

Wide-Character Part 2 Library Functions

USING WIDE CHARACTERS WILL NOT HANDLE BOTH ASCII AND UNICODE(wchar.h)

In C, handling strings is a common task. You've likely used functions like `strlen` to find the length of a string. For instance, given a character string:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello world!\n");
    char *pc = "Hello!";
    int iLength = strlen(pc);

    return 0;
}
```

`iLength` will be set to 6, representing the number of characters in the string.

Now, let's work with wide characters, which are typically used for multi-byte character encodings. You might define a pointer to a string of wide characters like this:

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

And then, you might try calling `strlen` again:

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

Here, problems arise. The C compiler may give you a warning about incompatible types. It will likely say something like:

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

The warning indicates that `strlen` is declared to accept a pointer to a char, but it's receiving a pointer to an unsigned short (a `wchar_t`).

When you run the program, you'll find that `iLength` is set to 1, not 6. What's happening here?

The 6 characters in the wide character string "Hello!" are stored as 16-bit values:

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

Memory-wise, on Intel processors, they are stored as:

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

`strlen` interprets the string as a collection of single-byte characters and assumes that the second byte (e.g., `0x00`) signifies the end of the string. As a result, it returns 1.

This exercise highlights the differences between the C language and its runtime library functions.

The C compiler interprets the string `L"Hello!"` as a collection of 16-bit short integers and properly handles array indexing and the `sizeof` operator.

However, runtime library functions like `strlen` are added during link time and expect single-byte character strings. When given wide-character strings, they don't behave as expected.

But fear not! You don't have to rewrite every C library function. Only functions that deal with string arguments need adjustments. Wide-character versions of these functions already exist.

For instance, the wide-character version of `strlen` is called `wcslen` ("wide-character string length"). It's declared in both `string.h` (where the regular `strlen` is declared) and `wchar.h`. The declarations look like this:

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

So, when you need to find the length of a wide-character string, you can call:

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

This function will return the correct value, such as 6 in this case.

Note that the character length of the string remains the same; only the byte length changes when moving to wide characters.

Most C runtime library functions with string arguments have wide-character counterparts. For example, `wprintf` is the wide-character version of `printf`.

These functions are declared both in `wchar.h` and in the header file where the regular function is declared.

By using wide-character functions, you can work effectively with wide-character strings in C.

This enables you to handle multi-byte character encodings like UTF-16 and UTF-32, which are crucial for internationalization and working with non-ASCII characters in your programs.

Here's a simple program that illustrates the concepts mentioned above:

above:

```
#include <stdio.h>
#include <wchar.h>
#include <string.h>

int main() {
    // Regular character string
    char *pc = "Hello!";
    int length1 = strlen(pc);

    // Wide character string
    wchar_t *pw = L"Hello!";
    int length2 = wcslen(pw);

    // Display the lengths
    printf("Character string length: %d\n", length1);
    printf("Wide character string length: %d\n", length2);

