Visual C++/MFC 入门教程

闻怡洋

第一章 VC 入门

1.1 如何学好VC
1.2 理解Windows消息机制
1.3 利用Visual C++/MFC开发Windows程序的优势
1.4 利用MFC进行开发的通用方法介绍
1.5 MFC中常用类、宏、函数介绍

第二章 图形输出

2.1 和GUI有关的各种对象
2.2 在窗口中输出文字
2.3 使用点、刷子、笔进行绘图
2.4 在窗口中绘制设备相关位图、图标、设备无关位图
2.5 使用各种映射方式
2.6 多边形和剪贴区域

第三章 文档视结构

3.1 文档 视图 框架窗口间的关系和消息传送规律
3.2 接收用户输入
3.3 使用菜单
3.4 文档、视、框架之间相互作用
3.5 利用序列化进行文件读写
3.6 MFC中所提供的各种视类介绍

第四章 窗口控件

4.1 Button
4.2 Static Box
4.3 Edit Box
4.4 Scroll Bar
4.5 List Box/Check List Box
4.6 Combo Box/Combo Box Ex
4.7 Tree Ctrl
4.8 List Ctrl
4.9 Tab Ctrl
4.A Tool Bar
4.B Status Bar
4.C Dialog Bar

4.D 利用AppWizard创建并使用ToolBar StatusBar Dialog Bar
4.E General Window
4.F 关于WM_NOTIFY的使用方法

第五章 对话框

5.1 使用资源编辑器编辑对话框
5.2 创建有模式对话框
5.3 创建无模式对话框
5.4 在对话框中进行消息映射
5.5 在对话框中进行数据交换和数据检查
5.6 使用属性对话框
5.7 使用通用对话框
5.8 建立以对话框为基础的应用
5.9 使用对话框作为子窗口

第六章 网络通信开发

6.1 WinSock介绍
6.2 利用WinSock进行无连接的通信
6.3 利用WinSock建立有连接的通信

※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※
※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※※

第一章　VC 入门

1.1 如何学好VC

这个问题很多朋友都问过我，当然流汗是必须的，但同时如果按照某种思路进行有计划的学习就会起到更好的效果。万事开头难，为了帮助朋友们更快的掌握 VC 开发，下面我将自己的一点体会讲一下：

1、需要有好的 C/C++基础。正所谓“磨刀不误砍柴工”，最开始接触 VC 时不要急于开始 Windows 程序开发，而是应该进行一些字符界面程序的编写。这样做的目的主要是增加对语言的熟悉程度，同时也训练自己的思维和熟悉一些在编程中常犯的错误。更重要的是理解并能运用 C++的各种特性，这些在以后的开发中都会有很大的帮助，特别是利用 MFC 进行开发的朋友对 C++一定要能熟练运用。
2、理解 Windows 的消息机制，窗口句柄和其他 GUI 句柄的含义和用途。了解和 MFC 各个类功能相近的 API 函数。
3、一定要理解 MFC 中消息映射的作用。
4、训练自己在编写代码时不使用参考书而是使用 Help Online。
5、记住一些常用的消息名称和参数的意义。

6、学会看别人的代码。
7、多看书，少买书，买书前一定要慎重。
8、闲下来的时候就看参考书。
9、多来我的主页。^O^

后面几条是我个人的一点意见，你可以根据需要和自身的情况选用适用于自己的方法。此外我将一些我在选择参考书时的原则：

对于初学者：应该选择一些内容比较全面的书籍，并且书籍中的内容应该以合理的方式安排，在使用该书时可以达到循序渐进的效果，书中的代码要有详细的讲解。尽量买翻译的书，因为这些书一般都比较易懂，而且语言比较轻松。买书前一定要慎重如果买到不好用的书可能会对自己的学习积极性产生打击。

对于已经掌握了 VC 的朋友：这种程度的开发者应该加深自己对系统原理，技术要点的认识。需要选择一些对原理讲解的比较透彻的书籍，这样一来才会对新技术有更多的了解，最好书中对技术的应用有一定的阐述。尽量选择示范代码必较精简的书，可以节约银子。

此外最好涉猎一些辅助性的书籍

1.2 理解 Windows 消息机制

Windows 系统是一个消息驱动的 OS，什么是消息呢？我很难说得清楚，也很难下一个定义（谁在嘘我），我下面从不同的几个方面讲解一下，希望大家看了后有一点了解。

1、消息的组成：一个消息由一个消息名称（UINT），和两个参数（WPARAM，LPARAM）。当用户进行了输入或是窗口的状态发生改变时系统都会发送消息到某一个窗口。例如当菜单转中之后会有 WM_COMMAND 消息发送，WPARAM 的高字中（HIWORD(wParam)）是命令的 ID 号，对菜单来讲就是菜单 ID。当然用户也可以定义自己的消息名称，也可以利用自定义消息来发送通知和传送数据。

2、谁将收到消息：一个消息必须由一个窗口接收。在窗口的过程（WNDPROC）中可以对消息进行分析，对自己感兴趣的消息进行处理。例如你希望对菜单选择进行处理那么你可以定义对 WM_COMMAND 进行处理的代码，如果希望在窗口中进行图形输出就必须对 WM_PAINT 进行处理。

3、未处理的消息到那里去了：M$为窗口编写了默认的窗口过程，这个窗口过程将负责处理那些你不处理消息。正因为有了这个默认窗口过程我们才可以利用 Windows 的窗口进行开发而不必过多关注窗口各种消息的处理。例如窗口在被拖动时会有很多消息发送，而我们都可以不予理睬让系统自己去处理。

4、窗口句柄：说到消息就不能不说窗口句柄，系统通过窗口句柄来在整个系统中唯一标识一个窗口，发送一个消息时必须指定一个窗口句柄表明该消息由那个窗口接收。而每个窗口都会有自己的窗口过程，所以用户的输入就会被正确的处理。例如有两个窗口共用一个窗口

过程代码，你在窗口一上按下鼠标时消息就会通过窗口一的句柄被发送到窗口一而不是窗口二。

5、示例：下面有一段伪代码演示如何在窗口过程中处理消息

```
LONG yourWndProc(HWND hWnd,UINT uMessageType,WPARAM wP,LPARAM)
{
switch(uMessageType)
{//使用 SWITCH 语句将各种消息分开
case(WM_PAINT):
doYourWindow(...);//在窗口需要重新绘制时进行输出
break;
case(WM_LBUTTONDOWN):
doYourWork(...);//在鼠标左键被按下时进行处理
break;
default:
callDefaultWndProc(...);//对于其它情况就让系统自己处理
break;
}
}
```

接下来谈谈什么是消息机制：系统将会维护一个或多个消息队列，所有产生的消息都回被放入或是插入队列中。系统会在队列中取出每一条消息，根据消息的接收句柄而将该消息发送给拥有该窗口的程序的消息循环。每一个运行的程序都有自己的消息循环，在循环中得到属于自己的消息并根据接收窗口的句柄调用相应的窗口过程。而在没有消息时消息循环就将控制权交给系统所以 Windows 可以同时进行多个任务。下面的伪代码演示了消息循环的用法：

```
while(1)
{
id=getMessage(...);
if(id == quit)
break;
translateMessage(...);
}
```

当该程序没有消息通知时 getMessage 就不会返回，也就不会占用系统的 CPU 时间。 图示消息投递模式

在 16 位的系统中系统中只有一个消息队列，所以系统必须等待当前任务处理消息后才可以发送下一消息到相应程序，如果一个程序陷如死循环或是耗时操作时系统就会得不到控制权。这种多任务系统也就称为协同式的多任务系统。Windows3.X 就是这种系统。

而 32 位的系统中每一运行的程序都会有一个消息队列，所以系统可以在多个消息队列中转

换而不必等待当前程序完成消息处理就可以得到控制权。这种多任务系统就称为抢先式的多任务系统。Windows95/NT 就是这种系统

1.3 利用 Visual C++/MFC 开发 Windows 程序的优势

MFC 借助 C++的优势为 Windows 开发开辟了一片新天地，同时也借助 ApplicationWizzard 使开发者摆脱离了那些每次都必写基本代码，借助 ClassWizard 和消息映射使开发者摆脱了定义消息处理时那种混乱和冗长的代码段。更令人兴奋的是利用 C++的封装功能使开发者摆脱 Windows 中各种句柄的困扰，只需要面对 C++中的对象，这样一来使开发更接近开发语言而远离系统。（但我个人认为了解系统原理对开发很有帮助）

正因为 MFC 是建立在 C++的基础上，所以我强调 C/C++语言基础对开发的重要性。利用 C++的封装性开发者可以更容易理解和操作各种窗口对象；利用 C++的派生性开发者可以减少开发自定义窗口的时间和创造出可重用的代码；利用虚拟性可以在必要时更好的控制窗口的活动。而且 C++本身所具备的超越 C 语言的特性都可以使开发者编写出更易用，更灵活的代码。

在 MFC 中对消息的处理利用了消息映射的方法，该方法的基础是宏定义实现，通过宏定义将消息分派到不同的成员函数进行处理。下面简单讲述一下这种方法的实现方法。

代码如下

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
//}}AFX_MSG_MAP
ON_COMMAND(ID_FONT_DROPDOWN, DoNothing)
END_MESSAGE_MAP()
```

经过编译后，代码被替换为如下形式（这只是作讲解，实际情况比这复杂得多）：

```
//BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
CMainFrame::newWndProc(...)
{
switch(...)
{
//{{AFX_MSG_MAP(CMainFrame)
// ON_WM_CREATE()
case(WM_CREATE):
OnCreate(...);
break;
//}}AFX_MSG_MAP
// ON_COMMAND(ID_FONT_DROPDOWN, DoNothing)
case(WM_COMMAND):
if(HIWORD(wP)==ID_FONT_DROPDOWN)
{
DoNothing(...);
}
break;
```

//END_MESSAGE_MAP()
}
}
newWndProc 就是窗口过程只要是该类的实例生成的窗口都使用该窗口过程。

所以了解了 Windows 的消息机制在加上对消息映射的理解就很容易了解 MFC 开发的基本思路了

1.4 利用 MFC 进行开发的通用方法介绍
以下是我在最初学习 VC 时所常用的开发思路和方法，希望能对初学 VC 的朋友有所帮助和启发。
开发需要读写文件的应用程序并且有简单的输入和输出可以利用单文档视结构
开发注重交互的简单应用程序可以使用对话框为基础的窗口，如果文件读写简单这可利用 CFile 进行
开发注重交互并且文件读写复杂的的简单应用程序可以利用以 CFormView 为基础视的单文档视结构
利用对话框得到用户输入的数据，在等级提高后可使用就地输入
在对多文档要求不强烈时尽量避免多文档视结构，可以利用分隔条产生单文档多视结构
在要求在多个文档间传递数据时使用多文档视结构
学会利用子窗口，并在自定义的子窗口包含多个控件达到封装功能的目的
尽量避免使用多文档多视结构
不要使用多重继承并尽量减少一个类中封装过多的功能

1.5 MFC 中常用类、宏、函数介绍
常用类
CRect：用来表示矩形的类，拥有四个成员变量：top left bottom right。分别表是左上角和右下角的坐标。可以通过以下的方法构造。

- CRect(int l, int t, int r, int b); 指明四个坐标
- CRect(const RECT& srcRect); 由 RECT 结构构造
- CRect(LPCRECT lpSrcRect); 由 RECT 结构构造
- CRect(POINT point, SIZE size); 有左上角坐标和尺寸构造
- CRect(POINT topLeft, POINT bottomRight); 有两点坐标构造

下面介绍几个成员函数：

- int Width( ) const; 得到宽度
- int Height( ) const; 得到高度
- CSize Size( ) const; 得到尺寸
- CPoint& TopLeft( ); 得到左上角坐标
- CPoint& BottomRight( ); 得到右下角坐标
- CPoint CenterPoint( ) const; 得当中心坐标

此外矩形可以和点（CPoint）相加进行位移，和另一个矩形相加得到“并”操作后的矩形。

- CPoint：用来表示一个点的坐标，有两个成员变量：x y。 可以和另一个点相加。

● CString：用来表示可变长度的字符串。使用 CString 可不指明内存大小，CString 会根据需要自行分配。下面介绍几个成员函数：

● GetLength 得到字符串长度
● GetAt 得到指定位置处的字符
● operator + 相当于 strcat
● void Format(LPCTSTR lpszFormat, ...); 相当于 sprintf
● Find 查找指定字符，字符串
● Compare 比较
● CompareNoCase 不区分大小写比较
● MakeUpper 改为小写
● MakeLower 改为大写

CStringArray：用来表示可变长度的字符串数组。数组中每一个元素为 CString 对象的实例。下面介绍几个成员函数：

● Add 增加 CString
● RemoveAt 删除指定位置 CString 对象
● RemoveAll 删除数组中所有 CString 对象
● GetAt 得到指定位置的 CString 对象
● SetAt 修改指定位置的 CString 对象
● InsertAt 在某一位置插入 CString 对象

常用宏：RGB、 TRACE、ASSERT、VERIFY

常用函数
● CWindApp* AfxGetApp();
● HINSTANCE AfxGetInstanceHandle( )
● HINSTANCE AfxGetResourceHandle( )
● int AfxMessageBox(LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0);用于弹出一个消息框

第二章 图形输出

2.1 和GUI有关的各种对象
2.2 在窗口中输出文字
2.3 使用点、刷子、笔进行绘图
2.4 在窗口中绘制设备相关位图、图标、设备无关位图
2.5 使用各种映射方式
2.6 多边形和剪贴区域

2．1 和 GUI 有关的各种对象

在 Windows 中有各种 GUI 对象（不要和 C++对象混淆），当你在进行绘图就需要利用这些对象。而各种对象都拥有各种属性，下面分别讲述各种 GUI 对象和拥有的属性

字体对象 CFont 用于输出文字时选用不同风格和大小的字体。可选择的风格包括：是否为斜体，是否为粗体，字体名称，是否有下划线等。颜色和背景色不属于字体的属性。关于如何创建和使用字体在 2.2 在窗口中输出文字中会详细讲解

刷子 CBrush 对象决定填充区域时所采用的颜色或模板。对于一个固定色的刷子来讲它的属性为颜色，是否采用网格和网格的类型如水平的，垂直的，交叉的等。你也可以利用 8*8 的位图来创建一个自定义模板的刷子，在使用这种刷子填充时系统会利用位图逐步填充区域。关于如何创建和使用刷子在 2.3 使用刷子、笔进行绘图中会详细讲解

画笔 CPen 对象在画点和画线时有用。它的属性包括颜色，宽度，线的风格，如虚线，实线，点划线等。关于如何创建和使用画笔在 2.3 使用刷子、笔进行绘图中会详细讲解

位图 CBitmap 对象可以包含一幅图像，可以保存在资源中。关于如何使用位图在 2.4 在窗口中绘制设备相关位图、图标、设备无关位图中会详细讲解

还有一种特殊的 GUI 对象是多边形，利用多边形可以很好的限制作图区域或是改变窗口外型。关于如何创建和使用多边形在 2.6 多边形和剪贴区域中会详细讲解。

在 Windows 中使用 GUI 对象必须遵守一定的规则。

首先需要创建一个合法的对象，不同的对象创建方法不同。

然后需要将该 GUI 对象选入 DC 中，同时保存 DC 中原来的 GUI 对象。

如果选入一个非法的对象将会引起异常。在使用完后应该恢复原来的对象，这一点特别重要，如果保存一个临时对象在 DC 中，而在临时对象被销毁后可能引起异常。有一点必须注意，每一个对象在重新创建前必须销毁，下面的代码演示了这一种安全的使用方法。

