# 目　　录

无忧书库
http://ebook.51soft.com

**第 1 章　Visual C++ 6.0 概述** …… (1)
1.1　Visual C++ 6.0 的特点 …… (2)
1.2　Visual C++ 6.0 的新增功能 …… (2)
1.3　Visual C++ 6.0 开发环境 …… (3)
1.4　Visual C++ 6.0 的安装 …… (3)

**第 2 章　Visual C++6.0 开发环境** …… (9)
2.1　Visual C++6.0 主窗口 …… (10)
2.2　Visual C++6.0 工具栏 …… (10)
2.2.1　Standard 工具栏 …… (11)
2.2.2　Build 工具栏 …… (11)
2.2.3　Build Minibar 工具栏 …… (12)
2.2.4　Resource 工具栏 …… (12)
2.2.5　WizardBar 工具栏 …… (13)
2.2.6　Debug 工具栏 …… (13)
2.2.7　Edit 工具栏 …… (14)
2.2.8　Database 工具栏和 Atl 工具栏 …… (14)
2.3　Visual C++6.0 菜单栏 …… (15)
2.3.1　File 菜单 …… (15)
2.3.2　Edit 菜单 …… (17)
2.3.3　View 菜单 …… (19)
2.3.4　Insert 菜单 …… (19)
2.3.5　Project 菜单 …… (19)
2.3.6　Build 菜单 …… (19)
2.3.7　Debug 菜单 …… (20)
2.3.8　Tools 菜单 …… (22)
2.4　项目与项目工作区 …… (23)
2.5　资源与资源编辑器 …… (25)
2.5.1　资源编辑器 …… (25)
2.5.2　对话框编辑器 …… (27)

2.5.3 菜单编辑器 …… (28)
2.5.4 加速键编辑器 …… (30)
2.5.5 串表编辑器 …… (31)
2.5.6 版本信息编辑器 …… (31)
2.5.7 图形编辑器 …… (32)
2.5.8 工具栏编辑器 …… (32)

**第3章 C++编程基础** …… (35)
3.1 C++语言的特征 …… (36)
3.1.1 一个简单的C程序 …… (36)
3.1.2 一个简单的C++程序 …… (37)
3.1.3 C++对C的扩展 …… (37)
3.1.4 C++的面向对象特征 …… (39)
3.2 C++标识符与关键字 …… (41)
3.2.1 标识符 …… (41)
3.2.2 关键字 …… (41)
3.3 C++数据类型 …… (42)
3.3.1 类型 char …… (42)
3.3.2 类型 short …… (43)
3.3.3 类型 int …… (44)
3.3.4 类型 long …… (44)
3.3.5 类型_intn …… (45)
3.3.6 浮点类型 …… (45)
3.3.7 常量 …… (46)
3.4 数据类型转换 …… (47)
3.4.1 自动数据类型转换 …… (47)
3.4.2 强制类型转换 …… (47)
3.5 C++存储类与变量的作用域和可见性 …… (48)
3.5.1 auto 存储类 …… (48)
3.5.2 register 存储类 …… (48)
3.5.3 extern 存储类 …… (48)
3.5.4 static 存储类 …… (49)
3.5.5 内部变量和外部变量 …… (50)
3.5.6 变量作用域总结 …… (51)
3.6 操作符 …… (51)
3.6.1 算术运算操作符 …… (51)
3.6.2 关系运算操作符 …… (52)
3.6.3 逻辑运算操作符 …… (53)
3.6.4 位操作符 …… (53)

3.6.5 增量和减量操作符 …… (53)
3.6.6 赋值和复合赋值操作符 …… (54)
3.6.7 条件操作符 …… (54)
3.6.8 逗号操作符 …… (55)
3.6.9 操作符的优先级和结合性 …… (55)
3.6.10 表达式 …… (57)
3.7 高级数据类型 …… (57)
3.7.1 枚举类型 …… (57)
3.7.2 联合(union) …… (58)
3.7.3 结构声明 …… (60)
3.7.4 数组和字符串 …… (62)
3.7.5 指针 …… (66)
3.7.6 typedef 声明语句 …… (68)
3.8 控制结构 …… (68)
3.8.1 条件语句 …… (68)
3.8.2 循环语句 …… (70)
3.8.3 转移语句 …… (71)
3.9 函数 …… (71)
3.10 类与对象 …… (72)
3.10.1 类的定义 …… (73)
3.10.2 对象的定义 …… (74)
3.10.3 构造函数与析构函数 …… (76)
3.10.4 内联成员函数 …… (78)
3.10.5 this 指针 …… (79)
3.10.6 静态数据成员和静态成员函数 …… (80)
3.10.7 友元函数 …… (81)
3.11 类继承 …… (82)
3.11.1 派生类与基类 …… (82)
3.11.2 派生类的对象和构造函数 …… (84)
3.11.3 多重派生类 …… (85)
3.11.4 类的多重继承 …… (87)
3.11.5 多层派生类 …… (90)
3.12 重载(overload) …… (92)
3.12.1 函数重载 …… (92)
3.12.2 操作符重载 …… (93)

**第 4 章 Visual C++ 6.0 工具程序** …… (97)
4.1 WinDiff 应用程序 …… (98)
4.1.1 WinDiff 命令行 …… (98)

4.1.2 WinDiff 应用程序界面 …… (98)
4.2 Spy++应用程序 …… (100)
4.2.1 Spy++工具条 …… (100)
4.2.2 Spy++的视图 …… (101)
4.3 Help Workshop 工具 …… (105)
4.3.1 帮助生成器的构成 …… (105)
4.3.2 帮助工程文件结构 …… (105)
4.3.3 创建帮助主题文件 …… (107)

**第5章 MFC 基础类库** …… (109)
5.1 MFC 的特点 …… (110)
5.2 MFC 的类层次 …… (110)
5.2.1 CObject 类 …… (110)
5.2.2 CMenu 类 …… (112)
5.2.3 文档结构子层次 …… (113)
5.2.4 边框窗口子层次 …… (114)
5.2.5 控件栏子层次 …… (115)
5.2.6 视图子层次 …… (115)
5.2.7 对话框子层次 …… (115)
5.2.8 控件子层次 …… (116)
5.2.9 绘图对象子层次 …… (116)
5.2.10 CDC 子层次 …… (117)
5.2.11 其他子层次 …… (117)
5.3 MFC 的通用集合类 …… (119)
5.3.1 CString 类 …… (120)
5.3.2 CTime 类 …… (120)
5.3.3 CStringArrary 类 …… (121)
5.3.4 CStringList 类 …… (122)
5.3.5 CMapStringToString 类 …… (123)

**第6章 Windows 应用程序开发基础** …… (125)
6.1 生成 MDI 应用程序 …… (126)
6.2 向导生成的文件及作用 …… (133)

**第7章 边框窗口、文档和视图** …… (145)
7.1 边框窗口 …… (146)
7.1.1 边框窗口和视图 …… (146)
7.1.2 边框窗口与文档边框窗口 …… (146)
7.1.3 边框窗口的创建和销毁 …… (146)

7.1.4 管理当前视图 …… (147)
7.1.5 管理菜单、控件栏和加速键 …… (147)
7.1.6 定制边框窗口 …… (148)
7.2 文档与视图 …… (149)
7.2.1 文档与视图之间的联系 …… (149)
7.2.2 使用文档管理数据 …… (150)
7.2.3 在视图中显示数据及与用户交互作用 …… (150)
7.2.4 滚动与缩放 …… (151)
7.2.5 多文档类型 …… (152)
7.2.6 多视图 …… (152)
7.2.7 分割窗口 …… (152)
7.3 SDI 应用程序编程步骤 …… (152)
7.3.1 建立应用程序基本框架 …… (153)
7.3.2 处理视图 …… (153)
7.3.3 处理文档 …… (158)
7.3.4 串行化处理 …… (163)
7.3.5 SDI 应用程序编程思路 …… (165)
7.3.6 完整程序清单 …… (166)

**第 8 章 绘图程序的实现** …… (189)
8.1 生成初始的应用程序框架 …… (190)
8.2 基本概念 …… (190)
8.2.1 设备描述表 …… (190)
8.2.2 坐标系统 …… (191)
8.2.3 绘图工具 …… (191)
8.2.4 绘图函数 …… (193)
8.2.5 映射模式 …… (196)
8.2.6 绘图模式 …… (199)
8.2.7 设置背景颜色 …… (200)
8.2.8 设置背景模式 …… (200)
8.3 文本处理 …… (201)
8.3.1 文本输出函数 …… (201)
8.3.2 设置文本属性 …… (202)
8.3.3 获取字符属性 …… (203)
8.3.4 使用字体 …… (203)
8.4 点位图及其操作 …… (205)
8.4.1 点位图操作常用函数 …… (205)
8.4.2 点位图的装入 …… (207)
8.4.3 点位图的显示 …… (209)

8.5 图标 …… (209)
8.6 光标 …… (210)
8.7 打印和打印预览 …… (211)
8.8 绘图功能的实现 …… (211)

**第9章 对话框和控件** …… (221)
9.1 对话框 …… (222)
9.1.1 模态对话框 …… (222)
9.1.2 非模态对话框 …… (223)
9.1.3 创建对话框资源 …… (223)
9.1.4 创建对话框类 …… (224)
9.1.5 创建并显示对话框 …… (225)
9.1.6 处理对话框消息 …… (225)
9.1.7 关闭对话框 …… (225)
9.1.8 管理对话框的 MFC 函数 …… (226)
9.2 控件 …… (226)
9.2.1 标准控件 …… (227)
9.2.2 附加的控件类 …… (232)
9.2.3 管理对话框控件的 MFC 函数 …… (233)
9.3 控件栏 …… (235)
9.3.1 工具栏 …… (235)
9.3.2 状态栏 …… (236)
9.3.3 对话栏 …… (236)
9.3.4 CControlBar 类 …… (237)
9.4 对话框编程 …… (237)
9.4.1 创建对话框模板资源 …… (237)
9.4.2 创建对话框类 …… (238)
9.4.3 定义成员函数 …… (239)
9.4.4 定义消息处理函数 …… (242)
9.4.5 修改对话框类代码 …… (243)
9.4.6 创建对话框类对象并显示对话框 …… (245)
9.4.7 创建并运行应用程序 …… (249)
9.4.8 完整的源程序清单 …… (250)

**第10章 动态链接库** …… (283)
10.1 什么是动态链接库 …… (284)
10.2 利用向导开发动态链接库 …… (285)
10.3 DLL 应用示例(DLL 部分) …… (286)
10.4 DLL 应用示例(调用部分) …… (290)

10.5　动态链接库与静态链接库应用的进一步比较 …… (291)

**第 11 章　ActiveX 控件及应用** …… (293)
11.1　ActiveX 控件应用程序的开发 …… (294)
11.1.1　开发 ActiveX 控件 …… (294)
11.1.2　事件、方法和属性 …… (296)
11.1.3　ActiveX 控件与 MFC 类库 …… (298)
11.2　程序代码及其作用分析 …… (298)
11.3　定制实现特定功能的 ActiveX 控件 …… (312)
11.3.1　改变控件的外观 …… (312)
11.3.2　实现特定事件处理函数 …… (315)
11.4　测试 MyActiveX 控件 …… (318)

**附录　命令与快捷键** …… (321)

# 第 1 章

# Visual C++ 6.0 概述

随着计算机多媒体技术和图形图像技术的发展,可视化技术受到了广泛重视,直接面向对象的可视化技术受到广大计算机专业人员和非专业人员的喜爱,越来越多的计算机程序员开始研究和应用可视化技术。

可视化技术包含两方面的含义:一是软件开发阶段的可视化,即编程可视化。二是利用计算机图形技术和方法,对大量的数据进行处理,并用图形图像的方法形象地加以显示。

## 1.1 Visual C++ 6.0 的特点

Visual C++ 6.0 提供了用于开发 Windows(包括 Windows 95 和 Windows NT)环境下的应用程序的简捷、快速、实用的开发环境。利用 Visual C++ 6.0 开发 Windows 应用程序具有很高的效率。

Visual C++ 6.0 提供了 MFC 类库,开发者只需做少量工作即可得到功能齐全的 Windows 应用程序。与使用 C 和 Windows SDK 开发 Windows 应用程序相比,使用 Visual C++ 6.0建立一个完美的 Windows 应用程序所花费的时间要少得多。

Visual C++ 6.0 提供了一个高度集成的工具集,使得在开发应用程序的全过程中都保证了较高效率。集成化便于程序开发,开发者可以同时在诸如编辑、建立、调试等不同任务之间快速切换,甚至可以同时执行。

图形化的可视特性使得 Visual C++ 6.0 简单易学。由于 Visual C++ 6.0(包括所有工具)是完全基于 Windows 的,因此,Windows 的所有优越性(一致性运算、多任务、多线程以及可嵌入字体等)它都具备。另外,丰富的文档、样本代码、联机信息等可以帮助各个层次的开发人员。

Visual C++ 6.0 包括了 Microsoft 的代码优化技术,此项技术保持了 Microsoft C++ 所有的优化属性,并作了进一步的完善,因而,用 Visual C++ 6.0 开发的程序运行速度更快。

## 1.2 Visual C++ 6.0 的新增功能

Visual C++ 6.0 在界面上与 Visual C++ 5.0 没有太大的差别,但在实现技术上与 Visual C++ 5.0 相比作了许多改进,主要表现在以下几个方面:

1. 编译器和连接器的改进

Visual C++ 6.0 编译器不仅支持 COM(对象控件模型)应用程序开发,还进一步简化了 COM 应用程序的开发过程;Visual C++ 6.0 编译器在代码生成方面进一步作了优化,使得目标程序代码更紧凑,运行速度更快;Visual C++ 6.0 增加了对不同微处理器的支持,在生成应用程序时可以根据微处理器的不同产生不同目标代码,这可通过新增加的编译开关/G3,/G4/,/G5,/G6 来实现;新增开关/EH,则增强了 Visual C++ 6.0 对异常的处理能力。

2. 更加完善的 MFC 库

Visual C++ 6.0 的类库在对于 Internet 和数据库的支持方面也作了较大改进。首先

Visual C + + 6.0 允许开发典型的基于 Internet 的应用程序，允许异步下载文件和设置应用程序的属性，并且在任务完成后，应用程序会自动释放系统资源供其他应用程序使用。总之 Visual C + + 6.0 使得用户的应用程序和 Internet 紧密地结合在一起；其次 Visual C + + 6.0 的 MFC 库增加了对 DAO 的支持，并将原来的 ODBC API 函数进行了封装，提供了一系列 ODBC 类，以支持 ODBC 3.0 标准。

3. 与开发环境同步进行的改进和提高

Visual C + + 6.0 与 Windows 98 和 Internet Explorer 4.0 相伴而生，所以 Visual C + + 6.0 也提供了对它们的支持，主要表现在支持 Internet Explorer 4.0 控件，完善了对 OLE，ODBC，ADO 的支持，支持动态 HTML、HTML 帮助系统以及类似于 Internet Explorer 的应用程序和界面。

## 1.3　Visual C + + 6.0 开发环境

Visual C + + 6.0 开发环境 Developer Studio 由在 Windows 95 或者 Windows NT 环境下运行的一套集成工具所组成，包含文本编辑器(Text Editor)、资源编辑器(Resource Editor)、项目建立工具(Project Build Faculties)、优化编译器(Optimizing Compiler)、增量连接器(Incremental linker)、源代码浏览器(Source Code Browser)、集成调试器(Integrated Debugger)和图形浏览器条(the Visual Wordbranch)。此外，Visual C + + 6.0 还提供了几个附加的工具，如进程观察器 Spy + +，文件夹比较工具 Windiff 等。

有了 Microsoft Visual C + + 6.0，很容易在个人计算机上开发基于 Windows 的应用程序、数据库应用程序、Internet 应用程序。因为编写基于文本的应用程序利用了原 C + + 的方法，而 Visual C + + 6.0 中又提供了预先写好的功能很强的类(Classe)集，这样极大地简化了 Visual C + + 6.0 程序的编写过程。

借助 Visual C + + 6.0 的 MFC 类库和 Visual C + + 6.0 的交互工具，用户不必编写代码就能创建一个 C + + 应用程序。

## 1.4　Visual C + + 6.0 的安装

运行 Visual C + + 6.0 所需的软、硬件配置应满足以下要求：

- Windows 95 或 Windows NT 及更高版本的 Windows 系统且安装有 4.0 以上版本的 Internet Explorer。
- 个人计算机，配有 Pentium 处理器。
- 8MB 以上内存，建议使用 16MB 内存或更大内存配置。
- 最少 200MB 的可用硬盘空间，更大的硬盘空间，可以安装更多的选件。
- 高密软盘驱动器。
- 800 * 600VGA 以上的显示器。

• CD-ROM 驱动器。

使用 Visual C + + 6.0 开发应用程序之前，必须先用安装程序将 Visual C + + 6.0 的开发环境安装到计算机上。

在 Windows 95 中安装 Visual C + + 6.0 企业版(Enterprise Edition)的过程如下：

(1) 启动 Windows 95。

(2) 在 CD-ROM 驱动器中插入 Microsoft Visual Studio 6.0 系统光盘(Visual C + + 6.0 是其组件之一)。

(3) 单击“开始”按钮，从“开始”菜单选择“运行”命令，弹出“运行”对话框。在“打开”文本框中键入 D:Setup.exe(这里的 D 是用户的 CD-ROM 驱动器符)，单击“确定”按钮，出现如图 1-1 所示的安装初始屏幕。在此屏幕中单击 Next 按钮，进入下一步。

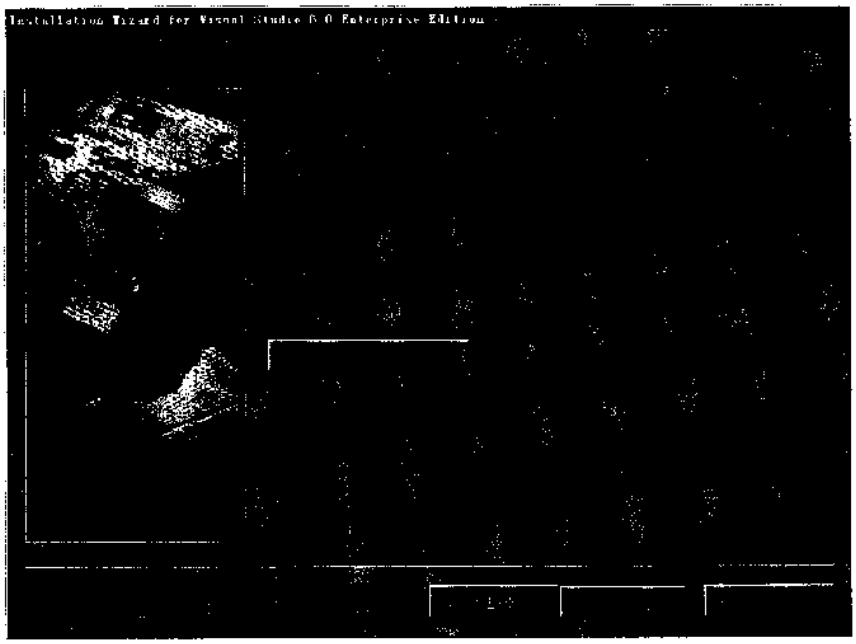


图 1-1 Microsoft Visual Studio 安装初始屏幕

(4) 在用户授权协议中(如图 1-2 所示)，选择 I accept the agreement，单击 Next 按钮，在出现的如图 1-3 所示的屏幕中输入序列号、用户名和用户所在单位，单击 Next 按钮，在新出现的安装选择屏幕中选择 Product，即按产品进行安装，然后单击 Next 按钮。进入下一步产品选择。

(5) 在产品清单(如图 1-4 所示)中选择 Visual C + + 6.0 Enterprise Edtion 后单击 Next 按钮，在接着出现的版权信息对话框中单击 Continue 按钮。

(6) 在组件清单(如图 1-5 所示)中根据需要，改变缺省的组件选项，建议增加 Graphics 选项，而不修改其他选项。单击 Continue 按钮即开始安装 Visual C + + 6.0，安装完成后，请重新启动 Windows。

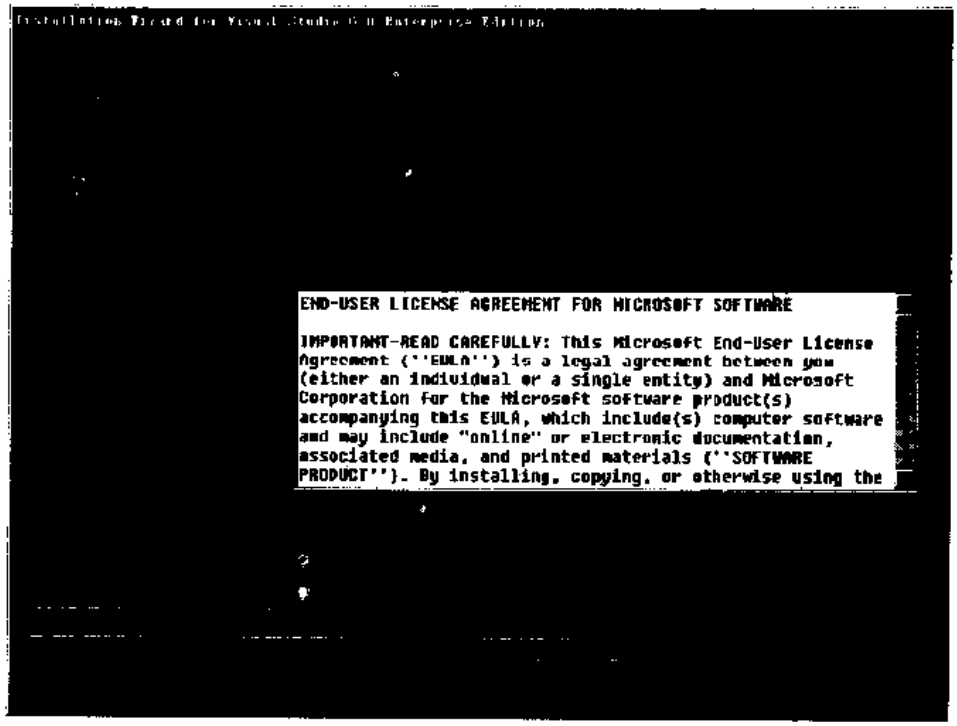



图 1-2　用户授权协议

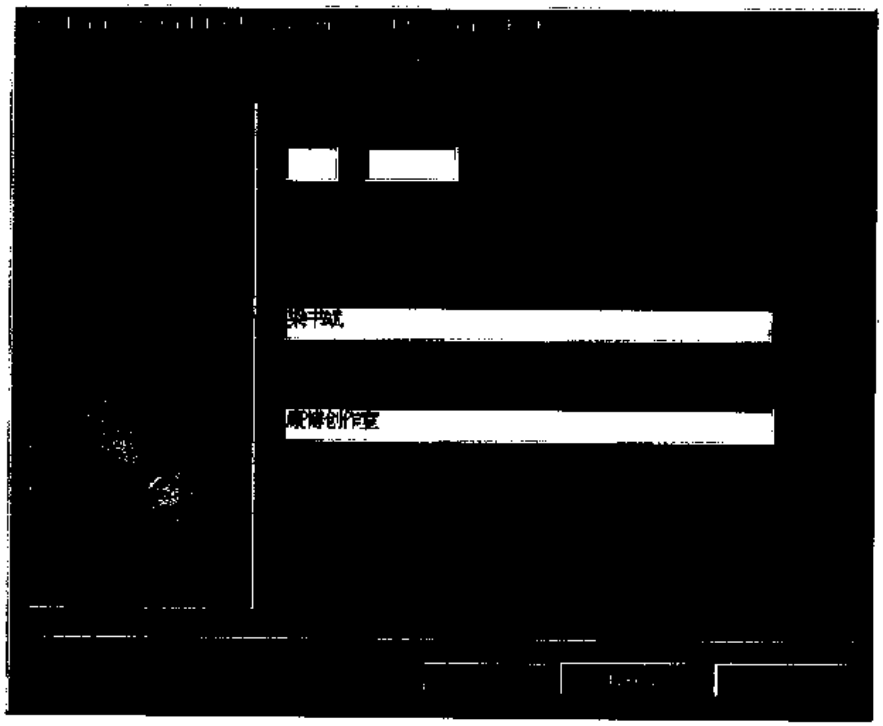


图 1-3　用户授权协议

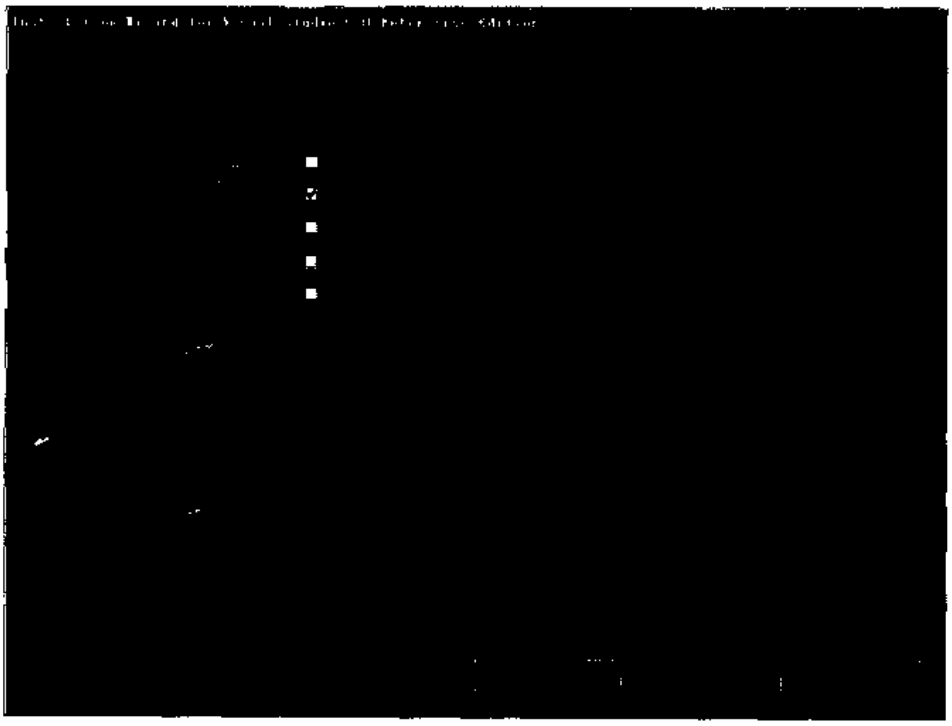


图 1-4 产品清单

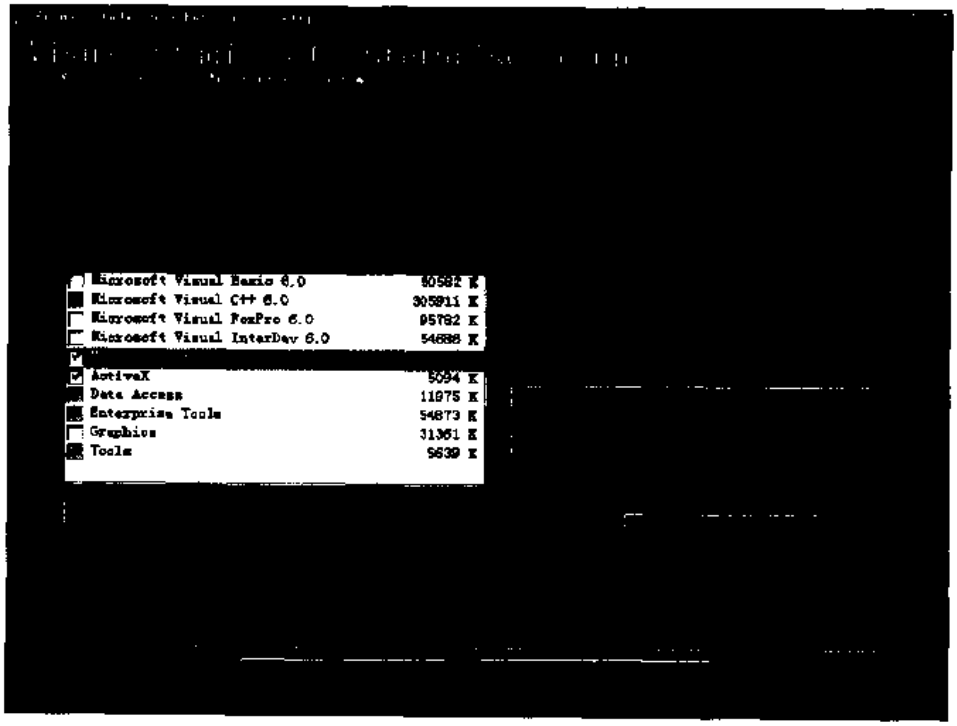



图 1-5 组件清单

**注释：**

安装前应检查硬盘是否有足够的空间来安装 Visual C++ 6.0，按照前面的安装选项需要大约 250MB 的硬盘空间。如果安装了更多的产品或组件则需要更多的硬盘空间。

**注释：**

Visual C++ 6.0 支持多语言，如果在安装过程中遇到所安装的语言支持程序与系统的不一样，安装程序会提问用户是否替换这些支持程序。

# 第 2 章

# Visual C++ 6.0 开发环境

Visual C + +6.0 提供了良好的可视化编程环境，使得在 Visual C + + 应用程序中可进行的各种操作，包括建立、打开、浏览、编辑、保存、编译、链接和调试等，点按鼠标按钮即可进行，故而非常方便。

## 2.1 Visual C + +6.0 主窗口

当正确安装 Visual C + +6.0 到 Windows 系统之后，单击“开始”按钮，指向“程序”(Programs)，“Microsoft Visual Studio 6.0”，最后单击“Microsoft Visual C + +6.0”，即可启动 Visual C + + 6.0，进入集成开发环境 Developer Studio，参见图 2-1 所示。

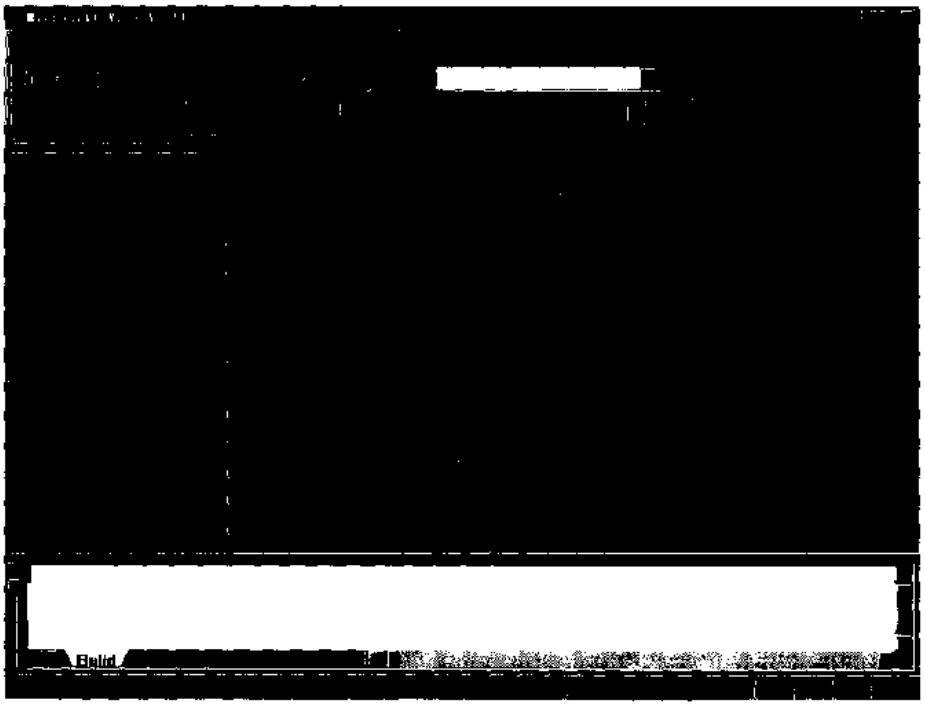



图 2-1 Visual C + +6.0 开发环境

集成开发环境的主窗口包括标题栏、菜单栏、工具栏、工作区窗口(缺省时处于船坞化状态)、源代码编辑窗口、输出窗口(缺省处于船坞化状态)和状态条。

从上图可以看出，构成 Visual C + + 集成开发环境的主窗口由 3 个子窗口组成，它们分别是：编辑窗口和船坞化的工作窗口及输出窗口。如果试着用鼠标拖动这两个船坞化窗口的边框也可以使它们成为普通的窗口。

下面分别介绍 Visual C + +6.0 集成开发环境的界面元素。

## 2.2 Visual C + +6.0 工具栏

为了使用工具栏，必须使工具栏出现在 C + + 的主窗口中，缺省时，C + + 已经使船坞化的 Standard 和 Build 两个工具栏显示在主窗口中。要使用其他工具栏，只需移动鼠标指向工

具栏的位置，然后单击右键，屏幕上弹出一快捷菜单，单击要使用的工具栏，使其前面出现√，则相应的工具栏就会出现在 C++ 主窗口中。

下面逐一介绍每一工具栏所含的各工具的作用与用法。

### 2.2.1 Standard 工具栏

Standard 工具栏含有 15 个工具，主要用于建立项目工作区及项目，如图 2-2 所示。

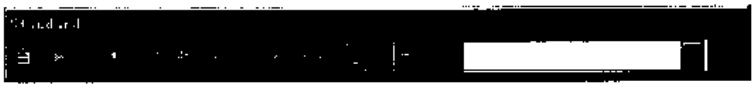


图 2-2 Standard 工具栏

各按钮的具体功能如下：

- New Text File：创建新的文本文件。
- Open：打开已有的文档。
- Save：保存文档。
- Save All：保存所有打开的文件。
- Cut：剪切选定的内容到剪贴板中。
- Copy：复制选定的内容到剪贴板中。
- Paste：在当前插入点处插入剪贴板中的内容。
- Undo：取消最后的操作。
- Redo：重复先前的操作。
- Workspace：显示或隐藏工作区窗口。
- Output：显示或者隐藏输出窗口。
- Windows list：管理当前打开的窗口。
- Find in Files：在多个文件中搜索字符串。
- Find：激活查找工具。
- Search：搜索联机文档。

### 2.2.2 Build 工具栏

Build 工具栏含有 8 个工具，如图 2-3 所示。当用户建立好应用程序项目后，可利用这些工具对项目进行编译、连接以及调试和运行。




图 2-3 Build 工具栏

各工具按钮的具体功能如下：

- Select Active Project：选择活动项目。
- Select Active Configuration：选择活动的配置。Visual C++ 提供了两种活动配置，它们

分别是 Win32 Release(基于 Win32 平台的发行版)和 Win32 Debug(基于 Win32 平台的调试版)。

- Compile:编译文件。要编译的文件必须是构成项目的一部分。
- Build:建立项目。
- Stop Build:停止建立项目。
- Execute Program:执行程序。
- Go:启动或继续程序的执行。
- Insert/Remove Breakpoint:插入或删除断点。

### 2.2.3　Build Minibar 工具栏

该工具栏包含有 6 个工具按钮,如图 2-4 所示。其功能分别相当于 Build 工具栏中的后 6 个工具按钮,主要用于建立较小的项目。



图 2-4　Build Minibar 工具栏

### 2.2.4　Resource 工具栏

Resource 工具栏含有 9 个创建资源的工具和 1 个显示资源状况的工具,如图 2-5 所示。这些工具用来帮助用户创建应用程序所需的各种资源。



图 2-5　Resource 工具栏

下面列出每个工具的名称及其快捷键,具体功能将在本章的 2.5 节中详细介绍。

- New Dialog:创建新对话框。
- New Menu:创建新菜单。
- New Cursor:创建新光标。
- New Icon:创建新图标。
- New Bitmap:创建新位图。
- New Toolbar:创建新工具栏。
- New Accelerator:创建新快捷键。
- New String Table:创建新串表。
- New Version:创建新版本信息。
- Resource Symbols:显示资源符号。

### 2.2.5　WizardBar 工具栏

WizardBar 工具栏含有 4 个工具，如图 2-6 所示。其中前 3 个实际上是列表框，最后一个是带缺省动作的列表框。利用该组工具可以方便地建立和修改应用程序中涉及的所有类。下面只列出各工具的名称及简单作用，详细功能描述见本章的 2.5 节。

- WizardBar C + + Class：在此列表框中选择要修改的类。
- WizardBar C + + Filter：在此列表框中选择要过滤的事件。
- WizardBar C + + Members：选择类成员。
- Wizard Action(Default)：根据前 3 个列表框中的选择执行不同的操作，缺省时的具体操作因前三者的不同而有所区别。




图 2-6　WizarBar 工具栏

### 2.2.6　Debug 工具栏

Debug 工具栏含有 16 个工具，如图 2-7 所示。利用这些工具可以调试已开发的项目，以便找到问题所在。只有处于调试运行状态时这一组工具才有效。



图 2-7　Debug 工具栏

- Restart：启动程序的执行，并处于调试状态。
- Stop Debugging：停止调试执行程序。
- Break Execution：中断程序的执行。
- Apply Code Change：认同程序代码的改变，即当用户在调试过程中改变了程序代码后，使用该工具使代码有效。
- Show Next Statement：显示下一条要执行的语句。
- Step Into：进入被调函数内单步执行。
- Step Over：单步执行，但跳过被调用的函数。
- Step Out：从被调用的函数内跳出，继续执行调用语句的下一条语句。
- Run to Cursor：运行到当前光标处。
- Quick Watch：快速查看当前的调试状态。
- Watch：打开一独立窗口，在其内显示用户要查看的变量的值和类型。用户需输入变量名，调试程序自动显示变量类型及其值。

• Varibles:打开一独立窗口,在该窗口内有 3 个选项卡,分别显示当前语句和上一条语句所用变量、正在执行函数的局部变量以及 this 指针所指对象的信息。

• Regisiters:打开一独立窗口,在该窗口内显示 CPU 各寄存器状态。

• Memory:打开一独立窗口,在该窗口内显示内存状态。

• Call Stack:打开一独立窗口,在该窗口内显示当前语句调用的所有函数,当前函数在栈顶。

• Disassembly:打开一独立窗口,在该窗口内显示反汇编代码。

### 2.2.7 Edit 工具栏

Edit 工具栏包含 8 个工具,如图 2-8 所示。这些工具用于源程序的编辑,提高用户查找和定位信息的效率。



图 2-8 Edit 工具栏

• Toggle Bookmark:在当前光标位置设置或取消书签。

• Next Bookmark:光标定位到下一个书签。

• Previous Bookmark:光标定位到上一个书签。

• Clear All Bookmark:清除所有书签。

• Find:在当前编辑窗口内查找。

• Increase Indent:增加缩进量。

• Decrease Indent:减少缩进量。

• Toggle Whitespace Display:锁定空白显示。

### 2.2.8 Database 工具栏和 Atl 工具栏

Database 工具栏有 3 个工具,而 Atl 工具栏只有一个工具,如图 2-9 所示。



图 2-9 Database 和 Atl 工具栏

Database 工具栏中的工具用于连接远程数据库服务器的操作。Atl 工具栏中的工具用于向用户项目添加 Atl 对象。

• Run:运行所选择的数据库对象。

• Add Data Connect:向活动项目中添加数据库连接。

• Add New Database Item:向数据库中添加新项目。

• New Atl Object:向当前项目添加新的 Atl 对象。

## 2.3　Visual C + +6.0 菜单栏

Visual C + +6.0 菜单栏包括的菜单有 File, Edit, View, Insert, Build, Debug, Tools, Window 和 Help 等 9 个主菜单,这与 5.0 版的菜单相同。下面简要介绍一下各菜单的功能。

### 2.3.1　File 菜单

File 菜单的各个命令主要完成文件的建立、保存、打开、关闭以及打印等工作。常用的命令有如下几个:

1. New 命令

选择该命令将打开一个对话框(如图 2-10 所示),通过该对话框用户可以创建新的文件、项目、工作区或者其他 Office 文档。

缺省时打开 Files 选项卡,在其中单击要创建的文件类型,然后在 File 文本框中键入要创建的文件的名字。可以创建的文件类型有:Active Server Page(服务器页文件)、Binary File(二进制文件)、Bitmap File(位图文件)、C/C + + Header File(C/C + +头文件)、C + + Source File(C + +源文件)、Cursor File(光标文件)、HTML Page(HTML 文件)、Icon File(图标文件)、Macro file(宏文件)、SQL Script File(SQL 脚本文件)、Resource Script(资源脚本文件)、Resource Template(资源模板文件)、Text File(文本文件)等。

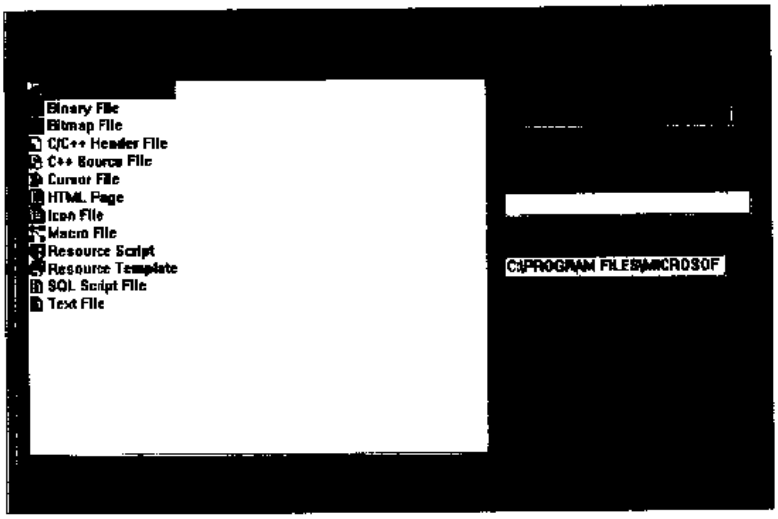



图 2-10　New 对话框的 Files 选项卡

选择 Projects 标签,将打开如图 2-11 所示的 Projects 选项卡。单击要创建的项目类型,然后在 Project name 文本框中键入项目的名字。可以创建的项目类型有:ATL COM AppWizard(ATL 应用程序)、Cluster Resource Type Wizard(簇资源类型)、Custom AppWizard(自定义的应用程序向导)、Database Project(数据库项目)、DevStudio Add-in Wizard(自动化宏)、Extended Stored Proc Wizard(基于 SQL 服务器的扩展存储进程)、ISAPI Extension Wizard(Internet 服务器或过滤器)、Makefile(旧版 C/C + +生成文件)、MFC ActiveX ControlWizard(ActiveX 控件程序)、MFC

AppWizard(exe)(MFC 可执行程序)、MFC AppWizard(dll)(MFC 动态链接库)、New Database Wizard(SQL 服务器数据库)、Utility Project(单元项目)、Win32 Application(Win32 应用程序)、Win32 Console Application (Win32 控制台应用程序)、Win32 Dynamic-Link Library(Win32 动态链接库)、Win32 Static library(Win32 静态库)。

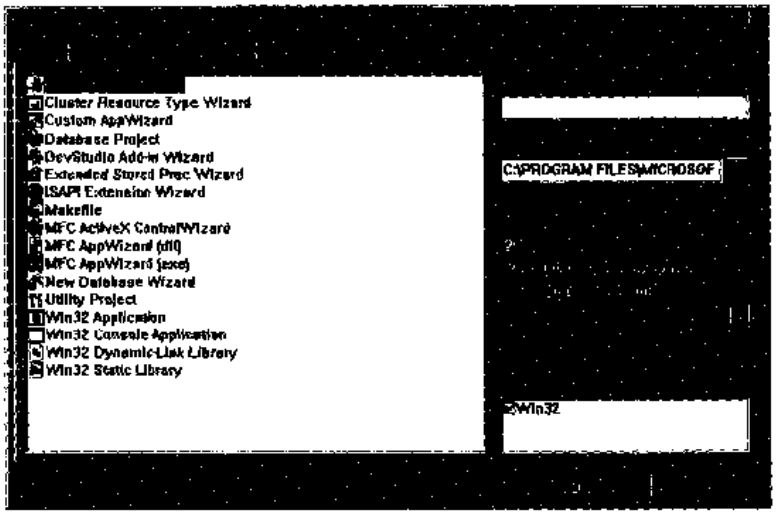



图 2-11 New 对话框的 Projects 选项卡

**注释:**

在 New 对话框的 Workspaces 和 Other Documents 选项卡中,可以创建各种类型的工作区文件和文档。

2. Open 命令

选择 Open 命令将弹出"打开"对话框用来打开已经存在的文件,如图 2-12 所示。这些文件可以是 C + + 文件、Web 文件、宏文件、资源文件、定义文件、工作区文件和项目文件等。

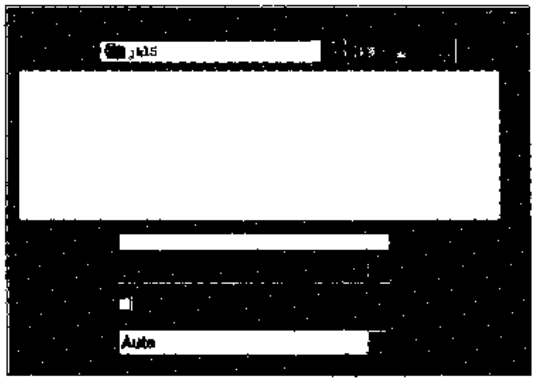



图 2-12 "打开"对话框

3. Close 命令

Close 命令关闭在活动窗口中打开的文件。要激活或选择某一窗口可以通过鼠标单击该窗口来实现。若文件未保存就关闭,则系统会提示用户保存该文件。

4. Open Workspace 命令

选择 Open Workspace 命令将弹出 Open Workspace 对话框。它主要用于打开工作区文件，也可用于打开其他类型的文件。

5. Save 命令

Save 命令用于保存活动的窗口或当前选定窗口中的文件内容。若所保存的文件是首次保存，则系统会提示 Save As 对话框；如果当前文件是只读方式打开的，则 Save 选项是灰色，表示此项不能使用，即不能修改该文件。

6. Save As 命令

该命令也用于保存文件，只是把已打开的文件用一个新的文件名加以保存而已。如果在保存文件时想要备份，则使用 Save As 选项，将该文件用另一个名字保存起来。

7. Save All 命令

Save 命令用于保存当前活动窗口或选定窗口中的文件，而 Save All 命令用来保存所有窗口内的文件内容。若某个或某些文件内容未保存过，则系统会自动提示用户输入有效的文件名。

8. Page Setup 命令

该命令用来设置和格式化打印结果。选择该命令后打开 Page Setup 对话框，如图 2-13 所示。利用该对话框可以为要打印的文档页设置标题、脚注，以及上、下、左、右 4 个边距。

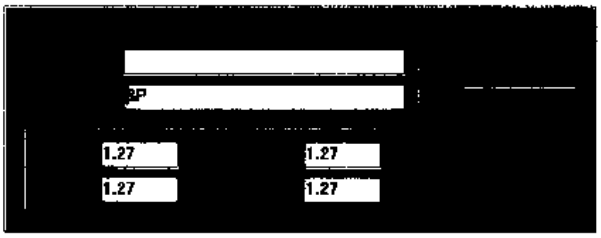



图 2-13　Page Setup 对话框

## 2.3.2　Edit 菜单

Edit 菜单包含用于编辑和搜索的命令，这些命令与其他的 Windows 应用程序中同名命令具有相似的功能。例如用于撤消操作的 Redo/Undo，用于剪贴操作的 Cut/Copy/Paste 命令。此外，Edit 菜单中还有如下一些常用的命令：

1. Find 命令

查找指定的字符串。该命令打开的 Find 对话框见图 2-14 所示。

在 Find 对话框中，可以在 Find What 文本框中输入欲查找的字符串，并设置查找的方向（Up 或 Down）。另外，可根据需要进行区分大小写字符串查找（Match case）或进行整词查找（Match whole word only）。

Regular Expression 复选框用来按照正则表达式查找文件中匹配的文本。正则表达式是指用特殊的字符序列去匹配文件中的某个文本模式。表 2-1 列出了正则表达式的查找模式。

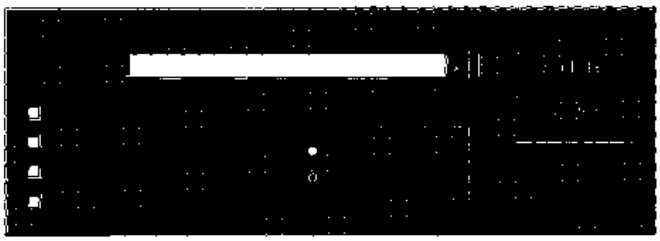


图 2-14　Find 对话框

表 2-1　正则表达式的查找模式

| 查找模式 | 含　义 |
|---|---|
| * | 匹配任意多个字母，如 Data * 1，其查找结果为 Data1，Data2，Datajkl1 |
| . | 匹配单个字符，如 Data.，其查找结果为 Data1，Data2，而不包括 Datajkl1 |
| ^ | 匹配以指定字符串开头的每一行，如 Data^，则所有以 Data 开头的行均是 |
| + | 匹配以指定字符串结束的字符串，如 + value，则结果是 ivalue，fvalue，dvalue 等 |
| $ | 匹配以指定字符串结束的每一行，如 // AFG _ MSG $，结果是以 // AFG _ MSG 结尾的所有行 |
| [] | 匹配指定集合中的字符，如 Data[A...Z]，结果可是 Dataa，Datab，而 Data1 则不是 |
| \ | 在搜索时准确匹配 \ 前的字符，如 Data[A...Zi \ 0...9]，结果可是 DataAi1，DataBi5 |
| \ { \ } | 括号间指定字符串可以任意重复。如 \ {abc \ } * result，其结果是：result，no result，nono result |

2．Go To 命令

选择 Go To 命令将弹出 Go To 对话框，可以指定如何将光标移到当前活动窗口的指定位置。

3．Bookmarks 命令

该命令可以设置、命名、删除和读取书签。

4．Breakpoints 命令

该命令用来设置、删除和查看断点。断点告诉调试器应该在何时何地中断程序的执行过程，以便检查程序代码、变量和寄存器值。断点分为位置断点、数据断点、消息断点和条件断点。

5. List Members 命令

列出光标处对象所属类的成员(包括其基类的成员)。可以通过双击相应成员名,将该成员添加到光标处。

6. Parameter Info 命令

该命令用来显示光标处函数的参数信息,以供书写函数调用时参考,从而保证函数调用时的参数表与其声明的原型相一致,减少错误发生的机会。

### 2.3.3 View 菜单

View 菜单包含用于检查源代码和调试信息的命令项,它可以用来设置按全屏方式显示窗口,或者打开工作区窗口、Output 窗口和各种调试窗口。另外,从该菜单中还可以访问 MFC ClassWizard。

### 2.3.4 Insert 菜单

Insert 菜单可以创建新的类、资源、窗体并将它们插入到文档中;可以将文件作为文本插入到文件中,也可以添加新的 ATL 对象到项目中。Instert 菜单还包括以下常用命令:

1. Resource Copy

该命令用来创建选定资源的备份,即复制选定的资源。

2. File As Text

该命令用来选择插入到文档中的文件。

3. New ATL Object

该命令用来启动 ATL Object Wizard,以便添加新的 ATL 对象到项目中。

### 2.3.5 Project 菜单

Project 菜单中的命令用于管理项目和工作区。可以选择指定的项目为工作区中的活动项目,也可以把文件、文件夹、数据链接以及可再用部件添加到项目中,还可以编辑和修改项目间的依赖关系。

### 2.3.6 Build 菜单

Build 菜单包括用于编译、建立和执行应用程序的命令。较为重要和常用的有 Compile、Build, Execute 和 Profile 等命令。

1. Compile 命令

此命令用于编译显示在源代码的编辑窗口中的源文件并检查源文件中是否有语法错误。在编译过程中若有警告或错误,则将在输出窗口中显示错误信息。可以向前或向后浏览输出窗口中的错误信息,然后按 F4 键在源代码编辑窗口中显示相应的代码行。

2. Build 命令

Build 命令用来建立项目中的所有文件。对于代码或组件结构比较复杂的项目，Build 都是很有力的工具。若建立过程中检测出某些语法错误，就将它们显示在输出窗口中。

3. Execute 命令

该命令用于运行程序。Visual C + + 将根据被运行程序的目标格式自动调用相应的环境(如 MSDOS、Windows 95 或 Windows NT 等)。

4. Profile 命令

Profile(剖视器)命令是用于检查程序运行行为的强有力的工具。它不是为了检查错误，而是为了使程序能更好地运行。选择 Profile 命令将弹出 Profile 对话框(如图 2-15)，可以进行以下剖视：

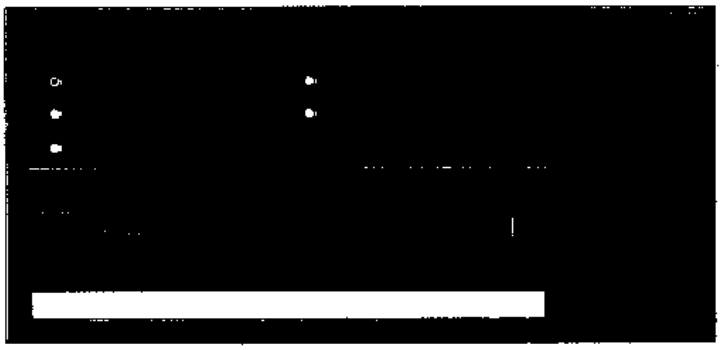


图 2-15 Profile 对话框

• Function timing(函数计时)：记录每个函数被调用的次数和在每个函数与被调函数上所花的时间。

• Function coverage(函数覆盖)：记录函数是否被调用过。在被执行的函数前用星号标记。

• Line Coverage(行覆盖)：用于确定哪一部分未被执行。在被执行的行前用星号标记。

• Merge(合并)：合并本次与上次的剖视运行结果，两次剖视类型必须相同。例如，同为行覆盖或同为函数计时。

• Custom(自定义)：定义剖视器运行时记录分析结果的批处理文件。

### 2.3.7 Debug 菜单

启动调试器后，Debug 菜单将取代 Build 菜单出现在菜单栏中。Debug 菜单中包括以下常用命令：

1. Go 命令

此命令用于调试过程中从当前语句启动或继续运行，等价于 Build 工具栏的 Go 按钮。

2. Restart 命令

系统重新装载程序到内存并放弃所有变量的当前值。

3. Stop Debugging 命令

用于中断当前的调试过程并返回正常的编辑状态。

4. Step into 命令

在调试过程中单步执行程序，而且当程序执行到某一函数调用语句时，进入该函数内部，从头单步执行。

5. Step over 命令

在调试过程中单步执行程序，但当程序执行到某一函数调用语句时，不进入该函数内部，而是直接执行该调用语句，接着再执行调用语句后面的语句。

6. Step out 命令

该命令与 Step into 配合使用。当执行 Step into 命令进入函数内部并开始单步执行时，若发现并不需要对该函数的内部进行单步调试，那么就可以使用 Step Out 命令使程序直接向下运行，直到从该函数内部返回，在该函数调用语句后面的语句处停下。

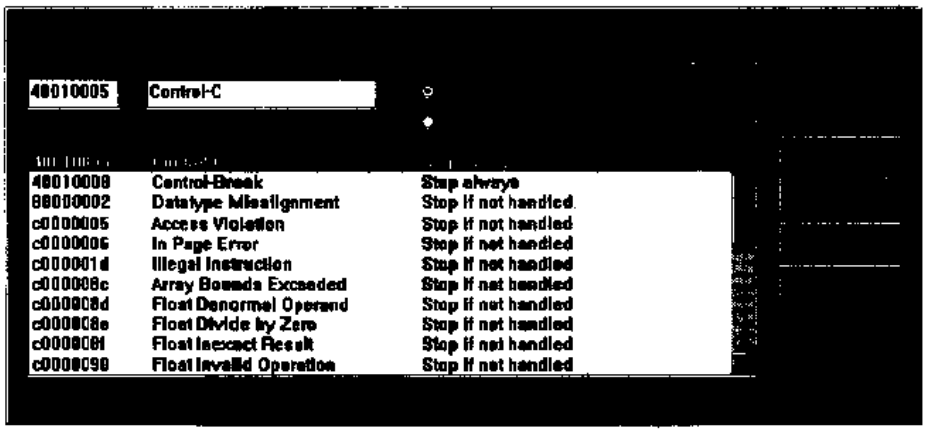



图 2-16　Exceptions 对话框

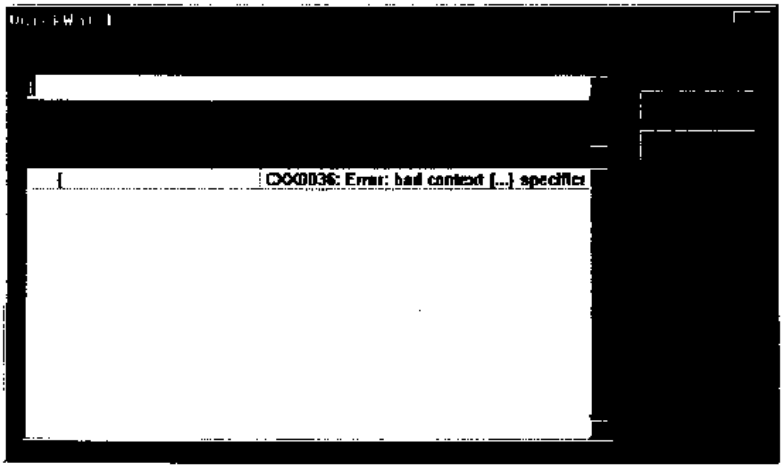



图 2-17　QuickWatch 对话框

7. Exceptions 命令

选择该命令将弹出 Exceptions 对话框(见图 2-16),显示与当前程序有关的所有异常,可以控制调试器如何处理系统异常并自定义异常。

8. Quick Watch 命令

选择该命令将弹出 QuickWatch 对话框,如图 2-17 所示。通过此对话框可以查看及修改变量和表达式,或将变量和表达式添加到 Watch 窗口。

### 2.3.8 Tools 菜单

Tools 菜单中的命令用于浏览用户程序中定义的符号、定制菜单与工具栏,激活常用的工具或更改选项和变量设置等。Tools 菜单包括以下常用命令:

1. Source Browse 命令

选择 Source Browse 命令将弹出 Browse 对话框,如图 2-18 所示。此对话框显示与程序中所有符号(类、函数、数据、宏和类型)有关的信息。使用 Browse 对话框可以查看以下信息:用户定义的符号以及引用它们的代码行;用户定义的函数、类、数据、宏和类型;所选类所继承的全部父类;由所选类派生而来的全部子类;所选函数调用的全部函数。

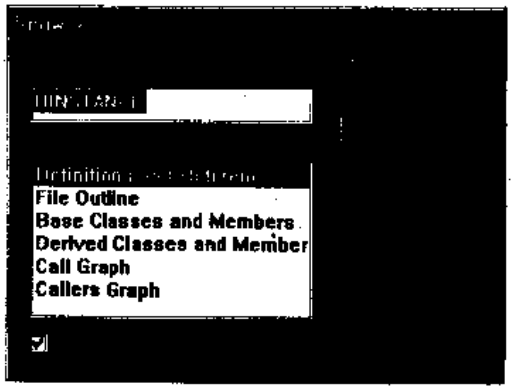



图 2-18 Browse 对话框

2. Register Control 命令

将用户开发的 OLE 控件库中的控件添加到系统注册信息库中,从而使用户的控件像其他的 OLE 服务器一样,可以被支持 OLE 的应用程序访问。

3. Error Lookup 命令

可以使用该命令在输入值的基础上恢复一个系统错误消息或模块错误消息。如果从调试器或其他支持 OLE 的应用程序上拖动了一个十六进制数或十进制数,它将自动恢复错误消息文本。

4. ActiveX Control Test Container 命令

该命令启动一个随 Visual C++ 一起发行的测试容器(Test Container)应用程序。此程序

是一个用于测试 ActiveX 控件的 ActiveX 控件容器。测试容器允许开发者通过改变控件的属性、使用其方法、激活其事件来测试其功能。另外测试容器可以显示数据捆绑通知 ActiveX 控件的日志并提供测试的持续机能工具。

5. OLE/COM Object Viewer 命令

该命令显示安装在计算机上的 ActiveX 控件和 OLE 对象以及它们支持的界面。利用它也可以编辑注册表和检查类型库。

6. Spy + + 命令

Spy + + 是一个 Win32 实用程序，可用于显示系统对象间的相互关系的图形化表示；搜索指定的窗口、线程、进程或消息；查看选定对象的属性；从视图中直接选择一个窗口、线程、进程或消息。

7. Customize 命令

此命令用于定制集成开发环境的界面。选择该命令将弹出 Customize 对话框，如图 2-19 所示，它可以对命令、工具栏、工具菜单、键盘加速键和宏进行定制。

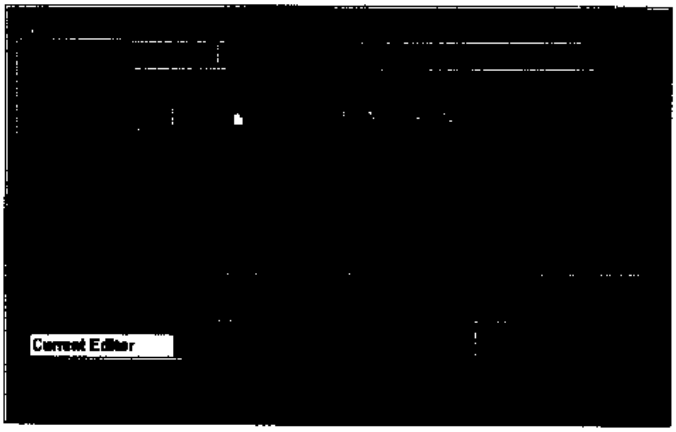



图 2-19　Customize 对话框

8、Macro 命令

该命令用于创建、编辑和运行宏。所谓宏就是用 Visual Basic Scripting Edition 语言编写的程序。正确运用宏可以简化在集成开发环境中的工作。因为在一个宏中可以结合几个命令，从而加快程序编制，或自动执行一系列复杂任务。

## 2.4　项目与项目工作区

集成开发环境以项目工作区(Project Workspace)的方式组织文件、项目和项目配置，通过项目工作区窗口可以查看和修改项目中的所有元素。每个项目工作区由工作区中的项目工作区文件组成。项目工作区文件的扩展名为 .dsw，用于描述工作区及其内容。工作区目录是项目工作区的根目录，添加到项目工作区中的项目可以位于不同的驱动器中。每一个项

目由一组项目配置文件和一组源文件组成。创建一个项目后，可以添加任何其他目录的文件到项目中。需注意的是，添加文件到项目中并不改变文件的位置，项目仅仅记录文件的名字和位置并在工作区窗口显示图标，以便说明在项目中该文件与其他文件的关系。在项目工作区中，项目之间的关系具有如下特征：

顶层项目(top level project)不依赖于任何项目，不包含任何其他项目的子项目。每个项目工作区至少有一个顶层项目。

项目与另一个项目有依赖关系。建立包含子项目的项目时，必须先建立子项目。任何项目都可以是其他项目的子项目。

创建或打开项目工作区时，项目工作区窗口中显示与项目有关的信息。工作区有 3 种面板，分别是类面板(ClassView)、资源面板(ResourceView)、文件面板(FilesView)。

1. 类面板(ClassView)

显示项目中定义的全部 C + + 类，且类以层次关系显示。利用该面板可以实现定义多个类、直接跳转到代码以及创建函数或方法声明等。扩展顶层文件夹可以显示类、扩展类或显示类成员。在项目工作区窗口单击 ClassView 选项，将以树图形式显示所有类，如图 2-20 所示。

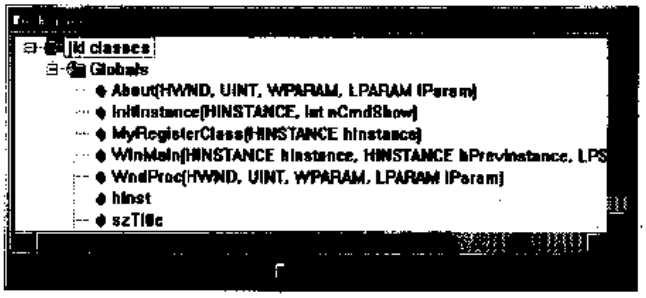



图 2-20 ClassView 面板

2. 资源面板(ResourceView)

单击 ResourceView 选项，将显示项目中包含的资源文件，如图 2-21 所示。扩展顶层文件夹可以显示资源类型，扩展资源类型可以显示其下的资源。

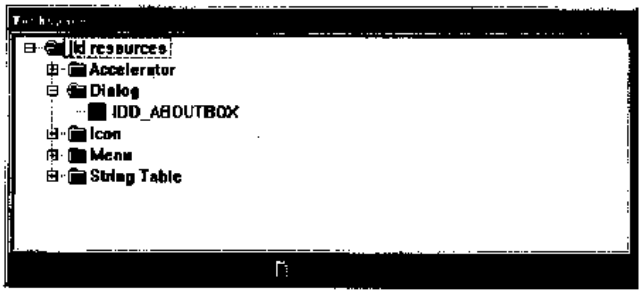



图 2-21 ResourceView 面板

3. 文件面板(FileView)

单击 FileView 选项,将显示项目之间以及组成项目的各文件之间的隶属关系,扩展顶层文件夹可以显示项目中的文件(如图 2-22 所示)。

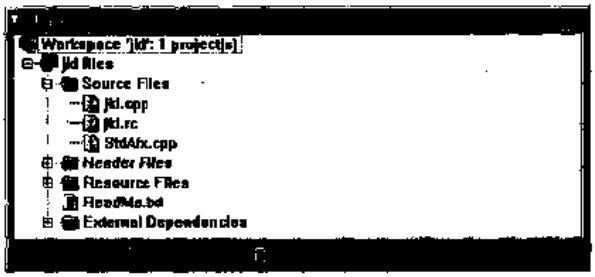



图 2-22　FileView 视图

在项目工作区中,双击某一选项时,如果该选项不是最底层选项,则自动展开下一级;如果该选项是最底层选项,则在编辑窗口打开该选项,以便进行编辑,并由集成开发环境自动识别该选项,从而调用相应的编辑器。

## 2.5　资源与资源编辑器

Visual C + +6.0 可以处理的资源包括加速键(Accelerator)、位图(Bitmap)、光标(Cursor)、对话框(Dialog)、图标(Icon)、菜单(Menu)、串表(String Table)、工具栏(Toolbar)和版本信息(Version)等。

### 2.5.1　资源编辑器

使用资源编辑器,可以创建新的资源,修改已有的资源,拷贝资源以及删除无用的资源。创建或打开资源时,系统将自动打开相应的编辑器。

1. 创建新资源

从 Insert 菜单中选择 Resource 命令,弹出 Insert Resource 对话框(见图 2-23)。从 Resource

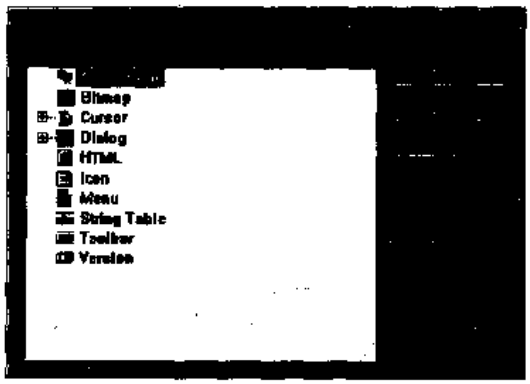



图 2-23　Insert Resource 对话框

type 列表中选择资源类型,然后单击 New 按钮即可创建新的资源。

**注释:**

也可以单击 Resource 工具栏中的相应按钮来创建新的资源。

2.查看和修改资源

打开 ResourceView 面板,单击 + 标记扩展每一分类来查看资源。可以用菜单命令复制、移动、粘贴或删除资源,也可双击打开相应的编辑器来修改资源,并运用资源属性对话框来修改资源的语言属性或条件属性。

3.资源模板

创建资源模板的方法与创建资源文件的方法基本相同,区别在于必须使用 File 菜单中的 Save As 命令将资源模板保存在 Microsoft Visual Studio \ Common \ MSDev98 \ Template 文件夹中。创建资源模板后,可在资源模板上创建新的资源。

4.资源符号

资源符号由映射到整数值上的字符串组成,用于在源代码或资源编辑器中引用资源或对象。通过资源属性框可以改变资源的符号名或符号值。具体的操作步骤如下:

(1) 在 ResourceView 视图中选择要处理的资源。

(2) 从 Edit 菜单中选择 Properties 命令或按 Alt + Enter 键,弹出相应的资源属性对话框,如图 2-24 所示。




图 2-24 资源属性对话框

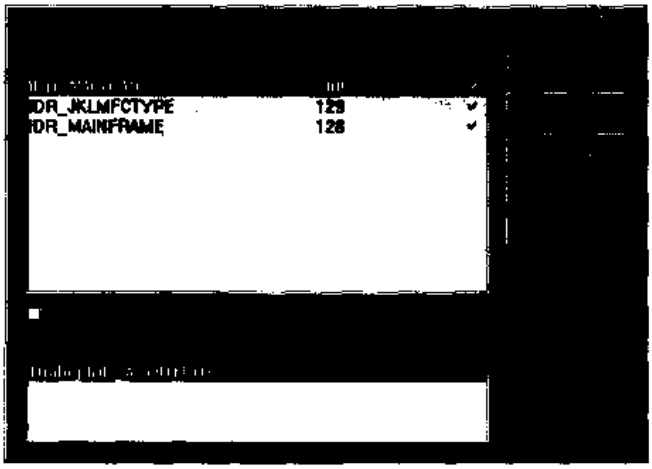



图 2-25 Resource Symbols 对话框

(3)在 ID 列表框中输入新的符号或符号值,或从已有的符号列表中选择一种符号。缺省时符号名或符号值自动保存在系统生成的资源文件 resource.h 中。当然,也可以在文本编辑器中修改 resource.h 文件来改变与多个资源或对象有关的符号。

**注释:**

符号名称前缀还用来代表资源或对象类型,如光标前缀的 IDC_;位图前缀的 IDB_等等。可以在图 2-25 所示的 Resource Symbols 对话框中查看资源符号的定义,还可以在其中创建和删除资源符号,或者快速切换到某个资源对应的资源编辑器中。

## 2.5.2　对话框编辑器

对话框编辑器(如图 2-26 所示)用于创建或编辑对话框资源或对话框模板,向对话框中添加控件或调整其布局,并测试对话框功能。另外,也可添加 ActiveX 控件,引入 Visual Basic 表单作为对话框资源,或者把对话框保存为模板以供将来使用。

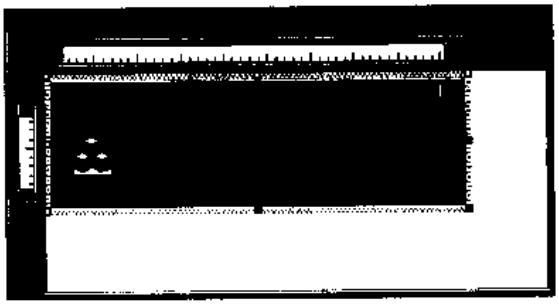


图 2-26　对话框编辑器

图 2-26 显示的是对话框编辑器窗口,一旦打开一对话框编辑器窗口,则同时出现 Controls 工具栏和 Dialog 工具栏,如图 2-27 所示。

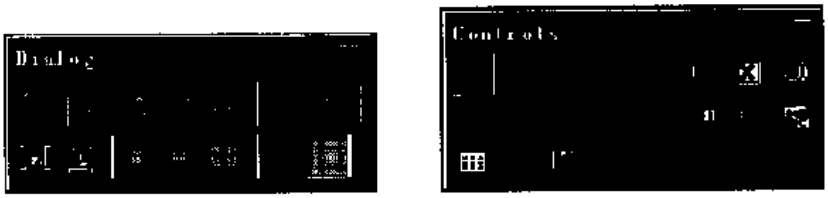


图 2-27　Dialog 工具栏和 Controls 工具栏

利用 Controls 工具栏可以向对话框添加控件。可以添加的控件有:静态文本(Static Text)、编辑框(Edit Box)、组合框(Group Box)、按钮(Button)、复选框(Check Box)、水平滚动条(Horizontal Scroll Bar)、垂直滚动条(Vertical Scroll Bar)、微调控件(Spin)、进展条(Progress)、网格(Grids)、热键(Hot Key)、列表控件(List Control)、树形控件(Tree Control)、制表控件(Tab Control)和动画(Animate),等等,还可以根据需要创建新的定制控件(Custom Control)。添加控件有以下两种方法:

- 将控件从 Controls 工具栏拖到对话框编辑器窗口的指定位置。
- 先单击 Controls 工具栏中要添加的控件,然后移动光标到要添加控件的位置,单击并

拖动鼠标。要添加多个同一类型的控件,应同时按住 Ctrl 键。选择控件后,可以对其进行移动、复制、删除或调动,也可以拖动尺寸句柄缩放控件。另外,可以使用控件的属性窗修改控件的属性。

Dialog 工具栏包括以下工具(从左上到右下):

- Test:运行对话框以测试对话框的外观和行为。
- Align Left:将选定的控件按左对齐格式放置。
- Align Right:将选定的控件按右对齐格式放置。
- Align Top:将选定的控件按上对齐格式放置。
- Align Bottom:将选定的控件按下对齐格式放置。
- Central Vertical:将选定的控件按中心垂直对齐格式放置。
- Central Horizontal:将选定的控件按中心水平对齐格式放置。
- Space Across:使选定的控件两两水平间隔相同。
- Space Row:使选定的控件两两垂直间隔相同。
- Make Same Width:使选定的控件有相同的宽度。
- Make Same Height:使选定的控件有相同的高度。
- Make Same Size:使选定的控件有相同的宽度和高度。
- Toggle Grid:在显示或隐藏网格间切换。

### 2.5.3 菜单编辑器

使用菜单编辑器可创建标准菜单和菜单项,为菜单或菜单项定义热键、加速键和状态栏提示,或者创建快捷菜单。图 2-28 为打开某一菜单资源后的菜单编辑器。



图 2-28 菜单编辑器

1. 创建菜单及菜单项

创建菜单栏中的菜单项的方法如下:

(1) 在菜单栏中选择新建菜单项的方框,或按 Tab 键(向左移)、Shift + Tab 键(向右移)或左右箭头键移到新建菜单项的方框(插入时按 Insert 键)。

(2) 输入菜单名。开始输入名字时,系统弹出 Menu Item Properties 对话框(见图 2-29)。在 Caption 文本框中输入菜单名。在相应的字母前加 &,则该字母为菜单助记符。

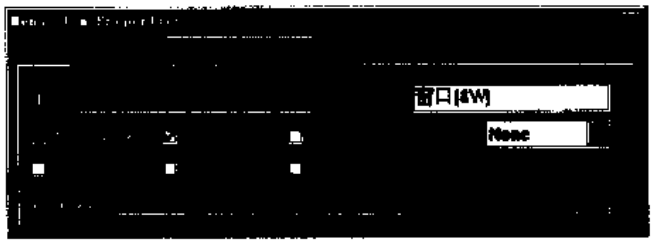



图 2-29　Menu Item Properties 对话框

添加菜单项的方法如下：

(1) 选择菜单条中的新建菜单项的方框，或者选中某个已有菜单项再按 Insert 键，新菜单项将自动插在该菜单项之前。

(2) 输入菜单项的名字。在开始输入名字时，系统弹出 Menu Item Properties 对话框。在对话框的 Caption 文本框输入菜单项名称。若为菜单定义助记符，则在相应的字母前加符号 &。

(3) 在 ID 列表框中输入菜单项的 ID 号或选取已有的 ID 号。系统可以根据选项名称自动生成一个 ID 值。

(4) 通过启用或禁止对话框中的各复选框为菜单项指定风格。

2. 定义加速键

利用菜单编辑器还可以定义加速键，具体操作步骤如下：

(1) 选择要定义加速键的菜单项，按 Alt + Enter 组合键。系统弹出 Menu Item Properties 对话框。

(2) 在 Caption 文本框中将加速键添加到菜单标题的后面。如果在菜单标题后输入转义符 \ t，则所有加速键都按左对齐方式显示。

(3) 在加速键编辑器中建立相应的加速键表条目，并赋予与菜单选项相同的 ID 号。

3. 创建快捷菜单

在应用程序中要使用快捷菜单，首先要创建它，然后再将其与应用程序代码链接。具体方法如下：

(1) 创建带空标题的菜单栏。

(2) 输入临时字符为菜单标题，以便在菜单栏中创建菜单项。

(3) 在菜单条下创建快捷菜单的菜单选项。

(4) 保存菜单资源。

(5) 添加以下代码到源文件中：

```
Menu menu;//装载并验证菜单资源
VERIFY(Menu.loadMenu(IDR _ MENU1));
(Menu * pPopup = Menu.GetSubMenu(0));
ASSERT(pPopup ! NULL);//显示菜单内容
PPopup- > TrackPopupMenu(TPM _ LEFALIGN | | TPM _ RIGHTBUTTON, x, y, Afx GetMain
```

Wnd());

创建快捷菜单的菜单资源后,应用程序代码装载菜单资源并使用函数 TrackPopup Menu()来显示菜单内容。在快捷菜单外单击鼠标,则快捷菜单消失。用户选择某个命令,则传递消息句柄给窗口。建立快捷菜单后,可以在菜单编辑器中单击鼠标右键,从弹出的快捷菜单中选择 View As Popup 命令来查看或修改所建立的菜单。

### 2.5.4 加速键编辑器

加速键通常是用键盘来执行菜单或工具栏上所用命令的快捷键。使用加速键编辑器可以添加、删除、更改和浏览项目所用到的加速键,可以查看和更改与加速键表中每个条目有关的资源标识符,还可以为某个菜单选项定义加速键。定义加速键后,可以使用 ClassWizard 为加速键命令编写要执行的代码。

图 2-30 所示为打开某一加速键表资源后的加速键编辑器。按 Ins 键或在表尾的方框中输入加速键名,将弹出 Accel Properties 对话框,如图 2-31 所示,用来在加速键表中添加新的加速键。可以在该对话框的 Key 列表框中输入键名,在 ID 列表框中输入加速键标识符。

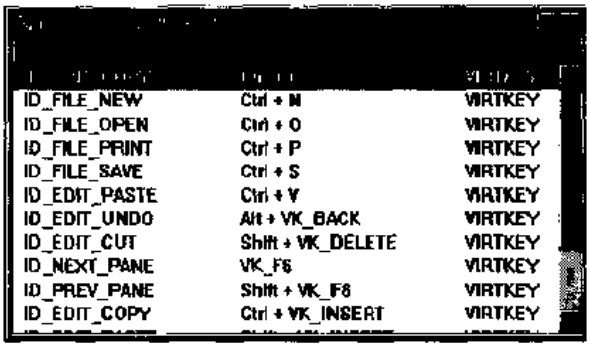



图 2-30 加速键编辑器



图 2-31 Accel Properties 对话框

Key 列表框键名必须满足以下条件:

- 0~255 间的整数,可以写成十进制,十六进制、八进制格式。
- 单个键盘字符,大写的 A~Z 或数字 0~9 既可解释成 ASCII 码值,也可看成虚拟键值,其他字符都看成 ASCII 字符。
- 单个 A~Z 间的字母键所产生的组合键的 ASCII 值。
- 任何合法的虚拟键标识符。可以单击 Key 列表框右侧的向下箭头来选择标准的虚

拟键标识符。

**注释：**

当输入 ASCII 值时，不能通过符号^和控制键的组合来产生相应的虚拟键值。

### 2.5.5　串表编辑器

串表是一种 Windows 资源，包含应用程序用到的串表的 ID 号、值和标题。每个应用程序只能有一个串表。

串表编辑器用于编辑应用程序的串表资源。使用串表编辑器，可以浏览串表、向串表添加新的串，从串表删除某一串，将串从某一段移到另一段，将串从某一资源文件移到另一资源文件，修改串及其标识符等。图 2-32 所示是打开某一应用程序的串表资源后的串表编辑器。

