What is the format of a double-null-terminated string with no strings?

devblogs.microsoft.com/oldnewthing/20091008-00

October 8, 2009

Raymond Chen

One of the data formats peculiar to Windows is the double-null-terminated string. If you have a bunch of strings and you want to build one of these elusive double-null-terminated strings out of it, it's no big deal.

H	e	l	l	o	\0	w	o	r	l	d	\0	\0

But what about the edge cases? What if you want to build a double-null-terminated string with no strings?

Let's step back and look at the double-null-terminated string with two strings in it. But I'm going to insert line breaks to highlight the structure.

H	e	l	l	o	\0
w	o	r	l	d	\0
\0					

Now I'm going to move the lines around.

Hello\0
world\0
\0

This alternate way of writing the double-null-terminated string is the secret. Instead of viewing the string as something terminated by two consecutive null terminators, let's view it as a list of null-terminated strings, with a zero-length string at the end. Alternatively, think of it as a packed array of null-terminated strings, with a zero-length string as the terminator.

This type of reinterpretation happens a lot in advanced mathematics. You have some classical definition of an object, and then you invent a new interpretation which agrees with the classical definition, but which gives you a different perspective on the system and even generalizes to cases the classical definition didn't handle.

For example, this "modern reinterpretation" of double-null-terminated strings provides another answer to a standard question:

How do I build a double-null-terminated string with an empty string as one of the strings in the list?

You can't, because the empty string is treated as the end of the list. It's the same reason why you can't put a null character inside a null-terminated string: The null character is treated as the terminator. And in a double-null-terminated string, an empty string is treated as the terminator.

One\0
\0
Three\0
\0

If you try to put a zero-length string in your list, you end up accidentally terminating it prematurely. Under the classical view, you can see the two consecutive null terminators: They come immediately after the `"One"`. Under the reinterpretation I propose, it's more obvious, because the zero-length string is itself the terminator.

If you're writing a helper class to manage double-null-terminated strings, make sure you watch out for these empty strings.

This reinterpretation of a double-null-terminated string as really a *list of strings with an empty string as the terminator* makes writing code to walk through a double-null-terminated string quite straightforward:

```
for (LPTSTR pszz = pszzStart; *pszz; pszz += lstrlen(pszz) + 1) {
 ... do something with pszz ...
}
```

Don't think about looking for the double null terminator. Instead, just view it as a list of strings, and you stop when you find a string of length zero.

This reinterpretation also makes it clear how you express a list with no strings in it at all: All you have is the zero-length string terminator.

```
 \0
```

Why do we even have double-null-terminated strings at all? Why not just pass an array of pointers to strings?

That would have worked, too, but it makes allocating and freeing the array more complicated, because the memory for the array and the component strings are now scattered about. (Compare absolute and self-relative security descriptors.) A double-null-terminated string occupies a single block of memory which can be allocated and freed at one time, which is very convenient when you have to serialize and deserialize. It also avoids questions like “Is it legal for two entries in the array to point to the same string?”

Keeping it in a single block of memory reduces the number of selectors necessary to represent the data in 16-bit Windows. (And this data representation was developed long before the 80386 processor even existed.) An array of pointers to 16 strings would require 17 selectors, if you used `GlobalAlloc` to allocate the memory: one for the array itself, and one for each string. Selectors were a scarce resource in 16-bit Windows; there were only 8192 of them available in the entire system. You don’t want to use 1% of your system’s entire allocation just to represent an array of 100 strings.

One convenience of double-null-terminated strings is that you can load one directly out of your resources with a single call to `LoadString`:

```
STRINGTABLE
BEGIN
 IDS_FILE_FILTER "Text files\0*.txt\0All files\0*.*\0"
END

TCHAR szFilter[80];
LoadString(hinst, IDS_FILE_FILTER, szFilter, 80);
```

This is very handy because it allows new filters to be added by simply changing a resource. If the filter were passed as an array of pointers to strings, you would probably put each string in a separate resource, and then the number of strings becomes more difficult to update.

But there is a gotcha in the above code, which we will look at next time.

Bonus Gotcha: Even though you may know how double-null terminated strings work, this doesn’t guarantee that the code you’re interfacing with understands it as well as you do. Consequently, you’d be best off putting the extra null terminator at the end if you are generating a double-null-terminated string, just in case the code you are calling expects the extra null terminator (even though it technically isn’t necessary). Example: The ANSI version of `CreateProcess` locates the end of the environment block by looking for two consecutive NULL bytes instead of looking for the empty string terminator.

Raymond Chen

Follow