第 3 章　程序设计实践——MFC 基础

程序设计实践旨在课程学习的基础上帮助读者掌握 C++应用系统的开发方法和技巧。通过对几个小型 C++应用程序实例设计与实现过程的分析，帮助读者掌握利用 C++开发应用系统的一般设计方法与实现步骤。作为教材内容的拓展，本章介绍 MFC 的基本知识。

3.1　MFC 概述

MFC（Microsoft Foundation Class Library，微软基础类库）是微软基于 Windows 平台下的 C++类库集合。MFC 包含了所有与系统相关的类，其中封装了大多数的 API（Application Program Interface）函数，提供了应用程序框架和开发应用程序的工具，如应用程序向导、类向导、可视化资源设计等高效工具，用消息映射处理消息响应，大大简化了 Windows 应用程序的开发工作，使程序员可以从繁重的编程工作中解脱出来，提高了工作效率。

1．MFC 与 Windows 编程

Windows 操作系统采用了图形用户界面，借助它提供的 API 函数，用户可以编出具有图形用户界面的程序。Windows 操作系统下的应用程序和控制台方式（MS-DOS）下的应用程序相比，具有如下特点：

（1）用户界面统一、友好。Windows 应用程序拥有相似的基本外观，包括窗口、菜单栏、工具栏、状态栏、滚动条等标准元素。

（2）独立于设备的图形操作。Windows 下的应用程序使用图形设备接口（Graphic Device Interface）屏蔽了不同设备的差异，提供了设备无关的图形输出能力。

（3）支持多任务。Windows 是一个多任务的操作环境，允许用户同时运行多个应用程序。

（4）事件驱动的程序设计。Windows 程序不是由事件的顺序来控制，而是由事件的发生与否来控制程序执行逻辑。而事件是与消息关联的，Windows 应用程序的消息来源有以下 4 种：

- 输入消息：包括键盘和鼠标的输入。
- 控制消息：用来与 Windows 的控制对象，如列表框、按钮、复选框等进行双向通信。
- 系统消息：对程序化的事件或系统时钟中断做出反应。
- 用户消息：这是程序员自己定义并在应用程序中主动发出的。

在 VC++中编写 Windows 应用程序有以下两种方法。

（1）直接调用 Windows 操作系统提供的 Windows API 函数来编写 Windows 应用程序。通过 Windows API 创建的 Windows 应用程序有两个基本部分：应用程序主函数 WinMain 和窗口函数。WinMain 函数是应用程序的入口点，相当于 C++控制台应用程序的主函数 main。与 main 函数一样，WinMain 函数名也是固定的。窗口函数的名字是用户自定义的，由系统调用，主要用来处理窗口消息，以完成特定的任务。使用 Windows API 编写 Windows 应用程序时，大量的程序代码必须由程序员自己编写，工作量大。

（2）使用 MFC 类库编写 Windows 应用程序。MFC 提供了大量预先编写好的类及支持代码，用于处理多项标准的 Windows 编程任务，如创建窗口、处理消息、添加工具栏和对话框

等。因此，使用 MFC 类库可以简化 Windows 应用程序的编写工作。

2．MFC 应用程序框架

MFC 封装了大部分 Windows API 函数、数据结构和宏，以面向对象的类提供给程序员，并提供了一个应用程序框架，简化和标准化了 Windows 程序设计。

MFC 中的各种类加起来有几百个，其中只有 5 个核心类对应用程序框架有影响：CWinApp、CDocument、CView、CFrameWnd 和 CDocTemplate。这 5 个类之中只有 CWinApp 是必不可少的类，CWinApp 的对象在应用程序中必须有一个，也只有一个，并且是一个全局对象。全局对象是在 Windows 操作系统调用 WinMain 之前建立的，它开通了程序执行的路径。在 MFC 编程中，入口函数 WinMain 被封装在 MFC 的应用程序框架内，称为 AfxWinMain，不需要也不可以再定义另一个 WinMain 函数。

应用程序框架（Application Framework）是一组类构造起来的大模型。它的出现使得开发人员不需要构建程序框架结构，其初始代码可以由应用程序向导自动完成。

3．MFC 应用程序向导

MFC 应用程序向导（MFC AppWizard）可以帮助程序员创建一个 MFC 应用程序框架，并且自动生成这个 MFC 应用程序框架所需要的全部文件。然后，程序员利用资源管理器和类向导（ClassWizard），为应用程序添加实现特定功能的代码，以实现应用程序所要求的功能。

在 Visual C++中，可以创建以下 3 类典型的 Windows 应用程序，它们都是通过 MFC 应用程序向导（AppWizard）创建的：

（1）基于对话框的应用程序：这类程序适合于文档较少而交互操作较多的应用场合，如 Windows 自带的计算器程序。

（2）单文档界面（SDI）应用程序：这类程序一次只能打开一个文档，如 Windows 自带的 Notepad 程序。

（3）多文档界面（MDI）应用程序：这类程序可以同时打开多个文档并进行处理，处理的过程中很容易进行切换，如 Microsoft Word。

下面通过构建应用程序 myprog1，简单介绍如何使用 MFC 应用程序向导。

启动 Microsoft Visual C++ 6.0，进入 Visual C++集成开发环境（IDE），进行如下操作：

（1）选择“文件（File）”→“新建（New）”菜单，弹出“新建”对话框。

（2）在“新建”对话框中，切换到“工程（Projects）”选项卡，从左边窗口所示的“项目类型”列表框中选择要创建的项目类型，这里选择“MFC AppWizard（exe）”。在“工程名称（Projects name）”文本框中键入应用程序名，这里为 myprog1。在“位置（Location）”文本框中键入用于存放应用程序的目录，这里为 F:\MYPROJ\myprog1。从“平台（Platforms）”列表框中选择可用的目标平台，默认平台为 Win32。最后单击“确定”按钮，弹出 MFC AppWizard-Step1 对话框。

（3）MFC 应用程序向导－步骤一（AppWizard-Step l of 6）对话框用于确定应用程序的结构：单文档、多文档或基本对话框，并为资源选择一种语言。

这里选择 Single document（单文档），其余步骤（步骤二至步骤六）取默认设置，单击“完成”按钮，显示应用程序所具有的特征。单击“确定”按钮，确认前面所做的选择都正确，则 MFC AppWizard 根据这些选择生成应用程序的相应源文件。

（4）完成上述步骤后，应用程序的框架即被生成，并在开发环境（Developer Studio）的程序工作区窗口中打开生成的程序。其中 ClassView 面板显示的是所创建的类和成员函数；

ResourceView 面板显示的是所创建的资源；FileView 面板显示的是所创建的初始文件。

（5）选择“组建（Build）”→“组建<myprog1.exe>”命令，编译、连接、建立运行程序。

（6）选择“组建（Build）”→“执行<myprog1.exe>”命令，运行应用程序。

从运行结果可以看出，尽管还未写入一句代码，但 myprog1 程序已经是一个完整的可执行程序了，其运行结果已包含标题栏、工具栏、菜单栏和一个打开的文档边框窗口。

生成应用程序框架后，这仅仅是一个程序的最基本的骨架，往往还需往项目中添加大量的代码，包括类、资源、消息处理函数等。

4. MFC 与消息映射

Windows 应用程序都是消息（Message）驱动的，消息处理是 Windows 应用程序的核心部分。消息是用来请求对象执行某一处理、某一行为的信息。某对象在执行相应的处理时，如果需要，它可以通过传递消息请求其他对象完成某些处理工作或回答某些信息。其他对象在执行所要求的处理活动时，同样可以通过传递消息与别的对象联系，即 Windows 应用程序的运行是靠对象间传递消息来完成的。消息实现了对象与外界，对象与其他对象之间的联系。

消息主要有 3 种类型：标准 Windows 消息、控件通知消息和命令消息。

（1）标准 Windows 消息。

除 WM-COMMAND 外，所有以“WM-”为前缀的消息都是标准 Windows 消息。标准 Windows 消息由窗口和视图处理，这类消息通常会含有用于确定如何对消息进行处理的一些参数。标准 Windows 消息都有缺省的处理函数，这些函数在 CWnd 类中进行了预定义。MFC 类库以消息名为基础形成这些处理函数的名称，这些处理函数的名称都有前缀“On”。

（2）控件通知消息。

控件通知消息包含从控件和其他子窗口传送给父窗口的 WM-COMMAND 通知消息。像其他标准的 Windows 消息一样，控件通知消息由窗口和视图处理，但当用户单击按钮控件时，发生的 BN-CLICKED 控件通知消息将作为命令消息来处理。

（3）命令消息。

命令消息包含来自用户界面对象（如菜单项、工具栏按钮和加速键等）的 WM-COMMAND 通知消息。命令消息的处理与其他消息的处理不同，命令消息可以被更广泛的对象（如文档、文档模板、应用程序对象、窗口和视图）处理。如果某条命令直接影响某个特定的对象，则应该让该对象来处理这条命令。

Windows 程序这种“接收消息－处理消息－再等消息”的往复过程即称为“消息循环”。消息循环是 Windows 应用程序与 MS-DOS 应用程序的最大差异点。

在 Windows 平台，程序员不能决定程序执行的流程，而只能决定接收到消息时的程序的动作（编写完成局部的代码）。在 Visual C++中编程不是考虑要让程序按照什么样的顺序执行，而应考虑在某一消息下程序应该干什么。

3.2 MFC 与菜单设计

在 Visual C++中设计应用程序菜单将主要用到资源编辑器（AppStudio）以及类向导（ClassWizard）工具，一般包括如下操作内容：

（1）使用资源编辑器（AppStudio）建立和编辑菜单项目。

（2）使用类向导工具创建与上述菜单项目对应的菜单消息处理函数框架。

（3）在菜单消息处理函数框架中写入消息处理代码。

下面通过实例介绍如何在 Visual C++集成开发环境中设计生成菜单、工具栏和加速键资源，并为其定义命令处理函数。

1. 生成新项目

用应用程序向导生成单文档的项目文件 Hello。

2. 编辑菜单

编辑由应用程序向导自动生成的菜单资源，进行下列操作：

（1）选择项目工作区的 Resource View 标签，切换到资源视图。

（2）选择“菜单”资源类型。

（3）双击菜单资源 IDR_MAINFRAME，将弹出菜单编辑窗口。

（4）编辑当前菜单：

- 删除：要删除某个菜单项或弹出菜单，可单击该菜单或用上下光标键选择，然后按 Del 键删除。
- 插入：要插入新菜单项，可选定窗口中的空白菜单框后按回车，将弹出属性（Properties）对话框，如图 3.1 所示。属性对话框用于输入菜单项的标题、资源 ID、菜单项在状态栏上显示的提示，并为该菜单项提供属性调整。也可以在选择一个已有的菜单项时按 Ins 键，在该菜单项上方插入一个空白菜单项，然后双击该菜单项进行编辑。要插入一个分隔线，只需将菜单项的分隔符（Seperator）属性打开即可。

图 3.1　“菜单项目属性”对话框

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

利用菜单编辑器在“编辑”和“查看”菜单之间插入“欢迎”弹出菜单，然后在“欢迎”弹出菜单下插入 4 个子菜单项，并在&SayHello 菜单下插入一个分隔符。菜单项属性设置如表 3.1 所示。