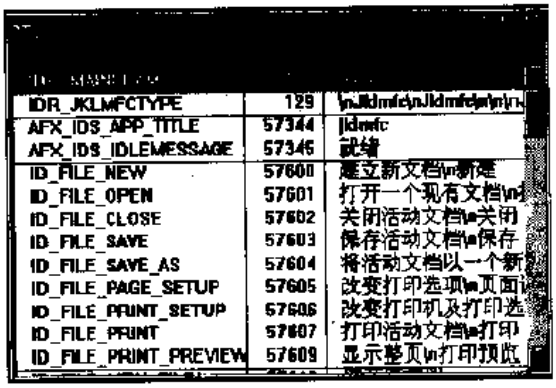



图 2-32　串表编辑器

### 2.5.6　版本信息编辑器

版本信息主要由公司名称、产品标识、产品版本号、版权和商标注册号组成。版本信息编辑器是用于编辑和维护版本信息的工具。每个应用程序只有一个版本信息资源，其名称为 VS_VER_SION_INFO。在应用程序中调用函数 GetFileVersionInfo 和 VerQueryUaWe 可以访问版本信息。图 2-33 是打开某一应用程序的版本信息资源后的版本信息编辑器。

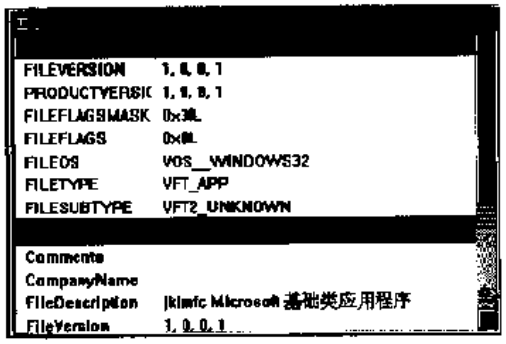



图 2-33　版本信息编辑器

### 2.5.7 图形编辑器

图形编辑器(如图 2-34 所示)由一套功能强大的绘图工具组成,用于绘制位图、图标和光标。图形编辑器打开后,将自动激活 Graphics 工具栏和 Colors 工具栏,如图 2-35 所示。Graphics 工具栏由选择(Select)、自由选择(Select Rogion)、颜色拾取(Select Color)、橡皮擦(Erase)、填充(Fill)、放大器(Magnify)、铅笔(Pencil)、画刷(Brush)、喷枪(Airbruah)、画线(line)、曲线(Curve)、文字(Text)、空心矩形(Rectangle)、边界矩形(Outlined Rectangle)、实心矩形(Filled Rectangle)、空心圆角框(Round Rect)、边界圆角框(Outlined Round Rect)、实心圆角框(Filled Round Rect)、空心椭圆(Ellipse)、边界椭圆(Outlined Ellipse)和实心椭圆(Filled Ellipse)等 21 个工具组成,供用户绘制图形、输入文本、擦除并管理视图。Colors 工具栏供用户选择绘图时所用的颜色。

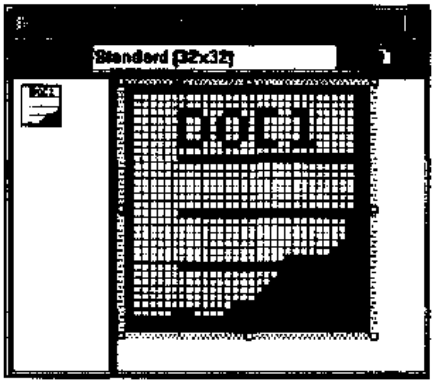



图 2-34 图形编辑器



图 2-35 Griphics 工具栏和 Colors 工具栏

### 2.5.8 工具栏编辑器

工具栏编辑器用于创建工具栏资源并可以将已有位图转换为工具栏资源。工具栏编辑器以图形方式显示要处理的工具栏及已被选择的工具按钮图像。图 2-36 所示为打开某一工具按钮后的工具栏编辑器。

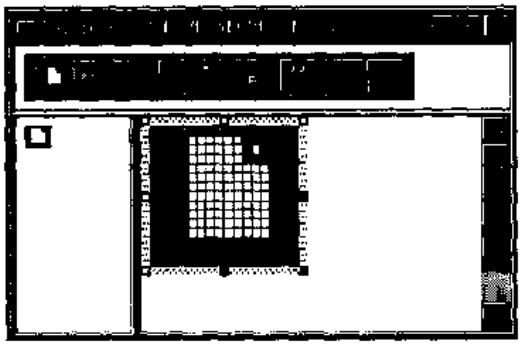


图 2-36　工具栏编辑器

# 第 3 章

# C++编程基础

在这一章我们将简单介绍 C++语言的基本语法和具体应用。如果读者已经非常熟悉 C++语言,则可以跳过这一章,也可以在学习 C++的过程中遇到了语法方面的问题时再回过头来查阅本章内容作为参考。

学习高级程序设计语言一般需要掌握它的数据类型、控制结构、函数调用规则以及输入和输出,C++也不例外。C 语言的特色主要体现为:简洁的关键字集合、快速的可执行能力、有限的类型检查、模块化结构、高度的兼容性等。C++继承了 C 的所有优点,更增加了对面向对象程序设计的支持。实际上,也主要是由于此特点,才使 C++取代了 C。

## 3.1 C++语言的特征

本节我们将通过介绍一个简单的 C 程序和一个 C++程序来概括 C 和 C++程序的特征,比较两者之间的共同点和不同点,并着重强调 C++区别于 C 的特性。

### 3.1.1 一个简单的 C 程序

程序清单 3-1 为一个简单的 C 程序。

**程序清单 3-1:一个简单的 C 程序**

```
/*这是一个简单的C程序,文件名是:Sample.c */
#include <stdio.h>
int main()
{
printf("Hello World");
return 0;
}
```

让我们简单解释一下这个 C 程序:

第一行是 C 的注释,C 要求注释以"/*"开始,以"*/"结束,二者之间的所有内容都会被编译器忽略,C 不允许注释符嵌套。

第二行是一个 C 独有的特性,它是诸多被称为预处理语句中的一条:

```
#include <stdio.h>
```

预处理语句实际上就是预编译指令。该语句指示编译器在源代码的指定行(即该语句所在行)把 stdio.h 中的代码包含到源程序中。

接下来的代码是一个函数定义:

```
int main()


    return 0;
```

所有的C程序都是由函数调用组成的。每一个C程序都必须有一个名为main()的函数,通常main()函数是程序执行的起始点,main()函数的return语句是程序执行的结束点,如果主函数main()没有return语句,则在主函数的结束处自然结束。在main()左边的int定义了函数的返回类型,具体值由return语句返回。此外C的每一个函数都可以带有参数,在这个例子中主函数main()没有参数。在main()函数头的后面是函数体。函数体由"{"开始,到"}"结束,"{"和"}"可以嵌套,但要注意正确地匹配。main()函数体一般由一系列变量定义和函数调用组成,变量定义部分用来定义main()函数用到的变量,程序要实现的具体功能则由被调用的函数来实现。

### 3.1.2 一个简单的C++程序

C++继承了C的所有特点,同时也增加了许多C所没有的新特征,程序清单3-2是一个简单的C++源程序。

**程序清单3-2:一个简单的C++程序**

```
//这是一个简单的C++程序
#include <Iostream.h>
int main(){
    cout<<"Hello World!";
return 0;
}
```

此程序实现了与上一个程序相同的功能,两者都是在标准输出文件(在这两个例子指控制台)中输出Hello World,与上面的C程序比较,这个C++程序有以下几个不同之处:

(1) C++中引入了单行注释符"//",此符号之后的内容在编译时被编译器忽略,此注释符只能用在一行中。如果需要多行注释,则必须在每一行的行首都加上"//"。不过,C++程序中仍然可以使用传统的C注释符"/*"和"*/"。

(2)输入/输出功能除了仍然可以使用原来的输入/输出函数外,C++增加了独具特色的流支持。

C++不同于C的特点还有很多,可以认为C++是C的扩展,也就是说C++是C的超集。

### 3.1.3 C++对C的扩展

C++语言在面向对象方面与结构化方法方面显著地扩展了C,主要包括以下几方面:

1. 枚举名

C++中枚举可以命名,一个枚举名就是一个类型名字,因此不必在类型名前加标识符enum。

2. 结构和类名

C++中定义的结构和类实际上就是一个用户定义的数据类型,声明这些类型的变量

时,在结构和类名前不必像 C 那样加 struct 和 class 标识符,而只须直接使用类型名。

3. 在程序块中声明变量

把几个语句用大括号"{"和"|"组合起来就构成了一个程序块。在 C 语句中变量的声明只允许出现在程序块的开始,而在 C++语句中,声明也是语句,因此允许出现在块内其他语句之后,这样就允许在使用该变量时再对其声明。

4. 作用域标识符

C++中增加了作用域标识符::,用来解决局部变量名和全局变量名的同名重复问题。在 C++中可用作用域标识符::对被同名各局部变量而隐藏的全局变量进行访问,而在 C 中是不允许的。作用域标识符::还可用来标识类成员。

5. 常量

C++中增加了常量类型,常量类型用标识符 const 声明,它的值在作用域内保持不变。C++中的常量与用预处理语句 # define 定义的符号有所不同:常量具有作用域属性,而且通过名字来访问,而预定义符号则由编译器在编译源程序时进行符号替换。

6. 无名联合

在 C 语句和 C++语句中都可以定义联合,所不同的是,C++中联合可以没有名字,即所谓的无名联合。无名联合可以在定义变量和成员的地方定义,但主要用于结构和类的定义之中。

7. 显式类型转换

C++中基本类型名和用户定义类型名(不包括它们的派生类型)可以作为映射函数,以函数调用方式来实现数据类型的显式转换。同时,C++还保留了 C 中的强制类型转换(cast convertion),两者作用是相同的。

8. 函数原型(function prototype)

C++在声明和定义函数时,函数及其返回值均应在它们的参数表中明确指出参数的类型,也就是说 C++采用函数原型作为其函数头。

9. 函数名重载

C++中允许相同的函数名出现在同一作用域内,这就是所谓的函数重载。当调用重载函数时,编译器根据其所带参数的个数和类型来加以选择,换句话说,C++虽然允许同一作用域内具有同名函数,但这些同名函数,要么参数个数不同,要么参数类型不同。

10. 缺省参数

C++中函数参数表的后面部分允许设置缺省值。当函数调用时给出的参数少于参数表中参数的总数时,所缺参数将自动取声明函数时参数表中设置的缺省值。

11. 带不定数目参数的函数

C++可以用省略号定义带有不定数目参数的函数,编译器对这类函数不做参数检查。

采用带不定数目参数的函数的目的是关闭对函数参数的类型检查，以增加函数界面的灵活性。这主要用于通用性很强且无法预料参数数目和类型的函数，如输入和输出函数等。

12. 引用(reference)

C + + 中用标识符 & 可以派生一个引用类型。如 int& 定义了一个对 int 类型的引用，也就是对 int 类型的变量赋予另一个名字。因此对引用进行操作，实际上就是对被引用的变量进行操作。

13. 内联(inline)函数

内联函数用标识符 inline 来定义。当程序编译时，对内联函数在其调用处进行内联替换，也就是将内联函数的代码在调用处插入，从而消除执行调用的开销。内联主要用于语句数目较少的小函数中。

14. 新增的运算符 new 和 delete

C + + 提供运算符 new 和 delete 使编程者可以很方便地直接控制堆的分配和释放。

15. 无类型 void

C + + 中无类型 void 的用途非常广泛。void 用于指明没有返回值的函数，也可以声明无类型指针，在程序中无类型指针可设置为指向任何类型的指针。

### 3.1.4　C + + 的面向对象特征

1. 类和数据封装性

C + + 采用类的结构使它具有面向对象编程语言的重要特征。类体现了数据抽象和数据隐藏的特性。类封装了一组数据结构和一组可作用于该数据结构的方法，它是对一组对象共性的抽象，该组对象具有相同的数据结构和方法，但每个对象都具有自己特定的数据值。类与类之间是通过发送和接收消息相联系的，接收消息的对象通过调用类的方法来实现相应的操作。

2. 访问限制符和数据隐藏性

访问限制符 private、protected、public 将类分成三部分：私有部分(private)、保护部分(protected)、公有部分(public)，各部分的访问权限是不相同的。

在私有部分声明的数据和函数只能由本类内声明的函数访问，非本类函数无权直接访问它们；保护部分声明的数据和函数则可由本类及其派生类的函数访问；公有部分声明的数据和函数则可以由任何函数访问。因此类的私有部分和保护部分对外是隐藏的，只是程度不同，而公有部分则对外是可见。通常，将类的方法定义为公有部分，因为只有通过这些方法才能操纵对象。

3. 构造函数和析构函数

类定义可包含一组构造函数(constructor)和一个析构函数(destructor)。构造函数保证了在声明类的对象时对其自动初始化，而析构函数则保证了对类的对象正常清除(clean up)。

4. 结构是类的特例

C++中结构是类的一个特例,它只有公有部分,而没有私有部分和保护部分。因此C++结构中可包含数据和函数,这和C语言不同,C中结构不包含成员函数。

5. 对象和消息

对象是面向对象编程的基本单元,可通过向对象发送消息来实现对对象的操纵,对象根据消息的内容调用相应的方法。通常,类的公有部分描述了可以发送给对象的全部消息,并给出了相应的方法。

C++中类实际上就是用户定义的类型,对象就是用户定义的类的实例,而方法则是由函数定义的,函数原型则描述了相应的消息模式。消息传递采用了类似函数调用的机制。例如,若object是一个对象,meth是类所定义的方法,它带有一个整数参数,则object.meth(3)表示将meth(3)消息传送给对象object。

6. 友员

友员是C++面向对象的另一个重要特征。通常,类的私有部分是禁止其他函数直接访问的,而友员机制则允许有选择地突破这种限制。只要在类定义中声明非成员函数或其他类为该类的友员,则这些友员就可以直接访问类的私有部分和保护部分。

7. 运算符重载

利用operator标识符可以给已有的运算符在类定义中赋予新的含义,使之可以作用于该类的对象,这就是运算符重载。运算符重载使得程序员能以更自然的表现方式实现对对象的操作。

8. 派生类

从已有的类可以派生新的类,前者称为基类,后者称为派生类。派生类除继承了基类的所有成员外,和其他类并没有什么区别,它也可以定义新的数据和方法。

9. 虚函数

C++中虚函数用来实现多态性(polymorphism)。多态性是指在由基类和派生类组成的层次结构中,不同派生类可以接受同名消息,但对应的方法却不相同。

10. 流(stream)

任何C++语言系统都提供流支持。类cin、cout、cerr用来实现对终端和文件的输入输出。cin、cout、cerr中的运算符还可以在用户定义类型中重载,从而可以很方便地实现用户定义类型的输入输出。

## 3.2 C++标识符与关键字

### 3.2.1 标识符

标识符是用来表示应用程序中的变量、常量、类型、函数和标号的名字。在定义变量、类型或函数时通过声明一个标识符来创建它,然后就可以在后面的程序中使用这个标识符来指向有关的内容。

C++的标识符必须是以下划线或字母开头的,是由字母、数字和下划线组成的序列。标识符可能包含任意多个数字、字母和下划线,但是只有前31个字符对编译器来说是有效的。C++对大小写是敏感的,这意味着C++编译器认为大写字母和小写字母是不同的标识符。

尽管使用下划线开头的标识符是合法的,但在自定义的标识符中最好不要这么用。因为系统例程的名字多以下划线开头,所以这样做容易引起冲突,从而产生错误。另外,包含有下划线开头的名字常常不能保证很好的移植性。

下面是一些合法标识符的例子:

```
itotal    frangel   szfirst_name lfrequency imax, iMAX NULL EOF
```

下面则不是合法的标识符:

1st_year 数字不能作为标识符的开头。

#social_security "#"不能作为标识符的开头。

NOT_MOVE! 非字母、数字、下划线字符不能包含在标识符中。

### 3.2.2 关键字

关键字是预定义的,具有特殊含义的标识符。只能按它们预定义的意义来使用。表3-1列出了C++中的关键字。在此表中前面标有"+"的关键字是C++所特有的,下划线开头的是Microsoft所特有的,使用时应包括下划线。其他的是C、C++和Microsoft三者共有的。至于每一个关键字的用法及含义将在后面相关章节中介绍,在此列出供编程时参考。

表3-1 C++关键字

| | | | |
|---|---|---|---|
| _asm | _fastcall | public | signed |
| except | (+)new | unigned | void |
| _virtual_inheritance | typedef | (+)using directive | do |
| throw | (+)class | default | _leave |
| case | goto | _int8 | struct |

续表

| | | | |
|---|---|---|---|
| _ finally | register | sizeof | auto |
| ( + )operator | using decarlation | volatile | explicit |
| typeid | uuid | double | mutable |
| const | ( + )delete | long | ( + )true |
| if | _ int16 | switch | catch |
| reinterpret _ cast | static | template | float |
| return | winmain | _ based | private |
| uuidof | dynamic _ cast | extern | typename |
| dllexport | main | naked | const _ cast |
| _ int32 | _ multiple | try | ( + )inline |
| staic _ cast | ( + )this | _ cdecl | _ inline |
| while | bool | for | short |
| else | false | ( + )protect | virtual |
| enum | namespcae | union | dllimport |
| _ single _ inheritance | _ try | continue | _ int64 |
| thread | char | _ declspec | _ stdcall |
| break | ( + )friend | int | xalloc |

## 3.3 C + + 数据类型

C + + 的基本数据类型分三类：整型、浮点型和指针型。char，short，int，intn 和 long 是整型；浮点类型用于处理包含小数部分的数值，C + + 支持三种浮点类型：float，double 和 long double；void 类型用于描述无值空集，主要用于声明不返回值的函数或指向任一类型的指针。

### 3.3.1 类型 char

类型 char 是包含 ASCII 字符(英文字母、数字、标点符号及某些特殊符号)的整数类型。它主要用来表示 ASCII 字符或较小的整数。在 C + + 中 char 类型的数可以作为整数来使用，适用于整数的运算同时也适用于 char 类型。也许在将来的 C + + 中 char 类型将用来表示单代码字符集(unicode)中的字符。

表 3-2　类型 char

| 类型名称 | 长度(字节) | 取值范围 |
|---|---|---|
| char | 1 | -128 ~ 127 或 0 ~ 255 |
| signed char | 1 | -128 ~ 127 |
| unsigned char | 1 | 0 ~ 255 |

**注释：**

缺省时，类型 char 即为 signed char。

程序清单 3-3 说明了如何在程序中使用 char 类型的数据。

**程序清单 3-3：在程序中使用 char 类型的数据**

```
//该程序演示如何使用 char 类型
//文件名为:MyCharDemo.C
#include <iostream.h>
int mian()
{
char x;
cout << "请输入一个字符" << endl;
cin >> x;
cout << "您输入的字符是" << x << endl;
return 0;
}
```

### 3.3.2　类型 short

类型 short 是 short int 的缩写，长度大于或等于类型 char，小于或等于类型 int，具体长度依赖于操作系统和编译系统。表 3-3 是 C++ 中 short 类型的约定。程序清单 3-4 说明了如何在程序中使用 short 类型的数据。

表 3-3　类型 short

| 类型名称 | 长度(字节) | 取值范围 |
|---|---|---|
| short(signed short) | 2 | -32768 ~ 32767 |
| unsigned short | 2 | 0 ~ 65535 |

**程序清单 3-4：在程序中使用 short 类型的数据**

```
//该程序演示如何使用 short 类型
//文件名为:MyShortDemo.C
```

```
#include <iostream.h>
int mian()
{
short x,y;
cout << "请输入第一个短整数" << endl;
cin >> x;
cout << "请输入第二个短整数" << endl;
cout << "您输入的两数之和是" << x+y << endl;
return 0;
}
```

### 3.3.3 类型 int

类型的 int 长度大于或等于类型 short,小于或等于类型 long。与 short 类型相同,它也依赖于操作系统和编译系统。表 3-4 是 C++ 中 int 类型的约定。程序清单 3-5 说明了如何在程序中使用 int 类型的数据。

表 3-4 类型 int

| 类型名称 | 长度(字节) | 取值范围 |
|---|---|---|
| int(signed int) | 4 | -2147483648 ~ 2147483647 |
| unsigned int | 4 | 0 ~ 4294967295 |

**程序清单 3-5:在程序中使用 int 类型的数据**

```
//该程序演示如何使用 int 类型
//文件名为:MyIntDemo.C
#include <iostream.h>
int mian()
{
int x,y,z;
cout << "请输入第一个整数" << endl;
cin >> x;
cout << "请输入第二个整数" << endl;
z = x/y;
cout << "您输入的两数之积是" << x * y << endl;
cout << "您输入的两数相除,余数是" << x-z * y << endl;
return 0;
}
```

### 3.3.4 类型 long

类型 long 的长度大于或等于 int,它一般用来表示较大范围的整数。从表 3-5 可以看出

long 类型与 int 类型的范围完全相同。程序清单 3-6 说明了如何在程序中使用 long 类型的数据。

**程序清单 3-6:在程序中使用 long 类型数据**

```
//该程序演示如何使用 long 类型
//文件名为:MylongDemo.C
#include <iostream.h>
int mian()
{
const long pi = 3.1415926;
int d;
long z;
ccut << "请输入一个角度" << endl;
cin >> x;
z = sin(d/2/pi);
cout << "您输入角度的正弦是" << z << endl;
      return 0;
}
```

**表 3-5 类型 long**

| 类型名称 | 长度(字节) | 取值范围 |
|---|---|---|
| long(signed long) | 4 | -2147483648 ~ 2147483647 |
| unsigned long | 4 | 0 ~ 4294967295 |

### 3.3.5 类型_intn

类型_intn 用于表示整数变量的大小。其中,n 表示位数,可以为 8,16,32,64。类型_intn通常用于编写可移值的代码,以便在多个不同平台上使用,在 C++中,数据类型 int8 与 char 等价,int16 与 short 等价,int32 与 long 等价。

### 3.3.6 浮点类型

浮点类型用于表示带小数的数。C++提供了三种浮点类型,如表 3-6 所示。

**表 3-6 C++中的浮点类型**

| 类型名称 | 长度(字节) | 取值范围 |
|---|---|---|
| float | 4 | 3.4E-38 ~ 3.4E+38 (正数) |
| double | 8 | 1.7E-308 ~ 1.7E+308 |
| long double | 8 | 1.2E-4932 ~ 1.2E+4932 |

下面的程序清单 3-7 说明了如何在程序中定义和使用浮点类型的变量。

**程序清单 3-7:在程序中使用浮点型变量**

```
//该程序演示如何使用浮点类型
//文件名为:MyfloatDemo.C
#include <iostream.h>
#include <iomanip.h>
int mian()
{
long loriginal_flags = cin.flags();
float fvalue;

cout << "请输入一个要格式化的浮点值" << endl;
cin >> fvalue;

cout << "标准格式:" << fvalue << endl;
cout.setf(ios::scientific);
cout << "科学记数法格式:" << fvalue << endl;

cout.setf(ios::fixed);
cout << "固定小数点格式:" << setprecision << fvalue << endl;
     cout.flags(loriginal_flags);
return 0;
}
```

### 3.3.7 常量

在 C++ 中,常量是指不可由程序修改的固定值。常量可以是任何数据类型,其中,字符型常量用单引号括起来,如'a',以 0 开头的常量为八进制,以 0X 开头的常量为十六进制。常量既可以直接书写在表达式中,也可以定义成一个符号常量。定义符号常量的方法有以下两种:

1. const 修饰符

C++ 中的 const 修饰符用于声明常量标识符,常量标识符的值在程序运行期间是不可更改的。

常量声明的语法如下:

```
const data_type_name user_ID = value;
```

其中 const 是声明常量的修饰符,data_type_name 是要声明的常量的数据类型,user_ID 为常量标识符,value 为常量的值。例如:

```
const iMin = 1,iSalePercentage = 25;
```

```
const float fbase_change = 34.34;
```

2. # define 编译器指令

利用 # define 编译器指令也可以定义一个常量，例如，在一个程序的开头有这样一条指令：

```
# define SALES_TEAM 10
```

编译器在编译程序时遇到 # define 指令时，将用 10 代替源程序文件中的所有 SALES_TEAM。

以上两种定义常量的方法在大多数情况下是通用的。但修饰符 const 使得“变量”的值不可变。在 C++中，变量具有存储特性，即变量只在程序的一定区域存在，用 const 修饰符所定义的常量也具有此特性，而 # define 指令所定义的常量则始终具有全局作用域。

## 3.4 数据类型转换

迄今为止，程序中用于某一特定指令的变量和常量都具有一样的类型。程序员可以编写出在不同类型的变量之间进行操作的指令。这些操作被称为混合模式操作。

### 3.4.1 自动数据类型转换

与一些其他的程序设计语言相比，C++可以实现从一种数据类型到另一种数据类型的自动转换。C++的自动类型转换总是把较低优先级的数据类型转换为较高优先级的数据类型。各种数据类型的优先级顺序是：double、float、long、int 和 short。

例如，当变量 x 和 y 都被声明为整型(int)，而 f1 和 f2 声明为浮点型(float)，有如下指令：

```
x = 3;
y = 4;
f1 = 7.0;
f2 = f1 + x/y;
```

由于 x/y 没有混合模式操作，因此不能执行自动数据类型转换，其结果为 0，于是分数部分被舍弃，所以 f2 的最终结果是 7.0。如果 x 或 y 被声明为 float，x/y 是混合模式操作，运算时将进行自动类型转换，因此其结果为 0.75，这样 f2 的最终结果是 7.75。

### 3.4.2 强制类型转换

如果 C++不进行自动类型转换，类型转换将被忽略。如上例的第一种情况，在这种情况下，程序设计者就必须进行明确的类型转换，以保证运算结果的正确性。这就是所谓的强制类型转换。C++提供的强制类型转换有两种方法：

(1) 在要转换的变量前加上括号，并在括号中指明欲转换的类型。此种方法被称为 cast 方法。例如，如果 ivalue1 和 ivalue2 被声明为整型变量，而 fvalue1 和 fvalue2 被声明为浮点类

型，则下面的三条指令执行相同的操作，最后的结果都是浮点类型：

```
fresult = fvalue1 + (float)ivalue1/ivalue2
fresult = fvalue1 + ivalue1/(float)ivalue2
fresult = fvalue1 + (float)ivalue1/(float)ivalue2
```

(2) 像调用函数那样，把目标数据类型名作为函数名，将欲转换的变量作为参数放到括号中。例如 c = cw/int(b)。

## 3.5 C++存储类与变量的作用域和可见性

存储类处理变量的内存分配方式和变量的生存期。C++支持四种存储类：auto，register，extern 和 static。

存储类用在变量定义之前，它们用于指示编译器如何存储一个变量。用 auto 和 register 声明的变量具有局部的生命周期；用 static 和 extern 声明的变量具有全局的生命周期。

这四个存储类声明可以影响变量或函数的可见性。可见性又称作用域，是指在源程序中可以用名字来引用变量或函数的区域。

### 3.5.1 auto 存储类

auto 存储类声明的变量都是局限于某个范围内的，即只能在某个范围使用。编译程序在堆栈中分配 auto 存储类的变量。在 C++中，关键字 auto 可以省略。

### 3.5.2 register 存储类

register 存储类的变量基本上与 auto 存储类的变量具有相同的含义，但是，编译器尽可能利用 CPU 的寄存器分配此类变量，以加快程序的运行速度。register 变量的声明方式如下：

```
register int i; //声明一个整数类型的 register 变量 i
```

### 3.5.3 extern 存储类

使用 extern 声明的变量为外部变量，它一般定义在程序外部。

当某个变量被定义为外部变量时，所有其他函数或程序段都可引用此变量。这种变量的作用域是所有的函数和程序段，一般用于在函数之间传递数据。

程序清单 3-8 演示了 extern 存储类的用法。

**程序清单 3-8：extern 存储类用法示例**

```
//这是 extern 存储类用法示例的第一个程序，
//此文件中的主函数 main()调用了 teststringfun()函数，但这个函数的定义部分
//不在此文件中，因而用 extern 关键字声明函数 teststringfun()为外部函数
//文件名：MyCallExtern.C
```

```
int a = 25896;
char * tinystr = "Hi!";
extern void teststringfun();
int main( ){
    teststringfun();
    return 0;
}
//这是 extern 存储类用法示例所需的第二个程序,
//在此文件中定义了 teststringfun()函数,供前面的文件调用
//文件名:MyExternDeclar.C

# include < iostream.h >
extern a;
extern tinystr;
void teststringfun(){
     cout < < a < < " \ ";
     cout < < tinystr < < " \ ";
}
```

### 3.5.4 static 存储类

静态变量也是一种局部变量,它和 auto 变量的最大区别在于这种变量在内存中是以固定地址存放的,而不是以堆栈方式存放的。因此,只要程序继续执行,静态变量就不会随着声明它的程序段的结束而消失。

程序清单 3-9 说明了如何使用 static 存储类。

**程序清单 3-9:static 存储类用法示例**

```
//这是个 static 存储类用法的示例程序
//文件名:MyStaticDemo.C

# include < iostream.h >
void func(){
     int i = 0;
     static int j = 0;
     cout < < "i = " < < i < < endl;
     cout < < "j = " < < j < < endl;
     i + + ;
     j + + ;
}
void main(){
     fun();
```

```
    fun();
}
```

在上面的程序中,i 为 auto 存储类变量,j 为 static 存储类变量,每次调用 fun()函数,i 的值始终是 0,j 的值则每调用一次增加 1。例如,第一次调用时 i = 0,j = 0,第二次调用时 i = 0,j = 1。

### 3.5.5 内部变量和外部变量

在所有函数定义之外声明的变量叫做外部变量,而在函数定义内声明的变量叫做内部变量。外部变量只能用 static 和 extern 存储类,而不能用 auto 和 register。外部变量在定义时可以对其初始化(隐式的或显式的)。例如:

```
static int i1;//隐式
static int i2 = 10;//显式
```

外部变量在整个源文件中在其定义出现之后的部分都是可见的。在同一个源文件中,变量在被声明之前是不可见的。同样,在程序的其他源文件中,除非有引用声明,否则它也是不可见的。

内部变量可以用四个存储类来修饰,不同的修饰方式具有不同的作用域。用 auto 和 register 存储类修饰的变量只在定义它的程序块内是有效的和可初始化的。用 static 修饰符声明的变量具有全局的生存周期,且退出其所在程序块后变量值不消失。用 extern 修饰符声明的变量实际上是对一个同名变量的引用,这个同名变量是在程序的任何一个源文件中定义的外部变量。另外,内部级的 extern 声明可用来定义一个只在这个声明所在块可见的外部级变量。

清单 3-10 程序很好地说明了上述概念:

**程序清单 3-10:变量的作用域可见性示范程序**

```
int ivalue1 = 1;//此语句在某一文件中,在此例中尚不能被编译

void main()
{
//引用上面所定义的整型变量 ivalue1
    extern int ivalue1;
//在 main()中只有 ivalue2 是可见的
    static int ivalue2;

//如果可能,变量 rvalue 存储在寄存器中,初始值为 0
register int rvalue = 0;

//int _value3 为 auto 存储类,初始值化为 0
    int int _value3 = 0;
```

```
//输出值是 1,0,0,0
    cout << ivalue1 << rvalue << ivalue2 << int_value3;
    function_a();
}

void function_a()
{
//存储全局变量 ivalue1 的地址
    static int * pivalue = &ivalue1;

//建立一个新的与全局变量 ivalue1 同名的局部变量,从而使全局的 ivalue1 不可见
    int ivalue1 = 32;
//新的局部变量 ivalue2 只有在函数 function_a 中可见

    static int ivalue2 = 2;
    ivalue2 += 2;

//输出值是 32,4 和 1
    cout << ivalue1 << rvalue << ivalue2 << *pivalue1;
}
```

### 3.5.6 变量作用域总结

变量有四个作用域:块(block)、函数(function)、文件(file)以及程序(program)。在块或函数中定义的变量只在这个块或函数中可见。在函数外部定义的变量在文件中从它的定义点到文件结尾是可见的。如果一个变量在源文件中被定义为外部变量,并且在其他的文件中也被声明为外部变量,则该变量具有程序作用域。

## 3.6 操作符

C++中有许多其他语言没有的操作符。这些操作符包括算术运算操作符、逻辑运算操作符、关系运算操作符、位操作符、增量操作符、减量操作符、条件操作符、分配操作符和复合赋值操作符等。

### 3.6.1 算术运算操作符

C++算术运算符中的"+"、"-"、"*"及"/"的工作方式与在大多数其他计算机编程语言中相同,这些运算符可用于C++所允许的任何内部数据类型。除号"/"用于整数时舍弃余数,我们可以用括号来增加优先级。表3-7列出了C++中的算术运算操作符。

表 3-7 算术运算操作符

| | 操作符名 | 操作符 | 功能 | 示例 | 运算优先级 |
|---|---|---|---|---|---|
| 双目运算 | 加 | + | 将两个操作数相加 | c = a + b 若 a = 1, b = 2, 则 c = 3 | 低 |
| | 减 | - | 用第一个操作数减去第二个操作 | c = a - b 若 a = 5, b = 2, 则 c = 3 | 低 |
| | 乘 | * | 两个操作数相乘 | c = a * b 若 a = 1, b = 2, 则 c = 2 | 中 |
| | 除 | / | 用第一个操作数的值除以第二个操作数的值 | c = a/b 若 a = 8, b = 2, 则 c = 4 | 中 |
| | 求余 | % | 求第一个操作数除以第二个操作数的余数 | c = a%b 若 a = 5, b = 2, 则 c = 1 | 中 |
| 单目运算 | 负数 | - | 将操作数取反 | c = - a 若 a = 1, 则 c = - 1 | 高 |
| | 增量 | + + | | 若 a = 3, 则 x = a + + 执行结果为 a = 3, x = 4, 而 x = + + a 执行结果为 a = 4, x = 4 | |
| | 减量 | - - | | 若 a = 3, 则 x = a - - 执行结果为 a = 3, x = 2, 而 x = - - a 执行结果为 a = 2, x = 2 | |

## 3.6.2 关系运算操作符

关系运算符用于测定一个量与另一个量的关系，根据测试结果返回“0”或“1”，即执行结果是“真”和“假”，在 C++ 中“真”是除 0 之外的任何值，而“假”就是 0。表 3-8 列出了 C++ 中的关系运算操作符。

表 3-8 关系运算操作符

| 操作符 | 含义 |
|---|---|
| ! = | 不等于 |
| < | 小于 |
| = = | 等于 |
| > = | 大于等于 |
| > | 大于 |
| < = | 小于等于 |

### 3.6.3　逻辑运算操作符

在 C + + 中所有的 0 均被看作“假”即 false，非零被看作“真”即 true。逻辑运算符用于根据真值表进行基本逻辑运算“与”、“或”和“非”。逻辑运算符可以与关系运算符混合使用，用来表示比较复杂的逻辑表达式。表 3-9 列出了 C + + 中的逻辑运算操作符。

表 3-9　逻辑运算操作符

| 操作符 | 含义 | 例子 |
| --- | --- | --- |
| ! | 非(NOT) | x = 3 y = ! x 则 y = 0(即 false) |
| && | 与(AND) | x = 3，y = 2，z = x&&y，则 z = 1 |
| \|\| | 或(OR) | |

### 3.6.4　位操作符

位操作符将变量看作是位的组合，而不是数字。利用位操作符可以实现用汇编语言才能完成的功能。表 3-10 列出了 C + + 中的位操作符。

表 3-10　位操作符

| 操作符 | 含义 | 例子(这里均为十六进制) |
| --- | --- | --- |
| & | 与 | 0xF1 & 0x35 结果是 0x31 |
| \| | 或 | 0xF1 \| 0x35 结果是 0xF5 |
| ^ | 异或 | 0xF1 ^ 0x35 结果是 0xC4 |
| ~ | 非 | ~0xF1 结果是 0xFF0E |
| < < | 左移 | 若整型 x = 65，则执行 x < < 1 后 x = 130，相当于乘以 2 |
| > > | 右移 | 若整型 x = 64，则执行 x > > 1 后 x = 32，相当于除以 2 |

### 3.6.5　增量和减量操作符

增量和减量操作符用简单的指令实现了变量的加 1 和减 1，这两个操作符广泛用于数组的下标和循环控制变量等。增量和减量操作符分为前缀和后缀两种。表 3-11 列出了 C + + 中的增量和减量操作符。

表 3-11 增量和减量操作符

| 操作符 | 含义 | 例子 |
| --- | --- | --- |
| ++ | 增量 | 若 a=3,则 x=a++ 执行后,a=3,x=4,而 x=++a 执行后,a=4,x=4 |
| -- | 减量 | 若 a=3,则 x=a-- 执行后,a=3,x=2,而 x=--a 执行后,a=2,x=2 |

### 3.6.6 赋值和复合赋值操作符

在 C++ 中的赋值运算符很有特色,一是赋值运算的右边可以是任何类型的表达式,二是具有复合赋值运算符,表 3-12 中列出了 C++ 中所有的赋值和复合赋值操作符。

表 3-12 赋值和复合赋值操作符

| 操作符 | 含义 | 例子 |
| --- | --- | --- |
| = | 赋值 | 若 x=1,y=x,则 y=1 |
| += | 加赋值 | 若 x=1,y=1,y+=x,则 y=2 |
| -= | 减赋值 | 若 x=1,y=1,y-=x,则 y=0 |
| *= | 乘赋值 | 若 x=2,y=3,y*=x,则 y=6 |
| /= | 除赋值 | 若 x=6,y=2,y/=x,则 y=3 |
| %= | 求余赋值 | 若 x=5,y=11,y%=x,则 y=1 |
| <<= | 左移赋值 | 若 x=1,y<<=x,则 y=2 |
| >>= | 右移赋值 | 若 x=1,y>>=x,则 y=0 |
| &= | 与赋值 | 若 x=1,y&=x,y=1 则 y=1 |
| ^= | 异或赋值 | 若 x=1,y=0,y^=x,则 y=1 |
| \|= | 或赋值 | 若 x=1,y=1,y\|=x,则 y=1 |

### 3.6.7 条件操作符

在正式的程序设计中可以使用条件操作符(?:),但是它更多地用于创建宏(macro)。此操作符的语法格式如下:

```
condition? true_expression:false_expression
```

如果条件为 TRUE,条件表达式的值为 true_expression 的值,否则,条件表达式的值为

false _ expression 的值。例如下面的语句：

```
if('A' < = c && c < = 'Z')
printf("%c",'a' + c - 'A');
else
printf("c%",c);
```

可以用条件操作符表示如下：

```
printf("c%", ('A' < = c && c < = 'Z')? ('a' + c - 'A'):c)
```

这两种语句都将变量 c 转换为小写字母输出。

### 3.6.8 逗号操作符

逗号操作符(,)用来在只允许求取一个表达式的地方求取两个表达式的值。逗号表达式的值最终是右边表达式的值。该操作符的语法格式如下：

```
lef _ expression,right _ expression
```

逗号操作符常常用在 for 循环中，该循环中需要进行不止一个变量的重复操作。例如：

```
for (min = 0,max = length - 1;min < max;min + + ,max - - ){
⋮
}
```

在上面的例子中有两个逗号操作符，它们分别同时执行两个赋值运算和两个算术运算。

### 3.6.9 操作符的优先级和结合性

在 C++中一个表达式可以综合利用多种操作符，此时必须确定对表达式的求值顺序，这就涉及到操作符的优先级和结合性。优先级高的操作符先计算，优先级低的操作符后计算，相同优先级的操作符则根据其结合性来计算，或从左向右计算(称之为左结合)，或从右向左计算(称之为右结合)。表 3-13 列出了 C++中所有操作符的优先级和结合性。在此表中所有一元运算符和赋值运算符具有右结合性，其他运算符则具有左结合性。在同一格中的运算操作符具有相同的优先级，上一行具有比下一行较高的优先级。

表 3-13 操作符的优先级和结合性

| 符号 | 名字或含义 | 结合性 |
|---|---|---|
| :: | 作用域 | 右 |
| () | 函数调用 | 左 |
| [] | 数组元素 | |
| -> | 结构成员指针 | |

续表

| 符号 | 名字或含义 | 结合性 |
| --- | --- | --- |
| . | 结构或联合的成员 | |
| sizeof | 以字节为单位的大小 | 右 |
| ++ | 前(后)自加 | 左 一元运算符 |
| -- | 前(后)自减 | |
| ! | 逻辑非 | |
| ~ | 按位取反 | |
| - | 一元减 | |
| + | 一元加 | |
| & | 取地址 | |
| * | 引用 | |
| new | 为程序分配内存 | 右 一元运算符 |
| delete | 释放程序内存 | |
| (type) | 类型强制转换 | |
| * | 乘 | 左 算术运算符 |
| / | 除 | |
| % | 取余 | |
| + | 加 | 左 算术运算符 |
| - | 减 | |
| << | 左移 | 左 逻辑运算符 |
| >> | 右移 | |
| < | 小于 | 左 关系运算符 |
| <= | 小于等于 | |
| > | 大于 | |
| >= | 大于等于 | |
| == | 相等 | 左 关系运算符 |

续表

| 符号 | 名字或含义 | 结合性 |
| --- | --- | --- |
| ! = | 不等 | |
| & | 按位与 | 左 逻辑运算符 |
| ^ | 按位异或 | 左 逻辑运算符 |
| \| | 按位或 | 左 逻辑运算符 |
| && | 逻辑与 | 左 逻辑运算符 |
| \|\| | 逻辑或 | 左 逻辑运算符 |
| ?: | 条件 | 左 条件运算符 |
| = | 赋值 | 右 赋值运算符 |
| * =,/ =,% =,+ =,- =,<< =,>> =,&=,^=,\| = | 复合赋值 | 右 |
| , | 逗号 | 左 逗号运算符 |

### 3.6.10 表达式

运算符、常量、变量和具有返回值的函数调用是组成表达式的元素。在C++中,表达式是这些元素的任意合法组合。如果表达式中的数据类型不一致,则需要进行数据类型转换,转换原则见3.4节。C++的表达式都有返回值,对于大多数表达式,其运算结果就是表达式的返回值,对于逗号表达式,其返回值是逗号右边的表达式的返回值。C++表达式的另一个特色是任何合法的表达式都可以用来表示应用于各种控制语句中的条件,而不像某些其他编程语言,条件只能用关系和逻辑运算符表达。对这类表达式的要求仅仅是:表达式的值为0,则认为条件取值为false,任何非0值都认为条件取值为true。

## 3.7 高级数据类型

### 3.7.1 枚举类型

C++采用enum关键字来声明枚举类型。枚举类型是一个连续的整型变量的列表,其中的每个常量都具有一个用户赋值的变量名。枚举类型实质上是命名的整型集合,此集合被称为枚举集。如果没有显式指明某个常量名对应的整数,则编译器自动按照这些名字的出现次序初始化0,1,2,...,依次类推。例如enum digital {zero,one,two,three}声明之后,zero代表0,one代表1,two代表2,three代表3。如果对某个常量明确指定取值,则其后继的每个

常量名在此基础上依次增加。如 enum romandigit {i = 1, ii, iii, iv, x = 10, xi} 声明之后，i 代表 1，ii 代表 2，iii 代表 3，iv 代表 4，x 代表 10，xi 代表 11。

当一个 enum 型变量的某一域名出现在程序中时，编译器用该域的整型数值取代此域名。在 C 语言中如果想要定义枚举变量的一次使用时，必须在此变量名前加上 enum 关键字，而在 C++ 中则不必如此。例如：

```
//C++ 枚举类型的应用例子
//文件名:MyEnume.C
#include <iostream.h>
enum Customer {Domestic, Canadian, Japanese};
int main(){
    Customer n = Domestic;//C 中应为 enum Coustomer n = Domestic;
    cout << "The Cuftoners nationality is Nationality No" << n << "\n";
    return 0;
}
```

以上例子说明了如何使用枚举类型。C 与 C++ 的主要区别在于 C++ 枚举标识的是一个类型名。这个特点允许用 enum 标识去说明一个枚举变量类型。程序清单 3-11 说明了如何在 C++ 程序中使用枚举类型：

**程序清单 3-11：在程序中使用枚举类型**

```
//在 C++ 程序中使用枚举类型
//文件名 MyEnumeDemo.C
#include <iostream.h>
enum playerclasses {warrior, gnome, wizard};
char * classname[3] = {"warrior","gnome","wizard"};
char *printclassstring(playerclasses i){
      return classname[i];
}
int main(){
    char * astring = printclassstring[wizard];
    cout << astring << endl;
return; //在 C++ 中,main()函数返回整数值
}
```

### 3.7.2 联合(union)

C++ 中联合(union)可以使不同的数据类型共享相同的内存位置。当定义一个 union 时，分配给它的存储空间要足以容纳 union 包含的最长的一个数据元素。union 的语法如下：

```
union tag_field{
    type field1;
```

```
    type field2;
    type field3;
    :
    type fieldn;
};
```

下面是一个 union 声明语句：

```
union EPluribus {
      short ShortVal;
      long LongVal;
      double doubleVal;
};
```

在声明联合(union)的同时也可以声明一个联合型变量，例如：

```
union EPluribus {
      short ShortVal;
      long LongVal;
      double doubleVal;
} value;
```

C++提供了一类特殊的联合，称作 anonymous union(无名联合)。无名联合和一般联合的最明显的区别在于前者没有标记名(tag name)。下面是一个无名联合的声明语句：

```
union {      //无名联合
      int anInt[2];
      div_t div, result;
}
```

无名联合具有以下限制条件：

- 无名联合如果在文件中是全局的，则必须声明为静态的(static)；
- 在面向对象的程序中，一个联合可以只有公用成员，在一个无名联合中声明私用或保护成员会引起编译错误。
- 无名联合不能有成员函数。

程序清单 3-12 是一个无名联合的例子。

**程序清单 3-12：在程序中使用无名联合**

```
//此例子示范了无名联合的应用
//文件名:MyAnonyunionDemo.C
#include <iostream.h>
#include <stdlib.h>
int main(){
    union{//angonymous union
```

```
        int anint[2];
        div_t result;
    };
cout << "enter an integer:" << "\n";
cin >> anint[0]; //number to be divided
cout << "divide by:" << "\n";
cin >> anint[1]; //divisor;
result = div(anint[0],anint[1]);
cout << "quotient:" << result.quot << "\n";
cout << "remainder:" << result.rem << "\n";
return 0;
}
```

程序中使用 math.h 函数 div()做两个数的除法运算，并打印出除法运算的结果。参与除法运算的被除数和除数分别保存在 anint[0]和 anint[1]中。函数 div()把除法运算的结果保存在 div_t.quot 和 div_t.rem 中，它们在内存中具有与 anint[0]和 anint[1]相同的地址。所以，除法运算完成以后，其结果与所做运算前的数值放在同一地址处。

### 3.7.3 结构声明

C 语言中的结构(用 struct 声明)实际上就是用户扩充的数据类型，是一个用于存放各种类型的数据聚合体的属性单元，而 C++中的结构是一个实实在在的数据类型。C++中提供的结构数据类型是一种复合数据类型。

C++中结构的定义的形式如下：

```
struct tag_field{
      member_type member1;
      member_type member2;
      member_type member3;
      .
      .
      .
      member_type membern;
};
```

结构的定义以分号结束，因为这是一个真正的 C++语句。下面的例子在定义结构的同时也声明了一个此结构类型的变量。

```
struct my_struct {
  char name[80], phone_number[80];
  int  age, height;
} my_friend;
strcpy(my_friend.name,"Mr. Wizard");
```

```
struct my_struct my_friends[100];
```

C + + 中，struct 是一个实际的数据类型，可以不带 struct 关键字来引用它们，例如：

```
struct data = {7.4,18884};
data datas[16];//在 C 中应是 struct data datas[16];
data *data, *Prv = datas;//在 C 中应是 struct data *data, *Prv = datas;
```

C + + 中，结构声明可以嵌套，即结构内的某一个数据类型又是一个结构。例如：

```
struct S {
    .
    .
    .
    struct S2{
          .
          .
          .
    {
    .
    .
    .
}x;
```

C 语言中，定义了一个 struct，就等于建立了一个结构属性单元，以后即可把它作为一个类型说明符来使用，且在声明具有所定义的结构类型的变量时关键字 struct 必不可少。而在 C + + 中，只需用所定义的结构类型名即可声明变量，而不需带关键字 struct。在上面例子的注释中列出了 C 的结构用法。

结构在 C + + 中是十分有用的构造数据类型，为了帮助读者理解，程序清单 3-13 列出了一个较完整的结构应用示例。

**程序清单 3-13：在程序中使用结构**

```
//这是一个结构用法示例程序
//文件名：MyStructDemo.C
# include <stdio.h>

# define iSTRING15 15
# define iSTRING20 20
# define iNULL_CHAR 1

struct stboat{
      char sztype[iSTRING15 + iNULL_CHAR];
      char szmodel[iSTRING15 + iNULL_CHAR];
```

```
        char sztitle[iSTRING15 + iNULL_CHAR];
        int iyear;
        long lmotor_hours;
    float fsaleprice;
    }stused_boat;

    int main(void)
    {
        printf("\nPlease enter the make of the boat:");
        gets(stused_boat.sztype);

        printf("\nPlease enter the model of the boat:");
        gets(stused_boat.szmodel);

        printf("\nPlease enter the title number for the boat:");
        gets(stused_boat.sztitle);

        printf("\nPlease enter the model year for the boat:");
        scanf("%d",&stused_boat.iyear);

        printf("\nPlease enter the current hours on the motor for the boat:");
        scanf("%d",&stused_boat.lmotor_hours);

        printf("\nPlease enter the purchase price of boat:");
        scanf("%d",&stused_boat.fsaleprice);

    printf("\n\n\n");
    printf("a %d %s %s with title number # %s\n",
        stused_boat.iyear,stused_boat.sztype,
        stused_szmodel,stused_boat.sztitle);
    printf("currently has %ld motor hours",
        stused_boat.lmotor_hours);
    printf("and was purchase for $ %8.2f\n",
        stused_boat.fsaleprice);
    return 0;
    }
```

### 3.7.4 数组和字符串

数组是由一公共变量名引出的同类型变量的集合。在 C++ 中，所有的数组都存贮在一片连续的内存单元中，最低地址对应于第一个元素，最高地址对应于最后一个元素。数组

可以是一维的或多维的。数组中具体的元素通过下标来访问。

最常用的是字符数组,因为在C++中没有设置字符串类型,而使用字符数组。用这种方法代替字符串类型,功能更强大,使用也更灵活。

1.一维数组

一维数组声明的一般格式如下:

```
type var_name[size];
```

这里的type是数组的基本类型,基本类型定义了该数组每个元素的数据类型,size定义该数组所包含的元素的个数。例如,下面声明了一个名为sample,包含10个元素的整型数组:

```
int sample[10];
```

数组中第一个元素的下标是0,第二个元素的下标是1,依次类推。

可以给数组分配一个值表对它进行初始化,每个值之间用逗号隔开,所有这些值括在{}中。例如:

```
int array[5]={1,2,3,4,5};
```

即:array[0]=1,array[1]=2,...array[4]=5。

C++中数组是一个复合数据类型,其元素可以是任何数据类型:整型、实型、字符型、数组类型、指针类型和结构类型等等。

2.无边界检查

C++对数组不进行边界检查,也就是说如果赋值时发生越界,则所赋值就会超出数组边界而赋给别的变量,甚至赋值给程序代码。发生这种情况时,既不产生编译错误,又不发生运行时错误,但极有可能导致程序失败。基于这些原因,程序员必须在必要时进行边界检查。

除了一维数组,C++中还可声明多维数组,声明数组时,若包含一个方括号[]就表示一维数组;若包含多个[],则表示多维数组。例如int array[5][6]。

3.字符串及字符串函数

一维数组最常见的用法是产生字符串。在C++中,字符串定义为以空字符结尾的字符型数组。空字符定义为"\0"(其值为0)。由于空字符作为结尾标志,因此在声明字符组时就需为所需长度加1。例如,char str[11]只能用来存贮10个字符。

在C和C++中有许多与字符串相关的函数,它们分别是:puts函数、gets函数、sprintf函数、fgets函数,fputs函数、strcpy函数、strcat函数、strcmp函数和strlen函数。下面简要介绍一下这些字符串函数:

- gets函数:从标准输入设备上读取字符串。使用该函数时,将以\n换行符结束字符串。
- puts函数:将字符串输出到屏幕上,语法形式为:puts(字符串)。用puts函数输入的

字符串都自动加上“\n”换行符。

- fgets 函数:从输入设备上读取指定的字符串,与 gets 的区别是 fgets 函数要限定输入字符串的长度。
- sprintf 函数:处理带有特殊格式字符的字符串。
- strcpy 函数:字符串拷贝函数,其语法形式为:

```
Strcpy(str1,str2);
```

表示把 Str2 字符串拷贝到 Str1 字符串内。

- Strcat 函数:用于字符串连接,其语法形式为:

```
Strcat(Str1,str2)
```

表示把 Str2 连接在 Str1 后。

- Stremp 函数:用于字符串比较,其语法形式为:

```
Strcat(Str1,str2)
```

表示把 Str1 和 Str2 进行比较,若二者相同,则返回零;若 Str1 小于 Str2,则返回负值;若 Str1 大于 Str2,则返回正值。

- Strlen 函数:计算字符串长度,其语法形式为:

```
Strlen(Str);
```

表示返回字符串 Str 的长度。该长度不包含字符串的结束字符“\0”。

程序清单 3-14 列出了一个示范程序,此程序综合运用了多维数组的许多特性,包括多维数组的定义、初始化,以及在程序中引用数组作为参数。

**程序清单 3-14:数组示范程序**

```
//此程序演示了如何定义、传递和使用多维数组
//文件名:MyArrayDemo.C
#include <iostream.h>

void vdisplay_results(char carray[][3][4]);

char cglobal_cubu[5][4][5]={
                    {
            {'P','L','A','N','E'},
            {'Z','E','R','O',''},
            {'','','','',''},
            {'R','O','W','','3'}
                    }
                    {
            {'P','L','A','N','E'},
```

```
                {'O','N','E','',''},
                {'R','O','W','','2'}
                        }
                        {
                {'P','L','A','N','E'},
                {'T','W','O','',''}
                        }
                        {
                {'P','L','A','N','E'},
                {'T','H','R','E','E'},
                {'R','O','W','','2'},
                {'R','O','W','','3'}
                        }
                        {
                {'P','L','A','N','E'},
                {'F','O','U','R',''},
                {'R','O','W','','2'},
                {'A','B','C','D','E'}
                        }
            };

    int imatrix[4][3] = {{1},{2},{3},{4}};

    main()
    {
        int irow_index,icolumn_index;
        char clocal_cube[2][3][4];
        cout << "sizeof clocal_cube = " << sizeof(clocal_cube) << endl;
        cout << "sizeof clocal_cube[0] = " << sizeof(clocal_cube[0]) << endl;
        cout << "sizeof clocal_cube[0][0] = " << sizeof(clocal_cube[0][0]) <<
endl;
        cout << "sizeof clocal_cube[0][0][0] = " << sizeof(clocal_cube[0][0][0]) <
< endl;

        vdisplay_reults(clocal_cube);

    cout << "cglobal_cube[0][1][2] is" << cglobal_cube[0][1][2] << endl;
    cout << "cglobal_cube[1][0][2] is" << cglobal_cube[1][0][2] << endl;
    cout << "\nprint part of the cglobal_cube's plane 0\n";

        for(irow_index = 0;irow_index < 4;irow_index + + ){
```

```
            for(icolumn_index=0; icolumn_index<5; icolumn_index++)
    cout<<cglobal_cube[0][irow_index][icolumn_index];
            cout<<endl;
      }

      cout<<"\nprint all of imatrix\n";
      for(irow_index=0;irow_index<4;irow_index++){
            for(icolumn_index=0; icolumn_index<3; icolumn_index++)
    cout<<imatrix[irow_index][icolumn_index];
            cout<<endl;
      }

      return 0;
}

void vdisplay_results(char carray[][3][4])
{
      cout<<"sizeof carray="<<sizeof(carray)<<endl;
      cout<<"sizeof carray[0]="<<sizeof(crray[0])<<endl;
cout<<"sizeof cglobal_cube="<<sizeof(cglobal_cube)<<endl;
cout<<"sizeof cglobal_cube[0]="<<sizeof(cglobal_cube[0])<<endl;
}
```

### 3.7.5 指针

理解指针并正确地使用指针是大多数成功的 C++ 程序设计的关键，原因有三：第一，指针为函数提供了修改调用参数的手段；第二，指针用支持 C++ 的动态存贮分配程序；第三，很多情况下，指针可用于代替数组以提高效率。此外，C++ 的许多功能都依赖于指针，因此，彻底理解指针非常重要。

另外要说明一点，指针既是 C++ 中一种最强大的功能，也是一种最危险的功能。例如，使用未初始化的指针或误用指针可能导致系统崩溃，而且，偶然错用指针引起的错误很难查找。

1.指针即地址

指针是保存另一对象内存地址的变量 。该地址一般是另一变量的内存位置，当然，也可能是端口或其他的专用 RAM 地址。如果一个变量保存另一个变量的地址，则称第一个变量指向第二个变量。

2.指针变量

要使指针成为变量，必须把变量声明为指针。指针声明由基本类型、星号和变量名组成。声明指针的一般形式为：

```
type * var_name;
```

其中,type是任一有效的C++数据类型,var_name是指针变量名。指针的基本类型决定指针变量可以指向的变量类型。例如,下面的语句声明了一个字符指针和两个整型指针:

```
char * p;
int * temp,* start;
```

3.指针操作符

C++中有两个专门的指针操作符:& 和 * 。& 是一元操作符,返回其操作数的内存地址。例如,语句 count_addr = &count 把变量 count 的内存地址赋值给 count_addr。这个地址是变量在计算机内存中的位置,它与变量 count 的值无关。由于操作符 & 返回其后变量的"地址",因此,可以把这条赋值语句描述为"count_addr 接受 count 的地址"。

第二个一元操作符 * 返回指针变量所指地址处的变量值。例如,如果 count_addr 保存 count 的内存地址,那么 val = * count_addr 语句将把 count 的值赋给 val。这样,这条语句可理解为"val 接受地址 count_addr 处的值"。请注意,这个指针操作符与乘号正好相同,容易引起混淆。

4.指针运算

指针只能参加两种运算:加和减。指针每增加1,就指向其下一个元素的内存单元,同样,指针每减少1,则指向前一个元素的内存单元。当然,指针并非只能加1或减1,它也可以加上或减去另一个整数。两个指向同一元素的指针可以相减,相减结果等价于指针所指向的两个元素的下标相减结果。也就是说,只有当两个指针指向同一个数组的不同元素时相减才有实际意义。

5.指针与数组

指针与数组关系密切。当指针指向数组的第一个元素时,几乎可以认为不带下标的数组名是指向数组的指针。考虑下面的语句:

```
char str[80], * pl;
pl = str;
```

在这里 pl 指向 str 的第一个元素。str[4]和 *(pl+4)都可得到第5个元素的值。

实质上,这两种方法都可实现数组的存取,但指针的运算速度要快于数组下标的运算速度。如果指针指向数组,也可以给指针加下标,如 pl[3]的值与 st[3]的值是同一个值。

指针也可以像其他数据类型一样数组化,即构成数组的每一个元素都是指针,声明格式如下:

```
数据类型 * 数组名[数组大小]
```

例如,语句 int * x[10]声明了一个指针数组,该数组的元素为指向整型变量的指针。

6.指针的初始化

声明指针之后,在给指针赋值前,指针的值是未知的。使用未赋值的指针可能使程序甚

至操作系统崩溃。按惯例,未指向任何对象的指针应赋空值,以标明该指针没有指向任何对象。然而,把空值赋给指针,并不能保证指针就"安全"了。如果在赋值语句的左边使用空值指针,那么仍将冒着使程序和操作系统崩溃的危险。因此,在对指针赋值时,一定要检查其是否为空值。

7.无类型指针

可以创建一个 void 指针,而后设置它等于另一个指针的值,它可以指向任意的数据类型。例如:

```
double * doublePtr;//declare a double pointer
double N2 = 4.1416;//declare a double Variable
void * aptr2;//declare a generic pointer
aptr = doublePtr;//Now it also points to double.
```

### 3.7.6 typedef 声明语句

在 C++ 中,可以使用 typedef 关键字来创建一个新的数据类型。使用 typedef 声明语句可以给已经由 C++ 定义过的任何类型标记一个新名字。最简单的 typedef 定义语法如下:

```
typedef type_name alternate_name;
```

在这里,type_name 是一个已有的类型名,alternate_name 是这个 type_name 的别名。在 C++ 中。用 typedef 重新定义数据类型可以提高程序的可靠性和可移植性。

## 3.8 控制结构

编程时,控制结构是通过控制语句来实现的。控制语句控制程序的执行流程,共分三类:第一类是条件语句和 switch 语句;第二类是循环控制语句 while,for 和 do while;最后一类是无条件转移语句 goto。

### 3.8.1 条件语句

C++ 语言支持四种基本的条件语句:if 语句,if...else 语句,?:条件语句和 switch 语句。

1.if 语句

if 语句的语法可描述如下:

```
if(表达式)
语句1;
else
语句2;
```

if 语句首先测试表达式的值,如果表达式的值不为 0,那么执行语句 1;如果表达式的值为 0,那么执行语句 2。这样,if 语句有选择地执行程序的一部分。if 语句的 else 及语句 2 部

分可以省略。有时为了测试多个条件,可以重复加入多个 else if,这样的 if 语句具有下列形式:

```
if(表达式1)
语句1;
else if(表达式2)
语句2;
else if(表达式3)
语句3;
```

if 语句的表达式可以有多种形式,前面已介绍过用算术表达式对变量赋值的例子,这里介绍另一种表达式——关系表达式,它用来比较两个值。例如,为了测试两个值是否相等,可以用关系操作符"==”来完成测试:

```
if(Number_of_peres==1)
cout<<"This is 1 penes"<<endl;
else
cout<<"There are"<<Number_of_peres<<endl;
```

若要测试一个值是否小于另一个值,可以使用操作符"<”,例如:

```
if(Number_of_peres<200)
cout<<"penes are less than 200"<<endl;
```

2.?:条件操作符

该条件操作符的语法形式如下:

```
表达式? 语句1:语句2;
```

例如,语句:

```
result=(a>b)? a:b;
```

与语句:

```
if(a>b)
result=a;
else
result=b;
```

是等价的。

3.switch 语句

switch 语句用于测试某一变量具有多个值时所执行的不同动作,语法形式如下:

```
switch(switch_expression){
case 常数1:
语句序列1
```

```
[break;]
case 常数 2:
语句序列 2
[break;]
⋮
default :
    语句序列 n
}
```

在执行 switch 语句时,首先在 case 条件下寻找相符的语句,找到后就执行有关的语句序列,直到遇上 break 语句或 switch 语句的结束符号才结束。default 部分表示当所有 case 条件都不符合时所要执行的语句序列。switch 可以嵌套。

### 3.8.2 循环语句

C++ 循环语句允许一系列语句重复执行,直到满足某个条件。C++ 包含三种循环控制语句:for 语句、while 语句和 do ...while 语句。下面依次介绍这三种循环语句。

1.for 语句

for 语句的语法描述如下:

```
for ( [initialization] ; [condition] ; [expression] ) statement
```

其中,initialization 为循环设置初始变量,可以是表达式或变量,在第一次执行循环时被初始化。condition 是条件表达式,其运算结果必须是 TRUE(非 0)或 FALSE(0)。为 FALSE 时,停止执行 statement 部分。expression 每执行一次循环被执行一次,expression 通常以某种方式增加或减少初始化变量 initialization。statement 部分当 condition 为真时,重复执行。

例如:

```
int i;  //初始化
for(i=0;i<=10;i++) cout<<i*i<<" ";
```

2.while 语句

第二类很有用的循环是 while 循环,它的语法描述如下:

```
while ( <condition> ) <statement>
```

其中,statement 部分重复执行,直到 condition 的值为假。如果没有给出 condition 部分,将执行无穷循环,等效于 while(true)。如果第一次 condition 为假,则 statement 一次也不执行。

3.do...while 语句

for 和 while 循环都是先测试条件,如果条件满足,则执行循环体中的语句,否执行循环体后面的语句,而在 do...while 语句中,条件测试是在循环体之后进行的,也就是说,循环体至少被执行一次。

do…while 语句的语法形式如下：

```
do statement while ( condition );
```

其中，statement 部分重复执行，直到 condition 的值为 false 时才停止执行 statement，因此 statement 部分至少被执行一次。do... while 语句也可以嵌套。

### 3.8.3 转移语句

C++语言转移语句包含 break 语句、continue 语句和 goto 语句。

1.break 语句

break 语句用于强制退出循环语句及中断 case 语句。用 break 语句退出循环后，将直接执行循环体后面的语句。

2.Continue 语句

continue 语句并非完全退出循环，而是使循环重新回到测试位置。

3.goto 语句

goto 语句是无条件转移语句，其语法形式如下：

```
goto Label _ name;
```

其中，Label _ name 是标号名，标识程序要转到的位置。

**注意：**

标号的用法和变量不一样，必须在其后面加上冒号，例如：

```
Label:
  i++
  ⋮
if i>5
goto Label;
  ⋮
```

## 3.9 函　　数

函数是由完成一定功能的语句序列所组成的独立模块，是结构化程序设计的基本结构。

1.函数的一般形式

函数的一般形式为：

```
函数类型(数据类型参数1,数据类型参数2,...)|
函数体(语句序列)
|
```

函数类型指明此函数是使用 return 语句返回值的类型，它可以是任何数据类型或 void。

形式参数说明表(数据类型参数1,数据类型、参数2,....)是以逗号分隔的变量类型和变量名表,调用函数时,这些变量接受实参值。函数可以没有形式参数,但圆括号是必须有的。函数体是函数功能的实现部分。

C++中,允许程序内含有多个名称相同的函数,这就是函数重载。下面声明的三个函数具有相同的函数名,但每个函数的自变量不同:

```
input(int i);   //输入一个整数;
input(float f);  //输入一个浮点数;
input(char ch);  //输入一个字符;
```

2.函数的参数

调用函数时一般需要向被调函数传送一些参数,为了保证参数的正确传送,C++要求每个函数都应有原型,也就是说在定义它之前应对形参类型及个数加以说明。

3.主函数的参数

有时在运行程序时需要把信息传送到程序中,主函数 main()通过参数可以接受这些信息。主函数有三个参数,它们分别是 argc、argv 和 env。argc 和 argv 用于接受命令行参数,env 用户接受程序运行于其上的操作系统的环境参数。

参数 argc 是整数,保存命令行参数个数。argc 至少为1,因为程序名被限定为第一个参数。参数 argv 是指向字符串数组的指针,数组中每个元素指向一个命令行实参,所有命令行实参都是字符串,如果必要,则由程序加以转换。命令行实参之间用空格和制表符分开。主函数 main()也可以带参数。

## 3.10 类与对象

类和对象是面向对象程序设计的基础,也是 C++区别于 C 语言的主要特征。面向对象技术具有全新的设计思想。客观世界是由各种各样的对象组成的,各种对象又有各自的内部属性和运动规律。面向对象技术就是面对客观世界,以对象为基本单元来分析、设计和实现一个系统。对象是系统的核心,它具有的如下基本特征:

(1) 模块性和隐蔽性。一个对象是一个可独立存在的实体,从外部看这个模块,只须了解这个模块具有哪些功能,至于这个模块的内部状态以及实现这些功能的细节都是不可见的,即隐蔽在模块内部。

(2) 继承性和类比性。每个具体的对象都在它所属的某一类对象(类)的层次结构中占据一定的位置。因此,下一层次的对象应具有上一层次对象的某些属性,这种性质称为继承。另一方面,当人们发现一些不同的对象具有某些相同的属性时,也常常把它们归并成类,这种归并称为类比性。

(3) 动态连接性。各种各样的对象以及它们之间的相互连接作用构成了不同的系统。因此,我们把对象和对象之间所具有的统一、方便以及动态传递消息的能力与机制称为动态连接性。

(4) 易维护性。任何一个对象都把如何实现本对象功能的细节隐藏在该对象的内部，因此无论是完善本对象的功能还是改变功能实现的细节，都在该对象内部处理，而不传给外部，这就增强了对象和整个系统的可维护性。

面向对象的程序设计提出了一些新概念，如类(class)、对象(object)、封装(encapsulation)、继承(inheritance)和多态性(ploypnorphism)等等，下面分别介绍。

### 3.10.1 类的定义

类(class)是指具有相同的结构和操作方法，并遵守相同规则的对象的集合。类的语法结构为类 = 数据 + 操作(函数)

类的定义格式如下：

```
class class_name{
private:
      私有数据成员和成员函数(缺省时即私有)
protected:
      保护数据成员和成员函数
public:
      公有数据成员和成员函数
};
```

其中，数据成员定义了该类具有的数据结构，成员函数则定义了可施加于这些数据的操作(方法)。private、protected 和 public 是访问权限符，它们将类的主体分为三部分，每部分都有数据成员和成员函数，但不同的访问权限符规定了这些部分具有不同的访问级别。

- public 表示的公有部分是全局透明的，它的数据成员和成员函数是开放的，既可由本类的成员函数访问，也可由程序的其他部分直接访问。
- private 表示的私有部分是完全隐藏的，它只能由本类的成员函数访问，而不允许程序的其他部分直接访问。
- protected 表示的保护部分介于以上两者之间，是半透明的，它可由本类的成员函数和派生类的成员函数直接访问。保护部分主要用于类的继承性。

程序清单 3-15 说明了一个简单的类的例子。

**程序清单 3-15:一个简单的类**

```
//简单的类例子
//文件名:Myclass.C
#include<iostream.h>
#include<math.h>

class counter{
      double data_value; //声明一个私有类型数据
public:
```

```
    void set_value(double);//声明一个公有类型的成员函数
    int get_value(void);//声明一个公有类型的成员函数
}sam;//声明类对象 sam

void  counter::set_value(double v )//成员函数的具体定义
{
    data_value=v;
}

double countor::get_value(void)//成员函数的具体定义
{
    double dd;
    dd=data_value;
    return(dd);
}

int counter::get_new(void)//成员函数的具体定义
{
    int ii;
    ii=int(data_value);
    return(ii);
}

main()//主函数
{
    sam.set_value(50.0);//设置初值
    cout<<"the value is:"<<sam.get_valuec<<endl;
    cout<<"the num is:"<<sam.get_new<<endl;
    return(0);
}
```

### 3.10.2 对象的定义

class 定义的是“类”类型，具有“类”类型的变量称为类的实例变量，类的实例称为对象。每个对象都是类的一个实例，各自具有自己的数据值，而数据结构由类统一定义。对象的声明与基本类型的变量声明一样，例如，要声明类 my_String 的对象 String1 和 String2，可用下面的语句。

```
my_String String1,String2;
```

对象 String1 和 String2 都具有自己的数据值，而数据值的类型和访问权限由类 my_String 定义。因此，程序中的其他部分不能直接访问它们。

对象的使用方法与结构的使用方法相似,例如:

```
String1.Set_String("Hello");
String2.Set_String("Visual C++");
String1.Display_String();
String2.Display_String();
```

另外,也可以在声明类之后再单独声明类的对象,例如:

```
class example //声明类 Example
{
      int i;
public:
      float f;
      void fun();
};
example A;//声明类对象 A
```

程序 3-16 演示了类与对象的关系。

**程序清单 3-16:类与对象关系示范程序**

```
//演示类与对象的关系
//文件名:MyClassObject.C
#include <iostream.h>
#include <string.h>
#include <stdio.h>

//以下是类定义部分
class mystring{
private:
  char string[30];
public:
  void set_string(char string[]){
  strcpy(this->string, string);
  }
  void display_string(void){
     cout << string << endl;
  }
};//类定义结束

//以下是主函数
void main(void){
    mystring thestring;
```

```
    thestring.set_string("hello");
    thestring.display_string();
    thestring.set_string("Visual C++ 6.0");
    thestring.display_string();
}//主函数结束
```

该程序运行结果为:Hello

```
Visual C++ 6.0
```

### 3.10.3 构造函数与析构函数

构造函数(constructor)是一种特殊的类成员函数,用于初始化类变量或分配内存空间。构造函数的定义方式和其他成员相同,但是构造函数用类名作函数名,它不能指定返回类型和显式的返回值。若定义了构造函数,则每当该类的对象被创建时,编译器都将自动调用构造函数初始化创建的对象。一个类可以有多个构造函数,当然,这些构造函数的形参各不相同——或个数不同或参数类型不同,或兼而有之。

为了建立类 cdate 的构造函数,在类定义中应增加下面的语句:

```
cdate(int year, int month, int day);
```

构造函数的孪生兄弟是析构函数(deconstructor),当对象要撤消时,系统将自动调用析构函数来清理对象所占的内存。

析构函数与构造函数具有相同的名字,但必须在前面加上"~"符号。当使用 delete 操作符或某个对象已执行完时,系统都会自动调用析构函数。析构函数不能带有参数,而且不可以重载。

程序清单 3-17 演示了构造函数的定义与使用方法。

**程序清单 3-17:构造函数的定义和使用方法**

```
//此程序演示了构造函数
//文件名:MyConst.C
#include <iostream.h>
class pdate{
private:
    int year,month,day;
public:
    pdate(){
        year = 1980;
        month = 1;
        day = 1;
    }//缺省构造函数

    pdate(int newyear, int newmonth,int newday){
```

```
        year = newyear;
        month = newmonth;
        day = newday;
    }//复制构造函数

    int get_year(void){
        return year;
    }

    int get_month(void){
        return month;
    }

    int get_day(void){
      return day;
    }
};//结束类定义

void main(){
    pdate date1(1995,3,4);
    pdate date2(1998,12,13);
    pdate date3;
    cout << "the data1 is" << date1.get_year() << "/"
        << date1.get_month() << "/" << date1.get_day() << endl;
    cout << "the data2 is" << date2.get_year() << "/"
        << date2.get_month() << "/" << date2.get_day() << endl;
    cout << "the data3 is" << date3.get_year() << "/"
        << date3.get_month() << "/" << date3.get_day() << endl;
    cout << date3.setdate(1997.12,23) << endl;
    cout << "the data3 is" << date3.get_year() << "/"
        << date3.get_month() << "/" << date3.get_day() << endl;
}//结束主函数
```

程序清单3-18演示了析构函数的定义与使用方法：

**程序清单3-18:析构函数例子**

```
//演示析构函数
//文件名:MyDeconst.C
#include<iostream.h>

class cdate{
```

```
private:
   int year,month,day;
public:
   cdate(int new_year,int new_month,int new_day);
   cdate();
   displaydate(void){
      cout<<"the data is"<<year<<"/"<<month<<"/"<<day<<endl;
   }
};//结束类定义
//构造函数定义
cdate::cdate(int new_year,int new_month,int new_day){
    year=new_year;
    month=new_month;
    day=new_day;
}//构造函数定义结束

//析构函数定义
cdate::cdate(){
   cout<<"destructor is called"<<endl;
}//析构函数定义结束

void main(void){
   cdate date(1997,12,11);
   date.displaydate();
   cout<<"exiting main"<<endl;
}//结束主函数
```

当使用对象指针进行动态内存分配时,若要撤消对象,必须使用 delete 操作符,以释放动态内存。

析构函数不是在程序结束时调用的,而是在执行 delete 释放对象所占动态内存时自动调用的,即当 delete 执行时,将自动调用析构函数。

通常将 delete 放在析构函数中,例如:

```
cdate::cd{
  delete pdate;
}
```

### 3.10.4 内联成员函数

成员函数在类的体内定义或体外定义是有重大区别的,在类的体内定义的函数被看作内联函数,而在类的体外定义的函数缺省情况下都是非内联函数(外联函数)。通过使用关键字 inline,可以将外联函数变成内联函数,例如:

```
class rectangle //定义类 rectangle
{
      int left;
      int top;
      int right;
      int bottom;
public:
      void get coord(int * l,int * t,int * r,int * b);
};
void inline getcoord(int * l, int * t, int * r ,int * b)//将成员函数 getcoord 定义
成内联函数
{
   * L = left;
   * t = top;
   * r = right;
   * b = bottom;
}
```

### 3.10.5 this 指针

this 指针是一个隐含的指针,每个成员函数都有一个 this 指针变量。this 指针用于指向以该成员函数所属类定义的对象,当某个对象调用该成员函数时,成员函数的 this 指针便指向该对象。使用 this 指针时,应引用操作符 - >,例如:

```
this - > Set _ String("Welcome to use Visual C + +");
this - > display _ string();
```

当调用成员函数时,自动向它传递一个变元,该变元是一个指向调用它的对象的指针。程序清单 3-19 演示了 this 指针的用法。

**程序清单 3-19:this 指针的概念和用法**

```
//this 指针的概念和用法
//文件名:Mythis.C
# include < iostream.H >
class cdate{
private:
     int Year, Month, Day;
public:
     void SetDate(int New _ year,int new _ month,int new _ day){
          this - > Year = new _ year;
          this - > Month = new _ month;
          this - > Day = new _ day;
```

```
    }

    int GetYear(void){
        return this->Year;
    }

    int GetMonth(void){
        return this->Month;
    }

    int GetDay(void){
        return this->Day;
    }
};//结束类定义

void main(void){
    CDate Date;
    Date.Set_Date(1997.11.3);
    cout<<"The data is"<<Date.GetYear()<<"/"<<Date.GetMonth()<<"/"
Date.GetDay()<<endl;
}//结束主函数
```

### 3.10.6 静态数据成员和静态成员函数

类包含静态的数据成员和成员函数。类的数据成员和成员函数对类的每个对象来说都是私有的，若希望该类的所有对象共享某个数据，可以把该数据声明为静态数据成员。静态数据成员与函数中的静态局部变量的作用相似，静态数据成员被分配一个固定的存储空间，该类的每个对象都可共享这个内存空间，即所有对象只有一个静态成员的值。

因此，静态数据成员实际上是该类的全局变量。如果某个对象改变了静态数据成员的值，则会影响所有的对象。静态数据成员声明时，在其前面增加关键字 static 即可。例如：

```
class Test//声明类 Test
{
public:
    static int Count;//静态数据成员
}
```

静态成员函数有以下一些特点：

(1) 类外代码可以使用类名和作用域操作符来调用静态成员函数，而无需引用类的实例，甚至类的实例可以不存在。

(2) 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。这是因为静态

成员函数没有 this 指针来存放对象的地址，因此如果要访问非静态数据成员，编译器将无法确定访问的数据成员属于哪个对象。

### 3.10.7 友元函数

C++提供了一种特殊的函数，允许访问类的私有类型数据及保护类型数据，这种函数称为友元函数(friend function)。友元函数的声明方式如下：

```
class 类名称
{
   type vars;
    ⋮
public:
   friend 函数类型,函数名称();//友元函数
    ⋮
}
```

程序清单 3-20 是一个使用友元函数的例子。

**程序清单 3-20：在程序中使用友元函数**

```
// 使用友元函数建立类之间的联系"桥梁"
//文件名:Myfriend.C
#include<iosteam.h>

class business; //向前引用类
//定义 home 类
class home{
private:
      float rent, car,  groceries;
public:
      home(float r, float c, float g){
            rent=r;car=c; groceries=g;
      }
      friend void print(home &, Business &); //声明友元函数
};
//定义 business 类
class Business{
      float office, software,telecom;
public:
      Business(float o, float s, float t){
          office=o;software=s;telecom=t;
      }
```

```
    friend void print (home &,Business &);//声明友元函数
    };
    void print(home hm, Business bz){
        cout << "\nrent:" << hm.rent;
        cout << "\ncar:" << hm.car;
        cout << "\ngroceries:" << hm.groceries;
        cout << "\noffice:" << bz.office;
        cout << "\nsoftware:" << bz.software;
        cout << "\ntelecom:" << bz.telecom;
        float total = hm.rent + hm.car + hm.groceries + bz.office + bz.software + bz.
telecom;
        cout << "\ntotal:" << total << "\n";
    }
    int main(){
        home hm (370.00,88.50,47.92);
        business bz(589.10,138.72,65.77);
        print(hm,bz);
        return 0;
    }
```

