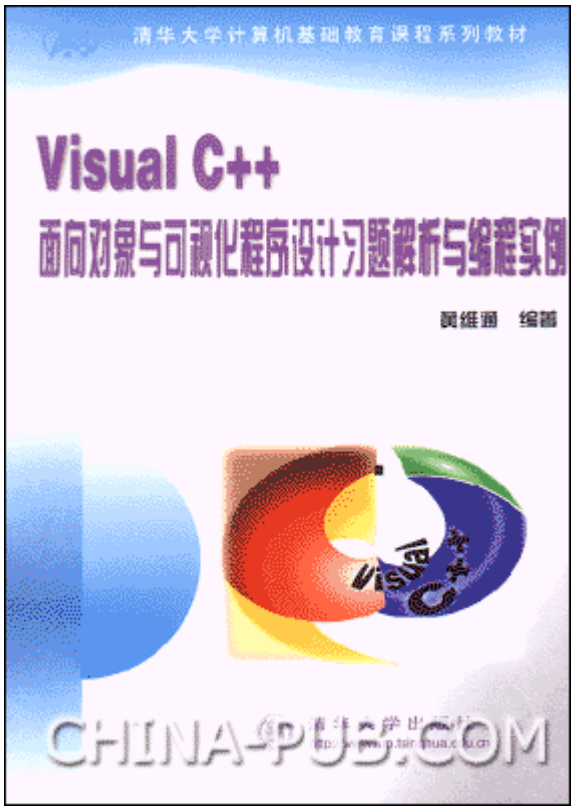


本书是“清华大学计算机基础教育课程系列教材”中的《Visual C++面向对象与可视化程序设计》一书的配套教材。书中全面具体地对该教材中各章的习题做了详细解答，对每一道题都进行了必要的分析和代码注释，力求通过实例让读者全面掌握面向对象与可视化程序设计的思路和开发技巧。同时，为让读者更好地掌握“Visual C++面向对象与可视化程序设计”的思想和方法，还增加了部分紧扣相关知识点的典型实例。书后附有使用 API 工具、MFC 工具和 Deb 吧工具开发、调试应用程序的基本步骤。

本书涉及的知识点包括 VisualC++的基础知识及开发环境、Windows 编程基础知识、应用 AH 函数进行可视化编程的基本方法和应用 MFC 进行可视化编程的思想方法。

本书通俗易懂，重点突出，偏重应用。不仅适合作为大专院校理工科学生学月 VisualC++面向对象编程的辅助教材，还适合于初步掌握 VisualC++的编程人员作为习题教材，同时也可供有关科研及开发人员参考。

目录

无忧书库
http://pcbook.51soft.com

第1篇 Visual C++基础知识

第1章 C++基础知识 …… 3
1.1 C++的发展历程 …… 3
1.2 一个简单的C++程序 …… 3
1.3 数据类型 …… 4
1.3.1 C++的基本数据类型 …… 4
1.3.2 复合数据类型 …… 4
1.4 常量及其定义 …… 8
1.5 变量 …… 9
1.6 指针 …… 10
1.6.1 指针的类型及其定义 …… 10
1.6.2 指针的初始化 …… 10
1.6.3 字符串指针 …… 11
1.6.4 引用类型 …… 11
1.7 C++中的控制结构 …… 11
1.7.1 条件语句 …… 12
1.7.2 循环语句 …… 13
1.7.3 转移语句 …… 14
1.8 函数 …… 15
1.9 类与对象 …… 16
1.9.1 类的定义 …… 16
1.9.2 对象 …… 17
1.9.3 内联函数 …… 19
1.10 构造函数和析构函数 …… 20
1.10.1 构造函数 …… 20
1.10.2 析构函数 …… 22
1.11 方法重载 …… 23
1.11.1 函数重载 …… 23
1.11.2 操作符重载 …… 25

1.12 友元 …… 27
1.13 this 指针 …… 28
1.14 继承 …… 30
1.14.1 派生类 …… 30
1.14.2 多重继承 …… 31
1.15 多态性和虚拟函数 …… 32
1.15.1 多态性 …… 32
1.15.2 虚拟函数 …… 33
1.15.3 虚拟析构函数 …… 36
1.16 流 …… 37
习题 …… 37

第 2 章 Visual C++6.0 开发环境 …… 38
2.1 Visual C++ 及其开发环境概述 …… 38
2.2 Visual C++6.0 的菜单栏 …… 39
2.2.1 File 菜单 …… 39
2.2.2 Edit 菜单 …… 41
2.2.3 View 菜单 …… 42
2.2.4 Insert 菜单 …… 46
2.2.5 Project 菜单 …… 47
2.2.6 Build 菜单 …… 48
2.2.7 Tools 菜单 …… 50
2.3 Visual C++ 6.0 的工具栏 …… 54
2.3.1 工具栏的构成 …… 54
2.3.2 工具栏的定制与修改 …… 55
2.4 项目与项目工作区 …… 55
2.4.1 创建新的项目工作区 …… 55
2.4.2 项目工作区窗口 …… 56
2.5 资源与资源编辑器 …… 58
2.6 联机帮助 …… 59
习题 …… 60

第 3 章 Windows 编程基础 …… 61
3.1 Windows 编程基础知识 …… 61
3.1.1 窗口 …… 62
3.1.2 事件驱动 …… 63

3.1.3　句柄和 Windows 消息 …… 63
3.2　Windows 应用程序常用消息 …… 65
3.3　Windows 中的事件驱动程序设计 …… 66
3.4　Windows 应用程序的基本结构 …… 68
3.4.1　Windows 应用程序的组成 …… 68
3.4.2　源程序组成结构 …… 68
3.4.3　应用程序举例 …… 74
习题 …… 77

第 2 篇　使用 API 函数进行可视化编程

第 4 章　Windows 的图形设备接口及 Windows 绘图 …… 81
4.1　图形设备接口 …… 81
4.1.1　图形设备接口的一些基本概念 …… 81
4.1.2　图形刷新 …… 83
4.1.3　获取设备环境 …… 85
4.1.4　映像模式 …… 85
4.2　绘图工具和颜色的设置与应用 …… 88
4.2.1　画笔的应用 …… 88
4.2.2　画刷的应用 …… 89
4.2.3　颜色的设置 …… 90
4.3　常用绘图函数 …… 90
4.4　应用实例 …… 93
习题 …… 103

第 5 章　文本的输出方法与字体的设置 …… 104
5.1　设置文本的设备环境 …… 104
5.1.1　字体句柄 …… 104
5.1.2　创建自定义字体 …… 105
5.1.3　设置字体和背景颜色 …… 105
5.2　文本的输出过程 …… 106
5.3　文本输出与字体设置编程实例 …… 108
习题 …… 116

第 6 章　Windows 应用程序对键盘与鼠标的响应 …… 117
6.1　应用程序对键盘动作的响应 …… 117

6.2 键盘响应编程实例 …… 120
6.3 应用程序对鼠标动作的响应 …… 127
6.4 鼠标响应编程实例 …… 130
习题 …… 134

第7章 资源在Windows编程中的应用 …… 135
7.1 菜单资源、加速键资源及其应用 …… 135
7.1.1 菜单资源的创建过程 …… 135
7.1.2 菜单项的操作 …… 139
7.1.3 菜单的动态创建 …… 141
7.1.4 加速键资源的创建过程 …… 142
7.1.5 菜单与加速键应用实例 …… 144
7.2 位图资源及其应用 …… 151
7.2.1 位图的概念 …… 151
7.2.2 位图资源的创建过程 …… 151
7.2.3 位图资源应用实例 …… 153
7.3 对话框资源及其应用 …… 156
7.3.1 模态对话框的编程方法 …… 157
7.3.2 模态对话框应用实例 …… 160
7.3.3 非模态对话框的编程方法 …… 166
7.3.4 非模态对话框应用实例 …… 167
7.4 通用对话框资源及其应用 …… 173
7.4.1 通用对话框的创建过程 …… 173
7.4.2 通用对话框应用实例 …… 175
7.5 图标资源及其应用 …… 180
7.5.1 图标资源的操作 …… 180
7.5.2 图标资源应用实例 …… 181
习题 …… 183

第8章 Windows标准控件在可视化编程中的应用 …… 185
8.1 控件概述 …… 185
8.2 常用子窗口型控件操作函数 …… 187
8.3 按钮控件 …… 188
8.3.1 按钮控件的分类及其特点 …… 188
8.3.2 按钮控件的创建过程 …… 189
8.3.3 按钮控件与应用程序之间的消息传递 …… 191

8.3.4　按钮控件应用实例 …… 191
8.4　滚动条控件 …… 197
8.4.1　滚动条控件的特点与分类 …… 197
8.4.2　滚动条控件的创建 …… 198
8.4.3　常用滚动条操作函数 …… 199
8.4.4　滚动条控件应用实例 …… 201
8.5　静态控件 …… 210
8.5.1　静态控件的特点 …… 210
8.5.2　静态控件的创建 …… 210
8.5.3　静态控件消息的发送 …… 211
8.5.4　静态控件应用实例 …… 212
8.6　列表框控件 …… 215
8.6.1　列表框控件的特点 …… 215
8.6.2　列表框控件的创建 …… 215
8.6.3　列表框控件与应用程序之间的消息传递 …… 216
8.6.4　列表框控件应用实例 …… 218
8.7　编辑框控件 …… 221
8.7.1　编辑框控件的特点 …… 221
8.7.2　编辑框控件的创建 …… 222
8.7.3　编辑框控件与应用程序之间的消息传递 …… 222
8.7.4　编辑框控件应用实例 …… 223
8.8　组合框控件 …… 228
8.8.1　组合框控件的特点 …… 228
8.8.2　组合框控件的创建 …… 229
8.8.3　组合框控件与应用程序之间的消息传递 …… 229
8.8.4　组合框控件应用实例 …… 230
习题 …… 233

第9章　文件及剪贴板操作 …… 234
9.1　文件操作 …… 234
9.1.1　Windows 98 文件操作的特点 …… 234
9.1.2　常用的文件操作函数 …… 234
9.1.3　文件操作应用实例 …… 238
9.2　文件中剪贴板的操作 …… 247
9.2.1　向剪贴板发送文本 …… 248
9.2.2　获取剪贴板中的文本 …… 250

9.2.3 剪贴板应用实例 …… 251
习题 …… 257

第3篇 使用MFC进行可视化编程

第10章 MFC基础知识 …… 261
10.1 MFC概述 …… 261
10.2 MFC中类的组织结构及主要的类的简介 …… 263
10.2.1 MFC中类的组织结构 …… 263
10.2.2 根类 …… 264
10.2.3 应用程序体系结构类 …… 265
10.2.4 可视对象类 …… 265
10.2.5 通用类 …… 267
10.2.6 OLE类 …… 268
10.2.7 ODBC数据库类 …… 268
10.3 MFC中的全局函数与全局变量 …… 269
10.4 应用程序向导 …… 269
习题 …… 274

第11章 编辑框类及其应用 …… 275
11.1 编辑框类的结构及其方法 …… 275
11.2 编辑框类编程实例 …… 278
习题 …… 288

第12章 菜单类及其应用 …… 289
12.1 菜单类简介 …… 289
12.1.1 菜单 …… 289
12.1.2 菜单资源模板 …… 289
12.1.3 菜单类的结构及其方法 …… 290
12.2 菜单类编程实例 …… 293
习题 …… 300

第13章 滚动条类及其应用 …… 301
13.1 滚动条类简介 …… 301
13.1.1 滚动条类的结构及其方法 …… 301
13.1.2 创建与初始化滚动条类 …… 302

13.2　滚动条类编程实例 …… 303
习题 …… 310

第 14 章　按钮类与列表框类及其应用 …… 312
14.1　按钮类与列表框类简介 …… 312
14.1.1　按钮类的结构及其方法 …… 312
14.1.2　列表框类的结构及其方法 …… 315
14.2　按钮类及列表框类编程实例 …… 321
习题 …… 336

第 15 章　工具条类及其应用 …… 338
15.1　工具条类简介 …… 338
15.1.1　工具条 …… 338
15.1.2　工具条资源模板 …… 338
15.1.3　工具条类的结构及其方法 …… 339
15.2　工具条类编程实例 …… 345
习题 …… 350

第 16 章　文档-视图结构及其应用 …… 351
16.1　概述 …… 351
16.1.1　单文档界面与多文档界面 …… 351
16.1.2　文档-视图结构 …… 352
16.2　文档类的结构及其方法 …… 352
16.3　构造文档类的派生类 …… 356
16.4　视图类的结构及其方法 …… 357
16.5　视图类的派生类 …… 360
16.6　文档模板类的结构及其方法 …… 362
16.7　SDI 与 MDI 编程实例 …… 367
习题 …… 374

参考文献 …… 376

第 1 篇

Visual C++ 基础知识

本篇介绍 C++ 及 Visual C++ 的基本概念和基本知识，力求使读者能对面向对象与可视化编程有个基本的了解，并为下一步的学习打下必要的基础。

本篇内容包括：

第 1 章　C++ 基础知识

第 2 章　Visual C++ 6.0 开发环境

第 3 章　Windows 编程基础

第 1 章

C++ 基础知识

1.1 C++的发展历程

C++是既适合于作为系统描述语言，也适合于编写应用软件的既面向对象又面向过程的一种混合型程序设计语言，它是在C语言的基础之上发展起来的。

在C语言推出之前，操作系统等系统软件主要是用汇编语言编写的(如著名的UNIX操作系统)。由于汇编语言依赖于计算机硬件，因此程序的可移植性和可读性就比较差。为了提高程序的可读性和可移植性，最好能采用高级语言来编写这些系统软件。然而，一般的高级语言难以实现汇编语言的某些功能(如汇编语言可以直接对硬件进行操作、对内存地址进行操作和位操作等)。人们设想有一种能集一般高级语言和低级语言特性于一身的语言。于是，C语言便应运而生了。

最初的C语言只是为描述和实现UNIX操作系统而提供的一种程序设计语言。1973年，贝尔实验室的K.Thompson和D.M.Ritchie两人合作把UNIX的90%以上的代码用C语言改写(即UNIX第五版)。后来C语言又作了多次改进，1978年以后，C语言已先后移植到大、中、小及微型机上，现在C语言已成为风靡全球的计算机程序设计语言。

到了80年代，美国AT&T贝尔实验室的Bjarne Stroustrup在C语言的基础上推出了C++程序设计语言。由于C++提出了把数据和在数据之上的操作封装在一起的类、对象和方法的机制，并通过派生、继承、重载和多态性等特征，实现了人们期待已久的软件复用和程序自动生成。这使得软件，特别是大型复杂软件的构造和维护变得更加有效和容易，并使软件开发能更自然地反映事物的本质，从而大大提高了软件的开发效率和质量。

C++越来越受到重视并得到广泛的应用，许多软件公司都为C++设计编译系统，如AT&T，Apple，Sun，Borland和Microsoft等，其中国内最为流行的应当是Borland公司的Borland C++和Microsoft公司的Visual C++。与此同时，许多大学和公司也在为C++编写各种不同的类库，其中Borland公司的OWL(Object Windows Library)和Microsoft公司的MFC(Microsoft Foundation Class)就是比较优秀的代表，尤其是Microsoft的MFC，在国内外得到了较为广泛的应用。

1.2 一个简单的C++程序

下面是一个用C++编写的例子，其功能是在屏幕上显示“Welcome !”，其程序代码

如下：

```
#include <iostream.h>                    //包含头文件
void main()                              //main 函数,程序入口
{                                        //程序体开始
    char str_greet[] = "Welcome!";       //定义一个数组并初始化
    cout << str_greet << endl;           //在屏幕上输出字符串内容并换行
}                                        //程序体结束
```

熟悉 C 语言的读者不难看出，用 C++ 编写的程序和用 C 编写的程序在程序结构上基本是相同的，都是以 main 函数作为程序的入口，两者都是以一对{}把函数中的语句括起来，而且两者都是以分号作为语句的结束标志。但是，两者也有一些不同之处，C++ 中是以 iostream.h 文件作为标准输入输出头文件，C 中是以 stdio.h 作为标准输入输出头文件；C++ 中采用符号“<<”作为标准输出，而不是通过 printf 函数来实现。

通过上面的例子，可以看出，C++ 语言和 C 语言两者之间既有紧密的联系，又各有自己的特点。下面的内容将介绍 C++ 程序设计中的一些基础知识，这部分内容，C++ 和 C 有很多是一致的。由于本书是面向已经熟悉 C 语言并初步掌握 C++ 语言的读者，因此，对 C++ 的内容只是做一个简单的总结性概述，如果读者对 C 及 C++ 语言很熟悉的话，可以跳过这部分内容的学习。

1.3 数据类型

在 C++ 中，任何数据在使用之前都要进行数据类型的定义，然后才能使用。

1.3.1 C++ 的基本数据类型

基本数据类型是语言预定义的抽象。在 C++ 中，每种基本数据类型都使用一个关键字来表示。C++ 的基本数据类型描述了机器硬件所支持的对象和可以对这些对象执行的操作。C++ 的基本数据类型分为三大类，即整型、浮点型和无值型(void)。表 1-1 是 C++ 所提供的基本数据类型及其值的范围。

在基于 Windows(包括 Windows 95/98, Windows NT)的程序设计中，Windows 定义并使用了许多非单一的数据类型，这些数据类型既简单又复杂，同一数据类型常常有一个以上的名字，这样设计的目的是为了有助于程序员编写更易读的代码，并通过不使用特定硬件的数据类型以促进跨平台、跨处理机的目的。例如，为了促进可移植性，在 Windows 中常使用数据类型的别名来代替具体的数据类型，如使用 UINT 代替使用无符号整数 unsigned int 等。

1.3.2 复合数据类型

C++ 中还提供了几种复合数据类型，这些复合数据类型包括数组、结构体、共用体、枚举、类和用户自定义类型。

表 1-1　C++所提供的各种基本数据类型及其值的范围

类　型	说　明	二进制位	值　域
char	字符型	8	-128 ~ 127
signed char	有符号字符型	8	-128 ~ 127
unsigned char	无符号字符型	8	0 ~ 255
int	整型	16	-32768 ~ 32767
signed int	有符号整型	16	-32768 ~ 32767
unsigned int	无符号整型	16	0 ~ 65535
short int	短整型	16	-32768 ~ 32767
signed short int	有符号短整型	16	-32768 ~ 32767
unsigned short int	无符号短整型	16	0 ~ 65535
long int	长整型	32	$-2^{31} \sim (2^{31}-1)$
signed long int	有符号长整型	32	$-2^{31} \sim (2^{31}-1)$
unsigned long int	无符号长整型	32	$0 \sim (2^{32}-1)$
float	浮点型	32	7 位有效位
double	双精度型	64	15 位有效位
long double	长双精度型	80	19 位有效位

1. 数组

数组是具有统一数据类型的对象的集合。每一个单独的对象并没有名字，但是它们可以通过其在数据中的位置来确定。例如，下面的语句定义了一个名为 Array 的数组，该数组共有 10 个元素。程序中通过数据元素在数组中的索引值来访问数组元素。

```
int i, Array[10];  //定义数组,值得注意的是,在数组中元素的下标是从 0 开始
Array[2] = 30;     //为数组元素赋值
i = Array[2];      //读取数组元素中的值,把它赋给变量 i
```

在 C++ 中不仅可以定义一维数组，还可以定义多维数组。多维数组中的每一维的范围大小都用一个中括号括起来。例如，下面的语句定义了一个二维数组，该数组的第一维共有 10 个元素，第二维共有 20 个元素。

```
int Array[10][20];
```

在一个二维数组中，第一个下标称为行下标，第二个下标称为列下标，上面定义的数组 Array 相当于是一个具有 10 行 20 列的矩阵。

数组在定义时，系统将自动为它分配一块连续的内存空间。此时，数组名指向这块空间的起始点，并且在程序中不可改变。也就是说，一个数组名相当于一个常量。

2. 结构体

在程序设计中，有时需要将某些相关的具有不同类型的数据组织到一个数据类型中。为此，C++ 中提供了这样一种复合数据类型，称之为结构体。定义一个结构体类型的一般形式为：

```
struct 结构体名
{
  成员表列;
};
```

例如，下面的语句中定义了一个有关学生的名为 student 的结构体，它包括学生的学号、姓名、性别和年龄等不同数据类型的信息。

```
struct student
{
    int StuNo;          //学生学号
    char StuName[20];   //学生姓名
    char Sex;           //学生性别
    int Age;            //学生年龄
};
```

上面定义了一个结构体类型，struct student(其中 struct 是关键字，不能省略)表示这是一个"结构体类型"。它包括 StuNo，StuName，Sex 和 Age 等不同数据类型的数据项，这些数据项均称为结构体的成员。应当注意：struct student 是程序设计者自己定义的类型名。它和系统已定义的标准类型(如 int，char，float 和 double 等)一样可以用来作为定义变量的类型。

定义了结构体类型后，还必须定义该结构体类型的变量，才能使用该结构体。定义结构体变量的方法有以下三种：

(1) 先定义结构体类型再定义变量名

例如，上面已定义一个结构体类型 struct student，可以用它来定义结构体 student 的两个变量 stu1，stu2。

```
struct student stu1,stu2;
```

变量 stu1，stu2 为 struct student 类型变量，即它们具有 struct student 类型的结构。

(2) 在定义类型的同时定义变量

例如，下面的语句在定义结构体类型 student 的同时定义了 stu1，stu2 两个变量。

```
struct student
{
    int StuNo;
    char StuName[20];
    char Sex;
    int Age;
}stu1,stu2;
```

(3) 直接定义结构体变量

例如，下面的语句中直接定义了两个结构体变量 stu1 和 stu2，但没有出现结构体类型名。

```
struct                          //此处省略了结构体名
{
    int StuNo;
    char StuName[20];
    char Sex;
    int Age;
}stu1,stu2;
```

值得指出的是，结构体成员可以嵌套，即结构体的成员可以是另一个结构体。

3. 共用体

在程序设计中，有时需要使几种不同类型的变量存放在同一内存单元中。这种使几种不同的数据类型的变量共占同一段内存的结构，称为“共用体”（有时也称“联合体”）类型的结构。“共用体”类型变量的定义形式为：

```
union 共用体名
{
    成员表列;
}变量表列;
```

例如，下面的语句中定义一个名为 data 的共用体，并定义了 i 和 ch 两个 data 共用体的成员及 u1 和 u2 两个共用体变量。

```
union data
{
    int i;
    char ch;
}u1,u2;
```

可以看出，共用体和结构体的定义相似。但它们的含义是不同的，二者的主要区别在于：结构体变量所占内存长度是各成员所占内存长度之和，每个成员分别占有其自己的内存单元；而共用体变量所占的内存长度等于最长的成员的长度。

4. 枚举

如果一个变量只有几种可能的值，可以定义为枚举类型。所谓枚举是指将变量的值一一列举出来，变量的值只限于在列举出来的值的范围内。

定义枚举类型由关键字 enum 开始。例如，下面的语句定义一个名为 color 的枚举类型，并定义了一个枚举类型的变量 My-color。

```
enum color {Red,Green,Blue,Yellow};
enum color  My-color;
```

定义的枚举变量 My-color 是枚举变量，它的值只能是 Red 到 Yellow 之一。例如，下面的语句给枚举变量 My-color 赋值。

```
My-color = Red;
```

5. 用户自定义数据类型

在 C++ 中可以用 typedef 定义新的类型名来代替已有的类型名。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。例如，下面的语句中用 INTEGER 来代表 int 类型：

```
typedef int INTEGER;
```

习惯上常把 typedef 定义的类型名用大写字母表示，以便与系统提供的标准类型标识符相区别。

基于 Windows 的程序还使用很多在 Windows 头文件中定义的结构化的数据类型，表 1-2 中列出了在 Windows 程序设计中使用较频繁的结构化的数据类型。

表 1-2 在 Windows 头文件中定义的结构化的 Win32 数据类型

数据类型	意 义
BITMAP	独立于逻辑设备的位图
LOGBRUSH	逻辑刷
LOGFONT	逻辑字体
LOGPEN	逻辑笔
MSG	窗口消息
POINT	点
RECT	矩形
WNDCLASS	窗口类结构

1.4 常量及其定义

常量是指用以表达有固定数值或字符值的单词符号，在程序中不允许修改。

通过使用关键字 const，将能使一个变量定义为一个常量。例如，下面的语句中定义了一个表示缓冲区大小的常量。

```
const int bufSize = 1024;
```

上面这一语句将使 bufSize 被定义为一个符号常量并初始化为 1024，如果在程序中试图改变 bufSize 的值，则将引起一个错误。由于一个 const 类型的量一经定义后就不能改变它的值，因此在定义时必须同时初始化。例如，下面的语句中定义了一个 double 类型的常量，但是没有在定义的同时初始化，这样就会产生错误。

```
const double PI;  //这条语句将产生错误
```

在 C++ 中，试图把一个符号常量的地址赋给一个指针同样会使编译器产生编译错误。为解决这一问题，程序员可以定义一个指向常量的指针。例如：

```
const double *pc;
```

pc 是一个指向 const double 型的对象的指针，但 pc 本身并不是一个常量。也就是说，pc 可以指向任何一个 double 型的对象，但由 pc 所指向的对象不可以通过 pc 来改变其值。

例如，下面的语句中定义了一个指向常量的指针，通过该指针可以访问不同的常量，但是不能改变各个常量的值。

```
pc = &mbBalance;     //正确
double d;
pc = &d;             //正确
d = 3.14;            //正确
*pc = 3.14;          //正确
```

常量还可以分为整型常量、浮点型常量和字符型常量等。例如，对于整数 20 来说，可以用下面三种形式表示：

```
20    //十进制表示
024   //八进制表示法
0x14  //十六进制表示法
```

一个浮点常量可以用科学计数法或通常的十进制表示法来表示。在使用科学计数法时，指数部分以字母 E 或 e 开始。一个浮点常量可用字母 F 或 f 标识为 float 型。如：

```
3.1415926F  0.1f    0.0  3e1  1.0e-3    2.
```

在 C++ 中，可被输出的字符常量用单引号括起来。例如：

```
'A'  '2'  ' '(空格符)
```

若想使用不可打印输出的字符、单引号、双引号和反斜线等字符时，可以使用转义字符序列表示法。C++ 中的转义字符如表 1-3 所示。

表 1-3　C++ 中的转义字符

字符形式	功　能
\n	换行
\t	横向跳格(即跳到下一个输出区)
\v	竖向跳格
\b	退格
\r	回车
\f	走纸换页
\a	响铃
\\	反斜杠字符
\'	单引号字符
\"	双引号字符
\?	问号
\ddd	1～3 位八进制数所代表的字符
\xhh	1～2 位十六进制数所代表的字符

一个字符串常量是一个字符数组。组成它的元素除了给出的字符序列外，还包括一个在结尾处的 null 字符。这个字符由编译器自动添加，用于标识一个字符串的终结。

1.5　变量

在程序执行过程中，值可变的量称为变量。每一个变量都有一个由程序员给出的变量名，以此来识别该变量；同时每一个变量又具有一个特定的数据类型。

在定义变量时，应给出该变量的数据类型和名字，还可以给这个变量赋一个初始值(不是必须这样做)，系统在编译时就会分配给该变量一个存储空间。

一个变量名是由标识符表示的，可以由字母、数字和下划线组成，但必须以一个字母或下划线开始。在组成变量名的元素中，大小写字母含义是不同的，也就是说，在 C++ 中是区分大小写字母的。

C++ 中有一套只供语言专用的单词集，称为保留字集。如预先定义好的类型标识符都是保留字。作为保留字的单词，不能再作为其他名字使用(如不能作为变量名、函数名等)。C++ 中的保留字如表 1-4 所示。

表 1-4 C++中的保留字

auto	const	enum	if	protected	sizeof	true	volatile
bool	continue	explicit	inline	public	static-cast	typedef	while
break	default	extern	int	try	static	typeid	
case	delete	false	long	reinterpret-cast	struct	typename	
catch	do	float	mutable	register	switch	union	
char	double	for	new	return	template	unsigned	
class	dynamic-cast	friend	operator	short	this	virtual	
const-cast	else	goto	private	signed	throw	void	

为了使不同的程序员设计的程序能够很好地互相交流，在基于 Windows 的 VC 程序中广泛采用匈牙利表示法来标识变量。匈牙利表示法用小写字母作为变量名的前缀，以说明变量的数据类型，其余部分描述该变量的意义与功能。

表 1-5 列举了 Windows 程序中常用的匈牙利表示法的前缀、类型及其说明。

表 1-5 常用匈牙利表示法前缀的类型及其说明

前 缀	类 型	说 明
ch	char	字符
b	BOOL	布尔值
n(I)	int	整数
w	WORD	16 位无符号数
l	LONG	32 位有符号数
dw	DWORD	32 位无符号数
lpsz	LPTSTR	指向字符串的 32 位指针
h	Handle	窗口对象句柄

1.6 指针

一个指针变量所存储的信息是一个对象在内存中的地址。通过指针，可以间接地访问对象。

1.6.1 指针的类型及其定义

每一个指针都有一个相应的类型。该类型用以说明指针所指地址中存放的数据类型。

一个指针所占的内存空间大小与一个内存地址所占空间相等。也就是说，一个 int 指针与一个 double 型指针占用同样大小的内存空间。在定义指针时，应在指针变量名前冠以星号“*”。

1.6.2 指针的初始化

一个指针可以被一个具有相同类型的对象的左值初始化。为取得一个对象的左值必须引入一个新的运算符，这个新的运算符被称为取地址运算符，其符号是“&”。例如，下

面的语句定义一个整型变量并用该整型变量的左值初始化一个指针变量。

```
int Stu-No = 6002;
int * ipStu-No = & Stu-No;  //使指针变量 ipStu-No 取得变量 Stu-No 的地址
```

一个指针还可以被具有同样类型的另一个指针所初始化。在这种情况下，将不会出现取地址运算符。

一个指针变量还可以赋以零值，这表明它现在并不指向任何一个对象，作为指针变量的零通常写作 NULL。这里还有另外一个指针类型：void *，它将使指针有能力指向任意类型的对象。

1.6.3　字符串指针

最常用的指针类型是 char *。这是因为 C++ 中，所有有关字符串的操作都是通过字符指针来完成的。

字符串常量的类型是一指向该字符串中第一个字符的字符指针。也就是说所有字符串常量都是类型 char * 的对象。程序员可以说明一个 char * 型的变量，并用一个字符串给它初始化。例如：下面就是一个定义字符串指针并加以初始化的例子。

```
char * stExample = "The example of string!";
```

1.6.4　引用类型

引用是一个别名。当建立引用时，程序用另一个变量或对象的名字初始化它。对引用的改动，实际上是对目标的改动。引用类型是指对给定类型对象的引用。一个引用类型变量在定义时首先是定义类型标识符，而后是取地址标识符，最后是变量名。其格式如下：

T& 标识符 = 左值表达式

它建立了标识符与左值表达式所标识的对象之间的映像关系，值得注意的是，对引用对象的定义必须初始化。例如，下面的语句中定义了两个引用类型，其中第一个是合法定义，第二个则是非法的定义。

```
int value;
int & Value1 = value;  //合法定义引用类型，声明 Value1 是对整数的引用，初始化为引用
                       value
int & Value2;          //非法定义引用类型
```

一个引用类型通常被认为是另一变量的别名，在使用时当作某个对象的第二个名字。所有在引用变量上施加的操作实际上都作用到被引用变量的身上。例如：

```
int a;
int & r1 = a;
int & r2 = r1;
r2 = 10;    //这里 r2 也引用对象 a，所以，对象 a 被更新为 10
```

1.7　C++ 中的控制结构

编写程序时最重要的就是要了解应用程序的控制结构。控制结构是通过控制语句来

实现的。

1.7.1 条件语句

C++语言支持下列基本的条件语句、if 语句，if…else 语句、if…elseif 语句、? 条件和 switch 语句。

1. if 语句

if 语句用于在程序中有条件地执行某一语句序列，语法形式如下：

```
if(expression)
{
statement;  //当 expression 为真时执行 statement 语句序列
…
}
```

2. if…else 语句

if…else 语句根据不同的条件分别执行不同的语句序列，其语法形式如下：

```
if(expression)
{
  statement1;  //当 expression 为真时执行 statement1 语句序列
  …
}
  else                    //当 expression 为假时执行 statement2 语句序列
{
  statement2;
  …
}
```

3. if…elseif…else 语句

if…elseif 语句用于进行多重判断，其语法形式如下：

```
if(expression1)
{
  statement1; //当 expression1 为真时执行 statement1 语句序列
  …
}
elseif(expression2)  //当 expression2 为真时执行 statement2 语句序列
{
  statement2;
  …
}
…
else
{
  statementN;          //当所有条件均为假时执行 statementN 语句序列
  …
```

}

4. ? 条件操作符

? 条件操作符可以简化条件表达式的表达形式，其语法形式如下：

```
expression1? expression2:expression3
```

即表达式 expression1 为真时执行 expression2 的表达式，结果为 expression2 表达式的值；否则执行 expression3 表达式，结果为 expression3 表达式的值。

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

相当于：

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

5. switch 语句

switch 语句用于测试某一个变量具有多个值时所执行的动作。switch 语句的语法形式为：

```
switch (expression)
{
case constant1 :  statement1 ; break; //当 expression 与 constant1 相符时，执行 statement1 语句序列
case constant2 :  statement2 ; break; //当 expression 与 constant2 相符时，执行 statement2 语句序列
...
default : statement                   //当所有条件都不满足时，执行 default 后的语句序列
}
```

执行该语句，就是当 expression 的值与某个 case 的 constant 相等时，就执行其后面的语句，如当 expression 的值与 constant2 相等时，就执行 statement2 语句。在 switch 语句中也提供了一个类似于 else 的结构，它就是 default 标号。如果在 switch 的所有 case 分支语句中没有一个常量和 switch 中表达式的值相匹配，这时，如果定义的 switch 语句结构中包含有 default 子句，程序将执行 default 子句后的程序代码。

1.7.2　循环语句

许多程序都需要处理一组相似的对象，因此 C++ 提供了一种重复性操作的语句，叫做循环语句。它允许程序重复执行某一组语句。

C++ 提供了三种形式的循环结构：while，do 和 for 语句。这三种结构最大的不同在于怎样控制循环。在所有的循环结构中，表达式为非零时为真，而为零时为假。

1. while 语句

while 语句特别适合于处理字符串和其他指针类型，其语法结构如下：

```
while(expression)
{
    statement;
}
```

只要 expression 的值为真，就重复执行 statement 中的语句。

2. for 语句

for 语句通常用于处理具有固定长度的数据结构，例如数组等。下面是 for 循环语句的语法结构：

```
for(expression1;expression2;expression3)
{
    statement;
}
```

其中，expression1 可以是一条声明或表达式，它一般用于对一组变量进行初始化或赋值。当然，它也可以是一个空语句。expression2 用于控制循环，只要 expression2 的值为真则循环体中的语句将被执行。expression3 在每次循环体结束后执行，它一般用于修改在 expression1 中初始化的变量。循环体可以是简单语句也可以是复合语句，如果在第一次计算 expression2 的值时就得到了假值，则循环体中的语句一次都不会执行。

3. do…while 语句

对于 while 语句或者 for 语句来说，循环体中的语句只有在 while 或者 for 语句中的条件为真时，才能被执行，而 do…while 循环语句则总能保证循环体被执行一次。do…while 语句的语法结构如下所示：

```
do
{
    statement;
    …
}while(expression);
```

注意，在 do…while 循环结构中，while 语句的括号内指定循环的条件，该句结尾处应以分号(;)结束。

1.7.3 转移语句

转移语句使函数内的程序无条件地改变控制权，即在程序间进行控制转移。C++中的转移语句包括 break，continue 和 goto 语句。

1. break 语句

break 语句用于强制退出循环语句以及 case 语句的执行。当程序遇到这一语句后，将直接执行循环体后面的语句。例如：

```
for(i=0;i<10;i++)
{
    if (i>3)
    break;  //跳出循环,执行 cout 输出语句
}
cout<<"break";
```

上面的程序段，当遇到 break 语句时，整个循环被终止，开始执行 cout<<"break"语句。

2. continue 语句

continue 语句仅仅使当前进行的这一次最内层的循环中止，程序流程转去执行对真值

条件的计算。如果这时循环的真值条件为真，则开始下一次循环，如果为假则中止循环。

continue 语句与 break 语句不同，它并不中止整个循环的进行，而仅仅中止当前这一次循环的运行。例如：

```
for(i=0;i<100;i++)
{
    if(i>3)
    continue;  //继续执行循环动作
    cout<<"i"<<i<<endl;
}
```

执行上面的程序段时，当循环变量 i 大于 3 以后，就不再执行 continue 语句后面的输出语句，而是直接返回循环体的顶部开始执行。

3. goto 语句

goto 语句也能用来跳出循环和 switch 语句。它的一般语法形式为：

goto 语句标号；

其中的语句标号用一个标识符来表示，它的命名规则与命名变量名的规则相同，可以由字母、数字和下划线组成，其第一个字符必须为字母或下划线。例如，语句"goto Label 1;"是合法的，而语句"goto 1_Label;"则是不合法的。

由于大量使用 goto 语句将会使程序的流程无规律，程序的可读性降低，因此在程序设计中提倡少用 goto 语句，但也不是绝对不用，一般来说，可以用于以下两个方面：

(1) 与 if 语句一起构成循环语句。

(2) 从循环体中跳转到循环体外，尤其是在从多层嵌套循环的内层直接跳转到循环的外层时，可以考虑使用 goto 语句。

下面的程序是通过 goto 语句的跳转和 if 语句相配合实现循环的例子。

```
#include<iostream.h>
void main()
{
    int i=1,Sum=0;
    Loop: if(i<=20)                                    //跳转标号
      {
          Sum+=i;
          i++;
          goto Loop;                                   //跳转到 Loop 标识的那一行
      }
    cout<<"the summary of 1~20\t"<<Sum<<endl;          //输出结果
}
```

上面的程序执行结果为：

```
the summary of 1~20      210
```

1.8　函数

一个大型的程序一般可以分成一系列"单一功能模块"的集合。在 C++ 中，"单一功

能模块"就是函数。一个完整的 C++ 程序一般包含一个主函数和若干个子函数，主函数可以调用其他的子函数，子函数之间也可以互相调用。利用函数可以大大降低程序设计的工作量。很多编译系统本身就带有很多预定义的函数，并把它们以库函数的形式提供给程序设计人员。函数定义的一般形式如下：

```
类型标识符 函数名(形参列表)
{
    函数体;
}
```

在形参列表中，程序员必须指定每一个参数的数据类型。

在调用函数时，大多数情况下，主调函数和被调函数之间有参数传递的关系。因此，程序中广泛使用的函数还是有参数的函数。在有参数的函数中，在定义函数时的函数名后的括号中的变量名被称为"形式参数"(简称"形参")，在调用函数时，函数名后面括号中的表达式称为"实际参数"(简称"实参")。

熟悉 C 语言的读者不难了解，函数中的参数可以是变量、数组名和指针等内容，在这里就不再详细介绍了。

在 C++ 语言中，函数声明允许程序内含有多个名称相同的函数，这就是函数的重载，有关重载的内容，请参见 1.11 节的内容。

1.9 类与对象

传统的结构化语言都是采用面向过程的方法来解决问题。在面向过程的程序设计方法中，代码和数据是分离的，因此，程序的可维护性较差。面向对象(Object Oriented)程序设计方法则是把数据及处理这些数据的函数封装到一个类中，类是 C++ 的一种数据类型，而使用类的变量则称为对象

在对象内，只有属于该对象的成员函数才可能存取该对象的数据成员，这样，其他函数就不会无意中破坏其内容，从而达到保护和隐藏数据的效果。

与传统的面向过程的程序设计方法相比，面向对象的程序设计方法有三个优点：第一，程序的可维护性好。面向对象程序易于阅读和理解，程序员只需了解必要的细节，因此降低了程序的复杂性；第二，程序的易修改性好，即程序员可以很容易地修改、添加或删除程序的代码，这是通过添加或删除对象来完成的；第三，对象可以使用多次，即可重用性好。程序员可以根据需要将类和对象保存起来，随时插入到应用程序中，无需作什么修改。

面向对象程序设计方法提出了一些全新的概念，如类、封装(Encapsulation)、继承(Inheritance)和多态性(Polymorphism)等。

1.9.1 类的定义

类(Class)是 C++ 的精华，是进行封装和数据隐藏的工具。通过它把逻辑上相关的实体联系起来，并具备从外部对这些实体进行访问的手段。和函数一样，应用类也是C++

中模块化程序设计的手段之一。但是，函数是将逻辑上有关的语句和数据集合在一起，主要用于执行；而类则是逻辑上有关的函数及其数据的集合，它主要不是用于执行，而是提供所需要的资源。在使用一个类之前必须先定义类，定义一个类的语法格式如下所示：

```
class 类名:基类名
{
private:
    私有成员数据及函数;
protected:
    保护成员数据及函数;
public:
    公共成员数据及函数;
}[类的对象声明];
```

从上面的定义可以看到，一个类含有私有(private)成员、保护成员(protected)和公共成员(public)三部分。默认时在类中定义的项都是私有的。私有变量和函数只能被该类本身的成员函数存取或调用；保护成员除可以被本类中的成员函数访问外，还可以被本类派生的类的成员函数访问，因此用于类的继承；公共成员可以被本类以外的函数访问，是类与外部的接口。

类是面向对象程序设计最基本的单元，在设计面向对象程序时，首先要以类的方式描述实际待解决的问题，也就是将问题所要处理的数据定义成类的私有或公共类型的数据，同时将处理问题的方法定义成类的私有或公有的成员函数。

类也可以嵌套声明，例如：

```
class My-student
{
    class boy                          //嵌入类 boy,作为类 My-student 的成员之一
    {
        char boy-name[20];
        int boy-age;
    }my-boy-student;
    class girl                         //嵌入类 girl,作为类 My-student 的成员之一
    {
        char girl-name[20];
        int girl-age;
    }my-girl-student;
public:
    void student-input(void);
    void student-output(void);
}
```

在上面的例子中，类 My-student 的定义中嵌套了类 boy 和 girl。

1.9.2　对象

在 C++ 中，对象是声明为类类型的一个数据项，是类的实际变量。程序员也可以定

义类的变量。这个定义的类的变量在 C++ 中就被称为对象。对象有时也称为类的实例(Instance)。由此可见,类是程序中的一个静态的概念,而对象是程序中的一个动态的概念。

在 C++ 中有两种方法可以定义类的对象:

第一种是在定义类的同时直接定义类的对象,即在定义类的右大括号"}"后直接写出属于该类的对象名表列,即:

```
class 类名
{
  成员变量表列;
  成员函数表列;
}对象名表列;
```

还有一种是在定义好类后,再定义类的对象,其一般格式如下:

```
类名 对象 1[,对象 2,…];
```

在 C++ 中通常也把类的成员函数称为类的方法。成员函数的原型一般在类的定义中声明,在类的定义中声明其成员函数的语法与声明普通的函数所用的语法完全相同。方法的具体实现既可以在类的定义内部完成(这种方式定义的类的方法有时也称为类的内联函数),也可以在类的定义之外进行,而且方法的具体实现既可以和类的定义放在同一个源文件中,也可以放在不同的源文件中。

类的方法的具体实现和普通函数的具体实现只是在函数的头部有略微不同的格式。一般来说,如果类的方法的定义是在类的外部实现的,则在定义方法时必须把类名放在方法名之前,中间用作用域运算符("::")隔开,其一般形式如下所示:

```
类名::方法名
```

这样,即使几个类中的方法名相同,也可以用这种形式把它们区分开来。和普通函数一样,类的方法也应该既有返回值类型,也有参数表列(当然也可以没有参数)。一个方法的头部各部分如下所示:

```
访问控制:函数返回值 函数名(参数表列);
```

下面的一段程序实现在程序中定义一个名为 angle 的类。

```
#include<iostream.h>
#include<math.h>
const double ANGTORAD=0.01745329325;  //定义弧度和度之间的转换比例,即弧度的值
class angle                           //定义类 angle
{
    double value;                     //类 angle 的私有成员数据
public:                               //类 angle 的公共成员函数
    void SetValue(double);
    double GetSine(void);
}deg;                                 //声明类 angle 的对象 deg

void angle::SetValue(double a)        //类 angle 的成员函数 SetValue 的具体实现
```

```
    value = a;
}
double angle::GetSine(void)                    //类 angle 的成员函数 GetSine 的具体实现
{
    double temp;
    temp = sin(ANG-TO-RAD * value);
    return temp;
}

void main()
{
    deg.SetValue(60.0);                        //给类 angle 的成员变量 value 赋值
    cout << "The sine of the angle is:";
    cout << deg.GetSine() << endl;             //输出正弦值
}
```

1.9.3　内联函数

类的方法也可以声明和定义成内联函数。内联函数是指那些定义在类体内的成员函数，即该函数的函数体放在类体内。内联函数在调用时不像一般的函数那样要转去执行被调用函数的函数体，执行完成后再转回调用函数中，然后执行其后语句，而是在调用函数处用内联函数体的代码来替换，这样将会提供运行速度。因此内联函数主要是解决程序的运行效率问题。值得注意的是，内联函数一定要在调用之前定义，并且内联函数无法递归调用。在 C++ 中可以使用下面两种格式定义类的内联函数：

(1) 当在函数的外部定义时，把关键字 inline 加在函数定义之前。例如，下面的程序段中定义的类 angle 的 SetValue 方法被定义成内联函数。

```
class angle                                    //定义类 angle
{
  private:
     double value;
  public:
     void SetValue(double);
};
inline void angle::SetValue(double x)          //定义内联函数
{
     value = x;
}
```

(2) 把函数原型声明和方法的定义合并，放入类定义中。例如，下面的程序段在声明类 angle 的 SetValue 方法后，紧接着就定义该方法的具体实现。

```
class angle                          //定义类 angle
{
```

```
  private:
      double value;              //定义私有数据成员
  public:
      void SetValue(double x)  //定义内联函数
      {
          value = x;
      }
};
```

1.10 构造函数和析构函数

C++ 中有几类特殊的成员函数，这些函数决定了如何建立、初始化、拷贝及删除对象。构造函数(Constructor)和析构函数(Destructor)是其中最重要的两种。和一般的成员函数一样，构造函数和析构函数既可以在类的内部声明和定义，也可以在类的内部声明，在类的外部定义。如果一个类含有构造函数，则在建立该类的对象时系统会自动调用它；而如果一个类含有析构函数，则在销毁该类的对象时系统会自动调用它。

1.10.1 构造函数

构造函数是一种特殊的成员函数，它主要用来为对象分配内存空间，对类的数据成员进行初始化并执行对象的其他内部管理操作。构造函数的特点是构造函数的名字和它所在的类名相同，当定义该类的对象时，构造函数完成对此对象的初始化。它可以接收参数并允许重载。当一个类含有多个构造函数时，编译程序为了确定调用哪一个构造函数，需要把对象中使用的参数和构造函数的参数表进行比较，这个过程与在普通的函数重载中进行选择的过程是一样的。

下面的程序中定义了一个包含构造函数的类。

```
class student
{
  private:
      int Num;
      float Score;
  public:
      student(int No,float s)  //声明类 student 的构造函数，该函数名与类名 student 一致
      {
          Num = No;
          Score = s;
      }
...
};
```

上面定义的类名为 student，其构造函数名也为 student，该函数用来给对象赋初值。构造函数是在定义对象的同时调用的，其一般形式为：

类名　对象名（实参表）;

关于构造函数还必须说明以下几点：

（1）构造函数和普通函数一样也可以有参数，但不能有返回值。这是因为构造函数通常是在定义一个新的对象时调用，它无法检查构造函数的返回值。

（2）在实际应用中，如果没有给类定义构造函数，则编译系统将为该类生成一个默认的构造函数，该默认的构造函数没有参数，只是简单地把对象中的每个实例变量初始化为 0。

（3）构造函数可以有默认参数。例如，下面定义的类 My-class 中的构造函数带有一个默认参数 x = 3.5，y = 8.5。

```
class My-class
{
    float a,b;
  public:
    My-class(float x = 3.5,float y = 8.5)              //带有默认参数的构造函数
    {
        a = x;
        b = y;
    }
    …
};
```

构造函数 My-class 的两个参数均为默认参数，在定义对象时可以省略实参。因此，可以用下面的语句来定义类 My-class 的对象。

```
My-class c1(2,4);
My-class c2(4);
My-class c3;
```

（4）构造函数也可以没有参数。例如，下面定义的类 My-class 中的构造函数就没有参数，该类的构造函数执行的功能是在屏幕上输出一句“NO PARAMETER”。

```
class My-class
{
  public:
    My-class()                  //没有参数的构造函数
    {
        cout <<"NO PARAMETER" << endl;
    }
};
```

（5）在重载没有参数的构造函数和有默认参数的构造函数时，有可能产生二义性。例如，下面定义的类 special-class 中就会产生二义性。

```
class special-class
{
      int i;
```

```
public:
    special_class();
    special_class(int x=1);
};
...
void main()
{
    special_class c1(10);          //正确,使用 c1:: special_class(int)构造函数
    special_class c2;              //错误,无法精确调用构造函数
    ...
};
```

上面的程序中定义了两个重载函数 special_class,其中一个没有参数,另一个有一个默认参数。如果在定义对象时不给出参数,则会产生二义性,因为编译系统无法确定应当使用哪一个构造函数。在实际应用中,要注意避免这种情况。在类的定义中,还允许其他类的对象作为它的数据成员,这种数据成员称为成员对象。

1.10.2 析构函数

析构函数也是类中的特殊成员函数,与定义它的类具有相同的名字,但要在前面加上一个波浪号(~)。析构函数没有参数,也没有返回值,而且也不能重载,因此一个类中只能有一个析构函数。

析构函数执行与构造函数相反的操作,通常用于释放分配给对象的存储空间。当程序超出类对象的作用域时,或者当对一个类指针使用运算符 delete 时,系统将自动调用析构函数。

和构造函数一样,如果在类的定义中不定义析构函数,编译系统将为之产生一个默认的析构函数,对于大多数类来说,默认的析构函数就能满足要求。如果在一个对象完成其操作之前还需要做一些内部处理,则应定义析构函数。例如,下面定义的类 My class 中定义了该类的析构函数。

```
class My_class
{
    char * str;
    int MaxLen;
public:
    My_class(char * )    //定义类 My class 的构造函数
    {
        str = new char[maxLen];
    }
    ~My_class()          //构造类 My class 的析构函数
    {
        cout << "Here delete the str" << endl;
        delete str;
```

```
    ;
    void GetString(char * );
};
```

上面的例子是析构函数和构造函数最常见的用法，即在构造函数中用运算符 new 为字符串分配存储空间，最后在析构函数中用运算符 delete 释放已分配的存储空间。析构函数也可以在类定义的内部声明而在类定义的外部定义。例如，下面的析构函数 My class 在类定义的外部定义如下：

```
My-class::~My-class()
{
    delete str;
}
```

1.11　方法重载

重载是 C++的一个重要特征，它包含函数重载和操作符重载。

1.11.1　函数重载

所谓函数重载是指同一个函数名可以对应着多个函数的实现。函数重载允许一个程序内声明多个名称相同的函数，这些函数可以完成不同的功能，并可以带有不同的类型、不同数目的参数及返回值。使用函数重载可以减轻用户的记忆负担，并使程序的结构简单、易懂。

函数重载要求编译器能够唯一地确定调用一个函数时应执行哪个函数代码，即采用哪个函数的实现。确定函数实现时，要求从函数参数的个数和类型上来区分。也就是说，函数重载时，要求函数的参数个数或参数类型不同。

例如，下面的程序中实现的是两个不同参数类型的重载。在类 My-class 中对方法 plus 进行了重载，通过重载，使得在求两个数的和时不用再区分整数和浮点数之间的不同之处，而只需直接调用类 My-class 的方法 plus 即可。

```
#include<iostream.h>
#include<math.h>
#include<stdlib.h>
class My-class
{
public:
    int plus(int,int);
    double plus(double, double);
};

int My-class::plus(int x, int y)
{
    return x+y;
```

```
}
double My class::plus(double x, couble y)
{
      return x + y;
}

void main()
{
      My class Data;
      cout <<"The result for plus(int,int) is:" << Data.plus(5,10) << endl;
      cout <<"The result for plus(double, double)is:"<< Data.plus(5.0,10.0) << endl;
}
```

对照上面的程序,可以看出,两个重载的成员函数分别调用求整型数之和及求浮点数之和的函数 plus 来对不同的参数进行求和运算。

不仅在类的成员函数上可以实现重载,在构造函数上,也可以实现函数的重载。例如:

```
class sample
{
      int i;
  public:
      sample();                  //定义重载的构造函数
      sample(int);
      void display();
};

sample::sample()                 //定义构造函数 sample
{
      i = 0;
}

sample::sample(int x)            //定义构造函数 sample(int)
{
      i = x;
}
```

定义重载的构造函数后,声明对象时就可以根据不同的参数来分别调用不同的构造函数。例如:

```
void main()
{
      sample A;                  //自动调用构造函数 sample
      sample B(5);               //自动调用构造函数 sample(int)
}
```

在处理不同类型的数据时，成员函数的重载使得类具有更大的灵活性和通用性，这也有力地支持了通过类来对我们周围的客观世界进行抽象的程序设计的思想。

1.11.2　操作符重载

操作符重载的能力增强了 C++ 语言的可扩充性，对操作符的重载是将 C++ 语言中已有的操作符赋予新的功能，但与该操作符的本来含义不冲突，使用时只需根据操作符出现的位置来判断其具体执行哪一种运算。

C++ 中单目运算、双目运算、物理内存管理运算符 new 和 delete 以及指针、引用等运算操作均可以重载，但要注意的是，由于单目运算操作只能有一个参数，因此重载 ++ 和 -- 运算操作时，不可能区分是前置操作还是后置操作。

使用操作符重载时，必须用以下的方式来声明成员函数：

函数类型 operator #(形参表)

其中，operator 是关键字，# 表示欲重载的操作符，函数类型指明返回值类型，通常与类类型一致或为 void 型。以下是一个操作符“++”重载的例子。C++ 中约定，在增量运算操作符中，放上一个整数形参，就是后置运算符。

```
#include <iostream.h>
class OperClass                          //声明一个类
{
      int x;
public:
      OperClass();                       //构造函数
      OperClass operator ++();           //声明重载的操作符 ++，返回值类型为
                                         //OperClass 类
                                         //这里的 ++ 为前置运算
      OperClass operator ++(int);        //声明重载的操作符 ++，返回值类型为
                                         //OperClass 类
                                         //这里的参数 int 表示此 ++ 为后置运算
      void display();                    //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()
{
    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=++X;                                  //对象 X 调用前置操作符++,并将返回值赋给对象 Y
}
```

上例中介绍的是单目操作符重载,因此有前置运算和后置运算之分,若使用双目操作符重载,则没有这个问题,例如将上例作修改如下:

```
class OperClass
{
    int x;
public:
    OperClass();
    OperClass Operator+(OperClass);
                        //声明重载的操作符+,具有两个 OperClass 类类型的操作数,
                        //返回值也是 OperClass 类类型
    void display();
};
...
OperClass OperClass::OperClass+(OperClass A)
                        //定义重载的操作符 A 的具体操作
{
    OperClass B;
    B.x=x+A.x;
    return B;
}
```

```
void main()
{
    OperClass X,Y,Z;      //声明三个对象
    Z = X + Y;            //对象 X 调用重载操作符 + 与对象 Y 相加,并将返回值赋给 Z
}
```

上例中的重载操作符“+”是双目运算符,使用时与普通的“+”(加)没有什么差别。

由于成员函数具有一个隐含的 this 指针,所以当单目运算符作为类的成员时,重载该运算符不用带参数。

1.12　友元

类的主要特点是数据隐藏,即类的私有部分在该类的作用域之外是不可见的。但是,有时候可能需要在类的外部访问类的私有部分。为此,C++提供了一种方法,允许类外部的函数或者类具有该类的私有部分的特权,它通过关键字 friend 把其他类或非成员函数声明为一个类的“友元”。在类的内部,友元被作为该类的成员看待,并且对对象公用部分的访问没有任何限制。

友元函数的声明方式为:

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

例如,下面的程序段中定义了类 Friend_Class 及一个友元函数 Friend_Function。

```
class  Friend_Class          //类 Friend_Class 的定义
{
private:
    int nMemberData;
public:
                             //声明函数 Friend_Function 为类 Friend_Class 的友元
    friend void Friend_Function(Friend_Class class_member,int x);
};
                        //下面是友元函数 Friend_Function 的函数体,它要在类的外部定义
void Friend_Function(Friend Class class_member,int x)
{
    class_member.nMemberData = x;  //通过友元函数访问类的私有成员
}
```

上面的程序中声明的友元函数 Friend_Function 并不是类 Friend_Class 的成员函数,而

是一个普通的函数，不过由于它在类 Friend Class 的定义中被声明为友元函数，因此，函数 Friend-Function 可以访问类 Friend-Class 的私有部分。函数在实际操作时，是通过传递给它的类 Friend-Class 的对象 class member 进行对类的私有部分的访问的。

注意，友元函数的定义与成员函数的定义是不一样的，它与普通函数的定义形式基本相同。在它的前面没有类名和作用域运算符“::”。

类的友元可以是一个函数，也可以是一个类。例如，下面的程序中将整个教师类看成是学生类的友元：

```
class Student;
class Teacher

public:
    void assignGrades(Student& s);
    void adjustHours(Student& s);
    ...
protected:
    int noOfStudent;
    Student *pList[100];
};

class Student
{
public:
    friend class Teacher;  //友元类
    ...
    protected:
    int semesterHours;
    float gpa;
};
```

上面程序中，友元类是 Teacher。C++中规定，友元类必须在它被定义以前声明。因此，上面的例子中，类 Teacher 定义在类 Student 之前。一旦一个类被声明为另一个类的友元函数之后，该类的每一个成员函数都是另一个类的友元函数。一个类的友元的声明既可以在该类定义的公用部分声明，也可以在类的私有部分声明。两个类还可以相互定义为对方的友元，当两个类的联系较紧密时，把它们定义为相互的友元就更为有意义了。

尽管使用友元函数可以访问类中的私有数据，但为了确保数据的完整性及数据封装与隐藏的原则，建议尽量少用或不用友元函数。

1.13 this 指针

this 指针是指向一个类的对象的地址。this 是一种隐含指针，它隐含于每个类的成员函数之中，也就是说，每个成员函数都有一个 this 指针变量，this 指针指向该成员函数所属

的类的对象。当定义一个类的对象时，该对象的成员均含有由系统自动产生的指向当前对象的 this 指针。成员函数访问类中成员变量的格式可以写成：

this -> 成员变量

当一个对象调用成员函数时，该成员函数的 this 指针便指向这个对象。如果不同的对象调用同一个成员函数，C++编译器将根据成员函数的 this 指针所指向的不同对象类确定应引用哪一个对象的数据成员。也就是说，每个对象都有一个地址，而 this 指针所指的就是这个地址。

和其他数据类型一样，程序中也可以定义指向类对象的指针，在定义了类的指针后，还必须为其分配内存才能使用，类对象的指针定义及分配内存空间的一般格式为：

类名 * 指针名 = new 类名;

例如，下面的语句定义类 Student 的对象指针并为其分配内存：

```
Student * Student1 = new Student;
```

当通过类对象的指针访问类的成员时，通常可以使用运算符“->”，例如，下面的程序中，主函数通过指向类对象的指针调用类的成员函数。

```
#include <iostream.h>
class Class1                    //定义类 Class1
{
    int Value;                  //定义类的私有成员
public:
    Class1(int Val)             //类 Class1 的构造函数
    {
        Value = Val;            //对成员变量初始化
    }
    int GetValue(void)          //类 Class1 的成员函数
    {
        return Value;           //获取类对象的成员变量的值
    }
};

void main()                     //主函数
{
    Class1 Object1(888), * p;  //定义一个类 Class1 的对象和一个对象指针
    p = &Object1;               //使对象指针指向对象 Object1
    cout << "The value of Object1 is:" << p -> GetValue();  //通过对象指针调用成员函数
    cout << endl;
}
```

上面的程序执行结果如下：

The value of Object1 is:888

通过对象指针返问对象的成员变量或函数还可以通过点运算符“.”来实现，不过应注意要在对象指针前加上星号“*”。例如，上面通过对象指针访问类的成员函数的方法可

以改写如下：

```
cout<<"The value of Object1 is:"<<(*p).GetValue();
```

1.14 继承

类是 C++ 中进行数据封装的逻辑单位，C++ 还提供了一种继承机制，利用这种机制，用户可以通过增加、修改或替换给定类中的方法来对这个类进行扩充，以适应不同的应用要求。

利用继承机制，程序员可以在已有类的基础上构造新类，这一性质使得类支持分类的概念。如果不使用分类，则对每一个对象都定义其所有的性质。使用分类后，可以只定义某个对象的特殊性质。每一层的对象只需定义属于它本身的性质，其他性质可以从上一层"继承"下来。

1.14.1 派生类

派生类(也称子类)是 C++ 提供继承的基础，也是对原来的类进行扩充和利用的一种基本手段。C++ 派生类继承或修改原有类中的部分或全部方法，而且可以增加原来类中没有的新方法，以满足使用派生类的需要。

一个类可以继承另一个类的属性。其中被继承的类叫做基类(Base class)，继承后产生的类叫做派生类(Derived class)。有的时候，我们也把基类称为"父类"，而把派生类称为"子类"。

可以把派生类看作基类的扩展，它提供了一种简单、灵活且有效的机制对基类进行利用和扩展。派生类从基类中继承所有的公共部分，并可以增加数据成员和成员函数。这使程序员可以根据基类与派生类的差异来建立特定对象的新类，对相同部分的代码不必重新定义。此外，还可以为多个不同的类提供公用界面，使程序设计人员更容易表达类型间的关系，从而减少程序设计的工作量。

任何类都可以作为基类，一个基类可以有一个或多个派生类，一个派生类还可以成为另一个类的基类。

定义派生类的一般格式如下：

```
class 派生类名:[访问属性] 基类名
{
    ...
};
```

其中：

(1) class 是类定义的关键字，用于告诉编译器下面定义的是一个类。

(2) 派生类名是新定义的类名。

(3) 访问属性是访问说明符，可以是 private、public 和 protected 之一。此项的默认值为 private，派生类名和访问属性之间用冒号隔开。派生类的访问控制由访问属性来确定，它按下述方式来继承基类的访问属性：

- 如果访问属性为 public，则基类的 public 成员是派生类的 public 成员；基类的 private 成员是派生类的 private 成员；基类的 private 成员对基类仍保持 private 属性。
- 如果访问属性为 protected，则基类的 public 和 private 成员均是派生类的 protected 成员；基类的 private 成员对派生类仍保持 private 属性。具体来说，基类中声明为 protected 的数据只能被基类的成员函数或其派生类的成员函数访问；不能被派生类以外的成员函数访问。
- 如果访问属性为 private，则基类的 public 和 protected 成员都是派生类的 private 成员；基类的 private 成员对基类仍然保持 private 属性。也就说是，当访问属性为 private 时，派生类的对象不能访问基类中以任何方式定义的成员函数。

(4) 基类名可以有一个，也可以有多个。如果只有一个基类，则这种继承方式叫做简单继承；如果基类名有多个，则这种继承方式称为多重继承。各个基类名之间用逗号隔开。

1.14.2　多重继承

多重继承的格式与简单继承的格式基本相同，其一般用法如下所示：

class 派生类名:[访问属性]基类名表

其中，基类名表是两个或两个以上的基类名，各基类名之间用逗号隔开，在每个基类之前都应指明访问属性，默认的访问属性为 private。

例如，下面的程序中定义的类 MultiDerived 继承基类 Base1 和 Base2，它是实现多重继承的一个很好的例子。

```
#include<iostream.h>
class Base1                          //定义基类 Base1
{
protected:
    int m_B1;                        //定义基类的保护数据成员 m_B1
public:
    void Setm_B1(int x)              //定义基类的保护成员函数 Setm_B1
    {
        m_B1 = x;
    }
};

class Base2                          //定义基类 Base2
{
protected:
    int m_B2;                        //定义基类 Base2 的保护数据成员 m_B2
public:
    void Setm_B2(int x)              //定义基类 Base2 的保护成员函数 Setm_B2
    {
        m_B2 = x;
```

```
    }
};

class MultiDerived:public Base1,public Base2  //定义基类 Base1 和 Base2 的派生类
                                               //MultiDerived
{
public:
    void GetB1B2(void)              //存取继承自基类 Base1 和 Base2 中的数据成员
    {
        int Result;
        Result = m_B1 + m_B2;
        cout << "m_B1 + m_B2 = ";
        cout << Result << endl;
    }
};
void main(void)                     //主函数
{
    MultiDerived M;                 //定义派生类 MultiDerived 的对象
    M.Setm_B1(15);                  //调用继承自基类 Base1 的成员函数 Setm_B1
    M.Setm_B2(35);                  //调用继承自基类 Base2 的成员函数 Setm_B2
    M.GetB1B2();                    //调用派生类中自定义的成员函数 GetB1B2
}
```

上面程序的运行结果为：

m_B1 + m_B2 = 50

上面的程序中，类 MultiDerived 继承自基类 Base1 和 Base2，因此继承了两个类的成员，可以访问基类 Base1 和 Base2 中定义为 protected 和 public 的成员。在主函数中，定义了类 MultiDerived 的对象 M，然后分别调用类 Base1 和 Base2 中的成员函数，完成初始化、计算和输出操作。

多重继承在给程序设计带来极大方便的同时，也给程序带来了以下负面的问题。如果从类库的组织的角度看，多重继承必然会增加类库结构的复杂性，从而为程序的稳定性留下隐患。从程序设计的角度看，其负面影响就是容易带来二义性。如果基类的多个基类中都同时定义了同名的成员，则编译程序将不能准确地理解程序员的意图，从而导致错误，因此，使用多重继承时要谨慎。

1.15 多态性和虚拟函数

简单地讲，多态性就是“一种实现一种接口，多种方法”的技术，是面向对象程序设计的重要特性。

1.15.1 多态性

面向对象的语言多数都支持多态性。从本质上讲，多态性可以引用多个类的实例。

利用多态性，程序员可以向一个对象发送消息来完成一系列操作，而不用关心软件系统是如何来实现这些操作的。在系统设计阶段，当设计人员决定把某一类型的活动用于一个给定的对象时，并不关心这个对象如何解释这个活动以及这个方法如何实现，而只关心这个活动对这个对象所产生的作用。C++ 允许程序员向不同但有关的对象发送同样的消息和完成同样的操作，而让软件系统决定如何为给定的对象完成所需要的工作。

利用多态性，可以在基类和派生类中使用同样的函数名而定义不同的操作，从而实现"一个接口，多个方法"，这是一种在运行时出现的多态性，它通过派生类和虚拟函数来实现。虚拟函数是在基类中的成员函数前加上 virtual，然后在派生类中再加以定义的函数。当用指向派生类的对象的基类指针对函数进行访问时，系统将根据运行时指针所指向的实际对象来确定调用哪一个派生类的成员函数版本。当指针指向不同的对象时，执行的是虚拟函数的不同版本。

用多态性可以实现自上而下的设计方法。这是一种从全局出发，用类的层次结构来模拟客观世界的程序设计方法。通俗地说，多态性是指用一个相同的名字定义不同的函数，这些函数执行过程不同，但是有相似的操作，即用同样的接口访问不同的函数。运算符重载和函数重载就是一种多态性，这是编译时的多态性，也称静态多态性。而运行时的多态性则称作动态多态性。

1.15.2　虚拟函数

当调用重载函数时，编译系统对函数原型进行比较，以决定调用哪一个函数。但是，当指针既指向派生类又指向基类时，就会产生潜在的二义性问题。

例如，下面的程序就会产生二义性问题：

```
#include<iostream.h>
class Base1                    //定义基类 Base1
{
protected:
    int m_B1;                  //定义基类的保护数据成员 m_B1
public:
    void SetMember(int x)      //定义基类 Base1 的保护成员函数 Set Member
    {
        m_B1 = x;
    }
};

class Base2                    //定义基类 Base2
{
protected:
    int m_B2;                  //定义基类 Base2 的保护数据成员 m_B2
public:
    void SetMember(int x)      //定义基类 Base2 的保护成员函数 Set Member
    {
```

```
        m_B2 = x;
    }
};

class MultiDerived:public Base1,public Base2 //定义了基类 Base1 和 Base2 的派生类
                                              //MultiDerived
{
public:
    void GetB1B2(void)        //存取继承自基类 Base1 和 Base2 中的数据成员
    {
        int Result;
        Result = m_B1 + m_B2;
        cout << "m_B1 + m_B2 = " << cout << Result << endl;
    }
};
//主函数
void main(void)
{
    MultiDerived M;           //定义派生类 MultiDerived 的对象
    M.SetMember(10);          //调用继承自基类 Base1 的成员函数 Set Member
    M.SetMember(20);          //调用继承自基类 Base2 的成员函数 Set Member
    M.GetB1B2();              //调用派送类中自定义的成员函数 GetB1B2
}
```

在 VC 6.0 下编译上面的程序，编译程序将提示如下的错误信息，表明程序中存在二义性：

error C2385: 'MultiDerived::SetMember' is ambiguous

在继承机制下，编译程序有时难以决定调用哪一个重载函数，只有在运行时才能确定，这就是滞后绑定，用滞后绑定可以实现多态性。C++是通过虚拟函数来处理滞后绑定的。虚拟函数必须在基类中用关键字 virtual 加以说明。关键字 virtual 在基类中只能使用一次，而在派生类中使用的是重载的函数名。

例如，下面的程序中通过使用虚拟函数来解决基类、派生类中同名函数的调用引起的二义性：

```
#include<iostream.h>
class Base                                //定义基类 Base
{
public:
    virtual void VirtualFunc(void)        //虚拟函数 VirtualFunc 在基类中的定义
    {
        cout << "Here is Base \n";
    }
};
```

```
class Derived:public Base                    //派生类的定义
{
public:
    void VirtualFunc(void)                   //虚拟函数 VirtualFunc 在派生类中的定义
    {
        cout <<"Here is Derived";
    }
};
//主函数
void main()
{
    Base * BasePtr,BaseObject;               //定义指向基类的指针和基类的对象
    Derived DerivedObject;                   //定义派生类对象
    BasePtr = &BaseObject;                   //基类指针 BasePtr 指向基类对象 BaseObject
    BasePtr -> VirtualFunc();                //调用基类中定义的函数 VirtualFunc
    BasePtr = &DerivedObject;                //基类指针 BasePtr 指向派生类对象
                                             //DerivedObject
    BasePtr -> VirtualFunc();                //调用派生类中的函数 VirtualFunc
    cout << endl;
}
```

上面的程序执行结果如下：

Here is Base

Here is Derived

上面的程序中，通过定义指向基类的指针 BasePtr 来调用基类和派生类各自的函数 VirtualFunc。从运行结果可以看出，BasePtr 指针所指向的对象不同，所调用的程序也不同。也就是说，通过改变指针 BasePtr 所指向的对象，可以用一个指针变量调用不同的函数从而实现的“一个接口，多种方法”，这就是多态性。

上面的程序中，如果去掉关键字 virtual，则上面的程序的运行结果为：

Here is Base

Here is Base

程序在基类中定义了虚拟函数 VirtualFunc，在派生类中也定义了函数 VirtualFunc，该函数在两个类中重载。虚拟函数的重载和普通函数的重载是有区别的。从上面的程序中可以看出，当发送指向一个对象的指针的消息时，使用了形如“对象 - > 消息”的记号。如果没有关键字 virtual，则系统在编译时采用早期绑定，它根据该指针对象的类型确定与这个消息有关的对象。

当一个函数定义声明为 virtual 函数时，就要使用滞后绑定。在上面的例子中函数 VirtualFunc 被声明为虚拟函数，VirtualFunc 函数的地址生成一张表，而在运行时再指向一个具体的地址。每个对象都有一个内部指针指向这张表。

可以看出，使用虚拟函数可以使类的使用更为灵活，但这种灵活性是有代价的。即使

用虚拟函数比普通函数需要更多的开销，其原因为：在 C++ 中，采用一种叫做“函数指针表”的技术来实现虚拟函数的操作。每个类都有一张虚拟函数表，简称 VFT(Virtual Function Table)。在 VFT 中，类的每个虚函数都有一个相应的指针。例如，下面的类：

```
class VirtualClass

public:
    char *ClassName;
    VirtualClass(char ch);
    virtual ~VirtualClass(void);
    virtual void GetClassName(void);
```

为了能调用虚拟函数，必须把指针定位到相应的 VFT 表，这个指针是隐藏的。当调用虚拟函数时，用这个指针来查找 VFT 表，利用该表的索引指向虚拟函数的入口点。

在多数情况下，这种额外开销不会对程序的效率有明显的影响。但是，如果在程序运行期间有大量的循环，可能会降低效率。在这种情况下，一般不要使用虚拟函数。

在面向对象的程序设计中，多态性的重要性体现在：允许在基类中声明本类和派生类都共有的函数，同时允许在派生类中对其中的某些或全部函数进行特殊定义。用面向对象的术语来说，前者叫作“泛化”(或一般化、通用化)，后者叫作“特化”(或具体化、特殊化)。根据这一特性，我们可以设计一个抽象的基类，在该类中的函数是没有实现的，然后在各个派生类中定义这些函数。在基类中定义派生类所具有的通用接口(泛化)，而在派生类中定义各自的具体实现(特化)。因此，利用基类和派生类形成一种从抽象到具体、从一般到特殊的层次关系，是多态性应用的根本思想。基类提供了派生类直接使用的所有成员函数，而派生类必须定义这些函数的实现版本。由于基类定义了接口形式，所以它的任何派生类都使用同一接口。在 C++ 中，统一的接口是通过虚拟函数来实现的，这种方法更接近于人类的自然思维方法，所有派生类的对象都以同样的方式访问接口。此外接口和实现是分离的，这就为建立各种类库提供了方便。

1.15.3 虚拟析构函数

C++ 中不可以定义虚拟构造函数，但是可以定义虚拟析构函数。如果在基类和派生类中都定义了析构函数，而且希望程序能够根据需要执行基类中的析构函数或者派生类中的析构函数，则必须把基类中的析构函数定义为虚拟析构函数，否则不能实现多态性。

虚拟析构函数以关键字 virtual 开头，这样定义后，基类中的析构函数及派生类的析构函数全部是虚拟析构函数。与一般的虚拟函数不同，基类中虚拟函数的名字与派生类析构函数的名字可以不一样。

虚拟析构函数的概念虽然十分简单，但它在面向对象程序设计中却是一种十分重要的技巧。一般的处理方法是，当在程序中定义了基类和派生类时，把基类中的析构函数设置为虚拟析构函数。

1.16　流

C++中把数据之间的传输操作称作流。在C++中，流既可以表示数据从内存传送到某个载体或者设备中，即输出流；也可以表示数据从某个载体或者设备传送到内存缓冲区变量中，即输入流。数据在不同的设备之间传送后不一定会消失，广义地讲，也可以把与数据传送有关系的事务叫做流。例如，可以把文件变量叫做流。有时候，流还可以代表要进行传送的数据的结构、属性和特性，用一个名字来表示，叫做流类；而用流代表输入设备和输出设备，叫做流的对象。

有关输入输出的操作并没有在C++语言中定义，但它包含在C++的实现中，并作为C++的一个标准类库提供。在这里所讨论的I/O库指的是iostream库。任何一个使用iostream库的程序都必须包含头文件iostream.h。

1. 流对象cout和cin

C++把进行数据传送操作的设备也看作是对象。cout代表输出设备，cin代表输入设备。预定义输出流cout代表标准输出设备，即显示器；预定义输入流cin代表标准输入设备即键盘。

2. 插入操作符"<<"

插入操作符"<<"的左操作数是代表输出设备的对象，如cout，右操作数是要输出的内容。插入符"<<"是重载的左移位运算符。在遇到这个运算符时，C++编译器首先检查"<<"的左操作数和右操作数，以判别应该执行左移位还是执行插入操作，因此二者不会出现干扰。

3. 抽取操作符">>"

抽取符的左操作符是代表输入设备的对象，例如cin，右操作数是内存缓冲区中的变量。抽取操作符">>"是重载的右移位运算符。在遇到这个运算符时，C++编译程序首先检查">>"的左操作数和右操作数，以判别应该执行右移位还是执行插入操作，因此二者不会出现干扰。

习　题

1-1　C++语言的基本数据类型包含哪些？

1-2　C++语言中函数的作用是什么？

1-3　C++中的指针有哪些类型？它们都如何定义？

1-4　在C++中如何进行"类"的定义？

1-5　构造函数和析构函数的功能是什么？如何创建构造函数和析构函数？

1-6　C++中"派生"的含义是什么？

1-7　C++中"重载"的作用是什么？

第 2 章

Visual C++ 6.0 开发环境

2.1 Visual C++ 及其开发环境概述

Visual C++ 自诞生以来，一直是 Windows 环境下最主要的应用开发系统之一。Visual C++ 不仅是C++ 语言的集成开发环境，而且与 Win32 紧密相连，所以，利用 Visual C++ 可以完成各种各样的应用程序的开发，从底层软件直到上层直接面向用户的软件。而且，Visual C++ 强大的调试功能也为大型复杂软件的开发提供了有效的排错手段。

进入 90 年代以来，随着多媒体技术和图形图像技术的不断发展，可视化（Visual）技术得到广泛的重视，越来越多的计算机专业人员和非专业人员都开始研究并应用可视化技术。所谓可视化技术，一般是指软件开发阶段的可视化和对计算机图形技术和方法的应用。本书将讨论软件开发阶段的可视化，即可视化编程。Visual C++ 是一个很好的可视化编程工具，使用 Visual C++ 环境来开发 Windows 应用程序大大缩短了开发时间，而且它的界面更友好，便于程序员操作。在没有可视化开发工具之前，程序员要花几个月的时间来完成 Windows 程序的界面开发，而现在只需较少的时间就可完成。

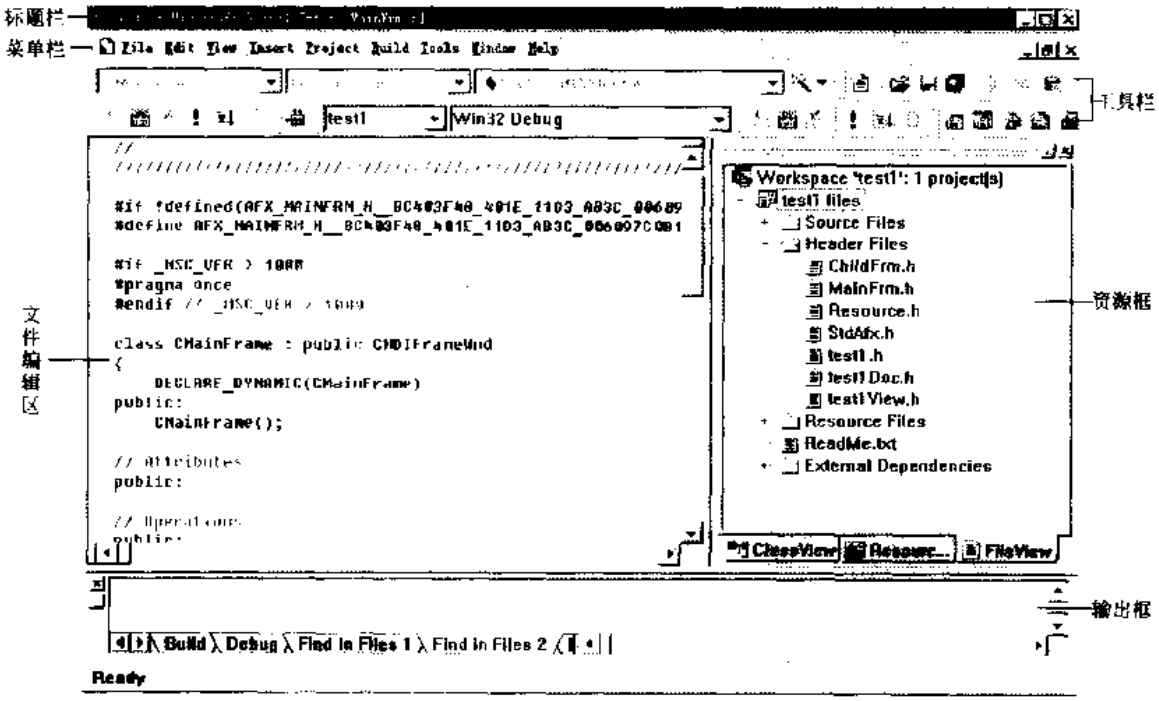


图 2-1 Visual C++ 6.0 开发环境

开发环境是程序员同 Visual C++ 的交互界面，通过它程序员可以访问 C++ 源代码编辑器、资源编辑器，使用内部调试器，并且可以创建工程文件。本章将介绍 Visual C++ 6.0 的用户界面，并对各种常用的窗口、菜单、按钮的意义和功能做较为详细的介绍，而对那些较少用到且只要求高级程序员掌握的内容则仅做简要的分析。

用鼠标单击“开始”|“程序”|Microsoft Visual Studio 6.0|Microsoft Visual C++ 6.0，就会显示如图 2-1 所示的窗口，图中标出了窗口中各组成部分的名称，而且显示了已装入 test1 工程文件的 Visual C++ 6.0 的开发环境，这是在建立了工程文件之后的结果。下面将重点介绍 Visual C++ 界面常用的组成部分。

2.2　Visual C++ 6.0 的菜单栏

在进行程序设计之前，先了解各个菜单命令的基本功能是很有必要的，因为程序员与开发界面打交道的大部分操作是通过菜单栏中的命令来完成的。菜单栏由若干个菜单组成，每个菜单又由多个选项或子菜单构成。

此外，在窗口的不同地方单击鼠标右键可以弹出相应的快捷菜单，通过快捷菜单可以执行与所处环境相关的命令。若在工具栏上单击鼠标右键，则弹出如图 2-2 所示的快捷菜单，通过它可以增减工具栏上的工具。

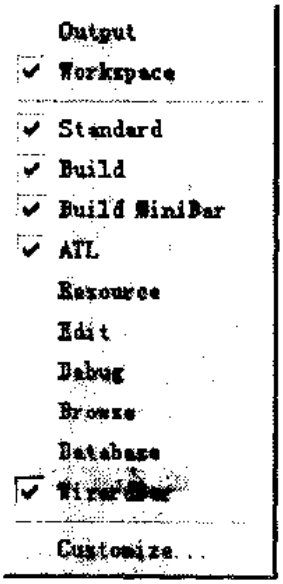


图 2-2　工具快捷菜单

2.2.1　File 菜单

File 菜单包括对文件进行操作的相关选项。

New 选项

选中该选项将打开 New 对话框，如图 2-3 所示。使用此对话框可以创建新的文件、项目、工作区或其他文档。

(1) Files 选项卡

通过该选项卡可以创建各种文件。只要先选中某种文件类型，再输入文件的名称就可以了。如果要将该文件添加到已有的项目中，只要选中 Add to project 复选框并且选择项目名即可。表 2-1 是 Visual C++ 6.0 可以创建的文件类型。

表 2-1　Visual C++ 6.0 可以创建的文件类型

文件类型	说　明	文件类型	说　明
Active Server Page	创建活动服务器	Binary File	创建二进制文件
Bitmap File	创建位图文件	C++ Source File	创建 C++ 源文件
C/C++ Header File	创建 C/C++ 头文件	Cursor File	创建光标文件
HTML Page	创建 HTML 文件	Icon File	创建图标文件
Macro File	创建宏文件	Resource Script	创建资源脚本文件
Resource Template	创建资源模板文件	SQL Script File	创建 SQL 脚本文件
Text File	创建文本文件		

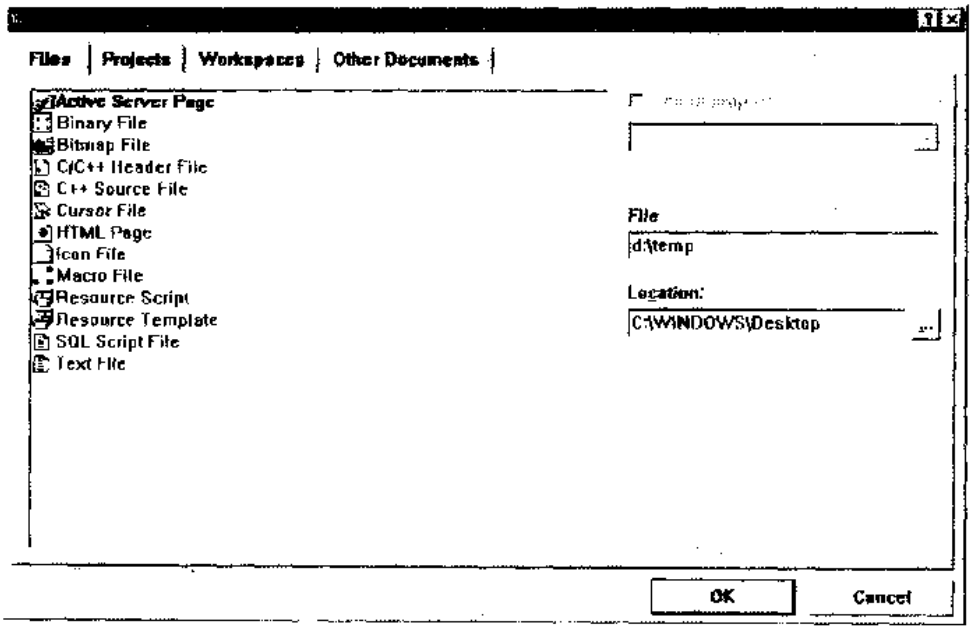


图 2-3 New 对话框中的 Files 选项卡

(2) Projects 选项卡

选中该选项卡可以创建各种新的 Visual C++ 工程文件。只要选择一种工程文件类型,此对话框就会提示用户输入工程文件的名称、存放位置以及程序员的平台(Platforms)类型(默认为 Win32),如图 2-4 所示。若要添加新项目到已打开的工作区中,请选中 Add to current workspace 单选按钮,否则 Visual C++ 将为你自动创建包含新项目的新工作区。如果要使新项目成为已有项目的子项目,请选中 Dependency of 复选框并指定项目名。

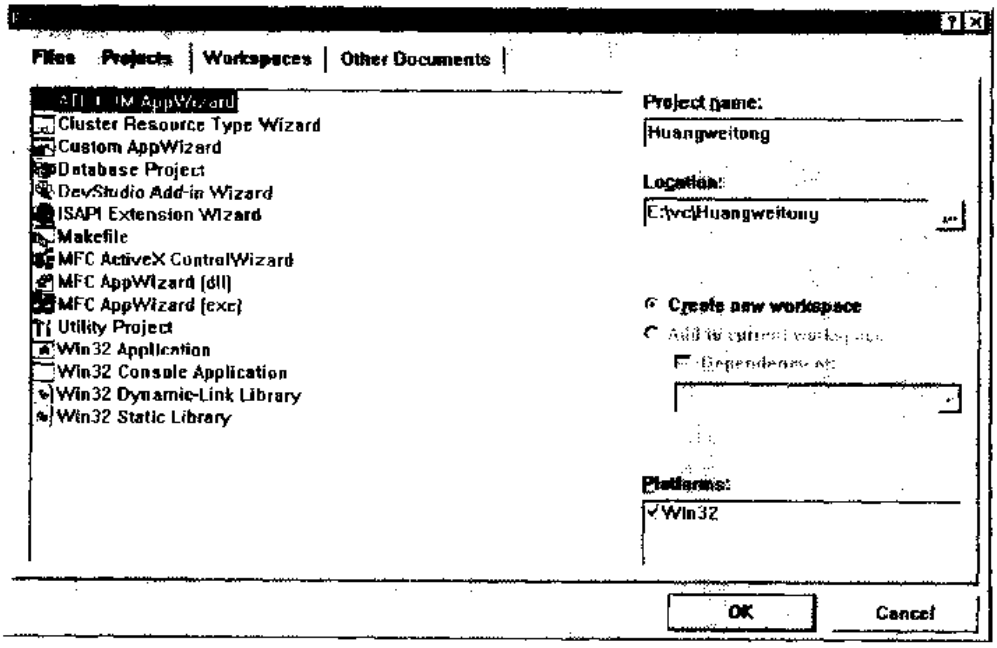


图 2-4 New 对话框中的 Projects 选项卡

表 2-2 是 Visual C++6.0 可以创建的项目类型。

表 2-2　Visual C++ 6.0 可以创建的项目类型

项目类型	说　明	项目类型	说　明
ATL COM AppWizard	创建 ATL 应用程序	Database Project	创建数据库项目
Win32 Dynamic-Link Library	创建 Win32 动态链接库	DevStudio Add-in Wizard	创建自动嵌入执行文件的宏
Custom AppWizard	创建自定义的 AppWizard	ISAPI Extension Wizard	创建 Internet 服务器或过滤器
Makefile	创建 Make 文件	MFC ActiveX ControlWizard	创建 ActiveX 控件程序
MFC AppWizard(dll)	创建 MFC 动态链接库	MFC AppWizard(exe)	创建 MFC 可执行程序
Win32 Application	创建 Win32 应用程序	Win32 Console Application	创建 Win32 控制台应用程序
Cluster Resource Type Wizard	通过它可以创建两种项目类型(Resource DLL 和 Cluster Administrator Extension DLL)	Utility Project	通过它可以创建一个项目,该项目只作为其他子项目的一个包容器,从而减少子项目的编联时间;但它本身并不包含任何文件
Win32 Static Library	创建 Win32 静态库		

(3) Workspaces 选项卡

创建新的工作区。

(4) Other Documents 选项卡

创建新的文档。如果要将新文档添加到已有项目中,请选中 Add to project 复选框,然后选择项目名。

2.2.2　Edit 菜单

1. Breakpoints 选项

选中该选项将弹出 Breakpoints 对话框,如图 2-5 所示,用于设置、删除和查看断点。断点实际上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,必要的话可以修改、继续执行或中断执行。

在 Visual C++ 6.0 中,断点分为位置断点(Location)、数据断点(Data)、消息断点(Messages)三种类型。位置断点通常在源代码的指定行、函数的开始或指定的内存地址处设置。当程序执行到指定位置时,位置断点将中断程序的执行。在位置断点中还可以使用图 2-5 中的 Condition 按钮设置断点条件。使用该功能时,必须先指定位置,然后再设置断点条件,仅当指定条件为真时才中断程序的执行。数据断点是在某一变量或表达式上设置。当变量或表达式的值改变时,数据断点将中断程序的执行。消息断点是在窗口函数 WndProc 上设置。当接收到指定的消息时,消息断点将中断程序的执行。

2. List Members 选项

该选项可以用来适当减轻程序员输入源程序代码的负担。输入代码时,如果在变量名之后键入"."或"->",那么系统会自动列表显示所有有效的成员名,这时只要键入成

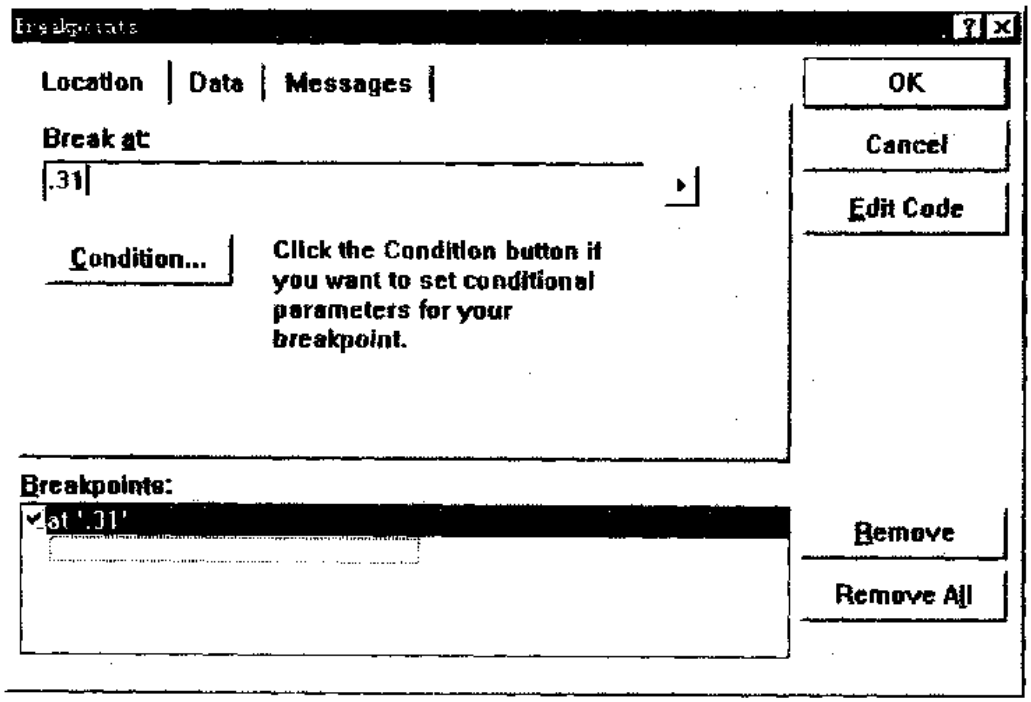


图 2-5 Breakpoints 对话框

员的前几个字母就可从列表中选中该成员，按 Tab 键即可完成输入；也可用滚动条找到待输入的成员名，然后用鼠标双击成员名来完成。

3. Type Info 选项

如果将鼠标指针停在源代码中某一变量或函数上，那么系统将显示所指变量或函数等的语法。

4. Parameter Info 选项

输入代码时，如果在输入函数名之后键入左括号，那么系统将显示该函数的完整原型，并用黑体显示其第一个参数。输入第一个参数值之后，接着就会出现第二个参数，依此类推。

5. Complete Word 选项

系统自动完成当前语句其余部分的输入。如果不能自动完成，则给出适当的提示辅助用户完成。

2.2.3 View 菜单

View 菜单中的命令主要用来改变窗口和工具栏的显示方式、检查源代码、激活调试时所用的各个窗口等。

1. ClassWizard 选项

选中该选项将弹出 MFC ClassWizard 对话框，如图 2-6 所示。ClassWizard 是一个适用于 MFC 应用程序的专用工具，该对话框又包括以下几个选项卡：

(1) Message Maps 选项卡

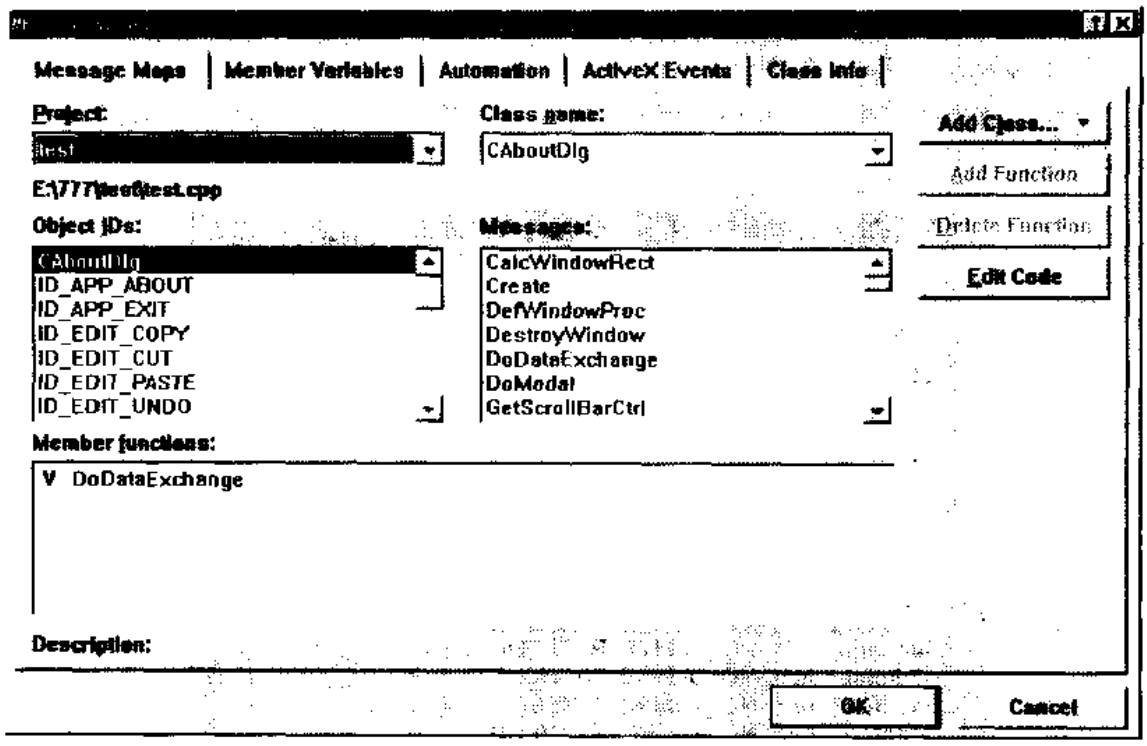


图 2-6　MFC ClassWizard 对话框

映像消息给与窗口、对话框、控件、菜单选项和加速键有关的处理函数，创建或删除消息处理函数，查看已经拥有消息处理函数的消息并跳转到相应的处理代码中去。

(2) Member Variables 选项卡

定义成员变量用于自动初始化、收集并验证输入到表单视图(Form View)中的数据，其中 Control IDs 是映像到成员变量的控件 ID 值，Type 是成员变量的类型，Member 是成员变量名。

(3) Automation 选项卡

创建新类时添加自动化方法和属性。其中 External names 列出的是已经添加到当前类中的自动化方法和属性的名称，Implementation 显示的是 External names 中的方法和属性是怎样实现的(S = stock property，C = custom，M = method，bold typeface = the default property)。

(4) ActiveX Events 选项卡

MSMQ 仅有两种事件(Event)：Arrived 和 ArrivedError，而且当消息(Message)或者错误(Error)到达消息队列时，MSMQEvent 就将这两种事件激活，而该对话框就是用来添加事件的。

