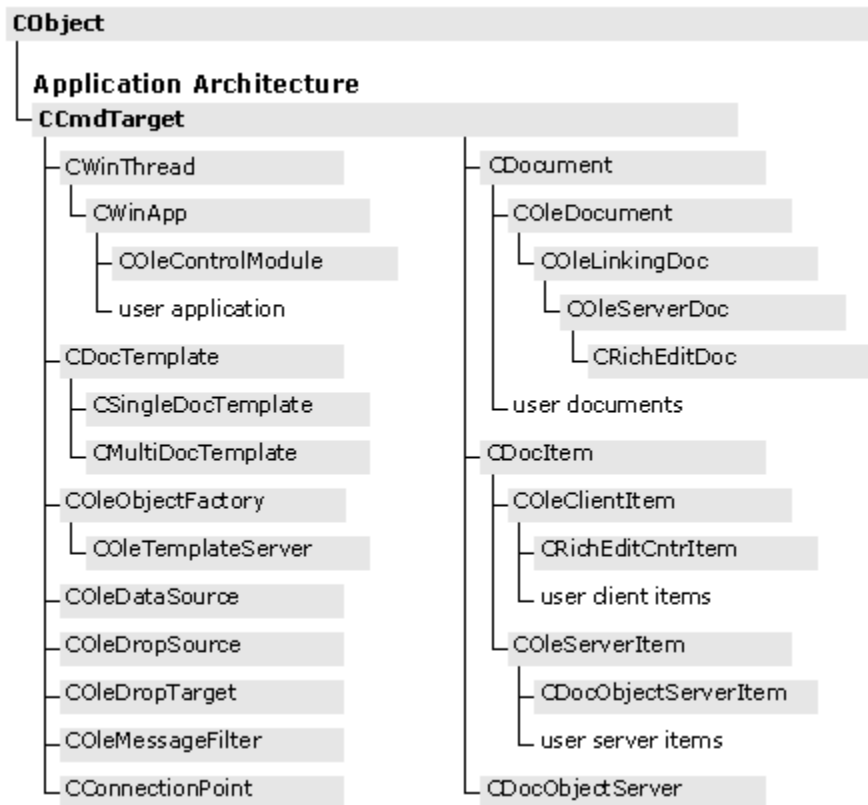
```
CInputDialog dlg;  
  
CString str=dlg.GetInput("输入", "请输入对话框的标题：");  
  
SetWindowText(str);
```

