# 第二部分　用户界面实例

本部分的例子集中讨论应用程序的用户界面，可以在 Developer Studio（开发环境）、MFC和Visual C++的帮助下创建它们。如果更希望用用户界面开发工具开发它们，本书的绝大多数例子涉及到这方面，并根据创建一个 MFC应用程序时应该考虑的顺序安排讨论主题。例 1概括了创建一个MFC应用程序的全局规划，并引用后面的例子进行说明。本部分中讨论的主题包括以下内容：

## 应用程序与环境

这部分的例子包括用 MFC规划应用程序的执行，既用应用程序向导，也用强制手段；包括应用程序与环境交互的大部分公共问题，如初始化屏幕、显示图标、处理命令行选项及保存优先选项。

## 菜单

下一个关注的内容是应用程序的菜单，即添加命令、更新状态、尝试修改外观。也包括了怎样用Class Wizard(类向导)在应用程序类添加菜单命令。

## 工具栏和状态栏

这部分的例子讨论用Developer Studio的编辑器创建工具栏和状态栏。例子包括更新两种类型控制条的控件，以反映应用程序状态变化，以及给任何一种控制条添加非标准控件。

## 视图

如果选择创建一个单文档界面或多文档界面应用程序，则应用程序的视图将是用户与应用程序交互的主要模式。创建的应用程序类型决定了要创建的视图的类型。视图的其他方面包括分割视图和有条件改变鼠标形状等。

## 对话框和对话条

对话框和对话条是与应用程序进行交互的第二种模式，它们可以是有模式或无模式的，可以全部由你自己建立，也可以定制一个系统提供的对话框。

## 控件窗口

按钮和编辑框一般出现在对话框中，它们通常叫做控件窗口(由操作系统提供的子窗口)。不仅可以把它们放进对话框，而且还可以把它们放进视图、对话条或任何有窗口的地方。

### 绘图

这部分的例子包括从绘图和文本到操纵位图。

# 第5章 应用程序与环境

欢迎进入应用程序与环境。本章将规划一个 MFC应用程序，确定它是一个对话框、单文档界面还是多文档界面，并把它从所有其他由Developer Studio产生的MFC应用程序中区别开。

**例1 规划MFC应用程序** 设计一个使用Developer Studio等的策略，即把应用程序的设计想法变成一个实际的应用程序。

**例2 用AppWizard创建一个MFC应用程序** 使用AppWizard产生一组类和资源，它们是MFC应用程序的基础。

**例3 用ClassWizard创建一个类** 用类向导把类添加到应用程序。

**例4 初始化应用程序屏幕** 控制应用程序窗口的大小和布置。

**例5 保存应用程序屏幕** 保存应用程序窗口的大小和位置，以供下次执行时使用。

**例6 处理命令行选项** 把命令行标志转换为应用程序中可以使用的布尔变量。

**例7 动态改变应用程序图标** 改变应用程序的图标，它不仅可以显示在应用程序的左上角，还可以显示在系统的任务栏里。

**例8 提示用户优先选项** 提示用户有关应用程序的选项。

**例9 保存和恢复用户的优先选项** 把应用程序选项保存到注册表里。

**例10 终止应用程序** 讨论一种退出的应用程序方法。

**例11 创建一个启动窗口** 我们将创建初始屏幕，显示应用程序的名字和公司。

## 5.1 例1 规划MFC应用程序

### 目标

使用Visual C++、MFC库和Developer Studio的向导和编辑器创建一个应用程序。

### 策略

我们将从确定最符合需要的 MFC应用程序类型开始，即选择对话框、单文档界面或多文档界面或者都不是；接着选择应用程序的最佳的视图和文档；然后讨论用户与 MFC应用程序进行交互的其他方法，以及可以用来添加这些交互界面的Developer Studio编辑器。如果应用程序中的功能可以与其他应用程序共享，则讨论库的选择；最后，因为Developer Studio创建的每个应用程序看起来几乎都相似，因此，将开发一些可以添加到应用程序并使它醒目的特征。

### 步骤

1. 选择应用程序类型

1) 使用Developer Studio的AppWizard可以自动创建三种类型的MFC应用程序(见例2)：对话框、单文档界面或多文档界面；也可以手工创建任何类型的复合应用程序。有关确定选择应用程序类型的详细讨论，参见第2章。为快速作出选择，请尝试使用下面的指南：

- 如果创建一个用户界面需求有限的应用程序，或如果想界面完全单一，那么就创建一个对话框应用程序。典型的对话框应用程序包括配置硬件设备的应用程序、屏幕保护程序和游戏程序等。
- 如果应用程序要编辑一个文档，应该选择另外两种应用程序类型。这里的“编辑一个文档”是广义上的意思，所指的文档可以是一个文本文件、电子数据表文件、第三方数据库的一个或多个表、或者是自己的二进制文件，甚至可以是大量硬件设备的储存设置。编辑仅仅表示对其中任何一个类型的文档进行添加、删除或修改操作。
- 单文档界面应用程序一次只允许处理一个文档。如果应用程序实际上一次只需处理一个文档，诸如监视一组硬件设备的应用程序，那么应该选择单文档界面；否则应该创建一个多文档界面应用程序，即使在开始时一次编辑多个文档并未显出有任何好处。
- 一个多文档界面应用程序允许一次编辑多个文档，它并不比一个单文档界面应用程序复杂，但却带来了一次至少查看多个文档的方便。

2) 如果决定要建立一个对话框应用程序，则该例子正好能处理它。一个对话框应用程序没有视图或文档，实际上界面上也没有更多的内容，然而，用户可以看看例子的剩余部分，以领会一些给应用程序添加色彩的方法。

2. 选择视图的类型

1) 如果选择一个单文档或多文档界面应用程序，在退出 AppWizard之前必须选择所需的视图类，在AppWizard的最后一步，可以选择下面视图类：

- 对于一个简单的文本编辑器应用程序，选择 CEditView。
- 对于一个能编辑多信息文本格式 (RTF)文件的应用程序，选择CRichEditView(这一选择将导致应用程序为文档类选择CRichEditDoc类)。
- 对于一个图形应用程序，选择CScrollView。
- 对于一个简单的监控或帐目管理应用程序，选择 CListView(例36)。
- 要着手创建一个资源管理器类型的应用程序，请选择 CTreeView(在以后的步骤中,可以手工添加一个CListView)。
- 在对话框模板外创建一个视图，选择 CFormView(一个对话框是一个被几个控件窗口占据的窗口，诸如按钮和编辑框)，有关该主题参见第1章。

2) 也可以在AppWizard的早期阶段为视图类间接地选择CRecordView或CDaoRecordView，同时在该阶段还得决定为应用程序添加什么样的数据库支持。如果在第二步中选择任何一项“DataBase View”选项，则该视图被添加，并允许轻松地访问一个 ODBC或DAO数据库中的记录。

3. 选择文档类型

应用程序可以与三种基本文档类型进行交互：平面文件 (flat file)、串行文件或数据库。通常，文档类型的选择取决于用户打算编写的应用程序类型，让我们讨论怎样选择：

- 一个平面二进制或文本文件是应用程序可以支持的最简单的文档，并且可以遵循你能构思的任何存储方法。要了解哪些MFC类支持平面文件，参见例63。

- 串行化文件代表MFC的有组织的存储二进制文件方式，以便于它们可以被容易地检索，甚至当它们是应用程序的早期版本时也可以被容易地检索；任何类型的文档，不管它是平面文件、电子数据表文件还是数据库数据都可以按照这种方式进行存储 (见例66到例70)。
- 应用程序也可以使用Microsoft Jet Engine DBMS(微软Jet引擎数据库管理系统)和任何其他第三方ODBC兼容的DBMS(见例72和73)。

4. 其他方面的考虑

1) 视图代表用户与文档进行交互的主要方法。然而，还有一些特征可以被添加到应用程序中，并允许用户与它们进行交互。

- 可以用菜单编辑器（Menu Editor）添加命令到主菜单。(参见例12、例13关于该主题的内容；有关弹出式菜单参见例21)
- 可以用工具栏编辑器（Toolbar Editor）添加工具栏和按钮(见例22)。
- 可以添加模式对话框，它允许用户在挂起应用程序时输入详细的信息 (见例40)。
- 也可以添加无模式对话框，调用它们时不必挂起应用程序 (见例41)。
- 可以添加对话条，它是工具栏和无模式对话框的综合，允许用户停放这种类型的对话框(见例45)。
- 也可以添加属性表，它是 Windows常用的，允许用户输入和保存它们的优先选项的方法(见例8)。

