第一章　开始

本书介绍了在 Microsoft Windows 98、Microsoft Windows NT 4.0 和 Windows NT 5.0 下程式写作的方法。这些程式用 C 语言编写并使用原始的 Windows Application Programming Interface（API）。如在本章稍後所讨论的，这不是写作 Windows 程式的唯一方法。然而，无论最终您使用什么方式写作程式，了解 Windows API 都是非常重要的。

正如您可能知道的，Windows 98 已成为使用 Intel 32 位元微处理器（例如 486 和 Pentium）的 IBM 相容型个人电脑环境上最新的图形作业系统之代表。Windows NT 是 IBM PC 相容机种以及一些 RISC（精简指令集电脑）工作站上使用的 Windows 工业增强型版本。

使用本书有三个先决条件。首先，您应该从使用者的角度熟悉 Windows 98。不要期望可以在不了解 Windows 使用者介面的情形下开发其应用程式。因此，我建议您在开发程式（或在进行其他工作）时使用执行 Windows 的机器来跑 Windows 应用程式。

第二，您应了解 C 语言。如果要写 Windows 程式，一开始却不想了解 C 语言，那不是一个好主意。我建议您在文字控制台环境中，例如在 Windows 98 MS-DOS 命令提示视窗下提供的环境中学习 C 语言。Windows 程式设计有时包括一些非文字模式程式设计的 C 语言部分；在这些情况下，我将针对这些问题提供讨论。但大多数情况下，您应非常熟悉该语言，特别是 C 语言的结构和指标。了解标准 C 语言执行期程式库的一些相关知识是有帮助的，但不是必要的。

第三，您应该在机器上安装一个适於进行 Windows 程式设计的 32 位元 C 语言编译器和开发环境。在本书中，假定您正在使用 Microsoft Visual C++ 6.0，该套装软体可独立购买，也可作为 Visual Studio 6.0 套装软体的一部分购买。

到此为止，我将不再假设您具有任何图形使用者介面（如 Windows）的程式写作经验。

WINDOWS 环境

Windows 几乎不需要介绍。然而人们很容易忘记 Windows 给办公室和家庭桌上型电脑所带来的重大改变。Windows 在其早期曾经走过一段坎坷的道路，征服桌上型电脑市场的前途一度相当渺茫。

Windows 简史

在 1981 年秋天 IBM PC 推出之後不久，MS-DOS 就已经很明显成为 PC 上的主流作业系统。MS-DOS 代表 Microsoft Disk Operating System（磁碟作业系统）。MS-DOS 是一个小型的作业系统。MS-DOS 提供给用户一种命令列介面，提供如 DIR 和 TYPE 的命令，也可以将应用程式载入记忆体执行。对於应用程式写作者，它提供了一组函式呼叫，进行档案的输入输出（I/O ）。对於其他的周边处理——尤其是将文字或图形写到显示器上——应用程式可以直接存取 PC 的硬体。

由於记忆体和硬体的限制，成熟的图形环境缓慢地才到来。当苹果电脑公司不幸的 Lisa 电脑在 1983 年 1 月发表时，它提供了不同於文字模式环境的另一种选择，并在 1984 年 1 月成为 Macintosh 上图形环境的一种标准。尽管 Macintosh 的市场占有率在下降，但是它仍然被认为是衡量所有其他图形环境的标准。包括 Macintosh 和 Windows 的所有图形环境，其实都要归功於 Xerox Palo Alto Research Center（PARC）在 70 年代中期所作的开拓性研究工作。

Windows 是由微软在 1983 年 11 月（在 Lisa 之後，Macintosh 之前）宣布，并在两年後（1985 年 11 月）发行。在此後的两年中，紧随著 Microsoft Windows 早期版本 1.0 之後，又推出了几种改进版本，以支援国际商业市场，并提供新型视讯显示器和印表机的驱动程式。

Windows 版本 2.0 是在 1987 年 11 月正式在市场上推出的。该版本对使用者介面做了一些改进。这些改进中最有效的是使用了可重叠式视窗，而 Windows 1.0 中使用的是并排式视窗。Windows 2.0 还增强了键盘和滑鼠介面，特别是加入了功能表和对话方块。

至此，Windows 还只要求 Intel 8086 或者 8088 等级的微处理器，以「实际模式」执行，只能存取位址在 1MB 以下的记忆体。Windows/386（在 Windows 2.0 之後不久发行的）使用 Intel 386 微处理器的「虚拟 8086」模式，实现将直接存取硬体的多个 MS-DOS 程式视窗化和多工化。为了统一起见，Windows 版本 2.1 被更名为 Windows/286。

Windows 3.0 是在 1990 年 5 月 22 日发表的。它将 Windows/286 和 Windows/386 结合到同一种产品中。Windows 3.0 有了一个很大的改变，这就是对 Intel 的 286、386 和 486 微处理器保护模式的支援。这能使 Windows 和 Windows 应用程式能存取高达 16MB 的记忆体。Windows 用於执行程式和维护档案的「外壳」程式得到了全面的改进。Windows 3.0 是第一个在家用和办公室市场上取得立足点的版本。

