

Using Unicode Normalization to Represent Strings

Unicode

Unicode is a worldwide character-encoding standard. The system uses Unicode exclusively for character and string manipulation. For a detailed description of all aspects of Unicode, refer to The Unicode Standard.

Compared to older mechanisms for handling character and string data, Unicode simplifies software localization and improves multilingual text processing. By using Unicode to represent character and string data in your applications, you can enable universal data exchange capabilities for global marketing, using a single binary file for every possible character code. Unicode does the following:

- Allows any combination of characters, drawn from any combination of scripts and languages, to co-exist in a single document.
- Defines semantics for each character.
- Standardizes script behavior.
- Provides a standard algorithm for bidirectional text.
- Defines cross-mappings to other standards.
- Defines multiple encodings of its single character set: UTF-7, UTF-8, UTF-16, and UTF-32. Conversion of data among these encodings is lossless.

Unicode supports numerous scripts used by languages around the world, and also a large number of technical symbols and special characters used in publishing. The supported scripts include, but are not limited to, Latin, Greek, Cyrillic, Hebrew, Arabic, Devanagari, Thai, Han, Hangul, Hiragana, and Katakana. Supported languages include, but are not limited to, German, French, English, Greek, Russian, Hebrew, Arabic, Hindi, Thai, Chinese, Korean, and Japanese. Unicode currently can represent the vast majority of characters in modern computer use around the world, and continues to be updated to make it even more complete.

Unicode-enabled functions are described in Conventions for Function Prototypes. These functions use UTF-16 (wide character) encoding, which is the most common encoding of Unicode and the one used for native Unicode encoding on Windows operating systems. Each code value is 16 bits wide, in contrast to the older code page approach to character and string data, which uses 8-bit code values. The use of 16 bits allows the direct encoding of 65,536 characters. In fact, the universe of symbols used to transcribe human languages is even larger than that, and UTF-16 code points in the range U+D800 through U+DFFF are used to form surrogate pairs, which constitute 32-bit encodings of supplementary characters. See Surrogates and Supplementary Characters for further discussion.

The Unicode character set includes numerous combining characters, such as U+0308 ("¨"), a combining dieresis or umlaut. Unicode can often represent the same glyph in either a "composed" or a "decomposed" form: for example, the composed form of "Ä" is the single Unicode code point "Ä" (U+00C4), while its decomposed form is "A" + "¨" (U+0041 U+0308). Unicode does not define a composed form for every glyph. For example, the Vietnamese lowercase "o" with circumflex and tilde

Using Unicode Normalization to Represent Strings

("õ") is represented by U+006f U+0302 U+0303 (o + Circumflex + Tilde). For further discussion of combining characters and related issues, see [Using Unicode Normalization to Represent Strings](#).

For compatibility with 8-bit and 7-bit environments, Unicode can also be encoded as UTF-8 and UTF-7, respectively. While Unicode-enabled functions in Windows use UTF-16, it is also possible to work with data encoded in UTF-8 or UTF-7, which are supported in Windows as multibyte character set code pages.

New Windows applications should use UTF-16 as their internal data representation. Windows also provides extensive support for code pages, and mixed use in the same application is possible. Even new Unicode-based applications sometimes have to work with code pages. Reasons for this are discussed in [Code Pages](#).

An application can use the `MultiByteToWideChar` and `WideCharToMultiByte` functions to convert between strings based on code pages and Unicode strings. Although their names refer to "MultiByte", these functions work equally well with single-byte character set (SBCS), double-byte character set (DBCS), and multibyte character set (MBCS) code pages.

Typically, a Windows application should use UTF-16 internally, converting only as part of a "thin layer" over the interface that must use another format. This technique defends against loss and corruption of data. Each code page supports different characters, but none of them supports the full spectrum of characters provided by Unicode. Most of the code pages support different subsets, differently encoded. The code pages for UTF-8 and UTF-7 are an exception, since they support the complete Unicode character set, and conversion between these encodings and UTF-16 is lossless.

Data converted directly from the encoding used by one code page to the encoding used by another is subject to corruption, because the same data value on different code pages can encode a different character. Even when your application is converting as close to the interface as possible, you should think carefully about the range of data to handle.

Data converted from Unicode to a code page is subject to data loss, because a given code page might not be able to represent every character used in that particular Unicode data. Therefore, note that `WideCharToMultiByte` might lose some data if the target code page cannot represent all of the characters in the Unicode string.

