第三章　视窗和讯息

在前两章，程式使用了同一个函式 MessageBox 来向使用者输出文字。MessageBox 函式会建立一个「视窗」。在 Windows 中，「视窗」一词有确切的含义。一个视窗就是萤幕上的一个矩形区域，它接收使用者的输入并以文字或图形的格式显示输出内容。

MessageBox 函式建立一个视窗，但这只是一个功能有限的特殊视窗。讯息视窗有一个带关闭按钮的标题列、一个选项图示、一行或多行文字，以及最多四个按钮。当然，必须选择 Windows 提供给您的图示与按钮。

MessageBox 函式非常有用，但下面不会过多地使用它。我们不能在讯息方块中显示图形，而且也不能在讯息方块中添加功能表。要添加这些物件，就需要建立自己的视窗，现在就开始。

自己的视窗

建立视窗很简单，只需呼叫 CreateWindow 函式即可。

好啦，虽然建立视窗的函式的确名为 CreateWindow，而且您也能在 /Platform SDK/User Interface Services/Windowing/Windows/Window Reference/Window Functions 找到此文件，但您将发现 CreateWindow 的第一个参数就是所谓的「视窗类别名称」，并且该视窗类别连接所谓的「视窗讯息处理程式」。在我们呼叫 CreateWindow 之前，有一点背景知识会对您大有帮助。

总体结构

进行 Windows 程式设计，实际上是在进行一种物件导向的程式设计（OOP）。这一点在 Windows 中使用得最多的物件上表现最为明显。这种物件正是 Windows 之所以命名为「Windows」的原因，它具有人格化的特徵，甚至可能会在您的梦中出现，这就是那个叫做「视窗」的东西。

桌面上最明显的视窗就是应用程式视窗。这些视窗含有显示程式名称的标题列、功能表甚至可能还有工具列和卷动列。另一类视窗是对话方块，它可以有标题列也可以没有标题列。

装饰对话方块表面的还有各式各样的按键、单选按钮、核取方块、清单方块、卷动列和文字输入区域。其中每一个小的视觉物件都是一个视窗。更确切地说，这些都称为「子视窗」或「控制项视窗」或「子视窗控制项」。

作为物件，使用者会在萤幕上看到这些视窗，并通过键盘和滑鼠直接与它

们进行交互操作。更有趣的是，程式写作者的观点与使用者的观点极其类似。视窗以「讯息」的形式接收视窗的输入，视窗也用讯息与其他视窗通讯。对讯息的理解将是学习如何写作 Windows 程式所必须越过的障碍之一。

这有一个 Windows 的讯息范例：我们知道，大多数的 Windows 程式都有大小合适的应用程式视窗。也就是说，您能够通过滑鼠拖动视窗的边框来改变视窗的大小。通常，程式将通过改变视窗中的内容来回应这种大小的变化。您可能会猜测（并且您也是正确的），是 Windows 本身而不是应用程式在处理与使用者重新调整视窗大小相关的全部杂乱程式。由於应用程式能改变其显示的样子，所以它也「知道」视窗大小改变了。

应用程式是如何知道使用者改变了视窗的大小的呢？由於程式写作者习惯了往常的文字模式程式，作业系统没有设置将此类讯息通知给使用者的机制。问题的关键在於理解 Windows 所使用的架构。当使用者改变视窗的大小时，Window 给程式发送一个讯息指出新视窗的大小。然後程式就可以调整视窗中的内容，以回应大小的变化。

「Windows 给程式发送讯息。」我们希望读者不要对这句话视而不见。它到底表达了什么意思呢？我们在这里讨论的是程式码，而不是一个电子邮件系统。作业系统怎么给程式发送讯息呢？

其实，所谓「Windows 给程式发送讯息」，是指 Windows 呼叫程式中的一个函式，该函式的参数描述了这个特定讯息。这种位於 Windows 程式中的函式称为「视窗讯息处理程式」。

无疑，读者对程式呼叫作业系统的做法是很熟悉的。例如，程式在打开磁片档案时就要使用有关的系统呼叫。读者所不习惯的，可能是作业系统呼叫程式，而这正是 Windows 物件导向架构的基础。

程式建立的每一个视窗都有相关的视窗讯息处理程式。这个视窗讯息处理程式是一个函式，既可以在程式中，也可以在动态连结程式库中。Windows 通过呼叫视窗讯息处理程式来给视窗发送讯息。视窗讯息处理程式根据此讯息进行处理，然後将控制传回给 Windows。

更确切地说，视窗通常是在「视窗类别」的基础上建立的。视窗类别标识了处理视窗讯息的视窗讯息处理程式。使用视窗类别使多个视窗能够属於同一个视窗类别，并使用同一个视窗讯息处理程式。例如，所有 Windows 程式中的所有按钮均依据同一个视窗类别。这个视窗类别与一个处理所有按钮讯息的视窗讯息处理程式（位於 Windows 的动态连结程式库中）联结。

在物件导向的程式设计中，物件是程式与资料的组合。视窗是一种物件，其程式是视窗讯息处理程式。资料是视窗讯息处理程式保存的资讯和 Windows

为每个视窗以及系统中那个视窗类别保存的资讯。

视窗讯息处理程式处理给视窗发送讯息。这些讯息经常是告知视窗，使用者正使用键盘或者滑鼠进行输入。这正是按键视窗知道它被「按下」的奥妙所在。在视窗大小改变，或者视窗表面需要重画时，由其他讯息通知视窗。

Windows 程式开始执行後，Windows 为该程式建立一个「讯息伫列」。这个讯息伫列用来存放该程式可能建立的各种不同视窗的讯息。程式中有一小段程式码，叫做「讯息回圈」，用来从伫列中取出讯息，并且将它们发送给相应的视窗讯息处理程式。有些讯息直接发送给视窗讯息处理程式，不用放入讯息伫列中。

如果您对这段 Windows 架构过於简略的描述将信将疑，就让我们去看看在实际的程式中，视窗、视窗类别、视窗讯息处理程式、讯息伫列、讯息回圈和视窗讯息是如何相互配合的。这或许会对您有些帮助。

HELLOWIN 程式

建立一个视窗首先需要注册一个视窗类别，那需要一个视窗讯息处理程式来处理视窗讯息。处理视窗讯息对每个 Windows 程式都带来了些负担。程式 3-1 所示的 HELLOWIN 程式中整个做的事情差不多就是料理这些事情。

