EduSTAR eBOOK
中教育星电子图书馆
中教育星
电子图书馆 EBOOK

【VisualC++ 教室】

Edustar
中教育星软件股份有限公司

第一课　Windows 编程和面向对象技术

Microsoft Windows 是一个基于 Intel x86 微处理芯片的个人计算机上的具有图形用户接口的多任务和多窗口的操作系统，它是对 MS-DOS 操作系统的扩展和延伸。与 MS-DOS 操作系统相比，它有许多优越之处：首先，它提供了比 MS-DOS 字符界面更为直观、友好的图形用户界面；其次，它可以一次运行多个程序，方便了用户的操作，提高了机器的利用率；再次，Windows 环境下的应用程序具有一致的外观和用户接口，用户只需要熟悉其中一两个程序，就可以触类旁通学会使用别的 Windows 应用程序。另外，Windows 还具有更好的虚拟内存管理和设备无关特性等等。由于 Windows 具有以上突出优点，Windows 平台上的软件开发和程序设计已成主流。这一讲首先介绍 Windows 发展简史，然后分析 Windows 操作系统的特点以及 Windows 程序设计的关键概念，最后介绍 Windows 程序设计的过程及开发工具。

1.1　Windows 发展历史
1.2　Windows 操作系统特点
1.3　Windows 应用程序设计的特点
1.4　Windows 应用程序的开发工具
1.5　面向对象和 Windows 编程

1.1 Windows 发展历史

Windows 起源可以追溯到 Xerox 公司进行的工作。1970 年，美国 Xerox 公司成立了著名的研究机构 Palo Alto Research Center(PARC)，从事局域网、激光打印机、图形用户接口和面向对象技术的研究，并于 1981 年宣布推出世界上第一个商用的 GUI(图形用户接口)系统：Star 8010 工作站。但如后来许多公司一样，由于种种原因，技术上的先进性并没有给它带来它所期望的商业上的成功。

当时，Apple Computer 公司的创始人之一 Steve Jobs，在参观 Xerox 公司的 PARC 研究中心后，认识到了图形用户接口的重要性以及广阔的市场前景，开始着手进行自己的 GUI 系统研究开发工作，并于 1983 年研制成功第一个 GUI 系统：Apple Lisa。随后不久，Apple 又推出第二个 GUI 系统 Apple Macintosh，这是世界上第一个成功的商用 GUI 系统。当时，Apple 公司在开发 Macintosh 时，出于市场战略上的考虑，只开发了 Apple 公司自己的微机上的 GUI 系统，而此时，基于 Intel x86 微处理器芯片的 IBM 兼容微机已渐露峥嵘。这样，就给 Microsoft 公司开发 Windows 提供了发展空间和市场。

Microsoft 公司早就意识到建立行业标准的重要性，在 1983 年春季就宣布开始研究开发 Windows，希望它能够成为基于 Intel x86 微处理芯片计算机上的标准 GUI 操作系统。它在 1985 年和 1987 年分别推出 Windows 1.03 版和 Windows2.0 版。但是，由于当时硬件和 DOS 操作系统的限制，这两个版本并没有取得很大的成功。此后，Microsoft 公司对 Windows 的内存管理、图形界面做了重大改进，使图形界面更加美观并支持虚拟内存。Microsoft 于 1990 年 5 月份推出 Windows3.0 并一炮打红。这个“千呼万唤始出来”的操作系统一经面世便在商业上取得惊人的成功：不到 6 周，Microsoft 公司销出 50 万份 Windows3.0 拷贝，打破了任何软件产品的 6 周销售记录，从而一举奠定了 Microsoft 在操作系统上的垄断地位。

一年之后推出的 Windows3.1 对 Windows 3.0 作了一些改进，引入 TrueType 字体技术，这是一种可缩放的字体技术，它改进了性能；还引入了一种新设计的文件管理程序，改进了系统的可靠性。更重要的是增加对象链接合嵌入技术(OLE)和多媒体技术的支持。Windows3.0 和 Windows3.1 都必须运行于 MS DOS 操作系统之上。

随后，Microsoft 借 Windows 东风，于 1995 年推出新一代操作系统 Windows95(又名 Chicago)，它可以独立运行而无需 DOS 支持。Windows95 是操作系统发展史上一个里程碑式的作品，它对 Windows3.1 版作了许多重大改进，包括：更加优秀的、面向对象的图形用户界面，从而减轻了用户的学习负担；全 32 位的高性能的抢先式多任务和多线程；内置的对 Internet 的支持；更加高级的多媒体支持(声音、图形、影像等)，可以直接写屏并很好的支持游戏；即插即用，简化用户配置硬件操作，并避免了硬件上的冲突；32 位线性寻址的内存管理和良好的向下兼容性等等。以后我们提到的 Windows 一般均指 Windows95。

1.2 Windows 操作系统特点

Windows 之所以取得成功，主要在于它具有以下优点：直观、高效的面向对象的图形用户界面，易学易用：

从某种意义上说，Windows 用户界面和开发环境都是面向对象的。用户采用“选择对象-操作对象”这种方式进行工作。比如要打开一个文档，我们首先用鼠标或键盘选择该文档，然后从右键菜单中选择“打开”操作，打开该文档。这种操作方式模拟了现实世界的行为，易于理解、学习和使用。

用户界面统一、友好、漂亮：

Windows 应用程序大多符合 IBM 公司提出的 CUA (Common User Acess)标准，所有的程序拥有相同的或相似的基本外观，包括窗口、菜单、工具条等。用户只要掌握其中一个，就不难学会其他软件，从而降低了用户培训学习的费用。

丰富的设备无关的图形操作：

Windows 的图形设备接口(GDI)提供了丰富的图形操作函数，可以绘制出诸如线、圆、框等的几何图形，并支持各种输出设备。设备无关意味着在针式打印机上和高分辨率的显示器上都能显示出相同效果的图形。

多任务：

Windows 是一个多任务的操作环境，它允许用户同时运行多个应用程序，或在一个程序中同时做几件事情。每个程序在屏幕上占据一块矩形区域，这个区域称为窗口，窗口是可以重叠的。用户可以移动这些窗口，或在不同的应用程序之间进行切换，并可以在程序之间进行手工和自动的数据交换和通信。

虽然同一时刻计算机可以运行多个应用程序，但仅有一个是处于活动状态的，其标题栏呈现高亮颜色。一个活动的程序是指当前能够接收用户键盘输入的程序。

1.3 Windows 应用程序设计的特点

如前所述，Windows 操作系统具有 MS-DOS 操作系统无可比拟的优点，因而受到了广大软件开发人员的亲睐。但是，熟悉 DOS 环境下软件开发的程序员很快就会发现，Windows 编程与 DOS 环境下编程相比有很大的不同。Windows 要求以一种全新的思维方式进行程序设计，主要表现为以下几点：

1.3.1 事件驱动的程序设计

传统的 MS-DOS 程序主要采用顺序的、关联的、过程驱动的程序设计方法。一个程序是一系列预先定义好的操作序列的组合，它具有一定的开头、中间过程和结束。程序直接控制程序事件和过程的顺序。这样的程序设计方法是面向程序而不是面向用户的，交互性差，用户界面不够友好，因为它强迫用户按照某种不可更改的模式进行工作。它的基本模型如图 1.1 所示。

事件驱动程序设计是一种全新的程序设计方法，它不是由事件的顺序来控制，而是由事件的发生来控制，而这种事件的发生是随机的、不确定的，并没有预定的顺序，这样就允许程序的的用户用各种合理的顺序来安排程序的流程。对于需要用户交互的应用程序来说，事件驱动的程序设计有着过程驱动方法无法替代的优点。它是一种面向用户的程序设计方法，它在程序设计过程中除了完成所需功能之外，更多的考虑了用户可能的各种输入，并针对性的设计相应的处理程序。它是一种“被动”式程序设计方法，程序开始运行时，处于等待用户输入事件状态，然后取得事件并作出相应反应，处理完毕又返回并处于等待事件状态。它的框图如图 1.2 所示:在图中，输入界面 1-4 并没有固定的顺序，用户可以随机选取，以任何合理的顺序来输入数据。

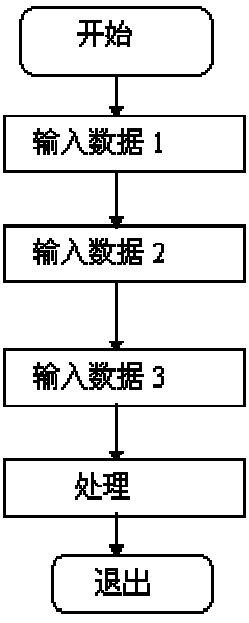


图 1.1 过程驱动模型

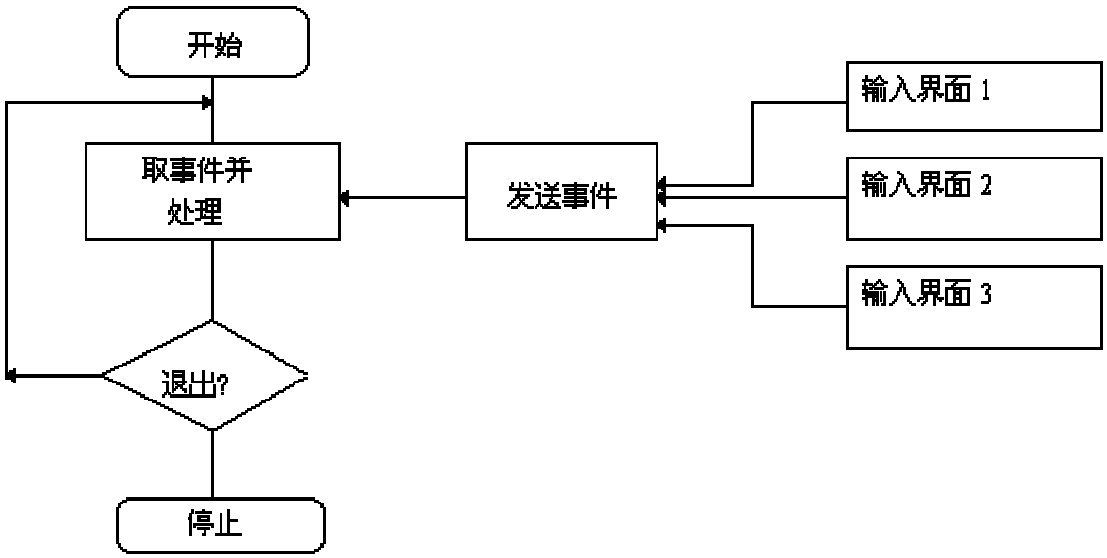


图 1.2 事件驱动程序模型

1.3.2 **消息循环与输入**

事件驱动围绕着消息的产生与处理展开，一条消息是关于发生的事件的消息。事件驱动是靠消息循环机制来实现的。

消息是一种报告有关事件发生的通知。

消息类似于 DOS **下的用户输入，但比** DOS **的输入来源要广，** Windows **应用程序的消息来源有以下四种：**(1)**输入消息：包括键盘和鼠标的输入。这一类消息首先放在系统消息队列中，然后由** Windows **将它们送入应用程序消息队列中，由应用程序来处理消息。**

(2)**控制消息：用来与** Windows **的控制对象，如列表框、按钮、检查框等进行双向通信。当用户在列表框中改动当前选择或改变了检查框的状态时发出此类消息。这类消息一般不经过应用程序消息队列，而是直接发送到控制对象上去。**

(3)**系统消息：对程序化的事件或系统时钟中断作出反应。一些系统消息，象** DDE **消息(动态数据交换消息)要通过** Windows **的系统消息队列，而有的则不通过系统消息队列而直接送入应用程序的消息队列，如创建窗口消息。**

(4)**用户消息：这是程序员自己定义并在应用程序中主动发出的，一般由应用程序的某一部分内部处理。**

在 DOS **应用程序下，可以通过** getchar()、getch()**等函数直接等待键盘输入，并直接向屏幕输出。而在** Windows **下，由于允许多个任务同时运行，应用程序的输入输出是由** Windows **来统一管理的。**

Windows **操作系统包括三个内核基本元件：**GDI, KERNEL ,USER**。其中** GDI(**图形设备接口**)**负责在屏幕上绘制像素、打印硬拷贝输出，绘制用户界面包括窗口、菜单、对话框等。系统内核** KERNEL **支持与操作系统密切相关的功能：如进程加载，文本切换、文件** I/O**，以及内存管理、线程管理等。**USER **为所有的用户界面对象提供支持，它用于接收和管理所有输入消息、系统消息并把它们发给相应的窗口的消息队列。消息**

队列是一个系统定义的内存块，用于临时存储消息；或是把消息直接发给窗口过程。每个窗口维护自己的消息队列，并从中取出消息，利用窗口函数进行处理。框图如下：

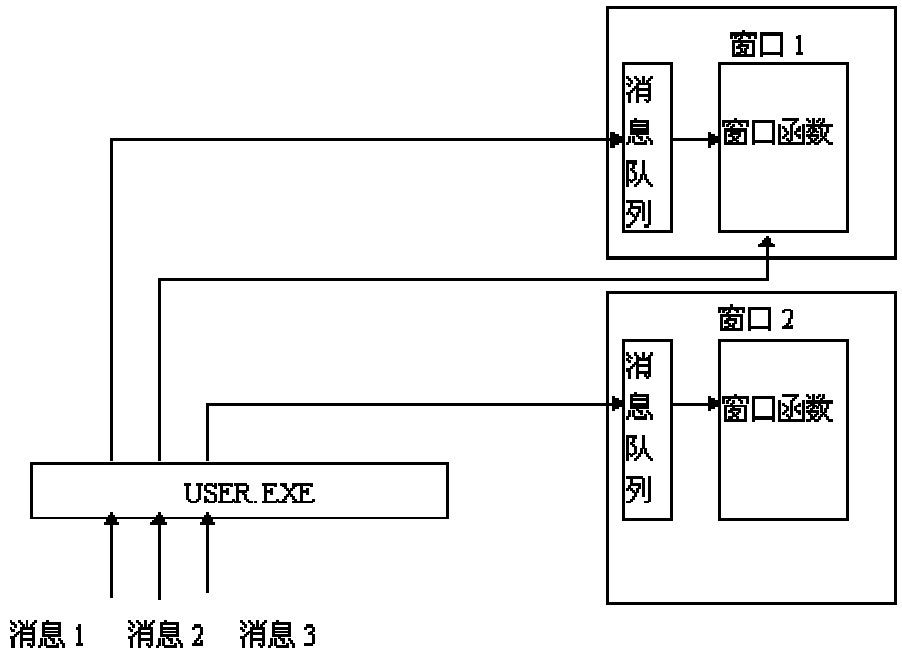


图 1.3 消息驱动模型

1.3.3 图形输出

Windows 程序不仅在输入上与 DOS 程序不同，而且在程序输出上也与 DOS 有着很大不同，主要表现为：1.DOS 程序独占整个显示屏幕，其他程序在后台等待。而 Windows 的每一个应用程序对屏幕的一部分进行处理。DOS 程序可以直接往屏幕上输出，而 Windows 是一个多窗口的操作系统，由操作系统来统一管理屏幕输出；每个窗口要输出内容时，必须首先向操作系统发出请求(GDI 请求)，由操作系统完成实际的屏幕输出工作。

2.Windows 程序的所有输出都是图形。Windows 提供了丰富的图形函数用于图形输出，这对输出图形是相当方便的，但是由于字符也被作为图形来处理，输出时的定位要比 DOS 复杂的多。

比如，在 DOS 字符方式下，我们可以写出如下程序用于输出两行文字：printf(“Hello,\n”);printf(“This is DOS program.\n”);而在 Windows 下要输出这两行文字所做的工作要复杂的多。因为 Windows 输出是基于图形的，它输出文本时不会象 DOS 那样自动换行，而必须以像素为单位精确定位每一行的输出位置。另外，由于 Windows 提供了丰富的字体，所以在计算坐标偏移量时还必须知道当前所用字体的高度和宽度。

3.Windows 下的输出是设备无关的。在 DOS 下编写过 Foxpro 程序的读者常常会有这样的体会，在编写打印报表程序时，要针对不同的打印机在程序中插入不同的打印控制码，用以控制换页、字体设置等选项。这样的程序编写起来繁琐，而且不容易移植(因为换一台不同型号的打印机就要重新修改程序)。而 Windows 下的应用程序使用图形设备接口(GDI)

来进行图形输出。GDI 屏蔽了不同设备的差异，提供了设备无关的图形输出能力，Windows 应用程序只要发出设备无关的 GDI 请求(如调用 Rectangle 画一个矩形)，由 GDI 去完成实际的图形输出操作。对于一台具有打印矩形功能的 PostScript 打印机来说，GDI 可能只需要将矩形数据传给驱动程序就可以了，然后由驱动程序产生 PostScript 命令绘制出相应的矩形；而对于一台没有矩形输出功能的点阵打印机来说，GDI 可能需要将矩形转化为四条线，然后向驱动程序发出画线的指令，在打印机上输出矩形。当然，这两种输出在用户看来并没有什么区别。

Windows 的图形输出是由图形设备接口(GDI)来完成的，GDI 是系统原始的图形输出库，它用于在屏幕上输出像素、在打印机上输出硬拷贝以及绘制 Windows 用户界面。

GDI 提供两种基本服务：创建图形输出和存储图象。GDI 提供了大量用于图形输出的函数，这些函数接收应用程序发出来的绘图请求、处理绘图数据并根据当前使用设备调用相应的设备驱动程序产生绘图输出。这些绘图函数分为三类：一是文字输出，二是矢量图形函数，用于画线、圆等几何图形，三是光栅(位图)图形函数，用于绘制位图。

GDI 识别四种类型的设备：显示屏幕、硬拷贝设备(打印机、绘图机)、位图和图元文件。前两者是物理设备，后两者是伪设备。一个伪设备提供了一种在 RAM 里或磁盘里存储图象的方法。位图存放的是图形的点位信息，占用较多的内存，但速度很快。图元文件保存的是 GDI 函数的调用和调用参数，占用内存较少，但依赖于 GDI，因此不可能用某个设备来创建图元文件，而且速度比位图要慢。

GDI 的图形输出是面向窗口的，面向窗口包含两层含义：(1)每个窗口作为一个独立的绘图接口来处理，有它自己的绘图坐标。当程序在一个窗口中绘图时，首先建立缺省的绘图坐标，原点(0，0)位于窗口用户区的左上角。每个窗口必须独立的维护自己的输出。

(2)绘图仅对于本窗口有效，图形在窗口边界会被自动裁剪，也就是说窗口中的每一个图形都不会越出边界。即使想越出边界，也是不可能的，窗口会自动的防止其他窗口传过来的任何像素。这样，你在窗口内绘图时，就不必担心会偶然覆盖其他程序的窗口，从而保证了 Windows 下同时运行多个任务时各个窗口的独立性。

1.3.4 用户界面对象

Windows 支持丰富的用户接口对象，包括：窗口、图标、菜单、对话框等等。程序员只需简单的几十行代码，就可以设计出一个非常漂亮的图形用户界面。而在 DOS 环境下，则需要大量的代码来完成同样的工作，而且效果也没有 Windows 提供的那么好。下面我们介绍一下用户界面对象中的一些术语和相关概念。

窗口

窗口是用户界面中最重要的部分。它是屏幕上与一个应用程序相对应的矩形区域，是用户与产生该窗口的应用程序之间的可视界面。每当用户开始运行一个应用程序时，应用程序就创建并显示一个窗口；当用户操作窗口中的对象时，程序会作出相应反应。用户通过关闭一个窗口

来终止一个程序的运行；通过选择相应的应用程序窗口来选择相应的应用程序。一个典型的窗口外观如图 1.4 所示。

图 1.4 窗口

边框

绝大多数窗口都有一个边框，用于指示窗口的边界。同时也用来指明该窗口是否为活动窗口，当窗口活动时，边框的标题栏部分呈高亮显示。用户可以用鼠标拖动边框来调整窗口的大小。

系统菜单框

系统菜单框位于窗口左上角，以当前窗口的图标方式显示，用鼠标点一下该图标(或按 ALT+空格键)就弹出系统菜单。系统菜单提供标准的应用程序选项，包括：Restore(还原窗口原有的大小)，Move(使窗口可以通过键盘上的光标键来移动其位置)，Size(使用光标键调整窗口大小)，Minimize(将窗口缩成图标)，Maximize(最大化：使窗口充满整个屏幕)和 Close(关闭窗口)。

标题栏

标题栏位于窗口的顶部，其中显示的文本信息用于标注应用程序，一般是应用程序的名字，以便让用户了解哪个应用程序正在运行。标题栏颜色反映该窗口是否是一个活动窗口，当为活动窗口时，标题栏呈现醒目颜色。鼠标双击标题栏可以使窗口在正常大小和最大化状态之间切换。在标题栏上按下鼠标器左键可以拖动并移动该窗口，按右键弹出窗口系统菜单。

菜单栏

菜单栏位于标题栏下方，横跨屏幕，在它上面列出了应用程序所支持的命令，菜单栏中的项是命令的主要分类，如文件操作、编辑操作。从菜单栏中选中某一项通常会显示一个弹出菜单，其中的项是对应于指定分类中的某个任务。通过选择菜单中的一个项(菜单项)，用户可以向程序发出命令，以执行某一功能。如选择“ 文件->打开... ”菜单项会弹出一

个打开文件对话框，让用户选择一个文件，然后打开这个文件。

一般的，以“ ... ”结尾的菜单项文本表明选择该项时会弹出一个对话框，让用户输入信息，然后执行操作，如“ 文件->打开... ”。若不以“ ... ”结尾，则表明选择该菜单项直接执行一个动作，如“ 编辑 ”菜单下的“ 粘贴 ”菜单项。若一个菜单项呈现灰色，则表明该菜单当前不可用。有时菜单项上还有加速键，加速键是一种键盘组合，它是菜单项的一种替代方式，可以让用户通过键盘直接发出命令；在键盘上按下这一键盘组合，就等效于选择了相应的菜单。如“ 粘贴(P) CTRL+V ”，就表示粘贴操作的加速键是 CTRL+V，按下 CTRL+V 就执行粘贴操作。

工具条

工具条一般位于菜单栏下方，在它上面有一组位图按钮，代表一些最常用的命令。工具条可以显示或隐藏。让鼠标在某个按钮上停一会儿，在按钮下方会出现一个黄色的小窗口，里面显示关于该按钮的简短说明，叫做工具条提示(ToolTip)。用户还可以用鼠标拖动工具条将其放在窗口的任何一侧。

客户区

客户区是窗口中最大的一块空白矩形区域，用于显示应用程序的输出。例如，字处理程序在客户区中显示文档的当前页面。应用程序负责客户区的绘制工作，而且只有和该窗口相对应的应用程序才能向该用户区输出。

垂直滚动条和水平滚动条

垂直滚动条和水平滚动条分别位于客户区的左侧和底部，它们各有两个方向相反的箭头和一个深色的长度可变的滚动块。可以用鼠标选中滚动条的箭头上下卷滚(选中垂直滚动条时)或水平卷滚(选中水平滚动条时)客户区的内容。滚动块的位置表示客户区中显示的内容相对于要显示的全部内容的位置，滚动块的长度表示客户区中显示的内容大小相对于全部内容大小的比例。

状态栏

状态栏是一般位于窗口底部，用于输出菜单的说明和其他一些提示信息(如鼠标器位置、当前时间、某种状态等)。

图标

图标是一个用于提醒用户的符号，它是一个小小的图象，用于代表一个应用程序。当一个应用程序的主窗口缩至最小时，就呈现为一个图标。

光标

Windows 的光标是显示屏上的一个位图，而不是 DOS 下的一条下划线。光标用于响应鼠标或其他定位设备的移动。程序可以通过改变光标的形状来指出系统中的变化。例如，程序常显示一个计时的光标，用于指示用户一些漫长的操作正在进行之中。程序也可以通过改变光标让用户知道程序进入了一种特殊模式，例如，绘图程序经常改变光标来反映被绘制对象的类型，是直线还是圆或其他。

插入符

插入符(caret)是一个微小并闪烁的位图，作为一个键盘控制的指针。控制键盘输入的窗口可以创建一个插入符去通知用户：窗口现在可以进行键盘输入。在 DOS 环境下，一般使用“光标”作为键盘指针，而在 Windows 中，“光标”被作为鼠标指针。

应用程序必须维护这个插入符。在 Windows 中，在一个时间只允许有一个插入符存在。因此，要使用插入符号作为键盘指针的应用程序必须在取得焦点时创建一个插入符号，并在失去焦点后删除它。

对话框

对话框是一种特殊的窗口，它提供了一种接收用户输入、处理数据的标准方法。特别的，当用户输入了一个需要附加信息 的命令时，对话框是接收输入的标准方法。比如，假设用户要求系统打开一个文件，对话框就可以提供一个让用户从一组文件中选择一个文件的标准方法。如前所述，在一般情况下，在选择菜单名字后面跟着省略号(...)的菜单项通常会弹出一个对话框。图 1.5 给出了查找对话框的一个例子。

图 1.5 查找对话框

控件

在图 1.5 中，查找对话框是一个独立的窗口，它显示信息并接收用户的输入。在对话框中，还包含了许多小的窗口，这些窗口被称为控件。控件是应用程序用来获得用户特定信息的窗口，比如要打开文件的名字或自动换行的设置等。应用程序也会通过控件获取所需的信息，以便控制程序的某种属性，如自动换行特性的开关。

控件总是与其他窗口连用，典型的是对话框，但也可以用在普通窗口之中。常见的控件有：按钮、编辑框、列表框、组合框、静态文本等等。

消息框

消息框是用于给用户一些提示或警告的窗口。例如，消息框能够在应用程序执行某项任务过程中出现问题时通知用户。下图所示的对话框警告用户输入了一个不合法的文件名。

图 1.6

1.3.5 资源共享

对于 DOS 程序来说，它运行时独占系统的全部资源，包括显示器、内存等，在程序结束时才释放资源。而 Windows 是一个多任务的操作系统，各个应用程序共享系统提供的资源，常见的资源包括：设备上下文，画刷，画笔，字体，对话框控制，对话框，图标，定时器，插入符号，通信端口，电话线等。

Windows 要求应用程序必须以一种能允许它共享 Windows 资源的方式进行设计，它的基本模式是这样的：1.向 Windows 系统请求资源；2.使用该资源；3.释放该资源给 Windows 以供别的程序使用。

即使最有经验的 Windows 程序员也常常会忽略第三步。如果忽略了这一步，轻则当时不出错，但过一会儿出现程序运行出现异常情况，或干扰别的程序正常运行；重则立即死机，比如设备上下文没有释放时。

在 Windows 应用程序设计中，CPU 也是一种非常重要的资源，因此应用程序应当避免长时间的占用 CPU 资源(如一个特别长的循环)；如果确实需要这样做，也应当采取一些措施，以让程序能够响应用户的输入。主存也是一个共享资源，要防止同时运行的多个应用程序因协调不好而耗尽内存资源。

应用程序一般不要直接访问内存或其他硬件设备，如键盘、鼠标、计数器、屏幕或串口、并口等。Windows 系统要求绝对控制这些资源，以保证向所有的应用程序提供公平的不中断的运行。如果确实要访问串并口，应当使用通过 Windows 提供的函数来安全的访问。

1.3.6 Windows 应用程序组成

前面介绍了 Windows 应用程序的特点，现在让我们看看编写一个 Windows 程序需要做哪些工作。编写一个典型的 Windows 应用程序，一般需要：1.C,CPP 源程序文件：源程序文件包含了应用程序的数据、类、功能逻辑模块(包括事件处理、用户界面对象初始化以及一些辅助例程)的定义。

2.H,HPP 头文件：头文件包含了 CPP、C 源文件中所有数据、模块、类的声明。当一个 CPP、C 源文件要调用另一个 CPP、C 中所定义的模块功能时，需要包含那个 CPP、C 文件对应的头文件。

3.资源文件：包含了应用程序所使用的全部资源定义，通常以.RC 为后缀名。注意这里说的资源不同与前面提到的资源，这里的资源是应用程序所能够使用的一类预定义工具中的一个对象，包括：字符串资源、加速键表、对话框、菜单、位图、光标、工具条、图标、版本信息和用户自定义资源等。

其中 CPP、C 和头文件同 DOS 下的类似，需要解释的是资源文件。在 DOS 程序设计过程中，所有的界面设计工作都在源程序中完成。而在 Windows 程序设计过程中，象菜单、对话框、位图等可视的对象被单独分离出来加以定义，并存放在资源源文件中，然后由资源编译程序编译为应用程序所能使用的对象的映象。资源编译使应用程序可以读取对象的二进制映象和具体数据结构，这样可以减轻为创建复杂对象所需要得程序设计工作。

程序员在资源文件中定义应用程序所需使用的资源，资源编译程序

编译这些资源并将它们存储于应用程序的可执行文件或动态连接库中。在 Windows 应用程序中引入资源有以下一些好处：1.降低内存需求：当应用程序运行时，资源并不随应用程序一起装入内存，而是在应用程序实际用到这些资源时才装入内存。在资源装入内存时，它们拥有自己的数据段，而不驻留于应用程序数据段中；当内存紧张时，可以废弃这些资源，使其占用的内存空间供他用，而当应用程序用到这些资源时才自动装入，这种方式降低了应用程序的内存需求，使一次可运行更多的程序，这也是 Windows 内存管理的优点之一。

2.便于统一管理和重复利用：将位图、图标、字符串等按资源文件方式组织便于统一管理和重用。比如，将所有的错误信息放到资源文件里，利用一个函数就可以负责错误提示输出，非常方便。如果在应用程序中要多次用到一个代表公司的徽标位图，就可以将它存放在资源文件中，每次用到时再从资源文件中装入。这种方式比将位图放在一个外部文件更加简单有效。

3.应用程序与界面有一定的独立性，有利于软件的国际化：由于资源文件独立于应用程序设计，使得在修改资源文件时(如调整对话框大小、对话框控制位置)，可以不修改源程序，从而简化了用户界面的设计。另外，目前所提供的资源设计工具一般都是采用“所见即所得”方式，这样就可以更加直观、可视的设计应用程序界面。由于资源文件的独立性，软件国际化工作也非常容易。比如，现在开发了一个英文版的应用程序，要想把它汉化，只需要修改资源文件，将其中的对话框、菜单、字符串资源等汉化即可，而无需直接修改源程序。

但是，应用程序资源只是定义了资源的外观和组织，而不是其功能特性。例如，编辑一个对话框资源，可以改变对话框的安排和外观，但是却没有也不可能改变应用程序响应对话框控制的方式。外观的改变可以通过编辑资源来实现，而功能的改变却只能通过改变应用程序的源代码，然后重新编译来实现。

Windows 应用程序的生成同 DOS 下类似，也要经过编译、链接两个阶段，只是增加了资源编译过程，基本流程如下图：

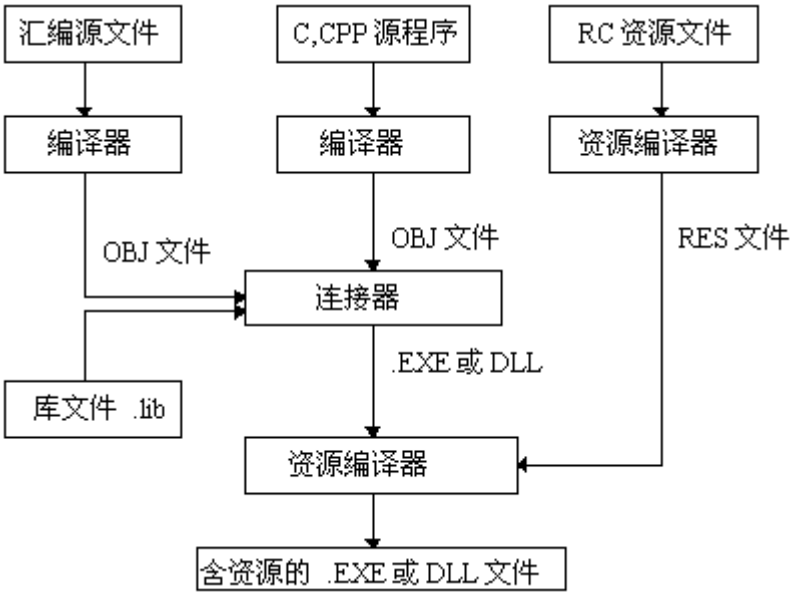


图 1.7 **应用程序生成过程**

C、CPP **编译器将** C **源程序编译成目标程序，然后使用连接程序将所有的目标程序(包括各种库)连接在一起，生成可执行程序。在制作** Windows **应用程序时，编译器还要为引出函数生成正确的入口和出口代码。**

连接程序生成的可执行文件还不能在 Windows **环境下运行，必须使用资源编译器对其进行处理。资源编译器对可执行文件的处理是这样的：如果该程序有资源描述文件，它就把已编译为二进制数据的资源加入到可执行文件中；否则，仅对该可执行文件进行相容性标识。应用程序必需经过资源编译器处理才可以在** Windows **环境下运行。**

[**上一页**][**下一页**]

本教程由 Visual C++**王朝**(Where programmers come together)**协助制作**,1999 **未经许可，请勿以任何形式复制或转载**/

1.4 Windows 应用程序的开发工具

由于 Windows 是 Microsoft 的产品，因而在早期阶段，开发工具只有 Microsoft C 和 SDK(Software Developer Kit:软件开发工具包)可供使用。利用 SDK 进行 Windows 程序的设计开发非常繁琐、复杂，代码可重用性差，工作量大，即便一个简单的窗口也需要几百行程序，令开发人员望而生畏。

随着 Windows 的逐渐普及，各大软件公司纷纷推出自己的 Windows 软件开发工具。国内用户比较熟悉的有 Borland C++2.0 以上版本以及用于数据库开发的 Foxpro 等等。其中 Borland C++支持面向对象的开发，在我国具有广大的用户群。

可视化技术和 CASE 技术研究的深入为我们带来了支持可视化编程特性的第三代开发工具，这一代开发工具有：Visual Basic,Visual C++,Borland C++ Builder,Delphi 和用于数据库开发的 PowerBuilder、Visual Foxpro 等等。

其中，Visual C++是美国 Microsoft 公司推出的 4GL 软件开发工具，目前已成为国内应用最广泛的高级程序设计语言之一，最新版本为 5.0 版。同其他软件开发工具相比，Visual C++具有以下优点：面向对象、可视化开发：提供了面向对象的应用程序框架 MFC(Microsoft Foundation Class:微软基础类库)，大大简化了程序员的编程工作，提高了模块的可重用性。Visual C++还提供了基于 CASE 技术的可视化软件自动生成和维护工具 AppWizard、ClassWizard、Visual Studio、WizardBar 等，帮助用户直观的、可视地设计程序的用户界面，可以方便的编写和管理各种类，维护程序源代码，从而提高了开发效率。用户可以简单而容易地使用 C/C++编程。

众多的开发商支持以及业已成为工业标准的 MFC 类库：MFC 类库已经成为事实上的工业标准类库，得到了众多开发商和软件开发工具的支持；另外，由于众多的开发商都采用 Visual C++进行软件开发，这样用 Visual C++开发的程序就与别的应用软件有许多相似之处，易于学习和使用。

Visual C++封装了 Windows 的 API(应用程序接口)函数、USER、KERNEL、GDI 函数，帮助我们弄清了许多函数的组织方法，隐去了创建、维护窗口的许多复杂的例行工作，简化了编程。

但是，由于 C/C++本身的复杂性，Visual C/C++对编程人员要求还是相当高的。它首先要求编程者要具有丰富的 C/C++语言编程经验，了解面向对象编程的基本概念，同时还必须掌握复杂的 MFC 类库。

1.5 面向对象和 Windows 编程

面向对象技术是目前流行的系统设计开发技术，它包括面向对象分析和面向对象程序设计。面向对象程序设计技术的提出，主要是为了解决传统程序设计方法——结构化程序设计所不能解决的代码重用问题。

结构化程序设计从系统的功能入手，按照工程的标准和严格的规范将系统分解为若干功能模块，系统是实现模块功能的函数和过程的集合。由于用户的需求和软、硬件技术的不断发展变化，按照功能划分设计的系统模块必然是易变的和不稳定的。这样开发出来的模块可重用性不高。

面向对象程序设计从所处理的数据入手，以数据为中心而不是以服务(功能)为中心来描述系统。它把编程问题视为一个数据集合，数据相对于功能而言，具有更强的稳定性。

面向对象程序设计同结构化程序设计相比最大的区别就在于：前者首先关心的是所要处理的数据，而后者首先关心的是功能。

面向对象程序设计是一种围绕真实世界的概念来组织模型的程序设计方法，它采用对象来描述问题空间的实体。关于对象这一概念，目前还没有统一的定义。一般的认为，对象是包含现实世界物体特征的抽象实体，它反映了系统为之保存信息和(或)与它交互的能力。它是一些属性及服务的一个封装体，在程序设计领域，可以用“对象=数据+作用于这些数据上的操作”这一公式来表达。

类是具有相同操作功能和相同的数据格式(属性)的对象的集合。类可以看作抽象数据类型的具体实现。抽象数据类型是数据类型抽象的表示形式。数据类型是指数据的集合和作用于其上的操作的集合，而抽象数据类型不关心操作实现的细节。从外部看，类型的行为可以用新定义的操作加以规定。类为对象集合的抽象，它规定了这些对象的公共属性和方法；对象为类的一个实例。苹果是一个类，而放在桌上的那个苹果则是一个对象。对象和类的关系相当于一般的程序设计语言中变量和变量类型的关系。

消息是向某对象请求服务的一种表达方式。对象内有方法和数据，外部的用户或对象对该对象提出的服务请求，可以称为向该对象发送消息。合作是指两个对象之间共同承担责任和分工。

面向对象的编程方法具有四个基本特征：

1.抽象：

抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。比如，我们要设计一个学生成绩管理系统，考察学生这个对象时，我们只关心他的班级、学号、成绩等，而不用去关心他的身高、体重这些信息。抽象包括两个方面，一是过程抽象，二是数据抽象。过程抽象是指任何一个明确定义功能的操作都可被使用者看作单个的实体看待，尽管这个操作实际上可能由一系列更低级的操作来完成。数据抽象定义了数据类型和施加于该类型对象上的操作，并限定了对象的值只能通过使用这些操作修改和观察。

2.继承：

继承是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类(子类)，而原始类称为新类的基类(父类)。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。这也体现了大自然中一般与特殊的关系。继承性很好的解决了软件的可重用性问题。比如说，所有的 Windows 应用程序都有一个窗口，它们可以看作都是从一个窗口类派生出来的。但是有的应用程序用于文字处理，有的应用程序用于绘图，这是由于派生出了不同的子类，各个子类添加了不同的特性。

3.封装：

封装是面向对象的特征之一，是对象和类概念的主要特性。封装是把过程和数据包围起来，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口访问其他对象。一旦定义了一个对象的特性，则有必要决定这些特性的可见性，即哪些特性对外部世界是可见的，哪些特性用于表示内部状态。在这个阶段定义对象的接口。通常，应禁止直接访问一个对象的实际表示，而应通过操作接口访问对象，这称为信息隐藏。事实上，信息隐藏是用户对封装性的认识，封装则为信息隐藏提供支持。封装保证了模块具有较好的独立性，使得程序维护修改较为容易。对应用程序的修改仅限于类的内部，因而可以将应用程序修改带来的影响减少到最低限度。

4. 多态性：

多态性是指允许不同类的对象对同一消息作出响应。比如同样的加法，把两个时间加在一起和把两个整数加在一起肯定完全不同。又比如，同样的选择编辑-粘贴操作，在字处理程序和绘图程序中有不同的效果。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。

面向对象程序设计具有许多优点：开发时间短，效率高，可靠性高，所开发的程序更强壮。由于面向对象编程的可重用性，可以在应用程序中大量采用成熟的类库，从而缩短了开发时间。

应用程序更易于维护、更新和升级。继承和封装使得应用程序的修改带来的影响更加局部化。

Windows 的最初开发是在 80 年代早期，在 C++出现之前，但是当时已经提出了面向对象式的程序设计思想。Windows 的开发者们已经意识到将界面上不同的项看作对象的好处，但是他们仍然被迫采用传统的 C 语言来处理这些对象。在 Windows 的界面设计和软件开发环境中，可以说处处贯穿着面向对象的思想。

在 Windows 中，程序的基本单位不是过程和函数，而是窗口。一个窗口是一组数据的集合和处理这些数据的方法和窗口函数。从面向对象的角度来看，窗口本身就是一个对象。Windows 程序的执行过程本身就

是窗口和其他对象的创建、处理和消亡过程。Windows 中的消息的发送可以理解为一个窗口对象向别的窗口对象请求对象的服务过程。因此，用面向对象方法来进行 Windows 程序的设计与开发是极其方便的和自然的。

采用面向对象的方法来进行 Windows 程序设计还可以简化对资源的管理。当我们将资源映射成一个 C++对象时，对资源的使用可以翻译成以下 C++顺序：

1.创建一个对象：如定义一个画笔对象

2.使用对象：用画笔绘图

3.撤消该对象

一个对象的创建是对一个对象的定义过程，可以由对象的构造函数处理对资源的请求过程。当某一个对象退出活动范围时，它的撤消可以由编译器来自动管理。各种资源和 Windows 结构都能以这种方式处理，如设备上下文、画笔、字体、画刷等等。

小 结

在这一讲中，我们主要向读者介绍了：

Windows 操作系统的特点：面向对象的图形用户界面，一致的用户接口，设备无关的图形输出以及多任务。

Windows 编程的四个特点：事件驱动、消息循环、图形输出、资源共享

Windows 的基本用户界面对象：包括窗口、标题栏、图标、光标、插入符号、对话框、控件等。

Windows 应用程序的基本组成和生成过程：Windows 应用程序包含 C、CPP、头文件和资源文件；Windows 应用程序的生成要经过编译和连接两个阶段。

Windows 应用程序的开发工具 Visual C++

面向对象编程的基础知识和运用面向对象技术进行 Windows 编程：对象、类、消息的概念；面向对象的编程特征：抽象，继承，封装和多态性。在 Windows 编程中，窗口本身就是一个重要的对象。

第二课　使用 Visual C++ 5.0

在这一章里，我们将介绍 Visual C++的集成开发环境 Visual Studio 及其组件，以及使用 Visual C++基础类库 MFC 编程的一些基础知识。

2.1 Visual C++可视化集成开发环境

2.2 创建、组织文件、工程和工作区

2.3 WIN32 开发

2.4 MFC 编程

2.5 移植 C Windows 程序到 MFC

2.6 Visual C++5.0 新特性

2.1 Visual C++可视化集成开发环境

Visual C++提供了一个支持可视化编程的集成开发环境：Visual Studio(又名 Developer Studio)。Developer Studio 是一个通用的应用程序集成开发环境，它不仅支持 Visual C++，还支持 Visual Basic,Visual J++,Visual InterDev 等 Microsoft 系列开发工具。Developer Studio 包含了一个文本编辑器、资源编辑器、工程编译工具、一个增量连接器、源代码浏览器、集成调试工具，以及一套联机文档。使用 Developer Studio，可以完成创建、调试、修改应用程序等的各种操作。

Developer Studio 采用标准的多窗口 Windows 用户界面，并增加了一些新特性，使得开发环境更易于使用，用户很容易学会它的使用方法。一个典型的 Developer Studio 用户界面如图 2.1 所示。

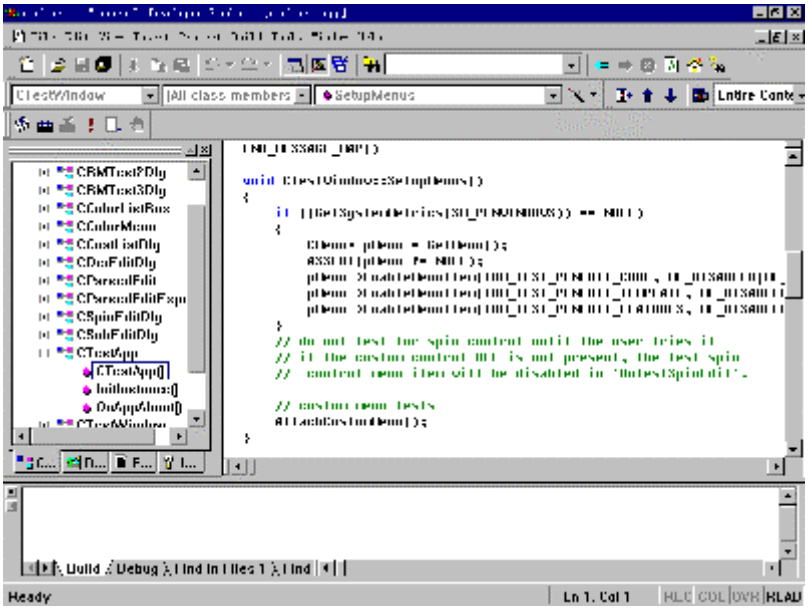


图 2.1 Developer Studio 用户界面

由于 Developer Studio 是一个可视化的开发工具，在介绍 Developer Studio 的各个组成部分之前，首先了解一下可视化编程的概念。可视化技术是当前发展迅速并引人注目的技术之一，它的特点是把原来抽象的数字、表格、功能逻辑等用直观的图形、图象的形式表现出来。可视化编程是它的重要应用之一。所谓可视化编程，就是指：在软件开发过程中，用直观的具有一定含义的图标按钮、图形化的对象取代原来手工的抽象的编辑、运行、浏览操作，软件开发过程表现为鼠标点击按钮和拖放图形化的对象以及指定对象的属性、行为的过程。这种可视化的编程方法易学易用，而且大大提高了工作效率。

Visual C++的集成开发环境 Developer Studio 提供了大量的实用工具以支持可视化编程特性，它们包括 项目工作区、ClassWizard、AppWizard、WizardBar、Component Gallery 等。下面我们将对它们作逐一介绍：

2.1.1 项目工作区

项目工作区是 Developer Studio 的一个最重要的组成部分，程序员的大部分工作都在 Developer Studio 中完成。Developer Studio 使用项目工

作区来组织项目、元素以及项目信息在屏幕上出现的方式。在一个项目工作区中，可以处理：

一个工程和它所包含的文件

一个工程的子工程

多个相互独立的工程

多个相互依赖的工程

一个项目工作区可包含由不同的开发工具包生成的工程，如 Visual C++和 Visual J++。在桌面上，项目工作区以窗口方式组织项目、文件和项目设置。项目工作区窗口一般位于屏幕左侧，如图 2.2 所示。项目工作区窗口底部有一组标签，用于从不同的角度(视图)察看项目中包含的工程和联机文档。

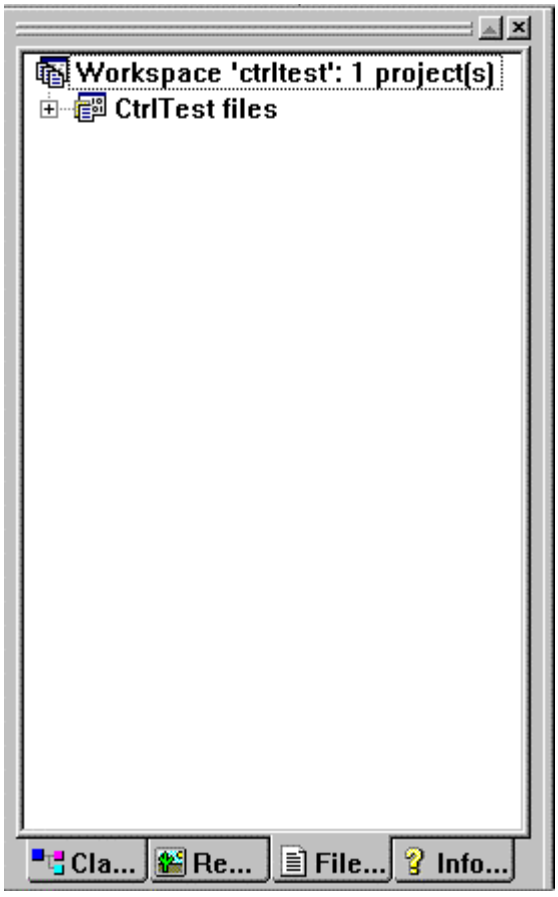


图 2.2　项目工作区窗口

每个项目视图都有一个相应的文件夹，包含了关于该项目的各种元素。展开该文件夹可以显示该视图方式下工作区的详细信息。项目工作区包含四种视图：

FileView(文件视图)：显示所创建的工程。展开文件夹可以察看工程中所包含的文件。

ClassView(类视图)：显示项目中定义的 C++类，展开文件夹显示工程中所定义的所有类，展开类可察看类的数据成员和成员函数以及全局变量、函数和类型定义。

ResourceView(资源视图)：显示项目中所包含的资源文件。展开文件

夹可显示所有的资源类型。

InfoView(文档视图)：显示联机文档目录表。展开目录表可以显示所有的帮助主题，双击主题将弹出 InfoViewer Topic 窗口，显示关于该主题的详细信息。要显示关于源程序窗口的关键字的相关信息，可以将光标移动到该关键字上，然后按下 F1 键。还可以使用 InfoViewer 显示来自 Internet WWW(万维网)的页面。

单击项目工作区底部的标签可以从一个视图切换到另一个视图。

每个视图都是按层次方式组织的。可以展开文件夹和其中的项察看其内容，或折叠起来察看其组织结构。在项目视图中，如果一项不可以再展开，那么它是可编辑的。双击这一项便可以打开相应的文档编辑器进行编辑：对类和源程序文件来说，是打开文本编辑器，对于对话框来说是打开对话框编辑器等。每个视图还支持右键快捷菜单。

使用文件视图(FileView)

FileView 窗格显示了工程文件和项目工作区中所包含的文件的逻辑关系。一个工作区可以包含多个工程，其中活动工程以黑体显示。活动配置决定了编译活动工程时的编译选项。活动工程是使用 Build 或 Rebuild All 时要编译的那一个工程。可以用 Build 菜单上的 Set Active Configuration 选择不同的活动配置；也可以在 Project 菜单上用 Set Active Project 选择不同的活动工程。

使用 FileView 可以：

察看文件；

管理文件：包括增加、删除、移动、重命名、拷贝文件等。

要增加一个文件到过程中，可以选择 Project->Add to Project->Files 菜单，弹出文件对话框，选择相应文件即可；要从工程中删除一个文件，可打开工程文件夹，选择相应文件，然后按 DEL 键。

使用类视图(ClassView)

ClassView 显示所有已定义的类以及这些类中的数据成员、成员变量。Visual C++自动从项目工作区中所包含的源程序文件中分离出类。

在 ClassView 中，文件夹代表工程文件名。展开 ClassView 顶层的文件夹后，显示工程中所包含的所有的类，如图 2.3。双击一个类的图标(或单击图标旁的+号)时，ClassView 展开该类并显示其类成员。

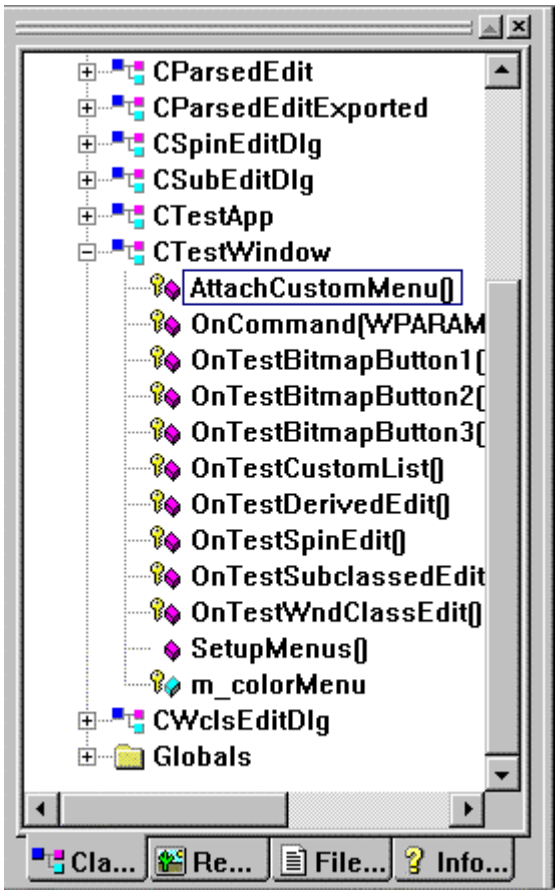


图 2.3　**类视图**

ClassView **使用图标标识类、类成员和工程中的其他项，图** 2.4 **显示了所有的图标和含义：**

图标	含义
	类
	保护类成员函数
	私有类成员函数
	公有类成员函数
	保护类数据成员
	私有类数据成员
	公有类数据成员

图 2.4　**类视图中图标的含义**

利用 ClassView **不仅可以浏览应用程序所包含的类以及类中的成员，还可以快速跳到一个类或成员的定义，步骤如下：**

1.选择要查找的定义或声明的符号

2.双击所选的符号名。

要打开关于某一个类声明的头文件，只需双击类名即可。

要想查找某一个变量的参考：

1.将光标定位在该符号处

2.按鼠标右键，弹出快捷菜单，选择 Reference(此时应确保已经生成了 Browse(浏览)文件,关于 Browse 文件的选项，可以在 Project -Settings 中设置)。

2.1.2 AppWizard(应用程序向导)

AppWizard 是 Visual C++提供的一个高级编程工具,它可以产生应用的 C++源代码框架。通过与另一个工具 ClassWizard 一起配合使用，可大大节省开发应用程序的时间和精力。

AppWizard 是一个标准的 C++源代码生成器。它通过一系列的对话框来提示用户输入所需创建的程序的信息，如它的名字和位置。用户还可以指定它是否具有一些特性，如多文档接口或工具条，对数据库、OLE 的支持等。然后 AppWizard 生成一些文件，这些文件构成程序的框架。由 AppWizard 生成的程序是一个基本的 Windows 程序，用户可以编译并运行——它实际什么也不做。它只是准备好增加那些为程序提供功能性的资源和代码。这样就节省了用户设计应用程序框架的时间和精力，用户所要做的工作只是直接往框架中添加自己的处理代码。

2.1.3 ClassWizard(类向导)

ClassWizard 是一个交互式工具，用来建立新的类，定制类，把消息映射成类成员函数，或者把控制框映射为类变量成员。在开发程序时，可用 ClassWizard 建立程序所需要的类,包括消息处理和消息映射例程(用于定位处理消息的代码)。

使用 ClassWizard，可以将成员函数或变量加入到一个类中，或修改已经存在的函数和变量。Wizard 使函数或变量放在何处，如何称呼它们以及其他一些细节问题大大简化。

ClassWizard 所能识别的类必须在 ClassView 数据库文件(.CLW)中登记。使用 ClassWizard 可以：

创建新类：从许多框架基类中派生出新类

映射消息到函数

新建、删除消息处理函数

察看已被处理的消息并跳到消息处理代码处

定义成员变量：这些变量会被自动初始化，释放，执行对话框数据检验等

创建新类时，自动加入方法和属性

处理现有的类和类库

在后续章节中,我们将举例说明 AppWizard 和 ClassWizard 这两个工具的用法。

2.1.4 WizardBar(向导工具条)

WizardBar 是一个可停泊的工具条，用于快速访问一些 Developer

Studio 最实用的功能，比如 ClassWizard 或 ClassView 的一些功能。WizardBar 会自动跟踪用户程序的上下文——比如，当文本编辑器中的光标从一个函数移动到另一个函数时，Wizard 的显示会自动更新。

WizardBar 工具条包含了三个相关的下拉列表框：类(Class)、过滤器(Filter)和成员(Member)，如图 2.5 所示。类列表框包含了应用程序定义的所有类。当前所选择的类决定可用的过滤器；所选的过滤器决定 Member 列表中显示的内容。选择 Member 中的一项，可以跳到相应的成员定义。WizardBar 最右边是一个 Action Control，单击 Action Control 的向下箭头符号会弹出一个菜单，用于执行跳到函数定义、增加消息处理函数等操作。

图 2.5 WizardBar

WizardBar 使得处理类、成员和资源更加方便。使用 WizardBar，可以：

增加一个新类

建立一个新的函数或方法

跳到一个已存在的函数或方法

2.1.5 Componet Gallery(组件画廊)

Componet Gallery 是一个组件库，它保存着以后可以共享和重用的代码。这些代码包括由 Visual C++自带的组件和从用户工程中增加到 Gallery 中去的用户自定义组件。

可以使用 Gallery 提供的组件增强用户编写的应用程序的功能。Visual C++提供了一组应用程序的常用组件，分为两类：Developer Studio Components 和 Register ActiveX Controls。其中 Developer Studio Components 包含了我们经常使用的一些标准的 Windows 应用程序特性，比如：弹出菜单，剪贴板，对话条，启动画面(Splash Window)，定制状态条(带时间显示)，日积月累对话框等。Register ActiveX controls 则用于往程序里添加 ActiveX 控件。

增加组件到应用程序中

组件总是加入到活动工程中。要增加组件到工程中，可以从 Project 菜单选取 Add to Project-Components and Controls。将弹出一个对话框，选择相应的组件，回答一系列问题后，Component Gallery 将自动在你的工程文件总加入该组件的 CPP、H 文件以及资源，并自动修改你的源程序，插入必要的代码。无需编程，即可增加一些常用功能，如启动画面。有关使用 Componet Gallery 的例子，可参见第三课有关“弹出菜单”一节。

2.1.6 Developer Studio 的一些快捷特性

为了使开发环境更易于使用，Developer Studio 还提供了一些快捷特性，包括：右键菜单、快速访问常用对话框、属性对话框、键盘快捷键等。

右键菜单(快捷菜单)

在 Developer Studio 的许多窗口中，可以在一个选中的对象或窗口背景上单击鼠标右键弹出快捷菜单。

快捷菜单包含了与当前区域或所选项相关的一组常用命令。大多数命令都可以从菜单条上访问，但用快捷菜单更加方便。

快速访问常用对话框：

在列表出现的窗口中，通常可以通过双击列表中的一项显示与该项相关的常用对话框。比如，要编辑字符串列表中的一项，可以双击它，弹出字符串编辑对话框，用于编辑双击的那一项。

属性对话框

选择一个项，单击右键，从快捷菜单上选择 Properties，就显示出关于该对象的属性对话框。属性对话框显示在一个属性页中，可能包含许多标签。弹出属性对话框的快捷键是 Alt+Enter。点一下对话框左上角的图钉按钮，可以把该属性对话框固定在所有窗口前面。如果有可编辑的属性，就可以在属性页中直接修改。下图是修改某个编辑框属性的一个例子。

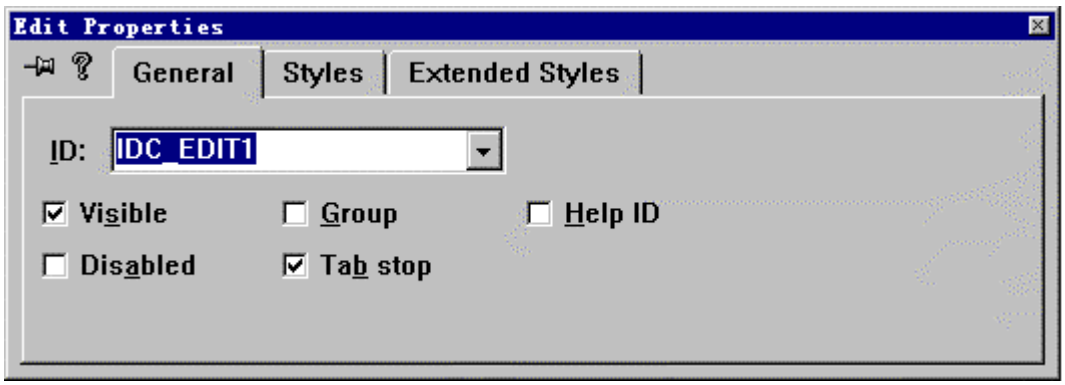


图 2.6　属性对话框

键盘快捷键

键盘快捷键为那些喜欢键盘而不喜欢用鼠标的用户提供了执行某一功能的替代方式。Developer Studio 为一些常用的命令分配了缺省得快捷键。用户可以自定义快捷键。

为了高效的使用 Developer Studio，建议读者记住以下一些常用的快捷键：

打开文件　File Open CTRL+O　打开一个已有的文件
保存文件　File Save CTRL+S　保存活动文档
编辑复制　Edit Copy CTRL+C
编辑剪切　Edit Cut CTRL+T
编辑粘贴　Edit Paste CTRL+V
编辑查找　Edit Find CTRL+F
编辑取消　Edit Undo CTRL+Z
下一条错误　F4
上一条错误　SHIFT+F4
下一个窗口　CTRL+F6
对象属性　ALT+ENTER

建立可执行文件 Build F7

运行 Execute CTRL+F5

建议为 View-Full Screen，Help-Search **增加自定义的快捷键。要自定义快捷键，可以选择** Tools-Customize **菜单来实现。**

上一页/下一页/

本教程由 Visual C++**王朝**(Where programmers come together)**协助制作未经许可，请勿以任何形式复制/**

2.2 创建、组织文件、工程和工作区

项目工作区是一个包含用户的所有相关项目和配置的实体。工程定义为一个配置和一组文件，用以生成最终的程序或二进制文件。一个项目工作区可以包含多个工程，这些工程既可以是同一类型的工程，也可以是由不同类型的工程(如 Visual C++和 Visual J++工程)。工作区现在以.dsw 为后缀名(以前为.mdp)，项目文件现在以.dsp(以前为.mak)为后缀名。

Developer Studio 提供了一个简单的对话框，用以创建项目工作区、工程、文件和其他文档。在创建工程时，可以指定一个新的项目工作区来包含它；或把新工程加入到当前打开的工作区中。在创建一个文件时，可以同时把它加入到工程中，也可以将现存的文件、文档加入到工程中。

2.2.1 新建工程

在新建一个工程时，可以把它加入到一个已有的项目工作区中，或同时创建一个新的项目工作区。如果加工程到一个已有的工作区中，则可以将该工程设为已有工程的子工程。

提示：Developer Studio 以工程名字来区分工程，所以要求每一个新建的工程必需有一个独一无二的名字，这样才能确保 Developer Studio 的工作区可以包含位于不同位置的工程。

要新建一个工程，可以：

1.在 File 菜单上，点 New，选择 Projects 标签。

2.从列表中选择项目类型

3. 点 Create New Workspace(新 建 工 作 区) 或 Add to Current Workspace(加入到当前工作区中)

4.要使新工程为子工程，可以选择 Dependency of 检查框，并从列表中选择一个工程。

5.在 Project Name 框中，输入新工程名，确保该名字必须与工作区中的别的工程名字不重名。

6.在 Location 框中，指定工程存放的目录：可以直接输入路径名，也可以按旁边的 Browse 按钮，浏览选择一个路径。

7.点 Platform 框中的相应检查框，指定工程的开发平台

输入完以上内容并按 OK 按钮后，根据所选的工程类型，会出现相应的 Wizard(向导)。通过一系列的对话框输入，快速生成工程的框架。

2.2.2 新建工作区

可以在新建工程的同时指定创建一个新的工作区，工作区文件名同该工程，后缀名为.dsw。也可以创建一个空的(不含任何工程的)工作区。

要创建一个空的工作区，可以：

在 File 菜单上，点 New 在随后弹出的对话框上，点 Workspace 标签从类型列表中选择 Blank Workspace 在 Workspace Name 框中输入名字，注意名字不能通它将要包含的工程同名在 Location 框中指定存放工作区文件的目录按 OK。

2.2.3 增加已有文件到工程中

打开包含目标工程的项目工作区文件

在 Project **菜单上，点** Add to Project**，然后点** Files

在 Insert Files into Project **对话框中，浏览并定位要加入到工程中的文件名，然后选择它们。**

从 Insert Into **中选择工程名字，然后点** OK**。**

如果工作区已经打开，而且要加入的文件也已打开，那么只要在该文件的编辑器中点鼠标右键，从快捷菜单上选择 Add to Project**，就可以把该文件加入到当前活动工程中。**

2.2.4 **打开工作区**

选择 File- Open Workspace**,指定要打开的工作区 或选择** File -Recent Workspaces **，从最近打开过的工作区列表中选择一个。**

2.2.5 **设置当前工程**

选择 Project Setting**，可以为当前工程设置编译、链接、**C/C++**等各种选项。**

2.3 WIN32 开发

Visual C++5.0 是一个全 32 位的软件开发工具，它完全支持 32 位的 Win32 平台开发。Win32 平台包括 32 位的 Windows 操作系统和软件开发系统 Win32 API。所谓 API(应用程序接口)指的是一组由操作系统提供的函数。Win32 API 是 Windows 平台上的一个 32 位的软件开发系统，它使应用程序可以充分利用 32 位 Windows 操作系统的能力。使用 Win32 API 写成的应用程序可以在 Windows95 或更高版本以及 Windows NT 上运行。

由于 Microsoft 在 Windows 3.x 及其 Win16 API 上取得巨大的成功，因此，在研制 Win32 API 时，首先考虑的就是保证 Win32 与 Win16 API 兼容，只有让软件开发者能将 Win16 代码很容易移植到 Win32 API 上，才有实际意义。Win32 API 在语法上只作了极小的改动，API 的命名与 Windows 的 Win16 API 相同，语义也相同，消息序号也相同。事实上，完全可以保存独立的源代码，并选择编译成 16 位的 Win16 程序或 32 位的 Win32 程序。

其次，如其名所示，在设计 Win32 API 时考虑到了充分利用 32 位处理器的能力。随着硬件的发展，内存和 CPU 价格的降低和性能的提高，32 位 CPU 的 486、Pentium 已成为主流。据有关数据显示，目前在我国家用计算机用户中，使用 Pentium 系列处理器的计算机已占 80%以上。如何充分利用当前 32 位(和 64 位)处理器的能力，并预见将来处理器的发展，就成为 Win32 设计时考虑的重要因素之一。

再次，为了摆脱操作系统对 Intel 处理器的依赖，使应用程序可以运行于各种处理器平台上，Win32 设计时增强了它的可移植性，提供了 Microsoft Windows95 和 Windows NT 之间的透明的移植能力。虽然 Windows95 只能运行于 Intel 平台上，但是 Win32 还支持 Windows NT，而 Windows NT 已经被移植到许多非 Intel 的处理器上，如 Alpha、RISC 硬件平台等。

Win32 可以应用于特定的操作系统，这种系统可以直接控制和处理 PC 硬件资源，而不必象 Win16 API 那样依赖于 MS-DOS 系统服务。然而，Win32 不是简单的由 Win16 从 16 位到 32 位的升级，更重要的在于它支持：高性能的抢先式多任务和多线程连续的 32 位地址空间和先进的内存管理对所有的可为进程共享的对象，解决了它的安全性问题内存映射文件

2.3.1 抢先式多任务和多线程

我们知道 Windows 是一个多任务操作系统，它提供了一次运行多个应用程序的能力。但是，Windows 3.x 和 Windows95 在多任务的实现上有所不同。

Windows 3.x 的多任务是一种由协作、软件方式产生的有限的非抢先式的多任务。它是借助于每个应用程序的消息循环这种软件协议方式来实现多任务的。Windows 3.x 管理所有的消息，并存放于系统的消息队列中。操作系统判断消息应归哪一个窗口去处理，再将消息发送给该窗口。每个应用程序窗口处于等待消息状态，直到有消息来，然后进行处理，

处理完毕将控制权交给操作系统。在对消息进行处理时，对于用户用键盘或者鼠标输入的任何命令，Win16 都不会理睬。比如，我们用 WORD 载入一个文件时，其他程序都得等待文件 I/O 操作完成才能获得响应。而且，一个应用程序切换到另一个应用程序时，需要较长的等待时间。各应用程序在取得消息、处理消息时是平等的，无优先级的，系统无法设置应用程序的优先级和时间片的大小。

Windows95 的多任务是一种抢先式多任务。比如，我们在用资源管理器复制一个文件的同时，还可以启动另外一个应用程序，如纸牌游戏，而且随时都可以切换回资源管理器，察看文件复制进度，系统始终保持较好的响应和灵活性。Windows95 的抢先式多任务机制不是用 Windows 3.x 下的软件调度来实现的。要了解抢先式多任务，我们需要首先了解一下进程和线程的概念。调入内存准备执行的应用程序叫做进程(process)。每个进程至少有一条线程，叫做主线程(primary thread)。一个进程包含代码、数据和其他属于应用程序的资源。一条线程包含一组指令，相关的 CPU 寄存器值和一个堆栈。

在抢先式多任务操作系统中，系统在所有运行的所有进程之间对 CPU 时间进行共享，从而保证每个进程都能频繁的访问处理器，并且实现指令的连续执行。这样，每个 Win32 进程都需要分配一个优先级，系统调度程序利用这种优先级来决定哪一时刻该运行哪一个进程。具有高优先级的进程(严格的说应当是线程)就是当前运行的哪一个。更高优先级的线程可以中断当前进程的执行。同一优先级的线程通过时间片来调度。一个线程处于以下三种状态之一：正在执行，挂起，准备运行。在单处理器环境下(如 Windows 95)，同一时刻只能运行一个线程。有关多线程，我们还将在后面的章节里作专门介绍。

为了在 Win32 中支持多线程进程结构，Win32 在原来 Win16 基础上增加了：

对进程以及线程创建、操纵的支持

对一个进程内线程之间的同步和同步对象的支持

一个统一的共享机制。

2.3.2 连续的地址空间和先进的内存管理

对于各种操作系统和平台来说，内存管理都是一个非常重要的问题。在 Windows3.1 下，有两种形式的内存管理函数调用：局部的和全局的。全局内存管理函数从物理内存中分配一段，然后返回一个句柄值。该句柄可以转换为一个 GlobalLock 函数所使用的远指针。基本处理过程如下：申请一块可移动的内存块锁定该内存块。因为 Windows 引入了虚拟内存管理，可以把内存块移动到硬盘交换文件中，所以在使用内存块之前，必须将它锁定在真正的内存 RAM 之中，也就是告诉操作系统，现在这块内存暂时由应用程序来管理。

对该内存块进行各种操作：如复制数据到内存块。

解锁内存，应用程序将对该内存的控制交与 Windows。

下面给出一个程序片段，来说明内存管理函数的用法。

HGLOBAL memHandle;//内存句柄

```
char far* lpMem;//假设长度为 memLen
memHandle=GlobalAlloc(GHND,memLen+1);//申请内存块，此处未做返回结果检查，
//事实上，申请内存有时会失败
memcpy(lpMem,string,textLen);//拷贝数据，其中 string 为一字符串变量，textLen 是这个
//字符串的长度
GlobalUnlock(memHandle);//解锁内存
...
GlobalFree((HGLOBAL) memHandle);//释放内存
```

全局内存对所有的应用程序都是可见的，不管是显式的还是隐式的请求。因为 Windows 3.x 的实现方式就是所有的进程在同一地址空间中运行。局部内存管理则是从 64KB 的段内分配对象并返回所分配内存的 16 位偏移量。

在 Win32 下，局部和全局内存管理函数基本相同，仍然可以使用可移动和可丢弃选项。但是它引入了连续(flat)的 32 位内存管理概念。

在 Win32 中，每个进程都有其特有的 32 位虚拟地址空间，该空间最大可达 4GB。如图所示，低端内存的 2GB 是用户可用的，高端内存的 2GB 为内核(Kernel)保留。其中，最高的 1GB 用于 VxD、内存管理和文件系统。下面的 1GB 用于共享的 Win32 DLL、内存映射文件和共享内存区域。进程所使用的虚拟地址不代表一个对象在内存的实际的物理位置(事实上，我们大部分的 PC 还没有配置 4GB 内存)。操作系统为每个进程维护一个映射表，根据该表将虚拟地址映射到真正的物理位置处(RAM 或者交换页文件中)。

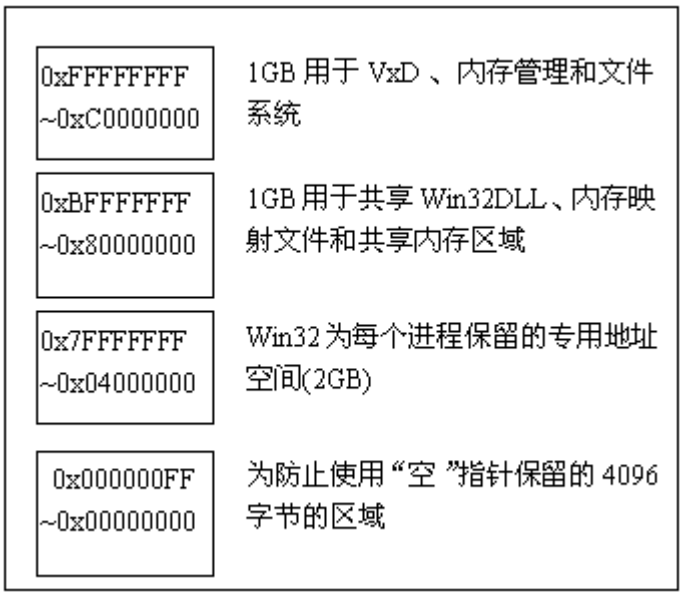


图 2.7 Windows95 的内存映射

在 Win32 下局部内存对象有一个 32 位句柄而不是 Windows 3.x 下的 16 位句柄，而且这个句柄是一个实际指针而不是一个相对于段的偏移量。

Win32 和 16 位 Windows 一个重要区别是：在 Win32 下，所有的进程都有自己独立的地址空间(在进程内部的线程仍然共享进程的内存变量)，全局内存不再对所有的 Windows 应用程序都可见。由于每个应用程序都有自己的地址空间，一个进程分配的内存在该进程的地址之外就不再可见。DDE 会话中使用的内存对接收者进程来说是透明的。这样，进程的安全性就得到大大提高，程序更加强壮。一个进程崩溃一般不会影响另外一个进程的执行。但是，这也给多个应用程序共享内存带来了困难。在许多情况下，需要在多个应用程序之间进行通讯和数据交换，这时，该怎么办呢"Win32 引入了内存映射文件，很好的解决了这个问题。

2.3.3 内存映射文件

内存映射文件是由一个文件到一块内存的映射。Win32 提供了允许应用程序把文件映射到一个进程的函数(CreateFileMapping)。这样，文件内的数据就可以用内存读/写指令来访问，而不是用 ReadFile 和 WriteFile 这样的 I/O 系统函数，从而提高了文件存取速度。

这种函数最适用于需要读取文件并且对文件内包含的信息做语法分析的应用程序，如对输入文件进行语法分析的彩色语法编辑器，编译器等。把文件映射后进行读和分析，能让应用程序使用内存操作来操纵文件，而不必在文件里来回地读、写、移动文件指针。

有些操作，如放弃“读”一个字符，在以前是相当复杂的，用户需要处理缓冲区的刷新问题。在引入了映射文件之后，就简单的多了。应用程序要做的只是使指针减少一个值。

映射文件的另一个重要应用就是用来支持永久命名的共享内存。要在两个应用程序之间共享内存，可以在一个应用程序中创建一个文件并映射之，然后另一个应用程序可以通过打开和映射此文件把它作为共享的内存来使用。

2.3.4 Win32s：Windows 3.x 对 Win32 API 的支持

我们经常会遇到 Win32s 这个词，它与 Win32 是有区别的。Win32s 的 s 的含义是指子集(subset)。它指的是，在一个 Win32 程序中移入一些 DLLs 和一个 VxD，使它运行于配置 80386 以上处理器的 Windows 3.x 系统之上，并且以一种增强模式运行(但有一定限制)。运行在 Windows 3.x/Win32s 系统上的 Win32 程序支持 32 位指针和 32 位寄存器，只需要在系统调用之前稍作形式替换。如果程序中使用大的数据结构或很多的计算时，Win32s 性能明显优于 16 位 Windows 版本，根据 Microsoft 的测试，性能可以提高两倍左右；如果程序只是大量的调用 Windows API，则 16 位版本的性能可能会强于 32 位版本，因为 Win32s 会对每一次 API 调用作一个从 16 位到 32 位的转换。

Win32s 子集同 Win32 相比，不支持：多线程，高级图形 API，异步文件 I/O，Unicode 和安全性；而且它是运行于 16 位的 Windows 系统上的。但是同 Win16 相比，有它的优越之处，目前在 16 位 Windows 程序开发方面有相当的潜力。

Visual C++4.1 及以前版本支持 Win32s，但 Visual C++5.0 不再支持 Win32s。

2.3.5 Win32 编程基础

Win32 数据类型

这里的数据类型指的是一些关键字，这些关键字定义了 Win32 中的函数中的有关参数和返回值的大小和意义。Win32 常用的数据类型有：

数据类型/描述/

HANDLE/定义一个 32 位无符号的整数，用作句柄/

HINSTANCE/定义一个 32 位的无符号整数，用作实例句柄/

HWND/定义一个 32 位的无符号整数，用作窗口句柄/

HDC/一个设备描述背景的句柄/

LONG/说明一个 32 位带符号整数/

LPSTR/定义一个线性的 32 位字符串指针/

UINT/定义一个新的 Win32 数据类型，它会把一个参数强制转换成 Windows3.x 应用中的 16 位值或 Win32 应用中的 32 位/

WCHAR/说明一个 16 位的 UNICODE 字符，用来表示世界上所有已知的书写语言的符号/

这里需要解释一下的是句柄。句柄是 Windows 编程的一个关键性的概念，编写 Windows 应用程序总是要和各种句柄打交道。所谓句柄，就是一个唯一的数，用以标识许多不同的对象类型，如窗口、菜单、内存、画笔、画刷、电话线路等。在 Win32 里，句柄是指向一个“无类型对象”(void*)的指针，也就是一个 4 字节长的数据。无论它的本质是什么，句柄并不是一个真正意义上的指针。从构造上看，句柄是一个指针，尽管它没有指向用于存储某个对象的内存位置。事实上，句柄指向一个包含了对该对象进行的引用的位置。句柄的声明是这样的：

typedef void *HANDLE

由于 Windows 是一个多任务操作系统，它可以同时运行多个程序或一个程序的多个副本。这些运行的程序称为一个实例。为了对同一程序的多个副本进行管理，Windows 引入了实例句柄。Windows 为每个应用程序建立一张表，实例句柄就好象是这张表的一个索引。

Windows 不仅使用句柄来管理实例，也用它来管理窗口、位图、字体、元文件、图标等系统资源。

标识符命名

在编程时，变量、函数的命名是一个极其重要的问题。好的命名方法使变量易于记忆且程序可读性大大提高。Microsoft 采用匈牙利命名法来命名 Windows API 函数和变量。匈牙利命名法是由 Microsoft 的著名开发人员、Excel 的主要设计者查尔斯·西蒙尼在他的博士论文中提出来的，由于西蒙尼的国籍是匈牙利，所以这种命名法叫匈牙利命名法。

匈牙利命名法为 C 标识符的命名定义了一种非常标准化的方式，这种命名方式是以两条规则为基础的：

1.标识符的名字以一个或者多个小写字母开头，用这些字母来指定数据类型。下表列出了常用的数据类型的标准前缀：

在 Windows 里定义数据类型的一些标准前缀

前缀/数据类型/

c/字符(char)/
s/短整数(short)/
cb/用于定义对象(一般为一个结构)尺寸的整数/
n/整数(integer)/
sz/以’\0’结尾的字符串/
b/字节/
i/int(整数)/
x/短整数(坐标 x)/
y/短整数(坐标 y)/
f/BOOL/
w/字(WORD，无符号短整数)/
l/长整数(long)/
h/HANDLE(无符号 int)/
m_/类成员变量/
fn/函数(function)/
dw/双字(DWORD，无符号长整数)/

2.在标识符内，前缀以后就是一个或者多个第一个字母大写的单词，这些单词清楚地指出了源代码内那个对象的用途。比如，m_szStudentName 表示一个学生名字的类成员变量，数据类型是字符串型。

从 16 位的 Win16 API 迁移到 Win32 API 注意点

1.数据类型字长的变化：

我们编写一个小程序来说明 Win32 下的常见数据类型的字长：

```
#include<windows.h>
#include<stdio.h>
void main(void)

{
printf(“sizeof(int) is %d\n”,sizeof(int));
printf(“sizeof(BYTE) is %d\n”,sizeof(BYTE));
printf(“sizeof(WORD) is %d\n”,sizeof(WORD));
printf(“sizeof(DWORD) is %d\n”,sizeof(DWORD));
printf(“sizeof(LONG) is %d\n”,sizeof(LONG));
printf(“sizeof(PVOID) is %d\n”,sizeof(PVOID));
printf(“sizeof(LPVOID) is %d\n”,sizeof(LPVOID));
}
```

使用 Visual C++编译运行该程序，输出结果如下：

```
sizeof(int) is 4
sizeof(BYTE) is 1
sizeof(WORD) is 2
sizeof(DWORD) is 4
sizeof(LONG) is 4
```

sizeof(PVOID) is 4

sizeof(LPVOID) is 4

从上面的输出结果我们看到：整数类型字长已经同长整数相同，PVOID **近指针和** LPVOID **远指针长度也相同。在编程过程中，我们要注意这些变化，凡是设计字长的问题最好还是采用可以移植的** sizeof **操作符来做。**

2.**内存模式变化：**

在 Win32 **平台下，不再有微模式、紧凑模式、中模式、大模式、巨模式、自定义内存模式之分，也不再有** 64KB **代码段和数据段的限制。只有一种内存模式，**Win32 **下的地址和代码均在线性寻址的** 2GB **的** 32 **位内存空间中。当然，编程时还是要考虑到实际内存限制的。**

3.**类型修饰符：**

在 Win32 **下，不再有远指针、近指针、巨型指针之分，三种指针类型完全相同。**32 **位的编译器会忽略所有的**_near、_far、_huge **关键字并一视同仁来处理。在** Win32 **中，象** LPSTR **和** PSTR **这种类型是等价的。**

4.**函数的变化：**

Win32API **设计时尽可能保证与** Win16API **兼容，但是仍然对一些函数作了修改。比如在** Win16 **下的** MoveTo **在** Win32 **下为** MoveToEx**。如果在编译程序时某个** API **函数没找到，试着在这个函数名后面加上** Ex**，**Ex **表示它是** Win16 **的扩展。**

2.4 MFC 编程

微软基础类库(MFC：Microsoft Foundation Class)是微软为 Windows 程序员提供的一个面向对象的 Windows 编程接口，它大大简化了 Windows 编程工作。使用 MFC 类库的好处是：首先，MFC 提供了一个标准化的结构，这样开发人员不必从头设计创建和管理一个标准 Windows 应用程序所需的程序，而是“站在巨人肩膀上”，从一个比较高的起点编程，故节省了大量的时间；其次，它提供了大量的代码，指导用户编程时实现某些技术和功能。MFC 库充分利用了 Microsoft 开发人员多年开发 Windows 程序的经验，并可以将这些经验融入到你自己开发的应用程序中去。

对用户来说，用 MFC 开发的最终应用程序具有标准的、熟悉的 Windows 界面，这样的应用程序易学易用；另外，新的应用程序还能立即支持所有标准 Windows 特性，而且是用普通的、明确定义的形式。事实上，也就是在 Windows 应用程序界面基础上定义了一种新的标准——MFC 标准。

为了更好的理解 MFC，我们有必要了解一下 MFC 的历史。

2.4.1 MFC 历史

开始，Microsoft 建立了一个 AFX 小组，AFX 代表 Application Framework，即应用程序框架。据说创建该小组原意是为了发布一个 Borland C++的 OWL 的竞争性产品，因为那时侯 Borland 公司的应用程序框架 OWL(object Windows Language)已经做的相当成功。AFX 小组象 OWL 那样，提出了一个高度抽象 Windows API 的一个类库。

他们采用自顶向下的设计方法，逐步将对象抽象出来，并施加到 Windows 上。然后，他们试着花了几个月时间用这个类库来编写应用程序，结果发现这个类库偏离 Windows API 实在太远，过分抽象并没有太大的实用性，相反大大降低了应用程序的效率。

于是，他们干脆放弃了整个 AFX 类库，对类库进行重新设计。这次，他们采用了自底向上的方法，从已有的 Windows API 着手，将类建立在 Windows API 对象基础上，设计出后来成为 MFC1.0 的一个类库。但是，你现在仍然可以看到 AFX 时期的痕迹，许多源程序文件有 afx 前缀，如 afxabort.cpp,afxmem.cpp。MFC 延用了许多 AFX 类库的宏，因此我们经常会看到以 AFX 开头的宏。

AFX 小组实际上做了两件工作：MFC 类库和对 MFC 的 IDE 支持(即资源编译器和操作向导)。在 1994 年 4 月份之后，AFX 的名字停止使用，该小组成员成为 Visual C++开发组的一部分，即现在的 MFC 小组。

MFC1.0 版于 1992 年同 Microsoft C/C++7.0 同时发布。它提供了对 Windows API 简单的抽象和封装，还没有我们现在常用的文档/视结构特性。但它引入了 CObject，通过 CArchive 的持续化和其他一些 MFC 中仍然使用的特性，从而奠定了 MFC 的基础。

MFC2.0 在 MFC1.0 基础上增加了文档/视结构框架、OLE1.0 类、消息映射和公用对话框类，废弃了 1.0 版中的 CModalDialog 类并将它的功

能移入到 CDialog 中，并增加了工具条、对话条、分割视窗的支持。MFC2.1 随同 Visual C++ 1.1 for NT 发布，它把 MFC2.0 移植到了 Win32 上。MFC2.5 随同 Visual C++1.5 一起发布，它引入了 OLE 2.0 和 ODBC 类。它是最后的官方的 16 位发行版，于 93 年 12 月发布。目前，在开发 16 位 Windows 程序时，Visual C++1.5 和 MFC 2.5 仍然有大量的用户。随后的 MFC2.51、2.52 纠正了 MFC.25 中的一些错误，增加了标签式对话框、WinSock 和 MAPI(Microsoft 电子邮件应用程序接口)支持。MFC3.1 同 Visual C++2.1 一起于 1995 年 1 月份发布，它引入了 Windows95 公共控件(包括动画、热键、图象列表、工具条提示等等)。MFC4.0 于 1995 年 12 月份同 Visual C++4.0 一起发布。Microsoft 直接从 Visual C++2.0 一下子跳过一个版本号，升级到了 4.0，以保持 MFC 版本号和 Visual C++版本号的一致性，但这种一致性又在 Visual C++5.0 中打破了。在 MFC4.0 中增加了 CSynchronize,CMutex,CEvent,CMultiLock,CShellNew 以更好的支持多线程以及 Windows 95 的其他一些特性。Visual C++还引入了 Component Gallery(组件画廊)、STL 支持和大量的新特性。MFC4.1 最重要的特性是支持 Win32s。许多 MFC 开发者一直都在使用该版本。MFC4.1 修正了 4.0 的一些错误并增加了 Internet 特性。MFC4.2 增加了 ISAPI 和 OCX 容器支持。

MFC4.21 于 1997 年 3 月 19 日同 Visual C++5.0 一起发布，它是目前最新和最完善的 MFC 版本。它只增加了对微软的 IntelliMouse(智能鼠标器)的支持。现在 MFC 版本号又不与 Visual C++匹配了。

MFC 发行版列表如下：

MFC Release MSVC Release 16 位或 32 位 备注

1.0 16 简单的 封装 Windows

2.0 1.0 16 增加了文档/视结构

2.1 1.1 for NT 32 第一个 NT 的发行版

2.5 1.5 16 OLE/ODBC,最后一个

16 位版本

2.51 2.0 16 修正错误

2.52 2.1 16 增加标签式对话框

2.52b 2.2 16

2.5c 4.0 16

3.0 2.0 32 标签式对话框、可停泊工具条

3.1 2.1 32 Winsock/MAPI, Windows 公共控 制

3.2 2.2 32

4.0 4.0 32 Win 95, 线程类, OCX 容器

4.1 4.1 32 sweeper (WinInet) classes

**以上是最后支持 Win32s 的版本

4.2 4.2 32 修正错误, ISAPI classes

4.2b internet dl 32 修正错误

4.21 5.0 32 IntelliMouse™ support.

2.4.2 MFC 类库概念和组成

类库是一个可以在应用中使用的相互关联的 C++类的集合。类库有些随编译器提供，如 Borland C++ Turbo Vision 等；有的是由其他软件公司销售，如用于数据库开发的 CodeBase；有的则是由用户自己开发的。比如图象处理类库完成图象显示、格式转换、量化等；串行通信类库用于支持串行口输入输出。有些情况下用户可以直接利用类库中包含的类定义应用程序所需的变量，有时则需要从类库所提供的类中派生出新的类，这依赖于类库的设计和具体的应用程序。

Microsoft 提供了一个基础类库 MFC，其中包含用来开发 C++和 C++ Windows 应用程序的一组类。基础类库的核心是以 C++形式封装了大部分的 Windows API。类库表示窗口、对话框、设备上下文、公共 GDI 对象如画笔、调色板、控制框和其他标准的 Windows 部件。这些类提供了一个面向 Windows 中结构的简单的 C++成员函数的接口。

MFC 可分为两个主要部分：(1)基础类(2)宏和全程函数。

MFC 基础类

MFC 中的类按功能来分可划分为以下几类：

基类

应用程序框架类

应用程序类

命令相关类

文档/视类

线程类

可视对象类

窗口类

视类

对话框类

属性表

控制类

菜单类

设备描述表

绘画对象类

通用类

文件

诊断

异常

收集

模板收集

其他支持类

OLE2 类

OLE 基类

OLE 可视编辑包装程序类

OLE 可视编辑服务器程序类

OLE 数据传输类

OLE 对话框类
其他 OLE 类
数据库类
宏和全局函数
若某个函数或变量不是某个类的一个成员，那么它是一个全程函数或变量。
Microsoft 基本宏和全程函数提供以下功能：
数据类型
运行时刻对象类型服务
诊断服务
异常处理
CString 格式化及信息框显示
消息映射
应用消息和管理
对象连接和嵌入(OLE)服务
标准命令和 Windows IDs
约定：全程函数以“ Afx ”为前缀，所有全程变量都是以“ afx ”为前缀，宏不带任何特别前缀，但是全部大写。
常见的全局函数和宏有： AfxGetApp， AfxGetMainWnd， AfxMessageBox，DEBUG_NEW 等，我们会在后面的章节中用到并对它们进行介绍。
从继承关系来看，又可将 MFC 中的类分成两大类：大多数的 MFC 类是从 CObject 继承下来；另外一些类则不是从 CObject 类继承下来，这些类包括：字符串类 CString，日期时间类 CTime，矩形类 CRect，点 CPoint 等，它们提供程序辅助功能。
由于 MFC 中大部分类是从 CObject 继承下来的，CObject 类描述了几乎所有的 MFC 中其他类的一些公共特性，因此我们有必要理解 CObject 类。
我们首先查看一下 CObject 类的定义，CObject 类定义如下清单 2.1 所示：
清单 2.1CObject 类的定义

```
// class CObject is the root of all compliant objects
class CObject
{
public:
// Object model (types, destruction, allocation)
virtual CRuntimeClass* GetRuntimeClass() const;
virtual ~CObject(); // virtual destructors are necessary
// Diagnostic allocations
void* PASCAL operator new(size_t nSize);
void* PASCAL operator new(size_t, void* p);
void PASCAL operator delete(void* p);
```

```
#if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
// for file name/line number tracking using DEBUG_NEW
void* PASCAL operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#endif
// Disable the copy constructor and assignment by default so you will get
// compiler errors instead of unexpected behaviour if you pass objects
// by value or assign objects.
protected:
CObject();
private:
CObject(const CObject& objectSrc); // no implementation
void operator=(const CObject& objectSrc); // no implementation
// Attributes
public:
BOOL IsSerializable() const;
BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Overridables
virtual void Serialize(CArchive& ar);
// Diagnostic Support
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
// Implementation
public:
static const AFX_DATA CRuntimeClass classCObject;
#ifdef _AFXDLL
static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};
```

CObject **类为派生类提供了下述服务：**

对象诊断

MFC **提供了许多诊断特性，它可以：**

输出对象内部信息：CDumpContext **类与** CObject **的成员函数** Dump **配合，用于在调试程序时输出对象内部数据。**

对象有效性检查：重载基类的 AssertValid **成员函数，可以为派生类的对象提供有效性检查。**

运行时访问类的信息：

MFC **提供了一个非常有用的特性，它可以进行运行时的类型检查。如果从** CObject **派生出一个类，并使用了以下三个宏**(IMPLEMENT_DYNAMIC **，** IMPLEMENT_ DYNCREATE **或** IMPLEMENT_SERIAL)**之一，就可以：**

运行时访问类名

安全可靠的把通用的 CObject **指针转化为派生类的指针**

比如，我们定义一个主窗口类

```
CMyFrame:public CFrameWnd
{
.....
}
```

然后我们使用这个类：

```
CMyFrame *pFrame=(CMyFrame*)AfxGetMainWnd();
pFrame->DoSomeOperation();
```

AfxGetMainWnd **是一个全局函数，返回指向应用程序的主窗口的指针，类型为** CWnd*，**因此我们必须对它进行强制类型转换，但我们如何知道是否转换成功了呢”我们可以使用** CObject **的** IsKindOf()**成员函数检查** pFrame **的类型，用法如下：**

```
ASSERT(pFrame->IsKindOf(RUN_TIMECLASS(CMyFrame)));
```

；

将上一语句插入到 pFrame-> DoSomeOperation()**之前，就可以在运行时作类型检查，当类型检查失败时，引发一个断言**(ASSERT)**，中断程序执行。**

对象持续性

通过与非 CObject **派生的档案类** CArchive **相结合，提供将多个不同对象以二进制形式保存到磁盘文件**(Serilization)**中以及根据磁盘文件中的对象状态数据在内存中重建对象**(Deserilization)**的功能。**

然而，MFC **不仅仅是一个类库，它还提供了一层建立在** Windows API **的** C++**封装上的附加应用程序框架。该框架提供了** Windows **程序需要的多数公共用户界面。**

所谓应用程序框架指的是为了生成一般的应用所必须的各种软组件的集成。应用框架是类库的一种超集。一般的类库只是一种可以用来嵌入任何程序中的、提供某些特定功能(如图象处理、串行通信)**的孤立的类的集合，但应用框架却定义了应用程序的结构，它的类既相互独立，又相互依赖，形成一个统一的整体，可以用来构造大多数应用程序。中国用户熟悉的** Borland C++**的** DOS **下的** Turbo Vision **和** Windows **下** OWL(Object Windows Language)**都是应用框架的例子。**

下面我们举个具体的例子来说明 MFC **所提供的应用程序框架，程序如清单** 2.2**。**

清单 2.2 **应用程序框架示例**

```
#include<afxwin.h>
//derived an application class
class CMinMFCApp:public CWinApp
{
public:
BOOL InitInstance();
};
//Derive the main window class
```

```
class CMainWindow:public CFrameWnd
{
public:
CMainWindow();
DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
END_MESSAGE_MAP()
/*CMinMFCApp Member Functions*/
BOOL CMinMFCApp::InitInstance()
{
m_pMainWnd=new CMainWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
/*CMainWindow member functions*/
CMainWindow::CMainWindow()//constructor
{
Create(NULL,
“Min MFC Application”,
WS_OVERLAPPEDWINDOW,
rectDefault,
NULL,
NULL);
}
/*an instance of type CMinMFCApp*/
CMinMFCApp ThisApp;
```

清单 2.2 **程序段定义了一个最小的** MFC **应用程序所需的框架程序。其中声明了** CMinMFCApp **类，它是从应用程序类** CWinApp **中派生下来的；和窗口** CMainWindow **类，它是从框架窗口** CFrameWnd **类派生出来。我们还用** CMinMFCApp **定义了一个全局对象** ThisApp**。读者也许会问，为什么没有** WinMain **函数”因为** MFC **已经把它封装起来了。在程序运行时，** MFC **应用程序首先调用由框架提供的标准的** WinMain **函数。在** WinMain **函数中，首先初始化由** CMinMFCApp **定义的唯一的实例，然后调用** CMinMFCApp **继承** CWinApp **的** Run **成员函数，进入消息循环。退出时调用** CWinApp **的** ExitInstance **函数。**

由上面的说明可以看到，应用程序框架不仅提供了构建应用程序所需要的类(CWinApp，CFrameWnd **等)，还定义了程序的基本执行结构。所有的应用程序都在这个基本结构基础上完成不同的功能。**

MFC **除了定义程序执行结构之外，还定义了三种基本的主窗口模型：单文档窗口，多文档窗口和对话框作为主窗口。**

Visual C++提供了两个重要的工具，用于支持应用程序框架，它们就是前面提到 AppWizard 和 ClassWizard。AppWizard 用于在应用程序框架基础上迅速生成用户的应用程序基本结构。ClassWizard 用于维护这种应用程序结构。

2.4.3 MFC 的优点

Microsoft MFC 具有以下不同于其它类库的优势：

完全支持 Windows 所有的函数、控件、消息、GDI 基本图形函数，菜单及对话框。类的设计以及同 API 函数的结合相当合理。

使用与传统的 Windows API 同样的命名规则，即匈牙利命名法。

进行消息处理时，不使用易产生错误的 switch/case 语句，所有消息映射到类的成员函数，这种直接消息到方法的映射对所有的消息都适用。它通过宏来实现消息到成员函数的映射，而且这些函数不必是虚拟的成员函数，这样不需要为消息映射函数生成一个很大的虚拟函数表(V 表)，节省内存。

通过发送有关对象信息到文件的能力提供更好的判定支持，也可确认成员变量。

支持异常错误的处理，减少了程序出错的机会

运行时确定数据对象的类型。这允许实例化时动态操作各域

有较少的代码和较快的速度。MFC 库只增加了少于 40k 的目标代码，效率只比传统的 C Windows 程序低 5%。

可以利用与 MFC 紧密结合的 AppWizard 和 ClassWizard 等工具快速开发出功能强大的应用程序。

另外，在使用 MFC 时还允许混合使用传统的函数调用。

我们着重讲解一下 MFC 对消息的管理，这是编写 MFC 消息处理程序的基础。

2.4.4 MFC 对消息的管理

Windows 消息的管理包括消息发送和处理。为了支持消息发送机制，MFC 提供了三个函数：SendMessage、PostMessage 和 SendDlgItemMessage。而消息处理则相对来说显得复杂一些。MFC 采用了一种新的机制取代 C 语言编程时对 Windows 消息的 Switch/Case 分支，简化了 Windows 编程，使程序可读性、可维护性大大提高。

MFC 对消息的处理

MFC 不使用用 C 语言编写 Windows 程序时用的易产生错误的 switch/case 语句，而采用一种消息映射机制来决定如何处理特定的消息。这种消息映射机制包括一组宏，用于标识消息处理函数、映射类成员函数和对应的消息等。其中，用 afx_msg 放在函数返回类型前面，用以标记它是一个消息处理成员函数。类若至少包含了一个消息处理函数，那么还需要加上一个 DECLARE_MESSAGE_MAP()宏，该宏对程序执行部分所定义的消息映射进行初始化。清单 2.3 演示了消息处理函数的例子：

清单 2.3 消息处理函数例子

```
class CMainFrame:CFrameWnd{
public:
```

```
CMainFrame();
protected:
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnEditCopy();
afx_msg void OnClose();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

成员函数 OnCreate,OnEditCopy,OnClose **分别用来处理消息** WM_CREATE、ID_EDIT_COPY **和** WM_CLOSE**。其中，**WM_CREATE **和** WM_CLOSE **是系统预定义消息，包含在** Windows.h **中。而** ID_EDIT_COPY **是菜单** Edit->Copy **的标识，也就是用户选择** Edit->Copy **菜单项时产生的消息，一般在资源文件头文件中定义。在类的实现部分给出这三个成员函数的定义，以及特殊的消息映射宏。上面的例子的消息映射宏定义如下：**

```
BEGIN_MESSAGE_MAP(CMainFrame,CFrameWnd)
ON_WM_CREATE()
ON_COMMAND(ID_EDIT_COPY, OnEditCopy)
ON_WM_CLOSE()
END_MESSAGE_MAP()
```

消息映射宏由 BEGIN_MESSAGE_MAP() **和** END_MESSAGE_MAP()**。其中，**BEGIN_MESSAGE_MAP **宏包含两个参数** CMainFrame **类和** CFrameWnd**，分别代表当前定义的类和它的父类。在** BEGIN_MESSAGE_MAP()**和** END_MESSAGE_MAP()**之间，包含了主窗口要处理的各个** Windows **消息的入口。在本例中，包含三个消息。其中** ON_ WM_CREATE **被用来指定缺省的成员函数** OnCreate **与** WM_CREATE **相对应。在** MFC **中，包含了大量的预定义消息映射宏，用来指定各种成员函数与各种形如** WM_XXXX **消息相对应。如** ON_WM_CLOSE **宏指定了** WM_CLOSE **消息的处理成员函数为** OnClose**。这时侯，只需要写出要处理的消息就够了，不必再写出处理函数。消息映射宏** ON_COMMAND **则被用来将菜单项和用户自定义的命令同它们的处理成员函数联系起来。在上例中，用户选择** Edit->Copy **菜单项时，系统执行** OnEditCopy()**函数。**ON_COMMAND **宏的一般定义形式如下：**

```
ON_COMMAND(command,command_function)
```

其中，command **为菜单消息或用户自定义消息，**command_function **为消息处理函数。**MFC **允许用户自定义消息，常量** WM_USER **和第一个消息值相对应，用户必须为自己的消息定义相对于** WM_USER **的偏移值，偏移范围在** 0~0x3FFF **之间，这对绝大多数程序来说都是够用的。用户可以利用**#define **语句直接定义自己的消息：**

```
#define WM_USER1 (WM_USER+0)
```

#define WM_USER2 (WM_USER+1)
#define WM_USER3 (WM_USER+2)
下表列出了 Windows95 中 Windows 消息值的范围。
常 量/值/消息值范围/意 义/
WM_USER/0x0400/0x0000-0x03FF/Windows 消息/
//0x0400-0x7FFF/用户自定义的消息/
//0x8000-0xBFFF/Windows 保留值/
//0xC000-0xFFFF/供应用使用的字符串消息/
为了说明如何使用用户自定义消息，我们看一个例子，见程序清单2.4：
清单 2.4 使用用户自定义消息

```
#include<afxwin.h>
#define CM_APPLE (WM_USER+0)
#define CM_ORANGE (WM_USER+1)
class CMainFrame:CFrameWnd{
public:
CMainFrame();
protected:
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnClose();
//handle user select apple
afx_msg LRESULT CMApple(WPARAM wParam, LPARAM lParam);
//handle user select orange
afx_msg LRESULT CMOrange(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};
```

相应的消息映射如下：

```
BEGIN_MESSAGE_MAP(CMainFrame,CFrameWnd)
ON_WM_CREATE()
ON_MESSAGE(CM_APPLE, CMApple)
ON_MESSAGE(CM_ORANGE,CMOrange)
ON_WM_CLOSE()
END_MESSAGE_MAP()
```

第一个 ON_MESSAGE 宏用于指定 CM_APPLE 命令消息的处理成员函数为 CMApple，而第二个 ON_MESSAGE 宏用于指定 CM_ORANGE 命令消息的处理函数为 CMOrange。
消息的发送
Windows 应用程序允许应用程序向自己发送消息、向其他应用程序发送消息，甚至可以向 Windows 操作系统本身发送消息(比如要求关闭操作系统或重新启动操作系统)。Windows 提供了三个 API 函数用于发送消息，这三个函数是：SendMessage、PostMessage 和 SendDlgItemMessage。

SendMessage 用于向窗口发送消息，该函数说明如下：

```
LRESULT SendMessage(
HWND hWnd, //消息要发往的窗口的句柄
UINT Msg, //要发送的消息
WPARAM wParam, //消息的第一个参数
LPARAM lParam //消息的第二个参数
);
```

其中，hWnd 为接收消息窗口的句柄，参数 Msg 指定发送的消息，参数 wParam 和 lParam 依赖于消息 Msg。该函数调用目标窗口的窗口函数，直到目标窗口处理完该消息才返回。

PostMessage 函数同 SendMessage 类似，它把消息放在指定窗口创建的线程的消息队列中，然后不等消息处理完就返回，而不象 SendMessage 那样必须等到消息处理完毕才返回。目标窗口通过 GetMessage 或 PeekMessage 从消息队列中取出并处理。PostMessage 函数说明如下：

```
BOOL PostMessage(
HWND hWnd, //消息发往的窗口
UINT Msg, //要发送的消息
WPARAM wParam, //消息的第一个参数
LPARAM lParam //消息的第二个参数
);
```

其中，参数 hWnd 为接收消息的窗口的句柄，参数 Msg 指定所发送的消息，参数 wParam 和 lParam 依赖于消息 Msg。

SendDlgItemMessage 函数用于向对话框的某个控制发送消息，函数声明如下：

```
LONG SendDlgItemMessage(
HWND hDlg, //对话框句柄
int nIDDlgItem, //对话框控件的 ID
UINT Msg, //要发送的消息
WPARAM wParam, //消息的第一个参数
LPARAM lParam //消息的第二个参数
);
```

其中，hDlg 为包含目标控制的对话框的窗口句柄，参数 nIDDlgItem 为接收消息的对话框控制的整数标识符，参数 Msg 指定了所发送的消息，参数 wParam 和 lParam 提供附加的特定消息的信息。

MFC 将这三个函数封装为 CWnd 类的成员函数，隐藏了窗口句柄和对话框句柄。这三个成员函数用于向本窗口发送消息，函数的说明如下：

LRESULT SendMessage(UINT message, WPARAM wParam = 0, LPARAM lParam = 0);

BOOL PostMessage(UINT message, WPARAM wParam = 0, LPARAM lParam = 0);

LRESULT SendDlgItemMessage(int nID, UINT message, WPARAM wParam = 0, LPARAM lParam = 0);

2.4.5 学习 MFC 的方法

首先要对 Windows API 有一定的了解，否则无法深入学习 MFC。至少要知道 Windows 对程序员来说意味着什么，它能完成什么工作，它的一些常用数据结构等。

另一点是不要过分依赖于 Wizards。Wizards 能做许多工作，但同时掩饰了太多的细节。应当看看 AppWizard 和 ClassWizard 为你所做的工作。在 mainfrm.cpp 中运行调试器来观察一下 MFC 运行的流程。除非你理解了生成的代码的含义，否则无法了解程序是如何运行。

还有很重要的一点就是要学会抽象的把握问题，不求甚解。许多人一开始学习 Visual C++就试图了解整个 MFC 类库，实际上那几乎是不可能的。一般的学习方法是，先大体上对 MFC 有个了解，知道它的概念、组成、基本约定等。从最简单的类入手，由浅入深，循序渐进、日积月累的学习。一开始使用 MFC 提供的类时，只需要知道它的一些常用的方法、外部接口，不必要去了解它的细节和内部实现，把它当做一个模块或者说黑盒子来用，这就是一种抽象的学习方法。在学到一定程度时，再可以深入研究，采用继承的方法对原有的类的行为进行修改和扩充，派生出自己所需的类。

学习 MFC，最重要的一点是理解和使用 MFC 类库，而不是记忆。

2.5 移植 C Windows 程序到 MFC

Microsoft 和 Symantec 公司联合开发了一个工具，用于将 C Windows 移植到 MFC。这个工具叫作 MFC migration kit(MFC 移植工具)，可以从 VC++5.0 光盘的 MFCKIT 目录下找到这一工具。

2.6 Visual C++5.0 新特性

在 MFC 类库上，没有做什么大的改进，只是增加了对微软的 IntelliMouse(智能鼠标)的支持；对 Developer Studio，则作了一些修改，增加了一些新特性，使其更易于使用。这些新特性包括：

自动化和宏功能

可使用 Visual Basic 脚本语言自动化一些重复性过程和工作。宏记录功能可以根据用户的操作自动生成宏操作序列。Visual Studio 和它的组件都可以看作对象来处理，这意味着可以自动化诸如打开、编辑、关闭文档和调整窗口的操作。

可定制的工具条和菜单

可以灵活的定制菜单和工具条，使其更适合你的工作需要。比如，可以创建一个新的工具条和菜单；增加、删除菜单命令和工具条按钮等。

调试器

可以直接运行和调试程序，并用宏语言自动化调试工作。

支持 Internet 连接

可以直接在 Developer Studio 中察看 WWW 页面。可以使用全新的 InfoViewer 或自己注册的 Web 浏览器察看 Web 上的页面。当窗口中有一个 Web 地址(URL)时，可单击该地址察看 Web 页面。该特性可以让 Visual Studio 用户了解最新信息、获取更新的文档以及完成产品的升级和修正工作。

项目工作区和文件

一个新的便捷的项目系统允许一个工作区内包含多个不同的项目类型。比如说，可以创建一个包含 Visual C++工程和 J++ Applet 的工作区。

工作区现在以.dsw 为后缀名(以前为.mdp)项目文件现在以.dsp(以前为.mak)为后缀名。

Build 文件现在分为两种：内部(.dsp)和外部(.mak)。在 Developer Studio 中创建一新的工程或从以前版本的工程进行转换时创建内部 Build 文件。内部 Build 文件与 NMAKE 外部编译工具不兼容。可以通过选择 Project 菜单上的 Export Makefile 创建一个与 NMAKE 兼容的外部 Build 文件。

在工作区内可以包含多个并列的工程文件

要在一个工作区内增加一个工程，可以打开该工作区，然后选择 Project->Insert Project into Workspace...菜单项，在当前工作区中增加一个工程。

通过选择 Project->Set Active Project 菜单项，可以设置当前活动工程，也就是执行 Build 操作时编译的那一个工程。

这可能是令许多程序员为之欢呼雀跃的一个特性，因为这一特性对于在不同工程之间复制代码和资源是非常方便的。工程之间还支持鼠标直接拖放对象特性，这样用户可以直接从一个工程的资源文件中拖动一个对话框资源，然后放到另一个工程的资源文件中。而在以前版本中要完成类似的操作，必需先打开一个工程，然后打开另一个工程的资源文

件，再进行资源复制操作。

增强的资源编辑器

在 Visual C++中，可以在对话框中使用 WizardBar 将程序同可视化元素联系起来。

在使用加速键、对话框、菜单、字符串时，如果需要对多个项作同一修改，可以选择所有要改的项目，然后在 View 菜单中点 Properties；在 Properties 对话框中一次完成多个项的值的更改。

提示：要选择多个项，可以先用鼠标点某一项，然后按住 CTRL 键，再用鼠标点其他项。或用鼠标器拉框选择一个区域的多个项。要向从多个项中去掉一个选择，可以按 CTRL 键，点击那一项。

文本编辑器

Find in Files 命令现在支持两个独立的输出窗格，这样用户就可以保存上一次搜索的结果了。

小结

在这一章里，我们主要向读者介绍了：

Visual C++集成开发环境 Visual Studio 的使用：包括 Visual Studio 的组成，项目工作区概念及类视图(ClassView)、文件视图(FileView)、资源视图(ResourceView)、信息视图(InfoViewer)的使用，如何管理工程等。还介绍了 Visual Studio 的两个重要可视化编程工具：AppWizard、ClassWizard 的使用。AppWizard 提供一系列对话框，让程序员指定所要创建的应用程序的一些特性，然后自动生成框架程序，程序员只需要在框架基础上修改就可以开发出自己的应用程序。ClassWizard 提供了一种维护框架程序的手段，主要是增加、删除类，添加类数据程序、方法以及映射消息和成员函数等。

Win32 编程：Win32 API 是 32 位的 Windows 操作系统上的一个开发工具，它支持高性能的抢先式多任务和多线程、连续的 32 位地址空间和先进的内存管理、对所有的可为进程共享的对象，解决了它的安全性问题；它还支持内存映射文件。我们还介绍了 Win32 编程的一些基本概念，包括数据类型、变量命名方法(匈牙利命名法)。还简要介绍了由 Win16 向 Win32 移植时一些需要注意的地方。

MFC 类库：包括使用 MFC 的好处，MFC 的历史、MFC 基本类库的组成、约定，类库和应用程序框架的概念、MFC 对消息映射的管理、如何学习 MFC 类库等。

最后我们介绍了 Visual C++5.0 的一些新特性。

本教程由 Visual C++王朝(Where programmers come together)协助制作未经许可，请勿以任何形式复制/

第三课　窗口、菜单与消息框

在这一课里，我们将通过一个简单的例子说明窗口、菜单、消息框的编程技术，并介绍用户接口更新消息机制。

3.1 编写第一个窗口程序

现在我们开始编写全书的第一个程序。跟我们以前学习程序设计的方法不同(以前我们是输入完整程序，然后运行)，我们首先利用 Visual Studio 的可视化编程工具 AppWizard 生成框架程序，再往里边填写代码。这是一种“填空式”的编程方法：首先生成框架，然后根据目标程序的要求，看哪些地方需要修改，再往里填写代码。类似其他语言，我们把第一个程序命名为 Hello。

首先启动 AppWizard：在 File 菜单下选择 New，弹出 New 对话框，如图 3.1 所示。

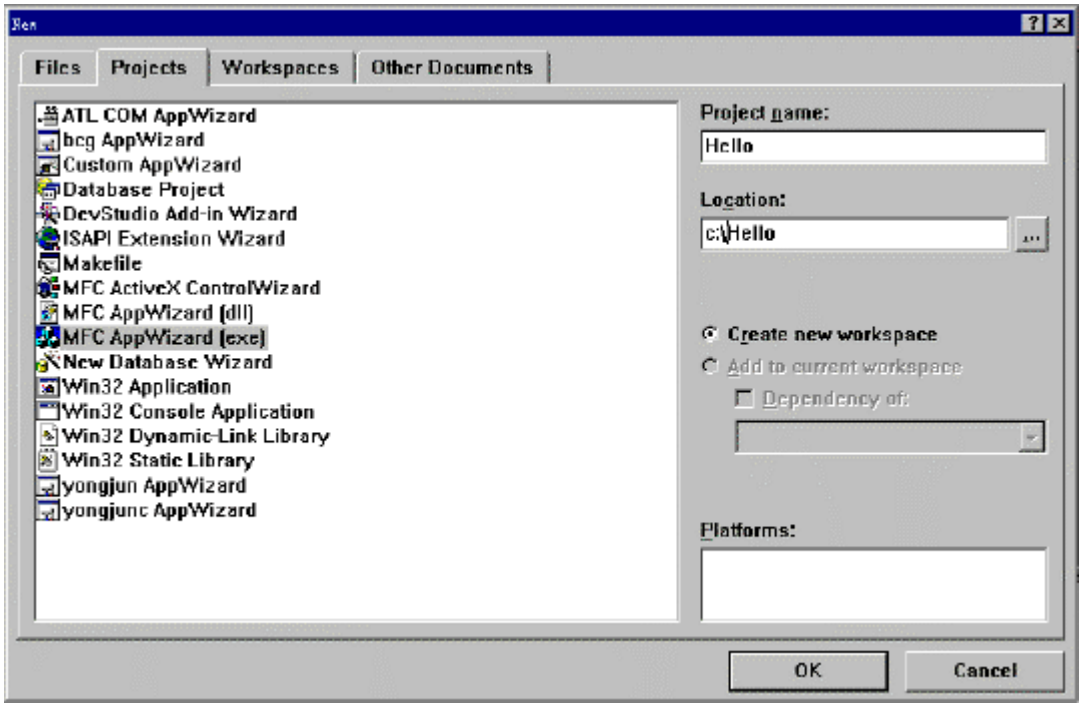


图 3.1 New 对话框

在对话框顶部有一排标签，用于选择要创建的文档的类型。选择 Projects 标签，然后在列表中选择 MFC AppWizard(exe)，告诉 Visual C++ 要使用 AppWizard 创建一个 EXE 程序；在 Project Name 编辑框中输入工程文件名 Hello，在 Location 中指定应用程序创建的位置，缺省情况下 AppWizard 会自动在当前目录下以工程文件名为名字创建一个新目录，在该目录下存放所有该工程的文件。这里将目录设置为“C:\Hello”，然后选择 OK 按钮，此时弹出 MFC-AppWizard-Step1 对话框，如图 3.2 所示。

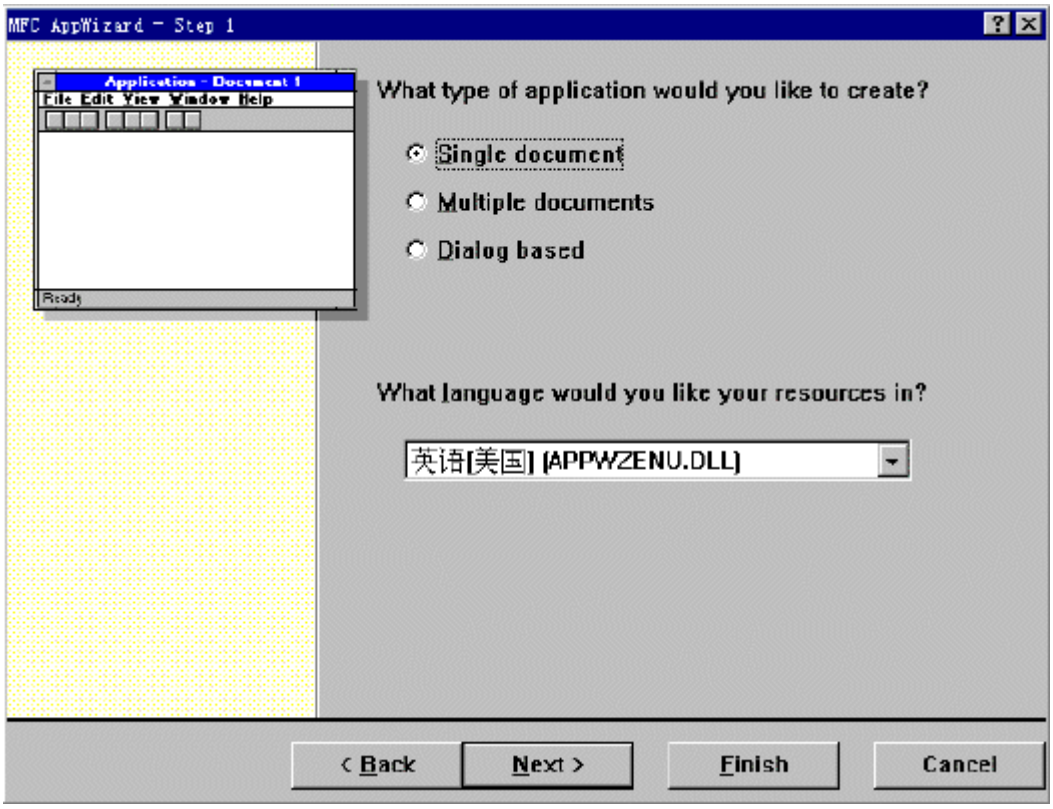


图 3.2 MFC AppWizard-Step 1 **对话框**

AppWizard **是一个自动化程序生成工具，它通过提示用户一系列对话框，来指定将要生成的应用程序的特性，然后自动生成相应的代码。下面，我们一步一步演示如何用** AppWizard **生成** Hello **程序。**

1.MFC AppWizard **当前显示** MFC AppWizard-Step1 **对话框。在这个对话框中，可以指定生成框架的类型，包括** Single Document(**单文档**)，Multiple Document(**多文档**)，Dialog Based(**基于对话框**)**三种。还可以从下拉列表框中选择语言，指定程序资源文件使用的语言类型。选择** Single Document，**此时** AppWizard **将生成一个单文档的应用程序框架，也就是说，应用程序运行时是一个单窗口的界面。点击** Next **按钮。**

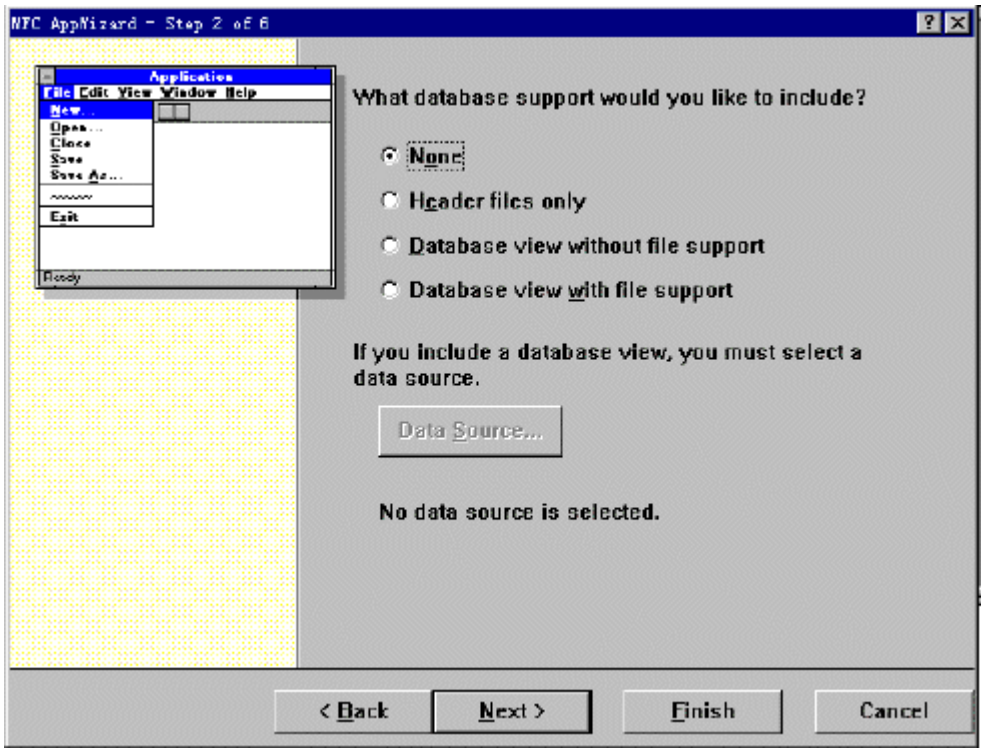


图 3.3 MFC AppWizard-Step 2 of 6 **对话框**

2.**此时** MFC AppWizard **显示图** 3.3 **所示** MFC AppWizard-Step 2 of 6 **对话框。该对话框用于指定数据库选项。**MFC AppWizard **支持数据库并可以生成数据库应用程序所必需的代码。选择缺省值** None**，不使用数据库特性。点击** Next **按钮，弹出** MFC AppWizard-Step3 **对话框，如图** 3.4 **所示。**

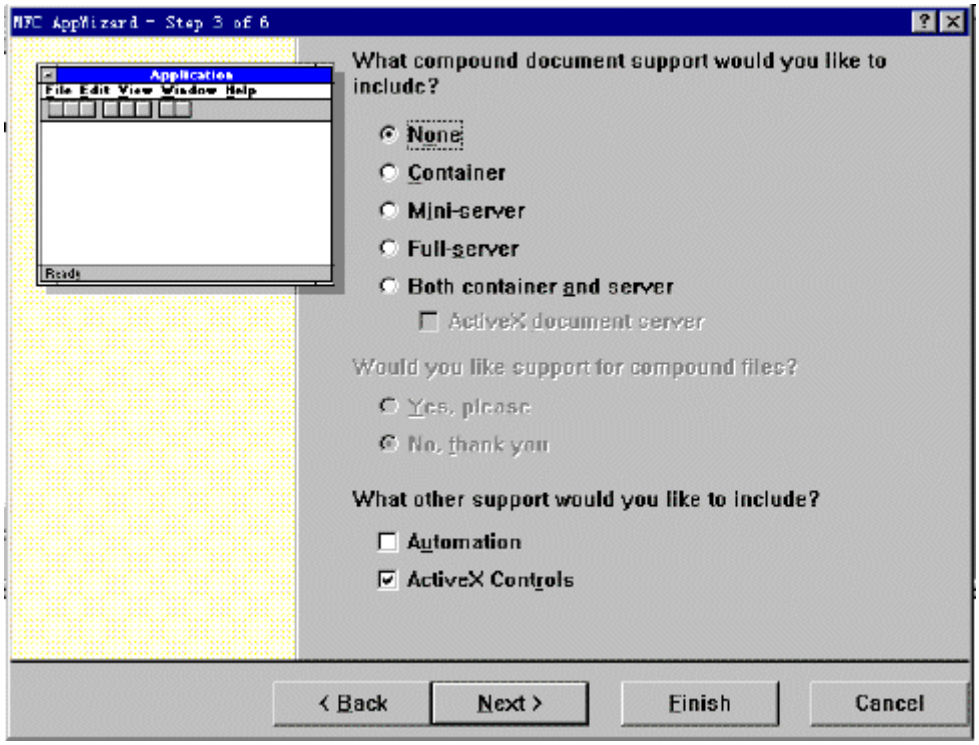


图 3.4 MFC AppWizard-Step 3 of 6 **对话框**

3.在 MFC AppWizard-Step 3 of 6 对话框中，可以指定 OLE 选项的复合文档类型。因为我们不用 OLE 特性，所以使用缺省值 None。点击 Next 按钮，进入下一个对话框。此时，屏幕显示 MFC-AppWizard-Step 4 of 6 对话框，如图 3.5。

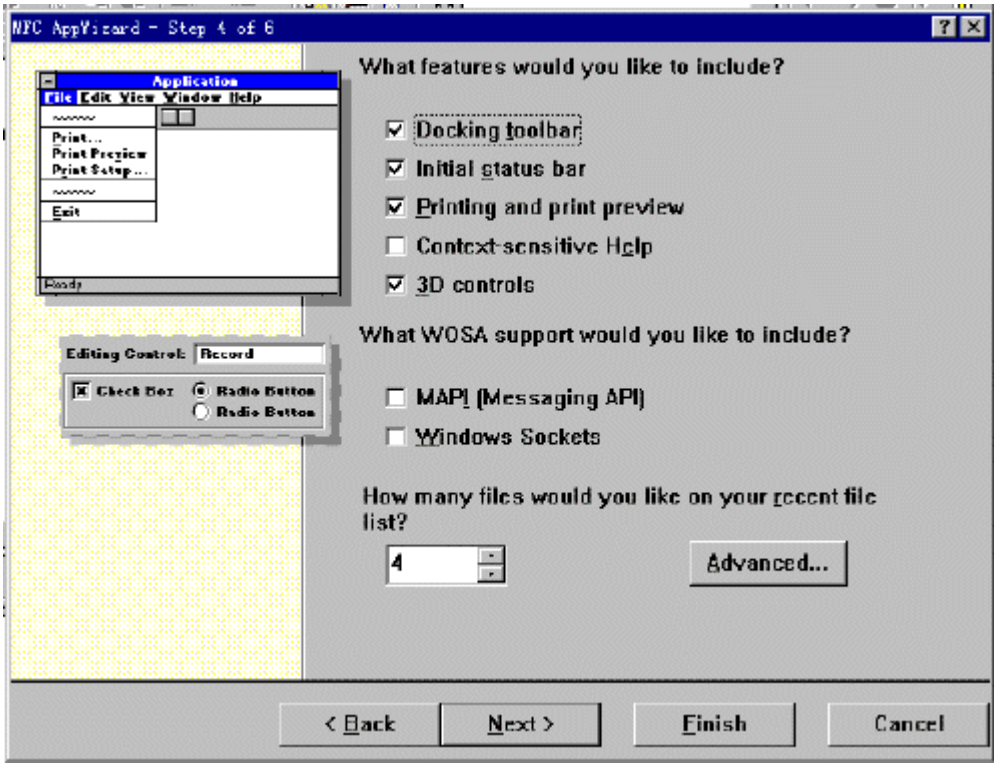


图 3.5 MFC AppWizard-Step 4 of 6 对话框

4.第四个对话框用于指定应用程序的外观，包括是否使用工具条、状态栏，是否让文档支持打印和打印预览功能，是否使用 3D 控制外观，以及是否支持在线帮助等。MFC AppWizard 还支持 WOSA(Windows 开放系统体系结构)，可以直接在基于文档的程序中加入 MAPI 电子邮件发送功能和 WinSocket 网络编程接口支持。另外，还可以指定文档的一些特性，包括后缀名等。有关文档/视结构的内容在后面章节中再作详细介绍。按照图 3.5 所示，设置各个选项，它支持工具条、状态栏，使用 3D 外观的控制。点击 Next 按钮，弹出 MFC AppWizard 5 of 6 对话框，如图 3.6 所示。

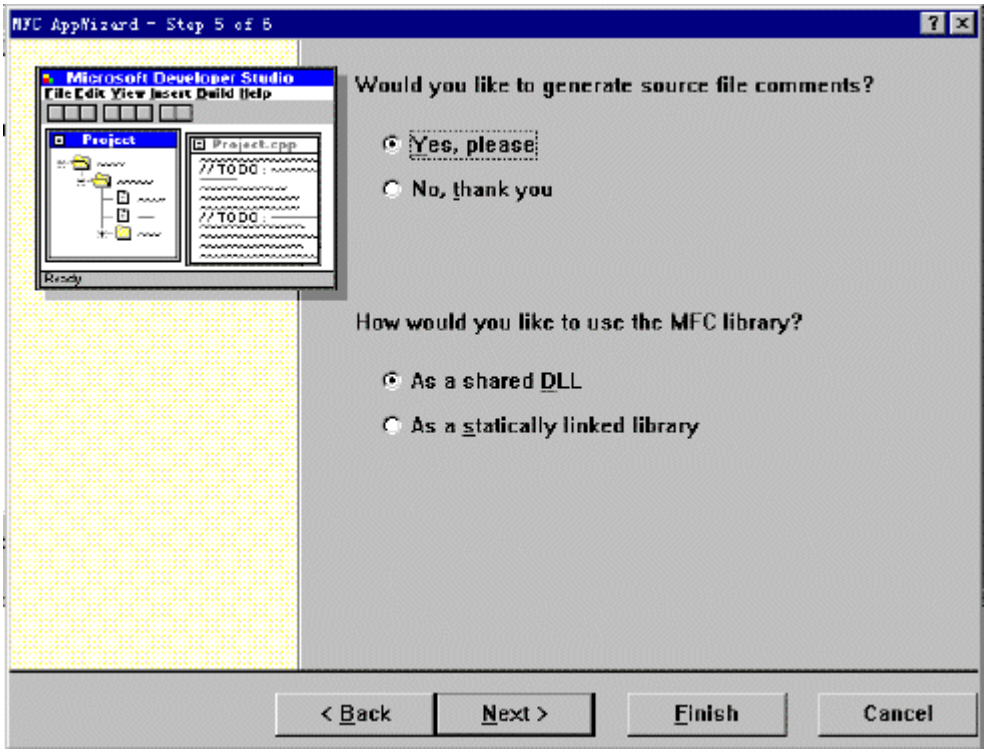


图 3.6 MFC AppWizard-Step 5 of 6 **对话框**

5.MFC AppWizard 5of 6 **有两个选项，让用户设置生成源代码的选项及编译选项。**

第一个选项是：Would you like to generate source file comments”**询问** MFC AppWizard **是否为生成的源代码添加必要的注释说明。注释说明有助于对应用程序源代码的学习和理解，因此一般选择“** Yes，Please **”。**

第二个选项是：How would you like to use the MFC library”

用户可以选择 As a shared DLL(**使用共享动态连接库**)**或** As a static linked library(**静态连接库**)**。使用** DLL **时，所有** MFC **的类存放在动态连接库中，因此可以使应用程序小一些，但是发布该应用程序可执行文件时必需随同提供必要的动态连接库。使用静态库时，应用程序所用到的** MFC **类都编译进了可执行文件之中，因此可执行文件比使用** DLL **方式的要大，但可以单独发行。一般的，对于小的应用程序可以采用静态库方式，对于大的应用程序一般采用动态连接库方式。本书中的例子全部采用动态连接库选项**(As a shared DLL)**。**

6.**点击** Next **按钮以进入** MFC AppWizard **的最后一个对话框。此时，将出现如图** 3.7 **所示的对话框。**

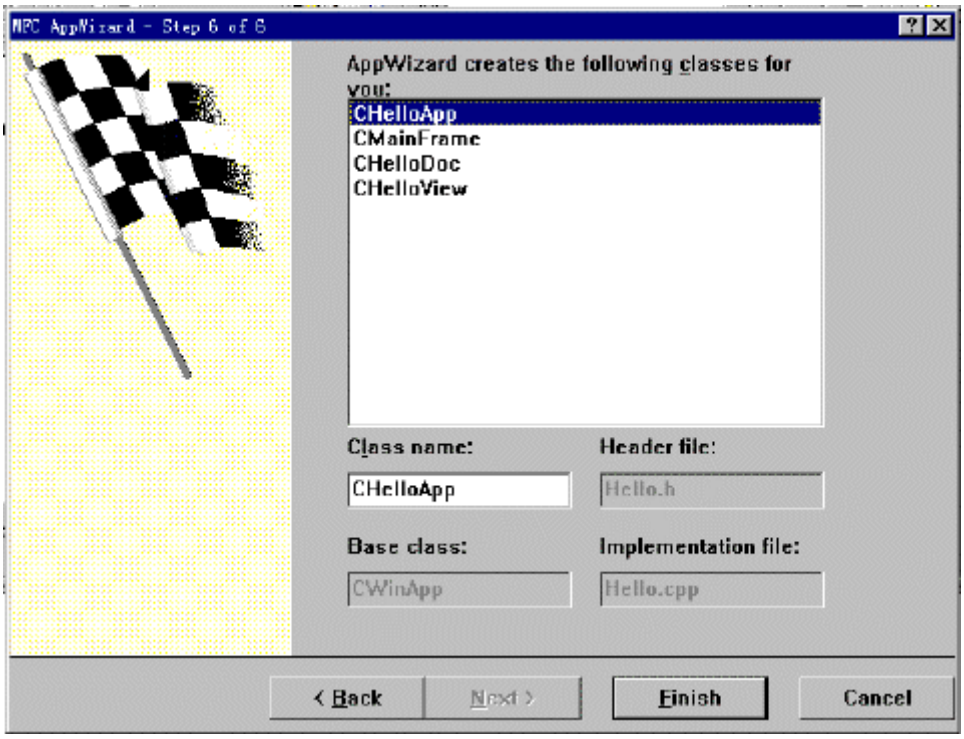


图 3.7 MFC AppWizard-Step 6 of 6 对话框

该对话框让用户选择 MFC AppWizard 将要创建的类的属性，包括指定存放类的文件名以及这些类的基类。在这一步，用户可以修改所创建的类的名字、对应的文件以及赖以派生的父类。对于 Hello 程序，直接对话框中的使用缺省设置。

7.点 Finish 按钮，弹出 New Project Information 对话框。它给出一个关于即将生成的应用程序的总体描述，包括应用程序类型(单文档)、要创建的类及所在文件、应用程序的一些特性(是否支持工具条、状态栏等)。按 Enter 键或点 OK 按钮，此时 AppWizard 将在 c:\hello 目录下生成 Hello 程序所有的框架文件。生成程序后，在项目工作区中自动打开 Hello.dsw 项目工作区文件。窗口标题将显示打开的项目名 Hello。

3.2 AppWizard 所创建的文件

AppWizard 在读者指定的 C:\Hello 目录下创建了许多文件，这些文件包含了框架程序的所有的类、全局变量的声明和定义。初学者面对这一大堆文件可能会不知所措。现在我们把各个文件的作用及对应的类介绍一下。

根据可选项，AppWizard 所创建的文件会略有不同。标准的 AppWizard 文件包括：

工作区文件、项目文件和 make 文件

应用程序源文件和头文件

资源文件

预编译头文件

按可选项增加的 AppWizard 文件

3.2.1 工作区、项目文件和 make 文件

Hello.dsw　这是 MFC 自动生成的工作区文件，它包含当前工作区所包含的项目的信息。

Hello.dsp　这是 MFC 生成的项目文件，它包含当前项目的设置、所包含的文件等信息。

Hello.MAK　这是 MFC 项目的项目文件，这也是与 NMAKE 兼容的文件。如果选择了 External make 文件可选项，则可人工对它编辑，但不能利用 Visual C++许多项目编辑特性。

Hello.CLW　这个文件含有被 ClassWizard 用来编辑现有类或增加新类的信息。ClassWizard 还用这个文件来保存创建和编辑消息映射和对话框数据所需的信息，或是创建虚拟成员函数所需的信息。

3.2.2 应用程序源文件和头文件

根据应用程序的类型—单文档、多文档或基于对话框的，AppWizard 将创建下述应用程序源文件和头文件中的某些文件。在本例中，AppWizard 生成了如下文件：

Hello.h　这是应用程序的主头文件，它含有所有全局符号和用于包含其它头文件的#include 伪指令。

Hello.CPP　这个文件是应用程序的主源文件。它将创建 CHelloApp 类的一个对象(从 CWinApp 派生)，并覆盖 InitInstance 成员函数。

MainFrm.cpp，MainFrm.h　这两个文件将从 CFrameWnd(SDI 应用程序)或 CMDIFrameWnd(MDI 应用程序)派生 CMainFrame 类。如果在 AppWizard 的 Application Options 页(6 步中的第 4 步)中选择了对应的可选项的话，CMainFrame 类将处理工具条按钮和状态条的创建。MAINFRM.CPP 文件还含有 MFC 应用程序提供的默认工具条按钮的对象 ID——叫做 buttons 数组。

HelloDoc.cpp，HelloDoc.h　这些文件从 CDocument 类派生并实现名为 CHelloDoc 的文档类，并含有用于初始化文档、串行化(保存和装入)文档和用于调试诊断的一些成员函数的框架。

HelloView.cpp，HelloView.h　这些文件派生并实现名为 CHelloView

的视类，用于显示和打印文档数据。CHelloView 类是从 CView 或它的派生类派生出来的，含有绘制视和用于调试诊断的一些成员函数框架。

3.2.3 资源文件

AppWizard 会创建一些与资源相关的文件。

Hello.RC，RESOURCE.H，Hello.rc2 这是项目的头文件及其资源文件。资源文件含有一般 MFC 应用程序的默认菜单定义和加速键表、字符串表。它还指定了缺省的 About 对话框和一个图标文件(RES\Hello.ICO)。资源文件了标准的 MFC 类的资源。如果指定了支持工具条，它还将指定工具条位图文件(RES\TOOLBAR.BMP)。Hello.rc2 用于存放 Visual Studio 不可直接编辑的资源。

3.2.4 预编译头文件：STDAFX.CPP，STDAFX.H

这两个文件用于建立一个预编译的头文件 Hello.PCH 和一个预定义的类型文件 STDAFX.OBJ。由于 MFC 体系结构非常大，包含许多头文件，如果每次都编译的话比较费时。因此，我们把常用的 MFC 头文件都放在 stdafx.h 中，如 afxwin.h、afxext.h、afxdisp.h、afxcmn.h 等，然后让 stdafx.cpp 包含这个 stdafx.h 文件。这样，由于编译器可以识别哪些文件已经编译过，所以 stdafx.cpp 就只编译一次，并生成所谓的预编译头文件(因为它存放的是头文件编译后的信息，故名)。如果读者以后在编程时不想让有些 MFC 头文件每次都被编译，也可以将它加入到 stdafx.h 中。采用预编译头文件可以加速编译过程。

3.3 编译和链接 Hello 程序

虽然我们到现在为止还没有写任何一行代码，但我们确实得到了一个完整的可运行的程序。要编译运行程序，可以选择 Build-(或按快捷键 F7)，编译该程序。编译完后再选择 Build-Execute Hello.exe(或按快捷键 CTRL+F5)，运行该程序。也可以直接按 CTRL+F5，系统提示是否编译，回答“ Yes ”，Visual Studio 将自动编译链接并运行 Hello.exe 程序。

提示：在 Build 菜单下有 Compile,Build,Rebuild All 三个菜单项用于编译程序。其中 Compile 用于编译当前打开的活动文档；Build 只编译工程中上次修改过的文件，并链接程序生成可执行文件。如果以前没有作过编译，它会自动调用 Rebuild All 操作，依次编译资源文件、源程序文件等；Rebuild All 不管文件是否作过修改，都会编译工程中的所有源文件。由于编译链接过程中会产生大量的中间文件和目标文件，它们占用许多硬盘空间，因此 Visual Studio 在 Build 下提供了 Clean 菜单项用于清除这些中间文件。用户在完成一个工程后，应及时清理这些中间文件，否则硬盘很快会被耗尽。/

Hello.exe 程序执行后，显示如图 3.8 所示的窗口。窗口标题为 Untitled-Hello。现在我们要在窗口内显示“ Hello,world ”字样，为此需要手工编辑代码。在类视图(ClassView)中点 CHelloView 前面的加号，展开 CHelloView 树，显示它的类成员函数和数据成员。双击 OnDraw 函数，Visual Studio 将打开 HelloView.cpp 文件并将光标定位在 OnDraw 函数定义开始处。在 OnDraw 函数中手工加入代码，如下所示(黑体字为自己加入的代码)。

