第一章 Windows 编程概述

本章讨论 Windows 程序设计，其主要内容如下：首先，以一般方式讨论什么是 Windows？应用程序怎样与它进行交互？每个 Windows 应用程序必须遵循哪些原则？其次开发了一个应用程序框架，这一系列中所有其他 Windows 程序都可以此为基础来开发。正如读者将要看到的，所有 Windows 程序都有某些公共特征，而框架程序中所包含的正是这些公共特征。

1.1 什么是 Windows

Windows 究竟是什么？在某种程度上，这要看读者是一个终端用户还是一个程序员。从用户的观点来看，Windows 是一个“外壳”，用户可通过它与程序进行交互。然而，从程序员的观点来看，Windows 是一个面向图形的多任务操作系统，它由几百个 API（应用程序接口）函数组成，并支持一种特殊的应用程序设计方法。从程序员的观点来看，Windows 是提供各种相互关联的服务的巨大工具箱。当正确使用的时候，它允许产生共享同一界面的多个应用程序。

Windows 的目标是，一个对该系统仅有基本的熟悉程度的人能够坐下来实际运行任何应用程序，而无需任何先期训练。从理论上讲，如果用户能够运行一个 Windows 程序，他就能够运行所有的 Windows 程序。当然，实际上最有用的程序仍然要求用户受过某种训练，以便更有效地使用这些程序，但至少这些训练可被局限在程序做什么而不是用户应怎样与之交互上。事实上，在一个 Windows 应用程序中，许多代码是为支持用户界面而编写的。

对读者来说，非常重要的是要理解：不是每一个在 Windows 下运行的程序都必须提供一个 Windows 风格的界面，而只有那些利用了 Windows 功能的程序看起来和感觉起来才像一个 Windows 程序。尽管读者可以不理会基本的 Windows 程序设计原则，但最好为此找到足够的理由，因为这些程序的用户可能会因为程序看起来不像一个 Windows 程序而感到恼火。如果读者正在为 Windows 编写应用程序，那么就应遵循普遍接受的 Windows 程序设计原则。

如前所述，Windows 是面向图形的，这意味着它提供了一个图形用户界面。图形硬件和视频模式各不相同，但这些差异都由 Windows 来处理。这意味着，在大多数情况下，用户程序不必关心正在使用的是何种图形硬件或视频模式。

1.1.1 Windows 与 Windows NT

正如读者可能知道的，现在有两种不同的 Windows，一种是 16 位的 Windows，它在 DOS 下运行，且支持 16 位程序设计环境，这就是通常使用的那种 Windows。16 位 Windows 的当前版本是 3.1 版。另一种 Windows 系统是 Windows NT，它是一个独立的操作系统，支持完整的 32 位编程环境。Windows NT 是相当新的，而且在写作本书时它尚未普遍使用。

从编程的角度来看，Windows 3.1 程序和 Windows NT 程序之间的差异相对而言是较小

的。本书中第一章到第十二章的例子是设计来在 Windows 3.1 下运行的。Windows NT 编程及实例在第十三章到第十五章讨论。本书侧重于 16 位 Windows 的原因是，在写作本书时，Windows 3.1 是更为常用的编程环境。虽然如此，有关 Windows 3.1 和 Windows NT 的特殊区别在全书中每一适当位置都作了标注。

1.1.2 桌面模型

虽然也有若干例外，但基于窗口的用户界面的要点在于在屏幕上为用户提供一个桌面的"等价物"。在桌面上，用户会发现几张不同的"纸"，一张叠着一张。通常，最上一页下面的各页也有部分可见。在 Windows 中，桌面的等价物就是屏幕，而纸张的等价物就是屏幕上的窗口。在桌面上，用户可能会移动纸张，也许会把某张纸移到顶部，或者改变另外一张纸，使其一部分可见。Windows 也允许对其窗口作同样的操作。当选择了某个窗口时，可使它成为"当前窗口"，这意味着把它放在所有其他窗口的上面。可以放大和缩小窗口，也可以移动它。简言之，Windows 允许用户以控制桌面的方式来控制屏幕。

1.1.3 鼠标接口

与 DOS 不同的是，Windows 允许用户使用鼠标来进行几乎所有的控制、选择和绘图操作。当然，说它"允许"使用鼠标是过于保守的说法。事实上，Windows 界面是为鼠标设计的，但它也允许使用键盘。尽管一个应用程序对鼠标置之不管是可能的，但它在这样做的同时，也就违反了 Windows 的基本设计原则。

1.1.4 图标和图形/图像

Windows 允许(但不是要求)使用图标和位图(bit-map)图形/图像。使用图标和图形/图像的理论依据简单得像一句格言：图片胜过千言万语。

图标是一个较小的符号，代表一些功能或程序。可通过将鼠标移到图标上并双击按钮来激活它。图形/图像通常被用来将信息快速传递给用户。

1.1.5 菜单和对话框

除了标准窗口之外，Windows 也提供了用以完成特殊目的的窗口，其中最常用的窗口就是菜单和对话框。简单地说，菜单就是其中含有许多项的特殊窗口，用户可从中选择所需的项。但是，在建立 Windows 程序时，并不是必须提供菜单选择，程序员可以利用内置的菜单功能简单地建立一个标准菜单。

