
学校的理想装备
电子图书·学校专集
校园网上的最佳资源
Visual C++MFC
简明教程
BOOK
内部资料 非卖品

Visual C++ MFC 简明教程

原著：Marshall Brain　编译：张圣华

第一部分：MFC 导论

Visual C++ 不仅仅是一个编译器。它是一个全面的应用程序开发环境，使用它你充分利用具有面向对象特性的 C++ 来开发出专业级的 Windows 应用程序。为了能充分利用这些特性，你必须理解 C++ 程序设计语言。掌握了 C++，你就必须掌握 Microsoft 基本类库 (MFC) 的层次结构。该层次 结构包容了 Windows API 中的用户界面部分，并使你能够很容易地以面向对象的方式建立 Windows 应用程序。这种层次结构适用于所有版本的 Windows 并彼此兼容。你用 MFC 所建立的代码是完全可移植的。

该教程将向你介绍 MFC 的基本概念和术语以及事件驱动程序设计方法。在本节中，你将会输入、编译和运行一个简单的 MFC 程序。下一节中将向你详细解释这些代码。第三部分讨论了 MFC 控制和如何定制它们。第四部分将介绍消息映射，你将会处理 MFC 的事件。

什么是 MFC?

如果你要建立一个 Windows 应用程序，应该如何下手？

好的开端是从设计用户界面开始。首先，你要决定什么样的用户能使用该程序并根据需要来设置相应的用户界面对象。Windows 用户界面有一些标准的控制，如按钮、菜单、滚动条和列表等，这对那些 Windows 用户已经是很熟悉了。 要记住的是，作为程序员必须选择一组控制并决定如何把它们安排到屏幕上。传统上，你需要在纸上做一下用户界面的草图，直到对各元素感到满意为止。这对于一些比较小的项目，以及一些大项目的早期原型阶段是可以的。

下一步，是要实现代码。为任何 Windows 平台建立应用程序时，程序员都有两种选择：C 或 C++。 使用 C，程序员是在 Windows 应用程序界面 (API) 的水平上编写代码。该界面是由几百个 C 函数所组成,这些函数在 Windows API 参考手册中都有介绍。对于 Windows NT，API 被称为 “Win32 API”，以区别于其用于 Windows 3.1 的 16 位 API。

Microsoft 也提供了 C++ 库，它位于任何 Windows API 之上，能够使程序员的工作更容易。它就是 Microsoft 基本类库 (MFC)，该库的主要优点是效率高。它减少了大量在建立 Windows 程序时必须编写的代码。同时它还提供了所有一般 C++ 编程的优点，例如继承和封装。MFC 是可移植的，例如，在 Windows 3.1 下编写的代码可以很容易地移植到 Windows NT 或 Windows 95 上。因此，MFC 很值得推荐的开发 Windows 应用程序的方法，在本教程自始至终使用的都是 MFC。

当是使用 MFC 时，你编写的代码是用来建立必要的用户界面控制并定制其外观。同时你还要编写用来响应用户操作这些控制的代码。例如，如果用户单击一个按钮时，你应该有代码来响应。这就是事件驱动代码，它构成了所有应用程序。一旦应用程序正确的响应了所有允许的控制，它的任务也就完成了。

你可以看出，使用 MFC 进行 Windows 编程时是一件比较容易的的过程。本教程的目的是比较详细地教你如何快速建立专业级的应用程序的技术。Visual C++ 应用程序开发程序环境特别适合于使用 MFC (也有其它开发环境使用 MFC，译者注)，所以一起学习 MFC 和 Visual C++ 能够增强你的开发程序的能力。

Windows 词汇

在 Windows 用户界面和软件开发中所要用到的词汇都是基本和唯一的。对于新接触该环境的用户，下面复习几个定义以便使我们的讨论更加容易。

Windows 应用程序使用几个标准的控制：

- 静态文本标签
- 按钮
- 列表框
- 组合框(一种更高级的列表框)
- 单选按钮
- 检查按钮
- 编辑框(单行和多行)
- 滚动条

你可以通过代码或“资源编辑器”来建立这些控制，在资源编辑器中可以建立对话框和这些控制。在本教程中，我们将使用代码来建立它们。

Windows 支持几种类型的应用程序窗口。一个典型的应用程序应该活动在称为“框架窗口”中。一个框架窗口是一个全功能的主窗口，用户可以改变尺寸、最小化、最大化等。Windows 也支持两种类型的对话框：模式和无模式对话框。模式对话框一旦出现在屏幕上，只有当它退出时，屏幕上该应用程序的其余部分才能响应。无模式对话框出现在屏幕上时，程序的其余部分也可以作出响应，它就象浮动在上面一样。

最简单的 Windows 应用程序是使用单文档界面(SDI)，只有一个框架窗口。Windows 的钟表、PIF 编辑器、记事本等都是 SDI 应用程序的例子。Windows 也提供了一种称为多文档界面的组织形式，它可用于更复杂的应用程序。MDI 系统允许用户在同一应用程序中同时可以查看多个文档。例如,一个文本编辑器可以允许用户同时打开多个文本文件。使用 MDI 时，应用程序有一个主窗口，在主窗口中有一些子窗口，每个子窗口中各自包含有各自的文档。在 MDI 框架中，主窗口有一个主菜单，它对主框架中最顶端窗口有效。各子窗口都可以缩成图标或展开，MDI 主窗口也可以变成桌面上的一个图标。MDI 界面可能会给你一种第二桌面的感觉，它对窗口的管理和删除混乱的窗口有很大的帮助。

你所建立的没一个应用程序都会使用它自己的一套控制、菜单结构以及对话框。应用程序界面的好坏取决于你如何选择和组织这些界面对象。Visual C++ 中的资源编辑器可以使你能容易的建立和定制这些界面对象。

事件驱动软件和词汇

所有基于窗口的 GUI 都包含相同的基本元素，它们的操作方式都是相同的。在屏幕上，用户所看到的是一组窗口，每个窗口都包含有控制、图标、对象以及一些处理鼠标和键盘的元素。从用户角度来看，各系统的界面对象都是相同的：按钮、滚动条、图标、对话框以及下拉菜单等等。尽管这些界面元素的“外观和感觉”可能有些不同，但这些界面对象的工作方式都是相同的。例如，滚动条对于 Windows、Mac 和 Motif 可能有些不同，但他们的作用完全是一样的。

从程序员的角度来看，这些系统在概念上是相似的，尽管它们可能有很大的不同。为了建立 GUI 程序，程序员第一步要把所有需要的用户界面控制都放到窗口上。例如，如果程序员要建立一个从摄氏到华氏的转换的简单程序，则程序员所选择的用户界面对象来完成并在屏幕上把结果显示出来。在这个简单的程序中，程序员可能需要用户在一个可编辑的编辑框中输入温度值，在一个不可编辑的编辑框中显示转换结果，然后让用户可以单击一个标有“退出”的按钮来退出应用程序。

因为是用户来操作应用程序的控制，所以程序必须作出响应。所做的响应依赖于用户使用鼠标或键盘在不同控制上的操作。屏幕上的每个用户界面对象对事件的响应是不同的。例如，如果用户单击退出按钮，则该按钮必须更新屏幕、加亮它自己。然后程序必须响应退出。

Windows 所用的模式也是类似的。在一个典型的应用程序中，你将建立一个主窗口，并且在其中放置了一些用户界面控制。这些控制通常被称为子窗口——它们就象一些在主窗口中的更小更特殊的子窗口。作为程序员，你应该通过函数调用来发送信息操作这些控制、通过把信息发送给你到代码来响应用户的操作。

如果你从未做过事件驱动程序设计，则所有这些对你来说可能是很陌生的。但是，事件驱动程序设计方式是很容易理解的。具体的细节对不同的系统可能有些不同，但是其基本概念是类似的。在一个事件驱动界面中，应用程序会在屏幕上绘制几个界面对象，如按钮、文本区和菜单。应用程序通常通过一段称为事件循环的的代码来响应用户的操作。用户可以使用鼠标或键盘来任意操作屏幕上的对象。例如，用户用鼠标单击一个按钮。用鼠标单击就称为一个事件。事件驱动系统把用户的动作如鼠标单击和键盘操作定义为事件，也把系统操作如更新屏幕定义为事件。

在比较低级的编程方法中，如用 C 直接编写 Windows API 应用程序，代码量是非常大的，因为你所要照顾的细节太多了。例如，你用某种类型的结构来接收单击鼠标事件。你的事件循环中的代码会查看结构中不同域，以确定哪个用户界面对象受到了影响，然后会完成相应的操作。当屏幕上有很多对象时，应用程序会变得很大。只是简单地处理哪个对象被单击和对它需要做些什么要花费大量的代码。

幸运的是，你可以在比较高级的方法来进行编程，这就是使用 MFC。在 MFC 中，几乎所有的低级的细节处理都为你代办了。如果你把某一用户界面对象放在屏幕上，你只需要两行代码来建立它。如果用户单击一个按钮，则按钮自己会完成一切必要的操作，从更新屏幕上的外观到调用你程序中的预处理函数。该函数包含有对该按钮作出相应操作的代码。MFC 为你处理所有的细节：你建立按钮并告知它特定的处理函数，则当它被按下时，它就会调用相应的函数。第四部分介绍了怎样使用消息映射来处理事件。