表 3.1　菜单项属性设置

菜单名	菜单性质	对象 ID 名称	菜单提示
欢迎	弹出菜单	无	无
Say hello	命令菜单	ID_SAY_HELLO	Hello!
Red	命令菜单	ID_RED	The color is red
Green	命令菜单	ID_GREEN	The color is green
Blue	命令菜单	ID_BLUE	The color is blue

3. 为菜单项定义命令处理函数

用类向导为上面创建的几个菜单生成和映射消息处理成员函数。进行下列操作：

（1）选择“查看（View）”→“建立类向导（ClassWizard）”菜单，将弹出 MFC ClassWizard 对话框，如图 3.2 所示。

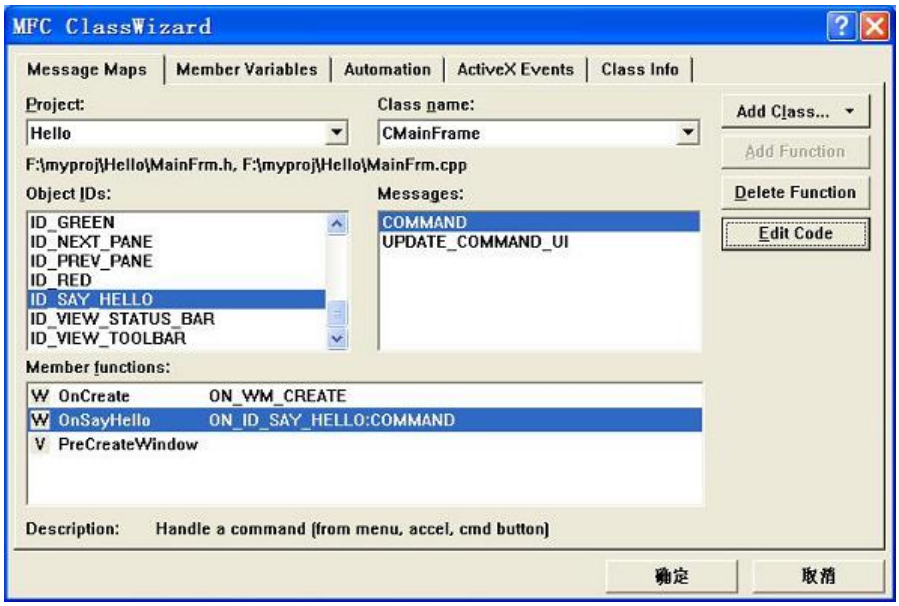

图 3.2　用 ClassWizard 增加菜单消息成员函数

（2）选择 Message Maps 选项卡，在 Class name 下拉列表中选择 CMainFrame 类。在 Object IDs 中选择 ID_SAY_HELLO，在 Messages 栏中双击 COMMAND，弹出 Add Member Function 对话框。对话框中给出默认的成员函数 OnSayHello，单击 OK 按钮接收默认的成员函数名。此时成员函数 OnSayHello 就出现在 Member functions 列表框中，后面是所映射的消息。

如此，依次为 ID_RED、ID_BLUE、ID_GREEN 增加消息处理成员函数 OnRed、OnBlue、OnGreen。

（3）双击 Member Functions 列表框中的 OnSayHello，编辑 OnSayHello 成员函数。在 OnSayHello 成员函数体中加入如下代码：

```
void CMainFrame::OnSayHello()
{
    // TODO: Add your command handler code here
    AfxMessageBox("Hello!");
}
```

另外在 CHelloView::OnDraw(CDC *pDC)加入代码如下：

```
void CHelloView::OnDraw(CDC* pDC)
{
    CHelloDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    pDC->TextOut(0,0,"Hello Weclome to use MFC !");
}
```

下面再来编写 OnRed、OnBlue、OnGreen 三个函数。首先双击 CMainFrame 类名，在 MainFrm.h 中加入数据成员，如下所示：

```
class CMainFrame:public CFrameWnd{
...
//Attributes
public:
int m_nColor;
enum{RED=0,GREEN=1,BLUE=2};
...
}
```

加入数据成员后，还要对它进行初始化，初始化工作在 CMainFrame()构造函数中完成。

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

OnRed、OnBlue、OnGreen 三个函数的程序清单如下所示：

```
void CMainFrame::OnRed()
{
    // TODO: Add your command handler code here
    m_nColor=RED;
    AfxMessageBox("Color is red!");
}

void CMainFrame::OnGreen()
{
    // TODO: Add your command handler code here
    m_nColor=GREEN;
    AfxMessageBox("Color is green!");
}

void CMainFrame::OnBlue()
{
    // TODO: Add your command handler code here
    m_nColor=BLUE;
    AfxMessageBox("Color is blue!");
}
```

这样，当选择颜色时，就会提示不同的消息框显示当前选择的颜色。如果要求选择不同的颜色后，在菜单名前显示一个勾，表明这是当前选项。要实现这一功能，就要使用 MFC 框架的更新命令用户界面消息机制。

4. 更新命令用户界面消息

一般情况下，菜单项和工具栏按钮都不止一种状态，应用程序经常需要根据内部状态来对菜单项和工具栏按钮的外观作相应的改变。例如，在没有选择任何内容时，编辑菜单的“复制”、“剪切”等项是灰色显示。有时，还会看到在菜单项旁边有标记，表示它是选中的还是不选中的。工具栏也有类似的情况，如果按钮不可用也可以被置成无效。

（1）更新机制。

MFC 应用程序框架引入了更新命令用户界面消息来专门解决这一问题。在下拉菜单之前，或在工具栏按钮处于空闲循环期间，MFC 会发一个更新命令，这将导致命令更新处理函数的调用。命令更新处理函数可以根据情况，使用户界面对象（主要指菜单项和工具栏按钮）允许或禁止。

对于每一个菜单项，将对应两种消息：COMMAND 和 UPDATE_COMMAND_UI。其中 UPDATE_COMMAND_UI（用户界面更新消息）用于处理菜单项和工具栏按钮的更新。每一个菜单命令都对应于一个 UPDATE_COMMAND_UI 消息。可以为 UPDATE_COMMAND_UI 编写消息处理函数来处理用户界面（包括菜单和工具栏按钮）的更新。如果一条命令有多个用户界面对象（比如一个菜单项和一个工具栏按钮），两者都被发送给同一个消息处理函数。这样，对于所有等价的用户界面对象来说，可以把用户界面更新代码封装在同一地方。

（2）编写消息处理函数。

当框架给消息处理函数发送更新命令时，它给函数传递一个指向 CCmdUI 对象的指针。这个对象包含了相应的菜单项或工具栏按钮的指针。更新处理函数利用该指针调用菜单项或工具栏按钮的命令接口函数来更新用户界面对象（包括灰化、使能，选中菜单项和工具栏按钮等）。下面给菜单项编写更新消息处理函数来更新用户界面。操作如下：

选择“查看”→“建立类向导”菜单，在弹出的 MFC ClassWizard 对话框中选择 Message Maps 选项卡。在 Object IDs 列表框中选择 ID_RED，在 Messages 列表框中双击 UPDATE_COMMAND_UI，弹出 Add Member Function 对话框，单击 OK 按钮接收默认函数名为 OnUpdateRed。依次给 ID_BLUE、ID_GREEN 增加 OnUpdateBlue 和 OnUpdateGreen 用户界面更新消息处理函数。程序清单如下：

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

void CMainFrame::OnUpdateGreen(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_nColor==GREEN);
}

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

现在编译运行 Hello 程序。当打开“欢迎”菜单时，如图 3.3 所示，在 Red 菜单项前已经打了一个勾。

5. 编辑工具栏

应用程序向导（AppWizard）已经为程序自动生成了一个工具栏资源。如果程序只需一个工具栏，则不需要增加工具栏资源了，只需修改和编辑它即可。操作如下：

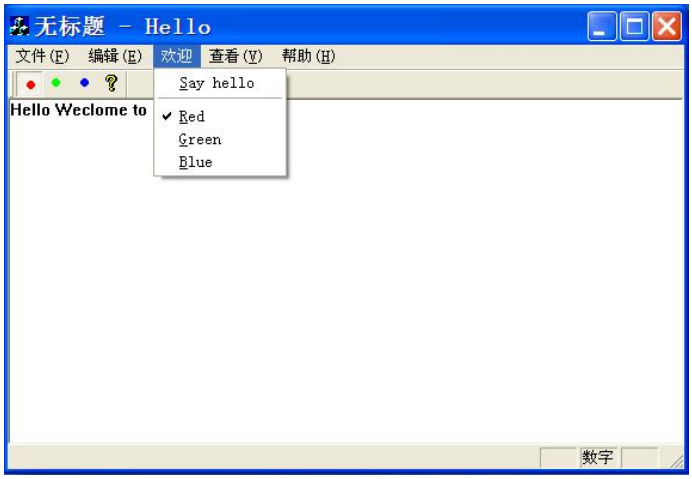

图 3.3　Hello 程序运行结果

（1）将项目工作区切换到资源视图，并展开名为 IDR_MAINFRAME 的 Toolbar（工具栏）资源。双击“IDR_MAINFRAME”，Developer Studio 会打开一个功能强大的工具栏资源编辑窗口，如图 3.4 所示。

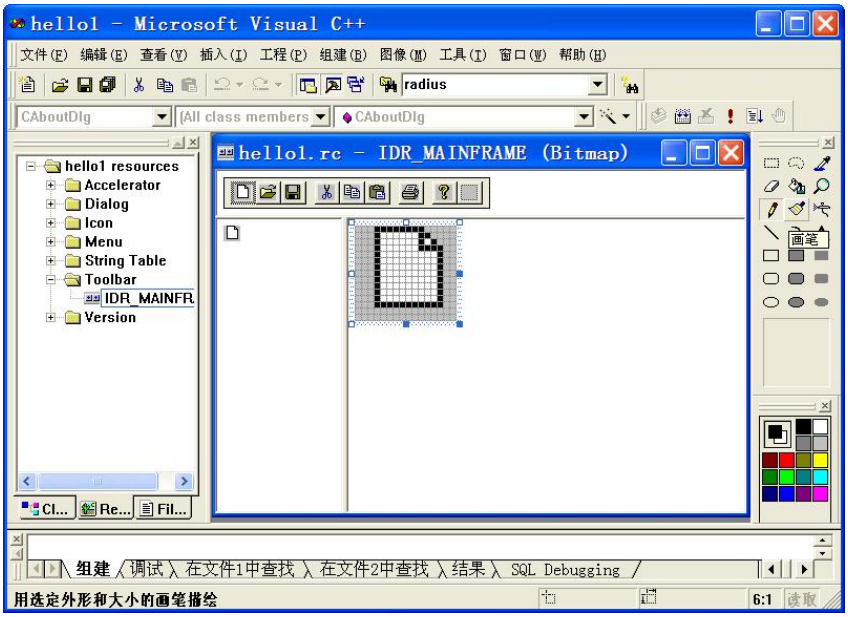

图 3.4　工具栏资源编辑窗口

该窗口的上部显示出了工具栏上的按钮，当用户用鼠标选择某一按钮时，在窗口的下部会显示该按钮的位图。在窗口旁边有一个绘图工具面板和一个颜色面板，供用户编辑按钮位图时使用。

