# Why do some Windows functions fail if I pass an unaligned Unicode string?

devblogs.microsoft.com/oldnewthing/20250605-00/?p=111250

June 5, 2025


A customer found that if they passed Unicode strings (which in Windows means strings encoded as UTF-16LE using the two-byte data type `wchar_t` as code units) which are not on even addresses, then some—but not all—functions fail to accept those strings. Why isn't this documented?

This is one of the ground rules for programming: Pointers must be properly aligned unless explicitly permitted otherwise.

In the C and C++ languages, forming an unaligned pointer is explicitly specified to return no useful value.

In C:

> (6.3.2.3 Pointers) If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.

In C++:

> [expr.static.cast](13) If the original pointer value represents the address A of a byte in memory and A does not satisfy the alignment requirement of T, then the resulting pointer value is unspecified.

Therefore, simply creating a misaligned pointer already takes you outside the world of allowable (in C) or at least meaningful (in C++) operations, so you shouldn't be surprised that using misaligned pointers results in nonsense.

As for why certain functions get more upset than others, it's all a matter how how those functions use the pointers and who detects the misaligned pointer.

If you are using a processor that is alignment-sensitive, you will probably get a failure when the code tries to read the data from that pointer. If the access is made in user mode, you will get an access violation exception, and the process will probably crash. If

the access is made in kernel mode, the kernel mode parameter validator will probably return an invalid parameter error. (Kernel mode must protect itself from user mode.)

If you are using a processor that forgives misaligned data accesses, then you may get away with it for a while, until the code does something with the data that requires alignment. For example, atomic operations typically require aligned data, even on processors that are normally forgiving of misalignment.

And even though x86-64 is generally alignment-forgiving, there are still places where it is alignment sensitive. For example, some instructions involving SIMD registers require alignment. SIMD registers are often used for copying blocks of memory around, and since `wchar_t` has 2-byte alignment, the `switch` statement for performing block copies has only 8 legal starting points out of 16, since all the odd addresses are invalid. If you pass an odd address, you might well fall through the `switch` statement and perform garbage copies.

The Microsoft C++ compiler has a special nonstandard keyword `__unaligned` for declaring that a pointer may be unaligned, and this tells the compiler that any accesses to the data behind that pointer must use instructions that are alignment-forgiving. For some processors, this can be quite expensive.

Limit your use of misaligned pointers to places where misaligned pointers are expressly permitted. You can tell where those places are by looking for the Windows SDK macro `UNALIGNED`. For example:

```
LWSTDAPI_(int)
    SHFormatDateTimeA(
        _In_ const FILETIME UNALIGNED * pft,
        _Inout_opt_ DWORD * pdwFlags,
        _Out_writes_(cchBuf) LPSTR pszBuf,
        UINT cchBuf);
```