2) 当开始为应用程序添加一层又一层的功能时，可能想知道哪一个类应该包含哪一种功能，例如，哪一个类应该处理哪一种命令消息和窗口消息？可以轻松地从一个类访问另一个类中的数据和功能，这从附录D中可以看到。但是这些功能开始时应位于哪里呢？这里是些简略的指导，可帮助你作出决策。

- 应用程序类从CWApp派生，并不对任何窗口进行控制。除了控制应用程序的创建和卸载外，它自身应该有少数重要的附加功能，这些功能包括处理命令行标志和提供一种定制的打开文档的方法。应用程序类还提供一些应用程序范围的服务，诸如后台处理和超分类等。
- 主框架类从CFrameWnd类派生，控制应用程序的主窗口，负责所有应用程序范围的界面，包括工具栏、状态栏、菜单和对话条。然而，如果这些条中的任何一个有新增的功能的话，它应该被封装到它自己的类中。对用户优先选项的支持也通常可以在主框架类中发现。
- 文档类从CDocument派生，应该包括任何属于应用程序文档的数据。对于真实的 C++封装，不允许文档类对它的数据进行直接访问（甚至从视图类也不行），而应包含封装函数以访问它的数据。文档类还应包括装入和保存一个文档所必须的所有功能，这些文档包括从简单的二进制文件到ODBC数据库。如果应用程序不做任何其他事情而只访问一个ODBC数据库，则文档类可以只包含打开和关闭那个数据库所必须的逻辑，因为数据库是数据的主要仓储地，文档类对它自己来说是独立的。从一个存储设备中获得信息，并把它取出来交给视图，但是几乎不把信息存储到其他类中。
- 视图类从CView派生，应当包括查看和编辑文档类中数据所必须的所有逻辑。任何专门作用于文档的菜单或工具栏命令，诸如剪切或粘贴，应该在视图类中得到处理；所

有影响视图的鼠标消息应当在此处处理；所有绘图、报表、编辑、选择和打印应在此处使用；所有的对话框和弹出式菜单应在此处产生。如果这个类或任何一个类变得规模很大时，应该分解出任何公共的功能以形成一个新的基类。创建一个新的CMyBaseView类，并把一些基本功能放到该类中；或者可以把一些功能封装到它自己的类中。视图中选择、剪切和粘贴函数是它自己的类的一个很好代表。

- 其他类应尽可能多地封装属于它们自己的功能。对话框类应包括提示用户所必需的任何内容，对话条、工具栏和任务栏也应该这样。一个自画控件应当从它自己的类中绘制。
- 每当从应用程序中分解出公共功能，并把它们放到一个基类中时，可以把新的基类放进一个MFC扩展类中。然后就可以把该新的 MFC扩展类放进一个动态链接库中，应用程序便可以共享这一功能了。

**注意** 作为另一个经验作法，如果发现一个类经常访问另一个类的函数和数据，则有必要把那个功能放到另一个类中。

3) 最后，有一些方法用来标识应用程序，以帮助辨别它与每天用微软的 Developer Studio建立的其他3700个应用程序的不同。

- 增加一个在应用程序开始启动时显示的启动窗口 (见例11)。
- 显示一个指示进展的动画，而不是一个“ Please Wait ”消息(见例43)。
- 使工具栏按钮更大并把文字粘到它的上面 (见例24与例25)。
- 把其他类型的状态消息放到状态栏上 (见例31)。
- 在工具栏上增加控件而不是按钮，或者使用一个对话条 (见例28与例45)。

**说明**

用MFC编写应用程序，需要不断地决策是使用一个已存在的 MFC特征，还是自己编写一个特征。知道了第一部分的基础知识，应该能够轻松地编写一个按钮控件窗口，然后完全控制最终产品 —— 没有什么可怀疑的。如果它不能如所期望的那样工作，可以跟踪调试器直接进入这段程序，查明原因，如果需要的话，改正它。然而，创建一个按钮控件的工作已经完成，并经过验证的；这也可以被广泛地发现和理解，并允许他人更加容易地在中止的地方继续编写代码。如果提供函数的 MFC方法不是轻而易得的，建议应该继续搜索，直到发现一个或证实确实不存在这样的一种方法。一个用 MFC编写的而看上去在本质上与它一点不象的程序，是对它所用技术的浪费。按 MFC和Developer Studio原来的目的进行使用时，将会发现创建应用程序比想象中能达到的速度更快。

**CD说明**

CD上没有该例子相应的工程（Project）。

## 5.2 例2 用AppWizard创建一个MFC应用程序

**目标**

使用Developer Studio的AppWizard创建一个对话框、单文档界面或多文档界面应用程序。

**步骤**

使用AppWizard创建一个应用程序

1) 单击Developer Studio的File(文件)菜单中的New(新建)命令，以显示New对话框，选择MFC(AppWizard (exe))，然后输入需要创建工程的名称和目录 (见图5-1)。注意，几乎在所有的工程文件的内部和外部都使用该名字，因此，此处的任何错误在以后都是很难改正的。

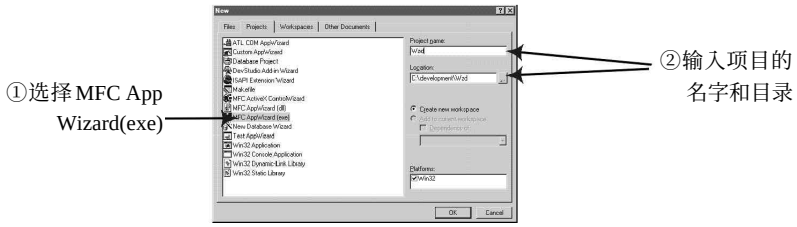


图5-1 指定应用程序的文件名和位置

2) AppWizard的第一步是选择应用程序的类型 (见图5-2)，如果还不了解应用程序类型，参见前面的例子以形成工程所基于的应用程序类型的思想。该例子的其余部分假定已选定了一个单文档界面或多文档界面应用程序。

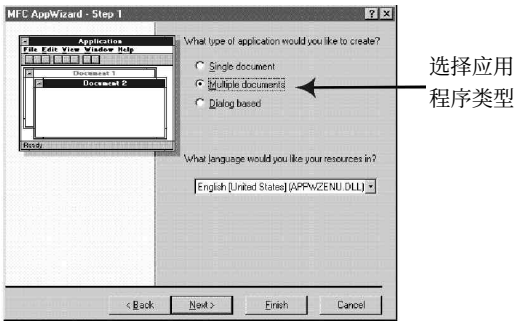


图5-2 选择应用程序类型

3) AppWizard的第二步要求指定应用程序所需要的数据库支持种类。选择 Header Files Only(只有头文件)使AppWizard只添加支持数据库访问的MFC类。因此，可以用例72或例73访问一个ODBC或DAO数据库，选择Database View whtiout File Support(数据库视图，没有文件支持)或者Database View With File Support(数据库支持，具有文件支持)使AppWizard创建一个具有特殊的视图和文档类的简单的数据库编辑器。如果选择 Database View without File Support，AppWizard将不添加标准文件打开命令到应用程序菜单 (即：File/New, File/Open等)。从理论上讲，如果只访问一个数据库，无论如何也不需这些命令 —— 当应用程序开始运行时，合适的数据库将被自动打开。然而，如果应用程序既要访问平面文件，又要访问数据库文件，则应该选择Database View with File Support。

4) AppWizard的第三步要求指定应用程序所需要的 COM支持。该书中的例子只采用了默认选项。

5) AppWizard的第四步要求选择一些基本的应用程序选项 (见图5-3)，可以选择应用程序

是否有一个工具栏或状态，是否添加打印命令到菜单，以及是否包括支持 e-mail或网络通信。Recent File list(当前文件列表)是应用程序打开的最近几个文件的一个列表，它由应用程序自动维护，用户必须决定这里的n是多少。单击Advanced(高级)按钮进行更高级的选项设置。