例子

理解一个典型的 MFC 程序的结构和样式的最好方法是输入一段小程序，然后编译和运行它。下面的程序是一段简单的“hello world”程序。这对很多 C 程序员都是很熟悉了，让我们看一下如何用 MFC 方法来实现。如果你是第一次看到这类程序，也许比较难理解。这没关系，我们后面会详细介绍。现在你只要用 Visual C++ 环境中建立、编译和运行它就可以了。

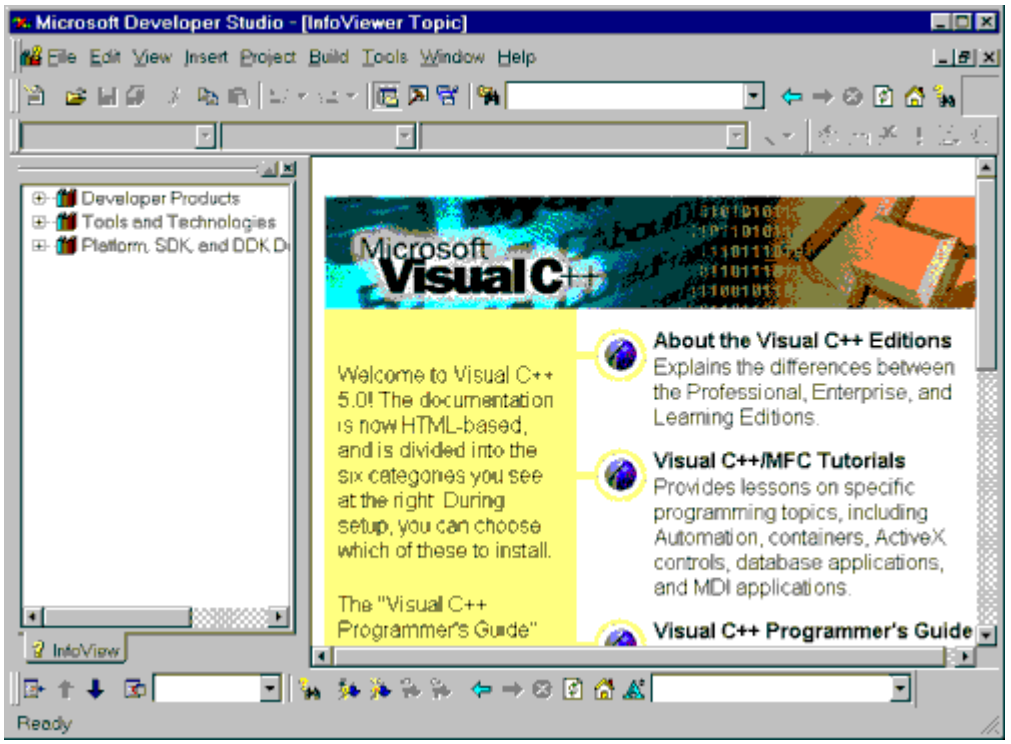
```
//hello.cpp

#include <afxwin.h>

// 说明应用程序类
class CHelloApp : public CWinApp
{
   public:
      virtual BOOL InitInstance();
};
```

```
// 建立应用程序类的实例
CHelloApp HelloApp;

// 说明主窗口类
class CHelloWindow : public CFrameWnd
{
    CStatic* cs;
    public:
        CHelloWindow();
};

// 每当应用程序首次执行时都要调用的初始化函数
BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CHelloWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

// 窗口类的构造函数
CHelloWindow::CHelloWindow()
{
    // 建立窗口本身
    Create(NULL,
           "Hello World!",
           WS_OVERLAPPEDWINDOW,
           CRect(0,0,200,200));

    // 建立静态标签
    cs = new CStatic();
    cs->Create("hello world",
               WS_CHILD|WS_VISIBLE|SS_CENTER,
               CRect(50,80,150,150),
               this);
}
```

上面的这段程序如果用 C 来实现，得需要几页的代码。这个简单的例子做了三件事。第一，它建立了一个应用程序对象。你所编写的每个 MFC 程序都有一个单一的程序对象，它是处理 MFC 和 Windows 的初始细节的。第二，应用程序建立了一个窗口来作为应用程序的主窗口。最后，在应用程序的窗口中建立了一个静态文本标签，它包含有“hello world”几个字。在第二部分中我们会仔细研究这段程序，以理解其结构。

启动 VC++，如果你是刚刚安装好，则你会在屏幕上看到一个带有工具栏的空窗口。如果 VC++ 已经在该机器上使用过了，则所显示的窗口可能有些不同，因为 VC++ 会记忆和

自动重新打开上次使用后退出时的项目和文件。我们需要的是它没有装如任何项目和代码。如果程序启动后弹出对话框指示不能打开某些文件，你只要单击“No”即可。在“Window”菜单中选取“Close All”选项关闭所有窗口。在“File”菜单中选取“Close”选项来关闭其它窗口。现在，你就处于开始状态了。如果你安装 VC++后，第一次运行，则屏幕应如下所示：



如果你以后不希望看到“InfoViewer Topic”窗口，你可以用按钮把它关掉。如果以后需要的话，你还可以单击工具栏上的“主页”按钮来打开该窗口。

现在一切都正常了。正如你所看到的，顶部是菜单和几个工具栏。左边的窗口所显示的是在线帮助内容，你可以双击某项标题来浏览其内容。在线帮助的内容是十分丰富的。

现在该做什么了？你所要做的是输入上面的程序，然后便宜并运行它。开始之前，要检查以下你的硬盘上至少要留有 5MB 的剩余空间。

建立项目和编译代码

为了在 Visual C++ 中编译代码，你必须要建立一个项目。为了这么小的程序来建立一个项目可能有点小题大作，但是，在任何实际的程序中，项目的概念是非常有用的。一个项目主要保存着下面三种不同类型的信息：

1. 它可以记住建立一个可执行程序所需要的所有源程序代码文件。在这个简单的例子中，文件 HELLO.CPP 是唯一的源文件，但是在一个大型的应用程序中，为了便于管理和维护，你可以会有许多个不同的源文件。项目会维护这些不同文件的列表，并当你要建立下一个新的可执行程序时，在必要时编译它们。
2. 它会记住针对你的应用程序所使用的编译器和连接器选项。例如，它会记住把哪个库连接到了执行程序中，你是否预编译了头文件等等。
3. 它会记住你想要建立的项目类型：一个控制台应用程序，或一个窗口应用程序等等。

如果你已经对项目文件有所了解，则会很容易明白作为机器产生的项目文件的作用。

现在，我们来建立一个简单的项目，并用它来编译 HELLO.CPP。

为此，首先从“File”菜单中选择“New”选项。在“Projects”标签中，加单击“Win32 Application”。在“Location”域中输入一个合适的路径名或单击“Browse”按钮来选择一个。在“Project name”中输入“hello”作为项目名称。这时候你会看到“hello”也会出现在“Location”域中。单击“OK”按钮。Visual C++ 会建立一个新的称为 HELLO 的目录，并把所有的项目文件 HELLO.OPT、HELLO.NCB、HELLO.DSP 和 HELLO.DSW 都放到该目录中。如果你退出，以后再重新打开该项目，则可选择 HELLO.DSW。

现在，在屏幕的左边，出现了三个标签。InfoView 标签仍然在，又新出现了 ClassView 和 FileView 标签。ClassView 标签会把你程序中所有的类都列出来，FileView 标签给出了项目中文件的列表。

现在可以输入程序的代码了。在“File”菜单中选择“New”选项来建立一个编辑窗口。在出现的对话框中，选择“Files”标签和“Text File”。则会出现 Visual C++ 的智能编辑器，你可以用它来输入上面的程序代码。输入代码时，你会发现编辑器会自动把不同类型的文本变成不同的颜色，如注释、关键字字符串等的颜色都不同。如果你要改变其颜色或关闭颜色功能，可选择“Tools”菜单中“Options”选项，然后选择“Format”标签和“Source Windows”选项就可以修改。

输入完代码后，选择“File”菜单中的“Save”选项来保存。在 Visual C++ 新建立的目录中，把它存成 HELLO.CPP 文件。

现在选择在“Project”菜单中选择“Add To Project”选项，再选“Files...”。你会看到一个对话框供你选择要添加的文件。在本例子中，选择 HELLO.CPP 文件。

在屏幕的左边，单击 FileView 标签，并双击标有 HELLO 的图标。你会看到名为 HELLO.CPP 的文件。单击 ClassView 标签，并双击文件夹图标，你会看到程序中所有的类。任何时候你都可以使用 FileView 来删除项目的文件，你只要单击该文件，然后按键盘上的 delete 键。

