What is __wchar_t (with the leading double underscores) and why am I getting errors about it?

devblogs.microsoft.com/oldnewthing/20161201-00

December 1, 2016

Raymond Chen

The Microsoft Visual C++ compiler has a compiler option called `/Zc:wchar_t` which lets you control what the symbol `wchar_t` means.

According to the C++ standard, `wchar_t` is a distinct native type, and that's what the Visual C++ compiler defaults to. However, you can set `/Zc:wchar_t-`, and that suppresses the intrinsic definition of `wchar_t`, allowing you to define it to whatever you like. And for Windows, this historically means

```
typedef unsigned short wchar_t;
```

because Windows predates the versions of the C and C++ standards that introduced `wchar_t` as a native type.

So now you have a problem if you are writing a library that will be consumed both by old-school code written with `wchar_t` defined as an alias for `unsigned short` and by new-school code written with `wchar_t` as an intrinsic type. What data type do you use for your string parameters?

Well, if your library uses C linkage, then you're in luck. Since the intrinsic `wchar_t` is a 16-bit unsigned integer in Visual C++, it is binary-compatible with `unsigned short`, so you can declare your function as accepting `wchar_t` in the header file, and each client will interpret it through their own `wchar_t`-colored glasses: Code that is wearing the `/Zc:wchar_t` glasses will see the native `wchar_t` Type. Code that is wearing the `/Zc:wchar_t-` glasses will see an `unsigned short`. And since C linkage is not decorated, you can export one function that accepts a `wchar_t`, and it will interoperate with either definition.

That works for undecorated functions, but what about languages like C++ that use decoration to encode the types of the parameters? Which decoration do you use?

Let's do both.

What you do is write two versions of your function, one that takes an `unsigned short` and one that takes a `__wchar_t`. That double-underscore version represents "The native type for `wchar_t` that is used by `/Zc:wchar_t`."

In other words, `/Zc:wchar_t` results in the compiler internally doing the equivalent of

```
typedef __wchar_t wchar_t;
```

It makes the symbol `wchar_t` an alias for the internal `__wchar_t` type.

So let's say you have a function called `DoSomething` that takes a wide string, and you want to accept clients compiled with either setting for `/Zc:wchar_t`.

```
// Something.h

bool DoSomething(const __wchar_t* s);
bool DoSomething(const unsigned short* s);
```

This declares two versions of the function. The first will be matched by code compiled with `/Zc:wchar_t`. The second will be matched by code compiled with `/Zc:wchar_t-`.

Your implementation goes like this:

```
// Something.cpp
#include <Something.h>

bool DoSomethingWorker(const wchar_t* s)
{
 ... implementation ...
}

bool DoSomething(const __wchar_t* s)
{
    return DoSomethingWorker(reinterpret_cast<const wchar_t*>(s));
}

bool DoSomething(const unsigned short* s)
{
    return DoSomethingWorker(reinterpret_cast<const wchar_t*>(s));
}
```

As noted earlier, callers who compile with `/Zc:wchar_t` will match the first version of `DoSomething`; callers who compile with `/Zc:wchar_t-` will match the second. But both of them funnel into a common implementation, which we declare with `wchar_t`, so that it matches the `/Zc:wchar_t` setting used by the library itself.

Okay, so to answer the opening question: `__wchar_t` is the name for the intrinsic data type for wide strings. If you compile with `/Zc:wchar_t`, then that's the data type that `wchar_t` maps to. The funny name exists so that code compiled with `/Zc:wchar_t-` can access it too,

and so that code which wants to be `/Zc:wchar_t` -agnostic can explicitly refer to the internal native type.

Raymond Chen

Follow