    return 0;
}
```

let's dive into the program in depth, breaking it down into paragraphs to provide a clear understanding of its functionality and concepts.

Introduction:

The program is a simple C application that illustrates the difference between character strings and wide character strings and introduces the use of wide-character library functions.

It showcases the fundamental principles of handling strings in both the regular character (single-byte) format and the wide character (multi-byte) format.

Character String Handling:

In the program's first part, it deals with regular character strings. A **character pointer pc** is defined, initialized with the string "Hello!" using double quotes.

The `strlen` function, which is a standard C library function, is then applied to `pc`. This function calculates the length of the string, counting the number of characters until it reaches the null-terminating character.

In this case, it returns the value 6, which represents the number of characters in the string "Hello!".

Wide Character String Introduction:

The program then transitions to wide character strings. A wide character pointer `pw` is declared, initialized with the string `L"Hello!"`.

The `L` before the string signifies that it is a wide character string. Wide character strings are used to handle multi-byte character encodings like UTF-16 and UTF-32, which are essential for dealing with non-ASCII characters and internationalization in C programs.

Wide Character String Handling:

The program proceeds to use the `wcslen` function to determine the length of the wide character string `pw`. **`wcslen`** is the wide-character counterpart of `strlen` and is used specifically for wide character strings.

It functions similarly to `strlen`, counting the number of characters until the null-terminating character is encountered. In this case, it returns the value 6, signifying the character length of the wide character string "Hello!".

Output and Comparison:

The program displays the results of both string length calculations using `printf`. It prints two lines of text, each displaying the length of a string. The first line shows the length of the regular character string, which is 6.

The second line reveals the length of the wide character string, which is also 6. This demonstrates that the character length of a string remains the same regardless of whether you are working with

regular or wide characters.

The primary distinction lies in the byte length, which differs when handling wide characters due to the encoding of multi-byte characters.

Conclusion:

This program serves as a basic introduction to handling character and wide character strings in C.

It highlights the significance of using wide-character library functions for multi-byte character encodings like UTF-16 and UTF-32.

Understanding these concepts is essential for internationalization and dealing with non-ASCII characters within C programs.

By using wide character strings and appropriate library functions, developers can work effectively with diverse character encodings, ensuring their programs are accessible and inclusive.

PROGRAMS THAT HANDLE BOTH ASCII AND UNICODE(tchar.h)

Introduction:

The notes begin by discussing the drawbacks of using Unicode in your C programs.

The primary concern is that Unicode encoding uses twice as much space as traditional ASCII encoding.

Additionally, the wide-character runtime library functions are larger than their regular character counterparts, increasing program size.

To address this, developers might consider maintaining two versions of a program, one with ASCII strings and the other with Unicode strings.

However, it is often preferable to maintain a single source code file that can be compiled for either ASCII or Unicode.

Using TCHAR.H:

One solution for maintaining a single source code file that supports both ASCII and Unicode is to use the TCHAR.H header file provided by Microsoft Visual C++.

While TCHAR.H is not part of the ANSI C standard, it offers a set of alternative names for runtime library functions that accept string parameters.

These alternative names, often referred to as "generic" function names, can work with either Unicode or non-Unicode versions of the functions.

The selection of whether to use Unicode or not is typically based on the presence or absence of the `_UNICODE` identifier.

Defining Generic Function Names:

In programs that use TCHAR.H and define the `_UNICODE` identifier, generic function names like `_tcslen` are defined to be equivalent to their wide-character counterparts.

For example, `_tcslen` is defined as `wcslen`. If the `_UNICODE` identifier is not defined, these generic function names are instead linked to their ASCII versions, such as `strlen`.

Using TCHAR Data Type:

TCHAR.H also introduces a new data type named TCHAR.

If the `_UNICODE` identifier is defined, TCHAR becomes equivalent to `wchar_t`.

Otherwise, it is a regular `char` data type.

This uniform data type simplifies handling character data within the program, abstracting away the differences between ASCII and Unicode encodings.

Handling String Literals:

One remaining challenge is handling string literals, particularly the "L" prefix for wide-character strings.

TCHAR.H defines a macro named `__T`, which uses a special syntax involving a "token paste" (`##`) to append the letter "L" to a string literal if `_UNICODE` is defined.

This macro, `__T`, effectively transforms `"Hello!"` to `L"Hello!"` when necessary.

Additionally, two other macros, `_T` and `_TEXT`, are defined to be the same as `__T`, allowing developers to choose their preferred level of conciseness when defining string literals.

Conclusion:

In summary, the use of TCHAR.H and its associated macros simplifies the process of maintaining a single source code file that supports both ASCII and Unicode encodings.

It provides a unified approach to handling character data and string literals, abstracting away the complexities of encoding.

This enables developers to create programs that are versatile and can easily adapt to different character encodings, ensuring compatibility and portability across diverse platforms and internationalization requirements.

Let's provide a comprehensive code example that incorporates the concepts discussed in the notes. We'll create a simple C program that utilizes TCHAR.H, generic function names, and demonstrates how to define and handle string literals for both ASCII and Unicode encodings.