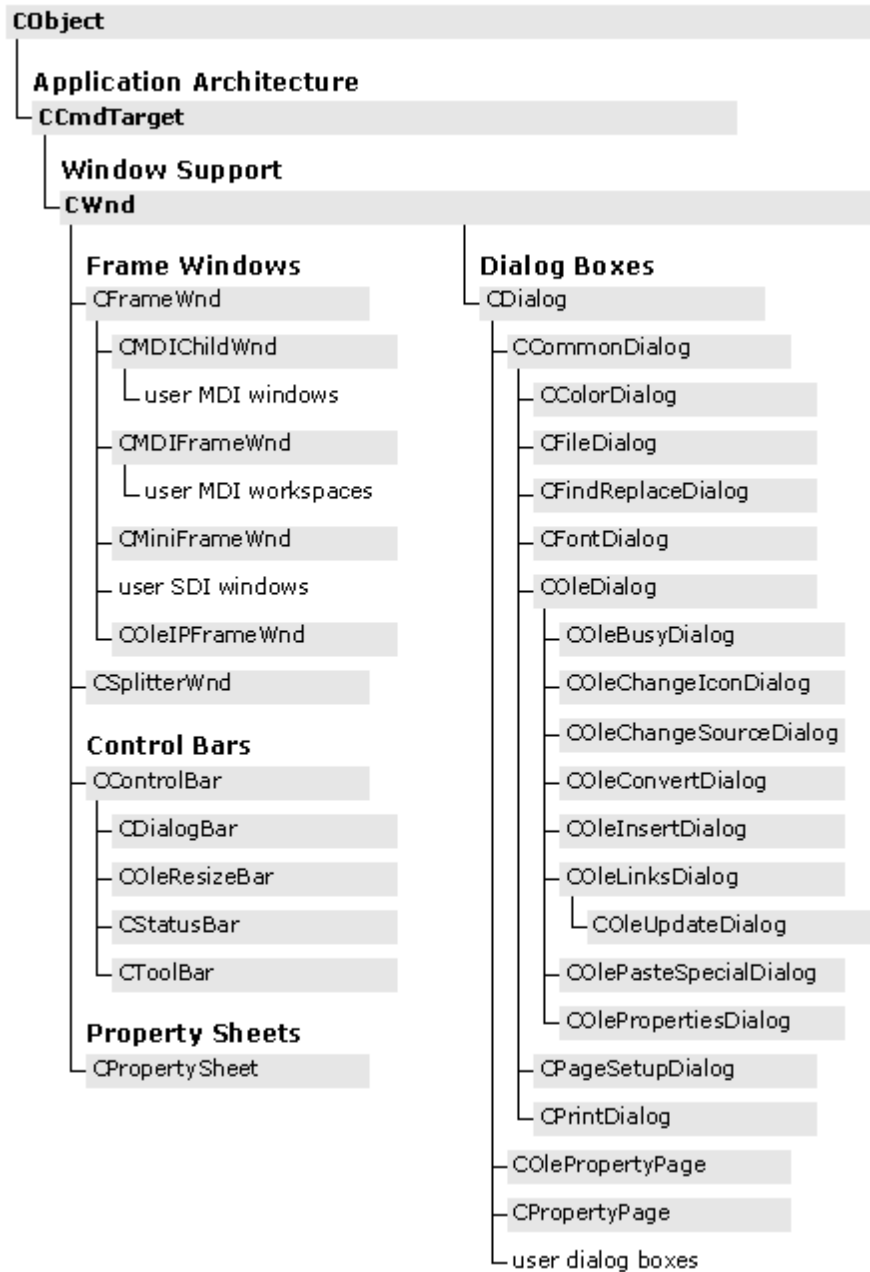
- 注意：
  - 如果在CInputDialog::GetInput成员函数中没有对对象变量str进行正确的初始化(如直接使用return m\_strInput等)，那么上面的代码的Debug版本在运行时将会出现Assertion失败。这种现象的根源在于CString内部所使用的内存分配方式。由于CString所使用的内存分配方式相当的复杂，因此，我们不在这里深入的讨论这一现象，仅仅指出存在这种问题而已。

编译并生成应用程序ExtDllTest，然后将动态链接库ExtDllDemo.dll拷贝到应用程序ExtDllTest的目录或系统目录中，再运行ExtDllTest，已检验动态链接库ExtDllTest的工作是否正确。

