WTL

(Windows Template Library)

使用教程

前言

在你开始使用 WTL 或者在本文章的讨论区张贴消息之前，我想请你先阅读下面的材料。你需要开发平台SDK(Platform SDK)。你要使用WTL不能没有它，你可以使用在线升级安装开发平台SDK，也可以 下载全部文件后在本地安装。在使用之前要将SDK的包含文件(.h头文件)和库文件(.Lib文件)路径添加到VC的搜索目录， SDK有现成的工具完成这个工作，这个工具位于开发平台SDK程序组的“Visual Studio Registration”文件夹里。

你需要安装 WTL。你可以从微软的网站上下载WTL的 7.0 版，在安装之前可以先查看“Introduction to WTL - Part 1”和“Easy installation of WTL”这两篇文章，了解一下所要安装的文件的信息，虽然现在这些文章有些过时，但还是可以提供很多有用的信息。有一件我认为不该在本篇文章中提到的事是告诉VC如何搜索WTL的包含文件路径，如果你用的VC6，用鼠标点击 Tools\Options，转到Directories标签页，在显示路径的列表框中选择Include Files，然后将WTL的包含文件的存放路径添加到包含文件搜索路径列表中。

你需要了解 MFC。很好地了解 MFC 将有助于你理解后面提到的有关消息映射的宏并能够编辑那些标有“不要编辑(DO NOT EDIT)”的代码而不会出现问题。

你需要清楚地知道如何使用 Win32 API 编程。如果你是直接从 MFC 开始学习 Windows 编程，没有学过 API 级别的消息处理方式，那很不幸你会在使用 WTL 时遇到麻烦。如果不了解 Windows 消息中 WPARAM 参数和 LPARAM 参数的意义，应该明白需要读一些这方面的文章(在 CodeProject 有大量的此类文章)。

你需要知道 C++ 模板的语法，你可以到VC Forum FAQ 相关的连接寻求答案。

我只是讨论了一些涵盖 VC 6 的特点，不过据我了解所有的程序都可以在 VC 7 上使用。由于我不使用 VC 7，我无法对那些在 VC 7 中出现的问题提供帮助，不过你还是可以放心的在此张贴你的问题，因为其他的人可能会帮助你。

对本系列文章的总体介绍:

WTL 具有两面性，确实是这样的。它没有 MFC 的界面(GUI)类库那样功能强大，但是能够生成很小的可执行文件。如果你象我一样使用 MFC 进行界面编程，你会觉得 MFC 提供的界面控件封装使用起来非常舒服，更不用说 MFC 内置的消息处理机制。当然，如果你也象我一样不希望自己的程序仅仅因为使用了 MFC 的框架就增加几百 K 的大小的话，WTL 就是你的选择。

当然，我们还要克服一些障碍:

ATL 样式的模板类初看起来有点怪异,

没有类向导的支持，所以要手工处理所有的消息映射。

MSDN 没有正式的文档支持，你需要到处去收集有关的文档，甚至是查看 WTL 的源代码。

买不到参考书籍

没有微软的官方支持

ATL/WTL 的窗口与 MFC 的窗口有很大的不同，你所了解的有关 MFC 的知识并不全部适用与 WTL。

从另一方面讲，WTL 也有它自身的优势:

不需要学习或掌握复杂的文档/视图框架。

具有 MFC 的基本的界面特色，比如 DDX/DDV 和命令状态的自动更新功能(译者加：比如菜单的 Check 标记和 Enable 标记)。

增强了一些 MFC 的特性(比如更加易用的分隔窗口)。

可生成比静态链接的 MFC 程序更小的可执行文件(译者加：WTL 的所有源代码都是静态链接到

你的程序中的)。
你可以修正自己使用的 WTL 中的错误(BUG)而不会影响其他的应用程序(相比之下，如果你修正了有 BUG 的 MFC/CRT 动态库就可能会引起其它应用程序的崩溃。
如果你仍然需要使用 MFC，MFC 的窗口和 ATL/WTL 的窗口可以“和平共处”。(例如我工作中的一个原型就使用了了 MFC 的 CFrameWnd，并在其内包含了 WTL 的 CSplitterWindow，在 CSplitterWindow 中又使用了 MFC 的 CDialogs -- 我并不是为了炫耀什么，只是修改了 MFC 的代码使之能够使用 WTL 的分割窗口，它比 MFC 的分割窗口好的多)。
在这一系列文章中，我将首先介绍 ATL 的窗口类，毕竟 WTL 是构建与 ATL 之上的一系列附加类，所以需要很好的了解 ATL 的窗口类。介绍完 ATL 之后我将介绍 WTL 的特性以并展示它是如何使界面编程变得轻而易举。

第一部分　ATL 中的 GUI 类

对第一章的简单介绍

WTL 是个很酷的工具，在理解这一点之前需要首先介绍 ATL。WTL 是构建与 ATL 之上的一系列附加类，如果你是个严格使用 MFC 的程序员那么你可能没有机会接触到 ATL 的界面类，所以请容忍我在开始 WTL 之前先罗索一些别的东西，绕道来介绍一下 ATL 是很有必要地。

在本文的第一部分，我将给出一点 ATL 的背景知识，包括一些编写 ATL 代码必须知道的基本知识，快速的解释一些令人不知所措的 ATL 模板类和基本的 ATL 窗口类。

ATL 背景知识

ATL 和 WTL 的发展历史

“活动模板库”(Active Template Library)是一个很古怪的名字，不是吗？那些年纪大的人可能还记得它最初被称为“网络组件模板库”，这可能是它更准确的称呼，因为 ATL 的目的就是使编写组件对象和 ActiveX 控件更容易一些(ATL 是在微软开发新产品 ActiveX-某某的过程中开发的，那些 ActiveX-某某现在被称为某某.NET)。由于 ATL 是为了便于编写组件对象而存在的，所以只提供了简单的界面类，相当于 MFC 的窗口类(CWnd)和对话框类(CDialog)。幸运的是这些类非常的灵活，能够在其基础上构建象 WTL 这样的附加类。

WTL 现在已经是第二次修正了，最初的版本是 3.1，现在的版本是 7(WTL 的版本号之所以这样选择是为了与 ATL 的版本匹配，所以不存在 1 和 2 这样的版本号)。WTL 3.1 可以与 VC 6 和 VC 7 一起使用，但是在 VC 7 下需要定义几个预处理标号。WTL 7 向下兼容 WTL 3.1，并且不作任何修改就可以与 VC 7 一起使用，现在看来没有任何理由还使用 3.1 来进行新的开发工作。

ATL-style 模板

即使你能够毫不费力地阅读 C++的模板类代码，仍然有两件事可能会使你有些头晕，以下面这个类的定义为例：

```
class  CMyWnd : public CWindowImpl<CMyWnd>
{
    ...
};
```

这样作是合法的，因为 C++的语法解释说即使 CMyWnd 类只是被部分定义，类名 CMyWnd 已经被列入递归继承列表，是可以使用的。将类名作为模板类的参数是因为 ATL 要做另一件诡秘的事情，那就是编译期间的虚函数调用机制。

如果你想要了解它是如何工作地，请看下面的例子：

```
template <class T>
class B1
{
public:
    void SayHi()
    {
        T* pT = static_cast<T*>(this);   // HUH?? 我将在下面解释
```

```
        pT->PrintClassName();
    }
protected:
    void PrintClassName() { cout << "This is B1"; }
};
class D1 : public B1<D1>
{
    // No overridden functions at all
};

class D2 : public B1<D2>
{
protected:
    void PrintClassName() { cout << "This is D2"; }
};

main()
{
    D1 d1;
    D2 d2;

    d1.SayHi();    // prints "This is B1"
    d2.SayHi();    // prints "This is D2"
}
```

这句代码 static_cast<T*>(this) 就是窍门所在。它根据函数调用时的特殊处理将指向 B1 类型的指针 this 指派为 D1 或 D2 类型的指针，因为模板代码是在编译其间生成的，所以只要编译器生成正确的继承列表，这样指派就是安全的。(如果你写成：

```
class D3 : public B1<D2>
```

就会有麻烦) 之所以安全是因为 this 对象只可能是指向 D1 或 D2(在某些情况下)类型的对象，不会是其他的东西。注意这很像 C++的多态性(polymorphism)，只是 SayHi()方法不是虚函数。

要解释这是如何工作的，首先看对每个 SayHi()函数的调用，在第一个函数调用，对象 B1 被指派为 D1，所以代码被解释成：

```
void B1<D1>::SayHi()
{
    D1* pT = static_cast<D1*>(this);

    pT->PrintClassName();
}
```

由于 D1 没有重载 PrintClassName()，所以查看基类 B1，B1 有 PrintClassName()，所以 B1 的 PrintClassName()被调用。

现在看第二个函数调用 SayHi()，这一次对象被指派为 D2 类型，SayHi()被解释成：

```
void B1<D2>::SayHi()
{
```

```
    D2* pT = static_cast<D2*>(this);

    pT->PrintClassName();
}
```

这一次，D2 含有 PrintClassName()方法，所以 D2 的 PrintClassName()方法被调用。

这种技术的有利之处在于：

不需要使用指向对象的指针。

节省内存，因为不需要虚函数表。

因为没有虚函数表所以不会发生在运行时调用空指针指向的虚函数。

所有的函数调用在编译时确定(译者加：区别于 C++的虚函数机制使用的动态编连)，有利于编译程序对代码的优化。

节省虚函数表在这个例子中看起来无足轻重(每个虚函数只有 4 个字节)，但是设想一下如果有 15 个基类，每个类含有 20 个方法，加起来就相当可观了。

ATL 窗口类

好了，关于 ATL 的背景知识已经讲的构多了，到了该正式讲 ATL 的时候了。ATL 在设计时接口定义和实现是严格区分开的，这在窗口类的设计中是最明显的，这一点类似于 COM，COM 的接口定义和实现是完全分开的(或者可能有多个实现)。

ATL 有一个专门为窗口设计的接口，可以做全部的窗口操作，这就是 CWindow。它实际上就是对 HWND 操作的包装类，对几乎所有以 HWND 句柄为第一个参数的窗口 API 的进行了封装，例如：SetWindowText() 和 DestroyWindow()。CWindow 类有一个公有成员 m_hWnd，使你可以直接对窗口的句柄操作，CWindow 还有一个操作符 HWND，你可以讲 CWindow 对象传递给以 HWND 为参数的函数，但这与 CWnd::GetSafeHwnd()(译者加：MFC 的方法)没有任何等同之处。

CWindow 与 MFC 的 CWnd 类有很大的不同，创建一个 CWindow 对象占用很少的资源，因为只有一个数据成员，没有 MFC 窗口中的对象链，MFC 内部维持这一个对象链，此对象链将 HWND 映射到 CWnd 对象。还有一点与 MFC 的 CWnd 类不同的是当一个 CWindow 对象超出了作用域，它关联的窗口并不被销毁掉，这意味着你并不需要随时记得分离你所创建的临时 CWindow 对象。

在 ATL 类中对窗口过程的实现是 CWindowImpl。CWindowImpl 含有所有窗口实现代码，例如：窗口类的注册，窗口的子类化，消息映射以及基本的 WindowProc()函数，可以看出这与 MFC 的设计有很大的不同， MFC 将所有的代码都放在一个 CWnd 类中。

还有两个独立的类包含对话框的实现，它们分别是 CDialogImpl 和 CAxDialogImpl，CDialogImpl 用于实现普通的对话框而 CAxDialogImpl 实现含有 ActiveX 控件的对话框。

定义一个窗口的实现

任何非对话框窗口都是从 CWindowImpl 派生的，你的新类需要包含三件事情：

一个窗口类的定义

一个消息映射链

窗口使用的默认窗口类型，称为 window traits

窗口类的定义通过 DECLARE_WND_CLASS 宏或 DECLARE_WND_CLASS_EX 宏来实现。这辆个宏定义了一个 CWndClassInfo 结构，这个结构封装了 WNDCLASSEX 结构。DECLARE_WND_CLASS 宏让你指定窗口类的类名，其他参数使用默认设置，而 DECLARE_WND_CLASS_EX 宏还允许你指定窗口类的类型和窗口的背景颜色，你也可以用 NULL 作为类名，ATL 会自动为你生成一个类名。

让我们开始定义一个新类，在后面的章节我会逐步的完成这个类的定义。

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
public:
   DECLARE_WND_CLASS(_T("My Window Class"))
};
```

接下来是消息映射链，ATL 的消息映射链比 MFC 的简单的多，ATL 的消息映射链被展开为 switch 语句，switch 语句正确的消息处理者并调用相应的函数。使用消息映射链的宏是 BEGIN_MSG_MAP 和 END_MSG_MAP，让我们为我们的窗口添加一个空的消息映射链。

```
class CMyWindow : public CWindowImpl<CMyWindow>
{
public:
   DECLARE_WND_CLASS(_T("My Window Class"))

   BEGIN_MSG_MAP(CMyWindow)
   END_MSG_MAP()
};
```

我将在下一节展开讲如何如何添加消息处理到消息映射链。最后，我们需要为我们的窗口类定义窗口的特征，窗口的特征就是窗口类型和扩展窗口类型的联合体，用于创建窗口时指定窗口的类型。窗口类型被指定为参数模板，所以窗口的调用者不需要为指定窗口的正确类型而烦心，下面是是同 ATL 类 CWinTraits 定义窗口类型的例子：

```
typedef               CWinTraits<WS_OVERLAPPEDWINDOW               |
WS_CLIPCHILDREN,WS_EX_APPWINDOW> CMyWindowTraits;

class    CMyWindow    :    public    CWindowImpl<CMyWindow,    CWindow,
CMyWindowTraits>
{
public:
   DECLARE_WND_CLASS(_T("My Window Class"))

   BEGIN_MSG_MAP(CMyWindow)
   END_MSG_MAP()
};
```

调用者可以重载 CMyWindowTraits 的类型定义，但是一般情况下这是没有必要的，ATL 提供了几个预先定义的特殊的类型，其中之一就是 CFrameWinTraits，一个非常棒的框架窗口：

```
typedef    CWinTraits<WS_OVERLAPPEDWINDOW    |    WS_CLIPCHILDREN    |
WS_CLIPSIBLINGS,
                                        WS_EX_APPWINDOW | WS_EX_WINDOWEDGE>
CFrameWinTraits;
```

填写消息映射链

ATL 的消息映射链是对开发者不太友好的部分，也是 WTL 对其改进最大的部分。类向导至少可以让你添加消息响应，然而 ATL 没有消息相关的宏和象 MFC 那样的参数自动展开功能，在 ATL 中只有三种类型的消息处理，一个是 WM_NOTIFY，一个是 WM_COMMAND，第三类是其他窗口消息，让我们开始为我们的窗口添加 WM_CLOSE 和 WM_DESTROY 的消息相应函数。

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow,
CFrameWinTraits>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
    END_MSG_MAP()

    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        DestroyWindow();
        return 0;
    }

    LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        PostQuitMessage(0);
        return 0;
    }
};
```

你可能注意到消息响应函数的到的是原始的 WPARAM 和 LPARAM 值，你需要自己将其展开为相应的消息所需要的参数。还有第四个参数 bHandled，这个参数在消息相应函数调用被 ATL 设置为 TRUE，如果在你的消息响应处理完之后需要 ATL 调用默认的 WindowProc()处理该消息，你可以讲 bHandled 设置为 FALSE。这与 MFC 不同， MFC 是显示的调用基类的响应函数来实现的默认的消息处理的。

让我们也添加一个对 WM_COMMAND 消息的处理，假设我们的窗口有一个 ID 为 IDC_ABOUT 的 About 菜单：

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow,
CFrameWinTraits>
{
public:
    DECLARE_WND_CLASS(_T("My Window Class"))

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
        COMMAND_ID_HANDLER(IDC_ABOUT, OnAbout)
    END_MSG_MAP()
```

```
    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        DestroyWindow();
        return 0;
    }

    LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        PostQuitMessage(0);
        return 0;
    }

    LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
    {
        MessageBox ( _T("Sample ATL window"), _T("About MyWindow") );
        return 0;
    }
};
```

需要注意得是 COMMAND_HANDLER 宏已经将消息的参数展开了，同样，NOTIFY_HANDLER 宏也将 WM_NOTIFY 消息的参数展开了。

高级消息映射链和嵌入类

ATL 的另一个显著不同之处就是任何一个 C++类都可以响应消息，而 MFC 只是将消息响应任务分给了 CWnd 类和 CCmdTarget 类，外加几个有 PreTranslateMessage()方法的类。ATL 的这种特性允许我们编写所谓的 “嵌入类”，为我们的窗口添加特性只需将该类添加到继承列表中就行了，就这么简单！

一个基本的带有消息映射链的类通常是模板类，将派生类的类名作为模板的参数，这样它就可以访问派生类中的成员，比如 m_hWnd(CWindow 类中的 HWND 成员)。让我们来看一个嵌入类的例子，这个嵌入类通过响应 WM_ERASEBKGND 消息来画窗口的背景。

```
template <class T, COLORREF t_crBrushColor>
class CPaintBkgnd : public CMessageMap
{
public:
    CPaintBkgnd() { m_hbrBkgnd = CreateSolidBrush(t_crBrushColor); }
    ~CPaintBkgnd() { DeleteObject ( m_hbrBkgnd ); }

    BEGIN_MSG_MAP(CPaintBkgnd)
        MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBkgnd)
    END_MSG_MAP()

    LRESULT OnEraseBkgnd(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
```

```
    {
        T*   pT = static_cast<T*>(this);
        HDC  dc = (HDC) wParam;
        RECT rcClient;

        pT->GetClientRect ( &rcClient );
        FillRect ( dc, &rcClient, m_hbrBkgnd );
        return 1;    // we painted the background
    }

protected:
    HBRUSH m_hbrBkgnd;
};
```

让我们来研究一下这个新类。首先，CPaintBkgnd 有两个模板参数：使用 CPaintBkgnd 的派生类的名字和用来画窗口背景的颜色。(t_ 前缀通常用来作为模板类的模板参数的前缀)CPaintBkgnd 也是从 CMessageMap 派生的，这并不是必须的，因为所有需要响应消息的类只需使用 BEGIN_MSG_MAP 宏就足够了，所以你可能看到其他的一些嵌入类的例子代码，它们并不是从该基类派生的。

构造函数和析构函数都相当简单，只是创建和销毁 Windows 画刷，这个画刷由参数 t_crBrushColor 决定颜色。接着是消息映射链，它响应 WM_ERASEBKGND 消息，最后由响应函数 OnEraseBkgnd()用构造函数创建的画刷填充窗口的背景。在 OnEraseBkgnd()中有两件事需要注意，一个是它使用了一个派生的窗口类的方法(即 GetClientRect ())，我们如何知道派生类中有 GetClientRect()方法呢？如果派生类中没有这个方法我们的代码也不会有任何抱怨，由编译器确认派生类 T 是从 CWindow 派生的。另一个是 OnEraseBkgnd()没有将消息参数 wParam 展开为设备上下文(DC)。(WTL 最终会解决这个问题，我们很快就可以看到，我保证)

要在我们的窗口中使用这个嵌入类需要做两件事：首先，将它加入到继承列表：

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow,
CFrameWinTraits>,
                public CPaintBkgnd<CMyWindow, RGB(0,0,255)>
```

其次，需要 CMyWindow 将消息传递给 CPaintBkgnd，就是将其链入到消息映射链，在 CMyWindow 的消息映射链中加入 CHAIN_MSG_MAP 宏：

```
class CMyWindow : public CWindowImpl<CMyWindow, CWindow,
CFrameWinTraits>,
                public CPaintBkgnd<CMyWindow, RGB(0,0,255)>
{
...
typedef CPaintBkgnd<CMyWindow, RGB(0,0,255)> CPaintBkgndBase;

    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
        COMMAND_HANDLER(IDC_ABOUT, OnAbout)
        CHAIN_MSG_MAP(CPaintBkgndBase)
```

```
    END_MSG_MAP()
...
};
```

任何 CMyWindow 没有处理的消息都被传递给 CPaintBkgnd。应该注意的是 WM_CLOSE，WM_DESTROY 和 IDC_ABOUT 消息将不会传递，因为这些消息一旦被处理消息映射链的查找就会中止。使用 typedef 是必要地，因为宏是预处理宏，只能有一个参数，如果我们将 CPaintBkgnd<CMyWindow, RGB(0,0,255)>作为参数传递，那个“,”会使预处理器认为我们使用了多个参数。

你可以在继承列表中使用多个嵌入类，每一个嵌入类使用一个 CHAIN_MSG_MAP 宏，这样消息映射链就会将消息传递给它。这与 MFC 不同，MFC 地 CWnd 派生类只能有一个基类，MFC 自动将消息传递给基类。

ATL 程序的结构

到目前为止我们已经有了一个完整地主窗口类(即使不完全有用)，让我们看看如何在程序中使用它。一个 ATL 程序包含一个 CComModule 类型的全局变量_Module，这和 MFC 的程序都有一个 CWinApp 类型的全局变量 theApp 有些类似，唯一不同的是在 ATL 中这个变量必须命名为 _Module。

下面是 stdafx.h 文件的开始部分：

```
// stdafx.h:
#define STRICT
#define VC_EXTRALEAN

#include <atlbase.h>       // 基本的 ATL 类
extern CComModule _Module;  // 全局_Module
#include <atlwin.h>        // ATL 窗口类
```

atlbase.h 已经包含最基本的 Window 编程的头文件，所以我们不需要在包含 windows.h，tchar.h 之类的头文件。在 CPP 文件中声明了_Module 变量：

```
// main.cpp:
CComModule _Module;
```

CComModule 含有程序的初始化和关闭函数，需要在 WinMain()中显示的调用，让我们从这里开始：

```
// main.cpp:
CComModule _Module;

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
                   LPSTR szCmdLine, int nCmdShow)
{
    _Module.Init(NULL, hInst);
    _Module.Term();
}
```

Init()的第一个参数只有 COM 的服务程序才有用，由于我们的 EXE 不含有 COM 对象，我们只需将 NULL 传递给 Init()就行了。 ATL 不提供自己的 WinMain()和类似 MFC 的消息泵，所以我们需要创建 CMyWindow 对象并添加消息泵才能使我们的程序运行。< br>

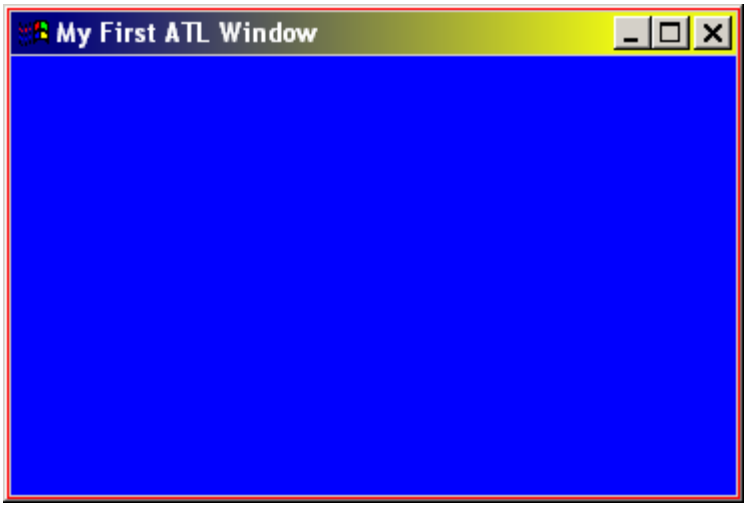
```
// main.cpp:
#include "MyWindow.h"
```

```
CComModule _Module;

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
                LPSTR szCmdLine, int nCmdShow)
{
    _Module.Init(NULL, hInst);

    CMyWindow wndMain;
    MSG msg;

    // Create & show our main window
    if ( NULL == wndMain.Create ( NULL, CWindow::rcDefault,
                            _T("My First ATL Window") ))
    {
        // Bad news, window creation failed
        return 1;
     }

    wndMain.ShowWindow(nCmdShow);
    wndMain.UpdateWindow();

    // Run the message loop
    while ( GetMessage(&msg, NULL, 0, 0) > 0 )
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    _Module.Term();
    return msg.wParam;
}
```

上面的代码唯一需要说明的是 CWindow::rcDefault，这是 CWindow 中的成员(静态数据成员)，数据类型是 RECT。和调用 CreateWindow() API 时使用 CW_USEDEFAULT 指定窗口的宽度和高度一样，ATL 使用 rcDefault 作为窗口的最初大小。

在 ATL 代码内部，ATL 使用了一些类似汇编语言的魔法将主窗口的句柄与相应的 CMyWindow 对象联系起来，在外部看来就是可以毫无问题的在线程之间传递 CWindow 对象，而 MFC 的 CWnd 却不能这样作。

这就是我们的窗口：



我得承认这确实没有什么激动人心的地方。我们将添加一个 About 菜单并显示一个对话框，主要是为它增加一些情趣。

ATL 中的对话框

我们前面提到过，ATL 有两个对话框类，我们的 About 对话框使用 CDialogImpl。生成一个新对话框和生成一个主窗口几乎一样，只有两点不同：

窗口的基类是 CDialogImpl 而不是 CWindowImpl。

你需要定义名称为 IDD 的公有成员用来保存对话框资源的 ID。

现在开始为 About 对话框定义一个新类：

```
class CAboutDlg : public CDialogImpl<CAboutDlg>
{
public:
    enum { IDD = IDD_ABOUT };

    BEGIN_MSG_MAP(CAboutDlg)
    END_MSG_MAP()
};
```

ATL 没有在内部实现对“OK”和“Cancel”两个按钮的响应处理，所以我们需要自己添加这些代码，如果用户用鼠标点击标题栏的关闭按钮， WM_CLOSE 的响应函数就会被调用。我们还需要处理 WM_INITDIALOG 消息，这样我们就能够在对话框出现时正确的设置键盘焦点，下面是完整的类定义和消息响应函数。

```
class CAboutDlg : public CDialogImpl<CAboutDlg>
{
public:
    enum { IDD = IDD_ABOUT };

    BEGIN_MSG_MAP(CAboutDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        MESSAGE_HANDLER(WM_CLOSE, OnClose)
        COMMAND_ID_HANDLER(IDOK, OnOKCancel)
        COMMAND_ID_HANDLER(IDCANCEL, OnOKCancel)
    END_MSG_MAP()
```

```
    LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        CenterWindow();
        return TRUE;    // let the system set the focus
    }

    LRESULT OnClose(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
        EndDialog(IDCANCEL);
        return 0;
    }

    LRESULT OnOKCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
    {
        EndDialog(wID);
        return 0;
    }
};
```

我使用一个消息响应函数同时处理“OK”和“Cancel”两个按钮的 WM_COMMAND 消息，因为命令响应函数的 wID 参数就已经指明了消息是来自“OK”按钮还是来自“Cancel”按钮。

显示对话框的方法与 MFC 相似，创建一个新对话框类的实例，然后调用 DoModal()方法。现在我们返回主窗口，添加一个带有 About 菜单项的菜单用来显示我们的对话框，这需要再添加两个消息响应函数，一个是响应 WM_CREATE，另一个是响应菜单的 IDC_ABOUT 命令。

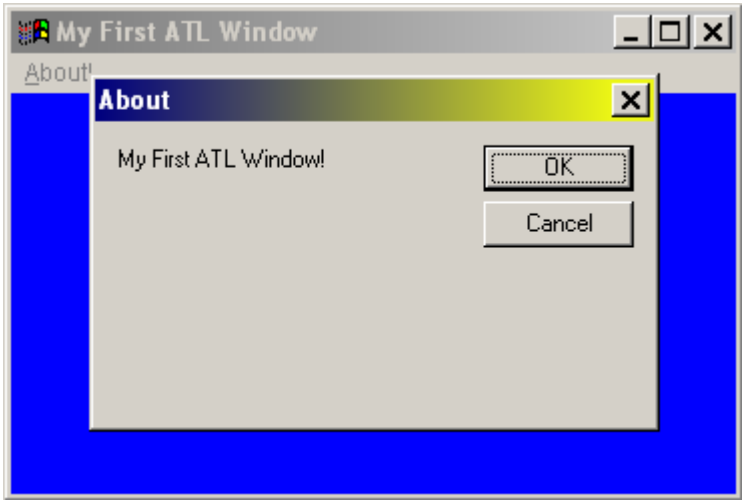
```
class    CMyWindow    :    public    CWindowImpl<CMyWindow,    CWindow,
CFrameWinTraits>,
                public CPaintBkgnd<CMyWindow,RGB(0,0,255)>
{
public:
    BEGIN_MSG_MAP(CMyWindow)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        COMMAND_ID_HANDLER(IDC_ABOUT, OnAbout)
        // ...
        CHAIN_MSG_MAP(CPaintBkgndBase)
    END_MSG_MAP()

    LRESULT OnCreate(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
    {
    HMENU hmenu = LoadMenu ( _Module.GetResourceInstance(),
                          MAKEINTRESOURCE(IDR_MENU1) );
```

```
        SetMenu ( hmenu );
        return 0;
    }

    LRESULT OnAbout(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
    {
    CAboutDlg dlg;

        dlg.DoModal();
        return 0;
    }
    // ...
};
```

在指定对话框的父窗口的方式上有些不同，MFC 是通过构造函数将父窗口的指针传递给对话框而在 ATL 中是将父窗口的指针作为 DoModal() 方法的第一个参数传递给对话框的，如果象上面的代码一样没有指定父窗口，ATL 会使用 GetActiveWindow() 得到的窗口(也就是我们的主框架窗口)作为对话框的父窗口。

对 LoadMenu() 方法的调用展示了 CComModule 的另一个方法－GetResourceInstance()，它返回你的 EXE 的 HINSTANCE 实例，和 MFC 的 AfxGetResourceHandle() 方法相似。(当然还有 CComModule::GetModuleInstance()，它相当于 MFC 的 AfxGetInstanceHandle()。)

这就是主窗口和对话框的显示效果：



我会继续讲 WTL，我保证!