(5) Class Info 选项卡

提供有关类的信息。

2. Resource Symbols 选项

该选项将打开资源符号浏览器，从中可以浏览和编辑资源符号。资源编辑符号是映像到整数值上的一串字符，相当于资源的一个代号，因此可以在源代码或资源编辑器中通

过资源符号引用资源。

3. Resource Includes 选项

选中该选项将弹出 Resource Includes 对话框，如图 2-7 所示。对话框中显示资源符号文件名和预处理指令。

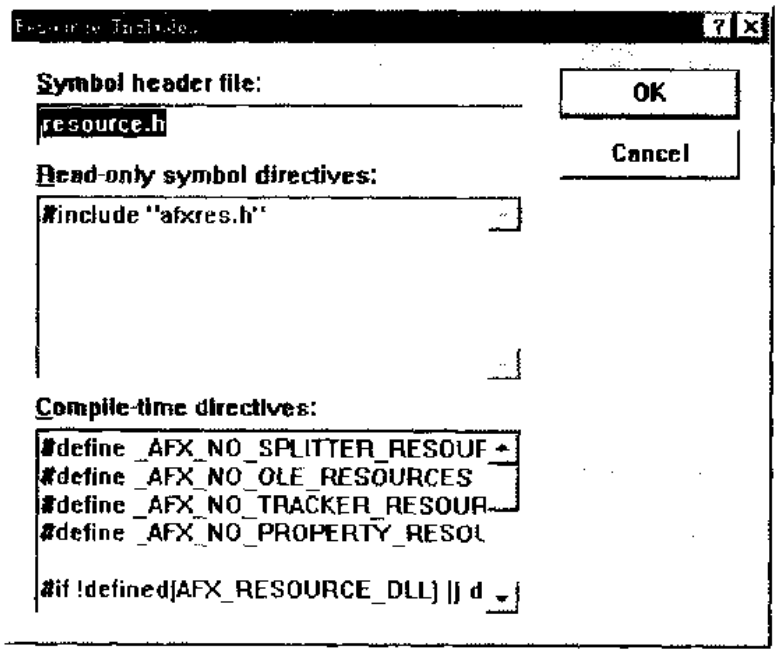


图 2-7 Resource Includes 对话框

系统默认将所有资源符号保存在文件 resource.h 中。如果同一目录中有多个资源文件，就必须改变系统默认的资源符号文件名。为此，可以在 Symbol header files 文本框中键入新的名字以保存资源文件的资源符号。

通常，只要在一个资源文件中保存所有的资源就足够了，但 Visual C++ 6.0 允许用户使用多个文件来保存资源，此时就必须用 Compile-time directives 文本框。此文本框中所列出的资源文件，可以是独立于主资源文件中的资源而创建及编辑的文件、含有编译指示的文件、含有用户定制资源的文件和标准的 MFC 资源文件等，它们在编译时将成为可执行文件的一部分。

4. Workspace 选项

如果工作区没有显示或者不是处于当前状态，选择该项将显示工作区窗口。

5. Output 选项

在输出窗口(Output)中显示程序建立过程(如编译、链接等)的有关信息，并且显示调试运行时的输出结果。

6. Debug Windows 选项

选择该选项将弹出级联子菜单，子菜单中的命令只有在调试运行状态才可用。下面就来分别介绍该子菜单中的各个选项。

(1) Watch 选项

选择该选项，系统将弹出 Watch 窗口，其中有 Watch1、Watch2、Watch3 和 Watch4 四个选项卡，在每个选项卡中有一系列用户指定的变量或表达式值。为了查看方便，在一个选项卡中可以显示一组相关的变量或表达式的值。

在 Watch 窗口中，系统自动为用户保留一个空的矩形框供用户添加变量或表达式用添加一个新的变量或表达式的步骤如下：

① 选中窗口中某个选项卡的空框。

② 单击左边的 Name 域，输入变量或表达式。

③ 按回车键，相应的值自动出现在右边的 Value 域中，同时，出现一个新的空矩形框

(2) Call Stack 选项

选择该选项将弹出 Call Stack 窗口，其中可以显示函数调用的参数类型及数值，相关的设置可以在 Tools 菜单下的 Options 对话框中进行，从 Debug 选项卡中得到信息。

如果用户要查看一个函数调用的行为，可以进行如下步骤：

① 在该函数的适当位置上设置一个断点。

② 运行 Debug 菜单下的 Run to Cursor 命令。

③ 从 View 菜单下选用 Call Stack 命令。

于是，各个函数调用按调用的先后次序排在 Call Stack 窗口中，排在第一位的是当前调用的函数(或者最内层嵌套的函数)。

(3) Memory 选项

选择该选项将弹出 Memory 窗口，该窗口用于显示自指定的任何地址处开始的内存内容。其中有从某一地址开始的所有地址空间(程序中所用到的)内的数值。起始地址在 Address 文本框内可以任意输入，输入后按回车键便可显示从该地址开始的内存中的内容。

若要改变内存数据的显示格式，右击该窗口，在快捷菜单中选择 Byte Format 以单字节形式显示；Short Hex Format 以两个字节形式显示；Long Hex Format 以四个字节形式显示。

(4) Variables 选项

选择该选项将弹出 Variables 窗口，该窗口用于快速访问程序中的一些变量。其中有三个选项卡：

- Auto 选项卡：显示当前语句及上一语句使用的变量，它还显示使用 Step Over 及 Step Out 处的返回值。
- Locals 选项卡：显示当前函数中的局部变量。
- This 选项卡：显示由 This 指针所指向的对象。

某个选项卡对应的页面内均有变量名称和数值两个域，调试器自动调用它们。若要查看其中的信息，只要程序运行到断点处终止时，打开该窗口即可。

(5) Registers 选项

选择该选项，Registers 窗口就会出现。该窗口在 Visual C++ 6.0 的调试器中用于显示 CPU 寄存器及浮点型堆栈的内容，同时，用户还可以根据需要改变任何寄存器的内容。若要改变其中数值的大小，将光标放在该值上输入新数值就行了。

(6) Disassembly 选项

选择该选项将弹出 Disassembly 窗口。默认情况下显示程序的汇编代码及附加源代码的标注信息。

在源代码中,每行代码作为一个整体来对待,执行过程中不能从一行中的一条语句跳至另一条语句。在 Disassembly 窗口中,程序以汇编语言的形式出现,一行为一条指令,用户可以在任何一条指令上设置断点。单行执行命令一次只执行一条命令,即一行代码。

2.2.4 Insert 菜单

1. New Class 选项

选中该选项将打开 New Class 对话框,如图 2-8 所示。该对话框用来创建新类并添加到项目中。

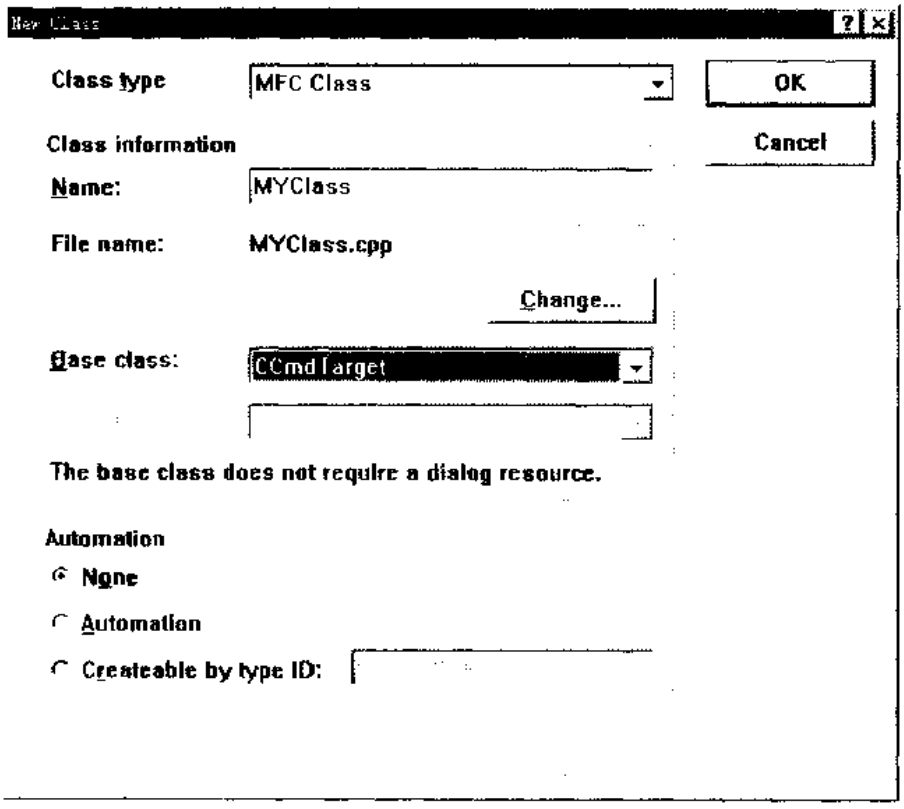


图 2-8 New Class 对话框

2. New Form 选项

选中该选项将打开 New Form 对话框,以便创建新的表单并添加到项目中,该对话框的形式与 New Class 对话框非常相似。

3. Resource 选项

选中该选项将打开 Insert Resource 对话框,以便创建新的资源或插入到资源文件中

4. Resource Copy 选项

要选中该选项必须先选中某个资源，否则呈灰色。该选项用于复制选定的资源。

5. New ATL Object 选项

该选项将打开 ATL Object Wizard 对话框，如图 2-9 所示。以便添加新的 ATL 对象到项目中。

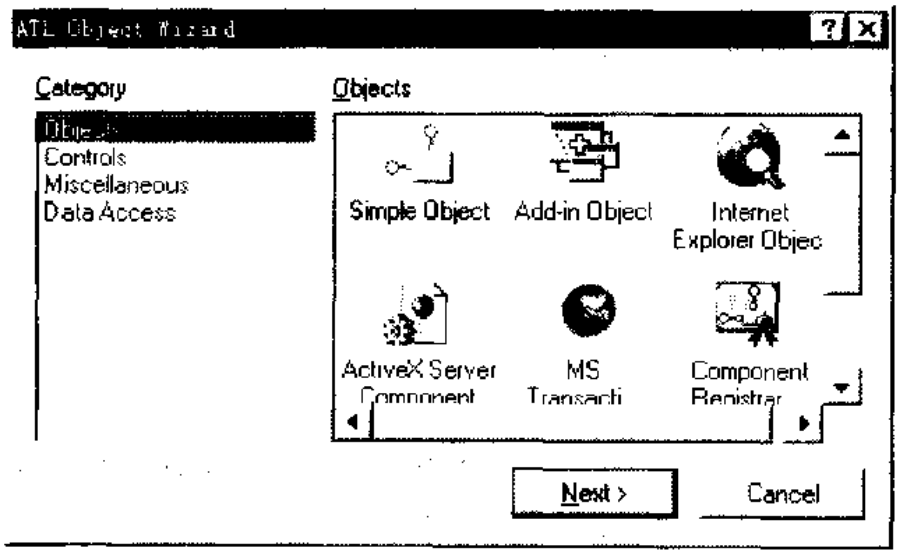


图 2-9　ATL Object Wizard 对话框

2.2.5　Project 菜单

使用 Project 菜单可以创建、修改和存储正在编辑的工程文件。

什么是工程文件呢？工程文件是一种机制，它组合了一个应用程序的所有源文件的组成部分(应用程序可以是 Windows 程序、DLL，也可以是 LIB 文件)。工程文件实际上包含在有 .MAK 扩展名的文件中，并非所有的 .MAK 文件都是 Visual C++ 工程文件。使用 AppWizard 是创建工程文件的一种方法，它同时可以创建与 Visual C++ 兼容的工程文件，或者只能用 NMake 使用的工程文件。

下面说明 Project 菜单的选项。

1. Select Active Project 选项

选择指定的项目为工作区的活动项目。

2. Add To Project 选项

选择该选项将打开一个级联子菜单，如图 2-10 所示。通过此子菜单，可以把文件、文件夹、数据链接以及 VC 组件或控件添加到当前项目中，在该子菜单中可以进行如下操作：

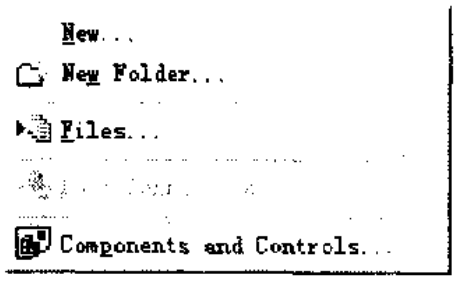


图 2-10　Add To Project 级联子菜单

(1) New：在工作区中创建新的文件，选中该项弹出的对话框基本同用 File | New 命令打开的对话框

相同

(2) New Folder:插入新的文件夹到当前项目中。

(3) Files:插入已有文件到当前项目中。

(4) Data Connection:添加数据链接到当前项目中。

(5) Components and Controls:选择该项将弹出 Components and Controls Gallery 对话框,在此对话框中,可以插入 VC 组件或者已注册的 ActiveX 控件。插入时相当于插入相关的头文件(.h 文件)和实现文件(.cpp 文件),并且更新工作区窗口的信息。

3. Settings 选项

选择该选项将弹出 Project Settings 对话框,可以对项目进行一系列的配置。其中,General 选项卡(如图 2-11 所示)用来设置基础类用法和目录。

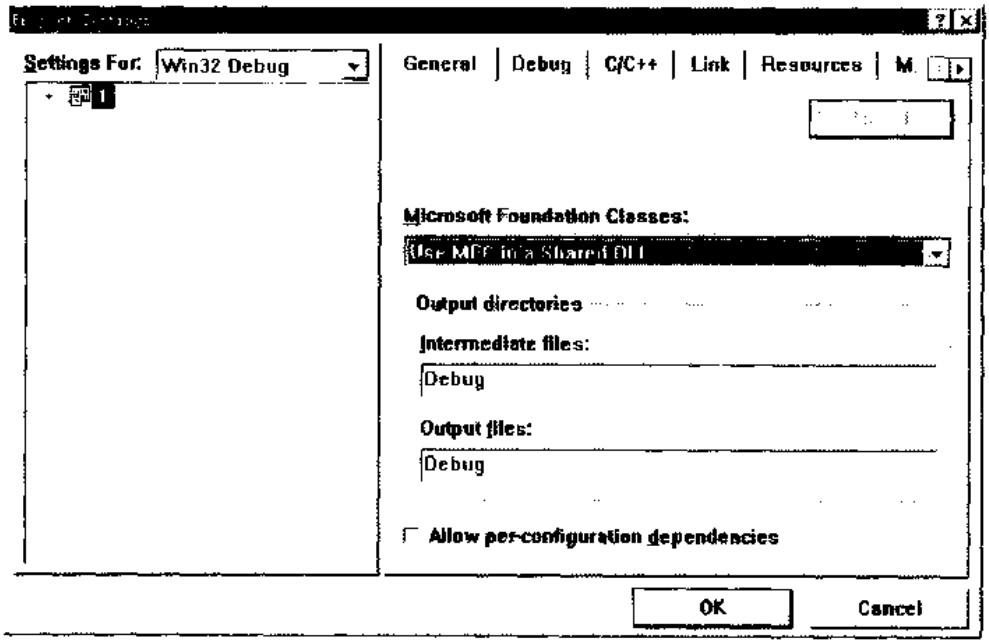


图 2-11 Project Settings 对话框

- Microsoft Foundation Classes:指出你的应用程序如何与 MFC 链接的。
- Intermediate files:指出用来存放中间文件(如对象和资源文件)的位置(默认为 Debug)。
- Output files:指出最后的输出文件的位置(默认为 Debug)。
- 如果要按外部制作文件(exported makefile)方式为每个项目配置导出不同的可建立项目,请选择 Allow per-configuration dependencies 复选框。

2.2.6 Build 菜单

Build 菜单用于编译、建立和执行应用程序。

1. Compile 选项

可以通过选择该选项让 Visual C++编译当前文件。这个菜单选项只编译当前文件,它不调用链接器或者任何其他的工具。如果当前文件是工程文件的资源脚本,将调用

Resource Compiler。编译过程如果检查出错误(如警告或错误信息),那么将在输出窗口中显示错误信息。为得到错误代码的位置,可以在错误信息处单击鼠标右键,然后在弹出的快捷菜单中选 Go To Error/Tag 选项,这样就能在源代码窗口中显示出有错的代码行。

2. Build 选项

由于一个 Windows 应用程序是由多个文件组成,而且这些文件可能由不同的程序员编写,并且用不同的编译器编译过,这就使一个应用程序相当复杂。用 Build 这个工具就可以使编译、链接这个应用程序所花的时间大大减少。此菜单选项查看所有的文件,只对最近修改过的源文件进行编译和链接。如果没有创建错误,其他的工程文件建立工具将被调用来创建最后的工程文件。

3. Rebuild All 选项

该选项与 Build 选项的唯一区别就是 Rebuild All 选项允许用户编译所有的源文件,而不管它们何时曾经被修改过。

4. Batch Build 选项

选择该选项将打开 Batch Build 对话框,如图 2-12 所示,能单步重新建立多个工程文件。在该对话框中,用户可以指定要建立的项目。在默认状态下,Visual C++ 提供了两种目标应用程序类型:Win32 Release(发行版)和 Win32 Debug(调试版)。如果在主应用程序之外工程文件还包括.DLL 文件或.LIB 文件,并且希望重新建立工程文件的所有部分,那么这个菜单选项是非常有用的。

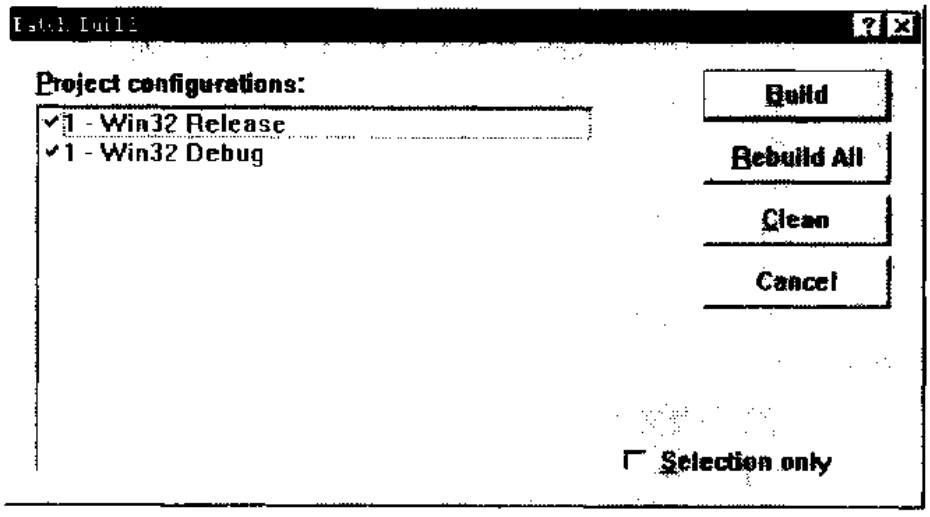


图 2-12　Batch Build 对话框

5. Clean 选项

删除项目的中间文件和输出文件。当遇到小项目占有大量磁盘空间时,这一功能是十分有用的。

6. Start Debug 选项

该选项将弹出子菜单,其中含有用于启动调试器运行的子选项:

(1) Go:从当前语句开始执行程序直到遇到断点或遇到程序结束。

(2) Step Into:单步执行程序,并在遇到函数调用时进入函数内部后再单步执行。

(3) Run To Cursor:调试运行程序时,使程序在运行到当前光标所在位置时停止,事实上,这相当于设置一个临时断点。

(4) Attach to Process:调试过程直接进入到正在运行的进程中。

启动调试器后,Debug 选项将代替 Build 选项出现在菜单栏中,使 Debug 选项可以控制程序的执行。

7. Debugger Remote Connection 选项

对远程调试链接设置进行编辑。

8. Set Active Configuration 选项

选择活动项目的配置,如 Win32 Release 和 Win32 Debug。

9. Profile 选项

该选项用于启动剖析器。剖析器(Profiler)是用于分析程序代码运行行为的强有力的工具。利用剖析器提供的信息,可以找出代码中哪些部分是高效的,哪些部分要更加仔细地加以检查。此外,利用剖析器还可以给出未执行代码区域的诊断信息。剖析器通常不是用于查错,而是用于使程序能更好地运行。例如,确定算法是否高效、函数是否被频繁使用、代码片断是否在测试过程中覆盖等等。

使用剖析器之前,必须通过 Project Settings 对话框的 Link 选项卡打开 Enable profiling。注意,打开 Enable profiling 将关闭 Link incrementally。如果要进行剖析,还必须包括调试信息。

2.2.7 Tools 菜单

Tools 菜单允许用户简单快速地访问多个不同的开发工具,如定制工具栏与菜单、激活常用的工具(如 Spy++ 等)或者更改选项等。

1. Source Browse 选项

默认情况下,在建立项目时,编译器会创建与项目中每一程序文件信息有关的.SBR 文件。实用程序 BSCMAKE 将汇编这些.SBR 文件为单个浏览信息数据库,浏览信息数据库的名字由项目基名加上扩展名.BSC 组成。

选择 Source Browse 选项将弹出 Browse 对话框,如图 2-13 所示。从中显示与程序中所有符号(类、函数、数据、宏和类型)有关的信息。对于不同类型的对象或上下文,Browse 对

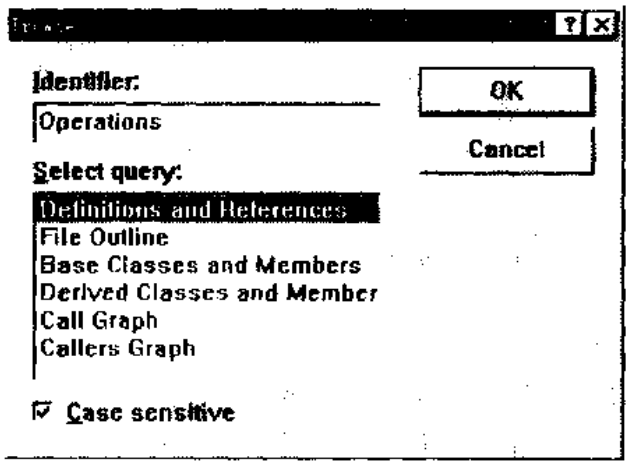


图 2-13 Browse 对话框

话框的外观和控制是不同的。

通常，使用 Browse 对话框主要可以检查一系列信息，如源文件中所有符号的信息、引用某个符号的所有源代码行、基类与成员之间的关系、派生类与成员之间的关系，以及调用的图形等信息。

默认时，系统在建立项目过程中自动创建浏览信息数据库。当然，为了加速项目的建立过程，可以关闭默认设置，需要时再打开。可以使用 Project Settings 对话框的 Browse Info 选项卡来设置是否在建立项目过程中创建浏览信息文件。

2. Visual Component Manager 选项

该工具用于组织、寻找和插入组件(Components)到某个 Visual Studio 项目中，而且它还可以识别已存在的且可利用的对象。选择该选项将弹出 Vcm.vbd 窗口，如图 2-14 所示。

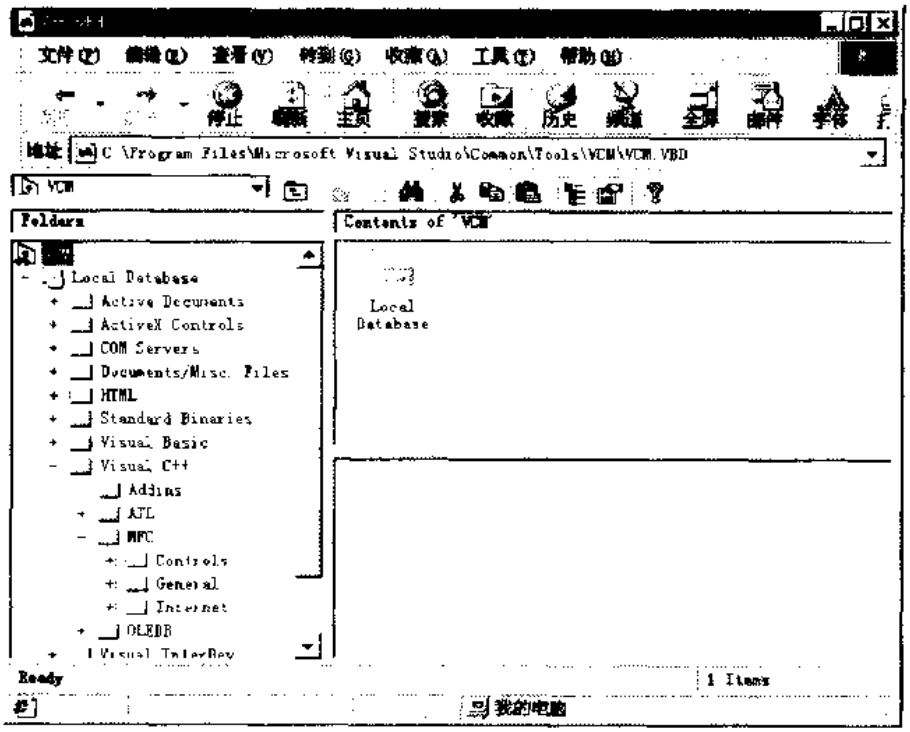


图 2-14　Vcm.vbd 窗口

3. Error Lookup 选项

该选项检查大多数 Win32 API 函数返回的标准错误代码信息。

4. OLE/COM Object Viewer 选项

选择该选项弹出的对话框如图 2-15 所示。它提供了安装在所有 OLE 以及 ActiveX 对象的信息。

5. MFC Tracer 选项

执行或调试程序时，该选项用于激活各种级别的调试消息，并由 MFC 将调试消息发送到输出窗口。

6. Spy++ 选项

Spy++ 是 Windows 可用的最有价值的工具之一。它不仅可以监视消息、进程、线程和

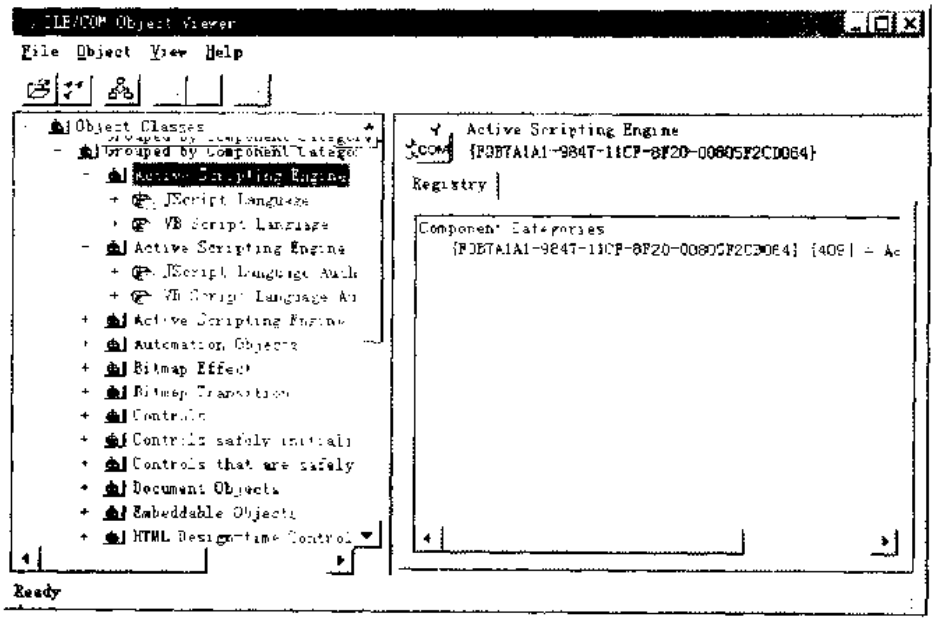


图 2-15 OLE/COM Object Viewer 对话框

窗口，而且还能看见这些元素之间的关系。例如，当监视一条消息时，可以在窗口中看见这条消息的作用。

7. Options 选项

选中该选项将打开 Options 对话框，以便对 Visual C++ 6.0 的环境设置（如调试器设置、窗口设置、目录设置、工作区设置、兼容性设置和格式设置等）进行更改。Options 对话框中有如下选项卡：

（1）Editor 选项卡：可以设定编辑窗口格式（滚动条、编辑模式、窗口再循环等）、存储选项，以及表述完成选项（Auto list members、Auto type info、Auto parameter info、Code comments 等 Edit 菜单中的几个选项的自动功能设置），如图 2-16 所示。

（2）Debug 选项卡：可以为 Disassembly、Registers、Call Stack 和 Memory 等编译选项配置窗口。还允许 Just-in-Time 调试模式和远程调试。

（3）Compatibility 选项卡：设置 Visual C++ 6.0 的兼容性。

（4）Build 选项卡：关于建立工程文件的设置。

（5）Directories 选项卡：可使用该选项卡为每个平台设定 Executable、Include、Include、Library 和 Resource 文件的默认目录。

（6）Workspace 选项卡：可在此选项卡中配置工作空间，如图 2-17 所示。Docking Views 列表框中列出了处于当前状态的窗口，用户可以设置这些窗口（Output、Watch、Locals、Registers、Memory、Call Stack、Disassembly 和 Workspace）的打开或关闭状态。

（7）Data View 选项卡：该选项卡可以设置用于 Microsoft SQL Server 和 Oracle databases 的查询（queries）和存储过程（stored procedures），如图 2-18 所示。

下面介绍一下该选项卡的各个选项：

- Limit SQL Output：当运行一个存储过程时，在输出窗口能够显示的最大数据行数

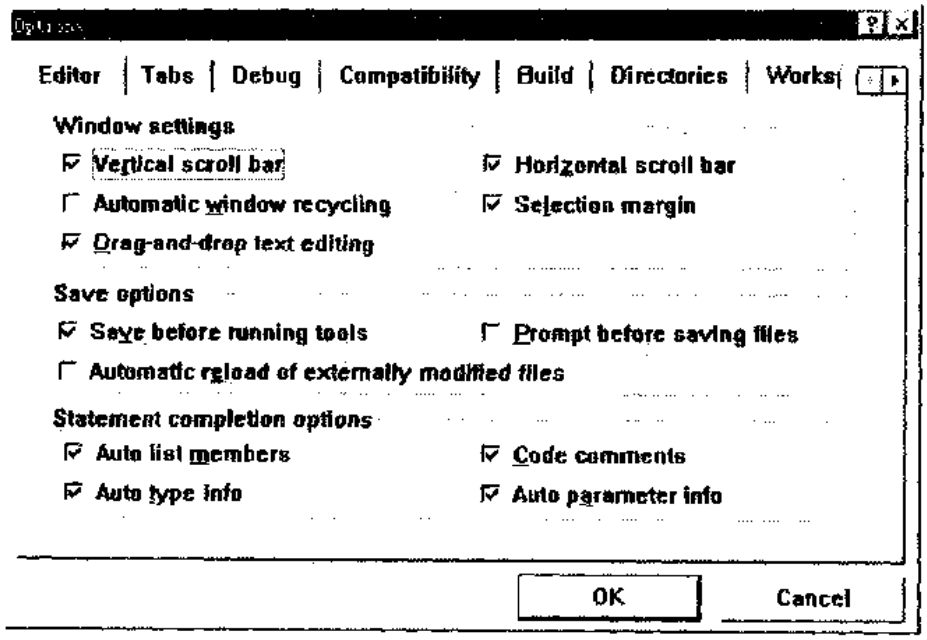


图 2-16　Options 对话框

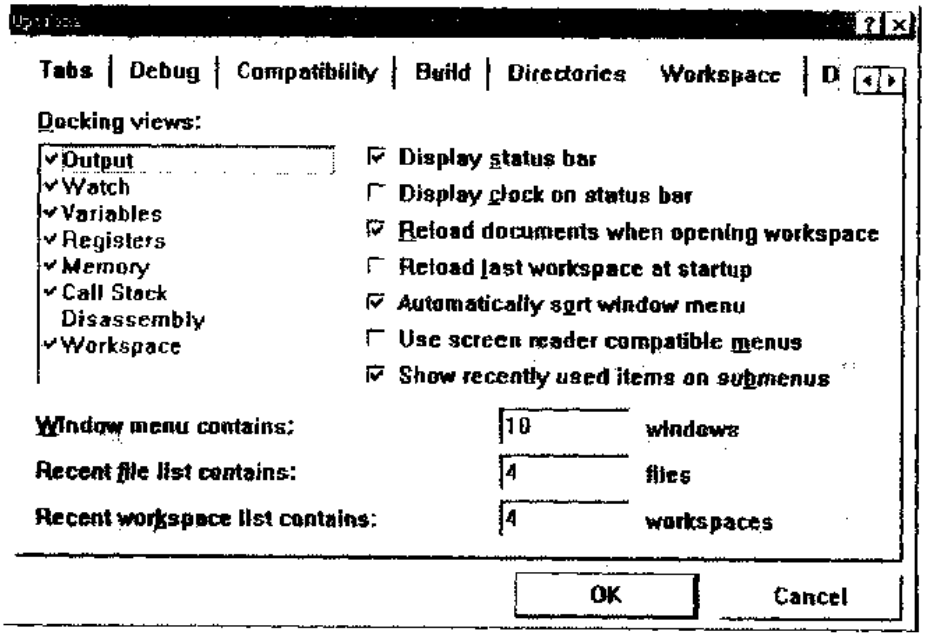


图 2-17　Workspace 选项卡

(默认是 500 行)。

- SQL Query Time-out：该数值代表进行一次 SQL 查询的超时时间(默认是 30 秒)。用户可以自己设置该项。
- SQL Login Time-out：系统为用户在一个 SQL Server 数据库中注册的超时时间(默认是 15 秒)。用户可以自己设置。
- View System Objects：如果要求数据视图显示为你的数据库所创造的系统对象(比

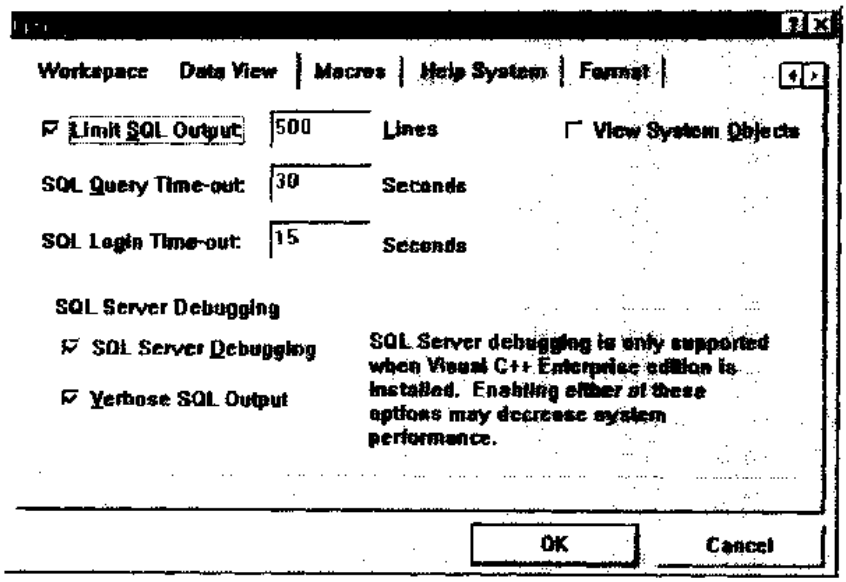


图 2-18 Data View 选项卡

如 Microsoft SQL Server 为存储某个 SQL Server 数据库属性而建立的"the sysproperties table"),请选择该复选框。

- SQL Server Debugging:如果想要拥有存储过程(stored procedures)的 Microsoft SQL Server 调试器,请选择该复选框(该特征只有 Visual C++ 5.0 企业版以上的版本才拥有)。
- Verbose SQL Output:如果希望在调试存储过程(stored procedures)期间,Microsoft SQL Server 驱动器发出的所有输出结果显示在输出窗口的 Debug 表中,请选择该复选框。

2.3 Visual C++ 6.0 的工具栏

工具栏是一种图形化的操作界面,具有直观和快捷的特点,熟练掌握工具栏的使用对提高编程效率非常有帮助。

工具栏由某些操作按钮组成,分别对应着某些菜单选项或命令的功能。用户可以直接用鼠标单击这些按钮来完成指定的功能。

2.3.1 工具栏的构成

Visual C++ 6.0 包含有十几种工具栏。默认时,屏幕工具栏区域显示有两个工具栏,即 Standard 工具栏和 Build MiniBar 工具栏,如图 2-19 所示。

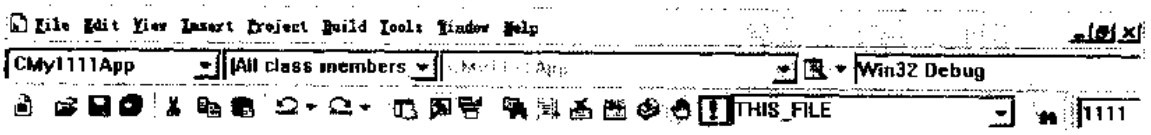


图 2-19 VC 中常用的工具栏

2.3.2　工具栏的定制与修改

用户可以根据自己的需要添加新的工具栏，或者对已有工具栏进行修改，如添加新按钮、删除已有按钮等。若需要添加工具栏，只需选择 Tools | Customize | New 选项，然后输入相应的名字即可。

2.4　项目与项目工作区

Visual C++ 6.0 以项目工作区的形式来组织文件、项目和项目配置。每个项目工作区由工作区目录中的项目工作区文件组成。项目工作区文件(.DSW)含有工作区的定义和项目中所包含文件的所有信息。

2.4.1　创建新的项目工作区

如果要创建新的空工作区，请按下列步骤进行：

(1) 选择 File 菜单后，选中 New 选项。

(2) 在打开的 New 对话框中选择 Workspaces 选项卡，如图 2-20 所示。

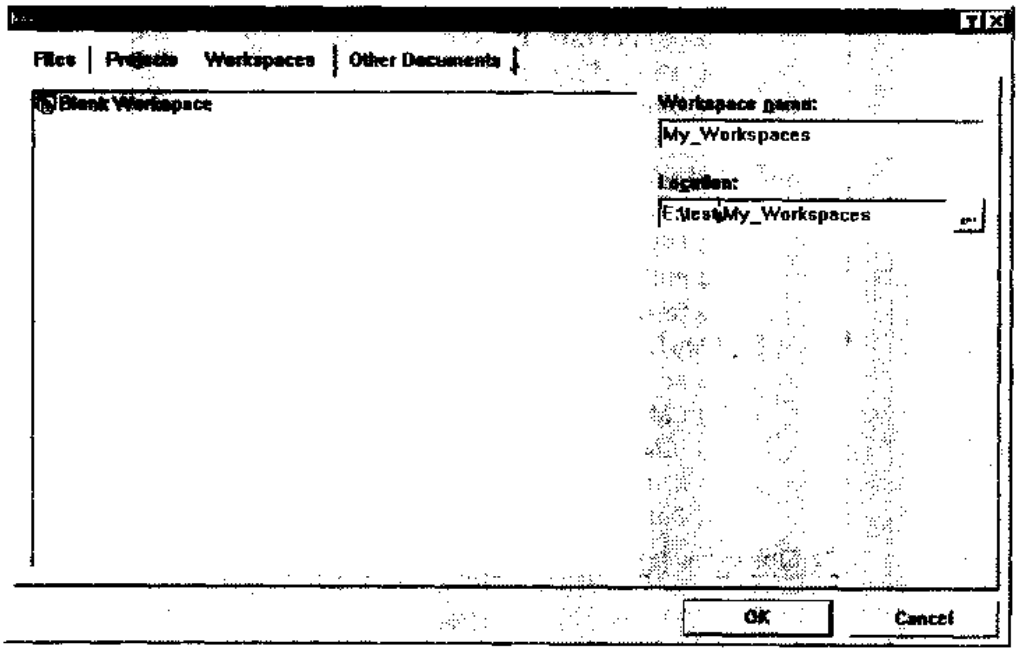


图 2-20　New 对话框的 Workspaces 选项卡

(3) 接着输入工作区名字(如 My_Workspaces)，并指定工作区目录(如 Test)，再单击 OK 按钮，这样一个新的空工作区就生成了。

(4) 可用 Open 选项在新的空工作区添加已经存在的项目。通过 New 对话框的 Projects 选项卡创建新的项目，并选中 Add to current workspace 复选框。

2.4.2 项目工作区窗口

Visual C++ 6.0项目工作区由 ClassView(类显示)面板、ResourceView(资源显示)面板及 FileView(文件显示)面板等组成。

项目工作区中,某个文件夹中各成分的关系表示的是项目中各组成部分的逻辑关系,而不是各组成部分的物理位置。双击文件夹中最低层的对象时,就会打开相应的编辑器,出现一个编辑窗口。例如,在 FileView 中双击文本文件,就会打开源程序编辑器;在 ResourceView中双击一个资源时,会打开资源编辑器。

下面分别介绍 ClassView、ResourceView 及 FileView 面板。

1. ClassView 面板

ClassView 面板列出了当前项目中的类(图 2-21),通过该面板可以浏览类的成员。每个成员的左边都有一个或多个图标,这些图标表示该成员是成员变量还是成员函数,以及成员的访问类型等。如果成员是保护或私有类型,那么它的左边会有一个钥匙图标。

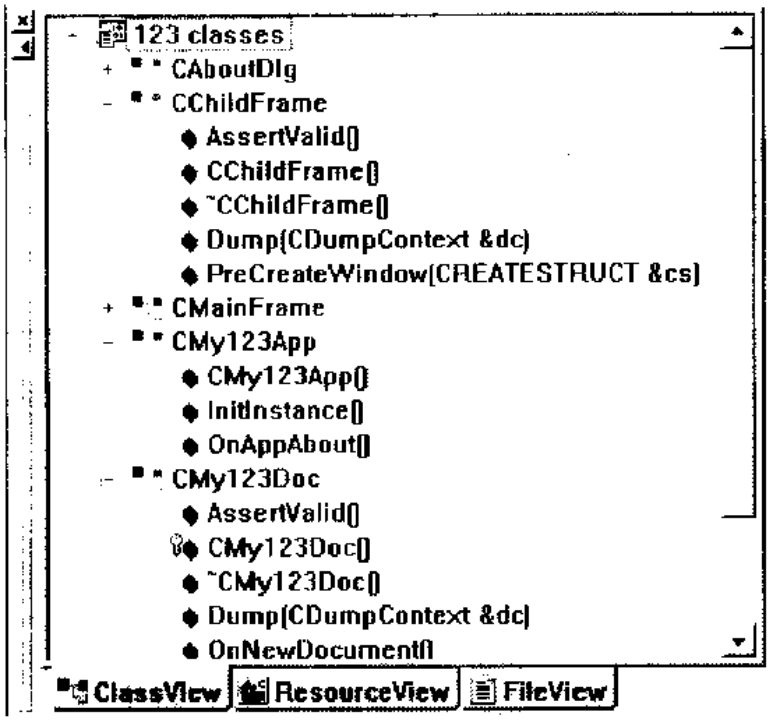


图 2-21 ClassView 面板

通过双击某个类或成员,可以在源代码窗口查看相应的源代码。双击一个类,将立即打开声明该类的头文件,且光标会停留在类的声明处;双击某个成员变量,光标将停留在该变量的声明处;双击某个成员函数,光标将停留在成员函数的定义或实现处。

2. ResourceView 面板

该面板拥有当前项目中包含的所有资源。扩展顶层文件夹可以显示资源类型,如图 2-22 所示。如有对话框资源 Dialog、图标资源 Icon 等。双击某个低层图标或者低层资源文件名就可以打开相应的资源编辑器。

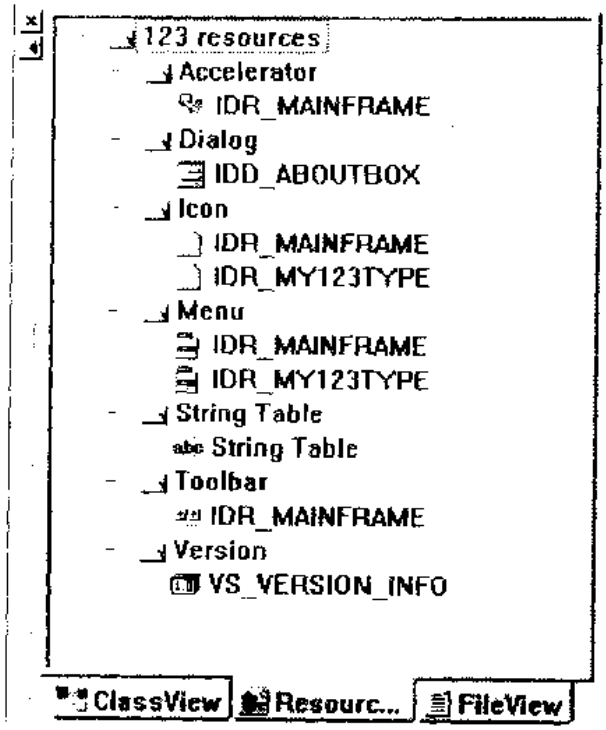


图 2-22　ResourceView 面板

3．FileVeiw 面板

FileView 面板用于显示当前项目中各项目之间的包含关系和项目中包含的所有文件。扩展顶层文件夹就可以显示项目的所有文件，如图2-23所示。双击某个文件名或图

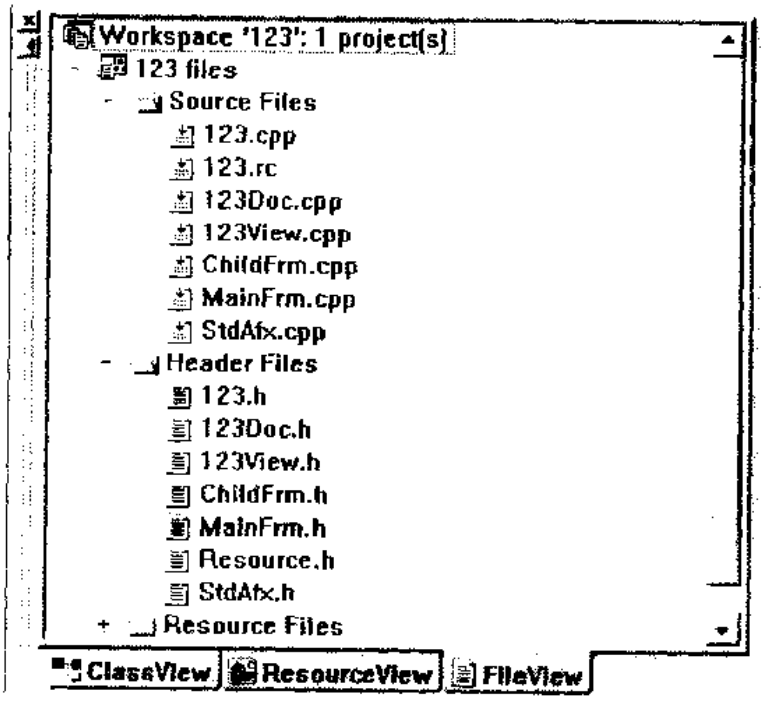


图 2-23　FileView 面板

标就可以打开相应的源程序编辑窗口。

2.5 资源与资源编辑器

进行 Windows 编程时，读者会遇到各种不同类型的资源，如图标、光标、菜单、位图、对话框等。在 Visual C++ 6.0 中，资源是一种界面成分，用户可以从中获得信息并在其中执行某个动作。但是，资源本身并没有定义界面功能，它仅是应用程序中元素的视角表现。如果应用程序中并未对某个选项指定任何动作，那么用户选择了这一命令时什么也不发生。所以，资源的界面功能必须由应用程序来定义。对于较早的 Visual C++ 对象，所有资源客体(对话框、菜单、字符串表等)不得不都被定义在资源描述文件 .rc 中，而且需要用特殊的编辑器和编译器来编辑资源并将它们链接到应用程序中去。Visual C++ 6.0 虽然依旧使用 .rc 文件，但却提供有强大功能的资源编辑器来帮助用户完成资源的创建和修改，并且可以自动将资源编译链接到应用程序中去。

在 Visual C++ 6.0 中，源代码或资源编辑器为了引用资源和用户自定义的其他对象，于是就引入了一种与资源一一对应的符号，从而使资源能够以字符串的方式被引用。

资源符号是映像到某个整数值上的一串字符，是一种标识符。每当创建一个新的资源或资源对象时，系统就会为其提供一个默认的资源符号名称并赋一个整数值。该定义保存在 resource.h 文件中，读者可以通过 View | Resource Symbol 来打开资源符号浏览器 Resource Symbols对话框，该对话框中列出了已被使用的资源符号的名称和相应的整数值。如果要显示只读状态的资源符号，请选中 Show read-only symbols 复选框，此时，未被使用的资源符号也一并显示出来，如图 2-24 所示。

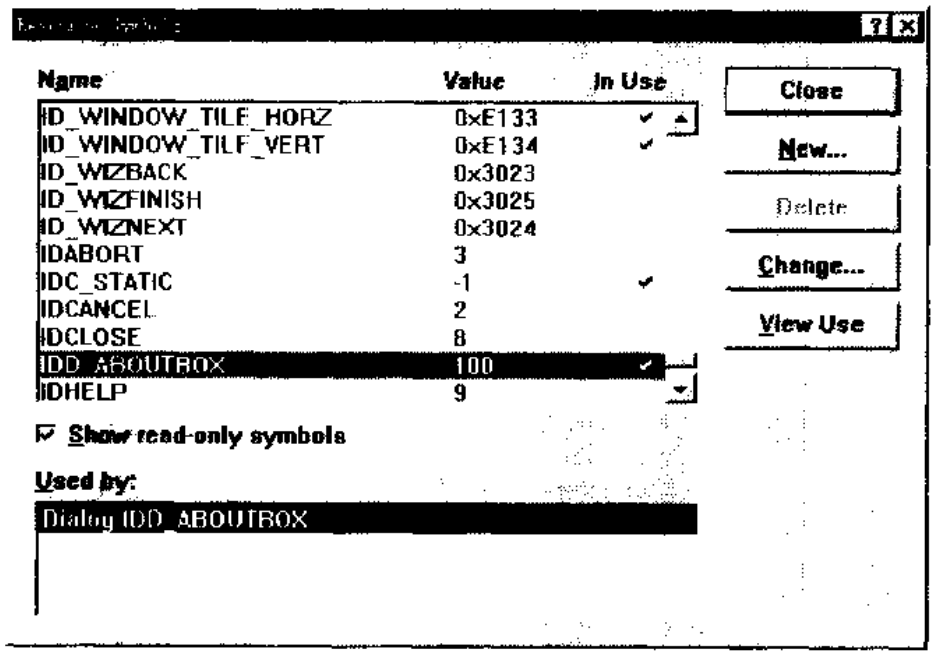


图 2-24 Resource Symbols 对话框

若要新建资源符号，则只要在图 2-24 中单击 New 按钮，在弹出的 New Symbols 对话框中的 Name 文本框中输入新资源符号的名称即可。由于在同一个应用程序中，资源符号不能重复。因此，程序员们都形成了习惯，按照一种标准来定义资源符号，表 2-3 是 Visual C++ 6.0 中提供的一些定义资源符号常用的前缀。

表 2-3　资源符号定义的常用前缀

类　别	资源符号前缀	代表的资源或对象
资源	IDR_	加速键或菜单及相关资源
	IDD_	对话框资源
	IDC_	光标资源
	IDI_	图标资源
	IDB_	位图资源
菜单项命令控制	IDM_	菜单项
	ID_	命令项
	IDC_	控件
字符串	IDS_	字符串表中的字符串
	IDP_	消息框中使用的字符串

资源符号的取值范围为 0 ~ 32767，超过此范围系统会提示出错信息。另外资源组件（如对话框控制）的标识符取值范围为 0 ~ 65535 或 - 32768 ~ 32767。若在图 2-24 中单击 View Use 按钮，系统就会自动进入该资源符号所对应的资源编辑器中。

2.6　联机帮助

Visual C++ 6.0 提供了详细的帮助信息，用户通过选择集成开发环境中的 Help 菜单下的 Contents 命令就可以进入帮助系统。在源文件编辑器中把光标定位在一个需要查询的单词处，然后按 F1 键也可以进入 Visual C++ 6.0 的帮助系统。

Visual C++ 6.0 的帮助系统很像微软的 Internet Explorer，确实该帮助系统也是一个很好的 HTML 浏览器，它不仅可以浏览以 HTML 文件组织成的静态帮助文件，而且帮助系统还提供了全文搜索的功能，如图 2-25 所示。当用户在搜索选项卡中输入所要搜索的单词时，帮助系统将在所有的帮助文件中进行全文搜索。同时，Visual C++ 6.0 的帮助系统还提供了很好的文件分类，通过"活动子集"下拉列表框，用户可以缩小全文搜索的范围。

Visual C++ 6.0 帮助系统中提供的帮助文件主要包括以下五个方面：

- Visual C++ Programmer's Guide：Visual C++ 程序员指南。
- Visual C++ User's Guide：Visual C++ 用户指南。
- Visual C++ Tutorials：Visual C++ 教程。
- Microsoft Foundation Class and Templates：微软基础类库和模板参考信息。
- Languages and Libraries for Visual C++：Visual C++ 语言和库文件信息。

用户通过 Visual C++ 6.0 的帮助系统可以获得几乎所有的 Visual C++ 6.0 的技术信息，这也是 Visual C++ 作为一个非常友好的开发环境所具有的一个非常有特色的

优点。

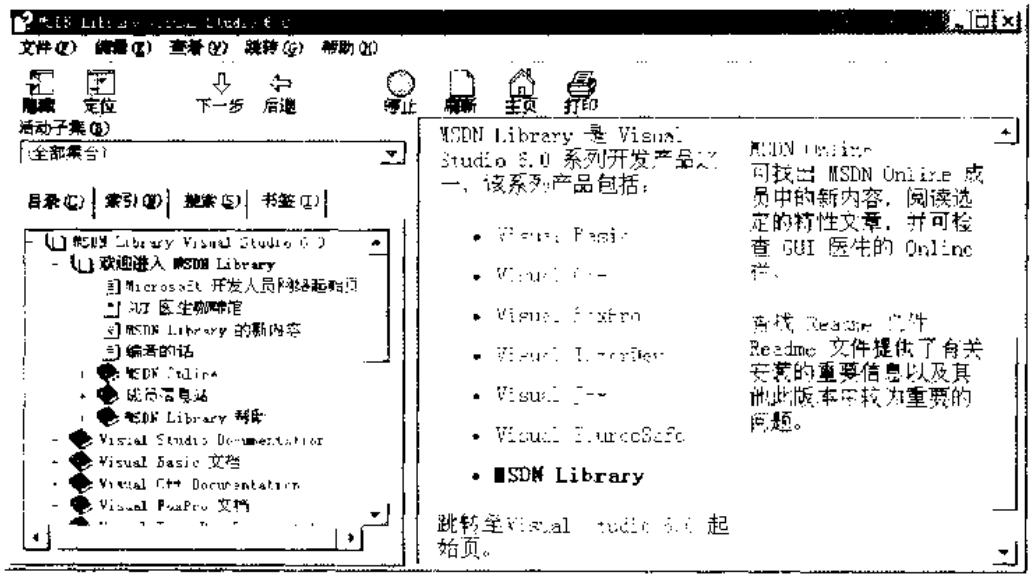


图 2-25　Visual C++ 6.0 的 Help 界面

习　　题

2-1　在 VC 环境中如何创建项目文件？

2-2　如何利用 VC 的资源编辑器？

2-3　如何充分利用 VC 提供的丰富的联机帮助功能？

第 3 章

Windows 编程基础

3.1 Windows 编程基础知识

Microsoft Windows 是一个应用于微型计算机上的具有图形化用户界面的操作系统，它为应用程序提供了一个由一致的窗口和菜单结构构成的多任务环境。

目前的 Windows 应用软件开发平台大多是“可视(Visual)”的，它往往是一个集成了下列系统的可用资源和开发工具的综合性开发平台：

- Windows 语言的源程序编辑器和编译器。
- 程序调试工具，包括源程序语法检查、可执行程序修改和运行监视等。
- 系统函数库和系统函数开发工具。
- 资源管理器，包括图形化窗口及其组成元素的多种对象的编辑器。
- 可选择并构成具体语句或源程序结构的例程库及帮助文件。
- 应用程序帮助文件和安装开发工具包。
- 其他功能。

Windows 的程序设计语言，包括 Visual C++、Visual Basic、Visual Java 等，都被称为“面向对象(object-oriented)”的程序设计语言。在 Windows 编程中，“对象(object)”是指 Windows 的规范部件，包括各种窗口、菜单、按钮、对话框及程序模块等，这些多样化的“对象”能够充分满足构成应用软件操作界面的需要，因此编写 Windows 程序的相当一部分工作是创建对象和为对象属性赋值。对象在具有规范形态的基础上还具有规范的操作模式，如能够对鼠标或键盘的规范操作产生规范的程序分支响应。用户可以采用传统的源程序代码编写方法编写程序，也可以采用 Windows 特色的交互式操作方法构造程序。采用交互式方法时，可视化开发平台给出了许多选用的对象，程序员可以选择所需的对象并为对象的属性确定参数值，由此搭建起应用程序的“大框架”。在这个“大框架”中，程序员根据需要进一步编写必要的细节代码段，最后构成完整的应用程序。

在 Windows 版本系列中，下列特点是始终坚持并不断发展的：

- 图形化的窗口界面。
- 多任务方式的运行环境。
- 虚拟化的设备接口，如图形设备接口(GDI)，它是与设备无关的图形化显示模式，

使多样化的图形硬件和软件设备都能够运行于 Windows。

- 以虚拟内存为核心的内存管理
- 网络功能及应用程序，包括 Microsoft 网络、通用基础网络协议和 Internet 网络等。
- 多媒体功能及应用程序，包括图形、图像、声音、动画和开发工具等。
- 功能丰富的用户管理工具和实用软件。

在用 Visual C++ 开发面向对象应用程序时，主要有两种方法，一种是使用 Windows 提供的 Windows API 函数，另一种方法是直接使用 Microsoft 提供的 MFC 类库。

API 是应用程序编程接口(Application Programming Interface)的缩写。Windows API 是 Windows 系统和 Windows 应用程序间的标准程序接口。API 为应用程序提供系统的各种特殊函数及数据结构定义，Windows 应用程序可以利用上千个标准 API 函数调用系统功能。

根据 Windows API 函数完成的功能，可将其分为三类：

- 窗口管理函数：实现窗口的创建、移动和修改功能。
- 图形设备(GDI)函数：实现与设备无关的图形操作功能。
- 系统服务函数：实现与操作系统有关的多种功能。

到目前为止，Microsoft 已经发布了几个 Windows API 版本，其中 WIN32 用于 Windows NT，Windows 98 的 API 接近于 WIN32。

MFC 类库集成了大量已经预先定义好的类，用户可以根据编程的需要调用相应的类，或根据需要自定义有关的类。本书将重点讲述 API 函数及 MFC 类库的应用，并通过一些实例来加深对它们的理解。

利用 Windows API 函数和 MFC 类库编写 Windows 应用程序与编写 DOS 应用程序有很大区别，掌握 Windows 编程方法必须首先了解以下内容：

- 窗口的概念。
- 事件驱动的概念。
- 消息及其在编程中的应用。
- 对象与句柄。
- 匈牙利表示法。

3.1.1 窗口

窗口是 Windows 应用程序基本的操作单元，是应用程序与用户之间交互的接口环境，也是系统管理应用程序的基本单位。一个基本的 Windows 应用程序窗口的组成如图 3-1 所示。编写一个 Windows 应用程序首先应创建一个或多个窗口，随后应用程序的运行过程即是窗口内部、窗口与窗口之间、窗口与系统之间进行数据处理与数据交换的过程。

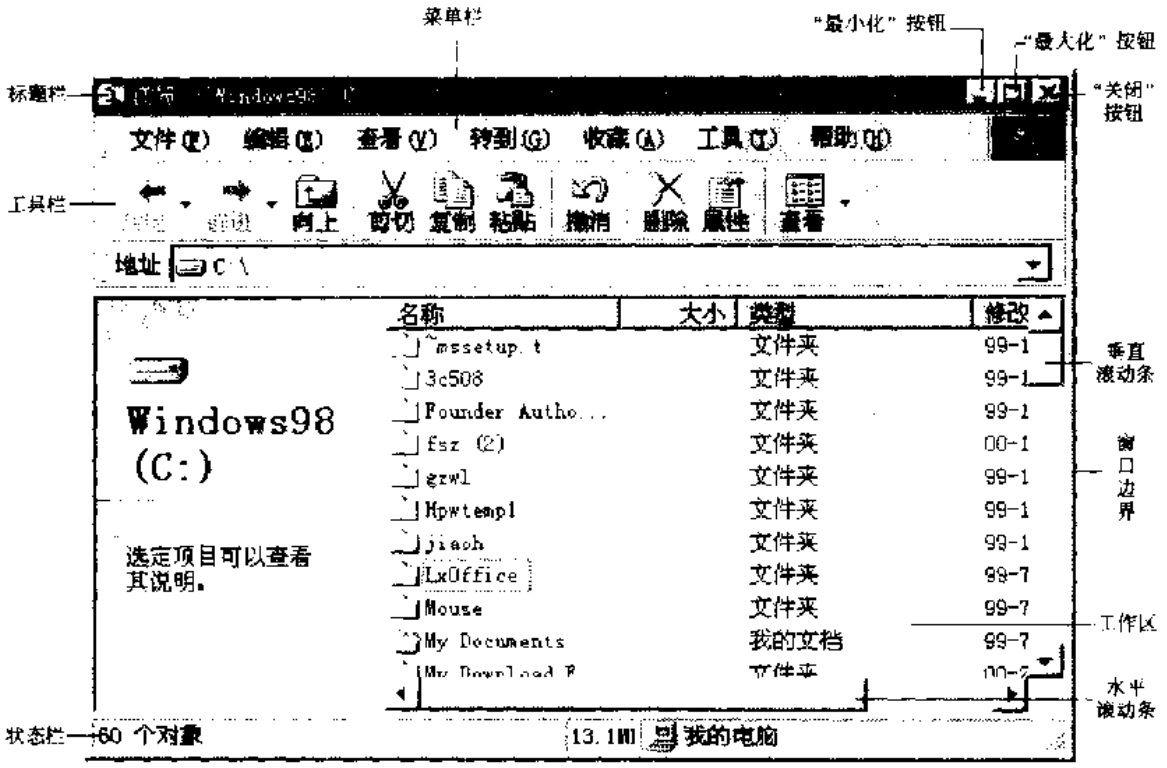


图 3-1　Windows 应用程序窗口的组成

3.1.2　事件驱动

Windows 程序设计围绕事件或消息的产生驱动运行处理函数(过程)。所谓消息是描述事件发生的信息。例如按下鼠标键时,系统就会产生一条特定的消息,标识鼠标按键事件的发生。Windows 程序的执行顺序取决于事件发生的顺序,程序的执行顺序是由顺序产生的消息驱动的,但是消息的产生往往并不要求有次序之分。程序员可以针对消息类型编写程序以处理接收的消息,或者发出其他消息以驱动其他程序,但是不必预先确定消息产生的次序。

事件驱动编程方法对于编写交互式程序很有用处,用这一方法编写的程序使程序避免了死板的操作模式,从而使用户能够按照自己的意愿采用灵活多变的操作形式。

3.1.3　句柄和 Windows 消息

1. 句柄

句柄(handle)是整个 Windows 编程的基础。一个句柄是指 Windows 使用的一个唯一的整数值,是一个 4 字节长的数值,用于标识应用程序中不同的对象和同类对象中不同的实例,诸如一个窗口、按钮、图标、滚动条、输出设备、控件或者文件等。应用程序通过句柄能够访问相应的对象信息。

在 Windows 应用程序中,句柄的使用是很频繁的,表 3-1 是部分常用句柄类型及其

说明。

表 3-1 常用句柄类型及其说明

句柄类型	说 明	句柄类型	说 明
HWND	标识窗口句柄	HDC	标识设备环境句柄
HINSTANCE	标识当前实例句柄	HBITMAP	标识位图句柄
HCURSOR	标识光标句柄	HICON	标识图标句柄
HFONT	标识字体句柄	HMENU	标识菜单句柄
HPEN	标识画笔句柄	HFILE	标识文件句柄
HBRUSH	标识画刷句柄		

Windows 应用程序利用 Windows 消息(Message)与其他的 Windows 应用程序及 Windows 系统进行信息交换。由于 Windows 应用程序是消息或事件驱动的,因此 Windows 消息的工作机制就显得很重要。Windows 中消息由三部分组成:消息号、字参数和长字参数。消息号由事先定义好的消息名标识;字参数(wParam)和长字参数(lParam)用于提供消息的附加信息,附加信息的含义与具体消息号的值有关。在 Windows 中,消息往往用一个结构体 MSG 来表示,结构体 MSG 的定义如下:

```
typedef struct tagMSG
{
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

其中:

- hwnd 是用以检索消息的窗口句柄,若此参数为 null,则可检索所有驻留在消息队列中的消息。
- message 是代表一个消息的消息值,每个 Windows 消息都有一个消息值,该值由 windows.h 头文件中的宏定义来标识。
- wParam 和 lParam 是包含有关消息的附加信息,它随不同的消息而有所不同。
- time 指定消息送至队列的时间。
- pt 指定消息发送时,屏幕光标的位置。pt 的数据类型 POINT 也是一个结构体,POINT 的定义如下:

```
typedef struct tagPOINT
{
    LONG x,y;                    // x 和 y 分别表示屏幕的横坐标和纵坐标
} POINT;
```

2. 消息

Visual C++中存在几种系统定义的消息分类,不同的前缀符号经常用于消息宏识别消息所属的分类,系统定义的消息宏前缀如下:

- BM 表示按钮控制消息。
- CB 表示组合框控制消息。
- DM 表示默认下压式按钮控制消息。
- EM 表示编辑控制消息。
- LB 表示列表框控制消息。
- SBM 表示滚动条控制消息。
- WM 表示窗口消息。

Windows 编程中常用的消息有：窗口管理消息、初始化消息、输入消息、系统消息、剪贴板消息、控制处理消息、控制通知消息、滚动条通知消息、非用户区消息、MDI(多文档界面)消息、DDE(动态数据交换)消息，以及应用程序自定义的消息等。

应用程序自定义的消息可以供内部应用程序和系统内其他进程通信使用。不同类型的 Windows 消息的取值范围如表 3-2 所示。

表 3-2　不同 Windows 消息类型的取值范围

消息类型	取值范围
系统定义消息(部分 I)	0x0000 ~ 0x03FF
用户定义内部消息	0x0400 ~ 0x07FF
系统定义消息(部分 II)	0x8000 ~ 0xBFFF
用户定义外部消息	0xC000 ~ 0xFFFF

3.2　Windows 应用程序常用消息

1. WM_LBUTTONDOWN

单击鼠标左键时产生此消息，其附加消息参数 wParam 包含一个整数值，标识鼠标键的按下状态。鼠标状态参数名称及其说明如表 3-3 所示；长参数 lParam 的低字节包含当前光标的 X 坐标，高字节包含当前光标的 Y 坐标。

表 3-3　鼠标键状态参数及其说明

鼠标键状态参数	说　明
MK_LBUTTON	标识按下鼠标左键
MK_MBUTTON	标识按下鼠标中键
MK_RBUTTON	标识按下鼠标右键

此外，相似的消息还有：

- WM_LBUTTONUP：放开鼠标左键时产生。
- WM_RBUTTONDOWN：单击鼠标右键时产生。
- WM_RBUTTONUP：放开鼠标右键时产生。
- WM_LBUTTONDBLCLK：双击鼠标左键时产生。
- WM_RBUTTONDBLCLK：双击鼠标右键时产生。

2. WM_KEYDOWN

这是在按下一个非系统键时产生的消息。系统键是指实现系统操作的组合键，例如 Alt 与某个功能键的组合以实现系统菜单操作等。其附加消息参数 wParam 为按下键的虚拟键码，虚拟键码用以标识按下或释放的键，例如功能键 F1 的虚拟键码在 Windows.h 文件中定义为 VK_F1，lParam 记录了按键的重复次数、扫描码、转移代码、先前键的状态等信息。此外相似的消息还有 WM_KEYUP，它是放开非系统键时产生。

3. WM_CHAR

这也是按下一个非系统键时产生的消息。附加信息参数 wParam 为按键的 ASCII 码，lParam 与 WM_KEYDOWN 中的 lParam 的意义相同。

4. WM_CREATE

此消息是由 CreateWindow 函数发出的消息。附加信息参数 wParam 未用，lParam 包含一个指向 CREATESTRUCT 数据结构的指针，该结构是传递给 CreateWindow 函数的参数的副本。

5. WM_CLOSE

关闭窗口时产生此消息。附加信息参数 wParam 和 lParam 均未用。

6. WM_DESTROY

消除窗口时由 DestroyWindow 函数发出此消息。附加信息参数 wParam 和 lParam 均未用。

7. WM_QUIT

这是退出应用程序时由 PostQuitMessage 函数发出的消息。附加信息参数 wParam 含有退出代码，退出代码标识应用程序退出运行时的有关信息；附加信息参数 lParam 未用。

8. WM_PAINT

当发生用户区移动或显示事件、用户窗口改变大小的事件、程序通过滚动条滚动窗口时，均产生一条 WM_PAINT 消息。此外，当下拉式菜单关闭并需要恢复被覆盖的部分以及 Windows 清除对话框或消息框等对象，并需要恢复被覆盖的部分时，将产生 WM_PAINT 消息。

3.3 Windows 中的事件驱动程序设计

基于 DOS 的应用程序主要使用顺序的、过程驱动的程序设计方法。顺序的、过程驱动的程序有一个明显的开始、明显的过程和一个明显的结束，因此，程序能直接控制程序事件或过程的顺序。

基于 Windows 的应用程序设计方法与 DOS 程序设计方法的不同就在于 Windows 程序是事件驱动的，事件驱动的程序不是由程序的顺序来控制，而是由事件的发生来控制。

假设有这样一个应用程序，该程序的功能是实现计算一个学期中进行了三次测验后一个班的平均成绩。在一个顺序的、过程驱动的程序中，可以设想用下面的步骤来实现该应用程序要求实现的功能。

(1) 输入学生姓名。

（2）输入第一次测验的分数。

（3）输入第二次测验的分数。

（4）输入第三次测验的分数。

（5）计算并显示平均分数

其逻辑流程图可以用图 3-2 来表示。

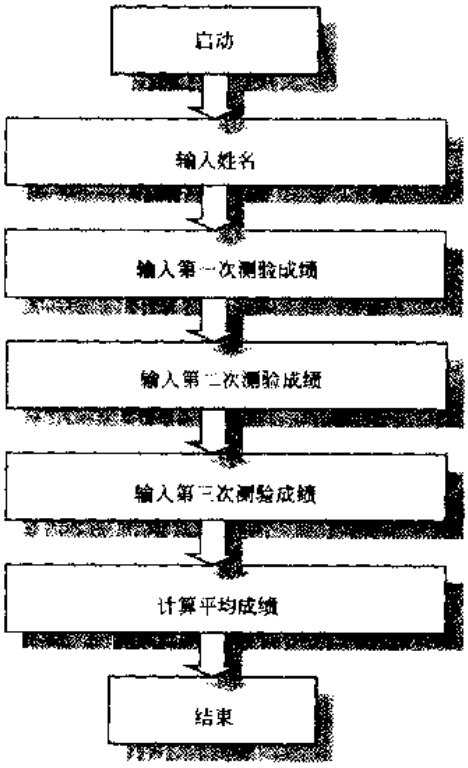


图 3-2　用过程驱动的方法来计算平均成绩

这种设计是基于过程驱动的，它使用户只能按照程序规定好的步骤进行操作。尽管在顺序的、过程驱动的程序中也有许多处理异常的方法，但是这些异常处理也是顺序的、过程驱动的结构。

事件驱动程序设计是围绕着消息的产生与处理而展开的。一条消息是关于发生的事件的信息。作为一个 Windows 程序员，其工作就是对正开发的应用程序所要发出或要接收的消息进行排序和管理。由于 Windows 消息是事件驱动的，消息是不会以任何预定义顺序出现的。图 3-3 是基于事件驱动的计算学生平均成绩的程序流程示意图。

按图 3-3 流程所设计的基于事件驱动的程序所实现的功能和用图 3-2 中的流程所设计的基于过程的程序所实现的功能是相同的。但是，用户可以在不同的窗口中来回切换，并不需要按顺序按步骤地进行数据的输入。例如，可以直接进入输入第三次测验成绩的窗口来输入第三次测验成绩，而不必先输入第一、二次测验成绩。

事件驱动程序方法提供了许多便利，对于那些需要大范围的用户干预的应用程序来说，更显其优越性。

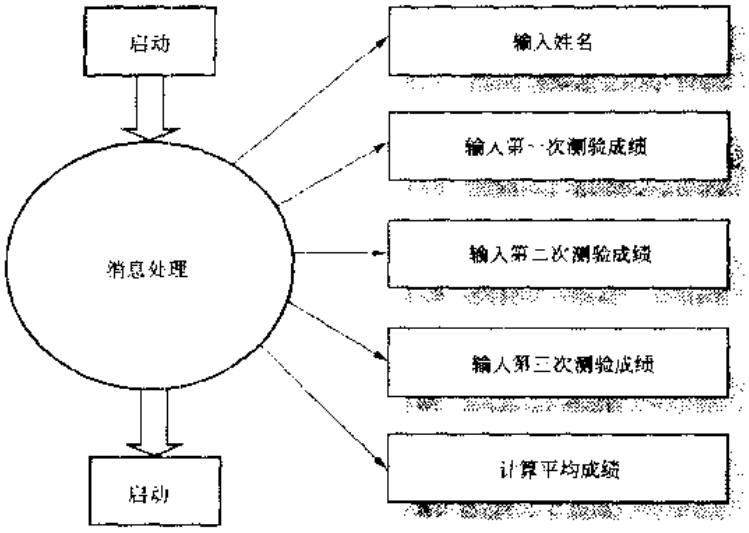


图 3-3　用事件驱动的方法来计算平均成绩

3.4　Windows 应用程序的基本结构

3.4.1　Windows 应用程序的组成

一个完整的 Windows 应用程序通常由表 3-4 所示五种类型的文件组成。

表 3-4　Windows 应用程序五种文件的扩展名及类型

扩展名	文件类型	备　注
.C 或 .CPP	C 语言源程序文件	
.H	头文件	头文件包含源程序文件需要的外部常量、变量、数据结构和函数定义及其说明
.DEF	模块定义文件	模块定义文件定义程序模块的属性，在 Windows95/98 中，该文件可省略
.RC	资源描述文件	资源描述文件定义源程序使用的资源
.MAK	工程文件	各种源程序文件编译后生成工程文件，经进一步编译成为可执行文件

3.4.2　源程序组成结构

Windows 的应用程序具有相对固定的基本结构，其中由入口函数 WinMain 和窗口函数构成基本框架，并包含各种数据类型、数据结构与函数等。入口函数 WinMain 和窗口函数是 Windows 应用程序的主体。

1. WinMain 函数

WinMain 函数是所有 Windows 应用程序的入口，类似 C 语言中的 Main 函数，其功能是完成一系列的定义和初始化工作，并产生消息循环。消息循环是整个程序运行的核心。

WinMain 函数实现以下功能：

- 注册窗口类，建立窗口及执行其他必要的初始化工作。
- 进入消息循环，根据从应用程序消息队列接收的消息，调用相应的处理过程。
- 当消息循环检索到 WM_QUIT 消息时终止程序运行。

WinMain 函数有三个基本的组成部分：函数说明、初始化和消息循环。

(1) 函数说明

WinMain 函数的说明如下：

```
int WINAPI WinMain
    (
    HINSTANCE hThisInst,   //应用程序当前实例句柄
    HINSTANCE hPrevInst,   //应用程序其他实例句柄
    LPSTR lpszCmdLine,     //指向程序命令行参数的指针
    int nCmdShow           //应用程序开始执行时窗口显示方式的整数值标识
    )
```

值得注意的是，Windows 应用程序可能并行地多次执行，因而可能出现同一个应用程序的多个窗口同时存在的情况，这也是 Windows 操作系统能进行多任务管理的特点。Windows 系统将应用程序每一次执行称为该应用程序的一个实例(instance)，并使用一个实例句柄来唯一地标识它。

(2) 初始化

初始化包括窗口类的定义、注册、创建窗口实例和显示窗口四部分。

① 窗口类定义

在 Windows 应用程序中，窗口类定义了窗口的形式与功能。窗口类定义通过给窗口类数据结构 WNDCLASS 赋值完成，该数据结构中包含窗口类的各种属性。窗口类定义常用以下函数：

- LoadIcon 函数

LoadIcon 函数的作用是在应用程序中加载一个窗口图标。其原型为：

```
HICON LoadIcon
    (
    HINSTANCE hInstance,   //图标资源所在的模块句柄；如为 NULL，则使用系统预定义图标
    LPCTSTR lpIconName     //图标资源名或系统预定义图标标识名
    )
```

- LoadCursor 函数

LoadCursor 函数的作用是在应用程序中加载一个窗口光标。其原型为：

```
HCURSOR LoadCursor
    (
    HINSTANCE hInstance,   //光标资源所在的模块句柄；如为 NULL，则使用系统预定义光标
    LPCTSTR lpCursorName   //光标资源名或系统预定义光标标识名
    )
```

- GetStockObject 函数

应用程序还经常调用函数 GetStockObject 获取系统提供的背景刷，其原型为：

```
HBRUSH GetStockObject(int nBrush);  // nBrush 为系统提供的背景刷的标识名
```

② 注册窗口类

Windows 系统本身提供部分预定义的窗口类，程序员也可以自定义窗口类，窗口类必须先注册后使用。窗口类的注册由函数 RegisterClass( )实现。其形式为：

```
RegisterClass(&wndclass);    //wndclass 为窗口类结构
```

RegisterClass 函数的返回值为布尔型，注册成功则返回值为真。

③ 创建窗口

创建一个窗口类的实例由函数 CreateWindow( )实现，该函数的原型为：

```
HWND CreateWindow
        (
        LPCTSTR lpszClassName,    //窗口类名
        LPCTSTR lpszTitle,        //窗口标题名
        DWORD dwStyle,            //创建窗口的样式，常用窗口样式如表 3-5 所示
        int X,                    //窗口左上角横坐标
        int y,                    //窗口左上角纵坐标
        int nWidth,               //窗口宽度
        int nHeight,              //窗口高度
        HWND hwndParent,          //该窗口的父窗口句柄
        HMENU nMenu,              //窗口主菜单句柄
        HINSTANCE hInstance,      //创建窗口的应用程序当前句柄
        LPVOID lpParam            //指向一个传递给窗口的参数值的指针
        )
```

表 3-5 常用窗口样式

标　识	说　明
WS_BORDER	创建一带边框的窗口
WS_CAPTION	创建一带标题栏的窗口
WS_HSCROLL	创建一带垂直滚动条的窗口
WS_MAXIMIZEBOX	创建一带最大化框的窗口
WS_MAXIMIZE	创建一最大化的窗口
WS_MINIMIZEBOX	创建一带最小化框的窗口
WS_MINIMIZE	创建一最小化的窗口
WS_OVERLAPPED	创建一带边框和标题的窗口
WS_OVERLAPPEDWINDOW	创建一带边框、标题栏、系统菜单及最大、最小化框的窗口
WS_POPUP	创建一弹出式窗口
WS_POPUPWINDOW	创建一带边框和系统菜单的弹出式窗口
WS_SYSMENU	创建一带系统菜单的窗口
WS_VSCROLL	创建一带水平滚动条的窗口

在实际的应用中，可定义组合式的窗口样式，例如下面的语句表示带有垂直和水平滚动条的弹出式窗口：

WM_HSCROLL|WM_VSCROLL|WM_POPUP

④ 显示窗口

窗口类的显示由 ShowWindow 和 UpdateWindow 函数实现。应用程序调用 ShowWindow 函数在屏幕上显示窗口，其形式为：

```
ShowWindow(hwnd,nCmdShow);
```

其中，hwnd 为窗口句柄，nCmdShow 为窗口显示形式标识，表 3-6 列出了常用显示形式标识及其说明。

表 3-6　常用显示形式标识及其说明

标　识	说　明
SW_HIDE	隐藏窗口
SW_SHOWNORMAL	显示并激活窗口
SW_SHOWMINIMIZE	显示并最小化窗口
SW_SHOWMAXIMIZE	显示并最大化窗口
SW_SHOWNOACTIVE	显示但不激活窗口
SW_RESTORE	恢复窗口的原来位置及尺寸

显示窗口后，应用程序常常调用 UpdateWindow 函数更新并绘制用户区，并发出 WM_PAINT 消息。其形式为：

```
UpdateWindow(hwnd);
```

(3) 消息循环

Windows 应用程序的运行以消息为核心。Windows 将产生的消息放入应用程序的消息队列中，而应用程序的 WinMain 函数从消息循环提取队列中的消息，并将其传递给窗口函数的相应过程处理。

消息循环的常见格式如下：

```
MSG Msg;
...
while(GetMessage(&Msg,NULL,0,0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

其中，函数 GetMessage 的作用是从消息队列中读取一条消息，并将消息放在一个 MSG 结构中。其形式为：

```
GetMessage
    (
    lpMSG,              //指向 MSG 结构的指针
    hwnd,
    nMsgFilterMin,      //用于消息过滤的最小消息号值
    nMsgFilterMax       //用于消息过滤的最大消息号值
    )
```

通过设置参数 nMsgFilterMin 和 nMsgFilterMax 可实现消息的过滤，即仅处理所确定的消息号范围内的消息。如果两个参数都为 0，则不过滤消息。

TranslateMessage 函数负责将消息的虚拟键转换为字符信息，其形式为：

```
TranslateMessage(lpMSG);
```

DispatchMessage 函数将参数 lpMSG 指向的消息传送到指定窗口函数，其形式为：

```
DispatchMessage(lpMSG);
```

当 GetMessgae 函数返回零值，即检索到 WM_QUIT 消息时，程序将结束循环并退出。

2. 窗口函数

窗口函数定义了应用程序对接收到的不同消息的响应，其中包含了应用程序对各种可能接收到的消息的处理过程，是消息处理分支控制语句的集合。通常，窗口函数由一个或多个 switch 语句组成。每一条 case 语句对应一种消息，当应用程序接收到一个消息时，相应的 case 语句被激活并执行相应的响应程序模块。

窗口函数是应用程序处理接收到的消息的函数，其中包含了应用程序对各种可能接收到的消息的处理过程。

窗口函数的一般形式如下：

```
LRESULT CALLBACK WndProc
    (
    HWND hwnd,                  //窗口句柄
    UINT message,               //代表所处理消息的消息值
    WPARAM wParam,              //消息的附加参数
    LPARAM lParam               //消息的附加参数
    )
{
...
switch(message)                 //  message 为标识消息的消息值
{
    case …
        ...
        break;
    …
    case WM_DESTROY:
        PostQuitMessage(0);
    default:
         return DefWindowProc(hwnd,message,wParam,lParam);
}
return(0);
}
```

窗口函数的主体是消息处理语句，该语句由一系列 case 语句组成。程序员只需根据窗口可能收到的消息在 case 语句中编写相应的处理程序段即可。

在 case 语句的消息处理程序段中一般都有对消息 WM_DESTROY 的处理。如前所述，

该消息是关闭窗口时发出的。一般情况下，应用程序调用函数 PostQuitMessage 响应这条消息。PostQuitMessage 函数的原型如下：

```
void PostQuitMessage(int nExitCode);  //nExitCode 为应用程序退出代码
```

函数 PostQuitMessage 的作用是向应用程序发出 WM_QUIT 消息，请求退出。除此之外，应用程序通过在消息处理程序段中加入如下语句，为未定义处理过程的消息提供默认处理：

```
default: return DefWindowProc(hwnd,message,wParam,lParam);
```

函数 DefWindowProc 是系统默认的处理过程，以保证所有发送到该窗口的消息均得以处理。

3. 数据类型

Windows 应用程序的源程序中包含种类繁多的数据类型，windows.h 是用户调用系统功能的关键，该文件中定义了 Windows 系统使用的数据类型，其中包括许多简单类型和结构。常用的部分 Windows 数据类型及其说明如表 3-7 所示。

表 3-7　常用的部分 Windows 数据类型及其说明

数据类型	说　明
WORD	16 位无符号整数
LONG	32 位有符号整数
DWORD	32 位无符号整数
HANDLE	句柄
UINT	32 位无符号整数
BOOL	布尔值
LPTSTR	指向字符串的 32 位指针
LPCTSTR	指向字符串常量的 32 位指针

4. 数据结构

Windows C 程序中数据结构比一般的 C 语言数据结构复杂。常用的数据结构有：

(1) MSG

数据结构 MSG 中包含一个消息的全部信息，既是消息发送的格式，也是 Windows 编程中最基本的数据结构之一。有关 MSG 结构的定义请参见 3.1.3 节的内容。

(2) WNDCLASS

结构 WNDCLASS 包含一个窗口类的全部信息，也是 Windows 编程中使用的基本数据结构之一。应用程序通过定义一个窗口类确定窗口的属性。其定义如下：

```
typedef struct tagWNDCLASS
{
    UINT style;                 //窗口类的样式，一般设置为 0
    WINDPROC lpfnWndProc;       //指向窗口函数的指针
    Int cbClsExtra;             //分配在窗口类结构后的字节数
    Int cbWndExtra;             //分配在窗口实例后的字节数
    HINSTANCE hInstance;        //定义窗口类的应用程序的实例句柄
```

```
    HICON hIcon;                //窗口类的图标
    HCURSOR hCursor;            //窗口类的光标
    HBRUSH hbrBackground;       //窗口类的背景刷
    LPCTSTR lpszMenuName;       //窗口类菜单资源名
    LPCTSTR lpszClassName;      //窗口类名
WNDCLASS;
```

(3) POINT

POINT 结构定义了屏幕上或窗口中的一个点的 X 坐标和 Y 坐标。POINT 结构也是应用程序中最常用的结构之一,有关 POINT 的定义请参见 3.1.3 节的内容。

(4) RECT

RECT 结构定义了一个矩形区域,其中包含该矩形区域的左上角和右下角两个点的 X 坐标和 Y 坐标。其定义如下:

```
typedef struct tagRECT
                {
                    LONG left;    //矩形框左上角 X 坐标
                    LONG top;     //矩形框左上角 Y 坐标
                    LONG right;   //矩形框右下角 X 坐标
                    LONG bottom;  //矩形框右下角 Y 坐标
                }RECT;
```

3.4.3 应用程序举例

作为应用 Windows API 函数进行编程的入门,综合以上讲述的基本内容,下面就通过一个简单的窗口示例程序说明如何编写简单的 Windows 应用程序。

【例 3-1】

应用程序窗口示例。本例的目的在于说明创建 Windows 窗口的方法及过程。

程序代码如下:

```
//windows.h 文件中包含应用程序中所需的数据类型和数据结构的定义
# include <windows.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);    //窗口函数说明
//- - - - - - - - - - - - 以下初始化窗口类 - - - - - - - - - - - - - - - -
int WINAPI WinMain( HINSTANCE  hInstance,    //WinMain 函数说明
                    HINSTANCE  hPrevInst,
                    LPSTR      lpszCmdLine,
                    int        nCmdShow)
{
        HWND hwnd ;
        MSG Msg ;
        WNDCLASS wndclass ;
        char lpszClassName[ ] = "窗口";          //窗口类名
```

```
    char lpszTitle[] = "My Windows";          //窗口标题名

    //窗口类的定义
    wndclass.style = 0;                       //窗口类型为默认类型
    wndclass.lpfnWndProc = WndProc;           //窗口处理函数为 WndProc
    wndclass.cbClsExtra = 0;                  //窗口类无扩展
    wndclass.cbWndExtra = 0;                  //窗口实例无扩展
    wndclass.hInstance = hInstance;           //当前实例句柄
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION);
                                              //窗口的最小化图标为默认图标
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW);
                                              //窗口采用箭头光标
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH);
                                              //窗口背景为白色
    wndclass.lpszMenuName = NULL;             //窗口中无菜单
    wndclass.lpszClassName = lpszClassName;
                                              //窗口类名为"窗口示例"
//-------------以下进行窗口类的注册-------------
    if( ! RegisterClass( &wndclass))          //如果注册失败则发出警告声音
    {
        MessageBeep(0);
        return FALSE;
    }

    //创建窗口
    hwnd = CreateWindow(lpszClassName,        //窗口类名
                        lpszTitle,            //窗口实例的标题名
                        WS_OVERLAPPEDWINDOW,  //窗口的风格
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,        //窗口左上角坐标为默认值
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,        //窗口的高和宽为默认值
                        NULL,                 //此窗口无父窗口
                        NULL,                 //此窗口无主菜单
                        hInstance,            //创建此窗口的应用程序的当前句柄
                        NULL);                //不使用该值

    //显示窗口
    ShowWindow( hwnd, nCmdShow);
    //绘制用户区
```

```
    UpdateWindow(hwnd);
    //消息循环
    while( GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage( &Msg) ;
        DispatchMessage( &Msg) ;
    }

    return Msg.wParam;  //消息循环结束即程序终止时将信息返回系统
}

//窗口函数
LRESULT CALLBACK WndProc(  HWND hwnd,
                           UINT message,
                           WPARAM  wParam,
                           LPARAM  lParam)
{
    switch(message)
    {
    case WM_DESTROY:
        PostQuitMessage(0);//调用 PostQuitMessage 发出 WM_QUIT 消息
    default:                    //默认时采用系统消息默认处理函数
        return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return(0);
}
```

上述程序的运行结果如图 3-4 所示。

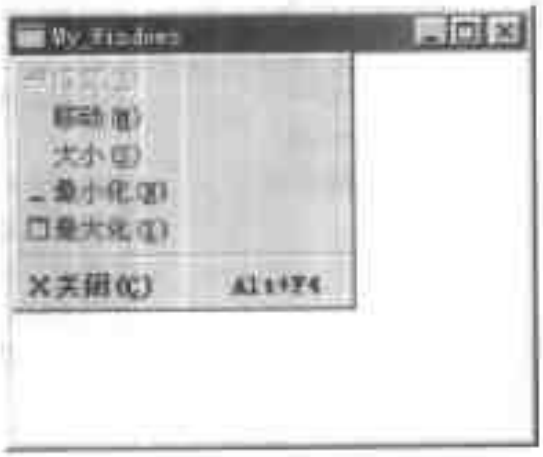


图 3-4　窗口示例程序的运行结果

习　题

3-1　Windows 编程中窗口的含义是什么?

3-2　事件驱动的特点是什么?

3-3　Windows 应用程序中的消息传递是如何进行的? 请举例说明。

3-4　句柄的作用是什么? 请举例说明。

3-5　一个 Windows 应用程序的最基本构成应有哪些部分?

3-6　应用 Windows API 函数编程时有什么特点?