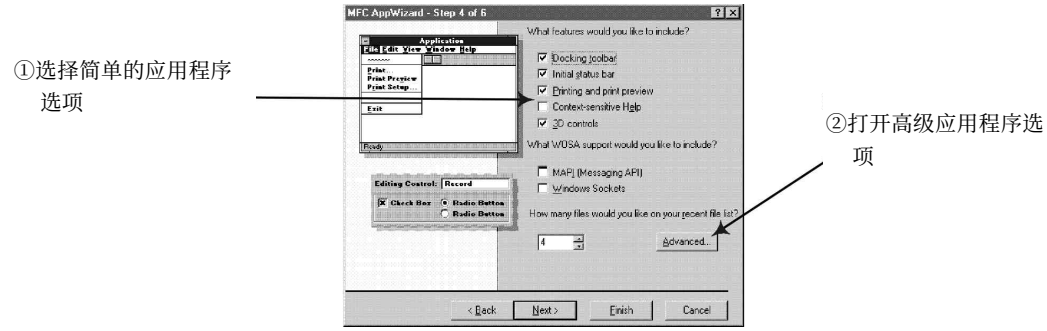


图5-3 选择应用程序选项

6) 高级选项的第一页要选择将在应用程序标题栏中显示的标题。如果创建一个串行化它的文档到磁盘的应用程序，则可以选择应用程序追加给那些文件的文件扩展名，然后就可以在打开或保存文档时显示的文件对话框的过滤域中编辑文本 (见图5-4)。

7) 高级选项的第二页要求在应用程序中添加视图分割 (Splitting)能力，它将提供允许动态分割他们的视图的一个菜单命令(见例37)。也可以决定是否要求应用程序主窗口或子窗口初始最大化或初始最小化，以及用户是否能够重调它们的大小（见图 5-5和例4)。

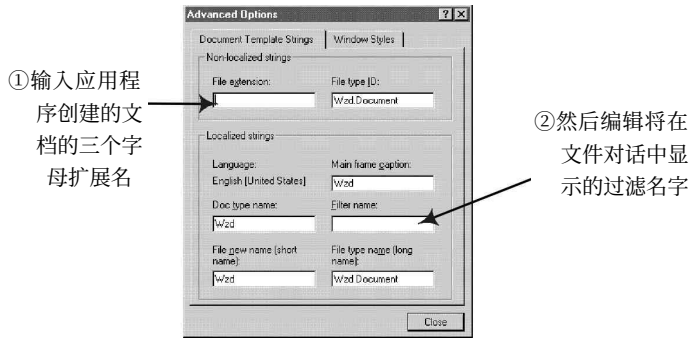


图5-4 指定应用程序标题、缺省文件扩展名和文件对话文本

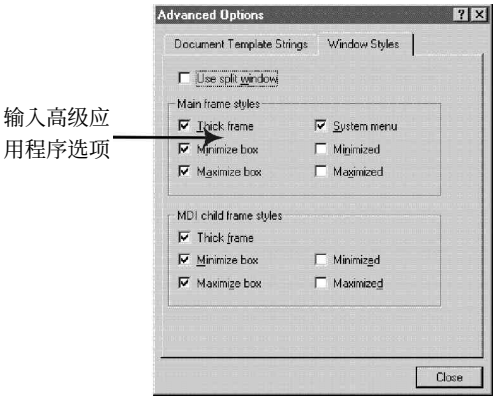


图5-5 指定应用程序的框架窗口选项

8) 在AppWizard的第五步，必须决定是用MFC库静态链接，还是用一个共享的MFC DLL进行链接(见图5-6)。静态链接MFC库使应用程序相当庞大，但用户从来不必考虑正确版本的MFC DLL当前是否已经安装在系统上。如果计划用MFC库创建自己的DLL，则必须链接MFC DLL。

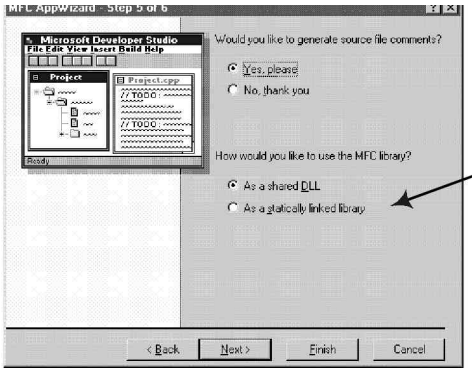


图5-6 选择怎样链接MFC

9) AppWizard的最后一步，允许改变应用程序的视图类。关于选项的描述。参见上个例子。对于没有列出的任何视图类，选择缺省视图类 —— 可以在以后编辑名字(见图5-7)。

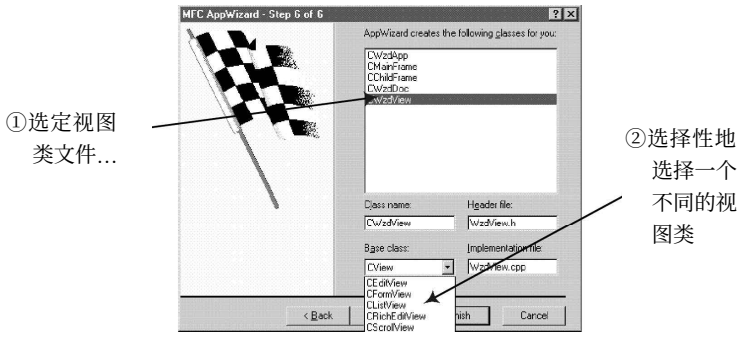


图5-7 选择一个视图类

10) 现在，AppWizard将继续建立应用程序的所有类，这些类是创建一个完全可以执行应用程序(虽然特征贫乏)所必须的。只要单击Developer Studio的Build/Build All菜单命令，便可以建立执行文件。

**说明**

- 如果忘了用AppWizard把某项内容添加到应用程序中，虽然不如自动建立容易，但还是可以在以后人工添加它。为了查明还必须给应用程序添加什么内容，从创建两个新工程开始，一个具有期望的特征，而另一个没有；然后观察由这两个工程产生的源码的不同。只要把不同点添加到缺少内容的应用程序，即可实现手工添加。
- 除了特别提到的例子，本书中的大多数例子基于用所有缺省值创建一个 MDI应用程序。
- 如果以后确实想改变应用程序名，首先，删除目录中的所有二进制文件；然后，使用一个可以跨越多个文件搜索和替换工具，用新名字替换旧名字。确信已替换了所有形

式的工程名，包括所有大小、所有小写和混合情况。 (搜索和替换工具在 Internet上可得。)

**CD说明**

CD上没有该例子相应的工程。

## 5.3 例3 用ClassWizard创建一个类

**目标**

把一个类添加到MFC应用程序，该类既可以扩展一个MFC类，也可以独立存在。

**步骤**

1. 扩展一个已存在的MFC类

1) 单击Developer Studio的View / ClassWizard菜单命令，打开MFC ClassWizard对话框，然后单击Add Class(添加类)按钮(见图5-8)，将出现一个下拉菜单，从中应选择 New…以打开New Class(新类)对话框。

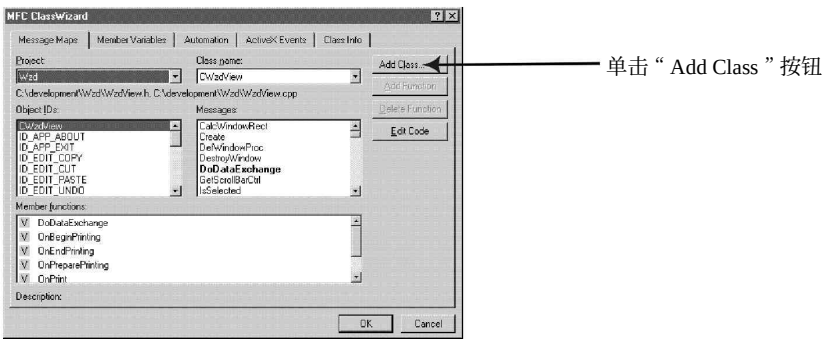


图5-8 用ClassWizard创建一个新类

2) 输入新类的名字，在新类的名字前面添加一个“ C ”(当创建类的 .h和.cpp文件时，ClassWizard将删除这个“ C ”)，然后从可用的MFC类列表中选择一个基类(见图5-9)。如果选择CRecordSet，ClassWizard还将引导用户通过捆绑一个数据库表到新类所必须的步骤。要从CWnd类派生，可以选择“ generic CWnd ”；要派生CSplitterWnd，选择“ splitter ”。如果想派生的MFC类没有列出(诸如用CToolBar)，则选择一个类似的名字(如CToolBarCtrl),然后编辑由此产生的文件。