任何 Windows 的历史介绍都必须包括一些 OS/2 的说明，OS/2 是对 DOS 和 Windows 的另一种选择，最初是由 Microsoft 和 IBM 合作开发的。OS/2 版本 1.0

（只有文字模式）在 Intel 286（或者後来的）微处理器上运行，在 1987 年末发布。在 1988 年 10 月的 OS/2 版本 1.1 中出现了管理图形使用者介面的 PM（Presentation Manager）。PM 最初的设计构想是成为 Windows 的一种保护模式版本，但是图形 API 改变程度太大，致使软体生产厂商很难提供对这两种平台的支援。

到 1990 年 9 月，IBM 和 Microsoft 之间的冲突达到了高峰，导致这两个公司最後分道扬镳。IBM 接管了 OS/2，而 Microsoft 明确表示 Windows 将是他们作业系统策略的中心。虽然 OS/2 仍然拥有一些狂热的崇拜者，但是它远不及 Windows 这样的普及程度。

Microsoft Windows 版本 3.1 是 1992 年 4 月发布的，其中包括的几个重要特性是 TrueType 字体技术（给 Windows 带来可缩放的轮廓字体 ）、多媒体（声音和音乐 ）、物件连结和嵌入（OLE：Object Linking and Embedding）和通用对话方块。跟 OS/2 一样，Windows 3.1 只能在保护模式下运作，并且要求至少配置了 1MB 记忆体的 286 或 386 处理器。

在 1993 年 7 月发表的 Windows NT 是第一个支援 Intel 386、486 和 Pentium 微处理器 32 位元保护模式的 Windows 版本。Windows NT 提供 32 位元平坦定址，并使用 32 位元的指令集。（本章後面我会谈到一些定址空间的问题 ）。Windows NT 还可以移植到非 Intel 处理器上，并在几种使用 RISC 晶片的工作站上执行。

Windows 95 是在 1995 年 8 月发布的。和 Windows NT 一样，Windows 95 也支援 Intel 386 或更高等级处理器的 32 位元保护模式。虽然它缺少 Windows NT 中的某些功能，诸如高安全性和对 RISC 机器的可携性等，但是 Windows 95 具有需要较少硬体资源的优点。

Windows 98 在 1998 年 6 月发布，具有许多加强功能，包括执行效能的提高、更好的硬体支援以及与网际网路和全球资讯网（WWW）更紧密的结合。

Windows 方面

Windows 98 和 Windows NT 都是支援 32 位元优先权式多工（preemptive multitasking）及多执行绪的图形作业系统。Windows 拥有图形使用者介面（GUI），这种使用者介面也称作「视觉化介面」或「图形视窗环境」。有关 GUI 的概念可追溯至 70 年代中期，在 Alto 和 Star 等机器上以及 SmallTalk 等环境中由 Xerox PARC 所作的研究工作。该项研究的成果後来被 Apple Computer 和 Microsoft 引入主流并流行起来。虽然有一些争议，但现在已非常清楚，GUI 是（Microsoft 的 Charles Simonyi 的说法）一个在个人电脑工业史上集各方面技术大成於一体的最重要产物。

所有 GUI 都在点矩阵对应的视讯显示器上处理图形。图形提供了使用萤幕的最佳方式、传递资讯的视觉化丰富多彩环境，以及能够 WYSIWYG（what you see is what you get：所见即所得）的图形视讯显示和为书面文件准备好格式化文字输出内容。

在早期，视讯显示器仅用於回应使用者通过键盘输入的文字。在图形使用者介面中，视讯显示器自身成为使用者输入的一个来源。视讯显示器以图示和输入设备（例如按钮和卷轴）的形式显示多种图形物件。使用者可以使用键盘（或者更直接地使用滑鼠等指向装置）直接在萤幕上操纵这些物件，拖动图形物件、按下滑鼠按钮以及滚动卷轴。

因此，使用者与程式的交流变得更为亲密。这不再是一种从键盘到程式，再到视讯显示器的单向资讯流动，使用者已经能够与显示器上的物件直接交互作用了。

使用者不再需要花费长时间学习如何使用电脑或掌握新程式了。Windows 让这一切成真，因为所有应用程式都有相同的基本外观和感觉。程式占据一个视窗——萤幕上的一块矩形区域。每个视窗由一个标题列标识。大多数程式功能由程式的功能表开始。用户可使用卷轴观察那些无法在一个萤幕中装下的资讯。某些功能表项目触发对话方块，用户可在其中输入额外的资讯。几乎在每个大的 Windows 程式中都有一个用於开启档案的特殊对话方块。该对话方块在所有这些 Windows 程式中看起来都一样（或接近相同），而且几乎总是从同一功能表选项中启动。

一旦您了解使用一个 Windows 程式的方法，您就非常容易学习其他的 Windows 程式。功能表和对话方块允许用户试验一个新程式并探究它的功能。大多数 Windows 程式同时具有键盘介面和滑鼠介面。虽然 Windows 程式的大多数功能可通过键盘控制，但使用滑鼠要容易得多。