（2）删除“？”按钮前面的所有按钮。方法是用鼠标将要删除的按钮拖出工具栏即可。

（3）先选中“？”按钮后面的空白按钮，然后在该按钮的放大位图上用红色画一个实心圆圈，以表示选择红色。再选中空白按钮，并用绿色在放大位图上画一个实心圆圈，以表示绿色。同理画一个表示蓝色的按钮。

通过用鼠标拖动按钮调整按钮的位置，调整后的位置如图 3.3 的工具栏所示。

（4）分别为 3 个新加的按钮指定命令 ID。方法是，先选中一个按钮，然后按回车键，在弹出的属性对话框中输入 ID，或从 ID 下拉列表中分别选择与按钮功能相同的菜单项的 ID，这样同样的命令既可以通过菜单执行，也可以通过工具栏执行。

（5）为按钮指定命令提示。分别双击 3 个颜色按钮，在弹出的属性对话框中的提示（Prompt）栏内已有的信息后分别输入工具栏提示，分别为：

The color is red.\nred

The color is green.\ngreen

The color is blue.\nblue

在提示栏中，两者用“\n”分隔开。当鼠标移动到某个菜单项或工具栏上的按钮时，在状态栏中就会显示状态栏提示，当鼠标在某个按钮上停留片刻后，就会弹出一个黄色的工具提示窗口显示“\n”后的信息。

6. 定义加速键

除了上述的菜单和工具栏外，加速键还是产生命令的另一来源。加速键通常用于快捷地激活某个菜单命令或者工具栏命令，因此，大多数情况下，加速键与菜单或者工具栏是对应的。

下面为 SayHello 菜单增加一个加速键 Ctrl+H。操作如下：

（1）将项目工作区切换到资源视图，选择 Accelerator 资源类型，双击 IDR_MAINFRAME 加速键资源，打开加速键编辑窗口。

（2）编辑加速键资源。要删除加速键，可以直接按 Del 键。要增加加速键，可以按 Ins 键，则弹出加速键属性对话框。在 ID 下拉列表框中选择 ID_SAY_HELLO，在“键（K）”中输入 H，选择 Ctrl 复选框和“虚拟键（VirtKey）”单选按钮，完成加速键设置，如图 3.5 所示。

图 3.5 加速键的设置

（3）关闭加速键编辑窗口。

最后编译运行程序，测试菜单、工具栏和加速键，三者都能产生命令消息，程序运行结果如图 3.3 所示。借助 Visual C++集成开发环境的应用程序向导、类向导、资源编辑器等工具，生成和修改菜单、工具栏和加速键资源，并为三者产生的命令消息定义处理函数框架是很容易的事情。

3.3 MFC 与对话框设计

对话框是一种特殊类型的窗口，绝大多数 Windows 程序都通过对话框与用户进行交互。在 Visual C++中，对话框既可以单独组成一个简单的应用程序，又可以成为文档/视图结构程序的资源。

在 Visual C++中，创建对话框程序一般包括如下步骤：

（1）使用资源编辑器（AppStudio）在应用程序中新建用户对话框窗口。

（2）放置控件并设置控件属性。

（3）为控件连接变量。

（4）使用类向导（ClassWizard）创建与上述用户对话框窗口控件对应的消息处理函数框架。

（5）在消息处理函数中写入消息处理代码。

新建的默认的对话框有 OK 和 Cancel 两个按钮，在窗口的旁边有一个控件面板，在控件面板上用鼠标选择一个控件，然后在对话框中单击，则相应的控件就被放置到了对话框窗口中。图 3.6 显示了控件面板上的按钮所代表的控件。

图 3.6　控件面板

对话框中每个控件都有属性，选中控件或对话框后按回车键，则会弹出一个属性对话框，属性对话框用来设置控件或对话框的各种属性。一个典型的控件属性对话框如图 3.7 所示。

图 3.7　控件属性对话框

控件属性含义如下：

- ID 属性。用于指定控件的标识符，Windows 依靠 ID 来区分不同的控件。程序经常用它来加载资源或者选取某个窗口。一般使用 ID 区别一个窗口内的子窗口；
- 标题（Caption）属性。静态正文、组框、按钮、复选框、单选按钮等控件可以显示

标题，用来对控件进行文字说明。控件标题中的字符&使紧跟其后的字符有下划线，按 Alt+下划线字母将启动该控件。若控件是一个单选按钮，则 Alt+下划线字符将选择该按钮；若是复选框，则相当于对该复选框按空格键；若是按钮，则将激活按钮命令；若控件是一个静态正文，则将激活按 Tab 顺序紧随其后的下一个控件；

- 可见（Visible）属性。用来指定控件创建后是否为默认可见的；
- 已禁用（Disable）属性。使控件允许或禁止，一个禁止的控件呈灰色显示，不能接收任何输入；
- 制表位（Tabstop）属性。用户可以按 Tab 键移动到具有制表位属性的控件上。Tab 移动的顺序可以由用户指定。按 Ctrl+D 组合键则 Tab 顺序会显示出来，用户可以用鼠标来重新指定 Tab 顺序。默认的 Tab 顺序是控件的创建次序；
- 组（Group）属性。用来指定一组控件，用户可以用箭头键在该组控件内移动。同一组内的单选按钮具有互斥的特性，即在这些单选按钮中只能有一个是选中的。如果一个控件具有 Group 属性，则这个控件以及按 Tab 顺序紧随其后的所有控件都属于一组，直到遇到另一个有 Group 属性的控件为止。

下面以一个简单的文本编辑器说明 MFC 中对话框的设计方法。该文本编辑器使用单选按钮实现对文本字体的编辑，使用复选框实现对文本字型的编辑，使用微调控件实现对文本颜色的编辑修改。

1. 生成新项目

用应用程序向导生成基本对话框的项目文件 myedit，得到默认的对话框如图 3.8 所示。

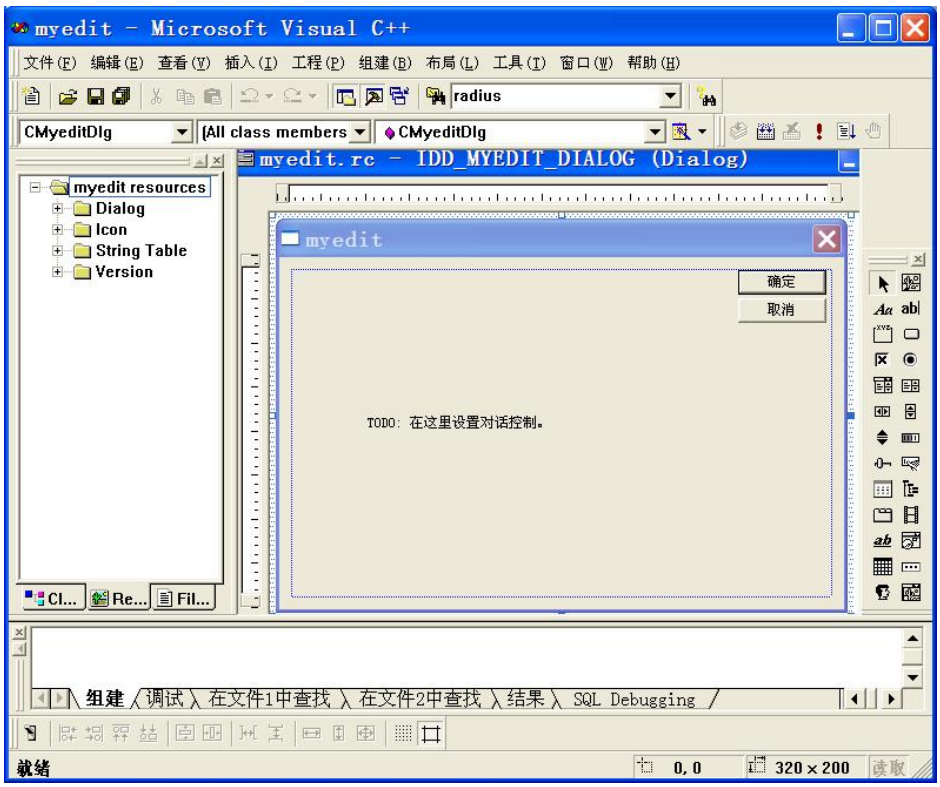

图 3.8　默认的对话框

2. 放置控件并设置控件属性

删除默认对话框中不需要的控件，重新设计对话框界面，界面设计如图 3.9 所示。设置控件属性如表 3.2 所示。

图 3.9　文本编辑对话框界面

表 3.2　主要控件属性

控件名称	ID	标题	备注
文本编辑框	IDC_EDIT1		其他属性默认
单选按钮	IDC_ST_RADIO	宋体	设置组属性
单选按钮	IDC_LS_RADIO	隶书	其他属性默认
单选按钮	IDC_KT_RADIO	楷体	其他属性默认
复选框	IDC_CT_CHECK	粗体	其他属性默认
复选框	IDC_XT_CHECK	斜体	其他属性默认
复选框	IDC_XHX_CHECK	下划线	其他属性默认
文本编辑框	IDC_Red		其他属性默认
文本编辑框	IDC_Green		其他属性默认
文本编辑框	IDC_Blue		其他属性默认
微调控件	IDC_Red_SPIN		设置自动结伴及自动结伴整数
微调控件	IDC_Green_SPIN		同上
微调控件	IDC_Blue_SPIN		同上

注意：3 个单选按钮为一组，在第 1 个单选按钮的属性设置对话框设置“组”属性。3 个微调控件的作用是调节文本颜色，需要设置“自动结伴”和“设置结伴整数”两个属性，将微调控件与左侧的编辑框设置在一起，如图 3.10 所示。

图 3.10　设置微调（spin）控件

另外需设置各控件的 Tab 顺序，使 3 个单选按钮为一组，同时使文本编辑框与微调控件自然绑定在一起，具体操作为：选择“布局”→“Tab 顺序”菜单，设置各控件 Tab 顺序如图 3.11 所示。

图 3.11　设置各控件的 Tab 顺序

3. 为控件连接成员变量

成员变量和控件连接时，分为两类：Value（值）和 Control（控件）。和控件连接后的具有 Value 类的成员变量，变量的变化可以通过控件体现出来（需执行函数 UpdateData(false)），控件的值或状态变化时也会反映到相应的变量中（需执行函数 UpdateData(true)）。和控件连接后的具有 Control 类的成员变量，可以访问控件类的成员函数，这给编程带来很大的方便。本例为各控件连接的成员变量见表 3.3。

表 3.3　和控件连接的成员变量

变量名称	类（Catalog）	变量类型	控件 ID
m_TextEdit	Value	CString	IDC_EDIT1
m_TextFont	Control	CEdit	IDC_EDIT1
m_CT	Value	BOOL	IDC_CT_CHECK
m_XHX	Value	BOOL	IDC_XHX_CHECK
m_XT	Value	BOOL	IDC_XT_CHECK
m_Blue_Edit	Value	UINT	IDC_Blue
m_Green_Edit	Value	UINT	IDC_Green
m_Red_Edit	Value	UINT	IDC_Red