```
OnDraw(CDC* pDC)
{
CPen pen1,pen2;
pen1.CreatePen(PS_SOLID,2,RGB(128,128,128));//创建对象
pen2.CreatePen(PS_SOLID,2,RGB(128,128,0));//创建对象
CPen* pPenOld=(CPen*)pDC->SelectObject(&pen1);//选择对象进 DC
drawWithPen1...
(CPen*)pDC->SelectObject(&pen2);//选择对象进 DC
drawWithPen2...
pen1.DeleteObject();//再次创建前先销毁
pen1.CreatePen(PS_SOLID,2,RGB(0,0,0));//再次创建对象
(CPen*)pDC->SelectObject(&pen1);//选择对象进 DC
drawWithPen1...
pDC->SelectObject(pOldPen);//恢复
}
```

此外系统中还拥有一些库存 GUI 对象，你可以利用

CDC::SelectStockObject(SelectStockObject(int nIndex)

选入这些对象，它们包括一些固定颜色的刷子，画笔和一些基本字体.

- BLACK_BRUSH Black brush.
- DKGRAY_BRUSH Dark gray brush.
- GRAY_BRUSH Gray brush.

- HOLLOW_BRUSH Hollow brush.
- LTGRAY_BRUSH Light gray brush.
- NULL_BRUSH Null brush.
- WHITE_BRUSH White brush.
- BLACK_PEN Black pen.
- NULL_PEN Null pen.
- WHITE_PEN White pen.
- ANSI_FIXED_FONT ANSI fixed system font.
- ANSI_VAR_FONT ANSI variable system font.
- DEVICE_DEFAULT_FONT Device-dependent font.
- OEM_FIXED_FONT OEM-dependent fixed font.
- SYSTEM_FONT The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later, the system font is proportional width; earlier versions of Windows use a fixed-width system font.
- SYSTEM_FIXED_FONT The fixed-width system font used in Windows prior to version 3.0. This object is available for compatibility with earlier versions of Windows.
- DEFAULT_PALETTE Default color palette. This palette consists of the 20 static colors in the system palette.

这些对象留在 DC 中是安全的，所以你可以利用选入库存对象来作为恢复 DC 中 GUI 对象。

大家可能都注意到了绘图时都需要一个 DC 对象，DC（Device Context 设备环境）对象是一个抽象的作图环境，可能是对应屏幕，也可能是对应打印机或其它。这个环境是设备无关的，所以你在对不同的设备输出时只需要使用不同的设备环境就行了，而作图方式可以完全不变。这也就是 Windows 耀眼的一点设备无关性。如同你将对一幅画使用照相机或复印机将会产生不同的输出，而不需要对画进行任何调整。DC 的使用会穿插在本章中进行介绍

2.2 在窗口中输出文字

在这里我假定读者已经利用 ApplicationWizard 生成了一个 SDI 界面的程序代码。接下来的你只需要在 CView 派生类的 OnDraw 成员函数中加入绘图代码就可以了。在这里我需要解释一下 OnDraw 函数的作用，OnDraw 函数会在窗口需要重绘时自动被调用，传入的参数 CDC* pDC 对应的就是 DC 环境。使用 OnDraw 的优点就在于在你使用打印功能的时候传入 OnDraw 的 DC 环境将会是打印机绘图环境，使用打印预览时传入的是一个称为 CPreviewDC 的绘图环境，所以你只需要一份代码就可以完成窗口/打印预览/打印机绘图三重功能。利用 Windows 的设备无关性和 M$为打印预览所编写的上千行代码你可以很容易的完成一个具有所见即所得的软件。

输出文字一般使用 CDC::BOOL TextOut(int x, int y, const CString& str)和 CDC::int DrawText(const CString& str, LPRECT lpRect, UINT nFormat)两个函数，对 TextOut 来讲只能输出单行的文字，而 DrawText 可以指定在一个矩形中输出单行或多行文字，并且可以规定对齐方式和使用何种风格。nFormat 可以是多种以下标记的组合（利用位或操作）以达到选择输出风格的目的。

pDC-> TextOut(100,100,”wqewr”);

- DT_BOTTOM 底部对齐 Specifies bottom-justified text. This value must be combined with

DT_SINGLELINE.

● DT_CALCRECT 计算指定文字时所需要矩形尺寸 Determines the width and height of the rectangle. If there are multiple lines of text, DrawText will use the width of the rectangle pointed to by lpRect and extend the base of the rectangle to bound the last line of text. If there is only one line of text, DrawText will modify the right side of the rectangle so that it bounds the last character in the line. In either case, DrawText returns the height of the formatted text, but does not draw the text.

● DT_CENTER 中部对齐 Centers text horizontally.

● DT_END_ELLIPSIS or DT_PATH_ELLIPSIS Replaces part of the given string with ellipses, if necessary, so that the result fits in the specified rectangle. The given string is not modified unless the DT_MODIFYSTRING flag is specified.

You can specify DT_END_ELLIPSIS to replace characters at the end of the string, or DT_PATH_ELLIPSIS to replace characters in the middle of the string. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much as possible of the text after the last backslash.

● DT_EXPANDTABS Expands tab characters. The default number of characters per tab is eight.

● DT_EXTERNALLEADING Includes the font 担 external leading in the line height. Normally, external leading is not included in the height of a line of text.

● DT_LEFT 左对齐 Aligns text flush-left.

● DT_MODIFYSTRING Modifies the given string to match the displayed text. This flag has no effect unless the DT_END_ELLIPSIS or DT_PATH_ELLIPSIS flag is specified.

Note Some uFormat flag combinations can cause the passed string to be modified. Using DT_MODIFYSTRING with either DT_END_ELLIPSIS or DT_PATH_ELLIPSIS may cause the string to be modified, causing an assertion in the CString override.

● DT_NOCLIP Draws without clipping. DrawText is somewhat faster when DT_NOCLIP is used.

● DT_NOPREFIX 禁止使用&前缀 Turns off processing of prefix characters. Normally, DrawText interprets the ampersand (&) mnemonic-prefix character as a directive to underscore the character that follows, and the two-ampersand (&&) mnemonic-prefix characters as a directive to print a single ampersand. By specifying DT_NOPREFIX, this processing is turned off.

● DT_PATH_ELLIPSIS

● DT_RIGHT 右对齐 Aligns text flush-right.

● DT_SINGLELINE 单行输出 Specifies single line only. Carriage returns and linefeeds do not break the line.

● DT_TABSTOP 设置 TAB 字符所占宽度 Sets tab stops. The high-order byte of nFormat is the number of characters for each tab. The default number of characters per tab is eight.

● DT_TOP 定部对齐 Specifies top-justified text (single line only).

● DT_VCENTER 中部对齐 Specifies vertically centered text (single line only).

● DT_WORDBREAK 每行只在单词间被折行 Specifies word-breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by lpRect. A carriage return 杭 inefeed sequence will also break the line.

在输出文字时如果希望改变文字的颜色，你可以利用 CDC::SetTextColor(COLORREF crColor)进行设置，如果你希望改变背景色就利用 CDC::SetBkColor(COLORREF crColor)，很多时候你可能需要透明的背景色你可以利用 CDC::SetBkMode(int nBkMode)设置，可接受的参数有

- OPAQUE Background is filled with the current background color before the text, hatched brush, or pen is drawn. This is the default background mode.
- TRANSPARENT Background is not changed before drawing.

接下来讲讲如何创建字体，你可以创建的字体有两种：库存字体 CDC::CreateStockObject(int nIndex)和自定义字体。
在创建非库存字体时需要填充一个 LOGFONT 结构并使用 CFont::CreateFontIndirect(const LOGFONT* lpLogFont)（可以参考文章在同一系统中显示 GB 字符和 BIG5 字符），或使用 CFont::CreateFont(int nHeight, int nWidth, int nEscapement, int nOrientation, int nWeight, BYTE bItalic, BYTE bUnderline, BYTE cStrikeOut, BYTE nCharSet, BYTE nOutPrecision, BYTE nClipPrecision, BYTE nQuality, BYTE nPitchAndFamily, LPCTSTR lpszFacename)其中的参数和 LOGFONT 中的分量有一定的对应关系。下面分别讲解参数的意义：

nHeight 字体高度（逻辑单位）等于零为缺省高度，否则取绝对值并和可用的字体高度进行匹配。
nWidth 宽度（逻辑单位）如果为零则使用可用的横纵比进行匹配。
nEscapement 出口矢量与 X 轴间的角度
nOrientation 字体基线与 X 轴间的角度
nWeight 字体粗细，可取以下值
Constant Value
FW_DONTCARE 0
FW_THIN 100
FW_EXTRALIGHT 200
FW_ULTRALIGHT 200
FW_LIGHT 300
FW_NORMAL 400
FW_REGULAR 400
FW_MEDIUM 500
FW_SEMIBOLD 600
FW_DEMIBOLD 600
FW_BOLD 700
FW_EXTRABOLD 800
FW_ULTRABOLD 800
FW_BLACK 900
FW_HEAVY 900

bItalic 是否为斜体

bUnderline 是否有下划线
cStrikeOut 是否带删除线
nCharSet 指定字符集合，可取以下值
Constant Value
ANSI_CHARSET 0
DEFAULT_CHARSET 1
SYMBOL_CHARSET 2
SHIFTJIS_CHARSET 128
OEM_CHARSET 255

nOutPrecision 输出精度
OUT_CHARACTER_PRECIS OUT_STRING_PRECIS
OUT_DEFAULT_PRECIS OUT_STROKE_PRECIS
OUT_DEVICE_PRECIS OUT_TT_PRECIS
OUT_RASTER_PRECIS

nClipPrecision 剪辑精度，可取以下值
CLIP_CHARACTER_PRECIS CLIP_MASK
CLIP_DEFAULT_PRECIS CLIP_STROKE_PRECIS
CLIP_ENCAPSULATE CLIP_TT_ALWAYS
CLIP_LH_ANGLES

nQuality 输出质量，可取以下值
- DEFAULT_QUALITY Appearance of the font does not matter.
- DRAFT_QUALITY Appearance of the font is less important than when PROOF_QUALITY is used. For GDI raster fonts, scaling is enabled. Bold, italic, underline, and strikeout fonts are synthesized if necessary.
- PROOF_QUALITY Character quality of the font is more important than exact matching of the logical-font attributes. For GDI raster fonts, scaling is disabled and the font closest in size is chosen. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

nPitchAndFamily 字体间的间距
lpszFacename 指定字体名称，为了得到系统所拥有的字体可以利用 EmunFontFamiliesEx。（可以参考文章在同一系统中显示 GB 字符和 BIG5 字符）

此外可以利用 CFontDialog 来得到用户选择的字体的 LOGFONT 数据。

最后我讲一下文本坐标的计算，利用 CDC::GetTextExtent(const CString& str)可以得到字符串的在输出时所占用的宽度和高度，这样就可以在手工输出多行文字时使用正确的行距。另外如果需要更精确的对字体高度和宽度进行计算就需要使用 CDC::GetTextMetrics(LPTEXTMETRIC lpMetrics) 该函数将会填充 TEXTMETRIC 结构，该结构中的分量可以非常精确的描述字体的各种属性。

2.3 使用点、刷子、笔进行绘图

在 Windows 中画点的方法很简单，只需要调用 COLORREF CDC::SetPixel(int x, int y, COLORREF crColor)就可以在指定点画上指定颜色，同时返回原来的颜色。COLORREF CDC::GetPixel(int x, int y)可以得到指定点的颜色。在 Windows 中应该少使用画点的函数，因为这样做的执行效率比较低。

刷子和画笔在 Windows 作图中是使用最多的 GUI 对象，本节在讲解刷子和画笔使用方法的同时也讲述一写基本作图函数。

在画点或画线时系统使用当前 DC 中的画笔，所以在创建画笔后必须将其选入 DC 才会在绘图时产生效果。画笔可以通过 CPen 对象来产生，通过调用 CPen::CreatePen(int nPenStyle, int nWidth, COLORREF crColor)来创建。其中 nPenStyle 指名画笔的风格，可取如下值：

- PS_SOLID 实线 Creates a solid pen.
- PS_DASH 虚线，宽度必须为一 Creates a dashed pen. Valid only when the pen width is 1 or less, in device units.
- PS_DOT 点线，宽度必须为一 Creates a dotted pen. Valid only when the pen width is 1 or less, in device units.
- PS_DASHDOT 点划线，宽度必须为一 Creates a pen with alternating dashes and dots. Valid only when the pen width is 1 or less, in device units.
- PS_DASHDOTDOT 双点划线，宽度必须为一 Creates a pen with alternating dashes and double dots. Valid only when the pen width is 1 or less, in device units.
- PS_NULL 空线，使用时什么也不会产生 Creates a null pen.
- PS_ENDCAP_ROUND 结束处为圆形 End caps are round.
- PS_ENDCAP_SQUARE 结束处为方形 End caps are square.

nWidth 和 crColor 为线的宽度和颜色。

刷子是在画封闭曲线时用来填充的颜色，例如当你画圆形或方形时系统会用当前的刷子对内部进行填充。刷子可利用 CBrush 对象产生。通过以下几种函数创建刷子：

- BOOL CreateSolidBrush(COLORREF crColor); 创建一种固定颜色的刷子
- BOOL CreateHatchBrush(int nIndex, COLORREF crColor); 创建指定颜色和网格的刷子，nIndex 可取以下值：
- HS_BDIAGONAL Downward hatch (left to right) at 45 degrees
- HS_CROSS Horizontal and vertical crosshatch
- HS_DIAGCROSS Crosshatch at 45 degrees
- HS_FDIAGONAL Upward hatch (left to right) at 45 degrees
- HS_HORIZONTAL Horizontal hatch
- HS_VERTICAL Vertical hatch
- BOOL CreatePatternBrush(CBitmap* pBitmap); 创建以 8*8 位图为模板的刷子

在选择了画笔和刷子后就可以利用 Windows 的作图函数进行作图了，基本的画线函数有以下几种

- CDC::MoveTo(int x, int y); 改变当前点的位置
- CDC::LineTo(int x, int y); 画一条由当前点到参数指定点的线
- CDC::BOOL Arc(LPCRECT lpRect, POINT ptStart, POINT ptEnd); 画弧线
- CDC::BOOL Polyline(LPPOINT lpPoints, int nCount); 将多条线依次序连接

基本的作图函数有以下几种：

- CDC::BOOL Rectangle(LPCRECT lpRect); 矩形
- CDC::RoundRect(LPCRECT lpRect, POINT point); 圆角矩形
- CDC::Draw3dRect(int x, int y, int cx, int cy, COLORREF clrTopLeft, COLORREF clrBottomRight); 3D 边框
- CDC::Chord(LPCRECT lpRect, POINT ptStart, POINT ptEnd); 扇形
- CDC::Ellipse(LPCRECT lpRect); 椭圆形
- CDC::Pie(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
- CDC::Polygon(LPPOINT lpPoints, int nCount); 多边形

对于矩形，圆形或类似的封闭曲线，系统会使用画笔绘制边缘，使用刷子填充内部。如果你不希望填充或是画出边缘，你可以选入空刷子（NULL_PEN）或是（NULL_BRUSH）空笔。

下面的代码创建一条两象素宽的实线并选入 DC。并进行简单的作图：

```
{
...
CPen pen;
pen.CreatePen(PS_SOLID,2,RGB(128,128,128));
CPen* pOldPen=(CPen*)dc.SelectObject(&pen);
dc.SelectStockObject(NULL_BRUSH);//选入空刷子
dc.Rectangle(CRect(0,0,20,20));//画矩形
...
}
```

2.4 在窗口中绘制设备相关位图、图标、设备无关位图

在 Windows 中可以将预先准备好的图像复制到显示区域中，这种内存拷贝执行起来是非常快的。在 Windows 中提供了两种使用图形拷贝的方法：通过设备相关位图（DDB）和设备无关位图（DIB）。

DDB 可以用 MFC 中的 CBitmap 来表示，而 DDB 一般是存储在资源文件中，在加载时只需要通过资源 ID 号就可以将图形装入。BOOL CBitmap::LoadBitmap(UINT nIDResource)可以装入指定 DDB，但是在绘制时必须借助另一个和当前绘图 DC 兼容的内存 DC 来进行。通过 CDC::BitBlt(int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, DWORD dwRop)绘制图形，同时指定光栅操作的类型。BitBlt 可以将源 DC 中位图复制到目的 DC 中，其中前四个参数为目的区域的坐标，接下来是源 DC 指针，然后是源 DC 中的起始坐标，由于 BitBlt 为等比例复制，所以不需要再次指定长宽，（StretchBlt 可以进行缩放）最后一个参数为光栅操作的类型，可取以下值：

- BLACKNESS 输出区域为黑色 Turns all output black.

● DSTINVERT 反色输出区域 Inverts the destination bitmap.
● MERGECOPY 在源和目的间使用 AND 操作 Combines the pattern and the source bitmap using the Boolean AND operator.
● MERGEPAINT 在反色后的目的和源间使用 OR 操作 Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
● NOTSRCCOPY 将反色后的源拷贝到目的区 Copies the inverted source bitmap to the destination.
● PATINVERT 源和目的间进行 XOR 操作 Combines the destination bitmap with the pattern using the Boolean XOR operator.
● SRCAND 源和目的间进行 AND 操作 Combines pixels of the destination and source bitmaps using the Boolean AND operator.
● SRCCOPY 复制源到目的区 Copies the source bitmap to the destination bitmap.
● SRCINVERT 源和目的间进行 XOR 操作 Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
● SRCPAINT 源和目的间进行 OR 操作 Combines pixels of the destination and source bitmaps using the Boolean OR operator.
● WHITENESS 输出区域为白色 Turns all output white.
下面用代码演示这种方法：

```
CYourView::OnDraw(CDC* pDC)
{
CDC memDC;//定义一个兼容 DC
memDC.CreateCompatibleDC(pDC);//创建 DC
CBitmap bmpDraw;
bmpDraw.LoadBitmap(ID_BMP);//装入 DDB
CBitmap* pbmpOld=memDC.SelectObject(&bmpDraw);//保存原有 DDB，并选入新 DDB 入 DC
pDC->BitBlt(0,0,20,20,&memDC,0,0,SRCCOPY);// 将源 DC 中 (0,0,20,20) 复制到目的 DC(0,0,20,20)
pDC->BitBlt(20,20,40,40,&memDC,0,0,SRCAND);// 将源 DC 中 (0,0,20,20) 和目的 DC(20,20,40,40)中区域进行 AND 操作
memDC.SelectObject(pbmpOld);//选入原 DDB
}
```

（图标并不是一个 GDI 对象，所以不需要选入 DC）在 MFC 中没有一个专门的图标类，因为图标的操作比较简单，使用 HICON CWinApp::LoadIcon(UINT nIDResource)或是 HICON CWinApp::LoadStandardIcon(LPCTSTR lpszIconName) 装入后就可以利用 BOOL CDC::DrawIcon(int x, int y, HICON hIcon)绘制。由于在图标中可以指定透明区域，所以在某些需要使用非规则图形而且面积不大的时候使用图标会比较简单。下面给出简单的代码：

```
OnDraw(CDC* pDC)
{
HICON hIcon1=AfxGetApp()->LoadIcon(IDI_I1);
HICON hIcon2=AfxGetApp()->LoadIcon(IDI_I2);
```

```
pDC->DrawIcon(0,0,hIcon1);
pDC->DrawIcon(0,40,hIcon2);
DestroyIcon(hIcon1);
DestroyIcon(hIcon2);
}
```

同样在 MFC 也没有提供一个 DIB 的类，所以在使用 DIB 位图时我们需要自己读取位图文件中的头信息，并读入数据，并利用 API 函数 StretchDIBits 绘制。位图文件以 BITMAPFILEHEADER 结构开始，然后是 BITMAPINFOHEADER 结构和调色版信息和数据，其实位图格式是图形格式中最简单的一种，而且也是 Windows 可以理解的一种。我不详细讲解 DIB 位图的结构，提供一个 CDib 类供大家使用，这个类包含了基本的功能如：Load,Save,Draw。DownLoad CDib 4K

2.5 使用各种映射方式

所谓的映射方式简单点讲就是坐标的安排方式，系统默认的映射方式为 MM_TEXT 即 X 坐标向右增加，Y 坐标向下增加，(0,0)在屏幕左上方，DC 中的每一点就是屏幕上的一个象素。也许你会认为这种方式下是最好理解的，但是一个点和象素对应的关系在屏幕上看来是正常的，但到了打印机上就会很不正常。因为我们作图是以点为单位并且打印机的分辨率远远比显示器高（800DPI 800 点每英寸）所以在打印机上图形看起来就会很小。这样就需要为打印另做一套代码而加大了工作量。如果每个点对应 0.1 毫米那么在屏幕上的图形就会和打印出来的图形一样大小。

通过 int CDC::SetMapMode(int nMapMode)可以指定映射方式，可用的有以下几种：

● MM_HIENGLISH 每点对应 0.001 英寸 Each logical unit is converted to 0.001 inch. Positive x is to the right; positive y is up.

● MM_HIMETRIC 每点对应 0.001 毫米 Each logical unit is converted to 0.01 millimeter. Positive x is to the right; positive y is up.

● MM_LOENGLISH 每点对应 0.01 英寸 Each logical unit is converted to 0.01 inch. Positive x is to the right; positive y is up.

● MM_LOMETRIC 每点对应 0.001 毫米 Each logical unit is converted to 0.1 millimeter. Positive x is to the right; positive y is up.

● MM_TEXT 象素对应 Each logical unit is converted to 1 device pixel. Positive x is to the right; positive y is down.

以上几种映射默认的原点在屏幕左上方。除 MM_TEXT 外都为 X 坐标向右增加，Y 坐标向

上增加，和自然坐标是一致的。所以在作图是要注意什么时候应该使用负坐标。而且以上的映射都是 X-Y 等比例的，即相同的长度在 X，Y 轴上显示的长度都是相同的。观看不同映射效果图 DownLoad Sample

另外的一种映射方式为 MM_ANISOTROPIC，这种方式可以规定不同的长宽比例。在设置这中映射方式后必须调用 CSize CDC::SetWindowExt(SIZE size)和 CSize CDC::SetViewportExt(SIZE size)来设定长宽比例。系统会根据两次设定的长宽的比值来确定长宽比例。下面给出一段代码比较映射前后的长宽比例：

```
OnDraw(CDC* pDC)
{
CRect rcC1(200,0,400,200);
pDC->FillSolidRect(rcC1,RGB(0,0,255));
pDC->SetMapMode(MM_ANISOTROPIC );
CSize sizeO;
sizeO=pDC->SetWindowExt(5,5);
TRACE("winExt %d %d\n",sizeO.cx,sizeO.cy);
sizeO=pDC->SetViewportExt(5,10);
TRACE("ViewExt %d %d\n",sizeO.cx,sizeO.cy);
CRect rcC(0,0,200,200);
pDC->FillSolidRect(rcC,RGB(0,128,0));
}
```

上面代码在映射后画出的图形将是一个长方形。观看效果图 DownLoad Sample

最后讲讲视原点（viewport origin）,你可以通过调用 CPoint CDC::SetViewportOrg(POINT point)重新设置原点的位置，这就相对于对坐标进行了位移。例如你将原点设置在(20,20)那么原来的(0,0)就变成了(-20,-20)。

2.6 多边形和剪贴区域

多边形也是一个 GDI 对象，同样遵守其他 GDI 对象的规则，只是通常都不将其选入 DC 中。在 MFC 中多边形有 CRgn 表示。多边形用来表示一个不同与矩形的区域，和矩形具有相似的操作。如：检测某点是否在内部，并操作等。此外还得到一个包含此多边形的最小矩形。下面介绍一下多边形类的成员函数：

- CreateRectRgn 由矩形创建一个多边形
- CreateEllipticRgn 由椭圆创建一个多边形
- CreatePolygonRgn 创建一个有多个点围成的多边形
- PtInRegion 某点是否在内部
- CombineRgn 两个多边形相并
- EqualRgn 两个多边形是否相等

在本节中讲演多边形的意义在于重新在窗口中作图时提高效率。因为引发窗口重绘的原因是某个区域失效，而失效的区域用多边形来表示。假设窗口大小为 500*400 当上方的另一个窗口从(0,0,10,10)移动到(20,20,30,30)这时(0,0,10,10)区域就失效了，而你只需要重绘这部分区域而不是所有区域，这样你程序的执行效率就会提高。

通过调用 API 函数 int GetClipRgn(HDC hdc, HRGN hrgn)就可以得到失效区域，但是一般用不着那么精确而只需得到包含该区域的最小矩形就可以了，所以可以利用 int CDC::GetClipBox(LPRECT lpRect)完成这一功能。

第三章 文档视结构

3.1 文档 视图 框架窗口间的关系和消息传送规律

在 MFC 中 M$引入了文档-视结构的概念，文档相当于数据容器，视相当于查看数据的窗口或是和数据发生交互的窗口。（这一结构在 MFC 中的 OLE，ODBC 开发时又得到更多的拓展）因此一个完整的应用一般由四个类组成：CWinApp 应用类，CFrameWnd 窗口框架类，CDocument 文档类，CView 视类。（VC6 中支持创建不带文档-视的应用）

在程序运行时 CWinApp 将创建一个 CFrameWnd 框架窗口实例，而框架窗口将创建文档模板，然后有文档模板创建文档实例和视实例，并将两者关联。一般来讲我们只需对文档和视进行操作，框架的各种行为已经被 MFC 安排好了而不需人为干预，这也是 M$设计文档-视结构的本意，让我们将注意力放在完成任务上而从界面编写中解放出来。

在应用中一个视对应一个文档，但一个文档可以包含多个视。一个应用中只用一个框架窗口，对多文档界面来讲可能有多个 MDI 子窗口。每一个视都是一个子窗口，在单文档界面中父窗口即是框架窗口，在多文档界面中父窗口为 MDI 子窗口。一个多文档应用中可以包含多个文档模板，一个模板定义了一个文档和一个或多个视之间的对应关系。同一个文档可以属于多个模板，但一个模板中只允许定义一个文档。同样一个视也可以属于多个文档模板。（不知道我说清楚没有）

接下来看看如何在程序中得到各种对象的指针：

- 全局函数 AfxGetApp 可以得到 CWinApp 应用类指针
- AfxGetApp()->m_pMainWnd 为框架窗口指针
- 在框架窗口中：CFrameWnd::GetActiveDocument 得到当前活动文档指针
- 在框架窗口中：CFrameWnd::GetActiveView 得到当前活动视指针
- 在视中：CView::GetDocument 得到对应的文档指针
- 在文档中：CDocument::GetFirstViewPosition，CDocument::GetNextView 用来遍历所有和文档关联的视。
- 在文档中：CDocument::GetDocTemplate 得到文档模板指针
- 在多文档界面中：CMDIFrameWnd::MDIGetActive 得到当前活动的 MDI 子窗口

一般来讲用户输入消息（如菜单选择，鼠标，键盘等）会先发往视，如果视未处理则会发往框架窗口。所以定义消息映射时定义在视中就可以了，如果一个应用同时拥有多个视而当前

活动视没有对消息进行处理则消息会发往框架窗口

3.2 接收用户输入

在视中接收鼠标输入：

鼠标消息是我们常需要处理的消息，消息分为：鼠标移动，按钮按下/松开，双击。利用 ClassWizard 可以轻松的添加这几种消息映射，下面分别讲解每种消息的处理。

WM_MOUSEMOVE 对应的函数为 OnMouseMove(UINT nFlags, CPoint point)，nFlags 表明了当前一些按键的消息，你可以通过“位与”操作进行检测。

- MK_CONTROL Ctrl 键是否被按下 Set if the CTRL key is down.

- MK_LBUTTON 鼠标左键是否被按下 Set if the left mouse button is down.

- MK_MBUTTON 鼠标中间键是否被按下 Set if the middle mouse button is down.

- MK_RBUTTON 鼠标右键是否被按下 Set if the right mouse button is down.

- MK_SHIFT Shift 键是否被按下 Set if the SHIFT key is down.

point 表示当前鼠标的设备坐标，坐标原点对应视左上角。

WM_LBUTTONDOWN/WM_RBUTTONDOWN（鼠标左/右键按下）对应的函数为 OnLButtonDown/OnRButtonDown(UINT nFlags, CPoint point)参数意义和 OnMouseMove 相同。

WM_LBUTTONUP/WM_RBUTTONUP（鼠标左/右键松开）对应的函数为 OnLButtonUp/OnRButtonUp(UINT nFlags, CPoint point)参数意义和 OnMouseMove 相同。

WM_LBUTTONDBLCLK/WM_RBUTTONDBLCLK（鼠标左/右键双击）对应的函数为 OnLButtonDblClk/OnRButtonDblClk(UINT nFlags, CPoint point)参数意义和 OnMouseMove 相同。

下面我用一段伪代码来讲解一下这些消息的用法：

代码的作用是用鼠标拉出一个矩形
global BOOL fDowned;//是否在拉动
global CPoint ptDown;//按下位置
global CPoint ptUp;//松开位置