程式 3-1　HELLOWIN

```
HELLOWIN.C
/*------------------------------------------------------------------------
      HELLOWIN.C -- Displays "Hello, Windows 98!" in client area
                  (c) Charles Petzold, 1998
 ------------------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR szAppName[] = TEXT ("HelloWin") ;
      HWND  hwnd ;
      MSG   msg ;
      WNDCLAS     wndclass ;

      wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc  = WndProc ;
      wndclass.cbClsExtra     = 0 ;
      wndclass.cbWndExtra     = 0 ;
      wndclass.hInstance      = hInstance ;
```

```
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor   = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuNam   = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
    {
          MessageBox (     NULL, TEXT ("This program requires Windows NT!"),
                           szAppName, MB_ICONERROR) ;
          return 0 ;
    }
     hwnd = CreateWindow( szAppName,    // window class name
               TEXT ("The Hello Program"),  // window caption
               WS_OVERLAPPEDWINDOW,   // window style
               CW_USEDEFAULT,    // initial x position
               CW_USEDEFAULT,    // initial y position
               CW_USEDEFAULT,    // initial x size
               CW_USEDEFAULT,    // initial y size
               NULL,             // parent window handle
            NULL,          // window menu handle
            hInstance,        // program instance handle
            NULL) ;     // creation parameters

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
    {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
    }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     HDC              hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (message)
    {
     case WM_CREATE:
     PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC) ;
          return 0 ;

     case  WM_PAINT:
```

```
            hdc = BeginPaint (hwnd, &ps) ;

            GetClientRect (hwnd, &rect) ;

            DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
                  DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
                  EndPaint (hwnd, &ps) ;
                  return 0 ;

      case  WM_DESTROY:
            PostQuitMessage (0) ;
            return 0 ;
    }
  return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

程式建立一个普通的应用程式视窗，如图 3-1 所示。在视窗显示区域的中央显示「Hello, Windows 98!」。如果安装了音效卡，那么您还可以听到相应的朗读声音。

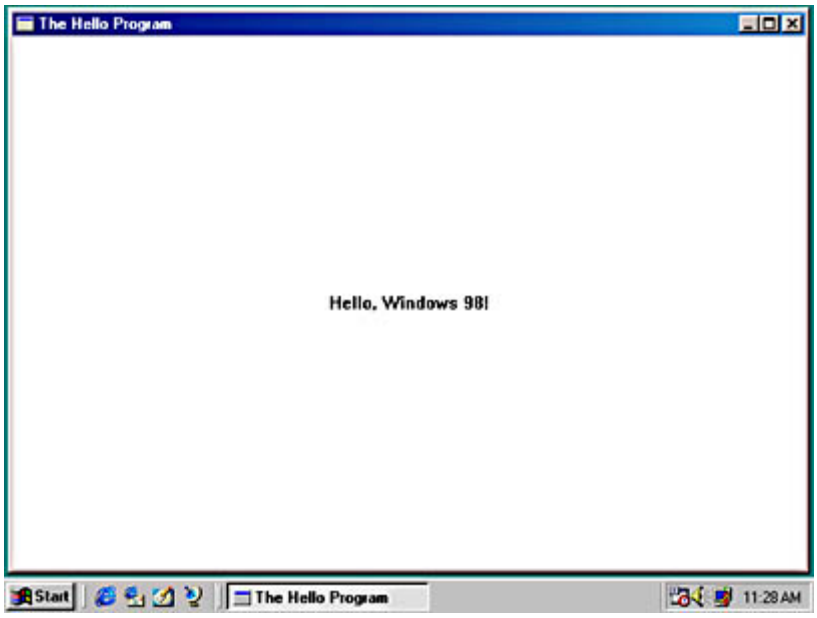

图 3-1 HELLOWIN 视窗

提醒您注意：如果您使用 Microsoft Visual C++ 为此程式建立新专案，那么您得加上连结程式所需的程式库档案。从 Project 功能表选择 Setting 选项，然後选取 Link 页面标签。从 Category 清单方块中选择 General ，然後在 Object/Library Modules 文字方块添加 WINMM.LIB （ Windows multimedia —— Windows 多媒体 ）。您这样做是因为 HELLOWIN 将使用多媒体功能呼叫，而内定的专案中又不包括多媒体程式库档案。不然连结程式报告了错误资讯，表明 PlaySound 函式不可用。

HELLOWIN 将存取档案 HELLOWIN.WAV，该档案在本书所附光碟的 HELLOWIN 目录中。执行 HELLOWIN.EXE 时，内定的目录必须是 HELLOWIN。在 Visual C++

中执行此程式时，虽然执行档会产生在 HELLOWIN 的 RELEASE 或 DEBUG 子目录中，但执行程式的目录还是必须在 HELLOWIN 中。

通盘考量

实际上，每一个 Windows 程式码中都包括 HELLOWIN.C 程式的大部分。没人能真正记住此程式的全部写法；通常，Windows 程式写作者在开始写一个新程式时总是会复制一个现有的程式，然後再做相应的修改。您可以按此习惯自由使用本书附带光碟中的程式。

上面提到，HELLOWIN 将在其视窗的中央显示字串。这种说法不是完全正确的。文字实际显示在程式显示区域的中央，它在图 3-1 中是标题列和边界范围内的大片白色区域。这区别对我们来说很重要；显示区域就是程式自由绘图并且向使用者显示输出结果的视窗区域。

如果您认真思考一下，将会发现虽然只有 80 行程式码，这个视窗却令人惊讶地具有许多功能。您可以用滑鼠按住标题列，在萤幕上移动视窗；可以按住大小边框，改变视窗的大小。在视窗大小改变时，程式自动地将「Hello, Windows 98!」字串重新定位在显示区域的中央。您可以按最大化按钮，放大 HELLOWIN 以充满整个萤幕；也可以按最小化按钮，将程式缩小成一个图示。您可以在系统功能表中执行所有选项（就是按下在标题列最左端的小图示）；也可以从系统功能表中选择 Close 选项，或者单击标题列最右端的关闭按钮，或者双击标题列最左端的图示，来关闭视窗以终止程式的执行。

我们将在本章的余下部分对此程式作一详细的检查。当然，我们首先要从整体上看一下。

与前两章中的范例程式一样，HELLOWIN.C 也有一个 WinMain 函式，但它还有另外一个函式，名为 WndProc。这就是视窗讯息处理程式。注意，在 HELLOWIN.C 中没有呼叫 WndProc 的程式码。当然，在 WinMain 中有对 WndProc 的参考，而这就是该函式要在程式开头附近宣告的原因。

Windows 函式呼叫

HELLOWIN 至少呼叫了 18 个 Windows 函式。下面以它们在 HELLOWIN 中出现的次序列出这些函式以及各自的简明描述：

LoadIcon	载入图示供程式使用。
LoadCursor	载入滑鼠游标供程式使用。
GetStockObject	取得一个图形物件（在这个例子中，是取得绘制视窗背景的画刷物件）。

RegisterClass	为程式视窗注册视窗类别。
MessageBox	显示讯息方块。
CreateWindow	根据视窗类别建立一个视窗。
ShowWindow	在萤幕上显示视窗。
UpdateWindow	指示视窗自我更新。
GetMessage	从讯息伫列中取得讯息。
TranslateMessage	转译某些键盘讯息。
DispatchMessage	将讯息发送给视窗讯息处理程式。
PlaySound	播放一个音效档案。
BeginPaint	开始绘制视窗。
GetClientRect	取得视窗显示区域的大小。
DrawText	显示字串。
EndPaint	结束绘制视窗。
PostQuitMessage	在讯息伫列中插入一个「退出程式」讯息。
DefWindowProc	执行内定的讯息处理。

这些函式均在 Platform SDK 文件中说明，并在不同的表头档案中宣告，其中绝大多数宣告在 WINUSER.H 中。

大写字母识别字

读者可能注意到，HELLOWIN.C 中有几个大写的识别字，这些识别字是在 Windows 表头档案中定义的。有些识别字含有两个字母或者三个字母的字首，这些字首後头接著一个底线：

CS_HREDRAW	DT_VCENTER	SND_FILENAME
CS_VREDRAW	IDC_ARROW	WM_CREATE
CW_USEDEFAULT	IDI_APPLICATION	WM_DESTROY
DT_CENTER	MB_ICONERROR	WM_PAINT
DT_SINGLELINE	SND_ASYNC	WS_OVERLAPPEDWINDOW

这些是简单的数值常数。字首指示该常数所属的类别，如表 3-1 所示。

表 3-1

字首	**类别**
CS	视窗类别样式

CW	建立视窗
DT	绘制文字
IDI	图示 ID
IDC	游标 ID
MB	讯息方块
SND	声音
WM	视窗讯息
WS	视窗样式

奉劝程式写作者不要费力气去记忆 Windows 程式设计中的数值常数。实际上，Windows 中使用的每个数值常数在表头档案中均有相应的识别字定义。

新的资料型态

HELLOWIN.C 中的其他识别字是新的资料型态，也在 Windows 表头档案中使用 typedef 叙述或者#define 叙述加以定义了。最初是为了便於将 Windows 程式从原来的 16 位元系统上移植到未来的使用 32 位元(或者其他)技术的作业系统上。这种作法并不如当时每个人想像的那样顺利，但是这种概念基本上是正确的。

有时这些新的资料型态只是为了方便缩写。例如，用於 WndProc 的第二个参数的 UINT 资料型态只是一个 unsigned int （无正负号整数），在 Windows 98 中，这是一个 32 位元的值。用於 WinMain 的第三个参数的 PSTR 资料型态是指向一个字串的指标，即是一个 char *。

其他资料型态的含义不太明显。例如，WndProc 的第三和第四个参数分别被定义为 WPARAM 和 LPARAM，这些名字的来源有点历史背景：当 Windows 还是 16 位元系统时，WndProc 的第三个参数被定义为一个 WORD，这是一个 16 位元的 无正负号短 （unsigned short）整数，而第四个参数被定义为一个 LONG，这是一个 32 位元有正负号长整数，从而导致了文字「PARAM」前面加上了前置字首「W」和「L」。当然，在 32 位元的 Windows 中，WPARAM 被定义为一个 UINT，而 LPARAM 被定义为一个 LONG（这就是 C 中的 long 整数型态），因此视窗讯息处理程式的这两个参数都是 32 位元的值。这也许有点奇怪，因为 WORD 资料型态在 Windows98 中仍然被定义为一种 16 位元的 无正负号 整数，因此「PARAM」前的「W」就有点误用了。

WndProc 函式传回一个型态为 LRESULT 的值，该值简单地被定义为一个 LONG。WinMain 函式被指定了一个 WINAPI 型态（在表头档案中定义的所有

Windows 函式都被指定这种型态），而 WndProc 函式被指定一个 CALLBACK 型态。这两个识别字都被定义为_stdcall，表示在 Windows 本身和使用者的应用程式之间发生的函式呼叫的呼叫参数传递方式。

HELLOWIN 还使用了 Windows 表头档案中定义的四种资料结构（我们将在本章稍後加以讨论）。这些资料结构如表 3-2 所示。

表 3-2

结构	含义
MSG	讯息结构
WNDCLASS	视窗类别结构
PAINTSTRUCT	绘图结构
RECT	矩形结构

前面两个资料结构在 WinMain 中使用，分别定义了两个名为 msg 和 wndclass 的结构，後面两个资料结构在 WndProc 中使用，分别定义了 ps 和 rect 结构。

代号简介

最後，还有三个大写识别字（见表 3-3），用於不同型态的「代号」：

表 3-3

识别字	含义
HINSTANCE	执行实体（程式自身）代号
HWND	视窗代号
HDC	装置内容代号

代号在 Windows 中使用非常频繁。在本章结束之前，我们将遇到 HICON（图示代号）、HCURSOR（滑鼠游标代号）和 HBRUSH（画刷代号）。

代号是一个（通常为 32 位元的）整数，它代表一个物件。Windows 中的代号类似传统 C 或者 MS-DOS 程式设计中使用的档案代号。程式几乎总是通过呼叫 Windows 函式取得代号。程式在其他 Windows 函式中使用这个代号，以使用它代表的物件。代号的实际值对程式来说是无关紧要的。但是，向您的程式提供代号的 Windows 模组知道如何利用它来使用相对应的物件。

匈牙利表示法

读者可能注意到，HELLOWIN.C 中有一些变数的名字显得很古怪。如 szCmdLine，它是传递给 WinMain 的参数。

许多 Windows 程式写作者使用一种叫做「匈牙利表示法」的变数命名通则。这是为了纪念传奇性的 Microsoft 程式写作者 Charles Simonyi。非常简单，变数名以一个或者多个小写字母开始，这些字母表示变数的资料型态。例如，szCmdLine 中的 sz 代表「以 0 结尾的字串」。在 hInstance 和 hPrevInstance 中的 h 字首表示「代号」；在 iCmdShow 中的 i 字首表示「整数」。 WndProc 的後两个参数也使用匈牙利表示法。正如我在前面已经解释过的，尽管 wParam 应该更适当地被命名为 uiParam（代表「无正负号整数」），但是因为这两个参数是使用资料型态 WPARAM 和 LPARAM 定义的，因此保留它们传统的名字。

在命名结构变数时，可以用结构名（或者结构名的一种缩写）的小写作为变数名的字首，或者用作整个变数名。例如，在 HELLOWIN. C 的 WinMain 函式中，msg 变数是 MSG 型态的结构；wndclass 是 WNDCLASSEX 型态的一个结构。在 WndPmc 函式中，ps 是一个 PAINTSTRUCT 结构，rect 是一个 RECT 结构。

匈牙利表示法能够帮助程式写作者及早发现并避免程式中的错误。由於变数名既描述了变数的作用，又描述了其资料型态，就比较容易避免产生资料型态不合的错误。

表 3-4 列出了在本书中经常用到的变数字首。

表 3-4

字首	**资料型态**
c	char 或 WCHAR 或 TCHAR
by	BYTE （无正负号字元）
n	short
i	int
x, y	int 分别用作 x 座标和 y 座标
cx, cy	int 分别用作 x 长度和 y 长度；C 代表「计数器」
b 或 f	BOOL (int)；f 代表「旗标」
w	WORD （无正负号短整数）
l	LONG （长整数）
dw	DWORD （无正负号长整数）
fn	function（函式）
s	string（字串）
sz	以位元组值 0 结尾的字串
h	代号
p	指标

注册视窗类别

视窗依照某一视窗类别建立，视窗类别用以标识处理视窗讯息的视窗讯息处理程式。

不同视窗可以依照同一种视窗类别建立。例如，Windows 中的所有按钮视窗——包括按键、核取方块，以及单选按钮——都是依据同一种视窗类别建立的。视窗类别定义了视窗讯息处理程式和依据此类别建立的视窗的其他特徵。在建立视窗时，要定义一些该视窗所独有的特徵。

在为程式建立视窗之前，必须首先呼叫 RegisterClass 注册一个视窗类别。该函式只需要一个参数，即一个指向型态为 WNDCLASS 的结构指标。此结构包括两个指向字串的栏位，因此结构在 WINUSER.H 表头档案中定义了两种不同的方式，第一个是 ASCII 版的 WNDCLASSA：