从程式写作者的角度看，一致的使用者介面来自於 Windows 建构功能表和对话方块的内置程式。所有功能表都有同样的键盘和滑鼠介面，因为这项工作是由 Windows 处理，而不是由应用程式处理。

为便於多个程式的使用，以及这些程式间资讯的交换，Windows 支援多工。在同一时刻能有多个 Windows 程式显示并运行。每个程式在萤幕上占据一个视窗。用户可在萤幕上移动视窗，改变它们的大小，在不同程式间切换，并从一个程式向另一个程式传送资料。因为这些视窗看起来有些像桌面上的纸（当然，这是电脑还未占据办公桌之前的年代），Windows 有时被称作：一个显示多个程式的「具象化桌面」。

Windows 的早期版本使用一种「非优先权式（non-preemptive）」的多工系

统。这意味著 Windows 不使用系统计时器将处理时间分配给系统中运行的多个应用程式，程式必须自愿放弃控制以便其他程式运行。在 Windows NT 和 Windows 98 中，多工是优先权式的，而且程式自身可分割成近乎同时执行的多个执行绪。

作业系统不对记忆体进行管理便无法实现多工。当新程式启动、旧程式终止时，记忆体会出现碎裂空间。系统必须能够将闲置的记忆体空间组织在一起，因此系统必须能够移动记忆体中的程式码和资料块。

即使是在 8088 微处理器上跑的 Windows 1.0 也能进行这类记忆体管理。在实际模式限制下，这种能力被认为是软体工程一个令人惊讶的成就。在 Windows 1.0 中，PC 硬体结构的 640KB 记忆体限制，在不要求任何额外记忆体的情况下被有效地扩展了。但 Microsoft 并未就此停步：Windows 2.0 允许 Windows 应用程式存取延伸记忆体（EMS）；Windows 3.0 在保护模式下，允许 Windows 应用程式存取高达 16MB 的扩展记忆体。Windows NT 和 Windows 98 通过成熟的 32 位元作业系统及平坦定址空间，摆脱了这些旧的限制。

Windows 上执行的程式可共用在称为「动态连结程式库」的档案中的常式。Windows 包括一个机制，能够在执行时连结使用动态连结程式库中常式的程式。Windows 自身基本上就是一个动态连结程式库的集合。

Windows 是一个图形介面，Windows 程式能够在视讯显示器和印表机上充分利用图形和格式化文字。图形介面不仅在外观上更有吸引力，而且还能够让使用者传递高层次的资讯。

Windows 应用程式不能直接存取萤幕和印表机等图形显示设备硬体。相反，Windows 提供一种图形程式语言（称作图形装置介面，或者 GDI），使显示图形和格式化文字更容易。Windows 虚拟化了显示硬体，使为 Windows 编写的程式可使用任何具有 Windows 装置驱动程式的视频卡或印表机，而程式无需确定系统相连的装置类型。

对 Windows 开发者来说，将与装置无关的图形介面输出到 IBM PC 上不是件轻松的事。PC 的设计是基於开放式架构的原则，鼓励第三方硬体制造商为 PC 开发周边设备，而且开发了大量这样的设备。虽然出现了多种标准，PC 上的传统 MS-DOS 程式仍不得不各自支援许多不同的硬体设备。这对 MS-DOS 文字处理软体来说非常普遍，它们连同 1 到 2 张有许多小档案的磁片一同销售，每个档案支援一种特定的印表机。Windows 程式不要求每个应用程式都自行开发这些驱动程式，因为这种支援是 Windows 的一部分。

动态连结

Windows 运作机制的核心是一个称作「动态连结」的概念。Windows 提供了

应用程式丰富的可呼叫函式，大多数用於实作其使用者介面和在视讯显示器上显示文字和图形。这些函式采用动态连结程式库（Dynamic Linking Library，DLL）的方式撰写。这些动态连结程式库是些具有.DLL 或者有时是.EXE 副档名的档案，在 Windows 98 中通常位於\WINDOWS\SYSTEM 子目录中，在 Windows NT 中通常位於 \WINNT\SYSTEM 和\WINNT\SYSTEM32 子目录中。

在早期，Windows 的主要部分仅通过三个动态连结程式库实作。这代表了 Windows 的三个主要子系统，它们被称作 Kernel、User 和 GDI。当子系统的数目在 Windows 最近版本中增多时，大多数典型的 Windows 程式产生的函式呼叫仍对应到这三个模组之一。Kernel（日前由 16 位元的 KRNL386.EXE 和 32 位元的 KERNEL32.DLL 实现）处理所有在传统上由作业系统核心处理的事务——记忆体管理、档案 I/O 和多工管理。User（由 16 位的 USER.EXE 和 32 位的 USER32.DLL 实作）指使用者介面，实作所有视窗运作机制。GDI（由 16 位的 GDI.EXE 和 32 位的 GDI32.DLL 实作）是一个图形装置介面，允许程式在萤幕和印表机上显示文字和图形。