```
OnLButtonDown(UINT nFlags, CPoint point)
{
fDowned=TRUE;
ptUp=ptDown=point;
DrawRect();
...
}

OnMouseMove(UINT nFlags, CPoint point)
{
if(fDowned)
{
DrawRect();//恢复上次所画的矩形
ptUp=point;
DrawRect();//画新矩形
}
}
OnLButtonUp(UINT nFlags, CPoint point)
{
if(fDowned)
{
DrawRect();//恢复上次所画的矩形
ptUp=point;
DrawRect();//画新矩形
fDowned=FALSE;
}
}

DrawRect()
{//以反色屏幕的方法画出 ptDown,ptUp 标记的矩形
CClientDC dc(this);
MakeRect(ptDown,ptUp);
SetROP(NOT);
Rect();
}
```

坐标间转换：在以上的函数中 point 参数对应的都是窗口的设备坐标，我们应该将设备坐标和逻辑坐标相区别，在图 32_g1 由于窗口使用了滚动条，所以传入的设备坐标是对应于当前窗口左上角的坐标，没有考虑是否滚动，而逻辑坐标必须考虑滚动后对应的坐标，所以我以黄线虚拟的表达一个逻辑坐标的区域。可以看得出同一点在滚动后的坐标值是不同的，这一规则同样适用于改变了映射方式的窗口，假设你将映射方式设置为每点为 0.01 毫米，那么设备坐标所对应的逻辑坐标也需要重新计算。进行这种转换需要写一段代码，所幸的是系统

提供了进行转换的功能 DC 的 DPtoLP，LPtoDP，下面给出代码完成由设备坐标到逻辑坐标的转换。

```
CPoint CYourView::FromDP(CPoint point)
{
CClientDC dc(this);
CPoint ptRet=point;
dc.PrepareDC();//必须先准备 DC，这在使用滚动时让 DC 重新计算坐标

//如果你作图设置了不同的映射方式，则在下面需要设置
dc.SetMapMode(...)
//
dc.DPtoLP(&ptRet);//DP->LP 进行转换
return ptRet;
}
```

在图 32_g1 中以蓝线标记的是屏幕区域，红线标记的客户区域。利用 ScreenToClient，ClientToScreen 可以将坐标在这两个区域间转换。

在视中接收键盘输入：

键盘消息有三个：键盘被按下/松开，输入字符。其中输入字符相当于直接得到用户输入的字符这在不需要处理按键细节时使用，而键盘被按下/松开在按键状态改变时发送。

WM_CHAR 对应的函数为 OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)，其中 nChar 为被按下的字符，nRepCnt 表明在长时间为松开时相当于的按键次数，nFlags 中的不同位代表不同的含义，在这里一般不使用。

WM_KEYDOWN/WM_KEYUP 所对应的函数为 OnKeyDown/OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)nChar 代表按键的虚拟码值，如 VK_ALT 为 ALT 键，VK_CONTROL 为 Ctrl 键。nFlags 各位的含义如下：

Value Description

0?5 Scan code (OEM-dependent value).

8 Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key).

9?0 Not used.

11?2 Used internally by Windows.

13 Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).

14 Previous key state (1 if the key is down before the call, 0 if the key is up).

15 Transition state (1 if the key is being released, 0 if the key is being pressed).

3.3 使用菜单

利用菜单接受用户命令是一中很简单的交互方法，同时也是一种很有效的方法。通常菜单作为一中资源存储在文件中，因此我们可以在设计时就利用资源编辑器设计好一个菜单。关于使用 VC 设计菜单我就不再多讲了，但你在编写菜单时应该尽量在属性对话框的底部提示（Prompt）处输入文字，这虽然不是必要的，但 MFC 在有状态栏和工具条的情况下会使用该文字，文字的格式为“状态栏出说明\n 工具条提示”。图 33_g1

我们要面临的任务是如何知道用户何时选择了菜单，他选的是什么菜单项。当用户选择了一个有效的菜单项时系统会向应用发送一个 WM_COMMAND 消息，在消息的参数中表明来源。在 MFC 中我们只需要进行一次映射，将某一菜单 ID 映射到一处理函数，图 33_g2。在这里我们在 CView 的派生类中处理菜单消息，同时我对同一 ID 设置两个消息映射，接下来将这两种映射的作用。

ON_COMMAND 映射的作用为在用户选择该菜单时调用指定的处理函数。如：ON_COMMAND(IDM_COMMAND1, OnCommand1)会使菜单被选择时调用 OnCommand1 成员函数。

ON_UPDATE_COMMAND_UI(IDM_COMMAND1, OnUpdateCommand1) 映射的作用是在菜单被显示时通过调用指定的函数来进行确定其状态。在这个处理函数中你可以设置菜单的允许/禁止状态，其显示字符串是什么，是否在前面打钩。函数的参数为 CCmdUI* pCmdUI，CCmdUI 是 MFC 专门为更新命令提供的一个类，你可以调用

- Enable 设置允许/禁止状态
- SetCheck 设置是否在前面打钩
- SetText 设置文字

下面我讲解一个例子：我在 CView 派生类中有一个变量 m_fSelected，并且在视中处理两个菜单的消息，当 IDM_COMMAND1 被选时，对 m_fSelected 进行逻辑非操作，当 IDM_COMMAND2 被选中时出一提示；同时 IDM_COMMAND1 根据 m_fSelected 决定菜单显示的文字和是否在前面打上检查符号，IDM_COMMAND2 根据 m_fSelected 的值决定菜单的允许/禁止状态。下面是代码和说明：下载示例代码 17K

```
void CMenuDView::OnCommand1()
{
m_fSelected=!m_fSelected;
TRACE("command1 selected\n");
}

void CMenuDView::OnUpdateCommand1(CCmdUI* pCmdUI)
{
pCmdUI->SetCheck(m_fSelected);//决定检查状态
```

```
pCmdUI->SetText(m_fSelected?"当前被选中":"当前未被选中");//决定所显示的文字
}


void CMenuDView::OnUpdateCommand2(CCmdUI* pCmdUI)
{//决定是否为允许
pCmdUI->Enable(m_fSelected);
}

void CMenuDView::OnCommand2()
{//选中时给出提示
AfxMessageBox("你选了 command2");
}
```

接下来再讲一些通过代码操纵菜单的方法，在 MFC 中有一个类 CMenu 用来处理和菜单有关的功能。在生成一个 CMenu 对象时你需要从资源中装如菜单，通过调用 BOOL CMenu::LoadMenu(UINT nIDResource)进行装入，然后你就可以对菜单进行动态的修改，所涉及到的函数有：

- CMenu* GetSubMenu(int nPos) 一位置得到子菜单的指针，因为一个 CMenu 对象只能表示一个弹出菜单，如果菜单中的某一项也为弹出菜单，就需要通过该函数获取指针。
- BOOL AppendMenu(UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL) 在末尾添加一项，nFlag 为 MF_SEPARATOR 表示增加一个分隔条，这样其他两个参数将会被忽略；为 MF_STRING 表示添加一个菜单项 uIDNewItem 为该菜单的 ID 命令值；为 MF_POPUP 表示添加一个弹出菜单项，这时 uIDNewItem 为另一菜单的句柄 HMENU。lpszNewItem 为菜单文字说明。
- BOOL InsertMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL)用于在指定位置插入一菜单，位置由变量 nPosition 指明。如果 nFlags 包含 MF_BYPOSITION 则表明插入在 nPosition 位置，如果包含 MF_BYCOMMAND 表示插入在命令 ID 为 nPosition 的菜单处。
- BOOL ModifyMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL)用于修改某一位置的菜单，如果 nFlags 包含 MF_BYPOSITION 则表明修改 nPosition 位置的菜单，如果包含 MF_BYCOMMAND 表示修改命令 ID 为 nPosition 处的菜单。
- BOOL RemoveMenu(UINT nPosition, UINT nFlags)用于删除某一位置的菜单。如果 nFlags 包含 MF_BYPOSITION 则表明删除 nPosition 位置的菜单，如果包含 MF_BYCOMMAND 表示删除命令 ID 为 nPosition 处的菜单。
- BOOL AppendMenu(UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp) 和 BOOL InsertMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp)可以添加一位图菜单，但这样的菜单在选中时只是反色显示，并不美观。（关于使用自绘 OwnerDraw 菜单请参考我翻译的一篇文章自绘菜单类）

视图中是没有菜单的，在框架窗口中才有，所以只有用

AfxGetApp()->m_pMainWnd->GetMenu()才能得到应用的菜单指针。

最后我讲一下如何在程序中弹出一个菜单，你必须先装入一个菜单资源，你必需得到一个弹出菜单的指针然后调用 BOOL TrackPopupMenu(UINT nFlags, int x, int y, CWnd* pWnd, LPCRECT lpRect = NULL)弹出菜单，你需要指定(x,y)为菜单弹出的位置，pWnd 为接收命令消息的窗口指针。下面有一段代码说明方法，下载示例代码 17K。当然为了处理消息你应该在 pWnd 指明的窗口中对菜单命令消息进行映射。

```
CMenu menu;
menu.LoadMenu(IDR_POPUP);
CMenu* pM=menu.GetSubMenu(0);
CPoint pt;
GetCursorPos(&pt);
pM->TrackPopupMenu(TPM_LEFTALIGN,pt.x,pt.y,this);
```

另一种做法是通过 CMenu::CreatePopupMenu() 建立一个弹出菜单，然后使用 TrackPopupMenu 弹出菜单。使用 CreatePopupMenu 创建的菜单也可以将其作为一个弹出项添加另一个菜单中。下面的伪代码演示了如何创建一个弹出菜单并进行修改后弹出：

```
CMenu menu1,menu2;
menu1.CreatePopupMenu
menu1.InsertMenu(1)
menu1.InsertMenu(2)
menu1.InsertMenu(3)

menu2.CreatePopupMenu
menu2.AppendMenu(MF_POPUP,1,menu1.Detach())  将弹出菜单加入  or InsertMenu...
menu2.InsertMenu("string desc");
menu.TrackPopupMenu(...)
```

3.4 文档、视、框架之间相互作用

一般来说用户的输入/输出基本都是通过视进行，但一些例外的情况下可能需要和框架直接发生作用，而在多视的情况下如何在视之间传递数据。

在使用菜单时大家会发现当一个菜单没有进行映射处理时为禁止状态，在多视的情况下菜单的状态和处理映射是和当前活动视相联系的，这样 MFC 可以保证视能正确的接收到各种消息，但有时候也会产生不便。有一个解决办法就是在框架中对消息进行处理，这样也可以保证当前文档可以通过框架得到当前消息。

在用户进行输入后如何使视的状态得到更新？这个问题在一个文档对应一个视图时是不存在的，但是现在有一个文档对应了两个视图，当在一个视上进行了输入时如何保证另一个视也得到通知呢？MFC 的做法是利用文档来处理的，因为文档管理着当前和它联系的视，由

它来通知各个视是最合适的。让我们同时看两个函数：

- void CView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
- void CDocument::UpdateAllViews(CView* pSender, LPARAM lHint = 0L, CObject* pHint = NULL)

当文档的 UpdateAllViews 被调用时和此文档相关的所有视的 OnUpdate 都会被调用，而参数 lHint 和 pHint 都会被传递。这样一来发生改变视就可以通知其他的兄弟了。那么还有一个问题：如何在 OnUpdate 中知道是那个视图发生了改变呢，这就可以利用 pHint 参数，只要调用者将 this 指针赋值给参数就可以了，当然完全可以利用该参数传递更复杂的结构。

视的初始化，当一个文档被打开或是新建一个文档时视图的 CView::OnInitialUpdate()会被调用，你可以通过重载该函数对视进行初始化，并在结束前调用父类的 OnInitialUpdate，因为这样可以保证 OnUpdate 会被调用。

文档中内容的清除，当文档被关闭时（比如退出或是新建前上一个文档清除）void CDocument::DeleteContents ()会被调用，你可以通过重载该函数来进行清理工作。

在单文档结构中上面两点尤其重要，因为软件运行文档对象和视对象只会被产生并删除一次。所以应该将上面两点和 C++对象构造和构析分清楚。

最后将一下文档模板（DocTemplate）的作用，文档模板分为两类单文档模板和多文档模板，分别由 CSingleDocTemplate 和 CMultiDocTemplate 表示，模板的作用在于记录文档，视，框架之间的对应关系。还有一点就是模板可以记录应用程序可以打开的文件的类型，当打开文件时会根据文档模板中的信息选择正确的文档和视。模板是一个比较抽想的概念，一般来说是不需要我们直接进行操作的。

当使用者通过视修改了数据时，应该调用 GetDocument()->SetModifiedFlag(TRUE)通知文档数据已经被更新，这样在关闭文档时会自动询问用户是否保存数据。

好象这一节讲的有些乱，大家看后有什么想法和问题请在 VCHelp 论坛上留言，我会尽快回复并且会对本节内容重新整理和修改

5 利用序列化进行文件读写

在很多应用中我们需要对数据进行保存，或是从介质上读取数据，这就涉及到文件的操作。我们可以利用各种文件存取方法完成这些工作，但 MFC 中也提供了一种读写文件的简单方法——“序列化”。序列化机制通过更高层次的接口功能向开发者提供了更利于使用和透明于字节流的文件操纵方法，举一个例来讲你可以将一个字串写入文件而不需要理会具体长度，读出时也是一样。你甚至可以对字符串数组进行操作。在 MFC 提供的可自动分配内存的类的支持下你可以更轻松的读/写数据。你也可以根据需要编写你自己的具有序列化功能的类。

序列化在最低的层次上应该被需要序列化的类支持，也就是说如果你需要对一个类进行序列化，那么这个类必须支持序列化。当通过序列化进行文件读写时你只需要该类的序列化函数就可以了。

怎样使类具有序列化功能呢？你需要以下的工作：

- 该类从 CObject 派生。
- 在类声明中包括 DECLARE_SERIAL 宏定义。
- 提供一个缺省的构造函数。
- 在类中实现 Serialze 函数
- 使用 IMPLEMENT_SERIAL 指明类名和版本号

下面的代码建立了一个简单身份证记录的类，同时也能够支持序列化。

```
in H
struct strPID
{
char szName[10];
char szID[16];
struct strPID* pNext;
};
class CAllPID : public CObject
{
public:
DECLARE_SERIAL(CAllPID)
CAllPID();
~CAllPID();

public:// 序列化相关
struct strPID* pHead;
//其他的成员函数
void Serialize(CArchive& ar);
};

in CPP
IMPLEMENT_SERIAL(CAllPID,CObject,1) // version is 1，版本用于读数据时的检测
void CAllPID::Serialize(CArchive& ar)
{
int iTotal;
if(ar.IsStoring())
{//保存数据
iTotal=GetTotalID();//得到链表中的记录数量
arr<>iTotal;
for(int i=0;i>*(((BYTE*)pID)+j);//读一个 strPID 中所有的数据
//修改链表
}
}
```

}

当然上面的代码很不完整，但已经可以说明问题。这样 CAllPID 就是一个可以支持序列化的类，并且可以根据记录的数量动态分配内存。在序列化中我们使用了 CArchive 类，该类用于在序列化时提供读写支持，它重载了<<和>>运算符号，并且提供 Read 和 Write 函数对数据进行读写。

下面看看如何在文档中使用序列化功能，你只需要修改文档类的 Serialize(CArchive& ar)函数，并调用各个进行序列化的类的 Serial 进行数据读写就可以了。当然你也可以在文档类的内部进行数据读写，下面的代码利用序列化功能读写数据：

```
class CYourDoc : public CDocument
{
void Serialize(CArchive& ar);
CString m_szDesc;
CAllPID m_allPID;
.....
}

void CYourDoc::Serialize(CArchive& ar)
{
if (ar.IsStoring())
{//由于 CString 对 CArchive 定义了<<和>>操作符号，所以可以直接利用>>和<<
ar<>m_szDesc;
}
m_allPID.Serialize(ar);//调用数据类的序列化函数
}
```

3.6 MFC 中所提供的各种视类介绍

MFC 中提供了丰富的视类供开发者使用，下面对各个类进行介绍：

CView 类是最基本的视类只支持最基本的操作。

CScrollView 类提供了滚动的功能，你可以利用 void CScrollView::SetScrollSizes(int nMapMode, SIZE sizeTotal, const SIZE& sizePage = sizeDefault, const SIZE& sizeLine = sizeDefault)设置滚动尺寸，和坐标映射模式。但是在绘图和接收用户输入时需要对坐标进行转换。请参见 3.2 接收用户输入。

CFormView 类提供用户在资源文件中定义界面的能力，并可以将子窗口和变量进行绑定。通过 UpdateData 函数让数据在变量和子窗口间交换。

CTreeView 类利用 TreeCtrl 界面作为视界面，通过调用 CTreeCtrl& CTreeView::GetTreeCtrl( )

const 得到 CTreeCtrl 的引用。

CListView 类利用 ListCtrl 界面作为视界面，通过调用 CTreeCtrl& CTreeView::GetTreeCtrl( ) const 得到 CListCtrl 的引用。

CEditView 类利用 Edit 接收用户输入，它具有输入框的一切功能。通过调用 CEdit& CEditView::GetEditCtrl( ) const 得到 Edit&的引用。void CEditView::SetPrinterFont(CFont* pFont)可以设置打印字体。

CRichEditView 类作为 Rich Text Edit（富文本输入）的视类，提供了可以按照格式显示文本的能力，在使用时需要 CRichEditDoc 的支持。

第四章 窗口控件

4.1 Button

按钮窗口(控件)在 MFC 中使用 CButton 表示，CButton 包含了三种样式的按钮，Push Button，Check Box，Radio Box。所以在利用 CButton 对象生成按钮窗口时需要指明按钮的风格。

创建按钮：BOOL CButton::Create(LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);其中 lpszCaption 是按钮上显示的文字，dwStyle 为按钮风格，除了 Windows 风格可以使用外（如 WS_CHILD|WS_VISUBLE|WS_BORDER）还有按钮专用的一些风格。

● BS_AUTOCHECKBOX 检查框，按钮的状态会自动改变 Same as a check box, except that a check mark appears in the check box when the user selects the box; the check mark disappears the next time the user selects the box.

● BS_AUTORADIOBUTTON 圆形选择按钮，按钮的状态会自动改变 Same as a radio button, except that when the user selects it, the button automatically highlights itself and removes the selection from any other radio buttons with the same style in the same group.

● BS_AUTO3STATE 允许按钮有三种状态即：选中，未选中，未定 Same as a three-state check box, except that the box changes its state when the user selects it.

● BS_CHECKBOX 检查框 Creates a small square that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style).

● BS_DEFPUSHBUTTON 默认普通按钮 Creates a button that has a heavy black border. The user can select this button by pressing the ENTER key. This style enables the user to quickly

select the most likely option (the default option).

● BS_LEFTTEXT 左对齐文字 When combined with a radio-button or check-box style, the text appears on the left side of the radio button or check box.

● BS_OWNERDRAW 自绘按钮 Creates an owner-drawn button. The framework calls the DrawItem member function when a visual aspect of the button has changed. This style must be set when using the CBitmapButton class.

● BS_PUSHBUTTON 普通按钮 Creates a pushbutton that posts a WM_COMMAND message to the owner window when the user selects the button.

● BS_RADIOBUTTON 圆形选择按钮 Creates a small circle that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style). Radio buttons are usually used in groups of related but mutually exclusive choices.

● BS_3STATE 允许按钮有三种状态即:选中,未选中,未定 Same as a check box, except that the box can be dimmed as well as checked. The dimmed state typically is used to show that a check box has been disabled.
rect 为窗口所占据的矩形区域，pParentWnd 为父窗口指针，nID 为该窗口的 ID 值。

获取/改变按钮状态：对于检查按钮和圆形按钮可能有两种状态，选中和未选中，如果设置了 BS_3STATE 或 BS_AUTO3STATE 风格就可能出现第三种状态：未定，这时按钮显示灰色。通过调用 int CButton::GetCheck( ) 得到当前是否被选中，返回 0：未选中，1：选中，2：未定。调用 void CButton::SetCheck(int nCheck);设置当前选中状态。

处理按钮消息：要处理按钮消息需要在父窗口中进行消息映射，映射宏为 ON_BN_CLICKED(id, memberFxn)id 为按钮的 ID 值，就是创建时指定的 nID 值。处理函数原型为 afx_msg void memberFxn( );

4.2 Static Box

静态文本控件的功能比较简单，可作为显示字符串，图标，位图用。创建一个窗口可以使用成员函数：
BOOL CStatic::Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);
其中 dwStyle 将指明该窗口的风格，除了子窗口常用的风格 WS_CHILD,WS_VISIBLE 外，你可以针对静态控件指明专门的风格。

- SS_CENTER,SS_LEFT,SS_RIGHT 指明字符显示的对齐方式。
- SS_GRAYRECT 显示一个灰色的矩形
- SS_NOPREFIX 如果指明该风格，对于字符&将直接显示，否则&将作为转义符，&将不显示而在其后的字符将有下划线，如果需要直接显示&必须使用&&表示。
- SS_BITMAP 显示位图
- SS_ICON 显示图标
- SS_CENTERIMAGE 图象居中显示

控制显示的文本利用成员函数 SetWindowText/GetWindowText 用于设置/得到当前显示的文本。

控制显示的图标利用成员函数 SetIcon/GetIcon 用于设置/得到当前显示的图标。

控制显示的位图利用成员函数 SetBitmap/GetBitmap 用于设置/得到当前显示的位图。下面一段代码演示如何创建一个显示位图的静态窗口并设置位图