后，此时你必须告诉项目要使用 MFC 库。如果你忽略了这一步，则项目在连接时会出错，而出错信息对你毫无帮助。选择“Project”菜单的“Settings”。在出现的对话框中选择“General”标签。在“Microsoft Foundation Classes”组合框中，选择“Use MFC in a Shared DLL”。然后关闭对话框。

我们已经建立了项目文件，并调整了设置，你现在可以准备编译 HELLO.CPP 程序了。在“Build”菜单中，你会发现有三个不同的编译选项：

1. Compile HELLO.CPP (只有当含有 HELLO.CPP 的窗口处于激活状态时才可)
2. Build HELLO.EXE
3. Rebuild All

第一个选项只是编译源文件并形成它们的目标文件。该选项不能完成连接任务，所以它只对快速编译一些源文件以检查错误有用。第二个选项编译自上次编译后所修改的所有源文件，并连接形成可执行文件。第三个选项要重新编译和连接所有的源文件。

我们可以选择“Build HELLO.EXE”来编译和连接代码。Visual C++ 会建立一个名为“Debug”的新子目录，并把 HELLO.EXE 放在该目录中。该子目录的文件都是可以再产生的，所以你可以任意删除它们。

如果你发现了编译错误，双击输出窗口中的错误信息。这时编辑器会把你带到出错的位置处。检查你的代码是否有问题，如果有，就修改之。如果你看到大量的连接错误，则可能你在建立项目对话框中所指定的项目类型不对。你可以把该项目所在的子目录删除，然后再重新按上面的步骤来建立。

为了执行该程序，你可选则“Build”菜单中的“Execute HELLO.EXE”选项。你就可

以看到你的第一个 MFC 程序了 -- 出现一个带有“hello world”的窗口。该窗口本身带有：标题栏、尺寸缩放区、最大和最小按钮等等。在窗口上，有一个标有“hello world”。请注意，该程序是完整的。你可以移动窗口、缩放窗口、最小化等。你只使用了很少的代码就完成了一个完整的 Window 应用程序。这就是使用 MFC 的优点。所有的细节问题都有 MFC 来处理。

结论

在本讲中，你已经成功地编译和执行了你的第一个 MFC 程序。你将来会用类似的步骤来建立的应用程序。你可以为每个项目建立单独的目录，或建立一个单独的项目文件，然后再添加或删除不同的源文件。

在下一讲中，我们将仔细研究该程序，你会更完整的理解它的结构。

第二部分：一个简单的 MFC 程序

在本将中，我们将一段一段地来研究上一将中提到的 MFC 应用程序，以便能理解它的结构和概念框架。我们将先介绍 MFC，然后在介绍如何用 MFC 来建立应用程序。

MFC 简介

MFC 是一个很大的、扩展了的 C++ 类层次结构，它能使开发 Windows 应用程序变得更加容易。MFC 是在整个 Windows 家族中都是兼容的，也就是说，无论是 Windows3.x、Windows95 还是 Windows NT，所使用的 MFC 是兼容的。每当新的 Windows 版本出现时，MFC 也会得到修改以便使旧的编译器和代码能在新的系统中工作。MFC 也回得到扩展，添加新的特性、变得更加容易建立应用程序。

与传统上使用 C 语言直接访问 Windows API 相反，使用 MFC 和 C++ 的优点是 MFC 已经包含和压缩了所有标准的“样板文件”代码，这些代码是所有用 C 编写的 Windows 程序所必需的。因此用 MFC 编写的程序要比用 C 语言编写的程序小得多。另外，MFC 所编写的程序的性能也毫无损失。必要时，你也可以直接调用标准 C 函数，因为 MFC 不修改也不隐藏 Windows 程序的基本结构。

使用 MFC 的最大优点是它为你做了所有最难做的事。MFC 中包含了上成千上万行正确、优化和功能强大的 Windows 代码。你所调用的很多成员函数完成了你自己可能很难完成的工作。从这点上将，MFC 极大地加快了你的程序开发速度。

MFC 是很庞大的。例如，版本 4.0 中包含了大约 200 个不同的类。万幸的是，你在典型的程序中不需要使用所有的函数。事实上，你可能只需要使用其中的十多个 MFC 中的不同类就可以建立一个非常漂亮的程序。该层次结构大约可分为几种不同的类型的类：

- 应用程序框架
- 图形绘制的绘制对象
- 文件服务
- 异常处理
- 结构 - List、Array 和 Map
- Internet 服务
- OLE 2
- 数据库
- 通用类

在本教程中，我们将集中讨论可视对象。下面的列表给出了部分类：

- CObject
- CCmdTarget
- CWinThread
- CWinApp
- CWnd
- CFrameWnd
- CDialog
- CView
- CStatic
- CButton
- CListBox
- CComboBox
- CEdit
- CscrollBar

在上面的列表中，有几点需要注意。第一，MFC 中的大部分类都是从基类 CObject 中继承下来的。该类包含有大部分 MFC 类所通用的数据成员和成员函数。第二，是该列表的简单性。CWinApp 类是在你建立应用程序是要用到的，并且任何程序中都只用一次。CWnd 类汇集了 Windows 中的所有通用特性、对话框和控制。CFrameWnd 类是从 CWnd 继承来的，并实现了标准的框架应用程序。CDialog 可分别处理无模式和有模式两种类型的对话框。CView 是用于让用户通过窗口来访问文档。最后，Windows 支持六种控制类型：静态文本框、可编辑文本框、按钮、滚动条、列表框和组合框(一种扩展的列表框)。一旦你理解了这些，你也就能更好的理解 MFC 了。MFC 中的其它类实现了其它特性，如内存管理、文档控制等。

为了建立一个 MFC 应用程序，你既要会直接使用这些类，而通常你需要从这些类中继承新的类。在继承的类中，你可以建立新的成员函数，这能更适用你自己的需要。你在第一讲中的简单例子中已经看到了这种继承过程，下面会详细介绍。CHelloApp 和 CHelloWindow 都是从已有的 MFC 类中继承的。

设计一个程序

在讨论代码本身之前，我们需要花些工夫来简单介绍以下 MFC 中程序设计的过程。例如，假如你要编一个程序来向用户显示“Hello World”信息。这当然是很简单的，但仍需要一些考虑。

“hello world”应用程序首先需要在屏幕上建立一个窗口来显示“hello world”。然后需要实际把“hello world”放到窗口上。我们需要但个对象来完成这项任务：

1. 一个应用程序对象，用来初始化应用程序并把它挂到 Windows 上。该应用程序对象处理所有的低级事件。
2. 一个窗口对象来作为主窗口。
3. 一个静态文本对象，用来显示“hello world”。

你用 MFC 所建立的每个程序都会包含头两个对象。第三个对象是针对该应用程序的。每个应用程序都会定义它自己的一组用户界面对象，以显示应用程序的输出和收集应用的输入信息。

一旦你完成了界面的设计，并决定实现该界面所需要的控制，你就需要编写代码来在屏幕上建立这些控制。你还会编写代码来处理用户操作这些控制所产生的信息。在“hello

world”应用程序中，只有一个控制。它用来输出“hello world”。复杂的程序可能在其主窗口和对话框中需要上百个控制。

应该注意，在应用程序中有两种不同的方法来建立用户控制。这里所介绍的是用 C++ 代码方式来建立控制。但是，在比较大的应用程序中，这种方法是不可行的。因此，在通常情况下要使用资源文件的图形编辑器来建立控制。这种方法要方便得多。

理解“hello world”的代码

下面列出了你在上一讲中已经输入、编译和运行的“hello world”程序的代码。添加行号是为了讨论方便。我们来一行行地研究它，你会更好的理解 MFC 建立应用程序的方式。

如果你还没有编译和运行该代码，应该按上一讲的方法去做。

```
1 //hello.cpp

2 #include <afxwin.h>

3 // Declare the application class
4 class CHelloApp : public CWinApp
5 {
6   public:
7       virtual BOOL InitInstance();
8 };

9 // Create an instance of the application class
10 CHelloApp HelloApp;

11 // Declare the main window class
12 class CHelloWindow : public CFrameWnd
13
14  CStatic* cs;
15  public:
16  CHelloWindow();
17 };

18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22  m_pMainWnd = new CHelloWindow();
23  m_pMainWnd->ShowWindow(m_nCmdShow);
24  m_pMainWnd->UpdateWindow();
25  return TRUE;
26 }

27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
```

```
29
30  // Create the window itself
31  Create(NULL,
32      "Hello World!",
33      WS_OVERLAPPEDWINDOW,
34      CRect(0,0,200,200));
35  // Create a static label
36  cs = new CStatic();
37  cs->Create("hello world",
38      WS_CHILD|WS_VISIBLE|SS_CENTER,
39      CRect(50,80,150,150),
40      this);
41 }
```

你把上面的代码看一遍，以得到一整体印象。该程序由六小部分组成，每一部分都起到很重要的作用。

首先，该程序包含了头文件 afxwin.h (第 2 行)。该头文件包含有 MFC 中所使用的所有的类型、类、函数和变量。它也包含了其它头文件，如 Windows API 库等。