第 2 篇

使用 API 函数进行可视化编程

API 函数在可视化编程中起了重要的作用。

在第 3 篇的 MFC 编程中，读者将会看到，在类的方法中大量地使用了 API 函数。因此，掌握 API 函数并熟练利用 API 函数进行可视化编程，是学习可视化编程的重要组成部分。

本篇内容包括：

第 4 章　Windows 的图形设备接口及 Windows 绘图

第 5 章　文本的输出方法与字体的设置

第 6 章　Windows 应用程序对键盘与鼠标的响应

第 7 章　资源在 Windows 编程中的应用

第 8 章　Windows 标准控件在可视化编程中的应用

第 9 章　文件及剪贴板操作

第 4 章

Windows 的图形设备接口及 Windows 绘图

Windows 图形设备接口(GDI,Graphics Device Interface)是为与设备无关的图形设计的所谓设备无关性,就是操作系统屏蔽了硬件设备的差异。因为计算机常与一系列不同的设备结合在一起,如打印机、绘图仪等输出设备以及显示设备等,因而设备无关性的图形能使用户编程时无需考虑特殊的硬件设置,这对 Windows 编程来说是非常重要的

4.1 图形设备接口

Windows 应用程序使用图形设备接口和 Windows 设备驱动程序来支持与设备无关的图形。图形设备接口(GDI)是 Windows 系统的重要组成部分,负责系统与用户或绘图程序之间的信息交换,并控制在输出设备上显示图形或文字。

计算机输出设备和显示设备种类繁多,包括不同技术标准的显示器、打印机、绘图仪等,每类设备又包含许多不同的型号。为了适应不同的设备,Windows 系统提供应用程序与具体设备分离的功能。操作系统管理并协调一系列输出设备驱动程序,将应用程序的图形输出请求转换为打印机、绘图仪、显示器或其他输出设备上的输出。GDI 的设备无关性是 Windows 操作系统的特色之一。对于开发人员而言,所要做的工作仅仅是在系统的帮助下建立一个与某个实际输出设备的关联,以要求系统加载相应的设备驱动程序,其他的具体输出操作则由系统实现。由此可见,Windows 系统分担了应用程序的硬件设备适配器功能。

4.1.1 图形设备接口的一些基本概念

设备描述表(Device Context)是一个用来确定任何设备的 GDI 输出的位置和形象的属性的集合。应用程序并不能直接存取设备描述表,但是,应用程序可以使用设备描述表的句柄来间接地存取设备描述表及其属性。当程序为设备描述表要求一个句柄时,就将创建一个设备描述表。创建的设备描述表包含了它所有的属性和默认值,应用程序可以修改这些属性。

应用程序对图形的操作均参照设备描述表中的属性执行,设备描述表的各个属性含义如表 4-1 所示。因此可以将设备描述表看成图形的"输出模板"。依靠这块模板,当程序员调用 GDI 函数输出图形或文字时,不必关心诸如背景颜色、字体等问题。

表 4-1　设备描述表属性及相关函数

属　性	默认值	相关函数
背景色	WHITE	GetBkColor SetBkColor
背景模式	OPAQUE	GetBkMode SetBkMode
位图	NONE	CreateBitMap CreateBitMapIndirect CreateCompatibleBitMap SelectObject
画刷	WHITE_BRUSH	CreateBrushIndirect CreateDIBPatternBrush CreateHatchBrush CreatePatternBrush CreateSolidBrush SelectObject
画刷起始位置	(0,0)	GetBrushOrg SetBrushOrg UnrealizeObject
剪截域	DISPLAY_SURFACE	ExcludeClipRect IntersectClipRect OffsetClipRgn SelectClipPath SelectObject SelectClipRgn
颜色调色板	DEFAULT_PALETTE	CreatePalette RealizePalette SelectPalette
当前笔的位置	(0,0)	GetCurrentPosition LineTo PolyLineTo MoveTo
绘图模式	R2_COPYPEN	GetROP2 SetROP2
字体	SYSTEM_FONT	CreateFont CreateFontIndirect SelectObject
字符间距	0	GetTextCharacterExtra SetTextCharacterExtra
映像模式	MM_TEXT	GetMapMode SetMapMode
画笔	BLACK_PEN	CreatePen CreatePenIndirect SelectObject

续表

属　性	默认值	相关函数
多边形填充模式	ALTERNATE	GetPolyFillMode SetPolyFillMode
缩放模式	BLACKONWHITE	SetStretchBltMode GetStretchBltMode
文本颜色	BLACK	GetTextColor SetTextColor
视图范围	(1,1)	GetViewportExt SetMapMode SetViewportExt ScaleViewportExt
视图原点	(0,0)	GetViewportOrg OffsetViewportOrg SetViewportOrg
窗口范围	(1,1)	GetWindowExt SetMapMode SetWindowExt ScaleWindowExt
窗口原点	(0,0)	GetWindowOrg OffsetWindowOrg SetWindowOrg

4.1.2　图形刷新

图形刷新是绘图过程中必须考虑的问题，图形刷新包括刷新的请求、系统对刷新请求的响应以及具体的刷新方法。

1.刷新请求

首先考虑这样一种实际情况：应用程序在窗口的用户区绘制了一个椭圆，然后显示一个颜色列表框，用户在列表框上选择填充椭圆内部的颜色。但是，显示的列表框覆盖了椭圆的一部分。现在的问题是，当用户结束颜色选择操作并关闭对话框后，应用程序将如何恢复椭圆被覆盖部分的颜色和形状。

Windows 应用程序大部分的用户操作都集中在用户区内，因此上述情况可能频繁出现。在窗口大小调整，窗口移动或其他对象覆盖后，都必须刷新窗口内用户区的内容，以恢复用户区内应有的显示形态。但是，Windows 系统并不总是记录窗口中需保存的内容，这样做既不现实又没有必要，系统只能在有限的几种情况下自动刷新。因此，应用程序必须具有及时处理刷新请求和刷新图形的功能。

Windows 系统通常通过发送 WM_PAINT 消息将刷新请求传递给应用程序。当用户区的内容需要刷新时，系统在应用程序的消息队列中加入该消息，以通知窗口函数执行刷新处理。

2. 系统对刷新请求的响应

一般情况下，当窗口需要刷新时，系统向应用程序消息队列发送 WM_PAINT 消息。刷新有三种可能，它们分别是窗口移动后的刷新、被覆盖区域的刷新以及对象穿越后的刷新。刷新请求的产生比较复杂，系统的响应也不尽相同。因此 Windows 系统对刷新请求的响应也相应分为以下三种情况：

(1) 窗口移动后的刷新

窗口移动后的刷新可以理解为下列事件的发生，这时系统将向应用程序发送产生 WM_PAINT 消息：

- 用户区移动或显示。
- 用户窗口大小改变。
- 程序通过滚动条滚动窗口。

(2) 被覆盖区域的刷新

当下面的事件发生时，Windows 系统将试图保存被覆盖的区域，以备以后刷新。此后如果系统不能有效刷新，则向应用程序发送 WM_PAINT 消息：

- 下拉式菜单关闭，并需要恢复被覆盖的部分。
- Windows 清除对话框或消息框等对象，并需要恢复被覆盖的部分。

对于这种情况，程序员必须有效地组织应用程序，使其能够在系统刷新失效时利用窗口处理函数刷新。

窗口被另一个窗口覆盖的区域称为无效区域。用户区中无效区域的产生可能导致系统向应用程序发送一条消息。

Windows 系统为每个窗口建立了一个 PAINTSTRUCT 结构，该结构中包含了包围无效区域的一个最小矩形的结构 RECT，这个矩形称为无效矩形。应用程序可以根据这个无效矩形执行刷新操作。

(3) 对象穿越后的刷新

对于下面的对象穿越后的情况，Windows 系统自动完成刷新任务，应用程序不必考虑：

- 光标穿过用户区。
- 图标拖过用户区。

因此，为了执行有效的刷新，应用程序必须全面分析系统可能发送的刷新请求，并根据不同的情况分别处理。这是编写应用程序的一个难点。

3. 刷新方法

常用的 Windows 应用程序刷新窗口的方法如下：

- 在内存中保持一个显示输出的副本，当需要重绘窗口时，将副本拷贝到相应的窗口中。该方法适用于刷新位图等复杂图形。
- 记录曾经发生的事件，在窗口需要刷新时重新调用窗口执行这个事件。
- 重新绘制图形，一般对于简单图形常采用重新绘制图形方法执行刷新。在应用程序中，将图形绘制处理程序放在消息 WM_PAINT 响应模块中，一旦程序接收到刷新请求即可重绘图形。

4.1.3　获取设备环境

获取设备环境是应用程序输出图形的先决条件，常用的两种获取设备环境的方法是调用函数 BeginPaint 或 GetDC。

1. 调用 BeginPaint 函数

应用程序响应 WM_PAINT 消息进行图形刷新时，主要通过调用 BeginPaint 函数获取设备环境，其形式为：

```
hdc = BeginPaint(hwnd,&ps);//ps 为 PAINTSTRUCT 类型结构,定义方式为:PAINTSTRUCT ps
```

PAINTSTRUCT 数据结构是 Windows 系统提供的标识无效区域的结构，其定义如下：

```
Typedef struct tagPAINTSTRUCT
    HDC hdc;                //设备环境句柄
    BOOL fErase;            // fErase 一般取真值,表示擦除无效矩形的背景
    RECT rcPaint;           //无效矩形标识
    BOOL fRestore;          //系统保留
    BOOL fIncUpdate;        //系统保留
    BYTE rgbReserved[16];   //系统保留
} PAINTSTRUCT;
```

rcPaint 为标准的 RECT 数据结构，其作用是标识无效矩形，该结构中包含了无效矩形的左上角和右下角的坐标。

系统调用 BeginPaint 函数获取设备环境的同时，填写 ps 结构，以标识需要刷新的无效矩形区，提供给后继过程进一步处理。

由 BeginPaint 函数获取的设备环境必须用 EndPaint 函数释放，其原型为：

```
void EndPaint(HWND hwnd,PAINTSTRUCT &ps);
```

2. 调用 GetDC 函数

如果 Windows 应用程序的绘图工作并非由 WM_PAINT 消息驱动，则需调用 GetDC 函数获取设备环境。其形式为：

```
hdc = GetDC(hwnd);
```

由 GetDC 函数获取的设备环境必须用 ReleaseDC 函数释放，其原型为：

```
void ReleaseDC(HWND hwnd);
```

两种获取设备环境方法的区别如表 4-2 所示。

表 4-2　BeginPaint 与 GetDC 的区别

项目 ＼ 函数	BeginPaint 函数	GetDC 函数
适用场合	只用于图形刷新时获取设备环境	使用较为广泛
操作区域	使用 BeginPaint 函数获取设备环境后，操作区域为无效区域	使用 GetDC 函数获取设备环境后，操作区域为整个用户区
释放设备环境所用函数	由 EndPaint 函数释放	由 ReleaseDC 函数释放

4.1.4　映像模式

映像模式是设备环境的内容之一，其优点是程序员可不必考虑输出设备的坐标系情

况，而在一个统一的逻辑坐标系中进行图形的绘制与操作。映像模式定义了将逻辑单位转化为设备的度量单位以及设备的 x 方向和 y 方向。Windows 中的映像模式如表 4-3 所示。

表 4-3 Windows 中的映像模式

映像模式	将一个逻辑单位映像为	坐标系设定
MM_ANISOTROPIC	系统确定	可选
MM_HIENGLISH	0.001 英寸	Y 向上，X 向右
MM_HIMETRIC	0.01 毫米	Y 向上，X 向右
MM_ISOTROPIC	系统确定	可选，但 x 轴和 y 轴的单位比例为 1:1
MM_LOENGLISH	0.01 英寸	Y 向上，X 向右
MM_LOMETRIC	0.1 毫米	Y 向上，X 向右
MM_TEXT	一个像素	Y 向下，X 向右
MM_TWIPS	1/1440 英寸	Y 向上，X 向右

映像模式对应用程序是很重要的。上述的映像模式中，MM_TEXT 映像模式得到了普遍的应用，是默认的映像模式。

MM_ANISOTROPIC 和 MM_ISOTROPIC 这两种模式通过将图形从程序员定义的逻辑坐标窗口映像到物理设备的视口以实现坐标转换。窗口是对应逻辑坐标系上程序员设定的一个区域，视口是对应于实际输出设备上程序员设定的一个区域。换言之，如果程序员设定的映像模式为 MM_ANISOTROPIC 和 MM_ISOTROPIC，则只需确定一个以逻辑坐标系为基础的窗口和一个以物理设备坐标系为基础的视口，Windows 系统即可按照窗口和视口的坐标比例自动调整图形。

这两种映像模式的不同是 MM_ISOTROPIC 模式要求将窗口中的对称图形映像到视口时仍为对称图形，这种要求可能导致系统强制变换视口。而 MM_ANISOTROPIC 模式则完全按照窗口和视口的坐标比例进行映像。

1. 坐标系统

在 Windows 应用程序中有好几种坐标系统，它们大致可以分为两大类：设备坐标系统和逻辑坐标系统。

在设备坐标系统中又有三种相互独立的坐标系统：屏幕坐标系统、窗口坐标系统和用户区坐标系统。这些设备坐标系统均以像素点来表示度量的单位。X 轴的正方向为从左到右，Y 轴的正方向为从上向下。注意，改变像素点数只是改变相关的视频模式，而改变度量单位将改变相关的设备描述表。

屏幕坐标系统使用整个屏幕坐标区作为坐标系统。

窗口坐标系统使用了边界在内的应用程序的窗口作为坐标区域。窗口边界的左上角是坐标系的原点。

用户区坐标系统是最经常使用的坐标系统。用户区是窗口工作区，不包括窗口边界、菜单栏及滚动条等。用户一般只操作应用程序的用户区，因此用户区坐标系统对大多数应用程序都是适用的。

其他的坐标系统都是逻辑坐标系统。其中映像模式规定了 GDI 函数中定义的逻辑单位如何转化为设备坐标。在画一个对象以前，Windows 操作系统会把这些逻辑单位翻译成相应的设备坐标系统中的单位。

2. 映像模式的设置

应用程序可获取设备环境的当前映像模式，并可根据需要设置映像模式。相关的函数为 SetMapMode 和 GetMapMode。调用设置映像模式函数 SetMapMode 可设置设备环境的映像模式，其形式为：

```
SetMapMode(hdc,nMapMode);  //nMapMode 为映像模式的整型标识符
```

调用 GetMapMode 函数可获取当前设备环境的映像模式，其形式为：

```
nMapMode = GetMapMode(hdc);
```

窗口区域的定义由 Set WindowExtEx 函数完成，其函数原型为：

```
BOOL SetWindowExtEx
    (
    HDC hdc,
    int nHeight,        //nHeight 为以逻辑单位表示的新窗口区域高度
    int nWidth,         //nWidth 为以逻辑单位表示的新窗口区域宽度
    LPSIZE lpSize       // lpSize 为保存函数调用前窗口区域尺寸的 SIZE 结构地址，
                        //如果取 NULL，则表示忽略调用前的尺寸
    );
```

视口区域的定义由 SetWindowExtEx 函数完成，其函数原型为：

```
BOOL SetViewportExtEx
    (
    HDC hdc,
    int nHeight,        //nHeight 为以物理设备单位表示的新视口区域高度
    int nWidth,         //nWidth 为以物理设备单位表示的新视口区域宽度
    LPSIZE lpSize       // lpSize 为保存函数调用前视口区域尺寸的 SIZE 结构地址，
                        //如果取 NULL，则表示忽略调用前的尺寸
    );
```

视口的默认原点和窗口的默认原点均为(0,0)。可通过调用函数 SetViewportOrgEx 和 SetWindowOrgEx 设定窗口与视口的原点。

SetWindowOrgEx 函数的原型为：

```
BOOL SetWindowOrgEx
    (
    HDC hdc,
    int X,              //X 和 Y 为以逻辑单位表示的新窗口原点坐标
    int Y,
    LPPOINT lpPoint     //lpPoint 为保存函数调用前原点坐标的 POINT 结构的地址，
                        //如果取 NULL，则忽略调用前的尺寸
    );
```

其中，SetWindowOrgEx 函数和 SetViewportOrgEx 函数只有在映像模式为 MM

ANISOTROPIC 和 MM_ISOTROPIC 时才有意义。

在建立了窗口、视口以及映像模式的概念后，就可以在窗口上绘制相应的图形了，在绘制图形之前，还必须选择绘图工具，如画笔或画刷，以及它们的颜色属性等。

4.2 绘图工具和颜色的设置与应用

Windows 绘图使用画笔和画刷进行，画笔的功能是画直线和曲线，画刷用于指定区域的填充

4.2.1 画笔的应用

画笔的操作包括创建画笔、将画笔选入设备环境和删除画笔。

1. 画笔的创建

使用画笔之前必须事先定义一个画笔句柄，形式如下：

```
HPEN hP;
```

定义画笔句柄完成后，可直接调用函数 GetStockObject 获取 Windows 系统定义的四种画笔这四种画笔分别是 WHITE_PEN、BLACK_PEN、DC_PEN 和 NULL_PEN。例如，获取画笔 BLACK_PEN 的形式如下：

```
hP = GetStockObject(BLACK_PEN);
```

当然，也可创建新画笔。创建新画笔的形式如下：

```
hP = CreatePen
    (
    int nPenStyle,        //确定画笔样式，可选样式及说明如表 4-4 所示
    int nWidth,           //画笔宽度，取 0 表示一个像素宽
    COLORREF rgbColor     //画笔颜色
    );
```

表 4-4 画笔样式及说明

样 式	说 明	样 式	说 明
PS_DASH	虚线	PS_INSIDEFRAME	实线(边框线)
PS_DASHDOT	点划线	PS_NULL	无
PS_DASHDOTDOT	双点划线	PS_SOLID	实线
PS_DOT	点线		

2. 将画笔选入设备环境

创建画笔后，必须调用 SelectObject 函数将其选入设备环境。其形式如下：

```
SelectObject(hdc,hP);  //hP 为所创建或获取的画笔句柄
```

调用该函数后，应用程序将使用句柄 hP 所指的画笔绘图，直到选入另外的一种画笔为止。SelectObject 函数的返回值中保存上一次使用的画笔句柄。

3. 删除画笔

不再使用当前画笔时，需调用函数 DeleteObject 删除画笔，以免占用内存空间。其形

式为：

```
DeleteObject(hP);
```

4.2.2　画刷的应用

画刷的创建和应用与画笔很相似，操作画刷也包括创建、选入设备环境和删除。

1. 画刷的创建

使用画刷需事先定义一个画刷句柄。形式如下：

```
HBRUSH hBr;   //hBr 为画刷句柄
```

定义画刷句柄后，可直接调用函数 GetStockObject 获取 Windows 系统提供的 7 种画刷，调用画刷的形式如下：

```
hBr = GetStockObject(nBrushStyle);    //nBrushStyle 为画刷样式，详见表 4-5
```

表 4-5　画刷的样式及其说明

样　式	说　明	样　式	说　明
BLACK_BRUSH	黑色画刷	LTGRAY_BRUSH	亮灰色画刷
DKGRAY_BRUSH	深灰色画刷	NULL_BRUSH	空画刷
GRAY_BRUSH	灰色画刷	WHITE_BRUSH	白色画刷
HOLLOW_BRUSH	虚画刷		

也可调用函数 CreateSolidBrush 和 CreateHatchBrush 创建画刷。调用函数 CreateSolidBrush 可创建一个具有指定颜色的单色画刷，调用形式如下：

```
hBr = CreateSolidBrush(rgbColor);  //rgbColor 为画刷颜色
```

调用函数 CreateHatchBrush 可创建具有指定阴影图案和颜色的画刷，其调用形式如下：

```
hBr = CreateHatchBrush
    (
    int nHatchStyle,          // nHatchStyle 为阴影模式标识，详见表 4-6
    COLORREF rgbColor         //画刷颜色
    );
```

表 4-6　画刷的阴影模式

样　式	说　明	样　式	说　明
HS_BDIAGONAL	45°从左上角到右下角的阴影线	HS_CROSS	垂直相交的阴影线
HS_DIAGCROSS	45°叉线	HS_HORIZONTAL	水平阴影线
HS_FDIAGONAL	45°从左下角到右上角的阴影线	HS_VERTICAL	垂直阴影线

2. 选入设备环境

创建画刷完成后，必须调用 SelectObject 函数将其选入设备环境中。其形式如下：

```
SelectObject(hdc,hBr);
```

3. 删除画刷

不再使用创建的画刷时，可以调用函数 DeleteObject 删除画刷，以释放占用的内存空

间，其形式为：

```
DeleteObject(hBr);
```

4.2.3　颜色的设置

Windows 使用 32 位无符号整数表示颜色，如图 4-1 所示。32 位整数中的低三位字节分别表示红、绿、蓝三个颜色值，每一个颜色值的范围是 0 ~ 255。

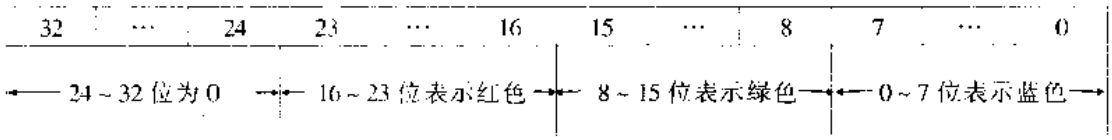


图 4-1　32 位无符号整数表示颜色

Windows 使用宏 RGB 定义绘图的颜色，其形式为：

```
RGB(nRed, nGreen,nBlue)
```

其中 nRed、nGreen 和 nBlue 分别表示红色值、绿色值和蓝色值，例如 RGB(255,0,0)表示红色，RGB(0,255,0)代表蓝色，RGB(0,0,255)表示绿色。

在定义了画笔或画刷及其属性以后，我们就可以利用这些画笔或画刷通过调用相关的绘图函数进行绘图操作。

4.3　常用绘图函数

Windows GDI 函数很多，下面只介绍常用的几种，更多的函数可参阅有关手册。

1. 设置画笔当前位置的函数 MoveToEx

设置画笔当前位置的函数 MoveToEx 的原型如下：

```
BOOL MoveToEx
    (
    HDC hdc,
    int X,              // X,Y 分别为新位置的逻辑坐标
    int Y,
    LPPOINT lpPoint     //lpPoint 为存放原画笔位置的 POINT 结构地址
    );
```

2. 从当前位置向指定坐标点画直线的函数 LineToEx

从当前位置向指定坐标点画直线的函数 LineToEx 的原型如下：

```
BOOL LineToEx(HDC hdc,int X,int Y); //其中 X 和 Y 为线段的终点坐标
```

3. 从当前位置开始，依次用线段连接 lpPoints 中指定各点的函数 Polyline

该函数的原型如下：

```
BOOL Polyline
    (
    HDC hdc,
    LPPOINT lpPoints,     // lpPoints 为指向包含各点坐标的 POINT 结构数组的指针
```

```
    int nCount                  // nCount 为 POINT 数组中点的个数
    );
```

4. 绘制椭圆弧线的函数 Arc

绘制椭圆弧线的函数 Arc 的原型如下：

```
BOOL Arc
    (
    HDC hdc,
    int X1,intY1,           //指定边框矩形左上角的逻辑坐标
    int X2,int Y2,          //指定边框矩形右下角的逻辑坐标
    int X3,int Y3,          //椭圆弧起始径线的确定点坐标
    int X4,int Y4           //椭圆弧终止径线的确定点坐标
    );
```

Arc 函数所画的椭圆弧线由给定边框矩形所形成的椭圆定义，这个矩形由左上角逻辑坐标(X1,Y1)和右下角逻辑坐标(X2,Y2)确定。该弧的起点是(X3,Y3)和矩形中心的连线与椭圆的交点，终点为(X4,Y4)和矩形中心的连线与椭圆的交点。而且该弧是从起点向终点逆时针画出，如图 4-2 所示。

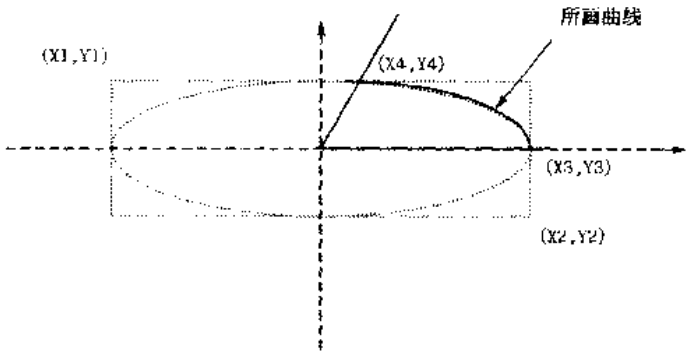


图 4-2　Arc 函数功能示意图

5. 使用当前画笔绘制一个饼图，并使用当前画刷进行填充的函数 Pie

该函数的原型如下：

```
BOOL Pie
    (
    HDC hdc,
    int X1,intY1,           //指定边框矩形左上角的逻辑坐标
    int X2,int Y2,          //指定边框矩形右下角的逻辑坐标
    int X3,int Y3,          //椭圆弧起始径线的确定点坐标
    int X4,int Y4           //椭圆弧终止径线的确定点坐标
    );
```

Pie 函数所画饼图为椭圆弧线和两条径线所围的区域，如图 4-3 所示。

6. 使用当前画笔绘制一个矩形，并使用当前画刷进行填充的函数 Rectangle

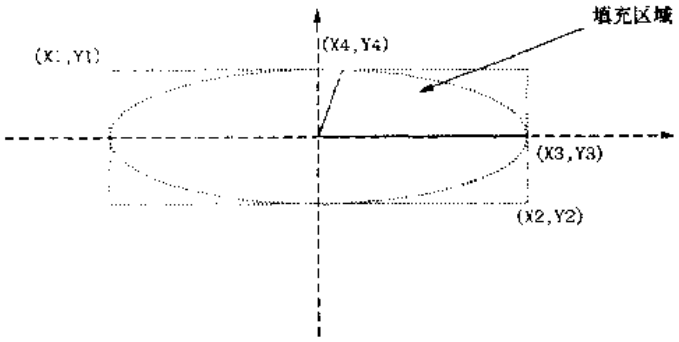


图 4-3 Pie 函数示意图

该函数的原型如下：

```
BOOL Rectangle
    (
    HDC hdc,
    int X1,int Y1,  //(X1,Y1)为矩形的左上角的逻辑坐标
    int X2,int Y2   //(X2,Y2)为矩形的右下角的逻辑坐标
    );
```

7. 使用当前画笔绘制一个圆角矩形，并使用当前画刷进行填充的函数 RoundRect

该函数的原型为：

```
BOOL RoundRect
    (
    HDC hdc,
    int X1,int Y1,     //(X1,Y1)为矩形左上角的逻辑坐标
    int X2,int Y2,     //(X2,Y2)为矩形右下角的逻辑坐标
    int nHeight,       // nHeight 为圆角的高度
    int nWidth         // nWidth 为圆角的宽度
    );
```

8. 使用当前画笔绘制一个椭圆，并使用当前画刷填充的函数 Ellipse

该函数的原型为：

```
BOOL Ellipse
    (
    HDC hdc,
    intX1,intY1,       //(X1,Y1)为边界矩形左上角的逻辑坐标
    intX2,intY2        //(X2,Y2)为边界矩形右下角的逻辑坐标
    );
```

9. 使用当前画笔绘制一个多边形，并使用当前画刷填充的函数 Polygon

该函数的原型如下：

```
BOOL Polygon
    (
```

```
    HDC hdc,
    LPPOINT lpPoints,  // lpPoints 为包含各点坐标的 POINT 数组的地址
    int nCount         // nCount 为多边形点的个数
    );
```

4.4　应用实例

【例 4-1】

利用绘图函数创建填充区。Windows 通过使用当前画笔画一个图形的边界，然后用当前的刷子填充这个图形来创建一个填充图形。共有三个填充图形，第一个是用深灰色画刷填充带圆角的矩形，第二个是采用亮灰色画刷填充一个椭圆形图，第三个是用虚画刷填充饼形图。读者会发现，使用虚画刷填充时，看不出填充效果。

下面是本例子的源程序代码：

```
#include<windows.h>
#include<stdlib.h>
#include<string.h>
long WINAPI WndProc
    (
    HWND hWnd,
    UINT iMessage,
    UINT wParam,
    LONG lParam
    );

BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
HWND hWndMain;

int WINAPI WinMain                          //主函数
    (
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
    )
{
    MSG Message;
    if(! InitWindowsClass(hInstance))
        return FALSE;
    if(! InitWindows(hInstance,nCmdShow))
```

```
        return FALSE;
    while(GetMessage(&Message,0,0,0))    //消息循环

        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
    return Message.wParam;
}

long WINAPI WndProc                      //消息处理函数
    (
    HWND hWnd,
    UINT iMessage,
    UINT wParam,
    LONG lParam
    )

    HDC hDC;                             //定义指向设备描述表的句柄
    HBRUSH hBrush;                       //定义指向画刷的句柄
    HPEN hPen;                           //定义指向画笔的句柄
    PAINTSTRUCT PtStr;                   //定义指向包含绘图信息的结构体变量
    switch(iMessage)                     //处理消息
    {
    case WM_PAINT:                                          //处理绘图消息
        hDC = BeginPaint(hWnd,&PtStr);
        SetMapMode(hDC,MM_ANISOTROPIC);                     //设置映像模式
        hPen = (HPEN)GetStockObject(BLACK_PEN);             //设置画笔为系统预定义的
                                                            //黑色画笔
        hBrush = (HBRUSH)GetStockObject(DKGRAY_BRUSH);      //采用系统预定义的深灰色
                                                            //画刷
        SelectObject(hDC,hBrush);                           //选择画刷
        SelectObject(hDC,hPen);                             //选择画笔
        RoundRect(hDC,50,120,100,200,15,15);                //圆角矩形
        hBrush = (HBRUSH)GetStockObject(LTGRAY_BRUSH);      //采用系统预定义的亮灰色
                                                            //画刷
        SelectObject(hDC,hBrush);                           //选择画刷
        Ellipse(hDC,150,50,200,150);                        //椭圆
        hBrush = (HBRUSH)GetStockObject(HOLLOW_BRUSH);      //采用系统预定义的虚画刷
        SelectObject(hDC,hBrush);                           //选择画刷
        Pie(hDC,250,50,300,100,250,50,300,50);              //饼形
        EndPaint(hWnd,&PtStr);                              //结束绘图
```

```
        return 0;
    case WM_DESTROY:                                    //结束应用程序
        PostQuitMessage(0);
        return 0;
    default:                                            //其他消息处理程序
        return(DefWindowProc(hWnd,iMessage,wParam,lParam));
    }
}

BOOL InitWindows(HINSTANCE hInstance,int nCmdShow)      //初始化窗口
{
    HWND hWnd;
    hWnd=CreateWindow("WinFill",                        //生成窗口
                      "填充示例程序",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT,
                      0,
                      CW_USEDEFAULT,
                      0,
                      NULL,
                      NULL,
                      hInstance,
                      NULL);
    if(! hWnd)
        return FALSE;
    hWndMain=hWnd;
    ShowWindow(hWnd,nCmdShow);                          //显示窗口
    UpdateWindow(hWnd);
    return TRUE;
}

BOOL InitWindowsClass(HINSTANCE hInstance)              //定义窗口类
{
    WNDCLASS WndClass;
    WndClass.cbClsExtra=0;
    WndClass.cbWndExtra=0;
    WndClass.hbrBackground=(HBRUSH)(GetStockObject(WHITE_BRUSH));
    WndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
    WndClass.hIcon=LoadIcon(NULL,"END");
    WndClass.hInstance=hInstance;
```

```
    WndClass.lpfnWndProc = WndProc;
    WndClass.lpszClassName = "Win?!!";
    WndClass.lpszMenuName = NULL;
    WndClass.style = CS_HREDRAW|CS_VREDRAW;
    return RegisterClass(&WndClass);
```

该例子运行的结果如图 4-4 所示

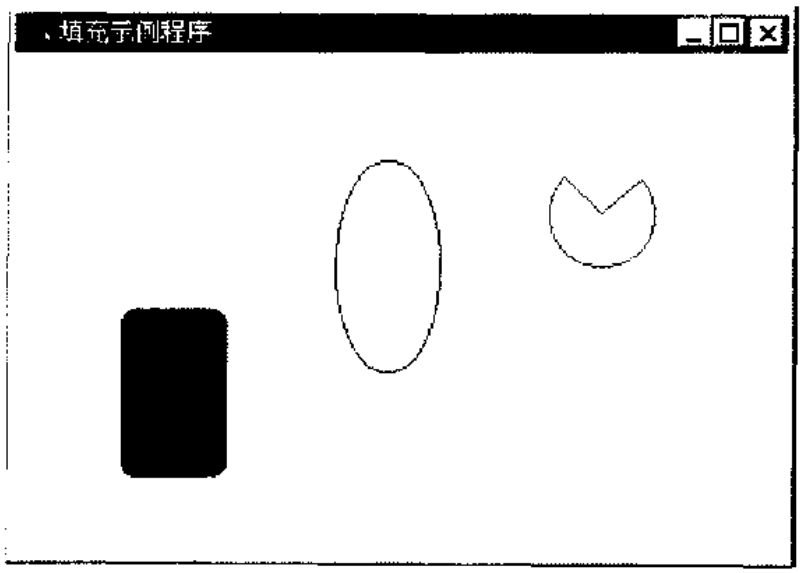


图 4-4　图形填充示例的运行结果

【例 4-2】

设置映像与使用映像模式。本例要求程序的初始阶段按模式 MM_TEXT 绘图，用户通过单击鼠标以改变映像模式，单击鼠标左键时，产生 WM_LBUTTONDOWN 消息，通知应用程序设置映像模式为 ISOTROPIC，并调用 InvalidateRect 函数刷新用户区；在用户单击鼠标右键时，产生 WM_RBUTTONDOWN 消息，应用程序将映像模式设置成ANISOTROPIC，同时刷新用户区。

本例的源程序代码如下：

```
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

int WINAPI WinMain
           (
           HINSTANCE hInstance,
           HINSTANCE hPrevInst,
```

```
            LPSTR lpszCmdLine,
            int nCmdShow
            )

{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[ ] = "映像模式";
    char lpszTitle[ ] = "My-Map-Mode";

    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;

    if( ! RegisterClass( &wndclass))
    {
        MessageBeep(0) ;
        return FALSE ;
    }

    hwnd = CreateWindow
            (
            lpszClassName,
            lpszTitle,
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            NULL,
            NULL,
            hInstance,
            NULL
```

```
        );

    ShowWindow( hwnd, nCmdShow);                    //显示窗口
    UpdateWindow(hwnd);                             //绘制用户区
    while( GetMessage(&Msg, NULL, 0, 0))            //消息循环
    {
        TranslateMessage( &Msg);
        DispatchMessage( &Msg);
    }

    return Msg.wParam;                  //消息循环结束时将信息返回系统
}

LRESULT CALLBACK WndProc                //窗口函数
    (
    HWND hwnd,
    UINT message,
    WPARAM  wParam,
    LPARAM  lParam
    )
{

    HDC hdc;
    PAINTSTRUCT ps;
    HBRUSH hB1,hB2;
    int nMode;                          //定义映像模式变量 nMode

    switch(message)
    {
        case WM_CREATE:                 //窗口创建时设置映像模式为 MM_TEXT
            nMode = MM_TEXT;
        case WM_LBUTTONDOWN:    //单击鼠标左键更改映像模式为 MM_ISOTROPIC
            nMode = MM_ISOTROPIC;
            InvalidateRect(hwnd,NULL,1);            //刷新用户区
            break;
        case WM_RBUTTONDOWN:    //单击鼠标右键更改显示模式为 MM_ANISOTROPIC
            nMode = MM_ANISOTROPIC;
            InvalidateRect(hwnd,NULL,1);            //刷新用户区
            break;
        case WM_PAINT:
            hdc = BeginPaint(hwnd,&ps);             //取得设备环境句柄
```

```
            SetMapMode(hdc,nMode);                    //设置映像模式
            SetWindowExtEx(hdc,150,150,NULL);         //设置窗口区域
            SetViewportExtEx(hdc,150,100,NULL);       //设置视口区域
            SetViewportOrgEx(hdc,150,60,NULL);        //设置视口原点
            hB1 = GetStockObject(WHITE_BRUSH);        //定义白色画刷
            hB2 = GetStockObject(BLACK_BRUSH);        //定义黑色画刷
            SelectObject(hdc,hB1);                    //将黑白画刷选入设备环境
            RoundRect(hdc,0,0,150,150,30,30);         //使用当前画笔绘制一个带圆角
                                                      //的矩形,并用当前画刷填充
            SelectObject(hdc,hB2);                    //更改当前画刷为黑色画刷
            Ellipse(hdc,0,10,150,140);                //使用当前画笔绘制椭圆,
                                                      //并用当前画刷填充
            EndPaint(hwnd,&ps);                       //释放设备环境句柄
            break;
        case WM_DESTROY:
            DeleteObject(hB1);                        //退出窗口时删除画刷
            DeleteObject(hB2);
            PostQuitMessage(0);
            break;
        default:
            return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

该程序的运行结果如图 4-5 所示。

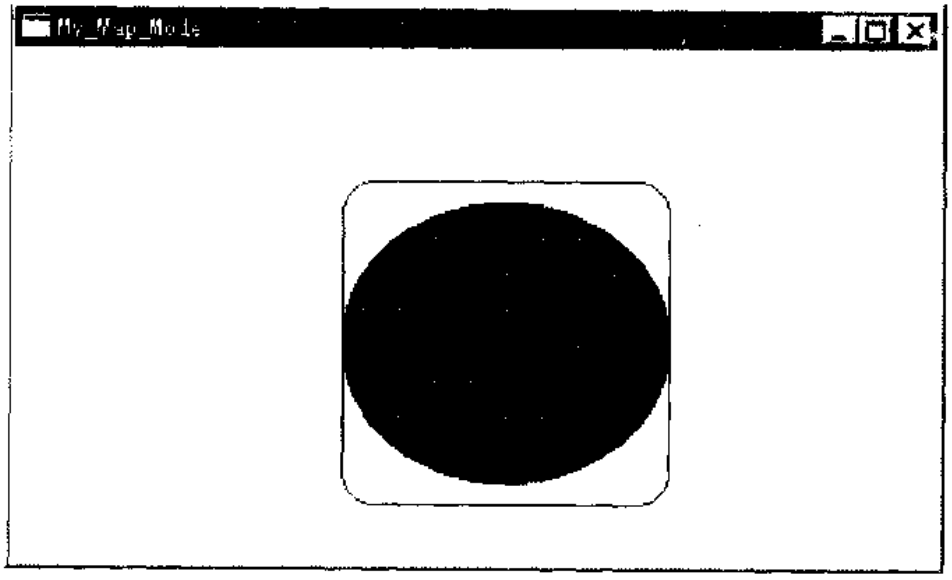


图 4-5　映像模式例程的运行结果

【例 4-3】

绘图与刷新。制订一种重新绘制图形的刷新方式，将图形绘制模块放在消息 WM PAINT 的处理过程中，当窗口需要刷新时，通知窗口函数重新绘制图形以完成刷新工作。本例要求先使用画笔和画刷绘制一个矩形，然后使用红色网格绘制一个椭圆，再使用绿色点划线绘制椭圆的轴线。

在一个复杂的应用程序中，由于窗口用户区使用频繁，局部窗口的刷新比较难掌握，因此，通过重新绘制图形的刷新方式就显得比较重要。

实现本例要求的代码如下：

```
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain
        (
        HINSTANCE hInstance,
        HINSTANCE hPrevInst,
        LPSTR lpszCmdLine,
        int nCmdShow
        )
{
        HWND hwnd;
        MSG Msg;
        WNDCLASS wndclass;
        char lpszClassName[] = "基本绘图";
        char lpszTitle[] = "My-Drawing";
        wndclass.style = 0;
        wndclass.lpfnWndProc = WndProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = hInstance;
        wndclass.hIcon = LoadIcon( NULL, IDI-APPLICATION);
        wndclass.hCursor = LoadCursor( NULL, IDC-ARROW);
        wndclass.hbrBackground = GetStockObject( WHITE-BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = lpszClassName;

        if( ! RegisterClass( &wndclass))
```

```
        {
            MessageBeep(0) ;
            return FALSE ;
        }

    hwnd = CreateWindow
        (
        lpszClassName,
        lpszTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    ) ;

    ShowWindow(hwnd, nCmdShow) ;
    UpdateWindow(hwnd);

    while( GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage(&Msg) ;
        DispatchMessage(&Msg) ;
    }

    return Msg.wParam;
}

LRESULT CALLBACK WndProc
    (
    HWND hwnd,
    UINT message,
    WPARAM  wParam,
    LPARAM  lParam
    )
```

```
    HDC hdc;
    PAINTSTRUCT ps;
    HPEN hP;                                    //定义画笔句柄
    HBRUSH  hB;                                 //定义画刷句柄

    switch(message)
    {
    case WM_PAINT:                              //通过响应 WM_PAINT 消息完成绘图工作
        hP = CreatePen(PS_DASHDOT,1,RGB(0,255,0));//自定义一个画笔,该画笔为绿色,
                                                    //所画线条为点划线,宽度为 1
        hB = CreateHatchBrush(HS_CROSS,RGB(255,0,0));
                                                //自定义一个红色网格状画刷
        hdc = BeginPaint(hwnd,&ps);             //取得设备环境句柄
        SetMapMode(hdc,MM_TEXT);                //设置映像模式,此程序使用默认模式,可省略
                                                //使用当前默认画笔、画刷进行绘图
        Rectangle(hdc,130,60,270,200);//使用当前画笔绘制矩形,用当前画刷填充
        SelectObject(hdc,hB);                   //更新画刷,设置自定义画刷为当前画刷
        Ellipse(hdc,130,70,270,190);            //使用当前画笔绘制一个椭圆,用当前画刷填充椭圆
        SelectObject(hdc,hP);                   //更新画笔,设置自定义画笔为当前画笔
        MoveToEx(hdc,100,130,NULL);             //使用当前画笔绘制轴线
        LineTo(hdc,300,130);
        MoveToEx(hdc,200,30,NULL);
        LineTo(hdc,200,230);
        EndPaint(hwnd,&ps);                     //释放设备环境句柄
        break;
    case WM_DESTROY:
        DeleteObject(hP);                       //退出窗口时删除画笔
        DeleteObject(hB);                       //退出窗口时删除黑色画刷
        PostQuitMessage(0);
        break;

    default:
        return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

该程序的运行结果如图 4-6 所示。

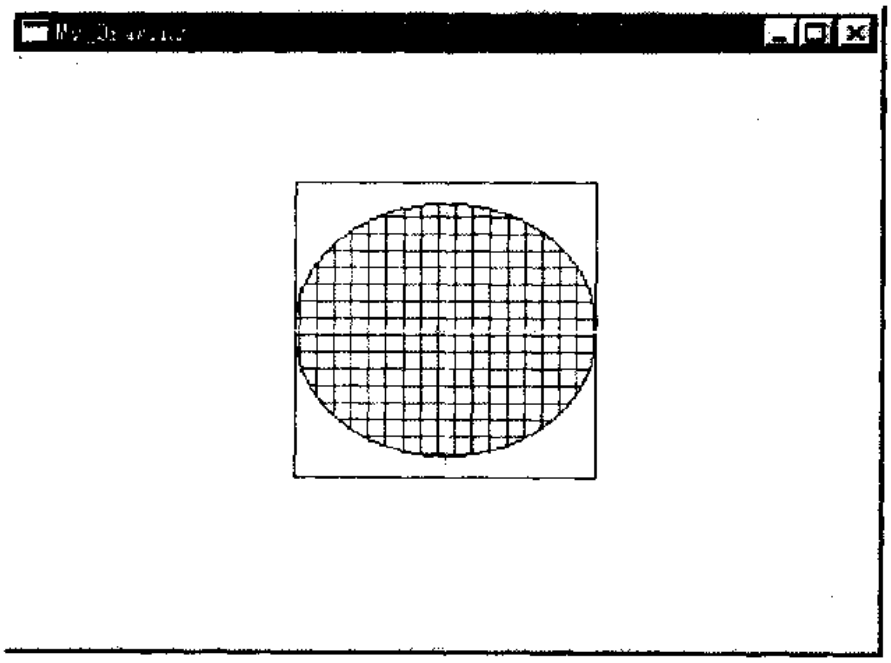


图 4-6　采用重新绘制图形完成窗口的刷新

习　题

4-1　什么是图形设备接口?

4-2　如何进行图形的刷新?

4-3　如何获取绘图工具的句柄?

4-4　如何定义映像模式?

4-5　请编写程序,要求如下:

(1) 定义一只红色的画笔,绘制一个等边五边形;

(2) 用不同颜色的线条连接互不相邻的两个点;

(3) 用不同的画刷颜色填充用上述方法所形成的图形中的每一个区域。

4-6　编写一个程序,在屏幕上出现一个圆心沿正弦曲线轨迹移动的实心圆。要求每隔四分之一周期,圆的填充色和圆的周边颜色都发生变化(颜色自己选取),同时,圆的半径在四分之一周期之内由正弦曲线幅值的 0.2 倍至 0.6 倍线性增长。

第 5 章

文本的输出方法与字体的设置

Windows 经常使用 GDI 进行文本输出。事实上，在 Windows 中，图形和文本并没有明显的界限，很多时候，Windows 把文本也当作图形对待。在一定意义上，任何内容都可以看成图形实体，文本事实上也是按照所选用字体的格式画出来的。一个字体包含了字符集中每一个字母、数字和标点符号的形状和外表的特殊信息。使用定义好的与设备无关的字体集，Windows 就能维护它的设备无关性，提供“所见即所得”的好处，这就意味着屏幕上显示的文本与用打印机或绘图仪等输出设备输出的文本是完全一样的。

在 Windows 编程中，文本操作首先要获得文本句柄。此外，应用程序还要设置字体、字符大小、字符颜色等有关属性，并将这些属性选入设备环境。

5.1 设置文本的设备环境

字体描述了所要显示的文本的大小、类型和外形，也就是说，字体包含了字符集中每个字符的一个特殊描述。在 Windows 中，字体一般又可以分成两大类型：逻辑字体和物理字体。逻辑字体定义的字符集是设备无关的，而物理字体则是为特殊设备设计的，因而是设备相关的。逻辑字体的开发相对设备字体来说更为困难，但是由于其与设备无关的特性使得逻辑字体更灵活，而且逻辑字体往往是可精确标度的，因此逻辑字体得到了广泛的应用。

5.1.1 字体句柄

Windows 系统提供了七种基本字体，如表 5-1 所示。

表 5-1 Windows 系统提供的基本字体

字 体	说 明	字 体	说 明
ANSI_FIXED	ANSI 标准的固定宽度的字体	ANSI_VAR	ANSI 标准的可变宽度的字体
DEFAULT_GUI	当前 GUI 的默认字体		
OEM_FIXED	原设备制造商（OEM）提供的标准字体	DEVICE_DEFAULT	当前图形设备的字体
		SYSTEM_FIXED	Windows 的标准固定宽度的字体
SYSTEM	Windows 提供的可变宽度的字体，它常作为默认字体		

常用的默认字体为 SYSTEM，Windows 使用该字体作为系统界面字体，选择系统字体一般需要执行如下步骤：

（1）定义字体句柄变量，语法如下：

```
HFONT hF;     //hF 为字体的句柄
```

（2）调用函数 GetStockObject 获得系统字体句柄，它返回的是系统的默认字体，语法如下：

```
hF = GetStockObject ();
```

（3）调用函数 SelectObject 将字体选入设备环境，语法如下：

```
SelectObject(hdc,hF);
```

5.1.2　创建自定义字体

系统提供的字体往往不能满足应用程序的需要，实际上，中文的字体是很丰富的，目前有 40 多种字体。程序员可调用函数 CreateFont 创建自定义字体，该函数的调用形式如下：

```
HFont = CreateFont
    (
    int nHeight,          //字体高度，取 0 则采用系统默认值，使用逻辑单位
    int nWidth,           //字体宽度，取 0 则由系统根据高宽比取最佳值，使用逻辑单位
    int nEscapement,      //每行文字相对于页底的角度，以十分之一度为单位
    int nOrientation,     //每个文字相对于页底的角度，以十分之一度为单位
    DWORD nWeight,        //字体粗细度，取值范围为 0～1000，例如 400 为正常字体，700 为黑体
    DWORD dwItalic,               //如果要求字体倾斜，则取非零
    DWORD dwUnderline,            //如果要求下划线，则取非零
    DWORD dwStrikeout,            //如果要求中划线，则取非零
    DWORD dwCharset,              //字体所属字符集
    DWORD dwOutputPrecision,      //输出精度，一般取默认值 OUT_DEFAULT_PRECIS
    DWORD dwClipPrecision,        //剪裁精度，一般取默认值 CLIP_DEFAULT_PRECIS
    DWORD dwQuality,              //输出质量，一般取默认值 DEFAULT_QUALITY
    DWORD dwPitchAndFamily,       //字体的间距及字体的系列，一般取默认值 DEFAULT_PITCH
    DWORD lpszFacename            //字体名
    )
```

其中参数 dwCharset 可取的系统字符集有：ANSI_CHARSET、OEM_CHARSET、SYMBOL_CHARSET、DEFAULT_CHARSET 和 SHIFTJIS_CHARSET 等。

5.1.3　设置字体和背景颜色

在了解了字体句柄及创建字体以后，在有关文本输出的编程中，还需要进一步了解字体的设置及背景颜色的设置，这样才能得到精美的输出效果。

应用程序通过调用函数 SetTextColor 设置字体颜色。其形式为：

```
SetTextColor(hdc,crColor);//crColor 为设置的颜色。
```

应用程序还可调用函数 SetBkColor 设置背景颜色。其形式为：

```
SetBkColor(hdc,crColor);
```

5.2 文本的输出过程

在定义了字体句柄、字体及字体颜色以后，我们就可以把设置的字体输出到相应的设备上。Windows 应用程序的文本输出过程比较复杂，因为程序员除了要确定输出内容外，还要管理输出的格式，由应用程序完成窗口用户区管理，Windows 系统并不参与窗口用户区的管理。这样虽然为程序员管理用户区提供了编程的自由，但也加重了编写应用程序的负担。例如，在用户区内输出文本，应用程序必须管理换行、后续字符的位置等输出格式，Windows 系统并未提供管理输出文本格式的函数。

文本的输出过程包括获取字体信息、格式化文本、调用函数输出文本等具体过程。

1. 获取字体信息

应用程序在输出文本之前必须获取当前使用字体的有关信息，如当前使用的字符高度等，以确定输出文本格式和下一行字符的输出位置。

Windows 程序中通过调用函数 GetTextMetrics 获取当前使用的字体信息。调用该函数时，系统将当前字体的信息拷贝到一个 TEXTMETRICS 结构的变量中。其形式为：

```
GetTextMetrics (hdc,&tm);     //tm 为 TEXTMETRICS 结构
```

系统定义的 TEXTMETRICS 的结构如下：

```
typedef struct tagTEXTMETRIC
{
    LONG tmHeight;                  //字符高度
    LONG tmAscent;                  //字符基线以上高度
    LONG tmDescent;                 //字符基线以下高度
    LONG tmInternalLeading;         //tmHeight 制订的字符高度顶部的控件
    LONG tmExternalLeading;         //行与行之间的间隔
    LONG tmAveCharWidth;            //平均字符宽度
    LONG tmMaxCharWidth;            //最大字符宽度
    LONG tmWeight;                  //字符的粗细度
    LONG tmOverhang;                //合成字体间附加的宽度
    LONG tmDigitizedAspectX;        //为输出设备设计的 X 轴尺寸
    LONG tmDigitizedAspectY;        //为输出设备设计的 Y 轴尺寸
    BCHAR tmFirstChar;              //字体中第一个字符值
    BCHAR tmLastChar;               //字体中最后一个字符值
    BCHAR tmDefaultChar;            //代替不在字体中字符的字符
    BCHAR tmBreakChar;              //作为分隔符的字符
    BYTE tmItalic;                  //非 0 则表示字体为斜体
    BYTE tmUnderlined;              //非 0 则表示字体有下划线
    BYTE tmStruckOut;               //非 0 则表示字符为删除字体
    BYTE tmPitchAndFamily;          //字体间距和字体族
    BYTE tmCharSet;                 //字符集
```

```
| TEXTMETRICS;
```

调用函数 GetTextMetrics 获取当前字体的 TEXTMETRICS 结构后，即可为其中的成员设置文本输出格式。

2. 格式化文本

格式化处理一般针对两种情况：一是在文本行中确定后续文本的坐标，二是在换行时确定下一行文本的坐标。

(1) 确定后续文本坐标

确定后续文本的坐标，应先获取当前的字符串的宽度，Windows 系统提供函数 GetTextExtentPoint32 完成这项任务，并把它存储于一个 SIZE 结构中。该函数的原型为：

```
BOOL GetTextExtentPoint32
    (
    HDC hdc,
    LPCTSTR lpszString,  //lpszString 为指定的字符串
    int nLength,         //nLength 为字符串中的字符数
    LPSIZE lpSize        //lpSize 为返回字符串宽度及高度的 SIZE 数据结构的地址
    );
```

SIZE 数据结构的定义如下：

```
typedef struct tagSIZE
{
    LONG cx;
    LONG cy;
| SIZE;
```

通过计算字符串的起始坐标与字符串宽度之和，即可得到后续文本的起始坐标。例如，X 轴起始坐标为 cx0，如果当前字符串的尺寸信息存储在 size 指向的 SIZE 结构中，则后续文本的起始坐标 cx1 为：

```
cx1 = cx0 + size.cx;
```

(2) 确定换行时文本坐标

通过计算当前行文本字符的高度与行间隔之和，即可得到换行时文本的起始坐标，而上述两个数值均可通过获取当前字体的信息得到，如果当前字体的信息存储在 tm 指向的 TEXTMETRICS 结构中，则换行时 Y 轴上文本的起始坐标 cy 为：

```
cy = tm.tmHeight + tm.tmExternalLeading;
```

3. 文本输出

Windows 编程中最常用的文本输出函数是 TextOut，其函数原型如下：

```
BOOL TextOut
    (
    HDC hdc,
    int X, int Y,        //(X,Y)为用户区中字符串的起始坐标
    LPCTSTR lpString,    //lpString 为显示的字符串
    int nCount           //nCount 为字符串中的字节数
```

```
    );
```

程序员调用函数 TextOut，以坐标(X,Y)为起点，输出字节数为 nCount、名为 lpstring 的字符串。

5.3 文本输出与字体设置编程实例

【例 5-1】

在用户窗口上输出几行字符串，当窗口接收到 WM_PAINT 消息后，显示的文本每次都被重新刷新。

本例的源程序代码如下：

```
#include<windows.h>
#include<stdlib.h>
#include<string.h>
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam);
BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
HWND hWndMain;

//主函数
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmd-
Line, int nCmdShow)
{
    MSG Message;
    if(! InitWindowsClass(hInstance))
        return FALSE;
    if(! InitWindows(hInstance,nCmdShow))
        return FALSE;
    while(GetMessage(&Message,0,0,0))          //消息循环
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
    return Message.wParam;
}

//消息处理函数
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam)
{
    static long nXChar,nCaps,nYChar;
    HDC hDC;                                   //定义指向设备描述表的句柄
```

```
    short x;
    TEXTMETRIC tm;
    short LnCount = 6;
    PAINTSTRUCT PtStr;                          //定义指向包含绘图信息的结构体变量
    stat    ic char * textbuf[ ] =
    {
    "This is the first line",
    "This is the second line",
    "This is the third line",
    "This is the fourth line",
    "This is the fifth line",
    "This is the sixth line"
    };
    switch(iMessage)                            //处理消息
    {
        case WM_CREATE:                         //处理窗口创建消息
            hDC = GetDC(hWnd) ;                 //获取当前设备描述表句柄
            GetTextMetrics(hDC,&tm);            //获取字体信息
            nXChar = tm.tmAveCharWidth;         //获取字符宽度
            nYChar = tm.tmHeight + tm.tmExternalLeading;
            nCaps = (tm.tmPitchAndFamily&1? 3:2) * nXChar/2;
            ReleaseDC(hWnd,hDC);                //释放当前设备描述表句柄
            return 0;
        case WM_PAINT:                          //处理重画消息
            hDC = BeginPaint(hWnd,&PtStr);      //开始绘画
            for(x = 0;x < LnCount;x = x + 1)
            {                                   //输出文本
              TextOut (hDC,nXChar,nYChar * (1 + x),
                       textbuf[x],lstrlen(textbuf[x]));
            }
            EndPaint(hWnd,&PtStr);
            return 0;
        case WM_DESTROY:                        //结束应用程序
            PostQuitMessage(0);
            return 0;
        default:                                //其他消息处理程序
            return(DefWindowProc(hWnd,iMessage,wParam,lParam)) ;
    }
}

BOOL InitWindowsClass(HINSTANCE hInstance)      //初始化窗口类
```

```
{
    WNDCLASS WndClass;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
    WndClass.hCursor = LoadCursor(NULL,IDC_ARROW);
    WndClass.hIcon = LoadIcon(NULL,"END");
    WndClass.hInstance = hInstance;
    WndClass.lpfnWndProc = WndProc;
    WndClass.lpszClassName = "WinText";
    WndClass.lpszMenuName = NULL;
    WndClass.style = CS_HREDRAW|CS_VREDRAW;
    return RegisterClass(&WndClass);
}

BOOL InitWindows(HINSTANCE hInstance,int nCmdShow)  //初始化窗口
{
    HWND hWnd;
    hWnd = CreateWindow("WinText",                  //生成窗口
                        "文本显示示例程序",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
    if(! hWnd)
        return FALSE;
    hWndMain = hWnd;
    ShowWindow(hWnd,nCmdShow);                      //显示窗口
    UpdateWindow(hWnd);
    return TRUE;
}
```

本例的运行结果如图 5-1 所示。

【例 5-2】

本程序通过在窗口中分五行分别显示五行文本，以说明在窗口的用户区中输出及格式化文本的方法。其中，第一行的文字是红色的；第二行是绿色的；第三行是蓝色的；第四

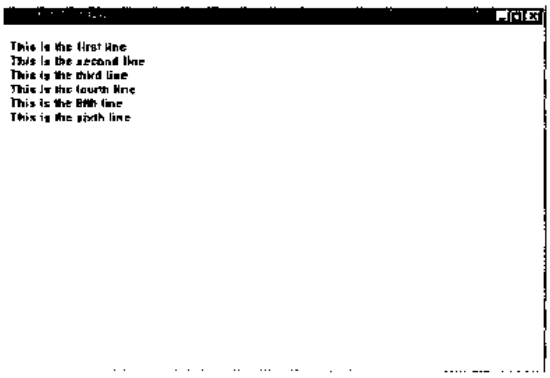


图 5-1　【例 5-1】的运行结果

行使用斜体文字，并带下划线；第五行的文字恢复为红色，但仍使用第四行字体的设置输出，其中最后一行实际上是两个字符串同行输出。本程序的界面效果如图 5-2 所示。

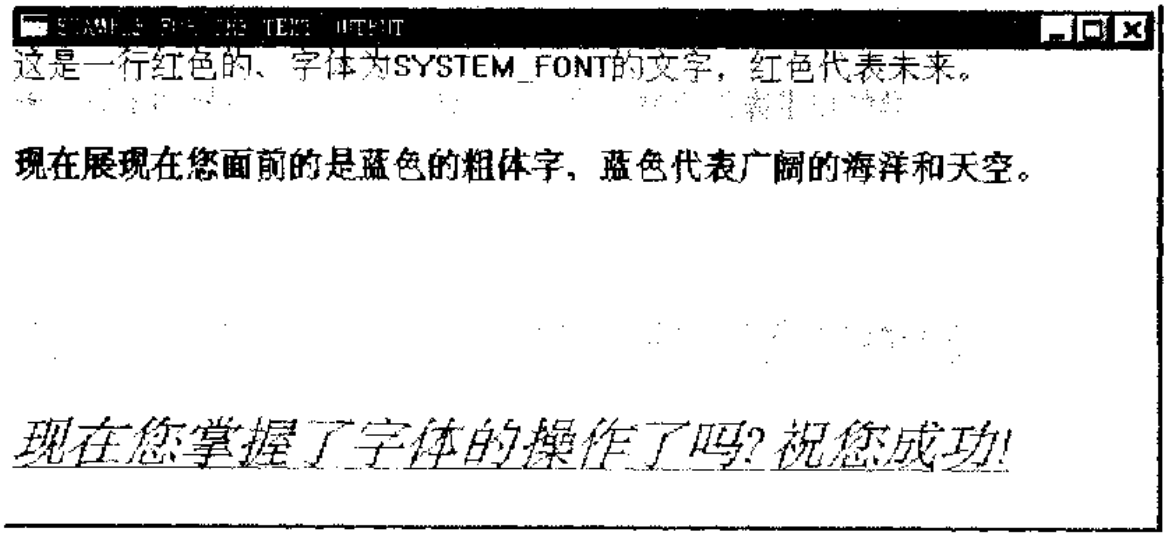


图 5-2　【例 5-2】的执行结果

本例题的源代码如下：

```
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain
        (
        HINSTANCE hInstance,
        HINSTANCE hPrevInst,
        LPSTR lpszCmdLine,
        int nCmdShow
```

```
)

HWND hwnd ;
MSG Msg ;
WNDCLASS wndclass ;
char lpszClassName[ ] = "文本输出";
char lpszTitle[ ] = "EXAMPLE FOR THE TEXT OUTPUT";
wndclass.style = 0;
wndclass.lpfnWndProc = WndProc ;
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon( NULL, IDI APPLICATION) ;
wndclass.hCursor = LoadCursor( NULL, IDC-ARROW) ;
wndclass.hbrBackground = GetStockObject( WHITE-BRUSH) ;
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = lpszClassName ;

if( ! RegisterClass( &wndclass))
{
    MessageBeep(0) ;
    return FAISE ;
}

hwnd = CreateWindow
    (
    lpszClassName,
    lpszTitle,
    WS-OVERLAPPEDWINDOW,
    CW USEDEFAULT,
    CW USEDEFAULT,
    CW USEDEFAULT,
    CW USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
    ) ;

ShowWindow( hwnd, nCmdShow) ;
UpdateWindow(hwnd);
```

```
    while( GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage( &Msg) ;
        DispatchMessage( &Msg) ;
    }

    return Msg.wParam;
}

LRESULT CALLBACK WndProc
    (
    HWND hwnd,
    UINT message,
    WPARAM  wParam,
    LPARAM  lParam
    )

{
    HDC hdc;
    HFONT hF_black,hF_big;  //定义两种字体句柄
    PAINTSTRUCT ps;
    TEXTMETRIC tm;          //定义一个 TEXTMETRIC 结构,用以记录字体信息
    char lpsz_1[] = "这是一行红色的、字体为 SYSTEM_FONT 的文字,红色代表未来。";
    char lpsz_2[] = "现在显示的是自定义字体,颜色为绿色,绿色代表生机勃勃。";
    char lpsz_3[] = "现在展现在您面前的是蓝色的粗体字,蓝色代表广阔的海洋和天空。";
    char lpsz_4[] = "当前字体为大号、斜体并带有下划线的文字。";
    char lpsz_5[] = "现在您掌握了字体的操作了吗?";
    char lpsz_6[] = "祝您成功!";

    int X = 0,Y = 0;
    SIZE size;              //定义一个 SIZE 类型的结构

    switch(message){
        case WM_PAINT:
            hdc = BeginPaint(hwnd,&ps);
            SetTextColor(hdc,RGB(255,0,0));          //设置文本颜色为红色
            GetTextMetrics(hdc,&tm);     //获取默认字体,写入 tm 结构中
            TextOut(hdc,X,Y,lpsz_1,strlen(lpsz_1)); //使用当前字体输出文本
            Y = Y + tm.tmHeight + 100 * tm.tmExternalLeading;
                                         //计算换行时下一行文本的输出坐标
```

```
//创建自定义字体
hF black = CreateFont
    (
    20,                          //字体的高度
    0,                           //由系统根据高宽比选取字体最佳宽度值
    0,                           //文本的倾斜度为 0,表示水平
    0,                           //字体的倾斜度为 0
    FW_HEAVY,                    //字体的粗度,FW_HEAVY 为最粗
    0,                           //非斜体字
    0,                           //无下划线
    0,                           //无删除线
    ANSI_CHARSET,                //表示所用的字符集为 ANSI_CHARSET
    OUT_DEFAULT_PRECIS,          //输出精度为默认精度
    CLIP_DEFAULT_PRECIS,         //剪裁精度为默认精度
    DEFAULT_QUALITY,             //输出质量为默认值
    DEFAULT_PITCH|FF_DONTCARE,          //字间距和字体系列使用默认值
    "粗体字"                            //字体名称
    );

SetTextColor(hdc,RGB(0,255,0));          //设置文本颜色为红色
SelectObject(hdc,hF_black);              //将自定义字体选入设备描述表
GetTextMetrics(hdc,&tm);                 //获取字体的信息,并写入 tm 结
                                         //构中
TextOut(hdc,X,Y,lpsz_2,strlen(lpsz_2));//使用当前字体输出文本
//换行继续输出文本,计算新行的起始 Y 坐标位置
Y = Y + tm.tmHeight + 10 * tm.tmExternalLeading;
GetTextExtentPoint32(hdc,lpsz_2,strlen(lpsz_2),&size);
                                         //获取字符串的宽度
SetTextColor(hdc,RGB(0,0,255));          //设置文本颜色为红色
TextOut(hdc,X,Y,lpsz_3,strlen(lpsz_3));//用当前字体输出文本
X = 0;                                   //计算下一行文本的输出起始
                                         //坐标
Y = Y + tm.tmHeight + 20 * tm.tmExternalLeading;
hF_big = CreateFont                      //引入新字体
      (
      30,                                //字体高度
      0,
      0,
      0,
      FW_NORMAL,
      1,                                 //定义斜体
```

```
                1,                                  //定义输出时带下划线
                0,
                ANSI_CHARSET,                       //所使用的字符集
                OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY,
                DEFAULT_PITCH|FF_DONTCARE,
                "大号字"
                );

            SelectObject(hdc,hF_big);  //将第二种自定义字体选入设备描述表
            SetTextColor(hdc,RGB(155,155,155));    //设置文本颜色为红色
            Y = Y + tm.tmHeight + 20 * tm.tmExternalLeading;
            TextOut(hdc,X,Y,lpsz_4,strlen(lpsz_4));//以当前字体输出文本
            SetTextColor(hdc,RGB(255,0,0));        //设置文本颜色为红色
            X = 0;                                 //设定输出文本的 X 坐标位置
            Y = Y + tm.tmHeight + 30 * tm.tmExternalLeading;
            TextOut(hdc,X,Y,lpsz_5,strlen(lpsz_5));//输出文本
            //在该行继续输出文本
            GetTextExtentPoint32(hdc,lpsz_5,strlen(lpsz_5),&size);
                                                   //获取字符串的宽度
            X = X + size.cx;                       //获取起始坐标
            TextOut(hdc,X,Y,lpsz_6,strlen(lpsz_6));  //输出文本

            EndPaint(hwnd,&ps);
            break;

        case WM_DESTROY:
            DeleteObject(hF_black);              //退出窗口时删除自定义字体
            DeleteObject(hF_big);

            PostQuitMessage(0);
            break;

        default:
            return  DefWindowProc(hwnd,message,wParam,lParam);
    }
return 0;
}
```

习 题

5-1 如何获取字体句柄?

5-2 如何创建自定义字体?

5-3 如何设置字体的颜色和背景色?

5-4 文本是如何输出的?

5-5 设计一个窗口,在窗口中有五行文字,字体分别为楷体、宋体、仿宋体、黑体和自定义字体,字号由 8 到 40 线性增长,每一行的文字相继出现后又消失,而且每一行文字的颜色由 RGB(0,0,0)到 RGB(255,255,255)线性增长。

5-6 编写程序,在某一个窗口上设计一行文字,如"VC 向您问好!"。这一行文字从窗口的左边向右滚动显示,而且每显示一轮,改变一次颜色和字体,一个周期为 4 种颜色,分别为红、绿、黄、蓝,四种字体分别为宋体、楷体、仿宋体和黑体。

第 6 章

Windows 应用程序对键盘与鼠标的响应

键盘和鼠标是 Windows 应用程序中非常重要的输入设备。键盘是一个基本的输入设备，鼠标在 Windows 提供的图形界面中的点击和拖放操作更是极大地方便了用户对应用软件的操作。本章将介绍在采用键盘和鼠标作为应用程序的基本输入设备时所涉及的基本概念和编程原则。

6.1 应用程序对键盘动作的响应

键盘作为输入设备，是 Windows 应用程序的一个十分重要的输入手段。当用户按下或释放一个键时，键盘驱动程序 KEYBOARD.DRV 中的键盘中断处理程序对所按键进行编码，并调用 Windows 的用户模块 USER.EXE 中的有关程序来生成键盘消息，最终发送到应用程序的消息队列中等待处理，而处理这些消息则是由应用程序的窗口过程来具体完成的。

键盘上每一个有意义的键都对应着一个唯一的标识值，我们称之为扫描码。当用户按下或释放某键时，都会产生扫描码，但扫描码是依赖于具体设备的，为达到设备无关性的要求，在应用程序中，往往使用的是与具体设备无关的虚拟码，虚拟码是由 Windows 系统定义的与设备无关的键的标识。

设备驱动程序截取键的扫描码后，把它翻译成虚拟码，这样，由于键盘的输入，就产生了一条消息，它含有扫描码、虚拟码以及其他与按键有关的消息，设备驱动程序就把这些消息放到系统的消息队列中去，Windows 从系统消息队列中取出这条消息，再把它发送到相应的线程消息队列中去，最后，由窗口过程从线程消息队列中取出键盘消息队列，进行必要的后续处理。图 6-1 就显示了这个过程。

虚拟码是一种与设备无关的键盘编码，它的值存放在键盘消息的 wParam 参数中，用以标识哪一个键被按下或释放，最常用的虚拟码已经在 windows.h 中定义，常用的虚拟码如表 6-1 所示。

操作系统在接收到键盘输入后，还要决定哪一个应用程序将响应输入。Windows 操作系统把消息发送给有“输入焦点(input focus)”的窗口。由于某些应用程序有好几个窗口，键盘还必须由该应用程序的众多窗口共享，但当按下某一个键时，却只有一个窗口过程能接收到该键盘消息，接收这个键盘消息的窗口就称为有“输入焦点”的窗口，该窗口应是活动窗口或者活动窗口的子窗口，有“输入焦点”的子窗口是所有输入消息的接收目标。

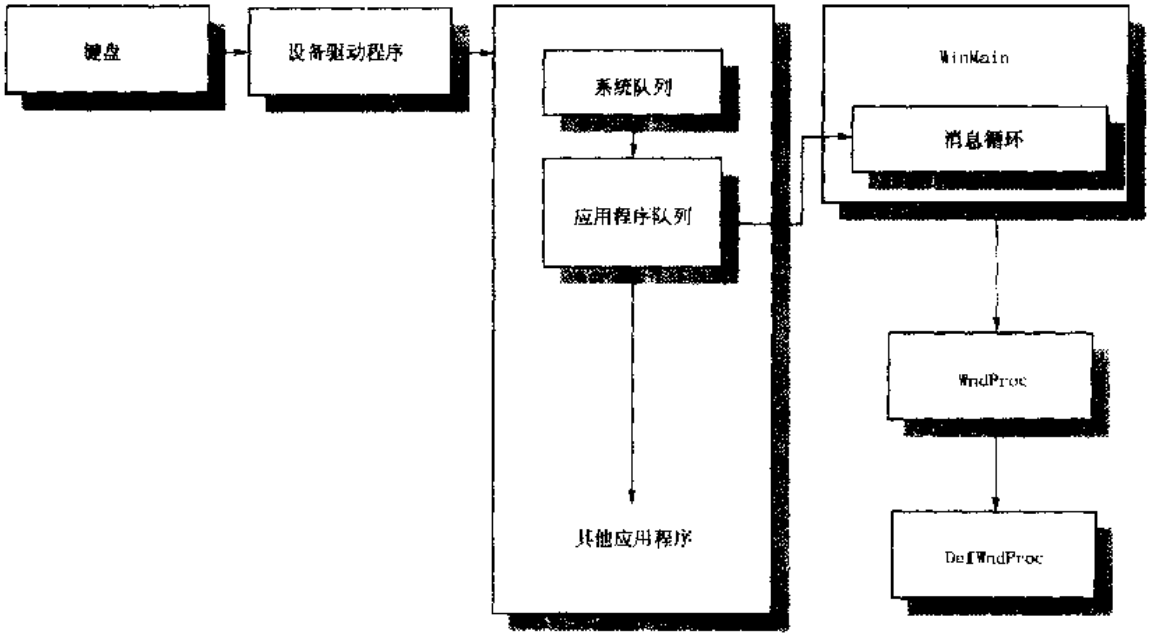


图 6-1　Windows 中处理键盘输入原理示意图

表 6-1　常用的虚拟码

符号常量名称	对应的键盘键或鼠标键	符号常量名称	对应的键盘键或鼠标键
VK_LBUTTON	鼠标左键	VK_BACK	退格键
VK_RBUTTON	鼠标右键	VK_TAB	制表键
VK_MBUTTON	鼠标中键	VK_RETURN	回车键
VK_SHIFT	Shift 键	VK_CONTROL	Ctrl 键
VK_MENU	Alt 键	VK_PAUSE	Pause 键
VK_CAPITAL	CapsLock 键	VK_ESCAPE	Esc 键
VK_PRIOR	PageUp 键	VK_NEXT	PageDown 键
VK_END	End 键	VK_HOME	Home 键
VK_LEFT	左箭头键	VK_RIGHT	右箭头键
VK_UP	上箭头键	VK_DOWN	下箭头键
VK_0 ~ VK_9	0 ~ 9 键	VK_A ~ VK_Z	A ~ Z 键

窗口函数通过捕获 WM_SETFOCUS 和 WM_KILLFOCUS 消息以确定当前窗口是否具有输入焦点，WM_SETFOCUS 表明窗口正在接收输入焦点，而 WM_KILLFOCUS 表示窗口失去输入焦点。

键盘消息可以分成两类，即按键消息和字符消息。每当用户按下或松开一个键时，就产生了一个按键消息。当一个按键的组合产生了一个可以显示的字符时，就产生了一个字符消息。例如，用户如果按下键 H，则将产生两个消息：一个是按下键消息，一个是松开键消息，当然还会产生一个附加的字符消息，因为这个按键消息组合是一个可显示的字符“H”。

按键消息一般又可以分成两类：系统按键消息和非系统按键消息。表 6-2 中列出了这些消息。系统按键消息对应于使用了 Alt 键与相关输入键的组合产生的消息，这些键一般由 Windows 系统内部直接处理，应用程序一般不必处理。如果应用程序处理了这些系统按键消息，还要调用 DefWindowsProc( )函数，以便不影响 Windows 对它们的处理。非系统消息则对应于那些不使用 Alt 键组合的按键消息。

表 6-2　按 键 消 息

消　息	类　型	含　义
WM_KEYDOWN	非系统	按下了非系统键消息
WM_KEYUP	非系统	松开了非系统键消息
WM_SYSKEYDOWN	系统	按下了系统键消息
WM_SYSKEYUP	系统	松开了系统键消息

在按键消息的两个变量 wParam 和 lParam 中包含了许多重要的信息。32 位的变量 lParam 根据其不同的位数表示的含义的不同可以分为以下七个部分：

(1) 重复计数位(0～15 位)：重复计数是消息所代表的按键数目，除键一直被按着外，重复计数是 1。当键按下并一直按着时，窗口过程往往来不及处理这些按键消息，此时 Windows 操作系统就将几个按键消息组合起来，并增加重复计数。

(2) OEM 扫描码(16～23 位)：OEM 扫描码是键盘发送的码值，由于此域是设备相关的，因而此值往往被忽略。

(3) 扩展键标志(24 位)：扩展键标志在有 Alt 键(或 Ctrl 键)按下时为 1，否则为 0。

(4) 保留位(25～28 位)：保留位是系统默认保留的，一般不用。

(5) 关联码(29 位)：关联码用来记录某键与 Alt 键的组合状态，若按下 Alt，当 WM_SYSKEYDOWN 消息送到某个激活的窗口时，其值为 1，否则为 0。

(6) 键的先前状态(位 30)：键的先前状态用于记录先前某键的状态，对于 WM_SYSKEYUP 消息，其值始终为 1。

(7) 转换状态(31 位)：转换状态的消息是始终按着某键所产生的消息，若某键原来是按下的，则其先前状态为 0。转换状态指示键被按下还是被松开。当键被按下时，对应于 WM_SYSKEYDOWN 消息，其值始终为 0，当键被松开时，其转换状态为 1，对应于 WM_SYSKEYUP 消息，其值始终为 1。

按键消息的 wParam 参数包含了识别按下的键的虚键码。键的扫描码是设备相关的，扫描码经过操作系统的转换后称为设备无关的虚键码，这些虚键码定义在 Windows 包含文件中。读者可以在相关的帮助文档中查到虚键码及其对应按键的信息。

在 WinMain 函数的消息循环中包含了 TranslateMessage 函数，它的主要功能是把按键消息转化为字符消息，但只有当键盘驱动程序把键盘字符映像成 ASCII 码后才能产生 WM_CHAR 消息。同样，字符消息也可以分成两类，即系统的和非系统的。表 6-3 中列出了所有的字符消息。

表 6-3 字 符 消 息

消 息	类 型	含 义
WM_CHAR	非系统	非系统字符
WM_DEADCHAR	非系统	非系统死字符
WM_SYSCHAR	系统	系统字符
WM_SYSDEADCHAR	系统	系统死字符

注：死字符是指本身不能显示但能修改其他字符显示的字符。

值得注意的是，WM_KEYDOWN 和 WM_KEYUP 的按键消息只能产生 WM_CHAR 和 WM_DEADCHAR 字符消息，WM_SYSKEYDOWN 和 WM_SYSKEYUP 按键消息只能产生 WM_SYSCHAR 和 WM_SYSDEADCHAR 字符消息。

Windows 支持两种字符集：OEM 和 ANSI。OEM 是 IBM 字符集，在 Windows 中已经很少使用；现在大多使用的是 ANSI 字符集。为了保持程序的兼容性，Windows 提供了几个用于转换这两种字符集的函数，如下所示：

(1) ToOem：将 ANSI 字符串转化为 OEM 字符串。

(2) ToOemBuff：将缓冲区中的 ANSI 字符串转化为 OEM 字符串。

(3) ToAnsi：将 OEM 字符串转化为 ANSI 字符串。

(4) OemToAnsi：将缓冲区中的 OEM 字符串转化为 ANSI 字符串。

在 Windows 操作系统中，使用光标(cursor)来指示当前鼠标的位置，用插字符(caret)指示当前正文位置。插字符是应用程序共享的系统资源，因此，Windows 桌面上只有一个插字符，并且只有拥有“输入焦点”的窗口才能拥有插字符。

6.2 键盘响应编程实例

【例 6-1】

键盘输入示例，本例演示键盘输入时所产生的消息序列，并在窗口的客户区显示对应的字符。

本例在用户窗口区输入“这是一个有关键盘操作的示例程序”的字符串。由于程序中缓冲区大小只有 30 个字符，因此此时缓冲区已满，在这种情况下，若再输入任何字符，就出现如图 6-2 所示的编号为(1)的错误提示；若用户按下键盘上的左箭头键，当光标位置到达本行的起始位置时，由于此时已无法再向左移动，因此就出现编号为(2)的错误信息；若此时按下 Esc 键，就出现编号为(3)的错误提示信息；若当前光标位置处于本行的起始位置，此时按下回退键(BackSpace)，则出现编号为(4)的错误提示信息；若字符缓冲区中已没有任何字符，此时按下了 Delete 键，则出现编号为(5)的错误提示信息。图 6-2 中有 5 个错误提示信息，但实际上，每一次仅出现其中的一个，为了读者方便阅读，把可能出现的错误提示集中在一起了。

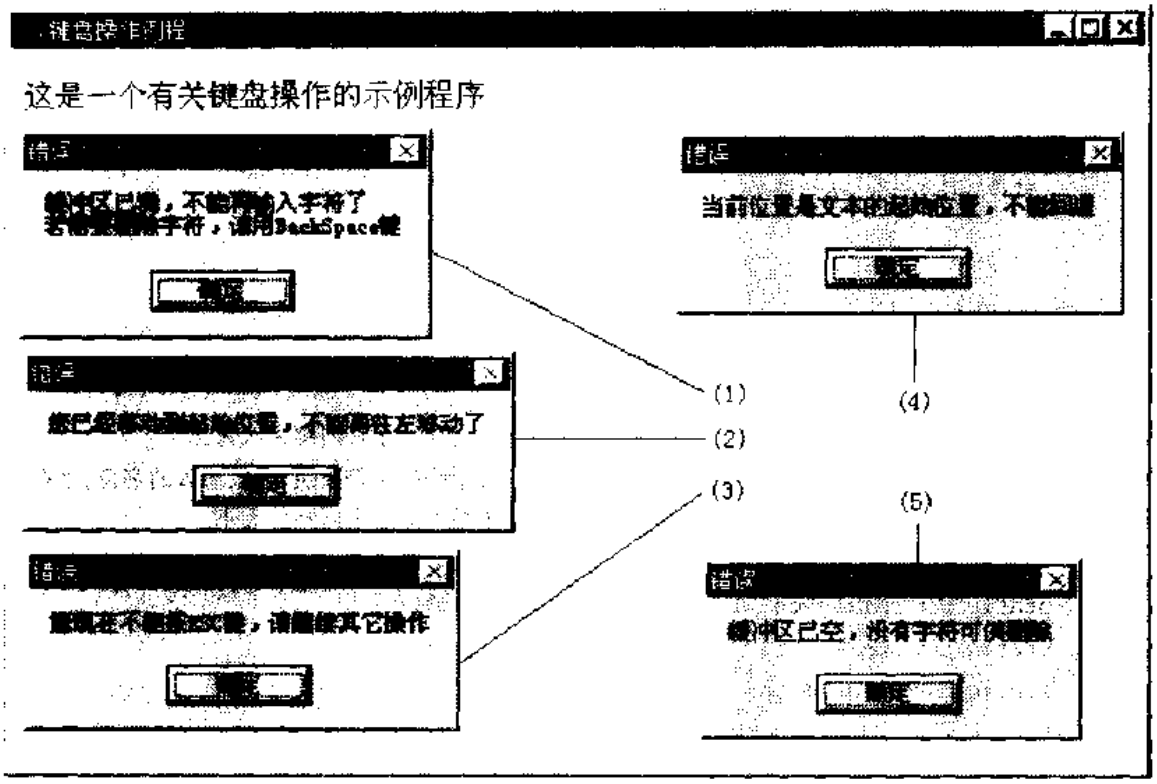


图 6-2　【例 6-1】的运行结果

本例题的源程序代码如下：

```
# include < windows.h >
# include < stdlib.h >
# include < string.h >
void WINAPI CaretPos (HWND hWnd,int nArrayPos,char * cCharBuf,int * xCaret,int *
                      yCaret,int nCharWidth);
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam);
BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
HWND hWndMain;

//主函数
int WINAPI WinMain (HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmd-
                    Line,int nCmdShow)
{
    MSG Message;
    if(! InitWindowsClass(hInstance))
            return FALSE;
    if(! InitWindows(hInstance,nCmdShow))
            return FALSE;
    while(GetMessage(&Message,0,0,0))//消息循环
```

```
    }
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
    return Message.wParam;
}

//消息处理函数
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam)
{
# define BufSize 30                          //设置存放字符的缓冲区大小
    static char cCharBuf[BufSize];           //设置静态字符数组,存放输入的字符,字符
                                             //个数不能超出缓冲区大小
    static int nNumChar = 0;                 //现有字符个数
    static int nArrayPos = 0;                //字符的位置
    static int nLnHeight;
    static int nCharWidth;
    static int xCaret,yCaret;
    int x;
    HDC hDC;
    TEXTMETRIC tm;
    PAINTSTRUCT PtStr;  //定义指向包含绘图信息的结构体变量
    switch(iMessage)  //处理消息
    {
        case WM_CHAR:                        //遇到非系统字符所做的处理
            if(wParam == VK_BACK)            //处理按下回退键消息
            {
                if(nArrayPos == 0) //如果已经在一行文字的开始处,则提示用户"不能回退"
                    MessageBox(hWnd,"当前位置是文本的起始位置,不能回退",NULL,MB_OK);
                else
                {
                    nArrayPos = nArrayPos-1; //每按一次回退键就回退一个字符的位置
                    CaretPos
                            (
                            hWnd,
                            nArrayPos,
                            cCharBuf,
                            &xCaret,
                            &yCaret,
                            nCharWidth
                            );
```

```
                nNumChar = nNumChar-1; //对现有字符总数进行计数
                InvalidateRect(hWnd,NULL,TRUE);
                                //刷新用户区,并向应用程序发送 WM_PAINT 消息
            }
        break;
    }
    if(wParam = = VK_ESCAPE) //处理按下 Escape 键消息
    {
        MessageBox(hWnd,"您现在不能按 ESC 键,请继续其他操作",NULL,MB_OK);
        break;
    }
    if(nNumChar > = BufSize) //如果写入的字符数超过缓冲区大小,则报警
    {
        MessageBox (hWnd,"缓冲区已满,不能再输入字符了 \n 若需要删除字符,
                请用 BackSpace 键",NULL,MB_OK);
        break;
    }
    for(x = nNumChar ;x > nArrayPos;x = x - 1)
        cCharBuf[x] = cCharBuf[x - 1];
    cCharBuf[nArrayPos] = (unsigned char)wParam;
    nArrayPos = nArrayPos + 1;
    nNumChar = nNumChar + 1;
    CaretPos(hWnd,nArrayPos,cCharBuf,&xCaret,&yCaret,nCharWidth);
    InvalidateRect(hWnd,NULL,TRUE);
    break;

case WM_CREATE:                         //处理窗口创建消息
    hDC = GetDC(hWnd);
    GetTextMetrics(hDC,&tm);            //获取字体信息
    nLnHeight = tm.tmHeight + tm.tmExternalLeading;
    nCharWidth = tm.tmAveCharWidth;
    yCaret = nLnHeight;
    ReleaseDC(hWnd,hDC);
    break;
case WM_SETFOCUS:                       //处理活动焦点消息
    CreateCaret(hWnd,0,0,nLnHeight);
    CaretPos(hWnd,nArrayPos,cCharBuf,&xCaret,&yCaret,nCharWidth);
    ShowCaret(hWnd);                    //在活动焦点的窗口中显示插字符
    break;
case WM_KILLFOCUS:                      //处理失去焦点消息
    DestroyCaret();
```

```
        break;
    case WM_KEYDOWN:                          //处理按下键消息
        switch(wParam)
        {
            case VK_END:                      //处理按下键为 End 时的消息
                nArrayPos = nNumChar;         //输入位置从本行的末尾开始

                CaretPos(hWnd,nArrayPos,cCharBuf,&xCaret,&yCaret,
                         nCharWidth);
                break;
            case VK_HOME:                     //处理按下键为 Home 时的消息
                nArrayPos = 0;                //输入位置为本行的起始位置
                CaretPos
                        (
                        hWnd,
                        nArrayPos,
                        cCharBuf,
                        &xCaret,
                        &yCaret,
                        nCharWidth
                        );
                break;
            case VK_DELETE:                   //处理按下键为 Delete 时的消息
                if(nArrayPos == nNumChar)  //输入位置处于本行的末尾
                    MessageBox(hWnd,"缓冲区已空,没有字符可供删除",NULL,MB_OK);
                else
                {
                  for(x = nArrayPos;x < nNumChar;x = x + 1)
                      cCharBuf[x] = cCharBuf[x + 1];
                                          //每删除一个字符,缓冲区中总字符数减 1
                  nNumChar = nNumChar - 1;
                  InvalidateRect(hWnd,NULL,TRUE); //用户区刷新
                }
            break;
            case VK_LEFT: //处理按下左方向键时的消息
                if(nArrayPos > 0)
                {

                nArrayPos = nArrayPos - 1;
                                          //当前输入位置往前移一个位置,再输入字符
                                          //时,等于插入字符
```

```
                    CaretPos(hWnd,nArrayPos,cCharBuf,&xCaret,&yCaret,nChar-
                        Width);
                    }
                    else //已经移到起始输入位置,不能再往前了
                        MessageBox(hWnd,"您已经移动到起始位置,不能再往左移动了",
                            NULL,MB_OK);
                    break;
                case VK_RIGHT://处理按下右方向键时的消息
                    if(nArrayPos < nNumChar)  //如果当前位置没有到缓冲区的最后
                                              位置,还能向右移动
                    {
                        nArrayPos = nArrayPos + 1;

                        CaretPos(hWnd,nArrayPos,cCharBuf,&xCaret,&yCaret,
                            nCharWidth);
                    }
                    else
                        MessageBox(hWnd,"已经到缓冲区的末尾,不能再向右移动了",
                            NULL,MB_OK);
                        break;
            }
            break;
        case WM_PAINT: //处理重画消息
            hDC = BeginPaint(hWnd,&PtStr);
            TextOut(hDC,nCharWidth,nCharHeight,cCharBuf,nNumChar);
                                        //输出缓冲区中的文本
            EndPaint(hWnd,&PtStr);
            break;
        case WM_DESTROY: //处理结束应用程序消息
            PostQuitMessage(0);//结束应用程序
            break;
        default://其他消息处理程序
            return(DefWindowProc(hWnd,iMessage,wParam,lParam)) ;
    }
    return 0;
}

void WINAPI CaretPos(HWND hWnd,int nArrayPos,char *cCharBuf,int *xCaret,int *
                    yCaret,int nCharWidth)    //处理插字符位置的函数
```

```
{
    DWORD dWord;
    HDC hDC;                                        //定义设备描述表句柄
    hDC = GetDC(hWnd);                              //取得当前设备描述表句柄
    ReleaseDC(hWnd,hDC);
    * xCaret = LOWORD(dWord) + nCharWidth;
    SetCaretPos( * xCaret, * yCaret);               //设置插字符位置坐标
}

BOOL InitWindowsClass(HINSTANCE hInstance)//初始化窗口类
{
    WNDCLASS WndClass;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
    WndClass.hCursor = LoadCursor(NULL,IDC_ARROW);
    WndClass.hIcon = LoadIcon(NULL,"END");
    WndClass.hInstance = hInstance;
    WndClass.lpfnWndProc = WndProc;
    WndClass.lpszClassName = "WinKeyboard";
    WndClass.lpszMenuName = NULL;
    WndClass.style = CS_HREDRAW|CS_VREDRAW;
    return RegisterClass(&WndClass);
}

BOOL InitWindows(HINSTANCE hInstance,int nCmdShow)//初始化窗口
{
    HWND hWnd;
    hWnd = CreateWindow("WinKeyboard",              //生成窗口
                        "键盘操作例程",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
    if(! hWnd)
```

```
        return FALSE;
    hWndMain = hWnd;
    ShowWindow(hWnd,nCmdShow);      //显示窗口
    UpdateWindow(hWnd);             //绘制用户区
    return TRUE;
}
```

6.3　应用程序对鼠标动作的响应

鼠标作为一种定位输入设备在 Windows 中得到了广泛的应用，通过鼠标的单击、双击功能和拖动功能，用户可以很容易地操作基于 Windows 图形界面的应用程序。Windows中通过光标来指示当前鼠标的位置，在 Windows 操作系统中预定义了几种光标，并在windows.h头文件中加以定义，这些系统预定义的光标如表 6-4 所示。

表 6-4　系统预定义的光标

代表预定义光标的常量	光标属性描述
IDC_ARROW	箭头光标
IDC_CROSS	十字光标
IDC_IDEAM	I 形文本光标
IDC_ICON	空图标
IDC_SIZE	右下有小方形的方形图标
IDC_SIZENESW	带有指向东北方和西南方箭头的光标
IDC_SIZENS	带有指向北方和南方箭头的光标
IDC_SIZENWSE	带有指向西北方和东南方箭头的光标
ISC_SIZEWE	带有指向东方和西方箭头的光标
IDC_UPARROW	垂直箭头光标
IDC_WAIT	计时光标(也称沙漏光标)

用户也可以通过图形编辑器自定义光标形式，将其保存在扩展名为.cur 的文件中。采用自定义光标时，需要在资源文件中定义光标资源，其形式为：

光标名 CURSOR 光标文件(.cur)

然后应用程序通过调用 LoadCursor 加载光标资源，其形式为：

```
LoadCursor(hThisInst,lpszCursorname);
```

其中，hThisInst 为应用程序当前实例句柄，lpszCursorname 为当前光标，应用程序加载光标资源常在定义窗口类时进行，例如，下面的语句为窗口类 wndclass 定义光标形式：

```
WNDCLASS     wndclass;
...
wndclass.hCursor = LoadCursor (hThisInst, IDC_WAIT);
...
```

此外，还可在应用程序中调用 LoadCursor 函数改变光标形式。

所谓鼠标的单击操作，实际上是指用户按下鼠标按钮并松开的这一全过程。此过程

可以用来选择对象;所谓鼠标的双击操作,实际上是指用户在很短的时间内(根据不同计算机的设置不同而不同,操作系统的默认时间为 0.5 秒)进行两次单击鼠标的操作,此动作可以激活所选项的默认操作;所谓鼠标的拖动操作,实际上是指用户按下鼠标按钮并在不松开鼠标按钮的情况下移动鼠标,此动作一般可以用来选择菜单和移动有关内容。

Windows 操作系统通过鼠标设备驱动程序接收鼠标输入。鼠标驱动程序在启动 Windows时装入,Windows 操作系统通过鼠标驱动程序能检测出鼠标是否存在。如果鼠标已经存在,则设备驱动程序将注意到 Windows 的任何鼠标事件。每当在窗口内有鼠标事件发生时,窗口就接收到一个鼠标事件(以消息的形式发送给应用程序的窗口)。注意,能接收鼠标事件的窗口并不一定要求是活动窗口或者是具有输入焦点的窗口。

当应用程序的用户区内产生一个鼠标事件时,就将产生一个用户区鼠标消息。表 6-5 中列出了所有的用户区鼠标消息。

表 6-5 用户区鼠标消息

消 息	含 义
WM_LBUTTONDOWN	用户区内按下鼠标左键
WM_LBUTTONUP	用户区内松开鼠标左键
WM_LBUTTONDBLCLK	用户区内双击鼠标左键
WM_MBUTTONDOWN	用户区内按下鼠标中键
WM_MBUTTONUP	用户区内松开鼠标中键
WM_MBUTTONDBLCLK	用户区内双击鼠标中键
WM_RBUTTONDOWN	用户区内按下鼠标右键
WM_RBUTTONUP	用户区内松开鼠标右键
WM_RBUTTONDBLCLK	用户区内双击鼠标右键
WM_MOUSEMOVE	鼠标在用户区内移动

在鼠标消息中,参数 lParam 包含了鼠标位置,lParam 字的低字节包含了鼠标位置的 x 坐标值,lParam 字的高字节包含了鼠标位置的 y 坐标值。lParam 所表示的坐标是相当于窗口的左上角为原点的坐标值;参数 wParam 内包含了一个指示各种虚键状态的值。wParam 参数是表 6-6 中所列值的组合。

表 6-6 wParam 的值

值	含 义
MK_CONTROL	按下了键盘上的 Ctrl 键
MK_LBUTTON	按下了鼠标左键,等同 WM_LBUTTONDOWN
MK_MBUTTON	按下了鼠标中键,等同 WM_MBUTTONDOWN
MK_RBUTTON	按下了鼠标右键,等同 WM_RBUTTONDOWN
MK_SHIFT	按下了键盘上的 Shift 键

通过用户区消息的 lParam 和 wParam 参数,程序员就可以确定鼠标的位置和鼠标键的状态。

对于鼠标消息的处理，一般又分为两种，一种是要对 Shift 和 Ctrl 键进行监测，另一种则不监测。

(1) 监测 Shift 键和 Ctrl 键的一般用如下代码：

```
case WM_LBUTTONDOWN:     //按下鼠标左键
    if((wParam&MK_CONTROL)&&(wParam&MK_SHIFT))
    ...                                  //Shift 和 Ctrl 键都被按下
    break;
case WM_LBUTTONUP:          //释放鼠标左键
    ...
    break;
    ...
```

(2) 不监测 Shift 键和 Ctrl 键的一般用如下代码：

```
case WM_LBUTTONDOWN:   //按下鼠标左键
    ...
    break;
case WM_LBUTTONUP:              //释放鼠标左键
    ...
    break;
    ...
```

前面已经谈到，对于鼠标双击，一般设定的双击时间间隔为 0.5 秒，这是 Windows 系统默认的时间间隔。当然，应用程序也可以调用 SetDoubleClickTime()函数来重新设定此值。此外，若要使窗口函数能接收鼠标双击产生的消息，则在注册窗口类时必须注明该窗口类具有 CS_DBLCLKS 属性，定义方式如下：

```
wndclass.style = CS_HREDRAW|CS_VERDRAW|CS_DBLCLKS;
```

若窗口不包含上述属性的定义，那么及时进行了双击操作，该窗口也只能接收到两条 WM_BUTTONDOWN 消息或两条 WN_BUTTONUP 消息。

每当在一个窗口的用户区以外的地方(例如在窗口的菜单、滚动条、工具条和标题条等处)产生了一个鼠标事件，就将产生一个非用户区鼠标消息，对于非用户区鼠标消息，往往不由应用程序进行具体处理，而是送往函数 DefWindowProc 进行处理。

通常情况下，只有当鼠标光标位于某一窗口的用户区或非用户区时，该窗口的窗口函数才能接收到鼠标消息，但是由于鼠标移动的随机性，难以保证光标始终不离开某一个窗口。如果要使某一个窗口能不间断地捕获鼠标消息，就必须对鼠标加以捕获，从而使 Windows 发送的所有鼠标消息均定向到某一个窗口，而不管鼠标的光标处于何处。

调用 SetCapture()函数即可实现对鼠标的捕捉，如 SetCapture(hWnd)；就可以向句柄为 hWnd 的窗口发送所有的鼠标消息。一旦从窗口捕获了鼠标，系统的键盘功能就暂时失效，其他窗口也无法得到鼠标消息，因此，当该窗口不再需要捕获所有的鼠标消息时，应及时调用 ReleaseCapture()函数释放鼠标，以便其他窗口可以正常地接收鼠标信息。

6.4 鼠标响应编程实例

【例 6-2】

鼠标输入示范程序。本例介绍如何响应鼠标信息和改变光标形状。用户在窗口的不同区域移动鼠标时，光标将显示不同的形状，如“十”字形光标、“水平双箭头”光标、“垂直双箭头”光标、“沙漏光标”(如图 6-3 所示)等。

本例的源程序代码如下：

```
#include<windows.h>
#include<stdlib.h>
#include<string.h>
void WINAPI CaretPos(HWND hWnd,int nArrayPos,char * cCharBuf,int * xCaret,int *
                     yCaret,int nCharWidth);
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam);
BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
HWND hWndMain;

//主函数
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,
                   int nCmdShow)
{
    MSG Message;
    if(! InitWindowsClass(hInstance))
        return FALSE;
    if(! InitWindows(hInstance,nCmdShow))
        return FALSE;
    while(GetMessage(&Message,0,0,0))//消息循环
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
    return Message.wParam;
}

//消息处理函数
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam)
{
    WORD x,y;                    //定义表示坐标的变量
    HCURSOR hCursor;             //定义表示鼠标的变量
    switch(iMessage)             //消息处理
    {
```

```
case WM_MOUSEMOVE:                    //处理鼠标移动消息
    x = LOWORD(lParam);               //取得鼠标所在位置在工作区中的坐标值
    y = HIWORD(lParam);
//当在左上角为(50,50),右下角为(400,300)的矩形区域中时,改变鼠标的形状
    if(x > = 50&&x < = 400&&y > = 50&&y < = 300)
    {
        if(x > = 50&&x < = 100&&y > = 50&&y < = 100)
        {
            hCursor = LoadCursor(NULL,IDC_CROSS); //加载一个"十"字型鼠标
            SetCursor(hCursor);//设置当前鼠标为上句中定义的"十"字型鼠标
        }
        if(x > = 100&&x < = 150&&y > = 50&&y < = 100)
        {
            hCursor = LoadCursor(NULL,IDC_SIZE); //加载一个双箭头型鼠标
            SetCursor(hCursor);  //设置当前鼠标为上句中定义的双箭头型鼠标
        }
        if(x > = 150&&x < = 200&&y > = 50&&y < = 100)
        {
            hCursor = LoadCursor(NULL,IDC_SIZE); //加载一个双箭头型鼠标
            SetCursor(hCursor);  //设置当前鼠标为上句中定义的双箭头型鼠标
        }
        if(x > = 50&&x < = 100&&y > = 100&&y < = 150)
        {
            //定义一个用于改变窗口对角线大小的双箭头型鼠标
            hCursor = LoadCursor(NULL,IDC_SIZENESW);
            SetCursor(hCursor);//设置当前鼠标为上句中定义的鼠标
        }
        if(x > = 100&&x < = 250&&y > = 100&&y < = 150)
        {
            //定义鼠标为改变窗口上下大小的双箭头型鼠标
            hCursor = LoadCursor(NULL,IDC_SIZENS);
            SetCursor(hCursor);
        }
        if(x > = 250&&x < = 400&&y > = 100&&y < = 150)
        {
            //加载一个用于改变窗口对角线大小的双箭头型鼠标
            hCursor = LoadCursor(NULL,IDC_SIZENWSE);
            SetCursor(hCursor);
        }
        if(x > = 50&&x < = 100&&y > = 150&&y < = 300)
        {
            //定义一个用于改变窗体左右方向上大小的双箭头型鼠标
            hCursor = LoadCursor(NULL,IDC_SIZEWE);
```

```
                SetCursor(hCursor);
            }
            if(x >= 100&&x <= 250&&y >= 150&&y <= 300)
            {
                //定义一个方向向上的箭头型鼠标
                hCursor = LoadCursor(NULL,IDC_UPARROW);
                SetCursor(hCursor);
            }
            if(x >= 250&&x <= 400&&y >= 150&&y <= 300)
            {
                //设置一个用于表示系统忙的鼠标
                hCursor = LoadCursor(NULL,IDC_WAIT);
                SetCursor(hCursor);
            }
        }
        else
        {
            hCursor = LoadCursor(NULL,IDC_ARROW); //其他区域设置成普通的箭头型鼠标
            SetCursor(hCursor);
        }
        return 0;
    case WM_DESTROY:            //处理结束应用程序消息
        PostQuitMessage(0);     //结束应用程序
        return 0;
    default:                    //其他消息处理程序
        return(DefWindowProc(hWnd,iMessage,wParam,lParam)) ;
    }
}

BOOL InitWindowsClass(HINSTANCE hInstance) //初始化窗口类
{
    WNDCLASS WndClass;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
    WndClass.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
    WndClass.hCursor = LoadCursor(NULL,IDC_ARROW);
    WndClass.hIcon = LoadIcon(NULL,"END");
    WndClass.hInstance = hInstance;
    WndClass.lpfnWndProc = WndProc;
    WndClass.lpszClassName = "WinMouse";
    WndClass.lpszMenuName = NULL;
    WndClass.style = CS_HREDRAW|CS_VREDRAW;
    return RegisterClass(&WndClass);
```

```
}

BOOL InitWindows(HINSTANCE hInstance,int nCmdShow) //初始化窗口
{
    HWND hWnd;
    hWnd = CreateWindow("WinMouse",  //生成窗口
                        "鼠标及光标形状设置示例",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
    if(! hWnd)
        return FALSE;
    hWndMain = hWnd;
    ShowWindow(hWnd,nCmdShow);//显示窗口
    UpdateWindow(hWnd);
    return TRUE;
}
```

本程序的运行结果如图 6-3 所示。

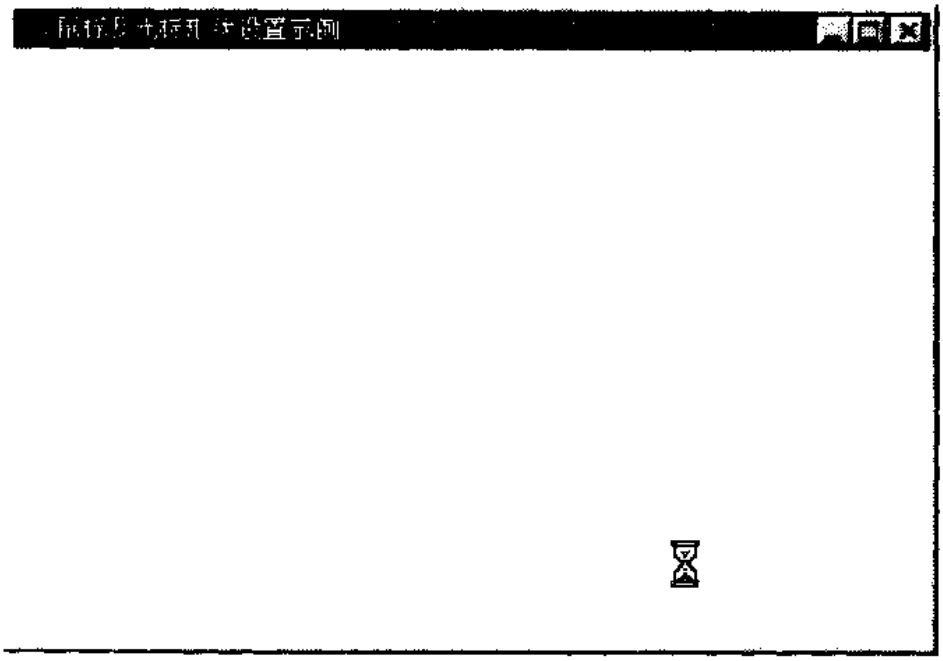


图 6-3　例【6-3】程序运行结果

习 题

6-1 应用程序如何响应键盘消息?

6-2 应用程序如何响应鼠标消息?