```
typedef struct tagWNDCLASSA
{
      UINT      style ;
      WNDPROC         lpfnWndProc ;
      int       cbClsExtra ;
      int          cbWndExtra ;
      HINSTANCE   hInstance ;
      HICON          hIcon ;
      HCURSOR     hCursor ;
      HBRUSH          hbrBackground ;
      LPCSTR      lpszMenuName ;
      LPCSTR      lpszClassName ;
}
WNDCLASSA, * PWNDCLASSA, NEAR * NPWNDCLASSA, FAR * LPWNDCLASSA ;
```

在这里提示一下资料型态和匈牙利表示法：其中的 lpfn 字首代表「指向函式的长指标」。（在 Win32 API 中，长指标和短指标（或者近程指标）没有区别。这只是 16 位元 Windows 的遗物。）cb 字首代表「位元组数」而且通常作为一个常数来表示一个位元组的大小。h 字首是一个代号，而 hbr 字首代表「一个画刷的代号」。lpsz 字首代表「指向以 0 结尾字串的指标」。

Unicode 版的结构定义如下：

```
typedef struct tagWNDCLASSW
{
      UINT          style ;
      WNDPROC     lpfnWndProc ;
      int         cbClsExtra ;
      int         cbWndExtra ;
      HINSTANCE        hInstance ;
      HICON       hIcon ;
      HCURSOR     hCursor ;
```

```
      HBRUSH      hbrBackground ;
      LPCWSTR     lpszMenuName ;
      LPCWSTR     lpszClassName ;
}
WNDCLASSW, * PWNDCLASSW, NEAR * NPWNDCLASSW, FAR * LPWNDCLASSW ;
```

与前者唯一的区别在於最後两个栏位定义为指向宽字串常数，而不是指向 ASCII 字串常数。

WINUSER.H 定义了 WNDCLASSA 和 WNDCLASSW 结构（以及指向结构的指标）以後，表头档案依据对 UNICODE 识别字的解释，定义了 WNDCLASS 和指向 WNDCLASS 的指标（包括一些向後相容的程式码）：

```
#ifdef UNICODE
typedef     WNDCLASSW      WNDCLASS ;
typedef     PWNDCLASSW     PWNDCLASS ;
typedef     NPWNDCLASSW      NPWNDCLASS ;
typedef     LPWNDCLASSW      LPWNDCLASS ;
#else
typedef     WNDCLASSA     WNDCLASS ;
typedef     PWNDCLASSA    PWNDCLASS ;
typedef     NPWNDCLASSA    NPWNDCLASS ;
typedef     LPWNDCLASSA    LPWNDCLASS ;
#endif
```

本书後面列出结构时，将只列出功用相同的结构定义，对 WNDCLASS 就像这样：