2. 创建一个非MFC类

如果不想用一个MFC类作为基类，则单击环境的 Insert/New Class菜单命令，以打开一个选择性的New Class对话框。这一选择性版增加了用以指定 Class Type的组合框。选择Generic Type要求指定自己的基类(如果有的话)。

好，刚刚已添加了一个类，但是……糟糕，拼错了它的名字，现在想删除它以便加入一个正确拼写的名字。这时便开始查找一个 Delete Class按钮，经过半小时的查找还是没有找到

它；突然，该误拼显得并非完全不好，因此，不再查找 Delete Class按钮，并继续使用它；或者继续进行到下一步，学习怎样人工删除一个类。

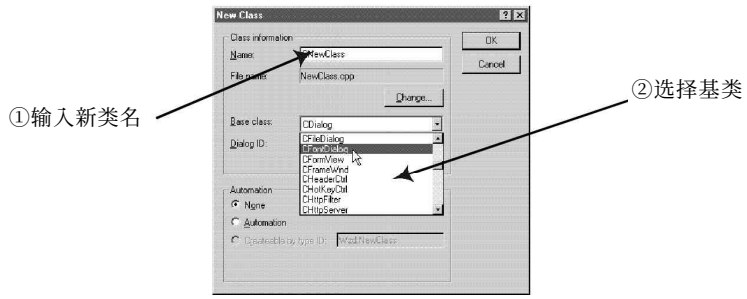


图5-9 选择一个基类

3. 从ClassWizard删除一个类

1) 首先，必须从文件的工程列表和工程的子目录中删除 ClassWizard创建的.cpp和.h文件。

2) 你也许会认为那就是所有必须做的了，但是并非如此，下一次应用 ClassWizard时，误拼的类象一个幽灵一样仍然在那里， ClassWizard对每一个在各自的.clw文件中创建的类保持一个记录。不过只要删除.clw文件，下次调用ClassWizard时，它就会告诉你没有找到.clw文件并提问是否想创建一个新的，回答 Yes，则ClassWizard将用工程目录中的.h文件建立一个新的.clw文件。

**说明**

- 从其他工程中插入一个类，只要从那个工程目录中直接拷贝相应的文件即可。ClassWizard不能识别该新类，直到做了下面这步工作：删除工程中的 .clw文件，并再次调用ClassWizard。当ClassWizard不能找到它的.clw文件时，它将提问是否想重建它，回答Yes。
- Developer Studio6.0版本的ClassWizard自动更新它的.clw文件。

**CD说明**

CD上没有该例子相应的工程。

## 5.4 例4 初始化应用程序屏幕

**目标**

设定应用程序初始屏幕的位置和大小。

**策略**

我们有两种可选方案。首先，创建应用程序时在 ClassWizard的高级选项中作出适当的选择；然而，如果想改变一个已有的应用程序中的选择，我们将把代码添加到 CMainFrame的PreCreateWindow( )中，以控制应用程序主窗口的初始位置和大小。

**步骤**

1. 使用AppWizard

1) 参考5.2节例2中第7步，单击Advanced按钮，选择Window Style标签。选择一个Thick Frame，允许用户通过拖动窗口的右下角来重新调整应用程序窗口的大小。选择 Minimized(最小化)或Maximized(最大化)，则使窗口初始最小化或最大化。

2) 如果想在以后改变选择，则需要直接编辑 CMainFrame::PreCreateWindow( )函数。直接编辑还允许为应用程序的初始外观作某些额外的变化。

2. 编辑CMainFrame::PreCreateWindow( )

1) 使应用程序位于屏幕中央，并只占据屏幕的 90%，可以使用下面的代码：

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   // center window at 90% of full screen
   int xSize = ::GetSystemMetrics (SM_CXSCREEN);
   int ySize = ::GetSystemMetrics (SM_CYSCREEN);
   cs.cx = xSize*9/10;
   cs.cy = ySize*9/10;
   cs.x = (xSize-cs.cx)/2;
   cs.y = (ySize-cs.cy)/2;
   return CMDIFrameWnd::PreCreateWindow(cs);
}
```

2) 如果还想从应用程序的标题栏中删除文档标题，则把下面的代码添加到 PreCreate Window( )中。

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

3)如果还想从应用程序标题栏中去掉最小化和最大化按钮，则添加：

```
cs.style &=~(WS_MAXIMIZEBOX|WS_MINIMIZEBOX);
```

4) 如果想使应用程序的大小固定，以致拖动窗口的右下角时没有反映，那么添加：

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

5) 如果想应用程序在开始执行时被最大化，那么找到应用程序类中的 ShowWindow()，并改变它，令它使用SW_SHOWMAXIMIZED标志而不用m_nCmdShow。

```
pMainFrame->ShowWindow(SW_SHOWMAXIMIZED); //or SW_SHOWMINIMIZED
pMainFrame->UpdateWindow();
```

6) 如果想在一个MDI应用程序中初始最大化一个子窗口，则把 PreCreateWindow( )添加到CChildFrame类中，并给它添加如下代码：

```
BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
   cs.style = WS_CHILD | WS_VISIBLE | WS_OVERLAPPED |
   WS_CAPTION | WS_SYSMENU | FWS_ADDTOTITLE |
   WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
   WS_MAXIMIZE;
   return CMDIChildWnd::PreCreateWindow(cs);
}
```

**说明**

- 如果初始最大化应用程序的窗口，还应该在 CMainFrame::PreCreateWindow( )中为它设置一个初始大小。当用户单击应用程序的恢复按钮时，应用程序窗口能够迅速缩小到在PreCreateWindow( )中设定的大小。
- 本例总是把应用程序窗口的初始大小设置为固定，并且位置也固定。用户作用于窗口大小或位置的变化都不作保存，保存窗口的大小和位置参见下一个例子。然而，如果使用下一个例子，仍然应该会使用本例。应用程序第一次在系统上运行时，它没有任何保存的设置值，因此，它需要使用这些初始设置值。
- 如果不设置窗口的初始大小和位置， Windows操作系统将根据级联 (Cascading)算法选择一个。每个新的应用程序的窗口被建立在右边，并在最后一个应用程序的底下，如第1章所示。

**CD说明**

在执行CD上的该工程时，应用程序窗口被初始最大化 (充满屏幕)，并且不能调整大小。

## 5.5 例5 保存应用程序屏幕

**目标**

保存应用程序屏幕的大小、位置和状态，包括每个工具栏或对话条的位置和大小，以便应用程序在下一次运行时能恢复它们。

**策略**

当应用程序关闭时，不仅保存主窗口的大小和位置，而且还把工具栏和状态栏的状态保存到系统的一个注册区。因此，当应用程序被再次打开时，我们将检索这些消息，并恢复窗口和工具栏等等。

**步骤**

1. 保存设置值

1) 在系统注册区中定义一个位置，在该位置用一个全局包含文件保存这些信息。“ Company ”是“ 公司 ”名。

```
#define COMPANY_KEY "Company"
#define SETTINGS_KEY "Settings"
#define WINDOWPLACEMENT_KEY "Window Placement"
```

2) 在应用程序类的InitInstance( )成员函数中，把COMPANY_KEY添加到SetRegistryKey( )中。

```
SetRegistryKey(COMPANY_KEY);
```

3) 用ClassWizard把一个 WM_CLOSE消息处理函数添加到 CMainFrame类中，用 SaveBarState( ) 可以保存条的位置和大小。用 GetWindowPlacement( )可以得到应用程序的当

前大小和位置，用WriteProfileBinary( )把它的结果保存到系统注册区。

```
void CMainFrame::OnClose()
{
   // save state of control bars
   SaveBarState("Control Bar States");
   // save size of screen
   WINDOWPLACEMENT wp;
   GetWindowPlacement(&wp);
   AfxGetApp()->WriteProfileBinary(SETTINGS_KEY,
      WINDOWPLACEMENT_KEY, (BYTE*)&wp,
      sizeof(WINDOWPLACEMENT));
   CMDIFrameWnd::OnClose();
}
```

2. 恢复设置值

1) 再次执行应用程序后，恢复工具栏到它们的初始状态，把下面的代码添加到CMainFrame类的OnCreate( )成员函数的开始位置。