续表

变量名称	类（Catalog）	变量类型	控件 ID
m_Red	Control	CSpinButtonCtrl	IDC_Red_SPIN
m_Green	Control	CSpinButtonCtrl	IDC_Green_SPIN
m_Blue	Control	CSpinButtonCtrl	IDC_Blue_SPIN

为控件连接成员变量，方法如下。

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，选择 Member Variables 选项卡，在 Class names 中选择对话框类名 CMyEditDlg，在 Control IDs 列表框中双击相应控件的 ID，或选中相应控件的 ID，单击 Add Variable 按钮，在弹出的对话框中（如图 3.12 所示）输入变量名及相应的类型。

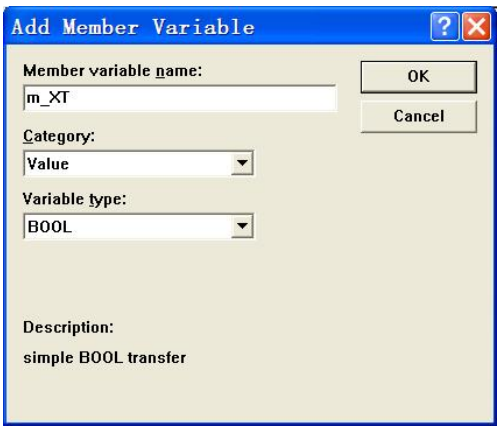

图 3.12　添加成员变量对话框

分别为文本编辑框控件、复选框控件连接成员变量，完成后如图 3.13 所示。

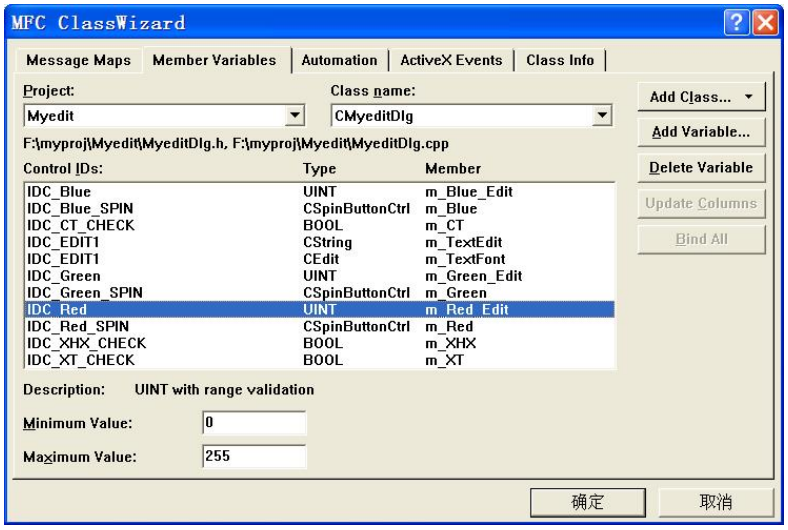

图 3.13　为控件添加的成员变量

在文件 MyeditDlg.h 中添加类型为 CString 的成员变量 temp 和类型为 CFont 的成员变量

fontText，保存设置字体的内容，创建新字体、字型，其代码设计如下：

```
class CMyeditDlg : public CDialog
{
// Construction
public:
    CMyeditDlg(CWnd* pParent = NULL);        // standard constructor（标准构造函数）
    CString temp;
    CFont fontText;                          //定义字体实例 fontText
……                                           //此处略去系统生成的相关语句
}
```

完成 MyeditDlg.h 中成员变量定义的代码如下:

```
class CMyeditDlg : public CDialog
{
// Construction
public:
    CMyeditDlg(CWnd* pParent = NULL);        // standard constructor
    CString temp;
    CFont fontText;//定义字体实例 fontText
    // Dialog Data
    //{{AFX_DATA(CMyeditDlg)
    enum { IDD = IDD_MYEDIT_DIALOG };
    CSpinButtonCtrl m_Red;
    CSpinButtonCtrl m_Green;
    CSpinButtonCtrl m_Blue;
    CEditm_TextFont;
    CString    m_TextEdit;
    BOOL       m_CT;
    BOOL       m_XHX;
    BOOL       m_XT;
    UINT       m_Blue_Edit;
    UINT       m_Green_Edit;
    UINT       m_Red_Edit;
     ……
}
```

4. 创建与控件对应的消息处理函数框架及添加代码

（1）初始化界面。

向对话框派生类 CMyeditDlg 中添加初始化成员函数 OnInitDialog()，在此成员函数中实现成员变量的初始化，方法如下。

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，在 Message Maps 选项卡的 Class names 中选择对话框类名 CMyEditDlg，在 Object IDS 中选择对话框资源标识符 CMyEditDlg，在 Messages 列表框中选择初始化对话框消息 WM_INITDIALOG。

双击 WM_INITDIALOG，重载虚成员函数 OnInitDialog。它仅在调用“文本编辑对话框”时被一次性调用，用于对话框控件、变量的初始化。OnInitDialog 中新添加的代码如下。

```
BOOL CMyeditDlg::OnInitDialog()
{
CDialog::OnInitDialog();
```

```
m_CT=m_XT=m_XHX=FALSE;
m_TextEdit=_T("中南大学");          //将文本编辑框初始化为“中南大学”
UpdateData(FALSE);//将变量的变化反映到与之相连的控件上
fontText.CreateFont(32,32,0,0,0,FALSE,FALSE,FALSE,DEFAULT_CHARSET,
OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,"黑体");
m_TextFont.SetFont(&fontText);      //将所设置字体施加在文本编辑框
    m_Red.SetRange(0,255);          //设置红色微调按钮范围
    m_Green.SetRange(0,255);        //设置绿色微调按钮范围
    m_Blue.SetRange(0,255);         //设置蓝色微调按钮范围
……//此处略去系统生成的相关语句
    return TRUE;                    // return TRUE  unless you set the focus to a control
}                                   //（返回真，除非对控件设置焦点）
```

Windows 程序设计中，要实现文本编辑框字体、字型等属性的设置，首先要调用 CFont 类成员函数 CreateFont()创建新字体、字型、字号，接着调用 SetFont()函数对文本编辑控件进行设置。

CreateFont()函数包含 14 个参数，其函数原型如下：

```
BOOL CreateFont(int nHeight,        //字体高度
int nWidth,                         //字体宽度
int nEscapement,
int nOrientation,
int nWeight,                        //粗体
BYTE bItalic,                       //斜体
BYTE bUnderline,                    //下划线
BYTE cStrikeOut,                    //删除线
BYTE nCharSet,                      //字符集
BYTE nOutPrecision,
BYTE nClipPrecision,BYTEnQuality,BYTEnPitchAndFamily,LPCTSTRlpszFacename
//字体
);
```

（2）为单选按钮和复选框添加消息处理函数。

添加响应鼠标单击字体单选按钮、字型复选框的 6 个成员函数，分别为：宋体 OnStRadio()、隶书 OnLsRadio()、楷体 OnKtRadio()、粗体 OnCtCheck()、斜体 OnXtCheck()和下划线 OnXhxCheck()。方法如下。

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，在 Message Maps 选项卡的 Class names 中选择对话框类名 CMyEditDlg，在 Object IDS 中选择单选按钮资源标识符 IDC_ST_RADIO，在 Messages 列表框中选择 BN_CLICKED（单击事件），再单击 Add Function…按钮，在弹出的对话框中用默认的函数名，单击 OK 按钮，完成 OnStRadio()函数的创建（如图 3.14 所示）。依次完成上述其他成员函数的创建。

为成员函数添加代码。方法如下。

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，在 Message Maps 选项卡的 Member function 列表框中选择刚才创建的成员函数 OnStRadio()，双击该行，或单击 Edit Code 按钮，弹出添加代码的窗口（如图 3.15 所示）。

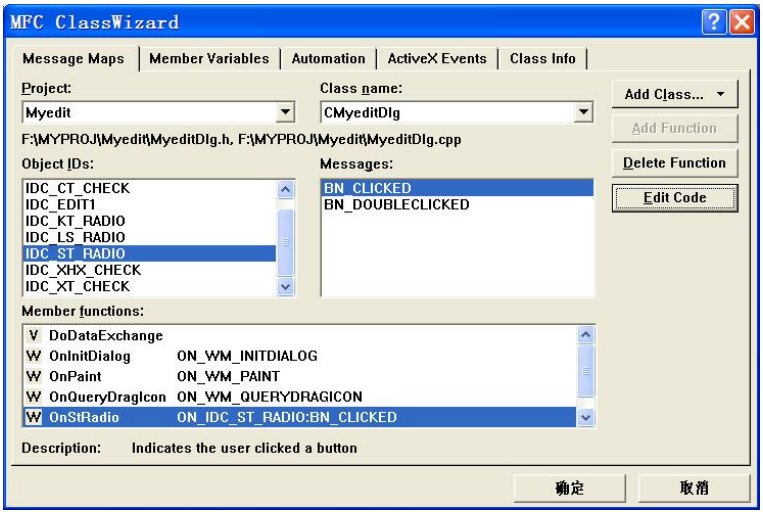

图 3.14　添加成员函数对话框

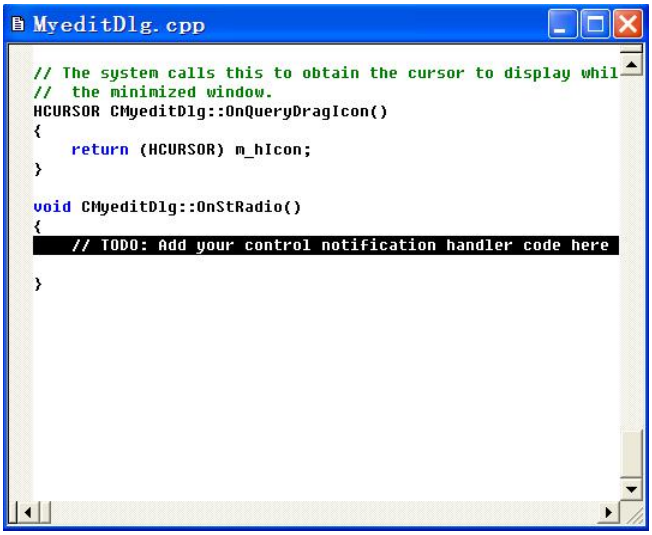

图 3.15　添加代码窗口

在上述窗口中添加如下代码：

```
void CMyeditDlg::OnStRadio()
{
    // TODO: Add your control notification handler code here（将控件的消息处理函数代码添加到下面）
fontText.CreateFont(32,32,0,0,(int)m_CT*1000,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);                    //创建字体
}
```