```
typedef struct
{
   UINT          style ;
      WNDPROC            lpfnWndProc ;
      int                cbClsExtra ;
      int             cbWndExtra ;
      HINSTANCE         hInstance ;
      HICON       hIcon ;
      HCURSOR     hCursor ;
      HBRUSH      hbrBackground ;
      LPCTSTR     lpszMenuName ;
      LPCTSTR     lpszClassName ;
}
WNDCLASS, * PWNDCLASS ;
```

我也不再著重说明指标的定义。一个程式写作者的程式不应该因为使用以 LP 或 NP 为字首的不同指标型态而被搅乱。

在 WinMain 中为 WNDCLASS 定义一个结构，通常像这样：

WNDCLASS wndclass ;

然後，你就可以初始化该结构的 10 个栏位，并呼叫 RegisterClass。

在 WNDCLASS 结构中最重要的两个栏位是第二个和最後一个，第二个栏位

(lpfnWndProc) 是依据这个类别来建立的所有视窗所使用的视窗讯息处理程式的位址。在 HELLOWIN.C 中，这个是 WndProc 函式。最後一个栏位是视窗类别的文字名称。程式写作者可以随意定义其名称。在只建立一个视窗的程式中，视窗类别名称通常设定为程式名称。

其他栏位依照下面的方法描述了视窗类别的一些特徵。让我们依次看看 WNDCLASS 结构中的每个栏位。

叙述

```
wndclass.style = CS_HREDRAW | CS_VREDRAW ;
```

使用 C 的位元「或」运算子结合了两个「视窗类别样式」识别字。在表头档案 WINUSER.H 中，已定义了一整组以 CS 为字首的识别字：

```
#define     CS_VREDRAW            0x0001
#define     CS_HREDRAW              0x0002
#define     CS_KEYCVTWINDOW      0x0004
#define     CS_DBLCLKS              0x0008
#define     CS_OWNDC          0x0020
#define     CS_CLASSDC              0x0040
#define     CS_PARENTDC             0x0080
#define     CS_NOKEYCVT             0x0100
#define     CS_NOCLOSE              0x0200
#define     CS_SAVEBITS             0x0800
#define     CS_BYTEALIGNCLIENT        0x1000
#define     CS_BYTEALIGNWINDOW        0x2000
#define     CS_GLOBALCLASS          0x4000
#define     CS_IME                   0x00010000
```

由於每个识别字都可以在一个复合值中设置一个位元的值，所以按这种方式定义的识别字通常称为「位元旗标」。通常我们只使用少数的视窗类别样式。HELLOWIN 中用到的这两个识别字表示，所有依据此类别建立的视窗，每当视窗的水平方向大小(CS_HREDRAW)或者垂直方向大小(CS_VREDRAW)改变之後，视窗要完全重画。改变 HELLOWIN 的视窗大小，可以看到字串仍然显示在视窗的中央，这两个识别字确保了这一点。不久我们就将看到视窗讯息处理程式是如何得知这种视窗大小的变化的。

WNDCLASS 结构的第二个栏位由以下叙述进行初始化：

```
wndclass.lpfnWndProc = WndProc ;
```

这条叙述将这个视窗类别的视窗讯息处理程式设定为 WndProc，即 HELLOWIN.C 中的第二个函式。这个过程将处理依据这个视窗类别建立的所有视窗的全部讯息。在 C 语言中，像这样在结构中使用函式名时，真正提供的是指向函式的指标。

下面两个栏位用於在视窗类别结构和 Windows 内部保存的视窗结构中预留一些额外空间：

```
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
```

程式可以根据需要来使用预留的空间。HELLOWIN 没有使用它们，所以设定值为 0。否则，和匈牙利表示法所指示的一样，这个栏位将被当成「预留的位元组数」。（在第七章的程式 CHECKER3 将使用 cbWndExtra 栏位。）

下一个栏位就是程式的执行实体代号（它也是 WinMain 的参数之一）：

```
wndclass.hInstance = hInstance ;
```

叙述

```
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
```

为所有依据这个视窗类别建立的视窗设置一个图示。图示是一个小的点阵图图像，它对使用者代表程式，将出现在 Windows 工作列中和视窗的标题列的左端。在本书的後面，您将学习如何为您的 Windows 程式自订图示。现在，为了方便起见，我们将使用预先定义的图示。

要取得预先定义图示的代号，可以将第一个参数设定为 NULL 来呼叫 LoadIcon。在载入程式写作者自订的图示时（图示应该存放在磁片上的.EXE 程式档案中），这个参数应该被设定为程式的执行实体代号 hInstance。第二个参数代表图示。对於预先定义图示，此参数是以 IDI 开始的识别字（「ID 代表图示」），识别字在 WINUSER.H 中定义。IDI_APPLICATION 图示是一个简单的视窗小图形。LoadIcon 函式传回该图示的代号。我们并不关心这个代号的实际值，它只用於设置 hIcon 栏位元的值。该栏位在 WNDCLASS 结构中定义为 HICON 型态，此型态名的含义为「handle to an icon（图示代号）」。

叙述

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

与前一条叙述非常相似。LoadCursor 函式载入一个预先定义的滑鼠游标（命名为 IDC_ARROW），并传回该游标的代号。该代号被设定给 WNDCLASS 结构的 hCursor 栏位。当滑鼠游标在依据这个类别建立的视窗的显示区域上出现时，它变成一个小箭头。

下一个栏位指定依据这个类别建立的视窗背景颜色。hbrBackground 栏位名称中的 hbr 字首代表「handle to a brush（画刷代号）」。画刷是个绘图词汇，指用来填充一个区域的著色样式。Windows 有几个标准画刷，也称为「备用(stock)」画刷。这里所示的 GetStockObject 呼叫将传回一个白色画刷的代号：

```
wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
```

这意味著视窗显示区域的背景完全为白色，这是一种极其普遍的做法。

下一个栏位指定视窗类别功能表。HElLOWIN 没有应用程式功能表，所以该栏位被设定为 NULL：

```
wndclass.lpszMenuName = NULL ;
```

最後，必须给出一个类别名称。对於小程式，类别名称可以与程式名相同，即存放在 szAppName 变数中的「HelloWin」字串。

```
wndclass.lpszClassName = szAppName ;
```

至於该字串由 ASCII 字元组成或由 Unicode 字元组成，取决於是否定义了 UNICODE 识别字。

在初始化该结构的 10 个栏位後，HELLOWIN 呼叫 RegisterClass 来注册这个视窗类别。该函式只有一个参数，即指向 WNDCLASS 结构的指标。实际上，RegisterClassA 函式将获得一个指向 WNDCLASSA 结构的指标，而 RegisterClassW 函式将获得一个指向 WNDCLASSW 结构的指标。程式要使用哪个函式来注册视窗类别，取决於发送给视窗的讯息包含 ASCII 文字还是 Unicode 文字。

现在有一个问题：如果用定义的 UNICODE 识别字编译了程式，程式将呼叫 RegisterClassW。该程式可以在 Microsoft Windows NT 中执行良好。但如果此程式在 Windows 98 上执行，RegisterClassW 函式并未真地被执行到。函式有一个进入点，但函式呼叫後只传回 0，表明错误。对於在 Windows 98 下执行的 Unicode 程式来说，这是一个通知使用者有问题并终止执行的好机会。这是本书中多数程式处理 RegisterClass 函式呼叫的方法：

```
if (!RegisterClass (&wndclass))
{
     MessageBox (     NULL, TEXT ("This program requires Windows NT!"),
                 szAppName, MB_ICONERROR) ;
     return 0 ;
}
```

由於 MessageBoxW 是可在 Windows 98 环境下执行的几个 Unicode 函式之一，所以其执行正常。

当然，这段程式假定 RegisterClass 不会因为其他原因而呼叫失败，诸如 WNDCLASS 结构中 lpfnWndProc 栏位被设定成 NULL 之类的错误。GetLastError 函式会帮助您确定在这样的情况下产生错误的原因。GetLastError 是 Windows 中常用的函式，它可以在函式呼叫失败时获得更多错误资讯。不同函式的文件将指出您是否能够用 GetLastError 来获得这些资讯。在 Windows 98 中呼叫 RegisterClassW 时，GetLastError 将传回 120。在 WINERROR.H 中您可以看到，值 120 与识别字 ERROR_CALL_NOT_IMPLEMENTED 相等。您也可以在/Platform SDK/Windows Base Services/Debugging and Error Handling/Error Codes/System Errors - Numerical Order 查看错误。

一些 Windows 程式写作者喜欢检查所有可能发生错误的函式呼叫的传回值。这么做确实有点道理，相信您也非常习惯在配置记忆体後检查错误。而许多

Windows 函式需要配置记忆体。例如，RegisterClass 需要配置记忆体，以保存视窗类别的资讯。如此一来，您就应该要检查这个函式的执行结果。另一方面说来，如果由於 RegisterClass 不能得到所需要的记忆体，它会宣告呼叫失败，而 Windows 大概也快当掉了。

在本书的范例程式中，我做了最少的错误检查。这不是因为我认为错误检查不是一个好方法，而是因为这会让我们在程式举例中分心。

最後，一个老经验是：在一些 Windows 范例程式中，您可能在 WinMain 中看到以下程式码：

