Contents

Visual C++ 中的文字和字串

2020/3/25 • • Edit Online

針對國際市場開發應用程式的重要層面，就是適當的本機字元集。ASCII 字元集會定義0x00 到0x7F 範圍內的字元。還有其他字元集（主要是歐洲），它會定義介於0x00 到0x7F 範圍內的字元與 ASCII 字元集相同，而且也會定義從 0x80 到0xFF 的擴充字元集。因此，8位的單位元組字元集（SBCS）就足以代表 ASCII 字元集，以及許多歐洲語言的字元集（英文）。不過，有些非歐洲字元集（例如日文漢字）所包含的字元比單一位元組編碼配置所能表示的還要多，因此需要多位元組字元集（MBCS）編碼。

本節內容

Unicode 和 MBCS

討論 Unicode C++和 MBCS 程式設計的視覺支援。

Unicode 的支援

描述 Unicode，這是支援所有字元集的規格，包括無法在單一位元組中表示的字元集。

多位元組字元集（MBCS）的支援

討論 MBCS，這是 Unicode 用來支援字元集的替代方法，例如日文和中文，無法以單一位元組表示。

tchar.h 中的泛型文字對應

針對許多資料類型、常式和其他物件，提供 Microsoft 特定的泛型文字對應。

如何：在各種字串類型之間轉換

示範如何將各種視覺C++字串類型轉換成其他字串。

相關章節

國際化

討論 C 執行時間程式庫中的國際支援。

國際範例

提供在視覺效果C++中示範國際化之範例的連結。

語言和國家/地區字串

提供 C 執行時間程式庫中的語言和國家/地區字串。

Unicode 和 MBCS

2020/12/12 • • Edit Online

Microsoft Foundation 類別 (MFC) 程式庫、適用于 Visual C++ 的 C 執行時間程式庫，以及啟用 Visual C++ 開發環境以協助您進行國際化程式設計。它們提供：

- Windows 上的 Unicode 標準支援。Unicode 是目前的標準，應該盡可能使用此標準。

  Unicode 是16位字元編碼，提供適用于所有語言的足夠編碼。所有 ASCII 字元都包含在 Unicode 中做為加寬的字元。

- 支援所有平臺上的多位元組字元集 (MBCS) 稱為雙位元組字集 (DBCS) 。

  DBCS 字元是由1或2個位元組所組成。某些位元組範圍會被設定為使用於前導位元組。前導位元組指定它和下列尾位元組組成單一雙位元組寬字元。您必須追蹤哪些位元組是前置位元組。在特定多位元組字元集中，前導位元組落在特定範圍內，後隨位元組也是如此。當這些範圍重迭時，可能需要評估內容，以判斷指定的位元組是否可做為前導位元組或後隨位元組。

- 支援工具，可簡化針對國際市場撰寫之應用程式的 MBCS 程式設計。

  在啟用 MBCS 的 Windows 作業系統版本上執行時，Visual C++ 的開發系統(包括整合式原始程式碼編輯器、偵錯工具和命令列工具)完全是 MBCS 啟用的。如需詳細資訊，請參閱 Visual C++ 中的 MBCS 支援。

> **NOTE**
> 在此檔中，MBCS 是用來描述多位元組字元的所有非 Unicode 支援。在 Visual C++ 中，MBCS 一律表示 DBCS。不支援超出2個位元組的字元集。

根據定義，ASCII 字元集是所有多位元組字元集的子集。在許多多位元組字元集中，0x00 - 0x7F 範圍中的每個字元都會與 ASCII 字元集中具有相同值的字元相同。例如，在 ASCII 和 MBCS 字元字串中，1位元組的 Null 字元 (' \ 0 ') 具有值0x00，並指出終止的 Null 字元。

請參閱

文字和字串
啟用國際

啟用國際化

2020/12/12 • • Edit Online

大部分傳統的 C 和 c + + 程式碼，都是針對國際應用程式無法正常運作的字元和字串操作所提出的假設。雖然 MFC 和執行時間程式庫都支援 Unicode 或 MBCS，但仍可讓您執行此作業。為了引導您，本節說明 Visual C++ 中「國際啟用」的意義：

- Unicode 和 MBCS 都是透過 MFC 函式參數清單和傳回型別中的可移植資料類型來啟用。這些類型是有條件地定義的，取決於您的組建是否定義符號 `_UNICODE` 或符號 `_MBCS` (這表示 DBCS) 。MFC 程式庫的不同變化會自動與您的應用程式連結，這取決於您的組建所定義的這兩個符號。
- 類別庫程式碼使用可移植的執行時間函式和其他方法來確保正確的 Unicode 或 MBCS 行為。
- 您仍然必須在程式碼中處理特定種類的國際化工作：
  - 使用在任一環境下使 MFC 可移植的相同可移植執行時間函數。
  - 使用宏，讓常值字串和字元可以在任一環境下移植 `_T` 。如需詳細資訊，請參閱 tchar 中的泛型文字對應。
  - 剖析 MBCS 下的字串時，請採取預防措施。Unicode 不需要這些預防措施。如需詳細資訊，請參閱 MBCS 程式設計秘訣。
  - 如果您在應用程式中混用 ANSI (8 位) 和 Unicode (16 位) 字元，請務必小心。您可以在程式的某些部分使用 ANSI 字元，並在其他部分使用 Unicode 字元，但不能將它們混合在相同的字串中。
  - 請勿在您的應用程式中硬式編碼字串。相反地，請將它們新增至應用程式的 .rc 檔，使其 STRINGTABLE 資源。然後您可以當地語系化應用程式，而不需要變更或重新編譯原始程式碼。如需 STRINGTABLE 資源的詳細資訊，請參閱 字串編輯器。

> **NOTE**
> 歐洲和 MBCS 字元集有一些字元，例如重音字母，字元碼大於0x80。因為大部分的程式碼都使用帶正負號的字元，所以在轉換為時，會將這些大於0x80 的字元延伸 `int` 。這是陣列索引的問題，因為符號擴充的字元(負)是陣列外部的索引。使用 MBCS 的語言(例如日文)也是唯一的。因為字元可能包含1或2個位元組，所以您應該一律同時操作這兩個位元組。

請參閱

Unicode 和 MBCS
國際化策略

國際化策略

2019/12/2 • • Edit Online

根據您的目標作業系統和市場，您會有數個國際化策略：

- 您的應用程式會使用 Unicode。

  使用 Unicode 特有的功能，且所有字元都是 16 位元寬（雖然您可以使用在您的程式的某些部分的 ANSI 字元，供特殊目的）。C 執行階段程式庫會提供僅限 Unicode 的程式設計中的函式、巨集和資料類型。MFC 是完全啟用 Unicode。

- 您的應用程式會使用 MBCS，而且可以執行任何 Win32 平台上。

  您使用 MBCS 特有的功能。字串可以包含單一位元組字元、雙位元組字元，或兩者。C 執行階段程式庫會提供僅限 MBCS 程式設計中的函式、巨集和資料類型。MFC 是完全 MBCS 啟用。

- 您的應用程式的原始程式碼寫入完整的可攜性，藉由重新編譯使用的符號 `_UNICODE` 或符號 `_MBCS` 定義，您可以產生使用的版本。如需詳細資訊，請參閱 < tchar.h 中的泛型文字對應。

  您使用完全可攜 C 執行階段函式、巨集和資料類型。MFC 的彈性支援任何這些策略。

這些主題的其餘部分會專注於撰寫完全的可攜式程式碼，您可以建置為 Unicode 或 MBCS。

另請參閱

Unicode 和 MBCS
地區設定和字碼頁

地區設定和字碼頁

2020/12/12 • • Edit Online

地區設定識別碼會反映特定地理區域的本機慣例和語言。一種指定語言可以在一個以上的國家/地區使用，例如巴西和葡萄牙都說葡萄牙語。反過來說，一個國家/地區可能有一種以上的官方語言。例如，加拿大有兩種語言：英文和法文。因此，加拿大有兩種不同的地區設定：Canadian-English 和加拿大法文。有些與地區設定相關的類別含有日期格式和貨幣值的顯示格式。