第 3 至 8 行从 MFC 说明的标准应用程序类 CWinApp 继承出了新的应用程序类 CHelloApp。该新类是为了要重载 CWinApp 中的 InitInstance 成员函数。InitInstance 是一个应用程序开始执行时要调用的可重载函数。

在第 10 行中，说明了应用程序作为全局变量的一个事例。该实例是很重要的，因为它要影响到程序的执行。当应用程序被装入内存并开始执行时，全局变量的建立会执行 CWinApp 类的缺省构造函数。该构造函数会自动调用在 18 至 26 行定义的 InitInstance 函数。

在第 11 至 17 中,CHelloWindow 类是从 MFC 中的 CFrameWnd 类继承来的。CHelloWindow 是作为应用程序在屏幕上的窗口。建立新的类以便实现构造函数、析构函数和数据成员。

第 18 至 26 行实现了 InitInstance 函数。该函数产生一个 CHelloWindow 类的事例，因此会执行第 27 行至 41 行中类的构造函数。它也会把新窗口放到屏幕上。

第 27 至 41 实现了窗口的构造函数。该构造函数实际是建立了窗口，然后在其中建立一个静态文本控制。

要注意的是，在该程序中没有 main 或 WinMain 函数，也没有事件循环。然而我们从上一讲在执行中知道它也处理了事件。窗口可以最大或最小化、移动窗口等等。所有这些操作都隐藏在主应用程序类 CWinApp 中，并且我们不必为它的事件处理而操心，它都是自动执行、在 MFC 中不可见的。

下一节中，将详细介绍程序的各部分。你可能不能马上全都理解得很好：但你最好先读完它以获得第一印象。在下一讲中，会介绍一些特殊的例子，并偶把各片段组合在一起，有助于你能更好的理解。

程序对象

用 MFC 建立的每个应用程序都要包括一个单一从 CWinApp 类继承来的应用程序对象。该对象必须被说明成全局的(第 10 行)，并且在你的程序中只能出现一次。

从 CWinApp 类继承的对象主要是处理应用程序的初始化，同时也处理应用程序主事件循环。CWinApp 类有几个数据成员和几个成员函数。在上面的程序中，我们只重载了一个 CWinApp 中的虚拟函数 InitInstance。

应用程序对象的目的是初始化和控制你的程序。因为 Windows 允许同一个应用程序的多个事例在同时执行，因此 MFC 把初始化过程分成两部分并使用两个函数 InitApplication 和 InitInstance 来处理它。此处，我们只使用了一个 InitInstance 函数，因为我们的程序很简单。当每次调用应用程序时都会调用一个新的事例。第 3 至 8 行的代码建立了一个称为 CHelloApp 的类，它是从 CWinApp 继承来的。它包含一个新的 InitInstance 函数，是从 CWinApp 中已存在的函数(不做任何事情)重载来的:

```
3 // Declare the application class
4 class CHelloApp : public CWinApp
5 {
6   public:
7       virtual BOOL InitInstance();
8 };
```

在重载的 InitInstance 函数内部，第 18 至 26 行，程序使用 CHelloApp 的数据成员 m_pMainWnd 来建立并显示窗口:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22  m_pMainWnd = new CHelloWindow();
23  m_pMainWnd->ShowWindow(m_nCmdShow);
24  m_pMainWnd->UpdateWindow();
25  return TRUE;
26 }
```

InitInstance 函数返回 TRUE 表示初始化已成功的完成。如果返回了 FALSE，则表明应用程序会立即终止。在下一节中我们将会看到窗口初始化的详细过程。

当应用程序对象在第 10 行建立时，它的数据成员(从 CWinApp 继承来的) 会自动初始化。例如，m_pszAppName、m_lpCmdLine 和 m_nCmdShow 都包含有适当的初始化值。你可参见 MFC 的帮助文件来获得更详细的信息。我们将使用这些变量中的一个。

窗口对象

MFC 定义了两个类型的窗口: 1) 框架窗口，它是一个全功能的窗口，可以改变大小、最小化、最大化等等; 2) 对话框窗口，它不能改变大小。框架窗口是典型的主应用程序窗口。

在下面的代码中，从 CFrameWnd 中继承了一个新的类 CHelloWindow:

```
11 // Declare the main window class
12 class CHelloWindow : public CFrameWnd
13
14  CStatic* cs;
15  public:
16  CHelloWindow();
17 };
```

它包括一个新的构造函数，同时还有一个指向程序中所使用的唯一用户界面控制的数据成员。你多建立的每个应用程序在主窗口中都会有唯一的一组控制。因此，继承类将有一个重载的构造函数以用来建立主窗口所需要的所有控制。典型情况下，该类会包含有一

个析构函数以便在窗口关闭时来删除他们。我们这里没有使用析构函数。在第四讲中，我们将会看到继承窗口类也会说明一个消息处理函数来处理这些控制在响应用户事件所产生的消息。

典型地，一个应用程序将有一个主应用程序窗口。因此，CHelloApp 应用程序类定义了一个名为 m_pMainWnd 成员变量来指向主窗口。为了建立该程序的主窗口，InitInstance 函数(第 18 至 26 行)建立了一个 CHelloWindow 事例，并使用 m_pMainWnd 来指向一个新的窗口。我们的 CHelloWindow 对象是在第 22 行建立的:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22  m_pMainWnd = new CHelloWindow();
23  m_pMainWnd->ShowWindow(m_nCmdShow);
24  m_pMainWnd->UpdateWindow();
25  return TRUE;
26 }
```

只建立一个简单的框架窗口是不够的。还要确保窗口能正确地出现在屏幕上。首先，代码必须要调用窗口的 ShowWindow 函数以使窗口出现在屏幕上(第 23 行)。其次，程序必须要调用 UpdateWindow 函数来确保窗口中的每个控制和输出能正确地出现在屏幕上(第 24 行)。

你可能奇怪，ShowWindow 和 UpdateWindow 函数是在哪儿定义的。例如，如果你要查看以便了解它们,你可能要查看 MFC 的帮助文件中的 CFrameWnd 定义部分。但是 CFrameWnd 中并不包含有这些成员函数。CFrameWnd 是从 CWnd 类继承来的。你可以查看 MFC 文档中的 CWnd，你会发现它包含有 200 多个不同的成员函数。显然，你不能在几分钟内掌握这些函数，但是你可以掌握其中的几个，如 ShowWindow 和 UpdateWindow。

现在让我们花几分钟来看一下 MFC 帮助文件中的 CWnd::ShowWindow 函数。为此，你你可以单击帮助文件中的 Search 按钮，并输入“ ShowWindow ”。找到后，你会注意到，ShowWindow 只有一个参数，你可以设置不同的参数值。我们把它设置成我们程序中 CHelloApp 的数据成员变量 m_nCmdShow (第 23 行)。m_nCmdShow 变量是用来初始化应用程序启动的窗口显示方式的。例如，用户可能在程序管理器中启动应用程序，并可通过应用程序属性对话框来告知程序管理器应用程序在启动时要保持最小化状态。m_nCmdShow 变量将被设置成 SW_SHOWMINIMIZED，并且应用程序会以图标的形式来启动，也就是说，程序启动后，是一个代表该程序的图标。m_nCmdShow 变量是一种外界与应用程序通讯的方式。如果你愿意，你可以用不同的 m_nCmdShow 值来试验 ShowWindow 的效果。但要重新编译程序才能看到效果。

第 22 行是初始化窗口。它为调用 new 函数分配内存。在这一点上，程序在执行时会调用 CHelloWindow 的构造函数。该构造函数在每次带类的事例被分配时都要调用。在窗口构造函数的内部，窗口必须建立它自己。它是通过调用 CFrameWnd 的 Create 成员函数来实现的(第 31 行):

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29
30  // Create the window itself
31  Create(NULL,
```

```
32      "Hello World!",
33      WS_OVERLAPPEDWINDOW,
34      CRect(0,0,200,200));
```

建立函数共传递了四个参数。通过查看 MFC 文档，你可以了解不同类型。NULL 参数表示使用缺省的类名。第二个参数为出现在窗口标题栏上的标题。第三个参数为窗口的类型属性。该程序使用了正常的、可覆盖类型的窗口。在下一讲中将详细介绍类型属性。第四个参数指出窗口应该放在屏幕上的位置和大小，左上角为(0,0)，初始化大小为 200×200 个象素。如果使用了 rectDefault，则 Windows 会为你自动放置窗口及大小。

因为我们的程序太简单了，所以它只在窗口中建立了一个静态文本控制。见第 35 至 40 行。下面将详细介绍。

静态文本控制

程序在从 CFrameWnd 类中继承 CHelloWindow 类时(第 11 至 17 行)时，说明了一个成员类型 CStatic 及其构造函数。

正如在前面所见到的，CHelloWindow 构造函数主要做两件事情。第一是通过调用 Create 函数(第 31 行)来建立应用程序的窗口。然后分配和建立属于窗口的控制。在我们的程序中，只使用了一个控制。在 MFC 中建一个对象总要经过两步。第一是为类的事例分配内存，然后是调用构造函数来初始化变量。下一步，调用 Create 函数来实际建立屏幕上的对象。代码使用这两步分配、构造和建立了一个静态文本对象(第 36 至 40 行)：

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29
30  // Create the window itself
31  Create(NULL,
32      "Hello World!",
33      WS_OVERLAPPEDWINDOW,
34      CRect(0,0,200,200));
35  // Create a static label
36  cs = new CStatic();
37  cs->Create("hello world",
38      WS_CHILD|WS_VISIBLE|SS_CENTER,
39      CRect(50,80,150,150),
40      this);
41 }
```