```
void CHelloView::OnDraw(CDC* pDC)
{
CHelloDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
CString str(“Hello,World!”);
pDC->TextOut(10,10,str);
}
```

编译并运行该程序，弹出如图 3.9 所示的窗口。在窗口左上角显示“ Hello，World！”。

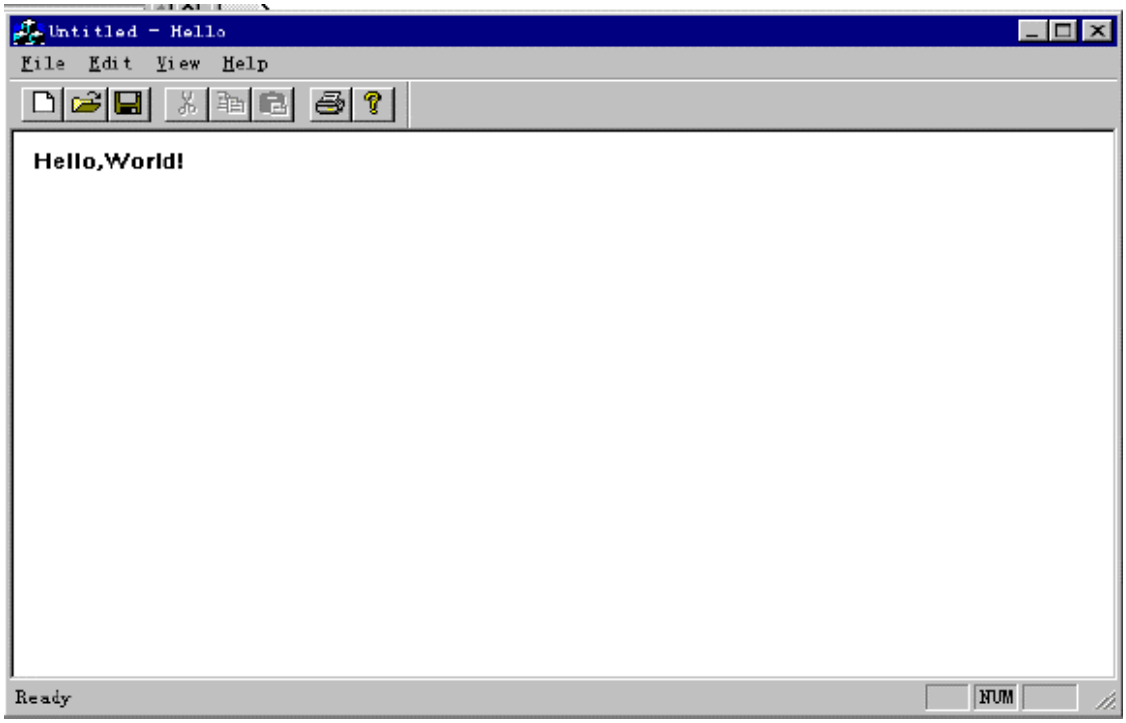


图 3.9 **显示“** Hello,World **!”的窗口**

到现在为止，我们只写了两行代码，就完成了 SDK **下需要几百行程序才能完成的工作。这应当归功于** Visual Studio **提供的** AppWizard **以及** MFC **框架在幕后所作的大量工作。为了更好的理解和设计基于** MFC **框架的程序，我们分析一下** MFC **框架所做的工作。**

上一页/下一页/

本教程由 Visual C++**王朝**(Where programmers come together)**协助制作未经许可，请勿以任何形式复制/**

3.4 应用程序执行机制

3.4.1 WinMain 函数

在 DOS 下，程序的执行是从 main 函数开始的。在 Windows 下，对应的函数是 WinMain。但是，如果浏览 Hello 程序的所有的方法和全局函数，是找不到 WinMain 函数的。MFC 考虑到典型的 Windows 程序需要的大部分初始化工作都是标准化的，因此把 WinMain 函数隐藏在应用程序的框架中，编译时会自动将该函数链接到可执行文件中。程序员可以重写 WinMain 函数，但一般不需要这么做。

下面的程序清单 3-1 给出了 WinMain 函数的代码。其中，_tWinMain 函数在 \DevStudio\Vc\Mfc\src\AppModul.cpp 中定义，它所调用的 AfxWinMain 函数在同一目录下的 WinMain.cpp 中定义。名字是 _tWinMain 函数而不是 WinMain，是考虑到对不同字符集的支持，在 tchar.h 中有_tWinMain 的宏定义。在 ANSI 字符集下编译时，_tWinMain 就变成 WinMain，在 Unicode 下编译时，_tWinMain 就变成 wWinMain。

提示：Unicode 是具有固定宽度、统一的文本和字符的编码标准。由于 Unicode 采用的是 16 位编码，因此可以包含世界各地的书写系统的字符和技术符号(如中文也在 Unicode 之中)，从而克服了 ASCII 码在表示多语言文本上的不足之处，扩大了 ASCII 码 7 位编码方案的好处。Unicode 同等地对待所有的字符，并且在表示各种语言的任何字符时既不需要换码序列(escape)也不需要控制代码。Win32 和 Visual C++很好的支持 Unicode 字符集。

清单 3-1 _tWinMain 函数定义

```
// export WinMain to force linkage to this module
extern int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow);
#ifdef _MAC
extern “C” int PASCAL
#else
extern “C” int WINAPI
#endif
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow)
{
// call shared/exported WinMain
return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

AfxWinMain 函数定义：

```
////////////////////////////////////////////////////////////////////////////
// Standard WinMain implementation
// Can be replaced as long as 'AfxWinInit' is called first
```

```
int AFXAPI AfxWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPTSTR lpCmdLine, int nCmdShow)
{
ASSERT(hPrevInstance == NULL);
int nReturnCode = -1;
CWinApp* pApp = AfxGetApp();
// AFX internal initialization
if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
goto InitFailure;
// App global initializations (rare)
ASSERT_VALID(pApp);
if (!pApp->InitApplication())
goto InitFailure;
ASSERT_VALID(pApp);
// Perform specific initializations
if (!pApp->InitInstance())
{
if (pApp->m_pMainWnd != NULL)
{
TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
pApp->m_pMainWnd->DestroyWindow();
}
nReturnCode = pApp->ExitInstance();
goto InitFailure;
}
ASSERT_VALID(pApp);
nReturnCode = pApp->Run();
ASSERT_VALID(pApp);
InitFailure:
#ifdef _DEBUG
// Check for missing AfxLockTempMap calls
if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
{
TRACE1("Warning: Temp map lock count non-zero (%ld).\n",
AfxGetModuleThreadState()->m_nTempMapLock);
}
AfxLockTempMaps();
AfxUnlockTempMaps(-1);
#endif
AfxWinTerm();
return nReturnCode;
```

}

应用程序执行时，Windows **自动调用应用程序框架内部的** WinMain **函数。如清单** 3-1 **所示，**WinMain **函数会查找该应用程序的一个全局构造对象，这个对象是由** CWinApp **派生类构造的，有且只有一个。它是一个全局对象，因此在程序启动时，它就已经被构造好了。**

随后，WinMain **将调用这个对象的** InitApplication **和** InitInstance **成员函数，完成应用程序实例的初始化工作。随后，**WinMain **调用** Run **成员函数，运行应用程序的消息循环。在程序结束时，**WinMain **调用** AfxWinTerm **函数，做一些清理工作。**

3.4.2 **应用程序类**

每个应用程序必须从 CWinApp **派生出自己的应用程序类，并定义一个全局的对象。该应用程序类包含了** Windows **下应用程序的初始化、运行和结束过程。基于框架建立的应用程序必须有一个(且只能有一个)从** CWinApp **派生的类的对象。在** Hello **程序中，我们从** CWinApp **中派生出一个** CHelloApp **类，并定义了一个全局对象** theApp**。**CHelloApp **类在** hello.cpp **中定义。**

要访问应用程序类构造的对象，可以调用全局函数 AfxGetApp()**。**AfxGetApp()**返回一个指向全局对象的指针。可以通过对它进行强制类型转换，转换为我们派生的应用程序类。**

比如：

CHelloApp* pApp=(CHelloApp*)AfxGetApp()**；**

在 CHelloApp **应用程序类中，我们还重载了** CWinApp **的成员函数** InitInstance**。**InitInstance **函数主要完成以下工作：设置注册数据库，载入标准设置(最近打开文件列表等)、注册文档模板。其中注册文档模板过程中隐含地创建了主窗口。接着，处理命令行参数，显示窗口，然后返回、进入消息循环。下面的程序清单** 3.2 **给出了** Hello **程序的** InitInstance **函数代码。**

清单 3.2 InitInstance **函数**

```
// CHelloApp initialization
BOOL CHelloApp::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
// the specific initialization routines you do not need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
// Change the registry key under which our settings are stored.
```

```
// You should modify this string to be something appropriate
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(); // Load standard INI file options (including MRU)
// Register the application's document templates. Document templates
// serve as the connection between documents, frame windows and views.
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CHelloDoc),
RUNTIME_CLASS(CMainFrame), // main SDI frame window
RUNTIME_CLASS(CHelloView));
AddDocTemplate(pDocTemplate);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
// The one and only window has been initialized, so show and update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
return TRUE;
}
```

在 CWinApp **的派生类中，必须重载** InitInstance **函数，因为** CWinApp **并不知道应用程序需要什么样的窗口，它可以多文档窗口、单文档窗口，也可以是基于对话框的。**

Run **成员函数** WinMain **在初始化应用程序实例后，就调用** Run **函数来处理消息循环。**Run **成员函数不断执行消息循环，检查消息队列中有没有消息。如果有消息，**Run **将其派遣，交由框架去处理，然后返回继续消息循环。如果没有消息，**Run **将调用** OnIdle **来做用户或框架可能需要在空闲时才做的工作，象后面我们讲到的用户接口更新消息处理等。如果既没有消息要处理，也没有空闲时的处理工作要做，则应用程序将一直等待，直到有事件发生。当应用程序结束时，**Run **将调用** ExitInstance**。消息循环的流程图如图** 3-10 **所示。**

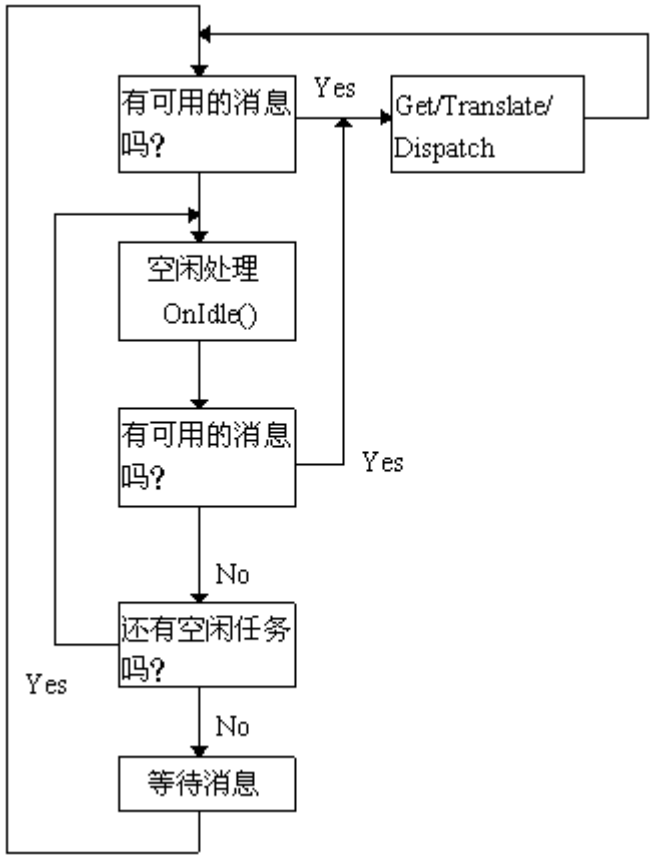


图 3-10 Run **成员函数的消息循环**

关闭应用程序

用户可以通过选择 File-Exit **菜单或点主窗口的关闭按钮，关闭主框架窗口，来终止应用程序。此时，应用程序类首先删除** m_pMainWnd **主框架窗口对象，然后退出** Run **函数，进而退出** WinMain**，在退出** WinMain **后删除** TheApp **对象。**

3.5 几种窗口类型

3.5.1 框架窗口

框架窗口为应用程序的用户界面提供结构框架，它是应用程序的主窗口，负责管理其包容的窗口，一个应用程序的最顶层的框架窗口是应用程序启动时创建的第一个窗口。

MFC 提供三种类型的框架窗口：单文档窗口，多文档窗口(MDI)，对话框。在 AppWizard 的第一个对话框中，就提供了选项，让用户选择应用程序是基于单文档、多文档还是对话框的。MFC 单文档窗口一次只能打开一个文档框架窗口，而 MDI 应用程序运行时，在应用程序的一个实例中打开多个文档框架窗口，这些窗口称作子窗口(Child Window)。这些文档可以是同一类型的，也可以是不同类型的。如 Visual Studio 就可以打开资源文件窗口和源程序窗口等不同类型的窗口。此时，激活不同类型的 MDI 子窗口，菜单也将相应变化。

MFC 提供了三个类 CFrameWnd、CMDIFrameWnd、CMDIChildWnd 和 CDialog 分别用于支持单文档窗口、多文档窗口和对话框。

CframeWnd 用于 SDI 框架窗口，形成单个文档及其视的边框。框架窗口既是应用程序的主框架窗口，也是当前文档对应的视图的边框。

CMDIFrameWnd 用于 MDI 应用程序的主框架窗口。主框架窗口是所有 MDI 文档窗口的容器，并与它们共享菜单条。MDI 框架窗口是出现在桌面中的顶层窗口。

CMDIChildWnd 用于在 MDI 主框架窗口中显示打开的各个文档。每个文档及其视都有一个 MDI 子框架窗口，子框架窗口包含在 MDI 主框架窗口中。子框架窗口看起来类似一般的框架边框窗口，但它是包含在主框架窗口中，而不是位于桌面的，并且为主窗口所裁剪。而且 MDI 子窗口没有自己的菜单，它与主 MDI 框架窗口共享菜单。

Cdialog 对话框是一种特殊类型的窗口，它的边框一般不可以调整，而且内部包含一些控件窗口。有关对话框作为主窗口的技术可以参见下一章。

要生成一个单文档窗口，主窗口就必须从 CFrameWnd 派生；要生成一个多文档窗口，主窗口就必须从 CMDIFrameWnd 派生，而且其中的子窗口必须从 CMDIChildWnd 派生出来；而基于对话框的窗口程序就要从 CDialog 派生出主窗口类。

子窗口子窗口就是具有 WS_CHILD 风格的窗口，且一定有一个父窗口。所有的控件都是子窗口。子窗口可以没有边框。子窗口被完全限制在父窗口内部。

父窗口 父窗口就是拥有子窗口的窗口。

弹出式窗口具有 WS_POPUP 风格，它可以没有父窗口。这种窗口几乎什么都没有，可看作一个矩形区域。

3.5.2 窗口的创建

窗口的创建分为两步：第一步是用 new 创建一个 C++的窗口对象，但是此时只是初始化窗口的数据成员，并没有真正创建窗口(这一点与一

般的对象有所不同)。

//第一步：创建一个 C++对象，其中 CMainFrame 是从 CFrameWnd 派生的对象。

CMainFrame* pMyFrame=new CMainFrame();//用 new 操作符创建窗口对象

或

CMainFrame MyFrame;//定义一个窗口对象，自动调用其构造函数

第二步是创建窗口。CFrameWnd 的 Create 成员函数把窗口给做出来，并将其 HWND 保存在 C++对象的公共数据成员 m_hWnd 中。

//第二步：创建窗口

pMyFrame->Create(NULL,“My Frame Window”);

或

MyFrame.Create(NULL,“My Frame Window”);

Create 函数的原形如下：

BOOL Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle = WS_OVERLAPPEDWINDOW, const RECT& rect = rectDefault, CWnd* pParentWnd = NULL, LPCTSTR lpszMenuName = NULL, DWORD dwExStyle = 0, CCreateContext* pContext = NULL);

Create 函数第一个参数为窗口注册类名，它指定了窗口的图标和类风格。这里我们使用 NULL 做为其值，表明使用缺省属性。第二个参数为窗口标题。其余几个参数指定了窗口的风格、大小、父窗口、菜单名等。

这个函数看起来比较复杂，对于 CFrameWnd 派生出来的窗口，我们可以使用 LoadFrame 从资源文件中创建窗口，它只需要一个参数。

pMyFrame->LoadFrame(IDR_FRAME)；

LoadFrame 使用该参数从资源中获取许多默认值，包括主边框窗口的标题、图标、菜单、加速键等。但是，在使用 LoadFrame 时，必须确保标题字符串、图标、菜单、加速键等资源使用同一个 ID 标识符(在本例中，我们使用 IDR_FRAME)。

提示：在 Hello 程序的 InitInstance 中我们看不到创建窗口的过程。实际上，在

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

AddDocTemplate(pDocTemplate); 程序片段中，我们看到，CSingleDocTemplate 构造函数的第二个参数就是 IDR_MAINFRAME。在构造函数内部，已经通过调用 m_pMainWnd->LoadFrame(IDR_MAINFRAME)，完成了应用程序主窗口的创建过程。

在 InitInstance 中，创建完窗口后，窗口调用 ShowWindow 成员函数来显示窗口。ShowWindow 带一个参数，指示窗口以何种方式显示(最大化、最小化或一般)。缺省方式为 SW_SHOW，但实际上我们经常希望应用程序启动时窗口最大化，此时可以将该参数该为 SW_SHOWMAXMIZED，即调用

m_pMainWnd->ShowWindow(SW_SHOWMAXIMIZED)；

在 MainFrm.cpp 中，我们还看到 CMainFrame 类有一个 OnCreate 方法。OnCreate 成员函数定义如清单 3.3。当调用 Create 或 CreateEx 时，操作系统会向窗口发送一条 WM_CREATE 消息。这一函数就是用来响应 WM_CREATE 消息的。

清单 3.3 OnCreate 成员函数定义

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
return -1;
if (!m_wndToolBar.Create(this) ||
!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
TRACE0(“Failed to create toolbar\n”);
return -1; // fail to create
}
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT)))
{
TRACE0(“Failed to create status bar\n”);
return -1; // fail to create
}
// TODO: Remove this if you don't want tool tips or a resizeable toolbar
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
// TODO: Delete these three lines if you don't want the toolbar to
// be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;
}
```

在 OnCreate 函数中，首先调用 CFrameWnd 的缺省处理方法 OnCreate 完成窗口创建工作。后面是应用程序主窗口的特定工作，在上面程序中，创建了工具条和状态栏(有关工具条和状态栏编程参见下一章有关内容)。可以在此处加入一些初始化工作，如从 INI 文件中载入设置，显示 Splash

Window(启动画面)等。

3.5.3 注册窗口

在传统的 Windows C 程序中，送给一个窗口的所有消息是在它的窗口函数中处理的。把一个窗口同它的窗口函数联系起来的过程称为注册窗口类。注册窗口包括对窗口指定一个窗口函数(给出窗口函数的指针)以及设定窗口的光标、背景刷子等内容。一个注册窗口类可以被多个窗口共享。注册窗口通过调用 API 函数 RegisterClass 来完成。

在 MFC 下，框架提供了缺省的自动窗口注册过程。框架仍然使用传统的注册类，而且提供了几个标准的注册类，它们在标准的应用程序初始化函数中注册。调用 AfxRegisterWndClass 全局函数就可以注册附加的窗口类，然后把已经注册的类传给 CWnd 的 Create 成员函数。用户可以定制自己的注册过程，以提供一些附加的特性。比如设置窗口的图标、背景、光标等。下面是注册窗口的例子。

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
// TODO: Modify the Window class or styles here by modifying
// the CREATESTRUCT cs
UINT ClassStyle=CS_VREDRAW|CS_HREDRAW;
cs.style=cs.style&(~FWS_ADDTOtitle);
cs.lpszClass = AfxRegisterWndClass(ClassStyle,
AfxGetApp()->LoadStandardCursor(IDC_ARROW),
(HBRUSH)(COLOR_WINDOW+1),//for brush
AfxGetApp()->LoadIcon(IDR_MAINFRAME));
return TRUE;
}
```

注册窗口在 CFrameWnd 的 PreCreateWnd 方法中完成。从成员函数名字 PreCreateWindow 中就可以看出来，注册窗口的工作必须在调用 Create 函数创建窗口之前完成。其中 cs.style=cs.style&(~FWS_ADDTOtitle) 指定窗口标题风格，关闭自动添加文档标题的功能。AfxRegisterWndClass 指定窗口使用箭头光标、背景刷子使用比窗口颜色标号大一的颜色、图标使用 IDR_MAINFRAME 标识符指定的图标(当然也可以使用其它图标)。用上面的程序段替换 Hello 程序 MainFrm.cpp 中的 PreCreateWindow 成员函数定义，并重新编译和运行程序。此时，窗口标题变成了 Hello，原来令人讨厌的“ Untitled- ”没有了，因为窗口风格中关闭自动添加当前文件名的风格。

3.5.4 关闭和销毁窗口

框架窗口不仅维护窗口的创建，还管理着窗口的关闭和销毁过程。关闭窗口时，操作系统依次向被关闭的窗口发送 WM_CLOSE 和 WM_DESTROY 消息。WM_CLOSE 消息的缺省处理函数 OnClose 将调用 DestroyWindow，来销毁窗口；最后，框架调用窗口的析构函数作清理工作并删除 C++窗口对象。

不要使用 C++的 delete 操作符来销毁框架窗口，而应当采用 CWnd

的 DestroyWindow 成员函数来销毁。DestroyWindow 首先删除子窗口，再删除窗口本身。若窗口以变量方式产生(即在堆栈上分配内存)，该窗口对象会被自动清除。若对象是用 new 操作符创建的(也就是在堆上分配内存的)，则需要用户自己处理。有关 DestroyWindow 问题在第五章对话框技术中还要作进一步解释。

OnClose()常用功能：保存窗口的一些状态、工具条状态，提示保存未保存的数据等等。

```
void CMainFrame::OnClose()
{
SaveBarState( “MyDockState” );//保存工具条状态
CFrameWnd::OnClose();
}
```

3.5.5 窗口激活

活动窗口必定是一个没有父窗口的顶层窗口，包括框架窗口和对话框。当顶层窗口被激活时，Windows 向窗口发送 WM_ACTIVATE 消息，对此消息的缺省处理是将活动窗口设为有输入焦点。

输入焦点用于表示哪个窗口有资格接收键盘输入消息。带有输入焦点的窗口或是一个活动窗口，或者是该活动窗口的子窗口。当一个顶层窗口获得输入焦点时，Windows 向该窗口发送 WM_SETFOCUS 消息，此窗口可将输入焦点重定位到它的子窗口上。子窗口不会自动获得输入焦点。失去输入焦点的窗口会收到 WM_KILLFOCUS 消息。当子窗口拥有输入焦点时，父窗口就不会处理键盘输入了。

3.6 使用菜单

现在我们要在主窗口中加入自己的菜单。菜单编程一般分三步：

1.编辑菜单资源，设置菜单属性(包括菜单名和 ID)；2.用 ClassWizard 自动映射菜单消息和成员函数；3.手工编辑成员函数，加入菜单消息处理代码。

3.6.1 编辑菜单资源

仍然使用我们前面生成的 Hello 程序，编辑由 AppWizard 自动生成的菜单资源。要编辑菜单资源：

(1) 选择项目工作区的 ResourceView 标签，切换到资源视图。

(2) 选择菜单资源类型。

(3) 选定菜单资源 IDR_MAINFRAME，双击该项或单击鼠标右键然后在弹出菜单中选择 Open 选项。Visual Studio 将弹出菜单编辑窗口，显示菜单资源 IDR_MAINFRAME，其中 IDR_MAINFRAME 是由 AppWizard 在创建该程序时自动生成的。

(4) 编辑当前菜单

要删除某个菜单项或弹出菜单，可用鼠标单击该菜单或用上下光标键来回选择，然后按 Del 键删除；要插入新菜单项，可选定窗口中的空白菜单框后按回车(或直接用鼠标双击该空白框)，Visual Studio 弹出 Properties(属性)对话框，如图 3-11 所示。属性对话框用于输入菜单项的标题、标识符、菜单项在状态栏上显示的提示(Prompt)，并为该菜单提供属性调整。也可以在选择一个已有的菜单项时按 Ins 键，以在该菜单项上方插入一个空白菜单项，然后双击该菜单项进行编辑。要插入一个分隔线，只需将菜单项的 Seperator 属性打开即可。

Visual Studio 支持鼠标拖曳调整菜单项位置。要调整菜单项位置，只需要选中某菜单项并将其拖至适当位置即可。

图 3.11 菜单编辑器和属性对话框

如上图，我们首先在 Edit 和 View 之间加入一个弹出菜单：用鼠标单击 View 菜单，按 Ins 键插入一个空白菜单项，双击该空白框弹出其属性对话框。在菜单属性栏输入“ &Test ”字样。然后在 Test 弹出菜单下加入以下四个菜单项，并在&Say Hello 菜单下加入一个分隔符。菜单项属性设置如下表：

菜单名	菜单 ID	菜单提示 (Prompt)
Say &Hello	ID_SAY_HELLO	Say hello to you!
&Red	ID_SELECT_RED	The color is red.
&Blue	ID_SELECT_BLUE	The color is blue.
&Yellow	ID_SELECT_YELLOW	The color is yellow.

提示：如果菜单中要使用中文，则除了在菜单名一项中输入中文外，还要将菜单资源的语言属性设置为中文。方法是：鼠标右键单击资源视图的菜单资源 IDR_MAINFRAME，弹出快捷菜单，选择 Properties，弹出整个菜单资源的属性对话框，如图 3-12 所示。在 Languages 下拉列表框中选择 Chinese (P.R.C.)。这样以后菜单就可以正确使用和显示中文了。如果其他资源如对话框或字符串要使用中文，也要将该资源的语言属性改为 Chinese(P.R.C)。/

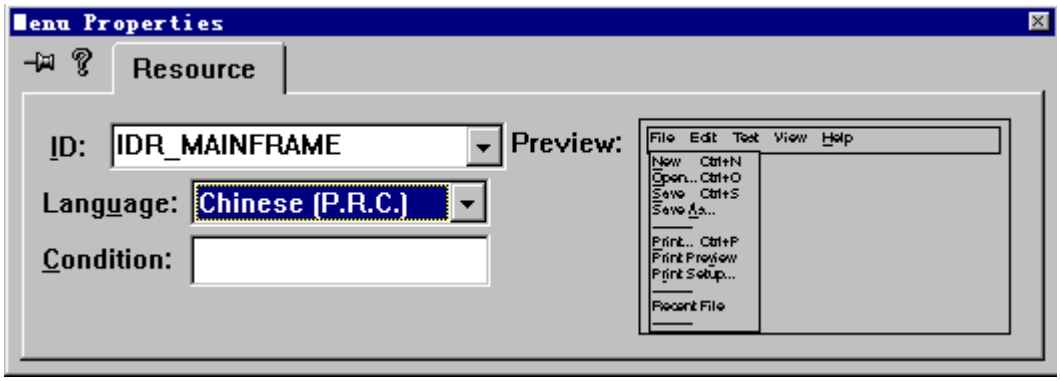


图 3.12 设置菜单语言属性

现在关闭菜单编辑器窗口。我们要为 Say Hello 菜单增加一个加速键 CTRL+H。要编辑加速键，选择 Accelerator 资源类型，双击打开 IDR_MAINFRAME 加速键资源。要删除加速键，可以直接按 Del 键。要增加加速键，可以按 Ins 键，弹出加速键属性对话框。在 ID 下拉列表框中选择 ID_SAY_HELLO，在 Key 一栏中输入 H，完成加速键设置。关闭加速键编辑窗口。

3.6.2 用 ClassWizard 自动映射菜单消息和成员函数

现在我们用 ClassWizard 为上面创建的几个菜单生成和映射消息处理成员函数。在此之前我们首先介绍一下 ClassWizard 的用法。

用 ClassWizard 管理类和 Windows 消息

ClassWizard 有助于创建 Windows 消息和命令处理函数、创建和管理类、创建类成员变量、创建 OLE Automation 的方法和属性、创建数据库类以及其他一些工作。

ClassWizard 也有助于覆盖 MFC 类中的虚函数。先选类，再选择需要覆盖的虚函数。该过程的其余部分与消息处理是类似的。

启动 ClassWizard 应用程序

从 View 菜单或源程序编辑窗口右键菜单中选择 ClassWizard(快捷键：Ctrl +W)，Developer Studio 将弹出 MFC ClassWizard 对话框。该对话框包含几个标签页，提供以下选项：

Message Maps：它管理消息和成员函数之间的映射关系。

Member Variables：它可以让用户加进一些数据成员，以便和各种控制进行数据交换。

Automations：它提供了各种特性支持 OLE2.0，包括为 OLE Automation 增加属性、方法以及处理事件。

ActiveX Events：为 ActiveX 控件增加属性、方法以及为 ActiveX 控件事件增加处理函数。

Class Info：它可以让用户创建新类，以便支持对话框和各种可视类(包括控制、窗口等)。还可以从类库文件导入类到当前工程中。

在这一章里，我们只用到 Message Maps 这一页，因此我们在这里只介绍 Message Maps 的使用。

Message Maps 选项

可以让用户加入成员函数来处理消息，删除成员函数以及对成员函数进行编辑。

Message Maps 页包括如下控制选项：

Projects 组合框：允许用户选择当前工作区中包含的工程。

Class Name 组合框：允许用户选择当前工程中的类。

Objects IDs 列表框：列出当前选中的类名及相关的 ID。对窗口和视来说，这些 ID 为菜单标识符；对对话框来说，这些 ID 为控制框的 ID。

Messages 列表框：列出当前所选类的可重载的虚方法以及可接收到的消息。

Member Functions 列表框：列出 ClassName 组合框中当前所选的项中所包含的所有成员函数。用户可以增加、删除成员函数，也可以对成员函数进行编辑。

Add Class...按钮：它允许用户往工程里添加新类。在按钮右边有一个向下的小箭头，表明按此按钮将弹出一个菜单。菜单包含两项：New...可以新建一个类；From a type Lib 用于从一个类库中导入类。

Add Function 按钮：它允许用户往 Member Functions 列表框中加进一个新的消息处理成员函数，该新增成员函数被用来响应 Message Maps 列表中当前所选中的消息。

Delete Function 按钮：用于删除 Member Functions 列表框中所选中的项。

Edit Code 按钮：它允许用户对 Member Functions 中所选中的项进行编辑，此时 Visual Studio 将关闭 MFC ClassWizard 对话框，并打开相应文件，并将光标定位在函数定义的开头处。

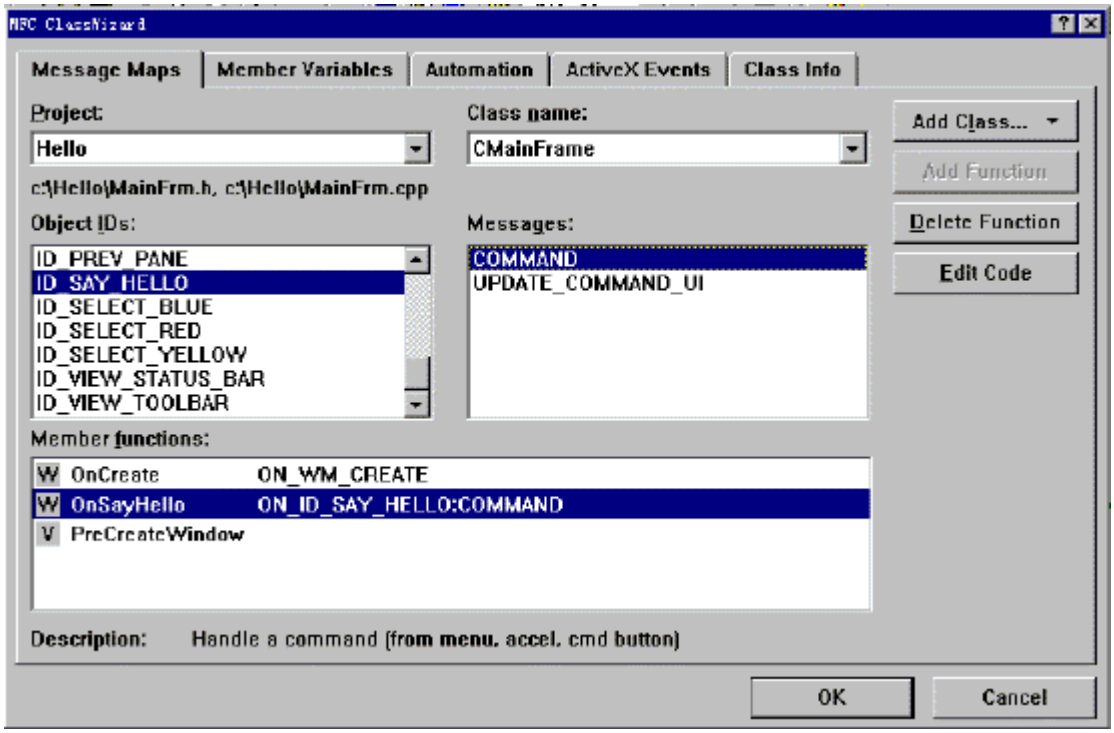


图 3-13 用 ClassWizard 增加菜单消息成员函数映射

现在我们就利用 ClassWizard 为 Hello 程序增加菜单消息和成员函数的映射。在 View 菜单下选择 ClassWizard，弹出 MFC ClassWizard 对话框。选择 Message Maps 页，在 Class Name 下拉列表中选择 CMainFrame 类。在 Object IDs 中选择 ID_SAY_HELLO，在 Messages 栏中双击 COMMAND，弹出 Add Member Function 对话框。对话框中给出缺省的成员函数 OnSayHello，按 OK 接收缺省的成员函数名。此时 OnSayHello 成员函数就出现在 Member Functions 列表框中，后面跟所映射的消息，如图 3-13 所示。列表框中开头的字母 W 表示窗口消息，V 表示可重载的虚方法 (Virtual Method)。如此，依次为 ID_SELECT_BLUE、ID_SELECT_RED、ID_SELECT_YELLOW 增加消息处理成员函数 OnSelectBlue、OnSelectRed、OnSelectYellow。然后双击 Member Functions 列表中的 OnSayHello，编辑 OnSayHello 成员函数。

3.6.3 手工添加代码

在 OnSayHello 成员函数体中加入语句。

```
void CMainFrame::OnSayHello()
{
AfxMessageBox(“Hello!”);
}
```

消息框函数

AfxMessageBox 用来弹出一个消息框，它的函数原型有两种版本：

```
int AfxMessageBox( LPCTSTR lpszText, UINT nType = MB_OK,
```

UINT nIDHelp = 0);

int AFXAPI AfxMessageBox(UINT nIDPrompt, UINT nType = MB_OK, UINT nIDHelp = (UINT) －1);

在第一种形式中，lpszText **表示在消息框内部显示的文本，消息框的标题为应用程序的可执行文件名(如** Hello)**。在第二种形式中，**nIDPrompt **为要显示的文本字符串在字符串表中的** ID**。函数调用时会自动从字符串表中载入字符串并显示在消息框中。**nType **为消息框中显示的按钮风格和图标风格的组合，可以采用|(或)操作符组合各种风格。**

按钮风格

MB_ABORTRETRYIGNORE **消息框中显示** Abort、Retry、Ignore **按钮**

MB_OK **显示** OK **按钮**

MB_OKCANCEL **显示** OK、Cancel **按钮**

MB_RETRYCANCEL **显示** Retry、Cancel **按钮**

MB_YESNO **显示** Yes、No **按钮**

MB_YESNOCANCEL **显示** Yes、No、Cancel **按钮**

图标风格

MB_ICONINFORMATION **显示一个** i **图标，表示提示**

MB_ICONEXCLAMATION **显示一个惊叹号，表示警告**

MB_ICONSTOP **显示手形图标，表示警告或严重错误**

MB_ICONQUESTION **显示问号图标，表示疑问**

比如，要在消息框中显示一个问号、一个“ Yes **”按钮、一个“** No **”按钮，可以以下面的方式调用** AfxMessageBox**。**

AfxMessageBox(“ Are you sure” ”，MB_YESNO|MB_ICONQUESTION)**；**

还有一个与 AfxMessageBox **类似的函数** MessageBox**，它是** CWnd **的类成员函数：**

int MessageBox(LPCTSTR lpszText,LPCTSTR lpszCaption = NULL,

UINT nType = MB_OK);

与 AfxMessageBox **不同的是，它多了一个** lpszCaption **参数，从名字上就可以推断出它表示消息框的标题，这样就可以设置消息框的标题，而不必采用可执行文件名作为标题了。**

两个函数的区别：AfxMessageBox **比** MessageBox **简单一些，因为它是一个全局函数所以不需要对应的一个窗口类，但是不能控制消息框标题，常用于调试程序时的内部数据输出或警告；**MessageBox **比较正式，常用在要提交的应用程序版本中，可以控制标题内容而不必采用含义不明的可执行文件名为标题。**

现在再来编写 OnSelectRed、OnSelectBlue、OnSelectYellow **三个函数。首先我们双击** CMainFrame **类名，在** MainFrm.h **中加入数据成员，如下所示：**

class CMainFrame:public CFrameWnd{

...

```
// Attributes
public:
int m_nColor;
enum{RED=0,BLUE=1,YELLOW=2};
...
}
```

注意我们这里使用了匈牙利命名法，建议读者也采用这种命名方法，以便提高程序可读性。加入数据成员后，还要对它进行初始化，初始化工作在 CMainFrame()**构造函数中完成。**

```
CMainFrame::CMainFrame()
{
m_nColor=RED;
}
```

OnSelectRed、OnSelectBlue、OnSelectYellow **三个函数修改后如清单** 3.4**：**

清单 3.4

```
void CMainFrame()::OnSelectRed()
{
m_nColor=RED;
SayColor();
}
void CMainFrame()::OnSelectBlue()
{
m_nColor=BLUE;
SayColor();
}
void CMainFrame()::OnSelectYellow()
{
m_nColor=YELLOW;
SayColor();
}
```

然后在 MainFrm.h **中加入** SayColor()**函数的声明：**

```
//Operations
public:
void SayColor();
```

在 MainFrm.cpp **中，在** OnSelectYellow()**成员函数后面，手工加入** SayColor()**函数的定义。**

```
void CMainFrame::SayColor()
{
switch(m_nColor)
{
case RED:
```

```
AfxMessageBox(“Color is red!”);
break;
case BLUE:
AfxMessageBox(“Color is blue!”);
break;
case YELLOW:
AfxMessageBox(“Color is yellow!”);
break;
}
}
```

这样，当我们选择颜色时，就会显示不同的消息框显示当前选择的颜色。但是用消息框显示当前选中的颜色似乎太繁琐了一些。我们在使用 WORD **编写文档时，注意到在选择不同的视图时，在视图菜单名前显示一个点，表明这是当前选择的视图。我们是否也可以这么做”回答是肯定的。要实现这一功能，就要使用** MFC **框架的更新命令用户接口消息机制。**

3.7 更新命令用户接口(UI)消息

一般情况下，菜单项和工具条按钮都不止一种状态，我们经常需要根据应用的内部状态来对菜单项和工具条按钮作相应的改变。例如，在我们没有选择任何内容时，编辑菜单下的“复制”、“剪切”等菜单是无效的(灰色显示)。有时，我们还会看到，在菜单项旁边可能还会有检查标记，表示它是选中的还是不选中的。比如，在 Word 的视图菜单下，根据用户所选的显示模式，在“普通”、“大纲”、“页面”“主控文档”前会出现一个点符号，标识当前所选的视图方式。工具条也有类似的情形，如果按钮不可用也可以被置成无效，或者可以被选中。

如果我们采用 SDK 来编程，那么我们就要跟踪与这些状态相关的变量所有可能发生变化的地方，并根据可能发生的变化作相应的处理。这样的工作非常复杂且容易遗漏。为此，MFC 应用程序框架引入了更新命令用户接口消息来简化这一工作。

在 ClassWizard 的 Message Map 页中，如果我们选择一个菜单 ID，在 Messages 列表框中就会出现两项：

COMMAND

UPDATE_COMMAND_UI

其中 UPDATE_COMMAND_UI 就是更新命令用户接口消息，专门用于处理菜单项和工具条按钮的更新。每一个菜单命令都对应于一个更新命令用户接口消息。可以为更新命令用户接口消息编写消息处理函数来处理用户接口(包括菜单和工具条按钮)的更新。如果一条命令有多个用户接口对象(比如一个菜单项和一个工具条按钮)，两者都被发送给同一个处理函数。这样，对于所有等价的用户接口对象来说，可以把用户接口更新代码封装在同一地方。

3.7.1 用户接口更新原理

为了理解用户接口更新机制，我们来看一下应用框架是如何实现用户接口更新的。当我们选择 Edit 菜单时，将产生一条 WM_INITMENUPOPUP 消息。框架的更新机制将在菜单拉下之前集体更新所有的项，然后再显示该菜单。

为了更新所有的菜单项，应用框架按标准的命令发送路线把该弹出式菜单中的所有菜单项的更新命令都发送出去。通过匹配命令和适当的消息映射条目(形式为 ON_UPDATE_COMMAND_UI)，并调用相应的更新处理器函数，就可以更新任何菜单项。比如，Edit 菜单下有 Undo、Cut、Copy、Paste 等四个菜单项，就要发送四条用户接口更新命令。如果菜单项的命令 ID 有一个更新处理器，它就会被调用进行更新；如果不存在，则框架检查该命令 ID 的处理函数是否存在，并根据需要使菜单有效或无效。

如果在命令发送期间找不到对应于该命令的 ON_UPDATE_COMMAND_UI 项，那么框架就检查是否存在一个命令的 ON_COMMAND 项，如果存在，则使该菜单有效，否则就使该菜单无效(灰化)。这种更新机制仅适用于弹出式菜单，对于顶层菜单象 File 和 Edit

菜单，就不能使用这种更新机制。

按钮的命令更新机制与菜单的命令接口更新机制类似，只是工具条按钮的命令接口更新在空闲循环时完成。有关工具条按钮的接口更新机制，我们将在下一章“工具条和状态栏”中作详细介绍。

3.7.2 用户接口更新机制编程

当框架给处理函数发送更新命令时，它给处理函数传递一个指向 CCmdUI 对象的指针。这个对象包含了相应的菜单项或工具条按钮的指针。更新处理函数利用该指针调用菜单项或工具条的命令接口函数来更新用户接口对象(包括灰化，使，使能，选中菜单项和工具条按钮等)。下面我们使用前面的例子演示如何使用用户接口更新机制：

1.按 Ctrl+W 激活 ClassWizard，选择 Message Map 选项页。

2.在 Object IDs 列表中选择 ID_SELECT_RED，在 Messages 列表中双击 ON_UPDATE _COMMAND_UI 条目，弹出 Add Member Function 对话框，缺省函数名为 OnUpdateSelect Red，按 OK 按钮接收此函数名。OnUpdateSelectRed 成员函数名就出现在 Member Functions 列表中。依次给 ID_SELECT_BLUE、ID_SELECT_YELLOW 增加 OnUpdateSelectBlue 和 OnUpdateSelectYellow 命令接口更新成员函数。

3.现在手工编辑刚才生成的成员函数，修改后形式如清单 3.5 所示：

清单 3.5

```
void CMainFrame::OnUpdateSelectBlue(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nColor==BLUE);
}
void CMainFrame::OnUpdateSelectRed(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nColor==RED);
}
void CMainFrame::OnUpdateSelectYellow(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_nColor==YELLOW);
}
```

现在编译运行 Hello 程序。当我们打开 Test 菜单时，如图 3-14 所示，在 Red 菜单项前已经有了一个对号。因为前面在 CMainFrame 构造函数中，我们已经将颜色初始化为 RED。如果选择 Blue，下次打开 Test 菜单时，在 Blue 菜单前就会有一个对号，而 Red 前面的对号则没有了。

图 3-14

类似的，要根据某个状态开关菜单，也可以为菜单生成命令接口更新成员函数。比如，在 Edit **菜单中，如果当前剪贴板没有内容，**Paste(**粘贴)菜单应当设为无效，程序可以这么写：**

```
void CMainFrame::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
pCmdUI->Enable(!IsClipboardEmpty());
}
```

其中 IsClipboardEmtpy()**是读者自己编写的函数，用于判断剪贴板中是否有内容**

3.8 快捷菜单

Windows95 和 Windows 3.x 在界面上的一个重大差别就是 Windows95 增加了功能强大的右键快捷菜单。在任何一个对象上按鼠标右键，就会弹出一个与所选当前对象相关的菜单，菜单中列出了一组针对当前对象的操作。在 Visual Studio 中就有大量这样的菜单。比如，在项目工作区中单击右键时弹出菜单，让用户选择 Docking View(停泊视图)、Hide(隐藏)和 Properties(属性)操作。现在我们来讨论如何使用 Visual C++为应用程序增加右键菜单。

这里我们也不是手工编程，而是使用 Visual Studio 提供的构件工具 Component Gallery(组件画廊)向框架窗口添加快捷菜单。有关 Componet Gallery 的概念参见第二课 2.1.5 节。选择 Project->Add to Project->Component and Controls 菜单，弹出 Component and Controls Gallery 对话框，选择 Developer Studio Components 目录，在该目录下选择 Popup Menu 构件，如图 3-15 所示。

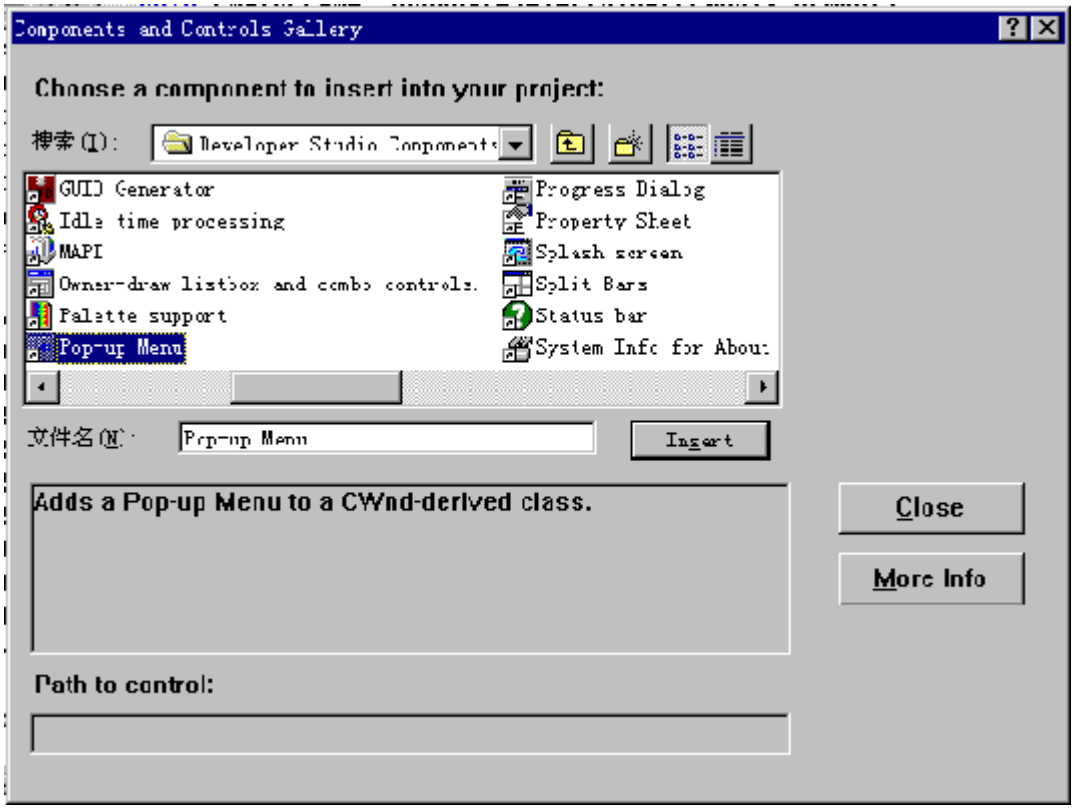


图 3-15

按 Insert 按钮。弹出 Poup Menu 对话框，在 Add popup menu to 下拉列表框中选择 CMainFrame，点 OK 按钮，关闭 Popup Menu 对话框。按 Close 按钮关闭 Component and Controls Gallery 对话框。编译运行 Hello，弹出窗口后按右键，就弹出如图 3-16 所示的快捷菜单。菜单中包含三项：cut、copy、paste。因为没有对应的消息矗立函数，所有这些菜单都是灰色的、非活动的。

图 3-16

现在，我们看看 Component Gallery 是如何实现快捷菜单的。首先看资源视图的菜单资源，Component Gallery 在其中增加了一个 ID 为 CG_IDR_POPUP_MAIN_FRAME 的菜单，菜单中包含了刚才我们看到的三个菜单项：cut、copy、paste。切换到类视图，浏览 CMainFrame 类，可以看到 CMainFrame 增加了一个 OnContextMenu 的成员函数，它是 CWnd 的一个方法，用于处理鼠标右键单击消息，原型如下：

```
afx_msg void OnContextMenu(CWnd* pWnd,CPoint point);
```

其中 pWnd 指向右键单击的窗口，它可以是一个本窗口的一个子窗口。比如，我们在工具条上单击右键时也弹出同样的菜单，工具条就是框架窗口的一个子窗口。OnContextMenu 函数定义如清单 3.6 所示。

清单 3.6 右键菜单

```
void CMainFrame::OnContextMenu(CWnd*, CPoint point)
{
// CG: This block was added by the Pop-up Menu component
{
if (point.x == -1 && point.y == -1){
//如果是键盘激活的快捷菜单，则在窗口左上角 5,5 的位置显示快捷菜单
CRect rect;
GetClientRect(rect);
ClientToScreen(rect);
point = rect.TopLeft();
point.Offset(5, 5);
}
//载入快捷菜单资源
CMenu menu;
VERIFY(menu.LoadMenu(CG_IDR_POPUP_MAIN_FRAME));
//取得本菜单的第一个子菜单
CMenu* pPopup = menu.GetSubMenu(0);
ASSERT(pPopup != NULL);
CWnd* pWndPopupOwner = this;
//如果当前窗口是一个子窗口，取其父窗口作为弹出菜单的拥有者
while (pWndPopupOwner->GetStyle() & WS_CHILD)
pWndPopupOwner = pWndPopupOwner->GetParent();
//在 point.x，point.y 处显示弹出式菜单并跟踪其选择项
pPopup->TrackPopupMenu(TPM_LEFTALIGN |
TPM_RIGHTBUTTON, point.x, point.y,pWndPopupOwner);
}
```

}

一般的，我们都可以使用 Component Gallery 的 Popup menu 构件为某个窗口、对话框、视图等增加快捷菜单而无需手工编程。我们要做的只是编辑修改缺省的菜单为我们自己的快捷菜单，并用 ClassWizard 生成必要的成员函数，在加入自己的代码。如果确实要手工做的话，首先应当用菜单编辑器增加一个菜单，然后为对应的窗口添加 OnContextMenu 方法，OnContextMenu 的定义可以参考上面的程序。

Component Gallery 的功能远不止向程序添加快捷菜单这一项，它还可以增加启动画面(Splash Window)、多页式对话框、口令检查对话框等多种功能。读者可以试着往 Hello 程序中添加 Splash Window 和口令对话框，体验一下 Component Gallery 的强大功能。

小结

在这一章里，我们主要向读者介绍：

如何使用 AppWizard 生成 Hello 框架程序，并手工修改代码，让窗口显示“ Hello,World !”，并介绍了 AppWizard 所生成的文件。

应用程序的执行机制：框架调用缺省的 WinMain 函数，首先执行 InitInstance 初始化应用程序类的一个实例，然后调用 Run 进入消息循环。

框架窗口的使用：包括窗口的创建、注册类、窗口的关闭和撤消。

在窗口中加入菜单，分为三步工作：第一步，用菜单编辑器编辑菜单；第二步，用 ClassWizard 生成消息处理函数；第三步，手工编辑消息处理函数，完成特定的功能。

用户接口更新消息：接口更新机制原理，如何控制菜单的使能、灰化、选中。

快捷菜单编程：使用 Component Gallery 给应用程序添加快捷菜单。

第四课　工具条和状态栏

在上一课中，同学们已经学到了一些基本的界面设计技术。这一课将指导大家如何设计实现工具条和状态栏，并进一步加深对消息驱动机制的理解。

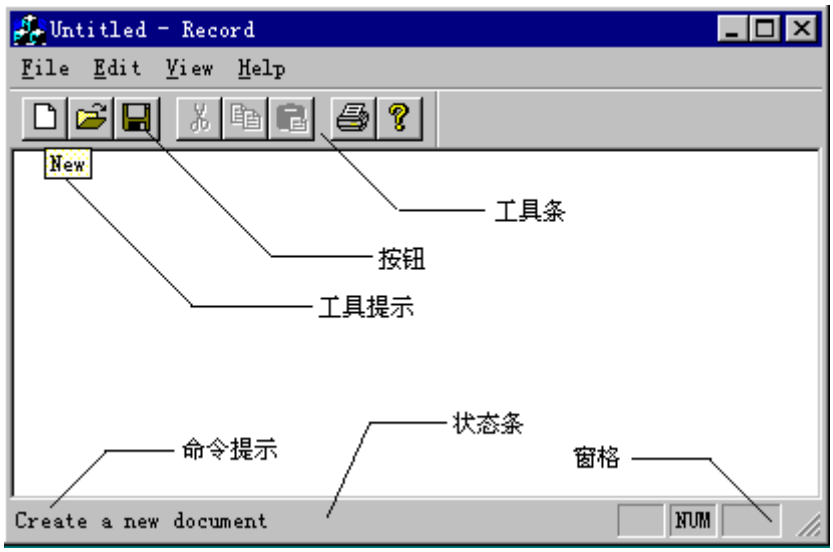


图 4.1 工具条和状态栏

如图 4.1 所示，工具条和状态栏是一个完善的 Windows 应用程序的重要组成部分(但不是必需的部分)。工具条一般位于主框架窗口的上部，上面有一些图形按钮。当用户用鼠标在某一按钮上单击时，程序就会执行相应的命令；当鼠标在按钮上停留片刻后，就会弹出一个黄色小窗口并显示工具提示。按钮的图形是它所代表功能的形象表示，人们对于形象图形的辨别速度要快于抽象文字，因此工具条提供了一种比菜单更快捷的用户接口。在一个标准的 Windows 应用程序中，工具条的大部分按钮执行的命令与菜单命令相同，这样做的目的是能同时提供形象和抽象的用户接口，以方便用户的使用。

状态栏位于主框架窗口的底部，主要用来显示一些提示信息，可细分为几个窗格。状态栏的主要功能是简要解释被选中菜单命令或工具条按钮命令，并显示 SCROLL LOCK、NUM LOCK 等键的状态。

这一课包括以下几个部分：

工具条的可视化设计

工具条的编程技术

状态栏的设计与实现

本课假定读者要编写一个采样声音的应用程序，并给出一个名为 Record 的具体例子。当然，这个例子不会真的具有采样声音的功能，它只是用来演示工具条和状态栏的使用。

4.1 工具条的可视化设计

从 4.0 版开始，Visual C++支持一种新型的工具条资源，这使得工具条的创建比以往更加方便灵活了。在 MFC 中，工具条的功能由类 CToolBar 实现。工具条资源和工具条类 CToolBar 是工具条的两个要素。创建工具条的基本步骤是：

1.创建工具条资源。

2.构建一个 CToolBar 对象。

3.调用 CToolBar::Create 函数创建工具条窗口。

4.调用 CToolBar::LoadToolBar 载入工具条资源。

使用缺省配置时，AppWizard 会自动创建一个工具条。如图 4.1 所示，这个工具条包含一些常用按钮，如打开文件、存盘、打印等等。用户可以修改这个工具条，去掉无用的按钮，加入自己需要的按钮。如果用户需要创建两个以上的工具条，则不能完全依赖 AppWizard，需要自己手工创建之。本节将分别讨论这两种方法。

4.1.1 利用 AppWizard 自动创建

自动创建工具条很简单，请读者按以下步骤操作：

选择 File->New 命令。

在弹出的标签式对话框中选 Projects 页，然后在该页中选中 MFC AppWizard (exe)项，并在 Project name 一栏中输入 Record 以创建一个名为 Record 的工程。按回车或用鼠标点击 Create 按钮后就进入了 MFC AppWizard 对话框。

在 MFC AppWizard 对话框的第一步中选中 Single document。这样就会创建一个单文档应用程序，若选择 Multiple documents 项，则将创建一个多文档应用程序。单文档程序一次只能打开一个窗口，显示一个文档的内容，而多文档程序一次可以打开多个窗口，显示多个文档的内容。

用鼠标点击 Finish 按钮，并在接着的对话框中按 OK 按钮。

完成以上操作后，工程 Record 被创建并被自动载入 Developer Studio 中。将项目工作区切换到资源视图，并展开资源，就会发现其中有一个名为 IDR_MAINFRAME 的 Toolbar(工具条)资源。用鼠标双击“IDR_MAINFRAME”，Developer Studio 会打开一个功能强大的工具条资源编辑窗口，如图 4.2 所示。该窗口的上部显示出了工具条上的按钮，当用户用鼠标选择某一按钮时，在窗口的下部会显示该按钮的位图。在窗口旁边有一个绘图工具面板和一个颜色面板，供用户编辑按钮位图时使用。

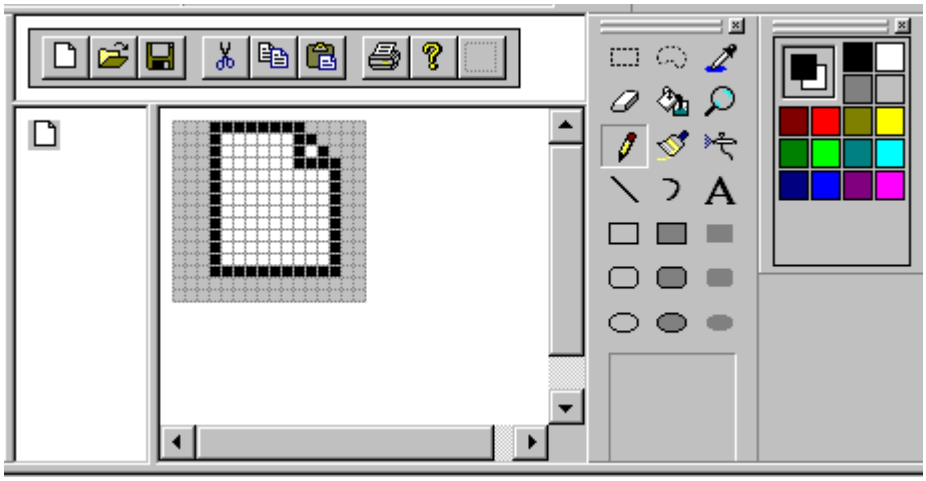


图 4.2 工具条资源编辑窗口

提示：如果读者看不到这两个面板，请在 Developer Studio 的工具条的空白处单击鼠标右键，并在随之弹出的菜单中选中 Graphics 和 Colors 两项。/

在修改工具条以前，首先要修改菜单资源。请按以下几步修改菜单资源:

将项目工作区切换至资源视图，选择并打开 menu(菜单)资源类型，双击名为 IDR_MAINFRAME 的菜单资源。

删除 Edit 菜单。

删除 File 菜单中除 Exit 以外的所有菜单项。

在 File 菜单后插入一个名为&Record 的新菜单，并在该菜单中插入&Start 和 St&op 两个菜单项，它们的命令 ID(标识符)分别为 ID_RECORD_START 和 ID_RECORD_STOP。Start 表示开始录音，而 Stop 表示停止录音。

修改后的菜单如图 4.3 所示。

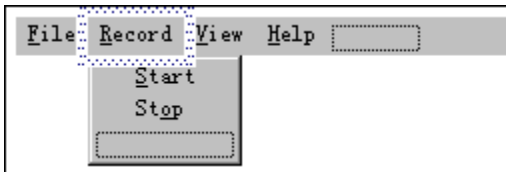


图 4.3 修改后的菜单

接下来的任务是修改工具条资源，具体步骤是：

选择并打开 Toolbar(工具条)资源类型，双击名为 IDR_MAINFRAME 的工具条资源以打开相应的资源编辑窗口。

删除“ ”按钮前面的所有按钮，删除的方法是用鼠标将要删除的按钮拖出工具条即可。

先选中“ ”按钮后面的空白按钮，然后在该按钮的放大位图上用红色画一个实心圆圈，以表示开始录音功能。再选中空白按钮，并用黑色在放大位图上画一个实心矩形，以表示停止功能。

通过用鼠标拖动按钮调整按钮的位置，调整后的位置如图 4.4 所示。

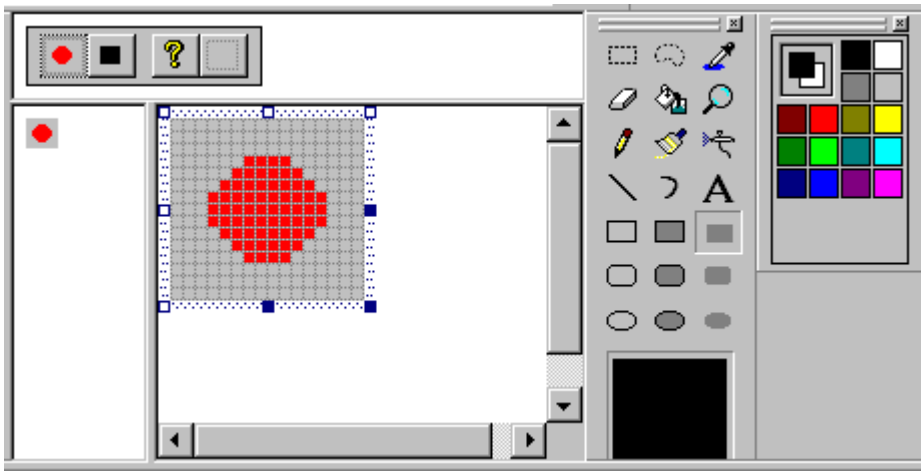


图 4.4 修改后的工具条资源

分别为两个新加的按钮指定命令 ID 为 ID_RECORD_START 和 ID_RECORD_STOP。指定 ID 的方法是先选中一个按钮，接着按回车键，在弹出的属性对话框中输入 ID(或从 ID 下拉列表中分别选择 ID_RECORD_START 和 ID_RECORD_STOP)。注意到这两个按钮的 ID 与 Record 菜单中的两个菜单项 Start 和 Stop 的 ID 相同，这样同样的命令既可以通过菜单执行，也可以通过工具条执行。

为两个新加的按钮指定命令提示。请分别在两个按钮的属性对话框中的 Prompt 栏内输入 Start record\nStart 和 Stop record\nStop。命令提示实际上是作为字符串保存在 String Table 字符串资源中的。命令提示用来解释命令的意义，分状态栏提示和工具提示两种，在 Prompt 栏中，二者由 \n 分隔开。当鼠标移动到某个菜单项或工具条上的按钮时，在状态栏中就会显示状态栏提示，当鼠标在某个按钮上停留片刻后，工具提示就会在一个黄色的弹出式窗口中显示出来。输入完成后，读者会发现 Record 菜单中的两个菜单项被自动加入了相同的提示信息，这说明两个按钮与两个菜单项确实是相对应的。

提示:如果觉得按钮太小，读者可以用鼠标拖动围绕按钮放大位图的虚框的右下角，把按钮放大些。注意工具条内的所有按钮都将被放大/

修改完后，读者可以编译并运行 Record，来看看修改的结果。读者很快会注意到 Start 和 Stop 菜单项及按钮都是灰色的。这个现象是正常的，其原因将在 4.2 节解释。有趣的是工具条可以被拖动(请在工具条的空白地方拖动)并停泊在主框架窗口的任何其它边上，并且工具条是可以浮动的，即当用鼠标双击工具条的空白处时，工具条变成了一个浮动窗口，可被拖动到屏幕上的任意地方。这些有趣的现象将在 4.2 节解释。不管怎么说，创建和修改工具条的任务已经完成了。

4.1.2 手工创建

如果想要再加一个工具条，那么 AppWizard 就无能为力了，必须手工创建。假设 Record 程序的声音采样频率有 11KHZ 和 44KHZ 两档选择，现在我们的任务是再创建一个工具条，可让用户对这两种档次进行选择。本来这样的功能应该位于第一个工具条内，但为了演示工具条的手工创建，这里不妨来个多此一举。

如果 Record 工程不在 Developer Studio 中，请选择命令 File->Open Workspace 打开 Record 工程。首先要对原来的菜单进行修改，步骤如下：

打开 IDR_MAINFRAME 菜单资源

双击 Record 菜单底端的空白项，在其属性窗口中选中 Separator，这样就加入了一条分隔线。

在分隔线下面加入两个菜单项，其属性如表 4.1 所示。

表 4.1 菜单项的属性

Caption	ID	Prompt
&Low quality	ID_LOW_QUALITY	Low quality(11k)/n11k
&High quality	ID_HIGH_QUALITY	High quality(44k)/n44k

接着要创建一个新的工具条资源，请按以下步骤进行：

选择 Insert->Resource 命令，然后在 Insert Resource 对话框中选中 Toolbar。按了 OK 按钮后，在 Toolbar 资源类下就会出现一个 ID 为 IDR_TOOLBAR1 的新资源。

在新工具条中加入两个按钮，如图 4.5 所示。每个按钮上都画了一些竖线，线稀的按钮代表低频率采样，线密的按钮代表高频率采样。

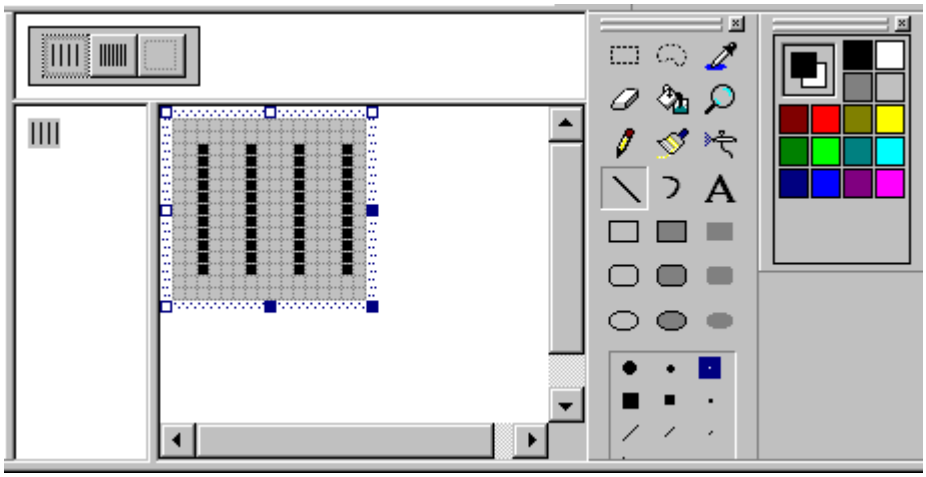


图 4.5 新创建的工具条资源

分别为两个新加的按钮指定命令 ID 为 ID_LOW_QUALITY 和 ID_HIGH_QUALITY。

要把这个新设计的工具条加入到程序中，需要在程序中加入一些与创建有关的源代码。在创建第一个工具条时，AppWizard 在程序中自动加入了创建源代码，通过仿制这些代码，我们很容易创建出第二个工具条。

在 MFC 中，工具条的功能由类 CToolBar 实现。工具条实际上是主框架窗口的子窗口，因此工具条对象应该依附于主框架窗口对象。在 AppWizard 创建的 MFC 程序中，主框架窗口的类名是 CMainFrame，该类是 MFC 标准类 CFrameWnd 类的派生类。将项目工作区切换至类视图并展开 CMainFrame 类，读者会发现该类有一个名为 m_wndToolbar 的成员。双击该成员，则 Developer Studio 会自动打开类 CMainFrame 所在的头文件，并将光标停在对 m_wndToolbar 成员的定义处。

提示:在类视图中双击某一个类名，则该类所在的头文件会自动打开。若双击某一个类的成员，则会自动切换到对该成员的定义处。/

对 m_wndToolBar 的定义如下：

CToolBar m_wndToolBar;

由此可见 m_wndToolBar 是一个 CToolBar 对象，它是 CMainFrame 的成员。现在请紧接着该成员加入一个新的成员：

CToolBar m_wndToolBar1;

m_wndToolBar1 代表第二个工具条。读者不要以为给 CMainFrame 加入一个 CToolBar 对象就完事了。实际的创建工具条的工作不会在构造 CToolBar 对象时完成，只有调用了类 CToolBar 的一些成员函数后，创建工作才能结束。

对工具条的实际创建工作在 CMainFrame::OnCreate 函数中完成。OnCreate 函数是在创建窗口时被调用的，这时窗口的创建已部分完成，窗口对象的 m_hWnd 成员中存放的 HWND 句柄也已有效，但窗口还是不可见的。因此一般在 OnCreate 函数中作一些诸如创建子窗口的初始化工作。

提示:初学者一个易犯的错误是在构造函数而不是在 OnCreate 中创建子窗口。在构造函数中，父窗口并没有创建，如果在这时创建子窗口，则将会因为得不到父窗口的有效 HWND 句柄而导致创建失败。/

找到 CMainFrame::OnCreate 函数，对该函数进行一些修改，修改的部分如清单 4.1 的黑体字所示。在以后，凡是程序中手工修改的部分，一般都会用黑体显示。

清单 4.1 修改后的 CMainFrame::OnCreate 函数

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
return -1;
if (!m_wndToolBar.Create(this) ||
!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
TRACE0("Failed to create toolbar\n");
return -1; // fail to create
}
if (!m_wndToolBar1.Create(this) ||
!m_wndToolBar1.LoadToolBar(IDR_TOOLBAR1))
{
TRACE0("Failed to create toolbar\n");
return -1; // fail to create
}
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT)))
{
TRACE0("Failed to create status bar\n");
```

```
return -1; // fail to create
}
// TODO: Remove this if you don't want tool tips or a resizeable toolbar
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
m_wndToolBar1.SetBarStyle(m_wndToolBar1.GetBarStyle()|CBRS_T
OOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
// TODO: Delete these three lines if you don't want the toolbar to
// be dockable
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
m_wndToolBar1.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
DockControlBar(&m_wndToolBar1);
return 0;
}
```

注意在 CMainFrame::OnCreate **函数首先调用了** CFrameWnd::OnCreate**。这行代码是** AppWizard **自动加入的，但有必要解释一下。** CMainFrame **是** CFrameWnd **类的继承类，在** CMainFrame::OnCreate **中首先要调用基类** CFrameWnd **的** OnCreate **函数，因为基类也要进行一些初始化工作，而基类的** OnCreate **函数不会自动调用，因此需要在继承类的** OnCreate **函数中显式调用。**OnCreate **实际上是** WM_CREATE **消息的消息处理函数，读者可能要问，为什么是派生类的** OnCreate **处理** WM_CREATE **消息，而不是基类的** OnCreate **呢。如果读者仔细观察** OnCreate **函数在** CMainFrame **类头文件中的说明，就会发现在该函数前有一个** afx_msg **前缀。**afx_msg **的作用与** virtual **关键字类似，它使得被说明的函数有虚拟函数的特性，即由继承类而不是基类的处理函数来处理消息。**

读者可以看出黑体的代码是仿照第一个工具条的创建代码进行编写的。事实上，笔者不过是将原来的代码复制一份，然后在略作修改而已。

提示: **读者不必为这种“抄袭”行为感到羞愧。由于** Visual C++**博大精深，各种类和函数成百上千，除非你有外星人一般的记忆力，否则是不可能记住所有东西的。用** Visual C++**编程，重要的是理解而不是记忆。只要你理解了程序的来龙去脉，就可以最大限度的利用现有的成熟代码，提高程序的开发效率和可靠性。**/

对第二个工具条的创建代码的解释是：首先，调用 CToolBar::Create **以创建工具条窗口，注意** Create **函数的参数是** this **指针，这是因为主框架窗口是工具条的父窗口。接着调用** CToolbar::LoadToolBar(IDR_TOOLBAR1)**以载入工具条资源。然后调用** CToolBar::SetBarStyle **指定工具条的风格，在调用该函数时先调用** CToolBar::GetBarStyle **取得工具条的风格，然后在原有风格的基础上又指定了** CBRS_TOOLTIPS**、** CBRS_FLYBY **和** CBRS_SIZE_DYNAMIC **风**

格，这使得工具条可显示工具提示，并可以动态改变尺寸。接着调用 CToolBar::EnableDocking(CBRS_ALIGN_ANY)使工具条是可以停泊的，但还需调用 CFrameWnd::EnableDocking(CBRS_ALIGN_ANY)，只有这样才能实现可停泊的工具条。最后调用 CFrameWnd::DockControlBar 以停泊工具条。

编译并运行 Record 看看，现在 Record 程序已经拥有两个工具条了。至此创建工具条的任务已经完成，下面需要对工具条编程，以使其能够发挥执行命令的功能。

4.2 工具条的编程技术

本节将讨论一些与工具条有关的编程技术，主要包括命令处理、命令更新、按钮风格和工具条的隐藏/显示等技术。

4.2.1 命令处理

要使菜单和工具条执行命令，光为它们指定命令 ID 是不行的，必须为每个命令 ID 定义命令处理函数。如果不为命令定义命令处理函数或下面将要提到的命令更新处理函数，则框架将自动使该命令对应的菜单项和按钮禁止(灰化)，这就是 4.1 节中的工具条按钮和菜单项灰化的原因。

利用 ClassWizard 可以很方便地加入命令处理函数，请读者按以下步骤操作：

按 Ctrl+W 键进入 ClassWizard。

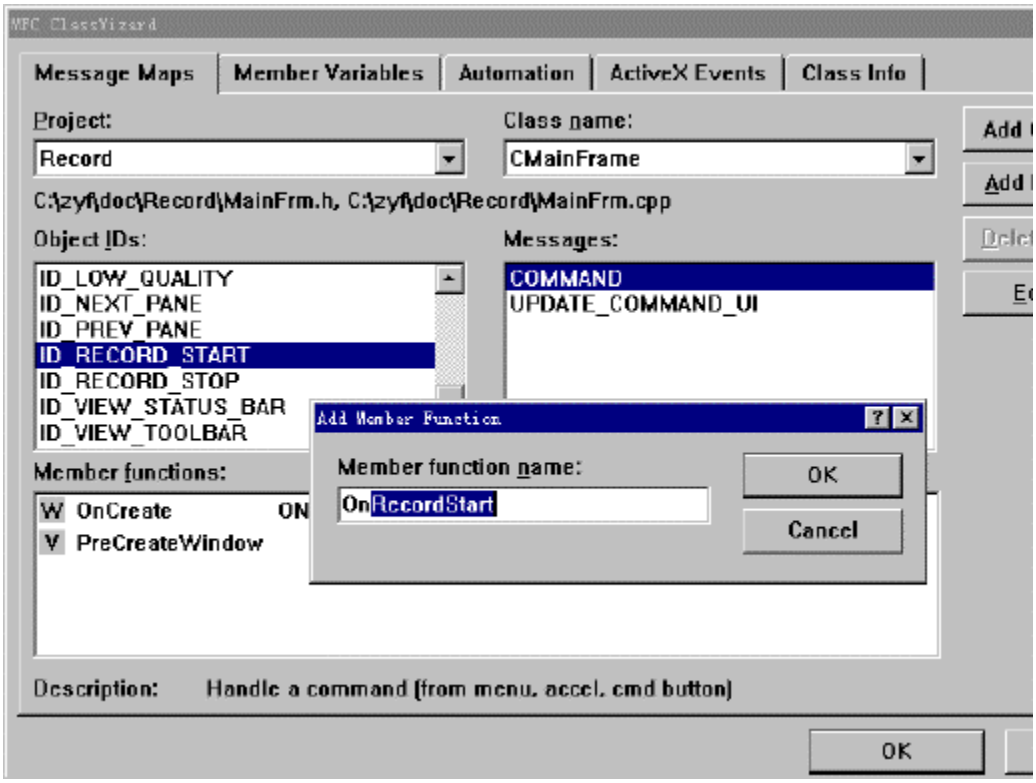


图 4.6 ClassWizard 对话框

如图 4.6 所示，在 Class name 栏中选择 CMainFrame，在 Object IDs 栏中选择 ID_RECORD_START，在 Messages 栏中双击 COMMAND 项，则 ClassWizard 会弹出一个对话框询问命令处理函数的名字，使用其提供的函数名即可。按 OK 按钮后，函数 OnRecordStart 就被加入到了 Member functions 栏中。

仿照第 2 步，为 ID_RECORD_STOP 定义一个命令处理函数。

按 OK 按钮关闭 ClassWizard 对话框。这时读者会发现 CMainFrame 类多了两个成员函数，OnRecordStart 和 OnRecordStop。

现在要在这两个命令处理函数中插入相应的源代码以实现其功能。当然，这里不会真的实现开始录音和停止录音的功能。我们只是让这两个函数发出一个声音，象征性地表示功能的执行，具体代码如清单 4.2

所示。

清单 4.2 OnRecordStart 和 OnRecordStop 函数

```
void CMainFrame::OnRecordStart()
{
// TODO: Add your command handler code here
MessageBeep((UINT)(-1));
}
void CMainFrame::OnRecordStop()
{
// TODO: Add your command handler code here
MessageBeep((UINT)(-1));
}
```

编译并运行 Record，可以看到 Start 和 Stop 命令已经可以执行了。

4.2.2 命令更新

虽然 Start 和 Stop 命令可以执行了，但是还有一个不足之处。在没有开始录音之前，Stop 命令应该是禁止的，也即对应的菜单项和按钮应是禁止的，这是因为此时没有必要执行该命令。录音开始后，Stop 命令应该允许，而 Start 命令则应变为禁止。我们可以利用 MFC 的命令更新机制实现此逻辑功能。

在菜单下拉之前，或在工具条按钮处在空闲循环期间，MFC 会发一个更新命令，这将导致命令更新处理函数的调用。命令更新处理函数可以根据情况，使用户接口对象(主要指菜单项和工具条按钮)允许或禁止。定义命令更新处理函数的方法如下：

按 Ctrl+W 键进入 ClassWizard。

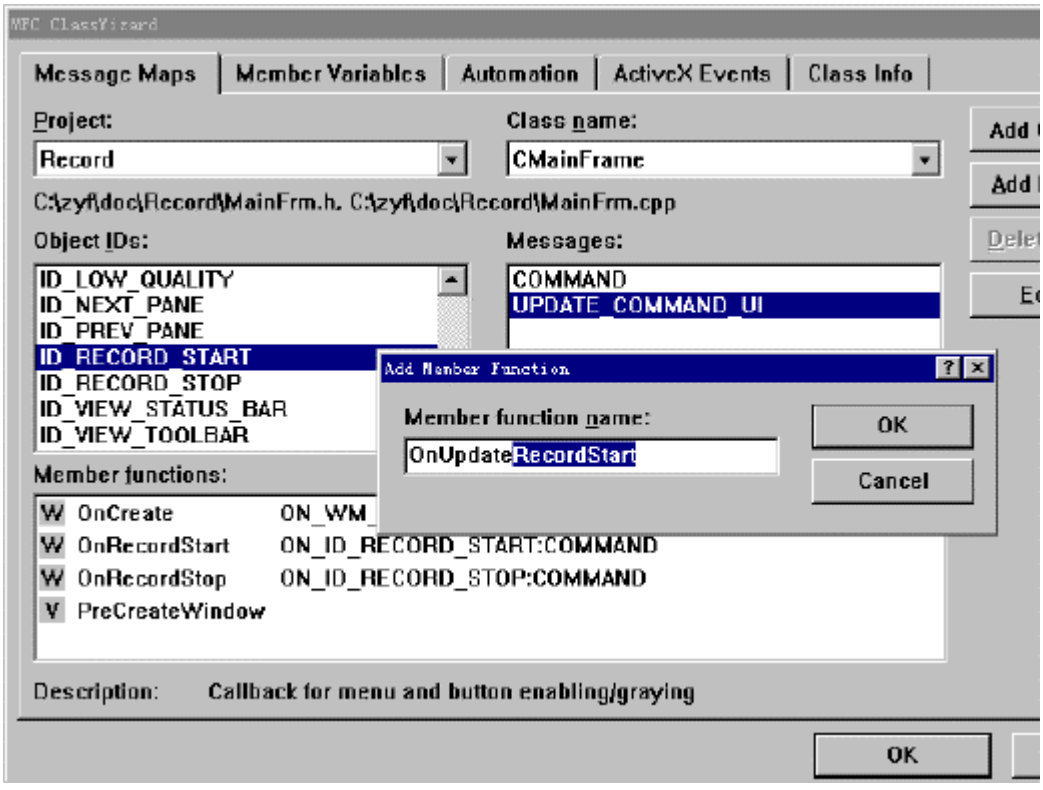


图 4.7 ClassWizard 对话框

如图 4.7 所示，在 Class name 栏中选择 CMainFrame，在 Object IDs 栏中选择 ID_RECORD_START，在 Messages 栏中双击 UPDATE_COMMAND_UI 项，则 ClassWizard 会弹出一个对话框询问命令更新处理函数的名字，使用其提供的函数名即可。按 OK 按钮后，函数 OnUpdateRecordStart 就被加入到了 Member functions 栏中。

仿照步 2，为 ID_RECORD_STOP 定义一个命令更新处理函数。

按 OK 按钮关闭 ClassWizard 对话框。这时读者会发现 CMainFrame 类多了两个成员函数，OnUpdateRecordStart 和 OnUpdateRecordStop。

命令更新处理函数有一个参数是 CCmdUI 类的指针，通过调用 CCmdUI 类的成员函数 Enable(TRUE)或 Enable(FALSE)可以使用户接口对象允许或禁止。需要给 CMainFrame 加一个布尔型成员变量以表明是否正在录音，这样命令更新处理函数可根据这个变量来决定用户接口对象的状态。请读者在 CMainFrame 类内加入下面一行代码：

```
BOOL m_bWorking;
```

接下来请读者按清单 4.3 进行修改。

清单 4.3 命令更新处理

```
CMainFrame::CMainFrame()
{
// TODO: add member initialization code here
m_bWorking=FALSE;
}
void CMainFrame::OnRecordStart()
{
// TODO: Add your command handler code here
MessageBeep((UINT)(-1));
m_bWorking=TRUE;
}
void CMainFrame::OnRecordStop()
{
// TODO: Add your command handler code here
MessageBeep((UINT)(-1));
m_bWorking=FALSE;
}
void CMainFrame::OnUpdateRecordStart(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(!m_bWorking);
}
void CMainFrame::OnUpdateRecordStop(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
```

```
pCmdUI->Enable(m_bWorking) ;
}
```

m_bWorking 的初值应是 FALSE，对它的初始化工作在 CMainFrame 的构造函数中完成。m_bWorking 的值在处理 Start 和 Stop 命令时会被更新以反映当前的状态。两个命令更新处理函数都调用了 CCmdUI::Enable，该函数根据 m_bWorking 的值来更新命令接口对象。

编译并运行 Record，现在 Start 和 Stop 命令的逻辑功能已经实现了。

4.2.3 按钮风格

在 Record 程序中，用户可以选择两种采样频率来录音。用户接口对象应该能反映出当前的采样频率。普通的工具条按钮在按下后会立刻弹起来，我们希望 Record 程序的频率选择按钮具有单选按钮的风格，即当用户选择了一个采样频率时，该采样频率对应的按钮一直处于按下的状态，而另一个频率选择按钮应处于弹起状态。

我们可以利用 CCmdUI::SetCheck 函数来实现这一功能，在命令更新函数中调用 CCmdUI::SetCheck(TRUE)或 CCmdUI::SetCheck(FALSE)可将用户接口对象设定为选中或不选中状态，当一个用户接口对象被选中时，相应的工具按钮会处于按下的状态，并且相应的菜单项的前面会加上一个选中标记。这里需要给 CMainFrame 类加一个布尔型成员变量以表明当前的采样频率。请读者在 CMainFrame 类内加入下面一行代码：

```
BOOL m_bHighQuality;
```

接下来请读者按清单 4.4 进行修改。

清单 4.4

```
CMainFrame::CMainFrame()
{
// TODO: add member initialization code here
m_bWorking=FALSE;
m_bHighQuality=TRUE;
}
void CMainFrame::OnHighQuality()
{
// TODO: Add your command handler code here
m_bHighQuality=TRUE;
}
void CMainFrame::OnLowQuality()
{
// TODO: Add your command handler code here
m_bHighQuality=FALSE;
}
void CMainFrame::OnUpdateHighQuality(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_bHighQuality);
```

```
}
void CMainFrame::OnUpdateLowQuality(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(!m_bHighQuality);
}
```

m_bHighQuality 的初值是 TRUE，即缺省时是高频采样，对它的初始化工作在 CMainFrame 的构造函数中完成。m_bHighQuality 的值在处理 High quality 和 Low quality 命令时会被更新以反映当前的状态。两个命令更新处理函数都调用了 CCmdUI::SetCheck，该函数根据 m_bHighQuality 的值来更新命令接口对象，从而使工具条按钮具有了单选按钮的风格。

编译并运行 Record，读者可以看到具有新风格的工具条按钮。当选择采样频率时，相应的菜单项前会出现一个选中标记，相应的工具条按钮会被按下。

4.2.4 工具条的隐藏/显示

读者可能已经试过了 Record 程序的 View 菜单的功能。通过该菜单用户可以隐藏/显示工具条和状态栏，这个功能是由 AppWizard 自动实现的。由于第二个工具条是手工建立的，因此它不会自动具备隐藏/显示功能。但我们可以通过编程来实现第二个工具条的隐藏/显示：

打开 IDR_MAINFRAME 菜单资源

在 View 菜单中加入一个名为 Toolbar1 的菜单项，指定其 ID 为 ID_VIEW_TOOLBAR1, 并在 Prompt 栏中输入 Show or hide the toolbar1\nToggle ToolBar1。

按 Ctrl+W 键进入 ClassWizard。在 Class name 栏中选择 CMainFrame，在 Object IDs 栏中选择 ID_VIEW_TOOLBAR1，并为该命令 ID 定义命令处理函数 OnViewToolbar1 和命令更新处理函数 OnUpdateViewToolbar1。

按清单 4.5 修改程序。

清单 4.5 显示/隐藏工具条

```
void CMainFrame::OnViewToolbar1()
{
// TODO: Add your command handler code here
m_wndToolBar1.ShowWindow(m_wndToolBar1.IsWindowVisible()”
SW_HIDE:SW_SHOW);
RecalcLayout();
}
void CMainFrame::OnUpdateViewToolbar1(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->SetCheck(m_wndToolBar1.IsWindowVisible());
}
```

调用 CWnd::ShowWindow(SW_SHOW) 或

CWnd::ShowWindow(SW_HIDE)可以显示或隐藏窗口。由于工具条也是窗口，CToolBar 是 CWnd 类的继承类，故该函数也是 CToolBar 的成员。在命令处理函数 OnViewToolbar1 中，我们调用 CToolBar::ShowWindow 来显示/隐藏工具条，在调用时会利用 CWnd::IsWindowVisible 函数作出判断，如果工具条是可见的，就传给 ShowWindow 函数 SW_HIDE 参数以隐藏工具条，否则，就传 SW_SHOW 参数显示工具条。接着要调用 CMainFrame::RecalcLayout 以重新调整主框架窗口的布局。

命令更新处理函数 OnUpdateViewToolbar1 会根据工具条是否可见使 View->Toolbar1 菜单项选中或不选中。

编译并运行 Record，现在 Record 程序已变得很有趣了。至此，读者已经掌握了工具条的一些实用编程技术。

4.3 状态栏的设计与实现

状态栏实际上是个窗口，一般分为几个窗格，每个窗格显示不同的信息。AppWizard 会为应用程序自动创建一个状态栏，该状态栏包括几个窗格，分别用来显示状态栏提示和 CAPS LOCK、NUM LOCK 、SCROLL LOCK 键的状态。在 MFC 中，状态栏的功能由 CStatusBar 类实现。

创建一个状态栏需要以下几个步骤：

构建一个 CStatusBar 对象。

调用 CStatusBar::Create 创建状态栏窗口。

调用 CStatusBar::SetIndicators 函数分配窗格，并将状态栏的每一个窗格与一个字符串 ID 相联系。

相应的代码读者可以在 Record 工程的 CMainFrame::OnCreate 成员函数中找到。如清单 4.6 所示。

清单 4.6 创建状态栏

```
…
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT)))
{
TRACE0(“Failed to create status bar\n”);
return -1; // fail to create
}
…
```

SetIndicators 函数的第一个参数 indicators 是一个 ID 数组，在 CMainFrame 类所在的 CPP 文件的开头部分可以找到该数组，如清单 4.7 所示。

清单 4.7 ID 数组

```
static UINT indicators[] =
{
ID_SEPARATOR, // status line indicator
ID_INDICATOR_CAPS,
ID_INDICATOR_NUM,
ID_INDICATOR_SCRL,
};
```

indicator 数组提供了状态栏窗格的分配信息，它的第一项一般为 ID_SEPARATOR，该 ID 对应的窗格用来显示命令提示信息，后三项都是字符串 ID，读者可以在 String Table 字符串资源中找到这三个字符串分别是 CAP、NUM 和 SCRL。它们对应的三个窗格用来显示键盘的状态。

现在让我们来给状态栏再加一个时间窗格，它将用来显示系统时间。显示的格式是 hh:mm:ss，即时:分:秒。

首先在 indicators 数组的 ID_SEPARATOR 项之后插入一个名为

ID_INDICATOR_CLOCK 的 ID。然后找到并双击名为 String Table 的字符串资源，打开字符串资源编辑窗口。接着在编辑窗口内按 Insert 键以插入一个新的字符串，请指定字符串的 ID 为 ID_INDICATOR_CLOCK，内容为 00:00:00。状态栏将根据字符串的长度来确定相应窗格的缺省宽度，所以指定为 00:00:00 就为时间的显示预留了空间。

提示:上述方法不能动态改变窗格宽度，并且有时是不精确的，当系统字体改变时，这种做法可能会导致一些误差。考虑到该方法简单直观，且一般情况下问题不大，故本文用它来举例。如果读者对动态、精确地指定窗格感兴趣，请参看 Visual C++ 5.0 随光盘提供的一个名为 NPP 的 MFC 例子(在 samples\mfc\general\npp 目录下)。/

时间窗格显示的时间必须每隔一秒钟更新一次。更新时间窗格的正文可调用 CStatusBar:: SetPaneText 函数，要定时更新，则应利用 WM_TIMER 消息。在 Windows 中用户可以安装一个或多个计时器，计时器每隔一定的时间间隔就会自动发出一个 WM_TIMER 消息，而这个时间间隔可由用户指定。MFC 的 Window 类提供了 WM_TIMER 消息处理函数 OnTimer，我们应在该函数内进行更新时间窗格的工作。

请读者利用 ClassWizard 给 CMainFrame 类加入 WM_TIMER 的消息处理函数 OnTimer 和 WM_CLOSE 消息的处理函数 OnClose，具体方法是在 Class name 栏中选择 CMainFrame，在 Object IDs 栏中选择 CMainFrame，在 Messages 栏中找到 WM_TIMER 和 WM_CLOSE 项，分别双击之然后按 OK 按钮退出 ClassWizard。CMainFrame::OnClose 函数是在关闭主框架窗口是被调用的，程序可以在该函数中做一些清除工作。

接下来请按清单 4.8 修改程序。

清单 4.8 CMainFrame 类的部分代码

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
…
SetTimer(1,1000,NULL);
return 0;
}
void CMainFrame::OnTimer(UINT nIDEvent)
{
// TODO: Add your message handler code here and/or call default
CTime time;
time=CTime::GetCurrentTime();
CString s=time.Format(“%H:%M:%S”);
m_wndStatusBar.SetPaneText(m_wndStatusBar.CommandToIndex(ID_I
NDICATOR_CLOCK),s);
CFrameWnd::OnTimer(nIDEvent);
}
void CMainFrame::OnClose()
{
```

```
// TODO: Add your message handler code here and/or call default
KillTimer(1);
CFrameWnd::OnClose();
}
```

在 CMainFrame::OnCreate **函数内调用了** CWnd::SetTimer **以安装一个计时器，** SetTimer **的第一个参数指定计时器** ID **为** 1**，第二个参数则规定了计时器的时间间隔为** 1000 **毫秒即** 1 **秒。这样，每隔** 1 **秒** OnTimer **函数就会被调用一次。**

在 OnTimer **函数中，首先构建了一个** CTime **对象，接着调用** CTime **的静态成员函数** GetCurrentTime **以获得当前的系统时间，然后利用** CTime::Format **函数返回一个按时**:**分**:**秒的格式表示的字符串，最后调用** CStatusBar::SetPaneText **来更新时间窗格显示的正文。**SetPaneText **的第一个参数是窗格的索引，对于某一个窗格** ID**，可调用** CStatusBar::CommandToIndex **来获得索引。**

在撤销主框架窗口时应关闭计时器，因此在 CMainFrame::OnClose **函数内调用了** KillTimer **函数。**

现在让我们来看一下 CMainFrame **的消息映射，在** CMainFrame **类所在** CPP **文件的开始部分可以找到该类的消息映射，如清单** 4.9 **所示。**

清单 4.9

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
ON_COMMAND(ID_RECORD_STOP, OnRecordStop)
ON_COMMAND(ID_RECORD_START, OnRecordStart)
ON_UPDATE_COMMAND_UI(ID_RECORD_START,
OnUpdateRecordStart)
ON_UPDATE_COMMAND_UI(ID_RECORD_STOP,
OnUpdateRecordStop)
ON_COMMAND(ID_HIGH_QUALITY, OnHighQuality)
ON_COMMAND(ID_LOW_QUALITY, OnLowQuality)
ON_UPDATE_COMMAND_UI(ID_HIGH_QUALITY,
OnUpdateHighQuality)
ON_UPDATE_COMMAND_UI(ID_LOW_QUALITY,
OnUpdateLowQuality)
ON_COMMAND(ID_VIEW_TOOLBAR1, OnViewToolbar1)
ON_UPDATE_COMMAND_UI(ID_VIEW_TOOLBAR1,
OnUpdateViewToolbar1)
ON_WM_TIMER()
ON_WM_CLOSE()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

读者可以看到，在消息映射表中，ClassWizard **为消息处理函数和命**

令处理函数自动加入了消息映射。自动加入的部分呈灰色显示，位于注释行//{{AFX_MSG_MAP 和//}}AFX_MSG_MAP 之间。命令处理函数由 ON_COMMAND 宏来映射，命令更新处理函数由 ON_UPDATE_COMMAND_UI，而 WM_消息的处理函数由 ON_WM_消息宏来映射。

提示:今后只要看到//{{AFX_...的注释对，则说明它们之间的部分是 ClassWizard 自动加入的，这部分呈灰色显示。请不要随便修改它们，更不能把手工加入的部分放在//{{AFX_...注释对内，否则有可能导致 ClassWizard 出错。/

编译并运行 Record ，可以看到状态栏的新变化，最终的界面如图 4.8 所示。

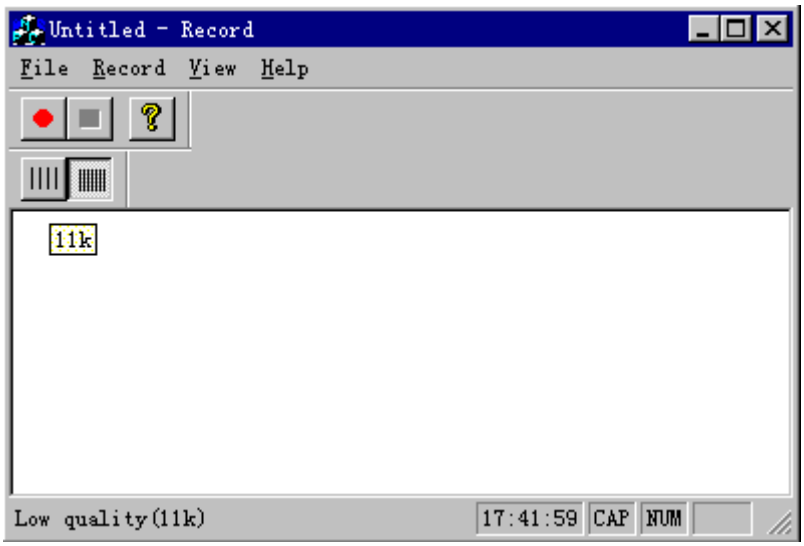


图 4.8 最终的 Record 程序

小 结

本章主要向读者介绍了工具条和状态栏的一些实用技术。要点如下：

在 MFC 中，创建一个窗口一般分两步：1.构建一个窗口对象。构建的方法是定义一个对象或用 new 操作符动态创建之。2.调用窗口类的 Create 成员函数。该函数把实际的窗口作出来，并将其 HWND 保存在窗口的公共数据成员 m_hWnd 中。

创建工具条和状态栏的工作是在 CMainFrame::OnCreate 函数中完成的，OnCreate 函数是在创建窗口时被调用的，这时窗口的创建已部分完成，窗口对象的 HWND 句柄也已有效，但窗口还是不可见的。因此一般在 OnCreate 函数中作一些诸如创建子窗口的初始化工作。

afx_msg 前缀保证了正确版本的消息处理函数被调用。

工具条有两个要素：工具条资源和工具条类 CToolBar。若用户只需要一个工具条，可利用 AppWizard 自动生成，然后再修改之。若需要多个工具条，则必须手工创建。

如果不为命令定义命令处理函数或命令更新处理函数，则框架将自动使该命令对应的用户接口对象(主要指菜单项和按钮)禁止。利用 ClassWizard 可以十分方便的加入命令处理函数和命令更新处理函数。

在菜单下拉之前，或在工具条按钮处在空闲循环期间，MFC 会发一个更新命令，这将导致命令更新处理函数的调用。命令更新处理函数利用 CCmdUI 类来更新用户接口对象。调用 CCmdUI::Enable 可使用户接口对象允许或禁止，调用 CCmdUI::SetCheck 可使用户接口对象选中或不选中。

调用 CWnd::ShowWindow 可以隐藏/显示一个窗口。

要在状态栏中插入新的窗格，需要在 indicator 数组中插入新的字符串 ID。而状态栏将根据这个字符串的长度来确定新窗格的缺省宽度。

调用 CStatusBar:: SetPaneText 可更新状态栏窗格显示的正文。

第五课　对话框

对话框是一种用户界面，它的主要功能是输出信息和接收用户的输入。对话框与控件是密不可分的，在每个对话框内一般都有一些控件，对话框依靠这些控件与用户进行交互．一个典型的对话框例子是选择了File-Open 命令后弹出的文件对话框．

对话框是一种复杂的用户界面，本章的讨论将围绕对话框和基本控件进行，主要包括以下几点：

5.1 对话框和控件的基本概念
5.2 对话框模板的设计
5.3 对话框类的设计
5.4 非模态对话框
5.5 标签式对话框
5.6 公用对话框
5.7 小结

5.1 对话框和控件的基本概念

5.1.1 对话框的基本概念

对话框(Dialog)实际上是一个窗口．在 MFC 中，对话框的功能被封装在了 CDialog 类中，CDialog 类是 CWnd 类的派生类．

对话框分为模态对话框和非模态对话框两种．大部分读者都会有这样的经历，当你通过 File-Open 命令打开一个文件对话框后，再用鼠标去选择菜单将只会发出嘟嘟声，这是因为文件对话框是一个模态对话框．模态对话框垄断了用户的输入，当一个模态对话框打开时，用户只能与该对话框进行交互，而其它用户界面对象收不到输入信息．我们平时所遇到的大部分对话框都是模态对话框。非模态对话框的典型例子是 Windows95 提供的写字板程序中的搜索对话框，搜索对话框不垄断用户的输入，打开搜索对话框后，仍可与其它用户界面对象进行交互，用户可以一边搜索，一边修改文章，这样就大大方便了使用．

本节主要介绍模态对话框，在第四节将介绍非模态对话框．

从 MFC 编程的角度来看，一个对话框由两部分组成：

对话框模板资源．对话框模板用于指定对话框的控件及其分布，Windows 根据对话框模板来创建并显示对话框．

对话框类．对话框类用来实现对话框的功能，由于对话框行使的功能各不相同，因此一般需要从 CDialog 类派生一个新类，以完成特定的功能．

相应地，对话框的设计包括对话框模板的设计和对话框类的设计两个主要方面．

与对话框有关的消息主要包括 WM_INITDIALOG 消息和控件通知消息。在对话框创建时，会收到 WM_INITDIALOG 消息，对话框对该消息的处理函数是 OnInitDialog 。

OnInitDialog 的主要用处是初始化对话框。对话框的控件会向对话框发送控件通知消息，以表明控件的状态发生了变化。

5.1.2 控件的基本概念

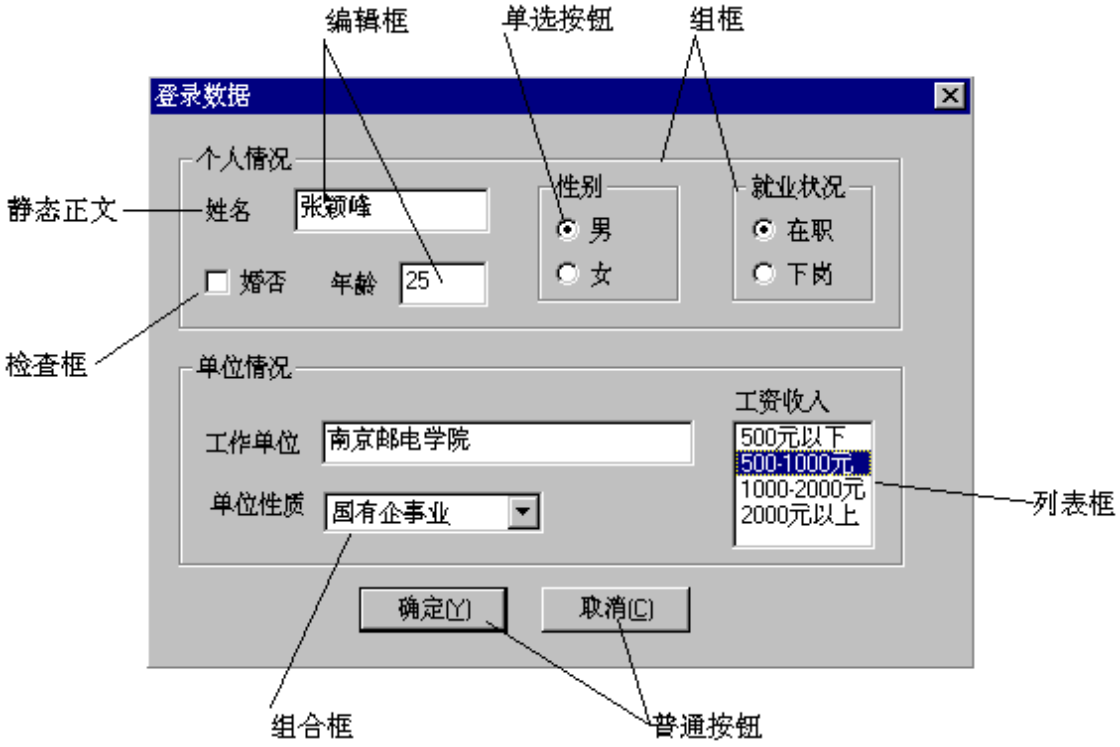


图 5.1 **对话框中的控件**

控件(Control)**是独立的小部件，在对话框与用户的交互过程中，控件担任着主要角色．控件的种类较多，图** 5.1 **显示了对话框中的一些基本的控件．**MFC **的控件类封装了控件的功能，表** 5.1 **介绍了一些常用的控件及其对应的控件类．**

表 5.1

控件	**功　能**	**对应控件类**
静态正文 (Static Text)	**显示正文，一般不能接受输入信息。**	CStatic
图片 (Picture)	**显式位图、图标、方框和图元文件，一般不能接受输入信息．**	CStatic
编辑框 (Edit Box)	**输入并编辑正文，支持单行和多行编辑．**	CEdit
命令按钮 (Pushbutton)	**响应用户的输入，触发相应的事件．**	CButton
检查框 (Check Box)	**用作选择标记，可以有选中、不选中和不确定三种状态。**	CButton
单选按钮 (Radio Button)	**用来从两个或多个选项中选中一项．**	CButton
组框 (Group Box)	**显示正文和方框，主要用来将相关的一些控件聚成一组．**	CButton
列表框 (List Box)	**显示一个列表，用户可以从该列表中选择一项或多项．**	CListBox
组合框 (Combo Box)	**是一个编辑框和一个列表框的组合．分为简易式、下拉式和下拉列表**	CComboBox

	式．	
滚动条 (Scroll Bar)	**主要用来从一个预定义范围值中迅速而有效地选取一个整数值．**	CScrollBar

控件实际上都是窗口，所有的控件类都是 CWnd **类的派生类．控件通常是作为对话框的子窗口而创建的，控件也可以出现在视窗口，工具条和状态条中．**

5.2 对话框模板的设计

利用 Developer Studio 提供的可视化设计工具，用户可以方便地设计对话框模板.