6-3 设计一个窗口,在该窗口中练习键盘的响应,要求如下:

(1) 单击键盘上的向上箭头时,屏幕显示"You had hitted the UP key";

(2) 单击 Shift 键时,屏幕显示"You had hitted the SHIFT key";

(3) 单击 Ctrl 键时,屏幕显示"You had hitted the CTRL key";

(4) 单击 Ctrl + A 键时,屏幕显示"You had hitted the CTRL + A key";

(5) 单击 Shift + B 键时,屏幕显示"You had hitted the SHIFT + B key"。

6-4 设计一个鼠标应用程序,当单击鼠标左键时,窗口中显示"LEFT BUTTON";当单击鼠标右键时,窗口中显示"RIGHT BUTTON"。把窗口分成五个区域,这五个区域颜色不同,要求当鼠标在这五个区域移动时,分别显示不同的鼠标样式。

第 7 章

资源在 Windows 编程中的应用

在 Windows 应用程序中可以使用几种不同类型的资源，如加速键、位图、对话框和菜单等。这些资源可以使用 VC ++ 6.0 提供的资源编辑器来实现创建和编辑。资源是 Windows 应用程序用户界面的重要组成部分，资源的使用极大地方便了 Windows 应用程序的界面设计。

7.1 菜单资源、加速键资源及其应用

菜单是 Windows 图形用户界面中窗口的重要组成部分。菜单可使用户直观地了解并方便地使用应用程序所提供的各项功能；使用加速键资源可使菜单的操作更灵活快捷，两种资源往往密不可分。

菜单由以下部分组成：

(1) 窗口主菜单栏(位于窗口的标题栏下方，其菜单项通常为下拉式菜单)；

(2) 下拉式菜单框；

(3) 菜单项热键标识；

(4) 菜单项加速键标识；

(5) 菜单项分隔线。

此外，菜单项前常有选中标志以标识其被选中。

7.1.1 菜单资源的创建过程

创建菜单可分为定义和加载两个步骤：

1. 定义菜单

菜单在资源描述文件中的定义形式为：

```
菜单名 MENU [,选项]
{
    菜单项列表
}
```

菜单的定义格式由四部分组成，它们分别是菜单名、MENU 关键字、载入特性选项和菜单项。

(1) 菜单名：菜单名标识特定的菜单，应用程序通过菜单名加载指定菜单。

(2) MENU 关键字：用来标识资源的性质。

(3) 载入特性选项:用以标识菜单所具有的载入特性,常用的载入特性选项及其说明如表 7-1 所示。

表 7-1 菜单的特性

选 项	说 明
DISCARDABLE	当不再需要菜单时可丢弃
FIXEDP	将菜单保存在内存中的固定位置
LOADONCALL	需要时加载菜单
MOVEABLE	菜单在内存中可移动
PRELOAD	立即加载菜单

(4) 菜单项:菜单项是菜单的组成部分。应用程序在资源描述文件中使用关键字 POPUP 和 MENUITEM 定义菜单项。

① POPUP 语句

POPUP 语句定义弹出式菜单,其形式为:

```
POPUP"菜单项名"[,选项];
```

程序员还可在菜单项名中加入符号"&",以定义该菜单项的热键。例如定义弹出式菜单项"编辑"的形式如下:

```
POPUP"编辑(&E)";
```

该菜单项使用 Alt + E 键作为热键。

菜单项的常用选项及其说明见表 7-2。

表 7-2 菜单常用选项及其说明

选 项	说 明
MENUBARBREAK	菜单项纵向分隔标志
CHECKED	显示选中标志
INACTIVE	禁止一个菜单项
GRAYED	禁止一个菜单项并使其变灰显示

POPUP 定义的弹出式菜单项还包含子菜单。

② MENUITEM 语句

MENUITEM 语句用于定义菜单项,其形式为:

```
MENUITEM"菜单项名"菜单项标识(ID)[,选项]
```

其中:ID 为菜单项标识,即在 WM_COMMAND 消息中系统发送给应用程序的菜单标识值。WM_COMMAND 消息中字参数 wParam 中包含选中菜单项的标识。换言之,应用程序通过菜单项的标识值确认每一个菜单项消息。菜单项标识可为 0 到 65535 之间的任意一个整数,应用程序可自由选择,但每个菜单项的标识必须惟一。标识值常在应用程序头文件中定义。此外,使用下面的语句可创建菜单中的水平分隔符:

```
MENUITEM  SEPARATOR
```

例如,应用程序需要在其名为 Menu 的窗口菜单中创建一个"文件"弹出式菜单。该菜单含有名称为"新建"、"打开"、"关闭"、"保存"、"另存为"及"退出"等菜单项。要求应用

程序运行时系统可根据需要调整该菜单在内存中的位置,以提高内存的利用率;菜单项均使用热键;并且“退出”项与其他菜单项之间用分隔线分开。该菜单在资源描述文件中的定义如下:

```
#include <windows.h>
#include "Menu.h"
//菜单定义
Menu MENU MOVEABLE                                    //Menu 为窗口菜单的名称
{
    POPUP"文件(&F)"                                   //定义"文件"弹出式菜单
    {
        MENUITEM"新建"( &N)",IDM_NEW
        MENUITEM"打开"( &O)",IDM_OPEN
        MENUITEM"关闭"( &C)",IDM_CLOSE
        MENUITEM"保存"( &S)",IDM_SAVE
        MENUITEM"另存为"( &A)",IDM_SAVEAS
        MENUITEM  SEPARATOR                           //分隔线
        MENUITEM"退出(&X)",IDM_EXIT
    }
}
```

Menu.h 文件中定义了菜单项标识所对应的数值:

```
.....
#define    IDM_NEW          10
#define    IDM_OPEN         11
#define    IDM_CLOSE        12
#define    IDM_SAVE         13
#define    IDM_SAVEAS       14
#define    IDM_EXIT         15
...
```

2. 加载菜单资源

在 Windows 应用程序中加载菜单的方法有如下三种:

(1) 在窗口类中加载菜单

在窗口类的定义中加载菜单资源通过下列语句实现:

```
...
WNDCLASS  wndclass;
...
wndclass.lpszMenuName = lpszMenuName;
...
```

应用程序在窗口类中加载菜单后,该类窗口将使用此菜单作为默认菜单。

(2) 在创建窗口时加载菜单

应用程序也可在调用函数 CreateWindow 创建窗口时加载窗口菜单。这时,应用程序需首先调用函数 LoadMenu 加载菜单。该函数的原型为:

```
HMENU LoadMenu
    (
    hInstance,      // hInstance 为当前程序的实例句柄
    lpszMenuName    // lpszMenuName 为窗口菜单名
    );
```

例如,应用程序希望在创建的窗口中加载名称为 Menu 的窗口菜单,其形式为:

```
…
HWND hwnd;
HMENU hmenu;
…
hmenu = LoadMenu(hInstance,"Menu");
hwnd = CreateWindow( …,…,…,…,
                     …,…,…,…,
                     hmenu, …,…,);
```

如果创建的窗口希望使用所属窗口类的默认菜单,则应设置函数 CreateWindow 的菜单句柄参数为 NULL。

(3) 动态加载菜单

应用程序调用函数 LoadMenu 获取菜单句柄后,还可通过调用函数 SetMenu 动态地加载菜单,以提高应用程序的灵活性。该函数的原型为:

```
BOLL SetMenu
(
HWND hwnd,      // Hwnc 为窗口句柄
HMENU hmenu     // hmenu 为菜单句柄
);
```

动态加载菜单可在同一个应用程序中实现多种不同的菜单操作界面。例如应用程序定义了两种菜单 Menu1 和 Menu2 及相应的两套菜单项的功能实现。通过调用函数 SetMenu 在两个菜单之间切换,该应用程序可实现两种界面的操作及相应的菜单项功能,程序段如下:

```
HMENU hmenu1;
HMENU hmenu2;
…
hmenu = LoadMenu(hInstance,"Menu1");//应用程序定义窗口时使用 Menu1 为窗口初始菜单
hwnd = CreateWindow( …,…,…,…,
                     …,…,…,…,
                     hmenu1, …,…,);
…

Hmenu2 = LoadMenu(hInstance,"Menu2"); //应用程序将窗口菜单切换为 Menu2
SetMenu(hwnd,hmenu2);
…
```

7.1.2　菜单项的操作

创建菜单后，应用程序可调用 API 函数访问菜单项的属性，或编辑菜单项，实现对菜单的动态控制，包括禁止或激活菜单项，设置或取消选中标志、增加、删除或修改菜单项等操作。

1. 禁止或激活菜单项

应用程序创建菜单时，通过在资源描述文件中设定菜单项的选项以指定该菜单项的初始状态为禁止或激活，或调用函数 EnableMenuItem 改变其初始状态，该函数的原型为：

```
DWORD EnableMenuItem(HMENU hmenu, UINT wIDEnableItem,UINT dwEnable)
```

其中：wIDEnableItem 为被禁止或激活的菜单项标识，根据 dwEnable 的取值，可能为该菜单项的 ID 值，也可能为该菜单项在菜单中的位置；dwEnable 为菜单项操作标识，常用标识及其说明见表 7-3。

表 7-3　EnableMenuItem 函数中 dwEnable 参数的操作标识及其说明

标　识	说　明	标　识	说　明
MF_BYCOMMAND	表明以 ID 值标识菜单项	MF_ENABLED	激活菜单项
MF_BYPOSITION	表明以位置标识菜单项	MF_GRAYED	禁止菜单项并使其变灰显示
MF_DISABLED	禁止菜单项		

例如，禁止弹出式菜单“文件”中的“打开”项的形式如下：

```
EnableMenuItem(hmenu, IDM_OPEN, MF_BYCOMMAND | MF_DISABLED);
```

2. 设置或取消选中标志

应用程序可在菜单旁显示一个选中标志，如打上“√”标记，以表明用户选择了该项。除在资源描述文件中设置菜单项为 CHECKED 外，应用程序还可通过调用函数 CheckMenuItem 设置或取消选中标志，该函数的原型为：

```
DWORD CheckMenuItem
    (
    HMENU hmenu,
    UINT wIDCheckItem,      //wIDCheckItem 为设置或取消选中标志的菜单项标识
                            //根据 dwCheck 的取值，可能为该菜单项的 ID 值，也可能
                            //为该菜单项在菜单中的位置
    UINT dwCheck            // dwCheck 为操作标识，常用标识及其说明见表 7-4
    );
```

表 7-4　CheckMenuItem 函数中 dwCheck 参数的常用标识及其说明

标　识	说　明	标　识	说　明
MF_BYCOMMAND	表明以 ID 值标识菜单项	MF_CHECKED	添加选中标志
MF_BYPOSITION	表明以位置标识菜单项	MF_UNCHECKED	删除选中标志

3. 增加菜单项

程序员可在应用程序中通过两种形式动态地增加菜单项：

(1) 在菜单的尾部增加菜单项

应用程序可调用函数 AppendMenu 在菜单的尾部增加菜单项,该函数的原型为:

```
BOOL AppendMenu
    (
    HMENU hmenu,
    UINT dwFlags,          // dwFlags 为新加入的菜单项类型标识或其他信息
    UINT dwIDNewItem,      // dwIDNewItem 为新加入菜单项的标识
    LPCTSTR lpNewItem      //新加入的菜单项内容,取决于 dwFlags 参数
    );
```

值得注意的是,dwIDNewItem 一般情况下是插入项的 ID 值。如果加入的是一个弹出式菜单,则该参数为弹出式菜单的句柄。lpNewItem 取决于 dwFlags 参数,一般情况下为新加入菜单项的名称。如果 dwFlags 为 MF_BITMAP,则该参数包含一个位图句柄。

例如在弹出式菜单"文件"的末尾增加一项"关于"的形式如下:

```
AppendMenu (hmenu, MF_ENABLED, IDM_ABOUT,"关于(&A)");
```

(2) 在菜单中插入菜单项

应用程序也可调用函数 InsertMenu 在菜单中插入新的菜单项,该函数的原型为:

```
BOOL InsertMenu
    (
    HMENU hmenu,           //菜单句柄
    UINT wPosition,        //指定新菜单项插入的位置
    UINT dwFlag,           //新加入的菜单项的信息及对 wPosition 的解释
    UINT dwIDNewItem,      //新加入的菜单项的标识
    LPCTSTR lpNewItem      //新插入的菜单项的内容
    );
```

对于上面的函数原型,wPosition 由参数 dwFlag 解释其意义,如果 dwFlag 为 MF_BYCOMMAND,则该参数为插入位置的下一个菜单项的 ID 值;如果 dwFlag 为 MF_BYPOSITION,则该参数为插入的位置号,菜单的第一个菜单项的位置号为 0。dwIDNewItem 一般情况下是插入项的 ID 值。如果加入的是一个弹出式菜单,则该参数为弹出式菜单的句柄;lpNewItem 取决于 dwFlags 参数,一般情况下为新插入菜单项的标识,如果 dwFlag 为 FM_BITMAP,则该参数包含一个位图句柄。

例如,在弹出式菜单"文件"的"退出"项之前加入新的菜单项"打印"的语句如下:

```
InsertMenu (hmenu, IDM_EXIT, MF_BYCOMMAND |MF_ENABLED,
            IDM_PRINT,"打印(&P)");
```

4. 删除菜单项

应用程序可调用函数 DeleteMenu 删除菜单项,该函数的原型为:

```
BOOL DeleteMenu
    (
    HMENU hmenu,
    UINT wPosition,        //指定要删除的菜单项的位置
    UINT dwFlag            //对 wPosition 的解释
```

```
);
```

对于 wPosition，由参数 dwFlag 解释其意义，如果 dwFlag 为 MF-BYCOMMAND，则该参数为菜单项的 ID 值；如果 dwFlag 为 MF BYPOSITION ，则该参数为菜单项的位置号；

例如，删除弹出式“文件”菜单中的“另存为”项的形式如下：

```
DeleteMenu (hmenu, IDM-SAVEAS, MF-BYCOMMAND)
```

值得注意的是，如果菜单项含有弹出式菜单，则删除该菜单项时该弹出式菜单也同时被删除。

5. 修改菜单项

应用程序可调用函数 ModifyMenu 修改菜单中的某个项，该函数原型为：

```
BOOL ModifyMenu
    (
    HMENU hmenu,
    UINT wPosition,          //指定需修改的菜单项位置
    UINT dwFlag,
    UINT dwIDNewItem,        // 一般为修改后菜单项的标识
    LPCTSTR lpNewItem        //一般为修改后的菜单项名
    );
```

对于 wPosition，如果 dwFlag 为 MF_BYCOMMAND，则该参数为菜单项的 ID 值；如果 dwFlag 为 MF-BYPOSITION，则该参数为菜单项的位置号。

例如，修改弹出式菜单“文件”中“打开”项为“加载”项的语句如下：

```
ModifyMenu (hmenu, IDM-OPEN, MF BYCOMMAND , IDM-LOAD"加载(&L)");
```

应用文件除使用资源描述文件中定义的菜单外，还可以动态地创建菜单。

7.1.3　菜单的动态创建

动态创建菜单可以使系统资源更加节省，在应用程序中动态创建菜单分两个步骤：

(1) 调用函数 CreateMenu 创建空的弹出式菜单。CreateMenu 函数的原型如下：

```
HMENU CreateMenu (void)
```

(2) 调用函数 AppendMenu 或 InsertMenu 在该菜单中加入菜单项。调用函数 AppendMenu 或 InsertMenu 的形式为：

```
AppendMenu
    (
    hmenu,                  //已有的菜单句柄
    MF_POPUP,               //创建的新弹出式菜单句柄
    (UINT )hmenuPopup,      //该弹出式菜单的名字
    lpszTitle
    );
```

例如，在应用程序的窗口菜单中动态创建弹出式菜单“编辑”的过程如下：

```
...
HMENU  hmenu,//主窗口菜单句柄
hPopupmenu; //创建新的菜单句柄
```

```
...
定义菜单句柄
hPopupmenu:CreateMenu();
...
//在弹出式菜单中增加菜单项
AppendMenu(hPopupmenu,MF_STRING,IDM_COPY,"复制(&E)");
//将弹出式菜单加入到菜单中
AppendMenu(hmenu,MF_POPUP,(UINT)hmenuPopup,"编辑(&E)");
...
```

7.1.4 加速键资源的创建过程

加速键资源是常伴随菜单使用的一种非常有用的资源,创建加速键的步骤如下:

1. 在资源描述文件中定义加速键资源

在资源描述文件中加速键资源的定义形式与菜单定义相似:

```
加速键表名 ACCELERATORS
{
  加速键的定义
}
```

加速键定义的格式为:

```
键名,加速键标识(ID),[类型][NOINVERT][ALT][SHIFT][CONTROL]
```

其中:

- 加速键标识:与所表示的菜单项标识相同的标识值;
- 类型:标识该键为标准键还是虚拟键;
- NOINVERT:表示当使用加速键时,菜单项不高亮度显示;
- ALT,SHIFT, CONTROL:表示组合键的组合方式。

常用的加速键有两种形式:

① "^char",id

与 Ctrl 键组合的加速键。例如"文件"菜单中"保存"项的加速键可定义为:

```
"^S", IDM_SAVE
```

② nCode, id VIRTKEY

使用虚拟键作为加速键。虚拟键是系统提供与设备无关的键码,如键盘上的 F 功能键、方向键、Delete 键等。Windows.H 文件中以 VK 开头定义虚拟键,如 VK_F1,VK_DELETE 等。例如将 F1 键定义为"帮助"菜单项的加速键,其形式如下:

```
VK_F1,IDM_HELLP,VIRKEY
```

下面是资源描述文件对名为 Menu 的窗口菜单项的加速键定义:

```
#include<windows.h>
#include"Menu.h"
...
//菜单定义
...
```

```
//加速键表定义
Menu ACCELERATORS //加速键表名为"Menu"
{
    "^N", IDM_NEW
    "^O", IDM_OPEN
    "^S", IDM_SAVE
}
```

2. 加载加速键资源

在应用程序定义加速键资源句柄后，即可通过调用函数 LoadAccelerators 加载加速键资源，其形式为：

```
...
HACCEL hAccel;
...
hAccel = LoadAccelerators
    (
    hInstance,          // hInstance 为当前程序实例句柄
    lpAccelName         // lpAccelName 为加速键表名
    );
```

3. 翻译加速键

应用程序使用加速键的目的是快捷地切换到需要的菜单项，因此，应用程序必须完成加速键消息到菜单消息的翻译。该翻译操作经常在应用程序的消息循环中进行，其形式如下：

```
while(GetMessage(&Msg,NULL,0,0))
{
    if(! TranslateAccelerators (hwnd,hAccel,&Mag))
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
}
```

函数 TranslateAccelerators 是翻译操作的核心，该函数的原型为：

```
int TranslateAccelerators
    (
    HWND hwnd,              //为窗口句柄
    HACCEL  hAccel,         //为加速键表句柄
    lpMSG lpMsg             //为指向 MSG 结构的指针
);
```

函数 TranslateAccelerators 的作用是对照加速键表，将相关的按键消息 WM_KEYDOWN 和 WM_KEYUP 翻译成 WM_COMMAND 或 WM_SYSCOMMAND 消息。其特点是将翻译后的 WM_COMMAND 或 WM_SYSCOMMAND 消息直接发往窗口，而不在消息队列中等待。消息

翻译完成后，函数返回非 0 值。

7.1.5 菜单与加速键应用实例

【例 7-1】

菜单资源及其创建。本例创建一个通用的窗口菜单的构架，用户可通过选择“文件”菜单中的“创建统计计算菜单项”选项动态地创建主菜单中的“统计计算”菜单项。图 7-1，是一个基本的菜单构架，图 7-2 是动态创建“统计计算”菜单后的界面。

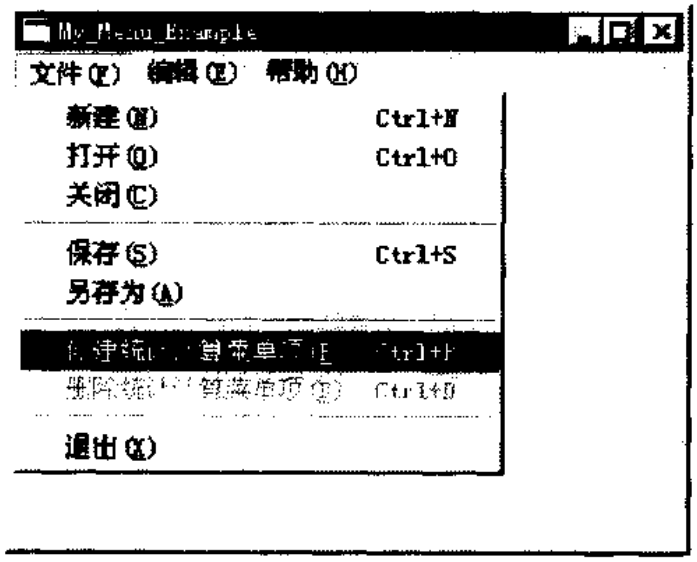


图 7-1 【例 7-1】的运行界面

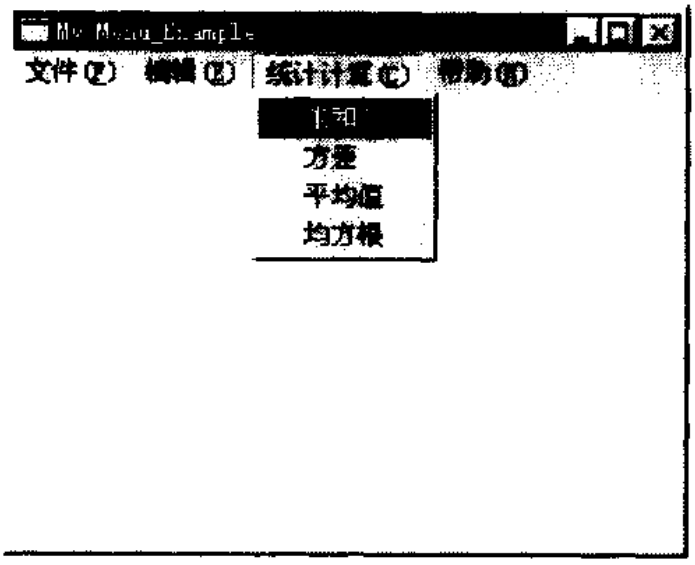


图 7-2 动态创建的“统计计算”菜单

本程序的源代码如下：

```
//Menu.c
```

```
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define IDM_NEW             10
#define IDM_OPEN            11
#define IDM_CLOSE           12
#define IDM_SAVE            13
#define IDM_SAVEAS          14
#define IDM_ADDPRINTMENU    15
#define IDM_DELPRINTMENU    16
#define IDM_EXIT            17
#define IDM_CUT             18
#define IDM_COPY            19
#define IDM_PASTE           20
#define IDM_DELETE          21
#define IDM_HELP            22
#define IDM_LINE            23
#define IDM_RECT            24
#define IDM_RRECT           25
#define IDM_ELLIP           26

HMENU hmenu,hprintmenu; //定义菜单句柄
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

int WINAPI WinMain
    (
    HINSTANCE hInstance,
    HINSTANCE hPrevInst,
    LPSTR lpszCmdLine,
    int nCmdShow
    )

{
    HWND hwnd ;
    MSG Msg ;
    HACCEL hAccel;
    WNDCLASS wndclass ;
    char lpszMenuName[] = "Menu";
    char lpszClassName[] = "菜单";
    char lpszTitle[] = "My-Menu-Example";
```

```
wndclass.style = 0;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION);
wndclass.hCursor = LoadCursor( NULL, IDC_ARROW);
wndclass.hbrBackground = GetStockObject( WHITE_BRUSH);
wndclass.lpszMenuName = lpszMenuName ; //加载窗口类菜单
wndclass.lpszClassName = lpszClassName ;

if( ! RegisterClass( &wndclass))
{
    MessageBeep(0) ;
    return FALSE ;
}

hwnd = CreateWindow
              (
              lpszClassName,
              lpszTitle,
              WS_OVERLAPPEDWINDOW,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              NULL,
              NULL,
              hInstance,
              NULL
              );

ShowWindow( hwnd, nCmdShow) ;
UpdateWindow(hwnd);

//加载加速键资源
hAccel = LoadAccelerators(hInstance,lpszMenuName);

while( GetMessage(&Msg, NULL, 0, 0))
{
    //在消息循环中截获加速键消息
```

```
        if (! TranslateAccelerator(hwnd,hAccel,&Msg))
        {
        TranslateMessage( &Msg) ;
        DispatchMessage( &Msg) ;
        }
    }

    return Msg.wParam;
}

LRESULT CALLBACK WndProc
    (
    HWND hwnd,
    UINT message,
    WPARAM  wParam,
    LPARAM  lParam
    )

{
    switch(message)
    {
    //处理菜单消息
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDM_NEW:
            break;
        case IDM_OPEN:
            break;
        case IDM_SAVE:
            break;
        case IDM_SAVEAS:
            break;
        case IDM_CLOSE:
            break;
              //在主菜单栏中添加"统计计算"菜单
        case IDM_ADDPRINTMENU:
              //获取主菜单句柄
            hmenu = GetMenu(hwnd);
              //动态创建菜单
            hprintmenu = CreateMenu();
              //在创建的菜单中增加菜单项
```

```
            AppendMenu(hprintmenu,MF_ENABLED,IDM_LINE,"求和");
            AppendMenu(hprintmenu,MF_ENABLED,IDM_RECT,"方差");
            AppendMenu(hprintmenu,MF_ENABLED,IDM_RRECT,"平均值");
            AppendMenu(hprintmenu,MF_ENABLED,IDM_ELLIP,"均方根");
              //将创建的弹出式菜单插入主菜单中
            InsertMenu(hmenu,2,MF_POPUP|MF_BYPOSITION,(UINT)hprintmenu,
                       "统计计算(&C)");
              //相应改变菜单中有关统计计算单项的属性
            EnableMenuItem(hmenu,IDM_ADDPRINTMENU,MF_GRAYED);
            EnableMenuItem(hmenu,IDM_DELPRINTMENU,MF_ENABLED);
            DrawMenuBar(hwnd);                          //重新显示窗口菜单
            break;

        //从主菜单中删除弹出式统计计算菜单
    case IDM_DELPRINTMENU:
        DeleteMenu(hmenu,2,MF_BYPOSITION);  //删除统计计算菜单项
        //相应改变"文件"菜单中有关统计计算菜单项的属性
        EnableMenuItem(hmenu,IDM_ADDPRINTMENU,MF_ENABLED);
        EnableMenuItem(hmenu,IDM_DELPRINTMENU,MF_GRAYED);
        DrawMenuBar(hwnd);                          //重新显示窗口菜单
        break;
    case IDM_EXIT:
        //选择"退出"菜单项时,向应用程序发出 WM_DESTROY 消息
        SendMessage(hwnd,WM_DESTROY,0,0);
        break;
    case IDM_CUT:
        break;
    case IDM_COPY:
        break;
    case IDM_PASTE:
        break;
    case IDM_DELETE:
        break;
    }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
```

```
}
```

本例程用到的资源文件源代码如下：

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
#include "menu_exam.h"
/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Chinese (P.R.C.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)
#ifdef _WIN32
LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED
#pragma code_page(936)
#endif //_WIN32
/////////////////////////////////////////////////////////////////////////////
// Menu
//

MENU MENU DISCARDABLE
BEGIN
    POPUP "文件(&F)"
    BEGIN
        MENUITEM "新建(&N)\t Ctrl + N",          IDM_NEW
        MENUITEM "打开(&O)\t Ctrl + O",          IDM_OPEN
        MENUITEM "关闭(&C)",                     IDM_CLOSE
        MENUITEM SEPARATOR
        MENUITEM "保存(&S)\t Ctrl + S",          IDM_SAVE
        MENUITEM "另存为(&A)",                   IDM_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "创建统计计算菜单项(&P)\t Ctrl + P", IDM_ADDPRINTMENU
        MENUITEM "删除统计计算菜单项(&D)\t Ctrl + D", IDM_DELPRINTMENU, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "退出(&X)",                     IDM_EXIT
```

```
    END
    POPUP "编辑(&E)"
    BEGIN
        MENUITEM "剪切(&T)\tCtrl+X",          IDM_CUT
        MENUITEM "复制(&C)\tCtrl+C",          IDM_COPY
        MENUITEM "粘贴(&P)\tCtrl+V",          IDM_PASTE
        MENUITEM "删除(&D)\tDel",             IDM_DELETE
    END
        MENUITEM "帮助(&H)",                  IDM_HELP
END

//////////////////////////////////////////////////////////////////////////////
// Accelerator
//

MENU ACCELERATORS MOVEABLE PURE
BEGIN
    "^N",           IDM_NEW,            ASCII
    "^O",           IDM_OPEN,           ASCII
    "^S",           IDM_SAVE,           ASCII
    "^P",           IDM_ADDPRINTMENU,   ASCII
    "^D",           IDM_DELPRINTMENU,   ASCII
    "^X",           IDM_CUT,            ASCII
    "^C",           IDM_COPY,           ASCII
    "^V",           IDM_PASTE,          ASCII
    VK_DELETE,      IDM_DELETE,         VIRTKEY
END
```

该程序的实现步骤如下：

(1) 调用函数 GetMenu 获取窗口主菜单的句柄。该函数的原型如下：

```
int  GetMenu(HWND hwnd);
```

(2) 应用程序按照前述的过程建立新菜单、加入菜单项并插入到窗口的主菜单的指定位置。在创建新的菜单后，应用程序还通过调用函数 EnableMenuItem 禁止“创建统计计算菜单项”并将其暗淡显示。

(3) 调用函数 DrawMenuBar 重新显示改变后的窗口主菜单。该函数的原型如下：

```
BOOL DrawMenuBar(HWND hwnd);
```

创建“统计计算”菜单项后，可通过选择“文件”下拉菜单中的“删除统计计算菜单项”删除创建的“统计计算”菜单项。此时，应用程序调用函数 DeleteMenu 删除该菜单项，并调用函数 EnableMenuItem 恢复“创建统计计算菜单项”的属性。此外，该程序在响应“文件”菜单中的“退出”命令时还使用了函数 SendMessage。函数 SendMessage 在应用程序中经常使用，其功能是向窗口发送消息。该函数的原型为：

```
LONG SendMessage(HWND hwnd, UINT Msg, DWORD dwParam, LONG lParam);
```

其中：Msg 为发送的消息；dwParam 和 lParam 均为消息的附加信息。

7.2　位图资源及其应用

7.2.1　位图的概念

位图是一种数字化的图形表示形式，是表示一个图像目标的系列数据。应用程序使用位图能很快地将预先定义好的物体画到屏幕上。位图中的每个像素点由位图文件中的一位或多位数据表示。整个位图的信息被细化为每个像素点的属性值。

与设备相关的位图是与特定的显示设备相联系的，这种位图的位和显示输出设备的像素之间关系较密切。与设备无关的位图与特殊的显示设备之间的关系较松散，这种位图表示的是图像的外形而不是位图的位与输出设备像素之间的关系。

对于绘画或照片一类的位图数据量一般较大，因此为了提高显示刷新速度，位图操作须在内存中进行。用于位图操作的系统设备环境为内存设备环境。应用程序首先要通过调用函数 CreateCompatibleDC 向系统申请获取内存设备环境。其形式为：

```
hdcmem = CreateCompatibleDC(hdc);
```

与设备环境相似，内存设备环境也有设备描述表。应用程序获取内存设备环境后，调用函数 SelectObject 将位图文件内容选入内存设备环境之后，即可直接在内存设备环境中操作位图，如绘制图形及编辑等。操作位图结束后，应用程序必须调用 DeleteDC 释放内存设备环境，其形式为：

```
DeleteDC(hdcmem);          // hdcmem:内存设备环境句柄
```

7.2.2　位图资源的创建过程

位图操作过程包括定义、加载或创建、选入内存设备环境和输出。

1. 定义位图句柄

定义一个位图句柄，其形式为：

```
HBITMAP  hBm;
```

2. 加载或创建位图

应用程序调用函数 LoadBitMap 加载位图并获得位图的句柄，其形式为：

```
hBm = LoadBitMap
    (
    hInstance,          // hInstance 为当前应用程序实例句柄
    lpszName            // lpszName 为位图名称
    );
```

此外，应用程序还可通过调用函数 CreateCompatibleBitmap 创建位图。其形式为：

```
hBm = CreateCompatibleBitmap
    (
    hdc,
    nWidth,       //位图宽度
      nHeight     //位图高度
    );
```

应用程序经常通过响应消息 WM_CREATE 完成加载或创建位图的操作，其一般形式为：

```
case WM_CREATE:
    ...
    hdc = GetDC(hwnd);                          //获取设备环境
    hdcmem = CreateCompatibleDC(hdc);           //获取内存设备环境
    ...                                         //进行一系列操作
    ReleaseDC(hwnd,hdc);                        //释放设备环境
```

3. 选入内存设备环境

获取了内存设备环境句柄后，应用程序需调用 SelectObject 函数将位图选入内存设备环境中，其形式如下：

```
SelectObject(hdcmem,hBm);
```

将位图选入内存设备环境后，即可对其进行编辑。

4. 输出位图

最后，应用程序调用函数 BitBlt 在指定的设备上输出内存中的位图。函数 BitBlt 将位图从内存设备环境拷贝到设备环境中，其原型如下：

```
BOOL BitBlt
    (
    HDC hdcDest,                    //目的设备环境句柄
    int Xdest, int Ydest,           //标识目的设备显示位图的基点(位图左上角坐标)
    int Width, int nHeitght,        //目的设备中用于显示位图的区域的高和宽
    HDC hdcScr,                     //源设备环境句柄
    int  XScr,int Yscr,             //标识源设备中位图的左上角坐标
    DWORD dwRop                     //标识位图显示方式，操作码及其说明，见表 7-5
    )
```

表 7-5 dwRop 操作码及说明

操 作 码	说 明
BLACKNESS	输出全黑色位图
DSTINVERT	目标位图执行“取反”操作
MERGECOPY	将源位图和模板执行“与”操作
MERGEPAINT	将源位图和模板执行“或”操作
NOTSRCCOPY	在拷贝之前将源位图执行“取反”操作
NOTSRCERASE	将源位图和目标位图执行“或”操作，再执行“取反”操作
PATCOPY	将模板拷贝到目标位置上
PAINTVERT	将模板和目标位图执行“异或”操作
SRCCOPY	将源位图拷贝到目标位图(常用)
SRCAND	将源位图和目标位图执行“与”操作
SRCPAINT	将源位图和目标位图执行“或”操作
SRCERASE	将目标位图执行“取反”操作，再与源位图执行“与”操作
SRCINVERT	将源位图和目标位图执行“异或”操作
WHITENESS	输出全白色位图

另外,应用程序在输出位图之前,经常需要调用函数 GetObject 获取位图的尺寸。函数 GetObject 的作用是获取指定对象的信息并将其拷贝到指定的缓冲区内,该函数的原型为:

```
int GetObject
(
HANDLE hObject,          //对象句柄
int nCount,              //拷贝到缓冲区的字节数
LPVOID lpObject          //接收信息的缓冲区地址
);
```

应用程序调用该函数获取位图尺寸的形式为:

```
GetObject
(
hBitmap,                 //位图句柄
sizeof(BITMAP),          //BITMAP 结构的大小
(LPVOID)&bm              //BITMAP 结构的地址
);
```

应用程序调用函数 GetObject 后,将指定位图的信息写入 BITMAP 结构中。数据结构 BITMAP 在位图操作中经常使用,其定义如下:

```
typedef struct tagBITMAP
|
LONG bmType;             //说　明
LONG bmWidth;            //位图类型
LONG bmHeight;           //位图宽度
LONG bmWidthBytes;       //位图高度
WORD bmPlanes;           //每一光栅行的字节数
WORD bmBitsPixel;        //位图中位面的数目
LPVOID bmBits;           //位图中每个像素的位数
|BITAMP;                 //位图位值的地址
```

7.2.3　位图资源应用实例

【例 7-2】

位图资源应用示例。本例调用一幅坦克的图片,并在用户窗口区上显示。

具体的源程序代码如下:

```
//位图示例.c
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

//定义变量
```

```
HDC hdc;                //设备环境句柄
HDC hdcmem;             //内存设备环境句柄
HBITMAP hBm;            //位图句柄
BITMAP bm;              //BITMAP 结构

int WINAPI WinMain
    (
    HINSTANCE hInstance,
    HINSTANCE hPrevInst,
    LPSTR lpszCmdLine,
    int nCmdShow
    )

{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;

    char lpszClassName[] = "位图";
    char lpszTitle[] = "Example For BitMap";

    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;

    if( ! RegisterClass( &wndclass))
    {
        MessageBeep(0) ;
        return FALSE ;
    }

    hwnd = CreateWindow(lpszClassName,
                        lpszTitle,
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
```

```
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL,
                        NULL,
                        hInstance,
                        NULL) ;

hBm = LoadBitmap(hInstance,"pic7_2");              //加载位图
GetObject( hBm, sizeof( BITMAP), (LPVOID)&bm) ; //获取位图尺寸
ShowWindow( hwnd, nCmdShow) ;
UpdateWindow(hwnd);

while( GetMessage(&Msg, NULL, 0, 0))
   {
      TranslateMessage( &Msg) ;
      DispatchMessage( &Msg) ;
   }

return Msg.wParam;
}

LRESULT CALLBACK WndProc(  HWND hwnd,
                           UINT message,
                           WPARAM  wParam,
                           LPARAM  lParam)
{   PAINTSTRUCT ps;
    switch(message){
    case WM_CREATE:
    hdc = GetDC(hwnd);                    //获取设备环境句柄
    hdcmem = CreateCompatibleDC(hdc);     //获取内存设备环境句柄
    ReleaseDC(hwnd,hdc);                  //释放设备环境句柄

    case WM_PAINT:
        hdc = BeginPaint(hwnd,&ps);
        SelectObject(hdcmem,hBm);         //将位图选入内存设备环境

        //调用函数 BitBlt 将内存设备环境中的位图拷贝到设备环境中
        BitBlt(hdc,180,80,bm.bmWidth,bm.bmHeight,hdcmem,0,0,SRCCOPY);

        EndPaint(hwnd,&ps);               //释放设备环境句柄
        break;
```

```
    case WM_DESTROY:
        DeleteObject(hBm);    //释放位图
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

本例的资源文件如下：

```
pic7-2 BITMAP pic7-2.bmp    //定义位图资源
```

程序的运行结果如图 7-3 所示。

图 7-3 [例 7-2]程序的运行结果

7.3 对话框资源及其应用

对话框是一个弹出式窗口，窗口一般用于程序需要用户输入或者需要和用户进行交互活动的场合。一般来说，对话框消息的处理在独立的对话框函数内进行。对话框中包含了众多的控件，如按钮、文本框、滚动条、列表框、编辑框等。对话框的主要形式有“模态对话框”和“非模态对话框”两类。

如果一个应用程序中包含有对话框，则应用程序中必须包含一个对话框函数，这个函数与窗口函数类似，只不过窗口函数用于处理与窗口有关的消息，而对话框函数用于处理

和对话框有关的消息。

对话框资源是一种非常有用的重要资源，Windows 应用程序通常采用对话框资源作为与用户之间直接交互的工具。

对话框资源通常有如下功能：

- 发送消息，如警告消息、提示框消息。
- 接收输入，如用户输入的消息。
- 提供消息，如常见的“关于”对话框。

模态对话框不允许用户在关闭对话框之前切换到应用程序的其他窗口。当一个模态对话框初始化时，对话框的消息循环将处理消息，但并不返回给 WinMain 函数。

非模态对话框允许用户在该对话框与应用程序其他窗口之间的切换，即对话框和其他应用程序的窗口之间进行来回切换。非模态对话框从 WinMain 函数的消息循环中接收输入。使用模式对话框还是使用非模态对话框取决于应用程序及其实现。

7.3.1　模态对话框的编程方法

模态对话框的编程要经历定义对话框资源、显示对话框、构造对话框消息处理函数以及关闭对话框等具体操作。

1.定义对话框资源

创建对话框首先应在应用程序的资源描述文件中定义对话框，一般的形式为：

```
对话框名 DIALOG[载入特性选项]X,Y,Width,Height[设置选项]
{
    对话框的控件定义
}
```

对话框的定义由以下部分组成：

- 对话框名：应用程序通过对话框名标识对话框资源。
- DIALOG 关键字。
- 载入特性选项：对话框资源可选的载入特性选项与菜单资源相同。
- 对话框位置及外型尺寸：其中 X,Y 为对话框在窗口中的左上角坐标；Width,Height 为对话框的宽度与高度。
- 设置选项：设置选项常用的有 CAPTION(标题)和 STYLE(样式)。

对话框的样式选项决定了对话框资源的外型特点。除一般的窗口样式外，Windows 系统还为对话框提供一些特有的样式。常用的对话框样式及其说明见表 7-6。

表 7-6　对话框常用样式及其说明

样　式	说　明	样　式	说　明
DS_3DLOOK	使用三维边框	DS_SHADOW	带阴影的对话框
DS_FIXEDSYS	使用 SYSTEM_FIXED 字体	DS_CENTER	对话框居中
DS_MODALFRAME	使用细实线边框	DS_SETFOREGROUND	置对话框于前台
DS_SYSMODAL	系统模式对话框		

对话框样式常用组合样式形式表示，下列语句定义了一个含标题栏的弹出式对话框：

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
```

• 控件定义：对话框中常用的控件及其说明见表 7-7(有关控件的内容，详见第 8 章)。

表 7-7 对话框中常用控件及其说明

控 件	说 明	控 件	说 明
CHECKBOX	复选框	LISTBOX	列表框
COMBOBOX	组合框	LTEXT	文本左对齐的静态控件
CTEXT	文本居中的静态控件	PUSHBUTTON	按钮
DEFPUSHBUTTON	默认按钮	RADIOBUTTON	单选按钮
EDITTEXT	编辑框	RTEXT	文本右对齐的静态控件
GROUPBOX	组框	SCROLLBAR	滚动条
ICON	图标		

2. 调用函数 DialogBox 显示对话框

在资源描述文件中定义对话框资源后，应用程序可通过调用 DialogBox 函数在窗口中显示对话框，该函数的原型为：

```
int DialogBox
(
HINSTANCE hInstance,        //当前应用程序的实例句柄
LPCTSTR lpszDialogName,     //对话框资源的名称
HWND hwndParent,            //拥有该对话框的窗口句柄
WNDPROC lpDialogFunc        //对话框处理函数的地址
)
```

3. 构造对话框消息处理函数

对话框接收的消息都在相应的对话框消息处理函数中处理，对话框消息处理函数的一般形式为：

```
BOOL CALLBACK DlgProc
    (
    HWND hDlg,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
    )
{
    switch(message)
    {
        case  WM_INITDIALOG:
            return 1;
        case  WM_COMMAND
            switch (LOWORD (wParam ))
```

```
                {
                    case …
                      …
                      break;
                    case …
                      …
                      break;
                      …
                }
            break;
    }
    return 0;
}
```

对话框消息处理函数具有与主窗口函数相似的参数，但两者存在以下三点不同：

- 函数的返回值不同。对话框消息处理函数返回 BOOL 值，而主窗口返回 LRESULT 值。
- 对话框消息处理函数不处理某些消息。对话框消息处理函数不需处理 WM_PAINT，WM_DESTROY 及 WM_CREATE 消息。
- 对未定义处理过程消息的处理不同。主窗口函数通过调用函数 DefWindowProc 完成对未定义处理过程消息的处理，而对话框消息处理函数如果接收到未定义处理过程的消息，则返回 FALSE(return 0)。

在对话框消息处理函数中经常处理以下两类消息：

(1) WM_INITDIALOG 消息

对话框在响应消息 WM_ INITDIALOG 时完成其初始化操作，在功能上与主窗口函数的 WM_COMMAND 消息相似。

(2) WM_COMMAND 消息

对话框在响应消息 WM_COMMAND 时，通过查看消息字参数(wParam)中的低位字节与控件标识(ID)相比较的结果，以确定产生交互请求的控件并据此转入相应的处理过程处理。

4. 关闭对话框

在对话框窗口函数中，调用函数 EndDialog 可以关闭对话框，关闭对话框函数的一般形式为：

```
EndDialog (hdlg,nResult) ;
```

上式中，hdlg 为对话框句柄；nResult 为从对话框返回到 DialogBox 函数的值。

Windows 消息框是模态对话框的一种特殊形式，应用程序可通过调用函数 MessageBox 快捷地生成一些简单但又常用的 Windows 消息框。MessageBox 函数的原型为：

```
int MessageBox
(
```

```
HWND hwnd,              //拥有消息框的窗口
LPCTSTR lpszText,       //消息框中显示的字符串
LPCTSTR lpszCaption,    //作为标题的字符串
DWORD dwType            //指定消息框的内容,其常用的标识及说明见表 7-8
);
```

表 7-8　dwType 常用标识及其说明

标　识	说　明
MB_ABORTRETRYIGNORE	含有 Abort、Retry 和 Ignore 按钮的消息框
MB_ICONEXCLAMATION	含有惊叹号图标的消息框
MB_ICONQUESTION	含有问号图标的消息框
MB_ICONSTOP	含有停止图标的消息框
MB_OK	含有一个 OK 按钮的消息框
MB_OKCANCEL	含有 OK 和 CANCEL 按钮的消息框
MB_RETRYCANCEL	含有 RETRY 和 CANCEL 按钮的消息框
MB_YESNO	含有 YES 和 NO 按钮的消息框
MB_YESNOCANCEL	含有 YES、NO 和 CANCEL 按钮的消息框

应用程序常使用上述标识的组合，如下面的标识组合表示含有 YES，NO，CANCLE 按钮及惊叹号图标的消息框：

```
MB_ICONEXCLAMATION | MB_YESNOCANCEL
```

该函数返回值中包含用户的交互信息，例如用户按下 OK 键，则函数返回标识值 IDOK。表 7-9 为消息框中用户操作与 MessageBox 返回值之间的对应关系。

表 7-9　用户操作与 MessageBox 返回值之间的对应关系

返回值	用户操作
IDABORT	按下 Abort 按钮
IDCANCEL	按下 Cancel 按钮
IDIGNORE	按下 Ignore 按钮
IDNO	按下 No 按钮
IDOK	按下 Ok 按钮
IDRETRY	按下 Retry 按钮
IDYES	按下 Yes 按钮

7.3.2　模态对话框应用实例

【例 7-3】

模态对话框例子。该对话框含有标题栏、系统菜单、简要信息及一个按钮控件。

应用程序在资源文件中定义了该对话框，并在对话框处理函数 DlgProc 中处理按钮消息 IDOK 及窗口关闭消息等。该对话框是典型的模态对话框，用户必须单击“确定”按钮后或关闭该对话框之后，方可切换到应用程序的其他窗口。此外，该程序在“保存”菜单项处理过程中使用了消息框，在保存操作结束时通知用户。

本例子的实现代码如下：

```
# include < windows.h >
# include "7-3.h"
HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM); //定义对话框窗口处理函数
int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInst,
                    LPSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    HACCEL hAccel;
    WNDCLASS wndclass ;
    char lpszMenuName[] = "Menu";
    char lpszClassName[] = "模态对话框";
    char lpszTitle[] = "模态对话框样例";

    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI-APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC-ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE-BRUSH) ;
    wndclass.lpszMenuName = lpszMenuName ;
    wndclass.lpszClassName = lpszClassName ;

    if( ! RegisterClass( &wndclass))
        {
          MessageBeep(0) ;
          return FALSE ;
        }

    hwnd = CreateWindow(lpszClassName,
                        lpszTitle,
                        WS-OVERLAPPEDWINDOW,
                        CW-USEDEFAULT,
```

```
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL,
                        NULL,
                        hInstance,
                        NULL) ;

    ShowWindow( hwnd, nCmdShow) ;
    UpdateWindow(hwnd);

    hInst = hInstance;
    hAccel = LoadAccelerators(hInst,lpszMenuName);

    while( GetMessage(&Msg, NULL, 0, 0))
    {
        if (! TranslateAccelerator(hwnd,hAccel,&Msg))
        {
            TranslateMessage( &Msg) ;
            DispatchMessage( &Msg) ;
        }
    }
    return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch(message)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
          case IDM_NEW:
              break;

          case IDM_OPEN:
              //打开文件操作
```

```
            //文件打开成功则显示消息框
            MessageBox(hwnd,"文件已经打开!","文件打开",MB_OK);
            break;

        case IDM_SAVE:
            //存储操作
            //文件保存成功则显示消息框
            MessageBox(hwnd,"文件保存成功!","文件保存",MB_OK);
            break;
        case IDM_SAVEAS:
            break;
        case IDM_CLOSE:
            break;
        case IDM_EXIT:
            SendMessage(hwnd,WM_DESTROY,0,0);
            break;
        case IDM_CUT:
            break;
        case IDM_COPY:
            break;
        case IDM_PASTE:
            break;
        case IDM_DELETE:
            break;
        case IDM_ABOUT:
            DialogBox(hInst,"About",hwnd,(DLGPROC)DlgProc); //创建对话框
            break;
          }
      break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

```
//对话框窗口处理函数
BOOL CALLBACK DlgProc(HWND hdlg,UINT message,WPARAM wParam,
                      LPARAM lParam)
{
    switch(message)
    {
        //初始化对话框
        case WM_INITDIALOG:
            return 1;

        //处理对话框消息
        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                  case IDOK:
                      //关闭对话框
                      EndDialog( hdlg, 0) ;
                      return 1;
            }
            break ;
        case WM_CLOSE:
            EndDialog( hdlg, 0) ;
        return 1;
    }
    return 0;
}
```

本例的资源程序代码如下：

```
#include <windows.h>
#include "7-3.h"
Menu MENU DISCARDABLE
  {
POPUP "文件(&F)"
  {
  MENUITEM "新建(&N)\tCtrl+N",      IDM_NEW
  MENUITEM "打开(&O)\tCtrl+O",      IDM_OPEN
  MENUITEM "关闭(&C)",              IDM_CLOSE
  MENUITEM SEPARATOR
  MENUITEM "保存(&S)\tCtrl+S",      IDM_SAVE
  MENUITEM "另存为(&A)",            IDM_SAVEAS
```

```
    MENUITEM SEPARATOR
    MENUITEM "退出(&X)",                    IDM_EXIT
    }

  POPUP "编辑(&E)"
    {
    MENUITEM "剪切(&T)\tCtrl+X",     IDM_CUT
    MENUITEM "复制(&C)\tCtrl+C",     IDM_COPY
    MENUITEM "粘贴(&P)\tCtrl+V",     IDM_PASTE
    MENUITEM "删除(&D)\tDel",        IDM_DELETE
    }

  POPUP "帮助(&H)"
    {
    MENUITEM "关于(&A)...",          IDM_ABOUT
    }
  }

  Menu ACCELERATORS
    {
    "^N",          IDM_NEW
    "^O",          IDM_OPEN
    "^S",          IDM_SAVE
    "^X",          IDM_CUT
    "^C",          IDM_COPY
    "^V",          IDM_PASTE
    VK_DELETE,     IDM_DELETE,     VIRTKEY
    }

  About DIALOG  10, 10, 100, 50
  STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
  CAPTION "关于"
    FONT 18, "楷体"  //定义对话框中的文本字体类型及大小
      {
      CTEXT          "Windows 模态对话框", -1,13,10,80,10
      DEFPUSHBUTTON "确定",IDOK,35,30,30,12
      }
```

本例的头文件代码如下：

```
  #define IDM_NEW          10
```

```
#define IDM_OPEN        11
#define IDM_CLOSE       12
#define IDM_SAVE        13
#define IDM_SAVEAS      14
#define IDM_EXIT        15
#define IDM_CUT         16
#define IDM_COPY        17
#define IDM_PASTE       18
#define IDM_DELETE      19
#define IDM_ABOUT       20
```

本例的运行结果如图 7-4 所示。

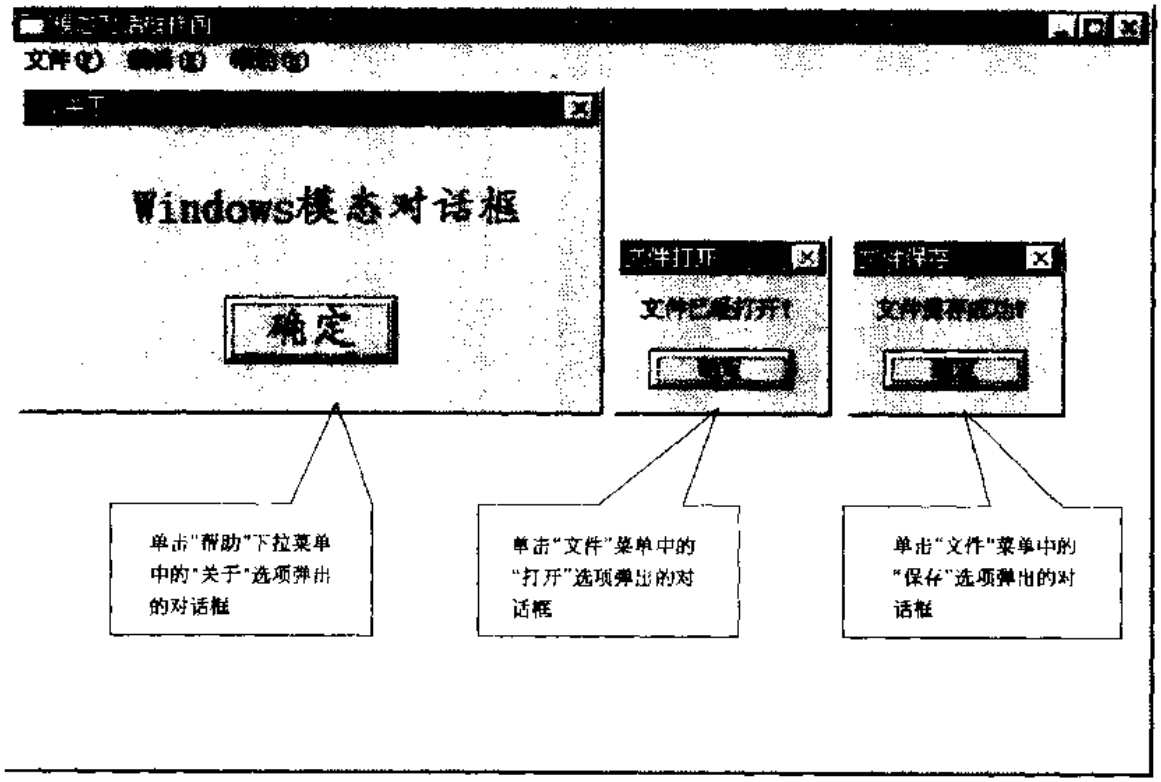


图 7-4　【例 7-3】程序的运行结果

7.3.3　非模态对话框的编程方法

非模态对话框与模态对话框的编程比较类似，但在对话框的定义、对话框的创建及消息处理上略有差别。

非模态对话框编程方法如下：

1. 定义对话框样式

非模态对话框的定义一般形式如下：

```
STYLE  WS_POPUP | WS_CAPTION | WS_VISIBLE
```

非模态对话框允许用户在该对话框与应用程序其他窗口之间切换，因此标识该对话框内容的标题一般不可省略。尤其值得注意的是非模态对话框样式中应包含 WS_VISIBLE，否则非模态对话框将无法在屏幕上显示。

2. 创建对话框

非模态对话框的创建由函数 CreateDialog 完成。该函数的原型如下：

```
HWND CreateDialog
    (
    HINSTANCE hInstance,          //当前应用程序实例句柄
    LPCTSTR lpszDialogName,       //对话框名
    HWND hwndParent,              //拥有该对话框的窗口句柄
    WNDPROC lpDialogFunc          //对话框处理函数地址
    );
```

3. 消息循环部分的处理

由于非模态对话框并不禁止应用程序向其他窗口发送消息，因此，在 WinMain 函数的消息循环中必须包含截获发往非模态对话框的消息，并将其发往相应的对话框处理函数进行处理的语句。其消息循环过程的一般形式为：

```
while (GetMessage (&Msg,NULL,0,0))
|
if (! IsDialogMessage (hdlg,&Msg))
    {
        TranslateMessage (&Msg);
        DispatchMessage (&Msg);
    }
}
```

应用程序调用函数 IsDialogMessage 判断消息是否发往非模态对话框，如果是，则将消息发送到对话框处理函数进行处理。其中，hdlg 为应用程序通过调用函数 CreateDialog 获取的非模态对话框句柄，该函数的原型为：

```
BOOL IsDialogMessage (HWND hdlg,LPMSG lpMsg)     // lpMsg 为指向 MSG 结构的指针
```

4. 关闭对话框

非模态对话框调用函数 DestroyWindow 关闭对话框，该函数的原型为：

```
BOOL DestroyWindow (HWND hdlg)
```

7.3.4　非模态对话框应用实例

【例 7-4】

本例中，在“关于”窗口不关闭的前提下，仍可以对“文件”下拉菜单进行操作，如选“文件”下拉菜单中的“创建”选项，即可弹出“创建”窗口，如图 7-5 所示。

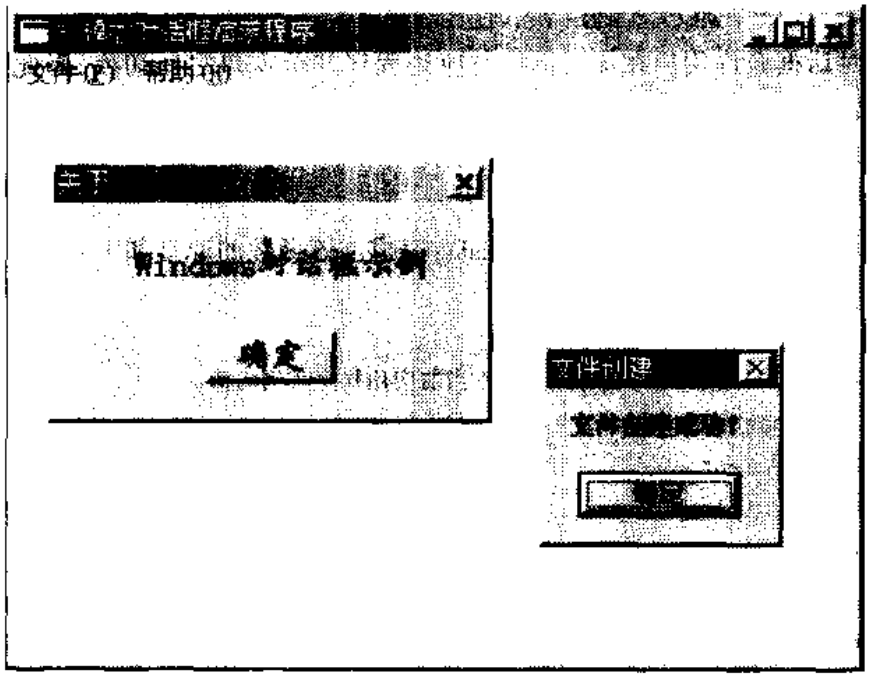


图 7-5 非模态对话框样例运行结果

本例的源程序代码如下：

```
#include <windows.h>
#include "7-4.h"
HWND hdlg; //对话框句柄
HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM); //定义消息框窗口处理函数

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInst,
                    LPSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    HACCEL hAccel;
    WNDCLASS wndclass ;
    char lpszMenuName[] = "Menu";
    char lpszClassName[] = "非模态对话框";
    char lpszTitle[] = "非模态对话框演示程序";

    wndclass.style = 0;
```

```
wndclass.lpfnWndProc = WndProc ;
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
wndclass.lpszMenuName = lpszMenuName ;
wndclass.lpszClassName = lpszClassName ;

if( ! RegisterClass( &Wndclass))
{
  MessageBeep(0) ;
  return FALSE ;
}

hwnd = CreateWindow( lpszClassName,
                     lpszTitle,
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     NULL,
                     NULL,
                     hInstance,
                     NULL) ;

ShowWindow( hwnd, nCmdShow) ;
UpdateWindow(hwnd);
hInst = hInstance;
hAccel = LoadAccelerators(hInst,lpszMenuName);

while( GetMessage(&Msg, NULL, 0, 0))
{
//截获非模态对话框消息并发往处理函数
    if(! IsDialogMessage(hdlg,&Msg))
    {
        if (! TranslateAccelerator(hwnd,hAccel,&Msg))
        {
            TranslateMessage( &Msg) ;
```

```
            DispatchMessage( &Msg) ;
        }
    }
    return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM  wParam,
                         LPARAM  lParam)
{
    switch(message)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case IDM_NEW:
              MessageBox(hwnd,"文件创建成功!","文件创建",MB_OK);
              break;
            case IDM_OPEN:
              MessageBox(hwnd,"文件打开成功!","文件打开",MB_OK);
              break;
            case IDM_SAVE:
              MessageBox(hwnd,"文件保存成功!","文件保存",MB_OK);
              break;
            case IDM_SAVEAS:
              break;
            case IDM_CLOSE:
              break;
            case IDM_EXIT:
              SendMessage(hwnd,WM_DESTROY,0,0);
              break;
            case IDM_ABOUT:
              //调用函数 CreateDialog 创建非模态对话框
              hdlg = CreateDialog(hInst,"About",hwnd,(DLGPROC)DlgProc);
              break;
            }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
```

```
        default:
            return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}

//对话框窗口处理函数
BOOL CALLBACK DlgProc(HWND hdlg,UINT message,WPARAM wParam,
                                    LPARAM lParam)
{
    switch(message)
    {
        //初始化消息框
        case WM_INITDIALOG:
            return 1;

            //处理消息框消息
        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                case IDOK:
                    //关闭消息框
                    DestroyWindow( hdlg) ;
                    return 1   ;
            }
                break;

        case WM_CLOSE:
            DestroyWindow( hdlg) ;
            return 1;
    }
    return 0;
    }
```

本例的头文件代码如下：

```
# define IDM_NEW          10
# define IDM_OPEN         11
# define IDM_CLOSE        12
# define IDM_SAVE         13
# define IDM_SAVEAS       14
```

```
#define IDM_EXIT        15
#define IDM_CUT         16
#define IDM_COPY        17
#define IDM_PASTE       18
#define IDM_DELETE      19
#define IDM_ABOUT       20
```

本例的资源文件代码如下：

```
#include <windows.h>
#include "7-4.h"

Menu MENU DISCARDABLE
    {
        POPUP "文件(&F)"
            {
                MENUITEM "新建(&N)\t Ctrl+N",        IDM_NEW
                MENUITEM "打开(&O)\t Ctrl+O",        IDM_OPEN
                MENUITEM "关闭(&C)",                 IDM_CLOSE
                MENUITEM SEPARATOR
                MENUITEM "保存(&S)\t Ctrl+S",        IDM_SAVE
                MENUITEM "另存为(&A)",               IDM_SAVEAS
                MENUITEM SEPARATOR
                MENUITEM "退出(&X)",                 IDM_EXIT
            }

        POPUP "帮助(&H)"
            {
                MENUITEM "关于(&A)...",              IDM_ABOUT
            }

        Menu ACCELERATORS
        {
            "^N",           IDM_NEW
            "^O",           IDM_OPEN
            "^S",           IDM_SAVE
            VK_DELETE,      IDM_DELETE,     VIRTKEY
        }

About DIALOG  50, 50, 100, 50
STYLE DS_MODALFRAME|WS_POPUP | WS_CAPTION , WS_SYSMENU|WS_VISIBLE
                        //非模态对话框的风格中应包含 WS_VISIBLE
        CAPTION "关于"
```

```
FONT 12,"楷体"
  {
    CTEXT          "Windows 对话框示例",-1,13,10,80,10
    DEFPUSHBUTTON "确定",IDOK,35,30,30,12
  }
```

7.4　通用对话框资源及其应用

程序员除使用自定义的对话框外，还可以直接调用 Windows 系统中提供的通用对话框模板生成对话框资源。Windows 系统提供了一系列常用的通用对话框，如“颜色”对话框、“字体”对话框、“打开”对话框、“保存”对话框、“打印机设置”对话框等，这些对话框模板在系统提供的 COMMDLG.h 头文件中定义。

7.4.1　通用对话框的创建过程

通用对话框的创建过程，需要经历填充对话框模板结构和调用函数显示对话框两个步骤。

1. 填充对话框模板数据结构

应用程序在使用通用对话框前，应先填充要使用的对话框模板的数据结构。系统各通用对话框模板对应的数据结构的名称分别是：

- OPENFILENAME：“打开”通用对话框模板数据结构。
- CHOOSECOLOR：“颜色”对话框模板数据结构。
- CHOOSEFONT：“字体”对话框模板数据结构。
- PRINTDLG：“打印”对话框模板数据结构。
- PAGESETUPDLG：“页面设置”对话框模板数据结构。
- FINDREPLACE：“查找”对话框模板数据结构。

其中最常用的数据结构是 OPENFILENAME 结构，“打开”和“另存为”对话框都使用该结构完成定义。该数据结构的定义如下：

```
typedef struct tagOPENFILENAME
{
  DWORD  lStructSize;        //该数据结构的长度
  HWND  hwndOwner;           //拥有该数据结构的窗口句柄
  HANDLE    hInstance;       //对话框模板句柄
  LPCSTR    lpstrFilter;     //标识对话框初始文件过滤器的字符串
  LPSTR  lpstrCustomFilter;  //自定义的标识对话框初始文件过滤器的字符串
  DWORD  nMaxCustFilter;     //lpstrCustomFilter 缓冲区的大小，一般要求大于 40 字节
  DWORD  nFilterIndex;       //文件过滤器正文串的索引值
  LPSTR  lpstrFile;          //指向文件名列表框中文件的指针，对话框退出时，该指针指
                               向打开的文件名
  DWORD  nMaxFile;           //lpstrFile 指定的缓冲区的大小
  LPSTR  lpstrFileTitle;     //指向选定文件名的指针
```

```
    DWORD   nMaxFileTitle;      //lpstrFileTitle 指定的缓冲区的大小
    LPCSTR  lpstrInitialDir;    //指向对话框初始文件目录文件的指针
    LPCSTR  lpstrTitle;         //对话框的标题
    DWORD   Flags;              //对话框的标识,详见表 7-10
    UINT    nFileOffset;        //lpstrFile 中从路径名到文件名的偏移
    UINT    nFileExtension;     //lpstrFile 中从路径名到文件扩展名的偏移
    LPCSTR  lpstrDefExt;        //指向默认扩展名的指针
    LPARAM  lCustData;          //应用程序自定义的数据,将其传给相应 lpfnHook 确定的
                                  钩子函数
    UINT  (CALLBACK * lpfnHook)(HWND,UINT,WPARAM,LPARAM)  // lpfnHook 钩子
                                //函数指针,该函数的功能与对话框窗口处理函数相同
    LPCSTR  lpTemplateName;     //对话框模板的标签
} OPENFILENAME;
```

其中 Flags 为一组“OFN_”形式的标识,程序员可通过这些标志设置对话框的特征。表 7-10 列举了 Flag 标志的常用标识及其说明。

表 7-10 Flag 常用标识及其说明

标 识	说 明
OFN_ALLOWMULTISELECT	允许文件名列表框中多项选择
OFN_CREATEPROMPT	在当前文件不存在的情况下询问是否建立新文件
OFN_ENABLEHOOK	表明钩子函数可激活
OFN_ENABLETEMPALATE	使用对话框模板建立对话框
OFN_ENABLETEMPALATEHANDLE	使用 hInstance 参数定义的对话框模板建立对话框
OFN_EXTENSIONDIFFERENT	表明用户输入的文件扩展名不同于 lpstrDefExt 中确定的扩展名
OFN_FILEMUSTEXIST	必须输入现有文件的文件名
OFN_HIDEREADONLY	隐藏只读框
OFN_OVERWRITEPROMPT	使用“另存为”对话框时,当用户保存一现有文件时弹出确认消息框
OFN_PATHMUSTEXIST	必须输入现有路径名
OFN_READONLY	含有只读框
OFN_SHOWHELP	含有帮助按钮

2. 调用函数显示对话框

Windows 系统提供一系列 API 函数用以显示通用对话框,表 7-11 列出了该类函数及其功能。

表 7-11 通用对话框函数及其功能

函 数 名	说 明	函 数 名	说 明
ChooseFont	显示“字体”对话框	GetSaveFileName	显示“保存”对话框
ChooseColor	显示“颜色”对话框	PageSetupDlg	显示“页面设置”对话框
FindText	显示“查找”对话框	PrintDlg	显示“打印”对话框
GetOpenFileName	显示“打开”对话框	ReplaceText	显示“替换”对话框

例如，调用函数 GetOpenFileName 显示“打开”对话框的形式如下：

```
GetOpenFileName(&ofn);     //ofn 为 OPENFILENAME 结构
```

7.4.2　通用对话框应用实例

【例 7-5】

通用对话框的应用。本程序将调用系统提供的“打开”对话框。在响应消息 WM_CREATE 过程中完成 OPENFILENAME 结构的初始化。当用户选择“打开”菜单项时，应用程序设置相应的 Flags 为：

```
OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST|OFN_HIDEREADONLY
```

并且调用函数 GetOpenFileName 显示对话框，当用户选择“另存为”菜单项时，应用程序设置相应的 Flag 为：

```
OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST|OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT
```

并且调用函数 GetSaveFileName 显示对话框。

本例题的源代码如下：

```
# include <windows.h>
# include <commdlg.h>
# include <stdio.h>
# include <stdlib.h>
# include "7-5.h"

OPENFILENAME ofn;//定义一个 OPENFILENAME 结构
HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine,
                   int nCmdShow)
{
        HWND hwnd ;
        MSG Msg ;
        HACCEL hAccel;
        WNDCLASS wndclass ;
        char lpszMenuName[ ] = "Menu";
        char lpszClassName[ ] = "通用对话框";
        char lpszTitle[ ] = "通用对话框样例";

        wndclass.style = 0;
        wndclass.lpfnWndProc = WndProc ;
```

```
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
wndclass.lpszMenuName =  lpszMenuName ;
wndclass.lpszClassName = lpszClassName ;

if( ! RegisterClass( &wndclass))
    {
      MessageBeep(0) ;
      return FALSE ;
    }

hwnd = CreateWindow(lpszClassName,
                    lpszTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    NULL,
                    NULL,
                    hInstance,
                    NULL) ;

ShowWindow( hwnd, nCmdShow) ;
UpdateWindow(hwnd);
hInst = hInstance;
hAccel = LoadAccelerators(hInst,lpszMenuName);

while( GetMessage(&Msg, NULL, 0, 0))
{

    if (! TranslateAccelerator(hwnd,hAccel,&Msg))
      {
      TranslateMessage( &Msg) ;
      DispatchMessage( &Msg) ;
      }
```

```
    }

    return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)

{
    static char szFilter[] = "Text Files (*.TXT)\0*.txt\0"\
                             "All Files (*.*)\0*.*\0\0";
    static char lpstrFileName[]="";

    switch(message)
    {
    case WM_CREATE:

        //初始化对话框结构
        ofn.lStructSize       = sizeof(OPENFILENAME);
        ofn.hwndOwner         = hwnd;
        ofn.hInstance         = NULL;
        ofn.lpstrFilter       = szFilter;
        ofn.lpstrCustomFilter= NULL;
        ofn.nMaxCustFilter    = 0;
        ofn.nFilterIndex      = 1;
        ofn.lpstrFile         = NULL;
        ofn.nMaxFile          = 0;
        ofn.lpstrFileTitle    = NULL;
        ofn.nMaxFileTitle     = 0;
        ofn.lpstrInitialDir   = NULL;
        ofn.lpstrTitle        = NULL;
        ofn.Flags             = 0;
        ofn.nFileOffset       = 0;
        ofn.nFileExtension    = 0;
        ofn.lpstrDefExt       = NULL;
        ofn.lCustData         = 0;
        ofn.lpfnHook          = NULL;
        ofn.lpTemplateName    = NULL;

        break;
```

```
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case IDM_NEW:
              break;
            case IDM_OPEN:
                //设置"打开"对话框
              ofn.lpstrFile = lpstrFileName;
              ofn.Flags = OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST|OFN_HIDEREADONLY;
              ofn.lpstrTitle="打开";
                //调用函数 GetOpenFileName 显示对话框
              GetOpenFileName(&ofn);
                //文件读写操作
            case IDM_SAVEAS:
                //设置"另存为"对话框
              ofn.lpstrFile = lpstrFileName;
              ofn.Flags = OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST
                          |OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT;
              ofn.lpstrTitle="另存为";
                //调用函数 GetSaveFileName 显示对话框
              GetSaveFileName(&ofn);
            case IDM_CLOSE:
              break;
            case IDM_EXIT:
              SendMessage(hwnd,WM_DESTROY,0,0);
              break;
            case IDM_ABOUT:
              MessageBox(hwnd,"通用对话框示例","关于",MB_OK);
              break;
            }
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return  DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

本例题的资源程序代码如下：

```
# include <windows.h>
# include "7-5.h"

Menu MENU DISCARDABLE
{
    POPUP "文件(&F)"
    {
        MENUITEM "新建(&N)\tCtrl+N",          IDM_NEW
        MENUITEM "打开(&O)\tCtrl+O",          IDM_OPEN
        MENUITEM "关闭(&C)",                  IDM_CLOSE
        MENUITEM SEPARATOR
        MENUITEM "另存为(&A)",                IDM_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "退出(&X)",                  IDM_EXIT
    }

    POPUP "帮助(&H)"
    {
        MENUITEM "关于(&A)...",               IDM_ABOUT
    }
}

Menu ACCELERATORS
{
    "^N",       IDM_NEW
    "^O",       IDM_OPEN
    "^S",       IDM_SAVE
}
```

本例题的头文件代码如下：

```
# define IDM_NEW        10
# define IDM_OPEN       11
# define IDM_CLOSE      12
# define IDM_SAVEAS     14
# define IDM_SAVE       13
# define IDM_EXIT       15
# define IDM_ABOUT      20
```

本例题的运行结果如图 7-6 所示。

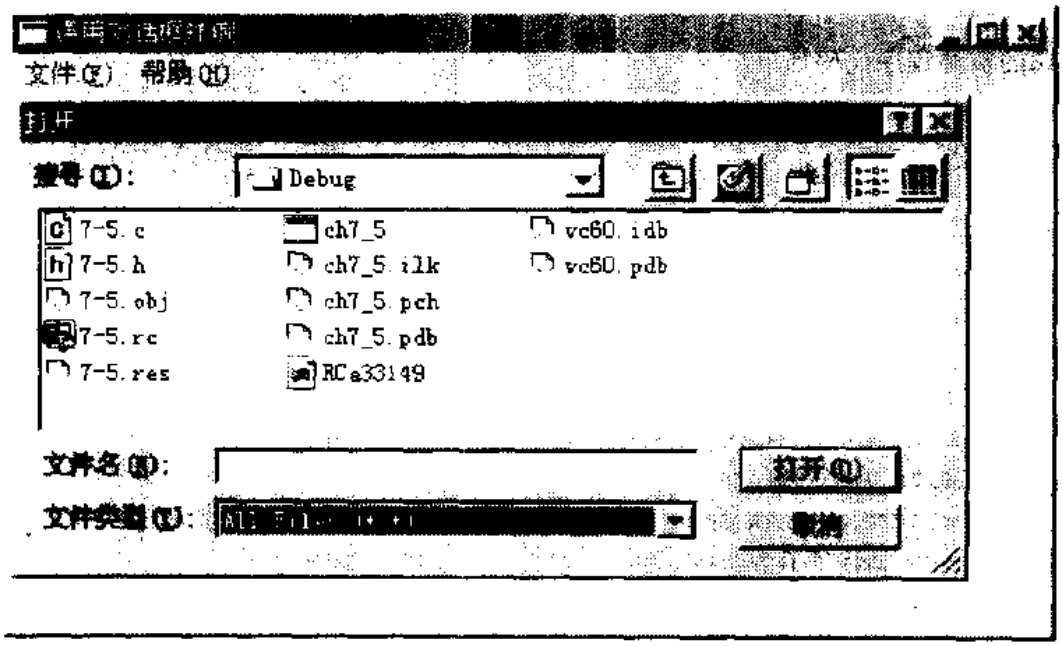


图 7-6　通用对话框样例运行结果

7.5　图标资源及其应用

一个图标就是代表一个应用程序的特殊的最小位图。在图标上双击鼠标就可以执行该应用程序，图标资源可以由 VC 自带的图标资源编辑器来创建。

7.5.1　图标资源的操作

图标资源的操作类似于前面谈到的位图操作，也要经历图标的创建、在资源文件中的定义、图标的加载等过程。

1. 图标资源的创建

用户可以利用 Windows 系统提供的图标，也可以通过图形编辑器自定义图标形式，并保存在扩展名为 .ico 的文件中。Windows 系统提供的图标标识及其形状如表 7-12 所示。

表 7-12　Windows 系统提供的图标标识及其形状

标　识	形　状
IDI_APPLICATION	默认图标
IDI_ASTERISK	信息图标
IDI_EXCLAMATION	惊叹号图标
IDI_HAND	停止图标
IDI_QUESTION	问号图标

2. 在资源文件中定义图标资源

当采用自定义图标时，必须在资源文件中定义该图标，其形式如下：

图标名　ICON　图标文件名(.ico)

3. 在应用程序中加载图标

应用程序是通过调用函数 LoadIcon 进行图标资源的加载的，此过程经常是在定义窗口类时进行，其形式为：

```
WNDCLASS wndclass;
...
wndclass.hIcon = LoadIcon(hThisInst, lpszIconName);
...
```

其中，hThisInst 为应用程序的当前实例句柄，lpszIconName 是图标名。

7.5.2　图标资源应用实例

【例 7-6】

下面的程序是在应用程序中使用图标资源的一个例子。程序所使用的图标文件名为 Winupd.ico，在为本例程序指定了这个图标后，在资源管理器中就可以看到在可执行文件的文件名(本例的可执行文件的文件名为 7-6.exe)的上面有一个相应的图标，如图 7-7 所示。

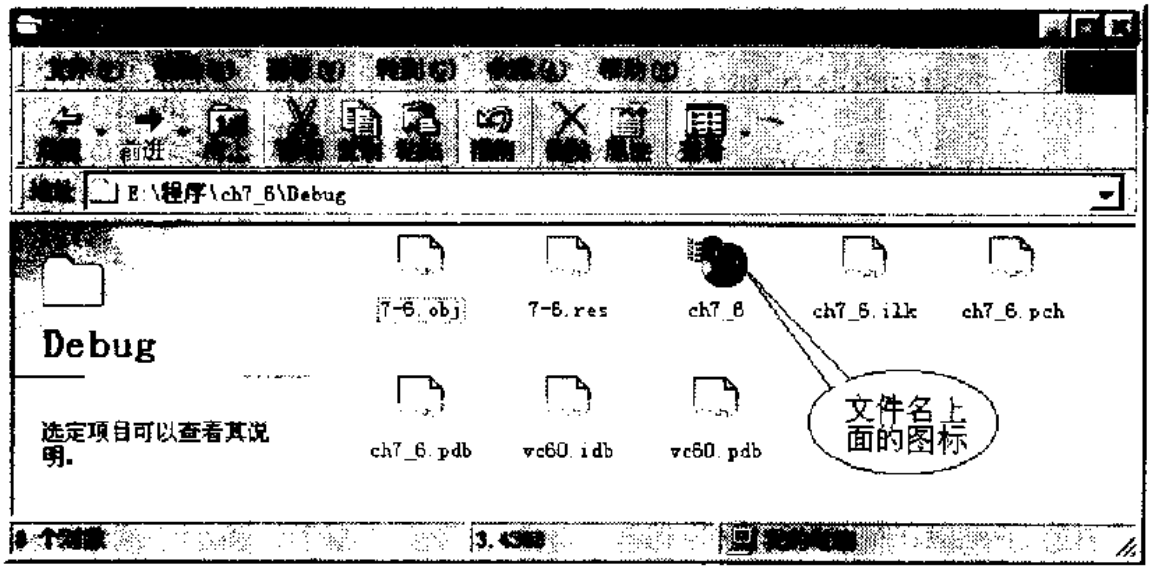


图 7-7　图标资源应用实例运行结果

本例程序中使用的图标资源文件是 WINUPD ICON c:\windows\WINUPD.ICO

本例源程序代码如下：

```
#include<windows.h>
#include<stdlib.h>
#include<string.h>
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam);
BOOL InitWindowsClass(HINSTANCE hInstance);
BOOL InitWindows(HINSTANCE hInstance,int nCmdShow);
```

```
HWND hWndMain;

//主函数
int WINAPI WinMain
    (
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
    )
{
    MSG Message;
    if(! InitWindowsClass(hInstance))
        return FALSE;
    if(! InitWindows(hInstance,nCmdShow))
        return FALSE;
    while(GetMessage(&Message,0,0,0))//消息循环
    {
        TranslateMessage(&Message);
        DispatchMessage(&Message);
    }
    return Message.wParam;
}

//消息处理函数
long WINAPI WndProc(HWND hWnd,UINT iMessage,UINT wParam,LONG lParam)
{
    switch(iMessage)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default:
            return(DefWindowProc(hWnd,iMessage,wParam,lParam));
    }
}

BOOL InitWindowsClass(HINSTANCE hInstance)//初始化窗口类
{
    WNDCLASS WndClass;
    WndClass.cbClsExtra = 0;
    WndClass.cbWndExtra = 0;
```

```
    WndClass.hbrBackground = (HBRUSH)(GetStockObject(WHITE_BRUSH));
    WndClass.hCursor = LoadCursor(hInstance,IDC_ARROW);
    WndClass.hIcon = LoadIcon(NULL,"WINUPD"); //加载图标资源
    WndClass.hInstance = hInstance;
    WndClass.lpfnWndProc = WndProc;
    WndClass.lpszClassName = "WinIcon";
    WndClass.lpszMenuName = NULL;
    WndClass.style = CS_HREDRAW|CS_VREDRAW;
    return RegisterClass(&WndClass);
}

BOOL InitWindows(HINSTANCE hInstance,int nCmdShow)
{
    HWND hWnd;
    hWnd = CreateWindow("WinIcon",  //生成窗口
                        "图标显示示例程序",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        0,
                        CW_USEDEFAULT,
                        0,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
    if(! hWnd)
        return FALSE;
    hWndMain = hWnd;
    ShowWindow(hWnd,nCmdShow);//显示窗口
    UpdateWindow(hWnd);
    return TRUE;
}
```

习　　题

7-1　简述菜单资源的创建过程。

7-2　在程序中如何操作菜单项？

7-3　如何利用位图资源？

7-4　如何应用对话框资源？

7-5　模态对话框与非模态对话框有何区别？在编程上有何不同？

7-6　如何利用图标资源？

7-7 创建一个菜单，其中有三个菜单项，分别为“文件”、“计算”和“帮助”。其中，“文件”菜单项包含“打开”、“保存”、“保存为”、“退出”等项；“计算”菜单包含“总和”、“方差”、“均方根”等项；“帮助”菜单包含“计算方差帮助”、“计算均方根帮助”和“关于”等项。

7-8 在一个窗口中央加载一个位图，位图尺寸为窗口面积的四分之一。当单击向上箭头时，位图向上不断移动；当移动到窗口的上边界时，窗口显示“不能再向上移动了”字样；然后单击向下箭头，让位图不断下移，当到达窗口的下边界时，屏幕显示“不能再向下移动了”字样。

7-9 编写一个窗口应用程序，其中有一个 VC 菜单，该菜单下面有“显示”、“隐藏”、“退出”选项，当选择“显示”选项时，窗口中显示一个对话框，在对话框中显示“我们一起来学习 VC++”；当选择“隐藏”选项时，对话框消失；选择“退出”选项时，退出应用程序的运行。

第 8 章

Windows 标准控件在可视化编程中的应用

控件是 Windows 图形用户界面的主要组成部分之一，用户通过操作控件对象完成与应用程序之间的交互。控件的使用集中体现了 Windows 系统面向对象的特点。

8.1 控件概述

作为一般窗口，控件具有通用的窗口属性。可以使用 ShowWindow( )函数和 MoveWindow( )函数等窗口管理函数来显示或隐藏控件、改变控件的位置、尺寸以及进行其他操作。控件在应用程序中以独立的子窗口和对话框控件两种形式存在。因此，控件的创建形式也有两种，一种是在对话框中创建控件，这样，当应用程序启动该对话框时，Windows 系统就会为对话框创建控件，如按钮控件等都是通过这种方式建立的；另一种是通过调用 CreateWindow( )或 CreateWindowEx( )函数来创建控件，CreateWindowEx( )函数除了多一个 dwExStyle 参数以外，其他参数和 CreateWindow 函数的参数一样。这两个函数在窗口定义上提供了许多灵活性。不同的窗口和控制风格与不同的预定义窗口类相结合，可以提供丰富的子窗口控制组合。用 CreateWindow( )或 CreateWindowEx( )函数创建窗口类控件，必须指定控件的窗口类，每个控件都应属于某个窗口类，这种窗口类也像主窗口的窗口类那样，在应用程序中定义并注册，但在更多的情况下，程序员使用的是 Windows 系统预定义的窗口类。

Windows 系统提供的标准控件主要包括：按钮控件、滚动条控件、静态控件、列表框控件、编辑框控件和组合框控件等。表 8-1 中列出了系统预定义的窗口类。

表 8-1 系统预定义的窗口类

窗口类名	说 明
BUTTON	代表一个按钮的小长方形的子窗口(按钮控件)
COMBOBOX	代表一个选择列表框的子窗口(组合框控件)
EDIT	代表一个接收用户输入的文本输入子窗口(编辑框控件)
LISTBOX	代表字符串列表的子窗口(列表框控件)
SCROLLBAR	代表一个滚动条的子窗口(滚动条控件)
STATIC	代表一个显示静态文本的子窗口(静态控件)

子窗口控件通过向其父窗口发送 WM_COMMAND 消息进行通信。WM_COMMAND 的 wParam 参数包含了子窗口的标志 ID，而 lParam 参数的低位是子窗口的句柄，高位是通知代码(notification code)，通知代码将告诉父窗口用户在子窗口中所进行的操作。不同的子

窗口控件类型发送给父窗口的通知代码是不一样的，表 8-2 中列出了不同的子窗口控件类型发送给父窗口的通知代码。

表 8-2 子窗口控件和相应的通知代码

子窗口控件	通知代码	说 明
按钮控件	BN_CLICKED	用户在按钮子窗口中单击
	BN_DOUBLECLICKED	用户在按钮子窗口中双击
	BN_SETFOCUS	按钮获得输入焦点
	BN_KILLFOCUS	按钮失去输入焦点
编辑框控件	EN_CHANGE	用户在编辑框子窗口中更改了输入框中的数据
	EN_ERRSPACE	编辑框的空间已用完
	EN_HSCROLL	水平滚动条被按下并被激活
	EN_KILLFOCUS	编辑框失去输入焦点
	EN_MAXTEXT	输入的正文数超过了编辑框的最大容量
	EN_SETFOCUS	编辑框子窗口获得输入焦点
	EN_UPDATE	编辑框子窗口将更新显示内容
	EN_VSCROLL	垂直滚动条被按下并激活
列表框控件	LBN_DBLCLK	字符串列表框中的字符串被双击
	LBN_ERRSPACE	分配给字符串列表框的内存已经用完
	LBN_KILLFOCUS	字符串列表框失去焦点
	LBN_SELCHANGE	在字符串列表框进行的选择发生了改变
	LBN_SETFOCUS	字符串列表框获得输入焦点
组合框控件	CBN_DBLCLK	选择列表框中的字符串被双击
	CBN_DROPDOWN	选择列表框将被取消
	CBN_EDITCHANGE	选择列表框中的正文将被修改
	CBN_EDITUPDATE	选择列表框中的正文将被更新
	CBN_ERRSPACE	分配给选择列表框的内存已用完
	CBN_KILLFOCUS	选择列表框失去焦点
	CBN_SELCHANGE	选择列表框中的选择项发生改变
	CBN_SETFOCUS	选择列表框获得焦点
滚动条控件		没有与滚动条相关的通知代码
静态控件		没有与静态文本框相关的通知代码

应用程序窗口可调用函数 SendMessage 向特定的子窗口发送消息，以指示其动作。例如用户单击单选按钮时，应用程序窗口可调用函数 SendMessage 向该单选按钮发送 BM SETCHECK 消息，为该按钮设置选中符号，其形式为：

```
SendMessage (hwndRadioButton,BM SETCHECK,1,0);
```

使用对话框控件时，应用程序可调用函数 SendDlgItemMessage 向指定的对话框控件发送消息，其形式为：

```
SendDlgItemMessage (hdlg,ID,message,wParam,lParam);
```

其中 message 为所发消息，应用程序向控件发送的消息的字参数与长参数包含该消息的相关信息，其含义取决于具体的控件消息。

8.2　常用子窗口型控件操作函数

应用程序对控件的操作，实质上是子窗口操作的一种类型(对话框控件可看作对话框的子窗口)。Windows 系统提供一系列的 API 函数完成对子窗口控件的操作，常用窗口函数的功能、原型及说明如下：

1. EnableWindow 函数

用户通过该函数，可以激活或禁止窗口，函数的原型如下：

```
BOOL EnableWindow (HWND hwnd,BOOL bEnable);
```

其中，bEnable 为激活或禁止标识，当 bEnable 值为 Ture 时该窗口可接收用户输入；bEnable 值为 False 时禁止该窗口接收输入。

2. MoveWindow 函数

利用该函数可以移动窗口或改变窗口的大小，函数的原型如下：

```
BOOL MoveWindow
(
HWND hwnd,
int X,int Y,
int nWidth,
int nHeight,
BOOL bReprint        //表明窗口移动后是否重画
);
```

bReprint 表明窗口移动后是否重画，如果父窗口的位置或大小改变，子窗口相对于父窗口的位置与大小不变。

3. SetWindowText 函数

利用该函数可以设置窗口的标题或文体，函数的原型如下：

```
BOOL SetWindowText (HWND hwnd,LPCTSTR lpszString)
```

其中 lpszString 为设置的文本内容。

4. GetWindowText 函数

利用该函数可以获取指定窗口标题或文本，该函数返回从指定窗口获取的文本字节数，函数的原型如下：

```
int GetWindowText
(
HWND hwnd,
LPSTR lpszString,     //存放文本字符串的缓冲区
int nMax              //最大可获取的字符串长度
);
```

另外，对于对话框按钮控件，应用程序可调用函数 SetDlgItemText 和 GetDlgItemText 完成相同操作。

SetDlgItemText 函数原型为：

```
BOOL SetDlgItemText
(
HWND hdlg,              //对话框句柄
int ID,
LPCTSTR lpszString      //设置的文本内容
);
```

GetDlgItemText 函数的原型如下：

```
int GetDlgItemText
(
HWND hdlg,              //对话框句柄
int ID,
LPSTR lpszString,       //存放文本字符串的缓冲区
int nMax                //最大可获取的字符串长度
);
```

该函数返回从指定的控件获取的文本字节数。

5. DestroyWindow 函数

该函数可以关闭窗口，如果应用程序关闭父窗口，则其中的子窗口同时被关闭，函数的原型如下：

```
BOOL DestroyWindow (HWND hwnd )
```

8.3 按钮控件

8.3.1 按钮控件的分类及其特点

按钮控件是 Windows 对话框中最常用的控件之一。按钮控件的类型比较丰富，其中主要有普通按钮、单选按钮、复选框按钮和组框按钮等。

1. 普通按钮(PUSHBUTTON)与默认普通按钮(DEFPUSHBUTTON)

普通按钮和默认普通按钮是最常用的按钮，其外观为矩形条，按钮上可设置文本或图标、位图等。该类型按钮的作用是帮助用户触发指定动作。当用户单击按钮时，应用程序立即执行相应动作。

2. 单选按钮(RADIOBUTTON)与自动单选按钮(AUTORADIOBUTTON)

单选按钮的外形为按钮文本和其左侧的小圆框，当单选按钮被选中时，该项的圆框将加点显示。单选按钮所包含的各选项之间一般存在互斥性，即用户只能选择其中某个选项。

自动单选按钮与普通单选按钮的区别在于：当用户选择自动单选按钮时，系统可自动消除其他单选按钮的选中标志，以保证互斥性；普通单选按钮则要求程序员编写相应的程序完成互斥操作。

3. 复选框(CHECKBOX)

复选框的外形为按钮文本和其左侧的小方框，当复选框被选中时，该项的方框将加对号显示。

复选框常用来显示一组选项供用户选择。与单选按钮不同，其各选项之间不存在互斥性，用户可同时选择其中一个或多个选项。

4. 组框(GROUPBOX)

组框的外形为左上角包含文字的矩形框，组框是一种特殊的按钮形式，虽然它属于按钮类控件，但既不处理鼠标和键盘输入，也不向其父窗口发送消息，其主要作用在于将控件分隔成不同的组并加以说明。

8.3.2　按钮控件的创建过程

按钮控件的创建有两种方式，一种是创建控件型的子窗口，另一种是在对话框中定义控件。

1. 创建一个控件型的子窗口

创建一个窗口包括两个基本步骤，创建并注册一个窗口类的实例和创建并显示窗口实例。

一个窗口类提供了创建一个窗口所需要的基本信息，同时也为创建的窗口指定了一个处理其中的消息的函数。当创建一个窗口类时，实际上已创建了该窗口类的一个实例。在创建了窗口类的实例后，在无需再为每个窗口的 WNDCLASS 结构重定义许多值的情况下，就可以很容易地创建几个窗口了。

在定义了窗口类的实例后，下一步就是创建窗口本身。创建一个窗口时，可以使用两个 Windows 函数。第一个函数是前面介绍过的 CreateWindow 函数，另一个函数是 CreateWindowEx 函数，它除了提供几个扩展的风格位之外，与 CreateWindow 函数基本相同。在 Windows API 中 CreateWindowEx 的函数原型定义如下：

```
HWND CreateWindowEx( DWORD dwExStyle,   // 扩展的窗口风格
                     LPCTSTR lpClassName,  // 指向已注册的窗口类的指针
                     LPCTSTR lpWindowName, // 指向窗口名称的指针
                     DWORD dwStyle,        // 窗口风格
                     int x,                // 水平方向上的位置
                     int y,                // 垂直方向上的位置
                     int nWidth,           // 窗口宽度
                     int nHeight,          // 窗口高度
                     HWND hWndParent,      // 指向父窗口的句柄
                     HMENU hMenu,          // 指向菜单资源的句柄
                     HINSTANCE hInstance,  // 指向应用程序实例的句柄
                     LPVOID lpParam        // 指向创建窗口所需的参数的指针
                     );
```

其中，参数 dwExStyle 用于指定窗口的扩展风格，可以是下列值之一：

- WS_EX_DLGMODALFRAME：创建一个双边框窗口；如果同时定义了 dwStyle 的值为 WS_CAPTION，则窗口还包含一个标题栏。
- WS_EX_NOPARENTNOTIFY：创建一个子窗口；当子窗口创建或者撤消时，将不向父窗口发送 WM_PARENTNOTIFY 消息。

参数 lpClassName 参数用于指定用于创建窗口的窗口类，用户可以给该参数赋予自行定义的并已注册过的窗口类，同时也可以给参数赋予系统预定义的公共窗口类。

在 Windows API 中，CreateWindow 函数的形式如下：

```
CreateWindow ("BUTTON",                    //预定义窗口类名
    Title,                                 //按钮标题(文本或位图等)
    WS_CHILD | WS_VISIBLE |…,              //样式
    X,Y,                                   //按钮控件的起始坐标
    nWidth,nHeight,                        //按钮控件的宽度与高度
    hwndParant,                            //拥有该按钮控件的窗口句柄
    ID,                                    //按钮控件标识
    hInstance,                             //应用程序当前实例句柄
    NULL
    );
```

按钮作为一种子窗口形式，也具有一些窗口的样式，如 WS_CHILD，WS_VISIBLE 等。除此以外，按钮控件还具有许多独特的样式。表 8-3 所示为常用的按钮样式及其说明。

表 8-3 按钮控件样式及其说明

样 式	说 明	样 式	说 明
BS_PUSHBUTTON	普通按钮	BS_ICON	表示按钮上带有图标
BS_DEFPUSHBUTTON	默认按钮	BS_BITMAP	表示按钮上带有位图
BS_RADIOBUTTON	单选按钮	BS_CENTER	按钮上文本居中
BS_CHECKBOX	复选框	BS_LEFT	按钮上文本左对齐
BS_OWNERDRAW	自绘按钮	BS_RIGHT	按钮上文本右对齐
BS_TEXT	表示按钮上存在文本	BS_USERBUTTON	用户定义按钮

2. 在对话框定义中创建控件

在对话框定义中创建控件，其一般形式为：

Control-type[Title,]ID, X, Y, nWidth, nHeight, [, Style]

其中：

- Control-type 为控件类型，常用的对话框控件类型及其说明见表 8-4；

表 8-4 常用对话框控件类型及其说明

控件类型	说 明	控件类型	说 明
PUSHBUTTON	按钮控件	CTEXT	居中静态控件
DEFPUSHBUTTON	默认按钮控件	RTEXT	右对齐静态控件
CHECKBOX	复选框控件	EDITTEXT	编辑框控件
RADIOBUTTON	单选按钮控件	SCROLLBAR	滚动条控件
GROUPBOX	组合框控件	LISTBOX	列表框控件
LTEXT	左对齐静态控件	COMBOBOX	组合框控件

- title 为控件的标题或内容；
- X，Y 为控件的起始坐标；

- nWidth,nHeight 分别为控件的宽度与高度;
- ID 为控件标识;
- style 为控件样式。

控件的创建过程的重点是控件样式的选取。不同样式的选取不但关系到控件的外形,而且决定控件所能实现的功能,程序员必须根据实际需要采用相应控件样式。

8.3.3　按钮控件与应用程序之间的消息传递

应用程序创建按钮控件后,可通过接收控件发出的消息得知用户的请求,并可通过向指定按钮发送消息对其进行操作。

1. 接收按钮控件消息

在控件概述中已介绍,当用户与控件交互时,控件向其父窗口发送 WM_COMMAND 消息,该消息的 wParam 参数的低字节中包含控件的标识值(ID),高字节中包含控件向父窗口发出的通知代码。

2. 向按钮控件发送消息

应用程序可调用函数 SendMessage 向按钮控件发出消息,对其进行操作,其形式如下:

```
SendMessage (hwnd,message,wParam,lParam);
```

message 为向控件子窗口发出的消息,常用消息及其说明见表 8-5 所示。

表 8-5　按钮消息及其说明

消 息 名	说　明	消 息 名	说　明
BM_GETCHECK	返回单选按钮或复选框的选中状态	BM_SETSTATE	设置按钮或复选框的状态
BM_SETCHECK	设置单选按钮或复选框的选中状态	BM_GETIMAGE	返回按钮图标或位图句柄
BM_GETSTATE	返回按钮或复选框的样式	BM_SETIMAGE	设置按钮图标或位图句柄
BM_SETSTYLE	设置按钮或复选框的样式		

SendMessage 函数使用字参数确定同种消息中的不同动作,例如:

设置复选框和单选按钮的选中标志时字参数应取 1,其形式为:

```
SendMessage (hwnd,BM_SETCHECK,1,0);
```

取消选中标志时字参数应取 0,其形式为:

```
SendMessage (hwnd,BM_SETCHECK,0,0);
```

对于对话框按钮控件,应用程序可调用函数 SendDlgItemMessage 完成相同操作。

8.3.4　按钮控件应用实例

【例 8-1】

创建如图 8-1 和图 8-2 所示的按钮控件系列。当单击第一个按钮时,按钮上的文字"这里是一个按钮,按我吧!"就变成"你已按下了按钮";第二个按钮标记为"这是缺省按钮,按下试试看",当按下此按钮后,按钮的标记信息就变成"按钮已被按下";此外还有单选按钮、复选按钮、自动复选框及组框控件等。

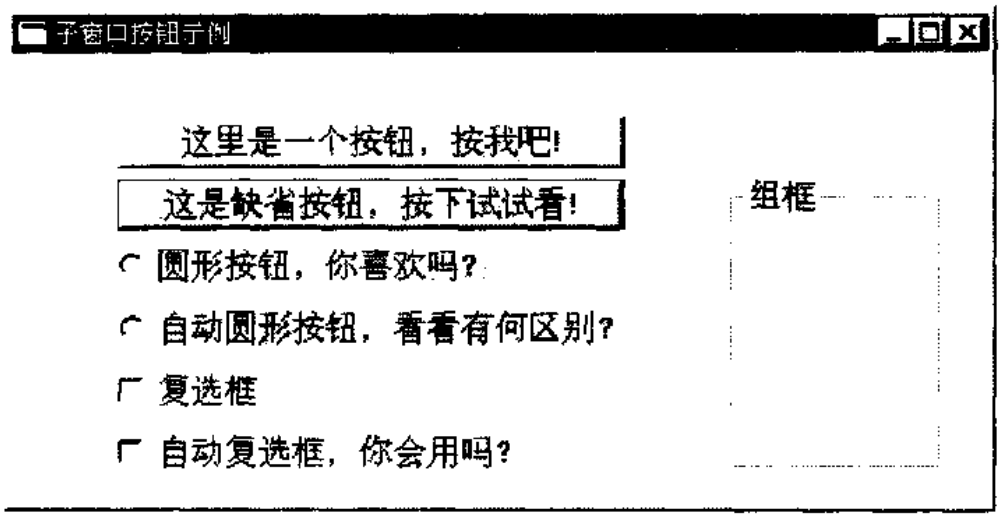


图 8-1 子窗口按钮示例

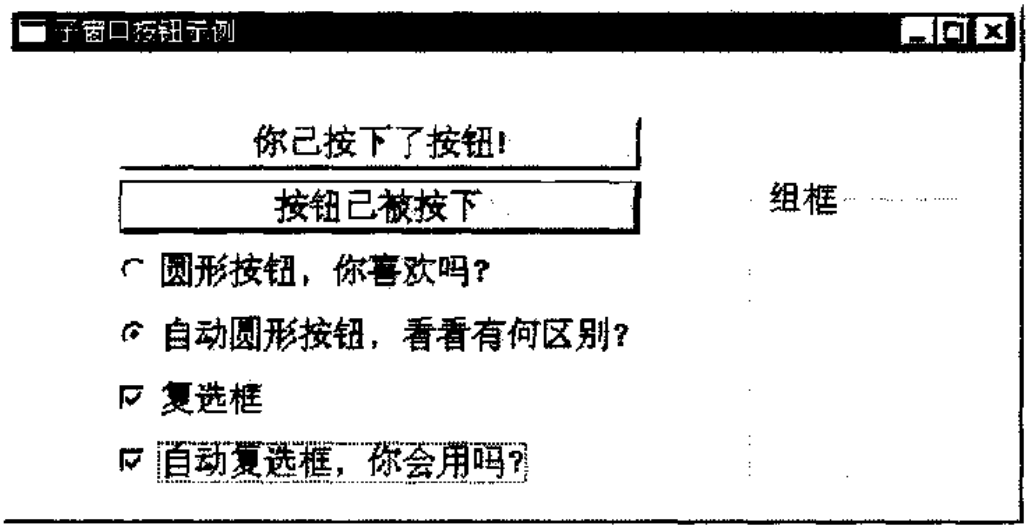


图 8-2 按钮被按下后的响应

本例的源代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "8-1.h"

HINSTANCE hInst;
char lpszClassName[] = "BUTTON"; //预定义窗口类名
int nCheckBoxSt;
HWND hwndPush,hwndDefPush,hwndRadio,hwndAutoRadio,hwndCheckBox,
     hwndAutoCheckBox,hwndGroupBox;//定义控件子窗口句柄
```

```
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM.LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine,
                   int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[] = "控件";
    char lpszTitle[] = "子窗口按钮示例";
    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI-APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC-ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;

    if( ! RegisterClass( &wndclass))
    {
      MessageBeep(0) ;
      return FALSE ;
      }

    hwnd = CreateWindow( lpszClassName,
                         lpszTitle,
                         WS-OVERLAPPEDWINDOW,
                         CW-USEDEFAULT,
                         CW-USEDEFAULT,
                         CW USEDEFAULT,
                         CW-USEDEFAULT,
                         NULL,
                         NULL,
                         hInstance,
                         NULL) ;
    hInst = hInstance;
    ShowWindow( hwnd, nCmdShow) ;
```

```
    UpdateWindow(hwnd);
    while( GetMessage(&Msg, NULL, 0, 0))
    {
      TranslateMessage( &Msg) ;
      DispatchMessage( &Msg) ;
    }
return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    int X = 50,Y = 30; //控件起始坐标
    switch(message)
    {
      //通过响应 WM_CREATE 消息创建控件子窗口
    case WM_CREATE:
        //创建按钮子窗口
        hwndPush = CreateWindow(lpszClassName1,
                                "这里是一个按钮,按我吧!",
                                BS_PUSHBUTTON|WS_CHILD|WS_VISIBLE,
                                X,Y,
                                240,25,
                                hwnd,
                                (HMENU)IDB_PUSHBUTTON,
                                hInst,
                                NULL);
        //创建默认按钮子窗口
        hwndDefPush = CreateWindow(lpszClassName1,
                                "这是默认按钮,按下试试看!",
                                BS_DEFPUSHBUTTON|WS_CHILD|WS_VISIBLE,
                                X,Y + 30,
                                240,25,
                                hwnd,
                                (HMENU)IDB_DEFPUSHBUTTON,
                                hInst,
                                NULL) ;

        //创建单选按钮控件子窗口
        hwndRadio = CreateWindow(lpszClassName1,
```

```
                    "单选按钮,你喜欢吗?",
                    BS_RADIOBUTTON|WS_CHILD|WS_VISIBLE,
                    X,Y+60,
                    240,25,
                    hwnd,
                    (HMENU)IDB_RADIOBUTTON,
                    hInst,
                    NULL);

//创建自动单选按钮子窗口
hwndAutoRadio = CreateWindow(lpszClassName1,
                    "自动圆形按钮,看看有何区别?",
                    BS_AUTORADIOBUTTON|WS_CHILD|WS_VISIBLE,
                    X, Y+90,
                    240,25,
                    hwnd,
                    (HMENU)IDB_AUTORADIOBUTTON,
                    hInst,
                    NULL);

//创建复选框控件子窗口
hwndCheckBox = CreateWindow(lpszClassName1,
                        "复选框",
                        BS_CHECKBOX|WS_CHILD|WS_VISIBLE,
                        X,Y+120,
                        240,25,
                        hwnd,
                        (HMENU)IDB_CHECKBOX,
                        hInst,
                        NULL);

//创建自动复选框控件子窗口
hwndAutoCheckBox = CreateWindow(lpszClassName1,
                    "自动复选框,你会用吗?",
                    BS_AUTOCHECKBOX|WS_CHILD|WS_VISIBLE,
                    X, Y+150,
                    240,25,
                    hwnd,
                    (HMENU)IDB_AUTOCHECKBOX,
                    nInst,
                    NULL);
```

```
        //创建组框控件子窗口
        hwndGroupBox = CreateWindow(lpszClassName1,
                                    "组框",
                                    BS_GROUPBOX|WS_CHILD|WS_VISIBLE,
                                    X+290, Y+30,
                                    100,140,
                                    hwnd,
                                    (HMENU)IDB_GROUPBOX,
                                    hInst,
                                    NULL);

          break;
//处理控件消息
case WM_COMMAND:
  switch(LOWORD(wParam))
  {
    //当用户单击按钮时,改变按钮文本
    case IDB_PUSHBUTTON:
        SetWindowText(hwndPush,"你已按下了按钮!");
        break;

    //当用户单击按钮时,改变按钮文本
    case IDB_DEFPUSHBUTTON:
        SetWindowText(hwndDefPush,"按钮已被按下");
        break;

    //当用户单击单选按钮时,设置按钮为选中状态
    case IDB_RADIOBUTTON:
        SendMessage(hwndRadio,BM_SETCHECK,1,0);
        break;

    case IDB_AUTORADIOBUTTON:
        break;

    case IDB_CHECKBOX:
        //根据复选框状态设置复选框
        //如果当前为选中状态,则设置为非选中状态
        if (nCheckBoxSt == 0)
        {
          SendMessage(hwndCheckBox,BM_SETCHECK,1,0);
          nCheckBoxSt = 1;
        }
```

```
                //如果当前为非选中状态,则设置为选中状态
                else
                {
                  SendMessage(hwndCheckBox,BM_SETCHECK,0,0);
                  nCheckBoxSt = 0;
                }
                break;
            case IDB_AUTOCHECKBOX:
                break;
            case IDB_GROUPBOX:
                break;
          }
          break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return 0;
    }
```

本例的头文件如下：

```
#define IDB_PUSHBUTTON          10
#define IDB_DEFPUSHBUTTON       11
#define IDB_RADIOBUTTON         12
#define IDB_AUTORADIOBUTTON     13
#define IDB_CHECKBOX            14
#define IDB_AUTOCHECKBOX        15
#define IDB_GROUPBOX            16
#define IDB_GROUPBUTTON         17
```

8.4　滚动条控件

滚动条控件是 Windows 窗口操作中常用的工具。

8.4.1　滚动条控件的特点与分类

滚动条控件最直接的功能是当应用程序显示的内容超过窗口的范围时，用户可通过拖动滚动条遍历整个窗口内容。滚动条在功能上分为垂直与水平滚动条，分别实现窗口内容纵向和横向的滚动。

滚动条在形式上又可分为窗口滚动条和子窗口滚动条控件(包括对话框滚动条)两

种。窗口滚动条由系统创建,其位置和尺寸固定;子窗口滚动条由应用程序创建,其位置和尺寸由程序员确定。

此外,滚动条控件还可作为调节音量、颜色的工具。在应用程序开发中常见到类似的滚动条使用方法。

8.4.2 滚动条控件的创建

Windows 应用程序中滚动条的创建有以下三种方法。

1. 创建窗口滚动条

应用程序在主窗口的创建过程中通过设置窗口类的样式为 WS_VSCROLL 和 WS_HSCROLL 实现窗口滚动条的创建,其形式为:

```
hwnd = CreateWindow( lpszClassName,
                     lpszTitle,
                     //在窗口的创建样式中加入滚动条样式
                     WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     NULL,
                     NULL,
                     hInstance,
                     NULL  ) ;
```

创建的窗口流动条具有缺省位置和默认的滚动框形状。

2. 创建滚动条控件子窗口

子窗口滚动条控件的创建与其他控件相似,由应用程序调用函数 CreateWindow 实现,其形式如下:

```
CreateWindow ("SCROLLBAR",                    //预定义窗口类名
              lpszScrollTitle,                //滚动条标题,常为 NULL
              WS_CHILD | WS_VISIBLE…,         //滚动条样式
              X,Y,                            //滚动条起始坐标
              nWidth,nHeight,                 //滚动框宽度与高度
              hwndParant,                     //拥有该滚动条的窗口句柄
              ID,                             //滚动条标识
              hInstance,                      //应用程序当前实例句柄
              NULL
              ) ;
```

其中,滚动条样式除窗口类样式外,常用的还有 SBS_VERT 和 SBS_HORZ,分别表示垂直滚动条和水平滚动条。

3. 创建对话框中的滚动条控件

对话框中滚动条在对话框的定义中完成创建操作，其一般形式如下：

```
SCROLLBAR ID,X,Y,nWidth,nHeight,[style]
```

应用程序创建滚动条控件后，即可通过该控件发出的消息得知用户对滚动条的操作，并可调用 API 函数操作指定的滚动条。当用户操作窗口滚动条滚动窗口时，滚动条向应用程序发出 WM_VSCROLL 或 WM_HSCROLL 消息。消息中的字参数(wParam)低字节中包含了滚动条动作的标识，表 8-6 列出了常用的标识及其说明。

表 8-6　常用滚动条动作标识及其说明

标　识	说　明	标　识	说　明
SB_TOP	滚动到滚动条顶端或最右端	SB_LINEDOWN	向下滚动一行
SB_BOTTOM	滚动到滚动条底端或最左段	SB_LINEUP	向上滚动一行
SB_RIGHT	滚动到右边	SB_LINELEFT	向左滚动一行
SB_LEFT	滚动到左边	SB_LINERIGHT	向右滚动一行
SB_PAGEUP	向上滚动一页	SB_THUMBPOSITION	滚动框移动到新位置
SB_PAGEDOWN	向下滚动一页	SB_THUMBTRACK	滚动框被拖动
SB_PAGELEFT	向左滚动一页	SB_ENDSCROLL	滚动到最终位置
SB_PAGERIGHT	向右滚动一页		

其中比较特殊的是 SB_THUMBPOSITION 标识和 SB_THUMBTRACK 标识，当滚动条发出这两条消息时，消息的字参数高字节中包含了滚动条的位置。

作为子窗口控件的滚动条，其消息发送与按钮等其他控件有所不同。滚动条控件并不发出 WM_COMMAND 消息，而是与窗口滚动条一样，发送 WM_VSCROLL 或 WM_HSCROLL 消息。

两者的区别在于消息的长参数(lParam)所包含的内容不同。滚动条控件消息的长参数的高字节中包含控件句柄，而窗口滚动条消息的长参数高字节则为 0。

此外，可应用 API 函数操作滚动条控件，例如设置滚动块位置等。

8.4.3　常用滚动条操作函数

常用滚动条操作函数有：

1. SetScrollRange 函数

该函数的功能是设置滚动条的滚动范围，函数原型如下：

```
int SetScrollRange
(
  HWND hwnd,      //窗口句柄
  int nBar,       //滚动条特性标识，常用标识及其说明见表 8-7 所示
  int nMaxPos,    //滚动条最大滚动位置
  int nMinPos,    //滚动条最小滚动位置
  BOOL bRedraw    //当滚动条改变时，是否重绘标志；非 0 表示重绘
  );
```

值得提出的是：窗口句柄对应于不同的对象，它对应着响应的窗口句柄，如对于窗口

滚动条为窗口句柄；对于独立滚动条子窗口为该子窗口句柄；对于对话框控件为对话框句柄。

表 8-7 常用滚动条特性标识及其说明

标 识	说 明
SB_CTL	表示该滚动条为控件滚动条
SB_HORZ	表示该滚动条为窗口水平滚动条
SB_VERT	表示该滚动条为窗口垂直滚动条

滚动条的滚动位置为一组离散值，例如设置滚动条的滚动范围为 0～99，则该滚动条共有 100 个可能的位置。

2. SetScrollPos 函数

利用该函数可以设置滚动块的位置，函数的原型如下：

```
int SetScrollPos
(
    HWND hwnd,      //窗口句柄，同 SetScrollRange 函数
    int nBar,       //滚动条的特性标识，同 SetScrollRange 函数
    int nPos,       //滚动块的位置，其取值应在滚动条滚动范围之内
    BOOL bRedraw    //意义同 SetScrollRange 函数
);
```

3. GetSystemMetrics 函数

应用程序常使用 GetSystemMetrics 函数获取系统提供的滚动条尺寸，其形式有两种：

获取系统水平滚动条的高度：GetSystemMetrics(SM_CYHSCROLL)

获取系统垂直滚动条的宽度：GetSystemMetrics(SM_CXVSCROLL)

4. GetScrollpos 函数

利用该函数可以获取滚动块的位置，函数的原型如下：

```
int GetScrollPos
(
  HWND hwnd,            //窗口句柄，同 SetScrollRange 函数
  int nBar              //滚动条特性标识
);
```

函数返回当前滚动块的位置。

5. GetScrollRange 函数

利用该函数可以获取滚动条的滚动范围，函数的原型如下：

```
BOOL GetScrollRange
(
  HWND hwnd,            //窗口句柄，同 SetScrollRange 函数
  int nBar,             //滚动条特性标识
  LPINT lpMinPos,       //接收最小位置的整形变量的地址
  LPINT lpMaxPos        //接收最大位置的整形变量的地址
);
```

8.4.4　滚动条控件应用实例

【例 8-2】

本例要求创建一个带滚动条的颜色框，在颜色框中，通过拉动滚动条，使其显示的区域产生变化，同时，可以通过颜色的设置来改变颜色框的颜色，如图 8-3 所示。

本例的运行结果如图 8-3 所示。

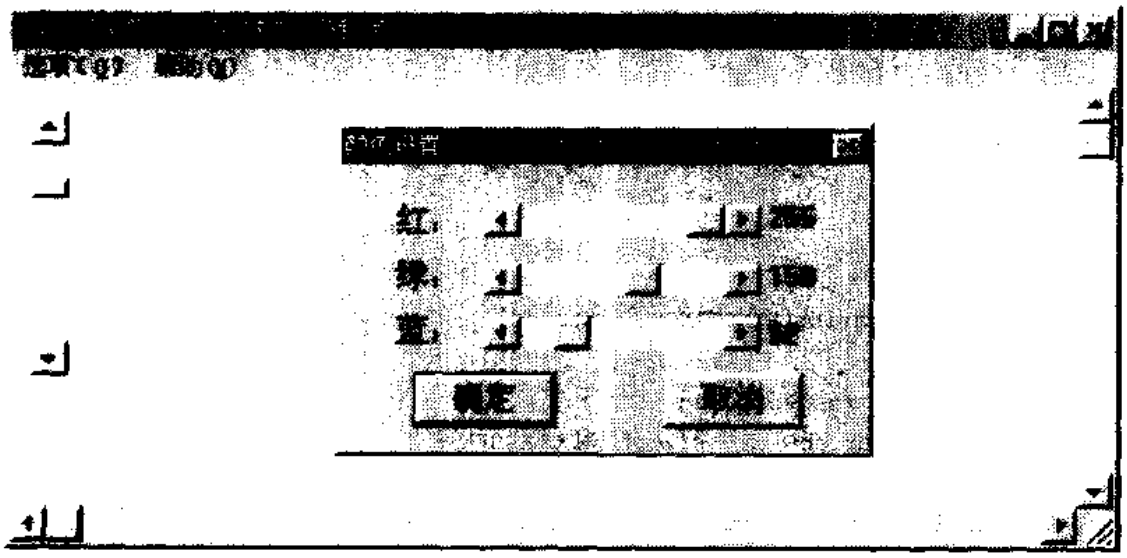


图 8-3　【例 8-2】的运行结果

本例的源程序代码如下：

```
//介绍滚动条控件的编辑使用方法

# include < windows.h >
# include < string.h >
# include "8_2.h"

int APIENTRY WinMain
        (
        HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow
        )

{
    MSG msg;
    if (! InitApplication(hInstance))
    return(FALSE);
```

```
    if (! InitInstance(hInstance,nCmdShow))
    return(FALSE);

    SendMessage(hWndMain,WM_COMMAND,IDM_DRAW,0L);
    while(GetMessage(&msg,NULL,0,0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

    return(msg.wParam);

}

BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASSEX wcexScrllBar;
    wcexScrllBar.cbSize = sizeof(WNDCLASSEX);
    wcexScrllBar.style = 0;
    wcexScrllBar.lpfnWndProc = (WNDPROC)MainWndProc;
    wcexScrllBar.cbClsExtra = 0;
    wcexScrllBar.cbWndExtra = 0;
    wcexScrllBar.hInstance = hInstance;
    wcexScrllBar.hIcon = LoadIcon(hInstance,"ScrollBarIcon");
    wcexScrllBar.hCursor = LoadCursor(NULL,IDC_ARROW);
    wcexScrllBar.hbrBackground = GetStockObject(WHITE_BRUSH);
    wcexScrllBar.lpszMenuName = "ScrollBarMenu";
    wcexScrllBar.lpszClassName = "ScrollBarWClass";
    wcexScrllBar.hIconSm = LoadIcon(hInstance,"SmallIcon");
    return (RegisterClassEx(&wcexScrllBar));
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance;
    hWndMain = CreateWindow(
        "ScrollBarWClass",
        "滚动条控件的编程方法示范程序",
        WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
```

```
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
        );

    if (! hWndMain)
        return(FALSE);

    crRGB[0] = 0;
    crRGB[1] = 0;
    crRGB[2] = 0;

    ShowWindow(hWndMain, nCmdShow);
    UpdateWindow(hWndMain);
    return(TRUE);
}

LRESULT APIENTRY MainWndProc(HWND hWnd,UINT message, WPARAM
        wParam, LPARAM lParam)
    {
        HDC hDC;
        HBRUSH hBrush;
        static HWND hWndScrollBar;
        static SCROLLINFO stScrollInfo;
        static int nRectWidth;

        switch(message)
        {
        case WM_CREATE:
            hWndScrollBar = CreateWindowEx(0L,
            "SCROLLBAR",
            (LPSTR)NULL,
            WS_CHILD | SBS_VERT,
            10,
            30-GetSystemMetrics(SM_CYVSCROLL),
            19,
            100 + 2 * GetSystemMetrics(SM_CYVSCROLL),
            hWnd,
            (HMENU)NULL,
            hInst,
```

```
        (LPVOID)NULL
        );

        stScrollInfo.cbSize = sizeof(SCROLLINFO);
        stScrollInfo.fMask = SIF_POS | SIF_RANGE | SIF_PAGE;
        stScrollInfo.nMin = 1;
        stScrollInfo.nMax = 100;
        stScrollInfo.nPos = 1;
        stScrollInfo.nPage = 10;

// 设置滚动条控件的各方面的特性参数
        SetScrollInfo(hWndScrollBar,SB_CTL,&stScrollInfo,FALSE);
        ShowWindow(hWndScrollBar,TRUE);
        nRectWidth = 100;
        break;

    case WM_VSCROLL:
        if ((HWND)lParam! = hWndScrollBar)
        break;

        switch(LOWORD(wParam))
        {
        case SB_LINEUP:
            nRectWidth += 1;
            break;
        case SB_LINEDOWN:
            nRectWidth -= 1;
            break;
        case SB_PAGEUP:
            nRectWidth += 10;
            break;
        case SB_PAGEDOWN:
            nRectWidth -= 10;
            break;
        case SB_THUMBPOSITION:
            nRectWidth = 101 - HIWORD(wParam);
            break;
        default:
            return(DefWindowProc(hWnd,message,wParam,lParam));
        }

        if(nRectWidth < 1)
```

```
                nRectWidth = 1;
            else if (nRectWidth > 100)
                nRectWidth = 100;
        stScrollInfo.fMask = SIF_POS;
        stScrollInfo.nPos = 101 - nRectWidth;
        SetScrollInfo(hWndScrollBar,SB_CTL,&stScrollInfo,TRUE);
        SendMessage(hWndMain,WM_COMMAND,IDM_DRAW,0L);
        break;

    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDM_RGB:
            DialogBox(hInst,"RGBDlg",hWnd,(DLGPROC)RGBDlgProc);
            break;
        case IDM_DRAW:
            hDC = GetDC(hWnd);
            hBrush = CreateSolidBrush(RGB(255,255,255));
            SelectObject(hDC,hBrush);
            Rectangle(hDC,30,30,130,130);
            hBrush = CreateSolidBrush(RGB(crRGB[0],crRGB[1],crRGB[2]));
            SelectObject(hDC,hBrush);
            Rectangle(hDC,30,130 - nRectWidth,30 + nRectWidth,130);
            ReleaseDC(hWnd,hDC);
            break;
        case IDM_EXIT:
            PostQuitMessage(0);
            break;
        case IDM_ABOUT:
            DialogBox(hInst,"AboutBox",hWnd,(DLGPROC)About);
            break;
        default:
            return(DefWindowProc(hWnd,message,wParam,lParam));
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return(DefWindowProc(hWnd,message,wParam,lParam));
    }
return(0);
```

```
}
BOOL APIENTRY RGBDlgProc (HWND hDlg,UINT message,WPARAM wParam,
                          LPARAM lParam)
{

    static int CurrentVal[3],PrevVal[3];
    static HWND hwScroll[3];
    char szString[10];
    int i;

    switch(message)
    {
    case WM_INITDIALOG:
    // 分别设置滚动条的范围和初始位置
        for(i = 0;i < 3;i ++ )
        {
            CurrentVal[i] = crRGB[i];
            PrevVal[i] = crRGB[i];
            hwScroll[i] = GetDlgItem(hDlg,IDC_RED + i);
            SetScrollRange(hwScroll[i],SB_CTL,0,255,FALSE);
            SetScrollPos(hwScroll[i],SB_CTL,CurrentVal[i],TRUE);
            SetDlgItemText(hDlg,IDC_VALRED + i,
            itoa(CurrentVal[i],szString,10));
        }

        SetFocus(hwScroll[0]);
        break;

    //处理滚动条通知消息
    case WM_HSCROLL:
    //首先确定本条消息来自于哪个滚动条
        i = 0;
        while(hwScroll[i]! = (HWND)lParam)
        i ++ ;
        switch(LOWORD(wParam))
        {
        case SB_LINEDOWN:
            CurrentVal[i] ++ ;
            if(CurrentVal[i] > 255)
            CurrentVal[i] = 255;
            break;
        case SB_LINEUP:
```

```
            CurrentVal[i] - - ;
            if(CurrentVal[i] < 0)
            CurrentVal[i] = 0;
            break;
        case SB_PAGEDOWN:
            CurrentVal[i] + = (255 - CurrentVal[i])/2;
            break;
        case SB_PAGEUP:
            CurrentVal[i] - = CurrentVal[i]/2;
            break;
        case SB_THUMBTRACK:
            CurrentVal[i] = HIWORD(wParam);
            break;
        case SB_TOP:
            CurrentVal[i] = 0;
            break;
        case SB_BOTTOM:
            CurrentVal[i] = 255;
            break;
        }
        //设定滚动条的位置及其对应的文字
        SetScrollPos(hwScroll[i],SB_CTL,CurrentVal[i],TRUE);
        SetDlgItemText(hDlg,IDC_VALRED + i,
        itoa(CurrentVal[i],szString,10));
        crRGB[i] = CurrentVal[i];
        SendMessage(hWndMain,WM_COMMAND,IDM_DRAW,0L);
        break;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDOK:
            for(i = 0;i < 3;i ++ )
                crRGB[i] = CurrentVal[i];
            EndDialog(hDlg,TRUE);
            return(TRUE);

        case IDCANCEL:
            for(i = 0;i < 3;i ++ )
                crRGB[i] = CurrentVal[i];
            EndDialog(hDlg,TRUE);
            return (TRUE);
```

```
        }
        break;
    }
    return(FALSE);
}
BOOL APIENTRY About(HWND hDlg,UINT message,WPARAM wParam,LPARAM
                    lParam)
{
    switch(message)
    {
    case WM_INITDIALOG:
        return(TRUE);
    case WM_COMMAND:
        if(LOWORD(wParam) == IDOK||LOWORD(wParam) == IDCANCEL)
        {
        EndDialog(hDlg,TRUE);
        return(TRUE);
        }
        break;
    }
    return(FALSE);
}
```

本例的头文件代码如下：

```
//"选项"菜单
#define IDM_RGB          101
#define IDM_EXIT         102
#define IDM_DRAW         103

//"帮助"菜单
#define IDM_ABOUT        200

// 对话框标识符
#define IDC_RED          500
#define IDC_GREEN        501
#define IDC_BLUE         502
#define IDC_VALRED       505
#define IDC_VALGREEN     506
#define IDC_VALBLUE      507
#define IDC_STATIC       -1

//全局变量
```

```
HINSTANCE hInst;
HWND hWndMain;
int crRGB[3];

// 函数原型说明

// 过程
LRESULT APIENTRY MainWndProc(HWND,UINT,WPARAM,LPARAM);
BOOL APIENTRY About(HWND,UINT,WPARAM,LPARAM);
BOOL APIENTRY RGBDlgProc(HWND,UINT,WPARAM,LPARAM);

// 函数
BOOL InitApplication(HINSTANCE);
BOOL InitInstance(HINSTANCE,int);
```

本例的资源文件代码如下：

```
//资源文件 Scrllbar.rc
#include "windows.h"
#include "8_2.h"

//Menu

SCROLLBARMENU MENU DISCARDABLE
BEGIN
    POPUP "选项(&O)"
        BEGIN
            MENUITEM "颜色设置(&C)…",        IDM_RGB
            MENUITEM SEPARATOR
            MENUITEM "退出(&X)",              IDM_EXIT
        END
    POPUP "帮助(&H)"
        BEGIN
            MENUITEM "关于(&A)…",             IDM_ABOUT
        END
END

// Dialog

ABOUTBOX DIALOG DISCARDABLE 20, 25, 124, 77
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "关于使用滚动条编程的示范"
FONT 14,"隶书_GB2312"
```

```
BEGIN
    PUSHBUTTON "确定",IDOK,43,51,40,14
    CTEXT      "本程序用来说明滚动条的使用",IDC_STATIC,4,9,115,8
    CTEXT      "VC++教材",IDC_STATIC,30,24,76,10,NOT WS_GROUP
    CTEXT      "一九九九年九月",IDC_STATIC,36,37,65,8,NOT WS_GROUP
    ICON       "SCROLLBARICON",IDC_STATIC,11,23,18,20
    CONTROL    "",IDC_STATIC,"Static",SS_BLACKFRAME,40,34,56,1
END

RGBDLG DIALOG DISCARDABLE 10,80,132,73
STYLE DS_MODALFRAME | WS_POPUP   WS_CAPTION | WS_SYSMENU
CAPTION "颜色设置"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON  "确定",IDOK,18,53,36,14
    PUSHBUTTON     "取消",IDCANCEL,81,53,35,14
    SCROLLBAR      IDC_RED,36,10,69,8,WS_TABSTOP
    SCROLLBAR      IDC_GREEN,36,25,69,8,WS_TABSTOP
    SCROLLBAR      IDC_BLUE,36,39,69,8,WS_TABSTOP
    LTEXT          "红:",IDC_STATIC,14,10,15,10
    LTEXT          "绿:",IDC_STATIC,14,24,15,10
    LTEXT          "蓝:",IDC_STATIC,14,38,15,10
    LTEXT          "0",IDC_VALRED,107,9,23,10
    LTEXT          "0",IDC_VALGREEN,107,24,23,10
    LTEXT          "0",IDC_VALBLUE,107,39,23,10
END
```

8.5　静态控件

静态控件是一种包含正文或图形的小窗口，应用程序通常使用静态控件标记其他控制窗口或分隔不同组别的控件。

8.5.1　静态控件的特点

一般情况下，静态控件既不接收用户输入也不发出消息，其特点与前面介绍的组框类似。然而，应用程序可通过设置静态控件的样式使其能够响应用户输入，向应用程序发送消息。这时的静态文本在功能上相当于超文本。

8.5.2　静态控件的创建

创建静态控件常用形式有两种：

1. 创建静态控件子窗口

应用程序调用函数 CreateWindow 创建该类型的静态控件，其形式为：

```
CreateWindow("STATIC",                          // 预定义窗口类名
             lpszStaticTitle,                   // 静态控件标题(文本或位图等)
             WS_CHILD | WS_VISIBLE|WS_BORDER|…,  // 样式，详见表 8-8
             X,Y,                               // 静态控件起始坐标
             nWidth,nHeight,                    // 静态控件宽度与高度
             hwndParant,                        // 拥有该静态控件的窗口句柄
             ID,                                // 静态控件标识
             HInstance,                         // 应用程序当前实例句柄
             NULL
             |
```

表 8-8　常用静态控件样式及其说明

样　式	说　明	样　式	说　明
SS_LEFT	左对齐文本	SS_CENTERIMAGE	位图居中
SS_RIGHT	右对齐文本	SS_NOTIFY	用户在静态控件中单击时，强制发出消息
SS_CENTER	文本居中		
SS_ICON	包含图标	SS_USERITEM	用户定
SS_BITMAP	包含位图		

2. 创建对话框中的静态控件

静态控件作为对话框控件时，其创建过程在对话框的定义中完成，其一般形式为：

```
Static-Type          // Static-Type 为静态控件类型关键字
Tilte,               //静态控件标题(文本或位图等)
ID,                  //静态控件标识
X,Y,                 //起始坐标
nWidth,nHeight,      //滚动框的宽度和高度
[,style]             //滚动条样式
```

其中：Static-Type 为静态控件类型关键字，常用类型有：LTEXT(左对齐文本)、RTEXT(右对齐文本)和 CTEXT(文本居中)。

8.5.3　静态控件消息的发送

一般情况下静态控件不发送消息。但在实际应用中，常需要静态文本能够像超文本那样响应用户的输入，向应用程序发送控件消息。这时应用程序需在创建静态控件时加入 SS_NOTIFY 样式。该样式允许静态控件向其父窗口发送 WM_COMMAND 消息，该消息的字参数(wParam)的低字节中包含静态控件的 ID，高字节中包含通知码，表 8-9 列出了静态控件可使用的通知码及其说明；长参数 lParam 中包含该静态控件的句柄。

表 8-9 静态控件使用的通知码及其说明

通知码	说 明	通知码	说 明
STN_CLICKED	单击静态控件	STN_ENABLE	激活静态控件
STN_DBLCLK	双击静态控件	STN_DISABLE	禁止静态控件

8.5.4 静态控件应用实例

【例 8-3】

本例通过演示位图静态控件的创建与操作方法，说明静态控件消息的强制生成与处理过程。如图 8-4 所示的位图控件，当单击位图时，就报告该位图的尺寸，如图 8-5 所示。

本例的源程序代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
//定义静态控件标识
#define IDS_BITMAP 101
char lpszClassName1[] = "STATIC";
char lpszText[] = "";
HWND hwndStatic; //静态控件句柄
HINSTANCE hInst;
HDC hdc;
```

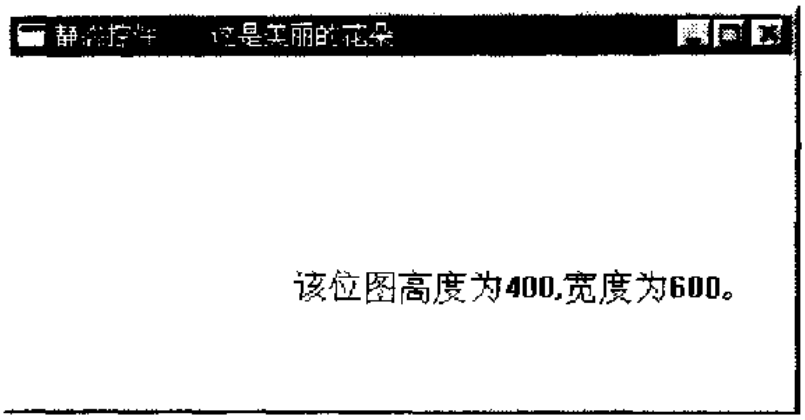


图 8-5　【例 8-3】的运行结果

```
HBITMAP hBm;
BITMAP bm;
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInst,
                    LPSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[] = "控件";
    char lpszTitle[] = "静态控件 - - 这是美丽的花朵";
    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;
    if( ! RegisterClass( &wndclass))
    {
    MessageBeep(0) ;
    return FALSE ;
    }
    hInst = hinstance;
```

```
    hwnd = CreateWindow(lpszClassName,
                        lpszTitle,
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL,
                        NULL,
                        hInstance,
                        NULL) ;
    hBm = LoadBitmap(hInst,"Internet Wall"); //加载位图资源
    GetObject( hBm, sizeof( BITMAP), (LPVOID)&bm) ; //获取位图信息
    sprintf(lpszText,"该位图高度为%d,宽度为%d",bm.bmHeight,bm.bmWidth);
    ShowWindow( hwnd, nCmdShow) ;
    UpdateWindow(hwnd);
    while( GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage( &Msg) ;
        DispatchMessage( &Msg) ;
    }
    return Msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch(message)
    {
        case WM_CREATE:
        //创建位图静态控件
        hwndStatic = CreateWindow(lpszClassName1,
                    "Internet-Wall",
                    //设置静态控件风格
                    WS_CHILD|WS_VISIBLE|WS_BORDER|SS_BITMAP
                    |SS_NOTIFY,//该风格允许静态控件强制发送消息
                    30,30,
                    0,0, //系统将根据位图尺寸自动调节静态控件的大小
                    hwnd,
                    (HMENU)IDS_BITMAP,
```

```
                hInst,
                NULL) ;
            break;

        case WM_COMMAND:
        switch (LOWORD(wParam))
        {
            //处理控件消息
            case IDS_BITMAP:
                DestroyWindow(hwndStatic);
                hdc = GetDC(hwnd);
                //在窗口中显示位图大小信息
                TextOut(hdc,130,100,lpszText,strlen(lpszText));
                ReleaseDC(hwnd,hdc);
                break;
        }
            break;
    case WM_DESTROY:
        DeleteObject(hBm);
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

本例的资源文件如下：

```
Internet Wall BITMAP E:\Internet_Wall.bmp
```

8.6　列表框控件

8.6.1　列表框控件的特点

列表框常用于集中显示同种类型的内容，如同类型文件等。列表框一般具有如下特点：

(1) 可提供大量的可选项(需要时自动显示滚动条)；

(2) 可设置单选(单个选项)或多选(多项选择)功能；

(3) 单选时，单击列表项，被选的项以“反相”显示表示被选中；再次单击该选项，恢复为非选中状态。

8.6.2　列表框控件的创建

创建列表框控件也有两种方式，一种是创建子窗口列表框控件，另一种是创建对话框

控件的列表框。

1. 创建子窗口列表框控件

应用程序调用函数 CreateWindow 创建该类列表框，其形式为：

```
CreateWindow("LISTBOX",                    //预定义窗口类名
             NULL;                         //列表框一般无标题
             WS_CHILD|WS_VISIBLE|…,        //样式，常用样式及其说明见表 8-10
             X,Y,                          //列表框起始坐标
             nWidth,nHeight,               //列表框宽度与高度
             hwndParant,                   //拥有该列表框的窗口句柄
             ID,                           //列表框标识
             hInstance,                    //应用程序当前实例句柄
             NULL
             );
```

表 8-10　列表框常用样式及其说明

样　式	说　明	样　式	说　明
LBS_NOTIFY	可向父窗口发出消息	LBS_EXTENDEDSEL	扩展多选列表框
LBS_SORT	按字母顺序排列表项	LBS_NOREDRAW	列表框改变时不重画
LBS_MULTIPLESEL	多选列表框	LBS_STANDARD	标准样式

其中 LBS_STANDARD 代表列表框最常用的样式组合：

```
WS_BORDER|WS_VSCROLL|LBS_SORT|LBS_NOTIFY
```

2. 创建作为对话框控件的列表框

该类型列表框的创建在应用程序中对话框的定义中完成，其形式如下：

```
LISTBOX ID,X,Y,nWidth,nHeight,[,style]
```

其中：

- LISTBOX：控件关键字。
- nWidth，nHeight：列表框的宽度和高度。
- ID：列表框标识。
- style：列表框的样式。
- X，Y：起始坐标。

8.6.3　列表框控件与应用程序之间的消息传递

应用程序创建列表框控件后，可通过接收控件发出的消息得知用户的请求，并可通过向列表框发送消息对其进行操作。

1. 列表框向应用程序发送消息

当用户与列表框交互时，列表框向应用程序发出 WM_COMMAND 消息。该消息字参数(wParam)的高字节为标识列表框动作的消息通知码(如 LBN_DBLCLK 标识用户双击)；低字节为控件标识值。应用程序中常用的列表框通知码及其说明见表 8-11。

表 8-11　常用列表框通知码及其说明

通　知　码	说　　明
LBN-SELCHANGE	表明列表框中的用户选择已发生改变
LBN-DBCLK	双击
LBN-SELCANCLE	列表框中的选择被取消
LBN-SETFOCUS	列表框收到输入焦点
LBN-KILLFOCUS	列表框失去输入焦点

2. 应用程序向列表框发送消息

应用程序对列表框的操作通过调用函数 SendMessage 或 SendDlgItemMessage 向其发送各种消息完成。表 8-12 为常用列表框消息和说明。

表 8-12　常用列表框消息及其说明

标　　识	说　　明	标　　识	说　　明
LB-ADDFILE	在文件列表中加入指定文件	LB-GETTEXTLEN	获取指定项长高
LB-ADDSTRING	在列表框中加入列表项	LB-GETTOPINDEX	获取列表框中第一项的索引值
LB-DELETESTRING	在列表框中删除列表项		
LB-DIR	在列表框中列出指定文件	LB-INSERTSTRING	在列表框的指定位置加入一项
LB-FINDSTRING	在列表框中查找指定项		
LB-GETCOUNT	获取多选列表框中的项数	LB RESETCONTENT	清空列表框
LB-GETCURSEL	获取列表框中当前选中项的索引值	LB-SETSEL	设置多选列表框中指定项的选中状态
LB-GETSEL	获取列表框中指定项的选中状态	LB-SETCURSEL	设置单选列表框中指定项的选中状态
LB-GETSELCOUNT	获取多选列表框中选中的项数	LB-SETTOPINDEX	设置列表框中第一项的索引值
LB-GETTEXT	获取指定项文本		

应用程序向列表框发送消息时应注意下列问题：

(1) 索引号的使用

① 应用程序通过列表框中项的索引值操作指定项，其中第一项的默认索引号为 0，最后一项的索引号为 -1。例如，在列表框的第 3 个位置插入项的语句为：

```
SendMessage(hwndlist, LB_INSERTSTRING,2,(LPARAM)szNewItem);
```

szNewItem 为插入项文本。

② 应用程序常通过向列表框发送 LB-GETCURSEL 消息获取当前选中的索引号，形式为：

```
nIndex = SendMessage(hwndlist,LB_GETCURSEL,0,0L);
```

(2) 多选列表框消息发送

当应用程序设置列表框的样式为 LBS-EXTENDEDSEL 和 LBS-MULTIPLSEL 时，用户可在该列表框中选取多个项。当用户选择列表框中的多个项时，应用程序可通过向列表框发送 LB-GETSELCOUNT 消息获取选中的项数，其形式为：

```
nCount = SendMessage(hwndlist,LB_GETSELCOUNT,0,0L);
```

其中 nCount 为返回选中项数。

此外,应用程序还可向列表框发送 LB_GETSEL 消息获取指定项的选中状态,其形式为:

```
nSelect = SendMessage(hwndlist,LB_GETSEL,nIndex,0L);
```

其中 nSelect 为返回该项的选中状态。若为非 0 则表明索引号为 nIndex 的列表项被选中。

(3) 发送 LB_DIR 消息

列表框常见的用法是显示一系列文件名、目录或驱动器,这一功能由应用程序向列表框发出 LB_DIR 消息实现。该消息的字参数(wParam)表示文件属性值,长参数(lParam)表示所列文件的特征。常用的形式为:

```
nFileNum = SendMessage(hwndlist,LB_DIR,FILE_LIST,(LPARAM)"*.exe");
```

其中:

- nFileNum:函数返回列表框中指定文件的数目;
- hwndlist:列表框句柄;
- FILE_LIST:表示所列文件属性值的 wParam 参数,表 8-13 所示为系统规定的文件属性值及其说明。

表 8-13 常用文件属性值及其说明

数值(16 进制)	说 明	数值(16 进制)	说 明
4000	列出驱动器名	0002	列出隐含文件名
0000	列出普通文件名	0004	列出系统文件名
0001	列出只读文件名	0010	列出上述文件及子目录名

如"0x4010"表示列出驱动器名及当前目录的子目录名和所有文件名。

8.6.4 列表框控件应用实例

【例 8-4】

创建一个采用常用样式的单选列表框,并在该列表框中列出磁盘驱动器名、当前目录的文件和子目录名等,如图 8-6 所示。

本例的源程序代码如下:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#define IDL_SAMPLE 101 //定义列表框标识
#define FILE_LIST 0x4010 //定义文件属性值
HWND hwndList;
HINSTANCE hInst;
char lpszClassName[] = "ListBox";
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
```

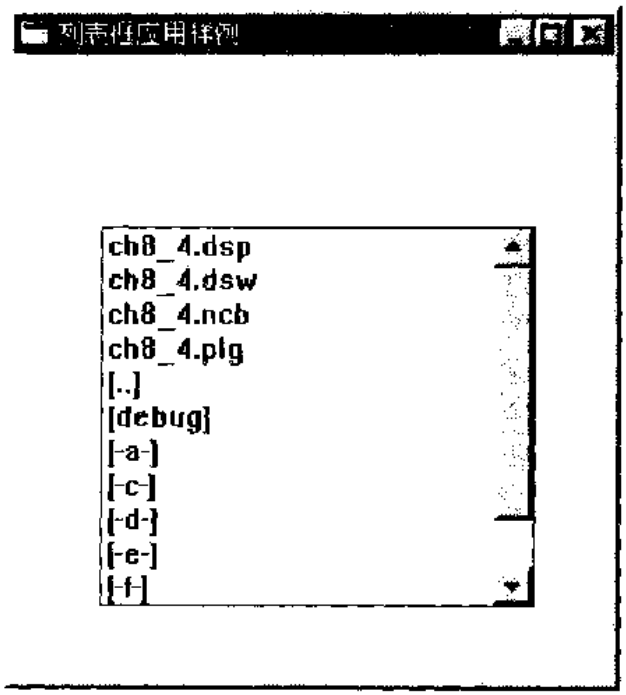


图 8-6　【例 8-4】的运行结果

```
int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInst,
                    LPSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[ ] = "控件";
    char lpszTitle[ ] = "列表框应用样例";
    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;
    if( ! RegisterClass( &wndclass))
    {
      MessageBeep(0) ;
```

```
    return FALSE ;

  hwnd = CreateWindow(lpszClassName,
                      lpszTitle,
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      NULL,
                      NULL,
                      hInstance,
                      NULL) ;
  ShowWindow( hwnd, nCmdShow) ;
  UpdateWindow(hwnd);
  hInst = hInstance;
  while( GetMessage(&Msg, NULL, 0, 0))
      {
          TranslateMessage( &Msg) ;
          DispatchMessage( &Msg) ;
      }
  return Msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hwnd,
                      UINT message,
                      WPARAM wParam,
                      LPARAM lParam)
{

  int nIndex;
  switch(message)
  {
      case WM_CREATE:
          hwndList = CreateWindow(lpszClassName1,
                      NULL,
                      //设置列表框风格
                      LBS_EXTENDEDSEL|LBS_STANDARD|WS_CHILD|WS_VISIBLE|LBS_
                      MULTIPLESEL,
                      40, 80,
                      200,180,
                      hwnd,
```

```
                    (HMENU)IDL_SAMPLE,
                    hInst,
                    NULL) ;
            //设置列表框中的所列文件类型
            SendMessage(hwndList,LB_DIR,FILE_LIST,(LPARAM)"*.*");
            break;
        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
            //处理列表框消息
            case IDL_SAMPLE :
                if(HIWORD(wParam) == LBN_DBLCLK)
                {
                    //获取当前选中项的索引值
                    nIndex = SendMessage(hwndList,LB_GETCURSEL,0,0L);
                    //删除该列表项
                    SendMessage(hwndList,LB_DELETESTRING,nIndex,0L);
                }
            break;
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

8.7　编辑框控件

8.7.1　编辑框控件的特点

编辑框的主要作用在于接收用户键盘输入，用户可在编辑框中编辑输入。编辑框是 Windows 应用程序中一种重要的交互工具。

编辑框控件通过选取相应的样式可具有以下功能特点：

- 可自动转换用户输入，如大小写转换等；
- 可使用某个字符(如 *)代替显示用户的输入；
- 可支持多行编辑等。

8.7.2 编辑框控件的创建

编辑框控件的创建也有两种方式：

1. 创建编辑框子窗口控件

应用程序调用函数 CreateWindow 创建该类编辑框，其形式为：

```
CreateWindow("EDIT",                    //预定义窗口类名
             NULL,                      //编辑框一般无标题
             WS_CHILD|WS_VISIBLE|…,     //样式，常用的样式及其说明见表 8-14
             X,Y,                       //编辑框起始坐标
             nWidth,nHeight,            //编辑框的宽度与高度
             hwndParant,                //拥有该编辑框的窗口句柄
             ID,                        //编辑框标识
             hInstance,                 //应用程序当前实例句柄
             NULL
             );
```

表 8-14 编辑框常用样式及其说明

样 式	说 明	样 式	说 明
ES_AUTOHSCROLL	当用户输入超出窗口时，窗口自动水平滚动，该样式只能与 ES_MULTILINE 一起使用	ES_RIGHT	文本右对齐显示
		ES_READONLY	只读编辑框
		ES_UPPERCASE	自动将输入大写显示
ES_AUTOVSCROLL	当用户输入超出窗口时，窗口自动垂直滚动	ES_LOWERCASE	自动将输入小写显示
		ES_PASSWORD	以"*"显示用户输入
ES_CENTER	文本居中显示	ES_MULTILINE	多行编辑框
ES_LEFT	文本左对齐显示		

2. 创建对话框中的编辑框控件

该类型编辑框的创建在应用程序中对话框的定义中完成，其形式如下：

```
EDITTEXT ID,X,Y,nWidth,nHeight,[,style]
```

其中，EDITTEXT 为控件关键字；Style 为编辑框的样式。

8.7.3 编辑框控件与应用程序之间的消息传递

应用程序创建编辑框控件后，可通过接收控件发出的消息得知用户的请求，并可通过向编辑框发送消息对其进行操作。

1. 编辑框向应用程序发送消息

与列表框相似，编辑框通过向其父窗口发送 WM_COMMAND 消息通知应用程序用户的交互信息。该消息字参数(wParam)的低字节为控件标识；高字节为标识列表框动作的消息通知码。常用的通知码及其说明见表 8-15 所示。

表 8-15　编辑框常用通知码及其说明

通知码	说　明	通知码	说　明
EN-SETFOCUS	编辑框获取输入焦点	EN MAXTEXT	编辑框中的用户输入已达到允许的最大字节数
EN-KILLFOCUS	编辑框失去输入焦点		
EN-CHANGE	编辑框的内容发生改变	EN-HSCROLL	编辑框中的内容水平滚动
EN-UPDATE	编辑框的内容被更新	EN VSCROLL	编辑框中的内容垂直滚动

2. 应用程序向编辑框发送消息

应用程序对编辑框的操作通过调用函数 SendMessage 或 SendDlgItemMessage 向其发送各种消息完成。表 8-16 所示为常用编辑框消息及其说明。

表 8-16　常用编辑框消息及其说明

消　息	说　明	消　息	说　明
EM-GETRECT	获取编辑框矩形的尺寸	EM-REPLACESEL	替换编辑框中的选中文本
EM-SETRECT	设置编辑框矩形的尺寸	EM-SETPASSWORDCHAR	设置密码编辑框中的替代字符
EM-LINESCROLL	设置滚动条滚动的步长		
EM-SETHANDLE	设置输入内容缓冲区句柄	EM-GETPASSWORDCHAR	获取密码编辑框中的替代字符
EM-GETHANDLE	获取输入内容缓冲区句柄		
EM-LINELENGTH	获取文本行长度	EM-SETREADONLY	设置编辑框为只读
EM-GETFONT	获取编辑框使用的字体	EM-GETSEL	获取编辑框中的选中文本
EM-GETLINECOUNT	获取多行编辑框的文本行数	SETSEL	设置编辑框中的选中文本

8.7.4　编辑框控件应用实例

【例 8-5】

本例介绍一个包含编辑框控件的"乘法器"实例程序。通过此例，读者可以理解编辑框控件的创建过程。如图 8-7 所示，当读者在乘数的输入框中输入一个数，然后单击"×"（乘号），再在乘数的输入框中输入另一个数，在结果输出框中就可以看到运算的结果。如本例第一次输入数字 5，第二次输入数字 8，得到的结果为 40。

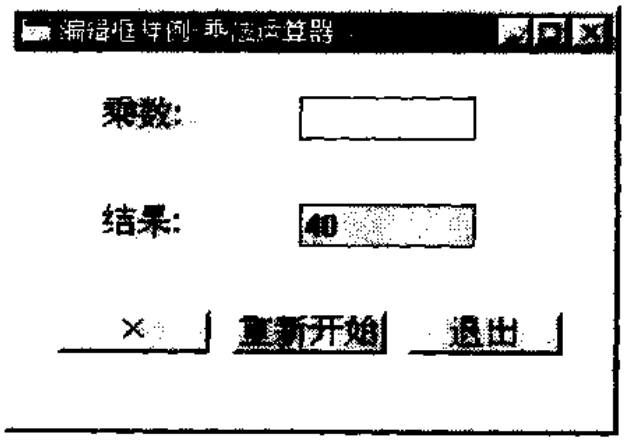


图 8-7　利用编辑框控件编制"乘法运算器"界面

本例的源程序代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "8-5.h"
HWND hEditItem,hEditResult,                          //编辑框句柄
     hButtonAdd,hButtonReset,hButtonCancel,          //按钮句柄
     hStatic1,hStatic2;                              //静态文本句柄
HINSTANCE hInst;
char lpszAddItem[]="";
char lpszResult[]="";
char *stop;          //为函数 strtol 定义的终止字符指针(详见程序说明)
long nAddItem=1,nResult=1;
int nMax;
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine,
                   int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[] = "编辑框";
    char lpszTitle[] = "编辑框样例-乘法运算器";
    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW);
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;
    if( ! RegisterClass( &wndclass))
    {
        MessageBeep(0) ;
        return FALSE ;
    }
    hwnd = CreateWindow(lpszClassName,
```

```
                    lpszTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    280,200,
                    NULL,
                    NULL,
                    hInstance,
                    NULL) ;
    ShowWindow( hwnd, nCmdShow) ;
    UpdateWindow(hwnd);
    hInst = hInstance;
    while( GetMessage(&Msg, NULL, 0, 0))
    {
        TranslateMessage( &Msg) ;
        DispatchMessage( &Msg) ;
    }
    return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch(message)
    {
    case WM_CREATE:

        //定义各种子窗口控件
        hStatic1 = CreateWindow( "STATIC",
                                 "乘数:",
                                 WS_CHILD|WS_VISIBLE,
                                 40,20,
                                 50,20,
                                 hwnd,
                                 (HMENU)IDS_1,
                                 hInst,
                                 NULL) ;

        hStatic2 = CreateWindow( "STATIC",
```

```
                                "结果:",
                                WS_CHILD|WS_VISIBLE,
                                40,70,
                                50,20,
                                hwnd,
                                (HMENU)IDS_2,
                                hInst,
                                NULL) ;

    hEditItem = CreateWindow( "EDIT",
                                NULL,
                                WS_CHILD|WS_VISIBLE|ES_LEFT|WS_BORDER,
                                130,20,
                                80,20,
                                hwnd,
                                (HMENU)IDE_ADDITEM,
                                hInst,
                                NULL) ;
    hEditResult = CreateWindow( "EDIT",
                                NULL,
                                WS_CHILD|WS_VISIBLE|ES_LEFT|WS_BORDER|
                                ES_READONLY,
                                130,70,
                                80,20,
                                hwnd,
                                (HMENU)IDE_RESULT,
                                hInst,
                                NULL) ;

    hButtonAdd = CreateWindow( "BUTTON",
                                "×",
                                WS_CHILD|WS_VISIBLE,
                                20,120,
                                70,20,
                                hwnd,
                                (HMENU)IDB_ADD,
                                hInst,
                                NULL) ;

    hButtonReset = CreateWindow( "BUTTON",
                                "重新开始",
```

```
                                WS_CHILD|WS_VISIBLE,
                                100,120,
                                70,20,
                                hwnd,
                                (HMENU)IDB_RESET,
                                hInst,
                                NULL) ;

        hButtonCancel = CreateWindow( "BUTTON",
                                "退出",
                                WS_CHILD|WS_VISIBLE,
                                180,120,
                                70,20,
                                hwnd,
                                (HMENU)IDB_CANCLE,
                                hInst,
                                NULL) ;
//设置"结果"编辑框的初始值
            SetWindowText(hEditResult,"0");
            break;
        case WM_SETFOCUS:
            //设置编辑框获取输入焦点
            SetFocus(hEditItem);
            break;
        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
            case IDB_ADD:
                                                    //获取编辑框中用户输入的文本长度
                nMax = GetWindowTextLength(hEditItem) + 1;
                                                        //获取编辑框中的用户输入
                GetWindowText(hEditItem,lpszAddItem,nMax);
                                                    //将用户输入的字符串转换成整形
                nAddItem = strtol(lpszAddItem,&stop,10);
                nResult += nAddItem;
                //将结果转换成字符串
                _ltoa(nResult,lpszResult,10);
                //在"结果"编辑框中输出结果
                SetWindowText(hEditResult,lpszResult);
                //在"乘数"编辑框中清除用户输入,准备接收新的输入
                SetWindowText(hEditItem,"");
```

```
            //使"乘数"编辑框重新得到输入焦点
            SendMessage(hwnd,WM_SETFOCUS,0,0L);
            break;
        case IDB_RESET:
            //在"乘数"编辑框中清除用户输入,准备接收新的输入
            SetWindowText(hEditItem,"");
            //使"乘数"编辑框重新得到输入焦点
            SendMessage(hwnd,WM_SETFOCUS,0,0L);
            //恢复"结果"编辑框的初始状态
            SetWindowText(hEditResult,"0");
            nResult = 1;
            nAddItem = 1;
            break;
        case IDB_CANCLE:
            SendMessage(hwnd,WM_DESTROY,0,0L) ;
            break;
            :
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return 0;
}
```

本例的头文件代码如下：

```
#define IDS_1               101
#define IDS_2               102
#define IDE_RESULT          103
#define IDE_ADDITEM         104
#define IDB_RESET           105
#define IDB_CANCLE          106
#define IDB_ADD             107
```

8.8 组合框控件

8.8.1 组合框控件的特点

组合框是两种预定义窗口的组合形式。在 Windows 编程中使用单一控件往往不能完全满足与用户交互的需要，最常见的组合框例子是列表及与其相联系的静态文本和编

辑框。Windows 应用程序中大量使用该类型的组合框。常用的组合框中列表框以隐藏的形式出现在编辑框下，当用户单击编辑框右侧的箭头时将弹出列表框。

8.8.2　组合框控件的创建

组合框的创建也分为两种形式：

1. 创建独立子窗口组合框

应用程序调用函数 CreateWindow 创建该类组合框，其形式为：

```
CreateWindow("COMBOBOX",          //预定义窗口类名
             NULL,                //组合框一般无标题
             CBS-DROPDOWN|…,      //样式
             X,Y,                 //组合框起始坐标
             nWidth,nHeight,      //组合框的宽度与高度
             hwndParent,          //拥有该组合框的窗口句柄
             ID,                  //组合框标识
             hInstance,           //应用程序当前实例句柄
             NULL
             )
```

组合框样式除包含各组成部分样式外，还有其独特样式，这些样式及说明见表 8-17。

表 8-17　组合框常用样式及其说明

样　式	说　明
CBS-DROPDOWN	组合框由列表框和编辑框组成，列表框平时不可见
CBS-DROPDOWNLIST	组合框由列表框和静态文本组成，列表框平时不可见
CBS-AUTOHSCROLL	编辑框中自动水平滚动
CBS-SORT	列表框中各项按字母顺序排列
CBS-SIMPLE	组合框中列表框可见

值得注意的是，创建组合框生成的句柄并不能操作组合框中的任一部分，只能操作整个组合框。

2. 创建作为对话框控件的组合框

该类型组合框的创建在应用程序中对话框的定义中完成，其形式如下：

```
COMBOBOX ID,X,Y,nWidth, nHeight, [,style]
```

其中 COMBOBOX 为控件关键字，style 为组合框的样式。

8.8.3　组合框控件与应用程序之间的消息传递

应用程序创建组合框后，可通过接收控件发出的消息得知用户的请求，并可通过向组合框控件发送消息对其进行操作。

1. 组合框向应用程序发送消息

组合框通过向其父窗口发送 WM-COMMAND 消息通知应用程序用户的交互信息。该消息字参数(wParam)的低字节为控件标识，高字节为标识组合框动作的消息通知码。常

用通知码见表 8-18。

表 8-18 组合框常用通知码及其说明

通知码	说 明	通知码	说 明
CBN_SELCHANGE	组合框中列表框部分所选中项发生改变	CBN_EDITUPDATE	组合框中的编辑框将显示修改过的文本
CBN_DBLCLK	双击	CBN_DROPDOWN	组合框中的列表框将下拉
CBN_SETFOCUS	组合框收到输入焦点	CBN_CLOSEUP	组合框中的列表框将隐藏
CBN_KILLFOCUS	组合框失去输入焦点		
CBN_EDITCHANGE	组合框中的编辑框中的文本发生改变		

2. 应用程序向组合框发送消息

应用程序对组合框的操作通过使用函数 SendMessage 或 SendDlgItemMessage 向组合框发送消息进行。对组合框的操作实际上是对组合框中各成员的操作。例如，应用程序向组合框中的列表框发出 CB_ADDSTRING 消息，可在列表框中加入一项。表 8-19 所示为常用的组合框消息及其说明。

表 8-19 常用组合框消息及其说明

消 息	说 明	消 息	说 明
CB_SHOWDROPDOWN	显示下拉列表框	CB_GETCURSEL	获取列表框中的选中项的索引值
CB_ADDSTRING	在列表框中加入新项	CB_GETCOUNT	获取列表框中的项的数目
CB_DELETESTRING	在列表框中删除新项	CB_GETLBTEXT	获取列表框中的指定项的文本
CB_INSERTSTRING	在列表框中插入新项	CB_GETLBTEXTLEN	获取列表框中指定项的文本长度
CB_FINDSTRING	在列表框中查询列表项	CB_LIMITEXT	限制编辑框中的字符串长度
CB_RESETCONTENT	清空列表框	CB_GETEDITSEL	获取编辑框中的选择
CB_DIR	在列表框中显示指定目录及文件	CB_SETEDITSEL	设置编辑框中的选择
CB_SETCURSEL	设置列表框中的选中项，该项将在编辑框中显示		

8.8.4 组合框控件应用实例

【例 8-6】

本例创建一个如图 8-8 所示的组合框控件，当单击下三角按钮时，显示可选文件的名字。

本例的源程序代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
//定义组合框标识
```

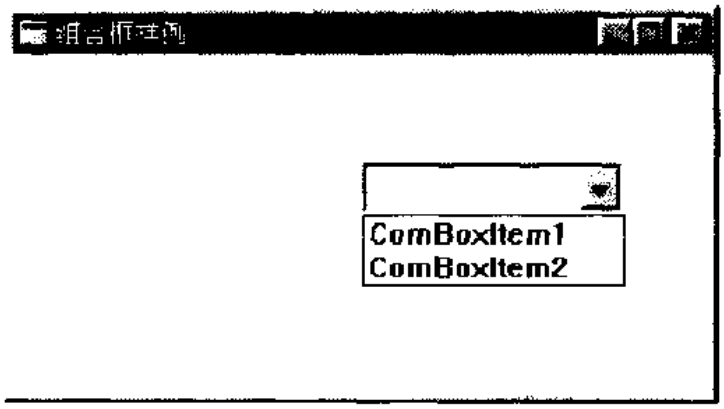


图 8-8　组合框控件的应用

```
#define IDC_SAMPLE1 101
#define IDC_SAMPLE2 102
#define IDC_SAMPLE3 103

HWND hComBox1,hComBox2,hComBox3;
HINSTANCE hInst;
char lpszClassName1[] = "COMBOBOX";

LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain( HINSTANCE hInstance,
          HINSTANCE hPrevInst,
          LPSTR lpszCmdLine,
          int nCmdShow)
{
    HWND hwnd ;
    MSG Msg ;
    WNDCLASS wndclass ;
    char lpszClassName[] = "控件";
    char lpszTitle[] = "组合框样例";
    wndclass.style = 0;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon = LoadIcon( NULL, IDI_APPLICATION) ;
    wndclass.hCursor = LoadCursor( NULL, IDC_ARROW) ;
    wndclass.hbrBackground = GetStockObject( WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = lpszClassName ;
    if( ! RegisterClass( &wndclass))
```

```
        }
        MessageBeep(0) ;
        return FALSE ;
        }
    hwnd = CreateWindow(lpszClassName,
                        lpszTitle,
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL,
                        NULL,
                        hInstance,
                        NULL) ;

    ShowWindow( hwnd, nCmdShow) ;
    UpdateWindow(hwnd);
    hInst = hInstance;
    while( GetMessage(&Msg, NULL, 0, 0))
    {
          TranslateMessage( &Msg) ;
          DispatchMessage( &Msg) ;
    }
    return Msg.wParam;

LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch(message)
    {
        case WM_CREATE:
    //创建由列表框和静态控件组成的组合框
            hComBox2 = CreateWindow(lpszClassName1,
                                    NULL,
                                    CBS_DROPDOWNLIST|WS_CHILD|WS_VISIBLE,
                                    160,50,
                                    120,150,
                                    hwnd,
```

```
                                (HMENU)IDC_SAMPLE2,
                                hInst,
                                NULL) ;
            SetFocus(hComBox2);
        //在各组合框中加入列表项
            SendMessage(hComBox2,CB_ADDSTRING,0,(LPARAM)"ComBoxItem1");
            SendMessage(hComBox2,CB_ADDSTRING,0,(LPARAM)"ComBoxItem2");
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd,message,wParam,lParam);
    }
    return 0;
}
```

习　题

8-1　控件有哪些类型?

8-2　如何创建滚动条控件?

8-3　按钮控件的特点是什么?

8-4　如何创建列表框控件?

8-5　如何创建编辑框控件?

8-6　如何创建组合框控件?

8-7　创建一个水平滚动条控件,用这个控件来控制字体的大小,字体大小由 50 到 100 变化;同时创建一个垂直滚动条,用此滚动条来控制字体的颜色,颜色由 RGB(0,0,255)到 RGB(0,0,0)变化。

8-8　创建一个下拉列表框,框中显示由 A 到 Z 的大写字母。

8-9　创建一个列表框,框中显示当前硬盘的路径,并提供对路径的选择。

8-10　创建一个编辑框控件,在此编辑框中输入某一个英文单词,在另外一个输出框中输出此单词和该单词的字符数。

8-11　创建一个组合框,在该组合框中,安排一组单选按钮,内容分别为“北京”、“上海”、“天津”;创建一组复选框,内容为“最高气温”和“平均气温”;再创建另一组合框显示温度值“30”和“20”;创建一个“确定”按钮,单击“确定”按钮后,在编辑框中显示选择的结果。

第 9 章

文件及剪贴板操作

文件操作是 Windows 应用程序经常要涉及的内容，利用 Windows 的 API 函数，能很好地实现文件操作的功能。

9.1 文件操作

9.1.1 Windows 98 文件操作的特点

Windows 98 是 32 位操作系统，最大文件长度可达 4GB。因此，同 16 位操作系统相比，一次可读写的文件内容大大增加，操作速度也大大加快。同时，由于 32 位操作系统可支持长达 56 字节的文件名，因此可以用长文件名，以更清楚地识别文件的功能及含义。此外，由于 Windows 系统是多任务多进程的操作系统，允许多个应用程序或一个程序的多个实例同时运行。因此，允许同一文件被不同进程访问。

9.1.2 常用的文件操作函数

1. CreateFile 函数

该函数用以创建一个文件，或打开一个现存文件，如果该函数调用成功，则返回一个文件句柄，应用程序可使用该文件句柄执行文件读写操作。该函数的原型如下：

```
HANDLE CreateFile
(
LPCTSTR lpszName,                          //文件名
DWORD dwDesiredAccess,                     //文件访问形式标识，读操作为 GENERIC
                                           //READ，写操作为 GENERIC-WRITE
DWORD dwShareMode,                         //文件的共享模式标识
LPSECURITY_ATTRIBUTES lpSecurityAttributes, //指向安全属性数据结构的指针，常设置
                                           //为 NULL
DWORD dwCreationDisposition,               //文件创建操作标识，常用的标识及其说
                                           //明见表 9-1
DWORD dwFlagsAndAttributes,                //文件属性标识，常用的属性标识及其说
                                           //明见表 9-2
HANDLE hTemplateFile                       //指向文件属性模板的句柄
);
```

表 9-1　文件创建操作常用标识及其说明

标　识	说　明
CREATE_NEW	创建一个文件，如果该文件已存在，则函数调用失败
CREATE_ALWAYS	创建一个文件，如果该文件已存在，则该文件被重写
OPEN_EXISTING	打开一个文件，如果该文件不存在，则函数调用失败
OPEN_ALWAYS	打开一个文件，如果该文件不存在，则创建该文件
TRUNCATE_EXITING	打开一个文件，如果该文件不存在，则函数调用失败；如果文件存在，则将其截为 0 字节长

表 9-2　常用文件属性标识及其说明

标　识	说　明
FILE_ATTRIBUTE_NORMAL	常规文件
FILE_ATTRIBUTE_READONLY	只读文件
FILE_ATTRIBUTE_HIDDEN	隐藏文件
FILE_ATTRIBUTE_SYSTEM	系统文件
FILE_ATTRIBUTE_ARCHIVE	归档文件
FILE_FLAG_WRITE_THROUGH	不经过缓冲区直接向磁盘写文件
FILE_FLAG_NO_BUFFERING	不经过缓冲区直接打开文件
FILE_FLAG_RANDOM_ACCESS	该文件将被随机存取
FILE_FLAG_SEQUENTIAL_SCAN	该文件将按从开始到结束的方式顺序存取

其中：

- dwShareMode：文件共享模式有三类，若该文件不能共享，则其值为 0；若该文件为可读共享，其值为 FILE_SHARE_READ；若该文件为可写共享，则其值为 FILE_SHARE_WRITE。
- lpSecurityAttributes：指向安全属性数据结构 SECURITY_ATTRIBUTES 的指针。该数据结构的定义如下：

```
typedef struct tagSECURITY_ATTRIBUTES
{
  DWORD    nlength;                    //该数据结构的长度
  LPVOID    lpSecurityDescriptor;      //安全描述符指针
  BOOL    bInheritHandle;              //是否继承句柄标识
}SECURITY_ATTRIBUTES;
```

- hTemplateFile：该参数常用来表示一个指向文件属性模板的句柄。当创建文件时，该模板可为该文件提供包括文件安全描述符、文件属性及文件扩展属性等内容。

以上属性标识在不冲突的基础上可组合使用，例如一个只读、并可随机存取的文件属性为：

```
FILE_ATTRIBUTE_READONLY|FILE_FLAG_RANDOM_ACCESS
```

当应用程序调用函数 CreateFile 创建新文件时，该文件具有以下特征：

(1) 文件的长度为 0 字节；

(2) 如果 hTemplateFile 参数不为 NULL，则该文件属性由此句柄提供的文件属性模板

设定；如果 hTemplateFile 参数为 NULL，则该文件的属性由 dwFlagsAndAttributes 和 lpSecurityAttributes 等参数提供。

此外，应用程序还可通过调用该函数打开指定的系统管道。例如打开 COM1 的语句如下：

```
hCom1 = CreateFile ("COM1"
                    GENERIC_READ | READ | GENERIC_WRITE,
                    0,
                    NULL,
                    0,
                    OPEN_EXITING,
                    NULL) ;
```

函数返回打开后的管道句柄，应用程序可使用该句柄读写指定管道。

2. ReadFile 函数

该函数的功能是从文件指针的位置开始读取一个现存文件中的数据，该函数的原型如下：

```
BOOL ReadFile
(
HANDLE hFile,                          //打开的文件句柄
LPVOID lpBuffer,                       //指向接收读入数据缓冲区地址的指针
DWORD nNumberOfBytesToRead,            //将从文件中读入的数据字数
LPDWORD lpNumberOfBytesRead,           //指向实际读取字节数的指针
LPOVERLAPPED lpOverlapped              //指向一个 OVERLAPPED 数据结构的指针
                                       //如果该参数设置为 NULL，则应用程序从
                                       //文件的当前位置读取数据
);
```

3. WriteFile 函数

WriteFile 函数是写文件函数之一。其作用是从文件指针的位置开始向一个现存文件中写入数据，该函数的原型如下：

```
BOOL WriteFile
(
HANDLE hFile,                          //打开的文件句柄
LPVOID lpBuffer,                       //指向写入数据的缓冲区地址的指针
DWORD nNumberOfBytesToWrite,           //将向文件中写入的数据字数
LPDWORD lpNumberOfBytesWritten,        //指向实际写入字节数的指针
LPOVERLAPPED lpOverlapped              //指向一个 OVERLAPPED 数据结构的指针
                                       //如果该参数设置为 NULL，则应用程序
                                       //从文件的当前位置写入数据
);
```

4. GetFileInformationByHandle 函数

通过 GetFileInformationByHandle 函数可以获取指定文件的信息，并将其保存在 BY

HANDL E_FILE_INFORMATION 数据结构中。该函数的原型如下：

```
BOOLGetFileInformationByHandle
(
HANDLE hFile,
LPBY-HANDEL-FILE-INFORMATION lpFileInformation
);
```

其中：

- hFile 为文件句柄。
- lpFileInformation 为指向 BY-HANDLE-FILE-INFORMATION 结构的指针。

BY-HANDLE-FILE-INFORMATION 结构的定义如下：

```
typedef struct tagBY-HANDLE-FILE-INFORMATION
{
DWORD dwFileAttributes;                    //文件的属性标识
FILETIME ftCreationTime;                   //文件创建时间
FILETIME ftLastAccessTime;                 //文件最后访问时间
FILETIME ftLastWriteTime;                  //文件最后写入时间
DWORD dwVolumnSerialNumber;                //文件的卷标号码
DWORD nFileSizeHigh;                       //文件大小的高字节
DWORD nFileSizeLow;                        //文件大小的低字节
DWORD nNumberOfLinks;                      //文件的链接号
DWORD nfileIndexHigh;                      //文件的索引号高字节
DWORD nfileIndexLow;                       //文件的索引号低字节
}BY-HANDLE-FILE-INFORMATION;
```

5. SearchPath 函数

SearchPath 函数是 Windows 应用程序经常使用的文件操作函数。其作用是根据给定搜索路径查找指定文件并返回指定文件的路径，该函数的原型如下：

```
DWORD SearchPath
(
LPCTSTR lpPath                  //指定搜索路径，若为 NULL，则采用默认路径
LPCTSTR lpFileName,             //搜索的文件名
LPCTSTR lPExtension,            //搜索文件的扩展名
DWORD nBufferLength,            //用来接收文件路径名的缓冲区长度
LPTSTR lpBuffer,                //指向接收文件路径名的缓冲区
LPTSTR * lPFilepart             //指向路径名中文件名部分地址的指针，如果调用该函数
                                //未指定搜索路径，则采用系统默认路径进行搜索
);
```

lpFileName 默认的搜索次序为：

(1) 当前目录。

(2) Windows 操作目录。

(3) Windows 系统所在目录。

(4) PATH 环境变量包含的目录。

6. SetFilePointer 函数

SetFilePointer 函数是 Windows 应用程序经常使用的文件操作函数，通过此函数，可以设置文件指针的位置，该函数的原型如下：

```
DWORD SetFilePointer
(
HANDLE hFile,                    //文件句柄
LONG lpDistanceToMove,           //指针移动的字节数
LPLONG lpDistanceToMoveHigh,     //指向指针移动距离高 32 位数的指针
DWORD dwMoveMethod               //指针移动参考位置标识，详见表 9-3
);
```

表 9-3 参考位置标识及其说明

标 识	说 明
FILE_BEGIN	参考位置为文件的开始
FILE_CURRENT	参考位置为当前文件指针位置
FILE_END	参考位置为文件的结尾

9.1.3 文件操作应用实例

【例 9-1】

本例创建一个具有打开文件功能的对话框，用户运行该程序后可以打开文件，并进行文件的编辑与保存。

本例的源代码如下：

```
# include < windows.h >
# include < commdlg.h >
# include < string.h >
# include < stdio.h >
# include "9-1.h"
int APIENTRY WinMain(
        HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow)
{
    MSG msg;
    if(! InitApplication(hInstance))
      return(FALSE);
    if(! InitInstance(hInstance,nCmdShow))
      return(FALSE);
    while(GetMessage(&msg,NULL,0,0))
    {
```

```
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  return(msg.wParam) ;
}

BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASSEX wcexFileDlg;
    wcexFileDlg.cbSize = sizeof(WNDCLASSEX);
    wcexFileDlg.style = 0;
    wcexFileDlg.lpfnWndProc = (WNDPROC)MainWndProc;
    wcexFileDlg.cbClsExtra = 0;
    wcexFileDlg.cbWndExtra = 0;
    wcexFileDlg.hInstance = hInstance;
    wcexFileDlg.hIcon = LoadIcon(hInstance,"FileDlgIcon");
    wcexFileDlg.hIconSm = LoadIcon(hInstance,"FileDlgSmallIcon");
    wcexFileDlg.hCursor = LoadCursor(NULL,IDC_ARROW);
    wcexFileDlg.hbrBackground = GetStockObject(WHITE_BRUSH);
    wcexFileDlg.lpszMenuName = "FileDlgMenu";
    wcexFileDlg.lpszClassName = "FileDlgWClass";

    return(RegisterClassEx(&wcexFileDlg));
}

BOOL InitInstance(HINSTANCE hInstance,int nCmdShow)
{
      HWND hWnd;
      hInst = hInstance;
      hWnd = CreateWindow
          (
          "FileDlgWClass",
          "使用文件通用对话框的示范程序",
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          NULL,
          NULL,
          hInstance,
```

```
            NULL
            );
            if(! hWnd)
                return(FALSE);
            ShowWindow(hWnd,nCmdShow);
            UpdateWindow(hWnd);
            return(TRUE);
}

LRESULT APIENTRY MainWndProc
        (
        HWND hWnd,
        UINT message,
        WPARAM wParam,
        LPARAM lParam
        )
{
    static HWND hWndEdit;
    switch(message)
    {
    case WM_CREATE:
         hWndEdit = CreateWindow
                  (
                  "EDIT",
                  NULL,
                  WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | ES_LEFT | ES_MULTI
                           LINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL,
                  0,
                  0,
                  0,
                  0,
                  hWnd,
                  (HMENU)ID_EDITCHILD,
                  (HINSTANCE)GetWindowLong(hWnd,GWL_HINSTANCE),
                  NULL);
        break;
    case WM_SIZE:
        MoveWindow
                  (
                  hWndEdit,
                  0,
                  0,
```

```
                    LOWORD(lParam),
                    HIWORD(lParam),
                    TRUE);
        break;
    case WM_COMMAND:
        switch (LOWORD(wParam))
                {
                  case IDM_FILEOPEN:
                      OpenTextFile(hWnd,hWndEdit);
                      break;
                  case IDM_FILESAVE:
                      SaveTextFile(hWnd,hWndEdit);
                      break;
                  case IDM_EXIT:
                      SendMessage(hWnd,WM_DESTROY,0L,0L);
                      break;
                  case IDM_ABOUT :
                      DialogBox(hInst,"AboutBox",hWnd,(DLGPROC)About);
                      break ;
                  default :
                      return(DefWindowProc(hWnd,message,wParam,lParam));
                }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return(DefWindowProc(hWnd,message,wParam,lParam));
    }
    return(0);
}

//      函数：OpenTextFile ( HWND , HWND )
//      用途：调用通用对话框函数，显示“打开”对话框，检取用户选择的
//          文本文件，然后打开该文本文件到编辑控件中

void OpenTextFile(HWND hWnd,HWND hWndEdit)
{
    OPENFILENAME OpenFileName;
    char szDirName[MAX_PATH] = "";
    char szFile[MAX_PATH] = "\0";
    char szFileTitle[MAX_PATH] = "\0";
```

```
// OPENFILENAME 结构的过滤器规格说明
    char szFilter[] = {"文本文件(*.txt)\0*.txt\0C 源程序文件(*.c)\0*.c\0 所
                       有文件\0*.*0"};
    OpenFileName.lStructSize = sizeof(OPENFILENAME);
    OpenFileName.hWndOwner = hWnd;
    OpenFileName.hInstance = (HANDLE)hInst;
    OpenFileName.lpstrFilter = szFilter;
    OpenFileName.lpstrCustomFilter = (LPTSTR)NULL;
    OpenFileName.nMaxCustFilter = 0L;
    OpenFileName.nFilterIndex = 1L;
    OpenFileName.lpstrFile = szFile;
    OpenFileName.nMaxFile = sizeof(szFile);
    OpenFileName.lpstrFileTitle = szFileTitle;
    OpenFileName.nMaxFileTitle = sizeof(szFileTitle);
    OpenFileName.lpstrInitialDir = NULL;
    OpenFileName.lpstrTitle = "请选取一个文本文件";
    OpenFileName.nFileOffset = 0;
    OpenFileName.nFileExtension = 0;
    OpenFileName.lpstrDefExt = "*.txt";
    OpenFileName.lCustData = 0;
    OpenFileName.Flags = OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST|OFN_HIDEREADONLY;
    if(GetOpenFileName(&OpenFileName))
    {
        HANDLE hFile;
        DWORD dwFileSize,dwBytesRead;
        char * lpBufPtr;
        // 打开指定的文件
        hFile = CreateFile
              (
              OpenFileName.lpstrFile,
              GENERIC_READ,
              FILE_SHARE_READ,
              NULL,
              OPEN_EXISTING,
              FILE_ATTRIBUTE_NORMAL,
              (HANDLE)NULL
              );
         if(hFile! = INVALID HANDLE VALUE)
         {
            //检取文件字节大小
            dwFileSize = GetFileSize(hFile,NULL);
```

```
            if(dwFileSize! =0xFFFFFFFF)
            {
            // 分配用来读入文件的缓冲区
                lpBufPtr=(char *)malloc(dwFileSize);
                if(lpBufPtr! =NULL)
                    {
                    // 把文件内容读入缓冲区
                      ReadFile(hFile,(LPVOID)lpBufPtr,dwFileSize,&dwBytesRead,
                      NULL);
                      if (dwBytesRead! =0)
                      {
                            //把文件内容装入多行编辑控件中
                             SendMessage(hWndEdit,WM_SETTEXT,0,(LPARAM)lpBufP-
                             tr);
                            SetWindowText(hWnd,OpenFileName.lpstrFile);
                      }
                        else
                             MessageBox (hWnd,"读入字节数为零。",NULL,MB_OK|MB
                                    ICONEXCLAMATION);
                       free(lpBufPtr);
                    }
                    else
                         MessageBox(NULL,"内存分配失败!",NULL,MB_OK|MB ICONEXCLA-
                                MATION);
                    }
                else
                    MessageBox(NULL,"检取文件大小失败",NULL,MB_OK|MB_ICONEXCLA
                            MATION);
                  CloseHandle(hFile);
            }
            else
              MessageBox(hWnd,"打开文件失败",NULL,MB_OK|MB_ICONEXCLAMATION);
    }
}
//    函数:saveTextFile (HWND, HWND)
//    用途:调用通用对话框函数,显示"另存为"对话框,由用户选择文本名,然后把编辑控件中的
//        内容保存到该文件中
void SaveTextFile(HWND hWnd, HWND hWndEdit )
{
OPENFILENAME OpenFileName;
char szDirName[MAX_PATH]="";
char szFile[MAX_PATH]="\0";
```

```
char szFileTitle[MAX_PATH] = "\0";
char szFilter[] = {"All Files\0*.*\0"};
OpenFileName.lStructSize = sizeof(OPENFILENAME);
OpenFileName.hWndOwner = hWnd;
OpenFileName.hInstance = (HANDLE)hInst;
OpenFileName.lpstrFilter = szFilter;
OpenFileName.lpstrCustomFilter = (LPTSTR)NULL;
OpenFileName.nMaxCustFilter = 0L;
OpenFileName.nFilterIndex = 1L;
OpenFileName.lpstrFile = szFile;
OpenFileName.nMaxFile = sizeof(szFile);
OpenFileName.lpstrFileTitle = szFileTitle;
OpenFileName.nMaxFileTitle = sizeof(szFileTitle);
OpenFileName.lpstrInitialDir = NULL;
OpenFileName.lpstrTitle = "另存为";
OpenFileName.nFileOffset = 0;
OpenFileName.nFileExtension = 0;
OpenFileName.lpstrDefExt = NULL;
OpenFileName.lCustData = 0;
OpenFileName.Flags = OFN_OVERWRITEPROMPT;
if (GetSaveFileName (&OpenFileName))
{       HANDLE hFile;
        DWORD dwFileSize,dwBytesWrite;
        char  * lpBufPtr;
  //创建文件(如果已经有了同名文件,则覆盖该文件)
        hFile = CreateFile
          (
          OpenFileName.lpstrFile,
          GENERIC_READ|GENERIC_WRITE,
          0,
          NULL,
          CREATE_ALWAYS,
          FILE_ATTRIBUTE_NORMAL,
          (HANDLE)NULL);
if(hFile! = INVALID_HANDLE_VALUE)
{
  //检取编辑控件内容的字节大小
     dwFileSize = SendMessage(hWndEdit,WM_GETTEXTLENGTH,0,0);
     if(dwFileSize! = 0)
       {
            // 分配文件缓冲区
```

```
            lpBufPtr = (char *)malloc(dwFileSize);
            if(dwFileSize! = 0)
              {
              // 把编辑控件中的内容装到文件缓冲区
                 SendMessage(hWndEdit,WM_GETTEXT,dwFileSize,(LPARAM)lpBufPtr);
              //把文件缓冲区中的数据存写到文件中
                 WriteFile (hFile, (LPVOID) lpBufPtr, dwFileSize, &dwBytesWrite,
                           NULL);
                 SetWindowText(hWnd,OpenFileName.lpstrFile);
                 free(lpBufPtr);
                   }
                 else
                   MessageBox (NULL,"内存分配失败!", NULL, MB_OK | MB_ICONEXCLA-
                   MATION);
                 }
               else
                 MessageBox(NULL,"编辑控件中内容为空!",NULL,MB_OK|MB_ICONEXCLA
                           MATION);
                 CloseHandle(hFile);
              }
          else
              MessageBox(hWnd,"创建文件失败!",NULL,MB_OK|MB_ICONEXCLAMATION);
          }
      }
      BOOL APIENTRY About(HWND hDlg,UINT message,WPARAM wParam,LPARAM lparam)
      {
          switch (message)
          {
              case WM_INITDIALOG:
                  return (TRUE );
              case WM_COMMAND:
                  if(LOWORD(wParam) == IDOK||LOWORD(wParam) == IDCANCEL)
                  {
                      EndDialog(hDlg,TRUE);
                      return (TRUE);
                  }
                  break;
        }
        return(FALSE);
    }
```

本例的头文件代码如下：

```
//  菜单命令
```

```
//          "文件"菜单
# define IDM_FILEOPEN                100
# define IDM_FILESAVE                101
# define IDM_EXIT                    102
//          "帮助"菜单
# define IDM_ABOUT                   110
//编辑控件标号
# define ID_EDITCHILD                200
# define IDC_STATIC                  -1
//全局变量
HINSTANCE hInst;
//函数原型
//过程
LRESULT APIENTRY MainWndProc(HWND,UINT,WPARAM,LPARAM);
BOOL APIENTRY About(HWND,UINT,WPARAM,LPARAM);
//函数
BOOL    InitApplication(HINSTANCE);
BOOL    InitInstance(HINSTANCE,int);
void    OpenTextFile(HWND hWnd,HWND hWndEdit);
void    SaveTextFile(HWND hWnd,HWND hWndEdit);
```

本例的资源文件代码如下：

```
# include "windows.h"
# include "9-1.h"
// Menu
FILEDLGMENU MENU PRELOAD DISCARDABLE

BEGIN
        POPUP"文件(&F)"
        BEGIN
                MENUITEM "打开(&O)…"              IDM_FILEOPEN
                MENUITEM "另存为(&A)…"            IDM_FILESAVE
                MENUITEM  SEPARATOR
                MENUITEM "退出(&X)"           IDM_EXIT
        END
        POPUP "帮助"
        BEGIN
                MENUITEM "关于(&A)…",         IDM_ABOUT
        END
  END
//Diaalog
ABOUTBOX DIALOG DISCARDABLE 20, 25, 124 , 77
```

```
STYLE DS-MODALFRAME | WS-POPUP | WS-VISIBLE | WS-CAPTION| WS-SYSMENU CAPTION"关于文
件通用控件使用示范程序"
FONT 14,"隶书-GB2312"
BEGIN
    PUSHBUTTON "确定",IDOK,43 ,51 , 40, 14
    CTEXT "文件通用控件使用示范程序",IDC-STATIC , 22, 9,92,8
    CTEXT "VC ++ 示例",IDC STATIC ,30,24,76,10,NOT WS-GROUP
    CTEXT "一九九九年九月",IDC-STATIC,36, 37 ,65, 8 ,NOT WS-GROUP
    ICON "FILEDLGICON",IDC-STATIC,11,23,18,20
    CONTROL " " ,IDC-STATIC,"Static",SS-BLACKFRAME,40,34,56,1
END
```

运行结果如图 9-1 所示。

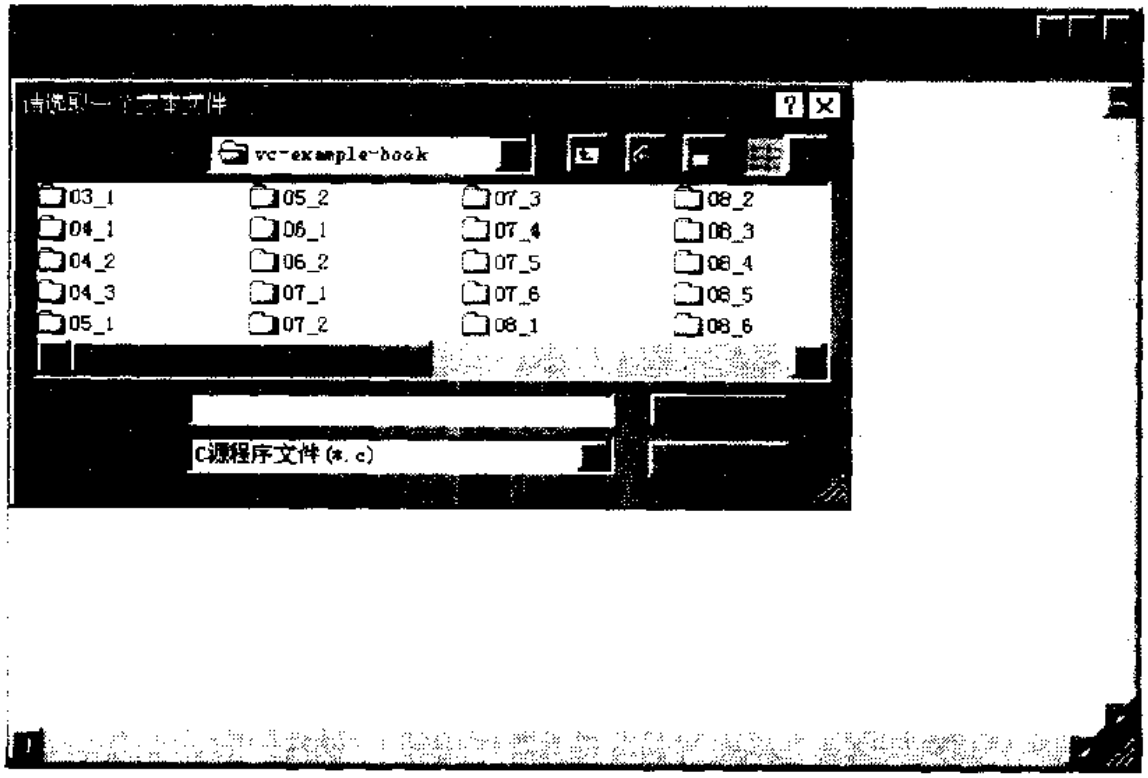


图 9-1　文件操作实例程序运行结果

9.2　文件中剪贴板的操作

剪贴板在文件的操作中是非常有用的，通过剪贴板的操作，可以对某一部分内容在同一个文件的不同位置进行复制、移动，或是在不同文件之间进行某一特定内容的复制。

使用剪贴板首先应考虑的问题是其支持的数据格式。在使用剪贴板之前，应用程序需将准备操作剪贴板的数据初始化为系统预定义的数据格式。表 9-4 所示为系统预定义的剪贴板数据格式及其说明。

表 9-4 系统预定义剪贴板数据格式及其说明

数据格式	说 明
CF_TEXT	NULL 结尾的文本
CF_OEMTEXT	NULL 结尾的 OEM 文本
CF_METAFILEPICT	图元文件
CF_BITMAP	与设备有关的位图
CF_DIB	与设备无关的位图
CF_ENHMETAFILE	增强型元文件
CF_SYLK	SYLK 数据交换格式
CF_DIF	DIF 数据交换格式
CF_TIFF	TIFF 数据交换格式
CF_PALETTE	与 CF_DIB 相关的调色板格式

9.2.1 向剪贴板发送文本

应用程序向剪贴板发送文本的操作过程可分为如下 5 个步骤：

1. 拷贝文本到全局内存

应用程序首先调用 GlobalAlloc 函数为文本分配全局存储空间，如果分配全局内存成功，函数返回 TRUE，否则返回 FALSE，该函数的原型为：

```
HGLOBAL GlobalAlloc
(
UINT uFlags,     //函数分配内存形式标识。常用标识及其说明见表 9-5
DWORD dwBytes    //分配字节数
)
```

表 9-5 GlobalAlloc 函数常用标识及其说明

分配标识	说 明
GMEM_DISCARDABLE	分配可丢弃内存
GMEM_MOVEABLE	分配可移动内存
GMEM_FIXED	分配固定内存
GMEM_ZEROINIT	分配内存时将内存内容初始化为 0
GMEM_DDESHARE	分配共享内存
GHND	GMEM_MOVEABLE 和 GMEM_ZEROINIT 的组合
GPTR	GMEM_FIXED 和 GMEM_ZEROINIT 的组合

应用程序为文本分配全局内存后，即可调用 GlobalLock 函数锁定分配的内存块，并返回指向该内存块的指针。该函数的原型为：

```
LPVOID GlobalLock(HGLOBAL hMem) //hMem 为内存块地址
```

应用程序使用 GlobalLock 函数返回的指针将指定文本拷贝到全局内存中。

把数据拷贝到内存以后，应用程序应及时调用函数 GlobalUnLock 解锁内存句柄，如果解锁成功，则返回 TRUE，否则返回 FALSE。

该函数的原型为：

```
BOOL GlobalUnLock (HGLOBAL hMem); //hMem 为内存句柄
```

2. 打开剪贴板

该步骤由应用程序调用 OpenClipboard 函数完成，该函数的原型为：

```
BOOL OpenClipboard (HWND hwnd) //hwnd 为打开剪贴板的窗口句柄
```

在任何时候，系统只允许一个程序打开剪贴板。如果存在另一个程序未及时关闭剪贴板，则 OpenClipboard 函数将返回 FALSE。

3. 删除剪贴板中的内容

该步骤由应用程序调用函数 EmptyClipboard 完成，该函数清除剪贴板中的所有句柄，其原型为：

```
BOOL EmptyClipboard(VOID);
```

4. 向剪贴板传送文本全局内存句柄

该步骤由应用程序调用函数 SetClipboardData 完成，该函数初始化剪贴板数据格式并将内存句柄传送给剪贴板，其原型为：

```
HANDLE SetClipboardData
(
UINT uFormat,     //数据格式标识
HANDLE hMem       //数据句柄
);
```

应用程序一旦将文本内存句柄传递给剪贴板，则该句柄属于剪贴板，应用程序不能再对其进行操作。

5. 关闭剪贴板

该步骤由应用程序调用函数 CloseClipboard 完成，该函数的原型为：

```
BOOL CloseClipboard(VOID)
```

下面的程序段是应用程序向剪贴板发送文本的一般过程：

```
HANDLE hText;
LPTSTR lpString,lpText;
...
    case IDM_COPY:
        //分配全局内存
        if(! (hText - GlobalAlloc(GHND,Sizeof(lpSring)))
        {
          MessageBox(hwnd,"全局内存分配失败!","提示",MB_OK);
          break;
        }
        lpText = GlobalLock(hText);              //锁定文本内存句柄并返回文本指针
        lstrcpy(lpText,lpString),                //拷贝文本
        GlobalUnlock(hText);                     //解锁文本内存句柄
        if(! OpenClipboard(hwnd))
            {
            MessageBox(hwnd,"剪贴板打开失败!","提示",MB_OK);
```

```
        break;
        }
    EmptyClipboard( );                          //清除剪贴板
    SetClipboardData(CF-TEXT,hText);            //设置剪贴板文本
    CloseClipboard();                           //关闭剪贴板
    hText = NULL
    break;
```

该程序段的最后,应用程序显式地设置文本内存句柄为 NULL,以避免应用程序再通过该句柄执行其他操作。

9.2.2 获取剪贴板中的文本

应用程序从剪贴板上获取文本的操作过程可分为如下 4 个步骤:

1. 打开剪贴板

应用程序首先调用函数 OpenClipboard 打开剪贴板。

2. 检查剪贴板数据格式

该步骤由函数 IsClipboardFormatAvailable 完成。该函数的原型为:

```
BOOL IsClipboardFormatAvailable(UINT uformat);
```

当剪贴板中包含指定数据格式时,函数返回 TRUE,否则返回 FALSE。

3. 获取剪贴板文本

该步骤由函数 CetClipboardData 完成,该函数的原型为:

```
HANDLE GetClipboardData(UINT uformat); //uformat 为数据格式标识
```

函数调用成功则返回剪贴板上文本的内存句柄,否则返回 NULL。

应用程序成功获取剪贴板文本内存句柄后,即可调用函数 GlobalLock 返回指向文本的指针。由于关闭剪贴板后该指针自动失效,应用程序应及时保存文本的副本。

获取的内存句柄属于剪贴板,而不属于应用程序,因此,应用程序读取剪贴板中的数据后,只能调用函数 GlobalUnlock 解锁该内存句柄,而无法将其释放。

4. 关闭剪贴板

最后,应用程序调用函数 CloseClipboard 关闭剪贴板。

下面的程序段是应用程序获取剪贴板文本的一般形式:

```
HANDLE hText;
LPTSTR lpString,lpText;
…
    case IDM_PASTE:
        //检查剪贴板的数据格式
        if (! IsClipboardFormatAvailable(CF-TEXT))
            {
                MessageBox(hwnd,"剪贴板上无文本数据","提示",MB_OK)
                Break;
            }
        //打开剪贴板
```

```
        if(! OpenClipboard(hwnd))
        {
            MessageBox(hwnd,"剪贴板打开失败","提示",MB_OK)
            Break;
        }
        //获取剪贴板文本内存句柄
        if(! (hText = GetClipboardData(CF_TEXT)))
        {
            MessageBox(hwnd,"无法读取剪贴板数据","提示",MB_OK);
            CloseClipboard();
            Break;
        }
        lpText = GlobalLock(hText);                    //锁定文本内存名柄并返回文本指针
        //复制剪贴板文本内容
        lpString = (LPTSTR)malloc(GlobalSize(hText));
        lpstrcpy(lpString,lpText);
        GlobalUnlock(hText);                           //解锁文本内存句柄
        CloseClipboard();                              //关闭剪贴板
...
        break;
...
```

9.2.3　剪贴板应用举例

【例 9-2】

本例要求创建一个对话框，具有剪切、复制、粘贴、取消动作及退出等功能。

本例的源程序代码如下：

```
# include <windows.h>
# include <commctrl.h>
# include <richedit.h>
# include <string.h>
# include <stdio.h>
# include <stdlib.h>
# include "9.2.h"

HINSTANCE    hInst;
HWND         hWndMain;
HWND         hWndRichEdit;
CHAR         szRtfFileName[MAX_PATH];
HCURSOR      hCursorWait;
HCURSOR      hCursorNormal;
HANDLE       hRTFLib;
```

```
int APIENTRY WinMain
    (
    HINSTANCE  hInstance,
    HINSTANCE  hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
    )
{
    MSG msg;
    hRTFLib = LoadLibrary("RICHED32.DLL");

    if (! InitApplication(hInstance))
      return (FALSE);

    if (! InitInstance(hInstance, nCmdShow))
      return (FALSE);

    hCursorWait = LoadCursor(NULL, IDC_WAIT);
    while (GetMessage(&msg, NULL, 0, 0))
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
    return (msg.wParam);
}

BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASSEX wcexStatusBar;
    wcexStatusBar.cbSize = sizeof(WNDCLASSEX);
    wcexStatusBar.style = 0;
    wcexStatusBar.lpfnWndProc = (WNDPROC)MainWndProc;
    wcexStatusBar.cbClsExtra = 0;
    wcexStatusBar.cbWndExtra = 0;
    wcexStatusBar.hInstance = hInstance;
    wcexStatusBar.hIcon = LoadIcon(hInstance, "RichEditIcon");
    wcexStatusBar.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcexStatusBar.hbrBackground = GetStockObject(WHITE_BRUSH);
    wcexStatusBar.lpszMenuName = "RichEditMenu";
    wcexStatusBar.lpszClassName = "RichEditWClass";
    wcexStatusBar.hIconSm = LoadIcon(hInstance,"RichEditSmallIcon");
    return (RegisterClassEx(&wcexStatusBar));
```

```
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    hInst = hInstance;
    hWnd = CreateWindow(
    "RichEditWClass",
    "剪贴板操作示范程序",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
    NULL,NULL,hInstance,NULL);
    if(! hWnd)
    return(FALSE);
    hWndMain = hWnd;
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
    return(TRUE);
}

LRESULT APIENTRY MainWndProc ( HWND hWnd, UINT message, WPARAM wParam, LPARAM
                               lParam)
{
    static TCHAR szBuf[128];
    static TCHAR szFontString[1024];
    UINT uItem;

    switch (message)
    {

        case WM_CREATE:
            {
            // 创建多格式文字编辑控件
            hWndRichEdit = CreateWindowEx
                (
                WS_EX_CLIENTEDGE, // 使多格式文字编辑控件有一个客户型边界(凹形)
                "RICHEDIT",       // 多格式文字编辑控件的窗口类名称
                NULL,             // 多格式文字编辑控件中的初始文字
               WS_CHILD|WS_VISIBLE|ES_MULTILINE|ES_SAVESEL|
                            WS_HSCROLL|WS_VSCROLL,//窗口风格
                0, 0, 0, 0,       // 位置和尺寸待定
```

```
            hWnd,              // 父窗口句柄
            (HMENU)ID_RICHEDIT, // 控件标识号
            hInst,             //实例句柄
            NULL
            );
        lstrcpy(szRtfFileName,"粘贴示例");
        SetWindowText(hWnd, szRtfFileName);
        }
        break;
    case WM_SIZE:
        MoveWindow
            (
            hWndRichEdit,
            0,
            0,
            LOWORD(lParam),
            HIWORD(lParam),
            TRUE
            );
        break;
case WM_INITMENUPOPUP:
    if((BOOL)HIWORD(lParam))    // 如果是系统菜单,则不做检查
        break;
    //检查是否可以进行撤销操作,并由此设置菜单项的状态
    if(SendMessage(hWndRichEdit,EM_CANUNDO,0,0))
        EnableMenuItem(GetMenu(hWndMain),IDM_EDITUNDO,MF_BYCOMMAND|MF_EN-
                       ABLED);
    else
        EnableMenuItem(GetMenu(hWndMain),
                       IDM_EDITUNDO,MF_BYCOMMAND|MF_GRAYED);
    //检查是否可以进行粘贴操作,并由此设置菜单项的状态
    if(SendMessage(hWndRichEdit,EM_CANPASTE,0,0))
        EnableMenuItem(GetMenu(hWndMain),
                       IDM_EDITPASTE,MF_BYCOMMAND|MF_ENABLED);
    else
        EnableMenuItem(GetMenu(hWndMain),
                       IDM_EDITPASTE,MF_BYCOMMAND|MF_GRAYED);
    break;
case WM_COMMAND:
    uItem = LOWORD(wParam);
```

```
            switch (uItem)
            {
              case IDM_EDITUNDO:
                  SendMessage(hWndRichEdit, WM_UNDO, 0, 0);
                  break;
              case IDM_EDITCUT:
                  SendMessage(hWndRichEdit, WM_CUT, 0, 0);
                  break;
              case IDM_EDITCOPY:
                  SendMessage(hWndRichEdit, WM_COPY, 0, 0);
                  break;
              case IDM_EDITPASTE:
                  SendMessage(hWndRichEdit, WM_PASTE, 0, 0);
                  break;
              case IDM_EDITDELETE:
                  SendMessage(hWndRichEdit, WM_CLEAR, 0, 0);
                  break;
              case IDM_EDITSELECTALL:
                  SendMessage(hWndRichEdit, EM_SETSEL, 0, -1);
                  break;
              case IDM_EXIT:
                  SendMessage(hWnd, WM_CLOSE, 0, 0L);
                  return 0;
              default:
                  return(DefWindowProc(hWnd, message, wParam, lParam));
            }
            break;
            if (hRTFLib);
            {
                FreeLibrary(hRTFLib);
                hRTFLib = NULL;
            }
            PostQuitMessage(0);
            break;
        default:
            return(DefWindowProc(hWnd,message,wParam,lParam));
    }
    return(0);
}
```

本例的头文件代码如下：

```
//        "编辑"菜单
```

```
#define IDM_EDITUNDO        200
#define IDM_EDITCUT         201
#define IDM_EDITCOPY        202
#define IDM_EDITPASTE       203
#define IDM_EDITDELETE      204
#define IDM_EDITSELECTALL   205
#define IDM_EXIT            100
#define ID_RICHEDIT         1000
#define IDC_RTFFILENAME     1009
//  函数原型说明
LRESULT APIENTRY MainWndProc(HWND, UINT, WPARAM, LPARAM);
//其他函数
BOOL  InitApplication(HINSTANCE);
BOOL  InitInstance(HINSTANCE,int);
VOID  UpdateCharFormatState(HWND,HWND);
```

本例的资源程序代码如下：

```
#include "windows.h"
#include "winuser.h"
#include "9-2.h"

//Menu

RICHEDITMENU MENU DISCARDABLE
BEGIN
  POPUP "编辑(&E)"
    BEGIN
    MENUITEM "剪切(&T)",                IDM_EDITCUT
    MENUITEM "复制(&C)",                IDM_EDITCOPY
    MENUITEM "粘贴(&P)",                IDM_EDITPASTE
    MENUITEM "删除(&D)",                IDM_EDITDELETE
    MENUITEM SEPARATOR
    MENUITEM "撤销(&U)",                IDM_EDITUNDO,GRAYED
    MENUITEM SEPARATOR
    MENUITEM "全部选定(&A)",            IDM_EDITSELECTALL
    MENUITEM SEPARATOR
    MENUITEM "退出(&X)",                IDM_EXIT
    END
END
```

本例的运行结果如图 9-2 所示。

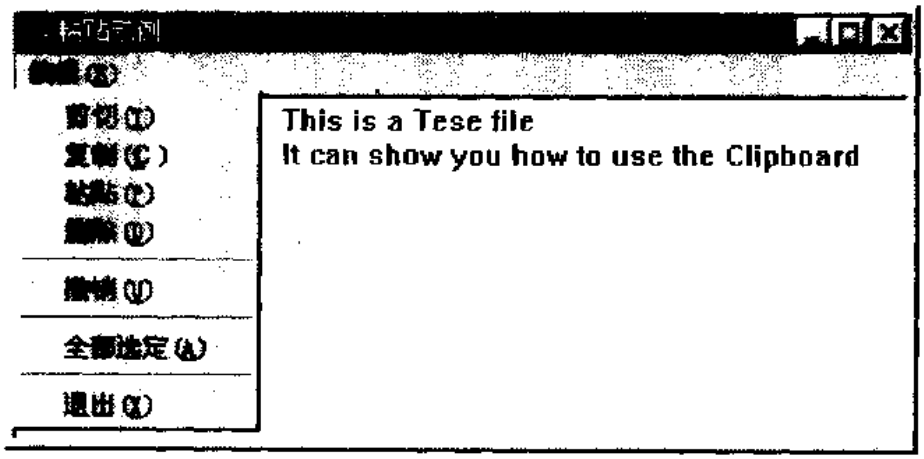


图 9-2　【例 9-2】的运行结果

习　题

9-1　文件操作的特点是什么?