Windows 98 支援应用程式可使用的上千种函式呼叫。每个函数都有一个描述名称，例如 CreateWindow。该函数（如您所猜想的）为程式建立新视窗。所有应用程式可以使用的 Windows 函式都在表头档案里预先宣告过。

在 Windows 程式中，使用 Windows 函式的方式通常与使用如 strlen 等 C 语言程式库函式的方式相同。主要的区别在於 C 语言程式库函式的机械码连结到您的程式码中，而 Windows 函式的程式码在您程式执行档外的 DLL 中。

当您执行 Windows 程式时，它通过一个称作「动态连结」的过程与 Windows 相接。一个 Windows 的 .EXE 档案中有使用到的不同动态连结程式库的参考资料，所使用的函式即在那些动态连结程式库中。当 Windows 程式被载入到记忆体中时，程式中的呼叫被指向 DLL 函式的入口。如果该 DLL 不在记忆体中，就把它载入到记忆体中。

当您连结 Windows 程式以产生一个可执行档案时，您必须连结程式开发环境提供的特定「引用程式库（import library）」。这些引用程式库包含了动态连结程式库名称和所有 Windows 函式呼叫的引用资讯。连结程式使用该资讯在.EXE 档案中建立一个表格，在载入程式时，Windows 使用它将呼叫转换为 Windows 函式。

WINDOWS 程式设计选项

为说明 Windows 程式设计的多种技术，本书提供了许多范例程式。这些程式使用 C 语言撰写并原原本本的使用 Windows API 来开发程式。我将这种方法

称作「古典」Windows 程式设计。这是我们在 1985 年为 Windows 1.0 写程式的方法，它今天仍是写作 Windows 程式的有效方法。

API 和记忆体模式

对於程式写作者来说，作业系统是由本身的 API 定义的。API 包含了所有应用程式能够使用的作业系统函式呼叫，同时包含了相关的资料型态和结构。在 Windows 中，API 还意味著一个特殊的程式架构，我们将在每章的开头进行研究。

一般而言，Windows API 自 Windows 1.0 以来一直保持一致，没什么重大改变。具有 Windows 98 程式写作经验的 Windows 程式写作者会对 Windows 1.0 程式的原始码感觉非常熟悉。API 改变的一种方式是进行增强。Windows 1.0 支援不到 450 个函式呼叫，现在已有了上千种函式呼叫。

Windows API 和它的语法的最大变化来自於从 16 位元架构向 32 位元架构转化的过程中。Windows 从版本 1.0 到版本 3.1 使用 16 位元 Intel 8086、8088、和 286 微处理器上所谓的分段记忆体模式，由於相容性的原因，从 386 开始的 32 位元 Intel 微处理器也支援该模式。在这种模式下，微处理器暂存器的大小为 16 位元，因此 C 的 int 资料型态也是 16 位元宽。在分段记忆体模式下，记忆体位址由两个部分组成——一个 16 位元段（segment）指标和一个 16 位偏移量（offset）指标。从程式写作者的角度看，这非常凌乱并带来了 long 或 far 指标（包括段位址和偏移量位址）和 short 或 near 指标（包括带有假定段位址的偏移量位址）的区别。

从 Windows NT 和 Windows 95 开始，Windows 支援使用 Intel 386、486 和 Pentium 处理器 32 位元模式下的 32 位元平坦定址记忆体模式。C 语言的 int 资料型态也扩展为 32 位元的值。为 32 位元版本 Windows 编写的程式使用简单的平坦线性空间定址的 32 位元指标值。

用於 16 位元版本 Windows 的 API（Windows 1.0 到 Windows 3.1）现在称作 Win16。用於 32 位元版本 Windows 的 API（Windows 95、Windows 98 和所有版本的 Windows NT）现在称作 Win32。许多函式呼叫在从 Win16 到 Win32 的转变中保持相同，但有些需要增强。例如，图像座标点由 Win16 中的 16 位元值变为 Win32 中的 32 位元值。此外，某些 Win16 函式呼叫返回一个包含在 32 位元整数值中的二维座标点。这在 Win32 中不可能，因此增加的新函式呼叫以不同方式运作。

所有 32 位元版本的 Windows 都支援 Win16 API（以确保和旧有应用程式相容）和 Win32 API（以运行新应用程式）。非常有趣的是，Windows NT 与 Windows 95 及 Windows 98 的工作方式不同。在 Windows NT 中，Win16 函式呼叫通过一

个转换层被转化为 Win32 函式呼叫，然後被作业系统处理。在 Windows 95 和 Windows 98 中，该操作正相反：Win32 函式呼叫通过转换层转换为 Win16 函式呼叫，再由作业系统处理。

在同一时刻有两个不同的 Windows API 集（至少名称不同）。Win32s（「s」代表「subset（子集）」）是一个 API，允许程式写作者编写在 Windows 3.1 上执行的 32 位元应用程式。该 API 仅支援已被 Win16 支援的 32 位元函式版本。此外，Windows 95 API 一度被称作 Win32c（「c」代表「compatibility（相容性）」），但该术语已被抛弃了。