```
LoadBarState("Control Bar States");
```

2) 要从系统注册表中恢复应用程序的主窗口，在应用程序类中找到 ShowWindow( )函数，并用下面的代码替换它。注意，现在我们用 SetWindowPlacement( )恢复主窗口到它的初始大小和位置。

```
BYTE *p;
UINT size;
WINDOWPLACEMENT *pWP;
if (GetProfileBinary(SETTINGS_KEY, WINDOWPLACEMENT_KEY, pWP, &size))
{
   pMainFrame->SetWindowPlacement(pWP);
   delete []pWP;
}
else
{
   pMainFrame->ShowWindow(m_nCmdShow);
}
pMainFrame->UpdateWindow();
```

**说明**

- 要把其他选项保存到系统注册表中，以及有关从应用程序访问系统注册表的详细讨论，参见例9。
- 本例只有当用户执行一次应用程序后才能指定应用程序主窗口的位置，要在第一次执行时初始化应用程序的窗口，请参看上一个例子。

**CD说明**

当在CD上执行该工程时，可以重定位主窗口，以及重设置主窗口的大小，然后退出应用程序。当再次执行该应用程序时，应用程序窗口将恢复到上次退出时的任何大小和位置。

## 5.6 例6 处理命令行选项

### 目标

让应用程序处理这里所见的命令行标志。

```
>myapp /c /d
```

### 策略

一个MFC应用程序可以用CCommandLineInfo类的成员函数ParseParam( )处理一些标准标志。要添加我们自己的标志，而仍然能够支持另外一些标志，我们将从 CCommandLineInfo派生类，然后重载ParseParam( )。

### 步骤

1. 创建一个新的CCommandLineInfo类

1) 用ClassWizard创建一个派生于CCommandLineInfo的新类。在新类中，为应用程序要处理的每个新的标志添加一个Boolean或String成员变量。

```
class CWzdCommandLineInfo : public CCommandLineInfo
{
public:
    BOOL m_bAFlag;
    BOOL m_bCFlag;
    BOOL m_bDAFlag;
    CString m_sArg;
```

2) 添加一个ParseParam( )函数，以重载基类的ParseParam( )函数。

```
// Operations
public:
    void ParseParam(const TCHAR* pszParam,BOOL bFlag,BOOL bLast);
};
```

3) 如下实现ParseParam( )：

```
void CWzdCommandLineInfo::ParseParam(const TCHAR* pszParam,
    BOOL bFlag,BOOL bLast)
{
    CString sArg(pszParam);
    if (bFlag)
    {
        m_bAFlag = !sArg.CompareNoCase("a");
        m_bCFlag = !sArg.CompareNoCase("c");
        m_bDAFlag = !sArg.CompareNoCase("da");
    }

    // m_strFileName gets the first nonflag name
    else if (m_strFileName.IsEmpty())
    {
        m_sArg=sArg;
    }
```

```
    CCommandLineInfo::ParseParam(pszParam,bFlag,bLast);
}
```

注意到变量pszParam包括命令行中的下一项。如果 pszParam的后面是一个— (连字符)或/(正斜杠)字符，则bFlag变量为TRUE，这些字符将被删除；如果 pszParam是一行中最后一个变量，则bLast为TRUE。确信最后调用基类的ParseParam( )，否则标准标志不被处理。

4) 有关命令行消息类的详细清单，参见本节的“清单 —— 命令行消息类”。

2. 把新的命令行消息类插到应用程序类中

1) 在应用程序类中找到ParseCommandLine( )，并用该新类替换CCommandLineInfo类。

```
// Parse command line for standard shell commands, DDE, file open
CWzdCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
```

2) 现在，命令行选项不能作为cmdInfo变量的成员变量。

```
if (cmdInfo.m_bAFlag)
{
    :  :  :
}
```

3) 要使这些选项在整个应用程序中可得，则把 cmdInfo嵌入应用程序中，并访问它的成员变量。

```
AfxGetApp()->m_cmdInfo.m_bAFlag;
```

**说明**

- 标准MFC标志如下，真正处理这些标准命令行发生在 ProcessShellCommand (cmdInfo)中，它正好在应用程序类中ParseCommandLine( )之后。

| | |
|---|---|
| nothing | 使应用程序试图打开一个新文档 |
| filename | 使应用程序试图以文档方式打开文件名 |
| /p filename | 使应用程序打开并打印给定的文件名到默认的打印机 |
| /pt filename | 与上面相同，但输入到指定的打印机 |
| printer driver port | |
| /dde | 使应用程序开始运行，并等待DDE命令 |
| /Automation | COM标志 |
| /Embedding | |
| /Unregister | |
| /Unregserver | |

- 处理非标准标志(如名字)会有点复杂，我们认为出现的第一个非标准标志是文档文件名。然而，一旦一个文件名被发现，可以根据目的攫取任何非标准标志，这就是说，除非遇到/pt标志，在这种情况下，下面三个非标准标志变量用来初始化打印。为了简化起见，也可通过不把/pt标志传递给基类中的ParseParam( )来禁用/pt标志。
- 当然，如果不需继续支持前面所示的标准 MFC标志，则可以更加自由地行动。只要不用调用基类的ParseParam( )，可以使用任何标志或非标准标志选项。但是，不要因为能用非标准标志，而轻易放弃这些标准标志提供的功能。

### CD说明

当在CD上执行该工程时，在Wzd.cpp的ParseCmdLine( )函数处设置一个断点，监视新的CWzdCommandLineInfo转换命令行参数到以后能在应用程序中使用的标志。

### 清单——命令行消息类

```
#if !defined WZDCOMMANDLINEINFO_H
#define WZDCOMMANDLINEINFO_H

// WzdCommandLineInfo.h : header file
//

/////////////////////////////////////////////////////////////////////
// CWzdCommandLineInfo window

class CWzdCommandLineInfo : public CCommandLineInfo
{

// Construction
public:
    CWzdCommandLineInfo();

// Attributes
public:
    BOOL m_bAFlag;
    BOOL m_bCFlag;
    BOOL m_bDAFlag;
    CString m_sArg;

// Operations
public:
    void ParseParam(const TCHAR* pszParam,BOOL bFlag, BOOL bLast);

// Overrides

// Implementation
public:
    virtual ~CWzdCommandLineInfo();
};

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

#endif

// WzdCommandLineInfo.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "WzdCommandLineInfo.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////
// CWzdCommandLineInfo

CWzdCommandLineInfo::CWzdCommandLineInfo()
{
    m_bAFlag = FALSE;
    m_bCFlag = FALSE;
    m_bDAFlag = FALSE;
    m_sArg=_T("");
}

CWzdCommandLineInfo::~CWzdCommandLineInfo()
{
}

//////////////////////////////////////////////////////////////////
void CWzdCommandLineInfo::ParseParam(const TCHAR* pszParam, BOOL bFlag,
    BOOL bLast)
{
    CString sArg(pszParam);
    if (bFlag)
    {
    m_bAFlag = !sArg.CompareNoCase("a");
    m_bCFlag = !sArg.CompareNoCase("c");
    m_bDAFlag = !sArg.CompareNoCase("da");
    }

    // m_strFileName gets the first nonflag name
    else if (m_strFileName.IsEmpty())
    {
        m_sArg=sArg;
    }
    CCommandLineInfo::ParseParam(pszParam,bFlag,bLast);
}
```

## 5.7 例7 动态改变应用程序图标

**目标**

改变应用程序的图标，甚至使图标有动感 (见图5-10)，该图标也可以出现在系统的任务栏里，并显示应用程序的进展，即使窗口最小化时也可以。

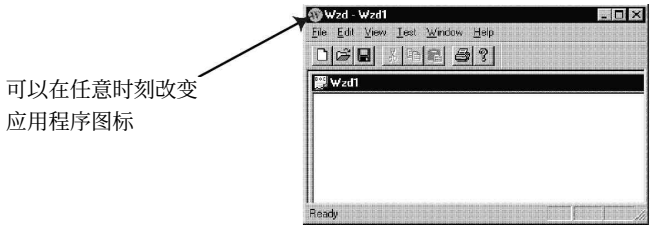


图5-10 应用程序的图标可以在主窗口和系统的任务栏中被改变

**策略**

用MFC的CWnd::SetIcon( )函数改变图标。

**步骤**

改变应用程序的图标

用下面的代码可以动态地改变应用程序的图标。从这里可以看到，也可以先用CWinApp::LoadIcon( )从应用程序资源中装入图标。