CStatic 构造函数是在为其分配内存时调用的，然后就调用了 Create 函数来建立 CStatic 控制的窗口。Create 函数所使用的参数与窗口建立函数所使用的参数是类似的(第 31 行)。第一个参数指定了控制中所要显示的文本内容。第二个参数指定了类型属性。类型属性在下一讲中将详细介绍。在次我们使用的是子窗口类型(既在别的窗口中显示的窗口)，还有它是可见的，还有文本的显示位置是居中的。第三个参数决定了控制的大小和位置。第四参数表示该子窗口的父窗口。已经建立了一个静态控制，它将出现在应用程序窗口上，并显示指定的文本。

结论

第一次浏览该代码，也可能不是很熟悉和有些让人烦恼。但是不要着急。从程序员的

观点来看，整个程序的主要工作就是建立了 CStatic 控制(36 至 40 行)。在下一讲中，我们详细向你介绍 36 至 40 行代码的含义，并可看到定制 CStatic 控制的几个选项。

第三部分：MFC 样式

控制是用来建立 Windows 应用程序用户界面的用户界面对象。你所见到的大部分 Windows 应用程序和对话框只不过是由一些控制所组成的、用来实现程序功能的东西。为了建立有效的应用程序，你必须完全理解在 Windows 应用程序中应该如何合理的使用控制。有六个基本的控制：CStatic、CButton、CEdit、CList、CComboBox 和 CScrollBar。另外，Windows 95 又增加了 15 增强了的控制。你需要理解的是那个控制能做些什么、你应该如何控制它的外表和行为以及如何让控制能响应用户事件。只要掌握了这些，再加上掌握了菜单和对话框，你就可以建立你所想象的任何 Windows 应用程序。你可以象本教程这样用程序代码来建立控制，也可以使用资源编辑器通过资源文件来建立。当然，对话框编辑器更方便些，它对于已经基本掌握了控制的情况下特别有用。

最简单的控制是 CStatic，它是用来显示静态文本的。CStatic 类没有任何数据成员，它只有少量的成员函数：构造函数、Create 函数(用于获取和设置静态控制上的图标)等等。它不响应用户事件。因为它的简单性，所以最好把它作为学习 Windows 控制的开端。

在本讲中，我们从 CStatic 着手，看一下如何修改和定制控制。在下一讲中，我们将学习 CButton 和 CScrollBar 类，以理解事件处理的概念。一旦你理解和掌握了所有控制极其类，你就可以建立完整的应用程序了。

基　础

MFC 中的 CStatic 类是用来显示静态文本信息的。这些信息能够可以作为纯信息(例如，显示在信息对话框中的错误消息)，或作为小的标签等。在 Windows 应用程序的文件打开对话框中，你会发现有六个这样的标签。

CStatic 控制还有几种其它的显示格式。你可以通过修改标签的样式来使它表现为矩形、边框或图标等。

CStatic 控制总是作为子窗口的形式出现的。典型情况下，其父窗口是应用程序的主窗口或对话框。正如上一讲所介绍的，你用两行代码就可以建立一个静态控制：

```
CStatic *cs;
...
cs = new CStatic();
cs->Create("hello world",
   WS_CHILD|WS_VISIBLE|SS_CENTER,
   CRect(50,80, 150, 150),
   this);
```

这两行代码是典型的 MFC 建立所有控制的代码。调用 new 来为 CStatic 类的事例分配内存，然后调用类的构造函数。构造函数是用来完成类所需要的初始化功能的。Create 函数建立控制并把它放到屏幕上。

Create 函数有五个参数：

lpszText: 指定了要显示的文本。

rect:　　控制文本区域的位置、大小和形状。

pParentWnd: 指明 CStatic 控制的父窗口。该控制会出现在其父窗口中，且其位置是

相对于其父窗口的用户区域而言的。

nID:　整数值，表示该控制的标识符。

dwStyle: 最重要的参数。它控制着控制的外观和行为。

CStatic 样式

所有的控制都有各种显示样式。样式是在用 Create 函数建立控制时传递给它的 dwStyle 参数所决定的。对 CStatic 有效的样式简介如下：

从 CWnd 继承来的样式:

- WS_CHILD　CStatic 所必须的。
- WS_VISIBLE　表示该控制对用户应该是可见的。
- WS_DISABLED 表示该控制拒绝接受用户事件。
- WS_BORDER　控制的文本区域带有边框。

CStatic 固有的样式:

- SS_BLACKFRAME 该控制区域以矩形边界显示。颜色与窗口框架相同。
- SS_BLACKRECT ？该控制以填充的矩形显示。颜色与当前的窗口框架相同。
- SS_CENTER 文本居中。
- SS_GRAYFRAME　控制以矩形边框方式显示。颜色与当前桌面相同。
- SS_GRAYRECT　该控制以填充的矩形显示。颜色与当前的桌面相同。
- SS_ICON　控制以图标形式显示。文本作为图标在资源文件的名称。rect 参数只控制位置。
- SS_LEFT　文本居左显示。文字可回绕。
- SS_LEFTNOWORDWRAP 文本居左显示。多余的文字被剪裁。
- SS_NOPREFIX 表示字符串中的"&"字符不表示为加速前缀。
- SS_RIGHT　文本居右显示。文字可回绕。
- SS_SIMPLE　只简单的显示一行文本。任何 CTLCOLOR 信息都被其父窗口忽略。
- SS_USERITEM 用户定义项。
- SS_WHITEFRAME 控制以矩形边框方式显示。颜色与当前窗口背景颜色相同。
- SS_WHITERECT　控制以填充矩形方式显示。颜色与当前窗口背景颜色相同。

这些常数中，“SS”(Static Style)开头的表示只能用于 CStatic 控制。以“WS”(Window Style)开头的常数表示可适用于所有窗口，它们定义在 CWnd 对象中。CWnd 中还有很多以“WS”样式常数。你可以在 MFC 文档中的 CWnd::Create 函数中找到它们。上面的四种是只用于 CStatic 对象的。

CStatic 对象至少要带有两个样式：WS_CHILD 和 WS_VISIBLE。该控制必须作为另一窗口的子窗口来建立。如果不使用 WS_VISIBLE，则所建立的控制是看不见的。WS_DISABLED 控制着标签对事件的响应，因为 CStatic 不接收键盘或鼠标事件，所以使用该项是多余的。

所有的其它样式选项都是可选的，它们控制着标签的外观。在 CStatic::Create 函数中使用这些控制，可以控制 CStatic 在屏幕上的显示。

CStatic 文本的外观

下面的代码对于理解 CStatic 是有帮助的。它与上一讲中介绍的代码类似，但是修改了 CStatic 的建立部分。

```
//static1.cpp
#include <afxwin.h>
```

```
// Declare the application class
class CTestApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};

// Create an instance of the application class
CTestApp TestApp;

// Declare the main window class
class CTestWindow : public CFrameWnd
{
CStatic* cs;
public:
CTestWindow();
};

// The InitInstance function is called
// once when the application first executes
BOOL CTestApp::InitInstance()
{
m_pMainWnd = new CTestWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}

// The constructor for the window class
CTestWindow::CTestWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CStatic Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));

// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);

// Create a static label
cs = new CStatic();
```

```
    cs->Create("hello world",
        WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
        r,
        this);
    }
```

下面是窗口构造函数加上了行编号：

```
    CTestWindow::CTestWindow()
    {
        CRect r;

        // Create the window itself
    1       Create(NULL,
            "CStatic Tests",
            WS_OVERLAPPEDWINDOW,
            CRect(0,0,200,200));
        // Get the size of the client rectangle
    2       GetClientRect(&r);
    3       r.InflateRect(-20,-20);
        // Create a static label
    4       cs = new CStatic();
    5       cs->Create("hello world",
            WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
            r,
            this);
    }
```

首先在单击 1 行中调用 CTestWindow::Create 函数。它是 CFrameWnd 对象的 Create 函数，因为 CTestWindow 从 CFrameWnd 继承了其行为。所以第一行中的代码指定了窗口大小应该为 200×200 个象素，窗口的左上角被初始化在屏幕的 0,0 位置处。常数 rectDefault 可用 CRect 参数来替代。

在第 2 行，调用了 CTestWindow::GetClientRect，向它传递了&r 参数。GetClientRect 函数是从 CWnd 类继承来的。变量 r 是 CRect 类型的，并且在函数的开头部分被说明为局部变量。