```
if (!hPrevInstance)
{
      wndclass.cbStyle = CS_HREDRAW | CS_VREDRAW ;

            初始化其他 wndclass

      RegisterClass (&wndclass) ;
}
```

这是出於「旧习难改」的原因。在 16 位元的 Windows 中，如果您启动正在执行的程式的一个新执行实体，WinMain 的 hPrevInstance 参数将是前一个执行实体的执行实体代号。为节省记忆体，两个或多个执行实体就可能会共用相同的视窗类别。这样，视窗类别就只在 hPrevInstance 是 NULL 的时候才注册，这表明程式没有其他执行实体。

在 32 位元的 Windows 中，hPrevInstance 总是 NULL。此程式码会正常执行，而实际上也没必要检查 hPrevInstance。

建立视窗

视窗类别定义了视窗的一般特徵，因此可以使用同一视窗类别建立许多不同的视窗。实际呼叫 CreateWindow 建立视窗时，可能指定有关视窗的更详细的资讯。

Windows 程式设计新手有时会混淆视窗类别和视窗之间的区别，以及为什么一个视窗的所有特徵不能被一次设定好。实际上，以这种方式分开这些样式资讯是非常方便的。例如，所有的按钮视窗都可以依据同样的视窗类别来建立，与这个视窗类别相关的视窗讯息处理程式位於 Windows 内部。由视窗类别来负责处理按钮的键盘和滑鼠输入，并定义按钮在萤幕上的外观形象。从这一点看来，所有的按钮都是以同样的方式工作的。但是并非所有的按钮都是一样的。它们可以有不同的大小，不同的萤幕位置，以及不同的字串。後面的这样一些特徵是视窗定义的一部分，而不是视窗类别定义的。

传递给 RegisterClass 函式的资讯会在一个资料结构中设定好，而传递给 CreateWindow 函式的资讯会在函式单独的参数中设定好。下面是 HELLOWIN.C 中的 CreateWindows 呼叫，每一个栏位都做了完整的说明：

```
hwnd = CreateWindow (szAppName,    // window class name
      TEXT (      "The Hello Program"),  // window caption
            WS_OVERLAPPEDWINDOW,       // window style
      CW_USEDEFAULT,              // initial x position
            CW_USEDEFAULT,              // initial y position
            CW_USEDEFAULT,              // initial x size
            CW_USEDEFAULT,              // initial y size
            NULL,                     // parent window handle
            NULL,                   // window menu handle
      hInstance,                // program instance handle
      NULL) ;                   // creation parameters
```

在这里，我不想提实际上有 CreateWindowA 函式和 CreateWindowW 函式，两个函式分别将前两个参数当成 ASCII 或者 Unicode 字串来处理。

标记为「window class name」的参数是 szAppName，它含有字串「HelloWin」——这是程式注册的视窗类别名称。这就是我们建立的视窗联结视窗类别的方式。

此程式建立的视窗是一个普通的重叠式视窗。它含有一个标题列，标题列左边有一个系统功能表按钮，标题列右边有缩小、放大和关闭图示，四周还有一个表示视窗大小的边框。这是标准样式的视窗，名为 WS_OVERLAPPEDWINDOW，出现在 CreateWindow 的「视窗样式」参数中。如果看一下 WINUSER.H，您将会发现此样式是几种位元旗标的组合：

```
#define     WS_OVERLAPPEDWINDOW (WS_OVERLAPPED| \
            WS_CAPTION                        | \
            WS_SYSMENU                        | \
            WS_THICKFRAME                       | \
            WS_MINIMIZEBOX                      | \
            WS_MAXIMIZEBOX)
```

「视窗标题」是显示在标题列中的文字。

注释著「initial x position」和「initial y position」的参数指定了视窗左上角相对於萤幕左上角的初始位置。由於这些参数使用 CW_USEDEFAULT 识别字，指示 Windows 使用重叠视窗的内定位置。（CW_USEDEFAULT 定义为 0x80000000。）内定情况下，Windows 依次对新建立的视窗定位，使各视窗左上角的垂直和水平距离在萤幕上按一定的大小递增。与此类似，注释著「initial x size」和「initial y size」的参数分别指定视窗的宽度和高度。同样使用了 CW_USEDEFAULT 识别字，表明希望 Windows 使用内定尺寸。

在建立一个「最上层」视窗，如应用程式视窗时，注释为「父视窗代号」

的参数设定为 NULL。通常，如果视窗之间存在有父子关系，则子视窗总是出现在父视窗的上面。应用程式视窗出现在桌面视窗的上面，但不必为呼叫 CreateWindow 而找出桌面视窗的代号。

因为视窗没有功能表，所以「视窗功能表代号」也设定为 NULL。「程式执行实体代号」设定为执行实体代号，它是作为 WinMain 的参数传递给这个程式的。最後，「建立参数」指标设定为 NULL，可以用这个参数存取稍後程式中可能引用到的资料。

CreateWindow 传回被建立的视窗的代号，该代号存放在变数 hwnd 中，後者被定义为 HWND 型态（「视窗代号型态」）。Windows 中的每个视窗都有一个代号，程式用代号来使用视窗。许多 Windows 函式需要使用 hwnd 作为参数，这样，Windows 才能知道函式是针对哪个视窗的。如果一个程式建立了许多视窗，则每个视窗均有一个代号。视窗代号是 Windows 程式所处理最重要的代号之一。

显示视窗

在 CreateWindow 呼叫传回之後，Windows 内部已经建立了这个视窗。这就是说，Windows 已经配置了一块记忆体，用来保存在 CreateWindow 呼叫中指定视窗的全部资讯跟一些其他资讯，而 Windows 稍後就是依据视窗代号找到这些资讯的。

然而，光是这样子，视窗并不会出现在视讯显示器上。您还需要两个函式呼叫，一个是：

```
ShowWindow (hwnd, iCmdShow) ;
```

第一个参数是刚刚用 CreateWindow 建立的视窗代号。第二个参数是作为参数传给 WinMain 的 iCmdShow。它确定最初如何在萤幕上显示视窗，是一般大小、最小化还是最大化。在开始功能表中安装程式时，使用者可能做出最佳选择。如果视窗按一般大小显示，那么 WinMain 接收到後传递给 ShowWindow 的就是 SW_SHOWNORMAL；如果视窗是最大化显示的，则为 SW_SHOWMAXIMIZED。而如果视窗只显示在工作列上，则是 SW_SHOWMINNOACTIVE。

ShowWindow 函式在显示器上显示视窗。如果 ShowWindow 的第二个参数是 SW_SHOWNORMAL，则视窗的显示区域就会被视窗类别中定义的背景画刷所覆盖。函式呼叫

```
UpdateWindow (hwnd) ;
```

会重画显示区域。它经由发送给视窗讯息处理程式（即 HELLOWIN.C 中的 WndProc 函式）一个 WM_PAINT 讯息做到这一点。後面，我们将说明 WndProc 如何处理这个讯息。

讯息回圈

呼叫 UpdateWindow 之後，视窗就出现在视讯显示器上。程式现在必须准备读入使用者用键盘和滑鼠输入的资料。Windows 为当前执行的每个 Windows 程式维护一个「讯息伫列」。在发生输入事件之後，Windows 将事件转换为一个「讯息」并将讯息放入程式的讯息伫列中。

程式通过执行一块称之为「讯息回圈」的程式码从讯息伫列中取出讯息：

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

msg 变数是型态为 MSG 的结构，型态 MSG 在 WINUSER.H 中定义如下：

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

POINT 资料型态也是一个结构，它在 WINDEF.H 中定义如下：

```
typedef struct tagPOINT
{
      LONG  x ;
      LONG  y ;
}
POINT, * PPOINT;
```

讯息回圈以 GetMessage 呼叫开始，它从讯息伫列中取出一个讯息：

