What does the "l" in lstrcmp stand for?

devblogs.microsoft.com/oldnewthing/20110317-00

March 17, 2011

Raymond Chen

If you ask Michael Kaplan, he'd probably say that it stands for *lame*. In his article, Michael presents a nice chart of the various L-functions and their sort-of counterparts. There are other L-functions not on his list, not because he missed them, but because they don't have anything to do with characters or encodings. On the other hand, those other functions help shed light on the history of the L-functions. Those other functions are *lopen*, *lcreat*, *lread*, *lwrite*, *lclose*, and *llseek*. There are all L-version sort-of counterparts to *open*, *creat*, and *read*, *write*, *close*, and *lseek*. Note that we've already uncovered the answer to the unasked question "Why does llseek have two L's?" The first L is a prefix (whose meaning we will soon discover) and the second L comes from the function it's sort-of acting as the counterpart to. But what does the L stand for? Once you find those other L-functions, you'll see next door the H-functions *hread* and *hwrite*. As we learned a while back, being lucky is simply observing things you weren't planning to observe. We weren't expecting to find the H-functions, but there they were, and they blow the lid off the story. The H prefix in *hread* and *hwrite* stands for *huge*. Those two functions operated on so-called *huge pointers*, which is 16-bit jargon for pointers to memory blocks larger than 64KB. To increment your average 16:16 pointer by one byte, you increment the bottom 16 bits. But when the bottom 16 bits contain the value 0xFFFF, the increment rolls over, and where do you put the carry? If the pointer is a huge pointer, the convention is that the byte that comes after `S:0xFFFF` is `(S+__AHINCR):0x0000`, where `__AHINCR` is a special value exported by the Windows kernel. If you allocate memory larger than 64KB, the `GlobalAlloc` function breaks your allocation into 64KB chunks and arranges them so that incrementing the selector by `__AHINCR` takes you from one chunk to the next. Working backwards, then, the L prefix therefore stands for *long*. These functions explicitly accept far pointers, which makes them useful for 16-bit Windows programs since they are independent of the program's memory model. Unlike the L-functions, the standard library functions like `strcpy` and `read` operate on pointers whose size match the data model. If you write your program in the so-called *medium memory model*, then all data pointers default to *near* (i.e., they are 16-bit offsets into the default data segment), and all the C runtime functions operate on near pointers. This is a problem if you need to, say, read some data off the disk into a block of memory you allocated with `GlobalAlloc`: That memory is expressible only as a far pointer, but the *read* function accepts a near pointer. To the rescue comes the `lread` function,

which you can use to read from the disk into your far pointer. How did Windows decide which C runtime functions should have corresponding L-functions? They were the functions that Windows itself used internally, and which were exported as a courtesy. Okay, now let's go back to the Lame part. Michael Kaplan notes that the `lstrcmp` and `lstrcmpi` functions actually are sort-of counterparts to `strcoll` and `strcolli`. So why weren't these functions called `lstrcoll` and `lstrcolli` instead?

Because back when `lstrcmp` and `lstrcmpi` were being named, the `strcoll` and `strcolli` functions hadn't been invented yet! It's like asking, "Why did the parents of General Sir Michael Jackson give him the same name as the pop singer?" or "Why didn't they use the Space Shuttle to rescue the Apollo 13 astronauts?"

Raymond Chen

Follow