关于DLL还有很多课题可以研究，但是由于篇幅所限，我们在这里仅给出一些最基本的概念和方法，更详细的参考资料可以查阅MFC的联机文档。

# 附表1 MFC类库层次表





## CObject

user objects

### Exceptions

CException

CArchiveException

CDaoException

CDBException

CFileException

CInternetException

CMemoryException

CNotSupportedException

COleException

COleDispatchException

CResourceException

CUserException

### File Services

CFile

CMemFile

CSharedFile

COleStreamFile

CMonikerFile

CAsyncMonikerFile

CDataPathProperty

CCachedDataPathProperty

CSocketFile

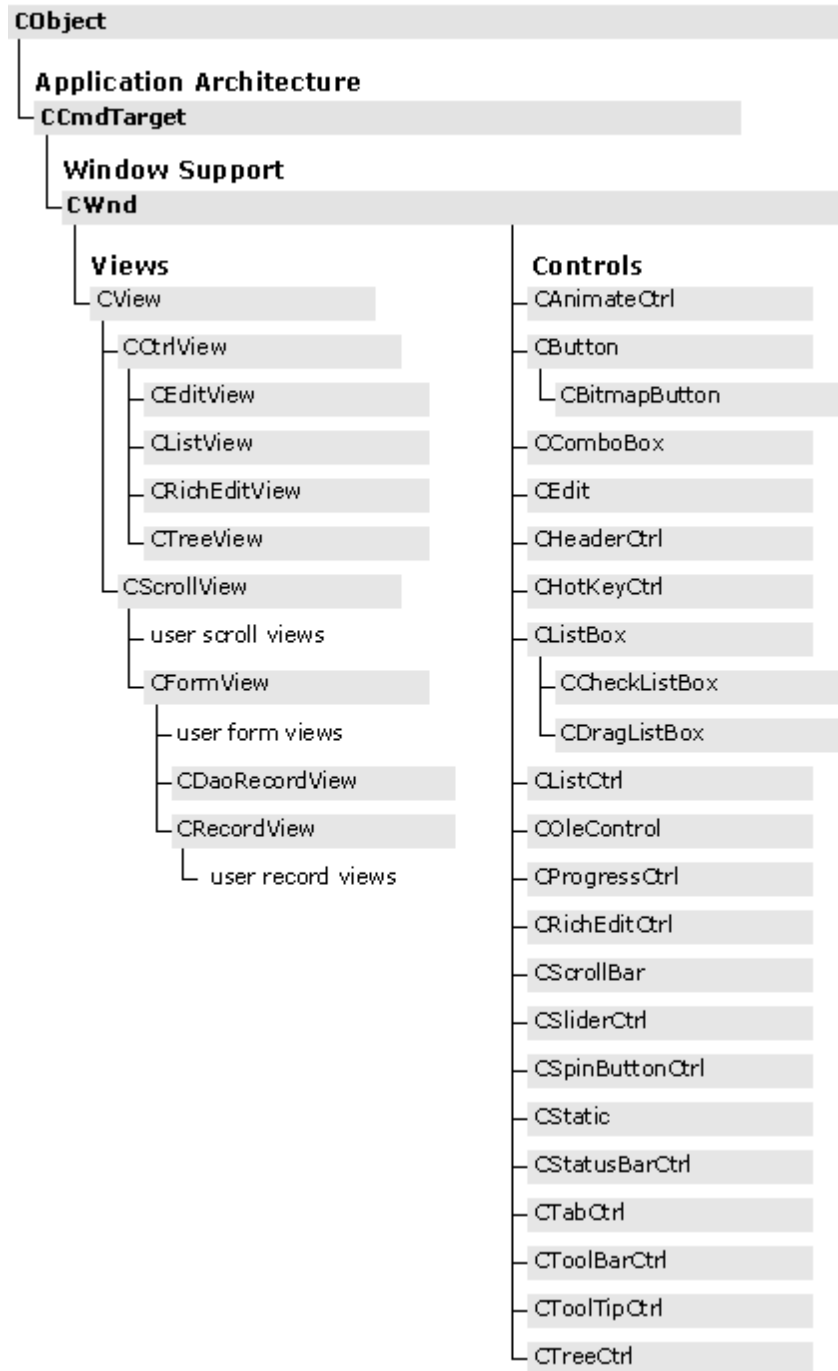
CStdioFile

CInternetFile

CGopherFile

CHttpFile

CRecentFileList



## CObject

### Graphical Drawing

- CDC
  - CClientDC
  - CMetaFileDC
  - CPaintDC
  - CWindowDC

### Control Support

- CDockState
- CImageList

### Graphical Drawing Objects

- CGdiObject
  - CBitmap
  - CBrush
  - CFont
  - CPalette
  - CPen
  - CRgn

### Menus

- CMenu

### ODBC Database Support

- CDatabase
  - CRecordset
    - user recordsets
  - CLongBinary

### DAO Database Support

- CDaoDatabase
  - CDaoQueryDef
  - CDaoRecordset
  - CDaoTableDef
  - CDaoWorkspace

### Synchronization

- CSyncObject
  - CCriticalSection
  - CEvent
  - CMutex
  - CSemaphore

### Windows Sockets

- CAsyncSocket
  - CSocket

### Arrays

- CArray (template)
- CByteArray
- CWordArray
- CObArray
- CPtrArray
- CStringArray
- CJIntArray
- CWordArray
- arrays of user types