9-2　如何进行剪贴板的操作?

9-3　创建一个包含“文件”菜单的窗口,“文件”菜单中包含“打开”、“新建”、“保存”、“另存为”和“退出”等选项。要求单击“打开”选项时,弹出文件列表框,选择文件并在文本编辑框中显示该文件的内容;单击“新建”选项,可以创建新文件;单击“保存”选项,可以保存文件;单击“另存为”选项,可以对已经打开的文件更改文件名再保存。

9-4　在习题 9-3 中增加“编辑”菜单,其中有“复制”和“粘贴”选项,可以把选中的部分内容输出到剪贴板,然后再拷贝到指定的位置。

第 3 篇

使用 MFC 进行可视化编程

在掌握了应用 API 函数进行可视化编程的方法之后，本篇将进一步学习利用 MFC 进行可视化编程。MFC 是 Visual C++ 提供的用来编写 Windows 应用程序的 C++ 类集，其中封装了大部分 Windows API 函数及 Windows 控件。MFC 编程的一个特点是系统提供了一个“向导”(Wizard)，利用“向导”，可形成应用程序的基本框架。程序员只需在形成的框架的基础上进一步添加必要的实现代码，即可编写出自己的 Windows 应用程序。

本篇内容包括：

第 10 章　MFC 基础知识

第 11 章　编辑框类及其应用

第 12 章　菜单类及其应用

第 13 章　滚动条类及其应用

第 14 章　按钮类与列表框类及其应用

第 15 章　工具条类及其应用

第 16 章　文档-视图结构及其应用

第 10 章

MFC 基础知识

在 VC++ 的编程中，从前面的章节内容，读者可能已经看到，利用 Windows API 函数进行编程时，大量的代码需要用户自己编写，用户编程的工作量较大。从本章开始，我们将讨论 VC++ 的另一种编程方法，利用 MFC 和向导(Wizard)来编写 Windows 应用程序，即首先使用 Class Wizard 来生成 Windows 应用程序的基本框架，然后用 ClassWizard 来建立应用程序的类、消息处理、数据处理函数或定义控件的属性、事件和方法，最后把各应用程序所要求的功能添加到类中。

10.1 MFC 概述

MFC 是用来编写 Windows 应用程序的 C++ 类集，该类集以层次结构组织起来，其中封装了大部分 Windows API 函数和 Windows 控件，它所包含的功能涉及到整个 Windows 操作系统。MFC 不仅为用户提供了 Windows 图形环境下应用程序的框架，而且还提供了创建应用程序的组件。使用 MFC 类库和 Visual C++ 提供的高度可视的应用程序开发工具，可使应用程序开发变得更简单，极大地缩短开发周期，提高代码的可靠性和可重用性。

代码重用是 C++ 长期寻求的目标。对于 C++ 程序员而言，重用通常是指从先前已有的基类派生新的 C++ 类的技术。MFC 正好提供了大量的基类供程序员根据不同的应用环境进行扩充。

MFC 提供的类库对程序设计的高度抽象，使得程序员的主要精力不用放在程序设计的细节实现上，而放在程序的功能拓展上面，它同时允许在编程过程中自定义和扩展应用程序中的类，MFC 同时还允许调用 Windows API，从而使应用程序能以最小的规模实现最丰富的功能，而且能提供高效率的运行代码。

MFC 是 C++ 语言中的一个安全子集，它大大简化了使用 C++ 开发基于 Windows 的应用程序的工作。MFC 精心设计的类库结构以一种直观的软件包的形式把进行 Windows 应用开发这一过程所需的各种程序模块有机地组织起来，经验丰富的 C++ 开发人员可以使用 MFC 实现 C++ 中的高级功能。

MFC 被设计成可移植于众多的平台，允许其应用程序适用于多种不同平台。对于多种编译器，MFC 也是可以移植的，而且有许多软件开发公司已经把它作为一种基于 Windows开发标准的应用程序构架。

MFC 之所以得名为微软基础类库，是因为它代表着正在不断带给 Windows 操作系统开发人员以最好支持的基本类结构。它所包含的类分层结构和功能具有可伸缩性，这种

可伸缩性使得 MFC 始终能跟上软件发展的潮流，而且在功能扩展的同时，MFC 还很好地保持了程序的向下兼容性。

1992 年 4 月，第一个 MFC(Microsoft Foundation Class)版本即 MFC 1.0，伴随着 Microsoft C/C++ 7.0 版一起发布。当时的 MFC 1.0 中主要包括两种类型的类：用于应用程序中非图形部分的类和用于应用程序的图形用户界面(Graphics Device Interface，缩写为 GDI)功能的 Windows 相关类。

1993 年 2 月，MFC 2.0 伴随着 Visual C++ 1.0 版一起发布。其被扩展后的核心功能在原有的基础上又增添了一些新的构造类，这些构造类有助于组织和构造应用程序，以及对 Windows 应用程序中部分界面元素的高层抽象，这些高层抽象有助于程序员优化界面，使得应用程序更易操作。

1993 年 12 月，MFC 2.5 与 Visual C++ 1.5 一起发布。这一版本的 MFC 添加了与数据库进行连接的 ODBC 数据库类，ODBC 数据库类允许应用程序访问具有 ODBC 驱动程序的数据库中的数据，程序员可以通过 ODBC(Open Database Connection，开放数据库互连，简称 ODBC)对某些数据源进行存取。这一版本的 MFC 还全面支持 OLE 的众多性能，不过这一版本的 MFC 是基于 16 位的应用程序而开发的。

1994 年 9 月，MFC 3.0 作为 Visual C++ 2.0 的一部分公开发布。这个版本的 MFC 增添了对开发 32 位应用程序的支持，具有更丰富的用户界面风格，以及对 Win32 API 和 OLEControl 扩展的更多支持。MFC 3.0 将 MFC 的影响扩展到大多数基于 Win32 的应用程序的核心底层结构。随后的两个版本中又添加了对 Windows 公共控件和 Sockets 等的支持。

1995 年 10 月推出的 MFC 4.0 中包含了在 Windows 95 和 Windows NT 操作系统中大多数新的 Windows 公共控件，它能进一步支持 OLE 的扩展功能，使得开发人员可以建立、使用并且可以和其他开发人员共享 OLE 控件。MFC 4.0 除了继续扩大对 ODBC 的支持外，还提供了一种新的数据存取对象(Data Access Object，简称 DAO)类。通过 DAO 类，程序员可以直接存取 Miscrosoft Jet 数据库引擎，它是一种和 Microsoft Access for Windows 98 和 Miscrosoft Visual Basic 中的引擎完全相同的引擎。DAO 类包含了 Jet 数据库引擎的 OLE COM 接口，开发人员不必亲自编写 SQL 程序就可以实现对数据库的操作。当使用 DAO 类时，程序员可以有计划地访问和操纵数据库内的数据，并管理数据库、数据库对象及数据库结构。MFC 中还引进了线程同步对象的概念，为了管理线程的同步操作，MFC 提供了一个新的基类——CSyncObject，以及代表公用同步技术的几个派生对象。MFC 还提供了对 Windows 的消息应用程序接口(Message Application Programming Interface，简称 MAPI)的支持，通过 MFC 的这一扩展，程序员可以很容易开发出用于建立、处理、转送以及存储邮件消息的应用程序。MFC 还进一步增强了对 Windows Sockets 的支持，通过 MFC 可以较容易地实现在 Windows 操作系统环境下的网络通信程序的编写。

目前使用的 MFC 4.21 增强了对数据库应用程序的支持。可以使用 ODBC 类和高性能的 ODBC 驱动程序来访问提供 ODBC 支持的数据库中的数据；可以通过数据访问对象(DAO)类通过编程语言来访问和操纵数据库中的数据并管理数据库、数据库对象与结构。同时 MFC 4.21 对 Internet 也提供了强有力的支持。

10.2　MFC 中类的组织结构及主要的类的简介

10.2.1　MFC 中类的组织结构

目前的 MFC 版本中包含了 100 多个类，不同的类实现不同的功能，类之间既有区别又有联系。MFC 同时还是一个应用程序框架，它帮助定义应用程序的结构，以及为应用程序处理许多杂务，事实上，MFC 封装了一个程序操作的每一方面。在 MFC 程序中，程序员很少需要直接调用 Windows API 函数，而是通过定义 MFC 类的对象并调用对象的成员函数来实现相应的功能。

MFC 类库中类是以层次结构的方式组织起来的，几乎每个子层次结构都与一具体的 Windows 实体相对应，一些主要的接口类管理了难以掌握的 Windows 接口。这些接口包括：窗口类、GDI 类、对象链接和嵌入类(OLE)、文件类、对象 I/O 类、异常处理类以及集合类等。

MFC 库中的类按层次关系划分可分为如下若干类：

(1) 根类：CObject

(2) 应用程序体系结构类

- 应用程序和线程支持类
- 命令相关类
- 文档类
- 视类(体系结构)
- 框架窗口类(体系结构)
- 文档模板类

(3) 窗口、对话框和控件类

- 框架窗口类(窗口)
- 视类(窗口)
- 对话框类
- 控件类
- 控件条类

(4) 绘图和打印类

- 输出(设备相关)类
- 绘图工具类

(5) 简单数据类型类

(6) 数组、列表和映像类

- 数组类
- 列表类
- 映像类

(7) 文件和数据库类

- 文件 I/O 类
- DAO 类
- ODBC 类

(8) Internet 和网络工作类

- ISAPI 类
- Windows Socket 类
- Win32 Internet 类

(9) OLE 类

- OLE 容器类
- OLE 服务器类
- OLE 拖放和数据传输类
- OLE 普通对话框类
- OLE 动画类
- OLE 控件类
- 活动文档类
- 其他文档类

(10) 调试和异常类

- 调试支持类
- 异常类

下面几小节简单介绍 MFC 中一些主要的类和某些子层次结构。

10.2.2 根类

CObject 类是 MFC 的抽象基类,是 MFC 中多数类和用户自定义子类的根类,它为程序员提供了许多编程所需的公共操作。这些操作包括:对象的建立和删除,串行化支持,对象诊断输出,运行时信息以及集合类的兼容等。

串行化是对象本身往返于存储介质的一个存储过程。串行化的结果是使数据"固定"在存储介质上。CObject 类定义两个在串行化操作中起重要作用的成员函数:Serialize 和 IsSerializable。程序可以调用一个由 CObject 派生的对象的 IsSerializable 函数来确定该对象是否支持串行化操作。建立一个支持串行化的类的步骤之一是重载继承自 CObject 类的 Serialize 函数,并提供串行化数据成员的派生类的专用代码。

CObject 的派生类同时还支持运行时类型信息。运行时的类型信息机制允许程序检索对象的类名及其他信息。CObject 提供两个成员函数来支持运行时类型信息:IsKindOf 和 GetRuntimeClass。函数 IsKindOf 指示一个对象是否属于规定的类或者是从规定的类中派生出来的。CRuntimeClass 类对象中包含了一个类的运行时信息,包括这个类的类名、基类名等信息,通过它可以很容易地获得一个指定类的运行时刻信息。

10.2.3　应用程序体系结构类

该类用于构造应用程序框架的结构，它能提供多数应用程序公用的功能。编写程序的任务是填充框架，添加应用程序专用的功能。应用程序体系结构类主要有与命令相关的类、窗口应用程序类、文档/视类和线程基类等。

1. 命令相关类：CCmdTarget 类

该类是 CObject 的子类，它是 MFC 库中所有具有消息映像属性类的基类。消息映像规定了当一对象接收到消息命令时，应调用哪一个函数对该消息进行处理。程序员很少需要从 CCmdTarget 类中派生出新类，如窗口类(CWnd)、应用程序类(CWinApp)、文档模板类(CDocTemplate)、文档类(CDocument)、视类(CView)及框架窗口类(CFrameWnd)等。

2. 窗口应用程序类：CWinApp 类

每个应用程序只有一个应用程序对象，在运行程序中该对象与其他对象相互协调，它是从 CWinApp 类中派生出来。CWinApp 类封装了初始化、运行、终止应用程序的代码

3. 文档/视类

文档对象由文档模板对象创建，管理应用程序的数据。视对象表示一个窗口的客户区，显示文档数据并允许用户与之交互。这些模板及基类有：

- CDocTemplate：文档模板基类。文档模板负责协调文档、视和框架窗口的创建。
- CSingleDocTemplate：单文档界面(SDI)的文档模板。
- CMultiDocTemplate：多文档界面(MDI)的文档模板。
- CDocument：应用程序专用文档的基类。
- CView：显示文档数据的应用程序专有视的基类。

4. 线程基类：CWinThread 类

所有线程的基类，可直接使用。CWinApp 类就是从 CWinThread 类中派生出来的。

10.2.4　可视对象类

1. 窗口类：CWnd 类

该类提供了 MFC 中所有窗口类的基本功能。它是 CCmdTarget 类的子类。创建 Windows 窗口要分两步进行：首先引入构造函数，构造一个 CWnd 对象，然后调用 Create 函数建立 Windows 窗口并将它连到 CWnd 对象上。MFC 中还从 CWnd 类派生出进一步的窗口类型以完成更具体的窗口创建工作，这些派生类有：

- CFrameWnd：框架窗口类，SDI 应用程序主框架窗口的基类。
- CMDIFrameWnd：多文档框架窗口类，MDI 应用程序主框架窗口的基类。
- CMDIChildWnd：多文档框架窗口类，MDI 应用程序文档框架窗口的基类。

2. 视类：CView 类

该类表示框架窗口的客户区和显示文档数据并接收输入的客户区，其派生类主要有如下几种：

- CScrollView：具有滚动功能的视的基类。
- CFormView：其布局在对话资源中定义的滚动视。

• CEditView：具有文本编辑、查找、替换和滚动功能的视。

3. 菜单类：CMenu 类

该类是 CObject 类的子类，用于管理菜单。它是一个 Windows HMenu 的封装，提供了与窗口有关的菜单资源建立、修改、跟踪及删除的成员函数。

4. 对话框类：CDialog 类

由于对话框是一个特殊的窗口，所以该类是从 CWnd 类中派生出来的。对话框子层次结构包括通用对话框类 CDialog 以及支持文件选择、颜色选择、字体选择、打印、替换文本的公共对话框子类。这些子类包括：

• CFileDialog：提供打开或保存一个文件的标准对话框。
• CColorDialog：提供选择一种颜色的标准对话框。
• CFontDialog：提供选择一种字体的标准对话框。
• CPrintDialog：提供打印一个文件的标准对话框。
• CFindReplaceDialog：提供一次查找并替换操作的标准对话框。
• CDialog：该类可用于建立模态对话框和无模态对话框模型，是该子层次结构的根。

5. 控件类

控件子层次结构包括若干类，使用这些类可建立静态文本、命令按钮、位图按钮、列表框、组合框、滚动条、编辑框等。这些直观控件为 Windows 应用程序提供了各种输入和显示界面。主要的控件类如下：

• CStatic：静态文本控件窗口。常用于标注、分隔对话框或窗口中的其他控件。
• CButton：按钮控件窗口。该类为对话框或窗口中的按钮、检查框或单选按钮提供一个总的接口。
• CEdit：编辑控件窗口。编辑控件用于接收用户的文字输入。
• CRichEditCtrl：多信息编辑控件。除了编辑控件的功能外，还支持字符和图形格式，以及 OLE 对象。
• CScrollBar：滚动条控件窗口。该类提供滚动条的功能，用做对话框或窗口中的一个控件，用户可通过它在某一范围内定位。
• CProgressCtrl：进度指示控件窗口。用于指示一个操作的进度。
• CSliderCtrl：游标控件窗口。包括一个可移动的游标，用户可移动游标选择一个值或一个范围。
• CListBox：列表框控件窗口。列表框用于显示一个组列表项，用户可以进行观察和选择。
• CComboBox：组合框控件窗口。组合框由一个编辑控件加一个列表框组成。
• CBitmapButton：带有位图而非文字标题的按钮。
• CSpinButtonCtrl：带有一个双向箭头的按钮，单击某个箭头按钮可增大或减小值。
• CAnimateCrtl：动画显示控件窗口，显示一个简单的视频图像。
• CToolTipCtrl：一个小的弹出式窗口，显示一行文本，描述应用程序中一个工具的作用。
• CHotKeyCtrl：热键控件窗口，使用户可以创建一个“热键”，以快速地执行某项操作。

6. 控件条类:CControlBar 类

控件条子层次结构为工具条、状态条、对话条和分割窗口建立模型。该类是 CToolBar、CStatusBar、CDialogBar 的基类,负责管理工具条、状态条、对话条的一些成员函数。控件条指的是连接在主窗口框架的顶部或底部的小窗口,它具有如下基类:

- CStatusBar:状态条控件窗口的基类。
- CToolBar:包含非基于 HWND 的位图式命令按钮的工具条控件窗口。
- CDialogBar:控件条形式的非模态对话框。

7. 绘画对象类:CGdiObject 类

图形绘画对象子层次结构以 CGdiObject 类为根类,可用于建立绘画对象模型,如画笔、刷子、字体、位图、调色板等。这些子类有:

- CBitmap:封装一个 GDI 位图,提供一个操作位图的接口。
- CBrush:封装一个 GDI 画刷,可被选择为设备描述表的当前画刷。
- CFont:封装一种 GDI 字体,可被选择为设备描述表的当前字体。
- CPalette:封装一个 GDI 调色板,用作应用程序和一彩色输出设备,如显示器之间的接口。
- CPen:封装一种 GDI 画笔,可被选择为设备描述表的当前画笔。
- CRgn:封装一个 GDI 域,用于操作窗口内的椭圆域或多边形域。该类与 CDC 类的裁剪成员函数一起使用。

8. 设备描述表类:CDC 类

该类及其子类支持设备描述表对象,是 CObject 类的子类。CDC 类是一个较大的类,包括许多成员函数,如映像函数、绘画工具函数、区域函数等,通过这些成员函数可以完成所有的绘画工作。CDC 类具有如下的子类:

- CPaintDC:显示描述表。用于窗口的 OnPaint 成员函数和视的 OnDraw 成员函数中,自动调用 BeginPaint 进行构造,调用 EndPaint 进行析构。
- CClientDC:窗口客户的显示描述表。例如,用于在快速响应鼠标事件时进行绘画。
- CWindowDC:整个窗口的显示描述表,包括客户区和框架区。
- CMetaFileDC:Windows 元文件的设备描述表。Windows 元文件包含一个图形设备接口(GDI)命令序列,该序列可被重新执行而创建一幅图像。对 CMetaFileDC 的成员函数的调用记录在一个元文件中。

10.2.5　通用类

此分类中的类提供了许多通用服务,例如文件 I/O、诊断和异常处理等,此外还包括如数组和列表等存放数据集的类。

1. 文件类:CFile 类和 CArchive 类

如果想编写自己的输入/输出处理函数,可以使用 CFile 类和 CArchive 类,一般不必再从这些类中派生新类。如果使用程序框架,则只需提供关于文档如何将其内容串行化的详细信息,File 菜单上的 Open 和 Save 命令的默认实现将会处理文件 I/O(使用类 CArchive)。如下是部分文件类:

- CFile 类：提供访问二进制磁盘文件的总接口。
- CMemFile 类：提供访问驻内存文件的总接口。
- CStdioFile 类：提供访问缓存磁盘文件的总接口，通常采用文本方式。
- CArchive 类：与 CFile 对象一起通过串行化实现对象的永久存储。

2. 异常类：CException 类

该类是所有异常情况的基类。不能直接建立 CException 对象，程序员只能建立派生类的对象。可以使用派生类来捕获指定的异常情况。CException 的派生类如下：

- CArchiveException 类：归档异常。
- CFileException 类：有关文件的异常。
- CMemoryException 类：内存不够异常。
- CNotSupportedException 类：使用未支持特征产生的异常。
- CResourceException 类：装载 Windows 资源失败产生的异常。
- CUseException 类：用于停止用户启动的操作异常。

3. 模板收集类

这些类可以将多种对象存放到数组、列表和"映像"中。但这些收集类是模板，它们的参数确定了存放在集合中的对象类型。CArray、CMap 和 CList 类使用全局帮助函数，帮助函数通常必须定制。类型指针类是类库中其他类的包装类，利用这些包装类，应用程序可借助于编译器的类型检查以避免出错。下列是一部分模板收集类：

- CArray 类：将元素存储在数组中。
- CMap 类：将键映像到值。
- CList 类：将元素存储在一链表中。
- CTypedPtrList 类：将对象指针存储在一链表中的类型。
- CTypedPtrArray 类：将对象指针存储在一数组中的类型。
- CTypedPtrMap 类：将链映像到值的类型，键和值都为指针。

10.2.6 OLE 类

对象连接与嵌入(OLE)子层次结构为支持 OLE 提供了 9 个类，这 9 个类分为 3 大类：普通类、客户类和服务器类。其中 COleDocument、COleItem、COleException 为支持 OLE 的普通类，COleClientDoc、COleClientItem 为支持 OLE 的客户类，COleServer、COleTemplate、COleServerDoc、COleServerItem 为支持 OLE 的服务类。这些类提供了支持 OLE 的所有功能。

10.2.7 ODBC 数据库类

开放数据库连接子层次结构提供了一些类来支持 ODBC 特征，即通过这些类可开发数据库应用程序来访问多个数据库文件。该层次结构中主要包括的类有：

- CDataBase：封装对一数据源的连接，通过此连接应用程序可在该数据源上进行操作。
- CRecordset：封装从一数据源选出的一组记录。
- CRecordView：提供直接连接一记录集对象的格式视。

- CFieldExchange：提供上下文信息，支持记录字段交换，即在字段数据成员、记录对象的参数数据成员及数据源上的对应列表之间进行数据交换。
- CLongBinary：封装一存储句柄，用于存储二进制大对象，例如位图等。
- CDBException：对数据存取处理过程中的失败产生的异常。

10.3　MFC 中的全局函数与全局变量

在 MFC 中提供的所有函数及成员变量中，以 Afx 开头的函数除数据库类函数和 DDX（Dialog Data Exchange）函数外，在目前的版本中，都表示为一个全局函数。而以 Afx 为前缀的变量，都是全局变量。这些函数与变量可以在任何地方、任何 MFC 应用程序中调用。

表 10-1　MFC 中的部分全局函数

函数名	功能简介
AfxAbort	无条件终止一个应用程序
AfxBeginThread	创建一个新线程并执行它
AfxEndThread	终止当前正在执行的线程
AfxFormatString	格式化字符串
AfxMessageBox	显示一个 Windows 消息框
AfxGetApp	返回当前应用程序对象的指针
AfxGetInstanceHandle	返回标识当前应用程序对象的句柄
AfxRegisterWndClass	登记用于创建 Windows 窗口的窗口类

10.4　应用程序向导

VC ++ 6.0 为了减轻程序员的工作量，特别增强了应用程序向导的功能。应用程序向导为程序员提供了一个基于 MFC 的应用程序框架，程序员只需在此基础上添加实现特定功能的代码即可。通过应用程序向导建立应用程序框架一般可以通过以下步骤来实现。

（1）在 File 菜单下选择 New，在 New 对话框（图 10-1）中选择 MFC AppWizard（exe），在 Project Name 文本输入框中输入新建的项目名如 Huangweitong 后，单击 OK 按钮。

（2）在弹出的 MFC AppWizard-Step 1 对话框（如图 10-2 所示）中可以设置应用程序的类型，指定应用程序的结构是否采用文档-视图结构（将在第 16 章详细叙述），以及资源文件所使用的语种等。应用程序的类型包括以下几种：

- Single Document：单文档应用程序。
- Multiple Documents：多文档应用程序。
- Dialog based：基于对话框的应用程序。

应用程序资源文件所使用的语种可以通过下拉列表选择，在中国一般都选用中文作为资源文件的语种。

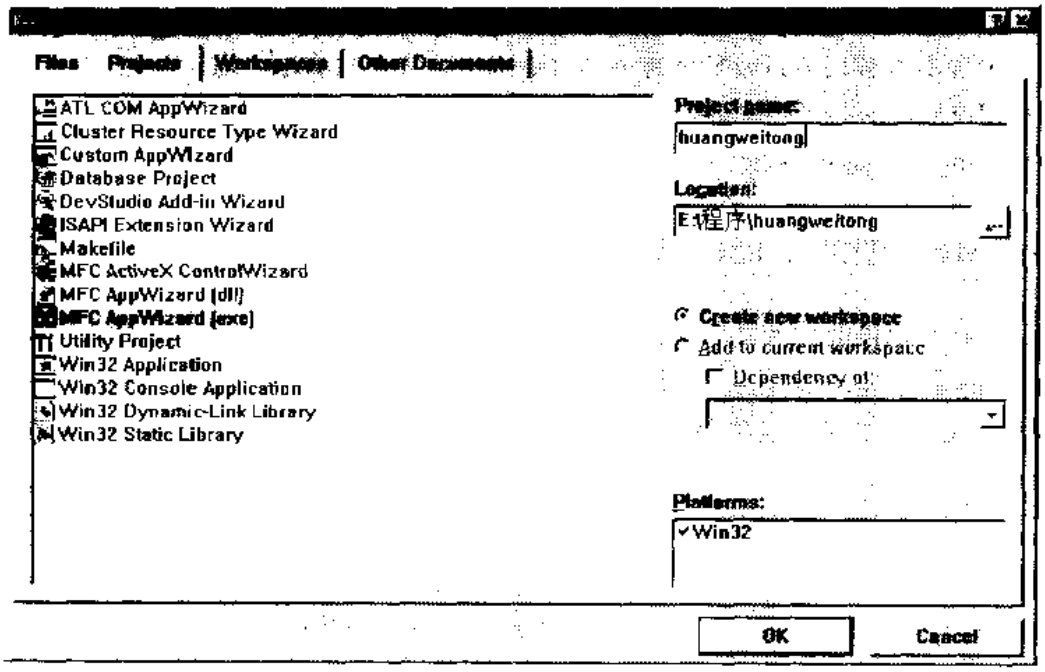


图 10-1 New 对话框

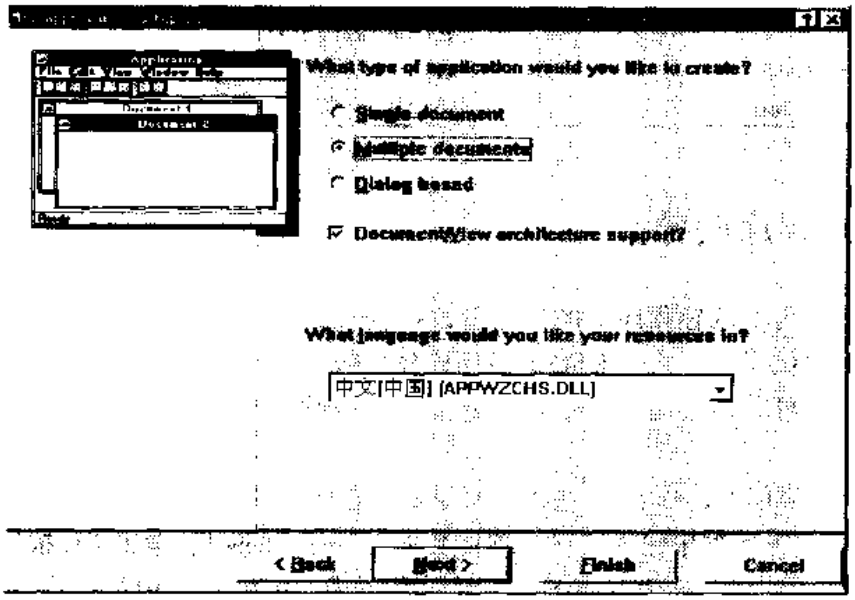


图 10-2 MFC AppWizard-Step 1 对话框

(3) 在设置好上述选项后,单击 Next 按钮,将弹出 MFC AppWizard-Step 2 of 6 对话框(如图 10-3 所示)。在 MFC AppWizard-Step 2 of 6 对话框中可以设置应用程序所使用的数据库支持,它可以是以下值之一:

- None:在应用程序中忽略所有的数据库支持。
- Header files only:包括定义基本数据库类的头文件,但不创建对应特定表的数据库类或视图类。

- Database view without file support：创建对应指定表的一个数据库类和一个视图类，不附加标准文件支持。
- Database view with file support：创建对应指定表的一个数据库类和一个视图类，并附加标准文件支持。

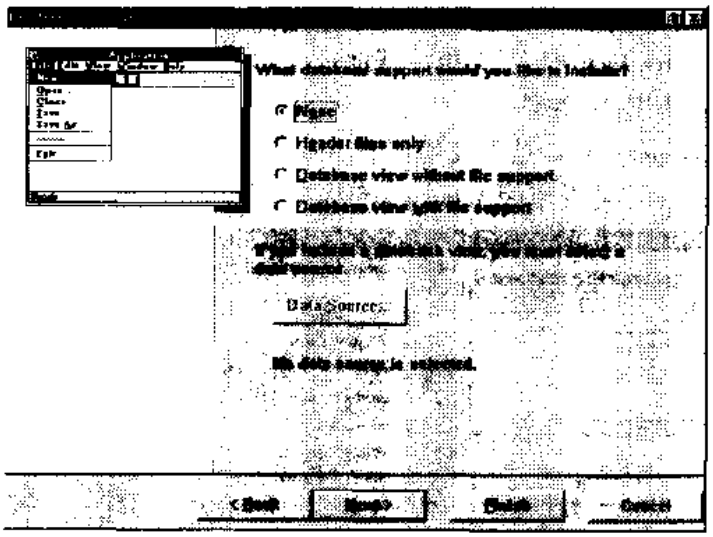


图 10-3　MFC AppWizard-Step 2 of 6 对话框

(4) 设置好应用程序的数据库支持后，单击 Next 按钮，将弹出 MFC AppWizard-Step 3 (如图 10-4 所示) of6 对话框。在 MFC AppWizard - Step3 of6 对话框中可以设置应用程序

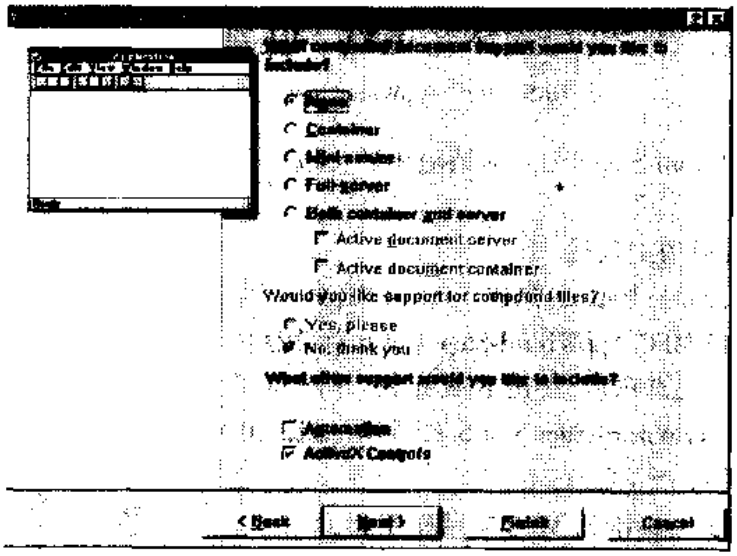


图 10-4　MFC AppWizard-Step 3 of 6 对话框

所支持的复合文档类型，这一步使程序员可以向应用程序中加入 OLE 支持。设置好应用程序对 OLE 的支持后，单击 Next 按钮，进入 MFC AppWizard-Step 4 of 6 对话框(如图 10-5 所示)。

(5) 在 MFC AppWizard-Step 4 of 6 对话框中可以设置应用程序的外观。其中可以配置的选项如下：

- Docking Toolbar：一个工具栏。
- Initial status bar：一个初始状态栏。
- Printing and print preview：打印和打印预览支持。
- Context-sensitive Help：上下文相关帮助。
- 3D controls：使用 3D 控件。
- MAPI (Messaging API)：是否使用消息 API。
- Windows Sockets：是否使用 Windows 套接字。

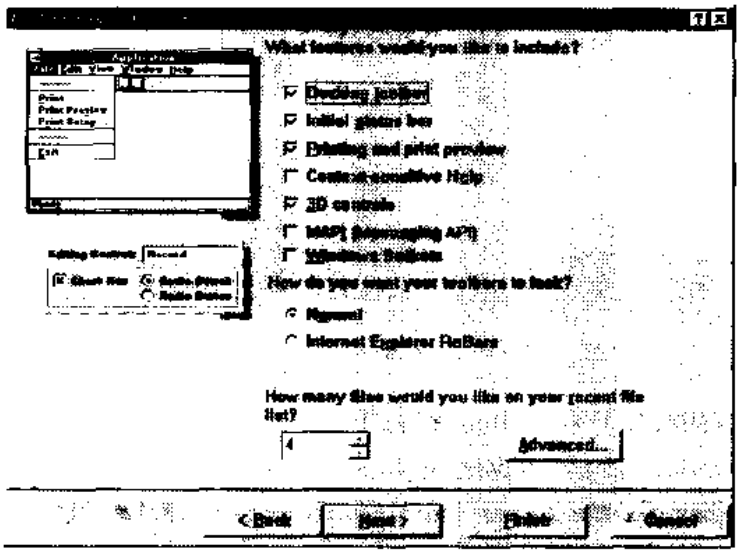


图 10-5 MFC AppWizard-Step 4 of 6 对话框

在 MFC AppWizard-Step 4 of 6 对话框中还可以设置工具栏的外观，它可以是以下两个值之一：

- Normal：普通的工具栏。
- Internet Explorer ReBars：具有 Internet Explorer 风格的工具栏。

同时还可以在 MFC AppWizard-Step 4 of 6 对话框中设置需要显示的最近使用的文件个数。在设置好上述选项后，单击 Next 按钮，弹出 MFC AppWizard-Step 5 of 6 对话框。

(6) 在 MFC AppWizard-Step 5 of 6 对话框(如图 10-6 所示)中可以设置应用程序的风格，它可以是下面两个值之一：

- MFC Standard：标准的 MFC 应用程序。
- Windows Explorer：具有 Windows Explorer 风格的应用程序。

在 MFC AppWizard-Step 5 of 6 对话框中还可以设置是否在应用程序向导生成的代码中加注注释。它可以是下面两个值之一：

- Yes, please：在向导生成的代码中加注注释。
- No, thank you：在向导生成的代码中不加注注释。

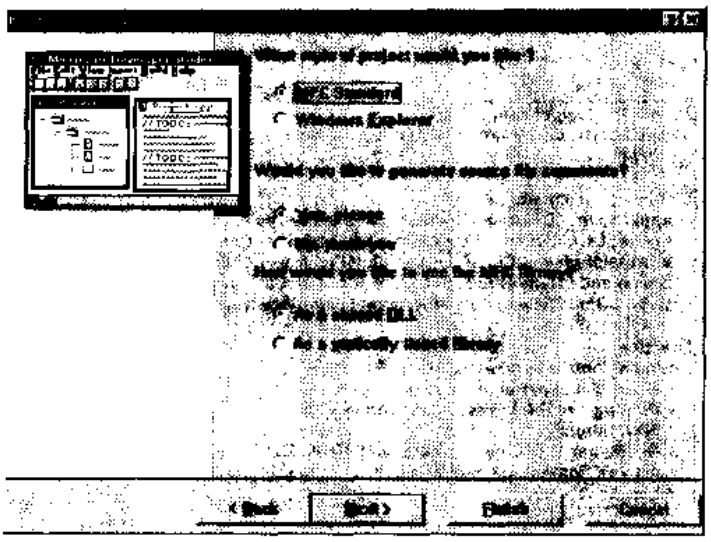


图 10-6　MFC AppWizard-Step 5 of 6 对话框

同时在 MFC AppWizard-Step 5 of 6 对话框中还可以设置使用 MFC 库文件的方式，它可以是下面两个值之一：

- As a shared DLL：以共享动态链接库的方式使用 MFC 库文件。
- As a Statically linked library：以静态链接库的方式使用 MFC 库文件。

(7) 在设置好上述选项后，单击 Next 按钮，进入 MFC AppWizard-Step 6 of 6 对话框(如图 10-7 所示)。在 MFC AppWizard-Step 6 of 6 对话框中可以设置向导生成的文件名和类名。

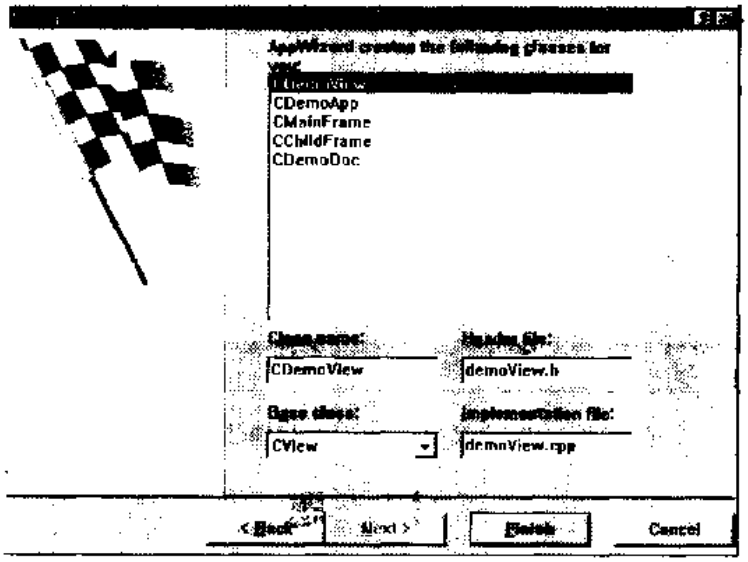


图 10-7　MFC AppWizard-Step 6 of 6 对话框

(8) 在设置好文件名和类名后，单击 Finish 按钮，弹出 New Project Information 对话框，在该对话框中列出了关于新建立的应用程序工程文件的相关信息，如图 10-8 所示。

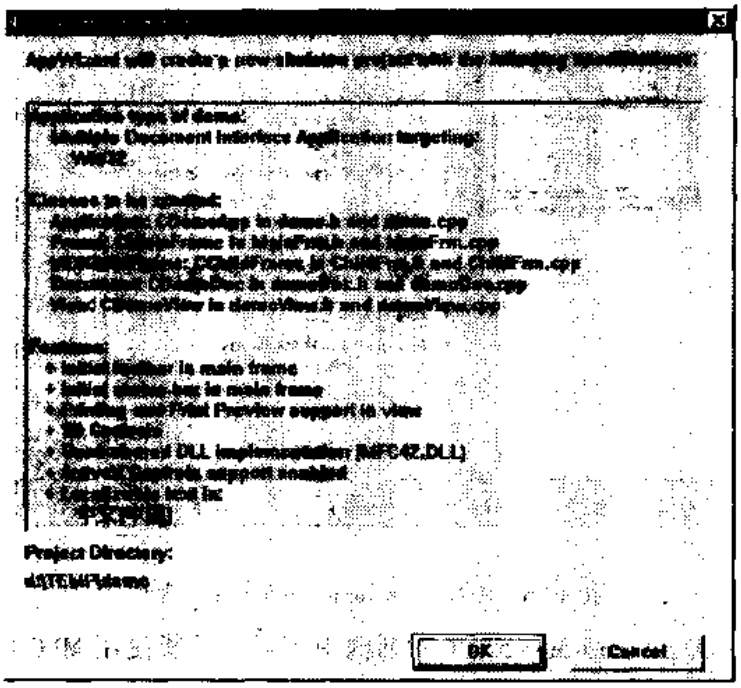


图 10-8 New Project Information 对话框

这样，应用程序向导所有的工作就全部完成了，这时向导已经为用户生成了一个可执行的应用程序框架，编译并执行该应用程序，结果如图 10-9 所示。

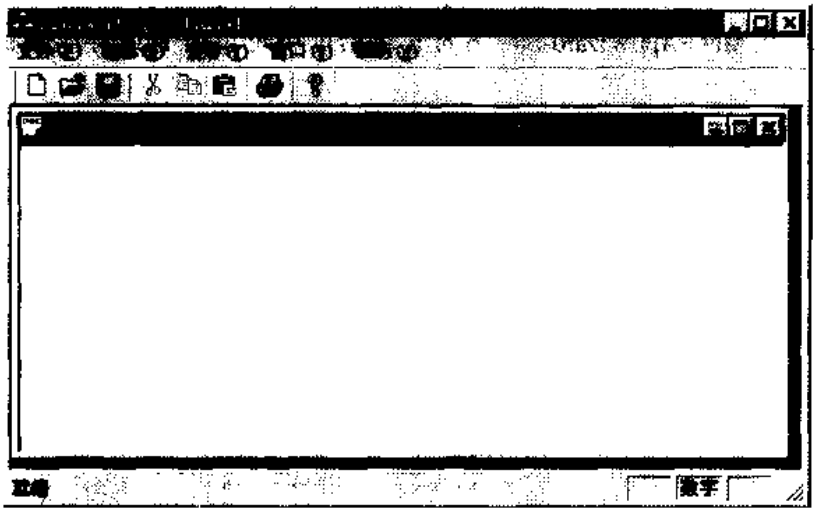


图 10-9 MFC 自动生成的应用程序框架的运行结果

习 题

10-1 简述 MFC 类的层次结构。

10-2 MFC 类层次中主要包含了哪些类？

10-3 如何使用应用程序向导？

10-4 在应用程序向导中能够创建哪些类型的文件？

第 11 章

编辑框类及其应用

8.7 节中讲述了如何使用编辑框控件开发带有编辑框的应用程序，本章将通过一个实例学习如何使用 MFC 中的编辑框类在对话框窗口中制作编辑框。

11.1 编辑框类的结构及其方法

编辑框控件看起来是个非常简单的矩形窗口，但它具有许多功能。编辑框控件可以自带滚动条，显示多行文本。事实上，Windows 中的记事本(Notepad)应用程序就是一个带有控件菜单的编辑框控件。编辑框控件有两种形式，一种是单行编辑框控件，另一种是多行编辑框控件。熟悉编辑框的制作对于 Windows 编程来说是很重要的。

MFC 在类 CEdit 中提供标准的 Windows 编辑框控件服务，CEdit 是 CWnd 类直接派生来的，它具有 CWnd 的所有功能，它在 MFC 类库中的层次位置如图 11-1 所示。

像大多数包含标准 Windows 控件的 MFC 类一样，CEdit 类的结构很复杂。当创建 CEdit 类的对象时，MFC 自动赋予该对象一个标准的 Windows 编辑框控件，它定义了 CEdit 对象，其中包括方法原型。CEdit 类的定义如下所示：

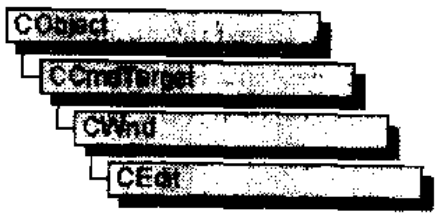


图 11-1 CEdit 类在 MFC 类库中的层次位置

```
class CEdit : public CWnd
{
DECLARE_DYNAMIC(CEdit)
// 定义构造函数
public:
CEdit();
BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

// 定义成员函数
BOOL CanUndo() const;
int GetLineCount() const;
BOOL GetModify() const;
void SetModify(BOOL bModified = TRUE);
void GetRect(LPRECT lpRect) const;
DWORD GetSel() const;
```

```
void GetSel(int& nStartChar, int& nEndChar) const;
HLOCAL GetHandle() const;
void SetHandle(HLOCAL hBuffer);
#if (WINVER >= 0x400)
void SetMargins(UINT nLeft, UINT nRight);
DWORD GetMargins() const;
void SetLimitText(UINT nMax);
UINT GetLimitText() const;
CPoint PosFromChar(UINT nChar) const;
int CharFromPos(CPoint pt) const;
#endif

// 注意:lpszBuffer的第一个字必须包含buffer的长度
int GetLine(int nIndex, LPTSTR lpszBuffer) const;
int GetLine(int nIndex, LPTSTR lpszBuffer, int nMaxLength) const;

// 操作
void EmptyUndoBuffer();
BOOL FmtLines(BOOL bAddEOL);

void LimitText(int nChars = 0);
int LineFromChar(int nIndex = -1) const;
int LineIndex(int nLine = -1) const;
int LineLength(int nLine = -1) const;
void LineScroll(int nLines, int nChars = 0);
void ReplaceSel(LPCTSTR lpszNewText, BOOL bCanUndo = FALSE);
void SetPasswordChar(TCHAR ch);
void SetRect(LPCRECT lpRect);
void SetRectNP(LPCRECT lpRect);
void SetSel(DWORD dwSelection, BOOL bNoScroll = FALSE);
void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);
BOOL SetTabStops(int nTabStops, LPINT rgTabStops);
void SetTabStops();
BOOL SetTabStops(const int& cxEachStop); //

// 剪贴板操作
BOOL Undo();
void Clear();
void Copy();
void Cut();
void Paste();
```

```
BOOL SetReadOnly(BOOL bReadOnly = TRUE);
int GetFirstVisibleLine() const;
TCHAR GetPasswordChar() const;

// Implementation
public:
virtual ~CEdit();      //虚拟析构函数
};
```

编辑框控件的默认模式是在一行显示所有编辑文本，表 11-1 是通用 CEdit 类的方法。

表 11-1　通用 CEdit 类的方法

方　法	说　明
CanUndo( )	决定一个编辑操作是否可以撤销
Clear( )	从编辑框控件中删除当前的选择(如果有的话)
Copy( )	将编辑框控件当前的选择(如果有的话)以 CF_TEXT 格式复制到剪贴板中
Cut( )	剪下编辑框控件中的当前选择(如果有的话)并以 CF_TEXT 格式复制到剪贴板中
EmptyUndoBuffer( )	消除一个编辑框控件的"撤销"标志
GetFirstVisibleLine( )	确定编辑框控件中的最上面的可视行
GetModify( )	确定一个编辑框控件的内容是否可修改
GetPasswordChar( )	当用户输入文本时，获得编辑框控件中显示的密码字符
GetRect( )	获得一个编辑框控件的格式化矩形
GetSel( )	获得编辑框控件中当前选择的开始和结束字符位置
LimitText( )	限定用户可能输入一编辑框控件的文本长度
LineFromChar( )	获得包含指定字符下标的行的行号
LineLength( )	获得编辑框控件中的一行的长度
LineScroll( )	滚动多行编辑框控件的文本
Paste( )	将剪贴板的数据插入到编辑框控件中当前的光标位置，只有当前剪贴板中数据格式为 CF_TEXT 时方可插入
ReplaceSel( )	用指定文本替代编辑框控件中当前选择的部分
SetModify( )	设置或清除编辑框控件的修改标志
SetPasswordChar( )	当用户输入文本时设置或删除一个显示于编辑框控件中的密码字符
SetReadOnly( )	将编辑框控件设置为只读状态
SetSel( )	在编辑框控件中选择字符的范围
Undo( )	取消最后一个编辑框控件操作

当编辑框控件具有 ES_MULTILINE 样式时，多行编辑框控件支持在编辑窗口进行多行文本编辑，表 11-2 是多行编辑所支持的 CEdit 类的方法。

表 11-2 多行编辑所支持的 CEdit 类的方法

方 法	说 明
FmtLines( )	设置在多行编辑框控件中包含软分行符
GetHandle( )	获得当前分配给一个多行编辑框控件的内存的句柄
GetLine( )	从一编辑框控件中获得一行文本
GetLineCount( )	获得多行编辑框控件的行数
LineIndex( )	设置多行编辑框控件中一行的字符下标
SetHandle( )	设置多行编辑框控件将要用到的句柄
SetRect( )	设置多行编辑框控件的格式化矩形并更新控件
SetRectNP( )	设置多行编辑框控件的格式化矩形并且不重绘控件窗口
SetTabStops( )	在多行编辑框控件中设置制表(tab)位

11.2 编辑框类编程实例

【例 11-1】

编写基于对话框的应用程序，其窗口布局如图 11-2 所示，标题为 Application of Edit Box。在程序主窗口中有两个编辑框，分别为Edit1和Edit2，而且都带有水平和垂直滚动

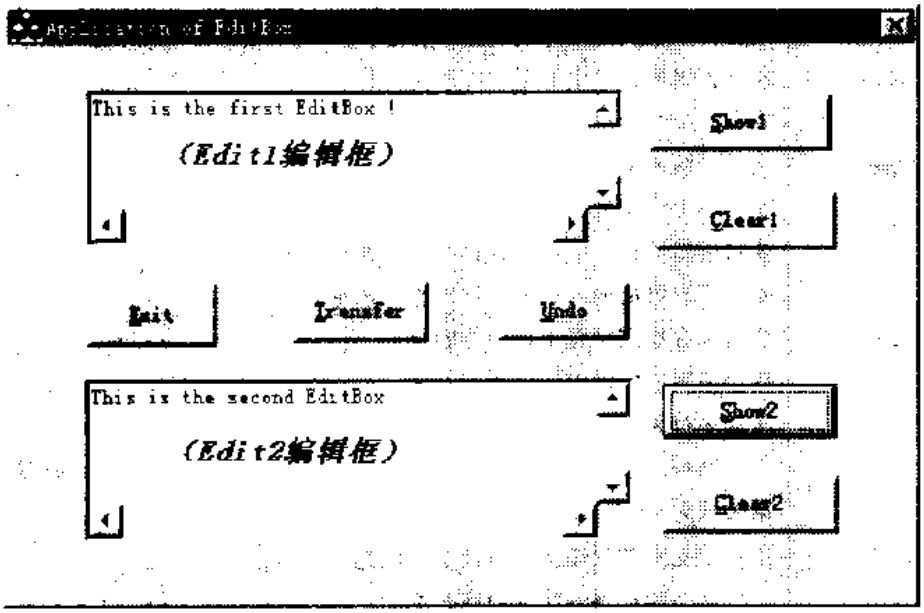


图 11-2 【例 11-1】应用程序的界面

条。在这两个编辑框中可进行多行编辑。主窗口中还有 Show1、Show2、Clear1、Clear2、Exit、Undo 和 Transfer 七个按钮。若单击 Show1 按钮，则在 Edit1 编辑框中显示一段文本"This is the first EditBox!"；单击 Clear1 按钮，则 Edit1 编辑框中的内容被清除；若单击 Show2 按钮，则在 Edit2 编辑框中显示一段文本"This is the second EditBox"；单击 Clear2 按钮，则 Edit2 编辑框中的内容被清除；如果单击 Transfer 按钮，则把 Edit1 编辑框的内容复制到 Edit2 的编辑框中去；单击 Undo 按钮，则取消编辑框中的上一次操作，再单击一次 Undo 按钮，又显示

刚才的内容;若单击 Exit 按钮,则退出程序的运行。

1. 建立应用程序的工程文件

用 MFC 的 AppWizard 建立一个应用程序时,首先要生成该应用程序的工程文件,其步骤如下:

(1) 在 Visual C++ 的主菜单栏中选择 File 菜单中的 New 选项,出现一个 New 对话框,该窗口中有四个选项卡:Files、Projects、Workspaces 和 Other Documents。

(2) 选择 Projects 选项卡,并在该选项卡中选择 MFC AppWizard(exe)项。

(3) 在 Projects name 编辑框中输入要创建的工程文件的名称。这里给应用程序取名为 EditBox。

(4) Location 编辑框中描述的是放置该工程文件的路径(位置),可以根据用户需要输入自己希望放置该文件的位置。

(5) 在 Platforms 编辑框中选择 Win32,因为这里使用的 Visual C++ 6.0 是在 32 位操作系统下。

(6) 确定工程文件的类型、名称和路径选项后,单击 OK 按钮,就可以开始制作该工程文件了。

(7) Visual C++ 显示 MFC AppWizard-Step1 对话框。这是生成工程文件的第一步,在这一步中,用户将选择创建什么类型的应用程序以及资源文件使用什么语言。

- 因为要生成的应用程序是基于对话框的应用程序,所以在"What type of application would you like to careate?"中,选择 Dialog based 选项。
- 在"What language would you like your resources in?"中,选择"中文[中国](APPWZCHS.DLL)"项。
- 单击 Next 按钮,进入下一步。

(8) Visual C++ 显示 MFC AppWizard-Step 2 of 4 对话框,这是生成工程文件的第二步。

- 在"What features would you like to include?"问题中,选择 About Box(关于 About 对话框)和 3D Controls(使用三维图形控件)选项。
- 在"Please enter a title for your dialog"编辑框中可以给应用程序主窗口命名。这个例子中使用工程文件的名称 Application of EditBox,此名称将显示在应用程序窗口的标题栏。
- 确定以上各项后,单击 Next 按钮,进入下一步。

(9) Visual C++ 显示 MFC AppWizard-Step 3 of 4 对话框。

- 在"Would you like to generate source file comments?"中,询问是否在生成源代码文件中加注释,应选择 Yes,因为在 AppWizard 生成的应用程序源代码文件(.cpp 文件)中加上代码的注释,有助于阅读和理解源代码的含义。
- 在"How would you like to use the MFC library?"问题中,选择使用 MFC 的动态链接库(DLL),而不是静态链接库。使用动态链接库可以在应用程序运行的时候链接起来,而不是在编译的时候就链接,使生成的应用程序.exe 文件比较小。
- 单击 Next 按钮,进入最后一步。

(10) Visual C++ 显示出生成工程文件的最后一步，出现 MFC AppWizard-Step 4 of 4 对话框，可以看到 AppWizard 给出的所创建的类名和文件名。如果用户愿意，可以改变类的名称，但一般不必改变它，本例子中对这个窗口不做任何改动。

(11) 单击 Finish 按钮，显示一个 New Project Information 对话框，该对话框显示了前面几步所选择的全部设置，包括应用程序类型、新创建的类以及应用程序的全部特征，还显示了该应用程序将生成在哪个路径之中。通过这些信息，用户可以最后检查一下应用程序的设置是否完全正确，如果发现不对的地方，可以单击 Cancel 键取消前面的操作，用 AppWizard 重新制作。检查后若完全正确，则单击 Exit 按钮，AppWizard 完成应用程序的自动生成工作，在指定的目录下生成应用程序框架所必须的全部文件。

2. 设计应用程序的界面

AppWizard 已经为 EditBox 应用程序生成了工程文件和工程工作区文件，在 EditBox 的工作区中选择 ResourceView 选项卡。

因为 EditBox 应用程序是基于对话框的，所以 AppWizard 生成一个对话框窗口作为应用程序的主窗口，并且给这个对话框取名为 IDD_EDITBOX_DIALOG，现在要做的工作就是在这个对话框窗口中进行界面设计。

(1) 在 ResourceView 选项卡中打开 Dialog 资源组，然后双击 IDD_EDITBOX_DIALOG。Visual C++ 6.0 将在右边的工作台中显示可以进行可视化编辑的 IDD_EDITBOX_DIALOG 对话框，并打开一个控件工具栏。控件工具栏在 ResourceView 选项卡的左边，它提供了 VC 中所有可建立的控件类型，用户只要把鼠标停在某个控件标识上，即可显示该控件的类型，如图 11-3 中把鼠标停在编辑框控件上，下面就显示 Edit Box 字样。

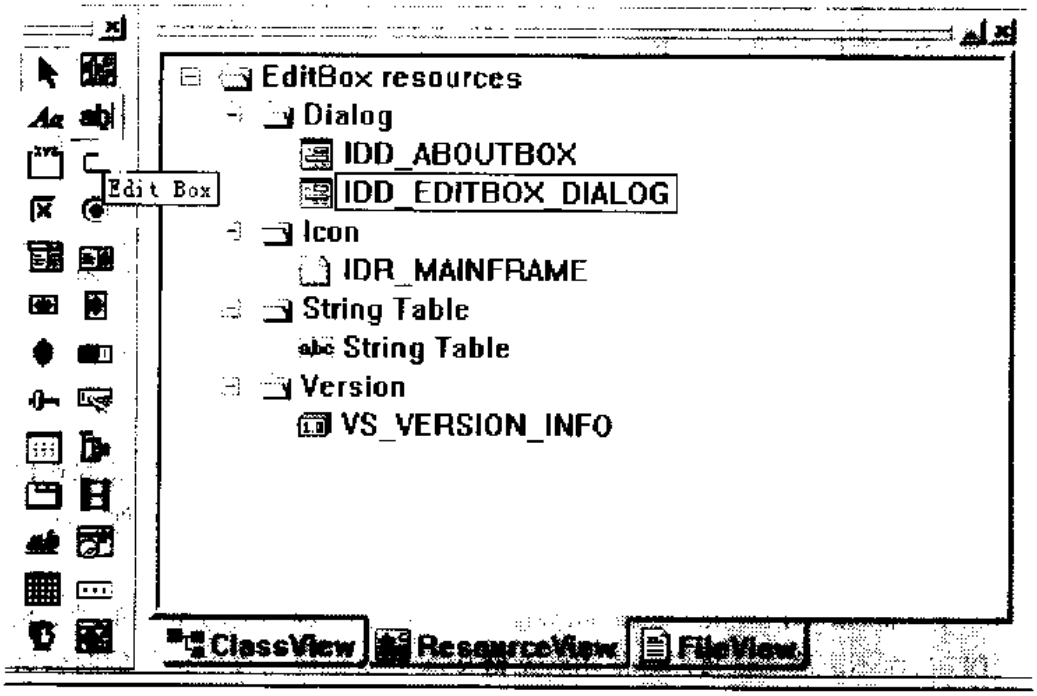


图 11-3 控件工具栏

(2) 删除 AppWizard 在对话框窗口中生成的控件，即删除一个 Exit 按钮、一个 Cancel 按钮和一个文本框，删除这些资源后，才能在对话框中制作 EditBox 应用程序的界面。

(3) 单击控件工具栏中的 Edit Box 按钮，在编辑状态下的对话框中布置如图 11-1 所示的上面的编辑框。然后将鼠标移到 Edit1 编辑框上，单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，显示一个属性窗口，在此窗口输入控件的 ID 标识值，这个标识值将在随后的编程中为识别该控件提供依据，Edit1 编辑框的 ID 为 IDC_EDIT1，设为默认值。

(4) 使用控件工具栏中的各种命令按钮制作应用程序的对话框窗口中的其他界面元素，完成以后如图 11-1 所示。

用同样的方法设置这些界面元素的属性，如表 11-3 所示。

表 11-3　对话框中各个对象的属性设置

对　象	ID	Caption
编辑框	IDC_EDIT1	无
编辑框	IDC_EDIT2	无
命令按钮	IDC_SHOW1_BUTTON	Show1
命令按钮	IDC_CLEAR1_BUTTON	Clear1
命令按钮	IDC_SHOW2_BUTTON	Show2
命令按钮	IDC_CLEAR2_BUTTON	Clear2
命令按钮	IDC_COPY_BUTTON	Transfer
命令按钮	IDC_EXIT_BUTTON	&Exit
命令按钮	IDC_UNDO_BUTTON	&Undo

现在我们已经完成了对话框窗口的设计，这都是通过可视化编程实现的。然而现在的对话框不能执行任何操作，因为还没有给它添加任何代码。

3. 给界面对象连接变量

每一个控件实际上就是一个对象，对 MFC 类库中的函数调用都是通过对象实现的。EditBox 中的编辑框是从 CEdit 类中派生出来的。每一个具体的界面对象都有一个 ID 标识值，如 Clear1 按钮的 ID 标识值为 IDC_CLEAR1_BUTTON，它是 CButton 的对象。在源程序中对这个对象操作，就是通过对与它相连接的变量进行操作来完成的。这个连接的变量就是类中的一个成员变量。

下面是给编辑框 IDC_EDIT1 连接一个变量 m_Edit1 的步骤：

(1) 把鼠标移到编辑框 Edit1 上，单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令，此时显示一个 MFC ClassWizard 对话框。

(2) 选择 Member Variables 选项卡，如图 11-4 所示。

(3) 在 Control IDs 列表框中单击 IDC_EDIT1 项，使之高亮显示。

(4) 单击右边的 Add Variable 按钮，此时 Visual C++ 显示一个 Add Member Variable 对话框。如图 11-5 所示。

(5) 设置成员变量名为 m_Edit1，类别为 Control，变量类型为 CEdit，如图 11-5 所示。

注意，不要将 m_Edit1 设为 CString 类型，因为只有设为 CEdit 类型，才能够调用 CEdit 类中的一些成员函数，如 Copy()，Paste() 等，这样才能完成本应用程序中对 Transfer 按钮要求复制内容的操作的响应。

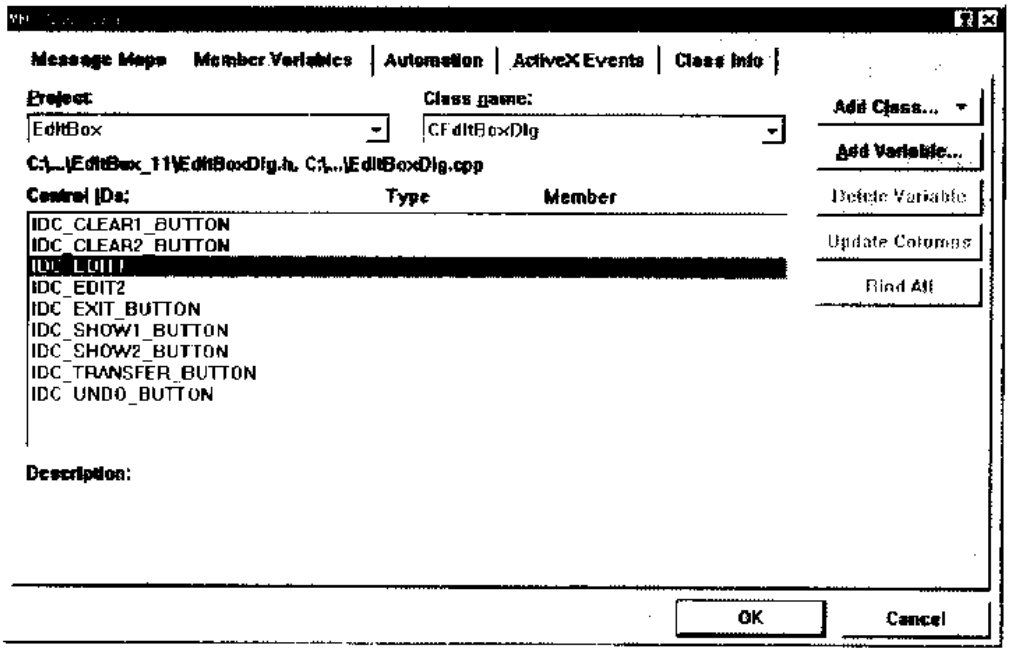


图 11-4 MFC ClassWizard 对话框中的 Member Variables 选项卡

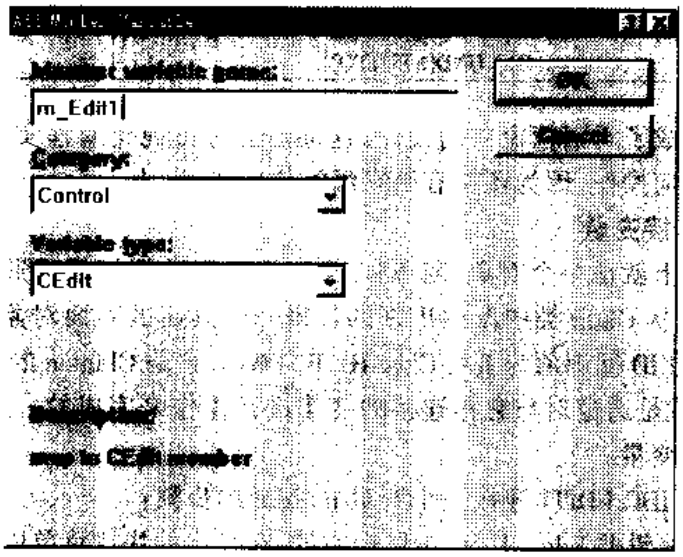


图 11-5 Add Member Variable 对话框

(6) 单击 OK 按钮,Visual C++ 把 m_Edit1 变量加到了 CEditBoxDlg 类中。

(7) 单击 ClassWizard 中的 OK 按钮,这样编辑框 EDIT1 就连接了一个变量 m_Edit1。

可以用同样的方法给编辑框 EDIT2 等其他对象连接各自的变量,增加变量之后 MFC ClassWizard 对话框将变为图 11-6 所示的样子。

4. 修改编辑框的特性

若此时运行该程序,在 Edit 文本框中不能进行多行输入,只能单行输入文本,而且如果用户按回车键将终止应用程序,同时,文本框中也没有水平和垂直滚动条。

下面，我们来完善应用程序中编辑框的特性，使之能够进行多行文本的输入。

在 EditBox 的编辑状态下，将鼠标移到编辑框 Edit1 上，单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，此时弹出 Edit Properties 对话框。选择 Styles 选项卡，然后选中 Multiline 复选框，说明可以进行多行输入；为增加水平及垂直滚动条，选中 Horizontal scroll、Auto HScroll、Vertical scroll、Auto Vscroll、Want return 和 Border 复选框，如图 11-7 所示。

其中，Want return 复选框很重要，如果不选择这一项的话，程序运行时在编辑框中按回车键就会终止应用程序；若选择了这一项的话，按回车键就可以进行下一行的输入，实现多行文本的输入。MFC 中 CEdit 控件的窗口样式如表 11-4 所示。

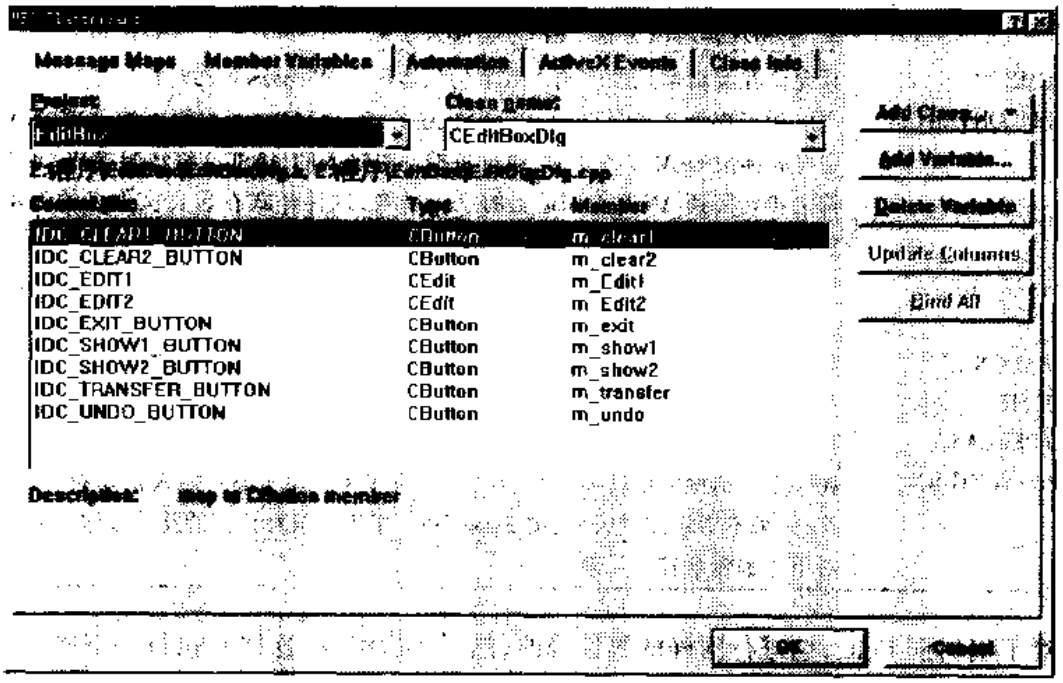


图 11-6　增加成员变量

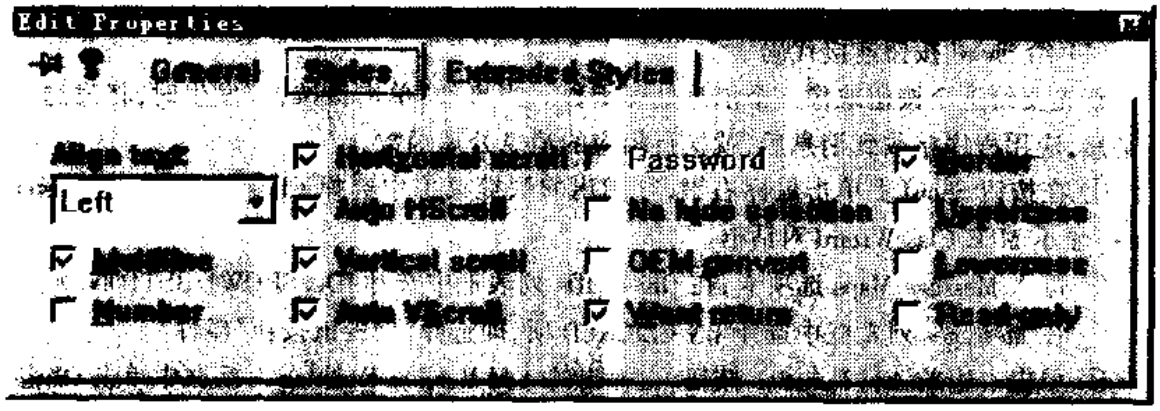


图 11-7　IDC_EDIT1 的属性对话框

表 11-4 CEdit 控件的样式

样式宏	说 明
ES_AUTOHSCROLL	当用户在行尾输入一个字符时，文本自动向右滚动 10 个字符，当用户接受 Enter 键时，控件文本滚动回到零位置
ES_AUTOVSCROLL	当用户在最后一行按 Enter 键时自动将文本向上滚动一页
ES_CENTER	在多行编辑器中将文本置中
ES_LEFT	将文本左对齐
ES_LOWERCASE	在编辑框控件中输入时自动将所有字符转换成小写
ES_MULTILINE	指定编辑框控件是一个多行编辑框控件(默认时为单行)
ES_NOHIDESEL	不做编辑框控件的默认动作，该动作是：当控件失去输入光标时，隐藏所选文本，当控件接受输入光标时，显示所有文本
ES_NUMBER	只允许在编辑框控件中输入数字()只适用于 Windows 95/98()
ES_OEMCONVERT	当应用程序调用 Windows API 函数 AnsiToOem()将一个编辑框控件中的 ANSI 字符转换成 OEM 字符串时，通过将控件中输入的 ANSI 字符集转换成 OEM 字符集再转回到 ANSI，确保适当的字符转换，这种样式在包含文件名的编辑框控件中非常有用
ES_PASSWORD	将编辑框控件中的所有输入的字符以“*”显示，应用程序可以用 SetPasswordChar 方法来设置显示不同字符
ES_READONLY	防止用户在编辑框控件中输入或编辑文本
ES_RIGHT	在单行编辑框控件中文本右对齐
ES_UPPERCASE	将编辑框控件中输入时自动将所有字符转换成大写字符
ES_WANTRETURN	在一个对话框的单行编辑框控件中当用户按 Enter 键时插入回车符(\r)字符；如果不指定此样式，则按 Enter 键时，相当于按对话框的默认按钮。此样式对单行编辑框控件无效

对于编辑框 Edit2，也进行同样的属性修改，然后再次编译、连接和执行一下本应用程序，此时发现在编辑框中可以进行多行文本的输入，按回车键也不会终止应用程序，并且编辑框也有了水平和垂直滚动条。

5. 添加代码

(1) 给 Show1 按钮连接代码

首先要给这个按钮映像一个消息处理函数。当事件“单击该按钮”发生时，将发送一个消息，应用程序接收到消息后，就执行相应的消息处理函数。

① 在按钮 Show1 上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令，Visual C++ 显示 MFC ClassWizard 对话框。

② 进入 Message Maps 选项卡，在 Object IDs 列表框中选择 IDC_SHOW1_BUTTON 项。

③ 在 Messages 列表框中选择 BN_CLICKED 项，即选择“单击该按钮”事件。

④ 单击右边的 Add Function 按钮，在弹出的 Add Member Function 对话框中单击 OK 按钮，接受建议的函数名 OnShow1Button()，如图 11-8 所示，此时在 MFC ClassWizard 对话框的 Member functions 列表框中增加了一个成员函数 OnShow1Button()。

⑤ 单击 MFC ClassWizard 对话框中的 Edit Code 按钮，此时 Visual C++ 显示源代码文

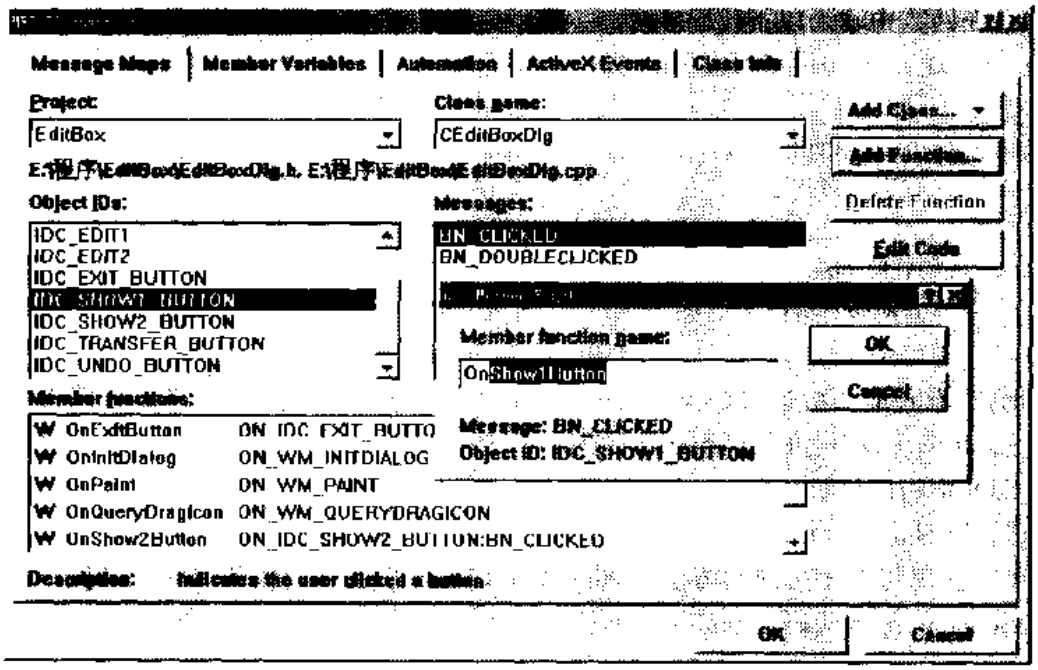


图 11-8　MFC ClassWizard 中的 Message Maps 选项卡

件(.cpp),并将光标停在函数 OnShow1Button()处,等待用户定义函数的内容。

⑥ 在函数中输入以下代码:

```
void CEditBoxDlg::OnShow1Button()
{
    // TODO: 在此处加入自己的控件通知句柄代码
    // 代码编写开始
    m_Edit1.SetSel(0,-1);// 选中编辑框 IDC_EDIT1 中的全部内容
    m_Edit1.ReplaceSel("This is the first EditBox!");
                                        //用新的文件代替原有的文本
    // 代码编写结束
}
```

程序中 SetSel()函数是 CEdit 类中的成员函数,因为 m_Edit1 是属于 CEdit 类的对象,故可以调用该类所有的成员函数。ReplaceSel()函数也是 CEdit 类的成员函数。

(2) 给 Clear1 按钮连接代码

① 在按钮 Clear1 上单击鼠标右键,在弹出的快捷菜单中选择 ClassWizard 命令,显示 MFC ClassWizard 的对话框。

② 在 Object IDs 列表框中选择 IDC_CLEAR1 BUTTON 项。

③ 在 Messages 列表中选择 BN_CLICKED 项,即选择"单击该按钮"事件。

④ 单击右边的 Add Function 按钮,在弹出的窗口中接受建议的函数名 OnClear1Button(),此时,在该对话框的 Member functions 列表框中增加了一个成员函数 OnClear1Button()。

⑤ 单击 ClassWizard 窗口中的 Edit Code 按钮,此时 Visual C++ 显示源代码文件(.cpp),并将光标停在函数 OnClear1Button()处,等待用户定义函数的内容。

⑥ 在函数中输入以下代码：

```
void CEditBoxDlg::OnClear1Button()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    m_Edit1.SetSel(0,-1);//表示选中编辑框 IDC_EDIT1 中的全部内容
    m_Edit1.ReplaceSel("");//用空字符串代替所选中的文本,即把所选的文本删除掉
    // 代码编写结束
}
```

现在，看看前面给这两个按钮编写的代码是否正确。编译、连接并运行该程序，然后单击 Show1 按钮，此时在编辑框 Edit1 中显示“This is the first EditBox!”；单击 Clear1 按钮，编辑框 Edit1 中的内容被清除，由此说明编写的代码是正确的。

(3) 给 Show2 和 Clear2 按钮连接代码

可以使用与上述相同的方法为 Show2 和 Clear2 按钮添加代码，OnShow2Button 和 OnClear2Button 的函数代码分别如下所示。

```
void CEditBoxDlg::OnShow2Button()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    m_Edit2.SetSel(0,-1);//表示选中编辑框 IDC_EDIT2 中的全部内容
    m_Edit2.ReplaceSel("This is the second EditBox!");
                        //用新的文本代替所有原有的文本
    // 代码编写结束
}

void CEditBoxDlg::OnClear2Button()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    m_Edit2.SetSel(0,-1);//表示选中编辑框 IDC_EDIT2 中的全部内容
    m_Edit2.ReplaceSel("");
                    //用空字符串代替所选中的文本,即把所选的文本删除掉
    //代码编写结束
}
```

现在，来检查一下这两个按钮编写的代码是否正确。编译运行该应用程序，单击 Show2 按钮，在编辑框 EDIT2 中显示“This is the second EditBox!”，单击 Clear2 按钮，编辑框 EDIT2 中的内容被清除，说明编写的代码是正确的。

(4) 给按钮 IDC_Transfer_BUTTON 连接代码

按钮 IDC_Transfer_BUTTON 就是界面中显示的 Transfer 按钮，单击这个按钮时把上面编辑框中的内容复制到下面的编辑框中。在这里我们将用到 CEdit 类中实现复制和拷贝

功能的函数。

① 在按钮 IDC_Transfer BUTTON 上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 ObjectIDs 列表框中选择 IDC_Transfer_BUTTON 项。

③ 在 Messages 列表中选择 BN_CLICKED 项。

④ 单击右边的 Add Function 按钮，在弹出的窗口中输入函数名 OnCopyButton()，此时，在 ClassWizard 对话框的 Member functions 列表中增加了一个成员函数 OnCopyButton()处，等待用户定义函数的内容。

⑤ 在函数中输入以下代码：

```
void CEditBoxDlg::OnCopyButton()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    m_Edit1.SetSel(0, 1);
    m_Edit1.Copy();//将 m_Edit1 编辑框中所选的内容拷贝到剪贴板上
    m_Edit2.SetSel(0,-1)
    m_Edit2.ReplaceSel("");
    m_Edit2.Paste();//将剪贴板中的内容粘贴到 m_Edit2 编辑框中
    // 代码编写结束
}
```

程序中 Copy()函数和 Paste()函数是 CEdit 类中的成员函数。m_Edit2.SetSel(0, -1)和 m_Edit2.ReplaceSel("")这两条语句是把 m_Edit2 编辑框中的内容全部清空，Paste()函数是把剪贴板中的内容粘贴到当前光标处，所以首先要把 m_Edit2 编辑框中的内容清空，否则就会粘贴在原有内容的后面

注意，在编写这段代码中，要分清楚调用函数的不同对象。Copy()函数是把 m_Edit1 编辑框中所选的内容拷贝到剪贴板上，故调用该函数的对象是 m_Edit1，而 Paste()函数是把剪贴板中的内容粘贴到 m_Edit2 编辑框中，故调用该函数的对象是 m_Edit2。

(5) 给 Undo 按钮连接代码

下面给 Undo 按钮连接代码：

① 在按钮 Undo 上单击鼠标右键。在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 Object IDs 列表中选择 IDC_UNDO_BUTTON 项。

③ 在 Messages 列表框中选择 BN_CLICKED 项。

④ 单击右边的 Add Function 按钮，在弹出的对话框中接受建议的函数名 OnUndoButton()。此时，在 ClassWizard 对话框的 Member functions 列表框中增加了一个成员函数 OnUndoButton()。

⑤ 单击 MFC ClassWizard 对话框中的 Edit Code 按钮，此时 Visual C++ 显示源代码文件(.cpp)，并将光标停在函数 OnUndoButton()处，等待用户输入函数的内容。

⑥ 在函数中输入以下代码：

```
void CEditBoxDlg::OnUndoButton()
{
```

```
    // TODO:在此加入自己的控件通知句柄代码
    // 编写代码开始
    m_Edit1.Undo();// Undo()函数是 CEdit 类中的成员函数,表示取消编辑框中上一次操作
    m_Edit2.Undo();// 编写代码结束
}
```

(6) 给 Exit 按钮连接代码

① 在按钮 Exit 上单击鼠标右键,在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 Object IDs 列表框中选择 IDC_EXIT_BUTTON 项。

③ 在 Messages 列表框中选择 BN_CLICKED 项。

④ 单击右边的 Add Function 按钮,在弹出的对话框中接受建议的函数名 OnOKButton(),此时,在 ClassWizard 对话框的 Member functions 列表框中增加了一个成员函数 OnOKButton()。

⑤ 单击 ClassWizard 对话框中的 Edit Code 按钮,此时 Visual C ++ 显示源代码文件(.cpp),并将光标停在函数 OnOKButton()处,等待用户定义函数的内容。

⑥ 在函数中输入以下代码:

```
void CEditBoxDlg::OnOKButton()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 编写代码开始
    OnOK();
    // 编写代码结束
}
```

现在,我们已经完成了界面中所有控件的代码连接,编译、连接后,就可以运行该应用程序了。

习 题

11-1 编辑框控件是如何使用的?

11-2 编辑框控件的类结构是如何定义的?

11-3 编辑框控件是如何响应消息的?