## 3.11 类 继 承

面向对象的程序设计中，允许某个类继承其他类的属性。被继承的类称为基类(base class)或父类(parent class)，继承的类称为派生类(derived class)或子类(child class)。

### 3.11.1 派生类与基类

基类与派生类具有如下关系：从任何已存在的类派生出的新类称为派生类，前者称为基类或父类。

派生类继承了基类的全部成员，并且还可以增加基类所没有的新数据成员和方法，以满足派生类的特殊应用。派生类是实现继承性的基础。

派生类的语法结构如下：

```
class <派生类>:<访问权限符>,<基类>
{
private:
  新增私有数据成员和成员函数
protected:
  新增保护数据成员和成员函数
public:
  新增公有数据成员和成员函数
```

```
|
```

访问权限有三种：public、protected 和 private，缺省时为 public。这三种访问权限的数据类型在派生类意义如下：

- 公有类型数据(public)：可供程序中任何函数访问。
- 私有类型数据(private)：仅供本类中的成员函数使用，这类数据不能传递给继承者。
- 保护类型数据(protected)：供成员函数使用，但可以传递给派生类使用。

在类的定义中，公有类型的数据允许其他函数访问，而私有类型数据只能由本类中的成员函数访问。但是，当派生类继承基类的特性时，派生类无法访问基类中的私有数据。若想让基类中的某些数据不仅可供成员函数访问，而且可将特性传给派生类，那么就可以将其声明成保护类型。

如果将派生类的访问权限声明为 public，那么所有基类的公有数据在派生类中仍是公有数据，所有基类的保护数据在派生类中仍是保护数据。如果将派生类的访问权限声明为 private，那么所有基类的私有数据和保护数据在派生类中都变成私有数据。

程序清单 3-21 是一个关于继承的简单的例子。

**程序清单 3-21：基类与派生类的关系**

```
//本程序说明基类与派生类的关系
//文件名 Myherit.C
class parent //声明基类
|
protected://保护数据供成员函数使用，也可传递给派生类使用
    int x;
public://公有数据，可供任何函数使用
    parent();
    void change();
    void display();
|;//parant 声明结束

parent::parent() //声明构造函数
|
  x=0;
  i=0;
|

void parent::change()//声明成员函数
{
    x++;
    i++;
}
```

```
void parent::display()//声明成员函数
{
    cout<<"x="<<x<<endl;
}

class son:public parent
//声明 son 为 parent 的派生类,访问权限为 public,即 parent 类的公有数据
//在 son 中仍为公有数据,parent 类的保护数据在 son 中仍为保护数据
{
public:
   void modify();//声明成员函数
};//派生类 son 的声明结束

void son::modify()//声明成员函数
{
   x++;
}
```

### 3.11.2 派生类的对象和构造函数

清单 3-21 中利用基类 parent 声明了一个派生类 son,声明派生类后,在主程序中就可以声明派生类的对象,如下所示:

```
void main()//主函数开始
{
    son A;//用派生类 son 来定义一个对象 A
    parent B;//用基类 parent 来定义一个对象 B
    cout<<"Display derived class Object A";
    A.display();//对象 A 的显示结果
    A.change();//调用从基类中继承的成员函数
    A.modify();//调用派生类中声明的成员函数
    cout<<"Display base class Object B";
    B.change();
    B.modify();
}
```

在这个例子中,派生类的对象 A 可以调用从基类中继承的成员函数,而基类的对象 B 却不可以调用派生类中新声明的成员函数。例如,B.modify()是非法的。

前面已说过,在声明类对象时,可以利用构造函数来设置类对象的初值。同样,在派生类中也可以利用构造函数来进行初始化工作。下面先声明派生类 son 的构造函数,然后声明派生类对象:

```
class son: public parent
{
public:
    son():parent()//声明派生类的构造函数
    {
        .
        .
        .
    }
void modify();
};
void main() //主程序开始
{
    son A;//声明派生类对象
        //此时,调用相应的构造函数 son()来进行初始化
      .
      .
      .
}
```

在主程序中,声明 son 类的对象 A 时,就会调用相应的 son()构造函数执行初始化操作,而调用 son()时,又会间接调用 parent 类的构造函数 parent()来设置初始值。

### 3.11.3 多重派生类

多重派生类是从一个基类派生出多个不同派生类的一对多关系。各个派生类除了具有基类的特性外,还可针对各自的需要加入属于自己的相应特性。

程序清单 3-22 是一个多重派生类的例子。

**程序清单 3-22:多重派生类的示例程序**

```
//本程序说明多重派生类的用法
//文件名:Mymulder.C
const int LEN = 80;

class price_class//定义基类 price_class
{
protected://定义保护类型的数据
    char goods_name[LEN];
    int b_price, p_price;
public:
    void get_price()//声明公有成员函数
```

```
    {
        cout << "Enter goods name:";
        cin >> g_name;
        cout << "Enter bcoks price:";
        cin >> b_price;
        cout << "Enter pens price:";
        cin >> p_price;
    }

    void display()//声明公有成员函数
    {
        cout << "goods name:" << g_name: << endl;
        cout << "books price:" << b_price: << endl;
        cout << "pens price:" << p_price: << endl;
    }
};//基类 price_class 声明结束
class cs:public price_class//声明派生类 cs
{
    int c_price;
public
    void get_price();//声明派生类的成员函数
    {
        price_class::get_price();//调用基类的成员函数
        cout << "Computer price:";
        cin >> c_Price;
    }
    void display()//声明派生类的成员函数
    {
      price_class::display;//调用基类的成员函数
      cout << "Computer price:" << c_price << endl;
    }
};//派生类 cs 声明结束

class ps:public price_class//声明一个派生类 ps
{
    int p_price
public:
    void get_price();//声明派生类的成员函数
    {
        price_class::get_price();//调用基类的成员函数
        cout << "milk price:" << p_price << endl;
```

```
    }
};//派生类 ps 声明结束

class chs:public price_class//声明派生类 chs

    int ch_price;
public:
    void  get_price()//声明派生类的成员函数
    {
        price_class::get_price();//调用基类的成员函数
        cout<<"Price:";
        cin>>ch_price;
    }
    void display();//声明该派生类的成员函数
    {
        price_class::display();//调用基类的成员函数
        cout<<"check price:"<<ch_price<<endl;
    }
};//派生类 chs 声明结束
```

在这个例子中,通过基类 price_class 派生出 cs、ps 和 chs 三个派生类。三个派生类除了都继承基类的特性外,各自定义了属于自己的私有数据和成员函数。

### 3.11.4 类的多重继承

由一个基类产生派生类实现的继承称为单继承。由多个基类产生派生类实现的继承称为多重继承。在单继承中,派生类只能有一个基类,基类和派生类构成链形结构;在多重继承中,派生类可以同时有多个基类,基类和派生类构成有向图。

多重继承的派生类可以有多个基类,它同时继承了这些基类的所有成员。因此,每个派生类的对象都包含了所有基类的数据成员和所有成员函数,也包含了基类的数据成员的值,就像每个基类的对象都作为它的一个无名成员一样。

多重继承的定义形式如下:

```
class <派生类>:<访问权限符><基类1>
                <访问权限符><基类2>
        .
        .
        .
                <访问权限符><基类n>
{
private:
    新增私有数据成员和成员函数
```

```
protected:
    新增保护数据成员和成员函数
public:
    新增公有数据成员和成员函数
};
```

程序清单 3-23 说明了多重继承的应用。

**程序清单 3-23:多重继承**

```
//本程序从点和圆这两个基类派生出一个圆柱类
//文件名:MyMulInherity.C
#include<iostream.h>

const float PAI=3.1415926;
class b_point
{
private:
   float pointx,pointy;
public:
   b_point(float x, float y){
        pointx=x;
        pointy=y;
   }
   void setpoint(float x, float y){
        pointx=x;
        pointy=y;
   }
   float getx(void){
        return pointx;
   }
   float getY(void){
         return pointy;
   }
};//类 b_point 声明结束

class b_circle
{
private;
    float radius;
public
    b_circle(float r=0){
```

```
        radius = r;
    }
    void set_circle_radius(float r){
        radius = r;
    }
    float getradius(void){
        return radius;
    }
    float get_area(void){
        return PAI * radius * radius;}
};//类b_circle声明结束

class d_cylinder:public b_point, public b_circle
{
private:
    float height;
public:
    d_cylinder(float x, float y, float r, float h):b_point(x,y),b_circle(ra-
dius)
    {
        height = h;
    }
    void setheight(float h){
        height = h
    }
float getheight(void){
    return height;
}
};//类d_cylinder声明结束

void main(void){
    d_cylinder_column(1.2,4.3,6.5,3.3);
    cout << "Circle Center:x = " << Column.getX() << ",y = " << column.getY() << 
endl;
    cout << "Circle Radius = " << column.getradius << endl;
    cout << "circle Area = " << column.getarea() << endl;
    cout << "Column Height = " << column.getheight() << endl;
    cout << "Column Volume = " << column.getArea() * column.getheight() << endl;
}
```

### 3.11.5 多层派生类

在 C++ 中,还有另一种继承,即多层派生类,它具有如下结构:

基类
继承
派生类 1
继承
⋮
继承
派生类 n
继承

程序清单 3-24 举例说明了多层派生类的用法。

**程序清单 3-24:多层派生类**

```
//本程序演示多层派生类的定义和使用方法
//文件名:MyMultClass.C
#include<iostream.h>
#include<stdlib.h>

const int Len=50;
class stack //声明基类 stack
{
protected:
    int head;
    int stack[Len];
public:
    stack(){head=0;};//声明基类的构造函数
    void push(int val)//声明基类的成员函数
    {
        head++;
        stack[head]=val;
    }
    void pop()//声明基类的成员函数
    {
        int temp;
        temp=stack[head];
        head--;
        return temp;
    }
```

```
};

class op_stack:public stack//声明第一层派生类 op_stack
{
public:
      void push(int val)//声明第一层派生类的成员函数
      {
          if(head > LEN){
             cout < <"stack overflow!" < < endl;
             exit();
          }
          else
          stack::push(val);
       }
       int pop()//声明第一层派生类的成员函数
       {
          if chead(1) {
             cout < <"stack underflow!" < endl;
             exit(1);
          }
          return stack::pop();
        }
};

class use_stack:public op_stack//声明第二层派生类 use_stack
{
public
      void producestack()//声明第二层派生类的成员函数
      {
          int val;
          cout < <"input data;";
          cin > > val;
          While((val < 0)||(val > 100)){
          cout < <"input Error!" < < endl;
          cout < <"please input again:";
          cin > > val;
          use_stack::push(val);
      }
   }
}
void main(){
```

```
    use_stack A;//声明第二层派生类的对象 A
    A.producestack();
    A.producestack();
    cout << "→" << A.pop() << endl;
    cout << "→" << A.pop() << endl;
}
```

## 3.12 重载(overload)

重载包含函数重载和操作符重载。

### 3.12.1 函数重载

函数重载是指两个或两个以上的函数取同一名字,而使用不同类型或不同数目的参数。函数重载的基本规则有如下两条:

- 函数必须包含两个不同参数的自变量表。
- 两个函数自变量表中的自变量至少有一个是不相同的。

程序清单 3-25 示范了函数的重载。

**程序清单 3-25:函数的重载**

```
//重载函数
//文件名:MyOverFun.C
#include<iostream.h>

//先声明要重载的函数
void printit (char *lab, char *instrument,int number);
void printit (int n);
void printit (char *message);

int main1()//调用重载的函数
{
    printit("高功率","计算机",18);
    printit(300);
    printit("There are enough ability to do .\n");
    return 0;
}
//重载函数
void printit (char *lab, char *instrument, int number)
{
    cout << lab << "\n";
```

```
    cout << instrument << "\n";
    cout << number << "\n";
}

void printit(int n){
    cout << n << "\n\";
}

void printit(char &message){
    cout << message << "\n\n";
}
```

### 3.12.2 操作符重载

操作符重载是将C++中的已有操作符赋予新的功能。使用操作符重载时,必须用以下方式来声明成员函数:

```
函数类型 operator#(形参表)
```

其中,operator是用于定义操作符的关键字,#表示欲定义的操作符,函数类型指明返回值的类型,通常与类类型一致或为void类型。

在C++中有三种通用的操作符:算术操作符、逻辑操作符和位操作符。可以将所有的三种类型的操作符重载,让它们去完成程序员需要的动作而取代其默认情况下的操作。

程序清单3-26是一个操作符重载的示例。

**程序清单3-26:操作符的重载示例程序**

```
//本程序演示操作符的重载
//文件名:MyOperaOver.C
#include<iostream.h>

class operclass//声明一个类
{
      int X;
public:
      operclass();//声明构造函数
      operclass operator++();
      //声明重载的操作符++,返回值类型operclass类,
      //这里的++为前置运算
      operclass operator++(int);
      //声明重载的操作符++,返回值类型operclass类,
      //这里的++为后置运算
```

```
    void display();//声明成员函数
};
operclass::operclass() //定义构造函数
{
   X=0;
}
void operclass::display()//定义成员函数
{
   cout<<"X="<<X<<endl;
}
operclass operclass::operator++()//定义重载前置操作符++的具体操作
{
    operclass A;
    X++; //进行正常的整数加1操作
    A.X=X;
    return A;
}
operclass operclass::operator++(int)//定义重载后置操作符++的具体操作
{
     operclass A;
     X++;//进行正常的整数加1操作
     A.X=X;
     return A;
}
void main()//主程序开始
{
    operclass X,Y;//声明两个对象
    X.display();//对象X调用成员函数display()
    ++X;//对象X调用前置操作符++
    Y++.display()//对象Y调用后置操作++,并调用成员函数display()
}
```

本例实现了单目操作符重载,有前置运算和后置运算之分。很多不同的操作符都可以按照不同的方法进行重载,下面列出了操作符重载的规则和技巧:

(1) 当操作符的操作数是一种固定数据类型时不能重载此操作符。例如,当一个二元操作符的两个操作数都为固定数据类型时不能重新定义它。在重定义这个二元操作符之前,至少它的一个操作数必须是一个对象。

(2) 如果想写一条语句,让此句中的某个操作符影响一个对象,那么必须重载此操作符使它正确地操作此对象。如果想把某个对象当做一个未重载的操作符的对象,编译程序将产生错误信息。

(3) 只能重载现有的操作符,不能发明新操作符来扩展C++。

(4) 不能修改一个操作符所操纵的操作数个数。

(5) 不能修改一个操作符的优先级。

(6) 不能修改一个操作符的组织形式。当一个操作数出现在两个具有相同优先级的操作符中间时,它或者与左边的操作符或者与右边的操作符相结合。

**表3-14 不能重载的操作符**

| 操作符 | 操作符名 | 为什么不能重载 |
|---|---|---|
| :? | 三元操作符 | C++中不存在定义一个三元操作符的语法 |
| . | 成员操作符 | C++的设计者认为维护此操作符对成员的访问能力的安全性非常重要 |
| :: | 域分解操作符 | 它是一种特殊类型操作符。语法上看它很像一个普通的操作符但它在左边的操作数是一个类型名,而不是一个表达式 |
| -> | 成员指针操作符 | C++的设计者认为维护此操作符对成员的访问能力的安全性非常重要 |
| Sizeof | Sizeof操作符 | 像域分解操作符一样,Sizeof操作符是针对名字的操作符 |

# 第 4 章

# Visual C++ 6.0 工具程序

Visual C++ 6.0 提供了许多实用的工具程序,这些工具一方面作为 Visual C++ 的有机组成部分,可以提高开发效率;另一方面,也可以作为独立的 Windows 应用程序而使用。本章介绍在开发 Visual C++ 6.0 应用程序中最常用的 WinDiff、Spy++ 和 Help Workshop 等工具,为读者学习和掌握 Visual C++ 6.0,并利用其开发实用的应用程序提供帮助。

## 4.1 WinDiff 应用程序

WinDiff 应用程序主要用来比较两个文件或两个目录之间的内容的差异性。利用它可以方便地比较和修改两个文件或两个目录之间的内容。WinDiff 的可执行文件为 C:\Program Files\Microsoft Visual Studio\Common\Tools\WINDIFF.EXE,这里的 C:\Program Files\Microsoft Visual Studio\ 为安装目录。

### 4.1.1 WinDiff 命令行

也可直接以命令行格式启动 Windiff 应用程序。以命令行格式启动 WinDiff 时不必再利用其菜单来设置比较选项。命令行格式如下:

```
WinDiff path1 [path2][-D][-O][-Nname][-S[option]Savefile]
```

各参数含义解释如下:

path1:比较 path1 目录和当前目录中的文件。

Path1、path2:比较 path1 目录和 path2 目录中的文件。

-D:只比较一个目录。

-O:提纲查阅(并不自动扩展目录)。

-Nname:在比较结束后使用 NET SEND name 命令发送通知。

-S[option]:使用下面的选项存储所有文件的列表:-s:比较两个目录中相同的文件;-l:只比较第一个目录中的文件;-r:只比较第二个目录中的文件;-d:比较两个目录中不同的文件。

Savefile:存储比较结果的文本文件名字。

### 4.1.2 WinDiff 应用程序界面

启动 WinDiff 应用程序后主窗口仅显示菜单及窗口用户工作区,当利用 File 菜单打开要比较的两个文件(或目录)时,主窗口将显示比较结果。

WinDiff 最常用来比较两个文件或两个目录。选择 File 菜单的 Compare Files 命令,打开 Select First File 对话框,如图 4-1 所示,在此选择要比较的第一个文件名,然后单击"打开"按钮,将打开 Select Second File 对话框,选择要比较的第二个文件名。选定两个文件后,就可以利用 WinDiff 比较它们了。

选择 File 菜单的 Compare D 精读 tories 命令,将打开 Select Directories 对话框,如图 4-2 所示,在 Dir1 文本框中输入要比较的第一个目录(文件夹)名,在 Dir2 文本框中输入要比较的

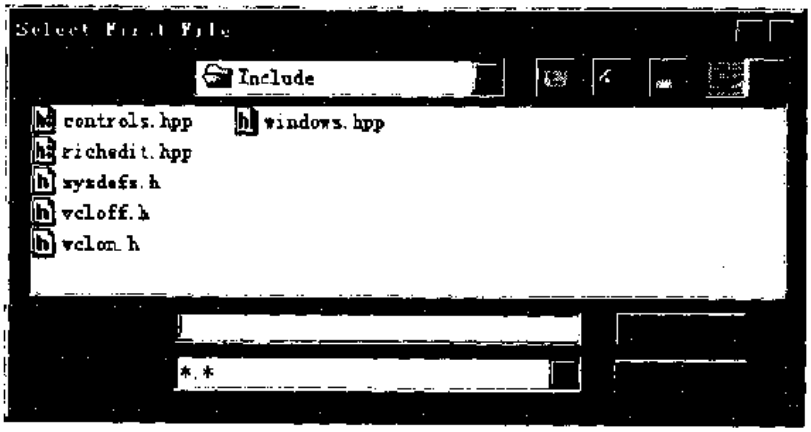



图 4-1　选择比较的第一个文件

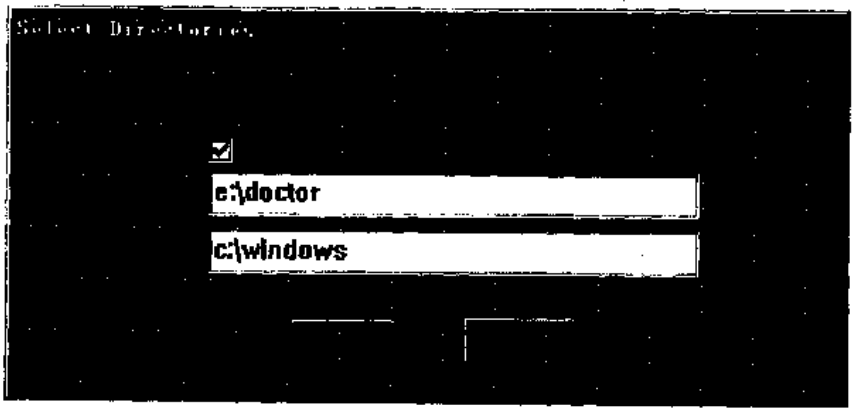
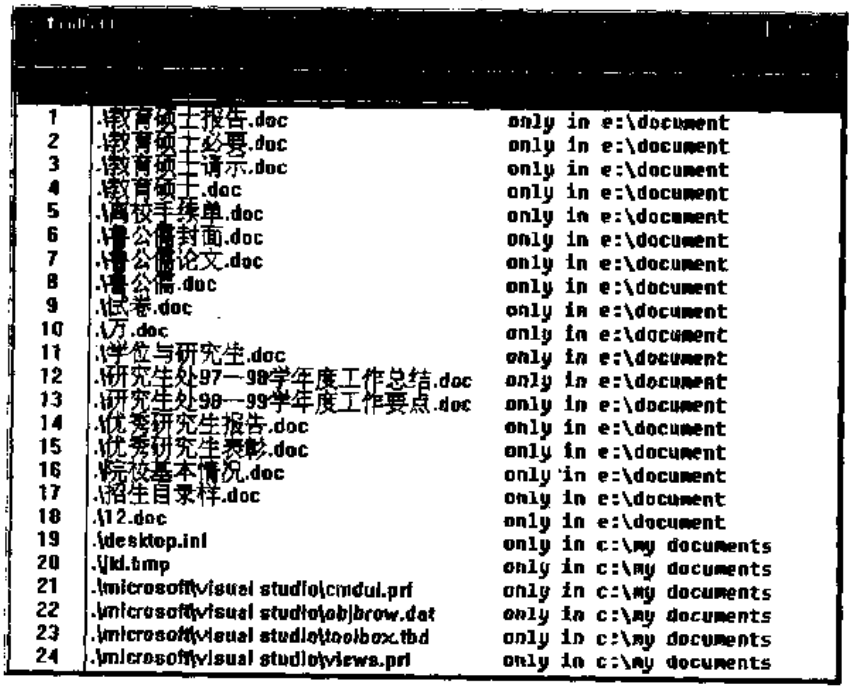



图 4-2　选择比较的第一个目录



| | | |
|---|---|---|
| 1 | .\教育硕士报告.doc | only in e:\document |
| 2 | .\教育硕士必要.doc | only in e:\document |
| 3 | .\教育硕士请示.doc | only in e:\document |
| 4 | .\教育硕士.doc | only in e:\document |
| 5 | .\离校手续单.doc | only in e:\document |
| 6 | .\博公信封面.doc | only in e:\document |
| 7 | .\博公信论文.doc | only in e:\document |
| 8 | .\博公信.doc | only in e:\document |
| 9 | .\试卷.doc | only in e:\document |
| 10 | .\万.doc | only in e:\document |
| 11 | .\学位与研究生.doc | only in e:\document |
| 12 | .\研究生处97—98学年度工作总结.doc | only in e:\document |
| 13 | .\研究生处98—99学年度工作要点.doc | only in e:\document |
| 14 | .\优秀研究生报告.doc | only in e:\document |
| 15 | .\优秀研究生表彰.doc | only in e:\document |
| 16 | .\院校基本情况.doc | only in e:\document |
| 17 | .\招生目录样.doc | only in e:\document |
| 18 | .\12.doc | only in e:\document |
| 19 | .\desktop.ini | only in c:\my documents |
| 20 | .\ld.tmp | only in c:\my documents |
| 21 | .\microsoft\visual studio\cmdui.prf | only in c:\my documents |
| 22 | .\microsoft\visual studio\obj\brow.dat | only in c:\my documents |
| 23 | .\microsoft\visual studio\toolbox.tbd | only in c:\my documents |
| 24 | .\microsoft\visual studio\views.prf | only in c:\my documents |

图 4-3　WinDiff 进行目录比较后的主窗口

第二个目录名，正确输入后，按回车键，WinDiff 主窗口将显示比较结果，如图 4-3 所示。

## 4.2　Spy + + 应用程序

Spy + + 应用程序是一个基于 Win32 的应用程序，利用它可以图形化地查阅系统的进程、线程、窗口以及窗口消息等信息。Spy + + 应用程序的可执行文件为 C: \ Program Files \ Microsoft Visual Studio \ Common \ Tools \ SPYXX.EXE，这里的 C: \ Program Files \ Microsoft Visual Studio \ 为安装目录。Spy + + 是一个标准的 MDI 应用程序，我们可以在其主窗口中打开多个视图作为其文档窗口。启动 Spy + + 后，其主窗口如图 4-4 所示。




图 4-4　Spy + + 的主窗口

### 4.2.1　Spy + + 工具条

Spy + + 工具条显示在主窗口的菜单条的下面。它提供了一系列的快捷命令，这些命令包括打开新视图、启动和停止消息流的显示、改变消息流选项、清除消息流窗口以及查找窗口等。View 菜单中的 Toolbar 命令可以显示或隐藏此工具条。

Spy + + 工具条上的按钮及其作用如表 4-1 所示。

表 4-1　Spy + + 工具栏按钮

| 按钮 | 快捷键 | 作用 |
|---|---|---|
| Windows | Ctrl + W | 建立一个窗口，以树形结构显示系统中所有窗口和控件 |
| Processes | Ctrl + P | 建立一个窗口，以树形结构显示系统中所有进程 |
| Threads | Ctrl + T | 建立一个窗口，以树形结构显示系统中所有线程 |

续表

| 按钮 | 快捷键 | 作用 |
| --- | --- | --- |
| Message Log | Ctrl + M | 建立一个窗口，用于显示窗口消息。按下该按钮将启动 Message Options 对话框，利用该对话框可以选择要显示其消息的窗口 |
| Find Window | Ctrl + F | 打开 Find Window 对话框，如图 4-5 所示。可以在此对话框中选择一个要查看其消息或属性的窗口 |
| Start/Stop Logging | F8 | 启动/停止消息登录并显示消息流，此按钮只有在一个消息窗口被激活时才有效 |
| Logging Options | Ctrl + o | 显示 Message Options 对话框，如图 4-6 所示。利用该对话框选择要查阅的窗口和消息类型，此按钮只有在一个消息窗口被激活时才有效 |
| Clear Log | Delete | 清除活动消息窗口的内容，此按钮只有在一个消息窗口被激活时才有效 |
| Find | Alt + F3 | 在当前视图中搜索匹配的窗口、进程、线程或消息 |
| Find Next | F3 | 搜索当前视图，寻找匹配的下一个窗口、进程、线程或消息，此按钮及其相关的选项只有在有多个有效搜索结果的情况下才有效。如在使用窗口句柄作为搜索准则时此按钮不可用，因为系统中窗口句柄具有唯一性 |
| Find Previou | Shift + F3 | 搜索当前视图，寻找匹配的下一个窗口、进程、线程或消息，此按钮及其相关的选项只有在有多个有效搜索结果的情况下才有效 |

### 4.2.2　Spy + + 的视图

启动 Spy + + 后，它自动打开一个标题为“Windows 1”的窗口，该窗口显示系统中所有窗口和控制的树形结构图，见图 4-4。除了窗口视图外，Spy + + 还有三个视图：进程视图、线程视图和消息视图。

1. 窗口视图

窗口视图显示了系统中所有的窗口和控制的树结构。在窗口视图中，当前的桌面系统窗口位于树结构的顶部，其他窗口都是桌面窗口的子窗口，这些窗口按其家族关系列表，兄弟之间以字母顺序排列。单击窗口前面的 + 或 - 就可以展开或折叠相应的窗口。

双击窗口视图中某一窗口的标题，将弹出 Window Properties 对话框，如图 4-7 所示。该

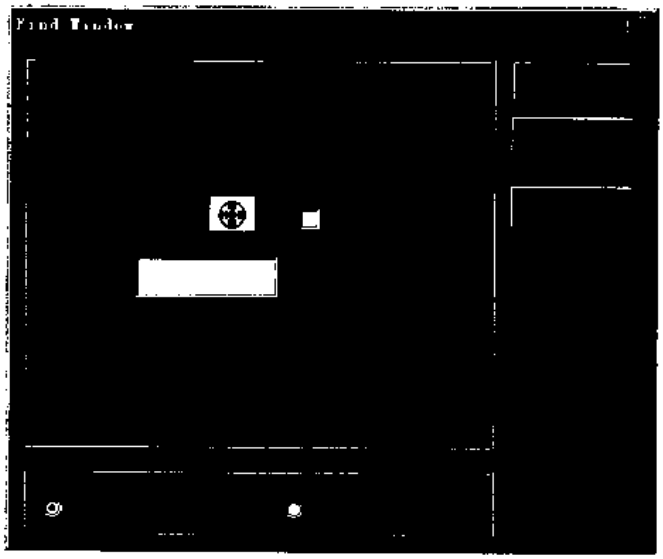


图 4-5　Find Window 对话框

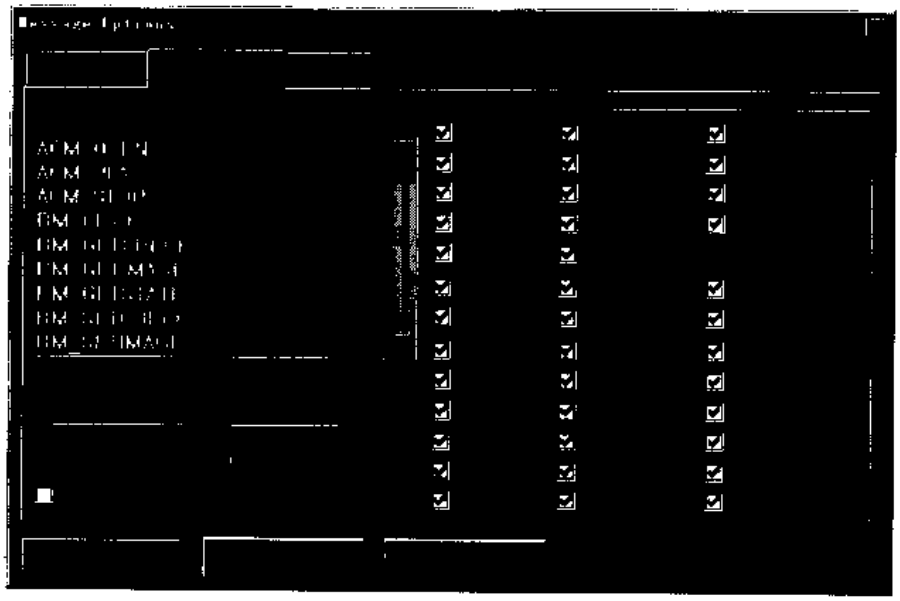


图 4-6　Message Options 对话框

对话框显示了所击窗口的详细信息。这些信息包括窗口的一般属性、风格、在家族中的位置、所属类和相关进程。

2. 进程视图

Win32 应用程序支持多进程，每个进程均可拥有一个或多个线程，而每个线程则可以拥有一个或多个相关联的顶层窗口，每个顶层窗口又可以拥有一系列子窗口。这样的分层关系，可通过 Spy + + 的进程视图以图形化的形式显示。

在 Spy 菜单中选择 Processes 命令，或单击工具条中的进程工具，或按 Ctrl + P 快捷键都可以打开进程视图，如图 4-8 所示。

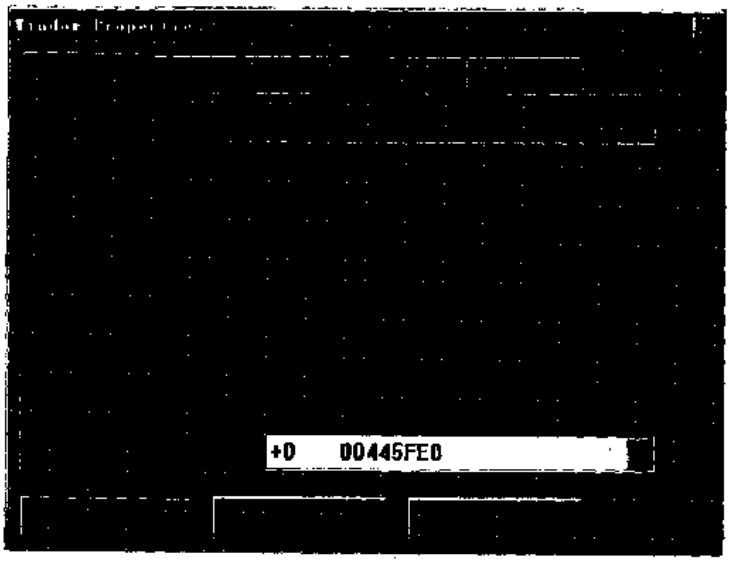



图 4-7　Window Properties 对话框

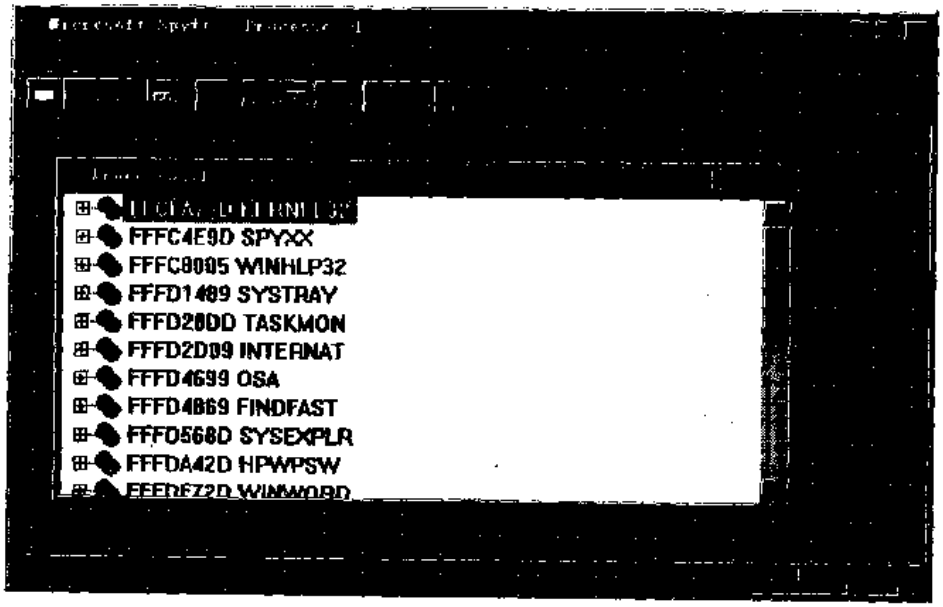



图 4-8　打开了进程视图的 Spy + + 主窗口

**注释：**

在图 4-8 中，我们关闭了窗口视图，实际上 Spy + + 的各种视图可以同时打开。

3. 线程视图

线程视图用于显示与窗口相联系的所有线程，利用线程视图可以查找一个特殊的线程。选择 Spy 菜单的 Threads 命令，或工具条中的线程工具或按 Ctrl + T 快捷键均可打开线程视图，如图 4-9 所示。

**注释：**

在图 4-9 所示的 Spy + + 的主窗口中既有窗口视图，又有进程视图和线程视图。

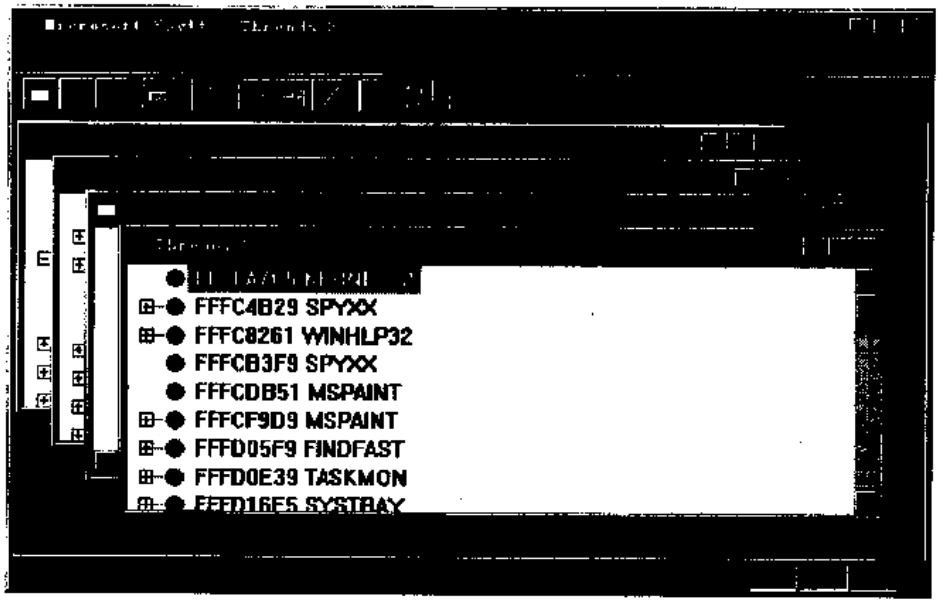



图 4-9　打开线程视图的 Spy + + 主窗口

4.消息视图

Windows 环境下应用程序的每一窗口都有与之相联系的消息流，消息视图可以用来查看消息流。我们可以为一个线程或进程建立一个消息视图，利用该视图可以查看发往指定线程或进程的所有窗口消息。

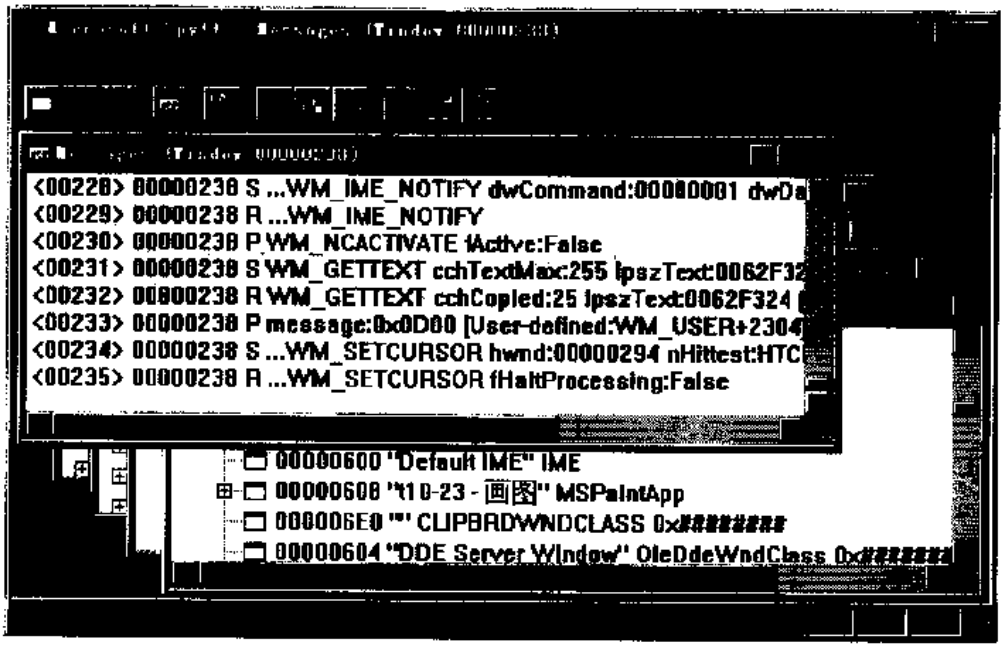



图 4-10　打开消息视图的 Spy + + 的主窗口

要打开消息视图，首先分别在窗口视图、进程视图或线程视图中选择要查看的窗口、进程或线程，然后选择 Spy 菜单的 Message 命令，弹出 Message Options 对话框，如图 4-6 所示。

接着，在 Message Options 对话框中选择所需要的消息选项，最后单击 OK 按钮开始登录消息。图 4-10 显示了打开消息视图的 Spy + + 的主窗口。

在消息视图中信息显示分为两列，第一列为窗口句柄，第二列为消息代码，代码后的消息参数和返回值显示在表的最右边。

在图 4-10 所示的消息视图中，每一个消息行上面均包含一个 P、S 或 R 的代码。这些代码的含义如下：

• P：这条消息由 PostMessage 函数发送到消息队列中。没有关于这条消息被最终处理的信息。

• S：这条消息是使用 SendMessage 函数发送的。这意味发送者要等到接收者处理并返回这条消息之后才重新取得控制权，因此，接收者可能有返回值返回给发送者。

• R：每个 S 行都有一个相近的 R(返回)行与之对应，它列出消息返回值。有时消息调用是嵌套的，这意味着一个消息句柄又发送了另一条消息。

## 4.3　Help Workshop 工具

Help Workshop 即为帮助生成器，它可用来帮助用户开发帮助文件(.hlp)，编辑工程文件和主题文件，并测试和报告帮助文件。帮助生成器在工程文件中组织主题文件(.rtf)、位图以及其他信息源，并最终生成一个可以利用 Windows 帮助系统来查看的帮助文件。

### 4.3.1　帮助生成器的构成

整个帮助生成器由以下几个文件和文档组成：

• Help Workshop：即帮助生成器本身，它包括 Hcw.exe 和 Hcrtf.exe，利用它可以编辑帮助工程文件和主题文件，编译、测试帮助文件并建立关于帮助文件的报告。

• 帮助系统开发者指南：其对应的文件是 Hcw.hlp，该文件描述了如何利用帮助生成器开发、编译并生成帮助文件。

• 2.0 版热点编辑器：其可执行文件是 Shed.exe，利用它可以创建具有多热点的图形。

• 1.1 版多分辨率位图编译器：利用它可以建立具有不同分辨率的位图，并且可以将这些图形组合到一个图形中，以适应不同的显示设备。

以上所介绍的帮助生成器的各组件均可在启动帮助生成器后，通过其集成的开发环境来调用。启动帮助生成器后其主窗口显示如图 4-11 所示。

### 4.3.2　帮助工程文件结构

帮助工程文件(Help Project File)是一种简单的 ASCII 文本文件，它以 .HPJ 为扩展名。工程文件用于把关于当前帮助工程的信息传递给 Help 编译器。这些信息包括 RTF 格式的主题文件名字和相关位图的目录位置、错误描述、警告级别、敏感的帮助信息以及帮助系统窗口的位置、大小和颜色等。每个帮助工程文件由一个或多个段组成，每一段的段名由一对括号([])括起来，并由它说明该段的用途。在帮助工程文件中可使用的段及其含义如下所述：

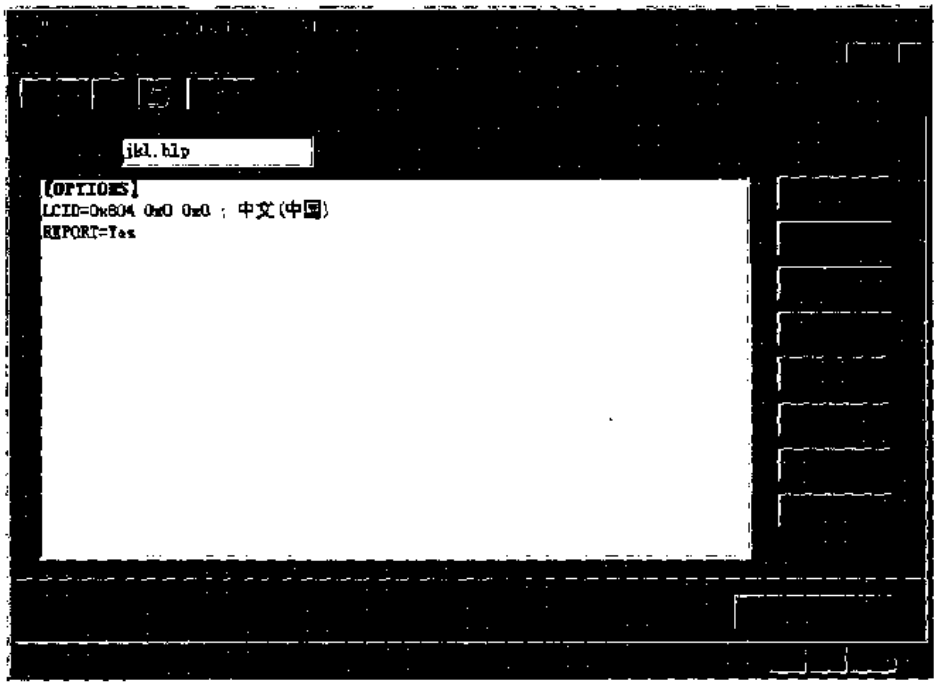



图 4-11　帮助生成器主窗口

[ALIAS]:给同一个主题赋予一个或多个主题标识符。

[BAGGAGE]:列出在帮助文件中放置的其他文件。

[BITMAPS]:指定所开发的帮助系统中要用到的位图文件。如果用户在帮助工程文件中通过使用 BMROOT 或 ROOT 选项列出了位图文件所在的路径,则可以在帮助工程文件中省去此段内容。

[BUILDTAGS]:指定有效的工程标识。此选项是为了保持与旧版本的兼容而保留下来的。

[CONFIG]:指定用户定义的非标准菜单和按钮命令,这些命令经常用来引用帮助文件和注册的动态链接库中的函数的宏命令。

[CONFIG:name]:与[CONFIG]段功能相同,但它用于独立的子窗口的配置(包括窗口名和窗口标题)

[EXCLUDE]:指定不包括在帮助文件中的特定主题。

[FILES]:指定用来创建帮助的主题文件,该段是必须的。

[FONTS]:指定主题文件的字体,该字体将代替主题文件的字体、大小和所基于的字符集。

[INCLUDE]:指定将包含到帮助文件中的主题文件。

[MACROS]:指定当用户选择此段所列关键字的索引时要运行的宏。

[MAP]:将主题标识符和数字联系起来。

[OPTIONS]:指定在创建帮助工程文件的过程中所要用到的控制和选项,该段位于整个帮助文件开始。

[WINDOWS]:定义帮助系统中主窗口或次级窗口的尺寸、大小和颜色等类型特征。

**注意:**

对帮助工程文件的各段都应通过帮助生成器自身来修改,一般应避免直接对帮助工程文件加以修改。

### 4.3.3　创建帮助主题文件

Windows 帮助系统是由一系列主题构成的,每个主题都可以被看作一个信息单元,可用它来描述一个特定的内容,例如,一个索引定义、一系列跳转、一幅位图以及一个概念描述等。在一个主题文件中可以包含一个或多个主题。当然,这些主题可以位于不同的文件中。

Windows 帮助系统每次只能显示一个主题,但可以使用热点连接(弹出式会话和跳转式会话)以及搜索主题关键字来显示其他主题的内容。帮助主题文件是标准的 ASCII 文件,可以用任何的文本编辑器来创建,也可以使用产生 RTF 格式文件的字处理软件间接地创建。在帮助主题文件中可以利用各种控制语句和帮助宏语句来完成对组成帮助文件的字体、图形、热点以及搜索关键字和次级窗口等元素的设置和控制。

1. 主题标识符和上下文标识数

Windows 帮助系统使用两种标识符:主题标识符和上下文标识数。下面分别加以介绍:

(1) 主题标识符(Topic ID)

Windows 帮助系统使用主题标识符来识别主题,主题标识符是由字母和数字组成的字符串,最长不能超过 255 个字符。为了识别主题,必须给它设置一个主题标识符,该字符串应放在主题页的"#"脚注文本中,其有效字符是 A ~ Z(大小写均可),0 ~ 9,句点(.),下划线(_)。

**例 1**:在"#"脚注中使用的主题标识符:

脚注定义部分:

```
# $ My First Topic
```

脚注部分:

```
# one
$ Topic1
```

**例 2**:热点定义部分中用来说明跳转的主题标识符:

```
Topic1
```

(2) 上下文标识数(Context ID Numbers)

上下文标识数是 Windows 帮助系统用来识别主题的数字。这些数字要在帮助工程文件的[MAP]段中予以说明,或者在[MAP]段中包含说明上下文标识数的头文件。

**例 1**:在帮助工程文件的[MAP]段中说明上下文标识数:

```
[MAP]
one 1000
```

```
two 2000
```

**例 2**:在头文件中说明上下文标识数:

```
# define one 1000
# define two 2000
```

**例 3**:在热点定义中用来说明跳转宏 Jumpmacro()所使用的上下文标识数:

```
Topic1 ! Jumpmacro("Demo.hlp",2000)
```

2. *帮助主题文件的创建*

帮助主题文件由一页或多页组成,每页称为一个主题(或主题页),每一个主题包括主题标题、主题正文、脚注、热点和图形等,不同的主题之间可以进行跳转或在主题内生成弹出式窗口,以便得到某一关键词的更为详细的信息。

(1) 帮助主题文件标题的创建

主题页的第一行是一个标题。尽管 Windows 帮助系统并不需要主题一定具有标题,但通过这些标题可以识别不同的主题(或章节),并且可以描述相应的内容。另外,虽然 WindHelp 给用户提供了搜索主题的不同方法,但最好还是利用标题来识别一个主题。为了突出标题,可以改变标题字体的大小,加黑字体或使用彩色字体。

(2) 帮助主题主题页(正文)的书写

紧跟着标题的就是主题正文。每一个主题应在不同的页上(利用强制分页)。段落的行长无关紧要,因为在编译时将根据窗口的右边界自动回行。Windows 帮助系统将源文件的每一页作为.HLP 文件的一个独立主题。

在主题文件中,跳转短语(超文本跳转)是以双下划线为标志的,后边跟有一个可隐藏文本格式的标记(或表示跳转目标)。在帮助窗口中(用户使用帮助所见到的窗口),超文本跳转短语以下划线形式出现,并显示为绿色。例如,在主题文件中双下划线短语How to Make Help File 在帮助窗口将显示为绿色的How to Make Help File,将鼠标放到它的下面,鼠标指针会变成一个小手形状。

下面为在主题文件中建立跳转单词或短语的步骤:

第一步:将光标移动到欲输入跳转短语的地方;

第二步:选择编辑器的双下划线格式;

第三步:以双下划线格式输入跳转短语;

第四步:取消双下划线格式,选择编辑器的隐藏文本格式;

第五步:输入与跳转目标相连接的主题标识符(Ontext String)或跳转标记。

帮助内容设计完成后,打开新的一页,将每个跳转页都写在单独的一页上。可通过在跳转标记上添加脚注把跳转页文本和跳转短语连接起来。

# 第 5 章

# MFC 基础类库

MFC 是 Microsoft Foundation Class 的缩写，即 Microsoft 基础类库，它是 Visual C++ 软件的一部分。MFC 库中的所有类形成了建立应用程序的框架，利用这些类可以充分支持 Windows 应用程序开发。构建 MFC 库的主要目的是为方便 Windows 应用程序的开发而设计。

MFC 封装了 Windows 应用程序的接口(API)函数、数据结构和宏，并以对象的形式提供给程序员。它充分利用了面向对象技术的数据抽象特性，利用它将大大简化 Windows 编程过程，提高编程效率。

## 5.1 MFC 的特点

提供 Windows 类库的厂商虽然不止 Microsoft 公司一家，但 Microsoft 公司所提供的 MFC 库相对来说具有以下一些优势：

• MFC 完整地封装了 Windows API 函数。MFC 库为经常使用的 Windows API 函数提供支持，包括窗口函数、消息、控件、菜单、对话框、GDI(图形设备接口)对象(如字体、笔刷、画笔及位图)、对象链接，以及多文档界面(MDI)。此外 MFC 类库也提供了具有共性的应用程序的操作，如打印、打印预览、工具条、状态条、数据库支持和 OLE 支持等。

• MFC 库提供了自动消息处理功能。MFC 库消除了最易产生错误的来源，即 Windows API 的消息循环。MFC 库将自动处理每一个 Windows 消息，以代替使用标准 switch case 语句，且每一条 Windows 消息被直接映射到一个进行处理的成员函数。

• MFC 库可以和谐地同以 C 语言为基础的 Windows 应用程序共存。MFC 库最重要的特征之一就是具有同以 C 语言为基础的使用 Windows API 开发的 Windows 应用程序共存的能力。在同一程序中，程序员可以同时使用 MFC 的类和 Windows API 调用。

• MFC 易学易用，因为它使用与 Windows API 相同的命名约定。这种命名约定使得程序员从名字就可以知道类的功能。

## 5.2 MFC 的类层次

MFC 的类构成了一个完整的体系，该体系由一个被称为 CObject 的类作为基类，其他类大部分由该类派生而来。这样整个体系包括若干个用于管理一个 Windows 应用程序的不同方面的子层次，这些子层次包括窗口(Windows)类、图形设备接口(GDI)类、对象链接和嵌入(OLE)类、文件和数据库类、对象输入输出类、异常类、集合类和混杂的支持类。各类之间的关系见图 5-1。

### 5.2.1 CObject 类

MFC 库中的大多数类是从 CObject 类派生出来的。CObject 类是 MFC 库的抽象基类，即是 MFC 库的根类。CObject 派生出的类所具有的优点包括：支持串行化，运行时类信息的可访问性，以及可打印出用于调试的诊断信息。CObject 类的声明在头文件 afx.h 中，为了进一步熟悉 CObject 类，程序清单 5-1 列出了头文件 afx.h 中与之相关的代码。

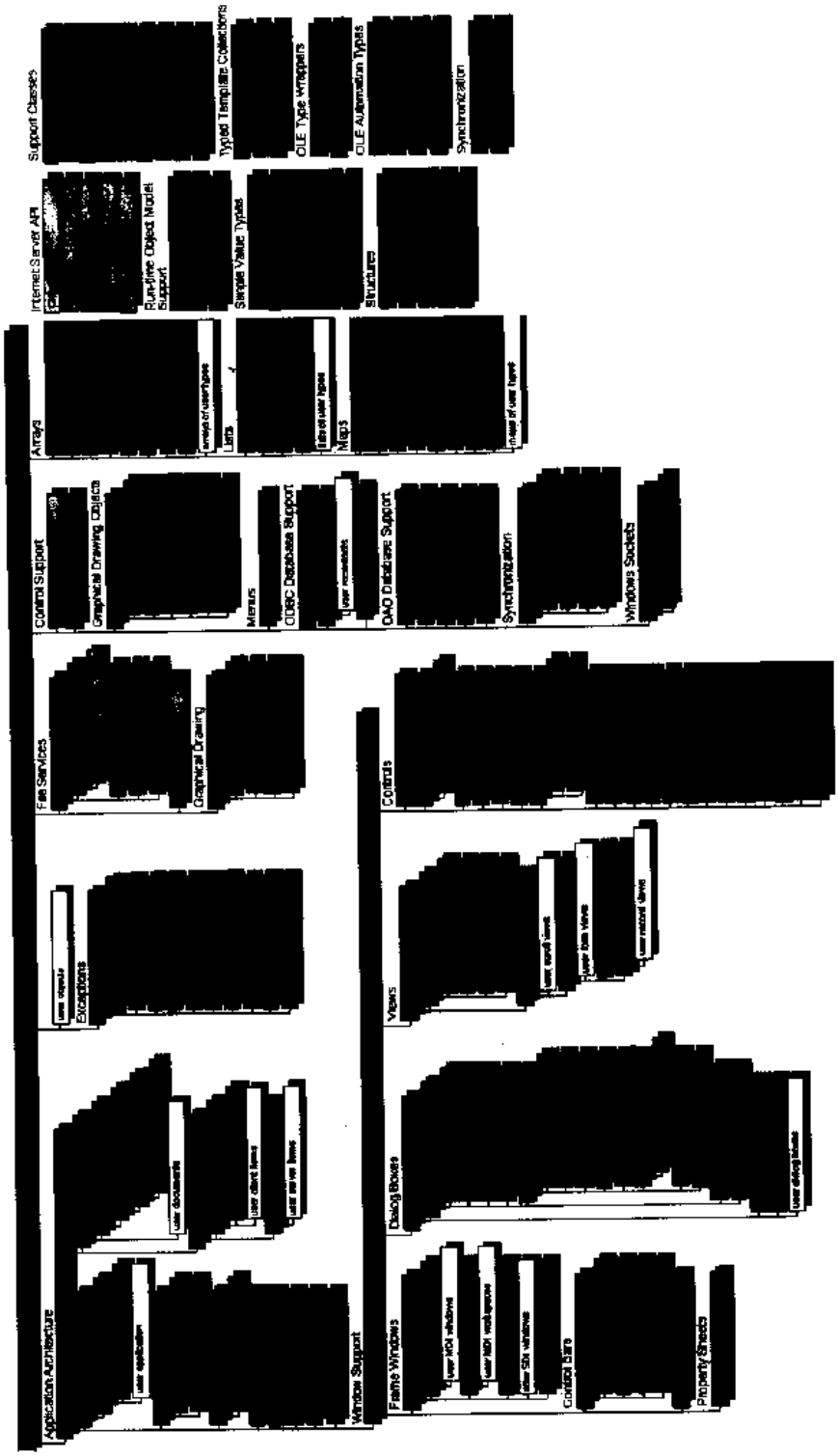

Support Classes
Typed Template Collections
OLE Type Wrappers
OLE Automation Types
Synchronization
Internet Server API
Runtime Object Model Support
Simple Value Types
Structures
Arrays
Lists
Maps
Control Support
Graphical Drawing Objects
Menus
ODBC Database Support
DAO Database Support
Synchronization
Windows Sockets
File Services
Graphical Drawing
Commands
Exceptions
Views
Application Architecture
Dialog Boxes
Window Support
Frame Windows
Control Bars
Property Sheets

图 5-1　MFC 的类层次

**程序清单 5-1:CObject 类的源代码**

```
class CObject{public: // Object model (types, destruction, allocation) virtual
CRuntimeClass* GetRuntimeClass() const; virtual ~CObject(); // virtual destruc-
tors are necessary
    // Diagnostic allocations
    void* PASCAL operator new(size_t nSize);
    void* PASCAL operator new(size_t, void* p);
    void PASCAL operator delete(void* p);
  #if _MSC_VER .= 1200
    void PASCAL operator delete(void* p, void* pPlace);
#endif
#if defined(_DEBUG) && ! defined(_AFX_NO_DEBUG_CRT)
    // for file name/line number tracking using DEBUG_NEW
    void* PASCAL operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
#if _MSC_VER = 1200
    void PASCAL operator delete(void *p, LPCSTR lpszFileName, int nLine);
#endif
#endif
    // Disable the copy constructor and assignment by default so you will get
    // compiler errors instead of unexpected behaviour if you pass objects  // by
value or assign objects.
protected:
    CObject();
private:
    CObject(const CObject& objectSrc);  // no implementation
    void operator=(const CObject& objectSrc);  // no implementation  // At-
tributes
public:
    BOOL IsSerializable() const;
    BOOL IsKindOf(const CRuntimeClass* pClass) const;  // Overridables
    virtual void Serialize(CArchive& ar);
#if defined(_DEBUG) || defined(_AFXDLL)  // Diagnostic Support
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif  // Implementation
public:
    static const AFX_DATA CRuntimeClass classCObject;
#ifdef _AFXDLL
    static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};
```

## 5.2.2 CMenu 类

CMenu 类直接继承于 MFC 类库的根类 CObject,如图 5-2 所示。



图 5-2 CMenu 类在 MFC 类层次中的位置

CMenu 类所提供的成员函数用于菜单的管理，包括创建、更新、跟踪和消除与一个窗口相关联的菜单资源。

对于比较简单的应用程序，如程序的窗口只有一套菜单，一般无须明确调用该类。只有当应用程序需要有几套菜单，并随着当前所处环境的不同菜单也各不相同时，方需调用该类来维护菜单。

使用该类的一般方法是：首先利用菜单编辑器建立菜单资源（菜单编辑器的使用请参阅第 2 章的相关内容），然后调用 CMenu 类的成员函数来维护菜单，最后调用 CWnd::SetMenu 将菜单显示于窗口上，从而使当前窗口菜单为新设置的菜单。新菜单将随着所依附的窗口的销毁而销毁。具体步骤如下所述：

（1）利用菜单编辑器建立菜单资源。

（2）调用 CMenu 类的成员函数 LoadMenu 或 LoadMenuIndirect 将一个菜单装入，这两个成员函数返回非零。函数原型如下：

```
BOOL LoadMenu( LPCTSTR lpszResourceName );
BOOL LoadMenu( UINT nIDResource );
BOOL LoadMenuIndirect( const void* lpMenuTemplate );
```

（3）调用 CWnd 类的成员函数 SetMenu 使得菜单依附于特定窗口，SetMenu 的函数原型如下：

```
BOOL SetMenu( CMenu* pMenu );
```

该函数将窗口类的当前菜单设置为指定的菜单，但它并不销毁以前的菜单，如果需要销毁以前的菜单，则请调用 CMenu 类的成员函数 DestroyMenu。

（4）窗口关闭前应调用 CMenu 类的成员函数 Detach 将菜单与窗口分离，随着窗口的关闭，其所依附的菜单也将被销毁。

### 5.2.3　文档结构子层次

文档结构子层次用来构建应用程序的框架，它的根是 CCmdTarget 类。该子层次的所有类及其继承关系如图 5-3 所示。在这些类中提供了大多数应用程序通用的函数。一般地，应该从这些类中派生新类，在新类中重载已存在的成员函数，从而使应用程序符合实际需要。Visual C++ 向导程序可产生几种应用程序，这些应用程序以不同的方式利用了应用程序框架：单文档界面（SDI）和多文档界面（MDI）程序利用了其中大部分特性，而基于对话框和窗体的应用程序以及动态链接库则只利用其中的部分特性。创建基于文档的应用程序一般需要经过以下几步：

第一步：为每一种类型的文档定义一个派生自 CDocument 类的派生类。

第二步：添加用于存储文档数据的成员变量。

第三步：编写用于实现读取和修改文档数据的成员函数。文档视图充当了文档和文档数据之间的中介。一方面将文档数据以便于阅读的方式显示在文档视图中，另一方面，用户对视图的修改也通过文档视图保存在文档中。

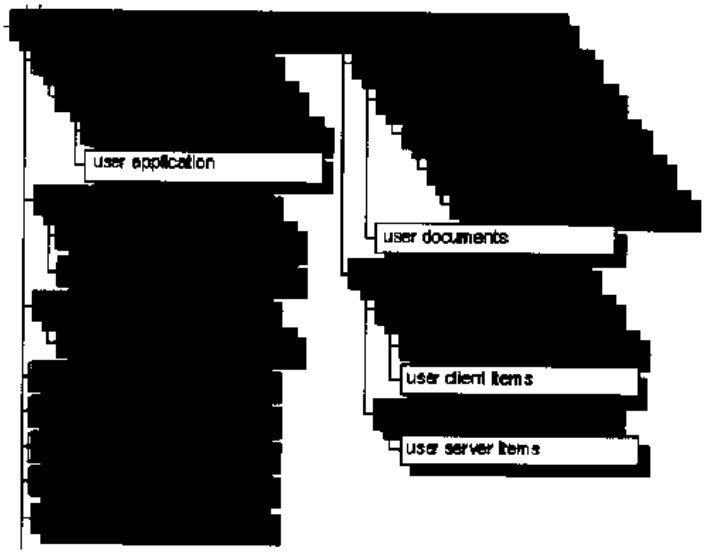



图 5-3　文档结构子层次的所有类及其派生类

### 5.2.4　边框窗口子层次

边框窗口子层次的根是 CFrameWnd 类，而 CFrameWnd 类继承自 CCmdTarget 类的子类 CWnd。这些类用来模型化基本的单文档界面(SDI)窗口和比较复杂的多文档界面(MDI)窗口。图 5-4 表示了此子层次在 MFC 类体系中的相对位置。

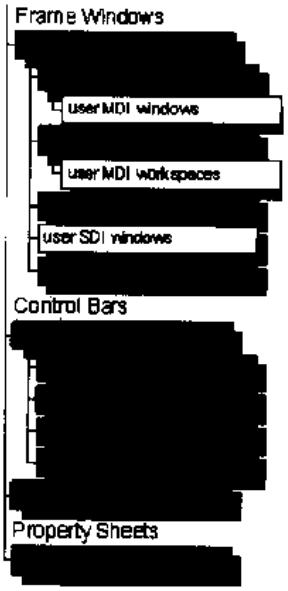



图 5-4　边框窗口子层次的所有类及其派生类

在开发基于文档的应用程序中，框架窗口即通过该类来实现，其应用技巧参见第 6 章和第 7 章的相关内容。

### 5.2.5　控件栏子层次

控件栏子层次有着与边框窗口子层次相同的父类，它用来模拟工具栏、状态栏、对话栏和分隔窗，如图 5-5 所示。

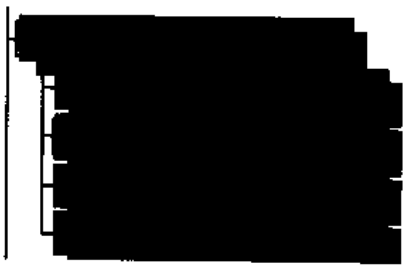


图 5-5　控件栏子层次及其派生类

### 5.2.6　视图子层次

视图子层次以 CView 为其根类，而 CView 类也是继承自 CWnd。CView 是可以在其他窗口中创建的子窗口，它可以提供一个特殊的接收外来输入的结构窗口。CView 类的层次结构如图 5-6 所示。

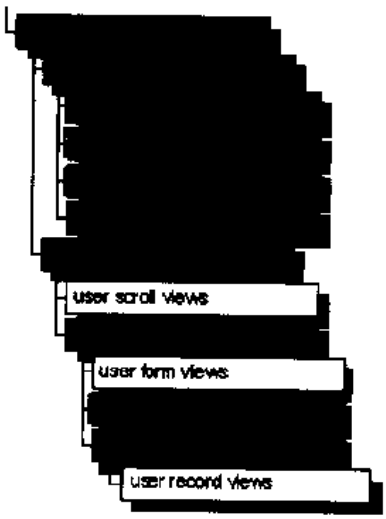



图 5-6　视图子层次的根类 CView 及其派生类

CView 类广泛用于基于文档的应用程序中，利用该类的派生类可图形化地管理文档数据，并将用户对文档的操作通过它来实现。CView 类可用来实现用户自定义视图类的基本功能。在程序运行时视图类用于视图的实现。一个视图只能分配给一个文档，但是一个文档可以拥有多个视图。

### 5.2.7　对话框子层次

对话框子层次包含了通用对话框类 CDialog 以及支持类，包括选择色彩、选择字体、打

印、查找/替换字符串等通用对话框。CDialog 类也是继承自 CWnd 类，图 5-7 显示了该层次的所有类。

在基于对话框的应用程序中和文档应用程序中设计对话框时，都应从该层次的根类或其任一子类中派生一个用户自定义类以实现特定功能。

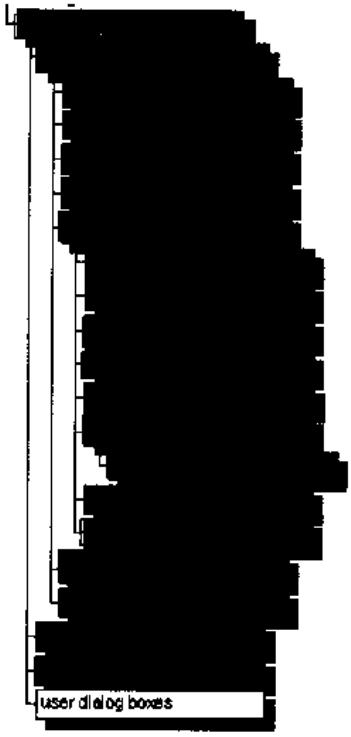



图 5-7 对话框子层次所包括的类



图 5-8 控件子层次及其派生类

### 5.2.8 控件子层次

控件子层次继承自根类 CObject 的子类 CWnd，该子层次包括常见的 Windows 控件，这些控件包括静态文本、命令按钮、位图按钮、列表框、组合框、编辑控件和 VB 控件的类。控件子层次的结构如图 5-8 所示。

各种应用程序中所涉及的几乎所有控件都直接或间接地继承自此层次中的某一个类。

### 5.2.9 绘图对象子层次

在 Windows 中所有图形化的输出都是绘制在被称为设备描述符(Device Context，简称 DC)的区域内。绘图对象子层次中所包含的类即用于模型化绘图对象，如画笔、刷子、字体、位图和调色板等，它实现了 Windows 的 GDI 绘图原语。该层次的根类是继承自总根类 CObject 的 CGDIObject 类，其层次结构见图 5-9 所示。

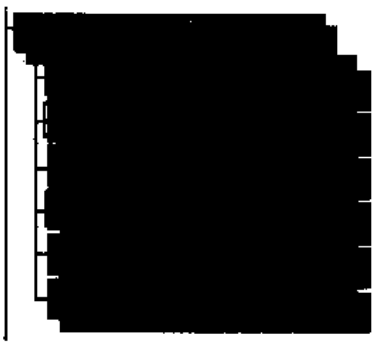


图 5-9　绘图对象子层次及其派生类

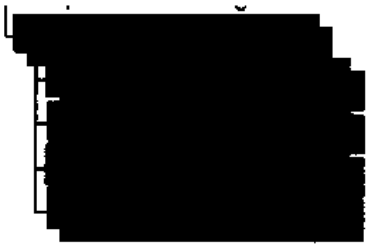


图 5-10　CDC 类及其派生类

### 5.2.10　CDC 子层次

CDC 类是 CObject 的子类，也是 MFC 库中的 CDC 子层次的根类。CDC 类包含支持 CDC 子孙类提供的功能的成员函数，如图 5-10 所示。该子层次定义的类主要用于在设备上绘制文本或图形，若该类与绘图对象子层次配合使用，则可以描绘十分丰富的格式化文本和各种各样的图形。

### 5.2.11　其他子层次

除了上述类及子层次之外，MFC 类库中还包括如下几个子层次：

(1) 异常子层次：该子层次提供的类用来处理各种类型的异常错误，如图 5-11 所示。这些异常包括内存与文件操作异常、数据库操作异常、资源异常、对象链接与嵌入异常等。

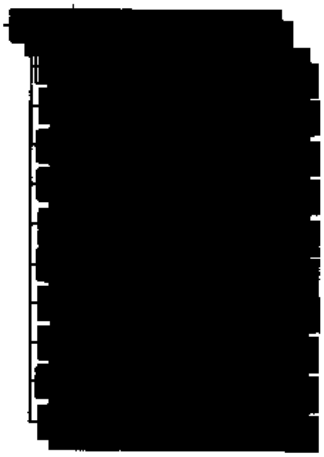


图 5-11　异常子层次



图 5-12　文件服务子层次

(2) 文件服务子层次：该层次提供了各种用于处理文件的类，如图 5-12 所示。利用这些类可以操纵各种文件，它们对文件所能实施的操作远远超过文档自动化对于文件的操纵。

(3) 集合子层次：该子层次包含了在程序设计中用得非常普遍的数据结构，它包含数组、链接列表和映射表等集合类。图 5-13 分类显示了集合类子层次的构成。利用这些类可

以方便地管理动态列表和对象数组类。

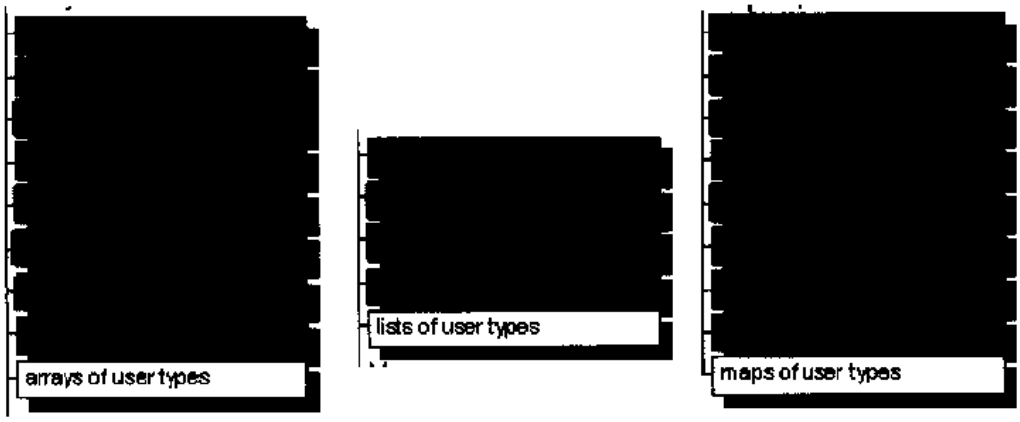



图 5-13　构成集合类的各种类之间的相互关系

(4) 数据库支持类子层次:该子层次支持访问数据库中的数据源,它包括 ODBC 数据库支持和 DA0 数据库支持。这两类的根类都继承自 MFC 类库的总根类 CObject。图 5-14 把这两部分并列显示,表明它们继承自同一个父类而且功能相近。




图 5-14　数据库支持类

(5) OLE 2.0 支持子层次:该子层次提供的类支持对象的链接与嵌入。该子层次所包括的类既有继承自总根类 CObject 的子类 CCmdTarget 的派生类(如 COleDataSource、COleDrop-Source 和 COleDropTarget 等),又有不继承自任何类的类(如 COleDataObject 等),还有其他的继承关系。



图 5-15　单值类型子层次

(6) 单值类型子层次:该子层次通常用来模型化使用的对象,如字符串、矩形等,如图 5-15所示。

(7) 结构子层次:该子层次用来模型化涉及文件状态、打印内存状态和窗口创建等信息,该子层次还包括了分析命令行参数的类。如图 5-16 所示。



图 5-16　结构子层次

(8) 运行时对象模型化支持子层次:该子层次用于对象的动态访问和文档的串行化等。

(9)Internet 服务支持子层次:该子层次提供了编写 Internet 服务器程序所需的基本类,从这些类派生出的新类就可以根据需要开发 Internet 服务器程序。该层次所包括的类如图 5-17 所示。



图 5-17　Internet 服务支持子层次

## 5.3　MFC 的通用集合类

MFC 提供的通用集合类可与 MS _ DOS, Windows 兼容使用,也可与以前编好的程序或 Quick _ Win 一起使用。集合类支持各种不同的数据结构,并提供了对应用程序中使用信息的跟踪。

MFC 类库中在 MS _ DOS, Windows 下使用的通用集合类有:

CString、CTime、CTimeSpan、CFile、CStdioFile、CMenuFile、CFileStatus、CArchive、CException。

集合类提供了通用数据结构,可以支持动态数组、队列、链表、堆栈、树、排序数组和映射这些特殊的结构。MFC 提供了集合类的三个基本类型:数组、链表和映射。

- 数组:MFC 为字节数组、字数组、双字数组、CSrting CObject 指针和无值(Void)指针提供了相应的类。
- 链表:MFC 为 CString、CObject 指针和无值指针的链表提供了相应的类。
- 映射:MFC 提供了映射集合,即一个关键字映射到一个值。

### 5.3.1 CString 类

CString 类的成员函数实现对字符串的各种处理,它支持可变长度的字符串。该类所提供的函数和操作符的使用语法与基本的字符串连接和字符串比较操作符非常相似。CString 对象比二进制字符数组更易使用。MFC 提供的 CString 类支持字符串的动态操作。

一个串实现成一个 CStirng 类对象时,可以使用具有重载操作的简单的"+"操作符来处理它。在没有程序员干涉的情况下,CStirng 类对象也可以自动地增大或缩小来填充或节省空间。

CString 类中常见的成员函数如下:

- CString(const char * ):构造函数,初始化所建立的 CString 对象。
- CString(const CString &Z):拷贝构造函数。
- int GetLength(void):返回对象字符串长度。
- int Compare(const char * ):对象字符串比较,如果两字符串相同则返回 0,否则返回 1(CString 对象大于参数字符串)或 -1(CString 对象小于参数字符串)。
- CString Left(int nCount):返回对象字符串左边 ncount 个字符。
- CString Right(int nCount):返回对象字符串右边 ncount 个字符。
- Void Empty(void):使 String 对象为空,释放内存。
- BOOL IsEmpty(void):测试 CString 对象是否为空,如果 CString 对象为空,则返回 TRUE,否则返回 FALSE。
- Void MakeUpper(void):把 CString 对象转换为大写字符。
- Void MakeLower(void):把 CString 对象转换为小写字符。
- Void MakeReverse(void):反转 CString 对象中字符的顺序。

在 CString 中,可以使用下列三种运算符:

- +:将两个字符串相连接,返回新字符串。
- =:把一个新字符赋值给 CString 对象。
- +=:把一个新字符连接在 CString 对象字符串的后面。

### 5.3.2 CTime 类

MFC 库提供了两个与时间有关的类:CTime 和 CtiemSpen。CTime 类对象表示一个特定的时间和日期;CTimeSpan 类对象则表示一个时间间隔。

CTime 对象把一个时间和日期封装成一个 time_t 数据类型,它遵循 ANSI 的数据类型标准。CTime 类具有运行时的成员函数,它可以在 24 小时制时间与 a.m,p.m 时间之间进行转换,把日期在 Julian 形式与 Grigorian 形式之间进行转换。另一个成员函数可以把 time_t 值转换成表示年、月、日、时、分和秒的整数。CTime 类具有重载的插入(<<)和提取(>>)操作符,用于归档和诊断输出。

CTime 的常用成员函数如下:

- CTime(int nYear, int nMonth, int nDay, int nHour, int nMin, int nSec):初始化所建立的

CTime 对象。

- int GetYear(void):返回对象的年份。
- int GetMonth (void):返回对象的月份。
- int GetDay(void):返回对象的日期。
- int GetHour(void):返回对象的时针值。
- int GetMinute(void):返回对象的分针值。
- int GetSecond(void):返回对象的秒针值。
- CString Format( UINT nFormatID ):产生一个 CTime 对象相应的格式化字符串。Format ()成员函数内最常用的参数是"C%",表示将对象的日期及时间转换成字符串类型。
- static CTime PASCAL GetCurrentTime( void):返回 CTime 对象,该对象表示系统当前的日期和时间。

程序清单 5-2 举例说明了 CTime 的应用。

**程序清单 5-2:CTime 类应用举例**

```
//CTime 应用实例
//文件名:MyCTime.C

#include <iostream.h.
#include <afx.h.
Void Main()|
  CTime TheTime(1993,1,1,7,35,30);
    cout << "The Date and Timeis:" << TheTime.GetMonth()
      << "/" << TheTime.GetDay() << "/" << TheTime.GetYear()
      << " " << TheTime.GetHour() << ":" << TheTime.GetMunite()
      << ":" << TheTime.GetSecond() << endl;
    TheTime = CTime::GetCurrentTime();
    CString TheString = theTime.Format("%C");
    cout << "The Current Rate and Time is:" << TheString;
  }
```

## 5.3.3　CStringArrary 类

CStringArray 类是一个数组集合类。它的成员函数如下:

- CStringArray():类构造函数。
- int GetSize(void):返回数组中的元素数目。
- Void SetSize( int nNewSize, int nGrowBy = ):设置对象数组中元素的数目。
- void RemoveAll(void):删除对象数组的所有内容。
- int Add( CObject * newElement ):在数组末端增加内容。
- void InsertAt( int nStartIndex, CObArray * pNewArray):在数组对象中指定的下标值处

插入新内容。

- void RemoveAt( int nIndex, int nCount = 1 ):删除指定的下标处的一个元素。

程序清单 5-3 举例说明了 CStringArrary 的应用。

**程序清单 5-3:CStringArrary 类的应用**

```
#include<iostream.h.
#include"quikwin.h"

void main(){
  int i,Num;
  CStrig TheString1="Hello";
  CString TheString2="Visual C++";
  CStringArry TheStrArr;
  TheStrArr.Add(TheString1);
  TheStrArr.Add(TheString2);
  Num = TheStrARr.GetSize();
  cout <"The size of array is"<<Num<<endl;
  for (i=0,i<Num;i++)
    cout<TheStrArr.GetAt(i)<<endl;
    cout <<endl;
    TheStrArr.InertAt(1,"Wclcome to use");
    Num = The Str Arr.Get Size();
    cout<<"The size of array is "<<Num<<endl<<endl;
    for (i=0;i<Num,i++)
    cout<<TheStrArr.GetAt(i)<<endl;
    cout<<endl;
    TheStrArr.RemoveAll();
    Num=TheStrArr.Get Size();
    cout<<"The Size of array is"<<Num<<endl;
}
```

### 5.3.4 CStringList 类

CStringList 类是一个链表集合类。链表集合类对于处理无值指针、CObject 派生类对象和 CString 是十分方便的。

CStringList 类常用的成员函数如下:

- CStringList():构造函数。
- CStringList * GetHead(void):返回表头内容。
- CStringList * GetTail(void):返回表尾内容。
- CStringList * RemoveHead(void):删除表头。

