Why do some structures end with an array of size 1?

devblogs.microsoft.com/oldnewthing/20040826-00

August 26, 2004

Raymond Chen

Some Windows structures are variable-sized, beginning with a fixed header, followed by a variable-sized array. When these structures are declared, they often declare an array of size 1 where the variable-sized array should be. For example:

```
typedef struct _TOKEN_GROUPS {
    DWORD GroupCount;
    SID_AND_ATTRIBUTES Groups[ANYSIZE_ARRAY];
} TOKEN_GROUPS, *PTOKEN_GROUPS;
```

If you look in the header files, you'll see that ANYSIZE_ARRAY is #define'd to 1, so this declares a structure with a trailing array of size one.

With this declaration, you would allocate memory for one such variable-sized TOKEN_GROUPS structure like this:

```
PTOKEN_GROUPS TokenGroups =
   malloc(FIELD_OFFSET(TOKEN_GROUPS, Groups[NumberOfGroups]));
```

and you would initialize the structure like this:

```
TokenGroups->GroupCount = NumberOfGroups;
for (DWORD Index = 0; Index = NumberOfGroups; Index++) {
  TokenGroups->Groups[Index] = …;
}
```

Many people think it should have been declared like this:

```
typedef struct _TOKEN_GROUPS {
    DWORD GroupCount;
} TOKEN_GROUPS, *PTOKEN_GROUPS;
```

(In this article, code that is wrong or hypothetical will be italicized.)

The code that does the allocation would then go like this:

```
PTOKEN_GROUPS TokenGroups =
   malloc(sizeof(TOKEN_GROUPS) +
          NumberOfGroups * sizeof(SID_AND_ATTRIBUTES));
```

This alternative has two disadvantages, one cosmetic and one fatal.

First, the cosmetic disadvantage: It makes it harder to access the variable-sized data. Initializing the *TOKEN_GROUPS* just allocated would go like this:

```
TokenGroups->GroupCount = NumberOfGroups;
for (DWORD Index = 0; Index = NumberOfGroups; Index++) {
  ((SID_AND_ATTRIBUTES *)(TokenGroups + 1))[Index] = …;
}
```

The real disadvantage is fatal. The above code **crashes** on 64-bit Windows. The SID_AND_ATTRIBUTES structure looks like this:

```
typedef struct _SID_AND_ATTRIBUTES {
    PSID Sid;
    DWORD Attributes;
    } SID_AND_ATTRIBUTES, * PSID_AND_ATTRIBUTES;
```

Observe that the first member of this structure is a pointer, PSID. The SID_AND_ATTRIBUTES structure requires pointer alignment, which on 64-bit Windows is 8-byte alignment. On the other hand, the proposed *TOKEN_GROUPS* structure consists of just a DWORD and therefore requires only 4-byte alignment. *sizeof(TOKEN_GROUPS)* is four.

I hope you see where this is going.

Under the proposed structure definition, the array of SID_AND_ATTRIBUTES structures will **not** be placed on an 8-byte boundary but only on a 4-byte boundary. The necessary padding between the GroupCount and the first SID_AND_ATTRIBUTES is missing. The attempt to access the elements of the array will crash with a STATUS_DATATYPE_MISALIGNMENT exception.

Okay, you may say, then why not use a zero-length array instead of a 1-length array?

Because time travel has yet to be perfected.

Zero-length arrays did not become legal Standard C until 1999. Since Windows was around long before then, it could not take advantage of that functionality in the C language.

Raymond Chen

Follow