依次为隶书 OnLsRadio()，楷体 OnKtRadio()，粗体 OnCtCheck()，斜体 OnXtCheck()和下划线 OnXhxCheck()函数添加实现代码。各个函数的实现代码如下。

```
void CMyeditDlg::OnLsRadio()
```

```
{
    // TODO: Add your control notification handler code here
temp=_T("华文隶书");
fontText.DeleteObject();
fontText.CreateFont(32,32,0,0,(int)m_CT*1000,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);              //创建字体
}

void CMyeditDlg::OnKtRadio()
{
    // TODO: Add your control notification handler code here
temp=_T("楷体_GB2312");
fontText.DeleteObject();
fontText.CreateFont(32,32,0,0,(int)m_CT*1000,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);              //创建字体
}

void CMyeditDlg::OnCtCheck()
{
    // TODO: Add your control notification handler code here
m_CT=!m_CT;                                 //粗体状态取反
fontText.DeleteObject();
if(m_CT)
fontText.CreateFont(32,32,0,0,1000,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
else
fontText.CreateFont(32,32,0,0, 0,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);
}

void CMyeditDlg::OnXtCheck()
{
    // TODO: Add your control notification handler code here
m_XT=!m_XT;                                 //斜体状态取反
fontText.DeleteObject();
fontText.CreateFont(32,32,0,0,(int)m_CT*1000,m_XT,m_XHX,FALSE,
DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
```

```
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);
}

void CMyeditDlg::OnXhxCheck()
{
    // TODO: Add your control notification handler code here
m_XHX=!m_XHX;      //下划线状态取反
fontText.DeleteObject();
fontText.CreateFont(32,32,0,0,(int)m_CT*1000,m_XT,m_XHX,
FALSE,DEFAULT_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
DEFAULT_PITCH|FF_SWISS,temp);
m_TextFont.SetFont(&fontText);
}
```

（3）添加改变文本颜色的消息处理函数

调用父窗口（对话框）的 CWnd::OnCtrlColor()成员函数来实现对文本编辑框控件重新着色。具体实现方法是在 CMyeditDlg 类中重载 OnCtrlColor()成员函数，在此函数中添加代码，指定文本控件颜色的属性，方法如下：

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，在 Message Maps 选项卡的 Class name 列表框中选择 CMyeditDlg 类，在 Object IDs 中选择 CMeditDlg，在 Messages 列表框中选择 WM_CTLCOLOR，单击 Add Function 按钮，接受默认的函数名得到 OnCtlColor 函数，如图 3.16 所示。单击 Edit Code 按钮，在弹出的代码窗口添加如下代码：

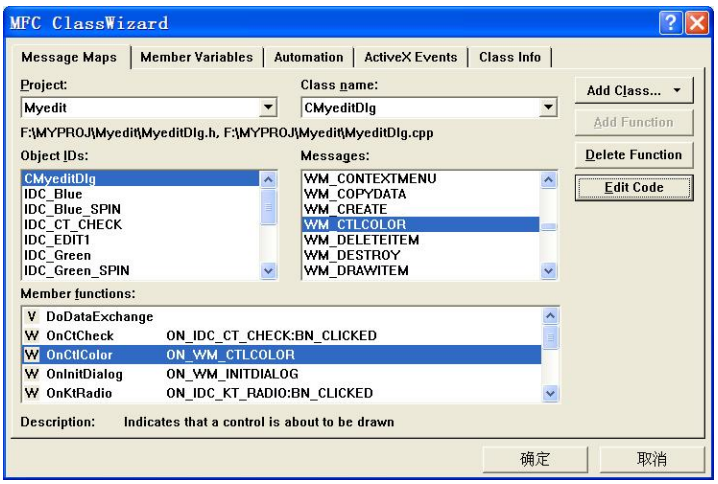

图 3.16　添加 OnCtlColor 成员函数对话框

```
HBRUSH CMyeditDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
    HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
    // TODO: Change any attributes of the DC here
    if(pWnd->GetDlgCtrlID()==IDC_EDIT1){
    //如果要设置颜色的控件是文本框 IDC_EDIT1
```

```
        UpdateData(true);   //更新 m_Red_Edit 等与文本编辑框连接的变量
            pDC->SetBkMode(TRANSPARENT);    //设置透明背景
            pDC->SetTextColor(RGB(m_Red_Edit,m_Green_Edit,m_Blue_Edit));
        // TODO: Return a different brush if the default is not desired
        return hbr;
    }
```

添加成员函数 OnChangeRed、OnChangeGreen、OnChangeBlue，动态响应微调控件的颜色设置。方法如下。

选择“查看”→“建立类向导”菜单，打开 MFC ClassWizard 对话框，在 Message Maps 选项卡的 Class name 下拉列表中选择 CMyeditDlg 类，在 Object IDs 列表框中选择 IDC_Red，在 Messages 列表框中选择 EN_CHANGE 消息，单击 Add Function 按钮，接受默认的函数名得到 OnChangeRed 函数，单击 Edit Code 按钮，在弹出的代码窗口输入以下代码完成 OnChangeRed 函数。

```
void CMyeditDlg::OnChangeRed()
{
    if(fontText.GetSafeHandle()!=NULL)          //如果 fontText 已创建
    {   m_TextFont.SetFont(&fontText,true);
        //设置字体，向父窗口发送 WM_CTRLCOLOR 消息
    }
}
```

用同样的方法完成 OnChangeGreen 和 OnChangeBlue 函数，其代码如下：

```
void CMyeditDlg::OnChangeGreen(){
    if(fontText.GetSafeHandle()!=NULL)          //如果 fontText 已创建
    {   m_TextFont.SetFont(&fontText,true);
        //设置字体，向父窗口发送 WM_CTRLCOLOR 消息
    }
}
void CMyeditDlg::OnChangeBlue() {
    if(fontText.GetSafeHandle()!=NULL)          //如果 fontText 已创建
    {   m_TextFont.SetFont(&fontText,true);
        //设置字体，向父窗口发送 WM_CTRLCOLOR 消息
    }
}
```

程序运行结果如图 3.17 所示。

图 3.17　程序运行结果

3.4 MFC 与绘图

Windows 应用程序的图形设备接口（Graphics Device Interface，GDI）被抽象化为设备环境（Device Context，DC）。设备环境又可称为设备描述表或设备文本，是 Windows 应用程序与设备驱动程序和输出设备（显示器和打印机等）之间的连接桥梁。在 Windows 应用程序向窗口用户区输出信息前，必须先获得一个设备环境对象，如果没有获得设备环境对象，则应用程序和相应信息输出窗口之间就没有任何通道。

在 MFC 类库中，CDC 类是定义设备环境对象的类，所有的绘图函数都在 CDC 类中定义，因此，CDC 类是所有其他 MFC 设备环境的基类，任何类型的设备环境对象都可以调用这些绘图函数。

要在 Windows 程序中输出图形或文字时，首先要调用该输出对象的设备环境，然后才运行与之相对应的 CDC 类的成员函数进行绘图，屏幕画面的图形坐标系在缺省情况下，设屏幕左上角的坐标值为(0,0)，坐标系的逻辑单位为像素。最后要将设备环境释放给 Windows，恢复原来的状态以备下次调用。

1. 屏幕绘图的主要函数

CDC 类的主要绘图成员函数有以下 7 种。

（1）CDC::SetPixel 函数。执行该函数可以在屏幕画面上显示一个点。其基本调用格式为：

```
pDC->SetPixel(x,y,RGB(0,0,255));
```

SetPixel()函数中的第 1、2 个参数为拟画点在屏幕上的显示坐标，第 3 个参数为拟画点颜色的 COLORREF 型变量。该例句的执行结果是在屏幕画面的(x,y)处显示一个蓝点。

（2）CDC::MoveTo 函数和 CDC::LineTo 函数。执行这两个函数可以在屏幕画面上显示一条直线，其基本调用格式为：

```
pDC->MoveTo(x1,y1);
pDC->LineTo(x2,y2);
```

由 MoveTo()函数确定直线的起始位置(x1,y1)，再由 LineTo()函数指定直线的终点(x2,y2)，最后得到所要绘制的直线段。如果接着执行 LineTo()函数则可绘制出连续的折线段。该两例句的执行结果是在屏幕画面坐标系中连接点(x1,y1)与点(x2,y2)绘制出一直线段。绘制直线段的函数不指定颜色，要改变线段的颜色可通过 GDI 的对象“笔”来实现。

（3）CDC::Ellipse 函数。执行该函数可以在屏幕画面上显示一个椭圆（或圆，圆是一种长半轴和短半轴相等的椭圆，所以绘制圆和椭圆只要一个设备环境类的成员函数）。其基本调用格式为：

```
pDC->Ellipse(x1,y1,x2,y2);
```

Ellipse()函数用当前设备环境刷子填充椭圆内部并用当前设备环境笔画椭圆的边线，该函数中的第 1 个参数是拟画椭圆的最左点横坐标，第 2 个参数是拟画椭圆的最上点纵坐标，第 3 个参数是拟画椭圆的最右点横坐标，第 4 个参数是拟画椭圆的最下点纵坐标。

（4）CDC::FillRect 函数。执行该函数可以在屏幕画面上显示一个矩形。其基本调用格式为：

```
pDC->FillRect(CRect(point.x1,point.y1,point.x2,point.y2),&newBrush);
```

FillRect()函数用当前设备环境刷子填充矩形内部（不画边线）。该函数中的第 1 个参数为拟画矩形的左上点和右下点的 CRect 类型参数（其中的 4 个值分别为拟画矩形的左上点横

坐标和纵坐标、右下点横坐标和纵坐标），第 2 个参数是为画此矩形所定义的刷子的对象（名称）。

（5）CDC::Rectangle 函数。执行该函数可以在屏幕画面上显示一个矩形。其基本调用格式为：

```
pDC->Rectangle(x1,y1,x2,y2);
```

Rectangle()函数用当前设备环境刷子填充矩形内部并用当前设备环境笔画矩形的边线。该函数中的第 1 个参数和第 2 个参数分别是拟画矩形的左上点横坐标和纵坐标，第 3 个参数和第 4 个参数分别是拟画矩形的右下点横坐标和纵坐标。

（6）CDC::Polygon 函数。执行该函数可以在屏幕画面上显示一个多边形，其基本调用格式为：

```
pDC->Polygon(pt,5);
```

Polygon()函数用当前设备环境刷子填充多边形内部并用当前设备环境笔画多边形的边线。该函数中的第 1 个参数为 CPoint 类对象的数组，第 2 个参数为数组的元素个数，此例句中的 pt 即为 CPoint 的对象数组 pt[5]。该例句的执行结果是在屏幕画面上显示一个五边形，此五边形是用直线将(pt[0].x,pt[0].y)、(pt[1].x,pt[1].y)、(pt[2].x,pt[2].y)、(pt[3].x,pt[3].y)、(pt[4].x,pt[4].y)等五个点首尾顺序连接而成的。

多边形的内部填充有两种模式：WINDING 和 ALTERNATE，缺省情况下为 ALTERNATE 模式。当填充模式设置为 WINDING 时，系统将填充整个多边形区域；当填充模式为 ALTERNATE 时，系统将填充在每条扫描线奇数号和偶数号多边形之间的区域。调用 CDC::SetPolyFillMode()函数可以对多边形进行填充模式的设置，其调用格式为：

```
pDC->SetPolyFillMode(WINDING);或 pDC->SetPolyFillMode(ALTERNATE);
```