我会继续讲 WTL 的，只是会在第二部分。我觉得既然这些文章是写给使用 MFC 的开发者的，所以有必要在投入 WTL 之前先介绍一些 ATL。如果你是第一次接触到 ATL，那现在你就可以尝试写一些小程序，处理消息和使用嵌入类，你也可以尝试用类向导支持消息映射链，使它能够自动添加消息响应。现在就开始，右键单击 CMyWindow 项，在弹出的上下文菜单中单击“Add Windows Message Handler”菜单项。

在第二部分，我将全面介绍基本的 WTL 窗口类和一个更好的消息映射宏。

第二部分　WTL 中的 GUI 基础类

对第二部分的介绍

好了，现在正式开始介绍 WTL！在这一部分我讲的内容包括生成一个基本的主窗口和 WTL 提供的一些友好的改进，比如 UI 界面的更新(如菜单上的选择标记)和更好的消息映射机制。为了更好地掌握本章的内容，你应该安装 WTL 并将 WTL 库的头文件目录添加到 VC 的搜索目录中，还要将 WTL 的应用程序生成向导复制到正确的位置。WTL 的发布版本中有文档具体介绍如何做这些设置，如果遇到困难可以查看这些文档。

WTL 总体印象

WTL 的类大致可以分为几种类型：

主框架窗口的实现- CFrameWindowImpl, CMDIFrameWindowImpl

控件的封装- CButton, CListViewCtrl

GDI 对象的封装- CDC, CMenu

一些特殊的界面特性 - CSplitterWindow, CUpdateUI, CDialogResize, CCustomDraw

实用的工具类和宏- CString, CRect, BEGIN_MSG_MAP_EX

本篇文章将深入地介绍框架窗口类，还将简要地讲一下有关的界面特性类和工具类，这些界面特性类和工具类中绝大多数都是独立的类，尽管有一些是嵌入类，例如：CDialogResize。

开始写 WTL 程序

如果你没有用 WTL 的应用程序生成向导也没关系(我将在后面介绍这个向导的用法)，WTL 的程序的代码结构很像 ATL 的程序，本章使用的例子代码有别于第一章的例子，主要是为了显示 WTL 的特性，没有什么实用价值。

这一节我们将在 WTL 生成的代码基础上添加代码，生成一个新的程序，程序主窗口的客户区显示当前的时间。stdafx.h 的代码如下：

```
#define STRICT
#define WIN32_LEAN_AND_MEAN
#define _WTL_USE_CSTRING

#include <atlbase.h>      // 基本的 ATL 类
#include <atlapp.h>       // 基本的 WTL 类
extern CAppModule _Module; // WTL 派生的 CComModule 版本
#include <atlwin.h>       // ATL 窗口类
#include <atlframe.h>     // WTL 主框架窗口类
#include <atlmisc.h>      // WTL 实用工具类，例如：CString
#include <atlcrack.h>     // WTL 增强的消息宏
```

atlapp.h 是你的工程中第一个包含的头文件，这个文件内定义了有关消息处理的类和 CAppModule，CAppModule 是从 CComModule 派生的类。如果你打算使用 CString 类，你需要手工定义_WTL_USE_CSTRING 标号，因为 CString 类是在 atlmisc.h 中定义的，而许多包含在 atlmisc.h 之前的头文件都会用到 CString，定义_WTL_USE_CSTRING 之后，atlapp.h 就会向前声明 CString 类，其他的头文件就知道 CString 类的存在，从而避免编译器为此大惊小怪。

接下来定义框架窗口。我们的 SDI 窗口是从 CFrameWindowImpl 派生的，在定义窗口类时使

用 DECLARE_FRAME_WND_CLASS 代替前面使用的 DECLARE_WND_CLASS。下面时 MyWindow.h 中窗口定义的开始部分：

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
   DECLARE_FRAME_WND_CLASS(_T("First WTL window"), IDR_MAINFRAME);

   BEGIN_MSG_MAP(CMyWindow)
      CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
   END_MSG_MAP()
};
```

DECLARE_FRAME_WND_CLASS 有两个参数，窗口类名(类名可以是 NULL，ATL 会替你生成一个类名)和资源 ID，创建窗口时 WTL 用这个 ID 装载图标，菜单和加速键表。我们还要象 CFrameWindowImpl 中的消息处理(例如 WM_SIZE 和 WM_DESTROY 消息)那样将消息链入窗口的消息中。

现在来看看 WinMain()函数，它和第一部分中的例子代码中的 WinMain()函数几乎一样，只是创建窗口部分的代码略微不同。

```
// main.cpp:
#include "stdafx.h"
#include "MyWindow.h"

CAppModule _Module;

int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow )
{
   _Module.Init ( NULL, hInstance );

CMyWindow wndMain;
MSG msg;

   // Create the main window
   if ( NULL == wndMain.CreateEx() )
      return 1;      // Window creation failed

   // Show the window
   wndMain.ShowWindow ( nCmdShow );
   wndMain.UpdateWindow();

   // Standard Win32 message loop
   while ( GetMessage ( &msg, NULL, 0, 0 ) > 0 )
      {
      TranslateMessage ( &msg );
      DispatchMessage ( &msg );
```

```
        }

    _Module.Term();
    return msg.wParam;
}
```

CFrameWindowImpl 中的 CreateEx()函数的参数使用了常用的默认值，所以我们不需要特别指定任何参数。正如前面介绍的，CFrameWindowImpl 会处理资源的装载，你只需要使用 IDR_MAINFRAME 作为 ID 定义你的资源就行了(译者注：主要是图标，菜单和加速键表)，你也可以直接使用本章的例子代码。

如果你现在就运行程序，你会看到主框架窗口，事实上它没有做任何事情。我们需要手工添加一些消息处理，所以现在是介绍 WTL 的消息映射宏的最佳时间。

WTL 对消息映射的增强

将 Win32 API 通过消息传递过来的 WPARAM 和 LPARAM 数据还原出来是一件麻烦的事情并且很容易出错，不幸得是 ATL 并没有为我们提供更多的帮助，我们仍然需要从消息中还原这些数据，当然 WM_COMMAND 和 WM_NOTIFY 消息除外。但是 WTL 的出现拯救了这一切！

WTL 的增强消息映射宏定义在 atlcrack.h 中。(这个名字来源于“消息解密者”，是一个与 windowsx.h 的宏所使用的相同术语)首先将 BEGIN_MSG_MAP 改为 BEGIN_MSG_MAP_EX，带_EX 的版本产生“解密”消息的代码。

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()
};
```

对于我们的时钟程序，我们需要处理 WM_CREATE 消息来设置定时器，WTL 的消息处理使用 MSG_作为前缀，后面是消息名称，例如 MSG_WM_CREATE。这些宏只是代表消息响应处理的名称，现在我们来添加对 WM_CREATE 消息的响应：

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    // OnCreate(...) ?
};
```

WTL 的消息响应处理看起来有点象 MFC，每一个处理函数根据消息传递的参数不同也有不同的原型。由于我们没有向导自动添加消息响应，所以我们需要自己查找正确的消息处理函数。幸运的是 VC 可以帮我们的忙，将鼠标光标移到“MSG_WM_CREATE”宏的文字上按 F12 键就可以来到这个宏的定义代码处。如果是第一次使用这个功能，VC 会要求从新编译全部文件以建立浏览信息数据库(browse info database)，建立了这个数据库之后，VC 会打开 atlcrack.h 并将代码定位到 MSG_WM_CREATE 的定义位置：

```
#define MSG_WM_CREATE(func) \
    if (uMsg == WM_CREATE) \
    { \
        SetMsgHandled(TRUE); \
        lResult = (LRESULT)func((LPCREATESTRUCT)lParam); \
        if(IsMsgHandled()) \
            return TRUE; \
    }
```

标记为红色的那一行非常重要，就是在这里调用实际的消息响应函数，他告诉我们消息响应函数有一个 LPCREATESTRUCT 类型的参数，返回值的类型是 LRESULT。请注意这里没有 ATL 的宏所用的 bHandled 参数，SetMsgHandled()函数代替了这个参数，我会对此作些简要的介绍。

现在为我们的窗口类添加 OnCreate()响应函数：

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    LRESULT OnCreate(LPCREATESTRUCT lpcs)
    {
        SetTimer ( 1, 1000 );
        SetMsgHandled(false);
        return 0;
    }
};
```

CFrameWindowImpl 是直接从 CWindow 类派生的，所以它继承了所有 CWindow 类的方法，如 SetTimer()。这使得对窗口 API 的调用有点象 MFC 的代码，只是 MFC 使用 CWnd 类包装这些 API。

我们使用 SetTimer()函数创建一个定时器，它每隔一秒钟(1000 毫秒)触发一次。由于我们需要让 CFrameWindowImpl 也处理 WM_CREATE 消息，所以我们调用 SetMsgHandled(false)，让消息通过 CHAIN_MSG_MAP 宏链入基类，这个调用代替了 ATL 宏使用的 bHandled 参数。(即使 CFrameWindowImpl 类不需要处理 WM_CREATE 消息，调用 SetMsgHandled (false)让消息流入基类是个好的习惯，因为这样我们就不必总是记着哪个消息需要基类处理那些消息不需要基类处理，这和 VC 的类向导产生的代码相似，多数的派生类的消息处理函数的开始或结尾都会调用基类的消息处理函数)

为了能够停止定时器我们还需要响应 WM_DESTROY 消息，添加消息响应的过程和前面一样，MSG_WM_DESTROY 宏的定义是这样的：

```
#define MSG_WM_DESTROY(func) \
    if (uMsg == WM_DESTROY) \
    { \
        SetMsgHandled(TRUE); \
```

```
        func(); \
        lResult = 0; \
        if(IsMsgHandled()) \
            return TRUE; \
    }
```

OnDestroy()函数没有参数也没有返回值，CFrameWindowImpl 也要处理 WM_DESTROY 消息，所以还要调用 SetMsgHandled(false)：

```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()



    void OnDestroy()
    {
        KillTimer(1);
        SetMsgHandled(false);
    }
};
```

接下来是响应 WM_TIMER 消息的处理函数，它每秒钟被调用一次。你应该知道怎样使用 F12 键的窍门了，所以我直接给出响应函数的代码：

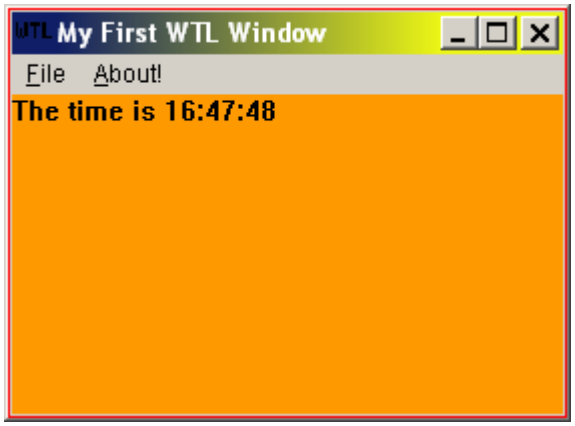
```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    void OnTimer ( UINT uTimerID, TIMERPROC pTimerProc )
    {
        if ( 1 != uTimerID )
            SetMsgHandled(false);
        else
            RedrawWindow();
    }
};
```

这个响应函数只是在每次定时器触发时重画窗口的客户区。最后我们要响应 WM_ERASEBKGND 消息，在窗口客户区的左上角显示当前的时间。

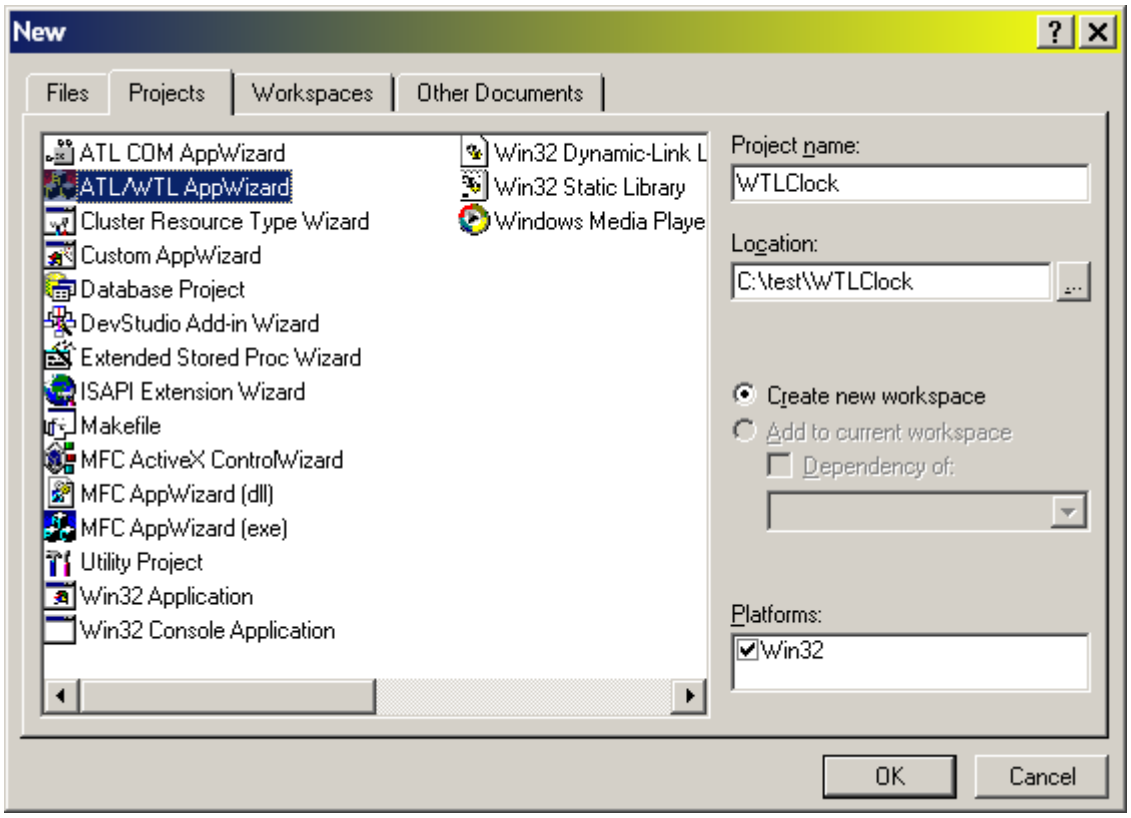
```
class CMyWindow : public CFrameWindowImpl<CMyWindow>
{
public:
    BEGIN_MSG_MAP_EX(CMyWindow)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        MSG_WM_ERASEBKGND(OnEraseBkgnd)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMyWindow>)
    END_MSG_MAP()

    LRESULT OnEraseBkgnd ( HDC hdc )
    {
    CDCHandle  dc(hdc);
    CRect     rc;
    SYSTEMTIME st;
    CString    sTime;

        // Get our window's client area.
        GetClientRect ( rc );

        // Build the string to show in the window.
        GetLocalTime ( &st );
        sTime.Format ( _T("The time is %d:%02d:%02d"),
                    st.wHour, st.wMinute, st.wSecond );

        // Set up the DC and draw the text.
        dc.SaveDC();

        dc.SetBkColor ( RGB(255,153,0);
        dc.SetTextColor ( RGB(0,0,0) );
        dc.ExtTextOut ( 0, 0, ETO_OPAQUE, rc, sTime,
                     sTime.GetLength(), NULL );

        // Restore the DC.
        dc.RestoreDC(-1);
        return 1;    // We erased the background (ExtTextOut did it)
    }
};
```

这个消息处理函数不仅使用了 CRect 和 CString 类，还使用了一个 GDI 包装类 CDCHandle。对于 CString 类我想说的是它等同与 MFC 的 CString 类，我在后面的文章中还会介绍这些包装类，现在你只需要知道 CDCHandle 是对 HDC 的简单封装就行了，使用方法与 MFC 的 CDC

类相似，只是 CDCHandle 的实例在超出作用域后不会销毁它所操作的设备上下文。
所有的工作完成了，现在看看我们的窗口是什么样子：



例子代码中还使用了 WM_COMMAND 响应菜单消息，在这里我不作介绍，但是你可以查看例子代码，看看 WTL 的 COMMAND_ID_HANDLER_EX 宏是如何工作的。
从 WTL 的应用程序生成向导能得到什么
WTL 的发布版本附带一个很棒的应用程序生成向导，让我们以一个 SDI 应用为例看看它有什么特性。
使用向导的整个过程
在 VC 的 IDE 环境下单击 File|New 菜单，从列表中选择 ATL/WTL AppWizard，我们要重写时钟程序，所以用 WTLClock 作为项目的名字：



在下一页你可以选择项目的类型，SDI，MDI 或者是基于对话框的应用，当然还有其它选项，如下图所示设置这些选项，然后点击“下一步”：

在最后一页你可以选择是否使用 toolbar，rebar 和 status bar，为了简单起见，取消这些选项并单击“结束”。

查看生成的代码

向导完成后，在生成的代码中有三个类：CMainFrame, CAboutDlg, 和 CWTLClockView，从名字上就可以猜出这些类的作用。虽然也有一个是视图类，但它仅仅是从 CWindowImpl 派生出来的一个简单的窗口类，没有象 MFC 那样的文档/视图结构。

还有一个_tWinMain()函数，它先初始化 COM 环境，公用控件和_Module，然后调用全局函数 Run()。Run()函数创建主窗口并开始消息循环，Run()调用 CMessageLoop::Run()，消息泵实际上是位于CMessageLoop::Run()内，我将在下一个章节介绍CMessageLoop 的更多细节。

CAboutDlg 是 CDialogImpl 的派生类，它对应于 ID IDD_ABOUTBOX 资源，我在第一部分已经介绍过对话框，所以你应该能看懂 CAboutDlg 的代码。

CWTLClockView 是我们的程序的视图类，它的作用和 MFC 的视图类一样，没有标题栏，覆盖整个主窗口的客户区。 CWTLClockView 类有一个 PreTranslateMessage()函数，也和 MFC 中的同名函数作用相同，还有一个 WM_PAINT 的消息响应函数。这两个函数都没有什么特别之处，只是我们会填写 OnPaint()函数来显示时间。

最后是我们的 CMainFrame 类，它有许多有趣的新东西，这是这个类的定义缩略版本：

```
class CMainFrame : public CFrameWindowImpl<CMainFrame>,
                public CUpdateUI<CMainFrame>,
                public CMessageFilter,
                public CIdleHandler
{
public:
    DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)

    CWTLClockView m_view;

    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL OnIdle();

    BEGIN_UPDATE_UI_MAP(CMainFrame)
    END_UPDATE_UI_MAP()

    BEGIN_MSG_MAP(CMainFrame)
        // ...
        CHAIN_MSG_MAP(CUpdateUI<CMainFrame>)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
    END_MSG_MAP()
};
```

CMessageFilter 是一个嵌入类，它提供 PreTranslateMessage()函数，CIdleHandler 也是一个嵌入类，它提供了 OnIdle()函数。CMessageLoop, CIdleHandler 和 CUpdateUI 三个类互相协同完成界面元素的状态更新(UI update)，就像 MFC 中的 ON_UPDATE_COMMAND_UI 宏一样。

CMainFrame::OnCreate()中创建了视图窗口并保存这个窗口的句柄，当主窗口改变大小时视图窗口的大小也会随之改变。 OnCreate()函数还将 CMainFrame 对象添加到由 CAppModule 维持的消息过滤器队列和空闲处理队列，我将在稍后介绍这些。

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                             LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
    m_hWndClient = m_view.Create(m_hWnd, rcDefault, NULL, |
```

```
                                    WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
                                      WS_CLIPCHILDREN, WS_EX_CLIENTEDGE);

    // register object for message filtering and idle updates
    CMessageLoop* pLoop = _Module.GetMessageLoop();
    pLoop->AddMessageFilter(this);
    pLoop->AddIdleHandler(this);

    return 0;
}
```

m_hWndClient 是 CFrameWindowImpl 对象的一个成员变量，当主窗口大小改变时此窗口的大小也将改变。

在生成的 CMainFrame 中还添加了对 File|New, File|Exit, 和 Help|About 菜单消息的处理。我们的时钟程序不需要这些默认的菜单项，但是现在将它们留在代码中也没有害处。现在可以编译并运行向导生成的代码，不过这个程序确实没有什么用处。如果你感兴趣的话可以深入 CMainFrame::CreateEx()函数的内部看看主窗口和它的资源是如何被加载和创建得。

我们的下一步 WTL 之旅是 CMessageLoop，它掌管消息泵和空闲处理。

CMessageLoop 的内部实现

CMessageLoop 为我们的应用程序提供一个消息泵，除了一个标准的 DispatchMessage/TranslateMessage 循环外，它还通过调用 PreTranslateMessage()函数实现了消息过滤机制，通过调用 OnIdle()实现了空闲处理功能。下面是 Run ()函数的伪代码:

```
int Run()
{
MSG msg;

    for(;;)
        {
        while ( !PeekMessage(&msg) )
            DoIdleProcessing();

        if ( 0 == GetMessage(&msg) )
            break;   // WM_QUIT retrieved from the queue

        if ( !PreTranslateMessage(&msg) )
            {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            }
        }

    return msg.wParam;
}
```

那些需要过滤消息的类只需要象 CMainFrame::OnCreate()函数那样调用 CMessageLoop:: AddMessageFilter()函数就行了，CMessageLoop 就会知道该调用那个 PreTranslateMessage()函数，同样，如果需要空闲处理就调用 CMessageLoop::AddIdleHandler()函数。

需要注意得是在这个消息循环中没有调用 TranslateAccelerator()或 IsDialogMessage()函数，因为 CFrameWindowImpl 在这之前已经做了处理，但是如果你在程序中使用了非模式对话框，那你就需要在 CMainFrame:: PreTranslateMessage()函数中添加对 IsDialogMessage()函数的调用。

CFrameWindowImpl 的内部实现

CFrameWindowImpl 和它的基类 CFrameWindowImplBase 提供了对 toolbars，rebars，status bars，工具条按钮的工具提示和菜单项的掠过式帮助，这些也是 MFC 的 CFrameWnd 类的基本特征。我会逐步介绍这些特征，完整的讨论 CFrameWindowImpl 类需要再写两篇文章，但是现在看看 CFrameWindowImpl 是如何处理 WM_SIZE 和它的客户区就足够了。需要记住一点前面提到的东西，m_hWndClient 是 CFrameWindowImplBase 类的成员变量，它存储主窗口内的“视图”窗口的句柄。

CFrameWindowImpl 类处理了 WM_SIZE 消息：

```
LRESULT OnSize(UINT /*uMsg*/, WPARAM wParam, LPARAM /*lParam*/, BOOL&
bHandled)
{
   if(wParam != SIZE_MINIMIZED)
   {
      T* pT = static_cast<T*>(this);
      pT->UpdateLayout();
   }

   bHandled = FALSE;
   return 1;
}
```

它首先检查窗口是否最小化，如果不是就调用 UpdateLayout()，下面是 UpdateLayout()：

```
void UpdateLayout(BOOL bResizeBars = TRUE)
{
RECT rect;

   GetClientRect(&rect);

   // position bars and offset their dimensions
   UpdateBarsPosition(rect, bResizeBars);

   // resize client window
   if(m_hWndClient != NULL)
      ::SetWindowPos(m_hWndClient, NULL, rect.left, rect.top,
         rect.right - rect.left, rect.bottom - rect.top,
         SWP_NOZORDER | SWP_NOACTIVATE);
}
```

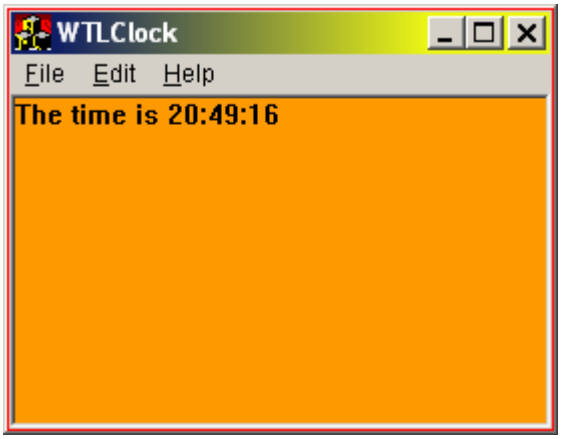
注意这些代码是如何使用 m_hWndClient 得，既然 m_hWndClient 是一般窗口的句柄，它就可能是任何窗口，对这个窗口的类型没有限制。这一点不像 MFC，MFC 在很多情况下需要 CView 的派生类(例如分隔窗口类)。如果你回过头看看 CMainFrame::OnCreate ()就会看到它创建了一个视图窗口并赋值给 m_hWndClient，由 m_hWndClient 确保视图窗口被设置为正确的大小。

回到前面的时钟程序

现在我们已经看到了主窗口类的一些细节，现在回到我们的时钟程序。视图窗口用来响应定时器消息并负责显示时钟，就像前面的 CMyWindow 类。下面是这个类的部分定义：

```
class CWTLClockView : public CWindowImpl<CWTLClockView>
{
public:
    DECLARE_WND_CLASS(NULL)

    BOOL PreTranslateMessage(MSG* pMsg);

    BEGIN_MSG_MAP_EX(CWTLClockView)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        MSG_WM_CREATE(OnCreate)
        MSG_WM_DESTROY(OnDestroy)
        MSG_WM_TIMER(OnTimer)
        MSG_WM_ERASEBKGND(OnEraseBkgnd)
    END_MSG_MAP()
};
```

使用 BEGIN_MSG_MAP_EX 代替 BEGIN_MSG_MAP 后，ATL 的消息映射宏可以和 WTL 的宏混合使用，前面的例子在 OnEraseBkgnd()中显示(画)时钟，现在被被搬到了 OnPaint()中。新窗口看起来是这个样子的：

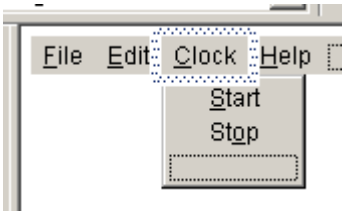


最后为我们的程序添加 UI updating 功能，为了演示这些用法，我们为窗口添加 Start 菜单和 Stop 菜单用于开始和停止时钟，Start 菜单和 Stop 菜单将被适当的设置为可用和不可用。

界面元素的自动更新(UI Updating)

空闲时间的界面更新是几件事情协同工作的结果：CMessageLoop 对象，嵌入类 CIdleHandler 和 CUpdateUI，CMainFrame 类继承了这两个嵌入类，当然还有 CMainFrame 类中的 UPDATE_UI_MAP 宏。CUpdateUI 能够操作 5 种不同的界面元素：顶

级菜单项(就是菜单条本身)，弹出式菜单的菜单项，工具条按钮，状态条的格子和子窗口(如对话框中的控件)。每一种界面元素都对应 CUpdateUIBase 类的一个常量：
菜单条项: UPDUI_MENUBAR
弹出式菜单项: UPDUI_MENUPOPUP
工具条按钮: UPDUI_TOOLBAR
状态条格子: UPDUI_STATUSBAR
子窗口: UPDUI_CHILDWINDOW
CUpdateUI 可以设置 enabled 状态，checked 状态和文本(当然不是所有的界面元素都支持所有状态，如果一个子窗口是编辑框它就不能被 check)。菜单项可以被设置为默认状态，这样它的文字会被加重显示。
要使用 UI updating 需要做四件事：
主窗口需要继承 CUpdateUI 和 CIdleHandler
将 CMainFrame 的消息链入 CUpdateUI
将主窗口添加到模块的空闲处理队列
在主窗口中添加 UPDATE_UI_MAP 宏
向导生成的代码已经为我们做了三件事，现在我们只需要决定那个菜单项需要更新和他们是么时候可用什么时候不可用。
添加控制时钟的新菜单项
在菜单条添加一个 Clock 菜单，它有两个菜单项：IDC_START and IDC_STOP:



然后在 UPDATE_UI_MAP 宏中为每个菜单项添加一个入口:

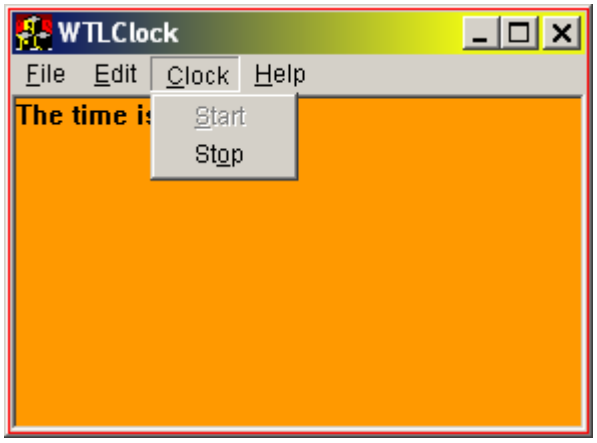
```
class CMainFrame : public ...
{
public:
    // ...
    BEGIN_UPDATE_UI_MAP(CMainFrame)
        UPDATE_ELEMENT(IDC_START, UPDUI_MENUPOPUP)
        UPDATE_ELEMENT(IDC_STOP, UPDUI_MENUPOPUP)
    END_UPDATE_UI_MAP()
    // ...
};
```

我们只需要调用 CUpdateUI::UIEnable()就可以改变这两个菜单项的任意一个的使能状态时。UIEnable()有两个参数，一个是界面元素的 ID，另一个是标志界面元素是否可用的 bool 型变量(true 表示可用,false 表示不可用)。

这套体系比 MFC 的 ON_UPDATE_COMMAND_UI 体系笨拙一些，在 MFC 中我们只需编写处理函数，由 MFC 选择界面元素的显示状态，在 WTL 中我们需要告诉 WTL 界面元素的状态在何时改变。当然，这两个库都是在菜单将要显示的时候才应用菜单状态的改变。
调用 UIEnable()
现在返回到 OnCreate()函数看看是如何设置 Clock 菜单的初始状态。

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                             LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
   m_hWndClient = m_view.Create(...);

   // register object for message filtering and idle updates
   // [omitted for clarity]

   // Set the initial state of the Clock menu items:
   UIEnable ( IDC_START, false );
   UIEnable ( IDC_STOP, true );

   return 0;
}
```

我们的程序开始时 Clock 菜单是这样的：



CMainFrame 现在需要处理两个新菜单项，在视图类调用它们开始和停止时钟时处理函数需要翻转这两个菜单项的状态。这是 MFC 的内建消息处理无法想象的地方之一。在 MFC 的程序中，所有的界面更新和命令消息处理必须完整的放在视图类中，但是在 WTL 中，主窗口类和视图类通过某种方式沟通；菜单由主窗口拥有，主窗口获得这些菜单消息并做相应的处理，要么响应这些消息，要么发送给视图类。

这种沟通是通过 PreTranslateMessage()完成的，当然 CMainFrame 仍然要调用 UIEnable()。CMainFrame 可以将 this 指针传递给视图类，这样视图类也可以通过这个指针调用 UIEnable()。在这个例子中我选择的这种解决方案导致主窗口和视图成为紧密耦合体，但是我发现这很容易理解(和解释!)。

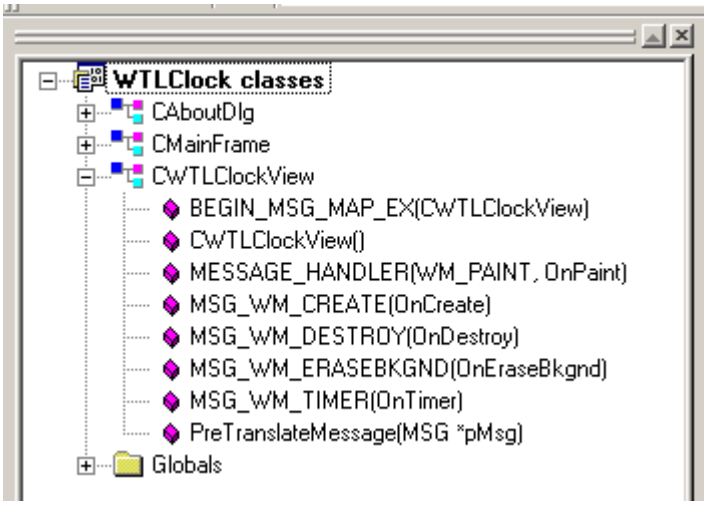
```
class CMainFrame : public ...
{
public:
   BEGIN_MSG_MAP_EX(CMainFrame)
      // ...
      COMMAND_ID_HANDLER_EX(IDC_START, OnStart)
      COMMAND_ID_HANDLER_EX(IDC_STOP, OnStop)
   END_MSG_MAP()
```

```
    // ...
    void OnStart(UINT uCode, int nID, HWND hwndCtrl);
    void OnStop(UINT uCode, int nID, HWND hwndCtrl);
};

void CMainFrame::OnStart(UINT uCode, int nID, HWND hwndCtrl)
{
    // Enable Stop and disable Start
    UIEnable ( IDC_START, false );
    UIEnable ( IDC_STOP, true );

    // Tell the view to start its clock.
    m_view.StartClock();
}

void CMainFrame::OnStop(UINT uCode, int nID, HWND hwndCtrl)
{
    // Enable Start and disable Stop
    UIEnable ( IDC_START, true );
    UIEnable ( IDC_STOP, false );

    // Tell the view to stop its clock.
    m_view.StopClock();
}
```

每个处理函数都更新 Clock 菜单，然后在视图类中调用一个方法，选择在视图类中使用是因为时钟是由视图类控制得。StartClock() 和 StopClock()得代码没有列出，但可以在这个工程得例子代码中找到它们。

消息映射链中最后需要注意的地方

如果你使用 VC 6，你会注意到将 BEGIN_MSG_MAP 改为 BEGIN_MSG_MAP_EX 后 ClassView 显得有些杂乱无章：



出现这种情况是因为 ClassView 不能解释 BEGIN_MSG_MAP_EX 宏，它以为所有得 WTL 消息映射宏是函数定义。你可以将宏改回为 BEGIN_MSG_MAP 并在 stdafx.h 文件得结尾处添加

这两行代码来解决这个问题：

```
#undef BEGIN_MSG_MAP
#define BEGIN_MSG_MAP(x) BEGIN_MSG_MAP_EX(x)
```

下一站，1995

我们现在只是掀起了 WTL 的一角，在下一篇文章我会为我们的时钟程序添加一些 Windows 95 的界面标准，比如工具条和状态条，同时体验一下 CUpdateUI 的新东西。例如试着用 UISetCheck()代替 UIEnable()，看看菜单项会有什么变化。

第三部分　工具栏和状态栏

对第三部分的介绍

自从作为 Windows 95 的通用控件出现以来，工具条和状态条就变成了很普遍的事物。由于 MFC 支持浮动的工具条从而使它们更受欢迎。随着通用控件的更新，Rebars(最初 被称为 Coollbar)使得工具条有了另一种展示方式。在第三部分，我将介绍 WTL 对这些控制条的支持和如何在你的程序中使用它们。

主窗口的工具条和状态条

CFrameWindowImpl 有三个 HWND 类型的成员变量在窗口创建时被初始化，我们已经见过 m_hWndClient，它是填充主窗口客户区的“视图”窗口的句柄，现在我们遇到了另外两个：

m_hWndToolBar: 工具条或 Rebar 的窗口句柄

m_hWndStatusBar: 状态条的窗口句柄

CFrameWindowImpl 只支持一个工具条，也没有像 MFC 那样的可多点停靠的工具条，如果你想使用多个工具条又不想修改 CFrameWindowImpl 的内部代码，你就需要使用 Rebar。我将介绍它们二者并演示如何使用应用程序向导添加工具条和 Rebar。< br>

CFrameWindowImpl::OnSize() 消息响应函数调用了 UpdateLayout()，UpdateLayout()做两件事： 从新定位所有控制条和改变视图窗口的大小使之填充整个客户区。实际工作是由 UpdateBarsPosition()完成的，UpdateLayout ()只是调用了该函数。实现的代码相当简单，向工具条和状态条发送 WM_SIZE 消息，由这些控制条的默认窗口处理过程将它们定位到主窗口的顶部或底部。

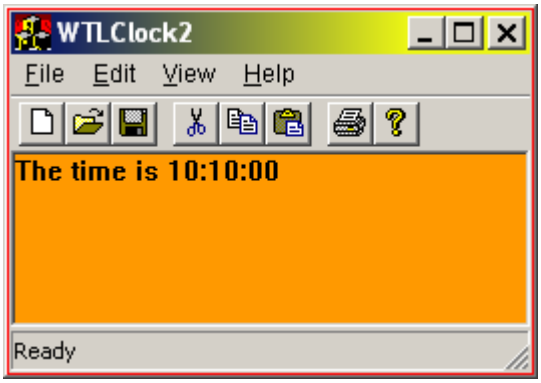
当你告诉应用程序向导给你的窗口添加工具条和状态条时，向导就在 CMainFrame::OnCreate()中添加了创建它们的代码。现在我们来看看这些代码，当然是为了再写一个时钟程序。

向导为工具条和状态条生成得代码

我们将开始一个新的工程，让向导为主窗口创建工具条和状态条。首先创建一个名为 WTLClock2 的新工程，在向导的第一页，选 SDI 并使“生成 CPP 文件”检查框被选中：

在第二页，取消 Rebar 使向导仅仅创建一个普通的工具条：

从第二部分的程序中复制相应的代码，新程序看起来是这样的：



CMainFraCMainFrame 如何创建工具条和状态条

在这个例子中，向导向 CMainFrame::OnCreate()函数添加了更多的代码，这些代码的作用就是创建控制条并通知 CUpdateUI 更新工具条上的按钮。

```
LRESULT CMainFrame::OnCreate(UINT /*uMsg*/, WPARAM /*wParam*/,
                             LPARAM /*lParam*/, BOOL& /*bHandled*/)
{
   CreateSimpleToolBar();
   CreateSimpleStatusBar();

   m_hWndClient = m_view.Create(...);

// ...

   // register object for message filtering and idle updates
   CMessageLoop* pLoop = _Module.GetMessageLoop();
   ATLASSERT(pLoop != NULL);
   pLoop->AddMessageFilter(this);
   pLoop->AddIdleHandler(this);

   return 0;
}
```

这是新添加的代码的开始部分，CFrameWindowImpl::CreateSimpleToolBar()函数使用资源 IDR_MAINFRAME 创建工具条并将其句柄赋值给 m_hWndToolBar，下面是 CreateSimpleToolBar()函数的代码：

```
BOOL CFrameWindowImpl::CreateSimpleToolBar(
   UINT nResourceID = 0,
   DWORD dwStyle = ATL_SIMPLE_TOOLBAR_STYLE,
   UINT nID = ATL_IDW_TOOLBAR)
{
   ATLASSERT(!::IsWindow(m_hWndToolBar));

   if(nResourceID == 0)
      nResourceID = T::GetWndClassInfo().m_uCommonResourceID;
```

```
    m_hWndToolBar = T::CreateSimpleToolBarCtrl(m_hWnd, nResourceID,
TRUE,
                                        dwStyle, nID);
    return (m_hWndToolBar != NULL);
}
```

参数：

nResourceID

工具条资源得 ID。如果使用默认值 0 作为参数，程序将使用 DECLARE_FRAME_WND_CLASS 宏指定得资源，这里使用的 IDR_MAINFRAME 是向导生成的代码。

dwStyle

工具条的类型或样式。默认值 ATL_SIMPLE_TOOLBAR_STYLE 被定义为 TBSTYLE_TOOLTIPS，子窗口和可见三种风格的结合，这使得鼠标移到按钮上时工具条会弹出工具提示。

nID

工具条的窗口 ID，通常都会使用默认值。

CreateSimpleToolBar()首先检查是否已经创建了一个工具条，然后调用 CreateSimpleToolBarCtrl()函数创建工具条控制，CreateSimpleToolBarCtrl()返回的工具条控制句柄保存在 m_hWndToolBar 中。CreateSimpleToolBarCtrl()负责读出资源并创建相应的工具条按钮，然后返回工具条窗口的句柄。这部分的代码相当长，我不在这里做 具体介绍，如果你对此感兴趣得话何以在 atlframe.h 中找到这些代码。

