# 第二章　Unicode 简介

在第一章中，我已经预告，C 语言中在 Microsoft Windows 程式设计中扮演著重要角色的任何部分都会讲述到，您也许在传统文字模式程式设计中还尚未遇到过这些问题。宽字元集和 Unicode 差不多就是这样的问题。

简单地说，Unicode 扩展自 ASCII 字元集。在严格的 ASCII 中，每个字元用 7 位元表示，或者电脑上普遍使用的每字元有 8 位元宽；而 Unicode 使用全 16 位元字元集。这使得 Unicode 能够表示世界上所有的书写语言中可能用於电脑通讯的字元、象形文字和其他符号。Unicode 最初打算作为 ASCII 的补充，可能的话，最终将代替它。考虑到 ASCII 是电脑中最具支配地位的标准，所以这的确是一个很高的目标。

Unicode 影响到了电脑工业的每个部分，但也许会对作业系统和程式设计语言的影响最大。从这方面来看，我们已经上路了。Windows NT 从底层支援 Unicode（不幸的是，Windows 98 只是小部分支援 Unicode）。先天即被 ANSI 束缚的 C 程式设计语言通过对宽字元集的支援来支援 Unicode。下面将详细讨论这些内容。

自然，作为程式写作者，我们通常会面对许多繁重的工作。我已试图透过使本书中的所有程式「Unicode 化」来减轻负担。其含义会随著本章对 Unicode 的讨论而清晰起来。

## 字元集简史

虽然不能确定人类开始讲话的时间，但书写已有大约 6000 年的历史了。实际上，早期书写的内容是象形文字。每个字元都对应於发声的字母表则出现於大约 3000 年前。虽然人们过去使用的多种书写语言都用得好好的，但 19 世纪的几个发明者还是看到了更多的需求。Samuel F. B. Morse 在 1838 年到 1854 年间发明了电报，当时他还发明了一种电报上使用的代码。字母表中的每个字元对应於一系列短的和长的脉冲（点和破折号）。虽然其中大小写字母之间没有区别，但数字和标点符号都有了自己的代码。

Morse 代码并不是以其他图画的或印刷的象形文字来代表书写语言的第一个例子。1821 年到 1824 年之间，年轻的 Louis Braille 受到在夜间读写资讯的军用系统的启发，发明了一种代码，它用纸上突起的点作为代码来帮助盲人阅读。Braille 代码实际上是一种 6 位元代码，它把字元、常用字母组合、常用单字和标点进行编码。一个特殊的 escape 代码表示後续的字元代码应解释为大写。

一个特殊的 shift 代码允许後续代码被解释为数字。

Telex 代码，包括 Baudot （以一个法国工程师命名，该工程师死于 1903 年）以及一种被称为 CCITT #2 的代码（1931 年被标准化），都是包括字元和数字的 5 位元代码。

## 美国标准

早期电脑的字元码是从 Hollerith 卡片（号称不能被折叠、卷曲或毁伤）发展而来的，该卡片由 Herman Hollerith 发明并首次在 1890 年的美国人口普查中使用。6 位元字元码系统 BCDIC（Binary-Coded Decimal Interchange Code：二进位编码十进位交换编码）源自 Hollerith 代码，在 60 年代逐步扩展为 8 位元 EBCDIC，并一直是 IBM 大型主机的标准，但没使用在其他地方。

美国资讯交换标准码（ASCII：American Standard Code for Information Interchange）起始於 50 年代後期，最後完成於 1967 年。开发 ASCII 的过程中，在字元长度是 6 位元、7 位元还是 8 位元的问题上产生了很大的争议。从可靠性的观点来看不应使用替换字元，因此 ASCII 不能是 6 位元编码，但由於费用的原因也排除了 8 位元版本的方案（当时每位元的储存空间成本仍很昂贵）。这样，最终的字元码就有 26 个小写字母、26 个大写字母、10 个数字、32 个符号、33 个代号和一个空格，总共 128 个字元码。ASCII 现在记录在 ANSI X3.4-1986 字元集——用於资讯交换的 7 位元美国国家标准码（7-Bit ASCII：7-Bit American National Standard Code for Information Interchange），由美国国家标准协会（American National Standards Institute）发布。图 2-1 中所示的 ASCII 字元码与 ANSI 文件中的格式相似。

ASCII 有许多优点。例如，26 个字母代码是连续的（在 EBCDIC 代码中就不是这样的）；大写字母和小写字母可通过改变一位元资料而相互转化；10 个数位的代码可从数值本身方便地得到（在 BCDIC 代码中，字元「0」的编码在字元「9」的後面！）

最棒的是，ASCII 是一个非常可靠的标准。在键盘、视讯显示卡、系统硬体、印表机、字体档案、作业系统和 Internet 上，其他标准都不如 ASCII 码流行而且根深蒂固。

| | 0- | 1- | 2- | 3- | 4- | 5- | 6- | 7- |
|---|---|---|---|---|---|---|---|---|
| -0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| -1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| -2 | STX | DC2 | ″ | 2 | B | R | b | r |
| -3 | ETX | DC3 | # | 3 | C | S | c | s |

| -4 | EOT | DC4 | $ | 4 | D | T | d | t |
|---|---|---|---|---|---|---|---|---|
| -5 | ENQ | NAK | % | 5 | E | U | e | u |
| -6 | ACK | SYN | & | 6 | F | V | f | v |
| -7 | BEL | ETB | ’ | 7 | G | W | g | w |
| -8 | BS | CAN | ( | 8 | H | X | h | x |
| -9 | HT | EM | ) | 9 | I | Y | I | y |
| -A | LF | SUB | * | : | J | Z | j | z |
| -B | VT | ESC | + | ; | K | [ | k | { |
| -C | FF | FS | , | < | L | \ | l | \| |
| -D | CR | GS | - | = | M | ] | m | } |
| -E | SO | RS | . | > | N | ^ | n | ~ |
| -F | SI | US | / | ? | O | _ | o | DEL |