请读者按前面章节介绍的方法利用 AppWizard 建立一个名为 Register 的 MFC 应用程序，并在进入 MFC AppWizard 对话框后按下面几步操作：在第 1 步中选中 Single document 以建立一个单文档应用程序.在第 4 步中使 Docking toolbar 项不选中，这样 AppWizard 就不会创建工具条．在第 6 步中先选择 CRegisterView，然后在 Base class 栏中选择 CEditView，这样 CRegisterView 将是 CEditView 的继承类，从而使视图具有了编辑功能.

编译并运行 Register，读者会发现 Register 居然是个编辑器，它可以打开、编辑和保存文本文件.

当然，Register 的目的不仅仅是个编辑器。假设要对某一地区的就业情况进行调查，我们希望 Register 程序能够登录就业情况数据并将数据存储起来.

要登录数据，用对话框是再合适不过了。一个典型的就业情况登录对话框如图 5.1 所示，本节的任务就是设计如图 5.1 的中文对话框模板.

切换至资源视图，选择 Insert-Resource 命令，并在 Insert Resource 对话框中双击 Dialog 项。完成后在资源视图中会出现一个名为 IDD_DIALOG1 的新的对话框模板资源。双击 IDD_DIALOG1，则会打开该对话框模板的编辑窗口，如图 5.2 所示。缺省的对话框模板有 OK 和 Cancel 两个按钮，在窗口的旁边有一个控件面板，在控件面板上用鼠标选择一个控件，然后在对话框中点击，则相应的控件就被放置到了对话框模板中。图 5.3 显示了控件面板上的按钮所代表的控件。读者不用记忆图 5.3 的内容，如果不能确定控件的类型，可将鼠标在某个控件按钮上停留片刻，则会显示一个工具提示，指出该按钮所代表控件的名称。

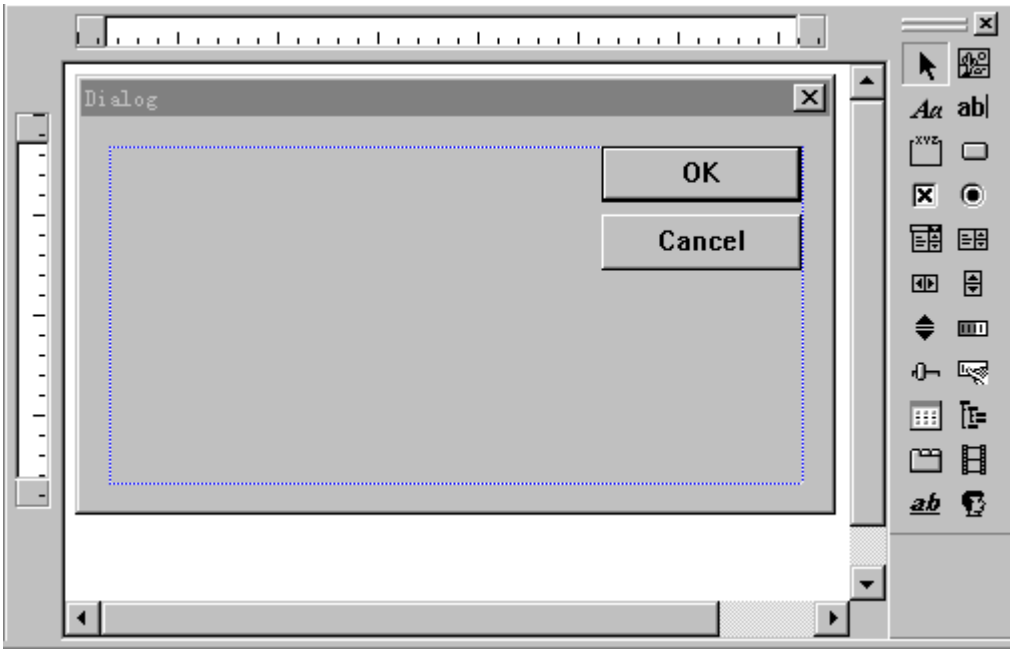


图 5.2 缺省的对话框模板

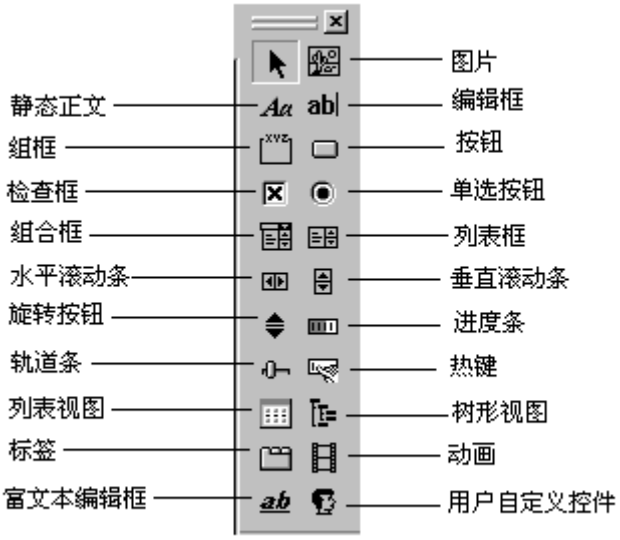


图 5.3 控件面板

提示:若读者看不到控件面板，请在 Developer Studio 的工具条的空白处单击鼠标右键，并在随之弹出的菜单中选中 Controls。/

读者可以在对话框模板中随意加几个控件试试看。当用鼠标选择对话框或控件时，会出现一个围绕它的虚框，拖动虚框的边界可以改变对话框或控件的大小，在 Developer Studio 的状态条中会显示出所选对象的坐标和尺寸。控件可以被拖动，也可以按箭头键来移动选中的控件。在拖动控件时若按住 Ctrl 键，则控件会被复制。

用户可以一次选择多个控件，选择的方法有两个：1。 在对话框的空白处拖动鼠标，则拖动出来的虚线框内的控件将被选中。2。在选择控件时按住 Ctrl 键，则可以多重选择。

选中控件或对话框后按回车键，则会弹出一个属性对话框，属性对话框用来设置控件或对话框的各种属性。属性对话框是标签式对话框，第一页是常规属性(General)。一个典型的控件属性对话框如图 5.4 所示．如果对属性对话框中的选项的意思不明白，可以按 F1 键获得帮助．

图 5.4 控件属性对话框

在控件属性对话框的常规属性中，有一些控件共同的属性：

ID 属性。用于指定控件的标识符，Windows 依靠 ID 来区分不同的控件。

Caption(标题)属性。静态正文、组框、按钮、检查框、单选按钮等控件可以显示标题，用来对控件进行文字说明。控件标题中的字符&使紧跟其后的字符有下划线，按 Alt+下划线将启动该控件。若控件是一个单选按钮，则 Alt+下划线字符将选择该按钮；若是检查框，则相当于对该检查框按空格键；若是按钮，则将激活按钮命令；若控件是一个静态正文，则将激活按 tab 顺序紧随其后的下一个控件。

Visible 属性。用来指定控件是否是可见的。

Disable 属性。使控件允许或禁止，一个禁止的控件呈灰色显示，不能接收任何输入。

Tabstop 属性。用户可以按 Tab 键移动到具有 Tabstop 属性的控件上。Tab 移动的顺序可以由用户指定。按 Ctrl+D 则 Tab 顺序会显示出来，如图 5.5，用户可以用鼠标来重新指定 Tab 顺序。缺省的 Tab 顺序是控件的创建次序。

Group 属性。用来指定一组控件，用户可以用箭头键在该组控件内移动。在同一组内的单选按钮具有互斥的特性，即在这些单选按钮中只能有一个是选中的。如果一个控件具有 Group 属性，则这个控件以及按 Tab 顺序紧随其后的所有控件都属于一组的，直到遇到另一个有 Group 属性的控件为止。

现在就开始进行对话框模板的设计。首先，用鼠标选中对话框，按回车键，在弹出的属性对话框中将 ID 改为 IDD_REGISTER 并指定对话框的标题为“登录数据”。需要注意的是，由于要在对话框中显示汉字，因此必须设定正确的语种和字体。请读者在工作区资源视图的 Dialog 类型中单击鼠标选中 IDD_REGISTER 项，然后按 Alt+Enter 键，并在弹出的属性对话框中的 Language 栏中选择 Chinese(P.R.C.)。接着，打开模板的属性对话框，单击 Font...按钮，并选择“宋体”。

接着，请将对话框模板上的所有控件删除，删除的办法是选择控件后按 Del 键。为了容纳所有需要的控件，需将对话框的尺寸扩大到 280×180。然后，请读者按图 5.1 和表 5.2 来设计对话框模板。

提示：对话框的尺寸单位不是象素，而是与字体的大小有关。X 方向上一个单位等于字符平均宽度的 1/4，Y 方向上一个单位等于字符平均高度的 1/8。这样，随着字体的改变，对话框单位也会改变，对话框本身的总体比例保持不变。

表 5.2

控件类型	ID	标题(Caption)	其它属性
组框(个人情况)	缺省	个人情况	缺省
组框(单位情况)	缺省	单位情况	缺省
静态正文(姓名)	缺省	姓名	缺省
编辑框(姓名)	IDC_NAME		缺省
检查框(婚否)	IDC_MARRIED	婚否	缺省

静态正文(年龄)	缺省	年龄	缺省
编辑框(年龄)	IDC_AGE		缺省
组框(性别)	缺省	性别	缺省
单选按钮(男)	IDC_SEX	男	Group、Tabstop
单选按钮(女)	缺省	女	缺省
组框(就业状况)	缺省	就业状况	缺省
单选按钮(在职)	IDC_WORK	在职	Group、Tabstop
单选按钮(下岗)	IDC_WORK1	下岗	缺省
静态正文(工作单位)	缺省	工作单位	缺省
编辑框(工作单位)	IDC_UNIT		缺省
静态正文(单位性质)	缺省	单位性质	缺省
组合框(单位性质)	IDC_KIND		Drop List、不排序(不选中 Sort 风格)、初始化列表项(见下文说明)
静态正文(工资收入)	缺省	工资收入	缺省
列表框(工资收入)	IDC_INCOME		不排序(不选中 Sort)
按钮(确定)	IDOK	确定(&Y)	缺省
按钮(取消)	IDCANCEL	取消(&C)	缺省

请注意组合框 IDC_KIND 的 Drop List 属性，Drop List 属性是在属性对话框的 Styles(风格)页的 Type 栏中选择的，这使得 IDC_KIND 成为一个下拉列表式组合框。组合框有简易式(Simple)、下拉式(Dropdown)和下拉列表式(Drop List)三种。简易式组合框包含一个编辑框和一个总是显示的列表框。下拉式组合框同简易式组合框的区别在于仅当单击下滚箭头时才出现列表框。下拉列表式组合框也有一个下拉的列表框，但它的编辑框是只读的，不能输入字符。组合框 IDC_KIND 不要自动排序，因此需在 Styles 页中使 Sort 项不被选中。

组合框的列表项可以在设计模板时初始化，而列表框的初始化只能在程序中进行。请读者在组合框 IDC_KIND 的属性对话框的 General 页中输入以下几个列表项，以作为单位性质的选项。输入时要注意，换行时不要按回车键，而应按 Ctrl+回车键。

国有企事业
集体企业
私有企业
中外合资
外商独资

组合框控件的一个与众不同之处是它有两个尺寸，一个是下拉前的

尺寸，一个是下拉后的尺寸。当用鼠标点击组合框上的箭头后，可设定下拉后的尺寸。

控件最好都放在对话框模板的蓝色虚框内，控件之间的距离不要太近，否则有可能造成不正确的显示。

安置好控件之后，下一步的任务是指定 Tab 顺序。按 Ctrl+D 键后，会显示当前的 Tab 顺序，通过用鼠标点击控件可以设定新的 Tab 顺序，如果想放弃本次修改，在对话框的空白处点击一下即可。请读者按图 5.5 安排 Tab 顺序。

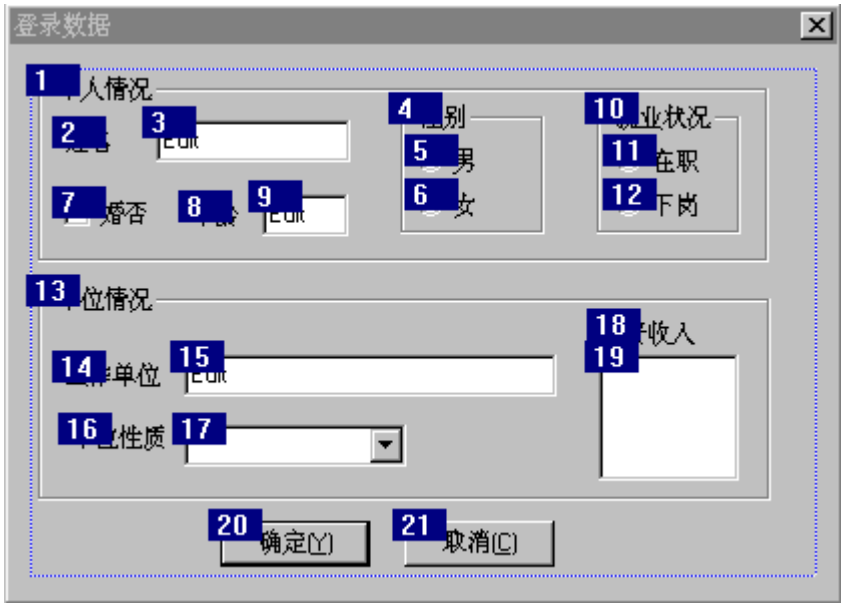


图 5.5 对话框的 Tab 顺序

最后，需要测试一下对话框。按 Ctrl+T，则会弹出一个当前模板的测试对话框，这个对话框的外观和基本行为与程序中将要弹出的对话框一样。这样，读者不用编译运行程序，通过测试对话框就可以评估对话框是否合乎要求。如果发现了错误或不满意的地方，可按 ESC 键退出测试对话框并重新修改对话框模板。

至此，对话框模板的设计就完成了。

5.3 对话框类的设计

完成对话框模板的设计后，就需要设计一个对话框类以实现对话框的功能。设计对话框类主要包括下面几步：

创建对话框类。该类应从 CDialog 类派生。

为对话框类加入与控件相对应的成员变量。

为对话框进行初始化工作。

增加对控件通知消息的处理

5.3.1 对话框类的创建

利用 ClassWizard，程序员可以十分方便的创建 MFC 窗口类的派生类，对话框类也不例外。请读者按以下几步操作：

打开 IDD_REGISTER 对话框模板，然后按 Ctrl+W 进入 ClassWizard。

进入 ClassWizard 后，ClassWizard 发现 IDD_REGISTER 是一个新的对话框模板，于是它会询问是否要为 IDD_REGISTER 创建一个对话框类。按 OK 键确认。

如图 5.6 在 Create New Class 对话框中，在 Name 栏中输入 CRegisterDialog，在 Base class 栏中选择 CDialog，在 Dialog ID 栏中选择 IDD_REGISTER。按 Create 按钮后，对话框类 CRegisterDialog 即被创建。

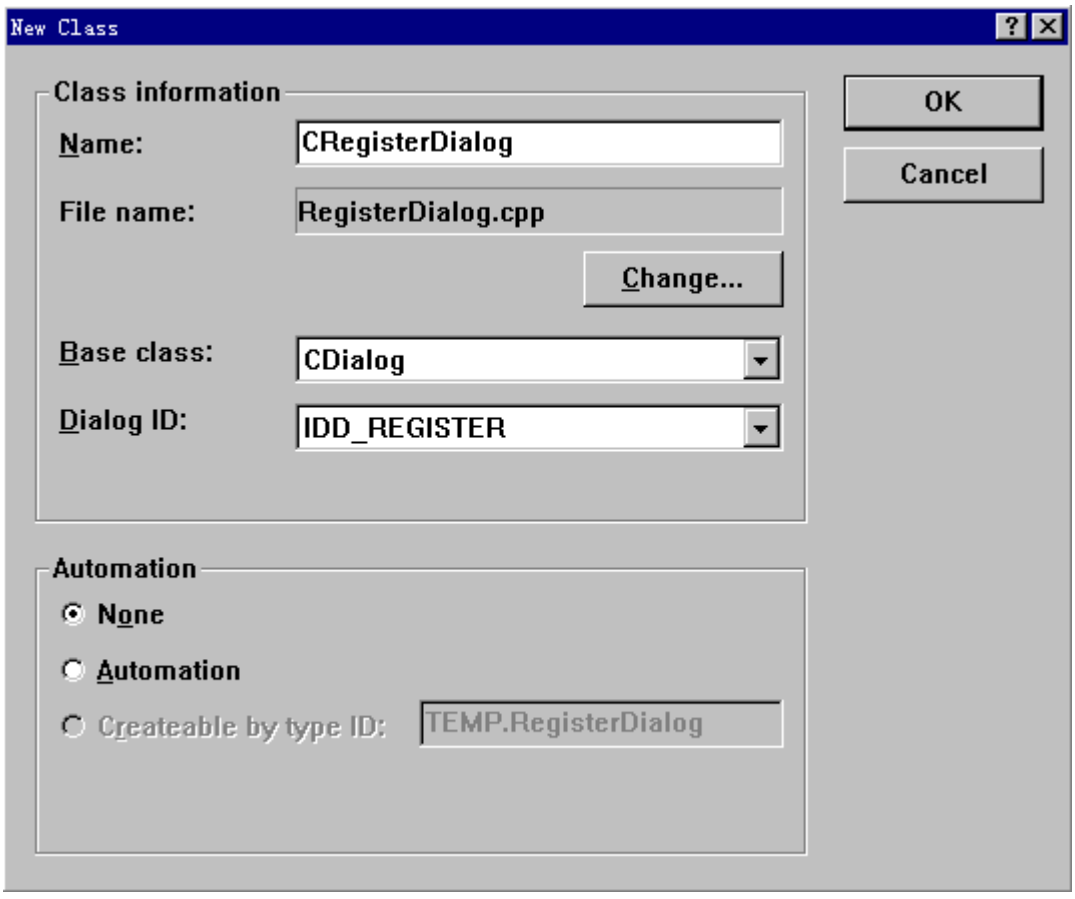


图 5.6 Create New Class 对话框

ClassWizard 自动使类 CRegesterDialog 与 IDD_REGISTER 模板联系起来。

提示：只要想创建的类是某一 MFC 窗口类的派生类，一般都可以利用 ClassWizard 来自动完成创建。创建的一般方法是：打开 ClassWizard，选择 Add Class->New，然后在 Create New Class 对话框中输入新类的类名，选择其 MFC 基类，如果是对话框类，则还要选择对话框的 ID。

5.3.2 为对话框类加入成员变量

对话框的主要功能是输出和输入数据，例子中的登录数据对话框的任务就是输入数据。对话框需要有一组成员变量来存储数据。在对话框中，控件用来表示或输入数据，因此，存储数据的成员变量应该与控件相对应。

与控件对应的成员变量即可以是一个数据，也可以是一个控件对象，这将由具体需要来确定。例如，可以为一个编辑框控件指定一个数据变量，这样就可以很方便地取得或设置编辑框控件所代表的数据，如果想对编辑框控件进行控制，则应该为编辑框指定一个 CEdit 对象，通过 CEdit 对象，程序员可以控制控件的行为。需要指出的是，不同类的控件对应的数据变量的类型往往是不一样的，而且一个控件对应的数据变量的类型也可能有多种。表 5.3 说明了控件的数据变量的类型。

表 5.3

控件	数据变量的类型
编辑框	CString, int, UINT, long, DWORD, float, double, short, BOOL, COleDateTime, COleCurrency
普通检查框	BOOL(真表示被选中，假表示未选中)
三态检查框	int(0 表示未选中，1 表示选中，2 表示不确定状态)
单选按钮(组中的第一个按钮)	int(0 表示选择了组中第一个单选按钮，1 表示选择了第二个...，-1 表示没有一个被选中)
不排序的列表框	CString(为空则表示没有一个列表项被选中), int(0 表示选择了第一项，1 表示选了第二项，-1 表示没有一项被选中)
下拉式组合框	CString, int(含义同上)
其它列表框和组合框	CString(含义同上)

利用 ClassWizard 可以很方便地为对话框类 CRegisterDialog 加入成员变量。请读者按下列步骤操作。

按 Ctrl+W 进入 ClassWizard。

选择 ClassWizard 上部的 Member Variables 标签，然后在 Class name 栏中选择 CRegisterDialog。这时，在下面的变量列表中会出现对话框控件的 ID，如图 5.7 所示。

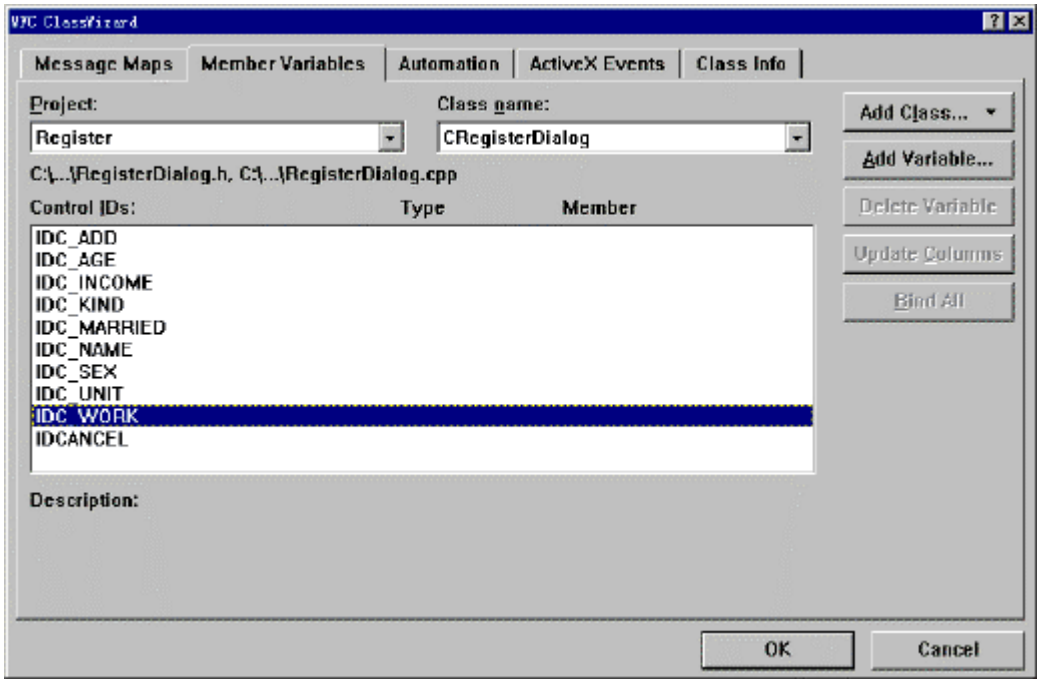


图 5.7 ClassWizard 对话框

双击列表中的 ID_AGE 会弹出 Add Member Variable 对话框，如图 5.8 所示。在 Member variable name 栏中输入 m_nAge，在 Category 栏中选择 Value，在 Variable type 栏中选择 UINT。按 OK 按钮后，数据变量 m_nAge 就会被加入到变量列表中。

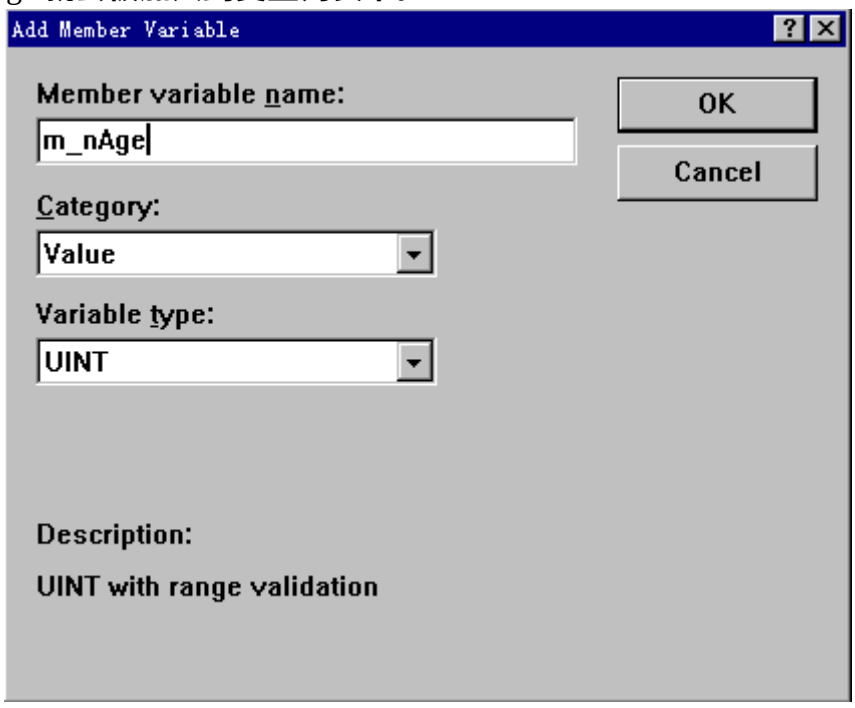


图 5.8 Add Member Variable 对话框

仿照第 3 步和表 5.4，为各个控件加入相应的成员变量。

将 m_nAge 的值限制在 16 到 65 之间。方法是先选择 m_nAge，然后在 ClassWizard 对话框的左下角输入最大和最小值。m_nAge 代表年龄，这里规定被调查的人的年龄应在 16 岁以上，64 岁以下。有了这个限制后，

对话框会对输入的年龄值进行有效性检查，若输入的值不在限制范围内，则对话框会提示用户输入有效的值。

表 5.4

控件 ID	变量类型	变量名
IDC_AGE	UINT	m_nAge
IDC_INCOME	CString	m_strIncome
IDC_INCOME	CListBox	m_ctrlIncome
IDC_KIND	CString	m_strKind
IDC_MARRIED	BOOL	m_bMarried
IDC_NAME	CString	m_strName
IDC_SEX	Int	m_nSex
IDC_UNIT	CString	m_strUnit
IDC_WORK	Int	m_nWork

读者会注意到控件 IDC_INCOME 居然有两个变量，一个是 CString 型的，一个是 CListBox 型的，这是完全合法的，不会引起任何冲突。之所以要加入 CListBox 型的变量，是因为列表框的初始化要通过 CListBox 对象进行。

提示:在 ClassWizard 中可分别为一个控件指定一个数据变量和一个控件对象，这样做的好处是即能方便地获得数据，又能方便地控制控件。

5.3.3 对话框的初始化

对话框的初始化工作一般在构造函数和 OnInitDialog 函数中完成。在构造函数中的初始化主要是针对对话框的数据成员。读者可以找到 CRegisterDialog 的构造函数，如清单 5.1 所示。

清单 5.1 CRegisterDialog 的构造函数

```
CRegisterDialog::CRegisterDialog(CWnd* pParent /*=NULL*/)
: CDialog(CRegisterDialog::IDD, pParent)
{
//{{AFX_DATA_INIT(CRegisterDialog)
m_nAge = 0;
m_strIncome = _T(“”);
m_strKind = _T(“”);
m_bMarried = FALSE;
m_strName = _T(“”);
m_nSex = -1;
m_strUnit = _T(“”);
m_nWork = -1;
//}}AFX_DATA_INIT
}
```

可以看出，对数据成员的初始化是由 ClassWizard 自动完成的。若读者对初值的含义还不太清楚，请参看表 5.3。

在对话框创建时，会收到 WM_INITDIALOG 消息，对话框对该消息的处理函数是 OnInitDialog。调用 OnInitDialog 时，对话框已初步创建，对话框的窗口句柄也已有效，但对话框还未被显示出来。因此，可以在

OnInitDialog 中做一些影响对话框外观的初始化工作。OnInitDialog 对对话框的作用与 OnCreate 对 CMainFrame 的作用类似。

提示:MFC 窗口的初始化工作一般在 OnCreate 成员函数中进行，但对话框的初始化工作最好在 OnInitDialog 中进行。/

OnInitDialog 是 WM_INITDIALOG 消息的处理函数，所以要用 ClassWizard 为 RegisteritDialog 类增加一个 WM_INITDIALOG 消息的处理函数，增加的方法是进入 ClassWizard 后，先选中 MessageMaps 标签，然后在 Class name 中选择 CRegisterDialog，在 Object IDs 栏中选择 CRegisterDialog，在 Messages 栏中找到 WM_INITDIALOG 并双击之，最后按 OK 按钮退出 ClassWizard。

请读者按清单 5.2 修改 OnInitDialog 函数。

清单 5.2 OnInitDialog 函数

```
BOOL CRegisterDialog::OnInitDialog()
{
CDialog::OnInitDialog();
// TODO: Add extra initialization here
m_ctrlIncome.AddString("500 元以下");
m_ctrlIncome.AddString("500-1000 元");
m_ctrlIncome.AddString("1000-2000 元");
m_ctrlIncome.AddString("2000 元以上");
return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE
}
```

CRegisterDialog::OnInitDialog()的主要任务是对工资收入列表框的列表项进行初始化。调用 CListBox::AddString 可将指定的字符串加入到列表框中。由于该列表是不自动排序的，因此 AddString 将表项加在列表框的末尾。

5.3.4 对话框的数据交换机制

对话框的数据成员变量存储了与控件相对应的数据。数据变量需要和控件交换数据，以完成输入或输出功能。例如，一个编辑框即可以用来输入，也可以用来输出：用作输入时，用户在其中输入了字符后，对应的数据成员应该更新；用作输出时，应及时刷新编辑框的内容以反映相应数据成员的变化。对话框需要一种机制来实现这种数据交换功能，这对对话框来说是至关重要的。

MFC 提供了类 CDataExchange 来实现对话框类与控件之间的数据交换(DDX)，该类还提供了数据有效机制(DDV)。数据交换和数据有效机制适用于编辑框、检查框、单选按钮、列表框和组合框。

数据交换的工作由 CDialog::DoDataExchange 来完成。读者可以找到 CRegisterDialog::DoDataExchange 函数，如清单 5.3 所示。

清单 5.3 DoDataExchange 函数

```
void CRegisterDialog::DoDataExchange(CDataExchange* pDX)
{
```

```
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CRegisterDialog)
DDX_Control(pDX, IDC_INCOME, m_ctrlIncome);
DDX_LBString(pDX, IDC_INCOME, m_strIncome);
DDX_CBString(pDX, IDC_KIND, m_strKind);
DDX_Check(pDX, IDC_MARRIED, m_bMarried);
DDX_Text(pDX, IDC_NAME, m_strName);
DDX_Radio(pDX, IDC_SEX, m_nSex);
DDX_Text(pDX, IDC_UNIT, m_strUnit);
DDX_Radio(pDX, IDC_WORK, m_nWork);
DDX_Text(pDX, IDC_AGE, m_nAge);
DDV_MinMaxUInt(pDX, m_nAge, 16, 65);
//}}AFX_DATA_MAP
}
```

读者可以看出，该函数中的代码是由 ClassWizard 自动加入的。DoDataExchange 只有一个参数，即一个 CDataExchange 对象的指针 pDX。在该函数中调用了 DDX 函数来完成数据交换，调用 DDV 函数来进行数据有效检查。

当程序需要交换数据时，不要直接调用 DoDataExchange 函数，而应该调用 CWnd::UpdateData。UpdataData 函数内部调用了 DoDataExchange。该函数只有一个布尔型参数，它决定了数据传送的方向。调用 UpdateData(TRUE)将数据从对话框的控件中传送到对应的数据成员中，调用 UpdateData(FALSE)则将数据从数据成员中传送给对应的控件。

在缺省的 CDialog::OnInitDialog 中调用了 UpdateData(FALSE)，这样，在对话框创建时，数据成员的初值就会反映到相应的控件上。若用户是按了 OK(确定)按钮退出对话框，则对话框认为输入有效，就会调用 UpdataData(TRUE)将控件中的数据传给数据成员。图 5.9 描绘了对话框的这种数据交换机制。

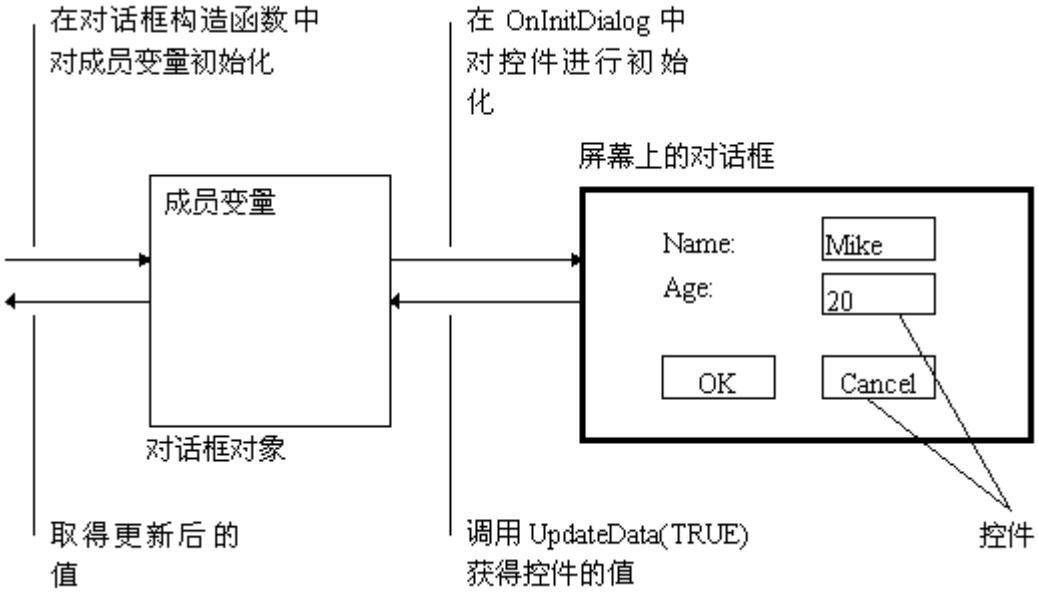


图 5.9 **对话框的数据交换**

5.3.5 **对话框的运行机制**

在程序中运行模态对话框有两个步骤：

在堆栈上以变量的形式构建一个对话框对象。

调用 CDialog::DoModal ( )**。**

DoModal **负责对模态话框的创建和撤销。在创建对话框时，**DoModal **的任务包括载入对话框模板资源、调用** OnInitDialog **初始化对话框和将对话框显示在屏幕上。完成对话框的创建后，**DoModal **启动一个消息循环，以响应用户的输入。由于该消息循环截获了几乎所有的输入消息，使主消息循环收不到对对话框的输入，致使用户只能与模态对话框进行交互，而其它用户界面对象收不到输入信息。**

若用户在对话框内点击了 ID **为** IDOK **的按钮(通常该按钮的标题是“确定”或“**OK**”)，或按了回车键，则** CDialog::OnOK **将被调用。**OnOK **首先调用** UpdateData(TRUE)**将数据从控件传给对话框成员变量，然后调用** CDialog::EndDialog **关闭对话框。关闭对话框后，**DoModal **会返回值** IDOK**。**

若用户点击了 ID **为** IDCANCEL **的按钮(通常其标题为“取消”或“**Cancel**”)，或按了** ESC **键，则会导致** CDialog::OnCancel **的调用。该函数只调用** CDialog::EndDialog **关闭对话框。关闭对话框后，**DoModal **会返回值** IDCANCEL**。**

程序根据 DoModal **的返回值是** IDOK **还是** IDCANCEL **就可以判断出用户是确定还是取消了对对话框的操作。**

在弄清了对话框的运行机制后，下面让我们来就可以实现 Register **程序登录数据的功能。**

首先，将 Register **工程的工作区切换至资源视图。打开** IDR_MAINFRAME **菜单资源，在** Edit **菜单的底端加入一个名为“登录数**

据”的新菜单项，并令其 ID 为 ID_EDIT_REGISTER(最好在该项之前加一条分隔线，以便和前面的菜单项分开)。注意不要忘了把菜单资源的语种设置成中文，否则菜单中将显示不出中文来。设置的方法是先在工作区资源视图中选择 IDR_MAINFRAME 菜单资源，然后按 Alt+Enter 键，并在弹出的属性对话框中的 Language 栏中选择 Chinese(P.R.C.)。

接着，用 ClassWizard 为该菜单命令创建命令处理函数 CRegisterView::OnEditRegister。注意，OnEditRegister 是类 CRegisterView 的成员函数，这是因为 CRegisterView 要负责打开和关闭登录数据对话框，并将从对话框中输入的数据在视图中输出。

然后，请读者在 RegisterView.cpp 文件的开头加入下面一行

```
#include “RegisterDialog.h”
```

最后，按清单 5.4 修改程序。

清单 5.4 OnEditRegister 函数

```
void CRegisterView::OnEditRegister()
{
// TODO: Add your command handler code here
CRegisterDialog dlg;
if(dlg.DoModal()==IDOK)
{
CString str;
//获取编辑正文
GetWindowText(str);
//换行
str+=“\r\n”;
str+=“姓名:”;
str+=dlg.m_strName;
str+=“\r\n”;
str+=“性别:”;
str+=dlg.m_nSex””女”:”男”;
str+=“\r\n”;
str+=“年龄:”;
CString str1;
//将数据格式输出到字符串对象中
str1.Format(“%d”,dlg.m_nAge);
str+=str1;
str+=“\r\n”;
str+=“婚否:”;
str+=dlg.m_bMarried””已婚”:”未婚”;
str+=“\r\n”;
str+=“就业状况:”;
str+=dlg.m_nWork””下岗”:”在职”;
str+=“\r\n”;
```

```
str+="工作单位:";
str+=dlg.m_strUnit;
str+="\r\n";
str+="单位性质:";
str+=dlg.m_strKind;
str+="\r\n";
str+="工资收入:";
str+=dlg.m_strIncome;
str+="\r\n";
//更新编辑视图中的正文
SetWindowText(str);
}
}
```

在 OnEditRegister 函数中，首先构建了一个 CRegisterDialog 对象，然后调用 CDialog::DoModal 来实现模态对话框。如果 DoModal 返回 IDOK，则说明用户确认了登录数据的操作，程序需要将录入的数据在编辑视图中输出。程序用一个 CString 对象来作为编辑正文的缓冲区，CString 是一个功能强大的字符串类，它的最大特点在于可以存储动态改变大小的字符串，这样，用户不必担心字符串的长度超过缓冲区的大小，使用十分方便。

在输出数据时，程序首先调用 CWnd::GetWindowText 获得编辑正文，这是一个多行的编辑正文。CWnd::GetWindowText 用来获取窗口的标题，若该窗口是一个控件，则获取的是控件内的正文。CRegisterView 是 CEditView 的继承类，而 CEditView 实际上包含了一个编辑控件，因此在 CRegisterView 中调用 GetWindowText 获得的是编辑正文。

然后，程序在该编辑正文的末尾加入新的数据。在程序中大量使用了 CString 类的重载操作符“＋＝”，该操作符的功能是将操作符右侧字符串添加到操作符左侧的字符串的末尾。注意在多行编辑控件中每行末尾都有一对回车和换行符。在程序中还调用了 CString::Format 来将数据格式化输出到字符串中，Format 的功能与 sprintf 类似。最后，调用 CWnd::SetWindowText 来更新编辑视图中的正文。

编译并运行 Register，打开登录数据对话框，输入一些数据试试。现在，Register 已经是一个简易的数据库应用程序了，它可以将与就业情况有关的数据输出到一个编辑视图中。用户可以编辑视图中的正文，并将结果保存在文本文件中。

5.3.6 处理控件通知消息

虽然 Register 已经可以登录数据了，但读者会很快会发现该程序还有一些不完善的地方：

登录完一个人的数据后，对话框就关闭了，若用户有很多人的数据要输入，则必须频繁地打开对话框，很不方便。在登录数据时，应该使对话框一直处于打开状态。

登录数据对话框分个人情况和单位情况两组，若被调查人是下岗职

工，则不必输入单位情况。程序应该能够对用户的输入及时地作出反应，即当用户选择了“ 下岗 ”单选按钮时，应使单位情况组中的控件禁止。一个禁止的控件呈灰色示，并且不能接收用户的输入。

要解决上述问题，就必须对控件通知消息进行处理。当控件的状态因为输入等原因而发生变化时，控件会向其父窗口发出控件通知消息。例如，如果用户在登录数据对话框中的某一按钮(包括普通按钮、检查框和单选按钮)上单击鼠标，则该按钮会向对话框发送 BN_CLICKED 消息。对话框根据按钮的 ID 激活相应的 BN_CLICKED 消息处理函数，以对单击按钮这一事件作出反应。通过对按钮的 BN_CLICKED 消息的处理，我们可以使登录数据对话框的功能达到上述要求。

首先，让我们来解决第一个问题。我们的设想是修改原来的“ 确定(Y) ”按钮，使得当用户点击该按钮后，将数据输出到视图中，并且对话框不关闭，以便用户输入下一个数据。请读者按下面几步进行修改。

修改登录数据对话框的“ 确定(Y) ”按钮，使该按钮的标题变为“ 添加(&A) ”，ID 变为 IDC_ADD。这样，当用户点击该按钮后，对话框会收到 BN_CLICKED 消息。由于这个 BN_CLICKED 消息对应的按钮 ID 不是 IDOK，不会触发 OnOK 消息处理函数，因此不会关闭对话框。

为按钮 IDC_ADD 的 BN_CLICKED 消息创建消息处理函数。创建的方法是进入 ClassWizard 后，选 Message Maps 页并在 Class name 栏中选择 CRegisterDialog，然后在 Object IDs 栏中选择 IDC_ADD，在 Messages 栏中双击 BN_CLICKED。在确认使用缺省的消息处理函数名 OnAdd 后，按回车键退出 ClassWizard。

OnAdd 要向编辑视图输出正文，就必须获得一个指向 CRegisterView 对象的指针以访问该对象。为此，请在 CRegisterDialog 类的说明中加入下面一行 Cwnd* m_pParent;注意不要加在 AFX 注释对中。

为实现 IDC_ADD 按钮的功能，请按清单 5.5 和清单 5.6 修改程序。主要的改动是把原来由 CRegiserView::OnEditRegister 完成的在视图中输出数据的任务交给 CRegisterDialog::OnAdd 来完成。

清单 5.5 CRegisterView::OnEditRegister 函数

```
void CRegisterView::OnEditRegister()
{
// TODO: Add your command handler code here
CRegisterDialog dlg(this);
dlg.DoModal();
}
```

清单 5.6 CRegisterDialog 类的部分源代码

```
CRegisterDialog::CRegisterDialog(CWnd* pParent /*=NULL*/)
: CDialog(CRegisterDialog::IDD, pParent)
{
//{{AFX_DATA_INIT(CRegisterDialog)
. . . . . .
//}}AFX_DATA_INIT
```

```
m_pParent=pParent;
}
void CRegisterDialog::OnAdd()
{
// TODO: Add your control notification handler code here
//更新数据
UpdateData(TRUE);
//检查数据是否有效
if(m_strName==“” || m_nSex<0 || m_nWork<0 || m_strUnit==“”
|| m_strKind==“” || m_strIncome==“”)
{
AfxMessageBox(“请输入有效数据”);
return;
}
CString str;
//获取编辑正文
m_pParent->GetWindowText(str);
//换行
str+=“\r\n”;
str+=“姓名:”;
str+=m_strName;
str+=“\r\n”;
str+=“性别:”;
str+=m_nSex””女”:”男”;
str+=“\r\n”;
str+=“年龄:”;
CString str1;
//将数据格式输出到字符串对象中
str1.Format(“%d”,m_nAge);
str+=str1;
str+=“\r\n”;
str+=“婚否:”;
str+=m_bMarried””已婚”:”未婚”;
str+=“\r\n”;
str+=“就业状况:”;
str+=m_nWork””下岗”:”在职”;
str+=“\r\n”;
str+=“工作单位:”;
str+=m_strUnit;
str+=“\r\n”;
str+=“单位性质:”;
str+=m_strKind;
```

```
str+="\r\n";
str+="工资收入:";
str+=m_strIncome;
str+="\r\n";
//更新编辑视图中的正文
m_pParent->SetWindowText(str);
}
```

CRegisterDialog 的构造函数有一个参数 pParent，该参数是一个指向 CWnd 对象的指针，用于指定对话框的父窗口或拥有者窗口。在 CRegisterView:: OnEditRegister 函数中，在构建 CRegisterDialog 对象时指定了 this 参数，this 指针指向 CRegisterView 对象本身。这样在调用 CRegisterDialog 的构造函数时，this 指针值被赋给了 CRegisterDialog 的成员 m_pParent。OnAdd 函数可利用 m_pParent 来访问对话框的拥有者即 CRegisterView 对象。

提示：术语父窗口(Parent)是相对于子窗口而言。若某一个窗口拥有一个子窗口(Child)，则该窗口就被称为子窗口的父窗口。子窗口就是具有 WS_CHILD 风格的窗口，子窗口依赖于父窗口且完全被限制在父窗口内部。拥有者窗口(owner)相对于被拥有者窗口而言。若某一个窗口拥有一个非子窗口，则该窗口被称为拥有者窗口。被拥有窗口(owned)不具有 WS_CHILD 风格，可在屏幕上任意移动。/

当用户用鼠标点击 IDC_ADD 按钮时，该按钮的 BN_CLICKED 消息处理函数 CRegisterDialog::OnAdd 将被调用。在 OnAdd 中，首先调用了 UpdateData(TRUE)以把数据从控件传给对话框的数据成员变量。然后，程序要对数据的有效性进行检查，如果输入的数据不完全有效，则会显示一个消息对话框，提示用户输入有效的数据。接下来进行的工作是在视图中输出数据，这部分代码与清单 5.4 类似，读者应该比较熟悉了。

完成上述工作后，登录数据对话框就变得较为实用了。打开对话框后，用户可以方便地输入多人的数据，只有按了取消按钮后，对话框才会关闭。

接下来让我们来解决第二个问题。解决该问题的关键在于当用户点击“在职”或“下岗”单选按钮时，程序要对收到的 BN_CLICKED 消息作出响应。有些读者可能会想到为两个单选按钮分别创建 BN_CLICKED 消息处理函数，这在只有两个单选按钮的情况下是可以的，但如果一组内有多个单选按钮，则分别创建消息处理函数就比较麻烦了。利用 MFC 提供的消息映射宏 ON_CONTROL_RANGE 可以避免这种麻烦，该映射宏把多个 ID 连续的控件发出的消息映射到同一个处理函数上。这样，我们只要编写一个消息处理函数，就可以对“在职”和“下岗”两个单选按钮的 BN_CLICKED 消息作出响应。ClassWizard 不支持 ON_CONTROL_RANGE 宏，所以我们必须手工创建单选按钮的消息映射和消息处理函数。

首先，在 CRegisterDialog 类的头文件中加入消息处理函数的声明，该函数名为 OnWorkClicked，如清单 5.7 所示。

清单 5.7 BN_CLICKED **消息处理函数** OnWorkClicked **的声明**

```
. . . . . .
protected:
void OnWorkClicked(UINT nCmdID);
// Generated message map functions
//{{AFX_MSG(CRegisterDialog)
virtual BOOL OnInitDialog();
afx_msg void OnAdd();
//}}AFX_MSG
. . . . . .
```

然后，在 CRegisterDialog **类的消息映射中加入** ON_CONTROL_RANGE **映射，如清单** 5.8 **所示。**ON_CONTROL_RANGE **映射的形式是** ON_CONTROL_RANGE

清单 5.8 **在** CRegisterDialog **类的消息映射中加入** ON_CONTROL_RANGE **映射**

```
BEGIN_MESSAGE_MAP(CRegisterDialog, CDialog)
//{{AFX_MSG_MAP(CRegisterDialog)
ON_BN_CLICKED(IDC_ADD, OnAdd)
//}}AFX_MSG_MAP
ON_CONTROL_RANGE(BN_CLICKED, IDC_WORK, IDC_WORK1,
OnWorkClicked)
END_MESSAGE_MAP()
```

ON_CONTROL_RANGE **消息映射宏的第一个参数是控件消息码，第二和第三个参数分别指明了一组连续的控件** ID **中的头一个和最后一个** ID**，最后一个参数是消息处理函数名。如果读者是按表** 5.2 **的顺序放置控件的则** IDC_WORK **和** IDC_WORK1 **应该是连续的。这样，无论用户是在** IDC_WORK **还是在** IDC_WORK1 **单选按钮上单击，都会调用** OnWorkClicked **消息处理函数。**

提示：如果不能确定两个 ID **是否是连续的，请用** File->Open **命令打开** resource.h **文件，在该文件中有对控件** ID **值的定义。如果发现两个** ID **是不连续的，读者可以改变对** ID **的定义值使之连续，但要注意改动后的值不要与别的** ID **值发生冲突。**

最后，在 CRegisterDialog **类所在** CPP **文件的最后插入消息处理函数** CRegisterDialog::OnWorkClicked**，如清单** 5.9 **所示。**

清单 5.9 CRegisterDialog::OnWorkClicked **消息处理函数**

```
void CRegisterDialog::OnWorkClicked(UINT nCmdID)
{
//判断“在职”单选按钮是否被选中
if(IsDlgButtonChecked(IDC_WORK))
{
//使控件允许
GetDlgItem(IDC_UNIT)->EnableWindow(TRUE);
```

```
GetDlgItem(IDC_KIND)->EnableWindow(TRUE);
GetDlgItem(IDC_INCOME)->EnableWindow(TRUE);
}
else
{
//清除编辑框的内容并使之禁止
GetDlgItem(IDC_UNIT)->SetWindowText(“”);
GetDlgItem(IDC_UNIT)->EnableWindow(FALSE);
//使组合框处于未选择状态并使之禁止
CComboBox *pComboBox=(CComboBox *)GetDlgItem(IDC_KIND);
pComboBox->SetCurSel(-1);
pComboBox->EnableWindow(FALSE);
//使列表框处于未选择状态并使之禁止
m_ctrlIncome.SetCurSel(-1);
m_ctrlIncome.EnableWindow(FALSE);
}
}
```

OnWorkClicked 函数判断“在职”单选按钮是否被选中。若该按钮被选中，则使单位情况组中的控件允许，若该按钮未被选中，则说明“下岗”按钮被选中，这时应使控件禁止，清除编辑框中的正文， 并且使组合框和列表框处于未选中状态。

在 OnWorkClicked 函数中主要调用了下列函数：

CWnd::IsDlgButtonChecked 函数，用来判断单选按钮或检查框是否被选择，该函数的声明为 UINT IsDlgButtonChecked(int nIDButton) const; 参数 nIDButton 为按钮的 ID。若按钮被选择，则函数返回 1，否则返回 0，若按钮处于不确定状态，则返回值为 2。

CWnd::GetDlgItem 函数，用来获得指向某一控件的指针，该函数的声明为 CWnd* GetDlgItem(int nID) const;参数 nID 为控件的 ID。该函数返回一个指定控件的 CWnd 对象指针，通过该指针，程序可以对控件进行控制。

CWnd::EnableWindow 函数，该函数使窗口允许或禁止，禁止的窗口呈灰色显示，不能接收键盘和鼠标的输入。该函数的声明是 BOOL EnableWindow(BOOL bEnable = TRUE);若参数 bEnable 的值为 TRUE，则窗口被允许，若 bEnable 的值为 FALSE，则窗口被禁止。

CListBox::SetCurSel 和 CComboBox::SetCurSel 函数功能类似，用来使列表中的某一项被选中，选中的项呈高亮度显示。函数的声明是 int SetCurSel(int nSelect);参数 nSelect 指定了新选项的索引，第一项的索引值为 0，若 nSelect 的值为-1，那么函数将清除以前的选择，使列表处于未选择状态。

有时，需要将 GetDlgItem 返回的 CWnd 指针强制转换成控件对象的指针，以便调用控件对象专有的成员函数对控件进行控制。例如，在程序中 GetDlgItem(IDC_KIND)返回的指针被强制转换成 CComboBox 类

型，只有这样，才能调用 CComboBox::SetCurSel 成员函数。

为了对控件进行查询和控制，在程序中采用了两种访问控件的方法。一种方法是直接利用 ClassWizard 提供的控件对象，例如 m_ctrlIncome 列表框对象。另一种方法是利用 CWnd 类提供的一组管理对话框控件的成员函数，例如程序中用到的 GetDlgItem 和 IsDlgButtonChecked。这两种方法是在对话框内访问控件的常用方法，读者都应该掌握。表 5.5 列出了管理对话框控件的 Cwnd 成员函数。

表 5.5 用来管理对话框控件的 CWnd 成员函数

函数名	功能
CheckDlgButton	选中或不选中按钮控件。
CheckRadioButton	选择一个指定的单选按钮并使同组内的其它单选按钮不被选择。
DlgDirList	往一个列表框中添加文件、目录或驱动器的列表。
DlgDirListComboBox	往一个组合框中的列表框内添加文件、目录或驱动器的列表。
DlgDirSelect	从一个列表框中获得当前选择的文件、目录或驱动器。
DlgDirSelectBomboBox	从一个组合框中获得当前选择的文件、目录或驱动器。
GetCheckedRadioButton	返回指定的单选按钮组中被选择的单选按钮的 ID。
GetDlgItem	返回一个指向一给定的控件的临时对象的指针。
GetDlgItemInt	返回在一个指定的控件中由正文表示的数字值。
GetDlgItemText	获得在一个控件内显示的正文。
GetNextDlgGroupItem	返回一个指向一组控件内的下一个或上一个控件的临时对象的指针。
GetNextDlgTabItem	返回下一个 tab 顺序的控件的临时对象的指针。
IsDlgButtonChecked	返回一个按钮控件的状态。
SendDlgItemMessage	把一个消息传送给一个控件。
SetDlgItemInt	将一个整数转换为正文，并将此正文赋给控件。
SetDlgItemText	设置一个控件显示的正文。

编译并运行 Register 看看，现在的登录数据对话框已经比较令人满意了。

5.4 非模态对话框

5.4.1 非模态对话框的特点

与模态对话框不同，非模态对话框不垄断用户的输入，用户打开非模态对话框后，仍然可以与其它界面进行交互。

非模态对话框的设计与模态对话框基本类似，也包括设计对话框模板和设计 CDialog 类的派生类两部分。但是，在对话框的创建和删除过程中，非模态对话框与模态对话框相比有下列不同之处：

非模态对话框的模板必须具有 Visible 风格，否则对话框将不可见，而模态对话框则无需设置该项风格。更保险的办法是调用 CWnd::ShowWindow(SW_SHOW)来显示对话框，而不管对话框是否具有 Visible 风格。

非模态对话框对象是用 new 操作符在堆中动态创建的，而不是以成员变量的形式嵌入到别的对象中或以局部变量的形式构建在堆栈上。通常应在对话框的拥有者窗口类内声明一个指向对话框类的指针成员变量，通过该指针可访问对话框对象。

通过调用 CDialog::Create 函数来启动对话框，而不是 CDialog::DoModal，这是模态对话框的关键所在。由于 Create 函数不会启动新的消息循环，对话框与应用程序共用同一个消息循环，这样对话框就不会垄断用户的输入。Create 在显示了对话框后就立即返回，而 DoModal 是在对话框被关闭后才返回的。众所周知，在 MFC 程序中，窗口对象的生存期应长于对应的窗口，也就是说，不能在未关闭屏幕上窗口的情况下先把对应的窗口对象删除掉。由于在 Create 返回后，不能确定对话框是否已关闭，这样也就无法确定对话框对象的生存期，因此只好在堆中构建对话框对象，而不能以局部变量的形式来构建之。

必须调用 CWnd::DestroyWindow 而不是 CDialog::EndDialog 来关闭非模态对话框。调用 CWnd::DestroyWindow 是直接删除窗口的一般方法。由于缺省的 CDialog::OnOK 和 CDialog::OnCancel 函数均调用 EndDialog，故程序员必须编写自己的 OnOK 和 OnCancel 函数并且在函数中调用 DestroyWindow 来关闭对话框。

因为是用 new 操作符构建非模态对话框对象，因此必须在对话框关闭后，用 delete 操作符删除对话框对象。在屏幕上一个窗口被删除后，框架会调用 CWnd::PostNcDestroy，这是一个虚拟函数，程序可以在该函数中完成删除窗口对象的工作，具体代码如下 void CModelessDialog::PostNcDestroy{delete this; //删除对象本身}这样，在删除屏幕上的对话框后，对话框对象将被自动删除。拥有者对象就不必显式的调用 delete 来删除对话框对象了。

必须有一个标志表明非模态对话框是否是打开的。这样做的原因是用户有可能在打开一个模态对话框的情况下，又一次选择打开命令。程序根据标志来决定是打开一个新的对话框，还是仅仅把原来打开的对话框激活。通常可以用拥有者窗口中的指向对话框对象的指针作为这种标志，当对话框关闭时，给该指针赋 NULL 值，以表明对话框对象已不存

在了。

提示：在 C++编程中，判断一个位于堆中的对象是否存在的常用方法是判断指向该对象的指针是否为空。这种机制要求程序员将指向该对象的指针初始化为 NULL 值，在创建对象时将返回的地址赋给该指针，而在删除对象时将该指针置成 NULL 值。/

根据上面的分析，我们很容易把 Register 程序中的登录数据对话框改成非模态对话框。这样做的好处在于如果用户在输入数据时发现编辑视图中有错误的数据，那么不必关闭对话框，就可以在编辑视图中进行修改。

请读者按下面几步操作：

在登录数据对话框模板的属性对话框的 More Styles 页中选择 Visible 项。

在 RegisterView.h 头文件的 CRegisterView 类的定义中加入 public:CRegisterDialog* m_pRegisterDlg;

在 RegisterView.h 头文件的头部加入对 CRegisterDialog 类的声明 class CRegisterDialog;加入该行的原因是在 CRegisterView 类中有一个 CRegisterDialog 类型的指针，因此必须保证 CRegisterDialog 类的声明出现在 CRegisterView 之前，否则编译时将会出错。解决这个问题有两种办法，一种办法是保证在#include “RegisterView.h”语句之前有#include “RegisterDialog.h”语句，这种办法造成了一种依赖关系，增加了编译负担，不是很好；另一种办法是在 CRegisterView 类的声明之前加上一个对 CRegisterDialog 的声明来暂时“蒙蔽”编译器，这样在有#include “RegisterView.h”语句的模块中，除非要用到 CRegisterDialog 类，否则不用加入#include “RegisterDialog.h”语句。

在 RegisterDialog.cpp 文件的头部的#include 语句区的末尾添加下面两行#include “RegisterDoc.h”#include “RegisterView.h”

利用 ClassWizard 为 CRegisterDialog 类加入 OnCancel 和 PostNcDestroy 成员函数。加入的方法是进入 ClassWizard 后选择 Message Maps 页，并在 Class name 栏中选择 CRegisterDialog。然后，在 Object IDs 栏中选择 IDCANCEL 后，在 Messages 栏中双击 BN_CLICKED，这就创建了 OnCancel。要创建 PostNcDestroy，先在 Object IDs 栏中选择 CRegisterDialog，再在 Messages 栏中双击 PostNcDestroy 即可。

分别按清单 5.10 和 5.11，对 CRegisterView 类和 CRegisterDialog 类进行修改。

清单 5.10 CRegisterView 类的部分代码

```
CRegisterView::CRegisterView()
{
// TODO: add construction code here
m_pRegisterDlg=NULL; //指针初始化为 NULL
}
void CRegisterView::OnEditRegister()
{
```

```
// TODO: Add your command handler code here
if(m_pRegisterDlg)
m_pRegisterDlg->SetActiveWindow(); //激活对话框
else
{
//创建非模态对话框
m_pRegisterDlg=new CRegisterDialog(this);
m_pRegisterDlg->Create(IDD_REGISTER,this);
}
}
```

清单 5.11 CRegisterDialog **的部分代码**

```
void CRegisterDialog::PostNcDestroy()
{
// TODO: Add your specialized code here and/or call the base class
delete this; //删除对话框对象
}
void CRegisterDialog::OnCancel()
{
// TODO: Add extra cleanup here
((CRegisterView*)m_pParent)->m_pRegisterDlg=NULL;
DestroyWindow(); //删除对话框
}
```

CRegisterView::OnEditRegister **函数判断登录数据对话框是否已打开，若是，就激活对话框，否则，就创建该对话框。该函数中主要调用了下列函数：**

调用 CWnd::SetActiveWindow **激活对话框，该函数的声明为** CWnd* SetActiveWindow( );**该函数使本窗口成为活动窗口，并返回原来活动的窗口。**

调用 CDialog::Create **来显示对话框，该函数的声明为** BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL); **参数** nIDTemplate **是对话框模板的** ID。pParentWnd **指定了对话框的父窗口或拥有者。**

当用户在登录数据对话框中点击“取消”按钮后，CRegisterDialog::OnCancel **将被调用，在该函数中调用** CWnd::DestroyWindow **来关闭对话框，并且将** CRegisterView **的成员** m_pRegisterDlg **置为** NULL **以表明对话框被关闭了。调用** DestroyWindow **导致了对** CRegisterDialog::PostNcDestroy **的调用，在该函数中用** delete **操作符删除了** CRegisterDialog **对象本身。**

编译并运行 Register**，现在登录数据对话框已经变成一个非模态对话框了。**

5.4.2 **窗口对象的自动清除**

一个 MFC **窗口对象包括两方面的内容：一是窗口对象封装的窗口，**

即存放在 m_hWnd 成员中的 HWND(窗口句柄)，二是窗口对象本身是一个 C++对象。要删除一个 MFC 窗口对象，应该先删除窗口对象封装的窗口，然后删除窗口对象本身。

删除窗口最直接方法是调用 CWnd::DestroyWindow 或::DestroyWindow，前者封装了后者的功能。前者不仅会调用后者，而且会使成员 m_hWnd 保存的 HWND 无效(NULL)。如果 DestroyWindow 删除的是一个父窗口或拥有者窗口，则该函数会先自动删除所有的子窗口或被拥有者，然后再删除父窗口或拥有者。在一般情况下，在程序中不必直接调用 DestroyWindow 来删除窗口，因为 MFC 会自动调用 DestroyWindow 来删除窗口。例如，当用户退出应用程序时，会产生 WM_CLOSE 消息，该消息会导致 MFC 自动调用 CWnd::DestroyWindow 来删除主框架窗口，当用户在对话框内按了 OK 或 Cancel 按钮时，MFC 会自动调用 CWnd::DestroyWindow 来删除对话框及其控件。

窗口对象本身的删除则根据对象创建方式的不同，分为两种情况。在 MFC 编程中，会使用大量的窗口对象，有些窗口对象以变量的形式嵌入在别的对象内或以局部变量的形式创建在堆栈上，有些则用 new 操作符创建在堆中。对于一个以变量形式创建的窗口对象，程序员不必关心它的删除问题，因为该对象的生命期总是有限的，若该对象是某个对象的成员变量，它会随着父对象的消失而消失，若该对象是一个局部变量，那么它会在函数返回时被清除。

对于一个在堆中动态创建的窗口对象，其生命期却是任意长的。初学者在学习 C++编程时，对 new 操作符的使用往往不太踏实，因为用 new 在堆中创建对象，就不能忘记用 delete 删除对象。读者在学习 MFC 的例程时，可能会产生这样的疑问，为什么有些程序用 new 创建了一个窗口对象，却未显式的用 delete 来删除它呢”问题的答案就是有些 MFC 窗口对象具有自动清除的功能。

如前面讲述非模态对话框时所提到的，当调用 CWnd::DestroyWindow 或::DestroyWindow 删除一个窗口时，被删除窗口的 PostNcDestroy 成员函数会被调用。缺省的 PostNcDestroy 什么也不干，但有些 MFC 窗口类会覆盖该函数并在新版本的 PostNcDestroy 中调用 delete this 来删除对象，从而具有了自动清除的功能。此类窗口对象通常是用 new 操作符创建在堆中的，但程序员不必操心用 delete 操作符去删除它们，因为一旦调用 DestroyWindow 删除窗口，对应的窗口对象也会紧接着被删除。

不具有自动清除功能的窗口类如下所示。这些窗口对象通常是以变量的形式创建的，无需自动清除功能。

所有标准的 Windows 控件类。

从 CWnd 类直接派生出来的子窗口对象(如用户定制的控件)。

切分窗口类 CSplitterWnd。

缺省的控制条类(包括工具条、状态条和对话条)。

模态对话框类。

具有自动清除功能的窗口类如下所示，这些窗口对象通常是在堆中

创建的。

主框架窗口类(直接或间接从 CFrameWnd 类派生)。

视图类(直接或间接从 CView 类派生)。

读者在设计自己的派生窗口类时，可根据窗口对象的创建方法来决定是否将窗口类设计成可以自动清除的。例如，对于一个非模态对话框来说，其对象是创建在堆中的，因此应该具有自动清除功能。

综上所述，对于 MFC 窗口类及其派生类来说，在程序中一般不必显式删除窗口对象。也就是说，既不必调用 DestroyWindow 来删除窗口对象封装的窗口，也不必显式地用 delete 操作符来删除窗口对象本身。只要保证非自动清除的窗口对象是以变量的形式创建的，自动清除的窗口对象是在堆中创建的，MFC 的运行机制就可以保证窗口对象的彻底删除。

如果需要手工删除窗口对象，则应该先调用相应的函数(如 CWnd::DestroyWindow)删除窗口，然后再删除窗口对象．对于以变量形式创建的窗口对象，窗口对象的删除是框架自动完成的．对于在堆中动态创建了的非自动清除的窗口对象，必须在窗口被删除后，显式地调用 delete 来删除对象(一般在拥有者或父窗口的析构函数中进行)．对于具有自动清除功能的窗口对象，只需调用 CWnd::DestroyWindow 即可删除窗口和窗口对象。注意，对于在堆中创建的窗口对象，不要在窗口还未关闭的情况下就用 delete 操作符来删除窗口对象．

提示：在非模态对话框的 OnCancel 函数中可以不调用 CWnd::DestroyWindow，取而代之的是调用 CWnd::ShowWindow(SW_HIDE)来隐藏对话框．在下次打开对话框时就不必调用 Create 了，只需调用 CWnd::ShowWindow(SW_SHOW)来显示对话框．这样做的好处在于对话框中的数据可以保存下来，供以后使用．由于拥有者窗口在被关闭时会调用 DestroyWindow 删除每一个所属窗口，故只要非模态对话框是自动清除的，程序员就不必担心对话框对象的删除问题．/

5.5 标签式对话框

在设计较为复杂的对话框时，常常会遇到这种情况：对某一事物的设置或选项需要用到大量的控件，以至于一个对话框放不下，而这些控件描述的是类似的属性，不能分开。用普通的对话框技术，这一问题很难解决。

MFC 提供了对标签式对话框的支持，可以很好的解决上述问题。标签式对话框实际上是一个包含了多个子对话框的对话框，这些子对话框通常被称为页(Page)。每次只有一个页是可见的，在对话框的顶端有一行标签，用户通过单击这些标签可切换到不同的页。显然，标签式对话框可以容纳大量的控件。在象 Word 和 Developer Studio 这样复杂的软件中，用户会接触到较多的标签式对话框，一个典型的标签式对话框如图 5.10 所示。

图 5.10 典型的标签式对话框

5.5.1 标签式对话框的创建

为了支持标签式对话框，MFC 提供了 CPropertySheet 类和 CPropertyPage 类。前者代表对话框的框架，后者代表对话框中的某一页。CPropertyPage 是 CDialog 类的派生类，而 CPropertySheet 是 CWnd 类的派生类。虽然 CPropertySheet 不是 CDialog 类的派生类，但使用 CPropertySheet 对象的方法与使用 CDialog 对象是类似的。标签式对话框是一种特殊的对话框，因此，和普通对话框相比，它的设计与实现既有许多相似之处，又有一些不同的特点。

创建一个标签式对话框一般包括以下几个步骤：

分别为各个页创建对话框模板，去掉缺省的 OK 和 Cancel 按钮。每页的模板最好具有相同的尺寸，如果尺寸不统一，则框架将根据最大的页来确定标签对话框的大小。在创建模板时，需要在模板属性对话框中指定下列属性：

指定标题(Caption)的内容。标题的内容将显示在该页对应的标签中。

选择 TitleBar、Child、ThinBorder 和 Disable 属性。

根据各个页的模板，用 ClassWizard 分别为每个页创建 CPropertyPage

类的派生类。这一过程与创建普通对话框类的过程类似，不同的是在创建新类对话框中应在 Base class 一栏中选择 CPropertyPage 而不是 CDialog。

用 ClassWizard 为每页加入与控件对应的成员变量，这个过程与为普通对话框类加入成员变量类似。

程序员可直接使用 CPropertySheet 类，也可以从该类派生一个新类。除非要创建一个非模态对话框，或要在框架对话框中加入控件，否则没有必要派生一个新类。如果直接使用 CPropertySheet 类，则一个典型的标签式对话框的创建代码如清单 5.12 所示，该段代码也演示了标签式对话框与外界的数据交换。这些代码通常是放在显示对话框的命令处理函数中。可以看出，对话框框架的创建过程及对话框与外界的数据交换机制与普通对话框是一样的，不同之处是还需将页对象加入到 CPropertySheet 对象中。如果要创建的是模态对话框，应调用 CPropertySheet::DoModal，如果想创建非模态对话框，则应该调用 CPropertySheet::Create。

若从 CPropertySheet 类派生了一个新类，则应该将所有的页对象以成员变量的形式嵌入到派生类中，并在派生类的构造函数中调用 CPropertySheet::AddPage 函数来把各个页添加到对话框中。这样，在创建标签式对话框时就不用做添加页的工作了。

清单 5.12 典型的标签式对话框创建代码

```
void CMyView::DoModalPropertySheet()
{
CPropertySheet propsheet;
CMyFirstPage pageFirst; // derived from CPropertyPage
CMySecondPage pageSecond; // derived from CPropertyPage
// Move member data from the view (or from the currently
// selected object in the view, for example).
pageFirst.m_nMember1 = m_nMember1;
pageFirst.m_nMember2 = m_nMember2;
pageSecond.m_strMember3 = m_strMember3;
pageSecond.m_strMember4 = m_strMember4;
propsheet.AddPage(&pageFirst);
propsheet.AddPage(&pageSecond);
if (propsheet.DoModal() == IDOK)
{
m_nMember1 = pageFirst.m_nMember1;
m_nMember2 = pageFirst.m_nMember2;
m_strMember3 = pageSecond.m_strMember3;
m_strMember4 = pageSecond.m_strMember4;
. . .
}
}
```

5.5.2 标签式对话框的运行机制

标签式对话框的初始化包括框架对话框的初始化和页的初始化。页的初始化工作可在 OnInitDialog 函数中进行，而框架对话框的初始化应该在 OnCreate 函数中完成。

根据 CPropertySheet::DoModal 返回的是 IDOK 还是 IDCANCEL，程序可判断出关闭对话框时按的是 OK 还是 Cancel 按钮，这与普通对话框是一样的。

如果标签式对话框是模态对话框，在其底部会有三个按钮，依次为 OK、Cancel 和 Apply(应用)按钮，如果对话框是非模态的，则没有这些按钮。OK 和 Cancel 按钮的意义与普通对话框没什么两样，Apply 按钮则是标签对话框所特有的。普通的模态对话框只有在用户按下了 OK 按钮返回后，对话框的设置才能生效，而设计 Apply 按钮的意图是让用户能在不关闭对话框的情况下使对话框中的设置生效。由此可见，Apply 的作用与前面例子中登录数据的“添加”按钮类似，用户不必退出对话框，就可以反复进行设置，这在某些应用场合下是很有用的。

为了对上述三个按钮作出响应，CPropertyPage 类提供了 OnOK、OnCancel 和 OnApply 函数，用户可覆盖这三个函数以完成所需的工作。需要指出的是这三个函数并不是直接响应按钮的 BN_CLICKED 消息的，但在按钮按下后它们会被间接调用。这些函数的说明如下：

virtual void OnOK( );在按下 OK 或 Apply 按钮后，该函数将被调用。缺省的 OnOK 函数几乎什么也不干，象数据交换和关闭对话框这样的工作是在别的地方完成的，这与普通对话框的 OnOK 函数是不同的。

virtual void OnCancel( );在按下 Cancel 按钮后，该函数将被调用。缺省的 OnCancel 函数也是几乎什么都不干。

virtual BOOL OnApply( );在按下 OK 或 Apply 按钮后，该函数将被调用。缺省的 OnApply 会调用 OnOK 函数。函数的返回值如果是 TRUE，则对话框中的设置将生效，否则无效。

按理说，CPropertySheet 类也应该提供上述函数，特别是 OnApply。但奇怪的是，MFC 并未考虑 CPropertySheet 类的按钮响应问题。读者不要指望能通过 ClassWizard 来自动创建按钮的 BN_CLICKED 消息处理函数，如果需要用到这类函数，那么只好手工创建了。

下列几个 CPropertyPage 类的成员函数也与标签对话框的运行机制相关。

void SetModified(BOOL bChanged = TRUE);该函数用来设置修改标志。若参数 bChanged 为 TRUE，则表明对话框中的设置已改动，否则说明设置未改动。该函数的一个主要用途是允许或禁止 Apply 按钮。在缺省情况下，Apply 按钮是禁止的。只要一调用 SetModified(TRUE)，Apply 按钮就被允许，而调用 SetModified(FALSE)并不一定能使 Apply 按钮禁止，只有在所有被标为改动过的页都调用了 SetModified(FALSE)后，Apply 按钮才会被禁止。另外，该函数对 OnApply 的调用也有影响，当 Apply 按钮被按下后，只有那些被标为改动过的页的 OnApply 函数才会被调用。在调用该函数之前，程序需要判断页中的内容是否已被修改，

可以通过处理诸如 BN_CLICKED、EN_CHANG 这样的控件通知消息来感知页的内容的改变。

virtual BOOL OnSetActive( );当页被激活或被创建时，都会调用该函数。该函数的缺省行为是若页还未创建，就创建之，若页已经创建，则将其激活，并调用 UpdateData(FALSE)更新控件。用户可覆盖该函数完成一些刷新方面的工作。

virtual BOOL OnKillActive( );当原来可见的页被覆盖或被删除时，都会调用该函数。该函数的缺省行为是调用 UpdateData(TRUE)更新数据。用户可覆盖该函数完成一些特殊数据的有效性检查工作。

需要说明的是，标签对话框中的所有页不一定都会被创建。实际上，那些从未打开过的页及其控件是不会被创建的。因此，在 CPropertyPage 类的派生类中，只有在确定了页已存在后，才能调用与对话框及控件相关的函数(如 UpdateData)。如果收到控件通知消息，或 OnSetActive 函数被调用，则说明页已经存在。正是由于上述原因，使得标签式对话框的内部数据交换只能在 OnSetActive 和 OnKillActive 函数中进行。

5.5.3 标签式对话框的具体实例

通过上面的分析，读者对标签式对话框已经比较了解了。现在，让我们在前面做过的 Register 程序中加入一个标签式对话框来试验一下其功能。

在 Register 程序的登录数据对话框中有“个人情况”和“单位情况”两组控件，显然，我们可以创建一个标签式对话框并把两组控件分别放到两个页中。为了简单起见，我们仅要求输入姓名和单位名，简化后的标签式对话框如图 5.11 所示。

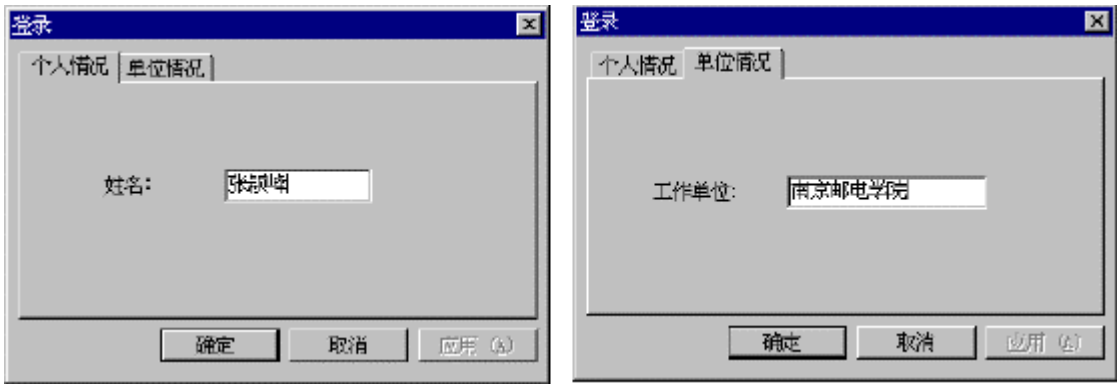


图 5.11 简化后的标签式对话框

通过对标签式对话框的分析，读者已经知道 CPropertySheet 类未对 Apply 按钮的控件通知消息进行处理，这是一个不足之处。Register 的新版本将向读者演示如何在 CPropertySheet 类的派生类中手工加入 Apply 按钮的 BN_CLICKED 消息处理函数。另外，新版本还演示了对话框与外部对象交流的一种较好办法，即通过发送用户定义消息来向外部对象传递信息。在登录数据对话框中，与外界交流的方法是在对话框内部直接访问派生的视图对象，这样做的优点是方便快捷，缺点则是对外界依赖较大，不利于移植。而用发送用户定义消息的方法则可以避免这个缺点。

具体工作请按下面几步进行：

在菜单资源中的 Edit 菜单的“登录数据...”项的后面插入一个名为“标签式对话框...”的菜单项，并指定其 ID 为 ID_EDIT_PROPDLG。然后用 ClassWizard，在 CRegisterView 类内为该菜单命令创建命令处理函数 OnEditPropdlg，该函数将用来显示标签式对话框。

为标签式对话框的第一页创建对话框模板。去掉缺省的 OK 和 Cancel 按钮。注意应选择中文语种和宋体字体。在属性对话框中，指定对话框的 ID 为 IDD_PERSONAL，标题为“个人情况”，在 Styles 页中，选中 TitleBar 项，并在 Style 栏中选择 Child，在 Border 栏中选择 ThinBorder。在 More Styles 页中，选中 Disable。然后，在模板中加入控件，如图 5.11 和表 5.6 所示。

表 5.6

控件类型	控件 ID	控件标题
静态正文	缺省	姓名：
编辑框	IDC_NAME	

用 ClassWizard 为模板 IDD_PERSONAL 创建 CPropertyPage 类的派生类，类名为 CPersonalPage。在该类中为控件 IDC_NAME 加入对应的成员变量，变量名为 m_strName，类型为 CString。为控件 IDC_NAME 加入 EN_CHANGE 消息处理函数 OnChangeName，当编辑框的内容被改变时，控件会向对话框发出 EN_CHANGE 消息。在 OnChangeName 中，应该使 Apply 按钮允许。

仿照步 2，为标签式对话框的第二页创建对话框模板。指定其 ID 为 IDD_UNIT，标题为“单位情况”。在模板中加入的控件如图 5.11 和表 5.7 所示。

表 5.7

控件类型	控件 ID	控件标题
静态正文	缺省	工作单位：
编辑框	IDC_UNIT	

用 ClassWizard 为模板 IDD_UNIT 创建 CPropertyPage 类的派生类，类名为 CUnitPage。在该类中为控件 IDC_UNIT 加入对应的成员变量，变量名为 m_strUnit，类型为 CString。为控件 IDC_UNIT 加入 EN_CHANGE 消息处理函数 OnChangeUnit。

用 ClassWizard 创建一个 CPropertySheet 的派生类，类名为 CRegisterSheet。

在 CRegisterApp 类的头文件的开头加入下面一行 #define WM_USER_OUTPUT (WM_USER+200)WM_USER_OUTPUT 不是标准的 Windows 消息，而是一个用户定义消息。在本例中，当标签式对话框的 Apply 按钮被按下后，程序会向编辑视图发送该消息，编辑视图对应的消息处理函数应该输出对话框的数据。用户定义消息的编码范围是 WM_USER－0x7FFF。

请读者按清单 5.13、5.14、5.15 修改程序，限于篇幅，这里仅列出了需要修改的部分源代码。

清单 5.13 CPersonalPage 类和 CUnitPage 类的部分代码

```
void CPersonalPage::OnChangeName()
{
// TODO: Add your control notification handler code here
SetModified(TRUE); //使 Apply 按钮允许
UpdateData(TRUE);
}
void CUnitPage::OnChangeUnit()
{
// TODO: Add your control notification handler code here
SetModified(TRUE); //使 Apply 按钮允许
UpdateData(TRUE);
}
```

当页中的编辑框的内容被改变时，页会收到 EN_CHANGE **消息，这将导致** OnChangeName **或** OnChangeUnit **被调用。对该消息的处理是使** Apply **按钮允许并调用** UpdateData(TRUE)**更新数据。**

清单 5.14 CRegisterSheet **类的部分代码**

```
//文件 RegisterSheet.h
class CRegisterSheet : public CPropertySheet
{
. . . . . .
// Construction
public:
CRegisterSheet(UINT nIDCaption, CWnd* pParentWnd = NULL, UINT iSelectPage = 0);
CRegisterSheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL, UINT iSelectPage = 0);
public:
CPersonalPage m_PersonalPage;
CUnitPage m_UnitPage;
. . . . . .
protected:
//{{AFX_MSG(CRegisterSheet)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG
afx_msg void OnApplyNow();
DECLARE_MESSAGE_MAP()
};
//文件 RegisterSheet.cpp
#include “stdafx.h”
#include “Register.h”
#include “PersonalPage.h”
#include “UnitPage.h”
```

```
#include “RegisterSheet.h”
CRegisterSheet::CRegisterSheet(LPCTSTR pszCaption, CWnd* pParentWnd, UINT iSelectPage)
:CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
AddPage(&m_PersonalPage); //向标签对话框中添加页
AddPage(&m_UnitPage);
}
BEGIN_MESSAGE_MAP(CRegisterSheet, CPropertySheet)
//{{AFX_MSG_MAP(CRegisterSheet)
// NOTE - the ClassWizard will add and remove mapping macros here.
//}}AFX_MSG_MAP
ON_BN_CLICKED(ID_APPLY_NOW, OnApplyNow)
END_MESSAGE_MAP()
void CRegisterSheet::OnApplyNow()
{
CFrameWnd* pFrameWnd = (CFrameWnd*) AfxGetMainWnd();
//获取指向视图的指针
CView* pView = pFrameWnd->GetActiveFrame()->GetActiveView();
//发送用户定义消息，在视图中输出信息
pView->SendMessage(WM_USER_OUTPUT, (WPARAM)this);
m_PersonalPage.SetModified(FALSE);
m_UnitPage.SetModified(FALSE); //使 Apply 按钮禁止
}
```

在 CRegisterSheet 类内嵌入了 CPersonalPage 和 CUnitPage 对象，在该类的构造函数中调用 CPropertySheet::AddPage 将两个页添加到对话框中。

标签式对话框的 OK、Cancel 和 Apply 按钮的 ID 分别是 IDOK、IDCANCEL 和 ID_APPLY_NOW。在按下 Apply 按钮后，CRegisterSheet 对象应该作出响应，由于 ClassWizard 不能为 CRegisterSheet 类提供 Apply 按钮的 BN_CLICKED 消息处理函数，故必须手工声明和定义消息处理函数 OnApplyNow，并在消息映射表中手工加入 ID_APPLY_NOW 的 BN_CLICKED 消息映射，该映射是通过 ON_BN_CLICKED 宏实现的。

函数 OnApplyNow 用 CWnd::SendMessage 向视图发送用户定义消息 WM_USER_OUTPUT，并调用 CPropertyPage::SetModified(FALSE)来禁止 Apply 按钮。在发送消息时，将 this 指针作为 wParam 参数一并发送，这是因为视图对象需要指向 CRegisterSheet 对象的指针来访问该对象。该函数演示了如何在程序的任意地方获得当前活动视图的方法：首先，调用 AfxGetMainWnd()返回程序主窗口的 CWnd 类指针，然后将该指针强制转换成 CFrameWnd 类型，接着调用 CFrameWnd::GetActiveFrame 返回当前活动的框架窗口的一个 CFrameWnd 型指针，最后调用 CFrameWnd::GetActiveView 返回当前活动视图的一个 Cview 型指针。

在函数 OnApplyNow **中主要调用了下列函数：**

CWnd* AfxGetMainWnd( );**该函数返回一个指向程序的主窗口** CWnd **指针。程序的主窗口可以是一个框架窗口，也可以是一个对话框。**

virtual CFrameWnd* GetActiveFrame( );**函数返回一个** CFrameWnd **型的指针。如果是** MDI(**多文档界面**)**程序，则该函数将返回当前活动的子框架窗口，如果是** SDI(**单文档界面**)**程序，该函数将返回主框架窗口本身。**

CView* GetActiveView( ) const;**返回一个指向当前活动视图的** Cview **型指针。**

LRESULT SendMessage(UINT message, WPARAM wParam = 0, LPARAM lParam = 0);**用于向本窗口发送消息。**SendMessage **会直接调用发送消息的处理函数，直到发送消息被处理完后该函数才返回。参数** message **说明了要发送的消息，**wParam **和** lParam **则提供了消息的附加信息。**

清单 5.15 CRegisterView **类的部分代码**

```
//文件 RegisterView.h
class CRegisterView : public CEditView
{
. . . . . .
// Generated message map functions
protected:
//{{AFX_MSG(CRegisterView)
afx_msg void OnEditRegister();
afx_msg void OnEditPropdlg();
//}}AFX_MSG
afx_msg LRESULT OnOutput(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};
//文件 RegisterView.cpp
#include “stdafx.h”
#include “Register.h”
#include “RegisterDoc.h”
#include “RegisterView.h”
#include “RegisterDialog.h”
#include “PersonalPage.h”
#include “UnitPage.h”
#include “RegisterSheet.h”
BEGIN_MESSAGE_MAP(CRegisterView, CEditView)
. . . . . .
ON_MESSAGE(WM_USER_OUTPUT, OnOutput)
END_MESSAGE_MAP()
void CRegisterView::OnEditPropdlg()
{
```

```
// TODO: Add your command handler code here
CRegisterSheet RegisterSheet("登录");
RegisterSheet.m_PersonalPage.m_strName="张颖峰";
RegisterSheet.m_UnitPage.m_strUnit="南京邮电学院";
if(RegisterSheet.DoModal()==IDOK)
OnOutput((WPARAM)&RegisterSheet,0);
}
//用户定义消息 WM_USER_OUTPUT 的处理函数
LRESULT CRegisterView::OnOutput(WPARAM wParam, LPARAM lParam)
{
CRegisterSheet *pSheet=(CRegisterSheet*)wParam;
CString str;
GetWindowText(str);
str+="\r\n";
str+="姓名:";
str+=pSheet->m_PersonalPage.m_strName;
str+="\r\n";
str+="工作单位:";
str+=pSheet->m_UnitPage.m_strUnit;
str+="\r\n";
SetWindowText(str);
return 0;

}
```

OnEditPropdlg 函数负责初始化和创建标签式对话框，这一过程与创建普通对话框差不多。如果用户是按 OK 按钮返回的，则调用 OnOutput 函数输出数据。

CRegisterView 类的 OnOutput 函数负责处理标签对话框发来的用户定义消息 WM_USER_OUTPUT。用户定义消息的处理函数只能用手工的方法加入。用户定义消息的消息映射是用 ON_MESSAGE 宏来完成的。

函数 OnOutput 的两个参数 wParam 和 lParam 分别对应消息的 wParam 和 lParam 值。该函数从 wParam 参数中获得指向 CRegisterSheet 对象的指针，然后将该对象中的数据输出到视图中。

编译并运行 Register，试一试自己设计的标签式对话框。

5.6 公用对话框

在使用 Windows 的过程中，用户经常会遇到一些常用的有特定用途的对话框。例如，当选择 File->Open，会弹出一个文件选择的对话框，用户可以在其中选择想要打开的文件。象文件选择这样的对话框，使用的非常普遍，因此 Windows 系统本身提供了对该对话框的支持，用户不必自己设计文件选择对话框。与文件选择对话框类似的还有颜色选择、字体选择、打印和打印设置以及正文搜索和替换对话框。这五种对话框均由 Windows 支持，被称为公用对话框。

MFC 提供了一些公用对话框类，它们均是 CDialog 类的派生类，封装了公用对话框的功能。表 5.6 列出了 MFC 的公用对话框类。

表 5.6 公用对话框类

通用对话框类	用途
CColorDialog	选择颜色
CFileDialog	选择文件名，用于打开和保存文件
CFindReplaceDialog	正文查找和替换
CFontDialog	选择字体
CPrintDialog	打印和打印设置

通用对话框类使用方便，读者只需知道怎样创建对话框和访问对话框的数据，不必关心它们的内部细节。

5.6.1 CColorDialog 类

CColorDialog 类用于实现 Color(颜色选择)公用对话框。Color 对话框如图 5.12 所示，在 Windows 的画板程序中，如果用户在颜色面板的某种颜色上双击鼠标，就会显示一个 Color 对话框来让用户选择颜色。

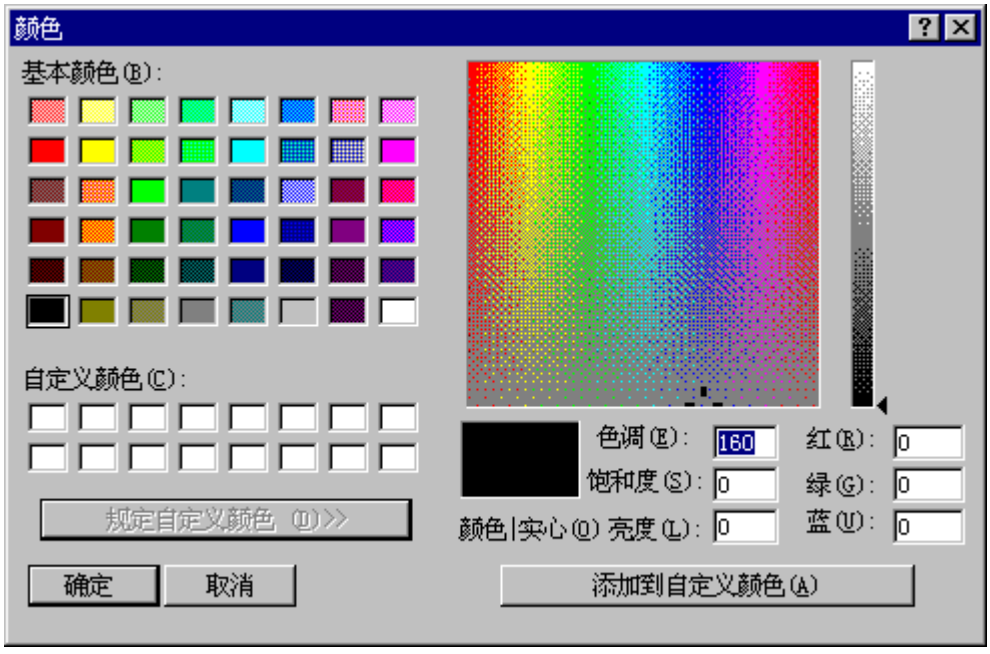


图 5.12 Color 对话框

Color 对话框的创建与一般的对话框没什么两样：首先是在堆栈上构建一个 CColorDialog 对象，然后调用 CColorDialog::DoModal( )来启动对

话框。CColorDialog 的构造函数为

CColorDialog(COLORREF clrInit = 0, DWORD dwFlags = 0, CWnd* pParentWnd = NULL);

参数 clrInit 用来指定初始的颜色选择，dwFlags 用来设置对话框，pParentWnd 用于指定对话框的父窗口或拥有者窗口。

根据 DoModal 返回的是 IDOK 还是 IDCANCEL 可知道用户是否确认了对颜色的选择。DoModal 返回后，调用 CColorDialog::GetColor()可以返回一个 COLORREF 类型的结果来指示在对话框中选择的颜色。COLORREF 是一个 32 位的值，用来说明一个 RGB 颜色。GetColor 返回的 COLORREF 的格式是 0x00bbggrr，即低位三个字节分别包含了蓝、绿、红三种颜色的强度。

读者将在后面的章节中看到颜色选择对话框的例子。

5.6.2 CFileDialog 类

CFileDialog 类用于实现文件选择对话框，以支持文件的打开和保存操作。用户要打开或保存文件，就会和文件选择对话框打交道，图 5.13 显示了一个标准的用于打开文件的文件选择对话框。用 MFC AppWizard 建立的应用程序中自动加入了文件选择对话框，在 File 菜单选 Open 或 Save As 命令会启动它们。

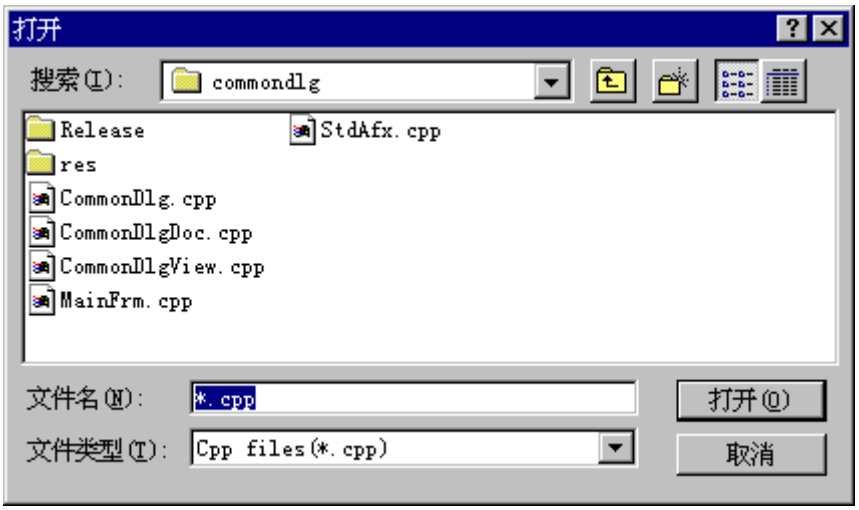


图 5.13 文件选择对话框

文件选择对话框的创建过程与一般对话框的类似，首先是在堆栈上构建一个 CFileDialog 对象，然后调用 CFileDialog::DoModal( )来启动对话框。文件对话框的构造函数为

CFileDialog(BOOL bOpenFileDialog, LPCTSTR lpszDefExt = NULL, LPCTSTR lpszFileName = NULL, DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, LPCTSTR lpszFilter = NULL, CWnd* pParentWnd = NULL);

如果参数 bOpenFileDialog 的值为 TRUE，将创建 Open(打开文件)对话框，否则就创建 Save As(保存文件)对话框。参数 lpszDefExt 用来指定缺省的文件扩展名。lpszFileName 用于规定初始文件名。dwFlags 用于设置对话框的一些属性。lpszFilter 指向一个过滤字符串，用户如果只想选

择某种或某几种类型的文件，就需要指定过滤字符串。参数 pParentWnd 是指向父窗口或拥有者窗口的指针。

过滤字符串有特定的格式，它实际上是由多个子串组成，每个子串由两部分组成，第一部分是过滤器的字面说明，如“ Text file (*.txt) ”，第二部分是用于过滤的匹配字符串，如“ *.txt ”，子串的两部分用竖线字符“ | ”分隔开。各子串之间也要用“ | ”分隔，且整个串的最后两个字符必须是两个连续的竖线字符“ || ”。一个典型的过滤字符串如下面所示：

char szFilter[]=

“ All files (*.*)|*.*|Text files(*.txt)|*.txt|Word documents(*.doc)|*.doc|| ”;

若 CFileDialog::DoModal 返回的是 IDOK，那么可以用表 5.7 列出的 CFileDialog 类的成员函数来获取与所选文件有关的信息。

表 5.7 CFileDialog 类辅助成员函数

函数名	用途
GetPathName	返回一个包含有全路径文件名的 CString 对象。
GetFileName	返回一个包含有文件名(不含路径)的 CString 对象。
GetFileExt	返回一个只含文件扩展名的 CString 对象。
GetFileTitle	返回一个只含文件名(不含扩展名)的 CString 对象。

5.6.3 CFindReplaceDialog 类

CFindReplaceDialog 类用于实现 Find(搜索)和 Replace(替换)对话框，这两个对话框都是非模态对话框，用于在正文中搜索和替换指定的字符串。图 5.14 显示了一个 Find 对话框，图 5.15 显示了一个 Replace 对话框。

图 5.14 Find 对话框

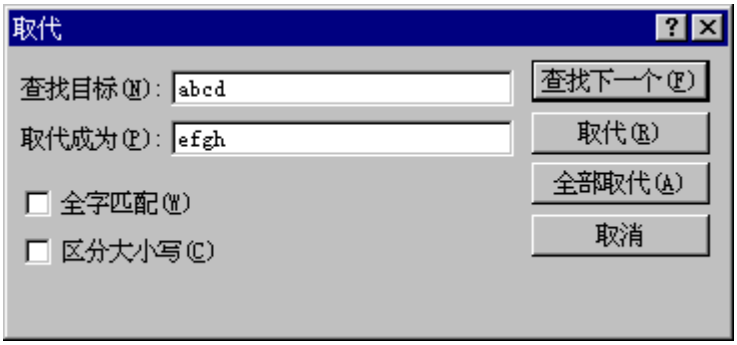


图 5.15 Replace 对话框

由于 Find 和 Replace 对话框是非模式对话框，它们的创建方式与其它四类公用对话框不同。CFindReplaceDialog 对象是用 new 操作符在堆中创建的，而不是象普通对话框那样以变量的形式创建。要启动

Find/Replace 对话框，应该调用 CFindReplaceDialog::Create 函数，而不是 DoModal。Create 函数的声明是

BOOL Create(BOOL bFindDialogOnly, LPCTSTR lpszFindWhat, LPCTSTR lpszReplaceWith = NULL, DWORD dwFlags = FR_DOWN, CWnd* pParentWnd = NULL);

当参数 bFindDialogOnly 的值为 TRUE 时，创建的是 Find 对话框，为 FALSE 时创建的是 Replace 对话框。参数 lpszFindWhat 指定了要搜索的字符串，lpszReplaceWith 指定了用于替换的字符串。dwFlags 用来设置对话框，其缺省值是 FR_DOWN(向下搜索)，该参数可以是几个 FR_XXX 常量的组合，用户可以通过该参数来决定诸如是否要显示 Match case、Match Whole Word 检查框等设置。参数 pParentWnd 指明了对话框的父窗口或拥有者窗口。

Find/Replace 对话框与其它公用对话框的另一个不同之处在于它在工作过程中可以重复同一操作而对话框不被关闭，这就方便了频繁的搜索和替换。CFindReplaceDialog 类只提供了一个界面，它并不会自动实现搜索和替换功能。CFindReplaceDialog 使用了一种特殊的通知机制，当用户按下了操作的按钮后，它会向父窗口发送一个通知消息，父窗口应在该消息的消息处理函数中实现搜索和替换。

CFindReplaceDialog 类提供了一组成员函数用来获得与用户操作有关的信息，如表 5.8 所示，这组函数一般应在通知消息处理函数中调用。

表 5.8 CFindReplaceDialog 类的辅助成员函数

函数名	用途
FindNext	如果用户点击了 Findnext 按钮，该函数返回 TRUE。
GetNotifier	返回一个指向当前 CFindReplaceDialog 对话框的指针。
GetFindString	返回一个包含要搜索字符串的 CString 对象。
GetReplaceString	返回一个包含替换字符串的 CString 对象。
IsTerminating	如果对话框终止了，则返回 TRUE。
MatchCase	如果选择了对话框中的 Match case 检查框，则返回 TRUE。
MatchWholeWord	如果选择了对话框中的 Match Whole Word 检查框，则返回 TRUE。
ReplaceAll	如果用户点击了 Replace All 按钮，该函数返回 TRUE。
ReplaceCurrent	如果用户点击了 Replace 按钮，该函数返回 TRUE。
SearchDown	返回 TRUE 表明搜索方向向下，返回 FALSE 则向上。

CEditView 类自动实现了 Find 和 Replace 对话框的功能，但 MFC AppWizard 并未提供相应的菜单命令。读者可以在前面的 Register 工程的 Edit 菜单中加入 &Find... 和 &Replace... 两项，并令其 ID 分别为 ID_EDIT_FIND 和 ID_EDIT_REPLACE，则 Find/Replace 对话框的功能

就可以实现。

5.6.4 CFontDialog 类

CFontDialog 类支持 Font(字体)对话框，用来让用户选择字体。图 5.16 显示了一个 Font 对话框。Font 对话框的创建过程与 Color 对话框的类似，首先是在堆栈上构建一个 CFontDialog 对象，然后调用 CFontDialog::DoModal 来启动对话框。

图 5.16 Font 对话框

CFontDialog 类的构造函数如下所示

CFontDialog(LPLOGFONT lplfInitial = NULL, DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS, CDC* pdcPrinter = NULL, CWnd* pParentWnd = NULL);

参数 lplfInitial 指向一个 LOGFONG 结构，用来初始化对话框中的字体设置。dwFlags 用于设置对话框。pdcPrinter 指向一个代表打印机的 CDC 对象，若设置该参数，则选择的字体就为打印机所用。pParentWnd 用于指定对话框的父窗口或拥有者窗口。

若 DoModal 返回 IDOK，那么可以调用 CFontDialog 的成员函数来获得所选字体的信息，这些函数在表 5.9 列出。

表 5.9 CFontDialog 类的辅助成员函数

函数名	用途
GetCurrentFont	用来获得所选字体的属性。该函数有一个参数，该参数是指向 LOGFONT 结构的指针，函数将所选字体的各种属性写入这个 LOGFONT 结构中。
GetFaceName	返回一个包含所选字体名字的 CString 对象。
GetStyleName	返回一个包含所选字体风格名字的 CString 对象。
GetSize	返回所选字体的尺寸(以 10 个象素为单位)。
GetColor	返回一个含有所选字体的颜色的

	COLORREF 型值。
GetWeight	返回所选字体的权值。
IsStrikeOut	若用户选择了空心效果则返回 TRUE，否则返回 FALSE。
IsUnderline	若用户选择了下划线效果则返回 TRUE，否则返回 FALSE。
IsBold	若用户选择了黑体风格则返回 TRUE，否则返回 FALSE。
IsItalic	若用户选择了斜体风格则返回 TRUE，否则返回 FALSE。

5.6.5 CPrintDialog 类

CPrintDialog 类支持 Print(打印)和 Print Setup(打印设置)对话框，通过这两个对话框用户可以进行与打印有关的操作。图 5.17 显示了一个 Print 对话框，图 5.18 显示了一个 Print Setup 对话框。

图 5.17 Print 对话框

图 5.18 Print Setup **对话框**

Print **和** Print Setup **对话框的创建过程与** Color **对话框类似。该类的构造函数是**

CPrintDialog(BOOL bPrintSetupOnly, DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE | PD_NOSELECTION, CWnd* pParentWnd = NULL);

参数 bPrintSetupOnly **的值若为** TRUE**，则创建的是** Print **对话框，否则，创建的是** Print Setup **对话框。**dwFlags **用来设置对话框，缺省设置是打印出全部页，禁止** From **和** To **编辑框(即不用确定要打印的页的范围)，**PD_USEDEVMODECOPIES **使对话框判断打印设备是否支持多份拷贝和校对打印**(Collate)**，若不支持，就禁止相应的编辑控件和** Collate **检查框。**pParentWnd **用来指定对话框的父窗口或拥有者窗口。**

程序可以调用如表 5.10 **所示的** CPrintDialog **的成员函数来获得打印参数。**

表 5.10 CPrintDialog **的辅助成员函数**

函数名	**用途**
GetCopies	**返回要求的拷贝数。**
GetDefaults	**在不打开对话框的情况下返回缺省打印机的缺省设置，返回的设置放在** m_pd **数据成员中。**
GetDeviceName	**返回一个包含有打印机设备名的** CString **对象。**
GetDevMode	**返回一个指向** DEVMODE **结构的指针，用来查询打印机的设备初始化信息和设备环境信息。**
GetDriverName	**返回一个包含有打印机驱动程序名的** CString **对象。**
GetFromPage	**返回打印范围的起始页码。**
GetToPage	**返回打印范围的结束页码。**
GetPortName	**返回一个包含有打印机端口名的** CString **对象。**
GetPrinterDC	**返回所选打印设备的一个** HDC **句柄。**

PrintAll	若要打印文档的所有页则返回 TRUE。
PrintCollate	若用户选择了 Collate Copies 检查框(需要校对打印拷贝)则返回 TRUE。
PrintRange	如果用户要打印文档的一部分页,则返回 TRUE。
PrintSelection	若用户想打印当前选择的部分文档，则返回 TRUE。

用缺省配置的 MFC AppWizard 建立的程序支持 Print 和 Print Setup 对话框，用户可以在 File 菜单中启动它们。

5.6.6 公用对话框的使用实例

现在，让我们来测试一下公用对话框的使用。请读者用 AppWizard 创建一个单文档的 MFC 应用程序，名为 CommonDlg。注意别忘了在 AppWizard 的第一步中选 Single document。

CommonDlg 程序要对所有的公用对话框进行了测试。为此，首先要提供用户命令接口。请读者在 CommonDlg 的菜单资源中插入一个名为 &Common 的新菜单，这个菜单插在 Help 菜单之前。然后，在 Common 菜单中，请按表 5.11 创建菜单项。

表 5.11 Common 菜单的菜单项

Caption	ID
&Color...	ID_COMMON_COLOR
&Open file...	ID_COMMON_OPENFILE
&Save file...	ID_COMMON_SAVEFILE
&Font...	ID_COMMON_FONT
&Print...	ID_COMMON_PRINT
P&rint setup...	ID_COMMON_PRINTSETUP
F&ind...	ID_COMMON_FIND
&Replace...	ID_COMMON_REPLACE

接下来的工作是编写测试程序的源代码。首先，利用 ClassWizard 为表 5.11 的菜单项创建消息处理函数，注意这些处理函数都是 CCommonDlgView 的成员。接着，请按清单 5.10 和 5.11 修改程序。限于篇幅，这里仅列出与测试相关的部分源代码。

清单 5.10 头文件 CommonDlgView.h

```
class CCommonDlgView : public CView
{
. . . . . .
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
void DispPrintInfo(CPrintDialog& dlg);
protected:
CFont m_Font; //正文的字体
COLORREF m_ForeColor; //正文的前景色
COLORREF m_BackColor; //正文的背景色
```

```
    CFindReplaceDialog *m_pFindReplaceDlg;
    BOOL m_bFindOnly;
    // Generated message map functions
    protected:
    //Find 和 Replace 对话框通知消息处理函数
    afx_msg LRESULT OnFindReplaceCmd(WPARAM, LPARAM
lParam);
    //{{AFX_MSG(CCommonDlgView)
    afx_msg void OnCommonColor();
    afx_msg void OnCommonFont();
    afx_msg void OnCommonOpenfile();
    afx_msg void OnCommonSavefile();
    afx_msg void OnCommonPrint();
    afx_msg void OnCommonPrintsetup();
    afx_msg void OnCommonFind();
    afx_msg void OnCommonReplace();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
    };
    . . . . . .
    清单 5.11  文件 CCommonDlgView.cpp
    #include “stdafx.h”
    #include “CommonDlg.h”
    #include “CommonDlgDoc.h”
    #include “CommonDlgView.h”
    #ifdef _DEBUG
    #define new DEBUG_NEW
    #undef THIS_FILE
    static char THIS_FILE[] = __FILE__;
    #endif
    IMPLEMENT_DYNCREATE(CCommonDlgView, CView)
    //获取对本进程唯一的消息编号
    static const UINT nMsgFindReplace
= ::RegisterWindowMessage(FINDMSGSTRING);
    BEGIN_MESSAGE_MAP(CCommonDlgView, CView)
    //{{AFX_MSG_MAP(CCommonDlgView)
    ON_COMMAND(ID_COMMON_COLOR, OnCommonColor)
    ON_COMMAND(ID_COMMON_FONT, OnCommonFont)
    ON_COMMAND(ID_COMMON_OPENFILE, OnCommonOpenfile)
    ON_COMMAND(ID_COMMON_SAVEFILE, OnCommonSavefile)
    ON_COMMAND(ID_COMMON_PRINT, OnCommonPrint)
    ON_COMMAND(ID_COMMON_PRINTSETUP,
```

```
OnCommonPrintsetup)
    ON_COMMAND(ID_COMMON_FIND, OnCommonFind)
    ON_COMMAND(ID_COMMON_REPLACE, OnCommonReplace)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW,
CView::OnFilePrintPreview)
    ON_REGISTERED_MESSAGE(nMsgFindReplace,
OnFindReplaceCmd)
    END_MESSAGE_MAP()
    CCommonDlgView::CCommonDlgView()
    {
    // TODO: add construction code here
    //缺省前景色为黑色，背景色为白色，字体为系统字体
    m_ForeColor=0;
    m_BackColor=0xFFFFFF;
    m_Font.CreateStockObject(SYSTEM_FONT);
    m_pFindReplaceDlg=NULL;
    }
    void CCommonDlgView::OnDraw(CDC* pDC)
    {
    CCommonDlgDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    int x,y;
    CFont *pOldFont;
    TEXTMETRIC TM;
    int textHeight;
    //设置正文的字体
    pOldFont=pDC->SelectObject(&m_Font);
    //设置正文的前景色和背景色
    pDC->SetTextColor(m_ForeColor);
    pDC->SetBkColor(m_BackColor);
    //计算每行正文的高度
    pDC->GetTextMetrics(&TM);
    textHeight=TM.tmHeight+TM.tmExternalLeading;
    //输出正文
    x=5;y=5;
    pDC->TextOut(x,y,”ABCDEFG”);
    y+=textHeight;
```

```
pDC->TextOut(x,y,”abcdefg”);
//恢复原来的字体
pDC->SelectObject(pOldFont);
}
void CCommonDlgView::OnCommonColor()
{
// TODO: Add your command handler code here
CColorDialog dlg;
if(dlg.DoModal()==IDOK)
{
m_BackColor=dlg.GetColor();
//重绘视图
Invalidate();
UpdateWindow();
}
}
void CCommonDlgView::OnCommonFont()
{
// TODO: Add your command handler code here
CFontDialog dlg;
if(dlg.DoModal()==IDOK)
{
LOGFONT LF;
//获取所选字体的信息
dlg.GetCurrentFont(&LF);
m_ForeColor=dlg.GetColor();
//建立新的字体
m_Font.DeleteObject();
m_Font.CreateFontIndirect(&LF);
Invalidate();
UpdateWindow();
}
}
void CCommonDlgView::OnCommonOpenfile()
{
// TODO: Add your command handler code here
//过滤字符串
char szFileFilter[]=
“Cpp files(*.cpp)|*.cpp|”
“Header files(*.h)|*.h|”
“All files(*.*)|*.*||”;
CFileDialog dlg(TRUE, //Open 对话框
```

```
“cpp”, //缺省扩展名
“*.cpp”,
OFN_HIDEREADONLY|OFN_FILEMUSTEXIST, //文件必须存在
szFileFilter,
this);
if(dlg.DoModal()==IDOK)
{
CString str=“The full path name is:”;
str+=dlg.GetPathName();
AfxMessageBox(str);
}
}
void CCommonDlgView::OnCommonSavefile()
{
// TODO: Add your command handler code here
char szFileFilter[]=
“Cpp files(*.cpp)|*.cpp|”
“Header files(*.h)|*.h|”
“All files(*.*)|*.*||”;
CFileDialog dlg(FALSE, //Save 对话框
“cpp”,
“*.cpp”,
OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
szFileFilter,
this);
if(dlg.DoModal()==IDOK)
{
CString str=“The file name is:”;
str+=dlg.GetFileName();
AfxMessageBox(str);
}
}
void CCommonDlgView::OnCommonPrint()
{
// TODO: Add your command handler code here
CPrintDialog dlg(FALSE, PD_ALLPAGES); //Print 对话框
//设置 Print 对话框的属性
dlg.m_pd.nCopies=2;
dlg.m_pd.nMinPage=1;
dlg.m_pd.nMaxPage=50;
dlg.m_pd.nFromPage=1;
dlg.m_pd.nToPage=50;
```

```
if(dlg.DoModal()==IDOK)
DispPrintInfo(dlg);
}
void CCommonDlgView::OnCommonPrintsetup()
{
// TODO: Add your command handler code here
CPrintDialog dlg(TRUE) //Print Setup 对话框
if(dlg.DoModal()==IDOK)
DispPrintInfo(dlg);
}
void CCommonDlgView::DispPrintInfo(CPrintDialog& dlg)
{
CString str;
CString temp;
str+="Driver name:";
str+=dlg.GetDriverName();
str+="\nDevice name:";
str+=dlg.GetDeviceName();
str+="\nPort name:";
str+=dlg.GetPortName();
str+="\nNumber of copies:";
temp.Format("%d",dlg.GetCopies());
str+=temp;
str+="\nCollate:";
str+=dlg.PrintCollate()""Yes":"No";
str+="\nPrint all:";
str+=dlg.PrintAll()""Yes":"No";
str+="\nPrint range:";
str+=dlg.PrintRange()""Yes":"No";
str+="\nSelection:";
str+=dlg.PrintSelection()""Yes":"No";
str+="\nFrom page:";
temp.Format("%d",dlg.GetFromPage());
str+=temp;
str+="\nTo page:";
temp.Format("%d",dlg.GetToPage());
str+=temp;
AfxMessageBox(str);
}
void CCommonDlgView::OnCommonFind()
{
// TODO: Add your command handler code here
```

```
//判断是否已存在一个对话框
if(m_pFindReplaceDlg)
{
if(m_bFindOnly)
{
//若 Find 对话框已打开，则使之成为活动窗口
m_pFindReplaceDlg->SetActiveWindow();
return;
}
else
//关闭 Replace 对话框
m_pFindReplaceDlg->SendMessage(WM_CLOSE);
}
m_bFindOnly=TRUE;
//创建 Find 对话框
m_pFindReplaceDlg=new CFindReplaceDialog;
m_pFindReplaceDlg->Create(TRUE,NULL,NULL,FR_DOWN,this);
}
void CCommonDlgView::OnCommonReplace()
{
// TODO: Add your command handler code here
//判断是否已存在一个对话框
if(m_pFindReplaceDlg)
{
if(!m_bFindOnly)
{
//若 Replace 对话框已打开，则使之成为活动窗口
m_pFindReplaceDlg->SetActiveWindow();
return;
}
else
//关闭 Find 对话框
m_pFindReplaceDlg->SendMessage(WM_CLOSE);
}
m_bFindOnly=FALSE;
//创建 Replace 对话框
m_pFindReplaceDlg=new CFindReplaceDialog;
m_pFindReplaceDlg->Create(FALSE,NULL,NULL,FR_DOWN,this);
}
//Find 和 Replace 对话框通知消息处理函数
LRESULT CCommonDlgView::OnFindReplaceCmd(WPARAM,
LPARAM lParam)
```

```
{
//判断对话框是否被关闭
if(m_pFindReplaceDlg->IsTerminating())
m_pFindReplaceDlg=NULL;
return 0;
}
```

让我们先来看看对 Color 对话框的测试。在 CCommonDlgView::OnCommonColor 中创建了一个 Color 对话框，在此处该对话框的用途是为视图中显示的正文指定背景色。在 CCommonDlgView 的构造函数中将背景色 m_BackColor 的初值设置为白色(0x00000000)。若 DoModal 返回 IDOK，则调用 CColorDialog::GetColor 获取用户选择的颜色并将之保存在 m_BackColor 成员中。然后，调用 Invalidate 和 UpdateWindow 函数以重绘视图。这两个函数的说明如下：

void Invalidate(BOOL bErase = TRUE);该函数的作用是使整个窗口客户区无效。窗口的客户区无效意味着需要重绘，例如，如果一个被其它窗口遮住的窗口变成了前台窗口，那么原来被遮住的部分就是无效的，需要重绘。这时 Windows 会在应用程序的消息队列中放置 WM_PAINT 消息。MFC 为窗口类提供了 WM_PAINT 的消息处理函数 OnPaint，OnPaint 负责重绘窗口。视图类有一些例外，在视图类的 OnPaint 函数中调用了 OnDraw 函数，实际的重绘工作由 OnDraw 来完成。参数 bErase 为 TRUE 时，重绘区域内的背景将被擦除，否则，背景将保持不变。

void UpdateWindow( );该函数的作用是使窗口立即重绘。调用 Invalidate 等函数后窗口不会立即重绘，这是由于 WM_PAINT 消息的优先级很低，它需要等消息队列中的其它消息发送完后才能被处理。调用 UpdateWindow 函数可使 WM_PAINT 被直接发送到目标窗口，从而导致窗口立即重绘。

在 CCommonView::OnDraw 函数中调用了 CDC::SetBkColor 来设置背景色。CDC 类用于绘图，在后面的几章里将会对其作详细介绍。CDC::TextOut 函数用于输出正文。两个函数的说明如下：

virtual COLORREF SetBkColor(COLORREF crColor);用于设置背景色。参数 crColor 指定了背景色的 RGB 值。返回的是原来的背景色。

BOOL TextOut(int x, int y, const CString& str);在指定的位置输出正文。参数 x 和 y 指定了输出起点的横向和纵向坐标。str 参数是输出的字符串。若该函数调用成功则返回 TRUE。

对文件选择对话框的测试比较简单。在 CCommonDlgView::OnCommonOpenfile 和 CCommonDlgView:OnCommonSavefile 函数中，分别创建了一个 Open 对话框和一个 Save 对话框。在创建 Open 对话框时，在 CFileDialog 的构造函数中规定了 OFN_FILEMUSTEXIST 属性，这样当用户试图打开一个不存在的文件时，对话框会发出错误信息并让用户从新选择文件。在创建 Save 对话框时，在 CFileDialog 的构造函数中规定了 OFN_OVERWRITEPROMPT 属性，这样，当用户试图覆盖一个已存在的

文件时，对话框会询问用户是否真的要覆盖该文件。

若用户确认了对文件的选择，那么在文件选择对话框关闭后，程序会将所选文件的文件名或全路径文件名输出到屏幕上。

Find 和 Replace 对话框的创建工作分别由 CCommonDlgView::OnCommonFind 和 CCommonDlgView::OnCommonReplace 完成。

在 OnCommonFind 函数中，首先判断是否已经打开了一个 Find/Replace 对话框。这个判断是完全必要的，因为 Find/Replace 对话框是非模态对话框，打开一个对话框后，用户有可能通过菜单再次执行 Find 或 Replace 命令。成员 m_pFindReplaceDlg 是指向 CFindReplaceDialog 对象的指针，若该指针不为空，则说明对话框已打开。接着，根据成员 m_bFindOnly 来判断原先打开的是否是 Find 对话框，如果原先打开的是一个 Find 对话框，则此时不必创建新的对话框，只需激活已打开的 Find 对话框就行了；如果原先打开的是一个 Replace 对话框，则应该先关闭该对话框，然后再创建 Find 对话框。然后，给 m_bFindOnly 赋 TRUE 值，以表明现在打开的是一个 Find 对话框。最后，创建一个非模态的 Find 对话框，请注意其过程与创建模态对话框的不同之处：

对话框对象是用 new 操作符在堆上创建的，而不是以变量的形式创建。

对话框的启动是靠调用 Create 函数实现的，而不是 DoModal 函数。

调用 CWnd::SetActiveWindow 以激活窗口。调用 CWnd::SendMessage(WM_CLOSE)来关闭窗口，这是因为 WM_CLOSE 消息会导致 CWnd::DestroyWindow 函数的调用。

OnCommonReplace 函数的过程与 OnCommonFind 函数类似。在该函数中，对 m_bFindOnly 赋值 FALSE 以表明打开的是 Replace 对话框。

Find/Replace 对话框通知消息的处理函数是 CCommonDlgView::OnFindReplaceCmd，这个消息处理函数及消息映射均是手工加入的。请注意在 CommonDlgView.cpp 文件的开头部分定义了一个静态全局变量 nMsgFindReplace

static const UINT nMsgFindReplace = :: RegisterWindowMessage(FINDMSGSTRING);

nMsgFindReplace 变量用于存放消息码，这个消息是由函数 RegisterWindowMessage 提供的，该函数的声明为

UINT RegisterWindowMessage(LPCTSTR lpString);

参数 lpString 是一个消息字符串。调用 RegisterWindowMessage 函数会返回一个 Windows 注册消息，注册消息的编码在系统中是唯一的。当有多个应用程序需要处理同一个消息时，应调用该函数注册消息。如果消息是本应用程序专有的，则不必注册。如果两个应用程序使用相同的字符串注册消息，则会返回相同的消息，这样，通过该消息，两个应用程序可以进行通信。

注册消息的消息映射宏是 ON_REGISTERED_MESSAGE，在 CommonDlgView 的消息映射中可以找到它。

在函数 OnFindReplaceCmd 中应该进行实际的搜索和替换工作，但在本例中该函数什么工作也不作。该函数只是判断一下对话框是否被关闭，若是，则给 m_pFindReplaceDlg 赋 NULL 值，以表明对话框已不存在了。

Font 对话框的创建由函数 CCommonView:: OnCommonFont 完成。该函数收集了用户选择的字体的信息，并利用这些信息创建新的字体。成员 m_ForeColor 用来保存所选字体的颜色，成员 m_Font 是一个 CFont 对象，用来保存用户选择的字体。在 CCommonView 的构造函数中，m_ForeColor 被初始化成黑色(0xFFFFFF)，m_Font 被初始化为系统字体。系统字体的获得是通过调用 CGdiObject::CreateStockObject(SYSTEM_FONT)实现的，该函数用于获得系统库存的绘图对象，包括字体、画笔、刷子、调色板等。

在 OnCommonFont 函数中，主要调用了下列函数：

调用 CFontDialog:: GetCurrentFont 以获得用户选择字体的信息，该函数的声明为 void GetCurrentFont(LPLOGFONT lplf);参数 lplf 是一个指向 LOGFONT 结构的指针，LOGFONT 结构用来存放与字体有关的信息。

调用 CFontDialog::GetColor 来获得所选字体的颜色(前景色)。

调用 CGdiObject:: DeleteObject()来删除存放在 CFont 对象 m_Font 中的老字体。

调用 CFont::CreateFontIndirect 以创建一种字体，该函数的声明是 BOOL CreateFontIndirect(const LOGFONT* lpLogFont);参数 lpLogFont 是一个指向 LOGFONT 结构的指针，函数根据该结构提供的信息来初始化 Cfont 对象。

调用 CWnd::Invalidate 和 CWnd::UpdateWindow 重绘视图。

在 CCommonView::OnDraw 函数中，利用选择的字体和颜色输出两行正文。当视图需要重绘时，OnDraw 就会被调用。在 OnDraw 函数中主要调用了下列函数：

在输出正文前，调用 CDC::SelectObject 指定输出正文的字体，输出完成后，调用 CDC::SelectObject 恢复被替换的字体。SelectObject 有五个版本，用于为绘图指定画笔、刷子、字体、位图等绘图对象。在用该函数指定绘图对象时，应该把被替换的对象保存起来，在绘图完成后，需要再次调用该函数恢复被替换的绘图对象。如果不进行恢复，则可能会使设备对象 CDC 中含有非法的句柄。指定字体的 SelectObject 函数的声明是 virtual CFont* SelectObject(CFont* pFont);参数 pFont 是指向 CFont 对象的指针。函数返回一个 CFont 对象的指针，指向被替代的字体。

调用 CDC::SetTextColor 来指定正文显示的前景色，该函数的声明为 virtual COLORREF SetTextColor(COLORREF crColor);参数 crColor 指定了 RGB 颜色。函数返回的是原来的正文颜色。