（7）CDC::Arc 函数。执行该函数可以在屏幕画面上显示一段圆弧线，其基本调用格式为：

```
pDC->Arc((x0-r),(y0+r),(x0+r),(y0-r),x1,y1,x2,y2);
```

Arc()函数中的第 1、2 个参数和第 3、4 个参数分别为圆弧线所在圆的外切矩形的左上顶点坐标((x0-r),(y0+r))和右下顶点坐标((x0+r),(y0-r))，第 5、6 个参数为圆弧线段起点坐标(x1,y1)，第 7、8 个参数为圆弧线段终点坐标(x2,y2)。如果把起点和终点取为同一个点，则画出一封闭圆。

2. 坐标设定

绘图函数都需要指定坐标值，即其所绘图形是在某个具体的坐标系中显示的。Visual C++系统提供了不同的坐标设定（映射）模式，见表 3.4。

表 3.4　Visual C++系统的常见映射模式

映射模式	代码坐标单位	坐标系特征
MM_TEXT	1px	设备坐标。屏幕左上角为坐标原点，X 轴向向右，Y 轴向向下
MM_LOMETRIC	0.1mm	逻辑坐标。坐标原点位置可自由设置，X 轴向向右，Y 轴向向上
MM_HIMETRIC	0.01mm	逻辑坐标。坐标原点位置可自由设置，X 轴向向右，Y 轴向向上

在默认情况下，设备环境使用 MM_TEXT 映射模式，该映射模式下的视图窗口坐标为设备坐标，其坐标系原点在用户窗口的左上角点，X 轴向水平向右，Y 轴向垂直向下，基本单位为一个像素，坐标值必须是正整数值。

以映射模式代码为参数，通过调用 CDC::SetMapMode()函数即可选定窗口屏幕的坐标系

（设定映射模式）。映射模式设定后，GDI 内部的映射机制将把传给绘图函数的坐标值转换成设备坐标值（像素），从而在屏幕用户窗口区显示相应的图形。通过调用 CDC::SetViewportOrg() 可以设定坐标系的原点位置。

设置一个屏幕逻辑坐标系的典型语句为：

```
CRect rect;                          //定义一个 CRect 类对象 rect
GetClientRect(&rect);                //得到一个视图窗口的矩形边界，
                                     //其结果值保存在 CRect 对象 rect 中
pDC->SetMapMode(MM_LOMETRIC);  //设定映射模式
pDC->SetViewportOrg(int(rect.right/2),int(rect.bottom/2));
                                     //把逻辑坐标系的原点设在用户窗口的中心（点）
```

3. 画笔的使用

Visual C++中的画笔用来绘制直线、曲线或填充图形的边线，是 Windows 图形设备接口（Graphics Device Interface，GDI）对象之一。在使用 Visual C++的画笔之前必须先创建或选择画笔对象，使用 CPen 类的成员函数 CreatePen()可以创建一只笔，而调用 CDC::SelectStockObject() 可以选择一只库笔。

创建一只笔的典型语句为：

```
CPen Mypen;                                  //定义一个画笔对象 Mypen
CPen* poldPen;                               //定义一个 CPen 类对象指针 poldPen
Mypen.CreatePen(PS_SOLID,1,RGB(255,0,0));    //生成对应于 Mypen 的笔的 GDI 对象
poldPen=pDC->SelectObject(&Mypen);           //把设备环境的笔调换成新生成的笔 Mypen，
                                             //同时返回指向原设备环境的笔的指针 poldPen
pDC->SelectObject(poldPen);                  //恢复原来的设备环境笔
Mypen.DeleteObject();                        //把 Mypen 管理的笔的 GDI 对象从系统内存中消除
```

4. 刷子的使用

Visual C++中的刷子用来给图形内部着色，它也是 Windows 图形设备接口对象之一。在使用 Visual C++的刷子之前必须先创建或选择刷子对象。使用 CBrush 类的成员函数 CreateSolidBrush()或 CreateHatchBrush()可以创建一只刷子，而调用 CDC::SelectStockObject() 可以选择一只库刷子。

创建一只刷子的典型语句为：

```
CBrush Mybrush;                              //定义一个刷子对象 Mybrush
CBrush* poldBrush;                           //定义一个 CBrush 类对象指针 poldBrush
Mybrush.CreateSolidBrush(RGB(255,0,0));      //生成对应于 Mybrush 的刷子的 GDI 对象
poldBrush=pDC->SelectObject(&Mybrush);       //把设备环境刷子调换成新刷子 Mybrush
                                             //同时返回指向原设备环境刷子的指针 poldBrush
pDC->SelectObject(poldBrush);                //恢复原来的设备环境刷子
Mybrush.DeleteObject();                      //把 Mybrush 管理的刷子的 GDI 对象从系统内存中消除
```

5. 实例

下面的实例是在窗口中绘制一些常见的图形，通过该例介绍图形绘制的一般方法。

（1）建立 MyFigure 应用程序框架。

用应用程序向导建立单文档应用程序 MyFigure。

（2）添加 MFC 计算类头文件预编译指令。

在应用程序 MyFigure 的视图类实现文件 MyFigureView.cpp 的第 5 行之后添加 MFC 计算类头文件 math.h 预编译指令。添加后 MyFigureView.cpp 文件的前 6 行局部形式为：

```
// MyFigureView.cpp : of the CMyFigureView 类的实现
#include "stdafx.h"
#include "MyFigure.h"
#include "MyFigureDoc.h"
#include "MyFigureView.h"
#include "math.h"                    //用户添加
```

（3）添加各图形绘制子函数的声明。

在应用程序 MyFigure 的视图类头文件 MyFigureView.h 中添加各绘图成员函数的声明，完成后的代码如下：

```
class CMyfigureView : public CView
{//此处省掉系统生成的若干源代码…
public:
//implements（实现）
void MyFillRect(CDC* pDC,COLORREF color,CPoint point,int length,int width);
void MyRectangle(CDC* pDC,int x1,int y1,int x2,int y2,COLORREF color);
void MyLine(CDC* pDC,int x1,int y1,int x2,int y2,int line_width,COLORREF line_color);
void MyEllipse(CDC* pDC,int x0,int y0,int radius,COLORREF color);
void MyArc(CDC* pDC,int x0,int y0,int radius,double r_start,double r_end,int width,COLORREF color);
void MyPolygon(CDC* pDC,int x0,int y0,COLORREF color);
//此处省掉系统生成的若干源代码…
}
```

（4）写入各图形绘制子函数的程序代码。

在应用程序 MyFigure 的视图类实现文件 MyFigureView.cpp 的最后写入各图形绘制子函数的程序代码。

画矩形填充图形子函数（不画边界线），调用该子函数可以在指定位置显示一个指定大小和颜色的矩形填充图形，程序代码如下：

```
void CMyFigureView::MyFillRect(CDC* pDC,COLORREF color,CPoint point,int length,int width)
{//参数意义：刷子的颜色 color，矩形的左上点 point，矩形长度 length，矩形宽度 width
CBrush my_brush(color);              //定义刷子的一个对象，并对该对象初始化
CBrush* brush0;                      //定义一个 CBrush 类对象的指针
brush0 =pDC->SelectObject(&my_brush);
//把设备环境的刷子调换成新生成的刷子，同时返回指向原设备环境的刷子的指针
pDC->FillRect(CRect(point.x,point.y,point.x+length,point.y+width),&my_brush);
//用当前刷子画矩形，不画边线
pDC->SelectObject(brush0);           //恢复原来的设备环境刷子
my_brush.DeleteObject();             //释放用户定义的刷子
}
```

画矩形填充图形子函数（画边界线），程序代码如下：

```
void CMyFigureView::MyRectangle(CDC* pDC,int x1,int y1,int x2,int y2,COLORREF color)
{//参数意义：左上点(x1,y1)，右下点(x2,y2)，刷子的颜色 color
CBrush my_brush;                     //定义刷子的一个对象
CBrush* brush0;                      //定义一个 CBrush 类对象的指针
CPen* my_pen;                        //定义一个 CPen 类对象的指针
my_pen=(CPen*)pDC->SelectStockObject(NULL_PEN);
```

```
                    //在设备环境中分配一只透明库笔，同时返回指向原设备环境的笔的指针
                    //该透明笔用于画矩形的边线（不显示边线），缺省状态下系统将用黑笔画矩形边线
my_brush.CreateSolidBrush(color);          //生成对应于 my_brush 的刷子的 GDI 对象
brush0=pDC->SelectObject(&my_brush);
//把设备环境的刷子调换成新生成的刷子，同时保存原设备环境的刷子的指针
pDC->Rectangle(x1,y1,x2,y2);               //用当前刷子画矩形，用当前笔画边线
pDC->SelectObject(my_pen);                 //释放库笔，恢复原来的设备环境笔
pDC->SelectObject(brush0);                 //恢复原来的设备环境刷子
my_brush.DeleteObject();                   //释放用户刷子
}
```

画实直线子函数，调用该子函数可以在指定位置显示一条指定粗细和颜色的直线段。程序代码如下：

```
void CMyFigureView::MyLine(CDC* pDC,int x1,int y1,int x2,int y2,int line_width,COLORREF
line_color)
{//参数意义：起点(x1,y1)，终点(x2,y2)，线宽 line_width，线色 line_color
CPen my_pen;                               //定义笔的一个对象
CPen* pen0;                                //定义一个 CPen 类对象的指针
my_pen.CreatePen(PS_SOLID,line_width,line_color);
//生成对应于 my_pen 的笔的 GDI 对象
pen0=pDC->SelectObject(&my_pen);
//把设备环境的笔调换成新生成的笔，同时返回指向原设备环境的笔的指针
pDC->MoveTo(x1,y1);                        //直线的起点
pDC->LineTo(x2,y2);                        //直线的终点
pDC->SelectObject(pen0);                   //恢复原来的设备环境笔
my_pen.DeleteObject();                     //释放 my_pen
}
```

画圆填充图形子函数，用该子函数可以在指定位置显示一个指定半径和颜色的圆形填充图形。程序代码如下：

```
void CMyFigureView::MyEllipse(CDC* pDC,int x0,int y0,int radius,COLORREF color)
{//参数意义：圆心位置(x0,y0)，半径 radius，颜色 color
CBrush my_brush;                           //定义刷子的一个对象
CBrush* brush0;                            //定义一个 CBrush 类对象的指针
CPen* my_pen;                              //定义一个 CPen 类对象的指针
my_pen=(CPen*)pDC->SelectStockObject(NULL_PEN);
        //在设备环境中分配一只透明库笔，同时返回指向原设备环境的笔的指针
        //该透明笔用于画圆的边线，缺省状态下系统将用黑笔画圆的边线
my_brush.CreateSolidBrush(color);          //生成对应于 my_brush 的刷子的 GDI 对象
brush0=pDC->SelectObject(&my_brush);
        //把设备环境的刷子调换成新生成的刷子，同时返回指向原设备环境的刷子的指针
pDC->Ellipse(x0-radius,y0-radius,x0+radius,y0+radius);
        //用当前刷子画圆，用当前笔画边线
pDC->SelectObject(my_pen);                 //释放库笔，恢复原来的设备环境笔
pDC->SelectObject(brush0);                 //恢复原来的设备环境刷子
my_brush.DeleteObject();                   //释放用户定义的刷子
}
```