When modernizing code page-based legacy applications to use Unicode, you can use generic functions and the `TEXT` macro to maintain a single set of sources from which to compile two versions of your application. One version supports Unicode and the other one works with Windows code pages. Using this mechanism, you can convert even very large applications from Windows code pages to Unicode while maintaining application sources that can be compiled, built, and tested at all phases of the conversion. For more information, see [Conventions for Function Prototypes](#).

Unicode characters and strings use data types that are distinct from those for code page-based characters and strings. Along with a series of macros and naming conventions, this distinction minimizes the chance of accidentally mixing the two types of character data. It facilitates compiler type checking to ensure that only Unicode parameter values are used with functions expecting Unicode strings.

Using Unicode Normalization to Represent Strings

Using Unicode Normalization to Represent Strings

Applications can use Unicode to represent strings in multiple forms. As Unicode acceptance has grown, especially via the Internet, the need has arisen to eliminate non-essential differences in Unicode strings. Multiple representations for a combination of characters complicate software, for example, when a Web server responds to a page request or a linker seeks a particular identifier in a library.

Caution Different Unicode strings can appear to be visually identical, raising security concerns. For more information, see [Security Considerations: International Features](#).

In response to this requirement, the Unicode Consortium has defined a process called "normalization," which produces one binary representation for any of the equivalent binary representations of a character. Once normalized, two strings are equivalent if and only if they have identical binary representations. The normalization eliminates some differences but preserves case.

To use Unicode normalization, an application can call the `NormalizeString` and `IsNormalizedString` functions for rearrangement of strings according to Unicode 4.0 TR#15. Normalization can help improve security by reducing alternate string representations that have the same linguistic meaning. Remember, however, that normalization cannot eliminate alternate representations entirely.

For a detailed description of the Unicode standards for normalization, refer to [Unicode Standard Annex #15: Unicode Normalization Forms \(UAX #15\)](#).

Caution Because normalization can change the form of a string, security mechanisms or character validation algorithms should usually be implemented after normalization. For more information, see [Security Considerations: International Features](#).

Provide Multiple Representations of the Same String

In many cases, Unicode allows multiple representations of what is, linguistically, the same string. For example:

Capital A with dieresis (umlaut) can be represented either as a single Unicode code point "Ä" (U+00C4) or the combination of Capital A and the combining Dieresis character ("A" + "¨", that is, U+0041 U+0308). Similar considerations apply for many other characters with diacritic marks.

Capital A itself can be represented either in the usual manner (Latin Capital Letter A, U+0041) or by Fullwidth Latin Capital Letter A (U+FF21). Similar considerations apply for the other simple Latin letters (both uppercase and lowercase) and for the katakana characters used in writing Japanese.

The string "fi" can be represented either by the characters "f" and "i" (U+0066 U+0069) or by the ligature "fi" (U+FB01). Similar considerations apply for many other combinations of characters for which Unicode defines ligatures.

Use the Four Defined Normalization Forms

Your applications can perform Unicode normalization using several algorithms, called "normalization forms," that obey different rules. The Unicode Consortium has defined four normalization forms: NFC

Using Unicode Normalization to Represent Strings

(form C), NFD (form D), NFKC (form KC), and NFKD (form KD). Each form eliminates some differences but preserves case. Win32 and the .NET Framework support all four normalization forms. The NLS enumeration type `NORM_FORM` supports the four standard Unicode normalization forms. Forms C and D provide canonical forms for strings. Non-canonical forms KC and KD provide further compatibility, and can reveal certain semantic equivalences that are not apparent in forms C and D. However, they do so at the expense of a certain loss of information, and generally should not be used as a canonical way to store strings.

Of the two canonical forms, form C is a "composed" form and form D is a "decomposed" form. For example, form C uses the single Unicode code point "Ä" (U+00C4), while form D uses ("A" + "¨", that is U+0041 U+0308). These render identically, because "¨" (U+0308) is a combining character. Form D can use any number of code points to represent a single code point used by form C.

If two strings are identical in either form C or form D, they are identical in the other form. Furthermore, when correctly rendered, they display indistinguishably from one another and from the original non-normalized string.