对话框所提供的交互操作比菜单要复杂。例如，应用程序可利用对话框来输入文件名。

1.2 Windows 怎样与用户程序交互

当为别的操作系统编写程序时，通常是由用户程序为了与操作系统进行交互而作初始化工作。例如，在一个 DOS 程序中，通常是由这个程序来完成诸如请求输入/输出这类事情。不同之处在于，以"传统方式"编写的程序是调用操作系统，而不是由操作系统调用该程序，然而 Windows 在大多数场合是以相反的方式工作的，即由 Windows 调用用户程序，该过程

是如下运作的：一个 Windows 程序一直等待，直到 Windows 发送一条“消息”给它，消息是由 Windows 调用的一个特殊函数传送给用户程序的。一旦收到一条消息，用户程序便被期待作出适当的动作。当对一条消息作出响应时，用户程序可以调用一或多个 Windows API 函数，而与此有关的初始化工作仍是由 Windows 完成的。

Windows 可能发送许多不同种类的消息给用户程序。例如，每当鼠标按钮在一个属于用户程序的窗口内被按下时，Windows 就会发送一条“鼠标按下”消息给用户程序；每当一个属于用户程序的窗口需要重画时，另一类消息会被发送给用户程序；当用户程序需要接受输入时，每当用户按下一个键，就有一条消息被发送给用户程序。请记住，对用户程序而言，消息是随机到达的，这就是 Windows 程序类似中断驱动程序的原因。程序无法知道下一条消息是什么。

最后，在处理之前，发送给用户程序的消息被存储在与用户程序相关的一个“消息队列”中。因此，不会由于用户程序正忙于处理另外的消息而导致消息被丢失。消息会在队列中等待，直到用户程序准备好处理它为止。

1.3 Windows 中的多任务

如前所述，Windows 是一个多任务操作系统。作为一个多任务操作系统，在采用“非占先式多任务”这一点上，它是有些独特的。这意味着在系统中运行的每个程序都保留着对处理器的占用，直到放弃为止，这与其他操作系统通过采用基于时间片的占先式任务切换来实现的那一类多任务是根本不同的。在占先式任务切换中，操作系统简单地停止运行一个程序，并以一种轮流的方式转而去运行另一个程序。请记住，Windows 不是这样工作的。一个 Windows 程序必须主动放弃 CPU。

注意：不像 Windows 3.1，Windows NT 采用占先式多任务，并且大多数程序员认为这是一个重大的改进(Windows NT 多任务将在第十四章讨论)。尽管 Windows 与 Windows NT 采用两种不同形式的多任务，但程序员为它们编写程序的一般方法是相同的。

Windows 程序必须遵循的最重要的原则之一是：必须在暂停运行前将控制交回给 Windows。这样，Windows 才能将控制交给另一任务。如读者将要看到的，将控制返回给 Windows 通常是很容易的。不管怎样，请记住，一个程序独占处理器，从而有效地终止其他任务也是可能的。

1.4 应用程序接口(API)

Windows 环境是通过一个称作 API(应用程序接口)的基于调用的接口来访问的，由几百个 API 函数提供 Windows 执行的所有系统服务。

API 有一个子系统，称作 GDI(图形设备界面)，它是 Windows 的一部分，用以提供与设备无关的图形支持。正是 GDI 函数使得 Windows 应用程序能在许多不同种类的硬件上运行。

有两种版本的 API：16 位 Windows 使用标准的 16 位 API；Windows NT 使用 Win32，它是新的 32 位 API。

1.5 窗口的组成部分

在开始介绍 Windows 程序设计的各个特殊方面之前，还需要定义一些重要的术语。图 1.1 给出了一个标准窗口，它的各个元素都已指出。

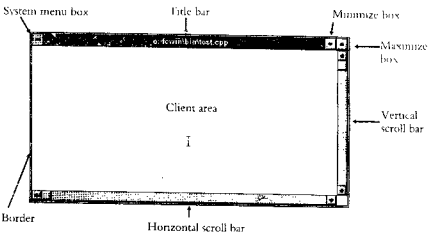

图 1.1 标准窗口的元素

所有窗口都有一个边框，它定义了窗口的边界，并且可用来移动窗口或改变窗口的大小。在窗口顶部有几项：最左边是系统菜单图标，就是通常所说的系统菜单框，在此框上单击鼠标按钮会使系统菜单显示出来；系统菜单框右边是窗口标题；最右边是最小框和最大框。客户区是窗口中用户程序活动的地方。大多数窗口还有水平或垂直滚动条，可用来移动窗口中的文本。

1.6 Windows 应用程序的基础知识

在着手开发 Windows 应用程序框架之前，还需要讨论一些基本概念，它们对所有 Windows 程序都是通用的。

1.6.1 WinMain()函数

所有 Windows 程序的运行都从对 WinMain()函数的调用开始。注意，Windows 程序中没有 main()函数。WinMain()有一些区别于用户应用程序中其他函数的特殊性质。首先，它必须作为一个 PASCAL 函数来编译。如读者可能知道的，调用函数的方法因计算机语言的不同而各有差异。两种最常见的调用约定是 C 调用约定和 PASCAL 调用约定。由于各种技术原因，Windows 通过遵照 PASCAL 调用约定调用 WinMain()函数来运行用户程序。其次，WinMain()函数有四个参数，以便在用户程序开始运行时接收与窗口有关的信息(这些参数