画圆弧线段子函数，调用该子函数可以在指定位置显示一条圆弧线段。程序代码如下：

```
void CMyFigureView::MyArc(CDC* pDC,int x0,int y0,int radius,double r_start,double r_end,int
width,COLORREF color)
{//参数意义：圆心位置(x0,y0)，半径 radiu，起点角度值 r_start，终点角度值 r_end，线粗细 width，
 //颜色 color
CPen my_pen;                                    //定义笔的一个对象
CPen* pen0;                                     //定义一个 CPen 类对象的指针
double px0,py0,pxd,pyd,PI;
PI=3.1415;                                      //圆周率
px0=radius*cos(r_start*PI/180.0)+x0;py0=radius*sin(r_start*PI/180.0)+y0;
                                                //起点坐标值
pxd=radius*cos(r_end*PI/180.0)+x0;pyd=radius*sin(r_end*PI/180.0)+y0;//终点坐标值
my_pen.CreatePen(PS_SOLID,width,color);         //生成对应于 my_pen 的笔的 GDI 对象
pen0=pDC->SelectObject(&my_pen);
                //把设备环境的笔调换成新生成的笔，同时返回指向原设备环境的笔的指针
pDC->Arc((x0-radius),(y0+radius),(x0+radius),(y0-radius),int(px0),int(py0),
int(pxd),int(pyd));                             //圆弧线函数
pDC->SelectObject(pen0);                        //恢复原来的设备环境笔
my_pen.DeleteObject();                          //释放 my_pen
}
```

画五角星填充图形子函数，调用该子函数可以在指定位置显示一个指定颜色的五角星填充图形。程序代码如下：

```
void CMyFigureView::MyPolygon(CDC* pDC,int x0,int y0,COLORREF color)
{//参数意义：中心点坐标(x0,y0)，颜色 color
CPoint pt[6];                                   //定义顶点数组
CBrush my_brush;                                //定义刷子的一个对象
CBrush* brush0;                                 //定义一个 CBrush 类对象的指针
CPen* my_pen;                                   //定义一个 CPen 类对象的指针
my_brush.CreateSolidBrush(color);               //生成对应于 my_brush 的刷子的 GDI 对象
brush0=pDC->SelectObject(&my_brush);            //把设备环境的刷子调换成新生成的刷子
                                                //同时返回指向原设备环境的刷子的指针
my_pen=(CPen*)pDC->SelectStockObject(NULL_PEN);
                //在设备环境中分配一只透明库笔，同时返回指向原设备环境的笔的指针
                //该透明笔用于画多边形的边线，缺省状态下系统将用黑笔画多边形边线
double Angle=(720.0/57.295)/5;                  //顶点角度
for(int i=0;i<5;i++){                           //计算 5 个顶点的坐标值
pt[i].x=x0+int(sin(double(i)*Angle)*300.0);
pt[i].y=y0+int(cos(double(i)*Angle)*300.0);
}
pDC->SetPolyFillMode(WINDING);                  //该填充模式下，系统填充整个多边形区域
pDC->Polygon(pt,5);                             //多边形绘制函数
pDC->SelectObject(brush0);                      //恢复原来的设备环境刷子
my_brush.DeleteObject();                        //释放 my_brush
pDC->SelectObject(my_pen);                      //释放库笔，恢复原来的设备环境笔
}
```

（5）设置逻辑坐标并调用各图形绘制子函数。

在 MyFigureView.cpp 文件的绘制函数 OnDraw 中设置逻辑坐标系并调用各图形绘制子函数。程序代码如下:

```
void CMyFigureView::OnDraw(CDC* pDC)
{
CMyFigureDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
CRect rect;                                          //定义一个 CRect 类的对象(本次键入)
GetClientRect(&rect);                                //得到一个视图窗口的矩形边界
pDC->SetMapMode(MM_LOMETRIC);                        //设定映射模式
pDC->SetViewportOrg(int(rect.right/2),int(rect.bottom/2));  //设定坐标原点
MyFillRect(pDC,RGB(255,0,0),(0,0),600,400);          //矩形（1）子函数
MyRectangle(pDC,-300,0,0,-500,RGB(100,110,120));     //矩形（2）子函数
MyEllipse(pDC,500,-300,200,RGB(0,255,0));            //圆子函数
MyPolygon(pDC,-350,300,RGB(0,0,255));                //五角星子函数
MyLine(pDC,-600,300,600,-300,4,RGB(255,0,0));        //实直线子函数
MyLine(pDC,-600,-300,600,300,4,RGB(0,255,0));        //实直线子函数)
MyArc(pDC,100,0,800,150,200,4,RGB(0,0,0));           //圆弧线段子函数
MyArc(pDC,-100,0,800,-40,40,4,RGB(0,0,0));           //圆弧线段子函数
}
```

编译、连接后，运行该程序，结果如图 3.18 所示。

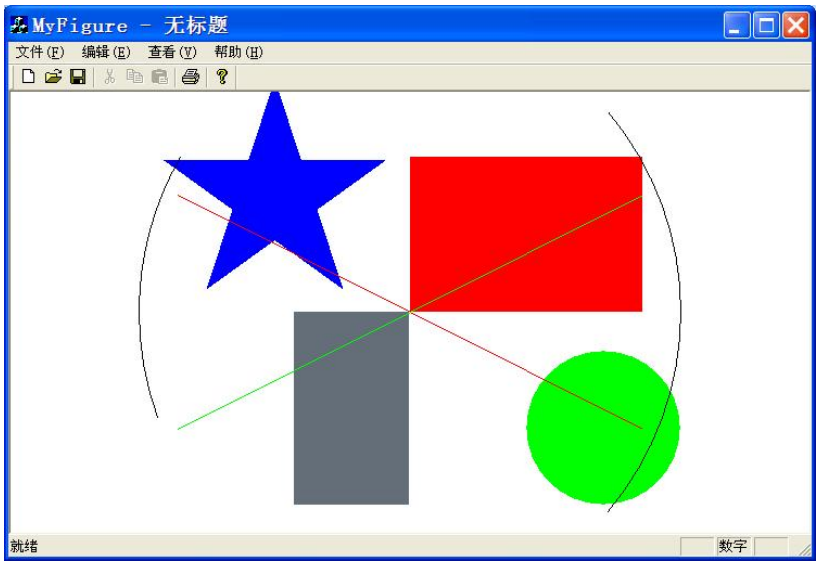

图 3.18　绘图程序运行结果

综合设计题

1．设计一个如图 3.19 所示的“显示信息”对话框程序。当单击“确定”按钮后，在右边的编辑框中显示有关信息。

图 3.19　“显示信息”对话框

（1）首先创建一个对话框程序框架，然后放置各控件，进行属性设计，各文本编辑框和复选框按表 3.5 连接变量，其余控件不需要连接变量。

表 3.5　与控件连接的变量

控件	标题	变量名	变量类别	变量类型
左文本编辑框	无	m_e1	Value	CString
右文本编辑框	无	m_e2	Value	CString
复选框	旅游	m_c1	Value	BOOL
	运动	m_c2	Value	BOOL
	音乐	m_c3	Value	BOOL

（2）“确定”按钮的消息处理函数：

```
void CEx03Dlg::OnButton1()
{
    // TODO: Add your control notification handler code here
    UpdateData(TRUE);
    CString s;
    s=m_e1;
    s+=" 爱好 :";
    if (m_c1) s+=" 旅游 ";
    if (m_c2) s+=" 运动 ";
    if (m_c3) s+=" 音乐 ";
    m_e2=s;
    UpdateData(FALSE);
}
```

2．编写一个进行算术运算的计算器程序，界面如图 3.20 所示。

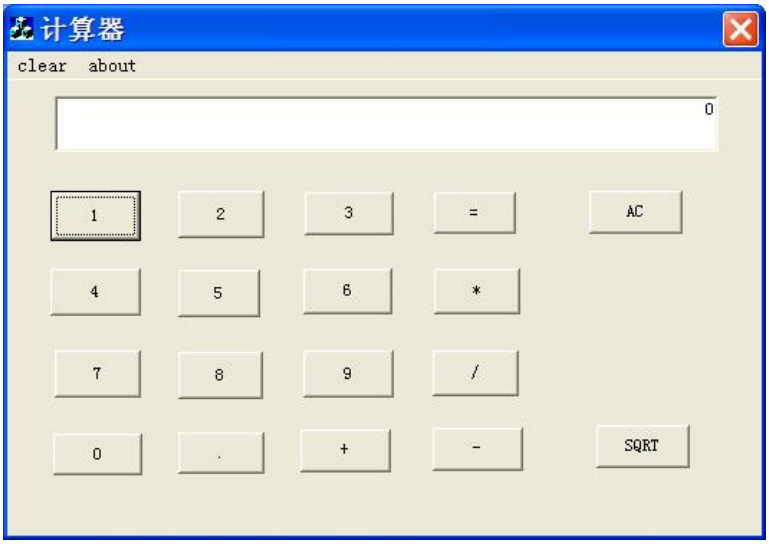

图 3.20　计算器界面

建立工程文件 calc_mfc，主要的控件 ID 及其对应的变量如表 3.6 所示。

表 3.6　控件及与控件连接的变量

控件 ID	Caption	连接变量名称	变量类型	备注
IDC_EDIT1		m_dNum	Double（Value）	编辑框
IDC_BTN_0	0			按钮
IDC_BTN_1	1			按钮
IDC_BTN_2	2			按钮
IDC_BTN_3	3			按钮
IDC_BTN_4	4			按钮
IDC_BTN_5	5			按钮
IDC_BTN_6	6			按钮
IDC_BTN_7	7			按钮
IDC_BTN_8	8			按钮
IDC_BTN_9	9			按钮
IDC_BTN_DOT	.			按钮
IDC_BTN_ADD	+			按钮
IDC_BTN_SUB	-			按钮
IDC_BTN_MUL	*			按钮
IDC_BTN_DIV	/			按钮
IDC_BTN_SQRT	SQRT			按钮
IDC_BTN_EQUAL	=			按钮
IDC_BTN_AC	AC			按钮

（1）在 IDC_EDIT1 属性的样式（Style）选项卡中，将排列文本（Align text）设置为靠

右（right）。

（2）向 CCalc_mfcDlg 类添加如表 3.7 所示的成员变量。

表 3.7　新增加的成员变量

变量名称	变量类型	变量属性
m_nDotNo	int	public
m_nDotSign	int	public
m_dPre	double	public
m_dCur	double	public
m_dNext	double	public
m_strPre	CString	public
m_strCur	CString	public
m_strNext	CString	public

（3）在 calc_mfcDlg.cpp 的 OnInitDialog()函数中添加如下代码将变量初始化：

```
m_dPre=0;
m_dCur=0;
m_dNext=0;
m_strPre=_T("");
m_strCur=_T("");
m_strNext=_T("");
m_nDotSign=0;
m_nDotNo=0;
```