語言決定文字和日期格式的轉換，而國家/地區決定當地慣例。每種語言都有唯一的對應（以字碼頁表示），包括字母 (中的字元，例如標點符號和數位) 。字碼頁是字元集，與語言相關。因此，地區設定是語言 、國家/地區和字碼頁的唯一組合。您可以藉由呼叫 setlocale 函數，在執行時間變更地區設定和字碼頁設定。

不同的語言可能會使用不同的字碼頁。例如，ANSI 字碼頁1252用於英文和大部分的歐洲語言，而 ANSI 字碼頁932則用於日文漢字。幾乎所有的字碼頁都會共用最低128字元的 ASCII 字元集 (0x00 至 0x7F) 。

任何單一位元組字碼頁都可以在資料表中表示，(包含256個專案) 作為位元組值與字元之間的對應 (包括數位和標點符號) 或字元。任何多位元組字碼頁也可以表示為非常大型的資料表 (具有64K 專案) 雙位元組值到字元之間。不過在實務上，通常會以資料表的形式表示第一個 256 (單一位元組) 個字元，以及做為雙位元組值的範圍。

如需字碼頁的詳細資訊，請參閱 Code Pages。

C 執行時間程式庫有兩種類型的內部字碼頁：地區設定和多位元組。您可以在程式執行期間變更目前的字碼頁 (請參閱) 的 setlocale 和 _setmbcp 功能檔。此外，執行時間程式庫可能會取得並使用作業系統字碼頁的值，此值在程式執行期間是固定的。

當地區設定字碼頁變更時，與地區設定相關的函式的行為會變更為所選字碼頁所規定的行為。依預設，所有地區設定相依的函式都會開始執行，並使用 "C" 地區設定特有的地區設定字碼頁。您可以藉由呼叫函式，來變更內部地區設定字碼頁 (以及其他地區設定特有的屬性) `setlocale` 。呼叫 `setlocale` (LC_ALL, "") 會將地區設定設定為作業系統使用者地區設定所指定的地區設定。

同樣地，當多位元組字碼頁變更時，多位元組函式的行為會變更為所選擇字碼頁所指示的行為。根據預設，所有多位元組函式都會開始執行，並使用對應至作業系統預設字碼頁的多位元組字碼頁。您可以藉由呼叫函數來變更內部多位元組字碼頁 `_setmbcp` 。

C 執行時間函式會 `setlocale` 設定、變更或查詢部分或所有目前程式的地區設定資訊。_Wsetlocale常式是寬字元版本的 `setlocale` ; 的引數和傳回值 `_wsetlocale` 是寬字元字串。

請參閱

Unicode 和 MBCS
字元集可攜性的優點

字元集移植性的優點

2020/12/12 • • Edit Online

即使您目前不想要對應用程式進行國際化，也可以利用 MFC 和 C 執行時間可攜性功能來獲益：

- 編碼可能可讓您的程式碼基底具彈性。您稍後可以輕鬆地將它移至 Unicode 或 MBCS。
- 使用 Unicode 讓您的應用程式更有效率。由於 Windows 使用 Unicode，因此必須轉譯與作業系統傳遞的非 Unicode 字串，而這會造成額外負荷。

請參閱

Unicode 和 MBCS
Unicode 的支援

Unicode 的支援

2020/12/12 • • Edit Online

Unicode 是支援所有字元集的規格，包括無法在單一位元組中表示的字元集。如果您是針對國際市場進行程式設計，建議您使用 Unicode 或 多位元組字元集 (MBCS) 。或者，撰寫程式的程式碼，讓您可以藉由變更開關來建立程式碼。

寬字元是 2 個位元組的多語系字元碼。有數萬個字元（包括全球新式運算中使用的所有字元，包括技術符號和特殊發行字元），可以根據 Unicode 規格，以使用 UTF-16 編碼的單一寬字元來表示。只有一個寬字元才能表示的字元可以使用 Unicode 代理配對功能，以 Unicode 配對表示。由於幾乎每個通用的字元都是在單一16位寬字元的 UTF-16 中表示，使用寬字元可簡化使用國際字元集的程式設計。使用以 UTF-16LE (的位元組由大到小) 編碼的寬字元，是 Windows 的原生字元格式。

寬字元字串會以 `wchar_t[]` 陣列呈現，且由 `wchar_t*` 指標指向。任何 ASCII 字元都可以寬字元呈現，方法是在該字元前附加字母 L。例如，L'\0' 是結束寬 (16 位元) NULL 字元。同樣地，任何 ASCII 字串常值都可以寬字元字串常值呈現，方法是在 ASCII 常值前附加字母 L (L"Hello")。

一般而言，寬字元會比多位元組字元佔用更多的記憶體空間，但處理起來更快。此外，在多位元組編碼中，一次只能表示一個地區設定，而世界中的所有字元集都是由 Unicode 標記法同時表示。

MFC 架構完全啟用 Unicode，MFC 透過使用可移植巨集，完成啟用 Unicode，如下表所示。

MFC 中的可移植資料型別

[illegible]	[illegible]
`char` , `wchar_t`	`_TCHAR`
`char*` 、 `LPSTR` (的 Win32 資料類型) 、 `LPWSTR`	`LPTSTR`
`const char*` 、 `LPCSTR` (的 Win32 資料類型) 、 `LPCWSTR`	`LPCTSTR`

類別會 `CString` 使用 `_TCHAR` 做為其基底，並提供可輕鬆轉換的函式和運算子。除了作業的基本單位是 16 位元的字元而非 8 位元的位元組之外，Unicode 的大部分字串作業都可以使用用於處理 Windows ANSI 字元集的相同邏輯，來進行撰寫。與使用多位元組字元集不同，您無需 (且不應該) 將 Unicode 字元視為兩個不同的位元組。但是，您必須處理以一組代理的寬字元表示的單一字元可能。一般情況下，請勿撰寫假設字串長度的程式碼與其所包含的字元數目（不論是窄或寬）相同。

您想要做什麼事？

- 使用 MFC Unicode 和多位元組字元集 (MBCS) 支援
- 在程式中啟用 Unicode
- 在我的程式中啟用 Unicode 和 MBCS
- 使用 Unicode 來建立國際化程式
- 瞭解 Unicode 的優點
- 使用 wmain，以便可以將寬字元引數傳遞至程式
- 請參閱 Unicode 程式設計的摘要

- 了解位元組寬度可移植性的一般文字對應

請參閱

文字和字串
支援使用 wmain

wmain 使用的支援

2019/12/2 • • Edit Online

Visual C++支援定義**wmain**函式, 並將寬字元引數傳遞至您的 Unicode 應用程式。您可以使用類似于 `main` 的格式, 將正式參數宣告為 wmain。然後您可以傳遞寬字元引數以及 (選擇性的) 一個指向程式的寬字元環境指標。**wmain** 的 `argv` 與 `envp` 參數都是 `wchar_t*` 類型。例如:

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

NOTE

MFC Unicode 應用程式 `wWinMain` 會使用做為進入點。在此情況下 `CWinApp::m_lpCmdLine` , 是 Unicode 字串。請務必使用 `wWinMainCRTStartup` /ENTRY連結器選項設定。

如果您的程式使用 **main** 函式, 則多位元組字元環境就會在程式啟動時由執行階段程式庫建立。環境的寬字元複本只有在需要時才建立 (例如, 藉著呼叫 `_wgetenv` 或 `_wputenv` 函式)。在第一次呼叫 `_wputenv` `_wgetenv` 時, 或在第一次呼叫時, 如果 MBCS 環境已經存在, 則會建立對應的寬字元字串環境。然後, `_wenviron` 全域變數會指向該環境, 這是通用變數的寬字元版本 `_environ` 。此時, 環境的兩個複本 (MBCS 和 Unicode) 會同時存在, 並在程式的整個生命週期中由執行時間系統維護。

同樣的, 如果您的程式使用 **wmain** 函式, 寬字元環境在程式啟動時建立, 並且由 `_wenviron` 全域變數指著。在第一次呼叫 `_putenv` 或 `getenv` `_environ` 時, 會建立 MBCS (ASCII) 環境, 並由全域變數指向。