图 2-1 ASCII 字元集

## 国际方面

ASCII 的最大问题就是该缩写的第一个字母。ASCII 是一个真正的美国标准，所以它不能良好满足其他讲英语国家的需要。例如英国的英镑符号（£）在哪里？

英语使用拉丁（或罗马）字母表。在使用拉丁语字母表的书写语言中，英语中的单词通常很少需要重音符号（或读音符号）。即使那些传统惯例加上读音符号也无不当的英语单字，例如 cöoperate 或者 résumé，拼写中没有读音符号也会被完全接受。

但在美国以南、以北，以及大西洋地区的许多国家，在语言中使用读音符号很普遍。这些重音符号最初是为使拉丁字母表适合这些语言读音不同的需要。在远东或西欧的南部旅游，您会遇到根本不使用拉丁字母的语言，例如希腊语、希伯来语、阿拉伯语和俄语（使用斯拉夫字母表）。如果您向东走得更远，就会发现中国象形汉字，日本和朝鲜也采用汉字系统。

ASCII 的历史开始於 1967 年，此後它主要致力於克服其自身限制以更适合於非美国英语的其他语言。例如，1967 年，国际标准化组织（ISO：International Standards Organization）推荐一个 ASCII 的变种，代码 0x40、0x5B、0x5C、0x5D、0x7B、0x7C 和 0x7D「为国家使用保留」，而代码 0x5E、0x60 和 0x7E 标为「当国内要求的特殊字元需要 8、9 或 10 个空间位置时，可用於其他图形符号」。这显然不是一个最佳的国际解决方案，因为这并不能保证一致性。但这却显示了人们如何想尽办法为不同的语言来编码的。

# 扩展 ASCII

在小型电脑开发的初期，就已经严格地建立了 8 位元位元组。因此，如果使用一个位元组来保存字元，则需要 128 个附加的字元来补充 ASCII。1981 年，当最初的 IBM PC 推出时，视讯卡的 ROM 中烧有一个提供 256 个字元的字元集，这也成为 IBM 标准的一个重要组成部分。

最初的 IBM 扩展字元集包括某些带重音的字元和一个小写希腊字母表（在数学符号中非常有用），还包括一些块型和线状图形字元。附加的字元也被添加到 ASCII 控制字元的编码位置，这是因为大多数控制字元都不是拿来显示用的。

该 IBM 扩展字元集被烧进无数显示卡和印表机的 ROM 中，并被许多应用程式用於修饰其文字模式的显示方式。不过，该字元集并没有为所有使用拉丁字母表的西欧语言提供足够多的带重音字元，而且也不适用於 Windows。Windows 不需要图形字元，因为它有一个完全图形化的系统。

在 Windows 1.0（1985 年 11 月发行）中，Microsoft 没有完全放弃 IBM 扩展字元集，但它已退居第二重要位置。因为遵循了 ANSI 草案和 ISO 标准，纯 Windows 字元集被称作「ANSI 字元集」。ANSI 草案和 ISO 标准最终成为 ANSI/ISO 8859-1-1987 ， 即 「 American National Standard for Information Processing-8-Bit Single-Byte Coded Graphic Character Sets-Part 1: Latin Alphabet No 1」，通常也简写为「Latin 1」。

在 Windows 1.0 的《Programmer's Reference》中印出了 ANSI 字元集的最初版本，如图 2-2 所示。

| 0- | 1- | 2- | 3- | 4- | 5- | 6- | 7- | 8- | 9- | A- | B- | C- | D- | E- | F- | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -0 | * | * | 0 | @ | P | ` | p | * | * | ° | À | Ð | à | ð | | |
| -1 | * | * | ! | 1 | A | Q | a | q | * | * | ¡ | ± | Á | Ñ | á | ñ |
| -2 | * | * | ″ | 2 | B | R | b | r | * | * | ¢ | ² | Â | ò | â | ò |
| -3 | * | * | # | 3 | C | S | c | s | * | * | £ | ³ | Ã | ó | ã | ó |
| -4 | * | * | $ | 4 | D | T | d | t | * | * | ¤ | ´ | Ä | ô | ä | ô |
| -5 | * | * | % | 5 | E | U | e | u | * | * | ¥ | µ | Å | õ | å | õ |
| -6 | * | * | & | 6 | F | V | f | v | * | * | ¦ | ¶ | Æ | ö | æ | ö |
| -7 | * | * | ’ | 7 | G | W | g | w | * | * | § | • | Ç | * | ç | * |
| -8 | * | * | ( | 8 | H | * | h | * | * | * | ¨ | ¸ | È | ø | è | ø |
| -9 | * | * | ) | 9 | I | Y | I | y | * | * | © | ¹ | É | Ù | é | ù |
| -A | * | * | * | : | J | Z | j | z | * | * | ª | º | Ê | Ú | ê | ú |
| -B | * | * | + | ; | K | [ | k | { | * | * | « | » | Ë | Û | ë | û |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -C | * | * | , | < | L | \ | l | \| | * | * | ¬ | ¼ | Ì | Ü | ì | ü |
| -D | * | * | - | = | M | ] | m | } | * | * | | ½ | Í | Ý | í | ý |
| -E | * | * | . | > | N | ^ | n | ~ | * | * | ® | ¾ | Î | Þ | î | þ |
| -F | * | * | / | ? | * | _ | o | del | * | * | ¯ | ¿ | Ï | ß | ï | ÿ |

* - not applicable

图 2-2 Windows ANSI 字元集（基於 ANSI/ISO 8859-1）

空方框表示该位置未定义字元。这与 ANSI/ISO 8859-1 的最终定义一致。ANSI/ISO 8859-1 仅显示了图形字元，而没有控制字元，因此没有定义 DEL。此外，代码 0xA0 定义为一个非断开的空格（这意味著在编排格式时，该字元不用於断开一行），代码 0xAD 是一个软连字元（表示除非在行尾断开单词时使用，否则不显示）。此外，ANSI/ISO 8859-1 将代码 0xD7 定义为乘号（*），0xF7 为除号（/）。Windows 中的某些字体也定义了从 0x80 到 0x9F 的某些字元，但这些不是 ANSI/ISO 8859-1 标准的一部分。