将在本章稍后论及)。

注意:Windows NT 使用的调用约定是 WINAPI,而不是 PASCAL。在写作本书时,PASCAL 与 WINAPI 是等价的,但以后就不一定了。

1.6.2 窗口函数

所有 Windows 程序都必须包含一个特殊的函数,它不是由用户程序调用,而是由 Windows 调用的。这个函数通常被称作窗口函数。每当 Windows 需要发送一条消息给用户程序时,它就调用窗口函数。Windows 借助这个函数与用户程序进行通信。窗口函数接收消息并将其保存到参数中。所有窗口函数必须声明为 LONG FAR PASCAL。用 Windows 术语来讲,任何由 Windows 调用的函数都称为"回调函数"。

注意:在 Windows NT 中,回调函数(包括窗口函数)必须用 CALLBACK 类型说明符来声明。许多编译器也支持在 16 位 Windows 程序中使用这种类型。

除了接收 Windows 发送的消息之外,窗口函数还必须初始化消息所指定的操作。窗口函数的主体一般由 switch 语句构成,它使消息与程序对不同消息作出的不同响应联系起来。应用程序并不需要对 Windows 传来的任何消息都作出响应;对那些与程序无关的消息,可以让 Windows 对其进行缺省处理。Windows 总是产生许多不同类型的消息,而其中许多消息是由 Windows 本身处理的,应用程序完全可以对这些无关的消息置之不理。

所有 Windows 消息均为 16 位整型值,每条消息都含有相应的附加信息。

注意:Windows NT 中的消息为 32 位数值。

1.6.3 窗口类

当首次运行 Windows 应用程序时,需要定义并登录窗口类(window class),此处"class"代表风格或类型,这与 C++ 中的含义有所不同。登录窗口类就是将窗口的风格及功能告知 Windows。但是,登录窗口类并不意味着生成窗口,实际生成窗口还需执行其他操作。

1.6.4 消息循环

如前所述,Windows 以发送消息的方式与应用程序进行通信。所有 Windows 应用程序均应在 WinMain()函数中建立消息循环,此循环从应用程序的消息队列中读取待处理的消息,然后将此消息分发(dispatch)给 Windows,Windows 以此消息为参数来调用应用程序的窗口函数。上述传送消息的方式似乎很复杂,但所有的 Windows 应用程序都应遵循这种工作方式。这样做的原因是强迫应用程序不时地将控制返回给 Windows,以支持 Windows 的非占先式多任务操作。

1.6.5 Windows 中的数据类型

Windows 不采用标准的 C/C++ 数据类型,如 int 或 char * 等。事实上,Windows 使用的所有数据类型都在 WINDOWS.H 头文件中作了定义,此文件应包含于所有 Windows 应用程序中,而且支持 Windows 应用程序的所有编译器都必须提供这一头文件。常见的 Windows 数据类型包括 HANDLE,HWND,BYTE,WORD,DWORD,LONG,BOOL 以及 LPSTR。HANDLE 类型为 16 位整型值,可用作句柄。所谓句柄,就是用来标识资源的数值。HWND 是用作

窗口句柄的 16 位整数。句柄类型的变量均以大写字母 H 开头。BYTE 为无符号字符。WORD 为 16 位整数，它是 16 位无符号整数的“别名”。DWORD 为 32 位无符号整数。UINT 为无符号整数。LONG 为 32 位长整型值。BOOL 为整数，这种数据类型用于指明数据“真”或“假”。LPSTR 为字符串指针。LPCSTR 为字符串常量指针。

除了上述基本类型之外，Windows 还定义了若干结构。本书的第一个框架程序中用到了两种结构：MSG 和 WNDCLASS。MSG 结构用于存放 Windows 消息，而 WNDCLASS 结构则用于定义窗口类型。本章的后面部分将对这两种结构予以说明。

另外还有两种重要的数据类型，即 LPARAM 和 WPARAM，这两种数据类型用于存放与 Windows 消息相对应的数值。LPARAM 为长型整值，WPARAM 为无符号整数。

注意，Windows 开发环境的早期版本中没有定义 LPARAM 和 WPARAM 类型，因此，这两种类型不可用于早期版本的程序。一般来说，LPARAM 为 LONG 的类型定义(typedef)，WPARAM 为 UINT 的类型定义。早期版本的应用程序可用这两种基本类型来代替 LPARAM 和 WPARAM。

1.6.6 模块定义文件

所有的 Windows 应用程序都含有相应的模块定义文件，该文件描述并定义了与 Windows 程序相关的各种特性。模块定义文件的扩展名为 .DEF。有关模块定义文件的内容，将在讨论了 Windows 框架程序之后再予以讨论。