```
AfxGetMainWnd()->SetIcon(
   AfxGetApp()->LoadIcon(IDI_STATUS_ICON), // icon handle
   TRUE);                    // FALSE=16x16 bit icon
```

注意我们用CWinApp::LoadIcon( )从应用程序的资源中装入图标。

**说明**

- SetIcon( )不仅可以改变应用程序的图标，也可以改变任何有系统菜单的应用程序窗口的图标，包括子框架窗口和对话框。例如，从视图类中改变子框架窗口的图标，可以用下面的代码：

```
GetParentFrame()->SetIcon(
   AfxGetApp()->LoadIcon(IDI_STATUS_ICON),
   TRUE);
```

- 仅仅需要在运行时改变一个图标时，使用 SetIcon( )，否则，应该用Developer Studio的编辑器改变图标，在应用程序资源的图标文件下编辑 IDR_MAINFRAME图标。

**注意** 确信既编辑了32×32位图标，又编辑了16×16位图标。许多新手只编辑32×32位图标，并花费大量的时间查找为什么他们的图标不能被改变。创建一个 MDI应用程序时，也要确信在Icon文件夹中编辑另一个图标，以改变子框架图标。

- 前面已提过，应用程序最小化时可以改变应用程序图标，以指示它还在处理一个命令。只要检查AfxGetMainWnd( )—>IsIconic( )，如果为TRUE，则用SetIcon( )更新图标。
- CButton类也有一个SetIcon函数。然而，在本例中用它设置显示在按钮表面的图标。但是，确信创建该按钮时使用BS_ICON按钮风格。

**CD说明**

在CD上执行该工程时，单击Test/Wzd命令，以改变应用程序的图标。

## 5.8 例8 提示用户优先选项

### 目标

维护用户的优先选项(见图5-11)。

### 策略

用属性表和属性页提示用户有关他们的优先选项；首先，用菜单编辑器添加一个Options(选项)菜单到主菜单；然后，通过创建一个属性表，用ClassWizard处理该命令；并将为应用程序支持的选项添加属性页到属性表中。

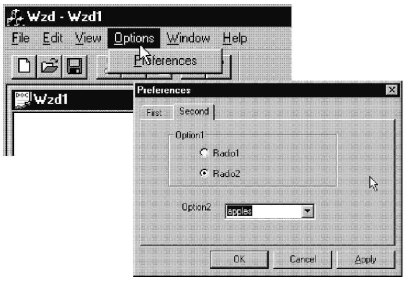


图5-11 用属性表和属性页维护用户的优先选项

### 步骤

1. 创建属性页

1) 用对话框编辑器创建一个或多个对话框模板 (见例38)，这些模板应该包含应用程序支持的所有优先选项。每个模板的风格应该是 Thin边框、Child和没有系统菜单，不管为该模板选择什么标题都将成为该优先选项标签上看到的名字。尽管使得属性页有相同的大小，但是，整个属性表的大小将根据添加到它的最大对话框模板的大小来确定。

2) 用ClassWizard为每个对话框模板创建一个类，从 CPropertyPage派生它们。为对话框中的每个控件创建一个成员变量 (见例39)，也为每个控件添加消息处理函数，指示控件被修改。在那个处理函数中，调用 SetModified (TRUE)，这将通知属性表启用 Apply按钮。

```
void CFirstPage::OnChange()
{
   SetModified(TRUE);
}
```

3) 在第一个属性页的类中，用 ClassWizard重载OnOK( )，同时在该类中，把用户定义的称为WM_APPLY的窗口消息发送到 CMainFrame类。还要在那个类中调用 SetModified (FALSE)关闭Apply按钮；每当OK或Apply按钮被击中时，每页上的 OnOK( )函数就被调用，但是只需在一页中处理它。我们将在以后分析 WM_APPLY消息的效应。

```
#define WM_APPLY WM_USER+1
:  :  :
void CFirstPage::OnOK()
{
   AfxGetMainWnd()->SendMessage(WM_APPLY);
   SetModified(FALSE);

   CPropertyPage::OnOK();
}
```

4) 有关属性页类的详细清单参见本节的“ 清单 —— 属性页类 ”。

2. 创建属性表

1) 用菜单编辑器创建一个新的主菜单项 —— Options，它只有一个唯一的子菜单项 ——

Preferences。

2) 在CMainFrame类中，用ClassWizard处理Preferences子菜单项。

3) 首先用CPropertySheet类创建的一个属性表，以处理该 Preferences命令；然后用AddPage( )将前面创建的类作为属性值添加到属性表中；用任何当前设置值，初始化该页的成员变量，然后用DoModal显示属性表。

```
void CMainFrame::OnOptionsPreferences()
{
   CPropertySheet sheet(_T("Preferences"),this);
   m_pFirstPage=new CFirstPage;
   m_pSecondPage=new CSecondPage;

   sheet.AddPage(m_pFirstPage);
   sheet.AddPage(m_pSecondPage);

   m_pFirstPage->m_bOption1 = m_bFirstOption1;
   m_pFirstPage->m_sOption2 = m_sFirstOption2;
   m_pSecondPage->m_nOption1 = m_nSecondOption1;
   m_pSecondPage->m_sOption2 = m_sSecondOption2;

   sheet.DoModal();
   delete m_pFirstPage;
   delete m_pSecondPage;
}
```

在最后一步中，来自属性页的值从不存储在产生它们的应用程序中，那是 WM_APPLY消息进来的地方。

4) 通过把下面的项目添加到 MainFrm.cpp的消息映像中，为WM_APPLY窗口消息手工添加一个消息处理程序。注意把它放到 //}}AFX_MSG_MAP符号下面，否则ClassWizard可能删除它。

```
ON_MESSAGE_VOID(WM_APPLY_OnApply);
```

在MainFrm.h中定义OnApply( )如下：

```
afx_msg void OnApply();
```

5) 现在执行CMainFrame中的OnApply( )，以便它把来自属性页的变量存回到应用程序中。前面已提过，每当属性表上的 Apply或OK按钮被单击时，该消息将被发送。

```
void CMainFrame::OnApply()
{
   m_bFirstOption1 = m_pFirstPage->m_bOption1;
   m_sFirstOption2 = m_pFirstPage->m_sOption2;
   m_nSecondOption1 = m_pSecondPage->m_nOption1;
   m_sSecondOption2 = m_pSecondPage->m_sOption2;
}
```

**说明**

- 习惯上，应用程序优先选项在 CMainFrame类中被处理，尽管真正的选项存储的地方依据它使用的地方而不同。

- 如果想把用户自己的按钮添加到属性表，从 CPropertySheet派生类，并在前面的例子中使用该派生类；要添加用户自己的按钮，首先，必须使属性页大得足以处理它们；用ClassWizard把一个处理 WM_CREATE的消息处理函数添加到新类中，用那里的MoveWindow( )使表成为需要的大小。要创建自己的按钮，则使用例 46所示的方法。
- 在对话框编辑器中看见标签控制工具时，首要假定它具有与属性表相同的功能。事实上，它没有任何功能，标签控件与列表控件更相似，因为它只跟踪所在的页，并且由你负责用一个对话框填充一属性页；如果完全可能的话，尽量避免单独使用一个标签控件。

### CD说明

在CD上执行该工程时，单击Options/Preferences菜单命令以打开一个属性表。

### 清单 —— 属性页类

```
#if !defined AFX_FIRSTPAGE_H
#define AFX_FIRSTPAGE_H

// FirstPage.h : header file
//
/////////////////////////////////////////////////////////////////////
// CFirstPage dialog

class CFirstPage : public CPropertyPage
{
    DECLARE_DYNCREATE(CFirstPage)

// Construction
public:
    CFirstPage();
    ~CFirstPage();

// Dialog Data
    //{{AFX_DATA(CFirstPage)
    enum {IDD = IDD_FIRST_PAGE};
    BOOL m_bOption1;
    CString m_sOption2;
    //}}AFX_DATA

// Overrides
    // ClassWizard generate virtual function overrides
    //{{AFX_VIRTUAL(CFirstPage)
    public:
    virtual void OnOK();
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
```

```
// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CFirstPage)
    afx_msg void OnChange();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
// FirstPage.cpp : implementation file
//