11-4 用 MFC 向导创建编辑框,在编辑框中实现算术加、减、乘和除的运算。

11-5 用 MFC 向导创建一个程序,在程序的运行界面上能输入 10 个任意数据,然后对这 10 个数据进行统计计算,如平均值、方差、均方根等。

11-6 建立一个程序进行数据的管理,设有 10 组数据,每一组数据有 5 个元素,要求该程序能实现如下功能:

(1) 从键盘依次输入 10 组数据;

(2) 调出任意的一组数据并显示出来;

(3) 修改任意一组数据或一组数据中的任意一个元素;

(4) 求出任意一组数据中的最大值和最小值。

第 12 章

菜单类及其应用

用户与应用程序的交互通常是从菜单栏中选择命令或用快捷键的形式，因此，菜单和加速器资源在 VC++编程中是很重要的。本章将介绍如何使用 MFC 菜单类设计一个用户自己的菜单栏。对于一个基于对话框的应用程序，Visual C++的 AppWizard 不会在对话框中产生一个菜单。如果要在对话框中显示一个菜单，必须把它作为一个资源，连接到对话框窗口中。

12.1 菜单类简介

12.1.1 菜单

菜单允许用户用一种标准的方法与 Windows 应用程序进行交互；一个样式一致的用户界面对象——菜单栏。

菜单一般存放在资源脚本中，在程序运行时才调用。一个菜单项一般有两个必要的部分，它们分别是字符串名(如 file)和整数标识符。一般来说，菜单的标识符 ID 在资源的头文件(Resource.h)中定义，如：

```
#define IDM_MENU1 129
```

其中，前缀 IDM 指明是菜单项。

在菜单资源创建后，必须将它链接到类中，通过提供类方法来处理每个菜单项。

12.1.2 菜单资源模板

菜单资源模板用一种很简单的结构在资源脚本文件中定义，下面就是本章要讨论的例子的菜单资源模板，各项的说明已同时以注释的形式标出：

```
IDM_MENU1 MENU DISCARDABLE
BEGIN
    POPUP "&File"                          //菜单名
    BEGIN                                  //菜单 File 开始
        MENUITEM "&Voice", IDC_VOICE_MENU  //菜单项
        MENUITEM "&Show", IDC_SHOW_MENU    //菜单项
        MENUITEM SEPARATOR                 //分隔线
        MENUITEM "&Exit", IDC_EXIT_MENU    //菜单项
    END                                    //菜单 File 结束
    POPUP "&Help"                          //定义 Help 菜单
```

```
    BEGIN
        MENUITEM "&About", IDC_ABOUT_MENU          //菜单项
    END                                            //菜单 Help 定义结束
END
```

这个菜单资源模板是在创建菜单资源时由系统自动产生的，内容保存在 Menu.rc 资源文件中，其中 Menu 是本例的工程文件名。从上面的代码可以知道，每一个下拉菜单由 POPUP 定义菜单名，菜单名下的命令选项由 BEGIN...END 组成一组，命令选项的格式是：

MENUITEM "命令名", ID 值

在上例中有两个下拉菜单，菜单名分别为 File 和 Help。

12.1.3 菜单类的结构及其方法

菜单类(CMenu)是从 CObject 类派生的，它在 MFC 类库中的层次位置如图 12-1 所示。

下面是 CMenu 类的定义，从中可大致了解这个类的功能。



图 12-1 CMenu 类在 MFC 类库中的层次位置

```
class CMenu : public CObject

  DECLARE_DYNCREATE(CMenu)
public:

// Constructors
  CMenu();

  BOOL CreateMenu();
  BOOL CreatePopupMenu();
  BOOL LoadMenu(LPCTSTR lpszResourceName);
  BOOL LoadMenu(UINT nIDResource);
  BOOL LoadMenuIndirect(const void* lpMenuTemplate);
  BOOL DestroyMenu();

// Attributes
  HMENU m_hMenu;          // must be first data member
  HMENU GetSafeHmenu() const;
  operator HMENU() const;

  static CMenu* PASCAL FromHandle(HMENU hMenu);
  static void PASCAL DeleteTempMap();
  BOOL Attach(HMENU hMenu);
  HMENU Detach();

// CMenu Operations
  BOOL DeleteMenu(UINT nPosition, UINT nFlags);
```

```
    BOOL TrackPopupMenu(UINT nFlags, int x, int y,
                                        CWnd* pWnd, LPCRECT lpRect = 0);
    BOOL operator==(const CMenu& menu) const;
    BOOL operator!=(const CMenu& menu) const;

// CMenuItem Operations
    BOOL AppendMenu(UINT nFlags,UINT nIDNewItem = 0,
                                    LPCTSTR lpszNewItem = NULL);
    BOOL AppendMenu(UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp);
    UINT CheckMenuItem(UINT nIDCheckItem, UINT nCheck);
    UINT EnableMenuItem(UINT nIDEnableItem, UINT nEnable);
    UINT GetMenuItemCount() const;
    UINT GetMenuItemID(int nPos) const;
    UINT GetMenuState(UINT nID, UINT nFlags) const;
    int GetMenuString(UINT nIDItem, LPTSTR lpString, int nMaxCount,
                                    UINT nFlags) const;
    int GetMenuString(UINT nIDItem, CString& rString, UINT nFlags) const;
    BOOL GetMenuItemInfo(UINT nIDItem, LPMENUITEMINFO lpMenuItemInfo,
                                    BOOL fByPos = FALSE);
    CMenu* GetSubMenu(int nPos) const;
    BOOL InsertMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem = 0,
                                    LPCTSTR lpszNewItem = NULL);
    BOOL InsertMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem,
                                    const CBitmap* pBmp);
    BOOL ModifyMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem = 0,
                                    LPCTSTR lpszNewItem = NULL);
    BOOL ModifyMenu(UINT nPosition, UINT nFlags, UINT nIDNewItem,
                                    const CBitmap* pBmp);
    BOOL RemoveMenu(UINT nPosition, UINT nFlags);
    BOOL SetMenuItemBitmaps(UINT nPosition, UINT nFlags,
                const CBitmap* pBmpUnchecked, const CBitmap* pBmpChecked);
    BOOL CheckMenuRadioItem(UINT nIDFirst, UINT nIDLast, UINT nIDItem, UINT
    nFlags);
    BOOL SetDefaultItem(UINT uItem, BOOL fByPos = FALSE);
    UINT GetDefaultItem(UINT gmdiFlags, BOOL fByPos = FALSE);

// Context Help Functions
    BOOL SetMenuContextHelpId(DWORD dwContextHelpId);
    DWORD GetMenuContextHelpId() const;

// Overridables (must override draw and measure for owner-draw menu items)
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
```

```
    virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);

// Implementation
public:
    virtual ~CMenu();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
    static CMenu* PASCAL CMenu::FromHandlePermanent(HMENU hMenu);
};
```

CMenu 类提供了许多处理菜单和菜单项的方法，这些方法分别是构造方法、菜单操作方法、菜单项操作方法和虚拟方法。

构造方法用来建立 Windows 菜单并在运行时将它们附加到 CMenu 对象上，表 12-1 是 CMenu 的构造方法。

表 12-1 CMenu 的构造方法

方 法	说 明
Attach( )	把一个标准的 Windows 菜单句柄附加到 CMenu 对象上
CreateMenu( )	创建一个空菜单并把它附加到 CMenu 对象上
CreatePopupMenu( )	创建一个弹出式菜单并把它附加到 CMenu 对象上
DeleteTempMap( )	删除由 FromHandle( )构造函数创建的任何临时 CMenu 对象
DestroyMenu( )	去掉附加到 CMenu 对象上的菜单并释放该菜单占有的任何内存
Detach( )	从 CMenu 对象上拆开 Windows 菜单句柄并返回该句柄
FromHandle( )	当给定 Windows 菜单句柄时，返回 CMenu 对象指针
GetSafeHmenu( )	返回由 CMenu 对象封装的菜单句柄成员(m_hMenu)
LoadMenu( )	从可执行文件装入菜单资源并把它附加到 CMenu 对象上
LoadMenuIndirect( )	从内存中的菜单模板中装入菜单并把它附加到 CMenu 对象上

菜单操作方法只有两个，DeleteMenu( )和 TrackPopupMenu( )，用来处理菜单的顶层操作。DeleteMenu( )删除某个特定的菜单项，如果被删除的菜单项有相关的弹出式菜单，此弹出式菜单的句柄也要被删除并释放内存；TrackPopupMenu( )在一个 POINT 结构所指定的位置显示一个快捷菜单。

菜单项操作方法是用来处理实际菜单项的，这些方法是对菜单操作方法的补充。菜单项操作特定的 CMenu 类的方法如表 12-2 所示。

表 12-2 特定于菜单项操作的 CMenu 类的方法

方 法	说 明
AppendMenu( )	把一个新项加到给定的菜单的末端
CheckMenuItem( )	在弹出式菜单中，把一个校验标记放到下一个菜单项或从一个菜单项中取消一个校验标记

续表

方　法	说　明
CheckMenuRadioItem( )	在此组中，把一个单选按钮放到菜单项旁边或从全部其他菜单项里取消一个已存在的单选按钮
EnableMenuItem( )	去掉(添加)一个菜单项
GetMenuContextHelpId( )	检索与菜单结合的帮助上下文 ID
GetMenuItemCount( )	在弹出式或顶层菜单中获得项的成员
GetMenuItemId( )	为设置在指定位置的菜单项获得菜单项标识符
GetMenuState( )	获得指定菜单项的状态或弹出式菜单中的菜单项成员
GetMenuString( )	获得指定菜单项的标记
GetSubMenu( )	获得指向弹出式菜单的指针
InsertMenu( )	在指定位置插入新的菜单项，把其他项向下移
ModifyMenu( )	在指定位置改变已存在的菜单项
RemoveMenu( )	从指定菜单删除与弹出式菜单结合的菜单项
SetMenuContextHelpId( )	设置与菜单有关的帮助上下文 ID
SetMenuItemBitmaps( )	与菜单项有关的指定校验标记位图

虚拟方法使应用程序可以重写它们来提供 CMenu 派生类中自绘菜单，它共有两个方法，即 DrawItem( )和 MeasureItem( )。DrawItem( )在自绘菜单形状改变时由 MFC 调用，MeasureItem( )在自绘菜单项创建时由 MFC 调用来得到菜单的大小。

12.2　菜单类编程实例

【例 12-1】

编写程序，其界面如图 12-2 所示。应用程序的标题为 Application of Menu，其主窗口中

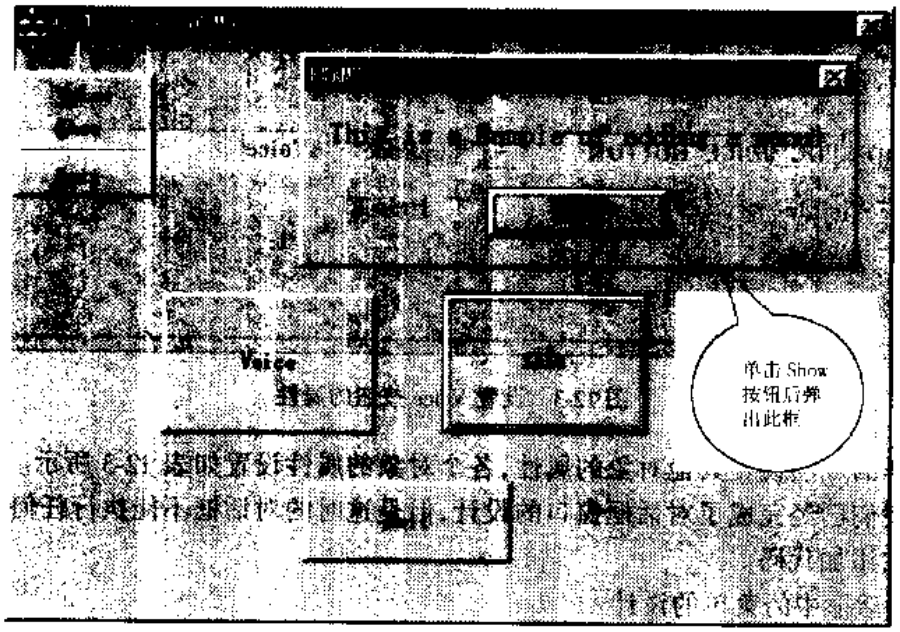


图 12-2　【例 12-1】程序主窗口

有三个按钮，它们分别为 Voice、Show 和 Exit，此外还有一个包含有两个菜单项 File 和 Help 的菜单栏。本应用程序的程序名为 Menu。在窗口中单击 File 菜单，出现一个下拉菜单。单击菜单中的 Voice 选项，可以听到 PC 机喇叭发出蜂鸣声；单击菜单中的 Show 选项，则弹出一个对话框，显示“This is a Sample of adding menu!”。同样，若在窗口中单击相应的按钮，也会有同样的效果。若单击 Help 菜单，也会弹出一个下拉菜单。单击 Exit 按钮，退出本应用程序。

1. 建立应用程序的工程文件

建立 Menu 应用程序工程文件的步骤与上一章内容中建立工程文件的步骤相似，只是在 New 对话框的 Project 选项卡中的 Projects name 编辑框中输入的工程文件的名称为 Menu。

在 MFC AppWizard-Step 2of 6 对话框中（生成工程文件的第二步），在 Please enter a title for your dialog 编辑框中可以给应用程序主窗口命名。这个例子中使用工程文件的名称 Application of Menu，显示在应用程序窗口的标题栏。

2. 应用程序的界面设计

（1）对话框中按钮的设计

AppWizard 已经为 Menu 应用程序生成了工程文件和工程工作区文件，在 Menu 的工作区中选择 ResourceView 选项卡。因为 Menu 应用程序是基于对话框的，所以 AppWizard 生成一个对话框窗口作为应用程序的主窗口，并且给这个对话框取名为 IDD_MENU_DIALOG，现在要做的工作就是在这个对话框窗口中进行界面设计。

在对话框窗口上安排一个 Voice 按钮、一个 Show 按钮和一个 Exit 按钮，按钮的布局参照图 12-2。在处于编辑状态的对话框中，将鼠标移到命令按钮 Voice 上，单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，在弹出的 Menu Item Properties 对话框中设置其属性，如图 12-3 所示。

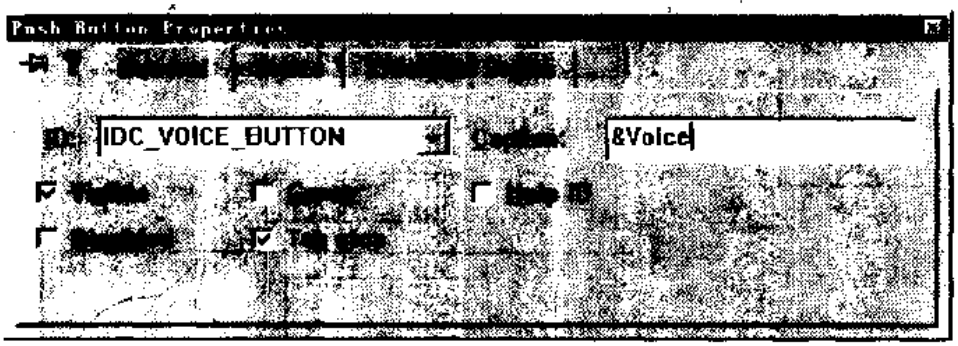


图 12-3 设置 Voice 按钮的属性

用同样的方法设置其他对象的属性，各个对象的属性设置如表 12-3 所示。

现在我们已经完成了对话框窗口的设计，但是这时的对话框不能执行任何操作，因为还没有给它添加代码。

（2）对话框中的菜单的设计

AppWizard 生成的对话框窗口中是没有菜单的，本例需要在应用程序的资源中增加一

个菜单资源。

表 12-3　对话框中各个对象的属性

对　象	ID	Caption
命令按钮	IDC_VOICE_BUTTON	&Voice
命令按钮	IDC_SHOW_BUTTON	&Show
命令按钮	IDC_EXIT_BUTTON	&Exit

① 选择 Insert 菜单中的 Resource 命令，显示 Insert Resource(增加资源)对话框，如图 12-4 所示。选择其中的 Menu 项，使其高亮显示，单击右边的 New 按钮，这样就在资源文件 Menu.rc 中增加了一项资源，即一个菜单。如何才能在 Menu.rc 中看到这个新增的菜单呢？必须打开工作区窗口，选择 View 菜单中的 Workspace 命令，显示出 Application of Menu 应用程序的工作区窗口。

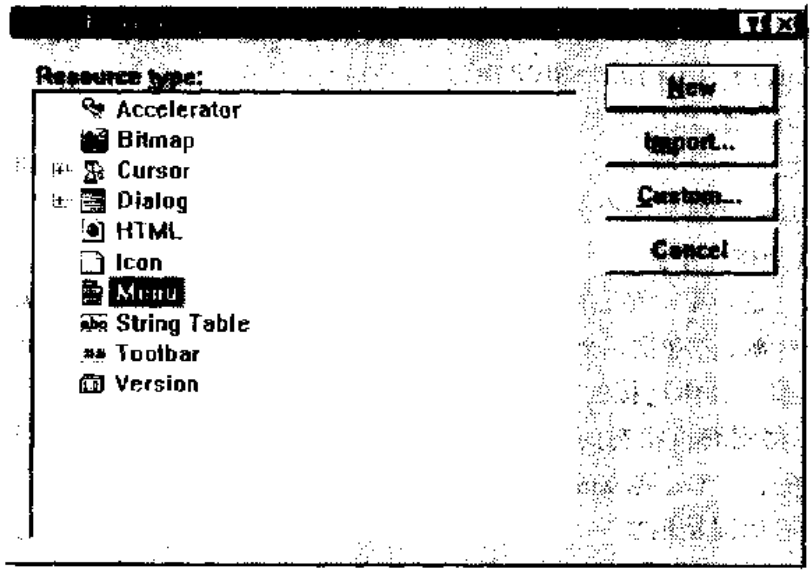


图 12-4　Insert Resource 对话框

② 选择工作区窗口中的 Resource View 选项卡，这时就可以看到新增的菜单资源了。

③ 单击工作区窗口中 Menu 项左边的“+”号，在下面显示出该资源的 ID 号。Visual C++给出新增菜单默认的 ID 为 IDR_MENU1。

④ 双击 ID 号，则显示一个可编辑的菜单栏，该菜单栏的最左边是一个带虚框的矩形框，这个矩形框表示最左边的一个菜单项。

⑤ 双击矩形框，Visual C++显示一个 Menu Item Properties(菜单属性)对话框。设置该菜单项的标题为 &File，如图 12-5 所示。

这时在 File 菜单项的右边和下面各出现一个带虚框的矩形框，右边的虚框可以输入另一个菜单，用同样的方法在右边的虚框中输入 &Help。

下面的虚框用于输入 File 菜单的第一个菜单项的名称。双击 File 菜单标题下的矩形框，此时仍然显示 Menu Item Properties 对话框，在该对话框中设置该菜单项的标题 Caption

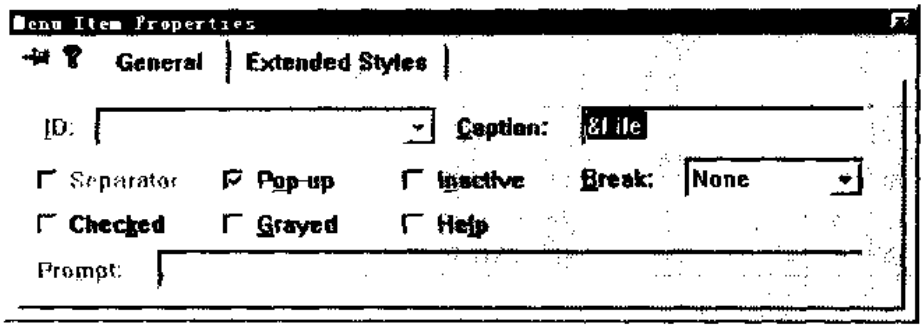


图 12-5 Menu Item Properties 对话框

为 &Voice，设置 ID 为 ID_VOICE_MENU。用同样的方法设置 File 下拉菜单中的 Show 选项的 ID 为 ID_SHOW_MENU，Caption 为 &Show。若要在 Show 下面显示一条分隔线，只要在 Menu Item Properties 对话框中选择 Separator 复选框即可。这条线实际上也是一个菜单项，只不过显示的不是一些字母，而是以分隔线的形式显示出来。接着在分隔线的下面加上一个菜单项 Exit，ID 为 ID_EXIT_MENU，Caption 为 &Exit。

用同样的方法给 Help 菜单添加一个 About 菜单项。

现在我们已经制作好了一个菜单，主菜单有两项：File 和 Help。File 的下拉菜单中有 Voice、Show 和 Exit 三项和一个分隔线；Help 下拉菜单中有 About 项。

3. 给菜单连接一个类

菜单实际上是一个对象，在具体的编程中必须给它连接一个类。因为这个菜单是对话框中的一个资源，它将调用对话框中的一些成员，所以把这个菜单与对话框的派生类 CMenuDlg 相连接。具体连接的步骤如下：

① 将鼠标移到编辑状态的菜单上，单击鼠标右键，在弹出的快捷菜单中选中 ClassWizard 命令。在显示 ClassWizard 对话框的同时，显示出 Adding a Class 对话框，如图 12-6 所示。该对话框询问是增加一个新的类，还是选择一个已经存在的类进行连接。这里要连接的类是与 Menu 应用程序相关联的对话框类 CMenuDlg，这个类已经存在，所以选择

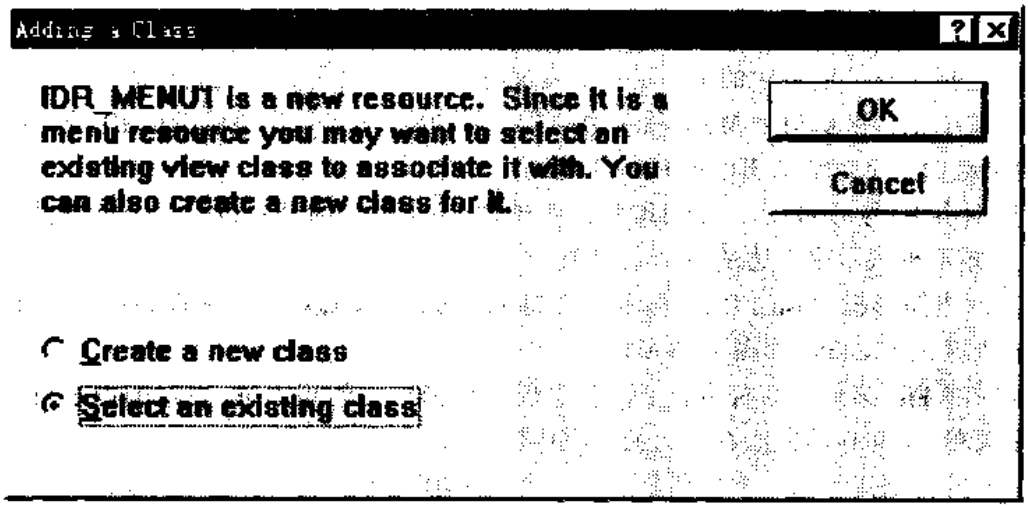


图 12-6 Adding a Class 对话框

Select an existing class 项。

② 单击 OK 按钮后，在 Select Class 对话框中选择 CMenuDlg，这时，Visual C++ 的 ClassWizard 在 CMenuDlg 中增加了几项，即在 Object IDs 列表框中增加了新增菜单资源中几个菜单项的 ID。这样，菜单 IDR_MENU1 就已经和 CMenuDlg 类连接起来了。

4. 连接菜单和应用程序的主窗口

菜单 IDR_MENU1 连接了一个类，这只是表示该菜单可以调用程序中这个类的成员函数和成员变量，此时在应用程序的主窗口中仍然没有显示菜单，这是因为菜单还没有和应用程序主窗口连接起来。下面介绍菜单与应用程序的主窗口对话框(即 IDD_MENU_DIALOG)的连接。

(1) 在 Menu 的工作区的 Resource View 选项卡中，双击 Dialog 文件夹中的 IDD_MENU_DIALOG，此时弹出 IDD_MENU_DIALOG 对话框。

(2) 在对话框的任何一个空白域单击鼠标右键，然后在弹出的快捷菜单中选择 Properties 命令，弹出 Dialog Properties 对话框，如图 12-7 所示。在对话框中的 Menu 组合框中选择 IDR_MENU1。

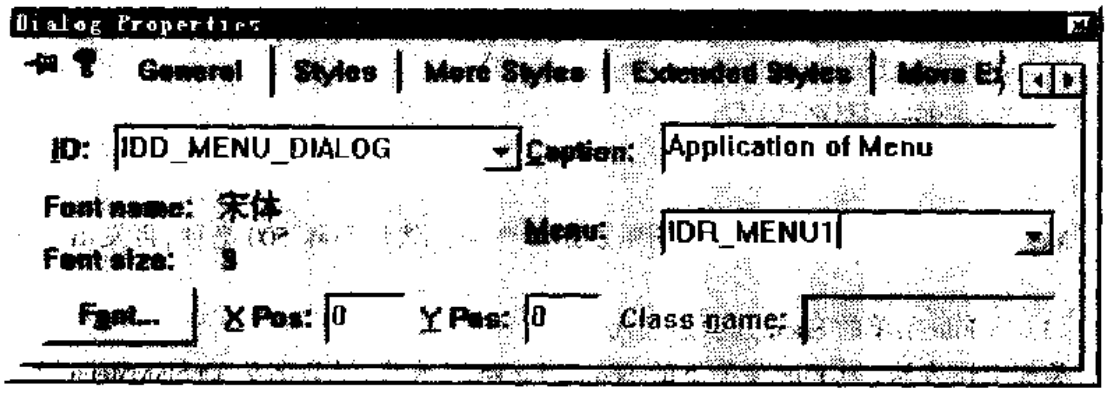


图 12-7　Dialog Properties 对话框

这样菜单 IDR_MENU1 就与应用程序的主窗口 IDD_MENU_DIALOG 建立了连接。

5. 应用程序的代码编程

(1) 给 File 菜单的 Exit 选项编写代码

① 在菜单 IDR_MENU1 上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 Object IDs 列表框中选择 IDC_EXIT_MENU 项。

③ 在 Messages 列表框中选择 COMMAND 项，这里与命令按钮不同，不是 BN_CLICKED。

④ 单击右边的 Add Function 按钮，在弹出的窗口中接受建议函数名 OnExitMenu()。此时，在 ClassWizard 窗口的 Member functions 列表框中增加了一个成员函数 OnExitMenu()。

⑤ 单击 ClassWizard 窗口的 Edit Code 按钮，Visual C++ 显示源代码文件(.cpp)，并将光标停在函数 OnExitMenu()处，等待用户定义函数的内容。

⑥ 在函数 OnExitMenu()中输入以下代码：

```
void CMenuDlg::OnExitMenu()
```

```
    //TODO:在此处添加自己的命令句柄代码
    //代码编写开始
    OnOK();
    //代码编写结束
}
```

因为在前面已经把菜单 IDR_MENU1 与 CMenuDlg 类相关联，OnExitMenu()函数是 CMenuDlg 类的成员函数，所以可以调用 CMenuDlg 的成员函数 OnOK()。

(2) 给 File 菜单的 Voice 选项连接代码

用与上述类似的方法为菜单中的 Voice 选项编写响应代码，函数名为 OnVoiceMenu()。在 OnVoiceMenu()函数中输入以下代码：

```
void CMenuDlg::OnVoiceMenu()
{
    //TODO:在此处添加自己的命令句柄代码
    //代码编写开始
    MessageBeep((WORD)-1);
    DWORD iTime = GetCurrentTime()
    while(GetCurrentTime()) < (iTime + 500));
    MessageBeep((WORD)-1);
    //代码编写结束
}
```

MessageBeep()函数使计算机的喇叭发出蜂鸣声。然后间隔 500 毫秒，再发出一声蜂鸣。

(3) 给 File 菜单的 Show 选项连接代码

类似上述方法，为 File 菜单下的 Show 建立响应函数，函数名为 OnShowMenu()。在 OnShowMenu()函数中输入以下代码：

```
void CMenuDlg::OnShowMenu()
{
    // TODO:在此处添加自己的代码
    // 代码编写开始
    MessageBox("This is sample of adding a menu.");
    //代码编写结束
}
```

这里用一个 MessageBox()函数显示一个信息窗口。

(4) 给 Help 菜单的 About 选项连接代码

为 Help 菜单下的 About 项添加响应函数代码，响应函数的函数名为 OnAboutMenu()。在 OnAboutMenu()函数中输入以下代码：

```
void CMenuDlg::OnAboutMenu()
{
    // TODO:在此处添加自己的命令句柄代码
    // 代码编写开始
```

```
    MessageBox("This is a sample.");
    //代码编写结束
}
```

这里用一个 MessageBox()函数显示一个信息窗口。实际上在 Windows 风格的界面中，选择 About 菜单项都是显示一个模态对话框，在后面的章节中将会详细讲解如何增加一个对话框资源。

(5) 给命令按钮连接代码

应用程序 Menu 的最后一步是给三个命令按钮增加代码。这三个命令按钮的功能与 File 菜单中的三个选项 Voice、Show 和 Exit 的功能完全一样。

首先给这三个命令按钮生成消息处理函数：

① 在对话框 IDD_MENU_DIALOG 上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 Object IDs 列表框中选择 IDC_VOICE_BUTTON 项。

③ 在 Messages 列表框中双击 BN_CLICKED 项。

④ 在弹出的窗口中接受建议的函数名 OnVoiceButton()，此时，在 ClassWizard 窗口的 Member functions 列表中增加了一个成员函数 OnVoiceButton()。

⑤ 在 Object IDs 列表框中选择 IDC_EXIT_BUTTON 项。

⑥ 在 Messages 列表中双击 BN_CLICKED 项。

⑦ 在弹出的窗口中接受建议的函数名 OnShowButton()。

⑧ 同理可以创建 OnExitButton()和 OnVoiceButton()函数，以响应这三个按钮的操作。

OnVioceButton()函数的代码如下：

```
void CMenuDlg:: OnVoiceButton()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    OnVioceMenu();
    // 代码编写开始
}
```

根据要求，Voice 按钮和 Voice 菜单项的功能完全相同，而且 OnVoiceButton()函数和 OnVoiceMenu()函数一样，都是 CMenuDlg 的成员函数，所以在 OnVoiceButton()函数中调用 OnVoiceMenu()函数。

OnShowButton()函数的代码如下：

```
void CMenuDlg::OnShowButton()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    OnShowMenu();
    // 代码编写开始
}
```

根据要求，Show 按钮和 Show 菜单项的功能完全相同，而且 OnShowButton()函数和 OnShowMenu()函数一样，都是 CMenuDlg 的成员函数，所以在 OnShowButton()函数中调用 OnShowMenu()函数。

OnExitButton()函数的代码如下：

```
void CMenuDlg::OnExitButton()
{
    // TODO:在此处加入自己的控件通知句柄代码
    // 代码编写开始
    OnExitMenu();
    // 代码编写开始
}
```

根据要求 Exit 按钮和 Exit 菜单项的功能完全相同，而且 OnExitButton()函数和 OnExitMenu()函数一样，都是 CMenuDlg 的成员函数，所以在 OnExitButton()函数中调用 OnExitMenu()函数。

Menu 应用程序到现在为止，已经全部编写完毕，经过编译运行测试，完全达到设计要求。

习　题

12-1　菜单类的结构是如何定义的？

12-2　如何给菜单连接一个类？

12-3　创建一个包含有“文件”、“编辑”和“计算”三个菜单的应用程序。其中，“文件”菜单包含“打开”、“新建”、“打印”和“退出”等基本功能，“编辑”菜单包含“复制”、“粘贴”和“查找”等功能，“计算”菜单包含“录入数据”、“删除数据”、“修改数据”、“计算求和”和“计算平方和”等功能。而且，在未输入数据时，“计算”菜单中的其他选项不可用。

12-4　创建一个包含有“代码操作”菜单的应用程序。在“代码操作”菜单中包含有“显示 ASCII”、“显示 256 色”等功能，其中选择“显示 ASCII”选项时，在编辑框中显示 0～255 的 ASCII 字符；当选择“显示 256 色”时，在窗口中出现一个填充的矩形框，而且该矩形框的填充色从 0～255 动态依次变化，变化间隔为 0.5 秒。

12-5　创建一个带有“时间”菜单的应用程序。“时间”菜单中包含“年、月、日”、“小时、分、秒”、“万年历”和“退出”选项，其中选择“年、月、日”时，在对话框中显示当时的日期；选择“小时、分、秒”时，在对话框中显示当时的时间；当选择“万年历”时，在对话框中显示指定的日期是星期几，如输入“1999 年 9 月 9 日”，则输出“星期四”。

第 13 章

滚动条类及其应用

滚动条是一个交互式的、高度可视化的控件，它对用户的输入有几种不同的反应。滚动条中包括一个滑块，这个滑块能够沿滚动条运动；在滚动条的两端还有一组按钮。滚动条控件与属于窗口的滚动条是不一样的，处于窗口的滚动条是由该窗口创建、管理和释放的，而滚动条控件是由用户创建、管理和释放的。滚动条在窗口中可以水平或垂直地设置，当单击滚动条两端的按钮箭头时滚动条移动的距离称为滑块的滚动单位，滚动单位可以根据程序的需要进行设置。

作为任何一个窗口的子控件，一个滚动条可以通过通知代码来创建，但也可以用对话框资源模板来创建。一个滚动条向它的主窗口发通知消息，这个主窗口通常是由 CDialog 类派生的，可以通过编写消息映像和消息处理方法来获取和处理这些消息，消息映像和消息处理方法在滚动条的父类中被执行。

13.1 滚动条类简介

13.1.1 滚动条类的结构及其方法

与其他的 MFC 控件类一样，滚动条类 CScrollBar 是 CWnd 的直接派生类，它同时继承了 CWnd 的所有功能。它在 MFC 类库中的层次位置如图 13-1 所示。

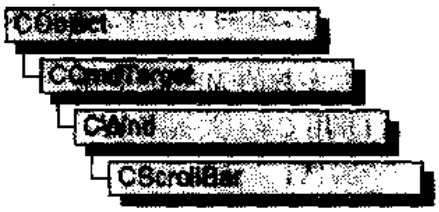


图 13-1 CScrollBar 类在 MFC 类库中的层次位置

VC++ 在 Afxwin.h 中定义了 CScrollBar 类的结构，结构定义如下：

```
class CScrollBar : public CWnd
{
    DECLARE_DYNAMIC(CScrollBar)

    //
    public:
    CScrollBar();          //构造函数
    BOOL Create(DWORD dwStyle, const RECT& rect, CWnd * pParentWnd, UINT nID);

    // 成员函数(类的方法)
    int GetScrollPos() const;
    int SetScrollPos(int nPos, BOOL bRedraw = TRUE);
    void GetScrollRange(LPINT lpMinPos, LPINT lpMaxPos) const;
```

```
    void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE);
    void ShowScrollBar(BOOL bShow = TRUE);
    BOOL EnableScrollBar(UINT nArrowFlags = ESB_ENABLE_BOTH);
    BOOL SetScrollInfo(LPSCROLLINFO lpScrollInfo, BOOL bRedraw = TRUE);
    BOOL GetScrollInfo(LPSCROLLINFO lpScrollInfo, UINT nMask = SIF_ALL);
    int GetScrollLimit();

    // Implementation
    public:
    virtual ~CScrollBar();      //析构函数
};
```

从上述滚动条类的定义中,就可以基本了解CScrollBar类的方法。滚动条类的主要方法的含义如表13-1所示。

表13-1 CScrollBar类的方法

方 法	说 明
EnableScrollBar()	使滚动条的一个或两个箭头有效或无效
GetScrollInfo()	获得滚动条的消息
GetScrollLimit()	获得滚动条的范围
GetScrollPos()	获得滚动条当前的位置
GetScrollRange()	获得制定滚动条的当前最大和最小滚动位置
SetScrollInfo()	设置滚动条的消息
SetScrollPos()	设置滚动块当前的位置
SetScrollRange()	设置制定滚动条的最大和最小滚动位置
ShowScrollBar()	显示或隐藏滚动条

CScrollBar类提供了一组方法用于操纵控件和数据。

13.1.2 创建与初始化滚动条类

像大多数的MFC的对象一样,创建一个CScrollBar对象一般要执行以下步骤:

(1) 用C++关键字new和构造函数CScrollBar::CScrollBar()为一个CScrollBar对象分配一个实例;

(2) 初始化CScrollBar对象,将一个Windows滚动条赋予它,并用CScroll::Create()方法设置参数和样式。

例如,下面的代码分配一个滚动条控件对象并返回指向该对象的指针:

```
CScrollBar::pMyScroll = new CScrollBar;
```

调用CScrollBar::Create()方法初始化指针,方法如下:

```
BOOL Create(DWORD dwStyle,const RECT& rect, CWnd * pParentWnd, UINT nID);
```

其中,参数dwStyle指明滚动条窗口的样式,样式可以是一般窗口的样式;rect表示一个控件的大小和位置,它是一个矩形;pParentWnd是指向控件所属窗口的指针,nID是父窗口用来与滚动条通信的控件标识。

在设置滚动条控件时，还要设置滚动条的范围，设置滚动条范围是通过调用 CScrollBar::SetScrollRange()进行的，比如，设置滚动范围为 - 100 ~ 100 的垂直滚动条，方法为：

```
pMyScroll->SetScrollRange(SB_VERT,-100,100);
```

在设置滚动条的滚动范围后，还必须设置滚动块的当前位置。设置滚动块的当前位置通过函数 SetScrollPos()函数来完成，如果滚动块的位置在 - 100 ~ 100 的中间，即为 0 的位置，可以按如下方法设置：

```
pMyScroll->SetScrollPos(0);
```

13.2　滚动条类编程实例

【例 13-1】

编写一个应用程序，其主窗口如图 13-2 所示。主窗口标题为 Application of ScrollBar。在这个窗口中，有一个滚动条，滚动条下面有一个编辑框，滚动条两边各有两个命令按钮。滚动条的滚动范围设为 0 ~ 20，当前值为 10，滚动条下面的编辑框中显示当前位置的值。单击滚动条向上或向下的箭头按钮，滚动条上的滚动块向上或向下移动一格，编辑框中的数字加 1 或减 1；单击滚动条中滚动块与两端箭头之间的区域，滚动块上移或下移三格，编辑框中的数字加 3 或减 3；按住滚动块，上下拖动，编辑框中的数字随着滚动块的移动而随之发生变化。

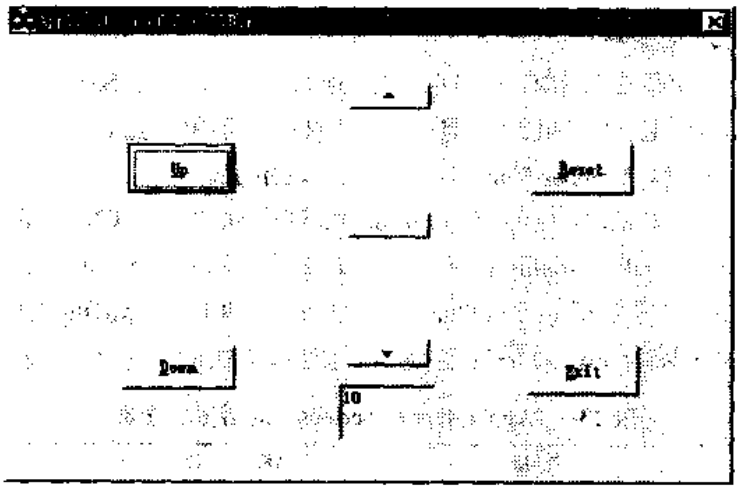


图 13-2　滚动条应用程序主窗口

本例的按钮功能如下：

- Up 按钮：单击 Up 按钮，滚动块移到最上边，编辑框的数字变为 0。
- Down 按钮：单击 Down 按钮，滚动块移到最下边，编辑框的数字变为 20。
- Reset 按钮：单击 Reset 按钮，滚动块移到中间，编辑框的数字变为 10。
- Exit 按钮：单击 Exit 按钮，退出应用程序。

1. 应用程序的可视化编程部分

工程文件的创建类似于前面章节中介绍的方法，假定 AppWizard 已经为 Application of

ScrollBar 应用程序生成了工程文件和工程工作区文件，因为 Application of ScrollBar 应用程序是基于对话框的，所以 AppWizard 生成一个对话框作为应用程序的主窗口，并且给这个对话框取名为 IDD_DIALOG，现在要做的工作就是在这个对话框中进行界面设计。

在控件工具栏中选择相应的控件，根据图 13-2 的要求进行相应控件的布局后，在编辑状态的对话框中，将鼠标移到各个控件上，单击鼠标右键，在弹出的快捷菜单中选择 Properties 命令，在 Properties 对话框中，对各个控件进行属性设置。各个对象属性设置如表 13-2 所示。

表 13-2 对话框中各个对象的属性

对 象	ID	Capttion
滚动条	IDC_SCROLLBAR	无
编辑框	IDC_EDIT1	无
Up 按钮	IDC_UP_BUTTON	&Up
Down 按钮	IDC_DOWN_BUTTON	&Down
Reset 按钮	IDC_RESET_BUTTON	&Reset
Exit 按钮	IDC_EXIT_BUTTON	&Exit

按表 13-1 中的内容，可以完成对话框的设计，但是现在的对话框不能执行任何操作，因为还没有给它添加任何代码。

2. 应用程序的代码编程部分

(1) 给滚动条连接变量

现在，我们给滚动条控件 IDC_SCROLLBAR 连接一个变量 m_ScrollBar。把鼠标移到滚动条 IDC_SCROLLBAR 上，单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令，弹出 ClassWizard 对话框；选择 Member Variables 选项卡，确定 Project 列表中为 SCROLLBAR，确定 Class name 列表框中为 CSCROLLDlg；在 Control IDs 列表框中单击 IDC_SCROLLBAR 项，使之高亮显示；单击右边的 Add Variable 按钮，设置成员变量名为 m_Scrollbar，类别为 Control，变量类型为 CScrollBar，这样就把 m_Scrollbar 变量加到了 CSCROLLBarDlg 类中。

用同样的方法可以为 Edit 对话框连接一个变量，本例的变量表如表 13-3 所示。

表 13-3 Application of ScrollBar 对话框变量表

ID	变量名	类 别	类 型
IDC_SCROLLBAR	m_Scrollbar	Control	CScrollbar
IDC_EDIT1	m_Edit	Control	CEdit

(2) 初始化滚动条

运行本应用程序时，一进入主窗口，滚动条的滚动块应位于中间位置，而且滚动条的最小值和最大值为 0 和 20，编辑框中显示的值是滚动条当前位置的值。

在 ClassWizard 中的 Member functions 列表框中选 OnInitDialog 项，使之高亮显示，单击 Edit Code 按钮对函数 OnInitDialog()进行代码编辑。

```
BOOL CSCROLLBARDlg::OnInitDialog()
{
```

```
CDialog::OnInitDialog();

//将 About 菜单项加入到系统菜单中
//IDM_ABOUTBOX 必须在系统命令范围内
ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
ASSERT(IDM_ABOUTBOX < 0xF000);

CMenu* pSysMenu = GetSystemMenu(FALSE);
if (pSysMenu != NULL)
{
    CString strAboutMenu;
    bstrAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())

        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }

}

//为该对话框设置图标,当该应用程序的主窗口不是一个对话框时,程序将自动设置图标
SetIcon(m_hIcon, TRUE);    // Set big icon
SetIcon(m_hIcon, FALSE);   // Set small icon
// TODO: Add extra initialization here 增加代码开始
m_Scrollbar.SetScrollRange(0,20);
m_Scrollbar.SetScrollPos(10);
char sPos[10];
itoa(m_Scrollbar.GetScrollPos(),sPos,10);
m_Edit.SetSel(0,-1);
m_Edit.ReplaceSel(sPos);
UpdateData(FALSE);
//代码增加结束
return TRUE; // return TRUE unless you set the focus to a control
}
```

在 OnInitDialog()函数中,函数 SetScrollRange()是类 CScrollBar 中的成员函数,用来设置滚动条的滚动范围,本例的范围为 0 ~ 20。在设置了滚动条的范围后,通过函数 m_Scrollbar.SetScrollPos(10)来设置滚动条的当前的位置,函数 SetScrollPos()是类 CScrollBar 中的成员函数,用于设置滚动条的位置。变量 m_Scrollbar 是与滚动条相连接的变量,因为它的类型是 CScrollBar 类,所以可以用它来调用函数 SetScrollPos()。

因为 GetScrollPos()函数得到的当前位置的值是数值型,要把它转换成字符型,才能显示在编辑框中,因此,在 OnInitDialog()函数中有如下两条语句:

```
char sPos[10];
```

```
itoa(m_Scrollbar.GetScrollPos(),sPos,10);
```

其中：sPos 是字符串变量，itoa()函数是把数值型变量转换成字符型的函数。

m_Edit.SetSel(0, -1)和 m_Edit.ReplaceSel(sPos)两个语句是选择编辑框的全部文本，并由 sPos 字符串来代替，代替结束后，要把刚才设置的值传给屏幕，用带 FALSE 参数的 UpDateData()函数来完成：UpdateData(FALSE)。

(3) 给滚动条消息添加代码

在 ClassWizard 中的 Objece IDs 列表框中有一个 WM_VSCROLL 选项，此即为垂直滚动条的消息，为此消息加上一个成员函数 OnVScroll()，这是一个与类 CSCROLLDlg 相对应的消息处理函数。如果对话框中有多个滚动条的话，就要确定不同滚动条所响应的消息。

OnVScroll()函数的响应代码如下：

```
void CSCROLLBARDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
  // TODO: 在此处加入消息句柄代码或调用默认函数
  //代码编写开始
  char sPos[10];
  int iNowPos;
  switch(nSBCode)
  {
     if(pScrollBar == &m_Scrollbar)
     {
       case SB_THUMBTRACK: //拖动滚动滑块时
           m_Scrollbar.SetScrollPos(nPos);
           itoa(nPos,sPos,10);
           m_Edit.SetSel(0, -1);
           m_Edit.ReplaceSel(sPos);
       case SB_LINEDOWN:    //单击滚动条向下的箭头
           iNowPos = m_Scrollbar.GetScrollPos();
           iNowPos = iNowPos + 1;
           if(iNowPos > 20)
              iNowPos = 20;
           m_Scrollbar.SetScrollPos(iNowPos);
           itoa(m_Scrollbar.GetScrollPos(),sPos,10);
           m_Edit.SetSel(0, -1);
           m_Edit.ReplaceSel(sPos);
           break;
      case SB_LINEUP:    //单击滚动条向上的箭头
           iNowPos = m_Scrollbar.GetScrollPos();
           iNowPos = iNowPos - 1;
           if(iNowPos < 0)
              iNowPos = 0;
           m_Scrollbar.SetScrollPos(iNowPos);
```

```
            itoa(m_Scrollbar.GetScrollPos(),sPos,10);
            m_Edit.SetSel(0, -1);
            m_Edit.ReplaceSel(sPos);
            break;
        case SB_PAGEDOWN:     //单击滚动条下面的箭头与滚动块之间的区域
            iNowPos = m_Scrollbar.GetScrollPos();
            iNowPos = iNowPos + 3;
            if(iNowPos > 20)
               iNowPos = 20;
            m_Scrollbar.SetScrollPos(iNowPos);
            itoa(m_Scrollbar.GetScrollPos(),sPos,10);
            m_Edit.SetSel(0, -1);
            m_Edit.ReplaceSel(sPos);
            break;
        case SB_PAGEUP:     //单击滚动条上面的箭头与滚动块之间的区域
            iNowPos = m_Scrollbar.GetScrollPos();
            iNowPos = iNowPos-3;
            if(iNowPos < 0)
               iNowPos = 0;
            m_Scrollbar.SetScrollPos(iNowPos);
            itoa(m_Scrollbar.GetScrollPos(),sPos,10);
            m_Edit.SetSel(0, -1);
            m_Edit.ReplaceSel(sPos);
            break;
        }
    }
    //代码编写结束
    CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

函数 OnVScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)有三个参数，第一个参数 nSBCode 表示滚动条发生的是哪一件事情，如单击上边箭头，或者单击下边箭头等；第二个参数 nPos 表示当前滚动块在滚动条中的位置；第三个参数 pScrollBar 表示与事件相关联的是哪一个滚动条。

MFC 传递给 OnVScroll()方法(或水平滚动条方法 OnVScroll())的可能的通知代码请参见第 8 章的有关内容。

滚动条本身没有与之相对应的事件，对滚动条的操作是通过对话框的 WM_HSCROLL 事件，即在 OnVScroll()函数中编写代码。那么，若对话框中有几个滚动条的话，如何确定是对哪个滚动条进行操作呢？这就需要通过一个条件判断语句实现，在上面的代码中有一句 if 语句：

```
void CSCROLLBARDlg::OnVScroll(UINT nSBCode,nPos,CScrollBar * pSCROLLBar)
{
```

```
    ...
    ...
    if(pScrollBar == &m_Scrollbar)
    {
        ...
        ...
    }
}
```

这句代码是判断函数 OnVScroll()的第三个参数是否为滚动条的对象名 m_Scrollbar，如果判断为真，则执行的事件与该滚动条相关。

当拖动滚动块时，参数 nSBCode 的值为 CB_THUMBTRACK，它的代码如下：

```
case SB_THUMBTRACK:     //拖动滚动滑块时
m_Scrollbar.SetScrollPos(nPos);
itoa(nPos,sPos,10);
m_Edit.SetSel(0,-1);
m_Edit.ReplaceSel(sPos);
```

其中代码 m_Scrollbar.SetScrollPos(nPos)是用来设置滚动块的位置。由于 nPos 是整型参数，用以表示滚动块的位置，而在 Edit1 编辑框中显示的内容却是字符型参数，因此，通过 itoa(nPos,sPos,10)语句将滚动块所在位置的数值型变量转换为字符型，以便在编辑框中显示出来。

在处理 SB_LINEDOWN 消息过程中，使用了如下的方法：

```
case SB_LINEDOWN:     //单击滚动条向下的箭头
    iNowPos = m_Scrollbar.GetScrollPos();
    iNowPos = iNowPos + 1;
    if(iNowPos > 20)
       iNowPos = 20;
    m_Scrollbar.SetScrollPos(iNowPos);
    itoa(m_Scrollbar.GetScrollPos(),sPos,10);
    m_Edit.SetSel(0, -1);
    m_Edit.ReplaceSel(sPos);
    break;
```

该语句段首先调用函数 GetScrollPos()得到滑块的当前位置，然后使滑块的当前位置以 1 递增，并用

```
if(iNowPos > 20)
    iNowPos = 20;
```

两条语句保证最大值不超过 20。

SB_LINEUP 是用户单击滚动条的向上箭头时传递的参数，当用户单击向上箭头时，滑块当前位置以 1 递减，并用 if 语句保证最小值不小于 0，它的代码与上面的代码类似，只是将语句：

```
iNowPos = iNowPos + 1;
```

改为：

```
iNowPos = iNowPos - 1;
```

并且用 if 语句保证滚动块的位置最小为 0。

SB_PAGEDOWN 消息是在用户单击滚动条向下箭头与滚动块之间的区域时传递的参数。实际上，本例要求一次单击，滑块位置以 3 递增，该递增通过

```
iNowPos = iNowPos + 3;
```

语句来实现。

SB_PAGEUP 与 SB_PAGEDOWN 的区别则是滚动块的位置以 3 递减，通过语句

```
iNowPos = iNowPos-3;
```

来实现。

(4) 给 Exit 按钮添加代码

按钮 Exit 的实现方法为 OnExitButton()，该方法的代码如下：

```
void CSELECTBOXDlg::OnExitButton()
{
    // 在此处加入控件通知句柄代码
    //编写代码开始
    OnOK();
    //编写代码结束
}
```

(5) 给 Up 按钮添加代码

按钮 Up 的实现方法为 OnUpButton()，该方法的代码如下：

```
void CSCROLLBARDlg::OnUpButton()
{
    // TODO:在此处加入控件通知句柄代码
    //代码编写开始
    m_Scrollbar.SetScrollPos(0);
    m_Edit.SetSel(0,-1);
    m_Edit.ReplaceSel("0");
    //代码编写结束
}
```

代码中 m_Edit.SetSel(0, -1)表示选中编辑框中的所有字符，并通过 m_Edit.ReplaceSel("0")语句用 0 代替编辑框中的所有内容。

(6) 给 Down 按钮添加代码

按钮 Down 的方法名称为 OnDownButton()，该方法的代码如下：

```
void CSCROLLBARDlg::OnDownButton()
{
    // TODO: 在此处加入通知句柄代码
    //代码编写开始
    m_Scrollbar.SetScrollPos(20);
    m_Edit.SetSel(0, 1);
    m_Edit.ReplaceSel("20");
    //代码编写结束
```

代码中 m_Edit.SetSel(0, -1)表示选中编辑框中的所有字符，并通过 m_Edit.ReplaceSel("20")语句用 20 代替编辑框中的所有内容。

(7) 给 Reset 按钮添加代码

按钮 Reset 的方法名称为 OnResetButton()，该方法的实现代码如下：

```
void CSCROLLBARDlg::OnResetButton()
{
    // TODO: 在此处加入通知句柄代码
    //代码编写开始
    m_Scrollbar.SetScrollPos(10);
    m_Edit.SetSel(0,-1);
    m_Edit.ReplaceSel("10");
    //代码编写结束
}
```

通过 m_Scrollbar.SetScrollPos(10)语句来设置滑块的当前位置，此时，通过 m_Edit.SetSel(0, -1)语句选中编辑框中的内容，并通过 m_Edit.ReplaceSel("10")语句用 10 代替编辑框中的所有内容，即复位到中间位置。

(8) 将编辑框设置为只读

在 Application of SELECTINGBOX 应用程序中，编辑框只是显示滚动块的位置，不需要进行编辑，因此必须将其属性改为只读。修改属性的方法是在编辑框 m_Edit 上单击鼠标右键，在弹出的快捷菜单中选择 Properties，并在 Edit Properties 对话框的 Styles 选项卡中选中 Read Only 复选框，如图 13-3 所示。

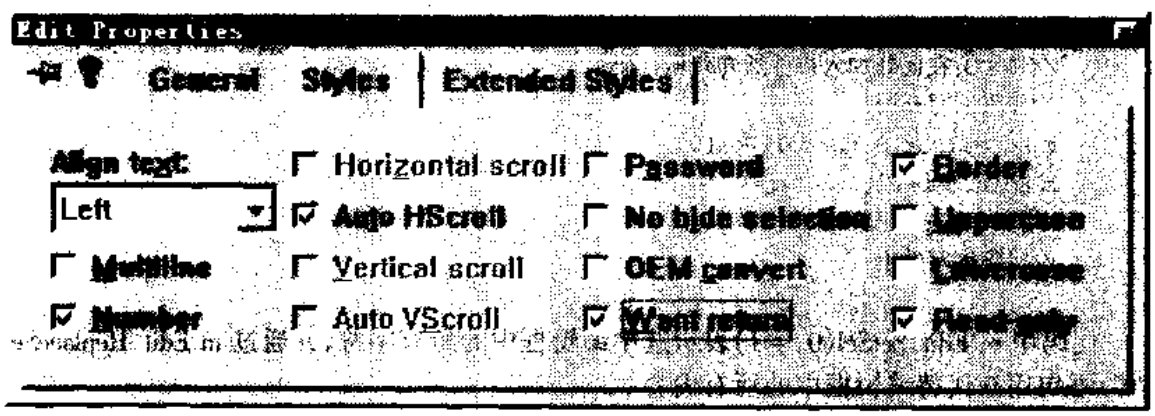


图 13-3　修改属性的操作

进行完以上步骤后，编译并运行即可。

习　题

13-1　滚动条分为几类？其类结构是如何定义的？

13-2　滚动条控件是如何进行消息传递的？

13-3　创建三个水平滚动条，分别用来控制红、绿、蓝三种基本颜色的变化，并在编辑框中显示当时 RGB 的值，变化的颜色效果在一个椭圆中以填充椭圆的方式表现出来。

13-4　创建两个水平滚动条，用来控制 X 和 Y 数值的变化，X 和 Y 的变化范围均为 0～100，X 和 Y 当时的值分别在一个编辑框中显示出来，然后在“计算”菜单中选择“求和”或“求差”，计算结果在窗口的另一个编辑框中显示出来。

13-5　创建一个进度条，用来显示一个文件读入的进度。

第 14 章

按钮类与列表框类及其应用

8.3 和 8.6 节中分别介绍了按钮控件和列表框控件，本章介绍分别封装了这两个控件的按钮类和列 Q 表框类。

14.1 按钮类与列表框类简介

14.1.1 按钮类的结构及其方法

按钮是一个矩形窗口，通常包含一个标题用来说明按钮的作用。按钮控件类是从 CWnd 类继承下来的，它在 MFC 窗口类中的位置如图 14-1 所示。



图 14-1 CButton 类在 MFC 窗口类的层次结构中的位置

VC 在 Afxwin.h 头文件中定义了 CButton 类的结构，结构代码如下：

```
class CButton : public CWnd
{
    DECLARE_DYNAMIC(CButton)

// Constructors
public:
    CButton();
    BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle,
                const RECT& rect, CWnd* pParentWnd, UINT nID);

// Attributes
    UINT GetState() const;
    void SetState(BOOL bHighlight);
    int GetCheck() const;
    void SetCheck(int nCheck);
    UINT GetButtonStyle() const;
    void SetButtonStyle(UINT nStyle, BOOL bRedraw = TRUE);

#if (WINVER >= 0x400)
    HICON SetIcon(HICON hIcon);
```

```
    HICON GetIcon() const;
    HBITMAP SetBitmap(HBITMAP hBitmap);
    HBITMAP GetBitmap() const;
    HCURSOR SetCursor(HCURSOR hCursor);
    HCURSOR GetCursor();
#endif

// Overridables (for owner draw only)
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);

// Implementation
public:
    virtual ~CButton();
protected:
    virtual BOOL OnChildNotify(UINT, WPARAM, LPARAM, LRESULT*);
};
```

像所有的窗口一样，按钮也可以有窗口样式组合，因为按钮就是窗口，它们可以使用任意组合的 CWnd 样式。除了一般的窗口样式外，按钮还可以使用特殊的按钮格式（在 WINUSER.H 中定义），表 14-1 就是 CButton 类中按钮特有的样式。

表 14-1　CButton 类中按钮特有的样式

样式宏	说　明
BS_3STATE	此样式为复选框，但是方框可以被选中，也可以变暗（表明复选框无效）
BS_AUTO3STATE	此样式为复选框，但是方框可以被选中，也可以变暗（表明复选框无效），当用户选择方框时，它的选中状态会自动转换
BS_AUTOCHECKBOX	此样式为复选框，当用户选择方框时，它的选中状态会自动转换
BS_AUTORADIOBUTTON	此样式为单选按钮，当用户选中圆圈时，按钮自动显示自己被选中的状态，同时使其他具有相同样式的同组的单选按钮处于不被选中的状态
BS_CHECKBOX	创建一个复选框，默认情况下它的右边有标题，当与 BS_LEFTTEXT 一起使用时，标题在左边
BS_DEFPUSHBUTTON	创建一个下压式按钮
BS_GROUPBOX	创建一个带标题的分组框
BE_LEFTTEXT	当和一个单选按钮或复选按钮一起使用时，使得按钮文字出现在按钮的左边
BS_OWNERDRAW	创建自定义按钮，当按钮外形被改变时，MFC 自动调用 DrawItem() 方法，在使用 CBitmapButton 类时，这种样式必须被指定
BS_PUSHBUTTON	创建一个下压式按钮，用户单击此按钮时，向系统发送 WM_COMMAND 消息
BS_RADIOBUTTON	创建单选按钮

用于按钮控件的消息映像有 ON_BN_CLICKED、ON_BN_DBLCLICKED 和 ON_COMMAND，其含义分别为单击按钮发送消息、双击按钮发送消息和单击按钮时发送，ON_COM-

MAND 与 ON_BN_CLICKED 类似

此外，MFC 还提供了一个 CBitmapButton 类，允许用户以图标的方式显示按钮，它是在 CButton 类下派生的，其层次结构如图 14-2 所示。

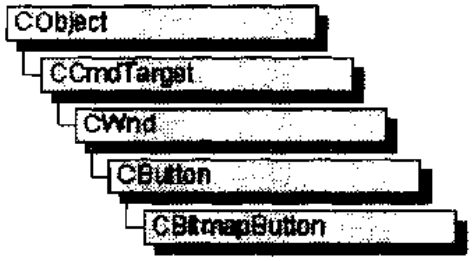


图 14-2 CBitmapButton 在 MFC 中的层次位置

CBitmapButton 类的定义如下：

```
class CBitmapButton : public CButton
{
    DECLARE_DYNAMIC(CBitmapButton)
    public:
    // Construction
    CBitmapButton();
    BOOL LoadBitmaps
            (LPCTSTR lpszBitmapResource,
            LPCTSTR lpszBitmapResourceSel = NULL,
            LPCTSTR lpszBitmapResourceFocus = NULL,
            LPCTSTR lpszBitmapResourceDisabled = NULL);
    BOOL LoadBitmaps(UINT nIDBitmapResource,
            UINT nIDBitmapResourceSel = 0,
            UINT nIDBitmapResourceFocus = 0,
            UINT nIDBitmapResourceDisabled = 0);
    BOOL AutoLoad(UINT nID, CWnd* pParent);

    // Operations
    void SizeToContent();

    // Implementation:
    public:
    #ifdef DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
    #endif
    protected:
    // all bitmaps must be the same size
    CBitmap m_bitmap;              // normal image (REQUIRED)
    CBitmap m_bitmapSel;           // selected image (OPTIONAL)
    CBitmap m_bitmapFocus;         // focused but not selected (OPTIONAL)
    CBitmap m_bitmapDisabled;      // disabled bitmap (OPTIONAL)

    virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
};
```

CButton 类提供了如下的访问方法，如表 14-2 所示。

CBitmapButton 类定义了两个初始化方法以增强一个标准下压按钮的功能，并且提供

了装载位图的简便方法，它们分别为 LoadBitmaps()和 AutoLoad()，MFC 调用方法 DrawItem()自动在一个按钮的用户区上画上位图，即用户可以自定义按钮。LoadBitmaps()方法为一个 CBitmapButton 对象附上位图，最多可以有 4 个位图，这些位图从用于程序的资源文件中读取；用户也可以使用 AutoLoad()方法将一个对话框按钮和一个 CBitmapButton()对象联系起来，有时必须调整位图的尺寸大小，可以通过 SizeToContent()方法进行调整。

表 14-2　类 CButton 的方法

方　法	说　明
GetBitmap()	获得用 SetBitmap()方法设置的位图的句柄
GetButtonStyle()	获得有关按钮控件样式的信息
GetCheck()	获得一个按钮控件选中的状态
GetCursor()	获得通过 SetCursor()方法设置的光标图像的句柄
GetIcon()	获得由 SetIcon()方法设置的图标的句柄
GetState()	获得一个按钮控件的选中、选择和聚焦状态
SetBitmap()	指定按钮上显示的位图
SetButtonStyle()	改变按钮样式
SetCheck()	设置一个按钮控件的选中状态
SetCursor()	指定一个按钮控件上的光标图像
SetIcon()	指定一个按钮上显示的图标
SetState()	设置一个按钮控件的选择状态

复选按钮控件所支持的选项只有两种状态，常用于只有两种完全相反状态的情况下；单选按钮适用于在同一组属性相同的数据中选一个数据；下压按钮适用于消息的发送。

分组框实际上没有太多的操作，只是用于在窗口中对特定的区域划分范围，并把相同属性的数据划并在一起，使窗口简洁明了。

14.1.2　列表框类的结构及其方法

对于存在若干数据项并要从中进行选择的情况，一个方便的方法是使用列表框。列表框是一个矩形窗口，在矩形窗口中包含一些列的字符串，也可以包含其他的数据元素。列表框允许用户在列表框中选择一项或多项，因此有两种样式的列表框，即单选项列表框和多选项列表框，而且列表框可以自带滚动条。单选项列表框只允许用户一次选择一项，而多选项列表框则可以一次选择多项。CListBox 类在 MFC 类层次结构中的位置如图 14-3 所示。



图 14-3　CListBox 类在 MFC 类层次结构中的位置

列表框经常用在对话框里，如用列表框选择文件名、目录等。列表框有一个预定义的键盘接口，用户可以用键盘上的箭头和 PageUp 或 PageDown 键在列表框中进行数据的选择，或通过适当的样式设置，允许与 Shift 或 Ctrl 键组合使用。

1. 列表框类的结构

MFC定义了列表框类的结构，其结构如下：

```
class CListBox : public CWnd

    DECLARE_DYNAMIC(CListBox)

// Constructors
public:
    CListBox();
    BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);

// Attributes

    // for entire listbox
    int GetCount() const;
    int GetHorizontalExtent() const;
    void SetHorizontalExtent(int cxExtent);
    int GetTopIndex() const;
    int SetTopIndex(int nIndex);
    LCID GetLocale() const;
    LCID SetLocale(LCID nNewLocale);
#if (WINVER >= 0x400)
    int InitStorage(int nItems, UINT nBytes);
    UINT ItemFromPoint(CPoint pt, BOOL& bOutside) const;
#endif

    // for single-selection listboxes
    int GetCurSel() const;
    int SetCurSel(int nSelect);

    // for multiple-selection listboxes
    int GetSel(int nIndex) const;     // also works for single-selection
    int SetSel(int nIndex, BOOL bSelect = TRUE);
    int GetSelCount() const;
    int GetSelItems(int nMaxItems, LPINT rgIndex) const;
    void SetAnchorIndex(int nIndex);
    int GetAnchorIndex() const;

    // for listbox items
    DWORD GetItemData(int nIndex) const;
    int SetItemData(int nIndex, DWORD dwItemData);
    void* GetItemDataPtr(int nIndex) const;
```

```
    int SetItemDataPtr(int nIndex, void * pData);
    int GetItemRect(int nIndex, LPRECT lpRect) const;
    int GetText(int nIndex, LPCSTR lpszBuffer) const;
    void GetText(int nIndex, CString& rString) const;
    int GetTextLen(int nIndex) const;
    // Settable only attributes
    void SetColumnWidth(int cxWidth);
    BOOL SetTabStops(int nTabStops, LPINT rgTabStops);
    void SetTabStops();
    BOOL SetTabStops(const int& cxEachStop);    // takes an 'int'

    int SetItemHeight(int nIndex, UINT cyItemHeight);
    int GetItemHeight(int nIndex) const;
    int FindStringExact(int nIndexStart, LPCTSTR lpszFind) const;
    int GetCaretIndex() const;
    int SetCaretIndex(int nIndex, BOOL bScroll = TRUE);

// Operations
    // manipulating listbox items
    int AddString(LPCTSTR lpszItem);
    int DeleteString(UINT nIndex);
    int InsertString(int nIndex, LPCTSTR lpszItem);
    void ResetContent();
    int Dir(UINT attr, LPCTSTR lpszWildCard);

    // selection helpers
    int FindString(int nStartAfter, LPCTSTR lpszItem) const;
    int SelectString(int nStartAfter, LPCTSTR lpszItem);
    int SelItemRange(BOOL bSelect, int nFirstItem, int nLastItem);

// Overridables (must override draw, measure and compare for owner draw)
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
    virtual int CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
    virtual void DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
    virtual int VKeyToItem(UINT nKey, UINT nIndex);
    virtual int CharToItem(UINT nKey, UINT nIndex);

// Implementation
public:
    virtual ~CListBox();
protected:
```

```
virtual BOOL OnChildNotify(UINT, WPARAM, LPARAM, LRESULT *);
:
```

像所有的窗口一样，列表框也有窗口样式的组合，由于它们本身是窗口，因此，除可用窗口样式外，还可以使用如表 14-3 所示的样式的组合。

从上面的 CListBox 类的定义中可以看出，MFC 将标准 Windows 列表框消息（如 LBS_INSERTSTRING 或 LBS_ADDSTRING）封装入 CListBox 类方法中，MFC 程序只需处理通知消息。具有 LBS_NOTIFY 样式的列表框向它的所有者发送通知消息，它的所有者通常是一个 CDialog 派生的类，可以通过对每个消息编写消息映像项和消息处理方法来捕获和处理这些消息。消息映像项和处理方法都是在列表框自己的类中执行。

表 14-3 列表框控件可用的样式

样 式	说 明
LBS_DISABLENOSCROLL	当列表框不需要滚动条时，滚动条无效
LBS_EXTENDSEL	允许使用鼠标及特殊键组合进行多项选择
LBS_HASSTRINGS	指明一个自绘的列表框，其中包括字符串选项，列表框负责为字符串分配内存，指定项的文字可以用 GetText() 方法检索
LBS_MULTICOLUMN	指明一个多列列表框，它含有一个水平滚动条，可以用 SetColumnWidth() 方法设置列的宽度
LBS_MULTIPLESEL	用户通过单击或双击一项进行选择或取消选择
LBS_NOINTEGRALHEIGHT	将列表框设置为创建时指定的大小
LBS_NOREDRAW	列表框在变化时不重绘，用户可以在任何时候发送 WM_SETREDRAW 消息改变这种模式
LBS_NOSEL	指明列表框包含只能看不能选择的项
LBS_NOTIFY	当用户单击或双击时向父窗口发送消息
LBS_OWNERDRAWFIXED	指明列表框的所有者负责填写列表项，且列表框具有相同的高度
LBS_OWNERDRAWVARIABLE	指明列表框的所有者负责填写列表项，且列表框可以有不同高度
LBS_SORT	列表项按字母顺序排列
LBS_STANDARD	此样式是 LBS_NOTIFY、LBS_SORT、WS_VSCROLL 和 WS_BORDER 的组合
LBS_USETABSTOPS	告知列表框在加入字符串列表项时加入 tab 字符
LBS_WANTKEYBOARDINPUT	允许应用程序通过发送 WM_VKEYTOITEM 和 WM_CHARTOITEM 消息给列表框的所有者来处理键盘输入

消息映像项使用下面的基本格式：

```
ON-Message(ControlID,ClassMethod);
```

在这个语法中，Message 是要处理的消息，ControlID 是发送消息的列表框控件的子窗口标识，ClassMethod 是处理消息的父类方法的名字。

所有者类声明中的方法原型如下：

```
afx_msg void ClassMethod();
```

表 14-4 显示了消息映像项，它用于处理列表框通知。

表 14-4　CListBox 消息的消息映像项

消息映像项	说　明
ON-LBN-DBLCLK	当用户双击选项时具有 LBS NOTIFY 样式的列表框向所有者发送此消息
ON LBN-ERRSPACE	列表框不能分配足够内存以满足要求
ON-LBN-KILLFOCUS	当列表框失去输入焦点时出现此消息
ON-LBN-SELCANCEL	当取消当前列表框选择时，具有 LBS NOTIFY 样式的列表框向所有者发送此消息
ON-LBN-SELCHANGE	当列表框中的选择改变时，具有 LBS-NOTIFY 样式的列表框向它的父窗口发送此通知。如果选择是用 CListBox::SetCurSel()类方法改变的，则不发送通知。对多项选择列表框来说，当用户按箭头键时，即使选择不变也发送此通知。

2. 列表框类的方法

正如 CListBox 类声明中所见，CListBox 类为处理和操纵列表框和列表框数据提供了许多方法，这些方法可分为通用方法、单选列表框方法、多选列表框方法、特定字符串方法和虚拟方法等几种。

（1）通用方法

通用方法用来获得和设置列表框数据的值和属性。所有的 CListBox 列表框都有这些方法，包括单选列表框、多选列表框和自绘列表框等。表 14-5 给出了每个通用方法的简单描述。

表 14-5　CListBox 类的通用方法

方　法	描　述
GetCount()	获得列表框中列表项数目
GetHorizontalExtent()	获得列表框的水平滚动宽度(按像素)
GetItemData()	获得列表框项有关的 32 位值
GetItemDataPtr()	获得指向列表框项的指针
GetItemHeight()	获得列表框中项的高度
GetItemRect()	获得列表框项边界矩形
GetLocale()	获得列表框的位置局部标识(LCID)
GetSel()	确定列表框项的选择状态
GetText()	把列表框中字符串复制到缓冲区
GetTextLen()	返回列表框字符串的长度(按字节)
GetTopIndex()	获得列表框中第一个可见项的下标(基于 0)
ItemFromPoint()	确定和返回离某点最近的列表框项的下标
SetColumnWidth()	设置多列列表框的列宽度
SetHorizontalExtent()	设置列表框的水平滚动宽度(按像素)
SetItemData()	设置与一列表框项有关的 32 位值
SetDataPtr()	设置一列表框项的指针
SetItemHeight ()	设置列表框中项的高度
SetLocale()	设置列表框的位置标识(LCID)
SetTabStops()	设置列表框的制表位(Tab-Stop)位置
SetTopIndex()	设置列表框中第一个可见项的下标(基于 0)

注意：方法 GetLocale()和 SetLocale()都与处理位置有关，但什么是位置呢？位置是一

值表,这些值代表了与语言有关的用户首选项信息。Windows 98 在安装时设置了默认位置,每个位置都有唯一的 32 位位置标识(LCID)。用户可以在控制面板(Control Panel)的 International applet 中修改 LCID。

(2) 特定于单项选择的方法

列表框的默认模式是单选项模式;所有的通用方法均适用于单选项列表框。只有两个类方法专门处理单选项列表框:GetCurSel()和 SetCurSel()。从它们的名字可以看出,GetCurSel()方法获得当前被选择的列表框项的下标(基于 0)。相反 SetCurSel()方法是选择列表框项。

(3) 特定于多项选择的方法

多选项列表框扩展了标准单项选择列表框的能力,特定于多项选择的方法可以解决在一个列表框中选择多项带来的复杂性。这些方法作为通用方法的补充在表 14-6 中描述。

表 14-6 特定于多项选择列表框的 CListBox 类方法

方 法	说 明
GetAnchorIndex()	获得多项选择列表框中当前定位项的下标
GetCaretIndex()	获得多项选择列表框中具有光标矩形的项的下标
GetSelCount()	获得多项选择列表框中当前所选的项的数目
GetSelItems()	将所有当前被选列表框项下标放入一整型数组缓冲区
SelItemRange()	切换多选择列表框项范围的选择状态
SetAnchorIndex()	在多项选择列表框中扩充选择设置开始(定位)项
SetCaretIndex()	在多项选择列表框中指定下标项设置光标矩形
SetSel()	在多项选择列表框中切换项目的选择状态

(4) 特定于字符串的方法

特定于字符串的方法适用于单选择和多选择两种模式的列表框,它们处理列表框中的字符串项。表 14-7 列出了用于 CListBox 对象的字符串方法。

表 14-7 CListBox 类中字符串特定的方法

方 法	说 明
AddString()	在列表框中加入一个字符串
DeleteString()	从列表框中删除一个字符串
Dir()	在列表框的当前目录中存入新的文件名
FindString()	在列表框中搜索一字符串
FindStringExact()	在列表框中搜索第一个与指定搜索字符串匹配的字符串
InsertString()	在列表框指定下标处插入一字符串
ResetContent()	清除列表框中的所有项
SelectString()	在单选列表框中搜索并选择一字符串

(5) 虚拟方法

虽然表 14-5、14-6 和 14-7 中所列的方法都非常有用,但是只能如实地(as-is)调用它们。CListBox 类还声明了几个虚拟方法,你可以从 CListBox 类中派生一些类替换到你的类中。表 14-8 列出了 CListBox 类的虚拟方法,通过替换可以完成 MFC 没有直接提供的功能。

表 14-8　可被替换的 CListBox 类的虚拟方法

方　法	说　明
CharToItem( )	可以替换此方法来为自绘列表框(没有字符串)处理 WM_CHAR 消息
CompareItem( )	由 MFC 调用以得到排序的自绘列表框中的新项的位置
DeleteItem( )	当用户从自绘列表框中删除一项时 MFC 调用此方法
DrawItem( )	当确定自绘列表框项必须重绘时 MFC 调用此方法
MeasureItem( )	当一自绘列表框被创建时,MFC 调用此方法来决定列表框的维数
VKeyToItem( )	用户可替换此方法来处理具有 LBS_WANTKEYBOARDINPUT 样式的列表框的 WM_KEYDOWN 消息

3. 创建和初始化 CListBox 对象

CListBox 对象像大多数 MFC 对象一样,使用两步构造过程。创建一个列表框要执行下列步骤:

(1) 用 C++ 关键字 new 和构造函数 CListBox::CListBox( )为 CListBox 对象分配一个实例。

(2) 初始化 CListBox 对象并赋于它一个 Windows 列表框,通过方法 CListBox::Create( )设置列表框的参数和样式。例如,下面的代码分配一个 CListBox 对象并返回指向该对象的指针:

```
CListBox * pMyListBox = new CListBox;
```

指针 pMyListBox 用 CListBox::Create( )方法进行初始化。该方法声明如下:

```
BOOL Create
    (
    DWORD dwStyle,          // dwStyle 是列表框控件的窗口样式
    const RECT& rect,       // rect 是一个矩形,它指明控件的大小和位置
    CWnd* pParentWnd,       // pParentWnd 是指向控件所有者的指针
    UINT nID                // nID 是父窗口用来与列表框通信的控件标识
    );
```

其中,列表框控件的窗口样式 dwStyle 可以是任意一般窗口样式与列表框特有样式的组合。

14.2　按钮类及列表框类编程实例

本节通过编写一个简单的应用程序来学习如何在对话框窗口中应用按钮类、列表框类等控件类。

【例 14-1】

编写应用程序,其主窗口如图 14-4 所示,标题为 Application of SELECTBOX。在这个窗口中,包含有三个标题分别为 Check Box、Radio Box 和 Combo Box 的分组框,在 Check Box 分组框中包含有两个复选框 Date 和 Time,以及 Enable、Disable、Show Again 和 Hide the Clock Setting 四个按钮;在 Radio Box 分组框中,含有两个子分组框、一个按钮和一个编辑框,子分组框名字分别为 Sex Selecting 和 Age Range,它们分别包含了一组单选按钮,此外还有一个名字为 Show the Sex and Age 的按钮。在 Combo Box 分组框中,有一个 Coursef 分组框(其中包含 3 个单选按钮)、一个名字为 Record 的下拉列表框、一个 Show-Combo 按钮和一个编辑框。

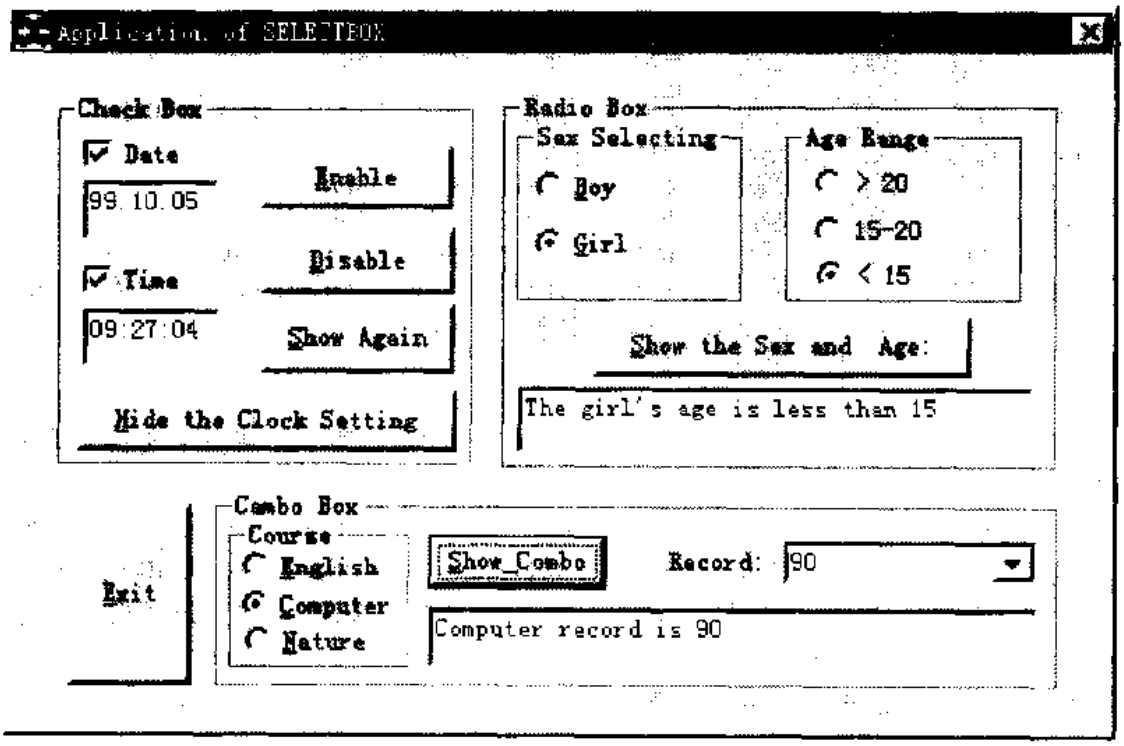


图 14-4 应用程序示例界面

本例的功能如下：

(1) Check Box 分组框中的控件

- Date 复选框：单击 Date 复选框，在其下面的编辑框中显示当前的日期，并在复选框中显示选中标志。
- Time 复选框：单击 Time 复选框，在其下面的编辑框中显示当前的系统时间，并在复选框中显示选中标志。
- Disable 按钮：单击 Disable 按钮上面的两个复选框变成无效，不响应操作，并且复选框和编辑框都变灰。
- Enable 按钮：单击 Enable 按钮，复选框又变成有效，可对其进行操作。
- Hide the Clock Setting 按钮：单击 Hide 按钮，隐藏掉复选框和编辑框，使它们不可见。
- Show Again 按钮：重新显示被隐藏的复选框和编辑框。

(2) Radio Box 分组框中的控件

- Sex Selecting 子分组框：在此子分组框中有 Boy 和 Girl 单选按钮，可单击其中的任何一项进行性别的选择。
- Age Range 子分组框：在此框中进行年龄段的选择。
- Show the Sex and Age 按钮：单击此命令按钮，在其下面的编辑框中显示一行信息，报告当前单选按钮的状态。

(3) Combo Box 分组框中的控件

- Course 子分组框：在此子分组框中有 English、Computer 和 Nature 三门课的选项，单击其中的任何一项进行课程科目的选择。
- Record 下拉列表框：在此框中进行成绩的选择。

- Show-Combo 按钮：单击此命令按钮，在其下面的编辑框中显示一行信息，报告当前单选按钮及下拉列表框的状态。

(4) Exit 按钮

单击此按钮，退出应用程序。

1. 应用程序的界面设计

因为 Application of SELECTBOX 应用程序是基于对话框的，所以应用 AppWizard 生成一个对话框窗口作为应用程序的主窗口，并且给这个对话框取名为 IDD_SELECTBOX_DIALOG，现在要做的工作就是在这个对话框窗口中进行界面设计。

在处于编辑状态的对话框中，通过控件工具进行各个控件的属性的设置，各控件的 ID 等属性如表 14-9 所示。

表 14-9　Application of SELECTBOX 应用程序对话框中各个对象的属性

对　象	所处位置	ID	Caption
分组框 Check Box	Check Box 分组框	IDC_STATIC	Check Box
Date 复选框	Check Box 分组框	IDC_DATE_CHECK	Date
Time 复选框	Check Box 分组框	IDC_TIME_CHECK	Time
Date 编辑框	Check Box 分组框	IDC_DATE_EDIT	无
Time 编辑框	Check Box 分组框	IDC_TIME_EDIT	无
Enable 命令按钮	Check Box 分组框	IDC_ENABLE_BUTTON	&Enable
Disable 命令按钮	Check Box 分组框	IDC_DISABLE_BUTTON	&Disable
Shoe Age 命令按钮	Check Box 分组框	IDC_SHOW_BUTTON	&Show Again
Hide the Clock Setting 命令按钮	Check Box 分组框	IDC_HIDE_BUTTON	&Hide the Clock Setting
Radio Box 分组框	Radio Box 分组框	IDC_STATIC	Radio Box
Sex Selecting 单选按钮组	Radio Box 分组框	IDC_STATIC	Sex Selecting
Age Range 单选按钮组	Radio Box 分组框	IDC_STATIC	Age Range
Boy 单选按钮	Radio Box 分组框	IDC_Boy_RADIO	&Boy
Girl 单选按钮	Radio Box 分组框	IDC_Girl_RADIO	&Girl
> 20 单选按钮	Radio Box 分组框	IDC_Age1_RADIO	> 20
15-20 单选按钮	Radio Box 分组框	IDC_Age2_RADIO	15-20
< 15 单选按钮	Radio Box 分组框	IDC_Age3_RADIO	< 15
Show the Sex and Age 命令按钮	Radio Box 分组框	IDC_Show_Sex_Age_BUTTON	&Show the Sex and Age:
Result 编辑框	Radio Box 分组框	IDC_Result_EDIT	无
Combo Box 分组框	Combo Box 分组框	IDC_STATIC	Combo Box
Course 分组框	Combo Box 分组框	IDC_STATIC	Course
Show Combo 命令按钮	Combo Box 分组框	IDC_SHOW_COMBO_BUTTON	&Show Combo
Show Combo 编辑框	Combo Box 分组框	IDC_SHOW_COMBO_EDIT	无
English 单选按钮	Combo Box 分组框	IDC_ENGLISH_RADIO	&Engliah
Computer 单选按钮	Combo Box 分组框	IDC_COMPUTER_RADIO	&Computer
Nature 单选按钮	Combo Box 分组框	IDC_NATURE_RADIO	&Nature
Exit 命令按钮	Application of SELECTBOX 对话框	IDC_EXIT_BUTTON	&Exit

表中单选按钮组的 ID 是相同的，都为 IDC_STATIC，说明不同的控件可以有相同的 ID。因为单选按钮组只是用来显示一组内容的属性，在编写代码时用不着单选按钮组控件，所以 ID 相同也无所谓。

对于成组的单选按钮，只在每组的第一个按钮的属性窗口中选中 Group 复选框，即在 Sex Selecting 组中只有 Boy 单选按钮选中 Group 属性，Age Range 组中只有“>20”单选按钮选中 Group 属性。而且在设计的过程中，同一组单选按钮必须一个接一个地放进对话框中，中间不能插入其他的控件。

为什么必须这样做呢？因为每一个控件都有 ID 值，Visual C++ 按照放入对话框中的先后顺序，给每个控件赋一个 ID 值，所以控件的 ID 值是连续的。设置了 Group 属性的控件之间的控件为一组。每个控件 ID 值都是可以查看和修改的，单击 View 菜单，选择 ID = Resource Symbols 命令，弹出一个对话框，如图 14-5 所示。该对话框中列出了所有控件的 ID 值，其中，Age1、Age2 和 Age3 的 ID 值是连续的。

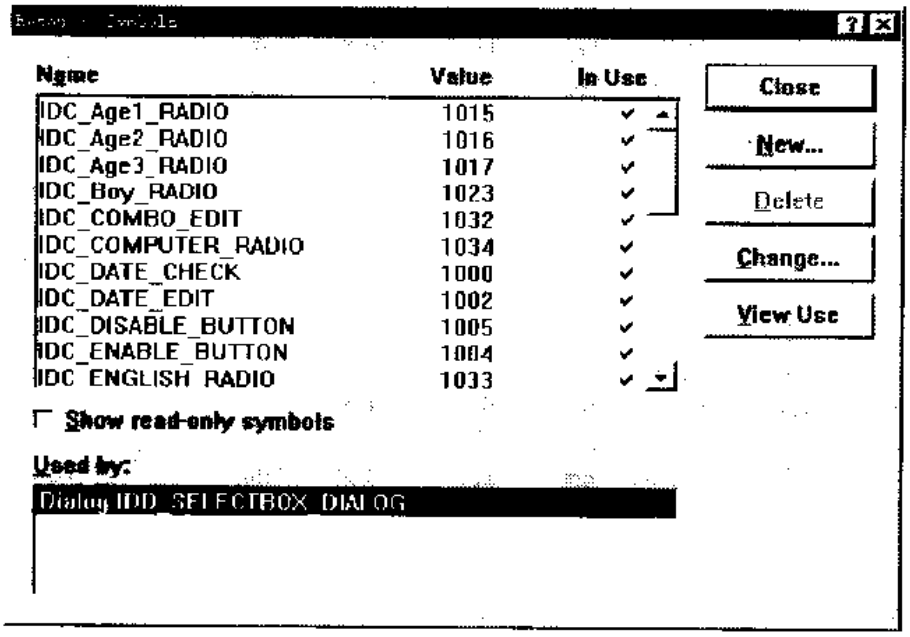


图 14-5 Resource Symbols 对话框

注意，对于成组的单选按钮控件，如果发现其 ID 值不连续，就要把它从对话框中删掉，然后重新加入到对话框中；而对于其他控件的 ID 值，连续与否没有特别的要求。

此外，还可以从 resource.h 的资源头文件中得到 ID 值，本应用程序的 resource.h 文件的代码如下：

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by SELECTBOX.rc
//
#define IDM_ABOUTBOX                    0x0010
#define IDD_ABOUTBOX                    100
#define IDS_ABOUTBOX                    101
```

```
#define IDD_SELECTBOX_DIALOG            102
#define IDR_MAINFRAME                   128
#define IDC_DATE_CHECK                  1000
#define IDC_TIME_CHECK                  1001
#define IDC_DATE_EDIT                   1002
#define IDC_TIME_EDIT                   1003
#define IDC_ENABLE_BUTTON               1004
#define IDC_DISABLE_BUTTON              1005
#define IDC_SHOW_BUTTON                 1006
#define IDC_HIDE_BUTTON                 1007
#define IDC_Age1_RADIO                  1015    //Age1,Age2 和 Age3 的 ID 值是连续的
#define IDC_Age2_RADIO                  1016
#define IDC_Age3_RADIO                  1017
#define IDC_Show_Sex_Age_BUTTON         1019
#define IDC_Result_EDIT                 1020
#define IDC_Boy_RADIO                   1023    //Boy 和 Girl 的 ID 值是连续的
#define IDC_Girl_RADIO                  1024
#define IDC_EXIT_BUTTON                 1027
#define IDC_SHOW_COMBO_BUTTON           1028
#define IDC_RECORD_COMBO                1031
#define IDC_COMBO_EDIT                  1032
#define IDC_ENGLISH_RADIO               1033
#define IDC_COMPUTER_RADIO              1034
#define IDC_NATURE_RADIO                1035

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1036
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

2. 应用程序的代码编程部分

(1) 给各个控件连接变量

在进行程序的代码编程过程之前，必须给每一个控件连接变量，控件的变量如表 14-10所示。

(2) 给有关按钮、编辑框等控件连接方法

与有关按钮、编辑框等控件连接的方法如表 14-11 所示。

表 14-10 控件及其连接的变量

ID	变量名	类 型
IDC_DATE_CHECK	m_DateCheck	BOOL
IDC_TIME_CHECK	m_TimeCheck	BOOL
IDC_DATE_EDIT	m_DateEdit	CEdit
IDC_TIME_EDIT	m_TimeEdit	CEdit
IDC_Boy_RADIO	m_SexRadio	CButton
IDC_Age1_RADIO	m_AgeRadio	CButton
IDC_Result_EDIT	m_ResultEdit	CEdit
IDC_ENGLISH_RADIO	m_English	int
IDC_COMBO_EDIT	m_ComboEdit	CString
IDC_RECORD_COMBO	m_Record	CComboBox
IDC_SHOW_COMBO_BUTTON	m_Combo_Button	CButton

表 14-11 与控件连接的方法

ID	方 法
IDC_DATE_CHECK	OnDateCheck()
IDC_TIME_CHECK	OnTimeCheck()
IDC_ENABLE_BUTTON	OnEnableButton()
IDC_DISABLE_BUTTON	OnDisableButton()
IDC_SHOW_BUTTON	OnShowButton()
IDC_HIDE_BUTTON	OnHideButton()
IDC_Show_Sex_Age_BUTTON	OnShowSexAgeButton()
IDC_SHOW_COMBO_BUTTON	OnShowComboButton()
IDC_EXIT_BUTTON	OnExitButton()

值得注意的是，每一组单选按钮中只有第一个按钮可以赋予变量名，其他的单选按钮不能获得变量名，所以图 14-5 中只有 IDC_Boy_RADIO 和 IDC_Age1_RADIO 能连接变量。

(3) 方法的实现

① 给复选框 IDC_DATE_CHECK 添加代码

OnDateCheck()方法的实现代码如下：

```
void CSELECTBOXDlg::OnDateCheck()
{
    // 在此处加入控件通知句柄代码
    //添加代码开始
    UpdateData(TRUE);
    if(m_DateCheck == TRUE)
    {
        CTime tNow;
        tNow = CTime::GetCurrentTime();
        CString sNow = tNow.Format("%y.%m.%d");
        m_DateEdit.SetSel(0,-1);
```

```
        m_DateEdit.ReplaceSel(sNow);
    }
    else
    {
        m_DateEdit.SetSel(0, -1);
        m_DateEdit.ReplaceSel("");
    }
    UpdateData(FALSE);
    //添加代码结束

}
```

输入的第一条语句：

```
UpdateData(TRUE);
```

表示以当前的屏幕显示内容更新控件的变量，也就是说，从屏幕上读取变量的值。因为在下面要判断复选框的状态，所以在这里需要获得当前复选框的状态。

接下来用一个 if 判断语句来对复选框的不同状态进行设置。

```
if(m_DateCheck == TRUE)
{
  ...
  ...
}
  else
  {
    ...
    ...
  }
```

因为要在编辑框中显示当前的日期，所以要用 CTime 类，CTime 类的结构如下：

```
class CTime
{
public:
// Constructors
    static CTime PASCAL GetCurrentTime();
    CTime();
    CTime(time_t time);
    CTime(int nYear, int nMonth, int nDay, int nHour, int nMin, int nSec,
        int nDST = -1);
    CTime(WORD wDosDate, WORD wDosTime, int nDST = -1);
    CTime(const CTime& timeSrc);
    CTime(const SYSTEMTIME& sysTime, int nDST = -1);
    CTime(const FILETIME& fileTime, int nDST = -1);
    const CTime& operator=(const CTime& timeSrc);
```

```
    const CTime& operator=(time_t t);
// Attributes
    struct tm* GetGmtTm(struct tm* ptm = NULL) const;
    struct tm* GetLocalTm(struct tm* ptm = NULL) const;
    BOOL GetAsSystemTime(SYSTEMTIME& timeDest) const;
//以下定义了 CTime 类的方法名
    time_t GetTime() const;
    int GetYear() const;
    int GetMonth() const;       // month of year (1 = Jan)
    int GetDay() const;         // day of month
    int GetHour() const;
    int GetMinute() const;
    int GetSecond() const;
    int GetDayOfWeek() const;  // 1 = Sun, 2 = Mon, ..., 7 = Sat

// Operations
    // time math
    CTimeSpan operator-(CTime time) const;
    CTime operator-(CTimeSpan timeSpan) const;
    CTime operator+(CTimeSpan timeSpan) const;
    const CTime& operator+=(CTimeSpan timeSpan);
    const CTime& operator-=(CTimeSpan timeSpan);
    BOOL operator==(CTime time) const;
    BOOL operator!=(CTime time) const;
    BOOL operator<(CTime time) const;
    BOOL operator>(CTime time) const;
    BOOL operator<=(CTime time) const;
    BOOL operator>=(CTime time) const;
    // formatting using "C" strftime
    CString Format(LPCTSTR pFormat) const;
    CString FormatGmt(LPCTSTR pFormat) const;
    CString Format(UINT nFormatID) const;
    CString FormatGmt(UINT nFormatID) const;

#ifdef _UNICODE
    // for compatibility with MFC 3.x
    CString Format(LPCSTR pFormat) const;
    CString FormatGmt(LPCSTR pFormat) const;

#endif
    // serialization
#ifdef _DEBUG
```

```
    friend CDumpContext& AFXAPI operator<<(CDumpContext& dc, CTime time);
#endif
    friend CArchive& AFXAPI operator<<(CArchive& ar, CTime time);
    friend CArchive& AFXAPI operator>>(CArchive& ar, CTime& rTime);
private:
    time_t m_time;                    //此处引入 time_t 数据类型
};
```

CTime 类的对象描述一个绝对的时间和日期，从上面的 CTime 类的定义中可以看出，它引入了 time_t 数据类型。

```
CTime tNow;
tNow = CTime::GetCurrentTime();
```

这两条语句是定义一个 CTime 类的对象，将当前的时间和日期赋给对象 tNow。下面一条语句是将时间值转换为字符串类型：

```
CString sNow.Format("%Y.%m.%d");
```

其中函数 Format 的参数%Y 是日期的年的表示法，%m 是月的表示法(01 ~ 12)，%d 是日的表示法(01 ~ 31)。如果当前的日期是 1995 年 4 月 14 日，那么执行该语句后，字符串 sNow 的值为 1995.04.14。

接下来是将字符串的值显示在编辑框 IDC_DATE_EDIT 中，用下面两条语句实现：

```
m_DateEdit.SetSel(0, -1);
m_DateEdit.ReplaceSel("");
```

最后一句：

```
UpdateData(FALSE);
```

是用变量的值刷新屏幕。实际上，在这里可以省略掉该语句，因为我们设定的与编辑框相关联的变量是 CEdit 类型；如果是 CString 类型的话，则必须加上这条语句来更新屏幕。

② 为复选框 IDC_TIME_CHECK 添加代码

OnTimeCheck()方法的实现代码如下：

```
void CSELECTBOXDlg::OnTimeCheck()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    UpdateData(TRUE);
    if(m_TimeCheck == TRUE)
    {
        CTime tNow;
        tNow = CTime::GetCurrentTime();
        CString sNow = tNow.Format("%I:%M:%S");
        m_TimeEdit.SetSel(0, -1);
        m_TimeEdit.ReplaceSel(sNow);
    }
    else
```

```
    {
        m_TimeEdit.SetSel(0,-1);
        m_TimeEdit.ReplaceSel("");
    }
    UpdateData(FALSE);
    //添加代码结束
}
```

其中函数 Format 的参数%I 是时间的小时表示法(01 ~ 12),%m 是分的表示法(00 ~ 59),%d 是秒的表示法(00 ~ 59)。

③ 给 Enable 按钮添加代码

OnEnableButton()方法的实现代码如下:

```
void CSELECTBOXDlg::OnEnableButton()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    GetDlgItem(IDC_DATE_CHECK)->EnableWindow(TRUE);
    GetDlgItem(IDC_TIME_CHECK)->EnableWindow(TRUE);
    m_DateEdit.EnableWindow(TRUE);
    m_TimeEdit.EnableWindow(TRUE);
    //添加代码结束
}
```

输入函数的前两条语句

```
GetDlgItem(IDC_DATE_CHECK)->Enable Window(TRUE);
GetDlgItem(IDC_TIME_CHECK)->Enable Window(TRUE);
```

是使两个复选框可选。在这里是调用一个指向对象的指针函数 GetDlgItem()。函数 GetDlgItem()是 CWnd 类的成员函数,因为 CDialog 类是 CWnd 类的派生类,而 CSELECTBOXDlg 类又是 CDialog 类的派生类,所以 CSELECTBOXDlg 类继承了基类 CWnd 的成员函数,故可调用 CWnd 的成员函数。

下面两句是使编辑框可用,这里是用成员变量来调用函数。

```
m_DateEdit.EnableWindow(TRUE);
m_TimeEdit.EnableWindow(TRUE);
```

这两句也可以用指针来表示:

```
GetDlgItem(IDC_DATE_EDIT)->EnableWindow(TRUE);
GetDlgItem(IDC_TIME_EDIT)->EnableWindow(TRUE);
```

那么对于程序中的语句:

```
GetDlgItem(IDC_DATE_CHECK)->EnableWindow(TRUE);
GetDlgItem(IDC_TIME_CHECK)->EnableWindow(TRUE);
```

是否可以用变量来调用函数呢?答案是否定的。因为与编辑框相关联的变量是 CEdit 类型的,当然可以调用成员函数。而与复选框相关联的变量是 BOOL 类型的,并不是 CButton 类型的,所以不能用它来调用成员函数。即下面两条语句是错误的:

```
m_DateCheck.EnableWindow(TRUE);
m_TimeCheck.EnableWindow(TRUE);
```

④ 为 Disable 按钮连接代码

OnDisableButton()方法的实现代码如下：

```
void CSELECTBOXDlg::OnDisableButton()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    GetDlgItem(IDC_DATE_CHECK)->EnableWindow(FALSE);
    GetDlgItem(IDC_TIME_CHECK)->EnableWindow(FALSE);
    m_DateEdit.EnableWindow(FALSE);
    m_TimeEdit.EnableWindow(FALSE);
    //添加代码结束
}
```

在这里，代码与 OnEnableButton()方法相类似，只是把 TRUE 改成 FALSE，即使复选框无效，无法对复选框进行任何操作。

⑤ 为 Show Again 按钮添加代码

在函数 OnShowButton()方法中添加实现代码如下：

```
void CSELECTBOXDlg::OnShowButton()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    GetDlgItem(IDC_DATE_CHECK)->EnableWindow(SW_SHOW);
    GetDlgItem(IDC_TIME_CHECK)->EnableWindow(SW_SHOW);
    GetDlgItem(IDC_DATE_EDIT)->EnableWindow(SW_SHOW);
    m_DateEdit.ShowWindow(SW_SHOW);
    GetDlgItem(IDC_TIME_EDIT)->EnableWindow(SW_SHOW);
    m_TimeEdit.ShowWindow(SW_SHOW);
    //添加代码结束
}
```

输入函数的前两条语句

```
GetDlgItem(IDC_DATE_CHECK)->ShowWindow(SW_SHOW);
GetDlgItem(IDC_TIME_CHECK)->ShowWindow(SW_SHOW);
```

是使两个复选框可见。在这里是调用一个指向对象的指针函数 GetDlgItem()。下面两句是使编辑框可见，这里也是用指针变量来调用函数。

```
GetDlgItem(IDC_DATE_CHECK)->ShowWindow(SW_SHOW);
GetDlgItem(IDC_TIME_CHECK)->ShowWindow(SW_SHOW);
```

⑥ 为 Hide the Clock Setting 按钮添加实现代码

在函数 OnHideButton()方法中输入以下实现代码：

```
void CSELECTBOXDlg::OnHideButton()
```

```
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    GetDlgItem(IDC_DATE_CHECK)->EnableWindow(SW_RTDR);
    GetDlgItem(IDC_TIME_CHECK)->EnableWindow(SW_RTDR);
    m_DateEdit.ShowWindow(SW_HIDE);
    m_TimeEdit.ShowWindow(SW_HIDE);
    //添加代码结束
}
```

函数 ShowWindow()是基类 CWnd 的成员函数,它表示是否显示对象窗口,参数 SW SHOW 表示显示,SW HIDE 表示隐藏。

⑦ 给 Show the Sex and Age 按钮添加代码

OnShowSexAgeButton()方法的实现代码如下:

```
void CSELECTBOXDlg::OnShowSexAgeButton()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    char sEdit[50];
    int iSexRADIO;
    int iAgeRadio;
    iSexRadio = GetCheckedRadioButton(IDC_Boy_RADIO,IDC_Girl_RADIO);
    if(iSexRadio == IDC_Boy_RADIO)
        strcpy(sEdit,"The boy's age is");
    if(iSexRadio == IDC_Girl_RADIO)
        strcpy(sEdit,"The girl's age is");
    iAgeRadio = GetCheckedRadioButton(IDC_Age1_RADIO,IDC_Age3_RADIO);
    if(iAgeRadio == IDC_Age1_RADIO)
        strcat(sEdit," great than 20");
    if(iAgeRadio == IDC_Age2_RADIO)
        strcat(sEdit," between 15 and 20");
    if(iAgeRadio == IDC_Age3_RADIO)
        strcat(sEdit," less than 15");
    m_ResultEdit.SetSel(0,-1);
    m_ResultEdit.ReplaceSel(sEdit);
    //添加代码结束
}
```

在上面的代码中,首先声明一个字符串变量,用来存放显示在编辑框中的字符串:

```
char sEdit[50];
```

然后声明两个变量来表示两组单选按钮的状态:

```
int iSexRadio;
int iAgeRadio;
```

因为用单选按钮的 ID 值来表示状态，所以定义变量为整型变量。接下来确定 Sex Selecting单选按钮组的状态：

```
iSexRadio = GetCheckedRadioButton(IDC_Boy_RADIO,IDC_Girl_RADIO);
```

函数 GetCheckedRadioButton 的第一个参数表示该组第一个单选按钮的 ID 号，第二个参数表示该组中最后一个单选按钮的 ID 号。该函数返回一个整数，这个整数是两个参数之间被选中的单选按钮 ID 号。将这个 ID 号的值赋给变量 iSexRadio。

再接下来，用 if 语句判断被选中的单选按钮，并执行相应的代码。

```
if(iSexRadio == IDC_Boy_RADIO)
    strcpy(sEdit,"The boy's age is");
if(iSexRadio == IDC_Girl_RADIO)
    strcpy(sEdit,"The girl's age is");
```

If 语句中判断被选中的 ID 是否为等号后面的 ID，如果是则执行下面的 strcpy 函数，把相应字符串复制到 sEdit 字符数组中去。

对于 Age Range 单选按钮组也用同样的方法。先确定 Age 组单选按钮的状态：

```
iAgeRadio = GetCheckedRadioButton(IDC_Age1_RADIO,IDC_Age3_RADIO);
```

接下来，用 if 语句判断被选中的单选按钮，并执行相应的代码。

```
if(iAgeRadio == IDC_Age1_RADIO)
    strcat(sEdit," great than 20");
if(iAgeRadio == IDC_Age2_RADIO)
    strcat(sEdit," between 15 and 20");
if(iAgeRadio == IDC_Age3_RADIO)
    strcat(sEdit," less than 15");
```

strcat()函数用于将一个字符串连接到另一个字符串的后面，在本例中，就是把相应的字符串连接到字符数组 sEdit 已有的字符的后面。

最后，将字符串在编辑框中显示出来：

```
m_ResultEdit.SetSel(0,-1);
m_ResultEdit.ReplaceSel(sEdit);
```

⑧ 为 Show Combo 按钮添加代码

OnShowComboButton()方法的实现代码如下：

```
void CSELECTBOXDlg::OnShowComboButton()
{
    //添加代码开始
    UpdateData(TRUE);
    char sCourseEdit[30];
    char sRecordEdit[15];
    int iCourseRadio;
    iCourseRadio = GetCheckedRadioButton(IDC_ENGLISH_RADIO,IDC_NATURE_RADIO);
    if(iCourseRadio == IDC_ENGLISH_RADIO)
        strcpy(sCourseEdit,"English record is");
    if(iCourseRadio == IDC_COMPUTER_RADIO)
```

```
        strcpy(sCourseEdit,"Computer record is ");
    if(iCourseRadio == IDC_NATURE_RADIO)
        strcpy(sCourseEdit,"Natural record is ");
    m_Record.GetWindowText(sRecordEdit,15);
    strcat(sCourseEdit,"");
    strcat(sCourseEdit,sRecordEdit);
    m_ComboEdit = T(sCourseEdit);
    UpdateData(FALSE);
    //添加代码结束
}
```

输入的第一条语句

```
UpdateData(TRUE);
```

用来将屏幕的状态显示给变量。

首先声明两个字符串变量,用来存放显示在编辑框中的字符串:

```
char sCourseEdit[30];
char sRecordEdit[15];
```

其中 sCourseEdit 是用来存放编辑框的内容。

然后声明一个变量来表示单选按钮的状态:

```
int iCourseRadio;
```

因为用单选按钮 ID 值来表示状态,所以定义变量为整型变量。

接下来确定 Sex 单选按钮组的状态:

```
iCourseRadio = GetCheckedRadioButton(IDC_ENGLISH_RADIO,IDC_NATURE_RADIO);
```

再接下来,由 if 语句判断被选中的单选按钮,并执行相应代码。

```
if(iCourseRadio == IDC_ENGLISH_RADIO)
    strcpy(sCourseEdit,"English record is ");
if(iCourseRadio == IDC_COMPUTER_RADIO)
    strcpy(sCourseEdit,"Computer record is ");
if(iCourseRadio == IDC_NATURE_RADIO)
    strcpy(sCourseEdit,"Natural record is ");
```

if 语句中判断被选中的 ID 是否为等号后面的 ID,如果是则执行下面的 Strcpy 函数。

函数 GetWindowText()的第一个参数是一个字符串变量,用来存放分组框的编辑框的内容,第二个参数是一个整数,它是将拷贝到第一个参数中提示的字符串的字符的最大数目。函数 GetWindowText()并不是类 CComboBox 的成员函数,而是类 CWnd 的成员函数,但类 CComboBox 是类 CWnd 的派生类,所以可以调用该函数。由于我们在显示的时候是将两个变量的内容连接在一起,所以在两个变量之间加一个空格:

```
strcat(sCourseEdit,"");
```

strcat 函数用于将一个字符串连接到另一个字符串的后面。然后将第二个变量连接在第一个变量的后面。

最后,结果通过下面的语句在编辑框中显示:

```
m_ComboEdit = T(sCourseEdit);
```

```
UpdateData(FALSE);
```

⑨ 给 OnExitButton()方法添加代码

OnExitButton()的实现代码如下：

```
void CSELECTBOXDlg::OnExitButton()
{
    //在此处加入控件通知句柄代码
    //添加代码开始
    OnOK();
    //添加代码结束
}
```

(4) 初始化单选按钮

当我们一运行 Application of SELECTBOX 应用程序，单选按钮 Boy 和 Girl 就设置了选中标志。下面我们对应用程序进行初始化。在对话框上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令，Visual C++ 显示 ClassWizard 对话框。选择 Message Maps 选项卡，我们会发现在 Member functions 中，Visual C++ 已经生成了初始化函数 OnInitDialog。AppWizard 生成工程文件时，就在源代码文件中声明了该函数。单击 Member functions 列表框中的 OnInitDialog 项，对 OnInitDialog()函数输入用户自定义的内容。

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

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
        }
    }

    // Set the icon for this dialog.  The framework does this automatically
    // when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon
```

```
    // 在此加入其他初始化代码(Add extra initialization here)
    // 添加代码开始,为单选按钮及列表框进行初始化定义
    CheckRadioButton(IDC_Boy_RADIO,IDC_Girl_RADIO,IDC_Boy_RADIO);
    CheckRadioButton(IDC_Age1_RADIO,IDC_Age3_RADIO,IDC_Age2_RADIO);
    UpdateData(FALSE);
    m_English = 0;
    m_Record.AddString("85");
    m_Record.AddString("90");
    m_Record.AddString("95");
    m_Record.SelectString(-1,"95");
    UpdateData(FALSE);
    //代码添加结束
    return TRUE;// return TRUE unless you set the focus to a control
}
```

函数 CheckRadioButton()的第一个参数是该组中第一个单选按钮的 ID,第二个参数是该组中最后一个单选按钮的 ID,第三个参数是在该组中被选中的单选按钮的 ID。

我们给 SexSelecting 组赋单选按钮初值为 Boy,所以第三个参数为 IDC_Girl_RADIO;给 Age Range 组单选按钮赋初值为"15-20",所以第三个参数为 IDC_Age2_RADIO。同时,由于已经声明了变量 m_EnglishRadio 为 int 类型,所以可以用下面这种形式来选择单选按钮:

```
m_EnglishRadio = 0;
```

表示选中的是第一个选项 IDC_ENGLISH_RADIO,与语句:

```
CheckRadioButton(IDC_ENGLISH_RADIO,IDC_COMPUTER_RADIO,IDC_NATURE_RADIO);
```

的功能一样。若输入 m_EnglishRadio = 1,则选择的是第二个选项 IDC_COMPUTER_RADIO。

在设置了单选按钮的变量后,就用函数 UpdateData(FALSE)来修改屏幕。在分组框的列表框中,使用函数 AddString 加入可选的几项:

```
m_Record.AddString("85");
m_Record.AddString("90");
m_Record.AddString("95");
```

但函数 AddString 只是在分组框的列表框中加入选项,并不能在分组框的编辑框中显示出来,因此还要加上下面的语句:

```
m_Record.SelectString(-1,"95");
```

在分组框的编辑框中显示默认的初始值。

习 题

14-1 按钮控件是如何应用的?