1.7 一个 Windows 框架应用程序

至此，所有的基本背景知识都已讨论完毕。下面结合一个简单的 Windows 应用程序具体加以说明。如前所述，所有的 Windows 应用程序都有一些共同点，下面的应用程序体现了这些特点。在 Windows 编程的过程中，应用程序框架很常用，这是因为它确实有“值得采用的价值”。与 DOS 应用程序不同，最小的 Windows 程序也得有大约 50 条语句，而最小的 DOS 程序大约只有 5 条语句。因此，在开发 Windows 应用程序时，程序员大都喜欢采用程序框架。

最小的 Windows 应用程序包含两个函数：WinMain()函数和窗口函数。WinMain()函数必须完成下述操作：

(1) 定义窗口类。

(2) 登录窗口类。

(3) 生成某一类型的窗口。

(4) 显示窗口。

(5) 执行消息循环。

窗口函数必须响应所有相关的消息。由于框架程序的唯一功能是显示窗口，因此该程序必须响应的唯一消息就是告知应用程序，用户结束了程序的运行。

在深入讨论之前，先看一下下面的程序。该程序是最小的 Windows 框架程序，它生成一个标准窗口，窗口中包含系统菜单，利用它可以把窗口放大到最大或缩小到最小，也可移动、缩放或关闭窗口。窗口中还包含标准的最大、最小图标。

注意，在编译此 Windows 框架程序之前，需要准备好其模块定义文件。本章后面部分列

出了该模块定义文件。

```
/* skel:简单的框架应用程序 */

#include <windows.h>

LONG FAR PASCAL WndProc(HWND,UNIT,WPARAM,LPARAM);

char szProgName[] = "ProgName";        /* 窗口类名 */

int PASCAL WinMain(HINSTANCE hInst,HINSTANCE hPreInst,
                   LPSTR lpszCmdLine,int nCmdShow)
{
  HWND hWnd;
  MSG lpMsg;
  WNDCLASS wcApp;

  /* 如果这是程序的第一个实例,那么窗口类必须被登录 */
  if (! hPreInst) {                    /* 如果没有先前实例 */
    wcApp.lpszClassName  = szProgName;  /* 窗口类名 */
    wcApp.hInstance      = hInst;       /* 指向该实例的句柄 */
    wcApp.lpfnWndProc    = WndProc;     /* 窗口函数 */
    wcApp.hCursor        = LoadCursor(NULL,IDC_ARROW);
    wcApp.hIcon          = LoadIcon(NULL,IDI_APPLICATION);
    wcApp.lpszMenuName   = NULL;        /* 无主菜单 */

    /* 使窗口的背景为白色 */
    wcApp.hbrBackground = GetStockObject(WHITE_BRUSH);
    wcApp.style          = 0;           /* 使用缺省窗口风格 */
    wcApp.cbClsExtra     = 0;           /* 无额外信息 */
    wcApp.cbWndExtra     = 0;           /* 无额外信息 */

    /* 登录窗口类 */
    if (! RegisterClass (&wcApp))
      return FALSE;
  }

  /* 既然窗口已被登录,就可创建一个窗口 */
  hWnd = CreateWindow (
      szProgName,                   /* 窗口名 */
      "Skeleton Application",       /* 标题 */
      WS_OVERLAPPEDWINDOW,          /* 普通窗口类型 */
      CW_USEDEFAULT,                /* X 坐标 — 由 Windows 决定 */
      CW_USEDEFAULT,                /* Y 坐标 — 由 Windows 决定 */
      CW_USEDEFAULT,                /* 宽度 — 由 Windows 决定 */
      CW_USEDEFAULT,                /* 高度 — 由 Windows 决定 */
      (HWND)NULL,                   /* 无父窗口 */
      (HWND)NULL,                   /* 无主菜单 */
```

```
    (HANDLE)hInst,                  /* 该实例的句柄 */
    (LPSTR)NULL                     /* 无多余参数 */
);

/* 显示窗口 */
ShowWindow(hWnd,nCmdShow);
UpdateWindow(hWnd);

/* 创建消息循环 */
while (GetMessage(&lpMsg,NULL,0,0))
{
  TranslateMessage(&lpMsg);         /* 允许使用键盘 */
  DispatchMessage(&lpMsg);          /* 将控制返回给 Windows */
}
return(lpMsg.wParam);
}

/* 该函数由 Windows 调用并从消息队列传送消息 */
LONG FAR PASCAL WndProc(HWND hWnd,UINT messg,
                        WPARAM wParam,LPARAM lParam)
{
  switch (messg) {
    case WM_DESTROY:                /* 结束程序 */
      PostQuitMessage(0);
      break;

    /* 使 Windows 处理任何未在先前情况下声明的消息 */
    default:
      return(DefWindowProc(hWnd,messg,wParam,lParam));
  }
  return(0L);
}
```

下面一步一步地剖析此程序。

首先，所有的 Windows 应用程序均应包含头文件 WINDOWS. H。如前所述，该文件中包含 Windows 所调用的 API 函数的原型以及各种类型与定义。例如，数据类型 HWND 及 WNDCLASS 就是在 WINDOWS. H 中定义的。