```
CStatic* pstaDis=new CStatic;
pstaDis->Create("",WS_CHILD|WS_VISIBLE|SS_BITMAP|SSCENTERIMAGE,CRect(0,0,40,40),pWnd,1);
CBitmap bmpLoad;
bmpLoad.LoadBitmap(IDB_TEST);
pstaDis->SetBitmap(bmpLoad.Detach());
```

4.3 Edit Box

Edit 窗口是用来接收用户输入最常用的一个控件。创建一个输入窗口可以使用成员函数：BOOL CEdit::Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);
其中 dwStyle 将指明该窗口的风格，除了子窗口常用的风格 WS_CHILD,WS_VISIBLE 外，你可以针对输入控件指明专门的风格。

- ES_AUTOHSCROLL,ES_AUTOVSCROLL 指明输入文字超出显示范围时自动滚动。
- ES_CENTER,ES_LEFT,ES_RIGHT 指定对齐方式
- ES_MULTILINE 是否允许多行输入
- ES_PASSWORD 是否为密码输入框，如果指明该风格则输入的文字显示为*
- ES_READONLY 是否为只读
- ES_UPPERCASE,ES_LOWERCASE 显示大写/小写字符

控制显示的文本利用成员函数 SetWindowText/GetWindowText 用于设置/得到当前显示的文本。

通过 GetLimitText/SetLimitText 可以得到/设置在输入框中输入的字符数量。

由于在输入时用户可能选择某一段文本，所以通过 void CEdit::GetSel(int& nStartChar, int& nEndChar)得到用户选择的字符范围，通过调用 void CEdit::SetSel(int nStartChar, int

nEndChar, BOOL bNoScroll = FALSE)可以设置当前选择的文本范围，如果指定 nStartChar=0 nEndChar=-1 则表示选中所有的文本。void ReplaceSel(LPCTSTR lpszNewText, BOOL bCanUndo = FALSE)可以将选中的文本替换为指定的文字。

此外输入框还有一些和剪贴板有关的功能，void Clear( );删除选中的文本，void Copy( );可将选中的文本送入剪贴板，void Paste( );将剪贴板中内容插入到当前输入框中光标位置，void Cut( );相当于 Copy 和 Clear 结合使用。

最后介绍一下输入框几种常用的消息映射宏：

- ON_EN_CHANGE 输入框中文字更新后产生
- ON_EN_ERRSPACE 输入框无法分配内存时产生
- ON_EN_KILLFOCUS / ON_EN_SETFOCUS 在输入框失去/得到输入焦点时产生

使用以上几种消息映射的方法为定义原型如：afx_msg void memberFxn( );的函数，并且定义形式如 ON_Notification(id, memberFxn)的消息映射。如果在对话框中使用输入框，Class Wizard 会自动列出相关的消息，并能自动产生消息映射代码。

4.4 Scroll Bar

Scroll Bar 一般不会单独使用，因为 SpinCtrl 可以取代滚动条的一部分作用，但是如果你需要自己生成派生窗口，滚动条还是会派上一些用场。创建一个滚动条可以使用成员函数：：BOOL CEdit::Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);
其中 dwStyle 将指明该窗口的风格，除了子窗口常用的风格 WS_CHILD,WS_VISIBLE 外，你可以针对滚动条指明专门的风格。

- SBS_VERT 风格将创建一个垂直的滚动条。
- SBS_HORZ 风格将创建一个水平的滚动条。

在创建滚动条后需要调用 void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE)设置滚动范围，
int GetScrollPos( )/int SetScrollPos( )用来得到和设置当前滚动条的位置。

void ShowScrollBar(BOOL bShow = TRUE);用来显示/隐藏滚动条。

BOOL EnableScrollBar(UINT nArrowFlags = ESB_ENABLE_BOTH)用来设置滚动条上箭头是否为允许状态。nArrowFlags 可取以下值：

- ESB_ENABLE_BOTH 两个箭头都为允许状态
- ESB_DISABLE_LTUP 上/左箭头为禁止状态
- ESB_DISABLE_RTDN 下/右箭头为禁止状态
- ESB_DISABLE_BOTH 两个箭头都为禁止状态

如果需要在滚动条位置被改变时得到通知，需要在父窗口中定义对消息

WM_VSCROLL/WM_HSCROLL 的映射。方法为在父窗口类中重载
afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)/afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
所使用的消息映射宏为：ON_WM_VSCROLL( ),ON_WM_HSCROLL( )，在映射宏中不需要指明滚动条的 ID，因为所有滚动条的滚动消息都由同样的函数处理。在 OnHScroll/OnVScroll 的第三个参数会指明当前滚动条的指针。第一个参数表示滚动条上发生的动作，可取以下值：

- SB_TOP/SB_BOTTOM 已滚动到顶/底部
- SB_LINEUP/SB_LINEDOWN 向上/下滚动一行
- SB_PAGEDOWN/SB_PAGEUP 向上/下滚动一页
- SB_THUMBPOSITION/SB_THUMBTRACK 滚动条拖动到某一位置，参数 nPos 指明当前位置（参数 nPos 在其它的情况下是无效的）
- SB_ENDSCROLL 滚动条拖动完成（用户松开鼠标）

4.5 List Box/Check List Box

ListBox 窗口用来列出一系列的文本，每条文本占一行。创建一个列表窗口可以使用成员函数：
BOOL CListBox::Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);
其中 dwStyle 将指明该窗口的风格，除了子窗口常用的风格 WS_CHILD,WS_VISIBLE 外，你可以针对列表控件指明专门的风格。

- LBS_MULTIPLESEL 指明列表框可以同时选择多行
- LBS_EXTENDEDSEL 可以通过按下 Shift/Ctrl 键选择多行
- LBS_SORT 所有的行按照字母顺序进行排序

在列表框生成后需要向其中加入或是删除行，可以利用：
int AddString(LPCTSTR lpszItem)添加行，
int DeleteString(UINT nIndex)删除指定行，
int InsertString(int nIndex, LPCTSTR lpszItem)将行插入到指定位置。
void ResetContent( )可以删除列表框中所有行。
通过调用 int GetCount( )得到当前列表框中行的数量。

如果需要得到/设置当前被选中的行，可以调用 int GetCurSel( )/int SetCurSel(int iIndex)。如果你指明了选择多行的风格，你就需要先调用 int GetSelCount( )得到被选中的行的数量，然后 int GetSelItems(int nMaxItems, LPINT rgIndex)得到所有选中的行，参数 rgIndex 为存放被选中行的数组。通过调用 int GetLBText(int nIndex, LPTSTR lpszText)得到列表框内指定行的字符串。

此外通过调用 int FindString(int nStartAfter, LPCTSTR lpszItem)可以在当前所有行中查找指定的字符传的位置，nStartAfter 指明从那一行开始进行查找。
int SelectString(int nStartAfter, LPCTSTR lpszItem)可以选中包含指定字符串的行。

在 MFC 4.2 版本中添加了 CCheckListBox 类，该类是由 CListBox 派生并拥有 CListBox 的所有功能，不同的是可以在每行前加上一个检查框。必须注意的是在创建时必须指明 LBS_OWNERDRAWFIXED 或 LBS_OWNERDRAWVARIABLE 风格。

通过 void SetCheckStyle(UINT nStyle)/UINT GetCheckStyle( )可以设置/得到检查框的风格，关于检查框风格可以参考 4.1 Button 中介绍。通过 void SetCheck(int nIndex, int nCheck)/int GetCheck(int nIndex)可以设置和得到某行的检查状态，关于检查框状态可以参考 4.1 Button 中介绍。

最后介绍一下列表框几种常用的消息映射宏：

- ON_LBN_DBLCLK 鼠标双击
- ON_EN_ERRSPACE 输入框无法分配内存时产生
- ON_EN_KILLFOCUS / ON_EN_SETFOCUS 在输入框失去/得到输入焦点时产生
- ON_LBN_SELCHANGE 选择的行发生改变

使用以上几种消息映射的方法为定义原型如：afx_msg void memberFxn( );的函数，并且定义形式如 ON_Notification(id, memberFxn)的消息映射。如果在对话框中使用列表框，Class Wizard 会自动列出相关的消息，并能自动产生消息映射代码。

4.6 Combo Box/Combo Box Ex

组合窗口是由一个输入框和一个列表框组成。创建一个组合窗口可以使用成员函数：
BOOL CListBox::Create(LPCTSTR lpszText, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID = 0xffff);
其中 dwStyle 将指明该窗口的风格，除了子窗口常用的风格 WS_CHILD,WS_VISIBLE 外，你可以针对列表控件指明专门的风格。

- CBS_DROPDOWN 下拉式组合框
- CBS_DROPDOWNLIST 下拉式组合框，但是输入框内不能进行输入
- CBS_SIMPLE 输入框和列表框同时被显示
- LBS_SORT 所有的行按照字母顺序进行排序

由于组合框内包含了列表框，所以列表框的功能都能够使用，如可以利用：
int AddString(LPCTSTR lpszItem)添加行，
int DeleteString(UINT nIndex)删除指定行，
int InsertString(int nIndex, LPCTSTR lpszItem)将行插入到指定位置。
void ResetContent( )可以删除列表框中所有行。
通过调用 int GetCount( )得到当前列表框中行的数量。

如果需要得到/设置当前被选中的行的位置，可以调用 int GetCurSel( )/int SetCurSel(int iIndex)。通过调用 int GetLBText(int nIndex, LPTSTR lpszText)得到列表框内指定行的字符串。

此外通过调用 int FindString(int nStartAfter, LPCTSTR lpszItem)可以在当前所有行中查找指定的字符传的位置，nStartAfter 指明从那一行开始进行查找。
int SelectString(int nStartAfter, LPCTSTR lpszItem)可以选中包含指定字符串的行。

此外输入框的功能都能够使用，如可以利用：
DWORD GetEditSel( ) /BOOL SetEditSel(int nStartChar, int nEndChar)得到或设置输入框中被选中的字符位置。
BOOL LimitText(int nMaxChars)设置输入框中可输入的最大字符数。
输入框的剪贴板功能 Copy,Clear,Cut,Paste 动可以使用。

最后介绍一下列表框几种常用的消息映射宏：

- ON_CBN_DBLCLK 鼠标双击
- ON_CBN_DROPDOWN 列表框被弹出
- ON_CBN_KILLFOCUS / ON_CBN_SETFOCUS 在输入框失去/得到输入焦点时产生
- ON_CBN_SELCHANGE 列表框中选择的行发生改变
- ON_CBN_EDITUPDATE 输入框中内容被更新

使用以上几种消息映射的方法为定义原型如：afx_msg void memberFxn( );的函数，并且定义形式如 ON_Notification(id, memberFxn)的消息映射。如果在对话框中使用组合框，Class Wizard 会自动列出相关的消息，并能自动产生消息映射代码。

在 MFC 4.2 中对组合框进行了增强，你可以在组合框中使用 ImageList，有一个新的类 CComboBoxEx（由 CComboBox 派生）来实现这一功能。在 CComboBoxEx 类中添加了一些新的成员函数来实现新的功能：首先你需要调用
CImageList* SetImageList(CImageList* pImageList);来设置 ImageList，然后调用
int InsertItem(const COMBOBOXEXITEM* pCBItem);来添加行，其中 COMBOBOXEXITEM 定义如下：

```
typedef struct
{
UINT mask;
int iItem;
LPTSTR pszText;
int cchTextMax;
int iImage;
int iSelectedImage;
int iOverlay;
int iIndent;
LPARAM lParam;
} COMBOBOXEXITEM, *PCOMBOBOXEXITEM;
```

你需要设置 mask=CBEIF_IMAGE|CBEIF_TEXT，并设置 iItem 为插入位置，设置 pszText 为显示字符串，设置 iImage 为显示的图标索引。下面的代码演示了如何进行插入：

```
/*m_cbeWnd 为已经创建的 CComboBox 对象
```

```
m_list 为 CImageList 对象 IDB_IMG 为 16*(16*4)的位图，每个图片为 16*16 共 4 个图标*/
m_list.Create(IDB_IMG,16,4,RGB(0,0,0));
m_cbeWnd.SetImageList(&m_list);

COMBOBOXEXITEM insItem;
insItem.mask=CBEIF_IMAGE|CBEIF_TEXT;
insItem.iItem=0;
insItem.iImage=0;
insItem.pszText="Line 1";
m_cbeWnd.InsertItem(&insItem);
insItem.iItem=1;
insItem.iImage=1;
insItem.pszText="Line 2";
m_cbeWnd.InsertItem(&insItem);
```

通过调用 int DeleteItem(int iIndex);来删除行，并指明行的位置。
通过调用 BOOL GetItem(COMBOBOXEXITEM* pCBItem)/BOOL SetItem(const COMBOBOXEXITEM* pCBItem);来得到/设置行数据。

4.7 Tree Ctrl

树形控件 TreeCtrl 和下节要讲的列表控件 ListCtrl 在系统中大量被使用，例如 Windows 资源管理器就是一个典型的例子。

树形控件可以用于树形的结构，其中有一个根接点(Root)然后下面有许多子结点，而每个子结点上有允许有一个或多个或没有子结点。MFC 中使用 CTreeCtrl 类来封装树形控件的各种操作。通过调用 BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);创建一个窗口，dwStyle 中可以使用以下一些树形控件的专用风格：

- TVS_HASLINES 在父/子结点之间绘制连线
- TVS_LINESATROOT 在根/子结点之间绘制连线
- TVS_HASBUTTONS 在每一个结点前添加一个按钮，用于表示当前结点是否已被展开
- TVS_EDITLABELS 结点的显示字符可以被编辑
- TVS_SHOWSELALWAYS 在失去焦点时也显示当前选中的结点
- TVS_DISABLEDRAGDROP 不允许 Drag/Drop
- TVS_NOTOOLTIPS 不使用 ToolTip 显示结点的显示字符

在树形控件中每一个结点都有一个句柄（HTREEITEM），同时添加结点时必须提供的参数是该结点的父结点句柄，（其中根 Root 结点只有一个，既不可以添加也不可以删除）利用 HTREEITEM InsertItem(LPCTSTR lpszItem, HTREEITEM hParent = TVI_ROOT, HTREEITEM hInsertAfter = TVI_LAST);可以添加一个结点，pszItem 为显示的字符，hParent 代表父结点的句柄，当前添加的结点会排在 hInsertAfter 表示的结点的后面，返回值为当前创建的结点的句柄。下面的代码会建立一个如下形式的树形结构：

```
+--- Parent1
□□+--- Child1_1
□□+--- Child1_2
```

□□+--- Child1_3
+--- Parent2
+--- Parent3

/*假设 m_tree 为一个 CTreeCtrl 对象，而且该窗口已经创建*/
HTREEITEM hItem,hSubItem;
hItem = m_tree.InsertItem("Parent1",TVI_ROOT);在根结点上添加 Parent1
hSubItem = m_tree.InsertItem("Child1_1",hItem);//在 Parent1 上添加一个子结点
hSubItem = m_tree.InsertItem("Child1_2",hItem,hSubItem);//在 Parent1 上添加一个子结点，排在 Child1_1 后面
hSubItem = m_tree.InsertItem("Child1_3",hItem,hSubItem);

hItem = m_tree.InsertItem("Parent2",TVI_ROOT,hItem);
hItem = m_tree.InsertItem("Parent3",TVI_ROOT,hItem);

如果你希望在每个结点前添加一个小图标，就必需先调用 CImageList* SetImageList(CImageList * pImageList, int nImageListType);指明当前所使用的 ImageList，nImageListType 为 TVSIL_NORMAL。在调用完成后控件中使用图片以设置的 ImageList 中图片为准。然后调用
HTREEITEM InsertItem(LPCTSTR lpszItem, int nImage, int nSelectedImage, HTREEITEM hParent = TVI_ROOT, HTREEITEM hInsertAfter = TVI_LAST);添加结点，nImage 为结点没被选中时所使用图片序号，nSelectedImage 为结点被选中时所使用图片序号。下面的代码演示了 ImageList 的设置。
/*m_list 为 CImageList 对象
IDB_TREE 为 16*(16*4)的位图，每个图片为 16*16 共 4 个图标*/
m_list.Create(IDB_TREE,16,4,RGB(0,0,0));
m_tree.SetImageList(&m_list,TVSIL_NORMAL);
m_tree.InsertItem("Parent1",0,1);//添加，选中时显示图标 1，未选中时显示图标 0

此外 CTreeCtrl 还提供了一些函数用于得到/修改控件的状态。
HTREEITEM GetSelectedItem( ); 将返回当前选中的结点的句柄。BOOL SelectItem(HTREEITEM hItem);将选中指明结点。
BOOL GetItemImage(HTREEITEM hItem, int& nImage, int& nSelectedImage) / BOOL SetItemImage(HTREEITEM hItem, int nImage, int nSelectedImage)用于得到/修改某结点所使用图标索引。
CString GetItemText(HTREEITEM hItem) /BOOL SetItemText(HTREEITEM hItem, LPCTSTR lpszItem);用于得到/修改某一结点的显示字符。
BOOL DeleteItem(HTREEITEM hItem);用于删除某一结点，BOOL DeleteAllItems( );将删除所有结点。

此外如果想遍历树可以使用下面的函数：
HTREEITEM GetRootItem( );得到根结点。
HTREEITEM GetChildItem(HTREEITEM hItem);得到子结点。

HTREEITEM GetPrevSiblingItem/GetNextSiblingItem(HTREEITEM hItem);得到指明结点的上/下一个兄弟结点。
HTREEITEM GetParentItem(HTREEITEM hItem);得到父结点。

树形控件的消息映射使用 ON_NOTIFY 宏，形式如同：ON_NOTIFY(wNotifyCode, id, memberFxn)，wNotifyCode 为通知代码，id 为产生该消息的窗口 ID，memberFxn 为处理函数，函数的原型如同 void OnXXXTree(NMHDR* pNMHDR, LRESULT* pResult)，其中 pNMHDR 为一数据结构，在具体使用时需要转换成其他类型的结构。对于树形控件可能取值和对应的数据结构为：

- TVN_SELCHANGED 在所选中的结点发生改变后发送，所用结构：NMTREEVIEW
- TVN_ITEMEXPANDED 在某结点被展开后发送，所用结构：NMTREEVIEW
- TVN_BEGINLABELEDIT 在开始编辑结点字符时发送，所用结构：NMTVDISPINFO
- TVN_ENDLABELEDIT 在结束编辑结点字符时发送，所用结构：NMTVDISPINFO
- TVN_GETDISPINFO 在需要得到某结点信息时发送，（如得到结点的显示字符）所用结构：NMTVDISPINFO

关于 ON_NOTIFY 有很多内容，将在以后的内容中进行详细讲解。

关于动态提供结点所显示的字符：首先你在添加结点时需要指明 lpszItem 参数为：LPSTR_TEXTCALLBACK。在控件显示该结点时会通过发送 TVN_GETDISPINFO 来取得所需要的字符，在处理该消息时先将参数 pNMHDR 转换为 LPNMTVDISPINFO，然后填充其中 item.pszText。但是我们通过什么来知道该结点所对应的信息呢，我的做法是在添加结点后设置其 lParam 参数，然后在提供信息时利用该参数来查找所对应的信息。下面的代码说明了这种方法：