现在，Windows NT 和 Windows 98 都被认为能够支援 Win32 API。然而，每个作业系统依然都支援某些不被别的作业系统支援的某些功能特性。因为它们的相同之处是相当可观的，所以有可能编写在两个作业系统下都可执行的程式。而且，人们普遍认为这两个产品最终会合而为一。

语言选项

使用 C 语言和原始的 API 不是编写 Windows 98 程式的唯一方法。然而，这种方法却提供给您最佳的性能、最强大的功能和在发掘 Windows 特性方面最大的灵活性。可执行档案相对较小且运行时不要求外部程式库（自然，Windows DLL 自身除外）。最重要的是，不管您最终以什么方式开发 Windows 应用程式，熟悉 API 会使您对 Windows 内部有更深入的了解。

虽然我认为学习古典的 Windows 程式设计对任何 Windows 程式写作者都是重要的，我没有必要建议使用 C 和 API 编写每个 Windows 应用程式。许多程式写作者，特别是那些为公司内部开发程式或在家编写娱乐程式的程式写作者喜欢轻松的开发环境，例如 Microsoft Visual Basic 或者 Borland Delphi（它结合了物件导向的 Pascal 版本）。这些环境使程式写作者将精力集中於应用程式的使用者介面和相关使用者介面物件的程式码上。要学习 Visual Basic，您也许需要参考 Microsoft Press 的一些其他图书，例如 Michael Halvorson1996 年著的《Learn Visual Basic Now》。

在专业程式写作者中——特别是那些开发商业应用程式的程式写作者——Microsoft Visual C++和 Microsoft Foundation Class Library（MFC）是近年来流行的选择。MFC 在一组 C++物件类别中封装了许多 Windows 程式设计中的琐碎细节。Jeff Prosise 的《Programming Windows with MFC,第二版》（Microsoft Press,1999 年）提供了 MFC 程式的写作指南。

最近，Internet 和 World Wide Web 的流行大力推广著 Sun Microsystems 的 Java，这是一个受 C++启发却与微处理器无关的程式设计语言，而且结合了

可在几个作业系统平台上执行的图形应用程式开发工具组。Microsoft Press 有一本关於 Microsoft J++（Microsoft 的 Java）开发工具的好书，《Programming Visual J++ 6.0》（1998 年），由 Stephen R. Davis 著。

显然，很难说哪种方法更有利於开发 Windows 应用程式。更主要的是，也许是应用程式自身的特性决定了所使用的工具。不管您最後实际上使用什么工具写作程式，学习 Windows API 将使您更深入地了解 Windows 工作的方式。Windows 是一个复杂的系统，在 API 上增加一个程式写作层并未减少它的复杂性，仅仅是掩盖了它，早晚您会碰到它。了解 API 会给您更好的补救机会。

在原始的 Windows API 之上的任何软体层都必定将您限制在全部功能的一个子集内。您也许发现，例如，使用 Visual Basic 编写应用程式非常理想，然而它不允许您做一个或两个很简单的基本工作。在这种情况下，您将不得不使用原始的 API 呼叫。API 定义了作为 Windows 程式写作者所需的一切。没有什么方法比直接使用 API 更万能的了。

MFC 尤其问题百出。虽然它大幅简化了某些工作（例如 OLE），我却经常发现要让它们按我所想的去工作时，会在其他特性（例如 Document/View 架构）上碰壁。MFC 还不是 Windows 程式设计者所追求的灵丹妙药，很少有人认为它是一个好的物件导向设计的模型。MFC 程式写作者从他们使用的物件类别定义如何工作中受益颇深，并会发现他们经常参考 MFC 原始码，搞懂这些原始码是学习 Windows API 的好处之一。

程式开发环境

在本书中，假定您正使用 Microsoft Visual C++ 6.0，标准版、专业版和企业版都可以。经济的标准版足以应付本书中的程式设计需求。Visual C++ 还是 Visual Studio 6.0 中的一部分。

Microsoft Visual C++ 套装软体中包括 C 编译器和其他编译及连结 Windows 程式所需的档案和工具等。它还包括 Visual C++ Developer Studio，一个可编辑原始码、以交谈方式建立资源（如图示和对话方块）以及编辑、编译、执行和测试程式的环境。

如果您正使用 Visual C++ 5.0，则需要为 Windows 98 和 Windows NT 5.0 更新表头档案和引用程式库，这些东西可从 Microsoft 的网站上得到。在 http://www.microsoft.com/msdn/ ，选择「 Downloads 」，然後选择「 Platform SDK 」（软体开发套件），您就能在选择的目录中下载和安装更新档案。要让 Microsoft Developer Studio 浏览这些目录，可以从「 Tool 」功能表项选择「 Options 」然後按下「 Directories 」标签。

Microsoft 网站上的 msdn 部分代表「Microsoft Developer Network（Microsoft 软体开发者网路）」。这是一个向程式写作者提供了经常更新的 CD-ROM 的计划，这些 CD-ROM 中包含了程式写作者在 Windows 开发中所需的最新东西。您也可以订阅 MSDN，这样就避免经常得从 Microsoft 的网站下载档案。