调用 CDC:: GetTextMetrics 函数获得与绘图字体有关的各种信息，该函数的声明为 BOOL GetTextMetrics(LPTEXTMETRIC lpMetrics) const; 参数 lpMetrics 是一个指向 TEXTMETRIC 结构的指针，该结构包含有字体的信息，其中 tmHeight 成员说明了字体的高度，tmExternalLeading 成员说明了行与行之间的空白应该是多少。把这两个值相加就得到了每行

正文的高度。

调用 CDC::TextOut 在指定位置输出正文。

在 CCommonDlgView::OnCommonPrint 和 CCommonDlgView::OnCommonPrintsetup()函数中，分别创建了一个 Print 对话框和 Print Setup 对话框。在创建 Print 对话框时，通过 CPrintDialog 对象的 m_pd 成员，对对话框进行了一些初始化，这包括对拷贝份数、打印范围等的设置。在两个对话框 DoModal 返回 IDOK 后，均调用 CCommonDlgView::DispPrintInfo 报告打印信息。DispPrintInfo 函数的代码较简单，这里就不作解释了。

小结

本课的要点为：

对话框的设计包括对话框模板的设计和对话框类的设计。对话框模板的设计是通过模板编辑器来完成的。对话框类的设计可借助 ClassWizard 来完成，这包括创建 CDialog 类的派生类，为对话框类增加与控件对应的成员变量，增加控件通知消息的处理函数等。

对话框的数据成员的初始化工作一般在其构造函数中完成，而对话框和控件的初始化是在 OnInitDialog 函数中完成的。

模态对话框拥有自己的消息循环，它垄断了用户的输入。模态对话框对象是以变量的形式构建的，CDialog::DoModal 用来启动一个模态对话框，在对话框关闭后该函数才返回。如果用户按下了 IDOK 按钮确认设置，那么 DoModal 返回 IDOK，若用户按下了 IDCANCEL 按钮取消设置，则 DoModal 返回 IDCANCEL。

非模态对话框与应用程序共用消息循环，它不垄断用户的输入。非模态对话框对象应该用 new 操作符在堆中创建，应该调用 CDialog::Create 而不是 CDialog::DoModal 来显示对话框，需要注意对话框的可见性问题。应该调用 CWnd::DestroyWindow 而不是 CDialog::EndDialog 来关闭非模态对话框，所以一般需要重新编写 OnOK 和 OnCancel 函数。非模态对话框对象应该是自动清除的，所以应该重写 PostNcDestroy 函数并在该函数中用 delete 删除对象本身。

除了主框架窗口类、视图类和非模态对话框类以外，MFC 的窗口类一般都是非自动清除的。不必调用 delete 来删除一个具有自动清除功能的窗口对象。

标签式对话框由多个页(子对话框)组成，可以容纳大量的控件。CPropertySheet 类代表对话框的框架，CPropertyPage 类代表莫一页。标签式对话框有一个特殊的 Apply 按钮，可以使用户在不退出对话框的情况下使设置生效。

Windows 支持五种公用对话框，包括文件选择、颜色选择、字体选择、打印和打印设置以及正文搜索和替换对话框。正文搜索和替换对话框与其它公用对话框不同，它是一个非模态对话框。

第六课　控件

在上一课中，同学们已经接触到了一些常用的控件。控件实际上是子窗口，在应用程序与用户进行交互的过程中，控件是主要角色。因此，有必要对控件进行详细的讨论。

Windows 提供了五花八门的标准控件，这些控件可粗分为两类。一类是在 Windows 3.x 就已支持的传统控件，一类是 Windows 95/NT 支持的新型 Win32 控件。Windows 提供控件的目的就是方便程序与用户的交互。应用程序应该根据自己的实际情况，选择合适的控件。

不管是什么类型的控件，一般都具有 WS_CHILD 和 WS_VISIBLE 窗口风格．WS_CHILD 指定窗口为子窗口，WS_VISIBLE 使窗口是可见的．另外，大部分控件还具有 WS_TABSTOP 风格，WS_TABSTOP 使控件具有 Tabstop 属性．

MFC 提供了大量的控件类，它们封装了控件的功能。通过这些控件类，程序可以方便地创建控件，对控件进行查询和控制。所有的控件类都是 CWnd 类的直接或间接派生类．

在学习这一讲之前，有几个问题需要先行说明：

在本节中，同学们会经常遇到控件类的 Create 成员函数，该函数负责创建控件．在上一章中同学们已经试验过，只要把控件放入对话框模板中，在调用 DoModal 或 Create 创建对话框时，框架会根据模板资源中的信息自动地创建控件。但有时需要用手工动态地创建控件，这通常需要按下面的步骤进行：

构建一个控件对象。

调用控件对象的成员函数 Create 来创建控件。

在 6.3 节中将对控件的创建进行详细讨论。

在上一章中，介绍了用 ClassWizard 为对话框类创建与传统控件对应的成员变量的方法．成员变量可以是数据变量或控件对象．需要指出的是，对于新的 Win32 控件，只能创建控件对象，不能创建数据变量．

在控件类的函数说明中，读者会经常看到 LPCTSTR 参数类型，LPCTSTR 是一个宏，相当于 const char far *，它用来说明指向常量字符串的指针．MFC 的字符串类 CString 定义了一个与 LPCTSTR 同名的操作符，该操作符可以把一个 CString 对象转换成一个常量字符串．因此，如果函数的参数是用 LPCTSTR 来说明的，则既可以向该参数传递一个指向常量字符串的指针，也可以传递一个 CString 对象．

这一讲将对一些常用的控件及其控件类进行较详细的讨论，讨论的侧重点包括控件的创建、控件类的成员函数以及控件的通知消息。具体讲，本章主要包括以下主要内容：

传统控件

新型 Win32 控件

控件的技术总结

在非对话框窗口中使用控件

如何设计新的控件

小结

6.1 传统控件

在上一课的表 5.1 已经列出了 Windows 的传统控件及其对应的控件类。在这些控件中，读者应该重点掌握命令按钮、选择框、单选按钮、编辑框、列表框和组合框。

6.1.1 传统控件的控件通知消息

控件通过向父窗口发送控件通知消息来表明发生了某种事件．例如，当用户在按钮上单击鼠标时，按钮控件会向父窗口发送 BN_CLICKED 消息．传统控件的通知消息实际上是通过 WM_COMMAND 消息发给父窗口的(滚动条除外)，在该消息的 wParam 中含有通知消息码(如 BN_CLICKED)和控件的 ID，在 lParam 中则包含了控件的句柄．

利用 ClassWizard 可以很容易地为控件通知消息加入消息映射和消息处理函数，这在上一章中已经演示过了．传统控件的消息映射宏是 ON_XXXX，其中 XXXX 表示通知消息码，如 BN_CLICKED．ON_XXXX 消息映射如下所示，该宏有两个参数，一个是控件的 ID，一个是消息处理函数名．

```
ON_XXXX(nID, memberFxn)
```

消息处理函数的声明应该有如下形式：

```
afx_msg void memberFxn( );
```

例如，某按钮的 BN_CLICKED 消息的消息映射及其处理函数的声明如下所示

```
ON_BN_CLICKED(IDC_ADD,OnAdd)
afx_msg void OnAdd( );
```

有时，为了处理方便，需要把多个 ID 连续的控件发出的相同消息映射到同一个处理函数上．这就要用到 ON_CONTROL_RANGE 宏．ON_CONTROL_RANGE 消息映射宏的第一个参数是控件消息码，第二和第三个参数分别指明了一组连续的控件 ID 中的头一个和最后一个 ID，最后一个参数是消息处理函数名。例如，要处理一组单选按钮发出的 BN_CLICKED 消息，相应的消息映射如下所示：

```
ON_CONTROL_RANGE(BN_CLICKED, IDC_FIRST, IDC_LAST, OnRadioClicked)
```

函数 OnRadioClicked 的声明如下，该函数比上面的 OnAdd 多了一个参数 nID 以说明发送通知消息的控件 ID．

```
afx_msg void OnRadioClicked(UINT nID);
```

ClassWizard 不支持 ON_CONTROL_RANGE 宏，所以需要手工建立消息映射和消息处理函数．

提示:事实上，在使用 ClassWizard 时只要运用一个小小的技巧，就可以把不同控件的通知消息映射到同一个处理函数上，也可以把一个控件的不同通知消息映射到同一个处理函数上．这个技巧就是在用 ClassWizard 创建消息处理函数时，指定相同的函数名即可．此方法的优点在于控件的 ID 不必是连续的，缺点是处理函数没有 nID 参数，因而不能确定是哪一个控件发送的消息．/

6.1.2 静态控件

静态控件包括静态正文(Static Text)和图片控件(Picture)。静态正文控件用来显示正文。图片控件可以显示位图、图标、方框和图元文件，在图片控件中显示图片的好处是不必操心图片的重绘问题。静态控件不能接收用户的输入。在上一章中，读者已经用过静态正文和组框控件。图片控件的例子可以在 AppWizard 创建的 IDD_ABOUTBOX 对话框模板中找到，在该模板中有一个图片控件用来显示图标。

静态控件的主要起说明和装饰作用。MFC 的 CStatic 类封装了静态控件。CStatic 类的成员函数 Create 负责创建静态控件，该函数的声明为

BOOL Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);

参数 lpszText 指定了控件显示的正文。dwStyle 指定了静态控件的风格，表 6.1 显示了静态控件的各种风格，dwStyle 可将这些风格组合起来。rect 是一个对 RECT 或 CRect 结构的引用，用来说明控件的位置和尺寸。pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE。

表 6.1 静态控件的风格

控件风格	含义
SS_BLACKFRAME	指定一个具有与窗口边界同色的框(缺省为黑色)。
SS_BLACKRECT	指定一个具有与窗口边界同色的实矩形(缺省为黑色)。
SS_CENTER	使显示的正文居中对齐，正文可以回绕。
SS_GRAYFRAME	指定一个具有与屏幕背景同色的边框。
SS_GRAYRECT	指定一个具有与屏幕背景同色的实矩形。
SS_ICON	使控件显示一个在资源中定义的图标，图标的名字有 Create 函数的 lpszText 参数指定。
SS_LEFT	左对齐正文，正文能回绕。
SS_LEFTNOWORDWRAP	左对齐正文，正文不能回绕。
SS_NOPREFIX	使静态正文串中的&不是一个热键提示符。
SS_NOTIFY	使控件能向父窗口发送鼠标事件消息。
SS_RIGHT	右对齐正文，可以回绕。
SS_SIMPLE	使静态正文在运行时不能被改变并使正文显示在单行中。
SS_USERITEM	指定一个用户定义项。
SS_WHITEFRAME	指定一个具有与窗口背景同色的框(缺省为白色)。
SS_WHITERECT	指定一个具有与窗口背景同色的实心矩形(缺省为白色)。

除了上表中的风格外，一般还要为控件指定 WS_CHILD 和 WS_VISIBLE 窗口风格。一个典型的静态正文控件的风格为 WS_CHILD|WS_VISIBLE|SS_LEFT。

对于用对话框模板编辑器创建的静态控件，可以在控件的属性对话框中指定表 6.1 中列出的控件风格。例如，可以在静态正文控件的属性对话框中选择 Simple，这相当于指定了 SS_SIMPLE 风格。

Cstatic 类主要的成员函数在表 6.2 中列出。可以利用 CWnd 类的成员函数 GetWindowText，SetWindowText 和 GetWindowTextLength 等函数来查询和设置静态控件中显示的正文．

表 6.2 CStatic 类的主要成员函数

函数声明	用途
HBITMAP SetBitmap(HBITMAP hBitmap);	指定要显示的位图。
HBITMAP GetBitmap( ) const;	获取由 SetBitmap 指定的位图。
HICON SetIcon(HICON hIcon);	指定要显示的图标。
HICON GetIcon( ) const;	获取由 SetIcon 指定的图标。
HCURSOR SetCursor(HCURSOR hCursor);	指定要显示的光标图片。
HCURSOR GetCursor( );	获取由 SetCursor 指定的光标。
HENHMETAFILE SetEnhMetaFile(HENHMETAFILE hMetaFile);	指定要显示的增强图元文件。
HENHMETAFILE GetEnhMetaFile( ) const;	获取由 SetEnhMetaFile 指定的图元文件。

静态控件较简单，故这里就不举例说明了。

6.1.3 按钮控件

按钮是指可以响应鼠标点击的小矩形子窗口。按钮控件包括命令按钮(Pushbutton)、检查框(Check Box)、单选按钮(Radio Button)、组框(Group Box)和自绘式按钮(Owner-draw Button)。命令按钮的作用是对用户的鼠标单击作出反应并触发相应的事件，在按钮中既可以显示正文，也可以显示位图。选择框控件可作为一种选择标记，可以有选中、不选中和不确定三种状态。单选按钮控件一般都是成组出现的，具有互斥的性质，即同组单选按钮中只能有一个是被选中的。组框用来将相关的一些控件聚成一组．自绘式按钮是指由程序而不是系统负责重绘的按钮。

按钮主要是指命令按钮、选择框和单选按钮。后二者实际上是一种特殊的按钮，它们有选择和未选择状态。当一个选择框处于选择状态时，在小方框内会出现一个“√”，当单选按钮处于选择状态时，会在圆圈中显示一个黑色实心圆。此外，检查框还有一种不确定状态，这时检查框呈灰色显示，不能接受用户的输入，以表明控件是无效的或无意义的。

按钮控件会向父窗口发出如表 6.3 所示的控件通知消息。

表 6.3 按钮控件的通知消息

消息	含义
BN_CLICKED	用户在按钮上单击了鼠标。
BN_DOUBLECLICKED	用户在按钮上双击了鼠标。

FC 的 CButton 类封装了按钮控件。CButton 类的成员函数 Create 负责创建按钮控件，该函数的声明为

BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 lpszCaption **指定了按钮显示的正文。**dwStyle **指定了按钮的风格，如表** 6.4 **所示，**dwStyle **可以是这些风格的组合。**rect **说明了按钮的位置和尺寸。**pParentWnd **指向父窗口，该参数不能为** NULL**。**nID **是按钮的** ID**。如果创建成功，该函数返回** TRUE**，否则返回** FALSE**。**

表 6.4 **按钮的风格**

控件风格	**含义**
BS_AUTOCHECKBOX	**同** BS_CHECKBOX**，不过单击鼠标时按钮会自动反转。**
BS_AUTORADIOBUTTON	**同** BS_RADIOBUTTON**，不过单击鼠标时按钮会自动反转。**
BS_AUTO3STATE	**同** BS_3STATE**，不过单击按钮时会改变状态。**
BS_CHECKBOX	**指定在矩形按钮右侧带有标题的选择框。**
BS_DEFPUSHBUTTON	**指定缺省的命令按钮，这种按钮的周围有一个黑框，用户可以按回车键来快速选择该按钮。**
BS_GROUPBOX	**指定一个组框。**
BS_LEFTTEXT	**使控件的标题显示在按钮的左边。**
BS_OWNERDRAW	**指定一个自绘式按钮。**
BS_PUSHBUTTON	**指定一个命令按钮。**
BS_RADIOBUTTON	**指定一个单选按钮，在圆按钮的右边显示正文。**
BS_3STATE	**同** BS_CHECKBOX**，不过控件有三种状态：选择、未选择和变灰。**

除了上表中的风格外，一般还要为控件指定 WS_CHILD**、**WS_VISIBLE **和** WS_TABSTOP **窗口风格，**WS_TABSTOP **使控件具有** Tabstop **属性。创建一个普通按钮应指定的风格为** WS_CHILD|WS_VISIBLE|WS_TABSTOP**。创建一个普通检查框应指定风格** WS_CHILD|WS_VISIBLE|WS_TABSTOP| BS_AUTOCHECKBOX**。创建组中第一个单选按钮应指定风格** WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_GROUP| BS_AUTORADIOBUTTON**，组中其它单选按钮应指定风格则不应该包括** WS_TABSTOP **和** WS_GROUP**。**

对于用对话框模板编辑器创建的按钮控件，可以在控件的属性对话框中指定表 6.4 **中列出的控件风格。例如，在命令按钮的属性对话框中选择** Default button**，相当于指定了** BS_DEFPUSHBUTTON**。**

CButton **类的主要的成员函数有：**

UINT GetState( ) const;**该函数返回按钮控件的各种状态。可以用下列屏蔽值与函数的返回值相与，以获得各种信息。**

0x0003**。用来获取检查框或单选按钮的状态。**0 **表示未选中，**1 **表示被选中，**2 **表示不确定状态(仅用于检查框)。**

0x0004**。用来判断按钮是否是高亮度显示的。非零值意味着按钮是高亮度显示的。当用户点击了按钮并按主鼠标左键时，按钮会呈高亮度显示。**

0x0008**。非零值表示按钮拥有输入焦点。**

void SetState(BOOL bHighlight);当参数 bHeightlight 值为 TRUE 时，该函数将按钮设置为高亮度状态，否则，去除按钮的高亮度状态。

int GetCheck( ) const;返回检查框或单选按钮的选择状态。返回值 0 表示按钮未被选择，1 表示按钮被选择，2 表示按钮处于不确定状态(仅用于检查框)。

void SetCheck(int nCheck);设置检查框或单选按钮的选择状态。参数 nCheck 值的含义与 GetCheck 返回值相同。

UINT GetButtonStyle( ) const;获得按钮控件的 BS_XXXX 风格。

void SetButtonStyle(UINT nStyle, BOOL bRedraw = TRUE);设置按钮的风格。参数 nStyle 指定了按钮的风格。bRedraw 为 TRUE 则重绘按钮，否则就不重绘。

HBITMAP SetBitmap(HBITMAP hBitmap);设置按钮显示的位图。参数 hBitmap 指定了位图的句柄。该函数还会返回按钮原来的位图。

HBITMAP GetBitmap( ) const;返回以前用 SetBitmap 设置的按钮位图。

HICON SetIcon(HICON hIcon);设置按钮显示的图标。参数 hIcon 指定了图标的句柄。该函数还会返回按钮原来的图标。

HICON GetIcon( ) const;返回以前用 SetIcon 设置的按钮图标。

HCURSOR SetCursor(HCURSOR hCursor);设置按钮显示的光标图。参数 hCursor 指定了光标的句柄。该函数还会返回按钮原来的光标。

HCURSOR GetCursor( );返回以前用 GetCursor 设置的光标。

另外，可以使用下列的一些与按钮控件有关的 CWnd 成员函数来设置或查询按钮的状态。用这些函数的好处在于不必构建按钮控件对象，只要知道按钮的 ID，就可以直接设置或查询按钮。

void CheckDlgButton(int nIDButton, UINT nCheck);用来设置按钮的选择状态。参数 nIDButton 指定了按钮的 ID。nCheck 的值 0 表示按钮未被选择，1 表示按钮被选择，2 表示按钮处于不确定状态。

void CheckRadioButton(int nIDFirstButton, int nIDLastButton, int nIDCheckButton);用来选择组中的一个单选按钮。参数 nIDFirstButton 指定了组中第一个按钮的 ID，nIDLastButton 指定了组中最后一个按钮的 ID，nIDCheckButton 指定了要选择的按钮的 ID。

int GetCheckedRadioButton(int nIDFirstButton, int nIDLastButton);该函数用来获得一组单选按钮中被选中按钮的 ID。参数 nIDFirstButton 说明了组中第一个按钮的 ID，nIDLastButton 说明了组中最后一个按钮的 ID。

UINT IsDlgButtonChecked(int nIDButton) const;返回检查框或单选按钮的选择状态。返回值 0 表示按钮未被选择，1 表示按钮被选择，2 表示按钮处于不确定状态(仅用于检查框)。

可以调用 CWnd 成员函数 GetWindowText，GetWindowTextLength 和 SetWindowText 来查询或设置按钮中显示的正文.

MFC 还提供了 CButton 的派生类 CBitmapButton。利用该类可以创建一个拥有四幅位图的命令按钮，按钮在不同状态时会显示不同的位图，

这样可以使界面显得生动活泼。如果读者对 CBitmapButton 感兴趣，可以参看 VC5.0 随盘提供的 MFC 例子 CTRLTEST。

在上一章的 Register 例子中已演示了各种按钮控件的使用，故这里就不再举例了。

6.1.4 编辑框控件

编辑框(Edit Box)控件实际上是一个简易的正文编辑器，用户可以在编辑框中输入并编辑正文。编辑框既可以是单行的，也可以是多行的，多行编辑框是从零开始编行号的．在一个多行编辑框中，除了最后一行外，每一行的结尾处都有一对回车换行符(用＂\r\n＂表示)．这对回车换行符是正文换行的标志，在屏幕上是不可见的．

编辑框控件会向父窗口发出如表 6.5 所示的控件通知消息。

表 6.5

消息	含义
EN_CHANGE	编辑框的内容被用户改变了。与 EN_UPDATE 不同，该消息是在编辑框显示的正文被刷新后才发出的。
EN_ERRSPACE	编辑框控件无法申请足够的动态内存来满足需要。
EN_HSCROLL	用户在水平滚动条上单击鼠标。
EN_KILLFOCUS	编辑框失去输入焦点。
EN_MAXTEXT	输入的字符超过了规定的最大字符数。在没有 ES_AUTOHSCROLL 或 ES_AUTOVSCROLL 的编辑框中，当正文超出了编辑框的边框时也会发出该消息。
EN_SETFOCUS	编辑框获得输入焦点。
EN_UPDATE	在编辑框准备显示改变了的正文时发送该消息。
EN_VSCROLL	用户在垂直滚动条上单击鼠标。

MFC 的 CEdit 类封装了编辑框控件。CEdit 类的成员函数 Create 负责创建按钮控件，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 指定了编辑框控件风格，如表 6.6 所示，dwStyle 可以是这些风格的组合。rect 指定了编辑框的位置和尺寸。pParentWnd 指定了父窗口，不能为 NULL。编辑框的 ID 由 nID 指定。如果创建成功，该函数返回 TRUE，否则返回 FALSE．

表 6.6 编辑框控件的风格

控件风格	含义
ES_AUTOHSCROLL	当用户在行尾键入一个字符时，正文将自动向右滚动 10 个字符，当用户按回车键时，正文总是滚向左边。
ES_AUTOVSC	当用户在最后一个可见行按回车键时，正文向上

ROLL	滚动一页。
ES_CENTER	在多行编辑框中使正文居中。
ES_LEFT	左对齐正文。
ES_LOWERCASE	把用户输入的字母统统转换成小写字母。
ES_MULTILINE	指定一个多行编辑器。若多行编辑器不指定 ES_AUTOHSCROLL 风格，则会自动换行，若不指定 ES_AUTOVSCROLL，则多行编辑器会在窗口中正文装满时发出警告声响。
ES_NOHIDESEL	缺省时，当编辑框失去输入焦点后会隐藏所选的正文，当获得输入焦点时又显示出来。设置该风格可禁止这种缺省行为。
ES_OEMCONVERT	使编辑框中的正文可以在 ANSI 字符集和 OEM 字符集之间相互转换。这在编辑框中包含文件名时是很有用的。
ES_PASSWORD	使所有键入的字符都用“*”来显示。
ES_RIGHT	右对齐正文。
ES_UPPERCASE	把用户输入的字母统统转换成大写字母。
ES_READONLY	将编辑框设置成只读的。
ES_WANTRETURN	使多行编辑器接收回车键输入并换行。如果不指定该风格，按回车键会选择缺省的命令按钮，这往往会导致对话框的关闭。

除了上表中的风格外，一般还要为控件指定 WS_CHILD、WS_VISIBLE、WS_TABSTOP 和 WS_BORDER 窗口风格，WS_BORDER 使控件带边框。创建一个普通的单行编辑框应指定风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP |WS_BORDER|ES_LEFT|ES_AUTOHSCROLL，这将创建一个带边框、左对齐正文、可水平滚动的单行编辑器。要创建一个普通多行编辑框，还要附加 ES_MULTILINE|ES_WANTRETURN|ES_AUTOVSCROLL |WS_HSCROLL| WS_VSCROLL 风格，这将创建一个可水平和垂直滚动的，带有水平和垂直滚动条的多行编辑器。

对于用对话框模板编辑器创建的编辑框控件，可以在控件的属性对话框中指定表 6.6 中列出的控件风格。例如，在属性对话框中选择 Multi-line 项，相当与指定了 ES_MULTILINE 风格。

编辑框支持剪贴板操作。CEdit 类提供了一些与剪贴板有关的成员函数，如表 6.7 所示。

表 6.7 与剪切板有关的 CEdit 成员函数

函数声明	用途
void Clear( )	清除编辑框中被选择的正文。
void Copy( )	把在编辑框中选择的正文拷贝到剪贴板中。
void Cut( )	清除编辑框中被选择的正文并把这些正文拷贝到剪贴板中。

void Paste( )	将剪贴板中的正文插入到编辑框的当前插入符处。
BOOL Undo( )	撤消上一次键入。对于单行编辑框，该函数总返回 TRUE，对于多行编辑框，返回 TRUE 表明操作成功，否则返回 FALSE。

可以用下列 CEdit 或 CWnd 类的成员函数来查询编辑框。在学习下面的函数时，读者会经常遇到术语字符索引．字符的字符索引是指从编辑框的开头字符开始的字符编号，它是从零开始编号的．也就是说，字符索引实际上是指当把整个编辑正文看作一个字符串数组时，该字符所在的数组元素的下标．

int GetWindowText(LPTSTR lpszStringBuf, int nMaxCount) const;void GetWindowText(CString& rString) const;这两个函数均是 CWnd 类的成员函数，可用来获得窗口的标题或控件中的正文。第一个版本的函数用 lpszStringBuf 参数指向的字符串数组作为拷贝正文的缓冲区，参数 nMaxCount 可以拷贝到缓冲区中的最大字符数，该函数返回以字节为单位的实际拷贝字符数(不包括结尾的空字节)。第二个版本的函数用一个 CString 对象作为缓冲区。

int GetWindowTextLength( ) const;CWnd 的成员函数，可用来获得窗口的标题或控件中的正文的长度。

DWORD GetSel( ) const;void GetSel(int& nStartChar, int& nEndChar) const;两个函数都是 CEdit 的成员函数，用来获得所选正文的位置。GetSel 的第一个版本返回一个 DWORD 值，其中低位字说明了被选择的正文开始处的字符索引，高位字说明了选择的正文结束处的后面一个字符的字符索引，如果没有正文被选择，那么返回的低位和高位字节都是当前插入符所在字符的字符索引。GetSel 的第二个版本的两个参数是两个引用，其含义与第一个版本函数返回值的低位和高位字相同。

int LineFromChar(int nIndex = －1) const;CEdit 的成员函数，仅用于多行编辑框，用来返回指定字符索引所在行的行索引(从零开始编号)。参数 nIndex 指定了一个字符索引，如果 nIndex 是-1，那么函数将返回选择正文的第一个字符所在行的行号，若没有正文被选择，则该函数会返回当前的插入符所在行的行号。

int LineIndex(int nLine = －1) const;CEdit 的成员函数，仅用于多行编辑框，用来获得指定行的开头字符的字符索引，如果指定行超过了编辑框中的最大行数，该函数将返回-1。参数 nLine 是指定了从零开始的行索引，如果它的值为-1，则函数返回当前的插入符所在行的字符索引。

int GetLineCount( ) const;CEdit 的成员函数，仅用于多行编辑框，用来获得正文的行数。如果编辑框是空的，那么该函数的返回值是 1。

int LineLength(int nLine = －1) const;CEdit 的成员函数，用于获取指定字符索引所在行的字节长度(行尾的回车和换行符不计算在内)。参数 nLine 说明了字符索引.如果 nLine 的值为-1,则函数返回当前行的长度(假如没有正文被选择)，或选择正文占据的行的字符总数减去选择正文的字符数(假如有正文被选择)。若用于单行编辑框，则函数返回整个正文的长度。

int GetLine(int nIndex, LPTSTR lpszBuffer) const;int GetLine(int nIndex, LPTSTR lpszBuffer, int nMaxLength) const;CEdit 的成员函数，仅用于多行编辑框，用来获得指定行的正文(不包括行尾的回车和换行符)。参数 nIndex 是行号，lpszBuffer 指向存放正文的缓冲区，nMaxLength 规定了拷贝的最大字节数，若。函数返回实际拷贝的字节数，若指定的行号大于编辑框的实际行数，则函数返回 0。需要注意的是，GetLine 函数不会在缓冲区中字符串的末尾加字符串结束符(NULL)．

下列 CWnd 或 CEdit 类的成员函数可用来修改编辑框控件。

void SetWindowText(LPCTSTR lpszString);CWnd 的成员函数，可用来设置窗口的标题或控件中的正文。参数 lpszString 可以是一个 CString 对象，或是一个指向字符串的指针。

void SetSel(DWORD dwSelection, BOOL bNoScroll = FALSE);void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);CEdit 的成员函数，用来选择编辑框中的正文。参数 dwSelection 的低位字说明了选择开始处的字符索引，高位字说明了选择结束处的字符索引。如果低位字为 0 且高位字节为-1，那么就选择所有的正文，如果低位字节为-1，则取消所有的选择．参数 bNoScroll 的值如果是 FALSE，则滚动插入符并使之可见，否则就不滚动．参数 nStartChar 和 nEndChar 的含义与参数 dwSelection 的低位字和高位字相同．

void ReplaceSel(LPCTSTR lpszNewText, BOOL bCanUndo = FALSE);CEdit 的成员函数，用来将所选正文替换成指定的正文．参数 lpszNewText 指向用来替换的字符串．参数 bCanUndo 的值为 TRUE 说明替换是否可以被撤消的．

在调用上述函数时，如果涉及的是一个多行编辑框，那么除了 LineLength 和 GetLine 函数外，都要把回车和换行符考虑在内．例如，假设在编辑框中有如下几行正文：

abcd
efg
ij

那么字母＂e＂的字符索引是 6 而不是 4，因为＂abcd＂后面还有一对回车换行符．调用 LineLength(7)会返回第二行的长度 3．调用 LineIndex(2)会得到 11．调用 LineFromChar(8)会返回 1．如果没有选择任何正文，并且插入符在字母＂e＂上，那么调用 GetSel 返回值的低位和高位字都是 6．

通过分析上述函数，我们可以总结出一些查询和设置编辑框的方法．

调用 CWnd 的成员函数 GetWindowText 和 SetWindowText 可以查询和设置编辑框的整个正文，在上一章的 Register 程序中，我们就使用过这两个函数．

如果想对多行编辑框逐行查询，那么应该先调用 GetLineCount 获得总行数，然后再调用 GetLine 来获取每一行的正文．下面一段代码演示了如何对多行编辑框进行逐行查询．

```
char buf[40];
```

```
int total=MyEdit.GetLineCount();
int i,length;
for(i=0;i<total;i++)
{
length=MyEdit.GetLine(i,buf,39);
buf[length]=0; //加字符串结束符
. . . . . .
}
```

可以利用 LineIndex 和 LineFromChar 来在字符索引和字符的行列坐标之间相互转换．下列代码演示了在已知字符索引的情况下，如何获得对应的行列坐标：int row,column;row=MyEdit.LineFromChar(charIndex);column=charIndex-MyEdit.LineIndex(row);下列代码演示了在已知字符的行列坐标的情况下，如何获得对应的字符索引：int charIndex;charIndex=MyEdit.LineIndex(row)+column;不难看出字符索引与对应的行列坐标的关系是：字符索引=LineIndex(行坐标)+列坐标．

对于选择正文的查询和设置，应该利用函数 GetSel、SetSel 和 ReplaceSel．

可以利用 GetSel 和 SetSel 来查询和设置插入符的位置．SetSel 可以使编辑框滚动到插入符的新位置．要获取插入符的行列坐标，可用下面的代码实现：MyEdit.SetSel(-1,0); // 取消正文的选择 int start,end,row,column;MyEdit.GetSel(start,end); //start 或 end 的值就是插入符的字符索引 row=MyEdit.LineFromChar(start); //获取插入符的行坐标 column=start-MyEdit.LineIndex(row); //获取插入符的列坐标下面的代码演示了如何把插入符移到指定的行和列：MyEdit.SetSel(-1,0); //取消正文的选择 int charIndex=MyEdit.LineIndex(row)+column;MyEdit.SetSel(charIndex,charIndex);

可以利用 ReplaceSel 函数在 插入符处插入正文，典型的代码如下所示：MyEdit.SetSel(-1,0); //取消正文的选择 MyEdit.ReplaceSel(“ ..... ”);

可以利用 ReplaceSel 清除编辑框中的正文，典型的代码如下所示：MyEdit.SetSel(0,-1); //选择全部正文 MyEdit.ReplaceSel(“ ”);

在后面的小节中，读者将会看到使用编辑框的例子．

6.1.5 滚动条控件

滚动条(Scroll Bar)主要用来从某一预定义值范围内快速有效地进行选择．滚动条分垂直滚动条和水平滚动条两种．在滚动条内有一个滚动框，用来表示当前的值．用鼠标单击滚动条，可以使滚动框移动一页或一行，也可以直接拖动滚动框．滚动条既可以作为一个独立控件存在，也可以作为窗口、列表框和组合框的一部分．Windows 95 的滚动条支持比例滚动框，即用滚动框的大小来反映页相对于整个范围的大小．Windows 3.x 使用单独的滚动条控件来调整调色板、键盘速度以及鼠标灵敏度，在 Windows 95 中，滚动条控件被轨道条取代(参见 6.2.3)不提

倡使用单独的滚动条控件．

需要指出的是，从性质上划分，滚动条可分为标准滚动条和滚动条控件两种．标准滚动条是由 WS_HSCROLL 或 WS_VSCROLL 风格指定的，它不是一个实际的窗口，而是窗口的一个组成部分(例如列表框中的滚动条)，只能位于窗口的右侧(垂直滚动条)或底端(水平滚动条)．标准滚动条是在窗口的非客户区中创建的．与之相反，滚动条控件并不是窗口的一个零件，而是一个实际的窗口，可以放置在窗口客户区的任意地方，它既可以独立存在，也可以与某一个窗口组合，行使滚动窗口的职能．由于滚动条控件是一个独立窗口，因此可以拥有输入焦点，可以响应光标控制键，如 PgUp、PgDown、Home 和 End．

MFC 的 CScrollBar 类封装了滚动条控件．CScrollBar 类的 Create 成员函数负责创建控件，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 指定了控件的风格．rect 说明了控件的位置和尺寸．pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE．

要创建一个普通的水平滚动条控件，应指定风格 WS_CHILD|WS_VISIBLE|BS_HORZ．要创建一个普通的垂直滚动条控件，应指定风格 WS_CHILD|WS_VISIBLE|BS_VERT．

主要的 CScrollBar 类成员函数如下所示：

int GetScrollPos( ) const;该函数返回滚动框的当前位置．若操作失败则返回 0．

int SetScrollPos(int nPos, BOOL bRedraw = TRUE);该函数将滚动框移动到指定位置．参数 nPos 指定了新的位置．参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之．函数返回滚动框原来的位置．若操作失败则返回 0．

void GetScrollRange(LPINT lpMinPos, LPINT lpMaxPos) const;该函数对滚动条的滚动范围进行查询．参数 lpMinPos 和 lpMaxPos 分别指向滚动范围的最小最大值．

void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE);该函数用于指定滚动条的滚动范围．参数 nMinPos 和 nMaxPos 分别指定了滚动范围的最小最大值．由这两者指定的滚动范围不得超过 32767．当两者都为 0 时，滚动条将被隐藏．参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之．

BOOL GetScrollInfo(LPSCROLLINFO lpScrollInfo, UINT nMask);该函数用来获取滚动条的各种状态，包括滚动范围、滚动框的位置和页尺寸．参数 lpScrollInfo 指向一个 SCROLLINFO 结构，该结构如下所示：
typedef struct tagSCROLLINFO {UINT cbSize; //结构的尺寸(字节为单位)UINT fMask; /*说明结构中的哪些参数是有效的，可以是屏蔽值的组合，如 SIF_POS|SIF_PAGE，若为 SIF_ALL 则整个结构都有效*/int nMin; //滚动范围最大值，当 fMask 中包含 SIF_RANGE 时有效 int nMax; //滚动

范围最小值，当 fMask 中包含 SIF_RANGE 时有效 UINT nPage; /*页尺寸，用来确定比例滚动框的大小，当 fMask 中包含 SIF_PAGE 时有效*/int nPos; //滚动框的位置，当 fMask 中包含 SIF_POS 有效 int nTrackPos; /*拖动时滚动框的位置，当 fMask 中包含 SIF_TRACKPOS 时有效，该参数只能查询，不能设 置，最好不要用该参数来查询拖动时滚动框的位置*/} SCROLLINFO;typedef SCROLLINFO FAR *LPSCROLLINFO;参数 nMask 的意义与 SCROLLINFO 结构中的 fMask 相同．函数在获得有效值后返回 TRUE，否则返回 FALSE．

BOOL SetScrollInfo(LPSCROLLINFO lpScrollInfo, BOOL bRedraw = TRUE);该函数用于设置滚动条的各种状态，一个重要用途是设定页尺寸从而实现比例滚动框．参数 lpScrollInfo 指向一个 SCROLLINFO 结构，参数 bRedraw 表示是否需要重绘滚动条，如果为 TRUE，则重绘之．若操作成功，该函数返回 TRUE，否则返回 FALSE．

CWnd 类也提供了一些函数来查询和设置所属的标准滚动条．这些函数与 CScrollBar 类的函数同名，且功能相同，但每个函数都多了一个参数，用来选择滚动条．例如，CWnd:: GetScrollPos 的声明为

int GetScrollPos(int nBar) const;参数 nBar 用来选择滚动条，可以为下列值：SB_HORZ //指定水平滚动条 SB_VERT //指定垂直滚动条

无论是标准滚动条，还是滚动条控件，滚动条的通知消息都是用 WM_HSCROLL 和 WM_VSCROLL 消息发送出去的．对这两个消息的确省处理函数是 CWnd::OnHScroll 和 CWnd::OnVScroll，它们几乎什么也不做．一般需要在派生类中对这两个函数从新设计，以实现滚动功能．这两个函数的声明为

afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);

afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);参数 nSBCode 是通知消息码，如表 6.8 所示．nPos 是滚动框的位置，只有在 nSBCode 为 SB_THUMBPOSITION 或 SB_THUMBTRACK 时，该参数才有意义．如果通知消息是滚动条控件发来的，那么 pScrollBar 是指向该控件的指针，如果是标准滚动条发来的，则 pScrollBar 为 NULL．

表 6.8　滚动条的通知消息码

消息	含义
SB_BOTTOM/SB_RIGHT(二者的消息码是一样的，因此可以混用，下同)	滚动到底端(右端)．
SB_TOP/SB_LEFT	滚动到顶端(左端)．
SB_LINEDOWN/SB_LINERIGHT	向下(向右)滚动一行(列)．
SB_LINEUP/SB_LINELEFT	向上(向左)滚动一行(列)．
SB_PAGEDOWN/SB_PAGERIGHT	向下(向右)滚动一页．
SB_PAGEUP/SB_PAGELEFT	向上(向左)滚动一页．
SB_THUMBPOSITION	滚动到指定位置．
SB_THUMBTRACK	滚动框被拖动．可利用该消息

	来跟踪对滚动框的拖动．
SB_ENDSCROLL	滚动结束．

6.1.8 小节的例子中，读者将学会如何使用滚动条以及如何编写自己的 OnHScroll 函数．

6.1.6 列表框控件

列表框主要用于输入，它允许用户从所列出的表项中进行单项或多项选择，被选择的项呈高亮度显示．列表框具有边框，并且一般带有一个垂直滚动条．列表框分单选列表框和多重选择列表框两种．单选列表框一次只能选择一个列表项，而多重选择列表框可以进行多重选择．对于列表项的选择，微软公司有如下建议：

单击鼠标选择一个列表项，单击一个按钮来处理选择的项．

双击鼠标选择一个列表项是处理选择项的快捷方法．

列表框会向父窗口发送如表 6.9 所示的通知消息．

表 6.9 列表框控件的通知消息

消息	含义
LBN_DBLCLK	用户用鼠标双击了一列表项．只有具有 LBS_NOTIFY 的列表框才能发送该消息．
LBN_ERRSPACE	列表框不能申请足够的动态内存来满足需要．
LBN_KILLFOCUS	列表框失去输入焦点．
LBN_SELCANCEL	当前的选择被取消．只有具有 LBS_NOTIFY 的列表框才能发送该消息．
LBN_SELCHANGE	单击鼠标选择了一列表项．只有具有 LBS_NOTIFY 的列表框才能发送该消息．
LBN_SETFOCUS	列表框获得输入焦点．
WM_CHARTOITEM	当列表框收到 WM_CHAR 消息后，向父窗口发送该消息．只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息．
WM_VKEYTOITEM	当列表框收到 WM_KEYDOWN 消息后，向父窗口发送该消息．只有具有 LBS_WANTKEYBOARDINPUT 风格的列表框才会发送该消息．

MFC 的 CListBox 类封装了列表框．CListBox 类的 Create 成员函数负责列表框的创建，该函数的声明是

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 指定了列表框控件的风格，如表 6.10 所示，dwStyle 可以是这些风格的组合．rect 说明了控件的位置和尺寸．pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE．

表 6.10 列表框控件的风格

控件风格	含义

LBS_EXTENDEDSEL	支持多重选择.在点击列表项时按住 Shift 键或 Ctrl 键即可选择多个 项.
LBS_HASSTRINGS	指定一个含有字符串的自绘式列表框.
LBS_MULTICOLUMN	指定一个水平滚动的多列列表框,通过调用 CListBox::SetColumnWidth 来设置每列的宽度.
LBS_MULTIPLESEL	支持多重选择.列表项的选择状态随着用户对该项单击或双击鼠标而翻转.
LBS_NOINTEGRALHEIGHT	列表框的尺寸由应用程序而不是 Windows 指定. 通常, Windows 指定尺寸会使列表项的某些部分隐藏起来.
LBS_NOREDRAW	当选择发生变化时防止列表框被更新,可发送 WM_SETREDRAW 来改变该风格.
LBS_NOTIFY	当用户单击或双击鼠标时通知父窗口.
LBS_OWNERDRAWFIXED	指定自绘式列表框,即由父窗口负责绘制列表框的内容, 并且列表项有相同的高度.
LBS_OWNERDRAWVARIABLE	指定自绘式列表框,并且列表项有不同的高度.
LBS_SORT	使插入列表框中的项按升序排列.
LBS_STANDARD	相当于指定了 WS_BORDER|WS_VSCROLL|LBS_SORT |LBS_NOTIFY.
LBS_USETABSTOPS	使列表框在显示列表项时识别并扩展制表符(‘ \t ’),缺省的制表宽度是 32 个对话框单位.
LBS_WANTKEYBOARDINPUT	允许列表框的父窗口接收 WM_VKEYTOITEM 和 WM_CHARTOITEM 消息,以响应键盘输入.
LBS_DISABLENOSCROLL	使列表框在不需要滚动时显示一个禁止的垂直滚动条.

除了上表中的风格外, 一般还要为列表框控件指定 WS_CHILD、WS_VISIBLE、WS_TABSTOP、WS_BORDER 和 WS_VSCROLL 风格.要创建一个普通的单选择列表框, 应指定的风格为 WS_CHILD|WS_VISIBLE|WS_TABSTOP|LBS_STANDARD.要创建一个多重选择列表框, 应该在单选择列表框风格的基础上再加上 LBS_MULTIPLESEL 或 LBS_ EXTENDEDSEL.如果不希望列表框排序,就不能使用 LBS_STANDARD 风格.

对于用对话框模板编辑器创建的列表框控件, 可以在控件的属性对话框中指定表 6.10 中列出的控件风格。例如, 在属性对话框中选择 Sort 项, 相当与指定了 LBS_SORT 风格。

CListBox 类的成员函数有数十个之多. 我们可以把一些常用的函数

分为三类，在下面列出．需要说明的是，可以用索引来指定列表项，索引是从零开始的．

首先，CListBox 成员函数提供了下列函数用于插入和删除列表项．

int AddString(LPCTSTR lpszItem);该函数用来往列表框中加入字符串，其中参数 lpszItem 指定了要添加的字符串．函数的返回值是加入的字符串在列表框中的位置，如果发生错误，会返回 LB_ERR 或 LB_ERRSPACE(内存不够)．如果列表框未设置 LBS_SORT 风格，那么字符串将被添加到列表的末尾，如果设置了 LBS_SORT 风格，字符串会按排序规律插入到列表中．

int InsertString(int nIndex, LPCTSTR lpszItem);该函数用来在列表框中的指定位置插入字符串．参数 nIndex 给出了插入位置(索引)，如果值为-1，则字符串将被添加到列表的末尾．参数 lpszItem 指定了要插入的字符串．函数返回实际的插入位置，若发生错误，会返回 LB_ERR 或 LB_ERRSPACE．与 AddString 函数不同，InsertString 函数不会导致 LBS_SORT 风格的列表框重新排序．不要在具有 LBS_SORT 风格的列表框中使用 InsertString 函数，以免破坏列表项的次序．

int DeleteString(UINT nIndex);该函数用于删除指定的列表项，其中参数 nIndex 指定了要删除项的索引．函数的返回值为剩下的表项数目，如果 nIndex 超过了实际的表项总数，则返回 LB_ERR．

void ResetContent( );该函数用于清除所有列表项．

int Dir(UINT attr, LPCTSTR lpszWildCard);该函数用来向列表项中加入所有与指定通配符相匹配的文件名或驱动器名．参数 attr 为文件类型的组合，如表 6.11 所示．参数 lpszWildCard 指定了通配符(如*.cpp，*.*等)．

表 6.11 Dir 函数 attr 参数的含义

值	含义
0x0000	普通文件(可读写的文件)．
0x0001	只读文件．
0x0002	隐藏文件．
0x0004	系统文件．
0x0010	目录．
0x0020	文件的归档位已被设置．
0x4000	包括了所有与通配符相匹配的驱动器．
0x8000	排除标志．若指定该标志，则只列出指定类型的文件名，否则，先要列出普通文件，然后再列出指定的文件．

下列的 CListBox 成员函数用于搜索、查询和设置列表框．

int GetCount( ) const;该函数返回列表项的总数，若出错则返回 LB_ERR．

int FindString(int nStartAfter, LPCTSTR lpszItem) const;该函数用于对列表项进行与大小写无关的搜索．参数 nStartAfter 指定了开始搜索的位置，合理指定 nStartAfter 可以加快搜索速度，若 nStartAfter 为-1，则从头开始搜索整个列表．参数 lpszItem 指定了要搜索的字符串．函数返回

与 lpszItem 指定的字符串相匹配的列表项的索引，若没有找到匹配项或发生了错误，函数会返回 LB_ERR．FindString 函数先从 nStartAfter 指定的位置开始搜索，若没有找到匹配项，则会从头开始搜索列表．只有找到匹配项，或对整个列表搜索完一遍后，搜索过程才会停止，所以不必担心会漏掉要搜索的列表项．

int GetText(int nIndex, LPTSTR lpszBuffer) const;void GetText(int nIndex, CString& rString) const;用于获取指定列表项的字符串．参数 nIndex 指定了列表项的索引.参数 lpszBuffer 指向一个接收字符串的缓冲区．引用参数 rString 则指定了接收字符串的 CString 对象．第一个版本的函数会返回获得的字符串的长度，若出错，则返回 LB_ERR．

int GetTextLen(int nIndex) const;该函数返回指定列表项的字符串的字节长度．参数 nIndex 指定了列表项的索引．若出错则返回 LB_ERR．

DWORD GetItemData(int nIndex) const;每个列表项都有一个 32 位的附加数据．该函数返回指定列表项的附加数据，参数 nIndex 指定了列表项的索引．若出错则函数返回 LB_ERR．

int SetItemData(int nIndex, DWORD dwItemData);该函数用来指定某一列表项的 32 位附加数据．参数 nIndex 指定了列表项的索引．dwItemData 是要设置的附加数据值．

提示:列表项的 32 位附加数据可用来存储与列表项相关的数据，也可以放置指向相关数据的指针．这样，当用户选择了一个列表项时，程序可以从附加数据中快速方便地获得与列表项相关的数据．/

int GetTopIndex( ) const;该函数返回列表框中第一个可见项的索引，若出错则返回 LB_ERR．

int SetTopIndex(int nIndex);用来将指定的列表项设置为列表框的第一个可见项，该函数会将列表框滚动到合适的位置．参数 nIndex 指定了列表项的索引．若操作成功，函数返回 0 值，否则返回 LB_ERR．

提示：由于列表项的内容一般是不变的，故 CListBox 未提供更新列表项字符串的函数.如果要改变某列表项的内容,可以先调用 DeleteString 删除该项，然后再用 InsertString 或 AddString 将更新后的内容插入到原来的位置．/

下列 CListBox 的成员函数与列表项的选择有关．

int GetSel(int nIndex) const;该函数返回指定列表项的状态．参数 nIndex 指定了列表项的索引．如果查询的列表项被选择了，函数返回一个正值，否则返回 0，若出错则返回 LB_ERR．

int GetCurSel( ) const;该函数仅适用于单选择列表框，用来返回当前被选择项的索引，如果没有列表项被选择或有错误发生，则函数返回 LB_ERR．

int SetCurSel(int nSelect);该函数仅适用于单选择列表框，用来选择指定的列表项．该函数会滚动列表框以使选择项可见．参数 nIndex 指定了列表项的索引，若为-1，那么将清除列表框中的选择．若出错函数返回 LB_ERR．

int SelectString(int nStartAfter, LPCTSTR lpszItem);该函数仅适用于

单选择列表框，用来选择与指定字符串相匹配的列表项．该函数会滚动列表框以使选择项可见．参数的意义及搜索的方法与函数 FindString 类似．如果找到了匹配的项，函数返回该项的索引，如果没有匹配的项，函数返回 LB_ERR 并且当前的选择不被改变．

int GetSelCount( ) const;该函数仅用于多重选择列表框，它返回选择项的数目，若出错函数返回 LB_ERR．

int SetSel(int nIndex, BOOL bSelect = TRUE);该函数仅适用于多重选择列表框，它使指定的列表项选中或落选．参数 nIndex 指定了列表项的索引，若为-1，则相当于指定了所有的项．参数 bSelect 为 TRUE 时选中列表项，否则使之落选．若出错则返回 LB_ERR．

int GetSelItems(int nMaxItems, LPINT rgIndex) const;该函数仅用于多重选择列表框，用来获得选中的项的数目及位置．参数 nMaxItems 说明了参数 rgIndex 指向的数组的大小．参数 rgIndex 指向一个缓冲区，该数组是一个整型数组，用来存放选中的列表项的索引．函数返回放在缓冲区中的选择项的实际数目，若出错函数返回 LB_ERR．

int SelItemRange(BOOL bSelect, int nFirstItem, int nLastItem);该函数仅用于多重选择列表框，用来使指定范围内的列表项选中或落选．参数 nFirstItem 和 nLastItem 指定了列表项索引的范围．如果参数 bSelect 为 TRUE，那么就选择这些列表项，否则就使它们落选．若出错函数返回 LB_ERR．

在 6.1.8 小节的例子中，读者将会看到对列表框的测试．

6.1.7 组合框控件

组合框把一个编辑框和一个单选择列表框结合在了一起．用户既可以在编辑框中输入，也可以从列表框中选择一个列表项来完成输入．如上一章所提到的，组合框分为简易式(Simple)、下拉式(Dropdown)和下拉列表式(Drop List)三种．简易式组合框包含一个编辑框和一个总是显示的列表框。下拉式组合框同简易式组合框类似，二者的区别在于仅当单击下滚箭头后列表框才会弹出。下拉列表式组合框也有一个下拉的列表框，但它的编辑框是只读的，不能输入字符。

Windows 中比较常用的是下拉式和下拉列表式组合框，在 Developer Studio 中就大量使用了这两种组合框．二者都具有占地小的特点，这在界面日益复杂的今天是十分重要的．下拉列表式组合框的功能与列表框类似．下拉式组合框的典型应用是作为记事列表框使用，既把用户在编辑框中敲入的东西存储到列表框组件中，这样当用户要重复同样的输入时，可以从列表框组件中选取而不必在编辑框组件中从新输入．在 Developer Studio 中的 Find 对话框中就可以找到一个典型的下拉式组合框．

要设计一个记事列表框，应采取下列原则：

在创建组合框时指定 CBS_DROPDOWNLIST 风格．

要限制列表项的数目，以防止内存不够．

如果在编辑框中输入的字符串不能与列表框组件中的列表项匹配，那么应该把该字符串插入到列表框中的 0 位置处．最老的项处于列表的

末尾．如果列表项的数目超出了限制，则应把最老的项删除．

如果在编辑框中输入的字符串可以与列表框组件中的某一项完全匹配，则应该先把该项从列表的当前位置删除，然后在将其插入道列表的 0 位置处．

组合框控件会向父窗口发送表 6.12 所示的通知消息．

表 6.12 组合框控件的通知消息

消息	含义
CBN_CLOSEUP	组合框的列表框组件被关闭．简易式组合框不会发出该消息．
CBN_DBLCLK	用户在某列表项上双击鼠标．只有简易式组合框才会发出该消息．
CBN_DROPDOWN	组合框的列表框组件下拉．简易式组合框不会发出该消息．
CBN_EDITCHANGE	编辑框的内容被用户改变了。与 CBN_EDITUPDATE 不同，该消息是在编辑框显示的正文被刷新后才发出的。下拉列表式组合框不会发出该消息．
CBN_EDITUPDATE	在编辑框准备显示改变了的正文时发送该消息。下拉列表式组合框不会发出该消息．
CBN_ERRSPACE	组合框无法申请足够的内存来容纳列表项．
CBN_SELENDCANCEL	表明用户的选择应该取消．当用户在列表框中选择了一项，然后又在组合框控件外单击鼠标时就会导致该消息的发送．
CBN_SELENDOK	用户选择了一项，然后按了回车键或单击了下滚箭头．该消息表明用户确认了自己所作的选择．
CBN_KILLFOCUS	组合框失去了输入焦点．
CBN_SELCHANGE	用户通过点击或移动箭头键改变了列表的选择．
CBN_SETFOCUS	组合框获得了输入焦点．

MFC 的 CComboBox 类封装了组合框．需要指出的是，虽然组合框是编辑框和列表框的选择，但是 CComboBox 类并不是 CEdit 类和 CListBox 类的派生类，而是 CWnd 类的派生类．

CComboBox 的成员函数 Create 负责创建组合框，该函数的说明如下：

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 指定了组合框控件的风格，如表 6.10 所示，dwStyle 可以是这些风格的组合．rect 说明的是列表框组件下拉后组合框的位置和尺寸．pParentWnd 指向父窗口，该参数不能为 NULL。nID 则说明了控件的 ID。如果创建成功，该函数返回 TRUE，否则返回 FALSE．

提示：在用 Create 函数创建组合框时，参数 rect 说明的是包括列表框组件在内的组合框的位置和尺寸，而不是列表框组件隐藏时的编辑框

组件尺寸．要设置编辑框组件的高度，可以调用成员函数 SetItemHeight(-1,cyItemHeight)，其中参数 cyItemHeight 指定了编辑框的高度(以像素为单位)．/

表 6.13 组合框的风格

控件风格	含义
CBS_AUTOHSCROLL	使编辑框组件具有水平滚动的风格．
CBS_DROPDOWN	指定一个下拉式组合框．
CBS_DROPDOWNLIST	指定一个下拉列表式组合框．
CBS_HASSTRINGS	指定一个含有字符串的自绘式组合框．
CBS_OEMCONVERT	使编辑框组件中的正文可以在 ANSI 字符集和 OEM 字符集之间相互转换。这在编辑框中包含文件名时是很有用的。
CBS_OWNERDRAWFIXED	指定自绘式组合框，即由父窗口负责绘制列表框的内容，并且列表项有相同的高度．
CBS_OWNERDRAWVARIABLE	指定自绘式组合框，并且列表项有不同的高度．
CBS_SIIMPLE	指定一个简易式组合框．
CBS_SORT	自动对列表框组件中的项进行排序．
CBS_DISABLENOSCROLL	使列表框在不需要滚动时显示一个禁止的垂直滚动条．
CBS_NOINTEGRALHEIGHT	组合框的尺寸由应用程序而不是 Windows 指定．通常，由 Windows 指定尺寸会使列表项的某些部分隐藏起来．

CBS_SIMPLE、CBS_DROPDOWN 和 CBS_DROPDOWNLIST 分别用来将组合框指定为简易式、下拉式和下拉列表式．一般还要为组合框指定 WS_CHILD、WS_VISIBLE、WS_TABSTOP、WS_VSCROLL 和 CBS_AUTOHSCROLL 风格．如果要求自动排序，还应指定 CBS_SORT 风格．

对于用对话框模板编辑器创建的组合框控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Dropdown，相当于指定了 CBS_DROPDOWN．

CComboBox 类的成员函数较多．其中常用的函数可粗分为两类，分别针对编辑框组件和列表框组件．可以想象，这些函数与 CEdit 类和 CListBox 类的成员函数肯定有很多类似之处，但它们也会有一些不同的特点．如果读者能从＂组合框是由编辑框和列表框组成＂这一概念出发，就能够很快的掌握 CComboBox 的主要成员函数．

事实上，绝大部分 CComboBox 的成员函数都可以看成是 CEdit 或 CListBox 成员函数的翻版．函数的功能，函数名，甚至函数的参数都是类似的．为了方便学习，在下面列出 CComboBox 类的成员函数时，采用了与对应的 CEdit 或 CListBox 成员函数相比较的做法．在成员函数的列表中，分别列出了成员函数名，对应的 CEdit 或 CListBox 成员函数，以及二者之间的不同之处．不同之处是指函数的功能、参数以及返回值

有什么差别.

针对编辑框组件的主要成员函数如表 6.14 所示．该表的前三个函数实际上是 CWnd 类的成员函数，可用来查询和设置编辑框组件．

表 6.14 针对编辑框组件的 CComboBox 成员函数

成员函数名	对应的 CEdit 成员函数	与 CEdit 成员函数的不同之处
CWnd::GetWindowText	CWnd::GetWindowText	无．
CWnd::SetWindowText	CWnd::SetWindowText	无．
CWnd::GetWindowTextLength	CWnd::GetWindowTextLength	
GetEditSel	GetSel 的第一个版本	仅函数名不同．
SetEditSel	SetSel 的第二个版本	函数名不同，且无 bNoScroll 参数．
Clear	Clear	无．
Copy	Copy	无．
Cut	Cut	无．
Paste	Paste	无．

与 CListBox 的成员函数类似，针对列表框组件的 CComboBox 成员函数也可以分为三类.表 6.15 列出了用于插入和删除列表项的成员函数，表 6.16 列出了用于搜索、查询和设置列表框的成员函数，与列表项的选择有关的成员函数在表 6.17 中列出．需要指出的是，如果这些函数出错，则反回 CB_ERR，而不是 LB_ERR．另外，排序的组合框具有的是 CBS_SORT 风格，而不是 LBS_SORT．

6.15 用于插入和删除列表项的 CComboBox 成员函数

成员函数名	对应的 CListBox 成员函数	与 CListBox 成员函数的不同之处
AddString	AddString	无．
InsertString	InsertString	无．
DeleteString	DeleteString	无．
ResetContent	ResetContent	无．
Dir	Dir	无．

6.16 用于搜索、查询和设置列表框的 CComboBox 成员函数

成员函数名	对应的 CListBox 成员函数	与 CListBox 成员函数的不同之处
GetCoun	GetCoun	无．

t	t	
FindString	FindString	无.
GetLBText	GetText	仅函数名不同.
GetLBTextLen	GetTextLen	仅函数名不同.
GetItemData	GetItemData	无.
SetItemData	SetItemData	无.
GetTopIndex	GetTopIndex	无.
SetTopIndex	SetTopIndex	无.

表 6.17 与列表项的选择有关的 CComboBox 成员函数

成员函数名	对应的 CListBox 成员函数	与 CListBox 成员函数的不同之处
GetCurSel	GetCurSel	无.
SetCurSel	SetCurSel	新选的列表项的内容会被拷贝到编辑框组件中.
SelectString	SelectString	新选的列表项的内容会被拷贝到编辑框组件中.

另外，CComboBox 的 ShowDropDown 成员函数专门负责显示或隐藏列表框组件，该函数的声明为

```
void ShowDropDown( BOOL bShowIt = TRUE );
```

如果参数 bShowIt 的值为 TRUE，那么将显示列表框组件，否则，就隐藏之．该函数对简易式组合框没有作用．

6.1.8 测试传统控件的一个例子

现在让我们编写一个程序来测试一下上面介绍的一些传统控件．该程序名为 CtrlTest，其界面如图 6.1 所示．前面介绍的程序都是基于框架窗口的，而 CtrlTest 程序是一个基于对话框的应用程序，即以对话框作为程序的主窗口．该程序主要对组合框、列表框、多行编辑框和滚动条控件进行了测试，其中：

Input 组合框是一个记事列表框．在编辑框组件中输入字符串，或从列表框组件中选择以前输入过的字符串，然后按 Add 按钮，该字符串就会被加入到 List 列表框中．

List 列表框是一个多重选择列表框．该列表框具有 LBS_EXTENDEDSEL 风格，用户可以单击鼠标进行单项选择，也可以按住 Shift 或 Ctrl 键后单击鼠标来进行多重选择．用户可以按 Delete 键删除

列表框中选择的项．

History of SELCHANGE 多行编辑框．该编辑框用于跟踪 Input 组合框的列表框组件发出的 CBN_SELCHANGE 通知消息，编辑框对该消息的响应是显示 XXXX selected，以表明用户新选择了一个列表项．读者通过该编辑框可以了解组合框是在什么情况下发送 CBN_SELCHANGE 通知消息的．按 Clear 按钮将清除编辑框．

水平滚动条控件的滚动范围是 0 — 50．在滚动条的左边有一个静态正文控件用来动态反映当前滚动框的位置．

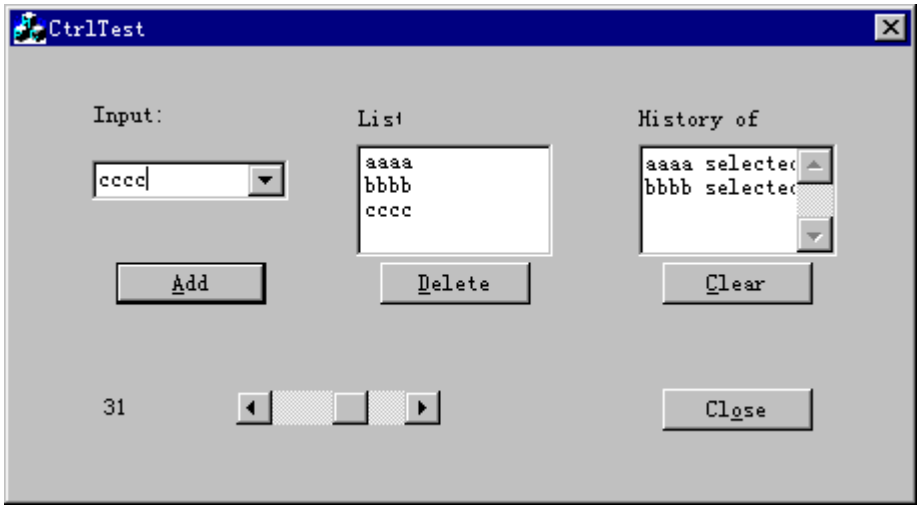


图 6.1 CtrlTest 程序

首先，让我们用 AppWizard 建立一个基于对话框的 MFC 应用程序．这一过程很简单，先将新建的工程命名为 CtrlTest，然后在 MFC AppWizard 对话框的第一步中选择 Dialog based 就行了．

AppWizard 会自动建立一个用于应用程序主窗口的对话框模板 IDD_CTRLTEST_DIALOG 及其对应的对话框类 CCtrlTestDlg．对该对话框的使用与普通对话框并没有什么不同，只不过在程序启动后对话框会自动显示出来，而当用户关闭对话框后，应用程序也就终止了．如果读者观察 CCtrlTestApp:: InitInstance 函数就会发现，该函数调用 DoModal 来显示一个 CCtrlTestDlg 对话框，并使 m_pMainWnd 指针指向 CCtrlTestDlg 对象，从而使该对话框成为程序的主窗口．

接下来，需要设计 IDD_CTRLTEST_DIALOG 对话框模板．请读者将该模板上除 OK 按钮以外的控件都删除掉，将 OK 按钮的标题改为 Cl&ose，并去掉该按钮的 Default button(缺省按钮)属性．当用户在对话框内按回车键时，会激活缺省按钮，一般应该把用来确认用户输入操作的按钮设计成缺省按钮．在本例中，显然应该把 Add 按钮设计成缺省按钮，而不是 Close 按钮．这样，用户在 Input 组合框中输入字符串后，按回车键就可以将该串加入到 List 列表框中．

请读者根据图 6.1 和表 6.18，向 IDD_CTRLTEST_DIALOG 对话框模板中加入测试用的控件．

表 6.18

控件类型	ID	标　　题	其它属性

		(Caption)	
静态正文	**缺省**	Input:	**缺省．**
组合框	ID_COMBOBOX		**去掉** Sort **属性．**
命令按钮	IDC_ADD	&Add	**选择** Default button **属性．**
静态正文	**缺省**	List:	**缺省．**
列表框	IDC_LISTBOX		**在** Selection **栏中选择** Extended，**并去掉** Sort **属性．**
命令按钮	IDC_DELETE	&Delete	**缺省．**
静态正文	**缺省**	History of SELCHANGE	**缺省．**
编辑框	IDC_MULTIEDIT		**选择** Multi-line，Vertical scroll，AutoVScroll **和** Want return **属性．**
命令按钮	IDC_CLEAR	&Clear	**缺省．**
静态正文	IDC_INDICATOR	**缺省**	**缺省．**
滚动条	IDC_SCROLLBAR		**缺省．**

接着，利用 ClassWizard **为** CCtrlTestDlg **类加入成员变量，如表** 6.19 **所示，这些成员变量都是控件对象．**

表 6.19 CCtrlTestDlg **类的成员变量**

控件 ID	**变量类型**	**变量名**
IDC_COMBOBOX	CComboBox	m_ComboBox
IDC_LISTBOX	CListBox	m_ListBox
IDC_MULTIEDIT	CEdit	m_MultiEdit
IDC_INDICATOR	CStatic	m_Indicator
IDC_SCROLLBAR	CScrollBar	m_ScrollBar

接下来，用 ClassWizard **为** CCtrlTestDlg **类加入控件通知消息处理函数，如表** 6.20 **所示．**

表 6.20 CCtrlTestDlg **的控件通知消息处理函数**

Object IDS	Messages	Member functions
IDC_ADD	BN_CLICKED	OnAdd(**缺省名**)
IDC_DELETE	BN_CLICKED	OnDelete(**缺省名**)
IDC_CLEAR	BN_CLICKED	OnClear(**缺省名**)
IDC_COMBOBOX	CBN_SELCHANGE	OnSelchangeCombobox(**缺省名**)
CCtrlTestDlg	WM_HSCROLL	OnHScroll(**缺省名**)

最后，请读者按清单 6.1 **修改源代码，限于篇幅，这里仅列出需要手工修改的那一部分．**

清单 6.1 CCtrlTestDlg **类的部分源代码**

```
// CtrlTestDlg.cpp : implementation file
//
```

```
#define MAX_HISTORY 5
. . . . . .
BOOL CCtrlTestDlg::OnInitDialog()
{
. . . . . .
// TODO: Add extra initialization here
m_ScrollBar.SetScrollRange(0,50);
m_Indicator.SetWindowText("0");
m_ComboBox.SetFocus(); //使组合框获得输入焦点
returnFALSE; //  返回 FALSE 以表明为某一控件设置了输入焦点
}
void CCtrlTestDlg::OnAdd()
{
// TODO: Add your control notification handler code here
int i;
CString str;
m_ComboBox.GetWindowText(str);
m_ListBox.AddString(str);
i=m_ComboBox.FindString(-1,str);
if(i>=0)
{
m_ComboBox.DeleteString(i);
m_ComboBox.InsertString(0,str); //将匹配项移到 0 位置
}
else
{
m_ComboBox.InsertString(0,str);
if(m_ComboBox.GetCount()>MAX_HISTORY)
m_ComboBox.DeleteString(m_ComboBox.GetCount()-1);  //删除旧的
项
}
}
void CCtrlTestDlg::OnClear()
{
// TODO: Add your control notification handler code here
m_MultiEdit.SetSel(0,-1);
m_MultiEdit.ReplaceSel("");
}
void CCtrlTestDlg::OnDelete()
{
// TODO: Add your control notification handler code here
int i,count;
```

```
int *pBuffer;
count=m_ListBox.GetSelCount();
if(count<=0)return;
pBuffer=new int[count];
m_ListBox.GetSelItems(count,pBuffer);
for(i=count-1;i>=0;i--) //倒序删除选择项
m_ListBox.DeleteString(pBuffer[i]);
delete pBuffer;
}
void CCtrlTestDlg::OnHScroll(UINT nSBCode, UINT nPos,
CScrollBar* pScrollBar)
{
// TODO: Add your message handler code here and/or call default
int nScrollMin,nScrollMax,nScrollPos;
int nPageSize;
CString str;
if(&m_ScrollBar!=pScrollBar)return;
nScrollPos=m_ScrollBar.GetScrollPos();
m_ScrollBar.GetScrollRange(&nScrollMin,&nScrollMax);
nPageSize=(nScrollMax-nScrollMin)/10; //指定页长
switch(nSBCode)
{
case SB_LEFT:
nScrollPos=nScrollMin;
break;
case SB_RIGHT:
nScrollPos=nScrollMax;
break;
case SB_LINELEFT:
nScrollPos-=1;
break;
case SB_LINERIGHT:
nScrollPos+=1;
break;
case SB_PAGELEFT:
nScrollPos-=nPageSize;
break;
case SB_PAGERIGHT:
nScrollPos+=nPageSize;
break;
case SB_THUMBPOSITION:
nScrollPos=nPos; //由参数 nPos 获取滚动框的位置
```

```
break;
case SB_THUMBTRACK:
nScrollPos=nPos; //由参数 nPos 获取滚动框的位置
break;
default:;
}
if(nScrollPos<nScrollMin)nScrollPos=nScrollMin;
if(nScrollPos>nScrollMax)nScrollPos=nScrollMax;
if(nScrollPos!=m_ScrollBar.GetScrollPos())
m_ScrollBar.SetScrollPos(nScrollPos); //设置滚动框的新位置
str.Format("%d",nScrollPos);
m_Indicator.SetWindowText(str); //更新静态正文
CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
void CCtrlTestDlg::OnSelchangeCombobox()
{
// TODO: Add your control notification handler code here
int length=m_MultiEdit.GetWindowTextLength();
CString str;
m_MultiEdit.SetSel(-1,0);
m_MultiEdit.SetSel(length,length); //移动插入符到编辑正文的末尾
m_ComboBox.GetLBText(m_ComboBox.GetCurSel(),str);
str+=" selected\r\n";
m_MultiEdit.ReplaceSel(str);
}
```

在 OnInitDialog 成员函数中，对一些控件进行了初始化，包括设置滚动条的范围，将静态正文的显示置为 "0"，以及使组合框获得输入焦点．注意，缺省时，OnInitDialog 返回 TRUE，而新版的函数返回了 FALSE．如果 OnInitDialog 返回 TRUE，那么 Windows 将使 tab 顺序最靠前的可输入控件获得输入焦点，如果返回 FALSE，则表明在 OnInitDialog 函数中人为地使某个控件获得输入焦点，函数返回后系统就不会再设置输入焦点了．有时，只要合理的安排了控件的 tab 顺序，就不必在 OnInitDialog 中人为设置输入焦点．

当用户点击 Add 按钮或按回车键后，成员函数 OnAdd 被调用．该函数将组合框的编辑框中的字符串加入到 List 列表框的末尾，并将该字符串存入到记事列表框中．这时函数会判断，如果在记事列表中没有匹配的项，则把字符串插入 0 位置，并在必要时删除最老的列表项，在本例中，记事列表框最多可以容纳 5 项；如果在记事列表中有匹配的项，那么就把该项移到 0 位置．

当用户点击 Delete 按钮时，成员函数 OnDelete 被调用，该函数根据 CComboBox::GetSelCount 获得选择项的数目，并根据这个数目动态创建一个整型数组以存放选择项的索引．然后，调用 CComboBox::GetSelItems

来获取选择项的索引．最后，把这些选择项删除．注意，这里是倒序删除的，如果按顺序删除，则会使选择项的索引产生错位．

成员函数 OnSelchangeCombobox 是 Input 组合框的 CBN_SELCHANGE 消息的处理函数．该函数先把多行编辑框的插入符移到编辑正文的末尾，然后从插入符处加入一行形如"XXXX selected"的字符串，以表明用户从记事列表框中新选择了哪个列表项．值得一提的是，上一章的 Register 程序是用 SetWindowText 来在编辑正文中插入新的正文的，此方法有一个缺点，就是不能把插入符滚动到新修改过的地方．在本例中，插入编辑正文的方法是先调用 CEdit::SetSel 移动插入符到指定位置(必要时要滚动以使该位置可见)，然后再调用 CEdit::ReplaceSel 插入新的正文，这样做的好处是编辑框总是滚动到新修改过的地方，从而使得新修改过的地方总是可见的．

对滚动条控件的测试是在 OnHScroll 成员函数中完成的．该函数是对话框也即父窗口对水平滚动条控件产生的 WM_HSCROLL 消息的处理函数．该函数负责移动滚动框并及时更新静态正文的显示以反映滚动框的当前位置．在函数的开头，首先判断是不是 m_ScrollBar 滚动条发来的消息，这是因为可能会有几个滚动条控件．在该函数中有一个大的 switch 分枝语句，用来获取滚动框的新位置．需要指出的是，对于 SB_THUMBPOSITION 和 SB_THUMBTRACK 这两种情况，应该从 OnHScroll 函数的 nPos 参数中获取滚动框的新位置．对于 SB_THUMBTRACK，不要企图用 CScrollBar::GetScrollPos 来获取滚动框的新位置，因为该函数不能正确返回拖动时的滚动框位置．另一个要注意的问题是 Windows 本身不会自动地使滚动框移动到新位置上，所以需要在 OnHScroll 中调用 CScrollBar::SetScrollPos 来移动滚动框．

6.2 新的 Win32 控件

从 Windows 95 和 Windows NT 3.51 版开始，Windows 提供了一些先进的 Win32 控件．这些新控件弥补了传统控件的某些不足之处，并使 Windows 的界面丰富多彩且更加友好．MFC 的新控件类封装了这些控件，新控件及其对应的控件类如表 6.21 所示．

表 6.21 新的 Win32 控件及其控件类

控件名	功能	对应的控件类
动画(Animate)	可播放 avi 文件．	CAnimateCtrl
热键(Hot Key)	使用户能选择热键组合．	CHotKeyCtrl
列表视图(List View)	能够以列表、小图标、大图标或报告格式显示数据．	CListCtrl
进度条(Progress Bar)	用于指示进度．	CProgressCtrl
滑尺(Slider)	也叫轨道条(Trackbar)，用户可以移动滑尺来在某一范围中进行选择．	CSliderCtrl
旋转按钮(Spin Button)	有时被称为上下控件．有一对箭头按钮，用来调节某一值的大小．	CSpinButtonCtrl
标签(Tab)	用来作为标签使用．	CTabCtrl
树形视图(Tree View)	以树状结构显示数据．	CTreeCtrl

本节将主要介绍列表视图、树形视图、进度条、旋转按钮和滑尺控件，动画控件将在第十二章介绍．

6.2.1 Win32 控件的通知消息

较之传统的 Windows 3.x 控件，新的 Win32 控件更加复杂和先进．在新控件发送通知消息的同时，往往还需要附加一些数据来描述控件的状态．传统的 WM_COMMAND 消息通知机制显然不能完成这一任务，因为 WM_COMMAND 消息的 wParam 和 lParam 已经被占满了(见 6.1.1)，无法容纳新的数据．

在 Win32 中，采用新的 WM_NOTIFY 消息来实现新控件的消息通知机制．在该消息的 wParam 中含有控件的 ID，lParam 中则有一个指针，这个指针指向一个结构．这个结构要么是 NMHDR 结构，要么是一个以 NMHDR 结构作为第一个成员的扩充结构．通过 NMHDR 结构及其扩充结构可以传递附加数据．从理论上讲，可以通过扩充结构传送任意多的数据．需要指出的是，由于 NMHDR 结构是扩充结构的第一个成员，因此 lParam 中的指针即可以认为是指向 NMHDR 结构的，也可以认为指向包含 NMHDR 结构的扩充结构的．

NMHDR 结构如下所示：

```
typedef struct tagNMHDR {
HWND hwndFrom; //控件窗口的句柄
UINT idFrom; //控件的 ID
UINT code; //控件的通知消息码
```

} NMHDR;

一个典型的扩充结构如下所示，该结构用于列表视图控件的 LVN_KEYDOWN 通知消息．

typedef struct tagLV_KEYDOWN {
NMHDR hdr; //NMHDR 结构作为第一个成员
WORD wVKey;
UINT flags;
} LV_KEYDOWN;

有些控件通知消息是所有 Win32 控件共有的，这些消息在表 6.22 中列出．

表 6.22 Win32 控件共有的通知消息

通知消息码	含义
NM_CLICK	用户在控件上单击鼠标左键．
NM_DBLCLK	用户在控件上双击鼠标左键．
NM_RCLICK	用户在控件上单击鼠标右键．
NM_RDBLCLK	用户在控件上双击鼠标右键．
NM_RETURN	用户在控件上按回车键．
NM_SETFOCUS	控件获得输入焦点．
NM_KILLFOCUS	控件失去输入焦点．
NM_OUTOFMEMORY	内存不够．

WM_NOTIFY 的消息映射由宏 ON_NOTIFY 负责，该消息映射宏具有如下形式：

ON_NOTIFY(wNotifyCode, id, memberFxn)

参数 wNotifyCode 说明了通知消息码，参数 id 是控件的 ID，第三个参数则是消息处理函数名．消息处理函数应该按下面的形式声明，其中参数 pNotifyStruct 指向 NMHDR 及其扩充结构，参数 result 指向一个处理结果．

afx_msg void memberFxn(NMHDR * pNotifyStruct, LRESULT * result);

利用 ClassWizard 可以很方便地加入 WM_NOTIFY 消息映射及其处理函数，一个典型的 WM_NOTIFY 消息映射如下所示，其中 LVN_KEYDOWN 是 IDC_LIST1 列表视图控件发出的通知消息．

ON_NOTIFY(LVN_KEYDOWN, IDC_LIST1, OnKeydownList1)

消息处理函数 OnKeydownList1 的定义如下面所示．在函数中 ClassWizard 自动把 pNHHDR 指针强制转换成 LV_KEYDOWN 型并赋给 pLVKeyDow 指针，这样，在函数中可通过这两个指针访问 LV_KEYDOWN 扩充结构及其所含的 NMHDR 结构．另外，在函数返回时，ClassWizard 自动将处理结果赋 0 值．

```
void CMyDlg::OnKeydownList1(NMHDR* pNMHDR, LRESULT* pResult)
{
LV_KEYDOWN* pLVKeyDow = (LV_KEYDOWN*)pNMHDR;
```

```
// TODO: Add your control notification handler
// code here
*pResult = 0;
}
```

可以利用 ON_NOTIFY_RANGE **宏把多个** ID **连续的控件发出的相同消息映射到同一个处理函数上，具体形式如下，其中参数** id **和** idLast **分别说明明了一组连续的控件** ID **中的头一个和最后一个** ID**．**

ON_NOTIFY_RANGE(wNotifyCode, id, idLast, memberFxn)

相应的消息处理函数应按下面的形式声明，与普通的 WM_NOTIFY **消息处理函数相比，该函数多了一个参数** id **用来说明发送通知消息的控件** ID**．**

afx_msg void memberFxn(UINT id, NMHDR * pNotifyStruct, LRESULT * result);

ClassWizard **不支持** ON_NOTIFY_RANGE **宏，所以需要手工建立消息映射和消息处理函数．**

6.2.2 旋转按钮控件

旋转按钮 (Spin Button) **有时也被称为上下控件** (Up-Down Control)**．** Windows 95 **控制面板中的日期/时间程序中就有两个典型的旋转按钮，如图** 6.2 **所示．旋转按钮由两个箭头按钮组成，用户在箭头按钮上单击鼠标可以在某一范围内增加或减少某一个值．旋转按钮一般不会单独存在，而是和编辑框或静态正文组成一个多部件控件来共同显示和控制某一个值，用户可以用旋转按钮修改编辑框中的数字，也可以直接在编辑框中修改．例如，在图** 6.2 **中，在旋转按钮的左测有一个编辑框，用户可以在编辑框中直接输入新的年份，也可以用旋转按钮来增减编辑框中的年份．通常，把与旋转按钮在一块的编辑框或静态正文称为“伙伴”** (buddy)**．**

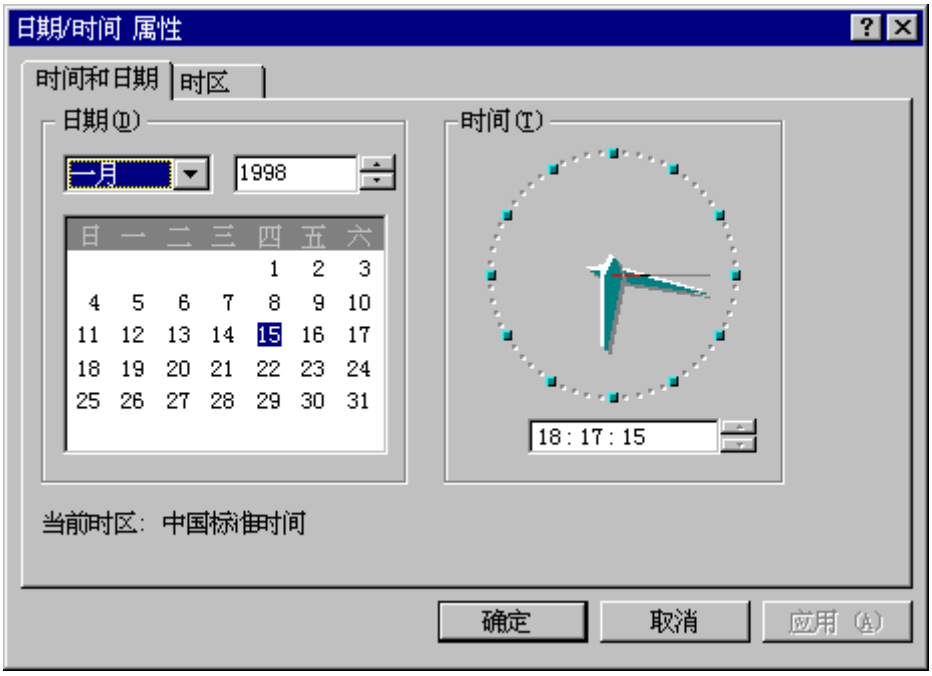


图 6.2 日期/时间程序中的旋转按钮

MFC 的 CSpinButtonCtrl 类封装了旋转按钮的功能．CSpinButtonCtrl 的成员函数 Create 负责创建控件，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 是如表 6.23 所示的各种控件风格的组合．

表 6.23 旋转按钮控件的风格

控件风格	含义
UDS_HORZ	指定一个水平旋转按钮．若不指定该风格则创建一个垂直的旋转按钮．
UDS_WRAP	当旋转按钮增大到超过最大值时，自动重置为最小值，当减小至低于最小值时，自动重置为最大值．
UDS_ARROWKEYS	当用户按下向下或向上箭头键时，旋转按钮值递增或递减．
UDS_SETBUDDYINT	旋转按钮将自动更新伙伴控件中显示的数值，如果伙伴控件能接受输入，则可在伙伴控件中输入新的旋转按钮值．
UDS_NOTHOUSANDS	伙伴控件中显示的数值每隔三位没有千位分隔符．
UDS_AUTOBUDDY	自动使旋转按钮拥有一个伙伴控件．
UDS_ALIGNRIGHT	旋转按钮在伙伴控件的右侧．
UDS_ALIGNLEFT	旋转按钮在伙伴控件的左侧．

除上表的风格外，一般还要为旋转按钮指定 WS_CHILD 和 WS_VISIBLE 风格．创建一个有伙伴的垂直旋转按钮控件，一般应指定的风格为 WS_CHILD|WS_VISIBLE|UDS_AUTOBUDDY|UDS_SETBUDDYINT．对于用对话框模板创建的旋转按钮控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Auto buddy，相当于指定了 UDS_AUTOBUDDY 风格．

在对话框模板中，可以方便地为旋转按钮指定一个伙伴控件．首先，应该在旋转按钮控件的属性对话框中选择 Auto buddy 和 Set buddy integer 属性，并在 Alignment 栏中选择 Left 或 Right，然后就可以确定伙伴控件了．需要指出的是，旋转按钮并不是把离它最近的控件作为伙伴的．伙伴的选择是以 tab 顺序为参照的，伙伴控件的 tab 顺序必需紧挨着按钮控件且比它小．例如，如果某一控件的 tab 顺序是 3，而旋转按钮的 tab 顺序是 4，则不论这两个控件距离有多远，在程序运行时，旋转按钮都会自动与该控件结合在一起，形成伙伴关系．

提示：在本章的开头说过，用 ClassWizard 无法为 Win32 控件创建数据变量．但我们可以为旋转按钮的伙伴控件(如编辑框)创建一个数据变量，该变量可看成是旋转按钮的数据变量．/

通过 CSpinButtonCtrl 的成员函数，可以对旋转按钮进行查询和设置：

用 GetRange 和 SetRange 来查询和设置旋转按钮值的范围，缺省时值的范围是 1-100．这两个函数的声明为 void GetRange(int &lower, int& upper) const;void SetRange(int nLower, int nUpper);第一个参数是最小值，该值不能小于 UD_MINVAL，第二个参数是最大值，该值不能大于 UD_MAXVAL．值的范围不能超过 UD_MAXVAL．

用 GetPos 和 SetPos 来查询和设置旋转按钮的当前值．函数的声明为 int GetPos( ) const;int SetPos(int nPos);

用 GetBase 和 SetBase 来查询和设置旋转按钮值的计数制．函数的声明为 UINT GetBase( ) const;int SetBase(int nBase);如果参数 nBase 是 10，则伙伴控件中显示的数值是十进制的，如果 nBase 是 16，则是十六进制的．

用 GetBuddy 和 SetBuddy 来查询和设置旋转按钮的伙伴．上面已讲了在对话框模板中设置伙伴控件的方法，如果是用 Create 手工创建旋转按钮，则可以用 SetBuddy 来设置伙伴．函数的声明为 CWnd* GetBuddy( ) const;CWnd* SetBuddy(CWnd* pWndBuddy);参数 pWndBuddy 是指向伙伴控件对象的 CWnd 型指针．

可以用 GetAccel 和 SetAccel 来查询和设置旋转按钮的加速值．在平时，在旋转按钮上按一下只会增/减一个单位，而当按住按钮超过一定时间时，递增或递减的幅度将会加大到指定的加速值，从而加快了增减的速度．如果对缺省的加速值不满意，可以用 SetAccel 设置新的加速值．可以有一套以上的加速值．函数的声明为 UINT GetAccel(int nAccel, UDACCEL* pAccel) const;BOOL SetAccel(int nAccel, UDACCEL* pAccel);参数 nAccel 指定了 UDACCEL 结构数组的大小．参数 pAccel 指向一个 UDACCEL 结构数组．UDACCEL 结构含有加速值的信息，其定义如下 typedef struct {

int nSec; //加速值生效需要的时间(以秒为单位)

int nInc; //加速值

} UDACCEL;

旋转按钮常被认为是一个简化的滚动条．除了表 6.22 列出的通知消息外，旋转按钮特有的滚动通知消息是通过 WM_HSCROLL 和 WM_VSCROLL 消息发出的．消息处理函数 OnHScroll 或 OnVScroll 分别用来处理水平或垂直旋转按钮的事件通知．由于伙伴控件中的内容会自动随旋转按钮变化，所以旋转按钮的通知消息意义不大．如果非要处理通知消息，一个典型的 OnVscroll 函数如下所示：

```
void CMyDialog::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
CSpinButtonCtrl* pSpin=(CSpinButtonCtrl*)pScrollBar;
int nPosition;
if(pSpin= =&m_Spin) //判断是否是该旋转按钮发来的消息
{
nPosition=m_Spin.GetPos( ); //获取旋转按钮的当前值
```

. . . . . .
}
. . . . . .
}

6.2.3 滑尺控件

滑尺(Slider)有时也被称作轨道条(Trackbar)，在轨道条中有一个滑尺，在轨道条上通常会标有刻度，用户通过移动滑尺，可以在一个指定的范围内选择一个不精确的值．轨道条可用来调节一个模拟量，也可以用来在一些离散值中进行选择．在 Windows 95 中，大量使用了轨道条控件，例如，在控制面板中的键盘和鼠标设置程序中就使用了轨道条控件，如图 6.3 所示．轨道条不仅接受鼠标输入，也可以接受象左右箭头键、PgUp 和 PgDown 这样的键盘输入．

图 6.3 鼠标设置程序中的轨道条控件

与选择按钮不同，轨道条是一种模糊型的输入控件，用户不需要进行精确的选择，只要大致调整一下大小就行了．轨道条的这种特性非常符合人的行为习惯，因而在有些情况下是很有用，例如，对于音量的调节，显然用轨道条比用旋转按钮更符合人的日常习惯．

轨道条的滑尺的移动具有离散性．例如，如果指定轨道条的范围是 5，那么滑尺只能在包括轨道条两端在内的 6 个均匀的位置上移动．当然，如果范围很大，则用户就感觉不出是离散的了．

轨道条控件与传统的滚动条控件有很多相似之处，实际上，前者是对后者的一种改进．除了表 6.22 列出的通知消息外，轨道条控件是依靠

WM_HSCROLL 和 WM_VSCROLL 来发送与滑尺有关的通知消息的，并且通知消息与滚动条极为相似．通知消息包括 TB_BOTTOM、TB_LINEDOWN 、 TB_LINEUP 、 TB_TOP 、 TB_PAGEDOWN 、TB_PAGEUP 、 TB_ENDTRACK 、 TB_THUMBPOSITION 、TB_THUMBTRACK．对照滚动条的通知消息，读者不难明白这些消息码的含义．其中前四个消息只有在用键盘移动滑尺时才会发出，最后两个消息只有在用鼠标拖动滑尺时才会发出．与滚动条不同的是，Windows 会自动把滑尺移动到新位置上．

MFC 的 CSliderCtrl 类封装了轨道条．CSliderCtrl 类的 Create 成员函数负责控件的创建，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数参数 dwStyle 是如表 6.24 所示的各种控件风格的组合．

表 6.24 轨道条控件的风格

控件风格	含义
TBS_HORZ	指定一个水平轨道条．该风格是默认的．
TBS_VERT	指定一个垂直轨道条．
TBS_AUTOTICKS	在范围设定后，自动为轨道条加上刻度．
TBS_NOTICKS	轨道条无刻度．
TBS_BOTTOM	在水平轨道条的底部显示刻度，可与 TBS_TOP 一起使用．
TBS_TOP	在水平轨道条的顶部显示刻度，可与 TBS_BOTTOM 一起使用．
TBS_RIGHT	在垂直轨道条的右侧显示刻度，可与 TBS_LEFT 一起使用．
TBS_LEFT	在垂直轨道条的左侧显示刻度，可与 TBS_RIGHT 一起使用．
TBS_BOTH	在轨道条的上下部或左右两侧都显示刻度．
TBS_ENABLESELRANGE	在轨道条中显示一个选择范围．

除上表的风格外，一般还要为轨道条指定 WS_CHILD 和 WS_VISIBLE 风格．要创建一个具有刻度的水平轨道条，一般应指定风格为 WS_CHILD|WS_VISIBLE|TBS_HORZ| TBS_AUTOTICKS．对于用对话框模板创建的轨道条控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Autoticks，相当于指定了 TBS_AUTOTICKS 风格．

通过调用 CSliderCtrl 类的成员函数，可以对轨道条进行查询和设置：

用 GetRange 和 SetRange 来查询和设置轨道条的范围，缺省的范围是 0-100．函数的声明为 void GetRange(int& nMin, int& nMax) const;void SetRange(int nMin, int nMax, BOOL bRedraw = FALSE);参数 nMin 和 nMax 分别是最小和最大值，参数 bRedraw 为 TRUE 时将重绘控件．

用 GetPos 和 SetPos 来查询和设置轨道条的当前值．函数的声明为 int GetPos( ) const;void SetPos(int nPos);

用 GetLineSize 和 SetLineSize 来查询和设置在按一下左箭头键或右箭头键时滑尺的移动量，该移动量的缺省值是 1 个单位．函数的声明为 int GetLineSize( ) const;int SetLineSize(int nSize);

用 GetPageSize 和 SetPageSize 来查询和设置滑尺的块移动量，块移动量是指当按下 PgUp 或 PgDown 键时滑尺的移动量．函数的声明为 int GetPageSize( ) const;int SetPageSize(int nSize);

用 SetTicFreq 设置轨道条的刻度的频度．缺省的频度是每个单位都有一个刻度，在范围较大时，为了使刻度不至于过密，需要调用该函数设置一个合理的频度．函数的声明为 void SetTicFreq(int nFreq);参数 nFreq 说明了两个刻度之间的间隔．

用函数 SetTic 来在指定位置设置刻度．Windows 自动显示的刻度是均匀的，利用该函数可以人为设置不均匀的刻度，该函数的声明为 BOOL SetTic(int nTic);

用函数 ClearTics 来清除所有的刻度．该函数的声明为 void ClearTics(BOOL bRedraw = FALSE);

6.2.4 进度条控件

进度条(Progress Bar)的用途是向用户显示程序的进度．进度条是 Win32 控件中最简单的控件，只需少数设置即可．Windows 95 中使用进度条的一个例子是磁盘扫描(ScanDisk)程序，如图 6.4 所示．进度条显示的数据是不精确的，它是一种模糊型的输出控件．

图 6.4 磁盘扫描程序中的进度条

MFC 的 CProgressCtrl 类封装了进度条控件．该类的 Create 成员函数负责创建控件，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

进度条没有专门的风格，所以参数 dwStyle 只能指定普通的窗口风格，一般只需指定 WS_CHILD|WS_VISIBLE 就可以了．

CProgressCtrl 类提供了少量的成员函数用来设置进度条：

void SetRange(int nLower, int nUpper);该函数用来设置进度条的范围．参数 nLow 和 nUpper 分别指定了最小和最大值，缺省时进度条的范围是 0-100．

int SetPos(int nPos);用来设置进度条的当前进度．函数返回的是进度条的前一个进度．

int StepIt( );使进度增加一个步长，步长值是由 SetStep 函数设置的，缺省的步长值是 10．函数返回进度条的前一个进度．

int SetStep(int nStep);用来设置步长值．函数返回原来的步长值．

6.2.5 树形视图控件

树形视图(Tree View)是一种特殊的列表，它能以树形分层结构显示数据．在 Windows 95 的资源管理器(Windows Explorer)的左侧窗口中就有一个用于显示目录的典型的树形视图，如图 6.5 所示．在树形视图中，每个表项显示一个标题(Label)，有时还会显示一幅图象，图象和标题分别提供了对数据的形象和抽象描述．通过图 6.5 可以看出，树形视图可以很清楚的显示出数据的分支和层次关系．由此可见，树形视图非常适合显示象目录，网络结构等这样的复杂数据．传统的列表框不能分层显示数据，因此树形视图可以看作是对列表框的一种重要改进．

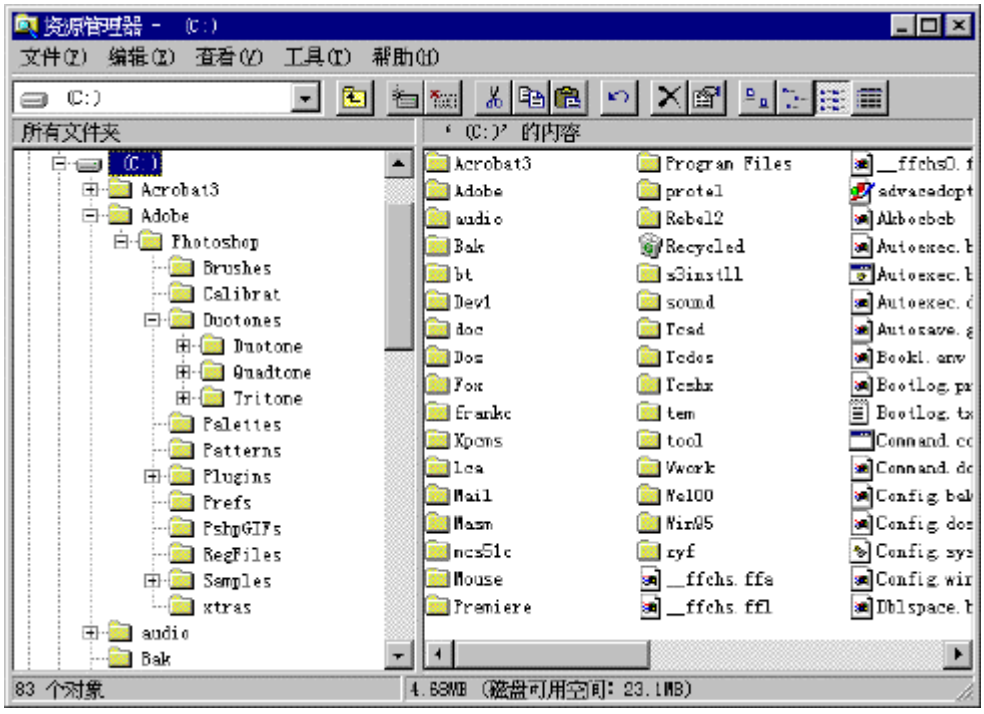


图 6.5 资源管理器中的树形视图和列表视图

树形视图是一种复杂的控件，它的复杂性体现在数据项之间具有分支和层次关系．例如，如果要向树形视图中加入新的项，则必需描述出该项与树形视图中已有项的相互关系，而不可能象往列表框中加入新项那样，调用一下 AddString 就完事了．另外，树形视图可以在每一项标题

的左边显示一幅图象，这使控件显得更加形象生动，但同时也增加了控件的复杂程度．

在讨论如何使用树形视图控件以前，有必要先介绍一下与该控件有关的一些数据类型：

HTREEITEM 型句柄．Windows 用 HTREEITEM 型句柄来代表树形视图的一项，程序通过 HTREEITEM 句柄来区分和访问树形视图的各个项．

TV_ITEM 结构．该结构用来描述一个表项，它包含了表项的各种属性，其定义如下

```
typedef struct _TV_ITEM
{ tvi
UINT    mask; /*包含一些屏蔽位(下面的括号中列出)的组合，用来表明结构的哪些成员是有效的*/
HTREEITEM hItem; //表项的句柄(TVIF_HANDLE)
UINT state; //表项的状态(TVIF_STATE)
UINT stateMask; //状态的屏蔽组合(TVIF_STATE)
LPSTR pszText; //表项的标题正文(TVIF_TEXT)
int cchTextMax; //正文缓冲区的大小(TVIF_TEXT)
int iImage; //表项的图象索引(TVIF_IMAGE)
int iSelectedImage; //选中的项的图象索引(TVIF_SELECTEDIMAGE)
int cChildren; /*表明项是否有子项(TVIF_CHILDREN)，为 1 则有，为 0 则没有*/
LPARAM lParam; //一个 32 位的附加数据(TVIF_PARAM)
} TV_ITEM, FAR *LPTV_ITEM;
```

如果要使树形视图的表项显示图象，需要为树形视图建立一个位图序列，这时，iImage 说明表项显示的图象在位图序列中的索引，iSelectedImage 则说明了选中的表项应显示的图象，在绘制图标时，树形视图可以根据这两个参数提供的索引在位图序列中找到对应的位图．lParam 可用来放置与表项相关的数据，这常常是很有用的．state 和 stateMask 的常用值在表 6.25 中列出，其中 stateMask 用来说明要获取或设置哪些状态．

表 6.25 树形视图表项项的常用状态

状态	对应的状态屏蔽	含义
TVIS_SELECTED	同左	项被选中．
TVIS_EXPANDED	同左	项的子项被展开．
TVIS_EXPANDEDONCE	同左	项的子项曾经被展开过．
TVIS_CUT	同左	项被选择用来进行剪切和粘贴操作．
TVIS_FOCUSED	同左	项具有输入焦点．
TVIS_DROPHILITED	同左	项成为拖动操作的目标．

TV_INSERTSTRUCT 结构．在向树形视图中插入新项时要用到该结

构，其定义为

typedef struct _TV_INSERTSTRUCT {
HTREEITEM hParent; //父项的句柄
HTREEITEM hInsertAfter; //说明应插入到同层中哪一项的后面
TV_ITEM item;
} TV_INSERTSTRUCT;

如果 hParent 的值为 TVI_ROOT 或 NULL，那么新项将被插入到树形视图的最高层(根位置)．hInsertAfter 的值可以是 TVI_FIRST、TVI_LAST 或 TVI_SORT，其含义分别是将新项插入到同一层中的开头、最后或排序插入．

NM_TREEVIEW 结构．树形视图的大部分通知消息都会附带指向该结构的指针以提供一些必要的信息．该结构的定义为

typedef struct _NM_TREEVIEW { nmtv
NMHDR hdr; //标准的 NMHDR 结构
UINT action; //表明是用户的什么行为触发了该通知消息
TV_ITEM itemOld; //旧项的信息
TV_ITEM itemNew; //新项的信息
POINT ptDrag; //事件发生时鼠标的客户区坐标
} NM_TREEVIEW;

TV_KEYDOWN 结构．提供与键盘事件有关的信息．该结构的定义为

pedef struct _TV_KEYDOWN { tvkd
NMHDR hdr; //标准的 NMHDR 结构
WORD wVKey; //虚拟键盘码
UINT flags; //为 0
} TV_KEYDOWN;

TV_DISPINFO 结构．提供与表项的显示有关的信息．该结构的定义为

typedef struct _TV_DISPINFO { tvdi
NMHDR hdr;
TV_ITEM item;
} TV_DISPINFO;

MFC 的 CTreeCtrl 类封装了树形视图．该类的 Create 成员函数负责控件的创建，该函数的声明为

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

其中参数 dwStyle 是如表 6.26 所示的控件风格的组合．

表 6.26 树形视图的风格

控件风格	含义
TVS_HASLINES	在父项与子项间连线以清楚地显示结构．
TVS_LINESATROOT	只在根部画线．
TVS_HASBUTTONS	显示带有“+”或“-”的小方框来表示某项

	能否被展开或已展开.
TVS_EDITLABELS	用户可以编辑表项的标题.
TVS_SHOWSELALWAYS	即使控件失去输入焦点,仍显示出项的选择状态.
TVS_DISABLEDRAGDROP	不支持拖动操作.

除上表的风格外，一般还要指定 WS_CHILD 和 WS_VISIBLE 窗口风格．对于用对话框模板创建的树形视图控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框中选择 Has buttons，相当于指定了 TVS_HASBUTTONS 风格．

CTreeCtrl 类提供了大量的成员函数．对于常用的函数，这里结合实际应用的需要，介绍如下：

向树形视图中插入新的表项．首先应提供一个 TV_INSERTSTRUCT 结构并在该结构中对插入项进行描述．如果要在树形视图中显示图象，则应该先创建一个 CImageList 对象并使该对象包含一个位图序列，然后调用 SetImageList 为树形视图设置位图序列．然后调用 InsertItem 函数把新项插入到树形视图中．函数的声明为

CImageList* SetImageList(CImageList * pImageList, int nImageListType);参数 pImageList 指向一个 CImageList 对象，参数 nImageListType 一般应为 TVSIL_NORMAL．

HTREEITEM InsertItem(LPTV_INSERTSTRUCT lpInsertStruct);参数 lpInsertStruct 指向一个 TV_INSERTSTRUCT 结构．函数返回新插入项的句柄．

用 DeleteItem 来删除指定项，用 DeleteAllItems 删除所有项．函数的声明为 BOOL DeleteItem(HTREEITEM hItem);BOOL DeleteAllItems( ); 操作成功则函数返回 TRUE，否则返回 FALSE．

树形视图控件会根据用户的输入自动展开或折叠子项．但有时需要在程序中展开或折叠指定项,则应该调用 Expand,该函数的声明为 BOOL Expand(HTREEITEM hItem, UINT nCode);参数 hItem 指定了要展开或折叠的项．参数 nCode 是一个标志，指定了函数应执行的操作，它可以是 TVE_COLLAPSE(折叠)、TVE_COLLAPSERESET(折叠并移走所有的子项)、TVE_EXPAND(展开)或 TVE_TOGGLE(在展开和折叠状态之间翻转)．

要查询或设置选择项，应调用 GetSelectedItem 或 SelectItem．函数的声明为 HTREEITEM GetSelectedItem( );BOOL SelectItem(HTREEITEM hItem);

要对指定的项查询或设置，可调用 GetItem 和 SetItem．用这两个功能强大的函数，几乎可以查询和设置项的所有属性，包括表项的正文、图像及选择状态．函数的声明为 BOOL GetItem(TV_ITEM* pItem);BOOL SetItem(TV_ITEM* pItem);参数 pItem 是指向 TV_ITEM 结构的指针，函数是通过该结构来查询或设置指定项的，在调用函数前应该使该结构的 hItem 成员有效以指定表项．CTreeCtrl 还提供了一系列函

数可完成 GetItem 和 SetItem 的部分功能，其中 GetItemState、GetItemText、GetItemData、GetItemImage 和 ItemHasChildren 函数用于查询，SetItemState、SetItemText、SetItemData 和 SetItemImage 函数用于设置．

在使用树形视图控件时，一个经常遇到的问题是对于一个已知表项，如何找到与该项有某种关系的项，例如，父项、子项、兄弟项、下一个或前一个可见的项．利用功能强大的 GetNextItem 函数，可以解决这个问题．该函数也可以用来搜索具有某种状态的表项．GetNextItem 在遍历树形视图时是很有用的，它的声明为 HTREEITEM GetNextItem(HTREEITEM hItem, UINT nCode);参数 hItem 指定了一个项．参数 nCode 是一个标志，标明了与指定项的关系，nCode 可以是如表 6.27 所示的各种标志．如果找到相关的项，函数返回该项的句柄，否则函数返回 NULL．

表 6.27 关系标志

标志	含义
TVGN_CARET	返回当前的选择项．
TVGN_CHILD	返回指定表项的子项．
TVGN_DROPHILITE	返回拖动操作的目标项．
TVGN_FIRSTVISIBLE	返回第一个可见项．
TVGN_NEXT	返回指定项的下一个兄弟项(Sibling Item)．
TVGN_NEXTVISIBLE	返回指定项的后一个可见项．
TVGN_PARENT	返回指定项的父项．
TVGN_PREVIOUS	返回指定项的前一个兄弟项．
TVGN_PREVIOUSVISIBLE	返回指定项的前一个可见项．
TVGN_ROOT	返回位于最高层(根位置)的第一个表项．

CTreeCtrl 类提供了一系列的成员函数来完成 GetNextItem 的某一项功能，包括 GetRootItem、GetFirstVisibleItem、GetNextVisibleItem、GetPrevVisibleItem、GetChildItem、GetNextSiblingItem、GetPrevSiblingItem、GetParentItem、GetSelectedItem 和 GetDropHilightItem．

除了表 6.22 列出的控件消息外，树形视图控件还会发送自己特有的通知消息，其中常用的有下面这几个：

TVN_SELCHANGING 和 TVN_SELCHANGED．在用户改变了对表项的选择时，控件会发送这两个消息．消息会附带一个指向 NM_TREEVIEW 结构的指针，程序可从该结构中获得必要的信息．两个消息都会在该结构的 itemOld 成员中包含原来的选择项的信息，在 itemNew 成员中包含新选择项的信息，在 action 成员中表明是用户的什么行为触发了该通知消息(若是 TVC_BYKEYBOARD 则表明是键盘，若是 TVC_BYMOUSE 则表明是鼠标，若是 TVC_UNKNOWN 则表示未知)．两个消息的不同之处在于，如果 TVN_SELCHANGING 的消息处理函数返回 TRUE，那么就阻止选择的改变，如果返回 FALSE，则允许改变．

TVN_KEYDOWN．该消息表明了一个键盘事件．消息会附带一个指

向 TV_KEYDOWN 结构的指针，通过该结构程序可以获得按键的信息．

TVN_BEGINLABELEDIT 和 TVN_ENDLABELEDIT．分别在用户开始编辑和结束编辑项的标题时发送．消息会附带一个指向 TV_DISPINFO 结构的指针，程序可从该结构中获得必要的信息．在前者的消息处理函数中，可以调用 GetEditControl 成员函数返回一个指向用于编辑标题的编辑框的指针，如果处理函数返回 FALSE，则允许编辑，如果返回 TRUE，则禁止编辑．在后者的消息处理函数中，TV_DISPINFO 结构中的 item.pszText 指向编辑后的新标题，如果 pszText 为 NULL，那么说明用户放弃了编辑，否则，程序应负责更新项的标题，这可以由 SetItem 或 SetItemText 函数来完成．

树形视图控件还可以支持拖放操作，限于篇幅，这里就不作介绍了．

6.2.6 列表视图控件

列表视图(List View)用来成列地显示数据．在 Windows 95 的资源管理器的右侧窗口中就有一个典型的列表视图，如图 6.5 所示．列表视图的表项通常包括图标(Icon)和标题(Label)两部分，它们分别提供了对数据的形象和抽象描述．列表视图控件是对传统的列表框的重大改进，它能够以下列四种格式显示数据．读者可以在资源管理器中的视图(View)菜单中切换列表视图的显示格式，来看看四种格式的不同之处．

大图标格式(Large Icons)．可逐行显示多列表项，图标的大小可由应用程序指定，通常是 32×32 像素，在图标的下面显示标题．

小图标格式(Small Icons)．可逐行显示多列表项，图标的大小可由应用程序指定，通常是 16×16 像素，在图标的右面显示标题．表项以行的方式组织．

列表格式(List)．与小图标格式类似．不同之处在于表项是逐列多列显示的．

报告格式(Report 或 Details)．每行仅显示一个表项，在标题的左边显示一个图标，表项可以不显示图标而只显示标题．表项的右边可以附加若干列子项(Subitem)，子项只显示正文．在控件的顶端还可以显示一个列表头用来说明各列的类型．列表视图的报告格式很适合显示报表(如数据库报表)．

在讨论如何使用列表视图控件以前，显向读者介绍一下与该控件有关的一些数据类型：

LV_COLUMN 结构．该结构仅用于报告式列表视图，用来描述表项的某一列．要想向表项中插入新的一列，需要用到该结构．LV_COLUMN 结构的定义为

```
typedef struct _LV_COLUMN {
UINT mask; //屏蔽位的组合(见下面括号)，表明哪些成员是有效的．
int  fmt;  /*该列的表头和子项的标题显示格式(LVCF_FMT)．可以是 LVCFMT_CENTER、LVCFMT_LEFT 或 LVCFMT_RIGHT*/
int cx; //以像素为单位的列的宽度(LVCF_FMT)
LPTSTR pszText; //指向存放列表头标题正文的缓冲区(LVCF_TEXT)
int cchTextMax; //标题正文缓冲区的长度(LVCF_TEXT)
```

int iSubItem; //说明该列的索引(LVCF_SUBITEM)

} LV_COLUMN;

LV_ITEM 结构．该结构用来描述一个表项或子项，它包含了项的各种属性，其定义为

typedef struct _LV_ITEM {

UINT mask; //屏蔽位的组合(见下面括号)，表明哪些成员是有效的

int iItem; //从 0 开始编号的表项索引(行索引)

int iSubItem; /*从 1 开始编号的子项索引(列索引)，若值为 0 则说明该成员无效，结构描述的是一个表项而不是子项*/

UINT state; //项的状态(LVIF_STATE)

UINT stateMask; //项的状态屏蔽

LPTSTR pszText; //指向存放项的正文的缓冲区(LVIF_TEXT)

int cchTextMax; //正文缓冲区的长度(LVIF_TEXT)

int iImage; //图标的索引(LVIF_IMAGE)

LPARAM lParam; // 32 位的附加数据(LVIF_PARAM)

} LV_ITEM;

其中 lParam 成员可用来存储与项相关的数据，这在有些情况下是很有用的．state 和 stateMask 的值如表 6.28 所示，stateMask 用来说明要获取或设置哪些状态．

表 6.28 列表视图的状态

状态	对应的状态屏蔽	含义
LVIS_CUT	同左	项被选择用来进行剪切和粘贴操作．
LVIS_DROPHILITED	同左	项成为拖动操作的目标．
LVIS_FOCUSED	同左	项具有输入焦点．
LVIS_SELECTED	同左	项被选中．

NM_LISTVIEW 结构．该结构用于存储列表视图的通知消息的有关信息，大部分列表视图的通知消息都会附带指向该结构的指针．NM_LISTVIEW 的定义为

typedef struct tagNM_LISTVIEW {

NMHDR hdr; //标准的 NMHDR 结构

int iItem; //表项的索引，若为-1 则无效

int iSubItem; //子项的索引，若为 0 则无效

UINT uNewState; //项的新状态

UINT uOldState; //项原来的状态

UINT uChanged; /*取值与 LV_ITEM 的 mask 成员相同，用来表明哪些状态发生了变化*/

POINT ptAction; //事件发生时鼠标的客户区坐标

LPARAM lParam; //32 位的附加数据

} NM_LISTVIEW;

LV_DISPINFO 结构．该结构包含了与项的显示有关的信息，其定义为

typedef struct tagLV_DISPINFO {
NMHDR hdr;
LV_ITEM item;
} LV_DISPINFO;

LV_KEYDOWN **结构．该结构包含一些与键盘有关的信息，其定义为**

typedef struct tagLV_KEYDOWN {
NMHDR hdr;
WORD wVKey; //**虚拟键盘码**
UINT flags; //**总为** 0
} LV_KEYDOWN;

MFC **的** CListCtrl **类封装了列表视图控件．该类的** Create **函数负责创建控件，函数的声明为**

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

其中参数 dwStyle **是如表** 6.29 **所示的控件风格的组合．**

表 6.29 **列表视图的风格**

控件风格	**含义**
LVS_ALIGNLEFT	**当显示格式是大图标或小图标时，标题放在图标的左边．缺省情况下标题放在图标的下面．**
LVS_ALIGNTOP	**当显示格式是大图标或小图标时，标题放在图标的上边．**
LVS_AUTOARRANGE	**当显示格式是大图标或小图标时，自动排列控件中的表项．**
LVS_EDITLABELS	**用户可以修改标题．**
LVS_ICON	**指定大图标显示格式．**
LVS_LIST	**指定列表显示格式．**
LVS_NOCOLUMNHEADER	**在报告格式中不显示列的表头．**
LVS_NOLABELWRAP	**当显示格式是大图标时，使标题单行显示．缺省时是多行显示．**
LVS_NOSCROLL	**列表视图无滚动条．**
LVS_NOSORTHEADER	**报告列表视图的表头不能作为排序按钮使用．**
LVS_OWNERDRAWFIXED	**由控件的拥有者负责绘制表项．**
LVS_REPORT	**指定报告显示格式．**
LVS_SHAREIMAGELISTS	**使列表视图共享图像序列．**
LVS_SHOWSELALWAYS	**即使控件失去输入焦点，仍显示出项的选择状态．**
LVS_SINGLESEL	**指定一个单选择列表视图．缺省时可以多项选择．**

LVS_SMALLICON	指定小图标显示格式.
LVS_SORTASCENDING	按升序排列表项.
LVS_SORTDESCENDING	按降序排列表项.

除上表的风格外，一般还要指定 WS_CHILD 和 WS_VISIBLE 窗口风格. 风格组合 WS_CHILD|WS_VISIBLE|LVS_REPORT|LVS_AUTOARRANGE|LVS_EDITLABLES|LVS_SINGLESEL 将指定一个自动排列的、可编辑标题的、单选择报告式列表视图控件．要指定大图标、小图标或列表式的列表视图控件，则应该把 LVS_REPORT 换成 LVS_ICON、LVS_SMALLICON 或 LVS_LIST.

对于用对话框模板创建的列表视图控件，可以在控件的属性对话框中指定上表中列出的控件风格。例如，在属性对话框的 Styles 页的 View 栏中选择 Icon，相当于指定了 LVS_ICON 风格.

CListCtrl 类提供了大量的成员函数．在这里，我们结合实际应用来介绍一些常用的函数：

列的插入和删除．在以报告格式显示列表视图时，一般会显示一列表项和多列子项．在初始化列表视图时，先要调用 InsertColumn 插入各个列，该函数的声明为 int InsertColumn(int nCol, const LV_COLUMN* pColumn);其中参数 nCol 是新列的索引，参数 pColumn 指向一个 LV_COLUMN 结构，函数根据该结构来创建新的列．若插入成功，函数返回新列的索引，否则返回-1．要删除某列，应调用 DeleteColumn 函数，其声明为 BOOL DeleteColumn(int nCol);

表项的插入．要插入新的表项，应调用 InsertItem．如果要显示图标，则应该先创建一个 CImageList 对象并使该对象包含用作显示图标的位图序列．然后调用 SetImageList 来为列表视图设置位图序列．函数的声明为

int InsertItem(const LV_ITEM* pItem);参数 pItem 指向一个 LV_ITEM 结构，该结构提供了对表项的描述．若插入成功则函数返回新表项的索引，否则返回-1.

CImageList* SetImageList(CImageList* pImageList, int nImageList);参数 pImageList 指向一个 CImageList 对象，参数 nImageList 用来指定图标的类型，若其值为 LVSIL_NORMAL，则位图序列用作显示大图标，若值为 LVSIL_SMALL，则位图序列用作显示小图标．可用该函数同时指定一套大图标和一套小图标.