```
char szOut[8][3]={"No.1","No.2","No.3"};

//添加结点
HTREEITEM hItem = m_tree.InsertItem(LPSTR_TEXTCALLBACK,...)
m_tree.SetItemData(hItem, 0 );
hItem = m_tree.InsertItem(LPSTR_TEXTCALLBACK,...)
m_tree.SetItemData(hItem, 1 );
//处理消息
void CParentWnd::OnGetDispInfoTree(NMHDR* pNMHDR, LRESULT* pResult)
{
TV_DISPINFO* pTVDI = (TV_DISPINFO*)pNMHDR;
pTVDI->item.pszText=szOut[pTVDI->item.lParam];//通过 lParam 得到需要显示的字符在数组中的位置
*pResult = 0;
}
```

关于编辑结点的显示字符：首先需要设置树形控件的 TVS_EDITLABELS 风格，在开始编辑时该控件将会发送 TVN_BEGINLABELEDIT，你可以通过在处理函数中返回 TRUE 来取消

接下来的编辑，在编辑完成后会发送 TVN_ENDLABELEDIT，在处理该消息时需要将参数 pNMHDR 转换为 LPNMTVDISPINFO，然后通过其中的 item.pszText 得到编辑后的字符，并重置显示字符。如果编辑在中途中取消该变量为 NULL。下面的代码说明如何处理这些消息：

```
//处理消息 TVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditTree(NMHDR* pNMHDR, LRESULT* pResult)
{
TV_DISPINFO* pTVDI = (TV_DISPINFO*)pNMHDR;
if(pTVDI->item.lParam==0);//判断是否取消该操作
*pResult = 1;
else
*pResult = 0;
}
//处理消息 TVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditTree(NMHDR* pNMHDR, LRESULT* pResult)
{
TV_DISPINFO* pTVDI = (TV_DISPINFO*)pNMHDR;
if(pTVDI->item.pszText==NULL);//判断是否已经取消取消编辑
m_tree.SetItemText(pTVDI->item.hItem,pTVDI->pszText);//重置显示字符
*pResult = 0;
}
```

上面讲述的方法所进行的消息映射必须在父窗口中进行（同样 WM_NOTIFY 的所有消息都需要在父窗口中处理）

4.8 List Ctrl

列表控件可以看作是功能增强的 ListBox，它提供了四种风格，而且可以同时显示一列的多中属性值。MFC 中使用 CListCtrl 类来封装列表控件的各种操作。通过调用 BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);创建一个窗口，dwStyle 中可以使用以下一些列表控件的专用风格：

- LVS_ICON LVS_SMALLICON LVS_LIST LVS_REPORT 这四种风格决定控件的外观，同时只可以选择其中一种，分别对应：大图标显示，小图标显示，列表显示，详细报表显示
- LVS_EDITLABELS 结点的显示字符可以被编辑，对于报表风格来讲可编辑的只为第一列。
- LVS_SHOWSELALWAYS 在失去焦点时也显示当前选中的结点
- LVS_SINGLESEL 同时只能选中列表中一项

首先你需要设置列表控件所使用的 ImageList，如果你使用大图标显示风格，你就需要以如下形式调用：

```
CImageList* SetImageList( CImageList* pImageList, LVSIL_NORMAL);
```

如果使用其它三种风格显示而不想显示图标你可以不进行任何设置，否则需要以如下形式调用：

```
CImageList* SetImageList( CImageList* pImageList, LVSIL_SMALL);
```

通过调用 int InsertItem(int nItem, LPCTSTR lpszItem);可以在列表控件中 nItem 指明位置插入一项，lpszItem 为显示字符。除 LVS_REPORT 风格外其他三种风格都只需要直接调用 InsertItem 就可以了，但如果使用报表风格就必须先设置列表控件中的列信息。

通过调用 int InsertColumn(int nCol, LPCTSTR lpszColumnHeading, int nFormat , int nWidth, int nSubItem);可以插入列。iCol 为列的位置，从零开始，lpszColumnHeading 为显示的列名，nFormat 为显示对齐方式，nWidth 为显示宽度，nSubItem 为分配给该列的列索引。

在有多列的列表控件中就需要为每一项指明其在每一列中的显示字符，通过调用 BOOL SetItemText(int nItem, int nSubItem, LPTSTR lpszText);可以设置每列的显示字符。nItem 为设置的项的位置，nSubItem 为列位置，lpszText 为显示字符。下面的代码演示了如何设置多列并插入数据：

```
m_list.SetImageList(&m_listSmall,LVSIL_SMALL);//设置 ImageList
m_list.InsertColumn(0,"Col 1",LVCFMT_LEFT,300,0);//设置列
m_list.InsertColumn(1,"Col 2",LVCFMT_LEFT,300,1);
m_list.InsertColumn(2,"Col 3",LVCFMT_LEFT,300,2);

m_list.InsertItem(0,"Item 1_1");//插入行
m_list.SetItemText(0,1,"Item 1_2");//设置该行的不同列的显示字符
m_list.SetItemText(0,2,"Item 1_3");
```

此外 CListCtrl 还提供了一些函数用于得到/修改控件的状态。
COLORREF GetTextColor( )/BOOL SetTextColor(COLORREF cr);用于得到/设置显示的字符颜色。
COLORREF GetTextBkColor( )/BOOL SetTextBkColor(COLORREF cr);用于得到/设置显示的背景颜色。
void SetItemCount(int iCount);用于得到添加进列表中项的数量。
BOOL DeleteItem(int nItem);用于删除某一项，BOOL DeleteAllItems( );将删除所有项。
BOOL SetBkImage(HBITMAP hbm, BOOL fTile , int xOffsetPercent, int yOffsetPercent);用于设置背景位图。
CString GetItemText(int nItem, int nSubItem);用于得到某项的显示字符。

列表控件的消息映射同样使用 ON_NOTIFY 宏，形式如同：ON_NOTIFY(wNotifyCode, id, memberFxn)，wNotifyCode 为通知代码，id 为产生该消息的窗口 ID，memberFxn 为处理函数，函数的原型如同 void OnXXXList(NMHDR* pNMHDR, LRESULT* pResult)，其中 pNMHDR 为一数据结构，在具体使用时需要转换成其他类型的结构。对于列表控件可能取值和对应的数据结构为：

- LVN_BEGINLABELEDIT 在开始某项编辑字符时发送，所用结构：NMLVDISPINFO
- LVN_ENDLABELEDIT 在结束某项编辑字符时发送，所用结构：NMLVDISPINFO
- LVN_GETDISPINFO 在需要得到某项信息时发送，（如得到某项的显示字符）所用结构：NMLVDISPINFO

关于 ON_NOTIFY 有很多内容，将在以后的内容中进行详细讲解。

关于动态提供结点所显示的字符：首先你在项时需要指明 lpszItem 参数为：LPSTR_TEXTCALLBACK。在控件显示该结点时会通过发送 TVN_GETDISPINFO 来取得所需要的字符，在处理该消息时先将参数 pNMHDR 转换为 LPNMLVDISPINFO，然后填充其中 item.pszText。通过 item 中的 iItem,iSubItem 可以知道当前显示的为那一项。下面的代码演示了这种方法：

```
char szOut[8][3]={"No.1","No.2","No.3"};

//添加结点
m_list.InsertItem(LPSTR_TEXTCALLBACK,...)
m_list.InsertItem(LPSTR_TEXTCALLBACK,...)
//处理消息
void CParentWnd::OnGetDispInfoList(NMHDR* pNMHDR, LRESULT* pResult)
{
LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
pLVDI->item.pszText=szOut[pTVDI->item.iItem];//通过 iItem 得到需要显示的字符在数组中的位置
*pResult = 0;
}
```

关于编辑某项的显示字符：（在报表风格中只对第一列有效）首先需要设置列表控件的 LVS_EDITLABELS 风格，在开始编辑时该控件将会发送 LVN_BEGINLABELEDIT，你可以通过在处理函数中返回 TRUE 来取消接下来的编辑，在编辑完成后会发送 LVN_ENDLABELEDIT，在处理该消息时需要将参数 pNMHDR 转换为 LPNMLVDISPINFO，然后通过其中的 item.pszText 得到编辑后的字符，并重置显示字符。如果编辑在中途中取消该变量为 NULL。下面的代码说明如何处理这些消息：

```
//处理消息 LVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditList(NMHDR* pNMHDR, LRESULT* pResult)
{
LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
if(pLVDI->item.iItem==0);//判断是否取消该操作
*pResult = 1;
else
*pResult = 0;
}
//处理消息 LVN_BEGINLABELEDIT
void CParentWnd::OnBeginEditList(NMHDR* pNMHDR, LRESULT* pResult)
{
LV_DISPINFO* pLVDI = (LV_DISPINFO*)pNMHDR;
if(pLVDI->item.pszText==NULL);//判断是否已经取消取消编辑
```

```
m_list.SetItemText(pLVDI->item.iItem,0,pLVDI->pszText);//重置显示字符
*pResult = 0;
}
```

上面讲述的方法所进行的消息映射必须在父窗口中进行（同样 WM_NOTIFY 的所有消息都需要在父窗口中处理）。

如何得到当前选中项位置：在列表控件中没有一个类似于 ListBox 中 GetCurSel()的函数，但是可以通过调用 GetNextItem(-1, LVNI_ALL | LVNI_SELECTED);得到选中项位置

4.9 Tab Ctrl

Tab 属性页控件可以在一个窗口中添加不同的页面，然后在页选择发生改变时得到通知。MFC 中使用 CTabCtrl 类来封装属性页控件的各种操作。通过调用 BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);创建一个窗口，dwStyle 中可以使用以下一些属性页控件的专用风格：

- TCS_BUTTONS 使用按钮来表示页选择位置
- TCS_MULTILINE 分行显示页选择位置
- TCS_SINGLELINE 只使用一行显示页选择位置

在控件创建后必需向其中添加页面才可以使用，添加页面的函数为：
BOOL InsertItem(int nItem, LPCTSTR lpszItem);nItem 为位置，从零开始，lpszItem 为页选择位置上显示的文字。如果你希望在页选择位置处显示一个图标，你可以调用
BOOL InsertItem(int nItem, LPCTSTR lpszItem, int nImage);nImage 指明所使用的图片位置。（在此之前必须调用 CImageList * SetImageList(CImageList * pImageList);设置正确的 ImageList）

此外 CTabCtrl 还提供了一些函数用于得到/修改控件的状态。
int GetCurSel( )/int SetCurSel(int nItem);用于得到/设置当前被选中的页位置。
BOOL DeleteItem(int nItem)/BOOL DeleteAllItems( );用于删除指定/所有页面。
void RemoveImage(int nImage);用于删除某页选择位置上的图标。

属性页控件的消息映射同样使用 ON_NOTIFY 宏，形式如同：ON_NOTIFY(wNotifyCode, id, memberFxn)，wNotifyCode 为通知代码，id 为产生该消息的窗口 ID，memberFxn 为处理函数，函数的原型如同 void OnXXXTab(NMHDR* pNMHDR, LRESULT* pResult)，其中 pNMHDR 为一数据结构，在具体使用时需要转换成其他类型的结构。对于列表控件可能取值和对应的数据结构为：

- TCN_SELCHANGE 在当前页改变后发送，所用结构：NMHDR
- TCN_SELCHANGING 在当前页改变时发送可以通过返回 TRUE 来禁止页面的改变，所用结构：NMHDR

一般来讲在当前页发生改变时需要隐藏当前的一些子窗口，并显示其它的子窗口。下面的伪代码演示了如何使用属性页控件：

```
CParentWnd::OnCreate(...)
{
m_tab.Create(...);
m_tab.InsertItem(0,"Option 1");
m_tab.InsertItem(1,"Option 2");
Create a edit box as the m_tab's Child
Create a static box as the m_tab's Child
edit_box.ShowWindow(SW_SHOW); // edit box 在属性页的第一页
static_box.ShowWindow(SW_HIDE); // static box 在属性页的第二页
}
void CParentWnd::OnSelectChangeTab(NMHDR* pNMHDR, LRESULT* pResult)
{//处理页选择改变后的消息
if(m_tab.GetCurSel()==0)
{//根据当前页显示/隐藏不同的子窗口
edit_box.ShowWindow(SW_SHOW);
static_box.ShowWindow(SW_HIDE);
}
else
{//
edit_box.ShowWindow(SW_HIDE);
static_box.ShowWindow(SW_SHOW);
}
}
```

4.A Tool Bar

工具条也是常用的控件。MFC 中使用 CToolBar 类来封装工具条控件的各种操作。通过调用 BOOL Create(CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP, UINT nID = AFX_IDW_TOOLBAR);创建一个窗口，dwStyle 中可以使用以下一些工具条控件的专用风格：

- CBRS_TOP 工具条在父窗口的顶部
- TCBRS_BOTTOM 工具条在父窗口的底部
- CBRS_FLOATING 工具条是浮动的

创建一个工具条的步骤如下：先使用 Create 创建窗口，然后使用 BOOL LoadToolBar(LPCTSTR lpszResourceName);直接从资源中装入工具条，或者通过装入位图并指明每个按钮的 ID，具体代码如下：

```
UINT uID[5]={IDM_1,IDM_2,IDM_3,IDM_4,IDM_5};
m_toolbar.Create(pParentWnd);
m_toolbar.LoadBitmap(IDB_TOOLBAR);
m_toolbar.SetSizes(CSize(20,20),CSize(16,16));//设置按钮大尺寸和按钮上位图的尺寸
m_toolbar.SetButtons(uID,5);
```

AppWizard 在生成代码时也会同时生成工具条的代码，同时还可以支持停靠功能。所以一般

是不需要直接操作工具条对象。

工具条上的按钮被按下时发送给父窗口的消息和菜单消息相同，所以可以使用 ON_COMMAND 宏进行映射，同样工具条中的按钮也支持 ON_UPDATE_COMMAND_UI 的相关操作，如 SetCheck,Enable，你可以将按钮的当作菜单上的一个具有相同 ID 菜单项。

在以后的章节 4.D 利用 AppWizard 创建并使用 ToolBar StatusBar Dialog Bar 会给出使用的方法。

4.B Status Bar

状态条用于显示一些提示字符。MFC 中使用 CStatusBar 类来封装状态条控件的各种操作。通过调用

BOOL Create(CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM, UINT nID = AFX_IDW_STATUS_BAR);创建一个窗口，dwStyle 中可以使用以下一些状态条控件的专用风格：

- CBRS_TOP 状态条在父窗口的顶部
- TCBRS_BOTTOM 状态条在父窗口的底部

创建一个状态条的步骤如下：先使用 Create 创建窗口，然后调用 BOOL SetIndicators(const UINT* lpIDArray, int nIDCount);设置状态条上各部分的 ID，具体代码如下：

```
UINT uID[2]={ID_SEPARATOR,ID_INDICATOR_CAPS};
m_stabar.Create(pParentWnd);
m_stabar.SetIndicators(uID,2);
```

通过 CString GetPaneText(int nIndex)/BOOL SetPaneText(int nIndex, LPCTSTR lpszNewText, BOOL bUpdate = TRUE)可以得到/设置状态条上显示的文字。

Tip：在创建状态条时最好将状态条中所有的部分 ID（除 MFC 自定义的几个用于状态条的 ID 外）都设置为 ID_SEPARATOR，在生成后调用

void SetPaneInfo(int nIndex, UINT nID, UINT nStyle, int cxWidth);改变其风格，ID 和宽度。

AppWizard 在生成代码时也会同时生成状态条的代码。所以一般是不需要直接创建状态条对象。此外状态条上会自动显示菜单上的命令提示（必须先在资源中定义），所以也不需要人为设置显示文字。

状态条支持 ON_UPDATE_COMMAND_UI 的相关操作，如 SetText，Enable。

在以后的章节 4.D 利用 AppWizard 创建并使用 ToolBar StatusBar Dialog Bar 会给出使用的方法。

4.C Dialog Bar

Dialog Bar 类似一个静态的附在框架窗口上的对话框，由于 Dialog Bar 可以使用资源编辑器

进行编辑所以使用起来就很方便，在设计时就可以对 Dialog Bar 上的子窗口进行定位。用于显示一些提示字符。MFC 中使用 CDialogBar 类来 Dialog Bar 控件的各种操作。通过调用 BOOL Create(CWnd* pParentWnd, UINT nIDTemplate, UINT nStyle, UINT nID);创建一个窗口，nIDTemplate 为对话框资源，nID 为该 Dialog Bar 对应的窗口 ID，nStyle 中可以使用以下一些状态条控件的专用风格：

- CBRS_TOP Dialog Bar 在父窗口的顶部
- TCBRS_BOTTOM Dialog Bar 在父窗口的底部
- CBRS_LEFT Dialog Bar 在父窗口的左部
- CBRS_RIGHT Dialog Bar 在父窗口的右部

对于 Dialog Bar 的所产生消息需要在父窗口中进行映射和处理，例如 Dialog Bar 上的按钮，需要在父窗口中进行 ON_BN_CLICKED 或 ON_COMMAND 映射，Dialog Bar 上的输入框可以在父窗口中进行 ON_EN_CHANGE，ON_EN_MAXTEXT 等输入框对应的映射。

Dialog Bar 支持 ON_UPDATE_COMMAND_UI 的相关操作，如 SetText，Enable。

在以后的章节 4.D 利用 AppWizard 创建并使用 ToolBar StatusBar Dialog Bar 会给出使用的方法。

4.D 利用 AppWizard 创建并使用 ToolBar StatusBar Dialog Bar

运行时程序界面如界面图，该程序拥有一个工具条用于显示两个命令按钮，一个用于演示如何使按钮处于检查状态，另一个根据第一个按钮的状态来禁止/允许自身。（设置检查状态和允许状态都通过 OnUpdateCommand 实现）此外 Dialog Bar 上有一个输入框和按钮，这两个子窗口的禁止/允许同样是根据工具条上的按钮状态来确定，当按下 Dialog Bar 上的按钮时将显示输入框中的文字内容。状态条的第一部分用于显示各种提示，第二部分用于利用 OnUpdateCommand 显示当前时间。同时在程序中演示了如何设置菜单项的命令解释字符（将在状态条的第一部分显示）和如何设置工具条的提示字符（利用一个小的 ToolTip 窗口显示）。

生成应用：利用 AppWizard 生成一个 MFC 工程，图例，并设置为单文档界面图例，最后选择工具条，状态条和 ReBar 支持，图例

修改菜单：利用资源编辑器删除多余的菜单并添加一个新的弹出菜单和三个子菜单，图例，分别是：

名称 ID 说明字符

Check IDM_CHECK SetCheck Demo\nSetCheck Demo

Disable IDM_DISABLE Disable Demo\nDisable Demo

ShowText on DialogBar IDM_SHOW_TXT ShowText on DialogBar Demo\nShowText on DialogBar

\n 前的字符串将显示在状态条中作为命令解释，\n 后的部分将作为具有相同 ID 的工具条按钮的提示显示在 ToolTip 窗口中。

修改 Dialog Bar：在 Dialog Bar 中添加一个输入框和按钮，按钮的 ID 为 IDM_SHOW_TXT 与一个菜单项具有相同的 ID，这样可以利用映射菜单消息来处理按钮消息（当然使用不同

ID 值也可以利用 ON_COMMAND 来映射 Dialog Bar 上的按钮消息，但是 ClassWizard 没有提供为 Dialog Bar 上按钮进行映射的途径，只能手工添加消息映射代码）。图例

修改工具条：在工具条中添加两个按钮，ID 值为 IDM_CHECK 和 IDM_DISABLE 和其中两个菜单项具有相同的 ID 值。图例

利用 ClassWizard 为三个菜单项添加消息映射和更新命令。图例

修改 MainFrm.h 文件

```
//添加一个成员变量来记录工具条上 Check 按钮的检查状态。
protected:
BOOL m_fCheck;
//手工添加状态条第二部分用于显示时间的更新命令，和用于禁止/允许输入框的更新命令
//{{AFX_MSG(CMainFrame)
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnCheck();
afx_msg void OnUpdateCheck(CCmdUI* pCmdUI);
afx_msg void OnDisable();
afx_msg void OnUpdateDisable(CCmdUI* pCmdUI);
afx_msg void OnShowTxt();
afx_msg void OnUpdateShowTxt(CCmdUI* pCmdUI);
//}}AFX_MSG
//上面的部分为 ClassWizard 自动产生的代码
afx_msg void OnUpdateTime(CCmdUI* pCmdUI); //显示时间
afx_msg void OnUpdateInput(CCmdUI* pCmdUI); //禁止/允许输入框
```

修改 MainFrm.cpp 文件

```
//修改状态条上各部分 ID
#define ID_TIME 0x705 //作为状态条上第二部分 ID
static UINT indicators[] =
{
ID_SEPARATOR, // status line indicator
ID_SEPARATOR, //先设置为 ID_SEPARATOR，在状态条创建后再进行修改
};
//修改消息映射
//{{AFX_MSG_MAP(CMainFrame)
ON_WM_CREATE()
ON_COMMAND(IDM_CHECK, OnCheck)
ON_UPDATE_COMMAND_UI(IDM_CHECK, OnUpdateCheck)
ON_COMMAND(IDM_DISABLE, OnDisable)
ON_UPDATE_COMMAND_UI(IDM_DISABLE, OnUpdateDisable)
```