API 文件

本书不是 Windows API 权威的正式文件的替代品。那组文件不再以印刷形式出版，它仅能从 CD-ROM 或 Internet 上取得。

当您安装 Visual C++ 6.0 时，您将得到一个包括 API 文件的线上求助系统。您可通过订阅 MSDN 或使用 Microsoft 网站上的线上求助系统更新该文件。连接到 http://www.microsoft.com/msdn/ ，并选择「 MSDN Library Online 」。

在 Visual C++ 6.0 中，从「 Help 」功能表项选择「 Contents 」项目开启 MSDN 视窗。API 文件按树形结构组织，寻找标有「 Platform SDK 」的部分，所有在本书中引用的文件都来自於该部分。我将向您介绍如何从「 Platform SDK 」开始寻找以斜线分层分门别类的文件的位置。（我知道「Platform SDK」是整个 MSDN 知识库中较为晦涩的部分，但我敢保证那是 Windows 程式设计的基本核心。）例如，对於如何在 Windows 程式中使用滑鼠的文件，您可参考/ Platform SDK / User Interface Services / User Input / Mouse Input。

我在前面提到 Windows 大致分为 Kernel、User 和 GDI 子系统。kernel 介面在/ Platform SDK / Windows Base Services 中，User 介面函式在 / Platform SDK / User Interface Services 中，GDI 位於 / Platform SDK / Graphics and Multimedia Services / GDI 中。

编写第一个 WINDOWS 程式

现在是开始写些程式的时候了。为了便於对比，让我们以一个非常短的 Windows 程式和一个简短的文字模式程式开始。这会帮助我们找到使用开发环境并感受建立和编译程式机制的正确方向。

文字模式（Character-Mode）模型

程式写作者们喜爱的一本书是《The C Programming Language》（Prentice Hall，1978 年和 1988 年），由 Brian W. Kernighan 和 Dennis M. Ritchie（亲切地称为 K&R）编著。该书的第一章以一个显示「hello, world」的 C 语言程式开始。

这里是在《The C Programming Language》第一版第 6 页中出现的程式：

```
main ()
{
      printf ("hello, world\n") ;
}
```

以前 C 程式写作者在使用 printf 等 C 执行期程式库函式时，无需先宣告它们。但这是 90 年代，我们愿意给编译器一个在我们的程式中标出错误的机会。这里是在 K&R 第二版中修正的程式：

```
#include <stdio.h>
main ()
{
      printf ("hello, world\n") ;
}
```

该程式仍然是那么短。但它可通过编译并执行得很好，但当今许多程式写作者更愿意清楚地说明 main 函式的返回值，在这种情况下 ANSI C 规定该函式必须返回一个值：

```
#include <stdio.h>
int main ()
{
      printf ("hello, world\n") ;
      return 0 ;
}
```

我们还可以包括 main 的参数，把程式弄得更长一些，但让我们暂且这样就好了——包括一个 include 宣告、程式的进入点、一个对执行期程式库函式的呼叫和一个 return 语句。

同样效果的 Windows 程式

Windows 关於「hello, world」程式的等价程式有和文字模式版本完全相同的元件。它有一个 include 宣告、一个程式进入点、一个函式呼叫和一个 return 语句。下面便是该程式：

```
/*------------------------------------------------------------------------
HelloMsg.c -- Displays "Hello, Windows 98!" in a message box
      (c) Charles Petzold, 1998
------------------------------------------------------------------------*/
#include <windows.h>
int WINAPI WinMain (   HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0);
return 0 ;
}
```

在剖析该程式之前，让我们看一下在 Visual C++ Developer Studio 中建

立新程式的方式。

首先，从 File 功能表中选 New 。在 New 对话方块中，单击 Projects 页面标签，选择 Win32 Application 。在 Location 栏中，选择一个子目录，在 Project Name 栏中，输入该专案的名称，此时该名称是 HelloMsg ，这便是在 Location 栏中显示的目录的子目录。 Create New Workspace 核取方块应该勾起来， Platforms 部分应该显示 Win32 ，选择 OK 。

将会出现一个标题为 Win32 Application - Step 1 Of 1 的对话方块，指出要建立一个 Empty Project ，并按下 Finish 按钮。

从 File 功能表中再次选择 New 。在 New 对话方块中，选择 Files 页面标签，选择 C++ Source File 。 Add To Project 核取方块应被选中，并应显示 HelloMsg 。在 File Name 栏中输入 HelloMsg.c ，选中 OK 。

现在您可输入上面所示的 HELLOMSG.C 档案，您也可以选择 Insert 功能表和 File As Text 选项从本书附带的 CD-ROM 上复制 HELLOMSG.C 的内容。

从结构上说，HELLOMSG.C 与 K&R 的「hello,world」程式是相同的。表头档案 STDIO.H 已被 WINDOWS.H 所代替，进入点 main 被 WinMain 所代替，而且 C 语言执行时期程式库函式 printf 被 Windows API 函式 MessageBox 所代替。然而，在程式中有许多新东西，包括几个陌生的大写识别字。