上述程序所用的窗口函数名为 WndProc()，此函数的原型应包含于应用程序中。同时，该函数应在应用程序的模块定义文件中定义为 EXPORT 类型。

如前所述，Windows 程序从 WinMain()开始运行。WinMain()含有四个参数。hInst 和 hPreInst 分别是应用程序的当前实例及前一个实例的句柄。由于 Windows 是多任务系统，因此，很可能会出现同时运行一个应用程序的多个实例的情况。如果前一个实例不存在，则 hPreInst 为 0。lpszCmdLine 所指的字符串中包含命令行参数，命令行参数是应用程序开始运行时定义的。nCmdShow 参数确定了在程序开始运行时是显示窗口还是显示图标。

函数内部定义了三个变量：hWnd 为应用程序的窗口句柄，lpMsg 结构变量中存放着 Windows 消息，而 wcApp 结构则用于定义窗口类。

1.7.1 定义窗口类

WinMain()的前两步操作就是定义窗口类并登录窗口。不过，在 Windows 中窗口只能登录一次，这样，若程序正在运行另一实例，则不应第二次定义并登录窗口类。应用程序可检查 hPreInst 的值，看它是否为零，不为零时表示原先的实例已登录了窗口类。若 hPreInst 为零，则 WinMain()可定义并登录这一窗口类。

注意：由于 Windows NT 的进程之间相互独立，因而窗口类亦不能共享，各个实例均应登录自己的窗口类，这是 Windows NT 与 16 位 Windows 的本质区别。

定义窗口类是通过给 WNDCLASS 结构赋值完成的。WNDCLASS 结构定义如下：

```
typedef struct tagWNDCLASS
{
  UINT        style;          /* 窗口类型 */
  WNDPROC     lpfnWndProc;    /* 窗口函数 */
  int         cbClsExtra;     /* 额外信息 */
  int         cbWndExtra;     /* 额外信息 */
  HINSATANCE  hInstance;      /* 该实例的句柄 */
  HICON       hIcon;          /* 最小化图标的句柄 */
  HCURSOR     hCursor;        /* 鼠标光标的句柄 */
  HBRUSH      hbrBackground;  /* 背景颜色 */
  LPCSTR      lpszMenuName;   /* 主菜单名 */
  LPCSTR      lpszClassName;  /* 窗口类名 */
} WNDCLASS;
```

从程序中可以看出，指向窗口类的 lpszClassName 指针此时指向“ProgName”字符串。hInst 所指定的当前实例句柄被赋给 hInstnce 域，窗口函数的地址被赋给 lpfnWndProc 域。

所有 Windows 应用程序均需定义鼠标光标或最小图标的缺省形状。应用程序可自行定义这些资源，也可采用系统自带的资源。框架程序就选用了系统自带的资源。

最小图标形状由 API 函数 LoadIcon()加载，该 API 函数的原型如下：

```
HICON LoadIcon (HINSTANCE hInst,LPCSTR lpszName);
```

该函数返回图标的句柄。hInst 为包含图标的模块的句柄，lpszName 为图标名称。不过，若要使用系统自带的图标，则应将第一个参数置为 NULL，而将第二个参数置为下列值之一：

图 标 宏	形 状
IDI_APPLICATION	缺省图标
IDI_ASTERISK	信息图标
IDI_EXCLAMATION	感叹号图标
IDI_HAND	终止符图标
IDI_QUESTION	问号图标

API 函数 LoadCursor()可用于加载鼠标光标，该函数具有如下原型：

```
HCURSOR LoadCursor (HINSTANCE hInst,LPCSTR,lpszName);
```

此函数返回光标资源的句柄。这里，hInst 指定了包含鼠标光标的模块的句柄，lpszName 则定义了光标的名称；不过，若采用系统自带的光标，则应将第一个参数置为 NULL，而将第二个参数置为与该光标相对应的宏。常用的系统光标如下：

光 标 宏	形 状
IDC_ARROW	缺省箭头指针
IDC_CROSS	十字交叉线
IDC_IBEAM	垂直的"I"形光标
IDC_WAIT	砂漏形光标

注意，若 hIcon 和 hCursor 均定义为 NULL，则采用缺省的图标及光标。

由于没有主菜单，从而 lpszMenuName 元素定义为 NULL。

框架程序所生成的窗口的背景颜色定义为白色，这一画刷的句柄可通过 API 函数 GetStockObject()获得。所谓画刷，是指以预定义的大小、颜色及模式来"涂抹"屏幕的资源。函数 GetStockObject()可用于获取一系列标准显示对象的句柄，包括画刷、画笔(用于画线)及字体等，其原型如下：

```
HGDIOBJ GetStockObject (int Object);
```