```
ON_COMMAND(IDM_SHOW_TXT, OnShowTxt)
ON_UPDATE_COMMAND_UI(IDM_SHOW_TXT, OnUpdateShowTxt)
//}}AFX_MSG_MAP
//以上部分为 ClassWizard 自动生成代码
ON_UPDATE_COMMAND_UI(ID_TIME, OnUpdateTime) ////显示时间
ON_UPDATE_COMMAND_UI(IDC_INPUT_TEST, OnUpdateInput) //禁止/允许输入框
//修改 OnCreate 函数，重新设置状态条第二部分 ID 值
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
....
// by wenyy  修改状态条上第二部分信息
m_wndStatusBar.SetPaneInfo(1,ID_TIME,SBPS_NORMAL,60);//set the width
return 0;
}
//修改经过映射的消息处理函数代码
void CMainFrame::OnCheck()
{
//在 Check 按钮被按下时改变并保存状态
m_fCheck=!m_fCheck;
}

void CMainFrame::OnUpdateCheck(CCmdUI* pCmdUI)
{
//Check 按钮是否设置为检查状态
pCmdUI->SetCheck(m_fCheck);
}

void CMainFrame::OnDisable()
{
//Disable 按钮被按下
AfxMessageBox("you press disable test");
}

void CMainFrame::OnUpdateDisable(CCmdUI* pCmdUI)
{
//根据 Check 状态决定自身禁止/允许状态
pCmdUI->Enable(m_fCheck);
}

void CMainFrame::OnShowTxt()
{
//得到 Dialog Bar 上输入框中文字并显示
CEdit* pE=(CEdit*)m_wndDlgBar.GetDlgItem(IDC_INPUT_TEST);
CString szO;
```

```
pE->GetWindowText(szO);
AfxMessageBox(szO);
}

void CMainFrame::OnUpdateShowTxt(CCmdUI* pCmdUI)
{
//Dialog Bar 上按钮根据 Check 状态决定自身禁止/允许状态
pCmdUI->Enable(m_fCheck);
}

void CMainFrame::OnUpdateInput(CCmdUI* pCmdUI)
{
//Dialog Bar 上输入框根据 Check 状态决定自身禁止/允许状态
pCmdUI->Enable(m_fCheck);
}

void CMainFrame::OnUpdateTime(CCmdUI* pCmdUI)
{
//根据当前时间设置状态条上第二部分文字
CTime timeCur=CTime::GetCurrentTime();
char szOut[20];
sprintf( szOut, "%02d:%02d:%02d", timeCur.GetHour(),
timeCur.GetMinute(),timeCur.GetSecond());
pCmdUI->SetText(szOut);
}
```

4.E General Window

从 VC 提供的 MFC 类派生图中我们可以看出窗口的派生关系，派生图，所有的窗口类都是由 CWnd 派生。所有 CWnd 的成员函数在其派生类中都可以使用。本节介绍一些常用的功能给大家。

改变窗口状态：

BOOL EnableWindow(BOOL bEnable = TRUE);可以设置窗口的禁止/允许状态。BOOL IsWindowEnabled( );可以查询窗口的禁止/允许状态。

BOOL ModifyStyle(DWORD dwRemove, DWORD dwAdd, UINT nFlags = 0)/BOOL ModifyStyleEx(DWORD dwRemove, DWORD dwAdd, UINT nFlags = 0);可以修改窗口的风格，而不需要调用 SetWindowLong

BOOL IsWindowVisible( ) 可以检查窗口是否被显示。

BOOL ShowWindow(int nCmdShow);将改变窗口的显示状态，nCmdShow 可取如下值：

- SW_HIDE 隐藏窗口
- SW_MINIMIZE SW_SHOWMAXIMIZED 最小化窗口
- SW_RESTORE 恢复窗口

- SW_SHOW 显示窗口
- SW_SHOWMINIMIZED 最大化窗口

改变窗口位置：
void MoveWindow(LPCRECT lpRect, BOOL bRepaint = TRUE);可以移动窗口。
void GetWindowRect(LPRECT lpRect) ;可以得到窗口的矩形位置。
BOOL IsIconic( ) ;可以检测窗口是否已经缩为图标。
BOOL SetWindowPos(const CWnd* pWndInsertAfter, int x, int y, int cx, int cy, UINT nFlags);
可以改变窗口的 Z 次序，此外还可以移动窗口位置。

使窗口失效，印发重绘：
void Invalidate(BOOL bErase = TRUE);使整个窗口失效，bErase 将决定窗口是否产生重绘。
void InvalidateRect(LPCRECT lpRect, BOOL bErase = TRUE)/void InvalidateRgn(CRgn* pRgn, BOOL bErase = TRUE);将使指定的矩形/多边形区域失效。

窗口查找：
static CWnd* PASCAL FindWindow(LPCTSTR lpszClassName, LPCTSTR lpszWindowName);
可以以窗口的类名和窗口名查找窗口。任一参数设置为 NULL 表对该参数代表的数据进行任意匹配。如 FindWindow("MyWnd",NULL)表明查找类名为 MyWnd 的所有窗口。
BOOL IsChild(const CWnd* pWnd) 检测窗口是否为子窗口。
CWnd* GetParent( ) 得到父窗口指针。
CWnd* GetDlgItem(int nID) 通过子窗口 ID 得到窗口指针。
int GetDlgCtrlID( ) 得到窗口 ID 值。
static CWnd* PASCAL WindowFromPoint(POINT point);将从屏幕上某点坐标得到包含该点的窗口指针。
static CWnd* PASCAL FromHandle(HWND hWnd);通过 HWND 构造一个 CWnd*指针，但该指针在空闲时会被删除，所以不能保存供以后使用。

时钟：
UINT SetTimer(UINT nIDEvent, UINT nElapse, void (CALLBACK EXPORT* lpfnTimer)(HWND, UINT, UINT, DWORD));可以创建一个时钟，如果 lpfnTimer 回调函数为 NULL，窗口将会收到 WM_TIMER 消息，并可以在 afx_msg void OnTimer(UINT nIDEvent);中安排处理代码
BOOL KillTimer(int nIDEvent);删除一个指定时钟。

可以利用重载来添加消息处理的虚函数：
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);窗口被创建时被调用
afx_msg void OnDestroy( );窗口被销毁时被调用
afx_msg void OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI);需要得到窗口尺寸时被调用
afx_msg void OnSize(UINT nType, int cx, int cy);窗口改变大小后被调用
afx_msg void OnMove(int x, int y);窗口被移动后时被调用
afx_msg void OnPaint( );窗口需要重绘时时被调用，你可以填如绘图代码，对于视图类不需要重载 OnPaint，所有绘图代码应该在 OnDraw 中进行
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);接收到字符输入时被调用

afx_msg void OnKeyDown/OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);键盘上键被按下/放开时被调用
afx_msg void OnLButtonDown/OnRButtonDown(UINT nFlags, CPoint point);鼠标左/右键按下时被调用
afx_msg void OnLButtonUp/OnRButtonUp(UINT nFlags, CPoint point);鼠标左/右键放开时被调用
afx_msg void OnLButtonDblClk/OnRButtonDblClk(UINT nFlags, CPoint point);鼠标左/右键双击时被调用
afx_msg void OnMouseMove(UINT nFlags, CPoint point);鼠标在窗口上移动时被调用

4.F 关于 WM_NOTIFY 的使用方法
WM_NOTIF 在 WIN32 中得到大量的应用，同时也是随着 CommControl 的出现 WM_NOTIFY 成为了 CommControl 的基本消息。可以这样说 CommControl 的所有的新增特性都通过 WM_NOTIFY 来表达。同时 WM_NOTIFY 也为 CommControl 的操作带来了一致性。

WM_NOTIFY 消息中的参数如下：
idCtrl = (int) wParam;
pnmh = (LPNMHDR) lParam; 其中 lParam 为一个

typedef struct tagNMHDR
{
HWND hwndFrom;
UINT idFrom;
UINT code;
} NMHDR; 结构指针
从消息的参数我们已经可以分辩出消息的来源，但是这些信息还不足以分辩出消息的具体含义。所以我们需要更多的数据来得到更多的信息。MS 的做法是对每种不同用途的通知消息都定义另一种结构来表示，同时这中结构里包含了 struct tagNMHDR，所以你只要进行一下类型转换就可以得到数据指针。例如对于 LVN_COLUMNCLICK 消息（用于在 ListCtrl 的列表头有鼠标点击是进行通知），结构为：
typedef struct tagNMLISTVIEW{
NMHDR hdr;
int iItem;
int iSubItem;
UINT uNewState;
UINT uOldState;
UINT uChanged;
POINT ptAction;
LPARAM lParam;
} NMLISTVIEW, FAR *LPNMLISTVIEW;

在这个结构的最开始也就包含了 struct tagNMHDR，所以在不损失数据和产生错误的情况下向处理消息的进程提供了更多的信息。

此外通过 WM_NOTIFY 我们可以一种完全一样的方式进行消息映射，如同在前几章中所见到的一样。
使用如下形式：ON_NOTIFY(wNotifyCode, id, memberFxn)。
处理函数也有统一的原型：afx_msg void memberFxn(NMHDR * pNotifyStruct, LRESULT * result);
在 MFC 消息映射的内部将根据定义消息映射时所使用的 wNotifyCode 和 WM_NOTIFY 中参数中 pnmh->code（pnmh = (LPNMHDR) lParam）进行匹配，然后调用相应的处理函数。

还有一点是利用 WM_NOTIFY/ON_NOTIFY_REFLECT 可以在窗口内部处理一些消息，从而建立可重用的控件。大家可以参考 Build Reusable MFC Control Classes。目前我也准备在空闲时翻译这篇文章。

第五章 对话框

5.1 使用资源编辑器编辑对话框

在 Windows 开发中弹出对话框是一种常用的输入/输出手段，同时编辑好的对话框可以保存在资源文件中。Visual C++提供了对话框编辑工具，利用编辑工具可以方便的添加各种控件到对话框中，而且利用 ClassWizard 可以方便的生成新的对话框类和映射消息。

首先资源列表中按下右键，可以在弹出菜单中选择“插入对话框”，如图 1。然后再打开该对话框进行编辑，你会在屏幕上看到一个控件板，如图 2。你可以将所需要添加的控件拖到对话框上，或是先选中后再在对话框上用鼠标画出所占的区域。

接下来我们在对话框上产生一个输入框，和一个用于显示图标的图片框。之后我们使用鼠标右键单击产生的控件并选择其属性，如图 3。我们可以在属性对话框中编辑控件的属性同时也需要指定控件 ID，如图 4，如果在选择对话框本身的属性那么你可以选择对话框的一些属性，包括字体，外观，是否有系统菜单等等。最后我们编辑图片控件的属性，如图 5，我们设置控件的属性为显示图标并指明一个图标 ID。

接下来我们添加一些其他的控件，最后的效果如图 6。按下 Ctrl-T 可以测试该对话框。此外在对话框中还有一个有用的特性，就是可以利用 Tab 键让输入焦点在各个控件间移动，要达到这一点首先需要为控件设置在 Tab 键按下时可以接受焦点移动的属性 Tab Stop，如果某一个控件不打算利用这一特性，你需要清除这一属性。然后从菜单“Layout”选择 Tab Order 来确定焦点移动顺序，如图 7。使用鼠标依此点击控件就可以重新规定焦点移动次序。最后按下 Ctrl-T 进行测试。

最后我们需要为对话框产生新的类，ClassWizard 可以替我们完成大部分的工作，我们只需要填写几个参数就可以了。在编辑好的对话框上双击，然后系统回询问是否添加新的对话框，选择是并在接下来的对话框中输入类名就可以了。ClassWizard 会为你产生所需要的头文件和 CPP 文件。然后在需要使用的地方包含相应的头文件，对于有模式对话框使用 DoModal()产生，对于无模式对话框使用 Create()产生。相关代码如下；

```
void CMy51_s1View::OnCreateDlg()
{//产生无模式对话框
```

```
CTestDlg *dlg=new CTestDlg;
dlg->Create(IDD_TEST_DLG);
dlg->ShowWindow(SW_SHOW);
}

void CMy51_s1View::OnDoModal()
{//产生有模式对话框
CTestDlg dlg;
int iRet=dlg.DoModal();
TRACE("dlg return %d\n",iRet);
}
```

下载例子。如果你在调试这个程序时你会发现程序在退出后会有内存泄漏，这是因为我没有释放无模式对话框所使用的内存，这一问题会在以后的章节 5.3 创建无模式对话框中专门讲述。

关于在使用对话框时 Enter 键和 Escape 键的处理：在使用对话框是你会发现当你按下 Enter 键或 Escape 键都会退出对话框，这是因为 Enter 键会引起 CDialog::OnOK()的调用，而 Escape 键会引起 CDialog::OnCancel()的调用。而这两个调用都会引起对话框的退出。在 MFC 中这两个成员函数都是虚拟函数，所以我们需要进行重载，如果我们不希望退出对话框那么我们可以在函数中什么都不做，如果需要进行检查则可以添加检查代码，然后调用父类的 OnOK()或 OnCancel()。相关代码如下；

```
void CTestDlg::OnOK()
{
AfxMessageBox("你选择确定");
CDialog::OnOK();
}
void CTestDlg::OnCancel()
{
AfxMessageBox("你选择取消");
CDialog::OnCancel();
}
```

5.2 创建有模式对话框

使用有模式对话框时在对话框弹出后调用函数不会立即返回，而是等到对话框销毁后才会返回（请注意在对话框弹出后其他窗口的消息依然会被传递）。所以在使用对话框时其他窗口都不能接收用户输入。创建有模式对话框的方法是调用 CDialog::DoModal()。下面的代码演示了这种用法：

```
CYourView::OnOpenDlg()
{
CYourDlg dlg;
```

```
int iRet=dlg.DoModal();
}
```

CDialog::DoModal()的返回值为 IDOK，IDCANCEL。表明操作者在对话框上选择“确认”或是“取消”。由于在对话框销毁前 DoModal 不会返回，所以可以使用局部变量来引用对象。在退出函数体后对象同时也会被销毁。而对于无模式对话框则不能这样使用，下节 5.3 创建无模式对话框中会详细讲解。

你需要根据 DoModal()的返回值来决定你下一步的动作，而得到返回值也是使用有模式对话框的一个很大原因。

使用有模式对话框需要注意一些问题，比如说不要在一些反复出现的事件处理过程中生成有模式对话框，比如说在定时器中产生有模式对话框，因为在上一个对话框还未退出时，定时器消息又会引起下一个对话框的弹出。

同样的在你的对话框类中为了向调用者返回不同的值可以调用 CDialog::OnOK()或是 CDialog::OnCancel()以返回 IDOK 或 IDCANCEL，如果你希望返回其他的值，你需要调用 CDialog::EndDialog(int nResult);其中 nResult 会作为 DoModal()调用的返回值。

下面的代码演示了如何使用自己的函数来退出对话框：下载例子

```
void CMy52_s1View::OnLButtonDown(UINT nFlags, CPoint point)
{//创建对话框并得到返回值
CView::OnLButtonDown(nFlags, point);
CTestDlg dlg;
int iRet=dlg.DoModal();
CString szOut;
szOut.Format("return value %d",iRet);
AfxMessageBox(szOut);
}
//重载 OnOK,OnCancel
void CTestDlg::OnOK()
{//什么也不做
}
void CTestDlg::OnCancel()
{//什么也不做
}
//在对话框中对三个按钮消息进行映射
void CTestDlg::OnExit1()
{
CDialog::OnOK();
}
void CTestDlg::OnExit2()
{
```

```
CDialog::OnCancel();
}
void CTestDlg::OnExit3()
{
CDialog::EndDialog(0XFF);
}
```

由于重载了 OnOK 和 OnCancel 所以在对话框中按下 Enter 键或 Escape 键时都不会退出，只有按下三个按钮中的其中一个才会返回。

此外在对话框被生成是会自动调用 BOOL CDialog::OnInitDialog()，你如果需要在对话框显示前对其中的控件进行初始化，你需要重载这个函数，并在其中填入相关的初始化代码。利用 ClassWizard 可以方便的产生一些默认代码，首先打开 ClassWizard，选择相应的对话框类，在右边的消息列表中选择 WM_INITDIALOG 并双击，ClassWizard 会自动产生相关代码，代码如下：

```
BOOL CTestDlg::OnInitDialog()
{
/*先调用父类的同名函数*/
CDialog::OnInitDialog();
/*填写你的初始化代码*/
return TRUE;
}
```

有关对对话框中控件进行初始化会在 5.4 在对话框中进行消息映射中进行更详细的讲解

5.3 创建无模式对话框

无模式对话框与有模式对话框不同的是在创建后其他窗口都可以继续接收用户输入，因此无模式对话框有些类似一个弹出窗口。创建无模式对话框需要调用
BOOL CDialog::Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);之后还需要调用
BOOL CDialog::ShowWindow(SW_SHOW);进行显示，否则无模式对话框将是不可见的。相关代码如下：

```
void CYourView::OnOpenDlg(void)
{
/*假设 IDD_TEST_DLG 为已经定义的对话框资源的 ID 号*/
CTestDlg *dlg=new CTestDlg;
dlg->Create(IDD_TEST_DLG,NULL);
dlg->ShowWindows(SW_SHOW);
/*不要调用  delete dlg;*/
}
```

在上面的代码中我们新生成了一个对话框对象，而且在退出函数时并没有销毁该对象。因为如果此时销毁该对象（对象被销毁时窗口同时被销毁），而此时对话框还在显示就会出现错

误。那么这就提出了一个问题：什么时候销毁该对象。我时常使用的方法有两个：
在对话框退出时销毁自己：在对话框中重载 OnOK 与 OnCancel 在函数中调用父类的同名函数，然后调用 DestroyWindow()强制销毁窗口，在对话框中映射 WM_DESTROY 消息，在消息处理函数中调用 delete this;强行删除自身对象。相关代码如下：

```
void CTestDlg1::OnOK()
{
CDialog::OnOK();
DestroyWindow();
}

void CTestDlg1::OnCancel()
{
CDialog::OnCancel();
DestroyWindow();
}

void CTestDlg1::OnDestroy()
{
CDialog::OnDestroy();
AfxMessageBox("call delete this");
delete this;
}
```

这种方法的要点是在窗口被销毁的时候，删除自身对象。所以你可以在任何时候调用 DestroyWindow()以达到彻底销毁自身对象的作用。（DestroyWindow()的调用会引起 OnDestroy()的调用）
通过向父亲窗口发送消息，要求其他窗口对其进行销毁：首先需要定义一个消息用于进行通知，然后在对话框中映射 WM_DESTROY 消息，在消息处理函数中调用消息发送函数通知其他窗口。在接收消息的窗口中利用 ON_MESSAGE 映射处理消息的函数，并在消息处理函数中删除对话框对象。相关代码如下：

```
/*更改对话框的有关文件*/
CTestDlg2::CTestDlg2(CWnd* pParent /*=NULL*/)
: CDialog(CTestDlg2::IDD, pParent)
{/*m_pParent 为一成员变量，用于保存通知窗口的指针，所以该指针不能是一个临时指针*/
ASSERT(pParent);
m_pParent=pParent;
//{{AFX_DATA_INIT(CTestDlg2)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
}
void CTestDlg2::OnOK()
{
```

```
CDialog::OnOK();
DestroyWindow();
}

void CTestDlg2::OnCancel()
{
CDialog::OnCancel();
DestroyWindow();
}

void CTestDlg2::OnDestroy()
{
CDialog::OnDestroy();
/*向其他窗口发送消息，将自身指针作为一个参数发送*/
m_pParent->PostMessage(WM_DELETE_DLG,(WPARAM)this);
}

/*在消息接收窗口中添加消息映射*/
/*在头文件中添加函数定义*/
afx_msg LONG OnDelDlgMsg(WPARAM wP,LPARAM lP);
/*添加消息映射代码*/
ON_MESSAGE(WM_DELETE_DLG,OnDelDlgMsg)
END_MESSAGE_MAP()
/*实现消息处理函数*/
LONG CMy53_s1View::OnDelDlgMsg(WPARAM wP,LPARAM lP)
{
delete (CTestDlg2*)wP;
return 0;
}
/*创建对话框*/
void CMy53_s1View::OnTest2()
{
CTestDlg2 *dlg=new CTestDlg2(this);
dlg->Create(IDD_TEST_DLG_2);
dlg->ShowWindow(SW_SHOW);
}
```

在这种方法中我们利用消息来进行通知，在 Window 系统中利用消息进行通知和传递数据的用法是很多的。

同样无模式对话框的另一个作用还可以用来在用户在对话框中的输入改变时可以及时的反映到其他窗口。下面的代码演示了在对话框中输入一段文字，然后将其更新到视图的显示区域中，这同样也是利用了消息进行通知和数据传递。

```
/*在对话框中取出数据，并向其他窗口发送消息和数据，将数据指针作为一个参数发送*/
void CTestDlg2::OnCommBtn()
{
char szOut[30];
GetDlgItemText(IDC_OUT,szOut,30);
m_pParent->SendMessage(WM_DLG_NOTIFY,(WPARAM)szOut);
}