要删除某表项，应调用 DeleteItem，要删除所有的项，应调用 DeleteAllItems．一旦表项被删除，其子项也被删除．函数的声明为 BOOL DeleteItem(int nItem);BOOL DeleteAllItems( );

调用 GetItemText 和 SetItemText 来查询和设置表项及子项显示的正文．SetItemText 的一个重要用途是对子项进行初始化．函数的声明为 int GetItemText(int nItem, int nSubItem, LPTSTR lpszText, int nLen) const;CString GetItemText(int nItem, int nSubItem) const;BOOL SetItemText(int nItem, int nSubItem, LPTSTR lpszText);其中参数 nItem 是

表项的索引(行索引)，nSubItem 是子项的索引(列索引)，若 nSubItem 为 0 则说明函数是针对表项的．参数 lpszText 指向正文缓冲区，参数 nLen 说明了缓冲区的大小．第二个版本的 GetItemText 返回一个含有项的正文的 Cstring 对象．

调用 GetItem 和 SetItem 来查询和设置．用这两个功能强大的函数，几乎可以查询和设置指定项的所有属性，包括正文、图标及选择状态．函数的声明为 BOOL GetItem(LV_ITEM* pItem) const;BOOL SetItem(const LV_ITEM* pItem);参数 pItem 是指向 LV_ITEM 结构的指针，函数是通过该结构来查询或设置指定项的，在调用函数前应该使该结构的 iItem 或 iSubItem 成员有效以指定表项或子项．CListCtrl 还提供了一系列函数可完成 GetItem 和 SetItem 的部分功能，其中 GetItemState、GetItemText 和 GetItemData 函数用于查询，SetItemState、SetItemText 和 SetItemData 函数用于设置．

要查询表项的数目，应该调用 GetItemCount，其声明为 int GetItemCount( );

要寻找与指定表项项相关的表项，或寻找具有某种状态的表项，应该调用 GetNextItem．该函数的一个重要用处是搜索被选择的表项．函数的声明为 int GetNextItem(int nItem, int nFlags) const;参数 nItem 是指定项的索引，参数 nFlags 是如表 6.30 所示的标志，用来指定查询的关系．函数返回搜索到的表项的索引，若未找到则返回-1．

表 6.30 关系标志

标志	含义
LVNI_ABOVE	返回位于指定表项上方的表项．
LVNI_ALL	缺省标志，返回指定表项的下一个表项(以索引为序)．
LVNI_BELOW	返回位于指定表项下方的表项．
LVNI_TOLEFT	返回位于指定表项左边的表项．
LVNI_TORIGHT	返回位于指定表项右边的表项．
LVNI_DROPHILITED	返回拖动操作的目标表项．
LVNI_FOCUSED	返回具有输入焦点的表项．
LVNI_SELECTED	返回被选择的表项．

要对表项进行排列、排序和搜索，可分别调用 Arrange、SortItems 和 FindItems 函数来完成．

有时需要在列表视图创建后动态地改变其显示格式，例如，资源管理器中的列表视图就可以在四中显示格式之间切换．改变显示格式其实就是改变列表视图的风格，要改变控件的风格，应先调用::GetWindowLong 获取控件原来的风格，并对其进行修改，然后调用::SetWindowLong 设置新的风格．这两个函数不是成员函数，而是 Windows API 函数，用来查询和设置窗口的风格．

除了表 6.22 列出的控件消息外，列表视图控件还会发送自己特有的通知消息，其中常用的有下面这几个：

LVN_ITEMCHANGING 和 LVN_ITEMCHANGED．当列表视图的状

态发生变化时，会发送这两个通知消息．例如，当用户选择了新的表项时，程序就会收到这两个消息．消息会附带一个指向 NM_LISTVIEW 结构的指针，消息处理函数可从该结构中获得状态信息．两个消息的不同之处在于，前者的消息处理函数如果返回 TRUE，那么就阻止选择的改变，如果返回 FALSE，则允许改变．

LVN_KEYDOWN．该消息表明了一个键盘事件．消息会附带一个指向 LV_KEYDOWN 结构的指针，通过该结构程序可以获得按键的信息．

LVN_BEGINLABELEDIT 和 LVN_ENDLABELEDIT．分别在用户开始编辑和结束编辑标题时发送．消息会附带一个指向 LV_DISPINFO 结构的指针．在前者的消息处理函数中，可以调用 GetEditControl 成员函数返回一个指向用于编辑标题的编辑框的指针，如果处理函数返回 FALSE，则允许编辑，如果返回 TRUE，则禁止编辑．在后者的消息处理函数中，LV_DISPINFO 结构中的 item.pszText 指向编辑后的新标题，如果 pszText 为 NULL，那么说明用户放弃了编辑，否则，程序应负责更新表项的标题，这可以由 SetItem 或 SetItemText 函数来完成．

列表视图控件还可以支持拖放操作，这里就不详细介绍了．

6.2.7 测试新型 Win32 控件的一个例子

本小节将向读者提供一个测试 Win32 控件的例子．测试程序名为 Ctrl32，其界面如图 6.6 所示，该程序对前面介绍的五种 Win32 控件均进行了测试：

对树形视图的测试着重演示了如何在各个层次上加入表项以及如何使表项显示图象，表项在平常状态下和选中状态下将显示不同的图象．

对列表视图的测试包括如何生成一个报告式列表视图，如何在四个显示格式间切换以及如何使表项显示图标．读者可以在列表视图下面的下拉列表式组合框选择不同的显示格式．

轨道条的测试包括初始化及 WM_HSCROLL 消息的处理．进度条的进度将会随着滑尺位置的改变而改变．

演示了如何为旋转按钮指定伙伴控件以及旋转按钮的初始化．

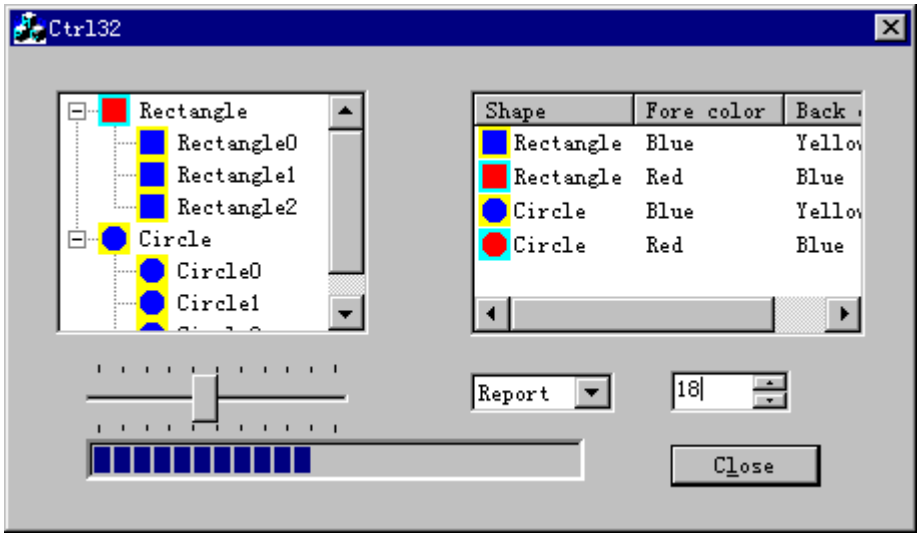


图 6.6 Ctrl32 测试程序

请读者按下列步骤操作：

用 AppWizard 建立一个基于对话框的 MFC 应用程序，程序名为 Ctrl32。

插入两个位图(Bitmap)资源，其 ID 分别是 IDB_BITMAP1 和 IDB_BITMAP2，树形视图和列表视图将这用两幅位图来为表项显示图象。两个位图的尺寸分别为 64×16 和 128×32。每个位图都包含 4 个子图，每个子图中都有一个颜色不同的矩形或圆，请按图 6.7 和表 6.31 绘制。两个位图的子图都是一样的，只不过大小不同，一个是 16×16，一个是 32×32。

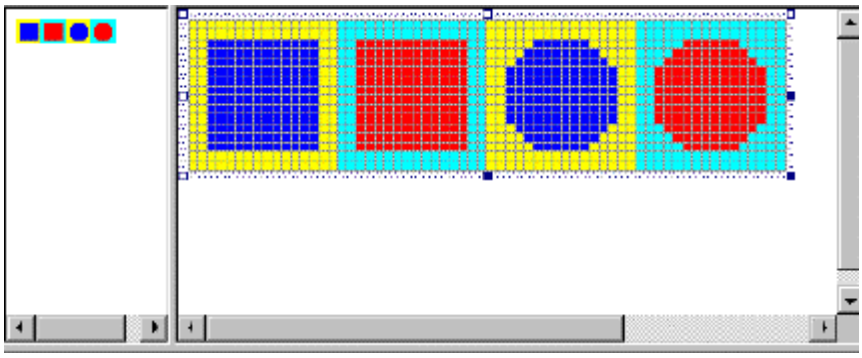


图 6.7 用于树形视图和列表视图的位图

表 6.31

子图的形状	前景色	背景色
矩形	蓝色	黄色
矩形	红色	淡蓝色
圆	蓝色	黄色
圆	红色	淡蓝色

打开 IDD_CTRL32_DIALOG 对话框模板资源，清除 OK 按钮外的所有控件并把 OK 按钮的标题改为 C&lose。将对话框的大小调整为 300×150，然后按图 6.6 和表 6.32 加入控件，并按表 6.32 用 ClassWizard 为 CCtrl32Dlg 类加入成员变量，注意，这些成员变量都是控件对象。

表 6.32

控件类型	控件 ID	需设置的属性	对应的控件对象名
树形视图	缺省	选择 Has buttons、Has lines、Lines at root 和 Edit labels。	m_Tree
列表视图	缺省	选择 Report 格式、Auto arrange、Edit lables	m_List
轨道条	缺省	选择 Tick marks 和 Autoticks。	m_Slider
进度条	缺省	缺省	m_Progress
组合框	缺省	选择 Drop List 类型，不选择 Sort。加入 4 个表项：Icon、Small icon、List 和 Report。	m_ListBox
编辑框	缺省	缺省。要注意其 tab 顺序	

		比旋转按钮小 1.	
旋转按钮	缺省	选择 Auto buddy 和 Set buddy integer	m_Spin

在 CCtrl32Dlg 类的定义处为改类加入下面两个成员，这两个 CImageList 对象用来向树形视图和列表视图提供位图序列. CImageList m_SmallImageList;CImageList m_LargeImageList;

用 ClassWizard 为 CCtrl32Dlg 类加入如表 6.33 所示的消息处理函数.其中 OnHScroll 函数用来处理轨道条的通知消息,OnSelchangeCombo 用来切换列表视图的显示格式.

表 6.33 CCtrl32Dlg 类的控件通知消息处理函数

Object IDS	Messages	Member functions
CCtrl32Dlg	WM_HSCROLL	OnHScroll
IDC_COMBO1	CBN_SELCHANGE	OnSelchangeCombo

最后，请读者按清单 6.2 修改程序.

清单 6.2 CCtrl32Dlg 类的部分源代码

```
// Ctrl32Dlg.cpp : implementation file
. . . . . .
char *szLabel[2]={“Rectangle”,”Circle”};
char *szColumn[3]={“Shape”,”Fore color”,”Back color”};
char *szData[4][3]=
{{“Rectangle”,”Blue”,”Yellow”},
{“Rectangle”,”Red”,”Blue”},
{“Circle”,”Blue”,“Yellow”},
{“Circle”,”Red”,”Blue”}};
DWORD
nStyle[4]={LVS_ICON,LVS_SMALLICON,LVS_LIST,LVS_REPORT};
BOOL CCtrl32Dlg::OnInitDialog()
{
CDialog::OnInitDialog();
. . . . . .
// TODO: Add extra initialization here
//初始化旋转按钮
m_Spin.SetRange(0,200);
m_Spin.SetPos(0);
//初始化轨道条
m_Slider.SetRange(0,20);
m_Slider.SetTicFreq(2);
m_Slider.SetLineSize(2);
m_Slider.SetPageSize(4);
m_Slider.SetPos(0);
//初始化进度条
m_Progress.SetRange(0,20);
m_Progress.SetPos(0);
```

```
//创建位图序列，用于树形视图和列表视图显示图像
m_SmallImageList.Create(IDB_BITMAP1,16,0,FALSE);  //16*16 的位图序列
m_LargeImageList.Create(IDB_BITMAP2,32,0,FALSE);  //32*32 的位图序列
//初始化树形视图
TV_INSERTSTRUCT tvInsert;
HTREEITEM hItem;
int i,j;
char buffer[20];
m_Tree.SetImageList(&m_SmallImageList,TVSIL_NORMAL);
tvInsert.item.mask=TVIF_TEXT|TVIF_IMAGE|TVIF_SELECTEDIMAGE;
for(i=0;i<2;i++)
{
tvInsert.hParent=NULL; //指定该项位于最高层
tvInsert.hInsertAfter=TVI_LAST;
tvInsert.item.pszText=szLabel[i];
tvInsert.item.iImage=i*2; //指定表项显示的图象
tvInsert.item.iSelectedImage=i*2+1; //指定选择状态下应显示的图象
hItem=m_Tree.InsertItem(&tvInsert);
for(j=0;j<3;j++)
{
tvInsert.hParent=hItem; //指定该项为子项
tvInsert.hInsertAfter=TVI_SORT;
sprintf(buffer,"%s%d",szLabel[i],j);
tvInsert.item.pszText=buffer;
m_Tree.InsertItem(&tvInsert);
}
}
//初始化列表视图
LV_COLUMN lvc;
LV_ITEM lvi;
m_List.SetImageList(&m_SmallImageList,LVSIL_SMALL);
m_List.SetImageList(&m_LargeImageList,LVSIL_NORMAL);
m_ComboBox.SelectString(-1,"Report");
lvc.mask=LVCF_FMT|LVCF_WIDTH|LVCF_TEXT|LVCF_SUBITEM;
lvc.fmt=LVCFMT_LEFT;
for(i=0;i<3;i++) //插入各列
{
lvc.pszText=szColumn[i];
if(i==0)
```

```
lvc.cx=m_List.GetStringWidth(szColumn[0])+50;
else
lvc.cx=m_List.GetStringWidth(szColumn[i])+15;
lvc.iSubItem=i;
m_List.InsertColumn(i,&lvc);
}
lvi.mask=LVIF_TEXT|LVIF_IMAGE;
lvi.iSubItem=0;
for(i=0;i<4;i++) //插入表项
{
lvi.pszText=szData[i][0];
lvi.iItem=i;
lvi.iImage=i;
m_List.InsertItem(&lvi);
for(j=1;j<3;j++)
m_List.SetItemText(i,j,szData[i][j]);
}
return TRUE; // return TRUE unless you set the focus to a control
}
void CCtrl32Dlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
// TODO: Add your message handler code here and/or call default
CSliderCtrl* pSlider=(CSliderCtrl*)pScrollBar;
//判断是否是 m_Slider 轨道条发送的消息
if(&m_Slider!=pSlider) return;
m_Progress.SetPos(m_Slider.GetPos());
CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
void CCtrl32Dlg::OnSelchangeCombo()
{
// TODO: Add your control notification handler code here
long lStyle;
lStyle=GetWindowLong(m_List.GetSafeHwnd(),GWL_STYLE);
//清除所有与显示格式有关的风格标志
lStyle&=~(LVS_ICON|LVS_SMALLICON|LVS_LIST|LVS_REPORT);
lStyle|=nStyle[m_ComboBox.GetCurSel()];
//设置新的风格
SetWindowLong(m_List.GetSafeHwnd(),GWL_STYLE,lStyle);
m_List.Invalidate(); //刷新
}
```

对控件的初始化工作在 OnInitDialog 中完成．函数中使用的各种结

构和函数在前面均介绍过，并不难懂．唯一需要说明的是 CImageList 对象的使用．CImageList 对象用来存储多个大小相同的图象，如果程序中要用到大量的尺寸相同的位图或图标，可以用 CImageList 对象把它们组织成图象序列来使用，通过指定序列的索引，可以获得序列中的图象．树形视图和列表视图均使用 CImageList 对象来设置图象序列．在 OnInitDialog 函数中，调用了 CImageList::Create 来创建一个图象序列．该函数的声明为

BOOL Create(UINT nBitmapID, int cx, int nGrow, COLORREF crMask);

参数 nBitmapID 是位图资源的 ID，在该位图中包含了一些尺寸相同的子图，参数 cx 说明了序列中每幅图象的宽度(以像素为单位)，参数 nGrow 说明了当对象包含的图象序列增大时应预留的空位个数，参数 crMask 如果为 TRUE，则说明图象序列中包含屏蔽图象．

在函数中调用了 CTreeCtrl::SetImageList 和 CListCtrl::SetImageList 来为树形视图和列表视图设置图象序列．注意列表视图对象 m_List 设置了两个图象序列，分别用于小图标和大图标显示格式．在插入表项时，只要指定了图象序列的索引，表项就可以显示相应的图象．

OnHScroll 函数负责处理轨道条发出的通知消息，函数根据轨道条的当前位置设置进度条的进度．OnSelchangeCombo 函数响应用户对下拉列表式组合框的选择，函数先调用 CWnd::GetWindowLong 获取列表视图原来的风格，然后调用 CWnd::SetWindowLong 设置新的风格(注意调用 CWnd::GetSafeHwnd 可以获得窗口的句柄)，最后调用 CWnd::Invalidate 刷新列表视图．

Ctrl32 程序只是对常用的 Win 32 控件进行了一些基本的测试。Visual C++提供了一个全面测试 Win 32 控件的 MFC 例子 cmnctrls(在 samples \ mfc \ general \ cmnctrls 目录下)，有兴趣的读者可以研究一下。

6.3 技术总结

在上一章和本章中，读者已经接触和使用了各种控件。这些控件虽然五花八门，但它们却具有一些共同的特点。本节的目的就是讨论这些共同点，以使读者能在概念上更好地理解控件。

6.3.1 所有的控件都是窗口

确切地说，所有的控件都是子窗口。控件窗口都具有 WS_CHILD 风格，它们总是依附于某一个父窗口。所有 MFC 的控件类都是基本窗口类 CWnd 的直接或间接派生类，这意味着可以调用 CWnd 类的某些成员函数来查询和设置控件。常用于控件的 CWnd 成员函数在表 6.34 列出，这些函数对所有的控件均适用。

表 6.34 常用于控件的 CWnd 成员函数

函数名	用途
ShowWindow	调用 ShowWindow(SW_SHOW)显示窗口，调用 ShowWindow(SW_HIDE)则隐藏窗口。
EnableWindow	调用 EnableWindow(TRUE)允许窗口，调用 EnableWindow(FALSE)则禁止窗口。一个禁止的窗口呈灰色显示且不能接受用户输入。
DestroyWindow	删除窗口。
MoveWindow	改变窗口的位置和尺寸。
SetFocus	使窗口具有输入焦点。

例如，如果想把一个编辑框控件隐藏起来，可以用下面这行代码完成。

```
m_MyEdit.ShowWindow(SW_HIDE);
```

6.3.2 控件的创建方法

控件的创建有自动和手工两种常用方法。

控件的自动创建是通过向对话框模板中添加控件实现的。到目前为止，读者所使用的控件都是用这种方法创建的。当调用对话框类的 DoModal 和 Create 显示对话框时，框架会根据对话框模板资源提供的控件信息自动地创建控件。这种方法的优点是方便直观，用户可以在对话框模板编辑器的控件面板中选择控件，可以在对话框模板中调整控件的位置和大小，还可以通过属性对话框设置控件的风格。

手工创建控件是一种比较专业的方法，包括下面两步：

构建一个控件对象。通常的做法是把控件对象嵌入到父窗口(如对话框)对象中，即以成员变量的形式定义一个控件对象。这样，在构建父窗口对象时，控件对象会被自动构建。程序也可以用 new 操作符创建控件对象，但要注意 MFC 的控件对象不具有自动清除的功能，因此需要在关闭父窗口时用 delete 操作符删除控件对象(参见 5.4.2)。

调用控件对象的 Create 成员函数创建控件。一般来说，如果要在对话框中创建控件，那么应该在 OnInitDialog 函数中调用 Create，如果要在非对话框窗口中创建控件，则应该在 OnCreate 函数中调用 Create。

清单 6.3 是一个手工创建控件的实例。

清单 6.3 控件的手工创建

```
#define ID_EXTRA_EDIT 100
class CMyDialog : public CDialog
{
protected:
CEdit m_edit; // Embedded edit object
public:
virtual BOOL OnInitDialog();
};
. . . . . .
BOOL CMyDialog::OnInitDialog()
{
CDialog::OnInitDialog();
CRect rect(85, 110, 180, 210);
m_edit.Create(WS_CHILD | WS_VISIBLE | WS_TABSTOP |
ES_AUTOHSCROLL | WS_BORDER, rect, this, ID_EXTRA_EDIT);
m_edit.SetFocus();
return FALSE;
}
```

不难看出，控件的手工创建是在程序中通过控件对象完成的，与对话框模板无关．在 Create **函数中，需要提供控件的风格，控件的尺寸和位置，控件的** ID **等信息．手工创建实际上是一种动态创建过程，程序可以在任何时侯根据需要来创建，不一定非要在** OnInitDialog **中进行．**

控件并不是对话框所专有的，某些非对话框窗口也可以拥有控件．如果要在象工具条或状态栏这样的非对话框窗口中创建控件，就必需用手工创建方法．自动创建由于要依赖对话框模板，因此只适用于对话框．

6.3.3 访问控件的方法

控件是一种交互的工具，应用程序需要通过某种方法来访问控件以对其进行查询和设置．访问控件有四种方法：

利用对话框的数据交换功能访问控件．这种方法适用于自动创建的控件．先用 ClassWizard **为对话框类加入与控件对应的数据成员变量，然后在适当的时侯调用** UpdateData**，就可以实现对话框和控件的数据交换．这种方法只能交换数据，不能对控件进行全面的查询和设置，而且该方法不是针对某个控件，而是针对所有参与数据交换的控件．另外，对于新型的** Win32 **控件，不能用** ClassWizard **创建数据成员变量．因此，该方法有较大的局限性．**

通过控件对象来访问控件．控件对象对控件进行了封装，它拥有功能齐全的成员函数，用来查询和设置控件的各种属性．通过控件对象来访问控件无疑是最能发挥控件功能的一种方法，但这要求程序必需创建控件对象并使该对象与某一控件相连．对于自动创建的控件，可利用 ClassWizard **方便地创建与控件对应的控件对象．对于手工创建的控件，因为控件本身就是通过控件对象创建的，所以不存在这一问题．**

利用 CWnd **类的一些用于管理控件的成员函数来访问控件．这些函**

数已在表 5.5 列出．只要向这些函数提供控件的 ID，就可以对该控件进行访问．使用这些函数的好处是无需创建控件对象，就可以对控件的某些常用属性进行查询和设置．该方法对自动和手工创建的控件均适用．

用 CWnd::GetDlgItem 访问控件．该函数根据参数说明的控件 ID，返回指定控件的一个 CWnd 型指针，程序可以把该指针强制转换成相应的控件类指针，然后通过该指针来访问控件．该方法对自动和手工创建的控件均适用．在上一章中就曾经使用过这种方法，读者可参见 5.3.6．其实该方法与通过控件对象来访问控件的方法在本质上是一样的，在表 5.5 中亦包括 GetDlgItem 函数，但为了强调其重要性，这里把它单独列为一种方法．

6.3.4 控件及控件对象的删除

当关闭父窗口时，控件会被自动删除，因此在一般情况下不必操心删除问题．如果由于某种需要想手工删除控件，可以调用 CWnd::DestroyWindow 来完成．

对于控件对象的删除，有两种情况．若控件对象是以成员变量的形式创建的，那么该对象将会随着父窗口对象的删除而被删除，因此在程序中无需操心．若控件对象是用 new 操作符在堆中创建的，则必需在关闭父窗口时用 delete 操作符删除对象，这是因为所有 MFC 的控件类都是非自动清除的(参见 5.4.2)．

6.3.5 控件通知消息

传统控件和 Win32 控件采用了不同的通知消息机制，请参见 6.1.1 和 6.2.1。

6.4 在非对话框窗口中使用控件

控件并不是对话框独有的，事实上，很多非对话框窗口都可以使用控件．比较典型的应用是在表单视图、工具条和状态栏中使用控件．

6.4.1 在表单视图中使用控件

MFC 提供了一个名为 CFormView 的特殊视图类，我们称其为表单视图．表单视图是指用控件来输入和输出数据的视图，用户可以方便地在表单视图中使用控件．表单视图具有对话框和滚动视图的特性，它使程序看起来象是一个具有滚动条的对话框．在有些情况下，用表单视图比用普通视图更符合用户的需要，例如，在向数据库输入数据时，显然用表单的形式可以更习惯些．

用 AppWizard 可以方便地创建基于表单视图的应用程序，只要在 MFC AppWizard 对话框的第六步先选择 CView，然后在 Base class 栏中选择 CFormView，AppWizard 就会创建一个基于 CFormView 的应用程序．

读者可以按上述方法建立一个名为 Test 的应用程序．在 Test 工程的资源中，读者会发现一个 ID 为 IDD_TEST_DIALOG 的对话框模板，该对话框模板可供用户放置和安排控件．在程序运行时，框架根据该对话框模板创建 CFormView 对象，并根据模板的信息在表单视图中自动创建控件．与设计对话框类相类似，用户可以用 ClassWizard 为表单视图类加入与控件对应的成员变量，可以调用 UpdateData 在控件和成员变量之间交换数据，但对控件的初始化工作是在 OnInitialUpdate 函数而不是在 OnInitDialog 函数中进行的．

基于表单视图的应用程序与基于对话框的应用程序都是在应用程序中直接使用控件，但二者有很多不同之处．基于对话框的应用程序是用一个对话框来作为程序的主窗口的，因而程序的主窗口的特性与对话框类似，如窗口的大小不能改变，程序没有菜单条、工具条和状态栏等．基于表单视图的应用程序仍然是基于 Doc/View 框架结构的(见七、八、九章)，只是视图被换成了表单视图，也就是说，应用程序的窗口可以改变大小，程序有菜单条、工具条和状态栏，且程序仍然可以 Dov/View 运行机制来处理文档．

表单视图比较简单，这里就不举例说明了．在第十章，读者会看到使用表单视图的例子．

6.4.2 在工具条和状态栏中使用控件

一个专业的程序常常会在工具条和状态栏中加入一些控件以方便用户的使用．例如，在 Developer Studio 的工具条中就有不少组合框，而在状态栏中则常常会显示一个进度条来表明工作的进度．

如果读者想在自己程序的工具条和状态栏中加入控件，则需要掌握一些技巧．在本小节，我们将结合一个具体实例来演示这些技巧．例程的名为 CtrlInBar，其界面如图 6.8 所示．可以看出，该程序在工具条中创建了一个组合框，在状态栏中创建了一个进度条．

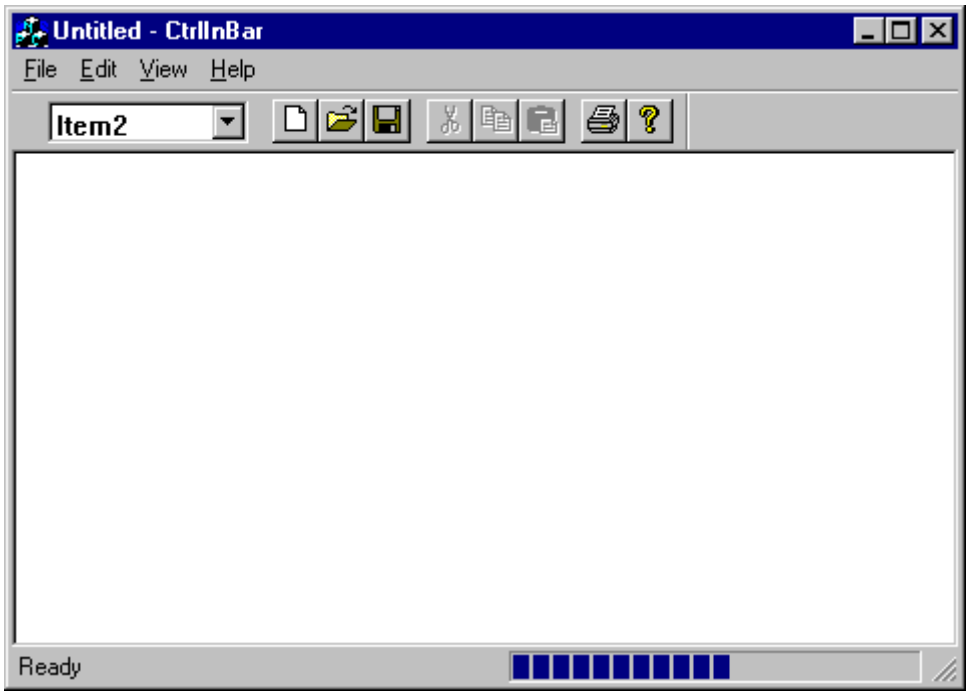


图 6.8 CtrlInBar 程序

现在让我们开始工作．首先，请读者用 AppWizard 建立一个名为 CtrlInBar 的单文挡 MFC 应用程序，然后，请按清单 6.4 修改源代码．注意在程序中编写了一个 CToolBar 类的派生类 CMyToolBar，以及一个 CStatusBar 的派生类 CMyStatusBar，这两个类与 CMainFrame 类在同一模块中．

清单 6.4 在工具条和状态栏中创建控件的有关代码

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////
#define IDC_MYCOMBO 100
class CMyToolBar : public CToolBar
{
public:
CComboBox m_ComboBox;
BOOL CreateComboBox(int nIndex);
};
#define ID_INDICATOR_PROGRESS 100
class CMyStatusBar : public CStatusBar
{
public:
CProgressCtrl m_Progress;
int m_nProgressPane;
BOOL CreateProgressCtrl(int nPane);
afx_msg void OnSize(UINT nType, int cx, int cy);
DECLARE_MESSAGE_MAP()
```

```
};
class CMainFrame : public CFrameWnd
{
. . . . . .
protected: // control bar embedded members
CMyStatusBar m_wndStatusBar;
CMyToolBar m_wndToolBar;
. . . . . .
};
// MainFrm.cpp : implementation of the CMainFrame class
//
. . . . . .
static UINT indicators[] =
{
ID_SEPARATOR, // status line indicator
ID_SEPARATOR
};
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{

if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
return -1;
if (!m_wndToolBar.Create(this) ||
!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
TRACE0("Failed to create toolbar\n");
return -1; // fail to create
}
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT)))
{
TRACE0("Failed to create status bar\n");
return -1; // fail to create
}
// TODO: Remove this if you don't want tool tips or a resizeable toolbar
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
m_wndToolBar.CreateComboBox(0); //在工具条的最左边创建组合框
m_wndStatusBar.CreateProgressCtrl(1); //在第二个窗格创建进度条
m_wndToolBar.m_ComboBox.AddString("Item1");
m_wndToolBar.m_ComboBox.AddString("Item2");
```

```
m_wndToolBar.m_ComboBox.AddString(“Item3”);
m_wndToolBar.m_ComboBox.AddString(“Item4”);
m_wndStatusBar.m_Progress.SetRange(0,200);
m_wndStatusBar.m_Progress.SetPos(100);
. . . . . .
}
////////////////////////////////////////////////////////////////////////
//CMyToolBar
//参数 nIndex 是按钮的索引，函数将在该按钮的左侧创建组合框
BOOL CMyToolBar::CreateComboBox(int nIndex)
{
if(m_ComboBox.GetSafeHwnd()) //防止重复创建
return FALSE;
CToolBarCtrl &ToolBarCtrl=GetToolBarCtrl();
TBBUTTON button;
CRect rect;
button.fsStyle=TBSTYLE_SEP;
ToolBarCtrl.InsertButton(nIndex,&button); //插入空位
ToolBarCtrl.InsertButton(nIndex,&button);
ToolBarCtrl.InsertButton(nIndex,&button);
//设置空位的宽度(处于中间的空位用来容纳组合框)
SetButtonInfo(nIndex+1,IDC_MYCOMBO,TBBS_SEPARATOR,100);
SetButtonInfo(nIndex, ID_SEPARATOR, TBBS_SEPARATOR, 12);
SetButtonInfo(nIndex+2, ID_SEPARATOR, TBBS_SEPARATOR, 12);
GetItemRect(nIndex+1, &rect); //获取中间空位的坐标
rect.top = 3;
rect.bottom = rect.top + 100;
if
(!m_ComboBox.Create(CBS_DROPDOWNLIST|WS_VISIBLE|WS_TABS
TOP,
rect, this, IDC_MYCOMBO))
return FALSE;
m_ComboBox.SetItemHeight(-1,15); //设置编辑框组件的高度
return TRUE;
}
////////////////////////////////////////////////////////////////////////
//CMyStatusBar
BEGIN_MESSAGE_MAP(CMyStatusBar, CStatusBar)
ON_WM_SIZE()
END_MESSAGE_MAP()
//参数 nPane 是窗格的索引，函数将在该窗格内创建进度条控件
BOOL CMyStatusBar::CreateProgressCtrl(int nPane)
```

```
{
if(m_Progress.GetSafeHwnd()) //防止重复创建
return FALSE;
//设置该窗格的宽度为 200
SetPaneInfo(nPane,GetItemID(nPane),SBPS_NORMAL,200);
CRect rect(0,0,1,1);
if(!m_Progress.Create(WS_CHILD|WS_VISIBLE,rect,this,
ID_INDICATOR_PROGRESS))
return FALSE;
m_nProgressPane=nPane;
return TRUE;
}
void CMyStatusBar::OnSize(UINT nType, int cx, int cy)
{
CStatusBar::OnSize(nType, cx, cy);
if(m_Progress.GetSafeHwnd()==NULL) return;
CRect rect;
GetItemRect(m_nProgressPane,&rect);
m_Progress.MoveWindow(rect); //调整控件的位置和尺寸
}
```

CMyToolBar 类可以在工具条中指定按钮的左边放置一个下拉列表式组合框，并在组合框的两端留出空位．该类的 CreateComboBox 成员负责创建组合框，参数 nIndex 是工具条按钮的索引，需注意的是工具条的一个空位也要占有一个索引．在 CreateComboBox 中，主要调用了下列函数：

调用 CToolBar::GetToolBarCtrl 返回一个 CToolBarCtrl 对象．从 4.0 版开始，CToolBar 类是在新控件类 CToolBarCtrl 类的基础上实现的，后者具有更强大的功能．例如 CToolBarCtrl 提供了 CToolBar 没有的 InsertButton 成员函数．

调用 CToolBarCtrl::InsertButton 在 nIndex 索引处插入三个空位．

调用 CToolBar::SetButtonInfo 设置空位的宽度，其中中间的空位有 100 像素宽，用来容纳组合框．

调用 CToolBar::GetItemRect 获得中间空位的坐标．

调用 CComboBox::Create 函数创建组合框．注意 rect 对象说明的是包括列表框在内的组合框的尺寸．

调用 CComboBox::SetItemHeight 设置编辑框组件的高度．

CMyStatusBar 类可以在指定的状态栏窗格中放置一个进度条．该类的 CreateProgressCtrl 成员负责创建进度条，参数 nPane 是窗格的索引．在该函数中主要调用了下列函数：

调用 CStatusBar::SetPaneInfo 设置窗格的宽度为 200．在调用该函数时，先调用 CStatusBar::GetItemID 返回窗格的 ID．

调用 CProgressCtrl::Create 创建控件．

大家可能会奇怪，CProgressCtrl::Create 创建的控件只有 1×1 大小．这是由于在调用该函数创建控件时，状态栏的大小往往并未确定．这时如果调用 CStatusBar::GetItemRect，只能得到 0 坐标，而不能得到正确的窗格坐标，所以程序只好先创建一个 1×1 的控件．

工具条中按钮和控件的尺寸及其相对于工具条的位置不会随外界因素发生变化，而状态栏则不同，当用户改变了框架窗口的宽度时，状态栏的宽度也会随之改变，并且它会重新调整各窗格的大小和位置，此时如果不及时调整进度条的坐标，那么进度条与所在窗格之间将发生错位．调整进度条的大小和位置的工作由 CMyStatusBar::OnSize 函数完成．当窗口的尺寸发生改变后，窗口会收到 WM_SIZE 消息，OnSize 是 WM_SIZE 消息的处理函数．在 CMyStatusBar::OnSize 函数中，先调用 CStatusBar::GetItemRect 获得进度条所在窗格的坐标，然后调用 CWnd::MoveWindow 来调整进度条控件的坐标．

在窗口形成时，也会收到 WM_SIZE 消息，这时 OnSize 函数可以及时调整进度条的大小和位置．

6.5 设计新的控件类

虽然 Windows 提供了大量的控件，但不一定总能满足用户的需要．有时，用户需要一些有特殊功能的控件．例如，有时希望编辑框控件只能接受数字输入，当用户输入非数字字符时，编辑框控件会发出声响来提醒用户．在这种情况下，标准的 CEdit 类就无能为力了．

当控件无法满足需要时，用户可以从原来的控件类派生一个新类．通过合理地设计派生类，可以修改控件的行为和属性以达到用户的要求．利用 ClassWizard 的强大功能，读者可以方便地创建和设计控件类的派生类．

6.5.1 创建标准控件类的派生类

这个任务可以用 ClassWizard 完成，其具体步骤如下：

按 Ctrl+W 进入 ClassWizard．

单击 Add Class 按钮并选择 New...菜单项，则打开 Create New Class 对话框．

在 Create New Class 对话框的 Name 栏中输入派生类的名字，在 Base class 栏中选择一个标准的控件类做为基类，然后按 Create 按钮即可．

6.5.2 利用 MFC 的控件通知消息反射机制完善派生类的功能

创建好派生类后，接下来的任务就是修改新类的代码以完善其功能．例如，为新类添加必要的成员变量，提供新的成员函数以及消息处理函数等等．其中为控件添加消息处理函数是最重要的，因为这直接关系着控件新功能的实现．

与控件有关的消息包括控件本身接收的消息和发给父窗口的通知消息两种．利用 ClassWizard 可以方便地为派生类创建这两类消息的处理函数．读者也许会感到奇怪，控件通知消息不是由父窗口处理的吗，难道控件本身也有机会处理通知消息”

答案是肯定的．从 4.0 版开始，MFC 提供了一种消息反射机制(Message Reflection)，可以把控件通知消息反射回控件．具体地讲，当父窗口收到控件通知消息时，如果父窗口有该消息的处理函数，那么就由父窗口处理该消息，如果父窗口不处理该消息，则框架会把该消息反射给控件，这样控件就有机会处理该消息了．由此可见，新的消息反射机制并不破坏原来的通知消息处理机制．

消息反射机制为控件提供了处理通知消息的机会，这在有些情况下是很有用的．例如，如果派生类想改变控件的背景色，就需要处理 WM_CTLCOLOR 通知消息．大多数控件在需要重绘时，会向父窗口发送 WM_CTLCOLOR 消息，父窗口在处理该消息时会返回一个刷子用来画控件的背景．如果按传统的方法，由父窗口来处理这个消息，则加重了控件对象对父窗口的依赖程度，每当使用这样一个新控件时，都要在父窗口中提供控件的 WM_CTLCOLOR 消息处理函数，这显然违背了面向对象的原则．若由控件自己处理 WM_CTLCOLOR 消息，则使得控件对象具有更大的独立性，而父窗口也可以省去一些不必要的工作．

读者可以在自己的程序中用 ClassWizard 创建一个 CEdit 类的派生类试试．在派生类的消息列表中，在有些消息前面有一个＂＝＂符号，这

表明这些消息是可以反射的通知消息．读者可以按照通常的方法创建反射消息的处理函数．

6.5.3 利用 SubclassDlgItem 函数动态连接控件和控件对象

要在程序中创建新设计的控件，显然不能用自动创建的办法，因为对话框模板对新控件的特性一无所知．程序可以用手工方法创建控件，在调用派生类的 Create 函数时，派生类会调用基类的 Create 函数创建控件．用 Create 函数创建控件是一件比较麻烦的工作，程序需要为函数指定一大堆的控件风格以及控件的坐标和 ID．特别是控件的坐标，没有经验的程序员很难确切地安排控件的位置和大小，往往需要反复调整．利用 MFC 的 CWnd::SubclassDlgItem 提供的动态连接功能，可以避免 Create 函数的许多麻烦，该函数大大简化了在对话框中创建派生控件的过程．

大家知道，在用手工方法创建控件时，先要构建一个控件对象，然后再用 Create 函数在屏幕上创建控件窗口，也就是说，控件的创建工作是由控件对象完成的．动态连接的思路则不同，SubclassDlgItem 可以把对话框中已有的控件与某个窗口对象动态连接起来，该窗口对象将接管控件的消息处理，从而使控件具有新的特性．SubclassDlgItem 函数的声明为

```
BOOL SubclassDlgItem( UINT nID, CWnd* pParent );
```

参数 nID 是控件的 ID，pParent 是指向父窗口的指针．若连接成功则函数返回 TRUE，否则返回 FALSE．

综上所述，要在程序中使用派生控件，应该按下面两步进行：

在对话框模板中放置好基类控件．

在对话框类中嵌入派生控件类的对象．

在 OnInitDialog 中调用 SubclassDlgItem 将派生类的控件对象与对话框中的基类控件相连接，则这个基类控件变成了派生控件．

例如，如果要在对话框中使用新设计的编辑框控件，应先在对话框模板中的合适位置放置一个普通的编辑框，然后，在 OnInitDialog 函数中按下面的方式调用 SubclassDlgItem 即可：

```
BOOL CMyDialog::OnInitDialog()
{
CDialog::OnInitDialog();
m_MyEdit.SubclassDlgItem(IDC_MYEDIT, this);
return TRUE;
}
```

小结

这一讲对控件的创建和使用进行了较全面的介绍，其要点为：

Windows 的控件分为传统控件和新型 Win32 控件两类。传统控件包括静态控件、按钮、编辑框、滚动条、列表框和组合框。Win32 控件包括列表视图、树形视图、进度条、旋转按钮、轨道条、热键和标签。

传统控件的通知消息一般是通过 WM_COMMAND 消息发给父窗口的，Win32 控件的通知消息则是通过 WM_NOTIFY 消息发送的，该消息可以附带大量信息。

由于所有的控件都是子窗口，所有的控件类都是 CWnd 类的派生类。因此可以用象 ShowWindow、EnableWindow 和 MoveWindow 这样的 CWnd 成员函数来控制控件。

控件的创建有自动和手工两种常用方法． 控件的自动创建是通过向对话框模板中添加控件实现的。手工创建则需要构建一个控件对象并调用相应的 Create 函数。

访问控件的方法包括：1、通过对话框的数据交换功能来查询和设置控件；2、通过控件对象来访问控件；3、利用 CWnd 类的一些用于管理控件的成员函数来访问控件．

控件对象一般是以成员变量的形式嵌入到父窗口对象中的。如果控件是创建在堆中的，则在父窗口被关闭时显示调用 delete 删除控件对象。

并不是只有对话框才能使用控件，控件可以出现在表单视图、工具条和状态栏等窗口中。

可以用 ClassWizard 生成标准控件类的派生类，以便设计出用户定制的控件。利用 MFC 的控件通知消息反射机制可以完善派生类的功能，利用 CWnd::SubclassDlgItem 提供的动态连接功能，可以把对话框模板中的控件与定制的控件对象连接起来。

第七课　文档/视结构

这一讲介绍文档/视结构的基本概念，并结合一个简单的文本编辑器的例子说明文档视结构的内部运行机制和使用。

文档/视图概念

文档视结构程序实例

让文档视结构程序支持卷滚

定制串行化

不使用串行化的文档视结构程序

小 结

7.1 文档/视图概念

7.1.1 概念

在文档视结构里，文档是一个应用程序数据基本元素的集合，它构成应用程序所使用的数据单元；另外它还提供了管理和维护数据的手段。

文档是一种数据源，数据源有很多种，最常见的是磁盘文件，但它不必是一个磁盘文件，文档的数据源也可以来自串行口、网络或摄像机输入信号等。在第十二章“多线程和串行通信编程”中，我们展示了如何使用串行口作为数据输入的文档/视结构程序。文档对象负责来自所有数据源的数据的管理。

视图是数据的用户窗口，为用户提供了文档的可视的数据显示，它把文档的部分或全部内容在窗口中显示出来。视图还给用户提供了一个与文档中的数据交互的界面，它把用户的输入转化为对文档中数据的操作。每个文档都会有一个或多个视图显示，一个文档可以有多个不同的视图。比如，在 Excel 电子表格中，我们可以将数据以表格方式显示，也可以将数据以图表方式显示。一个视图既可以输出到窗口中，也可以输出到打印机上。

图 7-1 说明了文档及其视图之间的关系。

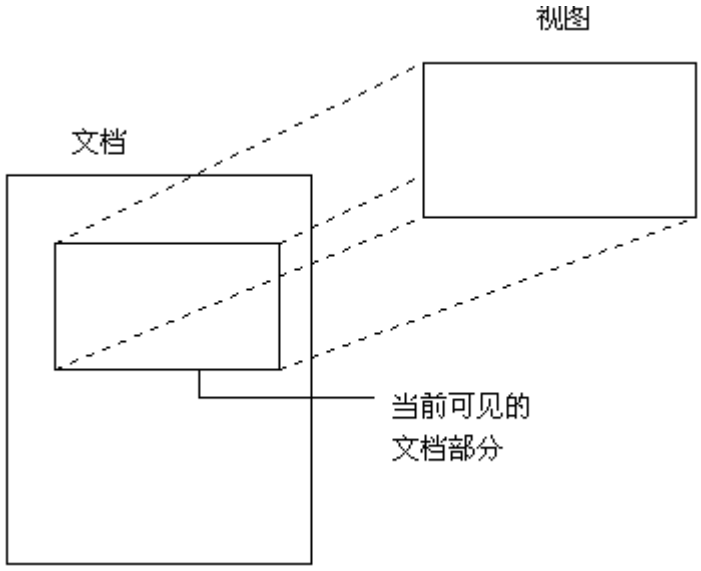


图 7-1 文档和视图

MFC 的文档/视结构机制把数据同它的显示以及用户对数据的操作分离开来。所有对数据的修改由文档对象来完成。视图调用这个对象的方法来访问和更新数据。

7.1.2 两类文档视结构程序

有两种类型的文档视结构程序：单文档界面(SDI)应用程序和多文档界面(MDI)应用程序。

在单文档界面程序中，用户在同一时刻只能操作一个文档。象 Windows95 下的 NotePad 记事本程序(如图 7-2 所示)就是这样的例子。在这些应用程序中，打开文档时会自动关闭当前打开的活动文档，若文档

修改后尚未保存，会提示是否保存所做的修改。因为一次只开一个窗口，因此不象 WORD 那样需要一个窗口菜单。单文档应用程序一般都提供一个 File 菜单，在该菜单下有一组命令，用于新建文档(New)、打开已有文档(Open)、保存或换名存盘文档等。这类程序相对比较简单，常见的应用程序有终端仿真程序和一些工具程序。

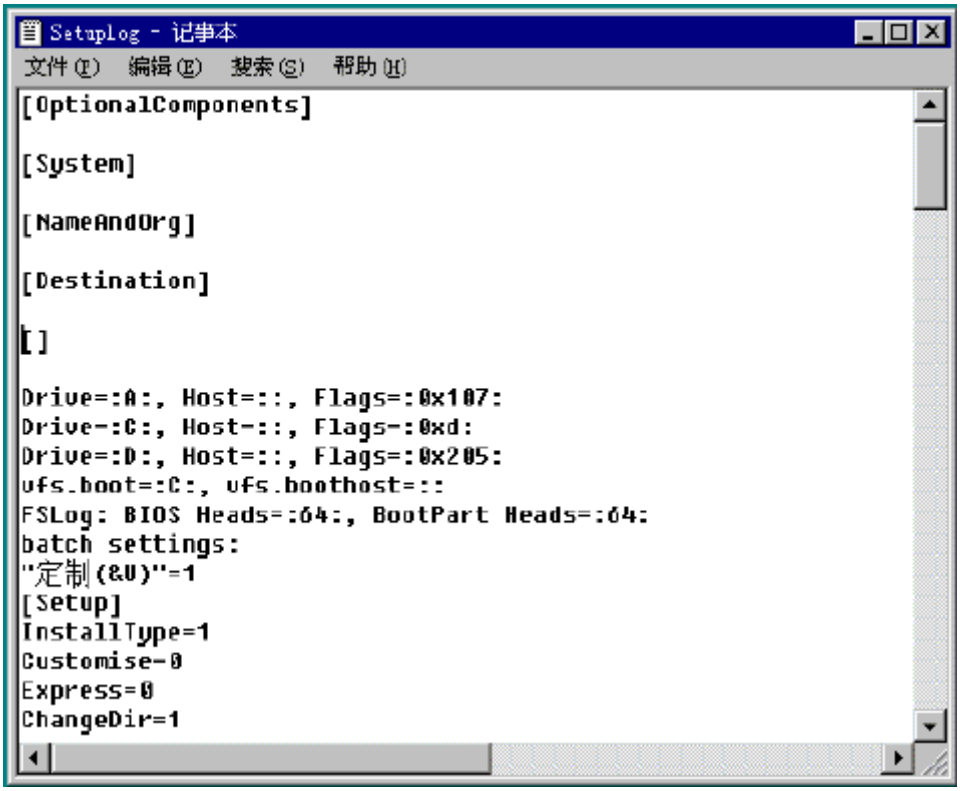


图 7-2 单文档程序(记事本)

一个多文档界面应用程序也能操作文档，但它允许同时操作多个文档。如图 7-2，Microsoft Word 就是这样的例子。你可以打开多个文件(同时也就为每个文件打开一个窗口)，可以通过切换活动窗口激活相应的文档进行编辑。多文档应用程序也提供一个 File 菜单，用于新建、打开、保存文档。与单文档应用程序不同的是，它往往还提供提供一个 Close(关闭)菜单项，用于关闭当前打开的文档。多文档应用程序还提供一个窗口菜单，管理所有打开的子窗口，包括对子窗口的新建、关闭、层叠、平铺等。关闭一个窗口时，窗口内的文档也被自动关闭。在这一章里，我们只讨论单文档界面应用程序的编制，有关多文档技术在下一章里再做讨论。

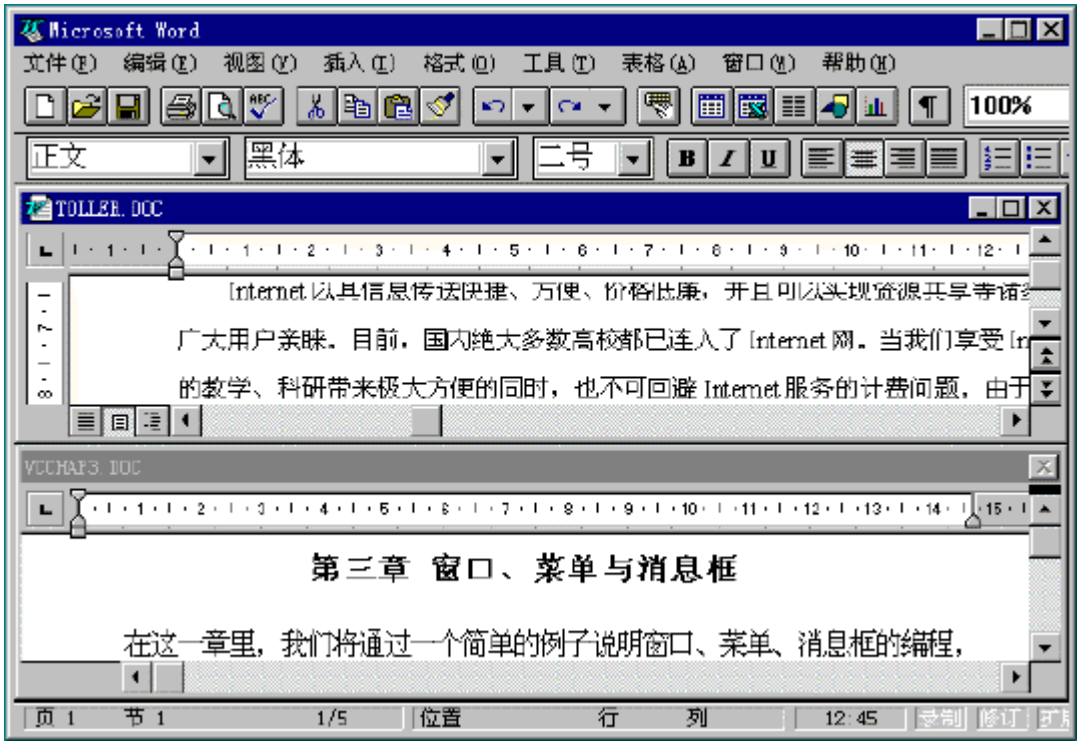


图 7-3 多文档程序(Microsoft Word)

7.1.3 **使用文档/视结构的意义**

文档视结构的提出对于广大程序员来说是一个福音，它大大简化了多数应用程序的设计开发过程。文档视结构带来的好处主要有：

a.**首先是将数据操作和数据显示、用户界面分离开。这是一种“分而治之”的思想，这种思想使得模块划分更加合理、模块独立性更强，同时也简化了数据操作和数据显示、用户界面工作。文档只负责数据管理，不涉及用户界面；视图只负责数据输出与用户界面的交互，可以不考虑应用程序的数据是如何组织的，甚至当文档中的数据结构发生变化时也不必改动视图的代码。**

b.MFC **在文档/视结构上提供了许多标准操作界面，包括新建文件、打开文件、保存文件、打印等，减轻了用户的工作量。用户不必再书写这些重复的代码，从而可以把更多的精力放到完成应用程序特定功能的代码上：主要是从数据源中读取数据和显示。**

c.**支持打印预览和电子邮件发送功能。用户无需编写代码或只需要编写很少的代码，就可以为应用程序提供打印预览功能。同样的功能如果需要自己写的话，需要数千行的代码。另外，**MFC **支持在文档视结构中以电子邮件形式直接发送当前文档的功能，当然本地要有支持** MAPI(**微软电子邮件接口**)**的应用程序，如** Microsoft Exchange**。可以这样理解：**MFC **已经把微软开发人员的智慧和技术溶入到了你自己的应用程序中。**

由于文档视结构功能如此强大，因此一般我们都首先使用 AppWizard **生成基于文档/视结构的单文档或多文档框架程序，然后在其中添加自己的特殊代码，完成应用程序的特定功能。但是，并非所有基于窗口的应用程序都要使用文档/视结构。象** Visual C++**随带的例子** Hello、MDI **都没有使用文档/视结构。有两种情况不宜采用文档、视结构：**

(1)**不是面向数据的应用或数据量很少的应用程序，不宜采用文档/视结构。如一些工具程序包括磁盘扫描程序、时钟程序，还有一些过程控制程序等。**

(2)**不使用标准的窗口用户界面的程序，象一些游戏等。**

7.2 文档视结构程序实例

下面，我们以一个简单的文本编辑器为例，说明文档/视结构的原理及应用。由于我们重在讨论文档/视结构而不是编辑器的实现，因此这个编辑器设计的非常简单：用户只能逐行输入字符，以回车结束一行并换行，不支持字符的删除和插入，也没有光标指示当前编辑位置。另外，用户可以选择编辑器显示文本时所使用的字体。

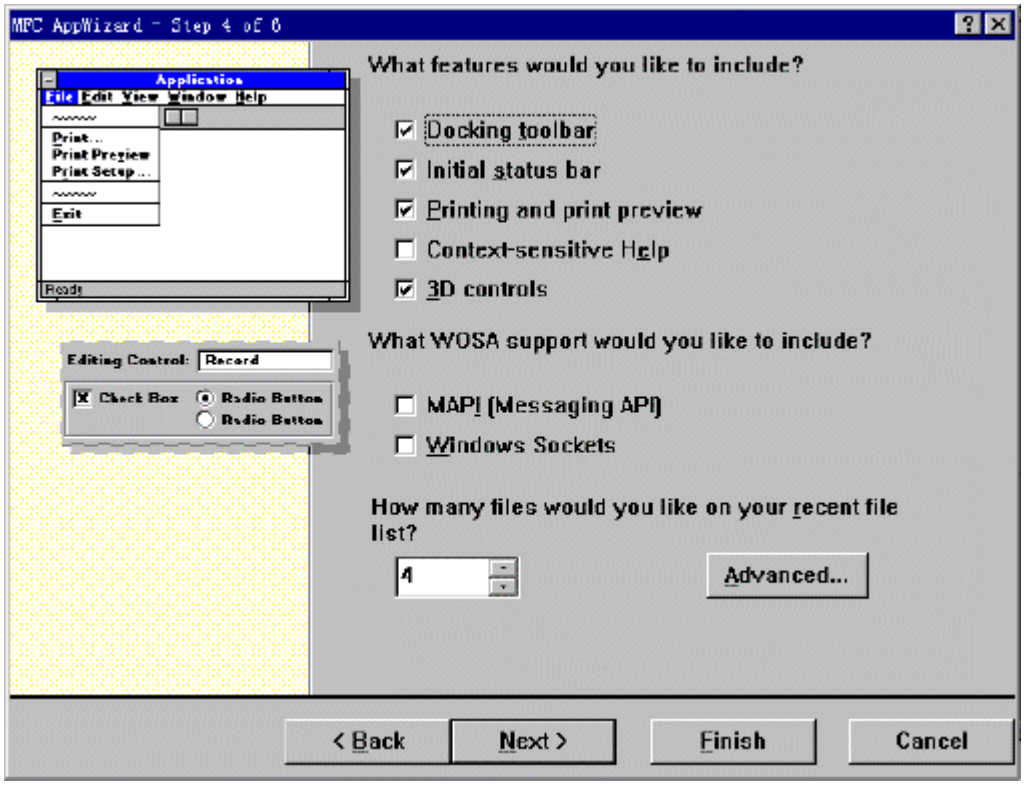


图 7-4

首先，使用 AppWizard 生成编辑器程序的框架：在 New 对话框的 Project Name 编辑框中输入项目名为 Editor。在 AppWizard 的第一步选择 Single document ，这将创建一个 SDI 应用程序。AppWizard 第二和第三步选项使用缺省值。在 AppWizard Step 4 of 6 对话框中，如图 7-4 所示，细心的读者或许会注意到在这一页里，有一个 Advanced 按钮，以前没有提到过。现在掀击该按钮，弹出 Advanced Option 对话框，如图 7-5 所示。Advanced Option 对话框是用来设置文档视结构和主框架窗口的一些属性的。

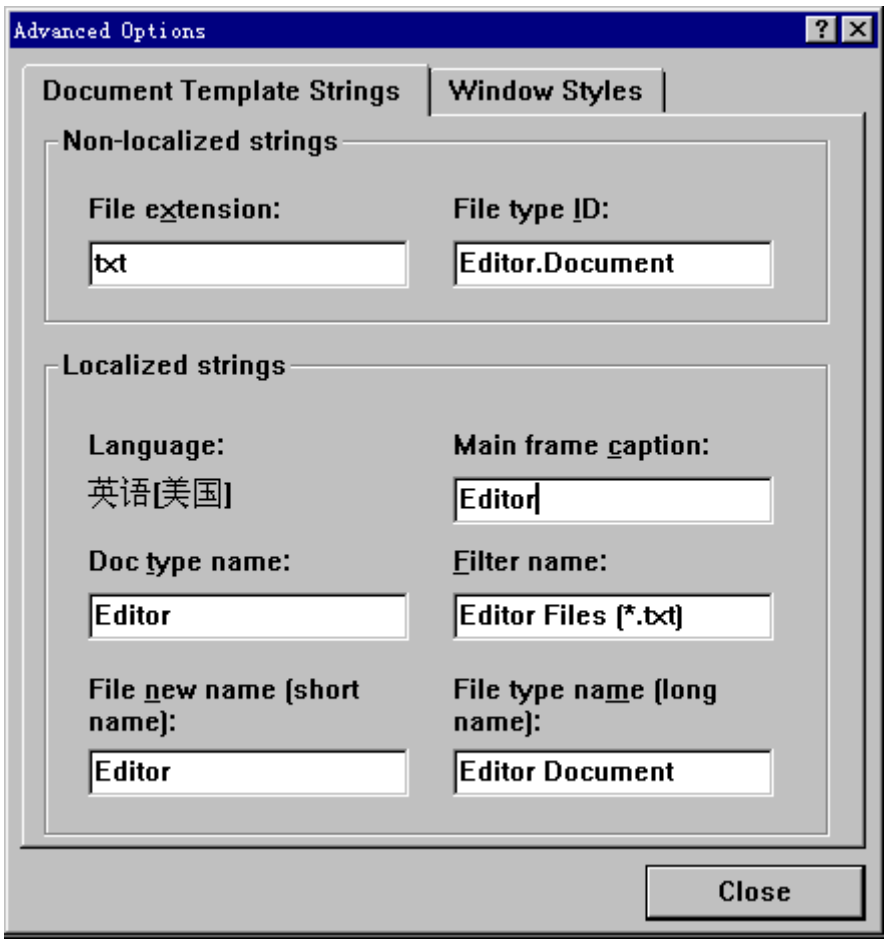


图 7-5

该对话框提供两个标签页，一页是 Document Template String(文档模板字符串，有关文档模板字符串，我们还将在后面作详细介绍)，用于设置文档视结构的一些属性。它包括以下几个编辑框：

File Extension：指定应用程序创建的文档所用的文件名后缀。输入后缀名 txt(不需要．号。)，表明 Editor 使用文本文件的后缀名 TXT。

File ID：用于在 Windows95 的注册数据库中标识应用程序的文档类型。

MainFrame Caption：主框架窗口使用得标题，缺省情况下与项目名相一致，你当然可以将它改为任何你喜欢的名字，如 Editor for Windows 等。

Doc Type name：文档类型名，指定与一个从 CDocument 派生的文档类相关的文档类型名。

Filter Name：用作“打开文件”、“保存文件”对话框中的过滤器。当你在 File Extension 中输入后缀名是，Visual Studio 会自动给你生成一个过滤器：Editor Files(*.txt)。这样，当你在 Open File 对话框中选择 Editor Files(*.txt)时，只有以 txt 为后缀名的文件名显示在文件名列表中。

File new name(short name)：用于指定在 new 对话框中使用的文档名。当应用程序支持多种文档类型时，选择 File-New 菜单项会弹出一个对话框，列出应用程序所支持的所有文档类型，供用户选择。选择一种文档

类型后，自动创建相应类型的文档。这里我们只支持编辑器这一种文档类型，故使用缺省值。

File Type name(long name)：用于指定当应用程序作为 OLE Automation 服务器时使用的文档类型名。使用缺省值。

另一页是 Window Styles，用于设置主框架窗口的一些属性，包括框架窗口是否使用最大化按钮、最小化按钮，窗口启动时是否最大化或最小化等。这里我们使用缺省值，不需要作任何修改。

按 OK 按钮，关闭 Advanced Option 对话框。

AppWizard 后面的几页对话框都使用缺省值。创建完 Editor 框架程序后，Visual Studio 自动打开 Editor 工程。现在要修改 Editor 框架程序，往程序中添加代码，实现编辑器功能。

7.2.1 文档/视结构中的主要类

在 Editor 框架程序中，与文档视结构相关的类有 CEditorApp、CMainFrame、CEditorView 和 CEditorDoc，它们分别是应用程序类 CWinApp、框架窗口类 CFrameWnd、视图类 CView 和文档类 CDocument 的派生类。

应用程序对象

其中，应用程序类负责一个且唯一的一个应用程序对象的创建、初始化、运行和退出清理过程。如果在 AppWizard 生成框架时指定使用单文档或多文档，AppWizard 会自动将 File 菜单下的 New、Open 和 Printer Setup(打印机设置)自动映射到 CWinApp 的 OnFileNew、OnFileOpen、OnFilePrintSetup 成员函数，让 CWinApp 来处理以上这些消息。如清单 7.1，浏览 CEditorApp 类的定义文件有关消息映射的代码。

清单 7.1 CEditorApp 的消息映射

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

这表明，框架已经给我们生成了有关新建文档、打开文档以及打印设置的标准代码，我们不必再去做这些重复的工作了。那么，当我们新建或打开一个文档时，应用程序怎么知道要创建什么样的文档以及创建什么样的视图、框架窗口来显示该文档的呢”在文档/视结构中，应用程序

通过为应用程序所支持的每一种文档创建一个文档模板，来创建和管理所有的文档类型并为它们生成相应的视图和框架窗口。

文档模板

文档模板负责创建文档、视图和框架窗口。一个应用程序对象可以管理一个或多个文档模板，每个文档模板用于创建和管理一个或多个同种类型的文档(这取决于应用程序是单文档 SDI 程序还是多文档 MDI 程序)。那些支持多种文档类型(如电子表格和文本)的应用程序，有多种文档模板对象。应用程序中的每一种文档，都必需有一种文档模板和它相对应。比如，如果应用程序既支持绘图又支持文本编辑，就需要一种一种绘图文档模板和文本编辑模板。在下一章里，我们举了一个这样的例子，来说明多种文档模板的实现技术。

MFC 提供了一个文档模板类 CDocTemplate 支持文档模板。文档模板类是一个抽象的基类，它定义了文档模板的基本处理函数接口。由于它是一个抽象基类，因此不能直接用它来定义对象而必需用它的派生类。对一个单文档界面程序，使用 CSingleDocTemplate(单文档模板类)，而对于一个多文档界面程序，使用 CMultipleDocTemplate。

文档模板定义了文档、视图和框架窗口这三个类的关系。通过文档模板，我们可以知道在创建或打开一个文档时，需要用什么样的视图、框架窗口来显示它。这是因为文档模板保存了文档和对应的视图和框架窗口的 CRuntimeClass 对象的指针。此外，文档模板还保存了所支持的全部文档类的信息，包括这些文档的文件扩展名信息、文档在框架窗口中的名字、代表文档的图标等信息。

提示 :每个从 CObject 派生的类都与一个 CRuntimeClass 结构相关联。通过这个结构，你可以在程序运行时刻获得关于一个对象和它的基类的信息。在函数参数需要作附加类型检查时，这种运行时刻判别对象类型的能力是非常重要的。C++本身并不支持运行时刻类信息。CRuntimeClass 结构包含一个以\0 结尾的字符串类名、整型的该类对象大小、基类的运行时刻信息等。/

一般在应用程序的 InitInstance 成员函数实现中创建一个或多个文档模板，如清单 7.2。

清单 7.2 CEditorApp 的 InitInstance 成员函数定义

```
BOOL CEditorApp::InitInstance()
{
//标准的初始化代码
//.....
// Register the application's documenttemplates. Document templates
// serve as the connection betweendocuments, frame windows and views.
CSingleDocTemplate* pDocTemplate；
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CEditorDoc),
RUNTIME_CLASS(CMainFrame), // main SDIframe window
```

RUNTIME_CLASS(CEditorView))；
AddDocTemplate(pDocTemplate)；
//其他的初始化代码和主框架窗口显示过程
//.....
// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);
// Parse command line for standardshell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on thecommand line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
// The one and only window has beeninitialized, so show and update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
// Enable drag/drop open
m_pMainWnd->DragAcceptFiles();
}

在 InitInstance 中，首先声明一个 CSingleDocTemplate*类型的单文档模板对象指针(因为这里的文本编辑器使用单文档界面)。然后创建该类型的模板对象。如果要使用多文档界面，只需要将这里的 CSingleDocTemplate 改为 CMultiDocTemplate，当然 CMainFrame 也要改为从 CFrameWnd 改为 CMDIChildWnd 或其派生类。

在 CSingleDocTemplate 构造函数中，还包含一个 IDR_MAINFRAME 参数。它指向一个字符串资源，这个字符串给出了文档所使用及显示时所要求的几个选项，包括文档名字、文档的文件扩展名、在框架窗口上显示的名字等等，我们称之为文档模板字符串。有关文档模板字符串还将在下一章使用多个文档模板这一节作详细阐述，因此这里就不展开讲了。

然后 InitInstance 调用 AddDocTemplate 将创建好的文档模板加入到应用程序可用的文档模板链表中去。这样，如果用户选择了 File-New 或 File-Open 菜单要求创建或打开一个文档时，应用程序类的 OnNewDocument 成员函数和 OnOpenDocument()成员函数就可以从文档模板链表中检索出文档模板提示用户选择适当的文档类型并创建文档及相关的视图、框架窗口。

文档

Editor 的文档类 CEditorDoc 从 CDocument 派生下来，它规定了应用程序所用的数据。如果需要在应用程序中提供 OLE 功能，则需要从 COleDocument 或其派生类派生出自己的文档类。

视图

Editor 的视图类从 CView 派生，它是数据的用户窗口。视图规定了

用户查看文档数据以及同数据交互的方式。有时一个文档可能需要多个视图。

如果文档需要卷滚，需要从 CScrollView 派生出视图类。如果希望视图按一个对话框模板资源来布置用户界面，可以从 CFormView 派生。由于 CFormView 经常同数据库打交道，因此我们把它放在第十章“ 数据库技术 ’中结合数据库技术讲解。感兴趣的读者可以先看看 Visual C++ MFC 例子 CHKBOOK(在 SAMPLES\MFC\GENERAL\CHKBOOK 目录下)。

框架窗口

视图在文档框架窗口中显示，它是框架窗口的子窗口。框架窗口作用有二：一是为视图提供可视的边框，还包括标题条、一些标准的窗口组件(最大、最小化按钮、关闭按钮)，象一个容器一样把视图装起来。二是响应标准的窗口消息，包括最大化、最小化、调整尺寸等。当框架窗口关闭时，在其中的视图也被自动删除。视图和框架窗口关系如图 7-6 所示：

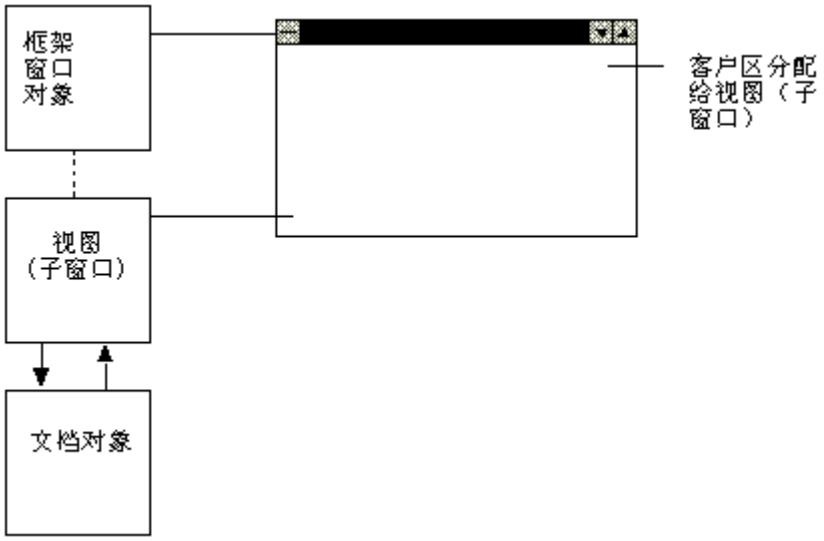


图 7-6 视图和框架窗口的关系

对于 SDI 程序，文档框架窗口也就是应用程序的主框架窗口。在 MDI 应用程序中，文档框架窗口是显示在主框架窗口中的子窗口(通常是 CMDIChildWnd 或其派生类)。

可以从主框架窗口类派生出新类来包含你的视图，并指定框架的风格和其他特征。如果是 SDI 程序，则从 CFrameWnd 派生出文档框架窗口：

```
class CMainFrame:public CFrameWnd
{
...
}；
```

如果是 MDI 窗口，则需要从 CMDIFrameWnd 派生出主框架窗口，同时在从 CMDIChildWnd 或其派生类派生出一个新类，来定制特定文档窗口的属性和功能。

在应用程序运行过程中，以上几种类型的对象相互协作，来处理命

令和消息。一个且唯一的一个应用程序对象管理一个或多个文档模板，每个文档模板创建和管理一个(SDI)或多个文档(MDI)。用户通过包含在框架窗口中的视图来浏览和操作文档中的数据。在 SDI 应用程序中，以上对象关系如图 7-7 所示。

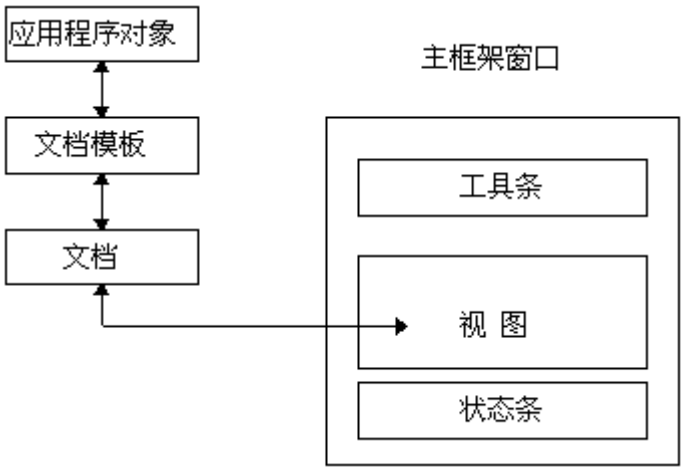


图 7-7 在 SDI 程序中各对象的关系

7.2.2 设计文本编辑器的文档类

弄清这些对象的关系以后，就可以着手往框架里填写代码，实现我们的文本编辑器程序了。从以上分析可以看出，文档视结构程序的主要工作在于文档和视图的设计。

首先设计文档。程序=数据+算法，在 MFC 文档/视结构中，最关键的就是文档的设计。怎样保存用户输入的文本行"方法之一是保存一组指针，每个指针指向一个文本行。如果使用 C 语言来写这个程序的话，需要分配内存来存放这些指针，还要自己编写文本行的动态分配、增加、删除等例程。但是 MFC 简化这些工作，它提供了集合类(collection classes)。

集合类是用来容纳和处理一组对象或标准数据类型变量的 C++类。每个集合类对象可以看作一个单独的对象。类成员函数可作用于集合的所有元素。MFC 提供两种类型的集合类：

基于模板的集合类

非基于模板的集合类

这两种集合类 对用户来说非常相似。基于模板的集合所包含的元素是用户自定义的数据结构或者说是抽象的数据结构，它以数组、链表和映射表三种方式组织用户自定义的数据结构。使用基于模板的集合类需要用户作一些类型转换工作。非基于 模板的集合类提供的是一组现成的、用于某种预定义的数据类型(如 CObject、WORD、BYTE、DWORD、字符串等)的集合。在设计程序时，如果所用的数据类型是预定义的，如下面的编辑要用到的字符串，则使用非基于模板的集合类；如果所用得数据类型是用户自定义的数据结构类型，那就要用到基于模板的集合类。

根据对象在集合中的组织合存储方式，集合类又可分为三种类型：链表、数组、映射(或字典)。应当根据特定的编程问题，选择适当的类型。

链表：链表类用双向链表实现有序的、非索引的元素链表。链表有

一个头或尾。很容易从头或尾增加或删除元素、遍历所有元素，在中间插入或删除元素。链表在需要增加、删除元素的场合效率很高。非基于模板的链表有三种：CObList、CPtrList、CStringList，分别用于管理对象指针、无类型指针和字符串。可以使用链表创建堆栈和队列。

要访问链表的成员，可以使用 GetNext 和 GetHeadPosition()。

要删除链表的成员，可以用 GetHeadPosition()和 GetNext()来遍历链表，然后用 delete 删除其中的对象，最后调用 RemoveAll 删除链表所包含的指针。

数组类提供一个可动态调整数组大小的、有序的、按整数索引的对象数组。数组在内存中连续的存放固定长度的数组元素。数组的最大优点是可以随时存取任一元素。数组类包括基于模板的 CArray，它可以存放任何类型的数据；MFC 还为字节、字、双字、CString 对象、CObject 指针和无类型指针提供了预定义的类。数组的元素可以通过一个以零为基础的整数下标直接进行访问。下标操作符([])可用于设置或检取数组元素。如果要设置一个超过数组当前范围的元素，可以指定该数组是否自动增大。但是如果要调整数组大小时，则数组占用的内存块需要重新移动，效率很低。如果不要求调整数组大小，则对数组集合的访问和对标准 C 数组的访问一样快。在使用数组之前，应使用 SetSize 建立其大小，并分配内存。若不用 SetSize，象数组添加元素时会导致频繁的再分配内存和拷贝数据。数组类适用于那些需要快速检索、很少需要增加或删除元素的集合。

数组通过 GetAt(索引值)来访问数组中的成员。

要删除数组中的成员，可以用 GetSize()取得大小，然后遍历数组中成员，用 delete 删除，然后调用 RemoveAll()清除其中的指针数据。

下面是使用数组模板类的例子：

```
CArray<CMyClass,CMyClass&>myArray;
CMyClass myClass;
myArray->Add(myClass);
```

映射类以一种字典的方式组织数据。每个元素由一个关键字和一个数值项组成，关键字用作数值项的标识符，在集合中不允许重复，必须是唯一的。如果给出一个关键字，映射类会很快找到对应的数值项。映射查找是以哈希表的方式进行的，因此在映射中查找数值项的速度很快。除了映射类模板外，预定义的映射类能支持 CString 对象、字、CObject 指针和无类型指针。比如，CMapWordToOb 类创建一个映射表对象后，就可以用 WORD 类型的变量作为关键字来寻找对应的 CObject 指针。映射类最适用于需要根据关键字进行快速检索的场合。

要访问映射中的数据，可以用 GetStartPosition()定位到开始处，再用 GetNextAssoc 访问映射表中的成员。

要删除映射中的数据，可以用 GetStartPosition 和 GetNextAssoc 遍历并用 delete 删除对象，然后调用 RemoveAll。

下面是使用 CMap 模板类的例子：

```
CMap<CString,LPCSTR,CPerson,CPerson&>myMap;
```

```
CPerson person;
LPCSTR lpstrName=“Tom”;
myMap->SetAt(lpstrName,person);
```

有关集合类的使用可以参见 MFC 的例子 COLLECT。

对于文本编辑器，由于需要动态增加和删除每一行字符串，因此使用 CStringList 来保存文本编辑器的数据，CStringList 中的每一个元素是 CString 类型的，它代表一行字符。可以把 CString 看作一个字符数组，但它提供了丰富的成员函数，比字符数组功能强大的多。

另外，还需要增加一个数据成员 nLineNum，用于指示当前编辑行行号。如清单 7.3，在文档类的头文件 EditorDoc.h 中，加入以下代码：

清单 7.3 CEditorDoc.h

```
class CEditorDoc : public CDocument
{
protected: // create from serializationonly
CEditorDoc();
DECLARE_DYNCREATE(CEditorDoc)
// Attributes
public:
CStringList lines;
int nLineNum;
...
};
```

在定义了文档数据成员后，还要对文档数据成员进行初始化。

初始化文档类的数据成员

当用户启动应用程序，或从应用程序的 File 菜单种选择 New 选项时，都需要对文档类的数据成员进行初始化。一般的，类的数据成员的初始化都是在构造函数中完成的，在构造函数调用结束时对象才真正存在。但对于文档来说却不同，文档类的数据成员初始化工作是在 OnNewDocument 成员函数中完成的，此时文档对象已经存在。为什么呢”这是因为：在单文档界面(SDI)应用程序中，在应用程序启动时，文档对象就已经被创建。文档对象直到主框架窗口被关闭时才被销毁。在用户选择 File-New 菜单时，应用程序对象并不是销毁原来的文档对象然后重建新的文档对象，而只是重新初始化(Re-Initialization)文档对象的数据成员，这个初始化工作就是应用程序对象的 OnFileNew()消息处理成员函数通过调用 OnNewDocument()函数来完成的。试想，如果把初始化数据成员的工作放在构造函数中的话，由于对象已经存在，构造函数就无法被调用，也就无法完成初始化数据成员的工作。为了避免代码的重复，在应用程序启动时，应用程序对象也是通过调用 OnNewDocument 成员函数来初始化文档对象的数据成员的。如果是多文档界面(MDI)程序，则数据成员的初始化也可以放到构造函数中完成。因为在 MDI 中，选择 File->New 菜单时，应用程序对象就让文档模板创建一个新文档并创建对应的框架窗口和视图。但是，为了保证应用程序在单文档和多文档界面

之间的可移植性，我们还是建议将文档数据成员的初始化工作放在OnNewDocument()中完成，因为在 MDI 的应用程序对象的 OnFileNew 成员函数中，同样会调用文档对象的 OnNewDocument 成员函数。

在 OnNewDocument 成员函数中手工加入代码，如清单 7.4。

清单 7.4 OnNewDocument 成员函数

```
BOOL CEditorDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse thisdocument)
nLineNum=0;
POSITION pos;
pos=lines.GetHeadPosition();
while(pos!=NULL)
{
((CString)lines.GetNext(pos)).Empty();
}
lines.RemoveAll();
return TRUE;
}
```

其中 pos 类型为 POSITION，相当于链表的指针，指向链表当前元素。CStringList 的成员函数 GetHeadPosition()返回链表头指针。链表的 GetNext()函数以当前指针为参数，返回下一个元素指针，同时修改 pos，使它指向下一个元素。使用强制类型转换将 GetNext()函数返回的元素指针转化为 CString 类型，然后调用 CString::Empty()方法清除该行中的所有字符。通过一个 while 循环，清除所有文本行的数据。最后调用 CStringList 的 RemoveAll()成员函数，清除链表中的所有指针(注意：此时这些指针指向的元素已经被清除)。

提示：应用程序对象的成员函数 CWinApp::OnFileNew()在选择 File 菜单的 New 命令时被调用，缺省时在 InitInstance()中也会被调用。原理是在 InitInstance()中有一个命令行参数的执行过程，当命令行上没有参数时，函数 ParseCommandLine(cmdInfo)会调用 CCommandLineInfo:: 把 m_nShellCommand 成员置为 CCommandLineInfo::FileNew，这导致 ProcessShellCommand 成员函数调用 OnFileNew。用户可在 InitInstance()中显式的调用 OnFileNew()。/

应用程序对象的 OnFileNew 消息处理流程如下：首先判断应用程序是否有多个文档模板，若是，则显示一个对话框让用户选择创建哪种类型的文档(模板)。对话框中显示的字符串是与文档模板对象的构造函数的第一个参数相对应的字符串(若资源中无相应字符串则不显示)。然后该函数调用 CDocManager::OpenDocumentFile(NULL)成员函数，打开一个新

文件。CDocManager::OpenDocumentFile 函数调用了 CSingleDocTemplate 的 OpenDocumentFile，后者完成实际的创建文档、框架、视图工作。文档模板的 OpenDocumentFile 首先判断文档是否已经被创建，若未创建，则创建一个新文档。然后根据文件名参数是否为空，分别调用 CDocument 的 OnNewDocument( )和 CDocument 的 OnOpenDocument()函数。CDocument 的 OnNewDocument 首先调用 DeleteContents()，并将文档修改标志该为 FALSE(关闭窗口时将根据文档修改标志决定是否提示用户保存文档)。

清理文档类的数据成员

在关闭应用程序删除文档对象时，或用 File->Open 菜单打开一个文档时，需要清理文档中的数据。同文档的初始化一样，文档的清理也不是在文档的析构函数中完成，而是在文档的 CDocument::DeleteContents()成员函数中完成的(想想为什么")。析构函数只用于清除那些在对象生存期都将存在的数据项。DeleteContents()成员函数的调用有两个作用：

1.删除文档的数据；

2 确信一个文档在使用前为空。

前面已经说到，OnNewDocument 函数会调用 DeleteContents()函数。在用户选择 File->Open 菜单时，应用程序对象调用应用程序类的 OnFileOpen 成员函数，CWinApp::OnFileOpen 调用内部的文档管理类 CDocManager::OnFileOpen()成员函数，提示用户输入文件名。然后调用 CWinApp::OpenDocumentFile 打开一个文件。OpenDocumentFile 在打开文件后首先调用 DeleteContents 成员函数清理文档中的数据，确保消除以前打开的文档的数据被清理掉。

缺省的 DeleteContents 函数什么也不做。你需要重载 DeleteContents 函数，并编写自己的文档清理代码。要重载 DeleteContents 成员函数：

从 View 菜单下选择 ClassWizard，启动 ClassWizard，选择 Message Maps 页。在 ClassName 下拉列表框中选择 CEditorDoc，从 ObjectIDs 列表框选择 CEditorDoc，在 Message 列表框双击 DeleteContents。此时 DeleteContents 出现在 Member functions 列表框中，并被选中。点 Edit Code 按钮，开始编辑 DeleteContents 函数定义。在 DeleteContents 函数体中加入代码后，如清单 7.5 所示：

清单 7.5 CEditorDoc 的 DeleteContents 成员函数

```
void CEditorDoc::DeleteContents()
{
// TODO: Add your specialized code hereand/or call the base class
nLineNum=0;
/*删除集合类的数据：
用 GetHeadPosition 和 GetNext 遍历并用 delete 删除其中的数据，然后调用 RemoveAll()删除链表所包含的指针
*/
POSITION pos;
pos=lines.GetHeadPosition();
```

```
while(pos!=NULL)
{
((CString)lines.GetNext(pos)).Empty();
//调用 CString 的 Empty()方法清除文本行的数据,对于其它类型的对
//象，应当调用 delete 删除该对象
}
lines.RemoveAll();
CDocument::DeleteContents();
}
```

编辑器的 DeleteContents()实现与 OnNewDocument()基本相同，别的程序则可能会有所不同。

CDocument::OnOpenDocument 成员函数在调用 DeleteContents()函数后，将文档修改标记设置为 FALSE(未修改)，然后调用 Serialize 进行文档的串行化工作。

读写文档——串行化

文档对象的串行化是指对象的持续性，即对象可以将其当前状态，由其成员变量的值表示，写入到永久性存储体(通常是指磁盘)中。下次则可以从永久性存储体中读取对象的状态，从而重建对象。这种对象的保存和恢复的过程称为串行化。对象的可持续性允许你将一个复杂的对象网络保存到永久性存储体中，从而在对象从内存中删去后仍保持它们的状态。以后，可以从永久性存储器中载入对象并在内存中重载。保存和载入可持续化、串行化的数据通过 CArchive 对象作为中介来完成。

文档的串行化在 Serialize 成员函数中进行。当用户选择 File Save、Save As 或 Open 命令时，都会自动执行这一成员函数。AppWizard 只给出了一个 Serialze()函数的框架，读者要做的时定制这个 Serialize 函数。Serialize()函数由一个简单的 if-else 语句组成：

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

在框架中，Serialize 函数的参数 ar 是一个 CArchive 类型对象，它包含一个 CFile 类型的文件指针(类似于 C 语言的文件指针),执行一个文件。CArchive 对象为读写 CFile(文件类)对象中的可串行化数据提供了一种类型安全的缓冲机制。通常 CFile 代表一个磁盘文件；但它也可以是一个内存文件(CMemFile 对象)或剪贴板。一个给定的 CArchive 对象只能读数据

或写数据，而不能同时读写数据。当保存数据到 archive 对象中时，archive 把它放在一个缓冲区中。直到缓冲区满，才把数据写入它所包含的文件指针指向的 CFile 对象中。同样的，当从 archive 对象读数据时，archive 对象从文件中读取内容到缓冲区，然后再从缓冲区读入到可串行化的对象中。这种缓冲机制减少了访问物理磁盘的次数，从而提高了应用程序的性能。

在创建和使用一个 CArchive 对象之前，必须先创建一个 CFile 文件类对象。而且还必须确保 archive 的载入和保存状态同文件打开模式相兼容。幸运的是，应用程序框架已经为我们做好了这些工作。

当应用程序响应 File->Open、File-Save 和 File-Save As 命令时，应用程序框架都会通过调用 CDocument 成员函数(对于 File->Open 调用 OnOpenDocument，对于 File->Save 和 File->Save As 调用 OnSaveDocument)创建 CFile 对象，并以适当的方式打开文件，对于 File->Open 是打开文件并读，对于 Save 和 SaveAs 是打开文件并写。然后框架会自动把文件对象连接到一个 CArchive 对象上，并设置 CArchive 的读写方式。

在 Editor 的 Serialize()函数体内，我们看到 CArchive 对象有一个 IsStoring()成员函数。该成员函数告诉串行化函数是需要写入还是读取串行数据。如果数据要写入(Save 或 Save As)，IsStoring()返回布尔值 TRUE；如果数据是被读取，则返回 FALSE。

现在添加串行化操作代码，实现编辑器文档的读写功能。修改后的 Serialize()函数形式如清单 7.6。

清单 7.6 CEditorDoc 的串行化方法

```
////////////////////////////////////////////////////////////////////////////
// CEditorDoc serialization
voidCEditorDoc::Serialize(CArchive& ar)
{
CString s(“”);
int nCount=0;
CString item(“”);
if (ar.IsStoring())
{
POSITION pos;
pos=lines.GetHeadPosition();
if(pos==NULL)
{
return;
}
while(pos!=NULL)
{
item=lines.GetNext(pos);
ar<<item;
item.Empty();//clear the line buffer
```

```
}
}
else
{
// TODO: add loading code here
while(1)
{
try{
ar>>item;
lines.AddTail(item);
nCount++;
}
catch(CArchiveException *e)
{
if(e->m_cause!=CArchiveException::endOfFile)
{
TRACE0("Unknown exception loadingfile!\n");
throw;
}else
{
TRACE0("End of filereached...\n");
e->Delete();
}
break;
}
}
nLineNum=nCount;
}
}
```

在 If **子句中，从字符串链表中逐行读取字符串，然后通过调用** CArchive **对象的**<<**操作符，将文本行写入** ar **对象中。在** else **子句中，从** CArchive **对象逐一读入字符串对象，然后加入到链表中。由于在** Serialize() **函数的载入文档调用之前，框架已经调用** CDocument **的** DeleteContents() **成员函数作好了清理工作，这里不必再重复清理字符串链表。在载入字符串对象的同时，统计了字符串的个数即文本行数。由于这里使用** CString **的串行化，因此获得的文件不同于普通的文本文件。**

文档串行化与一般文件处理方式最大的不同在于：在串行化中，对象本身对读和写负责。在上面的例子中， CArchive **并不知道也不需要知道** CString **类的文本行内部数据结构，它只是调用** CString **类的串行化方法实现对象到文件的读写操作，也就是说，实际完成读写操作的是** CString **类，** CArchive **只是对象到** CFile **类的对象的一个中介。而文档的串行化正是通过调用文档中需要保存的各个对象的串行化方法来完成的。这几**

个对象的关系如图 7-8 所示。这里的对象必须是 MFC 对象，如果想让自己设计的对象也具有串行化能力，就必须定制该对象的串行化方法。有关定制串行化对象的技术在后面再作详细介绍。

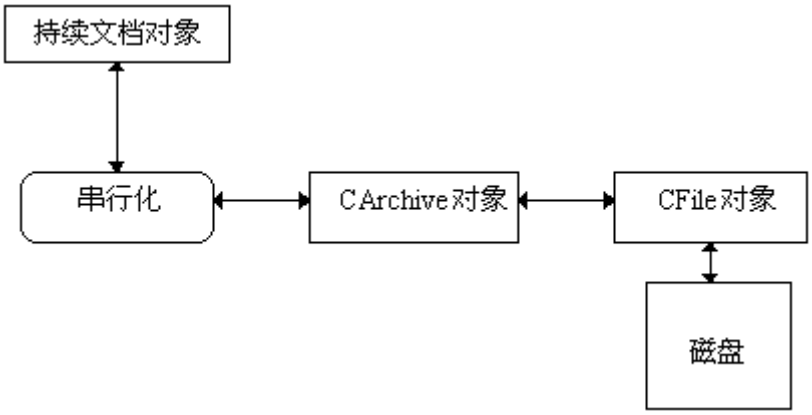


图 7-8 文档对象和文件对象

CArchive 对象使用重载的插入(<<)和提取(>>)操作符执行读和写操作。有人会说，这种方式很象 C++的输入输出流。其实，一个 archive 对象就是可以理解成一种二进制流。象输入/输出流一样，一个 archive 对象与一个文件相关联，并提供缓冲读写机制。但是，一个输入/输出流处理的是 ASCII 字符，而一个 archive 对象处理的是二进制对象。

如果不是使用框架创建和希望自己创建 CArchive 的话，可以这么做：

```
CFile file;//声明一个 CFile 类对象
file.Open(“ c:\\readme.txt ”,CFile::modeCreate|CFile::modeWrite);//打开文件
CArchivear(&file,CArchive::store);//用指向 file 的指针创建 CArchive 类对
//象，指定模式为 store 即存储，如果需要从 CArchive//中载入，可设为 load
...//一些串行化工作
ar.Close();//首先关闭 CArchive，然后关闭 file
file.Close();
```

在文档中引用视图类

有时要在文档对象中访问视图对象，而一个文档可能会对应多个视图，此时可以采用如下方法：

```
POSITION pos=GetFirstViewPosition();//获取视图链表的头指针
CEditorView*MyView=(CMyView*)GetNextView(pos)；
```

7.2.3 文本编辑器的视图类

视图类数据成员设计

现在设计文本编辑器的视图类。由于编辑器需要提供显示字体选择功能，因此在编辑器内增加一个数据成员代表当前所用的字体。另外，还需要两个变量 lHeight 和 cWidth 分别代表所用字体的高度和宽度，以便控制输出，因为 Windows 以图形方式输出，输出文本也需要程序员自己计算坐标。修改后的视图类如下面的片段所示：

```
class CEditorView : public CView
{
protected: // create from serializationonly
CEditorView();
DECLARE_DYNCREATE(CEditorView)
CFont* pFont;
int lHeight;
int cWidth;
...
}
```

也许有人会问：既然文档类包含应用程序的数据，而视图只负责输出，为什么不把数据全部放在文档类之中呢”从应用程序角度来看，视图是不包含数据的，显示文档的所有数据都是从文档对象中读取的。但这并不意味着视图不能包含数据成员。视图是从 CView **派生出来的类，作为类，它当然可以包含数据成员。而且，为了显示输出的需要，它经常包含一些与显示相关的数据成员。设计文档视结构的关键就是确切的定义用户文档应当包含哪些信息。那么，如何合理分配文档和视图的数据成员呢”一条简单的原则是：如何使用更方便，就如何分配数据成员。另外，还要看该数据成员是否需要保存到文档中，如果要保存到文档中，就必须放在文档中。因为文档可以对应多个视图，如果放在视图中，由于不同的视图的数据成员可以有不同的数值，这样文档保存时就不知道该使用哪一个数值了。一般的，与显示相关的数据成员都可以放在视图类中。在上面的文本编辑器中，我们并不需要保存编辑器使用何种字体这一信息，而这一信息又与文档显示密切相关，因此把它放在视图类中是很恰当的。这样的话，还可以用多个使用不同字体的视图观察同一文档。但是，如果编辑器是一个类似于** Microsoft WORD **之类的字处理器，在显示中支持多种字体的同一屏幕输出，这时需要保存字体信息，就要把字体信息放在文档类中了。**

视图数据成员的初始化

在文档类中，通过成员函数 OnNewDocument()**来完成文档类数据成员的初始化工作。视图类也提供了一个** CView::OnInitialUpdate()**成员函数来初始化视图类的数据成员。**

在以下情况下，应用程序将自动执行视图类的 OnInitialUpdate()**来初始化视图类数据成员：**

用户启动应用程序

从 File **菜单选择** New **菜单项，**CWinApp::OnFileNew **在调用** CDocument::OnNewDocument **后即调用** OnInitialUpdate **准备绘图输出；**

用 File->Open **命令打开一个文件，此时希望清除视图原有的显示内容**

在编辑器中要做的主要工作是对编辑器使用的字体的初始化，见清单 7.7**。**

清单 7.7 **视图的** OnInitialUpdate **方法**

```
void CEditorView::OnInitialUpdate()
{
// TODO: Add your specialized code hereand/or call the base class
CDC *pDC=GetDC();
pFont=new CFont();
if(!(pFont->CreateFont(0,0,0,0,FW_NORMAL,FALSE,FALSE,FALSE,
ANSI_CHARSET,OUT_TT_PRECIS,CLIP_TT_ALWAYS,
DEFAULT_QUALITY,DEFAULT_PITCH,"CourierNew")))
{
pFont->CreateStockObject(SYSTEM_FONT);
}
CFont*oldFont=pDC->SelectObject(pFont);
TEXTMETRIC tm;
pDC->GetTextMetrics(&tm);
lHeight=tm.tmHeight+tm.tmExternalLeading;
cWidth=tm.tmAveCharWidth;
pDC->SelectObject(oldFont);
CView::OnInitialUpdate();
}
```

OnInitialUpdate()首先调用 GetDC()取得当前窗口的设备上下文指针并存放在 pDC 中。设备上下文(简称 DC，英文全称是 device context)Windows 数据结构，它描述了在一个窗口中绘图输出时所需的信息，包括使用的画笔、画刷、当前选用的字体及颜色(前景色和背景色)、绘图模式，以及其它所需要的绘图信息。MFC 提供一个 CDC 类封装设备上下文，以简化存取 DC 的操作。

然后 OnInitialUpdate()创建视图显示时所用的字体。同前面提到的其他 MFC 对象如框架窗口一样，字体对象的创建也分为两步：第一步，创建一个 C++对象，初始化 CFont 的实例；第二步，调用 CreateFont()创建字体。除了 CreateFont 之外，还有两个创建字体的函数：CreateFontIndirect 和 FromHandle()，前者要求一个指向所需字体的 LOGFONT(逻辑字体)的指针作参数，后者需要一个字体句柄作参数。如果 CreateFont()因为某种原因失败，那么就调用 CreateStockObject()从预定义的 GDI 对象中创建字体。

注意：在 Windows 的 GDI 中，包含一些预定义的 GDI 对象，无需用户去创建，马上就可以拿来使用。这些对象称作库存(Stock)对象。库存对象包括 BLACK_BRUSH(黑色画刷)、DKGRAY_BRUSH(灰色画刷)、HOLLOW_BRUSH(空心画刷)、WHITE_BRUSH(白色画刷)、空画刷、黑色画笔、白色画笔以及一些字体和调色板等。CGdiObject::CreateStockObject()并不真正创建对象，而只是取得库存对象的句柄，并将该句柄连到调用该函数的 GDI 对象上。/

然后调用 CDC 的 SelectObject()方法，将字体选入到设备上下文中。SelectObject()函数原型如下：

```
CPen* SelectObject( CPen* pPen);
CBrush* SelectObject( CBrush*pBrush );
virtual CFont* SelectObject(CFont* pFont );
CBitmap* SelectObject( CBitmap*pBitmap );
int SelectObject( CRgn* pRgn );
```

SelectObject 的参数可以是一个画笔、画刷、字体、位图或区域，它们统称为 GDI(图形设备接口)对象。SelectObject 将一个 GDI 对象选入到一个设备上下文中，新选中的对象将替换原有的同类型对象，然后返回指向被替换的对象的指针。SelectObject()知道它所选中的对象的类型，且总是返回同类的旧对象的指针。还要存储返回的 CFont 指针，在退出 OnInitialUpdate 之前调用 pDC->SelectObject(oldFont)，将 CDC 重新设置成原来的初始状态。

读者以后编程也应当养成这样一个习惯：在用 SelectObject 选择新的 GDI 对象时，应当保存指向原先使用的 GDI 对象的指针，在绘图结束后，再用 SelectObject 选择原来的对象，设置 CDC 为其初始状态。否则的话，会有非法句柄留在设备上下文对象中，积累下去将导致无法预见的错误。但是，如果该设备上下文是自己创建而不是用参数传递过来的，则不必恢复画笔或刷子。象上面的例子，其实用户不必在退出时恢复原来的字体。而在下面要讲的 OnDraw 函数中，由于 pDC 是框架传给 OnDraw 的，因此在退出时必须恢复设备上下文中原来的字体设置。总之，如果用户能肯定画笔或刷子等 GDI 对象废弃以前设备对象会被销毁，则不必恢复设备上下文中 GDI 对象的设置。不过，为概念上的明确，还是建议调用恢复过程。

TEXTMETRIC 是一个数据结构，它包含字体的宽度、高度、字的前后空白等字段。调用 CDC::GetTextMetrics()获取字体的 TEXTMETRIC，从而取得字体的宽度和高度等信息。最后调用 CView 类的 OnInitialUpdate()函数来画视图。

由于在堆栈上创建了视图所用的字体对象 pFont，在关闭视图时就需要删除该字体对象。这部分工作在视图的析构函数中完成。修改视图的析构函数：

```
CEditorView::~CEditorView()
{
if(pFont!=NULL)
delete pFont;
}
```

视图的绘制

现在要让视图显示编辑器中的文本。AppWizard 为视图类 CEditorView 生成了一个 OnDraw()方法，当需要重画视图时，该函数就会被调用。清单 7.8 是编辑器的 OnDraw 函数定义：

清单 7.8 视图的 OnDraw 方法

```
/////////////////////////////////////////////////////////////////////////////
// CEditorView drawing
```

```
void CEditorView::OnDraw(CDC* pDC)
{
CEditorDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native datahere
CFont *oldFont;
//选择新字体
oldFont=pDC->SelectObject(pFont);
//纵向 yval 坐标为 0
int yval=0;
POSITION pos;
CString line;
//取得文本行链表的头指针
if(!(pos=pDoc->lines.GetHeadPosition()))
{
return;
}
//循环输出各文本行
while(pos!=NULL)
{
line=pDoc->lines.GetNext(pos);
pDC->TextOut(0,
yval,
line,
line.GetLength());
//更新 y 坐标值，让它加上文本行所用字体的高度
yval+=lHeight;
}
//恢复原来 DC 所用的字体
pDC->SelectObject(pFont);
}
```

框架调用视图的 CView::OnDraw(CDC* pDC)**方法完成屏幕显示、打印、打印预览功能，对于不同的输出功能它会传递不同的** DC **指针给** OnDraw()**函数。**

在 OnDraw()**函数中，首先调用** GetDocument()**函数，取得指向当前视图所对应的文档的指针。通过这个指针，来访问文档中的数据。以后在视图中修改文档中的数据，也是通过** GetDocument()**来取得文档指针，再通过该文档指针修改文档中的数据。**

在绘图时，可以通过传给 OnDraw **函数的一个设备上下文** DC **的指针** pDC **进行** GDI **调用。开始绘图之前，往往需要选择** GDI **资源(或** GDI **对象，包括画笔、刷子、字体等)，将它选入到设备上下文中。在文本编辑器中，我们选择一种字体** pFont **到设备上下文中，以后在窗口客户区**

的文本输出就都会使用该字体绘制。在绘制过程中，绘图代码是设备无关的，也就是说它并不需要知道目前使用的是什么设备(屏幕、打印机或其他绘图设备)。

读者以前如果用 Borland C++或 SDK 编写过 Windows 程序的话，都会知道：当窗口或窗口的一部分变成无效的话(比如其他窗口从本窗口上拖过、窗口调整大小等)，操作系统就会向窗口发送一条 WM_PAINT 消息。窗口接收到该消息之后，调用 Borland C++的 EvPaint()或 Visual C++的 OnPaint()完成窗口绘制工作。这里 OnDraw()函数也同样完成窗口绘图输出，这两者有什么关系呢”

我们先看一下 OnPaint()函数：

```
void CMyWindow::OnPaint()
{
CPaintDC dc(this); //用于窗口绘制的设备上下文
CString str(“ Hello,world! ”);
...
//绘图输出代码
dc.TextOut(10,10,str,str.GetLength());
}
```

在 OnPaint()函数中，首先创建一个 CPaintDC 类的对象 dc。CPaintDC 必需也只能用在 WM_PAINT 消息处理中。在 CPaintDC 类对象 dc 的构造函数中，调用了在 SDK 下需要显式调用的 BeginPaint 函数，取得处理 WM_PAINT 消息时所需的设备上下文。然后 OnPaint()函数使用该设备上下文完成各种输出。在 OnPaint()函数退出时，dc 对象被删除。在 dc 对象的析构函数中，包含了对 EndPaint 函数的调用。EndPaint 一方面释放设备上下文，另一方面还从应用消息队列中删除 WM_PAINT 消息。如果在处理 WM_PAINT 时不使用 CPaintDC，则 WM_PAINT 不被消除，会产生不断重画的现象。

视图是一个子窗口，它自然也从窗口类继承了 OnPaint()成员函数，用以响应 WM_PAINT 消息。类似于上面的例子，视图 OnPaint 处理函数首先创建一个与显示器相匹配的 CPaintDC 类的设备上下文对象 dc，但是 OnPaint 不再直接完成窗口输出，而是将设备上下文传给 OnDraw()成员函数，由 OnDraw()函数去完成窗口输出。当打印输出时，框架会调用视图的 DoPreparePrinting 创建一个与打印机相匹配的设备上下文并将该 DC 传递给 OnDraw()函数，由 OnDraw 函数完成打印输出。这样，OnDraw()函数就把用于屏幕显示和打印机输出的工作统一起来，真正体现了设备无关的思想。如果想知道当前 OnDraw 函数是在用于屏幕显示还是打印输出，可以调用 CView::IsPrinting()函数。当处于打印状态时，IsPrinting()返回 TRUE；在用于屏幕显示时，返回 FALSE。

文档修改时通知视图的更新

当文档以某种方式变化时，必须通知视图作相应的更新即重绘，以反应文档的变化。这种情况通常发生在用户通过视图修改文档时。此时，视图将调用文档的 UpdateAllViews 成员函数通知同一文档的所有视图对

自己进行更新。UpdateAllViews 将调用每个视图的 OnUpdate 成员函数，使视图的客户区无效。

5 视图的消息处理

视图作为一个子窗口，当然可以处理消息。但是应用程序运行时，除了视图外，还有应用程序对象、主框架窗口、文档等，它们都是可以处理消息的。那么消息传递过程是什么样的呢”

MFC 的命令消息按以下方式传递：

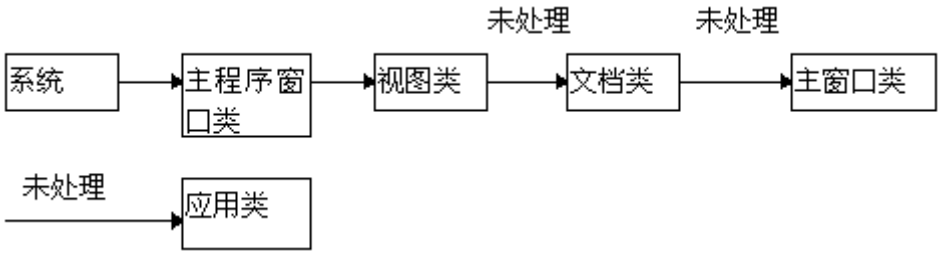


图 7-9 文档视结构中的消息传递

键盘消息处理

前面的视图绘制就是完成窗口消息 WM_PAINT 的处理。编辑器要接收用户的键盘输入，就必须处理键盘消息；另外，在用户输入字符时，还必须马上就把用户输入的内容在屏幕上显示出来。

用 ClassWizard 生成处理 WM_CHAR 消息的函数 OnChar()，然后打开该函数进行编辑。修改后的 OnChar 函数如清单 7.9：

清单 7.9 CEditorView 的 OnChar()成员函数

```
void CEditorView::OnChar(UINT nChar,UINT nRepCnt, UINT nFlags)
{
CEditorDoc* pDoc=GetDocument();
CClientDC dc(this);
CFont *oldFont;
//选择新字体
oldFont=dc.SelectObject(pFont);
CString line(“”);//存放编辑器当前行字符串
POSITION pos=NULL;//字符串链表位置指示
if(nChar=='\r')
{
pDoc->nLineNum++;
}
else
{
//按行号返回字符串链表中位置值
pos=pDoc->lines.FindIndex(pDoc->nLineNum);
if(!pos)
{
//没有找到该行号对应的行，因此它是一个空行，
//我们把它加到字符串链表中。
```

```
line+=(char)nChar;
pDoc->lines.AddTail(CString(line));
}
else{
//当前文本行还没有换行结束，因此将文本加入到行末
line=pDoc->lines.GetAt(pos);
line+=(char)nChar;
pDoc->lines.SetAt(pos,line);
}
TEXTMETRIC tm;
dc.GetTextMetrics(&tm);
dc.TextOut(0,
(int)pDoc->nLineNum*tm.tmHeight,
line,
line.GetLength());
}
pDoc->SetModifiedFlag();
dc.SelectObject(oldFont);
CView::OnChar(nChar,nRepCnt,nFlags);
}
```

因为编辑器要将用户输入内容加入到文本行缓冲区中，因此首先调用 GetDocument()获取指向文档的指针，以便对文档中的数据进行修改。

为了在收到键盘输入消息后在窗口中输入字符，需要定义一个 CClientDC 类的对象 dc。CClientDC 是用于管理窗口客户区的设备上下文对象，它在构造函数中调用 GetDC()取得窗口客户区设备上下文，在析构函数中调用 ReleaseDC()释放该设备上下文。CClientDC 同样用于在窗口客户区的输出，它与 CPaintDC 不同之处在于：

CPaintDC 专门用于在窗口 OnPaint()中的输出，而不能用于其它非窗口重画消息的处理。如果不是在 OnDraw 或 OnPaint()中绘图，则需要创建一个 CClientDC 对象，然后调用 CClientDC 的方法来完成绘图输出。

OnChar()接下去处理用户输入。如果输入是一个回车，则将总行数 nLineNum 加一，否则将输入字符加到当前行行末。最后调用 TextOut 函数输出当前编辑中的文本行。

最后调用文档的 SetModifiedFlag()方法设置文档的修改标志。SetModifiedFlag()函数原型如下：

```
void SetModifiedFlag( BOOLbModified = TRUE );
```

从函数原型可以看出，函数缺省参数为 TRUE。当调用 SetModifiedFlag 时，将文档内的修改标志置为真。如果用户执行了 Save 或 Save As 操作，则将文档的修改标志置为假。这样，当用户关闭文档的最后一个视图时，框架根据该修改标记决定是否提示用户保存文档中的数据到文件。如果用户上次作了修改还没有存盘，则弹出一个消息框，提示是否保存文件。这些都是框架程序来完成的。

用户如果在视图的其它任何地方修改了文档，也必须调用 SetModifiedFlag 来设置文档修改标记，以便关闭窗口时让框架提示保存文档。

菜单消息处理

现在还要增加一个菜单，用户选择菜单时会弹出一个字体选择对话框，让用户选择视图输出文档时所用的字体。用菜单编辑器在 View 菜单下增加一个菜单项“ Select Font ”，菜单项相关参数如下：

菜单名：Select &Font

菜单 ID：ID_SELECT_FONT

提示文字：Select a font forcurrent view

然后用 ClassWizard 为该菜单项生成消息处理函数 SelectFont。在选择消息响应的类时，用户可以选择文档、视图、框架或应用程序类，这根据具体情况而定。如果操作是针对某一视图(比如象本例中改变字体操作)，则消息处理放在视图中比较合适。如果操作是针对文档的(比如要显示文档中对象的属性等)，则放在文档中处理比较合适。如果选项对应用程序中的所有文档和视图都有效(即是全局的选项)，那么可以把它放在框架窗口中。

修改 OnSelectFont()函数，使它能显示字体选择对话框,修改后的 OnSelect 函数见清单 7.10：

清单 7.10 OnSelectFont()函数

```
void CEditorView::OnSelectFont()
{
CFontDialog dlg;
if(dlg.DoModal()==IDOK)
{
LOGFONT LF;
//获取所选字体的信息
dlg.GetCurrentFont(&LF);
//建立新的字体
pFont->DeleteObject();
pFont->CreateFontIndirect(&LF);
Invalidate();
UpdateWindow();
}
}
```

在 OnSelectFont()消息处理函数中，首先定义一个选择字体公用对话框，然后显示该对话框，返回所选的字体。有关选择字体公用对话框的知识参见第五章对话框技术。字体对话框通过 GetCurrentFont()返回逻辑字体信息。所谓逻辑字体是一种结构，它包含了字体的各种属性的描述，包括字体的名字、宽度、高度和是否斜体、加粗等信息。字体对象首先通过 DeleteObject 删除原来的字体对象，然后通过 CreateFontIndirect、利用逻辑字体的属性来创建字体。由于我们选择了一种新的字体，所以要

用新的字体来重绘视图。为此，调用 Invalidate()函数向视图发送 WM_PAINT 消息。由于 WM_PAINT 消息级别比较低，不会立即被处理。因此，再调用 UpdateWindow()强制窗口更新。这也是一种常用的技巧。

现在已经完成了编辑器文档类和视图类的设计，对主框架窗口类不需要修改。编译、链接并运行程序，弹出文本编辑器窗口。试着输入几行文本，存盘。然后再载入刚才保存的文件，如图 7-10。在 File-Exit 菜单项上面，有一个文件名列表，列出最近打开过的文件，这个表称作 MRU 表(MRU 是英文 Most Recently Used 的缩写)。可以从 MRU 中选择一个文件名，打开该文件。

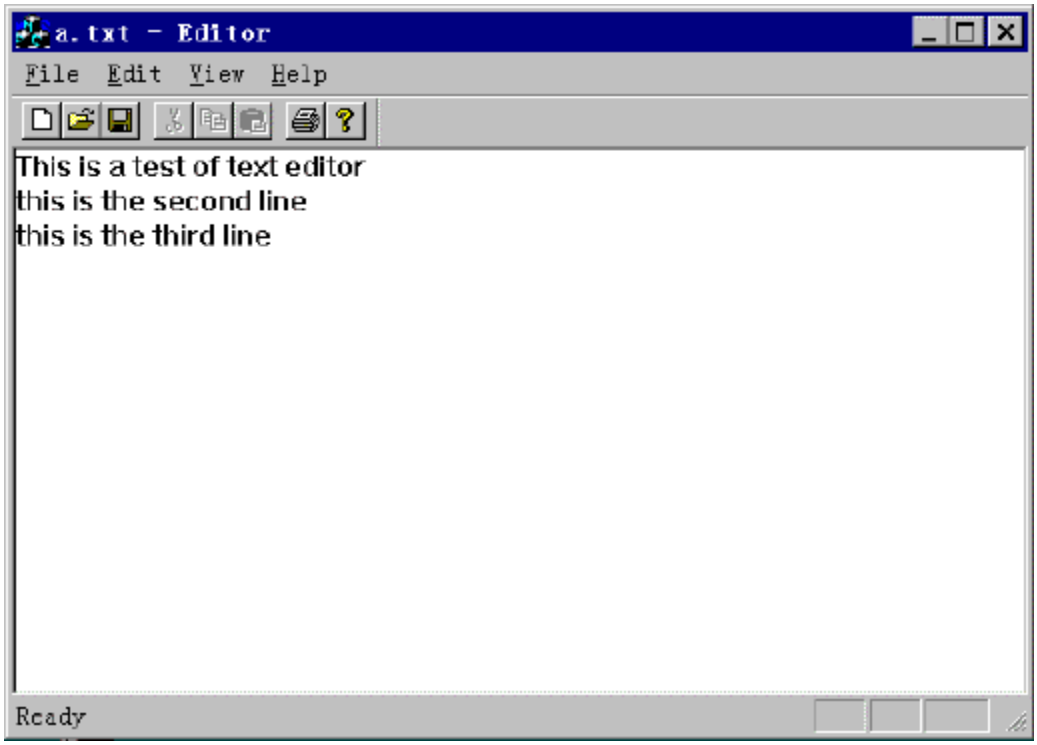


图 7-10 一个简单的文本编辑器

7.3 让文档视结构程序支持卷滚

但是，编辑器现在还不支持卷滚。当文本行超过窗口大小时，窗口并不自动向上滚动以显示输入的字符。当打开一个文件时，如果文件大小超过窗口大小，也无法通过卷滚视图来看文档的全部内容。现在我们要让编辑器增加卷滚功能。

7.3.1 逻辑坐标和设备坐标

在引入文档卷滚功能之前，首先要介绍以下逻辑坐标和设备坐标这两个重要概念。

在 Windows 中，文档坐标系称作逻辑坐标系，视图坐标系称为设备坐标系。它们之间的关系如下图所示：

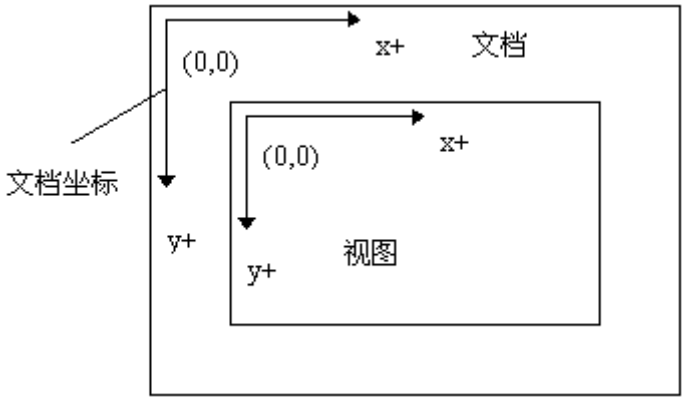


图 7-11 文档坐标和视图坐标

逻辑坐标按照坐标设置方式(又成为映射模式)可分为 8 种，它们在坐标上的特性如下表所示：

表 7-1 各种映射模式下的坐标转换方式

映射模式	逻辑单位	x 递增方向	y 递增方向
MM_TEXT	像素	向右	向下
MM_LOMETRIC	0.1mm	向右	向上
MM_HIMETRIC	0.01mm	向右	向上
MM_LOENGLISH	0.01inch	向右	向上
MM_HIENGLISH	0.001inch	向右	向上
MM_TWIPS	1/1440inch	向右	向上
MM_ISOTROPIC	可调整(x=y)	可选择	可选择
MM_ANISOTROPIC	可调整(x!=y)	可选择	可选择

我们一般使用的映射模式是 MM_TEXT，它也是缺省设置。在该模式下，坐标原点在工作区左上角，而 x 坐标值是向右递增，y 坐标值是向下递增，单位值 1 代表一个像素。

要设置映射模式，可以调用 CDC::SetMapMode()函数。

```
CClientDC dc;
nPreMapMode=dc.SetMapMode(nMapMode);
```

它将映射模式设置为 nMapMode，并返回前一次的映射模式 nPreMapMode，GetMapMode 可取得当前的映射模式：

CClientDC dc;

nMapMode=dc.GetMapMode();

MFC 绘图函数都使用逻辑坐标作为位置参数。比如

CString str(“ Hello,world! ”);

dc.TextOut(10,10,str,str.GetLength());

这里的(10,10)是逻辑坐标而不是像素点数(只是在缺省映射模式 MM_TEXT 下，正好与像素点相对应)，在输出时 GDI 函数会将逻辑坐标(10,10)依据当前映射模式转化为“ 设备坐标 ”，然后将文字输出在屏幕上。

设备坐标以像素点为单位，且 x 轴坐标值向右递增，y 轴坐标值向下递增，但原点(0,0)位置却不限定在工作区的左上角。依据设备坐标的原点和用途，可以将 Windows 下使用的设备坐标系统分为三种：工作区坐标系统，窗口坐标系统和屏幕坐标系统。

(1)工作区坐标系统：

工作区坐标系统是最常见的坐标系统，它以窗口客户区左上角为原点(0,0)，主要用于窗口客户区绘图输出以及处理窗口的一些消息。鼠标消息 WM_LBUTTONDOWN、WM_MOUSEMOVE 传给框架的消息参数以及 CDC 一些用于绘图的成员都是使用工作区坐标。

(2)屏幕坐标系统：

屏幕坐标系统是另一类常用的坐标系统，以屏幕左上角为原点(0,0)。以 CreateDC(“ DISPLAY ” , ...)或 GetDC(NULL)取得设备上下文时，该上下文使用的坐标系就是屏幕坐标系。

一些与窗口的工作区不相关的函数都是以屏幕坐标为单位，例如设置和取得光标位置的函数 SetCursorPos()和 GetCursorPos()；由于光标可以在任何一个窗口之间移动，它不属于任何一个单一的窗口，因此使用屏幕坐标。弹出式菜单使用的也是屏幕坐标。另外，CreateWindow、MoveWindow、SetWindowPlacement()等函数用于设置窗口相对于屏幕的位置，使用的也是屏幕坐标系统。

(3)窗口坐标系统：

窗口坐标系统以窗口左上角为坐标原点,它包含了窗口控制菜单、标题栏等内容。一般情况下很少在窗口标题栏上绘图，因此这种坐标系统很少使用。

三类设备坐标系统关系如下图所示：

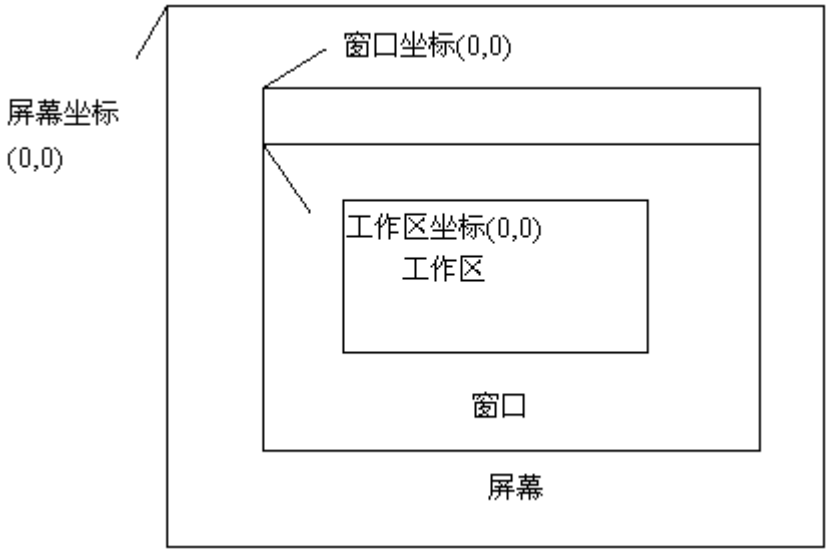


图 7-12.**三类设备坐标**

MFC **提供** ClientToScreen()、ScreenToClient()**两个函数用于完成工作区坐标和屏幕坐标之间的转换工作。**

voidScreenToClient(LPPOINTlpPoint)const;

voidScreenToClient(LPRECTlpRect)const;

voidClientToScreen(LPPOINTlpPoint)const;

voidClientToScreen(LPRECTlpRect)const;

其实，我们在前面介绍弹出式菜单时已经使用了 ClientToScreen **函数。在那里，由于弹出式菜单使用的是屏幕坐标，因此当处理弹出式菜单快捷键** shift+F10 **时，如果要在窗口左上角**(5,5)**处显示快捷菜单，就必须先调用** ClientToScreen **函数将客户区坐标**(5,5)**转化为屏幕坐标。**

```
CRect rect;
GetClientRect(rect);
ClientToScreen(rect);
point =rect.TopLeft();
point.Offset(5,5);
```

在视图滚动后，如果用户在视图中单击鼠标，那么会得到鼠标位置的设备(视图)坐标。在使用这个数据处理文档(比如画点或画线)时，需要把它转化为文档坐标。这是因为利用 MFC **绘图时，所有传递给** MFC **作图的坐标都是逻辑坐标。当调用** MFC **绘图函数绘图时，**Windows **自动将逻辑坐标转换成设备坐标，然后再绘图。设备上下文类** CDC **提供了两个成员函数** LPToDP **和** DPToLP **完成逻辑坐标和设备坐标之间的转换工作。如其名字所示那样，**LPToDP **将逻辑坐标转换为设备坐标，**DPToLP **将设备坐标转换为逻辑坐标。**

void LPtoDP(LPPOINT lpPoints, int nCount = 1) const;

void LPtoDP(LPRECT lpRect) const;

voidLPtoDP(LPSIZE lpSize) const;

voidDPtoLP(LPPOINT lpPoints, int nCount = 1) const;

voidDPtoLP(LPRECT lpRect) const;

voidDPtoLP(LPSIZE lpSize) const;

7.3.2 **滚动文档**

由于 MFC **绘图函数使用的是逻辑坐标，因此用户可以在一个假想的通常是比视图要大的“文档窗口”中绘图；**Windows **自动在幕后完成坐标转换工作，并将落在视图范围内的那一部分“文档窗口”显示出来，其余的部分被裁剪。**

但是光这样还不能卷滚文档。要卷滚显示文档，还必须知道文档卷滚到了什么位置；一旦用户拖动滚动条时要告诉视图改变在文档中的相应位置。所有这些，由 MFC **的** CScrollView **来完成。**

MFC **提供了** CScrollView **类，简化了滚动需要处理的大量工作。除了管理文档中的滚动操作外，**MFC **还通过调用** Windows API **函数画出滚动条、箭头和滚动光标。它还负责处理：**

用户初始化滚动条范围(通过滚动视图的 SetScrollRange()**方法)**

处理滚动条消息，并滚动文档到相应位置

管理窗口和视图的尺寸大小

调整滚动条上滑块(或称拇指框)的位置，使之与文档当前位置相匹配

程序员要做的工作是：

从 CScrollView **类中派生出自己的视图类，以支持卷滚**

提供文档大小，确定滚动范围和设置初始位置

协调文档位置和屏幕坐标

要让应用程序支持卷滚，可以在用 AppWizard **生成框架程序时就指定视图的基类为** CSrollView**。可以在** AppWizard **的** MFC AppWizard-Step 6 of 6 **对话框中，在对话框上方应用程序所包含的类中选择** CEditorView**，然后在** Base Class **下拉列表框中选择应用程序视图类的基类为** CScrollView**，如图** 7-11 **所示：**

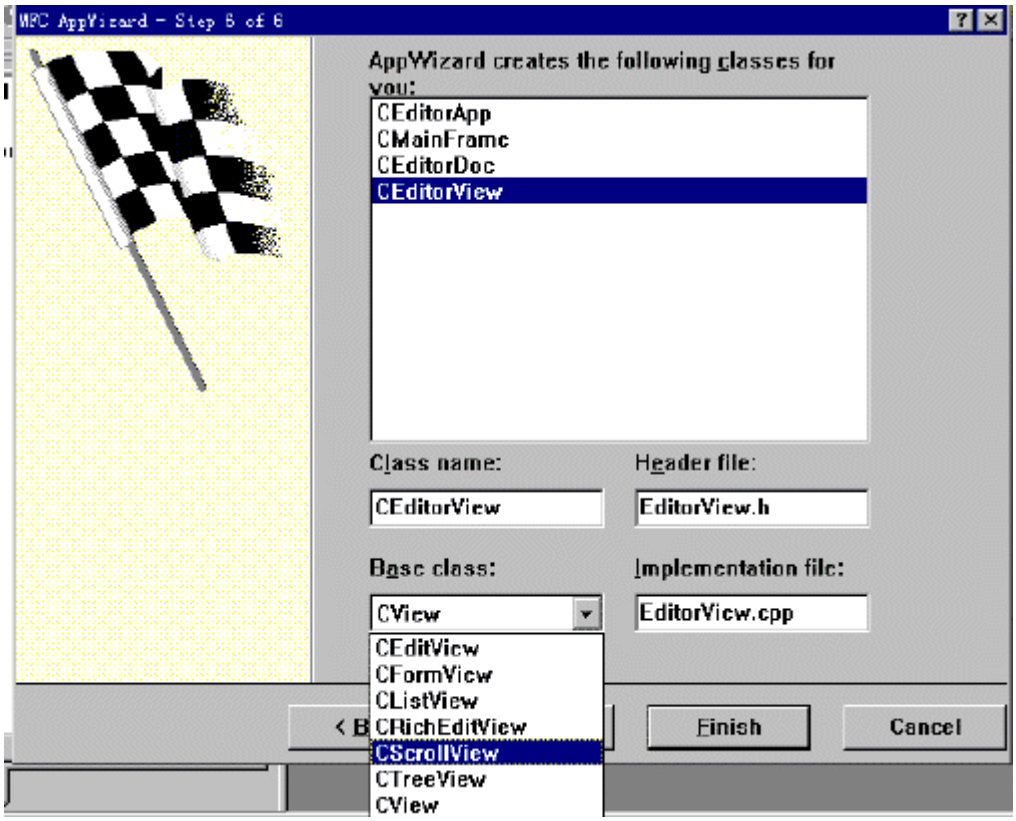


图 7-13 为应用程序的视图类指定基类

现在我们要手工修改 CEditorView，**使它的基类为** CScrollView。

1.**修改视图类所对应的头文件，将所有用到** CView **的地方改为** CScrollView。**通常，首先修改视图类赖以派生的父类，形式如下：**

classCEditorView:public CScrollView

2.**修改视图类实现的头文件，把所有用到** CView **的地方改为** CScrollView。**首先修改** IMPLEMENT_DYNACREATE **一行：**

IMPLEMENT_DYNACREATE(CEditorView,CScrollView)

然后修改 BEGIN_MESSAGE_MAP **宏**

BEGIN_MESSAGE_MAP(CEditorView,CScrollView)

然后将其他所有用到 CView **的地方改为** CScrollView。

一个更简单的方法是：使用 Edit-Replace **功能，进行全局替换。**

到现在为止，已经将编辑器视图类 CEditorView **的基类由** CView **转化为** CScrollView。

现在，要设置文档大小，以便让 CScrollView **知道该如何处理文档。视图必需知道文档的卷滚范围，这样才能确定何时卷滚到文档的头部和尾部，以及当拖动卷滚条的滑块时按适当比例调整文档当前显示位置。**

为此，我们首先在文档类 CEditorDoc **的头文件** editordoc.h **中增加一个** CSize **类型的数据成员** m_sizeDoc **用以表示文档的大小。**CSize **对象包含** cx **和** cy **两个数据成员，分别用于存放文档的** x **方向坐标范围和** y **方向坐标范围。另外，还要提供一个成员函数** GetDocSize()**来访问该文档大小范围数据成员。修改后的** editordoc.h **如清单** 7.11。

清单 7.11 CEditorDoc **头文件**