- CStringList * RemoveTail(void)：删除表尾。
- POSITION AddHead(CStringvar)：将字符串放到表头。
- POSITION AddTail(CStringvar)：将字符串放到表尾。
- Void RemoveAll(void)：删除表中所有元素，并释放内存。
- POSItion GetHeadPosition(void)：返回表头的位置。
- Position GettailPosition(void)：返回表尾的位置。
- CStringList * GetNext(POSITION &pPosition)：返回下一个字符串的位置。
- CStringList * Getprev(POSITION &pPosion)：返回前一个字符串的位置。
- POSITION FindIndex( int nIndex )：返回查找字符串的位置。
- int GetCount(void)：返回表中字符串的个数。
- BOOL IsEmpty(void)：检查表中是否有元素，若空则返回 TRUE，否则返回 FALSE。

### 5.3.5　CMapStringToString 类

CMapStringToString 类是一个映射集合类。映射集合类是一个字典，它把与关键字相联系的对象映射到集合中，以后便可以使用关键字来确定条目在映射中的位置。

该类常用的成员函数如下：

- CMapStringToString()：构造函数。
- void SetAt( LPCTSTR key, CString * newValue )：先找到标识，然后将指定字符串放入映射存储装置内。
- BOOL Lookup( LPCTSTR key, CString * &rValue )：查找字符串，若找到，则返回非零值。
- Viod RemoveAll(Void)：删除映射存储设备内所有字符串对象。
- int GetCount(void)：返回映射存储设备内字符串对象的个数。
- BOOL IsEmpty(void)：检查映射存储设备是否为空，若空，则返回 TRUE，否则返回 FALSE。

程序清单 5-4 的例子示范了该类的应用。

**程序清单 5-4：CStringList 类的应用**

```
#include<iostream.h.
#include<quickwin.h.
Void FindByKeyw(CMapStringTostring &Map, CString KeyW){
  CString Book;
  if (map.Lookup(KeyW, Book)){
    cout<<"Find Key Word:"<<KeyW<<"in"<<endl;
    cout<<Book<<endl<<endl;
  }
  else
    cout<"Not find Key Wind:"<KeyW<<endl;
```

```
}//end of FindByKeyW

Void main(){
  CMapStringToString Map;
  CString Book;
  CString KeyW;
  Book = "programming With Visual C + +";
  KeyW = "Visual C + +";
  Map.SetAt (KeyW, Book);
  Book = "programming with Visual C + +6.0";
  KeyW = "BC";
  Map.SetAt(Key V, Book);
  Book = "programming with Visual C + +";
  KeyW = "TC";
  Map[KeyW] = Book;
  FindKeyW(map,"VC");
  FindKeyW(map,"TC");
  FindkeyW(map,"BC");
  FindKeyW(map,"VB");
  }
```

# 第 6 章

# Windows 应用程序开发基础

在 Visual C++ 6.0 中，开发 Windows 应用程序有以下几种方法：

（1）直接调用 Windows 环境提供的 Win32API 函数来编写 Windows 应用程序。实际上该方法适用于任何 Windows 环境下的 C/C++ 语言编程。

（2）使用 MFC 类库和活动模板库（ATL）直接编写 Windows 应用程序。

（3）既使用 MFC 类库和 ATL，也使用向导（Wizard）来编写 Windows 应用程序。在这种情况下用 MFC AppWizard、MFC ActiveX ControlWizard、ISAPI Extension Wizard 和 ATL COM AppWizard 等向导来生成 Windows 应用程序的基本源文件，然后用 Class Wizard 来建立应用程序的类和消息处理代码并定义控件的属性、事件和方法，最后把各应用程序所要求的功能添加到程序中。

本章介绍 Visual C++ 提供的应用程序开发向导工具。

MFC AppWizard 用于创建基于 MFC 类库的 Windows 应用程序框架文件。MFC AppWizard 在创建框架文件时充分利用了 MFC 的基本类库中的类，从而使这些文件包括了构造一个基于 Windows 的应用程序所必须的所有文件，如源文件、头文件、资源文件和模块定义文件等。

利用 MFC AppWizard 建立基于 MFC 类库的应用程序时，根据要建立的应用程序的不同，需要 6 步（或 4 步），每一步要设置不同的选项，这样就可得到开发应用程序的基本文件，然后再利用 Class Wizard 来补充应用程序尚未实现的具体功能。

## 6.1 生成 MDI 应用程序

生成一个具有 MDI 接口的应用程序，只需利用 MFC AppWizard 向导，逐步回答向导所提出的问题，即可生成一个初步的应用程序，然后再利用其他工具（如 Class Wizard）充实、完善应用程序。其具体步骤如下：

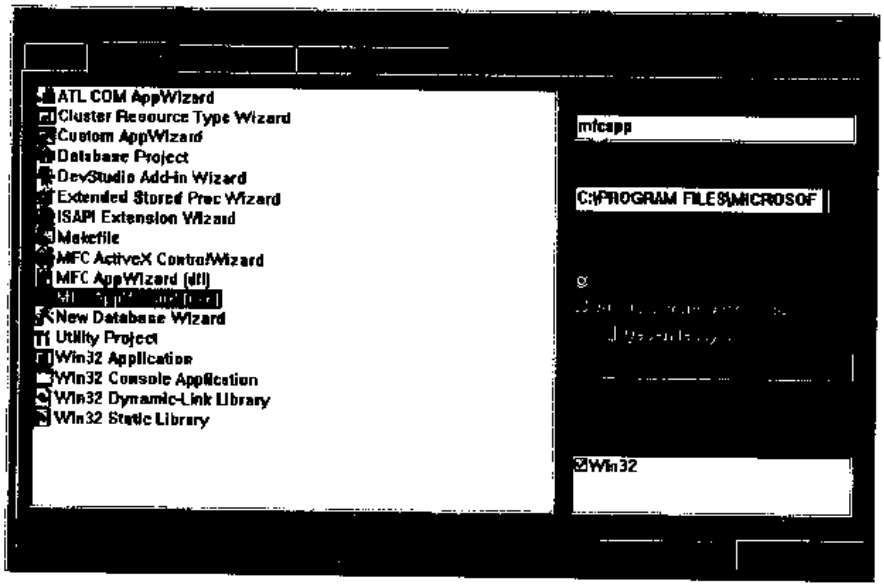



图 6-1 建立新项目对话框

(1) 在 Developer Studio 工作平台环境下,选择 File 菜单中的 New 命令,选中 Project 选项卡,如图 6-1 所示。

(2) 单击所需项目类型,即 MFC AppWizard,在 Project name 文本框中输入项目名称,再根据需要修改其他默认选项,单击 OK 按钮即可启动 MFC AppWizard,出现 MFC AppWizard Step 1 对话框,如图 6-2 所示。

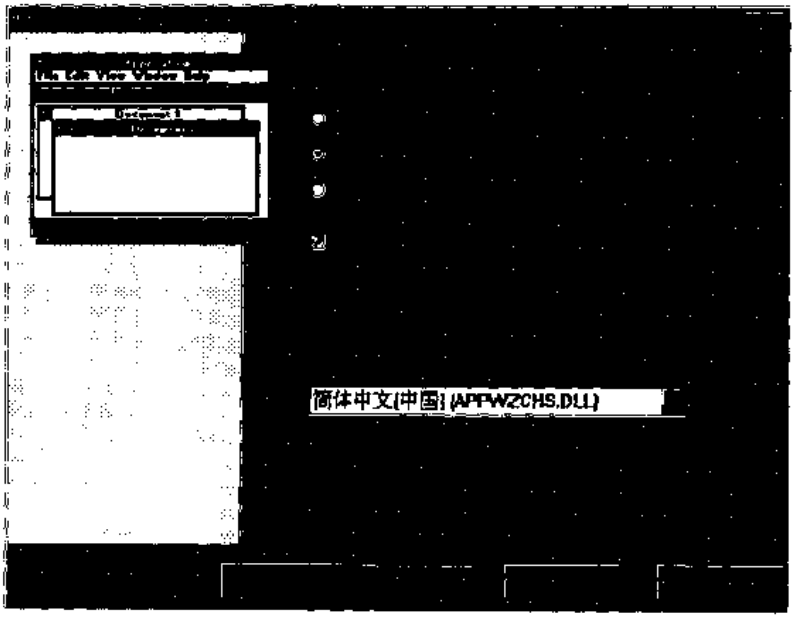



图 6-2　MFC AppWizard Step 1 对话框

(3) 在 MFC AppWizard Step 1 对话框中根据应用程序的需要,选择应用程序的类型,共有三种类型,即 Multiple documents(简称 MDI 即多文档)、Single document(简称 SDI 即单文档)

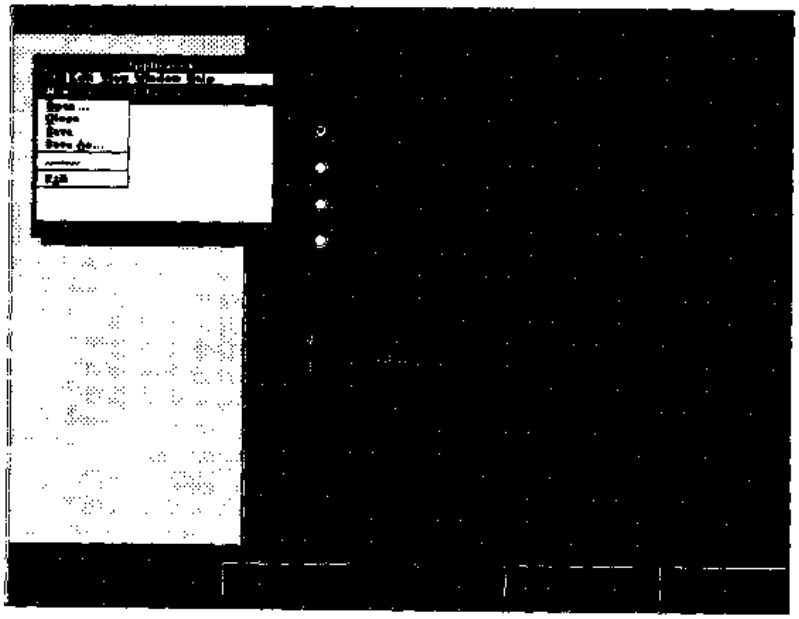


图 6-3　MFC AppWizard Step 2 of 6 对话框

和 Dialog based(即基于对话框),此时应选择 Multiple documents,然后根据需要选择资源文件的语言(缺省时是操作系统支持的语言,在本例中为简体中文)并确定是否选择 Document/View architecture support 选项(在这我们选中该复选框),如果不需要则清除该复选框,这样最终生成的应用程序长度将减少,但是也将不能利用 MFC 支持应用程序打开磁盘文件。最后单击 Next 按钮,进入下一步,出现 MFC AppWizard Step 2 of 6 对话框,如图 6-3 所示。

(4) 选择以下数据库支持选项之一:

• None: 选择该选项则向导所建立的应用程序不具备数据库支持,该选项也是缺省选项。

• Header files only: 选择该选项则应用程序具有基本的数据库支持功能,这样向导将建立头文件(AFXDB.H)和连接库,但是不建立任何特定的数据库类。用户可以建立记录集用来查看和更新记录。

• Database view without file support: 选择该选项则向导所建立的应用程序包含头文件、连接库、记录视图和记录集(该选项只有在上一步选中 Document/View architecture support 复选框时才有效)。该选项包括了文档支持,但不支持串行化。如果选中该项,还必须确定数据源,可通过单击 Data Source 按钮,由 Database Option 对话框进行设置。按照这些设置生成的程序将具有由 CRecordView 派生的类,此类作为 View 类,也可以与由向导建立的 CRecordset 派生类相关联。

• Database view with file support: 选择该选项可使应用程序包含数据库头文件、连接库、记录视图和记录集(只有在上一步选中 Document /View architecture support 复选框时才有效),并可使应用程序支持文档串行化操作。一般情况下数据库基于记录的操作远远多于基于文件的操作,因此不需要串行化支持。

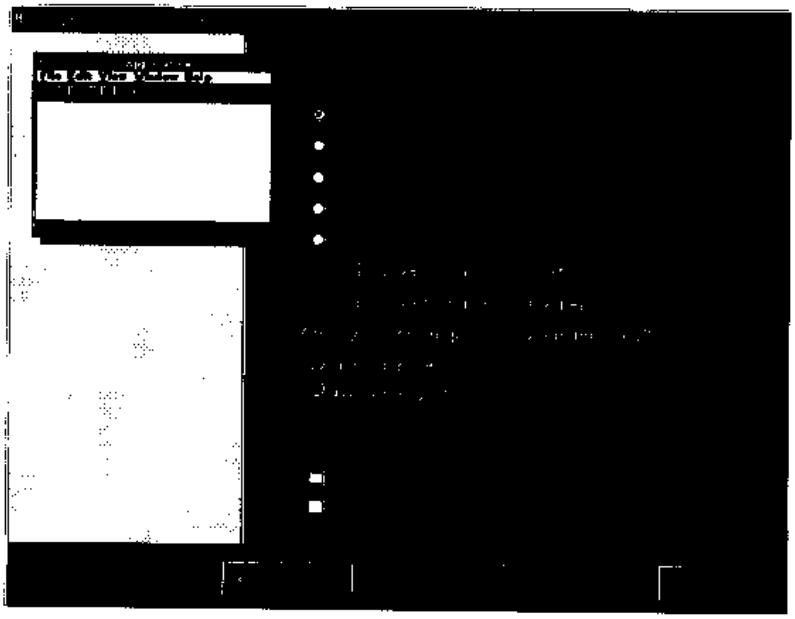


图 6-4 MFC AppWizard Step 3 of 6 对话框

在此保留缺省的 None 选项,单击 Next 按钮,进入下一步,出现 MFC AppWizard Step 3 of 6

对话框，如图 6-4 所示。

(5) 由于在 MFC AppWizard Step 1 对话框中选择了多文档应用程序，所以此处可以选择以下复合文档支持选项：

- None：缺省设置，应用程序不支持 ActiveX 控件。
- Container：应用程序可以包含被链接和嵌入的对象。
- Mini – Server：应用程序可以创建嵌入的对象。
- Full – Server：应用程序可以支持 ActiveX 控件，创建的 ActiveX 对象包含在复合文档中。
- Both container and server：应用程序既可作为容器，又可作为服务器。
- Yes, please：使用复合文档格式来串行化容器应用程序的文档。
- No, thank you：不使用复合文档格式来串行化容器应用程序的文档。
- Automation：使应用程序支持自动化。
- ActiveX Controls：使应用程序使用 ActiveX 控件，它也是缺省的设置。

在此保留 None 选项，并取消 ActiveX Controls 复选框，单击 Next 按钮，进入下一步，出现 MFC AppWizard Step 4 of 6 对话框，如图 6-5 所示。

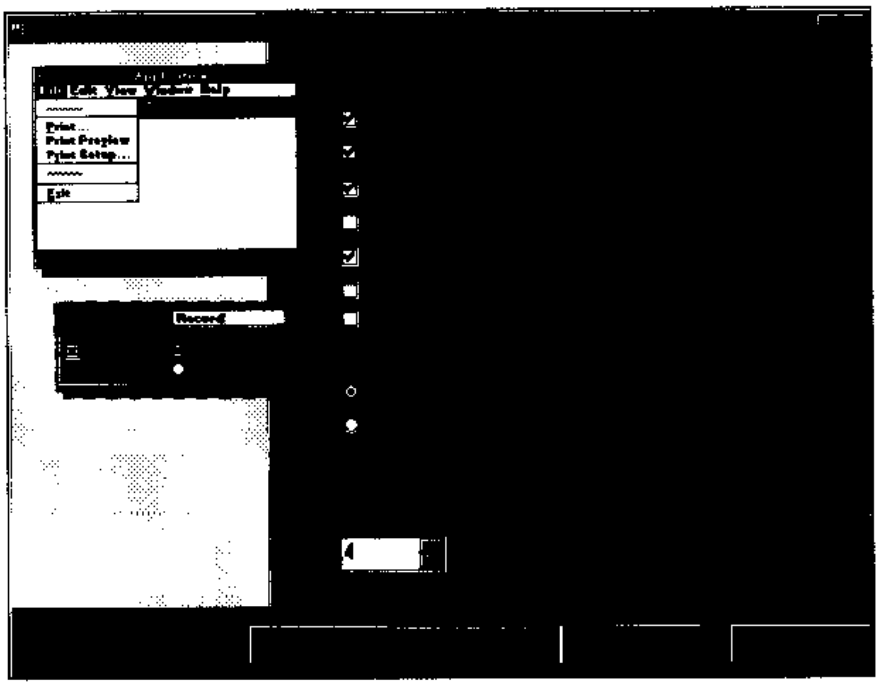


图 6-5　MFC AppWizard Step 4 of 6 对话框

(6) 在此对话框中可选择应用程序包含的界面元素：

- Docking toolbar：添加船坞化的工具栏到应用程序中。
- Initial statusbar：添加初始状态条到应用程序中。
- Printing and print preview：添加代码以使应用程序可以调用打印、打印设置及打印预览功能(View 类的成员函数)等命令

• Context – sensitive Help:添加上下文敏感帮助到应用程序中。所谓上下文敏感帮助是指单击鼠标右键所出现的 What is this(这是什么)帮助。

• 3D Controls:使应用程序的控件具有 3D 外观。

• MAPI(Messaging API):使应用程序可以创建、传输和存储邮件消息。

• Windows Sockets:使应用程序可以与 TCP/IP 网络通信。

• How many files would you like on your recent file list:使应用程序可以跟踪最近使用的文件,缺省时,应用程序 Files 菜单底部列出最近使用过的 4 个文件。

**注释:**

Visual C++ 6.0 增加了一个用来选择工具栏外观的选项,共有两种:即传统型(Normal)和 IE 型(Internet Explorer ReBars)。

• Advanced 按钮:可打开一对话框用来设置应用程序文档模板和框架属性,如图 6-6 所示。

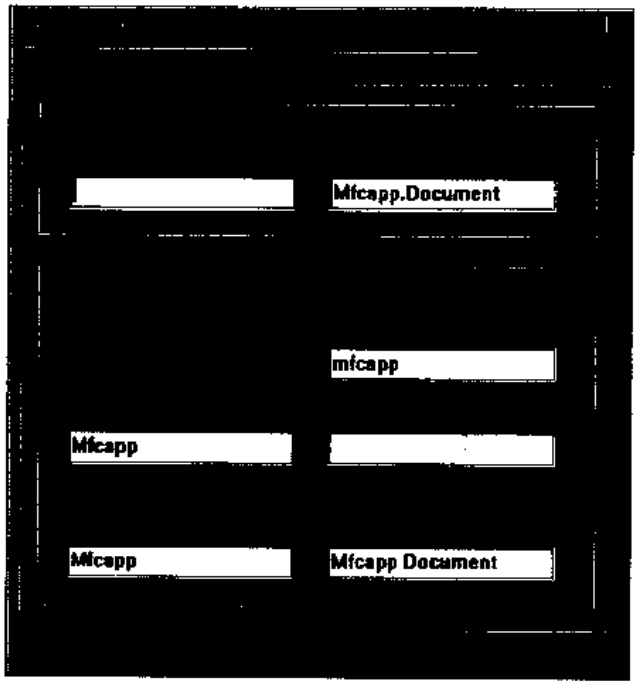



图 6-6 Advanced Options 对话框

为了简化程序,我们使用图 6-5 所示的默认选项,单击 Next 按钮,进入下一步,出现图 6-7所示的 MFC AppWizard Step 5 of 6 对话框。

(7) 在此可设置以下项目选项:

• 设置项目风格:可选择标准 MFC 风格或 Windows 资源管理器风格,缺省为标准 MFC 风格。

• 选择是否由向导在最终产生的源文件中添加注释:缺省时选中该项,向导自动向源文件中添加注释并产生 ReadMe.txt 文件。

• 选择使用 MFC 库的方式:使用方式有两种,即作为共享的动态链接库(可减少应用

程序长度)和作为静态库连接到应用程序中。

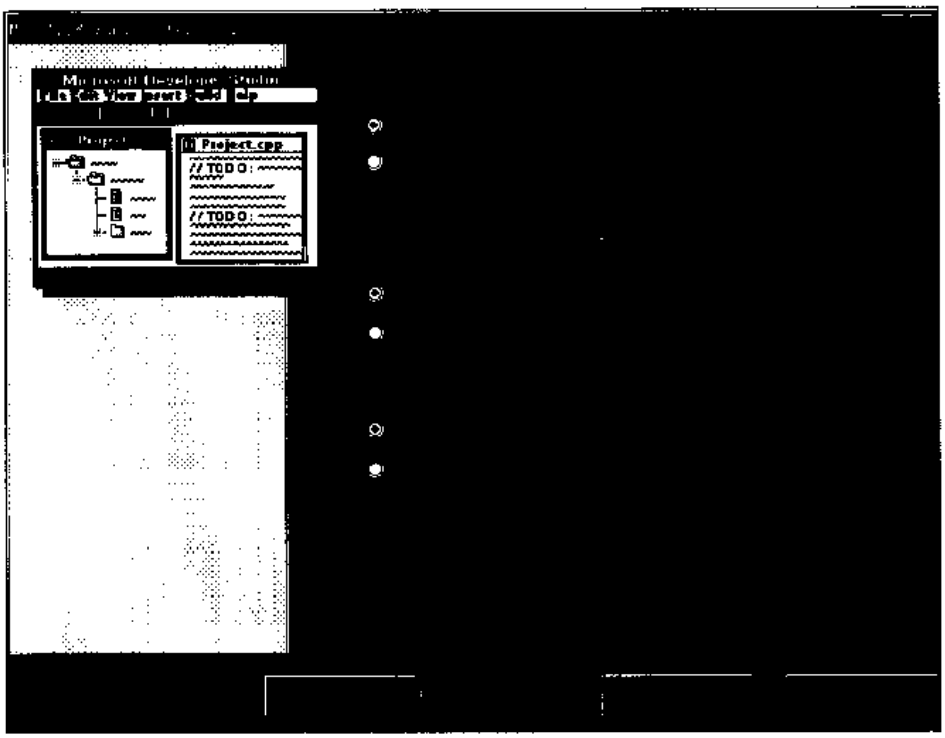


图 6-7　MFC AppWizard Step 5 of 6 对话框

做好如图 6-7 所示的选择后,单击 Next 按钮,进入下一步,出现如图 6-8 所示的 MFC AppWizard Step 6 of 6 对话框。

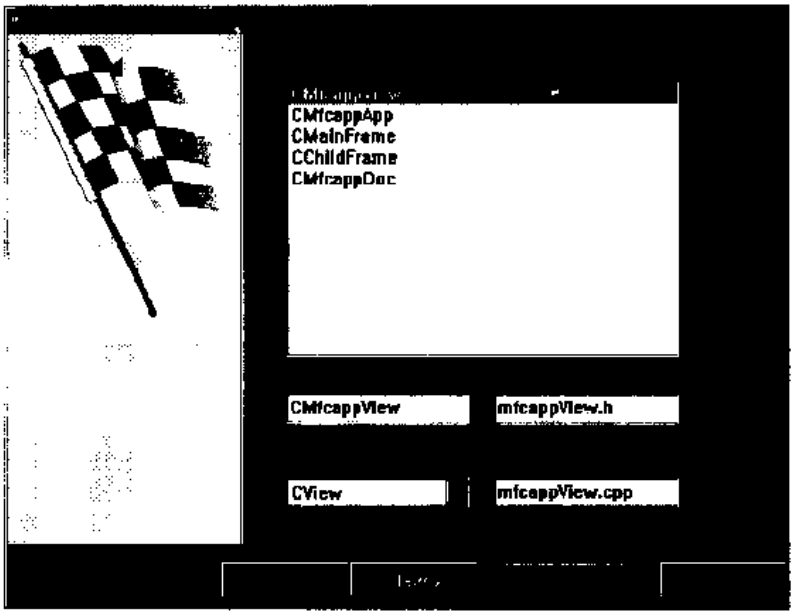



图 6-8　MFC AppWizard Step 6 of 6 对话框

(8) 设置类名及继承关系。在图 6-8 所示的对话框中显示了向导为应用程序建立的所

有类以及每个类的基类、名称以及所在文件。要修改某一个类，只需在 AppWizard creates the following classes for you 列表框中单击要修改的类，然后，在 Class name 文本框中修改类名，在 Base class 下拉列表框中选择所继承的基类，在 Header file 文本框中修改头文件名，在 Implementation file 文本框中修改实现文件名。

设置好各项后单击 Finish 按钮，出现 New project Information 对话框(如图 6-9 所示)。该对话框显示了利用向导所建立的应用程序的类型、主要类、该应用程序所具有的特性和此项目所在的项目目录等信息，单击 OK 按钮。

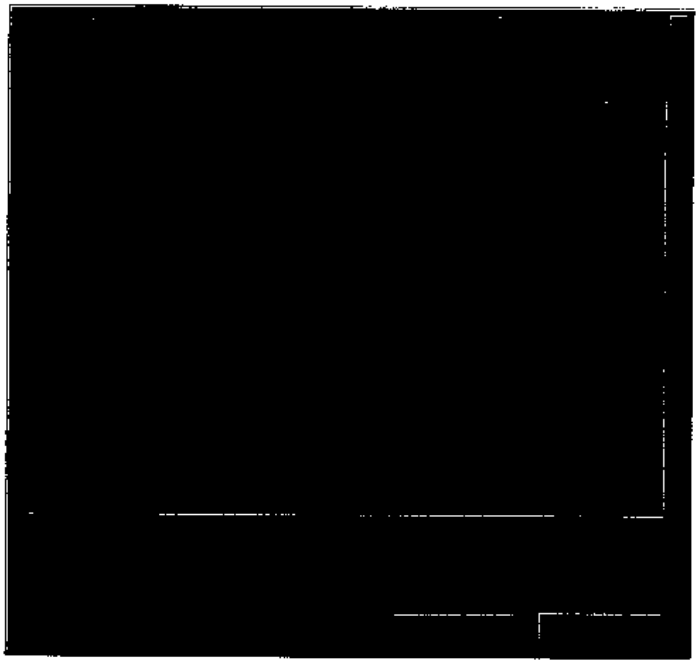


图 6-9 New Project Information 对话框

(9) MFC AppWizard 为应用程序生成必要的初始化文件并在项目工作区窗口打开生成的项目。如图 6-10 所示。

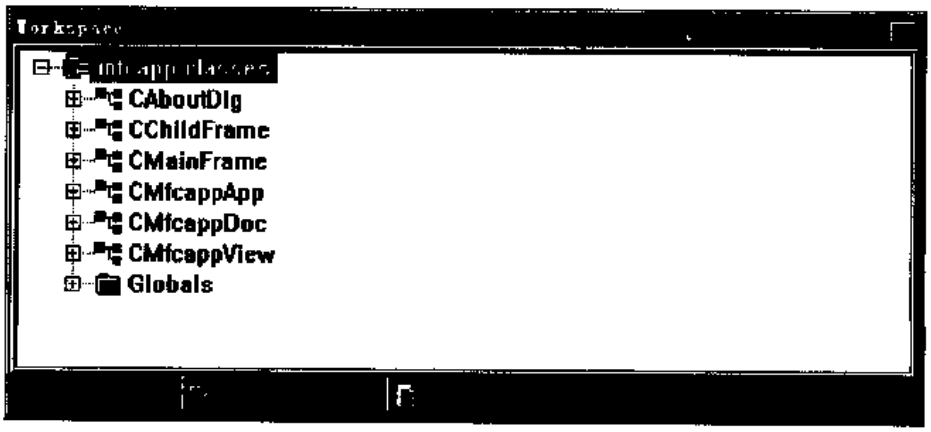



图 6-10 在项目工作区窗口打开项目 mfcapp

(10) 用 MFC AppWizard 生成应用程序的初始化文件后，就可建立并运行应用程序。要建立应用程序，首先选择要建立的项目版本，然后从 Build 菜单中选择 Build * * * .exe 命令。建立应用程序之后，可从 Build 菜单选择 Execute 命令运行应用程序。

**注释：**

生成 SDI(单文档接口)应用程序与生成 MDI 接口应用程序所遵循的基本步骤大致相同，只是生成 SDI 应用程序需要在图 6-2 所示的 MFC AppWizard Step 1 对话框中选择 Single document 选项，然后依次完成各步向导的提示，MFC AppWizard 即可为应用程序生成必要的初始化文件并在项目工作区窗口打开生成的 SDI 项目，如图 6-11 所示。

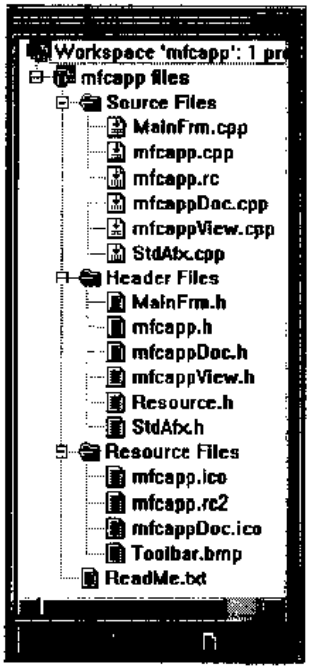



图 6-11　在项目工作区窗口打开项目 mfcsdi

## 6.2　向导生成的文件及作用

通过应用前面所介绍的向导，我们已经拥有了 Windows 应用程序要求的基本要素，至此程序已具有了标准窗口、菜单栏和工具栏，当然，这些菜单和工具绝大部分还不具备实际功能。那么窗口、菜单和工具从何而来呢？它们来自向导所产生的一系列文件，包括资源文件、C + + 源代码文件和 C + + 头文件等。

1.向导所产生的资源文件

向导产生的标准 Windows 应用程序所需的资源文件有：

• Mfcsdi.ico：该文件是应用程序运行时的图标，在应用程序运行时将显示在主窗口标题条的最左端，可以通过双击它调用图标编辑器来进行修改。

• Mfcsdi.rc2:这是 Visual C++不能直接编辑的资源文件,一般情况下可以暂不考虑,如果需要修改,请使用标准的 Windows 资源定义语句。

• MfcsdiDoc.ico:这也是一个图标文件,只是在应用程序运行时显示在文档窗口标题栏的最左端,如果文档窗口处于最大化状态,则显示在菜单栏的左端,它是文档的图形化代表。同样地,也可以通过双击它来调用图标编辑器进行修改。

• Toolbar.bmp:工具栏位图文件,它是应用程序中的工具栏的图形表示,双击它则可调用映像编辑器修改它。映像编辑器也可以用来编辑 GIF 文件、JPEG 文件、图标文件以及光标文件。

2.向导产生的 C++源代码文件

• MainFrm.cpp:此文件是实现 MDI 应用程序所要求的框架窗口的 C++源代码,它包含了框架类 CMainFrame,该类继承自 CFrameWnd,用于控制所有多文档接口应用程序的框架特性。程序清单 6-1 列出了 MainFrame.cpp 的完整源代码并做了注释。

**程序清单 6-1:CMainFrame 类的实现部分**

```
// MainFrm.cpp : CMainFrame 类的实现部分

#include "stdafx.h"
#include "Mfcsdi.h"
#include "MainFrm.h"

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

// CMainFrame
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
//以下为消息映射部分,消息映射以 BEGIN_MESSAGE  MAP 宏开始
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)

    // 注意:ClassWizard(类向导)将可能添加或删除消息映射宏
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()//消息映射以 END_MESSAGE_MAP 宏结束,

static UINT indicators[] =
{
```

```
        ID_SEPARATOR, // 状态栏指示器
        ID_INDICATOR_CAPS, // 大小写切换键(Caps Lock)状态指示
        ID_INDICATOR_NUM, // 数字锁定键(Num Lock)状态指示
        ID_INDICATOR_SCRL, // 滚动锁定键(Scroll Lock)状态指示
    };

        // CMainFrame 类的构造函数和析构函数

    CMainFrame::CMainFrame()
    {
        // 在这里添加成员初始化代码

    }

    CMainFrame::~CMainFrame()
    {
    }

    int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
    {
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

        if (! m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        ! m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
      {
        TRACE0("Failed to create toolbar\n");
        return -1;      // 若建立工具栏失败,则返回-1
      }
      if (! m_wndStatusBar.Create(this) ||
        ! m_wndStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof
(UINT)))
      {
        TRACE0("Failed to create status bar\n");
        return -1;      //若建立状态栏失败,则返回-1
      }
      // 如果不想要船坞化的工具栏,则删除下面的三行代码
      m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);EnableDocking(CBRS_ALIGN_
ANY);
```

```
    DockControlBar(&m_wndToolBar);

    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
if( ! CFrameWnd::PreCreateWindow(cs) )
  return FALSE;
  //通过修改 CREATESTRUCT cs 可以修改窗口类及其风格

  return TRUE;
}

// CMainFrame 调试版所需要的诊断信息
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
  CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
  CFrameWnd::Dump(dc);
}

  #endif //_DEBUG

  // 在下面添加 CMainFrame 类的消息处理函数
```

• mfcsdi.cpp:这是应用程序的主文件,它包含了 CMfcsdiApp 类,该类继承自 CWinApp 类。程序清单 6-2 列出了 mfcsdi.cpp 完整源代码并做了相应的注释:

**程序清单 6-2:应用程序主文件 mfcsdi.cpp 的源代码**

```
// mfcsdi.cpp :定义应用程序的类行为。

#include "stdafx.h"
#include "mfcsdi.h"
#include "MainFrm.h"
#include "mfcsdiDoc.h"
#include "mfcsdiView.h"
```

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

// CMfcsdiApp

BEGIN_MESSAGE_MAP(CMfcsdiApp, CWinApp)
  //{{AFX_MSG_MAP(CMfcsdiApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    // 注意:ClassWizard(类向导)将可能添加或删除消息映射宏
  //}}AFX_MSG_MAP
  //标准的基于文档的文件命令
  ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
  ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
  //标准的打印设置命令
  ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
  END_MESSAGE_MAP()

  // CMfcsdiApp 类的构造函数

CMfcsdiApp::CMfcsdiApp()
{
    //在这里添加构造函数代码
}

// 在本程序中有且只有一个 CMfcsdiApp 对象

CMfcsdiApp theApp;

/////////////////////////
// CMfcsdiApp initialization

BOOL CMfcsdiApp::InitInstance()
{
    AfxEnableControlContainer();

    // 标准的初始化部分
    //如果不想使用这些特性并想缩短最终可执行程序的长度,
```

```
    //应删除下面给定的所不需要的初始化例程
#ifdef _AFXDLL
    Enable3dControls();     //当在动态连接库中使用 MFC 时调用此函数
#else
    Enable3dControlsStatic();     //当静态连接 MFC 库时调用此函数
#endif

    //改变此应用程序所存贮的注册键,应该修改此字符串,
    //它可以是公司名或组织名等等
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings(); // 装入标准的 INI 文件选项(包括 MRU)
    //注册应用程序文档模板,文档模板用于连结文档与框架窗口和视图

    CSingleDocTemplate * pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
      IDR_MAINFRAME,
      RUNTIME_CLASS(CMfcsdiDoc),
      RUNTIME_CLASS(CMainFrame), //SDI 主框架窗口
      RUNTIME_CLASS(CMfcsdiView));
    AddDocTemplate(pDocTemplate);

    // 分析标准的外壳命令、DDE 及打开文件的命令行
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // 分派命令行命令
    if (! ProcessShellCommand(cmdInfo))
    return FALSE;

    // 有且只有一个窗口被初始化,显示并更新该窗口
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

    //应用程序 About 命令所用 CAboutDlg 类的对话

class CAboutDlg : public CDialog
{
```

```
public:
    CAboutDlg();

//对话数据
    //{{AFX_DATA(CAboutDlg),
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA,

    // ClassWizard 产生的重载的虚函数。
    //{{AFX_VIRTUAL(CAboutDlg),
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    //}}AFX_VIRTUAL

// 实现部分
protected:
    //{{AFX_MSG(CAboutDlg)
    // 消息处理程序
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
    //消息要处理程序
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
//运行对话框的命令
void CMfcsdiApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}
```

• Mfcsdi.rc:此文件描述了本应用程序的全部资源(除了 Visual C++ 所不能直接编辑的资源之外)。在此例中所用到的资源有:加速键、对话框、图标、菜单、串表、工具栏和版本信息。

• MfcsdiDoc.cpp:该文件包含了 CMfcsdiDoc 类,此文件用于保存文档数据,并且能够打开和保存文件。程序清单 6-3 列出了 MfcsdiDoc.cpp 的完整源代码。

**程序清单 6-3:文档类 CmfcsdiDoc 的实现文件 MfcsdiDoc.cpp 的源代码**

```
// mfcsdiDoc.cpp:CmfcsdiDoc 类的实现部分

#include "stdafx.h"
#include "mfcsdi.h"
#include "mfcsdiDoc.h"

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

// CMfcsdiDoc

IMPLEMENT_DYNCREATE(CMfcsdiDoc, CDocument)

BEGIN_MESSAGE_MAP(CMfcsdiDoc, CDocument)
    //{{AFX_MSG_MAP(CMfcsdiDoc)
    // 注意:ClassWizard(类向导)将可能添加或删除消息映射宏
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// CMfcsdiDoc 构造函数和析构函数

CMfcsdiDoc::CMfcsdiDoc()
{
    // 在这里添加构造函数代码
```

```
}

CMfcsdiDoc::~CMfcsdiDoc()
{
}

BOOL CMfcsdiDoc::OnNewDocument()
{
    if (! CDocument::OnNewDocument())
      return FALSE;

    //添加重新初始化代码。
    //SDI 文档将再次利用该文档
    return TRUE;
}

// CMfcsdiDoc 串行化
void CMfcsdiDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
    //添加保存文件代码。
    }
    else
    {
    //添加装入文件代码
    }
}

// CMfcsdiDoc 调试信息

#ifdef _DEBUG
void CMfcsdiDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMfcsdiDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
```

```
#endif //_DEBUG
```

• MfcsdiView.cpp:此文件提供了文档的视图,在此处实现的 CMfcsdiView 类继承自 View 类,程序清单 6-4 列出 MfcsdiView.cpp 的完整源代码。

**程序清单 6-4:视图类实现文件 MfcsdiView.cpp 的完整源代码**

```
// mfcsdiView.cpp : CMfcsdiView类的实现。
#include "stdafx.h"
#include "mfcsdi.h"
#include "mfcsdiDoc.h"
#include "mfcsdiView.h"

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

// CMfcsdiView

IMPLEMENT_DYNCREATE(CMfcsdiView, CView)

BEGIN_MESSAGE_MAP(CMfcsdiView, CView)//消息映射开始
    //{{AFX_MSG_MAP(CMfcsdiView)
    // 注意:ClassWizard(类向导)将可能添加或删除消息映射宏
    //}}AFX_MSG_MAP
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
  END_MESSAGE_MAP()//消息映射结束

    // CMfcsdiView构造函数和析构函数

  CMfcsdiView::CMfcsdiView()
{
    //在这里添加构造函数代码

}

CMfcsdiView::~CMfcsdiView()
```

```
}
}

BOOL CMfcsdiView::PreCreateWindow(CREATESTRUCT& cs)
{
    // 通过修改 CREATESTRUCT cs 来修改窗口类及风格

    return CView::PreCreateWindow(cs);
}

// CMfcsdiView 绘图

void CMfcsdiView::OnDraw(CDC * pDC)
{
    CMfcsdiDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    //添加描绘特定数据的代码
}

// CMfcsdiView 打印

BOOL CMfcsdiView::OnPreparePrinting(CPrintInfo * pInfo)
{
    // 缺省的打印预处理
    return DoPreparePrinting(pInfo);
}

void CMfcsdiView::OnBeginPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    //添加打印前的额外初始代码
}

void CMfcsdiView::OnEndPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    //添加打印后的清理代码
}

// CMfcsdiView 调试信息

#ifdef _DEBUG
void CMfcsdiView::AssertValid() const
```

```
{
    CView::AssertValid();
}

void CMfcsdiView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMfcsdiDoc * CMfcsdiView::GetDocument() //非调试版则作为内嵌函数
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMfcsdiDoc)));
    return (CMfcsdiDoc *)m_pDocument;
}
#endif //_DEBUG
```

• StdAfx.cpp:利用 MFC AppWizard 所产生的任何类型的应用程序均有此文件,它用于包含 MFC 标准头文件。

**程序清单 6-5:StdAfx.cpp 文件**

```
// stdafx.cpp : 此源文件用于包含所有标准的头文件
// mfcsdi.pch : 预编译头文件
// stdafx.obj : 包含了预编译类型信息

#include "stdafx.h"
```

3. MFC AppWizard 产生的 C++ 头文件

MFC AppWizard 产生的头文件刚好与前面所介绍的 C++ 源代码文件一一对应。在头文件中,MFC AppWizard 自动产生了所用到的各种类的声明,而在 C++ 源代码文件中则是其对应的实现部分。这些头文件有:MainFrm.h(主窗口类头文件)、Mfcsdi.h(应用程序类头文件)、MfcsdiDoc.h(应用程序文档类头文件)、MfcsdiView.h(应用程序视图类头文件)、Resource.h(资源头文件)和 StdAfx.h(包含所有头文件的头文件)。

4. 向导所产生的其他文件

向导除了生成以上对于建立应用程序必不可少的文件外,还生成了以下文件:

• ReadMe.txt 文件:此文件描述了项目所涉及到的所有文件及每一个文件的作用。

• 扩展名为 dsp 的项目文件:此文件实际上就是项目本身,它可以供其他用户共享。

• 扩展名为 clw 的文件:此文件包含的信息供 ClassWizard 在编辑已存在类和添加新类时使用,同时 ClassWizard 在建立和编辑消息映射和对话框消息映射所需的信息也保存在此文件中。

• 工作区文件:它保存了应用程序项目信息。

# 第 7 章

# 边框窗口、文档和视图

边框窗口、文档和视图是 MFC 应用程序最主要的对象。Visual C++ 分别提供了边框窗口、文档和视图的多种基类。正确理解三者的关系，并且可正确地选择相应的基类派生出能够解决实际问题的派生类是开发基于文档的应用程序所必不可少的。本章先对这 3 个概念做简单介绍，然后通过实例详细地介绍如何正确运用它们进行程序开发。

## 7.1 边框窗口

边框和边框所围的内容组成边框窗口，即应用程序主窗口。边框和边框所围的内容在 MFC 类库中分别用不同的类来表示。边框由窗口类管理，这些窗口类包括：SDI 应用程序边框窗口所继承的类 CFrameWnd，MDI 应用程序的主边框窗口所继承的类 CMDIFrameWnd，MDI 应用程序的文档边框窗口所继承的类 CMDIChildWnd 等。对于边框窗口，程序要完成的主要工作是设置窗口外观，MDI 应用程序还要维护活动文档窗口与非活动文档窗口之间的切换。对于边框所围的的内容则由视图类管理。所有视图类都继承自 CView 类。

### 7.1.1 边框窗口和视图

视图窗口是边框窗口的子窗口，也是应用程序要显示数据的区域，一般称之为用户工作区。边框窗口在视图窗口周围提供了一个可见的边框，一个标题栏以及其他标准窗口控件（如控制菜单和控制按钮）。用户编写基于文档的应用程序时，大部分工作就是在编写继承自 CView 类的派生类，利用该派生类来显示文档数据，并根据用户操作来维护视图和更新文档数据。所有的绘制操作及文档的串行化均在视图类的实现文件中完成。

### 7.1.2 边框窗口与文档边框窗口

每个应用程序都有一个边框窗口，或称为主边框窗口。主边框窗口的标题栏中显示有应用程序的名称及图标。每个文档都有一个文档边框窗口，一个文档边框窗口至少含有一个视图用于显示文档的数据。每个 SDI 应用程序都有一个从 CFrameWnd 类派生的边框窗口，该窗口既是主边框窗口，又是文档边框窗口。

每个 MDI 应用程序的主边框窗口都是从 CMDIFrame 类派生的，而文档边框窗口则是从 CMDIChildWnd 类派生的。应用程序运行时文档边框窗口只能显示在主边框窗口的用户工作区。

创建应用程序边框时，MFC AppWizard 为应用程序的主边框窗口派生一个边框窗口类，缺省情况下为 CMainFrame，头文件为 MainFrm.h，实现文件为 MainFrm.CPP。

SDI 应用程序的 CMainFrame 类是从 CFrameWnd 类派生的，MDI 应用程序的 CMainFrame 类是从 CMDIFrameWnd 类派生的。

### 7.1.3 边框窗口的创建和销毁

应用程序的边框窗口都是动态建立的。SDI 应用程序只有一个边框窗口，它随着应用程序的运行而建立，随着应用程序的结束而销毁；对于 MDI 应用程序，主边框窗口随着应用

程序的运行而建立，随着应用程序的结束而销毁，文档边框窗口则在用户打开文档或创建新文档时建立，文档关闭则文档边框窗口也随之关闭。

边框窗口的创建需要按照以下几个步骤来进行：

(1) 选择创建文档的命令(包括新建和打开文档)时，将调用文档模板来创建文档对象、视图以及显示视图的文档边框窗口。缺省情况下，新建文档操作将调用 CWinApp 类的成员函数 OnFileNew，打开文档操作将调用 CWinApp 类的成员函数 OnFileOpen(如果需要可以重载此函数)。这两个命令的消息映射位于应用程序的主文件中。

(2) 创建文档边框窗口时，文档模板先创建边框窗口类的一个对象，相应代码位于应用程序的主文件中。

(3) 调用边框窗口对象的 LoadFrame 成员函数从资源中获取创建信息并创建 Windows 窗口，把窗口句柄与边框窗口对象链接起来。实现这些功能的程序代码如下：

```
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CMyMFCSDIDoc),
RUNTIME_CLASS(CMainFrame),       // 主 SDI 边框窗口
RUNTIME_CLASS(CMyMFCSDIView));
AddDocTemplate(pDocTemplate);
```

(4) 创建文档边框窗口的子窗口。

边框窗口的销毁随着窗口的关闭自动调用析构函数，并由析构函数释放其所占用的资源。

### 7.1.4　管理当前视图

边框窗口可以跟踪当前活动的视图。如果边框窗口含有多个视图，则当前视图就是最近使用的视图。当活动视图改变时，边框窗口通过调用 CView 的成员函数 OnActivateView 来通知当前视图。OnActivateView 函数原型为：

```
virtual void OnActivateView( BOOL bActivate, CView* pActivateView, CView* pDeactiveView );
```

其中，bActivate 用来确定视图是否被激活；pActivateView 指向要激活的视图对象；pDeactivateView 指向不被激活的视图对象。

### 7.1.5　管理菜单、控件栏和加速键

边框窗口通常使用 ON_UPDATE_COMMAND_UI 来管理用户对象，如菜单、工具栏按钮和状态栏的更新。

对于 MDI 应用程序，MDI 主边框窗口管理菜单栏、标题栏、工具栏按钮和其他控件。当没有活动的 MDI 子窗口时，MDI 主边框窗口包含由菜单栏使用的缺省菜单；当有活动的 MDI 子窗口时，MDI 主边框窗口的菜单栏由活动的 MDI 子窗口菜单所取代。

MDI应用程序中,出现在 Windows 菜单中的标准命令都有缺省实现方式(由 CMDIFrameWnd 类提供)。

每个边框窗口维护一个可选的加速键表,使用加速键表可以自动转换键盘加速键,这样可以非常容易地定义调用菜单命令的加速键。

### 7.1.6 定制边框窗口

通过重设 CWnd 的成员函数 PreCreateWindow 来定制边框窗口。该函数的原型为:

```
virtual BOOL PreCreateWindow( CREATESTRUCT& cs );
```

若要重设 PreCreateWindow 函数,需先在创建窗口前修改 CREATESTRUCT 结构替换缺省设置。

CREATESTRUCT 结构的定义如下:

```
typedef struct tagCREATESTRUCT{
LPVOID lpCreateParams;     //指向用于创建窗口的数据
HANDLE hInstance;     //拥有新窗口的本实例句柄
HMENE hMenu;     //由新窗口使用的菜单
   HWND   hwndParent;     //拥有新窗口的文本窗口。如为顶层窗口,则为 NULL
   int    cy;     //新窗口的高度
   int    cx;     //新窗口的宽度
   int    y;     //新窗口左上角的 y 坐标(相对父窗口而言)
   int    x;     //新窗口左上角的 x 坐标(相对父窗口而言)
   LONG    style;     //新窗口的风格
LPCSTR lpszName;     //新窗口的名字
LPCSTR lpszClass;     //新窗口的窗口类名
DWORD dwExStyle;     //新窗口的扩展风格
}CREATESTRUCT;
```

**注释:**

窗口类风格可以为:

WS _ BORDER,WS _ CAPTION,WS _ CHILD,

WS _ CLIPCHILDREN,WS _ CLIPSBLINGS,WS _ DISABLED,

WS _ DLGFRAME,WS _ GROUP,WS _ HSCROLL,

WS _ MAXMIZE,WS _ MAXIMIZEBOX,WS _ MINIMIZE,

WS _ MINIMIZEBOX,WS _ OVERLAPPED,WS _ OVERLAPPEDWINDOW,

WS _ POPUP,WS _ POPUPWINDOW,WS _ SYSMENU,

WS _ TABSTOP,WS _ THICKFRAME,WS _ VISIBIE,

WS _ VSCROLL。

扩展风格可以为:

WS _ EX _ ABSPOSITION,WS _ EX _ ACCEPTFILES,

WS _ EX _ CLIENTEDGE, WS _ EX _ CONTEXTHELP,

WS _ EX _ CONTROLPARENT, WS _ EX _ DLGMODALFRAME,

WS _ EX _ LEFT, WS _ EX _ LEFTSCROLLBAR,

WS _ EX _ LTRREADING, WS _ EX _ MDICHILD,

WS _ EX _ NOPARENTNOTIFY, WS _ EX _ OVERLAPPEDWINDOW,

WS _ EX _ PALETTEWINDOW, WS _ EX _ RIGHT,

WS _ EX _ RIGHTSCROLLBAR, WS _ EX _ RTLREADING,

WS _ EX _ SMCAPTION, WS _ EX _ STATICEDGE,

WS _ EX _ TOOLWINDOW, WS _ EX _ TOPMOST,

WS _ EX _ TRANSPARENT, WS _ EX _ WINDOWEDGE。

lpszClass 用于存放 Windows 窗口类的名称。Windows 窗口类不是 C + + 类，而是由 Windows 系统维护的数据结构，它可保存通用的特征供创建窗口时使用。

注册 Windows 窗口类必须调用全局函数 AfxRegisterWndClass，其原型为：

```
LPCTSTR AFXAPI AfxRegisterWndClass( UINT nClassStyle, HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0, HICON hIcon = 0 );
```

其中，参数 nClassStyle 用于定义窗口类风格，类风格可以为：CS _ BYTEALIGNCLIENT, CS _ BYTEALIGNWINDOW, CS _ CLASSDE, CS _ DBLCLDKS, CS _ GLOBASCALSS, CS _ HREDRAW, CS _ NOCLOSE, CS _ OWNDC, CS _ PARENTDC, CS _ SAVEBITS 和 CS _ UREDTAW。

hCursor 为光标资源句柄；hbrBackground 为背景画刷资源句柄；hIcon 为图标资源句柄。

## 7.2　文档与视图

文档用来管理应用程序的数据，视图用于显示文档并管理与用户的交互作用。实际上视图在用户与文档之间起桥梁作用。文档打开时，视图类的实现文件中应该有读取的文档数据，并将其显示于视图中。用户在视图区绘制的对象也应当通过视图类的实现文件添加到适当的文档类成员中，这些数据存入文档的操作由文档串行化来实现。

### 7.2.1　文档与视图之间的联系

总的来说，文档与视图之间存在以下关系：

• 由 MFC AppWizard 创建的文档类是从 CDocument 类派生的，视图类是从 CView 类派生的。

• CDocument 类为应用程序定义的文档类提供基本功能，而 CView 类为应用程序定义的视图类提供基本功能。

• 视图和文档链接在一起，在文档与用户间起中介作用。视图在屏幕上显示文档数据并把用户输入转换成对文档的操作。

• 类库中文档与视图数据的显示以及用户对数据所做的操作分离。数据的所有更改

都通过文档类管理,而视图则调用这个接口来访问和更新数据。

### 7.2.2 使用文档管理数据

文档是由 CDocument 类派生的对象。当利用 AppWizard 建立一个应用程序样板时,Visual C++会自动地提供一个文档对象的边框结构并把实现此结构的代码放入文件 YourAppDOC.CPP 中(这里的 YourApp 是具体项目名称)。管理文档的数据是通过 CDocument 类派生的类对象来进行的。这样,可以利用 CDocument 类的固有能力来管理数据的串行化和其他任务。CDocument 的一个特征是它通常用在由 AppWizard 创建的程序中。当利用 AppWizard 产生一个应用程序边框并且创建一个文件 YourAppDoc.cpp 内的对象时,AppWizard 将自动地把菜单命令 Save, Save As 和 Open 与你的文档对象联系起来。

### 7.2.3 在视图中显示数据及与用户交互作用

视图以图形方式显示文档数据,接受用户输入并将其解释成对文档的操作。编程时,对视图类的处理主要包括以下几个方面:

- 处理视图类的 OnDraw 成员函数,该函数负责提供文档数据。
- 将 Windows 消息与用户界面对象及视图类中的消息处理函数链接。
- 实现消息处理函数,以便解释用户的输入。
- 根据需要,在派生类视图中重设 CView 的其他成员函数,如重设 OnInitialUpdate,以便进行必要的视图初始化工作等。

此外,MFC 类库还提供了几个专用的派生视图类,利用这些视图类可以增强应用程序视图的功能,它们是:

- CScrollView 类:提供自动滚动和缩放显示功能。
- CFormView 类:提供可滚动的视图,用于显示由对话框控件组成的表单。
- CRecordView 和 CdaiRecordView 类:用于在控件中显示数据库表中字段的表单视图。
- CEditView 类使视图具有可编辑文本控件的特性,可以使用 CEditView 对象实现简单的文本编辑器。

1.在视图中绘制

所有应用程序的绘制工作都是在视图的 OnDraw 成员函数中实现的。在 MFC AppWizard 中,已定义了 OnDraw 函数的基本结构:

```
vcid CMyAppView:: OnDraw (CDC * pDC){
CMyAppDoc * pDoc = GetDocument();
ASSERT_VALID(pDoc);
//在此处添加绘制代码
}
```

函数 OnDraw 通过调用文档成员函数来获取文档数据,并通过调用传给 OnDraw 的设备文本对象的成员函数来显示数据。当文档数据更改时,视图必须重新绘制以反映数据的

更改。

成员函数 OnUpdate 使视图的整个客户区无效时，可以重设 OnUpdate 函数以便反映文档数据的更改情况。

CDocument 类的成员函数 OnUpdateAllViews 和 CView 类的成员函数 OnUpdate 都包含用于描述文档被修改信息的参数。OnUpdateAllViews 函数的原型为：

```
void UpdateAllViews( CView* pSender, LPARAM lHint = 0L,
CObject* pHint = NULL );
```

OnUpdate 函数的的原型为：

```
virtual void OnUpdate( CView* pSender, LPARAM lHint, CObject* pHint );
```

如果所有的视图都要更新，则参数 pSender 为 Null，否则为指向视图的指针；lHint 包含与修改有关的信息，可以传递任何数据，pHint 为指向保存修改信息的对象，可以传递从 CObject 类派生的任何对象的指针。

2.与用户交互作用

在视图类中，通常要定义消息处理函数，以便处理标准 Windows 消息(由鼠标和键盘动作产生)和命令(来自菜单、工具栏按钮和加速键)。

消息处理函数将用户动作或命令解释为数据输入、选择或编辑，例如，要处理 Edit 菜单的剪切、复制、粘贴或删除等与剪贴板有关的命令，消息处理函数就要调用 CWnd 类中与剪贴板有关的成员函数，以便将选中的数据传进或传出剪贴板。

### 7.2.4 滚动与缩放

CScrollView 类是由通用的 CView 类派生的一个专用视图类，支持自动滚动和缩放显示功能。

1.滚动显示

当文档数据增加或缩小视图的边框窗口时，文档通常会比视图窗口大，这时，视图必须支持滚动。

任何视图都可以在 OnHScroll 和 OnVScroll 消息处理函数中处理滚动条消息。利用自动滚功能的视图类必须从 CScrollView 类派生。

首次创建滚动视图时，要在视图类中重设 CView 的成员函数 OnInitialOnUpdate 或 OnUpdate，并从中调用 CScrollView 的成员函数 SetScrollSize 来计算滚动视图的大小。此外，SetScrollSize 函数还可用于设置视图的映射模式、滚动视图的大小、水平和垂直滚动量。该函数的原型为：

```
void SetScrollSizes( int nMapMode, SIZE sizeTotal, const SIZE& sizePage = sizeDefault, const SIZE& sizeLine = sizeDefault );
```

其中，nMapMode 用于设置视图的映射模式，可以为 MM _ TEXT，MM _ HIMETRIC，MM _ TWIPS，MM _ HIENGLISH，MM _ LOMETRIC 和 MM _ LOENGLISH。

sizeTotal 用来指定滚动视图的大小;sizePage 用来指定每次单击滚动框时要滚动多少页;sizeLine 用来指定每次单击滚动箭头时要滚动多少行。

2.缩放显示

调用 CScorollView 的缩放显示功能可使视图自动适应边框窗口的大小。要使用缩放显示功能,可以在视图类中重设 CView 的成员函数 OnInitialUpdate,并从中调用 CScrollView 的成员函数 SetScaleToFitSize。SetScaleToFitSize 的函数原型为:

```
void SetScaleToFitSize( SIZE sizeTotal );
```

其中,sizeTotal 是被缩放视图的水平和垂直大小。

### 7.2.5 多文档类型

MFC AppWizard 创建的应用程序边框开始时只支持单文档类。要使用多文档类,必须创建新的文档类。

要创建新的文档类,可使用 Class Wizard 对话框中的 Add Class 选项,从 Class Type 下拉列表框中选择 CDocument 类作为派生新类的基类,并提供必要的文档信息,然后再实现这个新类的数据。

创建新的文档类,必须在应用程序对象的成员函数 CMultiDocTemplate 中增加对 AddDocTemplate 函数的调用。

### 7.2.6 多视图

MFC 支持以下 3 种多视图模式:

- 同一文档的多个视图对象,每个对象显示在独立的 MDI 子窗口中。
- 同一文档的边框窗口中有同一文档的多个视图对象。
- 单个边框窗口中有不同类的视图对象,多个视图共享单个边框窗口,每个视图从不同的类构造。

### 7.2.7 分割窗口

在分割窗口中,窗口可以被分割成多个可滚动块。可以使用分割条调整每个分割部分(面板)的相对大小,每个面板都是同一文档的一个视图。在动态分割窗口中,视图是属于同一文档类的;在静态分割窗口中,视图是属于不同文档类的。这两种分割窗口都由 CSplitterWnd 类支持。

## 7.3 SDI 应用程序编程步骤

这一节首先利用 MFC AppWizard 生成一简单的 SDI 应用程序框架,然后再利用 Class Wizard 工具对此项目进行进一步丰富,使其可以在视图中绘制直线,并能够将所绘制的直线保存到文档中,以后可由此应用程序打开这些文档进行进一步的修改。

### 7.3.1　建立应用程序基本框架

利用 MFC AppWizard 建立应用程序的基本框架的详细步骤可参见第 6 章中所做的介绍。MFC AppWizard 可为应用程序生成开始文件和项目文件，但是 MFC AppWizard 生成 SDI 应用程序的基本边框时，只派生文档类、视图类、主边框窗口类和应用程序类。

以第 6 章生成的 mfcsdi 为例，mfcsdi 的文档类为 CmfcsdiDoc。头文件为 mfcsdi.h，它定义了该类结构。成员函数的实现则在文件 mfcsdiDoc.Cpp 中。要使文档能够处理更多的数据类型，可修改这两个文件，分别向其中添加更多的类定义及其实现。文档数据的更新操作（由串行化功能实现）的代码也位于此类的实现文件中。

mfcsdi 的视图类为 CmfcsdiView，头文件为 mfcsdiView.h，实现文件为 mfcsdiView.Cpp。具体绘制图形的实现部分就在此类中实现。

mfcsdi 的主边框窗口为 CMainFrame，是从 CFrameWnd 类派生的，头文件为 MainFrm.h，实现文件为 MainFrm.cpp。如果需要修改窗口风格则修改此类的相关文件即可。

mfsdi 的应用程序类为 CmfcsdiApp，头文件为 mfcsdi.h，实现文件为 mfcsdi.cpp。

### 7.3.2　处理视图

1. *在视图类中添加数据成员*

在视图类中可添加的新的数据成员用来管理在视图中绘制时的状态，如本例中的光标位置及状态，具体方法为：

(1)打开文件 MyMFCSDIView.h，将下列语句添加到 CMyMFCSDIView 类定义中：

```
protected:
 int m_Draw;        //表示是否正在拖动鼠标;
 HCURSOR m_HCursor;        //存放光标句柄
 CPoint m_pOld;        //存放光标的当前位置
 CPoint m_pOrigin;        //存放鼠标左键按下时光标所在的位置(画线的起点)
```

(2) 打开文件 MyMFCSDIView.Cpp。将以下代码添加到 CMyMFCSDIView 类的构造函数中来初始化数据成员 m_Draw 和 m_Hcursor：

```
m_Draw=0;//初始时,未拖动鼠标
m_HCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
```

其中，AfxGetApp 返回一个指向应用程序对象的指针，通过该指针调用 CWinApp 的成员函数 LoadStandardCursor 来得到十字型光标。

2. *用 ClassWizard 映射鼠标消息*

使用 ClassWizard 来映射鼠标消息的方法为：

(1) 从 View 菜单中选择 ClassWizard 命令，弹出 MFC ClassWizard 对话框。

(2) 选择 Message Maps 选项卡。

(3) 从 Class name 下拉列表框中选择要处理的类名 CMyMFCSDIView。

(4) 在 Object IDs 列表框中选择定义消息处理函数名 CMyMFCSDIView。

(5) 在 Messages 列表框中选择要映射的鼠标消息为 WM _ LBUTTONDOWN，然后单击 Add Function 按钮。这 5 步设置如图 7-1 所示。

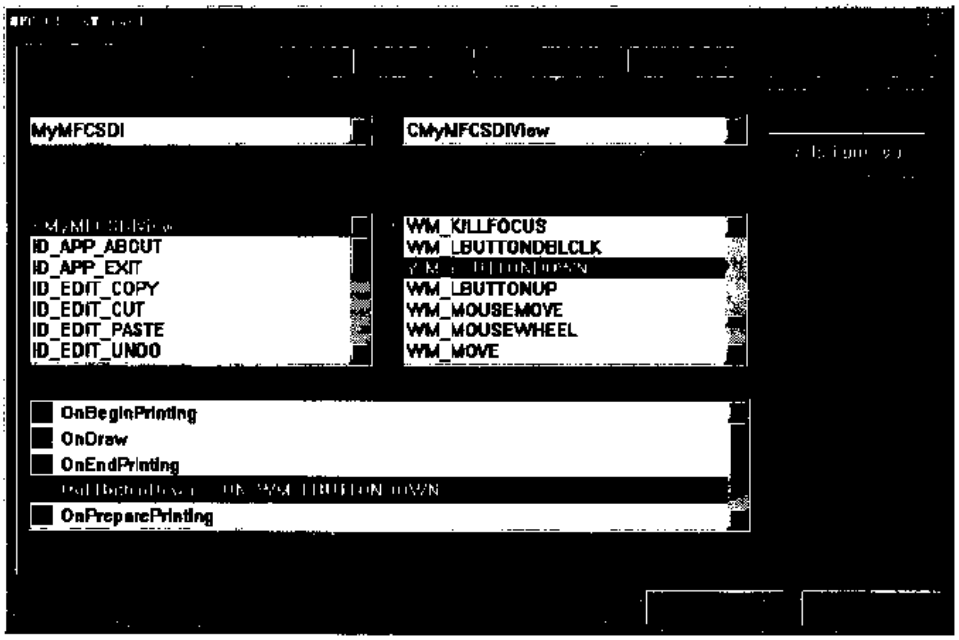



图 7-1　映射鼠标消息 WM _ LBUTTONDOWN 时的 MFC ClassWizard 对话框

(6)重复步骤(5)，映射鼠标消息 WM _ MOUSEMOVE 和 WM _ LBUTTONUP，分别如图7-2、图 7-3 所示。

生成消息处理函数时，ClassWizard 在头文件 MyMFCSDIView.h 中把以下函数声明添加到类定义中：

```
//生成的消息映射函数
protected:
  //{{AFX_MSG(CMyMFCSDIView)
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  afx_msg void OnMouseMove(UINT nFlags, CPoint point);
  afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

在实现文件 MyMFCSDIView.Cpp 中，ClassWizard 把以下消息映射宏添加到 CMyMFCSDIView 类的消息映射中：

```
//{{AFX_MSG_MAP(CMyMFCSDIView)
ON_WM_LBUTTONDOWN()
ON_WM_MOUSEMOVE()
```

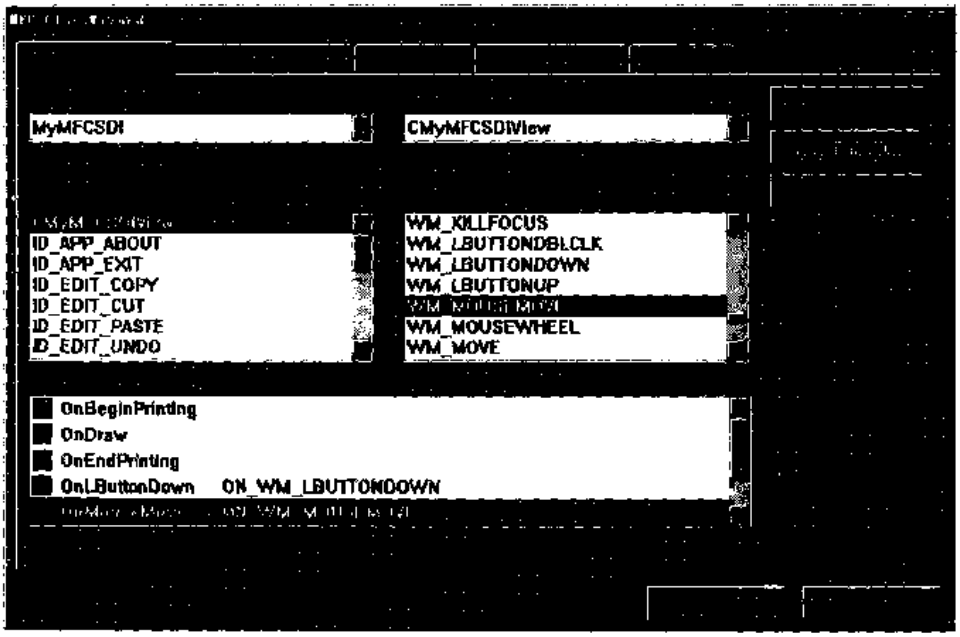



图 7-2　映射鼠标消息 WM _ MOUSEMOVE 时的 MFC ClassWizard 对话框

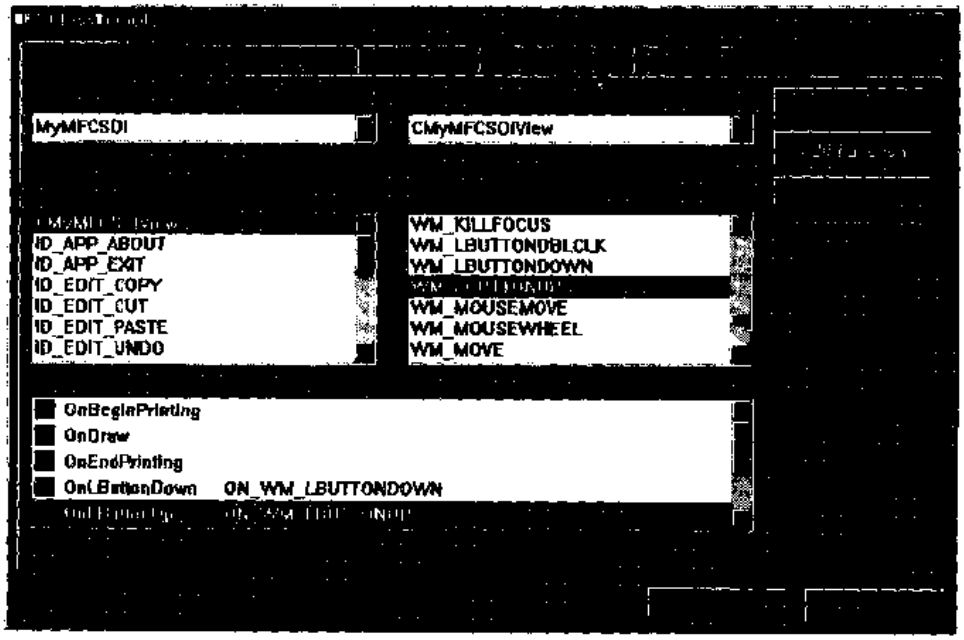



图 7-3　映射鼠标消息时 WM _ LBUTTONUP 的 MFC ClassWizard 对话框

```
ON _ WM _ LBUTTONJP()
//{|AFX _ MSG _ MAP
```

此外，ClassWizard 在实现文件 MyMFCSDIView.cpp 中插入消息处理函数的如下基本定义：

```
//鼠标左键按下的消息处理函数
void CMyMFCSDIView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
    // TODO: Add your message handler code here and/or call default
    //TODO:在此处添加消息处理函数代码和/或调用缺省代码
    CView::OnLButtonDown(nFlags, point);
}
//鼠标移动的消息处理函数
void CMyMFCSDIView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    //TODO:在此处添加消息处理函数代码和/或调用缺省代码
    CView::OnMouseMove(nFlags, point);
}
//鼠标左键放开的消息处理函数
void CMyMFCSDIView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    //TODO:在此处添加消息处理函数代码和/或调用缺省代码
    CView::OnLButtonUp(nFlags, point);
}
```

3.添加消息处理函数代码

映射鼠标消息后,接着要添加消息处理代码,以便能在视图窗口中画直线,具体方法如下。

(1)在消息处理函数 CMyMFCSDIView::OnLButtOnDown 中添加如下代码:

```
m_pOld = point;          //存放鼠标光标的当前位置
m_pOrigin = point;       //存放画线的起始位置
SetCapture();            //捕获鼠标
m_Draw = 1;              //表示正在拖动鼠标
RECT rect;
GetClientRect(&rect);    //获取视图窗口区客户的坐标
ClientToScreen(&rect);   //转换客户区坐标为屏幕坐标
ClipCursor(&rect);       //将光标限定在视图窗口客户区内,接着调用 CWnd 的成员函数
ClipCursor(&rect);       //SetCapture 使随后的鼠标消息都被送往视图窗口
```

**注释:**

如果要释放鼠标并恢复正常输入,可调用 Windows API 函数 ReleaseCapture。

其中,SetCapture 函数返回指向当前窗口对象的指针——函数的原型:CWnd * SetCapture();然后调用 CWnd 的成员函数 GetClientRect 获取窗口客户区的坐标,函数原型为:

```
void GetClientRect(LPRECT lpRect);
```

RECT 结构定义如下:

```
ypedef struct tagRECT {
    LONG left; //左上角 X 坐标
    LONG top; //左上角 Y 坐标
    LONG right; //右下角 X 坐标
    LONG bottom; //右下角 Y 坐标
} RECT;
```

然后,调用 CWnd 的成员函数 ClientToScreen 将窗口客户区坐标转换为屏幕坐标,函数原型为:

```
void ClientToScreen( LPRECT lpRect )
```

最后,调用 Windows API 函数 ClipCursor 将光标限定在窗口客户区内。函数原型为:

```
void ClipCursor(LPRECT lpRect);
```

(2)在消息处理函数 CMyMFCSDIView::OnMouseMove 中添加如下代码:

```
SetCursor(m_HCursor);//设置自定义光标
if(m_Draw){ //正在拖动鼠标
CClientDC dc(this); //创建设备文本对象
dc.SetROP2(R2_NOT); //设置绘图模式,以屏幕反转色为绘图颜色
dc.MoveTo(m_pOrigin);//删除前一 WM_MOUSEMOVE 画的直线
dc.LineTo(m_pOld);
dc.MoveTo(m_pOrigin);//以起始位置到当前位置画一条新的直线
dc.LineTo(point);
m_pOld=point;//存放鼠标光标的当前位置
}
```

(3)在消息处理函数 CMyMFCSDIView::OnLButtOnUp 中添加如下代码:

```
if(m_Draw){
  m_Draw=0;//表示鼠标已释放
  ReleaseCapture();//释放鼠标捕获并恢复正常输入
  ClipCursor(NULL);//使光标可以移到屏幕任意位置
}
```

4.定制视图窗口

要修改视图窗口的特性,可在 CMyMFCSDIView 的成员函数 PreCreateWindow 中添加以下代码:

```
cs.lpszClass=AfxRegisterWndClass(
  CS_HREDRAW|CS_VREDRAW,//类风格
  0,//不用光标
  (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
  0);//没有图标
```

```
return CView::PreCreateWindow(cs);
```

5.修改窗口标题栏

缺省时,标题栏中显示的标题名为“MyMFCSDI \ n \ nMyMFCS \ n \ n \ nMyMFCSDI. Document \ nMyMFCS Document”。若要修改标题栏,可执行以下步骤:

(1)在项目工作区窗口,切换到 ResourceView 选项卡。

(2)选择 String Table(串表)资源。

(3)在串编辑器中选择第一项,弹出 String Properties 对话框。

(4)在 Caption 文本框中将原有标题栏内容“MyMFCSDI \ n \ nMyMFCS \ n \ n \ nMyMFCSDI. Document \ nMyMFCS Document”改成“单文档绘图程序示例”,如图 7-4 所示。

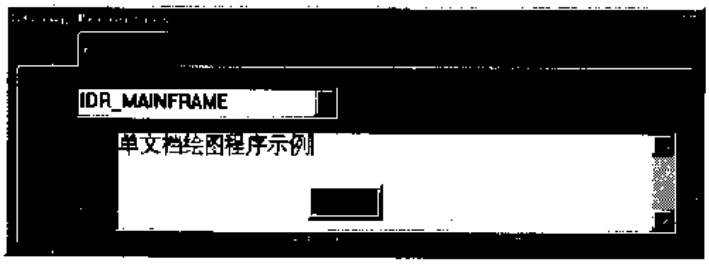



图 7-4 利用串表编辑器修改标题栏

(5) 按 Esc 键退出。

6.建立并运行程序

做完以上修改后将项目保存,然后建立并运行应用程序,具体步骤如下:

(1) 从 File 菜单中选择 Save all 命令将所有文件保存。

(2) 从 Build 菜单中选择 Build MyMFCSDI.exe 命令建立应用程序。

(3) 从 Build 菜单中选择 Execute MyMFCSDI.exe 命令运行应用程序并在其中画各种直线。

运行上面的程序后可以发现,我们已经能够在用户工作区绘制直线,但是还不能把所做结果保存到文档中,这是因为这个应用程序还未对文档进行处理。要使此程序能够处理文档,必须再进一步添加新代码,下面就来介绍文档处理的实现方法。

### 7.3.3 处理文档

对于这个应用程序,由于只处理直线,而且这些直线也仅有起点和终点,而没有其他任何信息,那么要保存直线只需保存直线的起点和终点坐标即可。当文档打开时也只需从文档中取得起点和终点信息,再在视图中把这些直线画出来即可实现文档数据的显示。下面介绍具体的实现过程。

1.在文档类中存放直线坐标

(1)这个程序的文档只包含直线,因此只需添加直线类,将直线类的声明添加在文档类

头文件 MyMFCSDIDoc.h 中，具体代码如下：

```
class CMyLine:public CObject //声明直线类 CMyLine
{
protected:
   int m_StartX;//直线起点的 X 坐标
   int m_StartY;//直线起点的 Y 坐标
   int m_EndX; //直线终点的 X 坐标
   int m_EndY; //直线终点的 Y 坐标
public:
   CMyLine(int StartX,int StartY,int EndX,int EndY);//直线的构造函数
   void DrawLine(CDC * pDC);//直线的成员函数 DrawLine 用来在视图中绘制直线
};
```

(2) 要让文档能够保存直线对象，还应向文档类的声明中添加新的数据成员，此数据成员应是刚声明的直线类对象，或指向直线类对象的指针，或指向直线类对象指针的数组。在此例中声明一个对象数组类 CObArray 的对象，该对象可以用来保存指向直线类的指针，同时为文档类添加成员函数 AddLine、GetLine 和 GetTotalLine。这 3 个函数分别用于把一个直线对象添加到 CObArray 对象中，取得 CObArray 对象中当前位置的直线和取得 CObArray 对象中的直线数。在文档类头文件 MyMFCSDIDoc.h 中添加这些成员及成员函数，代码如下：

```
protected:
   CObArray m_ObArray;
   public:
   void AddLine(int StartX,int StartY,int EndX,int EndY);
   CMyLine * GetLine(int indeX);
   int GetTotalLine();
```

(3)在文档类实现文件 MyMFCSDIDoc.Cpp 中编写新添加的成员函数的定义：

```
//直线类 CMyLine 的构造函数
CMyLine::CMyLine(int StartX,int StartY,int EndX,int EndY)
{
   m_StartX = StartX;
   m_StartY = StartY;
   m_EndX = EndX;
   m_EndY = EndY;
}
```

构造函数 CMyLine 用于初始化直线的数据成员，即起点和终点坐标。

```
//画直线函数
void CMyLine::DrawLine(CDC * pDC)
{
   pDC->MoveTo(mStartX,mStartY);
   pDC->LineTo(mEndX,mEndY);
```

```
};
```

成员函数 DrawLine 用存入当前 CMyLine 对象中的坐标画直线。

```
//添加直线对象成员函数
void CMyMFCSDIDoc::AddLine (int StartX,int StartY,int EndX,int EndY)
{
CMyLine * pMyLine = new CMyLine(StartX,StartY,EndX,EndY);
MObArray.Add(pMyLine);
};
```

成员函数 AddLine 创建新的 CMyLine 对象，并调用 CObArray 的成员函数 Add 向 mObArray 中添加 CMyLine 对象指针。

```
//返回当前直线的成员函数
CMyLine * CMyMFCSDIDoc::GetLine(int index)
{
   if(index<0||index>mObArray.GetUpperBound())
     return 0;
   else return(CMyLine *mObArray->GetAt(index));
}
```

成员函数 GetLine 返回指向 CMyLine 对象的指针。mObArray 中存放的对象指针可以通过索引值进行引用。

GetLine 首先判断参数所指明的索引值是否为有效，然后调用 CObArray 的成员函数 GetAt 得到参数指定的 CMyLine 对象并返回相应的指针。CObArray 的成员函数 GetUpperBound 返回最大的有效索引值。

```
int CMyMFCSDIDoc::GetTotalLine()
{
return mObArray.GetSize();
};
```

其中，成员函数 GetTotalLine 通过调用 CObArray 类的成员函数 GetSize 得到存放在 mObArray 中的直线数。

2. 画完直线后存放直线坐标

在消息处理函数 CMyMFCSDIView::OnLButtOnUp 中添加如下代码：

```
//释放鼠标左键后，画直线结束，应将所画直线添加到 CObArray 对象中
CMyMFCSDIDoc * pDOC = GetDocument();
pDOC->AddLine(m_pOrigin.x,m_pOrigin.y, point.x,point.y);
```

3. 修改视图的 OnDraw 成员函数

向 OnDraw 成员函数中添加以下代码，以便修改视图：

```
int index;
index = pDoc - > GetTotalLine();
while(index - - )pDoc - > GetLine(index).DrawLine(pDOC);
```

以上代码调用 GetTotalLine 函数得到存放在文档对象中的直线数，然后依次画出各条直线。

4.删除文档数据

CDocument 类的所有派生类均应重载虚函数 DeleteContents，以便新建文档或再次打开文档时清除当前文档对象的所有数据。本例在头文件 MyMFCSDIDoc.h 的文档类 CMyMFCSDIDoc 的声明中添加以下代码来实现从文档中删除已添加的全部直线：

```
virtual void DeleteContents();
```