/*在消息接收窗口中*/
/*映射消息处理函数*/
ON_MESSAGE(WM_DLG_NOTIFY,OnDlgNotifyMsg)

/*在视图中绘制出字符串  m_szOut*/
void CMy53_s1View::OnDraw(CDC* pDC)
{
CMy53_s1Doc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
pDC->TextOut(0,0,"Display String");
pDC->TextOut(0,20,m_szOut);
}
/*处理通知消息，保存信息并更新显示*/
LONG CMy53_s1View::OnDlgNotifyMsg(WPARAM wP,LPARAM lP)
{
m_szOut=(char*)wP;
Invalidate();
return 0;
}
```

此外这种用法利用消息传递数据的方法对有模式对话框和其他的窗口间通信也一样有效。下载本节例子

5.4 在对话框中进行消息映射

利用对话框的一个好处是可以利用 ClassWizard 对对话框中各个控件产生的消息进行映射，ClassWizrd 可以列出各种控件可以使用的消息，并能自动产生代码。在本节中我们以一个例子来讲解如何在对话框中对子窗口消息进行映射同时还讲解如何对对话框中的子窗口进行初始化。

ID	类型
IDC_RADIO_TEST_1	圆形按钮
IDC_RADIO_TEST_2	圆形按钮
IDC_BUTTON_TEST	按钮

IDC_CHECK_TEST	检查按钮
IDC_TREE_TEST	树形控件
IDC_LIST_CTRL	List Ctrl
IDC_TAB_CTRL	Tab Ctrl
IDC_LIST_TEST	列表框
IDC_COMBO_TEST	组合框
IDC_EDIT_TEST	输入框

首先我们产生编辑好一个对话框，如图，在对话框中使用的控件和 ID 号如下表：

首先我们需要在对话框的 OnInitDialog()函数中对各个控件进行初始化，这里我们使用 CWnd* GetDlgItem(int nID)来通过 ID 号得到子窗口指针。（类似的函数还有 UINT GetDlgItemInt(int nID, BOOL* lpTrans = NULL, BOOL bSigned = TRUE) 通过 ID 号得到子窗口中输入的数字，int GetDlgItemText(int nID, CString& rString) 通过 ID 号得到子窗口中输入的文字）。代码如下：

```
BOOL CMy54_s1Dlg::OnInitDialog()
{
CDialog::OnInitDialog();
/*添加初始化代码*/
//初始化输入框
((CEdit*)GetDlgItem(IDC_EDIT_TEST))->SetWindowText("this is a edit box");
//初始化列表框
CListBox* pListB=(CListBox*)GetDlgItem(IDC_LIST_TEST);
pListB->AddString("item 1");
pListB->AddString("item 2");
pListB->AddString("item 3");
//初始化组合框
CComboBox* pCB=(CComboBox*)GetDlgItem(IDC_COMBO_TEST);
pCB->AddString("item 1");
pCB->AddString("item 2");
pCB->AddString("item 3");
//初始化 Tab Ctrl
CTabCtrl* pTab=(CTabCtrl*)GetDlgItem(IDC_TAB_TEST);
pTab->InsertItem(0,"Tab Page1");
pTab->InsertItem(1,"Tab Page2");
```

```
pTab->InsertItem(2,"Tab Page3");
//初始化 ListCtrl
CListCtrl* pList=(CListCtrl*)GetDlgItem(IDC_LIST_CTRL);
pList->InsertColumn(0,"Column 1",LVCFMT_LEFT,100);
pList->InsertItem(0,"Item 1");
pList->InsertItem(1,"Item 2");
pList->InsertItem(2,"Item 3");
//初始化 TreeCtrl
CTreeCtrl* pTree=(CTreeCtrl*)GetDlgItem(IDC_TREE_TEST);
pTree->InsertItem("Node1",0,0);
HTREEITEM hNode=pTree->InsertItem("Node2",0,0);
pTree->InsertItem("Node2-1",0,0,hNode);
pTree->InsertItem("Node2-2",0,0,hNode);
pTree->Expand(hNode,TVE_EXPAND);

return TRUE; // return TRUE unless you set the focus to a control
}
```

接下来我们需要利用 ClassWizard 对控件所产生的消息进行映射，打开 ClassWizard 对话框，选中相关控件的 ID，在右边的列表中就会显示出可用的消息。如我们对按钮的消息进行映射，在选中按钮 ID（IDC_BUTTON_TEST）后，会看到两个消息，如图，一个是 BN_CLICKED，一个是 BN_DOUBLECLICKED。双击 BN_CLICKED 后在弹出的对话框中输入函数名，ClassWizard 会产生按钮被按的消息映射。

然后我们看看对 TabCtrl 的 TCN_SELCHANGE 消息进行映射，如图，在 TabCtrl 的当前页选择发生改变时这个消息会被发送，所以通过映射该消息可以在当前页改变时及时得到通知。

最后我们对输入框的 EN_CHANGE 消息进行映射，如图，在输入框中的文本改变后该消息会被发送。相关的代码如下：

```
//头文件中相关的消息处理函数定义
afx_msg void OnButtonTest();
afx_msg void OnSelchangeTabTest(NMHDR* pNMHDR, LRESULT* pResult);
afx_msg void OnChangeEditTest();
//}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()

//CPP 文件中消息映射代码
ON_BN_CLICKED(IDC_BUTTON_TEST, OnButtonTest)
ON_NOTIFY(TCN_SELCHANGE, IDC_TAB_TEST, OnSelchangeTabTest)
ON_EN_CHANGE(IDC_EDIT_TEST, OnChangeEditTest)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

//消息处理函数
void CMy54_s1Dlg::OnButtonTest()
{
AfxMessageBox("you pressed a button");
}

void CMy54_s1Dlg::OnSelchangeTabTest(NMHDR* pNMHDR, LRESULT* pResult)
{
TRACE("Tab Select changed\n");
*pResult = 0;
}

void CMy54_s1Dlg::OnChangeEditTest()
{
TRACE("edit_box text changed\n");
}
```

对于其他的控件都可以采取类似的方法进行消息映射，下载例子。此外如果你对各种控件可以使用的消息不熟悉，你可以通过使用对话框，然后利用 ClassWizard 产生相关代码的方法来进行学习，你也可以将 ClassWizard 产生的代码直接拷贝到其他需要的地方（不瞒你说，我最开始就是这样学的 :-D 这也算一个小窍门）。

5.5 在对话框中进行数据交换和数据检查

MFC 提供两种方法在对话框中进行数据交换和数据检查（Dialog data exchange/Dialog data validation），数据交换和数据检查的思想是将某一变量和对话框中的一个子窗口进行关联，然后通过调用 BOOL UpdateData(BOOL bSaveAndValidate = TRUE)来指示 MFC 将变量中数据放入子窗口还是将子窗口中数据取到变量中并进行合法性检查。

在进行数据交换时一个子窗口可以和两种类型的变量相关联，一种是控件（Control）对象，比如说按钮子窗口可以和一个 CButton 对象相关联，这种情况下你可以通过该对象直接控制子窗口，而不需要象上节中讲的一样使用 GetDlgItem(IDC_CONTROL_ID)来得到窗口指针；一种是内容对象，比如说输入框可以和一个 CString 对象关联，也可以和一个 UINT 类型变量关联，这种情况下你可以直接设置/获取窗口中的输入内容。

而数据检查是在一个子窗口和一个内容对象相关联时在存取内容时对内容进行合法性检查，比如说当一个输入框和一个 CString 对象关联时，你可以设置检查 CString 的对象的最长/最小长度，当输入框和一个 UINT 变量相关联时你可以设置检查 UINT 变量的最大/最小值。在 BOOL UpdateData(BOOL bSaveAndValidate = TRUE)被调用后，合法性检查会自动进行，如果无法通过检查 MFC 会弹出消息框进行提示，并返回 FALSE。

设置 DDX/DDV 在 VC 中非常简单，ClassWizard 可以替你完成所有的工作，你只需要打开 ClassWizard 并选中 Member Variables 页，如图，你可以看到所有可以进行关联的子窗口 ID 列表，双击一个 ID 会弹出一个添加变量的对话框，如图，填写相关的信息后按下确定按钮就可以了。然后选中你刚才添加的变量在底部的输入框中输入检查条件，如图。

下面我们看一个例子，对话框上的子窗口如图设置，各子窗口的 ID 和关联的变量如下表：
ID 关联的变量 作用
IDC_CHECK_TEST BOOL m_fCheck 检查框是否被选中
IDC_RADOI_TEST_1 int m_iSel 当前选择的圆形按钮的索引
IDC_COMBO_TEST CString m_szCombo 组合框中选中的文本或是输入的文本
IDC_EDIT_TEST CString m_szEdit 输入框中输入的文本，最大长度为 5
IDC_LIST_TEST CListBox m_lbTest 列表框对象

这时候 ClassWizard 会自动生成变量定义和相关代码，在对话框的构造函数中可以对变量的初始值进行设置，此外在 BOOL CDialog::OnInitDialog()中会调用 UpdateData(FALSE)，即会将变量中的数据放入窗口中 。相关代码如下：

```
//头文件中的变量定义，ClassWizard 自动产生
// Dialog Data
//{{AFX_DATA(CMy55_s1Dlg)
enum { IDD = IDD_MY55_S1_DIALOG };
CListBox m_lbTest;
int m_iSel;
CString m_szEdit;
CString m_szCombo;
BOOL m_fCheck;
//}}AFX_DATA
//构造函数中赋初值
CMy55_s1Dlg::CMy55_s1Dlg(CWnd* pParent /*=NULL*/)
: CDialog(CMy55_s1Dlg::IDD, pParent)
{
```

```
//{{AFX_DATA_INIT(CMy55_s1Dlg)
m_iSel = -1;
m_szEdit = _T("");
m_szCombo = _T("");
m_fCheck = FALSE;
//}}AFX_DATA_INIT
.....
}
//ClassWizard 产生的关联和检查代码
void CMy55_s1Dlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CMy55_s1Dlg)
DDX_Control(pDX, IDC_LIST_TEST, m_lbTest);
DDX_Radio(pDX, IDC_RADIO_TEST_1, m_iSel);
DDX_Text(pDX, IDC_EDIT_TEST, m_szEdit);
DDV_MaxChars(pDX, m_szEdit, 5);
DDX_CBString(pDX, IDC_COMBO_TEST, m_szCombo);
DDX_Check(pDX, IDC_CHECK_TEST, m_fCheck);
//}}AFX_DATA_MAP
}
//在 OnInitDialog 中利用已经关联过的变量 m_lbTest
BOOL CMy55_s1Dlg::OnInitDialog()
{
CDialog::OnInitDialog();
...
// TODO: Add extra initialization here
//设置列表框中数据
m_lbTest.AddString("String 1");
m_lbTest.AddString("String 2");
m_lbTest.AddString("String 3");
m_lbTest.AddString("String 4");
return TRUE; // return TRUE unless you set the focus to a control
}
//对两个按钮消息处理
//通过 UpdateData(TRUE)得到窗口中数据
void CMy55_s1Dlg::OnGet()
{
if(UpdateData(TRUE))
{
//数据合法性检查通过，可以使用变量中存放的数据
CString szOut;
szOut.Format("radio =%d \ncheck is %d\nedit input is %s\ncomboBox input is %s\n",
m_iSel,m_fCheck,m_szEdit,m_szCombo);
```

```
AfxMessageBox(szOut);
}
//else 未通过检查
}
//通过 UpdateData(FALSE)将数据放入窗口
void CMy55_s1Dlg::OnPut()
{
m_szEdit="onPut test";
m_szCombo="onPut test";
UpdateData(FALSE);
}
```

在上面的例子中我们看到利用 DDX/DDV 和 UpdateData(BOOL)调用我们可以很方便的存取数据，而且也可以同时进行合法性检查。下载例子代码

5.6 使用属性对话框

属性对话框不同于普通对话框的是它能同时提供多个选项页，而每页都可以由资源编辑器以编辑对话框的方式进行编辑，这样给界面开发带来了方便。同时使用上也遵守普通对话框的规则，所以学习起来很方便。属性对话框由两部分构成：多个属性页（CPropertyPage）和属性对话框（CPropertySheet）。

首先需要编辑属性页，在资源编辑器中选择插入，并且选择属性对话框后就可以插入一个属性页，如图，或者选择插入一个对话框，然后将其属性中的 Style 设置为 Child，Border 设置为 Thin 也可以，如图。然后根据这个对话框资源生成一个新类，在选择基类时选择 CPropertyPage，ClassWizard 会自动生成相关的代码。

而对于 CPropertySheet 也需要生成新类，并且将所有需要加入的属性页对象都作为成员变量。属性对话框也分为有模式和无模式两种，有模式属性对话框使用 DoModal()创建，无模式属性对话框使用 Create()创建。下面的代码演示了如何创建属性对话框并添加属性页：

```
//修改 CPropertySheet 派生类的构造函数为如下形式
CSheet::CSheet()
:CPropertySheet("test sheet", NULL, 0)
{
m_page1.Construct(IDD_PAGE_1);
m_page2.Construct(IDD_PAGE_2);
AddPage(&m_page1);
AddPage(&m_page2);
}
//创建有模式属性对话框
void CMy56_s1Dlg::OnMod()
{
CSheet sheet;
sheet.DoModal();
}
```

```
//创建无模式属性对话框
void CMy56_s1Dlg::OnUnm()
{
CSheet *sheet=new CSheet;
sheet->Create();
}
```

对于属性对话框可以使用下面的一些成员函数：

CPropertyPage* CPropertySheet::GetActivePage( )得到当前活动页的指针。
BOOL CPropertySheet::SetActivePage(int nPage)用于设置当前活动页。
int CPropertySheet::GetPageCount()用于得到当前页总数。
void CPropertySheet::RemovePage(int nPage)用于删除一页。
而对于属性页来将主要通过重载一些函数来达到控制的目的：
void CPropertyPage::OnOK() 在属性对话框上按下“确定”按钮后被调用
void CPropertyPage::OnCancel() 在属性对话框上按下“取消”按钮后被调用
void CPropertyPage::OnApply() 在属性对话框上按下“应用”按钮后被调用
void CPropertyPage::SetModified(BOOL bChanged = TRUE) 设置当前页面上的数据被修改标记，这个调用可以使“应用”按钮为允许状态。
此外利用属性对话框你可以生成向导对话框，向导对话框同样拥有多个属性页，但同时只有一页被显示，而且对话框上显示的按钮为“上一步”，“下一步”/“完成”，向导对话框会按照你添加页面的顺序依次显示所有的页。在显示属性对话框前你需要调用 void CPropertySheet::SetWizardMode() 。使用向导对话框时需要对属性页的 BOOL CPropertyPage::OnSetActive( ) 进行重载，并在其中调用 void CPropertySheet::SetWizardButtons(DWORD dwFlags)来设置向导对话框上显示的按钮。
dwFlags 的取值可为以下值的“或”操作：
PSWIZB_BACK 显示“上一步”按钮
PSWIZB_NEXT 显示“下一步”按钮
PSWIZB_FINISH 显示“完成”按钮
PSWIZB_DISABLEDFINISH 显示禁止的“完成”按钮
void CPropertySheet::SetWizardButtons(DWORD dwFlags)也可以在其他地方调用，比如说在显示最后一页时先显示禁止的“完成”按钮，在完成某些操作后再显示允许的“完成”按钮。

在使用向导对话框时可以通过重载一些函数来达到控制的目的：

void CPropertyPage::OnWizardBack() 按下了“上一步”按钮。返回 0 表示有系统决定需要显示的页面，-1 表示禁止页面转换，如果希望显示一个特定的页面需要返回该页面的 ID 号。
void CPropertyPage::OnOnWizardNext() 按下了“下一步”按钮。返回值含义与 void CPropertyPage::OnWizardBack()相同。
void CPropertyPage::OnWizardFinish() 按下了“完成”按钮。返回 FALSE 表示不允许继续，否则返回 TRUE 向导对话框将被结束。
在向导对话框的 DoModal()返回值为 ID_WIZFINISH 或 IDCANCEL。下面的代码演示了如何创建并使用向导对话框：

```
//创建有模式向导对话框
void CMy56_s1Dlg::OnWiz()
{
CSheet sheet;
sheet.SetWizardMode();
int iRet=sheet.DoModal();//返回 ID_WIZFINISH 或 IDCANCEL
}
//重载 BOOL CPropertyPage::OnSetActive( )来控制显示的按钮
BOOL CPage1::OnSetActive()
{
((CPropertySheet*)GetParent())->SetWizardButtons(PSWIZB_BACK|PSWIZB_NEXT);
return CPropertyPage::OnSetActive();
}
BOOL CPage2::OnSetActive()
{
((CPropertySheet*)GetParent())->SetWizardButtons(PSWIZB_BACK|PSWIZB_FINISH);
return CPropertyPage::OnSetActive();
}
```

下载本节例子。

5.7 使用通用对话框

在 Windows 系统中提供了一些通用对话框如：文件选择对话框如图，颜色选择对话框如图，字体选择对话框如图。在 MFC 中使用 CFileDialog，CColorDialog，CFontDialog 来表示。一般来讲你不需要派生新的类，因为基类已经提供了常用的功能。而且在创建并等待对话框结束后你可以通过成员函数得到用户在对话框中的选择。

CFileDialog 文件选择对话框的使用：首先构造一个对象并提供相应的参数，构造函数原型如下：

CFileDialog::CFileDialog(BOOL bOpenFileDialog, LPCTSTR lpszDefExt = NULL, LPCTSTR lpszFileName = NULL, DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, LPCTSTR lpszFilter = NULL, CWnd* pParentWnd = NULL);参数意义如下：

bOpenFileDialog 为 TRUE 则显示打开对话框，为 FALSE 则显示保存对话文件对话框。

lpszDefExt 指定默认的文件扩展名。

lpszFileName 指定默认的文件名。

dwFlags 指明一些特定风格。

lpszFilter 是最重要的一个参数，它指明可供选择的文件类型和相应的扩展名。参数格式如：

"Chart Files (*.xlc)|*.xlc|Worksheet Files (*.xls)|*.xls|Data Files (*.xlc;*.xls)|*.xlc; *.xls|All Files (*.*)|*.*||";文件类型说明和扩展名间用 | 分隔，同种类型文件的扩展名间可以用 ; 分割，每种文件类型间用 | 分隔，末尾用 || 指明。

pParentWnd 为父窗口指针。

创建文件对话框可以使用 DoModal()，在返回后可以利用下面的函数得到用户选择：
CString CFileDialog::GetPathName( ) 得到完整的文件名，包括目录名和扩展名如：c:\test\test1.txt

CString CFileDialog::GetFileName( ) 得到完整的文件名，包括扩展名如：test1.txt

CString CFileDialog::GetExtName( ) 得到完整的文件扩展名，如：txt

CString CFileDialog::GetFileTitle ( ) 得到完整的文件名，不包括目录名和扩展名如：test1

POSITION CFileDialog::GetStartPosition( ) 对于选择了多个文件的情况得到第一个文件位置。

CString CFileDialog::GetNextPathName(POSITION& pos) 对于选择了多个文件的情况得到下一个文件位置，并同时返回当前文件名。但必须已经调用过 POSITION CFileDialog::GetStartPosition( )来得到最初的 POSITION 变量。

CColorDialog 颜色选择对话框的使用：首先通过 CColorDialog::CColorDialog(COLORREF clrInit = 0, DWORD dwFlags = 0, CWnd* pParentWnd = NULL)构造一个对象，其中 clrInit 为初始颜色。通过调用 DoModal() 创建对话框，在返回后调用 COLORREF CColorDialog::GetColor( )得到用户选择的颜色值。

CFontDialog 字体选择对话框的使用：首先构造一个对象并提供相应的参数，构造函数原型如下：
CFontDialog::CFontDialog(LPLOGFONT lplfInitial = NULL, DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS, CDC* pdcPrinter = NULL, CWnd* pParentWnd = NULL);构造一个对象，其中参数 lplfInitial 指向一个 LOGFONG 结构（该结构介绍请见 2.2 在窗口中输出文字），如果该参数设置为 NULL 表示不设置初始字体。pdcPrinter 指向一个代表打印机设备环境的 DC 对象，若设置该参数则选择的字体就为打印机所用。pParentWnd 用于指定父窗口。通过调用 DoModal()创建对话框，在返回后通过调用以下函数来得到用户选择：

void CFontDialog::GetCurrentFont(LPLOGFONT lplf);用来获得所选字体的属性。该函数有一个参数，该参数是指向 LOGFONT 结构的指针，函数将所选字体的各种属性写入这个 LOGFONT 结构中。

CString CFontDialog::GetFaceName( ) 得到所选字体名字。

int CFontDialog::GetSize( ) 得到所选字体的尺寸（以 10 个象素为单位）。

COLORREF CFontDialog::GetColor( ) 得到所选字体的颜色。

BOOL CFontDialog::IsStrikeOut( )

BOOL CFontDialog::IsUnderline( )

BOOL CFontDialog::IsBold( )
BOOL CFontDialog::IsItalic( )
得到所选字体的其他属性，是否有删除线，是否有下划线，是否为粗体，是否为斜体

5.8 建立以对话框为基础的应用

我认为初学者使用以对话框为基础的应用是一个比较好的选择，因为这样一来可以摆脱一些开发界面的麻烦，此外也可以利用 ClassWizard 自动的添加消息映射。

在 VC 中提供了生成“以对话框为基础的应用”的功能，你所需要选择的是在使用 AppWizard 的第一步选择“对话框为基础的应用”，如图。VC 会生成包含有应用派生类和对话框派生类的代码。在应用类的 InitInstance()成员函数中可以看到如下的代码：