```
GetMessage (&msg, NULL, 0, 0)
```

这一呼叫传给 Windows 一个指标，指向名为 msg 的 MSG 结构。第二、第三和第四个参数设定为 NULL 或者 0，表示程式接收它自己建立的所有视窗的所有讯息。Windows 用从讯息伫列中取出的下一个讯息来填充讯息结构的各个栏位，结构的各个栏位包括：

hwnd 接收讯息的视窗代号。在 HELLOWIN 程式中，这一参数与 CreateWindow 传回的 hwnd 值相同，因为这是该程式拥有的唯一视窗。

message 讯息识别字。这是一个数值，用以标识讯息。对於每个讯息，均有一个对应的识别字，这些识别字定义於 Windows 表头档案（其中大多数在 WINUSER.H 中），以字首 WM（「window message」，视窗讯息）开头。例如，

使用者将滑鼠游标放在 HELLOWIN 显示区域之内，并按下滑鼠左按钮，Windows 就在讯息伫列中放入一个讯息，该讯息的 message 栏位等於 WM_LBUTTONDOWN。这是一个常数，其值为 0x0201。

wParam	一个 32 位元的「message parameter（讯息参数）」，其含义和数值根据讯息的不同而不同。
lParam	一个 32 位元的讯息参数，其值与讯息有关。
time	讯息放入讯息伫列中的时间。
pt	讯息放入讯息伫列时的滑鼠座标。

只要从讯息伫列中取出讯息的 message 栏位不为 WM_QUIT(其值为 0x0012)，GetMessage 就传回一个非零值。WM_QUIT 讯息将导致 GetMessage 传回 0。

叙述

```
TranslateMessage (&msg) ;
```

将 msg 结构传给 Windows，进行一些键盘转换。（关於这一点，我们将在第六章中深入讨论。）

叙述

```
DispatchMessage (&msg) ;
```

又将 msg 结构回传给 Windows。然後，Windows 将该讯息发送给适当的视窗讯息处理程式，让它进行处理。这也就是说，Windows 将呼叫视窗讯息处理程式。在 HELLOWIN 中，这个视窗讯息处理程式就是 WndProe 函式。处理完讯息之後，WndProc 传回到 Windows。此时，Windows 还停留在 DispatchMessage 呼叫中。在结束 DispatchMessage 呼叫的处理之後，Windows 回到 HELLOWIN，并且接著从下一个 GetMessage 呼叫开始讯息回圈。

视窗讯息处理程式

以上我们所讨论的都是必要的负担：注册视窗类别，建立视窗，然後在萤幕上显示视窗，程式进入讯息回圈，然後不断从讯息伫列中取出讯息来处理。

实际的动作发生在视窗讯息处理程式中。视窗讯息处理程式确定了在视窗的显示区域中显示些什么以及视窗怎样回应使用者输入。

在 HELLOWIN 中，视窗讯息处理程式是命名为 WndProc 的函式。视窗讯息处理程式可任意命名（只要求不和其他名字发生冲突）。一个 Windows 程式可以包含多个视窗讯息处理程式。一个视窗讯息处理程式总是与呼叫 RegisterClass 注册的特定视窗类别相关联。CreateWindow 函式根据特定视窗类别建立一个视窗。但依据一个视窗类别，可以建立多个视窗。

视窗讯息处理程式总是定义为如下形式：

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam,

LPARAM lParam)

注意，视窗讯息处理程式的四个参数与 MSG 结构的前四个栏位是相同的。第一个参数 hwnd 是接收讯息的视窗的代号，它与 CreateWindow 函式的传回值相同。对於与 HELLOWIN 相似的程式（只建立一个视窗），这个参数是程式所知道的唯一视窗代号。如果程式是依据同一视窗类别（同时也是同一视窗讯息处理程式）建立多个视窗，则 hwnd 标识接收讯息的特定视窗。

第二个参数与 MSG 结构中的 message 栏位相同，它是标识讯息的数值。最後两个参数都是 32 位元的讯息参数，提供关於讯息的更多资讯。这些参数包含每个讯息型态的详细资讯。有时讯息参数是两个存放在一起的 16 位元值，而有时讯息参数又是一个指向字串或资料结构的指标。

程式通常不直接呼叫视窗讯息处理程式，视窗讯息处理程式通常由 Windows 本身呼叫。通过呼叫 SendMessage 函式，程式能够直接呼叫它自己的视窗讯息处理程式。我们将在後面的章节讨论 SendMessage 函式。

处理讯息

视窗讯息处理程式所接受的每个讯息均是用一个数值来标识的，也就是传给视窗讯息处理程式的 message 参数。Windows 表头档案 WINUSER.H 为每个讯息参数定义以「WM」（视窗讯息）为字首开头的识别字。

一般来说，Windows 程式写作者使用 switch 和 case 结构来确定视窗讯息处理程式接收的是什么讯息，以及如何适当地处理它。视窗讯息处理程式在处理讯息时，必须传回 0。视窗讯息处理程式不予处理的所有讯息应该被传给名为 DefWindowProc 的 Windows 函式。从 DefWindowProc 传回的值必须由视窗讯息处理程式传回。

在 HELLOWIN 中，WndProc 只选择处理三种讯息：WM_CREATE、WM_PAINT 和 WM_DESTROY。视窗讯息处理程式的结构如下：

```
switch (iMsg)
{
case  WM_CREATE :
      处理 WM_CREATE 讯息
      return 0 ;

case  WM_PAINT :
      处理 WM_PAINT 讯息
      return 0 ;

case  WM_DESTROY :
      处理 WM_DESTROY 讯息
      return 0 ;
```

```
}
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

呼叫 DefWindowProc 来为视窗讯息处理程式不予处理的所有讯息提供内定处理，这是很重要的。不然一般动作，如终止程式，将不会正常执行。

播放音效档案

视窗讯息处理程式接收的第一个讯息——也是 WndProc 选择处理的第一个讯息——是 WM_CREATE。当 Windows 在 WinMain 中处理 CreateWindow 函式时，WndProc 接收这个讯息。就是说，在 HELLOWIN 呼叫 CreateWindow 时，Windows 将做一些它必须做的工作。在这些工作中，Windows 呼叫 WndProc，将第一个参数设定为视窗代号，第二个参数设定为 WM_CREATE。WndProc 处理 WM_CREATE 讯息并将控制传回给 Windows。 Windows 然後可以从 CreateWindow 呼叫中传回到 HELLOWIN 中，继续在 WinMain 中进行下一步的处理。

通常，视窗讯息处理程式在 WM_CREATE 处理期间进行一次视窗初始化。HELLOWIN 对这个讯息的处理中播放一个名为 HELLOWIN.WAV 的音效档案。它使用简单的 PlaySound 函式来做到这一点。该函式说明在/Platform SDK/Graphics and Multimedia Services/Multimedia Audio/Waveform Audio 中，而文件在/Platform SDK/Graphics and Multimedia Services/Multimedia Reference/Multimedia Functions 中。

PlaySound 的第一个参数是音效档案的名称（它也可能是在 Control Panel 的 Sounds 中定义的一种声音的别名，或者是一个程式资源）。第二个参数只有当音效档案是一种资源时才被使用。第三个参数指定一些选项。在这个例子中，我指定第一个参数是一个档案名，并且非同步地播放声音，即 PlaySound 函式呼叫在音效档案开始播放时立即传回，而不会等待它的完成。在这种方法下，程式能够继续初始化。

WndProc 通过从视窗讯息处理程式中传回 0,结束了整个 WM_CREATE 的处理。

WM_PAINT 讯息

WndProc 处理的第二个讯息为 WM_PAINT。这个讯息在 Windows 程式设计中是很重要的。当视窗显示区域的一部分显示内容或者全部变为「无效」，以致於必须「更新画面」时，将由这个讯息通知程式。

显示区域的显示内容怎么会变得无效呢？在最初建立视窗的时候，整个显示区域都是无效的，因为程式还没有在视窗上画什么东西。第一条 WM_PAINT 讯息（通常发生在 WinMain 中呼叫 UpdateWindow 时）指示视窗讯息处理程式在显

示区域上画一些东西。

在使用者改变 HELLOWIN 视窗的大小後，显示区域的显示内容重新变得无效。读者应该还记得，HELLOWIN 中 wndclass 结构的 style 栏位设定为标志 CS_HREDRAW 和 CS_VREDRAW，这样的格式设定指示 Windows，在视窗大小改变後，就把整个视窗显示内容当成无效。然後，视窗讯息处理程式将收到一条 WM_PAINT 讯息。