该函数返回 object 所定义的对象的句柄。应用程序可使用下列系统画刷：

宏 名 称	背景颜色
BLACK_BRUSH	黑色
DKGRAY_BRUSH	深灰色
HOLLOW_BRUSH	透明(无色)
LTGRAY_BRUSH	浅灰色
WHITE_BRUSH	白色

style 元素定义了所生成的窗口的类型。若此值为 0，则 Windows 生成标准窗口，此窗口具有窗口的所有共同属性。不过，应用程序也可自行定义用户窗口类型，这是通过对 WINDOWS.H 中所定义的宏进行"或"运算得到的。这些宏常量均以 CS 开头。例如，可采用 CS HREDRAW|CS VREDRAW 来定义 style 元素，以生成标准窗口。

cbClsExtra 及 cbWndExtra 元素用于在窗口类结构中保留空间，这些元素可供应用程序使用，不过，本书中没有程序需要用到额外的空间，因此这两个参数均置为 0。

一旦完成了窗口类的定义，就可利用 API 函数 RegisterClass()来登录它。该函数的原型

如下：

```
ATOM RegisterClass (const WNDCLASS FAR * lpWClass);
```

该函数返回对应于窗口类的值。一般情况下，该返回值将被忽略。

1.7.2 生成窗口

一旦完成了窗口类的定义及登录工作，应用程序就可利用 API 函数 CreateWindow()来生成窗口。该函数的原型如下：

```
HWND CreateWindow (
        LPCSTR lpszClassName,          /* 窗口类名称 */
        LPCSTR lpszWinName,            /* 窗口名称 */
        DWORD dwStyle,                 /* 窗口类型 */
        int X,Y,                       /* 左上角坐标 */
        int Width,Height,              /* 窗口宽度及高度 */
        HWND hParent,                  /* 父窗口句柄 */
        HMENU hMenu,                   /* 主菜单句柄 */
        HINSTANCE hThisInst,           /* 拥有者句柄 */
        void FAR * lpszAdditional);    /* 附加信息 */
```

从框架程序中不难看出，CreateWindow()的许多参数均可缺省设置为 NULL。事实上，应用程序经常用宏 CW_USEDEFAULT 来告诉 Windows 为窗口选择适当的位置与大小，而不是分别定义 X,Y,Width 及 Height 参数。若当前窗口没有父窗口(这也正是本程序中的情况)，则 hParent 应定义为 NULL，也可定义为 HWND_DESKTOP。若窗口中不包含主菜单，则 hMenu 应置为 NULL。同样，若不要求其他信息(这是很常见的情况)，此时 lpszAdditional 也为 NULL。

剩下的四个参数是程序必须设置的。首先，lpszClassName 应指向窗口类名称，即登录窗口时的名称。lpszWinName 则指向窗口名称字符串，该字符串可以为空串(NULL)。但是，一般情况下窗口都是有名称的，所以该字符串也就不为空。所生成的窗口风格或类型由 dwStyle 的值确定。宏 WS_OVERLAPPEDWINDOW 规定了一个包含系统菜单、边框、最大框和最小框的标准窗口。应用程序可根据需要将若干最为常见的窗口风格组合成用户窗口风格，这只要对这些风格各异的宏作“或”运算即可。常见的窗口风格如下：

风　格　宏	窗口属性
WS_OVERLAPPED	带边框的重叠窗口
WS_MAXIMIZEBOX	最大框
WS_MINIMIZEBOX	最小框
WS_SYSMENU	系统菜单
WS_HSCROLL	水平滚动条
WS_VSCROLL	垂直滚动条

hThisInst 参数中应包含应用程序的当前实例的句柄。

在生成了窗口之后，窗口并未显示出来。要显示窗口，应调用 API 函数 ShowWindow()。该函数的原型如下：

```
BOOL ShowWindow (HWND hwnd,int nHow);
```

待显示的窗口的句柄由 hwnd 定义，显示方式由 nHow 定义。WinMain()中的 nCmdShow 参数值决定了程序运行时该窗口以图标形式还是以打开的窗口形式显示。应用程序中的后续调用可根据需要显示或删除此窗口。nHow 的常用值如下：

显 示 宏	作 用
SW_HIDE	删除窗口
SW_MINIMIZE	窗口缩小成图标
SW_MAXIMiZE	窗口放大到最大
SW_RESTORE	窗口恢复到正常大小

ShowWindow()函数返回原来的窗口显示状态。若窗口原来处于显示状态，则返回非零值；若窗口原来未显示，则返回零。

在消息循环开始之前调用的最后一个 API 函数是 UpdateWindow()，此函数向应用程序发送一条消息，告知应用程序修正(也就是重新显示)窗口客户区的内容。在前面的框架程序中，从技术上讲并不需要这一函数，但是，由于真正的 Windows 应用程序都要用到这一函数，所以，框架程序中也包含了此函数。UpdateWindow()函数的原型如下：

```
void UpdateWindow (HWND hWnd);
```

其中，hWnd 为要修正的窗口的句柄。

1.7.3 消息循环

WinMain()的最后一部分内容是消息循环，消息循环也是所有 Windows 应用程序的一个公共组成部分。消息循环的作用在于接收并处理由 Windows 发送的消息。在 Windows 应用程序的运行过程中，总是不断地发送消息，这些消息在被应用程序读入和处理之前存放在应用程序的消息队列中。每当应用程序要读入一条消息，都必须调用 API 函数 GetMessage()。该函数的原型如下：

```
BOOL GetMessage (MSG FAR * msg,HWND hwnd,UINT min,UINT max);
```

