

Re: CFile::Read problem ???

Source:

<http://www.tech-archive.net/Archive/WindowsCE/microsoft.public.windowsce.embedded.vc/2004-02/0419.html>

From: Doug Cook (*dcook_at_microsoft.com*)

Date: 02/19/04

Date: Thu, 19 Feb 2004 04:10:27 GMT

| Another question, how does the compiler know whether '\0' is ASCII or
| UNICODE?

|

| I used to do:

|

| TCHAR t = _T('\0'); // or L'\0' ?

|

| But I guess that's not necessary?

No, it isn't. As far as the C compiler is concerned, almost any kind of zero will work. In fact, you can pretty much always assign a char to a TCHAR, in the same way that you can safely assign an int to a long. The C compiler is happy to let you do whatever you want with your data (though it will sometimes give warnings).

'\0' by itself is a char. L'\0' is wchar_t. However, the compiler is happy to automatically convert '\0' to L'\0'. It will also convert L'\0' to '\0', but it may give you a warning about possible loss of data. In fact, the compiler will also convert an integer 0:

```
TCHAR t = 0; /* Works just fine. Might give a warning on some  
compilers. */
```

To be really technical, it is a bit misleading to simply consider char as ASCII and wchar_t as Unicode. The only guarantee that the C language gives you is that a char is an integer that requires one unit of memory, and wchar_t is an integer that has enough range to represent one wide character. It is usually safe (for now) to assume that "one unit of memory" is a byte or 8 bits, and that "enough range to represent one wide character" is 2 bytes or 16 bits. But any assumption beyond that is asking for trouble.

The compiler treats values of all simple types as numbers (simple types exclude classes, structs, unions, and arrays). The compiler only looks at the type's size (number of bytes of memory) and base-type (float, signed integer, unsigned integer, or pointer). Any function or method

that appears to handle chars differently than ints is simply overloaded for 8-bit ints (chars) and normal ints (i.e. 32-bit ints). The compiler does have a special form 'A' for chars, but 'A' is nothing more than an easy way to write ((char)65). In the same way, L'A' is just a shortcut for ((wchar_t)65). Strings are also just shortcuts: "ABC" is a shortcut for a char[] with the value { 65, 66, 67, 0 }, and L"ABC" is short for the wchar_t[] value { 65, 66, 67, 0 }.

So while char and wchar_t are just integers, ASCII and Unicode are much more complicated ideas. ASCII is a 7-bit mapping from integers to symbols. We tend to store ASCII symbols in 8 bit chars, but that's really the only thing "ASCII" and "char" have to do with each other. We tend to use wchar_t when we store UTF16 text, but that is all "Unicode" and "wchar_t" have to do with each other. For example, chars can store non-ASCII symbols (anything with the 8th bit set), chars can be used to store symbols using encodings other than ASCII, and a wchar_t can store ASCII (with 9 bits wasted per character). In addition, an array of wchar_t can store Unicode text (using the UTF16 encoding we expect when we hear someone say "Unicode"), but an array of char can also store Unicode text (using the UTF8 encoding that is very popular nowadays for Internet text).

A final note is that while we're used to using 1 char for 1 ASCII symbol, it isn't safe to assume that 1 wchar_t corresponds to 1 Unicode "character" (actually, the official term is "code point", not "character"). Unicode has too many characters to fit in a wchar_t, so some Unicode characters need two wchar_t values for storage. In addition, some languages have multiple "characters" that combine to form a single symbol on the screen.

Probably more than you ever wanted to know, but I hope it is helpful!