（4）通过 MFC 的“建立类向导”向 CCalc_mfcDlg 类添加各按钮的 BN_CLICKED 消息处理函数，如表 3.8 所示。

表 3.8　各按钮对应的消息处理函数

按钮的 ID 值	消息函数名	按钮的 ID 值	消息函数名
IDC_BTN_0	OnBtn0()	IDC_BTN_8	OnBtn8()
IDC_BTN_1	OnBtn1()	IDC_BTN_9	OnBtn9()
IDC_BTN_2	OnBtn2()	IDC_BTN_ADD	OnBtnAdd()
IDC_BTN_3	OnBtn3()	IDC_BTN_SUB	OnBtnSub()
IDC_BTN_4	OnBtn4()	IDC_BTN_MUL	OnBtnMul()
IDC_BTN_5	OnBtn5()	IDC_BTN_DIV	OnBtnDiv()
IDC_BTN_6	OnBtn6()	IDC_BTN_DOT	OnBtnDot()
IDC_BTN_7	OnBtn7()	IDC_BTN_EQUAL	OnBtnEqual()

（5）因为程序中用到了一些数学函数，所以在 calc_mfcDlg.cpp 开始处添加包含语句 #include "math.h"。

（6）在 calc_mfcDlg.cpp 源程序文件的末尾添加公有成员函数 SetNum。这是为了处理 0~9 这些数字按钮消息，每个消息用自己的值作为参数，调用这个成员函数，代码如下：

```
void CCalc_mfcDlg::SetNum(int i)
{
    if(m_strCur=="")
    {
       if(m_nDotSign==1)
        {
           m_dCur=m_dCur+i/(pow(10,m_nDotNo));
           m_nDotNo++;
        }
     else
          m_dCur=m_dCur*10+i;
    }
    else
    {
        m_dCur=0;
        if(m_nDotSign==1)
        {
           m_dCur=m_dCur+i/10;
           m_nDotNo++;
        }
       else
          m_dCur=m_dCur*10+i;
    }
    m_dNum=m_dCur;
    UpdateData(false);    //更新与变量连接的文本编辑框的值
    m_strCur=_T("");
}
```

在 calc_mfcDlg.h 文件的 CCalc_mfcDlg 类中添加成员函数的声明：void SetNum(int);。

OnBtn0()~OnBtn9()消息处理函数简单地调用 SetNum 函数即可，如 OnBtn0()函数的代码如下：

```
void CCalc_mfcDlg::OnBtn0()
{
      // TODO: Add your control notification handler code here
      SetNum(0);

}
```

（7）在 calc_mfcDlg.cpp 源程序文件的末尾添加公有成员函数 process()。

计算器程序的关键是计算的顺序，当按下运算符键（+、-、*、/等）时，它的右操作数还是未知的，因此要保存当前的运算符选择，然后键入要操作的数字，这一数字也要保存，等到下一次按下某个运算符时再将原来保存的运算符和数字进行运算，如此循环，直到按下“=”为止。

```
void CCalc_mfcDlg::process()
{
    if(m_strPre=="+")m_dPre+=m_dCur;
    if(m_strPre=="-")m_dPre-=m_dCur;
    if(m_strPre=="*")m_dPre*=m_dCur;
    if(m_strPre=="/")
```

```
{
if(m_dCur==0) m_dCur=1;
m_dPre/=m_dCur;
}
}
```

这里仅仅将为零的除数强迫设置为 1，可以自行设置出错处理。

下面是“加法”按钮的消息处理程序。

```
void CCalc_mfcDlg::OnBtnAdd()
{
        // TODO: Add your control notification handler code here
        if(m_dPre==0)
        {
          m_dPre=m_dCur;
          m_strPre="+";
          m_strCur=m_strPre;
        }
        else
        {
        process();
             m_strPre="+";
             m_strCur=m_strPre;
        }
       m_dNum=m_dPre;
       m_dCur=0;
       m_nDotSign=0;
       m_nDotNo=0;
       UpdateData(false);
}
```

其他消息处理程序只是将运算符“+”换为相应的“-”、“*”、“/”、“=”即可。“SQRT”的消息处理请自行完成。

（8）小数点的消息处理函数。

“.”不是运算符，它的消息处理函数如下：

```
void CCalc_mfcDlg::OnBtnDot()
{
        // TODO: Add your control notification handler code here
        m_nDotSign=1;
        m_nDotNo=1;
}
```

（9）“AC”按钮的消息处理函数。

“AC”按钮的功能是清零，请参考初始化代码完成该函数。

（10）在对话框窗口中添加菜单

①选择“插入”→“资源”菜单，在弹出的对话框中选择 Menu 项，单击“新建”按钮，在菜单编辑器窗口中建立两个菜单项 Clear 和 About，其 ID 分别为 menu_id1 和 menu_id2，如图 3.21 所示。

图 3.21 “菜单项目属性”对话框

为菜单定义消息处理函数，Clear 的消息处理函数代码为：

```
void CCalc_mfcDlg::Onid1()
{
    // TODO: Add your command handler code here
OnBtnAc();
}
```

About 的消息处理函数代码为：

```
void CCalc_mfcDlg::Onid2()
{
    // TODO: Add your command handler code here
CAboutDlg dlg;
dlg.DoModal();
}
```

②最后将菜单和对话框关联，方法是右击对话框，选择快捷菜单中的“属性”，在“对话属性”窗口中，选择“菜单”列表框中刚才建立的菜单 ID，如图 3.22 所示。

图 3.22 对话框属性窗口

3．建立一个单文档窗口应用程序 draw_circle，在菜单中增加一个“图形参数设置”菜单项（如图 3.23 所示），执行该菜单项可以弹出一个对话框设置需要绘制的圆的圆心及填充颜色（见图 3.24）。退出对话框后可在文档窗口区绘制出相应的圆。

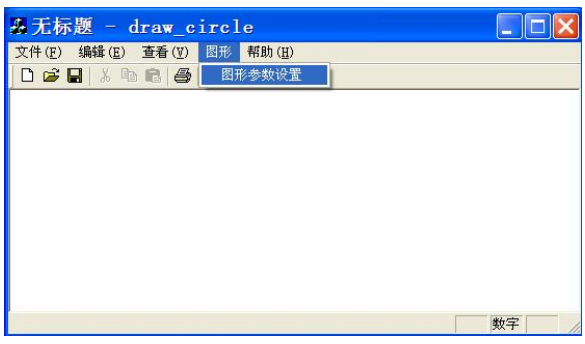

图 3.23 单文档窗口界面

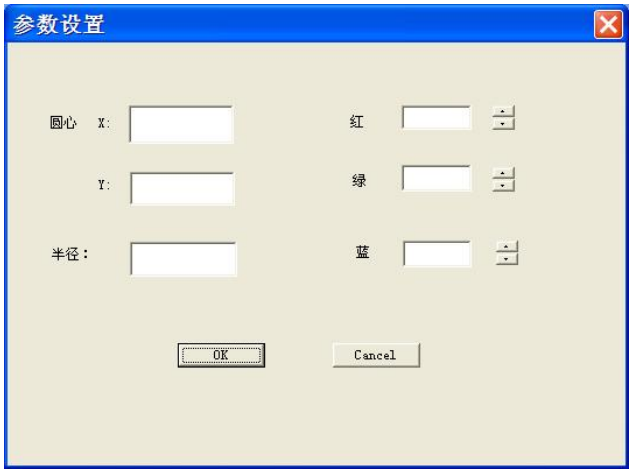

图 3.24　“参数设置”对话框

（1）创建单文档工程文件 draw_circle，为该窗口添加菜单、对话框资源。

（2）创建对话框类。新创建的对话框应有一个类与之对应并添加到工程中，在此创建 CSetDlg 类。打开新建的对话框，选择“查看”→“建立类向导”菜单，在弹出的对话框中选择 Creat a new class，在接下来的对话框中输入类名 CSetDlg，单击 OK 按钮。对话框中各控件及连接变量如表 3.9 所示。

表 3.9　对话框成员及各控件属性

控件	ID 值	属性	连接变量
X，Y Edit	IDC_X1，IDC_Y1 IDC_Radius	默认	m_intX1，m_intY1 m_Radius
Red Edit	IDC_RED	默认	m_intRed(UINT)， m_editRed(Control)
Green Edit	IDC_GREEN	默认	m_intGreen，m_editGreen
Blue Edit	IDC_BLUE	默认	m_intBlue，m_editBlue
Red Spin	IDC_SPIN1	选中“自动结伴”及“自动结伴整数”，其余默认	m_spinRed
Green Spin	IDC_SPIN2	同上	m_spinGreen
Blue Spin	IDC_SPIN3	同上	m_spinBlue

（3）为 CDraw_circleView 视图类（在 CDraw_circleView.h 文件中）添加成员变量（public）：

```
int m_startx;          //圆心 x 坐标
int m_starty           //圆心 y 坐标
int m_radius;          //记录圆半径
COLORREF m_color;      //记录圆的填充颜色
```

（4）为“图形参数设置”菜单项添加消息处理函数 OnSet（注意在 CDraw_circleView.cpp 中添加“#include CSetDlg.h”），其代码如下：

```
void CDraw_circleView::OnSet()
{    // TODO: Add your command handler code here
     CSetDlg dlg;
```

```
        if((dlg.DoModal())==IDOK){
            m_color = RGB(dlg.m_intRed, dlg.m_intGreen, dlg.m_intBlue);
            m_start.x = dlg.m_intX1 ;
            m_start.y = dlg.m_intY1;
           m_radius = dlg.m_Radius;                    //圆半径
        }
        Invalidate();                                  //使视图重画，调用 OnDraw()函数
    }
```

（5）修改视图类 OnDraw()函数

```
    void CDraw_circleView::OnDraw(CDC* pDC){
        CDraw_circleDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
        CPen pen(PS_SOLID,0,m_color);                  //构造当前颜色的画笔
        CPen *oldPen;
        oldPen = pDC->SelectObject(&pen);              //为 CDC 选中当前画笔
        CRect rect(m_start.x - m_radius,m_start.y –m_radius,
        m_start.x + m_radius,m_start.y + m_radius);
        pDC->Ellipse(&rect);
        pDC->SelectObject(oldPen);}
```

4．编写一个对话框程序，设计适当的界面，输入一元二次方程 $ax^2 + bx + c = 0$ 的系数 a、b、c，计算并输出方程的根 x_1 和 x_2。

5．编写一个对话框程序完成字符串转换，转换规则为：大写字母转换成小写；小写字母转换成大写；换行符和回车符不变；其余字符转换为“*”。要求在一个文本框中输入一个字符串，单击“转换”按钮，在另一个文本框中输出转换后的字符串。

6．设计一个如图 3.25 所示的“显示”对话框。当单击“确定”按钮后，用 AfxMessageBox 函数在窗口中显示相应教师的基本信息。

图 3.25　“显示”对话框界面

7．编写一个单文档的应用程序，并添加“绘图”菜单及“正弦”、“余弦”、“正切”、“余切”子菜单，单击绘图子菜单时在文档窗口区绘制出相应的曲线。