另請參閱

Unicode 的支援
Unicode 程式設計摘要
WinMain 函式

Unicode 程式設計摘要

2020/12/12 • • Edit Online

若要利用 Unicode 的 MFC 和 C 執行時間支援，您需要：

- 定義 `_UNICODE` 。

  `_UNICODE` 建立程式之前，請先定義符號。

- 指定進入點。

  在專案 [屬性頁] 對話方塊中，于 [**連結器**] 資料夾的 [**Advanced**] 頁面上，將 **進入點** 符號設定為 `wWinMainCRTStartup` 。

- 使用可移植的執行時間函數和類型。

  使用適當的 C 執行時間函數來處理 Unicode 字串。您可以使用 `wcs` 一系列的函式，但您可能會想要使用可完全攜的 (啟用國際) `_TCHAR` 宏。這些宏的開頭都是 `_tcs` ，它們會取代為函式系列的一個 `str` 。這些函式會在 *執行時間程式庫參考* 的 [國際化] 區段中詳細說明。如需詳細資訊，請參閱 tchar 中的泛型文字對應。

  使用 `_TCHAR` 與 Unicode 支援中所述的相關可移植資料類型。

- 正確處理常值字串。

  Visual C++ 的編譯器會將常值字串解釋為下列程式碼：

  ```
  L"this is a literal string"
  ```

  表示 Unicode 字元字串。您可以使用相同的前置詞做為常值字元。使用 `_T` 宏以一般方式撰寫常值字串的程式碼，因此它們會以 unicode 或 ANSI 字串的形式編譯為 unicode 字串，(包括不含 Unicode 的 MBCS) 。例如，不要這樣撰寫：

  ```
  pWnd->SetWindowText( "Hello" );
  ```

  使用：

  ```
  pWnd->SetWindowText( _T("Hello") );
  ```

  `_UNICODE` 若已定義，則 `_T` 會將常值字串轉譯為開頭前置的表單，否則 `_T` 會轉譯沒有 l 前置詞的字串。

  > **TIP**
  > `_T` 宏與 `_TEXT` 宏相同。

- 請小心將字串長度傳遞給函式。

  某些函式需要字串中的字元數;有些則需要位元組數目。例如，如果已 `_UNICODE` 定義，則下列對物件的呼叫 `CArchive` 將無法運作 (`str` 是 `CString`) ：

```
archive.Write( str, str.GetLength( ) );    // invalid
```

在 Unicode 應用程式中，長度會提供您的字元數，但不是正確的位元組數目，因為每個字元都是2個位元組寬。相反地，您必須使用：

```
archive.Write( str, str.GetLength( ) * sizeof( _TCHAR ) );    // valid
```

，指定要寫入的正確位元組數目。

不過，以字元為導向的 MFC 成員函式，而不是位元組導向的工作，不需要額外的編碼：

```
pDC->TextOut( str, str.GetLength( ) );
```

`CDC::TextOut` 接受多個字元，而不是位元組數。

- 使用 fopen_s，_wfopen_s 開啟 Unicode 檔案。

總而言之，MFC 與執行時間程式庫提供下列 Unicode 程式設計支援：

- 除了資料庫類別成員函式之外，所有的 MFC 函式都具有 Unicode 功能，包括 `CString` 。`CString` 也提供 Unicode/ANSI 轉換函數。
- 執行時間程式庫提供所有字串處理函數的 Unicode 版本。(執行時間程式庫也會提供適用于 Unicode 或 MBCS 的可移植版本。這些都是 `_tcs` 宏。)
- tchar 提供可移植的資料類型，以及用 `_T` 來轉譯常值字串和字元的宏。如需詳細資訊，請參閱 tchar 中的泛型文字對應。
- 執行時間程式庫提供寬字元版本的 `main` 。使用 `wmain` 讓您的應用程式可感知 Unicode。

請參閱

Unicode 的支援

多位元組字元集 (MBCS) 的支援

2020/12/12 • • Edit Online

多位元組字元集 (Multibyte Character Set，MBCS) 是支援字元集的舊方法，例如不能以單一位元組表示的日文和中文。如果您進行新開發，除了終端使用者看不見的系統字串之外，您應該對所有文字字串使用 Unicode。MBCS 是舊版技術，不建議用於新開發。

最常見的 MBCS 實作是雙位元組字元集 (DBCS)。一般而言，Visual C++ 尤其是 MFC，已完全支援 DBCS。

如需範例，請參閱 MFC 原始程式碼檔案。

對於其語言使用大型字元集的市場中使用的平台，Unicode 的最佳替代方案是 MBCS。MFC 使用可國際語系化的資料類型和 C 執行階段函式來支援 MBCS。您應該在您的相同程式碼中執行相同動作。

在 MBCS 下，字元會以 1 或 2 個位元組編碼。在 2 個位元組的字元中，第一個或前導位元組表示它和下一個位元組會解譯成一個字元。第一個位元組來自一個範圍的字碼，保留供做為前導位元組使用。哪個範圍的位元組可以做為前導位元組，取決於使用中的字碼頁。例如，日文的字碼頁 932 使用範圍 0x81 到 0x9F 來當做前導位元組，但韓文字碼頁 949 使用不同的範圍。

請在您的 MBCS 程式設計中考慮以下所有項目。

環境 MBCS 字元中的 MBCS 字元可以出現在字串中，例如檔案和目錄名稱。

編輯作業

MBCS 應用程式中的編輯作業，應該可對字元而不是位元組運作。插入號不應分割字元，**向右箭** 號應該右移一個字元，依此類推。**Delete** 應該刪除一個字元; **復原** 應該將它重新插入。

字串處理

在使用 MBCS 的應用程式中，字串處理會造成特殊的問題。在單一字串中會混用這兩種寬度的字元;因此，您必須記得檢查前導位元組。

執行階段程式庫支援

C 執行階段程式庫和 MFC 支援單一位元組、MBCS 和 Unicode 程式設計。單一位元組字串會使用 `str` 執行時間函式系列來處理、使用對應的函式處理 MBCS 字串 `_mbs` ，以及使用對應的函式來處理 Unicode 字串 `wcs` 。MFC 類別成員函式實作會使用可攜式執行階段，在正確的情況下，其會將函式對應至函式的正常 `str` 系列、MBCS 函式或 Unicode 函式，如「MBCS/Unicode 可攜性」中所述。

MBCS/Unicode 可攜性

使用 tchar .h 標頭檔，您可以從相同的來源建立單一位元組、MBCS 和 Unicode 應用程式。Tchar 會定義前置詞為 *_tcs* 的宏， `str` 視需要對應至、 `_mbs` 或 `wcs` 函數。若要建立 MBCS，請定義符號 `_MBCS` 。若要建立 Unicode，請定義符號 `_UNICODE` 。根據預設， `_UNICODE` 會針對 MFC 應用程式定義。如需詳細資訊，請參閱 tchar 中的泛型文字對應。

> **NOTE**
>
> 如果您同時定義和，則行為是未定義的 `_UNICODE` `_MBCS` 。

Mbctype.h 和 Mbstring.h 標頭檔會定義 MBCS 特有的函式和巨集，在某些情況下您可能需要。例如， `_ismbblead` 會告訴您在字串中的特定位元組是否為前導位元組。

針對國際可攜性，請使用 Unicode 或多位元組字元集來撰寫程式程式碼 (mbcs) 。

您想要做什麼事？

- 在我的程式中啟用 MBCS
- 在我的程式中啟用 Unicode 和 MBCS
- 使用 MBCS 來建立國際化程式
- 請參閱 MBCS 程式設計的摘要
- 了解位元組寬度可移植性的一般文字對應

請參閱

文字和字串
Visual C++ 中的 MBCS 支援

Visual C++ 中的 MBCS 支援

2020/12/12 • • Edit Online

在啟用 MBCS 的 Windows 版本上執行時，Visual C++ 的開發系統 (包括整合式原始程式碼編輯器、偵錯工具和命令列工具) 已啟用 MBCS，但記憶體視窗例外。