```
class CEditorDoc: public CDocument
{
protected: //create from serialization only
CEditorDoc();
DECLARE_DYNCREATE(CEditorDoc)
//保存文档大小
CSizem_sizeDoc;
// Attributes
public:
CSizeGetDocSize(){return m_sizeDoc;}
// Operations
public:
CStringListlines;
int nLineNum;
.....
};
```

既然增加了 m_sizeDoc 这一数据成员，就需要在 CEditorDoc 构造函数中进行初始化，给 m_sizeDoc 设置一合理的数值，比如说 x=700，y=800。构造函数如清单 7.12。

清单 7.12 CEditorDoc 的构造函数

```
CEditorDoc::CEditorDoc()
{
// TODO: addone-time construction code here
nLineNum=0;
m_sizeDoc=CSize(700,800);
}
```

一个设计优秀的应用程序应当能够动态调整文档的卷滚范围。比如，在 WORD 中新建一个文件时，在“页面模式”下将可卷滚范围设为一页大小。随着用户输入，逐渐增加文档的卷滚范围。但是这里为简明起见，将文档卷滚范围设为固定大小 700X800 点像素大小。设置文档大小通过由视图类的 CEditorView::OnInitialUpdate()调用 SetScrollSizes()成员函数来完成。

SetScrollSizes()用于设置文档卷滚范围。一般在重载 OnInitialUpdate() 成员函数或 OnUpdate()时调用该函数，用以调整文档卷滚特性。比如，在文档初始显示或文档大小作了调整之后。

清单 7.13 在 OnInitialUpdate()中设置卷滚范围

```
voidCEditorView::OnInitialUpdate()
{
// TODO: Addyour specialized code here and/or call the base class
CDC*pDC=GetDC();
pFont=newCFont();
if(!(pFont->CreateFont(0,0,0,0,FW_NORMAL,FALSE,FALSE,FALSE,
```

```
ANSI_CHARSET,OUT_TT_PRECIS,CLIP_TT_ALWAYS,
DEFAULT_QUALITY,DEFAULT_PITCH,"CourierNew")))
{
pFont->CreateStockObject(SYSTEM_FONT);
}
CFont*oldFont=pDC->SelectObject(pFont);
TEXTMETRICtm;
pDC->GetTextMetrics(&tm);
lHeight=tm.tmHeight+tm.tmExternalLeading;
cWidth=tm.tmAveCharWidth;
SetScrollSizes(MM_TEXT,GetDocument()->GetDocSize());
CScrollView::OnInitialUpdate();
}
```

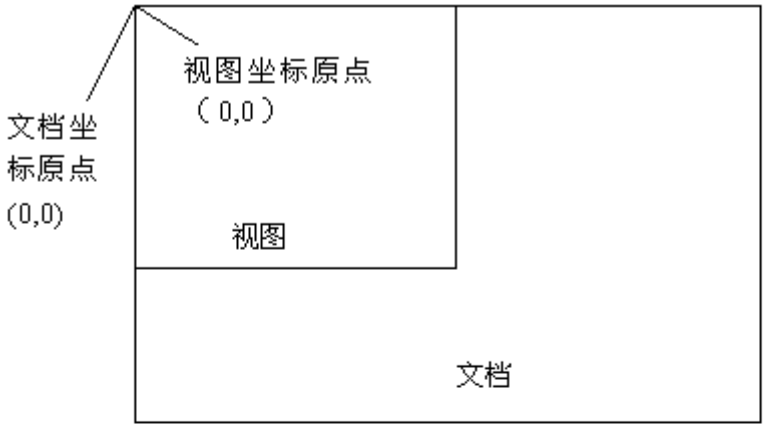
SetScrollSizes()**第一个参数为映射模式。**SetScrollSizes()**可以使用除** MM_ISOTROPIC **和** MM_ANISOTROPIC **之外的其他任何映射模式。** SetScrollSizes()**第二个参数为文档大小，用一个** CSize **类型的数值表示。**

另外，我们还要检查两个包含绘图输出功能的函数： CEditorView::OnChar()**和** CEditorView::OnDraw()**函数。**

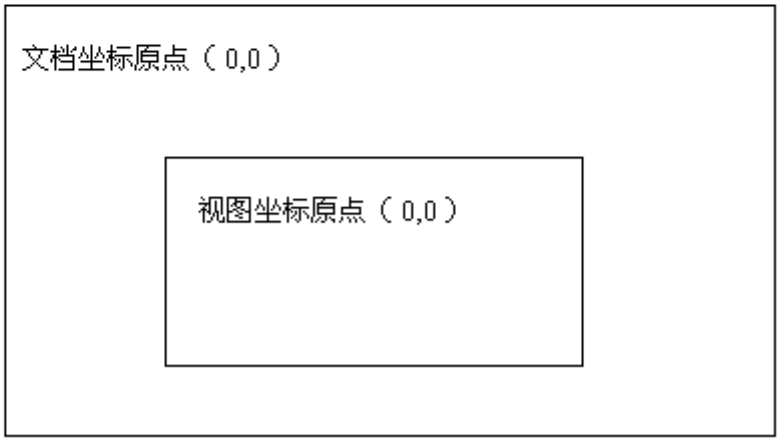
```
voidCEditorView::OnChar(UINT nChar, UINT nRepCnt, UINTnFlags)
{
CEditorDoc*pDoc=GetDocument();
CClientDCdc(this);
CStringline("");//存放编辑器当前行字符串
POSITIONpos=NULL;//字符串链表位置指示
if(nChar=='\r')
{
pDoc->nLineNum++;
}
else
{
//按行号返回字符串链表中位置值
pos=pDoc->lines.FindIndex(pDoc->nLineNum);
if(!pos)
{
//没有找到该行号对应的行，因此它是一个空行，
//我们把它加到字符串链表中。
line+=(char)nChar;
pDoc->lines.AddTail(CString(line));
}
else{
//there is aline,so add the incoming char to the end of
//the line
```

```
line=pDoc->lines.GetAt(pos);
line+=(char)nChar;
pDoc->lines.SetAt(pos,line);
}
TEXTMETRICtm;
dc.GetTextMetrics(&tm);
dc.TextOut(0,
(int)pDoc->nLineNum*tm.tmHeight,
line,
line.GetLength());
}
pDoc->SetModifiedFlag();
SetScrollSizes(MM_TEXT,GetDocument()->GetDocSize());
CScrollView::OnChar(nChar,nRepCnt,nFlags);
}
```

在程序运行开始的时侯，视图坐标原点和文档坐标原点是重合的。但是，当用户拖动滚动条时，视图原点就与文档原点不一致了，如图 7-14**。由于** GDI **是按照文档坐标(逻辑坐标)来输出图形的，这样自然就无法正确显示文档内容。**



文档滚动前



文档滚动后

图 7-14 **文档滚动前后文档坐标原点和视图坐标原点的变化**

这时，要想获得正确输出，就必须调整视图坐标，让视图坐标原点和文档坐标原点重合，如图 7-15 **所示。**

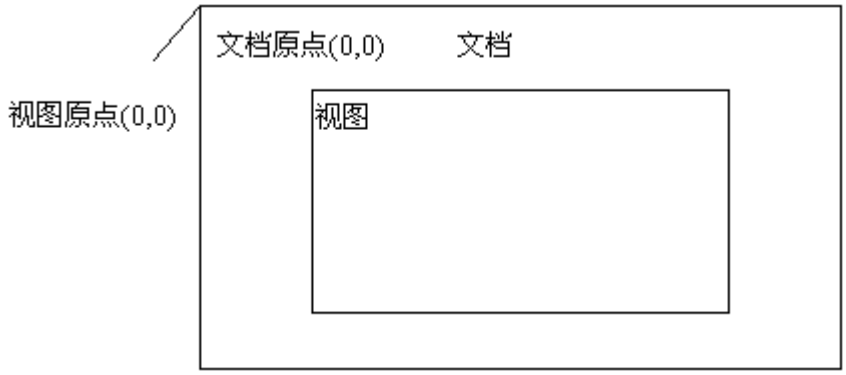


图 7-15 **调整视图设备上下文原点后**

CScrollView **视图类提供了一个** CScrollView::OnPrepareDC()**成员函数，完成视图设备上下文坐标原点的调整工作。**

现在修改 OnChar()**，加入** OnPrepareDC()**函数，见清单** 7.15**。**

清单 7.15 **修改后的** OnChar **成员函数**

```
voidCEditorView::OnChar(UINT nChar, UINT nRepCnt, UINTnFlags)
{
CEditorDoc*pDoc=GetDocument();
CClientDCdc(this);
OnPrepareDC(&dc);
CFont*pOldFont=dc.SelectObject(pFont);
CStringline(“”);//存放编辑器当前行字符串
POSITIONpos=NULL;//字符串链表位置指示
if(nChar=='\r')
{
pDoc->nLineNum++;
}
else
{
//按行号返回字符串链表中位置值
pos=pDoc->lines.FindIndex(pDoc->nLineNum);
if(!pos)
{
//没有找到该行号对应的行，因此它是一个空行，
//我们把它加到字符串链表中。
line+=(char)nChar;
pDoc->lines.AddTail(CString(line));
}
else{
//there is aline,so add the incoming char to the end of
//the line
line=pDoc->lines.GetAt(pos);
line+=(char)nChar;
pDoc->lines.SetAt(pos,line);
}
TEXTMETRICtm;
dc.GetTextMetrics(&tm);
dc.TextOut(0,
(int)pDoc->nLineNum*tm.tmHeight,
line,
line.GetLength());
}
pDoc->SetModifiedFlag();
dc.SelectObject(pOldFont);
SetScrollSizes(MM_TEXT,GetDocument()->GetDocSize());
CScrollView::OnChar(nChar,nRepCnt,nFlags);
}
```

但是，对于视图 OnDraw()函数，则不需要作这样的调整。这是因为，框架在调用 OnDraw()之前，已经自动调用了 OnPrepareDC()成员函数完成设备上下文坐标调整工作了。

提示：对于框架传过来的设备上下文，不需要调用 OnPrepareDC(), 因为框架知道它是用于绘图的，因此事先调用了 OnPrepareDC()作好了坐标调整工作。如果是自己构造或用 GetDC()取得得设备上下文，则需要调用 OnPrepareDC()完成设备上下文坐标调整工作。/

现在编辑器已经能够支持文档滚动了，如图 7-16。

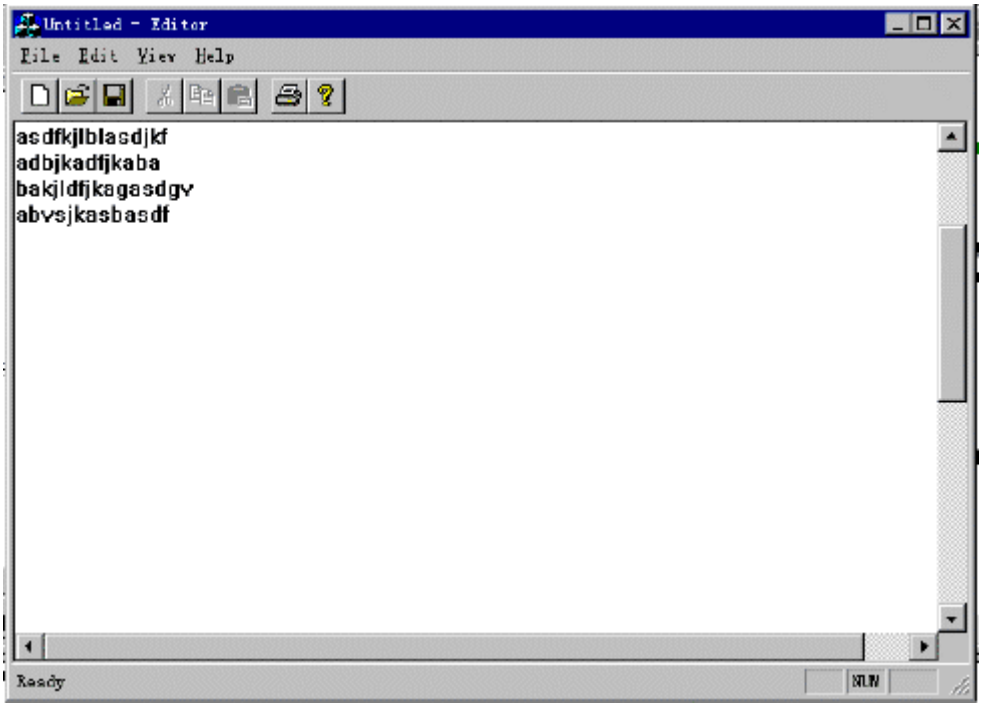


图 7-16 支持滚动的文本编辑器

7.4 定制串行化

前面编辑器的例子使用 CString 类的字符串来保存文本行，由于它是 MFC 类，因此可以串行化自己，将自己写入磁盘或从磁盘文件中读取二进制数据来建立对象。那么，如果不是标准的 MFC 类，比如用户自己定义的类，如何让它支持串行化呢”下面，我们结合前面第五章提到的就业调查表的例子来演示如何让用户定义的类支持串行化功能。

要让用户定义的类支持串行化，一般分为五步：

1.从 CObject 或其派生类派生出用户的类

2.重载 Serialize()成员函数，加入必要的代码，用以保存对象的数据成员到 CArchive 对象以及从 CArchive 对象载入对象的数据成员状态。

3.在类声明文件中，加入 DECLARE_SERIAL 宏。编译时，编译器将扩充该宏，这是串行化对象所必需的。

4.定义一个不带参数的构造函数。

5.在实现文件中加入 IMPLEMENT_SERIAL 宏。

```
classCRegister:public CObject
{
public:
DECLARE_SERIAL(CRegister)
//必需提供一个不带任何参数的空的构造函数
CRegister(){};
public:
CStringstrIncome;
CString strKind;
BOOL bMarried;
CString strName;
int nSex;
CString strUnit;
int nWork;
UINT nAge;
voidSerialize(CArchive&);
};
```

MFC 在从磁盘文件载入对象状态并重建对象时，需要有一个缺省的不带任何参数的构造函数。串行化对象将用该构造函数生成一个对象，然后调用 Serialize()函数，用重建对象所需的值来填充对象的所有数据成员变量。

构造函数可以声明为 public、protected 或 private。如果使它成为 protect 或 private，则可以确保它只被串行化过程所使用。

在类定义文件中给出 Serialize()的定义。它包括对象的保存和载入两部分。前面已经提到，CArchive 类提供一个 IsStoring()成员函数指示是保存数据到磁盘文件还是从磁盘文件载入对象。

```
voidCRegister::Serialize(CArchive& ar)
```

```
{
//首先调用基类的 Serialize()方法。
CObject::Serialize(ar);
if(ar.IsStoring())
{
ar<<strIncome;
ar<<strKind;
ar<<(int)bMarried;
ar<<strName;
ar<<nSez;
ar<<strUnit;
ar<<nWork;
ar<<(WORD)nAge;
}
else
{
ar>>strIncome;
ar>>strKind;
ar>>(int)bMarried;
ar>>strName;
ar>>nSex;
ar>>strUnit;
ar>>nWork;
ar>>(WORD)nAge;
}
}
```

我们看到，对象的串行化实际上是通过调用对象中的数据成员的串行化来完成的。

注意:CArchive **类的**>>**和**<<**操作符并不支持所有的标准数据类型。它支持的数据类型有：**CObject、BYTE、WORD、int、LONG、DWORD、float **和** double**。其他的类型的数据要进行串行化输入输出时，需要将该类型的数据转化为上述几种类型之一方可。**/

另外，在类的实现(类定义)文件开始处，还要加入 IMPLEMENT_SERIAL **宏。**

IMPLEMENT_SERIAL(CRegister, CObject, 1)

IMPLEMENT_SERIAL **宏用于定义一个从** CObject **派生的可串行化类的各种函数。宏的第一和第二个参数分别代表可串行化的类名和该类的直接基类。**

第三个参数是对象的版本号，它是一个大于或等于零的整数。MFC **串行化代码在将对象读入内存时检查版本号。如果磁盘文件上的对象的版本号和内存中的对象的版本号不一致，**MFC **将抛出一个** CArchiveException **异常，阻止程序读入一个不匹配版本的对象。**

现在，我们就可以象使用标准 MFC **类一样使用** CRegister **的串行化功能了。**

```
CArchive ar;
CRegisterreg1,reg2;
ar<<reg1<<reg2;
```

读者请试着在第五章职工调查表程序基础上，增加保存调查信息到文件以及从文件中读入调查表信息功能。对于多个调查表，可考虑采用 CObjList **链表保存多个对象的指针。**

串行化简化了对象的保存和载入，为对象提供了持续性。但是，串行化本身还是具有一定的局限性的。串行化一次从文件中载入所有对象，这不适合于大文件编辑器和数据库。对于数据库和大文件编辑器，它们每次只是从文件中读入一部分。此时，就要避开文档的串行化机制来直接读取和保存文件了。另外，使用外部文件格式(预先定义的文件格式而不是本应用程序定义的文件格式)的程序一般也不使用文档的串行化。下面我们就给出这样一个例子，说明在不使用串行化情况下如何读取和保存文件。

7.5 不使用串行化的文档视结构程序

在 MFC 例子中有一个 DIBLOOK(见 SAMPLES\MFC\GENERAL\DIBLOOK 目录)，它是一个位图显示程序，演示了在不使用串行化的情况下实现文档的输入输出功能。有关位图、调色板的使用在第十一章有详细介绍，这里只讨论与文档视结构相关的内容。我们先看 DIBLOOK 的文档声明和定义。

清单 7-16 CDibDoc 的类声明文件

```
// dibdoc.h : interface of the CDibDoc class
#include “dibapi.h”
class CDibDoc : public CDocument
{
protected: // create from serialization only
CDibDoc();
DECLARE_DYNCREATE(CDibDoc)
// Attributes
public:
HDIB GetHDIB() const
{ return m_hDIB; }
CPalette* GetDocPalette() const
{ return m_palDIB; }
CSize GetDocSize() const
{ return m_sizeDoc; }
// Operations
public:
void ReplaceHDIB(HDIB hDIB);
void InitDIBData();
// Implementation
protected:
virtual ~CDibDoc();
virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
protected:
HDIB m_hDIB;
CPalette* m_palDIB;
CSize m_sizeDoc;

#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
```

```
virtual BOOL OnNewDocument();
// Generated message map functions
protected:
//{{AFX_MSG(CDibDoc)
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
```

清单 7-17 CDibDoc **类的实现文件**

```
// dibdoc.cpp : implementation of the CDibDoc class
#include “stdafx.h”
#include “diblook.h”
#include <limits.h>
#include “dibdoc.h”
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CDibDoc
IMPLEMENT_DYNCREATE(CDibDoc, CDocument)
BEGIN_MESSAGE_MAP(CDibDoc, CDocument)
//{{AFX_MSG_MAP(CDibDoc)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CDibDoc construction/destruction
CDibDoc::CDibDoc()
{
```

//**初始化文档的** DIB **句柄和调色板**

```
m_hDIB = NULL;
m_palDIB = NULL;
m_sizeDoc = CSize(1,1); // dummy value to make CScrollView happy
}
CDibDoc::~CDibDoc()
{
if (m_hDIB != NULL)
{
::GlobalFree((HGLOBAL) m_hDIB);
}
if (m_palDIB != NULL)
{
```

```
delete m_palDIB;
}
}
BOOL CDibDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
return TRUE;
}
void CDibDoc::InitDIBData()
{
if (m_palDIB != NULL)
{
delete m_palDIB;
m_palDIB = NULL;
}
if (m_hDIB == NULL)
{
return;
}
// Set up document size
LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);
if (::DIBWidth(lpDIB) > INT_MAX ||::DIBHeight(lpDIB) >
INT_MAX)
{
::GlobalUnlock((HGLOBAL) m_hDIB);
::GlobalFree((HGLOBAL) m_hDIB);
m_hDIB = NULL;
CString strMsg;
strMsg.LoadString(IDS_DIB_TOO_BIG);
MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION |
MB_OK);
return;
}
m_sizeDoc = CSize((int) ::DIBWidth(lpDIB),
(int) ::DIBHeight(lpDIB));
::GlobalUnlock((HGLOBAL) m_hDIB);
// Create copy of palette
m_palDIB = new CPalette;
if (m_palDIB == NULL)
{
// we must be really low on memory
```

```
::GlobalFree((HGLOBAL) m_hDIB);
m_hDIB = NULL;
return;
}
if (::CreateDIBPalette(m_hDIB, m_palDIB) == NULL)
{
// DIB may not have a palette
delete m_palDIB;
m_palDIB = NULL;
return;
}
}
BOOL CDibDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
CFile file;
CFileException fe;
if (!file.Open(lpszPathName, CFile::modeRead | CFile::shareDenyWrite,
&fe))
{
ReportSaveLoadException(lpszPathName, &fe,
FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
return FALSE;
}
DeleteContents();
BeginWaitCursor();
// replace calls to Serialize with ReadDIBFile function
TRY
{
m_hDIB = ::ReadDIBFile(file);
}
CATCH (CFileException, eLoad)
{
file.Abort(); // will not throw an exception
EndWaitCursor();
ReportSaveLoadException(lpszPathName, eLoad,
FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
m_hDIB = NULL;
return FALSE;
}
END_CATCH
InitDIBData();
EndWaitCursor();
```

```
if (m_hDIB == NULL)
{
// may not be DIB format
CString strMsg;
strMsg.LoadString(IDS_CANNOT_LOAD_DIB);
MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION |
MB_OK);
return FALSE;
}
SetPathName(lpszPathName);
SetModifiedFlag(FALSE); // start off with unmodified
return TRUE;
}
BOOL CDibDoc::OnSaveDocument(LPCTSTR lpszPathName)
{
CFile file;
CFileException fe;
if (!file.Open(lpszPathName, CFile::modeCreate |
CFile::modeReadWrite | CFile::shareExclusive, &fe))
{
ReportSaveLoadException(lpszPathName, &fe,
TRUE, AFX_IDP_INVALID_FILENAME);
return FALSE;
}
// replace calls to Serialize with SaveDIB function
BOOL bSuccess = FALSE;
TRY
{
BeginWaitCursor();
bSuccess = ::SaveDIB(m_hDIB, file);
file.Close();
}
CATCH (CException, eSave)
{
file.Abort(); // will not throw an exception
EndWaitCursor();
ReportSaveLoadException(lpszPathName, eSave,
TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
return FALSE;
}
END_CATCH
EndWaitCursor();
```

```
SetModifiedFlag(FALSE); // back to unmodified
if (!bSuccess)
{
// may be other-style DIB (load supported but not save)
// or other problem in SaveDIB
CString strMsg;
strMsg.LoadString(IDS_CANNOT_SAVE_DIB);
MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION |
MB_OK);
}
return bSuccess;
}
void CDibDoc::ReplaceHDIB(HDIB hDIB)
{
if (m_hDIB != NULL)
{
::GlobalFree((HGLOBAL) m_hDIB);
}
m_hDIB = hDIB;
}
/////////////////////////////////////////////////////////////////////////////
// CDibDoc diagnostics
#ifdef _DEBUG
void CDibDoc::AssertValid() const
{
CDocument::AssertValid();
}
void CDibDoc::Dump(CDumpContext& dc) const
{
CDocument::Dump(dc);
}
#endif //_DEBUG
/////////////////////////////////////////////////////////////////////////////
// CDibDoc commands
```

DIBLOOK **读入和保存标准的** Windows **设备无关位图。在内存中，位图以一个** HDIB **句柄表示。**DIBLOOK **没有重载** CDocument::Serialize() **函数，而是重载了** CDocument::OnOpenDocument **和** CDocument::OnSaveDocument **函数。这两个函数使用框架传过来得文件路径名** pszPathName**，打开一个文件对象，读入或保存** DIB **数据。这就是说，**DIBLOOK **把本来在** Serialize()**中完成的对象保存和载入两个任务分别交与** OnSaveDocument()**函数和** OnOpenDocument()**函数去完成。如果读者希望绕过文档的串行化提供文档数据的保存和载入，也只需要重载**

这两个成员函数：OnOpenDocument()和 OnSaveDocument()，通过文件路径参数打开文件，从中读取应用程序数据或向文件里写入应用程序数据。

在 OnOpenDocument()中，还必需自己调用 DeleteContents()清除原来文档的数据，并调用 SetModifiedFlag(FALSE)。在 OnSaveDocument()中也要调用 SetModifiedFlag(FALSE)将文档修改标志改为 FALSE。

在 OnOpenDocument()函数开始处(见清单 7.18)，有一些地方需要解释一下。

清单 7.18 OnOpenDocument()函数

```
BOOL CDibDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
CFile file;
CFileException fe;
if (!file.Open(lpszPathName, CFile::modeRead | CFile::shareDenyWrite,
&fe))
{
ReportSaveLoadException(lpszPathName, &fe,
FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
return FALSE;
}
.....
}
```

7.5.1 文件操作

文件读写

OnOpenDocument 首先声明一个 CFile 类的对象。CFile 是 MFC 提供的一个类，它提供了访问二进制文件的接口。可以使用带参数的 CFile 构造函数创建对象，在构造函数中指定了文件名和打开文件的模式，这样在对象创建的同时也就打开了这个文件；也可以象本例那样使用不带参数的 CFile 构造函数构造一个 CFile 对象，然后调用 CFile::Open()打开一个文件。

BOOLCFile::Open(LPCTSTRlpszFileName,UINTnOpenFlags, CFileException*pError=NULL);

CFile::Open 成员函数带三个参数，第一个参数指定了要打开的文件的完整路径名，如“ c:\hello\hello.cpp ”;第二个参数指定打开文件的模式。

常见的文件打开模式有以下几种：

CFile::modeCreate：创建一个新文件，如果该文件已经存在，则把该文件长度置为零

CFile::modeNoTruncate：与 modeCreate 一起使用。告诉 CFile，如果要创建的文件已经存在，则不再将文件长度设置为零。这对于系统设置文件、日志文件等特别有用，因为第一次启动系统时，这些文件通常不存在，需要创建，而下次则只需要修改文件。

CFile::modeRead：打开文件用于读

CFile::modeWrite：打开文件用于写

CFile::modeReadWrite：打开文件且对文件可读可写

可以使用比特位或“|”对上述操作进行组合。比如，要打开文件写，可以用以下方式打开：

```
CFile file;
file.Open(“c:\\readme.txt”,CFile::modeCreate|CFile::modeWrite);
```

读文件

既然已经打开了文件，就可以对文件进行读写操作了。要读取文件内容到内存，可以调用 CFile::Read()。CFile::Read()函数原型如下：

UINTRead(void* lpBuf,UINTnCount);Read 函数包含两个参数，第一个参数是一个缓冲区指针，该缓冲区用于存放从文件读进来的内容。第二个参数是要读取的字节数。Read 函数返回实际读入的字节数。例如：

```
CFile file;
char buf[100];
int nBytesRead;
nBytesRead=file.Read(buf,100);
```

写文件

写文件与读文件操作方式类似，通过调用 CFile::Write 函数来完成。

```
void Write( const void* lpBuf, UINT nCount );
```

Write 函数第一个参数是指向要写入到文件中的缓冲区的指针，第二个参数是要写入到文件中的字节数。例如：

```
CFile file;
CString str(“This is a string.”);
file.Write(str,str.GetLength());
```

关闭文件

在完成文件读写操作后，要调用 CFile::Close 成员函数及时将文件关闭。

```
CFile file;
//一些读写操作.....
file.Close();
```

7.5.2 异常处理

在打开和保存文件时，我们并未作传统的错误检查，而是采用一种异常机制来处理错误。

异常处理为异常事件提供了结构化、规范化的服务。它一般是指处理错误状态。

我们先回顾一下传统的错误处理方式。传统的错误处理方式通常有两种：

1.返回错误值

2.使用 goto,setjmp/longjmp 协助报告错误

对于第一种技术，要求程序员记住各种错误代码，并且加入大量的检查情况。由于大多数错误是很少会发生，这样处理的结果是代码冗余性很大，效率不高。

第二种技术不但使程序可读性降低，更严重的是，使得函数里的对

象不能释放、删除。比如：

```
void SomeOperation()
{
CMyClass obj1;
if(error)goto errHandler;
...
}
...
errHandler:
//handler error
```

在上面的程序片断中，由于 goto **跳转，无法调用** obj **的析构函数在退出** SomeOperation()**函数时释放其所占的内存，造成内存泄漏。**

而且，以上两种错误处理方法都无法考虑到不可预见的错误。C++ **引入异常处理这一重要概念很好的解决了上述问题。异常处理在处理异常事件时会自动调用已经超出范围的局部对象的析构函数，这样就可以防止内存泄漏。**

下面是 OnSaveDocument()**函数中的异常处理代码：**

```
CFile file;
CFileException fe;
if (!file.Open(lpszPathName, CFile::modeCreate |
CFile::modeReadWrite | CFile::shareExclusive, &fe))
{
ReportSaveLoadException(lpszPathName, &fe,
TRUE, AFX_IDP_INVALID_FILENAME);
return FALSE;
}
// replace calls to Serialize with SaveDIB function
BOOL bSuccess = FALSE;
TRY
{
BeginWaitCursor();
bSuccess = ::SaveDIB(m_hDIB, file);
file.Close();
}
CATCH (CException, eSave)
{
file.Abort(); // will not throw an exception
EndWaitCursor();
ReportSaveLoadException(lpszPathName, eSave,
TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
return FALSE;
}
```

END_CATCH

异常处理由一个 TRY-CATCH-END_CATCH **结构组成。**TRY{ }**语句块中包含可能发生错误的代码，可以理解为“ 试运行 ”这一语句块。**CATCH{} END_CATCH **子块包含了错误处理代码。如果发生错误，就转入** CATCH{} END_CATCH **子块执行。该子块可以根据** CATCH **中的参数分析产生错误的原因，报告错误或做出相应处理。**

CATCH()**包含两个参数，第一个参数是异常类。**MFC **的异常有下列几种：**

MFC **异常类**	**处理的异常**
CMemoryException	**内存异常**
CNotSupportedException	**设备不支持**
CarchiveException	**档案**(archive)**异常**
CFileException	**文件异常**
OsErrorException	**把** DOS **错误转换为异常**
ErrnoToException	**把错误号转换为异常**
CResourceException	**资源异常**
COleException	OLE **异常**

用户还可以从 CException **类派生出自己的异常类，用以处理特定类型的错误。**

CATCH **的第二个参数是产生的异常的名字。**

引起异常的原因存放在异常的数据成员 m_cause **中。**OnSaveDocument()**只是简单的处理文件保存错误，并没有指出引起错误的原因。我们可以对它进行一些修改，使它能够报告引起错误的原因。**

```
...
TRY
{
...
}
CATCH(CFileException,e)
{
switch(e->m_cause)
{
case CFileException::accessDenied:
AfxMessageBox(“ Access denied! ”);
break;
case CFileException::badPath:
AfxMessageBox(“ Invalid path name ”);
break;
case CFileException::diskFull:
AfxMessageBox(“ Disk is full ”);
break;
case CFileException::hardIO:
```

```
AfxMessageBox(“Hardware error”);
break;
}
}
END_CATCH
...
}
```

用户也可以不必直接处理异常，而通过调用 THROW_LAST()**，把异常交给上一级** TRY-CATCH **结构来处理。其实，在** DIBLOOK **中，就是这么做的，请看** OnSaveDocument()**函数调用的** SaveDIB **函数的片段：**

```
BOOL WINAPI SaveDIB(HDIB hDib, CFile& file)
{
//...
TRY
{
//...
}
CATCH (CFileException, e)
{
//...
::GlobalUnlock((HGLOBAL) hDib);
THROW_LAST();
}
END_CATCH
//...
}
```

在 SaveDIB **中，并没有直接处理异常，而是通过调用** THROW_LAST()**，把异常交由调用它的上一级函数** OnSaveDocument() **去处理。**

异常并不仅仅用于错误处理。比如，在文本编辑器的 CEditorDoc::Serialize()**成员函数中，我们就利用读取文件引起的异常判断是否已经到了文件尾部。读者请回顾一下该函数。**

异常处理给程序的错误处理带来许多便利。但是，必需意识到异常处理并不是万能的。在加入异常处理后，程序员仍然有许多工作要做。更不可以滥用异常，因为异常会带来一些开销。应用程序应当尽可能排除可能出现的错误。

小结

本章主要讲述了以下内容：

文档、视图的基本概念：文档是数据源，它构成应用程序的数据，另外，它还提供存储和管理数据的手段。视图为用户提供了数据的可视显示，还提供了操作数据的界面。

两种类型的文档视结构程序：单文档应用程序和多文档应用程序

用 AppWizard 生成基于文档视图结构的文本编辑器程序

应用程序类、文档类、视图类、文档模板、框架窗口之间的相互关系

设计文档类：初始化、清理、串行化、在文档类中访问视图

设计视图类：初始化、绘制、消息处理、文档和视图类数据成员的合理分配、在视图类中修改文档内容

滚动视图：逻辑坐标系和设备坐标系，CScrollView 类，设置卷滚范围；

不使用串行化的文档视图结构程序设计：重载 OnNewDocument()和 OnOpenDocument 成员函数

文件处理：打开文件、读写文件

异常处理

第八课　多文档界面(MDI)

这一讲主要介绍多文档界面的窗口管理、多文档编程技术、分割视图、多个文档类型程序设计。在介绍多文档编程时，还结合绘图程序介绍了 GDI(图形设备接口)的使用。

多文档界面窗口

图形设备接口(GDI)

绘图程序

访问当前活动视图和活动文档

分隔视图

打印和打印预览

支持多个文档类型的文档视结构程序

防止应用程序运行时创建空白窗口

小结

8.1 多文档界面窗口

MDI 应用程序是另一类重要的文档视结构程序。它的特点是：用户一次可以打开多个文档，每个文档对应不同的窗口；主窗口的菜单会自动随着当前活动的子窗口的变化而变化；可以对子窗口进行层叠、平铺等各种操作；子窗口可以在 MDI 主窗口区域内定位、改变大小、最大化和最小化，当最大化子窗口时，它将占满 MDI 主窗口的全部客户区。MDI 不仅可以在同一时间内同时打开多个文档，还可以为同一文档打开多个视图。在 Windows 菜单下选择 New ，就为当前活动文档打开一个新的子窗口。

从程序员角度看，每个 MDI 应用程序必须有一个 CMDIFrameWnd 或其派生类的实例，这个窗口称作 MDI 框架窗口。CMDIFrameWnd 是 CFrameWnd 的派生类，它除了拥有 CFrameWnd 框架窗口类的全部特性外，还具有以下与 MDI 相关的特性：

与 SDI 不同，主框架窗口并不直接与一个文档和视图相关联。MDI 框架窗口拥有 MDICLIENT (MDI 客户窗口)，在显示或隐藏控制条(包括工具条、状态栏、对话条)时，重新定位该子窗口。

MDI 客户窗口是 MDI 子窗口的直接父窗口，它负责管理主框架窗口的客户区以及创建子窗口。每个 MDI 主框架窗口都有且只有一个 MDI 客户窗口。

MDI 主框架窗口、客户窗口和子窗口的关系如下图所示：

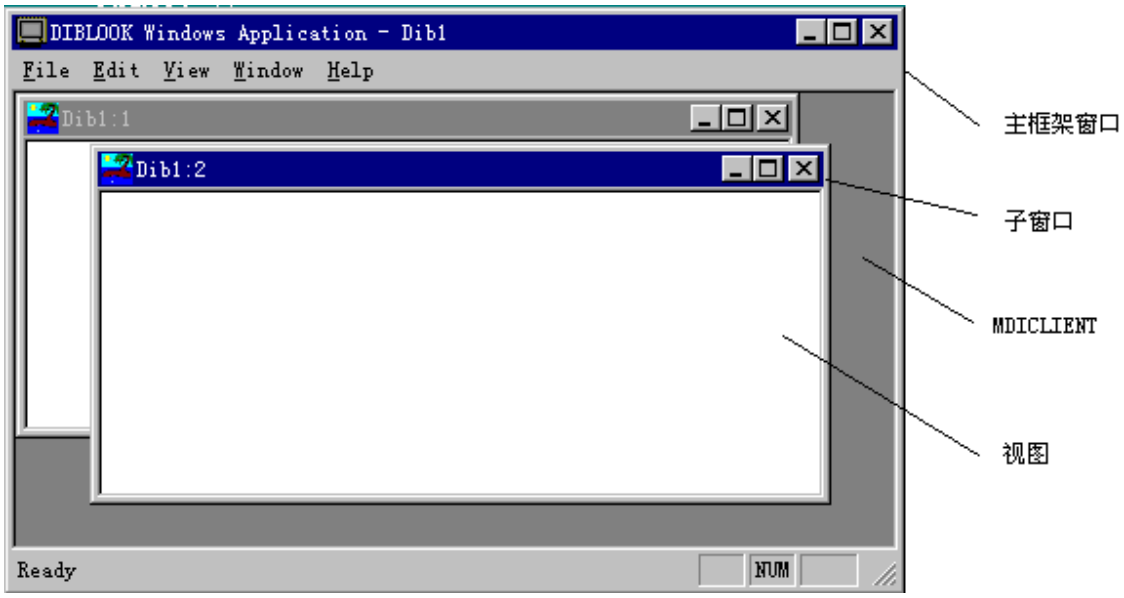


图 8-1 MDI 主框架窗口、客户窗口和子窗口

MDI 子窗口是 CMDIChildWnd 或其派生类的实例，CMDIChildWnd 是 CFrameWnd 的派生类，用于容纳视图和文档，它相当于 SDI 下的主框架窗口。每打开一个文档，框架就自动为文档创建一个 MDI 子窗口。一个 MDI 应用程序负责动态的创建和删除 MDI 子窗口。在任何时刻，最多只有一个子窗口是活动的(窗口标题栏颜色呈高亮显示) 。MDI 框架窗口始终与当前活动子窗口相关联，命令消息在传给 MDI 框架窗口之

前首先分派给当前活动子窗口。

在没有任何活动的 MDI 子窗口时，MDI 框架窗口可以拥有自己的缺省菜单。当有活动子窗口时，MDI 框架窗口的菜单条会自动被子窗口的菜单所替代。框架会自动监视当前活动的子窗口类型，并相应的改变主窗口的菜单。比如，在 Visual Studio 中，当选择对话框资源编辑窗口或源程序窗口时，菜单会有所不同。但是，对于程序员来说，只需要在 InitInstance 中注册文档时指定每一类子窗口(严格的讲是文档)所使用的菜单，而不必显式的通过调用函数去改变主框架窗口的菜单，因为框架会自动完成这一任务。

MDI 框架窗口为层叠、平铺、排列子窗口和新建子窗口等一些标准窗口操作提供了缺省的菜单响应。在响应新建子窗口命令时，框架调用 CDocTemplate::CreateNewFrame() 为当前活动文档创建一个子窗口。CreateNewFrame() 不仅创建子窗口，还创建与文档相对应的视图。

下面，我们结合一个绘图程序例子，介绍多文档界面技术。在此之前，我们首先要介绍一下如何在 Windows 中绘图以及 Windows 的图形设备接口(GDI)。

8.2 图形设备接口(GDI)

图形设备接口(GDI)是一个可执行程序，它接受 Windows 应用程序的绘图请求(表现为 GDI 函数调用)，并将它们传给相应的设备驱动程序，完成特定于硬件的输出，象打印机输出和屏幕输出。

GDI 负责 Windows 的所有图形输出，包括屏幕上输出像素、在打印机上输出硬拷贝以及绘制 Windows 用户界面。

8.2.1 三种图形输出类型

应用程序可以使用 GDI 创建三种类型的图形输出：矢量输出、光栅图形输出和文本输出。

矢量图形输出

矢量图形输出指的是创建线条和填充图形，包括点、直线、曲线、多边形、扇形和矩形的绘制。

光栅输出

光栅图形的输出是指光栅图形函数对以位图形式存储的数据进行操作，它包括各种位图和图标的输出。在屏幕上表现为对若干行和列的像素的操作，在打印机上则是若干行和列的点阵的输出。

光栅图形输出的优点是速度很快，它是直接从内存到显存的拷贝操作。缺点是需要额外的内存空间。Windows 在绘制界面时使用了大量的光栅输出。

文本输出

与 DOS 字符方式下的输出不同，Windows 是按图形方式输出的。这样，在输出文本时，必须以逻辑坐标为单位计算文本的输出位置，而不是象 DOS 下以文本行为单位输出文本。这比 DOS 下的文本输出要难一些。

但是，按图形方式输出文本也给文本输出带来很大的灵活性。用户可以通过调用各种 GDI 函数，制造出各种文本输出效果，包括加粗、斜体、设置颜色等。

Windows 还提供了一种 TrueType(写真字体)。TrueType 字体用一组直线和曲线命令及一些参数来描述字体的轮廓。Windows 可以通过参数来调整直线的长度和曲线的形状，从而实现对字体的自由缩放。

8.2.2 MFC 中与 GDI 有关的类

为了支持 GDI 绘图，MFC 提供了两种重要的类：设备上下文类，用于设置绘图属性和绘制图形；绘图对象类，封装了各种 GDI 绘图对象，包括画笔、刷子、字体、位图、调色板和区域。

设备上下文类

设备上下文类包括 CDC 和它的派生类 CClientDC、CPaintDC、CWindowDC、CMetaFileDC。

CDC 是设备上下文类的基类，除了一般的窗口显示外，还用于基于桌面的全屏幕绘制和非屏幕显示的打印机输出。CDC 类封装了所有图形输出函数，包括矢量、光栅和文本输出。

CClientDC(客户区设备上下文)用于客户区的输出，它在构造函数中

封装了 GetDC()，在析构函数中封装了 ReleaseDC()函数。一般在响应非窗口重画消息(如键盘输入时绘制文本、鼠标绘图)绘图时要用到它。用法是：

CClientDC dc(this);//this 一般指向本窗口或当前活动视图

dc.TextOut(10,10,str,str.GetLength());

//利用 dc 输出文本，如果是在 CScrollView 中使用，还要注意调

//用 OnPrepareDC(&dc)调整设备上下文的坐标。

CPaintDC 用于响应窗口重绘消息(WM_PAINT)是的绘图输出。CPaintDC 在构造函数中调用 BeginPaint()取得设备上下文，在析构函数中调用 EndPaint()释放设备上下文。EndPaint()除了释放设备上下文外，还负责从消息队列中清除 WM_PAINT 消息。因此，在处理窗口重画时，必须使用 CPaintDC，否则 WM_PAINT 消息无法从消息队列中清除，将引起不断的窗口重画。CPaintDC 也只能用在 WM_PAINT 消息处理之中。

CWindowDC 用于窗口客户区和非客户区(包括窗口边框、标题栏、控制按钮等)的绘制。除非要自己绘制窗口边框和按钮(如一些 CD 播放程序等)，否则一般不用它。

CMetaFileDC 专门用于图元文件的绘制。图元文件记录一组 GDI 命令，可以通过这一组 GDI 命令重建图形输出。使用 CMetaFileDC 时，所有的图形输出命令会自动记录到一个与 CMetaFileDC 相关的图元文件中。

图形对象类

图形对象类包括 CGdiObject、画笔、刷子、字体、位图、调色板、区域等。CGdiObject 是图形对象类的基类，但该类不能直接为应用程序所使用。要使用 GDI 对象，必须使用它的派生类：画笔、刷子、字体、位图、区域等等。

使用图形对象要注意两点：

1.同其他 MFC 对象一样，GDI 对象的创建也要分为两步：第一步，是定义一个 GDI 绘图对象类的实例；第二步调用该对象的创建方法真正创建对象。

2.创建对象：使用该对象，首先要调用 CDC::SelectObject()将它选入到设备上下文中，同时保存原来的设置到一个 GDI 对象指针比如说 pOldObject 中。在使用完后，再用 SelectObject(pOldObject)恢复原来的设置。但是，如果该设备上下文是用户自己创建的，则不必恢复原来设置，因为框架会在该设备上下文生存期结束时删除该设备上下文，同时也就删除了原来存放于该设备上下文中的绘图对象设置。

下面介绍各种对象的用法：

画笔(CPen)：封装 GDI 画笔，可被选中设备上下文中当前所用得笔。画笔用于绘制对象的边框以及直线和曲线。缺省画笔画一条与一个像素等宽的黑色实线。

要使用画笔，首先要定义一个画笔：

CPen pen;

然后创建画笔。创建画笔有两种方法：

一是使用 CPen::CreatePen(int nPenStyle,int nWidth,DWORD crColor) **进行初始化。第一个参数是笔的风格。**nPenStyle **可选值有：**

PS_SOLID **实线**

PS_DOT **虚线**

PS_INSIDEFRAME **在一个封闭形状的框架内画线，若设定的颜色不能在调色板种找到且线宽大于** 1，Windows **会使用一种混色。**

PS_NULL **空的画笔，什么也不画**

第二个参数是线的宽度，按逻辑单位。若线宽设为 0，**则不管是什么映射模式下，线宽始终为一个像素。第三个参数是线的颜色，可以选** 16 **种** VGA **颜色中的一种。颜色的设置用一个** RGB **宏来指定。**

RGB **宏形式如下**

COLORREF RGB(cRed,cGreen,cBlue)

cRed、cGreen、cBlue **分别代表颜色的** RGB **三个分量，它们的取值在** 0-255 **之间。可以使用** RGB **组合成各种色彩。但是，这种表示法并不是很直观,因此我们把常见的**RGB **组合定义成新的宏并放在一个** colors.h **中，如清单** 8.1。

清单 8.1 **常见色彩定义**

```
/*COLORS.H -常见色彩定义 */
#ifndef _COLORS_H
#define _COLORS_H
//Main Colors
#define WHITE RGB(255,255,255)
#define BLACK RGB(0,0,0)
#define DK_GRAY RGB(128,128,128)
#define LT_GRAY RGB(192,192,192)
//dark colors
#define DK_RED RGB(128,0,0)
#define DK_GREEN RGB(0,128,0)
#define DK_BLUE RGB(0,0,128)
#define DK_PURPLE RGB(128,0,128)
#define DK_YELLOW RGB(128,128,0)
#define DK_CYAN RGB(0,128,128)
//bright colors
#define BR_RED RGB(255,0,0)
#define BR_GREEN RGB(0,255,0)
#define BR_BLUE RGB(0,0,255)
#define BR_PURPLE RGB(255,0,255)
#define BR_YELLOW RGB(255,255,0)
#define BR_CYAN RGB(0,255,255)
#endif
```

这样，要指定一个淡黄色的宽度为逻辑单位 1 **的实心笔，可以调用：**

```
pen.CreatePen(PS_SOLID,1,BR_YELLOW);
```

创建笔的另一个方法是使用库存对象。SelectStockObject 可从以下库存笔中选择一个：

BLACK_PEN 黑笔

NULL_PEN 空笔(不画线或边框)

WHITE_PEN 白笔

注：库存对象是由操作系统维护的用于绘制屏幕的常用对象，包括库存画笔、刷子、字体等。/

刷子(CBrush)：封装 GDI 刷子，可用作设备上下文中当前刷子。刷子用来填充一个封闭图形对象(如矩形、椭圆)的内部区域。缺省的刷子将封闭图形的内部填充成全白色。我们以前所创建的窗口内部都是白色就是窗口使用缺省刷子填充的结果。

可以用以下几种方法创建刷子：

(1)CreateSolidBrush(DWORD crColor)创建一个实心刷子，用一种颜色填充一个内部区域。

(2)CreateHatchBrush(int nIndex,DWORD crColor) ;创建一个带阴影的刷子，nIndex 代表一种影线模式：

图 8-2 刷子的各种影线效果

(3)用 CreatePatternBrush(CBitmap* pBitmap)

用一个位图作刷子，一般采用 8X8 的位图，因为刷子可以看作 8X8 的小位图。当 Windows 桌面背景采用图案(如 weave)填充时，使用的就是这种位图刷子。

(4)同样可以使用 SelectStockObject()从库存刷子中选取一个：

BLACK_BRUSH 黑色刷子

WHITE_BRUSH 白色刷子

DKGRAY_BRUSH 暗灰刷子

GRAY_BRUSH 灰色刷子

LTGRAY_BRUSH 淡灰色刷子

NULL_BRUSH 空刷子，内部不填充

字体(CFont)：封装了 GDI 字体对象，用户可以建立一种 GDI 字体，并使用 CFont 的成员函数来访问它。关于 CFont 类，我们在前面已经作了一些介绍，这里不再赘述，读者可以参见前一章内容。

位图(CBitmap) :封装一 GDI 位图，它提供成员函数装载和操作位图。

调色板(CPalette) :封装 GDI 调色板，它保存着系统可用的色彩信息，是应用程序和彩色输出设备上下文的接口。

区域 CRgn 类：封装 GDI 区域。区域是窗口内的一块多边形或椭圆形的区域。CRgn 用于设备环境(通常是窗口)内的区域操作。CRgn 通常与 CDC 的有关剪裁(clipping)的成员函数配合使用。

有关位图和调色板的使用在第十一章“多媒体编程”还要再作详细阐述。

8.2.3 常见的绘图任务

输出文本

字体大小计算：通过调用 GetTextMetrics()返回当前使用字体的尺寸

描述，如前面文 本编辑所演示的那样。

字体颜色设置：

设置前景色：CDC::SetTextColor(int nColor)；

设置背景色：CDC::SetBkColor(int nColor)；

例如：

dc.SetTextColor(WHITE);

dc.SetBkColor(DK_BLUE);

dc.TextOut(10,10,“White Text on blue background”,30);

文字输出：

除了我们前面介绍的文本输出函数 TextOut()之外，还有其他几个函数可用于文本输 出：

TabbedTextOut：象 TextOut 一样显示正文，但用指定的制表间隔扩充制表键 Tab。 在前面的文本编辑器中，当输入一个 Tab 时，TextOut 在屏幕上输出一个黑色方块。

ExtTextOut：在指定的矩形中显示正文。可以用该函数删去超出矩形的正文，用正 文背景填充矩形，调整字符间隔。

DrawText 在指定矩形种显示正文，可以用这个函数扩展制表键 Tab。在格式化矩 形时调整正文左对齐、右对齐或居中；还可以在一个词中断开以适应矩形边界。

画点

SetPixel 在指定坐标处按指定色彩画一点。

画线

MoveToEx 将直线起点移动到指定坐标处，LineTo 从起点开始画直线到终点处。使 用的线型由当前所用画笔指定。

画弧

Arc(int x1,int y1,int x2,int y2,int x3,int y3,int x4,int y4);

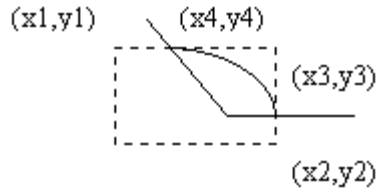


图 8-3 弧线的坐标定位

封闭图形

矩形 Rectangle

圆角矩形 RoundRect()

Ellipse 在一个矩形内画椭圆

Chord 弦形图

Pie 画饼形图

Polygon 生成封闭的多边形

PolyPolygon 画完整的一组多边形

其它常用的绘图函数还有：

FillRect：用指定颜色填充矩形且不画边线

Draw3dRect:**这是一个非常实用的函数，用于绘制各种** 3D **边框。它的函数原型如下：**

void Draw3dRect(LPCRECTlpRect, COLORREFclrTopLeft, COLORREFclrBottomRight);

void Draw3dRect(intx, inty, intcx, intcy, COLORREFclrTopLeft, COLORREFclrBottomRight);

通过设置上下边框的颜色 clrTopLeft **和** clrBottomRight**，可以绘制出凸出或 凹陷等各种效果的** 3D **边框。**

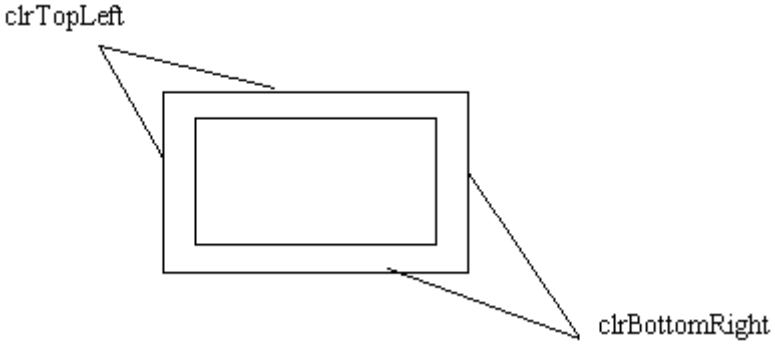


图 8-4 Draw3dRect **绘制** 3D **边框**

DrawFocusRect：**用点线画一个矩形框，内部不填充，边线是用于屏幕上当前色的相反色画出来的，故第二次画时，会擦除原来所画的线。**

ExtFloodFill：**用给定的颜色，利用当前刷子填充表面被一个边线包围的区域，用户可以有选择地填充一个由指定颜色组成的区域。**

FloodFill:**用给定的颜色，利用当前所选的刷子填充显示的底面被一个边线所包围的区域，如多边形区域的填充。**

FrameRect:**绘制矩形边框时内部不填充。**

InvertRect:**在某一矩形区域内反显现有颜色。**

8.3 绘图程序

在了解 GDI 的一些基本知识之后,我们就可以着手编写绘图程序了。这个绘图程序可以让读者用鼠标器在窗口内任意涂写，并可以保存所画的内容。这里我们参考了 Visual C++的例子 Scribble，并作了一些修改和简化。

8.3.1 MDI 应用程序框架

首先用 AppWizard 生成绘图程序的基本框架：

选择 File->New，弹出 New 对话框，选择 MFC AppWizard(exe)，并指定项目文件名为 Draw。

在 MFC AppWizard-Step1 对话框中指定框架类型为 Multiple Document(多文档，这是缺省设置)。

Step2,3 按缺省值。在 MFC AppWizard Step 4 of 6 对话框中，点“Advanced...”按钮，弹出 Advanced Options 对话框。在 File Extension 编辑框中指定文件名后缀为.drw，按 OK 关闭 Advanced Options 对话框。

Step5 按缺省设置。在 MFC AppWizard Step 6 of 6 中，在应用程序所包含的类列表中选择 CDrawView，并为其指定基类为 CScrollView，因为绘图程序需要卷滚文档。现在点 Finish 按钮生成绘图所需的应用程序框架。

在往框架里添加代码实现绘图程序之前，先看看多文档框架与单文档框架的差别。

AppWizard 为多文档框架创建了以下类：

CAboutDlg：“关于”对话框

CChildFrame：子框架窗口，用于容纳视图

CDrawApp：应用程序类

CDrawDoc：绘图程序视图类

CDrawView：绘图视图类

CMainFrame：主框架窗口，用来容纳子窗口，它是多文档应用程序的主窗口。

在生成的类上，MDI 比 SDI 多了一个 CChildFrame 子框架窗口类，而且 CMainFrame 的职责也不同了。

另外，MDI 和 SDI 在初始化应用程序实例上也有所不同。MDI 应用程序 InitInstance 函数如清单 8.2 定义。

清单 8.2 多文档程序的 InitInstance 成员函数定义

```
BOOL CDrawApp::InitInstance()
{
//一些初始化工作.....
// Register the application's document templates. Document templates
// serve as the connection between documents, frame windows and views.
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
```

```
IDR_DRAWTYPE,
RUNTIME_CLASS(CDrawDoc),
RUNTIME_CLASS(CChildFrame), // custom MDI child frame
RUNTIME_CLASS(CDrawView));
AddDocTemplate(pDocTemplate);
// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
return FALSE;
m_pMainWnd = pMainFrame;
// Enable drag/drop open
m_pMainWnd->DragAcceptFiles();
// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
// The main window has been initialized, so show and update it.
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
return TRUE;
}
```

在注册文档模板时，首先创建一个 CMultiDocTemplate **类型(在** SDI **下是** CSingleDocTemplate)**的模板对象，然后用** AddDocTemplate()**把它加入到文档模板链表中去。**

CMultiDocTemplate **构造函数带四个参数，第一个参数是文档使用的资源** ID **定义。第二个是文档类型，第三个是子窗口类型，第四个是视图类型。**

与 SDI **不同，由于** MDI **的主框架窗口并不直接与文档相对应，因此无法通过创建文档来创建主框架窗口，而需要自己去创建。**

```
//定义一个主窗口类指针，并创建一个窗口的空的实例
CMainFrame* pMainFrame = new CMainFrame;
//从资源文件中载入菜单、图标等信息，并创建窗口
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
return FALSE;
//将应用程序对象的主窗口指针数据成员设为当前创建的窗口
m_pMainWnd = pMainFrame;
```

8.3.2 设计绘图程序的文档类

Draw **需要保存用户在屏幕上涂写的每一个笔划。一副画由许多笔划组成，可以把它看作是笔划组成的链表。每一个笔划可以看作一个对象，它由许多点组成。这样，我们可以把绘图文档的数据看作是笔划对象** CStroke **组成的链表。另外，我们还需要一些数据成员表示当前画图所使用的画笔和画笔的宽度。**

修改后的文档类声明文件如清单 8-1**：**

清单 8.3 **文档类声明**

```
// DrawDoc.h : interface of the CDrawDoc class
//
/////////////////////////////////////////////////////////////////////////////
#if !defined(AFX_DRAWDOC_H__143330AE_85BC_11D1_9304_444553540000__INCLUDED_)
#define AFX_DRAWDOC_H__143330AE_85BC_11D1_9304_444553540000__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
class CDrawDoc : public CDocument
{
protected: // create from serialization only
CDrawDoc();
DECLARE_DYNCREATE(CDrawDoc)
// Attributes
public:
UINT m_nPenWidth; // current user-selected pen width
CPen m_penCur; // pen created according to
// user-selected pen style (width)
public:
CTypedPtrList<CObList,CStroke*> m_strokeList;
//获取当前使用的画笔，为视图所使用
CPen* GetCurrentPen() { return &m_penCur; }
protected:
CSize m_sizeDoc;
public:
CSize GetDocSize() { return m_sizeDoc; }
// Operations
public:
//往链表里增加一个笔划
CStroke* NewStroke();
// Operations
//用于初始化文档
```

```
protected:
void InitDocument();
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDrawDoc)
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CDrawDoc();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
//{{AFX_MSG(CDrawDoc)
// 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()
};
```

这里我们使用 指针链表模板来保存指向每个笔划的指针：

```
CTypedPtrList<CObList,CStroke*> m_strokeList;
```

其中“ <> ”第一个参数表示链表基本类型，第二个参数代表链表中所存放的元素的类型。

为了使用模板，还要修改 stdafx.h**，在其中加入** afxtempl..h **头文件，它包含了使用模板时所需的类型定义和宏：**

```
//.........
#define VC_EXTRALEAN // Exclude rarely-used stuff from Windows headers
#include <afxwin.h> // MFC core and standard components
#include <afxext.h> // MFC extensions
#include <afxtempl.h> // MFC templates
#include <afxdisp.h> // MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h> // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
//......
```

由于绘图程序需要卷滚文档，因此象前面的编辑那样，增加一个m_sizeDoc数据成员存放文档的大小。另外,还需要提供一个GetDocSize()来访问它。NewStroke()用于往链表里增加一个笔划。

现在，开始设计 CStroke 类。笔划可以看作由一系列点组成，这样CStroke 可以用一个点的数组来表示。另外，还需要一些成员函数来访问这个数组。我们还希望笔划能够自己绘制自己，并用串行化机制保存自己的数据。

CStroke 类定义清单如 8.4，我们把它在 CDrawDoc 类定义之前。

清单 8.4 CStroke 类定义

```
class CStroke : public CObject
{
public:
CStroke(UINT nPenWidth);//用笔的宽度构造一个画笔
//用于串行化笔划对象
protected:
CStroke(); //串行化对象所需的不带参数的构造函数

DECLARE_SERIAL(CStroke)
// Attributes
protected:
UINT m_nPenWidth; // one pen width applies to entire stroke
public:
//用数组模板类保存笔划的所有点
CArray<CPoint,CPoint> m_pointArray; // series of connected points
//包围笔划所有的点的一个最小矩形，关于它的作用以后会提到
CRect m_rectBounding; // smallest rect that surrounds all
// of the points in the stroke
// measured in MM_LOENGLISH units
// (0.01 inches, with Y-axis inverted)
public:
CRect& GetBoundingRect() { return m_rectBounding; }
//结束笔划，计算最小矩形
void FinishStroke();
// Operations
public:
//绘制笔划
BOOL DrawStroke(CDC* pDC);
public:
virtual void Serialize(CArchive& ar);
};
```

文档的初始化

文档的初始化在 OnNewDocument()和 OnOpenDocument()中完成。对

于 Draw 程序来说，两者的初始化相同，因此设计一个 InitDocument()函数用于文档初始化：

```
void CDrawDoc::InitDocument()
{
m_nPenWidth=2;
m_nPenCur.CreatePen(PS_SOLID,m_nPenWidth,RGB(0,0,0));
//缺省文档大小设置为 800X900 个逻辑单位
m_sizeDoc = CSize(800,900);
}
```

InitDocument()函数将笔的宽度初值设为 2，然后创建一个画笔对象。该对象在以后绘图是要用到。最后将文档尺寸大小设置为 800X900 个逻辑单位。

然后在 OnNewDocument()和 OnOpenDocument()中调用它：

```
void CDrawDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
InitDocument();
return TRUE;
}
```

AppWizard 并没有生成 OnOpenDocument()的代码，因此要用 ClassWizard 来生成 OnOpenDocument()的框架。生成框架后，在其中加入代码：

```
BOOL CDrawDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
if (!CDocument::OnOpenDocument(lpszPathName))
return FALSE;
// TODO: Add your specialized creation code here
InitDocument();
return TRUE;
}
```

文档的清理

在关闭文档的最后一个子窗口时，框架要求文档清理数据。文档清理在文档类的 DeleteContents()中完成。同样需要用 ClassWizard 生成 DeleteContents 的框架。

```
void CDrawDoc::DeleteContents()
{
// TODO: Add your specialized code here and/or call the base class
while (!m_strokeList.IsEmpty())
{
```

```
delete m_strokeList.RemoveHead();
}
CDocument::DeleteContents();
}
```

DeleteContents()**从头到尾遍里链表中的所有对象指针，并通过指针删除对象，然后用** RemoveHead()**删除该指针。**

文档的串行化

现在设计文档的 Serialize **函数，实现文档数据的保存和载入：**

```
void CDrawDoc::Serialize(CArchive& ar)
{
if (ar.IsStoring())
{
ar << m_sizeDoc;
}
else
{
ar >> m_sizeDoc;
}
m_strokeList.Serialize(ar);
}
```

文档的 Serialize()**函数首先分别保存和载入文档大小，然后调用** m_strokeList **的** Serialize()**方法。**m_strokeList.Serialize()**又会自动调用存放在** m_strokeList **中的每一个元素** CStroke **的串行化方法** CStroke.Serialize() **最终实现文档的串行化即文档所包含的对象的存储和载入。**

在 DrawDoc.cpp **的末尾加上** CStroke::Serialize()**函数的定义：**

```
void CStroke::Serialize(CArchive& ar)
{
if (ar.IsStoring())
{
ar << m_rectBounding;
ar << (WORD)m_nPenWidth;
m_pointArray.Serialize(ar);
}
else
{
ar >> m_rectBounding;
WORD w;
ar >> w;
m_nPenWidth = w;
m_pointArray.Serialize(ar);
}
}
```

CStroke 的 Serialize()依次保存(载入)笔划的矩形边界、线宽度以及点数组。注意 m_nPenWidth 是 UINT 类型的，>>和<<操作符并不支持 UINT 类型但却支持 WORD，因此要作 UINT 和 DWORD 之间的类型转换。点数组的串行化通过调用数组的每个 CPoint 类元素的 Serialize()完成，CPoint 类是 MFC 类，它本身支持串行化。

8.3.3 设计绘图程序的视图类

视图类数据成员

现在着手设计绘图程序的视图类。首先，需要在视图中增加两个数据成员：

```
class CDrawView : public CScrollView
{
protected: // create from serialization only
CDrawView();
DECLARE_DYNCREATE(CDrawView)
// Attributes
public:
CDrawDoc* GetDocument();
protected:
CStroke* m_pStrokeCur; // the stroke in progress
CPoint m_ptPrev; // the last mouse pt in the stroke in progress
// 其它数据成员和成员函数.....
}；
```

m_pStrokeCur 代表正在画的那一个笔划。m_ptPrev 保存鼠标上次移动位置。画图时，LineTo 从这个点到当前鼠标位置画一条直线。

视图初始化

接下去，要初始化视图。由于是卷滚视图，因此要在 OnInitialUpdate()中设置卷滚范围。在用户选择 File->New 菜单或 File->Open 菜单时，框架调用 OnInitialUpdate 函数。

```
void CDrawView::OnInitialUpdate()
{
SetScrollSizes(MM_LOENGLISH, GetDocument()->GetDocSize());
CScrollView::OnInitialUpdate();
}
```

注意我们这里将映射模式设置为 MM_LOENGLISH，MM_LOENGLISH 以 0.01 英寸为逻辑单位，y 轴方向向上递增，同 MM_TEXT 的 y 轴递增方向相反。

视图绘制

在 CDrawView::OnDraw()内完成视图绘制工作。在以前的文档视结构程序中，在需要绘图的时侯都是绘制整个窗口。如果窗口只有很小的一部分被覆盖，是否可以只绘制那些需要重画的部分”

回答是肯定的，而且大部分程序都这么做了。

比如，象下图这种情况：

图 8-5 窗口的重绘

当窗口 2 从窗口 1 上移开后，只需要重画阴影线所包围的区域就够了。

当 Windows 通知窗口要重绘用户区时,并非整个用户区都需要重绘，需要重绘的区域称为“ 无效矩形区 ”，如上图中的阴影区域。用户区中出现一个无效矩形提示 Windows 在应用程序队列中放置 WM_PAINT 消息。由于 WM_PAINT 消息优先级最低，可调用 UpdateWindows 直接立即向窗口发送 WM_PAINT 消息，从而立即重绘。无效矩形区限制程序只能在该区域中绘图，越界的绘图将被裁剪掉。下面三个函数与无效矩形有关：

InvalidateRect 产生一个无效矩形，并生成 WM_PAINT 消息

ValidateRect 使无效矩形区有效

GetUpdateRect 获得无效矩形坐标(逻辑)

Windows 为每个窗口保留一个 PAINTSTRUCT 结构，其中包含无效矩形区域的坐标值。

要想在自己的程序高效绘图、只绘制无效矩形，首先需要重载视图的 OnUpdate 成员函数。

virtualvoidCView::OnUpdate(CView* pSender,LPARAMlHint,CObject* pHint);

当调用文档的 UpdateAllViews 时，框架会自动调用 OnUpdate 函数，也可在视图类中直接调用该函数。OnUpdate 函数一般是这样处理的：访问文档，读取文档的数据，然后对视图的数据成员或控制进行更新，以反映文档的改动。可以用 OnUpdate 函数使视图的某部分无效。以便触发视的 OnDraw，利用文档数据重绘窗口。缺省的 OnUpdate 使窗口整个客户区都无效，在重新设计时，要利用提示信息 lHint 和 pHint 定义一个较小的无效矩形。修改后的 OnUpdate 成员函数如清单 8.5。

清单 8.5 修改后的 OnUpdate 成员函数

```
void CDrawView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
// TODO: Add your specialized code here and/or call the base class
// The document has informed this view that some data has changed.
if (pHint != NULL)
{
if (pHint->IsKindOf(RUNTIME_CLASS(CStroke)))
{
// The hint is that a stroke as been added (or changed).
// So, invalidate its rectangle.
CStroke* pStroke = (CStroke*)pHint;
CClientDC dc(this);
OnPrepareDC(&dc);
CRect rectInvalid = pStroke->GetBoundingRect();
dc.LPtoDP(&rectInvalid);
```

```
InvalidateRect(&rectInvalid);
return;
}
}
// We can't interpret the hint, so assume that anything might
// have been updated.
Invalidate(TRUE);
return;
}
```

这里，传给 pHint **指针的内容是指向需要绘制的笔画对象的指针。采用强制类型转换将它转换为笔划指针，然后取得包围该笔划的最小矩形。**OnPrepareDC **用于调整视图坐标原点。由于** InvalidateRect **需要设备坐标，因此调用** LPToDP(&rectInvalid)**将逻辑坐标转换为设备坐标。最后，调用** InvalidateRect **是窗口部分区域“无效”，也就是视图在收到** WM_PAINT **消息后需要重绘这一区域。**

InvalidateRect **函数原型为：**

```
void InvalidateRect( LPCRECT lpRect, BOOL bErase = TRUE );
```

第一个参数是指向要重绘的矩形的指针，第二个参数告诉视图是否要删除区域内的背景。

这样，当需要重画某一笔划时，只需要重画包围笔划的最小矩形部分就可以了，其他部分就不再重绘。这也是为什么在笔划对象中提供最小矩形信息的原因。

如果 pHint **为空，则表明是一般的重绘，此时需要重绘整个客户区。**

现在，在 OnDraw **中，根据无效矩形绘制图形，而不是重绘全部笔划，见清单** 8.6**。**

清单 8.6 **根据无效矩形绘制图形的** OnDraw **成员函数**

```
void CDrawView::OnDraw(CDC* pDC)
{
CDrawDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// Get the invalidated rectangle of the view, or in the case
// of printing, the clipping region of the printer dc.
CRect rectClip;
CRect rectStroke;
pDC->GetClipBox(&rectClip);
pDC->LPtoDP(&rectClip);
rectClip.InflateRect(1, 1); // avoid rounding to nothing
// Note: CScrollView::OnPaint() will have already adjusted the
// viewport origin before calling OnDraw(), to reflect the
// currently scrolled position.
// The view delegates the drawing of individual strokes to
// CStroke::DrawStroke().
```

```
CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
POSITION pos = strokeList.GetHeadPosition();
while (pos != NULL)
{
CStroke* pStroke = strokeList.GetNext(pos);
rectStroke = pStroke->GetBoundingRect();
pDC->LPtoDP(&rectStroke);
rectStroke.InflateRect(1, 1); // avoid rounding to nothing
if (!rectStroke.IntersectRect(&rectStroke, &rectClip))
continue;
pStroke->DrawStroke(pDC);
}
// TODO: add draw code for native data here
}
```

OnDraw 首先调用 GetClipBox 取得当前被剪裁区域(无效矩形区域)，它把矩形复制导 GetClipBox 的参数 rectClip 中。然后将 rectClip 的坐标由逻辑坐标转换为设备坐标。为了防止该矩形太小而无法包围其他内容，上下各放大一个单位。然后 OnDraw 遍历笔划链表中的所有笔划，获取它们的最小矩形，用 IntersectRect 看它是否与无效矩形相交。如果相交，说明笔划的部分或全部落在无效矩形中，此时调用笔划的 DrawStroke 方法画出该笔划。

图 8-6 根据包围笔划 的矩形是否与无效
矩形相交 ，判断笔划是否落入无效矩形中

为了获得笔划的最小包围矩形，需要在结束笔划时计算出包围笔划的最小矩形。因此为笔划提供两个方法：一个是 FinishStroke()，用于在笔划结束时计算最小矩形，见清单 8.7。

清单 8.7 CStroke::FinishStroke()成员函数

```
void CStroke::FinishStroke()
{
// Calculate the bounding rectangle. It's needed for smart
// repainting.
if (m_pointArray.GetSize()==0)
{
m_rectBounding.SetRectEmpty();
return;
}
CPoint pt = m_pointArray[0];
m_rectBounding = CRect(pt.x, pt.y, pt.x, pt.y);
for (int i=1; i < m_pointArray.GetSize(); i++)
{
// If the point lies outside of the accumulated bounding
// rectangle, then inflate the bounding rect to include it.
```

```
pt = m_pointArray[i];
m_rectBounding.left = min(m_rectBounding.left, pt.x);
m_rectBounding.right = max(m_rectBounding.right, pt.x);
m_rectBounding.top = max(m_rectBounding.top, pt.y);
m_rectBounding.bottom = min(m_rectBounding.bottom, pt.y);
}
// Add the pen width to the bounding rectangle. This is necessary
// to account for the width of the stroke when invalidating
// the screen.
m_rectBounding.InflateRect(CSize(m_nPenWidth,
-(int)m_nPenWidth));
return;
}
```

另一个是 DrawStroke()**，用于绘制笔划：**

```
BOOL CStroke::DrawStroke(CDC* pDC)
{
CPen penStroke;
if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, RGB(0,0,0)))
return FALSE;
CPen* pOldPen = pDC->SelectObject(&penStroke);
pDC->MoveTo(m_pointArray[0]);
for (int i=1; i < m_pointArray.GetSize(); i++)
{
pDC->LineTo(m_pointArray[i]);
}
pDC->SelectObject(pOldPen);
return TRUE;
}
```

鼠标绘图

鼠标绘图基本过程是：用户按下鼠标左键时开始绘图，在鼠标左键按下且移动过程中不断画线跟踪鼠标位置，当松开鼠标左键结束绘图。因此，需要处理三个消息：WM_LBUTTONDOWN、WM_MOUSEMOVE、WM_LBUTTONUP**。用** ClassWizard **为上述三个消息生成消息处理函数，并在其中手工加入代码，修改后的成员函数如下：**

清单 8.8 **鼠标消息处理函数** OnLButtonDown()

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// Pressing the mouse button in the view window starts a new stroke
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
```

```
CClientDC dc(this);
OnPrepareDC(&dc);
dc.DPtoLP(&point);
m_pStrokeCur = GetDocument()->NewStroke();
// Add first point to the new stroke
m_pStrokeCur->m_pointArray.Add(point);
SetCapture(); // Capture the mouse until button up.
m_ptPrev = point; // Serves as the MoveTo() anchor point for the
// LineTo() the next point, as the user drags the
// mouse.
return;
}
```

在鼠标左键按下，首先获得鼠标按下的位置坐标。由于它是设备坐标，因此先用 DPToLP **将它转换为逻辑坐标。在此之前，要用** OnPrepareDC() **对视图坐标原点进行调整。然后用** CDrawDoc **的** NewStroke()**成员函数创建一个笔划对象，并将笔划对象加入到笔划链表中。然后，将当前点坐标加入道笔划对象内部的点数组中。以后，当鼠标移动时，**OnMouseMove **就不断修改该笔划对象的内部数据成员(加入新的点到笔划对象的数组中)。另外，为了用** LineTo **画出线条，需要将当前鼠标位置保存到** m_ptPrev **中，以便出现一个新的点时，画一条从** m_ptPrev **到新的点的直线。**

但是，由于用户的鼠标可以在屏幕上任意移动。当鼠标移出窗口外时，窗口无法收到鼠标消息。此时，如果松开了鼠标左键，应用程序由于无法接受到该条消息而不会终止当前笔划，这样就造成了错误。如何避免这种情况发生呢”解决的办法是要让窗口在鼠标移出窗口外时仍然能接受到鼠标消息。幸好，Windows **提供了一个** API **函数** SetCapture() **解决了这一问题。**

CWnd::SetCapture()**用于捕获鼠标：无论鼠标光标位置在何处，都会将鼠标消息送给调用它的那一个窗口。在用完后，需要用** ReleaseCapture() **释放窗口对鼠标的控制，否则其他窗口将无法接收到鼠标消息。这一工作当然最好在鼠标左键松开** OnLButtonUp()**时来做。**

清单 8.9 OnLButtonUp **消息处理函数**

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// Mouse button up is interesting in the draw application
// only if the user is currently drawing a new stroke by dragging
// the captured mouse.
if (GetCapture() != this)
return; // If this window (view) didn't capture the mouse,
// then the user isn't drawing in this window.
CDrawDoc* pDoc = GetDocument();
```

```
CClientDC dc(this);
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
OnPrepareDC(&dc); // set up mapping mode and viewport origin
dc.DPtoLP(&point);
CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
dc.MoveTo(m_ptPrev);
dc.LineTo(point);
dc.SelectObject(pOldPen);
m_pStrokeCur->m_pointArray.Add(point);
// Tell the stroke item that we're done adding points to it.
// This is so it can finish computing its bounding rectangle.
m_pStrokeCur->FinishStroke();
// Tell the other views that this stroke has been added
// so that they can invalidate this stroke's area in their
// client area.
pDoc->UpdateAllViews(this, 0L, m_pStrokeCur);
ReleaseCapture(); // Release the mouse capture established at
// the beginning of the mouse drag.
return;
}
```

OnLButtonUp **首先检查鼠标是否被当前窗口所捕获，如果不是则返回。然后画出笔划最后两点之间的极短的直线段。接着，调用** CStroke::FinishStroke()**，请求** CStroke **对象计算它的最小矩形。然后调用** pDoc->UpdateAllViews(this, 0L, m_pStrokeCur)**通知其他视图更新显示。**

当一个视图修改了文档内容并更新显示时，一般的其它的对应于同一文档的视图也需要相应更新，这通过调用文档的成员函数 UpdateAllViews **完成。**

void UpdateAllViews(CView* pSender, LPARAM lHint = 0L, CObject* pHint =
NULL);

UpdateAllViews **带三个参数：**pSender **指向修改文档的视图。由于该视图已经作了更新，所以不再需要更新。比如，在上面的例子中，** OnLButtonUp **已经绘制了视图，因此不需要再次更新。如果为** NULL**，则文档对应的所有视图都被更新。**

lHint **和** pHint **包含了更新视图时所需的附加信息。在本例中，其他视图只需要重画当前绘制中的笔划，因此** OnLButtonUp **把当前笔划指针传给** UpdateAllViews **函数。该函数调用文档所对应的除** pSender **外的所有视图的** OnUpdate **函数，并将** lHint **和** pHint **传给** OnUpdate **函数通知更新附加信息。**

OnLButtonUp **最后释放对鼠标的控制，这样别的应用程序窗口就可**

以获得鼠标消息了。

结合上面讲到的知识，读者不难自行理解下面的 OnMouseMove 函数。

```
void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
// Mouse movement is interesting in the Scribble application
// only if the user is currently drawing a new stroke by dragging
// the captured mouse.
if (GetCapture() != this)
return; // If this window (view) didn't capture the mouse,
// then the user isn't drawing in this window.
CClientDC dc(this);
// CScrollView changes the viewport origin and mapping mode.
// It's necessary to convert the point from device coordinates
// to logical coordinates, such as are stored in the document.
OnPrepareDC(&dc);
dc.DPtoLP(&point);
m_pStrokeCur->m_pointArray.Add(point);
// Draw a line from the previous detected point in the mouse
// drag to the current point.
CPen* pOldPen = dc.SelectObject(GetDocument()->GetCurrentPen());
dc.MoveTo(m_ptPrev);
dc.LineTo(point);
dc.SelectObject(pOldPen);
m_ptPrev = point;
return;
}
```

至此，绘图程序的文档、视图全部设计完了，现在编译运行程序。程序启动后，在空白窗口中徒手绘图，如图 8-7 所示。

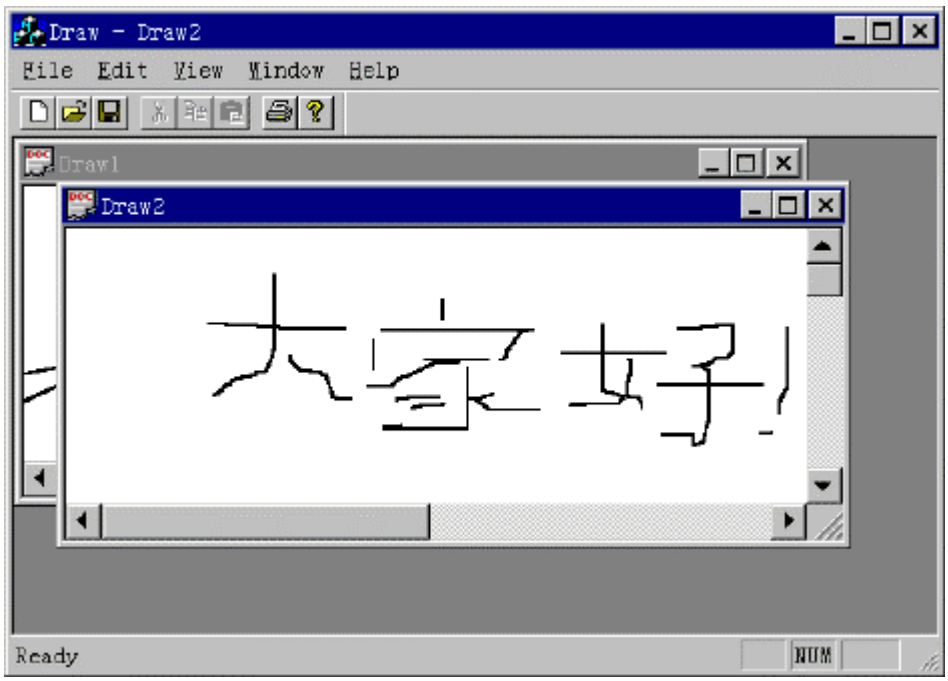


图 8-7 多文档绘图程序窗口

8.4 访问当前活动视图和活动文档

对于 SDI **程序，主框架窗口就是文档框窗，可以采用以下方法取得当前文档和视图：**

取得活动文档：

```
CMyDocument* pDoc;
pDoc=(CMyDocument*)((CFrameWnd*)AfxGetApp()->m_pMainWnd)
->GetActiveDocument();
```

或

```
pDoc=(CMyDocument*)((CFrameWnd*)AfxGetMainWnd())；
```

这两者是等效的。

取得活动视图：

```
CMyView* pView;
pView=(CMyView*)((CFrameWnd*)AfxGetApp()->m_pMainWnd)->G
etActiveView();
```

对于 MDI **程序，由于子窗口才是文档框窗，因此首先要用** GetActiveFrame()**取得活动子框架窗口，然后通过该子窗口获取活动文档和视图：**

```
CMDIChildWnd*
pChild=(CMDIChildWnd*)((CFrameWnd*)AfxGetApp()->m_pMainWnd)->
GetActiveFrame();
```

取得活动文档：

```
CMyDocument* pDoc=pChild->GetActiveDocument();
CMyView* pView=(CMyView*)pChild->GetActiveView();
```

可以把上述函数片段做成静态成员函数，比如：

```
static CMyDocument::GetCurrentDoc()
{
CMDIChildWnd*
pChild=(CMDIChildWnd*)((CFrameWnd*)AfxGetApp()-
>m_pMainWnd)->GetActiveFrame();
CMyDocument* pDoc=pChild->GetActiveDocument();
}
```

这样就可以通过以下方式取得活动文档(或视图)：

```
CMyDocument::GetCurrentDoc();
```

注：静态成员函数调用时不需要一个具体的对象与之相关联。

8.5　分割视图

分割窗口将窗口分成几个部分，每个部分通常代表一个视图(但也可以是具有子窗口标识的 CWnd 对象)，又称窗格。如图 8-8 所示。如果想在一个窗口里面观察文档的不同部分，或者是在一个窗口里用不同类型的视图(比如用图表和表格)观察同一个文档，那么采用分割窗口是非常方便的。许多优秀的软件都采用了分割窗口技术，因此我们有必要掌握分割窗口的用法。

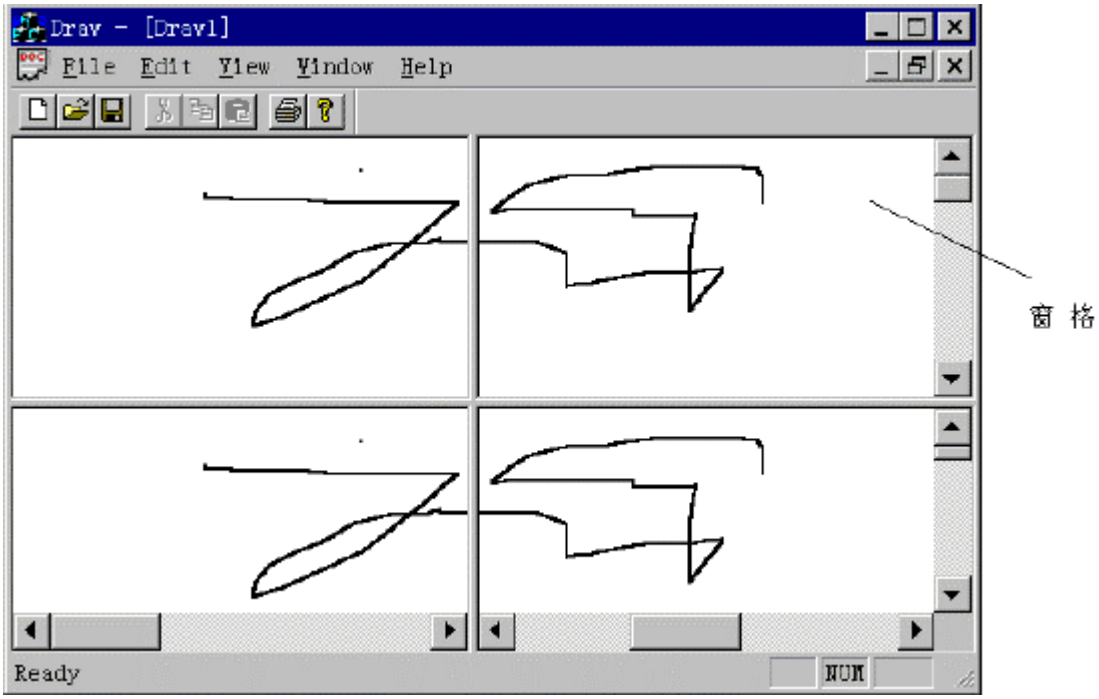


图 8-8　分割窗口

分割窗口分为两类：动态分割窗口和静态分割窗口。

动态分割窗口是指：用户可以动态的分割和除去分割窗口，分割窗口会创建和删除相应的窗格。Microsoft Word 就是使用动态分割窗口的例子，这是一种最常用的分割窗口。动态分割窗口最多可以有 2 行´ 2 列个窗格。

静态分割窗口是指：在窗口创建时，分割窗口的窗格就已经创建好了，且窗格的数量和顺序不会改变。窗格为一个分割条所分割，用户可以拖动分割条调整相应的窗格的大小。如图 8-9，Visual Studio 的图标编辑器就是静态分割窗口的例子。在编辑器的左边窗格，显示图标的缩微图像，在右边显示图标的编辑窗口，可以拖动中间的分割条调整两个窗格的大小。静态分割窗口最多可以有 16 行´ 16 列的窗格。

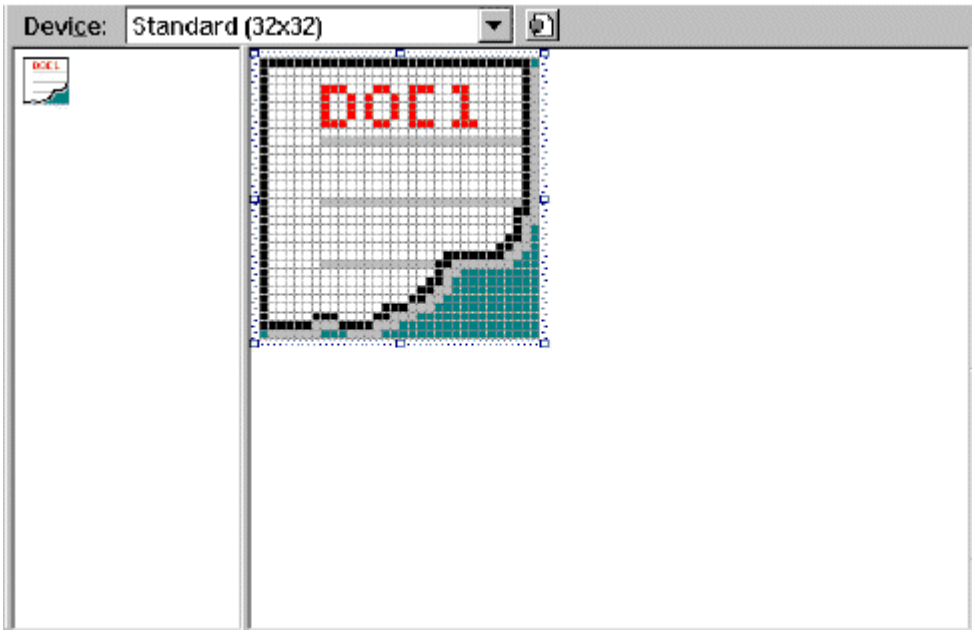


图 8-9 图标编辑器—静态分割窗口的例子

这里我们只介绍动态分割窗口的使用，有关静态分割窗口的用法，读者可以参考 Visual C++ 的例子 VIEWEX，它在 SAMPLES\MFC\GENERAL \VIEWEX 目录下。

要使文档视结构程序支持动态分割窗口，可以有三种方法：

1.在用 AppWizard 创建窗口时指定分割窗口风格：

在 MFC AppWizard Step 4 of 6 对话框中，点 Advanced 按钮。弹出 Advanced Options 对话框，选择 Window Styles 标签页。如图 8-10，选中该页的 Use Split Window 检查框。这样生成的应用程序就自动支持分割窗口功能。

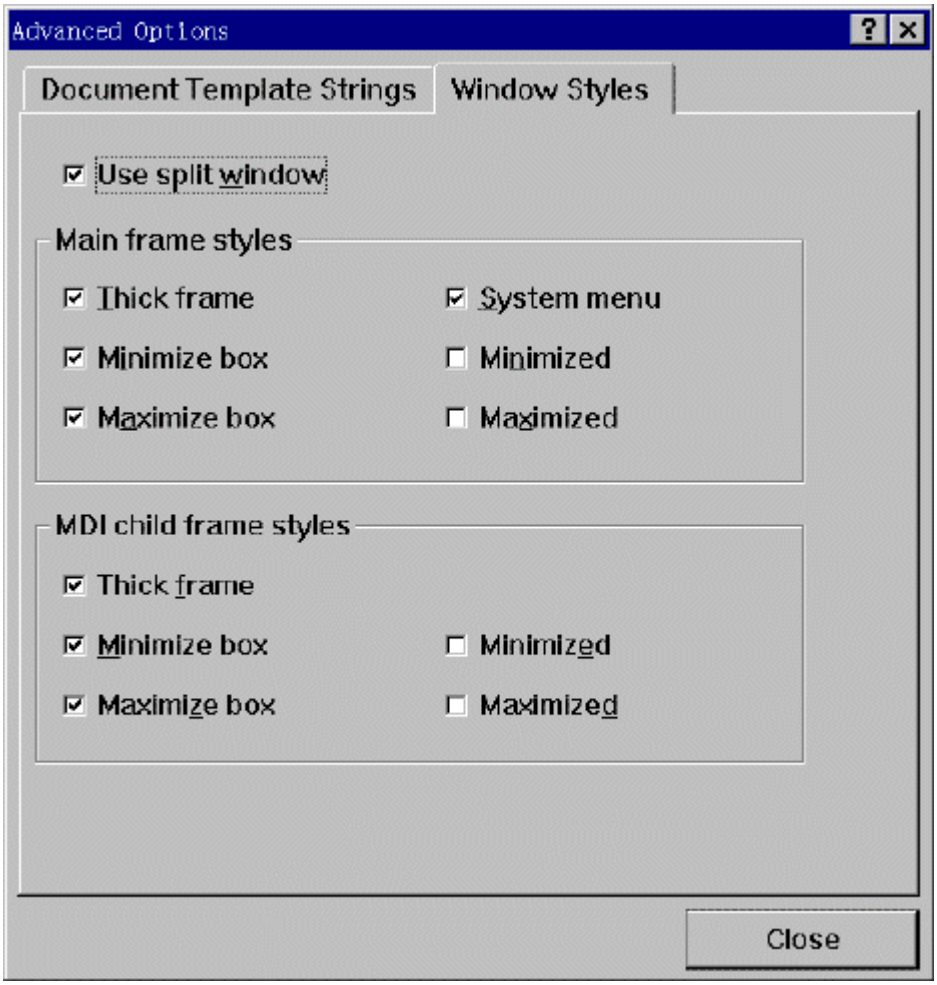


图 8-10 Window Styles 设置

如果应用程序已经生成，采用这种方法就不合适了。此时，可以使用下面的两种方法：

2.使用 Component Gallery 为已经生成的应用程序增加分割窗口功能：

打开相应的工程文件。选择 Project-Add To Project-Components and controls 菜单，弹出

Components and controls Gallery 对话框。双击 Developer Studio Components 目录，从该目录下选择 split Bars 控件。Visual C++提示 split Bar 对话框，对话框内有三个选项：Horizontal，Vertical 和 Both，用于指定在水平方向、垂直方向还是两个方向都使用分割窗口。选择 Both，点 OK 关闭 Split Bar 对话框，此时 Component Gallery 就将分割窗口功能添加到了 Draw 程序中。再点 OK 关闭 Components and controls Gallery 对话框。然后浏览应用程序类，看有什么变化。

在 childfrm.h 中，增加了以下内容：

```
// Generated message map functions
protected:
CSplitterWnd m_wndSplitter;
virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs,
```

CCreateContext* pContext);

m_wndSplitter 是一个 CSplitterWnd 类的对象。CSplitterWnd 是 MFC 提供的一个类，它提供了窗格分割控制，以及能被所有同一行或列上的窗格共享的滚动条。这些行和列的值都是从 0 开始的整数，第一个窗格的行数和列数都为 0。

另外还重载了子框架窗口的 OnCreateClient 方法。在该函数内部，创建了分割窗口控制：

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
// CG: The following block was added by the Split Bars component.
{
if (!m_wndSplitter.Create(this,
2, 2, // TODO: adjust the number of rows, columns
CSize(10, 10), // TODO: adjust the minimum pane size
pContext))
{
TRACE0("Failed to create split bar ");
return FALSE; // failed to create
}
return TRUE;
}
}
```

CSplitterWnd::Create 方法带五个参数，第一个参数代表父窗口指针，第二个参数和第三个参数告诉 CSplitterWnd 要多少行、多少列的窗格，第四个参数是一个 CSize 类型的数据，用于指定窗格的最小大小。

第三种方法是手工加入代码：

在需要分割视图的框架窗口中加入一个 CSplitterWnd 类型的数据成员；用 ClassWizard 重载框架窗口的 OnCreateClient 方法，在 OnCreateClient 方法的实现中，加入上面的代码

8.6 打印和打印预览

最后，我们要给绘图程序增加打印和打印预览功能。我们希望文档分两页打印，第一页为封面，打印文档名字。第二页输出文档内容，并在页眉上打印文档名字。虽然 AppWizard 已经自动生成了打印和打印预览的代码，但是许多情况下，并不能符合要求。

这是因为：

1.打印机和窗口(屏幕)显示的分辨率不同：打印机的分辨率用每英寸多少个点来描述，屏幕分辨率用单位面积的像素点来表示。对于同样的 Arial 字体下的一个字符，在屏幕上用 20 个像素表示，而在打印机上则需要 50 点。在编辑器程序中，使用的映射模式为 MM_TEXT，在这种模式下，一个逻辑单位对应于一个像素点。我们已经知道，Windows 是按照逻辑单位来绘图的。这样，根据 MM_TEXT 模式的逻辑单位(实际上也就是像素数目)决定比例的原则打印出来得内容自然要比屏幕上看到的要小的多。因此，前面在初始化视图 OnInitialUpdate 时候，在选择绘图的映射模式上，没有采用以前使用的缺省的 MM_TEXT 模式，而是采用了 MM_LOENGLISH。

2.窗口和打印机对边界的处理不同：窗口可以看作是无边界的，可以在窗口外面画，而不会引起错误，窗口会自动剪裁超出边界的图形。但打印机却不同，它是按页打印的。打印输出时必须自己处理分页和换页，如果不作这样的处理的话，行和行之间就会叠加起来。

要正确打印输出屏幕上的内容，就必须解决以上两个问题。对于第一个问题，有两种方法：一是利用 SetMapMode(int nMode) 设置别的映射模式，比如采用 MM_LOENGLISH，不用像素而是采用 0.01inch 来衡量。

要处理打印分页、换页，就必须修改框架处理打印消息的缺省行为，在其中计算和换页。此外，我们还希望在打印时在页眉处能够输出标题(使用文件名作为标题)、在页脚处输出页码。

为了实现打印和打印预览功能，首先需要了解 MFC 的打印体系结构，即框架是如何处理打印文档的要求的。

MFC 的打印工作大致上是这样进行的：

1.显示 Print 对话框

2.创建一个与当前打印机设置相匹配的设备上下文(CDC)对象。

3.设置要打印的页数

4.调用 CDC::StartDoc 开始打印

5.用 CDC::StartPage 开始打印一页

6.调用视图的 OnDraw()方法打印输出一页内容

7.用 CDC::EndPage 结束一页的打印

8.循环输出全部内容

9.用 CDC::EndDoc 结束打印

10.视图作打印的清理工作

框架的打印文档功能是从 OnPreparePrinting(CPrintInfo* pInfo)开始

的，在缺省的情况下，它只是简单的调用视图的 DoPreparePrinting()函数。DoPreparePrinting()显示 Print 对话框，并创建与打印机相匹配的设备上下文。如果要想改变打印机初始设置，可以在这里改。缺省设置下，使用 1 作为第一页编号(注意：打印的页号是从 1 开始编号而不是 0)，用 0xFFFF 作为文档的最编号。因为 Draw 要求分两页打印输出，因此要在这里设置打印页数。要设置打印页数，可以调用 CPrintInfo::SetMaxPage(nMaxPage)。同时还将预览页数也设置为两页。

```
BOOL CDrawView::OnPreparePrinting(CPrintInfo* pInfo)
{
pInfo->SetMaxPage(2); // the document is two pages long:
// the first page is the title page
// the second is the drawing
BOOL bRet = DoPreparePrinting(pInfo); // default preparation
pInfo->m_nNumPreviewPages = 2; // Preview 2 pages at a time
// Set this value after calling DoPreparePrinting to override
// value read from .INI file
return bRet;
}
```

DoPreparePrinting 显示 Print 对话框。返回时，CPrintInfo 结构包含了用户所指定的值，包括起止页号、最大页号、最小页号等。

OnBeginPrinting()在 OnPreparePrinting()被调用之后实际打印之前调用。OnBeginPrinting()用于分配 GDI 资源，这里使用缺省行为。

OnPrepareDC 用作屏幕显示时，在绘图前调整 DC。在用于打印时，OnPrepareDC 也完成类似功能。

OnPrint 完成真正的打印一页文档的工作。它把一个打印机设备上下文传给 OnDraw，由 OnDraw 负责打印输出。可以把那些适合于打印但是不适合于屏幕输出的工作，如打印页眉和页脚，放在 OnPrint()的重载中完成，然后再调用 OnDraw 完成打印和显示都需要的工作。现在，我们就在 OnPrint 中加入打印页眉和页脚的代码。OnPrint 不是由 AppWizard 自动生成的，首先要用 ClassWizard 为 CDrawView 增加 OnPrint()方法。然后添加绘图程序的特殊打印代码，见清单 8.10。

清单 8.10 OnPrint()成员函数

```
void CDrawView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
// TODO: Add your specialized code here and/or call the base class
if (pInfo->m_nCurPage == 1) // page no. 1 is the title page
{
PrintTitlePage(pDC, pInfo);
return; // nothing else to print on page 1 but the page title
}
CString strHeader = GetDocument()->GetTitle();
PrintPageHeader(pDC, pInfo, strHeader);
```

```
// PrintPageHeader() subtracts out from the pInfo->m_rectDraw the
// amount of the page used for the header.
pDC->SetWindowOrg(pInfo->m_rectDraw.left,-pInfo->m_rectDraw.top
);
// Now print the rest of the page
OnDraw(pDC);
}
```

OnPrint()首先根据 CPrintInfo 类型的 pInfo 中 m_nCurPage(保存当前打印页号信息)判断当前打印的是不是第一页。如果是第一页，就打印输出封面。否则，首先调用 PrintPageHeader 打印页眉。然后用 SetWindowOrg 调整打印输出原点位置。m_rectDraw 又是 CPrintInfo 结构的一个重要数据成员，它保存的是打印输出的矩形边界。最后将与打印机匹配的设备上下文传给 OnDraw，由 OnDraw 在打印机上输出。注意这里使用的映射模式为 MM_LOENGLISH，它的 y 轴方向是向上递增的。

PrintTitlePage 打印输出文档的封面。它首先定义一种逻辑字体，设置逻辑字体属性，然后由调用 CreateFontIndirect 由逻辑字体创建字体。SetTextAlign(TA_CENTER)将文本设置为居中输出。然后调用 TextOut 在打印矩形 m_rectDraw 上输出封面。PrintTitlePage 函数定义见清单 8.11。

清单 8.11 PrintTitlePage 成员函数

```
void CDrawView::PrintTitlePage(CDC* pDC, CPrintInfo* pInfo)
{
// Prepare a font size for displaying the file name
LOGFONT logFont;
memset(&logFont, 0, sizeof(LOGFONT));
logFont.lfHeight = 75; // 3/4th inch high in MM_LOENGLISH
// (1/100th inch)
CFont font;
CFont* pOldFont = NULL;
if (font.CreateFontIndirect(&logFont))
pOldFont = pDC->SelectObject(&font);
// Get the file name, to be displayed on title page
CString strPageTitle = GetDocument()->GetTitle();
// Display the file name 1 inch below top of the page,
// centered horizontally
pDC->SetTextAlign(TA_CENTER);
pDC->TextOut(pInfo->m_rectDraw.right/2, -100, strPageTitle);
if (pOldFont != NULL)
pDC->SelectObject(pOldFont);
}
```

PrintPageHeader 在页眉位置输出文件名，然后从 m_rectDraw 扣除页眉的大小。

```
void CDrawView::PrintPageHeader(CDC* pDC, CPrintInfo* pInfo,
```

```
CString& strHeader)
{
// Print a page header consisting of the name of
// the document and a horizontal line
pDC->SetTextAlign(TA_LEFT);
pDC->TextOut(0,-25, strHeader); // 1/4 inch down
// Draw a line across the page, below the header
TEXTMETRIC textMetric;
pDC->GetTextMetrics(&textMetric);
int y = -35 - textMetric.tmHeight; // line 1/10th inch below text
pDC->MoveTo(0, y); // from left margin
pDC->LineTo(pInfo->m_rectDraw.right, y); // to right margin
// Subtract out from the drawing rectange the space used by the header.
y -= 25; // space 1/4 inch below (top of) line
pInfo->m_rectDraw.top += y;
}
```

作为一个练习，读者可以修改 OnPrint()**并增加一个** PrintPageFooter() **函数，在每一页的页脚处输出打印的页号。注意调用** OnDraw **之前，要从** m_rectDraw **中扣除页脚的高度。**

8.7 支持多个文档类型的文档视结构程序

要支持多种文档类型，可以在 CWinApp 派生类对象中创建和注册附加的 CMultiDocTemplate 对象。在 MFC 应用程序中，要增加附加的文档类型，步骤可分为五步。下面我们试着将上一章的文本编辑器加到绘图程序中。这样程序不仅支持绘图，还支持文本编辑功能。

(1)使用 ClassWizard 创建新的文档类和视图类：

由于已经有了前面的文本编辑器程序，只需要将其中的文件拷贝过来就可以了，然后用 Project->Add To Project->Files 命令，将 EditorDoc.h、EditorDoc.cpp、EditorView.h、EditorView.cpp 加入到工程中。

(2)利用资源编辑器为新的文档类型增加新的字符串。

先看看绘图程序的文档模板字符串结构。打开字符串编辑器，找到 IDR_DRAWTYPE 字符串，它是这样定义的：

\nDraw\nDraw\nDraw Files(*.drw)\n.drw\nDraw.Document\nDraw Document

文档模板字符串包含 7 个由 '\n' 结尾并分隔的子串。如果子串不包含则 '\n' 作为一个占位字符出现，最后一个字符串后面不需要 '\n'。这些子串描述了文档的类型，它们分别代表：

1.窗口标题：如 Microsoft Word，该字符串仅出现在 SDI 程序中，对于多文档程序为空。因此 IDR_DRAWTYPE 以\n 开头。

2.文档名：在用户从 File 菜单选取 New 菜单项时，建立新文档名。新的文档名使用这个文档名字符串作为前缀，后面添加一个数字，用作缺省的新文件名，如 " Draw1 "、" Draw2 " 等。如果没有指定，则使用 " untitled " 作为缺省值。

3.新建文档类型名：当应用程序支持多个文档类型时，该字符串显示在 File New 对话框中。如果没有指定，则无法用 File-New 菜单项创建该类型的文档。

4.过滤器名：允许指定与这个文档类型相关的描述。此描述显示在 Open 对话框中的 Type 下拉列表中。

5.过滤器后缀：与过滤器名一起使用，指定与文档类型相关的文件的扩展名。对于绘图程序我们在前面已经指定为 " .drw "。

6.标注 Windows 维护的注册数据库中的文档类型 Id。应用程序运行时会将该 Id 加入到注册数据库中。这样 File Manager 就可以通过 Id 和下面的注册文档类型名打开相应的应用程序。

7.注册文档类型名：存放在注册数据库中，标识文档类型的名字。

现在我们要加入文本编辑器的文档模板字符串。在字符串编辑器中增加一个字符串资源，指定 ID 为 IDR_EDITORTYPE，内容为：

\nEditor\nEditor\nEditor Files(*.txt)\n.txt\nEditor.Document\nEditor Document

(3)利用资源编辑器增加附加的图标和菜单资源。注意这些资源的 ID 必须同第二步中创建文档模板字符串中所用的 ID 相同。CMultiDocTemplate 类利用该 ID 来识别与附加的文档类型相关的所有资

源(包括图标、菜单等)。可以在打开 Draw 项目工作区文件后，用 Project-Insert Project into Workspace 将 Editor 工程文件加入到 Draw 项目工作区中。然后从 Editor 中拷贝资源到 Draw 工程并更名为 IDR_EDITORTYPE。

(4) 在应用程序类的 InitInstance() 方法中，创建另一个 CMultiDocTemplate 对象，并用 CWinApp::AddDocTemplate()成员函数注册该模板对象。修改后的代码是这样的：

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
IDR_DRAWTYPE,
RUNTIME_CLASS(CDrawDoc),
RUNTIME_CLASS(CChildFrame), // custom MDI child frame
RUNTIME_CLASS(CDrawView));
AddDocTemplate(pDocTemplate);
CMultiDocTemplate* pDocTemplate2=new CMultiDocTemplate(IDR_EDITORTYPE,
RUNTIME_CLASS(CEditorDoc),
RUNTIME_CLASS(CMDIChildWnd),RUNTIME_CLASS(CEditorView));
AddDocTemplate(pDocTemplate2);
```

(5)最后，增加定制的串行化方法和绘图方法到新增的文档和视图类中。

对于 CEditorDoc 和 CEditorView，这一步工作已经在前面做好了。

现在编译并运行程序。

8.8 防止应用程序运行时自动创建空白窗口

在前面的 MDI 程序中，当应用程序启动时，都会自动创建一个空白窗口。但有时我们并不希望创建这样的空白窗口。比如，对于一个文件浏览器来说，空白窗口就没有什么意义。

要防止空白窗口的创建，首先就要明白这个窗口是如何被创建的。在 InitInstance()中，有一个命令行的执行过程，当命令行上没有参数时，函数 ParseCommandLine(cmdInfo) 会将 CCommandLineInfo::m_nShellCommand 成员置为 CCommandLineInfo::FileNew，这将导致 ProcessShellCommand 调用 CWinApp::OnFileNew 成员函数。要想防止程序开始时就调用 OnFileNew，解决方法之一是去掉与命令行有关的代码，但是这样就没有了命令行处理功能。另一种方法是在 ProcessShellCommand 调用之前加一句 cmdInfo.m_nShellCommand =CCommandLineInfo::FileNothing。具体代码见清单 8.12。

清单 8.12 不自动创建空白文档窗口的 InitInstance 成员函数定义

```
BOOL CDrawApp::InitInstance()
{
// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
// Alter behaviour to not open window immediately
cmdInfo.m_nShellCommand = CCommandLineInfo::FileNothing;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
//.....
}
```

小结

这一讲介绍多文档应用程序的设计：

多文档应用程序中的对象：主框架窗口对象、子框架窗口、文档、视图及其相互关系

GDI 图形设备接口编程：GDI 的三种类型输出、GDI 设备上下文类、GDI 绘图对象及其使用、常见的绘图任务(画点、画线、矩形等)

绘图程序设计：设计文档类

设计绘图程序视图类

在 MDI 中访问文档对象和视图对象

分割视图窗口

文档视结构程序的打印和打印预览

支持多种文档类型的文档视图应用程序的设计

避免在 MDI 运行时创建空白窗口

第九讲　创建用户模块

这一讲介绍 Visual C++的用户模块的概念，并介绍静态库、动态连接库、扩展类库的设计与使用。

用户模块

静态连接库

创建动态库

小结

9.1 用户模块

用户模块是由用户自己开发的、可以加入到最终用户(包括用户本人和其他使用该模块的人)应用程序中提供某一特定功能的函数和类的集合。

为了完成同样的工作，也可以向最终用户提供源程序。但是，使用用户模块有许多好处：首先是省去用户管理源代码的烦恼，用户许多情况下往往并不关心模块的内部实现，他只是想把它作为一个黑匣子使用。另外，模块的开发者有时候并不希望模块使用者看到源代码。还有，使用模块而不使用源代码还可以避免模块的函数名、变量名与最终用户的程序上的冲突。

用户模块可分为两大类：静态连接库和动态连接库。

静态连接库提供了函数的完整的目标代码，如果程序调用静态连接库中的函数，则在进行连接时连接程序将静态连接库中所包含的该函数的代码拷贝至运行文件中。

动态连接库是一个可执行模块，其包含的函数可以由 Windows 应用程序调用以执行一些功能。动态连接库主要为应用程序模块提供服务。Windows 内核的三个模块 USER.EXE、KENERL.EXE 和 GDI.EXE 实际上都是动态连接库，分别提供用户消息服务、进程管理、图形输出等服务。

动态连接库也包含了其所提供的函数的目标代码，但是在程序连接动态连接库中的函数时，连接程序并不将包含在动态连接库中的函数的目标代码拷贝至运行文件，而只是简单地记录了函数的位置信息(即包含于哪个动态连接库中以及在动态连接库中的位置)。有了这些信息后，程序在执行时，即可找到该函数的目标代码。因为只是在执行时才得到真正的连接，因此称为动态连接。提供函数在动态连接库中位置的信息存放在一个独立的文件中，这个文件就是引入库(IMPORT LIB)。

由于静态连接库将目标代码连接到应用程序中，当程序运行时，如果两个程序调用了同一静态库中的函数，内存中将出现该函数的多份拷贝。而动态连接库则更适合于多任务环境：当两个应用程序调用了同一动态连接库中的同一个函数时，内存中只保留该函数的一份拷贝，这样内存利用率更高。

利用动态连接库还可以实现资源共享：像 Windows 下的串行口、并行口驱动程序都是动态连接库；另外，Windows 下的字体也是动态连接库。

但是，静态库由于将目标代码连入应用程序中，应用程序可独立运行。而使用动态连接库时，随同应用程序还要提供动态连接库文件(DLL 文件)。比如，发布 Visual C++编写的程序时，如果使用了动态连接，则在提供可执行文件同时还需要提供 Visual C++的动态连接库。

应用程序和动态连接库都是完成一定功能的可执行模块。它们的区别是：应用程序有自己的消息循环，而动态连接库没有自己的消息循环(但是它可以发送消息)；应用程序一般是主动完成某一功能的，而动态连

接库主要是被动(在中断驱动程序中也主动完成一些功能)的提供服务。

9.2 静态连接库

9.2.1 创建静态库

现在以一个简单的数学函数库为例介绍静态库的创建和使用。

要创建静态库，选择 File->New 菜单，弹出 New 对话框。选择 Projects 标签，在项目类型列表框中选择 Win32 Static Library，在 Name 中输入 mymath，表明要创建一个 mymath.lib 的静态库文件。

然后用 Project->Add to Project->Files 菜单往 mymath 工程中加入以下两个文件：

1.头文件(见清单 9.1)：定义了 Summary 和 Factorial 两个函数，分别用于完成求和与阶乘。注意这里使用 C 风格的函数，需要加入 extern “C”关键字，表明它是 C 风格的外部函数。

清单 9.1 头文件

```
#ifndef _MYMATH_H
#define _MYMATH_H
extern “C”
{
int Summary(int n);
int Factorial(int n);
}
#endif
```

2.源文件：包含了 Summary 和 Factorial 函数的定义，见清单 9.2。

清单 9.2 源文件

```
int Summary(int n)
{
int sum=0;
int i;
for(i=1;i<=n;i++)
{
sum+=i;
}
return sum;
}
int Factorial(int n)
{
int Fact=1;
int i;
for(i=1;i<=n;i++)
{
Fact=Fact*i;
}
return Fact;
```

}

在 Build **菜单下，选择** Build **菜单下的** Build mymath.lib。Visual C++ **编译链接工程，在** mymath\debug **目录下生成** mymath.lib **文件。至此，静态连接库生成的工作就做完了。下面用一个小程序来测试这个静态库。**

提示：用户在交付最终静态连接库时，只需要提供.lib **文件和头文件，不需要再提供库的源代码。**/

9.2.2 **测试静态库**

用 AppWizard **生成一个基于对话框的应用程序** test。**打开** test **资源文件，修改** IDD_TEST_DIALOG **对话框资源，加入两个按钮。按钮** ID **和文字为：**

IDC_SUM “&Summary”

IDC_FACTORIAL “&Factorial”

如图 9-1 **所示。**

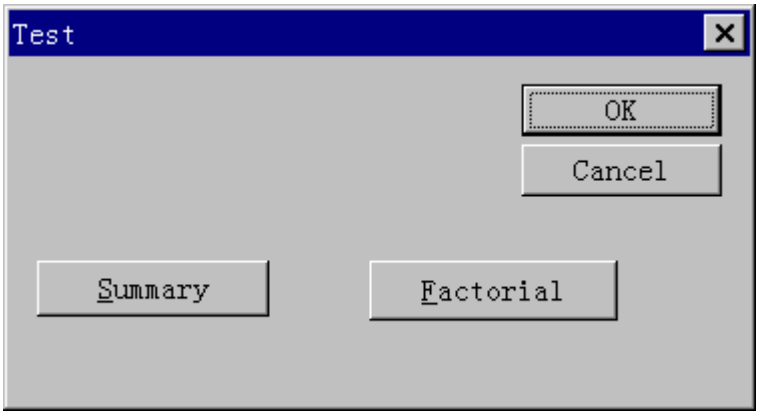


图 9-1 **修改** test **对话框**

用 ClassWizard **为上述两个按钮** Click **事件生成消息处理函数** OnSum **和** OnFactorial**，并加入代码**,**修改后的** OnSum **和** OnFactorial **见清单** 9.3。

清单 9.3 OnSum **和** OnFactorial **函数定义**

```
void CTestDlg::OnSum()
{
// TODO: Add your control notification handler code here
int nSum=Summary(10);
CString sResult;
sResult.Format(“Sum(10)=%d”,nSum);
AfxMessageBox(sResult);
}
void CTestDlg::OnFactorial()
{
// TODO: Add your control notification handler code here
int nFact=Factorial(10);
CString sResult;
sResult.Format(“10!=%d”,nFact);
AfxMessageBox(sResult);
```

}

由于要使用 mymath.lib **中的函数，首先要将** mymath.lib **和** mymath.h **两个文件拷贝到** test **目录下。然后用** Project->Add to Project->Files **命令，将** mymath.lib **加入到工程中。**

在 testdlg.cpp **文件头部，还要加入头文件** mymath.h **：**

```
#include “stdafx.h”
#include “Test.h”
#include “TestDlg.h”
#include “mymath.h”
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

编译运行 test **程序，点** Factorial **按钮，弹出如图** 9-2 **的消息框。**

图 9-2 Test **程序运行结果**

9.3 创建动态连接库

在一些情况下，必须使用动态连接库：

1.多个应用程序共享代码和数据：比如 Office 软件的各个组成部分有相似的外观和功能，这就是通过共享动态连接库实现的。

2.在钩子程序过滤系统消息时必须使用动态连接库

3.设备驱动程序必须是动态连接库

4.如果要在对话框编辑器中使用自己定义的控件，也必须使用动态连接库

5.动态连接库以一种自然的方式将一个大的应用程序划分为几个小的模块，有利于小组内部成员的分工与合作。而且，各个模块可以独立升级。如果小组中的一个成员开发了一组实用例程，他就可以把这些例程放在一个动态连接库中，让小组的其他成员使用。

6.为了实现应用程序的国际化，往往需要使用动态连接库。使用动态连接库可以将针对某一国家、语言的信息存放在其中。对于不同的版本，使用不同的动态连接库。在使用 AppWizard 生成应用程序时，我们可以指定资源文件使用的语言，这就是通过提供不同的动态连接库实现的。

MFC 支持两类动态连接库的创建：

用户动态连接库

MFC 扩展类库。

9.3.1 用户动态连接库(_USRDLL)

用户动态连接库一般使用 C 语言接口。要创建一个动态连接库，选择 File->New 菜单，弹出 New 对话框。在 Projects 标签页下，选择“ Win32 Dynamic-Link Library ”。Visual C++就会创建动态连接库所需的工程文件和 MAK 文件。

然后把下面两个文件加入到工程中(Project-Add to Project-Files 菜单)。

文件 1：mymaths.cpp

```
////////////////////////////
//mymaths.cpp
//

//a maths API DLL.
//
////////////////////////////
#include<windows.h>
//Declare the DLL functions prototypes
int Summary(int);
int Factorial(int);
////////////////////////////
//DllEntryPoint():The entry point of the DLL
```

```
//
//////////////////////
BOOL WINAPI DLLEntryPoint(HINSTANCE hDLL,DWORD dwReason,
LPVOID Reserved)
{
switch(dwReason)
{
case DLL_PROCESS_ATTACH:
{
//一些初始化代码
break;
}
case DLL_PROCESS_DETACH:
{
//一些用于清理的代码
break;
}
}
return TRUE;
}
int Summary(int n)
{
int sum=0;
int i;
for(i=1;i<=n;i++)
{
sum+=i;
}
return sum;
}
int Factorial(int n)
{
int Fact=1;
int i;
for(i=1;i<=n;i++)
{
Fact=Fact*i;
}
return Fact;
}
```

文件 2：mymaths.def

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Mymaths.DEF
;
;The DEF file for the Mymaths.DLL DLL.
;
LIBRARY mymaths
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD SINGLE
EXPORTS
;The names of the DLL functions
Summary
Factorial
```

在文件 mymaths.cpp **开头，声明了动态连接库所包含的两个函数：**Summary **和** Factorial**。接着是** DllEntryPoint()**函数的定义。**DllEntryPoint() **顾名思义是动态连接库的入口，应用程序通过该入口访问动态连接库提供的服务。**DllEntryPoint()**主体是一个** switch/case **语句：**

```
switch(dwReason)
{
case DLL_PROCESS_ATTACH:
{
//一些初始化代码
break;
}
case DLL_PROCESS_DETACH:
{
//一些用于清理的代码
break;
}
}
```

其中，在 case DLL_PROCESS_ATTACH **分支可加入动态连接库执行时的一些初始化代码。在** case DLL_PROCESS_DETACH **加入动态连接库被卸载时的一些清理代码，比如释放动态连接库运行时申请的内存等。**

在 DllEntryPoint()**函数后，是两个函数** Summary **和** Factorial **函数的定义。它们的定义与前面的静态库完全相同。在这里用户可以放入任何函数。**

另外，我们还需要一个 mymaths.def **文件。这个文件记录了可被外部应用程序使用的** DLL **库函数名字。这些名字信息和对应的函数位置的信息将被编译进动态连接库文件中，然后应用程序根据函数名字和函数位置对照表来找到对应的函数。**

按 F7 **编译工程，**Visual C++**就在** mymaths\debug **目录下生成一个** mymaths.dll **动态连接库文件。**

现在，我们来使用刚才生成的动态连接库。我们并不重新生成一个

程序，而是修改前面测试静态库时的 test 程序。首先，把 mymaths\debug 目录下的 mymaths.dll 拷贝到 test\debug 目录下。test 程序运行时，会在该目录下搜索动态连接库文件。然后修改 testdlg.h，在其中加入一个函数 LoadDLL()的声明，见清单 9.4。LoadDLL 用于载入动态连接库。

清单 9.4 修改后的对话框头文件

```
class CTestDlg : public CDialog
{
// Construction
public:
CTestDlg(CWnd* pParent = NULL); // standard constructor
protected:
void LoadDLL();
//.....
}
```

然后修改 testdlg.cpp，修改后如清单 9.5。

清单 95. TestDlg.cpp 文件

```
// TestDlg.cpp : implementation file
//
#include “stdafx.h”
#include “Test.h”
#include “TestDlg.h”
//#include “mymath.h” //注释掉 mymath.h 头文件
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//The instance of the Mymaths.DLL library
HINSTANCE ghMathsDLL=NULL;
//declare the Summary() function from the Mymaths.DLL libray.
typedef int (*SUMMARY)(int);
SUMMARY Summary;
//declare the Factorial() function from
//the Mymaths.DLL library.
typedef int (*FACTORIAL)(int);
FACTORIAL Factorial;
/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
//...
};
```

