

第 I 部分

Windows 和 MFC 基础

- 第 1 章 Hello, MFC
- 第 2 章 在窗口中绘图
- 第 3 章 鼠标和键盘
- 第 4 章 菜单
- 第 5 章 MFC 集合类
- 第 6 章 文件 I/O 和串行化
- 第 7 章 控件
- 第 8 章 对话框和属性表

第 1 章 Hello, MFC

短短数年以前,对首次学习编写 Microsoft Windows 程序的人们仅有有限的一些可供选择的编程工具。那时候,C 是 Windows Software Development Kit (SDK)使用的语言,而其他 Windows 编程环境像 Microsoft Visual Basic 还没有出现。大多数应用程序是用 C 语言编写的,那些无经验的程序员面临着艰巨的任务。他们不仅要学习有关新操作系统的一切,还要熟悉 Windows 提供的成百上千个不同的应用软件编程接口(API)函数。

今天,许多 Windows 程序仍然是用 C 语言编写的。但是多种 Windows 编程环境使具有商业品质的 Windows 程序可以用 C、C++、Pascal、Basic 以及许多其他语言来编写。而且,由于 Windows 的复杂性和 Windows API 所涉及领域的广泛性,这就急需有一种面向对象的编程语言。因此,C++ 就取代了 C,而成为专业 Windows 程序员所选用的语言。许多 Windows 程序员都承认 C++ 是 C 的强劲竞争对手,它带有一个类库用来抽象 API,并将窗口和其他对象的基本操作封装在可重复使用的类中,这样就使得 Windows 编程变得简单了。而且,非常多的 C++ 程序员已经选择了 Microsoft Foundation Class(它更为人们熟知的名字是首字母缩略词 MFC),作为他们的类库。虽然也可以使用其他类库,但是只有 MFC 是由编写操作系统的公司编制的。MFC 在被不断地更新以适应 Windows 操作系统最新的变动,并且它提供了一组内容全面的类,完整地表现了从窗口到 ActiveX 控件的所有类,使编写 Windows 应用程序更加容易。

如果您是从传统的 Windows 编程环境如 C 和 Windows SDK 进入 MFC 编程的,那么您已经熟悉了许多使用 MFC 编写 Windows 程序时需要理解的概念。但是,如果您是从面向字符的环境如 MS-DOS 或 UNIX 进入的,您将会发现 Windows 编程完全不同于您以前所做的工作。本章将首先概述 Windows 编程模型,并很快浏览一下 Windows 应用程序是如何工作的;接下来将介绍 MFC;在完成这些基础步骤之后,您将开发自己的第一个 Windows 应用程序——用 MFC 创建一个可调整尺寸的窗口,其中带有一个消息“Hello, MFC”。

1.1 Windows 编程模型

为传统操作系统编写的程序使用的是过程化的模型,程序从头到尾按顺序执行。每次程序调用从开始到结束所经的路径可能会不同,这是由于程序所接收的输入或运行的条件不同,但路径本身是可预测的。C 程序从第 1 行带有 main 函数的语句开始执行,到 main 函数返回值结束。在首尾行之间,函数 main 会调用其他函数,而这些函数可能会调用更多的函数,但始终是程序而不是操作系统决定何时调用哪个函数。

Windows 程序则不是这样执行的。它们使用如图 1-1 所示的事件驱动编程模型,应用程

序通过处理操作系统发送来的消息来响应事件。事件可能是一次击键、鼠标单击或是要求窗口更新的命令以及其他事情。Windows 程序的进入点是函数 WinMain,但是大多数操作是在称为窗口过程的函数中进行的。窗口过程函数处理发送给窗口的消息。WinMain 函数创建该窗口并进入消息循环,即获取消息或将其调度给窗口过程。消息被检索之前处于消息队列中等待。一个典型的应用程序的绝大部分操作是在响应它收到的消息,除了等待下一个消息到达以外,它几乎什么也不做。

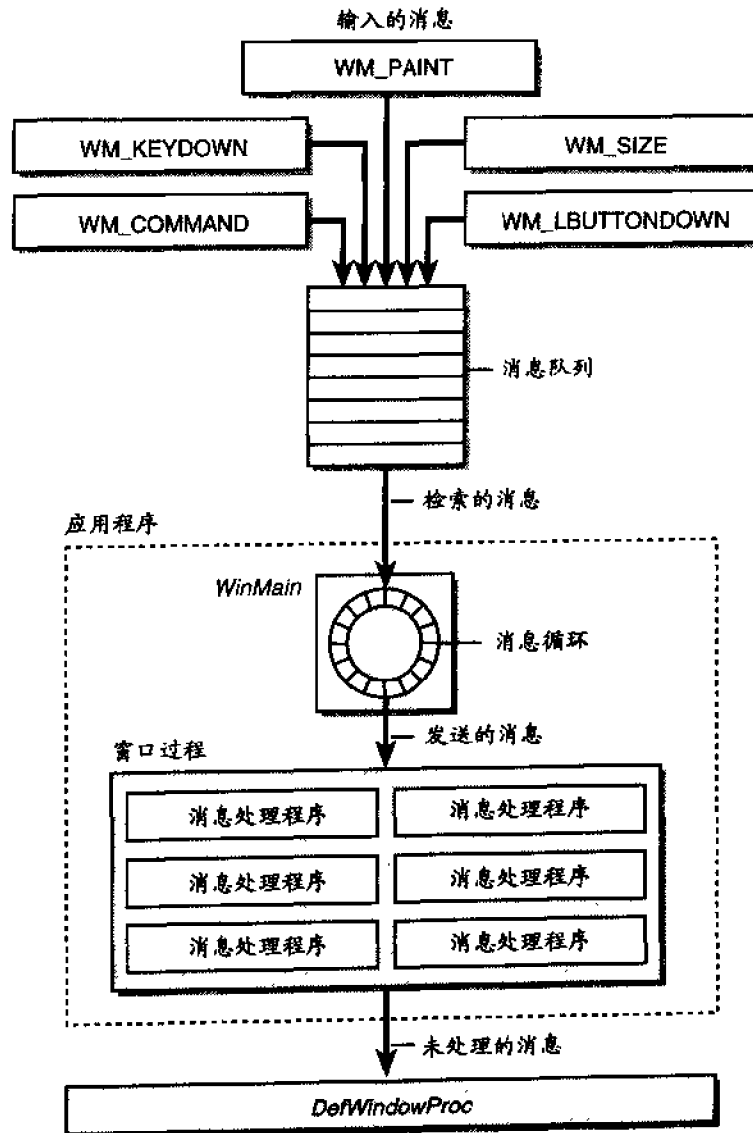


图 1-1 Windows 编程模型

WM_QUIT 消息通知应用程序该结束了,该消息在消息队列中被检索到之后,消息循环将停止。此消息通常在以下几种情况下出现:当用户从“文件”菜单选择了“退出”命令;单击了“关闭”按钮(在窗口的右上角带有×的按钮);或是从窗口系统菜单选择了“关闭”命令。在消息循环停止后,WinMain 函数返回,应用程序结束。

窗口过程一般要调用其他函数来帮助处理接收到的消息。它可以调用应用程序自己的函数,也可以调用 Windows 提供的 API 函数。API 函数包含在动态链接库(即 DLL)这样的专门模块中。Win32 API 包含成千上万个函数,应用程序可以调用它们执行多种任务,例如:创建一个窗口、画条线,以及对文件进行输入/输出处理。在 C 语言中,窗口过程一般作为一个巨大的函数实现,该函数包含一个很大的 switch 语句为每个消息都提供一个选项。处理特定消息的程序代码被称为消息处理程序。应用程序不能处理的消息被传递给了名为 DefWindowProc 的 API 函数,该函数对未被处理的消息提供默认响应。

1.1.1 消息,消息,还是消息

消息是从哪里来的,它们传送什么样的信息? Windows 定义了成百上千个不同的消息类型。大多数消息的开始字符为“WM”并带有下划线,例如 WM_CREATE 和 WM_PAINT。这些消息能按多种不同的方法分类,但是认识到消息在应用程序运行中所起的关键作用要比了解分类更重要。表 1-1 提供了 10 个最常用的消息。例如,当窗口内部需要重新绘制时,它就会得到一个 WM_PAINT 消息。Windows 程序可以被看作为一个消息处理程序的集合。进一步而言,一个程序响应消息的独特方式使它具有了自己的个性。

表 1-1 常用的 Windows 消息

| 消 息 | 发送的条件 |
|----------------|----------------------------|
| WM_CHAR | 从键盘输入字符 |
| WM_COMMAND | 用户选择菜单内的某项,或是控件给其父类发送了一个通知 |
| WM_CREATE | 生成窗口 |
| WM_DESTROY | 撤消窗口 |
| WM_LBUTTONDOWN | 按下鼠标左键 |
| WM_LBUTTONUP | 释放鼠标左键 |
| WM_MOUSEMOVE | 移动鼠标指针 |
| WM_PAINT | 窗口需要重新绘制 |
| WM_QUIT | 应用程序将结束 |
| WM_SIZE | 窗口尺寸被调整 |

消息以调用一个窗口的窗口过程的形式来表明自己的存在。与该调用相伴随的是 4 个输入参数:消息所指窗口的句柄、一个消息 ID 和两个名为 wParam 和 lParam 的 32 位参数。窗口句柄是一个唯一地标识窗口的 32 位值。在内部,该值引用一个数据结构,Windows 在其

中存储着有关窗口的信息,例如窗口的大小、风格及其在屏幕上的位置。消息 ID 是用来标识消息类型的一个数值:WM_CREATE、WM_PAINT 等等。wParam 和 lParam 包含关于特定消息类型的信息。例如,当一个 WM_LBUTTONDOWN 消息到达时,wParam 将保存一系列位标志以标识 Ctrl 和 Shift 键以及鼠标按钮的状态。当鼠标单击发生时,lParam 保存两个 16 位值来标识鼠标指针的位置。这些参数一起向窗口过程提供它所需要的处理 WM_LBUTTONDOWN 消息的所有信息。

1.1.2 Windows 程序设计,SDK 风格

如果您以前没用 C 语言编写过 Windows 程序,那么阅读一个简单程序的源代码应该是有启发意义的。图 1-2 中给出的程序用来生成一个窗口,并在窗口的左上角绘制一个椭圆来响应 WM_PAINT 消息。您将发现这些程序代码与 Charles Petzold 写的《Programming Windows》一书(1998,Microsoft Press)以及其他教授用 C 语言进行 Windows 程序设计的教材中提供的源代码很相似。

```
#include <windows.h>

LONG WINAPI WndProc (HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    WNDCLASS wc;
    HWND hwnd;
    MSG msg;

    wc.style = 0; // Class style
    wc.lpfnWndProc = (WNDPROC) WndProc; // Window procedure address
    wc.cbClsExtra = 0; // Class extra bytes
    wc.cbWndExtra = 0; // Window extra bytes
    wc.hInstance = hInstance; // Instance handle
    wc.hIcon = LoadIcon (NULL, IDI_WINLOGO); // Icon handle
    wc.hCursor = LoadCursor (NULL, IDC_ARROW); // Cursor handle
    wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1); // Background color
    wc.lpszMenuName = NULL; // Menu name
    wc.lpszClassName = "MyWndClass"; // WNDCLASS name

    RegisterClass (&wc);
    hwnd = CreateWindow (
        "MyWndClass", // WNDCLASS name
        "SDK Application", // Window title
        WS_OVERLAPPEDWINDOW, // Window style
        CW_USEDEFAULT, // Horizontal position
        CW_USEDEFAULT, // Vertical position
```

```

        CW_USEDEFAULT,        // Initial width
        CW_USEDEFAULT,        // Initial height
        HWND_DESKTOP,         // Handle of parent window
        NULL,                 // Menu handle
        hInstance,            // Application's instance handle
        NULL                   // Window-creation data
    );

    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;
    HDC hdc;

    switch (message) {
    case WM_PAINT:
        hdc = BeginPaint (hwnd, &ps);
        Ellipse (hdc, 0, 0, 200, 100);
        EndPaint (hwnd, &ps);
        return 0;

    case WM_DESTROY:
        PostQuitMessage (0);
        return 0;

    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

```

图 1-2 一个简单 Windows 程序的 C 语言源代码

WinMain 首先调用 API 函数 RegisterClass 来注册一个窗口类。窗口类定义了窗口的重要特性,如窗口过程地址、默认背景色以及图标等。这些属性通过填写一个 WNDCLASS 结构的字段值来定义,随后将传递给 RegisterClass。当应用程序生成一个窗口时,它必须指定一个窗口类,在该类能被使用之前,必须先对其进行注册。这就是为什么 RegisterClass 在程序

的开始即被调用的原因。要注意 WNDCLASS 类型窗口类与 C++ 中的窗口类不一样。为了避免混淆,我将在本书中用术语 WNDCLASS 来指代用 RegisterClass 注册的类,用术语“窗口类”指代从 MFC 的 CWnd 类派生出的 C++ 类。

一旦 WNDCLASS 被注册,WinMain 将调用最重要的 CreateWindow 函数来生成应用程序的窗口。传递给 CreateWindow 的第 1 个参数是 WNDCLASS 的名字,窗口将由此生成窗口。第 2 个参数是将在窗口的标题栏中显示的文本。第 3 个参数指定窗口样式。WS_OVERLAPPEDWINDOW 是一个常用的样式,它生成一个顶层窗口,该窗口带有可调整大小的边框、一个标题栏、一个系统菜单及最小化、最大化和关闭窗口按钮。

接下来的 4 个参数指定了窗口的初始位置和大小。CW_USEDEFAULT 用来告诉 Windows 使用默认值。最后 4 个参数依次指定:该窗口的父窗口的句柄(HWND_DESKTOP 用作应用程序的主窗口的父窗口);如果有的话,与窗口关联的菜单的句柄;应用程序的实例句柄(一个用来让程序员区分是程序自身还是模块 DLL 的值);以及一个指向特定应用程序的窗口生成数据的指针。我可以轻松地用本书的一节来讨论 CreateWindow 以及它的参数,但是稍后您将看到,MFC 将许多细节隐藏在类库中了。一个典型的 MFC 应用程序并不含有 WinMain 函数(至少不是您能看到的),它也不调用 RegisterClass 或 CreateWindow 函数。

由于生成时没使用 WS_VISIBLE,所以 CreateWindow 生成的窗口在屏幕上最初是不可见的。(如果使用 WS_VISIBLE,则它应该在 CreateWindow 函数的调用中与 WS_OVERLAPPEDWINDOW 结合应用。)因此在 WinMain 中,紧随 CreateWindow 后面的是对 ShowWindow 和 UpdateWindow 函数的调用,它们使窗口可见并确保 WM_PAINT 消息处理程序立刻被调用。

接下来是消息循环。为了检索并调度消息,WinMain 执行一个简单的反复调用 GetMessage、TranslateMessage 和 DispatchMessage 这 3 个 API 函数的 while 循环语句。GetMessage 检查消息队列。如果某个消息是有效的,则它将从队列中被删除并复制到 msg,否则,GetMessage 将停留在消息队列上直到消息有效。msg 是结构 MSG 的一个实例,其字段包含相关的消息参数,例如消息 ID 和消息被放置在队列中的时间。TranslateMessage 函数将一个指示字符键的键盘消息转换为更容易使用的 WM_CHAR 消息,DispatchMessage 函数则将消息发送给窗口过程。消息循环一直执行到 GetMessage 函数返回 0 值时结束,而此情形只有在 WM_QUIT 消息从消息队列中被检索到时才发生。这时 WinMain 结束,程序终止运行。

由 DispatchMessage 函数调度的消息将生成对窗口过程 WndProc 的调用。图 1-2 中的示例程序只是处理了两个消息类型:WM_PAINT 和 WM_DESTROY;所有其他消息被传递给了 DefWindowProc 函数进行默认处理。在 switch-case 块中将检查 message 参数传递来的消息 ID,并且执行相应的消息处理程序。在绘制开始以前,WM_PAINT 处理程序将调用 API 函数 BeginPaint 来获得一个设备环境句柄,当绘制完成后,API 函数 EndPaint 将释放该句柄。在两函数之间,API 函数 Ellipse 绘制了一个 200 像素宽、100 像素高的椭圆。设备环境句柄是一个具有奥妙功能的东西,它允许 Windows 应用程序在屏幕上绘图。没有它,像 Ellipse 这样的函数就不能工作。

WM_DESTROY 处理程序调用 PostQuitMessage API 函数给消息队列发送一个 WM_QUIT 消息,并最终促使程序停止运行。在 WM_DESTROY 消息被发送给窗口之后,紧接着窗口就被撤消了。当接收到一个 WM_DESTROY 消息时,顶层窗口必须调用 PostQuitMessage 函数,否则消息循环不会停止,程序也就永远不会结束。

1.1.3 匈牙利标记法和 Windows 数据类型

在图 1-2 中值得提到的是它使用的变量命名约定。老练的 Windows 程序员知道它是匈牙利标记法,即在每个变量名之前用一个或多个小写字符来标识变量的类型: h 代表句柄, n 代表整数,等等。表 1-2 中列出了一些常用的匈牙利标记法的前缀。前缀经常组合在一起形成新的前缀,例如, p 和 sz 组合形成 psz,代表“指向以零结尾的字符串的指针”。表 1-2 中给出的数据类型许多并不是标准 C/C++ 数据类型,而是相当特殊的在 Windows 头文件中定义的数据类型。例如, COLORREF 是一个 Windows 数据类型,用于存放 24 位 RGB 颜色值; BOOL 是存储 TRUE/FALSE 值的一个布尔数据类型;而 DWORD 是一个 32 位无符号整数。过一段时间,您也将会像了解您自己的编译器的数据类型一样,掌握这些数据类型。

表 1-2 常用匈牙利标记法的前缀

| 前缀 | 数据类型 | 前缀 | 数据类型 |
|--------|----------|----|----------|
| b | BOOL | l | 长整型 |
| c 或 ch | 字符 | n | 整型 |
| clr | COLORREF | p | 指针 |
| cx, cy | 水平或垂直距离 | sz | 以零结尾的字符串 |
| dw | DWORD | w | WORD |
| h | 句柄 | | |

大多数 MFC 程序员也使用匈牙利标记法。看一眼典型的 MFC 程序源代码,您会发现上百个 hs 和 lps 以及其他熟悉的前缀,有的前缀代表 MFC 自己的数据类型(例如, wnd 代表 CWnd 变量)。通常用 m_ 给成员变量加前缀以便明确指出该变量是一个类的成员。在堆栈上创建的临时 CString 变量可能用名字 strWndClass,但是如果它是一成员变量,则可被称为 m_strWndClass。当然,您不必非得接受这些规则,但是了解一下已确定的命名约定会帮助您编写出使其他程序员更容易读懂的代码。

1.1.4 SDK 程序设计展望

如果您以前从未从事过 Windows 程序设计,那么这些内容已经不易消化了,但是它却引出了一些很重要的概念。首先,Windows 是一个事件驱动、基于消息的操作系统。消息是在系统中发生的任何事情的核心,对于一个应用程序,几乎没有哪个操作不是接收消息的直接结果。第二,这里有许多不同的 API 函数和许多不同的消息类型,它们使应用程序开发变得

复杂,并且很难预知应用程序可能会遇到的所有执行方案。第三,看到如何进行 Windows 程序设计会使您对 MFC 和其他类库有一个基本的评价。MFC 并不像某些支持者所说的那样是包治百病的灵丹妙药,但它确实无可否认地使 Windows 程序设计的某些方面变得更容易了。并且,它为 Windows 程序设计提供的优良规则使程序员不必花更多的时间开发一个程序的结构组件,他们也可以不必过多地考虑传递给 CreateWindow 的样式标志及其他关于 API 的无关紧要的内容了。如果您没接触过 MFC,现在是时候了。Windows 程序设计从来没有如此容易过,MFC 的好处就在于 Microsoft 已经为您写好了几万行代码,并对其进行了测试。

1.2 MFC 简介

Microsoft 提供的 MFC 是放置 Windows API 的面向对象的包装的 C++ 类库。MFC 6.0 版本封装了大约 200 个类,其中的一些您可以直接使用,而另一些则主要作为您自己的类的基础类。一些 MFC 类极其简单,例如 CPoint 类,它代表一个点(一个由 x 和 y 坐标定义的位置)。有些类较复杂,例如 CWnd 类,它封装了窗口的功能。在 MFC 程序中,您并不经常直接调用 Windows API;而是从 MFC 类创建对象并调用属于这些对象的成员函数。在类库中定义的成员函数有几百个,其中许多是 Windows API 的简单封装,甚至与相应的 API 函数具有一样的名字。这种命名约定的一个明显好处是它加速了 C 程序员转变为 MFC 程序员的过程。想要移动一个窗口? 一个 C 程序员可能会调用 SetWindowPos API 函数。请在 MFC 手册中查阅一下 SetWindowPos,您将看到 MFC 也支持 SetWindowPos。它是 CWnd 类的一个成员,这可以理解为将窗口作为一个对象,而 SetWindowPos 是希望在该对象上进行的操作。

MFC 也是一个应用程序的框架结构。MFC 不仅仅是一个类集合,它还帮助定义了应用程序的结构并为应用程序处理许多杂务。以 CWinApp 类为例,该类代表应用程序自身,MFC 几乎封装了程序操作的所有方面。框架结构提供 WinMain 函数,而 WinMain 反过来调用应用程序对象的成员函数使程序运行下去。Run 是 WinMain 调用的一个 CWinApp 成员函数,它提供了一个消息循环,将消息送到应用程序的窗口。框架结构还提供了抽象功能,它远远超出了 Windows API 的功能。例如:MFC 的文档/视图体系结构在 API 上建造了一个功能强大的基础结构,它把程序中数据的图形表示(或称为视图)与数据本身分开。这种抽象对 API 而言完全是陌生的,而且在 MFC 框架结构之外或相似类库中也不存在。

1.2.1 使用 C++ 和 MFC 的好处

阅读本书就说明您有可能已经听到过对面向对象设计方法的传统的赞誉了:可重用性,代码和数据更紧密的捆绑,等等。您应该也已经对常用的面向对象程序设计(OOP)术语很熟悉了,比如对象、继承和封装,它们属于 C++ 语言。但是如果没有一个好的类库作为出发点,OOP 可能不会减少您编写代码的数量。

这正是 MFC 成功的地方。要给应用程序添加一个工具栏,使它能够被放置在窗口的各

个边上或浮在它自己的窗口上吗?没有问题:MFC 提供了一个 CToolBar 类,它可以为您做大量的工作。需要一个链接的列表或可调整尺寸的数组?这也容易: CList、CArray 以及其他 MFC 集合类为您的数据提供了封装的容器。另外,别忘了 COM、OLE 以及 ActiveX。在我们之中,很少有人希望或知道如何从头开始编写 ActiveX。在 COleControl 和 COlePropertyPage 等等这样的类中,MFC 提供了您所需的大量代码,从而简化了 ActiveX 控件的开发。

使用 MFC 的另一个优点是框架结构使用了很多技巧,使 Windows 对象,如窗口、对话框以及控件变得如同 C++ 中的对象了。假定您想编写一个可重复使用的列表框类,用来显示一个可浏览的、能显示 PC 主机中的驱动器和目录的列表。除非创建一个自定义控件来进行此工作,否则您无法用 C 来实现这样的列表框,这是因为在列表框中单击某一项会发送一个通知给列表框的父亲(列表框出现的对话框或窗口),父亲将处理那份通知。换句话说,列表框控件控制不了自己的命运;当驱动器或目录被改变时,是父亲来更新列表框中的内容。

用 MFC 则不会这样。在 MFC 应用程序中,窗口和对话框将发送给它们的未处理的通知反射回发送通知的控件。通过从 CListBox 派生出您自己的列表框类,您可以创建一个自含的且高度可重用的列表框类来响应通知。结果列表框实现了自己的行为,并可以通过仅仅在源代码文件添加一条 #include 语句来把它移植到另外一个应用程序中。这就是可重用性的实质。

1.2.2 MFC 的设计思想

在 Microsoft 的程序员开始创建 MFC 时,他们对未来的看法包括以下几个设计目标:

- MFC 应该给 Windows 操作系统提供一个面向对象的接口,支持可重用性、自包含性以及其它 OOP 原则。
- 实现上述目标的前提是不需要强加给系统过多的工作,或不增加应用程序对内存的不必要的开销。

第 1 个目标的实现可通过编写类来封装窗口、对话框以及其他对象,并引入某些关键的虚函数(覆盖这些虚函数可以改变派生类的功能)来完成。第 2 个目标要求 MFC 设计人员尽早就如何将窗口、菜单以及其他对象被 MFC 类(如 CWnd 和 CMenu)包装作出选择。有效使用内存当时是很重要的,今天也很重要,因为没人会喜欢产生臃肿代码的类库。

MFC 设计者所用的使类库带来的总开销减到最小的方法之一在 MFC 对象与 Windows 对象之间的关系中得到了体现。在 Windows 中,有关窗口特性和目前状态的信息被保存在操作系统拥有的内存中。这些信息对应用程序是隐藏的,应用程序只能处理窗口句柄或 HWND。MFC 并没有复制在 CWnd 类的数据成员中的与 HWND 有关的所有信息;事实上,MFC 通过将 HWND 存储在称为 m_hWnd 的公用 CWnd 数据成员中,而在 CWnd 内包装了一个窗口。作为规则,如果 Windows 通过某种类型的句柄展示一个对象,那么相应的 MFC 类

就会包含那个句柄的数据成员。如果您想要调用 API 函数,该函数要求一个句柄,但是您只有 CWnd 或 CWnd 指针,而不是 HWND,那么这些知识对您是有用的。

1.2.3 文档/视图体系结构

MFC 应用程序框架结构的基石是文档/视图体系结构,它定义了一种程序结构,这种结构依靠文档对象保存应用程序的数据,并依靠视图对象控制视图中显示的数据。MFC 在类 CDocument 和 CView 中为文档和视图提供了基础结构。CWinApp、CFrameWnd 和其他类与 CDocument 和 CView 合作,把所有的片段连在了一起。现在详细讨论文档/视图体系结构还为时过早,但是您至少应该对术语“文档/视图”比较熟悉,因为它必然会出现在有关 MFC 的任何讨论中。

文档和视图如此重要的原因在于,文档/视图应用程序从应用程序框架结构中得到了最大的好处。您可以不使用文档和视图来编写 MFC 程序(在本书的很多篇幅中我们这样做了,特别是从第 1 章到第 8 章),但是要想从框架结构中获得最大的好处并利用某些 MFC 的最高级特性,您就必须使用文档/视图体系结构。这其实并不像听上去那样限制严格,因为几乎任何依赖某类文档的程序都能用文档/视图生成。别让术语“文档”误导您,以为文档/视图体系结构对编写字处理器和电子表格程序有用。文档仅仅是程序数据的抽象表示。文档既可能是保存计算机象棋游戏中棋盘位置的字节数组,也可能是一个电子表格。

MFC 对文档/视图应用程序提供了什么样的支持呢?首先,文档/视图体系结构极大地简化了打印和打印预览、向磁盘中存储文档以及读取文档的过程,将应用程序变换为 Active 文档服务器(其文档可以在 Microsoft Internet Explorer 中打开)。在本书第二部分您将学到文档/视图体系结构,但是只有不用文档和视图完成一些程序设计以后,您才有可能逐步了解 MFC,而不会一下学得太多消化不了。

1.2.4 MFC 类的分层结构

MFC 提供了许多设计好的类来满足广泛的需要。您可以在本书封二找到一个易于使用的 MFC 6.0 类分层结构图。大多数 MFC 类都是从 CObject 中直接或间接地派生出来的。CObject 给那些继承它的类提供了 3 个重要的特性:

- 串行化支持
- 运行时类信息支持
- 诊断和调试支持

串行化是对象的永久数据流出或流入存储介质(如磁盘文件)的进程。把 CObject 作为基类,可以创建可串行化的类,其实例容易存储和重新创建。运行时类信息(RTCI)允许您在运行时检索对象的类名称以及对象的其他信息。RTCI 的执行不同于 C++ 中的运行时类型信息(RTTI)机制,因为它比 RTTI 要早出现好多年。嵌入在 CObject 中的诊断和调试支持允

许您对 CObject 派生类的实例执行有效性检查,并将状态信息转储到一个调试窗口。

CObject 对它的派生类还提供了别的好处。例如,重载 new 和 delete 运算符防止内存泄漏。如果从 CObject 派生类创建了一个对象,而没有在应用程序结束前删除它,那么 MFC 将会在调试输出窗口写一条警告消息。随着您对 MFC 越来越熟悉,这个 MFC 类基本特点的重要性也会变得越来越清晰。

1.2.5 AFX 函数

并非所有 MFC 提供的函数都是类成员。MFC 以全局函数的形式提供了自己各类的 API 函数,名字以 Afx 打头。类成员函数仅仅能在所属的对象说明体中被调用,而在任何时候任何地方都可使用 AFX 函数。

表 1-3 中列出了一些常用的 AFX 函数。AfxBeginThread 简化生成执行线程的进程。AfxMessageBox 和 Windows MessageBox 函数是全局等价的,但不同于 CWnd::MessageBox,它既可以从窗口类也可以从文档类被调用。AfxGetApp 和 AfxGetMainWnd 返回指向应用程序对象和应用程序主窗口的指针,当您想要访问这些对象的某个函数或数据成员,但又没有合适的指针时,这两个函数是有用的。AfxGetInstanceHandle 便于您将实例句柄传递给 Windows API 函数。(即使 MFC 程序也经常调用 API 函数!)