OnCreate()函数接下来会调用 CFrameWindowImpl::CreateSimpleStatusBar()函数，此函数创建状态条并将句柄存在 m_hWndStatusBar，下面是该函数的代码：

```
BOOL CFrameWindowImpl::CreateSimpleStatusBar(
    UINT nTextID = ATL_IDS_IDLEMESSAGE,
    DWORD dwStyle = ... SBARS_SIZEGRIP,
    UINT nID = ATL_IDW_STATUS_BAR)
{
    TCHAR szText[128];    // max text lentgth is 127 for status bars
    szText[0] = 0;
    ::LoadString(_Module.GetResourceInstance(), nTextID, szText, 128);
    return CreateSimpleStatusBar(szText, dwStyle, nID);
}
```

显示在状态条的文字是从字符串资源中装载的，这个函数的参数是：

nTextID

用于在状态条上显示的字符串的资源 ID，向导生成的 ATL_IDS_IDLEMESSAGE 对应的字符串是“Ready”。

dwStyle

状态条的样式。默认值包含了 SBARS_SIZEGRIP 风格，这使得状态条的右下角会显示一个改变窗口大小的标志。

nID

状态条的窗口 ID，通常都会使用默认值。

CreateSimpleStatusBar()调用另外一个重载函数创建状态条：

```
BOOL CFrameWindowImpl::CreateSimpleStatusBar(
    LPCTSTR lpstrText,
```

```
    DWORD dwStyle = ... SBARS_SIZEGRIP,
    UINT nID = ATL_IDW_STATUS_BAR)
{
    ATLASSERT(!::IsWindow(m_hWndStatusBar));
    m_hWndStatusBar = ::CreateStatusWindow(dwStyle, lpstrText, m_hWnd,
nID);
    return (m_hWndStatusBar != NULL);
}
```

这个重载的版本首先检查是否已经创建了状态条，然后调用 CreateStatusWindow()创建状态条，状态条的句柄存放在 m_hWndStatusBar 中。

显示和隐藏工具条和状态条

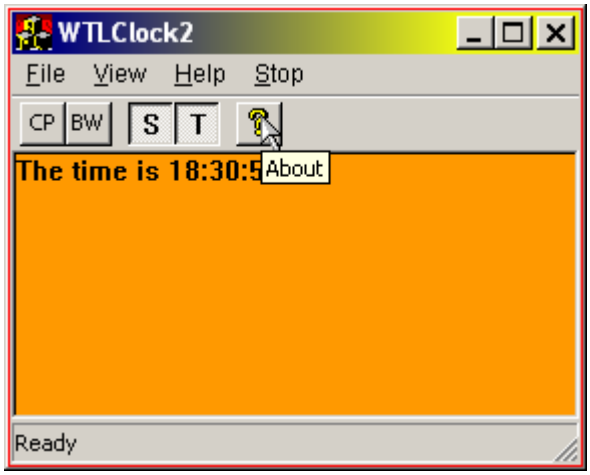
CMainFrame 类也有一个视图菜单，它有两个命令：显示/隐藏工具条和状态条，它们的 ID 是 ID_VIEW_TOOLBAR 和 ID_VIEW_STATUS_BAR。CMainFrame 类有这两个命令的响应函数，分别显示和隐藏相应的控制条，下面是 OnViewToolBar ()函数的代码：

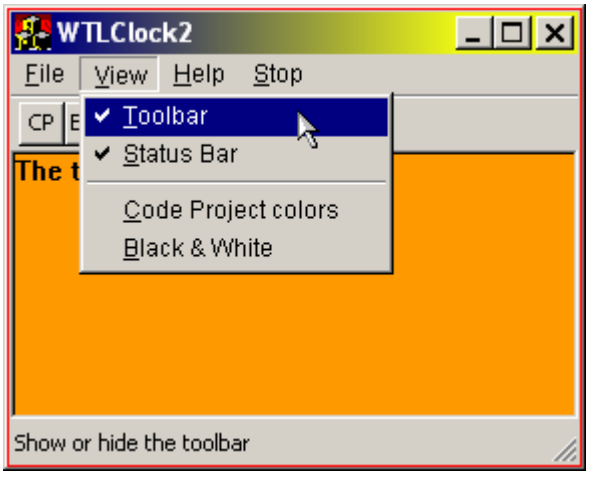
```
LRESULT CMainFrame::OnViewToolBar(WORD /*wNotifyCode*/, WORD /*wID*/,
                                  HWND /*hWndCtl*/, BOOL& /*bHandled*/)
{
    BOOL bVisible = !::IsWindowVisible(m_hWndToolBar);
    ::ShowWindow(m_hWndToolBar, bVisible ? SW_SHOWNOACTIVATE : SW_HIDE);
    UISetCheck(ID_VIEW_TOOLBAR, bVisible);
    UpdateLayout();
    return 0;
}
```

这些代码翻转控制条的显示状态，相应的翻转 View|Toolbar 菜单上的检查标记，然后调用 UpdateLayout()重新定位控制条并改变视图窗口的大小。

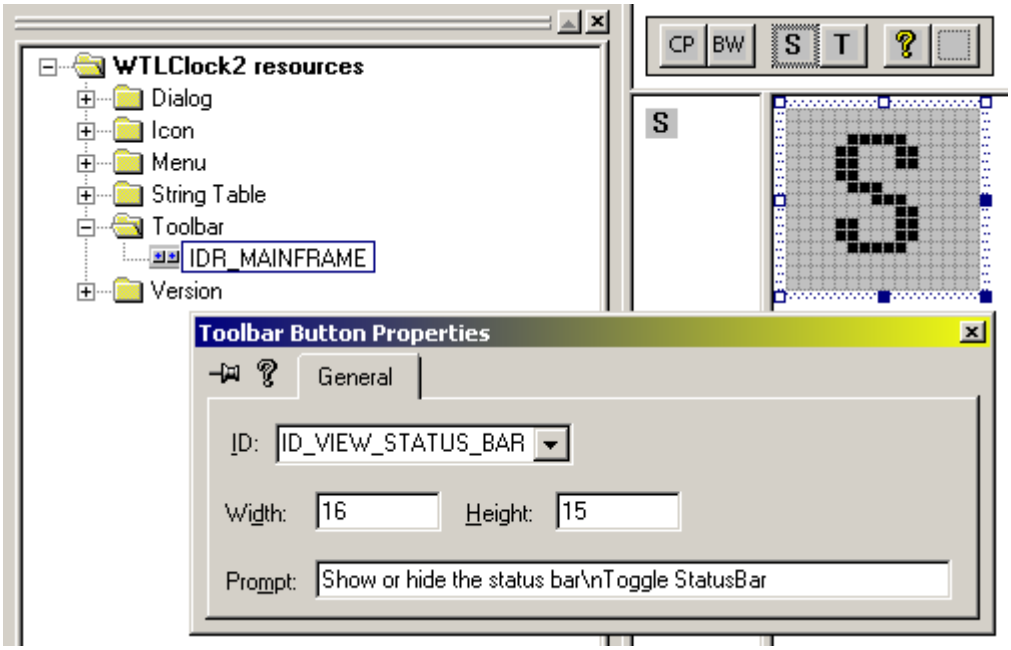
工具条和状态条的内在特征

MFC 的框架提供了很多好的特性，例如工具条按钮的工具提示和菜单项的掠过式帮助。WTL 中相对应的功能实现在 CFrameWindowImpl 类中。下面的屏幕截图显示了工具提示和掠过式帮助。





CFrameWindowImplBase 类有两个消息相应函数用来实现这些功能，OnMenuSelect()处理 WM_MENUSELECT 消 息，它像 MFC 那样查找掠过式帮助的字符串：首先装载与菜单资源 ID 相同的字符串资源，在字符串中查找 \n 字符，使用\n 之前的内容作为掠过帮助的内容。OnToolTipTextA() 和 OnToolTipTextW() 函数分别响应 TTN_GETDISPINFOA 消息和 TTN_GETDISPINFOW 消息，提供工具条按钮的工具提示。这两个处理函数和 OnMenuSelect ()函数一样装载相应的字符串，只是使用\n 后面的字符串。(边注：OnMenuSelect()和 OnToolTipTextA()函数对于 DBCS 字 符是不安全的，因为它在查找 \n 字符时没有检查 DBCS 字符串的头部和尾部)下面是工具条及其关联的帮助字符串的例子：



创建不同样式的工具条

如果你不喜欢在工具条上显示 3D 按钮(尽管从可用性观点来看平面的界面元素是件糟糕的事情),你可以通过改变 CreateSimpleToolBar()函数的参数来改变工具条的样式。例如，你可以在 CMainFrame::OnCreate()使用如下代码创 建一个 IE 风格的工具条：

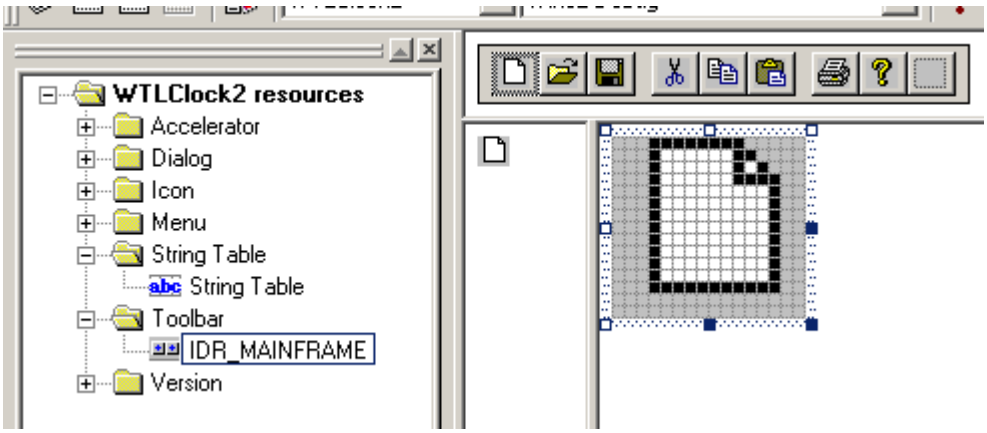
```
CreateSimpleToolBar ( 0, ATL_SIMPLE_TOOLBAR_STYLE |
                         TBSTYLE_FLAT | TBSTYLE_LIST );
```

如果你使用向导为你的程序添加了 manifest 文件，它就会在 Windows XP 系统上使用 6.0 版的通用控件，你不能选择按钮的类型，工具条会自动使用平面按钮即使你创建工具条时没有

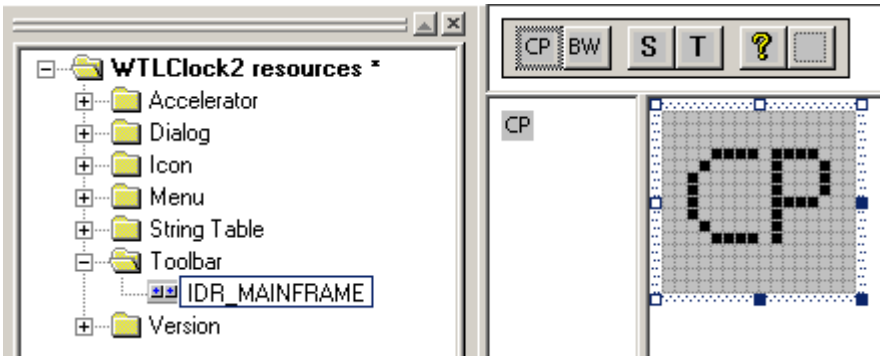
添加 TBSTYLE_FLAT 风格。

工具条编辑器

正如我们前面所见，向导为我们的程序创建了几个默认的按钮，当然只有 About 按钮有事件处理。你可以像在 MFC 的工程中一样使用工具条编辑器修改 工具条资源，CreateSimpleToolBarCtrl()用这个工具条资源创建工具条。下面是向导生成的工具条在编辑器中的样子：



对于我们的时钟程序，我们添加四个按钮，两个按钮用来改变视图窗口的颜色，另外两个用来显示/隐藏工具条和状态条。下面是我们的新工具条：



这些按钮是：

IDC_CP_COLORS: 将视图窗口颜色改为 CodeProject 网站的颜色

IDC_BW_COLORS: 将视图窗口颜色改为黑白颜色

ID_VIEW_STATUS_BAR: 显示或隐藏状态条

ID_VIEW_TOOLBAR: 显示或隐藏工具条

前两个按钮都有相应的菜单项，它们都调用视图类的一个新函数 SetColor()，向这个函数传递前景颜色和背景颜色，视图窗口用这两个参数改变窗 口的显示。响应这两个按钮的处理函数与响应相应的菜单项的处理函数在使用 COMMAND_ID_HANDLER_EX 宏上没有区别，你可以查看例子工程的 代码了解这些消息处理的细节。在下一节我将介绍状态条和工具条按钮的 UI 状态更新，使它们能够反映工具条或状态条当前的状态。

工具条按钮的 UI 状态更新

向导生成的代码已经为 CMainFrame 添加了对 View|Toolbar 和 View|Status Bar 两个菜单项的 Check 和 Uncheck 的 UI 更新处理。这和第二章的程序一样：对 CMainFrame 类的两个命令使用 UI 更新的宏：

```
BEGIN_UPDATE_UI_MAP(CMainFrame)
    UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP)
```

```
    UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP)
END_UPDATE_UI_MAP()
```

我们的时钟程序的工具条按钮与对应的菜单项有相同的 ID，所以第一步就是为每个宏添加 UPDUI_TOOLBAR 标志：

```
BEGIN_UPDATE_UI_MAP(CMainFrame)
    UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
    UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
END_UPDATE_UI_MAP()
```

还需要添加两个函数响应工具条按钮的更新，但幸运的是向导已经为我们做了，所以如果此时编译这个程序，菜单项和工具条按钮都会更新。

使一个工具条支持 UI 状态更新

如果查看 CMainFrame::OnCreate()的代码你就会发现一段新的代码,这段代码设置了两个菜单项的初始状态：

```
LRESULT CMainFrame::OnCreate( ... )
{
// ...
    m_hWndClient = m_view.Create(...);

    UIAddToolBar(m_hWndToolBar);
    UISetCheck(ID_VIEW_TOOLBAR, 1);
    UISetCheck(ID_VIEW_STATUS_BAR, 1);
// ...
}
```

UIAddToolBar()将工具条的窗口句柄传给 CUpdateUI，所以当需要更新按钮的状态时 CUpdateUI 会向这个窗口发消息。另一个重要的调用位于 OnIdle()中：

```
BOOL CMainFrame::OnIdle()
{
    UIUpdateToolBar();
    return FALSE;
}
```

当消息队列中没有消息等待时 CMessageLoop::Run()就会调用 OnIdle()，UIUpdateToolBar()遍历 UI 更新 表，寻找那些带有 UPDUI_TOOLBAR 标志又被 UISetCheck()之类的函数改变了状态的的界面元素(当然是工具条)，相应的改变按钮的状态。需要注意得是如果更新弹出式菜单的状态就不需要做以上两步，因为 CUpdateUI 响应 WM_INITMENUPOPUP 消息，只有接到此消息时才更 新菜单状态。

如果查看例子代码就会发现它也演示了如何更新框架窗口的菜单条上的顶级菜单项的状态。有一个菜单项是执行 Start 和 Stop 命令，起到开始和停止 时钟的作用，当然这需要做一些不平常的事情：菜单条上的菜单项总是处于弹出状态。为了完整的介绍 CUpdateUI 我将它们也加进例子代码中，要了解它们 可以查找对 UIAddMenuBar()和 UIUpdateMenuBar ()两个函数的调用。

使用 Rebar 代替简单的工具条

CFrameWindowImpl 也支持使用 Rebar 控件，使你的程序看起来像 IE，使用 Rebar 也是在程序中使用多个工具条的一个方法 (译者加：前面讲过，另一个方法就是修改 WTL 的源代码)。要使用 Rebar 需要在向导的第二页选上支持 Rebar 的检查框，如下所示：

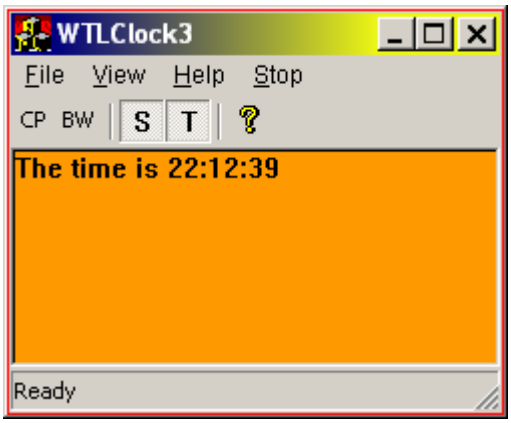
第二个例子工程 WTLClock3 就使用了 Rebar 控件，如果你正在跟着例子代码学习，那现在就打开 WTLClock3。

你首先会注意到创建工具条的代码有些不同，出现这种感觉是因为我们在程序中使用了 rebar。以下是相关的代码：

```
LRESULT CMainFrame::OnCreate(...)
{
    HWND hWndToolBar = CreateSimpleToolBarCtrl ( m_hWnd,
                       IDR_MAINFRAME, FALSE,
                       ATL_SIMPLE_TOOLBAR_PANE_STYLE );

    CreateSimpleReBar(ATL_SIMPLE_REBAR_NOBORDER_STYLE);
    AddSimpleReBarBand(hWndToolBar);
// ...
}
```

代码从创建工具条开始，只是使用了不同的风格，也就是 ATL_SIMPLE_TOOLBAR_PANE_STYLE，它定义在 atlframe.h 文件中，与 ATL_SIMPLE_TOOLBAR_STYLE 风格相似，只是附加了一些诸如 CCS_NOPARENTALIGN 之类的风格，这是使工具条作为 Rebar 的子窗口能够正常工作所必需的风格。

下一行代码是调用 CreateSimpleReBar()函数，该函数创建 Rebar 控件并将句柄存到 m_hWndToolBar 中。接下来调用 AddSimpleReBarBand()函数为 Rebar 创建一个条位并告诉 Rebar 这个条位上是一个工具条。



CMainFrame::OnViewToolBar()函数也有些不同，它只隐藏 Rebar 上工具条所在的条位而不是隐藏 m_hWndToolBar(如果隐藏 m_hWndToolBar 将隐藏整个 Rebar 而不仅仅是工具条)。

如果你使用多个工具条，只需像向导为我们生成的关于第一个工具条的代码那在 OnCreate()创建它们并调用 AddSimpleReBarBand()添加到 Rebar 就行了。CFrameWindowImpl 使用标准的 Rebar 控件，不像 MFC 那样支持可停 靠的工具条，你所能作得就是排列这些工具条在 Rebar 中的位置。

多窗格的状态条

WTL 另有一个状态条类实现多窗格的状态条，与 MFC 的默认的状态条一样有 CAPS，LOCK 和 NUM LOCK 指示器，这个类就是 CMultiPaneStatusBarCtrl，在 WTLClock3 例子工程中演示了如何使用这个类。这个类支持有限的 UI 更新，当弹出式菜单被显示时有“Default”属性的窗格会延伸到整个状态条的宽度用于显示菜单的掠过式帮助。

第一步就是在 CMainFrame 中声明一个 CMultiPaneStatusBarCtrl 类型的成员变量：

```
class CMainFrame : public ...
{
//...
protected:
    CMultiPaneStatusBarCtrl m_wndStatusBar;
};
```

接着在 OnCreate()中创建状态条并这只 UI 更新：

```
    m_hWndStatusBar = m_wndStatusBar.Create ( *this );
    UIAddStatusBar ( m_hWndStatusBar );
```

就像 CreateSimpleStatusBar()函数做得那样，我们也将状态条的句柄存放在 m_hWndStatusBar 中。

下一步就是调用 CMultiPaneStatusBarCtrl::SetPanes()函数建立窗格：

```
    BOOL SetPanes(int* pPanes, int nPanes, bool bSetText = true);
```

参数：

pPanes

存放窗格 ID 的数组

nPanes

窗格 ID 数组中元素的个数(译者加：就是窗格数)

bSetText

如果是 true，所有的窗格被立即设置文字，这一点将在下面解释。

窗格 ID 可以是 ID_DEFAULT_PANE，此 ID 用于创建支持掠过式帮助的窗格，窗格 ID 也可以

是字符串资源 ID。对于非默认的窗格 WTL 装载这个 ID 对应的字符串并计算宽度，并将窗格设置为相应的宽度，这和 MFC 使用的逻辑是一样的。
bSetText 控制着窗格是否立即显示相关的字符串，如果是 true，SetPanes()显示每个窗格的字符串，否则窗格就被置空。
下面是我们对 SetPanes()的调用：

```
   // Create the status bar panes.
int anPanes[] = { ID_DEFAULT_PANE, IDPANE_STATUS,
                  IDPANE_CAPS_INDICATOR };

   m_wndStatusBar.SetPanes ( anPanes, 3, false );
```

IDPANE_STATUS 对应的字符串是“@@@@”，这样应该有足够的宽度(希望是)显示两个时钟状态字符串“Running”和 “Stopped”。和 MFC 一样，你需要自己估算窗格的宽度，IDPANE_CAPS_INDICATOR 对应的字符串是“CAPS”。
窗格的 UI 状态更新
为了更新窗格上的文本，我们需要将相应的窗格添加到 UI 更新表：

```
   BEGIN_UPDATE_UI_MAP(CMainFrame)
      //...
      UPDATE_ELEMENT(1, UPDUI_STATUSBAR)  // clock status
      UPDATE_ELEMENT(2, UPDUI_STATUSBAR)  // CAPS indicator
   END_UPDATE_UI_MAP()
```

这个宏的第一个参数是窗格的索引而不是 ID，这很不幸，因为如果你重新排列了窗格，你要记得更新 UI 更新表。
由于我们在调用 SetPanes()是第三个参数是 false，所以窗格初始是空的。我们下一步要做得就是将时钟状态窗格的初始文本设为“Running”

```
   // Set the initial text for the clock status pane.
   UISetText ( 1, _T("Running") );
```

和前面一样，第一个参数是窗格的索引。UISetText()是状态条唯一支持的 UI 更新函数。
最后，在 CMainFrame::OnIdle()中添加对 UIUpdateStatusBar()函数的调用，使状态条的窗格能够在空闲时间被更新：

```
BOOL CMainFrame::OnIdle()
{
   UIUpdateToolBar();
   UIUpdateStatusBar();
   return FALSE;
}
```

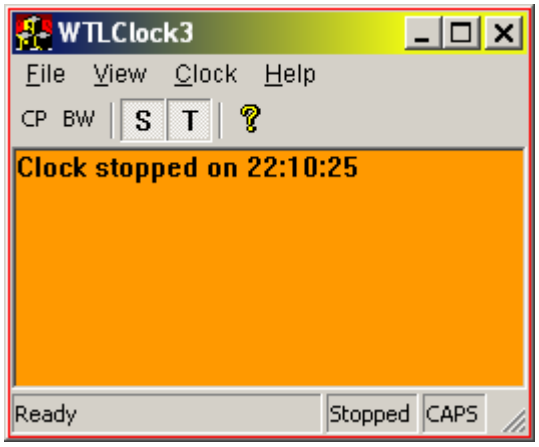
当你使用 UIUpdateStatusBar()时 CUpdateUI 的一个问题就暴露出来了——菜单项的文本在调用 UISetText()后没有 改变！如果你在看 WTLClock3 工程的代码，时钟的开始/停止菜单项被移到了 Clock 菜单，在菜单项命令的响应处理函数中设置菜单项的文本。无论如何，如果当前调用的是 UIUpdateStatusBar()，那么对 UISetText()的调用就不会起作用。我没有研究这个问题是否可以被修复，所 以如果你打算改变菜单的文本，你需要留意这个地方。
最后，我们需要检查 CAPS LOCK 键的状态，更新相应的两个窗格。这些代码是通过 OnIdle()被调用的，所以程序会在每次空闲时间检查它们的状态。

```
BOOL CMainFrame::OnIdle()
```

```
{
   // Check the current Caps Lock state, and if it is on, show the
   // CAPS indicator in pane 2 of the status bar.
   if ( GetKeyState(VK_CAPITAL) & 1 )
      UISetText ( 2, CString(LPCTSTR(IDPANE_CAPS_INDICATOR)) );
   else
      UISetText ( 2, _T("") );

   UIUpdateToolBar();
   UIUpdateStatusBar();
   return FALSE;
}
```

第一次调用 UISetText()时将从字符串资源中装载“CAPS”字符串，但是在 CString 的构造函数中使用了一个机灵的窍门(有充分的文档说明)。

在完成所有的代码之后，状态条看起来是这个样子：



承上启下：有关对话框的话题

在第四章我将介绍对话框的用法(包括 ATL 的类和 WTL 的增强功能)，控件的包装类和 WTL 有关对话框消息处理的改进。

引用和参考

"How to use the WTL multipane status bar control" by Ed Gadziemski 更详细的介绍了CMultiPaneStatusBarCtrl类的用法。

第四部分　对话框和控件

对第四章的介绍

MFC 的对话框和控件的封装真得可以节省你很多时间和功夫。没有 MFC 对控件的封装，你要操作控件就得耐着性子填写各种结构并写很多的 SendMessage 调用。MFC 还提供了对话框数据交换(DDX)，它可以在控件和变量之间传输数据。WTL 当然也提供了这些功能，并对控件的封装做了很多改进。本文将着眼于一个基于对话框的程序演示你以前用 MFC 实现的功能，除此之外还有 WTL 消息处理的增强功能。第五章将介绍高级界面特性和 WTL 对新控件的封装。

回顾一下 ATL 的对话框

现在回顾一下 第一章 提到的两个对话框类，CDialogImpl 和 CAxDialogImpl。CAxDialogImpl用于包含ActiveX控件的对话框。本文不准备介绍ActiveX控件，所以只使用CDialogImpl。

创建一个对话框需要做三件事：

创建一个对话框资源

从 CDialogImpl 类派生一个新类

添加一个公有成员变量 IDD，将它设置为对话框资源的 ID.

然后就像主框架窗口那样添加消息处理函数，WTL 没有改变这些，不过确实添加了一些其他能够在对话框中使用得特性。

通用控件的封装类

WTL 有许多控件的封装类对你应该比较熟悉，因为它们使用与 MFC 相同(或几乎相同)的名字。控件的方法的命名也和 MFC 一样，所以你可以参照 MFC 的文档使用这些 WTL 的封装类。不足之处是 F12 键不能方便地跳到类的定义代码处。

下面是 Windows 内建控件的封装类：

用户控件：CStatic, CButton, CListBox, CComboBox, CEdit, CScrollBar, CDragListBox

通用控件：CImageList, CListViewCtrl (CListCtrl in MFC), CTreeViewCtrl (CTreeCtrl in MFC), CHeaderCtrl, CToolBarCtrl, CStatusBarCtrl, CTabCtrl, CToolTipCtrl, CTrackBarCtrl (CSliderCtrl in MFC), CUpDownCtrl (CSpinButtonCtrl in MFC), CProgressBarCtrl, CHotKeyCtrl, CAnimateCtrl, CRichEditCtrl, CReBarCtrl, CComboBoxEx, CDateTimePickerCtrl, CMonthCalendarCtrl, CIPAddressCtrl

MFC 中没有的封装类：CPagerCtrl, CFlatScrollBar, CLinkCtrl (clickable hyperlink, available on XP only)

还有一些是 WTL 特有的类：CBitmapButton, CCheckListViewCtrl (带检查选择框的 list 控件), CTreeViewCtrlEx 和 CTreeItem (通常一起使用, CTreeItem 封装了 HTREEITEM), CHyperLink (类似于网页上的超链接对象，支持所有操作系统)

需要注意得一点是大多数封装类都是基于 CWindow 接口的，和 CWindow 一样，它们封装了 HWND 并对控件的消息进行了封装(例如，CListBox::GetCurSel()封装了 LB_GETCURSEL 消息)。所以和 CWindow 一样，创建一个控件的封装对象并将它与已经存在 的控件关联起来只占用很少的资源，当然也和 CWindow 一样，控件封装对象销毁时不销毁控件本身。也有一些例外，如 CBitmapButton, CCheckListViewCtrl 和 CHyperLink。

由于这些文章定位于有经验的 MFC 程序员，我就不浪费时间介绍这些封装类，它们和 MFC 相

应的控件封装相似。当然我会介绍 WTL 的新类：CBitmapButtonCBitmapButton 类与 MFC 的同名类有很大的不同，CHyperLink 则完全是新事物。

用应用程序向导生成基于对话框的程序

运行 VC 并启动 WTL 应用向导，相信你在做时钟程序时已经用过它了，为我们的新程序命名为 ControlMania1。在向导的第一页选择基于对话 框的应用，还要选择是使用模式对话框还是使用非模式对话框。它们有很大的区别，我将在第五章介绍它们的不同，现在我们选择简单的一种：模式对话框。如下所 示选择模式对话框和生成 CPP 文件选项：

第二页上所有的选项只对主窗口是框架窗口时有意义，现在它们是不可用状态，单击"Finish"，再单击"OK"完成向导。

正如你想的那样，向导生成的基于对话框程序的代码非常简单。_tWinMain()函数在 ControlMania1.cpp 中，下面是重要的部分：

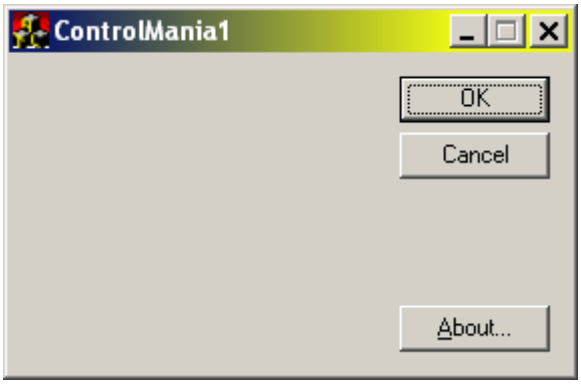
```
int WINAPI _tWinMain (
    HINSTANCE hInstance, HINSTANCE /*hPrevInstance*/,
    LPTSTR lpstrCmdLine, int nCmdShow )
{
    HRESULT hRes = ::CoInitialize(NULL);

    AtlInitCommonControls(ICC_COOL_CLASSES | ICC_BAR_CLASSES);

    hRes = _Module.Init(NULL, hInstance);

    int nRet = 0;
    // BLOCK: Run application
    {
        CMainDlg dlgMain;
```

```
        nRet = dlgMain.DoModal();
    }

    _Module.Term();
    ::CoUninitialize();
    return nRet;
}
```

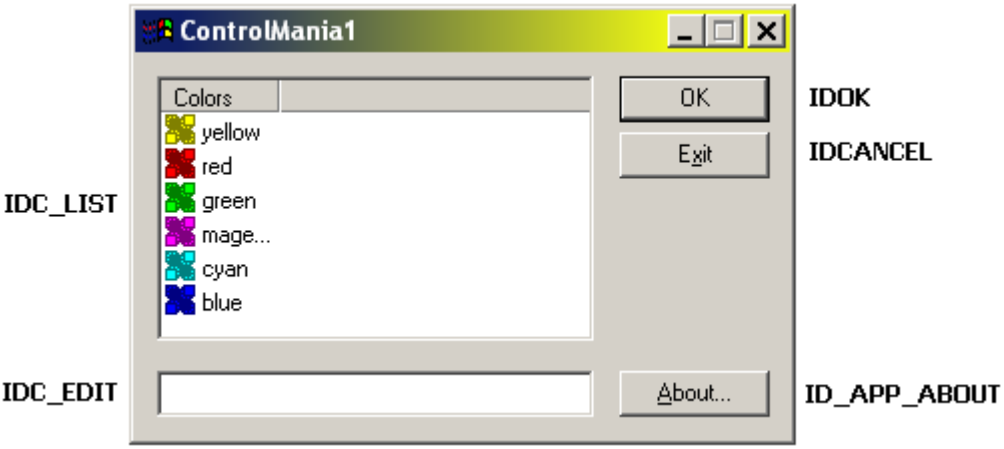
代码首先初始化 COM 并创建一个单线程公寓，这对于使用 ActiveX 控件的对话框是有必要得，接着调用 WTL 的功能函数 AtlInitCommonControls()，这个函数是对 InitCommonControlsEx()的封装。全局对象_Module 被初始化，主对话框显示出来。(注意所有使用 DoModal()创建的 ATL 对话框实际上是模式的，这不像 MFC，MFC 的所有对话框是非模式的，MFC 通过代码禁 用对话框的父窗口来模拟模式对话框的行为)最后，_Module 和 COM 被释放，DoModal()的返回值被用来作为程序的结束码。

将 CMainDlg 变量放在一个区块中是很重要的，因为 CMainDlg 可能有成员使用了 ATL 和 WTL 的特性， 这些成员在析构时也会用到 ATL/WTL 的特性，如果不使用区块，CMainDlg 将在 _Module.Term()(这个函数完成 ATL/WTL 的清理 工作)调用之后调用析构函数销毁自己(和成员)，并试图使用 ATL/WTL 的特性，这将导致程序出现诊断错误崩溃。(WTL 3 的向导生成的代码没有使用区块，使得我的一些程序在结束时崩溃)

你现在可以编译并运行这个程序，尽管它只是一个简陋的对话框：



CMainDlg 的代码处理了 WM_INITDIALOG, WM_CLOSE 和三个按钮的消息，如果你喜欢可以浏览一下这些代码，你应该能够看懂 CMainDlg 的声明，它的消息映射和它的消息处理函数。这个简单的工程还演示了如何将控件和变量联系起来，这个程序使用了几个控件。在接下来的讨论中你可以随时回来查看这些图表。



由于程序使用了 list view 控件，所以对 AtlInitCommonControls()的调用需要作些修

改，将其改为：

```
AtlInitCommonControls ( ICC_WIN95_CLASSES );
```

虽然这样注册的控件类比我们用到的多，但是当我们向对话框添加不同类型的控件时就不用随时记得添加名为 ICC_*的常量(译者加：以 ICC_开头的一系列常量)。

使用控件的封装类

有几种方法将一个变量和控件建立关联，可以使用 CWindows(或其它 Window 接口类，如 CListViewCtrl)，也可以使用 CWindowImpl 的派生类。如果只是需要一个临时变量就用 CWindow，如果需要子类化一个控件并处理发送给该控件的消息就需要使用 CWindowImpl。

ATL 方式 1 - 连接一个 CWindow 对象

最简单的方法是声明一个 CWindow 或其它 window 接口类，然后调用 Attach()方法，还可以使用 CWindow 的构造函数直接将变量与控件的 HWND 关联起来。

下面的代码三种方法将变量和一个 list 控件联系起来：

```
HWND hwndList = GetDlgItem(IDC_LIST);
CListViewCtrl wndList1 (hwndList);  // use constructor
CListViewCtrl wndList2, wndList3;

  wndList2.Attach ( hwndList );     // use Attach method
  wndList3 = hwndList;             // use assignment operator
```