理解这段代码时可能会有两个问题 1) GetClientRect 函数是干什么的? 2) CRect 变量是干什么的? 让我们先回答第一个问题。当你查看 MFC 文档中的 CWnd::GetClientRect 函数时，你会发现它返回一 CRect 类型，它包含了指定窗口的用户区域矩形。它保存的是参数的地址&r。该地址指向 CRect 的位置。CRect 类型是在 MFC 中定义的。用它处理矩形是非常方便的。如果你看以下 MFC 文档，就会看到其中定义了 30 多种处理矩形的成员函数和操作符。

在我们的情况下，我们要在窗口中间显示“ Hello World ”。因此，我们用 GetClientRect 来获取用户区域的矩形坐标。在第 3 行中调用了 CRect::InflateRect，同时还可以增大或减少了矩形的尺寸(参见 CRect::DeflateRect)。这里我们对矩形的各边减少了 20 个象素。如果不这样的话，标签周围边界就会超出窗口框架。

实际上，CStatic 是在第 4 和 5 行建立的。样式属性为居中并有边框。其大小和位置由 CRect 参数 r 确定的。

通过修改不同的样式属性，你可以理解 CStatic 的不同形式。例如，下面的代码包含有对 CTestWindow 构造函数进行了修改，所产生的控制有个位移：

```
CTestWindow::CTestWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CStatic Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));

// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);

// Create a static label
cs = new CStatic();
cs->Create("Now is the time for all good men to \
come to the aid of their country",
    WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
    r,
    this);
}
```

上面的代码除了所显示的文本比较长外没有什么不同。运行该代码你就可以看到，CStatic 在指定的区域内的文本已经回绕了，且没一行都是居中的。

如果边框矩形太小不能包含所有的文本行，则文本会被剪切以适应之。你减小矩形大小或增大字符串长度就可以看到 CStatic 的该特性。

在我们所看到的所有代码中，样式 SS_CENTER 是用来居中文本的。CStatic 也允许左对齐或右对齐。左对齐是用 SS_LEFT 来替代 SS_CENTER 属性。同样，右对齐是用 SS_RIGHT 来取代之。

SS_LEFTNOWORDWRAP 属性是用来关闭文本回绕的。它会强迫使用左对齐属性。

CStatic 的矩形显示模式

CStatic 也支持两种不同的矩形显示模式：填充矩形和框架。通常用这两种模式来把一组控制框在一起。例如，你可以把黑背景框架窗口作为一组编辑框的背景。你可以选择六种不同的样式：SS_BLACKFRAME、SS_BLACKRECT、SS_GRAYFRAME、SS_GRAYRECT、SS_WHITEFRAME 和 SS_WHITERECT。RECT 形成了一个填充的矩形，而 FRAME 组成一边框。其中的颜色标志，如 SS_WHITERECT 表示其颜色与窗口背景的颜色是相同的。尽管该颜色的缺省值是白色，但你可以使用控制面板来改变，此时矩形的颜色可能就不是白色的了。

当指定了矩形或框架属性后，CStatic 的文本字符串会被忽略。典型情况是传递一空字符串。你可以试验以下这些特性。

字体

你可以使用 CFont 类来改变 CStatic 的字体。MFC 中的 CFont 类保存着特殊 Windows

字体的单一实例。例如，一个实例的 CFont 类可能保存有 18 点的 Times 字体，而另一个可能保存着 10 点的 Courier 字体。你可以调用 SetFont 函数来修改字体。下面的代码给出了如何实现字体。

```
CTestWindow::CTestWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CStatic Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);
// Create a static label
cs = new CStatic();
cs->Create("Hello World",
    WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
    r,
    this);

// Create a new 36 point Arial font
font = new CFont;
font->CreateFont(36,0,0,0,700,0,0,0,
                ANSI_CHARSET,OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY,
                DEFAULT_PITCH|FF_DONTCARE,
                "arial");
// Cause the label to use the new font
cs->SetFont(font);
}
```

上面的代码开始于建立窗口和 CStatic。然后建立一 CFont 类型对象。字体变量应作为 CTestWindow 的数据成员来说明“ CFont *font ”。CFont::CreateFont 函数有 15 个参数，但是只有三个是最常用的。例如，36 指定了以点为单位的字体大小，700 指定了字体的密度(400 是正常“ normal ”, 700 为加黑“ bold ”,值的范围为 1 到 1000。FW_NORMAL 和 FW_BOLD 的含义实际上是相同的),“ arial ”是所用字体的名称。Windows 通常带有五种 True Type 字体(Arial、Courier New、Symbol、Times New Roman 和 Wingdings)，使用它们，你可以确保任何机器上都会有该字体。如果你使用了系统不知道的字体，则 CFont 会选择缺省字体，正如你在本教程所见到的。

要想更多的了解 CFont 类，可参见 MFC 文档。在 API 在线帮助文件中，有一篇文章对字体做了很好的概述。查找“ Fonts and Text Overview ”。

SetFont 函数是从 CWnd 继承来的。它是用来设置窗口的字体的，在我们的程序中是 CStatic 子窗口。你可能要问：“ 我怎样知道 CWnd 中的哪些函数可以用于 CStatic 在? ”你

只能在实践中来学习。花上一些时间来看一下 CWnd 的所有函数。你定会有所收获，并会发现哪些函数可用于定制控制。我们在选下一讲中看到 CWnd 类中的其它 Set 函数。

结论

在本教程中，我们勘察了 CStatic 的很多不同特性。有关从 CWnd 继承来的 Set 函数，我们将放到下一讲介绍，因为在那里更合适。

查看 Microsoft 文档中的函数

在 Visual C++ 5.x 中，查找你多不熟悉的函数是很简单的。所有的 MFC、SDK、Windows API 和 C/C++标准库函数都继承到同一个帮助系统中了。如果你不能确定所要的函数在哪儿，你可以使用帮助菜单中的 Search 选项来查找。所有相关的函数都会列出来的。

编译多个可执行程序

在本教程中，有几个例子程序。有两种方式来编译和运行它们。第一种方式是把每个程序都放到自己的目录中，然后为每个程序分别建立一个项目。使用该技术，你可以分别编译每个程序，并且可以同时或独立地使用他们。该方法的缺点是需要比较大的磁盘空间。

第二种方法是为所有的程序只建立一个目录。你可以一个项目文件。为了编译每个程序，你可以编辑项目和改变源文件。当你重新编译项目时，新的可执行程序就是你所选择的源文件的。该方法可以使用减少磁盘空间。

第四部分：消息映射

应用程序放在窗口中的任何用户界面对象都具有两种可控制的特性 :1) 它的外观,2) 它响应事件的行为。在上一讲中，你已经学习了 CStatic 控制和如何使用样式属性来定制用户界面对象的外观。这些概念可用于 MFC 中的所有不同控制类。

在本讲中，我们将介绍 CButton 控制，以理解消息映射和简单的事件处理。然后还要介绍使用 CScrollBar 控制的稍微复杂点的例子。

理解消息映射

在第二讲中，MFC 程序不包括主要函数或时间循环。所有的事件处理都是作为 CWinApp 的一部分在后台处理的。因为它们是隐藏的，所以我们需要一种方法来告诉不可见的时间循环通告我们应用程序所感兴趣的事件。这需要一种叫做消息映射的机制。消息映射识别感兴趣的事件然后调用函数来响应这些事件。

例如，如果你要编写一个程序，当用户按下标有“退出”的按钮时要退出应用程序。在程序中，你编写代码来建立按钮：你指示按钮应如何动作。然后，为其父窗口建立用户单击按钮时的消息映射，它试图要传递消息给其父窗口。为了建立父窗口的消息，你要建立截取消息映射的机制，并且使用按钮的消息。当一指定的按钮事件发生时，消息映射会请求 MFC 调用一指定的函数。在这种情况下，单击退出按钮就是所感兴趣的事件。然后你把退出应用程序的代码放到指定的函数中。

其它的工作就由 MFC 来做了。当程序执行时，用户单击“退出”按钮时，按钮就会自己加亮。然后 MFC 自动调用相应的函数，并且程序会终止。只使用很少的几行代码你就响应了用户事件。

CButton 类

在上一讲中所讨论的 CStatic 控制是唯一不响应用户时间的控制。Windows 中所有的其它控制都可响应用户事件。第一，当用户处理它们时，它们会自动更新其外观(例如，当用户单击按钮时，按钮会自己加亮以给用户一个反馈)。第二，每个不同的控制都要发送信息给你的代码以使程序能响应用户的需要。例如，当单击按钮时，按钮就会发送一个命令消息。如果你编写代码接收消息，则你的代码就能响应用户事件。

为了理解这个过程，我们从 CButton 控制开始。下面的代码说明了建立按钮的过程：

```
// button1.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
// Declare the application class
class CButtonApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
CButton *button;
public:
CButtonWindow();
};
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CButton Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
```

```
    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);

    // Create a button
    button = new CButton();
    button->Create("Push me",
        WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
        r,
        this,
        IDB_BUTTON);
    }
```