表 1-3 常用的 AFX 函数

| 函数名称 | 说 明 |
|----------------------|-----------------------------|
| AfxAbort | 无条件地终止一应用程序;通常在不可恢复错误发生时调用 |
| AfxBeginThread | 创建新的线程并开始执行它 |
| AfxEndThread | 终止当前执行的线程 |
| AfxMessageBox | 显示 Windows 消息框 |
| AfxGetApp | 返回指向应用程序对象的指针 |
| AfxGetAppName | 返回应用程序的名称 |
| AfxGetMainWnd | 返回指向应用程序主窗口的指针 |
| AfxGetInstanceHandle | 返回标识当前应用程序实例的句柄 |
| AfxRegisterWndClass | 为 MFC 应用程序注册自定义的 WNDCLASS 类 |

1.3 您的第一个 MFC 应用程序

有什么比编写一个在窗口中显示“Hello, MFC”的程序作为起点更好的呢?以 Brian Kernighan 和 Dennis Ritchie 编写的《C 程序设计语言》(1988, Prentice-Hall 出版社)中那个经典不朽的“hello, world”例子程序为基础,我将这个很小的程序称为 Hello,将说明使用 MFC 编写 Windows 应用程序所涉及到的基本原则。首先,您要仔细看一下 MFC 的 CWinApp 和 CFrameWnd 类,并注意其他类是如何从它们派生出来并插进应用程序的。您也将学习很重

要的 CPaintDC 类,它作为纽带,在窗口里画出文本与图形来响应 WM_PAINT 消息。最后,您将了解消息映射机制,MFC 用它来联系应用程序接收到的消息和您提供的处理这些消息的成员函数。

图 1-3 列出了 Hello 程序的源代码。Hello.h 包含两个派生类的声明。Hello.cpp 包含这些类的应用。C++ 程序员习惯上把类定义放在 .h 文件中,把源程序代码放在 .cpp 文件中。我们将在整本书中遵从这一习惯。对于包含成百上千个类的大型应用程序,把类声明和实现放在分开的源代码文件中也是有益的。对本书前几章中的程序不是这样,但晚些时候,当我们开始用文档和视图工作时,我们将为每个类提供它自己的 .h 和 .cpp 文件。在书后附带的光盘上,在文件夹 Chap01 中,您将看到包含 Hello.h 和 Hello.cpp 的文件夹以及包含可执行文件(Hello.exe)的文件夹。

Hello.h

```
class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CFrameWnd
{
public:
    CMainWindow();

protected:
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()
};
```

Hello.cpp

```
#include <afxwin.h>
#include "Hello.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
}
```

```
        return TRUE;
    }

    //////////////////////////////////////
    // CMainWindow message map and member functions

    BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
        ON_WM_PAINT ()
    END_MESSAGE_MAP ()

    CMainWindow::CMainWindow ()
    {
        Create (NULL, _T ("The Hello Application"));
    }

    void CMainWindow::OnPaint ()
    {
        CPaintDC dc (this);

        CRect rect;
        GetClientRect (&rect);

        dc.DrawText (_T ("Hello, MFC"), -1, &rect,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
    }
}
```

图 1-3 Hello 应用程序

图 1-4 显示了 Hello 的输出结果。在运行应用程序时,可以注意到窗口的功能全部有效了:可以移动它,调整其尺寸大小,使它变到最小、最大,也可关闭它;并且在窗口缩放时,“Hello, MFC”将在窗口中心重新绘出。

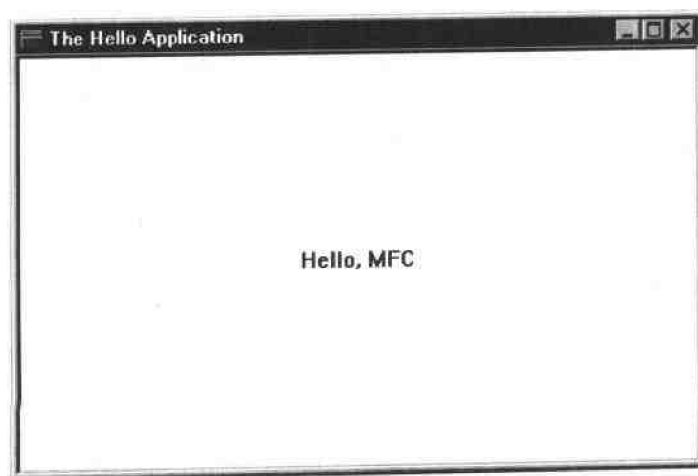


图 1-4 Hello 程序的窗口

大多数 Hello 的功能来自 Windows。例如,Windows 绘制窗口的外貌,或称为非用户区域:标题栏、标题栏上的按钮以及窗口的边框。您的责任就是创建窗口并处理 WM_PAINT 消息,该消息指出窗口内部(客户区)是全部还是部分需要更新。请阅读程序源代码,看一下 Hello 是如何工作的。

1.3.1 应用程序对象

MFC 应用程序的核心就是基于 CWinApp 类的应用程序对象。CWinApp 提供了消息循环来检索消息并将消息调度给应用程序的窗口。它还包括可被覆盖的、用来自定义应用程序行为的主要虚函数。一旦包含头文件 Afxwin.h,就可以将 CWinApp 以及其他 MFC 类引入应用程序中。一个 MFC 应用程序可以有且仅有一个应用程序对象,此对象必须声明为在全局范围内有效,以便它在程序开始时即在内存中被实例化。

Hello 的应用程序类被命名为 CMyApp。它在 Hello.cpp 中用如下语句进行了实例化:

```
CMyApp myApp;
```

CMyApp 的类声明在 Hello.h 中显示:

```
class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
```

CMyApp 没有声明任何数据成员,只是覆盖了一个从 CWinApp 类中继承来的函数。在应用程序的生存期内 InitInstance 的调用比较早,是在应用程序开始运行以后而窗口创建之前。事实上,除非 InitInstance 创建一个窗口,否则应用程序是不会有窗口的。这正是为什么即使最小的 MFC 应用程序也必须从 CWinApp 派生出一个类并覆盖 CWinApp::InitInstance 的原因。

InitInstance 函数

CWinApp::InitInstance 是一个虚函数,其默认操作仅包含一条语句:

```
return TRUE;
```

InitInstance 的目的是为应用程序提供一个自身初始化的机会。由 InitInstance 返回的值决定了框架结构接下来要执行的内容。从 InitInstance 返回 FALSE 将关闭应用程序。如果初始化正常,InitInstance 将返回 TRUE 以便允许程序继续进行。InitInstance 是用来执行程序每次开始时都需要进行的初始化工作的最好地方。至少,这意味着创建在屏幕上表现应用程序的窗口。

在 Hello.cpp 中, CMyApp 的 InitInstance 通过实例化 Hello 的 CMainWindow 类来创建 Hello 窗口。语句

```
m_pMainWnd = new CMainWnd;
```

构造了一个 CMainWindow 对象, 并将其地址复制到了应用程序对象的 m_pMainWnd 数据成员中。在窗口创建以后, InitInstance 就会通过 CMainWindow 指针调用 ShowWindow 和 UpdateWindow 函数来显示它, 要记住除非使用 WS_VISIBLE 属性, 否则窗口是不可见的:

```
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
```

ShowWindow 和 UpdateWindow 是所有窗口对象共用的 CWnd 成员函数, 其中包括 CFrameWnd 类的对象, CMainWindow 就是从 CFrameWnd 派生出来的。这些函数几乎就是对同样名称的 API 函数的包装。要从 MFC 程序调用一个常规的 Windows API 函数, 需要在函数名称前添加一个全局运算符 "::", 例如:

```
::UpdateWindow(hwnd);
```

这个记号确保即使对象具有相同名称的成员函数, 也可以调用 API 函数。在本书其余部分, Windows API 函数将用 "::" 标记以便与 MFC 成员函数区分开来。

ShowWindow 仅接收一个参数, 即一个整数, 用来指定窗口开始显示时处于最小化、最大化或者既不是最小也不是最大。根据 Windows 程序设计协议, Hello 把存储在应用程序对象的 m_nCmdShow 变量中的值传递给了 ShowWindow, 其中保存着传递给 WinMain 的 nCmdShow 参数。m_nCmdShow 的值通常是 SW_SHOWNORMAL, 指出窗口应该处于正常的非最小、非最大状态。然而, 根据用户启动应用程序的不同, Windows 偶尔也将如 SW_SHOWMAXIMIZED 或 SW_SHOWMINIMIZED 这样的值插进去。除非有特殊情况, 否则 InitInstance 应该总是给 ShowWindow 传递 m_nCmdShow 变量而不是硬编码的 SW_ 值。

UpdateWindow 立即重新绘制窗口来完成由 ShowWindow 启动的作业。工作完成后, InitInstance 返回 TRUE 以允许应用程序继续进行。

其他 CWinApp 可覆盖函数

InitInstance 仅仅是几个能够被覆盖的虚拟 CWinApp 成员函数之一, 可通过它来自定义应用程序对象的操作。在您的 MFC 文档中查找一下 CWinApp 可覆盖函数, 您将看到一个列表, 其中包含名字如 WinHelp 和 ProcessWndProcException 这样的函数。尽管很方便, 但这些函数中的大多数极少被覆盖。例如: 您可以使用 ExitInstance 在应用程序终止后清屏。如果使用 InitInstance 分配了内存或其他资源, ExitInstance 将是释放这些资源的完美地方。ExitInstance 默认的操作是做一些框架结构要求的清除事务, 因此如果您覆盖了 ExitInstance 就应该确保要调用基类版本。最终, 由 ExitInstance 返回的值是由 WinMain 返回的退出代码。

其他有趣的 CWinApp 可覆盖函数包括 OnIdle、Run 以及 PreTranslateMessage。用 OnIdle 可以方便地执行如垃圾回收这样的后台处理事务。因为是在应用程序“空闲”时才调用 OnIdle,也就是在没有消息等候处理时才调用它,这样就为处理低优先级的后台任务提供了一种优秀的机制,它不会大量产生个别的执行线程。在第 14 章中将详细讨论 OnIdle。您可以覆盖 Run 来自定义消息循环,用自己的消息循环代替它。如果想要在消息被调度以前执行一些专门的预处理,则可以覆盖 PreTranslateMessage,而不必自己编写整个新的消息循环。

1.3.2 MFC 如何使用应用程序对象

对于从未见过 MFC 应用程序源代码的人而言,Hello 程序显著的特点就是它没有包含自己定义的类以外的任何可执行代码。例如,它没有 main 或 WinMain 函数;在整个程序中唯一的语句就是具有全局有效性的用来实例化应用程序对象的语句。那么到底是什么启动了程序的运行,应用程序对象又是何时起作用的呢?

揭开这个谜的最好方法就是看看主框架的源代码。一个 MFC 提供的源代码文件(Winmain.cpp)中包含一个 AfxWinMain 函数,它在 MFC 中的作用相当于 WinMain。(实际上,当您购买了 Visual C++,同时也就得到了 MFC 提供的源代码。)AfxWinMain 广泛使用应用程序对象,这就是为什么应用程序对象必须作全局声明的原因。全局变量和对象在任何其他代码执行以前被创建,在 AfxWinMain 运行以前,应用程序对象必须在内存中存在。

运行一开始,AfxWinMain 就调用 AfxWinInit 函数来初始化主框架,并将 hInstance、nCmdShow 以及其他 AfxWinMain 函数参数复制给应用程序对象的数据成员,然后它调用 InitApplication 和 InitInstance。在 MFC 的 16 位版本中,只有传递给 AfxWinMain 的 hPrevInstance 参数是空时,才调用 InitApplication,这表明当前运行的是应用程序的唯一实例。在 Win32 环境下,hPrevInstance 总是空的,因此主框架不必再去检查。32 位应用程序能够像使用 InitInstance 一样方便地使用 InitApplication 初始化自身,但是 InitApplication 是为保证 MFC 先前版本的兼容性而提供的,因此不应该在 32 位 Windows 应用程序中使用。如果 AfxWinInit、InitApplication 或者 InitInstance 返回 0 值,则 AfxWinMain 终止而不是继续进行,应用程序被关闭。

只有在上述所有函数返回非零的值时,AfxWinMain 才执行以下关键的步骤。语句

```
pThread->Run();
```

调用应用程序对象的 Run 函数,该函数执行消息循环并开始向应用程序窗口发送消息。消息循环重复执行,直到 WM_QUIT 消息从消息队列中被检索到。这时 Run 跳出循环,并调用 ExitInstance,返回到 AfxWinMain 中。在执行了最后的一些清除工作之后,AfxWinMain 执行一个 return 语句结束应用程序。

1.3.3 框架窗口对象

MFC 的 CWnd 类及其派生类为窗口或应用程序创建的窗口提供了面向对象的接口。

Hello 的窗口类 CMainWindow 是从 MFC 的 CFrameWnd 类派生的,而后者又是从 CWnd 派生的。CFrameWnd 模仿框架窗口的行为。现在,可以把框架窗口作为顶层窗口看待,它是应用程序与外部世界的主要接口。在文档/视图体系结构大环境中,框架窗口作为视图、工具栏、状态栏以及其他用户界面(UI)对象的智能化容器起了更大的作用。

通过生成窗口对象并调用其 Create 或 CreateEx 函数,MFC 应用程序可以创建一个窗口。在 CMyApp::InitInstance 中,Hello 创建了一个 CMainWindow 对象。CMainWindow 的构造函数生成在屏幕上看到的窗口:

```
Create(NULL,_T("The Hello Application"));
```

_T 是一个宏,用来把字符串常量字符设置为中性。在本章稍后将讨论此内容。Create 是从 CFrameWnd 继承来的一个 CMainWindow 成员函数。在 CFrameWnd 中,由它自己定义的和从 CWnd 继承来的函数大约有 20 个,Create 是其中之一。CFrameWnd::Create 的原型如下:

```
BOOL Create(LPCTSTR lpszClassName,
            LPCTSTR lpszWindowName,
            DWORD dwStyle = WS_OVERLAPPEDWINDOW,
            const RECT& rect = rectDefault,
            CWnd* pParentWnd = NULL,
            LPCTSTR lpszMenuName = NULL,
            DWORD dwExStyle = 0,
            CCreateContext* pContext = NULL)
```

Create 接收的 8 个参数中的 6 个由默认值定义。Hello 只需执行最少量的工作,为函数的前 2 个参数指定值,对剩下 6 个参数接受默认值。第 1 个参数 lpszClassName 指定了窗口基于 WNDCLASS 类的名称。为此,将其设定为 NULL 将创建一个基于由主结构注册的 WNDCLASS 类的默认框架窗口。lpszWindowName 参数指定将在窗口的标题栏出现的文本。

dwStyle 参数指定窗口样式。默认值为 WS_OVERLAPPEDWINDOW。调用 Create 函数可以通过指定别的样式或选择组合样式来更改窗口样式。在 CFrameWnd::Create 文档中可以找到一个完整的窗口样式列表。在框架窗口中时常使用的两个样式是 WS_HSCROLL 和 WS_VSCROLL,它们在窗口客户区的底边和右边添加水平、垂直滚动条。语句

```
Create(NULL,_T("Hello"), WS_OVERLAPPEDWINDOW|WS_VSCROLL);
```

创建一个包含垂直滚动条的重叠型窗口。正如这个例子说明的那样,可以使用 C++ 的“|”运算符来组合多个样式。WS_OVERLAPPEDWINDOW 组合了 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_MINIMIZEBOX、WS_MAXIMIZEBOX 以及 WS_THICKFRAME 样式。因此,如果您想创建一个看上去与 WS_OVERLAPPEDWINDOW 窗口相似但在标题栏缺少最小化按钮的窗口,您可以这样调用 Create 函数:

```
Create(NULL,_T("Hello"), WS_OVERLAPPED|WS_CAPTION|
        WS_SYSMENU|WS_MINIMIZEBOX|WS_THICKFRAME);
```

指定窗口样式的另一个可选方法是覆盖窗口从 CWnd 继承来的 PreCreateWindow 虚函数,修改传递给 PreCreateWindow 的 CREATESTRUCT 结构的 style 字段。在主结构为您创建了应用程序主窗口之后,很容易获得此功能。正如在文档/视图应用程序中那样,当您的程序代码直接调用 Create 并且能控制传递给它的参数时,这样做就没必要了。在本书稍后部分,您将看到说明在何时以及如何使用 PreCreateWindow 的例子。

可以在 CFrameWnd::Create 的 dwExStyle 参数中指定附加的叫做“扩展样式”的窗口样式。出于历史原因,窗口样式被分为标准和扩展样式,通过调用 ::CreateWindowEx API 函数,Windows 3.1 补充了对附加窗口样式的支持。::CreateWindowEx 与 ::CreateWindow 很相似,只不过它的参数列表中包含了一个附加参数,用来指定窗口的扩展样式。Windows 3.1 只支持 5 种扩展样式,而 Windows 最近的版本则提供了更广泛的选项,其中包括 WS_EX_WINDOWEDGE 和 WS_EX_CLIENTEDGE 样式,它们使窗口的边框具有更明显的 3D 效果。由于 MFC 已经自动为您给框架窗口添加了这两个样式,因此您可以不必亲自指定它们。

在 dwStyle 参数之后是 rect,它是 C++ 中对 CRect 对象具有 C 风格的 RECT 结构的引用,它指定了窗口的在屏幕上的初始位置和尺寸。dwStyle 的默认值为 rectDefault,它是 CFrameWnd 类的静态成员,用于通知 Windows 选择窗口的默认初始位置和尺寸。如果愿意,您可以指定初始位置和尺寸,只要用屏幕上描述矩形的坐标值来初始化 CRect 对象,并将其传递给 Create 函数即可。下列语句创建一个标准的可重叠窗口,它的左上角位于屏幕左上角向右 32 像素、向下 64 像素的地方,初始宽度和高度分别为 320 像素和 240 像素:

```
Create(NULL,_T("Hello"),WS_OVERLAPPEDWINDOW,
    CRect(32,64,352,304));
```

注意,窗口的宽度和高度是分别由第 1 个参数与第 3 个参数的差,以及第 2 个参数与第 4 个参数的差确定的,而不是第 3 与第 4 个参数的绝对值。也就是说,CRect 对象指定了窗口在屏幕上所占据的矩形区域。传递给 CRect 构造函数的 4 个参数按顺序指定了矩形的左、上、右、下屏幕坐标。

Create 中的 pParentWnd 参数指定了窗口的父亲或所有者。目前不要考虑父亲和所有者的问题。由于顶层窗口没有父亲或所有者,所以它的这个参数总是为 NULL。(实际上,将 pParentWnd 设定为 NULL 就是使桌面窗口,即作为屏幕背景的窗口,成为此窗口的所有者。而这些细节的工作都是 Windows 的事。)

Create 的 lpszMenuName 参数指定与窗口有关的菜单。NULL 表明窗口无菜单。在第 4 章我们将开始学习使用菜单。

CFrameWnd::Create 的最后一个参数是 pContext,包含一个指向 CCreateContext 结构的指针,它在文档/视图应用程序中主结构初始化框架窗口时要用到。在文档/视图体系结构以外,此参数毫无意义,应该设置为 NULL。

Create 为程序员提供了极其繁多的选项。在初学阶段,大量的可选内容可能使您无所适从,如果您以前没有开发过 Windows 程序,尤其会这样。但是随着经验积累,您就会懂得什么时候、如何利用这些选项了。而且,如果您所需要的只是一个标准的 CFrameWnd 类型的窗口,那么类库就可以使用默认函数参数为您隐藏许多复杂的内容。这也正是 MFC 使 Windows 程序设计变得稍微容易了一些的一个例子。

1.3.4 绘制窗口

Hello 程序不能随心所欲地在屏幕上进行绘制。相反,它是通过响应来自 Windows 的 WM_PAINT 消息进行绘制的,此消息通知它该更新窗口了。

WM_PAINT 消息的发生可以有多种原因:由于移动了另一个窗口;由于 Hello 的窗口原先被遮盖的一部分显露出来了;或者由于窗口的大小改变了。不论诱因是什么,都要由应用程序来负责通过响应 WM_PAINT 消息绘制其窗口的客户区。由 Windows 来绘制非客户区,这样所有的应用程序将具有一致的外观。但是,如果应用程序不为客户区执行其自身的绘制例程,那么窗口的内部将是一片空白。

在 Hello 程序中,WM_PAINT 消息由 CMainWindow::OnPaint 来处理,当一个 WM_PAINT 消息抵达时都将调用它。OnPaint 的工作就是在窗口客户区的中央位置绘制“Hello, MFC”。它是通过构造一个名为 dc 的 CPaintDC 对象开始的:

```
CPaintDC dc (this);
```

MFC 的 CPaintDC 类是从 MFC 的更为一般的 CDC 类派生的,CDC 类封装了 Windows 设备环境,以及包含了绘制到屏幕、打印机和其他设备的几十个成员函数。在 Windows 中,所有的图形输出都通过设备环境对象执行,设备环境对象抽象了输出的物理目的地。CPaintDC 只在 WM_PAINT 消息处理程序中使用,它是 CDC 的一个特殊例子。应用程序在响应 WM_PAINT 消息进行绘制之前,必须调用 Windows 的::BeginPaint API 函数来获取一个设备环境,以准备将此设备环境用于绘制。当应用程序完成了绘制时,它必须调用::EndPaint 来释放设备环境和通知 Windows 绘制已经完成。如果应用程序在处理 WM_PAINT 消息时不能调用::BeginPaint 和::EndPaint,那么该消息将不会从消息队列中删除。不足为怪的是,CPaintDC 将从其构造函数调用::BeginPaint 和从其析构函数调用::EndPaint 来确保不出现这样的情况。

在 MFC 中,您一般都使用某种类型的 CDC 对象在屏幕上进行绘制,但您必须仅在 OnPaint 程序内使用 CPaintDC 对象。此外,在栈上创建 CPaintDC 对象是良好的编程习惯,这样当 OnPaint 结束时将自动调用它们的析构函数。如果需要,您可以使用 new 操作符来实例化一个 CPaintDC 对象,但是,在 OnPaint 结束之前删除那个对象将变得至关重要。否则,::EndPaint 将不会被调用,而且您的应用程序也将不能正确地重新绘制。

在创建了 CPaintDC 对象之后,OnPaint 将构造一个代表矩形的 CRect 对象,并调用 CWnd

`::GetClientRect` 以使用窗口的客户区的坐标来初始化这个矩形:

```
CRect rect;
GetClientRect(&rect);
```

`OnPaint` 然后调用 `CDC::DrawText` 在窗口的客户区中显示“Hello, MFC”:

```
dc.DrawText(_T("Hello, MFC"), -1, &rect,
    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

`DrawText` 是一个用于输出文本的功能强大、用途广泛的函数。它接收 4 个参数: 一个指向待显示的字符串的指针、字符串中的字符数(或者 -1, 如果字符串是以 NULL 字符终止的)、指定格式矩形的一个 `RECT` 结构或者 `CRect` 对象的地址, 以及指定输出选项的标记。在 `Hello` 中, `CMainWindow::OnPaint` 组合了 `DT_SINGLELINE`、`DT_CENTER` 和 `DT_VCENTER` 标记, 以显示一个在格式矩形中水平居中和垂直居中的单行文本。`rect` 描述了窗口的客户区, 因此输出结果在窗口中完全居中。

`DrawText` 的参数列表中明显缺少的是指定输出的基本属性(比如字体和文本颜色)的参数。这些和其他的输出特征都是设备环境的属性, 可以使用 `CDC` 成员函数, 比如 `SelectObject` 和 `SetTextColor` 等来控制。由于 `Hello` 不更改设备环境的任何属性, 因此将使用默认的字体和默认的文本颜色(黑色)。`DrawText` 还使用设备环境的当前背景颜色在它所输出的文本的周围填充一个小矩形。这个默认的颜色是白色, 因此, 如果系统的默认窗口背景颜色碰巧也是白色, 那么您将看不到该矩形。但是, 如果将窗口背景颜色更改为灰色, 那么白色的文本背景将像疼痛的拇指那样醒目。

在第 2 章中, 您将学会如何通过修改设备环境属性来自定义 `DrawText` 和其他 `CDC` 绘图函数的输出。一旦您知道了怎样做之后, 更改文本颜色或者告诉 `DrawText` 使用“透明”的像素来绘制文本背景将非常简单。

1.3.5 消息映射

来自 Windows 的 `WM_PAINT` 消息是如何转换为一个对 `CMainWindow::OnPaint` 的调用的呢? 答案就在于消息映射。消息映射是一个将消息和成员函数相互关联的表。当 `Hello` 的框架窗口收到了一个消息, MFC 将搜索该窗口的消息映射, 如果存在一个处理 `WM_PAINT` 消息的处理程序, 然后就调用 `OnPaint`。消息映射是 MFC 避免使用冗长的虚表的一种方式, 如果每个类对它可能会接收到的每个可能消息都有一个虚拟函数, 那么就需要虚表。从 `CCmdTarget` 派生的任何类都可以包含消息映射。MFC 为执行消息映射在内部所做的工作隐藏在某些十分复杂的宏当中, 但“使用”消息映射是相当简单的。下面就是您将消息映射添加到一个类中需要做的全部工作:

1. 通过将 `DECLARE_MESSAGE_MAP` 语句添加到类声明中, 声明消息映射。
2. 通过放置标识消息的宏来执行消息映射, 相应的类将在对 `BEGIN_MESSAGE_MAP`

和 `END_MESSAGE_MAP` 的调用之间处理消息。

3. 添加成员函数来处理消息。

Hello 的 `CMainWindow` 类只处理一种消息类型——`WM_PAINT`, 因此其消息映射的实现如下所示:

```
BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_PAINT ()
END_MESSAGE_MAP ()
```

`BEGIN_MESSAGE_MAP` 开始了消息映射, 并标识了消息映射所属的类和该类的基类(消息映射就像其他的类成员那样, 可以通过继承来传递。其中需要有基类名, 这样框架就可以在必要时查找基类的消息映射)。 `END_MESSAGE_MAP` 结束了消息映射。在 `BEGIN_MESSAGE_MAP` 和 `END_MESSAGE_MAP` 之间是消息映射条目。 `ON_WM_PAINT` 是一个在 MFC 头文件 `Afxmsg.h` 中定义的宏。它将处理 `WM_PAINT` 消息的条目添加到消息映射中。该宏不接收参数, 因为将 `WM_PAINT` 消息链接到名为 `OnPaint` 的类成员函数是不需要代码的。MFC 为 100 多种从 `WM_ACTIVATE` 到 `WM_WININICHANGE` 的 Windows 消息提供了宏。您可以从 MFC 文档中获取与特定的 `ON_WM` 宏相对应的消息处理程序的名称, 然而要自己推断名称也是相当容易的, 您可以将 `WM_` 替换为 `On`, 将除那些单词开头字母之外的所有其他字母变为小写字母。因此, 如 `WM_PAINT` 变为 `OnPaint`, `WM_LBUTTONDOWN` 变为 `OnLButtonDown` 等等。

您将需要参考 MFC 文档, 以决定一个消息处理程序接收何种类型的参数以及返回何种类型的值。 `OnPaint` 不接收参数, 也没有返回值, 而 `OnLButtonDown` 的函数原型是这样的:

```
afx_msg void OnLButtonDown (UINT nFlags, CPoint point)
```

`nFlags` 包含指定鼠标和 Ctrl 及 Shift 键的状态的位标记, `point` 标识了鼠标单击的位置。传递给消息处理程序的参数来自于伴随消息的 `wParam` 和 `lParam` 参数。但是, 鉴于 `wParam` 和 `lParam` 是普遍需要的, 因而传递给 MFC 消息处理程序的参数既是特定的也是类型安全的。

如果您希望处理一个 MFC 没有为之提供消息映射宏的消息时会怎么样呢? 您可以使用普通的 `ON_MESSAGE` 宏为消息创建一个条目, 该宏接受两个参数: 消息 ID 和对应类成员函数的地址。下面的语句就将 `WM_SETTEXT` 映射到了一个名为 `OnSetText` 的成员函数:

```
ON_MESSAGE (WM_SETTEXT, OnSetText)
```

`OnSetText` 应以如下的方式进行声明:

```
afx_msg LRESULT OnSetText (WPARAM wParam, LPARAM lParam);
```

由 MFC 提供的其他特殊用途的消息映射宏包括: `ON_COMMAND`, 它将菜单选择和其他的 UI 事件映射到类成员函数; `ON_UPDATE_COMMAND_UI`, 它将菜单项和其他 UI 对象连接到

保持它们与应用程序的内部状态同步的“更新处理程序”。下面的几个章节将向您介绍这些消息映射宏和其他的消息映射宏。

我们再回到 Hello 程序, CMainWindow 的 OnPaint 函数和消息映射是在 Hello.h 中使用下面的语句声明的:

```
afx_msg void OnPaint ();
DECLARE_MESSAGE_MAP ()
```

afx_msg 醒目地暗示, OnPaint 是一个消息处理程序。如果您愿意, 您可以将它省略, 因为在编译时它将简化为空白。术语 afx_msg 是用来表示一个其行为很像虚拟函数但却不需要虚表项的函数。在类声明中, DECLARE_MESSAGE_MAP 通常是最后的语句, 因为它使用 C++ 关键字来指定其成员的可见度。您可以在 DECLARE_MESSAGE_MAP 后面插入声明其他类成员的语句, 但是如果这样做了, 您还应当使用关键字 public、protected 或者 private 开头, 以确保您希望为这些成员设置的可见度。

1.3.6 消息映射的工作方式

您可以通过检查 Afxwin.h 中的 DECLARE_MESSAGE_MAP、BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 宏以及 Wincore.cpp 中的 CWnd::WindowProc 代码, 来找到消息映射是如何工作的。下面就是当您在代码中使用消息映射时其内部所进行的内容的纲要, 以及框架是如何使用由宏生成的代码和数据将消息转换为对对应类成员函数的调用的。

MFC 的 DECLARE_MESSAGE_MAP 宏在类声明中添加 3 个成员: 一个名为 _messageEntries 的私有的 AFX_MSGMAP_ENTRY 结构数组, 其中包含将消息与消息处理程序相关联的信息; 一个名为 messageMap 的静态 AFX_MSGMAP 结构, 其中包含一个指向类中的 _messageEntries 数组的指针和一个指向基类中的 messageMap 结构的指针; 以及一个名为 GetMessageMap 的虚拟函数, 该函数返回 messageMap 的地址(对于一个动态面不是静态链接到 MFC 的 MFC 应用程序, 其宏执行稍微有些不同, 但工作原理是相同的)。BEGIN_MESSAGE_MAP 包含 GetMessageMap 函数的实现, 和用来初始化 messageMap 结构的代码。BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 之间出现的宏填入到 _messageEntries 数组中, 而 END_MESSAGE_MAP 使用一个 NULL 条目标记了数组的结尾。对于下面的语句:

```
// In the class declaration
DECLARE_MESSAGE_MAP ()

// In the class implementation
BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_PAINT ()
END_MESSAGE_MAP ()
```

编译器的预处理程序将生成下面的代码:


```

// In the class declaration
private:
    static const AFX_MSGMAP_ENTRY _messageEntries[];
protected:
    static const AFX_MSGMAP messageMap;
    virtual const AFX_MSGMAP* GetMessageMap() const;
// In the class implementation
const AFX_MSGMAP* CMainWindow::GetMessageMap() const
    { return &CMainWindow::messageMap; }

const AFX_MSGMAP CMainWindow::messageMap = {
    &CFrameWnd::messageMap,
    &CMainWindow::_messageEntries[0]
};

const AFX_MSGMAP_ENTRY CMainWindow::_messageEntries[] = {
    { WM_PAINT, 0, 0, 0, AfxSig_vv,
      (AFX_PMSG)(AFX_PMSGW)(void (CWnd::*)(void))OnPaint },
    { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
};

```

只要这个基础结构的位置合适,框架就可以调用 GetMessageMap 来获取一个指向 CMainWindow 的 messageMap 结构的指针。然后它可以搜索 _messageEntries 数组来查看 CMainWindow 是否具有此消息的处理程序。此外,如果需要,它还能够维持一个指向 CFrameWnd 的 messageMap 结构的指针并搜索基类的消息映射。

下面就是对当一个 CMainWindow 的消息抵达时所发生事件的相当详细的描述。要分派此消息,框架调用了 CMainWindow 从 CWnd 继承下来的虚拟 WindowProc 函数。WindowProc 调用 OnWndMsg,而 OnWndMsg 又调用 GetMessageMap 来获取一个指向 CMainWindow::messageMap 的指针,并搜索 CMainWindow::_messageEntries 来获取一个其消息 ID 与当前正等待处理的 ID 相匹配的条目。如果找到了该条目,对应的 CMainWindow 函数(其地址与该消息 ID 一同存储在 _messageEntries 数组中)就被调用。否则,OnWndMsg 参考 CMainWindow::messageMap 获得一个指向 CFrameWnd::messageMap 的指针并为基类重复该过程。如果基类没有该消息的处理程序,则框架将上升一个级别,参考基类的基类,相当系统地沿着继承链向上走,直到它找到一个消息处理程序或者将该消息传递给 Windows 进行默认处理为止。图 1-5 从 CMainWindow 的消息映射条目开始,用示意图阐明了 CMainWindow 的消息映射,并表明了框架在搜索一个匹配特定消息 ID 的处理程序时所经过的路径。

MFC 的消息映射机制的作用相当于,它是将消息连接到消息处理程序而不使用虚拟函数的一种非常有效的方式。虚拟函数在空间上并不有效,因为它们需要虚表,而且即使虚表中的函数没有被覆盖,虚表也会消耗内存。相反,消息映射所使用的内存的数量与它所包含的条目的个数成比例。由于程序员要执行一个包含所有不同消息类型的处理程序的窗口类

是非常少见的,因此消息映射只是大约在每当 CWnd 使用 HWND 包装时保持几百字节的内存。

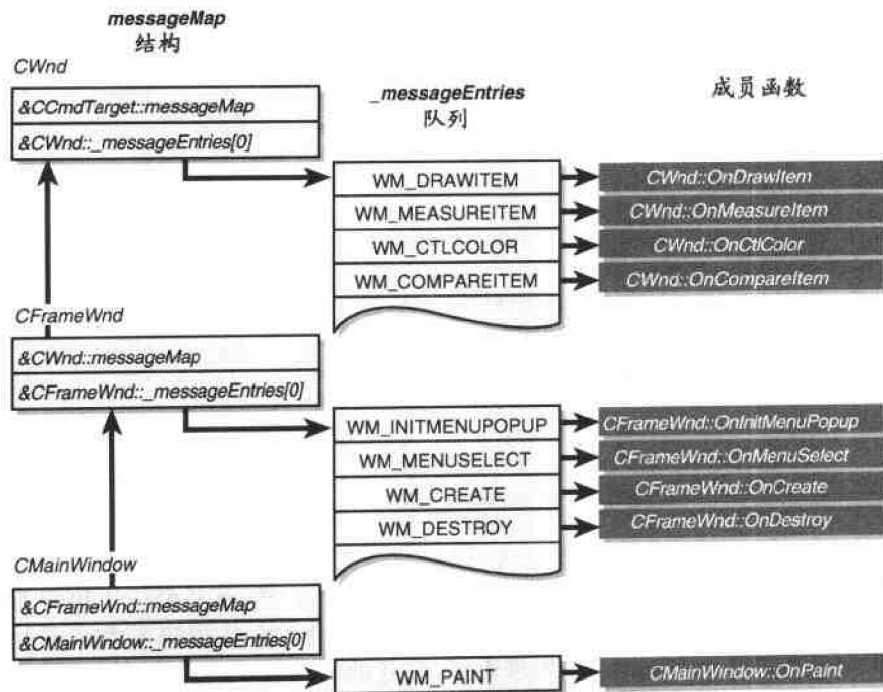


图 1-5 消息映射处理

1.3.7 Windows、字符集和 _T 宏

Microsoft Windows 98 和 Microsoft Windows NT 使用两种不同的字符集来构成字符和字符串。Windows 98 及其以前的版本使用 8 位的 ANSI 字符集,它类似于许多程序员都熟悉的 ASCII 字符集。Windows NT 和 Windows 2000 使用 16 位的 Unicode 字符集,它是 ANSI 字符集的一个超集。Unicode 适用于国际市场上销售的应用程序,因为它包含各种各样来自非 U.S. 字母表的字符。使用 ANSI 字符编译的程序可以在 Windows NT 和 Windows 2000 上运行,但 Unicode 程序运行起来要稍微快点,因为 Windows NT 和 Windows 2000 不需要在每个字符上执行 ANSI 到 Unicode 的转换。Unicode 应用程序不能在 Windows 98 运行,除非您将每个传递给 Windows 的字符串从 Unicode 转换为 ANSI 格式。

在编译应用程序时,您可以使用要么 ANSI 要么 Unicode 字符进行编译。如果您的应用程序将在 Windows 98 和 Windows 2000 上部署,它可能需要您将字符串变为与字符集无关的。然后,通过对工程的生成设置进行简单的更改或者在头文件中添加 #define,您可以告诉编译器是产生 ANSI 版本还是产生 Unicode 版本。如果您按如下方式将字符串常量编码:

```
"Hello"
```

那么编译器将从 ANSI 字符组成该字符串。如果您按如下方式声明了字符串：

```
L"Hello"
```

那么编译器将使用 Unicode 字符。但是如果您使用 MFC 的 `_T` 宏,如下所示：

```
_T("Hello")
```

如果定义了预处理程序符号 `_UNICODE`,那么编译器将使用 Unicode 字符,而如果没有定义该预处理程序符号,那么编译器将使用 ANSI 字符。如果您所有的字符串常量都使用 `_T` 宏声明,那么您可以通过定义 `_UNICODE` 生成一个特殊的仅适用于 Windows NT 的版本。隐式定义这个符号将定义一个名为 `UNICODE`(没有下划线)的相关符号,它将选择众多 Windows API 函数中的 Unicode 版本,这些 API 函数具有 ANSI 和 Unicode 两种版本。当然,如果您希望相同的可执行程序可以在这两种平台上运行,而且您不关心 ANSI 应用程序在 Windows NT 下导致的性能下降,那么您可以忘掉 `_T` 宏。在本书中我将通篇使用 `_T`,以使示例代码与字符集无关。

使用 `_T` 宏修饰字符串常量就足以使一个应用程序完全不关心其字符集吗? 回答是并不一定。您还必须做下面的工作：

- 将字符声明为 `TCHAR` 类型而不是 `char` 类型。如果定义了 `_UNICODE` 符号, `TCHAR` 将求值为 `wchar_t`,它是一个 16 位的 Unicode 字符。如果没有定义 `_UNICODE`, `TCHAR` 将变为普通古老的 `char`。
- 不要使用 `char*` 或者 `wchar_t*` 来声明 `TCHAR` 字符串的指针,而应当使用 `TCHAR*`,或者更佳的 `LPTSTR`(指向 `TCHAR` 字符串的指针)和 `LPCTSTR`(指向 `const TCHAR` 字符串的指针)数据类型。
- 不要认为一个字符只有 8 位宽。如果要将以字节表示的缓冲区长度转变为以字符表示的缓冲区大小,可以借助 `sizeof(TCHAR)` 划分缓冲区长度。
- 将对 C 运行时间库(例如, `strcpy`)中字符串函数的调用替换为 Windows 头文件 `Tchar.h`(例如, `_tstrcpy`)中的对应宏。

考虑下面的代码片段,其中使用了 ANSI 字符集：

```
char szMsg[256];
pWnd->GetWindowText (szMsg, sizeof (szMsg));
strcat (szMsg, " is the window title");
MessageBox (szMsg);
```

如果将上面的代码修订为与字符集无关,就是下面的代码：

```
TCHAR szMsg[256];
pWnd->GetWindowText (szMsg, sizeof (szMsg) / sizeof (TCHAR));
```

```
_tcscat (szMsg, _T (" is the window title"));
MessageBox (szMsg);
```

修订的代码使用普通的 TCHAR 数据类型,它不需要关心一个字符的大小,而且它使用更为常用的 TCHAR 兼容的字符串连接函数 _tcscat,而不是与 ANSI 字符集相关的 strcat。

关于 ANSI/Unicode 兼容的问题可以有更多的内容,但以上这些是问题的根本所在。要获取其他的信息,请参考随 Visual C++ 一同发布的联机文档,或者 Jeffrey Richter 编著的《Advanced Windows》(1997, Microsoft Press),其中包含了关于 Unicode 的非常好的章节内容和一个列出了 Tchar.h 中定义的字符串宏及其 C 运行时对应物的方便表格。

1.3.8 建立应用程序

本书附带的 CD-ROM 中包含了您在 Visual C++ 中使用 Hello 程序所需的全部内容。名为 \Chap01 \Hello 的文件夹包含了该程序的源代码和构成 Visual C++ 工程的所有文件。要打开该工程,只需从 Visual C++ 的 File 菜单中选择 Open Workspace,然后打开 Hello.dsw。如果您修改了应用程序,并希望重新生成,您可以从 Build 菜单中选择 Build Hello.exe。

您可以不必使用 CD-ROM 中的 Hello 文件。如果您愿意,您可以创建自己的工程和源代码中输入。下面就是在 Visual C++ 6 中创建一个新工程的步骤:

1. 选择 Visual C++ File 菜单中的 New,然后单击 Projects 标签,跳到 Projects 页面。
2. 选择 Win32 Application,然后在 Project Name 文本框中输入一个工程名。如果需要,您可以在 Location 文本框中更改路径名——工程及其源代码将存储的驱动器和文件夹。然后单击 OK。
3. 在 Win32 Application 窗口中,选择 An Empty Project,然后单击 Finish。
4. 将源代码文件添加到工程中。要从头开始输入一个源代码文件,请选择 File 菜单中的 New,选择文件类型,然后输入文件名。要确保选中了 Add To Project 框,这样该文件将添加到工程中。如果要将现有的源代码文件添加到工程中,请单击 Project 菜单,依次选择 Add To Project 和 Files,然后选择该文件。
5. 从 Project 菜单中选择 Settings。在 Project Settings 对话框中,要确保在左窗格中选择了该工程名,如果 General 页面没有显示出来,请单击 General 标签。从标签为 Microsoft Foundation Classes 的下拉列表中选择 Use MFC In A Shared DLL,然后单击 OK,以在 Visual C++ 中注册相应更改。

选择 Use MFC In A Shared DLL 将通过允许从 DLL 访问 MFC 使应用程序的可执行文件的大小达到最小。但是如果您选择 Use MFC In A Static Library,那么 Visual C++ 会将 MFC 代码链接到您的应用程序的 EXE 文件中,文件的大小将显著增长。静态链接使用磁盘空间的效率要比动态链接低,因为一个包含 10 个静态链接的 MFC 应用程序的磁盘中就包含了同一 MFC 库代码的 10 个拷贝。另一方面,静态链接的应用程序可以在任何 PC 上运行,不管

该 MFC DLL 是否存在。选择静态还是动态链接到 MFC 由您自己决定,但要记住:如果您发布一个动态链接的 EXE,您还将需要发布那个包容 MFC 的 DLL。对于使用 Visual C++ 版本 6 创建的发布版本的 MFC 应用程序来说,如果程序使用 ANSI 字符,则该 DLL 的名称为 Mfc42.dll;如果程序使用 Unicode 字符,则该 DLL 的名称为 Mfc42u.dll。

1.3.9 小结

我们在继续之前,首先停下来,回顾一下从 Hello 应用程序中学到的几个重要概念。当启动应用程序时发生的第一件事就是创建了一个全局范围的应用程序对象。MFC 的 AfxWinMain 函数调用该应用程序对象的 InitInstance 函数。InitInstance 构造了一个窗口对象,同时该窗口对象的构造函数创建了出现在屏幕上的窗口。创建了窗口之后,InitInstance 调用窗口的 ShowWindow 函数使它可见,并调用 UpdateWindow 函数给窗口发送第一个 WM_PAINT 消息。然后 InitInstance 返回,AfxWinMain 调用应用程序对象的 Run 函数启动消息循环。WM_PAINT 消息通过 MFC 的消息映射机制转换为对 CMainWnd::OnPaint 的调用,然后 OnPaint 通过创建一个 CPaintDC 对象和调用其 DrawText 函数,在窗口的客户区绘制了文本“Hello, MFC”。

如果您直接从 Windows SDK 使用 MFC,那么事情就可能变得相当古怪了。两步创建窗口? 应用程序对象? 不再需要 WinMain? 这与过去的 Windows 编程方式显然不同。但如果将 Hello 的源代码与前面的图 1-2 中的 C 程序清单相比较,您就会发现 MFC 不可否认地简化了事情。MFC 并不一定使源代码更易于理解——毕竟,Windows 编程还是 Windows 编程——但是,通过将大量的样板文件材料从源代码中移出而移入到类库中,MFC 减少了您所必须编写的代码的数量。这种性质,以及您可以通过从任何 MFC 类派生出您自己的类来改变 MFC 类的行为这一事实,使得 MFC 成为进行 Windows 编程的有效工具。当您开始接触到 Windows 的某些更高级特性或者创建 ActiveX 控件和其他基于 Windows 的软件组件时,这些优点将变得非常明显。使用 MFC,您就可以轻松自如地获得 ActiveX 控件。如果不使用 MFC——那就只能祝您好运了。

Hello 缺少构成一个功能完善的 Windows 程序的许多元素,但它却是逐步成长为一名 MFC 程序员的良好开端。在下面的章节中,您将了解到菜单、对话框和应用程序的用户界面的其他组件。您还将看到 Windows 程序是如何从鼠标和键盘读取用户输入的,以及关于在窗口中绘制的更多内容。第 2 章的开始部分是介绍一些其他的 CDC 绘图函数,以及演示了将滚动条添加到框架窗口的方法,这样您就可以查看到比窗口的客户区大得多的工作空间。这些都是积累成为一名 Windows 程序员所需的知识基础的下一个重要步骤。

第 2 章 在窗口中绘图

如果您一直使用电脑,那么您可能会记得,在 Microsoft Windows 产生之前图形编程是如何困难吧。那时如果运气好,您可以借助于较像样的图形库,用 DrawLine 和 DrawCircle 这样的程序来画些基本图形元素。若运气不好,您可能需要花大量时间编写自己的输出程序,并使它们不得不在这儿或那儿多花几毫秒。而且,您也知道,在过去无论是用自己还是别人编的代码画图,当出现新的图形标准,这在当时意味着 IBM 引入新的图形适配器如 EGA 或 VGA 时,您都要想办法支持最新的硬件。这意味着要买升级版的图形库,将新代码加入自己的程序,或给新视卡编写驱动程序。对图形编程人员来说,其工作平台就像是一个移动的标靶,从未在相当一段时间内保持不动。即便您成功地在可视屏幕上画出一粒珠子,那么为使程序与打印机和其他输出设备有适当的接口,也还有大量的工作需要做。

通过给微机平台引入其所需的与硬设无关的图形输出模式,Windows 改变了上述的被动局面。在 Windows 中,只要有相应的 Windows 驱动程序,您所编写的代码就可在任一图像适配器上运行。如今它可用于所有的适配器。在很大程度上,将输出发送到屏幕上的代码同样也可用于打印机和其他硬拷贝设备上。图形编程中这种方便省事的方法有许多优点。其中最主要的是,编程人员不再为程序运行的具体硬件环境费心了,而可将时间都用在编制应用程序代码上。其次,完成图形编程工作不再需要第三方的图形库,Windows 提供了广泛的图形 API 函数,它们可实现简单到画线,复杂到创建剪贴区,以作为其他输出例程的模板。

Windows 中负责图形输出的是 Graphics Device Interface(图形设备接口),或称 GDI。GDI 提供应用程序可调用的多种服务。这些服务一起构成了一种强大和通用的图形编程语言,其丰富的功能足可与一些第三方图形库相比。MFC 在图形 API 基础上工作,并用表示 Windows GDI 各成员的 C++ 类将接口转化为程序代码。

既然您已知道如何创建窗口,那么现在就可利用窗口做些事情了。第 1 章中的“Hello”应用程序使用了 CDC::DrawText 命令将文本输出到窗口。DrawText 只是 CDC 类为文本和图形输出提供的多个成员函数之一。本章将详细讲述 CDC 类和它的继承类,并介绍 3 种最常用的 GDI 基本元素:画笔、笔刷和字体,同时还将示范如何在窗口中添加滚动条。

2.1 Windows GDI

在单任务环境如 MS-DOS 中,运行中的应用程序随时可自由地做它想做的事情,无论是在屏幕上画一条线,重新编写适配器的调色板,还是转换到另一种图像模式。而在窗口化多

任务环境如 Windows 中,程序则无此自由。因为程序 A 的输出必须与程序 B 的输出相隔离。首先这意味着各程序的输出必须限制在自己的窗口中。GDI 使用一简单的机制保证在窗口中画图的各程序遵循这些规则。这种机制即为设备描述表(DC)。

当 Windows 程序在屏幕、打印机或其他输出设备上画图时,它并不是将像素直接输出到设备上,而是将图绘制到由设备描述表(DC)表示的逻辑意义上的“显示平面”上去。设备描述表是深寓于 Windows 中的一种数据结构,它包含 GDI 需要的所有关于显示平面情况的描述字段,包括相连的物理设备和各种各样的状态信息。在平面上画图之前,Windows 程序从 GDI 获取设备描述表句柄,并在每次调用 GDI 输出函数时将句柄返回给 GDI。若无有效的设备描述表句柄,则 GDI 不会画第一个像素点。通过设备描述表,GDI 可确保程序所画的任何图形都能剪贴到屏幕的特定区域。设备描述表在使 GDI 摆脱设备限制的过程中发挥了重要的作用。获得设备描述表句柄后,同一 GDI 函数可用来向多种输出设备上画图。

在使用 MFC 编制 Windows 程序时,设备描述表具有更加突出的作用。除了可作为通往各种输出设备的桥梁之外,设备描述表对象还封装了程序用来产生输出的 GDI 函数。在 MFC 中,您不用捕获设备描述表句柄和调用 GDI 输出函数,至少不必直接捕获和调用,而是通过创建一设备描述表对象并调用它的成员函数来画图。MFC 的 CDC 类将 Windows 设备描述表和获取设备描述表句柄的 GDI 函数就近封装在一起,而 CDC 派生类如 CPaintDC 和 CClientDC 则代表 Windows 应用程序使用的不同类型的设备描述表。

2.1.1 MFC 设备描述表类

在 MFC 应用程序中获取设备描述表的一种方法是调用 CWnd::GetDC,它返回指向表示 Windows 设备描述表的 CDC 对象的指针。在画图完毕时,要用 CWnd::ReleaseDC 释放由 CWnd::GetDC 获取的设备描述表指针。下面的程序代码由 GetDC 获取 CDC 指针,而后画图并最终调用 ReleaseDC 释放设备描述表:

```
CDC * pDC = GetDC ();
// Do some drawing
ReleaseDC (pDC);
```

若同样的程序代码出现在 OnPaint 处理程序中时,则需用 CWnd::BeginPaint 和 CWnd::EndPaint 分别代替 GetDC 和 ReleaseDC,以保证合理地处理 WM_PAINT 消息:

```
PAINTSTRUCT ps;
CDC * pDC = BeginPaint (&ps);
// Do some drawing
EndPaint (&ps);
```

GDI 还支持存储 GDI 命令序列的元文件,这些命令可重新执行以产生实际输出。为获取元文件输出的设备描述表,还要使用另一套函数来获取和释放 CDC 指针。而且,为获取

允许在窗口内任一地方画图的设备描述表 CDC 指针(与只允许在窗口客户区画图的设备描述表 CDC 指针不同),需要调用 `CWnd::GetWindowDC` 而不是 `GetDC`,但仍用 `ReleaseDC` 来释放设备描述表。

为避免要记住获取和释放设备描述表时需调用的函数(并且为确保在使用设备描述表的消息处理程序结束时设备描述表能合理地被释放),MFC 提供了一些 CDC 派生类,如表 2-1 所示。

表 2-1 专门用途的设备描述表类

| 类名 | 描 述 |
|--------------------------|---|
| <code>CPaintDC</code> | 用于在窗口客户区画图(仅限于 <code>OnPaint</code> 处理程序) |
| <code>CClientDC</code> | 用于在窗口客户区画图(除 <code>OnPaint</code> 外的任何处理程序) |
| <code>CWindowDC</code> | 用于在窗口内任意地方画图,包括非客户区 |
| <code>CMetaFileDC</code> | 用于向 GDI 元文件画图 |

这些类被设计为可直接进行实例化。各个类的构造函数和析构函数调用相应的函数捕获和释放设备描述表,从而使得设备描述表的使用非常方便简捷:

```
CPaintDC dc(this);
// Do some drawing
```

传送给类构造函数的指针确定了设备描述表所属的窗口。

当在栈上构造设备描述表对象时,若对象的生命周期结束,则它的析构函数会被自动调用。而且析构函数一旦被调用,设备描述表就会被返回给 Windows。在堆上用 `new` 创建设备描述表时,要注意亲自释放设备描述表。示例如下:

```
CPaintDC * pDC = new CPaintDC(this);
```

在这种情况下,有必要在创建设备描述表的函数结束之前执行删除语句

```
delete pDC;
```

以便调用对象的析构函数和释放设备描述表。在某些场合下,在堆上创建设备描述表要比在栈上创建更有用,但通常在栈上创建设备描述表对象并让编译程序执行删除任务会使您轻松很多。

CPaintDC 类

MFC 的 `CPaintDC` 类响应 `WM_PAINT` 消息,允许您在窗口客户区画图。但您只能在 `OnPaint` 处理程序中,而不能在其他地方使用它。`WM_PAINT` 消息在一个很重要的方面不同于其他 Windows 消息;如果处理程序调用 Windows 的 `::BeginPaint` 和 `::EndPaint` 函数失败(或 MFC 等价函数, `CWnd::BeginPaint` 和 `CWnd::EndPaint`),那么不管有多少绘图工作,都不能将

该消息从消息队列中删除。因此,应用程序将一遍又一遍地处理同一个 WM_PAINT 消息而陷入死循环。而通过分别从 CPaintDC 的构造函数和析构函数中调用 ::BeginPaint 和 ::EndPaint,CPaintDC 能保证这种情况不会发生。

CClientDC 和 CWindowDC 类

Windows 程序不是总将绘图限制在 OnPaint 上。如果您编写了这样一段应用程序,只要一单击鼠标,应用程序就会在屏幕上画一个圆圈,那么您可能希望在接收到按钮单击消息时立刻画圆,而不必等到 WM_PAINT 消息。

这正是 MFC CClientDC 类的目的。CClientDC 创建了可在 OnPaint 外使用的用户区域设备描述表。下面的消息处理程序使用了 CClientDC 和两个 CDC 成员函数,来完成了在鼠标左键被单击时画一个 X 连接窗口客户区的四角的功能。

```
void CMainWindow::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;
    GetClientRect(&rect);

    CClientDC dc(this);
    dc.MoveTo(rect.left, rect.top);
    dc.LineTo(rect.right, rect.bottom);
    dc.MoveTo(rect.right, rect.top);
    dc.LineTo(rect.left, rect.bottom);
}
```

left, right, top 和 bottom 是在 MFC 的 CRect 类中定义的公用成员变量,保存着矩形四边的坐标值。MoveTo 和 LineTo 是 CClientDC 从 CDC 继承来的画线函数。在后续的章节中,你很快就会详细了解这两个函数。

如果您不仅要使用窗口客户区,还要使用非客户区(标题栏、窗口边框等),则 MFC 提供有 CWindowDC 类。CWindowDC 与 CClientDC 相类似,但它代表的设备描述表包含了窗口边框之内的所有内容。有时程序员可以用 CWindowDC 创造特殊效果,例如用户自己绘制标题栏和带圆角的窗口。一般情况下 CWindowDC 并不常用。如果想在窗口非客户区作图,您可借助 OnNcPaint 处理程序捕获 WM_NCPAINT 消息,确定非客户区需要绘制的时间。与 OnPaint 不同,OnNcPaint 处理程序不需要(也不应当)调用 BeginPaint 和 EndPaint。

更少见的场合是程序需要全屏幕的访问权。此时可创建 CClientDC 或 CWindowDC 对象,并给其构造函数传送一个 NULL 指针。语句

```
CClientDC dc(NULL);
dc.Ellipse(0, 0, 100, 100);
```

将在屏幕左上角画一个圆。屏幕截取程序经常使用全屏 DC 访问整个屏幕。很显然,

除非有特殊情况,否则在自己的窗口外画图是件很不友好的事情。

2.1.2 设备描述表属性

当使用 CDC 输出函数在屏幕上画图时,输出的某些特性并没有在函数调用过程中规定,但可通过设备描述表自身获得。例如,在调用 CDC::DrawText 时,要指定待输出的字符串和显示该字符串的矩形区域,但不必规定文本颜色和字体,因为它们是设备描述表的属性。表 2-2 列出了一些设备描述表中最常用的属性和访问这些属性的 CDC 函数。

表 2-2 主要设备描述表属性

| Attribute | Default | Set with | Get with |
|-----------|-------------|-------------------|-------------------------|
| 文本颜色 | Black | CDC::SetTextColor | CDC::GetTextColor |
| 背景颜色 | White | CDC::SetBkColor | CDC::GetBkColor |
| 背景模式 | OPAQUE | CDC::SetBkMode | CDC::GetBkMode |
| 映射模式 | MM_TEXT | CDC::SetMapMode | CDC::GetMapMode |
| 绘图模式 | R2_COPYPEN | CDC::SetROP2 | CDC::GetROP2 |
| 当前位置 | (0,0) | CDC::MoveTo | CDC::GetCurrentPosition |
| 当前画笔 | BLACK_PEN | CDC::SelectObject | CDC::SelectObject |
| 当前画刷 | WHITE_BRUSH | CDC::SelectObject | CDC::SelectObject |
| 当前字体 | SYSTEM_FONT | CDC::SelectObject | CDC::SelectObject |

不同的 CDC 输出函数以不同的方式使用设备描述表的属性。例如,在用 LineTo 画线时,当前的画笔决定了线的颜色、宽度和样式(实线、虚线、点划线等等)。类似地,在使用 Rectangle 函数画矩形时,GDI 用当前的画笔画矩形区域的边界,并用当前的笔刷填充该矩形区域。所有的文本输出函数都采用当前的字体。在文本输出时,文本颜色和背景颜色决定了所有用到的颜色。文本颜色决定了字符的颜色,而背景颜色决定字符后面的填充色。在使用 LineTo 函数画虚线或点划线时,背景颜色还用于填充线段间空隙,或用来填充阴影画笔所画标记间的空白处。如果想忽略背景颜色,可将背景模式设置为“transparent”(透明),如:

```
dc.SetBkMode(TRANSPARENT);
```

在第 1 章的 Hello 程序中,在 DrawText 调用之前加入该语句会消除环绕着“Hello, MFC”的白色的矩形。这个白色矩形在背景颜色不是白色的时候是可见的。

最常用来定义设备描述表属性的 CDC 函数是 SelectObject。下面所列的是 6 个 GDI 对象,可由 SelectObject 选入设备描述表。

- 画笔(Pen)
- 画刷(Brush)

- 字体(Font)
- 位图(Bitmap)
- 调色板(Palette)
- 区域(Region)

在 MFC 中,CPen、CBrush 和 CFont 类分别代表了画笔、笔刷和字体(位图、调色板和区域将在第 15 章中介绍)。除非调用 SelectObject 以改变当前画笔、笔刷或字体,否则 GDI 将使用设备描述表中的默认值。默认画笔可画出一个像素点宽的黑实线;默认笔刷填充单一的白色;默认字体是大约 12 个点高的相当普通的比例字体。您可以创建自己的画笔、笔刷和字体,并将它们选入设备描述表从而改变输出的属性。例如,画一个单一红色的圆,并使它具有 10 个像素点宽的黑边。这时要创建一个 10 个像素点宽的黑色画笔和一个红色笔刷,并在调用 Ellipse 之前用 SelectObject 将它们选入设备描述表。如果 pPen 是指向 CPen 对象的指针,pBrush 是指向 CBrush 对象的指针,而 dc 代表一个设备描述表,则程序代码如下所示:

```
dc.SelectObject(pPen);
dc.SelectObject(pBrush);
dc.Ellipse(0, 0, 100, 100);
```

重载 SelectObject 以便接受各种类型对象的指针。它的返回值为先前选入设备描述表中的相同类型对象的指针。

每当从 Windows 中获取设备描述表时,设备描述表都被设置为默认值。因此,如果想在响应 WM_PAINT 消息时使用红色画笔和蓝色笔刷画您的窗口,则每逢 OnPaint 被调用时都要将所需的画笔和笔刷选入设备描述表,也就是创建一个新的 CPaintDC 对象。否则,将使用默认的画笔和笔刷。如果不想在使用设备描述表时反复对它进行初始化设定,那么可用 CDC::SaveDC 函数保存它的状态,并在下次使用时用 CDC::RestoreDC 将它恢复。另一种方法是,注册一个自定义的 WNDCLASS,其中包含 CS_OWNDC 样式,它使 Windows 为每个应用程序实例分配它已设置好的设备描述表。(有一个相关但很少使用的 WNDCLASS 样式,CS_CLASSDC,它分配一个“半私有”设备描述表。该设备描述表可被同一 WNDCLASS 创建的所有窗口共享。)将红色画笔和蓝色笔刷选入某个私有设备描述表后,如果没有被显示地替换,则它们依旧处于选中状态。

2.1.3 绘图模式

GDI 将像素点输出到逻辑显示平面上时,它不只是简单地输出像素点颜色。相反,它通过一系列的布尔运算将输出像素点的颜色和输出目标位置上像素点的颜色合成在一起。它所使用的逻辑关系由设备描述表当前的绘图模式确定。使用 CDC::SetROP2(Set Raster Operation To 的缩写形式)可更改绘图模式。默认绘图模式为 R2_COPYPEN,它将像素点复制到显示平面上。下表显示了另外 15 种可以选择的绘图模式。所有这些绘图模式代表了

一切可能的由 AND、OR、XOR 和 NOT 构成的运算。

为什么总需要改变绘图模式呢? 如果您不是将像素点复制到显示平面上,而是通过反转已有像素点的颜色来画线,那就容易多了;只需在画线前将绘图模式设置为 R2_NOT 即可:

```
dc.SetROP2 (R2_NOT);
dc.MoveTo (0, 0);
dc.LineTo (100, 100);
```

这个小技巧或许比您想象的更有用,用它来做橡皮筋线和矩形是非常方便的。关于这一点,看了第3章的例子您就会明白。

表 2-3 GDI 绘图模式

| 绘图模式 | 执行的运算 |
|-----------------|---------------------------|
| R2_NOP | dest = dest |
| R2_NOT | dest = NOT dest |
| R2_BLACK | dest = BLACK |
| R2_WHITE | dest = WHITE |
| R2_COPYPEN | dest = src |
| R2_NOTCOPYPEN | dest = NOT src |
| R2_MERGEPEENNOT | dest = (NOT dest) OR src |
| R2_MASKPENNOT | dest = (NOT dest) AND src |
| R2_MERGENOTPEN | dest = (NOT src) OR dest |
| R2_MASKNOTPEN | dest = (NOT src) AND dest |
| R2_MERGEPEN | dest = dest OR src |
| R2_NOTMERGEPEN | dest = NOT (dest OR src) |
| R2_MASKPEN | dest = dest AND src |
| R2_NOTMASKPEN | dest = NOT (dest AND src) |
| R2_XORPEN | dest = src XOR dest |
| R2_NOTXORPEN | dest = NOT (src XOR dest) |

2.1.4 映射模式

毫无疑问,对 Windows 编程新手来说,GDI 编程中最困难的部分就是映射模式(mapping mode)。简单地说,映射模式是设备描述表的属性,用于确定从逻辑坐标值到设备坐标值的转换方式。传送给 CDC 输出函数的是逻辑坐标值。设备坐标值是指窗口中相应的像素点







位置。用

```
dc.Rectangle(0, 0, 200, 100);
```

调用 Rectangle 函数时您不用告诉 GDI: 画一个 200 个像素点宽、100 个像素点高的矩形,而是告诉它画一个 200 个单位宽、100 个单位高的矩形。在默认映射模式 MM_TEXT 下,一个像素点恰恰相当于一个单位。但在其他映射模式中,逻辑单位被解释为不同的设备单位。例如,在 MM_LOENGLISH 映射模式中,一个单位相当于一英寸的百分之一。因此在这种模式下画 200 单位宽、100 单位长的矩形,得到的是一个 2 英寸宽、1 英寸长的矩形。使用非 MM_TEXT 映射模式可以方便地按比例缩放输出,这时尺寸和距离与输出设备的物理分辨率无关。

Windows 支持 8 种不同的映射模式,其属性见表 2-4。

表 2-4 GDI 映射模式

| 映射模式 | 一个逻辑单位 对应的距离 | X 和 Y 轴的方向 |
|----------------|--------------------------|--|
| MM_TEXT | 1 像素 |  |
| MM_LOMETRIC | 0.1 mm |  |
| MM_HIMETRIC | 0.01 mm |  |
| MM_LOENGLISH | 0.01 in. |  |
| MM_HIENGLISH | 0.001 in. |  |
| MM_TWIPS | 1/1440 in. (0.000 7 in.) |  |
| MM_ISOTROPIC | 用户自定义 (x 和 y 同等缩放) | 用户自定义 |
| MM_ANISOTROPIC | 用户自定义 (x 和 y 独立缩放) | 用户自定义 |

在 MM_TEXT 映射模式下画图时,所使用的坐标系请见图 2-1。原点在窗口的左上角,X 轴的正向向右,Y 轴的正向向下,并且一个单位相当于一个像素点。如果转换成“公制”映射模式,如 MM_LOENGLISH、MM_HIENGLISH、MM_LOMETRIC、MM_HIMETRIC 或 MM_TWIPS,则 Y 轴会翻转使正向朝上,并且逻辑单位被按比例转换为实际距离大小,而不是像素数。然而,原点的位置在左上角保持不变。需要注意的是在使用公制映射模式时,为使输出可

见,Y坐标必须用负值。语句

```
dc.Rectangle(0, 0, 200, 100);
```

在 MM_TEXT 映射模式下画了一个 200 个像素点宽、100 个像素点长的矩形。但这个语句在 MM_LOENGLISH 映射模式下却得不到输出,这是因为正的 Y 坐标值不在窗口的可见部分。如果想使该矩形显示出来,请按照下面的方法给 Y 坐标取负值:

```
dc.Rectangle(0, 0, 200, -100);
```

如果在向非 MM_TEXT 映射模式切换时应用程序的输出突然不见了,那么请检查 Y 坐标值的正负号。问题一般总是出在正的 Y 坐标值身上。

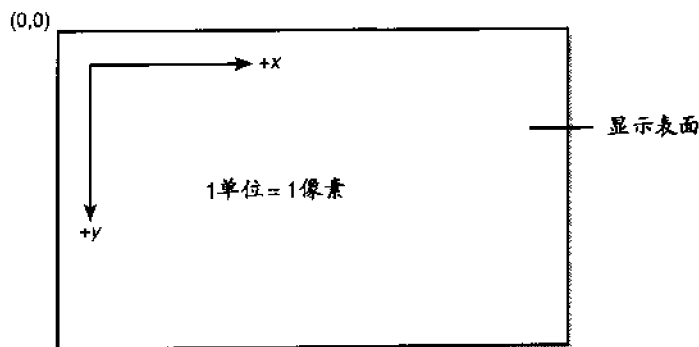


图 2-1 MM_TEXT 坐标系

MM_TEXT 为默认映射模式。如果要使用其他映射模式,可以调用 `CDC::SetMapMode` 来改变该默认映射模式。下列语句的功能是切换到 MM_LOMETRIC 映射模式,同时画一个长轴为 5 厘米、短轴为 3 厘米的椭圆:

```
dc.SetMapMode(MM_LOMETRIC);
dc.Ellipse(0, 0, 500, -300);
```

可以看出使用映射模式并没有特别的技巧。只是在使用 MM_ISOTROPIC 和 MM_ANISOTROPIC 模式时,以及在对非 MM_TEXT 映射模式下画的对象做命中测试时,情况才会显得稍微复杂些,然而做起来仍然并不困难。MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式将在下一小节介绍。

在使用公制映射模式时,需要注意的是,在显示屏幕上 1 逻辑英寸并不等于实际中的 1 英寸。也就是说,如果在 MM_LOENGLISH 映射模式下画一条 100 单位长的线,它可能不是正好 1 英寸长。原因在哪里? Windows 并不知道您的监视器的物理分辨率(dpi,是指监视器在水平或垂直方向上每英寸内可显示的点数)。(在将来的 Windows 版本中可能会改变这种情况。)打印机和其他硬拷贝设备则不同,打印驱动程序可以控制一个 600 dpi 的激光打印

机在1英寸内打出600个点,所以在MM_LOENGLISH映射模式下画的100个单位长的线在打印纸上的长度也正好是1英寸。

2.1.5 可编程映射模式

MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式在一个重要方面与其他映射模式不同,即是用户而不是 Windows 决定从逻辑坐标值转换成设备坐标值的方式。正因为如此,这两个映射模式有时也被称为“自行设计”或“可编程”映射模式。如果想建立1逻辑单位等于1厘米的映射模式,很简单,只要使用 MM_ANISOTROPIC 映射模式并设置相应的比例参数即可。

MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式最常用于根据窗口尺寸按比例自动调节画图的输出大小的场合。下面的程序代码段用 MM_ANISOTROPIC 映射模式画一个椭圆,使它与所在窗口的四边相接。

```
CRect rect;
GetClientRect(&rect);
dc.SetMapMode(MM_ANISOTROPIC);
dc.SetWindowExt(500, 500);
dc.SetViewportExt(rect.Width(), rect.Height());
dc.Ellipse(0, 0, 500, 500);
```

知道它如何起作用了吗?无论窗口的实际尺寸如何,Windows 被告知窗口的逻辑尺寸为500单位宽,500单位长。因此边界框从(0,0)伸展到(500,500),包含了整个窗口。按照这种方式初始化设备描述表,则得到的坐标系原点位于窗口左上角,X轴正向向右,Y轴正向向下。如果希望Y轴正向向上(如在公制映射模式下),那么只要将传送给 SetWindowExt 或 SetViewportExt 的Y坐标值取反,就可以达到倒转Y轴的目的了:

```
CRect rect;
GetClientRect(&rect);
dc.SetMapMode(MM_ANISOTROPIC);
dc.SetWindowExt(500, -500);
dc.SetViewportExt(rect.Width(), rect.Height());
dc.Ellipse(0, 0, 500, -500);
```

现在必须用负的Y坐标值在窗口中画图了。只有 MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式允许倒转X轴和Y轴的方向。正是因为这个原因,这两种映射模式的坐标轴方向才被定义为用户自定义类型。

MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式间唯一的区别在于:前者中X方向和Y方向具有同一个的缩放比例因子。也就是说,水平方向上的100个单位和竖直方向上的100个单位实际上长度相等。Isotropic 意味着“各个方向上相等”。用 MM_ISOTROPIC 映射

模式画圆和正方形是非常理想的。下面的程序代码是画一个圆,该圆直径为窗口宽度和高度中较小的那一个:

```
CRect rect;
GetClientRect (&rect);
dc.SetMapMode (MM_ISOTROPIC);
dc.SetWindowExt (500, 500);
dc.SetViewportExt (rect.Width (), rect.Height ());
dc.Ellipse (0, 0, 500, 500);
```

对于 Windows,再次将窗口的逻辑尺寸设为 500 单位×500 单位。然而,在把逻辑单位转换为设备单位时,GDI 将输出设备的长宽比也考虑进去了。第 14 章中的 Clock(时钟)程序用 MM_ISOTROPIC 映射模式画了一个圆的时钟,并使它随窗口的尺寸大小变化而自动缩放。如果没有 MM_ISOTROPIC 映射模式的帮助,就只能手动调整时钟的大小了。

现在简单介绍一下 SetWindowExt 和 SetViewportExt 函数。规定: SetWindowExt 设定“窗口范围”;SetViewportExt 设定“视口范围”。可以这样认为,窗口的尺寸以逻辑单位计算,视口的尺寸以设备单位或像素点计算。当 Windows 对逻辑坐标值和设备坐标值进行相互转换时,使用一对公式,其中包含窗口逻辑尺寸(窗口范围)、实际尺寸(视口范围)以及坐标原点位置。设定窗口范围和视口范围时,实际上是在自己的缩放比例参数内编程。一般说来,视口范围是画图所在窗口的大小(以像素点数目计算),而窗口范围是指以逻辑单位表示的窗口尺寸。

使用 SetWindowExt 和 SetViewportExt 时要注意:在 MM_ISOTROPIC 映射模式下,应该首先调用 SetWindowExt。否则,部分窗口客户区可能会因落在窗口的逻辑范围之外而不能使用。而在 MM_ANISOTROPIC 映射模式下,窗口范围和视口范围中先设置哪个都无关紧要。

2.1.6 坐标转换

调用 CDC::LPtoDP 函数可将逻辑坐标值转换为设备坐标值。反之,调用 CDC::DPtoLP 函数可将设备坐标值转换为逻辑坐标值。

如果您想得到以设备坐标值表示的窗口中心点位置,则只需取用像素点表示的窗口长度和宽度的平分值即可。CWnd::GetClientRect 返回以像素点表示的窗口尺寸。

```
CRect rect;
GetClientRect (&rect);
CPoint point (rect.Width () / 2, rect.Height () / 2);
```

如果想得到以 MM_LOENGLISH 单位表示的窗口中心点位置,则需使用 DPtoLP 函数。

```
CRect rect;
GetClientRect (&rect);
CPoint point (rect.Width () / 2, rect.Height () / 2);
```



```
CClientDC dc (this);
dc.SetMapMode (MM_LOENGLISH);
dc.DPtoLP (&point);
```

当 DPtoLP 返回时, point 中存储了以逻辑(即 MM_LOENGLISH)坐标值表示的中心点坐标。另一方面,如果想得到 MM_LOENGLISH 坐标值为(100,100)的点的像素点坐标值,则可调用 LPtoDP。

```
CPoint point (100, 100);
CClientDC dc (this);
dc.SetMapMode (MM_LOENGLISH);
dc.LPtoDP (&point);
```

在响应鼠标单击的命中测试中, LPtoDP 和 DPtoLP 是必不可少的。通常鼠标单击后得到的是设备坐标值,所以如果您已经用 MM_LOENGLISH 坐标值画了一个矩形,并且想知道鼠标单击是否发生在这个矩形之内,则需将矩形的逻辑坐标值转换为设备坐标值,或将单击鼠标得到的设备坐标值转换为逻辑坐标值。否则,您就是在比较不能相提并论的两个事物。

2.1.7 移动原点

默认方式下,设备描述表的原点位于显示平面的左上角。即使改变映射模式,也不会改变原点的位置。然而,同改变映射模式一样,您也可以移动原点。MFC 的 CDC 类提供了两个可移动原点的函数。CDC::SetWindowOrg 移动窗口的原点, CDC::SetViewportOrg 移动视口的原点。正常情况下,只能使用其中之一。同时使用两个会搞得一团糟。

假设您想将原点移到窗口中心点,以通过将输出中心调整到(0,0)点来使所画的图形处于窗口中间。假设 dc 是一个设备描述表对象,可用下面的方法达到上述目的:

```
CRect rect;
GetClientRect (&rect);
dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);
```

下面还有另一种方法可做到这一点。假设您是在 MM_LOENGLISH 映射模式下工作的:

```
CRect rect;
GetClientRect (&rect);
CPoint point (rect.Width () / 2, rect.Height () / 2);
dc.SetMapMode (MM_LOENGLISH);
dc.DPtoLP (&point);
dc.SetWindowOrg (-point.x, -point.y);
```

虽然 SetViewportOrg 和 SetWindowOrg 很容易混淆,但实际上两者间的区别很明显。用 SetViewportOrg 将视口原点移至 (x,y) 等价于通知 Windows 把逻辑点 (0,0) 映射成设备点 (x,y)。用 SetWindowOrg 将窗口原点移至 (x,y) 则恰恰相反,它等价于告诉 Windows 将逻辑

点 (x,y) 映射成设备点 $(0,0)$,即显示平面的左上角。在MM_TEXT映射模式下,两个函数间的区别就在于 x 和 y 的正负号。在其他映射模式下,区别就更显著了,因为SetViewportOrg处理设备坐标值,而SetWindowOrg处理逻辑坐标值。本章后面部分给出了使用这两个函数的例子。

最后一个例子。假定目前在MM_HIMETRIC映射模式下画图,这里一个单位等价于1/100毫米, x 轴正向向右, y 轴正向向上,现在想将原点移至窗口的左下角。下面有一种方法很容易做到这一点:

```
CRect rect;
GetClientRect(&rect);
dc.SetViewportOrg(0, rect.Height());
```

这样您就可以用原点在窗口左下角,并且 x 和 y 均为正的坐标值画图了。

2.1.8 坐标系小结

在谈到映射模式、窗口原点、视口原点以及其他与GDI处理坐标值有关的惯用语时,很容易把它们混淆起来。理解设备坐标系和逻辑坐标系间的区别有助于理清这些概念。

在设备坐标系中,距离长短以像素点数目来计量。设备上的 $(0,0)$ 点始终在显示平面的左上角, x 轴正向向右, y 轴正向向下。逻辑坐标系则全然不同。原点可放在任一位置,并且随着映射模式的不同, x 轴、 y 轴方向以及缩放比例因子(相当于一个逻辑单位的像素点数目)都会发生变化。更明白地说,它们随窗口范围和视口范围的不同而发生变化。MM_ISOTROPIC和MM_ANISOTROPIC映射模式允许改变这两个范围的大小,但其他映射模式则不允许。

有时,您会听到Windows程序员谈论“用户坐标值”和“屏幕坐标值”。用户坐标值是原点设立在窗口客户区左上角的设备坐标值。屏幕坐标值是原点位于屏幕左上角的设备坐标值。调用CWnd::ClientToScreen和CWnd::ScreenToClient函数可实现用户坐标值与屏幕坐标值之间的转换。当您第一次调用一个返回屏幕坐标值的Windows函数,并需要将返回的屏幕坐标值再传送给需要用户坐标值的函数(或者反过来)时,上面两个函数的用处就很明显了。

2.1.9 获取设备信息

有时,在输出之前捕获一些设备信息是很有帮助的。CDC::GetDeviceCaps函数可帮助用户检索到设备的各种信息,从设备支持的颜色数目到水平和竖直方向上可显示的像素点的数目。下面的程序代码实现屏幕宽和高的初始化,它将宽设为 cx ,高设为 cy ,都以像素点数目计算:

```
CClientDC dc(this);
int cx = dc.GetDeviceCaps(HORZRES);
```

```
int cy = dc.GetDeviceCaps(VERTRES);
```

如果屏幕分辨率为 $1\,024 \times 768$, 则 `cx` 和 `cy` 将分别被设为 1 024 和 768。

下表中列出了一些参数。将这些参数传送给 `GetDeviceCaps` 能获得与设备描述表有关的实际输出设备方面的信息。如何理解这些返回值, 在某种程度上取决于设备的类型。例如: 调用 `GetDeviceCaps` 并将屏幕 DC 的 `HORZRES` 参数传送给它, 则函数返回以像素点数目表示的屏幕宽。对打印机 DC 做如此调用, 则得到的是可打印页面的宽度, 同样是以像素点数目表示的。一般说来, 缩放比例值(例如, `LOGPIXELSX` 和 `LOGPIXELSY`)能返回打印机和其他硬拷贝设备上的正确值, 但对屏幕则不行。对于一个 600 dpi 的激光打印机, `LOGPIXELSX` 和 `LOGPIXELSY` 都返回 600。但对于屏幕, 无论屏幕的尺寸或分辨率如何, 两者可能都返回 96。

理解由 `NUMCOLORS`、`BITSPIXEL` 以及 `PLANES` 返回的颜色信息是需要一定技巧的, 这三个都是 `GetDeviceCaps` 的参数。对打印机或绘图仪来说, 参数 `NUMCOLORS` 通常给出设备能显示的颜色数目。对于单色打印机, `NUMCOLORS` 返回 2。

表 2-5 几个有用的 `GETDEVICECAPS` 参数

| 参数 | 返回值 |
|-------------------------|---|
| <code>HORZRES</code> | 以像点数目表示的显示平面宽度 |
| <code>VERTRES</code> | 以像点数目表示的显示平面高度 |
| <code>HORZSIZE</code> | 以毫米表示的显示平面宽度 |
| <code>VERTSIZE</code> | 以毫米表示的显示平面高度 |
| <code>LOGPIXELSX</code> | 水平方向上每逻辑英寸内像素点的数目 |
| <code>LOGPIXELSY</code> | 竖直方向上每逻辑英寸内像素点的数目 |
| <code>NUMCOLORS</code> | 如果是显示设备, 则返回静态颜色数目; 如果是打印机或绘图仪, 则返回能支持的颜色数目 |
| <code>BITSPIXEL</code> | 每个像素点的位的数目 |
| <code>PLANES</code> | 位平面的数目 |
| <code>RASTERCAPS</code> | 表述设备的某些特性的位标志, 如设备是否为调色板式、设备是否能显示位图 |
| <code>TECHNOLOGY</code> | 位标志, 确定设备类型——屏幕、打印机、绘图仪等 |

然而, 屏幕的颜色分辨率(屏幕可同时显示的颜色的数目)是由 `BITSPIXEL` 与 `PLANES` 相乘并取平方得到的, 如下所示:

```
CClientDC dc(this);
int nPlanes = dc.GetDeviceCaps(PLANES);
int nBPP = dc.GetDeviceCaps(BITSPIXEL);
int nColors = 1 << (nPlanes * nBPP);
```

如果在配备了 256 色视频适配器的微机上执行这段程序代码, 则 `nColors` 将等于 256。调用 `CetDeviceCaps` 并将参数 `NUMCOLORS` 传递给它, 则返回的不是 256 而是 20——Windows

给视频适配器调色板编程设定的“静态颜色”的数目。在第15章将详细介绍静态颜色以及屏幕和视频适配器的颜色特性。

为了使示例程序的输出与输出设备的物理属性一致,本书多次使用了 `GetDeviceCaps`。本章后面将首次使用 `GetDeviceCaps`,用屏幕的 `LOGPIXELSX` 和 `LOGPIXELSY` 参数在 `MM_TEXT` 映射模式下画一个1逻辑英寸长、1/4逻辑英寸高的矩形。

2.2 用 GDI 绘图

基本知识已经讲得很多了。到目前为止您可能会觉得我一直在答非所问。请相信,本章中所学的知识迟早会有用。现在我们先介绍几个将像素点输出到屏幕上的函数。

下面几小节介绍的函数只是可利用的 GDI 输出函数中的一部分。如果对每一个函数都详加介绍会占据比本章多得多的篇幅。读完本章,可参看 MFC 文档中 CDC 成员函数的完整列表。这样您会对 Windows GDI 覆盖范围之广有更深入的认识,并且在遇到问题时能知道到哪里寻求帮助。

2.2.1 画直线和曲线

MFC 的 CDC 类中包含了许多可用来画直线和曲线的成员函数。表 2-6 列出了一些关键函数。当然还有其他函数,但是这些函数已经能够很好地展示可供利用的画直线和画曲线函数的范围了。

表 2-6 用来画直线和曲线的 CDC 函数

| 函数 | 说 明 |
|---------------------------|--|
| <code>MoveTo</code> | 在画线前设定当前位置 |
| <code>LineTo</code> | 从当前位置画一条线到指定位置,并将当前位置移至线的终点 |
| <code>Polyline</code> | 将一系列点用线段连接起来 |
| <code>PolylineTo</code> | 从当前位置开始将一系列点用线段连接起来,并将当前位置移至折线的终点 |
| <code>Arc</code> | 画一个弧 |
| <code>ArcTo</code> | 画一个弧并将当前位置移至弧的终点 |
| <code>PolyBezier</code> | 画一条或多条贝塞尔样条曲线 |
| <code>PolyBezierTo</code> | 画一条或多条贝塞尔样条曲线,并将当前位置移至最后一段样条曲线的终点 |
| <code>PolyDraw</code> | 通过一组点画一系列线段和贝塞尔样条曲线,并将当前位置移至最后一个线段或样条曲线的终点 |

画直线是很简单的。只要将当前位置设在线的一端,调用 `LineTo` 并给出另一端点的坐标即可:

```
dc.MoveTo(0,0);
```

```
dc.LineTo(0, 100);
```

想接着前一条线再画一条线时,只需再次调用 `LineTo`。由于第一次调用 `LineTo` 时当前位置已设置在线的终点,这里就不需要第二次调用 `MoveTo` 了:

```
dc.MoveTo(0, 0);
dc.LineTo(0, 100);
dc.LineTo(100, 100);
```

运用 `Polyline` 或 `PolylineTo` 可以一次画多条线。两者间的区别在于: `PolylineTo` 使用设备描述表的当前位置,而 `Polyline` 则不需要。下面的语句绘制一个方框,该方框从描述顶边的点到另一边的距离为 100 单位:

```
POINT aPoint[5] = { 0, 0, 0, 100, 100, 100, 100, 0, 0, 0 };
ac.Polyline(aPoint, 5);
```

下面的语句运用 `PolylineTo` 画了同样的一个方框:

```
dc.MoveTo(0, 0);
POINT aPoint[4] = { 0, 100, 100, 100, 100, 0, 0, 0 };
dc.PolylineTo(aPoint, 4);
```

当 `PolylineTo` 返回时,当前位置设在最后一条线段的终点上——这里是点(0,0)。如果使用的是 `Polyline`,则当前位置不变。

Charles Petzold 的《Programming Windows》中有一个很好的例子。它展示了折线的用处以及为什么折线有用。下面的 `OnPaint` 函数基本上只是 Charles 程序的 MFC 改版。它实现的功能是用 `CDC::Polyline` 在窗口内画一条正弦曲线。

```
#include <math.h>
#define SEGMENTS 500
#define PI 3.1415926
.
.
.
void CMainWindow::OnPaint()
{
    CRect rect;
    GetClientRect(&rect);
    int nWidth = rect.Width();
    int nHeight = rect.Height();

    CPaintDC dc(this);
    CPoint aPoint[SEGMENTS];

    for (int i = 0; i < SEGMENTS; i++) {
        aPoint[i].x = (i * nWidth) / SEGMENTS;
        aPoint[i].y = (int)((nHeight / 2) *
            (1 - (sin((2 * PI * i) / SEGMENTS))));
    }
}
```

```
dc.Polyline(aPoint, SEGMENTS);  
}
```

用这段程序代码替换第1章 Hello 程序中的 OnPaint 函数,您就可以看到它的运行结果了。注意要用 CRect 函数 Width 和 Height 计算窗口内客户区的宽度和高度。

弧是从圆或椭圆周边上截取下来的一段曲线。用 CDC::Arc 画弧非常简单,只要给它椭圆的外接矩形和一对点,这对点指定了两条从椭圆中心引出的辅助线的端点坐标。辅助线与椭圆的交点就是所画弧的起点和终点。(该辅助线必须足够长,最终达到椭圆的圆周;否则得到的不是期望的结果。)下面的程序代码画了一段弧,它是 200 单位宽、100 单位高的椭圆的左上方四分之一部分。

```
CRect rect(0, 0, 200, 100);  
CPoint point1(0, -500);  
CPoint point2(-500, 0);  
dc.Arc(rect, point1, point2);
```

如果想翻转该弧,画出椭圆的右上方四分之一、右下方四分之一和左下方四分之一,则只需颠倒传送给 Arc 函数的 point1 和 point2 的次序即可。如果想知道该弧的终点(在用线和弧画三维饼图时会用到这项信息),则可改用 ArcTo 函数,并用 CDC::GetCurrentPosition 确定终点位置。这看起来很容易,但要小心。除了画弧之外, ArcTo 还会从原来的当前位置画一条线到弧的起始点。另外, ArcTo 是未在 Windows 98 中实现的少数 GDI 函数之一。如果在非 Windows NT 或 Windows 2000 操作平台上调用它,则不会有任何输出。

如果样条曲线更符合你的风格,那么 GDI 同样能发挥作用。CDC::PolyBezier 可用来画贝塞尔样条曲线——由两个终点和两个起“牵拉”作用的中间点确定的光滑曲线。贝塞尔样条曲线,或简单地说“贝塞尔”,最初是用来帮助工程师给汽车车身建立数学模型的,如今从字体到弹头设计都使用这种样条曲线。下面的程序代码段用两条贝塞尔样条曲线画了一个类似于 Nike 的有名的“swoosh”商标的图形。(参看图 2-2。)

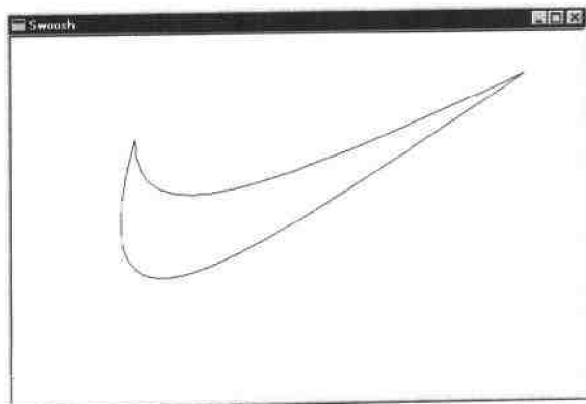


图 2-2 用贝塞尔样条曲线所画的著名鞋商标