MS-DOS 3.3（1987 年 4 月发行）向 IBM PC 用户引进了内码表（code page）的概念，Windows 也使用此概念。内码表定义了字元的映射代码。最初的 IBM 字元集被称作内码表 437，或者「MS-DOS Latin US）。内码表 850 就是「MS-DOS Latin 1」，它用附加的带重音字母（但 不是 图 2-2 所示的 Latin 1 ISO/ANSI 标准）代替了一些线形字元。其他内码表被其他语言定义。最低的 128 个代码总是相同的；较高的 128 个代码取决於定义内码表的语言。

在 MS-DOS 中，如果用户为 PC 的键盘、显示卡和印表机指定了一个内码表，然後在 PC 上创建、编辑和列印文件，一切都很正常，每件事都会保持一致。然而，如果用户试图与使用不同内码表的用户交换档案，或者在机器上改变内码表，就会产生问题。字元码与错误的字元相关联。应用程式能够将内码表资讯与文件一起保存来试图减少问题的产生，但该策略包括了某些在内码表间转换的工作。

虽然内码表最初仅提供了不包括带重音符号字母的附加拉丁字元集，但最终内码表的较高的 128 个字元还是包括了完整的非拉丁字母，例如希伯来语、希腊语和斯拉夫语。自然，如此多样会导致内码表变得混乱；如果少数带重音的字母未正确显示，那么整个文字便会混乱不堪而不可阅读。

内码表的扩展正是基於所有这些原因，但是还不够。斯拉夫语的 MS-DOS 内码表 855 与斯拉夫语的 Windows 内码表 1251 以及斯拉夫语的 Macintosh 内码表 10007 不同。每个环境下的内码表都是对该环境所作的标准字元集修正。IBM OS/2 也支援多种 EBCDIC 内码表。

但等一下，你会发现事情变得更糟糕。

## 双位元组字元集

迄今为止，我们已经看到了 256 个字元的字元集。但中国、日本和韩国的象形文字符号有大约 21,000 个。如何容纳这些语言而仍保持和 ASCII 的某种相容性呢？

解决方案（如果这个说法正确的话）是双位元组字元集（DBCS：double-byte character set）。DBCS 从 256 代码开始，就像 ASCII 一样。与任何行为良好的内码表一样，最初的 128 个代码是 ASCII。然而，较高的 128 个代码中的某些总是跟随著第二个位元组。这两个位元组一起（称作首位元组和跟随位元组）定义一个字元，通常是一个复杂的象形文字。

虽然中文、日文和韩文共用一些相同的象形文字，但显然这三种语言是不同的，而且经常是同一个象形文字在三种不同的语言中代表三件不同的事。Windows 支援四个不同的双位元组字元集：内码表 932（日文）、936（简体中文）、949（韩语）和 950（繁体汉字）。只有为这些国家（地区）生产的 Windows 版本才支援 DBCS。

双字元集问题并不是说字元由两个位元组代表。问题在於一些字元（特别是 ASCII 字元）由 1 个位元组表示。这会引起附加的程式设计问题。例如，字串中的字元数不能由字串的位元组数决定。必须剖析字串来决定其长度，而且必须检查每个位元组以确定它是否为双位元组字元的首位元组。如果有一个指向 DBCS 字串中间的指标，那么该字串前一个字元的位址是什么呢？惯用的解决方案是从开始的指标分析该字串！

## Unicode 解决方案

我们面临的基本问题是世界上的书写语言不能简单地用 256 个 8 位元代码表示。以前的解决方案包括内码表和 DBCS 已被证明是不能满足需要的，而且也是笨拙的。那什么才是真正的解决方案呢？

身为程式写作者，我们经历过这类问题。如果事情太多，用 8 位元数值已经不能表示，那么我们就试更宽的值，例如 16 位元值。而且这很有趣的，正是 Unicode 被制定的原因。与混乱的 256 个字元代码映射，以及含有一些 1 位元组代码和一些 2 位元组代码的双位元组字元集不同，Unicode 是统一的 16 位元系统，这样就允许表示 65,536 个字元。这对表示所有字元及世界上使用象形文字的语言，包括一系列的数学、符号和货币单位符号的集合来说是充裕的。

明白 Unicode 和 DBCS 之间的区别很重要。Unicode 使用（特别在 C 程式设计语言环境里）「宽字元集」。「Unicode 中的每个字元都是 16 位元宽而不是

8 位元宽。」在 Unicode 中，没有单单使用 8 位元数值的意义存在。相比之下，在双位元组字元集中我们仍然处理 8 位元数值。有些位元组自身定义字元，而某些位元组则显示需要和另一个位元组共同定义一个字元。

处理 DBCS 字串非常杂乱，但是处理 Unicode 文字则像处理有秩序的文字。您也许会高兴地知道前 128 个 Unicode 字元（16 位元代码从 0x0000 到 0x007F）就是 ASCII 字元，而接下来的 128 个 Unicode 字元（代码从 0x0080 到 0x00FF）是 ISO 8859-1 对 ASCII 的扩展。Unicode 中不同部分的字元都同样基於现有的标准。这是为了便於转换。希腊字母表使用从 0x0370 到 0x03FF 的代码，斯拉夫语使用从 0x0400 到 0x04FF 的代码，美国使用从 0x0530 到 0x058F 的代码，希伯来语使用从 0x0590 到 0x05FF 的代码。中国、日本和韩国的象形文字（总称为 CJK）占用了从 0x3000 到 0x9FFF 的代码。

Unicode 的最大好处是这里只有一个字元集，没有一点含糊。Unicode 实际上是个人电脑行业中几乎每个重要公司共同合作的结果，并且它与 ISO 10646-1 标准中的代码是一一对应的。Unicode 的重要参考文献是《The Unicode Standard，Version 2.0》（Addison-Wesley 出版社，1996 年）。这是一本特别的书，它以其他文件少有的方式显示了世界上书写语言的丰富性和多样性。此外，该书还提供了开发 Unicode 的基本原理和细节。