然后，在文档类的实现文件 MyMFCSDIDoc.Cpp 中添加以下代码：

```
void CMyMFCSDIDoc::DeleteContents(){
  int index;
  index = m_ObArray.GetSize();
  while(index)
  delete m_ObArray.GetAt(index - - );
  m_ObArray.RemoveAll();
 }
```

DeleteContents 首先调用 CObArray 的成员函数 GetSize 得到存放在 m_ObArray 中的直线数。然后，调用 CObArray 的成员函数 GetAt 得到每个指针并用 Delete 操作符删除相应的 CMyLine 对象。最后，调用 CObArray 的成员函数 RemoveAll 删除所有指针。

5.处理 Undo 命令

如果要使用 Edit 菜单的 Undo 命令撤消最后画的一条直线，首先将 Undo 命令映射到相应的消息处理函数。可以使用 ClassWizard 来映射 Undo 命令消息，方法如下：

(1) 从 View 菜单中选择 ClassWizard 命令，弹出 MFC ClassWizard 对话框。

(2) 从 Class name 下拉列表框选择要处理的类名 CMyMFCSDIDoc。

(3) 在 Object IDs 列表框选择标识符 ID_EDIT_UNDO。

(4) 在 Messages 框选择 COMMAND，单击 Add Function 按钮生成消息处理函数。

(5) 通过步骤(4)，生成消息 UPDATE_COMMAND_UI 的处理函数，如图 7-5 所示。

在生成消息处理函数时，Class Wizard 在文档类的头文件 MyMFCSDIDoc.h 中把以下的函数原型声明添加到类定义中：

```
//生成的消息映射函数 protected:
  //{{AFX_MSG(CMyMFCSDIDoc)
  afx_msg void OnEditUndo();
  afx_msg void OnUpdateEditUndo(CCmdUI * pCmdUI);
  //}}AFX_MSG
```

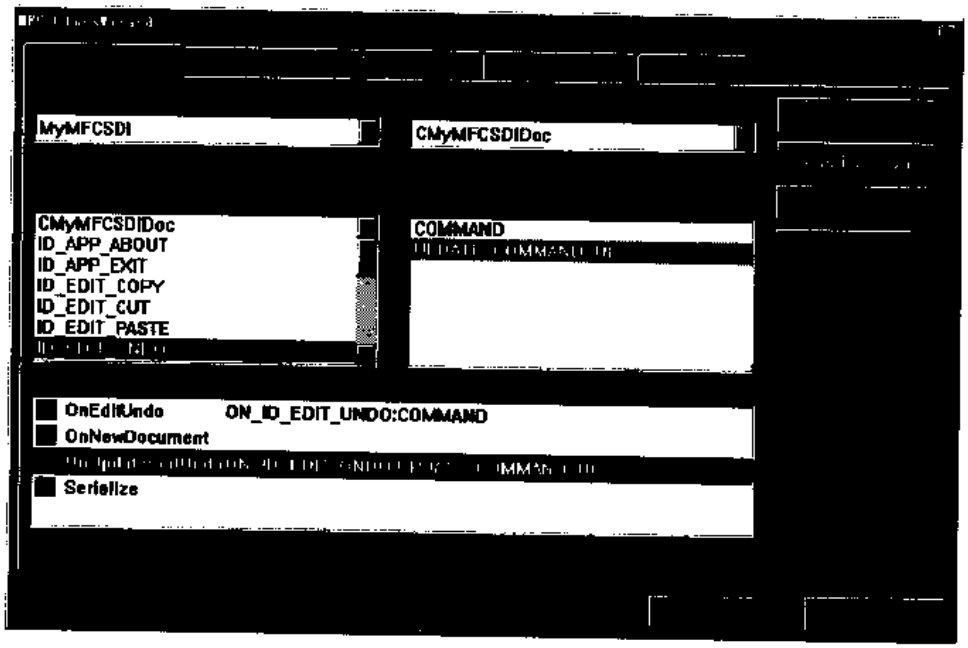



图 7-5 添加 Undo 命令消息映射

```
DECLARE_MESSAGE_MAP()
```

同时在文档类的实现文件 MyMFCSDIDoc.Cpp 中，Class Wizard 把以下消息映射宏添加到 CMyMFCSDIDoc 类的消息映射中：

```
//{{AFX_MSG_MAP(CMyMFCSDIDoc)
ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, OnUpdateEditUndo)
//}}AFX_MSG_MAP
```

此外，Class Wizard 还在文档类的实现文件 MyMFCSDIDoc.Cpp 中插入消息处理函数的基本定义：

```
void CMyMFCSDIDoc::OnEditUndo()
{
// TODO:在此处添加命令处理代码
}
void CMyMFCSDIDoc::OnUpdateEditUndo(CcmdUI * pCCmdUI)
{
// TODO:在此处添加命令更新处理代码
}
```

添加 CMyMFCSDIDoc::OnEditUndo 的代码如下：

```
int index;
index = m_ObArray.GetUpperBound();
if(index >= 0){
```

```
delete m_ObArray.GetAt(index);
m_ObArray.RemoveAt(index);
}
UpdateAllViews(NULL);
```

上述代码首先调用 CObArray 的成员函数 GetUpperBound，得到添加到文档中最后一条直线的索引值。然后调用 CObArray 的成员函数 GetAt，得到指向 CMyLine 对象的指针，并用操作符 Delete 删除相应的 CMyLine 对象。接下来调用 CObArray 的成员函数 RemoverxAt 删除指针，并调用 UpdateAllViews 更新视图窗口。最后，在 CMyMFCSDIDoc::OnUpdateEditUndo 中添加以下代码：

```
pCCmdUI->Enable(m_ObArray.GetSize());
```

完成上述代码后，就可以使用 Build 菜单的 Build MyMFCSDI.exe 和 Execute MyMFCSDI.exe 命令建立并运行应用程序，然后在应用程序的视图窗口中画各种直线。

### 7.3.4　串行化处理

所谓文档的串行化是指在文档打开时能够自动把文档中的数据转换为文档类所支持的对象，在保存文档时能够自动将文档类对象转换为文档数据格式。实现文档的串行化一般要经过以下几步：

1.添加串行化代码

重载文档基类 CDocument 的虚函数 Serialize，以使文档能够操纵由该类派生的文档类中的数据成员，在此例中就是向文档类实现文件中添加串行化代码：

```
void CMyMFCSDIDoc::Serialize(CArchive &ar)
{
if(ar.IsStoring()){
    //TODO::add Storing Code here
    m_ObArray.Serialize(ar);
  }
else{
    //TODO:add Loading Code here
  m_ObArray.Serialize(ar);
  }
}
```

2.编写 CMyLine 对象的串行化代码

由于此例的文档类 CMyMFCSDIDoc 只新增加了一个数据成员，即由 CObArray 类的对象来表示文档所容纳的数据，而 CObArray 类的实际数据是 CMyLine 类对象，所以最终应该重载 CMyLine 类的虚函数 Serialize。如果 CMyLine 类声明中没有虚函数 Serialize，则应添加该函数。总之，应编写 CMyLine 对象的串行化代码。

首先,在 CMyLine 类定义中添加两个宏 DECLARE _ SERIAL 和 IMPLEMENT _ SERIAL,并定义一个缺省的类构造函数,以便 CObArray 的成员函数 Serialize 可以将 CMyLine 类信息存放在文件内并使用类信息动态创建 CMyLine 类对象。在 CMyLine 类的声明中添加 DECLARE _ SERIAL 宏和缺省构造函数的代码如下:

```
CMyLine()
{}
DECLARE _ SERIAL(CMyLine)
```

这段代码在此例中位于文档类的头文件 CMyMFCSDIDoc.h 中。

在构造函数 CMyLine 之前添加 IMPLEMENT _ SERIAL 宏的代码如下:

```
IMPLEMENT _ SERIAL(CMyLine,CObject,1)
```

其中,CObArray 的成员函数 Serialize 将调用 CMyLine 类的 Serialize 函数进行实际的读写操作,可分为以下两步:

(1) 在 CMyLine 类中添加成员函数 Serialize 的声明:

```
virtual void Serialize(CArchive & ar);
```

(2) 在文档类的实现文件 MyMFCSDIDoc.Cpp 中添加成员函数 Serialize 的定义:

```
void CMyLine::Serialize (CArchive & ar){
  if(ar.IsStoring())
    ar < < (WORD)m _ StartX < < (WORD)m _ StartY < < (WORD)m _ EndX < < (WORD)m _
EndY;
  else
  ar > > (WORD&)m _ StartX > > (WORD&)m _ StartY > > (WORD&)m _ EndX > > (WORD&)m _
EndY;
 }
```

3.设置修改标志

打开文档并读入数据时,CDocument 将修改标志设为 FALSE。在文档数据被更改后,必须调用 CDocument 的成员函数 SetModifiedFlag 将修改标志设为 TRUE。

在 MyMFCSDI 程序中,只有添加直线或执行 Undo 命令时才会更改文档数据。因此,需分别在 AddLine 函数和 OnEditUndo 函数的最后添加以下代码:

```
SetModifiedFlag();
```

4.建立并运行程序

执行 Bulid 菜单的 Build MyMFCSDI.exe 和 Execute MyMFCSDI.exe 命令可以建立并运行应用程序,然后可在应用程序的视图窗口画各种直线。选择 File 菜单的 Save As 或 Save 命令可以保存所画的直线;选择 File 菜单的 Open 命令可以打开文件。

### 7.3.5 SDI应用程序编程思路

编写SDI应用程序首先应该正确理解本章前面几节所介绍的几个基本概念:主边框窗口、视图、文档和文档的串行化。本节结合前面的示例程序再对这几个概念作进一步阐述,以帮助读者加深对基本概念的理解和应用。

1.主边框窗口

在SDI应用程序中,主边框窗口同时也是文档的边框窗口,此窗口类CMainFrame的声明在MainFrm.h中。MFC类库中有其缺省实现,如果要修改它,需要重载其虚函数PreCreateWindow(CREATESTRUCT& cs),此虚函数的实现代码在文件MainFrm.CPP中。

对于需要文档串行化支持的应用程序,应在类声明的构造函数之后添加宏DECLARE_DYNCREATE(CMainFrame),因为文档串行化需要动态建立相应的类对象。

边框窗口类的实现文件MainFrm.CPP中有一个消息映射,即ON_WM_CREATE(),它处理函数OnCreate(LPCREATESTRUCT lpCreateStruct),该实现文件也定义了建立窗口的函数PreCreateWindow(CREATESTRUCT& cs)。在定制主边框窗口风格及窗口建立时,执行的是缺省操作,应分别修改这两个函数。在以上例子中对这两个函数未做任何修改。

2.视图

视图是应用程序的工作区,用户的各种输出(如文本的显示、图形的绘制等)都将由视图来表现,文档中的数据也要显示在视图中,因此定制视图是编写SDI应用程序非常重要的环节。

由于视图也是窗口,因而它们也具有自己的风格。视图风格的设置通过重载虚函数PreCreateWindow(CREATESTRUCT& cs)来实现。

打开文档后文档数据的显示通过视图类成员函数OnDraw()来实现,用户通过键盘、鼠标、菜单命令和工具等接口编辑文本和绘制图形的操作也由视图类来完成。文档的打印、打印设置及打印预览等操作可以看作是在打印机上显示视图,因而其功能的实现也应由视图类来完成。这些工作需要利用Class Wizard来映射键盘、鼠标、菜单命令和工具消息,并实现相应的消息处理函数。在上面的例子中,对鼠标左键的按下与释放及鼠标的移动建立了消息映射并实现了相应的处理代码,而对其他消息的处理则使用CView类的相应成员函数作为缺省消息处理函数。

如果要使应用程序具有动态建立视图类对象的功能,则应分别通过DECLARE_DYNCREATE(CMyMFCSDIView)宏和IMPLEMENT_DYNCREATE(CMyMFCSDIView, CView)宏来实现。

3.文档

文档是基于文档的应用程序涉及的最重要的概念,也是应用程序代码编写工作量较大的一部分。文档是应用程序处理数据的集合,文档中的数据要直观地显示给用户,例如,对于图形数据,应将图形绘制在用户窗口中,而不是将表示图形几何特性的二进制数据显示给用户。同时,用户通过各种命令产生的各种应用数据也应能转换为文档中的二进制数据保

存。此外,两者之间的转换应是自动进行的,而不是需要开发大量的代码来实现这些转换。MFC 类库提供了一个文档类 CDocument 可帮助用户完成上述转换功能。

文档要包含的各种数据类型都应在文档类中声明新的数据成员,并由这些数据成员表示相应的应用数据。例如,在上面的例子中,对象数组类 CObArray 就是文档类的一个数据成员。对象数组类 CObArray 是数组类的一种,它支持 CObject 对象指针,与数组非常相似,但是它可以随着所包含指针元素的多少而动态地缩小或扩大,以适应元素数量的变化。

文档中所包含的数据及其直观显示由文档串行化功能来实现。文档串行化自动实现二进制数据与具体数据形式之间的自动转换,即当打开文档时将二进制数据转换为相应类的数据成员,而保存文档时则将类的数据成员转换为二进制数据。要实现文档的串行化,需要重载 CDocument 类的虚函数 Serialize,同时对于文档要处理的每种数据类都应重载虚函数 Serialize。

要实现文档的串行化,文档类及文档要包含的数据类的声明部分都应有缺省构造函数和 DECLARE_SERIAL 宏,并在其实现部分的构造函数之前包括 IMPLEMENT_SERIAL 宏。

4.应用程序

Windows 环境下的每一个应用程序也都是一个类,一般称它们为应用程序类。应用程序类继承自 CWapp 类,它们是应用程序在运行时的入口。它的主要作用是维护全局性工作,如应用程序菜单的维护、应用程序进程的建立以及命令行信息的分析等各方面的工作。

### 7.3.6 完整程序清单

这一节将详细介绍如何使用文档类、视图类,并实现一个 SDI 应用程序。在这个应用程序的开发过程中,我们主要修改了由 MFC AppWizard 向导产生的应用程序边框的 4 个文件,它们分别是视图类声明文件(MyMFCSDIView.h)、视图类实现文件(MyMFCSDIView.cpp)、文档类声明文件(MyMFCSDIdoc.h)和文档类实现文件(MyMFCSDIdoc.cpp)。只要我们理解了此例子的原理和步骤,就可以开发出具有更多丰富功能的复杂的应用程序。为了对整个程序结构有一个整体印象,这里列出了此 SDI 应用程序所涉及的源文件和头文件,虽然次源文件也可以用代码形式列出,但由于用代码方式修改它们不如直接利用相关的资源编辑器来创建和修改它们方便,因而没有列出。

**程序清单 7-1:视图类声明文件 MyMFCSDIView.h**

```
// MyMFCSDIView.h :CMyMFCSDIView 类的接口
//
/////////////////////////////////////////

#if ! defined(AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_B31A29E86F7F__
INCLUDED_)
#define AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_B31A29E86F7F__IN-
CLUDED_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CMyMFCSDIView : public CView
{

protected:  //新增加保护数据成员
      int m_Draw;  //表示是否正在拖动鼠标
      HCURSOR m_HCursor;  //存放光标句柄
      CPoint m_pOld;  //存放光标的当前位置
      CPoint m_pOrigin;  //存放鼠标左键按下时光标所在的位置(画线的起点)

protected:  // 仅串行化创建
      CMyMFCSDIView();
      DECLARE_DYNCREATE(CMyMFCSDIView);

//特性
public:
      CMyMFCSDIDoc * GetDocument();

//操作
public:

//重载
      // ClassWizard 生成虚函数重载
      //{{AFX_VIRTUAL(CMyMFCSDIView)
      public:
      virtual void OnDraw(CDC * pDC);  // overridden to draw this view
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      protected:
      virtual BOOL OnPreparePrinting(CPrintInfo * pInfo);
      virtual void OnBeginPrinting(CDC * pDC, CPrintInfo * pInfo);
      virtual void OnEndPrinting(CDC * pDC, CPrintInfo * pInfo);
      //}}AFX_VIRTUAL

// 具体实现
public:
      virtual CMyMFCSDIView();
```

```
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

//生成消息映射函数
protected:
    //{{AFX_MSG(CMyMFCSDIView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // MyMFCSDIView.cpp 文件的调试版本
inline CMyMFCSDIDoc* CMyMFCSDIView::GetDocument()
   { return (CMyMFCSDIDoc*)m_pDocument; }
#endif

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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将在此处添加附加的声明

#endif  // !defined(AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_
B31A29E86F7F__INCLUDED_)
```

**程序清单 7-2:视图类实现文件 MyMFCSDIView.cpp**

```
// MyMFCSDIView.cpp : CMyMFCSDIView 类的实现
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MyMFCSDIDoc.h"
#include "MyMFCSDIView.h"
```

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

/////////////////////////////////////////
// CMyMFCSDIView

IMPLEMENT_DYNCREATE(CMyMFCSDIView, CView)

BEGIN_MESSAGE_MAP(CMyMFCSDIView, CView)
    //{{AFX_MSG_MAP(CMyMFCSDIView)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    //}}AFX_MSG_MAP
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////
// CMyMFCSDIView 的构造函数与析构函数

CMyMFCSDIView::CMyMFCSDIView()
{
    // 在此处添加构造函数代码
    m_Draw=0;  //初始时,未拖动鼠标
    m_HCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);


}

CMyMFCSDIView::CMyMFCSDIView()
{
}

BOOL CMyMFCSDIView::PreCreateWindow(CREATESTRUCT& cs)
{
```

```
    // 在此处修改 Window 类或其样式
    // CREATESTRUCT cs
    cs.lpszClass = AfxRegisterWndClass(
        CS_HREDRAW|CS_VREDRAW,  //类风格
        0,//不用光标
        (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
        0);//没有图标

    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////
// CMyMFCSDIView 绘制代码

void CMyMFCSDIView::OnDraw(CDC * pDC)
{
    CMyMFCSDIDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    int index;
    index = pDoc->GetTotalLine();
    while(index--) pDoc->GetLine(index)->DrawLine(pDC);

    // 在此处添加绘制代码的初始数据
}

/////////////////////////////////////////
// CMyMFCSDIView 打印代码

BOOL CMyMFCSDIView::OnPreparePrinting(CPrintInfo * pInfo)
{
    // 缺省打印准备
    return DoPreparePrinting(pInfo);
}

void CMyMFCSDIView::OnBeginPrinting(CDC * /* pDC */, CPrintInfo * /* pInfo
*/)
{
    // 在打印前添加额外的初始代码
}
```

```
void CMyMFCSDIView::OnEndPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    // 打印后进行清理
}

/////////////////////////////////////
// CMyMFCSDIView 诊断代码

#ifdef _DEBUG
void CMyMFCSDIView::AssertValid() const
{
    CView::AssertValid();
}

void CMyMFCSDIView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMyMFCSDIDoc * CMyMFCSDIView::GetDocument()
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyMFCSDIDoc)));
    return (CMyMFCSDIDoc *)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////
// CMyMFCSDIView 消息处理程序

void CMyMFCSDIView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 在此处添加消息处理程序代码或者调用缺省消息处理程序

    CView::OnLButtonDown(nFlags, point);

    m_pOld = point; //存放鼠标光标的当前位置
    m_pOrigin = point; //存入画线的起始位置
    SetCapture(); //捕获鼠标
    m_Draw = 1; //表示正在拖动鼠标
    RECT rect;
    GetClientRect(&rect); //获取视图窗口区客户的坐标
```

```
        ClientToScreen(&rect); //转换客户区坐标为屏幕坐标
        ClipCursor(&rect); //将光标限定在视图窗口客户区内,接着调用 CWnd 的成员函数
        ClipCursor(&rect); //SetCapture 使随后的鼠标消息都被送往视图窗口

}

void CMyMFCSDIView::OnMouseMove(UINT nFlags, CPoint point)
{
        // 在此处添加消息处理程序代码或者调用缺省消息处理程序

        CView::OnMouseMove(nFlags, point);

        SetCursor(m_HCursor);//设置自定义光标
        if(m_Draw){ //正在拖动鼠标
                CClientDC dc(this); //创建设备文本对象
                dc.SetROP2(R2_NOT); //设置绘图模式,以屏幕反转色为绘图颜色
                dc.MoveTo(m_pOrigin);//删除前一个 WM_MOUSEMOVE 画的直线
                dc.LineTo(m_pOld);
                dc.MoveTo(m_pOrigin);//以起始位置到当前位置画一条新的直线
                dc.LineTo(pcint);
                m_pOld = point;//存放鼠标光标的当前位置
                }

}

void CMyMFCSDIView::OnLButtonUp(UINT nFlags, CPoint point)
{
        // 在此处添加消息处理程序代码或调用缺省消息处理程序
        CView::OnLButtonUp(nFlags, point);

        if(m_Draw){
                m_Draw = 0;//表示鼠标已释放
                ReleaseCapture();//释放鼠标并恢复正常输入
                ClipCursor(NULL);//使光标可以移到屏幕任意位置
                }

        //释放鼠标左键后,画直线结束,应将所画直线添加到 CObArray 对象中
        CMyMFCSDIDoc * pDOC = GetDocument();
        pDOC -> AddLine(m_pOrigin.x,m_pOrigin.y, point.x,point.y);

}
```

**程序清单 7-3:文档类声明文件 MyMFCSDIdoc.h**

```
// MyMFCSDIDoc.h : CMyMFCSDIDoc 类的接口
//
/////////////////////////////////////////

#if ! defined(AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__
INCLUDED_)
#define AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__IN-
CLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CMyLine:public CObject //声明直线类 CMyLine
{
protected:
      int m_StartX;//直线起点的 X 坐标
      int m_StartY;//直线起点的 Y 坐标
      int m_EndX; //直线终点的 X 坐标
      int m_EndY; //直线终点的 Y 坐标
public:
      CMyLine(int StartX,int StartY,int EndX,int EndY);//直线的构造函数
      void DrawLine(CDC * pDC);//直线的成员函数 DrawLine 用来在视图中绘制直线
      CMyLine() { }
      DECLARE_SERIAL(CMyLine)

      virtual void Serialize(CArchive & ar);

};

class CMyMFCSDIDoc : public CDocument
{
protected: // 仅由串行化创建
      CMyMFCSDIDoc();
      DECLARE_DYNCREATE(CMyMFCSDIDoc)

// 特性
public:
```

```
protected:
        CObArray m_ObArray;
        public:
        void AddLine(int StartX,int StartY,int EndX,int EndY);
        CMyLine * GetLine(int indeX);
        int GetTotalLine();

        virtual void DeleteContents();

// 操作
public:

// 重载
        // ClassWizard 生成虚函数重载
        //{{AFX_VIRTUAL(CMyMFCSDIDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void Serialize(CArchive& ar);
        //}}AFX_VIRTUAL

// 具体实现
public:
        virtual CMyMFCSDIDoc();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// 生成消息映射函数
protected:
        //{{AFX_MSG(CMyMFCSDIDoc)
        afx_msg void OnEditUndo();
        afx_msg void OnUpdateEditUndo(CCmdUI * pCmdUI);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将在此处添加附加的声明

#endif // !defined(AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
```

**程序清单 7-4:文档类实现文件 MyMFCSDIDoc.cpp**

```
// MyMFCSDIDoc.cpp : 实现 CMyMFCSDIDoc 类
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MyMFCSDIDoc.h"

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

/////////////////////////////////////////
// CMyMFCSDIDoc

IMPLEMENT_DYNCREATE(CMyMFCSDIDoc, CDocument)

BEGIN_MESSAGE_MAP(CMyMFCSDIDoc, CDocument)
    //{{AFX_MSG_MAP(CMyMFCSDIDoc)
    ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
    ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, OnUpdateEditUndo)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

IMPLEMENT_SERIAL(CMyLine,CObject,1)

//直线类 CMyLine 的构造函数
CMyLine::CMyLine(int StartX,int StartY,int EndX,int EndY){
    m_StartX = StartX;
    m_StartY = StartY;
```

```
        m_EndX = EndX;
        m_EndY = EndY;
    }

    //画直线函数
    void CMyLine::DrawLine(CDC * pDC){
        pDC->MoveTo(m_StartX,m_StartY);
        pDC->LineTo(m_EndX,m_EndY);
    };

    //添加直线对象成员函数
    void CMyMFCSDIDoc::AddLine (int StartX,int StartY,int EndX,int EndY)
    {
        CMyLine * pMyLine = new CMyLine(StartX,StartY,EndX,EndY);
        m_ObArray.Add(pMyLine);
        SetModifiedFlag();
    };

    //返回当前直线的成员函数
    CMyLine * CMyMFCSDIDoc::GetLine(int index)
    {
        if(index<0||index>m_ObArray.GetUpperBound())
            return 0;
        else return(CMyLine *) m_ObArray.GetAt(index);
    }

    int CMyMFCSDIDoc::GetTotalLine()
    {
        return m_ObArray.GetSize();
    }

    void CMyLine::Serialize (CArchive & ar){
        if(ar.IsStoring())
            ar<<(WORD)m_StartX<<(WORD)m_StartY<<(WORD)m_EndX<<
(WORD)m_EndY;
        else
            ar>>(WORD&)m_StartX>>(WORD&)m_StartY>>(WORD&)m_EndX>>
(WORD&)m_EndY;
    }

    ////////////////////////////////////////
```

```
// CMyMFCSDIDoc 构造函数/析构函数

CMyMFCSDIDoc::CMyMFCSDIDoc()
{
      // 添加构造函数

}

CMyMFCSDIDoc::CMyMFCSDIDoc()
{
}

BOOL CMyMFCSDIDoc::OnNewDocument()
{
      if (! CDocument::OnNewDocument())
            return FALSE;

      // 添加重新初始化代码
      // 文档都将重用此文档

      return TRUE;
}

void CMyMFCSDIDoc::DeleteContents(){
      int index;
      index = m_ObArray.GetSize();
      while(index)
      delete m_ObArray.GetAt(--index);
      m_ObArray.RemoveAll();
}

////////////////////////////////////////
// CMyMFCSDIDoc 串行化

void CMyMFCSDIDoc::Serialize(CArchive &ar)
{
if(ar.IsStoring()){
        //添加存储代码
        m_ObArray.Serialize(ar);
    }
else{
```

```
        //添加加载代码
        m_ObArray.Serialize(ar);
    }
}

/////////////////////////////////////////
// CMyMFCSDIDoc 诊断

#ifdef _DEBUG
void CMyMFCSDIDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMyMFCSDIDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////
// CMyMFCSDIDoc 命令

void CMyMFCSDIDoc::OnEditUndo()
{
    // 添加用户的命令处理程序代码
    int index;
    index = m_ObArray.GetUpperBound();
    if(index >= 0){
        delete m_ObArray.GetAt(index);
        m_ObArray.RemoveAt(index);
        }
    UpdateAllViews(NULL);
    SetModifiedFlag();

}

void CMyMFCSDIDoc::OnUpdateEditUndo(CCmdUI * pCCmdUI)
{
    // 添加用户的命令更新处理程序代码
    pCCmdUI->Enable(m_ObArray.GetSize());
```

**程序清单7-5:应用程序类声明文件MyMFCSDI.h**

```
// MyMFCSDI.h : MyMFCSDI 应用程序的主头文件
//

#if ! defined(AFX_MYMFCSDI_H__4E2E9AB9_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
#define AFX_MYMFCSDI_H__4E2E9AB9_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols

/////////////////////////////////////////////
// CMyMFCSDIApp:
//

class CMyMFCSDIApp : public CWinApp
{
public:
    CMyMFCSDIApp();

// 重载
    // ClassWizard
    //{{AFX_VIRTUAL(CMyMFCSDIApp)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// 具体实现
    //{{AFX_MSG(CMyMFCSDIApp)
    afx_msg void OnAppAbout();
```

```
            // ClassWizard 在此处添加和删除成员函数
        //{{AFX_MSG
        DECLARE_MESSAGE_MAP()
};


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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将在此处插入附加的声明

#endif // !defined(AFX_MYMFCSDI_H__4E2E9A39_6799_11D2_8DE6_B31A29E86F7F
__INCLUDED_)
```

**程序清单 7-6:应用程序类实现文件 MyMFCSDI.cpp**

```
// MyMFCSDI.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MainFrm.h"
#include "MyMFCSDIDoc.h"
#include "MyMFCSDIView.h"

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

/////////////////////////////////////////
// CMyMFCSDIApp

BEGIN_MESSAGE_MAP(CMyMFCSDIApp, CWinApp)
    //{{AFX_MSG_MAP(CMyMFCSDIApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // ClassWizard 在此处添加和删除映射宏
    //}}AFX_MSG_MAP
```

```
    // 基于文档命令的标准文件
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // 标准打印设置命令
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////
// CMyMFCSDIApp 构造函数

CMyMFCSDIApp::CMyMFCSDIApp()
{
    // 在此处添加构造函数代码
}

/////////////////////////////////////////
// CMyMFCSDIApp 对象代码

CMyMFCSDIApp theApp;

/////////////////////////////////////////
// CMyMFCSDIApp 初始化

BOOL CMyMFCSDIApp::InitInstance()
{
    AfxEnableControlContainer();

    // 标准初始化

#ifdef _AFXDLL
    Enable3dControls(); // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif

    // 修改注册键标
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings(); // 加载标准 INT 文件选项(包括 MRU)

    // 注册应用程序的文档模板
```

```
    // 文档模板在文档、边框窗口和视图间起连接作用

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CMyMFCSDIDoc),
        RUNTIME_CLASS(CMainFrame), //主 SDI 边框窗口
        RUNTIME_CLASS(CMyMFCSDIView));
    AddDocTemplate(pDocTemplate);

    // 分析命令行
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // 分析命令行中指定的命令
    if (! ProcessShellCommand(cmdInfo))
        return FALSE;

    //只需初始化一个窗口,并显示和更新它
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}


/////////////////////////////////////////
// CAboutDlg 对话

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard 生成虚函数重载
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
```

```
        virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
        //}}AFX_VIRTUAL

// 具体实现
protected:
        //{{AFX_MSG(CAboutDlg)
                // 无消息处理程序
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange * pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
        //{{AFX_MSG_MAP(CAboutDlg)
                // 无消息处理程序
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App 命令运行对话
void CMyMFCSDIApp::OnAppAbout()
{
        CAboutDlg aboutDlg;
        aboutDlg.DoModal();
}

/////////////////////////////////////////////
// CMyMFCSDIApp 消息处理程序
```

**程序清单 7-7:主边框窗口类声明文件 MainFrm.h**

```
// MainFrm.h : CMainFrame 类的接口
//
/////////////////////////////////////////

#if ! defined(AFX_MAINFRM_H__4E2E9ABD_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
#define AFX_MAINFRM_H__4E2E9ABD_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CMainFrame : public CFrameWnd
{

protected: // 仅由串行化创建
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)

// 特性
public:

// 操作
public:

// 重载
    // ClassWizard 生成虚函数重载
    //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL

// 具体实现
public:
    virtual CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected: // 嵌入成员的控制条
      CStatusBar m_wndStatusBar;
      CToolBar m_wndToolBar;

// 生成消息映射函数
protected:
      //{{AFX_MSG(CMainFrame)
      afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
            // ClassWizard 将在此处添加和删除成员函数
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将在此处插入附加的声明

#endif // !defined(AFX_MAINFRM_H__4E2E9ABD_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
```

**程序清单 7-8:主边框窗口实现文件 MainFrm.cpp**

```
// MainFrm.cpp : 实现 CMainFrame 类
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MainFrm.h"

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

/////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
```

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
        // ClassWizard 将在此处添加和删除映射宏
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR, // 状态行指示器
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
//////////////////////////////////////
// CMainFrame 构造函数与析构函数

CMainFrame::CMainFrame()
{
    // 添加成员初始化代码

}

CMainFrame::CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (! m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISI-
    BLE | CBRS_TOP|CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE
    _DYNAMIC) || ! m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1; // fail to create
    }
```

```
    if (! m_wndStatusBar.Create(this) ||
        ! m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar \n");
        return -1; // fail to create
    }

    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( ! CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // 修改 Window 类或其样式
    // CREATESTRUCT cs

    return TRUE;
}
/////////////////////////////////////////
// CMainFrame 诊断

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif //_DEBUG
/////////////////////////////////////////
// CMainFrame 消息处理程序
```

# 第 8 章

# 绘图程序的实现

在 Windows 应用程序的开发中经常需要绘制各种各样的图形，这些图形既包括格式化的文本，又包括各类几何形状，还包括以点形式描绘的点位图。我们统一地把它们称为图形元素。Windows 系统提供了实现绘制这些图形元素的标准函数集，这些函数集又称为 Windows 图形设备接口。

在 Windows 系统中所有的图形输出都在被称为设备描述表的虚拟用户工作中绘制。MFC 类库既提供了不同类型的设备描述表的类，又提供了各种各样的绘制工具类，如点位图、笔刷、调色板和画笔等。

本章先介绍生成应用程序的框架，然后把后续各节所介绍的内容添加到应用程序中，最后编译并运行改写后的程序来检验各节介绍的函数的表现效果。

## 8.1 生成初始的应用程序框架

利用前面介绍的 MFC AppWizard，首先生成应用程序 Graph 的基本框架，各步选择如下：

(1) 项目名称为 Graph，项目类型选择为 MFC AppWizard(exe)。

(2) 应用程序类型为 SDI，其他使用缺省设置。

(3) 应用程序数据库支持选择 None，即不需数据库支持。

(4) 复合文档支持选择 None，即不支持任何复合文档。

(5) 选择应用程序窗口包括的界面元素，全部使用默认选项。

(6) 在应用程序风格设置对话框中选择 MFC 标准风格，其他选项使用缺省设置。

经过以上各步设置，MFC AppWizard 即自动生成应用程序 Graph 的基本框架。

## 8.2 基 本 概 念

编写 Windows 图形应用程序必须正确理解几个基本概念。例如设备描述表、坐标系统和绘图工具等。

### 8.2.1 设备描述表

设备描述表是 Windows 应用程序、设备驱动程序和输出设备(如打印机、绘图仪、显示器等)之间的桥梁，它定义了设备、画图工具和画图信息。图 8-1 显示了 Windows 应用程序经过设备描述表到设备驱动程序并最终到输出设备的信息流。

MFC 类库提供了与设备描述表相关的 5 个类。每一个类都封装了代表 Windows 设备描述表的句柄。在 MFC 类库中设备描述表也是一个 Windows 对象，它包括了设备的绘制属性，如显示或打印。调用绘图函数都是通过设备描述表对象来实现的。其他的设备描述表类都继承自 CDC 类，它们封装了特定的设备描述表功能，包括对 Windows 图元文件的支持。这 5 个设备描述表类及其功能如下：

• CDC：设备描述表类的根类，利用它可以访问整个显示设备和非显示设备(如打印机)。

• CPaintDC:窗口的 OnPaint 函数使用的显示描述表。该类的构造函数自动调用 BeginPaint函数,析构函数自动调用 EndPaint 函数。



图 8-1　由应用程序到输出设备的信息流

• CClientDC:窗口客户区的显示描述表,一般用来实时映射鼠标事件,例如绘制。

• CWindowDC:整个应用程序窗口,既包括客户区,又包括非客户区。

• CMetaFileDC:表示 Windows 图元文件的显示描述表。Windows 图元文件包含了一系列 GDI 绘图命令,这些命令用于重新建立相应的映像。

### 8.2.2　坐标系统

在 Windows 环境下,所有图形及文字的输出都基于坐标系统,图形坐标系统与基于文本的函数所使用的坐标系统是一致的。缺省的坐标系统以用户区域的左上角为原点,向右的每一个像素表示沿正 X 轴的一个单位。向下的每一个像素表示沿正 Y 轴的一个单位。

通过改变映射模式和显示原点,可修改这个坐标系统。映射模式定义坐标系统的单位。缺省模式为 MM _ TEXT,即每个单位为一个像素。也可指定映射模式以英寸或毫米为单位。

### 8.2.3　绘图工具

绘图工具包括画线的画笔、填充内部的画刷以及书写正文的字体工具。所有这些工具的使用都要经过三个步骤,即建立工具、选择工具和使用完后删除相应的工具。

1.调用 CDC 成员函数

调用 CDC 成员函数 SelectStockObject 可以选择库存绘图工具,该函数原型为:

```
virtual CGdiObject * SelectStockObject( int nIndex );
```

调用成功,返回指向 CGdiObject 对象的指针,否则,返回 NULL。nIndex 是所选择的库存对象宏代码。表 8-1 列出了库存画笔和画刷的宏代码。

表 8-1　画笔和画刷的宏代码及含义

| 宏代码 | 库存对象 |
|---|---|
| BLACK _ BRUSH | 黑色画刷 |
| DKGRAY _ BRUSH | 暗灰色画刷 |
| GRAY _ BRUSH | 灰色画刷 |

续表

| 宏代码 | 库存对象 |
| --- | --- |
| HOLLOW _ BRUSH | 透明窗口画刷 |
| LTGRAY _ BRUSH | 浅灰色画刷 |
| NULL _ BRUSH | 空画刷 |
| WHITE _ BRUSH | 白色画刷 |
| BLACK _ PEN | 黑色画笔 |
| NULL _ PEN | 空画笔 |
| WHITE _ PEN | 白色画笔 |

2. 创建定制的画笔工具

通过调用构造函数来创建定制的画笔工具。程序清单 8-1 说明了画笔对象的使用过程。

**程序清单 8-1:在程序中使用画笔**

```
void CGraphView::OnDraw(CDC * pDC)|
CGraphDoc * pDOC = GetDocument();
ASSERT _ VALID(pDOC);
//TODO:在此处添加重新绘制视图窗口的代码
CPen NewPen; //声明画笔对象
CPen * pOldPen;//保存原先画笔的指针
NewPen.CreatePen(PS _ SOLID,2,RGB(0,255,0));//初始化实线 2 像素宽的绿色画笔
pOldPen = pDC - > SelectObject(&newPen);
//以下是使用画笔代码(略)
                .
                .
                .
//恢复原先的画笔对象
pDC - > Selectobject(POldPen);
|
```

3. 创建定制的画刷工具

画刷使用过程与画笔类似,程序清单 8-2 说明了在绘图过程使用画刷对象的方法:

**程序清单 8-2:在程序中使用画刷**

```
void CGraphView::OnDraw(CDC * pDC){
CGraphDoc * pDOC = GetDocment();
ASSERT_VALID(pDOC);
//在此处添加新的绘制视图窗口的代码
CBrush NewBrush;//声明画刷对象
Cbrush * pOldBrush,//保存原先画刷的指针
NewBrush.CreateSolidBrush(PS_SOLID,21,RGB(0,0.255));//建立画刷
//将画刷选入设备文本对象
POld_Brush = pDC - > SelectObject(&NewBrush);
//使用画刷,代码略
                .
                .
                .
//恢复原先的画刷对象
pDC - > Selectobject(PO'dBrush);
}
```

4.颜色

颜色的类型是 COLORREF,表示 RGB 值,是一个 32 位的整数,包含红、绿、蓝三个颜色域。RGB()的定义为:

```
COLORREF RGB(BYTE bRed,BYTE bGreen,BYTE bBLue);
```

### 8.2.4　绘图函数

1.用某种颜色画点

用某种颜色画点,实际上就是设置像素。可用 CDC 的成员函数 SetPixel 来设置任何指定像素点的颜色,函数原型为:

```
COLORREF SetPixel( int x, int y, COLORREF crColor );
```

该函数将设置由参数 x 和 y 指定的坐标处的像素点,所用颜色尽量接近 crColor 所指定的颜色。要设置的点必须位于用户的视图区内,如果不在此区域内,该函数则被忽略。该函数的另一形式为:

```
COLORREF SetPixel( POINT point, COLORREF crColor );
```

功能与上一个函数相同,只不过是用结构 POINT 型变量来表示要设置点的 x 坐标和 y 坐标。

2.设置当前位置

画线时,有些函数往往从当前位置开始画起,其实当前位置可以被设置成任何位置,这

是通过调用 CDC 的成员函数 MoveTo 实现的。MoveTo 函数的原型为：

```
CPoint MoveTo( int x, int y );
```

其中，x 和 y 是要设置为当前位置的两个坐标。

3. 画简单线

画简单线可以使用 CDC 的成员函数 LineTo 实现，LineTo()函数使用当前选择的画笔绘制线条，函数原型为：

```
BOOL LineTo( int x, int y );
BOOL LineTo( POINT point );
```

从当前位置到目标位置用缺省的画笔画直线，此直线不包括起点本身，如果没有错误，终点将成为新的当前位置。

4. 画弧

使用 CDC 的成员函数 Arc 和 ArcTo，可以用缺省的画笔画一段不填充的椭圆弧。Arc 函数的原型为：

```
BOOL Arc( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
BOOL Arc( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

这两个函数画弧成功返回非 0，否则返回 0。函数中各参数的含义如下：

x1：包围弧的矩形的左上角 x 坐标(单位是逻辑单位)。

y1：包围弧的矩形的左上角 y 坐标(单位是逻辑单位)。

x2：包围弧的矩形的右下角 x 坐标(单位是逻辑单位)。

y2：包围弧的矩形的右下角 y 坐标(单位是逻辑单位)。

x3：弧的起点 x 坐标(单位是逻辑单位)，该点不必精确地位于弧上。

y3：弧的起点 y 坐标(单位是逻辑单位)，该点不必精确地位于弧上。

x4：弧的终点 x 坐标(单位是逻辑单位)，该点不必精确地位于弧上。

y4：弧的起点 y 坐标(单位是逻辑单位)，该点不必精确地位于弧上。

lpRect：表示围绕弧的矩形，它可以是 LPRECT 或 CRect 对象。

PtStart：表示弧的起点的 CPoint 或 POINT 对象。该点不必精确地位于弧上。

PtEnd：表示弧的终点的 CPoint 或 POINT 对象。该点不必精确地位于弧上。

该函数绘制的一段椭圆弧，是给定矩形所包围的椭圆的一部分。弧的实际起点和终点分别是矩形中心与(x3,y3)和(x4,y4)的连线与椭圆的交点。画弧方向为逆时针旋转。由于弧不是封闭图形，所以不进行填充。矩形的宽和高必须介于 2 到 32767 个逻辑单位之间。如果起点与终点相同则画出整个椭圆。

另外，画弧还可以使用 CDC 的成员函数 ArcTo，其函数原型为：

```
BOOL ArcTo( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
BOOL ArcTo( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

该函数中各参数的含义与功能同 Arc 函数中的相似，但是 Arc 执行后不改变当前位置，而 ArcTo 则改变当前位置。如果画弧成功当前位置则是弧的终点，否则不变。

5.画矩形

CDC 的两个成员函数 Rectangle 和 RoundRect 分别用于画矩形和圆角矩形。矩形内部用所选定的笔刷填充，矩形的边界用所选定的画笔画出。

函数 Rectangle 的原型为：

```
BOOL Rectangle( int x1, int y1, int x2, int y2 );
```

此函数成功调用后返回非 0 值，否则返回 0。函数中各参数的含义如下：

x1：指定矩形的左上角逻辑 x 坐标。

y1：指定矩形的左上角逻辑 y 坐标。

x2：指定矩形的右下角逻辑 x 坐标。

y2：指定矩形的右下角逻辑 y 坐标。

函数 Rectangle 的另一原形为：

```
BOOL Rectangle( LPCRECT lpRect )
```

功能与前面函数相同，只不过用指向 LPCRECT 结构的指针来表示矩形的四个角。

函数 RoundRect 的原型为：

```
BOOL RoundRect( int x1, int y1, int x2, int y2, int x3, int y3 )
BOOL RoundRect( LPCRECT lpRect, POINT point );
```

这两个函数分别与 Rectangle 函数的形式相似，只不过所画矩形为圆角矩形。

6.画椭圆和圆

使用 CDC 的成员函数 Ellipse，可以用当前画笔绘制一个用当前画刷填充的椭圆或圆。函数原型为：

```
BOOL Ellipse( int x1, int y1, int x2, int y2 );
BOOL Ellipse( LPCRECT lpRect );
```

这两个函数画椭圆成功后返回非 0，否则返回 0。所画椭圆高度为 y2 - y1，宽度为 x2 - x1。如果宽度或高度为 0，则不能画椭圆。

7.画饼图

饼图是一条弧和从弧的两个端点到中心的连线组成的图形。CDC 的成员函数 Pie 可用于画饼图，函数原型为：

```
BOOL Pie( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
BOOL Pie( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

这两个函数通过画椭圆弧来画饼形的扇形，椭圆弧的中心和两端点有连线连接，弧的中心是参数 x1，y1，x2，y2 所确定的矩形的中心，弧的起点和终点由参数 x3，y3，x4 和 y4 指定。

弧由选定的画笔逆时针画出。两条附加线从每个端点到弧中心。圆饼形区域用所选定的笔刷填充。

如果 x3 = y3 且 x4 = y4,结果便是只有一条单线的椭圆,线从椭圆中心到(x3,y3)。矩形的宽和高不得超过 32767。

8.画多边形

CDC 的成员函数 PolygOn 可用于画多边形。函数原型为:

```
BOOL Polygon( LPPOINT lpPoints, int nCount );
```

此函数连接两个以上顶点之间的直线来画出多边形。多边形使用当前的多边形填充模式来填充。如果多边形可自动闭合,则绘制连接最后的顶点与第一个顶点之间的直线。画多边形成功则返回非 0,否则返回 0。

参数 lpPoints 为多边形顶点数组地址,nCount 为数组中的顶点数。

**注释:**

CDC 成员函数中还包含有以下常用的绘图函数:FloodFill 函数、ExtFloodFill 函数、DrawIcon 函数、ScrollDC 函数、DrawFocusRect 函数、FillRect 函数、FrameRect 函数以及 InvertRect 函数。这些函数的功能描述请参阅联机帮助信息。

### 8.2.5 映射模式

映射模式是指在绘制过程中所依据的坐标系,它包括逻辑单位的实际大小和坐标轴的方向等。Windows 包含 8 种不同的 GDI 映射模式,如表 8-2 所示。

**表 8-2 GDI 映射模式**

| 映射模式 | 用途 |
|---|---|
| MM _ ANISOTROPIC | 把一个逻辑单位映射成任意一个大小的物理单位,X 和 Y 轴可以任意调节 |
| MM _ HIENGLISH | 把一个逻辑单位映射成 0.001 英寸,X 轴正方向朝右,Y 轴正方向朝上 |
| MM _ HIMETRTC | 把一个逻辑单位映射成 0.01 毫米,X 轴正方向朝右,Y 轴正方向朝上 |
| MM _ ISOTROPIC | 把一个逻辑单位映射为任意物理单位,X 轴与 Y 轴的单位比例为一比一 |
| MM _ LOENGLISH | 把一个逻辑单位映射成 0.01 英寸,X 轴正方向朝右,Y 轴正方向朝上 |
| MM _ LOMETRIC | 把一个逻辑单位映射成 0.1 毫米,X 轴正方向朝右,Y 轴正方向朝上 |
| MM _ TEXT | 缺省映射模式,把每个逻辑单位映射成一个设备像素,X 轴正方向朝右,Y 轴正方向朝下 |
| MM _ TWIPS | 把每个逻辑单位映射成一个打印点的 1/20,相当于 1/1440 英寸(一个打印点的 1/20),X 轴正方向朝右,Y 轴正方向朝上 |

映射模式分为强制映射模式和非强制映射模式。所谓强制映射模式就是指比例因子是固定的，应用程序不能改变映射到设备单位的逻辑单元数目。

1.强制映射模式

GDI 把映射模式中的 6 种定义为强制映射模式，表 8-3 列出了每种强制映射模式中的逻辑单位与物理单位的关系。

表 8-3　逻辑单位/物理单位转换表

| 强制映射模式 | 逻辑单位 | 物理单位 |
|---|---|---|
| MM _ HIENGLISH | 1000 | 1 英寸 |
| MM _ HIMETRIC | 100 | 1 毫米 |
| MM _ LOENGLISH | 100 | 1 英寸 |
| MM _ LOMETRIC | 10 | 1 毫米 |
| MM _ TEXT | 1 | 设备像素 |
| MM _ TWIPS | 1440 | 1 英寸 |

2.部分强制映射模式和非强制映射模式

非强制映射模式 MM _ ANISOTROPIC 和 MM _ ISOTROPIC 使用两个矩形区域来导出一个比例因子和方向，这两个矩形区域是窗口和观察口。窗口位于逻辑坐标空间内，而观察口位于物理坐标空间内，两者皆有原点、宽度和高度。原点可以是四个边角中的任何一个，宽度是从原点到对角的水平距离，高度是原点到对角的垂直距离。Windows 用观察口的宽度除以窗口的宽度来产生水平比例因子，用观察口的高度除以窗口的高度来产生垂直比例因子。这些比例因子确定了 Windows 映射到一定数量像素上的逻辑单位的数目。除确定比例因子外，窗口和观察口还确定了对象的方向。Windows 总是把窗口原点映射到观察口原点，将窗口宽度映射到观察口宽度，将窗口高度映射到观察口高度。

(1)部分强制映射模式

可以使用 MM _ ISOTROPIC 映射模式创建等比例输出，把一个在逻辑坐标中对称的对象映射成物理空间的对称对象。为了维持这种对称，GDI 缩小了观察口范围。缩小的数量依赖于要求的范围和设备的纵横比。这种映射模式称为部分强制映射模式，因为应用程序在改变比例因子时没有完全的控制权。

(2)非强制映射模式

MM _ ANISOTROPIC 是非强制模式，应用程序通过它可以完全改变垂直和水平比例因子，并可以在选择这种模式之后随意改变窗口和视口范围，同时在这种模式下 Windows 不会强制改变任一比例因子的值。

GDI 使用以下公式把逻辑点变换为设备点，或把设备点变换为逻辑点：

- 逻辑点变换设备点：

```
Dx = (Lx - xWO) * xVE/xWE + xVO
Dy = (Ly - yWO) * yVE/yWE + yVO
```

- 设备点变换为逻辑点

```
Lx = (Dx - xVO) * xWE/xVE + xWO
Ly = (Dy - yVO) * yWE/yVE + yWO
```

各变量的含义如表 8-4 所示。

表 8-4 逻辑点/设备点变换公式中各变量含义

| 变量 | 意义 |
|---|---|
| xWO | 窗口原点 x 坐标 |
| yWO | 窗口原点 y 坐标 |
| xWE | 窗口范围 x 坐标 |
| yWE | 窗口范围 y 坐标 |
| xVO | 观察口原点 x 坐标 |
| yVO | 观察口原点 y 坐标 |
| xVE | 观察口范围 x 坐标 |
| yVE | 观察口范围 y 坐标 |
| Lx | 逻辑坐标系的 x 坐标 |
| Ly | 逻辑坐标系的 y 坐标 |
| Dx | 设备 x 坐标 |
| Dy | 设备 y 坐标 |

**注释：**

可以调用 CDC 的成员函数 SetMapMode 来设置当前映射模式，函数原型为：

```
virtual int SetMapMode (int nMapMode);
```

3. 定义窗口区域

定义窗口的 x 范围和 y 范围，可使用 CDC 的成员函数 SetWindowExt。注意，SetWindowExt 函数只有当映射模式为 MM _ ANISOTROPIC 或 MM _ ISOTROPIC 映射模式时才有效。SetWin-

dowExt 函数的原型：

```
virtual CSize SetWindowExt( int cx, int cy );
virtual CSize SetWindowExt( SIZE size );
```

其中，参数 cx 和 cy 或者 Size 指定窗口的 x 范围和 y 范围。

4.定义视口

视口是接收输出的窗口区域。视口范围可以使用 CDC 的成员函数 SetViewportExt 来定义，SetViewportExt 只有在 MM _ ANISOTROPIC 和 MM _ ISOTROPIC 映射模式时才有效。SetViewportExt 函数原型为：

```
virtual CSize SetViewportExt( int cx, int cy );
virtual CSize SetViewportExt( SIZE size );
```

其中，参数 cx 和 cy 或者 Size 用于指定视口的 x、y 范围。

5.设置视口原点

缺省时，视口原点是窗口原点，坐标为(0,0)，可调用 CDC 的成员函数 SetViewportOrg() 来设置视口原点，函数原型为：

```
virtual CPoint SetViewportOrg( int x, int y );
virtual CPoint SetViewportOrg( POINT point );
```

其中，参数 x 和 y 或 point 用于指定新的视口原点。

6.逻辑坐标与设备坐标的转换

使用 CDC 的成员函数 DPtoLP，可在逻辑坐标和设备坐标之间转换，函数原型为：

```
void DPtoLP( LPPOINT lpPoints, int nCount = 1 ) const;
void DPtoLP( LPRECT lpRect ) const;
void DPtoLP( LPSIZE lpSize ) const;
```

其中，参数 LpPoints 是指向 POINT 结构或 CPoint 对象的数组，nCount 表示数组中的点数。LpRect 指向 RECT 结构或 CRect 对象，lpSize 指向 SIZE 结构或 CSize 对象。

另外，也可调用 LPtoDP 函数。函数原形为：

```
void LPtoDP( LPPOINT lpPoints, int nCount = 1 ) const;
void LPtoDP( LPRECT lpRect ) const;
void LPtoDP( LPSIZE lpSize ) const;
```

### 8.2.6　绘图模式

调用 CDC 的成员函数 SetROP2 可以改变绘图模式，函数原型为：

```
int SetROP2(int nDrawMode);
```

其中，参数 nDrawMode 指定所要求的绘图模式。绘图模式的宏及意义如表 8-5 所示。

表 8-5　绘图模式宏代码及意义

| 绘图模式 | 描述 |
|---|---|
| R2 _ BLACK | 像素总为黑色 |
| R2 _ WHITE | 像素总为白色 |
| R2 _ NOP | 像素保持不变 |
| R2 _ COPYPEN | 缺省绘图模式,像素为画笔颜色 |
| R2 _ NOT | 像素为显示颜色的反转色 |
| R2 _ NOTCOPYPEN | 像素为画笔颜色的反转色 |
| R2 _ MERGEPENNOT | 像素为画笔颜色与显示颜色的反转色的组合 |
| R2 _ MASKPENNOT | 像素为画笔颜色与显示颜色的反转色公共颜色的组合 |
| R2 _ MERGENOTPEN | 像素为画笔颜色反转色与显示颜色的组合 |
| R2 _ MASKNOTPEN | 像素为显示颜色与画笔颜色反转色公共颜色的组合 |
| R2 _ MERGEPEN | 像素为显示颜色与画笔颜色的组合 |
| R2 _ NOTMERGEPEN | 像素为 R2 _ MERGEPEN 颜色的反转色 |
| R2 _ MASKPEN | 像素为显示颜色与画笔颜色的公共颜色的组合 |
| R2 _ NOTMASKPEN | 像素为 R2 _ MASKPEN 颜色的反转色 |
| R2 _ XORPEN | 像素为画笔颜色或显示颜色的组合,但不同时为这两种颜色 |
| R2 _ NOTXOPEN | 像素为 R2 _ XORPEN 颜色的反转色 |

### 8.2.7　设置背景颜色

缺省时,背景颜色为白色。可以调用 CDC 的成员函数 SetBkColor 来设置新的背景颜色,函数原型为:

```
virtual COLORREF SetBkColor( COLORREF crColor );
```

其中,参数 crColor 用于指定新的背景颜色。

### 8.2.8　设置背景模式

利用 CDC 的成员函数 SetBkMode 来控制显示在屏幕上的背景颜色,SetBkMode 函数原型为:

```
int SetBkMode( int nBkMode);
```

其中，参数 nBkMode 指定背景模式，其值可以为 OPAQUE 或 TRANSPARENT，缺省时为 OPAQUE。为 OPAQUE 时，显示背景都改变为当前背景颜色；为 TRANSPARENT 时，不改变背景颜色。

## 8.3 文本处理

GDI 也提供了大量具有强大功能的文本输出函数，利用这些函数可以使得应用程序的文本输出丰富多彩，既可以有字体的变化，也可以有字形的变化。

### 8.3.1 文本输出函数

调用 CDC 的成员函数 TextOut 来输出文本，函数原型为：

```
virtual BOOL TextOut( int x, int y, LPCTSTR lpszString, int nCount );
BOOL TextOut( int x, int y, const CString& str );
```

参数 x 和 y 指定文本输出逻辑位置，参数 nCount 指定字符串长度，参数 lpszString 为要输出的字符串，参数 str 包含要输出字符的 CString 对象。

另外，CDC 的成员函数 DrawText，ExtTextOut 和 TabbedTextOut 均可以用来显示文本。

函数 DrawText 在指定的矩形区域内显示指定格式的文本。函数原型为：

```
virtual int DrawText( LPCTSTR lpszString, int nCount, LPRECT lpRect,UINT nFor-
mat );
int DrawText( const CString& str, LPRECT lpRect, UINT nFormat );
```

其中，参数 nCount 指定字符串长度，参数 lpszString 为待输出的字符串，str 为包含要输出字符的 CString 对象，lpRect 指定用于显示文本的矩形区域，nFormat 用于指定文本格式化的方法。

函数 ExtTextOut 用当前选择的字体在指定的矩形区域内显示字符串。函数原型为：

```
virtual BOOL ExtTextOut( int x, int y, UINT nOptiOns, LPCRECT lpRect, LPCTSTR lp-
szString, UINT nCount, LPINT lpDxWidths );
BOOL ExtTextOut( int x, int y, UINT nOptiOns, LPCRECT lpRect, const CString& str,
LPINT lpDxWidths );
```

其中，参数 x，y 给出指定字符串中第一个字符单元的坐标，nOptions 用于指定矩形类型，lpRect 用于确定矩形的大小，nCount 为要显示的字符数，lpDxWidths 为指向给出相邻字符单元原点之间距离值的一个数组，str 是包含要显示字符的 CString 对象。

函数 TabbedTextout 在指定位置显示字符串，并按指定制表间距扩展 Tab 符，函数原型为：

```
virtual CSize TabbedTextOut( int x, int y, LPCTSTR lpszStrirg, int nCount, int
nTabPositions, LPINT lpnTabStopPositions, int nTabOrigin );
CSize TabbedTextOut( int x, int y, const CString& str, int nTabPositions, LPINT
```

```
lpnTabStopPositions, int nTabOrigin );
```

其中，参数 x，y 指定文本输出的开始位置；参数 nCount 指定字符串长度，参数 lpszString 为要输出的字符串，参数 str 为包含要输入字符的 CString 对象，参数 nTabPositions、lpnTabStopPositions 和 nTabOrigin 用于指定制表间距。

### 8.3.2　设置文本属性

设置文本属性，可用以下方法：

(1) 成员函数 SetTextColor 按指定颜色设置文本颜色，函数原型为：

```
virtual COLORREF SetTextColor( COLORREF crColor );
```

其中，crColor 指定文本颜色为 RGB 色值。

(2) 成员函数 SetBKColor 按指定颜色设置背景颜色

(3) 成员函数 SetBKMode 设置背景模式

(4) 成员函数 SetMapMode 设置映射模式

(5) 成员函数 SetTextAlign 设置文本对齐方式，函数的原型为：

```
UINT SetTextAlign( UINT nFlags );
```

其中，nFlags 参数为文本对齐标志，其值如表 8-6 所示。

**表 8-6　文本对齐方式**

| 对齐标志 | 描述 |
|---|---|
| TA _ CENTER | 将点同边界矩形的水平中心对齐 |
| TA _ LEFT | 将点同边界矩形的左边界对齐 |
| TA _ RIGHT | 将点同边界矩形的右边界对齐 |
| TA _ BASELINE | 将点同所选字体的基线对齐 |
| TA _ BOTTOM | 将点同边界矩形的底线对齐 |
| TA _ TOP | 将点同边界矩形的顶线对齐 |
| TA _ NOUPDATECP | 在调用文本输出函数后，不更新当前位置 |
| TA _ UPDATECP | 在调用文本输出函数后，更新当前位置 |

(6) 成员函数 SetTextCharacterExtra 用于设置字符间的间隔值，函数原型为：

```
int SetTextCharacterExtra( int nCharExtra );
```

其中，参数 nCharExtra 用于指定要加到每个字符串的额外间隔值。

### 8.3.3　获取字符属性

使用 CDC 的成员函数 GetTextMetrics，可得到当前字体的完整描述，函数原型为：

```
BOOL GetTextMetrics( LPTEXTMETRIC lpMetrics ) const;
```

当前字体的描述存入在 TEXTMETRIC 结构中，TEXTMETRIC 结构定义如下：

```
typedef struct tagTEXTMETRIC {
int tmHeight; //字符的高度(ascent + descent)
int tmAscent; //字符上升的高度(基线以上的单元)
int tmDescent; //字符下降的高度(基线以下的单元)
int tmInternalLeading;//给出 tmHeight 字段设置的界限内的空隙数。注意符号和其他非英文符可以出现在这个区域内
int tmExternalLeading; //两行之间的空隙,此区域在字库区外,可以设置为零
int tmAveCharWidth; //字体中所有字符的平均宽度
int tmMaxCharWidth; //字体中最宽字符的宽度
int tmWeight; //字体的高度
BYTE tmItalic;//是否为斜体
BYTE tmUnderlined;//是否带下划线
BYTE tmStruckOut;//是否带删除线
BYTE tmFirstChar;//字体中的第一个字符
BYTE tmLastChar; //字体中的最后一个字符
BYTE tmDefaultChar;//给出将替代字库中没有的字符的字符值
BYTE tmBreakChar;//为进行排版处理,给出用于定义单词切分符的字符值
BYTE tmPitchAndFamily;//给出所选字库的间隔和字库族。字段的低位给出字库的字符间隔,为 1 时间隔可变,为 0 时间隔固定。4 个高位指定字库族
BYTE tmCharSet; //字体的字符集
int tmOverhang;//给出每一个字符串的额外宽度,用于合成字库,如斜体和黑体
int tmDigitizedAspectX;//设备的水平特征
int tmDigitizedAspectY;//设备的垂直特征
} TEXTMETRIC;
```

**注意：**

可以调用 CDC 的成员函数 GetTextExtent 来了解计算一串字符宽度的方式，函数原型为：

```
CSize GetTextExtent( LPCTSTR lpszString, int nCount ) const;
CSize GetTextExtent( const CString& str ) const;
```

### 8.3.4　使用字体

1.使用库存字体

Windows 支持 6 种库存字体，如表 8-7 所示。

表 8-7 Windows 库存字体及相应代码

| 库存字体 | 描述 |
| --- | --- |
| ANST _ FIXEDFONTANST | 固定系统字体 |
| ANST _ VARFONTANST | 可变系统字体 |
| DEVICE _ DEFAULT _ FONT | 与设备相关的字体 |
| OEM _ FIXED _ FONT | 设备固有字体 |
| SYSTEM _ FONT | 系统字体 |
| SYSTEM _ FIXEDFONT Windows | 早期版本使用的固定宽度的系统字体 |

2.创建用户字体

CFOnt 的成员函数 CreatFOnt 可用于创建逻辑字体,函数原型为:

```
BOOL CreateFOnt( int nHeight, int nWidth, int nEscapement, int nOrientatiOn, int
nWeight, BYTE bItalic, BYTE bUnderline, BYTE cStrikeOut, BYTE nCharSet, BYTE nOut-
PrecisiOn, BYTE nClipPrecisiOn, BYTE nQuality, BYTE nPitchAndFamily, LPCTSTR lpsz-
Facename );
```

下面解释参数的含义:

- nEscapement:指定文本显示时的角度(以十分之一度为单位)。
- nWeight:表示字体的浓淡程度(0—1000 之间),为 0 时则使用缺省浓度。
- bItalic:用于指定创建斜体字体,bItalic 非零时表示为斜体。
- nOutPrecisiOn:用于指定输出精度。输出精度定义了输出如何匹配要求的字库高度、宽度、字符方向、倾斜度和间距,它可取如下值:OUT _ CHARCTER _ PRECIS, OUT _ DEFAULT _ PRECIS, OUT _ STRING _ PRECIS, OUT _ STROKE _ PRECIS。
- nPitchAndFamily:用于指定字体的间距和字库族。
- lpSzFacename:用于指定字体名称。

此外,还可以使用 CFOnt 的成员函数 CreateFOntIndirect 创建逻辑字库。

Create FOntIndirect 函数原型为:

```
BOOL CreateFOntIndirect(const LOGFONT * lpLogFOnt);
```

其中,参数 lpLogFOnt 指向定义逻辑字体属性的 LOGFONT 结构。LOGFONT 的结构定义如下:

```
typedef struct tagLOGFONT{
LONG lfHeight;//字体高度
LONG lfWidth;//字体宽度
```

```
LONG lfEscapement;//文本显示的角度
LONG lfOrientation;//字体的角度
LONG lfWeight;//字体的磅数
BYTE lfItalic;//斜体字体
BYTE lfUnderline;//带下划线的字体
BYTE lfStrikeOut;//带删除线的字体
BYTE lfCharSet;//所需的字符集
BYTE lfOutPrecision;//输出的精度
BYTE lfClipPrecision;//裁剪的精度
BYTE lfQuality;//逻辑字体与输出没有提供的字体之间的精度
BYTE lfPitchAndFamily;//字符间距和像素族
CHAR lfFaceName[LF_FACESIZE];//字体名称
}LOGFONT;
```

## 8.4　点位图及其操作

点位图又常简称为位图，它是一种由位模式组成的，而不是由线组成的图形。位图有两种：设备相关的位图是依赖于特定的设备，只能在某个输出设备上显示的一种内存位映射，而设备无关的位图则描述了图形的实际外形，而不是由某种特定显示设备的图形内部表示来确定的，可适合于任何显示设备。

CBitmap 类用来管理点位图。实际编程时要先声明 CBitmap 类对象，然后调用适当的 CBitmap 类的成员函数来初始化位图对象。程序中使用点位图时一般应先建立点位图，接着装入点位图，然后可用各种方式显示点位图，在不需要点位图时可以删除它。

建立点位图的方式一般有四种：

(1) 使用各种画图实用程序，如 Windows 所提供的 MSPAINT、Visual C++ 集成的资源编辑器中的映射编辑器都是很好的画图工具。利用画图工具产生的点位图在程序中使用时应首先利用 Visual C++ 集成的资源编辑器加入到资源描述文件中，然后用 CBitmap 类的成员函数 LoadBitmap 将它装入。

(2) 在程序执行时建立一个空的点位图，再用前面介绍的绘图函数在空的点位图上“作画”。

(3) 先建立一个空的点位图，再将阵列数据填入。

(4) 先建立一个空的点位图，再依照一个已存在的设备无关位图来创建适用的点位图。

### 8.4.1　点位图操作常用函数

CDC 类提供的用于点位图操作的成员函数包括 PatBlt，BitBlt 和 StretchBlt。

1. PatBlt 函数

PatBlt 函数的原形为：

```
BOOL PatBlt( int x, int y, int nWidth, int nHeight, DWORD dwRop );
```

此函数在指定的设备上产生一种位模式。此模式是设备上已有的模式与所选定的笔刷的组合。参数 x 和 y 用于确定矩形区域的左上角坐标，nWidth 和 nHeight 是区域的宽度和高度，参数 dwRop 指定光栅操作码，此操作码决定了怎样组合此模式。表 8-8 列出了光栅操作码及含义。

**表 8-8 光栅操作码及含义**

| 代码 | 含义 |
|---|---|
| PATCOPY | 把模式复制到目标点位图 |
| PATINVERT | 使用 XOR 运算符来组合目的点位图和模式 |
| DSTINVERT | 反转目标点位图 |
| BLACKNESS | 使所有的输出变黑 |
| WHITENESS | 使所有的输出变白 |
| PATPAINT | 在直接的目标点位图上绘制 |

2. BitBlt 函数

BitBlt 函数的原型为：

```
BOOL BitBlt( int x, int y, int nWidth, int nHeight, CDC * pSrcDC, int xSrc, int ySrc, DWORD dwRop );
```

CDC 类的成员函数 BitBlt 将一个点位图从由 pSrcDC 给出的源设备上移到当前设备上，源设备和目标设备既可以是同一个设备对象，也可以是不同的设备对象。其中，参数 x 和 y 用于指定目标矩形区域的左上角坐标，nWidth 和 nHeight 为目标区域的宽度和高度，pSrcDC 是指向源设备对象的指针，xSrc 和 ySrc 指定的是源位图的左上角坐标，dwRop 指的是光栅操作码，它决定了位图的显示方式，取值如表 8-9 所示。

**表 8-9 光栅操作码及含义**

| 操作码 | 含义 |
|---|---|
| BLACKNESS | 使所有输出变成黑色 |
| DSTINVERT | 将目标位图反转 |
| MERGECOPY | 用布尔“与”(AND)操作符将源位图和目标位图结合起来 |
| MERGEPAINT | 用布尔“或”(OR)操作符将源位图和目标位图结合起来 |

续表

| 操作码 | 含义 |
| --- | --- |
| NOTSRCCOPY | 将反相的源位图拷贝到目标位图上 |
| NOTSRCERASE | 将目标位图和源位图的布尔“或”(OR)操作结果反转 |
| PATCOPY | 将图案拷贝到目标位图上 |
| SRCAND | 用布尔“与”(AND)操作将目标像素与源点位图结合 |
| SRCPAINT | 用布尔“或”(OR)操作将目标像素与源位图结合 |
| SRCINVERT | 用布尔“异或”(XOR)操作将目标像素与源位图结合 |
| WHITENESS | 使所有输出变成白色 |

3. StretchBlt 函数

StretchBlt 函数的原型为：

```
BOOL StretchBlt( int x, int y, int nWidth, int nHeight, CDC * pSrcDC, int xSrc,
int ySrc, int nSrcWidth, int nSrcHeight, DWORD dwRop );
```

CDC 的成员函数 StretchBlt 可以执行所有 BitBlt 可执行的操作，为了适应目标矩形框，有可能对点位图作压缩或放大处理。该函数利用目标设备描述表的变倍模式(由函数 SetStretchBltMode 设定)来决定如何进行点位图的压缩或放大处理。

## 8.4.2　点位图的装入

点位图的装入一般须经过两步：第一步，将点位图文件(扩展名为 .BMP)引入到资源文件中；第二步，调用 CBitmap 类的成员函数 LoadBitmap 将位图资源装入内存中。

1. 将点位图引入资源文件中

(1) 打开要使用点位图的项目文件

这可以通过 File 菜单的 Open 命令来完成，此命令弹出“打开”对话框，如图 8-2 所示。

(2) 向项目中添加点位图资源

选择 Insert 菜单的 Resource...命令，在 Insert Resource 对话框(如图 8-3 所示)中，选择 Bitmap。

**注释：**

如果要使用 Visual C++ 所提供的映像编辑器新建点位图，单击 New 按钮；如果要使用其他绘图工具已经绘好的点位图，单击 Import 按钮。例如要引入 Windows 提供的蓝天白云(文件为 C:\Windows\Clouds.bmp)，单击 Import 按钮，弹出 Import Resource 对话框，图 8-4 显示了在此对话框中所作的选择，选择各项后单击 Import 按钮即完成点位图资源的引入。

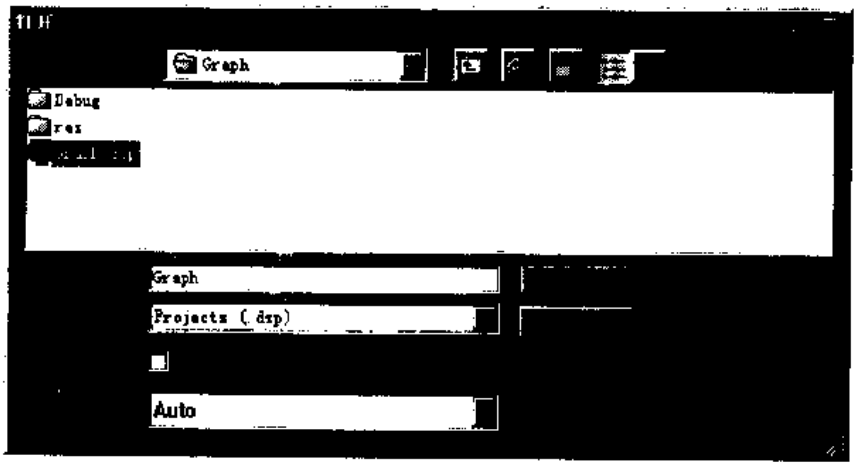



图 8-2 “打开”对话框

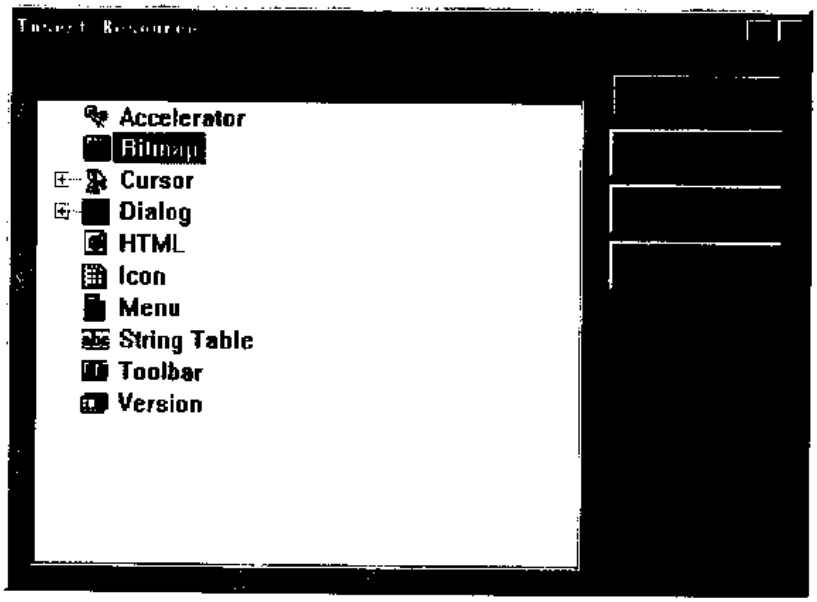



图 8-3 插入点位图时的 Insert Resource 对话框

2. 调用 CBitmap 类的成员函数 LoadBitmap 将位图资源装入内存

LoadBitmap 函数的原型为：

```
BOOL LoadBitmap( LPCTSTR lpszResourceName );
BOOL LoadBitmap( UINT nIDResource );
```

此函数装入当前应用程序中 lpszResourceName 所表示的资源名或 nIDResource 所代表的资源 ID。如果装入成功，返回指向 CBitmap 对象的指针，否则返回 NULL。

装入语句为：

```
MyBitmap.LoadBitmap(IDB_BITMAP1);
```

或：

```
MyBitmap.LoadBitmap(CLOUDS);
```

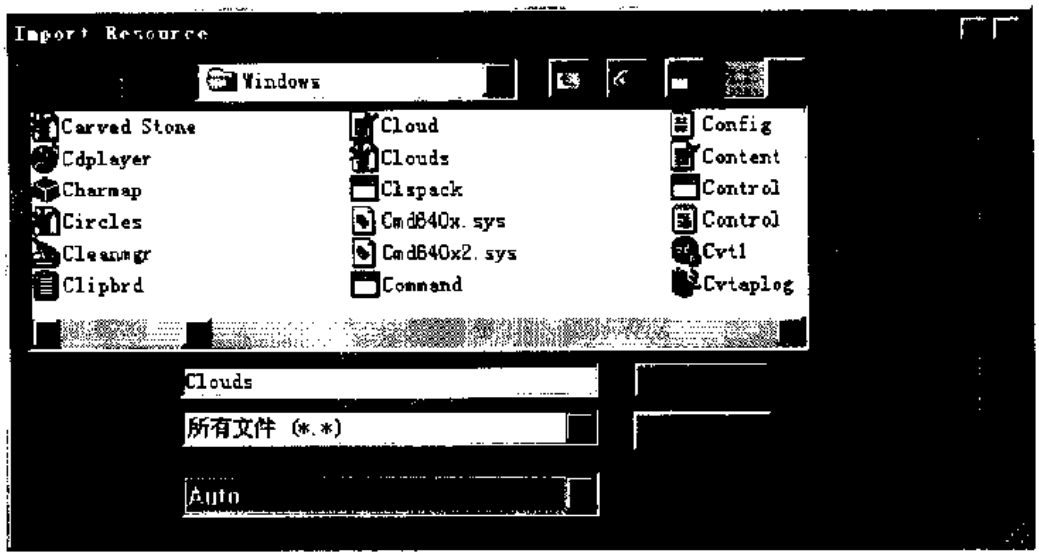



图 8-4　Import Resource 对话框

### 8.4.3　点位图的显示

在 Windows 环境下显示点位图的方式有以下几种：

- 使用 CDC 类的成员函数 BitBlt 将存储器中的点位图拷贝到屏幕上。
- 使用 CDC 类的成员函数 StretchBlt 将存储器中的点位图放大或缩小并显示在屏幕上。
- 使用 CBrush 类的成员函数 CreatePatternBrush 建立一个由点位图构成的笔刷，然后任何用到笔刷的 GDI 函数都可显示该点位图。
- 使用 SetDIBitsToDevice 函数将一与设备无关的点位图直接在输出设备上显示出来。

## 8.5　图　　标

图标是一个小图像，可用来代表一个缩小后的应用程序。图标也可以放在信息框或对话框中。

对应用程序而言，图标也算是一种资源，它是一种与设备相关的资源，在资源编译之前每个图标为一个单独的文件，含有一组位图图像，这组图像在外表上很相似，但分别对应不同的显示设备(如 CGA、EGA 和 VGA 等)。当应用程序要求使用某个图标时，它只需要用名字为其请求图标资源，然后由 Windows 确定哪个图标图像最适合当前显示器。

对于 Windows 系统预定的图标，必须调用 CWinApp 的成员函数 LoadStandardIcon 来装载，函数 LoadStandardIcon 原型为：

```
HICON LoadStandardIcon( LPCTSTR lpszIconName ) const;
```

参数 lpszIconName 用于指定 Windows 预定义的图标宏，取值如表 8-10 所示：

表 8-10 Windows 预定义的图标宏

| 宏代码 | 图标形状 |
|---|---|
| IDI _ APPLICATION | 缺省的应用程序图标 |
| IDI _ HAND | 手形图标(用于各种严重警告信息) |
| IDI _ QUESTION | 问号(用于提示信息) |
| IDI _ EXCLAMATION | 惊叹号(用于警告信息) |

对于使用图形编辑器创建的图标资源,必须调用 CWinApp 的成员函数 LoadIcon 来装载,函数原型为:

```
HICON LoadIcon( LPCTSTR lpszResourceName ) const;
HICON LoadIcon( UINT nIDResource ) const;
```

其中,参数 lpszResourceName 为图标的名字,nIDResource 为图标的资源符号。

以下代码演示了如何装载图标资源:

```
HICON hIcon;
hIcon = AfxGetApp() - > LoadIcon(IDI _ ICON1);
```

装载之后,调用 CDC 的成员函数 DrawIcon 来显示图标,函数原型为:

```
BOOL DrawIcon( int x, int y, HICON hIcon );
BOOL DrawIcon( POINT point, HICON hIcon );
```

其中,参数 x 和 y 或 point 用于指定图标显示位置的左上角坐标,hIcon 为图标句柄。

## 8.6 光　　标

光标是一组用来显示鼠标目前位置的特殊的点位图。它是一种 32 × 32 点阵的位图,可显示出被鼠标初始化的那个动作将发生在哪里。在 Windows 环境下,利用光标形状的变换来表示系统状态的改变。

Windows 系统提供的预定义光标必须调用 CWinApp 的成员函数 LoadStandardCursor 来装载。对于用图形编辑器建立的光标资源,必须调用 CWinApp 的成员函数 LoadCursor 来装载,函数 LoadStandardCursor 的原型为:

```
HCURSOR LoadStandardCursor( LPCTSTR lpszCursorName ) const;
```

函数 LoadCursor 的原型为:

```
HCURSOR LoadCursor( LPCTSTR lpszResourceName ) const;
HCURSOR LoadCursor( UINT nIDResource ) const;
```

其中,参数 lpszCursorName 为光标的名字。以下代码用来装载光标资源:

```
HCURSOR hcursor;
hcursor = AfxGetApp() - > LoadStandardCursor (IDI _ CURSOR1);
```

加载光标后，可使用 Windows API 函数 SetCursor 来设置光标形状。

## 8.7　打印和打印预览

文档打印时，OnDraw 函数接收到的设备描述表对象是与当前打印相关的 CDC 对象。在生成应用程序的基本边框时，已经提供有响应 File 菜单的 Print 和 Print Preview 命令的缺省实现。

例如：

```
//标准的打印命令
ON _ COMMAND(ID _ FILE _ PRINT, CView::OnFilePrint)
ON _ COMMAND(ID _ FILE _ PRINT _ DIRECT, CView::OnFilePrint)
ON _ COMMAND(ID _ FILE _ PRINT _ PREVIEW, CView::OnFilePrintPreview)
```

在 MFC 中，打印和打印预览的主要过程如下：

(1) 调用 OnPreparePrinting 函数创建与打印有关的设备描述表对象。

(2) 调用 OnBeginPrinting 函数并基于已创建的设备描述表对象来设置文档特性，将任何需要的信息保存在设备描述表对象中，例如分配字体、画笔、画刷和其他用于打印的对象。

(3) 调用 OnPrepareDC 函数为打印设置文本或绘图属性，调整视口原点以便打印当前页。

(4) 调用 OnPrint 函数，该函数在调用 OnDraw 前先选择一种字体或其他由 OnBeginPrinting 分配的对象，然后调用 OnDraw 函数执行输出。

(5) 调用 OnEndPrinting 删除由 OnBeginPrinting 分配的所有对象。

此外，还可调用 CDC 的成员函数 GetDevicecaps 和 Escape 来帮助管理打印任务。

## 8.8　绘图功能的实现

在本章第一节利用 MFC AppWizard 已经产生了应用程序的框架，但是此框架程序尚不能在屏幕(或打印机)上绘图。因此必须进一步修改框架程序，以使其能够在屏幕上绘制出各种图形和文本。

为简单起见，我们只对视图类 CGraphView 加以修改，以使该程序可以在视图区绘制本章前面所介绍的各种图形。

具体做法如下：执行 View 菜单的 ClassWizard 命令，出现 MFC ClassWizard 对话框，选择 Object IDs 列表框中的 CGraphView，选择 Messages 列表框中的 WM _ PAINT，然后单击 Add Function 按钮，MFC ClassWizard 对话框显示如图 8-5 所示，最后单击 Edit Code 按钮，添加 OnPaint 的实现代码。

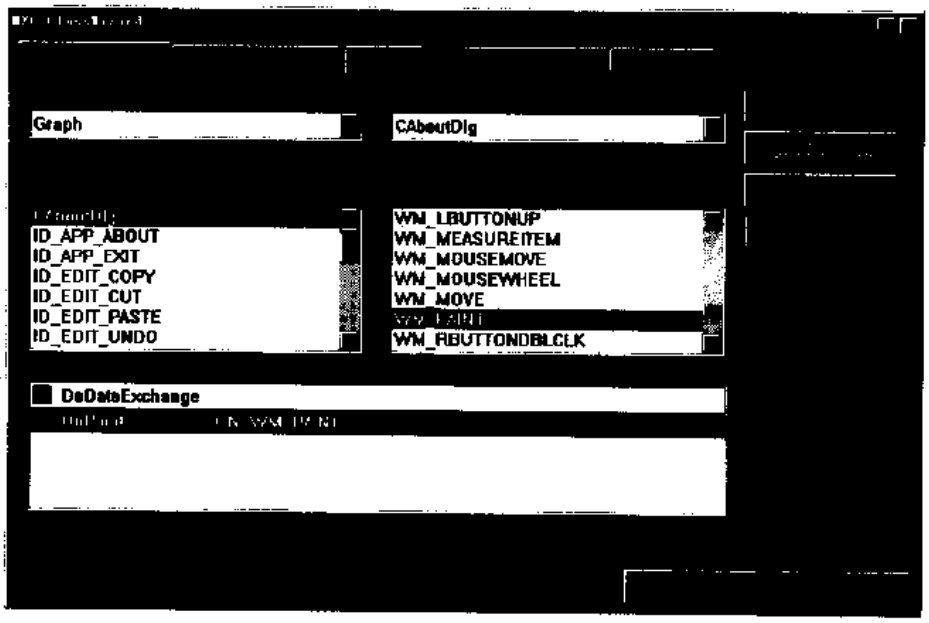



图 8-5　为视图类 CGraphView 添加代码

程序清单 8-3 列出了 CGraphView 类的全部实现代码，其中添加的成员函数 OnPaint 的代码以黑体显示。运行此程序的结果显示如图 8-6 所示。

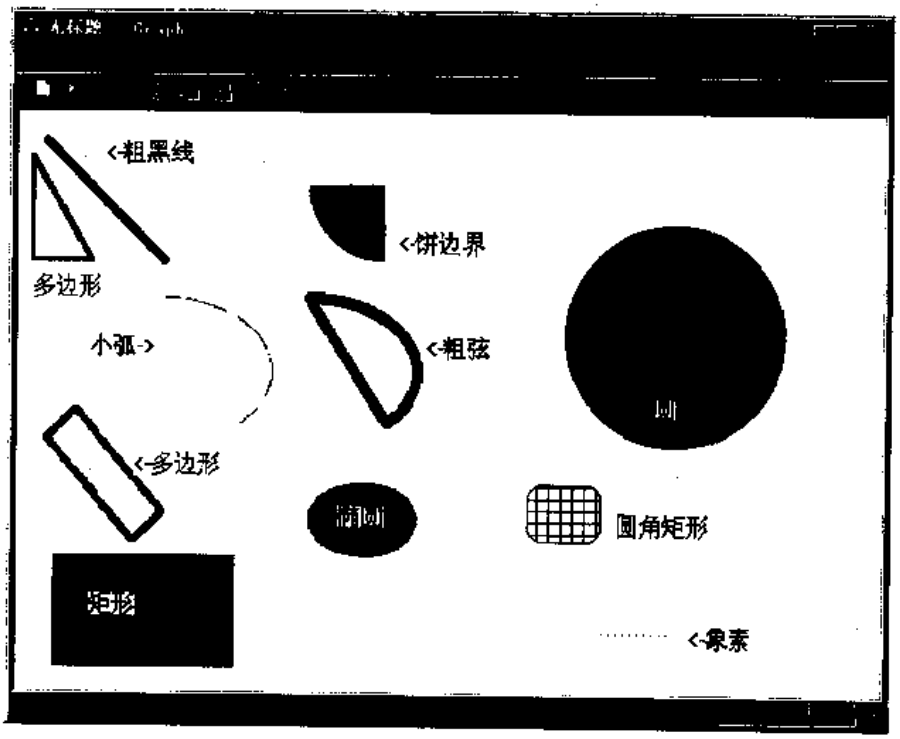



图 8-6　Graph 程序运行结果

**程序清单 8-3：修改后的 CGraphView 类的 OnPaint 成员函数**

```
#define _UNICODE
#define __UNICODE
// GraphView.cpp : 实现 CGraphView 类
//

#include "stdafx.h"
#include "Graph.h"

#include "GraphDoc.h"
#include "GraphView.h"

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

/////////////////////////////////////////////////////////////////////////////
// CGraphView

IMPLEMENT_DYNCREATE(CGraphView, CView)

BEGIN_MESSAGE_MAP(CGraphView, CView)
        //{{AFX_MSG_MAP(CGraphView)
        ON_WM_PAINT()
        //}}AFX_MSG_MAP
        // 标准打印命令
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CGraphView 构造函数/析构函数

CGraphView::CGraphView()
{
        // 添加函数代码
```

```
}

CGraphView::~CGraphView()
{
}

BOOL CGraphView::PreCreateWindow(CREATESTRUCT& cs)
{
        //修改 Window 类的样式
        // the CREATESTRUCT cs
        cs.lpszClass = AfxRegisterWndClass(
                CS_HREDRAW|CS_VREDRAW,//类风格
                0,//不用光标
                (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
                0);//没有图标


        return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CGraphView 绘制代码

void CGraphView::OnDraw(CDC* pDC)
{
        CGraphDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        //添加绘制代码
}

/////////////////////////////////////////////////////////////////////////////
// CGraphView 打印

BOOL CGraphView::OnPreparePrinting(CPrintInfo* pInfo)
{
        return DoPreparePrinting(pInfo);
}

void CGraphView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
        //添加打印前的额外初始化代码
```

```
}

void CGraphView::OnEndPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
        //添加打印后的清除代码
}

/////////////////////////////////////////////////////////////////////////////
// CGraphView 诊断

#ifdef _DEBUG
void CGraphView::AssertValid() const
{
        CView::AssertValid();
}

void CGraphView::Dump(CDumpContext& dc) const
{
        CView::Dump(dc);
}

CGraphDoc * CGraphView::GetDocument()
{
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CGraphDoc)));
        return (CGraphDoc *)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CGraphView 消息处理程序

void CGraphView::OnPaint()
{

        static DWORD dwColor[9] = {RGB(0,0,0),      //黑色
                    RGB(255,0,0), //红色
                    RGB(0,255,0), //绿色
                    RGB(0,0,255), //蓝色
                    RGB(255,255,0), //黄色
                    RGB(255,0,255), //紫色
```

```
            RGB(0,255,255), //洋红色
            RGB(127,127,127),//灰色
            RGB(255,255,255)};//白色
POINT polylpts[4],polygpts[5];

int xcoord;

CBrush newbrush;//新笔刷
CBrush * oldbrush;//老笔刷对象指针
CPen newpen;//新画笔
CPen * oldpen;//老画笔对象指针

CPaintDC dc(this);//绘图的设备描述表

//绘制填充色为红色的椭圆
newpen.CreatePen(PS_SOLID,1,dwColor[1]);//建立新画笔
oldpen = dc.SelectObject(&newpen);//选择所建立的新画笔
newbrush.CreateSolidBrush(dwColor[1]);//建立新笔刷
oldbrush = dc.SelectObject(&newbrush);//选择所建立的新笔刷
dc.Ellipse(275,300,200,250);//画椭圆
dc.TextOut(220,265,"椭圆",4);//输出文本
dc.SelectObject(oldbrush);//恢复原有笔刷
newbrush.DeleteObject();//删除新笔刷
dc.SelectObject(oldpen);//恢复原有画笔
newpen.DeleteObject();//删除新画笔

//用 Ellips 函数绘制并填充一个蓝色圆
newpen.CreatePen(PS_SOLID,1,dwColor[3]);//建立新画笔
oldpen = dc.SelectObject(&newpen);//选择所建立的新画笔
newbrush.CreateSolidBrush(dwColor[3]);//建立新笔刷
oldbrush = dc.SelectObject(&newbrush);//选择所建立的新笔刷
dc.Ellipse(375,75,525,225);//画椭圆
dc.TextOut(435,190,"圆",2);//输出文本
dc.SelectObject(oldbrush);//恢复原有笔刷
newbrush.DeleteObject();//删除新笔刷
dc.SelectObject(oldpen);//恢复原有画笔
newpen.DeleteObject();//删除新画笔

//绘制几个绿色像素点
for(xcoord = 400;xcoord < 450;xcoord += 5)
        dc.SetPixel(xcoord,350,0L);
```

```
dc.TextOut(460,345,"< - 像素",6);

//绘制几条粗黑线
newpen.CreatePen(PS_SOLID,6,dwColor[0]);
oldpen = dc.SelectObject(&newpen);
dc.MoveTo(20,20);
dc.LineTo(100,100);
dc.TextOut(60,20,"< - 粗黑线",8);
dc.SelectObject(oldpen);
newpen.DeleteObject();

//绘制一段蓝色弧
newpen.CreatePen(PS_DASH,1,dwColor[3]);
oldpen = dc.SelectObject(&newpen);
dc.Arc(25,125,175,225,175,225,100,125);
dc.TextOut(50,150,"小弧 - >",6);
dc.SelectObject(oldpen);
newpen.DeleteObject();

//绘制绿色的粗弦
newpen.CreatePen(PS_SOLID,8,dwColor[2]);
oldpen = dc.SelectObject(&newpen);
dc.Chord(125,125,275,225,275,225,200,125);
dc.TextOut(280,150,"< - 粗弦",6);
dc.SelectObject(oldpen);
newpen.DeleteObject();

//绘制边界为黑色,填充色为绿色的饼图
newpen.CreatePen(PS_SOLID,1,dwColor[0]);
oldpen = dc.SelectObject(&newpen);
newbrush.CreateSolidBrush(dwColor[2]);//建立新笔刷
oldbrush = dc.SelectObject(&newbrush);//选择所建立的新笔刷
dc.Pie(200,0,300,100,200,50,250,100);
dc.TextOut(260,80,"< - 饼边界",8);
dc.SelectObject(oldpen);
newpen.DeleteObject();
dc.SelectObject(oldbrush);//恢复原有笔刷
newbrush.DeleteObject();//删除新笔刷

//绘制黑色矩形,填充色为灰色
newbrush.CreateSolidBrush(dwColor[7]);//建立新笔刷
```

```
oldbrush = dc.SelectObject(&newbrush);//选择所建立的新笔刷
dc.Rectangle(25,300,150,375);
dc.TextOut(50,325,"矩形",4);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();

//绘制黑色圆角矩形,填充色为蓝色
newbrush.CreateHatchBrush(HS_CROSS,dwColor[3]);//建立新笔刷
oldbrush = dc.SelectObject(&newbrush);//选择所建立的新笔刷
dc.RoundRect(350,250,400,290,20,20);
dc.TextOut(410,270,"圆角矩形",8);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();

//用多边形函数 Polyline 绘制洋红色直线
newpen.CreatePen(PS_SOLID,3,dwColor[5]);
oldpen = dc.SelectObject(&newpen);
polylpts[0].x = 10;
polylpts[0].y = 30;
polylpts[1].x = 10;
polylpts[1].y = 100;
polylpts[2].x = 50;
polylpts[2].y = 100;
polylpts[3].x = 10;
polylpts[3].y = 30;
dc.Polyline(polylpts,4);
dc.TextOut(10,110,"多边形",6);
dc.SelectObject(oldpen);
newpen.DeleteObject();

//用多边形函数 Polylgon 绘制青色多边形,填充为斜黄线
newpen.CreatePen(PS_SOLID,4,dwColor[6]);
oldpen = dc.SelectObject(&newpen);
newbrush.CreateHatchBrush(HS_FDIAGONAL,dwColor[4]);
oldbrush = dc.SelectObject(&newbrush);
polygpts[0].x = 40;
polygpts[0].y = 200;
polygpts[1].x = 100;
polygpts[1].y = 270;
polygpts[2].x = 80;
polygpts[2].y = 290;
```

```
polygpts[3].x = 20;
polygpts[3].y = 220;
polygpts[4].x = 40;
polygpts[4].y = 200;
dc.Polygon(polygpts,5);
dc.TextOut(80,230,"< - 多边形",8);
dc.SelectObject(oldbrush);//恢复原有笔刷
newbrush.DeleteObject();//删除新笔刷
dc.SelectObject(oldpen);//恢复原有画笔
newpen.DeleteObject();//删除新画笔
```

}

# 第 9 章

# 对话框和控件

对话框是一些弹出式窗口(Pop _ up Window),应用程序利用它们可和用户进行交互操作。对话框中通常包含有一个或多个控件。本章介绍对话框的工作方式、设计原理和设计步骤以及如何向对话框添加控件。并利用上一章的应用程序 Graph 来示范对话框的开发过程,并通过添加的对话框来改变绘图工具的属性,如画笔风格、笔刷风格等。

## 9.1 对 话 框

对话框是应用程序用于显示或提示,并等待用户输入信息的弹出式窗口。对话框包括了一个或多个控件,利用这些控件,用户可以输入文本、选择选项,并完成某一特定命令。

要处理对话框,需提供一个对话框模板(Dialog Template),一个对话框函数和一些用于呼叫对话框的机制。

所谓对话框模板,是指描述对话框及它所包含控件的文本,模板可以用对话框编辑器来帮助建立,一旦模板已建立,就可以把它加入到资源描述文件.RC 中。对话框函数是一个回调函数,Windows 调用对话框函数并向它传送对话框的消息。

由 CDialog 类管理的对话框有两种类型,即模态对话框和非模态对话框。

模态对话框是最常用的,它要求用户在程序继续执行之前作出响应;非模态对话框允许用户为对话框提供信息,而无需关闭该对话框就可以回到前面的任务。

对话框通常由菜单中的命令来启动。例如 File 菜单中的 Open 和 Save as 命令都可启动一个对话框。

创建对话框模板可以利用资源编辑器中的对话框编辑器,而对话框类的建立则可以利用 Class Wizard 来完成。这两种工具的使用方法对于设计模态对话框和非模态对话框都是一样的。具体过程如下:

(1) 用对话框编辑器设计对话框并创建相应的对话框模板资源。

(2) 用 Class Wizard 创建对话框类。

(3) 把对话框资源的控件与对话框类的消息处理函数链接起来。

(4) 用 Class Wizard 添加与对话框控件有关的成员函数变量,并为控件指定对话框数据交换和对话框数据验证。

### 9.1.1 模态对话框

所谓模态对话框(Modal Dialog Boxes)是指这种对话框出现时,它的父窗口将暂时失效,只有处理完对话框所要求的动作后,才会将控制权交回给父窗口。例如从 Notepad 应用程序的 File 菜单中选择 Open 命令时,就会出现一个模态对话框,此时若不输入文件名或结束对话框,Notepad 就无法继续工作。

虽然可以赋以模态对话框任何窗口风格,但是 Windows 建议使用 DS _ MODALFRAME、WS _ CAPTION 及 WS _ SYSMENU。DS _ MODALFRAME 风格可使模态对话框具备它独有的厚边。

当屏幕上出现模态对话框时,它将拥有自己的消息循环,因此所有的输入窗口信息都不

会被主窗口的消息循环所获得。模态对话框出现时会令父窗口失败，因此模态对话框绝不能指定为 WS _ CHILD 风格，如果这样，父窗口失效时，子窗口（对话框自己）也会跟着失效，如此一来就无法继续执行了。

### 9.1.2　非模态对话框

非模态对话框（Modeless Dialog Boxes）又称为共存式对话框，它不会令父窗口失效，因此非模态对话框出现时，使用者可以继续执行父窗口的工作。例如 Word 应用程序中，通过 Edit 菜单中的 Find 命令启动的对话框即为非模态对话框。

大部分非模态对话框都指定 WS _ POPUP、WS _ CAPTION、WS _ BORDER 及 WS _ SYSTEMMENU 风格。典型的非模态对话框包括系统菜单和标题。

非模态对话框从主程序的消息循环中取得输入信息。如果非模态对话框中有控件存在，而且允许使用键盘去选择，那么必须在信息循环中调用 CWnd 类（或其继承类）的成员函数 IsDialogMessage 来判断输入信息是否要送给非模态对话框，如果是的话就处理它，否则按原来的途径来处理。

非模态对话框的建立靠 CDialog 类的成员函数 Create 来完成，关闭则用 CWnd 类的成员函数 DestroyWindow 来实现。

**注释：**

对话框除了模态和非模态两种外，还有一种所谓“排它”式的对话框。当该种对话框出现时，不仅其父窗口失去作用，就连其他的窗口也一并失效，这种窗口通常用来显示严重的错误信息。只要在对话框的窗口风格中加入 WS _ SYSMODAL 或用 SetSysmodalWindow( ) 来设定，就可轻而易举地建立一个“排它”式对话框。

### 9.1.3　创建对话框资源

可用 Microsoft Visual Studio 的对话框编辑器来设计对话框并创建对话框资源。利用对

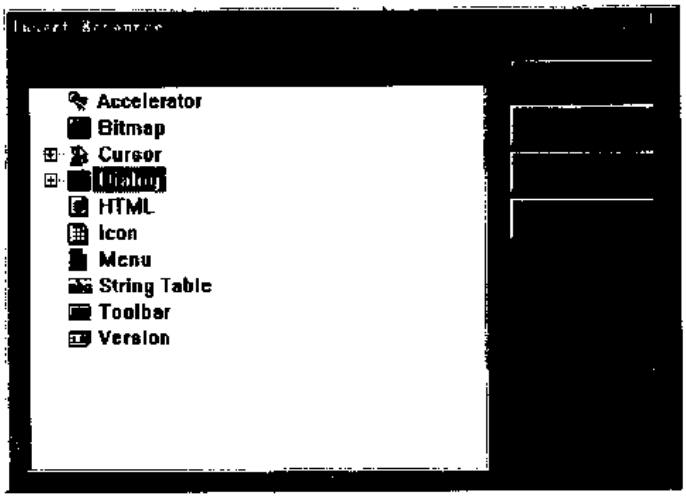



图 9-1　添加对话框资源时的 Insert Resource 对话框

话框可以很方便地添加各种控件到对话框上。在设计对话框时,应该使 Controls 和 dialog 工具栏出现,因为添加和布置控件时经常用到这两个工具栏。关于这两个工具栏的用法参见第 2 章的 2.2 节。在这里特别提醒读者,Dialog 工具栏中的第一个工具 Test 可用于测试对话框,利用它不必通过编译和链接应用程序就能进行测试,大大节约了时间,提高了开发效率。

要添加对话框资源,请选择 Insert 菜单中的 Resource...命令,并在出现的 Insert Resource 对话框中单击 Dialog 选项,如图 9-1 所示。然后单击 New 按钮,自动打开对话框编辑器,此时开发环境如图 9-2 所示。

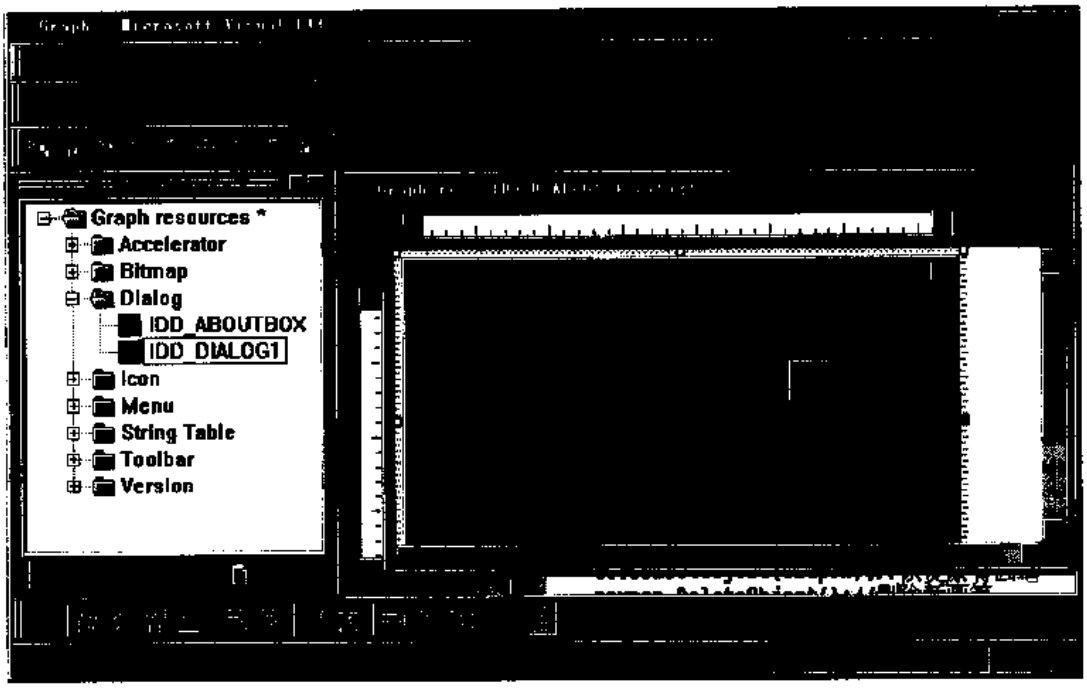



图 9-2 在进行对话框编辑时的开发环境

从图 9-2 可以看出,新建的对话框已经具有了两个按钮控件即 OK 和 Cancel。

### 9.1.4 创建对话框类

可用 Class Wizard 来创建对话框类并映射消息。Class Wizard 可帮助用户管理以下与对话框有关的任务:

- 从 CDialog 类继承新的对话框类来管理对话框。
- 映射 Windows 消息到对话框类。
- 声明对话框类的成员变量来表示对话框中的控件。
- 指定控件和成员变量之间如何交换数据。

用 Class Wizard 创建对话框类时,将在应用程序指定的.H 文件中添加对话框类的类声明;在.CPP 文件中添加类的消息映射、对话框的标准构造函数和成员函数 DoDataExchange 的新实现。

### 9.1.5 创建并显示对话框

创建对话框首先要构造对话框对象,然后创建对话框窗口。

1. 创建对话框

创建模态对话框时有固定的步骤可依循:

(1)调用两个 CDialog 公有构造函数中的任何一个来构造对话框对象。

(2)调用对话框对象的成员函数 DoModal 装载对话框资源,显示对话框,并管理与对话框的交互作用,直到用户选择 OK 或 Cancel。

创建非模态对话框的步骤与创建模态对话框类似,首先调用公有构造函数,然后调用对话框对象的成员函数 Create 装载对话框。

使用内存中的对话框模板还可以间接创建对话框,步骤如下:

(1)使用 DLGTEMPLATE 结构定义对话框的大小和风格。

(2)已经构造 CDialog 对象后,调用 CDialog 的成员函数 CreateIndirect 创建非模态对话框,或调用 MitModalIndirect 与 DoModal 创建模态对话框。

2. 初始化对话框

创建对话框后,可以调用对话框对象的成员函数 OnInitDialog 来初始化对话框。对于模态对话框,在调用 DoModal 时调用 OnInitDialog;对于非模态对话框,在调用 Create 时调用 OnInitDialog。

### 9.1.6 处理对话框消息

对话框也是 Windows 窗口,对话框的建立、用户对对话框中各控件的操作和对话框的终止都将产生消息,对话框应适当地处理这些消息,具体步骤如下:

(1) 使用 Class Wizard 把消息映射到对话框类中。

(2) 编写消息处理函数中的代码。

在 CDialog 继承类中,必须重设以下虚拟成员函数:

- 响应消息 WM _ INITDIALOG 初始化对话框时,重设成员函数 OnInitDialog。
- 单击 OK 按钮响应控件通知消息 BN _ CLICKED 时,重设成员函数 OnOK。
- 单击 Cancel 按钮响应控件通知消息 BN _ CLICKED 时,重设成员函数 OnCancel。

对话框对控件的各种操作一般应有相应的消息处理函数,在对话框类的声明中应有其相应原型,在对话框类的实现中应有相应的实现代码。

### 9.1.7 关闭对话框

选择 OK 或 Cancel 按钮可以关闭模态对话框。CDialog 类提供有缺省处理函数 OnOK 和 OnCancel。缺省处理函数将调用 CDialog 的成员函数 EndDialog 关闭对话框窗口。

对于非模态对话框,当其被关闭或删除时,缺省处理函数 OnClose 将调用 RestroyWindow 删除该对话框。

### 9.1.8 管理对话框的 MFC 函数

对话框类是从 CDialog 类继承的，CDialog 类提供了管理对话框的成员函数，可以在对话框类的消息处理函数中调用这些函数：

- EndDialog 用于关闭模态对话框，函数原型为：

```
void EndDialog(int uReslut);
```

- GetDef _ ID 返回对话框中的控件 ID，函数原型为：

```
DWORD GetDefID( ) const;
```

- GotoDlgCtrl 移入焦点到对话框的指定控件中。函数原型为：

```
void GotoDlgCtrl( CWnd * pWndCtrl );
```

其中，参数 pWndCtrl 标识接收输入焦点的控件。

- MapDialogRect 转换对话框单位为屏幕单位，函数原型为：

```
void MapDialogRect( LPRECT lpRect ) const;
```

其中，参数 lpRect 指向包含要转换的对话框坐标的 RECT 结构或 CRect 对象。

- NextDlgCtrl 移动输入焦点到对话框中的下一个控件上，函数原型为：

```
void NextDlgCtrl( ) const;
```

- PrevDlgCtrl 移动输入焦点到对话框中的上一个控件上，函数原型为：

```
void PrevDlgCtrl( ) const;
```

- SetDefID 更改对话框的缺省按钮控件，函数原型为：

```
void SetDefID( UINT nID );
```

## 9.2 控　　件

控件是一种特定类型的输入或输入窗口，例如可让用户输入字符串的“编辑控件”，如图 9-3 所示。

由于控件也是一种窗口，因此也必然拥有一个窗口类，而在窗口类中通常都会指定一个




图 9-3　编辑控件

窗口函数。但是因为控件的窗口类是预先设定的,因此 Windows 会为该控件预设一个窗口函数,所以每个控件表现出来的功能都不一样。

### 9.2.1　标准控件

在 Visual C++ 中已经提供的标准控件有:静态控件(CStatic)、按钮控件(CButton)、列表框控件(ClistBox)、组合框控件(CComboBox)、编辑控件(CEdit)和滚动条控件(CScrollBar)。下面分别介绍这些控件。

1.静态控件

静态控件用于显示文本串、框、矩形、光标、图标、位图和元文件。静态控件的基类是 CStatic,该类具有下列用来设置或取得控件相关属性的成员函数,如表 9-1 所示。

表 9-1　静态控件类 CStaic 的成员函数

| 函数 | 说明 |
|---|---|
| SetBitmap | 指定静态文本控件中要显示的位图 |
| SetIcon | 指定静态文本控件中要显示的图标 |
| SetCursor | 指定静态文本控件中要显示的光标 |
| SetEndMetaFile | 指定静态文本控件中要显示的元文件 |
| GetBitmap | 获取由 SetBitmap 设置的位图句柄 |
| GetIcon | 获取由 SetIcon 设置的图标句柄 |
| GetCursor | 获取由 SetCursor 设置的光标句柄 |
| GetEndMetaFile | 获取由 SetEndMetaFile 设置的元文件句柄 |

2.按钮控件

按钮控件是一种矩形子窗口,单击或双击可以执行某一任务。单击时,将产生控件通知消息 BN_CLICKED;双击时,将产生控件通知消息 BN_DOUBLECLICKED。按钮控件的基类是 CButton,它的成员函数及其作用如表 9-2 所示。

表 9-2　按钮控件类 CButton 的成员函数

| 函数 | 说明 |
|---|---|
| GetState | 获取单选按钮或复选框的状态 |
| SetState | 设置按钮控件的高亮状态 |

续表

| 函数 | 说明 |
| --- | --- |
| SetCheck | 设置单选按钮或复选框的选中状态(参数为 0 时代表未选中;参数为 1 时代表选中;参数为 2 时代表不确定) |
| GetButtonStyle | 返回按钮对象的 BS 风格值 |
| SetButtonStyle | 更改按钮对象的风格值 |
| SetIcon | 指定按钮上显示的图标 |
| GetIcon | 返回由 SetIcon 设置的图标句柄 |
| SetBitmap | 指定按钮上显示的位图 |
| SetCursor | 指定按钮上显示的光标 |
| GetBitmap | 返回由 SetBitmap 设置的位图句柄 |
| GetCrusor | 返回由 SetCrusor 设置的光标句柄 |

3.列表框控件

列表框控件用来显示项目列表,用户可以进行查看和选择。单项选择列表框中,只能选择一项;多项选择列表框中,可选择某一范围内的项目。

如果要处理列表框控件窗口传送给父窗口的控件通知消息,则必须为每条消息添加消息映射条目和消息处理函数。与列表框有关的常见消息如表 9-3 所示。

表 9-3 与列表框有关的常见消息

| 消息 | 产生的原因 |
| --- | --- |
| LBN _ DBCLICK | 双击列表框仅当列表框有 LBN _ NOTIFY 风格时传送此消息 |
| LBN _ ERRSPACE | 列表框不能分配足够的内存 |
| LBN _ KILLFOCUS | 列表框失去输入焦点 |
| LBN _ SETFOCUS | 列表框接收输入焦点 |
| LBN _ SELCANCEL | 取消当前列表框选择,当列表框有 LBS _ NOTIFY 风格时传送此消息 |
| LBN _ SELCHANGE | 列表框选择即将更改 |

用于操作列表框控件 CListBox 类的成员函数及其作用如表 9-4 所示。

表 9-4　列表框控件 CListBox 类的成员函数

| 函数 | 说明 |
| --- | --- |
| GetCount | 返回列表框中表项的数目 |
| GetHorizontalExtent | 返回列表框可滚动范围的宽度 |
| SetHorizontalExtent | 设置列表框的水平滚动宽度 |
| GetTopIndex | 返回列表框中第一个可见项的索引 |
| SetTopIndex | 指定特定的列表项为可见的 |
| GetItemRect | 返回列表框中当前显示列表项的边界矩形区域 |
| SelectString | 搜索并选择单项选择列表框中的指定字符串 |
| GetItemData | 返回与指定列表项有关的 32 位值 |
| SetItemDataPtr | 设置指向列表项的指针 |
| ItemFromPoint | 获取与指定点最近的列表项 |
| SetItemHeight | 设置列表项的高度 |
| GetItemHeight | 返回列表项的高度 |
| GetSel | 返回指定列表项的选择状态 |
| GetText | 复制列表项到缓冲区 |
| GetTextLen | 返回列表项的字节数 |
| SetColumnWidth | 设置多栏列表框的栏宽 |
| SetTabStops | 设置列表框中制表符的位置 |
| AddString | 添加字符串到列表框中 |
| DeleteString | 从列表框中删除字符串 |
| InsertString | 在列表框中的指定位置插入字符串 |
| ResetContent | 从当前目录添加文件名到列表框 |
| FindString | 搜索列表框中指定的字符串 |
| FindStringExact | 搜索列表框中第一个与指定字符串匹配的字符 |

4．编辑控件

编辑控件是一个矩形子窗口，可以用来输入文本信息。与编辑控件相关的常见消息如

表 9-5 所示。

表 9-5　与编辑控件相关的常见消息

| 消息 | 产生的原因 |
| --- | --- |
| EN _ ERRSPACE | 编辑控件不能分配足够的内存 |
| EN _ CHANGE | 改变编辑控件中的文本 |
| EN _ VSCROLL | 单击编辑控件的垂直滚动条 |
| EN _ KILLFOCUS | 编辑控件失去输入焦点 |
| EN _ SETFOCUS | 编辑控件接收输入焦点 |
| EN _ UPDATE | 编辑控件即将显示被更改的文本 |
| EN _ MAXTEXT | 当前插入超过编辑控件指定的字符数并已被截断 |
| ES _ AUTOHSCROLL | 要插入的字数超过编辑控件的宽度 |
| ES _ AUTOVSCROLL | 插入文本后导致总行数超过编辑控件的高度 |

编辑控件类 CEdit 的成员函数及其功能如表 9-6 所示。

表 9-6　编辑控件类 CEdit 的成员函数及其功能

| 函数 | 说明 |
| --- | --- |
| GetSel | 得到编辑控件中当前选择的起始和结束字符位置 |
| ReplaceSel | 用指定的文本替换编辑控件中的当前选择 |
| SetSel | 选择编辑控件中的字符范围 |
| Clear | 清除编辑控件中的当前选择 |
| Copy | 以 CF _ TEXT 格式复制编辑控件中的当前选择到剪贴板中 |
| Cut | 以 CF _ TEXT 格式删除编辑控件中的当前选择并将其复制到剪贴板中 |
| Paste | 以 CF _ TEXT 格式从剪贴板复制数据到编辑控件的当前位置 |
| Undo | 撤消最后的编辑控件操作 |
| CanUndo | 判断编辑控件操作是否可以撤消 |
| EmptyUndoButter | 重设或消除编辑控件的撤消标记 |
| GetModify | 判断编辑控件的内容是否已修改 |

续表

| 函数 | 说明 |
| --- | --- |
| SetModify | 设置或消除编辑控件的修改标记 |
| SetReadOnly | 设置编辑控件为只读状态 |
| GetPassWordChar | 得到编辑控件中显示的口令字符 |
| SetPassWordChar | 设置编辑控件中显示的口令字符 |
| GetFirstVisibleLine | 判断编辑控件中最顶端的可见行 |
| LineLength | 得到编辑控件的行长度 |
| LineScroll | 滚动编辑控件中指定行的文本 |
| LineFromChars | 得到包含指定字符索引的行号 |
| GetRect | 得到编辑控件的格式化区域 |
| LimitText | 限制可以输入到编辑控件中的文本长度 |
| GetLineCount | 得到多行编辑控件的行数 |
| GetLine | 从多行编辑控件中得到指定的文本行 |

5.组合框控件

组合框是由列表框和静态文本控件以及编辑控件组成的。

与组合框控件相关的消息如表 9-7 所示。

表 9-7　与组合框相关的消息

| 消息 | 产生的原因 |
| --- | --- |
| CBN _ SELCHANGE | 用户单击列表框或使用箭头键导致组合框的列表框选择即将被更改 |
| CBN _ CLOSEUP | 组合框的列表框已关闭 |
| CBN _ KILLFOCUS | 组合框失去输入焦点 |
| CBN _ SETFOCUS | 组合框接收输入焦点 |
| CBN _ SELENDOK | 用户选择一个列表项并按回车键 |
| CBN _ DBLOCK | 双击组合框中的列表项 |

续表

| 消息 | 产生的原因 |
| --- | --- |
| CBN _ DROPDOWN | 下拉组合框的列表框，但仅当组合框具有 CBS _ DROPDOWN 或 BS _ DROPDOWNLIST 风格时传送 |
| CBN _ SELENDCANCEL | 用户选择被取消 |
| CBN _ ERRSPACE | 组合框未分配到足够的空间 |
| CBN _ EDITCHANGE | 更改组合框中编辑控件部分的文本 |
| CBN _ EDITUPDATE | 编辑控件部分即将显示被更改的文本 |

6.滚动条控件

滚动条控件由滚动框和滚动框内的一个或两个滚动箭头组成，在 Windows 系列软件中使用得相当普通。与滚动条控件有关的 CScrollBar 类的成员函数如表 9-8 所示。

表 9-8 滚动条类 CScrollBar 的成员函数

| 函数 | 说明 |
| --- | --- |
| GetScrollPos | 得到滚动框的当前位置 |
| SetScrollPos | 设置滚动框的当前位置 |
| GetScrollRange | 得到指定滚动条的滚动范围 |
| SetScrollRange | 设置指定滚动条的滚动范围 |
| SetScrollInfo | 设置指定滚动条的有关的信息 |
| GetScrollInfo | 得到与指定滚动条有关的信息 |
| ShowScrollBar | 显示或隐藏滚动条 |
| GetScrollLimit | 得到滚动条的最大滚动位置 |
| EnableScrollBar | 启用或禁用一个或两个滚动箭头 |

## 9.2.2 附加的控件类

除 Windwos 标准控件类外，MFC 类库还提供其他控件类：BitmapButton(以位图作标签的按钮控件)、CCheckListBox(复选列表框)、CDragListBox (拖动列表框)、CToolBar(含有其他控件的工具栏)、CStatusBar(含有面板或字符的状态栏)以及 CDialogBar(从对话框模板资源创建的对话框)。

### 9.2.3 管理对话框控件的 MFC 函数

对话框类是从 CDialog 类继承的，而 CDialog 类又是从 CWnd 类继承的。CWnd 类提供了管理对话框和对话框控件的多个成员函数。

用于管理对话框控件的 CWnd 成员函数主要有 17 种，分别介绍如下：

(1) CheckDlgButton：用于选择或消除按钮以及更改按钮的状态。函数原型如下：

```
void CheckDlgButton( int nIDButton, UINT nCheck );
```

其中，参数 nIDButton 指定要修改的按钮。nCheck 指定要采取的动作，非 0，则放置选中标记到按钮中；为 0，则删除选中标记；为 2，按钮状态不确定。CheckDlgButton 传送 BM _ SETCHECK 消息给指定的按钮。

(2) DlgDirSelect：从文件/目录列表框中获取的当前选项，这里的选项可以是驱动器字母、文件名或目录名。函数原型为：

```
BOOL DlgDirSelect( LPTSTR lpString, int nIDListBox );
```

其中，lpString 参数为列表框中的当前选项。

(3) DlgDirListComboBox：往组合框的列表框中填充文件或目录表，即把消息 CB _ RESET-CONTENT 和 CB _ DIR 传送给组合框。函数原型为：

```
int DlgDirListComboBox( LPTSTR lpPathSpec, int nIDComboBox, int nIDStaticPath,
UINT nFileType );
```

(4) CheckRadioButton：选择给定的单选按钮并消除组内其他所有单选按钮，函数原型为：

```
void CheckRadioButton( int nIDFirstButton, int nIDLastButton, int nIDCheckBut-
ton );
```

其中，参数 nIDFirstButton 和 nIDLastButton 分别指定组内第一个和最后一个单选按钮，参数 nIDCheckButton 指定被选中的单选按钮。

(5) GetCheckedRaidioButton：返回指定组中当前被选中的单选按钮，函数原型为：

```
int GetCheckedRadioButton( int nIDFirstButton, int nIDLastButton );
```

(6) DlgDirList：往列表框中填充文件或目录，函数原型为：

```
int DlgDirList ( LPTSTR lpPathSpec, int nIDListBox, int nIDStaticPath, UINT
nFileType );
```

其中，参数 lpPathSpec 指向包含路径或文件名的字符串，参数 nIDListBox 指定列表框标识符，参数 nIDStaticPath 指定用于显示当前驱动器或目录的静态文本控件，参数 nFileType 指定要显示文件的属性。

(7) DlgDirSelectComboBox：从组合框的列表框中获取当前选择的文件或目录。函数原型为：

```
BOOL DlgDirSelectComboBox( LPTSTR lpString, int nIDComboBox );
```

其中,参数 lpString 为组合框的列表框中的当前选项。

(8) GetDlgItem:返回指向对话框中指定控件的指针,函数原型为:

```
CWnd * GetDlgItem(int nID);
```

其中,参数 nID 为指定的控件 ID。

(9) GetDlgItemText:获取在控件内显示的标题或文本,函数原型为:

```
int GetDlgItemText( int nID, LPTSTR lpStr, int nMaxCount ) const; int GetDlgItem-
Text( int nID, CString& rString ) const;
```

(10) GetDlgItemInt:返回指定控件中由文本表示的整数值,函数原型为:

```
UINT GetDlgItemInt( int nID, BOOL* lpTrans = NULL, BOOL bSigned = TRUE ) const;
```

(11) GetNextDlgGroupItem:返回指向一组控件中下一个控件的指针,函数原型为:

```
CWnd* GetNextDlgGroupItem( CWnd* pWndCtl, BOOL bPrevious = FALSE ) const;
```

其中,参数 pWndCtl 为指向当前控件的指针,参数 bPrevious 为 FALSE,表示返回下一个控件的指针。

(12) GetNextDlgTabItem:返回指向下一个或具有 WS_TABSTOP 风格的控件的指针,函数原型为:

```
CWnd* GetNextDlgTabItem( CWnd* pWndCtl, BOOL bPrevious = FALSE ) const;
```

参数 pWndCtl 为指向当前控件的指针,参数 bPrevious 为 FALSE,表示返回下一个控件的指针。

(13) IsDlgButtonChecked:判断按钮控件是否被选中,函数原型为:

```
UINT IsDlgButtonChecked(int nIDButton);
```

(14) IsDialogMessage:判断指定的消息是否为非模态对话框的消息,函数原型为:

```
BOOL IsDialogMessage( LPMSG lpMsg );
```

其中,参数 lpMsg 指向包含要检查消息的 MSG 结构。

(15) SendDlgItemMessage:传送消息给指定的控件,函数原型为:

```
SendDlgItemMessage( int nID, UINT message, WPARAM wParam = 0,
LPARAM lParam = 0 );
```

其中,参数 nID 为指定的控件 ID,参数 Message 为要传送的消息,参数 wParam 和 lParam 为与消息有关的附加消息。

(16) 函数 SetDlgItemInt:将整数转换为字符串并将其赋给控件,函数原型为:

```
void SetDlgItemInt( int nID, UINT nValue, BOOL bSigned = TRUE );
```

(17) SetDlgItemText:设置由对话框控件显示的文本,函数原型为:

```
void SetDlgItemText( int nID, LPCTSTR lpszString );
```

## 9.3 控 件 栏

控件栏包括工具栏、状态栏和对话栏。所有控件栏的基类都是 CControlBar。尽管 Visual C++为创建一个具备各种控件栏的应用程序提供了大量的帮助，但仍应该熟悉各种控件栏的编程，因为每一个应用程序都有自己独特的命令集。下面分别介绍各类控件栏的概念及编程要领。

### 9.3.1 工具栏

Visual C++提供两种创建工具栏的方法。一种是使用资源编辑器，创建过程如下：

(1) 创建工具栏资源。

(2) 构造 CToolBar 对象。

(3) 调用成员函数 Create 创建 Windows 工具栏并将其与 CToolBar 对象链接。

(4) 调用成员函数 LoadToolBar 装载工具栏资源。

另一种是直接调用成员函数，创建过程如下：

(1) 构造 CToolBar 对象。

(2) 调用成员函数 Create 创建 Windows 工具栏并将其与 CToolBar 对象链接。

(3) 调用成员函数 LoadBitMap 装载包含工具按钮图像的位图。

(4) 调用成员函数 SetButtons 设置按钮风格并使每个按钮与位图图像相关。

与工具栏相关的 CToolBar 类的成员函数如表 9-9 所示。

**表 9-9　CToolBar 类的成员函数**

| 函数 | 说明 |
|---|---|
| SetSizes | 设置按钮及位图的大小 |
| SetHeight | 设置工具栏的高度 |
| SetBitMap | 设置位图图像 |
| CommandToIndex | 返回指定命令 ID 的按钮索引 |
| GetItemID | 返回指定索引的按钮或分隔符的命令 ID |
| GetItemRect | 获取索引项的指定比例的显示区域 |
| GetButtonStyle | 获取按钮风格 |
| SetButtonStyle | 设置按钮风格 |
| GetButtonInfo | 获取按钮的 ID、风格和图像号 |

续表

| 函数 | 说明 |
| --- | --- |
| SetButtonInfo | 设置按钮的 ID、风格和图像号 |
| GetButtonText | 获取显示在按钮上的文本 |
| SetButtonText | 设置显示在按钮上的文本 |

### 9.3.2 状态栏

状态栏由输出面板或指示符组成。创建状态栏一般应遵循如下步骤：

(1) 构造 CStatusBar 对象。

(2) 调用成员函数 Create 创建状态栏窗口并将其与 CStatusBar 对象链接。

(3) 调用成员函数 SetIndicators 设置指示符的标识符为数组中相应元素指定的值，装载每个标识符指定的字符串资源并将字符串设为指示符文本。

与状态栏相关的 CStastusBar 类的成员函数如表 9-10 所示。

表 9-10 CStastusBar 类的成员函数

| 函数 | 说明 |
| --- | --- |
| CommandToIndex | 获取给定指示符 ID 的索引 |
| GetItemID | 获取给定索引的指示符 ID |
| GetItemRect | 获取给定索引的显示区域 |
| GetPaneInfo | 获取给定索引 ID、风格和宽度 |
| GetPaneStyle | 获取给定索引的指示符风格 |
| GetPaneText | 获取给定索引的指示符文本 |
| SetpaneInfo | 设置给定索引的 ID、风格和宽度 |

### 9.3.3 对话栏

对话栏是具有非模态对话框特性的工具栏。创建对话栏的过程如下：

(1) 在对话框编辑器中创建对话框模板资源。

(2) 从 Edit 菜单中选择 Properties 命令。

(3) 从属性对话框中选择 Style 并在 Style 框中选择 Child，在 Border 框中选择 None。

(4) 选择 More Styles 并清除 Visible 复选框。

(5) 选择 General 并清除 Caption 框。

(6) 构造 CDialogBar 对象。

(7) 调用成员函数 Create 创建对话栏窗口并将其与 CDialogBar 对象链接。

### 9.3.4　CControlBar 类

CControlBar 类是控件类 CStatusBar、CToolBar、CDialogBar 和 COleResizeBar 的基类。该类继承自 CWnd 类，因此它也是一个窗口类，具有与窗口类相似的特征，但是它又具有自己的新的成员函数，这些成员函数的名称如表 9-11 所示。

表 9-11　CControlBar 类的成员函数

| 函数 | 说 明 |
|---|---|
| GetBarStyle | 获取控件栏风格设置 |
| SetBarStyle | 设置控件栏风格 |
| GetCount | 返回控件栏中非 HWND 元素数目 |
| GetDockingFrame | 返回指向控件栏被船坞化的边框指针 |
| iSFloating | 判断控件栏是否为浮动的控件栏 |
| CalcFixedLayout | 返回控件栏的大小 |
| CalcDynamicLayout | 返回动态控件栏的大小 |
| EnableDocking | 使控件栏为船坞化或浮动 |

## 9.4　对话框编程

编写含有对话框的应用程序和编写其他应用程序的过程相似，但对于其中的对话框部分的设计则应遵循前面所介绍的方法和步骤。这一节通过实例来介绍具有对话框的应用程序的设计步骤。此例子利用上一章的例子，但是通过对话框改变绘图工具的风格。

### 9.4.1　创建对话框模板资源

要建立对话框首先需创建对话框模板，它是对话框的显示外观。创建对话框模板可以利用集成的对话框编辑器，在本例中，我们改变缺省画笔的粗细和线型，具体方法如下：

(1) 从 Insert 菜单中选择 Resource 命令，弹出 Insert Resource 对话框，如图 9-1 所示。

(2) 在 Resource type 列表中选择 Dialog，单击 New 按钮进入对话框编辑器，如图 9-2 所示。

(3) 按 Alt + Enter 组合键，弹出 Dialog Properties 对话框，在 General 选项卡的 Caption 文本

框中将标题改为“画笔”,如图 9-4 所示。

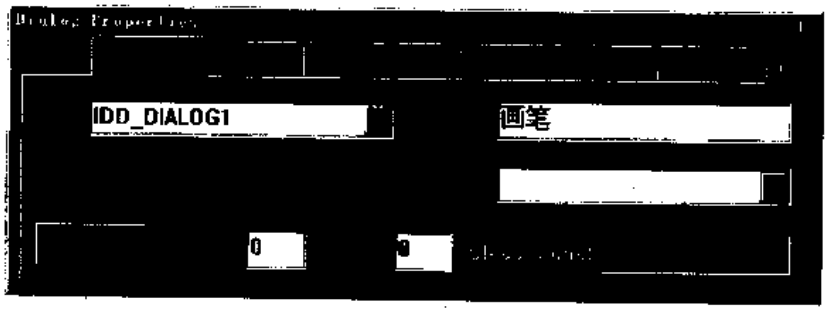



图 9-4 Dialog Properties 对话框

(4) 在对话框中添加控件并改变控件的属性。在本例中添加了一个编辑控件和一个组控件,其中组控件中包含了三个单选按钮。这里的编辑控件用于接受用户输入的整型值,此值用来设置画线时所采用的线宽,三个单选按钮用来设置线型,它们分别是实线、点线和虚线。

设计完成后测试该对话框,其显示结果如图 9-5 所示。



图 9-5 “画笔”对话框运行结果

(5) 制表顺序是指在对话框中按 Tab 键将输入焦点从一个控件移到另一个控件的顺序。改变制表顺序,从 Layout 菜单中选择 Tab Order 命令或按 Ctrl + D 组合键,然后,依次单击各个控件,于是控件被单击的顺序即为新的制表顺序,然后,按 Enter 键结束 Tab Order 方式。

(6) 从 File 菜单中选择 Save 命令保存对话框。

### 9.4.2 创建对话框类

应用 Class Wizard 可以创建管理对话框的对话框类。创建过程如下:

(1) 从 View 菜单中选择 ClassWizard 命令,弹出 Adding a Class 对话框,如图 9-6 所示。

(2) 单击 OK 按钮,弹出 New Class 对话框(如图 9-7 所示)。在 Name 文本框中输入对话框类名称 CPenDLG,从 Base Class 下拉列表中选择用于继承对话框类的基类为 CDialog,从 Dialog ID 下拉列表框选择资源 ID 的 IDD _ DIALOG1。

(3) 单击 OK 按钮,对话框类的名字将显示在 MFC ClassWizard 对话框的 Class Name 下拉列表中。



图 9-6　Adding a Class 对话框

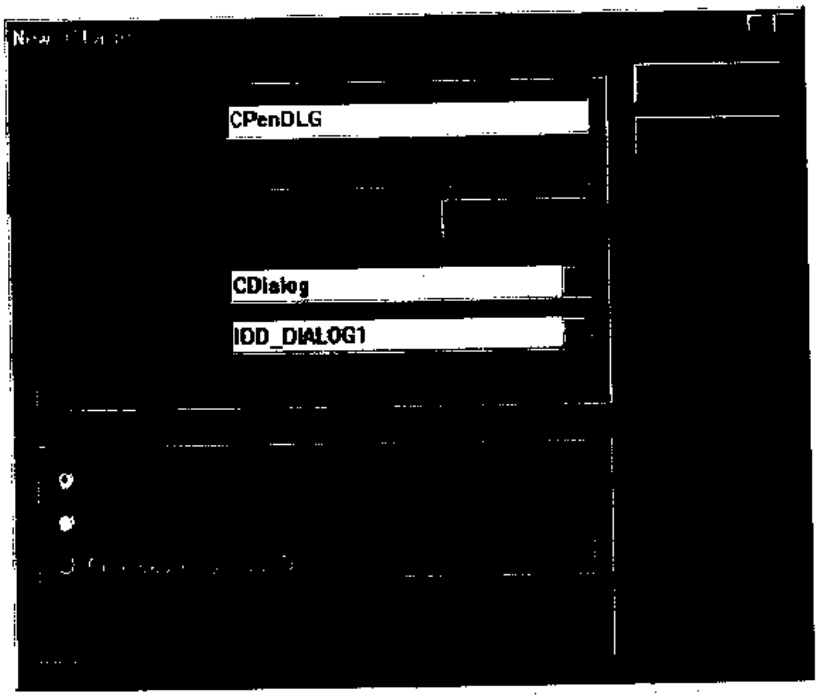



图 9-7　New Class 对话框

### 9.4.3　定义成员函数

Class Wizard 可以为对话框中的每个控件定义该控件对应类的数据成员。显示对话框时 MFC 自动将数据成员的值传送给相应的控件。

定义成员函数的方法如下：

(1) 在 MFC ClassWizard 对话框中，切换到 Member Variables 选项卡，如图 9-8 所示。Control IDs 列表框中显示了可以赋给数据成员的每个控件的控件 ID。

(2) 为编辑控件定义数据成员。从 Control IDs 列表框中选择 IDC _ LINEWIDTH _ INPUT 后，单击 Add Varibles，弹出 Add Member Variable 对话框，如图 9-9 所示。

(3) 在 Member variable name 文本框中输入 m _ pWidth，在 Variable type 下拉列表框中选择数据类型为 int。

(4) 单击 OK 按钮，返回 MFC ClassWizard 对话框。此时，两个新的文本框显示在对话框底部。在 Minimum Value 文本框中输入 1，在 Maximum Value 文本框中输入 6，如图 9-10 所示。

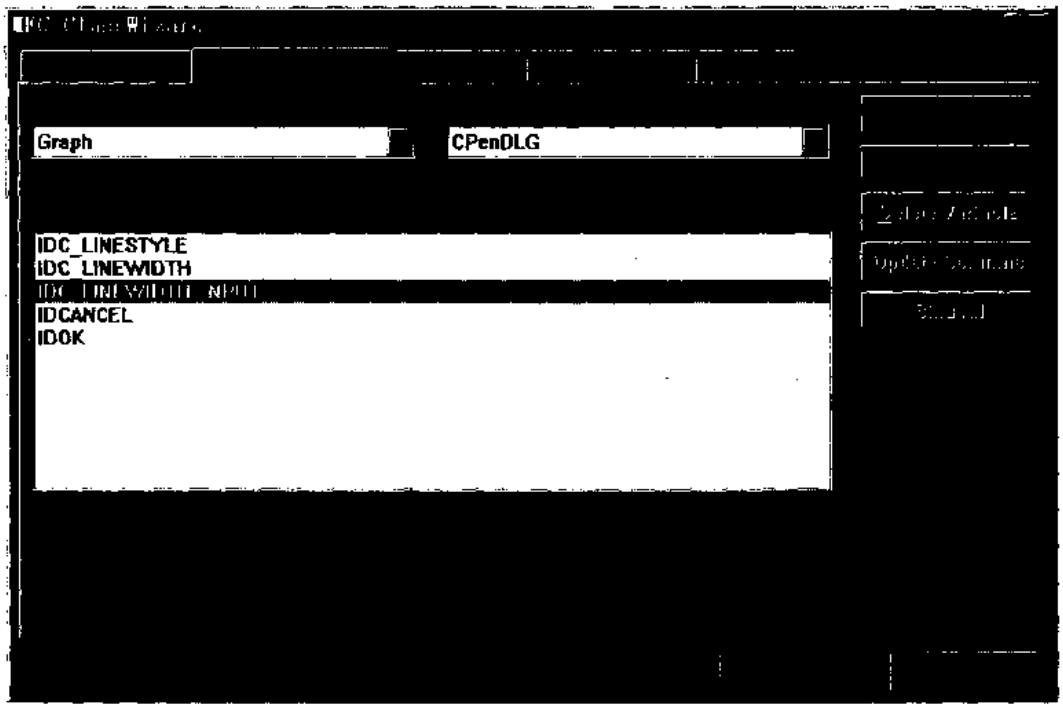



图 9-8 Member Variables 选项卡

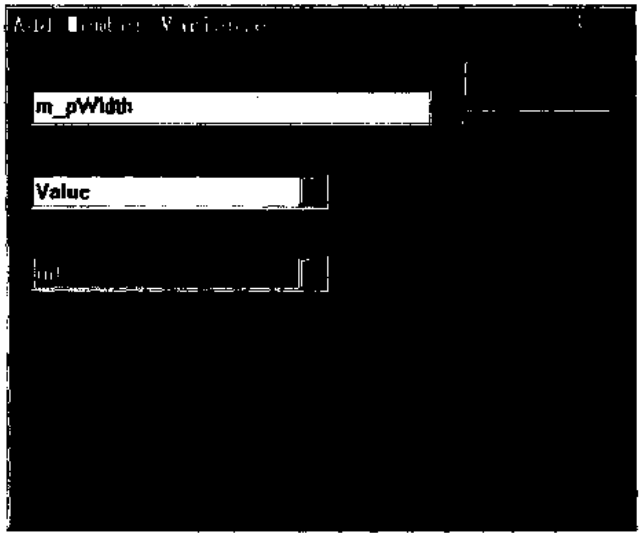



图 9-9 Add Member Variable 对话框

(5) 为管理编辑控件定义一个成员对象。从 Control IDs 列表框中选择 IDC _ LINEWIDTH _ INPUT,单击 Add Variables 按钮,弹出 Add Member Variable 对话框。在 Member variable name 文本中输入 m _ pWidthEdit,在 Category 下拉列表框中选择 Control,单击 OK 按钮,返回 MFC ClassWizard 对话框,如图 9-11 所示。

(6) 定义单选按钮组的数据成员。从 Control IDs 列表框中选择 IDC _ LINESTYLE,单击 Add Variable 按钮,弹出 Add Member Variable 对话框。在 Member variable name 文本框中输入 m _ pStyle,在 Variable type 列表框中选择 int,如图 9-12 所示。

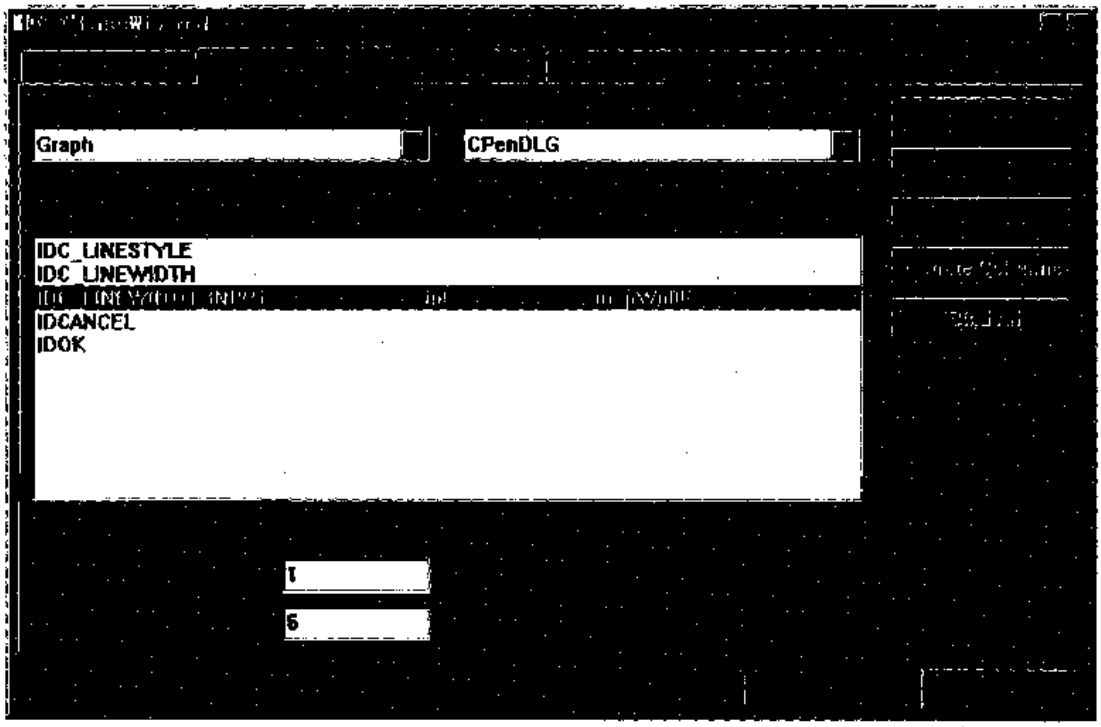



图 9-10　定义数据成员的有效范围

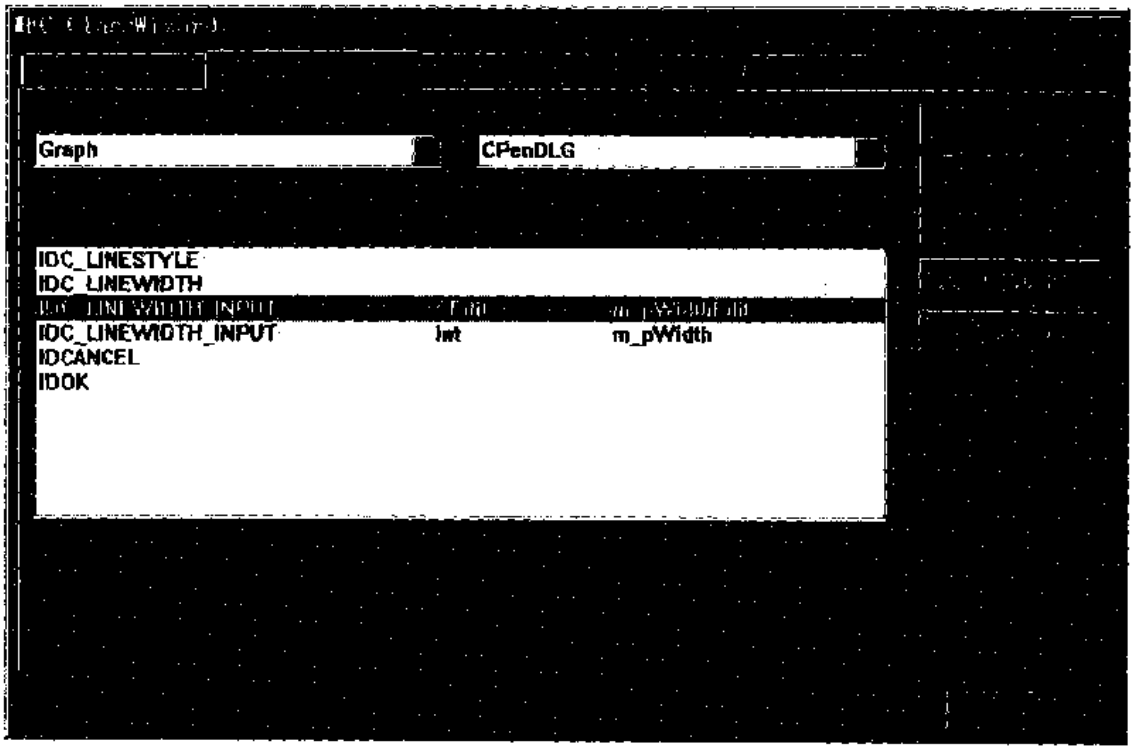



图 9-11　为编辑控件定义成员对象

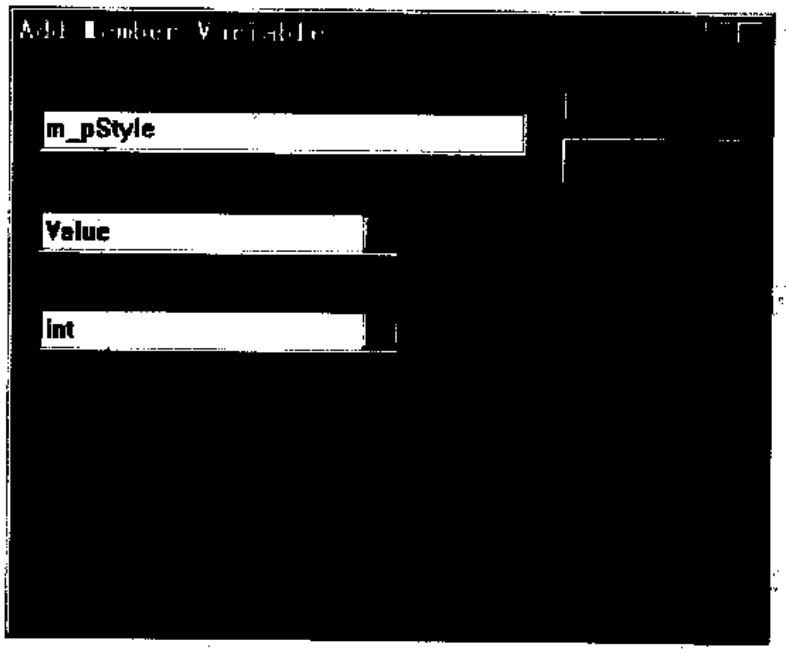



图 9-12 为单选按钮组定义数据成员

### 9.4.4 定义消息处理函数

如果要使对话框处理 Windows 消息，必须重设适当的消息处理函数，具体操作步骤如下：

(1) 从 View 菜单中选择 ClassWizard 命令，弹出 MFC Class Wizard 对话框。

(2) 切换到 Message Maps 选项卡。

(3) 从 Class Name 下拉列表中选择对话框类 CPenDLG。

(4) 定义初始化和重画对话框的消息处理函数。

在 Object IDs 列表框选择 CPenDlg。在 Messages 列表框中选择 WM _ INITDIALOG，单击 Add Function 按钮，建立响应消息 WM _ INITDIALOG 的处理函数，即对话框初始化时将调用的处理函数 OnInitDialog。在 Messages 列表框中选择 WM _ PAINT 并单击 Add Function 按钮，建立响应消息 WM _ PAINT 的处理函数，重画对话框时将调用处理函数 OnPaint。

(5) 为每个单选按钮提供消息处理函数。单击某个单选按钮时，将选中该单选按钮，而清除其他单选按钮，对于此过程不必编程，但当单击 OK 按钮结束对话框时，应当能够了解单选按钮的状态，以确定绘图时所使用的线型。因此应该对每个单选按钮编写消息处理函数，具体方法为：在 Object IDs 列表框中选择 IDC _ DASH，在 Messages 列表框中选择 BN _ CLICKED，单击 Add Function 按钮。依此方法添加其他单选按钮的消息处理函数。

(6) 定义当用户改变编辑控件的内容时接收控件内容的处理函数。在 Object IDs 列表框中选择 IDC _ LINEWIDTH _ INPUT，在 Messages 列表框中选择 EN _ CHANGE 并单击 Add Function 按钮。图 9-13 显示了添加所有的消息处理函数后的 MFC ClassWizard 对话框。

(7) 单击 OK 按钮，Class Wizard 将向对话框类的头文件中插入函数声明，向实现文件中插入完整的函数定义和实现框架，并更新类的消息映射以包含多个消息处理函数。

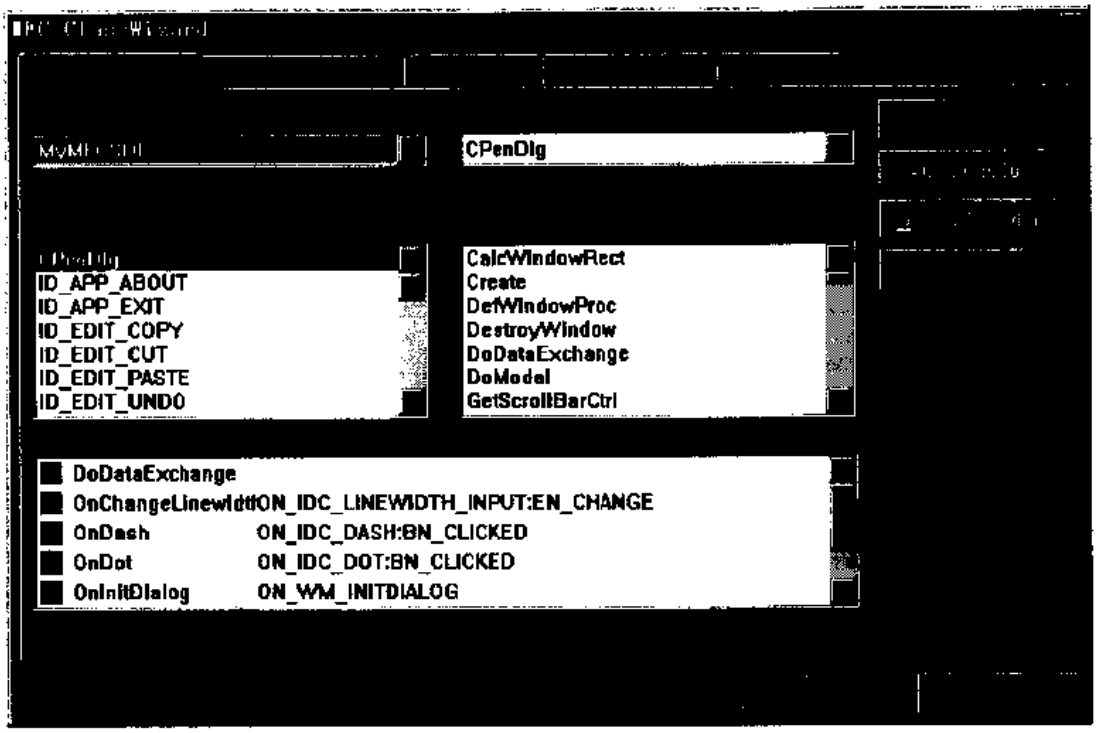



图 9-13 用 Class Wizard 添加消息处理函数

### 9.4.5 修改对话框类代码

定义成员变量和消息处理函数后，应修改对话框类，增加其具体实现代码，具体操作步骤如下：

(1) 在对话框类的头文件开始处添加枚举型定义(用于表示对话框中单选按钮的序号)：

```
enum{PEN_STYLE_SOLID,PEN_STYLE_DASH,PEN_STYLE_DOT};
```

(2) 在对话框类头文件的末尾添加数据成员：

```
m_pRectSample;
protected:
RECT m_pRectSample;
```

添加此类型的成员变量的目的是为了重画编辑框控件。

(3) 在对话框类的成员函数 OnInitDialog 中添加以下代码：

```
GetDlgItem(IDC_LINEWIDTH_INPUT)->GetWindowRect(&m_pRectSample);
                                    //确定编辑控件的矩形区域
ScreenToClient(&m_pRectSample);//转换为客户坐标
m_pWidthEdit.LimitText(1);//限制长度为1个字符
```

(4) 完成控件通知的消息处理代码。在消息处理函数 OnSolid 中添加以下代码：

```
if(IsDlgButtonChecked(IDC_SOLID)){
m_pStyle=PEN_STYLE_SOLID;
InvalidateRect(&m_pRectSample);
UpdateWindow();
}
```

在消息处理函数 OnDash 中添加以下代码：

```
if(IsDlgButtonChecked(IDC_DASH)){
m_pStyle=PEN_STYLE_DASH;
InvalidateRect(&m_pRectSample);
UpdateWindow();
}
```

在消息处理函数 OnDot 中添加以下代码：

```
if(IsDlgButtonChecked(IDC_DOT)){
m_pStyle=PEN_DOT;
InvalidateRect(&m_pRectSample);
UpdateWindow();
}
```

在消息处理函数 OnChangeLinewidthInput 中添加以下代码：

```
int tmp;
tmp=(int)GetDlgItemInt(IDC_WIDTH);
if(tmp>0)&&(tmp<7)){
m_pWidth=tmp;
InvalidateRect(&mpRectSample);
UpdateWindow();
}
```

(5) 在对话框类的成员函数 OnPaint 中添加以下代码：

```
CPen NewPen;//声明画笔对象
CPen &pOldpen;//保存原画笔的指针
//初始化画笔对象
switch(m_pStyle){
case PEN_STYLE_SOLID:
NewPen.CreatePen(PS_SOLID,m_pWidth,RGB(0,0,0));
break;
case PEN_STYLE_DASH:
NewPen.CreatePen(PS_DASH.m_pWidth,RGB(0,0,0));
break;
default:
```

```
NewPen.CreatePen(PS_SOLID,m_pWidth,RGB(0,0,0));
break;
}
//将画笔设置为没有文本对象
pOldpen = dc.SelectObject(&NewPen);
//使用画笔在组框中画线
dc.MoveTo(m_pRectSample.left + 10,m_pRectSample.top + 25);
dc.lineTo(m_pRectSample.right - 10,m_pRectSample.bottom - 25);
//恢复原先的画笔对象
dc.SelectObject(pOldPen);
```

### 9.4.6　创建对话框类对象并显示对话框

编写完对话框类的代码后，应创建对话框类对象并显示对话框，具体操作可分为如下三步：

第一步，在主菜单中添加菜单命令以便打开对话框，操作步骤如下：

(1) 在项目工作区窗口中，切换到 Resource View 面板。

(2) 双击菜单资源 IDR_MAINFRAME，打开菜单编辑器窗口。

(3) 在“查看”菜单前插入“选项”菜单。

(4) 在“选项”菜单中添加菜单选项“画笔”和“颜色”。

(5) 在加速键编辑器中建立相应的加速键表条目，并赋给与菜单选项相同的 ID 号。这一步是可选的，如果没有，则无法使用加速键。

第二步，使用 ClassWizard 添加菜单命令的消息处理函数，操作步骤如下：

(1) 从 View 菜单中选择 ClassWizard 命令，弹出 MFC ClassWizard 对话框。

(2) 切换到 Message Maps 选项卡。

(3) 从 Class name 下拉列表框中选择文档类 CMyDrawDoc，使消息处理函数被添加到应用程序的文档类中。

(4) 在 Object IDs 列表框中选择 ID_OPTION_COLOR。在 Messages 列表框中选择 COMMAND 并单击 Add Function 按钮。然后，在 Messages 列表框中选择 UPDATE_COMMAND_UI，并单击 Add Function 按钮。

(5) 用同样的方法定义画笔命令的处理函数并更新处理函数。

(6) 单击 OK 按钮，ClassWizard 向文档类的头文件中插入函数声明，向实现文件中插入完整的函数定义和实现框架，并更新类的消息映射以包含新的消息处理函数。

第三步，修改文档类的源代码，操作步骤如下：

(1) 在文档类的头文件中添加以下数据成员：

```
public:
int m_pWidth;
int m_pStyle;
```

```
COLORREF m_pColor;
```

(2) 在文档类的构造函数中添加以下初始化数据成员的代码：

```
m_pWidth=1;
m_Style=PEN_STYLE_SOLID;
m_pColor=RGB(0,0,0);
```

(3) 在文档类的实现文件中添加以下代码：

```
#include "PenDlg.h"
```

(4) 在消息处理函数 OnOptionPen 中添加以下代码：

```
CPenDLG PenDialog;//声明对话框类对象
//初始化对话框类对象的数据成员
PenDLG.m_pWidth=m_pWidth;
PenDLG.m_pStyle=m_pStyle;
//显示对话框
if(PenDialog.DoModal()==IDOK){
//保存对话框内设置的画笔宽度和风格
m_pWidth=PenDialog.m_pWidth;
m_pStyle=PenDialog.m_pStyle;
```

(5) 在文档类的头文件中将 Myline 类重新声明为：

```
class CMyLine:public CObject{
protected:
int m_StartX;//直线的起始坐标
int m_StartY;
int m_EndX;//直线的终点坐标
int m_EndY;
int m_pWidth;//画笔宽度
int m_PStyle;//画笔风格
COLORREF m=PColor;//画笔颜色
CMyline ()//缺省的构造函数
DECLARE_SERIAL(CMyline)
public:
CMyline(int StartX,int StartY,int EndX,int EndY,int Width,int Style,COLORREF
Color);
//构造函数
void Draw line(CDC * PDC);//画直线;
virtual void Serialize(CArchive&ar);//CMyline 对象的串行化
};
```

(6) 在文档类的实现文件中将 Myline 类的构造函数修改为：

```
CMyline::Cmyline(int StartX,int StartY, int EndX,int EndY,int Width,int Style,
COLORREF Color){
    m_StartX = StartX;
    m_StartY = StartY;
    m_EndX = EndX;
    m_EndY = EndY;
    m_pWidth = Width;
    m_pStyle = Style;
    m_pColor = Color;
    };
```

(7) 在文档类的实现文件中将 Myline::Drawline 修改为：

```
void CMyline::Drawline(CDC * PDC){
CPen NewPen;
CPen * poldPen;
//根据用户设置的画笔风格、宽度和颜色创建画笔对象
switch (m_pStyle){
case PNE_STYLE_SOLID:
NewPen.CreatePen(PS_SOLID,m_pWidth,m_pColor);
break;
case PEN_STYLE_DASH:
NewPen.CreatePen(PS_DASH,m_pWidth,m_pColor);
break;
case PEN_STYLE_DOT:
NewPen.CreatePen(PS_DOT,m_pWidth,m_pColor);
break;
}
pOldPen = PDC->SelectObject(&NewPen);
PDC->MoveTo(m_StartX,m_StartY);
PDC->LineTo(m_EndX,m_EndY);
PDC->SelectObject (Pold Pen);
}
```

(8)在文档类的头文件中将 MyDrawDOC::Addline 修改为：

```
void Addline (int StartX,int StartY,int EndX,int Endy,int Width,int Style,COL-
ORREF Color);
```

(9)在文档类的实现文件中将 Myline::Serialize 修改为：

```
void CMyline::Serialize(CArchive&ar){
if(ar.IsString())
ar<<(WORD)m_StartX<<(WORD)m_StartY<<(WORD)m_EndX<<(WORD)m_EndY<<
```

```
(WORD)m_pWidth<<(WORD)m_pStyle<<m_pColor;
    else
    ar>>(WORD&)m_StartX>>(WORD &)m_StartY>>(WORD&)m_EndX>>(WORD &)m_
EndY>>(WORD&)m_pWidth>>(WORD&)m_pStyle>>m_pColor;
    }
```

(10)在视图类的实现文件中将 CMyDrawView::OnMouseMove 修改为:

```
    void CMyDrawView::OnMouse Move(UINT nFlags,CPoint point){
    //TODO:在此处添加消息处理代码或调用缺省代码
    CClient DC dc(this);//声明没有文本对象
    OnPrepareDC(&dc);//调整视口原点
    dc.Dptolp(&point);//将设备坐标转换为逻辑坐标
    SetCursor(m_Hcursor);
    if(m_Draw){
    CMyDrawDoc * PDoc=GetDocument();
    CPen NewPen;
    CPen * pOldPen;
    //根据画笔风格、颜色和宽度创建画笔对象
    switch(pDOC->m_pStyle){
    case PEN_STYLE_SOLID:
    NewPen.CreatePen(PS_SOLID,PDOC->m_pWidth,PDOC->m_pColor);
    break;
    case PEN_STYLE_DASH:
    NewPen.CreatePen(PS_DASH,PDOC->m_pWidth,PDOC->m_pColor);
    break;
    case PEN_STYLE_DOT:
    NewPen.CreatePen(PS_DOT,PDOC->m_pWidth,PDOC->m_pColor);
    break;
    }
    pOldPen=dc.SelectObject(&NewPen);
    dc.SetRoR2(R2_NOT);
    dc.MoveTo(m_pOrigin);
    dc.LineTo(m_POld);
    dc.MoveTo(m_pOrigin);
    dc.lineTo(point);
    m_pOld=point;
    dc.SelectObject(pOldPen);
    }
    CSCrollView::OnMouseMove(nFlays,point);
    }
```

(11)在文档类的实现文件中将 CMyDrawDoc::Addline 修改为:

```
void CMyDrawDoc::Addline(int StartX,StartY,int EndX,int EndY, int Width,int
Style,COLORREF Color){
    CMyline * pMyLine = New CMyLine(StartX,StartY,EndX,EndY,Width,Style,Color);
    m_ObArray.Add (pMyLine);
    }
```

(12) 在视图类的实现文件中将 CMyDrawline::OnLButtonUp 修改为:

```
    }
        CView::OnLButtonUp(nFlags, point);

        CClientDC dc(this);
        OnPrepareDC(&dc);
        dc.DPtoLP(&point);
        if(m_Draw){
            m_Draw=0;//表示鼠标已释放
            ReleaseCapture();//释放鼠标并恢复正常输入
            ClipCursor(NULL);//使光标可以移动到屏幕任意位置
        }

        //释放鼠标左键后,画直线结束,应将所画直线添加到 CObarry 对象中
        CMyMFCSDIDoc * pDOC = GetDocument();
        pDOC->AddLine(m_pOrigin.x,m_pOrigin.y, point.x,point.y,
                    pDOC->m_pWidth,pDOC->m_pStyle,pDOC->m_pColor);

    }
```

(13)在视图类的实现文件中添加以下语句:

```
#include "PenDlg.h"
```

(14)在应用程序类的成员函数 InitInstance 中添加以下代码:

```
SetDialogBkColor(RGB(255,255,255),RGB(0,0,0));
```

### 9.4.7　创建并运行应用程序

创建并运行应用程序的方法如下:

(1) 选择 Build 菜单中的 Build MyDraw.exe 命令创建程序。

(2) 应用 Build 菜单的 Execute MyDraw.exe 命令运行程序。

MyDraw 程序的主菜单如图 9-14 所示。其中选项菜单中包含画笔和颜色两个菜单项。选择“画笔”命令将弹出“画笔”对话框,从中可以设置画笔风格和宽度,参见图 9-5 所示。选择“颜色”命令可弹出“颜色”对话框,从中可以设置画笔颜色。在此例中我们暂时还没有实

现“颜色”对话框，读者可依照前面介绍的画笔对话框编程的步骤补充“颜色”对话框的实现代码。

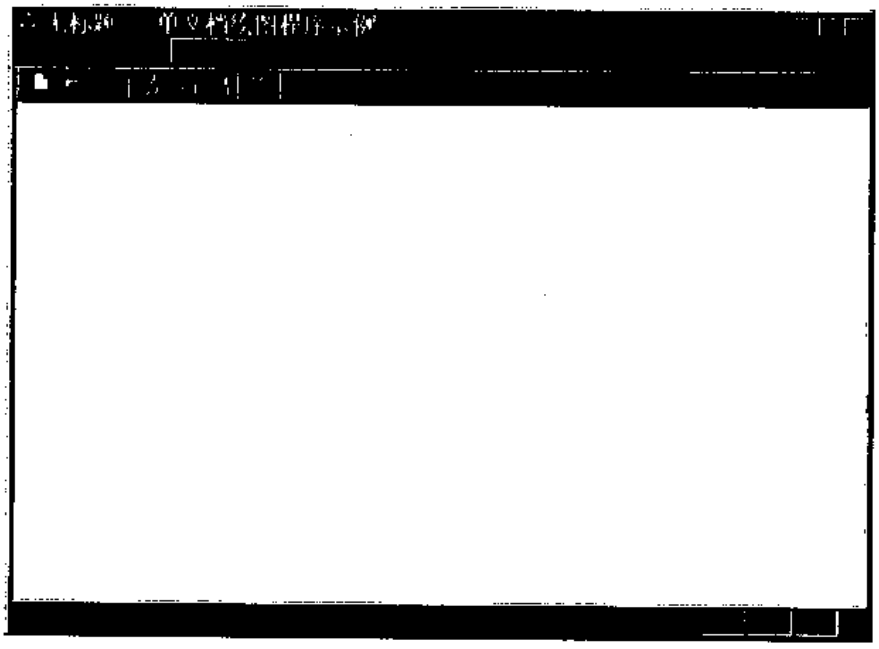


图 9-14　MyDraw 程序的主窗口

工具栏由 8 个工具按钮组成，当鼠标指针在工具栏某个按钮上停留时，工具栏会自动的弹出一个小的文本窗口，显示该按钮相应的说明。

### 9.4.8　完整的源程序清单

为了使读者了解开发过程全貌，下面列出此例的主要源程序代码。这些程序包括应用程序类定义及其实现文件、应用程序文档类定义及其实现文件、应用程序视图类定义及其实现文件、对话框类定义及其实现文件以及应用程序主窗口类定义及其实现文件。此程序已调试通过，可以运行。但是我们特别提醒读者，不要单纯依照此示例程序来开发带有对话框的应用程序，而应着重理解本章前三节关于对话框程序设计的原理。

**程序清单 9-1：应用程序类头文件**

```
//MyMFCSDI.h:应用程序类 CMyMFCSDIApp 定义头文件
//

#if ! defined(AFX_MYMFCSDI_H__4E2E9AB9_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
#define AFX_MYMFCSDI_H__4E2E9AB9_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
```

```
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
#endif

#include "resource.h"

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIApp:
// See MyMFCSDI.cpp for the implementation of this class
//

class CMyMFCSDIApp : public CWinApp
{
public:
        CMyMFCSDIApp();

// Overrides
        // ClassWizard生成的虚拟函数重载
        //{{AFX_VIRTUAL(CMyMFCSDIApp)
        public:
        virtual BOOL InitInstance();
        //}}AFX_VIRTUAL

// 实现
        //{{AFX_MSG(CMyMFCSDIApp)
        afx_msg void OnAppAbout();
                //ClassWizard在此处添加或删除成员函数
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};


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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 在此处添加附加的声明语句

#endif // !defined(AFX_MYMFCSDI_H__4E2E9AB9_6799_11D2_8DE6_B31A29E86F7F__
INCLUDED_)
```

**程序清单 9-2:应用程序 CMyMFCSDIAPP 类的实现文件**

```
// MyMFCSDI.cpp :定义此应用程序的类的行为
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MainFrm.h"
#include "MyMFCSDIDoc.h"
#include "MyMFCSDIView.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIApp

BEGIN_MESSAGE_MAP(CMyMFCSDIApp, CWinApp)
        //{{AFX_MSG_MAP(CMyMFCSDIApp)
        ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
                // ClassWizard 在此处添加或删除映射宏
        //}}AFX_MSG_MAP
        //基于文档命令的标准文件
        ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
        ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
        // 标准打印设置命令
        ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIApp 构造函数

CMyMFCSDIApp::CMyMFCSDIApp()
{
        //在此处添加构造函数代码
}

/////////////////////////////////////////////////////////////////////////////
```

```
// 唯一一个 CMyMFCSDIApp 对象代码

CMyMFCSDIApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIApp 初始化

BOOL CMyMFCSDIApp::InitInstance()
{
	AfxEnableControlContainer();

	// 标准初始化

# ifdef _AFXDLL
	Enable3dControls();	//当在共享的 DLL 中使用 MFC 时调用此语句
# else
	Enable3dControlsStatic(); //当静态链接到 MFC 时调用此语句
# endif

	SetRegistryKey(_T("Local AppWizard - Generated Applications"));

	LoadStdProfileSettings(); // Load standard INI file options (包括 MRU)

	//注册应用程序的文档模板
	//文档模板用作文档、边框窗口和视图的链接

	CSingleDocTemplate * pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CMyMFCSDIDoc),
		RUNTIME_CLASS(CMainFrame),	// 主 SDI 边框窗口
		RUNTIME_CLASS(CMyMFCSDIView));
	AddDocTemplate(pDocTemplate);

	// 分析命令行
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// 解释命令行中的命令
	if (! ProcessShellCommand(cmdInfo))
		return FALSE;
```

```
        SetDialogBkColor(RGB(255,255,255),RGB(0,0,0));

        // 显示并更新窗口
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();

        return TRUE;
}


/////////////////////////////////////////////////////////////////////////////
// 用作 App About 的 CAboutDlg 对话框

class CAboutDlg : public CDialog
{
public:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg)
        enum { IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard 生成的虚拟函数重载
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);
        //}}AFX_VIRTUAL

// 实现方式
protected:
        //{{AFX_MSG(CAboutDlg)
                // 无消息处理程序
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
```

```
}

void CAboutDlg::DoDataExchange(CDataExchange * pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
        //{{AFX_MSG_MAP(CAboutDlg)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// 运行对话框
void CMyMFCSDIApp::OnAppAbout()
{
        CAboutDlg aboutDlg;
        aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIApp 消息处理程序
```

**程序清单 9-3:应用程序文档类的定义文件**

```
// MyMFCSDIDoc.h : CMyMFCSDIDoc 类的接口
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__
INCLUDED_)
#define AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__INCLUDED
_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CMyLine:public CObject //声明直线类 CMyLine
```

```
{
protected:
        int m_StartX;//直线起点的 X 坐标
        int m_StartY;//直线起点的 Y 坐标
        int m_EndX; //直线终点的 X 坐标
        int m_EndY; //直线终点的 Y 坐标

public:
        int m_pWidth;//画笔粗细
        int m_pStyle;//画笔风格
        COLORREF n_pColor;//画笔颜色
//直线类的构造函数
        CMyLine(int StartX,int StartY,int EndX,int EndY,int Width,int Style,
COLORREF Color);
        void DrawLine(CDC * pDC);//直线的成员函数 DrawLine 用来在视图中绘制直线
        CMyLine();//直线类的缺省构造函数

        DECLARE_SERIAL(CMyLine)

        virtual void Serialize(CArchive & ar);

};

class CMyMFCSDIDoc : public CDocument
{

public:

        int m_pWidth;//画笔粗细
        int m_pStyle;//画笔风格
        COLORREF m_pColor; //画笔颜色

protected:
        CMyMFCSDIDoc();
        DECLARE_DYNCREATE(CMyMFCSDIDoc)

// 特性
protected:
        CObArray m_ObArray;

public:
```

```
        void AddLine (int StartX, int StartY, int EndX, int EndY, int Width, int
Style,COLORREF Color);
        CMyLine * GetLine(int indeX);
        int GetTotalLine();
        virtual void DeleteContents();

    // 操作
    public:

    // 重载
        // ClassWizard 生成的虚拟函数重载
        //{{AFX_VIRTUAL(CMyMFCSDIDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void Serialize(CArchive& ar);
        //}}AFX_VIRTUAL

    // 实现方式
    public:
        virtual CMyMFCSDIDoc();
    #ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
    #endif

    // 生成消息映射函数
    protected:
        //{{AFX_MSG(CMyMFCSDIDoc)
        afx_msg void OnEditUndo();
        afx_msg void OnUpdateEditUndo(CCmdUI * pCmdUI);
        afx_msg void OnOptionColor();
        afx_msg void OnUpdateOptionColor(CCmdUI * pCmdUI);
        afx_msg void OnOptionPen();
        afx_msg void OnUpdateOptionPen(CCmdUI * pCmdUI);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };

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

    //{{AFX_INSERT_LOCATION}}
```

```
// Microsoft Visual C++ 将添加附加的声明语句

#endif // ! defined(AFX_MYMFCSDIDOC_H__4E2E9ABF_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
```

**程序清单 9-4:应用程序文档类实现文件**

```
// MyMFCSDIDoc.cpp : CMyMFCSDIDoc 类的实现文件

#include "stdafx.h"
#include "MyMFCSDI.h"
#include "MyMFCSDIDoc.h"
#include "PenDLG.h"

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


/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIDoc


BEGIN_MESSAGE_MAP(CMyMFCSDIDoc, CDocument)
        //{{AFX_MSG_MAP(CMyMFCSDIDoc)
        ON_COMMAND(ID_EDIT_UNDO, OnEditUndo)
        ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, OnUpdateEditUndo)
        ON_COMMAND(ID_OPTION_COLOR, OnOptionColor)
        ON_UPDATE_COMMAND_UI(ID_OPTION_COLOR, OnUpdateOptionColor)
        ON_COMMAND(ID_OPTION_PEN, OnOptionPen)
        ON_UPDATE_COMMAND_UI(ID_OPTION_PEN, OnUpdateOptionPen)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

IMPLEMENT_SERIAL(CMyLine,CObject,1)
IMPLEMENT_DYNCREATE(CMyMFCSDIDoc, CDocument)
//直线类 CMyLine 的构造函数
CMyLine::CMyLine(int StartX,int StartY,int EndX,int EndY,int Width,int Style,
COLORREF Color)
```

```
}
        m_StartX = StartX;
        m_StartY = StartY;
        m_EndX = EndX;
        m_EndY = EndY;
        m_pWidth = Width;
        m_pStyle = Style;
        m_pColor = Color;
}

//画直线函数
void CMyLine::DrawLine(CDC * pDC){
        CPen NewPen;
        CPen * pOldPen;
        // 根据用户设置的画笔风格、粗细和颜色创建画笔对象
        switch(m_pStyle){
        case PEN_STYLE_SOLID:
        NewPen.CreatePen(PS_SOLID,m_pWidth,m_pColor);
        break;
        case PEN_STYLE_DASH:
        NewPen.CreatePen(PS_DASH,m_pWidth,m_pColor);
        break;
        case PEN_STYLE_DOT:
        NewPen.CreatePen(PS_DOT,m_pWidth,m_pColor);
        break;
        default:
        NewPen.CreatePen(PS_SOLID,m_pWidth,m_pColor);
        break;
        }
        pOldPen = pDC->SelectObject(&NewPen);
        pDC->MoveTo(m_StartX,m_StartY);
        pDC->LineTo(m_EndX,m_EndY);
        pDC->SelectObject(pOldPen);
};

//添加直线对象的成员函数
void CMyMFCSDIDoc::AddLine (int StartX,int StartY,int EndX,int EndY,int Width,
int Style,COLORREF Color)
{
        CMyLine * pMyLine = new CMyLine(StartX,StartY,EndX,EndY,Width,Style,
Color);
```

```
        m_ObArray.Add(pMyLine);
        SetModifiedFlag();
}

//返回当前直线的成员函数
CMyLine * CMyMFCSDIDoc::GetLine(int index)
{
        if(index<0||index>m_ObArray.GetUpperBound())
                return 0;
        else return(CMyLine *) m_ObArray.GetAt(index);
}

int CMyMFCSDIDoc::GetTotalLine()
{
        return m_ObArray.GetSize();
}

void CMyLine::Serialize (CArchive & ar){
        if(ar.IsStoring())
                ar<<(WORD)m_StartX<<(WORD)m_StartY
                 <<(WORD)m_EndX<<(WORD)m_EndY
                 <<(WORD)m_pWidth<<(WORD)m_pStyle<<m_pColor;
        else
                ar>>(WORD&)m_StartX>>(WORD&)m_StartY
                 >>(WORD&)m_EndX>>(WORD&)m_EndY
                 >>(WORD&)m_pWidth>>(WORD&)m_pStyle>>m_pColor;
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIDoc 构造函数/析构函数

CMyMFCSDIDoc::CMyMFCSDIDoc()
{
        m_pWidth=1;
        m_pStyle=PEN_STYLE_SOLID;
        m_pColor=RGB(0,0,0);

}

CMyMFCSDIDoc::CMyMFCSDIDoc()
{
```

```
}

BOOL CMyMFCSDIDoc::OnNewDocument()
{
        if (! CDocument::OnNewDocument())
                return FALSE;

        // 在此处添加重新初始化代码

        return TRUE;
}

void CMyMFCSDIDoc::DeleteContents(){
        int index;
        index = m_ObArray.GetSize();
        while(index)
        delete m_ObArray.GetAt( - - index);
        m_ObArray.RemoveAll();
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIDoc 串行化

void CMyMFCSDIDoc::Serialize(CArchive &ar)
{
if(ar.IsStoring()){
        //添加存储代码
        m_ObArray.Serialize(ar);
}
else{
        //添加加载代码
        m_ObArray.Serialize(ar);

}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIDoc 诊断代码

#ifdef _DEBUG
void CMyMFCSDIDoc::AssertValid() const
{
```

```
        CDocument::AssertValid();
}

void CMyMFCSDIDoc::Dump(CDumpContext& dc) const
{
        CDocument::Dump(dc);
}
#endif //_DEBUG

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIDoc 命令

void CMyMFCSDIDoc::OnEditUndo()
{
        //添加命令处理代码
        int index;
        index = m_ObArray.GetUpperBound();
        if(index >= 0){
                delete m_ObArray.GetAt(index);
                m_ObArray.RemoveAt(index);
        }
        UpdateAllViews(NULL);
        SetModifiedFlag();

}

void CMyMFCSDIDoc::OnUpdateEditUndo(CCmdUI * pCCmdUI)
{
        //添加命令更新 UI 的处理代码
        pCCmdUI->Enable(m_ObArray.GetSize());

}

void CMyMFCSDIDoc::OnOptionColor()
{
        // 在此处添加命令处理代码

}

void CMyMFCSDIDoc::OnUpdateOptionColor(CCmdUI * pCmdUI)
{
```

```
        // 在此处添加命令更新 UI 的处理代码

    }

    void CMyMFCSDIDoc::OnOptionPen()
    {
        // 在此处添加命令处理代码

        CPenDlg PenDialog;//声明对话框类对象

        //初始化对话框类对象的数据成员
        PenDialog.m_pWidth = m_pWidth;
        PenDialog.m_pStyle = m_pStyle;

        //显示对话框
        if (PenDialog.DoModal() == IDOK)
        //保存对话框内设置的画笔粗细及风格
            m_pWidth = PenDialog.m_pWidth;
            m_pStyle = PenDialog.m_pStyle;
        }

    }

    void CMyMFCSDIDoc::OnUpdateOptionPen(CCmdUI * pCmdUI)
    {
        // TODO: Add your command update UI handler code here

    }
```

**程序清单 9-5:应用程序视图类定义文件**

```
    // MyMFCSDIView.h : CMyMFCSDIView 类的接口
    //
    /////////////////////////////////////////////////////////////////////////////

    #if ! defined(AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_B31A29E86F7F__
INCLUDED_)
    #define AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_B31A29E86F7F__INCLUD-
ED_

    #if _MSC_VER > 1000
```

```
#pragma once
#endif // _MSC_VER > 1000


class CMyMFCSDIView : public CView
{

protected:                    //新增加保护型数据成员
        int m_Draw;           //表示是否正在拖动鼠标
        HCURSOR m_HCursor; //存放光标句柄
        CPoint m_pOld;        //存放光标的当前位置
        CPoint m_pOrigin; //存放鼠标左键按下时光标所在的位置(画线的起点)

protected:
        CMyMFCSDIView();
        DECLARE_DYNCREATE(CMyMFCSDIView)

// 特性
public:
        CMyMFCSDIDoc* GetDocument();

// 操作
public:

// Overrides
        // ClassWizard 生成的虚拟函数重载
        //{{AFX_VIRTUAL(CMyMFCSDIView)
        public:
        virtual void OnDraw(CDC* pDC); //通过重载来绘制该视图
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        //}}AFX_VIRTUAL

// 实现方式
public:
        virtual CMyMFCSDIView();
#ifdef _DEBUG
        virtual void AssertValid() const;
```

```
        virtual void Dump(CDumpContext& dc) const;
#endif

//生成消息映射函数
protected:
        //{{AFX_MSG(CMyMFCSDIView)
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG // MyMFCSDIView.cpp 的调试代码
inline CMyMFCSDIDoc* CMyMFCSDIView::GetDocument()
{ return (CMyMFCSDIDoc*)m_pDocument; }
#endif

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

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将添加附加的声明语句

#endif // !defined(AFX_MYMFCSDIVIEW_H__4E2E9AC1_6799_11D2_8DE6_B31A29E86F7F__INCLUDED_)
```

**程序清单 9-6:应用程序视图类的实现文件**

```
// MyMFCSDIView.cpp : CMyMFCSDIView 类的实现文件
#include "PenDLG.h"
#include "stdafx.h"
#include "MyMFCSDI.h"
#include "MyMFCSDIDoc.h"
#include "MyMFCSDIView.h"
#include "PenDLG.h"

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

```
////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView

IMPLEMENT_DYNCREATE(CMyMFCSDIView, CView)

BEGIN_MESSAGE_MAP(CMyMFCSDIView, CView)
    //{{AFX_MSG_MAP(CMyMFCSDIView)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    //}}AFX_MSG_MAP
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 构造函数/析构函数

CMyMFCSDIView::CMyMFCSDIView()
{
    //添加构造函数代码
    m_Draw=0;//初始时,未拖动鼠标
    m_HCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);


}

CMyMFCSDIView::~CMyMFCSDIView()
{
}

BOOL CMyMFCSDIView::PreCreateWindow(CREATESTRUCT& cs)
{
    // 修改 Window 类或窗口风格
    cs.lpszClass=AfxRegisterWndClass(
        CS_HREDRAW|CS_VREDRAW,//类风格
        0,//不用光标
        (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
        0);//没有图标
```

```
    return CView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 绘制代码

void CMyMFCSDIView::OnDraw(CDC * pDC)
{
    CMyMFCSDIDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    int index;
    index = pDoc -> GetTotalLine();
    while(index -- ) pDoc -> GetLine(index) -> DrawLine(pDC);

}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 打印代码

BOOL CMyMFCSDIView::OnPreparePrinting(CPrintInfo * pInfo)
{
    return DoPreparePrinting(pInfo);
}

void CMyMFCSDIView::OnBeginPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    //添加打印前的附加初始化代码
}

void CMyMFCSDIView::OnEndPrinting(CDC * /*pDC*/, CPrintInfo * /*pInfo*/)
{
    //添加打印后的清除代码
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 诊断代码

#ifdef _DEBUG
void CMyMFCSDIView::AssertValid() const
```

```
{
    CView::AssertValid();
}

void CMyMFCSDIView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMyMFCSDIDoc * CMyMFCSDIView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyMFCSDIDoc)));
    return (CMyMFCSDIDoc * )m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 消息处理程序

void CMyMFCSDIView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //添加消息处理程序代码

    CView::OnLButtonDown(nFlags, point);

    m_pOld = point;      //存放鼠标光标的当前位置
    m_pOrigin = point;     //存放画线的起始位置
    SetCapture();     //捕获鼠标
    m_Draw = 1;          //表示正在拖动鼠标
    RECT rect;
    GetClientRect(&rect); //获取视图窗口区客户的坐标
    ClientToScreen(&rect); //转换客户区坐标为屏幕坐标
    ClipCursor(&rect); //将光标限定在视图窗口客户区内,接着调用 CWnd 的成员函数
    ClipCursor(&rect); //SetCapture 使随后的鼠标消息都被送往视图窗口

}

void CMyMFCSDIView::OnMouseMove(UINT nFlags, CPoint point)
{
    // 添加消息处理代码
```

```
    CView::OnMouseMove(nFlags, point);
    CClientDC dc(this); //创建设备文本对象
    OnPrepareDC(&dc); //调整视口原点
    dc.DPtoLP(&point);
    SetCursor(m_HCursor);//设置自定义光标
    if(m_Draw){          //正在拖动鼠标
        CMyMFCSDIDoc * pDoc = GetDocument();
        CPen NewPen;
        CPen * pOldPen;

        switch(pDoc->m_pStyle){
        case PEN_STYLE_SOLID:
          NewPen.CreatePen(PS_SOLID,pDoc->m_pWidth,pDoc->m_pColor);
        break;
        case PEN_STYLE_DASH:
          NewPen.CreatePen(PS_DASH,pDoc->m_pWidth,pDoc->m_pColor);
            break;
        case PEN_STYLE_DOT:
          NewPen.CreatePen(PS_DOT,pDoc->m_pWidth,pDoc->m_pColor);
        break;
        }
        pOldPen = dc.SelectObject(&NewPen);
        dc.SetROP2(R2_NOT); //设置绘图模式,以屏幕反转色为绘图颜色
        dc.MoveTo(m_pOrigin);//删除前一 WM_MOUSEMOVE 画的直线
        dc.LineTo(m_pOld);
        dc.MoveTo(m_pOrigin);//从起始位置到当前位置画一条新的直线
        dc.LineTo(point);
        m_pOld = point;//存放鼠标光标的当前位置
        dc.SelectObject(pOldPen);
      }

}

void CMyMFCSDIView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 添加消息处理程序代码
    CView::OnLButtonUp(nFlags, point);

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.DPtoLP(&point);
```

```
        if(m_Draw){
            m_Draw=0;//表示鼠标已释放
            ReleaseCapture();//释放鼠标捕获并恢复正常输入
            ClipCursor(NULL);//使光标可以移到屏幕任意位置
        }

        //释放鼠标左键后画直线结束,应将所画直线添加到 CObarry 对象中
        CMyMFCSDIDoc * pDOC=GetDocument();
        pDOC->AddLine(m_pOrigin.x,m_pOrigin.y, point.x,point.y,pDOC->m_
pWidth,pDOC->m_pStyle,pDOC->m_pColor);

    }
```

**程序清单 9-7:应用程序主窗口类定义文件**

```
    // MyMFCSDIView.cpp : CMyMFCSDIView 类的实现文件
    //
    #include "PenDLG.h"
    #include "stdafx.h"
    #include "MyMFCSDI.h"
    #include "MyMFCSDIDoc.h"
    #include "MyMFCSDIView.h"
    #include "PenDLG.h"

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

    /////////////////////////////////////////////////////////////////////////////
    // CMyMFCSDIView

    IMPLEMENT_DYNCREATE(CMyMFCSDIView, CView)

    BEGIN_MESSAGE_MAP(CMyMFCSDIView, CView)
        //{{AFX_MSG_MAP(CMyMFCSDIView)
        ON_WM_LBUTTONDOWN()
        ON_WM_MOUSEMOVE()
        ON_WM_LBUTTONUP()
        //}}AFX_MSG_MAP
```

```
    // 标准打印命令
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 构造函数/析构函数

CMyMFCSDIView::CMyMFCSDIView()
{
    // TODO: add construction code here
    m_Draw=0;//初始时,未拖动鼠标
    m_HCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);


}

CMyMFCSDIView::CMyMFCSDIView()
{
}

BOOL CMyMFCSDIView::PreCreateWindow(CREATESTRUCT& cs)
{
    // 修改 Window 类或窗口风格
    cs.lpszClass=AfxRegisterWndClass(
        CS_HREDRAW|CS_VREDRAW,//类风格
        0,//不用光标
        (HBRUSH)::GetStockObject(WHITE_BRUSH),//设置白色背景
        0);//没有图标

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 绘制代码

void CMyMFCSDIView::OnDraw(CDC* pDC)
{
    CMyMFCSDIDoc* pDoc=GetDocument();
    ASSERT_VALID(pDoc);
```

```
    int index;
    index = pDoc->GetTotalLine();
    while(index--) pDoc->GetLine(index)->DrawLine(pDC);

}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 打印代码

BOOL CMyMFCSDIView::OnPreparePrinting(CPrintInfo* pInfo)
{
    //缺省打印准备
    return DoPreparePrinting(pInfo);
}

void CMyMFCSDIView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // 添加打印前附加的初始化代码
}

void CMyMFCSDIView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    //添加打印后的清除代码
}

////////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 诊断代码

#ifdef _DEBUG
void CMyMFCSDIView::AssertValid() const
{
    CView::AssertValid();
}

void CMyMFCSDIView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMyMFCSDIDoc* CMyMFCSDIView::GetDocument()
```

```
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyMFCSDIDoc)));
    return (CMyMFCSDIDoc*)m_pDocument;
}
#endif //_DEBUG

//////////////////////////////////////////////////////////////////////////
// CMyMFCSDIView 消息处理程序

void CMyMFCSDIView::OnLButtonDown(UINT nFlags, CPoint point)
{
    //添加消息处理程序代码

    CView::OnLButtonDown(nFlags, point);

    m_pOld=point;      //存放鼠标光标的当前位置
    m_pOrigin=point;      //存放画线的起始位置
    SetCapture();      //捕获鼠标
    m_Draw=1;          //表示正在拖动鼠标
    RECT rect;
    GetClientRect(&rect); //获取视图窗口区客户的坐标
    ClientToScreen(&rect); //转换客户区坐标为屏幕坐标
    ClipCursor(&rect); //将光标限定在视图窗口客户区内,然后调用 CWnd 的成员函数
    ClipCursor(&rect); //SetCapture 使随后的鼠标消息都被送往视图窗口

}

void CMyMFCSDIView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CView::OnMouseMove(nFlags, point);
    CClientDC dc(this); //创建设备文本对象
    OnPrepareDC(&dc); //调整视口原点
    dc.DPtoLP(&point);
    SetCursor(m_HCursor);//设置自定义光标
    if(m_Draw){          //正在拖动鼠标
        CMyMFCSDIDoc * pDoc=GetDocument();
        CPen NewPen;
        CPen * pOldPen;
```

```
            switch(pDoc->m_pStyle){
            case PEN_STYLE_SOLID:
                NewPen.CreatePen(PS_SOLID,pDoc->m_pWidth,pDoc->m_pColor);
                break;
            case PEN_STYLE_DASH:
                NewPen.CreatePen(PS_DASH,pDoc->m_pWidth,pDoc->m_pColor);
                break;
            case PEN_STYLE_DOT:
                NewPen.CreatePen(PS_DOT,pDoc->m_pWidth,pDoc->m_pColor);
                break;
            }
            pOldPen=dc.SelectObject(&NewPen);
            dc.SetROP2(R2_NOT); //设置绘图模式,以屏幕反转色为绘图颜色
            dc.MoveTo(m_pOrigin);//删除前一 WM_MOUSEMOVE 画的直线
            dc.LineTo(m_pOld);
            dc.MoveTo(m_pOrigin);//从起始位置到当前位置画一条新的直线
            dc.LineTo(point);
            m_pOld=point;//存放鼠标光标的当前位置
            dc.SelectObject(pOldPen);
        }

    }

    void CMyMFCSDIView::OnLButtonUp(UINT nFlags, CPoint point)
    {
        // TODO: Add your message handler code here and/or call default
        CView::OnLButtonUp(nFlags, point);

        CClientDC dc(this);
        OnPrepareDC(&dc);
        dc.DPtoLP(&point);
        if(m_Draw){
            m_Draw=0;//表示鼠标已释放
            ReleaseCapture();//释放鼠标并恢复正常输入
            ClipCursor(NULL);//使光标可以移到屏幕上的任意位置
        }

        //释放鼠标左键后,画直线结束,应将所画直线添加到 CObarry 对象中
        CMyMFCSDIDoc * pDOC=GetDocument();
        pDOC->AddLine(m_pOrigin.x,m_pOrigin.y, point.x,point.y,pDOC->m_
pWidth,pDOC->m_pStyle,pDOC->m_pColor);
```

```
}
```

**程序清单 9-8:应用程序主窗口的实现文件**

```
// MainFrm.cpp : CMainFrame 类的实现文件
//

#include "stdafx.h"
#include "MyMFCSDI.h"

#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
                // ClassWizard 将在此处添加或删除映射宏
        ON_WM_CREATE()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
        ID_SEPARATOR,
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame 构造函数/析构函数
```

```
CMainFrame::CMainFrame()
{
        // 添加成员初始化代码

}

CMainFrame::CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;

        if (! m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD|WS_VISIBLE
|CBRS_TOP|CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY|CBRS_SIZE_DYNAMIC)||
                ! m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
        {
                TRACE0("Failed to create toolbar\n");
                return -1;      // fail to create
        }

        if (! m_wndStatusBar.Create(this) ||
                ! m_wndStatusBar.SetIndicators(indicators,
                 sizeof(indicators)/sizeof(UINT)))
        {
                TRACE0("Failed to create status bar\n");
                return -1;      // fail to create
        }

        // 船坞化工具栏
        m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
        EnableDocking(CBRS_ALIGN_ANY);
        DockControlBar(&m_wndToolBar);

        return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
```

```
        if( ! CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        // 修改 Window 类或窗口风格

        return TRUE;
}

////////////////////////////////////////////////////////////////////////////
// CMainFrame 诊断代码

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
        CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
        CFrameWnd::Dump(dc);
}

#endif //_DEBUG

////////////////////////////////////////////////////////////////////////////
// CMainFrame 消息处理程序
```

**程序清单 9-9:对话框类定义文件**

```
// PenDLG.h : header file
//

enum {PEN_STYLE_SOLID,PEN_STYLE_DASH,PEN_STYLE_DOT};
////////////////////////////////////////////////////////////////////////////
// CPenDlg 对话框

class CPenDlg:public CDialog
{
// 构造函数
public:
        CPenDlg(CWnd* pParent = NULL); // standard constructor
```

```
// 对话框数据
        //{{AFX_DATA(CPenDlg)
        enum { IDD = IDD_DIALOG1 };
        CEdit m_pWidthEdit;
        int m_pWidth;
        int m_pStyle;
        //}}AFX_DATA


// 重载
        // ClassWizard 生成虚拟函数重载
        //{{AFX_VIRTUAL(CPenDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
        //}}AFX_VIRTUAL

// 实现方式
protected:

        // Generated message map functions
        //{{AFX_MSG(CPenDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnPaint();
        afx_msg void OnDash();
        afx_msg void OnDot();
        afx_msg void OnSolid();
        afx_msg void OnChangeLinewidthInput();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
protected:
        RECT m_pRectSample;
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将添加附加的声明语句
```

**程序清单 9-10:对话框类实现文件**

```
// PenDLG.cpp : 实现文件
//
```

```
#include "stdafx.h"
#include "MyMFCSDI.h"
#include "PenDLG.h"

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

//////////////////////////////////////////////////////////////////////////////
// CPenDlg 对话框

CPenDlg::CPenDlg(CWnd* pParent /* = NULL */)
        : CDialog(CPenDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CPenDlg)
        m_pWidth = 0;
        m_pStyle = -1;
        //}}AFX_DATA_INIT
}


void CPenDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CPenDlg)
        DDX_Control(pDX, IDC_LINEWIDTH_INPUT, m_pWidthEdit);
        DDX_Text(pDX, IDC_LINEWIDTH_INPUT, m_pWidth);
        DDV_MinMaxInt(pDX, m_pWidth, 1, 6);
        DDX_Radio(pDX, IDC_SOLID, m_pStyle);
        //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CPenDlg, CDialog)
        //{{AFX_MSG_MAP(CPenDlg)
        ON_WM_PAINT()
        ON_BN_CLICKED(IDC_DASH, OnDash)
        ON_BN_CLICKED(IDC_DOT, OnDot)
        ON_BN_CLICKED(IDC_SOLID, OnSolid)
```

```
        ON_EN_CHANGE(IDC_LINEWIDTH_INPUT, OnChangeLinewidthInput)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CPenDlg 消息处理程序

BOOL CPenDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // 添加附加的初始化代码

        GetDlgItem(IDC_LINEWIDTH_INPUT)->GetWindowRect(&m_pRectSample);
        ScreenToClient(&m_pRectSample);
        m_pWidthEdit.LimitText(1) ;
        return TRUE; //若不设置控件的焦点则返回 TRUE
                     //OCX Property Pages 异常时返回 FALSE
}

void CPenDlg::OnPaint()
{
        CPaintDC dc(this);

        //添加消息处理程序代码

        CPen NewPen; //声明新画笔对象
        CPen * pOldPen;//保存原画笔的指针
        //初始化画笔对象
        switch(m_pStyle){
        case PEN_STYLE_SOLID:
                NewPen.CreatePen(PS_SOLID,m_pWidth,RGB(0,0,0));
                break;
        case PEN_STYLE_DASH:
                NewPen.CreatePen(PS_DASH,m_pWidth,RGB(0,0,0));
                break;
        case PEN_STYLE_DOT:
                NewPen.CreatePen(PS_DOT,m_pWidth,RGB(0,0,0));
                break;
        default:
                NewPen.CreatePen(PS_SOLID,m_pWidth,RGB(255,0,0));
```

```
            break;
        }
        pOldPen = dc.SelectObject(&NewPen);
        //使用画笔在成组框中画线
        dc.MoveTo(m_pRectSample.left + 10,m_pRectSample.top + 10);
        dc.LineTo(m_pRectSample.right - 10,m_pRectSample.bottom - 10);
        //恢复原画笔对象
        dc.SelectObject(pOldPen);

}

void CPenDlg::OnDash()
{
        // 添加控件通知处理程序代码
        if (IsDlgButtonChecked(IDC_DASH)){
            m_pStyle = PEN_STYLE_DASH;
            InvalidateRect(&m_pRectSample);
            UpdateWindow();
        }

}

void CPenDlg::OnDot()
{
        //添加控件通知处理程序代码
        if (IsDlgButtonChecked(IDC_DOT)){
            m_pStyle = PEN_STYLE_DOT;
            InvalidateRect(&m_pRectSample);
            UpdateWindow();
        }

}

void CPenDlg::OnSolid()
{
        // 添加控件通知处理程序代码
        if (IsDlgButtonChecked(IDC_SOLID)){
            m_pStyle = PEN_STYLE_SOLID;
            InvalidateRect(&m_pRectSample);
            UpdateWindow();
        }
```

```
}

void CPenDlg::OnChangeLinewidthInput()
{
        //添加控件通知处理程序代码
        int tmp;
        tmp=(int)GetDlgItemInt(IDC_LINEWIDTH_INPUT);
        if ((tmp>0) && (tmp<7))
                m_pWidth=tmp;
                InvalidateRect(&m_pRectSample);
                UpdateWindow();
        }
}
```

# 第 10 章

# 动态链接库

在多任务环境中，为了提高系统资源的利用率和系统的整体性能，就应该设法使每一个应用程序尽量少地占用资源，对于系统内存更是如此。由于同时运行的多个应用程序中，往往有较多的程序段具有相同的代码，如 Windows 环境下的窗口维护和标准的数学函数等。这些操作代码相同，只是操作的数据不同，可以设想，让所有使用相同函数而又同时运行的应用程序都可使用这些共同函数的一个备份，则会大大缩短应用程序的长度，从而减少内存需求。动态链接库便是这一设想的具体实现。

## 10.1 什么是动态链接库

动态链接库（Dynamics Link Library，简称 DLL）是一个包含了若干函数的可执行模块，Windows 应用程序可以调用这些函数来完成实际的任务。DLL 在 Windows 环境中起着重要的作用——Windows 利用 DLL 来建立 Windows 应用程序可以利用的 Windows 函数和资源。

DLL 与运行函数库类似，如 C 语言的运行函数库。二者主要区别在于库代码的链接时机：DLL 是在运行期间被链接进来，此操作由 Windows 操作系统自身来完成，而运行函数库则是在程序链接期间由应用程序的链接器如 LINK.EXE 来完成。前者被称为“动态链接”，后者被称为“静态链接”。静态链接把函数代码链接到应用程序中，它增加了应用程序的最终可执行程序的长度。

使用静态链接库的主要优点是建立了一个应用程序可以利用的标准函数集，这样在开发应用程序时就无需包含函数库中所包含的函数的源代码。

静态链接库在多任务环境中运行时效率可能很低。一方面，如果两个应用程序同时运行，且它们使用了同一静态库中的函数，那么就要求系统中出现该函数的两个副本，这显然降低了内存的使用效率，由此可见两个应用程序若能共享该函数的单个副本就可以提高内存的使用效率，然而对于这一点，静态链接库无能为力。另一方面，对于比较复杂的问题，如果应用静态链接库将导致最终的应用程序长度超过系统内存而无法运行，事实上在没有引入动态链接库的 DOS 环境下，较大的应用程序是通过覆盖来缩短应用程序的长度的。只不过此时的覆盖绝大数针对具体的应用程序，而不供其他应用程序使用。

与静态链接库不同，动态链接库（DLL）则允许若干个应用程序共享某个函数的单个副本。事实上，每个 Windows 函数，如 GetMessage、CreateWindow 和 TextOut 均驻留于动态链接库——KERNEL.EXE、USE.EXE 和 GDI.EXE 的某一个之中。如果两个应用程序同时运行，且都使用了某个特定的 Windows 函数，那么它们将共享该函数代码的单个副本。

DLL 除了实现代码的共享外，还可以实现其他资源的共享，如数据和硬件资源的共享。例如，Windows 的字体实际上就是应用程序中以共享的文本输出数据，同样地，Windows 的设备驱动程序实际上也是允许应用程序共享硬件资源的动态链接库。

所有的 Windows 库均是动态链接库，虽然它们的文件名后缀可能是 DLL，也可能是 EXE。例如，KERNEL.EXE、USE.EXE 和 GDI.EXE 都是动态链接库。用户也可以开发自己的动态链接库，用以实现代码、数据或硬件资源的共享。

## 10.2　利用向导开发动态链接库

用户要开发自己的动态链接库，必须遵循动态链接库的应用规则，选择适当的开发软件。运行在 Windows 环境下的各种应用程序开发软件，都提供了动态链接库的开发工具，利用它们开发 DLL 各有千秋。下面介绍如何利用 Visual C++ 6.0 提供的基于 MFC 的 DLL 开发向导来开发用户自己的 DLL。

当你利用 MFC APPWizard(dll)来创建动态链接库时，能够获得实现动态链接库基本功能的初始应用代码。MFC 初始程序包括 C++ 源程序、资源文件和工程文件。所有这些文件都是基于 MFC 的。

创建动态链接库的过程有三步：

第一步：根据 MFC APPWizard(dll)提供的选项一步一步地定制应用程序。

第二步：编译并链接这些初始代码。

第三步：在向导所创建的初始代码中添加自己的代码，以实现具体功能。

下面是利用向导创建 MFC DLL 应用程序的步骤：

(1) 在 File 菜单中选择 New，在打开的对话框中选取 Projects 选项卡，如图 10-1 所示。

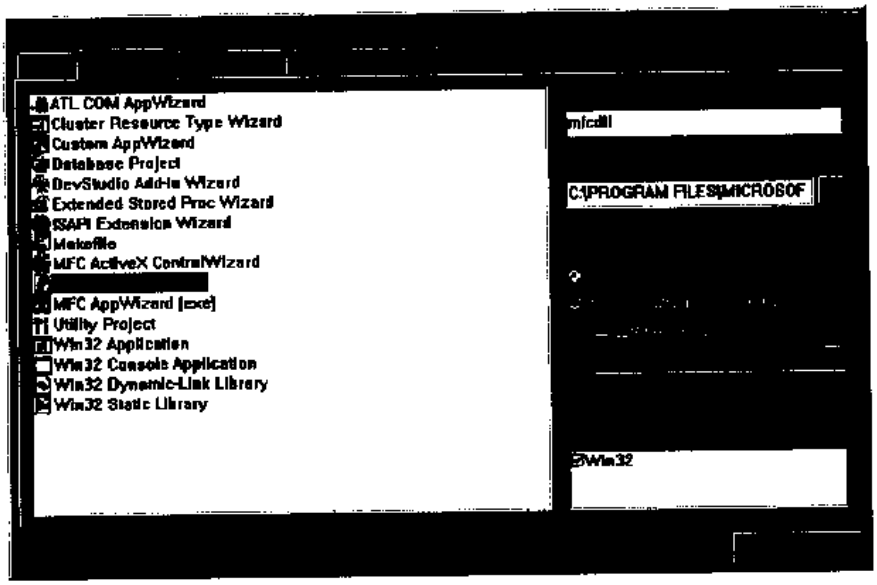



图 10-1　Projects 选项卡

(2) 输入项目名称(本例中为 mfcdll)、位置、工作区、依赖和硬件平台(这几项可采用系统默认设置)等选项，最后双击 MFC AppWizard(dll)图标。

(3) 在 MFC AppWizard (dll)生成选项的对话框(如图 10-2 所示)中选择要创建的 DLL 类型，以及是否有 Automation 和 Windows Sockets 特性支持。

下面简单介绍一下各选项的意义：

• Regular DLL with MFC statically linked：此选项在建立工程时把 MFC 库静态地链接到用户开发的动态链接库中。Win32 和 MFC 程序都可以调用动态链接库中的函数。

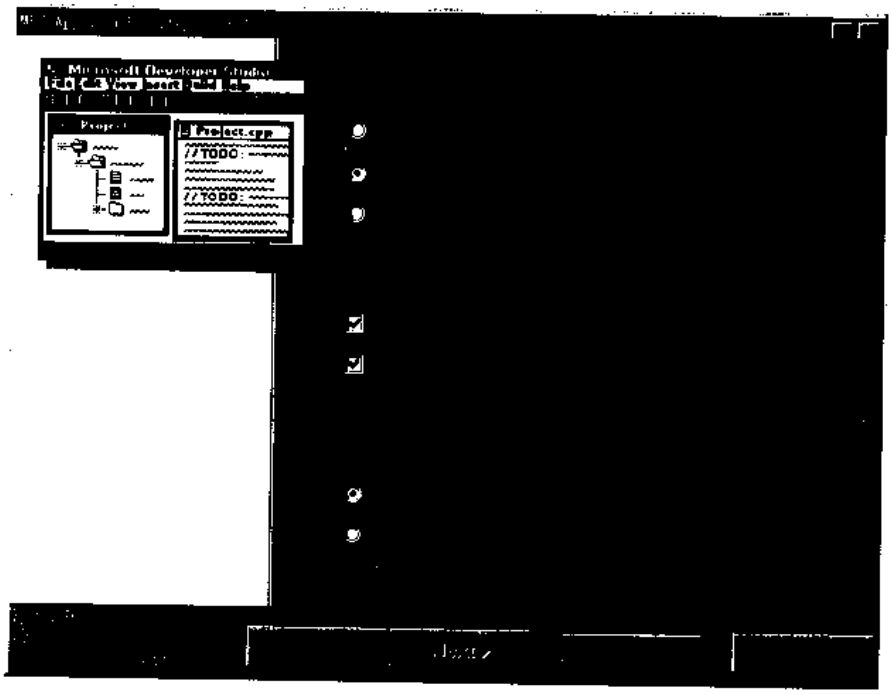


图 10-2 选择要创建的 DLL 类型

• Regular DLL using shared MFC DLL:此选项把 MFC 库作为共享的 DLL 库链接到用户开发的 DLL 中,用户程序在运行时调用这些 MFC 库函数。这样如果应用程序是由多个调用 MFC 库的可执行程序组成,就可以缩短内存和硬盘空间需求,同样地,Win32 和 MFC 程序都可以调用用户动态链接库中的函数。

• MFC Extension DLL (useing shared MFC DLL):此选项也是在运行时调用 MFC 库函数,同样地,如果用户的应用程序是由多个调用 MFC 库的可执行程序组成,那么也可以减少内存和硬盘空间需求。只有 MFC 程序可以调用动态链接库中的函数。

• Automation:利用该选项可使程序维护嵌入到其他程序中的对象,即把你的程序作为服务器供其他客户程序调用。

• Windows Sockets:启用该选项允许动态链接库使用 Windows Sockets 中的程序模块。Windows Sockets 为 Windows 操作系统的网络功能提供了套口程序。

• Would you like to generate source file comments:是否要求向导在程序代码中产生注释。

4. 做好以上选择后,单击 Finish 按钮,查看完简单描述后,再单击 OK 按钮即生成了初始代码。生成的初始代码有两个实现文件(mfcdll.cpp 和 StdAfx.cpp)、三个头文件(mfcdll.h、Resource.h 和 StdAfx.h)、一个模块定义文件(mfcdll.def)、一个资源文件(mfcdll.rc)、一个 Visual C++ 不能直接编辑的资源文件(mfcdll.rc2)和一个描述本项目组成的文件(ReadMe.txt)。在这里资源文件只包含一个版本资源,因此现在不必考虑此文件和后两个文件。

## 10.3 DLL 应用示例(DLL 部分)

经过前面所叙述的各步之后,我们已经得到了初始文件,现在对这些初始文件进一步适

当地添加代码就可以得到一个 DLL 应用程序的完整例子。下面依次介绍每个文件的作用及如何进行扩展,以使其实用化。

1. 头文件 mfcdll.h

头文件 mfcdll.h 用于描述要引用的任何函数(即供其他应用程序调用的函数)原型。在此例子中我们只在向导产生的文件基础上增加了一个函数 drawsin()。此函数用于绘制正弦波。由于 CPaintDC 类没有拷贝构造,所以无法将 CPaintDC 类对象以传值的形式传递给被调函数 drawsin,只有将其声明为指针变量。下面列出了完整的程序清单。

**程序清单 10-1:创建 DLL 时要引用的函数的原型声明文件**

```
// mfcdll.h : MFCDLL.DLL 的主头文件
//
#if ! defined(AFX_MFCDLL_H__6F58CF46_773C_11D2_8DE6_C92ACE0DD61E__INCLUDED_)
#define AFX_MFCDLL_H__6F58CF46_773C_11D2_8DE6_C92ACE0DD61E__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
#endif

#include "resource.h"          // 主资源符号
//下面的一条语句新增加的,此语句声明了由其他应用程序调用来绘制正弦波的函数 drawsin
_declspec (dllexport) int WINAPI drawsin(CPaintDC * dc int x,int y);


/////////////////////////////////////////////////////////////////////////////
// CMfcdllApp 类声明部分,其实现部分在文件 Mfcdll.cpp 中
//

class CMfcdllApp : public CWinApp
{
public:
        CMfcdllApp();//缺省的构造函数

// Overrides
        // ClassWizard 产生的虚函数重载
        //{{AFX_VIRTUAL(CMfcdllApp)
```

```
        //}}AFX_VIRTUAL

        //{{AFX_MSG(CMfcdllApp)
            // ClassWizard将增加或删除这里的成员函数
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将在语句之前插入其他声明语句
#endif // !defined(AFX_MFCDLL_H__6F58CF46_773C_11D2_8DE6_C92ACE0DD61E__
INCLUDED_)
```

2. 源代码文件 mfcdll.cpp

下面列出 mfcdll.cpp 的完整程序清单，在此清单中，我们修改了向导所产生的注释，增加我们所要引用的函数的定义。

**程序清单 10-2：CMfcdllApp 类的实现文件**

```
// mfcdll.cpp : 该文件定义了 DLL 的初始化例程
//

#include "stdafx.h"
#include "mfcdll.h"

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

//
// 注意!
//
//如果此 DLL 是与 MFC DLL 动态链接的，则其输出的任何调用 MFC
//的函数都必须在其前面加上 AFX_MANGR_STATE 宏
//这个宏必须出现在任何 MFC 调用之前。这意味着
//它必须是函数内部的第一条语句，甚至在任何对象变量声明之前，因为它们
//的构造函数有可能会调用 MFC DLL
```

```
//////////////////////////////////////////////////////////////////////////////
// CMfcdllApp

BEGIN_MESSAGE_MAP(CMfcdllApp, CWinApp)
//{{AFX_MSG_MAP(CMfcdllApp)
//ClassWizard 将在这里添加或删除映射宏
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////////////
// CMfcdllApp 类的构造函数

CMfcdllApp::CMfcdllApp()
{
        // 在此添加构造函数代码
        //将所有重要的初始化代码放入 InitInstance
}

//////////////////////////////////////////////////////////////////////////////
// 有且只有一个 CMfcdllApp 对象
CMfcdllApp theApp;
_declspec (dllexport) int WINAPI drawsin(CPaintDC * dc int x,int y){

        AFX_MANAGE_STATE(AfxGetStaticModuleState());

        CPaintDC dc(this);
        int i,j;
        double y;
        dc->MoveTo(x+0,y+150);
        dc->LineTO(x+0,y-150);
        dc->MoveTO(x,y);
        dc->LineTO(x+420,y);
        dc->MoveTO(x,y);
        dc->LineTO(x+0,y-150);
        for(i=1;i<400;i++){
                y=180.0*(exp(-i*0.01))*sin(3.1415926535*i*1440.0/400.0)/180.
0);
                dc->LineTO(x+i,y+(int)y);
        }
        return 1;
}
```

## 10.4 DLL 应用示例(调用部分)

在前面我们设计了一个动态链接库,此库中引用了一个函数 drawsin 供其他的应用程序调用。下面将再开发一个简单的应用程序,该程序将调用我们前面设计好的 DLL。

首先,生成一个 SDI 应用程序,详细步骤见第 6 章内容。在这个生成过程中所有各步都采用默认设置,项目名称为 SDIDLL。

接着,对生成的初始文件添加代码,以使其可以调用上一节生成的 DLL。

在 SDIDLLView.h 中类 SDIDLLView 的定义之前,添加如下代码:

```
extern int WINAPI drawsin(CPaint * dc int x,int y);
```

这里的关键字 extern 告诉编译器此函数是在当前程序的外部定义的。在创建过程中,链接器将搜索此函数,如果不能在合适的库中搜索到 drawsin()函数,将得到一条错误信息。

为了处理 WM _ PAINT 消息,增加 OnPaint()成员函数,并添加相应的消息处理函数,即在 OnPaint()函数增加相应的变量声明,并调用 DLL 中的函数 drawsin()的语句:

```
CPaintDC dc(this);
drawsin(&dc,100,100);//应将 CPaintDC 类的对象 dc 的地址传递给 DLL 中的函数 drawsin
```

做好以上修改后,创建应用程序,然后运行该应用程序,运行结果如图 10-3 所示。

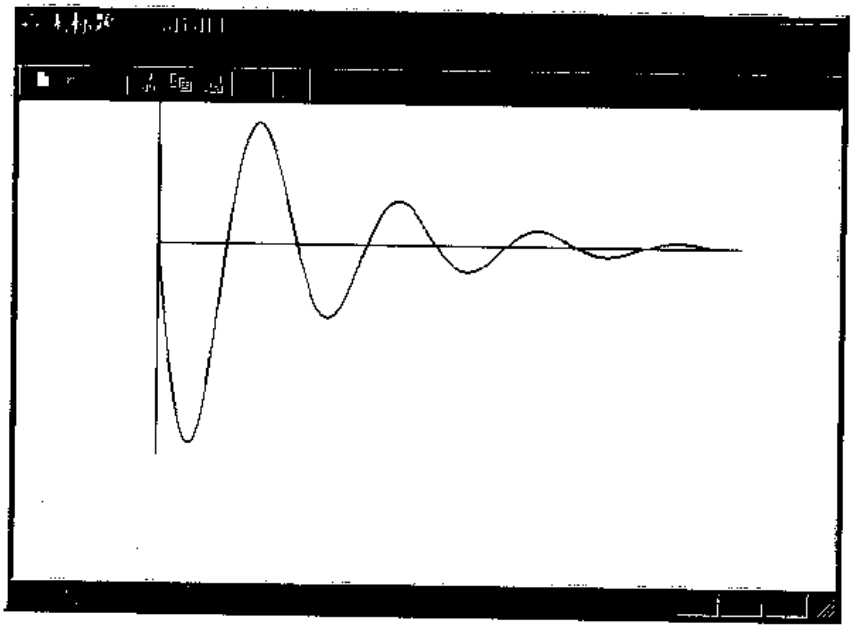


图 10-3 SDI 应用程序结果

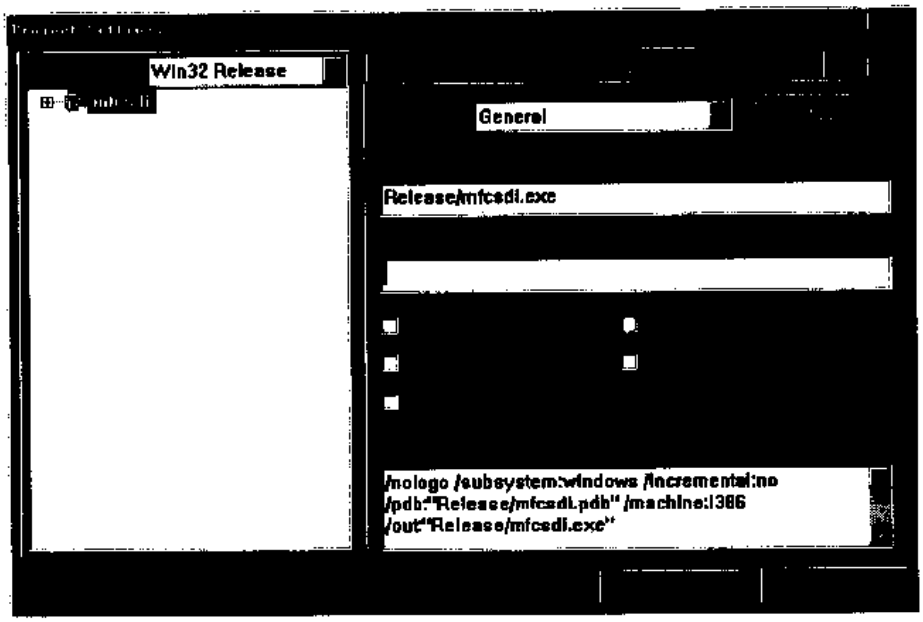



图 10-4　通过项目设置的 Link 选项卡指定 DLL 模块

## 10.5　动态链接库与静态链接库应用的进一步比较

按照前面的叙述，如果别人为我们开发好动态链接库，那么调用位于动态链接库中的函数与调用一般的静态链接库中的函数在应用程序开发时似乎没有什么不同之处，但是在本章的开始，我们已经指出：调用静态链接库中的函数由编译器查找函数所在模块（实际上就是被调用函数所在的文件），并将函数代码链接到应用程序中。而调用动态链接库中的函数在编译时暂不定位被调用的函数所在的模块，链接时也不将函数代码链接到程序中，只是到执行时直接执行函数代码。

我们的应用程序怎么知道这些函数所在模块呢？对于静态链接库，选择 Tools 菜单的 Options 命令，在打开的 Directs 选项卡中设置静态链接库（Library Files）所在目录，链接器就在这里所列出的目录中查找静态链接库。对于动态链接库，虽然被调用函数的代码不链接到应用程序中，但是应用程序也必须知道被调用函数所在模块，以便链接器在应用程序中添加必要的信息（这些必要的信息实际上就是所谓的引入库）使应用程序能够找到被调用函数所在的动态链接库及库文件位置。

通过上面的讨论可以看出，在应用程序中调用动态链接库中的函数必须进行适当的设置。具体设置步骤如下：选择 Project 菜单的 Settings 菜单项，打开项目设置对话框，如图 10-4 所示。

在此对话框中，Object/library moudles：确定 DLL 模块的目标位置。当设置好此项后应保证 10.3 节中开发的动态链接库所生成的文件 mfcdll.dll 位于 Windows 的系统目录，一般情况下是 C：\ Windows \ System，而 mfcdll.lib 与当前项目的目标模块（即 .Obj 文件）在同一目录中，此例是 C：\ Program Files \ Micorsoft Visual Studio \ MyProjects \ Sdidll \ debug 目录。

# 第 11 章

# ActiveX 控件及应用

Windows 提供了种类繁多的大量控件，这些控件被称为标准控件，如单选按钮、复选框、列表框和组合框等。许多开发者也设计了他们自己的控件，这此控件被为 ActiveX 控件或自定义控件。自定义控件最早起源于 Visual Basic，利用 Visual Basic 可以很容易地实现不属于标准控件的自定义控件，这些自定义控件由于具有.VBX 后缀的文件名而被称为 VBX。这些文件实际上就是上一章介绍的动态链接库(DLL)，只不过它们不用 DLL 作为扩展名而已。

许多 Visual C++程序员利用 Visual Basic 开发自己的自定义控件，然后将它们集成到 Visual C++应用程序中。Microsoft 公司在开发 Visual C++环境下的自定义控件时，Windows 系统也开始从 16 位的 Windows 3.1 向 32 位的 Windows 95 或 Windows NT 转移，这使得原来的 VBX 控件在新的系统下并不能按照程序员的意图工作，于是 Microsoft 公司决定与其扩展原来的 VBX 控件，倒不如重新设计自定义控件，使它能很好地支持 32 位平台，这样在 32 位的 Windows 95 和 Windows NT 环境下，自定义控件的名称由原来的 VBX 控件变成 ActiveX 控件。

为了支持用户更好地开发 ActiveX 控件，提高编程效率，在 Visual C++中提供了开发 ActiveX 控件的向导，利用该向导可以方便、快速地开发 ActiveX 控件。此向导能够帮助用户生成面向对象的 C++代码，它也充分地利用了 MFC 的全部优点。此外，与 Visual C++一起提供给用户的还有一个 ActiveX 控件测试容器工具，利用它可以测试用户开发的控件。

## 11.1 ActiveX 控件应用程序的开发

开发 ActiveX 控件应用程序必须解决两个方面的问题：一方面需要设计并完整地实现控件及其属性，这主要包括创建、编写并编译用于绘制控件和实现控件的各种方法的代码，所有的这些代码最终成为一个具有 OCX 后缀的小型动态链接库。另一方面使用此 ActiveX 控件的应用程序必须与控件的方法、数据等交互作用，因此必须仔细地设计接口。好的 ActiveX 控件必须与使用它的应用程序完全无关，也就是说它必须是完全可重用的，也正是基于这一点，才要求 ActiveX 控件必须是动态链接库。

### 11.1.1 开发 ActiveX 控件

Active 控件的开发无疑是一个复杂的课题，好在 MFC 类库和 Visual C++的向导工具为用户做了许多后台工作，用户只需开发 ActiveX 控件要做的实际工作，即要完成的任务。开发 ActiveX 控件一般需要三步来进行：

第一步，利用 MFC ActiveX ControlWizard 逐步选择有关创建 ActiveX 控件的选项，先生成初始程序代码。

第二步，在初始程序代码中添加代码，以增强其功能。

第三步，编译并链接前两步所产生的程序代码。

下面我们详细地讨论创建 ActiveX 控件的步骤：

(1) 从 File 菜单中选择 New 菜单项，在打开的 New 对话框中选中 Projects 选项卡。给出项目名、位置、工作区和基础平台等选项，如图 11-1 所示。

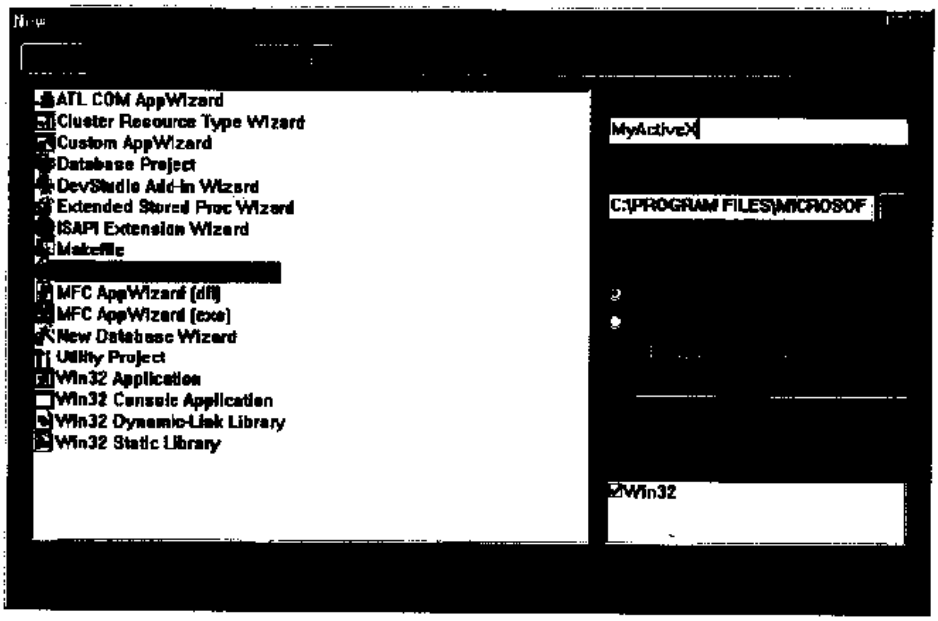



图 11-1　开发 ActiveX 控件的项目设置

(2) 双击 MFC ActiveX ControlWizard 图标或选中 MFC ActiveX ControlWizard 图标后，再单击 OK 按钮，在打开的图 11-2 所示的对话框中设置控件库特性。这时打开图 11-3 所示的对话框，在这里可以设置：库中包含的控件数目(每一个控件有一控件类和属性页类。假定设为 1)；是否想让项目文件中的控件有运行时许可；是否产生上下文敏感的帮助文件。

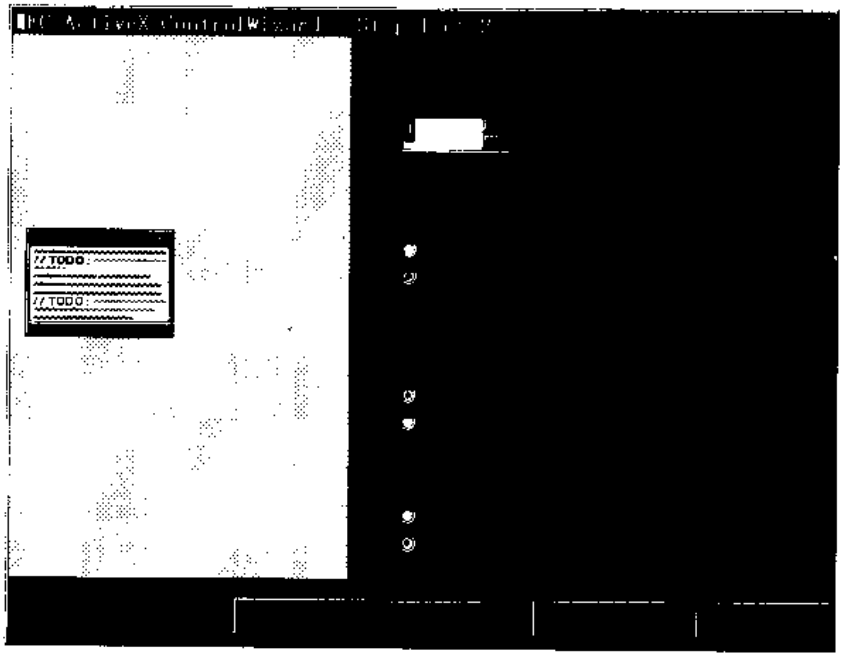



图 11-2　设置控件库特性

(3) 设置完毕后，单击 Next 按钮，打开图 11-3 所示的对话框。在此对话框中设置控件的浏览和编辑特性，以及此控件继承的窗口类。

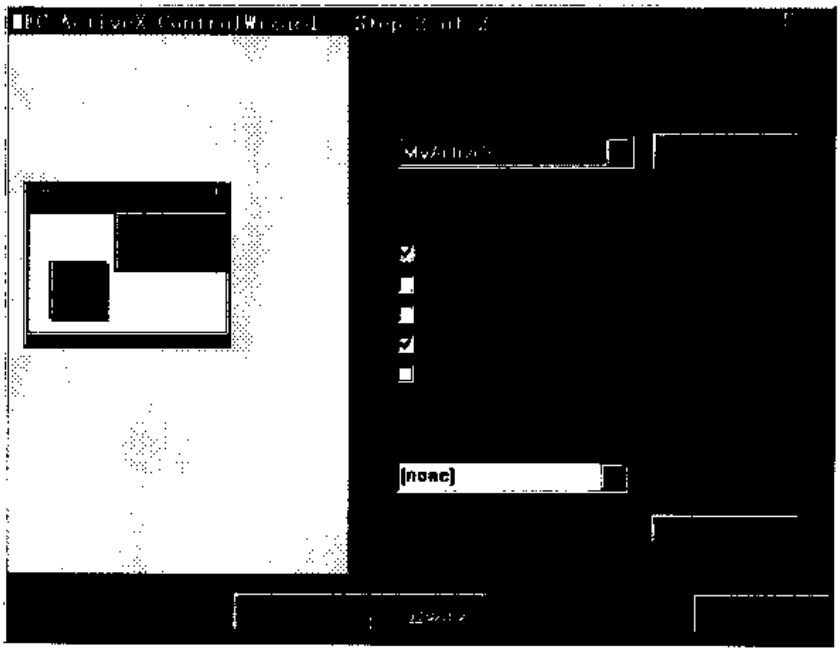


图 11-3　控件特性设置对话框

(4) 设置正确后,单击 Finish 按钮,在显示该项目的简单描述的对话框中再单击 OK 按钮。执行完以上各步,已经生成了一系列文件,这些文件包括源代码(.CPP)和相应的头文件(.H)、资源文件(.RC)、模块定义文件(.DEF)、对象描述语言文件(.ODL)以及把前面文件组织起来的工程文件(.DSP),这些文件与 ClassWizard 兼容,因此可以利用 ClassWizard 来定义控件的事件、属性和方法,其中有不少已经由 MFC 实现。

### 11.1.2　事件、方法和属性

1. 事件

事件是指一个控件对外部动作产生的响应或动作,例如,按键或单击鼠标。KeyUp 和 KeyDown 是固定事件的例子。

由于控件类继承自 MFC 类库中的 COleControl 类,可以使用一个新的消息映射。这些消息或事件被发送到使用控件的应用程序,此应用程序即为控件容器。当控件发生了一些事件时,应用程序将接收到有关这个事件的一些信息,而附加的信息通过事件参数来提供。对于所发生事件的具体处理由控件类的消息处理成员函数来完成。控件的消息映射除了继承自 COleControl 类的缺省的消息映射外,用户也可能添加新的消息映射,或者重载已有的消息映射。COleControl 类的缺省的消息映射代码见程序清单 11-1。

**程序清单 11-1:COleControl 类的缺省的消息映射代码**

```
protected:
        //{{AFX_MSG(COleControl)
        afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
```

```
        afx_msg void OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
        afx_msg void OnMButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnMButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnMButtonDblClk(UINT nFlags, CPoint point);
        afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnRButtonDblClk(UINT nFlags, CPoint point);
        afx_msg void OnInitMenuPopup(CMenu * , UINT, BOOL);
        afx_msg void OnMenuSelect(UINT nItemID, UINT nFlags, HMENU hSysMenu);
        afx_msg LRESULT OnSetMessageString(WPARAM wParam, LPARAM lParam);
        afx_msg void OnEnterIdle(UINT nWhy, CWnd * pWho);
        afx_msg void OnCancelMode();
        afx_msg void OnPaint(CDC * pDC);
        afx_msg BOOL OnEraseBkgnd(CDC * pDC);
        afx_msg void OnSysKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg void OnSysKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg int OnMouseActivate(CWnd * pDesktopWnd, UINT nHitTest, UINT
message);
        afx_msg LRESULT OnSetText(WPARAM wParam, LPARAM lParam);
        afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnKillFocus(CWnd * pNewWnd);
        afx_msg void OnSetFocus(CWnd * pOldWnd);
        afx_msg void OnNcPaint();
        afx_msg void OnNcCalcSize(BOOL bCalcValidRects, NCCALCSIZE_PARAMS *
lpncsp);
        afx_msg UINT OnNcHitTest(CPoint point);
        afx_msg void OnNcLButtonDown(UINT nHitTest, CPoint point);
        afx_msg BOOL OnSetCursor(CWnd * pWnd, UINT nHitTest, UINT message);
        afx_msg UINT OnGetDlgCode();
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnMove(int x, int y);
        afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
        //}}AFX_MSG
```

```
afx_msg LRESULT OnOcmCtlColorBtn(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorDlg(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorEdit(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorListBox(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorMsgBox(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorScrollBar(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorStatic(WPARAM wParam, LPARAM lParam);

DECLARE_MESSAGE_MAP()
```

**注释：**

以上代码摘自头文件 AFXCTRL.H，此文件位于 vc98 \ mfc \ include 文件夹中。

2. 方法和属性

为了与 ActiveX 控件产生交互，必须向使用它的应用程序提供一系列方法（函数）和属性（接口）。方法是执行外部代码以改变控件特征的控件函数，显然，这些函数只是在设计控件时实现。控件的特征一般包括外观、属性和行为。属性一般是指控件的颜色、文本、字体和控件中使用的其他元素。方法和属性构成了应用程序（控件容器）与控件交互作用的基本结构。利用这种结构就可以使控件的外观和取值发生改变。方法和属性可以利用 ClassWizard 来定义。

### 11.1.3 ActiveX 控件与 MFC 类库

ActiveX 控件继承自 MFC 类库中的 COleControl 类。COleControl 类在 MFC 类体系中的层次关系如图 11-4 所示。



图 11-4 COleControl 类在 MFC 类库中的位置

从 COleControl 类在 MFC 类体系中的位置可以看出，该类也是一个窗口，它继承自 CWnd 类，只不过它又增加了新的数据成员和成员函数，这一点与标准控件在本质上没有区别。

## 11.2 程序代码及其作用分析

按照 11.1.1 中介绍的详细步骤，利用 Visual C++ 提供的 MFC ActiveX ControlWizard 向导生成了开发 ActiveX 控件所需的一系列初始文件，包括源文件和相应的头文件。我们主要

关心的文件有：

1. Stdafx.cpp

用于包含 ActiveX 控件所需的所有标准头文件。它的内容除注释外只有一行：

```
#include "stdafx.h"
```

2. MyActiveX.cpp

负责 AcitveX 类的实现，并注册相应的动态链接库文件(.DLL)，而其声明部分在相应的头文件中。修改部分注释后的程序代码见程序清单 11-2 和程序清单 11-3。

**程序清单 11-2：CMyActiveXApp 类声明的头文件**

```
#if ! defined(AFX_MYACTIVEX_H__19058AEC_780C_11D2_8DE6_FDDD84A6F610__INCLUDED_)
#define AFX_MYACTIVEX_H__19058AEC_780C_11D2_8DE6_FDDD84A6F610__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// MyActiveX.h : MyActiveX.DLL 的主头文件

#if ! defined( __AFXCTL_H__ )
        #error include 'afxctl.h' before including this file
#endif

#include "resource.h"          // 该控件库的全部资源符号

////////////////////////////////////////////////////////////////////////////
// CMyActiveXApp 类的声明部分，其实现部分参见文件 MyActiveX.cpp

class CMyActiveXApp : public COleControlModule
{
public:
        BOOL InitInstance();
                int ExitInstance();
};

extern const GUID CDECL _tlid;
extern const WORD _wVerMajor;
extern const WORD _wVerMinor;
```

```
    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ 将在该行之前插入其他声明语句

    #endif // !defined(AFX_MYACTIVEX_H__19058AEC_780C_11D2_8DE6_FDDD84A6F610__INCLUDED_)
```

**程序清单 11-3:CMyActiveXApp 类的实现文件**

```
    // MyActiveX.cpp : CMyActiveXApp 和 DLL 注册的实现文件

    #include "stdafx.h"
    #include "MyActiveX.h"

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


    CMyActiveXApp NEAR theApp;

    const GUID CDECL BASED_CODE _tlid =
                    { 0x19058ae3, 0x780c, 0x11d2, { 0x8d, 0xe6, 0xfd, 0xdd, 0x84, 0xa6,
0xf6, 0x10 } };
    const WORD _wVerMajor = 1;
    const WORD _wVerMinor = 0;


    ////////////////////////////////////////////////////////////////////////////
    // CMyActiveXApp::DLL 初始化

    BOOL CMyActiveXApp::InitInstance()
    {
        BOOL bInit = COleControlModule::InitInstance();//基类的初始化函数

        if (bInit)
        {
            // 添加定制的初始化代码
        }
```

```
        return bInit;
}

////////////////////////////////////////////////////////////////////////////
// CMyActiveXApp::DLL 终止

int CMyActiveXApp::ExitInstance()
{
        // 添加定制的退出代码

        return COleControlModule::ExitInstance();//调用基类的退出函数
}

////////////////////////////////////////////////////////////////////////////
// DllRegisterServer 向系统注册库添加主键

STDAPI DllRegisterServer(void)
{
        AFX_MANAGE_STATE(_afxModuleAddrThis);

        if (! AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
                return ResultFromScode(SELFREG_E_TYPELIB);

        if (! COleObjectFactoryEx::UpdateRegistryAll(TRUE))
                return ResultFromScode(SELFREG_E_CLASS);

        return NOERROR;
}

////////////////////////////////////////////////////////////////////////////
// DllUnregisterServer 从系统注册库删除主键

STDAPI DllUnregisterServer(void)
{
        AFX_MANAGE_STATE(_afxModuleAddrThis);

        if (! AfxOleUnregisterTypeLib(_tlid, _wVerMajor, _wVerMinor))
                return ResultFromScode(SELFREG_E_TYPELIB);
```

```
    if (! COleObjectFactoryEx::UpdateRegistryAll(FALSE))
        return ResultFromScode(SELFREG_E_CLASS);

    return NOERROR;
}
```

3. MyActiveXCtl.cpp 文件

该文件提供了 ActiveX 控件的 OLE 类的真正实现。在这个例子中,这个类就是 CMyActiveXCtrl。开发控件的大部分工作都将花费在对此文件的修改上。同时也可以在此文件实现用户定制的 ActiveX 控件,该文件是真正的 OLE 类的实现部分,其声明部分见 MyActiveXCtl.h,程序清单 11-4 和程序清单 11-5 分别列出了这两个程序的完整清单。

**程序清单 11-4:控件类 MyActicveXCtl 的实现文件 MyActiveXCtl.cpp**

```
// MyActiveXCtl.cpp : 实现 CMyActiveXCtrl 控件类

#include "stdafx.h"
#include "MyActiveX.h"
#include "MyActiveXCtl.h"
#include "MyActiveXPpg.h"


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


IMPLEMENT_DYNCREATE(CMyActiveXCtrl, COleControl)


////////////////////////////////////////////////////////////////////////////
// 消息映射

BEGIN_MESSAGE_MAP(CMyActiveXCtrl, COleControl)
    //{{AFX_MSG_MAP(CMyActiveXCtrl)
    // ClassWizard 将添加或删除映射条目
    //}}AFX_MSG_MAP
    ON_OLEVERB(AFX_IDS_VERB_EDIT, OnEdit)
    ON_OLEVERB(AFX_IDS_VERB_PROPERTIES, OnProperties)
```

```
END_MESSAGE_MAP()


////////////////////////////////////////////////////////////////////////////
// 分配映射

BEGIN_DISPATCH_MAP(CMyActiveXCtrl, COleControl)
        //{{AFX_DISPATCH_MAP(CMyActiveXCtrl)
        //ClassWizard 将添加或删除映射条目
        //}}AFX_DISPATCH_MAP
        DISP_FUNCTION_ID(CMyActiveXCtrl, "AboutBox", DISPID_ABOUTBOX, About-
Box, VT_EMPTY, VTS_NONE)
END_DISPATCH_MAP()


////////////////////////////////////////////////////////////////////////////
//事件映射

BEGIN_EVENT_MAP(CMyActiveXCtrl, COleControl)
        //{{AFX_EVENT_MAP(CMyActiveXCtrl)
        // NOTE - ClassWizard will add and remove event map entries
        // DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_EVENT_MAP
END_EVENT_MAP()


////////////////////////////////////////////////////////////////////////////
// 属性页

// 添加所需的属性页
BEGIN_PROPPAGEIDS(CMyActiveXCtrl, 1)
        PROPPAGEID(CMyActiveXPropPage::guid)
END_PROPPAGEIDS(CMyActiveXCtrl)


////////////////////////////////////////////////////////////////////////////
// 初始化类代理

IMPLEMENT_OLECREATE_EX(CMyActiveXCtrl, "MYACTIVEX.MyActiveXCtrl.1",
        0x614bfbbe, 0x79a5, 0x11d2, 0x95, 0x93, 0x84, 0x90, 0x93, 0xdd, 0x4b, 0x36)
```

```
////////////////////////////////////////////////////////////////////////////
// 类型库标识和版本

IMPLEMENT_OLETYPELIB(CMyActiveXCtrl, _tlid, _wVerMajor, _wVerMinor)


////////////////////////////////////////////////////////////////////////////
// 接口标识

const IID BASED_CODE IID_DMyActiveX =
        { 0x614bfbbc, 0x79a5, 0x11d2, { 0x95, 0x93, 0x84, 0x90, 0x93, 0xdd,
0x4b, 0x36 } };
const IID BASED_CODE IID_DMyActiveXEvents =
        { 0x614bfbbd, 0x79a5, 0x11d2, { 0x95, 0x93, 0x84, 0x90, 0x93, 0xdd,
0x4b, 0x36 } };


////////////////////////////////////////////////////////////////////////////
// 控件类型信息

static const DWORD BASED_CODE _dwMyActiveXOleMisc =
        OLEMISC_ACTIVATEWHENVISIBLE |
        OLEMISC_SETCLIENTSITEFIRST |
        OLEMISC_INSIDEOUT |
        OLEMISC_CANTLINKINSIDE |
        OLEMISC_RECOMPOSEONRESIZE;

IMPLEMENT_OLECTLTYPE(CMyActiveXCtrl, IDS_MYACTIVEX, _dwMyActiveXOleMisc)


////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::CMyActiveXCtrlFactory::UpdateRegistry -
// 添加或删除 CMyActiveXCtrl 的系统注册条目

BOOL CMyActiveXCtrl::CMyActiveXCtrlFactory::UpdateRegistry(BOOL bRegister)
{

        if (bRegister)
                return AfxOleRegisterControlClass(
                        AfxGetInstanceHandle(),
```

```
                        m_clsid,
                        m_lpszProgID,
                        IDS_MYACTIVEX,
                        IDB_MYACTIVEX,
                        afxRegInsertable | afxRegApartmentThreading,
                        _dwMyActiveXOleMisc,
                        _tlid,
                        _wVerMajor,
                        _wVerMinor);
        else
                return AfxOleUnregisterClass(m_clsid, m_lpszProgID);
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::CMyActiveXCtrl - 构造函数

CMyActiveXCtrl::CMyActiveXCtrl()
{
        InitializeIIDs(&IID_DMyActiveX, &IID_DMyActiveXEvents);

        //初始化控件的实例数据
}

////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::CMyActiveXCtrl - 析造函数

CMyActiveXCtrl::CMyActiveXCtrl()
{
        //清除控件的析构函数
}
////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::OnDraw - Drawing function

void CMyActiveXCtrl::OnDraw(
                        CDC * pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
        // 在用户的绘制代码中放入以下代码
         pdc - > FillRect(rcBounds, CBrush::FromHandle((HBRUSH)GetStockObject
(WHITE_BRUSH)));
        pdc - > Ellipse(rcBounds);
```

```
}

////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::DoPropExchange - 支持代码

void CMyActiveXCtrl::DoPropExchange(CPropExchange* pPX)
{
        ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
        COleControl::DoPropExchange(pPX);
}

////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::OnResetState - 重设控件为缺省状态

void CMyActiveXCtrl::OnResetState()
{
        COleControl::OnResetState();

        //重设其他控件的状态
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl::AboutBox - 显示 About 对话框

void CMyActiveXCtrl::AboutBox()
{
        CDialog dlgAbout(IDD_ABOUTBOX_MYACTIVEX);
        dlgAbout.DoModal();
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXCtrl 消息处理程序
```