上面的代码与前面介绍的代码几乎相同。CButton 类的 Create 函数共有 5 个参数。前四个与 CStatic 的相同。第五个参数为按钮的资源 ID。资源 ID 是用来标识消息映射中按钮的唯一整数值。常数值 IDB_BUTTON 已经在程序的顶部做了定义。“IDB_”是任选的，只是该常量 ID 是用来表示按钮的。它的值为 100，因为 100 以内的值都为系统所保留。你可以使用任何大于 99 的值。

CButton 类所允许的样式属性与 CStatic 类的是不同的。定义了 11 个不同的“BS”(“Button Style”)常量。完整的“BS”常量列表可在用 Search 命令查找 CButton，并选择“button style”。这里我们要用的是 BS_PUSHBUTTON 样式，它表示我们要一正常的的按钮方式来显示该按钮。我们还使用了两个熟悉的“WS”属性：WS_CHILD 和 WS_VISIBLE。我们将在后面介绍其它一些样式。

当你运行代码时，会注意到按钮响应了用户事件。既它加亮了。除此之外它没有做任何事情，因为我们还没有教它怎样去做。我们需要编写消息映射来使按钮做一些感兴趣的事情。

建立消息映射

下面的代码包含有消息映射，也包含有新的处理单击按钮的函数(当用户单击按钮时会响一下喇叭)。它只是前面代码的一个简单的扩充：

```
    // button2.cpp
    #include <afxwin.h>
    #define IDB_BUTTON 100
    // Declare the application class
    class CButtonApp : public CWinApp
    {
    public:
    virtual BOOL InitInstance();
    };
    // Create an instance of the application class
    CButtonApp ButtonApp;
    // Declare the main window class
    class CButtonWindow : public CFrameWnd
    {
    CButton *button;
```

```
public:
CButtonWindow();
afx_msg void HandleButton();
DECLARE_MESSAGE_MAP()
};
// The message handler function
void CButtonWindow::HandleButton()
{
MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
ON_BN_CLICKED(IDB_BUTTON, HandleButton)
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CButton Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);
// Create a button
button = new CButton();
button->Create("Push me",
    WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
    r,
    this,
    IDB_BUTTON);
}
```

主要修改了三个方面:

1. CButtonWindow 的类说明现在包含了一个新的成员函数和一个新的表示消息映射的宏。HandleButton 函数是正常的 C++函数，它通过 afx_msg 标签确定为消息处理函数。该函数需要一些特殊的约束，例如，它必须是 void 型并且它不能接收任何参数。DECLARE_MESSAGE_MAP 宏建立了消息映射。函数和宏都必须是 public 型的。
2. HandleButton 函数作为成员函数以同样的方式来建立。在该函数中，我们调用了 Windows API 中的 MessageBeep 函数。
3. 用宏来建立消息映射。在代码中，你可以看见 BEGIN_MESSAGE_MAP 宏接收两各参数。第一个指定了使用消息映射的类的名称。第二个是基类。然后是 ON_BN_CLICKED 宏，接受两个参数控制的 ID 和该 ID 发送命令消息时所调用的函数。最后，消息映射用 END_MESSAGE_MAP 来结束。

当用户单击按钮时，它向其包含该按钮的父窗口发送了一个包含其 ID 的命令消息。那是按钮的缺省行为，这就是该代码工作的原因。按钮向其父窗口发送消息，是因为它是子窗口。父窗口截取该消息并用消息映射来确定所要调用的函数。MFC 来安排，只要指定的消息一出现，相应的函数就会被调用。

ON_BN_CLICKED 消息是 CButton 发送的唯一感兴趣的消息。它等同于 CWnd 中的 ON_COMMAND 消息，只是一个更简单方便的同义词而已。

改变大小的消息

在上面的代码中，由于有了消息映射，从 CFrameWnd 继承来的应用程序窗口认出按钮有按钮产生的单击消息并响应之。加入消息映射的 ON_BN_CLICKED 宏指定了按钮的 ID 和窗口在接收到来自按钮的命令消息时应调用的函数。因为只要用户单击了按钮，按钮就会自动把其 ID 发送父窗口，这样才能允许代码正确地处理按钮事件。

作为该应用程序的主窗口的框架窗口自己也有传递消息的能力。大约有 100 不同的消息可用，它们都是从 CWnd 类继承来的。从 MFC 帮助文件中浏览 CWnd 类的成员函数，你就会看到所有的这些消息。查看所有以“On”开头的成员函数。

你可能已经注意到了，至今为止所有的代码都不能很好地处理尺寸变化。当窗口变化大小时，窗口的框架会做相应的调整，但是窗口中调的内容仍原处不动。可以通过处理尺寸变化的事件来更好的处理这一问题。任何窗口发送的消息之一就是变尺寸消息。该消息是当改变形状时发出的。我们可以使用该消息来控制框架中子窗口的大小，如下所示：

```
// button3.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
// Declare the application class
class CButtonApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
```

```
CButton *button;
public:
CButtonWindow();
afx_msg void HandleButton();
afx_msg void OnSize(UINT, int, int);
DECLARE_MESSAGE_MAP()
};
// A message handler function
void CButtonWindow::HandleButton()
{
MessageBeep(-1);
}
// A message handler function
void CButtonWindow::OnSize(UINT nType, int cx,
int cy)
{
CRect r;
GetClientRect(&r);
r.InflateRect(-20,-20);
button->MoveWindow(r);
}
// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
ON_BN_CLICKED(IDB_BUTTON, HandleButton)
ON_WM_SIZE()
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CButton Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
```

```
    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);

    // Create a button
    button = new CButton();
    button->Create("Push me",
        WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
        r,
        this,
        IDB_BUTTON);
    }
```

为了理解上面的代码，从窗口的消息映射开始。你会发现入口 ON_WM_SIZE。该入口表示消息映射是对来自 CButtonWindow 对象的变尺寸消息发生响应。变尺寸消息是当用户改变窗口的大小时产生的。该消息来自窗口本身，而不是作为 ON_COMMAND 消息由按钮向其父窗口发送的。这是因为窗口框架不是子窗口。

要注意的是消息映射中的 ON_WM_SIZE 入口没有参数。你在 MFC 文档中 CWnd 类，消息映射中的 ON_WM_SIZE 入口总是调用 OnSize 函数，并且该函数必须接收三个参数。OnSize 函数必须是消息映射所属类的成员函数，并且该函数必须用 afx_msg 来说明(正如上面在 CButtonWindow 的定义中所见到的一样)。

如果你查看 MFC 文档，就会发现 CWnd 中有近 100 名为“ On... ”的函数。CWnd::OnSize 是其中之一。所有这些函数都在消息映射中有形如 ON_WM_对应的标签。例如，ON_WM_SIZE 对应 OnSize。ON_WM_入口不接收任何参数，如 ON_BN_CLICKED 一样。参数是假设的并自动传递给相应的如 OnSize 的“ On... ”函数。

重复一遍，因为它很重要：OnSize 函数总是与消息映射中的 ON_WM_SIZE 入口想对应。你必须命名处理函数 OnSize，并且它必须接收三个参数。不同的函数的参数会有所不同。

上面的代码中在 OnSize 函数自身的内部，有三行代码修改了按钮在窗口中的尺寸。你可以在该函数中输入任何你想要的代码。

调用 GetClientRect 是为了恢复窗口用户区域的新尺寸。该矩形会被缩小，并调用按钮的 MoveWindow 函数。MoveWindow 是从 CWnd 继承来的，改变尺寸和移动子窗口是在一步完成的。

当你执行上面改变窗口大小的程序时，你就会发现按钮自己能正确地改变大小。在代码中，变尺寸事件他国消息映射中的 OnSize 函数而产生一调用，它调用 MoveWindow 函数来改变按钮的大小。

窗口消息

查看 MFC 文档，你可以看到主窗口处理的各种各样的 CWnd 消息。有些与我们上面介绍的类似。例如，ON_WM_MOVE 消息是当用户移动窗口时发送的消息，ON_WM_PAINT 消息是当窗口的任何部分需要重画时发出的。至今为止，我们的所有程序，重画工作都是自动完成的，因为是控制自己来负责其外观。如果你自己使用 GDI 命令来在用户区域中绘制，应用程序就应负责重画工作。因此 ON_WM_PAINT 就变得重要了。

还有一些发送给窗口的事件消息更深奥。例如，你可以使用 ON_WM_TIMER 消息与 SetTimer 函数来使接收预先设置的时间间隔。下面的代码给出了该过程。当你运行该代码时，程序会每隔 1 秒钟鸣笛一声。你可以用其它更有用的功能来代替鸣笛。

```
// button4.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
#define IDT_TIMER1 200
// Declare the application class
class CButtonApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
CButton *button;
public:
CButtonWindow();
afx_msg void HandleButton();
afx_msg void OnSize(UINT, int, int);
afx_msg void OnTimer(UINT);
DECLARE_MESSAGE_MAP()
};
// A message handler function
void CButtonWindow::HandleButton()
{
MessageBeep(-1);
}
// A message handler function
void CButtonWindow::OnSize(UINT nType, int cx,
int cy)
{
CRect r;
GetClientRect(&r);
r.InflateRect(-20,-20);
button->MoveWindow(r);
}
// A message handler function
void CButtonWindow::OnTimer(UINT id)
{
MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
```

```
ON_BN_CLICKED(IDB_BUTTON, HandleButton)
ON_WM_SIZE()
ON_WM_TIMER()
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
m_pMainWnd = new CButtonWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CButton Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));
// Set up the timer
SetTimer(IDT_TIMER1, 1000, NULL); // 1000 ms.
// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);
// Create a button
button = new CButton();
button->Create("Push me",
    WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
    r,
    this,
    IDB_BUTTON);
}
```