记住 CWindow 的析构函数并不销毁控件窗口，所以在变量超出作用域时不需要将其脱离控件，如果你愿意的话还可以将其作为成员变量使用：你可以在 OnInitDialog()处理函数中建立变量与控件的联系。

ATL 方式 2 - 包容器窗口(CContainedWindow)

CContainedWindow 是介于 CWindow 和 CWindowImpl 之间的类，它可以子类化控件，在控件的父窗口中处理控件的消息，这 使得所有的消息处理都放在对话框类中，不需要为为每个控件生成一个单独的 CWindowImpl 派生类对象。需要注意的是不能用 CContainedWindow 处理 WM_COMMAND, WM_NOTIFY 和其他通知消息，因为这些消息是发给控件的父窗口的。

CContainedWindow 只是 CContainedWindowT 定义的一个数据类型，CContainedWindowT 才是真正的类， 它是一个模板类，使用 window 接口类的类名作为模板参数。这个特殊的 CContainedWindowT< CWindow>和 CWindow 功能一样，CContainedWindow 只是它定义的一个简写名称，要使用不同的 window 接口类只需将该类的类名作为模板参数就行了，例如 CContainedWindowT<CListViewCtrl>。

钩住一个 CContainedWindow 对象需要做四件事：

在对话框中创建一个 CContainedWindowT 成员变量。

将消息处理添加到对话框消息映射的 ALT_MSG_MAP 小节。

在对话框的构造函数中调用 CContainedWindowT 构造函数并告诉它哪个 ALT_MSG_MAP 小节的消息需要处理。

在 OnInitDialog()中调用 CContainedWindowT::SubclassWindow() 方法与控件建立关联。

在 ControlMania1 中，我对三个按钮分别使用了一个 CContainedWindow，对话框处理发送到每一个按钮的 WM_SETCURSOR 消息，并改变鼠标指针形状。

现在仔细看看这一步，首先，我们在 CMainDlg 中添加了 CContainedWindow 成员。

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
// ...
```

```
protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
};
```

其次，我们添加了 ALT_MSG_MAP 小节，OK 按钮使用 1 小节，Exit 按钮使用 2 小节。这意味着所有发送给 OK 按钮的消息将由 ALT_MSG_MAP(1)小节处理，所有发给 Exit 按钮的消息将由 ALT_MSG_MAP(2)小节处理。

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
public:
    BEGIN_MSG_MAP_EX(CMainDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAppAbout)
        COMMAND_ID_HANDLER(IDOK, OnOK)
        COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
    ALT_MSG_MAP(1)
        MSG_WM_SETCURSOR(OnSetCursor_OK)
    ALT_MSG_MAP(2)
        MSG_WM_SETCURSOR(OnSetCursor_Exit)
    END_MSG_MAP()

    LRESULT OnSetCursor_OK(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg);
    LRESULT OnSetCursor_Exit(HWND hwndCtrl, UINT uHitTest, UINT
uMouseMsg);
};
```

接着，我们调用每个 CContainedWindow 的构造函数，告诉它使用 ALT_MSG_MAP 的哪个小节。

```
CMainDlg::CMainDlg() : m_wndOKBtn(this, 1),
                       m_wndExitBtn(this, 2)
{
}
```

构造函数的参数是消息映射链的地址和 ALT_MSG_MAP 的小节号码，第一个参数通常使用 this，就是使用对话框自己的消息映射链，第二个参数告诉对象将消息发给 ALT_MSG_MAP 的哪个小节。

最后，我们将每个 CContainedWindow 对象与控件关联起来。

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...
    // Attach CContainedWindows to OK and Exit buttons
    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    return TRUE;
}
```

下面是新的 WM_SETCURSOR 消息处理函数：

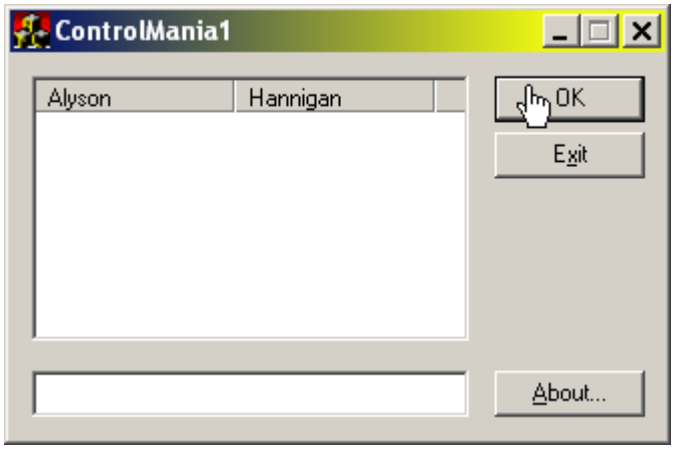
```
LRESULT CMainDlg::OnSetCursor_OK (HWND hwndCtrl, UINT uHitTest, UINT
uMouseMsg )
{
static HCURSOR hcur = LoadCursor ( NULL, IDC_HAND );

    if ( NULL != hcur )
        {
        SetCursor ( hcur );
        return TRUE;
        }
    else
        {
        SetMsgHandled(false);
        return FALSE;
        }
}

LRESULT CMainDlg::OnSetCursor_Exit ( HWND hwndCtrl, UINT uHitTest, UINT
uMouseMsg )
{
static HCURSOR hcur = LoadCursor ( NULL, IDC_NO );

    if ( NULL != hcur )
        {
        SetCursor ( hcur );
        return TRUE;
        }
    else
        {
        SetMsgHandled(false);
        return FALSE;
        }
}
```

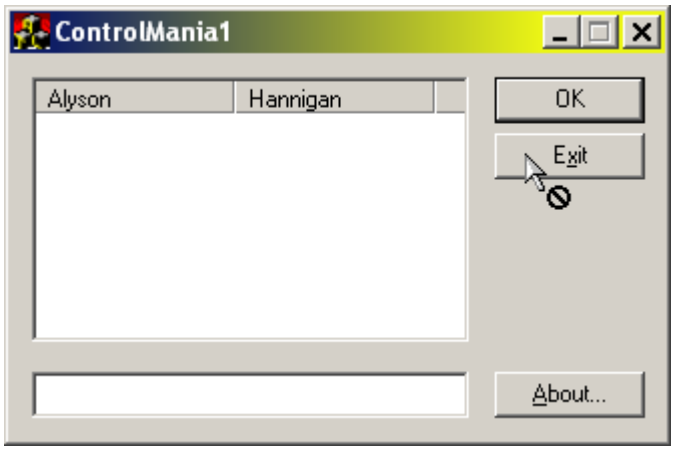
如果你还想使用按钮类的特性，你需要这样声明变量：

```
    CContainedWindowT<CButton> m_wndOKBtn;
```

这样就可以使用 `CButton` 类的方法。

当你把鼠标光标移到这些按钮上就可以看到 `WM_SETCURSOR` 消息处理函数的作用结果：





ATL 方式 3 - 子类化(Subclassing)

第三种方法创建一个 CWindowImpl 派生类并用它子类化一个控件。这和第二种方法有些相似，只是消息处理放在 CWindowImpl 类内部而不是对话框类中。

ControlMania1 使用这种方法子类化主对话框的 About 按钮。下面是 CButtonImpl 类，他从 CWindowImpl 类派生，处理 WM_SETCURSOR 消息：

```
class CButtonImpl : public CWindowImpl<CButtonImpl, CButton>
{
    BEGIN_MSG_MAP_EX(CButtonImpl)
        MSG_WM_SETCURSOR(OnSetCursor)
    END_MSG_MAP()

    LRESULT OnSetCursor(HWND hwndCtrl, UINT uHitTest, UINT uMouseMsg)
    {
    static HCURSOR hcur = LoadCursor ( NULL, IDC_SIZEALL );

        if ( NULL != hcur )
            {
            SetCursor ( hcur );
            return TRUE;
            }
        else
            {
```

```
            SetMsgHandled(false);
            return FALSE;
            }
    }
};
```

接着在主对话框声明一个 CButtonImpl 成员变量：

```
class CMainDlg : public CDialogImpl<CMainDlg>
{
// ...
protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
    CButtonImpl m_wndAboutBtn;
};
```

最后，在 OnInitDialog()种子类化 About 按钮。

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...
    // Attach CContainedWindows to OK and Exit buttons
    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    // CButtonImpl: subclass the About button
    m_wndAboutBtn.SubclassWindow ( GetDlgItem(ID_APP_ABOUT) );

    return TRUE;
}
```

WTL 方式 - 对话框数据交换(DDX)

WTL 的 DDX(对话框数据交换)很像 MFC，可以使用很简单的方法将变量和控件关联起来。首先，和前面的例子一样你需要从 CWindowImpl 派生一个新类，这次我们使用一个新类 CEditImpl，因为这次我们使用得是 Edit 控件。你还需要将#include atlddx.h 添加到 stdafx.h 中，这样就可以使用 DDX 代码。

要使主对话框支持 DDX，需要将 CWinDataExchange 添加到继承列表中：

```
class CMainDlg : public CDialogImpl<CMainDlg>,
                 public CWinDataExchange<CMainDlg>
{
//...
};
```

接着在对话框类中添加 DDX 链，这和 MFC 的类向导使用的 DoDataExchange()函数功能相似。对于不同类型的数据可以使用不同的 DDX 宏，我们使用 DDX_CONTROL 用来连接变量和控件，这次我们使用 CEditImpl 处理 WM_CONTEXTMENU 消息，使它能够在你右键 单控件时做一些事情。

```
class CEditImpl : public CWindowImpl<CEditImpl, CEdit>
{
    BEGIN_MSG_MAP_EX(CEditImpl)
```

```
        MSG_WM_CONTEXTMENU(OnContextMenu)
    END_MSG_MAP()

    void OnContextMenu ( HWND hwndCtrl, CPoint ptClick )
    {
        MessageBox("Edit control handled WM_CONTEXTMENU");
    }
};

class CMainDlg : public CDialogImpl<CMainDlg>,
                 public CWinDataExchange<CMainDlg>
{
//...

    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
    END_DDX_MAP()

protected:
    CContainedWindow m_wndOKBtn, m_wndExitBtn;
    CButtonImpl m_wndAboutBtn;
    CEditImpl   m_wndEdit;
};
```

最后，在 OnInitDialog()中调用 DoDataExchange()函数，这个函数是继承自 CWinDataExchange。 DoDataExchange()第一次被调用时完成相关控件的子类化工作，所以在这个例子中，DoDataExchange()子类化 ID 为 IDC_EDIT 的控件，将其与 m_wndEdit 建立关联。

```
LRESULT CMainDlg::OnInitDialog(...)
{
// ...
    // Attach CContainedWindows to OK and Exit buttons
    m_wndOKBtn.SubclassWindow ( GetDlgItem(IDOK) );
    m_wndExitBtn.SubclassWindow ( GetDlgItem(IDCANCEL) );

    // CButtonImpl: subclass the About button
    m_wndAboutBtn.SubclassWindow ( GetDlgItem(ID_APP_ABOUT) );

    // First DDX call, hooks up variables to controls.
    DoDataExchange(false);

    return TRUE;
}
```

DoDataExchange()的参数与 MFC 的 UpdateData()函数的参数意义相同，我会在下一节详细介绍。

现在运行 ControlMania1 程序，可以看到子类化的效果。鼠标右键单击编辑框将弹出消息框，当鼠标通过按钮上时鼠标形状会改变。

DDX 的详细内容

当然，DDX 是用来做数据交换的，WTL 支持在 Edit 控件和字符串之间交换数据，也可以将字符串解析成数字，转换成整型或浮点型变量，还支持 Check box 和 Radio button 组的状态与 int 型变量之间的转换。

DDX 宏

DDX 可以使用 6 种宏，每一种宏都对应一个 CWinDataExchange 类的方法支持其工作，每一种宏都用相同的形式：DDX_FOO(控件 ID, 变量)，每一种宏都可以支持多种类型的变量，例如 DDX_TEXT 的重载就支持多种类型的数据。

DDX_TEXT

在字符串和 edit box 控件之间传输数据，变量类型可以是 CString, BSTR, CComBSTR 或者静态分配的字符串数组，但是不能使用 new 动态分配的数组。

DDX_INT

在 edit box 控件和数字变量之间传输 int 型数据。

DDX_UINT

在 edit box 控件和数字变量之间传输无符号 int 型数据。

DDX_FLOAT

在 edit box 控件和数字变量之间传输浮点型(float)数据或双精度型数据(double)。

DDX_CHECK

在 check box 控件和 int 型变量之间转换 check box 控件的状态。

DDX_RADIO

在 radio buttons 控件组和 int 型变量之间转换 radio buttons 控件组的状态。

DDX_FLOAT 宏有一些特殊，要使用 DDX_FLOAT 宏需要在 stdafx.h 文件的所有 WTL 头文件包含之前添加一行定义：

```
#define _ATL_USE_DDX_FLOAT
```

这个定义是必要的，因为默认状态为了优化程序的大小而不支持浮点数。

有关 DoDataExchange()的详细内容

调用 DoDataExchange()方法和在 MFC 中使用 UpdateData()一样，DoDataExchange()的函数原型是：

```
BOOL DoDataExchange ( BOOL bSaveAndValidate = FALSE, UINT nCtlID = (UINT)-1 );
```

参数：

bSaveAndValidate

指示数据传输方向的标志。TRUE 表示将数据从控件传输给变量，FALSE 表示将数据从变量传输给控件。需要注意得是这个参数的默认值是 FALSE，而 MFC 的 UpdateData()函数的默认值是 TRUE。为了方便记忆，你可以使用 DDX_SAVE 和 DDX_LOAD 标号(它们分别被定义为 TRUE 和 FALSE)。

nCtlID

使用-1 可以更新所有控件，如果只想 DDX 宏作用于一个控件就使用控件的 ID。

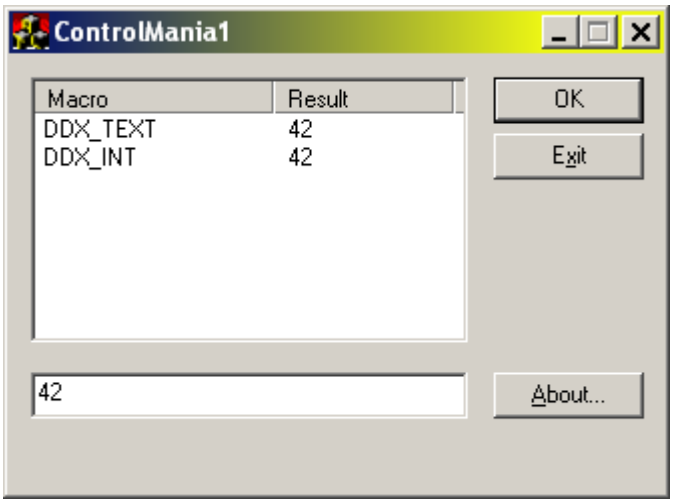
如果控件更新成功 DoDataExchange()会返回 TRUE，如果失败就返回 FALSE，对话框类有两个重载函数处理数据交换错误。一个是 OnDataExchangeError()，无论什么原因的错误都会调用这个函数，这个函数的默认实现在 CWinDataExchange 中，它仅仅是 驱动 PC 喇叭发出一声蜂鸣并将出错的控件设为当前焦点。另一个函数是 OnDataValidateError()，但

是要到本文的第五章介绍 DDV 时才用 得到。
使用 DDX
在 CMainDlg 中添加几个变量，演示 DDX 的使用方法。

```
class CMainDlg : public ...
{
//...
    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
        DDX_TEXT(IDC_EDIT, m_sEditContents)
        DDX_INT(IDC_EDIT, m_nEditNumber)
    END_DDX_MAP()

protected:
    // DDX variables
    CString m_sEditContents;
    int     m_nEditNumber;
};
```

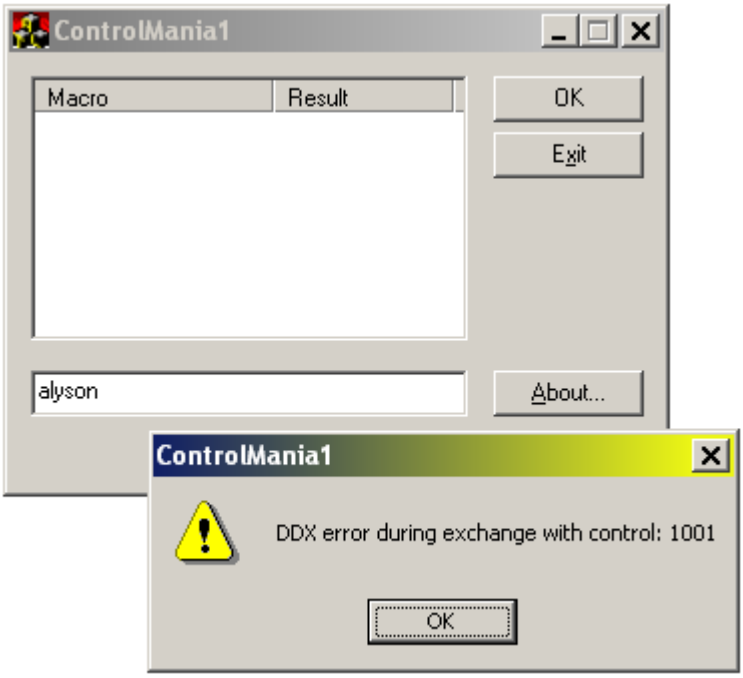
在 OK 按钮的处理函数中，我们首先调用 DoDataExchange()将将 edit 控件的数据传送给我们刚刚添加的两个变量，然后将结果显示在列表控件中。

```
LRESULT CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )
{
CString str;

    // Transfer data from the controls to member variables.
    if ( !DoDataExchange(true) )
        return;

    m_wndList.DeleteAllItems();

    m_wndList.InsertItem ( 0, _T("DDX_TEXT") );
    m_wndList.SetItemText ( 0, 1, m_sEditContents );

    str.Format ( _T("%d"), m_nEditNumber );
    m_wndList.InsertItem ( 1, _T("DDX_INT") );
    m_wndList.SetItemText ( 1, 1, str );
}
```

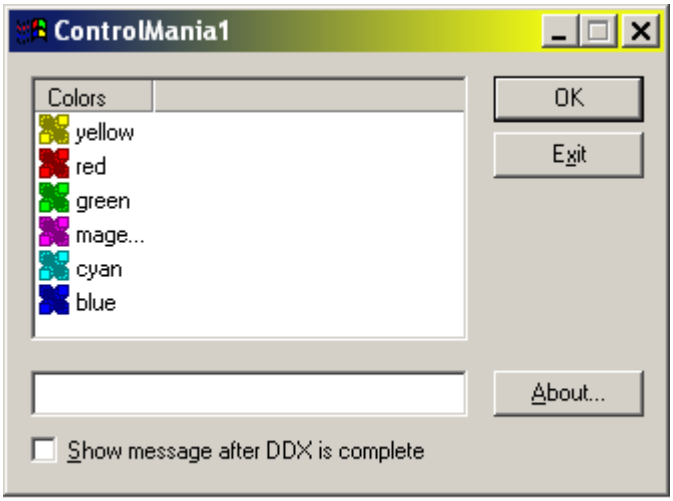


如果编辑控件输入的不是数字，DDX_INT 将会失败并触发 OnDataExchangeError()的调用，CMainDlg 重载了 OnDataExchangeError()函数显示一个消息框：

```
void CMainDlg::OnDataExchangeError ( UINT nCtrlID, BOOL bSave )
{
CString str;

   str.Format ( _T("DDX error during exchange with control: %u"),
nCtrlID );
   MessageBox ( str, _T("ControlMania1"), MB_ICONWARNING );

   ::SetFocus ( GetDlgItem(nCtrlID) );
}
```



作为最后一个使用 DDX 的例子，我们添加一个 check box 演示 DDX_CHECK 的使用：



IDC_SHOW_MSG

DDX_CHECK 使用的变量类型是 int 型，它的可能值是 0，1，2，分别对应 check box 的未选择状态，选择状态和不确定状态。你也可以使用常量 BST_UNCHECKED，BST_CHECKED，和 BST_INDETERMINATE 代替，对于 check box 来说只有选择和未选择两种状态，你可以将其视为布尔型变量。

以下是为使用 check box 的 DDX 而做的改动：

```
class CMainDlg : public ...
{
//...
    BEGIN_DDX_MAP(CMainDlg)
        DDX_CONTROL(IDC_EDIT, m_wndEdit)
        DDX_TEXT(IDC_EDIT, m_sEditContents)
        DDX_INT(IDC_EDIT, m_nEditNumber)
        DDX_CHECK(IDC_SHOW_MSG, m_nShowMsg)
    END_DDX_MAP()

protected:
    // DDX variables
    CString m_sEditContents;
    int     m_nEditNumber;
    int     m_nShowMsg;
};
```

在 OnOK()的最后，检查 m_nShowMsg 的值看看 check box 是否被选中。

```
void CMainDlg::OnOK ( UINT uCode, int nID, HWND hWndCtl )
{
    // Transfer data from the controls to member variables.
    if ( !DoDataExchange(true) )
        return;
//...
    if ( m_nShowMsg )
        MessageBox ( _T("DDX complete!"), _T("ControlMania1"),
                     MB_ICONINFORMATION );
```

```
}
```

使用其它 DDX_*宏的例子代码包含在例子工程中。

处理控件发送的通知消息

在 WTL 中处理通知消息与使用 API 方式编程相似，控件以 WM_COMMAND 或 WM_NOTIFY 消息的方式向父窗口发送通知事件，父窗口相应并做相应处理。少数其它的消息也可以看作是通知消息，例如：WM_DRAWITEM，当一个自画控件需要画自 己时就会发送这个消息，父窗口可以自己处理这个消息，也可以再将它反射给控件，MFC 采用得就是消息反射方式，使得控件能够自己处理通知消息，提高了代码 的封装性和可重用性。

在父窗口中响应控件的通知消息

以 WM_NOTIFY 和 WM_COMMAND 消息形式发送的通知消息包含各种信息。WM_COMMAND 消息的参数包含发送通知消息的控件 ID，控件的窗口句柄和通知代码，WM_NOTIFY 消息的参数还包含一个 NMHDR 数据结构的指针。ATL 和 WTL 有各种消息映射宏用来处理这些通知消 息，我在这里只介绍 WTL 宏，因为本文就是讲 WTL 得。使用这些宏需要在消息映射链中使用 BEGIN_MSG_MAP_EX 并包含 atlcrack.h 文 件。

消息映射宏

要处理 WM_COMMAND 通知消息需要使用 COMMAND_HANDLER_EX 宏：

COMMAND_HANDLER_EX(id, code, func)

处理从某个控件发送得某个通知代码。

COMMAND_ID_HANDLER_EX(id, func)

处理从某个控件发送得所有通知代码。

COMMAND_CODE_HANDLER_EX(code, func)

处理某个通知代码得所有消息，不管是从那个控件发出的。

COMMAND_RANGE_HANDLER_EX(idFirst, idLast, func)

处理 ID 在 idFirst 和 idLast 之间得控件发送的所有通知代码。

COMMAND_RANGE_CODE_HANDLER_EX(idFirst, idLast, code, func)

处理 ID 在 idFirst 和 idLast 之间得控件发送的某个通知代码。

例子:

COMMAND_HANDLER_EX(IDC_USERNAME, EN_CHANGE, OnUsernameChange): 处理从 ID 是 IDC_USERNAME 的 edit box 控件发出的 EN_CHANGE 通知消息。

COMMAND_ID_HANDLER_EX(IDOK, OnOK): 处理 ID 是 IDOK 的控件发送的所有通知消息。

COMMAND_RANGE_CODE_HANDLER_EX(IDC_MONDAY, IDC_FRIDAY, BN_CLICKED, OnDayClicked): 处理 ID 在 IDC_MONDAY 和 IDC_FRIDAY 之间控件发送的 BN_CLICKED 通知消息。

还有一些宏专门处理 WM_NOTIFY 消息，和上面的宏功能类似，只是它们的名字开头以“NOTIFY_”代替“COMMAND_”。

WM_COMMAND 消息处理函数的原型是：

```
  void func ( UINT uCode, int nCtrlID, HWND hwndCtrl );
```

WM_COMMAND 通知消息不需要返回值，所以处理函数也不需要返回值，WM_NOTIFY 消息处理函数的原型是：

```
  LRESULT func ( NMHDR* phdr );
```

消息处理函数的返回值用作消息相应的返回值，这不同于 MFC，MFC 的消息响应通过消息处理函数的 LRESULT*参数得到返回值。发送通知消息的 控件的窗口句柄和通知代码包含在 NMHDR 结构中，分别是 code 和 hendFrom 成员。和 MFC 一样的是如果通知消息发送的不是普通的 NMHDR 结构，你的消息处理函数应该将 phdr 参数转换成正确的类型。

我们将为 CMainDlg 添加 LVN_ITEMCHANGED 通知的处理函数，处理从 list 控件发出的这个通知，在对话框中显示当前选择的项目，先从添加消息映射宏和消息处理函数开始：

```
class CMainDlg : public ...
{
    BEGIN_MSG_MAP_EX(CMainDlg)
        NOTIFY_HANDLER_EX(IDC_LIST, LVN_ITEMCHANGED, OnListItemchanged)
    END_MSG_MAP()

    LRESULT OnListItemchanged(NMHDR* phdr);
//...
};
```

下面是消息处理函数：

```
LRESULT CMainDlg::OnListItemchanged ( NMHDR* phdr )
{
NMLISTVIEW* pnmlv = (NMLISTVIEW*) phdr;
int nSelItem = m_wndList.GetSelectedIndex();
CString sMsg;

    // If no item is selected, show "none". Otherwise, show its index.
    if ( -1 == nSelItem )
        sMsg = _T("(none)");
    else
        sMsg.Format ( _T("%d"), nSelItem );

    SetDlgItemText ( IDC_SEL_ITEM, sMsg );
    return 0;   // retval ignored
}
```

该处理函数并未用到 phdr 参数，我将他强制转换成 NMLISTVIEW*只是为了演示用法。

反射通知消息

如果你是用 CWindowImpl 的派生类封装控件，比如前面使用的 CEditImpl，你可以在类的内部处理通知消息而不是在对话框中，这就是通 知消息的反射，它和 MFC 的消息反射相似。不同的是在 WTL 中父窗口和控件都可以处理通知消息，而在 MFC 中只有控件能处理通知消息(译者加：除非你重载 WindowProc 函数，在 MFC 反射这些消息之前截获它们)。

如果需要将通知消息反射给控件封装类，只需在对话框的消息映射链中添加 REFLECT_NOTIFICATIONS()宏：

```
class CMainDlg : public ...
{
public:
    BEGIN_MSG_MAP_EX(CMainDlg)
        //...
        NOTIFY_HANDLER_EX(IDC_LIST,                    LVN_ITEMCHANGED,
OnListItemchanged)
        REFLECT_NOTIFICATIONS()
    END_MSG_MAP()
```

```
};
```

这个宏向消息映射链添加了一些代码处理那些未被前面的宏处理的通知消息，它检查消息传递的 HWND 窗口句柄是否有效并将消息转发给这个窗口，当然， 消息代码的数值被改变成 OLE 控件所使用的值，OLE 控件有与之相似的消息反射系统。新的消息代码值用 OCM_xxx 代替了 WM_xxx，但是消息的处理 方式和未反射前一样。

有 18 中被反射的消息：

控件通知消息: WM_COMMAND, WM_NOTIFY, WM_PARENTNOTIFY

自画消息: WM_DRAWITEM, WM_MEASUREITEM, WM_COMPAREITEM, WM_DELETEITEM

List box 键盘消息: WM_VKEYTOITEM, WM_CHARTOITEM

其它: WM_HSCROLL, WM_VSCROLL, WM_CTLCOLOR*

在你想添加反射消息处理的控件类内不要忘了使用 DEFAULT_REFLECTION_HANDLER()宏，DEFAULT_REFLECTION_HANDLER()宏确保将未被处理的消息交给 DefWindowProc()正确处理。 下面的例子是一个自画按钮类，它相应了从父窗口反射的 WM_DRAWITEM 消息。

```
class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>
{
public:
    BEGIN_MSG_MAP_EX(CODButtonImpl)
        MSG_OCM_DRAWITEM(OnDrawItem)
        DEFAULT_REFLECTION_HANDLER()
    END_MSG_MAP()

    void OnDrawItem ( UINT idCtrl, LPDRAWITEMSTRUCT lpdis )
    {
        // do drawing here...
    }
};
```

用来处理反射消息的 WTL 宏

我们现在只看到了 WTL 的消息反射宏中的一个：MSG_OCM_DRAWITEM，还有 17 个这样的反射宏。由于 WM_NOTIFY 和 WM_COMMAND 消息带的参数需要展开，WTL 提供了特殊的宏 MSG_OCM_COMMAND 和 MSG_OCM_NOTIFY 做这些事情。这些宏所作 的工作与 COMMAND_HANDLER_EX 和 NOTIFY_HANDLER_EX 宏相同，只是前面加了“REFLECTED_”，例如，一个树控件类 可能存在这样的消息映射链：

```
class CMyTreeCtrl : public CWindowImpl<CMyTreeCtrl, CTreeViewCtrl>
{
public:
  BEGIN_MSG_MAP_EX(CMyTreeCtrl)
    REFLECTED_NOTIFY_CODE_HANDLER_EX(TVN_ITEMEXPANDING,
OnItemExpanding)
    DEFAULT_REFLECTION_HANDLER()
  END_MSG_MAP()

  LRESULT OnItemExpanding ( NMHDR* phdr );
};
```

在ControlMania1对话框中用了一个树控件，和上面的代码一样处理TVN_ITEMEXPANDING

消息，CMainDlg 类的成员 m_wndTree 使用 DDX 连接到控件上，CMainDlg 反射通知消息，树控件的处理函数 OnItemExpanding()是这样的：

```
LRESULT CBuffyTreeCtrl::OnItemExpanding ( NMHDR* phdr )
{
NMTREEVIEW* pnmtv = (NMTREEVIEW*) phdr;

   if ( pnmtv->action & TVE_COLLAPSE )
      return TRUE;    // don't allow it
   else
      return FALSE;   // allow it
}
```

运行 ControlMania1，用鼠标点击树控件上的+/-按钮，你就会看到消息处理函数的作用—节点展开后就不能再折叠起来。

容易出错和混淆的地方

对话框的字体

如果你像我一样对界面非常讲究并且正在只用 windows 2000 或 XP，你就会奇怪为什么对话框使用 MS Sans Serif 字体而不是 Tahoma 字体，因为 VC6 太老了，它生成的资源文件在 NT 4 上工作的很好，但是对于新的版本就会有问题。你可以自己修改，需要手工编辑资源文件，据我所知 VC 7 不存在这个问题。

在资源文件中对话框的入口处需要修改 3 个地方：

对话框类型：将 DIALOG 改为 DIALOGEX

窗口类型：添加 DS_SHELLFONT

对话框字体：将 MS Sans Serif 改为 MS Shell Dlg

不幸的是前两个修改会在每次保存资源文件时丢失(被 VC 又改回原样)，所以需要重复这些修改，下面是改动之前的代码：

```
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 187, 102
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
  ...
END
```

这是改动之后的代码：

```
IDD_ABOUTBOX DIALOGEX DISCARDABLE  0, 0, 187, 102
STYLE DS_SHELLFONT | DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Shell Dlg"
BEGIN
  ...
END
```

这样改了之后，对话框将在新的操作系统上使用 Tahoma 字体，而在老的操作系统上仍旧使用 MS Sans Serif 字体。

_ATL_MIN_CRT

本文的论坛 FAQ已 经做过解释， ATL包含的优化设置让你创建一个不使用C运行库(CRT)的

程序，使用这个优化需要在预处理设置中添加_ATL_MIN_CRT标号，向导生成的代码在Release配置中默认使用了这个优化。由于我写程序总是会用到CRT函数，所以我总是去掉这个标号，如果你在CString类或DDX中用到了浮点运 算特性，你也要去掉这个标号。

继续

在第五章，我将介绍对话框数据验证(DDV)，WTL 对新控件的封装和自画控件、自定外观控件等一些高级界面特性。

第五部分　使用控件

在上一篇文章我们介绍了一些与对话框和控件有关的 WTL 的特性，它们和 MFC 的相应的类作用相同。本文将介绍一些新类实现高级界面特性新类：控件自画和自定外观控件，新的 WTL 控件，UI updating 和对话框数据验证(DDV)。

特别的自画和外观定制类

由于自画和定制外观控件在图形用户界面中是很常用的手段，所以WTL提供了几个嵌入类来完成这些令人厌烦的工作。我接着就会介绍它们，事实上我们在 上一个例子工程 ControlMania2 的结尾部分已经这么做了。如果你正随着我的讲解用应用程序生成向导创建新工程，请不要忘了使用无模式对话框，为 了使正常工作必须使用无模式对话框，我会在对话框中控件的UI Updating部分详细解释为什么这样作。

COwnerDraw

控件的自画需要响应四个消息：WM_MEASUREITEM, WM_DRAWITEM, WM_COMPAREITEM, 和 WM_DELETEITEM，在 atlframe.h 头文件中定义的 COwnerDraw 类可以简化这些工作，使用这个类就不需要处理这四个消息，你只 需将消息链入 COwnerDraw，它会调用你的类中的重载函数。

如何将消息链入 COwnerDraw 取决与你是否将消息反射给控件，两种方法有些不同。下面是 COwnerDraw 类的消息映射链，它使得两种方法的差别更加明显：

```
template <class T> class COwnerDraw
{
public:
  BEGIN_MSG_MAP(COwnerDraw<T>)
    MESSAGE_HANDLER(WM_DRAWITEM, OnDrawItem)
    MESSAGE_HANDLER(WM_MEASUREITEM, OnMeasureItem)
    MESSAGE_HANDLER(WM_COMPAREITEM, OnCompareItem)
    MESSAGE_HANDLER(WM_DELETEITEM, OnDeleteItem)
  ALT_MSG_MAP(1)
    MESSAGE_HANDLER(OCM_DRAWITEM, OnDrawItem)
    MESSAGE_HANDLER(OCM_MEASUREITEM, OnMeasureItem)
    MESSAGE_HANDLER(OCM_COMPAREITEM, OnCompareItem)
    MESSAGE_HANDLER(OCM_DELETEITEM, OnDeleteItem)
  END_MSG_MAP()
};
```