Unicode 有缺点吗？当然有。Unicode 字串占用的记忆体是 ASCII 字串的两倍。（然而压缩档案有助於极大地减少档案所占的磁碟空间。）但也许最糟的缺点是：人们相对来说还不习惯使用 Unicode。身为程式写作者，这就是我们的工作。

## 宽字元和 C

对 C 程式写作者来说，16 位元字元的想法的确让人扫兴。一个 char 和一个位元组同宽是最不能确定的事情之一。没几个程式写作者清楚 ANSI/ISO 9899-1990，这是「美国国家标准程式设计语言——C」（也称作「ANSI C」）通过一个称作「宽字元」的概念来支援用多个位元组代表一字元的字元集。这些宽字元与常用的字元完美地共存。

ANSI C 也支援多位元组字元集，例如中文、日文和韩文版本 Windows 支援的字元集。然而，这些多位元组字元集被当成单位元组构成的字串看待，只不过其中一些字元改变了後续字元的含义而已。多位元组字元集主要影响 C 语言程式执行时期程式库函式。相比之下，宽字元比正常字元宽，而且会引起一些编译问题。

宽字元不需要是 Unicode。Unicode 是一种可能的宽字元集。然而，因为本

书的焦点是 Windows 而不是 C 执行的理论，所以我将把宽字元和 Unicode 作为同义语。

## char 资料型态

假定我们都非常熟悉在 C 程式中使用 char 资料型态来定义和储存字元跟字串。但为了便於理解 C 如何处理宽字元，让我们先回顾一下可能在 Win32 程式中出现的标准字元定义。

下面的语句定义并初始化了一个只包含一个字元的变数：

```
char c = 'A' ;
```

变数 c 需要 1 个位元组来保存，并将用十六进位数 0x41 初始化，这是字母 A 的 ASCII 代码。

您可以像这样定义一个指向字串的指标：

```
char * p ;
```

因为 Windows 是一个 32 位元作业系统，所以指标变数 p 需要用 4 个位元组保存。您还可初始化一个指向字串的指标：

```
char * p = "Hello!" ;
```

像前面一样，变数 p 也需要用 4 个位元组保存。该字串保存在静态记忆体中并占用 7 个位元组——6 个位元组保存字串，另 1 个位元组保存终止符号 0。

您还可以像这样定义字元阵列：

```
char a[10] ;
```

在这种情况下，编译器为该阵列保留了 10 个位元组的储存空间。运算式 sizeof（a） 将返回 10。如果阵列是整体变数（即在所有函式外定义），您可使用像下面的语句来初始化一个字元阵列：

```
char a[] = "Hello!" ;
```

如果您将该阵列定义为一个函式的区域变数，则必须将它定义为一个 static 变数，如下：

```
static char a[] = "Hello!" ;
```

无论哪种情况，字串都储存在静态程式记忆体中，并在末尾添加 0，这样就需要 7 个位元组的储存空间。

## 宽字元

Unicode 或者宽字元都没有改变 char 资料型态在 C 中的含义。char 继续表示 1 个位元组的储存空间， sizeof （char） 继续返回 1。理论上，C 中 1 个位元组可比 8 位元长，但对我们大多数人来说，1 个位元组（也就是 1 个 char）是 8 位元宽。

C 中的宽字元基於 wchar_t 资料型态，它在几个表头档案包括 WCHAR.H 中都有定义，像这样：

```
typedef unsigned short wchar_t ;
```

因此，wchar_t 资料型态与无符号短整数型态相同，都是 16 位元宽。

要定义包含一个宽字元的变数，可使用下面的语句：

```
wchar_t c = 'A' ;
```

变数 c 是一个双位元组值 0x0041，是 Unicode 表示的字母 A。（然而，因为 Intel 微处理器从最小的位元组开始储存多位元组数值，该位元组实际上是以 0x41、0x00 的顺序保存在记忆体中。如果检查 Unicode 文字的电脑储存应注意这一点。）

您还可定义指向宽字串的指标：

```
wchar_t * p = L"Hello!" ;
```

注意紧接在第一个引号前面的大写字母 L（代表「long」）。这将告诉编译器该字串按宽字元保存——即每个字元占用 2 个位元组。通常，指标变数 p 要占用 4 个位元组，而字串变数需要 14 个位元组——每个字元需要 2 个位元组，末尾的 0 还需要 2 个位元组。

同样，您还可以用下面的语句定义宽字元阵列：

```
static wchar_t a[] = L"Hello!" ;
```

该字串也需要 14 个位元组的储存空间，sizeof (a) 将返回 14。索引阵列 a 可得到单独的字元。a[1] 的值是宽字元「e」，或者 0x0065。

虽然看上去更像一个印刷符号，但第一个引号前面的 L 非常重要，并且在两个符号之间必须没有空格。只有带有 L，编译器才知道您需要将字串存为每个字元 2 位元组。稍後，当我们看到使用宽字串而不是变数定义时，您还会遇到第一个引号前面的 L。幸运的是，如果忘记了包含 L，C 编译器通常会给提出警告或错误资讯。

您还可在单个字元文字前面使用 L 字首，来表示它们应解释为宽字元。如下所示：

```
wchar_t c = L'A' ;
```

但通常这是不必要的，C 编译器会对该字元进行扩充，使它成为宽字元。

## 宽字元程式库函式

我们都知道如何获得字串的长度。例如，如果我们已经像下面这样定义了一个字串指标：

```
char * pc = "Hello!" ;
```

我们可以呼叫

```
iLength = strlen (pc) ;
```

这时变数 iLength 将等於 6，也就是字串中的字元数。

太好了！现在让我们试著定义一个指向宽字元的指标：

```
wchar_t * pw = L"Hello!" ;
```

再次呼叫 strlen ：

```
iLength = strlen (pw) ;
```

现在麻烦来了。首先，C 编译器会显示一条警告消息，可能是这样的内容：

```
'function' : incompatible types - from 'unsigned short *' to 'const char *'
```