### Lists

- CList (template)
- CPtrList
- CObList
- CStringList
- lists of user types

### Maps

- CMap (template)
- CMapWordToPtr
- CMapPtrToWord
- CMapPtrToPtr
- CMapWordToOb
- CMapStringToPtr
- CMapStringToOb
- CMapStringToString
- maps of user types

### Internet Services

- CInternetSession
- CInternetConnection
  - CFTPConnection
  - CGopherConnection
  - CHttpConnection
- CFileFind
  - CFTPFileFind
  - CGopherFileFind
- CGopherLocator

## Classes Not Derived from CObject

### Internet Server API

CHtmlStream  
CHttpFilter  
CHttpFilterContext  
CHttpServer  
CHttpServerContext

### Run-time Object Model Support

CArchive  
CDumpContext  
CRuntimeClass

### Simple Value Types

CPoint  
CRect  
CSize  
CString  
CTime  
CTimeSpan

### Structures

CCommandLineInfo  
CCreateContext  
CMemoryState  
COleSafeArray  
CPrintInfo

### Support Classes

CComdUI  
COleCmdUI  
CDaoFieldExchange  
CDataExchange  
CDBVariant  
CFieldExchange  
COleDataObject  
COleDispatchDriver  
CPropExchange  
CRectTracker  
CWaitCursor

### Typed Template Collections

CTypedPtrArray  
CTypedPtrList  
CTypedPtrMap

### OLE Type Wrappers

CFontHolder  
CPictureHolder

### OLE Automation Types

COleCurrency  
COleDateTime  
COleDateTimeSpan  
COleVariant

### Synchronization

CMultiLock  
CSingleLock

附表2 ASCII码表 (0~127)

Ctl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`
^A	1	01	☐	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	☐	SIX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	EIX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOI	36	24	\$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	☐	BS	40	28	(	72	48	H	104	68	h
^I	9	09	○	HI	41	29	)	73	49	I	105	69	i
^J	10	0A	☐	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VI	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	⌋	CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E	⌋	SD	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	✱	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	▶	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◀	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	!!	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	☐	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	☐	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	≡	EIB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SIB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C	⌞	FS	60	3C	<	92	5C	\	124	7C	
^]	29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	Δ†

## 附录3 虚拟键码

符号常量	十六进制值	指定的鼠标或键盘按键
VK_LBUTTON	01	鼠标左键
VK_RBUTTON	02	鼠标右键
VK_CANCEL	03	Control-break 过程
VK_MBUTTON	04	鼠标中键
?	05?07	未定义
VK_BACK	08	BACKSPACE 键
VK_TAB	09	TAB 键
?	0A?0B	未定义
VK_CLEAR	0C	CLEAR 键
VK_RETURN	0D	ENTER 键
?	0E?0F	未定义
VK_SHIFT	10	SHIFT 键



VK_CONTROL	11	CTRL 键
VK_MENU	12	ALT 键
VK_PAUSE	13	PAUSE 键
VK_CAPITAL	14	CAPS LOCK 键
?	15?19	为 Kanji 系统保留
?	1A	未定义
VK_ESCAPE	1B	ESC 键
?	1C?1F	为 Kanji 系统保留
VK_SPACE	20	SPACEBAR
VK_PRIOR	21	PAGE UP 键
VK_NEXT	22	PAGE DOWN 键
VK_END	23	END 键
VK_HOME	24	HOME 键
VK_LEFT	25	LEFT ARROW 键
VK_UP	26	UP ARROW 键
VK_RIGHT	27	RIGHT ARROW 键
VK_DOWN	28	DOWN ARROW 键
VK_SELECT	29	SELECT 键
?	2A	由OEM厂商指定
VK_EXECUTE	2B	EXECUTE 键
VK_SNAPSHOT	2C	PRINT SCREEN键（用于Windows 3.0及以后版本）
VK_INSERT	2D	INS 键
VK_DELETE	2E	DEL 键
VK_HELP	2F	HELP 键
VK_0	30	0 键
VK_1	31	1 键
VK_2	32	2 键
VK_3	33	3 键
VK_4	34	4 键
VK_5	35	5 键
VK_6	36	6 键
VK_7	37	7 键
VK_8	38	8 键
VK_9	39	9 键
?	3A?40	未定义