消息由 lpMsg 所指的结构接收。所有的 Windows 消息都具有 MSG 结构类型：

```
/* 消息结构 */
```

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

消息发送到的目标窗口的句柄为 hwnd。所有的 Windows 消息都是 16 位整数，且存放于 message 中。相应于各消息的额外信息由 wParam 及 lParant 传递。应用程序开始运行的时间以微秒定义。POINT 结构定义如下：

```
typedef struct tagPOINT
{
    int x,y;
} POINT;
```

若应用程序消息队列中没有消息，则调用 GetMessage()时控制将返回给 Windows。非占先式多任务操作就是以这种方式完成的。

第二章将更详细地讨论消息。

GetMessage()中的 hwnd 参数决定了获取消息的窗口。通常，应用程序有多个窗口，而又只希望由某一窗口接收消息。若应用程序要接收所有送至程序的消息，则应将此参数置为 NULL。

GetMessage()函数的另外两个参数规定了接收消息的范围。一般情况下，应用程序要接收所有消息。要做到这一点，只需像此框架程序一样将 min 与 max 同时置为 0 即可。

当用户结束程序时，GetMessage()返回零，从而结束消息循环，否则此函数返回非零值。

在消息循环中调用了两个函数。第一个是 API 函数 TranslateMessage()，此函数将 Windows 所生成的虚拟键代码转换成实际的字符(本书后面的章节将讨论虚拟键)。尽管并非所有应用程序都必须调用 TransateMessage()，但大多数应用程序都调用此函数，以便将键盘功能完全融入应用程序中。

消息一经读入及翻译，应用程序就调用 DispatchMessage()将其发送给 Windows，然后由 Windows 保存此消息，直到传递给应用程序的窗口函数。

当 WinMain()函数结束时，它向 Windows 返回 lpMsg. wParam 的值，该数值为应用程序结束时所生成的返回码。

1.7.4 窗口函数

框架应用程序中的第二个函数是窗口函数。在本程序中，窗口函数名为 WndProc()。应用程序可采用不同的窗口函数名称。窗口函数以 MSG 结构的前四项内容为参数。本程序唯一用到的参数就是消息本身，不过，第二章将讲述更多有关此函数参数的内容。

事实上，框架程序中的窗口函数仅对一条消息作出响应，这就是 WM_DESTROY 消息。

这一消息是当用户终止程序时发出的。一旦接收到这条消息，应用程序就应调用 API 函数 PostQuitMessage()，此函数的参数为一个退出码，此退出码由 WinMain()的 lpMsg. wParam 返回。调用 PostQuitMessage()时将产生送往应用程序的 WM_QUIT 消息，从而使 GetMessage()返回 FALSE，以结束应用程序的运行。

WndProc()所接收的其他消息都通过调用 DefWindowProc()传递给 Windows，并由 Windows 对其进行缺省处理。这一步是不可缺少的，因为所有的消息均应得到这种或那种方式的处理。

1.8 框架程序的模块定义文件

如前所述，任何一个 Windows 应用程序都应包含相应的模块定义文件。所谓模块定义文件，是指一个定义了 Windows 应用程序的某些信息及设置的文本文件。

注意，有些编译器(如 Borland C/C++)自动提供缺省设置，从而使应用程序不需要模块定义文件，但是，在这种情况下最好还是采用模块定义文件，因为有了模块定义文件，就可以精确地控制与应用程序相应的各种参数。

注意，Windows NT 应用程序无需模块定义文件。

所有的模块定义文件均以. DEF 为扩展名。例如，上述框架程序的模块定义文件名为 SKEL. DEF。下面的模块定义文件可供读者参考。如果没有特殊要求，这样的模块定义文件已经完全能够满足本书程序实例的要求。

```
NAME            skel
DESCRIPTION     'skeleton Module Definition File'
EXETYPE         WINDOWS
STUB            'WINSTUB. EXE'
CODE            PRELOAD MOVEABLE
DATA            PRELOAD MOVEABLE MULTIPLE
HEAPSIZE        4096
STACKSIZE       9216
EXPORTS         WndProc
```

该文件中规定了程序名称及其描述，这两项内容均是任选的，该文件同时也规定了可执行文件与 Windows 兼容。STUB 语句告知编译器，当用户在 DOS(而非 Windows)环境下运行 Windows 应用程序时，将显示一条信息，说明该程序必须在 Windows 环境下运行。CODE 语句告知 Windows，要在启动时装载所有的程序(PRELOAD)，且代码可在内存中移动(MOVABLE)。该文件还说明，应用程序要在运行前被加载，并可在内存中移动。同时，应用程序的各个实例拥有各自的数据段(MULTIPLE)。堆(heap)和栈(stack)的大小也定义于此文件中。EXPORTS 语句把那些可被 Windows 使用的函数名告知编译器，也就是说，必须"输出"那些可被 Windows 调用的函数，让 Windows 知道其入口点。对此框架程序而言，唯一需要输出的函数是 WndProc，因为此函数是该框架程序的窗口函数。