```

int main() {
    // Define a TCHAR array to hold a string
    TCHAR str[] = _T("Hello, World!");

    // Use the _tprintf function to print the string
    _tprintf(_T("String: %s\n"), str);

    // Use generic function names like _tcslen to get the length of the string
    size_t length = _tcslen(str);
    _tprintf(_T("String Length: %d\n"), length);

    return 0;
}

```

In this code:

We include the necessary headers, including `<tchar.h>` to enable the use of `TCHAR.H` and generic function names.

We define a `TCHAR` array named `str` and initialize it with the string "Hello, World!" using the `_T` macro to handle different character encodings.

We use the `_tprintf` function, which is a generic function name for formatted output, to print the string. The `_T` macro is used to **ensure compatibility with both ASCII and Unicode encodings**.

We use the `_tcslen` function, another generic function name, to get the **length of the string**, and print the result.

This code demonstrates how to create a program that **can be compiled for either ASCII or Unicode encoding** by simply defining or not defining the `_UNICODE` identifier.

It also showcases the use of generic function names and the handling of string literals with the help of macros like `_T`.

These concepts allow you to maintain a single source code that works seamlessly with different character encodings.

So, USING WIDE CHARACTERS WILL NOT HANDLE BOTH ASCII AND UNICODE, BUT YOU CAN USE TCHAR TO MAKE A PROGRAM THAT CAN BE COMPILED FOR BOTH ASCII AND UNICODE, AM I RIGHT?

Yes, you are absolutely correct. Using wide characters alone will not handle both ASCII and Unicode.

To create a program that can be compiled for both ASCII and Unicode, you should use the **TCHAR.H** header and the **related macros and functions** it provides.

TCHAR.H, along with macros like `_T` and generic function names like `_tprintf` and `_tcslen`, allows you to write code that can **seamlessly switch between ASCII and Unicode** encodings.

By defining or not defining the `_UNICODE` identifier in your code, you can make your program compatible with different character encodings.

This approach makes it possible to maintain a single source code that can be compiled for both ASCII and Unicode, depending on your requirements.

MAINTAINING A SINGLE SOURCE FILE

There are, of course, certain **disadvantages to using Unicode**. First and foremost is that every string in your program will occupy **twice as much space**.

In addition, you'll observe that the functions in the wide-character run-time library are larger than the usual functions.

For this reason, you might want to **create two versions** of your program—one with ASCII strings and the other with Unicode strings.

However, the best solution is to maintain a single source code file that you can compile for either ASCII or Unicode.

However, maintaining a single source code that works for both ASCII and Unicode can be a challenge because the run-time library functions have different names and you're dealing with two character data types.

To address this issue, you can use the TCHAR.H header file included with Microsoft Visual C++. While it's not part of the ANSI C standard, it provides a set of alternative names for the normal run-time library functions that require string parameters, such as `_tprintf` and `_tcslen`.

These alternative names are often referred to as **"generic" function names** because they can work with either the **Unicode** or **non-Unicode** versions of the functions.

If you define an identifier named `_UNICODE` and include the `TCHAR.H` header file in your program, `_tcslen` is defined to be equivalent to `wcslen`, which is used for Unicode strings.

If `UNICODE` isn't defined, `_tcslen` is defined to be `strlen`, suitable for ASCII strings. This approach is applied to other functions as well.

To handle the two character data types, `TCHAR.H` introduces a new data type named `TCHAR`.

If `_UNICODE` is defined, `TCHAR` is defined as `wchar_t`, which is for wide characters.

Otherwise, `TCHAR` is simply defined as `char` for 8-bit characters.

Now, Let's address the issue of string literals.

If the `_UNICODE` identifier is defined, a macro called `__T` is defined like this:

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

This is somewhat obscure syntax, but it's part of the ANSI C standard for the C preprocessor.

It uses a token paste with two number signs (`##`) to **append the letter "L" to the macro parameter.**

For example, if the macro parameter is `"Hello!"`, then `L##x` becomes `L"Hello!"`.

If `_UNICODE` is not defined, the `__T` macro is simply defined as:
`#define __T(x) x.`

Additionally, two other macros, `_T` and `_TEXT`, are defined to be the

same as `__T`.

You can choose whether to use `_T` or `_TEXT` based on your preference.

Typically, you should define your string literals within the `_T` or `_TEXT` macro to indicate whether they consist of wide characters (if `_UNICODE` is defined) or 8-bit characters (if it's not).

Here's a concise example to illustrate these concepts:

```
#include <tchar.h>
#include <stdio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    _tprintf(_T("Hello, %s!\n"), _T("World"));
    return 0;
}
```

In this program, you use the generic function `_tprintf`, which can handle both ASCII and Unicode strings based on the `_UNICODE` identifier.

The `_T` macro is used to indicate that the string literals should be interpreted as wide characters when `_UNICODE` is defined, and as 8-bit characters when it's not.

This approach allows you to maintain a single source code that can be compiled for both ASCII and Unicode by toggling the `_UNICODE` identifier.

It simplifies the process of handling different character encodings in your program.

=====

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

=====

Let's expand on that part:

In the context of the `_T` and `_TEXT` macros used to handle both ASCII and Unicode strings, the `__T(x)` `L##x` syntax is employed within the `TCHAR.H` header file.

When `_UNICODE` is defined, this allows the macros to append an 'L' character in front of string literals, effectively converting them to wide character (Unicode) strings.

Here's a step-by-step explanation:

The `__T(x)` macro is defined as `L##x`. The `##` is a token-pasting operator in the C preprocessor, and it concatenates 'L' and the provided `x` into a single string.

If `_UNICODE` is defined in your source code, the `_T` and `_TEXT` macros are set to be equivalent to `__T(x)`.

This means that if `_UNICODE` is defined, string literals enclosed within `_T("text")` or `_TEXT("text")` will be transformed into wide character (Unicode) strings.

Here's a practical example to illustrate the usage of `_T` and `_TEXT`:

```
#include <tchar.h>
#include <stdio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    _tprintf(_T("Hello, %s!\n"), _T("World"));
    return 0;
}
```

In this example, the `_T` macro ensures that the string literals are correctly interpreted based on whether `_UNICODE` is defined or not.

When `_UNICODE` is defined, these literals are treated as wide character (Unicode) strings, and when it's not defined, they are

treated as 8-bit character strings.

This flexibility allows you to maintain a single source code for both ASCII and Unicode support in your program.

=====

But why are you not using define as a macro constant? why is it being used in the code ?

I understand your question. The `_T` and `_TEXT` macros are typically used in the code because they provide a way to switch between ASCII and Unicode string representations at the source code level, making it more flexible. These macros are often not directly implemented as `#define` macro definitions within the source code; instead, they are part of the system's header files, like `tchar.h`, which are included in the code.

Here's why this approach is beneficial:

Header File Implementation: `_T` and `_TEXT` macros are part of the standard Windows header file `tchar.h`. These header files are provided by the development environment or platform and are used to adapt your code for different configurations, including Unicode and non-Unicode builds.

Portability: By relying on the standard headers, your code remains more portable. You don't have to define these macros explicitly in your code, and you can switch between character encodings by changing the configuration settings or compiler flags (such as defining `_UNICODE`) without modifying your source code.

Consistency: Using system-defined macros like `_T` and `_TEXT` encourages consistency across different projects and promotes best practices for handling character encodings in Windows development.

Maintainability: It simplifies code maintenance. If you need to adapt your code for different character encodings or switch between ASCII and Unicode, you can do so by adjusting the project settings or compiler flags, rather than modifying every string literal in your code.

In essence, these macros are part of the wider framework provided by the development environment to enable multi-encoding support and

ensure code consistency. They are defined in system header files so that developers can rely on them to handle character encoding in a standardized and portable way.

=====

The ## is a token-pasting operator inside the C preprocessor, and it concatenates 'L' and the provided x into a single string. Ooh, that's the question i was asking, I thought that #def ## was for you to write yourself eg when am defining constants in C? but now i get it... it's already defined in the tchar.h header file, your work is to just apply it in your code, right?