注意，消息映射链的主要部分处理 WM_*消息，而 ATL 部分处理反射的消息，OCM_*。自画的通知消息就像 WM_NOTIFY 消息一样，你可以在父窗口处理它们，也可以将它们反射会控件，如果你使用前一种方法，消息被直接链入 COwnerDraw：

```
class CSomeDlg : public COwnerDraw<CSomeDlg>, ...
{
  BEGIN_MSG_MAP(CSomeDlg)
    //...
    CHAIN_MSG_MAP(COwnerDraw<CSomeDlg>)
  END_MSG_MAP()
```

```
  void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

当然，如果你想要控件自己处理这些消息，你需要使用 CHAIN_MSG_MAP_ALT 宏将消息链入 ALT_MSG_MAP(1)部分：

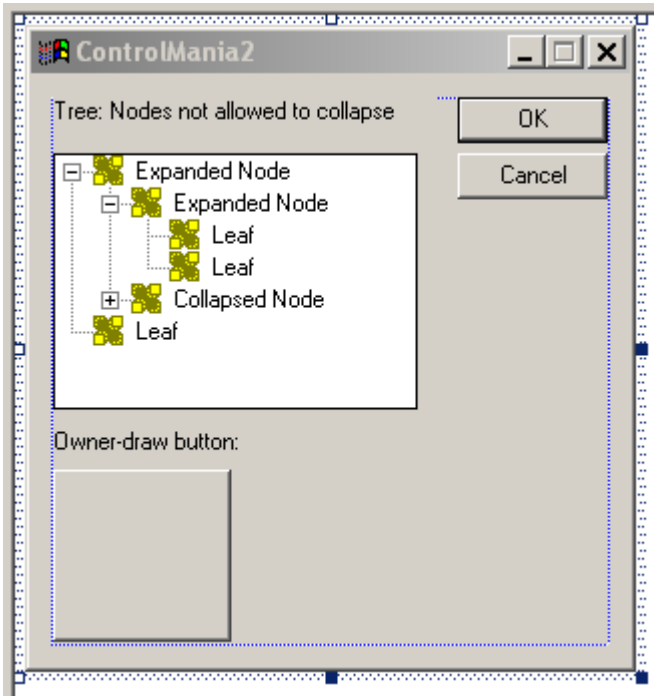
```
class CSomeButtonImpl : public COwnerDraw<CSomeButtonImpl>, ...
{
  BEGIN_MSG_MAP(CSomeButtonImpl)
    //...
    CHAIN_MSG_MAP_ALT(COwnerDraw<CSomeButtonImpl>, 1)
    DEFAULT_REFLECTION_HANDLER()
  END_MSG_MAP()

  void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

COwnerDraw 类将对消息传递的参数展开，然后调用你的类中的实现函数。上面的例子中，我们自己的类实现 DrawItem()函数，当有 WM_DRAWITEM 或 OCM_DRAWITEM 消息被链入 COwnerDraw 时，这个函数就会被调用。你可以重载的方法有：

```
void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
int  CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
void DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
```

如果你不想处理某个消息，你可以调用 SetMsgHandled(false)，消息会被传递给消息映射链中的其他响应者。 SetMsgHandled()事实上是 COwnerDraw 类的成员函数，但是它的作用和在 BEGIN_MSG_MAP_EX()中使用 SetMsgHandled()一样。

对于 ControlMania2，它从 ControlMania1 中的树控件开始，添加了自画按钮处理反射的 WM_DRAWITEM 消息，下面是资源编辑器中的新按钮：



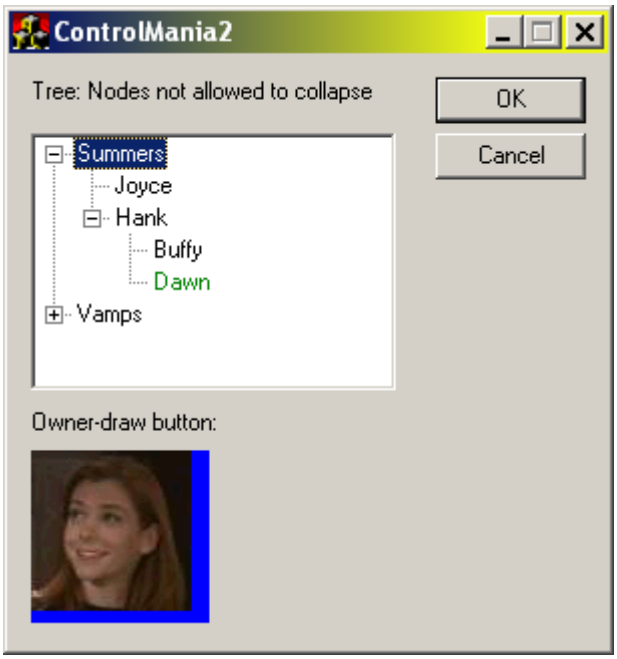
现在我们需要一个新类实现自画按钮：

```
class CODButtonImpl : public CWindowImpl<CODButtonImpl, CButton>,
                      public COwnerDraw<CODButtonImpl>
{
public:
   BEGIN_MSG_MAP_EX(CODButtonImpl)
      CHAIN_MSG_MAP_ALT(COwnerDraw<CODButtonImpl>, 1)
      DEFAULT_REFLECTION_HANDLER()
   END_MSG_MAP()

   void DrawItem ( LPDRAWITEMSTRUCT lpdis );
};
```

DrawItem()使用了像 BitBlt()这样的 GDI 函数向按钮的表面画位图，代码应该很容易理解，因为 WTL 使用的类名和函数名都和 MFC 类似。

```
void CODButtonImpl::DrawItem ( LPDRAWITEMSTRUCT lpdis )
{
// NOTE: m_bmp is a CBitmap init'ed in the constructor.
CDCHandle dc = lpdis->hDC;
CDC dcMem;

   dcMem.CreateCompatibleDC ( dc );
   dc.SaveDC();
   dcMem.SaveDC();

   // Draw the button's background, red if it has the focus, blue if
not.
   if ( lpdis->itemState & ODS_FOCUS )
      dc.FillSolidRect ( &lpdis->rcItem, RGB(255,0,0) );
   else
      dc.FillSolidRect ( &lpdis->rcItem, RGB(0,0,255) );

   // Draw the bitmap in the top-left, or offset by 1 pixel if the button
   // is clicked.
   dcMem.SelectBitmap ( m_bmp );

   if ( lpdis->itemState & ODS_SELECTED )
      dc.BitBlt ( 1, 1, 80, 80, dcMem, 0, 0, SRCCOPY );
   else
      dc.BitBlt ( 0, 0, 80, 80, dcMem, 0, 0, SRCCOPY );

   dcMem.RestoreDC(-1);
   dc.RestoreDC(-1);
}
```

我们的按钮看起来是这个样子：



CCustomDraw

CCustomDraw 类使用和 COwnerDraw 类相同的方法处理 NM_CUSTOMDRAW 消息，对于自定绘制的每个阶段都有相应的重载函数：

```
DWORD OnPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnPostErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);

DWORD OnItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPostPaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPreErase(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
DWORD OnItemPostEraset(int idCtrl, LPNMCUSTOMDRAW lpNMCD);

DWORD OnSubItemPrePaint(int idCtrl, LPNMCUSTOMDRAW lpNMCD);
```

这些函数默认都是返回 CDRF_DODEFAULT，如果想自画控件或返回一个不同的值，就需要重载这些函数：

你可能注意到上面的屏幕截图将“道恩”(Dawn：女名)显示成绿色，这是因为 CBuffyTreeCtrl 将消息链入 CCustomDraw 并重载了 OnPrePaint() 和 OnItemPrePaint()方法。向树控件中添加节点时，节点的 item data 字段被设置成 1，OnItemPrePaint()检查这个值，然后改变文字的颜色。

```
DWORD CBuffyTreeCtrl::OnPrePaint(int idCtrl,
                                 LPNMCUSTOMDRAW lpNMCD)
{
    return CDRF_NOTIFYITEMDRAW;
}

DWORD CBuffyTreeCtrl::OnItemPrePaint(int idCtrl,
                                     LPNMCUSTOMDRAW lpNMCD)
```

```
{
    if ( 1 == lpNMCD->lItemlParam )
        pnmtv->clrText = RGB(0,128,0);

    return CDRF_DODEFAULT;
}
```

CCustomDraw 类也有 SetMsgHandled()函数，你可以像在 COwnerDraw 类那样使用这个函数。

WTL 的新控件

WTL 有几个新控件，它们要么是其他封装类的扩展(像 CTreeViewCtrlEx)，要么是提供 windows 标准控件没有的新功能(像 CHyperLink)。

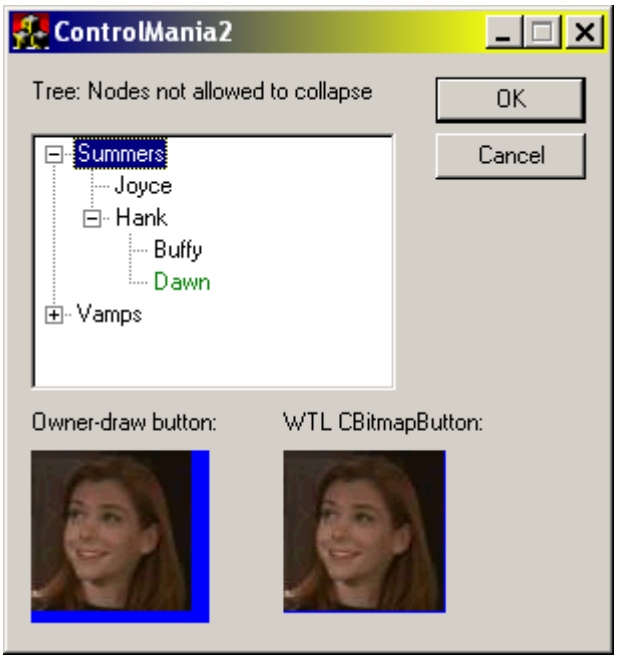
CBitmapButton

WTL 的 CBitmapButton 类声明在 atlctrlx.h 中,它比 MFC 的同名类使用起来要简单的多。WTL 的 CBitmapButton 类使用 image list 而不是单个的位图资源，你可以将多个按钮的图像放到一个位图文件中，减少 GDI 资源的占用。这对于使用很多图片并需要在 Windows 9X 系统上运行的程序很有好处，因为使用太多的单个位图将会很快耗尽 GDI 资源并导致系统崩溃。

CBitmapButton 是一个 CWindowImpl 派生类，它又很多特色：自动调整控件的大小，自动生成 3D 边框，支持 hot-tracking，每个按钮可以使用多个图像分别表示按钮的不同状态。在 ControlMania2 中，我们对前面的例子创建的自画按钮使用 CBitmapButton 类。现在 CMainDlg 对话框类中添加 CBitmapButton 类型的变量 m_wndBmpBtn，调用 SubclassWindow()函数或使用 DDX 将其和控件联系起来，将位图装载到 image list 并告诉按钮使用这个 image list，还要告诉按钮每个图像分别对应按钮的什么状态。下面是 OnInitDialog()函数中建立和使用这个按钮的代码段：

```
    // Set up the bitmap button
CImageList iml;

    iml.CreateFromImage ( IDB_ALYSON_IMGLIST, 81, 1, CLR_NONE,
                          IMAGE_BITMAP, LR_CREATEDIBSECTION );

    m_wndBmpBtn.SubclassWindow ( GetDlgItem(IDC_ALYSON_BMPBTN) );
    m_wndBmpBtn.SetToolTipText ( _T("Alyson") );
    m_wndBmpBtn.SetImageList ( iml );
    m_wndBmpBtn.SetImages ( 0, 1, 2, 3 );
```

默认情况下，按钮只是引用 image list，所以 OnInitDialog()不能 delete 它所创建的 image list。下面显示的是新按钮的一般状态，注意控件是如何根据图像的大小来调整自己的大小。



因为 CBitmapButton 是一个非常有用的类，我想介绍一下它的公有方法。

CBitmapButton methods

CBitmapButtonImpl 类包含了实现一个按钮的所有代码，除非你想重载某个方法或消息处理，你可以对控件直接使用 CBitmapButton 类。

CBitmapButtonImpl constructor

CBitmapButtonImpl(DWORD dwExtendedStyle = BMPBTN_AUTOSIZE,HIMAGELIST hImageList = NULL)

构造函数可以指定按钮的扩展样式(这与窗口的样式不冲突)和图像列表，通常使用默认参数就足够了，因为可以使用其他的方法设定这些属性。

SubclassWindow()

BOOL SubclassWindow(HWND hWnd)

SubclassWindow()是个重载函数，主要完成控件的子类化和初始化控件类保有的内部数据。

Bitmap button extended styles

DWORD GetBitmapButtonExtendedStyle()

DWORD SetBitmapButtonExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)

CBitmapButton 支持一些扩展样式，这些扩展样式会对按钮的外观和操作方式产生影响：

BMPBTN_HOVER

使用 hot-tracking，当鼠标移到按钮上时按钮被画成焦点状态。

BMPBTN_AUTO3D_SINGLE, BMPBTN_AUTO3D_DOUBLE

在 按钮图像周围自动产生一个三维边框，当按钮拥有焦点时会显示一个表示焦点的虚线矩形框。另外如果你没有指定按钮按下状态的图像，将会自动生成一个。BMPBTN_AUTO3D_DOUBLE 样式生成的边框稍微粗一些，其他特征和 BMPBTN_AUTO3D_SINGLE 一样。

BMPBTN_AUTOSIZE

按钮调整自己的大小以适应图像大小，这是默认样式。

BMPBTN_SHAREIMAGELISTS

如果指定这个样式，按钮不负责销毁按钮使用的 image list，如果不使用这个样式，

CBitmapButton 的析构函数会销毁按钮使用的 image list。

BMPBTN_AUTOFIRE

如果设置这个样式，在按钮上按住鼠标左键不放将会产生连续的 WM_COMMAND 消息。

调用 SetBitmapButtonExtendedStyle()时，dwMask 参数控制着那个样式将被改变，默认值是 0，意味着用新样式完全替换旧的样式。

Image list management

HIMAGELIST GetImageList()

HIMAGELIST SetImageList(HIMAGELIST hImageList)

调用 SetImageList()设置按钮使用的 image list。

Tooltip management

int GetToolTipTextLength()

bool GetToolTipText(LPTSTR lpstrText, int nLength)

bool SetToolTipText(LPCTSTR lpstrText)

CBitmapButton 支持显示工具提示(tooltip)，调用 SetToolTipText()指定显示的文字。

Setting the images to use

void SetImages(int nNormal, int nPushed = -1,int nFocusOrHover = -1, int nDisabled = -1)

调用 SetImages()函数告诉按钮分别使用 image list 的拿一个图像表示那个状态。nNormal 是必须的，其它是可选的，使用－1 表示对应的状态没有图像。

CCheckListViewCtrl

CCheckListViewCtrl 类在 atlctrlx.h 中定义，它是一个 CWindowImpl 派生类，实现了一个带检查框的 list view 控件。它和 MFC 的 CCheckListBox 不同，CCheckListBox 只是一个 list box，不是 list view。CCheckListViewCtrl 类非常简单，只添加了很少的函数，当然，它使用了一个新的辅助类 CCheckListViewCtrlImplTraits，它和 CWinTraits 类的作用类似，只是第三个参数是 list view 控件的扩展样式属性，如果你没有定义自己的 CCheckListViewCtrlImplTraits，它将使用没默认的样式：LVS_EX_CHECKBOXES | LVS_EX_FULLROWSELECT。

下面是一个定义 list view 扩展样式属性的例子，加入了一个使用这个样式的新类。(注意，扩展属性必须包含 LVS_EX_CHECKBOXES，否则会因起断言错误消息。)

```
typedef CCheckListViewCtrlImplTraits<
   WS_CHILD | WS_VISIBLE | LVS_REPORT,
   WS_EX_CLIENTEDGE,
   LVS_EX_CHECKBOXES | LVS_EX_GRIDLINES | LVS_EX_UNDERLINEHOT |
     LVS_EX_ONECLICKACTIVATE> CMyCheckListTraits;

class CMyCheckListCtrl :
   public CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl,
                                 CMyCheckListTraits>
{
private:
   typedef CCheckListViewCtrlImpl<CMyCheckListCtrl, CListViewCtrl,
                                  CMyCheckListTraits> baseClass;
public:
```

```
    BEGIN_MSG_MAP(CMyCheckListCtrl)
        CHAIN_MSG_MAP(baseClass)
    END_MSG_MAP()
};
```

CCheckListViewCtrl methods

SubclassWindow()

当子类化一个已经存在的 list view 控件时，SubclassWindow()查看 CCheckListViewCtrlImplTraits 的扩展样式属性并将之应用到控件上。未用到前两个参数(窗口样式和扩展窗口样式)。

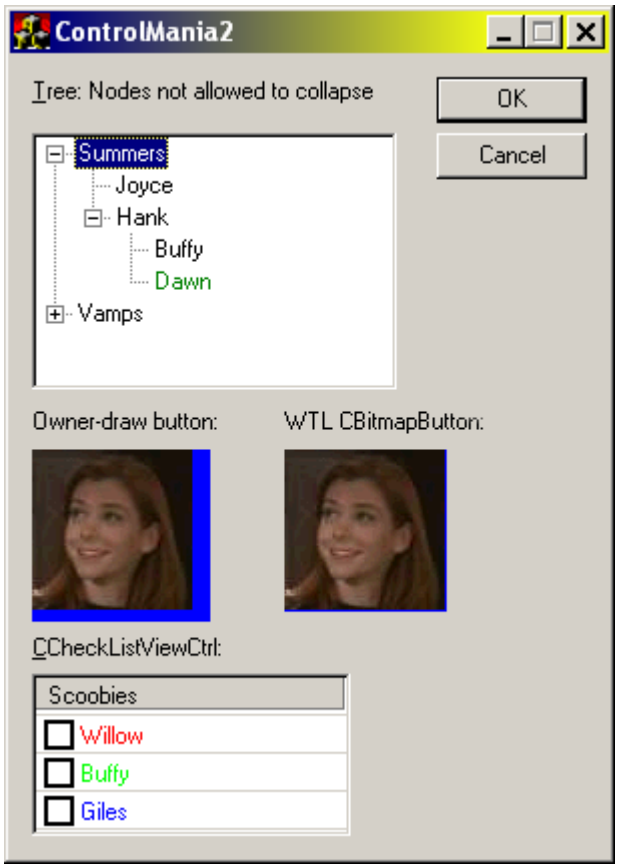
SetCheckState() and GetCheckState()

这些方法实际上是在 CListViewCtrl 中，SetCheckState()使用行的索引和一个布尔类型参数，该布尔参数的值表示是否 check 这一行。GetCheckState()以行索引未参数，返回改行的 checked 状态。

CheckSelectedItems()

这个方法使用 item 的索引作为参数，它翻转这个 item 的 check 状态，这个 item 必须是被选定的，同时还将其他所有被选择的 item 设置成相应状态(译者加：多选状态下)。你大概不会用到这个方法，因为 CCheckListViewCtrl 会在 check box 被单击或用户按下了空格键时设置相应的 item 的状态。

下面是 ControlMania2 中的 CCheckListViewCtrl 的样子：



CTreeViewCtrlEx and CTreeItem

有两个类使得树控件的使用简化了很多：CTreeItem 类封装了 HTREEITEM，一个 CTreeItem 对象含有一个 HTREEITEM 和一个指向包含这个 HTREEITEM 的树控件的指针，使你不必每次调用都引用树控件；CTreeViewCtrlEx 和 CTreeViewCtrl 一样，只是它的方法操作

CTreeItem 而不是 HTREEITEM。例如，InsertItem()函数返回一个 CTreeItem 而不是 HTREEITEM，你可以使用 CTreeItem 操作新添加的 item。下面是一个例子：

```
// Using plain HTREEITEMs:
HTREEITEM hti, hti2;

   hti = m_wndTree.InsertItem ( "foo", TVI_ROOT, TVI_LAST );
   hti2 = m_wndTree.InsertItem ( "bar", hti, TVI_LAST );
   m_wndTree.SetItemData ( hti2, 100 );

// Using CTreeItems:
CTreeItem ti, ti2;

   ti = m_wndTreeEx.InsertItem ( "foo", TVI_ROOT, TVI_LAST );
   ti2 = ti.AddTail ( "bar", 0 );
   ti2.SetData ( 100 );
```

CTreeViewCtrl 对 HTREEITEM 的每一个操作，CTreeItem 都有与之对应的方法，正像每一个关于 HWND 的 API 都有一个 CWindow 方法与之对应一样。查看 ControlMania2 的代码可以看到更多的 CTreeViewCtrlEx 和 CTreeItem 类的方法的演 示。

CHyperLink

CHyperLink 是一个 CWindowImpl 派生类，它子类化一个 static text 控件，使之变成可点击的超链接。CHyperLink 根据用户的 IE 使用的颜色画链接对象，还支持键盘导航。CHyperLink 类的构造函数没有参数，下面是其它的公有方法。

CHyperLink methods

CHyperLinkImpl 类内含实现一个超链接的全部代码，如果不需要重载它的方法或处理消息的话，你可以直接使用 CHyperLink 类。

SubclassWindow()

BOOL SubclassWindow(HWND hWnd)

重载函数 SubclassWindow()完成控件子类化，然后初始化该类保有的内部数据。

Text label management

bool GetLabel(LPTSTR lpstrBuffer, int nLength)

bool SetLabel(LPCTSTR lpstrLabel)

获得或设置控件显示的文字，如果不指定显示文字，控件会显示资源编辑器指定给控件的静态字符串。

Hyperlink management

bool GetHyperLink(LPTSTR lpstrBuffer, int nLength)

bool SetHyperLink(LPCTSTR lpstrLink)

获得或设置控件关联超链接的 URL，如果不指定超链接 URL，控件会使用显示的文字字符串作为 URL。

Navigation

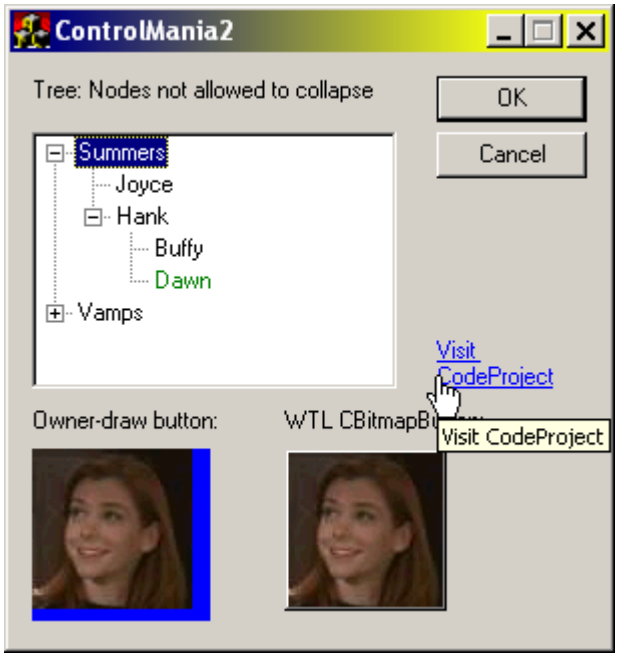
bool Navigate()

导航到当前超链接的 URL，该 URL 或者是由 SetHyperLink()函数指定的 URL，或者就是控件的窗口文字。

Tooltip management

没有公开的方法设置工具提示，所以需要直接使用 CToolTipCtrl 成员 m_tip。

下图显示的就是 ControlMania2 对话框中的超链接控件：



在 OnInitDialog()函数中设置 URL：

```
    m_wndLink.SetHyperLink ( _T("http://www.codeproject.com/") );
```

对话框中控件的 UI Updating

对话框中的的 UI updating 控制比 MFC 中简单得多，在 MFC 中，你需要响应未公开的 WM_KICKIDLE 消息，处理这个消息并触发控件的 updating，在 WTL 中，没有这个诡计，不过向导存在一个 BUG，需要手工添加一行代码解决这个问题。

首先需要记住的是对话框必须是无模式的，因为 CUpdateUI 需要在程序的消息循环控制下工作。如果对话框是模式的，系统处理消息循环，我们程序的空闲处理函数就不会被调用，由于 CUpdateUI 是在空闲时间工作的，所以没有空闲处理就没有 UI updating。

ControlMania2 的对话框是非模式的，类定义的开始部分很像是一个框架窗口类：

```
class CMainDlg : public CDialogImpl<CMainDlg>, public
CUpdateUI<CMainDlg>,
              public CMessageFilter, public CIdleHandler
{
public:
    enum { IDD = IDD_MAINDLG };

    virtual BOOL PreTranslateMessage(MSG* pMsg);
    virtual BOOL OnIdle();

    BEGIN_MSG_MAP_EX(CMainDlg)
        MSG_WM_INITDIALOG(OnInitDialog)
        COMMAND_ID_HANDLER_EX(IDOK, OnOK)
        COMMAND_ID_HANDLER_EX(IDCANCEL, OnCancel)
        COMMAND_ID_HANDLER_EX(IDC_ALYSON_BTN, OnAlysonODBtn)
    END_MSG_MAP()
```

```
    BEGIN_UPDATE_UI_MAP(CMainDlg)
    END_UPDATE_UI_MAP()
//...
};
```

注意 CMainDlg 类从 CUpdateUI 派生并含有一个 update UI 链。OnInitDialog()做了这些工作，这和前面介绍的框架窗口中的代码很相似：

```
    // register object for message filtering and idle updates
    CMessageLoop* pLoop = _Module.GetMessageLoop();
    ATLASSERT(pLoop != NULL);
    pLoop->AddMessageFilter(this);
    pLoop->AddIdleHandler(this);

    UIAddChildWindowContainer(m_hWnd);
```

只是这次我们不是调用 UIAddToolbar()或 UIAddStatusBar()，而是调用 UIAddChildWindowContainer()，它告诉 CUpdateUI 我们的对话框含有需要 updating 的字窗口，只要看看 OnIdle()，你会怀疑少了写什么：

```
BOOL CMainDlg::OnIdle()
{
    return FALSE;
}
```

你可能猜想这里应该调用另一个 CUpdateUI 的方法做一些实在的 updating 工作，你是对的，应该是这样的，向导在 OnIdle()中漏掉了一行代码，现在加上：

```
BOOL CMainDlg::OnIdle()
{
    UIUpdateChildWindows();
    return FALSE;
}
```

为了演示 UI updating，我们设定鼠标点击左边的位图按钮，使得右边的按钮变得可用或禁用。先在 update UI 链中添加一个消息入口，使用 UPDUI_CHILDWINDOW 标志表示此入口是子窗口类型：

```
    BEGIN_UPDATE_UI_MAP(CMainDlg)
        UPDATE_ELEMENT(IDC_ALYSON_BMPBTN, UPDUI_CHILDWINDOW)
    END_UPDATE_UI_MAP()
```

在左边的按钮的单击事件处理中，我们调用 UIEnable()来翻转另一个按钮的使能状态：

```
void CMainDlg::OnAlysonODBtn ( UINT uCode, int nID, HWND hwndCtrl )
{
static bool s_bBtnEnabled = true;

    s_bBtnEnabled = !s_bBtnEnabled;
    UIEnable ( IDC_ALYSON_BMPBTN, s_bBtnEnabled );
}
```

DDV

WTL 的对话框数据验证(DDV)比 MFC 简单一些，在 MFC 中你需要分别使用 DDX(对话框数据交换)宏和 DDV(对话框数据验证)宏，在 WTL 中只需一个宏就可以了，WTL 包含基本的数据验证

支持，在 DDV 链中可以使用三个宏：

DDX_TEXT_LEN

和 DDX_TEXT 一样，只是还要验证字符串的长度(不包含结尾的空字符)小于或等于限制长度。

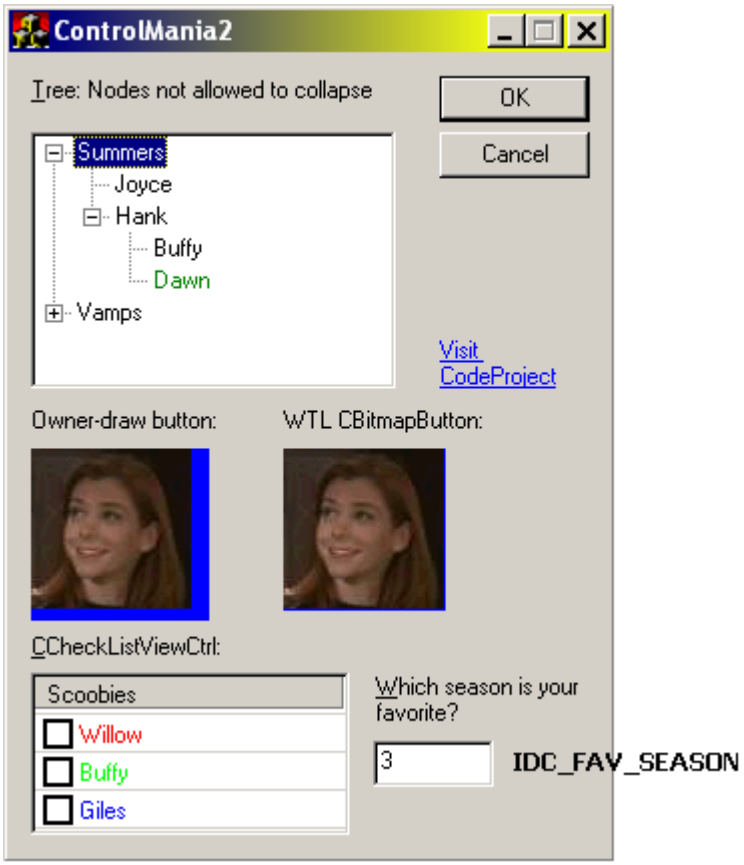
DDX_INT_RANGE and DDX_UINT_RANGE

和 DDX_INT，DDX_UINT 一样，还加了对数字的最大最小值的验证。

DDX_FLOAT_RANGE

除了像 DDX_FLOAT 一样完成数据交换之外，还验证数字的最大最小值。

ControlMania2 有一个 ID 是 IDC_FAV_SEASON 的 edit box，它和成员变量 m_nSeason 相关联。



由于有效的值是 1 到 7，所以使用这样的数据验证宏：

```
BEGIN_DDX_MAP(CMainDlg)
//...
    DDX_INT_RANGE(IDC_FAV_SEASON, m_nSeason, 1, 7)
END_DDX_MAP()
```

OnOK()调用 DoDataExchange()获得 season 的数值，并验证是在 1 到 7 之间。

处理 DDV 验证失败

如果控件的数据验证失败，CWinDataExchange 会调用重载函数 OnDataValidateError()，默认到处理是驱动 PC 喇叭发出声音，你可能想给出更友好的错误指示。OnDataValidateError()的函数原型是：

```
void OnDataValidateError ( UINT nCtrlID, BOOL bSave, _XData& data );
```

_XData 是一个 WTL 的内部数据结构，CWinDataExchange 根据输入的数据和允许的数据范围填充这个数据结构。下面是这个数据结构的定义：

```
struct _XData
```

```
{
    _XDataType nDataType;
    union
    {
        _XTextData textData;
        _XIntData intData;
        _XFloatData floatData;
    };
};
```

nDataType 指示联合中的三个成员那个是有意义的，nDataType 的取值可以是：

```
enum _XDataType
{
    ddxDataNull = 0,
    ddxDataText = 1,
    ddxDataInt = 2,
    ddxDataFloat = 3,
    ddxDataDouble = 4
};
```

在我们的例子中，nDataType 的值是 ddxDataInt，这表示_XData 中的_XIntData 成员是有效的，_XIntData 是个简单的数据结构：

```
struct _XIntData
{
    long nVal;
    long nMin;
    long nMax;
};
```

我们重载 OnDataValidateError()函数，显示错误信息并告诉用户允许的数值范围：

```
void CMainDlg::OnDataValidateError ( UINT nCtrlID, BOOL bSave, _XData&
data )
{
CString sMsg;

    sMsg.Format ( _T("Enter a number between %d and %d"),
                data.intData.nMin, data.intData.nMax );

    MessageBox ( sMsg, _T("ControlMania2"), MB_ICONEXCLAMATION );

    ::SetFocus ( GetDlgItem(nCtrlID) );
}
```

_XData 中的另外两个结构_XTextData 和_XFloatData 的定义在 atlddx.h 中，感兴趣的话可以打开这个文件查看一下。

改变对话框的大小

WTL引起我的注意的第一件事是对可调整大小对话框的内建的支持。在这之前我曾写过一篇关于这个主题的文章，详情请参考这篇文章。简单的说就是将CDialogResize类添加到对话框

的集成列表，在OnInitDialog()中调用 DlgResize_Init()，然后将消息链入CDialogResize。

继续

下一章，我将介绍如何在对话框中使用ActiveX控件和如何处理控件触发的事件。

参考

Using WTL's Built-in Dialog Resizing Class - Michael Dunn
Using DDX and DDV with WTL - Less Wright

第六部分　使用 ActiveX 控件

在第六章，我将介绍 ATL 对在对话框中使用 ActiveX 控件的支持，由于 ActiveX 控件就是 ATL 的专业，所以 WTL 没有添加其他的辅助类。 不过，在 ATL 中使用 ActiveX 控件与在 MFC 中有很大的不同，所以需要重点介绍。我将介绍如何包容一个控件并处理控件的事件，开发 ATL 应用程序相 对于 MFC 的类向导来说有点不方便。在 WTL 程序中自然可以使用 ATL 对包容 ActiveX 控件的支持。

例子工程演示如何使用 IE 的浏览器控件，我选择浏览器控件有两个好处：

每台计算机都有这个控件，并且

它有很多方法和事件，是个用来做演示的好例子。

我当然无法与那些花了大量时间编写基于 IE 浏览器控件的定制浏览器的人相比，不过，当你读完本篇文章之后，你就知道如何开始编写自己定制的浏览器！

从使用向导开始

创建工程

WTL 的向导可以创建一个支持包容 ActiveX 控件的程序，我将开始一个名为 IEHoster 的新工程。我们像上一章一样使用无模式对话框，只是这次要选上支持 ActiveX 控件包容(Enable ActiveX Control Hosting)，如下图：

选上这个 check box 将使我们的对话框从 CAxDialogImpl 派生，这样就可以包容 ActiveX 控件。在向导的第二页还有一个名为包容 ActiveX 控件的 check box，但是选择这个好像对最后的结果没有影响，所以在第一页就可以点击“Finish”结束向导。

向导生成的代码

在这一节我将介绍一些以前没有见过的新代码(由向导生成的)，下一节介绍 ActiveX 包容类的细节。

首先要看的文件是 stdafx.h，它包含了这些文件：

```
#include <atlbase.h>
#include <atlapp.h>

extern CAppModule _Module;

#include <atlcom.h>
#include <atlhost.h>
#include <atlwin.h>
#include <atlctl.h>
// .. other WTL headers ...
```

atlcom.h 和 atlhost.h 是很重要的两个，它们含有一些 COM 相关类的定义(比如智能指针 CComPtr)，还有可以包容控件的窗口类。

接下来看看 maindlg.h 中声明的 CMainDlg 类：

```
class CMainDlg : public CAxDialogImpl<CMainDlg>,
                 public CUpdateUI<CMainDlg>,
                 public CMessageFilter, public CIdleHandler
```