1.9 MAKE 文件实例

编译 Windows 应用程序要比编译 DOS 程序麻烦一些。编译前一般应设置各种各样的编译选项，并需要各种附加编译步骤。在编译此框架程序之前，应参考编译器手册中生成 Windows 程序的有关规定。

注意，不同的编译器在生成 Windows 应用程序时所要求的选项和步骤各不相同，同时，同一编译器的新旧版本编译 Windows 应用程序的方式一般也不相同。这样，编译 Windows 应用程序就没有一般的规则可循，而只能参考编译器手册中的说明。

由于 Windows 应用程序一般都需采用复杂的编译过程，许多程序员便采取了用 MAKE 文件编译其 Windows 应用程序的方法(MAKE 是工具程序，用于管理程序编译。不太了解 MAKE 文件的读者，可参考编译器手册中对 MAKE 文件的说明)。对于不同的编译器，其 MAKE 文件也不相同，这样就不存在通用的、可用于编译任何 Windows 应用程序的 MAKE 文件。用于编译本章所示的 Windows 框架程序的 MAKE 文件如下：

```
FILE=ske1
LIBPATH=c:\borlandc\lib
INCPATH=c:\borlandc\include

$(FILE).exe: $(FILE).obj $(FILE).def
  tlink /Tw /x /n /c $(LIBPATH)\c0ws $(FILE),\
  $(FILE),\,\
  $(LIBPATH)\cws+cs+import+mathws,\
  $(FILE)

$(FILE).obj: $(FILE).c
  bcc -c -W -I$(INCPATH) $(FILE).c
```

与 Borland C/C++兼容的 MAKE 文件如上所示。下面是与 Microsoft C/C++兼容的 MAKE 文件：

```
all: ske1.exe

ske1.obj: ske1.c
   cl -c   As -Csw -Oas ske1.c

ske1.exe: ske1.obj ske1.def
   link /NOD ske1,,,libw,slibcew,ske1.def
```

1.10 命名约定

在结束本章之前，再简单说明一下函数及变量命名的问题。对于刚刚接触 Windows 编程的人来说，本章所举的框架程序中的变量及参数名称可能显得离奇古怪，这是因为这些变

量及参数的命名遵循了 Microsoft 公司为 Windows 编程所制定的一系列命名原则。本书中的程序也都遵循这一原则。

Microsoft 采用了一种相当复杂的原则将数据类型融入到变量名称中，以完成变量的命名。这就是，数据类型前缀的小写字母放在变量名之前，变量名的第一个字母大写。类型前缀如表 1.1 所示。这种采用类型前缀的方法目前还有争议，而且也不通用。许多程序员采用这种方法，也有许多人不这么做。本书中的 Windows 程序都采用这种方法，不过，读者完全可以按自己的爱好选择别的命名方法。

表 1.1　类型前缀

前缀	数据类型
b	布尔型(单字节)
c	字符(单字节)
dw	无符号长整型
f	16 位位域(标志)
fn	函数
h	句柄
l	长整型
lp	长指针
n	短整型
p	短指针(也可为 np)
pt	长整型屏幕坐标
w	无符号短整型
sz	指向以 NULL 结尾的字符串的指针
lpsz	指向以 NULL 结尾的字符串的长指针
rgb	长整型 RGB 颜色值

第二章　消息处理

正如第一章所述，Windows 以向应用程序发送消息的方式与应用程序进行通信。基于这一原因，消息处理就成了所有 Windows 应用程序的核心问题。第一章已给出了 Windows 应用程序的框架。本章则将赋予"框架"以接收并处理若干常用的 Windows 消息的能力。

注意，本章中的程序采用第一章给出的模块定义文件。

2.1　什么是 Windows 消息

Windows 消息多种多样，各消息均以 16 位整数值表示（Windows NT 采用 32 位消息）。头文件 WINDOWS.H 定义了这些消息的标准名称，应用程序一般采用这些宏而非实际的整数值来代表消息。常用的 Windows 消息宏如下：

```
WM_CHAR
WM_PAINT
WM_MOVE
WM_LBUTTONUP
WM_LBUTTONDOWN
```

各消息还有两个相应的值，其中包含与消息有关的信息。一个是 WPARAM 类型的无符号整数，另一个是 LPARAM 类型的长整数。这两个参数常用于存放光标或鼠标的坐标及按键值，或者系统参数，如字符大小之类的数据。本章在讨论各种消息的同时，也将讲述 wParam 与 lParam 数值的含义。

如第一章所述，真正处理消息的是应用程序的窗口函数。窗口函数传递四个参数：消息目标窗口的句柄、消息本身以及 wParam 和 lParam。

有时长整型的 lParam（双字）由两则信息构成，为此，WINDOWS.H 头文件中定义了两个宏 LOWORD 和 HIWORD，它们用来获取长整数（如 lParam）的高位字和低位字信息。具体到 lParam，这两个宏的用法如下：

```
LOWORD(lParam)
HIWORD(lParam)
```

在本章的实例程序中，还有许多地方会用到这两个宏。

2.2　按键响应

按下键盘上的键时会生成一条常见的 Windows 消息，这就是 WM_CHAR。本节将扩充