#include "stdafx.h"
#include "wzd.h"
#include "FirstPage.h"
#include "WzdProject.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
// CFirstPage property page

IMPLEMENT_DYNCREATE(CFirstPage, CPropertyPage)

CFirstPage::CFirstPage() : CPropertyPage(CFirstPage::IDD)
{
    //{{AFX_DATA_INIT(CFirstPage)
    m_bOption1 = FALSE;
    m_sOption2 = _T("");
    //}}AFX_DATA_INIT
}

CFirstPage::~CFirstPage()
{
}

void CFirstPage::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CFirstPage)
    DDX_Check(pDX, IDC_CHECK, m_bOption1);
    DDX_Text(pDX, IDC_EDIT, m_sOption2);
```

```
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CFirstPage, CPropertyPage)
    //{{AFX_MSG_MAP(CFirstPage)
    ON_BN_CLICKED(IDC_CHECK, OnChange)
    ON_EN_CHANGE(IDC_EDIT, OnChange)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////
// CFirstPage message handlers

void CFirstPage::OnChange()
{
    SetModified(TRUE);
}

// only needed on one page!
void CFirstPage::OnOK()
{
    AfxGetMainWnd()->SendMessage(WM_APPLY);
    SetModified(FALSE);

    CPropertyPage::OnOK();
}
```

## 5.9 例9 保存和恢复用户优先选项

### 目标

保存并恢复用户在前面例子中选定的选项。

### 策略

使用六个CWinApp函数把选项值装入并保存到系统注册表中。

### 步骤

1. 配置应用程序

1) 把下面列出的 #define语句添加到主框架类的定义文件中，用用户公司名字来替换“ Company ”，以及用应用程序选项的描述性名字替换“ Optionx ”。

```
#define COMPANY_KEY "Company"
#define SETTINGS_KEY "Settings"
#define OPTION1_KEY "Option1"
#define OPTION2_KEY "Option2"
#define OPTION3_KEY "Option3"
#define OPTION4_KEY "Option4"
```

2) 在CMainFrame类中，用用户公司的系统注册表项修改InitInstance( )中的SetRegistrykey( )

调用。

```
SetRegistryKey(COMPANY_KEY);
```

2. 保存应用程序选项

1) 把一个新的成员函数添加到CMainFrame类中，用来保存选项。在该函数中，可用多达三种不同的CWinApp函数来保存整型、字符和二进制选项。

```
void CMainFrame::SaveOptions()
{
// integer options
   AfxGetApp()->WriteProfileInt(SETTINGS_KEY,OPTION2_KEY,
      m_nOption2);
// string options
   AfxGetApp()->
      WriteProfileString(SETTINGS_KEY,OPTION1_KEY,m_sOption1);
// binary options
   AfxGetApp()->WriteProfileBinary(SETTINGS_KEY, OPTION3_KEY,
      (BYTE*)&m_dOption3, sizeof(m_dOption3));
}
```

2) 用ClassWizard把一个WM_CLOSE消息处理函数添加到CMainFrame类中，并从那里调用SaveOptions( )。

3. 恢复应用程序选项

1) 把另一个成员函数添加到CMainFrame类中，用来装入选项。在那里，可以用另外三个CWinApp函数在系统注册表装入整数、字符型和二进制选项。

```
void CMainFrame::LoadOptions()
{
   // integer options
   m_nOption2=AfxGetApp()->
      GetProfileInt(SETTINGS_KEY,OPTION2_KEY, 3);

   // string options
   m_sOption1=AfxGetApp()->
      GetProfileString(SETTINGS_KEY,OPTION1_KEY,"Default");
   // binary options
   BYTE *p;
   UINT size;
   m_dOption3=33.3;
   if (AfxGetApp()->GetProfileBinary(SETTINGS_KEY,OPTION3_KEY,
      &p, &size))
   {
      memcpy(&m_dOption3,p,size);
      delete []p;
   }
}
```

2) 在CMainFrame类的OnCreate( )函数的开始处调用LoadOptions( ).。

GetProfileBinary( )和WriteProfileBinary( )是非文档化的，并且可能在所有版本的 MFC中都找不到，如果它不在用户的 MFC中，下面步骤可以创建备份的 LoadOptions( )和SaveOptions( )，下面的方法还允许用户从系统注册表的任何位置保存和装入选项。

4. 备用LoadOptions( )和SaveOptions( )

1) 在项目的包含文件中添加一项#define语句，并用应用程序名字替换“ Wzd ”。

```
#define APPLICATION_KEY "Software\\Company\\Wzd\\Settings"
```

2) 然后，把下面的代码写入SaveOptions( )函数中，该函数打开系统注册表，并把选项写到它里面，然后直接用Window API关闭它。

```
void CMainFrame::SaveOptions()
{
   // opens system registry for writing
   HKEY key;
   DWORD size, type, disposition;

   if (RegOpenKeyEx(HKEY_CURRENT_USER,APPLICATION_KEY,0,
      KEY_WRITE,&key)!=ERROR_SUCCESS)
   {
      RegCreateKeyEx(HKEY_CURRENT_USER,APPLICATION_KEY,0,"",
         REG_OPTION_NON_VOLATILE,KEY_ALL_ACCESS,NULL,
         &key,&disposition);
   }

   // writes an option that's a string
   type = REG_SZ;
   size = m_sOption1.GetLength();
   RegSetValueEx(key,OPTION1_KEY,0,type,
      (LPBYTE)LPCSTR(m_sOption1),size);

   // writes an option that's an integer
   type = REG_DWORD;
   size = 4;
   RegSetValueEx(
      key,OPTION2_KEY,0,type,(LPBYTE)&m_nOption2,size);

   // writes all other options
   type = REG_BINARY;
   size = sizeof(DOUBLE);
   RegSetValueEx(
      key,OPTION3_KEY,0,type,(LPBYTE)&m_dOption3,size);

   // closes system registry
   RegCloseKey(key);

}
```

3) 在LoadOptions( )中添加下面的代码，它将打开系统注册表，读进选项，然后关闭系统注册表。

```
void CMainFrame::LoadOptions()
{
   HKEY key;
   DWORD size, type;
   if (RegOpenKeyEx(
```

```
        HKEY_CURRENT_USER,APPLICATION_KEY,0,KEY_READ,&key) ==
        ERROR_SUCCESS)
    {
        // read string options
        size=128;
        type = REG_SZ;
        LPSTR psz = m_sOption1.GetBuffer(size);
        RegQueryValueEx(
            key,OPTION1_KEY,0,&type,(LPBYTE)psz,&size);
        m_sOption1.ReleaseBuffer();

        // read integer options
        size=4;
        type = REG_DWORD;
        RegQueryValueEx(key,OPTION2_KEY,0,&type,
            (LPBYTE)&m_nOption2,&size);

    // read all other options as binary bytes
    size=sizeof(DOUBLE);
    type = REG_BINARY;
    RegQueryValueEx(key,OPTION3_KEY,0,&type,
        (LPBYTE)&m_dOption3,&size);

    RegCloseKey(key);
    }
}
```

**说明**

- 用Windows目录中的REGEDIT.EXE应用程序可以编辑写到系统注册表的选项，可以在HKEY_CURRENT_USER/Software/Company下找到应用程序项。
- 本例中我们以整型、字符串型和二进制型来保存一个选项，事实上所有选项可以以二进制值形式保存。然而，只要有可能，尽量使用一个整型或字符串型，因为REGEDIT.EXE应用程序可以更轻松地编辑这些值，字符串还是以字符串出现，整型还是以整型出现，而二进制型只以一串字节的形式出现。
- 如果在本节中“配置应用程序”的第二步注释掉 SetRegistryKey( )，则选项将被保存到文件中而不是系统注册表中。该文件将在你的 \Windows目录下，并命名为 app.mi, app是应用程序的名字。旧版的 Windows用这种方法来保存选项。

**CD说明**

在CD上执行该工程时，在Mainfrm.CPP的LoadOptionsl( )和SaveOptions1( )处各设一断点，然后运行应用程序并终止它，以观察被装入并存储到系统注册表中的选项。

## 5.10 例10 终止应用程序

**目标**

询问用户是否真的要终止应用程序。

**策略**

用ClassWizard把一个消息处理函数添加到 MainFrame类中，用以处理WM_CLOSE消息，在该函数中我们将提出问题，然后有条件终止应用程序。

**步骤**

提出问题

1) 用ClassWizard在CMainFrame类中添加一个WM_CLOSE消息处理函数。

2) 把下面的代码添加到该消息处理函数中。