CMainDlg 现在是从 CAxDialogImpl 类派生的，这是使对话框支持包容 ActiveX 控件的第一步。

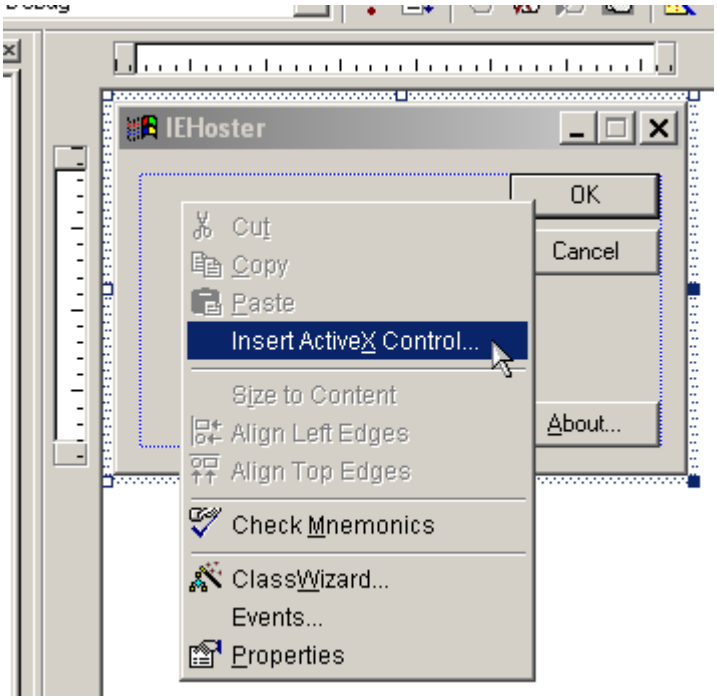
最后，看看 WinMain()中新加的一行代码：

```
int WINAPI _tWinMain(...)
{
//...
    _Module.Init(NULL, hInstance);

    AtlAxWinInit();

    int nRet = Run(lpstrCmdLine, nCmdShow);

    _Module.Term();
    return nRet;
}
```

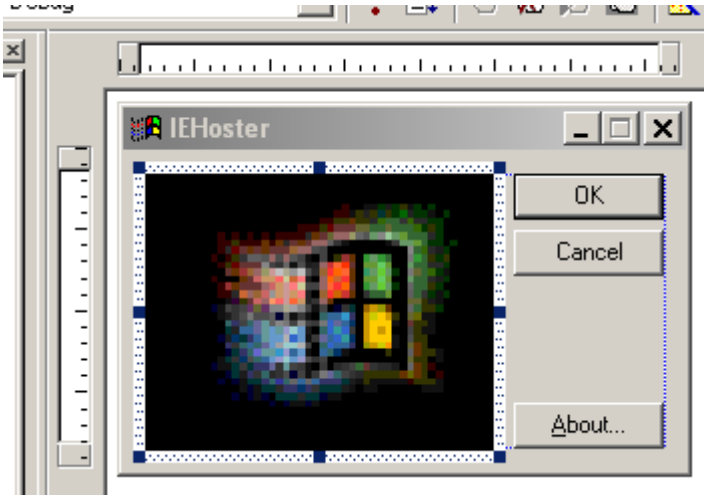
AtlAxWinInit()注册了一个类名未 AtlAxWin 的窗口类，ATL 用它创建 ActiveX 控件的包容窗口。

使用资源编辑器添加控件

和 MFC 的程序一样，ATL 也可以使用资源编辑器向对话框添加控件。首先，在对话框编辑器上点击鼠标右键，在弹出的菜单中选择“Insert ActiveX control”：



VC 将系统安装的控件显示在一个列表中，滚动列表选择“Microsoft Web Browser”，单击 Insert 按钮将控件加入到对话框中。查看控件的属性，将 ID 设为 IDC_IE。对话框中的控件显示应该是这个样子的：



如果现在编译运行程序，你会看到对话框中的浏览器控件，它将显示一个空白页，因为我们还没有告诉它到哪里去。

在下一节，我将介绍与创建和包容 ActiveX 控件有关的 ATL 类，同时我们也会明白这些类是如何与浏览器交换信息的。

ATL 中使用控件的类

在对话框中使用 ActiveX 控件需要两个类协同工作：CAxDialogImpl 和 CAxWindow。它们处理所有控件容器必须实现的接口方法，提供通用的功能函数，例如查询控件的某个特殊的 COM 接口。

CAxDialogImpl

第一个类是 CAxDialogImpl，你的对话框要能够包容控件就必须从 CAxDialogImpl 类派生而不是从 CDialogImpl 类派 生。CAxDialogImpl 类重载了 Create()和 DoModal()函数，这两个函数分别被全局函数 AtlAxCreateDialog() 和 AtlAxDialogBox()调用。既然

IEHoster 对话框是由 Create()创建的，我们看看 AtlAxCreateDialog()到底 做了什么工作。

AtlAxCreateDialog()使用辅助类_DialogSplitHelper 装载对话框资源，这个辅助类遍历所以对话框的控件，查找由资源编辑器创建的特殊的入口，这些特殊的入口表示这是一个 ActiveX 控件。例如，下面是 IEHoster.rc 文件中浏览器控件的入口：

```
CONTROL "",IDC_IE,"{8856F961-340A-11D0-A96B-00C04FD705A2}",
        WS_TABSTOP,7,7,116,85
```

第一个参数是窗口文字(空字符串)，第二个是控件的 ID，第三个是窗口的类名。_DialogSplitHelper:: SplitDialogTemplate()函数找到以'{'开始的窗口类名时就知道这是一个 ActiveX 控件的入口。它在内存中创建了一个临时对话框 模板，在这个新模板中这些特殊的控件入口被创建的 AtlAxWin 窗口代替，新的入口是在内存中的等价体：

```
CONTROL "{8856F961-340A-11D0-A96B-00C04FD705A2}",IDC_IE,"AtlAxWin",
        WS_TABSTOP,7,7,116,85
```

结果就是创建了一个相同 ID 的 AtlAxWin 窗口，窗口的标题是 ActiveX 控件的 GUID。所以你调用 GetDlgItem(IDC_IE)返回的值是 AtlAxWin 窗口的句柄而不是 ActiveX 控件本身。SplitDialogTemplate()函数完成工作后，AtlAxCreateDialog()接着调用 CreateDialogIndirectParam()函数使用修改后的模板创建对话框。

AtlAxWin and CAxWindow

正如上面讲到的，AtlAxWin 实际上是 ActiveX 控件的宿主窗口，AtlAxWin 还会用到一个特殊的窗口接口类： CAxWindow，当 AtlAxWin 从模板创建一个对话框后，AtlAxWin 的窗口处理过程，AtlAxWindowProc()，就会处理 WM_CREATE 消息并创建相应的 ActiveX 控件。ActiveX 控件还可以在运行其间动态创建，不需要对话框模板，我会在后面介绍这种方法。WM_CREATE 的消息处理函数调用全局函数 AtlAxCreateControl()，将 AtlAxWin 窗口的窗口标题作为参数传递给该函数，大家应该记得那实际就是浏览器控件的 GUID。AtlAxCreateControl()有会调用一堆其他函数，不过最终会用到 CreateNormalizedObject()函数，这个函数将窗口 标题转换成 GUID，并最终调用 CoCreateInstance()创建 ActiveX 控件。

由于 ActiveX 控件是 AtlAxWin 的子窗口，所以对话框不能直接访问控件，当然 CAxWindow 提供了这些方法通控件通信，最常用的一个 是 QueryControl()，这个方法调用控件的 QueryInterface()方法。例如，你可以使用 QueryControl()从浏览器控件 得到 IWebBrowser2 接口，然后使用这个接口将浏览器引导到指定的 URL。

调用控件的方法

既然我们的对话框有一个浏览器控件，我们可以使用 COM 接口与之交互。我们做得第一件事情就是使用 IWebBrowser2 接口将其引导到一个新 URL 处。在 OnInitDialog()函数中，我们将一个 CAxWindow 变量与包容控件的 AtlAxWin 联系起来。

```
CAxWindow wndIE = GetDlgItem(IDC_IE);
```

然后声明一个 IWebBrowser2 的接口指针并查询浏览器控件的这个接口，使用 CAxWindow::QueryControl()：

```
CComPtr<IWebBrowser2> pWB2;
HRESULT hr;
hr = wndIE.QueryControl ( &pWB2 );
```

QueryControl()调用浏览器控件的 QueryInterface()方法，如果成功就会返回 IWebBrowser2 接口，我们可以调用 Navigate()：

```
    if ( pWB2 )
```

```
        {
        CComVariant v;  // empty variant

        pWB2->Navigate ( CComBSTR("http://www.codeproject.com/"),
                    &v, &v, &v, &v );
        }
```

响应控件触发的事件

从浏览器控件得到接口非常简单，通过它可以单向的与控件通信。通常控件也会以事件的形式与外界通信，ATL 有专用的类包装连接点和事件相应，所以我们可以从控件接收到这些事件。为使用对事件的支持需要做四件事：

将 CMainDlg 变成 COM 对象

添加 IDispEventSimpleImpl 到 CMainDlg 的继承列表

填写事件映射链，它指示哪些事件需要处理

编写事件响应函数

CMainDlg 的修改

将 CMainDlg 转变成 COM 对象的原因是事件相应是基于 IDispatch 的，为了让 CMainDlg 暴露这个接口，它必须是个 COM 对象。 IDispEventSimpleImpl 提供了 IDispatch 接口的实现和建立连接点所需的处理函数，当事件发生时 IDispEventSimpleImpl 还调用我们想要接收的事件的处理函数。

以下的类需要添加到 CMainDlg 的集成列表中，同时 COM_MAP 列出了 CMainDlg 暴露的接口：

```
#include <exdisp.h>    // browser control definitions
#include <exdispid.h>  // browser event dispatch IDs

class CMainDlg : public CAxDialogImpl<CMainDlg>,
                 public CUpdateUI<CMainDlg>,
                 public CMessageFilter, public CIdleHandler,
                 public CComObjectRootEx<CComSingleThreadModel>,
                 public CComCoClass<CMainDlg>,
                 public       IDispEventSimpleImpl<37,       CMainDlg,
&DIID_DWebBrowserEvents2>
{
...
  BEGIN_COM_MAP(CMainDlg)
    COM_INTERFACE_ENTRY2(IDispatch, IDispEventSimpleImpl)
  END_COM_MAP()
};
```

CComObjectRootEx 类 CComCoClass 共同使 CMainDlg 成为一个 COM 对象，IDispEventSimpleImpl 的模板参数是事件的 ID，我们的类名和连接点接口的 IID。事件 ID 可以是任意正数，连接点对象的 IID 是 DIID_DWebBrowserEvents2，可以在浏览器控件的相关文档中找到这些参数，也可以查看 exdisp.h。

填写事件映射链

下一步是给 CMainDlg 添加事件映射链，这个映射链将我们感兴趣的事件和我们的处理函数联系起来。我们要看的第一个事件是 DownloadBegin，当浏览器开始下载一个页面时就会触发这个事件，我们响应这个事件显示“please wait”信息给用户，让用户知道浏览器正在忙。

在 MSDN 中可以查到 DWebBrowserEvents2::DownloadBegin 事件的原型

```
  void DownloadBegin();
```

这个事件没有参数，也不需要返回值。为了将这个事件的原型转换成事件响应链，我们需要写一个_ATL_FUNC_INFO 结构，它包含返回值，参数 的个数和参数类型。由于事件是基于 IDispatch 的，所以所有的参数都用 VARIANT 表示，这个数据结构的描述相当长(支持很多个数据类型)，以下 是常用的几个：

VT_EMPTY: void

VT_BSTR: BSTR 格式的字符串

VT_I4: 4 字节有符号整数，用于 long 类型的参数

VT_DISPATCH: IDispatch*

VT_VARIANT>: VARIANT

VT_BOOL: VARIANT_BOOL (允许的取值是 VARIANT_TRUE 和 VARIANT_FALSE)

另外，标志 VT_BYREF 表示将一个参数转换成相应的指针。例如，VT_VARIANT|VT_BYREF 表示 VARIANT*类型。下面是_ATL_FUNC_INFO 的定义：

```
#define _ATL_MAX_VARTYPES 8

struct _ATL_FUNC_INFO
{
    CALLCONV cc;
    VARTYPE  vtReturn;
    SHORT    nParams;
    VARTYPE  pVarTypes[_ATL_MAX_VARTYPES];
};
```

参数：

cc

我们的事件响应函数的调用方式约定，这个参数必须是 CC_STDCALL，表示是__stdcall 方式

vtReturn

事件响应函数的返回值类型

nParams

事件带的参数个数

pVarTypes

相应的参数类型，按从左到右的顺序

了解这些之后，我们就可以填写 DownloadBegin 事件处理的_ATL_FUNC_INFO 结构：

```
_ATL_FUNC_INFO DownloadInfo = { CC_STDCALL, VT_EMPTY, 0 };
```

现在，回到事件响应链，我们为每一个我们想要处理的事件添加一个 SINK_ENTRY_INFO 宏，下面是处理 DownloadBegin 事件的宏：

```
class CMainDlg : public ...
{
...
  BEGIN_SINK_MAP(CMainDlg)
    SINK_ENTRY_INFO(37, DIID_DWebBrowserEvents2, DISPID_DOWNLOADBEGIN,
                    OnDownloadBegin, &DownloadInfo)
  END_SINK_MAP()
```

```
};
```

这个宏的参数是事件的 ID(37，与我们在 IDispEventSimpleImpl 的继承列表中使用的 ID 一样)，事件接口的 IID，事件的 dispatch ID(可以在 MSDN 或 exdispid.h 头文件中查到),事件处理函数的名字和指向描述这个事件处理的_ATL_FUNC_INFO 结构的指针。

编写事件处理函数

好了，等了这么长时间(吹个口哨！)，我们可以写事件处理函数了：

```
void __stdcall CMainDlg::OnDownloadBegin()
{
  // show "Please wait" here...
}
```

现在来看一个复杂一点的事件，比如 BeforeNavigate2，这个事件的原型是：

```
void BeforeNavigate2 (
    IDispatch* pDisp, VARIANT* URL, VARIANT* Flags,
    VARIANT* TargetFrameName, VARIANT* PostData,
    VARIANT* Headers, VARIANT_BOOL* Cancel );
```

此方法有 7 个参数，对于 VARIANT 类型参数可以从 MSDN 查到它到底传递的是什么类型的数据，我们感兴趣的是 URL，是一个 BSTR 类型的字符串。

描述 BeforeNavigate2 事件的_ATL_FUNC_INFO 结构是这样的：

```
_ATL_FUNC_INFO BeforeNavigate2Info =
    { CC_STDCALL, VT_EMPTY, 7,
        { VT_DISPATCH, VT_VARIANT|VT_BYREF, VT_VARIANT|VT_BYREF,
          VT_VARIANT|VT_BYREF,                    VT_VARIANT|VT_BYREF,
VT_VARIANT|VT_BYREF,
          VT_BOOL|VT_BYREF }
};
```

和前面一样，返回值类型是 VT_EMPTY 表示没有返回值，nParams 是 7，表示有 7 个参数。接着是参数类型数组，这些类型前面介绍过了，例如 VT_DISPATCH 表示 IDispatch*。

事件响应链的入口与前面的例子很相似：

```
  BEGIN_SINK_MAP(CMainDlg)
    SINK_ENTRY_INFO(37, DIID_DWebBrowserEvents2, DISPID_DOWNLOADBEGIN,
                    OnDownloadBegin, &DownloadInfo)
    SINK_ENTRY_INFO(37,                      DIID_DWebBrowserEvents2,
DISPID_BEFORENAVIGATE2,
                    OnBeforeNavigate2, &BeforeNavigate2Info)
  END_SINK_MAP()
```

事件处理函数是这个样子：

```
void __stdcall CMainDlg::OnBeforeNavigate2 (
    IDispatch* pDisp, VARIANT* URL, VARIANT* Flags,
    VARIANT* TargetFrameName, VARIANT* PostData,
    VARIANT* Headers, VARIANT_BOOL* Cancel )
{
CString sURL = URL->bstrVal;

  // ... log the URL, or whatever you'd like ...
```

```
}
```

我打赌你现在是越来越喜欢 ClassWizard 了，因为当你向 MFC 的对话框插入一个 ActiveX 控件时 ClassWizard 自动为你完成了所有工作。

将 CMainDlg 转换成对象需要注意几件事情，首先必须修改全局函数 Run()，现在 CMainDlg 是个 COM 对象，我们必须使用 CComObject 创建 CMainDlg：

```
int Run(LPTSTR /*lpstrCmdLine*/ = NULL, int nCmdShow = SW_SHOWDEFAULT)
{
    CMessageLoop theLoop;
    _Module.AddMessageLoop(&theLoop);

CComObject<CMainDlg> dlgMain;

    dlgMain.AddRef();

    if ( dlgMain.Create(NULL) == NULL )
        {
        ATLTRACE(_T("Main dialog creation failed!\n"));
        return 0;
        }

    dlgMain.ShowWindow(nCmdShow);

    int nRet = theLoop.Run();

    _Module.RemoveMessageLoop();
    return nRet;
}
```

另一个可替代的方法是不使用 CComObject，而使用 CComObjectStack 类，并删除 dlgMain.AddRef()这一行代码， CComObjectStack 对 IUnknown 的三个方法的实现有些微不足道(它们只是简单的从函数返回)，因为它们不是必需的——这样的 COM 对象可以忽略对引用的计数，因为它们仅仅是创建在栈中的临时对象。

当然这并不是完美的解决方案， CComObjectStack 用于短命的临时对象，不幸的是只要调用它的任何一个 IUnknown 方法都会引发断言错误。因为 CMainDlg 对象在开 始监听事件时会调用 AddRef，所以 CComObjectStack 不适用于这种情况。

解决这个问题要么坚持使用 CComObject，要么从 CComObjectStack 派生一个 CComObjectStack2 类，允许对 IUnknow 方法调用。 CComObject 的那个不必要的引用计数并无大碍——人们不会注意到它的发生——但是如果你必须节省那个 CPU 时钟周期的话，你可以使用本章的例子工 程代码中的 CComObjectStack2 类。

回顾例子工程

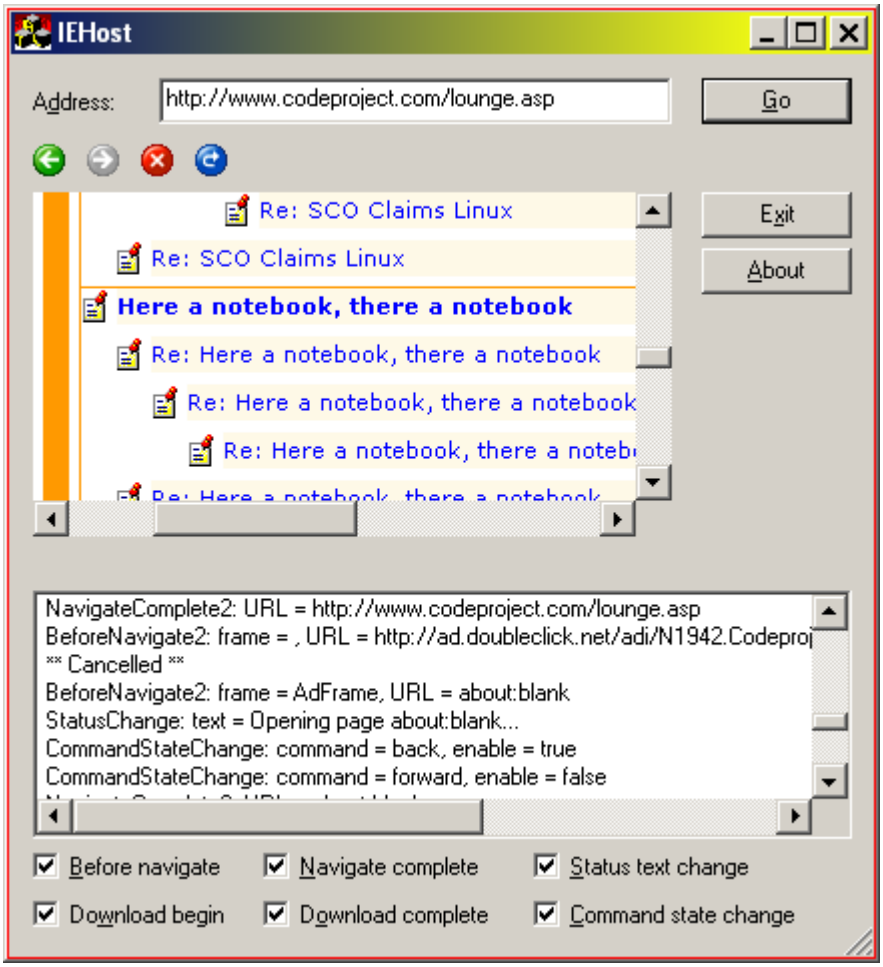
现在我们已经看到事件响应如何工作了，再来看看完整的 IEHoster 工程，它包容了一个浏览器控件并响应了 6 个事件，它还显示了一个事件列表，你会对浏览器如何使用它们提供带进度条的界面有个感性的认识，程序处理了以下几个事件：

BeforeNavigate2 和 NavigateComplete2：这些事件让程序可以控制 URL 的导航，如果你响应了 BeforeNavigate2 事件，你可以在事件的处理函数中取消导航。

DownloadBegin 和 DownloadComplete：程序使用这些事件控制“wait”消息，这表示浏览器正在工作。一个更优美的程序会像 IE 一样在此期间使用一段动画。
CommandStateChange：这个事件告诉程序向前和向后导航命令何时可用，应用程序将相应的按钮变为可用或不可用。
StatusTextChange：这个事件会在几种情况下触发，例如鼠标移到一个超链接上。这个事件发送一个字符串，应用程序响应这个事件，将这个字符串显示在浏览器窗口下的静态控件上。
程序有四个按钮控制浏览器工作：向后，向前，停止和刷新，它们分别调用 IWebBrowser2 相应的方法。
事件和伴随事件发送的数据都被记录在列表控件中，你可以看到事件的触发，你还可以关闭一些事件记录而仅仅观察其中的一辆个事件。为了演示事件处理的 重要作用，我们在 BeforeNavigate2 事件处理函数中检查 URL，如果发现“doubleclick.net”就取消导航。广告和弹出窗口过滤 器等一些 IE 的插件使用的就是这个方法而不是 HTTP 代理，下面就是做这些检查的代码。

```
void __stdcall CMainDlg::OnBeforeNavigate2 (
    IDispatch* pDisp, VARIANT* URL, VARIANT* Flags,
    VARIANT* TargetFrameName, VARIANT* PostData,
    VARIANT* Headers, VARIANT_BOOL* Cancel )
{
USES_CONVERSION;
CString sURL;

    sURL = URL->bstrVal;

    // You can set *Cancel to VARIANT_TRUE to stop the
    // navigation from happening. For example, to stop
    // navigates to evil tracking companies like doubleclick.net:
    if ( sURL.Find ( _T("doubleclick.net") ) > 0 )
        *Cancel = VARIANT_TRUE;
}
```

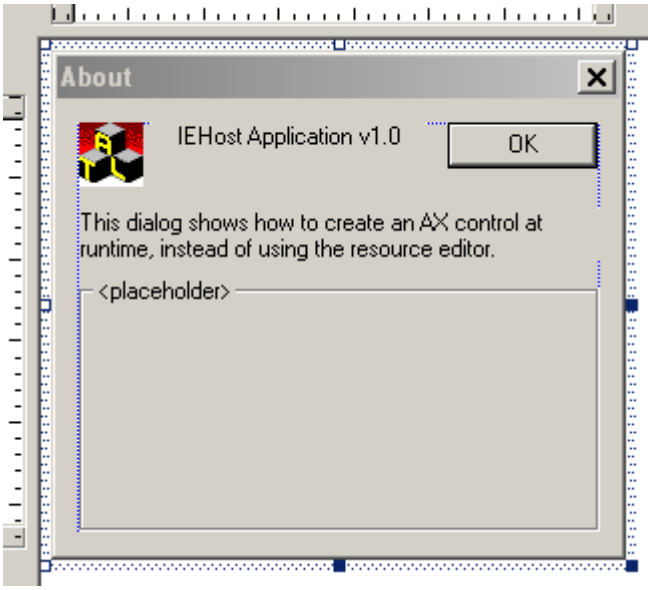
下面就是我们的程序工作起来的样子：



IEHoster 还使用了前几章介绍过得类：CBitmapButton(用于浏览器控制按钮)，CListViewCtrl(用于事件记录)，DDX (跟踪 checkbox 的状态)和 CDialogResize.

运行时创建 ActiveX 控件

出了使用资源编辑器，还可以在运行其间动态创建 ActiveX 控件。About 对话框演示了这种技术。对话框编辑器预先放置了一个 group box 用于浏览器控件的定位：



在 OnInitDialog()函数中我们使用 CAxWindow 创建了一个新 AtlAxWin，它定位于我们

预先放置好的 group box 的位置上(这个 group box 随后被销毁)：

```
LRESULT CAboutDlg::OnInitDialog(...)
{
CWindow wndPlaceholder = GetDlgItem ( IDC_IE_PLACEHOLDER );
CRect rc;
CAxWindow wndIE;

   // Get the rect of the placeholder group box, then destroy
   // that window because we don't need it anymore.
   wndPlaceholder.GetWindowRect ( rc );
   ScreenToClient ( rc );
   wndPlaceholder.DestroyWindow();

   // Create the AX host window.
   wndIE.Create ( *this, rc, _T(""),
                  WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN );
```

接下来我们用 CAxWindow 方法创建一个 ActiveX 控件，有两个方法可以选择：CreateControl()和 CreateControlEx()。CreateControlEx()用一个额外的参数返回接口指针，这样就不需要再调用 QueryControl ()函数。我们感兴趣的两个参数是第一个和第四个参数，第一个参数是字符串形式的浏览器控件的 GUID，第四个参数是一个 IUnknown*类型的指针， 这个指针指向 ActiveX 控件的 IUnknown 接口。创建控件后就可以查询 IWebBrowser2 接口，然后就可以像前面一样控制它导航到某个 URL。

```
CComPtr<IUnknown> punkCtrl;
CComQIPtr<IWebBrowser2> pWB2;
CComVariant v;

   // Create the browser control using its GUID.
   wndIE.CreateControlEx ( L"{8856F961-340A-11D0-A96B-00C04FD705A2}",
                           NULL, NULL, &punkCtrl );

   // Get an IWebBrowser2 interface on the control and navigate to a
page.
   pWB2 = punkCtrl;
   pWB2->Navigate ( CComBSTR("about:mozilla"), &v, &v, &v, &v );
}
```

对于有 ProgID 的 ActiveX 控件可以传递 ProgID 给 CreateControlEx()，代替 GUID。例如，我们可以这样创建浏览器控件：

```
   // 使用控件的 ProgID: 创建 Shell.Explorer:
   wndIE.CreateControlEx ( L"Shell.Explorer", NULL,
                           NULL, &punkCtrl );
```

CreateControl()和 CreateControlEx()还有一些重载函数用于一些使用浏览器的特殊情况，如果你的应用程序使用 WEb 页面作为 HTML 资源，你可以将资源 ID 作为第一个参数，ATL 会创建浏览器控件并导航到这个资源。IEHoster 包含一个 ID 为 IDR_ABOUTPAGE 的 WEB 页面资源，我们在 About 对话框中使用这些代码显示这个页面：

```
wndIE.CreateControl ( IDR_ABOUTPAGE );
```

这是显示结果：

例子代码对上面提到的三个方法都用到了，你可以查看 CAboutDlg::OnInitDialog()中的注释和未注释的代码，看看它们分别是如何工作的。

键盘事件处理

最后一个但是非常重要的细节是键盘消息。ActiveX 控件的键盘处理非常复杂，因为控件和它的宿主程序必须协同工作以确保控件能够看到它感兴趣的 消息。例如，浏览器控件允许你使用 TAB 键在链接之间切换。MFC 自己处理了所有工作，所以你永远不会意识到让键盘完美并正确的工作需要多么大的工作量。

不幸的是向导没有为基于对话框的程序生成键盘处理代码，当然，如果你使用 Form View 作为视图类的 SDI 程序，你会看到必要的代码已经被添加到 PreTranslateMessage()中。当程序从消息队列中得到鼠标或键盘消息 时，就使用 ATL 的 WM_FORWARDMSG 消息将此消息传递给当前拥有焦点的控件。它们通常不作什么事情，但是如果是 ActiveX 控件，WM_FORWARDMSG 消息最终被送到包容这个控件的 AtlAxWin，AtlAxWin 识别 WM_FORWARDMSG 消息并采取必要的措施看看是 否控件需要亲自处理这个消息。

如果拥有焦点的窗口没有识别 WM_FORWARDMSG 消息，PreTranslateMessage()就会接着调用 IsDialogMessage()函数，使得像 TAB 这样的标准对话框的导航键能正常工作。

例子工程的 PreTranslateMessage()函数中含有这些必需的代码，由于 PreTranslateMessage()只在无模式对话框中有效，所以如果你想在基于对话框的应用程序中正确使用键盘就必须使用无模式对话框。

继续

在下一章，我们将回到框架窗口并介绍如何使用分隔窗口。

第七部分　使用分隔窗口

随着使用两个分隔的视图管理文件系统的资源管理器在 Windows 95 中第一次出现，分隔窗口逐渐成为一种流行的界面元素。MFC 也有一个复杂的功能强大的分隔窗口类，但是要掌握它的用法确实有点难，并且它和文档/视图 框架联系紧密。在第七章我将介绍 WTL 的分隔窗口，它比 MFC 的分隔窗口要简单一些。WTL 的分隔窗口没有 MFC 那么多特性，但是易于使用和扩展。

本章的例子工程是用WTL重写的ClipSpy，如果你对这个程序不太熟悉，现在可以快速浏览一下本章内容，因为我只是复制了ClipSpy的功能而没用深入的解释它是如何工作的，毕竟这篇文章的重点是分隔窗口，不是剪贴板。

WTL 的分隔窗口

头文件 atlsplit.h 含有所有 WTL 的分隔窗口类，一共有三个类：CSplitterImpl，CSplitterWindowImpl 和 CSplitterWindowT，不过你通常只会用到其中的一个。下面将介绍这些类和它们的基本方法。

相关的类

CSplitterImpl 是一个有两个参数的模板类，一个是窗口界面类的类名，另一个是布尔型变量表示分隔窗口的方向：true 表示垂直方向， false 表示水平方向。CSplitterImpl 类包含了几乎所有分隔窗口的实现代码，它的许多方法是可重载的，重载这些方法可以自己绘制分隔条的外 观或者实现其它的效果。CSplitterWindowImpl 类是从 CWindowImpl 和 CSplitterImpl 两个类派生出来的，但是它的代 码不多，有一个空的 WM_ERASEBKGND 消息处理函数和一个 WM_SIZE 处理函数用于重新定位分隔窗口。

最后一个是 CSplitterWindowT 类，它从 CSplitterImpl 类派生，它的窗口类名是“WTL_SplitterWindow”。还有两个自定义数据类型通常用来取代上面的三个类：CSplitterWindow 用于垂直分隔窗口， CHorSplitterWindow 用于水平分隔窗口。

创建分割窗口

由于 CSplitterWindow 是从 CWindowImpl 类派生的，所以你可以像创建其他子窗口那样创建分隔窗口。分隔窗口将存在于整个主框 架窗口的生命周期，应该在 CMainFrame 类添加一个 CSplitterWindow 类型的变量。在 CMainFrame::OnCreate ()函数内，你可以将分隔窗口作为主窗口的子窗口创建，然后将其设置为主窗口的客户区窗口：

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
// ...
const DWORD dwSplitStyle = WS_CHILD | WS_VISIBLE |
                           WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
          dwSplitExStyle = WS_EX_CLIENTEDGE;

   m_wndSplit.Create ( *this, rcDefault, NULL,
                       dwSplitStyle, dwSplitExStyle );

   m_hWndClient = m_wndSplit;
}
```

创建分隔窗口之后，你就可以为每个窗格指定窗口或者做其他必要的初始化工作。

基本方法

```
bool SetSplitterPos(int xyPos = -1, bool bUpdate = true)
int GetSplitterPos()
```

可以调用 SetSplitterPos()函数设置分隔条的位置，这个位置表示分割条距离分隔窗口的上边界(水平分隔窗口)或左边界(垂直分隔窗 口)有多少个象素点。你可以使用默认值－1 将分隔条设置到分隔窗口的中间，使两个窗格大小相同，通常传递 true 给 bUpdate 参数表示在移动分隔条之 后相应的改变两个窗格的大小。GetSplitterPos()返回当前分隔条的位置，这个位置也是相对于分隔窗口的上边界或左边界。

```
bool SetSinglePaneMode(int nPane = SPLIT_PANE_NONE)
int GetSinglePaneMode()
```

调用 SetSinglePaneMode()函数可以改变分隔窗口的模式使单窗格模式还是双窗格模式，在单窗格模式下，只有一个窗格使可见的并且隐 藏了分隔条，这和 MFC 的动态分隔窗口相似(只是没有那个小钳子形状的手柄，它用于重新分隔分隔窗口)。对于 nPane 参数可用的值是 SPLIT_PANE_LEFT，SPLIT_PANE_RIGHT，SPLIT_PANE_TOP，SPLIT_PANE_BOTTOM，和 SPLIT_PANE_NONE，前四个指示显示那个窗格(例如，使用 SPLIT_PANE_LEFT 参数将显示左边的窗格，隐藏右边的窗格)，使用 SPLIT_PANE_NONE 表示两个窗格都显示。GetSinglePaneMode()返回五个 SPLIT_PANE_*值中的一个表示当前的模 式。

```
DWORD SetSplitterExtendedStyle(DWORD dwExtendedStyle, DWORD dwMask = 0)
DWORD GetSplitterExtendedStyle()
```

分隔窗口有自己的样式用于控制当整个分隔窗口改变大小时如何移动分隔条。有以下几种样式：

SPLIT_PROPORTIONAL：两个窗格一起改变大小

SPLIT_RIGHTALIGNED：右边的窗格保持大小不变，只改变左边的窗格大小

SPLIT_BOTTOMALIGNED：下部的窗格保持大小不变，只改变上边的窗格大小

如果既没有指定 SPLIT_PROPORTIONAL，也没有指定 SPLIT_RIGHTALIGNED/SPLIT_BOTTOMALIGNED，则分隔窗口会变成左对齐或上对齐。如果将 SPLIT_PROPORTIONAL 和 SPLIT_RIGHTALIGNED/SPLIT_BOTTOMALIGNED 一起使用，则优先选用 SPLIT_PROPORTIONAL 样式。

还有一个附加的样式用来控制分隔条是否可以被用户移动：

SPLIT_NONINTERACTIVE：分隔条不能被移动并且不相应鼠标

扩展样式的默认值是 SPLIT_PROPORTIONAL。