```
BOOL CMy58_s1App::InitInstance()
{
CMy58_s1Dlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: Place code here to handle when the dialog is
// dismissed with OK
}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
// dismissed with Cancel
}

return FALSE;
}
```

这是产生一个有模式对话框并创建它，在对话框返回后通过返回 FALSE 来直接退出。在设计时通过编辑对话框资源你可以设计好界面，然后通过 ClassWizard 映射消息来处理客户的输入，由于前几节已经讲过本节也就不再重复。

同样基于对话框的应用也同样可以使用属性对话框做为界面，或者是通过使用经过派生的通

用对话框作为界面。

提示：当你使用有模式对话框时最开始是无法隐藏窗口的，而只能在对话框显示后再隐藏窗口，所以这会造成屏幕的闪烁。一个解决办法就是采用无模式的对话框，无模式的对话框在创建后是隐藏的，直到你调用 ShowWindow(SW_SHOW)才会显示。相关代码如下：

```
BOOL CMy58_s1App::InitInstance()
{
//必须新生成一个对象，而不能使用局部变量
CMy58_s1Dlg* pdlg=new CMy58_s1Dlg;
m_pMainWnd = pdlg;
pdlg->Create();
return TRUE;
}
```

5.9 使用对话框作为子窗口

使用对话框作为子窗口是一种很常用的技术，这样可以使界面设计简化而且修改起来更加容易。

简单的说这种技术的关键就在于创建一个无模式的对话框，并在编辑对话框资源时指明 Child 风格和无边框风格，如图。接下来利用产生一个 CDialog 的派生类，并进行相关的消息映射。在创建子窗口时需要利用下面的代码：

```
int CMy59_s1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
return -1;

//创建子窗口
m_dlgChild.Create(IDD_CHILD_DLG,this);
//重新定位
m_dlgChild.MoveWindow(0,0,400,200);
//显示窗口
m_dlgChild.ShowWindow(SW_SHOW);
return 0;
}
```

此外还有一中类似的技术是利用 CFormView 派生类作为子窗口，在编辑对话框资源时也需要指明 Child 风格和无边框风格。然后利用 ClassWizard 产生以 CFormView 为基类的派生类，但是由于该类的成员函数都是受保护的，所以需要对产生的头文件进行如下修改：

```
class CTestForm : public CFormView
{
//将构造函数和构析函数改为共有函数
public:
```

```
CTestForm();
virtual ~CTestForm();
DECLARE_DYNCREATE(CTestForm)
...
}
```

有关创建子窗口的代码如下：

```
int CMy59_s1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
return -1;

//对于 CFormView 派生类必须新生成对象而不能使用成员对象
m_pformChild = new CTestForm;
//由于 CFormView 的成员受保护，所以必须对指针进行强制转换
CWnd* pWnd=m_pformChild;
pWnd->Create(NULL,NULL,WS_CHILD|WS_VISIBLE,CRect(0,210,400,400),this,1001,NULL
);
return 0;
}
```

最后你会看到如图的窗口界面，上方的对话框子窗口和下方的 FormView 子窗口都可以通过资源编辑器预先编辑好，下载代码。

提示：对于 CFormView 派生类必须新生成对象而不能使用成员对象，因为在 CView 的 OnDestroy()中会有如下代码：delete this;所以使用成员对象的结果会造成对象的二次删除而引发异常。

第六章 网络通信开发

6.1 WinSock 介绍

Windows 下网络编程的规范－Windows Sockets 是 Windows 下得到广泛应用的、开放的、支持多种协议的网络编程接口。从 1991 年的 1.0 版到 1995 年的 2.0.8 版，经过不断完善并在 Intel、Microsoft、Sun、SGI、Informix、Novell 等公司的全力支持下，已成为 Windows 网络编程的事实上的标准。

Windows Sockets 规范以 U.C. Berkeley 大学 BSD UNIX 中流行的 Socket 接口为范例定义了一套 Micosoft Windows 下网络编程接口。它不仅包含了人们所熟悉的 Berkeley Socket 风格的库函数；也包含了一组针对 Windows 的扩展库函数，以使程序员能充分地利用 Windows 消息驱动机制进行编程。Windows Sockets 规范本意在于提供给应用程序开发者一套简单的 API，并让各家网络软件供应商共同遵守。此外，在一个特定版本 Windows 的基础上，Windows Sockets 也定义了一个二进制接口（ABI），以此来保证应用 Windows Sockets API 的应用程

序能够在任何网络软件供应商的符合 Windows Sockets 协议的实现上工作。因此这份规范定义了应用程序开发者能够使用，并且网络软件供应商能够实现的一套库函数调用和相关语义。遵守这套 Windows Sockets 规范的网络软件，我们称之为 Windows Sockets 兼容的，而 Windows Sockets 兼容实现的提供者，我们称之为 Windows Sockets 提供者。一个网络软件供应商必须百分之百地实现 Windows Sockets 规范才能做到现 Windows Sockets 兼容。任何能够与 Windows Sockets 兼容实现协同工作的应用程序就被认为是具有 Windows Sockets 接口。我们称这种应用程序为 Windows Sockets 应用程序。Windows Sockets 规范定义并记录了如何使用 API 与 Internet 协议族（IPS，通常我们指的是 TCP/IP）连接，尤其要指出的是所有的 Windows Sockets 实现都支持流套接口和数据报套接口.应用程序调用 Windows Sockets 的 API 实现相互之间的通讯。Windows Sockets 又利用下层的网络通讯协议功能和操作系统调用实现实际的通讯工作。它们之间的关系如图

通信的基础是套接口（Socket），一个套接口是通讯的一端。在这一端上你可以找到与其对应的一个名字。一个正在被使用的套接口都有它的类型和与其相关的进程。套接口存在于通讯域中。通讯域是为了处理一般的线程通过套接口通讯而引进的一种抽象概念。套接口通常和同一个域中的套接口交换数据（数据交换也可能穿越域的界限，但这时一定要执行某种解释程序）。Windows Sockets 规范支持单一的通讯域，即 Internet 域。各种进程使用这个域互相之间用 Internet 协议族来进行通讯（Windows Sockets 1.1 以上的版本支持其他的域，例如 Windows Sockets 2）。套接口可以根据通讯性质分类；这种性质对于用户是可见的。应用程序一般仅在同一类的套接口间通讯。不过只要底层的通讯协议允许，不同类型的套接口间也照样可以通讯。用户目前可以使用两种套接口，即流套接口和数据报套接口。流套接口提供了双向的，有序的，无重复并且无记录边界的数据流服务。数据报套接口支持双向的数据流，但并不保证是可靠，有序，无重复的。也就是说，一个从数据报套接口接收信息的进程有可能发现信息重复了，或者和发出时的顺序不同。数据报套接口的一个重要特点是它保留了记录边界。对于这一特点，数据报套接口采用了与现在许多包交换网络（例如以太网）非常类似的模型。

一个在建立分布式应用时最常用的范例便是客户机/服务器模型。在这种方案中客户应用程序向服务器程序请求服务。这种方式隐含了在建立客户机/服务器间通讯时的非对称性。客户机/服务器模型工作时要求有一套为客户机和服务器所共识的惯例来保证服务能够被提供（或被接受）。这一套惯例包含了一套协议。它必须在通讯的两头都被实现。根据不同的实际情况，协议可能是对称的或是非对称的。在对称的协议中，每一方都有可能扮演主从角色；在非对称协议中，一方被不可改变地认为是主机，而另一方则是从机。一个对称协议的例子是 Internet 中用于终端仿真的 TELNET。而非对称协议的例子是 Internet 中的 FTP。无论具体的协议是对称的或是非对称的，当服务被提供时必然存在"客户进程"和"服务进程"。一个服务程序通常在一个众所周知的地址监听对服务的请求，也就是说，服务进程一直处于休眠状态，直到一个客户对这个服务的地址提出了连接请求。在这个时刻，服务程序被"惊醒"并且为客户提供服务－对客户的请求作出适当的反应。这一请求/相应的过程可以简单的用图表示。虽然基于连接的服务是设计客户机/服务器应用程序时的标准，但有些服务也是可以通过数据报套接口提供的。

数据报套接口可以用来向许多系统支持的网络发送广播数据包。要实现这种功能，网络本身必须支持广播功能，因为系统软件并不提供对广播功能的任何模拟。广播信息将会给网络造成极重的负担，因为它们要求网络上的每台主机都为它们服务，所以发送广播数据包的能力

被限制于那些用显式标记了允许广播的套接口中。广播通常是为了如下两个原因而使用的：1. 一个应用程序希望在本地网络中找到一个资源，而应用程序对该资源的地址又没有任何先验的知识。2. 一些重要的功能，例如路由要求把它们的信息发送给所有可以找到的邻机。被广播信息的目的地址取决于这一信息将在何种网络上广播。Internet 域中支持一个速记地址用于广播－INADDR_BROADCAST。由于使用广播以前必须捆绑一个数据报套接口，所以所有收到的广播消息都带有发送者的地址和端口。

Intel 处理器的字节顺序是和 DEC VAX 处理器的字节顺序一致的。因此它与 68000 型处理器以及 Internet 的顺序是不同的，所以用户在使用时要特别小心以保证正确的顺序。任何从 Windows Sockets 函数对 IP 地址和端口号的引用和传送给 Windows Sockets 函数的 IP 地址和端口号均是按照网络顺序组织的，这也包括了 sockaddr_in 结构这一数据类型中的 IP 地址域和端口域（但不包括 sin_family 域）。考虑到一个应用程序通常用与"时间"服务对应的端口来和服务器连接，而服务器提供某种机制来通知用户使用另一端口。因此 getservbyname()函数返回的端口号已经是网络顺序了，可以直接用来组成一个地址，而不需要进行转换。然而如果用户输入一个数，而且指定使用这一端口号，应用程序则必须在使用它建立地址以前，把它从主机顺序转换成网络顺序（使用 htons()函数）。相应地，如果应用程序希望显示包含于某一地址中的端口号（例如从 getpeername()函数中返回的），这一端口号就必须在被显示前从网络顺序转换到主机顺序（使用 ntohs()函数）。由于 Intel 处理器和 Internet 的字节顺序是不同的，上述的转换是无法避免的，应用程序的编写者应该使用作为 Windows Sockets API 一部分的标准的转换函数，而不要使用自己的转换函数代码。因为将来的 Windows Sockets 实现有可能在主机字节顺序与网络字节顺序相同的机器上运行。因此只有使用标准的转换函数的应用程序是可移植的。

在 MFC 中 MS 为套接口提供了相应的类 CAsyncSocket 和 CSocket，CAsyncSocket 提供基于异步通信的套接口封装功能，CSocket 则是由 CAsyncSocket 派生，提供更加高层次的功能，例如可以将套接口上发送和接收的数据和一个文件对象（CSocketFile）关联起来，通过读写文件来达到发送和接收数据的目的，此外 CSocket 提供的通信为同步通信，数据未接收到或是未发送完之前调用不会返回。此外通过 MFC 类开发者可以不考虑网络字节顺序和忽略掉更多的通信细节。

在一次网络通信/连接中有以下几个参数需要被设置：本地 IP 地址 - 本地端口号 - 对方端口号 - 对方 IP 地址。左边两部分称为一个半关联，当与右边两部分建立连接后就称为一个全关联。在这个全关联的套接口上可以双向的交换数据。如果是使用无连接的通信则只需要建立一个半关联，在发送和接收时指明另一半的参数就可以了，所以可以说无连接的通信是将数据发送到另一台主机的指定端口。此外不论是有连接还是无连接的通信都不需要双方的端口号相同。
在创建 CAsyncSocket 对象时通过调用
BOOL CAsyncSocket::Create(UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE, LPCTSTR lpszSocketAddress = NULL)通过指明 lEvent 所包含的标记来确定需要异步处理的事件，对于指明的相关事件的相关函数调用都不需要等待完成后才返回，函数会马上返回然后在完成任务后发送事件通知，并利用重载以下成员函数来处理各种网络事件：
标记 事件 需要重载的函数
FD_READ 有数据到达时发生 void OnReceive(int nErrorCode);

FD_WRITE 有数据发送时产生 void OnSend(int nErrorCode);
FD_OOB 收到外带数据时发生 void OnOutOfBandData(int nErrorCode);
FD_ACCEPT 作为服务端等待连接成功时发生 void OnAccept(int nErrorCode);
FD_CONNECT 作为客户端连接成功时发生 void OnConnect(int nErrorCode);
FD_CLOSE 套接口关闭时发生 void OnClose(int nErrorCode);

我们看到重载的函数中都有一个参数 nErrorCode，为零则表示正常完成，非零则表示错误。通过 int CAsyncSocket::GetLastError()可以得到错误值。

下面我们看看套接口类所提供的一些功能，通过这些功能我们可以方便的建立网络连接和发送数据。

BOOL CAsyncSocket::Create(UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, long lEvent = FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE, LPCTSTR lpszSocketAddress = NULL);用于创建一个本地套接口，其中 nSocketPort 为使用的端口号，为零则表示由系统自动选择，通常在客户端都使用这个选择。nSocketType 为使用的协议族，SOCK_STREAM 表明使用有连接的服务，SOCK_DGRAM 表明使用无连接的数据报服务。lpszSocketAddress 为本地的 IP 地址，可以使用点分法表示如 10.1.1.3。
BOOL CAsyncSocket::Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress = NULL)作为等待连接方时产生一个网络半关联，或者是使用 UDP 协议时产生一个网络半关联。
BOOL CAsyncSocket::Listen(int nConnectionBacklog = 5)作为等待连接方时指明同时可以接受的连接数，请注意不是总共可以接受的连接数。
BOOL CAsyncSocket::Accept(CAsyncSocket& rConnectedSocket, SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL)作为等待连接方将等待连接建立，当连接建立后一个新的套接口将被创建，该套接口将会被用于通信。
BOOL CAsyncSocket::Connect(LPCTSTR lpszHostAddress, UINT nHostPort);作为连接方发起与等待连接方的连接，需要指明对方的 IP 地址和端口号。
void CAsyncSocket::Close( );关闭套接口。
int CAsyncSocket::Send(const void* lpBuf, int nBufLen, int nFlags = 0)
int CAsyncSocket::Receive(void* lpBuf, int nBufLen, int nFlags = 0);在建立连接后发送和接收数据，nFlags 为标记位，双方需要指明相同的标记。
int CAsyncSocket::SendTo(const void* lpBuf, int nBufLen, UINT nHostPort, LPCTSTR lpszHostAddress = NULL, int nFlags = 0)
int CAsyncSocket::ReceiveFrom(void* lpBuf, int nBufLen, CString& rSocketAddress, UINT& rSocketPort, int nFlags = 0);对于无连接通信发送和接收数据，需要指明对方的 IP 地址和端口号，nFlags 为标记位，双方需要指明相同的标记。
我们可以看到大多数的函数都返回一个布尔值表明是否成功。如果发生错误可以通过 int CAsyncSocket::GetLastError()得到错误值。

由于 CSocket 由 CAsyncSocket 派生所以拥有 CAsyncSocket 的所有功能，此外你可以通过 BOOL CSocket::Create(UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, LPCTSTR lpszSocketAddress = NULL)来创建套接口，这样创建的套接口没有办法异步处理事件，所有的调用都必需完成后才会返回。

在上面的介绍中我们看到 MFC 提供的套接口类屏蔽了大多数的细节，我们只需要做很少的工作就可以开发出利用网络进行通信的软件。

6.2 利用 WinSock 进行无连接的通信

WinSock 提供了对 UDP（用户数据报协议）的支持，通过 UDP 协议我们可以向指定 IP 地址的主机发送数据，同时也可以从指定 IP 地址的主机接收数据，发送和接收方处于相同的地位没有主次之分。利用 CSocket 操纵无连接的数据发送很简单，首先生成一个本地套接口（需要指明 SOCK_DGRAM 标记），然后利用 int CAsyncSocket::SendTo(const void* lpBuf, int nBufLen, UINT nHostPort, LPCTSTR lpszHostAddress = NULL, int nFlags = 0)发送数据，int CAsyncSocket::ReceiveFrom(void* lpBuf, int nBufLen, CString& rSocketAddress, UINT& rSocketPort, int nFlags = 0)接收数据。函数调用顺序如图。

利用 UDP 协议发送和接收都可以是双向的，就是说任何一个主机都可以发送和接收数据。但是 UDP 协议是无连接的，所以发送的数据不一定能被接收，此外接收的顺序也有可能与发送顺序不一致。下面是相关代码：

```
/*
发送方在端口 6800 上向接收方端口 6801 发送数据
*/
//发送方代码：
BOOL CMy62_s1_clientDlg::OnInitDialog()
{
CDialog::OnInitDialog();

//创建本地套接口
m_sockSend.Create(6800,SOCK_DGRAM,NULL);
//绑定本地套接口
m_sockSend.Bind(6800,"127.0.0.1");
//创建一个定时器定时发送
SetTimer(1,3000,NULL);
...
}
void CMy62_s1_clientDlg::OnTimer(UINT nIDEvent)
{
static iIndex=0;
char szSend[20];
sprintf(szSend,"%010d",iIndex++);
//发送 UDP 数据
int iSend= m_sockSend.SendTo(szSend,10,6801,"127.0.0.1",0);
TRACE("sent %d byte\n",iSend);
...
}

//接收方代码
BOOL CMy62_s1_serverDlg::OnInitDialog()
```

```
{
CDialog::OnInitDialog();

//创建本地套接口
m_sockRecv.Create(6801,SOCK_DGRAM,"127.0.0.1");
//绑定本地套接口
m_sockRecv.Bind(6801,"127.0.0.1");
//创建一个定时器定时读取
SetTimer(1,3000,NULL);
...
}
void CMy62_s1_serverDlg::OnTimer(UINT nIDEvent)
{
char szRecv[20];
CString szIP("127.0.0.1");
UINT uPort=6800;
//接收 UDP 数据
int iRecv =m_sockRecv.ReceiveFrom(szRecv,10,szIP,uPort,0);
TRACE("received %d byte\n",iRecv);
...
}
/*
接收方采用同步读取数据的方式，所以没有读到数据函数调用将不会返回
*/
```

下载例子代码，62_s1_client 工程为发送方，62_s1_server 工程为接收方。

6.3 利用 WinSock 进行有连接的通信

WinSock 提供了对 TCP（传输控制协议）的支持，通过 TCP 协议我们可以与指定 IP 地址的主机建立，同时利用建立的连接可以双向的交换数据。利用 CSocket 操纵有连接数据交换很简单，但是在有连接的通信中必需有一方扮演服务器的角色等待另一方（客户方）的连接请求，所以服务器方需要建立一个监听套接口，然后在此套接口上等待连接。当连接建立后会产生一个新的套接口用于通信。而客户方在创建套接口后只需要简单的调用连接函数就可以创建连接。对于有连接的通信不论是数据的发送还是发送与接收的顺序都是有保证的。

下面的代码演示了如何建立连接和发送/接收数据：

```
/*
服务器方在端口 6802 上等待连接，当连接建立后关闭监听套接口
客户方向服务器端口 6802 发起连接请求
*/
BOOL CMy63_s1_serverDlg::OnInitDialog()
{
```

```
CDialog::OnInitDialog();

CSocket sockListen;
//创建本地套接口
sockListen.Create(6802,SOCK_STREAM,"127.0.0.1");
//绑定参数
sockListen.Bind(6802,"127.0.0.1");
sockListen.Listen(5);
//等待连接请求，m_sockSend 为成员变量，用于通信
sockListen.Accept(m_sockSend);
//关闭监听套接口
sockListen.Close();
//启动定时器，定时发送数据
SetTimer(1,3000,NULL);
...
}
void CMy63_s1_serverDlg::OnTimer(UINT nIDEvent)
{
static iIndex=0;
char szSend[20];
sprintf(szSend,"%010d",iIndex++);
//发送 TCP 数据
int iSend= m_sockSend.Send(szSend,10,0);
...
}
BOOL CMy63_s1_clientDlg::OnInitDialog()
{
CDialog::OnInitDialog();
//创建本地套接口
m_sockRecv.Create();
//发起连接请求
BOOL fC=m_sockRecv.Connect("127.0.0.1",6802);
TRACE("connect is %s\n",(fC)?"OK":"Error");
//启动定时器，定时接收数据
SetTimer(1,3000,NULL);
...
}
void CMy63_s1_clientDlg::OnTimer(UINT nIDEvent)
{
char szRecv[20];
//接收 TCP 数据
int iRecv =m_sockRecv.Receive(szRecv,10,0);
TRACE("received %d byte\n",iRecv);
if(iRecv>=0)
```

```
{
szRecv[iRecv]='\0';
m_szRecv=szRecv;
UpdateData(FALSE);
}
...
}
```

下载例子代码，63_s1_client 工程为客户，63_s1_server 工程为服务器方。