当使用者将 HELLOWIN 最小化，然後再次将视窗恢复为以前的大小时，Windows 将不会保存显示区域的内容。在图形环境下，视窗显示区域涉及的资料量很大。因此，Windows 令视窗无效，视窗讯息处理程式接收一条 WM_PAINT 讯息，并自动恢复其视窗的内容。

在移动视窗以致其相互重叠时，Windows 不保存一个视窗中被另一个视窗所遮盖的内容。在这一部分不再被遮盖之後，它就被标志为无效。视窗讯息处理程式接收到一条 WM_PAINT 讯息，以更新视窗的内容。

对 WM_PAINT 的处理几乎总是从一个 BeginPaint 呼叫开始：

```
hdc = BeginPaint (hwnd, &ps) ;
```

而以一个 EndPaint 呼叫结束：

```
EndPaint (hwnd, &ps) ;
```

在这两个呼叫中，第一个参数都是程式的视窗代号，第二个参数是指向型态为 PAINTSTRUCT 的结构指标。PAINTSTRUCT 结构中包含一些视窗讯息处理程式，可以用来更新显示区域的内容。我们将在下一章中讨论该结构的各个栏位。现在我们只在 BeginPaint 和 EndPaint 函式中用到它。

在 BeginPaint 呼叫中，如果显示区域的背景还未被删除，则由 Windows 来删除。它使用注册视窗类别的 WNDCLASS 结构的 hbrBackground 栏位中指定的画刷来删除背景。在 HELLOWIN 中， 这是一个白色备用画刷。这意味著，Windows 将通过把视窗背景设定为白色来删除视窗背景。BeginPaint 呼叫令整个显示区域有效，并传回一个「装置内容代号」。装置内容是指实体输出设备（如视讯显示器）及其装置驱动程式。在视窗的显示区域显示文字和图形需要装置内容代号。但是从 BeginPaint 传回的装置内容代号不能在显示区域之外绘图，读者可以试一试。EndPaint 释放装置内容代号，使之不再有效。

如果视窗讯息处理程式不处理 WM_PAINT 讯息（这是很罕见的），它们必须被传送给 DefWindowProc。DefWindowProc 只是依次呼叫 BeginPaint 和 EndPaint，以使显示区域有效。

呼叫完 BeginPaint 之後，WndProc 接著呼叫 GetClientRect：

```
GetClientRect (hwnd, &rect) ;
```

第一个参数是程式视窗的代号。第二个参数是一个指标，指向一个 RECT 型

态的 rectangle 结构。该结构有四个 LONG 栏位，分别为 left、top、right 和 bottom。GetClientRect 将这四个栏位设定为视窗显示区域的尺寸。left 和 top 栏位通常设定为 0，right 和 bottom 栏位设定为显示区域的宽度和高度（图元点数）。

WndProc 除了将该 RECT 结构指标作为 DrawText 的第四个参数传递外，不再对它做其他处理：

```
DrawText ( hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
           DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
```

DrawText 可以输出文字（正如其名字所表明的一样）。由於该函式要输出文字，第一个参数是从 BeginPaint 传回的装置内容代号，第二个参数是要输出的文字，第三个参数是 -1，指示字串是以位元组 0 终结的。

DrawText 最後一个参数是一系列位元旗标，它们均在 WINUSER.H 中定义（虽然由於其显示输出的效果，使得 DrawText 像一个 GDI 函式呼叫，但它确实因为相当高级的画图功能而成为 User 模组的一部分。此函式在/Platform SDK/Graphics and Multimedia Services/GDI/Fonts and Text 中说明）。旗标指示了文字必须显示在一行上，水平方向和垂直方向都位於第四个参数指定的矩形中央。因此，这个函式呼叫将让字串「Hello, Windows 98!」显示在显示区域的中央。

一旦显示区域变得无效（正如在改变大小时所发生的情况一样），WndProc 就接收到一个新的 WM_PAINT 讯息。WndProc 通过呼叫 GetClientRect 取得变化後的视窗大小，并在新视窗的中央显示文字。

WM_DESTROY 讯息

WM_DESTROY 讯息是另一个重要讯息。这一个讯息指示，Windows 正在根据使用者的指示关闭视窗。该讯息是使用者单击 Close 按钮或者在程式的系统功能表上选择 Close 时发生的（在本章的後面，我们将详细讨论 WM_DESTROY 讯息是如何生效的）。

HELLOWIN 通过呼叫 PostQuitMessage 以标准方式回应 WM_DESTROY 讯息：

```
PostQuitMessage (0) ;
```

该函式在程式的讯息伫列中插入一个 WM_QUIT 讯息。前面提到过，GetMessage 对於除了 WM_QUIT 之外的从讯息伫列中取出的所有讯息都传回非 0 值。而当 GetMessage 得到一个 WM_QUIT 讯息时，它传回 0。这将导致 WinMain 退出讯息回圈，并终止程式。然後程式执行下面的叙述：

```
return msg.wParam ;
```

结构的 wParam 栏位是传递给 PostQuitMessage 函式的值（通常是 0）。然

後 return 叙述将退出 WinMain 并终止程式。

WINDOWS 程式设计的难点

即使有了对 HELLOWIN 的说明，读者对程式的结构和原理可能仍然觉得神秘。在为传统环境编写简单的 C 程式时，整个程式可能包含在 main 函式中。而在 HELLOWIN 中，WinMain 只包含了注册视窗类别，建立视窗，从讯息伫列中取出讯息和发送讯息所必须的程式码。

程式的所有实际动作均在视窗讯息处理程式中发生。在 HELLOWIN 中，这些动作不多，WndProc 只是简单地播放了一个音效档案并在视窗中显示一个字串。但是在後面的章节中，读者将发现，Windows 程式所作的一切，都是回应发送给视窗讯息处理程式的讯息。这是概念上的主要难点之一，在开始写作 Windows 程式之前，必须先搞清楚。

别呼叫我，我会呼叫您

前面我们提到过，程式写作者已经熟悉了使用作业系统呼叫的做法。例如，C 程式写作者使用 fopen 函式打开档案。fopen 函式最终通过呼叫作业系统来打开档案，这一点问题也没有。

但是 Windows 不同，尽管 Windows 有 1000 个以上的函式可供程式呼叫，但 Windows 也呼叫使用者程式，比如前面定义的视窗讯息处理程式 WndProc。视窗讯息处理程式与视窗类别相关，视窗类别是程式呼叫 RegisterClass 注册的。依据该类别建立的视窗使用这个视窗讯息处理程式来处理视窗的所有讯息。Windows 通过呼叫视窗讯息处理程式对视窗发送讯息。

在第一次建立视窗时，Windows 呼叫 WndProc。在视窗关闭时，Windows 也呼叫 WndProc。视窗改变大小、移动或者变成图示时，从功能表中选择某一项目、挪动卷动列、按下滑鼠按钮或者从键盘输入字元时，以及视窗显示区域必须被更新时，Windows 都要呼叫 WndProc。

所有这些 WndProc 呼叫都以讯息的形式进行。在大多数 Windows 程式中，程式的主要部分都用来处理讯息。Windows 可以发送给视窗讯息处理程式的讯息通常都以 WM 开头的名字标识，并且都在 WINUSER.H 表头档案中定义。

实际上，从程式外呼叫程式内的常式这一种做法，在传统的程式设计中并非前所未闻。C 中的 signal 函式可以拦截 Ctrl-C 中断或作业系统的其他中断。为 MS-DOS 编写的老程式中经常有拦截硬体中断的程式码。

但在 Windows 中，这种概念扩展为包括一切事件。视窗中发生的一切都以讯息的形式传给视窗讯息处理程式。然後，视窗讯息处理程式以某种方式回应

这个讯息，或者将讯息传给 DefWindowProc，进行内定处理。

在 HELLOWIN 中，视窗讯息处理程式的 wParam 和 lParam 参数除了作为传递给 DefWindowProc 的参数外，不再有其他用处。这些参数给出了关於讯息的其他资讯，参数的含义与具体讯息相关。