```
void CMainFrame::OnClose()
{
   if (AfxMessageBox("Do you really want to exit?",MB_YESNO)
      == IDYES)
   {
      CMDIFrameWnd::OnClose();
   }
}
```

**说明**

- 当用户单击应用程序右上角的关闭按钮时 (或当他们选择File菜单中的Exit子菜单时)系统发送一个WM_CLOSE消息到应用程序。通过在这儿截获该消息来终止该命令的默认动作，即发送WM_DESTROY消息到应用程序的所有子窗口中。
- 在一个标准的MFC应用程序中，永远不需要使用该例子，因为 MFC应用程序是以文档为中心的，应当完全根据文档是否已经被修改而关闭应用程序。当修改一个文档时，应当用CDocument类的成员函数Modified (TRUE)标志文档已被修改。因此，当用户终止应用程序时， MFC会自动检查每个文档以查明它们是否已被修改，如果被修改，将询问用户是否想保存他们的改动；如果是， MFC将自动调用OnSaveModified( )应用程序；如果用户拒绝保存一个文档，关闭被取消；如果没有文档被修改，则应用程序不进行窥视就直接终止。
- 有关用ClassWizard添加一个消息处理函数的例子，参见例 59。

**CD说明**

在CD上执行该工程时，单击关闭按钮，弹出一个对话框提示，询问是否真的想终止应用程序；回答No，将使应用程序继续运行。

## 5.11 例11 创建一个启动窗口

**目标**

创建一个启动窗口，用以显示公司的标志和版权 (见图5-12)。

**策略**

尽我们所能，在应用程序类的 InitInstance()函数中，在最早时刻创建启动窗口。启动窗口

用一个位图类显示在普通窗口中，我们将在例 57中创建该位图类，以便窗口具有彩虹中的所有颜色。


应用程序的名字和版权正好在应用程序运行之前显示

图5-12 应用程序启动窗口可显示公司标志和版权

**步骤**

1. 创建一个启动窗口类

1) 用ClassWizard创建一个从一般CWnd类派生的普通窗口类。

2) 添加Create( )成员函数到该类，在该函数中装入启动窗口中显示的位图，以及在屏幕中央创建该窗口。用例 57中的位图类装入该位图，当把位图画到屏幕上时，允许保留它自己的调色板。

```
void CWzdSplash::Create(UINT nID)
{
   m_bitmap.LoadBitmapEx(nID,FALSE);
   int x = (::GetSystemMetrics (SM_CXSCREEN)-
      m_bitmap.m_Width)/2;
   int y = (::GetSystemMetrics (SM_CYSCREEN)-
      m_bitmap.m_Height)/2;
   CRect rect(x,y,x+m_bitmap.m_Width,y+m_bitmap.m_Height);
   CreateEx(0,AfxRegisterWndClass(0),"",
      WS_POPUP|WS_VISIBLE|WS_BORDER,rect,NULL,0);
}
```

3) 用ClassWizard添加一个WM_PAINT消息处理函数到该窗口类中，在这里用 BitBlt( )把位图绘制到屏幕上。

```
void CWzdSplash::OnPaint()
{
   CPaintDC dc(this); // device context for painting
   // get bitmap colors
   CPalette *pOldPal =
      dc.SelectPalette(m_bitmap.GetPalette(),FALSE);
   dc.RealizePalette();
   // get device context to select bitmap into
   CDC dcComp;
   dcComp.CreateCompatibleDC(&dc);
   dcComp.SelectObject(&m_bitmap);
   // draw bitmap
   dc.BitBlt(0,0,m_bitmap.m_Width,m_bitmap.m_Height, &dcComp,0,0,SRCCOPY);
   // reselect old palette
   dc.SelectPalette(pOldPal,FALSE);
}
```

4) 有关该启动窗口类的完备清单，参见本节的“清单——启动窗口类”。

2. 把启动窗口类插入到InitInstance( )中

1) 在应用程序类的InitInstance( )函数的开始处，创建该启动窗口类的一个实例：调用它的Create( )，并强制它进行绘制。

```
CWzdSplash wndSplash;
wndSplash.Create(IDB_WZDSPLASH);
wndSplash.UpdateWindow();    //send WM_PAINT
```

2) 因为启动窗口类创建在堆栈中，一旦InitInstance( )返回，该窗口将被自动销毁，因此，如果应用程序花费大量时间进行初始化，则不必采用在应用程序中加延时的方法，使启动窗口停留足够长的时间以确保读取显示的信息；如果应用程序花费很少时间进行初始化，或者担心高速机器把启动窗口变成影像，添加下面代码行到 InitInstance( )的某个地方以对应用程序进行延时处理。

```
// add if splash screen too short
Sleep(2000);  <<<<<<<
```

**说明**

- 创建一个具有动感的启动窗口，从CAnimateCtrl中派生启动窗口类。在Create( )中装入一个.avi文件，而不是一个位图文件，同时把窗口创建在屏幕中间。参见例 43有关CAnimateCtrl类的详尽讨论。

**CD说明**

在CD上执行该工程时，在显示应用程序窗口前先出现一个启动窗口，然后消失。

**清单——启动窗口类**

```
#if !defined WZDSPLASH_H
#define WZDSPLASH_H

// WzdSplash.h : header file
//

#include "WzdBitmap.h"

/////////////////////////////////////////////////////////////////////
// CWzdSplash window

class CWzdSplash : public CWnd
{

// Construction
public:
   CWzdSplash();

// Attributes
public:
```

```
// Operations
public:
    void Create(UINT nBitmapID);

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CWzdSplash)
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWzdSplash();

    // Generated message map functions
protected:
    //{{AFX_MSG(CWzdSplash)
    afx_msg void OnPaint();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CWzdBitmap m_bitmap;
};

/////////////////////////////////////////////////////////////////////////////
#include "WzdBitmap.h"

/////////////////////////////////////////////////////////////////////////////
// CWzdSplash window

class CWzdSplash : public CWnd
{

// Construction
public:
    CWzdSplash();

// Attributes
public:

// Operations
public:
    void Create(UINT nBitmapID);

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CWzdSplash)
    //}}AFX_VIRTUAL

// Implementation
```

```
public:
    virtual ~CWzdSplash();

    // Generated message map functions
protected:
    //{{AFX_MSG(CWzdSplash)
    afx_msg void OnPaint();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CWzdBitmap m_bitmap;
};

/////////////////////////////////////////////////////////////////////
#endif

// WzdSplash.cpp : implementation file
//

#include "stdafx.h"
#include "WzdSplash.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////
// CWzdSplash

CWzdSplash::CWzdSplash()
{
}

CWzdSplash::~CWzdSplash()
{
}

BEGIN_MESSAGE_MAP(CWzdSplash, CWnd)
    //{{AFX_MSG_MAP(CWzdSplash)
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////
// CWzdSplash message handlers

void CWzdSplash::OnPaint()
{
```

```
    CPaintDC dc(this);    // device context for painting
    // get bitmap colors
    CPalette *pOldPal = dc.SelectPalette(m_bitmap.GetPalette(),FALSE);
    dc.RealizePalette();

    // get device context to select bitmap into
    CDC dcComp;
    dcComp.CreateCompatibleDC(&dc);
    dcComp.SelectObject(&m_bitmap);

    // draw bitmap
    dc.BitBlt(0,0,m_bitmap.m_Width,m_bitmap.m_Height, &dcComp, 0,0,SRCCOPY);

    // reselect old palette
    dc.SelectPalette(pOldPal,FALSE);
}

void CWzdSplash::Create(UINT nID)
{
    m_bitmap.LoadBitmapEx(nID,FALSE);

    int x = (::GetSystemMetrics (SM_CXSCREEN)- m_bitmap.m_Width)/2;
    int y = (::GetSystemMetrics (SM_CYSCREEN)- m_bitmap.m_Height)/2;
    CRect rect(x,y,x+m_bitmap.m_Width,y+m_bitmap.m_Height);
    CreateEx(0,AfxRegisterWndClass(0),"",
        WS_POPUP|WS_VISIBLE|WS_BORDER,rect,NULL,0);
}
```