```
POINT aPoint1[4] = { 120, 100, 120, 200, 250, 150, 500, 40 };
POINT aPoint2[4] = { 120, 100, 50, 350, 250, 200, 500, 40 };
dc.PolyBezier (aPoint1, 4);
dc.PolyBezier (aPoint2, 4);
```

这两条样条曲线是彼此独立的,只是碰巧端点相同罢了。如果要将两条或更多的样条曲线连接成为连续曲线,则应在每条附加样条曲线的 POINT 数组中再添加三个点,并相应地增大 PolyBezier 第二个参数中的点数即可。

所有 GDI 画线和画曲线函数都有一个特点:从不画最后一个像素点。如果用下面语句画一条从(0,0)到(100,100)的线:

```
dc.MoveTo (0, 0);
dc.LineTo (100, 100);
```

则(0,0)位置上的像素和(1,1),(2,2)如此等等位置上的像素一样都被设置成了该线的颜色,但(100,100)上像素的颜色保持不变。如果还要画出该线的最后一个像素,则必须自己再画一次这个点。有一种画点的方法是使用 CDC::SetPixel 函数,该函数能够将单个点的颜色设置成指定的颜色。

2.2.2 画椭圆、多边形以及其他形状

GDI 不会把您限制在简单的直线和曲线上。它还允许您画椭圆、矩形、饼状楔形物以及其他封闭图形。MFC 的 CDC 类将相关的 GDI 函数封装在类成员函数中。在设备描述表对象中或通过指向设备描述表对象的指针可以方便地调用这些函数。表 2-7 列出了其中的几个函数。

表 2-7 用来画封闭图形的 CDC 函数

| 函数 | 说 明 |
|-----------|---------------------|
| Chord | 画一个由椭圆和直线相交后围成的封闭图形 |
| Ellipse | 画一个圆或椭圆 |
| Pie | 画一个饼状的楔形物 |
| Polygon | 连接一组点形成一个多边形 |
| Rectangle | 画一个带直角的矩形 |
| RoundRect | 画一个带圆角的矩形 |

画封闭图形的 GDI 函数以外接方框的坐标值作为参数。例如,在用 Ellipse 函数画圆时,不要求指定中心点和半径,但是要求指定该圆的外接方框。如下所示,可以显式地传送坐标值:

```
dc.Ellipse(0, 0, 100, 100);
```

或通过 RECT 结构或 CRect 对象来传送,如下所示:

```
CRect rect(0, 0, 100, 100);
dc.Ellipse(rect);
```

这个圆在绘制时,在外接方框左侧与直线 $x=0$ 相接,在上方与直线 $y=0$ 相接,但在右侧和下方分离直线 $x=100$ 和 $y=100$ 还有一个像素。也就是说,图形从外接方框的左、上边界画到右、下边界(但不包括右、下边界)。如果按照下面的方式调用 CDC::Rectangle 函数:

```
dc.Rectangle(0, 0, 8, 4);
```

则得到的输出见图 2-3。可以看出矩形的右、下边界落在 $x=7$ 和 $y=3$ 而不是 $x=8$ 和 $y=4$ 。

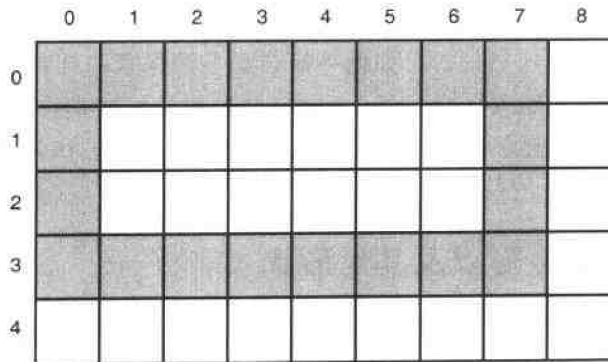


图 2-3 语句 dc.Rectangle(0, 0, 8, 4) 画出的矩形

Rectangle 和 Ellipse 的使用非常简单。只要给出外接方框,画图就全交给 Windows 了。如果要画一个带圆角的矩形,则应使用 RoundRect 而不用 Rectangle。

Pie 和 Chord 函数需要好好研究。这两个函数在语法上与前一部分介绍的 Arc 函数是一样的。区别表现在输出上(参见图 2-4)。Pie 画线将弧的端点与椭圆中心用直线相连,由此围成一个封闭图形。Chord 则直接连接弧的两个端点形成封闭图形。下面的 OnPaint 处理程序用 Pie 画了一个饼图,用它来描述四个季度的税收情况。

```
#include <math.h>
#define PI 3.1415926
.
.
.
void CMainWindow::OnPaint()
{
    CPaintDC dc(this);
    int nRevenues[4] = { 125, 376, 252, 184 };
}
```



```

CRect rect;
GetClientRect(&rect);
dc.SetViewportOrg(rect.Width()/2, rect.Height()/2);

int nTotal = 0;
for(int i=0; i<4; i++)
    nTotal += nRevenues[i];

int x1 = 0;
int y1 = -1000;
int nSum = 0;

for(i=0; i<4; i++) {
    nSum += nRevenues[i];
    double rad = ((double)(nSum * 2 * PI) / (double)nTotal) + PI;
    int x2 = (int)(sin(rad) * 1000);
    int y2 = (int)(cos(rad) * 1000 * 3) / 4;
    dc.Pie(-200, -150, 200, 150, x1, y1, x2, y2);
    x1 = x2;
    y1 = y2;
}
}

```

注意在画图之前要用 SetViewportOrg 将原点移至窗口的中心,这样所画的图也就在窗口的中间了。

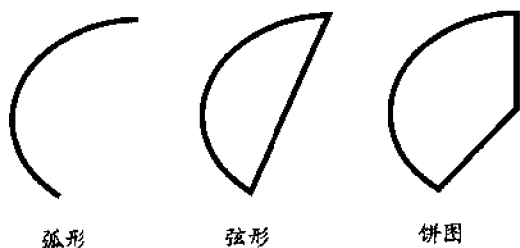


图 2-4 Arc、Chord 和 Pie 函数的输出结果

2.2.3 GDI 画笔和 CPen 类

Windows 用当前选入设备描述表的画笔绘制直线和曲线,并给用 Rectangle、Ellipse 以及其他图形生成函数画出的图形镶画边框。默认画笔画出的是一个像素点宽的黑色实线。如果要改变画线方式,则需创建一个 GDI 画笔,并由 CDC::SelectObject 将它选入设备描述表。

MFC 用类 CPen 表示 GDI 画笔。创建画笔的最简单的方法是构造一个 CPen 对象并把定义画笔所用的参数都传送给该对象。

```
CPen pen(PS_SOLID,1,RGB(255,0,0));
```

创建 GDI 画笔的第二种方法是构造一个没有初始化的 CPen 对象并调用 CPen::CreatePen:

```
CPen pen;
pen.CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
```

还有一种方法是构造一个没有初始化的 CPen 对象,向 LOGPEN 结构中填充描述画笔特性的参数,然后调用 CPen::CreatePenIndirect 生成画笔:

```
CPen pen;
LOGPEN lp;
lp.lopnStyle = PS_SOLID;
lp.lopnWidth.x = 1;
lp.lopnColor = RGB(255, 0, 0);
pen.CreatePenIndirect(&lp);
```

LOGPEN 的 lopnWidth 字段是一个 POINT 数据结构。结构中的 x 数据成员指定了画笔的宽度,y 数据成员没有用到。

如果画笔创建成功,则 CreatePen 和 CreatePenIndirect 返回 TRUE,反之,则返回 FALSE。如果是由 CPen 的构造函数创建画笔,则一旦不能创建画笔,就会有 CResourceException 型异常事件发生。这种情况只有在 Windows 内存严重缺乏时才会发生。

定义画笔需要三个特性:样式、宽度和颜色。上面给出的例子创建了一个样式为 PS_SOLID,宽度为 1 以及颜色为亮红的画笔。第一个参数传送给 CPen::CPen 和 CPen::CreatePen,指定笔的样式,即线的类型。PS_SOLID 创建的画笔能画出连续的实线。其他画笔样式见图 2-5。

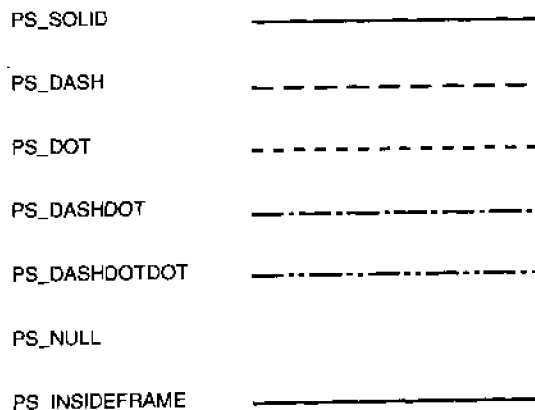


图 2-5 画笔样式

PS_INSIDEFRAME 是一种特殊的样式,所画的实线位于正在形成的图形的边框内。例

如,如果采用其他任何一种笔样式来画一个用 20 个单位宽的 PS_SOLID 笔绘制的直径为 100 个单位的圆,则从圆周外缘测得的真实直径是 120 个单位,请参见图 2-6。为什么会这样呢?实际上,这个笔所画的边框在理论圆的两侧都向外延伸了 10 个单位。用 PS_INSIDEFRAME 画同一个圆,则得到的圆的直径恰好是 100 个单位。PS_INSIDEFRAME 样式不影响由 LineTo 画得的直线,同时也不影响其他不使用外接矩形的函数所画的图形。

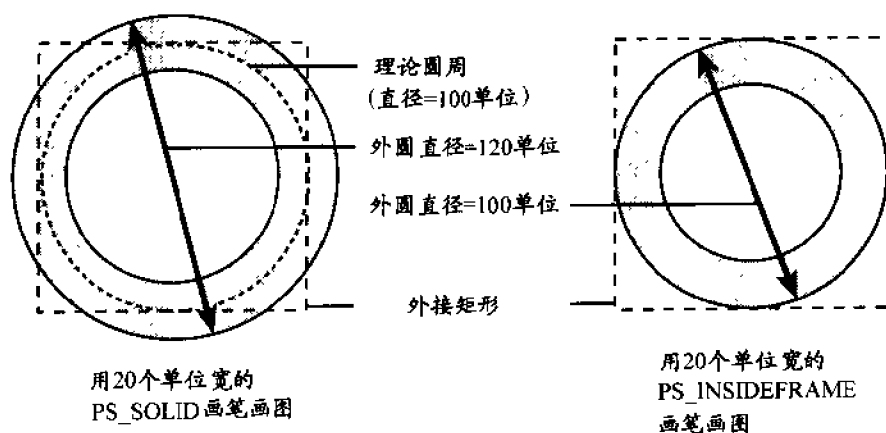


图 2-6 PS_INSIDEFRAME 笔样式

PS_NULL 样式创建的笔被 Windows 程序员称为“NULL 笔”。为什么要创建一个 NULL 笔呢?信不信由您,有时 NULL 笔会很有用。例如:假如您想画一个没有边框的红色实心圆。如果用 MFC 的 CDC::Ellipse 函数画这个圆,那么 Windows 会自动用当前选入设备描述表的笔给圆加边框。您无法告诉 Ellipse 函数不要加边框,但是可以将 NULL 笔选入设备描述表,这样该圆就没有可见的边框了。NULL 画刷的用法与此类似。如果您想保留这个边框,并且希望圆的内部是透明的,那么可在画圆之前将 NULL 画刷选入设备描述表。

传递给 CPen 的创建笔函数的第二个参数指定了笔所画线的宽度。笔宽度是以逻辑单位给出的,而逻辑单位的实际意义又取决于当前的映射模式。可以创建具有任一逻辑宽度的 PS_SOLID、PS_NULL 和 PS_INSIDEFRAME 笔,但 PS_DASH、PS_DOT、PS_DASHDOT 和 PS_DASHDOTDOT 笔则必须是 1 个逻辑单位宽。无论映射模式如何,在任一样式下把笔宽度指定为 0,都会产生宽为 1 个像素点的笔。

创建笔的第三个,也是最后一个要指定的参数是笔的颜色。Windows 使用 24 位 RGB 颜色模型,其中每一种颜色是由分别在 0 到 255 间变化的红、绿以及蓝色的值确定的。值越高,相应的颜色成分也越亮。RGB 宏将代表三个独立颜色成份的值合成为一个可传递给 GDI 的 COLORREF 值。语句

```
CPen pen (PS_SOLID, 1, RGB (255, 0, 0));
```

创建一个亮红色的笔,而语句

```
CPen pen (PS_SOLID, 1, RGB (255, 255, 0));
```

将红和绿合成起来,创建了一个亮黄色的笔。如果显示卡不支持 24 位颜色,则 Windows 能做到的是不断抖动那些无法直接显示的颜色。但是请注意,只有宽度大于 1 个逻辑单位的 PS_INSIDEFRAME 笔可使用抖动色方式。对于其他笔样式,Windows 会将笔颜色映射成可显示的最相近的原色。如果始终采用表 2-8 中的“主要”颜色,可以肯定在各种适配器上您都能得到想要的颜色。这些颜色是 Windows 在各种视频适配器颜色寄存器中编程设定的基本调色板中的一部分。这个调色板保证了一个公用颜色集合,供所有程序使用。

表 2-8 主要的 GDI 颜色

| 颜色 | R | G | B | 颜色 | R | G | B |
|----|-----|-----|-----|-----|-----|-----|-----|
| 黑 | 0 | 0 | 0 | 浅灰 | 192 | 192 | 192 |
| 蓝 | 0 | 0 | 192 | 亮蓝 | 0 | 0 | 255 |
| 绿 | 0 | 192 | 0 | 亮绿 | 0 | 255 | 0 |
| 青 | 0 | 192 | 192 | 亮青 | 0 | 255 | 255 |
| 红 | 192 | 0 | 0 | 亮红 | 255 | 0 | 0 |
| 品红 | 192 | 0 | 192 | 亮品红 | 255 | 0 | 255 |
| 黄 | 192 | 192 | 0 | 亮黄 | 255 | 255 | 0 |
| 深灰 | 128 | 128 | 128 | 白 | 255 | 255 | 255 |

怎样使用创建好的笔呢?很简单。只要将它选入设备描述表即可。下面的程序代码段创建了一个 10 单位宽的红色笔,并用它画了一个椭圆。

```
CPen pen (PS_SOLID, 10, RGB (255, 0, 0));
CPen * pOldPen = dc.SelectObject (&pen);
dc.Ellipse (0, 0, 100, 100);
```

椭圆被当前画笔的颜色或图案填充。画笔的默认值为白色。如果要改变默认值,则需创建一个 GDI 画笔并在调用 Ellipse 之前将它选入设备描述表。马上我会示范给您看。

扩展笔

如果基本笔样式不符合要求,那么还可以使用被称为“扩展”的另一类笔。Windows GDI 和 MFC 的 CPen 类支持这类笔。这些笔提供了更多的输出方式。例如:创建一种扩展笔,使它可以画由位图图像描述的图案或使用抖动色。通过指定端点样式(展平的、圆的、方的)和连接样式(斜面连接、斜角连接或圆角连接)可以精确控制端点和连接点。下面的程序代码创建的是 16 个单位宽的扩展笔,可画出带展平端点的绿色直线。两线相交处,连接端生成圆角,形成平滑过渡。

```
LOGBRUSH lb;
lb.lbStyle = BS_SOLID;
lb.lbColor = RGB(0, 255, 0);
CPen pen (PS_GEOMETRIC|PS_SOLID|PS_ENDCAP_FLAT|
          PS_JOIN_ROUND, 16, &lb);
```

Windows 对使用扩展笔有几项限制。除非图形首先作为一个“通路”画出并由 `CDC::StrokePath` 或一个相关函数生成,否则端点连接就无法进行。您可以通过在调用 `CDC::BeginPath` 和 `CDC::EndPath` 之间插入绘图命令来定义一条通路,如下所示:

```
dc.BeginPath();           // Begin the path definition
dc.MoveTo(0, 0);          // Create a triangular path
dc.LineTo(100, 200);
dc.LineTo(200, 100);
dc.CloseFigure();
dc.EndPath();             // End the path definition
dc.StrokePath();          // Draw the triangle
```

通路是 GDI 一个强有力的性能,用它它可以生成各种有趣的效果。第 15 章将详细介绍通路以及使用通路的 `CDC` 函数。

2.2.4 GDI 画刷和 CBrush 类

在默认情况下,由 `Rectangle`、`Ellipse` 以及其他 `CDC` 输出函数画出的封闭图形填充着白色像素点。通过创建 GDI 画刷并在画图之前将它选入设备描述表可以改变图形的填充颜色。

MFC 的 `CBrush` 类封装了 GDI 画刷。画刷有三种基本类型:单色、带阴影线和带图案。单色画刷填充的是单色。如果显示硬件不支持直接显示单色画刷的颜色,则 Windows 用可显示的抖动色模仿该颜色。阴影线画刷采用预先定义好的交叉线图案填充图形。这种图案共有六种,类似于机械和建筑用图中常见的那些阴影线。图案画刷用位图来填充图形。`CBrush` 类为每种画刷样式提供了一个构造函数。

只要将 `COLORREF` 的值传递给 `CBrush` 构造函数,您就可以只用一步创建一个单色画刷了:

```
CBrush brush (RGB(255, 0, 0));
```

或者,也可以创建一个没有初始化的 `CBrush` 对象,然后调用 `CBrush::CreateSolidBrush`,用两步创建一个单色画刷:

```
CBrush brush;
brush.CreateSolidBrush (RGB(255, 0, 0));
```

这两个例子创建的都是颜色为亮红的单色画刷。您还可以通过将 `LOGBRUSH` 初始化并调用 `CBrush::CreateBrushIndirect` 来创建画刷。和 `CPen` 构造函数一样,如果 GDI 内存不足,

所有用来创建画刷的 CBrush 构造函数都会产生资源异常问题,因而也就不能创建画刷了。

将阴影线索引和 COLORREF 值传送给 CBrush 的构造函数或调用 CBrush::CreateHatchBrush 都能创建一个阴影线画刷。语句

```
CBrush brush (HS_DIAGCROSS, RGB (255, 0, 0));
```

创建的阴影线画刷与下面语句创建的相同,其阴影线是由倾斜 45 度的垂直相交线构成的:

```
CBrush brush;  
brush.CreateHatchBrush (HS_DIAGCROSS, RGB (255, 0, 0));
```

HS_DIAGCROSS 是可供选择的六种阴影线样式之一(参见图 2-7)。在用阴影线画刷填充时,除非用 CDC::SetBkColor 改变设备描述表的当前背景色,或用 CDC::SetBkMode 把背景模式 OPAQUE 改成 TRANSPARENT,禁止背景填充,否则 Windows 就以默认的背景色(白色)填充阴影线间的空白处。语句:

```
CBrush brush (HS_DIAGCROSS, RGB (255, 255, 255));  
dc.SelectObject (&brush);  
dc.SetBkColor (RGB (192, 192, 192));  
dc.Rectangle (0, 0, 100, 100);
```

画了一个 100 单位 × 100 单位的正方形,并在浅灰色背景下填充了白色的交叉线。语句

```
CBrush brush (HS_DIAGCROSS, RGB (0, 0, 0));  
dc.SelectObject (&brush);  
dc.SetBkMode (TRANSPARENT);  
dc.Rectangle (0, 0, 100, 100);
```

在当前的背景色下画了一个填充着黑色交叉线的矩形。

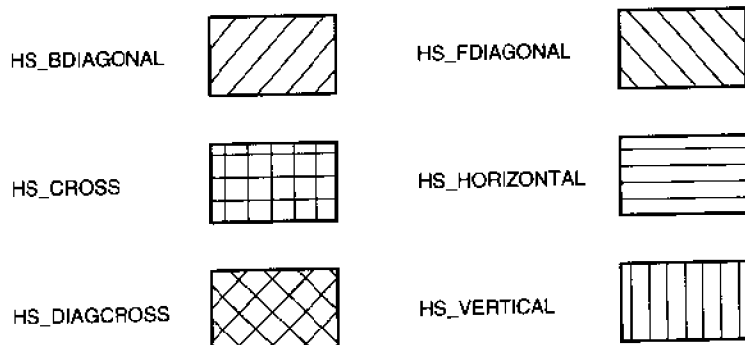


图 2-7 阴影线画刷

画刷原点

在使用抖动色或阴影线画刷时,应该注意设备描述表的属性之一:画刷原点。Windows在用阴影线或抖动色填充某个区域时,在水平和竖直方向上是按8个像素×8个像素的样式填充的。在默认方式下,该图案的原点,也就是“画刷原点”,是设备点(0,0)——窗口左上角上的屏幕像素点。这意味着一个原点在左上角,宽、高各为100个像素的矩形内的图案,与一个位置稍向左或右移动几个像素点的矩形内的图案相比,前者与矩形边框的符合程度要差一些。请参看图2-8。在多数应用场合中,这不会有什么影响。用户不可能注意到画刷对齐中小小的不同。然而在某些情况下,这却是个大问题。

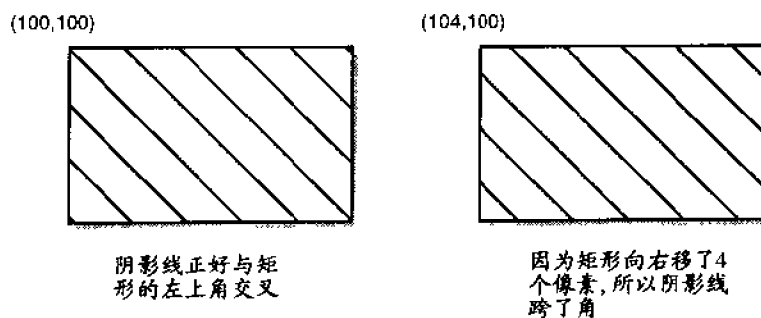


图 2-8 画刷对齐

假定您用阴影线画刷填充一个矩形,并且反复清除和重画向右或向左平移一个像素点后的矩形,以最终形成这个矩形的动态移动过程。如果在每次重画之前不把画刷原点重置在相对于矩形不动的定点上,则在矩形移动的同时阴影线图案也要“走”了。

如何解决这个问题呢?在将画刷选入设备描述表和画矩形之前,先在画刷对象中调用 `CGdiObject::UnrealizeObject`,允许画刷原点移动。然后调用 `CDC::SetBrushOrg` 将画刷原点和矩形的左上角对齐,如下所示:

```
CPoint point (x1, y1);
dc.LPtoDP (&point);
point.x %= 8;
point.y %= 8;
brush.UnrealizeObject();
dc.SetBrushOrg (point);
dc.SelectObject (&brush);
dc.Rectangle (x1, y1, x2, y2);
```

在这个例子中,point 是一个 `CPoint` 对象,它保存矩形左上角的逻辑坐标值。要将逻辑坐标值转换成设备坐标值(画刷原点通常是以设备坐标值给出的),需调用 `LPtoDP`。由于传递给 `SetBrushOrg` 的坐标值应在 0~7 范围内,所以还要执行模 8 运算。这样无论矩形画在窗

口的哪个部位,阴影线都能始终与边框保持一致。

2.2.5 画文本

您已经知道一种向窗口输出文本的方式了。CDC::DrawText 函数往显示平面写一串文本。在指定格式化矩形和一系列指示矩形中文本位置的选项标志后,DrawText 就知道往哪儿画它的输出了。第1章 Hello 程序中,语句

```
dc.DrawText(_T("Hello, MFC"), -1, &rect,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
```

将“Hello,MFC”画在窗口的中部。rect 是用窗口客户区坐标值初始化的矩形对象;而 DT_CENTER 和 DT_VCENTER 标志告诉 DrawText 将输出放在矩形的中心位置。

DrawText 是 MFC CDC 类成员中与文本有关的几个函数之一。其他的几个函数列在表 2-9 中。其中,最有用的一个是 TextOut。与 DrawText 一样,它能输出文本,但要接受一对指定文本输出位置的 x-y 坐标值,如果有要求它也可以使用当前位置。语句

```
dc.TextOut(0, 0, CString(_T("Hello, MFC")));
```

将“Hello,MFC”写到由 dc 代表的显示平面的左上角。与 TextOut 工作方式相似的还有一个名为 TabbedTextOut 的函数,它们之间唯一的区别在于,后者将制表符还原成白色的空格。(如果传递给 TextOut 的字符串包含制表符,则它们在大部分字体中都以矩形形状出现。)制表符的位置在调用 TabbedTextOut 时被指定。一个名为 ExtTextOut 的相关函数提供了另一种在文本周围给矩形填充背景色的方法。它使编程人员能够准确控制字符间的间隔。

在默认方式下,传递给 TextOut、TabbedTextOut 和 ExtTextOut 的坐标值都确定了文本中最左侧字符的左上角位置。但是传递给 TextOut 的坐标值与输出字符串中字符的关系,即所谓的文本对齐方式,是设备描述表的一个属性。用户可调用 CDC::SetTextAlign 改变这个属性。例如在

```
dc.SetTextAlign(TA_RIGHT);
```

语句执行之后,传递给 TextOut 的 x 坐标值定义的就是字符最右端的位置——非常适合显示右对齐文本。

您还可以用 TA_UPDATECP 标志调用 SetTextAlign,命令 TextOut 忽略传递给它的 x 和 y 参数,而改用设备描述表当前的位置。当文本对齐包含 TA_UPDATECP 时,每输出一个字符串,TextOut 就更新一次当前位置的 x 值。这个特性的用处之一是调节在同一行上输出的两个或两个以上字符串间的距离。

表 2-9 CDC 文本函数

| 函数 | 说 明 |
|----------------------|-----------------------------|
| DrawText | 在格式化矩形中显示文本 |
| TextOut | 在当前或指定位置输出一行文本 |
| TabbedTextOut | 输出一行含有制表符的文本 |
| ExtTextOut | 输出一行文本,并有选择地给矩形填充背景色或改变字符间距 |
| GetTextExtent | 计算当前字体下一个字符串的宽度 |
| GetTabbedTextExtent | 计算当前字体下含有制表符字符串的宽度 |
| GetTextMetrics | 返回当前字体的字体度量(字符高度、字符平均宽度等) |
| SetTextAlign | 给 TextOut 和其他输出函数设定对齐参数 |
| SetTextJustification | 确定对齐一串文本时需要增加的宽度 |
| SetTextColor | 设定设备描述表的文本输出颜色 |
| SetBkColor | 设定设备描述表的背景色,即确定输出字符背后的填充色 |

借助 GetTextMetrics 和 GetTextExtent 这两个函数可以获得当前选入设备描述表的字体信息。GetTextMetrics 传给 TEXTMETRIC 结构关于形成某种字体的字符的信息。GetTextExtent 返回指定字符串在该字体下生成的宽度,以逻辑单位表示。(如果字符串含有制表符,则调用 GetTabbedTextExtent) GetTextExtent 的一个作用是:在输出字符串之前,测量字符串的宽度。这样可以计算出对齐文本时单词间允许的间距。如果 nWidth 是指左、右页边间的距离,则下面的程序代码输出“Now is the time”,并使输出与左、右边距对齐。

```
CString string = _T("Now is the time");
CSize size = dc.GetTextExtent(string);
dc.SetTextJustification(nWidth - size.cx, 3);
dc.TextOut(0, y, string);
```

传递给 SetTextJustification 的第二个参数指定了字符串中断开符的数目。默认的断开符是空格符。SetTextJustification 被调用之后,对 TextOut 和有关文本输出函数的调用则会将由 SetTextJustification 第一个参数定义的空白均匀分配在所有断开符之间。

2.2.6 GDI 字体和 CFont 类

所有的 CDC 文本函数都使用当前选入设备描述表的字体。字体是指一组具有特定尺寸(高度)和字样的字符;字样是指字符共有的属性,如字的粗细度——正常体或黑体。在传统印刷式样中,字体尺寸大小是以 point(即点)为单位来度量的。一个点相当于 1/72 英寸。12-点字体中的字符就有 1/6 英寸高。但是在 Windows 中,针对输出设备的具体特性,字符的实际高度也稍有不同。typeface 项描述了字体的基本样式。Times New Roman 是一种字样;Courier New 是另一种。

与画笔或画刷一样,字体也是一个 GDI 对象。在 MFC 中,字体由 CFont 类的对象表示。

构造了 CFont 对象之后,就可以通过调用 CFont 对象的 CreateFont、CreateFontIndirect、CreatePointFont 或 CreatePointFontIndirect 函数创建带下划线的 GDI 字体了。如果想以像素为单位指定字体尺寸,则调用 CreateFont 或 CreateFontIndirect;如果想以点为单位指定字体尺寸,则调用 CreatePointFont 和 CreatePointFontIndirect。用 CreatePointFont 函数创建 12-点屏幕字体只需要两行程序代码:

```
CFont font;
font.CreatePointFont(120,_T("Times New Roman"));
```

而使用 CreateFont 函数则需要向设备描述表查询垂直方向上每英寸内像素的逻辑个数,并把点转换为像素:

```
CClientDC dc(this);
int nHeight = -((dc.GetDeviceCaps(LOGPIXELSY) * 12) / 72);
CFont font;
font.CreateFont(nHeight, 0, 0, 0, FW_NORMAL, 0, 0, 0,
    DEFAULT_CHARSET, OUT_CHARACTER_PRECIS, CLIP_CHARACTER_PRECIS,
    DEFAULT_QUALITY, DEFAULT_PITCH|FF_DONTCARE,
    _T("Times New Roman"));
```

顺便提一下,传递给 CreatePointFont 的数值是你所期望的点的大小的 10 倍。这样字体尺寸就可以控制到点的 1/10,考虑到大部分屏幕和其他常见输出设备相对较低的分辨率,对于大多数应用场合,这个精度已经足够用了。

传递给 CreateFont 的诸多参数之中,有确定字的粗细度和字符是否倾斜的参数。您不能用 CreatePointFont 创建一个粗黑、倾斜的字体,但用 CreatePointFontIndirect 就能达到目的了。下面的程序代码用 CreatePointFontIndirect 创建了一个 12-点、粗黑、倾斜的 Times New Roman 字体:

```
LOGFONT lf;
::ZeroMemory(&lf, sizeof(lf));
lf.lfHeight = 120;
lf.lfWeight = FW_BOLD;
lf.lfItalic = TRUE;
::lstrcpy(lf.lfFaceName,_T("Times New Roman"));

CFont font;
font.CreatePointFontIndirect(&lf);
```

LOGFONT 是一个结构,它的字段定义了字体的所有特性。::ZeroMemory 是一个 API 函数,它将一块内存清零。::lstrcpy 也是一个 API 函数,它将一个文本字符串从一个内存位置复制到另一个位置。您也可以用运行时的 C 函数 memset 和 strcpy 代替(实际上,应该调用 _tstrcpy 而不是 strcpy,这样就能处理 ANSI 或 Unicode 字符了),但是经常调用 Windows API 函

数减少了静态链接的程序代码,从而减小了可执行程序的大小。

创建字体之后,把它选入设备描述表并调用 DrawText、TextOut 和其他 CDC 文本函数,用这种字体绘图。下面的 OnPaint 处理程序在窗口的中部画了“Hello, MFC”。但是这次文本是用 72-点 Arial 字样画的,并带有阴影。(参见图 2-9。)

```
void CMainWindow::OnPaint ()
{
    CRect rect;
    GetClientRect (&rect);

    CFont font;
    font.CreatePointFont (72, _T ("Arial"));

    CPaintDC dc (this);
    dc.SelectObject (&font);
    dc.SetBkMode (TRANSPARENT);

    CString string = _T ("Hello, MFC");

    rect.OffsetRect (16, 16);
    dc.SetTextColor (RGB (192, 192, 192));
    dc.DrawText (string, &rect, DT_SINGLELINE |
        DT_CENTER | DT_VCENTER);

    rect.OffsetRect (-16, -16);
    dc.SetTextColor (RGB (0, 0, 0));
    dc.DrawText (string, &rect, DT_SINGLELINE |
        DT_CENTER | DT_VCENTER);
}
```

阴影效果是通过重画两次文本字符串实现的——一次画在从窗口中心向右、向下偏移几个像素点的位置,一次画在窗口的中心位置。MFC 的 CRect::OffsetRect 函数用这种将矩形不断地在 x 和 y 方向上偏移一段指定距离的方法,很容易地“移动”了矩形。

如果系统没有装 Times New Roman 字样,但您又要创建 Times New Roman 字体,这时会发生些什么呢? GDI 会选择系统中相近的字样,而不让这次调用失败。通过调用内部的字体映射算法, GDI 选择出与原字样最相近的一个,因此结果往往不是用户所期望的。但至少您的应用程序不会出现这种怪现象:在一个系统上可以正常输出文本,而在另一个系统上则不可思议地没有任何输出。



图 2-9 用 72-点、Arial 字体并带下拉阴影生成的“Hello, MFC”

2.2.7 光栅字体与 TrueType 字体

绝大部分 GDI 字体可分成两类：光栅字体和 TrueType 字体。光栅字体是按位图形式存储的，在以原尺寸显示时效果最好。Windows 提供了一种很有用的光栅字体，MS Sans Serif。这种字体(8-点大小)广泛用在按钮、单选按钮和其他对话框控件中。通过复制行和列上的像素，Windows 能按比例缩放光栅字体。但是由于有阶梯效果，显示出来的字往往不好看。

最好用的字体是 TrueType 字体。这种字体能按比例缩放为任意尺寸。同 PostScript 字体一样，TrueType 字体将字符轮廓存成数学公式，并包含 GDI TrueType 字体光栅化程序所需的“暗示”信息，用来提高字体的可伸缩性。无论您使用的是什么系统，它都装有下面四种 TrueType 字体，因为这四种字体都是由 Windows 提供的：

- Times New Roman
- Arial
- Courier New
- Symbol

在第 7 章，您将学到如何向系统查询字体信息以及如何列举安装的字体。在应用程序需要准确的字符输出时，或给用户显示安装字体列表时，这些信息就会有用处了。

2.2.8 旋转文本

关于 GDI 文本输出，常有一个问题是“怎样显示旋转了的文本呢？”有两种方法可以做到这一点，其中一种只能在 Microsoft Windows NT 和 Windows 2000 系统下生效，另一种方法则与

任一 32 位版本的 Windows 兼容。在这里介绍后一种方法。

秘诀是：调用 `CFont::CreateFontIndirect` 或 `CFont::CreatePointFontIndirect` 创建一种字体并在 `LOGFONT` 结构的 `lfEscapement` 和 `lfOrientation` 字段指定与期望的旋转角度(用度表示)成 10 倍的数值。然后,按正常方法输出,例如调用 `CDC::TextOut`。正常文本有值为 0 的取向,即不倾斜,画在水平方向上。如果将值设为 450,则文本逆时针旋转 45 度。下面的 `OnPaint` 处理程序以 15° 为增量单位不断增大 `lfEscapement` 和 `lfOrientation`,并用生成的字体画出放射状的文本阵列,参见图 2-10。

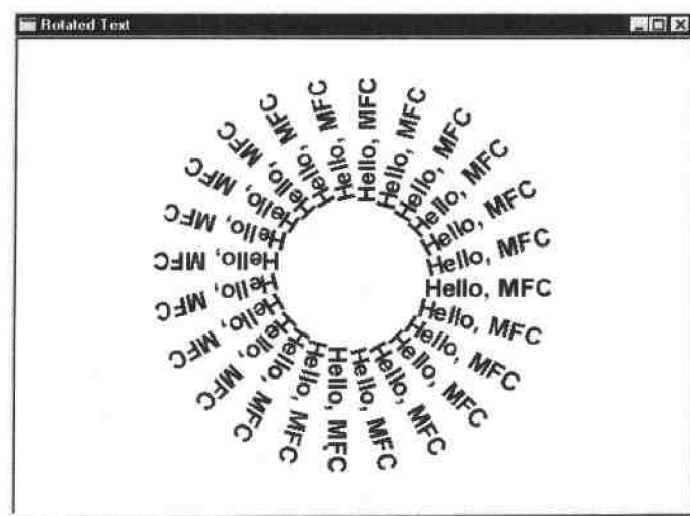


图 2-10 旋转文本

```
void CMainWindow::OnPaint ()
{
    CRect rect;
    GetClientRect (&rect);

    CPaintDC dc (this);
    dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);
    dc.SetBkMode (TRANSPARENT);

    for (int i = 0; i < 3600; i += 150) {
        LOGFONT lf;
        ::ZeroMemory (&lf, sizeof (lf));
        lf.lfHeight = 160;
        lf.lfWeight = FW_BOLD;
        lf.lfEscapement = i;
        lf.lfOrientation = i;
        ::lstrcpy (lf.lfFaceName, _T ("Arial"));
    }
}
```

```

    CFont font;
    font.CreatePointFontIndirect (&lf);

    CFont * pOldFont = dc.SelectObject (&font);
    dc.TextOut (0, 0, CString (_T ("Hello, MFC")));
    dc.SelectObject (pOldFont);
}

```

这种方法用于 TrueType 字体效果显著,但对光栅字体则完全不起作用。

2.2.9 备用对象

Windows 预先定义了一些画笔、画刷、字体以及其他一些 GDI 对象,它们在使用时不需要再显式地创建了。用户可用 `CDC::SelectStockObject` 将这些被称为备用对象的 GDI 对象选入设备描述表,或用 `CGdiObject::CreateStockObject` 将它们赋给已有的 `CPen`、`CBrush` 或其他对象。`CGdiObject` 是表示 GDI 对象的 `CPen`、`CBrush`、`CFont` 以及其他 MFC 类的一个基本类。

表 2-10 列出了一部分可用的备用对象。备用画笔有 `WHITE_PEN`、`BLACK_PEN` 和 `NULL_PEN`。`WHITE_PEN` 和 `BLACK_PEN` 画出一个像素宽的实线;而 `NULL_PEN` 不画任何东西。备用画刷包括一支白色画刷,一支黑色画刷和三种带灰色阴影的画刷。`HOLLOW_BRUSH` 和 `NULL_BRUSH` 是指称同一事物的两种方法——一支不画任何东西的画刷。`SYSTEM_FONT` 是默认情况下选入每个设备描述表的字体。

表 2-10 常用备用对象

| 对象 | 说 明 |
|--------------------------------|---|
| <code>NULL_PEN</code> | 不画任何东西的画刷 |
| <code>BLACK_PEN</code> | 画一个像素宽实线的黑笔 |
| <code>WHITE_PEN</code> | 画一个像素宽实线的白笔 |
| <code>NULL_BRUSH</code> | 不画任何东西的画刷 |
| <code>HOLLOW_BRUSH</code> | 不画任何东西的画刷(与 <code>NULL_BRUSH</code> 相同) |
| <code>BLACK_BRUSH</code> | 黑色画刷 |
| <code>DKGRAY_BRUSH</code> | 深灰色画刷 |
| <code>GRAY_BRUSH</code> | 中性灰色画刷 |
| <code>LTGRAY_BRUSH</code> | 浅灰色画刷 |
| <code>WHITE_BRUSH</code> | 白色画刷 |
| <code>ANSI_FIXED_FONT</code> | 固定间距的 ANSI 字体 |
| <code>ANSI_VAR_FONT</code> | 变间距的 ANSI 字体 |
| <code>SYSTEM_FONT</code> | 变间距系统字体 |
| <code>SYSTEM_FIXED_FONT</code> | 固定间距系统字体 |

假定现在要画一个没有边框的浅灰色的圆。该怎么办呢?这里有一种方法:

```

CPen pen (PS_NULL, 0, (RGB (0, 0, 0)));
dc.SelectObject (&pen);
CBrush brush (RGB (192, 192, 192));
dc.SelectObject (&brush);
dc.Ellipse (0, 0, 100, 100);

```

但是由于 NULL 画笔和浅灰色画刷是备用对象,所以更好的方法是:

```

dc.SelectStockObject (NULL_PEN);
dc.SelectStockObject (LTGRAY_BRUSH);
dc.Ellipse (0, 0, 100, 100);

```

下面的程序代码给出了第三种画这个圆的方法。这次备用对象被赋给 CPen 和 CBrush,而不是直接选入设备描述表:

```

CPen pen;
pen.CreateStockObject (NULL_PEN);
dc.SelectObject (&pen);

CBrush brush;
brush.CreateStockObject (LTGRAY_BRUSH);
dc.SelectObject (&brush);

dc.Ellipse (0, 0, 100, 100);

```

用哪种方法还要看自己的具体情况。第二种方法是最简短的。由于它没有创建任何 GDI 对象,所以选用这种方法绝对不会出现异常。

2.2.10 删除 GDI 对象

由 CGdiObject 派生类创建的画笔、画刷和其他对象都要占用内存资源,因此在使用完毕之后一定要删除它们。如果在栈上创建 CPen、CBrush、CFont 或其他 CGdiObject,那么在 CGdiObject 超出范围时,相关的 GDI 对象就会自动被删除。如果用 new 在堆上创建了一个 CGdiObject,则在特定时刻一定要删除它,以便调用它的析构函数。与 CGdiObject 相关联的 GDI 对象可以通过调用 CGdiObject::DeleteObject 被显式地删除。如果是备用对象,即便是由 CreateStockObject 创建的备用对象,也没必要专门去删除它。

在 16 位 Windows 中,GDI 对象经常引发资源泄漏问题。这时候因为某些程序不能删除它们创建的 GDI 对象,因此由 Program Manager 报告的可用系统资源(Free System Resources)的数量会随着应用程序的不断打开和终止渐渐减少。所有 32 位 Windows 跟踪程序所分配的资源,并在程序结束的时候删除它们。然而,在用不着 GDI 对象时删除这些对象依旧是很重要的。这样,在应用程序运行时 GDI 才不至于超出内存。试想如果每次调用 OnPaint 处理程序时都要创建 10 个画笔和画刷,但又没能删除它们,会产生什么问题呢?经过一段时间,

OnPaint 可能会创建成千上万的 GDI 对象,占据着 Windows GDI 的系统内存空间。很快,创建画笔和画刷的调用会失败,应用程序的 OnPaint 处理程序也会莫名其妙地停止工作。

Visual C++ 有一种简单的方法用来确定是否成功地删除了画笔、画刷和其他资源:只要在调试状态下运行应用程序的调试版本即可。在应用程序终结时,没有释放的资源会显示在调试窗口中。MFC 跟踪 CPen、CBrush 和其他 CObject 派生类的内存分配情况,并在对象没有删除时通知您。如果从调试信息还不能确定哪些对象没有删除,则可以在包含 Afxwin.h 的语句之后,添加一条语句到应用程序源代码中:

```
#define new DEBUG_NEW
```

(AppWizard 生成的应用程序自动包含了这条语句。)在未释放对象的调试信息中有文件名和行号可以帮助您准确地找到泄漏位置。

2.2.11 取消对 GDI 对象的选定

删除由用户创建的 GDI 对象是重要的,而不能删除已经选入设备描述表的 GDI 对象也同样不容忽视。试图用已删除的对象画图的程序代码是错误的。它没有崩溃的唯一原因在于:Windows GDI 中各处都有错误检查程序代码防止这种崩溃的发生。

遵守这条规则并不像听上去那么容易。下面的 OnPaint 处理程序允许删除已经选入设备描述表的画刷。知道为什么吗?

```
void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);
    CBrush brush (RGB (255, 0, 0));
    dc.SelectObject (&brush);
    dc.Ellipse (0, 0, 200, 100);
}
```

问题是这样的。CPaintDC 对象和 CBrush 对象是在栈上创建的。由于 CBrush 是第二个被创建的,所以它的析构函数首先被调用,相关的 GDI 画刷在 dc 超出范围之前就被删除了。通过先创建画笔后创建 DC,可以解决这个问题。但是如果程序代码的健壮性依赖于栈变量以特定顺序创建,那么这样的程序代码实际上是很糟糕的。就可维护性而言,这种程序简直是太可怕了。

解决方法是在 CPaintDC 对象超出范围之前将 CBrush 从设备描述表中提取出来。这里没有 UnselectObject 函数,我们可以通过选入另一个对象将当前对象从设备描述表中提出。大部分 Windows 编程人员习惯上首先保存每种对象类型首次调用 SelectObject 时返回的指针,然后用这个指针重选默认对象。还有一种同样有效的办法是将备用对象选入设备描述

表以取代当前选入的对象。前一种方法由下面的程序代码说明：

```
CPen pen (PS_SOLID, 1, RGB (255, 0, 0));
CPen * pOldPen = dc.SelectObject (&pen);
CBrush brush (RGB (0, 0, 255));
CBrush * pOldBrush = dc.SelectObject (&brush);
.
.
.
dc.SelectObject (pOldPen);
dc.SelectObject (pOldBrush);
```

第二种方法如下所示：

```
CPen pen (PS_SOLID, 1, RGB (255, 0, 0));
dc.SelectObject (&pen);
CBrush brush (RGB (0, 0, 255));
dc.SelectObject (&brush);
.
.
.
dc.SelectStockObject (BLACK_PEN);
dc.SelectStockObject (WHITE_BRUSH);
```

一个问题是为为什么要这样做呢？实际上这是不必要的。在最新版的 Windows 中，允许 GDI 对象在设备描述表释放的前一刻被删除并没有什么不好的影响，尤其是当你能确保在此期间没有画图程序执行时更是如此。但是通过取消选定选入的 GDI 对象而实现清除设备描述表仍然是 Windows 编程中的惯例。同时这也是一种好习惯，本书始终遵循这种惯例。

相应地，有时也在堆上创建 GDI 对象，如下所示：

```
CPen * pPen = new CPen (PS_SOLID, 1, RGB (255, 0, 0));
CPen * pOldPen = dc.SelectObject (pPen);
```

在某一时刻，必须将该画笔从设备描述表中选出并删除。完成该任务的程序代码可能是这样的：

```
dc.SelectObject (pOldPen);
delete pPen;
```

由于 SelectObject 函数返回一个指向被选出设备描述表的对象的指针，因此很容易让人想到用一条语句取消选定并删除该画笔：

```
delete dc.SelectObject (pOldPen);
```

但是请不要这样做。虽然这种方法对画笔管用,但不适用于画刷。为什么? 因为如果创建了两个一样的 CBrush, 则 32 位 Windows 通过只创建一个 GDI 画刷来节省内存, 这等于将两个 CBrush 指针合起来引用一个 HBRUSH。(同 HWND 确定窗口和 HDC 确定设备描述表一样, HBRUSH 是唯一确定 GDI 画刷的句柄。CBrush 将 HBRUSH 封装起来并将 HBRUSH 句柄存储在它的 m_hObject 数据成员中。)由于 CDC::SelectObject 用到由 MFC 维护的内部表格, 把从 SelectObject 返回的 HBRUSH 句柄转换为 CBrush 指针, 而且该表格形成 HBRUSH 和 CBrush 间的一一映射, 所以得到的 CBrush 指针可能和 new 返回的 CBrush 指针不一致。您要保证将 new 返回的指针传送给 delete。这样 GDI 对象和 C++ 对象就都能被正确地清除了。

2.2.12 标尺应用程序

了解 GDI 和封装了 GDI 的 MFC 类的最佳方法是编写程序代码。让我们从简单的应用程序开始。图 2-12 包含了标尺的源程序。它在屏幕上画了一个 12 英寸的标尺。标尺的输出见图 2-11。

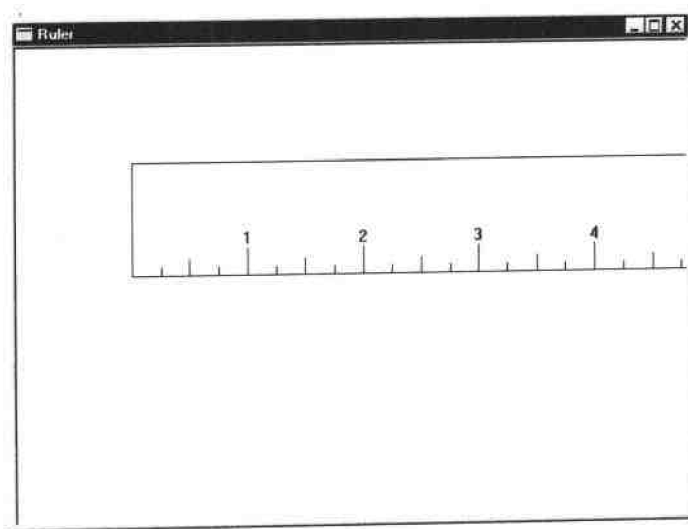


图 2-11 标尺窗口

Ruler.h

```
class CMyApp : public CWinApp
{
public:
```

```

        virtual BOOL InitInstance ();
    };

    class CMainWindow : public CFrameWnd
    {
    public:
        CMainWindow ();

    protected:
        afx_msg void OnPaint ();
        DECLARE_MESSAGE_MAP ()
    };

```

Ruler.cpp

```

#include <afxwin.h>
#include "Ruler.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_PAINT ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    Create (NULL, _T ("Ruler"));
}

void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);

    //
    // Initialize the device context.

```

```

//
dc.SetMapMode (MM_LOENGLISH);
dc.SetTextAlign (TA_CENTER|TA_BOTTOM);
dc.SetBkMode (TRANSPARENT);

//
// Draw the body of the ruler.
//

CBrush brush (RGB (255, 255, 0));
CBrush * pOldBrush = dc.SelectObject (&brush);
dc.Rectangle (100, -100, 1300, -200);
dc.SelectObject (pOldBrush);

//
// Draw the tick marks and labels.
//
for (int i = 125; i < 1300; i += 25) {
    dc.MoveTo (i, -192);
    dc.LineTo (i, -200);
}

for (i = 150; i < 1300; i += 50) {
    dc.MoveTo (i, -184);
    dc.LineTo (i, -200);
}

for (i = 200; i < 1300; i += 100) {
    dc.MoveTo (i, -175);
    dc.LineTo (i, -200);

    CString string;
    string.Format (_T ("%d"), (i / 100) - 1);
    dc.TextOut (i, -175, string);
}
}

```

图 2-12 标尺应用程序

标尺应用程序的结构与第1章给出的 Hello 应用程序相似。CMyApp 类代表应用程序本身。CMyApp::InitInstance 通过构造 CMainWindow 对象创建了一个主窗口,而 CMainWindow 的构造函数通过调用 Create 创建了屏幕上显示的窗口。CMainWindow::OnPaint 处理所有的画图任务。CDC::Rectangle 画标尺的主体,CDC::LineTo 和 CDC::MoveTo 画散列的刻度线。在画矩形之前,黄色画刷被选入设备描述表,标尺的主体因而被画成黄色。CDC::TextOut 画出数字标注,并通过调用带 TA_CENTER 和 TA_BOTTOM 标志的 SetTextAlign 和传

递给 TextOut 每个刻度线的顶点坐标将数字标注放在刻度线的上方。在首次调用 TextOut 之前,把设备描述表的背景样式设置为 TRANSPARENT。否则,标尺表面的数字会用白色背景画出。

标尺应用程序没有采用把要传递给 TextOut 的字符串硬编码的方法,而是调用 CString::Format 生成标尺上的文本。CString 是描述文本字符串的 MFC 类。CString::Format 与 C 语言中 printf 函数的作用相同,它将数字表达转换为文本,并用它们替代格式字符串中的占位符。用 C 语言工作的 Windows 编程人员常用 API 函数::wsprintf 将文本格式化。对于 CString 对象,Format 就能完成文本格式化,而不用调用其他外部函数。而且和::wsprintf 不同,Format 支持 printf 的所有格式代码,包括浮点型和字符型变量的格式代码。

标尺应用程序使用 MM_LOENGLISH 映射模式按比例缩放输出,如标尺上的 1 英寸对应于屏幕上的 1 逻辑英寸。拿一个真标尺与屏幕上的相比较,您会发现在大部分 PC 上,1 逻辑英寸要比实际中的 1 英寸长一点。如果标尺是往打印机上输出,则两者就完全一样了。

2.3 看看画了些什么

不幸的是,标尺的输出还有一个问题:除非在分辨率很高的视频适配器上运行该程序,否则您看不到它画的任何东西。即便在 1 280×1 204 像素的屏幕上,窗口也不能伸展到足够宽,以使整个输出显示出来。不能放入窗口客户区的部分被 GDI 剪除了。虽然可以修改样例程序使标尺变短,但对于在 640×480 屏幕上运行 Windows 的人来说仍然无济于事。不过,这里有更好的解决办法,它与屏幕分辨率毫无关系。这就是滚动条。

2.3.1 给窗口添加滚动条

滚动条是这样的一个窗口:窗口两端各有一个箭头,之间还有一个可用鼠标拖动的可移动“滑块”。滚动条可摆放在水平或垂直方向上,但不能放在其他任意角度方向上。单击滚动条的箭头、移动滑块,或单击滚动条体时,滚动条会向它所挂接的窗口发送消息来通知该窗口。无论发生什么情况,都由窗口决定如何响应消息,滚动条很少自己响应消息。例如:它并没有神奇地滚动窗口内容,它所做的是提供了一个直觉意义上的并被广泛认同的机制,那就是,前后滚动物理窗口无法容纳的虚拟画面。

给窗口中添加滚动条是 Windows 编程中最容易的事。如果要添加垂直滚动条,则创建 WS_VSCROLL 样式的窗口。如果要添加水平滚动条,则采用 WS_HSCROLL 样式。如果要同时添加水平滚动条和垂直滚动条,则两种样式都采用。回忆第 1 章,传递给 CFrameWnd::Create 的第三个参数就是窗口样式,它的默认值为 WS_OVERLAPPEDWINDOW。用下面的语句

```
Create(NULL, _T("My Application"));
```

创建的常规框架窗口应用程序可以用下面的语句创建一个具有垂直滚动条的框架窗口:

```
Create(NULL, _T("My Application"), WS_OVERLAPPEDWINDOW|WS_VSCROLL);
```

相应地, Windows 提供一个右侧放置的具有窗口客户区高度的滚动条。如果希望滚动条出现在左侧, 则可在 Create 可选参数 dwExStyle(第7个)中添入 WS_EX_LEFTSCROLLBAR 标志。

2.3.2 设定滚动条的范围、位置和页面大小

创建了滚动条之后, 还要给它设置范围、位置和页面大小等初始值。range 是一对整型数, 确定滚动条移动的上、下限。position 是一个整型值, 指定上述范围内的当前位置; 其值大小可从滚动条滑块的当前位置看出来。page size 设定滑块的尺寸, 直观地反映了窗口尺寸和可滚动视图尺寸间的关系。例如: 如果滚动条范围是 0 到 100, 页面尺寸是 50, 则滚动条滑块的尺寸是滚动条长度的一半。如果不设定页面尺寸, Windows 将采用默认值, 一个对您来说不合比例的滚动条滑块尺寸。

设定滚动条范围和位置的一种方法是调用 CWnd::SetScrollRange 和 CWnd::SetScrollPos 函数。语句

```
SetScrollRange(SB_VERT, 0, 100, TRUE);
```

将垂直滚动条的范围设置为 0 到 100, 而语句

```
SetScrollPos(SB_VERT, 50, TRUE);
```

将当前位置设在 50, 这样滚动条滑块就移动到滚动条的中间了。(对于水平滚动条, 则把 SB_VERT 改为 SB_HORZ。)滚动条内部保持着当前范围和位置的记录。可以在任意时刻调用 CWnd::GetScrollRange 和 CWnd::GetScrollPos 查询这些值。

传递给 SetScrollRange 和 SetScrollPos 的参数 TRUE 意味着: 需要重画滚动条来反映这些值的变化。参数值 FALSE 禁止重画。如果既不指定 TRUE 也不指定 FALSE, 则 SetScrollRange 和 SetScrollPos 均默认设置为 TRUE。一般情况下, 在调用这两个函数之后, 都要求滚动条重画自身, 除非两个函数在很短的时间里连续被调用。在很短的时间内两次重画滚动条会造成不好的闪动效果。如果要一起设置范围和位置, 可采用下面的方法:

```
SetScrollRange(SB_VERT, 0, 100, FALSE);
SetScrollPos(SB_VERT, 50, TRUE);
```

在最初的 Windows 版本中就有 SetScrollPos 和 SetScrollRange。在现在的版本中, 设定滚动条范围和位置的常用方法是调用 CWnd::SetScrollInfo 函数。除了经一个函数调用就能设定范围和位置外, SetScrollInfo 还提供了一种方法——only, 用来设置页面尺寸。SetScrollInfo

接收三个参数:

- SB_VERT 或 SB_HORZ 参数,确定滚动条是水平的还是垂直的(或 SB_BOTH,如果想一次初始化两个滚动条)
- 指向 SCROLLINFO 结构的指针
- 一个 BOOL 值(TRUE 或 FALSE),确定滚动条是否需要重画

SCROLLINFO 在 Winuser.h 中是如此定义的:

```
typedef struct tagSCROLLINFO
{
    UINT cbSize;
    UINT fMask;
    int nMin;
    int nMax;
    UINT nPage;
    int nPos;
    int nTrackPos;
} SCROLLINFO, FAR * LPSCROLLINFO;
```

cbSize 指定结构的大小,nMin 和 nMax 指定滚动条范围,nPage 指定页面尺寸,而 nPos 指定位置。在调用 SetScrollInfo 时没有用到 nTrackPos,但在用鼠标拖动滚动条滑块,辅助函数 GetScrollInfo 被调用来检索滚动条信息时,它返回滚动条滑块的位置。fMask 字段保存下面位标志中的一个或多个。

- SIF_DISABLENOSCROLL,使滚动条不可用
- SIF_PAGE,表示 nPage 保存有页面尺寸
- SIF_POS,表示 nPos 保存有滚动条位置
- SIF_RANGE,表示 nMin 和 nMax 保存有滚动条范围
- SIF_ALL,等价于 SIF_PAGE|SIF_POS|SIF_RANGE

SetScrollInfo 忽略位标志没有指定的字段。语句

```
SCROLLINFO si;
si.fMask = SIF_POS;
si.nPos = 50;
SetScrollInfo(SB_VERT, &si, TRUE);
```

设置位置时,不影响范围和页面尺寸,而语句

```
SCROLLINFO si;
si.fMask = SIF_RANGE|SIF_POS|SIF_PAGE; // Or SIF_ALL
si.nMin = 0;
si.nMax = 99;
si.nPage = 25;
si.nPos = 50;
```

```
SetScrollInfo(SB_VERT, &si, TRUE);
```

则在一次设置过程中确定了范围、页面尺寸和位置。在调用 SetScrollInfo 或 GetScrollInfo 之前不需要给 cbSize 设定初始值,因为 MFC 会替您初始化 cbSize。

当滚动条范围的上、下限设成相等时,滚动条就在窗口中消失了。但滚动条并不是完全消失,虽然看不见它,但它依旧在那儿。而且更重要的是:通过将上、下限设成不一样的值,又能让滚动条出现在窗口中。在将窗口放大到不需要滚动条的程度时,如果需要隐藏滚动条,这就成为一个很有用的技巧。SetScrollInfo 的 SIF_DISABLENOSCROLL 标志禁止滚动条接收以后的输入,但它不能使滚动条消失。将一个不可用的滚动条显示在窗口中会使用户感到困惑,他们自然想知道如果滚动条不能使用,为什么还放在那儿呢。

在设定滚动条的范围、页面尺寸和位置时,有一种方便的模型。假定要求窗口客户区为 100 单位高,而工作空间所需的滚动条为 400 单位高,则将滚动条范围设成 0~399,页面尺寸为 100。相应地,Windows 画出的滚动条滑块尺寸为滚动条高度的四分之一。滚动条位置为 0 时,滚动条滑块位于滚动条的顶端。随着滚动条滑块向下移动,窗口内容向上滚动与滚动条滑块移动距离相对应的量。如果限制滚动条的最大位置为 300(滚动条范围和页面尺寸值之差),则当工作空间末端出现在窗口底部时,滚动条滑块的底边也达到滚动条的底部。

2.3.3 使滚动条滑块大小和窗口尺寸同步变化

因为滚动条滑块的大小反映了窗口与虚拟工作空间宽或高的相对尺寸,所以当窗口尺寸变化时要更新滚动条滑块的大小。这很容易做到:每当窗口接收到 WM_SIZE 消息时,只要调用带 SIF_PAGE 标志的 SetScrollInfo 函数即可。在窗口建立时,传来第一个 WM_SIZE 消息。以后每当改变窗口尺寸都会有 WM_SIZE 消息传来。在 MFC 中,类的消息映射表里有 ON_WM_SIZE 一项,它将 WM_SIZE 消息引导到名为 OnSize 的处理程序中。该处理程序原型如下:

```
afx_msg void OnSize(UINT nType, int cx, int cy)
```

nType 参数通知窗口:窗口是否被最小化、最大化或用代码 SIZE_MINIMIZED、SIZE_MAXIMIZED 或 SIZE_RESTORED 修改了尺寸。cx 和 cy 是用户区新的宽度和高度,用像素表示。如果知道应用程序虚拟工作空间的尺寸,则滚动条滑块大小相应地也就能确定下来了。

2.3.4 处理滚动条消息

滚动条给它的拥有者(拥有它的窗口)发送消息,通知它有滚动条事件。水平滚动条发送 WM_HSCROLL 消息,垂直滚动条发送 WM_VSCROLL 消息。在 MFC 中,这些消息由窗口消息映射表中的 ON_WM_HSCROLL 和 ON_WM_VSCROLL 项引导到窗口的 OnHScroll 和 OnVScroll 函数中。滚动条消息处理程序的原型如下:

```
afx_msg void OnHScroll(UINT nCode, UINT nPos, CScrollBar * pScrollBar)
```



```
afx_msg void OnVScroll (UINT nCode, UINT nPos, CScrollBar * pScrollBar)
```

nCode 指定产生消息的事件的类型;如果滚动条滑块正在被拖动或拖动后刚刚被释放,则 nPos 保存了滚动条滑块的最新位置信息;而且对于通过向窗口添加 WS_HSCROLL 或 WS_VSCROLL 样式位而创建的滚动条,pScrollBar 为 NULL。

应用程序在 OnVScroll 的 nCode 参数中可能接收到的事件通知有 7 种,如表 2-11 所示。

表 2-11 OnVScroll 函数的 nCode 参数接收的事件

| 事件代码 | 发 送 时 间 |
|------------------|---|
| SB_LINEUP | 滚动条顶部的箭头被单击 |
| SB_LINEDOWN | 滚动条底部的箭头被单击 |
| SB_PAGEUP | 顶部箭头和滚动条滑块间的滚动条体被单击 |
| SB_PAGEDOWN | 底部箭头和滚动条滑块间的滚动条体被单击 |
| SB_ENDSCROLL | 鼠标键被释放,并且再没有 SB_LINEUP、SB_LINEDOWN、SB_PAGEUP 或 SB_PAGEDOWN 通知发来 |
| SB_THUMBTRACK | 滚动条滑块被拖动 |
| SB_THUMBPOSITION | 拖动后滚动条滑块被释放 |

水平滚动条与垂直滚动条发送的是一样的通知,但是通知的含义却稍有不同。对水平滚动条来说,SB_LINEUP 表示左箭头被单击,SB_LINEDOWN 表示右箭头被单击,SB_PAGEUP 是指左箭头与滚动条滑块间的滚动条体被单击,而 SB_PAGEDOWN 是指右箭头与滚动条滑块间的滚动条体被单击。如果您愿意,也可以用 SB_LINELEFT、SB_LINERIGHT、SB_PAGELEFT 和 SB_PAGERIGHT 替代 SB_LINEUP、SB_LINEDOWN、SB_PAGEUP 和 SB_PAGEDOWN。本章后面将专门讲述垂直滚动条,但是请注意所讲的也同样适用于水平滚动条。

如果用户单击滚动条或滚动条箭头,并始终不松开鼠标键,则一系列 SB_LINEUP、SB_LINEDOWN、SB_PAGEUP 或 SB_PAGEDOWN 通知就会很快地接踵而至,与按键被按下时生成自动重复键入的键代码流的情况类似。SB_ENDSCROLL 终止 UP 或 DOWN 通知,并指示已释放鼠标键。即使只是单击一次滚动条体或箭头也会生成一个 UP 或 DOWN 通知,后随 SB_ENDSCROLL 通知。同样,在滚动条滑块被拖动时,窗口要应付不断前来通知拇指新位置的 SB_THUMBTRACK 通知;而在鼠标被释放时,接收到 SB_THUMBPOSITION 通知。当 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知到来的时候,消息的 nPos 参数保留有最新的滚动条滑块位置。对于其他事件代码,nPos 的值没有定义。

程序如何响应滚动条事件的消息取决于编程人员。大部分使用滚动条的程序忽略 SB_ENDSCROLL 消息,而响应 SB_LINEUP、SB_LINEDOWN、SB_PAGEUP 和 SB_PAGEDOWN 消息。通常对 SB_LINEUP 和 SB_LINEDOWN 消息的响应是把窗口内容向上或向下滚动一行,并调用 SetScrollPos 或 SetScrollInfo 设定滚动条的新位置和更新滚动条滑块位置。“行”可以具有您希望的任何物理含义,它或指一个像素,或表示一行文本的高度。同样地,常见的对

SB_PAGEUP 和 SB_PAGEDOWN 消息的响应是向上或向下滚动一“页”或稍小于一“页”的距离(通常该距离定义为窗口客户区高度或比窗口客户区高度稍小一点的高度),并调用 SetScrollInfo 设定滚动条的新位置。任何事件中,更新滚动条位置都是编程者的责任。滚动条不会自己做这件事。

另一种不太常用的处理 UP 和 DOWN 通知的方法是通过调用 SetScrollPos 或 SetScrollInfo 不断移动滚动条滑块,但直到 SB_ENDSCROLL 通知传来延迟滚动窗口。我曾在一个多媒体应用程序中用过这个方法。程序对位置变化反应很慢,以致于给 CD-ROM 驱动器传送命令的延迟时间都不会阻碍滚动条滑块的平滑移动。

SB_THUMBTRACK 和 SB_THUMBPOSITION 通知的处理方法稍有不同。因为在滚动条滑块被拖动时,SB_THUMBTRACK 通知通常来得很快、很多,所以一些 Windows 应用程序忽略 SB_THUMBTRACK 通知而只对 SB_THUMBPOSITION 通知作出响应。在这种情况下,窗口在滚动条滑块被释放之前不会发生滚动。如果滚动窗口内容的速度能跟上 SB_THUMBTRACK 通知,那么程序对滚动条滑块被拖动、窗口内容随之滚动的用户输入的响应性就更好了。但滚动窗口并同时相应地更新滚动条位置依旧是编程者的责任。滚动条滑块在被向上或向下拖动时,Windows 将滚动条的移动以动画显示,但是如果不能调用 SetScrollPos 或 SetScrollInfo 响应 SB_THUMBTRACK 或 SB_THUMBPOSITION 通知,则滚动条滑块在被释放时将快速回到初始位置。

2.3.5 滚动窗口

现在既然您已了解了滚动条是如何工作的,那么就该考虑如何响应滚动条消息滚动窗口内容了。

最简单的方法是在每个滚动条消息传来时改变滚动条的位置,并调用 CWnd::Invalidate 实现重画滚动条。窗口的 OnPaint 处理程序会向滚动条查询它的当前位置,并根据该信息调整它的输出。不幸的是:用这种方法滚动窗口的速度很慢。用户单击向上箭头,窗口内容向上滚动一行,尽管大部分内容的显示位置不对,但都已在窗口内,重画整个窗口实际上是浪费时间。用来处理 SB_LINEUP 消息的更有效的办法是用数据块快速拷贝法将当前显示在窗口中的所有内容复制到第二行以后的位置,然后画出新的第一行。这就是 CWnd::ScrollWindow 的作用。

ScrollWindow 向上或向下、向左或向右滚动窗口客户区的整体或局部内容,并使用像素块快速传输方法实现一个或多个像素距离的滚动。而且它禁止滚动没有被滚动操作“覆盖”的窗口内容,这样下一个 WM_PAINT 消息就不会重画整个窗口。如果调用 ScrollWindow 将窗口向下滚动 10 个像素,则用数据块拷贝实现滚动,然后使窗口中的前 10 行无效。此时 OnPaint 被激活,只有前 10 行被重画。即使 OnPaint 要重画整个用户区的内容,由于大部分输出被剪贴过去,所以滚动性能还是提高了。有一个巧妙的 OnPaint 处理程序能进一步加快任务的执行,它将 GDI 调用限制在那些对窗口中无效矩形里的像素有影响的函数。第 10 和第 13 章将给出运用这种技巧优化滚动性能的示例程序。

ScrollWindow 接收四个参数。两个是必需的,两个是可选的。函数原型如下:

```
void ScrollWindow (int xAmount, int yAmount,
    LPCRECT lpRect = NULL, LPCRECT lpClipRect = NULL)
```

xAmount 和 yAmount 是带符号的整型数,确定垂直或水平方向上要滚动的像素数目。负值表示向左和向上滚动,正值表示向右和向下滚动。lpRect 指向一个 CRect 对象或 RECT 结构,指定客户区中待滚动的部分;而 lpClipRect 指向一个 CRect 对象或 RECT 结构,指定裁剪矩形。滚动整个客户区内容时,应将 lpRect 和 lpClipRect 指定为 NULL。语句

```
ScrollWindow (0, 10);
```

将窗口客户区所有内容向下滚动 10 个像素,并立即重画前十行。

应用程序无论输出文本、图形或同时输出文本和图形,您都可以调用 ScrollWindow。在 Windows 中,一切都是图形的——包括文本。

2.3.6 Accel 应用程序

现在让我们运用所学的知识编一个实现滚动的应用程序吧。Accel 画出一个类似于 Microsoft Excel 的窗口(参见图 2-13)。这个窗口描述的电子表格有 26 列宽、99 行高。表格太大,一次无法显示全部内容。然而滚动条可帮助用户浏览电子表格的全部内容。除了提供了动手实践前述准则的机会外,Accel 还从另一个角度证明了应用程序可以按比例缩放输出。Accel 没有使用非 MM_TEXT 映射模式,而是调用 CDC::GetDeviceCaps 向显示设备查询水平和垂直方向上每英寸内显示的像素数。然后,根据像素数画出每个单元格,使它成为 1 英寸宽 \times $\frac{1}{4}$ 英寸高。

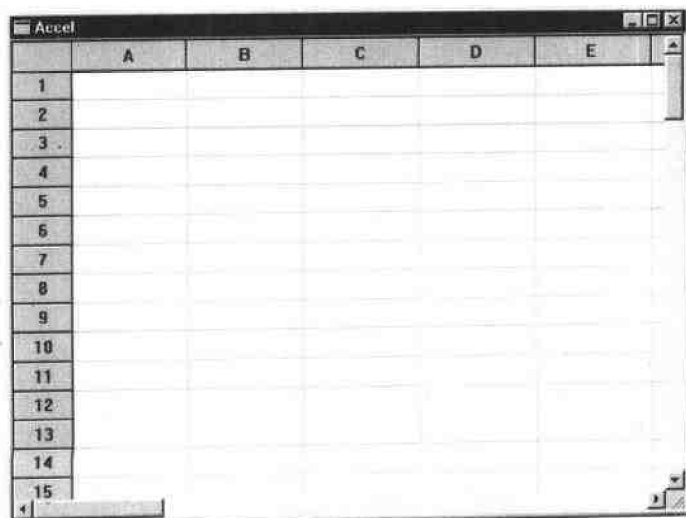


图 2-13 Accel 窗口

Accel.h

```

#define LINE_SIZE 8

class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CFrameWnd
{
protected:
    int m_nCellWidth;    // Cell width in pixels
    int m_nCellHeight;   // Cell height in pixels
    int m_nRibbonWidth;  // Ribbon width in pixels
    int m_nViewWidth;    // Workspace width in pixels
    int m_nViewHeight;   // Workspace height in pixels
    int m_nHScrollPos;   // Horizontal scroll position
    int m_nVScrollPos;   // Vertical scroll position
    int m_nHPageSize;    // Horizontal page size
    int m_nVPageSize;    // Vertical page size

public:
    CMainWindow();
protected:
    afx_msg void OnPaint();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnHScroll(UINT nCode, UINT nPos,
        CScrollBar * pScrollBar);
    afx_msg void OnVScroll(UINT nCode, UINT nPos,
        CScrollBar * pScrollBar);

    DECLARE_MESSAGE_MAP()
};

```

Accel.cpp

```

#include <afxwin.h>
#include "Accel.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance()

```

```

|
|   m_pMainWnd = new CMainWindow;
|   m_pMainWnd->ShowWindow (m_nCmdShow);
|   m_pMainWnd->UpdateWindow ();
|   return TRUE;
|

////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_CREATE ()
    ON_WM_SIZE ()
    ON_WM_PAINT ()
    ON_WM_HSCROLL ()
    ON_WM_VSCROLL ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    Create (NULL, _T ("Accel"),
            WS_OVERLAPPEDWINDOW|WS_HSCROLL|WS_VSCROLL);
}

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

    CClientDC dc (this);
    m_nCellWidth = dc.GetDeviceCaps (LOGPIXELSX);
    m_nCellHeight = dc.GetDeviceCaps (LOGPIXELSY) / 4;
    m_nRibbonWidth = m_nCellWidth / 2;
    m_nViewWidth = (26 * m_nCellWidth) + m_nRibbonWidth;
    m_nViewHeight = m_nCellHeight * 100;
    return 0;
}

void CMainWindow::OnSize (UINT nType, int cx, int cy)
{
    CFrameWnd::OnSize (nType, cx, cy);

    //
    // Set the horizontal scrolling parameters.
    //
    int nHScrollMax = 0;
    m_nHScrollPos = m_nHPageSize = 0;
}

```

```

    if (cx < m_nViewWidth) {
        nHScrollMax = m_nViewWidth - 1;
        m_nHPageSize = cx;
        m_nHScrollPos = min (m_nHScrollPos, m_nViewWidth -
            m_nHPageSize - 1);
    }

    SCROLLINFO si;
    si.fMask = SIF_PAGE|SIF_RANGE|SIF_POS;
    si.nMin = 0;
    si.nMax = nHScrollMax;
    si.nPos = m_nHScrollPos;
    si.nPage = m_nHPageSize;

    SetScrollInfo (SB_HORZ, &si, TRUE);

    //
    // Set the vertical scrolling parameters.
    //
    int nVScrollMax = 0;
    m_nVScrollPos = m_nVPageSize = 0;

    if (cy < m_nViewHeight) {
        nVScrollMax = m_nViewHeight - 1;
        m_nVPageSize = cy;
        m_nVScrollPos = min (m_nVScrollPos, m_nViewHeight -
            m_nVPageSize - 1);
    }

    si.fMask = SIF_PAGE|SIF_RANGE|SIF_POS;
    si.nMin = 0;
    si.nMax = nVScrollMax;
    si.nPos = m_nVScrollPos;
    si.nPage = m_nVPageSize;

    SetScrollInfo (SB_VERT, &si, TRUE);
}

void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);

    //
    // Set the window origin to reflect the current scroll positions.
    //
    dc.SetWindowOrg (m_nHScrollPos, m_nVScrollPos);

    //
    // Draw the grid lines.

```

```

//
CPen pen (PS_SOLID, 0, RGB (192, 192, 192));
CPen* pOldPen = dc.SelectObject (&pen);

for (int i=0; i<99; i++) {
    int y = (i * m_nCellHeight) + m_nCellHeight;
    dc.MoveTo (0, y);
    dc.LineTo (m_nViewWidth, y);
}

for (int j=0; j<26; j++)
    int x = (j * m_nCellWidth) + m_nRibbonWidth;
    dc.MoveTo (x, 0);
    dc.LineTo (x, m_nViewHeight);
}

dc.SelectObject (pOldPen);

//
// Draw the bodies of the rows and the column headers.
//
CBrush brush;
brush.CreateStockObject (LTGRAY_BRUSH);

CRect rcTop (0, 0, m_nViewWidth, m_nCellHeight);
dc.FillRect (rcTop, &brush);
CRect rcLeft (0, 0, m_nRibbonWidth, m_nViewHeight);
dc.FillRect (rcLeft, &brush);

dc.MoveTo (0, m_nCellHeight);
dc.LineTo (m_nViewWidth, m_nCellHeight);
dc.MoveTo (m_nRibbonWidth, 0);
dc.LineTo (m_nRibbonWidth, m_nViewHeight);

dc.SetBkMode (TRANSPARENT);

//
// Add numbers and button outlines to the row headers.
//
for (i=0; i<99; i++) {
    int y = (i * m_nCellHeight) + m_nCellHeight;
    dc.MoveTo (0, y);
    dc.LineTo (m_nRibbonWidth, y);

    CString string;
    string.Format ( "P (%d)", i + 1);

    CRect rect (0, y, m_nRibbonWidth, y + m_nCellHeight);
    dc.DrawText (string, &rect, DT_SINGLELINE)

```

```

        DT_CENTER|DT_VCENTER);

    rect.top++;
    dc.Draw3dRect (rect, RGB (255, 255, 255),
        RGB (128, 128, 128));
}

//
// Add letters and button outlines to the column headers.
//

for (j = 0; j < 26; j++) {
    int x = (j * m_nCellWidth) + m_nRibbonWidth;
    dc.MoveTo (x, 0);
    dc.LineTo (x, m_nCellHeight);

    CString string;
    string.Format (_T("%c"), j + 'A');

    CRect rect (x, 0, x + m_nCellWidth, m_nCellHeight);
    dc.DrawText (string, &rect, DT_SINGLELINE|
        DT_CENTER|DT_VCENTER);

    rect.left++;
    dc.Draw3dRect (rect, RGB (255, 255, 255),
        RGB (128, 128, 128));
}

}

void CMainWindow::OnHScroll (UINT nCode, UINT nPos, CScrollBar * pScrollBar)
{
    int nDelta;

    switch (nCode) {
    case SB_LINELEFT:
        nDelta = -LINE_SIZE;
        break;

    case SB_PAGELEFT:
        nDelta = -m_nHPageSize;
        break;

    case SB_THUMBTRACK:
        nDelta = (int) nPos - m_nHScrollPos;
        break;

    case SB_PAGERIGHT:
        nDelta = m_nHPageSize;
        break;
    }
}

```

```

        case SB_LINERIGHT:
            nDelta = LINE_SIZE;
            break;
        default: // ignore other scroll bar messages
            return;
    }

    int nScrollPos = m_nHScrollPos + nDelta;
    int nMaxPos = m_nViewWidth - m_nHPageSize;

    if (nScrollPos < 0)
        nDelta = -m_nHScrollPos;
    else if (nScrollPos > nMaxPos)
        nDelta = nMaxPos - m_nHScrollPos;

    if (nDelta != 0) {
        m_nHScrollPos += nDelta;
        SetScrollPos(SB_HORZ, m_nHScrollPos, TRUE);
        ScrollWindow(-nDelta, 0);
    }
}

void CMainWindow::OnVScroll(UINT nCode, UINT nPos, CScrollBar * pScrollBar)
{
    int nDelta;

    switch (nCode) {
        case SB_LINEUP:
            nDelta = -LINE_SIZE;
            break;

        case SB_PAGEUP:
            nDelta = -m_nVPageSize;
            break;

        case SB_THUMBTRACK:
            nDelta = (int) nPos - m_nVScrollPos;
            break;

        case SB_PAGEDOWN:
            nDelta = m_nVPageSize;
            break;

        case SB_LINEDOWN:
            nDelta = LINE_SIZE;
            break;
        default: // ignore other scroll bar messages
            return;
    }
}

```

```

    int nScrollPos = m_nVScrollPos + nDelta;
    int nMaxPos = m_nViewHeight - m_nVPageSize;

    if (nScrollPos < 0)
        nDelta = -m_nVScrollPos;
    else if (nScrollPos > nMaxPos)
        nDelta = nMaxPos - m_nVScrollPos;

    if (nDelta != 0) {
        m_nVScrollPos += nDelta;
        SetScrollPos(SB_VERT, m_nVScrollPos, TRUE);
        ScrollWindow(0, -nDelta);
    }
}

```

图 2-14 Accel 应用程序

当接收到 WM_CREATE 消息时, CMainWindow 的 OnCreate 处理程序调用 GetDeviceCaps。WM_CREATE 是窗口接收到的第一个消息。这个消息只发送一次,并在窗口的生存期之初就到——甚至在窗口还没有被显示在屏幕上之前。窗口消息映射表中的 ON_WM_CREATE 项将 WM_CREATE 消息连接到成员函数 OnCreate。有一些成员变量的值只能在运行中决定, OnCreate 则是初始化这些变量的理想方法。它的原型为:

```
afx_msg int OnCreate (LPCREATESTRUCT lpCreateStruct)
```

lpCreateStruct 指向一个 CREATESTRUCT 类型的结构,其中包含了一些关于窗口的重要信息,比如窗口的原始尺寸和在屏幕上的位置。由 OnCreate 返回的值决定了窗口的下一步行为。如果一切按计划进行,则 OnCreate 返回 0,通知 Windows 窗口已被正确初始化。如果 OnCreate 返回 -1,则 Windows 将不能创建这个窗口。OnCreate 处理程序原型如下:

```

int CMainWindow::OnCreate (LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate (lpCreateStruct) == -1)
        return -1;
    .
    .
    .
    return 0;
}

```

OnCreate 应该总是调用基类的 OnCreate 处理程序,以使框架结构有机会执行自己的窗口创建代码。在编写文档/视图应用程序时这一点尤其重要,因为 OnCreate 是由创建框架窗口中视图的 CFrameWnd::OnCreate 调用的。

您还会在窗口的 OnHScroll 和 OnVScroll 处理程序中发现实现滚动的代码。switch-case

逻辑将 `nCode` 传送的通知转换成带符号的 `nDelta` 值,表示窗口需要滚动的像素数目。一旦计算出 `nDelta`,滚动条位置会根据 `nDelta` 像素数改变,而滚动窗口由下面的语句实现:

```
m_nVScrollPos += nDelta;
SetScrollPos(SB_VERT, m_nVScrollPos, TRUE);
ScrollWindow(0, -nDelta);
```

以上用于垂直滚动条,

```
m_nHScrollPos += nDelta;
SetScrollPos(SB_HORZ, m_nHScrollPos, TRUE);
ScrollWindow(-nDelta, 0);
```

以上用于水平滚动条。

存储在 `m_nHScrollPos` 和 `m_nVScrollPos` 中的滚动条位置是如何添加到程序输出中的呢?在调用 `OnPaint` 绘制滚动操作所揭示的一部分工作空间时,`OnPaint` 用下面的语句重置窗口的原点:

```
dc.SetWindowOrg(m_nHScrollPos, m_nVScrollPos);
```

`CDC::SetWindowOrg` 通知 Windows 将逻辑点 (x,y) 映射为设备点 $(0,0)$ 。对客户区设备描述表来说,该点对应窗口客户区的左上角。上面的语句将坐标系原点向左移动 `m_nHScrollPos` 个像素点,向上移动 `m_nVScrollPos` 个像素点。如果 `OnPaint` 要把像素点画在 $(0,0)$,则 GDI 将坐标对显式转化为 $(-m_nHScrollPos, -m_nVScrollPos)$ 。如果滚动条位置为 $(0,100)$,那么程序输出的前面 100 行像素会被剪裁下来,而用户可以看见的输出是从第 101 行开始的。用这种方法重置原点可以在虚拟显示平面上简捷有效地移动滚动式窗口。

如果能放大窗口直到整个电子表格显示出来,那么你会发现这时滚动条消失了。这是因为如果窗口尺寸等于或大于虚拟工作空间,则 `CMainWindow::OnSize` 将滚动条范围设为 0。只要窗口尺寸发生变化,`OnSize` 处理程序就会更新滚动参数,使滚动条滑块准确反映窗口和虚拟工作空间的相对比例。

这样,所有的功能模块都具备了。用户单击滚动条或拖动滚动条滑块,`OnHScroll` 或 `OnVScroll` 接收消息并相应地更新滑块位置和滚动窗口;`SetWindowOrg` 移动绘图原点到当前滑块位置,`OnPaint` 重画窗口。尽管窗口尺寸对程序输出有一些物理限制,但现在整个工作空间都可以访问了,这只要少于 100 行附加代码就能做到。还能更容易吗?

这么问很好笑。因为这正是 MFC `CScrollView` 类的目的——使滚动实现起来更加容易些。`CScrollView` 是一个 MFC 类,封装了滚动式窗口的行为。只要告诉 `CScrollView` 需要多大的视图,其他工作就都由它处理了。另外,`CScrollView` 处理 `WM_VSCROLL` 和 `WM_HSCROLL` 消息,滚动窗口以响应滚动条事件,并在窗口尺寸变化时更新滚动条滑块的大小。

尽管将 `CScrollView` 写入一个应用程序(如 `Accel`)完全有可能,但 `CScrollView` 主要用于文档/视图应用程序。第 10 章将更详细地讲解 `CScrollView`,并介绍 MFC 提供的一些其他的视

图类。

2.4 遗留问题

在结束这一章之前,我们还要解决一些遗留问题。到现在,所有给出的程序都在 `InitInstance` 中用下面的语句创建了一个窗口:

```
m_pMainWnd = new CMainWindow;
```

因为对象是用 `new` 初始化的,所以 `InitInstance` 结束后对象还存留在内存中。实际上,直到用 `delete` 语句将它删除后,对象才会被清除。然而,在程序源代码中您找不到这个语句。在表面上这似乎是个问题,毕竟每个 C++ 编程人员都知道:每个 `new` 必须有一个 `delete` 来配合,否则对象就残留在内存中了。

可能正如您预料的,类库可以自动删除对象。更准确地说,对象自己删除了自己。这个小技巧的关键在于,窗口在消除之前接收到的最后一个消息是 `WM_NCDESTROY`。看看源代码中的 `CWnd::OnNcDestroy`,您会发现程序调用了一个虚函数 `PostNcDestroy`。`CFrameWnd` 覆盖 `PostNcDestroy` 并执行了

```
delete this;
```

语句。因此在框架窗口被清除时,与窗口有关的对象也自动被删除了。框架窗口在用户关闭应用程序时被清除。

值得注意的是,`CWnd` 自己实现的 `PostNcDestroy` 并没有删除相关的窗口对象。因此,如果您的窗口类是直接从 `CWnd` 派生来的,那么还要在派生类中覆盖 `PostNcDestroy` 并执行一个 `delete this` 语句。否则,`CWnd` 对象就不会被正确地删除。在下一章您就会明白我的意思。

第3章 鼠标和键盘

如果生活像电影上演的那样,那么传统的输入设备早就已经让位给语音识别单元、3D耳机以及别的人机接口配件了。但是,目前我们仍然保留着两个最普通的输入设备:鼠标和键盘。Microsoft Windows 自己处理一些鼠标和键盘输入,例如,当用户在菜单栏上单击一个项目时,会自动拉下一个菜单,并在菜单中的某个项目被选定后给应用程序发送一个 WM_COMMAND 消息。要编写一个不直接处理鼠标或键盘输入的功能完整的 Windows 程序是完全可能的,但是作为一个应用程序开发者,您最终将会发现有必要直接接收鼠标和键盘输入。如果要这样做,您需要了解 Windows 提供的鼠标和键盘接口。

不必奇怪,鼠标和键盘输入以消息的形式出现。设备驱动程序处理鼠标和键盘中断并将结果事件通知放在一个系统范围队列中,该队列称为原始输入队列。与常规消息一样,在原始输入队列的输入项也用 WM_ 作为消息标识符,但是其中的数据在被应用程序使用以前要做进一步的处理。操作系统拥有一个专门的线程来监视原始输入队列,它把每个从队列中出来的消息都转移到适当的线程消息队列中。稍后,对消息数据的处理是在接收应用程序的描述表中进行的,像对其他任何消息一样,消息最终将被检索并调度。

这种输入模式与 16 位 Windows 不同,它始终将鼠标和键盘消息保存在单个系统范围输入队列中直到消息被应用程序检索。这种安排是一种按部就班的方式,如果一个应用程序不能够及时处理输入的消息,那么它也必将阻止其他应用程序这样做。Win32 的异步输入模式解决了这个问题,它使用原始输入队列作为临时的保存缓冲区,并以最大的可能性将输入消息移动给线程消息队列。(在 32 位 Windows 操作系统中,调用 Windows API 函数的每个线程都有自己的消息队列,因此一个多线程应用程序具有多个消息队列。)一个 32 位应用程序并不会去检查消息队列,它懒惰地响应用户输入,但是这并不会影响在系统中运行的其他应用程序的响应能力。

学习响应 Windows 应用程序中鼠标和键盘的输入主要就是了解处理哪个消息的问题。本章将介绍鼠标和键盘消息以及多个 MFC 和 API 函数,这些函数对于消息处理是很有用的。我们将通过开发 3 个样例应用程序把这里给出的概念应用到现实世界中:

- TicTac, 一个井字游戏,用来说明如何响应鼠标单击
- MouseCap, 一个简单的绘图程序,用来说明鼠标的截获功能以及非客户区鼠标消息的处理过程
- VisualKB, 一个打字程序,用来将鼠标和键盘处理程序结合在一起,并列出了它接收到的键盘消息

我们还有很多内容要学,现在开始吧。

3.1 从鼠标获取输入

Windows 有 20 多种不同的消息用来报告与鼠标有关的输入事件。这些消息可分为两大类:客户区鼠标消息,用来报告窗口客户区里发生的事件,非客户区鼠标消息,它从属于窗口非客户区中的事件。一个“事件”可能是下列任何一种操作:

- 按下或释放一个鼠标键
- 双击鼠标键
- 移动鼠标

通常可以忽略非客户区内的事件,让 Windows 去处理它们。如果您的程序处理鼠标输入,只有客户区鼠标消息可能才是您关心的。

3.1.1 客户区鼠标消息

Windows 用表 3-1 列出的消息来报告窗口客户区中发生的鼠标事件。

表 3-1 客户区鼠标消息

| 消 息 | 发 送 条 件 |
|---------------------|-------------|
| WM_LBUTTONDOWN | 鼠标左键被按下 |
| WM_LBUTTONUP | 鼠标左键被释放 |
| WM_LBUTTONDOWNBLCLK | 鼠标左键被双击 |
| WM_MBUTTONDOWN | 鼠标中间键被按下 |
| WM_MBUTTONUP | 鼠标中间键被释放 |
| WM_MBUTTONDOWNBLCLK | 鼠标中间键被双击 |
| WM_RBUTTONDOWN | 鼠标右键被按下 |
| WM_RBUTTONUP | 鼠标右键被释放 |
| WM_RBUTTONDOWNBLCLK | 鼠标右键被双击 |
| WM_MOUSEMOVE | 在窗口客户区移动了光标 |

以 WM_LBUTTON 开头的消息属于鼠标左键,WM_MBUTTON 消息属于鼠标中键,WM_RBUTTON 消息属于右键。如果鼠标只有两个键,应用程序将不会接收 WM_MBUTTON 消息。(此规则有一个重要的例外:带有鼠标轮的鼠标。本章稍后内容将讨论这种鼠标。)如果鼠标仅有一个键,应用程序将不会接收 WM_RBUTTON 消息。绝大多数运行 Windows 的 PC 机都使用带有两个键的鼠标,我们几乎可以断定他们的鼠标右键肯定存在。但是,如果您想证实(或者如果想确定是否有第三个键存在),可以使用 Windows::GetSystemMetrics API 函数:

```
int nButtonCount = ::GetSystemMetrics(SM_CMOUSEBUTTONS);
```

返回值是鼠标键的数目,极少数情况下返回 0 值,说明没有安装鼠标。

WM_xBUTTONDOWN 和 WM_xBUTTONUP 消息报告鼠标键的按下和释放。WM_LBUTTONDOWNUP 消息通常跟随 WM_LBUTTONDOWN 消息,但不要认为这是理所当然的。鼠标消息被发送到光标下面的窗口(光标是鼠标指针的 Windows 术语),因此如果用户在一个窗口的客户区单击了左键,然后在释放之前将光标移动到了窗口外面,窗口会只接收一个 WM_LBUTTONDOWN 消息而不会有 WM_LBUTTONUP 消息。许多程序仅仅响应键被按下消息而忽略键释放消息,在此情形下两个消息配对就不重要了。如果有必要匹配使用,程序可以在接收到键按下消息后将鼠标“捕获”并在键被释放消息到来后将其释放。在此过程中,所有鼠标消息,包括那些属于窗口外事件的消息,都要送到执行鼠标捕获的窗口中。这就确保了无论鼠标键被释放时光标在何处,都会接收到鼠标键被释放消息。鼠标的捕获将在本章稍后部分介绍。

当同一个键在很短的时间内被连续单击两下时,第二次键按下消息将被 WM_xBUTTONDBLCLK 消息取代。并且,只有当窗口的 WNDCLASS 包含了类样式 CS_DBLCLKS 时方会这样。MFC 为框架窗口注册的默认 WNDCLASS 具有此样式,因此在默认情况下框架窗口接收双击消息。对于 CS_DBLCLKS 样式的窗口,在窗口客户区左键两次快速单击会产生下列顺序的消息:

```
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDBLCLK
WM_LBUTTONUP
```

但是,如果窗口没有注册接收双击消息,同一个键被单击两次会产生下列顺序的消息:

```
WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDOWN
WM_LBUTTONUP
```

应用程序怎样响应这些消息或根本不响应它们都完全取决于您。但是,您也应该注意不要对同一个键的连续单击和双击执行两个不相关的任务。一个单击消息总是优先于双击消息,这样产生两个消息的操作就不容易被分开。能够处理同一个键的单击和双击消息的应用程序通常在第一次单击后选中一个对象,在第二次单击时再在那个对象上执行一些操作。例如:当您在 Windows 资源管理器右窗格中双击一个文件夹时,第一次单击会选中文件夹,第二次单击将打开它。

WM_MOUSEMOVE 消息报告光标在窗口客户区内的移动。当鼠标移动时,在光标下面

的窗口会接收到快速报告光标最近位置的 WM_MOUSEMOVE 消息。Windows 用一种有趣的方法来投递 WM_MOUSEMOVE 消息,以防止运行缓慢的应用程序被光标移动时产生的大量报告位置的消息淹没。Windows 并没有把每次移动鼠标产生的 WM_MOUSEMOVE 消息装填在消息队列中,而是仅仅在一个内部数据结构中设置了一个标志。下一次应用程序检索消息时,Windows 由于设置了标志,将产生报告当前光标坐标的 WM_MOUSEMOVE 消息。这样,应用程序接收到的 WM_MOUSEMOVE 消息数量正好适合它们处理。如果光标很慢地移动,除非应用程序忙于执行其他任务,否则会报告光标轨迹上的所有点位置。但是如果光标飞快地划过屏幕,大多数应用程序只会接收到少数几个 WM_MOUSEMOVE 消息。

在 MFC 程序中,消息映射表的输入项将鼠标消息传递给处理这些消息的类成员函数。表 3-2 列出了客户区鼠标消息的消息映射宏和消息处理程序名称。

表 3-2 客户区鼠标消息的消息映射宏和消息处理程序

| 消息 | 消息映射宏 | 处理函数 |
|---------------------|------------------------|------------------|
| WM_LBUTTONDOWN | ON_WM_LBUTTONDOWN | OnLButtonDown |
| WM_LBUTTONUP | ON_WM_LBUTTONUP | OnLButtonUp |
| WM_LBUTTONDOWNBLCLK | ON_WM_LBUTTONDOWNBLCLK | OnLButtonDownBlk |
| WM_MBUTTONDOWN | ON_WM_MBUTTONDOWN | OnMButtonDown |
| WM_MBUTTONUP | ON_WM_MBUTTONUP | OnMButtonUp |
| WM_MBUTTONDOWNBLCLK | ON_WM_MBUTTONDOWNBLCLK | OnMButtonDownBlk |
| WM_RBUTTONDOWN | ON_WM_RBUTTONDOWN | OnRButtonDown |
| WM_RBUTTONUP | ON_WM_RBUTTONUP | OnRButtonUp |
| WM_RBUTTONDOWNBLCLK | ON_WM_RBUTTONDOWNBLCLK | OnRButtonDownBlk |
| WM_MOUSEMOVE | ON_WM_MOUSEMOVE | OnMouseMove |

OnLButtonDown 和其他客户区鼠标消息处理函数的原型如下:

```
afx_msg void OnMsgName (UINT nFlags, CPoint point)
```

其中 point 指出光标位置。在 WM_xBUTTONDOWN 和 WM_xBUTTONDOWNBLCLK 消息中,当键被按下时,point 指出光标此时的位置。WM_xBUTTONUP 消息中,point 指出键被释放时光标的位置。在 WM_MOUSEMOVE 消息中,point 指出最近的光标位置。在所有情况下,位置坐标都是以相对于窗口客户区左上角的设备坐标而报告的。如果一个 WM_LBUTTONDOWN 消息中 point.x 的值是 32 而 point.y 的值为 64 则说明鼠标键单击的位置在客户区左上角往右 32 像素往下 64 像素的地方。如果有必要,可以使用 MFC 的 CDC::DPtoLP 函数将这些坐标转换为逻辑坐标。

nFlags 参数指出了消息生成时鼠标键以及 Shift 键和 Ctrl 键的状态。通过测试表 3-3 列

出的位标志,可以从此参数中得到特殊按钮或键被释放还是按下的状态。

表 3-3 nFlags 参 数

| 掩码 | 涵 义 |
|------------|------------|
| MK_LBUTTON | 鼠标左键被按下 |
| MK_MBUTTON | 鼠标中间键被按下 |
| MK_RBUTTON | 鼠标右键被按下 |
| MK_CONTROL | Ctrl 键被按下 |
| MK_SHIFT | Shift 键被按下 |

当且仅当鼠标左键被按下时,表达式

```
nFlags &MK_LBUTTON
```

才是非零值,而此时只有 Ctrl 键处于按下状态

```
nFlags &MK_CONTROL
```

才是非零值。如果 Shift 或 Ctrl 键被按下,一些程序会对鼠标事件进行不同的响应。例如,通过检查在 nFlags 参数中的 MK_CONTROL 位以及 WM_MOUSEMOVE 消息,一个绘图程序可能会限制用户在按着 Ctrl 键移动鼠标时只可以绘制水平或垂直线。又如,在拖放操作完成时,Windows 内部命令将 MK_CONTROL 位解释为被拖放的对象应该被拷贝而不是被移动。

3.1.2 TicTac 应用程序

让我们来看一个从鼠标获取输入的示例应用程序,它说明了处理鼠标消息是很容易的。TicTac 是一个井字游戏程序,其输出结果如图 3-1 所示,它响应 3 种类型的客户区鼠标事件,单击左键,单击右键,以及双击左键。在空格上单击鼠标左键会在其中放置一个 X。而单击右键会把 O 放在空格中。(程序保证 X 和 O 轮流出现来防止作弊。)用鼠标左键双击分隔用的黑粗线可以清除棋盘并重新开始游戏。在每个 X 或 O 放置好以后,程序将检查谁是获胜者或者是否是和局。和局就是当 9 个方格都被填充以后,没有哪个选手在水平、垂直或对角线上把棋子摆成了一排。

除了提供对鼠标消息处理的实际演示之外,TicTac 还介绍了一些便于使用的新 MFC 函数,如 CWnd::MessageBox,它显示一个消息框,以及 CRect::PtInRect,用来迅速告诉您是否一个点落入了由 CRect 对象代表的矩形内。TicTac 的源程序代码在图 3-2 中给出。

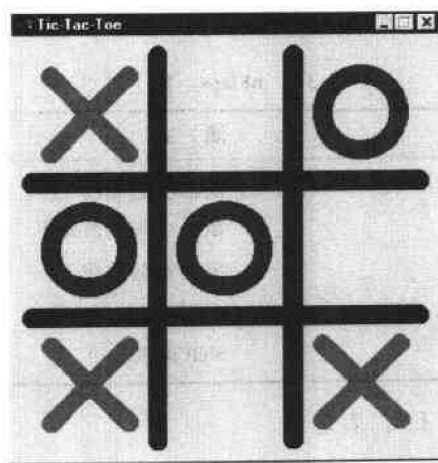


图 3-1 TicTac 程序窗口

TicTac.h

```
#define EX 1
#define OH 2

class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CWnd
{
protected:
    static const CRect m_rcSquares[9]; // Grid coordinates
    int m_nGameGrid[9];                // Grid contents
    int m_nNextChar;                    // Next character (EX or OH)
    int GetRectID(CPoint point);
    void DrawBoard(CDC * pDC);
    void DrawX(CDC * pDC, int nPos);
    void DrawO(CDC * pDC, int nPos);
    void ResetGame();
    void CheckForGameOver();
    int IsWinner();
    BOOL IsDraw();

public:
    CMainWindow();
};
```

```
protected:
    virtual void PostNcDestroy ();

    afx_msg void OnPaint ();
    afx_msg void OnLButtonDown (UINT nFlags, CPoint point);
    afx_msg void OnLButtonDblClk (UINT nFlags, CPoint point);
    afx_msg void OnRButtonDown (UINT nFlags, CPoint point);

    DECLARE_MESSAGE_MAP ()
};
```

TicTac.cpp

```
#include <afxwin.h>
#include "TicTac.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CWnd)
    ON_WM_PAINT ()
    ON_WM_LBUTTONDOWN ()
    ON_WM_LBUTTONDBLCLK ()
    ON_WM_RBUTTONDOWN ()
END_MESSAGE_MAP ()

const CRect CMainWindow::m_rcSquares[9] = {
    CRect ( 16,  16, 112, 112),
    CRect (128,  16, 224, 112),
    CRect (240,  16, 336, 112),
    CRect ( 16, 128, 112, 224),
    CRect (128, 128, 224, 224),
    CRect (240, 128, 336, 224),
    CRect ( 16, 240, 112, 336),
    CRect (128, 240, 224, 336),
    CRect (240, 240, 336, 336)
```

```

};

CMainWindow::CMainWindow()
{
    m_nNextChar = EX;
    ::ZeroMemory(m_nGameGrid, 9 * sizeof(int));

    //
    // Register a WNDCLASS.
    //
    CString strWndClass = AfxRegisterWndClass(
        CS_DBLCLKS, // Class style
        AfxGetApp() -> LoadStandardCursor(IDC_ARROW), // Class cursor
        (HBRUSH)(COLOR_3DFACE + 1), // Background brush
        AfxGetApp() -> LoadStandardIcon(IDI_WINLOGO) // Class icon
    );

    //
    // Create a window.
    //
    CreateEx(0, strWndClass, _T("Tic-Tac-Toe"),
        WS_OVERLAPPED|WS_SYSMENU|WS_CAPTION|WS_MINIMIZEBOX,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL);

    //
    // Size the window.
    //
    CRect rect(0, 0, 352, 352);
    CalcWindowRect(&rect);
    SetWindowPos(NULL, 0, 0, rect.Width(), rect.Height(),
        SWP_NOZORDER|SWP_NOMOVE|SWP_NOREDRAW);
}

void CMainWindow::PostNcDestroy()
{
    delete this;
}

void CMainWindow::OnPaint()
{
    CPaintDC dc(this);
    DrawBoard(&dc);
}

void CMainWindow::OnLButtonDown(UINT nFlags, CPoint point)
{
    //
    // Do nothing if it's O's turn, if the click occurred outside the

```

```

        // tic-tac-toe grid, or if a nonempty square was clicked.
        //
        if (m_nNextChar != EX)
            return;

        int nPos = GetRectID (point);
        if ((nPos == -1) || (m_nGameGrid[nPos] != 0))
            return;

        //
        // Add an X to the game grid and toggle m_nNextChar.
        //
        m_nGameGrid[nPos] = EX;
        m_nNextChar = OH;

        //
        // Draw an X on the screen and see if either player has won.
        //
        CClientDC dc (this);
        DrawX (&dc, nPos);
        CheckForGameOver ();
    }

void CMainWindow::OnRButtonDown (UINT nFlags, CPoint point)
{
    //
    // Do nothing if it's X's turn, if the click occurred outside the
    // tic-tac-toe grid, or if a nonempty square was clicked.
    //
    if (m_nNextChar != OH)
        return;

    int nPos = GetRectID (point);
    if ((nPos == -1) || (m_nGameGrid[nPos] != 0))
        return;

    //
    // Add an O to the game grid and toggle m_nNextChar.
    //
    m_nGameGrid[nPos] = OH;
    m_nNextChar = EX;

    //
    // Draw an O on the screen and see if either player has won.
    //
    CClientDC dc (this);
    DrawO (&dc, nPos);
    CheckForGameOver ();
}

```

```

void CMainWindow::OnLButtonDblClk (UINT nFlags, CPoint point)
{
    //
    // Reset the game if one of the thick black lines defining the game
    // grid is double-clicked with the left mouse button.
    //
    CClientDC dc (this);
    if (dc.GetPixel (point) == RGB (0, 0, 0))
        ResetGame ();
}

int CMainWindow::GetRectID (CPoint point)
{
    //
    // Hit-test each of the grid's nine squares and return a rectangle ID
    // (0-8) if (point.x, point.y) lies inside a square.
    //
    for (int i = 0; i < 9; i++) {
        if (m_rcSquares[i].PtInRect (point))
            return i;
    }
    return -1;
}

void CMainWindow::DrawBoard (CDC * pDC)
{
    //
    // Draw the lines that define the tic-tac-toe grid.
    //
    CPen pen (PS_SOLID, 16, RGB (0, 0, 0));
    CPen * pOldPen = pDC->SelectObject (&pen);

    pDC->MoveTo (120, 16);
    pDC->LineTo (120, 336);

    pDC->MoveTo (232, 16);
    pDC->LineTo (232, 336);

    pDC->MoveTo (16, 120);
    pDC->LineTo (336, 120);

    pDC->MoveTo (16, 232);
    pDC->LineTo (336, 232);

    //
    // Draw the Xs and Os.
    //
    for (int i = 0; i < 9; i++) {
        if (m_nGameGrid[i] == EX)

```

```

        DrawX(pDC, i);
    else if (m_nGameGrid[i] == OH)
        DrawO(pDC, i);
    }
    pDC->SelectObject(pOldPen);
}

void CMainWindow::DrawX(CDC * pDC, int nPos)
{
    CPen pen(PS_SOLID, 16, RGB(255, 0, 0));
    CPen * pOldPen = pDC->SelectObject(&pen);

    CRect rect = m_rcSquares[nPos];
    rect.DeflateRect(16, 16);
    pDC->MoveTo(rect.left, rect.top);
    pDC->LineTo(rect.right, rect.bottom);
    pDC->MoveTo(rect.left, rect.bottom);
    pDC->LineTo(rect.right, rect.top);

    pDC->SelectObject(pOldPen);
}

void CMainWindow::DrawO(CDC * pDC, int nPos)
{
    CPen pen(PS_SOLID, 16, RGB(0, 0, 255));
    CPen * pOldPen = pDC->SelectObject(&pen);
    pDC->SelectStockObject(NULL_BRUSH);

    CRect rect = m_rcSquares[nPos];
    rect.DeflateRect(16, 16);
    pDC->Ellipse(rect);

    pDC->SelectObject(pOldPen);
}

void CMainWindow::CheckForGameOver()
{
    int nWinner;

    //
    // If the grid contains three consecutive Xs or Os, declare a winner
    // and start a new game.
    //
    if (nWinner = IsWinner()) {
        CString string = (nWinner == EX) ?
            _T("X wins!") : _T("O wins!");
        MessageBox(string, _T("Game Over"), MB_ICONEXCLAMATION | MB_OK);
        ResetGame();
    }
}

```

```

//
// If the grid is full, declare a draw and start a new game.
//
else if (IsDraw ()) {
    MessageBox (_T ("It's a draw!"), _T ("Game Over"),
        MB_ICONEXCLAMATION | MB_OK);
    ResetGame ();
}

int CMainWindow::IsWinner ()
{
    static int nPattern[8][3] = {
        0, 1, 2,
        3, 4, 5,
        6, 7, 8,
        0, 3, 6,
        1, 4, 7,
        2, 5, 8,
        0, 4, 8,
        2, 4, 6
    };

    for (int i = 0; i < 8; i++) {
        if ((m_nGameGrid[nPattern[i][0]] == EX) &&
            (m_nGameGrid[nPattern[i][1]] == EX) &&
            (m_nGameGrid[nPattern[i][2]] == EX))
            return EX;

        if ((m_nGameGrid[nPattern[i][0]] == OH) &&
            (m_nGameGrid[nPattern[i][1]] == OH) &&
            (m_nGameGrid[nPattern[i][2]] == OH))
            return OH;
    }

    return 0;
}

BOOL CMainWindow::IsDraw ()
{
    for (int i = 0; i < 9; i++) {
        if (m_nGameGrid[i] == 0)
            return FALSE;
    }

    return TRUE;
}

void CMainWindow::ResetGame ()
{

```



```

m_nNextChar = EX;
::ZeroMemory(m_nGameGrid, 9 * sizeof(int));
Invalidate();
:

```

图 3-2 TicTac 应用程序

处理鼠标输入的第一步就是在消息映射表中为想要处理的消息添加输入项。在 TicTac.cpp 中, CMainWindow 的消息映射包含下列消息映射表输入项:

```

ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONDOWNBLCLK()
ON_WM_RBUTTONDOWN()

```

这 3 个语句把 WM_LBUTTONDOWN、WM_LBUTTONDOWNBLCLK 以及 WM_RBUTTONDOWN 消息与 CMainWindow 的成员函数 OnLButtonDown、OnLButtonDownBlcK 以及 OnRButtonDown 联系在一起。消息一到达, 游戏就开始了。

OnLButtonDown 处理程序处理 CMainWindow 客户区中的鼠标左键单击事件。在核实 m_nNextChar 是 X 而不是 O 之后(如果不是 X, 则什么也不执行就返回), OnLButtonDown 将调用保护成员函数 GetRectID 来判断是否单击发生在与井字游戏网格相对应的 9 个矩形中。矩形坐标保存在叫做 CMainWindow::m_rcSquares 的 CRect 对象的静态数组中。GetRectID 使用一个 for 循环来判断由消息处理程序传递来的光标位置是否处于任何一个网格之内:

```

for (int i = 0; i < 9; i++) {
    if (m_rcSquares[i].PtInRect(point))
        return i;
}
return -1;

```

如果传递来的点位于 CRect 对象代表的矩形内, 则 CRect::PtInRect 将返回一个非零值, 否则返回 0。如果对于任一个 m_rcSquares 数组中的矩形 PtInRect 返回非零值, 则 GetRectID 将返回矩形的 ID。ID 是一个从 0 到 8 的整数, 0 代表棋盘的左上角网格, 1 代表它右边的网格, 2 代表右上角的网格, 3 代表下一行最左端的网格, 等等。每个网格在 m_nGameGrid 数组中都有一个相应的元素, 初始值为零代表空格。如果所有对 PtInRect 的调用都没返回 TRUE, 则 GetRectID 将返回 -1 说明单击发生在网格之外, OnLButtonDown 将忽略鼠标单击事件。但是, 如果 GetRectID 返回一个有效的 ID 并且相应的网格是空的, 则 OnLButtonDown 就会在 m_nGameGrid 数组中记录 X 并调用 CMainWindow::DrawX 函数在网格中绘一个 X。DrawX 创建一个红色的 16 像素宽的笔并在 45 度方向绘制两条相互垂直的线。

OnRButtonDown 与 OnLButtonDown 的功能基本相同, 只不过它绘制一个 O 而不是 X。执行绘图的例程是 CMainWindow::DrawO 函数。在它调用 CDC::Ellipse 函数之前, DrawO 将一

个 NULL 画刷选入了设备描述表:

```
pDC->SelectStockObject(NULL_BRUSH);
```

这就防止了 O 的内部被设备描述表的默认白色画刷填充。(另外,还可以创建一个颜色与窗口背景色匹配的画刷并将其选进设备描述表中。但是由于不产生实际的屏幕输出,用一个 NULL 画刷绘图会稍微快些。)然后用如下语句绘出 O:

```
CRect rect = m_rcSquares[nPos];
rect.DeflateRect(16, 16);
pDC->Ellipse(rect);
```

第一个语句将代表网格的矩形拷贝到名为 rect 的本地 CRect 对象;第二句使用 CRect::DeflateRect 在每个方向上都把矩形“缩小”到 16 像素,形成圆的边框;第三句画圆。输出结果就是居于网格中心的一个漂亮的 O 形。

双击分隔网格的网格线会清除 X 及 O,开始新的一局。无可否认这样设计用户接口不是一个好方法,但它却提供了编写双击处理程序的一个机会。(更好的方法是添加一个“重新开局”按钮或“重新开局”菜单项,但是由于我们还没有涉及到菜单和控件,所以不得不等一段时间才能制作出完美的用户接口。)双击鼠标左键的事件由 CMainWindow::OnLButtonDblClk 处理,其中包含如下简单的语句:

```
CClientDC dc(this);
if(dc.GetPixel(point) == RGB(0, 0, 0))
    ResetGame();
```

为了判断双击是否发生在用来分隔棋盘网格的黑色粗线上,OnLButtonDblClk 将调用 CDC::GetPixel 来获得光标下像素的颜色,把它与黑色(RGB(0,0,0))做比较。如果匹配,则 ResetGame 将被调用来重新设置游戏。否则,OnLButtonDblClk 返回,双击被忽略。对于快速检测不规则的形状区域,检测光标下像素的颜色是一种有效的技术,但是要慎重选用那些非基本色,显示驱动程序可能会使这些颜色发生抖动。因为所有运行 Windows 的 PC 机都支持纯黑(RGB(0,0,0))和纯白(RGB(255,255,255))色,所以可以断定这些颜色不会抖动。

要与已有的用户接口指南取得一致,应用程序就不应该像 TicTac 那样,使用鼠标右键执行应用程序中特定的任务。相反,应该弹出上下文菜单来响应鼠标右键单击。在 WM_RBUTTONDOWN 消息被传递给系统进行默认处理之后,Windows 将把一个 WM_CONTEXTMENU 消息放在消息队列中。下一章您将学到更多与此有关的操作系统的特性。

消息框

在返回之前,TicTac 的 OnLButtonDown 和 OnRButtonDown 处理程序将调用 CMainWindow

∴ CheckGameOver 来找出游戏的获胜方或判断是否是和局。如果选手的任一方将 3 个 X 或 O 摆成了一排,或者已没有剩余的空格,则 CheckGameOver 将调用 CMainWindow 的 MessageBox 函数来显示一个消息框宣布结果,如图 3-3 所示。MessageBox 是所有窗口类从 CWnd 继承来的函数。因为它为在屏幕上显示一个消息并任意地获得一个响应提供了一种一步到位的方法,所以在程序处理中它是一个极其有用的工具。



图 3-3 Windows 消息框

CWnd::MessageBox 的原型如下:

```
int MessageBox (LPCTSTR lpszText, LPCTSTR lpszCaption = NULL,
                UINT nType = MB_OK)
```

lpszText 指定了消息框中正文的文本,lpszCaption 指定了消息框标题栏中的标题,nType 包含着一个或多个位标志,定义了消息框的样式。返回值标识了被单击后释放消息框的按钮。lpszText 和 lpszCaption 可以是指向常规文本字符串的指针或 CString 对象。(由于 CString 类重载了 LPCTSTR 运算符,因此总可以把一个 CString 传递给接受 LPCTSTR 数据类型的函数。) lpszCaption 值为 NULL 将在标题栏显示标题“Error”。

MessageBox 最简单的用处就是显示一个消息并等待用户单击消息框中的“OK”按钮:

```
MessageBox (_T ("Click OK to continue"),_T ("My Application"));
```

nType 接受默认值(MB_OK)意味着消息框将只有一个“OK”按钮而没有其他按钮。因此,唯一可能的返回值就是 IDOK。但是如果在退出应用程序之前,您想要使用消息框询问用户是否要保存一个文件,您可以使用 MB_YESNOCANCEL 类型:

```
MessageBox (_T ("Your document contains unsaved data. Save it?"),
            _T ("My Application"), MB_YESNOCANCEL);
```

现在消息框将包含 3 个按钮: Yes、No 以及 Cancel,从 MessageBox 函数返回的值为 IDYES、IDNO 或 IDCANCEL。然后,程序将检测返回值。如果是 IDYES,则在结束之前保存数据;如果是 IDNO,则不保存;如果是 IDCANCEL,则返回应用程序。在表 3-4 中列出了 6 个消息框样式以及对应的返回值,默认按钮用黑体字突出显示,默认按钮就是用户按回车键点中的按钮。

表 3-4 消息框类型

| 类型 | 按钮 | 可能返回的代码 |
|---------------------|----------------------|----------------------------|
| MB_ABORTRETRYIGNORE | Abort, Retry, Ignore | IDABORT, IDRETRY, IDIGNORE |
| MB_OK | OK | IDOK |
| MB_OKCANCEL | OK, Cancel | IDOK, IDCANCEL |
| MB_RETRYCANCEL | Retry, Cancel | IDRETRY, IDCANCEL |
| MB_YESNO | Yes, No | IDYES, IDNO |
| MB_YESNOCANCEL | Yes, No, Cancel | IDYES, IDNO, IDCANCEL |

在有多个按钮的消息框中,第一个(最左边的)按钮通常是默认按钮。您可以将 MB_DEFBUTTON2 或 MB_DEFBUTTON3 加入表示特定消息框样式的值中,使第二或第三个按钮成为默认按钮。语句

```
MessageBox(_T("Your document contains unsaved data. Save it?"),
    _T("My Application"), MB_YESNOCANCEL | MB_DEFBUTTON3);
```

显示与前一个相同的消息框,但 Cancel 按钮成为了默认按钮。

在默认情况下,消息框处于应用程序模式,就是说调用 MessageBox 函数的应用程序只有在消息框释放后才能结束。您可以把 MB_SYSTEMMODAL 加到 nType 参数中,使消息框处于系统模式。在 16 位 Windows 系统中,系统模式意味着直到消息框被释放,所有应用程序的输入都是被挂起的。在 Win32 环境中,Windows 让消息框作为最顶层窗口位于其他窗口上,但是用户仍然可以自由地切换到别的应用程序。系统模式消息框只应该被用在出现了要求立即引起注意的严重错误的情况下。

您可以通过使用 MB_ICON 标识符给消息框添加一些有趣的东西。MB_ICONINFORMATION 在消息框的左上角显示一个带有“i”字的小汽球,其中“i”代表“信息”。通常在给用户提供信息且无问题提出时使用“i”,如:

```
MessageBox(_T("No errors found. Click OK to continue"),
    _T("My Application"), MB_ICONINFORMATION | MB_OK);
```

MB_ICONQUESTION 显示一个问号来替代“i”,通常在查询如“在退出前保存吗?”这样的问题时使用。MB_ICONSTOP 显示一个带有 X 的红色圆圈,通常说明有不可恢复的错误发生,如:内存溢出错误使程序提前结束。最后,MB_ICONEXCLAMATION 显示一个包含感叹号的黄色三角形(参见图 3-3)。

MFC 以全局 AfxMessageBox 函数的形式为 CWnd::MessageBox 提供了一个可选对象,虽然两者很相似,但 AfxMessageBox 可以从应用程序类、文档类,以及别的非窗口类中调用。