Once normalized, strings cannot be consistently returned to their original representation. For example, if a string with a mixture of composed and decomposed character representations is converted to a normalized form, there is no way to un-normalize it to the original mixed string. Therefore, if an application requires the original representation of the string, it must store that representation explicitly. However, converting between the two canonical forms is reversible. A string in form C can be converted to form D and then back to form C, and the result is identical to the original form C string.

Forms KC and KD are similar to forms C and D, respectively, but these "compatibility forms" have additional mappings of compatible characters to the basic form of each character. Such mappings can cause minor character variations to be lost. They combine certain characters that are visually distinct. For example, they combine full-width and half-width characters with the same semantic meaning, or different forms of the same Arabic letter, or the ligature "fi" (U+FB01) and the character pair "fi" (U+0066 U+0069). They also combine some characters that might sometimes have a different semantic meaning, such as a digit written as a superscript, as a subscript, or enclosed in a circle. Because of this information loss, forms KC and KD generally should not be used as canonical forms of strings, but they are useful for certain applications.

Form KC is a composed form and form KD is a decomposed form. The application can go back and forth between forms KC and KD, but there is no consistent way to go from form KC or KD back to the original string, even if the original string is in form C or D.

Windows, Microsoft applications, and the .NET Framework generally generate characters in form C using normal input methods. For most purposes on Windows, form C is the preferred form. For example, characters in form C are produced by Windows keyboard input. However, characters imported from the Web and other platforms can introduce other normalization forms into the data stream.

The following examples are drawn from UAX #15, and illustrate the differences among the four normalization forms.

Using Unicode Normalization to Represent Strings

Original	Form D	Form C	Notes
"Äffin"	"A\u0308ffin"	"Äffin"	The ffi_ligature (U+FB03) is not decomposed, because it has a compatibility mapping, not a canonical mapping.
"Ä\uFB03n"	"A\u0308\uFB03n"	"Ä\uFB03n"	
"Henry IV"	"Henry IV"	"Henry IV"	The ROMAN NUMERAL IV (U+2163) is not decomposed.
"Henry \u2163"	"Henry \u2163"	"Henry \u2163"	
ga	ka +ten	ga	Different compatibility equivalents of a single Japanese character do not result in the same string in form C.
ka +ten	ka +ten	ga	
hw_ka +hw_ten	hw_ka +hw_ten	hw_ka +hw_ten	
ka +hw_ten	ka +hw_ten	ka +hw_ten	
hw_ka +ten	hw_ka +ten	hw_ka +ten	
kaks	k i + a m + ks f	kaks	

Original	Form KD	Form KC	Notes
"Äffin"	"A\u0308ffin"	"Äffin"	The ffi_ligature (U+FB03) is decomposed in form KC, but not in form C.
"Ä\uFB03n"	"A\u0308ffin"	"Äffin"	
"Henry IV"	"Henry IV"	"Henry IV"	The resulting strings here are identical in form KC.
"Henry \u2163"	"Henry IV"	"Henry IV"	
ga	ka +ten	ga	Different compatibility equivalents of a single Japanese character result in the same string in form KC.
ka +ten	ka +ten	ga	

Using Unicode Normalization to Represent Strings

hw_ka +hw_ten	ka +ten	ga	
ka +hw_ten	ka +ten	ga	
hw_ka +ten	ka +ten	ga	
kaks	k i + a m + ks f	kaks	Hangul syllables are maintained under normalization. In earlier Unicode versions, jamo characters like ks f had compatibility mappings to k f + s f. These mappings were removed in Unicode 2.1.9 to ensure that Hangul syllables are maintained.

Note: The two tables above have a copyright of © 1998-2006 Unicode, Inc. All Rights Reserved.

Use Composed Forms for Single Glyphs

Many character sequences that correspond to a single glyph do not have composed forms. Even when normalized by form C, a single visual glyph or logical text element can be composed of multiple Unicode code points. For example, several characters used in writing Lithuanian have double diacritics, as they have only decomposed forms. An example is lowercase U with macron and tilde ("ũ", U+016b U+0303, where the first code point is a lowercase U with macron and the second is a combining acute accent).

Example

A relevant example can be found in NLS: Unicode Normalization Sample.

NLS: Unicode Normalization Sample

The sample application described in this topic demonstrates the representation of strings using Unicode normalization.

The sample application calls all four Unicode normalization forms with the same input string. A call is then made with invalid Unicode to demonstrate how the index of bad character code works. Finally the application passes a string that expands to be abnormally long, requiring multiple string normalization calls to get an appropriate buffer size.