这条消息的意思是：宣告 strlen 函式时，该函式应接收 char 类型的指标，但它现在却接收了一个 unsigned short 类型的指标。您仍然可编译并执行该程式，但您会发现 iLength 等於 1。为什么？

字串「Hello!」中的 6 个字元占用 16 位元：

0x0048 0x0065 0x006C 0x006C 0x006F 0x0021

Intel 处理器在记忆体中将其存为：

48 00 65 00 6C 00 6C 00 6F 00 21 00

假定 strlen 函式正试图得到一个字串的长度，并把第 1 个位元组作为字元开始计数，但接著假定如果下一个位元组是 0，则表示字串结束。

这个小练习清楚地说明了 C 语言本身和执行时期程式库函式之间的区别。编译器将字串 L″Hello!″ 解释为一组 16 位元短整数型态资料，并将其保存在 wchar_t 阵列中。编译器还处理阵列索引和 sizeof 操作符，因此这些都能正常工作，但在连结时才添加执行时期程式库函式，例如 strlen。这些函式认为字串由单位元组字元组成。遇到宽字串时，函式就不像我们所希望那样执行了。

您可能要说：「噢，太麻烦了！」现在每个 C 语言程式库函式都必须重写以接受宽字元。但事实上并不是每个 C 语言程式库函式都需要重写，只是那些有字串参数的函式才需要重写，而且也不用由您来完成。它们已经重写完了。

strlen 函式的宽字元版是 wcslen（wide-character string length：宽字串长度），并且在 STRING.H（其中也说明了 strlen）和 WCHAR.H 中均有说明。strlen 函式说明如下：

```
size_t __cdecl strlen (const char *) ;
```

而 wcslen 函式则说明如下：

```
size_t __cdecl wcslen (const wchar_t *) ;
```

这时我们知道，要得到宽字串的长度可以呼叫

```
iLength = wcslen (pw) ;
```

函式将返回字串中的字元数 6。请记住，改成宽位元组後，字串的字元长度不改变，只是位元组长度改变了。

您熟悉的所有带有字串参数的 C 执行时期程式库函式都有宽字元版。例如，wprintf 是 printf 的宽字元版。这些函式在 WCHAR.H 和含有标准函式说明的表

头档案中说明。

## 维护单一原始码

当然，使用 Unicode 也有缺点。第一点也是最主要的一点是，程式中的每个字串都将占用两倍的储存空间。此外，您将发现宽字元执行时期程式库中的函式比常规的函式大。出於这个原因，您也许想建立两个版本的程式——一个处理 ASCII 字串，另一个处理 Unicode 字串。最好的解决办法是维护既能按 ASCII 编译又能按 Unicode 编译的单一原始码档案。

虽然只是一小段程式，但由於执行时期程式库函式有不同的名称，您也要定义不同的字元，这将在处理前面有 L 的字串文字时遇到麻烦。

一个办法是使用 Microsoft Visual C++包含的 TCHAR.H 表头档案。该表头档案不是 ANSI C 标准的一部分，因此那里定义的每个函式和巨集定义的前面都有一条底线。TCHAR.H 为需要字串参数的标准执行时期程式库函式提供了一系列的替代名称（例如，_tprintf 和_tcslen）。有时这些名称也称为「通用」函式名称，因为它们既可以指向函式的 Unicode 版也可以指向非 Unicode 版。

如果定义了名为_UNICODE 的识别字，并且程式中包含了 TCHAR.H 表头档案，那么_tcslen 就定义为 wcslen：

```
#define _tcslen wcslen
```

如果没有定义 UNICODE，则_tcslen 定义为 strlen：

```
#define _tcslen strlen
```

等等。TCHAR.H 还用一个新的资料型态 TCHAR 来解决两种字元资料型态的问题。如果定义了 _UNICODE 识别字，那么 TCHAR 就是 wchar_t：

```
typedef wchar_t TCHAR ;
```

否则，TCHAR 就是 char：

```
typedef char TCHAR ;
```

现在开始讨论字串文字中的 L 问题。如果定义了_UNICODE 识别字，那么一个称作__T 的巨集就定义如下：

```
#define __T(x) L##x
```

这是相当晦涩的语法，但合乎 ANSI C 标准的前置处理器规范。那一对井字号称为「粘贴符号（token paste）」，它将字母 L 添加到巨集引数上。因此，如果巨集引数是"Hello!"，则 L##x 就是 L"Hello!"。

如果没有定义_UNICODE 识别字，则__T 巨集只简单地定义如下：

```
#define __T(x) x
```

此外，还有两个巨集与__T 定义相同：

```
#define _T(x)  __T(x)
#define _TEXT(x)  __T(x)
```

在 Win32 console 程式中使用哪个巨集，取决於您喜欢简洁还是详细。基本地，必须按下述方法在_T 或_TEXT 巨集内定义字串文字：

```
_TEXT ("Hello!")
```

这样做的话，如果定义了_UNICODE，那么该串将解释为宽字元的组合，否则解释为 8 位元的字元字串。

## 宽字元和 WINDOWS

Windows NT 从底层支援 Unicode。这意味著 Windows NT 内部使用由 16 位元字元组成的字串。因为世界上其他许多地方还不使用 16 位元字串，所以 Windows NT 必须经常将字串在作业系统内转换。Windows NT 可执行为 ASCII、Unicode 或者 ASCII 和 Unicode 混合编写的程式。即，Windows NT 支援不同的 API 函式呼叫，这些函式接受 8 位元或 16 位元的字串（我们将马上看到这是如何动作的。）

相对於 Windows NT，Windows 98 对 Unicode 的支援要少得多。只有很少的 Windows 98 函式呼叫支援宽字串（这些函式列在《Microsoft Knowledge Base article Q125671》中;它们包括 MessageBox）。如果要发行的程式中只有一个.EXE 档案要求在 Windows NT 和 Windows 98 下都能执行，那么就不应该使用 Unicode，否则就不能在 Windows 98 下执行；尤其程式不能呼叫 Unicode 版的 Windows 函式。这样，将来发行 Unicode 版的程式时会处於更有利的位置，您应试著编写既为 ASCII 又为 Unicode 编译的原始码。这就是本书中所有程式的编写方式。