在上面的程序内部，我们建立了一个按钮，如前所示，改变尺寸的代码没有变动。在窗口的构造函数中，我们添加了 SetTimer 函数的调用。该函数接收三个参数：时钟的 ID(可以同时使用多个时钟，每次时钟关闭时都会把 ID 传递给所调用的函数)，时间以毫秒为单位。在这里，我们向函数传送了 NULL，以使窗口消息映射自己自动发送函数。在消息映射中，我们已经通知了 ON_WM_TIMER 消息，它会自动调用 OnTimer 函数来传递已经关闭了的时钟的 ID。

当程序运行时，它每隔 1 毫秒鸣笛一声。每次时钟的时间增量流逝，窗口都会发送消息给自己。消息映射选择消息给 OnTimer 函数，它鸣笛。你可以在此放置更有用的代码。

滚动条控制

Windows 用两种不同的方式来处理滚动条。一些控制，如编辑控制和列表控制，可以带有滚动条。在这种情况下，滚动条会被自动处理，不不要额外的代码来处理。

滚动条也可以作为单独的元件来使用。当这样使用时，滚动条就拥有独立的权力。你可以参见 MFC 参考手册中有关 CScrollBar 的有关章节。滚动条控制的建立与前面介绍的静态标签和按钮的一样。它有四个成员函数允许你设置和获取滚动条的位置和范围。

下面的代码演示了建立水平滚动条的过程和其消息映射：

```
// sb1.cpp
#include <afxwin.h>
#define IDM_SCROLLBAR 100
const int MAX_RANGE=100;
const int MIN_RANGE=0;
// Declare the application class
class CScrollBarApp : public CWinApp
{
public:
virtual BOOL InitInstance();
};
// Create an instance of the application class
CScrollBarApp ScrollBarApp;
// Declare the main window class
class CScrollBarWindow : public CFrameWnd
{
CScrollBar *sb;
public:
CScrollBarWindow();
afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
    CScrollBar* pScrollBar);
DECLARE_MESSAGE_MAP()
};
// The message handler function
void CScrollBarWindow::OnHScroll(UINT nSBCode,
UINT nPos, CScrollBar* pScrollBar)
{
MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CScrollBarWindow, CFrameWnd)
ON_WM_HSCROLL()
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CScrollBarApp::InitInstance()
{
```

```
m_pMainWnd = new CScrollBarWindow();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}
// The constructor for the window class
CScrollBarWindow::CScrollBarWindow()
{
CRect r;
// Create the window itself
Create(NULL,
    "CScrollBar Tests",
    WS_OVERLAPPEDWINDOW,
    CRect(0,0,200,200));

// Get the size of the client rectangle
GetClientRect(&r);
// Create a scroll bar
sb = new CScrollBar();
sb->Create(WS_CHILD|WS_VISIBLE|SBS_HORZ,
    CRect(10,10,r.Width()-10,30),
    this,
    IDM_SCROLLBAR);
sb->SetScrollRange(MIN_RANGE,MAX_RANGE,TRUE);
}
```

Windows 会区分水平和垂直滚动条，同时还支持 CScrollBar 中一称为尺寸盒的控制。尺寸盒是一个小方块。它处于水平和垂直滚动条的交叉处，呀鼠标拖动它会自动改变窗口的大小。在后面的代码中你看到如何用 Create 函数的 SBS_HORZ 样式来建立一水平滚动条。在建立了滚动条之后，马上用 SetScrollRange 中的 MIN_RANGE 和 MAX_RANGE 龙个常数给出了滚动条的范围 0~100(它们定义在程序的顶部)。

事件处理函数 OnHScroll 来自 CWnd 类。我们使用该函数是因为该代码建立了水平滚动条。对于垂直滚动条应使用 OnVScroll。在代码中，消息映射与滚动函数相联系，并使滚动条在用户操作时发出鸣笛声。当你运行该程序时，你可以单击箭头、拖动滚动条上的小方块等等。每次操作都会出现鸣笛声，但是滚动条上的小方块实际上不会移动，因为我们还没有把它与实际的代码相关联。

每次滚动条调用 OnHScroll 时，你的代码都要确定用户的操作。在 OnHScroll 函数内部，你可以检验传递给处理函数的第一参数，如下所示。如果你与上面的代码一起使用，滚动条的小方块就会移动到用户操作的位置处。

```
// The message handling function
void CScrollBarWindow::OnHScroll(UINT nSBCode,
UINT nPos, CScrollBar* pScrollBar)
{
int pos;
pos = sb->GetScrollPos();
```

```
switch ( nSBCode )
{
    case SB_LINEUP:
        pos -= 1;
        break;
    case SB_LINEDOWN:
        pos += 1;
        break;
    case SB_PAGEUP:
        pos -= 10;
        break;
    case SB_PAGEDOWN:
        pos += 10;
        break;
    case SB_TOP:
        pos = MIN_RANGE;
        break;
    case SB_BOTTOM:
        pos = MAX_RANGE;
        break;

    case SB_THUMBPOSITION:
        pos = nPos;
        break;
    default:
        return;
}
if ( pos < MIN_RANGE )
    pos = MIN_RANGE;
else if ( pos > MAX_RANGE )
    pos = MAX_RANGE;
sb->SetScrollPos( pos, TRUE );
}
```

SB_LINEUP 和 SB_LINEDOWN 的不同常数值在 CWnd::OnHScroll 函数文档中有介绍。上面的代码首先使用 GetScrollPos 函数来恢复滚动条的当前位置。然后使用开关语句来确定用户对滚动条的操作。SB_LINEUP 和 SB_LINEDOWN 常数值意味着垂直方向，但也可用于水平方向表示左右移动。SB_PAGEUP 和 SB_PAGEDOWN 是用在用户单击滚动条时。SB_TOP 和 SB_BOTTOM 用于当用户移动滚动条小方块到滚动条的顶部和底部。SB_THUMBPOSITION 用于当用户拖动小方块到指定位置时。代码会自动调整位置，然后确保它在设置其新位置时仍然在范围内。一旦设置了滚动条，小方块就会移动到适当的位置。

垂直滚动条的处理也是类似的，只是要用 OnVScroll 函数中的 SBS_VERT 样式。

理解消息映射

消息映射结构只能用于 MFC。掌握它和如何在你的代码中应用它是很重要的。

可能纯 C++使用者会对消息映射产生疑问：为什么 Microsoft 不用虚拟函数来替代消息映射？虚拟函数是 MFC 中处理消息映射的标准 C++方式,所以使用宏DECLARE_MESSAGE_MAP 和 BEGIN_MESSAGE_MAP 可能有些怪异。

MFC 使用消息映射来解决虚拟函数的基本问题。参见 MFC 帮助文件中的 CWnd 类。它包含 200 多个成员函数，所有的成员函数当不使用消息映射时都是虚拟的。现在来看一下所有 CWnd 类的子类。MFC 中大约有近 30 个类是以 CWnd 为基类的。这包括所有的可见控制如按钮、静态标签和列表。现在想象一下，MFC 使用虚拟函数，并且你建立一应用程序包含有 20 个控制。CWnd 中的 200 个虚拟函数中的每个都需要自己的虚拟函数表，并且一个控制的每个例程都应有一组 200 个虚拟函数与之关联。则程序可能就有近 4,000 个虚拟函数表在内存中，这对内存有限的机器来说是个大问题。因为其中的大部分是不用的。

消息映射复制了虚拟函数表的操作，但是它是基于需要的基础之上的。当你在消息映射中建立一个入口时，你是在对系统说，“ 当你看见一个特殊的消息时，请调用指定的函数 ”。只有这些函数实际上被重载到消息映射中，着就节省了内存和 CPU 的负担。

当你用 DECLARE_MESSAGE_MAP 和 BEGIN_MESSAGE_MAP 说明消息映射时，系统会通过你的消息映射选择所有的消息。如果消息映射处理了给定的消息，则你的函数会被调用，卸车也就停留在此。但是，如果你的消息映射中不包含某个消息的入口，则系统会把该消息发送第二个 BEGIN_MESSAGE_MAP 指定的类。那个类可能会也可能不会处理它，如此重复。最后，如果没有消息映射处理一给定的消息，该消息会到由一缺省的处理函数来处理。

结论

本讲中所介绍的所有消息映射处理概念可适用于 Windows NT 中所有的控制和窗口。在大部分情况下，你可以使用 ClassWizard 来安装消息映射的入口，它将在后面的有关 ClassWizard、AppWizard 和资源编辑器一文中介绍。