```
bool SetSplitterPane(int nPane, HWND hWnd, bool bUpdate = true)
void SetSplitterPanes(HWND hWndLeftTop, HWND hWndRightBottom, bool bUpdate = true)
HWND GetSplitterPane(int nPane)
```

可以调用 SetSplitterPane()为分隔窗口的窗格指派子窗口，nPane 是一个 SPLIT_PANE_*类型的值，表示设置拿一个窗 格。hWnd 是子窗口的窗口句柄。你可以使用 SetSplitterPane()将一个子窗口同时指定给两个窗格，对于 bUpdate 参数通常使用默认值，也就是告诉分隔窗口立即调整子窗口的大小以适应窗格的大小。可以调用 GetSplitterPane()得到某个窗格的子窗口句柄，如果窗格没有指派 子窗口则 GetSplitterPane()返回 NULL。

```
bool SetActivePane(int nPane)
int GetActivePane()
```

SetActivePane()函数将分隔窗口中的某个子窗口设置为当前焦点窗口，nPane 是 SPLIT_PANE_*类型的值，表示需要激活哪 个窗格，这个函数还可以设置默认的活动窗格(后

面介绍)。GetActivePane()函数查看所有拥有焦点的窗口，如果拥有焦点的窗口是窗格或窗格的 子窗口就返回一个 SPLIT_PANE_*类型的值，表示是哪个窗格。如果当前拥有焦点的窗口不是窗格的子窗口，那么 GetActivePane() 返回 SPLIT_PANE_NONE。

```
bool ActivateNextPane(bool bNext = true)
```

如果分隔窗口是单窗格模式，焦点被设到可见的窗格上，否则的话，ActivateNextPane()函数将调用 GetActivePane()查看拥有 焦点的窗口。如果一个窗格(或窗格内的子窗口)拥有检点，分隔窗口就将焦点设给另一个窗格，否则 ActivateNextPane()将判断 bNext 的值，如果是 true 就激活 left/top 窗格，如果是 false 则激活 right/bottom 窗格。

```
bool SetDefaultActivePane(int nPane)
bool SetDefaultActivePane(HWND hWnd)
int GetDefaultActivePane()
```

调用 SetDefaultActivePane()函数可以设置默认的活动窗格，它的参数可以是 SPLIT_PANE_*类型的值，也可以是窗口的 句柄。如果分隔窗口自身得到的焦点，可以通过调用 SetFocus()将焦点转移给默认窗格。GetDefaultActivePane()函数返回 SPLIT_PANE_*类型的值表示哪个窗格是当前默认的活动窗格。

```
void GetSystemSettings(bool bUpdate)
```

GetSystemSettings()读取系统设置并相应的设置数据成员。分隔窗口在 OnCreate()函数中自动调用这个函数，你不需要自己调用这个函数。当然，你的主框架窗口应该响应 WM_SETTINGCHANGE 并将它传递给分隔窗口，CSplitterWindow 在 WM_SETTINGCHANGE 消息的处理函数中调用 GetSystemSettings()。传递 true 给 bUpdate 参数，分隔窗口会根据新的设置重画自己。

数据成员

其他的一些特性可以通过直接访问 CSplitterWindow 的公有成员来设定，只要 GetSystemSettings()被调用了，这些公有成员也会相应的被重置。

m_cxySplitBar：控制分隔条的宽度(垂直分隔条)和高度(水平分隔条)。默认值是通过调用 GetSystemMetrics (SM_CXSIZEFRAME)(垂直分隔条) 或 GetSystemMetrics(SM_CYSIZEFRAME)(水平分隔条)得到的。

m_cxyMin：控制每个窗格的最小宽度(垂直分隔)和最小高度(水平分隔)，分隔窗口不允许拖动比这更小的宽度或高度。如果分隔窗口有 WS_EX_CLIENTEDGE 扩展属性，则这个变量的默认值是 0，否则其默认值是 2*GetSystemMetrics(SM_CXEDGE)(垂直分隔) 或 2*GetSystemMetrics(SM_CYEDGE)(水平分隔)。

m_cxyBarEdge：控制画在分隔条两侧的 3D 边界的宽度(垂直分隔)或高度(水平分隔)，其默认值刚好和 m_cxyMin 相反。

m_bFullDrag：如果是 true，当分隔条被拖动时窗格大小跟着调整，如果是 false，拖动时只显示一个分隔条的影子，直到拖动停止才调 整窗格的大小。默认值是调用 SystemParametersInfo(SPI_GETDRAGFULLWINDOWS)函数的返回值。

开始一个例子工程

既然我们已经对分隔窗口有了基本的了解，我们就来看看如何创建一个包含分隔窗口的框架窗口。使用 WTL 向导开始一个新工程，在第一页选择 SDI Application 并单击 Next，在第二页，如下图所示取消工具条并选择不使用视图窗口：

我们不使用分隔窗口是因为分隔窗口和它的窗格将作为“视图窗口”，在 CMainFrame 类中添加一个 CSplitterWindow 类型的数据成员：

```
class CMainFrame : public ...
{
//...
protected:
   CSplitterWindow  m_wndVertSplit;
};
```

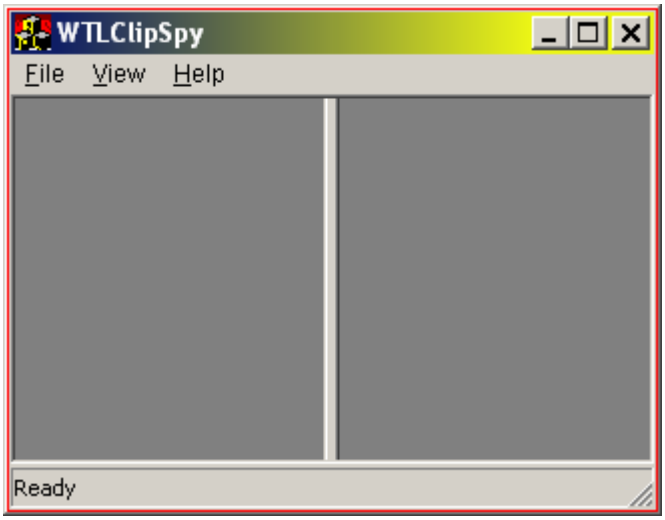
接着在 OnCreate()中创建分隔窗口并将其设为视图窗口：

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...
   // Create the splitter window
const DWORD dwSplitStyle = WS_CHILD | WS_VISIBLE |
                      WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
          dwSplitExStyle = WS_EX_CLIENTEDGE;

   m_wndVertSplit.Create ( *this, rcDefault, NULL,
                        dwSplitStyle, dwSplitExStyle );

   // Set the splitter as the client area window, and resize
   // the splitter to match the frame size.
   m_hWndClient = m_wndVertSplit;
   UpdateLayout();
```

```
    // Position the splitter bar.
    m_wndVertSplit.SetSplitterPos ( 200 );

    return 0;
}
```

需要注意的是在设置分隔窗口的位置之前要先设置 m_hWndClient 并调用 CFrameWindowImpl::UpdateLayout ()函数，UpdateLayout()将分隔窗口设置为初始时的大小。如果跳过这一步，分隔窗口的大小将不确定，可能小于 200 个象素点的宽度，最终导 致 SetSplitterPos()出现意想不到的结果。还有一种不调用 UpdateLayout()函数的方，就是先得到框架窗口的客户区坐标，然后使 用这个客户区坐标替换 rcDefault 坐标创建分隔窗口。使用这种方式创建的分隔窗口一开始就在正确的初始位置上，随后对位置调整的函数(例如 SetSplitterPos())都可以正常工作。

现在运行我们的程序就可以看到分隔条，即使没有创建任何窗格窗口它仍具有基本的行为。你可以拖动分隔条，用鼠标双击分隔条使其移到窗口的中间位置。



为了演示分隔窗口的不同使用方法，我将使用一个 CListViewCtrl 派生类和一个简单的 CRichEditCtrl，下面是从 CClipSpyListCtrl 类摘录的代码，我们在左边的窗格使用这个类：

```
typedef CWinTraitsOR<LVS_REPORT | LVS_SINGLESEL | LVS_NOSORTHEADER>
        CListTraits;

class CClipSpyListCtrl :
    public CWindowImpl<CClipSpyListCtrl, CListViewCtrl, CListTraits>,
    public CCustomDraw<CClipSpyListCtrl>
{
public:
    DECLARE_WND_SUPERCLASS(NULL, WC_LISTVIEW)

    BEGIN_MSG_MAP(CClipSpyListCtrl)
        MSG_WM_CHANGECBCHAIN(OnChangeCBChain)
        MSG_WM_DRAWCLIPBOARD(OnDrawClipboard)
        MSG_WM_DESTROY(OnDestroy)
```

```
        CHAIN_MSG_MAP_ALT(CCustomDraw<CClipSpyListCtrl>, 1)
        DEFAULT_REFLECTION_HANDLER()
    END_MSG_MAP()
//...
};
```

如果你看过前面的几篇文章就会很容易读懂这个类的代码。它响应 WM_CHANGECBCHAIN 消息，这样就可以知道是否启动和关闭了其它剪贴板查看程序，它还响应 WM_DRAWCLIPBOARD 消息，这样就可以知道剪贴板的内容是否改变。

由于分隔窗口窗格内的子窗口在程序运行其间一直存在，我们也可以将它们设为 CMainFrame 类的成员：

```
class CMainFrame : public ...
{
//...
protected:
    CSplitterWindow  m_wndVertSplit;
    CClipSpyListCtrl m_wndFormatList;
    CRichEditCtrl    m_wndDataViewer;
};
```

创建一个窗格内的窗口

既然已经有了分隔窗口和子窗口的成员变量，填充分隔窗口就是一件简单的事情了。先创建分隔窗口，然后创建两个子窗口，使用分隔窗口作为它们的父窗口：

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...
    // Create the splitter window
const DWORD dwSplitStyle = WS_CHILD | WS_VISIBLE |
                      WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
          dwSplitExStyle = WS_EX_CLIENTEDGE;

    m_wndVertSplit.Create ( *this, rcDefault, NULL,
                       dwSplitStyle, dwSplitExStyle );

    // Create the left pane (list of clip formats)
    m_wndFormatList.Create ( m_wndVertSplit, rcDefault );

    // Create the right pane (rich edit ctrl)
const DWORD dwRichEditStyle =
            WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
            ES_READONLY   |   ES_AUTOHSCROLL   |   ES_AUTOVSCROLL   |
ES_MULTILINE;

    m_wndDataViewer.Create ( m_wndVertSplit, rcDefault,
                         NULL, dwRichEditStyle );
    m_wndDataViewer.SetFont ( AtlGetStockFont(ANSI_FIXED_FONT) );
```

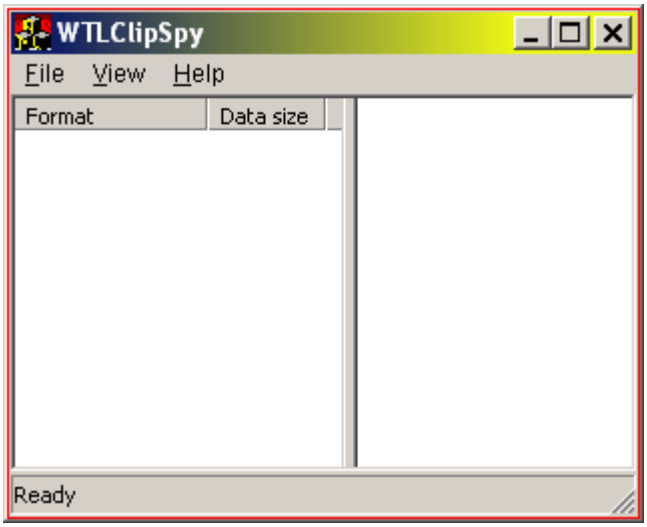
```
   // Set the splitter as the client area window, and resize
   // the splitter to match the frame size.
   m_hWndClient = m_wndVertSplit;
   UpdateLayout();

   m_wndVertSplit.SetSplitterPos ( 200 );

   return 0;
}
```

注意两个类的 Create()函数都用 m_wndVertSplit 作为父窗口，RECT 参数无关紧要，因为分隔窗口会重新调整它们的大小，所以可以使用 CWindow::rcDefault。

最后就是将窗口的句柄传递给分隔窗口的窗格，这一步也需要在 UpdateLayout()调用之前完成，这样最终所有的窗口都有正确的大小。

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...
   m_wndDataViewer.SetFont ( AtlGetStockFont(ANSI_FIXED_FONT) );

   // Set up the splitter panes
   m_wndVertSplit.SetSplitterPanes        (        m_wndFormatList,
m_wndDataViewer );

   // Set the splitter as the client area window, and resize
   // the splitter to match the frame size.
   m_hWndClient = m_wndVertSplit;
   UpdateLayout();

   m_wndVertSplit.SetSplitterPos ( 200 );

   return 0;
}
```

现在，list 控件上增加了几栏，结果看起来是这个样子：



需要注意的是分隔窗口对放进窗格的窗口类型没有限制，不像 MFC 那样必须是 CView 的派生类。窗格窗口只要有 WS_CHILD 样式就行了，没有任何其他限制。

消息处理

由于在主框架窗口和我们的窗格窗口之间加了一个分隔窗口，你可能想知道现在通知消息是如何工作的，比如，主框架窗口是如何收到 NM_CUSTOMDRAW 通知消息并将它反射给 list 控件的？答案就在 CSplitterWindowImpl 的消息链中：

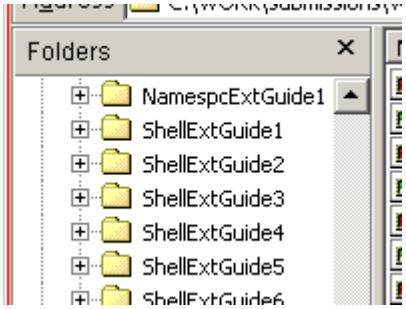
```
BEGIN_MSG_MAP(thisClass)
  MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBackground)
  MESSAGE_HANDLER(WM_SIZE, OnSize)
  CHAIN_MSG_MAP(baseClass)
  FORWARD_NOTIFICATIONS()
END_MSG_MAP()
```

最后的哪个 FORWARD_NOTIFICATIONS()宏最重要，回忆一下第四章，有一些通知消息总是被发送的子窗口的父窗口， FORWARD_NOTIFICATIONS()就是做了这些工作，它将这些消息转发给分隔窗口的父窗口。也就是说，当 list 窗口发送一个 WM_NOTIFY 消息给分隔窗口时(它是 list 的父窗口)，分隔窗口就将这个 WM_NOTIFY 消息转发给主框架窗口(它是分隔窗口的父窗口)。当主 框架窗口反射回消息时会将消息反射给 WM_NOTIFY 消息的最初发送者，也就是 list 窗口，所以分隔窗口并没有参与消息反射。

在 list 窗口和主框架窗口之间的这些消息传递并不影响分隔窗口的工作，这使得在程序中添加和移除分隔窗口非常容易，因为子窗口不需要做任何改变就可以继续工作。

窗格容器

WTL 还有一个被称为窗格容器的构件，它就像 Explorer 中左边的窗格那样，顶部有一个可以显示文字的区域，还有一个可选择是否显示的 Close 按钮：



就像分隔窗口管理两个窗格窗口一样，这个窗格容器也管理一个子窗口，当容器窗口的大小改变时，子窗口也相应的改变大小以便能够填充容器窗口的内部空间。

相关的类

这个窗格容器的实现需要两个类：CPaneContainerImpl 和 CPaneContainer，它们都在 atlctrlx.h 中声明。 CPaneContainerImpl 是一个 CWindowImpl 派生类，它含有窗格容器的完整实现， CPaneContainer 只是提供了一个类名，除非重载 CPaneContainerImpl 的方法或改变容器的外观，一般使用 CPaneContainer 就够了。

基本方法

```
HWND Create(
   HWND hWndParent, LPCTSTR lpstrTitle = NULL,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
WS_CLIPCHILDREN,
   DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL)
HWND Create(
   HWND hWndParent, UINT uTitleID,
   DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
WS_CLIPCHILDREN,
   DWORD dwExStyle = 0, UINT nID = 0, LPVOID lpCreateParam = NULL)
```

创建一个 CPaneContainer 窗口和创建其它子窗口一样。有两个 Create()函数，它们的区别仅仅是第二个参数不同。第一个函数需要传递一个字符串作为容器顶部区域显示的文字，第二个参数需要需要传一个字符串的资源 ID，其他参数只要使用默认值就行了。

```
DWORD SetPaneContainerExtendedStyle(DWORD dwExtendedStyle, DWORD
dwMask = 0)
DWORD GetPaneContainerExtendedStyle()
```

CPaneContainer 还有一些扩展样式用来控制容器窗口上 Close 按钮的布局方式：

PANECNT_NOCLOSEBUTTON：使用样式去掉顶部的 Close 按钮。

PANECNT_VERTICAL：设置这个样式后，顶部的文字区域将沿着容器窗口的左边界垂直放置。

扩展样式的默认值是 0，表示容器窗口是水平放置的，还有一个 Close 按钮。

```
HWND SetClient(HWND hWndClient)
HWND GetClient()
```

调用 SetClient()可以将一个子窗口指派给窗格容器，这和调用 CSplitterWindow 类的 SetSplitterPane() 方法作用类似。SetClient()同时返回原来的客户区窗口句柄而调用 GetClient()则可以得到当前的客户区窗口句柄。

```
BOOL SetTitle(LPCTSTR lpstrTitle)
BOOL GetTitle(LPTSTR lpstrTitle, int cchLength)
int GetTitleLength()
```

调用 SetTitle()可以改变容器窗口顶部显示的文字，调用 GetTitle()可以得到当前窗口顶部区域显示的文字,调用 GetTitleLength()可以得到当前显示的文字的字符个数(不包括结尾的空字符)。

```
BOOL EnableCloseButton(BOOL bEnable)
```

如果窗格容器使用的 Close 按钮，你可以调用 EnableCloseButton()来控制这个按钮的状态。

在分隔窗口中使用窗格容器

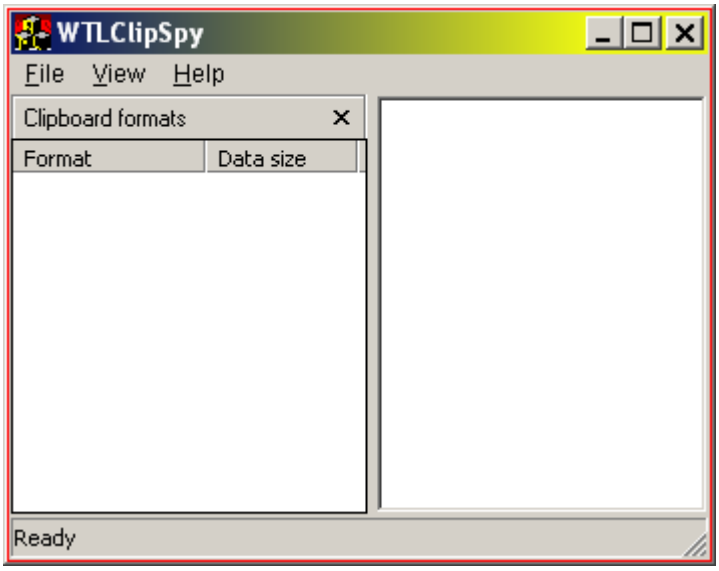
为了说明窗格容器的使用方法，我们将向 ClipSpy 的分隔窗口的左窗格添加一个窗格容器，

我们将一个窗格容器指派给左窗格取代原来使用的 list 控件，而将 list 控件指派给窗格容器。下面是在 CMainFrame::OnCreate()中为支持窗格容器而添加的代码。

```
LRESULT CMainFrame::OnCreate ( LPCREATESTRUCT lpcs )
{
//...
   m_wndVertSplit.Create ( *this, rcDefault, NULL, dwSplitStyle, dwSplitExStyle );

   // Create the pane container.
   m_wndPaneContainer.Create ( m_wndVertSplit, IDS_LIST_HEADER );

   // Create the left pane (list of clip formats)
   m_wndFormatList.Create ( m_wndPaneContainer, rcDefault );
//...
   // Set up the splitter panes
   m_wndPaneContainer.SetClient ( m_wndFormatList );
   m_wndVertSplit.SetSplitterPanes ( m_wndPaneContainer, m_wndDataViewer );
```

注意，现在 list 控件的父窗口是 m_wndPaneContainer，同时 m_wndPaneContainer 被设定成分隔窗口的左窗格。

下面是修改后的左窗格的外观，由于窗格容器在顶部的文本区域自己画了一个三维边框，所以我还要稍微修改一下边框的样式。这样看起来不是很好看，你可以自己调整样式知道你满意为止。(当然，你需要在 Windows XP 上测试一下哪个界面主题可以使得分隔窗口看起来“更有意思”。)



关闭按钮和消息处理

当用户用鼠标单击 Close 按钮时，窗格容器向父窗口发送一个 WM_COMMAND 消息，命令的 ID 是 ID_PANE_CLOSE。如果你在分隔窗 口中使用了窗格容器，你需要响应整个消息，调用 SetSinglePaneMode()隐藏这个窗格。(但是，不要忘了提供用户一个重新显示窗格的方法！)

CPaneContainer 的消息链也用到了 FORWARD_NOTIFICATIONS() 宏，和

CSplitterWindow 一样，窗格容器在客户窗口和它的父窗口之间传递通知消息。在 ClipSpy 这个例子中，在 list 控件和主框架窗口 之间隔了两个窗口(窗格容器和分隔窗口)，但是 FORWARD_NOTIFICATIONS()宏可以确保所有的通知消息被送到主框架窗口。

高级功能

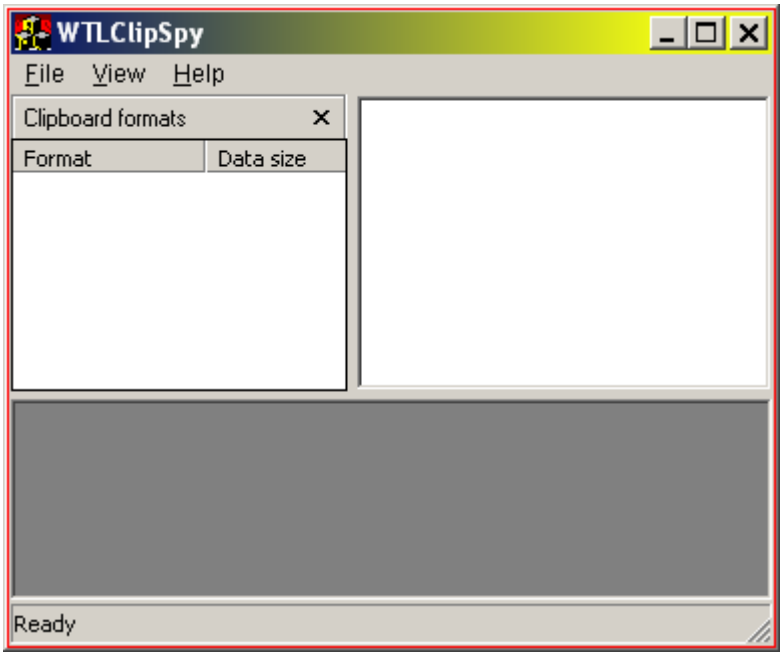
在这一节，我将介绍一些如何使用 WTL 的高级界面特性。

嵌套的分隔窗口

如果你要编写一个 email 的客户端程序，你可能需要使用嵌套的分隔条，一个水平的和一个垂直的分隔条。使用 WTL 很容易做到这一点：创建一个分隔窗口作为另一个分隔窗口的子窗口。为了演示这种效果，我将为 ClipSpy 添加一个水平分隔窗口。首先，添加一个名为 m_wndHorzSplitter 的 CHorSplitterWindow 类型的成员，像创建垂直分隔窗口 m_wndVertSplitter 那样创建这个水平分隔窗口，使水平分隔窗口 m_wndHorzSplitter 成为顶层窗口，将 m_wndVertSplitter 创建成 m_wndHorzSplitter 的子窗口。最后将 m_hWndClient 设置为 m_wndHorzSplitter，因为现在水平分隔窗口占据整个主框架窗口的客户区。

```
LRESULT CMainFrame::OnCreate()
{
//...
   // Create the splitter windows.
   m_wndHorzSplit.Create ( *this, rcDefault, NULL,
                     dwSplitStyle, dwSplitExStyle );

   m_wndVertSplit.Create ( m_wndHorzSplit, rcDefault, NULL,
                     dwSplitStyle, dwSplitExStyle );
//...
   // Set the horizontal splitter as the client area window.
   m_hWndClient = m_wndHorzSplit;

   // Set up the splitter panes
   m_wndPaneContainer.SetClient ( m_wndFormatList );
   m_wndHorzSplit.SetSplitterPane ( SPLIT_PANE_TOP, m_wndVertSplit );
   m_wndVertSplit.SetSplitterPanes      (      m_wndPaneContainer,
m_wndDataViewer );
//...
}
```

最终的结果是这个样子的：



在窗格中使用 ActiveX 控件

在分隔窗口的窗格中使用 ActiveX 控件与在对话框中使用 ActiveX 控件类似，使用 CAxWindow 类的方法在运行是创建控件，然后将这个 CAxWindow 指定给分隔窗口的窗格。下面演示了如何在水平分隔窗口下面的窗格中使用浏览器控件：

```
   // Create the bottom pane (browser)
CAxWindow wndIE;
const DWORD dwIEStyle = WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN |
                    WS_HSCROLL | WS_VSCROLL;

   wndIE.Create ( m_wndHorzSplit, rcDefault,
              _T("http://www.codeproject.com"), dwIEStyle );

   // Set the horizontal splitter as the client area window.
   m_hWndClient = m_wndHorzSplit;

   // Set up the splitter panes
   m_wndPaneContainer.SetClient ( m_wndFormatList );
   m_wndHorzSplit.SetSplitterPanes ( m_wndVertSplit, wndIE );
   m_wndVertSplit.SetSplitterPanes       (       m_wndPaneContainer,
m_wndDataViewer );
```

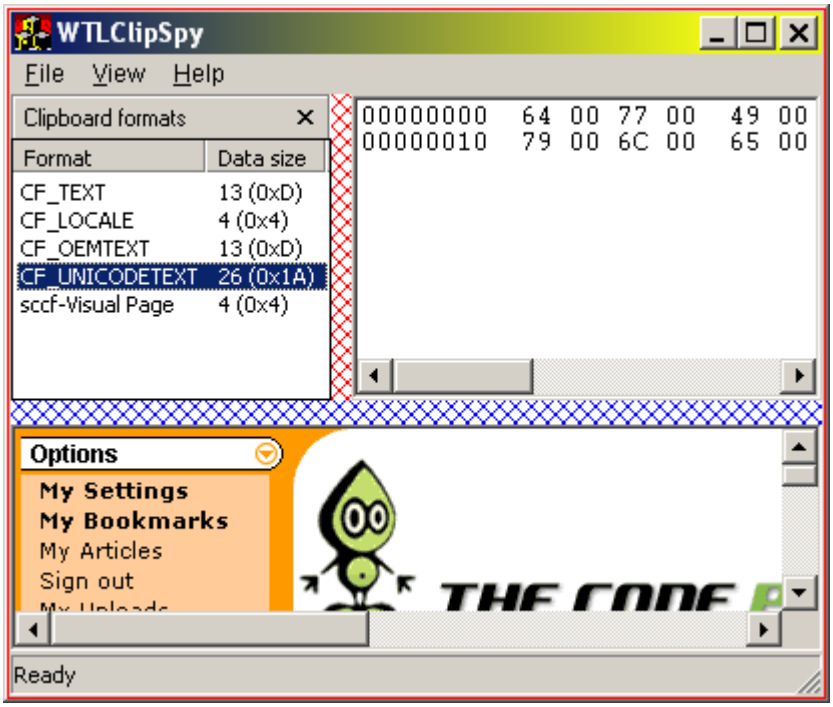
特殊绘制

如果你想改变分隔条的外观，例如在上面使用一些材质，你可以从 CSplitterWindowImpl 派生新类并重载 DrawSplitterBar()函数。如果你只是想调整一下分隔条的外观，可以复制 CSplitterWindowImpl 类的函数，然后稍做修改。 下面的例子就在分隔条中使用了斜交叉线图案。

```
template <bool t_bVertical = true>
class CMySplitterWindowT :
   public       CSplitterWindowImpl<CMySplitterWindowT<t_bVertical>,
```

```
t_bVertical>
{
public:
   DECLARE_WND_CLASS_EX(_T("My_SplitterWindow"),
                     CS_DBLCLKS, COLOR_WINDOW)

   // Overrideables
   void DrawSplitterBar(CDCHandle dc)
   {
   RECT rect;

      if ( m_br.IsNull() )
         m_br.CreateHatchBrush ( HS_DIAGCROSS,
                              t_bVertical ? RGB(255,0,0)
                                          : RGB(0,0,255) );

      if ( GetSplitterBarRect ( &rect ) )
      {
         dc.FillRect ( &rect, m_br );

         // draw 3D edge if needed
         if ( (GetExStyle() & WS_EX_CLIENTEDGE) != 0)
            dc.DrawEdge(&rect, EDGE_RAISED,
                        t_bVertical ? (BF_LEFT | BF_RIGHT)
                                    : (BF_TOP | BF_BOTTOM));
      }
   }

protected:
   CBrush m_br;
};

typedef CMySplitterWindowT<true>    CMySplitterWindow;
typedef CMySplitterWindowT<false>   CMyHorSplitterWindow;
```

这就是结果(将分隔条变宽是为了更容易看到效果)：



窗格容器内的特殊绘制

CPaneContainer 也有几个函数可以重载，用来改变窗格容器的外观。你可以从 CPaneContainerImpl 派生新类并重载你需要的方法，例如：

```
class CMyPaneContainer :
    public CPaneContainerImpl<CMyPaneContainer>
{
public:
    DECLARE_WND_CLASS_EX(_T("My_PaneContainer"), 0, -1)
//... overrides here ...
};
```

一些更有意思的方法是：

```
void CalcSize()
```

调用 CalcSize()函数只是为了设置 m_cxyHeader，这个变量控制着窗格容器的顶部区域的宽度和高度。不过 SetPaneContainerExtendedStyle()函数中有一个 BUG，导致窗格从水平切换到垂直时没有调用派生类的 CalcSize() 方法，你可以将 CalcSize()调用改为 pT->CalcSize()来修补这个 BUG。

```
HFONT GetTitleFont()
```

这个方法返回一个 HFONT，它被用来画顶部区域的文字，默认的值是调用 GetStockObject(DEFAULT_GUI_FONT)得到的字体，也就是 MS Sans Serif。如果你想改称更现代的 Tahoma 字体，你可以重载 GetTitleFont()方法，返回你创建的 Tahoma 字体。

```
BOOL GetToolTipText(LPNMHDR lpnmh)
```

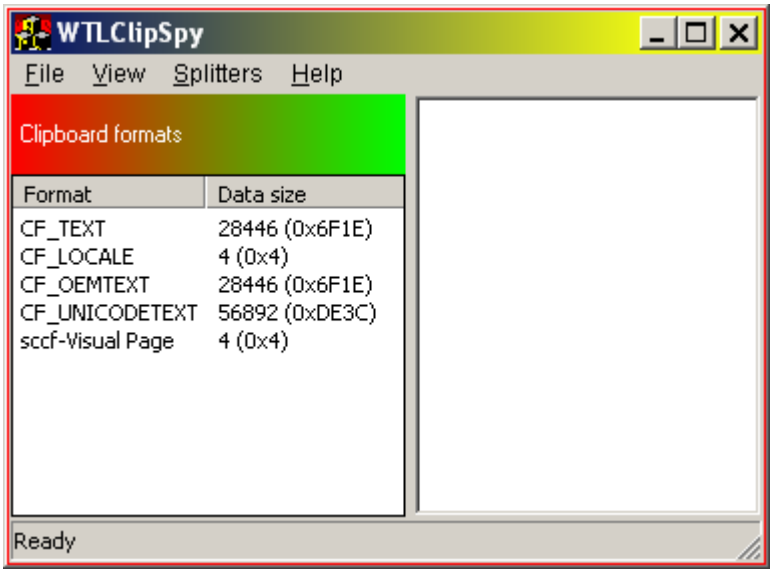
重载这个方法提供鼠标移到 Close 按钮时弹出的提示信息，这个函数实际上是 TTN_GETDISPINFO 的相应函数，你可以将 lpnmh 转换成 NMTTDISPINFO*,并设置这个数据结构内相应的成员变量。记住一点，你必须检查通知代码，它可能是 TTN_GETDISPINFO 或 TTN_GETDISPINFOW，你需要有区别的访问这两个数据结构。

```
void DrawPaneTitle(CDCHandle dc)
```

你可以重载这个方法自己画顶部区域，你可以用 GetClientRect()和 m_cxyHeader 来计算顶部区域的范围。下面的例子演示了在水平容器的顶部区域画一个渐变填充的背景：

```
void CMyPaneContainer::DrawPaneTitle ( CDCHandle dc )
{
RECT rect;

   GetClientRect(&rect);

TRIVERTEX tv[] = {
   { rect.left, rect.top, 0xff00 },
   { rect.right, rect.top + m_cxyHeader, 0, 0xff00 }
};
GRADIENT_RECT gr = { 0, 1 };

   dc.GradientFill ( tv, 2, &gr, 1, GRADIENT_FILL_RECT_H );
}
```

例子工程代码中演示了对这几个方法的重载，使得结果看起来是这个样子的：



从上面的图中可以看到，这个演示程序有一个 Splitters 菜单，通过它可以在各种风格的分隔条(包括自画风格)和窗格容器之间切换，比较它们之间的异同。你还可以锁定分隔条的位置，这是通过设置和取消 SPLIT_NONINTERACTIVE 扩展风格来实现的。

在状态栏显示进度条

正如我在前几篇文章中做得保证那样，新的 ClipSpy 也演示了如何在状态条上创建进展条，它和 MFC 版本得功能一样，几个相关得步骤是：

得到状态条第一个窗格得坐标范围 RECT

创建一个进展条作为状态条得子窗口，窗口大小就是哪个状态条窗格得大小

随着 edit 控件被填充的同时更新进展条的位置

这些代码在 CMainFrame::CreateProgressCtrlInStatusBar()函数中。

继续

在第八章我将介绍属性页和向导对话框的用法

参考

WTL Splitters and Pane Containers by Ed Gadziemski

第八部分　使用属性表

甚至在成为 Windows 95 的通用控件之前，使用属性表来表示一些选项就已经成为一种很流行的方式。向导模式的属性表通常用来引导用户安装软件或完成其他复杂的工作。WTL 对这 两种方式的属性表都提供了很好的支持，可以使用前面介绍的与对话框相关的特性，如 DDX 和 DDV。在本章我将演示如何创建一个基本的属性表和向导，如何处 理属性页发送的通知消息和事件。

WTL 的属性表类

实现一个属性表需要 CPropertySheetWindow 和 CPropertySheetImpl 两个类联合使用，它们都定义在 atldlgs.h 头文件中。CPropertySheetWindow 类是一个窗口接口类(也就是说是一个 CWindow 派生类)， CPropertySheetImpl 有消息映射链和窗口的完整实现，这和 ATL 的基本窗口类相似，它需要 CWindow 和 CWindowImpl 两个类 联合使用。