記憶體視窗不會將資料的位元組解譯為 MBCS 字元，不過能解譯為 ANSI 或 Unicode 字元 ANSI 字元固定為 1 個位元組大小，而 Unicode 字元則是 2 個位元組大小。使用 MBCS 時，字元可以是 1 個或 2 個位元組大小，解譯則是根據所使用的字碼頁而定。因此，記憶體視窗很難順利顯示 MBCS 字元，記憶體視窗不知道哪個位元組是字元的開頭。開發人員可以在 [記憶體] 視窗中查看位元組值，並查詢資料表中的值，以判斷字元標記法。這是可能的，因為開發人員會根據原始程式碼知道字串的起始位址。

Visual C++ 可接受雙位元組字元，不論是否適合這樣做。這包括對話方塊中的路徑名稱和檔案名，以及 Visual C++ 資源編輯器中的文字專案 (例如，對話方塊編輯器中的靜態文字，以及圖示編輯器中的靜態文字專案) 。此外，預處理器會辨識某些雙位元組指示詞(例如，語句中的檔案名 `#include`)，以及做為和 pragma 的引數 `code_seg` `data_seg` 。在原始程式碼編輯器中，會接受批註中的雙位元組字元和字串常值，雖然不是 C/c + + 語言專案 (例如變數名稱) 。

支援輸入法編輯器 (IME)

針對使用 MBCS (的東亞市場所撰寫的應用程式(例如日本) 一般都支援使用 Windows IME 來輸入單一和雙位元組字元。Visual C++ 開發環境包含對 IME 的完整支援。

日文鍵盤不直接支援中文字元。IME 會將輸入其他日文字母 (Romaji、片假名或平假名) 的語音字串轉換成其可能的漢字標記法。如果不明確，您可以從數個替代方案中選擇。當您選取了預期的中文字元時，IME 會將兩則 `WM_CHAR` 訊息傳遞至控制應用程式。

由 ALT + 按鍵組合啟用的 IME `會顯示為一組按鈕，(指標) 和轉換視窗。應用程式會在文字插入點放置視窗。應用程式必須藉 `WM_MOVE` `WM_SIZE` 由重新置放轉換視窗以符合目標視窗的新位置或大小，來處理和訊息。

如果您希望應用程式的使用者能夠輸入中文字元，則應用程式必須處理 Windows IME 訊息。如需 IME 程式設計的詳細資訊，請參閱 輸入方法管理員。

Visual C++ 偵錯工具

Visual C++ 偵錯工具可讓您在 IME 訊息上設定中斷點。此外，[記憶體] 視窗可以顯示雙位元組字元。

命令列工具

Visual C++ 命令列工具(包括編譯器、NMAKE 和資源編譯器 (# A0))已啟用 MBCS。您可以使用資源編譯器的/c 選項，在編譯應用程式的資源時變更預設的字碼頁。

若要在原始程式碼編譯時期變更預設地區設定，請使用 #pragma setlocale。

圖形工具

Visual C++ 的 Windows 工具(例如 Spy + + 和資源編輯工具)完全支援 IME 字串。

請參閱

支援多位元組字元集 (Mbcs)
MBCS 程式設計秘訣

MBCS 程式設計提示

2020/12/12 • • Edit Online

在新的開發，您應該為使用者可能看到的所有字串使用 Unicode 字元編碼。MBCS 是舊版的技術，已被 Unicode 取代。本節為必須維護使用 MBCS 且不適合轉換為 Unicode 之現有程式的開發人員提供提示。這些建議適用于未以 MFC 撰寫的 MFC 應用程式和應用程式。主題包括：

- 一般 MBCS 程式設計建議
- 遞增和遞減指標
- 位元組索引
- 字串中的最後一個字元
- 字元指派
- 字元比較
- 緩衝區溢位

請參閱

支援多位元組字元集 (Mbcs)

一般 MBCS 程式設計的建議

2020/12/12 • • Edit Online

使用下列秘訣：

- 為了提供彈性，請盡可能使用執行時間宏，例如 `_tcschr` 和 `_tcscpy` 。如需詳細資訊，請參閱 tchar 中的泛型文字對應。
- 使用 C 執行時間函式 `_getmbcp` 來取得目前字碼頁的相關資訊。
- 請勿重複使用字串資源。根據目的語言，指定的字串在轉譯時可能會有不同的意義。例如，應用程式主功能表上的 [檔案] 可能會以不同于對話方塊中的字串 "File" 轉譯。如果您需要使用多個相同名稱的字串，請針對每個字串使用不同的字串識別碼。
- 您可能想要瞭解您的應用程式是否在啟用 MBCS 的作業系統上執行。若要這樣做，請在程式啟動時設定旗標。請勿依賴 API 呼叫。
- 設計對話方塊時，請在靜態文字控制項的結尾允許大約30% 的額外空間來進行 MBCS 轉譯。
- 為您的應用程式選取字型時請務必小心，因為某些字型在所有系統上都無法使用。
- 當您選取對話方塊的字型時，請使用 Ms Shell Dlg，而不是 Ms Sans Serif 或 Helvetica。在建立對話方塊之前，系統會以正確的字型取代 MS Shell Dlg。使用 MS Shell Dlg 可確保作業系統中的任何變更都能用來處理這個字型。(MFC 會以 DEFAULT_GUI_FONT 或 Windows 95、Windows 98 和 Windows NT 4 上的系統字型取代 MS Shell Dlg，因為這些系統不會正確地處理 MS Shell Dlg。)
- 設計應用程式時，請決定哪些字串可以當地語系化。如果不確定，請假設任何指定的字串都將當地語系化。因此，請不要混用可當地語系化的字串。

請參閱

MBCS 程式設計秘訣
遞增和遞減指標

增量和遞減指標

2020/12/12 • • Edit Online

使用下列秘訣：

- 指向 [前導位元組]，而不是 [後隨位元組]。如果有後隨位元組的指標，通常會不安全。通常更安全的方式是向前掃描字串，而非反向。
- 有指標遞增/遞減函式和宏可在整個字元上移動：

  ```
  sz1++;
  ```

  會變成：

  ```
  sz1 = _mbsinc( sz1 );
  ```

  和函式會 `_mbsinc` `_mbsdec` 以單位正確遞增和遞減 `character`，不論字元大小為何。

- 針對遞減，您需要字串標頭的指標，如下所示：

  ```
  sz2--;
  ```

  會變成：

  ```
  sz2 = _mbsdec( sz2Head, sz2 );
  ```

  或者，您的標頭指標可能是字串中的有效字元，如下所示：

  ```
  sz2Head < sz2
  ```

  您必須有已知有效前導位元組的指標。

- 您可能會想要維護上一個字元的指標，以便更快速地呼叫 `_mbsdec`。

請參閱

MBCS 程式設計秘訣
位元組索引

位元組索引

2020/12/12 • • Edit Online

使用下列秘訣：

- 將 bytewise 索引用於字串時，會顯示類似于指標操作所造成的問題。請考慮此範例，其會掃描字串是否有反斜線字元：

  ```
  while ( rgch[ i ] != '\\' )
      i++;
  ```

  這可能會編制後隨位元組的索引，而不是前導位元組，因此可能不會指向 `character` 。

- 使用 _mbclen 函數來解決上述問題：

  ```
  while ( rgch[ i ] != '\\' )
      i += _mbclen ( rgch + i );
  ```

  這會正確地為前導位元組編制索引，因此為 `character` 。函式會 `_mbclen` 判斷字元大小 (1 或2個位元組) 。

請參閱

MBCS 程式設計秘訣
字串中的最後一個字元

字串中的最後一個字元

2020/12/12 • • Edit Online

使用下列秘訣：

- 在許多情況下，尾位元組範圍會重迭 ASCII 字元集。您可以安全地針對 (小於 32) 的任何控制字元使用 bytewise 掃描。
- 請考慮下面這行程式碼，這可能會檢查字串中的最後一個字元是否為反斜線字元：

```
if ( sz[ strlen( sz ) - 1 ] == '\\' )    // Is last character a '\'?
    // . . .
```

因為不 `strlen` 是 MBCS 感知，所以它會傳回多位元組字元串中的位元組數目，而不是字元數。此外，請注意，在某些字碼頁 (932（例如）)，'\' (0x5c) 是有效的後隨位元組，（ `sz` 是 C 字串）。