VK_A	41	A 键
VK_B	42	B 键
VK_C	43	C 键
VK_D	44	D 键
VK_E	45	E 键
VK_F	46	F 键
VK_G	47	G 键
VK_H	48	H 键
VK_I	49	I 键
VK_J	4A	J 键
VK_K	4B	K 键
VK_L	4C	L 键
VK_M	4D	M 键
VK_N	4E	N 键
VK_O	4F	O 键
VK_P	50	P 键
VK_Q	51	Q 键
VK_R	52	R 键
VK_S	53	S 键
VK_T	54	T 键
VK_U	55	U 键
VK_V	56	V 键
VK_W	57	W 键
VK_X	58	X 键
VK_Y	59	Y 键
VK_Z	5A	Z 键
VK_LWIN	5B	Left Windows 键 (Microsoft自然键盘)
VK_RWIN	5C	Right Windows 键 (Microsoft自然键盘)
VK_APPS	5D	Applications 键 (Microsoft自然键盘)
?	5E?5F	未定义
VK_NUMPAD0	60	数字小键盘上的 0 键
VK_NUMPAD1	61	数字小键盘上的 1 键
VK_NUMPAD2	62	数字小键盘上的 2 键
VK_NUMPAD3	63	数字小键盘上的 3 键

VK_NUMPAD4	64	数字小键盘上的 4 键
VK_NUMPAD5	65	数字小键盘上的 5 键
VK_NUMPAD6	66	数字小键盘上的 6 键
VK_NUMPAD7	67	数字小键盘上的 7 键
VK_NUMPAD8	68	数字小键盘上的 8 键
VK_NUMPAD9	69	数字小键盘上的 9 键
VK_MULTIPLY	6A	Multiply 键
VK_ADD	6B	Add 键
VK_SEPARATOR	6C	Separator 键
VK_SUBTRACT	6D	Subtract 键
VK_DECIMAL	6E	Decimal 键
VK_DIVIDE	6F	Divide 键
VK_F1	70	F1 键
VK_F2	71	F2 键
VK_F3	72	F3 键
VK_F4	73	F4 键
VK_F5	74	F5 键
VK_F6	75	F6 键
VK_F7	76	F7 键
VK_F8	77	F8 键
VK_F9	78	F9 键
VK_F10	79	F10 键
VK_F11	7A	F11 键
VK_F12	7B	F12 键
VK_F13	7C	F13 键
VK_F14	7D	F14 键
VK_F15	7E	F15 键
VK_F16	7F	F16 键
VK_F17	80H	F17 键
VK_F18	81H	F18 键
VK_F19	82H	F19 键
VK_F20	83H	F20 键
VK_F21	84H	F21 键
VK_F22	85H	F22 键
VK_F23	86H	F23 键
VK_F24	87H	F24 键
?	88?8F	未指定

VK_NUMLOCK	90	NUM LOCK 键
VK_SCROLL	91	SCROLL LOCK 键
?	92?B9	未指定
?	BA?C0	由 OEM 厂商指定
?	C1?DA	未指定
?	DB?E4	由 OEM 厂商指定
?	E5	未指定
?	E6	由 OEM 厂商指定
?	E7?E8	未指定
?	E9?F5	由 OEM 厂商指定
VK_ATTN	F6	Attn 键
VK_CRSEL	F7	CrSel 键
VK_EXSEL	F8	ExSel 键
VK_EREOF	F9	Erase EOF 键
VK_PLAY	FA	Play 键
VK_ZOOM	FB	Zoom 键
VK_NONAME	FC	为将来使用保留
VK_PA1	FD	PA1 键
VK_OEM_CLEAR	FE	Clear 键