Yes, you've got it! The ## operator is a built-in feature of the C preprocessor, and it's used to concatenate two tokens together into a single token. In the case of _T and _TEXT, these macros are pre-defined in header files like tchar.h.

Your job, as a developer, is to use these pre-defined macros in your code to handle string literals in a way that's consistent with your chosen character encoding (ASCII or Unicode).

So, you don't need to define _T and _TEXT yourself; they are provided by the development environment and associated header files like tchar.h.

Your role is to apply them to your code when you want to work with string literals and ensure that the appropriate macros are used for your chosen character encoding.

This approach makes it easier to write code that can be easily adapted to different character encodings without the need to manually change every string literal. It's a convenient and standardized way to handle character encodings in your Windows development projects.

=====

In this example, the _T macro ensures that the string literals are correctly interpreted based on whether _UNICODE is defined or not. How will the program know unicode is defined according to that statement?

The program will know whether _UNICODE is defined through compiler and project settings.

In most development environments, you can define or undefine `_UNICODE` through compiler flags or project settings.

When you define `_UNICODE` in your project settings or using compiler flags, it tells the compiler to use wide-character (Unicode) strings.

If `_UNICODE` is undefined, the program will use ASCII strings by default.

So, by adjusting the project or compiler settings, you can control whether `_UNICODE` is defined or not, which, in turn, determines how the `_T` macro behaves in your code.

This flexibility allows you to write code that can be easily adapted for either ASCII or Unicode character encodings without manually changing every string literal in your code.

=====

Are these still relevant in 2023?

Yes, wide character library functions are still useful in C programming as of 2023. They are used to work with strings that contain characters from different languages and character sets.

Wide character library functions are declared in the `wchar.h` header file and have names that start with the letter `w`. For example, **`wcslen()` is the wide character version of `strlen()`.**

Wide character library functions are very useful for working with internationalized applications.

By using wide character strings, ***you can ensure that your applications can support a wide range of languages and character sets.***

However, there are some things to keep in mind when using wide character library functions:

- Wide character library functions are **typically slower** than their single-byte counterparts.

- Wide character strings take up **more memory** than single-byte strings.
- Not all C library functions have wide character versions.