其中一個可能的解決方案是以這種方式來重寫程式碼：

```
char *pLast;
pLast = _mbsrchr( sz, '\\' );    // find last occurrence of '\' in sz
if ( pLast && ( *_mbsinc( pLast ) == '\0' ) )
    // . . .
```

此程式碼會使用 MBCS 函數 `_mbsrchr` 和 `_mbsinc` 。因為這些函式是 MBCS 感知的，所以它們可以區別 '\' 字元和後隨位元組 '\'。如果字串中的最後一個字元是 null ('\0')，則程式碼會執行一些動作。

請參閱

MBCS 程式設計秘訣
字元指派

字元設定

2020/12/12 • • Edit Online

請考慮下列範例，在此範例中，`while` 迴圈會掃描字串，將 ' X ' 以外的所有字元複製到另一個字串：

```
while( *sz2 )
{
    if( *sz2 != 'X' )
        *sz1++ = *sz2++;
    else
        sz2++;
}
```

程式碼會將位元組複製 `sz2` 到指向的位置 `sz1`，然後再將遞增 `sz1` 以接收下一個位元組。但是，如果中的下一個字元 `sz2` 是雙位元組字元，則只會 `sz1` 複製第一個位元組的指派。下列程式碼會使用可移植的函式，安全地複製字元，並使用另一個來遞增 `sz1` 和 `sz2` 正確：

```
while( *sz2 )
{
    if( *sz2 != 'X' )
    {
        _mbscpy_s( sz1, 1, sz2 );
        sz1 = _mbsinc( sz1 );
        sz2 = _mbsinc( sz2 );
    }
    else
        sz2 = _mbsinc( sz2 );
}
```

請參閱

MBCS 程式設計秘訣
字元比較

字元比較

2019/12/2 • • Edit Online

使用下列秘訣：

- 比較以 ASCII 字元一個已知的前導位元組正常運作：

```
if( *sz1 == 'A' )
```

- 比較兩個未知的字元需要使用其中一種 Mbstring.h> 中定義的巨集：

```
if( !_mbccmp( sz1, sz2) )
```

這可確保會比較是否相等的雙位元組字元的兩個位元組。

另請參閱

MBCS 程式設計提示
緩衝區溢位

緩衝區溢位

2020/12/12 • • Edit Online

當您將字元放入緩衝區時，不同的字元大小可能會造成問題。請考慮下列程式碼，此程式碼會將字串中的字元複製 `sz` 到緩衝區中 `rgch` ：

```
cb = 0;
while( cb < sizeof( rgch ) )
    rgch[ cb++ ] = *sz++;
```

問題是：最後一個位元組是複製前導位元組嗎？ 下列無法解決問題，因為它可能會造成緩衝區溢位：

```
cb = 0;
while( cb < sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

`_mbccpy` 呼叫會嘗試進行正確的動作，不論是1或2個位元組，都會複製完整的字元。但是，如果字元寬度為2個位元組，則不會考慮複製的最後一個字元可能無法容納緩衝區。正確的解決方案是：

```
cb = 0;
while( (cb + _mbclen( sz )) <= sizeof( rgch ) )
{
    _mbccpy( rgch + cb, sz );
    cb += _mbclen( sz );
    sz = _mbsinc( sz );
}
```

這段程式碼會測試迴圈測試中是否有可能的緩衝區溢位， `_mbclen` 並使用來測試所指向之目前字元的大小 `sz` 。藉由呼叫函式 `_mbsnbcpy` ，您就可以使用一行程式碼來取代迴圈中的程式碼 `while` 。例如：

```
_mbsnbcpy( rgch, sz, sizeof( rgch ) );
```

另請參閱

MBCS 程式設計秘訣

支援 ANSI

2020/12/12 • • Edit Online

大部分的 MFC 類別和方法都支援 ANSI 字元集，雖然 MFC 架構整體會逐漸演進到僅支援 Unicode 字元集。由於 Windows Vista 和 Windows 通用控制項6.1 版中的持續改進，支援數種 ANSI 類別和方法已被取代。如需詳細資訊，請參閱已 淘汰的 ANSI api 和 Unicode 支援。

請參閱

Unicode 的支援
已被取代的 ANSI 應用程式開發介面
Shell 和通用控制項版本

Tchar 中的 Generic-Text 對應

2020/12/12 • • Edit Online

為了簡化程式碼的傳輸以供國際化使用，Microsoft 執行時間程式庫提供了許多資料類型、常式和其他物件的 Microsoft 特定泛型文字對應。您可以使用這些在 tchar 中定義的對應，以撰寫可針對單一位元組、多位元組或 Unicode 字元集編譯的泛型程式碼，這取決於您使用語句所定義的資訊清單常數 `#define` 。泛型文字對應是與 ANSI 不相容的 Microsoft 延伸模組。

藉由使用 tchar，您可以從相同的來源建立單一位元組、多位元組字元集 (MBCS) 和 Unicode 應用程式。tchar 會定義宏 (，其中) 前置詞 `_tcs` ，其中包含正確的預處理器定義、對應至 `str` 、 `_mbs` 或 `wcs` 函數。若要建立 MBCS，請定義符號 `_MBCS` 。若要建立 Unicode，請定義符號 `_UNICODE` 。若要建立單一位元組的應用程式，請不 (預設) 。根據預設， `_UNICODE` 會針對 MFC 應用程式定義。

`_TCHAR` 資料類型是在 tchar 中有條件地定義。如果您的 `_UNICODE` 組建定義了符號， `_TCHAR` 則會定義為， `wchar_t` 否則，針對單一位元組和 MBCS 組建，則會定義為 `char` 。(`wchar_t` ，基本的 Unicode 寬字元資料類型是8位的16位對應項 `signed char` 。) 適用于國際應用程式，請使用 `_tcs` 以 `_TCHAR` 單位(而非位元組)操作的函數系列。例如， `_tcsncpy` 複本 `n` `_TCHARs` ，而不是 `n` 位元組。

因為某些單一位元組字元集 (SBCS) 字串處理函式採用 (簽署的) `char*` 參數，所以在定義時，類型不相符的編譯器警告結果 `_MBCS` 。有三種方式可以避免這個警告：

1. 在 tchar 中使用型別安全的內嵌函式 Thunk。這是預設行為。
2. 藉由在命令列上定義，在 tchar 中使用直接宏 `_MB_MAP_DIRECT` 。如果這麼做，就必須手動對應類型。這是最快速的方法，但不是型別安全。
3. 在 tchar 中使用型別安全靜態連結的程式庫函數 Thunk。若要這樣做，請在命令列上定義常數 `_NO_INLINING` 。這是最慢、但類型最安全的方法。

泛型文字對應的前置處理器指示詞

# DEFINE	已定義	未定義
`_UNICODE`	Unicode (寬字元)	`_tcsrev` 對應至 `_wcsrev`
`_MBCS`	多位元組字元	`_tcsrev` 對應至 `_mbsrev`
無 (預設值不是 `_UNICODE` 也不 `_MBCS` 定義)	SBCS (ASCII)	`_tcsrev` 對應至 `strrev`

例如，在 tchar 中定義的泛型文字函式， `_tcsrev` 會對應至 `_mbsrev` 您在程式中定義的，如果您已 `_MBCS` 定義，則為 `_wcsrev` `_UNICODE` 。若兩者皆否，則 `_tcsrev` 會對應至 `strrev` 。在 tchar 中提供其他資料類型對應，以方便程式設計，但 `_TCHAR` 最有用。

泛型文字資料類型對應

GENERIC-TEXT 已定義	_UNICODE & 已定義 _MBCS	_MBCS 已定義	_UNICODE 已定義
`_TCHAR`	`char`	`char`	`wchar_t`
`_TINT`	`int`	`unsigned int`	`wint_t`

GENERIC-TEXT [[[[[[	_UNICODE & [[[_MBCS	_MBCS [[[	_UNICODE [[[
`_TSCHAR`	`signed char`	`signed char`	`wchar_t`
`_TUCHAR`	`unsigned char`	`unsigned char`	`wchar_t`
`_TXCHAR`	`char`	`unsigned char`	`wchar_t`
`_T` 或 `_TEXT`	無效果 (已由前置處理器移除)	無效果 (已由前置處理器移除)	`L` (將下列字元或字串轉換成其 Unicode 對應)

如需常式、變數和其他物件的泛型文字對應清單，請參閱 Run-Time 程式庫參考中的 泛型文字 對應。

> **NOTE**
>
> 請勿使用 `str` 具有 Unicode 字串的函式系列，其可能包含內嵌的 null 位元組。同樣地，請不要使用 `wcs` 具有 MBCS (或 SBCS) 字串的函數系列。

下列程式碼片段示範如何使用 `_TCHAR` 與 `_tcsrev` 來對應到 MBCS、Unicode 與 SBCS 模型。

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

如果已 `_MBCS` 定義，預處理器會將此片段對應到下列程式碼：

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

如果已 `_UNICODE` 定義，預處理器會將此片段對應到下列程式碼：

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

如果兩者 `_MBCS` 都 `_UNICODE` 未定義，預處理器會將片段對應至單一位元組 ASCII 程式碼，如下所示：

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

因此，您可以撰寫、維護和編譯單一原始程式碼檔案，以使用這三種字元集中的任何一種來執行特定常式。

請參閱

文字和字串
使用 TCHAR。具有 _MBCS 程式碼的 H 資料類型

使用含有 _MBCS 程式碼的 TCHAR.H 資料類型

2020/12/12 • • Edit Online

當定義資訊清單常數時 `_MBCS`，指定的泛型文字常式會對應到下列其中一種常式：

- 適當地處理多位元組位元組、字元與字串的 SBCS 常式。在此情況下，字串引數的類型應為 `char*`。例如，`_tprintf` 對應至，`printf` 字串引數 `printf` 的類型為 `char*`。如果您的 `_TCHAR` 字串類型使用了泛型文字資料類型，則型式和實際的參數類型會對應 `printf` 到，因為會 `_TCHAR*` 對應到 `char*`。
- MBCS 特定常式。在此案例中，字串引數類型必須是 `unsigned char*`。例如，`_tcsrev` 對應到 `_mbsrev`，它預期並傳回類型為 `unsigned char*` 的字串。如果您的 `_TCHAR` 字串類型使用了泛型文字資料類型，可能會發生類型衝突，因為 `_TCHAR` 對應至類型 `char`。

下面是防止此類型衝突 (以及可能會產生的 C 編譯器警告或 C++ 編譯器錯誤) 的三種解決方式：

- 使用預設行為。tchar 會在執行時間程式庫中提供常式的泛型文字常式原型，如下列範例所示。