## Windows 表头档案类型

正如您在第一章所看到的那样，一个 Windows 程式包括表头档案 WINDOWS.H。该档案包括许多其他表头档案，包括 WINDEF.H，该档案中有许多在 Windows 中使用的基本型态定义，而且它本身也包括 WINNT.H。WINNT.H 处理基本的 Unicode 支援。

WINNT.H 的前面包含 C 的表头档案 CTYPE.H，这是 C 的众多表头档案之一，包括 wchar_t 的定义。WINNT.H 定义了新的资料型态，称作 CHAR 和 WCHAR：

```
typedef char CHAR ;
typedef wchar_t WCHAR ;      // wc
```

当您需要定义 8 位元字元或者 16 位元字元时，推荐您在 Windows 程式中使用的资料型态是 CHAR 和 WCHAR。WCHAR 定义後面的注释是匈牙利标记法的建议：一个基於 WCHAR 资料型态的变数可在前面附加上字母 wc 以说明一个宽字元。

WINNT.H表头档案进而定义了可用做8位元字串指标的六种资料型态和四个可用做 const 8 位元字串指标的资料型态。这里精选了表头档案中一些实用的

说明资料型态语句：

```
typedef CHAR * PCHAR, * LPCH, * PCH, * NPSTR, * LPSTR, * PSTR ;
typedef CONST CHAR * LPCCH, * PCCH, * LPCSTR, * PCSTR ;
```

字首 N 和 L 表示「near」和「long」，指的是 16 位元 Windows 中两种大小不同的指标。在 Win32 中 near 和 long 指标没有区别。

类似地，WINNT.H 定义了六种可作为 16 位元字串指标的资料型态和四种可作为 const 16 位元字串指标的资料型态：

```
typedef WCHAR * PWCHAR, * LPWCH, * PWCH, * NWPSTR, * LPWSTR, * PWSTR ;
typedef CONST WCHAR * LPCWCH, * PCWCH, * LPCWSTR, * PCWSTR ;
```

至此，我们有了资料型态 CHAR（一个 8 位的 char）和 WCHAR（一个 16 位的 wchar_t），以及指向 CHAR 和 WCHAR 的指标。与 TCHAR.H 一样，WINNT.H 将 TCHAR 定义为一般的字元类型。如果定义了识别字 UNICODE（没有底线），则 TCHAR 和指向 TCHAR 的指标就分别定义为 WCHAR 和指向 WCHAR 的指标；如果没有定义识别字 UNICODE，则 TCHAR 和指向 TCHAR 的指标就分别定义为 char 和指向 char 的指标：

```
#ifdef  UNICODE
typedef WCHAR TCHAR, * PTCHAR ;
typedef LPWSTR LPTCH, PTCH, PTSTR, LPTSTR ;
typedef LPCWSTR LPCTSTR ;
#else
typedef char TCHAR, * PTCHAR ;
typedef LPSTR LPTCH, PTCH, PTSTR, LPTSTR ;
typedef LPCSTR LPCTSTR ;
#endif
```

如果已经在某个表头档案或者其他表头档案中定义了 TCHAR 资料型态，那么 WINNT.H 和 WCHAR.H 表头档案都能防止其重复定义。不过，无论何时在程式中使用其他表头档案时，都应在所有其他表头档案之前包含 WINDOWS.H。

WINNT.H 表头档案还定义了一个巨集，该巨集将 L 添加到字串的第一个引号前。如果定义了 UNICODE 识别字，则一个称作 __TEXT 的巨集定义如下：

```
#define __TEXT(quote) L##quote
```

如果没有定义识别字 UNICODE，则像这样定义__TEXT 巨集：

```
#define __TEXT(quote) quote
```

此外， TEXT 巨集可这样定义：

```
#define TEXT(quote) __TEXT(quote)
```

这与 TCHAR.H 中定义_TEXT 巨集的方法一样，只是不必操心底线。我将在本书中使用这个巨集的 TEXT 版本。

这些定义可使您在同一程式中混合使用 ASCII 和 Unicode 字串，或者编写一个可被 ASCII 或 Unicode 编译的程式。如果您希望明确定义 8 位元字元变数和字串，请使用 CHAR、PCHAR（或者其他），以及带引号的字串。为明确地使用

16 位元字元变数和字串，请使用 WCHAR、PWCHAR，并将 L 添加到引号前面。对於是 8 位还是 16 位取决於 UNICODE 识别字的定义的变数或字串，要使用 TCHAR、PTCHAR 和 TEXT 巨集。

## Windows 函式呼叫

从 Windows 1.0 到 Windows 3.1 的 16 位元 Windows 中，MessageBox 函式位於动态连结程式库 USER.EXE。在 Windows 3.1 软体开发套件的 WINDOWS.H 中，MessageBox 函式定义如下：

```
int WINAPI MessageBox (HWND, LPCSTR, LPCSTR, UINT) ;
```

注意，函式的第二个、第三个参数是指向常数字串的指标。当编译连结一个 Win16 程式时，Windows 并不处理 MessageBox 呼叫。程式.EXE 档案中的表格，允许 Windows 将该程式的呼叫与 USER 中的 MessageBox 函式动态连结起来。

32 位的 Windows（即所有版本的 Windows NT，以及 Windows 95 和 Windows 98）除了含有与 16 位相容的 USER.EXE 以外，还含有一个称为 USER32.DLL 的动态连结程式库，该动态连结程式库含有 32 位元使用者介面函式的进入点，包括 32 位元的 MessageBox。

这就是 Windows 支援 Unicode 的关键：在 USER32.DLL 中，没有 32 位元 MessageBox 函式的进入点。实际上，有两个进入点，一个名为 MessageBoxA（ASCII 版），另一个名为 MessageBoxW（宽字元版）。用字串作参数的每个 Win32 函式都在作业系统中有两个进入点！幸运的是，您通常不必关心这个问题，程式中只需使用 MessageBox。与 TCHAR 表头档案一样，每个 Windows 表头档案都有我们需要的技巧。