```
//CAboutDlg 的一些成员函数定义
//CTestDlg 的一些成员函数定义
void CTestDlg::OnSum()
{
// TODO: Add your control notification handler code here
LoadDLL();
int nSum=Summary(10);
CString sResult;
sResult.Format("Sum(10)=%d",nSum);
AfxMessageBox(sResult);
}
void CTestDlg::OnFactorial()
{
// TODO: Add your control notification handler code here
LoadDLL();
int nFact=Factorial(10);
CString sResult;
sResult.Format("10!=%d",nFact);
AfxMessageBox(sResult);
}
void CTestDlg::LoadDLL()
{
//如果 DLL 已经载入，则返回
if(ghMathsDLL!=NULL)
{
return;
}
//载入 Mymaths.DLL 文件.
ghMathsDLL=LoadLibrary("mymaths.DLL");
//如果载入 DLL 失败，提示用户
if(ghMathsDLL==NULL)
{
AfxMessageBox("Cannot load DLL file!");
}
//获得 DLL 中 Summary 函数的地址
Summary=(SUMMARY)GetProcAddress(ghMathsDLL,"Summary");
//获得 DLL 中 Factorial 函数的地址
Factorial=(FACTORIAL)GetProcAddress(ghMathsDLL,"Factorial");
}
在 testdlg.cpp 文件开头，加入：
//The instance of the Mymaths.DLL library
HINSTANCE ghMathsDLL=NULL;
```

```
//declare the Summary() function from the Mymaths.DLL libray.
typedef int (*SUMMARY)(int);
SUMMARY Summary;
//declare the Factorial() function from
//the Mymaths.DLL library.
typedef int (*FACTORIAL)(int);
FACTORIAL Factorial;
```

首先加入一个 ghMathsDLL 的全局变量，它是动态连接库载入后的句柄(同应用程序一样，每个动态连接库载入都会有一个句柄和它相对应)。应用程序通过句柄访问库中的函数。然后加入 Summary 和 Factorial 函数指针的类型定义。

在 LoadDLL()函数定义中，检查动态连接库句柄是否为空；若为空，则用 LoadLibrary 载入该动态连接库。然后用 GetProcAddress 取得 Summary 和 Factorial 函数地址。

在 OnFactorial 和 OnSummary 函数开头，调用 LoadDLL()，载入动态连接库。现在编译运行程序，按 Factorial 按钮测试一下程序。

应用程序是如何查找 DLL 文件的

应用程序 test 按以下顺序查找动态连接库文件：

当前目录下(因此要将动态连接库拷贝至 DEBUG 目录下，因为可执行文件在该目录下)Windows 目录 Windows 系统目录 PATH 环境变量中设置的目录列入映射网络的目录表中的目录

调用动态连接库中的函数的方法

有两种方法可以调用动态连接库中的函数：

1.通过引入库：

利用 Visual C++提供的 IMPLIB 工具为动态连接库生成引入库,为引入库设计一个头文件：

```
#ifndef _MYMATH_H
#define _MYMATH_H
extern “C”
{
int Summary(int n);
int Factorial(int n);
}
#endif
```

将该头文件包含在使用动态连接库的源文件中，连接应用程序时会连接上该引入库。这样，应用程序就可以象使用静态连接库一样自由的使用动态连接库中的函数了。注意要把动态连接库拷贝到应用程序可执行文件所在的目录(\TEST\DEBUG)下。

这是一种常用的方法。实际上，应用程序就是通过这种方式访问 Windows 的 API 函数的。Windows 为其内核动态连接库生成引入库并提供了头文件。应用程序在编译时将引入库的信息带入可执行文件中，在运行时通过引入库信息访问 API 函数。

2. **直接指定库和函数地址**

这种方式适合于一些提供文件格式转换等服务的动态连接库。比如，一个程序带有多个动态连接库，分别用于访问 JPG、BMP、GIF **等多种图像文件格式，这些动态连接库提供了相同的库函数接口。此时，无法使用引入库方式指定库函数。可以采用下面的方法来解决这个问题。**

```
HANDLE hLibrary;
FARPROC lpFunc;
int nFormat;
if(nFormat==JPEG)//如果是 JPEG 格式，装入 JPEG 动态连接库
{
hLibrary=LoadLibrary(“JPEG.DLL”);
}
else//是 GIF 格式
hLibrary= LoadLibrary(“GIF.DLL”);
if(hLibrary>=32)
{
lpFunc=GetProcAddress(hLibrary,”ReadImage”);
if(lpFunc!=(FARPROC)NULL)
(*lpFunc)((LPCTSTR)strFileName);
FreeLibrary(hLibrary);
}
```

LoadLibrary **函数装入所需的动态连接库，并返回库的句柄。如果句柄小于** 32**，则载入库失败，错误含义参见有关手册。**GetProcAddress **函数使用函数名字取得函数的地址。利用该函数地址，就可以访问动态连接库的函数了。**

FreeLibrary **通过检查动态连接库的引用计数器，判断是否还有别的程序在使用这个动态连接库。如果没有，就从内存中移去该动态连接库；如果有，将动态连接库的使用计数器减** 1。LoadLibrary **则将引用计数加** 1。

在用户动态连接库中，也可以使用 MFC **类。这时，可以选择静态连接和动态连接两种方式使用** MFC **库。**

9.3.2 MFC **扩展类库**(_AFXDLL)

除了创建具有 C **语言接口的用户动态连接库外，**MFC **还允许用户在动态连接库中创建** MFC **类的派生类，这些类作为** MFC **类的自然延伸出现，可以为其他** MFC **应用程序所使用，就象使用普通的** MFC **类一样。**

创建扩展类库

要创建扩展类库，可以选择 File->New **菜单，在** Projects **类型中选择** MFC AppWizard(dll)**。弹出** MFC AppWizard 1of 1 **对话框，从中选择** MFC Extension DLL(using shared MFC DLL)。AppWizard **就会生成** Extension DLL **所需的框架。**

这里不再创建动态连接库，而是用 Visual C++**的例子** DLLHUSK **程序(在** SAMPLES\MFC\ADVANCED \DLLHUSK **目录下)说明扩展类库的**

创建和使用。

在 DLLHUSK 项目工作区中，包含三个工程：DLLHUSK，TESTDLL1，TESTDLL2。

TESTDLL1 和 TESTDLL2 分别定义了几个扩展类：CTextDoc、CHelloView 和 CListOutputFrame，DLLHUSK 是使用这些类的示例程序。

在 CListOutputFrame 声明中，要加入 AFX_EXT_CLASS，表明它是一个 MFC 扩展类。

```
class AFX_EXT_CLASS CListOutputFrame:public CMDIChildWnd
{
...
}
```

在函数定义处，还要包含 afxdllx.h 头文件

```
// Initialization of MFC Extension DLL
#include “afxdllx.h” // standard MFC Extension DLL routines
```

类的成员函数使用与应用程序中类的使用大致相同。

在 CListOutputFrame 类定义文件中，还提供了一个 C 函数。它的函数声明在类头文件 testdll2.h 中：

```
// Initialize the DLL, register the classes etc
extern “C” AFX_EXT_API void WINAPI InitTestDLL2();
```

这个函数用于初始化动态连接库和注册类：

```
// Exported DLL initialization is run in context of running application
extern “C” void WINAPI InitTestDLL2()
{
// create a new CDynLinkLibrary for this app
new CDynLinkLibrary(extensionDLL);
// nothing more to do
}
```

另外，源文件中还需要提供一个 DllMain 函数：

DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)

这个函数作用与前面的 DllEntryPoint 类似。

扩展类库也需要一个 DEF 文件，这个文件包含了动态连接库中可用的函数信息。由于现在动态连接库包含的是类，因此在函数命名上与用户动态连接库有所不同。

```
EXPORTS
“AddString@CListOutputFrame@@QAAXPBD@Z
““_7CListOutputFrame@@6B@
““_GCListOutputFrame@@UAAPAXI@Z
“OnEditCut@CListOutputFrame@@IAAXXZ
“_messageEntries@CListOutputFrame@@0QBUAFX_MSGMAP_ENTRY@@B
““0CListOutputFrame@@QAA@XZ
```

““1CListOutputFrame@@UAA@XZ
“Clear@CListOutputFrame@@QAAXXZ
“OnEditClear@CListOutputFrame@@IAAXXZ
“OnEditCopy@CListOutputFrame@@IAAXXZ
InitTestDLL2
.....

有关函数名扩展的技术参考 Visual C++**帮助文档。**

使用扩展类动态连接库

要使用扩展类库，要将类库的头文件包含在工程中。然后在适当位置初始化类库，DLLHusk **是在** InitInstance **中完成这一工作的。**

```
BOOL CHuskApp::InitInstance()
{
//...
InitTestDLL1();
InitTestDLL2();
//...
}
```

然后就可以象使用普通 MFC **类一样使用扩展类库中定义的类了。**

```
m_pListOut=new CListOutputFrame;
```

访问 DLL **中的资源**

当应用程序使用资源时，它按以下顺序查找资源：首先查找应用程序本身，看有没有对应的资源；如果没有，查找 MFC400.DLL(**或** MFC400D.DLL**，它包含调试信息**)**。再查找应用程序所带的动态连接库中的资源。如果想在** DLL **中直接使用资源而不经过以上搜索顺序，可以使用** AfxGetResouceHandle()**和** AfxSetResourceHandle()**函数。**

AfxGetResourceHandle()**和** AfxSetResouceHandle()**函数分别用来保存旧的资源句柄和设置新的资源句柄。比如，要想直接从** DLL **中载入一个位图资源，可以这么调用：**

```
CBitmap mybitmap;
HINSTANCE hInstOld=AfxGetResourceHandle()
AfxSetResouceHandler(extensionDLL.hModule);
if(!mybitmap.LoadBitmap(IDR_BITMAP));
{
//restore the old resouce chain and return error
AfxSetResouceHandle(hInstOld);
return FALSE;
}
AfxSetResouceHandle(hInstOld);
//use this bitmap...
return TRUE;
```

还可以使用 FindResource()**搜索资源表，寻找给定的资源。**

```
HRsrc FindResource(
```

HMODULEhModule,
LPCTSTRlpName,
LPCTSTRlpType
);

FindResource **带三个参数，第一个参数是模块句柄，第二个是要查找的资源名字，如‘** MYDIALOG **”，第三个是资源类型，可参见** Visual C++ **文档。如果查找成功，则返回该资源句柄。可以用** LoadResouce **以该句柄为参数装入资源。**

小结

本章介绍了用户模块的创建和使用。

用户模块是由用户自己开发的、可以加入到最终用户应用程序中提供某一特定功能的函数和类的集合。

用户模块包括静态连接库和动态连接库两大类：静态连接库将函数的目标代码直接连入到应用程序中；动态连接库只是给出函数入口信息，在调用时访问 DLL 文件中函数的目标代码。

创建静态连接库：指定工程类型为 Win32 Static Library，加入函数声明和定义，并编译和连接。提交函数库时只需要提供函数的 lib 文件和头文件。要使用静态库，可以将函数库和头文件包含在工程文件中。

创建动态连接库：提供函数定义、声明以及包含 DLL 文件函数信息的 DEF 文件。使用时需要将 DLL 文件拷贝至适当目录下。

两类动态连接库的创建：用户动态连接库和 MFC 扩展库。

第十课　数据库编程

MFC 提供了对数据库编程的强大支持。对于数据库的访问，MFC 提供了两组类：ODBC(Open Database Connectivity)和 DAO(Database Access Object)。利用这两个功能强大的类，用户可以方便的开发出基于 ODBC 或 DAO 的数据库应用。

这一讲将重点介绍下列内容：

数据库的基本概念

ODBC 基本概念

MFC 的 ODBC 类简介

CDatabase 类

CRecordset 类

CRecordView 类

用 AppWizard 和 ClassWizard 编写 Enroll 数据库应用例程

DAO 和 DAO 类

自动注册 DSN 和创建表

小结

10.1　数据库、DBMS 和 SQL

数据库是数据的集合，它由一个或多个表组成。每一个表中都存储了对一类对象的数据描述，一个典型的表如表 10.1 所示。表的每一列描述了对象的一个属性，如姓名、出生年月等，而表的每一行则是对一个对象的具体描述。一般将表中的一行称作记录(record)或行(row)，将表的每一列称作字段(field)或列(column)。数据库通常还包括一些附加结构用来维护数据。

表 10.1

学号	姓 名	出生年月	性别
1	李明	06/12/77	男
2	张芳	11/24/78	女

若一个数据库只有一个表，则称之为简单数据库。若数据库由多个相关的表组成，则称其为关系数据库。关系数据库利用公共关键字段将它的表联系起来，例如在表 10.1 中，可以将学号作为一个关键字段，如果数据库中还有一个学生成绩表并且也有学号字段，则可以通过学号这个关键字段将两个表联系起来。

DBMS(数据库管理系统)是一套程序，用来定义、管理和处理数据库与应用程序之间的联系，例如 FoxPro、Access、Sybase 等都是 DBMS。图 10.1 说明了用户、DBMS 和数据库三者的关系。

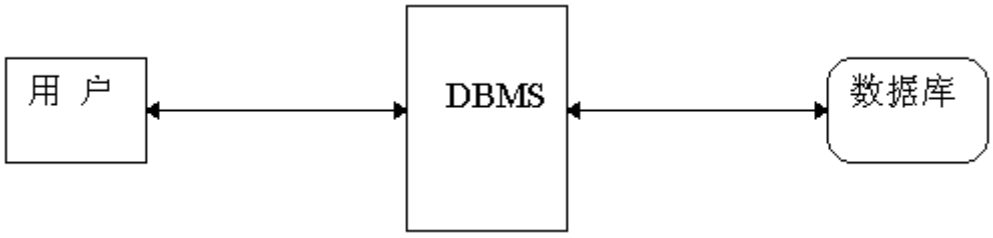


图 10.1 用户、DBMS、数据库三者的关系

SQL(Structured Query Language，结构化查询语言) 最早由 IBM 提出，是专门用来处理关系数据库的基于文本的语言。SQL 向数据库提供了完善而一致的接口，它不是独立的计算机语言，需要 DBMS 的支持方能执行。SQL 是一种标准的数据库语言，目前大多数 DBMS 都支持它。

10.2 ODBC 基本概念

ODBC(Open Database Connectivity，开放数据库互连)是微软公司开放服务结构(WOSA，Windows Open Services Architecture)中有关数据库的一个组成部分，它建立了一组规范，并提供了一组对数据库访问的标准API(应用程序编程接口)。这些 API 利用 SQL 来完成其大部分任务。ODBC 本身也提供了对 SQL 语言的支持，用户可以直接将 SQL 语句送给 ODBC。

一个基于 ODBC 的应用程序对数据库的操作不依赖任何 DBMS，不直接与 DBMS 打交道，所有的数据库操作由对应的 DBMS 的 ODBC 驱动程序完成。也就是说，不论是 FoxPro、Access 还是 Oracle 数据库，均可用 ODBC API 进行访问。由此可见，ODBC 的最大优点是能以统一的方式处理所有的数据库。

一个完整的 ODBC 由下列几个部件组成：

应用程序(Application)。

ODBC 管理器(Administrator)。该程序位于 Windows 95 控制面板(Control Panel)的 32 位 ODBC 内，其主要任务是管理安装的 ODBC 驱动程序和管理数据源。

驱动程序管理器(Driver Manager)。驱动程序管理器包含在 ODBC32.DLL 中，对用户是透明的。其任务是管理 ODBC 驱动程序，是 ODBC 中最重要的部件。

ODBC API。

ODBC 驱动程序。是一些 DLL，提供了 ODBC 和数据库之间的接口。

数据源。数据源包含了数据库位置和数据库类型等信息，实际上是一种数据连接的抽象。

各部件之间的关系如图 10.2 所示

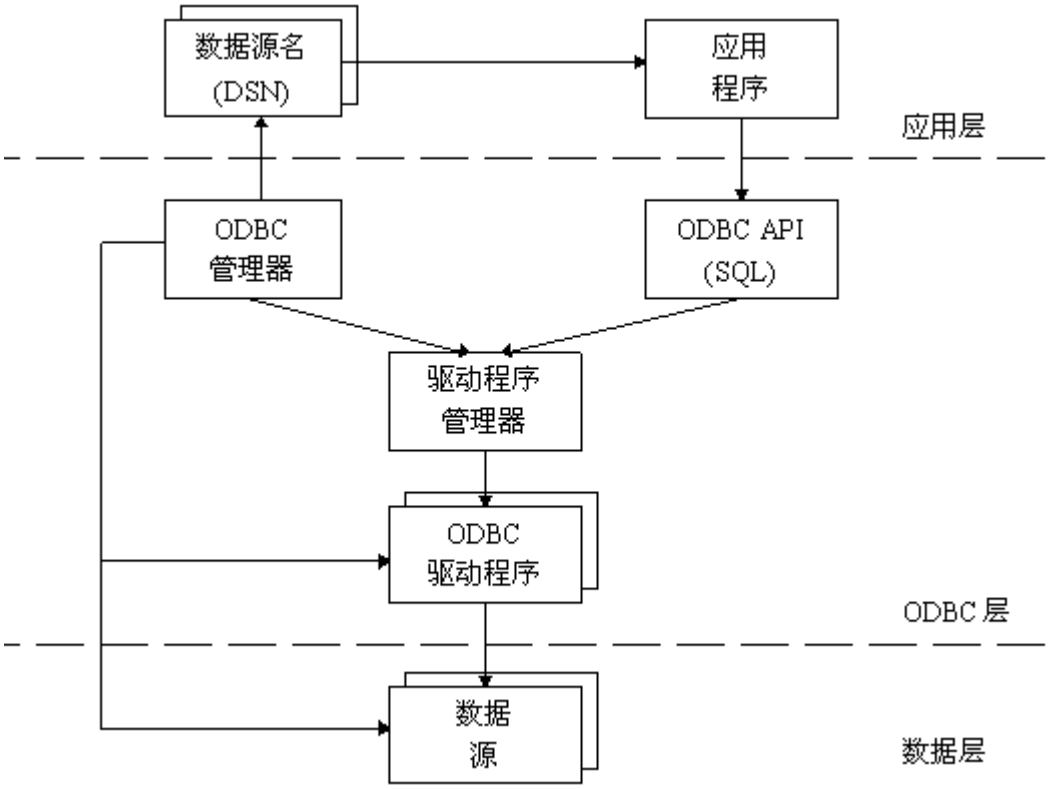


图 10.2 ODBC **部件关系图**

应用程序要访问一个数据库，首先必须用 ODBC **管理器注册一个数据源，管理器根据数据源提供的数据库位置、数据库类型及** ODBC **驱动程序等信息，建立起** ODBC **与具体数据库的联系。这样，只要应用程序将数据源名提供给** ODBC，ODBC **就能建立起与相应数据库的连接。**

在 ODBC **中，**ODBC API **不能直接访问数据库，必须通过驱动程序管理器与数据库交换信息。驱动程序管理器负责将应用程序对** ODBC API **的调用传递给正确的驱动程序，而驱动程序在执行完相应的操作后，将结果通过驱动程序管理器返回给应用程序。**

在访问 ODBC **数据源时需要** ODBC **驱动程序的支持。用** Visual C++ 5.0 **安装程序可以安装** SQL Server、 Access、 Paradox、 dBase、 FoxPro、 Excel、 Oracle **和** Microsoft Text **等驱动程序．在缺省情况下，**VC5.0 **只会安装** SQL Server、 Access、 FoxPro **和** dBase **的驱动程序．如果用户需要安装别的驱动程序，则需要重新运行** VC 5.0 **的安装程序并选择所需的驱动程序．**

上一页/下一页/

电脑报首页网络学院首页

本教程由 Visual C++**王朝**(Where programmers come together)**协助制作未经许可，请勿以任何形式复制/**

10.3 MFC 的 ODBC 类简介

MFC 的 ODBC 类对较复杂的 ODBC API 进行了封装，提供了简化的调用接口，从而大大方便了数据库应用程序的开发。程序员不必了解 ODBC API 和 SQL 的具体细节，利用 ODBC 类即可完成对数据库的大部分操作。

MFC 的 ODBC 类主要包括：

CDatabase 类：主要功能是建立与数据源的连接。

CRecordset 类：该类代表从数据源选择的一组记录(记录集)，程序可以选择数据源中的某个表作为一个记录集，也可以通过对表的查询得到记录集，还可以合并同一数据源中多个表的列到一个记录集中．通过该类可对记录集中的记录进行滚动、修改、增加和删除等操作。

CRecordView 类：提供了一个表单视图与某个记录集直接相连，利用对话框数据交换机制(DDX)在记录集与表单视图的控件之间传输数据。该类支持对记录的浏览和更新，在撤销时会自动关闭与之相联系的记录集。

CFieldExchange 类：支持记录字段数据交换(DFX)，即记录集字段数据成员与相应的数据库的表的字段之间的数据交换。该类的功能与 CDataExchange 类的对话框数据交换功能类似。

CDBException 类：代表 ODBC 类产生的异常。

概括地讲，CDatabase 针对某个数据库，它负责连接数据源；CRecordset 针对数据源中的记录集，它负责对记录的操作；CRecordView 负责界面，而 CFieldExchange 负责 CRecordset 与数据源的数据交换。

利用 AppWizard 和 ClassWizard，用户可以方便地建立数据库应用程序，但这并不意味着可以对 MFC 的 ODBC 类一无所知．读者应注意阅读后面几小节中的内容，为学习后面的例子打好基础．

10.4 CDatabase 类

要建立与数据源的连接，首先应构造一个 CDatabase 对象，然后再调用 CDatabase 的 Open 成员函数．Open 函数负责建立连接，其声明为

virtual BOOL Open(LPCTSTR lpszDSN, BOOL bExclusive = FALSE, BOOL bReadOnly = FALSE, LPCTSTR lpszConnect = “ODBC;”, BOOL bUseCursorLib = TRUE); throw(CDBException, CMemoryException);

参数 lpszDSN 指定了数据源名(构造数据源的方法将在后面介绍)，在 lpszConnect 参数中也可包括数据源名，此时 lpszDSN 必需为 NULL，若在函数中未提供数据源名且使 lpszDSN 为 NULL，则会显示一个数据源对话框，用户可以在该对话框中选择一个数据源．参数 bExclusive 说明是否独占数据源，由于目前版本的类库还不支持独占方式，故该参数的值应该是 FALSE，这说明数据源是被共享的．参数 bReadOnly 若为 TRUE 则对数据源的连接是只读的．参数 lpszConnect 指定了一个连接字符串，连接字符串中可以包括数据源名、用户帐号(ID)和口令等信息，字符串中的"ODBC"表示要连接到一个 ODBC 数据源上．参数 bUseCursorLib 若为 TRUE，则会装载光标库，否则不装载，快照需要光标库，动态集不需要光标库． 若连接成功，函数返回 TRUE，若返回 FALSE，则说明用户在数据源对话框中按了 Cancel 按钮。若函数内部出现错误，则框架会产生一个异常。

下面是一些调用 Open 函数的例子。

```
CDatabase m_db; //在文档类中嵌入一个 CDatabase 对象
//连接到一个名为"Student Registration"的数据源
m_db.Open(“Student Registration”);
//在连接数据源的同时指定了用户帐号和口令
m_db.Open(NULL,FALSE,FALSE,”ODBC;DSN=Student Registration;UID=ZYF;PWD=1234”);
m_db.Open(NULL); //将弹出一个数据源对话框
```

要从一个数据源中脱离，可调用函数 Close。在脱离后，可以再次调用 Open 函数来建立一个新的连接．调用 IsOpen 可判断当前是否有一个连接，调用 GetConnect 可返回当前的连接字符串。函数的声明为

```
virtual void Close( );
BOOL IsOpen( ) const; //返回 TRUE 则表明当前有一个连接
const CString& GetConnect( ) const;
```

CDatabase 的析构函数会调用 Close，所以只要删除了 CDatabase 对象就可以与数据源脱离。

10.5 CRecordset 类

CRecordset 类代表一个记录集．该类是 MFC 的 ODBC 类中最重要、功能最强大的类。

10.5.1 动态集、快照、光标和光标库

在多任务操作系统或网络环境中，多个用户可以共享同一个数据源。共享数据的一个主要问题是如何协调各个用户对数据源的修改。例如，当某一个应用改变了数据源中的记录时，别的连接至该数据源的应用应该如何处理。对于这个问题，基于 MFC 的 ODBC 应用程序可以采取几种不同的处理办法，这将由程序采用哪种记录集决定。

记录集主要分为快照(Snapshot) 和动态集(Dynaset)两种，CRecordset 类对这两者都支持。这两种记录集的不同表现在它们对别的应用改变数据源记录采取了不同的处理方法。

快照型记录集提供了对数据的静态视．快照是个很形象的术语，就好象对数据源的某些记录照了一张照片一样．当别的用户改变了记录时(包括修改、添加和删除)，快照中的记录不受影响，也就是说，快照不反映别的用户对数据源记录的改变．直到调用了 CRecordset::Requery 重新查询后，快照才会反映变化．对于象产生报告或执行计算这样的不希望中途变动的工作，快照是很有用的。需要指出的是，快照的这种静态特性是相对于别的用户而言的，它会正确反映由本身用户对记录的修改和删除，但对于新添加的记录直到调用 Requery 后才能反映到快照中．

动态集提供了数据的动态视．当别的用户修改或删除了记录集中的记录时，会在动态集中反映出来：当滚动到修改过的记录时对其所作的修改会立即反映到动态集中，当记录被删除时，MFC 代码会跳过记录集中的删除部分．对于其它用户添加的记录，直到调用 Requery 时，才会在动态集中反映出来。本身应用程序对记录的修改、添加和删除会反映在动态集中。当数据必须是动态的时侯，使用动态集是最适合的。例如，在一个火车票联网售票系统中，显然应该用动态集随时反映出共享数据的变化。

在记录集中滚动，需要有一个标志来指明滚动后的位置(当前位置)。ODBC 驱动程序会维护一个光标，用来跟踪记录集的当前记录，可以把光标理解成跟踪记录集位置的一种机制。

光标库(Cursor Library)是处于 ODBC 驱动程序管理器和驱动程序之间的动态链接库(ODBCCR32.DLL)．光标库的主要功能是支持快照以及为底层驱动程序提供双向滚动能力，高层次的驱动程序不需要光标库，因为它们是可滚动的．光标库管理快照记录的缓冲区，该缓冲区反映本程序对记录的修改和删除，但不反映其它用户对记录的改变，由此可见，快照实际上相当于当前的光标库缓冲区．

应注意的是，快照是一种静态光标(Static Cursor)．静态光标直到滚动到某个记录才能取得该记录的数据．因此，要保证所有的记录都被快照，可以先滚动到记录集的末尾，然后再滚动到感兴趣的第一个记录上．这样做的缺点是滚动到末尾需要额外的开销，会降低性能．

与快照不同，动态集不用光标库维持的缓冲区来存放记录．实际上，动态集是不使用光标库的，因为光标库会屏蔽掉一些支持动态集的底层驱动程序功能．动态集是一种键集驱动光标(Keyset-Driven Cursor)，当打开一个动态集时，驱动程序保存记录集中每个记录的键．只要光标在动态集中滚动，驱动程序就会通过键来从数据源中检取当前记录，从而保证选取的记录与数据源同步．

从上面的分析中可以看出，快照和动态集有一个共同的特点，那就是在建立记录集后，记录集中的成员就已经确定了．这就是为什么两种记录集都不能反映别的用户添加记录的原因．

10.5.2 域数据成员与数据交换

CRecordset 类代表一个记录集．用户一般需要用 ClassWizard 创建一个 CRecordset 的派生类．ClassWizard 可以为派生的记录集类创建一批数据成员，这些数据成员与记录的各字段相对应，被称为字段数据成员或域数据成员．例如，对于表 10.2 所示的将在后面例子中使用的数据库表，ClassWizard 会在派生类中加入 6 个域数据成员，如清单 10.1 所示．可以看出域数据成员与表中的字段名字类似，且类型匹配．

表 10.2　stdreg32.mdb 中的 Section 表

CourseID(Text)	SectionNo(Text)	InstructorID (Text)	RoomNo (Text)	Schedule (Text)	Capacity (int)
MATH101	1	KLAUSENJ	KEN-12	MWF10-11	40
MATH101	2	ROGERSN	WIL-1088	TTH3:30-5	15
MATH201	1	ROGERSN	WIL-1034	MWF2-3	20
MATH201	2	SMITHJ	WIL-1054	MWF3-4	25
MATH202	1	KLA	WIL-1054	MWF9-10	20
MATH202	2	ROGERSN	KEN-12	TTH9:30-11	15
MATH202	3	KLAUSENJ	WIL-2033	TTH3-4:30	15

清单 10.1　派生类中的域数据成员

```
class CSectionSet : public CRecordset
{
public:
. . . . . .
//{{AFX_FIELD(CSectionSet, CRecordset)
CString m_CourseID;
CString m_SectionNo;
CString m_InstructorID;
CString m_RoomNo;
CString m_Schedule;
```

```
int m_Capacity;
//}}AFX_FIELD
. . . . . .
};
```

域数据成员用来保存某条记录的各个字段，它们是程序与记录之间的缓冲区．域数据成员代表当前记录，当在记录集中滚动到某一记录时，框架自动地把记录的各个字段拷贝到记录集对象的域数据成员中．当用户要修改当前记录或增加新记录时，程序先将各字段的新值放入域数据成员中，然后调用相应的 CRecordset 成员函数把域数据成员设置到数据源中．

不难看出，在记录集与数据源之间有一个数据交换问题．CRecordset 类使用＂记录域交换＂(Record Field Exchange，缩写为 RFX)机制自动地在域数据成员和数据源之间交换数据．RFX 机制与对话数据交换(DDX)类似．CRecordset 的成员函数 DoFieldExchange 负责数据交换任务，在该函数中调用了一系列 RFX 函数．当用户用 ClassWizard 加入域数据成员时，ClassWizard 会自动在 DoFieldExchange 中建立 RFX．典型 DoFieldExchange 如清单 10.2 所示：

清单 10.2 典型的 DoFieldExchange 函数

```
void CSectionSet::DoFieldExchange(CFieldExchange* pFX)
{
//{{AFX_FIELD_MAP(CSectionSet)
pFX->SetFieldType(CFieldExchange::outputColumn);
RFX_Text(pFX, _T("[CourseID]"), m_CourseID);
RFX_Text(pFX, _T("[SectionNo]"), m_SectionNo);
RFX_Text(pFX, _T("[InstructorID]"), m_InstructorID);
RFX_Text(pFX, _T("[RoomNo]"), m_RoomNo);
RFX_Text(pFX, _T("[Schedule]"), m_Schedule);
RFX_Int(pFX, _T("[Capacity]"), m_Capacity);
//}}AFX_FIELD_MAP
}
```

10.5.3 SQL 查询

记录集的建立实际上主要是一个查询过程，SQL 的 SELECT 语句用来查询数据源．在建立记录集时，CRecordset 会根据一些参数构造一个 SELECT 语句来查询数据源，并用查询的结果创建记录集．明白这一点对理解 CRecordset 至关重要．SELECT 语句的句法如下：

```
SELECT rfx-field-list FROM table-name [WHERE m_strFilter]
[ORDER BY m_strSort]
```

其中 table-name 是表名，rfx-field-list 是选择的列(字段)．WHERE 和 ORDER BY 是两个子句，分别用来过滤和排序。下面是 SELECT 语句的一些例子：

```
SELECT CourseID, InstructorID FROM Section
SELECT * FROM Section WHERE CourseID=‘MATH202’ AND
```

Capacity=15

SELECT InstructorID FROM Section ORDER BY CourseID ASC

其中第一个语句从 Section 表中选择 CourseID 和 InstructorID 字段．第二个语句从 Section 表中选择 CourseID 为 MATH202 且 Capacity 等于 15 的记录，在该语句中使用了象＂AND＂或＂OR＂这样的逻辑连接符．要注意在 SQL 语句中引用字符串、日期或时间等类型的数据时要用单引号括起来，而数值型数据则不用．第三个语句从 Section 表中选择 InstructorID 列并且按 CourseID 的升序排列，若要降序排列，可使用关键字 DESC．

提示：如果列名或表名中包含有空格，则必需用方括号把该名称包起来。例如，如果有一列名为‘ Client Name ”，则应该写成‘ [Client Name] ”。/

10.5.4 记录集的建立和关闭

要建立记录集，首先要构造一个 CRecordset 派生类对象，然后调用 Open 成员函数查询数据源中的记录并建立记录集．在 Open 函数中，可能会调用 GetDefaultConnect 和 GetDefaultSQL 函数．函数的声明为

CRecordset(CDatabase* pDatabase = NULL);参数 pDatabase 指向一个 CDatabase 对象，用来获取数据源．如果 pDatabase 为 NULL，则会在 Open 函数中自动构建一个 CDatabase 对象．如果 CDatabase 对象还未与数据源连接，那么在 Open 函数中会建立连接，连接字符串(参见 10.3.1)由成员函数 GetDefaultConnect 提供．

virtual CString GetDefaultConnect( );该函数返回缺省的连接字符串．Open 函数在必要的时侯会调用该函数获取连接字符串以建立与数据源的连接．一般需要在 CRecordset 派生类中覆盖该函数并在新版的函数中提供连接字符串．

virtual BOOL Open(UINT nOpenType = AFX_DB_USE_DEFAULT_TYPE, LPCTSTR lpszSQL = NULL, DWORD dwOptions = none);throw(CDBException, CMemoryException);该函数使用指定的 SQL 语句查询数据源中的记录并按指定的类型和选项建立记录集．参数 nOpenType 说明了记录集的类型，如表 10.3 所示，如果要求的类型驱动程序不支持，则函数将产生一个异常．参数 lpszSQL 是一个 SQL 的 SELECT 语句，或是一个表名．函数用 lpszSQL 来进行查询，如果该参数为 NULL，则函数会调用 GetDefaultSQL 获取缺省的 SQL 语句．参数 dwOptions 可以是一些选项的组合，常用的选项在表 10.4 中列出．若创建成功则函数返回 TRUE，若函数调用了 CDatabase::Open 且返回 FALSE，则函数返回 FALSE．

表 10.3 记录集的类型

类型	含义
AFX_DB_USE_DEFAULT_TYPE	使用缺省值．
CRecordset::dynaset	可双向滚动的动态集．
CRecordset::snapshot	可双向滚动的快照．
CRecordset::dynamic	提供比动态集更好的动态特性，大部分

	ODBC 驱动程序不支持这种记录集．
CRecordset::forwardOnly	只能前向滚动的只读记录集．

表 10.4　创建记录集时的常用选项

选项	含义
CRecordset::none	无选项(缺省)．
CRecordset::appendOnly	不允许修改和删除记录，但可以添加记录．
CRecordset::readOnly	记录集是只读的．
CRecordset::skipDeletedRecords	有些数据库(如 FoxPro)在删除记录时并不真删除，而是做个删除标记，在滚动时将跳过这些被删除的记录．

virtual CString GetDefaultSQL( );Open

函数在必要时会调用该函数返回缺省的 SQL 语句或表名以查询数据源中的记录．一般需要在 CRecordset 派生类中覆盖该函数并在新版的函数中提供 SQL 语句或表名．下面是一些返回字符串的例子．“ Section ”// 选择 Section 表中的所有记录到记录集中‘ Section, Course ”//合并 Section 表和 Course 表的各列到记录集中

//对 Section 表中的所有记录按 CourseID 的升序进行排序，然后建立记录集

“ SELECT * FROM Section ORDER BY CourseID ASC ”

上面的例子说明，通过合理地安排 SQL 语句和表名，Open 函数可以十分灵活地查询数据源中的记录．用户可以合并多个表的字段，也可以只选择记录中的某些字段，还可以对记录进行过滤和排序．

上一小节说过，在建立记录集时，CRecordset 会构造一个 SELECT 语句来查询数据源．如果在调用 Open 时只提供了表名，那么 SELECT 语句还缺少选择列参数 rfx-field-list(参见 10.5.3)．框架规定，如果只提供了表名，则选择列的信息从 DoFieldExchange 中的 RFX 语句里提取．例如，如果在调用 Open 时只提供了 " Section " 表名，那么将会构造如下一个 SELECT 语句：

SELECT CourseID,SectionNo,InstructorID,RoomNo, Schedule,Capacity FROM Section

建立记录集后，用户可以随时调用 Requery 成员函数来重新查询和建立记录集．Requery 有两个重要用途：

使记录集能反映用户对数据源的改变(参见 10.5.1)．

按照新的过滤或排序方法查询记录并重新建立记录集．

在调用 Requery 之前，可调用 CanRestart 来判断记录集是否支持 Requery 操作．要记住 Requery 只能在成功调用 Open 后调用，所以程序应调用 IsOpen 来判断记录集是否已建立．函数的声明为

virtual BOOL Requery( );throw(CDBException, CMemoryException);

返回 TRUE 表明记录集建立成功，否则返回 FALSE．若函数内部出错则产生异常．

BOOL CanRestart( ) const; //若支持 Requery 则返回 TRUE

BOOL IsOpen( ) const; //若记录集已建立则返回 TRUE

CRecordset 类有两个公共数据成员 m_strFilter 和 m_strSort 用来设置对记录的过滤和排序．在调用 Open 或 Requery 前，如果在这两个数据成员中指定了过滤或排序，那么 Open 和 Requery 将按这两个数据成员指定的过滤和排序来查询数据源．

成员 m_strFilter 用于指定过滤器．m_strFilter 实际上包含了 SQL 的 WHERE 子句的内容，但它不含 WHERE 关键字．使用 m_strFilter 的一个例子为：

m_pSet->m_strFilter=“ CourseID= ‘ MATH101 ’”; //只选择 CourseID 为 MATH101 的记录

if(m_pSet->Open(CRecordset::snapshot, “ Section ”))

. . . . . .

成员 m_strSort 用于指定排序．m_strSort 实际上包含了 ORDER BY 子句的内容，但它不含 ORDER BY 关键字．m_strSort 的一个例子为

m_pSet->m_strSort=“ CourseID DESC ”; //按 CourseID 的降序排列记录

m_pSet->Open();

. . . . . .

事实上，Open 函数在构造 SELECT 语句时，会把 m_strFilter 和 m_strSort 的内容放入 SELECT 语句的 WHERE 和 ORDER BY 子句中．如果在 Open 的 lpszSQL 参数中已包括了 WHERE 和 ORDER BY 子句，那么 m_strFilter 和 m_strSort 必需为空．

调用无参数成员函数 Close 可以关闭记录集．在调用了 Close 函数后，程序可以再次调用 Open 建立新的记录集．CRecordset 的析构函数会调用 Close 函数，所以当删除 CRecordset 对象时记录集也随之关闭。

10.5.5 滚动记录

CRecordset 提供了几个成员函数用来在记录集中滚动，如下所示．当用这些函数滚动到一个新记录时，框架会自动地把新记录的内容拷贝到域数据成员中．

void MoveNext( ); //前进一个记录

void MovePrev( ); //后退一个记录

void MoveFirst( ); //滚动到记录集中的第一个记录

void MoveLast( ); //滚动到记录集中的最后一个记录

void SetAbsolutePosition(long nRows);该函数用于滚动到由参数 nRows 指定的绝对位置处．若 nRows 为负数，则从后往前滚动．例如，当 nRows 为-1 时，函数就滚动到记录集的末尾．注意，该函数不会跳过被删除的记录．

virtual void Move(long nRows, WORD wFetchType = SQL_FETCH_RELATIVE);该函数功能强大．通过将 wFetchType 参数指定为 SQL_FETCH_NEXT、SQL_FETCH_PRIOR、SQL_FETCH_FIRST、SQL_FETCH_LAST 和 SQL_FETCH_ABSOLUTE，可以完成上面五个函数的功能．若 wFetchType 为 SQL_FETCH_RELATIVE，那么将相对当前

记录移动，若 nRows 为正数，则向前移动，若 nRows 为负数，则向后移动．

如果在建立记录集时选择了 CRecordset::skipDeletedRecords 选项，那么除了 SetAbsolutePosition 外，在滚动记录时将跳过被删除的记录，这一点对象 FoxPro 这样的数据库十分重要．

如果记录集是空的，那么调用上述函数将产生异常．另外，必须保证滚动没有超出记录集的边界．调用 IsEOF 和 IsBOF 可以进行这方面的检测．

BOOL IsEOF( ) const;如果记录集为空或滚动过了最后一个记录，那么函数返回 TRUE，否则返回 FALSE．

BOOL IsBOF( ) const;如果记录集为空或滚动过了第一个记录，那么函数返回 TRUE，否则返回 FALSE．

下面是一个使用 IsEOF 的例子：

```
while(!m_pSet->IsEOF( ))
m_pSet->MoveNext( );
```

调用 GetRecordCound 可获得记录集中的记录总数，该函数的声明为

long GetRecordCount( ) const;要注意这个函数返回的实际上是用户在记录集中滚动的最远距离．要想真正返回记录总数，只有调用 MoveNext 移动到记录集的末尾(MoveLast 不行)．

10.5.6 修改、添加和删除记录

要修改当前记录，应该按下列步骤进行：

调用 Edit 成员函数．调用该函数后就进入了编辑模式，程序可以修改域数据成员．注意不要在一个空的记录集中调用 Edit，否则会产生异常．Edit 函数会把当前域数据成员的内容保存在一个缓冲区中，这样做有两个目的，一是可以与域数据成员作比较以判断哪些字段被改变了，二是在必要的时侯可以恢复域数据成员原来的值．若再次调用 Edit，则将从缓冲区中恢复域数据成员，调用后程序仍处于编辑模式．调用 Move(AFX_MOVE_REFRESH) 或 Move(0) 可退出编辑模式(AFX_MOVE_REFRESH 的值为 0)，同时该函数会从缓冲区中恢复域数据成员．

设置域数据成员的新值．

调用 Update 完成编辑．Update 把变化后的记录写入数据源并结束编辑模式．

要向记录集中添加新的记录，应该按下列步骤进行：

调用 AddNew 成员函数．调用该函数后就进入了添加模式，该函数把所有的域数据成员都设置成 NULL(注意，在数据库术语中，NULL 是指没有值，这与 C++的 NULL 是不同的)．与 Edit 一样，AddNew 会把当前域数据成员的内容保存在一个缓冲区中，在必要的时侯，程序可以再次调用 AddNew 取消添加操作并恢复域数据成员原来的值，调用后程序仍处于添加模式．调用 Move(AFX_MOVE_REFRESH)可退出添加模式，同时该函数会从缓冲区中恢复域数据成员．

设置域数据成员．

调用 Update．Update **把域数据成员中的内容作为新记录写入数据源，从而结束了添加．**

如果记录集是快照，那么在添加一个新的记录后，需要调用 Requery **重新查询，因为快照无法反映添加操作．**

要删除记录集的当前记录，应按下面两步进行：

调用 Delete **成员函数．该函数会同时给记录集和数据源中当前记录加上删除标记．注意不要在一个空记录集中调用** Delete**，否则会产生一个异常．**

滚动到另一个记录上以跳过删除记录．

上面提到的函数声明为：

virtual void Edit( );throw(CDBException, CMemoryException);

virtual void AddNew( );throw(CDBException);

virtual void Delete( );throw(CDBException);

virtual BOOL Update( );throw(CDBException);

若更新失败则函数返回 FALSE**，且会产生一个异常．**

在对记录集进行更改以前，程序也许要调用下列函数来判断记录集是否是可以更改的，因为如果在不能更改的记录集中进行修改、添加或删除将导致异常的产生．

BOOL CanUpdate( ) const; //**返回** TRUE **表明记录是可以修改、添加和删除的．**

BOOL CanAppend( ) const; //**返回** TRUE **则表明可以添加记录．**

10.6 CRecordView 类

CRecordView(记录视图)是 CFormView 的派生类，它提供了一个表单视图(参见 6.4.1)来显示当前记录.一个典型的记录视图如图 10.3 所示，用户可以通过表单视图显示当前记录．通过记录视图，可以修改、添加和删除数据．用户一般需要创建一个 CRecordView 的派生类并在其对应的对话框模板中加入控件.

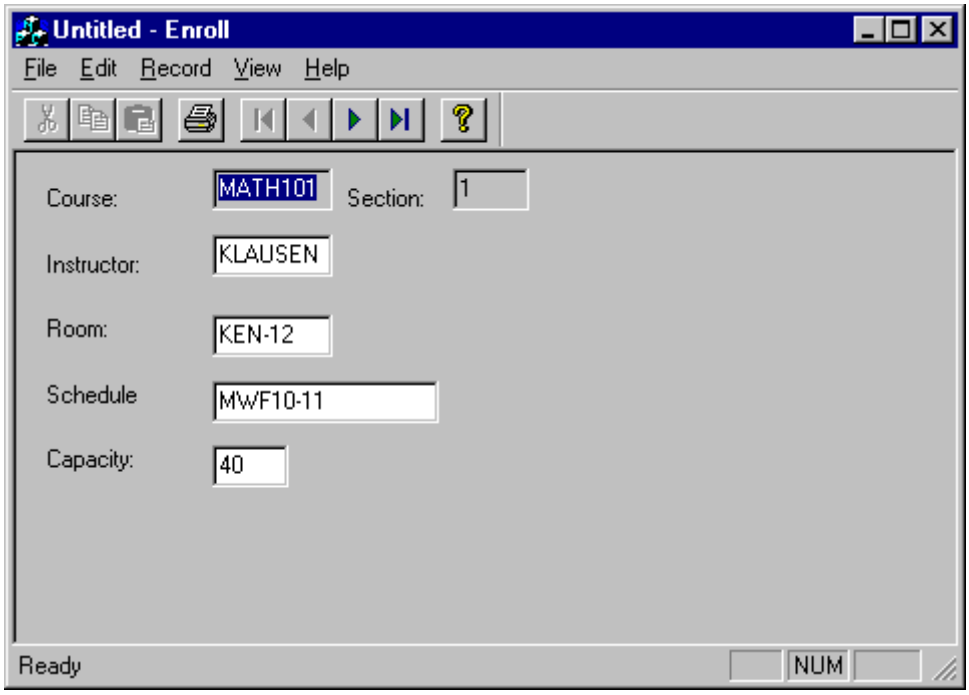


图 10.3 典型的记录视图

记录视图使用 DDX 数据交换机制在表单中的控件和记录集之间交换数据。在前面介绍的 DDX 都是在控件和控件父窗口的数据成员之间交换数据，而记录视图则是在控件和一个外部对象(CRecordset 的派生类对象)之间交换数据．清单 10.3 显示了一个 CRecordView 的派生类的 DoDataExchange 函数，读者可以看出，该函数是与 m_pSet 指针指向的记录集对象的域数据成员交换数据的，而且，交换数据的代码是 ClassWizard 自动加入的.在后面的例子中，将向读者介绍用 ClassWizard 连接记录视图与记录集对象的方法.

清单 10.3 用来与记录集对象的域数据成员交换数据的 DoDataExchange 函数

```
void CSectionForm::DoDataExchange(CDataExchange* pDX)
{
CRecordView::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CSectionForm)
DDX_FieldText(pDX, IDC_COURSE, m_pSet->m_CourseID, m_pSet);
DDX_FieldText(pDX, IDC_SECTION, m_pSet->m_SectionNo, m_pSet);
DDX_FieldText(pDX, IDC_INSTRUCTOR, m_pSet->m_InstructorID,
```

m_pSet);

DDX_FieldText(pDX, IDC_ROOM, m_pSet->m_RoomNo, m_pSet);

DDX_FieldText(pDX, IDC_SCHEDULE, m_pSet->m_Schedule, m_pSet);

DDX_FieldText(pDX, IDC_CAPACITY, m_pSet->m_Capacity, m_pSet);

//}}AFX_DATA_MAP

}

作为总结，图 10.4 **显示了** MFC **的** ODBC **应用程序中的** DDX **和** RFX **数据交换．**

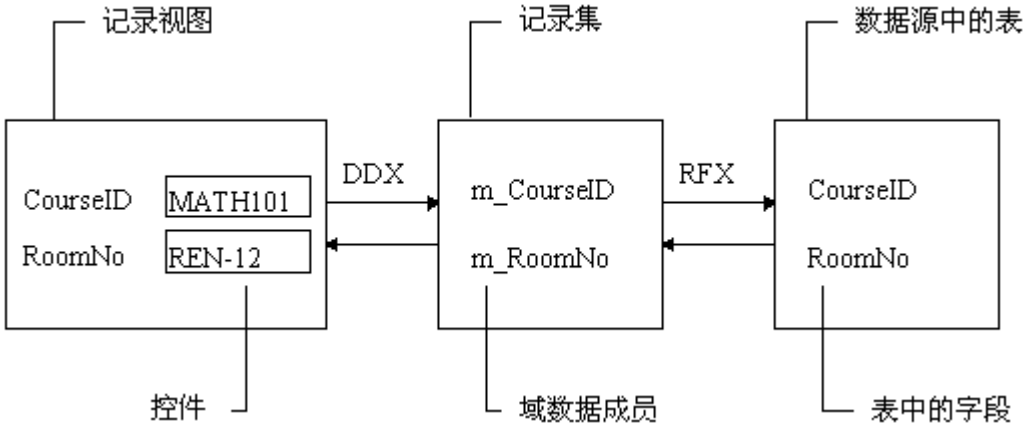


图 10.4 DDX **和** RFX **数据交换机制**

CRecordView **本身提供了对下面四个命令的支持：**

ID_RECORD_FIRST //**滚动到记录集的第一个记录**

ID_RECORD_LAST //**滚动到记录集的最后一个记录**

ID_RECORD_NEXT //**前进一个记录**

ID_RECORD_PREV //**后退一个记录**

CRecordView **提供了** OnMove **成员函数处理这四个命令消息，**OnMove **函数对用户是透明的，清单** 10.4 **列出了** OnMove **的源代码．**

清单 10.4 OnMove **函数**

```
BOOL CRecordView::OnMove(UINT nIDMoveCommand)
{
CRecordset* pSet = OnGetRecordset();
if (pSet->CanUpdate())
{
pSet->Edit();
if (!UpdateData())
return TRUE;
pSet->Update();
}
switch (nIDMoveCommand)
{
case ID_RECORD_PREV:
```

```
pSet->MovePrev();
if (!pSet->IsBOF())
break;
case ID_RECORD_FIRST:
pSet->MoveFirst();
break;
case ID_RECORD_NEXT:
pSet->MoveNext();
if (!pSet->IsEOF())
break;
if (!pSet->CanScroll())
{
// clear out screen since we're sitting on EOF
pSet->SetFieldNull(NULL);
break;
}
case ID_RECORD_LAST:
pSet->MoveLast();
break;
default:
// Unexpected case value
ASSERT(FALSE);
}
// Show results of move operation
UpdateData(FALSE);
return TRUE;
}
```

在函数的开头先调用 CRecordset::Edit **进入编辑模式，接着调用** UpdateData **将控件中的数据更新到记录集对象的域数据成员中，然后调用** CRecordset::Update **将域数据成员的值写入数据源．这说明** OnMove **在滚动记录的同时会完成对原来记录的修改．**

在函数的中间有一个分支语句用来处理四个不同的命令，在这个分支语句中调用了 CRecordset **的各种用于滚动记录的成员函数，这些函数在滚动到一个新的记录时会把该记录的内容设置到域数据成员中．在函数的末尾调用** UpdateData(FALSE)**把新的当前记录的内容设置到表单的控件中。**

由此可见，OnMove **一来一回完成了两次表单控件和数据源的数据交换过程．通过分析该函数，读者可以学会在浏览记录时如何控制** DDX **和** DFX **数据交换．**

10.7　学习 Enroll 例程

Visual C++提供了一个名为 Enroll 的例子来作为学习 MFC 数据库编程的教程．Enroll 分为四步，本节的任务就是指导读者完成前三步的 Enroll 例程，并对其进行较彻底的剖析．通过学习这三步例程，读者将掌握用 AppWizard 和 ClassWizard 创建 MFC 数据库应用程序的方法．

在开始学习 Enroll 例程时，读者也许会感到用 AppWizard 创建数据库应用很容易，似乎不用学习前面几节的内容．诚然，AppWizard 自动地为应用程序加入了许多与数据库有关的代码，大大简化了数据库应用的开发．但 AppWizard 不是万能的，它建立的数据库应用往往不能满足用户的需要．用户真正想知道的是如何不依赖 AppWizard 而编写自己的数据库应用程序，这也正是本章的宗旨所在．事实上，前面几节的分析以及后面进行的对 Enroll 例程的分析正是为这一宗旨服务的．

在学习 Enroll 以前，请读者先从 Visual C++ 5.0 的光盘上将 Enroll(在 samples \ mfc \ tutorial \ enroll 目录下)在前三步的例程拷到硬盘上，以供参考．另外，Enroll 要用到 Access 数据库 STDRED32.MDB，该文件在 VC5.0 的 Stdreg 例程中(在 samples \ mfc \ database \ stdreg 目录下)，请读者将该例子也拷贝到硬盘上．

10.7.1　注册数据源

ODBC 应用程序不能直接使用数据库，用户必需为要使用的数据库注册数据源．注册数据源的工作由 ODBC 管理器完成，该管理器位于 Windows 95 控制面板的 32 位 ODBC 内．现在让我们为 Access 数据库 STDREG32.MDB 注册数据源．

打开控制面板，双击“32 位 ODBC”图标，则会显示一个“ODBC 数据源管理器”，如图 10.5 所示。在管理器中选择“用户 DSN”页，用户 DSN 只对用户可见而且只能用户当前机器。

图 10.5 ODBC 数据源管理器

点击“添加”按钮，则会弹出一个“创建新数据源”对话框。在该对话框中选择 Microsoft Access Driver(*.mdb)，然后按完成按钮。接下来会显示一个 ODBC Microsoft Access 97 Setup 对话框，如图 10.6 所示，该对话框用来把数据库与一个数据源名连接起来。在 Data Source Name:栏中输入 Student Registration，然后点击 Select...按钮，在随后弹出的对话框中找到并选择 STDREG32.MDB。连按两个 OK 按钮后，一个名为 Student Registration 的新数据源就被注册到了管理器中。

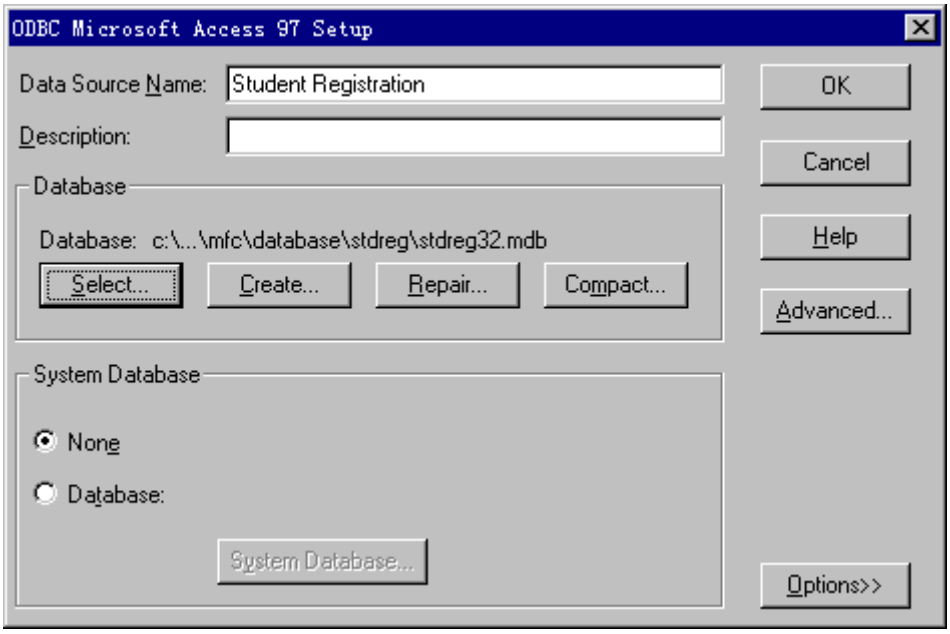


图 10.6 ODBC Microsoft Access 97 Setup 对话框

提示：如果要为 dBase 或 FoxPro 数据库注册数据源，则应该选择一个目录而不是文件作为数据源。这是因为在 Access 中，一个 MDB 文件可以包含多张表，因此可以认为一个 MDB 文件就是一个数据库。而在 dBase 或 FoxPro 中，一个 DBF 文件只能对应一张表，一个数据库可能会包含好几个 DBF 文件，所以可以认为某一个包含了若干 DBF 文件的目录是一个数据库。/

10.7.2 Enroll 的第一个版本

Enroll 第一个版本如图 10.3 所示，该程序具有浏览记录集和修改记录这两个基本功能。在修改了表单中的记录后，移动到一个新的记录上就可以保存对原记录的修改。注意在表单中 Course 和 Section 编辑框都是只读的，这是因为这两个字段的内容较重要，用户不能随便修改。Enroll 使用了 STDREG32.MDB 的 Section 表，该表是一张课程表，其内容如表 10.2 所示。

现在让我们来建立 Enroll 应用程序。首先，用 AppWizard 来完成 Enroll 程序框架的建立，请读者按下面几步进行：

启动 AppWizard，指定一个名为 Enroll 的 MFC 工程。

在 MFC AppWizard 的第一步选择 Single document。

在 MFC AppWizard 的第二步选择 Database view without file support，

然后点击 Data Source...按钮，在 Database Options 对话框中的 ODBC 组合框中选择 Student Registration 数据源，如图 10.7 所示。然后按“OK”按钮，则会打开一个 Select Database Table 对话框，如图 10.8 所示。在该对话框中选择 Section 表。按 OK 按钮退出。

在 MFC AppWizard 的第六步中，将类 CEnrollSet 改名为 CSectionSet，将类 CEnrollView 改名为 CSectionForm。

按 Finish 按钮，建立 Enroll 工程。

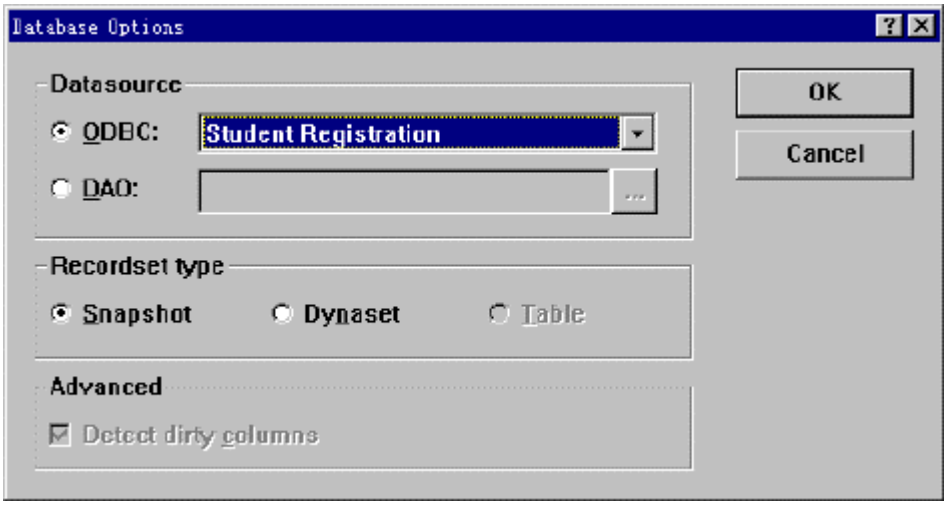


图 10.7 Database Options 对话框

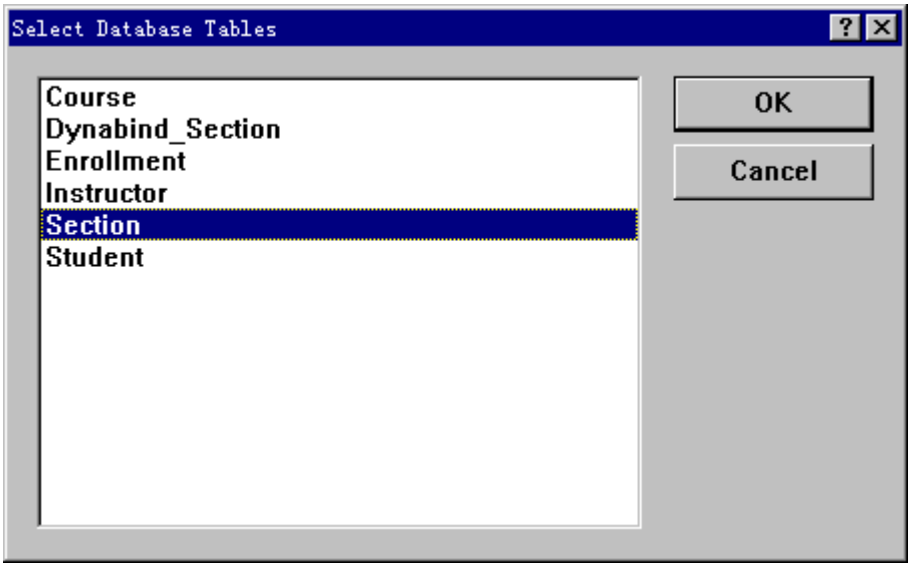


图 10.8 Select Database Table 对话框

打开工作区的类视图，可以发现 AppWizard 自动创建了一个记录集类 CSectionSet 和一个记录视图类 CSectionForm，这两个类分别是 CRecordset 和 CRecordView 的派生类。AppWizard 也为 CSectionSet 类自动创建了域数据成员。

打开工作区的资源视图，读者不难找到一个 ID 为 IDD_ENROLL_FORM 的对话框模板，该模板将被记录视图用来显示表单。清除该模板中的所有控件，并把模板的尺寸扩大到 183×110，然后按图 10.3 的式样放置控件。这里可以采取一个偷懒的方法：打开 VC5.0

已作好的第一个版本 Enroll 的资源文件(Enroll.rc)，找到并打开 IDD_ENROLL_FORM 对话框模板，按住 Ctrl 键并用鼠标选择模板中的所有控件，然后按 Ctrl+Insert 键拷贝所选的控件。切换到自己的 IDD_ENROLL_FORM 对话框模板，然后按 Shift+Insert 键把刚才拷贝的控件粘贴到模板中。

接下来的任务是用 ClassWizard 把表单中的控件与记录集的域数据成员连接起来，以实现控件与当前记录的 DDX 数据交换。请读者按如下步骤操作：

进入 ClassWizard，选择 Member Variables 页并且选择 CSectionForm 类。

在变量列表中双击 IDC_CAPACITY 项，则会显示 Add Member Variable 对话框。注意该对话框的 Member variable name 栏显示的是一个组合框，而不是平常看到的编辑框。如图 10.9 所示，在组合框的列表中选择 m_pSet->m_Capacity。按 OK 按钮确认。

仿照第 2 步，为其他控件连接记录集的域数据成员。根据控件的 ID，不难确定对应的域数据成员。

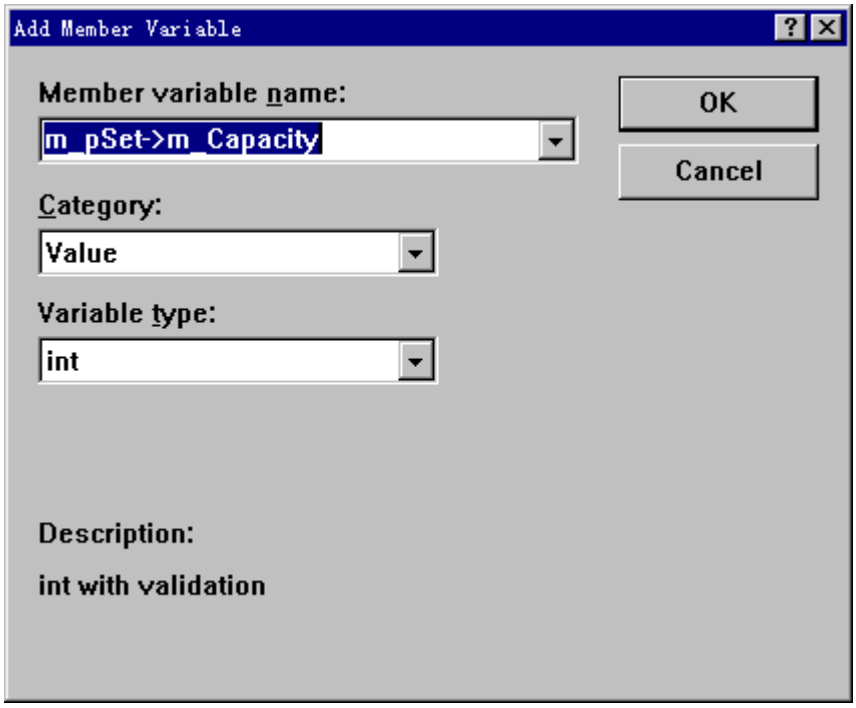


图 10.9 Add Member Variable 对话框

在 CSectionForm 类的定义内可以找到下面一行：

```
CSectionSet* m_pSet;
```

可见 m_pSet 是 CSectionForm 类的成员，它指向一个 CSectionSet 对象。用 ClassWizard 可以把控件与象记录集这样的“ 外部数据 ”连接起来，这是 ClassWizard 在数据库编程方面的一个特殊应用。

编译并运行 Enroll，读者会惊奇的发现 Enroll 居然是一个相当不错的记录浏览器，并且用户可以对记录进行修改。

现在，让我们来分析一下 AppWizard 和 ClassWizard 为 Enroll 干了哪

些事情。

在文档类 CEnrollDoc 的定义中，有如下一行：

```
CSectionSet m_sectionSet;
```

可见 AppWizard 在 CEnrollDoc 类中嵌入了一个 CSectionSet 对象。这相当于调用了构造函数 CSectionSet(NULL)，CSectionSet 类的构造函数的声明如下：CSectionSet(CDatabase* pDatabase = NULL);

函数的定义在清单 10.5 中列出。可以看出，构造函数调用了基类的构造函数，并对域数据成员进行了初始化。通过 10.5.4 我们知道，若传递 NULL 参数给 CRecordset 的构造函数，那么 CRecordset::Open 函数将自动构建一个 CDatabase 对象，并根据 CRecordset:: GetDefaultConnect 返回的连接字符串建立与数据源的连接。CSectionSet 提供了虚拟函数 GetDefaultConnect 的新版本，如清单 10.6 所示，在该函数中提供了数据源 Student Registration。

清单 10.5 CSectionSet 的构造函数

```
CSectionSet::CSectionSet(CDatabase* pdb)
: CRecordset(pdb)
{
//{{AFX_FIELD_INIT(CSectionSet)
m_CourseID = _T(“”);
m_SectionNo = _T(“”);
m_InstructorID = _T(“”);
m_RoomNo = _T(“”);
m_Schedule = _T(“”);
m_Capacity = 0;
m_nFields = 6;
//}}AFX_FIELD_INIT
m_nDefaultType = snapshot;
}
```

清单 10.6 派生类的 GetDefaultConnect 函数

```
CString CSectionSet::GetDefaultConnect()
{
return _T(“ODBC;DSN=Student Registration”);
}
```

至于记录集的建立，实际上是在 CRecordView:: OnInitialUpdate 中完成的，这部分代码对用户是透明的，这里在清单 10.7 中列出。在该函数中调用 CRecordset::Open 来建立记录集。在函数的开头调用了 OnGetRecordset 函数来获取与记录视图相连的记录集对象。CSectionForm 提供了虚拟函数 OnGetRecordset 的新版本，如清单 10.8 所示，该函数把 m_pSet 提交给调用者。至于 m_pSet 的初始化，则是在 CSectionForm::OnInitialUpdate 函数中完成的，如清单 10.9 所示。

清单 10.7 CRecordView:: OnInitialUpdate 函数

```
void CRecordView::OnInitialUpdate()
```

```
{
CRecordset* pRecordset = OnGetRecordset();
// recordset must be allocated already
ASSERT(pRecordset != NULL);
if (!pRecordset->IsOpen())
{
CWaitCursor wait;
pRecordset->Open();
}
CFormView::OnInitialUpdate();
}
```

清单 10.8 **派生类的** OnGetRecordset **函数**

```
CRecordset* CSectionForm::OnGetRecordset()
{
return m_pSet;
}
```

清单 10.9 **派生类的** OnInitialUpdate **函数**

```
void CSectionForm::OnInitialUpdate()
{
m_pSet = &GetDocument()->m_sectionSet;
CRecordView::OnInitialUpdate();
}
```

注意到在 CRecordView:: OnInitialUpdate **中调用** CRecordset::Open **时未提供任何参数，这意味着** Open **函数将从** CRecordset::GetDefaultSQL **中获取** SQL **信息。**CSectionSet **提供了虚拟函数** GetDefaultSQL **的新版本，如清单** 10.10 **所示，该函数返回了“** Section **”表名。**

清单 10.10 **派生类的** GetDefaultSQL **函数**

```
CString CSectionSet::GetDefaultSQL()
{
return _T(“[Section]”);
}
```

至于记录的滚动和修改的实现，请参看 10.6**。而与** DDX **和** DFX **有关的代码已在清单** 10.3 **和** 10.2 **中列出。**

如果读者对上面的分析还有不明白的地方，那么请再把本章的前几节内容再仔细阅读一遍。

10.7.3 Enroll **的第二个版本**

Enroll **的第二个版本向读者演示了在一个记录视图中使用两个相关联的记录集，以及记录的过滤和排序技术，该版本使读者真正接触到了关系数据库。本小节还将向读者介绍如何用** ClassWizard **建立记录集类，以及参数化记录集的方法。**

读者可以先运行 VC 5.0 **提供的** Enroll **例子的第二步看看。**Enroll **的界面有了一个变化，原来的** Course **编辑框被替换成了组合框，如图** 10.10

所示。组合框中的内容来自同一数据源的另一张表 Course 的 CourseID 字段。

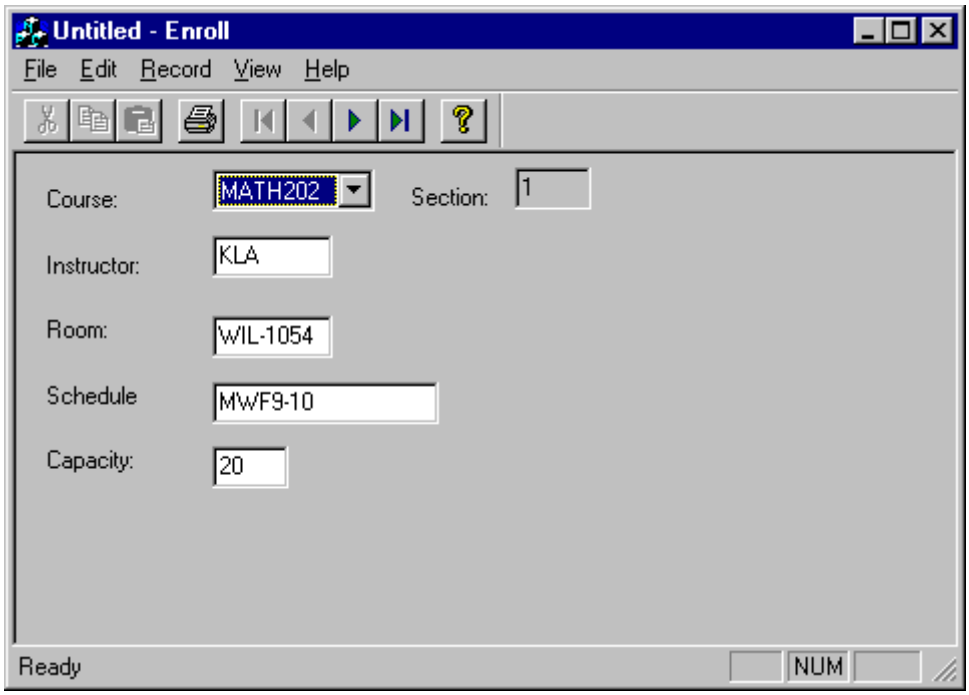


图 10.10 Enroll 的第二个版本

Course 表的内容如表 10.5 所示，与表 10.2 相对照，读者可以发现 Course 表和 Section 表有一个公共字段 CourseID。记录视图程序正是利用这个公共字段把两张表联系起来的。例如，当用户在 Course 组合框中选择了 MATH202 时，程序将选择 Section 表中所有 CourseID 为 MATH202 的记录并建立新的记录集。

事实上，在 STDREG32.MDB 的大部分表都共享了 CourseID。在主表 Course 表中，每个记录的 CourseID 是唯一的，我们称其为主关键字(Primary key)。在 Course 的相关表中，CourseID 不一定唯一，如 Section 表，我们称相关表中的 CourseID 为外关键字(Foreign key)。通过关键字可以把多张表联系到一起，这样的数据库就是关系数据库。SectionNo 也是一个关键字，在 Section 表中，SectioinNo 字段是主关键字，它的值是唯一的。

表 10.5 Course 表

CourseID(Text)	CourseTitle(Text)	Hours(int)
MATH101	Algebra	4
MATH201	CalculusI	4
MATH202	CalculusII	4

现在就让我们开始在上一小节 Enroll 的基础上制作新版本。若当前工程不是 Enroll，请读者打开上一小节创建的 Enroll 工程。

首先要把 Course 编辑框替换成组合框，这包括下面几步：

打开 IDD_ENROLL_FORM 对话框模板并删除 Course 编辑框。

在原来编辑框的位置加入一个组合框。打开该控件的属性对话框，令其 ID 为 IDC_COURSELIST，并在 Styles 页中选择 Drop List。注意要适当调整组合框的下拉列表尺寸(点击向下的箭头后扩大其尺寸)。

自上到下从新安排 Tab 顺序。

接下来要用 ClassWizard 做一些与新加的组合框有关的工作：

进入 ClassWizard，选择 Member Variables 页并选择 CSectionForm 类。单击列表中的 IDC_COURSE 项并按 Delete 键删除该项。然后双击 IDC_COURSELIST 项，在 Add Member Variable 对话框的组合框中选择 m_pSet->m_CourseID。

再次双击 IDC_COURSELIST，并为 CSectionForm 类加入一个名为 m_ctlCourseList 的 CComboBox 类成员。

选择 Message Maps 页，为 IDC_COURSELIST 组合框加入 CBN_SELENDOK 通知消息处理函数，函数名为 OnSelendokCourselist。该函数负责响应用户在组合框中选择的变化。

按 OK 按钮退出 ClassWizard。

接着，需要为 Course 表创建一个名为 CCourseSet 的记录集类，这个工作可由 ClassWizard 完成，请读者按下面几步操作：

进入 ClassWizard，点击 Add Class...按钮并在弹出的菜单中选择 New...，然后在 Create New Class 对话框中的 Name 栏中输入 CCourseSet，在 Base class 栏中选择 CRecordset，按 Create 按钮。

在弹出的 Database Options 对话框中，在 ODBC 组合框里选择 Student Registration 数据源。然后按 OK 按钮。

在弹出的 Select Database Tables 对话框中选择 Course 表。按 OK 确认。

看看新建的 CCourseSet 类，读者会发现 ClassWizard 自动为 CCourseSet 类创建了与 Course 表的字段相对应的域数据成员，并且建立了 DoFieldExchange 函数。ClassWizard 也为记录集类提供了新的 GetDefaultConnect 和 GetDefaultSQL 函数。

接着，在 CEnrollDoc 类的定义中，紧接着 m_sectionSet 成员，加入下面一行：

```
CCourseSet m_courseSet;
```

这样 CEnrollDoc 就包含了两个记录集。由于 CEnrollDoc 类用到了 CCourseSet 类，所以要在所有含有#include “EnrolDoc.h”语句的 CPP 文件中，在#include “EnrolDoc.h”语句的前面加上如下的 include 语句。这些 CPP 文件包括 CEnrollApp、CSectionForm 和 CEnrollDoc 类所在的 CPP 文件。

```
#include “CourseSet.h”
```

在 CSectionSet 类的定义中，紧接着域数据成员，在‘ //}}AFX_FIELD ”注释外加入下面一行。

```
CString m_strCourseIDParam;
```

m_strCourseIDParam 是记录集的参数数据成员，其作用将在后面说明。

最后，请读者按清单 10.11 和 10.12 修改程序。清单 10.11 列出的是 CSectionSet 类的部分源代码，清单 10.12 列出的是 CSectionForm 类的部分代码。

清单 10.11 CSectionSet **类的部分代码**

```
CSectionSet::CSectionSet(CDatabase* pdb)
: CRecordset(pdb)
{
. . .
m_nParams = 1; //只有一个参数数据成员
m_strCourseIDParam = “”;
}
void CSectionSet::DoFieldExchange(CFieldExchange* pFX)
{
. . .
pFX->SetFieldType(CFieldExchange::param);
RFX_Text(pFX, “CourseIDParam”, m_strCourseIDParam); //替换参数
}
```

清单 10.12 CSectionForm **类的部分代码**

```
void CSectionForm::OnInitialUpdate()
{
m_pSet = &GetDocument()->m_sectionSet;
CEnrollDoc* pDoc = GetDocument();
pDoc->m_courseSet.m_strSort = “CourseID”;
if (!pDoc->m_courseSet.Open())
return;
m_pSet->m_strFilter = “CourseID = “”; //使用参数
m_pSet->m_strCourseIDParam = pDoc->m_courseSet.m_CourseID;
m_pSet->m_strSort = “SectionNo”;
m_pSet->m_pDatabase = pDoc->m_courseSet.m_pDatabase; //共享
CDatabase
CRecordView::OnInitialUpdate();
m_ctlCourseList.ResetContent();
if (pDoc->m_courseSet.IsOpen())
{
while (!pDoc->m_courseSet.IsEOF())
{
m_ctlCourseList.AddString(
pDoc->m_courseSet.m_CourseID); //向表中加入 CourseID 字段
pDoc->m_courseSet.MoveNext();
}
}
m_ctlCourseList.SetCurSel(0);
}
void CSectionForm::OnSelendokCourselist()
{
```

```
if (!m_pSet->IsOpen())
return;
m_ctlCourseList.GetLBText(m_ctlCourseList.GetCurSel(),
m_pSet->m_strCourseIDParam);
m_pSet->Requery(); //重新查询
if (m_pSet->IsEOF())
{
m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
m_pSet->m_CourseID = m_pSet->m_strCourseIDParam;
}
UpdateData(FALSE);
}
```

在 CSectionForm::OnInitialUpdate 函数的开头部分，调用了 CRecordset::Open 建立 CCourseSet 记录集，在调用 Open 函数之前，指定了按 CourseID 字段排序记录集。关于调用 Open 函数的一些前因后果在前面已经解释过了，读者应该不难分析。接下来的代码读者可能就看不懂了，为什么在记录集的 m_strFilter 过滤字符串中会有一个“ “ ”号呢。

这是因为在本例中使用了“ 参数化记录集 ”技术。在记录集的 m_strFilter 和 m_strSort 中，可以用“ “ ”号作为参数使用，这样在指定过滤器和排序时可以更具灵活性。例如，在 OnInitialUpdate 函数中，是这样指定过滤器的：

```
m_pSet->m_strFilter = “CourseID = ““;
```

在调用 Open 或 Requery 时，“ “ ”将会被 CSectionSet::m_strCourseIDParam 中的内容取代。例如，如果指定 m_strCourseIDParam 为“ MATH101 ”，则 m_strFilter 将变成”CourseID = MATH101”。这样用户只要指定了 m_strCourseIDParam，就可以定制过滤器。象 m_strCourseIDParam 这样的成员被称作参数数据成员，同域数据成员一样，它是记录集所特有的。ClassWizard 不支持参数数据成员，用户只能手工加入之，且其名字由用户自己确定。

参数替换的工作实际上是由 CSectionSet::DoFieldExchange 中的 RFX 函数完成的，在 DoFieldExchange 函数的末尾，我们加入了下面两行：

```
pFX->SetFieldType(CFieldExchange::param);
RFX_Text(pFX, “CourseIDParam”, m_strCourseIDParam);
```

DoFieldExchange 可以识别域数据成员和参数数据成员。第一行调用用来表明随后的 RFX 函数用于参数替换。第二行是一个用于 m_strCourseIDParam 参数的 RFX 函数。RFX 第二个参数的名字可由用户确定，这里指定其为“ CourseIDParam ”。

用户可以在 m_strFilter 和 m_strSort 中使用一个或多个参数。在有多个参数的情况下，要注意 RFX 函数的调用次序应与参数出现的次序相对应。框架规定，用户应该先在 m_strFilter 中安排参数，然后才是 m_strSort。

CRecordset 有两个数据成员 m_nFields 和 m_nParams 分别用来统计域数据成员和参数数据成员的数目。前者由 ClassWizard 自动计数，而后

者必需由用户来维护。在 CSectionSet 的构造函数中，m_nParams 被置为 1，因为只有一个参数数据成员。

现在让我们继续研究 OnInitialUpdate。在调用基类的 OnInitialUpdate 以前，在程序中有这样一行代码：

m_pSet->m_pDatabase = pDoc->m_courseSet.m_pDatabase;

m_pDatabase 是 CRecordset 的公共成员，它是指向 CDatabase 对象的指针。如果应用程序使用了两个以上的记录集，在缺省情况下，每个记录集都会创建一个代表同一数据源的 CDatabase 对象(参见 10.5.4)，这显然没有必要。上面一行代码使 CSectionSet 记录集共享由 CCourseSet 记录集创建的 CDatabase 对象，这样，当在 CRecordView:: OnInitialUpdate 中调用 CRecordset::Open 打开 CSectionSet 记录集时，就不会再创建 CDatabase 对象了。

接下来，程序把 CCourseSet 的每一个记录的 CourseID 字段都加入到组合框中。

当用户在组合框中选择了新的 CourseID 时，在 CSectionForm::OnSelendokCourselist 中，就会把用户选择的 CourseID 设置到 m_strCourseIDParam 中，然后调用 CRecordset::Requery 按照定制的过滤器和排序重新查询和建立记录集。如果重新建立的记录集为空，则说明 Section 表中没有与指定的 CourseID 相对应的记录，这时记录集被自动设置为 NULL。在程序中，调用了 CRecordset:: SetFieldNull 把 m_CourseID 字段设置为非空。最后，调用 CRecordView::UpdateData 更新表单中的控件。

从程序中不难看出，使用参数化记录集的最大好处是可以在程序运行期间方便地指定过滤器和排序，这大大提高了程序的用户定制查询能力。

编译并运行 Enroll，试试新增加的功能。

10.7.4 Enroll 的第三个版本

Enroll 的第三个版本支持记录的添加和删除，该版本也演示了对数据库异常的处理。请读者先运行 VC5.0 提供的第三步 Enroll 程序看一看。第三版 Enroll 的 Record 菜单中多了三个命令：Add、Delete 和 Refresh，它们分别用来添加、删除和刷新记录。

当用户选择 Add 命令后，就进入了添加模式。这时除 Course 组合框外，所有的字段都被清除。用户可以在各编辑框内输入新记录的字段值，然后移动到别的记录上，这样就把新的记录保存到数据源中。用户也可以通过再次选择 Add 命令来保存新加的记录。

当用户选择 Delete 命令后，当前记录被删除，程序会自动滚动到下一个记录上。

Refresh 命令用来放弃记录的修改或添加操作，同时，恢复原记录的内容。

现在就让我们开始在上一小节 Enroll 的基础上制作新版本。若当前工程不是 Enroll，请读者打开上一小节创建的 Enroll 工程。

首先要为 Record 菜单添加三个菜单项，菜单项的各项属性在表 10.6

中列出。请把这三个菜单项加到 Record 菜单的开头，并且用一个分隔线和后面的命令隔开。

表 10.6

Caption	ID	Prompt
&Add	ID_RECORD_ADD	Add a new section
&Refresh	ID_RECORD_REFRESH	Cancel changes on form,or cancel Add
&Delete	ID_RECORD_DELETE	Delete section

接着，用 ClassWizard 为上面三个命令创建处理函数，函数名为缺省的。另外，需要为记录视图编写新的 OnMove 函数来处理滚动命令，这是因为原来的 OnMove 函数没有添加记录的功能。在 ClassWizard 的 CSectionForm 类的 Messages 列表中可以找到 OnMove 函数，请读者双击并建立该函数。

在 CSectionForm::OnMove 函数处理滚动命令时，必需要有一个标志来判断当前是否处于添加模式，以便向数据源中加入新记录或进行普通的滚动处理。请读者在 CSectionForm 类的定义中的适当位置加入下面两行：

```
protected:
BOOL m_bAddMode;
```

在前两个版本的 Enroll 中，Section 编辑框都是只读的，但在添加记录时，必需允许用户修改 Section 编辑框。这是因为 SectionNo 字段是 Section 表的主关键字，它的值必需唯一，如果在加入新记录时不改变原来的 SectionNo 字段,那么将会因为主关键字重复而导致异常产生。显然，我们需要一个 CEdit 对象来控制 IDC_SECTION 编辑框，请读者用 ClassWizard 为 CSectionForm 类加入一个与 IDC_SECTION 对应的 CEdit 型成员变量，变量的名字为 m_ctlSection。

最后，请读者按清单 10.13 修改程序。

清单 10.13 CSectionForm 类的部分源代码

```
CSectionForm::CSectionForm()
: CRecordView(CSectionForm::IDD)
{
. . .
m_bAddMode = FALSE;
}
void CSectionForm::OnSelendokCourselist()
{
if (!m_pSet->IsOpen())
return;
m_ctlCourseList.GetLBText(m_ctlCourseList.GetCurSel(),
m_pSet->m_strCourseIDParam);
if (!m_bAddMode)
{
```

```
m_pSet->Requery();
if (m_pSet->IsEOF())
{
m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
m_pSet->m_CourseID = m_pSet->m_strCourseIDParam;
}
UpdateData(FALSE);
}
}
void CSectionForm::OnRecordAdd()
{
if (m_bAddMode) //如果已处于添加模式，则完成添加操作
OnMove(ID_RECORD_FIRST);
CString strCurrentCourse = m_pSet->m_CourseID;
m_pSet->AddNew();
m_pSet->SetFieldNull(&(m_pSet->m_CourseID), FALSE);
m_pSet->m_CourseID = strCurrentCourse;
m_bAddMode = TRUE;
m_ctlSection.SetReadOnly(FALSE);
UpdateData(FALSE); //更新表单视图
}
void CSectionForm::OnRecordDelete()
{
TRY
{
m_pSet->Delete();
}
CATCH(CDBException, e)
{
AfxMessageBox(e->m_strError);
return;
}
END_CATCH
m_pSet->MoveNext(); //滚动到下一个记录
if (m_pSet->IsEOF()) //如果滚出了记录集的边界，则滚动到最后一个
记录
m_pSet->MoveLast();
if (m_pSet->IsBOF()) //如果记录变空了，则清除域数据成员
m_pSet->SetFieldNull(NULL);
UpdateData(FALSE); //更新表单视图
}
void CSectionForm::OnRecordRefresh()
```

```
{
if (m_bAddMode == TRUE)
{
m_pSet->Move(AFX_MOVE_REFRESH); //取消添加模式
m_ctlSection.SetReadOnly(TRUE);
m_bAddMode = FALSE;
}
UpdateData(FALSE); //更新表单视图
}
BOOL CSectionForm::OnMove(UINT nIDMoveCommand)
{
if (m_bAddMode)
{
if (!UpdateData())
return FALSE;
TRY
{
m_pSet->Update();
}
CATCH(CDBException, e)
{
AfxMessageBox(e->m_strError);
return FALSE;
}
END_CATCH
m_pSet->Requery(); //重新查询，使新加的记录对用户可见
UpdateData(FALSE);
m_ctlSection.SetReadOnly(TRUE);
m_bAddMode = FALSE;
return TRUE;
}
else
{
return CRecordView::OnMove(nIDMoveCommand);
}
}
```

我们先来看看 Add **命令的处理函数** CSectionForm::OnRecordAdd **函数。在该函数中，最重要的代码是调用** CRecordset::AddNew **进入添加模式。其余代码的解释如下：**

如果已处于添加模式，则调用 CSectionForm::OnMove **滚动到别的记录上，这导致新记录被保存到数据源中。通过滚动到别的记录上来完成添加操作是一种常用的方法。其实，缺省的** CRecordView::OnMove **就可**

以完成这一功能(参见 10.5.6 和 10.6)，但 CSectionForm::OnMove 有另外的考虑(见下面的说明)。

保存当前记录的 CourseID，并将它作为新记录的缺省值。

调用 CEdit::SetReadOnly(FALSE)把 Section 编辑框改成可输入的，以便用户输入新的值。

CSectionForm::OnMove 负责处理滚动命令。与缺省的 CRecordView::OnMove 函数不同的是，该函数对于添加模式下的滚动进行了重新处理：

在调用 CRecordset::Update 把新记录保存到数据源后，调用 CRecordset::Requery 重新查询记录集。这样做的原因是 Enroll 使用的是快照型记录集，快照不反映用户添加的记录，所以需要调用 Requery 重新查询以把新加的记录包含进记录集中。在调用 Requery 后，会自动滚动到第一个记录上，所以在添加模式下滚动记录总是滚动动到第一个记录上。

在调用 CRecordset::Update 时，对可能发生的异常进行了处理。这里将直接输出异常信息。

在调用 Requery 后，要调用 CRecordView::UpdateData(FALSE)来更新表单，并调用 CEdit::SetReadOnly(TRUE)使 Sectioin 编辑框变成只读的。

在 Delete 命令的处理函数 CSectionForm::OnDelete 中调用了 CRecordset::Delete 来删除记录，并对可能发生的异常进行了处理。在调用 Delete 后，滚动记录到新的位置上以跳过被删除的记录。

Refresh 命令的处理函数 CSectionForm::OnRefresh 用来放弃修改或添加记录的操作。对该函数的解释为：

如果当前处于添加模式，则调用 CRecordset::Move(AFX_MOVE_REFRESH)取消添加模式并恢复域数据成员的原值(参见 10.5.6)，把 Section 编辑框设置成只读的。

调用 CRecordView::UpdateData(FALSE)恢复表单视图中的记录。

CSectionForm::OnSelendokCourselist 函数中多了一个用来判断当前是否处于添加模式的 if 语句。如果处于添加模式，那么就不能调用 Requery 重新查询，因为此时 Course 组合框的作用仅仅是让用户选择一个字段值，而不是指定过滤器。

编译并运行 Enroll，试试新增加的功能。

本教程由 Visual C++王朝(Where programmers come together)协助制作未经许可，请勿以任何形式复制 /

10.8 DAO

10.8.1 什么是 DAO

DAO(Database Access Object)使用 Microsoft Jet 数据库引擎来访问数据库。Microsoft Jet 为象 Access 和 Visual Basic 这样的产品提供了数据引擎。

与 ODBC 一样，DAO 提供了一组 API 供编程使用。MFC 也提供了一组 DAO 类，封装了底层的 API，从而大大简化了程序的开发。利用 MFC 的 DAO 类，用户可以编写独立于 DBMS 的应用程序。

DAO 是从 Visual C++4.0 版开始引入的。一般地讲，DAO 类提供了比 ODBC 类更广泛的支持。一方面，只要有 ODBC 驱动程序，使用 Microsoft Jet 的 DAO 就可以访问 ODBC 数据源。另一方面，由于 DAO 是基于 Microsoft Jet 引擎的，因而在访问 Access 数据库(即*.MDB 文件)时具有很好的性能。

10.8.2 DAO 和 ODBC 的相似之处

DAO 类与 ODBC 类相比具有很多相似之处，这主要有下面几点：

二者都支持对各种 ODBC 数据源的访问。虽然二者使用的数据引擎不同，但都可以满足用户编写独立于 DBMS 的应用程序的要求。

DAO 提供了与 ODBC 功能相似的 MFC 类。例如，DAO 的 CDaoDatabase 类对应 ODBC 的 CDatabase 类，CDaoRecordset 对应 CRecordset，CDaoRecordView 对应 CRecordView，CDaoException 对应 CDBException。这些对应的类功能相似，它们的大部分成员函数都是相同的。

AppWizard 和 ClassWizard 对使用 DAO 和 ODBC 对象的应用程序提供了类似的支持。

由于 DAO 和 ODBC 类的许多方面都比较相似，因此只要用户掌握了 ODBC，就很容易学会使用 DAO。实际上，用户可以很轻松地把数据库应用程序从 ODBC 移植到 DAO。

Visual C++随盘提供了一个名为 DaoEnrol 的例子，该例实际上是 Enroll 的一个 DAO 版本。读者可以打开 DaoEnrol 工程看一看，它的源代码与 Enroll 的极为相似。读者可以按照建立 Enroll 的步骤来建立 DaoEnrol，其中只有若干个地方有差别，这主要有以下几点：

选取的数据源不同。在用 AppWizard 创建 DaoEnrol 时，以及在用 ClassWizard 创建 CDaoRecordset 类的派生类时，在 Database Options 对话框中应该选择 DAO 而不是 ODBC。而且 DAO 的数据源是通过选择一个.MDB 文件来指定的，即点击“ ... ”按钮后在文件对话框中选择要访问的.MDB 文件。

记录集的缺省类型不同。ODBC 记录集的缺省类型是快照(Snapshot)，而 DAO 则是动态集(Dynaset)。

参数化的方式不同。DAO 记录集的 m_strFilter 和 m_strSort 中的参数不是“ “ ”号，而是一个有意义的参数名。例如，在下面的过滤器中有一个名为 CourseIDParam 的参数。

m_pSet->m_strFilter =“CourseID = CourseIDParam”; 在 DoFieldExchange 函数中，有下面两行：pFX->SetFieldType(CDaoFieldExchange::param);DFX_Text(pFX, _T(“CourseIDParam”), m_strCourseIDParam);DFX 函数的第二个参数也是 CourseIDParam。

处理异常的方式不同。例如，在删除记录时，对异常的处理如下所示：

```
try
{
m_pSet->Delete();
}
catch(CDaoException* e)
{
AfxMessageBox(e->
m_pErrorInfo->m_strDescription);
e->Delete();
}
```

除了上述差别外，AppWizard 和 ClassWizard 也隐藏了一些细微的不同之处，例如，DAO 记录集是使用是 DFX 数据交换机制(DAO record field exchange)而不是 RFX，在 DAO 记录集的 DoFieldExchange 中使用的是 DFX 函数而不是 RFX 函数。

10.8.3 DAO 的特色

DAO 可以通过 ODBC 驱动程序访问 ODBC 数据源。但 DAO 是基于 Microsoft Jet 引擎的，通过该引擎，DAO 可以直接访问 Access、FoxPro、dBASE、Paradox、Excel 和 Lotus WK 等数据库。CDaoDatabase 类可以直接与这些数据库进行连接，而不必在 ODBC 管理器中注册 DSN。例如，下面的代码用来打开一个 FoxPro 数据库：

```
CDaoDatabase daoDb;
daoDb.Open( “ ”,FALSE,FALSE,”FoxPro 2.5;DATABASE=c:\\zyf”);
```

CDaoDatabase::Open 函数用来连接某个数据库，该函数的声明为：

```
virtual void Open( LPCTSTR lpszName, BOOL bExclusive = FALSE,
BOOL bReadOnly = FALSE, LPCTSTR lpszConnect =
_T(“”) );throw( CDaoException, CMemoryException );
```

参数 bExclusive 如果为 TRUE，则函数以独占方式打开数据库，否则就用共享方式。如果 bReadOnly 为 TRUE，那么就以只读方式打开数据库。如果要打开一个 Access 数据库，则可以在 lpszName 参数中指定 MDB 文件名。如果要访问非 Access 数据库，则应使该参数为“ ”，并在 lpszConnect 中说明一个连接字符串。连接字符串的形式一般为“数据库类型;DATABASE= 路径(文件)”，例如 “dBASE III;DATABASE=c:\\MYDIR”

Open 函数也可以打开一个 ODBC 数据源，但这需要相应的 ODBC 驱动程序，并需要在 ODBC 管理器中注册 DSN。此时 lpszConnect 的形

式为“ODBC;DSN=MyDataSource”。显然，用 DAO 访问象 FoxPro 这样的数据库时，直接打开比把它当作 ODBC 数据源打开要省事。

支持 DDL 是 DAO 对数据库编程良好支持的一个重要体现。DDL(Data Definition Language)在 SQL 术语中叫做“数据定义语言”，它用来完成生成、修改和删除数据库结构的操作。ODBC 类只支持 DML(Data Manipulation Language，数据操作语言)，不支持 DDL，所以用 ODBC 类只能完成数据的操作，不能涉及数据库的结构。要执行 DDL 操作，只有通过 ODBC API。而 DAO 类同时提供了对 DML 和 DDL 的支持，这意味着程序可以使用 DAO 类方便的创建数据库及修改数据库的结构。

与 ODBC 相比，DAO 提供了一些新类来加强其功能，这些新类包括：

CDaoTableDef 类提供了对表的结构的定义。调用 CDaoTableDef::Open 可以获得表的结构定义。调用 CDaoTableDef::Create 可以创建一张新表，调用 CDaoTableDef:: CreateField 可为表添加字段，调用 CDaoTableDef::CreateIndex 可以为表添加索引。调用 CDaoTableDef::Append 可以把新创建的表保存到数据库中。

CDaoQueryDef 类代表一个查询定义(Query definition)，该定义可以被存储到数据库中。

CDaoWorkspace 提供了数据工作区(Workspace)。一个工作区可以包含几个数据库，工作区可以对所属的数据库进行全体或单独的事务处理，工作区也负责数据库的安全性。如果需要，程序可以打开多个工作区。

DAO 的另一个重要特色在于它对 Access 数据库提供了强大的支持。由于 DAO 是基于 Microsoft Jet 引擎的，所以 DAO 肯定要在 Access 数据库上多作一些文章。例如，调用 CDaoDatabase::Create 可以直接建立一个 MDB 文件，代码如下所示：m_db.Create(“C:\\MYDIR\\MYDB.MDB”);

利用 AppWizard 和 ClassWizard，用户可以方便地开发出性能优良的基于 DAO 的 Access 数据库应用程序。

10.8.4 ODBC 还是 DAO

由于 DAO 可以访问 ODBC 数据源，下面几条可以作为 DAO 替代 ODBC 的理由：

在某些情况下可以获得更好的性能，特别是在访问 Microsoft Jet(.MDB)数据库时。

与 ODBC 兼容

DAO 允许数据有效检查

DAO 允许用户说明表与表之间的关系

当然，DAO 的出现并不意味着 ODBC 已经过时了。如果用户的工作必须严格限于 ODBC 数据源，尤其是在开发 Client/Server 结构的应用程序时，用 ODBC 有较好的性能。

本教程由 Visual C++王朝(Where programmers come together)协助制作未经许可，请勿以任何形式复制/

10.9 自动注册 DSN 和创建表

在开始编写自己的数据库应用程序时，读者很快会遇到两个令人头痛的问题。一是在访问 ODBC 数据源前，必须在 ODBC 管理器中手工注册 DSN(数据源名)。这样的应用程序要求用户作额外的工作，显得很不专业。另一个问题是 AppWizard 和 ClassWizard 并不支持表的创建，程序员似乎必须先用 DBMS 创建好表，然后才能使用。如果一个数据库应用程序不能自己创建表，那么它的功能将大打折扣。

事实上，通过一些技巧，可以使应用程序能够对用户透明地注册 DSN 并任意创建表。本节的目的就是教会读者如何解决这两个问题。在下面几个小节中，分别提供了 ODBC 和 DAO 的解决方案。

10.9.1 自动注册 DSN

无论是用 ODBC 还是 DAO 类，在访问 ODBC 数据源以前，都必须先注册 DSN。通过调用函数 SQLConfigDataSource，可以实现自动注册 DSN。当然，用 DAO 可以直接访问一些常用的数据库，而不必通过 ODBC 来访问(参见 10.8.3)。

清单 10.14 的代码演示了注册 DSN 的过程。该段代码先用 SQLConfigDataSource 注册一个名为 MYDB 的 FoxPro 2.5 数据源，然后调用 CDatabase::Open 函数与该数据源连接。注意在使用这段代码时，要包含 afxdb.h 头文件，读者可以把该文件放到 stdafx.h 中。

清单 10.14 自动注册 DSN

```
#include “afxdb.h”
. . .
CDatabase db;
if(!SQLConfigDataSource(NULL,ODBC_ADD_DSN, “Microsoft FoxPro Driver (*.dbf)”,
“DSN=MYDB\0”
“DefaultDir=c:\\mydir\0”
“FIL=FoxPro 2.5\0”
“DriverId=280\0”))
{
AfxMessageBox(“Can't add DSN!”);
return ;
}
TRY
{
db.Open(“MYDB”);
}
CATCH(CDBException, e)
{
AfxMessageBox(e->m_strError);
return;
```

```
}
END_CATCH
```

在注册 DSN 时，SQLConfigDataSource 函数的第二个参数应该是 ODBC_ADD_DSN，第三个参数指定了 ODBC 驱动程序，它的写法可以参照 ODBC 管理器的驱动程序页。第四个参数说明了数据源的各种属性，它是由一系列子串构成，每个子串的末尾必须有一个“\0”。最重要的属性是“DSN＝数据源名”,其它属性包括缺省目录以及驱动程序版本信息。在上例中，使用 FoxPro 2.5 的版本，所以 DriverId 应该是 280，对应地，FoxPro 2.6 的 DriverId 是 536，FoxPro 2.0 的是 24。

如果读者对 SQLConfigDataSource 函数的第四个参数的设置方法不清楚，那么可以打开 Windows 的注册表看一看已注册过的 DSN 的各项属性。运行 RegEdit 可以打开注册表，然后依次打开 HKEY_CURRENT_USER->Software->ODBC->ODBC.INI，就可以看到已注册的 DSN，打开各 DSN，则可以看到该 DSN 的各项属性，读者可以仿照 DSN 属性来设置第四个参数。

DSN 的名字必须唯一，因此如果要注册的 DSN 已被注册过，那么 SQLConfigDataSource 就修改原来 DSN 的属性。

10.9.2 用 ODBC 创建表

由于 ODBC 类不支持 DDL，所以只有通过 ODBC API 来创建表。程序需要调用 Cdatabase :: ExecuteSQL 来直接执行 SQL 语句。

清单 10.15 给出了创建表的一个例子，该程序先自动注册了一个名为 MYDB 的 FoxPro 2.5 数据源，然后创建了一个名为 OFFICES 的表(OFFICES.DBF 文件)，在这个表中有 OfficeID 和 OfficeName 两个 TEXT 型字段，长度分别为 4 和 10 个字节。注意，如果要使用这段代码，则需要包含 afxdb.h 和 odbcinst.h。

清单 10.15 ODBC 创建表的例子

```
#include “afxdb.h”
#include “odbcinst.h”
. . .
CDatabase db;
if(!SQLConfigDataSource(NULL,ODBC_ADD_DSN, “Microsoft FoxPro Driver (*.dbf)”,
“DSN=MYDB\0”
“DefaultDir=c:\\mydir\0”
“FIL=FoxPro 2.5\0”
“DriverId=280\0”))
{
AfxMessageBox(“Can't add DSN!”);
return ;
}
TRY
{
```

```
db.Open("MYDB");
db.ExecuteSQL("CREATE TABLE OFFICES (OfficeID TEXT(4)"
",OfficeName TEXT(10))"); }
CATCH(CDBException, e)
{
AfxMessageBox(e->m_strError);
return;

}
END_CATCH
```

ExecuteSQL 执行了一个实实在在的 SQL 语句 CREATE 来创建表，看来用户应该找本 SQL 方面的书研究研究。要注意一个数据库中的表名必须是唯一的，如果要创建的表已经存在，则 ExecuteSQL 会产生一个异常。

10.9.3 用 DAO 创建表

由于 DAO 类直接支持 DDL，所以用 DAO 类创建表比 ODBC 容易。DAO 的 CDaoTableDef 类提供了对表的结构的定义，该类提供了创建表的成员函数。

清单 10.16 演示了一段用 DAO 类创建表的例子。注意，若要使用这段代码，则应该包含 afxdao.h 头文件。在该例中，先与一个 FoxPro 2.5 数据库连接(实际上是一个目录)，然后再构建一个 CDaoTableDef 对象，接着调用 CDaoTableDef :: Create 函数创建一个名为 STUDENTS 的表(STUDENTS.DBF)，调用 CDaoTableDef :: CreateField 为该表创建了两字段，字段名分别是 ID 和 NAME，类型分别是 Integer 和 dbText，字段的长度分别为 2 和 10 个字节。最后调用 CDaoTableDef :: Append 把新创建的表保存到数据库中。

清单 10.16 DAO 创建表的例子

```
#include “ afxdao.h ”
. . .
CDaoDatabase daoDb;
try
{
daoDb.Open(“”,FALSE,FALSE,”FoxPro 2.5;DATABASE=d:\\zwin”);
CDaoTableDef table(&daoDb);
table.Create(“STUDENTS”);
table.CreateField(“ID”,dbInteger,2);
table.CreateField(“NAME”,dbText,10);
table.Append();
}
catch(CDaoException* e)
{
AfxMessageBox(e->
```

```
m_pErrorInfo->m_strDescription);
e->Delete();
return FALSE;
}
```

关于 CDaoDatabase :: Open **的说明请参见** 10.8.3**。注意一个数据库中的表名必须是唯一的，如果要创建的表已经存在，则** CDaoTableDef :: Create **会产生一个异常。**

小结

本章介绍了 MFC 的 ODBC 和 DAO 类，并向读者演示了编写数据库应用程序的方法。本章的要点包括：

关系数据库由多个相关的表组成，DBMS(数据库管理系统)是一套程序，用来定义、管理和处理数据库与应用程序之间的联系，SQL 是一种标准的数据库语言，目前大多数 DBMS 都支持它。

用 ODBC 和 DAO，用户可以编写独立于 DBMS 的数据库应用程序。

在访问 ODBC 数据源之前，应该安装相应的 ODBC 驱动程序，并在 ODBC 管理器中注册 DSN。

MFC 提供了 ODBC 类，其中 CDatabase 针对某个数据库，它负责连接数据源，CRecordset 针对数据源中的记录集，它负责对记录的操作，CRecordView 负责界面，而 CFieldExchange 负责 CRecordset 与数据源的数据交换。

记录集主要包括动态集和快照。快照提供了对数据的静态视，动态集提供了数据的动态视。

通过学习 Enroll 例子，读者可以掌握数据库编程的方法。

一般来说，DAO 提供了比 ODBC 类更广泛的支持。DAO 提供了几个新类，包括 CDaoTableDef、CDaoQueryDef、CDaoWorkspace 等。DAO 支持 DDL(数据定义语言)，DAO 对 Access 数据库提供了强大的支持。

通过手工编写少量代码，就可以使应用程序能够对用户透明地注册 DSN 并任意创建表。

第十一讲　多媒体编程

随着多媒体技术的迅猛发展和 PC 性能的大幅度提高，在 PC 机上运行的应用程序越来越多地采用了多媒体技术．如果你编写的应用程序能够发出美妙的声音，播放有趣的动画，无疑将会给人留下深刻的映象．

Windows 95 提供了对多媒体编程的良好支持，本章将帮助读者迅速掌握一些实用的多媒体编程技术，主要的内容包括：

调色板

位图

依赖于设备的位图(DDB)

与设备无关的位图(DIB)

动画控件

媒体控制接口(MCI)．

小结

11.1 调色板

11.1.1 调色板的原理

PC 机上显示的图象是由一个个像素组成的，每个像素都有自己的颜色属性。在 PC 的显示系统中，像素的颜色是基于 RGB 模型的，每一个像素的颜色由红(B)、绿(G)、蓝(B)三原色组合而成。每种原色用 8 位表示，这样一个的颜色就是 24 位的。以此推算，PC 的 SVGA 适配器可以同时显示 224 约一千六百多万种颜色。24 位的颜色通常被称作真彩色，用真彩色显示的图象可达到十分逼真的效果。

但是，真彩色的显示需要大量的视频内存，一幅 640×480 的真彩色图象需要约 1MB 的视频内存。由于数据量大增，显示真彩色会使系统的整体性能迅速下降。为了解决这个问题，计算机使用调色板来限制颜色的数目。调色板实际上是一个有 256 个表项的 RGB 颜色表，颜色表的每项是一个 24 位的 RGB 颜色值。使用调色板时，在视频内存中存储的不是的 24 位颜色值，而是调色板的 4 位或 8 位的索引。这样一来，显示器可同时显示的颜色被限制在 256 色以内，对系统资源的耗费大大降低了。

显示器可以被设置成 16、256、64K、真彩色等显示模式，前两种模式需要调色板。在 16 或 256 色模式下，程序必须将想要显示的颜色正确地设置到调色板中，这样才能显示出预期的颜色。图 11.1 显示了调色板的工作原理。使用调色板的一个好处是不必改变视频内存中的值，只需改变调色板的颜色项就可快速地改变一幅图象的颜色或灰度。

在 DOS 中，调色板的使用不会有什么问题。由于 DOS 是一个单任务操作系统，一次只能运行一个程序，因此程序可以独占调色板。在 Windows 环境下，情况就不那么简单了。Windows 是一个多任务操作系统，可以同时运行多个程序。如果有几个程序都要设置调色板，就有可能产生冲突。为了避免这种冲突，Windows 使用逻辑调色板来作为使用颜色的应用程序和系统调色板(物理调色板)之间的缓冲。

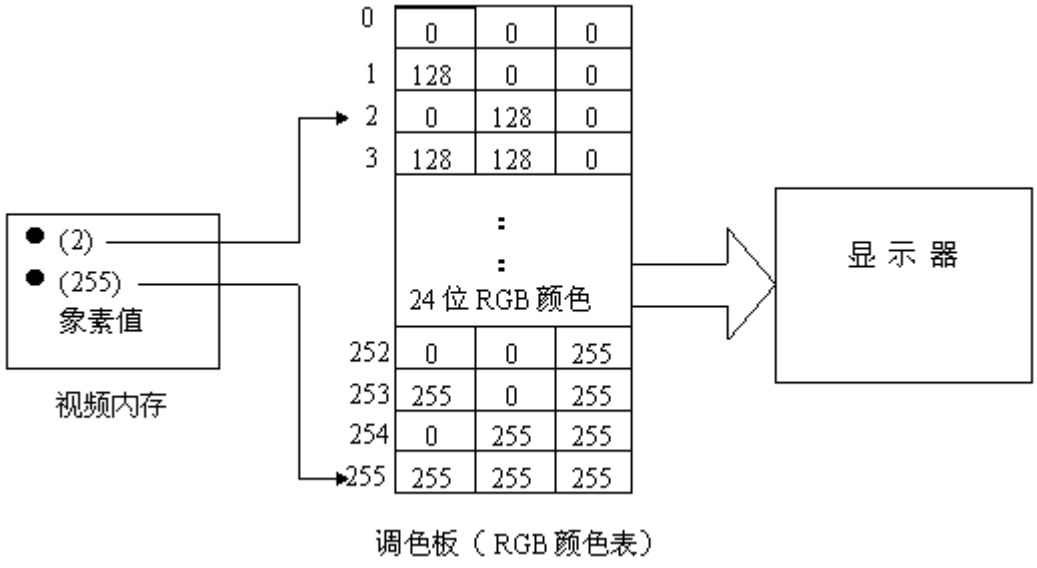


图 11.1 调色板工作原理

在 Windows 中，应用程序是通过一个或多个逻辑调色板来使用系统

调色板(物理调色板)。在 256 色系统调色板中，Windows 保留了 20 种颜色作为静态颜色，这些颜色用作显示 Windows 界面，应用程序一般不能改变。缺省的系统调色板只包含这 20 种静态颜色，调色板的其它项为空。应用程序要想使用新的颜色，必须将包含有所需颜色的逻辑调色板实现到系统调色板中。在实现过程中，Windows 首先将逻辑调色板中的项与系统调色板中的项作完全匹配，对于逻辑调色板中不能完全匹配的项，Windows 将其加入到系统调色板的空白项中，系统调色板总共有 236 个空白项可供使用，若系统调色板已满，则 Windows 将逻辑调色板的剩余项匹配到系统调色板中尽可能接近的颜色上。

每个设备上下文都拥有一个逻辑调色板，缺省的逻辑调色板只有 20 种保留颜色，如果要使用新的颜色，则应该创建一个新的逻辑调色板并将其选入到设备上下文中。但光这样还不能使用新颜色，程序只有把设备上下文中的逻辑调色板实现到系统调色板中，新的颜色才能实现。在逻辑调色板被实现到系统调色板时，Windows 会建立一个调色板映射表。当设备上下文用逻辑调色板中的颜色绘图时，GDI 绘图函数会查询调色板映射表以把像素值从逻辑调色板的索引转换成系统调色板的索引，这样当像素被输出到视频内存中时就具有了正确的颜色值。图 11.2 说明了这种映射关系，从图中读者可以体会到逻辑调色板的缓冲作用。在该图中，GDI 绘图函数使用逻辑调色板的索引 1 中的颜色来绘图，通过查询调色板映射表，得知系统调色板中的第 23 号索引与其完全匹配，这样实际输出到视频内存中的像素值是 23。注意图中还演示了颜色的不完全匹配，即逻辑调色板中的索引 15 和系统调色板中的索引 46。

每个要使用额外颜色的窗口都会实现自己的逻辑调色板，逻辑调色板中的每种颜色在系统调色板中都有相同或相近的匹配。调色板的实现优先权越高，匹配的精度也就越高。Windows 规定，活动窗口的逻辑调色板(如果有的话)具有最高的实现优先权。这是因为活动窗口是当前与用户交互的窗口，应该保证其有最佳的颜色显示。非活动窗口的优先权是按 Z 顺序自上到下确定的(Z 顺序就是重叠窗口的重叠顺序)。活动窗口有权将其逻辑调色板作为前景调色板实现，非活动窗口则只能实现背景调色板。

提示：术语活动窗口(Active window)或前台窗口(Foreground window)是指当前与用户交互的窗口，活动窗口的顶端的标题条呈高亮显示，而非活动窗口的标题条则是灰色的。活动窗口肯定是一个顶层窗口(Top-level window)，顶层窗口是指没有父窗口或父窗口是桌面窗口的窗口，这种窗口一般都有标题和边框，主要包括框架窗口和对话框。术语重叠窗口是指作为应用程序主窗口的窗口，我们可以把对话框看成是一种特殊的重叠式窗口。/

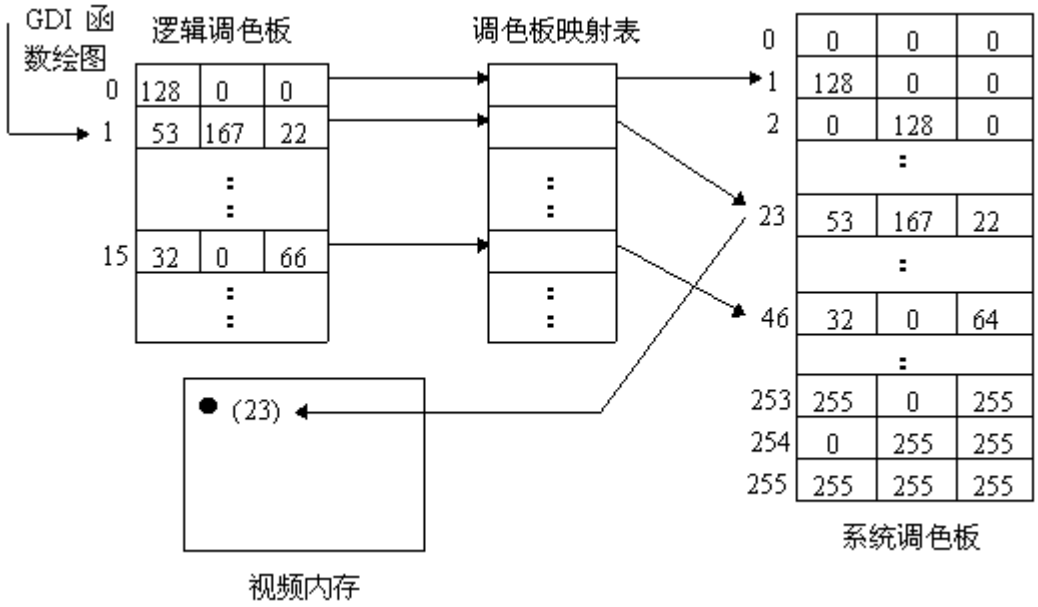


图 11.2 **调色板的映射关系**

11.1.2 **调色板的创建和实现**

MFC **的** CPalette **类对逻辑调色板进行了封装。该类的成员函数** CreatePalette **负责创建逻辑调色板，该函数的声明为：**

BOOL CreatePalette(LPLOGPALETTE lpLogPalette); //**成功则返回** TRUE**。**

参数 lpLogPalette **是一个指向** LPLOGPALETTE **结构的指针，** LPLOGPALETTE **结构描述了逻辑调色板的内容，该结构的定义为：**

typedef struct tagLOGPALETTE {

WORD palVersion; //Windows **版本号，一般是** 0x300

WORD palNumEntries; //**调色板中颜色表项的数目**

PALETTEENTRY palPalEntry[1]; //**每个表项的颜色和使用方法**

} LOGPALETTE;

结构中最重要的成员是 PALETTEENTRY **数组，数组项的数目由** palNumEntries **成员指定。**PALETTEENTRY **结构对调色板的某一个颜色表项进行了描述，该结构的定义为：**

typedef struct tagPALETTEENTRY {

BYTE peRed; //**红色的强度**(0~255**，下同**)

BYTE peGreen; //**绿色的强度**

BYTE peBlue; //**蓝色的强度**

BYTE peFlags;

} PALETTEENTRY;

成员 peFlags **说明了颜色表项的使用方法，在一般应用时为** NULL**，若读者对** peFlags **的详细说明感兴趣，可以查看** Visual C++**的联机帮助。**

可以看出，创建调色板的关键是在 PALETTEENTRY **数组中指定要使用的颜色。这些颜色可以是程序自己指定的特殊颜色，也可以从** DIB **位图中载入。逻辑调色板的大小可根据用户使用的颜色数来定，一般不能超过** 256 **个颜色表项。**

CreatePalette 只是创建了逻辑调色板，此时调色板只是一张孤立的颜色表，还不能对系统产生影响。程序必需调用 CDC::SelectPalette 把逻辑调色板选入到要使用它的设备上下文中，然后调用 CDC::RealizePalette 把逻辑调色板实现到系统调色板中。函数的声明为：

CPalette* SelectPalette(CPalette* pPalette, BOOL bForceBackground);

该函数把指定的调色板选择到设备上下文中。参数 pPalette 指向一个 CPalette 对象。参数 bForceBackground 如果是 TRUE，那么被选择的调色板总是作为背景调色板使用，如果 bForceBackground 是 FALSE 并且设备上下文是附属于某个窗口的，那么当窗口是活动窗口或活动窗口的子窗口时，被选择的调色板将作为前景调色板实现，否则作为背景调色板实现。如果使用调色板的是一个内存设备上下文，则该参数被忽略。函数返回设备上下文原来使用的调色板，若出错则返回 NULL。

UINT RealizePalette( );该函数把设备上下文中的逻辑调色板实现到系统调色板中。函数的返回值表明调色板映射表中有多少项被改变了。

如果某一个窗口要显示特殊的颜色，那么一般应该在处理 WM_PAINT 消息时实现自己的逻辑调色板。也就是说，在 OnPaint 或 OnDraw 函数中重绘以前，要调用 SelectPalette 和 RealizePalette。如果窗口显示的颜色比较重要，则在调用 SelectPalette 时应该指定 bForceBackground 参数为 FALSE。

前景调色板具有使用颜色的最高优先级，它有无条件占用系统调色板(20 种保留颜色除外)的权力，也就是说，如果需要，前景调色板将覆盖系统调色板的 236 个表项，而不管这些表项是否正被别的窗口使用。背景调色板则无权破坏系统调色板中的已使用项。

请读者注意，前景调色板应该是唯一。如果一个活动窗口同时要实现几个逻辑调色板，那么只能有一个调色板作为前景调色板实现，也即在调用 CDC::SelectPalette 时只能有一个 bForceBackground 被指定为 FALSE，其它的 bForceBackground 必需为 TRUE。通常是把具有输入焦点的窗口的调色板作为前景调色板实现，其它窗口只能使用背景调色板。如果活动窗口的子窗口全都使用前景调色板，则会导致程序的死循环。

提示:请读者注意区分活动窗口和有输入焦点的窗口。有输入焦点的窗口要么是活动窗口本身，要么是活动窗口的子窗口。也就是说，活动窗口不一定具有输入焦点，当活动窗口的子窗口获得输入焦点时，活动窗口就会失去输入焦点。/

11.1.3 使用颜色的三种方法

在调用 GDI 函数绘图时，可以用不同的方法来选择颜色。Windows 用 COLORREF 数据类型来表示颜色，COLORREF 型值的长度是 4 字节，其中最高位字节可以取三种不同的值，分别对应三种使用颜色的方法。表 11.1 列出了这些不同的取值及其含义。

表 11.1COLORREF 型值的最高位字节的含义

取值	含义
0x00	指定 RGB 引用。此时三个低位字节含有红、绿、蓝色的强度，Windows 将抖动 20 种保留的颜色来匹配指定的颜色，

	而不管程序是否实现了自己的调色板。
0x01	指定调色板索引引用。此时最低位字节含有逻辑调色板的索引，Windows 根据该索引在逻辑调色板中找到所需的颜色。
0x02	指定调色板 RGB 引用。此时三个低位字节含有红、绿、蓝色的强度，Windows 会在逻辑调色板中找到最匹配的颜色。

为了方便用户的使用，Windows 提供了三个宏来构建三种不同的 COLORREF 数据，它们是：

COLORREF RGB(BYTE bRed,BYTE bGreen,BYTE bBlue); //RGB 引用

COLORREF PALETTEINDEX(WORD wPaletteIndex); //调色板索引引用

COLORREF PALETTERGB(BYTE bRed,BYTE bGreen, //调色板 RGB 引用 BYTE bBlue);

例如，我们可以用上述三种方法来指定刷子的颜色。下面的代码用系统调色板中的红色建立一个刷子：

```
CBrush brush;
brush.CreateSolidBrush(RGB(255,0,0));
pDC->SelectObject(&brush);
```

下面的代码用逻辑调色板的索引 2 中的颜色来创建一个刷子：

```
pDC->SelectPalette(&m_Palette,FALSE);
pDC->RealizePalette( );
CBrush brush;
brush.CreateSolidBrush(PALETTEINDEX(2));
pDC->SelectObject(&brush);
```

下面的代码用逻辑调色板中最匹配的深灰色来创建一个刷子：