  ```
  char * _tcsrev(char *);
  ```

  在預設案例中，會 `_tcsrev` 透過 Libc 中的 Thunk 對應至的原型 `_mbsrev`。這 `_mbsrev` 會將傳入參數和傳出傳回值的類型從 `_TCHAR*` (()) 變更 `char *` 為 `unsigned char *`。當您使用時，這個方法可確保類型相符 `_TCHAR`，但因為函式呼叫的額外負荷，所以會相當緩慢。

- 透過在您的程式碼中併入下列前置處理器陳述式，以內嵌方式使用函式。

  ```
  #define _USE_INLINING
  ```

  這個方法會造成 tchar 中提供的內嵌函式 Thunk，以將泛型文字常式直接對應至適當的 MBCS 常式。下列程式碼摘錄自 tchar，提供如何完成這項操作的範例。

  ```
  __inline char *_tcsrev(char *_s1)
  {return (char *)_mbsrev((unsigned char *)_s1);}
  ```

  若您可以使用內嵌，這是最佳解決方式，因為它可以保證類型相符，而且沒有任何額外成本。

- 在您的程式碼中併入下列預處理器語句，以使用直接對應。

  ```
  #define _MB_MAP_DIRECT
  ```

  若您不想使用預設行為或無法使用內嵌，此方法提供快速替代方式。它會讓宏將一般文字常式直接對應到常式的 MBCS 版本，如下列 tchar 範例所示。

  ```
  #define _tcschr _mbschr
  ```

  當您採用這種方法時，必須小心確保針對字串引數和字串傳回值使用適當的資料類型。您可以使用類型轉換來確保適當的類型相符，或者可以使用 `_TXCHAR` 泛型文字資料類型。`_TXCHAR` 對應至 SBCS 程式碼中的類型，`char` 但對應至 `unsigned char` MBCS 程式碼中的類型。如需有關泛型文字宏的詳細資訊，請參閱《*執行時間程式庫參考*》中的 泛型文字對應。

請參閱

Tchar 中的泛型文字對應

如何：在各種字串類型之間轉換

2020/12/12 • • Edit Online

本主題示範如何將各種 Visual C++ 字串類型轉換成其他字串。涵蓋的字串類型包括 `char *` 、 `wchar_t*` 、 _bstr_t、CComBSTR、CString、basic_string和 System.String 。在所有情況下，會在轉換為新類型時進行字串的複本。對新字串所做的任何變更將不會影響原始字串，反之亦然。

範例：從 char * 轉換

說明

這個範例會示範如何從轉換成 `char *` 以上所列的其他字串類型。 `char *` 字串 (也稱為 C 樣式字串) 使用 null 字元指出字串的結尾。C 樣式字串通常每個字元都需要一個位元組，但也可以使用兩個位元組。在下列範例中， `char *` 字串有時稱為多位元組字元字串，因為從 Unicode 字串轉換所產生的字串資料。單一位元組和多位元組字元 (`MBCS`) 函數可以在 `char *` 字串上操作。

程式碼

```
// convert_from_char.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create and display a C style string, and then use it
    // to create different kinds of strings.
    char *orig = "Hello, World!";
    cout << orig << " (char *)" << endl;

    // newsize describes the length of the
    // wchar_t string called wcstring in terms of the number
    // of wide characters, not the number of bytes.
    size_t newsize = strlen(orig) + 1;

    // The following creates a buffer large enough to contain
    // the exact number of characters in the original string
    // in the new format. If you want to add more characters
    // to the end of the string, increase the value of newsize
    // to increase the size of the buffer.
    wchar_t * wcstring = new wchar_t[newsize];

    // Convert char* string to a wchar_t* string.
    size_t convertedChars = 0;
    mbstowcs_s(&convertedChars, wcstring, newsize, orig, _TRUNCATE);
    // Display the result and indicate the type of string that it is.
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert the C style string to a _bstr_t string.
    _bstr_t bstrt(orig);
    // Append the type of string to the new string
```

```
    // and then display the result.
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert the C style string to a CComBSTR string.
    CComBSTR ccombstr(orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert the C style string to a CStringA and display it.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert the C style string to a CStringW and display it.
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a CStringW correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the C style string to a basic_string and display it.
    string basicstring(orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert the C style string to a System::String and display it.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

範例：轉換 wchar_t *

說明

這個範例會示範如何從轉換成 `wchar_t *` 以上所列的其他字串類型。數種字串類型，包括 `wchar_t *` 執行寬字元格式。若要在多位元組和寬字元格式之間轉換字串，您可以使用單一函式呼叫，例如 `mbstowcs_s` 或類別的函式調用 `CStringA` 。