下面是 MessageBoxA 在 WINUSER.H 中定义的方法。这与 MessageBox 早期的定义很相似：

```
WINUSERAPI int WINAPI MessageBoxA (     HWND hWnd, LPCSTR lpText,
                        LPCSTR lpCaption, UINT uType) ;
```

下面是 MessageBoxW：

```
WINUSERAPI int WINAPI MessageBoxW (HWND hWnd, LPCWSTR lpText,
                        LPCWSTR lpCaption, UINT uType) ;
```

注意，MessageBoxW 函式的第二个和第三个参数是指向宽字元的指标。

如果需要同时使用并分别匹配 ASCII 和宽字元函式呼叫，那么您可在 Windows 程式中明确地使用 MessageBoxA 和 MessageBoxW 函式。但大多数程式写作者将继续使用 MessageBox。根据是否定义了 UNICODE，MessageBox 将与 MessageBoxA 或 MessageBoxW 一样。在 WINUSER.H 中完成这一技巧时，程式相当琐碎：

```
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif
```

这样，如果定义了 UNICODE 识别字，那么程式中所有的 MessageBox 函式呼叫实际上就是 MessageBoxW 函式；否则，就是 MessageBoxA 函式。

执行该程式时，Windows 将程式中不同的函式呼叫与不同的 Windows 动态连结程式库的进入点连结。虽然只有少数例外，但是，在 Windows 98 中不能执行 Unicode 版的 Windows 函式。虽然这些函式有进入点，但通常返回错误代码。应用程式注意这些返回的错误并采取一些合理的动作。

## Windows 的字串函式

正如前面谈到的，Microsoft C 包括宽字元和需要字串参数的 C 语言执行时期程式库函式的所有普通版本。不过，Windows 复制了其中一部分。例如，下面是 Windows 定义的一组字串函式，这些函式用来计算字串长度、复制字串、连接字串和比较字串：

```
ILength = lstrlen (pString) ;
pString = lstrcpy (pString1, pString2) ;
pString = lstrcpyn (pString1, pString2, iCount) ;
pString = lstrcat (pString1, pString2) ;
iComp = lstrcmp (pString1, pString2) ;
iComp = lstrcmpi (pString1, pString2) ;
```

这些函式与 C 程式库中对应的函式功能相同。如果定义了 UNICODE 识别字，那么这些函式将接受宽字串，否则只接受常规字串。宽字串版的 lstrlenW 函式可在 Windows 98 中执行。

## 在 Windows 中使用 printf

有文字模式、命令列C语言程式写作历史的程式写作者往往特别喜欢printf函式。即使可以使用更简单的命令（例如 puts），但 printf 出现在 Kernighan 和 Ritchie 的「hello, world」程式中一点也不会令人惊奇。我们知道，增强後的「hello, world」最终还是需要 printf 的格式化输出，因此我们最好从头开始就使用它。

但有个坏消息：在 Windows 程式中不能使用 printf。虽然 Windows 程式中可以使用大多数 C 的执行时期程式库——实际上，许多程式写作者更愿意使用 C 记忆体管理和档案 I/O 函式而不是 Windows 中等效的函式——Windows 对标准输入和标准输出没有概念。在 Windows 程式中可使用 fprintf，而不是 printf。

还有一个好消息，那就是仍然可以使用 sprintf 及 sprintf 系列中的其他函式来显示文字。这些函式除了将内容格式化输出到函式第一个参数所提供的字串缓冲区以外，其功能与 printfI 相同。然後便可对该字串进行操作（例如将其传给 MessageBox）。

如果您从未使用过 sprintf （我第一次开始写 Windows 程式时也没用过此函式），这里有一个简短的执行实体，printf 函式说明如下：

```
int printf (const char * szFormat, ...) ;
```

第一个参数是一个格式字串，後面是与格式字串中的代码相对应的不同类型多个参数。

sprintf 函式定义如下：

```
int sprintf (char * szBuffer, const char * szFormat, ...) ;
```

第一个参数是字元缓冲区；後面是一个格式字串。Sprintf 不是将格式化结果标准输出，而是将其存入 szBuffer。该函式返回该字串的长度。在文字模式程式设计中，

```
printf ("The sum of %i and %i is %i", 5, 3, 5+3) ;
```

的功能相同於

```
char szBuffer [100] ;
sprintf (szBuffer, "The sum of %i and %i is %i", 5, 3, 5+3) ;
puts (szBuffer) ;
```

在 Windows 中，使用 MessageBox 显示结果优於 puts。

几乎每个人都经历过，当格式字串与被格式化的变数不合时，可能使 printf 执行错误并可能造成程式当掉。使用 sprintf 时，您不但要担心这些，而且还有一个新的负担：您定义的字串缓冲区必须足够大以存放结果。Microsoft 专用函式_snprintf 解决了这一问题，此函式引进了另一个参数，表示以字元计算的缓冲区大小。

vsprintf 是 sprintf 的一个变形，它只有三个参数。vsprintf 用於执行有多个参数的自订函式，类似 printf 格式。vsprintf 的前两个参数与 sprintf 相同：一个用於保存结果的字元缓冲区和一个格式字串。第三个参数是指向格式化参数阵列的指标。实际上，该指标指向在堆叠中供函式呼叫的变数。va_list、va_start 和 va_end 巨集（在 STDARG.H 中定义）帮助我们处理堆叠指标。本章最後的 SCRNSIZE 程式展示了使用这些巨集的方法。使用 vsprintf 函式，sprintf 函式可以这样编写：

```
int sprintf (char * szBuffer, const char * szFormat, ...)
{
     int    iReturn ;
     va_list pArgs ;
     va_start (pArgs, szFormat) ;
     iReturn = vsprintf (szBuffer, szFormat, pArgs) ;
```

```
     va_end (pArgs) ;
     return iReturn ;
}
```

va_start 巨集将 pArg 设置为指向一个堆叠变数，该变数位址在堆叠参数 szFormat 的上面。

由於许多 Windows 早期程式使用了 sprintf 和 vsprintf，最终导致 Microsoft 向 Windows API 中增添了两个相似的函式。Windows 的 wsprintf 和 wvsprintf 函式在功能上与 sprintf 和 vsprintf 相同，但它们不能处理浮点格式。