```
pDC->SelectPalette(&m_Palette,FALSE);
pDC->RealizePalette( );
CBrush brush;
brush.CreateSolidBrush(PALETTERGB(20,20,20));
pDC->SelectObject(&brush);
```

11.1.4 与系统调色板有关的消息

为了协调各个窗口对系统调色板的使用，Windows 在必要的时侯会向顶层窗口和重叠窗口发送消息 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED。

当某一顶层或重叠窗口(如主框架窗口)被激活时，会收到 WM_QUERYNEWPALETTE 消息，在窗口创建之初也会收到该消息，该消息先于 WM_PAINT 消息到达窗口。如果活动窗口要使用特殊的颜色，则在收到该消息时应该实现自己的逻辑调色板并重绘窗口。如果窗口实现了逻辑调色板,那么 WM_QUERYNEWPALETTE 消息的处理函数应返回 TRUE。通常窗口在收到该消息后应该为有输入焦点的窗口(如视图)实现前景调色板，但如果程序觉得它显示的颜色并不重要，那么在收到该消息后可以把逻辑调色板作为背景调色板实现(指定 CDC::SelectPalette

函数的 bForceBackground 参数为 TRUE)，这样程序就失去了使用系统调色板的最高优先权。

当活动窗口实现其前景调色板并改变了系统调色板时，Windows 会向包括活动窗口在内的所有的顶层窗口和重叠窗口发送 WM_PALETTECHANGED 消息，在该消息的 wParam 参数中包含了改变系统调色板的窗口的句柄。其它窗口如果使用了自己的逻辑调色板，那么应该重新实现其逻辑调色板，并重绘窗口。这是因为系统调色板已经被改变了，必需重新建立调色板映射表并重绘，否则可能会显示错误的颜色。当然，非活动窗口只能使用背景调色板，所以显示的颜色肯定没有在前台的时侯好。要注意只有在活动窗口实现了前景调色板且改变了系统调色板时，才会产生 WM_PALETTECHANGED 消息。也就是说，如果窗口在调用 CDC::SelectPalette 时指定 bForceBackground 参数为 TRUE，那么是不会产生 WM_PALETTECHANGED 消息。

总之，WM_QUERYNEWPALETTE 消息为活动窗口提供了实现前景调色板的机会，而 WM_PALETTECHANGED 消息为窗口提供了适应系统调色板变化的机会。

需要指出的是，子窗口是收不到与调色板有关的消息的。因此，如果子窗口(如视图)要使用自己的逻辑调色板，那么顶层窗口或重叠窗口应该及时通知子窗口与调色板有关的消息。

11.1.5 具体实例

现在让我们来看一个使用调色板的演示程序。该程序名为 TestPal，如图 11.3 所示，该程序显示了两组红色方块，每组方块都是 16×16 共 256 个。左边的这组方块是用逻辑调色板画的，红色的强度从 0 到 255 递增，作为对比，在右边用 RGB 引用画出了 256 个递增的红色方块。读者可以对比这两组方块的颜色质量，以体会调色板索引引用和 RGB 引用的区别。该程序也着重向读者演示了处理调色板消息的方法。

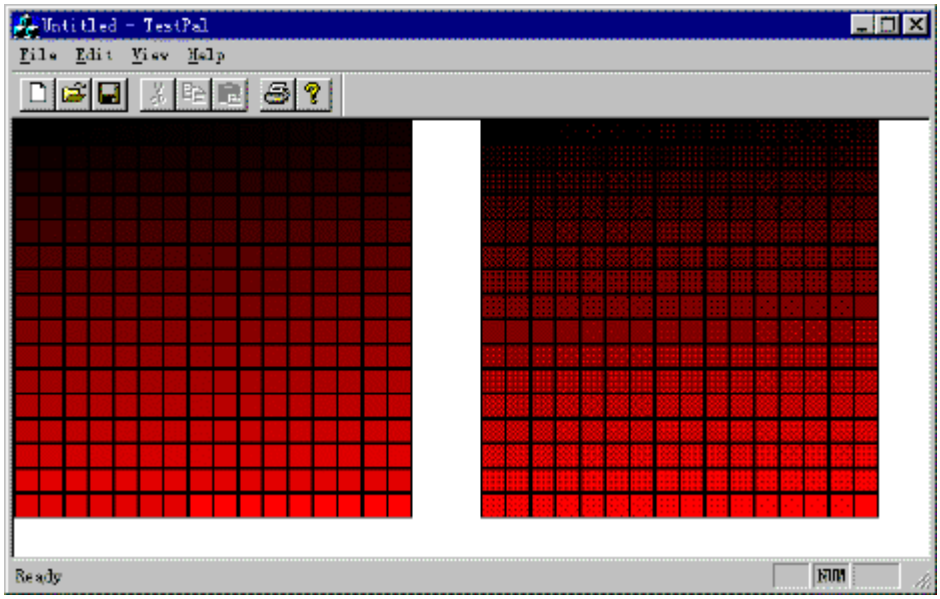


图 11.3 TestPal 程序

首先，请读者用 AppWizard 建立一个名为 TestPal 的 MFC 单文挡应

用程序。然后，用 ClassWizard 为 CMainFrame 类加入 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息的处理函数，使用缺省的函数名。接着，在 TestPal.h 文件中类 CTestPalApp 的定义前加入下面一行：

```
#define WM_DOREALIZE WM_USER+200
```

当收到调色板消息时，主框架窗口会发送用户定义的 WM_DOREALIZE 消息通知视图。

最后，请读者按清单 11.1 和 11.2 修改程序。

清单 11.1 CMainFrame 类的部分代码

```
BOOL CMainFrame::OnQueryNewPalette()
{
// TODO: Add your message handler code here and/or call default
GetActiveView()->SendMessage(WM_DOREALIZE);
return TRUE; //返回 TRUE 表明实现了逻辑调色板
}
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
CFrameWnd::OnPaletteChanged(pFocusWnd);
// TODO: Add your message handler code here
if(GetActiveView()!=pFocusWnd)
GetActiveView()->SendMessage(WM_DOREALIZE);
}
```

清单 11.2 CTestPalView 类的部分代码

```
// TestPalView.h : interface of the CTestPalView class
class CTestPalView : public CView
{
. . .
protected:
CPalette m_Palette;
. . .
afx_msg LRESULT OnDoRealize(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};
// TestPalView.cpp : implementation of the CTestPalView class
BEGIN_MESSAGE_MAP(CTestPalView, CView)
. . .
ON_MESSAGE(WM_DOREALIZE, OnDoRealize)
END_MESSAGE_MAP()
CTestPalView::CTestPalView()
{
// TODO: add construction code here
```

```
LPLOGPALETTE pLogPal;
pLogPal=(LPLOGPALETTE)malloc(sizeof(LOGPALETTE)+
sizeof(PALETTEENTRY)*256);
pLogPal->palVersion=0x300;
pLogPal->palNumEntries=256;
for(int i=0;i<256;i++)
{
pLogPal->palPalEntry[i].peRed=i; //初始化为红色
pLogPal->palPalEntry[i].peGreen=0;
pLogPal->palPalEntry[i].peBlue=0;
pLogPal->palPalEntry[i].peFlags=0;
}
if(!m_Palette.CreatePalette(pLogPal))
AfxMessageBox("Can't create palette!");
}
void CTestPalView::OnDraw(CDC* pDC)
{
CTestPalDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
CBrush brush,*pOldBrush;
int x,y,i;
pDC->SelectPalette(&m_Palette,FALSE);
pDC->RealizePalette();
pDC->SelectStockObject(BLACK_PEN);
for(i=0;i<256;i++)
{
x=(i%16)*16;
y=(i/16)*16;
brush.CreateSolidBrush(PALETTEINDEX(i)); //调色板索引引用
pOldBrush=pDC->SelectObject(&brush);
pDC->Rectangle(x,y,x+16,y+16);
pDC->SelectObject(pOldBrush);
brush.DeleteObject();
}
for(i=0;i<256;i++)
{
x=(i%16)*16+300;
y=(i/16)*16;
brush.CreateSolidBrush(RGB(i,0,0)); //RGB 引用
pOldBrush=pDC->SelectObject(&brush);
pDC->Rectangle(x,y,x+16,y+16);
```

```
pDC->SelectObject(pOldBrush);
brush.DeleteObject();
}
}
LRESULT CTestPalView::OnDoRealize(WPARAM wParam, LPARAM)
{
CClientDC dc(this);
dc.SelectPalette(&m_Palette,FALSE);
if(dc.RealizePalette()) //若调色板映射被改变则刷新视图
GetDocument()->UpdateAllViews(NULL);
return 0L;
}
```

在 CTestPalView 的构造函数中创建了一个含有 256 个递增红色的逻辑调色板。

当变为活动窗口以及窗口创建时，TestPal 程序的主框架窗口都会收到 WM_QUERYNEWPALETTE 消息，该消息的处理函数 OnQueryNewPalette 负责发送 WM_DOREALIZE 消息通知视图，并返回 TRUE 以表明活动窗口实现了逻辑调色板。WM_DOREALIZE 消息的处理函数 CTestPalView::OnDoRealize 为视图实现一个前景调色板，该函数中有一个判断语句可提高程序运行的效率：如果 CDC::RealizePalette 返回值大于零，则说明调色板映射表发生了变化，此时必须刷新视图，否则制图中的颜色将失真。如果 RealizePalette 返回零则说明调色板映射没有变化，这时就没有必要刷新视图。

无论是 TestPal 还是别的应用程序在实现前景调色板并改变了系统调色板时，TestPal 程序的主框架窗口都会收到 WM_PALETTECHANGED 消息。请注意该消息的处理函数 CMainFrame::OnPaletteChanged 有一个 pFocusWnd 参数，该参数表明是哪一个窗口改变了系统调色板。函数用 pFocusWnd 来判断，如果是别的应用程序实现了前景调色板，则通知视图调用 OnDoRealize 实现其逻辑调色板，注意虽然 CDC::SelectPalette 的 bForceBackground 参数是 FALSE，但这时视图的逻辑调色板是作为背景调色板实现的。如果是 TestPal 自己的视图实现了前景调色板，则没有必要调用 OnDoRealize。

请读者将 Windows 当前的显示模式设置为 256 色，然后编译并运行 TestPal，对比一下 RGB 引用与调色板索引引用的效果，读者不难发现左边用调色板索引引用输出的颜色比右边好的多。通过该程序我们可以看出，即使在系统调色板中已实现了丰富的红色的情况下，RGB 引用得到的红色仍然是 20 种保留颜色的抖动色。

读者可以打开 Windows 的画笔程序，并在该程序中打开一幅 256 色的位图(如 Windows 目录下的 Forest.bmp)。在画笔和 TestPal 程序之间来回切换，读者可以看到，由于两个应用程序都正确的处理了调色板消息，在前台的应用程序总是具有最好的颜色显示，而后台程序的颜色虽然有

些失真，但还比较令人满意。

需要指出的是，TestPal 程序只使用了一个逻辑调色板，所以它处理调色板消息的方法比较简单。如果程序要用到多个逻辑调色板，那么就需要采取一些新措施来保证只有一个逻辑调色板作为前景调色板使用。在 11.4 节读者可以看到使用多个逻辑调色板时的处理方法。

11.2 位图

Windows 用位图(Bitmap)来显示和保存图像，从单色的到 24 位真彩色图像都可以存储到位图中。

位图实际上是一个像素值阵列，像素阵列存储在一个字节数组中，每一个像素的位数可以是 1、4、8 或 24。单色位图的字节数组中的每一位代表一个像素，16 色位图的字节数组中每一个字节存储两个像素，256 色的位图每一个字节存储一个像素，而真彩色位图中每个像素用三个字节来表示。在 256 色以下的位图中存储的像素值实际上是调色板索引，在真彩色位图中存储的则是像素的 RGB 颜色值。

位图分为依赖于设备的位图(DDB)和与设备无关的位图(DIB)，二者有不同的用途。

11.3 依赖于设备的位图(DDB)

DDB(Device-dependent bitmap)依赖于具体设备，这主要体现在以下两个方面：

DDB 的颜色模式必需与输出设备相一致。例如，如果当前的显示设备是 256 色模式，那么 DDB 必然也是 256 色的，即一个像素用一个字节表示。

在 256 色以下的位图中存储的像素值是系统调色板的索引，其颜色依赖于系统调色板。

由于 DDB 高度依赖输出设备，所以 DDB 只能存在于内存中，它要么在视频内存中，要么在系统内存中。

11.3.1 DDB 的创建

MFC 的 CBitmap 类封装了 DDB。该类提供了几个函数用来创建 DDB：

BOOL LoadBitmap(LPCTSTR lpszResourceName);BOOL LoadBitmap(UINT nIDResource);该函数从资源中载入一幅位图，若载入成功则返回 TRUE。资源位图实际上是一个 DIB，该函数在载入时把它转换成了 DDB。

BOOL CreateBitmap(int nWidth, int nHeight, UINT nPlanes, UINT nBitcount, const void* lpBits);该函数用来创建一幅空白的 DDB。参数 nWidth 和 nHeight 以像素为单位说明了位图的宽度和高度。nPlanes 是 DDB 的色平面数，nBitcount 是每个色平面的颜色位数。一般来说，nPlanes 为 1，而 nBitcount 代表 DDB 中每个像素值所占的位数，但在创建 16 色 DDB 时，nPlanes 为 4,而 nBitcount 为 1。参数 lpBits 指向存储像素阵列的数组，该数组应该逐行存储位图的每个像素值。注意，数组中每行像素的数目必需是偶数个字节，如果是奇数，则应该用 0 补足。若创建成功函数返回 TRUE。

BOOL CreateCompatibleBitmap(CDC* pDC, int nWidth, int nHeight);该函数创建一个与指定设备上下文兼容的 DDB。参数 pDC 指向一个设备上下文，nWidth 和 nHeight 是 DDB 的尺寸。若创建成功函数返回 TRUE。

可以调用 CBitmap 的成员函数 GetBitmap 来查询 DDB 的各种属性(如尺寸)：

int GetBitmap(BITMAP* pBitMap);该函数用来获得与 DDB 有关的信息，参数 pBitMap 指向一个 BITMAP 结构。BITMAP 结构的定义为：

```
typedef struct tagBITMAP {
LONG bmType; //必需为 0
LONG bmWidth; //位图的宽度(以像素为单位)
LONG bmHeight; //位图的高度(以像素为单位)
LONG bmWidthBytes; //每一扫描行所需的字节数，应是偶数
WORD bmPlanes; //色平面数
WORD bmBitsPixel; //色平面的颜色位数
LPVOID bmBits; //指向存储像素阵列的数组
} BITMAP;
```

11.3.2 DDB 的用途

DDB 的主要用途是保存位图。要保存的位图可以来自资源位图，也可以是一个绘图的结果。

前面说过，在 256 色以下的显示模式中，DDB 中的像素值是系统调色板的索引。一般在系统调色板中除了保留的 20 种静态颜色外，其它表项都有可能被应用程序改变。如果 DDB 中有一些像素值是指向 20 种静态颜色以外的颜色，那么该位图的颜色将是不稳定的。因此，DDB 不能用来长期存储色彩丰富的位图。如果位图使用的大部分颜色都是 20 种保留色，则该位图可以用 CBitmap 对象保存在内存中。例如，用 CDC::LoadBitmap 载入的资源位图一般都是颜色较简单的位图，对于那些颜色比较丰富的位图，只有使用下面将要介绍的 DIB 才能长期保存。

在窗口中显示 DDB 的方法有些特别，其过程分以下几步：

构建一个 CDC 对象，然后调用 CDC::CreateCompatibleDC 创建一个兼容的内存设备上下文。

调用 CDC::SelectObject 将 DDB 选入内存设备上下文中。

调用 CDC::BitBlt 或 CDC::StretchBlt 将 DDB 从内存设备上下文中输出到窗口的设备上下文中。

调用 CDC::SelectObject 把原来的 DDB 选入到内存设备上下文中并使新 DDB 脱离出来。

下面这段代码在视图中显示了一个 DDB：

```
void CMyView::OnDraw( CDC* pDC)
{
. . .
CDC MemDC;
CBitmap *oldBmp;
BITMAP bmpInfo;
int bmWidth,bmHeight;
MemDC.CreateCompatibleDC(pDC);
oldBmp=MemDC.SelectObject(&m_Bitmap);  //m_Bitmap 是一个 CBitmap 对象
m_Bitmap.GetBitmap(&bmpInfo); //获取位图的尺寸
bmWidth=bmpInfo.bmWidth;
bmHeight=bmpInfo.bmHeight;
pDC->BitBlt(0,0,bmWidth,bmHeight,&MemDC,0,0,srcCOPY);
MemDC.SelectObject(oldBmp); //使位图 m_Bitmap 脱离设备上下文
. . .
}
```

函数 CDC::BitBlt 的声明为：

```
BOOL BitBlt( int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, DWORD dwRop );
```

该函数把源设备上下文中的位图复制到本身的设备上下文中，两个设备上下文可以是内存设备上下文，也可以是同一个设备上下文。参数 x

和 y 是目的矩形的逻辑坐标，参数 nWidth 和 nHeight 说明了目的矩形及源位图的宽和高。pSrcDC 指向源设备上下文，xSrc 和 ySrc 说明了源矩形相对于源位图左上角的偏移。参数 dwRop 指定了光栅操作(ROP)代码，一些常用的 ROP 代码如表 11.2 所示。

表 11.2 常用的 ROP 代码

ROP 码	含义
BLACKNESS	输出黑色
DSTINVERT	反转目的位图
MERGECOPY	用与操作把图案(Pattern)与源位图融合起来
MERGEPAINT	用或操作把反转的源位图与目的位图融合起来
NOTsrcCOPY	把源位图反转然后拷贝到目的地
NOTsrcERASE	用或操作融合源和目的位图，然后再反转
PATCOPY	把图案拷贝到目的位图中
PATINVERT	用异或操作把图案与目的位图相融合
PATPAINT	用或操作融合图案和反转的源位图，然后用或操作把结果与目的位图融合
srcAND	用与操作融合源位图和目的位图
srcCOPY	把源位图拷贝到目的位图
srcERASE	先反转目的位图，再用与操作将其与源位图融合
srcINVERT	用异或操作融合源位图和目的位图
srcPAINT	用或操作融合源位图和目的位图
WHITENESS	输出白色

函数 CDC::StretchBlt 的声明为：

BOOL StretchBlt(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, int nSrcWidth, int nSrcHeight, DWORD dwRop);

该函数把位图从源矩形拷贝到目的矩形中，如果源和目的矩形尺寸不同，那么将缩放位图的功能以适应目的矩形的大小。函数的大部分参数与 BitBlt 的相同，但多了两个参数 nSrcWidth 和 nSrcHeight 用来指定源矩形的宽和高。

DDB 的一个重要用途是用作设备上下文的显示表面。每一个设备上下文都包含有一个 DDB，该位图实际上是在显示设备的缓冲区中(如视频内存)，我们可以把它看做设备上下文的显示表面，设备上下文用 GDI 函数绘图实际上就是修改它所包含的 DDB(显示表面)的过程。

普通的设备上下文都是在屏幕上绘图的，而使用内存设备上下文则可以在系统内存中绘制图形。内存设备上下文是一种特殊的设备上下文，它将系统内存用作显示表面。程序可以使用内存设备上下文预先在系统内存中绘制复杂的图形，然后再快速地将其复制到实际的设备上下文的显示表面上，而绘制图形的结果仍保存在内存设备上下文的 DDB 中。

提示：有人可能会想到用 BitBlt 函数把绘图结果从显示设备拷贝到内存设备上下文中，这种方法可以工作，但有时会出错。当源矩形被别的窗口遮住时，BitBlt 会把别的窗口中的像素拷贝下来。/

内存设备上下文缺省的 DDB 是一个 1×1 的单色位图，如此小的显示表面显然是没有用的，因此程序一般要为内存设备对象选择一个合适

大小的彩色 DDB。

下面这段代码创建了一个内存设备上下文，并在其包含的 DDB 中画了一个灰色实心矩形，然后再把 DDB 输出到屏幕上。

```
void CMyView::OnDraw(CDC* pDC)
{
. . .
CDC MemDC;
CBitmap bm,*oldBmp;
MemDC.CreateCompatibleDC(pDC);  //创建一个兼容的内存设备上下文
bm.CreateCompatibleBitmap(pDC,100,50); //创建一个兼容的 DDB
oldBmp=MemDC.SelectObject(&bm);
MemDC.SelectStockObject(BLACK_PEN);
MemDC.SelectStockObject(GRAY_BRUSH);
MemDC.Rectangle(0,0,50,50); //在 DDB 中画一个矩形
pDC->BitBlt(0,0,100,50,&MemDC,0,0,srcCOPY);
MemDC.SelectObject(oldBmp); //使位图 bm 对象脱离设备上下文
. . .
}
```

在上面的代码中，绘图的结果保存在位图 bm 中，一旦调用 MemDC.SelectObject(oldBmp)使位图 bm 脱离设备上下文，该位图就可以被其它对象使用。

11.4 与设备无关的位图(DIB)

DIB(Device-indepentent bitmap)的与设备无关性主要体现在以下两个方面：

DIB 的颜色模式与设备无关。例如，一个 256 色的 DIB 即可以在真彩色显示模式下使用，也可以在 16 色模式下使用。

256 色以下(包括 256 色)的 DIB 拥有自己的颜色表，像素的颜色独立于系统调色板。

由于 DIB 不依赖于具体设备，因此可以用来永久性地保存图象。DIB 一般是以*.BMP 文件的形式保存在磁盘中的，有时也会保存在*.DIB 文件中。运行在不同输出设备下的应用程序可以通过 DIB 来交换图象。

DIB 还可以用一种 RLE 算法来压缩图像数据，但一般来说 DIB 是不压缩的。

11.4.1 DIB 的结构

与 Borland C++下的框架类库 OWL 不同，MFC 未提供现成的类来封装 DIB。尽管 Microsoft 列出了一些理由，但没有 DIB 类确实给 MFC 用户带来很多不便。用户要想使用 DIB，首先应该了解 DIB 的结构。

在内存中，一个完整的 DIB 由两部分组成：一个 BITMAPINFO 结构和一个存储像素阵列的数组。BITMAPINFO 描述了位图的大小，颜色模式和调色板等各种属性，其定义为

```
typedef struct tagBITMAPINFO {
BITMAPINFOHEADER bmiHeader;
RGBQUAD bmiColors[1]; //颜色表
} BITMAPINFO;
```

RGBQUAD 结构用来描述颜色，其定义为

```
typedef struct tagRGBQUAD {
BYTE rgbBlue; //蓝色的强度
BYTE rgbGreen; //绿色的强度
BYTE rgbRed; //红色的强度
BYTE rgbReserved; //保留字节，为 0
} RGBQUAD;
```

注意，RGBQUAD 结构中的颜色顺序是 BGR，而不是平常的 RGB。

BITMAPINFOHEADER 结构包含了 DIB 的各种信息，其定义为

```
typedef struct tagBITMAPINFOHEADER{
DWORD biSize; //该结构的大小
LONG biWidth; //位图的宽度(以像素为单位)
LONG biHeight; //位图的高度(以像素为单位)
WORD biPlanes; //必须为 1
WORD biBitCount //每个像素的位数(1、4、8、16、24 或 32)
DWORD biCompression; //压缩方式，一般为 0 或 BI_RGB (未压缩)
DWORD biSizeImage; //以字节为单位的图象大小(仅用于压缩位图)
LONG biXPelsPerMeter; //以目标设备每米的像素数来说明位图的水
```

平分辨率

LONG biYPelsPerMeter; //以目标设备每米的像素数来说明位图的垂直分辨率

DWORD biClrUsed; /*颜色表的颜色数，若为 0 则位图使用由 biBitCount 指定的最大颜色数*/

DWORD biClrImportant; //重要颜色的数目，若该值为 0 则所有颜色都重要

} BITMAPINFOHEADER;

与 DDB 不同，DIB 的字节数组是从图象的最下面一行开始的逐行向上存储的，也即等于把图象倒过来然后在逐行扫描。另外，字节数组中每个扫描行的字节数必需是 4 的倍数，如果不足要用 0 补齐。

DIB 可以存储在 *.BMP 或 *.DIB 文件中。DIB 文件是以 BITMAPFILEHEADER 结构开头的，该结构的定义为

typedef struct tagBITMAPFILEHEADER {

WORD bfType; //文件类型，必须为“ BM ”

DWORD bfSize; //文件的大小

WORD bfReserved1; //为 0

WORD bfReserved2; //为 0

DWORD bfOffBits; //存储的像素阵列相对于文件头的偏移量

} BITMAPFILEHEADER;

紧随该结构的是一个 BITMAPINFOHEADER 结构，然后是 RGBQUAD 结构组成的颜色表(如果有的话)，文件最后存储的是 DIB 的像素阵列。

DIB 的颜色信息储存在自己的颜色表中，程序一般要根据颜色表为 DIB 创建逻辑调色板。在输出一幅 DIB 之前，程序应该将其逻辑调色板选入到相关的设备上下文中并实现到系统调色板中，然后再调用相关的 GDI 函数(如::SetDIBitsToDevice 或::StretchDIBits)输出 DIB。在输出过程中，GDI 函数会把 DIB 转换成 DDB，这项工作主要包括以下两步：

将 DIB 的颜色格式转换成与输出设备相同的颜色格式。例如，在真彩色的显示模式下要显示一个 256 色的 DIB，则应该将其转换成 24 位的颜色格式。

将 DIB 像素的逻辑颜色索引转换成系统调色板索引。

11.4.2 编写 DIB 类

由于 MFC 未提供 DIB 类，用户在使用 DIB 时将面临繁重的 Windows API 编程任务。幸运的是，Visual C++提供了一个较高层次的 API，简化了 DIB 的使用。这些 API 函数实际上是由 MFC 的 DibLook 例程提供的，它们位于 DibLook 目录下的 dibapi.cpp、myfile.cpp 和 dibapi.h 文件中，主要包括：

ReadDIBFile //把 DIB 文件读入内存

SaveDIB //把 DIB 保存到文件中

CreateDIBPalette //从 DIB 中创建一个逻辑调色板

PaintDIB //显示 DIB

DIBWidth //返回 DIB 的宽度

DIBHeight //返回 DIB 的高度

如果读者对这些函数的内部细节感兴趣，那么可以研究一下 dibapi.cpp 和 myfile.cpp 文件，但要做好吃苦的准备。

即使利用上述 API，编写使用 DIB 的程序仍然不是很轻松。为了满足读者的要求，笔者编写了一个名为 CDib 的较简单的 DIB 类，该类是基于上述 API 的，它的主要成员函数包括：

BOOL Load(LPCTSTR lpszFileName);该函数从文件中载入 DIB，参数 lpszFileName 说明了文件名。若成功载入则函数返回 TRUE，否则返回 FALSE。

BOOL LoadFromResource(UINT nID);该函数从资源中载入位图，参数 nID 是资源位图的 ID。若成功载入则函数返回 TRUE，否则返回 FALSE。

CPalette* GetPalette()返回 DIB 的逻辑调色板。

BOOL Draw(CDC *pDC, int x, int y, int cx=0, int cy=0);该函数在指定的矩形区域内显示 DIB，它具有缩放位图的功能。参数 pDC 指向用于绘图的设备上下文，参数 x 和 y 说明了目的矩形的左上角坐标，cx 和 cy 说明了目的矩形的尺寸，cx 和 cy 若有一个为 0 则该函数按 DIB 的实际大小绘制位图，cx 和 cy 的缺省值是 0。若成功则函数返回 TRUE，否则返回 FALSE。

int Width(); //以像素为单位返回 DIB 的宽度

int Height(); //以像素为单位返回 DIB 的高度

CDib 类的源代码在清单 11.3 和 11.4 列出，CDib 类的定义位于 CDib.h 中，CDib 类的成员函数代码位于 CDib.cpp 中。对于 CDib 类的代码这里就不作具体解释了，读者只要会用就行。

清单 11.3 CDib.h

```
#if !defined MYDIB
#define MYDIB
#include “dibapi.h”
class CDib
{
public:
CDib();
~CDib();
protected:
HDIB m_hDIB;
CPalette* m_palDIB;
public:
BOOL Load(LPCTSTR lpszFileName);
BOOL LoadFromResource(UINT nID);
CPalette* GetPalette() const
{ return m_palDIB; }
```

```
BOOL Draw(CDC *pDC, int x, int y, int cx=0, int cy=0);
int Width();
int Height();
void DeleteDIB();
};
#endif
```
清单 11.4 Cdib.cpp
```
#include <stdafx.h>
#include “CDib.h”
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
CDib::CDib()
{
m_palDIB=NULL;
m_hDIB=NULL;
}
CDib::~CDib()
{
DeleteDIB();
}
void CDib::DeleteDIB()
{
if (m_hDIB != NULL)
::GlobalFree((HGLOBAL) m_hDIB);
if (m_palDIB != NULL)
delete m_palDIB;
}
```
//**从文件中载入** DIB
```
BOOL CDib::Load(LPCTSTR lpszFileName)
{
HDIB hDIB;
CFile file;
CFileException fe;
if  (!file.Open(lpszFileName,  CFile::modeRead|CFile::shareDenyWrite,
&fe))
{
AfxMessageBox(fe.m_cause);
return FALSE;
}
TRY
```

```
{
hDIB = ::ReadDIBFile(file);
}
CATCH (CFileException, eLoad)
{
file.Abort();
return FALSE;
}
END_CATCH
DeleteDIB(); //清除旧位图
m_hDIB=hDIB;
m_palDIB = new CPalette;
if (::CreateDIBPalette(m_hDIB, m_palDIB) == NULL)
{
// DIB 有可能没有调色板
delete m_palDIB;
m_palDIB = NULL;
}
return TRUE;
}
//从资源中载入 DIB
BOOL CDib::LoadFromResource(UINT nID)
{
HINSTANCE hResInst = AfxGetResourceHandle();
HRsrc hFindRes;
HDIB hDIB;
LPSTR pDIB;
LPSTR pRes;
HGLOBAL hRes;
//搜寻指定的资源
hFindRes = ::FindResource(hResInst, MAKEINTRESOURCE(nID),
RT_BITMAP);
if (hFindRes == NULL) return FALSE;
hRes = ::LoadResource(hResInst, hFindRes); //载入位图资源
if (hRes == NULL) return FALSE;
DWORD dwSize=::SizeofResource(hResInst,hFindRes);
hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE |
GMEM_ZEROINIT, dwSize);
if (hDIB == NULL) return FALSE;
pDIB = (LPSTR)::GlobalLock((HGLOBAL)hDIB);
pRes = (LPSTR) ::LockResource(hRes);
memcpy(pDIB, pRes, dwSize); //把 hRes 中的内容复制 hDIB 中
```

```
::GlobalUnlock((HGLOBAL) hDIB);
DeleteDIB();
m_hDIB=hDIB;
m_palDIB = new CPalette;
if (::CreateDIBPalette(m_hDIB, m_palDIB) == NULL)
{
// DIB 有可能没有调色板
delete m_palDIB;
m_palDIB = NULL;
}
return TRUE;
}
int CDib::Width()
{
if(m_hDIB==NULL) return 0;
LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);
int cxDIB = (int) ::DIBWidth(lpDIB); // Size of DIB - x
::GlobalUnlock((HGLOBAL) m_hDIB);
return cxDIB;
}
int CDib::Height()
{
if(m_hDIB==NULL) return 0;
LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);
int cyDIB = (int) ::DIBHeight(lpDIB); // Size of DIB - y
::GlobalUnlock((HGLOBAL) m_hDIB);
return cyDIB;
}
//显示 DIB，该函数具有缩放功能
//参数 x 和 y 说明了目的矩形的左上角坐标，cx 和 cy 说明了目的矩形的尺寸
//cx 和 cy 若有一个为 0 则该函数按 DIB 的实际大小绘制，cx 和 cy 的缺省值是 0
BOOL CDib::Draw(CDC *pDC, int x, int y, int cx, int cy)
{
if(m_hDIB==NULL) return FALSE;
CRect rDIB,rDest;
rDest.left=x;
rDest.top=x;
if(cx==0||cy==0)
{
cx=Width();
```

```
cy=Height();
}
rDest.right=rDest.left+cx;
rDest.bottom=rDest.top+cy;
rDIB.left=rDIB.top=0;
rDIB.right=Width();
rDIB.bottom=Height();
return ::PaintDIB(pDC->GetSafeHdc(),&rDest,m_hDIB,&rDIB,m_palDIB);
}
```

11.4.3 **使用** CDib **类的例子**

现在让我们来看一个使用 CDib **类的例子。如图** 11.4 **所示，程序名为** ShowDib**，是一个多文档应用程序，它的功能与** VC **的** DibLook **例程有些类似，可同时打开和显示多个位图。**

图 11.4 **用** ShowDib **来显示位图**

请读者用 AppWizard **建立一个名为** ShowDib **的** MFC **工程。程序应该用滚动视图来显示较大的位图，所以在** MFC AppWizard **的第** 6 **步应把** CShowDibView **的基类改为** CScrollView**。**

由于 ShowDib **程序要用到** CDib **类，所以应该把** dibapi.cpp、myfile.cpp、dibapi.h、CDib.cpp **和** CDib.h **文件拷贝到** ShowDib **目录下，并选择** Project->Add to Project->Files **命令把这些文件加到** ShowDib **工程中。**

在 ShowDib.h **文件中** CShowDibApp **类的定义之前加入下面一行：**

```
#define WM_DOREALIZE WM_USER+200
```

当收到调色板消息时，主框架窗口会发送用户定义的 WM_DOREALIZE **消息通知视图。**

接下来，需要用 ClassWizard 为 CMainFrame 加入 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息的处理函数，为 CShowDibDoc 类加入 OnOpenDocument 函数。

最后，请读者按清单 11.5、11.6 和 11.7 修改程序。

清单 11.5 CMainFrame 类的部分代码

```
// MainFrm.cpp : implementation of the CMainFrame class
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
CMDIFrameWnd::OnPaletteChanged(pFocusWnd);
// TODO: Add your message handler code here
SendMessageToDescendants(WM_DOREALIZE,  1);  //通知所有的子窗口
}
BOOL CMainFrame::OnQueryNewPalette()
{
// TODO: Add your message handler code here and/or call default
CMDIChildWnd* pMDIChildWnd = MDIGetActive();
if (pMDIChildWnd == NULL)
return FALSE; //  没有活动的 MDI 子框架窗口
CView* pView = pMDIChildWnd->GetActiveView();
pView->SendMessage(WM_DOREALIZE,0); //只通知活动视图
return TRUE; //返回 TRUE 表明实现了逻辑调色板
}
```

清单 11.6 CShowDibDoc 类的部分代码

```
// ShowDibDoc.h : interface of the CShowDibDoc class
#include “CDib.h”
class CShowDibDoc : public CDocument
{
. . .
// Attributes
public:
CDib m_Dib;
. . .
};
// ShowDibDoc.cpp : implementation of the CShowDibDoc class
BOOL CShowDibDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
if (!CDocument::OnOpenDocument(lpszPathName))
return FALSE;
// TODO: Add your specialized creation code here
BeginWaitCursor();
BOOL bSuccess=m_Dib.Load(lpszPathName); //载入 DIB
```

```
EndWaitCursor();
return bSuccess;
}
```

清单 11.7 CShowDibView **类的部分代码**

```
// ShowDibView.h : interface of the CShowDibView class
class CShowDibView : public CScrollView
{
. . .
afx_msg LRESULT OnDoRealize(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};
// ShowDibView.cpp : implementation of the CShowDibView class
BEGIN_MESSAGE_MAP(CShowDibView, CScrollView)
. . .
ON_MESSAGE(WM_DOREALIZE, OnDoRealize)
END_MESSAGE_MAP()
void CShowDibView::OnInitialUpdate()
{
CScrollView::OnInitialUpdate();
CSize sizeTotal;
// TODO: calculate the total size of this view
CShowDibDoc* pDoc = GetDocument();
sizeTotal.cx = pDoc->m_Dib.Width();
sizeTotal.cy = pDoc->m_Dib.Height();
SetScrollSizes(MM_TEXT, sizeTotal); //设置视图的滚动范围
}
void CShowDibView::OnActivateView(BOOL bActivate, CView* pActivateView, CView* pDeactiveView)
{
// TODO: Add your specialized code here and/or call the base class
if(bActivate)
OnDoRealize(0,0); //刷新视图
CScrollView::OnActivateView(bActivate, pActivateView, pDeactiveView);
}
LRESULT CShowDibView::OnDoRealize(WPARAM wParam, LPARAM)
{
CClientDC dc(this);
//wParam 参数决定了该视图是否实现前景调色板
dc.SelectPalette(GetDocument()->m_Dib.GetPalette(),wParam);
```

```
if(dc.RealizePalette())
GetDocument()->UpdateAllViews(NULL);
return 0L;
}
void CShowDibView::OnDraw(CDC* pDC)
{
CShowDibDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
pDoc->m_Dib.Draw(pDC,0,0); //输出 DIB
}
```

在程序中使用 CDib 对象的代码很简单。当用户在 ShowDib 程序中选择 File->Open 命令并从打开文件对话框中选择了一个 BMP 文件后，CShowDibDoc::OnOpenDocument 函数被调用，该函数调用 CDib::Load 载入位图。在 CShowDibView::OnDraw 中，调用 CDib::Draw 输出位图。在 CShowDibView::OnInitialUpdate 中，根据 DIB 的尺寸来确定视图的滚动范围。

需要重点研究的是 ShowDib 如何处理调色板问题的。ShowDib 是一个多文档应用程序，可以同时显示多幅位图。由于每个位图一般都有不同的调色板，这样就产生了共享系统调色板的问题。程序必须采取措施来保证只有一个视图的逻辑调色板作为前景调色板使用。

当主框架窗口收到 WM_QUERYNEWPALETTE 消息时，主框架窗口向具有输入焦点的视图发送 wParam 参数为 0 的 WM_DOREALIZE 消息，该视图的消息处理函数 CShowDibView::OnDoRealize 为视图实现前景调色板并在必要时重绘视图，这样活动视图中的位图就具有最佳颜色显示。

如果活动视图在实现其前景调色板时改变了系统调色板，或是别的应用程序的前景调色板改变了系统调色板，那么 Windows 会向所有顶层窗口和重叠窗口发送 WM_PALETTECHANGED 消息，DibLook 的主框架窗口也会收到该消息。主框架窗口对该消息的处理是向所有的视图发送 wParam 参数为 1 的 WM_DOREALIZE 消息，通知它们实现各自的背景调色板并在必要时重绘，这样所有的位图都能显示令人满意的颜色。

当某一视图被激活时，需要调用 OnDoRealize 来实现其前景调色板，这一任务由 CShowDibView:: OnActivateView 函数来完成。

11.5 动画控件

Windows 95 支持一种动画控件(Animate control)，动画控件可以播放 AVI 格式的动画片(AVI Clip)，动画片可以来自一个 AVI 文件，也可以来自资源中。合理地使用动画控件，可以使程序的界面更加形象生动。

11.5.1 动画控件的使用

MFC 的 CAnimateCtrl 类封装了动画控件，该类的 Create 成员函数负责创建动画控件，其声明为：

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

参数 dwStyle 是如表 11.3 所示的控件风格的组合，参数 rect 指定了控件的尺寸，pParentWnd 指向父窗口，nID 是控件的 ID。若创建成功则函数返回 TRUE。

表 11.3 动画控件的风格

风格	含义
ACS_CENTER	使动画片居于控件中央，并使动画片打开后控件窗口的尺寸和位置保持不变。如果不指定该风格，则控件的尺寸会自动调整来适应动画片的大小。
ACS_TRANSPARENT	使动画片的背景透明(不输出动画片的背景色)。
ACS_AUTOPLAY	一旦打开动画片后就一直重复播放。

除表中的风格外，一般还要为动画控件指定 WS_CHILD、WS_VISIBLE 和 WS_BORDER 窗口风格。例如，要创建一个能自动播放的动画控件，应该指定其风格为 WS_CHILD|WS_VISIBLE|WS_BORDER|ACS_AUTOPLAY。

用户可以向对话框模板中加入动画控件，在模板编辑器的控件面板上，动画控件是用一个电影胶片的图形来表示的。在动画控件的属性对话框中可以指定上表列出的风格。只要不指定 ACS_CENTER 风格，用户就不必关心动画控件的尺寸，因为在打开动画片时控件的尺寸会被自动调整成动画片的幅面大小。

CAnimateCtrl 类主要的成员函数包括：

BOOL Open(LPCTSTR lpszFileName);BOOL Open(UINT nID);Open 函数从 AVI 文件或资源中打开动画片，如果参数 lpszFileName 或 nID 为 NULL，则系统将关闭以前打开的动画片。若成功则函数返回 TRUE。

BOOL Play(UINT nFrom, UINT nTo, UINT nRep);该函数用来播放动画片。参数 nFrom 指定了播放的开始帧的索引，索引值必须小于 65536，若为 0 则从头开始播放。nTo 指定了结束帧的索引，它的值必须小于 65536，若为-1 则表示播放到动画片的末尾。nRep 是播放的重复次数，若为-1 则无限重复播放。若成功则函数返回 TRUE。

BOOL Seek(UINT nTo);该函数用来静态地显示动画片的某一帧。参数 nTo 是帧的索引，其值必须小于 65536，若为 0 则显示第一帧，若为-1

则显示最后一帧。若成功则函数返回 TRUE。

BOOL Stop( );停止动画片的播放。若成功则函数返回 TRUE。

BOOL Close( );关闭并从内存中清除动画片。若成功则函数返回 TRUE。

一般来说，应该把动画片放在资源里，而不是单独的 AVI 文件中。这样做可以使应用程序更容易管理，否则，如果应用程序要附带一大堆 BMP 或 AVI 文件，会给人一种凌乱和不专业的感觉。Visual C++不直接支持 AVI 资源，但用户可以创建一种新的资源类型来包含 AVI。在 VC 的一个名为 cmnctrls 的 MFC 例子中提供了几个 AVI 文件(如 dillo.avi)，如果用户要把象 dillo.avi 这样的 AVI 文件包含到程序的资源中，则应按以下几步去做：

在程序的资源视图中单击鼠标右键，并在弹出菜单中选择 Import...命令。

在文件选择对话框中选择 dillo.avi 文件，按 Import 按钮退出。

按 Import 按钮退出后，会出现一个 Custom Resource Type 对话框，如图 11.5 所示。如果是第一次向资源中加入 AVI 文件，那么应该在 Resource type 编辑框中为动画片类资源起一个名字(如 AVI)，若以前已创建过 AVI 型资源，则可以在直接在列表框中选择 AVI 型。按 OK 后，dillo.avi 就被加入到资源中。

按 Alt+Enter 键后，可以在属性对话框中修改资源的 ID。

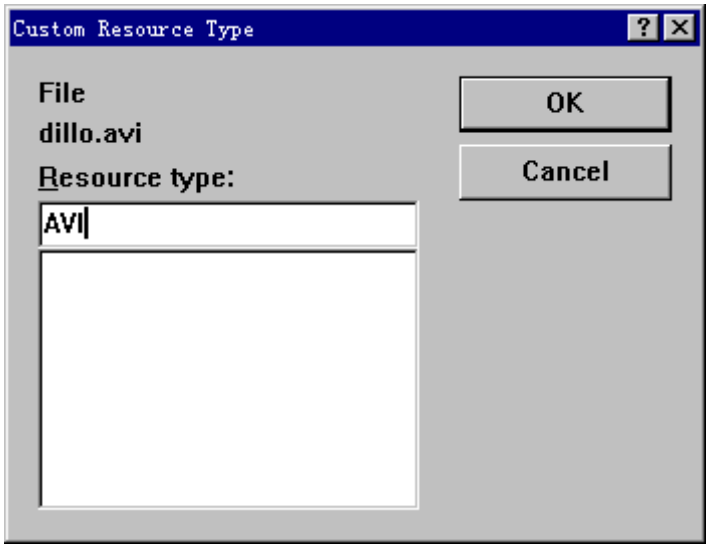


图 11.5 Custom Resource Type 对话框

创建动画控件的方法与创建普通控件相比并没有什么不同，用户可以用 ClassWizard 把动画控件和 CAnimateCtrl 对象联系起来。动画控件的使用很简单，下面的这段代码打开并不断重复播放一个资源动画，它们通常是位于 OnInitDialog 函数中：

```
m_AnimateCtrl.Open(IDR_AVI1);
m_AnimateCtrl.Play(0,-1,-1);
```

如果为动画控件指定了 ACS_AUTOPLAY 风格，则在调用 Open 后就会自动重复播放，不必调用 Play。程序一般不需要调用 Close 来关闭

动画片，因为这个任务在控件被删除时会自动完成。但如果在控件已包含一个动画片的情况下，需要打开一个新的动画片，则程序应先调用 Close 删除原来的动画片。

11.5.2 动画控件的局限

动画控件并不能播放所有的 AVI 文件，只有满足下列条件的 AVI 文件才能被播放：

AVI 文件必须是无声的，不能有声道。

AVI 文件必须是未压缩的，或是用 RLE 算法压缩的。

AVI 的调色板必须保持不变。

动画控件最大的局限性在于它只能显示系统调色板中缺省的颜色，因此如果用动画控件来播放一个 256 色的 AVI 文件，那么播放效果看起来就象一个 16 色的动画一样，很不理想。

总之，动画控件只能播放一些简单的，颜色数较少的 AVI 动画。如果要较满意地播放 256 色的 AVI 文件，就要利用下面介绍的 MCI 接口。

11.6 Win 32 的多媒体服务

Windows 95/NT 提供了丰富的多媒体服务功能，包括大量从低级到高级的多媒体 API 函数。利用这些功能强大的 API，用户可以在不同层次上编写多媒体应用程序。有关多媒体服务的内容完全可以写一本书，本节只是向读者简要地介绍一些最常用的多媒体服务。

在用 Visual C++开发多媒体应用时，用户必须在所有要用到多媒体函数的源程序中包含 MMSYSTEM.H 头文件，并且该文件位置应在 WINDOWS.H 头文件的后面。另外，在连接程序时要用到 WINMM.LIB 引入库，所以用户应该在 Project Settings 对话框的 Link 页的 Object/library modules 栏中加入 WINMM.LIB，或者在源程序中加入下面一行：

#pragma comment(lib, “winmm.lib”)

11.6.1 高级音频函数

Windows 提供了三个特殊的播放声音的高级音频函数：MessageBeep、PlaySound 和 sndPlaySound。这三个函数可以满足播放波形声音的一般需要，但它们播放的 WAVE 文件(波形声音文件)的大小不能超过 100KB，如果要播放较大的 WAVE 文件，则应该使用 MCI 服务。

MessageBeep 读者已经用过了，该函数主要用来播放系统报警声音。系统报警声音是由用户在控制面板中的声音(Sounds)程序中定义的，或者在 WIN.INI 的[sounds]段中指定。该函数的声明为：

BOOL MessageBeep(UINT uType);

参数 uType 说明了告警级，如表 11.4 所示。若成功则函数返回 TRUE。

表 11.4 系统告警级

级别	描述
-1	从机器的扬声器中发出蜂鸣声。
MB_ICONASTERISK	播放由 SystemAsterisk 定义的声音。
MB_ICONEXCLAMATION	播放由 SystemExclamation 定义的声音。
MB_ICONHAND	播放由 SystemHand 定义的声音。
MB_ICONQUESTION	播放由 SystemQuestion 定义的声音。
MB_OK	播放由 SystemDefault 定义的声音

在开始播放后，MessageBeep 函数立即返回。如果该函数不能播放指定的报警声音，它就播放 SystemDefault 定义的系统缺省声音，如果连系统缺省声音也播放不了，那么它就会在计算机的扬声器上发出嘟嘟声。在缺省时上表的 MB_系列声音均未定义。

MessageBeep 只能用来播放少数定义的声音，如果程序需要播放数字音频文件(*.WAV 文件)或音频资源，就需要使用 PlaySound 或 sndPlaySound 函数。

PlaySound 函数的声明为：

BOOL PlaySound(LPCSTR pszSound, HMODULE hmod,DWORD fdwSound);

参数 pszSound 是指定了要播放声音的字符串，该参数可以是 WAVE 文件的名字，或是 WAV 资源的名字，或是内存中声音数据的指针，或是

在系统注册表 WIN.INI 中定义的系统事件声音。如果该参数为 NULL 则停止正在播放的声音。参数 hmod 是应用程序的实例句柄，当播放 WAV 资源时要用到该参数，否则它必须为 NULL。参数 fdwSound 是标志的组合，如表 11.5 所示。若成功则函数返回 TRUE，否则返回 FALSE。

表 11.5 播放标志

标志	含义
SND_APPLICATION	用应用程序指定的关联来播放声音。
SND_ALIAS	pszSound 参数指定了注册表或 WIN.INI 中的系统事件的别名。
SND_ALIAS_ID	pszSound 参数指定了预定义的声音标识符。
SND_ASYNC	用异步方式播放声音，PlaySound 函数在开始播放后立即返回。
SND_FILENAME	pszSound 参数指定了 WAVE 文件名。
SND_LOOP	重复播放声音，必须与 SND_ASYNC 标志一块使用。
SND_MEMORY	播放载入到内存中的声音，此时 pszSound 是指向声音数据的指针。
SND_NODEFAULT	不播放缺省声音，若无此标志，则 PlaySound 在没找到声音时会播放缺省声音。
SND_NOSTOP	PlaySound 不打断原来的声音播出并立即返回 FALSE。
SND_NOWAIT	如果驱动程序正忙则函数就不播放声音并立即返回。
SND_PURGE	停止所有与调用任务有关的声音。若参数 pszSound 为 NULL，就停止所有的声音，否则，停止 pszSound 指定的声音。
SND_RESOURCE	pszSound 参数是 WAVE 资源的标识符，这时要用到 hmod 参数。
SND_SYNC	同步播放声音，在播放完后 PlaySound 函数才返回。

在 C:\WINDOWS\MEDIA 目录下有一个名为 The Microsoft Sound.wav 的声音文件，在 Windows 95 启动时会播放这个声音。下面我们用三种方法来调用 PlaySound 函数播出 Windows 95 的启动声音。

第一种方法是直接播出声音文件，相应的代码为：

PlaySound(“c:\\win95\\media\\The Microsoft Sound.wav”, NULL, SND_FILENAME | SND_ASYNC);

注意参数中的路径使用两个连续的反斜杠转义代表一个反斜杠。

第二种方法是把声音文件加入到资源中，然后从资源中播放声音。Visual C++支持 WAVE 型资源，用户在资源视图中单击鼠标右键并选择 Import 命令，然后在文件选择对话框中选择 The Microsoft Sound.wav 文件，则该文件就会被加入到 WAVE 资源中。假定声音资源的 ID 为 IDR_STARTWIN，则下面的调用同样会输出启动声音：

PlaySound((LPCTSTR)IDR_STARTWIN, AfxGetInstanceHandle(),

SND_RESOURCE | SND_ASYNC);

第三种方法是用 PlaySound **播放系统声音，**Windows **启动的声音是由** SystemStart **定义的系统声音，因此可以用下面的方法播放启动声音：**

PlaySound(“SystemStart”,NULL,SND_ALIAS|SND_ASYNC);

函数 sndPlaySound **的功能与** PlaySound **类似，但少了一个参数。函数的声明为：**

BOOL sndPlaySound(LPCSTR lpszSound, UINT fuSound);

除了不能指定资源名字外，参数 lpszSound **与** PlaySound **的是一样的。参数** fuSound **是如何播放声音的标志，可以是** SND_ASYNC、SND_LOOP、SND_MEMORY、SND_NODEFAULT、SND_NOSTOP **和** SND_SYNC **的组合，这些标志的含义与** PlaySound **的一样。**

可以看出，sndPlaySound **不能直接播放声音资源。要用该函数播放** WAVE **文件，可按下面的方式调用：**

sndPlaySound(“MYSOUND.WAV”,SND_ASYNC);

11.6.2 MCI

MCI(Media Control Interface**，媒体控制接口)向** Windows **程序提供了在高层次上控制媒体设备接口的能力。程序不必关心具体设备，就可以对激光唱机**(CD)**、视盘机、波形音频设备、视频播放设备和** MIDI **设备等媒体设备进行控制。对于程序员来说，可以把** MCI **理解为设备面板上的一排按键，通过选择不同的按键(发送不同的** MCI **命令)可以让设备完成各种功能，而不必关心设备内部实现。比如，对于** play**，视盘机和** CD **机有不同的反应(一个是播放视频，一个播放音频)，而对用户来说却只需要按同一按钮。**

应用程序通过向 MCI **发送命令来控制媒体设备。**MCI **命令接口分命令字符串和命令消息两种，两者具有相同的功能。命令字符串具有使用简单的特点，但是它的执行效率不如命令消息。**

所有的 MCI **命令字符串都是通过多媒体** API **函数** mciSendString **传递给** MCI **的，该函数的声明为：**

MCIERROR mciSendString(
LPCTSTR lpszCommand, //MCI **命令字符串**
LPTSTR lpszReturnString, //**存放反馈信息的缓冲区**
UINT cchReturn, //**缓冲区的长度**
HANDLE hwndCallback //**回调窗口的句柄，一般为** NULL
); //**若成功则返回** 0**，否则返回错误码。**

该函数返回的错误码可以用 mciGetErrorString **函数进行分析，该函数的声明为：**

BOOL mciGetErrorString(
DWORD fdwError, //**函数** mciSendString **或** mciSendCommand **返回的错误码**
LPTSTR lpszErrorText, //**接收描述错误的字符串的缓冲区**
UINT cchErrorText //**缓冲区的长度**
);

下面是使用 mciSendString 函数的一个简单例子：

```
char buf[50];
MCIERROR mciError;
mciError=mciSendString(“ open cdaudio ”,buf,strlen(buf),NULL);
if(mciError)
{
mciGetErrorString(mciError,buf,strlen(buf));
AfxMessageBox(buf);
return;
}
```

open cdaudio 命令打开 CD 播放器，如果出错(如驱动器内没有 CD)则返回错误码，此时可以用 mciGetErrorString 函数取得错误信息字符串。open 是 MCI 打开设备的命令，cdaudio 是 MCI 设备名。MCI 的设备类型在表 11.6 列出。

表 11.6 MCI 设备类型

设备类型	描述
animation	动画设备
cdaudio	CD 播放器
dat	数字音频磁带机
digitalvideo	某一窗口中的数字视频(不基于 GDI)
other	未定义的 MCI 设备
overlay	重叠设备(窗口中的模拟视频)
scanner	图象扫描仪
sequencer	MIDI 序列器
videodisc	视盘机
waveaudio	播放数字波形文件的音频设备

请读者注意，设备类型和设备名是不同的概念。设备类型是指响应一组共用命令的一类 MCI 设备，而设备名则是某一个 MCI 设备的名字。系统需要用不同的设备名来区分属于同一设备类型的不同设备。

设备名是在注册表或 SYSTEM.INI 的[mci]部分定义的，典型的[mci]段如下所示：

```
[mci]
cdaudio=mcicda.drv
sequencer=mciseq.drv
waveaudio=mciwave.drv
avivideo=mciavi.drv
videodisc=mcipionr.drv
```

等号的左边是设备名，右边是对应的 MCI 驱动程序。当安装了新的 MCI 驱动程序时，系统要用不同的设备名来区分。设备名通常与驱动程序中的设备类型名相同，如 cdaudio 和 waveaudio 等，但也有例外，如 avivideo 设备是一个 digitalvideo 类型的设备。

使用 MCI 设备一般包括打开、使用和关闭三个过程。MCI 的大部分

命令可以控制不同的媒体设备。例如，可以用 play 命令来播放 WAVE 文件、视频文件或 CD。表 11.7 列出常用的 MCI 命令字符串，表中大部分命令都具有通用性。在 MCI 命令的后面一般要跟一个设备名以指定操作的对象。

表 11.7 常用的 MCI 命令

命令	描述
capacility	查询设备能力
close	关闭设备
info	查询设备的信息
open	打开设备
pause	暂停设备的播放或记录
play	开始设备播放
record	开始
resume	恢复暂停播放或记录的设备
seek	改变媒体的当前位置
set	改变设置
status	查询设备状态信息
stop	停止设备的播放或记录

例如，上面的例子打开了一个 CD 播放机后，可以发送常用的命令来控制 CD 机：

play cdaudio from <位置> to <位置>。若省略 from 则从当前磁道开始播放，若省略 to 则播放到结束。

pause cdaudio。暂停播放。

stop cdaudio。停止播放。

resume cdaudio。继续被暂停的播放。

status cdaudio number of tracks。查询 CD 的磁道数。status cdaudio current track 可以查询当前磁道。

seek cdaudio to <位置>。移动到指定磁道。

set cdaudio door open/closed。弹出或缩进 CD 盘。

close cdaudio。关闭设备。

MCI 设备可以按简单设备和复合设备进行分类。象 cdaudio 这样的设备不使用文件，我们称之为简单设备，而复合设备在播放时要用到数据文件，如数字视频(digitalvideo)和波形音频(waveaudio)设备，我们把这些数据文件叫做设备元素。

在打开一个复合设备时要指定设备名和设备元素。例如，下面命令打开一个波形音频设备：

open mysound.wav type waveaudio

可以只为复合设备指定设备元素，例如：

open mysound.wav

如下面所示，系统通过查找注册表或 WIN.INI 的[mci extensions]可以确定打开哪一个设备。

[mci extensions]

mid=Sequencer

rmi=Sequencer

wav=waveaudio

avi=AVIVideo

有时，程序需要多次打开同一设备来播放不同的数据文件。例如，谁也不能否认在屏幕上同时播放两个 AVI **文件的可能性，在这种情况下，需要为每次打开的设备起一个不同的别名，这样** MCI **才能区分两个播放设备。例如，下面这段代码打开并播放了两个** AVI **文件：**

```
char buf[50];
mciSendString("open dillo.avi type avivideo alias dillo",buf,strlen(buf),NULL);
mciSendString("play dillo repeat",buf,strlen(buf),NULL); //重复播放
mciSendString("open search.avi type avivideo alias search",buf,strlen(buf),NULL);
mciSendString("play search",buf,strlen(buf),NULL);
```

在用 open **命令打开设备时，如果指定了别名，则以后对该设备的操作都要使用别名。**

到目前为止，我们使用的都是 MCI **命令字符串。读者可能已经有了这样的体会，命令字符串具有简单易学的优点，但这种接口与** C/C++**的风格相去甚远，如果程序要查询和设置大量数据，那么用字符串的形式将很不方便。**

MCI **的命令消息接口提供了** C **语言接口，它速度更快，并且更能符合** C/C++**程序员的需要。所有** MCI **命令消息都是通过** mciSendCommand **函数发送的，该函数的声明为：**

```
MCIERROR mciSendCommand(
MCIDEVICEID IDDevice, //设备的 ID，在打开设备时不用该参数
UINT uMsg, //命令消息
DWORD fdwCommand, //命令消息的标志
DWORD dwParam //指向包含命令消息参数的结构
); //若成功则返回 0，否则返回错误码
```

清单 11.8 **的代码演示了用** MCI **命令消息来打开和重复播放一个** AVI **文件：**

清单 11.8

```
MCI_DGV_OPEN_PARMS mciOpen;
UINT wDeviceID;
MCIERROR mciError;
mciOpen.lpstrDeviceType = "avivideo"; //设备名
mciOpen.lpstrElementName = "dillo.avi"; //设备元素
mciError=mciSendCommand(0, MCI_OPEN,
MCI_OPEN_TYPE|MCI_OPEN_ELEMENT, //使用了设备元素
(DWORD)&mciOpen);
if(mciError)
{
```

```
char s[80];
mciGetErrorString(mciError,s,80);
AfxMessageBox(s);
return ;
}
wDeviceID=mciOpen.wDeviceID; //保存设备 ID
MCI_DGV_PLAY_PARMS mciPlay;
mciError=mciSendCommand(wDeviceID, MCI_PLAY,
MCI_DGV_PLAY_REPEAT,
(DWORD)&mciPlay);
. . .
```

可以看出，用命令消息比用命令字符串要复杂的多。命令消息与命令字符串是对应的，例如，open **与** MCI_OPEN **完成的是一样的功能。变量** wDeviceID **用来保存设备的** ID**，系统用** ID **来标识不同的设备，以保证命令发给正确的对象。**

限于篇幅，对 MCI **的命令消息就不作详细介绍了。**

小结

本章的要点包括：

在 Windows 系统中，逻辑调色板在应用程序和系统调色板中起缓冲作用。应用程序不能直接改变和使用系统调色板。程序如果要使用 20 种保留颜色以外的新颜色，应先将其逻辑调色板选入相应的设备上下文并实现到系统调色板中，然后通过引用逻辑调色板中的颜色来使用系统调色板。

当前活动的应用程序中只能有一个窗口的逻辑调色板作为前景调色板实现，其它调色板应作为背景调色板实现。

DDB 依赖于设备，只能在内存中使用。DIB 不依赖具体设备，可以在磁盘文件中保存。MFC 提供了 DDB 类 CBitmap，但未提供 DIB 类，用户需要自己编写 DIB 类。

利用高级音频函数可以播放较短的 WAVE 文件，而 MCI(媒体控制接口)对控制媒体设备提供了更好和更全面的支持。MCI 向 Windows 程序提供了在高层次上控制媒体设备接口的能力。程序不必关心具体设备，就可以对激光唱机(CD)、视盘机、波形音频设备、视频播放设备和 MIDI 设备等媒体设备进行控制。

第十二章　多线程与串行通信

Windows 是一个多任务操作系统。传统的 Windows 3.x 只能依靠应用程序之间的协同来实现协同式多任务，而 Windows 95/NT 实行的是抢先式多任务。

在 Win 32(95/NT)中，每一个进程可以同时执行多个线程，这意味着一个程序可以同时完成多个任务。对于象通信程序这样既要进行耗时的工作，又要保持对用户输入响应的应用来说，使用多线程是最佳选择。当进程使用多个线程时，需要采取适当的措施来保持线程间的同步。

利用 Win 32 的重叠 I/O 操作和多线程特性，程序员可以编写出高效的通信程序。在这一讲的最后将通过一个简单的串行通信程序，向读者演示多线程和重叠 I/O 的编程技术。

多任务、进程和线程

线程的同步

串行通信与重叠 I/O

一个通信演示程序

小结

12.1 多任务、进程和线程

12.1.1 Windows 3.x 的协同多任务

在 16 位的 Windows 3.x 中，应用程序具有对 CPU 的控制权。只有在调用了 GetMessage、PeekMessage、WaitMessage 或 Yield 后，程序才有可能把 CPU 控制权交给系统，系统再把控制权转交给别的应用程序。如果应用程序在长时间内无法调用上述四个函数之一，那么程序就一直独占 CPU，系统会被挂起而无法接受用户的输入。

因此，在设计 16 位的应用程序时，程序员必须合理地设计消息处理函数，以使程序能够尽快返回到消息循环中。如果程序需要进行费时的操作，那么必须保证程序在进行操作时能周期性的调用上述四个函数中的一个。

在 Windows 3.x 环境下，要想设计一个既能执行实时的后台工作(如对通信端口的实时监测和读写)，又能保证所有界面响应用户输入的单独的应用程序几乎是不可能的。

有人可能会想到用 CWinApp::OnIdle 函数来执行后台工作，因为该函数是程序主消息循环在空闲时调用的。但 OnIdle 的执行并不可靠，例如，如果用户在程序中打开了一个菜单或模态对话框，那么 OnIdle 将停止调用，因为此时程序不能返回到主消息循环中！在实时任务代码中调用 PeekMessage 也会遇到同样的问题，除非程序能保证用户不会选择菜单或弹出模态对话框，否则程序将不能返回到 PeekMessage 的调用处，这将导致后台实时处理的中断。

折衷的办法是在执行长期工作时弹出一个非模态对话框并禁止主窗口，在消息循环内分批执行后台操作。对话框中可以显示工作的进度，也可以包含一个取消按钮以让用户有机会中断一个长期的工作。典型的代码如清单 12.1 所示。这样做既可以保证工作实时进行，又可以使程序能有限地响应用户输入，但此时程序实际上已不能再为用户干别的事情了。

清单 12.1 在协同多任务环境下防止程序被挂起的一种方法

```
bAbort=FALSE;
lpMyDlgProc=MakeProcInstance(MyDlgProc, hInst);
hMyDlg=CreateDialog(hInst, “Abort”, hwnd, lpMyDlgProc); //创建一个非模态对话框
ShowWindow(hMyDlg, SW_NORMAL);
UpdateWindow(hMyDlg);
EnableWindow(hwnd, FALSE); //禁止主窗口
. . .
while(!bAbort)
{
. . . //执行一次后台操作
. . .
while(PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
```

```
{
if(!IsDialogMessage(hMyDlg, &msg))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}
}
EnableWindow(hwnd, TRUE); //允许主窗口
DestroyWindow(hMyDlg);
FreeProcInstance(lpMyDlgProc);
```

12.1.2 Windows 95/NT 的抢先式多任务

在 32 位的 Windows 系统中，采用的是抢先式多任务，这意味着程序对 CPU 的占用时间是由系统决定的。系统为每个程序分配一定的 CPU 时间，当程序的运行超过规定时间后，系统就会中断该程序并把 CPU 控制权转交给别的程序。与协同式多任务不同，这种中断是汇编语言级的。程序不必调用象 PeekMessage 这样的函数来放弃对 CPU 的控制权，就可以进行费时的工作，而且不会导致系统的挂起。

例如，在 Windows3.x 中，如果某一个应用程序陷入了死循环，那么整个系统都会瘫痪，这时唯一的解决办法就是重新启动机器。而在 Windows 95/NT 中，一个程序的崩溃一般不会造成死机，其它程序仍然可以运行，用户可以按 Ctrl+Alt+Del 键来打开任务列表并关闭没有响应的程序。

12.1.3 进程与线程

在 32 位的 Windows 系统中，术语多任务是指系统可以同时运行多个进程，而每个进程也可以同时执行多个线程。

进程就是应用程序的运行实例。每个进程都有自己私有的虚拟地址空间。每个进程都有一个主线程，但可以建立另外的线程。进程中的线程是并行执行的，每个线程占用 CPU 的时间由系统来划分。

可以把线程看成是操作系统分配 CPU 时间的基本实体。系统不停地在各个线程之间切换，它对线程的中断是汇编语言级的。系统为每一个线程分配一个 CPU 时间片，某个线程只有在分配的时间片内才有对 CPU 的控制权。实际上，在 PC 机中，同一时间只有一个线程在运行。由于系统为每个线程划分的时间片很小(20 毫秒左右)，所以看上去好象是多个线程在同时运行。

进程中的所有线程共享进程的虚拟地址空间，这意味着所有线程都可以访问进程的全局变量和资源。这一方面为编程带来了方便，但另一方面也容易造成冲突。

虽然在进程中进行费时的工作不会导致系统的挂起，但这会导致进程本身的挂起。所以，如果进程既要进行长期的工作，又要响应用户的输入，那么它可以启动一个线程来专门负责费时的工作，而主线程仍然可以与用户进行交互。

12.1.4 线程的创建和终止

线程分用户界面线程和工作者线程两种。用户界面线程拥有自己的消息泵来处理界面消息，可以与用户进行交互。工作者线程没有消息泵，一般用来完成后台工作。

MFC 应用程序的线程由对象 CWinThread 表示。在多数情况下，程序不需要自己创建 CWinThread 对象。调用 AfxBeginThread 函数时会自动创建一个 CWinThread 对象。

例如，清单 12.2 中的代码演示了工作者线程的创建。AfxBeginThread 函数负责创建新线程，它的第一个参数是代表线程的函数的地址，在本例中是 MyThreadProc。第二个参数是传递给线程函数的参数，这里假定线程要用到 CMyObject 对象，所以把 pNewObject 指针传给了新线程。线程函数 MyThreadProc 用来执行线程，请注意该函数的声明。线程函数有一个 32 位的 pParam 参数可用来接收必要的参数。

清单 12.2 创建一个工作者线程

```
//主线程
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
//新线程
UINT MyThreadProc( LPVOID pParam )
{
CMyObject* pObject = (CMyObject*)pParam;
if (pObject == NULL ||
!pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
return -1; // 非法参数
// 用 pObject 对象来完成某项工作
return 0; // 线程正常结束
}
```

AfxBeginThread 的声明为：

```
CWinThread* AfxBeginThread( AFX_THREADPROC pfnThreadProc,
LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL, UINT
nStackSize = 0, DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES
lpSecurityAttrs = NULL );
```

参数 pfnThreadProc 是工作线程函数的地址。pParam 是传递给线程函数的参数。nPriority 是线程的优先级，一般是 THREAD_PRIORITY_NORMAL，若为 0，则使用创建线程的优先级。nStackSize 说明了线程的堆栈尺寸，若为 0 则堆栈尺寸与创建线程相同。dwCreateFlags 指定了线程的初始状态，如果为 0，那么线程在创建后立即执行，如果为 CREATE_SUSPENDED，则线程在创建后就被挂起。参数 lpSecurityAttrs 用来说明保密属性，一般为 0。函数返回新建的 CWinThread 对象的指针。

程序应该把 AfxBeginThread 返回的 CWinThread 指针保存起来，以便对创建的线程进行控制。例如，可以调用 CWinThread::SetThreadPriority

来设置线程的优先级，用 CWinThread::SuspendThread 来挂起线程。如果线程被挂起，那么直到调用 CWinThread::ResumeThread 后线程才开始运行。

如果要创建用户界面线程，那么必须从 CWinThread 派生一个新类。事实上，代表进程主线程的 CWinApp 类就是 CWinThread 的派生类。派生类必须用 DECLARE_DYNCREATE 和 IMPLEMENT_DYNCREATE 宏来声明和实现。需要重写派生类的 InitInstance、ExitInstance、Run 等函数。

可以使用 AfxBeginThread 函数的另一个版本来创建用户界面线程。函数的声明为：

CWinThread* AfxBeginThread(CRuntimeClass* pThreadClass, int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0, DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

参数 pThreadClass 指向一个 CRuntimeClass 对象，该对象是用 RUNTIME_CLASS 宏从 CWinThread 的派生类创建的。其它参数以及函数的返回值与第一个版本的 AfxBeginThread 是一样的。

当发生下列事件之一时，线程被终止：

线程调用 ExitThread。

线程函数返回，即线程隐含调用了 ExitThread。

ExitProcess 被进程的任一线程显示或隐含调用。

用线程的句柄调用 TerminateThread。

用进程句柄调用 TerminateProcess。

12.2 线程的同步

多线程的使用会产生一些新的问题，主要是如何保证线程的同步执行。多线程应用程序需要使用同步对象和等待函数来实现同步。

12.2.1 为什么需要同步

由于同一进程的所有线程共享进程的虚拟地址空间，并且线程的中断是汇编语言级的，所以可能会发生两个线程同时访问同一个对象(包括全局变量、共享资源、API 函数和 MFC 对象等)的情况，这有可能导致程序错误。例如，如果一个线程在未完成对某一大尺寸全局变量的读操作时，另一个线程又对该变量进行了写操作，那么第一个线程读入的变量值可能是一种修改过程中的不稳定值。

属于不同进程的线程在同时访问同一内存区域或共享资源时，也会存在同样的问题。

因此，在多线程应用程序中，常常需要采取一些措施来同步线程的执行。需要同步的情况包括以下几种：

在多个线程同时访问同一对象时，可能产生错误。例如，如果当一个线程正在读取一个至关重要的共享缓冲区时，另一个线程向该缓冲区写入数据，那么程序的运行结果就可能出错。程序应该尽量避免多个线程同时访问同一个缓冲区或系统资源。

在 Windows 95 环境下编写多线程应用程序还需要考虑重入问题。Windows NT 是真正的 32 位操作系统，它解决了系统重入问题。而 Windows 95 由于继承了 Windows 3.x 的部分 16 位代码，没能够解决重入问题。这意味着在 Windows 95 中两个线程不能同时执行某个系统功能，否则有可能造成程序错误，甚至会造成系统崩溃。应用程序应该尽量避免发生两个以上的线程同时调用同一个 Windows API 函数的情况。

由于大小和性能方面的原因，MFC 对象在对象级不是线程安全的，只有在类级才是。也就是说，两个线程可以安全地使用两个不同的 CString 对象，但同时使用同一个 CString 对象就可能产生问题。如果必须使用同一个对象，那么应该采取适当的同步措施。

多个线程之间需要协调运行。例如，如果第二个线程需要等待第一个线程完成到某一步时才能运行，那么该线程应该暂时挂起以减少对 CPU 的占用时间，提高程序的执行效率。当第一个线程完成了相应的步骤后，应该发出某种信号来激活第二个线程。

12.2.2 等待函数

Win32 API 提供了一组能使线程阻塞其自身执行的等待函数。这些函数只有在作为其参数的一个或多个同步对象(见下小节)产生信号时才会返回。在超过规定的等待时间后，不管有无信号，函数也都会返回。在等待函数未返回时，线程处于等待状态，此时线程只消耗很少的 CPU 时间。

使用等待函数即可以保证线程的同步，又可以提高程序的运行效率。最常用的等待函数是 WaitForSingleObject，该函数的声明为：

DWORD WaitForSingleObject(HANDLE hHandle, DWORD

dwMilliseconds);

参数 hHandle **是同步对象的句柄。参数** dwMilliseconds **是以毫秒为单位的超时间隔，如果该参数为** 0**，那么函数就测试同步对象的状态并立即返回，如果该参数为** INFINITE**，则超时间隔是无限的。函数的返回值在表** 12.1 **中列出。**

表 12.1 WaitForSingleObject **的返回值**

返回值	**含义**
WAIT_FAILED	**函数失败**
WAIT_OBJECT_0	**指定的同步对象处于有信号的状态**
WAIT_ABANDONED	**拥有一个** mutex **的线程已经中断了，但未释放该** MUTEX
WAIT_TIMEOUT	**超时返回，并且同步对象无信号**

函数 WaitForMultipleObjects **可以同时监测多个同步对象，该函数的声明为：**

DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles, BOOL bWaitAll, DWORD dwMilliseconds);

参数 nCount **是句柄数组中句柄的数目。**lpHandles **代表一个句柄数组。**bWaitAll **说明了等待类型，如果为** TRUE**，那么函数在所有对象都有信号后才返回，如果为** FALSE**，则只要有一个对象变成有信号的，函数就返回。函数的返回值在表** 12.2 **中列出。参数** dwMilliseconds **是以毫秒为单位的超时间隔，如果该参数为** 0**，那么函数就测试同步对象的状态并立即返回，如果该参数为** INFINITE**，则超时间隔是无限的。**

表 12.2 WaitForMultipleObjects **的返回值**

返回值	**说明**
WAIT_OBJECT_0 **到** WAIT_OBJECT_0+nCount-1	**若** bWaitAll **为** TRUE**，则返回值表明所有对象都是有信号的。如果** bWaitAll **为** FALSE**，则返回值减去** WAIT_OBJECT_0 **就是数组中有信号对象的最小索引。**
WAIT_ABANDONED_0 **到** WAIT_ ABANDONED_0+nCount-1	**若** bWaitAll **为** TRUE**，则返回值表明所有对象都有信号，但有一个** mutex **被放弃了。若** bWaitAll **为** FALSE**，则返回值减去** WAIT_ABANDONED_0 **就是被放弃** mutex **在对象数组中的索引。**
WAIT_TIMEOUT	**超时返回。**

12.2.3 **同步对象**

同步对象用来协调多线程的执行，它可以被多个线程共享。线程的等待函数用同步对象的句柄作为参数，同步对象应该是所有要使用的线程都能访问到的。同步对象的状态要么是有信号的，要么是无信号的。同步对象主要有三种：事件、mutex **和信号灯。**

事件对象(Event)**是最简单的同步对象，它包括有信号和无信号两种状态。在线程访问某一资源之前，也许需要等待某一事件的发生，这时用事件对象最合适。例如，只有在通信端口缓冲区收到数据后，监视线程才被激活。**

事件对象是用 CreateEvent 函数建立的。该函数可以指定事件对象的种类和事件的初始状态。如果是手工重置事件，那么它总是保持有信号状态，直到用 ResetEvent 函数重置成无信号的事件。如果是自动重置事件，那么它的状态在单个等待线程释放后会自动变为无信号的。用 SetEvent 可以把事件对象设置成有信号状态。在建立事件时，可以为对象起个名字，这样其它进程中的线程可以用 OpenEvent 函数打开指定名字的事件对象句柄。

mutex 对象的状态在它不被任何线程拥有时是有信号的，而当它被拥有时则是无信号的。mutex 对象很适合用来协调多个线程对共享资源的互斥访问(mutually exclusive)。

线程用 CreateMutex 函数来建立 mutex 对象，在建立 mutex 时，可以为对象起个名字，这样其它进程中的线程可以用 OpenMutex 函数打开指定名字的 mutex 对象句柄。在完成对共享资源的访问后，线程可以调用 ReleaseMutex 来释放 mutex，以便让别的线程能访问共享资源。如果线程终止而不释放 mutex，则认为该 mutex 被废弃。

信号灯对象维护一个从 0 开始的计数，在计数值大于 0 时对象是有信号的，而在计数值为 0 时则是无信号的。信号灯对象可用来限制对共享资源进行访问的线程数量。线程用 CreateSemaphore 函数来建立信号灯对象，在调用该函数时，可以指定对象的初始计数和最大计数。在建立信号灯时也可以为对象起个名字，别的进程中的线程可以用 OpenSemaphore 函数打开指定名字的信号灯句柄。

一般把信号灯的初始计数设置成最大值。每次当信号灯有信号使等待函数返回时，信号灯计数就会减 1，而调用 ReleaseSemaphore 可以增加信号灯的计数。计数值越小就表明访问共享资源的程序越多。

除了上述三种同步对象外，表 12.3 中的对象也可用于同步。另外，有时可以用文件或通信设备作为同步对象使用。

表 12.3 可用于同步的对象

对象	描述
变化通知	由 FindFirstChangeNotification 函数建立，当在指定目录中发生指定类型的变化时对象变成有信号的。
控制台输入	在控制台建立是被创建。它是用 CONIN$调用 CreateFile 函数返回的句柄，或是 GetStdHandle 函数的返回句柄。如果控制台输入缓冲区中有数据，那么对象是有信号的，如果缓冲区为空，则对象是无信号的。
进程	当调用 CreateProcess 建立进程时被创建。进程在运行时对象是无信号的，当进程终止时对象是有信号的。
线程	当调用 Createprocess 、CreateThread 或 CreateRemoteThread 函数创建新线程时被创建。在线程运行是对象是无信号的，在线程终止时则是有信号的。

当对象不再使用时，应该用 CloseHandle 函数关闭对象句柄。

清单 12.3 是一个使用事件对象的简单例子，在该例中，假设主线程要读取共享缓冲区中的内容，而辅助线程负责向缓冲区中写入数据。两个线程使用了一个 hEvent 事件对象来同步。在用 CreateEvent 函数创建事件对象句柄时，指定该对象是一个自动重置事件，其初始状态为有信号的。当线程要读写缓冲区时，调用 WaitForSingleObject 函数无限等待 hEvent 信号。如果 hEvent 无信号，则说明另一线程正在访问缓冲区；如果有信号，则本线程可以访问缓冲区，WaitForSingleObject 函数在返回后会自动把 hEvent 置成无信号的，这样在本线程读写缓冲区时别的线程不会同时访问。在完成读写操作后，调用 SetEvent 函数把 hEvent 置成有信号的，以使别的线程有机会访问共享缓冲区。

清单 12.3 使用事件对象的简单例子

```
HANDLE hEvent; //全局变量
//主线程
hEvent=CreateEvent(NULL, FALSE, TRUE, NULL);
if(hEvent= =NULL) return;
. . .
WaitForSingleObject(hEvent, INFINITE);
ReadFromBuf( );
SetEvent( hEvent );
. . .
CloseHandle( hEvent );
//辅助线程
UINT MyThreadProc( LPVOID pParam )
{
. . .
WaitForSingleObject(hEvent, INFINITE);
WriteToBuf( );
SetEvent( hEvent );
. . .
return 0; //  线程正常结束
}
```

12.2.4 关键节和互锁变量访问

关键节(Critical Seciton)与 mutex 的功能类似，但它只能由同一进程中的线程使用。关键节可以防止共享资源被同时访问。

进程负责为关键节分配内存空间，关键节实际上是一个 CRITICAL_SECTION 型的变量，它一次只能被一个线程拥有。在线程使用关键节之前，必须调用 InitializeCriticalSection 函数将其初始化。如果线程中有一段关键的代码不希望被别的线程中断，那么可以调用 EnterCriticalSection 函数来申请关键节的所有权，在运行完关键代码后再用 LeaveCriticalSection 函数来释放所有权。如果在调用 EnterCriticalSection 时关键节对象已被另一个线程拥有，那么该函数将无限期等待所有权。

利用互锁变量可以建立简单有效的同步机制。使用函数 InterlockedIncrement 和 InterlockedDecrement 可以增加或减少多个线程共享的一个 32 位变量的值，并且可以检查结果是否为 0。线程不必担心会被其它线程中断而导致错误。如果变量位于共享内存中，那么不同进程中的线程也可以使用这种机制。

12.3 串行通信与重叠 I/O

Win 32 系统为串行通信提供了全新的服务。传统的 OpenComm、ReadComm 、 WriteComm 、 CloseComm 等函数已经过时，WM_COMMNOTIFY 消息也消失了。取而代之的是文件 I/O 函数提供的打开和关闭通信资源句柄及读写操作的基本接口。

新的文件 I/O 函数(CreateFile、ReadFile、WriteFile 等)支持重叠式输入输出，这使得线程可以从费时的 I/O 操作中解放出来，从而极大地提高了程序的运行效率。

12.3.1 串行口的打开和关闭

Win 32 系统把文件的概念进行了扩展。无论是文件、通信设备、命名管道、邮件槽、磁盘、还是控制台，都是用 API 函数 CreateFile 来打开或创建的。该函数的声明为：

```
HANDLE CreateFile(
LPCTSTR lpFileName, // 文件名
DWORD dwDesiredAccess, // 访问模式
DWORD dwShareMode, // 共享模式
LPSECURITY_ATTRIBUTES lpSecurityAttributes, // 通常为 NULL
DWORD dwCreationDistribution, // 创建方式
DWORD dwFlagsAndAttributes, // 文件属性和标志
HANDLE hTemplateFile // 临时文件的句柄，通常为 NULL
);
```

如果调用成功，那么该函数返回文件的句柄，如果调用失败，则函数返回 INVALID_HANDLE_VALUE。

如果想要用重叠 I/O 方式(参见 12.3.3)打开 COM2 口，则一般应象清单 12.4 那样调用 CreateFile 函数。注意在打开一个通信端口时，应该以独占方式打开，另外要指定 GENERIC_READ、GENERIC_WRITE、OPEN_EXISTING 和 FILE_ATTRIBUTE_NORMAL 等属性。如果要打开重叠 I/O，则应该指定 FILE_FLAG_OVERLAPPED 属性。

清单 12.4

```
HANDLE hCom;
DWORD dwError;
hCom=CreateFile(“COM2”, // 文件名
GENERIC_READ | GENERIC_WRITE, // 允许读和写
0, // 独占方式
NULL,
OPEN_EXISTING, //打开而不是创建
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, // 重叠方式
NULL
);
if(hCom = = INVALID_HANDLE_VALUE)
```

```
{
dwError=GetLastError( );
. . . // 处理错误
}
```

当不再使用文件句柄时，应该调用 CloseHandle 函数关闭之。

12.3.2 串行口的初始化

在打开通信设备句柄后，常常需要对串行口进行一些初始化工作。这需要通过一个 DCB 结构来进行。DCB 结构包含了诸如波特率、每个字符的数据位数、奇偶校验和停止位数等信息。在查询或配置置串行口的属性时，都要用 DCB 结构来作为缓冲区。

调用 GetCommState 函数可以获得串口的配置，该函数把当前配置填充到一个 DCB 结构中。一般在用 CreateFile 打开串行口后，可以调用 GetCommState 函数来获取串行口的初始配置。要修改串行口的配置，应该先修改 DCB 结构，然后再调用 SetCommState 函数用指定的 DCB 结构来设置串行口。

除了在 DCB 中的设置外，程序一般还需要设置 I/O 缓冲区的大小和超时。Windows 用 I/O 缓冲区来暂存串行口输入和输出的数据，如果通信的速率较高，则应该设置较大的缓冲区。调用 SetupComm 函数可以设置串行口的输入和输出缓冲区的大小。

在用 ReadFile 和 WriteFile 读写串行口时，需要考虑超时问题。如果在指定的时间内没有读出或写入指定数量的字符，那么 ReadFile 或 WriteFile 的操作就会结束。要查询当前的超时设置应调用 GetCommTimeouts 函数，该函数会填充一个 COMMTIMEOUTS 结构。调用 SetCommTimeouts 可以用某一个 COMMTIMEOUTS 结构的内容来设置超时。

有两种超时：间隔超时和总超时。间隔超时是指在接收时两个字符之间的最大时延，总超时是指读写操作总共花费的最大时间。写操作只支持总超时，而读操作两种超时均支持。用 COMMTIMEOUTS 结构可以规定读/写操作的超时，该结构的定义为：

```
typedef struct _COMMTIMEOUTS {
DWORD ReadIntervalTimeout; // 读间隔超时
DWORD ReadTotalTimeoutMultiplier; // 读时间系数
DWORD ReadTotalTimeoutConstant; // 读时间常量
DWORD WriteTotalTimeoutMultiplier; // 写时间系数
DWORD WriteTotalTimeoutConstant; // 写时间常量
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

COMMTIMEOUTS 结构的成员都以毫秒为单位。总超时的计算公式是：

总超时=时间系数×要求读/写的字符数 + 时间常量

例如，如果要读入 10 个字符，那么读操作的总超时的计算公式为：

读总超时 = ReadTotalTimeoutMultiplier × 10 + ReadTotalTimeoutConstant

可以看出，间隔超时和总超时的设置是不相关的，这可以方便通信程序灵活地设置各种超时。

如果所有写超时参数均为 0，那么就不使用写超时。如果 ReadIntervalTimeout 为 0，那么就不使用读间隔超时，如果 ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 都为 0，则不使用读总超时。如果读间隔超时被设置成 MAXDWORD 并且两个读总超时为 0，那么在读一次输入缓冲区中的内容后读操作就立即完成，而不管是否读入了要求的字符。

在用重叠方式读写串行口时，虽然 ReadFile 和 WriteFile 在完成操作以前就可能返回，但超时仍然是起作用的。在这种情况下，超时规定的是操作的完成时间，而不是 ReadFile 和 WriteFile 的返回时间。

清单 12.5 列出了一段简单的串行口初始化代码。

清单 12.5 打开并初始化串行口

```
HANDLE hCom;
DWORD dwError;
DCB dcb;
COMMTIMEOUTS TimeOuts;
hCom=CreateFile(“ COM2 ”, // 文件名
GENERIC_READ | GENERIC_WRITE, // 允许读和写
0, // 独占方式
NULL,
OPEN_EXISTING, //打开而不是创建
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, // 重叠方式
NULL
);
if(hCom = = INVALID_HANDLE_VALUE)
{
dwError=GetLastError( );
. . . // 处理错误
}
SetupComm( hCom, 1024, 1024 ) //缓冲区的大小为 1024
TimeOuts. ReadIntervalTimeout=1000;
TimeOuts.ReadTotalTimeoutMultiplier=500;
TimeOuts.ReadTotalTimeoutConstant=5000;
TimeOuts.WriteTotalTimeoutMultiplier=500;
TimeOuts.WriteTotalTimeoutConstant=5000;
SetCommTimeouts(hCom, &TimeOuts); // 设置超时
GetCommState(hCom, &dcb);
dcb.BaudRate=2400; // 波特率为 2400
dcb.ByteSize=8; // 每个字符有 8 位
dcb.Parity=NOPARITY; //无校验
```

dcb.StopBits=ONESTOPBIT; //一个停止位

SetCommState(hCom, &dcb);

12.3.3 重叠 I/O

在用 ReadFile 和 WriteFile 读写串行口时，既可以同步执行，也可以重叠(异步)执行。在同步执行时，函数直到操作完成后才返回。这意味着在同步执行时线程会被阻塞，从而导致效率下降。在重叠执行时，即使操作还未完成，调用的函数也会立即返回。费时的 I/O 操作在后台进行，这样线程就可以干别的事情。例如，线程可以在不同的句柄上同时执行 I/O 操作，甚至可以在同一句柄上同时进行读写操作。“重叠”一词的含义就在于此。

ReadFile 函数只要在串行口输入缓冲区中读入指定数量的字符，就算完成操作。而 WriteFile 函数不但要把指定数量的字符拷入到输出缓冲中，而且要等这些字符从串行口送出去后才算完成操作。

ReadFile 和 WriteFile 函数是否为执行重叠操作是由 CreateFile 函数决定的。如果在调用 CreateFile 创建句柄时指定了 FILE_FLAG_OVERLAPPED 标志，那么调用 ReadFile 和 WriteFile 对该句柄进行的读写操作就是重叠的，如果未指定重叠标志，则读写操作是同步的。

函数 ReadFile 和 WriteFile 的参数和返回值很相似。这里仅列出 ReadFile 函数的声明：

```
BOOL ReadFile(
HANDLE hFile, // 文件句柄
LPVOID lpBuffer, // 读缓冲区
DWORD nNumberOfBytesToRead, // 要求读入的字节数
LPDWORD lpNumberOfBytesRead, // 实际读入的字节数
LPOVERLAPPED lpOverlapped // 指向一个 OVERLAPPED 结构
); //若返回 TRUE 则表明操作成功
```

需要注意的是如果该函数因为超时而返回，那么返回值是 TRUE。参数 lpOverlapped 在重叠操作时应该指向一个 OVERLAPPED 结构，如果该参数为 NULL，那么函数将进行同步操作，而不管句柄是否是由 FILE_FLAG_OVERLAPPED 标志建立的。

当 ReadFile 和 WriteFile 返回 FALSE 时，不一定就是操作失败，线程应该调用 GetLastError 函数分析返回的结果。例如，在重叠操作时如果操作还未完成函数就返回，那么函数就返回 FALSE，而且 GetLastError 函数返回 ERROR_IO_PENDING。

在使用重叠 I/O 时，线程需要创建 OVERLAPPED 结构以供读写函数使用。OVERLAPPED 结构最重要的成员是 hEvent，hEvent 是一个事件对象句柄，线程应该用 CreateEvent 函数为 hEvent 成员创建一个手工重置事件，hEvent 成员将作为线程的同步对象使用。如果读写函数未完成操作就返回，就那么把 hEvent 成员设置成无信号的。操作完成后(包括超时)，hEvent 会变成有信号的。

如果 GetLastError 函数返回 ERROR_IO_PENDING，则说明重叠操

作还为完成，线程可以等待操作完成。有两种等待办法：一种办法是用象 WaitForSingleObject 这样的等待函数来等待 OVERLAPPED 结构的 hEvent 成员，可以规定等待的时间，在等待函数返回后，调用 GetOverlappedResult。另一种办法是调用 GetOverlappedResult 函数等待，如果指定该函数的 bWait 参数为 TRUE，那么该函数将等待 OVERLAPPED 结构的 hEvent 事件。GetOverlappedResult 可以返回一个 OVERLAPPED 结构来报告包括实际传输字节在内的重叠操作结果。

如果规定了读/写操作的超时，那么当超过规定时间后，hEvent 成员会变成有信号的。因此，在超时发生后，WaitForSingleObject 和 GetOverlappedResult 都会结束等待。WaitForSingleObject 的 dwMilliseconds 参数会规定一个等待超时，该函数实际等待的时间是两个超时的最小值。注意 GetOverlappedResult 不能设置等待的时限，因此如果 hEvent 成员无信号，则该函数将一直等待下去。

在调用 ReadFile 和 WriteFile 之前，线程应该调用 ClearCommError 函数清除错误标志。该函数负责报告指定的错误和设备的当前状态。

调用 PurgeComm 函数可以终止正在进行的读写操作，该函数还会清除输入或输出缓冲区中的内容。

12.3.4 通信事件

在 Windows 95/NT 中，WM_COMMNOTIFY 消息已经取消，在串行口产生一个通信事件时，程序并不会收到通知消息。线程需要调用 WaitCommEvent 函数来监视发生在串行口中的各种事件，该函数的第二个参数返回一个事件屏蔽变量，用来指示事件的类型。线程可以用 SetCommMask 建立事件屏蔽以指定要监视的事件，表 12.4 列出了可以监视的事件。调用 GetCommMask 可以查询串行口当前的事件屏蔽。

表 12.4 通信事件

事件屏蔽	含义
EV_BREAK	检测到一个输入中断
EV_CTS	CTS 信号发生变化
EV_DSR	DSR 信号发生变化
EV_ERR	发生行状态错误
EV_RING	检测到振铃信号
EV_RLSD	RLSD(CD)信号发生变化
EV_RXCHAR	输入缓冲区接收到新字符
EV_RXFLAG	输入缓冲区收到事件字符
EV_TXEMPTY	发送缓冲区为空

WaitCommEvent 即可以同步使用，也可以重叠使用。如果串口是用 FILE_FLAG_OVERLAPPED 标志打开的，那么 WaitCommEvent 就进行重叠操作，此时该函数需要一个 OVERLAPPED 结构。线程可以调用等待函数或 GetOverlappedResult 函数来等待重叠操作的完成。

当指定范围内的某一事件发生后，线程就结束等待并把该事件的屏蔽码设置到事件屏蔽变量中。需要注意的是，WaitCommEvent 只检测调用该函数后发生的事件。例如，如果在调用 WaitCommEvent 前在输入缓

冲区中就有字符，则不会因为这些字符而产生 EV_RXCHAR 事件。

如果检测到输入的硬件信号(如 CTS、RTS 和 CD 信号等)发生了变化，线程可以调用 GetCommMaskStatus 函数来查询它们的状态。而用 EscapeCommFunction 函数可以控制输出的硬件信号(如 DTR 和 RTS 信号)。

12.4 一个通信演示程序

为了使读者更好地掌握本章的概念，这里举一个具体实例来说明问题。如图 12.1 所示，例子程序名为 Terminal，是一个简单的 TTY 终端仿真程序。读者可以用该程序打开一个串行口，该程序会把用户的键盘输入发送给串行口，并把从串口接收到的字符显示在视图中。用户通过选择 File->Connect 命令来打开串行口，选择 File->Disconnect 命令则关闭串行口。

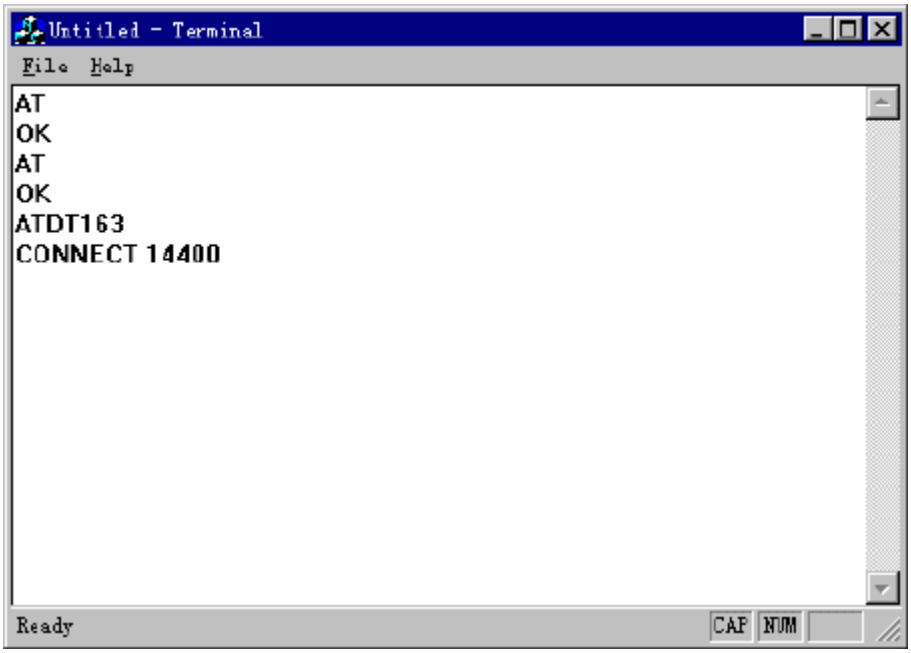


图 12.1 Terminal 终端仿真程序

当用户选择 File->Settings...命令时，会弹出一个 Communication settings 对话框，如图 12.2 所示。该对话框主要用来设置串行口，包括端口、波特率、每字节位数、校验、停止位数和流控制。

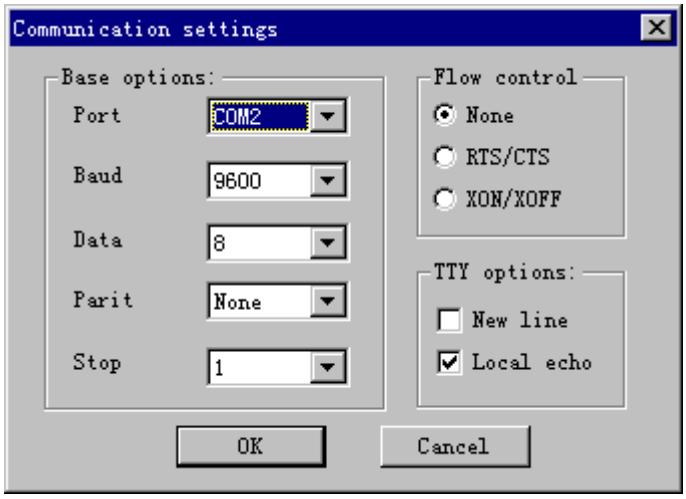


图 12.2 Communication settings 对话框

通过该对话框也可以设置 TTY 终端仿真的属性，如果选择 New Line(自动换行)，那么每当从串口读到回车符(‘\r’)时，视图中的正文

就会换行，否则，只有在读到换行符(‘ \n ’)时才会换行。如果选择 Local echo(本地回显)，那么发送的字符会在视图中显示出来。

终端仿真程序的特点是数据的传输没有规律。因为键盘输入速度有限，所以发送的数据量较小，但接收的数据源是不确定的，所以有可能会有大量数据高速涌入的情况发生。根据 Terminal 的这些特性，我们在程序中创建了一个辅助工作者线程专门来监视串行口的输入。由于写入串行口的数据量不大，不会太费时，所以在主线程中完成写端口的任务是可以的，不必另外创建线程。

现在就让我们开始工作。请读者按下面几步进行：

用 AppWizard 建立一个名为 Terminal 的 MFC 应用程序。在 MFC AppWizard 对话框的第 1 步选择 Single document，在第 4 步去掉 Docking toolbar 的选择，在第 6 步把 CTerminalView 的基类改为 CEditView。

在 Terminal 工程的资源视图中打开 IDR_MAINFRAME 菜单资源。去掉 Edit 菜单和 View 菜单，并去掉 File 菜单中除 Exit 以外的所有菜单项。然后在 File 菜单中加入三个菜单项，如表 12.5 所示。

表 12.5 新菜单项

标题	ID
Settings...	ID_FILE_SETTINGS
Connect	ID_FILE_CONNECT
Disconnect	ID_FILE_DISCONNECT

用 ClassWizard 为 CTerminalDoc 类创建三个与上表菜单消息对应的命令处理函数，使用缺省的函数名。为 ID_FILE_CONNECT 和 ID_FILE_DISCONNECT 命令创建命令更新处理函数。另外，用 ClassWizard 为该类加入 CanCloseFrame 成员函数。

用 ClassWizard 为 CTerminalView 类创建 OnChar 函数，该函数用来把用户键入的字符向串行口输出。

新建一个对话框模板资源，令其 ID 为 IDD_COMSETTINGS。请按图 12.2 和表 12.6 设计对话框模板。

表 12.6 通信设置对话框中的主要控件

控件	ID	属性设置
Base options 组框	缺省	标题为 Base options
Port 组合框	IDC_PORT	Drop List，不选 Sort，初始列表为 COM1、COM2、COM3、COM4
Baud rate 组合框	IDC_BAUD	Drop List，不选 Sort，初始列表为 300、600、1200、2400、9600、14400、19200、38400、57600
Data bits 组合框	IDC_DATABITS	Drop List，不选 Sort，初列表为 5、6、7、8
Parity 组合框	IDC_PARITY	Drop List，不选 Sort，初列表为 None、Even、Odd
Stop bits 组合框	IDC_STOPBITS	Drop List，不选 Sort，初列表为 1、1.5、2

Flow control 组框	缺省	标题为 Flow control
None 单选按钮	IDC_FLOWCTRL	标题为 None，选择 Group 属性
RTS/CTS 单选按钮	缺省	标题为 RTS/CTS
XON/XOFF 单选按钮	缺省	标题为 XON/XOFF
TTY options 组框	缺省	标题为 TTY options
New line 检查框	IDC_NEWLINE	标题为 New line
Local echo 检查框	IDC_ECHO	标题为 Local echo

打开 ClassWizard，为 IDD_COMSETTINGS 模板创建一个名为 CSetupDlg 的对话框类。为该类加入 OnInitDialog 成员函数，并按表 12.7 加入数据成员。

表 12.7 CSetupDlg 类的数据成员

控件 ID	变量名	数据类型
IDC_BAND	m_sBaud	CString
IDC_DATABITS	m_sDataBits	CString
IDC_ECHO	m_bEcho	BOOL
IDC_FLOWCTRL	m_nFlowCtrl	int
IDC_NEWLINE	m_bNewLine	BOOL
IDC_PARITY	m_nParity	int
IDC_PORT	m_sPort	CString
IDC_STOPBITS	m_nStopBits	int

按清单 12.6、12.7 和 12.8 修改程序。清单 12.6 列出了 CTerminalDoc 类的部分代码，清单 12.7 是 CTerminalView 的部分代码，清单 12.8 是 CSetupDlg 类的部分代码。在本例中使用了 WM_COMMNOTIFY 消息。虽然在 Win32 中，WM_COMMNOTIFY 消息已经取消，系统自己不会产生该消息，但 Visual C++对该消息的定义依然保留。考虑到使用习惯，Terminal 程序辅助线程通过发送该消息来通知视图有通信事件发生。

清单 12.6 CTerminalDoc 类的部分代码

```
// TerminalDoc.h : interface of the CTerminalDoc class
//
/////////////////////////////////////////////////////////////////////////////
#define MAXBLOCK 2048
#define XON 0x11
#define XOFF 0x13
UINT CommProc(LPVOID pParam);
class CTerminalDoc : public CDocument
{
protected: // create from serialization only
```

```
CTerminalDoc();
DECLARE_DYNCREATE(CTerminalDoc)
// Attributes
public:
CWinThread* m_pThread; // 代表辅助线程
volatile BOOL m_bConnected;
volatile HWND m_hTermWnd;
volatile HANDLE m_hPostMsgEvent; // 用于WM_COMMNOTIFY消息的事件对象
OVERLAPPED m_osRead, m_osWrite; // 用于重叠读/写
volatile HANDLE m_hCom; // 串行口句柄
int m_nBaud;
int m_nDataBits;
BOOL m_bEcho;

int m_nFlowCtrl;
BOOL m_bNewLine;
int m_nParity;
CString m_sPort;
int m_nStopBits;
// Operations
public:
BOOL ConfigConnection();
BOOL OpenConnection();
void CloseConnection();
DWORD ReadComm(char *buf,DWORD dwLength);
DWORD WriteComm(char *buf,DWORD dwLength);
// Overrides
. . .
};
////////////////////////////////////////////////////////////////////////////
// TerminalDoc.cpp : implementation of the CTerminalDoc class
//
#include “SetupDlg.h”
CTerminalDoc::CTerminalDoc()
{
// TODO: add one-time construction code here
m_bConnected=FALSE;
m_pThread=NULL;
m_nBaud = 9600;
m_nDataBits = 8;
m_bEcho = FALSE;
```

```
m_nFlowCtrl = 0;
m_bNewLine = FALSE;
m_nParity = 0;
m_sPort = “COM2”;
m_nStopBits = 0;
}
CTerminalDoc::~CTerminalDoc()
{
if(m_bConnected)
CloseConnection();
// 删除事件句柄
if(m_hPostMsgEvent)
CloseHandle(m_hPostMsgEvent);
if(m_osRead.hEvent)
CloseHandle(m_osRead.hEvent);
if(m_osWrite.hEvent)
CloseHandle(m_osWrite.hEvent);
}
BOOL CTerminalDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
((CEditView*)m_viewList.GetHead())->SetWindowText(NULL);
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
// 为 WM_COMMNOTIFY 消息创建事件对象，手工重置，初始化为有信号的
if((m_hPostMsgEvent=CreateEvent(NULL, TRUE, TRUE, NULL))==NULL)
return FALSE;
memset(&m_osRead, 0, sizeof(OVERLAPPED));
memset(&m_osWrite, 0, sizeof(OVERLAPPED));
// 为重叠读创建事件对象，手工重置，初始化为无信号的
if((m_osRead.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL))==NULL)
return FALSE;
// 为重叠写创建事件对象，手工重置，初始化为无信号的
if((m_osWrite.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL))==NULL)
return FALSE;
return TRUE;
}
```

```
void CTerminalDoc::OnFileConnect()
{
// TODO: Add your command handler code here
if(!OpenConnection())
AfxMessageBox("Can't open connection");
}
void CTerminalDoc::OnFileDisconnect()
{
// TODO: Add your command handler code here
CloseConnection();
}
void CTerminalDoc::OnUpdateFileConnect(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(!m_bConnected);
}
void CTerminalDoc::OnUpdateFileDisconnect(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here
pCmdUI->Enable(m_bConnected);
}
// 打开并配置串行口，建立工作者线程
BOOL CTerminalDoc::OpenConnection()
{
COMMTIMEOUTS TimeOuts;
POSITION firstViewPos;
CView *pView;
firstViewPos=GetFirstViewPosition();
pView=GetNextView(firstViewPos);
m_hTermWnd=pView->GetSafeHwnd();
if(m_bConnected)
return FALSE;
m_hCom=CreateFile(m_sPort, GENERIC_READ | GENERIC_WRITE,
0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL |
FILE_FLAG_OVERLAPPED,
NULL); // 重叠方式
if(m_hCom==INVALID_HANDLE_VALUE)
return FALSE;
SetupComm(m_hCom,MAXBLOCK,MAXBLOCK);
SetCommMask(m_hCom, EV_RXCHAR);
// 把间隔超时设为最大，把总超时设为 0 将导致 ReadFile 立即返回
```

并完成操作

```
TimeOuts.ReadIntervalTimeout=MAXDWORD;
TimeOuts.ReadTotalTimeoutMultiplier=0;
TimeOuts.ReadTotalTimeoutConstant=0;
/* 设置写超时以指定 WriteComm 成员函数中的
GetOverlappedResult 函数的等待时间*/
TimeOuts.WriteTotalTimeoutMultiplier=50;
TimeOuts.WriteTotalTimeoutConstant=2000;
SetCommTimeouts(m_hCom, &TimeOuts);
if(ConfigConnection())
{
m_pThread=AfxBeginThread(CommProc, this,
THREAD_PRIORITY_NORMAL,
0, CREATE_SUSPENDED, NULL); // 创建并挂起线程
if(m_pThread==NULL)
{
CloseHandle(m_hCom);
return FALSE;
}
else
{
m_bConnected=TRUE;
m_pThread->ResumeThread(); // 恢复线程运行
}
}
else
{
CloseHandle(m_hCom);
return FALSE;
}
return TRUE;
}
// 结束工作者线程，关闭串行口
void CTerminalDoc::CloseConnection()
{
if(!m_bConnected) return;
m_bConnected=FALSE;
//结束 CommProc 线程中 WaitSingleObject 函数的等待
SetEvent(m_hPostMsgEvent);
//结束 CommProc 线程中 WaitCommEvent 的等待
SetCommMask(m_hCom, 0);
//等待辅助线程终止
```

```
WaitForSingleObject(m_pThread->m_hThread, INFINITE);
m_pThread=NULL;
CloseHandle(m_hCom);
}
// 让用户设置串行口
void CTerminalDoc::OnFileSettings()
{
// TODO: Add your command handler code here
CSetupDlg dlg;
CString str;
dlg.m_bConnected=m_bConnected;
dlg.m_sPort=m_sPort;
str.Format("%d",m_nBaud);
dlg.m_sBaud=str;
str.Format("%d",m_nDataBits);
dlg.m_sDataBits=str;
dlg.m_nParity=m_nParity;
dlg.m_nStopBits=m_nStopBits;
dlg.m_nFlowCtrl=m_nFlowCtrl;
dlg.m_bEcho=m_bEcho;
dlg.m_bNewLine=m_bNewLine;
if(dlg.DoModal()==IDOK)
{
m_sPort=dlg.m_sPort;
m_nBaud=atoi(dlg.m_sBaud);
m_nDataBits=atoi(dlg.m_sDataBits);
m_nParity=dlg.m_nParity;
m_nStopBits=dlg.m_nStopBits;
m_nFlowCtrl=dlg.m_nFlowCtrl;
m_bEcho=dlg.m_bEcho;
m_bNewLine=dlg.m_bNewLine;
if(m_bConnected)
if(!ConfigConnection())
AfxMessageBox("Can't realize the settings!");
}
}
// 配置串行口
BOOL CTerminalDoc::ConfigConnection()
{
DCB dcb;
if(!GetCommState(m_hCom, &dcb))
return FALSE;
```

```
dcb.fBinary=TRUE;
dcb.BaudRate=m_nBaud; // 波特率
dcb.ByteSize=m_nDataBits; // 每字节位数
dcb.fParity=TRUE;
switch(m_nParity) // 校验设置
{
case 0: dcb.Parity=NOPARITY;
break;
case 1: dcb.Parity=EVENPARITY;
break;
case 2: dcb.Parity=ODDPARITY;
break;
default:;
}
switch(m_nStopBits) // 停止位
{
case 0: dcb.StopBits=ONESTOPBIT;
break;
case 1: dcb.StopBits=ONE5STOPBITS;
break;
case 2: dcb.StopBits=TWOSTOPBITS;
break;
default:;
}
// 硬件流控制设置
dcb.fOutxCtsFlow=m_nFlowCtrl==1;
dcb.fRtsControl=m_nFlowCtrl==1”
RTS_CONTROL_HANDSHAKE:RTS_CONTROL_ENABLE;
// XON/XOFF 流控制设置
dcb.fInX=dcb.fOutX=m_nFlowCtrl==2;
dcb.XonChar=XON;
dcb.XoffChar=XOFF;
dcb.XonLim=50;
dcb.XoffLim=50;
return SetCommState(m_hCom, &dcb);
}
// 从串行口输入缓冲区中读入指定数量的字符
DWORD CTerminalDoc::ReadComm(char *buf,DWORD dwLength)
{
DWORD length=0;
COMSTAT ComStat;
DWORD dwErrorFlags;
```

```
ClearCommError(m_hCom,&dwErrorFlags,&ComStat);
length=min(dwLength, ComStat.cbInQue);
ReadFile(m_hCom,buf,length,&length,&m_osRead);
return length;
}
// 将指定数量的字符从串行口输出
DWORD CTerminalDoc::WriteComm(char *buf,DWORD dwLength)
{
BOOL fState;
DWORD length=dwLength;
COMSTAT ComStat;
DWORD dwErrorFlags;
ClearCommError(m_hCom,&dwErrorFlags,&ComStat);
fState=WriteFile(m_hCom,buf,length,&length,&m_osWrite);
if(!fState){
if(GetLastError()==ERROR_IO_PENDING)
{
GetOverlappedResult(m_hCom,&m_osWrite,&length,TRUE);// 等待
}
else
length=0;
}
return length;
}
// 工作者线程，负责监视串行口
UINT CommProc(LPVOID pParam)
{
OVERLAPPED os;
DWORD dwMask, dwTrans;
COMSTAT ComStat;
DWORD dwErrorFlags;
CTerminalDoc *pDoc=(CTerminalDoc*)pParam;
memset(&os, 0, sizeof(OVERLAPPED));
os.hEvent=CreateEvent(NULL, TRUE, FALSE, NULL);
if(os.hEvent==NULL)
{
AfxMessageBox("Can't create event object!");
return (UINT)-1;
}
while(pDoc->m_bConnected)
{
ClearCommError(pDoc->m_hCom,&dwErrorFlags,&ComStat);
```

```
if(ComStat.cbInQue)
{
// 无限等待 WM_COMMNOTIFY 消息被处理完
WaitForSingleObject(pDoc->m_hPostMsgEvent, INFINITE);
ResetEvent(pDoc->m_hPostMsgEvent);
// 通知视图
PostMessage(pDoc->m_hTermWnd, WM_COMMNOTIFY,
EV_RXCHAR, 0);
continue;
}
dwMask=0;
if(!WaitCommEvent(pDoc->m_hCom, &dwMask, &os)) // 重叠操作
{
if(GetLastError()==ERROR_IO_PENDING)
// 无限等待重叠操作结果
GetOverlappedResult(pDoc->m_hCom, &os, &dwTrans, TRUE);
else
{
CloseHandle(os.hEvent);
return (UINT)-1;
}
}
}
CloseHandle(os.hEvent);
return 0;
}
BOOL CTerminalDoc::CanCloseFrame(CFrameWnd* pFrame)
{
// TODO: Add your specialized code here and/or call the base class
SetModifiedFlag(FALSE); // 将文档的修改标志设置成未修改
return CDocument::CanCloseFrame(pFrame);
}
```

毫无疑问，CTerminalDoc **类是研究重点。该类负责** Terminal **的通信任务，主要包括设置通信参数、打开和关闭串行口、建立和终止辅助工作线程、用辅助线程监视串行口等等。**

在 CTerminalDoc **类的头文件中，有些变量是用** volatile **关键字声明的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用** volatile **声明，该关键字的作用是防止优化编译器把变量从内存装入** CPU **寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。**

成员 m_bConnected **用来表明当前是否存在一个通信连接。** m_hTermWnd **用来保存是视图的窗口句柄。** m_hPostMsgEvent **事件对象**

用于 WM_COMMNOTIFY 消息的允许和禁止。m_pThread 用来指向 AfxBeginThread 创建的 CWinThread 对象，以便对线程进行控制。OVERLAPPED 结构 m_osRead 和 m_osWrite 用于串行口的重叠读/写，程序应该为它们的 hEvent 成员创建事件句柄。

CTerminalDoc 类的构造函数主要完成一些通信参数的初始化工作。OnNewDocument 成员函数创建了三个事件对象，CTerminalDoc 的析构函数关闭串行口并删除事件对象句柄。

OnFileSettings 是 File->Settings...的命令处理函数，该函数弹出一个 CSetupDlg 对话框来设置通信参数。实际的设置工作由 ConfigConnection 函数完成，在 OpenConnection 和 OnFileSettings 中都会调用该函数。

OpenConnection 负责打开串行口并建立辅助工作线程，当用户选择了 File->Connect 命令时，消息处理函数 OnFileConnect 将调用该函数。该函数调用 CreateFile 以重叠方式打开指定的串行口并把返回的句柄保存在 m_hCom 成员中。接着，函数对 m_hCom 通信设备进行各种设置。需要注意的是对超时的设定，将读间隔超时设置为 MAXDWORD 并使其它读超时参数为 0 会导致 ReadFile 函数立即完成操作并返回，而不管读入了多少字符。设置超时就规定了 GetOverlappedResult 函数的等待时间，因此有必要将写超时设置成适当的值，这样如果不能完成写串口的任务，GetOverlappedResult 函数会在超过规定超时后结束等待并报告实际传输的字符数。

如果对 m_hCom 设置成功，则函数会建立一个辅助线程并暂时将其挂起。在最后，调用 CWinThread:: ResumeThread 使线程开始运行。

OpenConnection 调用成功后，线程函数 CommProc 就开始工作。该函数的主体是一个 while 循环，在该循环内，混合了两种方法监视串行口输入的方法。先是调用 ClearCommError 函数查询输入缓冲区中是否有字符，如果有，就向视图发送 WM_COMMNOTIFY 消息通知其接收字符。如果没有，则调用 WaitCommEvent 函数监视 EV_RXCHAR 通信事件，该函数执行重叠操作，紧接着调用的 GetOverlappedResult 函数无限等待通信事件，如果 EV_RXCHAR 事件发生(串口收到字符并放入输入缓冲区中)，那么函数就结束等待。

上述两种方法的混合使用兼顾了线程的效率和可靠性。如果只用 ClearCommError 函数，则辅助线程将不断耗费 CPU 时间来查询，效率较低。如果只用 WaitCommEvent 来监视，那么由于该函数对输入缓冲区中已有的字符不会产生 EV_RXCHAR 事件，因此在通信速率较高时，会造成数据的延误和丢失。

注意到辅助线程用 m_PostMsgEvent 事件对象来同步 WM_COMMNOTIFY 消息的发送。在发送消息之前，WaitForSingleObject 函数无限等待 m_PostMsgEvent 对象，WM_COMMNOTIFY 的消息处理函数 CTerminalView::OnCommNotify 在返回时会把该对象置为有信号，因此，如果 WaitForSingleObject 函数返回，则说明上一个 WM_COMMNOTIFY 消息已被处理完，这时才能发下一个消息，在发消息前还要调用 ResetEvent 把 m_PostMsgEvent 对象置为无信号的，以供下

次使用。

由于 PostMessage 函数在消息队列中放入消息后会立即返回，所以如果不采取上述措施，那么辅助线程可能在主线程未处理之前重复发出 WM_COMMNOTIFY 消息，这会降低系统的效率。

可能有读者会问，为什么不用 SendMessage"该函数在发送的消息被处理完毕后才返回，这样不就不用考虑同步问题了吗"是的，本例中也可以使用 SendMessage，但该函数会阻塞辅助线程的执行直到消息处理完毕，这会降低效率。如果用 PostMessage，那么在函数立即返回后线程还可以干别的事情，因此，考虑到效率问题，这里使用了 PostMessage 而不是 SendMessage。

函数 ReadComm 和 WriteComm 分别用来从 m_hCom 通信设备中读/写指定数量的字符。ReadComm 函数很简单，由于对读超时的特殊设定，ReadFile 函数会立即返回并完成操作，并在 length 变量中报告实际读入的字符数。此时，没有必要调用等待函数或 GetOverlappedResult。在 WriteComm 中，调用 GerOverlappedResult 来等待操作结果，直到超时发生。不管是否超时，该函数在结束等待后都会报告实际的传输字符数。

CloseConnection 函数的主要任务是终止辅助线程并关闭 m_hCom 通信设备。为了终止线程，该函数设置了一系列信号，以结束辅助线程中的等待和循环，然后调用 WaitForSingleObject 等待线程结束。

清单 12.7 CTerminalView 类的部分代码

```
// TerminalView.h : interface of the CTerminalView class
/////////////////////////////////////////////////////////////////////////////
class CTerminalView : public CEditView
{
. . .
afx_msg LRESULT OnCommNotify(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
};
// TerminalView.cpp : implementation of the CTerminalView class
//
BEGIN_MESSAGE_MAP(CTerminalView, CEditView)
. . .
ON_MESSAGE(WM_COMMNOTIFY, OnCommNotify)
END_MESSAGE_MAP()
LRESULT CTerminalView::OnCommNotify(WPARAM wParam, LPARAM lParam)
{
char buf[MAXBLOCK/4];
CString str;
int nLength, nTextLength;
CTerminalDoc* pDoc=GetDocument();
```

```
CEdit& edit=GetEditCtrl();
if(!pDoc->m_bConnected ||
(wParam & EV_RXCHAR)!=EV_RXCHAR) // 是否是 EV_RXCHAR 事件”
{
SetEvent(pDoc->m_hPostMsgEvent);  // 允许发送下一个 WM_COMMNOTIFY 消息
return 0L;
}
nLength=pDoc->ReadComm(buf,100);
if(nLength)
{
nTextLength=edit.GetWindowTextLength();
edit.SetSel(nTextLength,nTextLength); //移动插入光标到正文末尾
for(int i=0;i<nLength;i++)
{
switch(buf[i])
{
case '\r': // 回车
if(!pDoc->m_bNewLine)
break;
case '\n': // 换行
str+=“\r\n”;
break;
case '\b': // 退格
edit.SetSel(-1, 0);
edit.ReplaceSel(str);
nTextLength=edit.GetWindowTextLength();
edit.SetSel(nTextLength-1,nTextLength);
edit.ReplaceSel(““); //回退一个字符
str=““;
break;
case '\a': // 振铃
MessageBeep((UINT)-1);
break;
default :
str+=buf[i];
}
}
edit.SetSel(-1, 0);
edit.ReplaceSel(str); // 向编辑视图中插入收到的字符
}
```

SetEvent(pDoc->m_hPostMsgEvent); // 允许发送下一个 WM_COMMNOTIFY 消息

return 0L;

}

void CTerminalView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)

{

// TODO: Add your message handler code here and/or call default

CTerminalDoc* pDoc=GetDocument();

char c=(char)nChar;

if(!pDoc->m_bConnected)return;

pDoc->WriteComm(&c, 1);

if(pDoc->m_bEcho)

CEditView::OnChar(nChar, nRepCnt, nFlags); // 本地回显

}

CTerminalView 是 CEditView 的派生类，利用 CEditView 的编辑功能，可以大大简化程序的设计。

OnChar 函数对 WM_CHAR 消息进行处理，它调用 CTerminalDoc::WriteComm 把用户键入的字符从串行口输出。如果设置了 Local echo，那么就调用 CEditView::OnChar 把字符输出到视图中。

OnCommNotify 是 WM_COMMNOTIFY 消息的处理函数。该函数调用 CTerminalDoc::ReadComm 从串行口输入缓冲区中读入字符并把它们输出到编辑视图中。在输出前，函数会对一些特殊字符进行处理。如果读者对控制编辑视图的代码不太明白，那么请参见 6.1.4。在函数返回时，要调用 SetEvent 把 m_hPostMsgEvent 置为有信号。

清单 12.8 CSetupDlg 类的部分代码

```
// SetupDlg.h : header file
//
class CSetupDlg : public CDialog
{
. . .
public:
BOOL m_bConnected;
. . .
};
// SetupDlg.cpp : implementation file
//
BOOL CSetupDlg::OnInitDialog()
{
CDialog::OnInitDialog();
// TODO: Add extra initialization here
GetDlgItem(IDC_PORT)->EnableWindow(!m_bConnected);
```

```
return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE
}
```

CSetupDlg **的主要任务是配置通信参数。在** OnInitDialog **函数中，要根据当前是否连接来允许/禁止** Port **组合框。因为在打开一个连接后，显然不能随便改变端口。**

小结

本章重点介绍了 Win 32 环境下的多线程和串行通信编程。本章的要点如下：

Windows 3.x 实行的是协同式多任务，应用程序必须“自觉”地放弃 CPU 控制权，否则系统会被挂起。

Windows 95/NT 实现了抢先式多任务，应用程序对 CPU 的控制时间由系统分配，系统可以在任何时候中断应用程序，并把控制权转交给别的程序。

在 Win 32 环境下，每个进程可以同时执行多个线程。线程是系统分配 CPU 时间片的基本实体，系统在所有线程之间快速切换以实现多任务。

由于同一进程的所有线程共享进程的虚拟地址空间、Windows 95 的重入问题、MFC 在对象级的线程不安全性以及线程之间的协调等原因，多个线程必须同步执行。同步机制是由同步对象和等待函数共同实现的。同步对象主要包括事件、mutex 和信号灯，进程和线程句柄、文件和通信设备也可以用作同步对象。

在 Win 32 中，传统的 OpenComm、ReadComm、WriteComm、CloseComm 等串行通信函数已经过时，WM_COMMNOTIFY 消息也消失了。程序应该调用 CreateFile 打开一个串行通信设备，用 ReadFile 和 WriteFile 来进行 I/O 操作，用 WaitCommEvent 来监视通信事件。ReadFile、WriteFile 和 WaitCommEvent 既可以同步操作，也可以重叠操作。

利用 Win 32 的重叠 I/O 操作和多线程特性，程序员可以编写出高效的通信程序。