程式碼

```
// convert_from_wchar_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
```

```
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create a string of wide characters, display it, and then
    // use this string to create other types of strings.
    wchar_t *orig = _T("Hello, World!");
    wcout << orig << _T(" (wchar_t *)") << endl;

    // Convert the wchar_t string to a char* string. Record
    // the length of the original string and add 1 to it to
    // account for the terminating null character.
    size_t origsize = wcslen(orig) + 1;
    size_t convertedChars = 0;

    // Use a multibyte string to append the type of string
    // to the new string before displaying the result.
    char strConcat[] = " (char *)";
    size_t strConcatsize = (strlen( strConcat ) + 1)*2;

    // Allocate two bytes in the multibyte output string for every wide
    // character in the input string (including a wide character
    // null). Because a multibyte character can be one or two bytes,
    // you should allot two bytes for each character. Having extra
    // space for the new string is not an error, but having
    // insufficient space is a potential security problem.
    const size_t newsize = origsize*2;
    // The new string will contain a converted copy of the original
    // string plus the type of string appended to it.
    char *nstring = new char[newsize+strConcatsize];

    // Put a copy of the converted string into nstring
    wcstombs_s(&convertedChars, nstring, newsize, orig, _TRUNCATE);
    // append the type of string to the new string.
    _mbscat_s((unsigned char*)nstring, newsize+strConcatsize, (unsigned char*)strConcat);
    // Display the result.
    cout << nstring << endl;

    // Convert a wchar_t to a _bstr_t string and display it.
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert the wchar_t string to a BSTR wide character string
    // by using the ATL CComBSTR wrapper class for BSTR strings.
    // Then display the result.

    CComBSTR ccombstr(orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // CW2A converts the string in ccombstr to a multibyte
        // string in printstr, used here for display output.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
        // The following line of code is an easier way to
        // display wide character strings:
        wcout << (LPCTSTR) ccombstr << endl;
    }

    // Convert a wide wchar_t string to a multibyte CStringA,
    // append the type of string to it, and display the result.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a wide character wchar_t string to a wide
```

```
    // character CStringW string and append the type of string to it
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a CStringW correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the wide character wchar_t string to a
    // basic_string, append the type of string to it, and
    // display the result.
    wstring basicstring(orig);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    // Convert a wide character wchar_t string to a
    // System::String string, append the type of string to it,
    // and display the result.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (wchar_t *)
Hello, World! (char *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

範例：從 _bstr_t 轉換

說明

這個範例會示範如何從轉換成 `_bstr_t` 以上所列的其他字串類型。`_bstr_t` 物件是封裝寬字元字串的方式 `BSTR`。BSTR 字串具有長度值，而且不會使用 null 字元來結束字串，但您轉換成的字串類型可能需要終止的 null。

程式碼

```
// convert_from_bstr_t.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;

int main()
{
    // Create a _bstr_t string, display the result, and indicate the
    // type of string that it is.
    _bstr_t orig("Hello, World!");
    wcout << orig << " (_bstr_t)" << endl;

    // Convert the wide character _bstr_t string to a C style
    // string. To be safe, allocate two bytes for each character
```

```
    // in the char* string, including the terminating null.
    const size_t newsize = (orig.length()+1)*2;
    char *nstring = new char[newsize];

    // Uses the _bstr_t operator (char *) to obtain a null
    // terminated string from the _bstr_t object for
    // nstring.
    strcpy_s(nstring, newsize, (char *)orig);
    strcat_s(nstring, newsize, " (char *)");
    cout << nstring << endl;

    // Prepare the type of string to append to the result.
    wchar_t strConcat[] = _T(" (wchar_t *)");
    size_t strConcatLen = wcslen(strConcat) + 1;

    // Convert a _bstr_t to a wchar_t* string.
    const size_t widesize = orig.length()+ strConcatLen;
    wchar_t *wcstring = new wchar_t[newsize];
    wcscpy_s(wcstring, widesize, (wchar_t *)orig);
    wcscat_s(wcstring, widesize, strConcat);
    wcout << wcstring << endl;

    // Convert a _bstr_t string to a CComBSTR string.
    CComBSTR ccombstr((char *)orig);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a _bstr_t to a CStringA string.
    CStringA cstringa(orig.GetBSTR());
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a _bstr_t to a CStringW string.
    CStringW cstring(orig.GetBSTR());
    cstring += " (CStringW)";
    // To display a cstring correctly, use wcout and
    // "cast" the cstring to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert the _bstr_t to a basic_string.
    string basicstring((char *)orig);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert the _bstr_t to a System::String.
    String ^systemstring = gcnew String((char *)orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (_bstr_t)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

範例：從 CComBSTR 轉換

說明

這個範例會示範如何從轉換成 `CComBSTR` 以上所列的其他字串類型。就像 _bstr_t 一樣， `CComBSTR` 物件是封裝寬字元 bstr 字串的方式。BSTR 字串具有長度值，而且不會使用 null 字元來結束字串，但您轉換成的字串類型可能需要終止的 null。

程式碼

```
// convert_from_ccombstr.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

int main()
{
    // Create and initialize a BSTR string by using a CComBSTR object.
    CComBSTR orig("Hello, World!");
    // Convert the BSTR into a multibyte string, display the result,
    // and indicate the type of string that it is.
    CW2A printstr(orig);
    cout << printstr << " (CComBSTR)" << endl;

    // Convert a wide character CComBSTR string to a
    // regular multibyte char* string. Allocate enough space
    // in the new string for the largest possible result,
    // including space for a terminating null.
    const size_t newsize = (orig.Length()+1)*2;
    char *nstring = new char[newsize];

    // Create a string conversion object, copy the result to
    // the new char* string, and display the result.
    CW2A tmpstr1(orig);
    strcpy_s(nstring, newsize, tmpstr1);
    cout << nstring << " (char *)" << endl;

    // Prepare the type of string to append to the result.
    wchar_t strConcat[] = _T(" (wchar_t *)");
    size_t strConcatLen = wcslen(strConcat) + 1;

    // Convert a wide character CComBSTR string to a wchar_t*.
    // The code first determines the length of the converted string
    // plus the length of the appended type of string, then
    // prepares the final wchar_t string for display.
    const size_t widesize = orig.Length()+ strConcatLen;
    wchar_t *wcstring = new wchar_t[widesize];
    wcscpy_s(wcstring, widesize, orig);
    wcscat_s(wcstring, widesize, strConcat);

    // Display the result. Unlike CStringW, a wchar_t does not need
    // a cast to (LPCTSTR) with wcout.
    wcout << wcstring << endl;

    // Convert a wide character CComBSTR to a wide character _bstr_t,
    // append the type of string to it, and display the result.
    _bstr_t bstrt(orig);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;
```

```
    // Convert a wide character CComBSTR to a multibyte CStringA,
    // append the type of string to it, and display the result.
    CStringA cstringa(orig);
    cstringa += " (CStringA)";
    cout << cstringa << endl;

    // Convert a wide character CComBSTR to a wide character CStringW.
    CStringW cstring(orig);
    cstring += " (CStringW)";
    // To display a cstring correctly, use wcout and cast cstring
    // to (LPCTSTR).
    wcout << (LPCTSTR)cstring << endl;

    // Convert a wide character CComBSTR to a wide character
    // basic_string.
    wstring basicstring(orig);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    // Convert a wide character CComBSTR to a System::String.
    String ^systemstring = gcnew String(orig);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (CComBSTR)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
Hello, World! (System::String)
```

範例：從 CString 轉換

說明

這個範例會示範如何從轉換成 `CString` 以上所列的其他字串類型。 `CString` 是以 TCHAR 資料類型為基礎，而這取決於是否已定義符號 `_UNICODE` 。如果未 `_UNICODE` 定義， `TCHAR` 則會定義為 char 且 `CString` 包含多位元組字元字串; 如果 `_UNICODE` 已定義， `TCHAR` 則會定義為， `wchar_t` 且 `CString` 包含寬字元字串。

`CStringA` 這是的多位元組字元串一律版本 `CString` ， `CStringW` 是僅限寬字元字串版本。 `CStringA` 也不 `CStringW` `_UNICODE` 會用來決定應該如何編譯。 `CStringA``CStringW` 在此範例中，會用來說明緩衝區大小配置和輸出處理的微小差異。