让我们从头开始。

表头档案

HELLOMSG.C 以一个前置处理器指示命令开始，实际上在每个用 C 编写的 Windows 程式的开头都可看到：

```
#include <windows.h>
```

WINDOWS.H 是主要的含入档案，它包含了其他 Windows 表头档案，这些表头档案的某些也包含了其他表头档案。这些表头档案中最重要的和最基本的是：

WINDEF.H 基本型态定义。

WINNT.H 支援 Unicode 的型态定义。

WINBASE.H Kernel 函式。

WINUSER.H 使用者介面函式。

WINGDI.H 图形装置介面函式。

这些表头档案定义了 Windows 的所有资料型态、函式呼叫、资料结构和常数识别字，它们是 Windows 文件中的一个重要部分。使用 Visual C++ Developer Studio 的 Edit 功能表中的 Find in Files 搜索这些表头档案非常方便。您还可以在 Developer Studio 中打开这些表头档案并直接阅读它们。

程式进入点

正如在 C 程式中的进入点是函数 main 一样，Windows 程式的进入点是 WinMain，总是像这样出现：

```
int WINAPI WinMain （  HINSTANCE hInstance,HINSTANCE hPrevInstance,
           PSTR szCmdLine,int iCmdShow）
```

该进入点在 / Platform SDK / User Interface Services / Windowing / Windows / Window Reference / Window Functions 中有说明。它在 WINBASE.H 中宣告如下：

```
int
WINAPI
WinMain（
      HINSTANCE hInstance,
      HINSTANCE hPrevInstance,
      LPSTR lpCmdLine,
      int nShowCmd
       ）;
```

您会注意到我在 HELLOMSG.C 中做了许多小改动。第三个参数在 WINBASE.H 中定义为 LPSTR，我将它改为 PSTR。这两种资料型态都定义在 WINNT.H 中，作为指向字串的指标。LP 字首代表「长指标」，这是 16 位元 Windows 下的产物。

我还在 WinMain 宣告中改变了两个参数的名称。许多 Windows 程式中的变数名使用一种称作「匈牙利表示法」的命名系统，该系统在变数名称前面增加了表示变数资料型态的短字首，我将在第三章更详细地讨论这个概念。现在仅需记住字首 i 表示 int、sz 表示「以零结束的字串」。

WinMain 函式宣告为返回一个 int 值。WINAPI 识别字在 WINDEF.H 定义，语句如下：

```
#define WINAPI __stdcall
```

该语句指定了一个呼叫约定，包括如何生产机械码以在堆叠中放置函式呼叫的参数。许多 Windows 函式呼叫宣告为 WINAPI。

WinMain 的第一个参数被称作「执行实体代号」。在 Windows 程式设计中，代号仅是一个应用程式用来识别某些东西的数字。在这种情况下，该代号唯一地标识该程式，还需要它在其他 Windows 函式呼叫中作为参数。在 Windows 的早期版本中，当同时运行同一程式多次时，您便创建了该程式的「多个执行实体（multiple instances）」。同一应用程式的所有执行实体共用程式和唯读的记忆体（通常是例如功能表和对话方块模板的资源）。程式通过检查 hPrevInstance 参数就能够确定自身的其他执行实体是否正在运行。然後它可以略过一些繁杂的工作并从前面的执行实体将某些资料移到自己的资料区域。

在 32 位元 Windows 版本中，该概念已被抛弃。传给 WinMain 的第二个参数

总是 NULL（定义为 0）。

WinMain 的第三个参数是用於执行程式的命令列。某些 Windows 应用程式利用它在程式启动时将档案载入记忆体。WinMain 的第四个参数指出程式最初显示的方式，可以是正常的或者是最大化地充满整个画面，或者是最小化显示在工作列中。我们将在第三章中介绍使用该参数的方法。

MessageBox 函式

MessageBox 函式用於显示短资讯。虽然，MessageBox 显示的小视窗不具有什么功能，实际上它被认为是一个对话方块。

MessageBox 的第一个参数通常是视窗代号，我们将在第三章介绍其含义。第二个参数是在讯息方块主体中显示的字串，第三个参数是出现在讯息方块标题列上的字串。在 HELLMSG.C 中，这些文字字串的每一个都被封装在一个 TEXT 巨集中。通常您不必将所有字串都封装在 TEXT 巨集中，但如果想将您的程式转换为 Unicode 字元集，这确是一个好主意。我将在第二章详细讨论该问题。

MessageBox 的第四个参数可以是在 WINUSER.H 中定义的一组以字首 MB_开始的常数的组合。您可从第一组中选择一个常数指出希望在对话方块中显示的按钮：

```
#define     MB_OK                                   0x00000000L
#define     MB_OKCANCEL                             0x00000001L
#define     MB_ABORTRETRYIGNORE                     0x00000002L
#define     MB_YESNOCANCEL                          0x00000003L
#define     MB_YESNO                                0x00000004L
#define     MB_RETRYCANCEL                          0x00000005L
```