This sample demonstrates the following NLS API functions:

```
IsNormalizedString  
NormalizeString
```

```
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
//  
// Copyright (c) Microsoft Corporation. All rights reserved.
```

Using Unicode Normalization to Represent Strings

```
// ===== Demonstration of Normalization APIs =====
#include "stdafx.h"
#include "windows.h"
#include <stdio.h>
#include <tchar.h>
#include "malloc.h"

// Print out a string using code points for the non-ASCII values
void DumpString(LPWSTR pInput)
{
    while (*pInput != 0)
    {
        if (*pInput < 0x80)
            wprintf(L"%c", *pInput);
        else
            wprintf(L"\\x%4.4x", *pInput);
        pInput++;
    }
    wprintf(L"\n");
}

// Check if normalized and display normalized output for a particular
normalization form
void TryNormalization(NORM_FORM form, LPWSTR strInput)
{
    // Test if the string is normalized
    if (IsNormalizedString(form, strInput, -1))
    {
        wprintf(L"Already normalized in this form\n");
    }
    else
    {
        // It was not normalized, so normalize it
        int    iSizeGuess;
        LPWSTR pBuffer;

        // How big is our buffer (quick guess, usually enough)
        iSizeGuess = NormalizeString(form, strInput, -1, NULL, 0);

        if (iSizeGuess == 0)
        {
            wprintf(L"Error %d checking for size\n", GetLastError());
        }

        while(iSizeGuess > 0)
        {
            pBuffer = (LPWSTR)malloc(iSizeGuess * sizeof(WCHAR));
            if (pBuffer)
            {
                // Normalize the string
            }
        }
    }
}
```

Using Unicode Normalization to Represent Strings

```
    int iActualSize = NormalizeString(form, strInput, -1, pBuffer,
iSizeGuess);
    iSizeGuess = 0;
    if (iActualSize <= 0 && GetLastError() != ERROR_SUCCESS)
    {
        // Error during normalization
        wprintf(L"Error %d during normalization\n", GetLastError());
        if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
        {
            // If the buffer is too small, try again with a bigger
buffer.
            wprintf(L"Insufficient buffer, new suggested buffer size
%d\n", -iActualSize);
            iSizeGuess = -iActualSize;
        }
        else if (GetLastError() == ERROR_NO_UNICODE_TRANSLATION)
        {
            wprintf(L"Invalid Unicode found at input character index
%d\n", -iActualSize);
        }
    }
    else
    {
        // Display the normalized string
        DumpString(pBuffer);
    }

    // Free the buffer
    free (pBuffer);
}
else
{
    wprintf(L"Error allocating buffer\n");
    iSizeGuess = 0;
}
}
}
```

```
int __cdecl wmain(int argc, WCHAR* argv[])
{
    // Tèst string t o nørmlize
    LPWSTR strInput = L"T\u00e8st string \uFF54\uFF4F n\u00f8rm\u00e4lize";

    wprintf(L"Comparison of Normalization Forms, input string::\n");
    DumpString(strInput);

    // Try it in the 4 forms
    wprintf(L"\n");
    wprintf(L"String in Form C:\n ");
    TryNormalization(NormalizationC, strInput);
}
```

Using Unicode Normalization to Represent Strings

```
wprintf(L"\n");
wprintf(L"String in Form KC:\n ");
TryNormalization(NormalizationKC, strInput);

wprintf(L"\n");
wprintf(L"String in Form D:\n ");
TryNormalization(NormalizationD, strInput);

wprintf(L"\n");
wprintf(L"String in Form KD:\n ");
TryNormalization(NormalizationKD, strInput);

// Note that invalid Unicode would show an error (illegal lone surrogate in
this case)
wprintf(L"\n");
wprintf(L"Attempt to normalize illegal lone surrogate:\n");
TryNormalization(NormalizationC, L"Bad surrogate is here: '\xd800'");

// Contrived strings can cause the initial size guess to be low
wprintf(L"\n");
wprintf(L"Attempt to normalize a string that expands beyond the initial
guess\n");
TryNormalization(NormalizationC,
    // These all expand to 2 characters
L"\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958"
L"\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958"
L"\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958"
L"\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958\u0958"
    // These all expand to 3 characters
L"\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c"
L"\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c\ufb2c");
}
```