程式碼

```
// convert_from_cstring.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"

using namespace std;
using namespace System;
```

```
int main()
{
    // Set up a multibyte CStringA string.
    CStringA origa("Hello, World!");
    cout << origa << " (CStringA)" << endl;

    // Set up a wide character CStringW string.
    CStringW origw("Hello, World!");
    wcout << (LPCTSTR)origw << _T(" (CStringW)") << endl;

    // Convert to a char* string from CStringA string
    // and display the result.
    const size_t newsizea = (origa.GetLength() + 1);
    char *nstringa = new char[newsizea];
    strcpy_s(nstringa, newsizea, origa);
    cout << nstringa << " (char *)" << endl;

    // Convert to a char* string from a wide character
    // CStringW string. To be safe, we allocate two bytes for each
    // character in the original string, including the terminating
    // null.
    const size_t newsizew = (origw.GetLength() + 1)*2;
    char *nstringw = new char[newsizew];
    size_t convertedCharsw = 0;
    wcstombs_s(&convertedCharsw, nstringw, newsizew, origw, _TRUNCATE );
    cout << nstringw << " (char *)" << endl;

    // Convert to a wchar_t* from CStringA
    size_t convertedCharsa = 0;
    wchar_t *wcstring = new wchar_t[newsizea];
    mbstowcs_s(&convertedCharsa, wcstring, newsizea, origa, _TRUNCATE);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert to a wide character wchar_t* string from
    // a wide character CStringW string.
    wchar_t *n2stringw = new wchar_t[newsizew];
    wcscpy_s( n2stringw, newsizew, origw );
    wcout << n2stringw << _T(" (wchar_t *)") << endl;

    // Convert to a wide character _bstr_t string from
    // a multibyte CStringA string.
    _bstr_t bstrt(origa);
    bstrt += _T(" (_bstr_t)");
    wcout << bstrt << endl;

    // Convert to a wide character _bstr_t string from
    // a wide character CStringW string.
    bstr_t bstrtw(origw);
    bstrtw += " (_bstr_t)";
    wcout << bstrtw << endl;

    // Convert to a wide character CComBSTR string from
    // a multibyte character CStringA string.
    CComBSTR ccombstr(origa);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Convert the wide character string to multibyte
        // for printing.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert to a wide character CComBSTR string from
    // a wide character CStringW string.
    CComBSTR ccombstrw(origw);

    // Append the type of string to it, and display the result.
    if (ccombstrw.Append(_T(" (CComBSTR)")) == S_OK)
```

```
    {
        CW2A printstrw(ccombstrw);
        wcout << printstrw << endl;
    }

    // Convert a multibyte character CStringA to a
    // multibyte version of a basic_string string.
    string basicstring(origa);
    basicstring += " (basic_string)";
    cout << basicstring << endl;

    // Convert a wide character CStringW to a
    // wide character version of a basic_string
    // string.
    wstring basicstringw(origw);
    basicstringw += _T(" (basic_string)");
    wcout << basicstringw << endl;

    // Convert a multibyte character CStringA to a
    // System::String.
    String ^systemstring = gcnew String(origa);
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;

    // Convert a wide character CStringW to a
    // System::String.
    String ^systemstringw = gcnew String(origw);
    systemstringw += " (System::String)";
    Console::WriteLine("{0}", systemstringw);
    delete systemstringw;
}
```

```
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (char *)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CComBSTR)
Hello, World! (basic_string)
Hello, World! (System::String)
```

範例：從 basic_string 轉換

說明

這個範例會示範如何從轉換成 `basic_string` 以上所列的其他字串類型。

程式碼

```
// convert_from_basic_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
```

```
using namespace std;
using namespace System;

int main()
{
    // Set up a basic_string string.
    string orig("Hello, World!");
    cout << orig << " (basic_string)" << endl;

    // Convert a wide character basic_string string to a multibyte char*
    // string. To be safe, we allocate two bytes for each character
    // in the original string, including the terminating null.
    const size_t newsize = (strlen(orig.c_str()) + 1)*2;
    char *nstring = new char[newsize];
    strcpy_s(nstring, newsize, orig.c_str());
    cout << nstring << " (char *)" << endl;

    // Convert a basic_string string to a wide character
    // wchar_t* string. You must first convert to a char*
    // for this to work.
    const size_t newsizew = strlen(orig.c_str()) + 1;
    size_t convertedChars = 0;
    wchar_t *wcstring = new wchar_t[newsizew];
    mbstowcs_s(&convertedChars, wcstring, newsizew, orig.c_str(), _TRUNCATE);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert a basic_string string to a wide character
    // _bstr_t string.
    _bstr_t bstrt(orig.c_str());
    bstrt += _T(" (_bstr_t)");
    wcout << bstrt << endl;

    // Convert a basic_string string to a wide character
    // CComBSTR string.
    CComBSTR ccombstr(orig.c_str());
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Make a multibyte version of the CComBSTR string
        // and display the result.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a basic_string string into a multibyte
    // CStringA string.
    CStringA cstring(orig.c_str());
    cstring += " (CStringA)";
    cout << cstring << endl;

    // Convert a basic_string string into a wide
    // character CStringW string.
    CStringW cstringw(orig.c_str());
    cstringw += _T(" (CStringW)");
    wcout << (LPCTSTR)cstringw << endl;

    // Convert a basic_string string to a System::String
    String ^systemstring = gcnew String(orig.c_str());
    systemstring += " (System::String)";
    Console::WriteLine("{0}", systemstring);
    delete systemstring;
}
```

```
Hello, World! (basic_string)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (System::String)
```

範例：從 System:: String 轉換

說明

這個範例示範如何從寬字元 (Unicode) System:: String 轉換為上列其他字串類型。

程式碼

```
// convert_from_system_string.cpp
// compile with: /clr /link comsuppw.lib

#include <iostream>
#include <stdlib.h>
#include <string>

#include "atlbase.h"
#include "atlstr.h"
#include "comutil.h"
#include "vcclr.h"

using namespace std;
using namespace System;
using namespace System::Runtime::InteropServices;

int main()
{
    // Set up a System::String and display the result.
    String ^orig = gcnew String("Hello, World!");
    Console::WriteLine("{0} (System::String)", orig);

    // Obtain a pointer to the System::String in order to
    // first lock memory into place, so that the
    // Garbage Collector (GC) cannot move that object
    // while we call native functions.
    pin_ptr<const wchar_t> wch = PtrToStringChars(orig);

    // Make a copy of the System::String as a multibyte
    // char* string. Allocate two bytes in the multibyte
    // output string for every wide character in the input
    // string, including space for a terminating null.
    size_t origsize = wcslen(wch) + 1;
    const size_t newsize = origsize*2;
    size_t convertedChars = 0;
    char *nstring = new char[newsize];
    wcstombs_s(&convertedChars, nstring, newsize, wch, _TRUNCATE);
    cout << nstring << " (char *)" << endl;

    // Convert a wide character System::String to a
    // wide character wchar_t* string.
    const size_t newsizew = origsize;
    wchar_t *wcstring = new wchar_t[newsizew];
    wcscpy_s(wcstring, newsizew, wch);
    wcout << wcstring << _T(" (wchar_t *)") << endl;

    // Convert a wide character System::String to a
    // wide character _bstr_t string.
    _bstr_t bstrt(wch);
```

```
    _bstr_t bstrt(wch);
    bstrt += " (_bstr_t)";
    cout << bstrt << endl;

    // Convert a wide character System::String
    // to a wide character CComBSTR string.
    CComBSTR ccombstr(wch);
    if (ccombstr.Append(_T(" (CComBSTR)")) == S_OK)
    {
        // Make a multibyte copy of the CComBSTR string
        // and display the result.
        CW2A printstr(ccombstr);
        cout << printstr << endl;
    }

    // Convert a wide character System::String to
    // a multibyte CStringA string.
    CStringA cstring(wch);
    cstring += " (CStringA)";
    cout << cstring << endl;

    // Convert a wide character System::String to
    // a wide character CStringW string.
    CStringW cstringw(wch);
    cstringw += " (CStringW)";
    wcout << (LPCTSTR)cstringw << endl;

    // Convert a wide character System::String to
    // a wide character basic_string.
    wstring basicstring(wch);
    basicstring += _T(" (basic_string)");
    wcout << basicstring << endl;

    delete orig;
}
```

```
Hello, World! (System::String)
Hello, World! (char *)
Hello, World! (wchar_t *)
Hello, World! (_bstr_t)
Hello, World! (CComBSTR)
Hello, World! (CStringA)
Hello, World! (CStringW)
Hello, World! (basic_string)
```

請參閱

ATL 和 MFC 字串轉換宏
與 C 樣式字串相關的 CString 作業
如何：將標準字串轉換為 System::String
如何：將 System::String 轉換為標準字串
如何：將 System::String 轉換為 wchar_t * 或 char*
使用 CComBSTR 進行程式設計
mbstowcs_s、_mbstowcs_s_l
wcstombs_s、_wcstombs_s_l
strcpy_s、wcscpy_s、_mbscpy_s
strcat_s、wcscat_s、_mbscat_s
pin_ptr (C++/CLI)