**程序清单 11-5:控件类 MyActicveXCtl 的声明文件 MyActiveXCtl.h**

```
#if ! defined(AFX_MYACTIVEXCTL_H__614BFBCC_79A5_11D2_9593_849093DD4B36__
INCLUDED_)
#define AFX_MYACTIVEXCTL_H__614BFBCC_79A5_11D2_9593_849093DD4B36__INCLUD-
```

```
ED_

    #if _MSC_VER > 1000
    #pragma once
    #endif // _MSC_VER > 1000

    // MyActiveXCtl.h : 声明 CMyActiveXCtrl ActiveX 控件类

    /////////////////////////////////////////////////////////////////////////////
    // CMyActiveXCtrl : 参见 MyActiveXCtl.cpp 的实现部分

    class CMyActiveXCtrl : public COleControl
    {
            DECLARE_DYNCREATE(CMyActiveXCtrl)

    // 构造函数
    public:
            CMyActiveXCtrl();

    // 重载
            // ClassWizard 生成虚拟函数重载
            //{{AFX_VIRTUAL(CMyActiveXCtrl)
            public:
            virtual void OnDraw(CDC* pdc, const CRect& rcBounds, const CRect& rcIn-
valid);
            virtual void DoPropExchange(CPropExchange* pPX);
            virtual void OnResetState();
            //}}AFX_VIRTUAL

    // 实现方式
    protected:
            CMyActiveXCtrl();

            DECLARE_OLECREATE_EX(CMyActiveXCtrl)
            DECLARE_OLETYPELIB(CMyActiveXCtrl)
            DECLARE_PROPPAGEIDS(CMyActiveXCtrl)
            DECLARE_OLECTLTYPE(CMyActiveXCtrl)

    // 消息映射
            //{{AFX_MSG(CMyActiveXCtrl)
                    // ClassWizard 将添加或删除成员函数
```

```
        //{{AFX_MSG
        DECLARE_MESSAGE_MAP()

    // 分配映射
        //{{AFX_DISPATCH(CMyActiveXCtrl)
            // ClassWizard 将添加或删除成员函数
        //}}AFX_DISPATCH
        DECLARE_DISPATCH_MAP()

        afx_msg void AboutBox();

    // 事件映射
        //{{AFX_EVENT(CMyActiveXCtrl)
            //ClassWizard 将添加或删除成员函数
        //}}AFX_EVENT
        DECLARE_EVENT_MAP()

    // 分配事件 ID 号
    public:
        enum {
        //{{AFX_DISP_ID(CMyActiveXCtrl)
            //ClassWizard 将添加或删除枚举元素
        //}}AFX_DISP_ID
        };
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ 将插入附加的声明语句

#endif // !defined(AFX_MYACTIVEXCTL_H__614BFBCC_79A5_11D2_9593_849093DD4B36__INCLUDED)
```

在程序清单 11-4 中，我们发现除了消息映射外，控件类与一般类的实现相比增加了分配映射和事件映射。这三种映射均由向导自动生成，用户不必直接修改。消息映射与一般应用程序的消息映射功能相同，分配映射实现了在几个应用程序之间调用方法和属性的手段。事件映射协助处理 ActiveX 控件事件。

4. MyActiveXPRG.CPP 和 MyActiveXPRG.H

控件属性页类的实现文件 MyActiveXPRG.CPP 和头文件 MyActiveXPRG.H 分别用于声明和实现属性页类。该类用于控制控件的属性，它继承自 COlePropertyPage 类。完整程序代码分别见程序清单 11-6 和程序清单 11-7。

**程序清单 11-6:属性页类的实现文件 MyActiveXPRG.CPP**

```
// MyActiveXPpg.cpp : 实现 CMyActiveXPropPage 属性页类

#include "stdafx.h"
#include "MyActiveX.h"
#include "MyActiveXPpg.h"

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


IMPLEMENT_DYNCREATE(CMyActiveXPropPage, COlePropertyPage)


////////////////////////////////////////////////////////////////////////////
// 消息映射

BEGIN_MESSAGE_MAP(CMyActiveXPropPage, COlePropertyPage)
        //{{AFX_MSG_MAP(CMyActiveXPropPage)
        //ClassWizard 将添加或删除消息映射条目
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()


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

IMPLEMENT_OLECREATE_EX(CMyActiveXPropPage, "MYACTIVEX.MyActiveXPropPage.1",
        0x614bfbbf, 0x79a5, 0x11d2, 0x95, 0x93, 0x84, 0x90, 0x93, 0xdd, 0x4b, 0x36)


////////////////////////////////////////////////////////////////////////////
// CMyActiveXPropPage::CMyActiveXPropPageFactory::UpdateRegistry -
// 添加或删除 CMyActiveXPropPage 的系统注册条目

BOOL CMyActiveXPropPage:: CMyActiveXPropPageFactory:: UpdateRegistry ( BOOL
bRegister)
```

```
{
        if (bRegister)
                return AfxOleRegisterPropertyPageClass(AfxGetInstanceHandle
  (),m_clsid, IDS_MYACTIVEX_PPG);
        else
                return AfxOleUnregisterClass(m_clsid, NULL);
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXPropPage::CMyActiveXPropPage - 构造函数

CMyActiveXPropPage::CMyActiveXPropPage() :
        COlePropertyPage(IDD, IDS_MYACTIVEX_PPG_CAPTION)
{
        //{{AFX_DATA_INIT(CMyActiveXPropPage)
        // ClassWizard 将添加成员初始化代码
        //}}AFX_DATA_INIT
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXPropPage::DoDataExchange - 移动数据

void CMyActiveXPropPage::DoDataExchange(CDataExchange* pDX)
{
        //{{AFX_DATA_MAP(CMyActiveXPropPage)
        //ClassWizard 将添加 DDP, DDX 和 DDV 调用
        //}}AFX_DATA_MAP
        DDP_PostProcessing(pDX);
}


////////////////////////////////////////////////////////////////////////////
// CMyActiveXPropPage 消息处理程序
```

**程序清单 11-7:属性页类的声明文件 MyActiveXPRG.H**

```
#if ! defined(AFX_MYACTIVEXPPG_H__614BFBCE_79A5_11D2_9593_849093DD4B36__
INCLUDED_)
#define AFX_MYACTIVEXPPG_H__614BFBCE_79A5_11D2_9593_849093DD4B36__INCLUD-
```

```
ED_

    #if _MSC_VER > 1000
    #pragma once
    #endif // _MSC_VER > 1000

    // MyActiveXPpg.h : 声明 CMyActiveXPropPage 属性页类

    ////////////////////////////////////////////////////////////////////////////
    // CMyActiveXPropPage :参见 MyActiveXPpg.cpp 的实现部分

    class CMyActiveXPropPage : public COlePropertyPage
    {
            DECLARE_DYNCREATE(CMyActiveXPropPage)
            DECLARE_OLECREATE_EX(CMyActiveXPropPage)

    // 构造函数
    public:
            CMyActiveXPropPage();

    //对话框数据
            //{{AFX_DATA(CMyActiveXPropPage)
            enum { IDD = IDD_PROPPAGE_MYACTIVEX };
                    //ClassWizard 将添加数据成员
            //}}AFX_DATA

    // 实现方式
    protected:
            virtual void DoDataExchange(CDataExchange* pDX);
    // 消息映射
    protected:
            //{{AFX_MSG(CMyActiveXPropPage)
                    // ClassWizard 将添加或删除成员函数
            //}}AFX_MSG
            DECLARE_MESSAGE_MAP()

    };

    //{{AFX_INSERT_LOCATION}}
    // Microsoft Visual C++ 将插入附加的声明语句
```

```
#endif // ! defined(AFX_MYACTIVEXPPG_H__614BFBCE_79A5_11D2_9593_
849093DD4B36__INCLUDED)
```

## 11.3 定制实现特定功能的 ActiveX 控件

在上一节利用向导生成了开发 ActiveX 控件的初始文件。但此时应用程序除了在容器应用程序中显示一个控件外，再没有什么具体功能了。为了使 ActiveX 控件实现某种具体功能，必须对前面的程序做进一步修改，这些修改包括修改控件的外观使其显示控件的特定功能对应的内容，重新定义控件的事件处理函数使其能够正确处理该控件的特定事件。

### 11.3.1 改变控件的外观

控件的外观一般包括形状、大小和颜色。在集成开发环境中通过 View 菜单的 ClassWizard 命令，遵照下面的操作步骤即可改变控件的外观：

(1) 在 MFC ClassWizard 对话框内激活 Automation 选项卡，如图 11-5 所示。

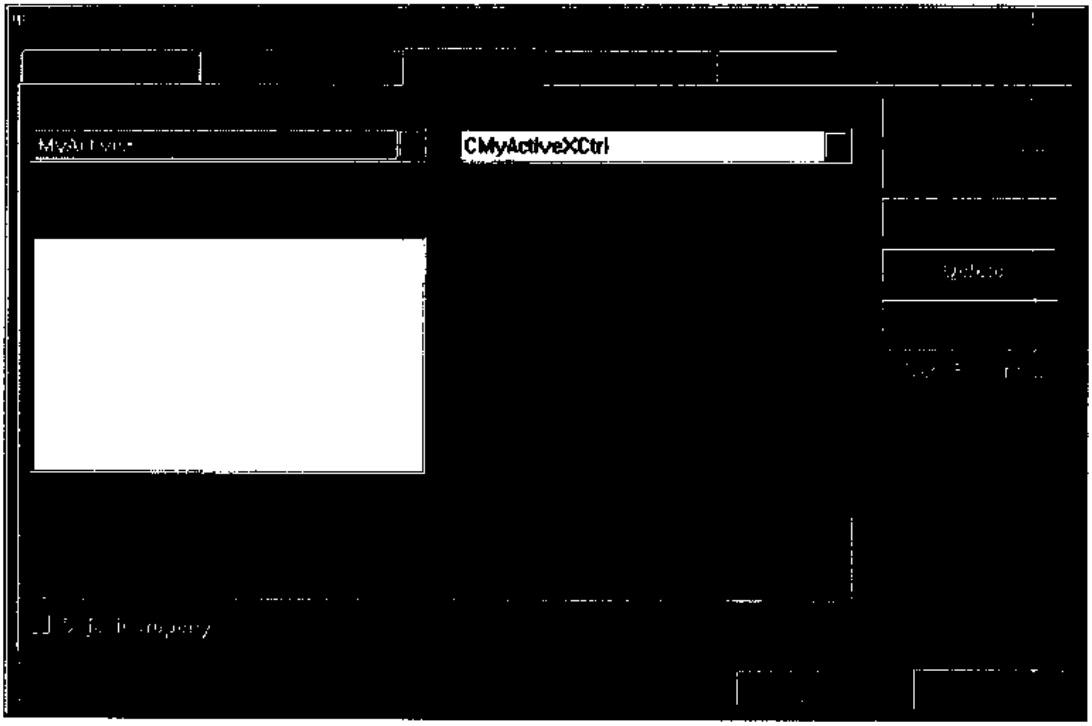



图 11-5 Automation 选项卡

(2) 在 Class name 下拉列表框中选择 CMyActiveXCtrl

(3) 单击 Add Property 按钮以显示 Add Property 对话框，如图 11-6 所示。

(4) 在 External name 文本框中输入 CtrlShape 作为外部名。

(5) 选择 Implementation 组框中的 Member variable(成员变量)单选按钮。

(6) 在 Type 下拉列表框中选择 BOOL。这时在 Notification function(通知函数)文本框中

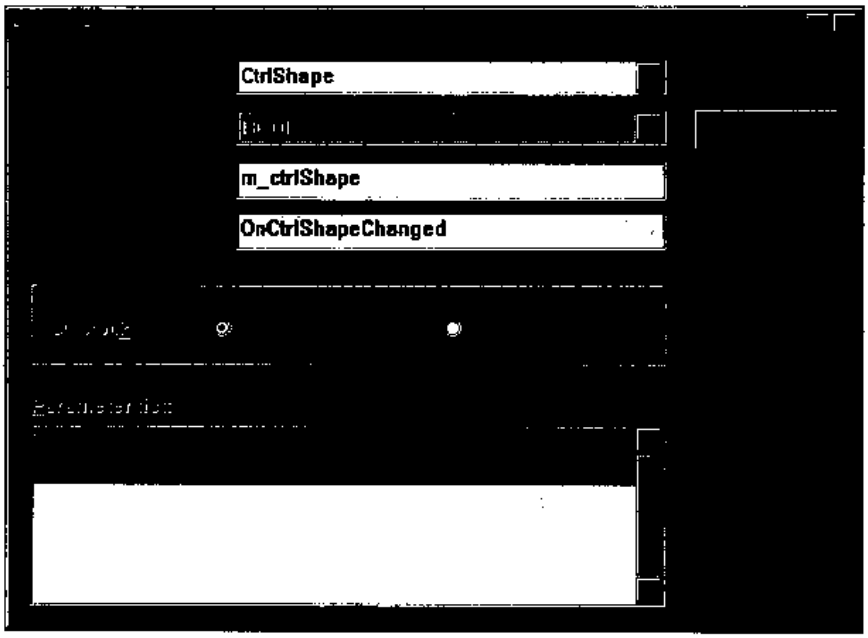



图 11-6　Add Property 对话框

显示 OnCtrlShapeChanged,成员变量自动变为 m _ ctrlShape。

(7) 单击 OK 按钮,返回 MFC ClassWizard 对话框的 Automation 选项卡。

(8) 再次单击 Add Property 按钮,显示 Add Property 对话框。

(9) 在 External name 下拉列表框中选择 BackColor。

(10) 选择 Implementation 组框中的 Stock(插口)单选按钮。

(11) 单击 OK 按钮,返回 MFC ClassWizard 对话框的 Automation 选项卡。

(12) 单击 OK 按钮接受以上设置并关闭 MFC ClassWizard 对话框。

此时我们来查看项目中有关的文件代码,可以发现在文件 MyActiveXCtl.H 中的分配映射中多了如下的程序代码:

```
BOOL m_ctrlShape;
afx_msg void OnCtrlShapeChanged();
```

在分配和事件 ID 部分增加了如下代码:

```
dispidCtrlShape = 1L;
```

在文件 MyActiveXCtl.CPP 中,有此函数的实现框架,只不过函数主体部分只有一个函数调用,必须增加实际的实现代码。程序代码如下:

```
void CMyActiveXCtrl::OnCtrlShapeChanged()
{
        //添加通知处理程序代码

        SetModifiedFlag();
```

此外，在文件 MyActiveXCtl.CPP 开头处的分配映射中增加了如下两个宏调用：

```
DISP_PROPERTY_NOTIFY(CMyActiveXCtrl, "CtrlShape", m_ctrlShape, OnCtrl-
ShapeChanged, VT_BOOL)
DISP_STOCKPROP_BACKCOLOR()
```

DISP_PROPERTY_NOTIFY 宏实现了 OLE 自动化通知属性。在本例中当拥有该控件的容器应用程序改变外部名的取值时，控件类的成员变量 m_ctrlShape 将改变，从而自动调用 OnCtrlShapeChanged 函数。

要实现控件外观的改变，必须修改控件类的成员函数 OnDraw，代码见程序清单 11-8 所示。

**程序清单 11-8：控件类的成员函数 OnDraw**

```
void CMyActiveXCtrl::OnDraw(
                        CDC * pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
        //添加用户的绘制代码
        CBrush * pOldBrush;
        CBrush NewBrush;
        CPen NewPen;
        Cpen * pOldPen;
        pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)GetStockObject
(WHITE_BRUSH)));

        //改变画笔风格
        NewPen.CreatePen(PS_SOLID,3,RGB(0,0,0));
        pOldPen=(CPen *)pdc->SelectObject(&NewPen);

        //建立黄色笔刷
        NewBrush.CreateSolidBrush(RGB(255,255,0));
        pOldBrush=(CBrush *)pdc->SelectObject(&NewBrush);

        //绘制并填充矩形
        pdc->Rectangle(rcBounds);

        //恢复原来的画笔和笔刷
        pdc->SelectObject(pOldBrush);
        pdc->SelectObject(pOldPen);

}
```

### 11.3.2　实现特定事件处理函数

对于一个控件而言，事件有很多种类，例如控件被单击、双击等。下面介绍如何使 MyActiveX 控件对鼠标事件作出反应，以便当光标位于控件上时若按下鼠标左键能够报告系统的日期和时间。

为了实现对单击事件的反应，可以按如下步骤进行：

(1) 打开 MFC ClassWizard 对话框，并激活 Automation 选项卡。

(2) 在 Class name 列表框中选择 CMyActiveXCtrl。

(3) 单击 Add Property 按钮，打开 Add Property 对话框。

(4) 在 External name 文本框中输入 HitCtrl。

(5) 在 Implementation 组合中选择 Member variable 单选按钮。

(6) 在 Type 下拉列表框中选择 OLE _ COLOR，清除 Notification function 文本框。

(7) 单击 OK 按钮关闭 Add Property 对话框，返回 Automation 选项卡，如图 11-7 所示。

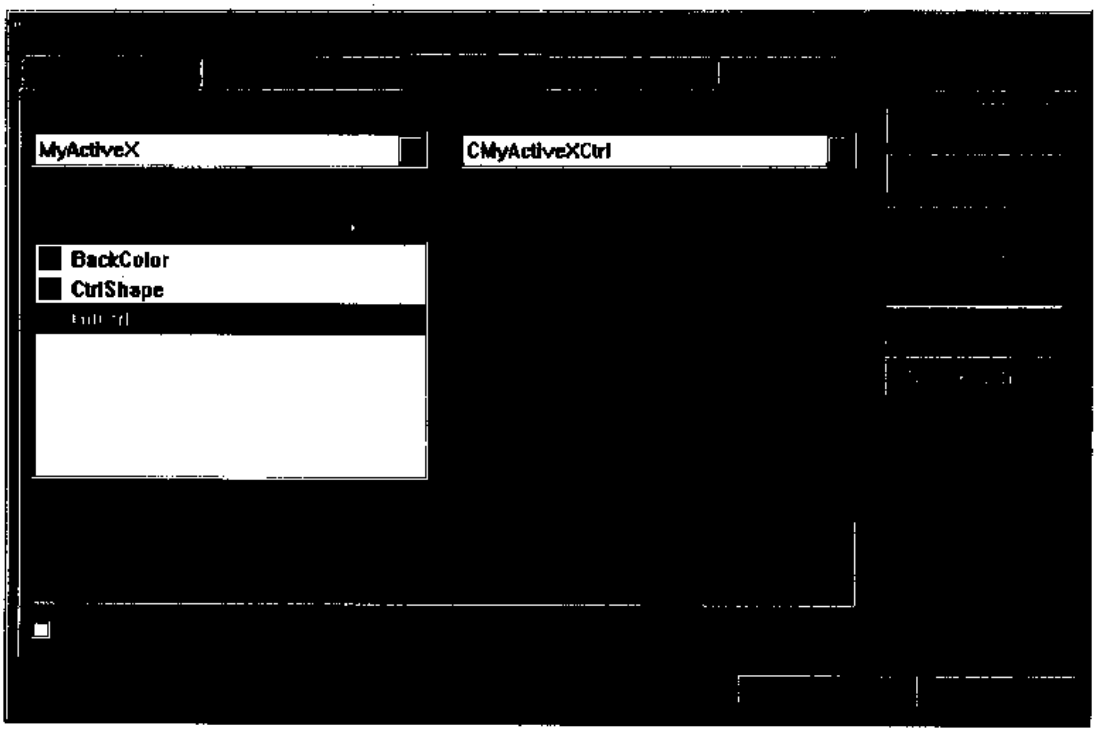



图 11-7　添加鼠标响应事件后的 Automation 选项卡

(8) 激活 Message Maps 选项卡以设置消息映射，如图 11-8 所示。

(9) 在 Class name 下拉列表框中选择 CMyActiveXCtrl。

(10) 在 Object IDs 列表框中选择 CMyActiveXCtrl

(11) 在 Messages 列表框中选择 WM _ LBUTTONDOWN，单击 Add Function 按钮，然后再选择 WM _ LBUTTONUP，再单击 Add Function 按钮。

(12) 选择 OK 按钮。

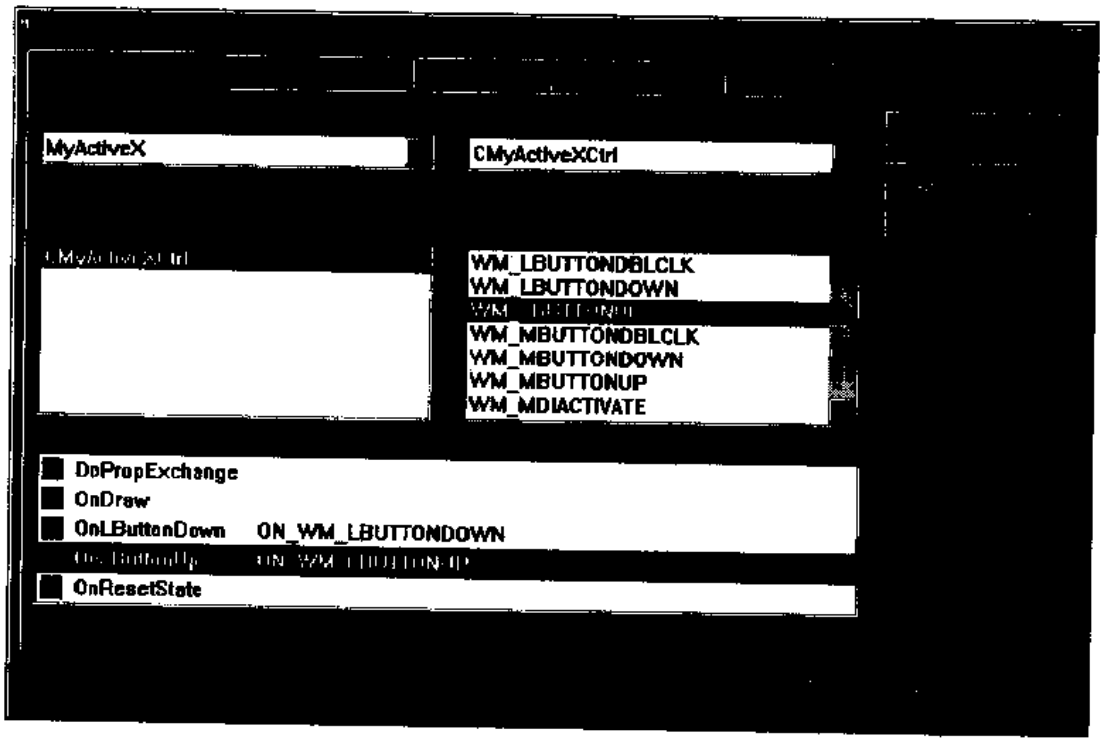



图 11-8 为控件添加消息映射的 Message Maps 选项卡

经过以上各步，ClassWizard 自动生成代码，主要在 CMyActiveXCtrl 类声明中增加了成员变量 m _ hitCtrl，成员函数 OnLButtonDown 和 OnLButtonUp，而在 CMyActiveXCtrl 类实现文件中增加了两个成员函数的基本框架。这两个成员函数就是事件处理函数。要实现这两个成员函数，必须在控件类 CMyActiveXCtrl 的声明文件中增加两个函数，这两个函数一个用来确定鼠标事件是否发生在控件内部，另一个用来当发生单击事件时改变控件的颜色，因此应在控件类的头文件的析构函数之后加入如下代码：

```
BOOL InMyActiveX(CPoint & point);
void HitMyActiveX(CDC * pdc);
```

这两个函数的实现部分在控件类的实现文件中，程序代码见程序清单 11-9 所示。

**程序清单 11-9：判断鼠标在控件内是否按下和改变控件颜色的函数的实现代码**

```
BOOL CMyActiveXCtrl::InCtrl(CPoint &point)
{

        return TRUE;

}


void CMyActiveXCtrl::HitCtrl(CDC * pdc)
{

        CBrush * pOldBrush;
```

```
    CBrush hitBrush(TranslateColor(m_hitCtrl));
    CRect rc;
    TEXTMETRIC tm;
    struct tm * date_time;
    time_t timer;

    //背景设置为透明
    pdc->SetBkMode(TRANSPARENT);
    GetClientRect(rc);

    pOldBrush=pdc->SelectObject(&hitBrush);

    //绘制并填充矩形
    pdc->Rectangle(rc);

    //得到日期和时间
    time(&timer);
    date_time=localtime(&timer);
    const CString strtime=asctime(date_time);

    //得到字体信息并打印
    pdc->GetTextMetrics(&tm);
    pdc->SetTextAlign(TA_CENTER|TA_TOP);
     pdc->ExtTextOut((rc.left+rc.right)/2,(rc.top+rc.bottom-tm.
tmHeight)/2,ETO_CLIPPED,rc,strtime,strtime.GetLength()-1,NULL);
    pdc->SelectObject(pOldBrush);

}
```

当鼠标左键在该控件区域按下时，调用 HitCtrl 函数在控件中显示系统时间，当放开时则重画该控件以恢复控件原貌。这两个事件处理函数的代码见程序清单 11-10 所示。

**程序清单 11-10：鼠标左键按下和放开的事件处理函数代码**

```
void CMyActiveXCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    //添加消息处理程序代码
    CDC *pdc;

    if(InCtrl(point)){
        pdc=GetDC();
        HitCtrl(pdc);
```

```
            ReleaseDC(pdc);
        }

        COleControl::OnLButtonDown(nFlags, point);
}

void CMyActiveXCtrl::OnLButtonUp(UINT nFlags, CPoint point)
{
        //添加消息处理程序代码

        if(InCtrl(point))
            InvalidateControl();

        COleControl::OnLButtonUp(nFlags, point);
}
```

通过程序清单 11-10 可以看出,当鼠标左键按下时调用控件类的成员函数 HitCtrl,并由此函数在控件区域内绘制系统时间。在鼠标左键按下的事件处理函数中笔刷 hitBrush 的建立利用了控件类的成员变量 m_hitCtrl,该变量的取值来自于控件类的一个非常重要的成员函数,此函数就是 DoProExchange。DoProExchange 函数中有一条语句(见程序清单 11-11 中的相关注释)专门用于实现外部变量与控件类成员变量之间的数据交换,并可以对变量赋于初始值。

**程序清单 11-11:控件类的成员函数 DoProExchange 的实现**

```
void CMyActiveXCtrl::DoPropExchange(CPropExchange * pPX)
{
        ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
        COleControl::DoPropExchange(pPX);

        //当单击时用浅灰色显示
        PX_Long(pPX,_T("HitCtrl"),(long &)m_hitCtrl,RGB(200,200,200));//此语句
实现外观名与成员变量间的数据交换

}
```

## 11.4 测试 MyActiveX 控件

Visual C++ 6.0 提供了专门用于测试 ActiveX 控件的实用工具,这就是 ActiveX 控件测试容器。选择该工具的 Edit 菜单的 Insert New Control 命令,将打开如图 11-9 所示的 Insert Control 对话框。

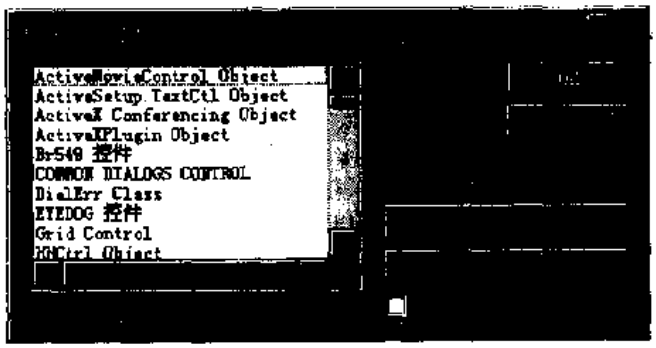



图 11-9　Insert Control 对话框

在 Insert Control 对话框的控件列表框中选择 MyActiveX Control，然后单击 OK 按钮，本章所开发的控件即出现在测试容器的主窗口中，此时控件显示为黄色矩形，当鼠标在控件上按下左键时，显示结果如图 11-10 所示。

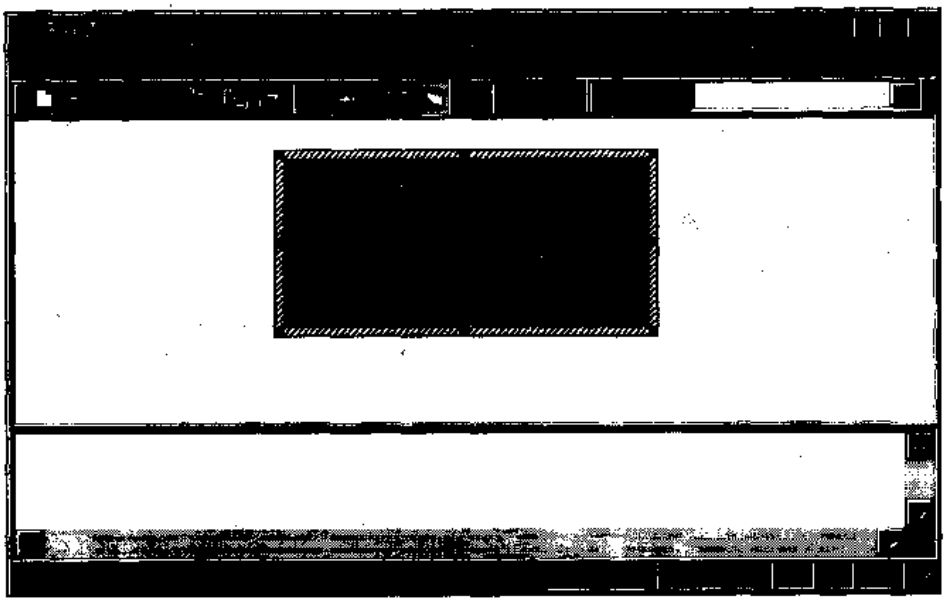


图 11-10　测试 MyActive 控件

当然，也可以将此控件插入到 Word 文档中。在 Word 中利用其“插入”菜单的“对象”命令，然后在对象类型列表框中选择 MyActiveX Control，最后，单击“确定”按钮。新的控件即被插入到 Word 文档中，如图 11-11 所示。当鼠标放在控件上按下左键时，将显示当前的时间和日期，如图 11-12 所示。

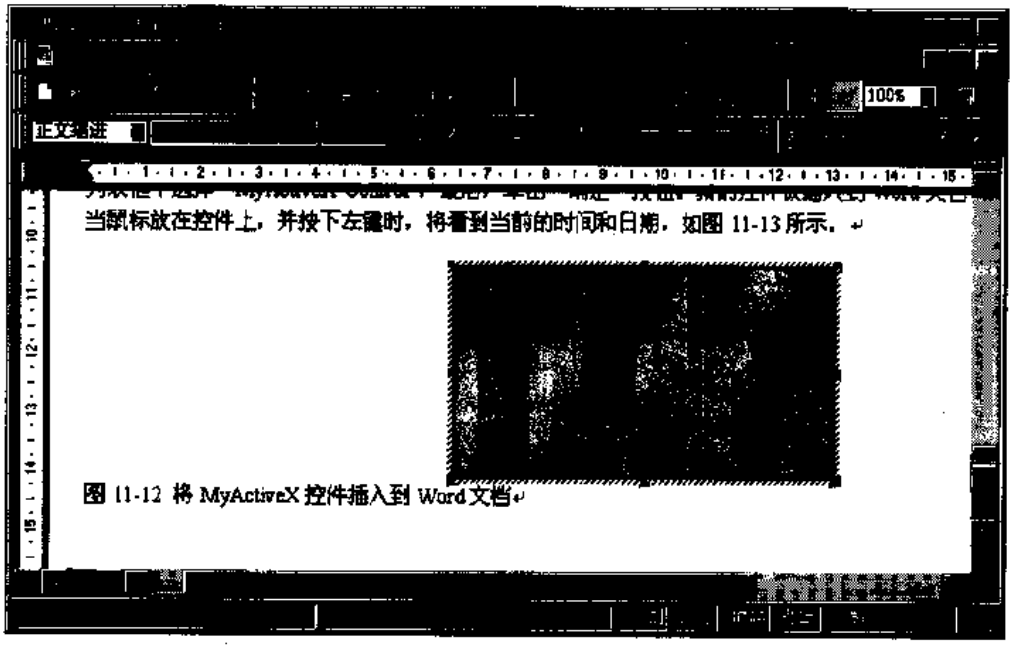



图 11-11　将 MyActiveX 控件插入到 Word 文档

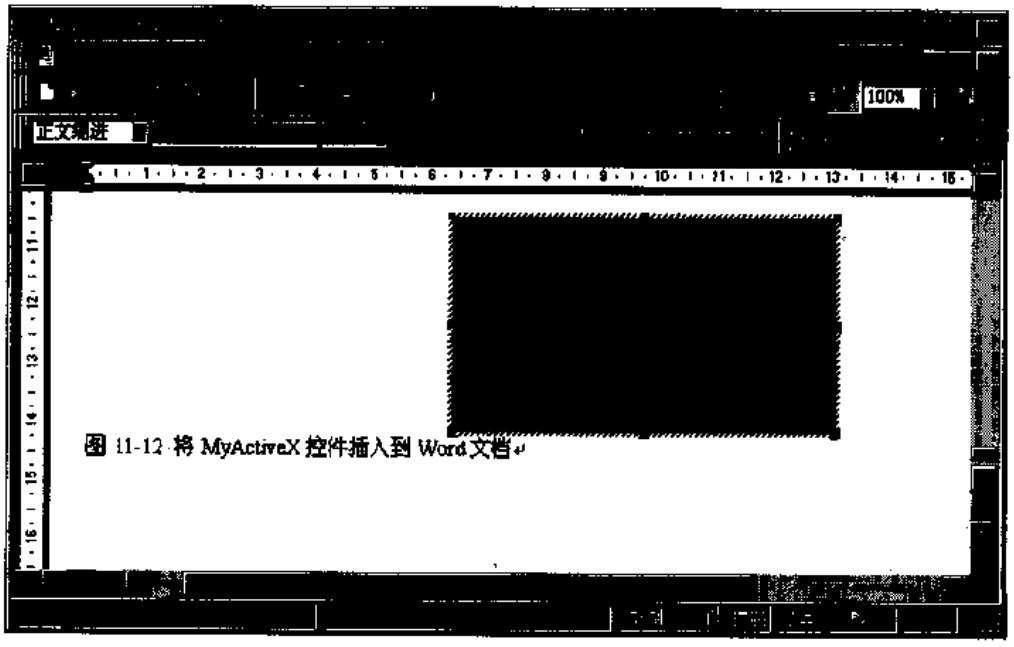



图 11-12　对控件按下鼠标左键时，控件显示时间和日期

# 附录

# 命令与快捷键

本附录列出了在创建各种 Visual C++ 项目时用到的各种按钮和菜单命令及其对应的快捷键。利用这些快捷键可以加快应用程序的开发速度，提高应用程序的开发效率。虽然有的命令没有快捷键，但为了保持 Visual C++ 命令集的完整性，也一并在这里列出，因此也可以把本附录作为快速命令索引表来使用。

**表 B-1 用于创建(Build)项目的命令与快捷键**

| 命令 | 快捷键 | 意义 |
|---|---|---|
| Build | F7 | 创建项目 |
| BuildBatch | | 创建多个项目 |
| BuildClean | | 删除中间文件和输出文件，即清理项目 |
| BuildCompile | Ctrl + F7 | 只编译文件，而不进行链接 |
| BuildConfigurations | | 编辑项目配置 |
| BuildExecute | Ctrl + F5 | 执行程序 |
| BuildRebuildAll | | 重新创建当前项目及其所有子项目 |
| BuildSetActiveConfiguration | | 选择活动项目并对其进行配置 |
| BuildStop | Ctrl + Break | 终止创建项目 |
| BuildToggleBuild | | 在创建的项目间比拟 |

**表 B-2 用于调试(debug)项目的命令与快捷键**

| 命令 | 快捷键 | 意义 |
|---|---|---|
| AttachToActive | | 跟踪正在运行的程序 |
| Go | F5 | 启动或继续程序的运行 |
| RunToCursor | Ctrl + F10 | 运行程序到光标所在行 |
| StepInto | F11 | 单步执行到下一条语句 |
| ToolsProfile | | 设置剖视器(Profile)选项，剖视程序并显示剖视数据 |
| ToolsRemoteConnection | | 编辑远程调试的链接设置 |
| ApplyCodeChanges | Alt + F10 | 使在调试过程中对源程序的修改有效 |
| DebugBreak | | 停止程序的执行，中止调试过程 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| DebugDisableAllBreakpoints | | 禁止所有的断点 |
| DebugEnableBreakpoint | Ctrl + F9 | 使能或禁止一个断点 |
| DebugExceptions | | 当发生异常时,修改调试行为 |
| DebugGo | F5 | 启动或继续程序 |
| DebugHexadecimalDisplay | | 十进制和十六制格式切换 |
| DebugMemoryNextFormat | Alt + F11 | 以下一种格式显示内存窗口 |
| DebugMemoryPrevFormat | Alt + Shift + F11 | 以上一种格式显示内存窗口 |
| DebugModules | | 显示当前装入的模块 |
| DebugQuickWatch | Shift + F9 | 立即对变量和表达式求值 |
| DebugRe-moveAllBreakpoints | Ctrl + Shift + F9 | 删除所有断点 |
| DebugRestart | Ctrl + Shift + F5 | 重新启动程序 |
| DebugRunToCursor | Ctrl + F10 | 运行程序到光标所在行 |
| DebugSetNextStatement | Ctrl + Shift + F10 | 设置指令指针为光标所在行 |
| DebugShowNextStatement | Alt + Num * | 显示当前指令指针所对应的源程序行 |
| DebugStepInto | F11 | 单步执行一条语句 |
| DebugStepIn-toSpecificFunction | | 单步执行所选择的函数 |
| DebugStepOut | Shift + F11 | 执行到当前函数的退出点处 |
| DebugStepOver | F10 | 跳过下一条语句 |
| DebugStepOverSource | | 跳过下一条源程序级语句 |
| DebugStopDebugging | Shift + F5 | 终止调试程序 |
| DebugThreads | | 设置调试线程属性 |
| DebugToggleBreakpoint | F9 | 插入或删除断点 |
| DebugToggleMixedMode | Ctrl + F11 | 对当前指令在源程序查看和反汇编查看之间切换 |
| UpdateImageToggle | | 使调试期间对源代码的修改有效 |

表 B-3 与编辑有关的命令及快捷键

(括号中的 Epsilon 和 Brief 表示仿真编辑器类型)

| 命令 | 快捷键 | 意义 |
|---|---|---|
| BookmarkClearAll | Ctrl + Shift + F2 | 清除窗口中的所有书签 |
| BookmarkDrop(Epsilon) | | 设置一个 Epsilon 书签 |
| BookmarkDrop1(Brief) | | 设置 Brief 书签 1 |
| BookmarkDrop10(Brief) | | 设置 Brief 书签 10 |
| BookmarkDrop2(Brief) | | 设置 Brief 书签 2 |
| BookmarkDrop3(Brief) | | 设置 Brief 书签 3 |
| BookmarkDrop4(Brief) | | 设置 Brief 书签 4 |
| BookmarkDrop5(Brief) | | 设置 Brief 书签 5 |
| BookmarkDrop6(Brief) | | 设置 Brief 书签 6 |
| BookmarkDrop7(Brief) | | 设置 Brief 书签 7 |
| BookmarkDrop8(Brief) | | 设置 Brief 书签 8 |
| BookmarkDrop9(Brief) | | 设置 Brief 书签 9 |
| BookmarkJumpToLast (Epsilon) | | 移到前一个 Epsilon 书签 |
| BookmarkNext | F2 | 移动光标到包含下一个书签的行 |
| BookmarkPrev | Shift + F2 | 移动光标到包含上一个书签的行 |
| BookmarkToggle | Ctrl + F2 | 对当前行设置或取消书签 |
| Bookmarks | Alt + F2 | 编辑或操作书签 |
| BrowseGoToDefinition | F12 | 显示个一个符号的定义 |
| BrowseGoToReference | Shift + F12 | 显示一个符号的引用 |
| BrowseNext | Ctrl + Num + | 显示下一个符号的定义或引用 |
| BrowsePopContext | Ctrl + Num * | 返回最后执行浏览(browse)操作的位置 |
| BrowsePrev | Ctrl + Num - | 显示上一个符号定义或引用 |
| CharLeft | Left Arrow | 光标左移一个字符 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| CharLeftExtend | Shift + Left Arrow | 向左扩展选择一个字符 |
| CharRight | Right Arrow | 光标右移一个字符 |
| CharRightExtend | Shift + Right Arrow | 向右扩展选择一个字符 |
| CharTranspose | | 围绕插入点交换字符 |
| CompleteWord | Ctrl + Space | 完成当前语句 |
| ConditionalDown | Ctrl + K | 查找下一个匹配的预处理条件 |
| ConditionalDownExtend | Ctrl + Shift + K | 扩展选择到下一个匹配的预处理条件 |
| ConditionalUp | Ctrl + J | 查找上一个匹配的预处理条件 |
| ConditionalUpExtend | Ctrl + Shift + J | 扩展选择到上一个匹配的预处理条件 |
| Copy | Ctrl + C | 把当前选择复制到剪贴板 |
| Cut | Ctrl + X | 把当前选择删除并移到剪贴板 |
| CutAppendNext | | 把当前选择删除并追加到剪贴板 |
| CutSelection | | 把当前选择删除并移到剪贴板 |
| DebugBreakpoints | Alt + F9 | 编辑程序中的断点 |
| Delete | Del | 删除当前选定的内容 |
| Delete Column | | 删除表列 |
| DeleteBack | Backspace | 删除选定内容,如果没有选择则删除光标左边的字符 |
| DeleteBlankLines | | 删除相邻空行到光标处 |
| DeleteHorizontalSpace | | 删除空格和制表符到光标处 |
| DeleteRelationship -FromDatabase | | 删除选定的数据库间的联系 |
| DeleteTableFromDatabase | | 删除数据库中选择的表 |
| DocumentEnd | Ctrl + End | 移动光标到文档结束处 |
| DocumentEndExtend | Ctrl + Shift + End | 扩展选择到文档结束处 |
| DocumentStart | Ctrl + Home | 移动光标到文档开始处 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| DocumentStartExtend | Ctrl + Shift + Home | 扩展选择到文档开始处 |
| EditToggleOvertype | Ins | 插入与改写状态切换 |
| End(Brief) | | 移动光标到当前行末、窗口底部或文件尾 |
| Find | Ctrl + F | 查找给定的字符串 |
| FindBackwardDlg | | 打开 Find 对话框,向后搜索 |
| FindForwardDlg | | 打开 Find 对话框,向前搜索 |
| FindInFiles | | 在多个文件中搜索一字符串 |
| FindNext | F3 | 查找下一个匹配的字符串 |
| FindNextWord | Ctrl + F3 | 查找下一个匹配的字符串 |
| FindPrev | Shift + F3 | 查找上一个与选定文本匹配的字符串 |
| FindPrevWord | Ctrl + Shift + F3 | 查找上一个与选定文本匹配的字符串 |
| FindRegExpr | | 应用规则表达式查找下一个字符串 |
| FindRegExprPrev | | 应用规则表达式查找上一个字符串 |
| FindRepeat | | 重复上一次查找 |
| FindReplace | Ctrl + H | 用不同的文本替换特定的文本 |
| FindReplaceRegExpr | | 应用规则表达式,用不同的文本替换特定的文本 |
| FindToggleAllDocs | | 交叉搜索打开的文档 |
| FindToggleCaseSensitivity | | 切换区分/不区分大小写匹配 |
| FindToggleMatchWord | | 切换整词匹配/不需要整词匹配 |
| FindToggleRegExpr | | 切换规则表达式匹配有效/无效 |
| FindTool | Ctrl + D | 激活查找工具 |
| GoTo | Ctrl + G | 光标定位到指定位置 |
| GoToErrorTag | | 移动光标到包含当前错误或标签的行 |
| GoToFirstRow | | 移动到当前查询结果的第一行 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| GoToIndentation | | 移动到缩排结束处 |
| GoToLastRow | | 移动到当前查询结果的最后一行 |
| GoToMatchBrace | Ctrl + ] | 查找下一个匹配的括号 |
| GoToMatchBraceExtend | Ctrl + Shift + ] | 扩展选择到下一个匹配的括号 |
| GoToNewRow | | 在当前查询结果的最后添加一个新行 |
| GoToNextErrorTag | F4 | 移动光标到包含下一个错误或标签的行 |
| GoToNextRow | | 移动到查询结果的下一行 |
| GoToPrevErrorTag | Shift + F4 | 移动光标到包含上一个错误或标签的行 |
| GoToPreviousRow | | 移动到查询结果的上一行 |
| GoToRow | | 定位到查询结果所确定的行 |
| Home | Home | 移动光标到当前行的行首或该行文本的开始处 |
| Home(Brief) | | 移动光标到当前行的行首、文本窗口顶部或文件开始处 |
| HomeExtend | Shift + Home | 扩展选择到当前行的行首或该行文本的开始处 |
| IndentSelectionToPrev | | 采用和上一行相同的缩进格式缩排选定的文本 |
| IndentToPrev | | 缩进到与上一行的文本头对齐 |
| InsertColumn | | 向表中插入一列 |
| InsertQuoted | | 插入下一个定界符 |
| LevelCutToEnd | | 剪切光标与下一个闭弧之间的文本 |
| LevelCutToStart | | 剪切光标与上一个开弧之间的文本 |
| LevelDown | | 查找下一级的闭括号 |
| LevelUp | | 查找上一级的开括号 |
| LineCut | Ctrl + L | 删除选定的行,并送到剪贴板上 |
| LineDelete | Ctrl + Shift + L | 删除选定的行 |
| LineDeleteToEnd | | 删除到当前行末 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| LineDeleteToStart | | 删除到当前行首 |
| LineDown | Down Arrow | 光标下移一行 |
| LineDownExtend | Shift + Down Arrow | 向下扩展选择一行 |
| LineEnd | End | 移动到当前行末 |
| LineEndExtend | Shift + End | 扩展选择到当前行末 |
| LineIndent | | 选定文本向右缩进一个制表位 |
| LineOpenAbove | | 在当前光标所在行之前插入新行 |
| LineOpenBelow | | 在当前光标所在行之后插入新行 |
| LineStart | | 移动光标到当前行的行首 |
| LineTranspose | Alt + Shift + T | 交换当前行与前一行 |
| LineUnindent | | 选定文本向左缩进一个制表位 |
| LineUp | Up Arrow | 光标向上移动一行 |
| LineUpExtend | Shift + Up Arrow | 向上扩展选择一行 |
| ListMembers | Ctrl + Alt + T | 弹出一完整的列表框 |
| PageDown | Page Down | 光标移动到下一页 |
| PageDownExtend | Shift + Page Down | 向下扩展选择一页 |
| PageUp | Page Up | 光标移动到上一页 |
| PageUpExtend | Shift + Page Up | 向上扩展选择一页 |
| ParaDown | | 光标移动到下一段开始处 |
| ParaUp | | 光标移动到上一段开始处 |
| ParameterInfo | Ctrl + Shift + Space | 提供有关函数参数的帮助 |
| Paste | Ctrl + V | 将剪贴板内容插入到光标处 |
| Redo | Ctrl + Y | 重新执行上一次撤消的动作 |
| RedoChanges | | 重新执行最后的动作,忽略光标的移动 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| RemoveFromDiagram | | 从查询中删除表或链接行 |
| RemoveTableFromDiagram | | 删除选定的表 |
| SearchIncremental | Ctrl + I | 启动向前的增量搜索 |
| SearchIncrementalBack | Ctrl + Shift + I | 启动向后的增量搜索 |
| SelectAll | Ctrl + A | 选择全部文档 |
| SelectAllColumns | | 选择表中所有列 |
| SelectChar | F8 | 启动字符选择模式 |
| SelectCharInclusive | | 启动字符包含选择模式 |
| SelectColumn | Ctrl + Shift + F8 | 按列选择文本块 |
| SelectHighlight | | 光标与标记之间的文本高亮显示 |
| SelectLine | Ctrl + F8 | 选择多行文本 |
| SelectPara | | 选择当前段 |
| SelectSwapAnchor | | 交换选定内容中的光标和标记 |
| SelectionFormat | Alt + F8 | 将选定文本进行缩进格式化 |
| SelectionLowerCase | Ctrl + U | 将选定文本改为小写 |
| SelectionTabify | | 将选定文本的空格用制表符替换 |
| SelectionUntabify | | 将选定文本的制表符用空格替换 |
| SelectionUpperCase | Ctrl + Shift + U | 将选定文本改为大写 |
| SentenceCut | Alt + Shift + L | 删除语句的剩余部分 |
| SentenceLeft | | 光标移动到上一句的开始处 |
| SentenceRight | | 光标移动到下一句的开始处 |
| SetPrimaryKey | | 将选定列设置为主键 |
| SetRepeatCount | | 为下一个命令设置数值参数或重复次数 |
| SetRepeatCount0 | | 设置数值参数或重复次数为 0 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| SetRepeatCount1 | | 设置数值参数或重复次数为 1 |
| SetRepeatCount2 | | 设置数值参数或重复次数为 2 |
| SetRepeatCount3 | | 设置数值参数或重复次数为 3 |
| SetRepeatCount4 | | 设置数值参数或重复次数为 4 |
| SetRepeatCount5 | | 设置数值参数或重复次数为 5 |
| SetRepeatCount6 | | 设置数值参数或重复次数为 6 |
| SetRepeatCount7 | | 设置数值参数或重复次数为 7 |
| SetRepeatCount8 | | 设置数值参数或重复次数为 8 |
| SetRepeatCount9 | | 设置数值参数或重复次数为 9 |
| ToggleViewWhitespace | Ctrl + Shift + 8 | 显示或隐藏空格符 |
| TypeInfo | Ctrl + T | 提供在编辑器中选定的变量、函数或方法的语法格式 |
| Undo | Ctrl + Z | 撤消最后一次动作 |
| UndoChanges | | 撤消最后一次动作，忽略光标移动命令 |
| WBAddNewMember | | 添加成员函数 |
| WBCreate-NewVirtualFunction | | 添加虚拟函数 |
| WBCreateNewWindows MessageHandler | | 添加消息处理函数 |
| WBDefaultAction | | 执行 WizardBar 的当前缺省动作 |
| WBGoToClassDefinition | | 光标移动到选定的类定义处 |
| WBGoToMember | | 光标移动到选定的函数定义或实现处 |
| WBGoToMemberDeclaration | | 光标移动到选定的函数声明处 |
| WBGoToNext | | 光标移动到当前文件的下一个函数处 |
| WBGoToPrevious | | 光标移动到当前文件的上一个函数处 |
| WBHelp | | 弹出 WizardBar 的帮助信息 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| WBOpenInclude | | 打开选定的头文件 |
| WindowEnd | | 光标移动到文本窗口的底部 |
| WindowLeftEdge | | 光标移动到文本窗口的左边 |
| WindowRightEdge | | 光标移动到文本窗口的右边 |
| WindowScrollDown | Ctrl + Up Arrow | 文件内容向下滚动一行 |
| WindowScrollLeft | | 向左滚动窗口 |
| WindowScrollRight | | 向右滚动窗口 |
| WindowScrollToBottom | | 窗口滚动到底部 |
| WindowScrollToCenter | | 文本行滚动到窗口中央 |
| WindowScrollToTop | | 文本行滚动到窗口顶部 |
| WindowScrollUp | Ctrl + Down Arrow | 文件内容向上滚动一行 |
| WindowStart | | 光标移动到文本窗口的顶部 |
| WordCapitalize | | 首字母变为大写 |
| WordDeleteToEnd | Ctrl + Del | 向右删除一个词 |
| WordDeleteToStart | Ctrl + Backspace | 向左删除一个词 |
| WordLeft | Ctrl + Left Arrow | 向左移动一个词 |
| WordLeftExtend | Ctrl + Shift + Left Arrow | 向左扩展选择一个词 |
| WordLowerCase | | 整个词变为小写 |
| WordRight | Ctrl + Right Arrow | 向右移动一个词 |
| WordRightExtend | Ctrl + Shift + Right Arrow | 向右扩展选择一个词 |
| WordTranspose | Ctrl + Shift + T | 交换当前词与前一个词 |
| WordUpperCase | | 整个词变为大写 |

表 B-4 与文件操作有关的命令及快捷键

| 命令 | 快捷键 | 意义 |
|---|---|---|
| FileApplicationExit | | 退出应用程序,提示保存文档 |
| FileClose | | 关闭文档 |
| FileGoTo | Ctrl + Shift + G | 打开基于所选文本的文件 |
| FileOpen | Ctrl + O | 打开已存在的文档 |
| FilePrint | Ctrl + P | 打印全部或部分文档 |
| FilePrintPageSetup | | 打印页面设置 |
| FileSave | Ctrl + S | 保存文件 |
| FileSaveAll | | 保存所有打开的文件 |
| FileSaveAllExit | | 保存所有打开的文件并退出集成开发环境(Developer Studio) |
| FileSaveAs | | 用新文件名保存文档 |
| New | Ctrl + N | 建立新的文档、项目或工作区 |
| NewBinary | | 建立新的二进制文件 |
| NewBitmap | | 建立新位图 |
| NewResourceScript | | 建立新的资源描述文件 |
| NewText | | 建立新的文本文件 |
| ProjectSaveResources | | 保存项目的资源 |
| ResourceOpenBinary | | 打开选定的资源进行二进制编辑 |
| SaveChangeScript | | 提示 SQL 脚本程序改变,以便保存 |
| SaveSelection | | 保存对选定对象的修改到数据库中 |
| WorkspaceClose | | 关闭工作区 |
| WorkspaceOpen | | 打开工作区文件 |
| WorkspaceSave | | 保存工作区文件 |

表 B-5 与帮助有关的命令

| 命令 | 快捷键 | 意义 |
|---|---|---|
| HelpAbout | | 显示 Visual C++ 及版权信息 |
| HelpKeyboard | | 显示所有的键盘命令 |
| HelpMOWBestOfWeb | | 打开最好的 Web 网页 |
| HelpMOWDevOnly | | 打开 Developer 产品主网页 |
| HelpMOWFAQ | | 打开 Developer Studio 97 FAQ 网页 |
| HelpMOWFeedback | | 打开 Developer 产品反馈网页 |
| HelpMOWFreeStuff | | 打开 Developer Studio 97 免费资料网页 |
| HelpMOWMSHome | | 打开 Microsoft 主页 |
| HelpMOWOnlineSupport | | 打开 Microsoft 在线支持网页 |
| HelpMOWProductNews | | 打开 Developer Studio 97 产品信息网页 |
| HelpMOWSearchWeb | | 打开搜索网页 |
| HelpMOWWebTutorial | | 打开 Web 辅导网页 |
| HelpSystemContents | | 显示在线文档内容表 |
| HelpSystemIndex | | 显示在线文档索引 |
| HelpSystemSearch | | 搜索在线文档 |
| HelpTechnicalSupport | | 显示 Developer Studio 技术支持信息 |
| HelpTipOfTheDay | | 显示“每天一个提示” |
| HelpToggleExtensionHelp | | 打开/关闭使用扩展帮助(按 F1 时) |

表 B-6 与图形映像(Image)处理有关的命令及快捷键

| 命令 | 快捷键 | 意义 |
|---|---|---|
| Image3DRectangleTool | | 利用选定的线宽画 3D 矩形 |
| ImageAdjustColors | | 利用颜色对话框选择新的画图颜色 |
| ImageAirbrushTool | A | 按选定的尺寸用空笔刷绘图 |
| ImageBrushLarger | + | 笔刷尺寸的每一个方向增加一个像素 |

续表

| 命令 | 快捷键 | 意义 |
|---|---|---|
| ImageBrushOutlineTool | Shift + O | 用当前绘图颜色轮廓化笔刷或选定的对象 |
| ImageBrushPixel | . | 设置笔刷尺寸为 1 个像素 |
| ImageBrushSmaller | - | 笔刷尺寸的每一个方向减少一个像素 |
| ImageBrushTool | D | 按选定的形状和尺寸用笔刷绘图 |
| ImageColorNext | ] | 改变绘图颜色为调色板的下一种颜色 |
| ImageColorPrev | [ | 改变绘图颜色为调色板的上一种颜色 |
| ImageColorSelectTool | , | 采用图中颜色作为绘图时所使用的颜色 |
| ImageCurveTool | | 用选定的形状和尺寸绘制 Z 形曲线 |
| ImageDeleteDevice | | 删除选定设备的映像 |
| ImageEllipseTool | E | 用选定的线宽绘制椭圆 |
| ImageEraseColorNext | \| | 改变删除颜色为调色板的下一种颜色 |
| ImageEraseColorPrev | \| | 改变删除颜色为调色板的上一种颜色 |
| ImageEraseTool | Shift + P | 删除部分映像 |
| ImageFillTool | F | 用当前颜色填充一区域 |
| ImageFilledEllipseTool | Shift + E | 绘制填充椭圆 |
| ImageFilledRectangleTool | Shift + R | 绘制填充矩形 |
| ImageFilledRoundRectTool | Shift + N | 绘制填充圆角矩形 |
| ImageFlipHorizontal | X | 水平翻转映像或所选定的部分 |
| ImageFlipVertical | Y | 垂直翻转映像或所选定的部分 |
| ImageGridSettings | | 设置映像编辑时的参考网格线 |
| ImageInvertColors | | 反转映像或选定部分的颜色 |
| ImageLineTool | L | 用选定的形状和尺寸绘制直线 |
| ImageLoadPalette | | 装入以前保存在文件中的调色板颜色 |
| ImageMagnify | M | 改变当前视图为最高 |
| ImageMagnifyTool | Shift + M | 扩大当前视图 |
| ImageNewDevice | Ins | 为不同的显示设备建立映像 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| ImageOpenDevice | | 为不同的显示设备选择映像 |
| ImageOutlinedEllipseTool | Ctrl + Shift + E | 用轮廓绘制填充椭圆 |
| ImageOutlinedRectangleTool | Ctrl + Shift + R | 用轮廓绘制填充矩形 |
| ImageOut-linedRoundRectTool | Ctrl + Shift + N | 用轮廓绘制填充圆角矩形 |
| ImagePencilTool | P | 用单像素铅笔绘图 |
| ImagePickupBrush | Ctrl + B | 用所选图形创建定制笔刷 |
| ImageRectSelectTool | S | 选择映像的矩形区域,然后可以移动、复制和编辑 |
| ImageRectangleTool | R | 用选定的线宽绘制矩形 |
| ImageRegionSelectTool | | 选择映像的不规则区域,然后可以移动、复制和编辑 |
| ImageRotate90 | Z | 将映像或所选部分旋转 90 度 |
| ImageRoundRectTool | N | 用选定的线宽绘制圆角矩形 |
| ImageSavePalette | | 将调色板颜色保存到一个文件中 |
| ImageSetHotspotTool | | 改变光标的热点 |
| ImageTextTool | T | 绘制文本 |
| ImageToggleGrid | G | 打开/关闭网格线 |
| ImageToggleOpaque | O | 使当前选定区域透明或不透明 |
| ImageToggleTileGrid | Ctrl + G | 打开/关闭平铺网格线 |
| ImageToggleToolbarEditor | | 使工具栏编辑可以应用于当前位图 |
| ImageWindowRectangleTool | | 画 3D 窗口边界矩形 |
| ImageZoomIn | > | 扩大当前视图 |
| ImageZoomOut | < | 缩小当前视图 |

表 B-7 向项目添加新元素的命令及快捷键

| 命令 | 快捷键 | 意义 |
|---|---|---|
| EditPreviewHTMLPage | | 预览当前 HTML 页 |
| InsertAccelerator | | 建立新的加速键 |
| InsertAcceleratorTable | Ctrl + 7 | 建立新的加速键表资源 |
| InsertBitmap | Ctrl + 5 | 建立新的位图资源 |
| InsertCursor | Ctrl + 3 | 建立新的光标资源 |
| InsertDatabaseItem | | 向数据库中插入新项目 |
| InsertDialog | Ctrl + 1 | 建立新的对话框资源 |
| InsertFile | | 插入文件到文档中 |
| InsertIcon | Ctrl + 4 | 建立新的图标资源 |
| InsertMenu | Ctrl + 2 | 建立新的菜单资源 |
| InsertResource | Ctrl + R | 建立其他类型的新资源 |
| InsertResourceCopy | | 建立选定资源的备份 |
| InsertString | | 建立新串 |
| InsertStringTable | Ctrl + 8 | 建立或打开串表资源 |
| InsertToolbar | Ctrl + 6 | 建立新的工具栏资源 |
| InsertVersionInfo | Ctrl + 9 | 建立或打开版本信息资源 |
| NewATLComp | | 向项目中添加新 ATL 对象 |
| NewClass | | 建立新类并将其添加到项目中 |
| NewForm | | 建立新窗体并将其添加到项目中 |
| VerInfoStringBlockDelete | | 删除一版本信息块 |
| VerInfoStringBlockInsert | | 添加一版本信息块 |

表 B-8 对象分布(Layout)有关的命令及快捷键

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| AutoLayoutBoth | | 自动改变对话框尺寸,以便需要时收缩或伸展对话框宽度 |
| AutoLayoutGrow | | 自动改变对话框尺寸,以便需要时伸展对话框宽度 |
| AutoLayoutNoResize | | 自动改变高度,而其宽度不变 |
| CheckMnemonicKeys | Ctrl + M | 检测是否有重复定义的资源 |
| ControlHeightDecrease | Shift + Up Arrow | 向上移动选定控件或对话框的底边框一个单元,减少高度 |
| ControlHeightIncrease | Shift + Down Arrow | 向下移动选定控件或对话框的底边框一个单元,增加高度 |
| ControlMoveDown | Down Arrow | 向下移动选定控件一个对话框单元 |
| ControlMoveLeft | Left Arrow | 向左移动选定控件一个对话框单元 |
| ControlMoveRight | Right Arrow | 向右移动选定控件一个对话框单元 |
| ControlMoveUp | Up Arrow | 向上移动选定控件一个对话框单元 |
| ControlWidthDecrease | Shift + Left Arrow | 向左移动选定控件或对话框的右边框一个对话框单元,减少宽度 |
| ControlWidthIncrease | Shift + Right Arrow | 向右移动选定控件或对话框的右边框一个对话框单元,增加宽度 |
| DialogGridToggle | | 打开/关闭对话框网格线 |
| DialogRulerToggle | | 打开/关闭标尺及位置指示 |
| InsertActiveXControl-IntoDialog | | 向对话框中插入新的 ActiveX 控件 |
| LayoutAlignBottom | Ctrl + Down Arrow | 使选定的控件底端对齐,且都向最后选择的控件看齐 |
| LayoutAlignHorizontalCenter | Shift + F9 | 使选定的控件整体水平居中 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| LayoutAlignLeft | Ctrl + Left Arrow | 使选定的控件左端对齐，且都向最后选择的控件看齐 |
| LayoutAlignRight | Ctrl + Right Arrow | 使选定的控件右端对齐，且都向最后选择的控件看齐 |
| LayoutAlignTop | Ctrl + Up Arrow | 使选定的控件顶端对齐，且都向最后选择的控件看齐 |
| LayoutAlignVerticalCenter | F9 | 使选定的控件整体垂直居中 |
| LayoutArrangeButtonsBottom | Ctrl + Shift + B | 沿对话框底部中央放置选择的按钮 |
| LayoutArrangeButtonsRight | Ctrl + B | 将选择的按钮放置到对话框的右上角 |
| LayoutCenterInDialog Horizontally | Ctrl + Shift + F9 | 使控件在对话框内水平居中 |
| LayoutCenterInDialog Vertically | Ctrl + F9 | 使控件在对话框内垂直居中 |
| LayoutFlipHorizontally | | 水平翻转对话框控件 |
| LayoutGuides | | 改变网格线或辅助线的可视性和间隔 |
| LayoutMakeSameHeight | | 使所选定的控件与其中的主控件具有相同的高度 |
| LayoutMakeSameSize | | 使所选定的控件与其中主控件具有相同的宽度和高度 |
| LayoutMakeSameWidth | | 使所选定的控件与其中的主控件具有相同的宽度 |
| LayoutSizeToContent | Shift + F7 | 改变选定控件的尺寸以适应标题文本 |
| LayoutSpaceEvenlyAcross | Alt + Right Arrow | 水平方向平均分布选定的控件 |
| LayoutSpaceEvenlyDown | Alt + Up Arrow | 垂直方向平均分布选定的控件 |
| LayoutTabOrder | Ctrl + D | 设置对话框内控件的制表顺序 |
| ResourceTest | Ctrl + T | 运行对话框测试其外观和行为 |
| SelectDialogGuideType | Ctrl + G | 依次在无标尺和网格线、有标尺、有网格线间切换，以方便对话框的编辑 |

表 B-9　项目(Project)管理相关子命令

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| BuildProjectExport | | 以外部生成文件格式引出项目 |
| FileNewInProject | | 在项目工作区建立新文档 |
| InsertComponent | | 向项目中插入图库组件 |
| InsertDatasource | | 向活动项目添加数据链接 |
| InsertFilesIntoProject | | 将已存在的文件插入到项目中 |
| InsertNewFolder | | 向项目中插入新的文件夹 |
| ProjectDependencies | | 编辑项目的依赖关系 |
| ProjectInsertIntoWorkspace | | 向工作区插入已存在的项目 |
| ProjectSelectTool | Ctrl + Alt + P<br>Alt + F7 | 编辑项目创建和调试设置 |
| SourceControlAdd | | 放置项目文件到源代码控制中 |
| SourceControlCheckIn | | 登记文件到源代码控制中 |
| SourceControlCheckOut | | 从源代码控制中注销文件 |
| SourceControlExplorer | | 请求源代码控制系统服务 |
| SourceControlGetLatest Version | | 从源代码控制取得文件最新版本 |
| SourceControlProperties | | 显示文件的源代码控制属性 |
| SourceControlRefresh | | 更新源代码控制文件状态 |
| SourceControlRemove | | 从源代码控制删除项目文件 |
| SourceControlShare | | 与另一个源代码控制项目共享一个文件 |
| SourceCon-trolShowDifferences | | 显示一个文件自注销以来发生的改变 |
| SourceControlShowHistory | | 显示源代码控制下一个或多个文件的历史 |
| SourceCon-trolUndoCheckOut | | 撤消文件的注销,不保留已做的修改 |

表 B-10　与数据库查询相关的命令

| 命令 | 快捷键 | 意义 |
|---|---|---|
| AddToOutput | | 将选定的列添加到查询输出 |
| ChangeQueryDelete | | 创建一删除查询 |
| ChangeQueryInsert | | 创建一插入查询 |
| ChangeQueryInsertValues | | 创建一值插入查询 |
| ChangeQueryMakeTable | | 创建一生成表查询 |
| ChangeQuerySelect | | 创建一选择查询 |
| ChangeQueryUpdate | | 创建一更新查询 |
| ClearResults | | 清除本次查询结果 |
| GroupBy | | 为选定的列添加分组选项 |
| RemoveFilter | | 删除过滤器并显示所有行 |
| RowsFromTableA | | 选择 < TableA > 的所有行 |
| RowsFromTableB | | 选择 < TableB > 的所有行 |
| SortAscending | | 按升序排列数据 |
| SortDescending | | 按降序排列数据 |
| Synchronize SQL | | 同步 SQL 查询 |
| VerifySQLSyntax | | 验证查询语法而不执行该查询 |

表 B-11　使用各种工具的命令及快捷键

| 命令 | 快捷键 | 意义 |
|---|---|---|
| AutoJoinSelectedTables | | 自动链接选定的多个表 |
| Browse | Alt + F12 | 执行选定对象或当前上下文的查询(即浏览查询结果) |
| BrowseCloseBrowseDB | | 关闭浏览信息数据库 |
| Cancel | Esc | 隐藏窗口或取消模态化 |
| DataOpen | | 打开对象 |
| DebugProcedure | | 调试选定的存贮过程 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| FileStopFindInFiles | | 如果正在查找则停止 |
| HideActivePane | | 隐藏活动查询窗口 |
| InsertTrigger | | 向表中添加新的触发器 |
| Macro | | 建立或编辑开发环境(Developer Studio)宏 |
| MacroAdd | | 添加宏 |
| MacroPauseRecording | | 暂停或继续当前宏的录制 |
| MacroPlayQuick | Ctrl + Shift + P | 播放快速宏 |
| MacroRecordQuick | Ctrl + Shift + R | 启动录制临时宏 |
| MacroReloadFile | | 重新装入当前宏文件中的宏 |
| MacroStopRecording | | 结束当前宏的录制 |
| NewDatabaseDiagram | | 创建一新的数据库图表 |
| NewExtendedStored Procedure | | 创建新的数据库扩展存贮过程 |
| NewOracleFunction | | 创建新的数据库函数 |
| NewOraclePackageBody | | 创建新的数据库封装实体 |
| NewOraclePackageSpec | | 创建给定的数据库封装实体 |
| NewOracleProcedure | | 创建新的数据库过程 |
| NewQuery | | 在数据库项目中创建新的本地查询 |
| NewStoredProcedure | | 在数据库中创建新的存贮过程 |
| NewTable | | 为新数据库添加新表 |
| NewView | | 为新数据库添加新视图 |
| OpenExtendedStored Procedure | | 打开扩展存贮过程 |
| RunProcedure | | 运行选定对象 |
| StopProcedure | | 停止选定的存贮过程的执行 |
| TableOpen | | 打开表 |
| ToolsCustomize | | 定制工具菜单和工具栏 |
| ToolsCustomizeToolbars | | 定制工具栏 |
| ToolsOptions | | 修改选项 |

续表

| 命令 | 快捷键 | 意义 |
|---|---|---|
| UpdateRemoteOutputFile | | 更新远程项目输出文件 |
| UserTool1 | | 激活用户自定义工具 1 |
| UserTool10 | | 激活用户自定义工具 10 |
| UserTool11 | | 激活用户自定义工具 11 |
| UserTool12 | | 激活用户自定义工具 12 |
| UserTool13 | | 激活用户自定义工具 13 |
| UserTool14 | | 激活用户自定义工具 14 |
| UserTool15 | | 激活用户自定义工具 15 |
| UserTool16 | | 激活用户自定义工具 16 |
| UserTool2 | | 激活用户自定义工具 2 |
| UserTool3 | | 激活用户自定义工具 3 |
| UserTool4 | | 激活用户自定义工具 4 |
| UserTool5 | | 激活用户自定义工具 5 |
| UserTool6 | | 激活用户自定义工具 6 |
| UserTool7 | | 激活用户自定义工具 7 |
| UserTool8 | | 激活用户自定义工具 8 |
| UserTool9 | | 激活用户自定义工具 9 |

**表 B-12 与视图查看(View)相关的命令及快捷键**

| 命令 | 快捷键 | 意义 |
|---|---|---|
| ActivateCallStackWindow | Alt + 7 | 激活调用栈窗口 |
| ActivateDisassemblyWindow | Alt + 8 | 激活反汇编窗口 |
| ActivateMemoryWindow | Alt + 6 | 激活内存窗口 |
| ActivateOutputWindow | Alt + 2 | 激活输出窗口 |
| ActivateRegistersWindow | Alt + 5 | 激活寄存器窗口 |
| ActivateVariablesWindow | Alt + 4 | 激活变量窗口 |
| ActivateWatchWindow | Alt + 3 | 激活观察窗口 |
| ActivateWorkspaceWindow | Alt + 0 | 激活工作区窗口 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| BrowseBaseClasses | | 显示选定类的基类图 |
| BrowseCallGraph | | 显示选定函数的调用图 |
| BrowseCallersGraph | | 显示选定函数的调用者图 |
| BrowseDefAndRefs | | 显示符号定义和引用 |
| BrowseDerivedClasses | | 显示选定类的继承类图 |
| BrowseOutline | | 显示选定的或当前文件的大纲 |
| ClassWizard | Ctrl + W | 编辑应用程序类及其资源所依赖的代码,即类向导 |
| DataDesign | | 在设计视图中打开一个对象 |
| DiagramZoomControl | | 缩放数据库图表 |
| Permissions | | 显示存贮过程的许可权 |
| Properties | Alt + Enter | 编辑当前选定对象的属性 |
| Refresh | | 刷新选定对象 |
| ResourceSetIncludes | | 编辑资源符号文件名和预处理指令 |
| ResourceSymbols | | 浏览并编辑资源文件中的资源符号 |
| ShowPanesDiagram | | 显示/关闭查询图表窗口 |
| ShowPanesGrid | | 显示/关闭查询栅格窗口 |
| ShowPanesResults | | 显示/关闭查询结果窗口 |
| ShowPanesSQL | | 显示/关闭查询 SQL 窗口 |
| ShowTableAsColumn | | 显示选定表的列名 |
| ShowTableAsCustom | | 显示自定义表名 |
| ShowTableAsGrid | | 显示选定表的列属性 |
| ShowTableAsKey | | 显示选定表的关键字列 |
| ShowTableAsMinimized | | 仅显示选定表表名 |
| ToggleCallStackWindow | | 显示/隐藏调用栈窗口 |
| ToggleDisassemblyWindow | | 显示/隐藏反汇编窗口 |
| ToggleFullScreen | | 打开/关闭全屏幕模式 |

续表

| 命令 | 快捷键 | 意义 |
|---|---|---|
| ToggleMemoryWindow | | 显示/隐藏内存窗口 |
| ToggleOutputWindow | | 显示/隐藏输出窗口 |
| ToggleRegistersWindow | | 显示/隐藏寄存器窗口 |
| ToggleVariablesWindow | | 显示/隐藏变量窗口 |
| ToggleWatchWindow | | 显示/隐藏观察窗口 |
| ToggleWorkspaceWindow | | 显示/隐藏工作区窗口 |
| WBActionButtonActive | | 激活 WizardBar |
| WBClassComboActive | | 激活 WizardBar 的类 |
| WBFilterComboActive | | 激活 WizardBar 的过滤器 |
| WBMembersComboActive | | 激活 WizardBar 的成员 |

表 B-13 与窗口管理相关的命令及快捷键

| 命令 | 快捷键 | 意义 |
|---|---|---|
| WindowCascade | | 层叠窗口 |
| WindowClose | | 关闭窗口 |
| WindowCloseAll | | 关闭所有打开的窗口 |
| WindowCycle | | 切换到下一个窗口 |
| WindowDeleteColumnRight | | 删除活动面板上分隔线右边的列 |
| WindowDeleteColumnLeft | | 删除活动面板上分隔线左边的列 |
| WindowDeleteRowDown | | 删除活动面板上分隔线以下的行 |
| WindowDeleteRowUp | | 删除活动面板上分隔线以上的行 |
| WindowDockingView | Alt + F6 | 启用/禁止窗口的船坞化特征 |
| WindowHide | Shift + Esc | 隐藏窗口 |
| WindowKillPane | | 删除当前分隔窗口 |
| WindowList | | 管理当前打开的窗口 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| WindowMaximize | | 最大化当前窗口 |
| WindowNewWindow | | 为文档再打开一个窗口 |
| WindowNext | | 激活下一个窗口 |
| WindowNextMDI | | 激活下一个未船坞化的窗口 |
| WindowNextPane | F6 | 激活下一个面板 |
| WindowPrevPane | Shift + F6 | 激活上一个面板 |
| WindowPrevious | | 激活上一个窗口 |
| WindowPreviousMDI | | 激活上一个未船坞化的窗口 |
| WindowSinglePane | | 删除所有分隔窗口,仅留单一窗口 |
| WindowSplit | | 把窗口分割为多个面板 |
| WindowSplitHorizontal | | 水平分割窗口 |
| WindowSplitVertical | | 垂直分割窗口 |
| WindowSwitchPaneDown | | 激活分隔窗口的下部 |
| WindowSwitchPaneLeft | | 激活分隔窗口的左部 |
| WindowSwitchPaneRight | | 激活分隔窗口的右部 |
| WindowSwitchPaneUp | | 激活分隔窗口的上部 |
| WindowTileHorizontally | | 水平排列各窗口 |
| WindowTileVertically | | 垂直排列各窗口 |

**表 B-14 与图表操作相关的命令**

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| AddRelatedTables | | 向图表中添加关联表 |
| Arrange Tables | | 重新排列表 |
| ArrangeSelection | | 自动布局选中的表 |
| AutosizeSelectedTables | | 自动调整所选择表的大小 |

续表

| 命令 | 快捷键 | 意义 |
| --- | --- | --- |
| ModifyCustomView | | 修改客户视图 |
| NewTextAnnotation | | 插入新文本注解 |
| RecalculatePageBreaks | | 重新计算图表页 |
| ShowRelationshipLabels | | 显示关联表 |
| TextAnnotationFont | | 改变文本注解字体 |
| ViewPageBreaks | | 显示图表页 |

# 无忧书库书籍版权申明

无忧书库提供的所有书籍、资料版权归原作者和出版商所有。

本站提供下载的书籍仅供个人学习、参考之用，任何集体、个人不得用于商业用途。

请您在下载书籍 **24** 小时之后删除书籍，购买正版书籍！

本站书籍如有侵犯您的版权，请与我们联系，我们将尽快删除。联系信箱：**pcbook@51soft.com**。

无忧书库
**http://pcbook.51soft.com**
**2000** 年 **4** 月 **13** 日