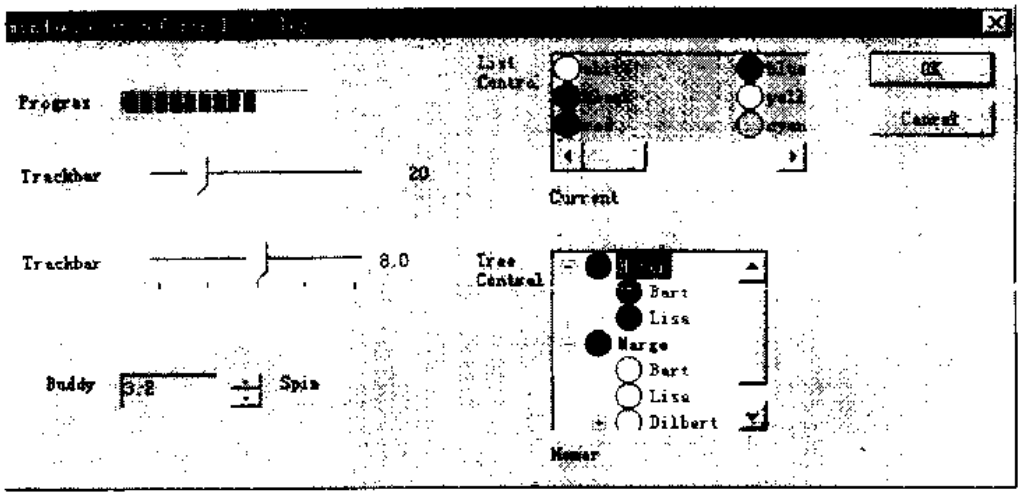
14-2 按钮控件分为几类?

14-3 各种按钮控件的类是如何定义的?

14-4　哪些按钮控件需要初始化?

14-5　创建一个显示成绩的单选按钮控件,成绩项包括"100"、"90"、"80"和"70"四档;创建一个复选按钮控件组,复选项为"加权"(对上述成绩的权量分别为 10、9、8、7),布置一个"计算"按钮和一个"退出"按钮,当单击"计算"按钮时,在编辑框中显示成绩的平均值。

14-6　创建一个如下图所示的界面



14-7　创建一个下拉列表框控件,表中能对 10 个学生的名字进行选择;另有一个列表框,列有 5 个有关学生兴趣爱好的选项。在两个下拉列表框中选择相应的内容,单击 OK 按钮后,在对话框中显示所选的学生的名字和相应的爱好。

第15章

工具条类及其应用

前面的章节中已经介绍了菜单，菜单是用户选择可用命令的一个最常用，也是最重要的手段。然而，在有些情况下，使用工具条可以更快捷、更有效地实现某些操作。

本章将介绍如何使用MFC工具条类在应用程序中设计一个用户自己定义的工具条。对于一个基于单文档或多文档界面的应用程序，Visual C++的AppWizard会在窗口中直接产生一个工具条，但很难掌握工具条的构建方法，因此本章中我们将建立一个不带工具条的工程文件，然后通过加入工具条资源和添加必要的代码来实现工具条的创建和应用。

15.1 工具条类简介

15.1.1 工具条

菜单允许用户用一种标准的方法（即样式一致的用户界面对象——菜单栏）与Windows应用程序进行交互。与菜单相比，工具条是直接可以看到的，它可以看作是图形化的菜单，是一种更快捷、更有效、更直观的命令输入方式。由于它占用屏幕空间，因此一般来说工具条包括的是最经常使用的命令。大型程序通常有多个工具条来为不同的用户任务服务。即使程序只有一个工具条，也要使用户在感觉它碍事时能将其隐藏。

工具条作为程序的一个资源一般存放在资源脚本中，在程序运行时才调用。一般来说，工具条的标识符ID在资源的头文件（resource.h）中定义，如：

```
#define IDR_TOORBAR  130
```

其中，IDR_TOORBAR与系统菜单、系统图标等项资源的ID都相同，这样可以使工具条资源创建后直接链接到主窗口中。

15.1.2 工具条资源模板

工具条资源模板用一种很简单的结构在资源脚本文件中定义，一个工具条资源脚本文件中定义了不同的按钮。同时，整个工具条对应一个.bmp位图。

下面就是本章要讨论的例子的工具条资源模板，各项的说明已同时以注释的形式标出：

```
// Toolbar
IDR_TOOLBAR TOOLBAR DISCARDABLE 16,15  //工具条，每个按钮的大小是16像素宽，15像素
                                       //高
BEGIN                                  //工具条定义开始
    BUTTON      ID_FILE_NEW            //工具条按钮的ID，对应"创建"新文件按钮
```

```
    BUTTON      ID_FILE_OPEN            //工具条按钮的 ID,对应"打开"文件按钮
    BUTTON      ID_FILE_SAVE_AS         //工具条按钮的 ID,对应"另存为"按钮
    SEPERATOR                           //分隔块
    BUTTON      ID_SHOW                 //工具条按钮的 ID,对应"显示"按钮
END                                     //工具条定义结束
...
//Bitmap
IDR_TOOLBAR  BITMAP  DISCARDABLE  "res\\toolbar.bmp"
                                        //与工具条相应的位图文件
```

这个资源模板是在创建工具条资源时由系统自动产生的,内容保存在 mytoolbar.rc 资源文件中,其中 mytoolbar 是工程文件名。从上面的代码可以知道,每一个工具条由保留字 TOOLBAR 来标识,格式是:

ID 值 TOOLBAR [工具条属性] 像素宽,像素高

每一个工具条上的按钮由 BUTTON 定义,工具条名下的按钮命令选项由 BEGIN…END 组合在一起,命令选项的格式是:

BUTTON ID 值

15.1.3　工具条类的结构及其方法

谈到工具条,首先让我们从 MFC 控制条谈起。MFC 的控制条类 CControlBar 是可用来接收某些命令输入并向用户显示状态消息的类。工具条类(CToolBar)是从 CControlBar 类派生的,它在 MFC 类库中的层次位置如图 15-1 所示:

从图 15-1 中可见,工具条类的基类为 CControlBar,而该基类是由 CWnd 类派生的,掌握这个继承关系对我们来说是很有用的。例如,由于所有的控制条都是由 CWnd 类派生的,所有的控制条都连接到一个 Windows API 窗口。因此,CWnd 的所有功能如创建、移动、显示和隐藏窗口等在用控制条工作时都是可用的。

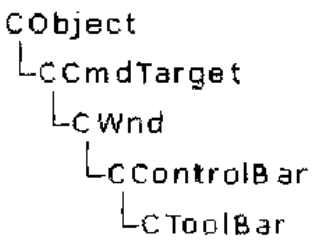


图 15-1　CToolBar 类在 MFC 类库中的层次位置

CToolBar 类有几个兄弟类,包括 CStatusBar(状态条)、COleResizeBar(OLE 控制条)和 CDialogBar(对话条),它们都是用来接收某些命令输入并向用户显示状态消息的类。

1. CToolBar 类的结构

CToolBar 类的定义中给出了该类的结构,从这个结构中,可大致了解 CToolBar 类的功能。下面是 CToolBar 类的定义:

```
class CToolBar : public CControlBar
{

    DECLARE_DYNAMIC(CToolBar)
// Construction
```

```
public:
    CToolBar();
    BOOL Create(CWnd* pParentWnd,
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP,
    UINT nID = AFX_IDW_TOOLBAR);
    BOOL CreateEx(CWnd* pParentWnd, DWORD dwCtrlStyle = TBSTYLE_FLAT,
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP,
    CRect rcBorders = CRect(0, 0, 0, 0),
    UINT nID = AFX_IDW_TOOLBAR);
    void SetSizes(SIZE sizeButton, SIZE sizeImage);
        // button size should be bigger than image
    void SetHeight(int cyHeight);
        // call after SetSizes, height overrides bitmap size
    BOOL LoadToolBar(LPCTSTR lpszResourceName);
    BOOL LoadToolBar(UINT nIDResource);
    BOOL LoadBitmap(LPCTSTR lpszResourceName);
    BOOL LoadBitmap(UINT nIDResource);
    BOOL SetBitmap(HBITMAP hbmImageWell);
    BOOL SetButtons(const UINT* lpIDArray, int nIDCount);
        // lpIDArray can be NULL to allocate empty buttons
// Attributes
public:
    // standard control bar things
    int CommandToIndex(UINT nIDFind) const;
    UINT GetItemID(int nIndex) const;
    virtual void GetItemRect(int nIndex, LPRECT lpRect) const;
    UINT GetButtonStyle(int nIndex) const;
    void SetButtonStyle(int nIndex, UINT nStyle);
    // for changing button info
    void GetButtonInfo(int nIndex, UINT& nID, UINT& nStyle, int& iImage) const;
    void SetButtonInfo(int nIndex, UINT nID, UINT nStyle, int iImage);
    BOOL SetButtonText(int nIndex, LPCTSTR lpszText);
    CString GetButtonText(int nIndex) const;
    void GetButtonText(int nIndex, CString& rString) const;
    // for direct access to the underlying common control
    CToolBarCtrl& GetToolBarCtrl() const;
// Implementation
public:
    virtual ~CToolBar();
    virtual CSize CalcFixedLayout(BOOL bStretch, BOOL bHorz);
    virtual CSize CalcDynamicLayout(int nLength, DWORD nMode);
    virtual int OnToolHitTest(CPoint point, TOOLINFO* pTI) const;
```

```
    virtual void OnUpdateCmdUI(CFrameWnd* pTarget, BOOL bDisableIfNoHndler);
    void SetOwner(CWnd* pOwnerWnd);
    BOOL AddReplaceBitmap(HBITMAP hbmImageWell);
    virtual void OnBarStyleChange(DWORD dwOldStyle, DWORD dwNewStyle);

#ifdef DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:
    HRSRC m_hRsrcImageWell; // handle to loaded resource for image well
    HINSTANCE m_hInstImageWell; // instance handle to load image well from
    HBITMAP m_hbmImageWell; // contains color mapped button images
    BOOL m_bDelayedButtonLayout; //used to manage when button layout should be
    done
    CSize m_sizeImage; // current image size
    CSize m_sizeButton; // current button size
    CMapStringToPtr* m_pStringMap; // used as CMapStringToUInt
    // implementation helpers
    void GetButton(int nIndex, TBBUTTON* pButton) const;
    void SetButton(int nIndex, TBBUTTON* pButton);
    CSize CalcLayout(DWORD nMode, int nLength = -1);
    CSize CalcSize(TBBUTTON* pData, int nCount);
    int WrapToolBar(TBBUTTON* pData, int nCount, int nWidth);
    void SizeToolBar(TBBUTTON* pData, int nCount, int nLength, BOOL bVert =
                     FALSE);
    void Layout(); // called for for delayed button layout

    //{{AFX_MSG(CToolBar)
    afx_msg UINT OnNcHitTest(CPoint);
    afx_msg void OnNcPaint();
    afx_msg void OnPaint();
    afx_msg void OnNcCalcSize(BOOL, NCCALCSIZE_PARAMS*);
    afx_msg void OnWindowPosChanging(LPWINDOWPOS);
    afx_msg void OnSysColorChange();
    afx_msg LRESULT OnSetButtonSize(WPARAM, LPARAM);
    afx_msg LRESULT OnSetBitmapSize(WPARAM, LPARAM);
    afx_msg LRESULT OnPreserveZeroBorderHelper(WPARAM, LPARAM);
    afx_msg BOOL OnNcCreate(LPCREATESTRUCT);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
    LRESULT OnSetSizeHelper(CSize& size, LPARAM lParam);
;
```

2. CToolBar 类的方法

CToolBar 类提供了许多工具条的处理方法，这些方法分别是构造方法、工具条按钮的操作方法和虚拟方法。

构造方法用于建立 Windows 工具条 CToolBar 对象并在运行时将它们附加到框架窗口上，表 15-1 是 CToolBar 的构造方法。

表 15-1 CToolBar 的构造方法

方 法	说 明
Create( )	创建一个工具条并把它附加到 CToolBar 对象上
CreateEx( )	创建一个定义了边界的工具条并把它附加到 CToolBar 对象上
SetSizes( )	设置按钮及位图大小
SetHeight( )	设置工具条的高度
LoadToolBar( )	装载工具条资源
LoadBitmap( )	装载包含工具按钮图像的位图
SetBitmap( )	设置位图图像
SetButtons( )	设置按钮并使每个按钮与位图图像相关

工具条按钮的操作方法是用来处理某一工具条按钮的，这些方法具体说明如表 15-2 所示。

表 15-2 工具条按钮的操作方法

方 法	说 明
CommandToIndex( )	返回给定命令的工具条按钮索引
GetItemID( )	返回指定索引的按钮或分隔符的 ID
GetItemRect( )	返回指定索引的按钮的显示区域
GetButtonStyle( )	获得按钮风格
SetButtonStyle( )	设置按钮风格
GetButtonInfo( )	获得按钮 ID、风格、图像号
SetButtonInfo( )	设置按钮 ID、风格、图像号
GetButtonText ( )	获得显示在按钮上的文本
SetButtonText( )	设置显示在按钮上的文本

下面介绍表 15-1 中各构造方法(或称之为函数)的功能。

(1) Create( )函数

Create( )函数的作用是创建并初始化工具条对象。与其他创建过程类似，采用两步法生成：首先要声明对象，然后再调用初始化函数 CToolBar::Create 生成工具条对象。该函数原型为：

```
BOOL Create
(
```

```
    CWnd * pParentWnd,          //pParentWnd 为指向工具条所在父窗口的指针
    DWORD dwStyle = WS_CHILD|WS_VISIBLE|CBRS_TOP, // dwStyle 为工具条的风格,见表
                                                  //15-3
    UINT nID = AFX_IDW_TOOLBAR      //nID 表示工具条子窗口的 ID 号
);
```

表 15-3　工具条风格

标　志	简单描述
WS_VISIBLE	使工具条窗口初始可见
CBRS_BOTTOM	初始时将工具条放到窗口底部
CBRS_FLYBY	鼠标光标在按钮上暂停时,显示命令描述
CBRS_NOALIGN	防止控制条在其父窗口改变大小时被复位
CBRS_TOOLTIPS	鼠标光标在按钮上暂停时,显示工具提示
CBRS_TOP	初始时将工具条放在窗口顶部
CBRS_SIZE_DYNAMIC	工具条的大小可变

如下述代码生成一工具条,工具条风格为"初始可见",且放置在窗口"顶部",工具条窗口的 ID 号的 16 进制值为 9100:

```
CToolBar m_ToolBar  new CToolBar( );
m_ToolBar.Create(this, WS_VISIBLE |CBRS_TOP,0x9100);
```

(2) LoadToolBar()函数

LoadToolBar()函数的作用是装载工具条资源。首先调用初始化函数 CToolBar::Create,然后再调用该函数装载资源。该函数原型为:

```
BOOL LoadToolBar(LPCTSTR lpszResourceName);//lpszResourceName 为字符串形式定义的
                                           //资源
BOOL LoadToolBar(UINT nIDResource);        //nIDResource 为整型形式定义的资源
```

例如,下述代码将整型形式定义的工具条资源 IDR_TOOLBAR 装载到变量 m_wndToolBar 中:

```
m_wndToolBar.LoadToolBar(IDR_TOOLBAR);
```

(3) SetBarStyle()函数

SetBarStyle()函数的作用是设置工具条风格。如果在调用初始化函数 CToolBar::Create 时设置的工具条风格不满足需要,可以再用这一函数进行设置。具体的工具条风格与表 15-1 中的说明相同。

如下述代码设定工具条的风格为:当鼠标光标在按钮上暂停时,显示工具提示和命令描述,并设定工具条的大小是可变的。

```
m_wndToolBar.SetBarStyle(CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
```

(4) LoadBitmap()函数

要将储存在位图资源中的按钮映像与程序的工具条相连接,可以通过函数调用 CToolBar::LoadBitmap 实现。该函数原型为:

```
BOOL LoadBitmap( LPCTSTR lpszResourceName ); //lpszResourceName 为字符串形式定义
                                              //的资源
BOOL LoadBitmap( UINT nIDResource ); //nIDResource 为整型形式定义的资源
```

如下述代码：

```
m_ToolBar.LoadBitmap(IDR_TOOLS);
```

功能是将储存在位图资源 IDR_TOOLS 中的按钮映像与程序的工具条变量 m_ToolBar 相连接。

(5) 工具条的移动

在默认状况下，一个 CToolBar 工具条只能被应用程序所移动，但也可以在程序中设置允许用户在程序运行过程中在框架窗口内移动工具条。为此，需向工具条及框架窗口发送消息，这可通过调用 CToolBar::EnableDocking 和 CFrame::EnableDocking 实现。二函数原型均如下：

```
void EnableDocking( DWORD dwStyle );
```

其中参数 dwStyle 为工具条风格，对 CToolBar 类其可取值如表 15-4 所示。

表 15-4 工具条停靠风格

风 格	意 义	风 格
CBRS_ALIGN_TOP	工具条可在客户区顶端移动	
CBRS_ALIGN_BOTTOM	工具条可在客户区底端移动	
CBRS_ALIGN_LEFT	工具条可在客户区左端移动	
CBRS_ALIGN_RIGHT	工具条可在客户区右端移动	
CBRS_ALIGN_ANY	工具条可在客户区任意位置移动	
CBRS_FLOAT_MULTI	允许在一个窗口内存在多个可移动控制条	对于 CFrame不可用

下面这段代码是实现允许用户移动工具条的常用代码：

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
```

在程序控制下，可通过调用 CFrameWnd::DockControlBar 来移动以及调用 CFrameWnd::FloatControlBar 来定位一个工具条。它们的原型及参数可以从系统帮助文件中查看，在此不再赘述。

(6) 工具条的显隐控制

由于工具条是一个窗口，它的显示或隐藏可以通过其父类 CWnd 的成员函数来实现。在改变工具条状态前知道当时工具条的状态有时是至关重要的，工具条的风格可以通过函数 CWnd::GetStyle 获取。该函数不带参数，其原型为：

```
DWORD GetStyle( ) const;
```

本函数返回当前控制条风格值。通过对函数 CWnd::SetStyle 的调用可以改变某些工具条窗口的风格，但该函数不能改变 WS_VISIBLE，这意味着要改变该风格必须调用其基类的成员函数 ShowWindow 来实现：将参数 SW_HIDE 传给函数以使工具条不可见或传递 SW_SHOWNORMAL 再次显示工具条。

一旦在程序中改变了工具条，就必须将这一改变通知框架窗口，为此需再次计算工具

条的位置，这可以通过调用函数 CFrameWnd::RecalcLayout 来实现。

15.2　工具条类编程实例

【例 15-1】

创建一名为 MyToolBar 的应用程序，界面如图 15-2 所示，其标题为“MyToolbar”。在应用程序的主窗口中有一个工具条，上面有四个按钮，分别为“N”、“O”、“S”和“sh”。在窗口中单击“显示(S)”执行菜单，则显示一个信息窗口，窗口信息为“这是关于工具条的样例程序”；若在窗口中单击工具条上相应的 sh 按钮，也会有同样的效果。若单击“查看(V)”菜单，也会弹出一个下拉菜单；单击“工具条(T)”项，可使工具条隐藏或出现。

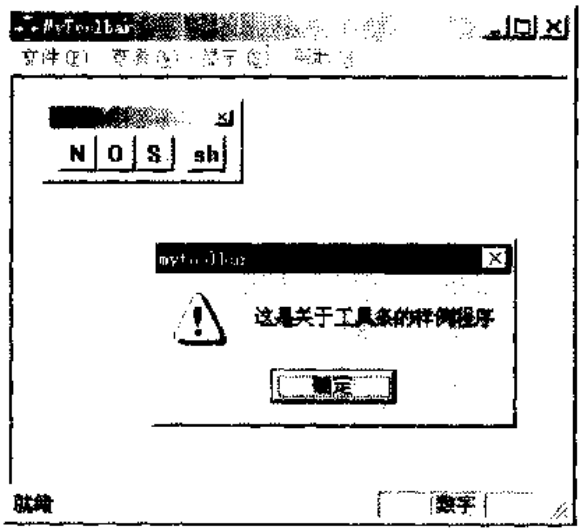


图 15-2　应用程序主窗口

1. 建立应用程序的工程文件

建立 MyToolbar 应用程序工程文件的步骤与前几章内容中建立工程文件的步骤相似，只是在 New 对话框的 Project 选项卡中的 Projects name 编辑框中输入的工程文件的名称为 MyToolBar。在 MFC AppWizard-Step 1 对话框中的“What type of application do you like to create?”中选择 Single Document 单选按钮，其他的都保留默认选项。在 MFC AppWizard-Step 4 of 6 对话框中的“What features do you like to include?”中清除 Docking Toolbar 和 Printing and print preview 多选框，其他的都保留默认选项。

2. 应用程序的可视化编程部分

(1) 对话框中的工具条的设计

此例中 AppWizard 生成的应用程序中是没有工具条的，我们需要在应用程序的资源中增加一个工具条资源。

① 选择 Insert 菜单中的 Resource 命令，显示 Insert Resource(增加资源)对话框如图15-3所示。选择其中的 Toolbar 项，使其高亮显示，单击右边的 New 按钮，此时在资源文件 MyToolbar.rc中就增加了一项资源，即一个工具条。

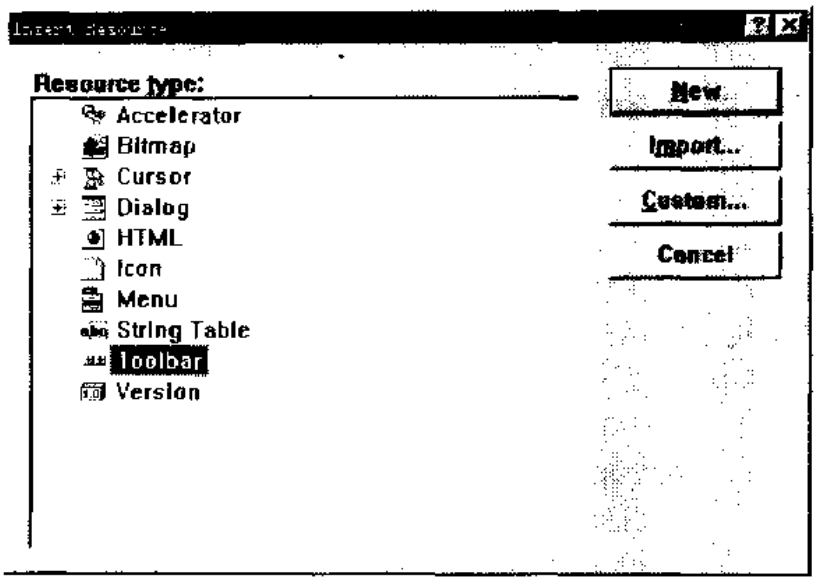


图 15-3 Insert Resource 对话框

② 如何才能在 MyToolbar.rc 中看到这个新增的工具条呢？必须打开工作区窗口，选择 View 菜单中的 Workspace 命令，显示出应用程序的工作区窗口。选择工作区窗口中的 Resource View 选项卡，这时就可以看到新增的工具条资源了。

③ 单击工作区窗口中 Toolbar 项左边的“+”号，在下面显示出该资源的 ID 号。Visual C++ 给出新增工具条默认的 ID 为 IDR_TOOLBAR1。这个由 Visual C++ 自动创建的名字（IDR_TOOLBAR1）并不具有清晰的意义，因此我们需要更改它的属性。

④ 右击 IDR_TOOLBAR1 条目，然后从弹出的快捷菜单中选择 Properties（属性），显示如图 15-4 所示的 Toolbar Properties（工具条属性）对话框。在 ID 域中输入新名字 IDR_TOOLBAR 即可。此时 File name（文件名）域中的文件名会被自动修改。

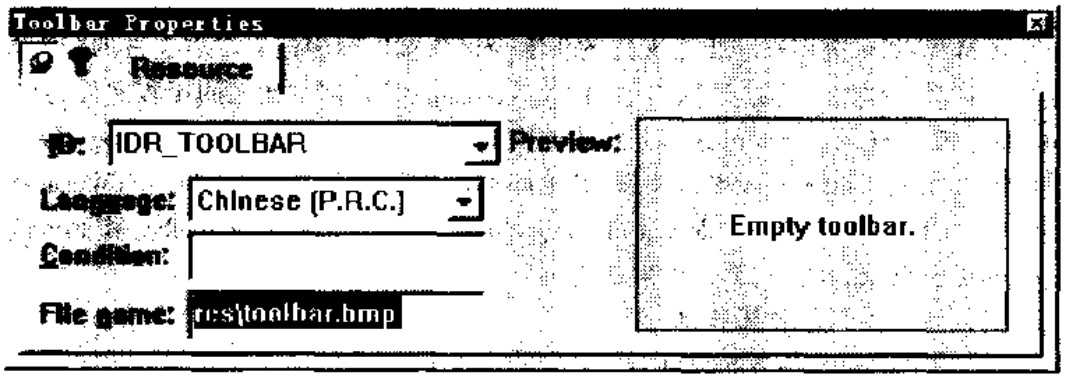


图 15-4 Toolbar Properties 对话框

⑤ 双击 ID 号，则显示一个可编辑的工具条，该工具条上有一个带虚框的矩形框，这个矩形框表示可编辑的一个工具条按钮。

（2）工具条按钮的编辑

① 加入工具条按钮

可以利用 Visual C++ 所提供的绘图工具直接进行按钮的编辑，这主要用来编辑自己的按钮。对于已经存在的图标，如果要把它作为一个按钮，可以先加入一个 ICON 资源，再通过拷贝和粘贴来放到 Toolbar 上。

② 删除工具条按钮

怎样删除工具条上的不需要的图标按钮呢？不能采用选中按钮并按 Delete 键的方法，这样只能删除按钮上的图案；正确的方法是直接用鼠标把该按钮拖出工具条以外。

③ 分隔工具条按钮

分隔工具条上的图符按钮的方法很简单，只需用鼠标将要移动的按钮拖动到工具条上需要的位置。图 15-5 所示的是一个包含四个按钮的工具条，其中 sh 按钮与前三个按钮已被分隔开。



图 15-5　包含四个按钮的工具条

④ 改变按钮的属性

首先选中工具条上的某个按钮(如第一个按钮)，然后选择 View 菜单中的 Properties 命令，将显示 Toolbar Button Properties 对话框，在其中将 ID 改为 ID-FILE-NEW，这是 Windows 默认的打开一个新文档的命令 ID。

为每个按钮设定 ID，它可以是一个新的 ID，也可以与某个菜单项 ID 相同。如果与某个菜单项 ID 相同，则可以直接激活该函数。从 Windows 工作机制来说，工具条是一个显示一系列位图按钮的子窗口，一旦创建了工具条并使其可见，它就将像菜单那样生成 WM-COMMAND 消息，因此应使工具条的命令 ID 与菜单的命令 ID 相同。这样，单击工具条上的某一位图按钮，就相当于选择了相应的菜单命令。

MFC 工具条也支持工具提示，工具提示可帮助用户理解单独一个位图按钮的作用。如允许这一特性，用户就可通过将鼠标光标移到一个工具条按钮上(并不单击按钮)来调用工具提示支持。过一会儿以后，就会在工具条按钮上方的一个小的正文窗口中显示工具提示，它是按钮功能的一个简略的文字提示。在如图 15-6 所示对话框中的 Prompt 文本框中，\n 前的文本是显示在状态栏中的文本，\n 后的文本显示为工具提示。用同样的方法设置其他的工具条按钮的 ID，如表 15-5 所示。

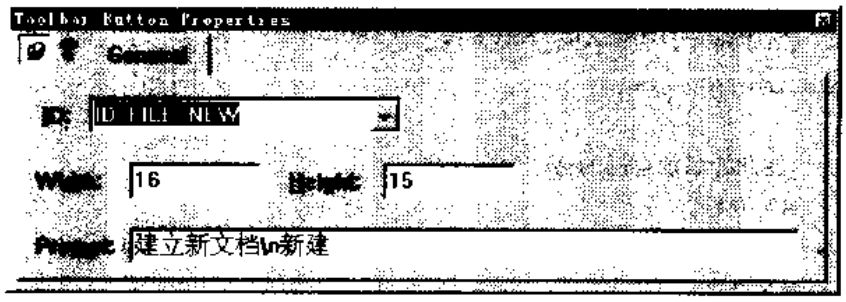


图 15-6　工具条属性

(3) 菜单的编辑

用在第12章学到的方法建立菜单项。先删除Edit及其所有的子菜单，然后在View

表 15-5 工具条按钮的属性

按 钮	ID	Prompt
N	ID_FILE_NEW	建立新文档 \ n 新建
O	ID_FILE_OPEN	打开一个现有文档 \ n 打开
S	ID_FILE_SAVE_AS	将活动文档以一个新文件名保存 \ n 另存为
sh	ID_SHOW	显示一个对话框 \ n 显示

下增加菜单项 1，在主菜单中增加菜单项 2，并把菜单项 2 的 Popup 多选框清空。菜单项 1 和菜单项 2 的其他属性如表 15-6 所示。

表 15-6 菜单的属性

菜单	ID	Caption
1	ID_VIEW_TOOLBAR	工具条(&T)
2	ID_SHOW	显示(&S)

3. 应用程序的代码编程

现在我们已经有了工具条和一些使其发挥作用的代码，但我们的实例程序中仍然缺少一个重要的特征。如果现在就运行它，会看不到工具条，因此还必须添加一些使菜单项和工具条用起来更方便的代码。

(1) 给菜单选项"菜单(S)"编写代码

① 在菜单 IDR_MAINFRAME 上单击鼠标右键，在弹出的快捷菜单中选择 ClassWizard 命令。

② 在 Object IDs 列表框中选择 ID_SHOW 项。

③ 在 Messages 列表框中选择 COMMAND 项，这里与命令按钮不同，不是 BN_CLICKED。

④ 单击右边的 Add Function 按钮，在弹出的对话框中接受建议函数名 OnShow()。

⑤ 这样在 ClassWizard 对话框的 Member functions 列表框中增加了成员函数 OnShow()，单击 Edit Code 按钮，Visual C++ 显示源代码文件(.cpp)，并将光标停在函数 OnShow() 处，等待用户定义函数的内容。

⑥ 在函数 OnShow()中输入以下代码：

```
void CMainFrame::OnShow()
{
    //TODO:在此处添加自己的命令句柄代码
    //代码编写开始
    AfxMessageBox("这是关于工具条的样例程序");//显示一个对话框
    //代码编写结束
}
```

(2) 添加工具条代码

有三个地方需要添加工具条代码。

① 首先为类 CMainFrame 加入一个成员变量 m_wndToolBar，它有两种方法。

一种方法是用 Class Wizard 进行，在 Workspace 窗口的 ClassView 选项卡中，找到 CMainFrame 类的标识，单击鼠标右键，选择 Add Member Variable 命令，在 Variable Type 编辑框中填入 CToolBar，在 Variable Declaration 编辑框中填入 m_wndToolBar，在 Access 中选择 protected 一项

另一种方法是在文件中直接添加所需的代码。在头文件 MainFrm.h 中，加入以下代码。

```
class CMainFrame : public CFrameWnd
{
      …
protected:  // control bar embedded members
    CToolBar   m_wndToolBar;      //加入一个成员变量 m_wndToolBar
      …
};
```

这段代码为类 CMainFrame 加入一个受保护的成员变量 m_wndToolBar。

② 加入代码以实现使工具条可见的功能

下一块代码出现在 MAINFRAME.CPP 文件中。这块代码设置工具条的特性，使你在启动程序时能看见它。在 CMainFrame 的 OnCreate 函数中加入以下代码：

```
// MainFrm.cpp : implementation of the CMainFrame class
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      …
    if (! m_wndToolBar.Create(this) ||                  //创建工具条
        ! m_wndToolBar.LoadToolBar(IDR_TOOLBAR))    //引入资源 IDR_TOOLBAR
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }
    …
};
```

若现在运行程序，就可以看见工具条了。

③ 更改属性

为了使工具条成为可"浮动"工具条(这样可以在应用程序中把它从一个地方移到另一个地方)，要考虑进行一些其他的编码。仍在 MAINFRAME.CPP 文件中，在 CMainFrame 的 OnCreate 函数中在刚才加入的代码后面加入以下代码：

```
// MainFrm.cpp : implementation of the CMainFrame class
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      …
      // TODO: Remove this if you don't want tool tips or a resizeable toolbar
      m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
```

```
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
    //设定工具条的风格为:当鼠标光标在按钮上暂停时,显示工具提示和命令描述,并设定工
    //具条的大小是可变的。
    // TODO: Delete these three lines if you don't want the toolbar to
    // be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
    //实现工具条的移动
    ...
}
```

MyToolBar 应用程序到现在为止,已经全部编写完毕,经过编译运行测试,完全达到设计要求。最后再说明一下,如果希望有两个工具条,只需要像刚才那样定义工具条资源并为类 CMainFrame 加入第二个成员变量 m_wndToolBar2,然后添加合适的响应程序代码即可。

习　题

15-1　工具条类的结构是如何定义的?

15-2　工具条需要哪些资源?

15-3　如何改变工具条的停靠风格?

15-4　创建一个包含有"粗体"和"下划线"三个工具条按钮的应用程序,能够对显示文本的属性进行改变。

15-5　创建一个包含有"显示 ASCII"、"显示 256 色"等工具条按钮的应用程序。选择"显示 ASCII"选项时,在编辑框中显示 0 ~ 127 的 ASCII 字符;当选择"显示 256 色"时,在窗口中出现一个填充的矩形框,而且矩形框的填充色从 0 ~ 127 动态依次变化,变化间隔为 0.5 秒。

15-6　创建一个带有"圆","矩形"按钮的工具条。选中"圆"工具条按钮,可以拖拉鼠标画一个圆;选中"矩形"工具条按钮,可以拖拉鼠标画一个矩形。

第 16 章

文档-视图结构及其应用

在前面的章节中已经介绍过了工具条，从而接触了一些带有文档界面的应用程序 Visual C++ 的 AppWizard 可根据需要直接生成可执行的文档界面的应用程序框架代码本章将介绍 Visual C++ 中的文档/视图结构的工作机制。这是一个相对比较复杂的问题，所涉及的内容也比较广泛，如文档-视图结构、使用 AppWizard 创建基于文档-视图结构的框架应用程序，以及文档类、视图类和文档模板类的应用。

16.1 概述

16.1.1 单文档界面与多文档界面

Visual C++ 中的 MFC 库支持三种不同的应用程序：单文档界面(SDI)、多文档界面(MDI)和基于对话框的应用程序。在前面的章节中已经介绍了很多基于对话框的应用程序，本章将介绍文档界面应用程序的编程方法。文档界面分为单文档界面(SDI)与多文档界面(MDI)。

从用户的角度看，SDI 的应用程序只有一个窗口。也就是说，SDI 的应用程序每次只能读写一个文件或文档，Windows 中的 Notepad(笔记本)就是 SDI 的应用程序的一个典型例子。

从用户的角度看，MDI 的应用程序每次可以读写多个文件或文档，可以同时对多个文档进行操作，可以有多个子窗口。Windows 中的 Word(文字处理软件)就是 MDI 的应用程序的典型例子。

使用 AppWizard 创建 SDI 界面的应用程序和创建 MDI 界面的应用程序的过程几乎是完全一样的。主要的差别在于：AppWizard 在创建单文档界面的应用程序时，不生成 CChildFrame 类，并且，单文档界面应用程序的 CMainFrame 类的基类为 CFrameWnd，而多文档界面应用程序的 CMainFrame 类的基类为 CMDIFrameWnd。

每一个文档可以有多个视图，但每个视图只能对应于一个确定的文档。因此，多文档程序需要解决的问题仅仅是多个文档的数据管理方法的问题。对多文档程序来说，最初的文档模板只支持主窗口，但每次打开一个新文档时都调用 CDocument 的函数 OnNewDocument，建立一个由 CMDIChildWnd 派生的新的 MDI 子窗口，这些窗口中保存着各种已打开的文档，所有的细节都由 MFC 库处理。例如，当用户单击 File 菜单下的 New 选项时，将生成另外一个文档并打开其窗口。但是，每个文档中初始时都是一片空白，因此还需加上能

对不同的窗口事件做出响应的代码。每个 MDI 子窗口都支持由类 CView 派生的普通类型的视图，这就意味着可以同以前一样不受约束地使用各种消息。

16.1.2 文档-视图结构

文档-视图结构是在 Visual C++ 中使用 MFC 开发基于文档的应用程序的基本框架。在这个框架中，数据的维护及其显示分别由两个不同但又彼此紧密相关的类——文档类和视图类负责。文档-视图结构在很多场合与传统的编程方式相比要更有利于应用程序的编写。

在文档-视图结构中，文档的任务是对数据进行管理和维护。数据通常被保存在文档类的成员变量中。在 Visual C++ 中，文档类通过一个称为串行化(serialize)的过程将数据保存到磁盘文件或数据库中。MFC 类库为数据的串行化提供了默认的支持，我们只需要在此基础上稍加修改，就可以为自定义的文档类提供串行化支持。文档类还可以处理命令消息，这里所谓的命令消息是指来自如菜单、工具栏按钮和加速键的 WM_COMMAND 通知消息。除了 WM_COMMAND 外，文档不能处理其他的 Windows 消息。由此可以看出，命令消息可以被多种对象处理，这些对象除了窗口和视图外，还可以是文档、文档模板或应用程序本身。这一点与 Windows 消息和控件通知消息不同，因为 Windows 消息和控件通知消息只能被窗口和视图处理。

视图类在文档和用户之间起中介作用。视图可以直接或间接地访问文档类中的这些成员变量，它从文档类中(而不是从存储介质中)将文档中的数据取出来，然后在屏幕上显示文档的数据。同时，它可以接收用户的输入，并接受用户的修改，通过调用文档和视图的接口将修改的信息反馈给文档类，实际的数据更新仍然由文档类来完成。

每一个文档可以有多个视图，但每个视图只能对应于一个确定的文档。也就是说，视图是文档的不同表现形式。比如，一个 *.html 文件，在浏览器里和在笔记本中打开的表现形式是不同的，但是它们操作的是同一个文件。这就是一个文档对应两个视图的例子。

总之，文档-视图结构的工作机制是：在通常情况下，视图通过 GetDocument 成员函数获得指向相关联的文档对象的指针，并通过该指针调用文档类的成员函数来从文档中读取数据。视图把数据显示于计算机屏幕上，用户通过与视图的交互来查看数据并对数据进行修改。然后，视图通过相关联的文档类的成员函数将经过修改的数据传递给文档对象。文档对象获得修改过的数据之后，对其进行必要的修改，最后保存到永久介质(如磁盘文件)中。

16.2 文档类的结构及其方法

所有的文档类都以 CDocument 类为其基类。CDocument 类提供了文档类所需要的最基本的功能实现，为文档对象以及文档和其他对象(如视对象、应用程序对象以及框架窗口等)交互的实现提供了一个框架。我们所做的工作基本上是在这个已有框架的基础上，添加与特定应用程序相关的实现代码。

1. CDocument 类

CDocument 类是从 CCmdTarget 类下派生的，它在 MFC 类库中的层次位置如图 16-1 所示。下面是 CDocument 类的定义，从本结构中，可大致了解这个类的功能。

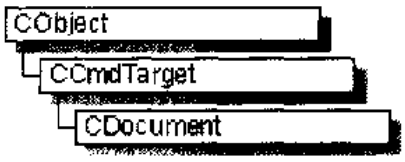


图 16-1　CDocument 类在 MFC 类库中的层次位置

```
// class CDocument is the main document data
abstraction
#ifdef _AFXDLL
class CDocument : public CCmdTarget
#else
class AFX_NOVTABLE CDocument : public CCmdTarget
#endif
{
    DECLARE_DYNAMIC(CDocument)

public:
// Constructors
    CDocument();
// Attributes
public:
    const CString& GetTitle() const;
    virtual void SetTitle(LPCTSTR lpszTitle);
    const CString& GetPathName() const;
    virtual void SetPathName(LPCTSTR lpszPathName, BOOL bAddToMRU = TRUE);
    CDocTemplate* GetDocTemplate() const;
    virtual BOOL IsModified();
    virtual void SetModifiedFlag(BOOL bModified = TRUE);
// Operations
    void AddView(CView* pView);
    void RemoveView(CView* pView);
    virtual POSITION GetFirstViewPosition() const;
    virtual CView* GetNextView(POSITION& rPosition) const;

    // Update Views (simple update - DAC only)
    void UpdateAllViews(CView* pSender, LPARAM lHint = 0L, CObject* pHint =
                        NULL);

    // Overridables
    // Special notifications
    virtual void OnChangedViewList(); // after Add or Remove view
```

```
    virtual void DeleteContents(); // delete doc items etc
// File helpers
    virtual BOOL OnNewDocument();
    virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
    virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
    virtual void OnCloseDocument();
    virtual void ReportSaveLoadException(LPCTSTR lpszPathName,
                    CException* e, BOOL bSaving, UINT nIDPDefault);
    virtual CFile* GetFile(LPCTSTR lpszFileName, UINT nOpenFlags,
          CFileException* pError);
    virtual void ReleaseFile(CFile* pFile, BOOL bAbort);

// advanced overridables, closing down frame/doc, etc.
    virtual BOOL CanCloseFrame(CFrameWnd* pFrame);
    virtual BOOL SaveModified(); // return TRUE if ok to continue
    virtual void PreCloseFrame(CFrameWnd* pFrame);

// Implementation
protected:
    // default implementation
    CString m_strTitle;
    CString m_strPathName;
    CDocTemplate* m_pDocTemplate;
    CPtrList m_viewList;          // list of views
    BOOL m_bModified;             // changed since last saved
  public:
    BOOL m_bAutoDelete;    // TRUE => delete document when no more views
    BOOL m_bEmbedded;      // TRUE => document is being created by OLE

  #ifdef DEBUG
    virtual void Dump(CDumpContext&) const;
    virtual void AssertValid() const;
  #endif //_DEBUG
    virtual ~CDocument();
// implementation helpers
    virtual BOOL DoSave(LPCTSTR lpszPathName, BOOL bReplace = TRUE);
    virtual BOOL DoFileSave();
    virtual void UpdateFrameCounts();
    void DisconnectViews();
    void SendInitialUpdate();
```

```
// overridables for implementation
    virtual HMENU GetDefaultMenu(); // get menu depending on state
    virtual HACCEL GetDefaultAccelerator();
    virtual void OnIdle();
    virtual void OnFinalRelease();
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
            AFX_CMDHANDLERINFO* pHandlerInfo);
    friend class CDocTemplate;

protected:
    // file menu commands
    //{{AFX_MSG(CDocument)
    afx_msg void OnFileClose();
    afx_msg void OnFileSave();
    afx_msg void OnFileSaveAs();
    //}}AFX_MSG
    // mail enabling
    afx_msg void OnFileSendMail();
    afx_msg void OnUpdateFileSendMail(CCmdUI* pCmdUI);
    DECLARE_MESSAGE_MAP()
};
```

2. CDocument 类的方法

CDocument 类提供了文档类所需要的最基本的功能实现。它提供的方法主要有一般方法和虚拟方法。表 16-1 是 CDocument 中的一般方法。

表 16-1　CDocument 的一般方法

方　法	说　明
GetTitle( )	获得文档标题
SetTitle( )	设置文档标题
GetPathName( )	获得文档数据文件的路径字符串
SetPathName( )	设置文档数据文件的路径字符串
GetDocTemplate( )	获得指向描述文档类型的文档模板的指针
AddView( )	向与文档相关联的视图列表添加指定的视图
RemoveView( )	从文档视图列表中删除视图
UpdateAllViews( )	通知所有视图，文档已被修改，它们应该重画
DisconnectViews( )	使文档与视图相分离
GetFile( )	获得指向 CFile 类型的指针

CDocument 类提供的虚拟方法使应用程序可以重写它们来提供 CDocument 派生类中的方法，CDocument 的虚拟方法的具体说明如表 16-2 所示。

表 16-2 CDocument 的虚拟方法

方 法	说 明
OnNewDocument( )	由 MFC 调用来建立文档
OnOpenDocument( )	由 MFC 调用来打开文档
OnSaveDocument( )	由 MFC 调用来保存文档
OnCloseDocument( )	由 MFC 调用来关闭文档
CanCloseFrame( )	确定观察文档的框架窗口是否被允许关闭
DeleteContents( )	在未撤消文档对象时删除文档数据
ReleaseFile( )	释放文件以允许其他应用程序使用
SaveModified( )	查询文档的修改状态并存储修改的文档
IsModified( )	确定文档从它最后一次存储后是否被修订过
SetModifiedFlag( )	设置文档从它最后一次存储后是否被修订过的布尔值
GetFirstViewPosition( )	获得视图列表头的位置
GetNextView( )	获得视图列表的下一个视图

16.3 构造文档类的派生类

1. 构造文档类的派生类的基本步骤

从上述内容可以看出，CDocument 类提供了文档类所需要的最基本的功能实现，它为文档对象以及文档和其他对象（如视图对象、应用程序对象以及框架窗口等）交互的实现提供了一个框架。我们在应用程序中需要做的工作基本上是在这个已有框架的基础上，添加与特定应用程序相关的方法实现。

本节将讨论如何从 CDocument 类派生自己的文档类。一般来说，从 CDocument 类派生自己的文档类所需的典型步骤为：

（1）为每一个文档类型从 CDocument 类（当然也可以是其他 CDocument 类的派生类）派生一个相应的文档类。

（2）为文档类添加成员变量。这些成员变量用来保存文档的数据，其他对象（如与文档相关联的视图）直接或间接地访问这些成员变量来读取或更新文档的数据。

（3）重载 Serialize 成员函数，实现文档数据的串行化。

如果您的应用程序只使用一种文档类型，那么，在创建应用程序工程文件时，AppWizard 已为我们完成了一部分工作。典型地，AppWizard 为应用程序框架生成一个 CDocument 类的派生类，在默认情况下该类的命名依赖于工程的名称。然后，AppWizard 在该文档类中重载了基类的几个成员函数，包括 OnNewDocument 和 Serialize 等。但是，AppWizard 在这些重载函数中只是简单地调用基类的相应函数，要根据应用程序的需要来修改它们。

2. 把文档数据保存到成员变量中

在使用文档/视图结构的应用程序中，通常使用文档类的成员变量来保存文档的数据，并使其他的对象（如与文档相关联的视图）可以访问这些成员变量，从而实现了文档和其他对象（主要是视图）的相互搭配使用。

3. 串行化

无论是保存文档或是打开文档，应用程序都是通过调用文档类的 Serialize 成员函数来完成串行化操作的。因此，在大多数情况下，都要通过重载 Serialize 成员函数来实现文档的串行化。Serialize 成员函数带有一个 CArchive 类型的参数，这是一个与所打开的文件相关联的对象。一般情况下，我们总是使用 CArchive 对象来保存和打开文档。

CArchive 对象是单向的，也就是说，同一个 CArchive 对象只能用于保存或读取两者之一，不能通过同一个 CArchive 对象既进行文档的保存，又进行文档的读取。在框架创建 CArchive 对象时，已根据用户选择的是"保存"("另存为")还是"打开"设置了 CArchive 对象的类型。可以使用 CArchive 类的成员函数 IsStoring 来检索当前 CArchive 对象的类型，从而得知用户所期望的操作是保存还是读取，从而执行不同的操作。

16.4　视图类的结构及其方法

1. CView 类

视图类(CView)是从 CWnd 类下派生的，它在 MFC 类库中的层次位置如图 16-2 所示。从图 16-2 中可见，CView 类的基类为 CWnd。掌握这个继承关系对我们来说是很有用的。例如，由于所有的从 CView 类派生的类也都是由 CWnd 类间接派生的，所有的这些类都具有 CWnd 的所有功能，如创建、移动、显示和隐藏窗口，因此 CView 类可以接收任何 Windows 消息，而 Cocument 类则不行。

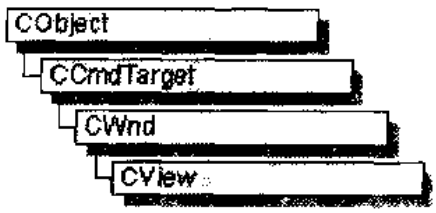


图 16-2　CView 类在 MFC 类库中的层次位置

下面是 CView 类的定义，从本结构中，可大致了解这个类的功能。

```
class CView : public CWnd
#else
class AFX_NOVTABLE CView : public CWnd
#endif
{
    DECLARE_DYNAMIC(CView)
// Constructors
protected:
    CView();
// Attributes
public:
    CDocument * GetDocument() const;
// Operations
public:
    // for standard printing setup (override OnPreparePrinting)
    BOOL DoPreparePrinting(CPrintInfo * pInfo);
// Overridables
public:
    virtual BOOL IsSelected(const CObject * pDocItem) const; // support for OLE
```

```
        // OLE scrolling support (used for drag/drop as well)
    virtual BOOL OnScroll(UINT nScrollCode, UINT nPos, BOOL bDoScroll = TRUE);
    virtual BOOL OnScrollBy(CSize sizeScroll, BOOL bDoScroll = TRUE);
    // OLE drag/drop support
    virtual DROPEFFECT OnDragEnter(COleDataObject * pDataObject,
            DWORD dwKeyState, CPoint point);
    virtual DROPEFFECT OnDragOver(COleDataObject * pDataObject,
            DWORD dwKeyState, CPoint point);
    virtual void OnDragLeave();
    virtual BOOL OnDrop(COleDataObject * pDataObject,
            DROPEFFECT dropEffect, CPoint point);
    virtual DROPEFFECT OnDropEx(COleDataObject * pDataObject,
            DROPEFFECT dropDefault, DROPEFFECT dropList, CPoint point);
    virtual DROPEFFECT OnDragScroll(DWORD dwKeyState, CPoint point);
    virtual void OnPrepareDC(CDC * pDC, CPrintInfo * pInfo = NULL);
    virtual void OnInitialUpdate(); // called first time after construct
protected:
        // Activation
        virtual void OnActivateView(BOOL bActivate, CView * pActivateView,
                                    CView * pDeactiveView);
        virtual void OnActivateFrame(UINT nState, CFrameWnd * pFrameWnd);
        // General drawing/updating
        virtual void OnUpdate(CView * pSender, LPARAM lHint, CObject * pHint);
        virtual void OnDraw(CDC * pDC) = 0;
        // Printing support
        virtual BOOL OnPreparePrinting(CPrintInfo * pInfo);
                // must override to enable printing and print preview
        virtual void OnBeginPrinting(CDC * pDC, CPrintInfo * pInfo);
        virtual void OnPrint(CDC * pDC, CPrintInfo * pInfo);
        virtual void OnEndPrinting(CDC * pDC, CPrintInfo * pInfo);
        // Advanced: end print preview mode, move to point
        virtual void OnEndPrintPreview(CDC * pDC, CPrintInfo * pInfo, POINT
                                       point,
            CPreviewView * pView);
// Implementation
public:
        virtual ~CView();
# ifdef DEBUG
        virtual void Dump(CDumpContext&) const;
        virtual void AssertValid() const;
# endif //-DEBUG
        // Advanced: for implementing custom print preview
```

```
        BOOL DoPrintPreview(UINT nIDResource, CView* pPrintView,
              CRuntimeClass* pPreviewViewClass, CPrintPreviewState* pState);
        virtual void CalcWindowRect(LPRECT lpClientRect,
                UINT nAdjustType = adjustBorder);
        virtual CScrollBar* GetScrollBarCtrl(int nBar) const;
        static CSplitterWnd* PASCAL GetParentSplitter(
              const CWnd* pWnd, BOOL bAnyState);
protected:
        CDocument* m_pDocument;
public:
        virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
              AFX_CMDHANDLERINFO* pHandlerInfo);
protected:
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        virtual void PostNcDestroy();
        // friend classes that call protected CView overridables
        friend class CDocument;
        friend class CDocTemplate;
        friend class CPreviewView;
        friend class CFrameWnd;
        friend class CMDIFrameWnd;
        friend class CMDIChildWnd;
        friend class CSplitterWnd;
        friend class COleServerDoc;
        friend class CDocObjectServer;

        //{{AFX_MSG(CView)
        afx_msg int OnCreate(LPCREATESTRUCT lpcs);
        afx_msg void OnDestroy();
        afx_msg void OnPaint();
        afx_msg int OnMouseActivate(CWnd* pDesktopWnd, UINT nHitTest, UINT mes-
                                    sage);
        // commands
        afx_msg void OnUpdateSplitCmd(CCmdUI* pCmdUI);
        afx_msg BOOL OnSplitCmd(UINT nID);
        afx_msg void OnUpdateNextPaneMenu(CCmdUI* pCmdUI);
        afx_msg BOOL OnNextPaneCmd(UINT nID);
        // not mapped commands - must be mapped in derived class
        afx_msg void OnFilePrint();
        afx_msg void OnFilePrintPreview();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
```

2. CView 类的方法

CView 类提供了文档类所需要的最基本的功能实现。它提供的方法分为一般方法和虚拟方法。表 16-3 是 CView 类的一般方法。

表 16-3　CView 类的一般方法

方　法	说　明
GetDocument( )	获得指向与视图相关联的文档的指针
DoPreparePrinting( )	初始化视窗

CView 类提供的虚拟方法使应用程序可以重写它们来提供 CView 派生类中的方法，CView 类的虚拟方法具体说明如表 16-4 所示。

表 16-4　CView 类的主要虚拟方法

方　法	说　明
IsSelected( )	确定文档是否被选中
OnScroll( )	当用户滚动时，CView 的响应
OnInitialUpdate( )	在类第一次构造后由 MFC 调用
OnDraw( )	允许文档由 MFC 调用
OnUpdate( )	由 MFC 调用对文档的修改进行响应
OnPrepareDC( )	在调用 OnDraw( )前允许由 MFC 调用修改设备描述表

此外，还有很多有关打印的 CView 的虚拟方法，如：OnPreparePrinting( )，OnBeginPrinting( )，OnPrint( )，OnEndPrinting( )，OnEndPrintPreview( )等。

一个视图类可以直接接受文档的数据，也可以通过成员函数的调用来实现。

当一个类的数据发生变化时，该文档所关联的视图类通常通过调用函数 CDocument::UpdateAllViews 来做出响应。这个函数是维护数据正确显示的常用手段。

另外，视图类中的两个成员函数 OnDraw(该函数在屏幕发生变化或因为焦点的变化需要重绘时调用；没有该函数，就不可能在程序的切换后保证屏幕的正确显示)和 OnUpdate 则对维护程序的正确显示负有重要的责任。(当一个文档发生变化时，绝大多数情况下总是希望其各个视图中的数据能得到同步的正确显示。)函数 UpdateAllViews 则是实现单文档多视图程序所不可缺少的手段。(在一个文档的任一视图发生变化时，通过该函数实现各视图的正确显示。)

16.5　视图类的派生类

本节按照各类在 MFC 中的层次讲解。由于面向对象语言的自身特性，父类所具有的特征，其派生出来的子类一般也具有，因此，掌握各类的派生关系，往往是深刻理解其行为的一个捷径。同时，我们也应该看到，由于各类与其父类的差别，它们又具有其自身的特性，而把握这些特性，又成为进一步理解其独特特征的必由之路。下面，我们分类简单地介绍一下视图类的各派生类的继承及成员函数情况。

1. CEditView 类

CEditView 类主要被设计用来支持类似编辑控件所要实现的功能，常见的文本操作基本上都是由该类支持实现的。值得注意的是，该类的直接基类不是 CView 类，而是 CCtrlView 类。

2. CRichEditView 类

该类主要提供 Rich 文本操作的支持（Rich 文本是既可以为文本，也可以为图形的一种特殊格式文本。）

3. CTreeView 类

该类主要提供一些树型控件所实现的功能的支持。它使一种数据的显示方式可以更富于变化。

4. CListView 类

该类与 CTreeView 类一样，更多的好处在于提供了一种简捷地实现数据的不同显示的途径，为数据的组织提供多种手段。

5. CScrollView 类

该类也是一个比较重要的类，但它主要提供视图的滚动显示。同时需要注意的是，该类的直接基类是 CView 类，这决定了其动作特点的特殊性。该类的具体细节请参考系统帮助。

读者也许会注意到，应用程序中可以用任何 CView 或 CView 的派生类作为本应用中视图类的基类，这就是为什么要重点考虑类选择的意义之所在。如果接受了 Visual C++ 的默认设置，那么应用程序对文档的操作都必须编写代码。当你要创建一种全新的文档时，上述设置并没有任何不妥，但绝大多数应用程序并非要创建一种全新的文档，而是建立一个与其他应用程序相似、并提供其他应用程序所缺乏特性的应用程序。选择合适的 CView 的派生类作为应用程序的基类将省去许多工作，原因在于该应用程序一经设计就

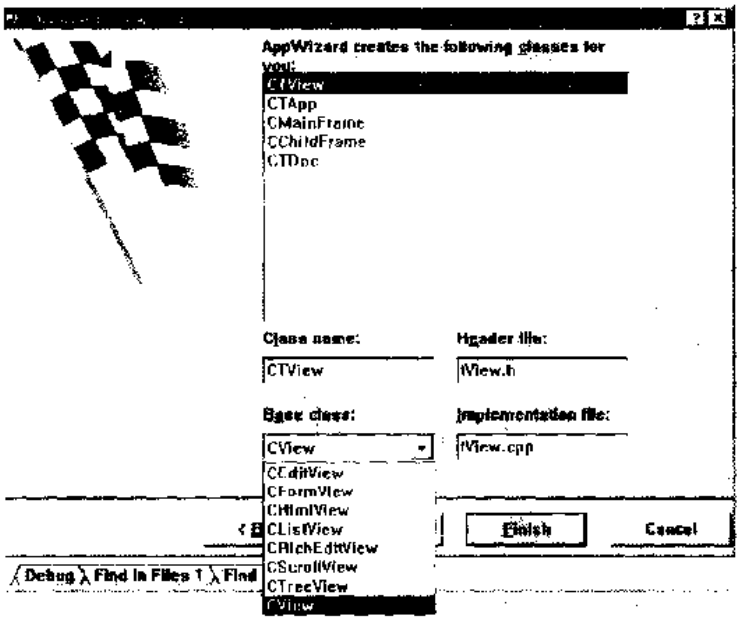


图 16-3　选择一个基类

已经知道应该如何完成任务。基类的选择可以在 MFC AppWizard 的第 6 步进行，如图 16-3 所示。

可以从 Base class 下拉列表框中为即将生成的应用程序自己的视图类选择一个合适的基类，这样可以节省很多编程工作。

16.6 文档模板类的结构及其方法

1. CDocTemplate 类

文档模板类(CDocTemplate)是从 CCmdTarget 类下派生的，它在 MFC 类库中的层次位置如图 16-4 所示。

从图 16-4 中可见，CDocTemplate 类的基类为 CCmdTarget。下面是 CDocTemplate 类的定义，从本结构中，可大致了解这个类的功能。



图 16-4 CDocTemplate 类在 MFC 类库中的层次位置

```
// class CDocTemplate creates documents
#ifdef _AFXDLL
class CDocTemplate : public CCmdTarget
#else
class AFX_NOVTABLE CDocTemplate : public CCmdTarget
#endif
{
        DECLARE_DYNAMIC(CDocTemplate)
// Constructors
protected:
        CDocTemplate(UINT nIDResource, CRuntimeClass* pDocClass,
              CRuntimeClass* pFrameClass, CRuntimeClass* pViewClass);
public:
        virtual void LoadTemplate();
// Attributes
public:
        // setup for OLE containers
        void SetContainerInfo(UINT nIDOleInPlaceContainer);
        // setup for OLE servers
        void SetServerInfo(UINT nIDOleEmbedding, UINT nIDOleInPlaceServer = 0,
        CRuntimeClass* pOleFrameClass = NULL, CRuntimeClass* pOleViewClass =
        NULL);
        // iterating over open documents
        virtual POSITION GetFirstDocPosition() const = 0;
        virtual CDocument* GetNextDoc(POSITION& rPos) const = 0;
// Operations
public:
```

```
        virtual void AddDocument(CDocument * pDoc);      // must override
        virtual void RemoveDocument(CDocument * pDoc);   // must override

        enum DocStringIndex
        {
        windowTitle,        // default window title
        docName,            // user visible name for default document
        fileNewName,        // user visible name for FileNew
        // for file based documents:
        filterName,         // user visible name for FileOpen
        filterExt,          // user visible extension for FileOpen
        // for file based documents with Shell open support:
        regFileTypeId,      // REGEDIT visible registered file type identifier
        regFileTypeName,    // Shell visible registered file type name
       };
       virtual BOOL GetDocString(
             CString& rString,
             enum DocStringIndex index) const; // get one of the info strings
      CFrameWnd * CreateOleFrame(CWnd * pParentWnd, CDocument * pDoc,
                                                    BOOL  bCreateView);
// Overridables
public:
        enum Confidence
        {
             noAttempt,
             maybeAttemptForeign,
             maybeAttemptNative,
             yesAttemptForeign,
             yesAttemptNative,
             yesAlreadyOpen
        };
        virtual Confidence MatchDocType(LPCTSTR lpszPathName,
                                  CDocument * & rpDocMatch);
        virtual CDocument * CreateNewDocument();
       virtual CFrameWnd * CreateNewFrame(CDocument * pDoc, CFrameWnd * pOther);
        virtual void InitialUpdateFrame(CFrameWnd * pFrame, CDocument * pDoc,
               BOOL bMakeVisible = TRUE);
        virtual BOOL SaveAllModified();     // for all documents
        virtual void CloseAllDocuments(BOOL bEndSession);
        virtual CDocument * OpenDocumentFile(
               LPCTSTR lpszPathName, BOOL bMakeVisible = TRUE) = 0;
                      //open named file
```

```
                        //if lpszPathName == NULL -> create new file with this type
        virtual void SetDefaultTitle(CDocument * pDocument) = 0;
// Implementation
public:
        BOOL m_bAutoDelete;
        virtual ~CDocTemplate();
        // back pointer to OLE or other server (NULL if none or disabled)
        CObject * m_pAttachedFactory;
        // menu & accelerator resources for in-place container
        HMENU m_hMenuInPlace;
        HACCEL m_hAccelInPlace;
        // menu & accelerator resource for server editing embedding
        HMENU m_hMenuEmbedding;
        HACCEL m_hAccelEmbedding;
        // menu & accelerator resource for server editing in-place
        HMENU m_hMenuInPlaceServer;
      HACCEL m_hAccelInPlaceServer;

#ifdef _DEBUG
        virtual void Dump(CDumpContext&) const;
        virtual void AssertValid() const;
#endif
        virtual void OnIdle();          // for all documents
        virtual BOOL OnCmdMsg(UINT nID, int nCode, void * pExtra,
                AFX_CMDHANDLERINFO * pHandlerInfo);

protected:
      UINT m_nIDResource;               // IDR_ for frame/menu/accel as well
      UINT m_nIDServerResource;         // IDR_ for OLE inplace frame/menu/accel
      UINT m_nIDEmbeddingResource;      // IDR_ for OLE open frame/menu/accel
      UINT m_nIDContainerResource;      // IDR_ for container frame/menu/accel
      CRuntimeClass * m_pDocClass;      // class for creating new documents
      CRuntimeClass * m_pFrameClass;    // class for creating new frames
      CRuntimeClass * m_pViewClass;     // class for creating new views
      CRuntimeClass * m_pOleFrameClass; // class for creating in-place frame
      CRuntimeClass * m_pOleViewClass;  // class for creating in-place view
      CString m_strDocStrings;     //'\n' separated names
                // The document names sub-strings are represented as one string:
                // windowTitle\ndocName\n...(see DocStringIndex enum)
};
```

2. CDocTemplate 类的方法

CDocTemplate 类提供了文档模板类所需要的最基本的功能实现，它提供的方法分为

一般方法和虚拟方法，如表 16-5 所示。

表 16-5　CDocTemplate 类的一般方法和虚拟方法

方　法	说　明
GetDocString( )	获得与文档相关的字符串
LoadTemplate( )	加载指定的模板
AddDocument( )	给文档模板添加指定的文档
RemoveDocument( )	从文档模板列表中删除文档
GetFirstDocPosition( )	获得与文档模板相关的第一个文档的位置
GetNextDoc( )	获得文档及下一个文档
CreateNewDocument( )	建立文档
CreateNewFrame( )	建立包含文档和视图的框架窗口
OpenDocumentFile( )	打开由路径名指定的文档
CloseAllDocument( )	关闭所有文档
SetDefaultTitle( )	显示文档窗口的标题栏中默认的标题
SaveAllModified( )	查询文档模板的修改状态并存储与之关联的所有文档

CDocTemplate 提供的虚拟方法使应用程序可以重写它们来提供 CDocTemplate 派生类中的方法。

3．文档模板 CDocTemplate 类的功能

在 Visual C++ 中，文档类、与文档类相关联的视图类以及为视图类提供显示的框架窗口都是由文档模板创建的。每一种文档类型都有一种文档模板与之相对应，文档模板负责创建和管理该文档类型的所有文档。文档、框架窗口和视图的创建过程之间的先后顺序为：创建文档 → 创建框架窗口 → 创建视图。

文档、视图和框架三者之间是相互关联、相互协调的，彼此都包含了指向对方的指针。它们之间的联系是通过文档模板的构造函数来实现的。文档模板的构造函数的原型如下：

```
CDocTemplate(UINT nIDResource,
             //nIDResource 为和文档类型一同使用的各种资源的资源标识符
             CRuntimeClass * pDocClass,
             //pDocClass 是指向 CDocument 的派生文档类的 CRuntimeClass 对象的指针
             CRuntimeClass * pFrameClass,
             //pFrameClass 是指向 CFrameClass 的派生框架类的 CRuntimeClass 对象的
             //指针
             CRuntimeClass * pViewClass
             //pViewClass 是指向 CView 的派生视图类的 CRuntimeClass 对象的指针
             );
```

在应用程序的 InitInstance( ) 方法中，通过使用文档模板的构造函数创建文档模板的对象：

```
BOOL CMymdiApp::InitInstance()
{
...
    CMultiDocTemplate * pDocTemplate2;
```

```
    pDocTemplate2 = new CMultiDocTemplate(
        IDR_MYMDITYPE2,
        RUNTIME_CLASS(CMymdiDoc2), // MDI 派生文档类的 CRuntimeClass 对象的指针
        RUNTIME_CLASS(CChildFrame), // MDI 派生子框架类的 CRuntimeClass 对象的指针
        RUNTIME_CLASS(CMymdiView2));// 创建文档模板的对象
//然后,使用 CWinApp::AddDocTemplate()方法将新模板添加到应用程序的文档模板列表中
    AddDocTemplate(pDocTemplate2);
    ...
}
```

文档模板字符串在资源文件中的定义描绘了文档模板的信息,它由至多九个子串组成　其格式如下:

```
nIDResource <WindowTitle>  \n          //在 SDI 应用程序中的标题栏上的标题
            <DocName> \n               //新文档的默认名
            <FileNewName> .n           //文档类型的名字
            <FilterName> \n            //匹配此类型文档的文件过滤器
            <FilterExt> \n             //文档的文件扩展名
            <RegFileTypeID> \n         //存储在寄存器中的文档类型的标识符
            <RegFileTypeName> \n       //存储在寄存器中的文档类型的标识符
            <FilterMacExt(FilterWinExt)> \n
            //允许从 FilterName 删除的文件扩展名
            <FilterMacName(FilterWinName)>
            //允许 Macintosh 的文件类型被指定在 FilterExt 中
```

利用 MFC 应用程序向导,这个工作就简化了许多。在 MFC 应用程序向导的第四步

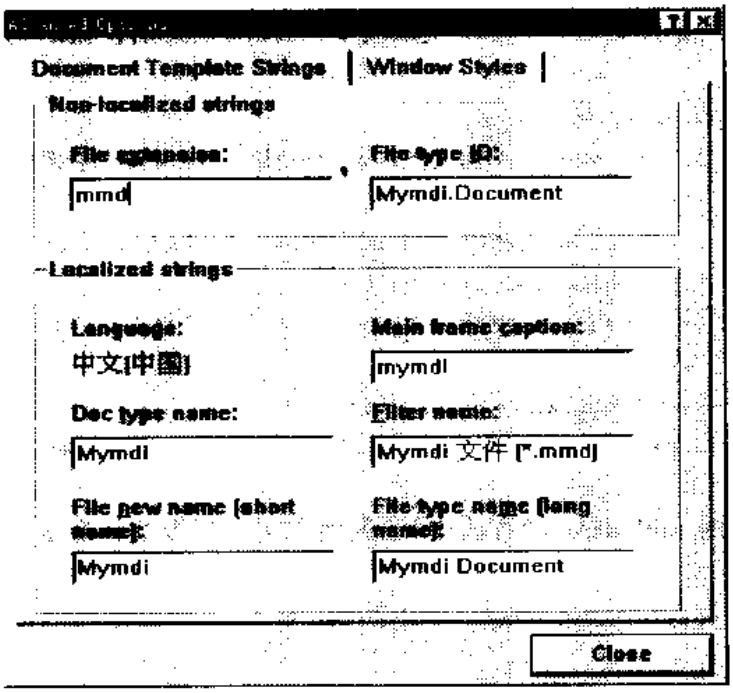


图 16-5　Advanced Options 对话框

MFC AppWizard-Step 4 of 6 对话框中有一个 Advanced 按钮。单击这个按钮，系统显示如图 16-5 所示的 Advanced Options 对话框，在这个对话框中可以设置文档参数以及用户使用时显示在标题条中的文本。在标题条中也将显示当前文档的名称。如果按图 16-5 所示设置 Advanced Options 对话框，在 File extention 对话框中输入 mmd，完成后的应用程序的文件将使用 .mmd 为扩展名，应用程序的标题为 mymdi。每次显示 FileOpen 或 File Save 对话框时，过滤器域显示为“Mymdi 文件(* .mmb)”。最后，当显示该应用程序创建的文档的属性对话框时，对话框将显示该文档的类型为 mymdi-mymdit。

16.7　SDI 与 MDI 编程实例

【例 16-1】

创建一个应用程序，其界面的标题为 MySdi。在应用程序的主窗口中显示一行文本“您好，单文档界面的例程!”，并始终出现在窗口的中央。在“编辑”菜单上有一个菜单项“改变显示文本”，单击该项可以弹出一个对话框，通过这个对话框可以改变主窗口中的显示文本。

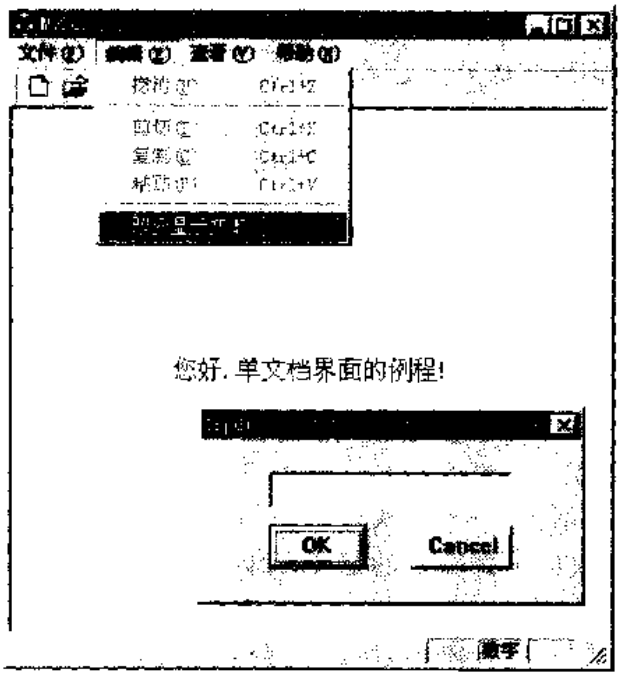


图 16-6　应用程序主窗口

(1) 创建工程文件

首先创建一个 MFC AppWizard (exe)工程文件，并取名为 Mysdi。建立 Mysdi 应用程序工程文件的步骤与前几章内容中建立工程文件的步骤相似，只是在 New 对话框的 Project 选项卡中的 Projects name 编辑框中输入的工程文件的名称为 Mysdi。

在 MFC AppWizard-Step 1 对话框中(该对话框是生成工程文件的第一步)，在 What type of application do you like to create? 中选择 Single Document 选项，其他的都保留默认选项，然

后单击 Finish 按钮，建立应用程序的工程文件。

(2) 添加资源

为程序添加如图16-7所示的对话框，然后使用ClassWizard为对话框生成CDialog类的派生类 CDlgInput，并为其中的 Edit Box 控件(其 ID为 IDC_EDIT1)添加相关联的成员变量 m_input，其类型为 CString。

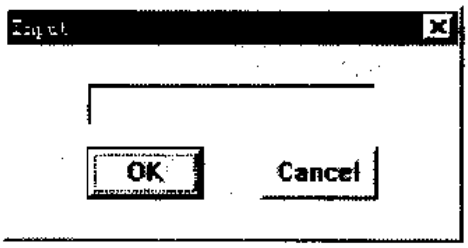


图 16-7 对话框资源

(3) 为 CMysdiDoc 类添加成员变量

为 CMysdiDoc 类添加成员变量可以有两种方法。

第一种方法是利用 Visual C++ 的可视化编程工具。具体步骤如下：在 Workspace 窗口的 ClassView 选项卡中展开 Mysdi classes，可以看到 AppWizard 为 Mysdi 程序生成的所有类。右击 CMysdiDoc 类，在弹出菜单中单击 Add Member Variable 选项，在 Variable Type 框中输入成员变量的类型 CString，在 Variable Declaration 框中输入成员变量名 m_str。由于我们希望其他类的对象可以访问该成员变量，因此在 Access 框中选择其访问类型为 Public。单击 OK，Visual C++ 将该成员变量的定义添加到类的定义中。

第二种方法是手动地将成员变量添加到类 CMysdiDoc 的定义中。在 Workspace 窗口中的 FileView 选项卡中展开 Mysdi files\Header Files。双击 MysdiDoc.h，Visual C++ 将在代码编辑窗口中打开文件 MysdiDoc.h，这个文件包括了 Mysdi 应用程序中的文档类 CMysdiDoc 的定义。

在 CMysdiDoc 的定义中，找到下面的程序代码段

```
// Attributes
public:
```

在此之后手工输入：

```
CString m_str;
```

在类 CMysdiDoc 的定义中包括多个 public 块。事实上，可以把这些定义都放到同一个 public 块中，在定义中包括多个 public 块只是为了区别并不同用途的公有成员。例如，在上面所示的代码中，将成员变量 m_str 的定义放到了 Attributes 块内；而如果使用 Add Member Variable 命令添加变量的话，Visual C++ 会将它添加到 Implementation 块中。这只是为了便于程序的阅读和维护，对于编译器而言，将公有成员的定义放到哪一个 public 块中其结果都是一样的，对于私有成员和保护成员也是一样。

(4) 文档变量初始化

为了测试该程序，我们在 CMysdiDoc 的 OnNewDocument 成员变量中为公有成员 m_str 赋以初值“您好，单文档界面的例程!”，方法是使用下面的代码替换 OnNewDocument 的实现代码中的“//TODO”注释：

```
BOOL CMymdiDoc2::OnNewDocument()
{
    if (! CDocument::OnNewDocument())
```

```
        return FALSE;
    m_str = "您好，单文档界面的例程!";
    return TRUE;
}
```

(5) 视图的输出

在 MFC 应用程序中，文档类是和视图类一起协作以完成应用程序功能的。下面我们将为 Mysdi 程序的视图类 CMysdiView 类的 OnDraw 成员函数添加一些代码，以将文档类中的 m_str 成员变量的内容显示到视图的框架窗口中。

```
void CMysdiView::OnDraw(CDC * pDC)
{
    CMysdiDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    CRect rectClient;
    GetClientRect(rectClient);                    // 获取当前客户区的指针
    CSize sizeClient = rectClient.Size();         // 获取当前客户区的大小
    CString str = pDoc->m_str;                    // 从文件中读取数据
    CSize sizeTextExtent = pDC->GetTextExtent(str);  // 用新选定的字体绘制字符串
    pDC->TextOut((sizeClient.cx-sizeTextExtent.cx)/2,(sizeClient.cy-sizeTex-
                tExtent.cy)/2,str);
}
```

(6) 文档串行化

下面，我们为示例程序 Mysdi 添加保存及打开文档的实现代码。

在此之前，首先应在“编辑”菜单中添加一条改变文档内容的选项，用户可以将对文档显示文本所做的修改保存到一个磁盘文件中，具体的实现方法如下：

① 单击 Workspace 窗口中的 ResourceView 选项卡，展开 Mysdi resources | Menu，双击 IDR_MAINFRAME，为“编辑”菜单添加一个菜单项“改变显示文本”，并设置新添加的菜单项的 ID 为 ID_EDIT_CHANGETEXT。

② 为示例程序添加如图 16-7 所示的对话框，然后使用 ClassWizard 为对话框生成 CDialog 类的派生类 CInputDlg，并为其中的 Edit Box 控件(其 ID 为 IDC_EDIT1)添加相关联的成员变量 m_input，其类型为 CString。

③ 使用 ClassWizard 在类 CMysdiDoc 中为菜单项“编辑|改变显示文本”(ID 为 ID_EDIT_CHANGETEXT)添加处理函数 OnEditChangeText，其代码如下：

```
void CMysdiDoc::OnEditChangeText()
{
    // TODO: Add your command handler code here
    CInputDlg inputDlg;                  //建立一个 CInputDlg 类的对象 inputDlg
    if( inputDlg.DoModal() == IDOK )     //使 inputDlg 成为一个模态对话框
    {
        m_str = inputDlg.m_input;        //获取输入的字符串
```

```
        UpdateAllViews(NULL);      //更新视图
    }
```

为使 CInputDlg 类在 CMysdiDoc 类中成为可识别的，必须在 mysdiDoc.cpp 文件中加入 CInputDlg 类的说明头文件 InputDlg.h：

```
// mysdiDoc.cpp : implementation of the CMysdiDoc class
#include "InputDlg.h" //加入头文件
```

④ 为了把这些修改保存到磁盘文件中，并在需要时可以打开所保存的磁盘文件读取文档，需要重载 CExampleDoc 类的 Serialize 函数来完成串行化。重载后的 Serialize 函数的代码如下：

```
// CMysdiDoc serialization
void CMysdiDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        { ar << m_str; }    //保存文档内容
    else
        { ar >> m_str; }    //读取文档内容
}
```

【例 16-2】

创建一个标题为 My MDI Test-Mymdi1 的多文档应用程序。程序运行后，可以打开若干个文档。如图 16-8 所示是打开的两个文档。其中，My MDI Test 是主窗口的标题，Mymdi1 是当前打开的活动文档的文档名。Mymdi21 是另一个文档，其功能与上一节的单文档程序的功能是相同的，它在应用程序的窗口中显示“您好，多文档界面的例程!”，并始终出现在窗口的中央，而且这句话是可以改变的。本应用程序还可以进行最简单的文本处理，是一个小型的文本编辑器。如果活动文档的类型改变，那么菜单的“编辑”项的子菜单会改变，工具条上的“打印”按钮的激活状态也不同，Mymdi1 的文档是可以打印的，而 Mymdi21 的文档不允许打印。

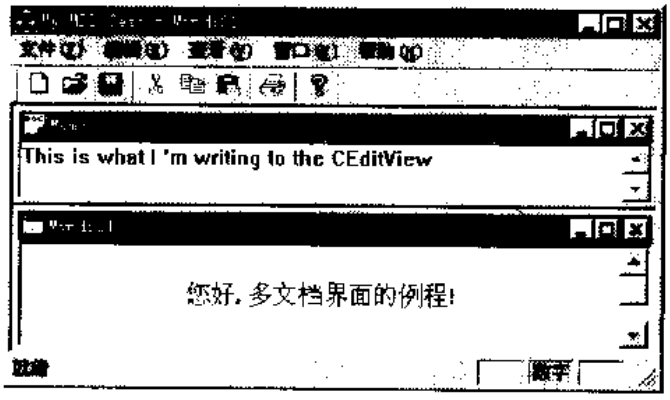


图 16-8　应用程序主窗口

1. 创建 MDI 工程文件

首先创建一个 MFC AppWizard (exe)工程文件，并取名为 Mymdi。建立 Mymdi 应用程序工程文件的步骤与前几章内容中建立工程文件的步骤相似，只是在 New 对话框的 Project 选

项卡中的 Projects name 编辑框中输入的工程文件的名称为 Mymdi。在 MFC AppWizard-Step 4 of 6 对话框中(该对话框是生成工程文件的第四步)单击 Advanced 按钮,按图 16-5 设置,其他选项都保留默认设置。在 MFC AppWizard-Step 6 of 6 对话框中(该对话框是生成工程文件的第六步)为 CmymdiView 类设置基类为 CEditView。其他选项都保留默认设置。

2. 创建第二种文档和视图类

在 View 菜单中选择 Class Wizard 选项。在 MFC ClassWizard 对话框的 Class Info 选项卡中单击 Add Class 按钮,选择 New 选项,在出现的对话框中按图 16-9 进行设置。

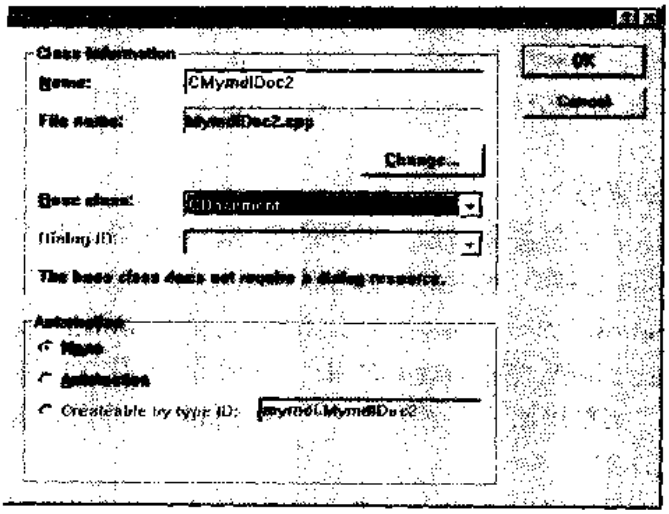


图 16-9　加入新类

单击 OK 按钮,则应用程序中增加了一种 CDocument 的派生类 CMymdiDoc2。用同种方法在应用程序中增加一种 CScrollView 的派生类 CMymdiView2。

3. 创建资源

(1) 文档、视图、框架窗口共同的资源 ID

在 Resource.h 文件中,手工加入下列代码:

```
#define IDR_MYMDITYPE2  135
```

这样就定义了文档、视图、框架窗口共同的资源 ID,以后定义的菜单、文档模板等资源均可以使用这个 ID。

(2) 文档模板资源

在 16.4.3 讲过,文档模板资源的格式是

```
nIDResource   <WindowTitle> \n
              <DocName> \n
              <FileNewName> \n
              <FilterName> \n
              <FilterExt> \n
              <RegFileTypeID> \n
              <RegFileTypeName> \n
              <FilterMacName(FilterWinName)>
```

对于第一个文档，我们利用应用程序向导直接产生了一个文档模板，现在必须手工地运用这种格式加入资源。具体的方法是在 mymdi.rc 文件中，加入下列代码：

```
IDR-MYMDITYPE2  "\nMymdi2\nMymdi2\nMymdi2 文件
(*.mm2)\n.mm2\nMymdi2.Document\nMymdi2 Document"
```

此外，为了使 Windows 能够识别.mm2 的文件，在 mymdi.reg 文件中，加入下列代码：

```
HKEY CLASSES-ROOT\.mm2 = Mymdi2.Document
HKEY-CLASSES-ROOT\Mymdi2.Document\shell\open\command = MYMDI.EXE %1
HKEY-CLASSES ROOT\Mymdi2.Document\shell\open\ddeexec = [open("%1")]
HKEY-CLASSES ROOT\Mymdi2.Document\shell\open\ddeexec\application = MYMDI
```

这样就在 Windows 的注册表中进行了注册。请特别注意黑体的文字，并与文档模板的资源说明进行比较。

(3) 菜单、对话框资源

为了实现可视化的程序，必须添加一系列的资源。用户可以将对文档显示文本所做的修改保存到一个磁盘文件中，需要的时候再从磁盘文件中打开并恢复该文档。

① 单击 Workspace 窗口中的 ResourceView 选项卡，展开 Mymdi resources|Menu，加入 ID 为 IDR-MYMDITYPE2 的菜单，使它与现有的 ID 为 IDR-MYMDITYPE 的菜单完全相同，然后为这个菜单的“编辑”菜单添加一个菜单项“改变显示文本”，并设置新添加的菜单项的 ID 为 ID-EDIT CHANGETEXT。

② 用与以前完全相同的方法，为程序添加如图 16-4 所示的对话框。然后使用 ClassWizard 为对话框生成 CDialog 类的派生类 CDlgInput，并为其中的 Edit Box 控件（其 ID 为 IDC-EDIT1）添加相关联的成员变量 m-input，其类型为 CString。

4. 代码编辑

(1) 创建文档模板类

用在 16.4.3 中描述的方法，在应用程序的 InitInstance()方法中，通过使用文档模板的构造函数创建文档模板的对象如下：

```
BOOL CMymdiApp::InitInstance()
{
...
    CMultiDocTemplate* pDocTemplate2;
        pDocTemplate2 = new CMultiDocTemplate(
        IDR-MYMDITYPE2,
        RUNTIME-CLASS(CMymdiDoc2), // MDI 派生文档类的 CRuntimeClass 对象的指针
        RUNTIME-CLASS(CChildFrame), // MDI 派生子框架类的 CRuntimeClass 对象的
                                    指针
        RUNTIME-CLASS(CMymdiView2));// 创建文档模板的对象
//然后，使用 CWinApp::AddDocTemplate()方法将新模板添加到应用程序的文档模板列表中
    AddDocTemplate(pDocTemplate2);
    ...
}
```

为使 CMymdiDoc2 类和 CMymdiView2 类在 CMymdiApp 类中成为可识别的，必须在 MymdiApp.cpp 文件中加入 CMymdiDoc2 类和 CMymdiView2 类的说明头文件 MymdiDoc2.h 和 MymdiView2.h：

```
// MymdiApp.cpp : implementation of the CMymdiApp class
#include "MymdiDoc2.h"          //加入头文件
#include "MymdiView2.h"
```

(2) 扩展 CMymdiDoc2 类

① 添加成员变量

首先用在上一节为 CMysdiDoc 类添加成员变量的同样的方法为 CmymdiDoc2 类添加成员变量。最终使得在 CMymdiDoc2.h 中增加下述的程序代码段：

```
// Attributes
public:
CString m_str;
```

② 文档变量初始化

用与上一节同样的方法，在 CmymdiDoc2 的 OnNewDocument 成员变量中为公有成员 m_str赋以初值"您好，多文档界面的例程！"。方法是使用下面的代码替换 OnNewDocument 的实现代码中的// TODO 注释：

```
BOOL CMymdiDoc2::OnNewDocument()
{
    if (! CDocument::OnNewDocument())
        return FALSE;
    m_str = "您好，多文档界面的例程！";
    return TRUE;
}
```

③ 添加菜单处理函数

使用 ClassWizard 在类 CmymdiDoc2 中为菜单项"改变显示文本"（ID 为 ID_EDIT-CHANGETEXT）添加处理函数 OnEditChangeText，其代码如下：

```
void CmymdiDoc2::OnEditChangeText()
{
    // TODO: Add your command handler code here
    CDlgInput inputDlg;//建立一个 CinputDlg 类的对象 inputDlg
    if( inputDlg.DoModal() == IDOK ) //使 inputDlg 成为一个模式对话框
    {
        m_str = inputDlg.m_input; //获取输入的字符串
        UpdateAllViews(NULL); //更新视图
    }
}
```

为使 CInputDlg 类在 CMymdiDoc 类中成为可识别的，必须在 MymdiDoc2.cpp 文件中加入 CInputDlg 类的说明头文件 InputDlg.h：

```
// MymdiDoc2.cpp : implementation of the CmymdiDoc2 class
```

```
#include "InputDlg.h" //加入头文件
```

④ 文档串行化

下面,为示例程序 Mymdi 添加保存及打开文档的实现代码。在此之前,先在"编辑"菜单中添加一条改变文档显示内容的选项,用户可以将对文档显示文本所做的修改保存到一个磁盘文件中,需要的时候再从磁盘文件中打开并恢复该文档。

为了把这些修改保存到磁盘文件中,并在需要时可以打开所保存的磁盘文件读取文档,必须重载 CExampleDoc 类的 Serialize 函数来完成串行化。重载后的 Serialize 函数的代码如下:

```
// CmymdiDoc2 serialization
void CmymdiDoc2::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
      { ar << m_str; } //保存文档内容
    else
      { ar >> m_str; } //读取文档内容
}
```

(3) 视图的输出

下面为 Mymdi 程序的视图类 CmymdiView2 类的 OnDraw 成员函数添加一些代码,以将文档类中的 m_str 成员变量的内容显示到视图的框架窗口中。可用下面的代码来替换类 CmymdiView2 的 OnDraw 成员函数:

```
void CmymdiView2::OnDraw(CDC * pDC)
{
    CmymdiDoc2 * pDoc = (CmymdiDoc2 * )GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    CRect rectClient; // 获取当前客户区的大小
    GetClientRect(rectClient); // 获取当前客户区的指针
    CSize sizeClient = rectClient.Size();// 获取当前客户区的大小
    CString str = pDoc->m_str;     // 从文件中读取数据
    CSize sizeTextExtent = pDC->GetTextExtent(str);  // 用新选定的字体绘制字符串
    pDC->TextOut((sizeClient.cx-sizeTextExtent.cx)/2,(sizeClient.cy-sizeTex-
                  tExtent.cy)/2,str);
                  // 确定文本输出位置
}
```

至此,所有的代码均已输入完毕,经编译、链接即可运行。

习 题

16-1 单文档类的结构是如何定义的?

16-2 多文档类的结构是如何定义的?

16-3　文档模板类的结构是如何定义的?

16-4　视图类的结构是如何定义的?

16-5　创建一个应用程序,其中的文档类中有一个字符串成员变量和一个数字变量,分别用两种视图进行显示,一种是直接显示,一种是将数字作为字符串输出的颜色。

16-6　创建一个单文档的应用程序,可以拖拉鼠标画多个圆,并把圆的坐标存储起来。

参 考 文 献

1 柳永新编著 .Windows 95 和 Windows NT4.0 特色编程 . 北京:清华大学出版社 .1997

2 柳永新、杨东凯、蒋晓瑜等编著 .Windows C 程序设计入门与提高 . 北京:清华大学出版社 .1999

3 木林森编著 .Visual C++ 6.0 使用与开发 . 北京:清华大学出版社,1998

4 潘爱民等译 .Visual C++ 技术内幕(第四版). 北京:清华大学出版社,1998

5 前导工作室译 .Visual C++ 语言参考手册 . 北京:清华大学出版,1999

6 王世同等编著 .Visual C++ 基础编程 . 北京:清华大学出版社,1999

7 周万千等编著 .Windows 95/98 操作系统编程实例详解 . 北京:电子工业出版社,1998

8 陈建春等编译 .Microsoft Visual C++ 图形系统开发技术基础 . 北京:电子工业出版社,1998

9 孙凤英等译 .MFC 开发 Windows 95/NT4 应用程序 . 北京:清华大学出版社,1998

10 陆润民等编著 .C 语言绘图教程 . 北京:清华大学出版社,1996

11 李红娟等译 .Win32 动画编程技术 . 北京:清华大学出版社,1996

12 王燕编著 . 面向对象的理论与 C++ 实践 . 北京:清华大学出版社,1997

13 吕凤翥编著 .C++ 语言基础教程 . 北京:清华大学出版社,1999

14 张国峰编著 .C++ 语言及其程序设计 . 北京:电子工业出版社,1997

15 张国峰编著 .C++ 程序设计实用教程 . 北京:清华大学出版社,1996

16 钱能编著 .C++ 程序设计教程 . 北京:清华大学出版社,1999

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