让我们来看一个例子。一旦视窗的显示区域大小发生了改变，Windows 就呼叫视窗的视窗讯息处理程式。视窗讯息处理程式的 hwnd 参数是改变大小的视窗的代号（请记住，一个视窗讯息处理程式能处理依据同一个视窗类别建立的多个视窗的讯息。参数 hwnd 让视窗讯息处理程式知道是哪个视窗在接收讯息）。参数 message 是 WM_SIZE。讯息 WM_SIZE 的参数 wParam 的值是 SIZE_RESTORED、SIZE_MINIMIZED、SIZE_MAXIMIZED、SIZE_MAXSHOW 或 SIZE_MAXHIDE（在 WINUSER.H 表头档案中分别定义为数字 0 到 4）。也就是说，参数 wParam 表明视窗是非最小化还是非最大化，是最小化、最大化，还是隐藏。

lParam 参数包含了新视窗的大小，新宽度和新高度均为 16 位元值，合在一起成为 32 位元的 lParam。WINDEF.H 中提供了帮助程式写作者从 lParam 中取出这两个值的巨集，我们将在下一章说明这个巨集。

有时候，DefWindowProc 处理完讯息後会产生其他的讯息。例如，假设使用者执行 HELLOWIN，并且使用者最终单击了 Close 按钮，或者假设用键盘或滑鼠从系统功能表中选择了 Close，DefWindowProc 处理这一键盘或者滑鼠输入，在检测到使用者选择了 Close 选项之後，它给视窗讯息处理程式发送一条 WM_SYSCOMMAND 讯息。WndProc 将这个讯息传给 DefWindowProc。DefWindowProc 给视窗讯息处理程式发送一条 WM_CLOSE 讯息来回应之。WndProc 再次将它传给 DefWindowProc。DestroyWindow 呼叫 DestroyWindow 来回应这条 WM_CLOSE 讯息。DestroyWindow 导致 Windows 给视窗讯息处理程式发送一条 WM_DESTROY 讯息。WndProc 再呼叫 PostQuitMessage，将一条 WM_QUIT 讯息放入讯息伫列中，以此来回应此讯息。这个讯息导致 WinMain 中的讯息回圈终止，然後程式结束。

伫列化讯息与非伫列化讯息

我们已经谈到过，Windows 给视窗发送讯息，这意味著 Windows 呼叫视窗讯息处理程式。但是，Windows 程式也有一个讯息回圈，它呼叫 GetMessage 从讯息伫列中取出讯息，并且呼叫 DispatchMessage 将讯息发送给视窗讯息处理程式。

那么，Windows 程式是依次等待讯息（类似于普通程式中相同的键盘输入），然後将讯息送到某地方去的吗？或者，它是直接从程式外面接收讯息的吗？实际上，两种情况都存在。

讯息能够被分为「伫列化的」和「非伫列化的」。伫列化的讯息是由 Windows 放入程式讯息伫列中的。在程式的讯息回圈中，重新传回并分配给视窗讯息处理程式。非伫列化的讯息在 Windows 呼叫视窗时直接送给视窗讯息处理程式。也就是说，伫列化的讯息被「发送」给讯息伫列，而非伫列化的讯息则「发送」给视窗讯息处理程式。任何情况下，视窗讯息处理程式都将获得视窗所有的讯息--包括伫列化的和非伫列化的。视窗讯息处理程式是视窗的「讯息中心」。

伫列化讯息基本上是使用者输入的结果，以击键（如 WM_KEYDOWN 和 WM_KEYUP 讯息）、击键产生的字元（WM_CHAR）、滑鼠移动（WM_MOUSEMOVE）和滑鼠按钮（WM_LBUTTONDOWN）的形式给出。伫列化讯息还包含时钟讯息（WM_TIMER）、更新讯息（WM_PAINT）和退出讯息（WM_QUIT）。

非伫列化讯息则是其他讯息。在许多情况下，非伫列化讯息来自呼叫特定的 Windows 函式。例如，当 WinMain 呼叫 CreateWindow 时，Windows 将建立视窗并在处理中给视窗讯息处理程式发送一个 WM_CREATE 讯息。当 WinMain 呼叫 ShowWindow 时，Windows 将给视窗讯息处理程式发送 WM_SIZE 和 WM_SHOWWINDOW 讯息。当 WinMain 呼叫 UpdateWindow 时，Windows 将给视窗讯息处理程式发送 WM_PAINT 讯息。键盘或滑鼠输入时发出的伫列化讯息信号，也能在非伫列化讯息中出现。例如，用键盘或滑鼠选择了一个功能表项时，键盘或滑鼠讯息就是伫列化的，而说明功能表项已选中的 WM_COMMAND 讯息则可能就是非伫列化的。

这一过程显然很复杂，但幸运的是，其中的大部分是由 Windows 解决的，不关我们的程式的事。从视窗讯息处理程式的角度来看，这些讯息是以一种有序的、同步的方式进出的。视窗讯息处理程式可以处理它们，也可以不处理。

当我说讯息是以一种有序的同步的方式进出时，我是说首先讯息与硬体的中断不同。在一个视窗讯息处理程式中处理讯息时，程式不会被其他讯息突然中断。

虽然 Windows 程式可以多执行绪执行，但每个执行绪的讯息伫列只为视窗讯息处理程式在该执行绪中执行的视窗处理讯息。换句话说，讯息回圈和视窗讯息处理程式不是并发执行的。当一个讯息回圈从其讯息伫列中接收一个讯息，然後呼叫 DispatchMessage 将讯息发送给视窗讯息处理程式时，直到视窗讯息处理程式将控制传回给 Windows，DispatchMessage 才能结束执行。

当然，视窗讯息处理程式能呼叫给视窗讯息处理程式发送另一个讯息的函式。这时，视窗讯息处理程式必须在函式呼叫传回之前完成对第二个讯息的处理。那时视窗讯息处理程式将处理最初的讯息。例如，当视窗程序呼叫 UpdateWindow 时，Windows 将呼叫视窗讯息处理程式来处理 WM_PAINT 讯息。视窗讯息处理程式处理 WM_PAINT 讯息结束以後，UpdateWindow 呼叫将把控制传回

给视窗讯息处理程式。

这也就是说视窗讯息处理程式必须是可重入。在大多数情况下，这不会带来问题，但是程式写作者应该意识到这一点。例如，假设您在视窗讯息处理程式中处理一个讯息时设置了一个静态变数，然後呼叫了一个 Windows 函式。在这个函式传回时，您还能保证那个变数的值还是原来那个吗？难说--很可能您呼叫的 Windows 函式产生了另外一个讯息，并且视窗讯息处理程式在处理这个讯息时改变了该变数的值。这也是在编译 Windows 程式时，有些编译最佳化选项必须关闭的原因之一。

在许多情况下，视窗讯息处理程式必须保存它从讯息中取得的资讯，并在处理另一个讯息时使用这些资讯。这些资讯可以储存在视窗的静态（static）变数或整体变数中。

当然，读者将在下面几章对此有一个更清楚的了解，因为视窗讯息处理程式将处理更多的讯息。

行动迅速

Windows 98 和 Windows NT 都是优先权式的多工环境。这意味著当一个程式在进行一项长时间工作时，Windows 可以允许使用者将控制切换到另一个程式中。这是一件好事，也是现在的 Windows 优越於以前 16 位元 Windows 的地方。

然而，由於 Windows 设计的方式，这种优先权式多工并不总是以您希望的样子工作。例如，假设您的程式花费一分钟左右来处理某一个讯息。是的，使用者可以将控制切换到另一个程式，但是却无法对您的程式进行任何动作。使用者无法移动您的程式视窗、缩放它、最小化、关闭它、什么都不能做。这是因为您的视窗讯息处理程式正忙於进行一项长时间的作业。表面上并不是视窗讯息处理程式在执行它自己的移动和缩放操作，但实际上确实是它在做。这就是 DefWindowProc 部分的工作，它必须被考虑为您的视窗讯息处理程式的一部分。

如果您的程式在处理某些讯息时需要长时间的作业的话，可以选择我在第二十章里描述的那些方法来做得更有优雅一些。即使是在优先权式多工环境中，也不应该让您的程式呆在萤幕上一动不动。这会让使用者讨厌的，他们会认为您的程式中有 bug、不标准的动作，说明档案没写好。最好让使用者觉得程式只停了一下子就把全部讯息中快速料理完了。