If you are working with ***internationalized applications***, it is recommended to use wide character library functions.

However, if you are working with ***performance-critical applications***, you should consider using single-byte strings instead.

Here are some examples of when you might use wide character library functions:

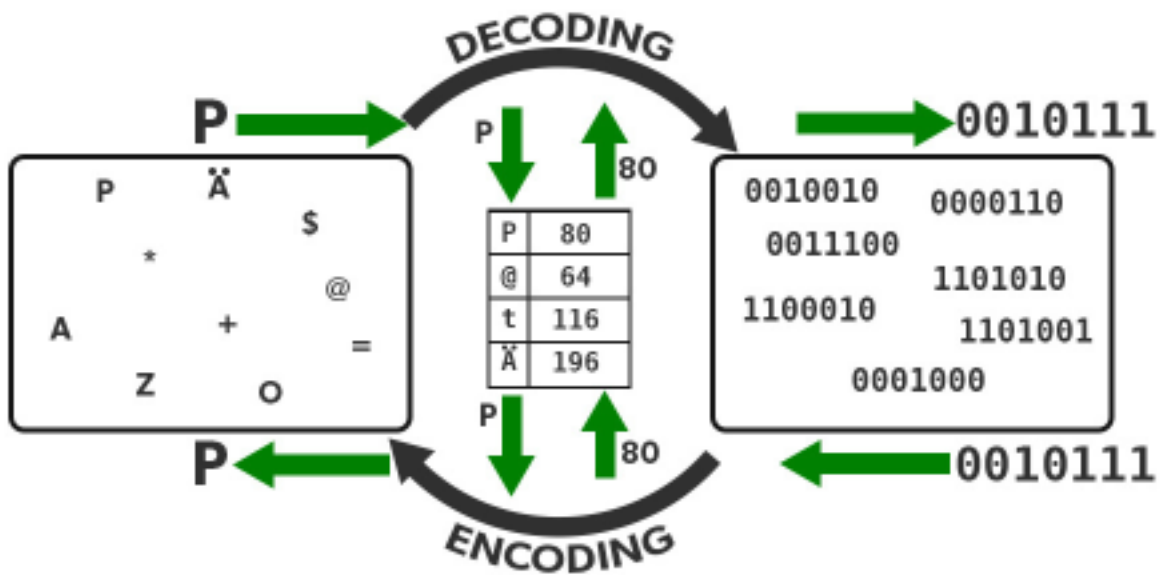
- You are developing a multilingual application.
- You are working with a database that contains characters from different languages.
- You are working with a text editor that needs to support different languages.
- You are working with a compiler that needs to support different languages.

Here are some examples of when you might not want to use wide character library functions:

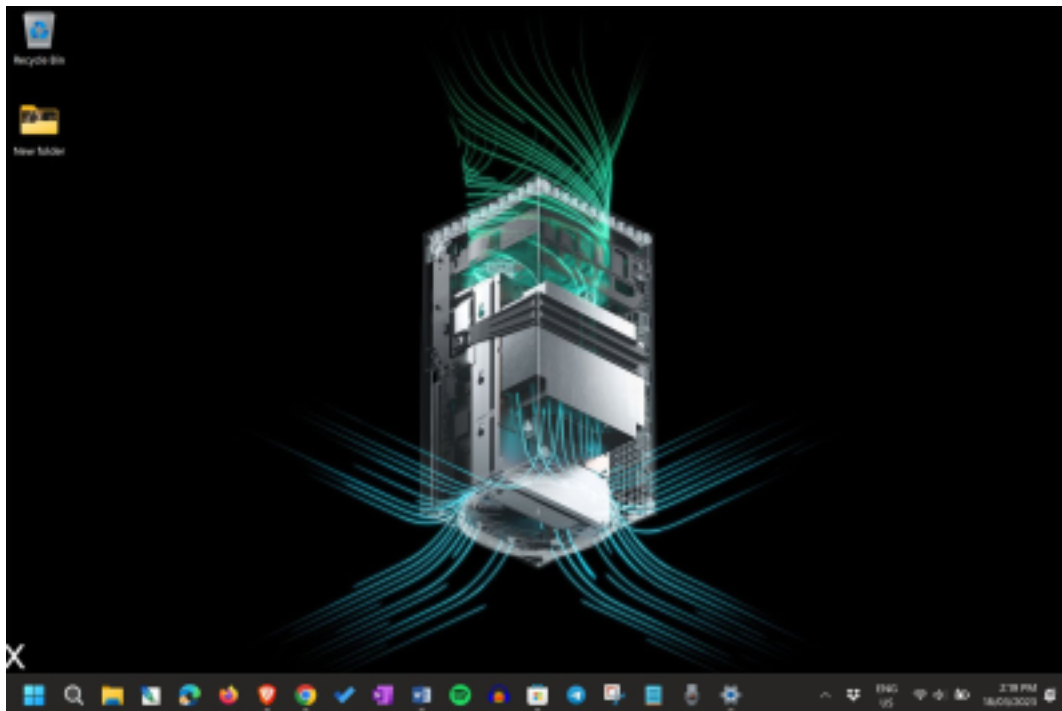
- You are developing a performance-critical application.
- You are working with a limited amount of memory.
- You are working with a library that does not have wide character versions of its functions.

WIDE CHARACTERS AND WINDOWS OPERATING SYSTEMS

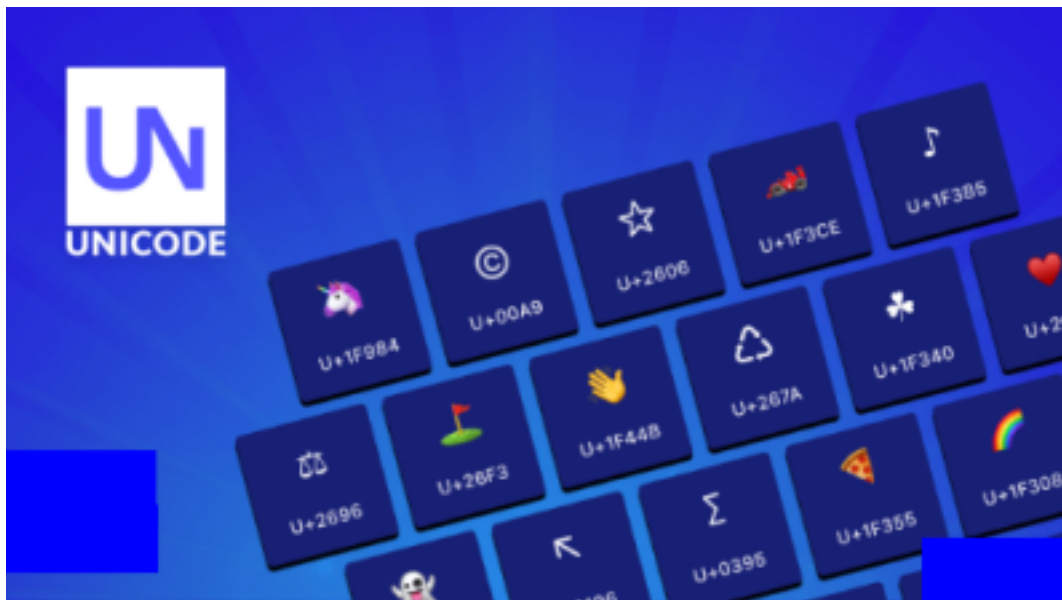
Windows 10 and 11 fully support Unicode, which is a character encoding standard that can represent characters from all major languages. This means that Windows internally uses character strings composed of 16-bit characters.



However, many programs are still written for ASCII, which is an older character encoding standard that can only represent 128 characters, mostly English letters and punctuation. When these programs run on Windows, Windows needs to convert their character strings from ASCII to Unicode.



Windows also supports programs that are written for Unicode. These programs can use the wide character versions of the Windows function calls. Wide character function calls have names that start with the letter w. For example, `wcslen()` is the wide character version of `strlen()`.



If you are developing a program that needs to run on both Windows and other operating systems, you should use the wide character versions of the Windows function calls. This will ensure that your program can support a wide range of languages and character sets.



Here are some examples of when you might want to use the wide character versions of the Windows function calls:

- You are developing a multilingual application.
- You are working with a database that contains characters from different languages.
- You are working with a text editor that needs to support different languages.
- You are working with a compiler that needs to support different languages.

Multilingual Testing in Software Testing



Here are some examples of when you might not want to use the wide character versions of the Windows function calls:

- You are developing a performance-critical application.
- You are working with a limited amount of memory.
- You are working with a library that does not have wide character versions of its functions.

Performance
Critical
Apps