如果在 HELLOMSG 中将第四个参数设置为 0，则仅显示「 OK 」按钮。可以使用 C 语言的 OR（|）操作符号将上面显示的一个常数与代表内定按钮的常数组合：

```
#define     MB_DEFBUTTON1                           0x00000000L
#define     MB_DEFBUTTON2                           0x00000100L
#define     MB_DEFBUTTON3                           0x00000200L
#define     MB_DEFBUTTON4                           0x00000300L
```

还可以使用一个常数指出讯息方块中图示的外观：

```
#define     MB_ICONHAND                               0x00000010L
#define     MB_ICONQUESTION                         0x00000020L
#define     MB_ICONEXCLAMATION                      0x00000030L
#define     MB_ICONASTERISK                           0x00000040L
```

这些图示中的某些有替代名称：

```
#define     MB_ICONWARNING              MB_ICONEXCLAMATION
#define     MB_ICONERROR                MB_ICONHAND
#define     MB_ICONINFORMATION          MB_ICONASTERISK
```

```
#define     MB_ICONSTOP                 MB_ICONHAND
```

虽然只有少数其他 MB_常数，但您可以自己参考表头档案或 / Platform SDK / User Interface Services / Windowing / Dialog Boxes / Dialog Box Reference / Dialog Box Functions 里的档案。

在本程式中，MessageBox 返回数值 1，但更严格地说它返回 IDOK，IDOK 在 WINUSER.H 中定义，等於 1。根据在讯息方块中显示的其他按钮，MessageBox 函式还可返回 IDYES、IDNO、IDCANCEL、IDABORT、 IDRETRY 或 IDIGNORE。

这个小的 Windows 程式真的与 K&R 的「hello, world」程式有著同等效果吗？您也许认为不是，因为 MessageBox 函式并没有「hello, world」中 printf 函数所具有的潜在格式化文字能力。但我们将在下一章中看到编写类似 printf 的 MessageBox 版本的方法。

编译、连结和执行

当您准备编译 HELLOMSG 时，您可从「 Build 」功能表中选择「 Build Hellomsg.exe 」，或者按 F7 ，或者在「 Build 」工具列中选择「 Build 」图示。（该图示的外观显示在「 Build 」功能表中。如果当前没有显示「 Build 」工具列，您可从「 Tools 」功能表中选择「 Customize 」并选择「 Toolbars 」页面标签，选中「 Build 」或者「 Build MiniBar 」。）

另一种方法，您可从「 Build 」功能表中选择「 Execute Hellomsg.exe 」，或者按「 Ctrl+F5 」，或者在「 Build 」工具列单击「 Execute Program 」图示（该图示看上去像一个红的感叹号），就会弹出一个讯息方块询问是否编译该程式。

正常情况下，在编译阶段，编译器从 C 原始码档案产生一个.OBJ（目标）档案。在连结阶段，连结程式结合.OBJ 档案和.LIB（库）档案以建立.EXE（可执行）档案。通过在「 Project 」页面标签上选择「 Settings 」并单击「 Link 」页面标签可以查看这些库档案的列表。特别地，您会注意到 KERNEL32.LIB、USER32.LIB 和 GDI32.LIB。这些是三个主要 Windows 子系统的「引用程式库」。它们包含了动态连结程式库的名称以及放进.EXE 档案的引用资讯。Windows 使用该资讯处理程式对 KERNEL32.DLL、USER32.DLL、GDI32.DLL 动态连结程式库中函数的呼叫。

在 Visual C++ Developer Studio 中，您可用不同的设定编译和连结程式。内定情况下，它们是「Debug」和「Release」。可执行档案被存放在以这些名称命名的子目录下。在 Debug 设定下，资讯被附加到 .EXE 档案中，这些资讯有助於测试程式和追踪原始码。

如果您喜欢在命令列下工作，附上的 CD-ROM 包含所有范例程式的.MAK（make）档案。（可通过「 Tools 」功能表选择「 Options 」，再选择「 Build 」页面标签，来告诉 Developer Studio 生成 make 档案。这里有一个核取方块需要勾选）。您需要执行在 Developer Studio 的 BIN 子目录下的 VCVARS32.BAT 来设置环境变数。要从命令列执行 make 档案，可以转到 HELLOMSG 目录并执行：

```
NMAKE /f HelloMsg.mak CFG="HelloMsg - Win32 Debug"
```

或者

```
NMAKE /f HelloMsg.mak CFG="HelloMsg - Win32 Release"
```

然後您可通过输入：

```
DEBUG\HELLOMSG
```

或者

```
RELEASE\HELLOMSG
```

从命令列执行.EXE 档案。

我已经在本书附上的 CD-ROM 中对专案档案中的内定 Debug 设定做了一个改动。在「 Project Settings 」对话方块中，选择「 C/C++ 」页面标签後，在「 Preprocessor Definitions 」栏中，我已定义了识别字 UNICODE。我将在下章中对此有更多的解释。