当然，随著宽字元的发表，sprintf 类型的函式增加许多，使得函式名称变得极为混乱。表 2-1 列出了 Microsoft 的 C 执行时期程式库和 Windows 支援的所有 sprintf 函式。

表 2-1

| | ASCII | 宽字元 | 常规 |
|---|---|---|---|
| 参数的变数个数 | | | |
| 标准版 | sprintf | swprintf | _stprintf |
| 最大长度版 | _snprintf | _snwprintf | _sntprintf |
| Windows 版 | wsprintfA | wsprintfW | wsprintf |
| 参数阵列的指标 | | | |
| 标准版 | vsprintf | vswprintf | _vstprintf |
| 最大长度版 | _vsnprintf | _vsnwprintf | _vsntprintf |
| Windows 版 | wvsprintfA | wvsprintfW | wvsprintf |

在宽字元版的 sprintf 函式中，将字串缓冲区定义为宽字串。在宽字元版的所有这些函式中，格式字串必须是宽字串。不过，您必须确保传递给这些函式的其他字串也必须由宽字元组成。

## 格式化讯息方块

程式 2-1 所示的 SCRNSIZE 程式展示了如何实作 MessageBoxPrintf 函式，该函式有许多参数并能像 printf 那样编排它们的格式。

程式 2-1　SCRNSIZE

```
SCRNSIZE.C
/*-------------------------------------------------------------------------
-
     SCRNSIZE.C --    Displays screen size in a message box
                (c) Charles Petzold, 1998
```

```
------------------------------------------------------------------------------
*/
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int CDECL MessageBoxPrintf (TCHAR * szCaption, TCHAR * szFormat, ...)
{
      TCHAR   szBuffer [1024] ;
      va_list pArgList ;

      // The va_start macro (defined in STDARG.H) is usually equivalent to:
      // pArgList = (char *) &szFormat + sizeof (szFormat) ;

      va_start (pArgList, szFormat) ;

      // The last argument to wvsprintf points to the arguments

      _vsntprintf (    szBuffer, sizeof (szBuffer) / sizeof (TCHAR),
                  szFormat, pArgList) ;

      // The va_end macro just zeroes out pArgList for no good reason
      va_end (pArgList) ;
      return MessageBox (NULL, szBuffer, szCaption, 0) ;
}
int WINAPI WinMain (   HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
      int cxScreen, cyScreen ;
      cxScreen = GetSystemMetrics (SM_CXSCREEN) ;
      cyScreen = GetSystemMetrics (SM_CYSCREEN) ;

      MessageBoxPrintf (     TEXT ("ScrnSize"),
                  TEXT ("The screen is %i pixels wide by %i pixels high."),
                  cxScreen, cyScreen) ;
      return 0 ;
}
```

经由从 GetSystemMetrics 函式得到的资讯，该程式以图素为单位显示了视讯显示的宽度和高度。GetSystemMetrics 是一个能用来获得 Windows 中不同物件的尺寸资讯的函式。事实上，我将在第四章用 GetSystemMetrics 函式向您展示如何在一个 Windows 视窗中显示和滚动多行文字。

## 本书与国际化

为国际市场准备的 Windows 程式不光要使用 Unicode。国际化超出了本书的范围，但在 Nadine Kano 所写的《Developing International Software for

Windows 95 and Windows NT》（Microsoft Press，1995 年）一书中涉猎了许多。

本书中的程式写作时被限制成既可使用也可不使用定义的 UNICODE 识别字来编译。这包括对所有字元和字串定义使用 TCHAR，对字串文字使用 TEXT 巨集，以及注意不要混淆位元组和字元。例如，注意 SCRNSIZE 中的 _vsntprintf 呼叫。第二个参数是缓冲区的字元大小。通常，您使用 sizeof （szBuffer）。但如果缓冲区中有宽字元，则返回的不是缓冲区的字元长度，而是缓冲区的位元组大小。您必须用 sizeof（TCHAR）将其分开。

通常，在 Visual C++ Developer Studio 中，可使用两种不同的设定来编译程式：Debug 和 Release。为简便起见，对本书的范例程式，我已修改了 Debug 设定，以便於定义 UNICODE 识别字。如果程式使用了需要字串作参数的 C 程式库函式，那么_UNICODE 识别字也在 Debug 设定中定义（要了解这是在哪里完成的，请从「Project」功能表中选择「Settings」，然後单击「C/C++」标签）。使用这种方式，这些程式就可以方便地被重新编译和连结以供测试。

本书中所有程式——无论是否为 Unicode 编译——都可以在 Windows NT 下执行。只有极少数情况例外。本书中按 Unicode 编译的程式不能在 Windows 98 中执行，而非 Unicode 版则可以。本章和第一章的程式就是两个特例。MessageBoxW 是 Windows 98 支援的少数宽字元 Windows 函式之一。在 SCRNSIZE.C 中，如果用 Windows 函式 wprintf 代替了_vsntprintf（您还必须删除该函式的第二个参数），那么 SCRNSIZE.C 的 Unicode 版将不能在 Windows 98 下执行，这是因为 Windows 98 不支援 wprintfW。

在本书的後面（特别在第六章，介绍键盘的使用时），我们将看到，编写能处理远东版 Windows 双字元集的 Windows 程式不是一件容易的事情。本书没有说明如何去做，并且基於这个原因，本书中的某些非 Unicode 版本的程式在远东版的 Windows 下不能正常执行。这也是 Unicode 对将来的程式设计如此重要的一条理由。Unicode 允许程式更容易地跨越国界。