CPropertySheetWindow 类封装了对各种 PSM_* 消息的处理，例如，SetActivePageByID()封装了 PSM_SETCURSELID 消息。CPropertySheetImpl 类管理一个 PROPSHEETHEADER 结构和一个 HPROPSHEETPAGE 类型的数组，CPropertySheetImpl 类还提供了

一些方法用来填充 PROPSHEETHEADER 结构，添加或删除属性页，你也可以使用 m_psh 成员变量直接操作 PROPSHEETHEADER 结构。

最后，CPropertySheet 类是 CPropertySheetImpl 类的一个特例，你可以直接使用它而不需要定制整个属性表。

CPropertySheetImpl 的方法

下面是 CPropertySheetImpl 类的一些重要方法。由于许多方法仅仅是对窗口消息的封装，所以就不在这里列出，你可以查看 atldlgs.h 中完整的函数清单。

```
CPropertySheetImpl(_U_STRINGorID title = (LPCTSTR) NULL,
                   UINT uStartPage = 0, HWND hWndParent = NULL)
```

CPropertySheetImpl 类的构造函数允许你使用一些常用的属性(默认值)，所以就不需要在调用其他的方法设置它们。title 指定显 示在属性表的标题栏的文字， _U_STRINGorID 是一个 WTL 的工具类，它可以自动转换 LPCTSTR 和资源 ID，例如，下面的两行代码都是正确的：

```
  CPropertySheetImpl mySheet ( IDS_SHEET_TITLE );
  CPropertySheetImpl mySheet ( _T("My prop sheet") );
```

IDS_SHEET_TITLE 是字符串的 ID。 uStartPage 是属性表启动时激活的属性页，是一个从 0 开始的索引。hWndParent 是属性表的父窗口的句柄。

```
BOOL AddPage(HPROPSHEETPAGE hPage)
BOOL AddPage(LPCPROPSHEETPAGE pPage)
```

添加一个属性页。如果这个属性页已经创建了，你可以使用第一个重载函数，使用属性页的句柄(HPROPSHEETPAGE)作为参数。通常是使用第 二个重载函数，使用这个重载函数只需设置一个 PROPSHEETPAGE 数据结构(后面会讲到，它和 CPropertyPageImpl 一起协同工作)，CPropertySheetImpl 会为你创建并管理这个属性页。

```
BOOL RemovePage(HPROPSHEETPAGE hPage)
BOOL RemovePage(int nPageIndex)
```

移除一个属性页，可以使用属性页的句柄或索引。

BOOL SetActivePage(HPROPSHEETPAGE hPage)
BOOL SetActivePage(int nPageIndex)
设置属性表的活动页面。可以使用属性页的句柄或索引。你可以在属性表创建(显示)之前使用这个方法动态的设置处于激活的属性页。
void SetTitle(LPCTSTR lpszText, UINT nStyle = 0)
使之属性表窗口的标题文字。nStyle可以是0或PSH_PROPTITLE,如果是PSH_PROPTITLE,则属性表就具有 PSH_PROPTITLE 样式，这样系统会在你通过 lpszText 参数指定的窗口标题前添加字符串“Properties for”。
void SetWizardMode()
设置 PSH_WIZARD 样式，将属性表改称向导模式，这个函数必须在属性表显示之前调用。
void EnableHelp()
设置 PSH_HASHELP 样式，将在属性表中添加帮助按钮。需要注意的是你还要在每个属性页中使帮助按钮可用并提供帮助才能使之生效。
INT_PTR DoModal(HWND hWndParent = ::GetActiveWindow())
创建并显示一个模式的属性表，返回正值表示操作成功，有关 PropertySheet() API 的帮助文档有有关返回值的详细解释，如果发生错误，属性表无法创建，DoModal()返回－1。
HWND Create(HWND hWndParent = NULL)
创建并显示一个无模式的属性表，返回值是窗口的句柄，如果发生错误，属性表无法创建，Create()返回 NULL。
WTL 的属性页类
WTL 对属性页的封装类与属性表的封装类相似，有一个窗口接口类 CPropertyPageWindow 和一个实现类 CPropertyPageImpl 。CPropertyPageWindow 很小，包含最常用的需要在作为父窗口的属性表中调用的方法。
CPropertyPageImpl 是从 CDialogImplBaseT 派生，由于属性页是从对话框资源中创建的,这就意味着所有可以在对话框中使用的WTL的特性都可以在属性页中使用,如 DDX 和DDV。CPropertyPageImpl 有两个主要作用：管理一个 PROPSHEETPAGE 数据结构(保存在成员变量 m_psp 中)，处理所有 PSN_开头的通知消息。对于很简单的属性页可以 直接使用 CPropertyPage 类，这个类只适合与用户没有任何交互的属性页，例如“关于”页面或者向导中的介绍页面
也可以创建含有 ActiveX 控件的属性页。首先，这需要在 stdafx.h 文件中添加对 atlhost.h 的包含，还要使用 CAxPropertyPageImpl 代替 CPropertyPageImpl。对于简单的页面可以使用 CAxPropertyPage 代替 CPropertyPage。
CPropertyPageImpl 的方法
CPropertyPageImpl 管理着一个 PROPSHEETPAGE 结构，也就是公有成员 m_psp。CPropertyPageImpl 还重载了 PROPSHEETPAGE* 操作符，所以你可以将 CPropertyPageImpl 传递给需要 LPPROPSHEETPAGE 或 LPCPROPSHEETPAGE 类型的参数的方法，例如 CPropertySheetImpl::AddPage()。
CPropertyPageImpl 的构造函数允许你设置页面的标题，标题通常显示在页面的 Tab 标签上：
CPropertyPageImpl(_U_STRINGorID title = (LPCTSTR) NULL)
如果你不想让属性表创建属性页面而是想手工创建页面，你可以调用 Create():
HPROPSHEETPAGE Create()
Create() 只是调用用 m_psp 做参数调用了 CreatePropertySheetPage() 。如果你向一个已经创建的属性表添加属性页或者向另一个不在控制的属性表添加属性页(例如，处理系

统 Shell 扩展的属性表)，那就只需要调用 Create()函数。
下面的三个方法用于设置属性页的各种标题文本：
void SetTitle(_U_STRINGorID title)
void SetHeaderTitle(LPCTSTR lpstrHeaderTitle)
void SetHeaderSubTitle(LPCTSTR lpstrHeaderSubTitle)
第一个方法改变页面标签的文字，另外几个用来设置 Wizard97 样式的向导中属性页顶部的文字。
void EnableHelp()
设置 m_psp 中的 PSP_HASHELP 标志，当本页面激活时使属性表的帮助按钮可用。
处理通知消息
CPropertyPageImpl 有一个消息映射处理 WM_NOTIFY。如果通知代码是 PSN_*的值，OnNotify()就会调用相应的通知处理函数。这使用了编译阶段虚函数机制，从而使得派生类可以很容易的重载这些处理函数。
由于 WTL 3 和 WTL 7 设计的改变，从而存在两套不同的通知处理机制。在 WTL 3 中通知处理函数返回的值与 PSN_*消息的返回值不同，例如，WTL 3 是这样处理 PSN_WIZFINISH 的：

```
case PSN_WIZFINISH:
lResult = !pT->OnWizardFinish();
break;
```

OnWizardFinish()期望返回 TRUE 结束向导，FALSE 阻止关闭向导。这个方法很简陋，但是 IE5 的通用控件对 PSN_WIZFINISH 处理的返回值添加了新解释，他返回需要获得焦点的窗口的句柄。WTL 3 的程序将不能使用这个特性，因为它对所有非 0 的返回值都做相同的处理。在 WTL 7 中，OnNotify() 没有改变 PSN_* 消息的返回值，处理函数返回任何文档中规定的合法数值和正确的行为。当然，为了向前兼容，WTL 3 仍然使用当前默认的工作方式，要使用 WTL 7 的消息处理方式，你必须在中 including atldlgs.h 一行之前添加一行定义：
#define _WTL_NEW_PAGE_NOTIFY_HANDLERS
编写新的代码没有理由不使用 WTL 7 的消息处理函数，所以这里就不介绍 WTL 3 的消息处理方式。
CPropertyPageImpl 为所有消息提供了默认的通知消息处理函数，你可以重载与你的程序有关的消息处理函数完成特殊的操作。默认的消息处理函数和相应的行为如下：
int OnSetActive() - 允许页面成为激活状态
BOOL OnKillActive() - 允许页面成为非激活状态
int OnApply() - 返回 PSNRET_NOERROR 表示应用操作成功完成
void OnReset() - 无相应的动作
BOOL OnQueryCancel() - 允许取消操作
int OnWizardBack() - 返回到前一个页面
int OnWizardNext() - 进行到下一个页面
INT_PTR OnWizardFinish() - 允许向导结束
void OnHelp() - 无相应的动作
BOOL OnGetObject(LPNMOBJECTNOTIFY lpObjectNotify) - 无相应的动作
int OnTranslateAccelerator(LPMSG lpMsg) - 返回 PSNRET_NOERROR 表示消息没有被处理
HWND OnQueryInitialFocus(HWND hWndFocus) - 返回 NULL 表示将按 Tab Order 顺序的第一个控件设为焦点状态

创建一个属性表

关于这些类的解释就全部讲完了，现在需要一个例子程序演示如何使用它们。本章的例子工程是一个简单的 SDI 程序，它在客户区显示一幅图片并使用一总颜色填充背景，使用的图片和颜色可以通过一个选项对话框(一个属性表)来设置，还有一个向导(稍后会介绍)。

最简单的属性表

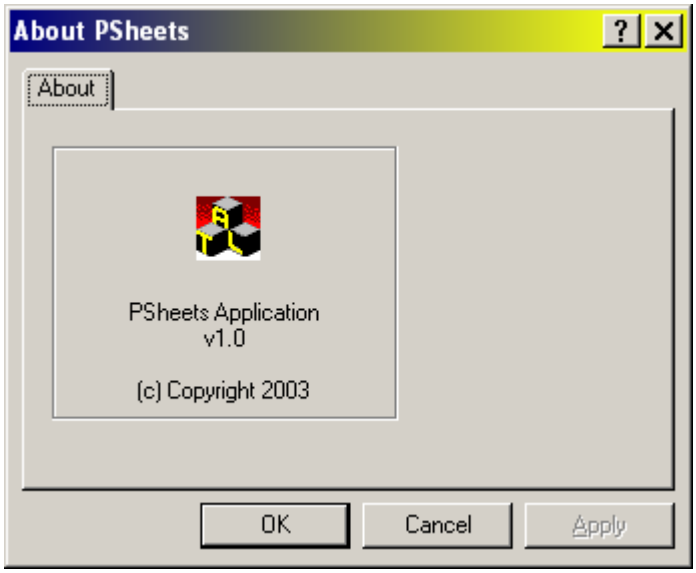
首先用 WTL 的向导创建一个 SDI 工程，然后为关于对话框添加一个属性表。首先改变向导创建的关于对话框样式，使它用起来像个属性页。

第一步就是去除 OK 按钮，因为属性表不希望属性页自己关闭。在 Style Tab 中，将对话框样式改为 Child，Thin Border，选择 Title Bar，在 More Styles tab，选择 Disabled。

第二步(也是最后一步)是在 OnAppAbout()的处理函数中创建一个属性表，我们使用非定制的 CPropertySheet 和 CPropertyPage 类：

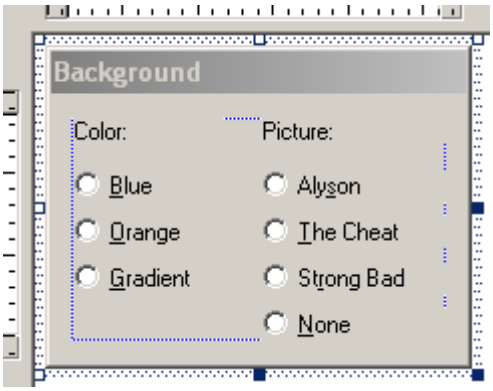
```
LRESULT CMainFrame::OnAppAbout(...)
{
CPropertySheet sheet ( _T("About PSheets") );
CPropertyPage<IDD_ABOUTBOX> pgAbout;

   sheet.AddPage ( pgAbout );
   sheet.DoModal();
   return 0;
}
```

结果看起来向下面这样：



创建一个有用的属性页

并不是每一个属性表中的每一个属性页都像关于对话框这么简单，大多数属性页需要使用 CPropertyPageImpl 的派生类，所以我们现在就看一个这样的类。我们创建了一个新的属性页用来设置客户区背景显示的图片，它是这个样子的：



这个对话框的样式和关于页面相同，我们需要一个新类来和这个属性页协同工作，我们将其命名为 CBackgroundOptsPage。这个类是从 CPropertyPageImpl 类派生的，它有一个 CWinDataExchange 来支持 DDX。

```
class CBackgroundOptsPage :
    public CPropertyPageImpl<CBackgroundOptsPage>,
    public CWinDataExchange<CBackgroundOptsPage>
{
public:
    enum { IDD = IDD_BACKGROUND_OPTS };

    // Construction
    CBackgroundOptsPage();
    ~CBackgroundOptsPage();

    // Maps
    BEGIN_MSG_MAP(CBackgroundOptsPage)
        MSG_WM_INITDIALOG(OnInitDialog)
        CHAIN_MSG_MAP(CPropertyPageImpl<CBackgroundOptsPage>)
    END_MSG_MAP()

    BEGIN_DDX_MAP(CBackgroundOptsPage)
        DDX_RADIO(IDC_BLUE, m_nColor)
        DDX_RADIO(IDC_ALYSON, m_nPicture)
    END_DDX_MAP()

    // Message handlers
    BOOL OnInitDialog ( HWND hwndFocus, LPARAM lParam );

    // Property page notification handlers
    int OnApply();

    // DDX variables
    int m_nColor, m_nPicture;
};
```

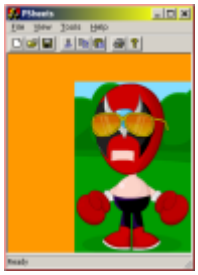
关于这个类需要注意几点：
有一个名为 IDD 的公有成员将对话框于资源联系起来。
消息映射链和 CDialogImpl 相似。
消息映射链将消息链入 CPropertyPageImpl，从而使我们能够处理与属性表相关的通知消息。
有一个 OnApply()处理函数在单击属性表中的 OK 按钮时保存用户的选择。
OnApply() 非常简单，它调用 DoDataExchange() 更新 DDX 变量，然后返回一个代码标识是否可以关闭这个属性表：

```
int CBackgroundOptsPage::OnApply()
{
   return DoDataExchange(true) ? PSNRET_NOERROR : PSNRET_INVALID;
}
```

我们还要在主窗口添加一个 Tools|Options 菜单来打开属性表，这个菜单的处理函数创建一个属性表，但是添加了一个新属性页 CBackgroundOptsPage。

```
void CMainFrame::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )
{
CPropertySheet sheet ( _T("PSheets Options"), 0 );
CBackgroundOptsPage pgBackground;
CPropertyPage<IDD_ABOUTBOX> pgAbout;

   pgBackground.m_nColor = m_view.m_nColor;
   pgBackground.m_nPicture = m_view.m_nPicture;

   sheet.m_psh.dwFlags |= PSH_NOAPPLYNOW;

   sheet.AddPage ( pgBackground );
   sheet.AddPage ( pgAbout );

   if ( IDOK == sheet.DoModal() )
      m_view.SetBackgroundOptions ( pgBackground.m_nColor,
                                    pgBackground.m_nPicture );
}
```

属性表的构造函数的第二个参数是 0，表示将索引是 0 的页面初始是可见的，你可以将其设为 1，使得属性表第一次显示时显示关于页面。既然是演示代码，我就偷个懒，使用一个公有变量与 CBackgroundOptsPage 属性页的 radio button 建立关联，在主窗口中直接为其赋初始值，当用户单击属性表的 OK 按钮时在将其读出来。
如果用户点击 OK 按钮，DoModal()发挥 IDOK，我们通知视图窗口使用新的图片和背景颜色。
下面是几个屏幕截图显示几个不同的样式的视图：

创建一个更好的属性表类

在 OnOptions()中创建属性表是个好主意，但是在这里使用很多初始化代码却非常糟糕，这不是 CMainFrame 应该做得事情。更好的方法是从 CPropertySheetImpl 派生一个新类，在这个类中完成这些任务。

```
#include "BackgroundOptsPage.h"

class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>
{
public:
   // Construction
   CAppPropertySheet ( _U_STRINGorID title = (LPCTSTR) NULL,
                       UINT uStartPage = 0, HWND hWndParent = NULL );

   // Maps
   BEGIN_MSG_MAP(CAppPropertySheet)
      CHAIN_MSG_MAP(CPropertySheetImpl<CAppPropertySheet>)
   END_MSG_MAP()

   // Property pages
   CBackgroundOptsPage        m_pgBackground;
   CPropertyPage<IDD_ABOUTBOX> m_pgAbout;
};
```

我们使用这个类封装属性表中各个属性页的细节，将初始化代码移到属性表内部完成，构造函数完成添加页面，并设置其他必需的标志：

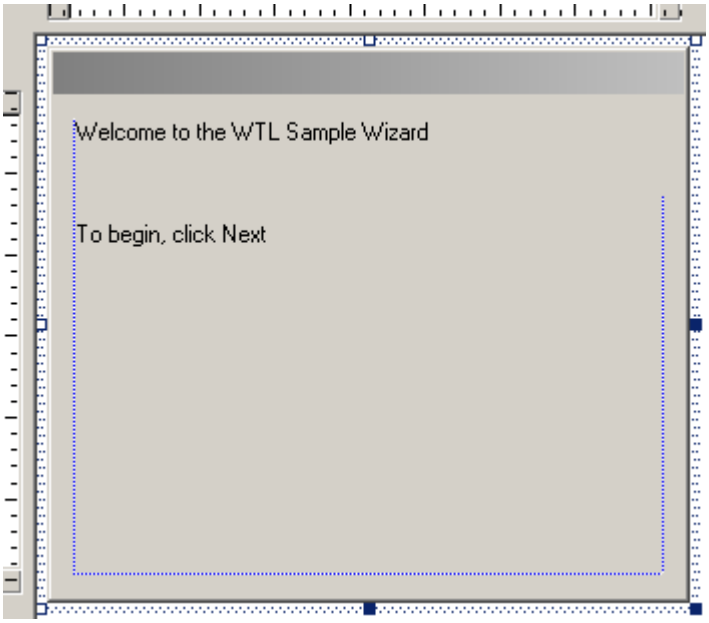
```
CAppPropertySheet::CAppPropertySheet  (  _U_STRINGorID  title,  UINT
uStartPage,
                                       HWND hWndParent ) :
   CPropertySheetImpl<CAppPropertySheet>   (   title,   uStartPage,
hWndParent )
{
   m_psh.dwFlags |= PSH_NOAPPLYNOW;

   AddPage ( m_pgBackground );
   AddPage ( m_pgAbout );
}
```

这样一来，OnOptions()处理函数就变得简单了一些：

```
void CMainFrame::OnOptions ( UINT uCode, int nID, HWND hwndCtrl )
{
CAppPropertySheet sheet ( _T("PSheets Options"), 0 );

    sheet.m_pgBackground.m_nColor = m_view.m_nColor;
    sheet.m_pgBackground.m_nPicture = m_view.m_nPicture;

    if ( IDOK == sheet.DoModal() )
        m_view.SetBackgroundOptions ( sheet.m_pgBackground.m_nColor,
                                      sheet.m_pgBackground.m_nPicture );
}
```

创建一个向导样式的属性表

创建一个向导和创建一个属性表很相似，这并不奇怪，只需稍做修改添加“上一步”和“下一步”按钮就行了。和 MFC 一样，你需要重载 OnSetActive()函数并调用 SetWizardButtons()使相应的按钮可用。我们先从一个简单的介绍页面开始，它的 ID 是 IDD_WIZARD_INTRO：



注意这个页面没有标题栏文字，因为向导中的所有的页面通常都有相同的标题，我更愿意在 CPropertySheetImpl 的构造函数中设置这些文字，然后每个页面使用这个字符串资源。这就是为什么我只需要改变一个字符串就能改变所有页面标题文字的原因。

关于这个页面的实现代码在 CWizIntroPage 类中：

```
class CWizIntroPage : public CPropertyPageImpl<CWizIntroPage>
{
public:
    enum { IDD = IDD_WIZARD_INTRO };

    // Construction
    CWizIntroPage();

    // Maps
    BEGIN_MSG_MAP(COptionsWizard)
```

```
        CHAIN_MSG_MAP(CPropertyPageImpl<CWizIntroPage>)
    END_MSG_MAP()

    // Notification handlers
    int OnSetActive();
};
```

构造函数使用(引用)一个字符串资源 ID 来设置页面的文字：

```
CWizIntroPage::CWizIntroPage() :
    CPropertyPageImpl<CWizIntroPage>( IDS_WIZARD_TITLE )
{
}
```

当这个页面激活时，字符串 IDS_WIZARD_TITLE ("PSheets Options Wizard")将出现在向导的标题栏。OnSetActive()仅仅使“下一步”按钮可用：

```
int CWizIntroPage::OnSetActive()
{
    SetWizardButtons ( PSWIZB_NEXT );
    return 0;
}
```

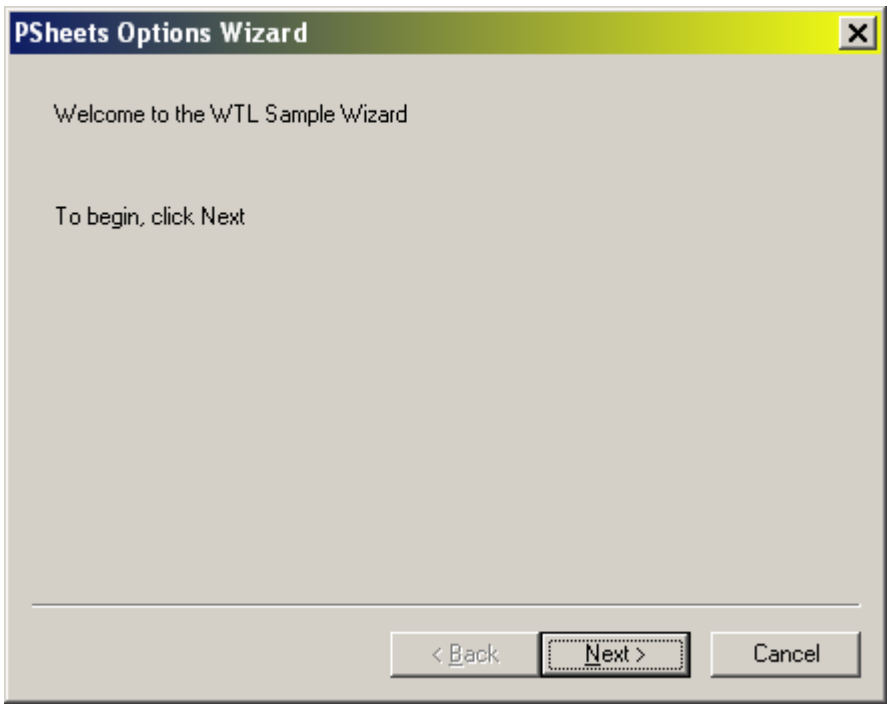
为了实现一个向导，我们需要创建一个类 COptionsWizard，还要在主窗口添加菜单 Tools|Wizard。 COptionsWizard 类的构造函数和 CAppPropertySheet 类的构造函数一样，只是设置必要的样式标志和添加页面。

```
class COptionsWizard : public CPropertySheetImpl<COptionsWizard>
{
public:
    // Construction
    COptionsWizard ( HWND hWndParent = NULL );

    // Maps
    BEGIN_MSG_MAP(COptionsWizard)
        CHAIN_MSG_MAP(CPropertySheetImpl<COptionsWizard>)
    END_MSG_MAP()

    // Property pages
    CWizIntroPage m_pgIntro;
};

COptionsWizard::COptionsWizard ( HWND hWndParent ) :
    CPropertySheetImpl<COptionsWizard> ( 0U, 0, hWndParent )
{
    SetWizardMode();

    AddPage ( m_pgIntro );
}
```

CMainFrame 类的 Tools|Wizard 菜单处理函数是这个样子：

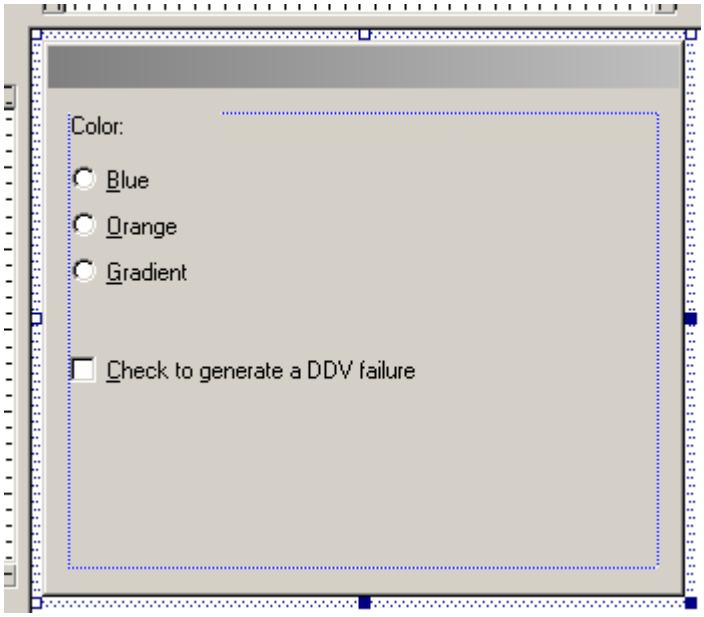
```
void CMainFrame::OnOptionsWizard ( UINT uCode, int nID, HWND hwndCtrl )
{
COptionsWizard wizard;

    wizard.DoModal();
}
```

这就是向导的效果：



添加更多的属性页，使用 DDV

为了使这个向导能够有点用处，我们要为其添加一个设置视图背景颜色的页面。这个页面还将有一个 checkbox 演示如何处理 DDV 验证失败并阻止向导进行到下一页。下面就是新的页面，ID 是 IDD_WIZARD_BKCOLOR：



这个类的实现代码在 CWizBkColorPage 类中，下面是相关的部分代码

```
class CWizBkColorPage :
    public CPropertyPageImpl<CWizBkColorPage>,
    public CWinDataExchange<CWizBkColorPage>
{
public:
    // some stuff removed for brevity...

    BEGIN_DDX_MAP(CWizBkColorPage)
        DDX_RADIO(IDC_BLUE, m_nColor)
        DDX_CHECK(IDC_FAIL_DDV, m_bFailDDV)
    END_DDX_MAP()

    // Notification handlers
    int OnSetActive();
    BOOL OnKillActive();

    // DDX vars
    int m_nColor;

protected:
    int m_bFailDDV;
};
```

OnSetActive()的工作和前面的介绍页面相同，它使“上一步”和“下一步”按钮可用。OnKillActive()是个新的处理函数，它 触发 DDV，然后检查 m_bFailDDV 的值，如果是 TRUE 就表示 checkbox 处于选中状态，OnKillActive()将阻止向导进行到下一 页。

```
int CWizBkColorPage::OnSetActive()
{
    SetWizardButtons ( PSWIZB_BACK | PSWIZB_NEXT );
    return 0;
}

int CWizBkColorPage::OnKillActive()
{
    if ( !DoDataExchange(true) )
        return TRUE;    // prevent deactivation

    if ( m_bFailDDV )
        {
        MessageBox (
          _T("Error box checked, wizard will stay on this page."),
          _T("PSheets"), MB_ICONERROR );

        return TRUE;    // prevent deactivation
        }
```

```
    return FALSE;      // allow deactivation
}
```

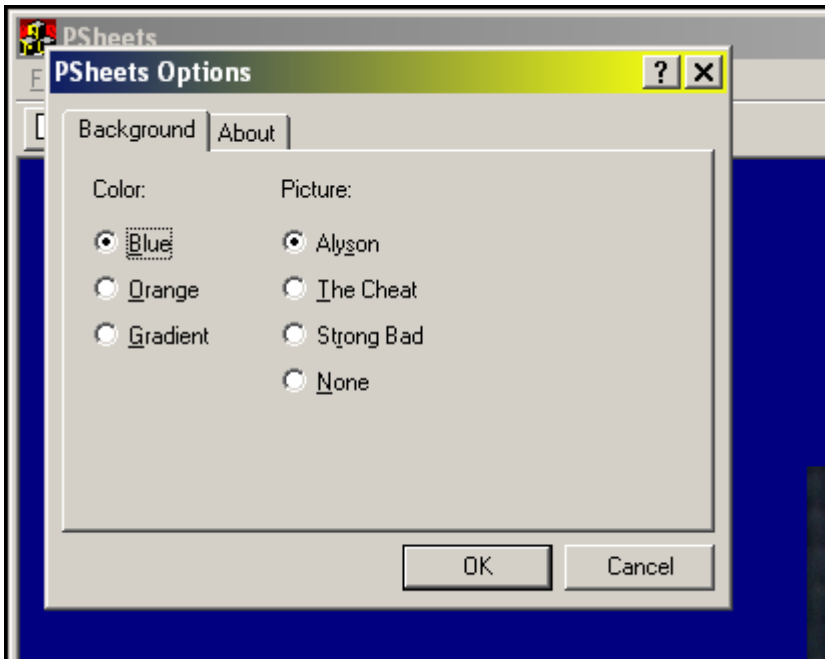
需要注意的是 OnKillActive()中做的事情也可以在 OnWizardNext()中完成，因为这两个处理函数都可以使向导维持在当前页 面。它们的不同之处在于 OnKillActive()在用户单击“上一步”和“下一步”按钮时被调用，而 OnWizardNext()只是在用户单击 “下一步”按钮时被调用。OnWizardNext()还被用来完成其它目的，比如，它可以直接将向导引导到指定的页面而不是按顺序的下一页。

例子工程的向导还有另外两个页面，CWizBkPicturePage 和 CWizFinishPage，由于它们和前面的两个页面相似，我就不再详细介绍它们，想了解它们的细节可以查看源代码。

其他的界面考虑

置中一个属性表

属性页和向导的默认位置是出现在父窗口的左上角：



这看起来有点不爽，还好有方法可以补救。第一种方法是重载 CPropertySheetImpl::PropSheetCallback()函数， 在这个函数中将属性表置中。PropSheetCallback()是 MSDN 中介绍的 PropSheetProc()的回调函数，操作系统在属性表创建 时调用这个函数，WTL 也是利用这个时间子类化属性表窗口的。所以我们的第一种尝试是：

```
class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>
{
//...
    static int CALLBACK PropSheetCallback(HWND hWnd, UINT uMsg, LPARAM
lParam)
    {
    int                                 nRet                                =
CPropertySheetImpl<CAppPropertySheet>::PropSheetCallback (
                                                hWnd, uMsg, lParam );
```

```
        if ( PSCB_INITIALIZED == uMsg )
            {
            // center sheet... somehow?
            }

        return nRet;
    }
};
```

正如你看到的，我们遇到了棘手的问题。PropSheetCallback()是一个静态方法，不能使用this 指针访问属性表窗口。那将这些代码 从CPropertySheetImpl::PropSheetCallback()中拷贝出来,然后添加我们自己的方法行不行呢？撇开刚才将代码和特定 版本的 WTL 联系在一起的方法(这已经被证明不是各好方法)，现在代码应该是这样的：

```
class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>
{
//...
    static int CALLBACK PropSheetCallback(HWND hWnd, UINT uMsg, LPARAM)
    {
        if(uMsg == PSCB_INITIALIZED)
        {
            // Code copied from WTL and tweaked to use CAppPropertySheet
            // instead of T:
            ATLASSERT(hWnd != NULL);
            CAppPropertySheet* pT = (CAppPropertySheet*)
                                    _Module.ExtractCreateWndData();
            // subclass the sheet window
            pT->SubclassWindow(hWnd);
            // remove page handles array
            pT->_CleanUpPages();

            // Our own code follows:
            pT->CenterWindow ( pT->m_psh.hwndParent );
        }

        return 0;
    }
};
```

这从理论上讲很完美，但是我试过，属性表的位置并未改变。显然，通用控件的代码在我们调用 CenterWindow()之后又改变了属性表窗口的位置。

必须放弃这个将代码封装到属性表类的方法，尽管它是个好的解决方案。我又回到原来的方案，即使用属性页窗口和属性表窗口相互协作是属性表窗口置中。我添加了一个用户定义消息UWM_CENTER_SHEET：

```
  #define UWM_CENTER_SHEET WM_APP
```

CAppPropertySheet 在它的消息映射链中处理这个消息：

```
class CAppPropertySheet : public CPropertySheetImpl<CAppPropertySheet>
{
//...
   BEGIN_MSG_MAP(CAppPropertySheet)
      MESSAGE_HANDLER_EX(UWM_CENTER_SHEET, OnPageInit)
      CHAIN_MSG_MAP(CPropertySheetImpl<CAppPropertySheet>)
   END_MSG_MAP()

   // Message handlers
   LRESULT OnPageInit ( UINT, WPARAM, LPARAM );

protected:
   bool m_bCentered;  // set to false in the ctor
};

LRESULT CAppPropertySheet::OnPageInit ( UINT, WPARAM, LPARAM )
{
   if ( !m_bCentered )
      {
      m_bCentered = true;
      CenterWindow ( m_psh.hwndParent );
      }

   return 0;
}
```

然后，每个属性页的 OnInitDialog() 方法发送这个消息到属性表窗口：

```
BOOL CBackgroundOptsPage::OnInitDialog ( HWND hwndFocus, LPARAM lParam )
{
   GetPropertySheet().SendMessage ( UWM_CENTER_SHEET );

   DoDataExchange(false);
   return TRUE;
}
```

添加 m_bCentered 标志确保属性表窗口只响应收到的第一个 UWM_CENTER_SHEET 消息。

在属性页中添加图标

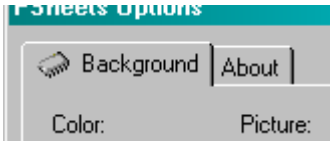
如果要使用属性表和属性页的未被成员函数封装的特性，就需要直接访问相关的数据结构：CPropertySheetImpl 类中的 PROPSHEETHEADER 类型(结构)成员 m_psh 和 CPropertyPageImpl 类中的 PROPSHEETPAGE 类型(结构)成员 m_psp。

例如：为例子中 Option 属性表中的 Background 页面添加一个图标，就需要添加一个标志并设置属性页的 PROPSHEETPAGE 结构中的几个成员：

```
CBackgroundOptsPage::CBackgroundOptsPage()
{
   m_psp.dwFlags |= PSP_USEICONID;
   m_psp.pszIcon = MAKEINTRESOURCE(IDI_TABICON);
```

```
    m_psp.hInstance = _Module.GetResourceInstance();
}
```

下面是这些代码的效果：



继续

我将在第九章介绍 WTL 的一些工具类，还有 GDI 对象和通用对话框的包装类。