The GetRegionData function fails if the buffer is allocated on the stack. Is it allergic to stack memory or something?

devblogs.microsoft.com/oldnewthing/20190107-00

January 7, 2019

Raymond Chen

If you pass a `NULL` buffer to the `GetRegionData` function, the return value tells you the required size of the buffer in bytes. You can then allocate the necessary memory and call `GetRegionData` a second time.

```
DWORD bytesRequired = GetRegionData(hrgn, 0, NULL);
RGNDATA* data = (RGNDATA*)malloc(bytesRequired);
data->rdh.dwSize = sizeof(data->rdh);
DWORD bytesUsed = GetRegionData(hrgn, bytesRequired, data);
```

This version of the code works just fine. We call the `GetRegionData` function to obtain the number of bytes required, then allocate that many bytes, and then call `GetRegionData` again to get the bytes.

However, this version doesn't work:

```
struct REGIONSTUFF
{
   ...
   char buffer[USUALLY_ENOUGH];
   ...
};

REGIONSTUFF stuff;
DWORD bytesRequired = GetRegionData(hrgn, 0, NULL);
RGNDATA* data = (RGNDATA*)(bytesRequired > sizeof(stuff.buffer) ?
                           malloc(bytesRequired) : stuff.buffer);
data->rdh.dwSize = sizeof(data->rdh);
DWORD bytesUsed = GetRegionData(hrgn, bytesRequired, data);
```

The idea here is that we preallocate a stack buffer that profiling tells us is usually big enough to hold the desired data. If the required size fits in our preallocated stack buffer, then we use it. Otherwise, we allocate the buffer from the heap. (Related.)

This version works fine in the case where the number of bytes required is larger than our preallocated stack buffer, so that the actual buffer is on the heap.

But this version fails (returns zero) if we decide to use the preallocated stack buffer.

Is `GetRegionData` allergic to stack memory?

No. That's not the problem.

My psychic powers told me that the `...` at the start of `struct REGIONSTUFF` had a total size that was not a multiple of four. The `buffer` member therefore was at an address that was misaligned for a `RGNDATA`, causing the code to run afoul of one of the basic ground rules for programming:

> Pointers are properly aligned.

And indeed, it turns out that the members at the start of the structure did indeed have a total size that was not a multiple of four. Let's say it went like this:

```
struct REGIONSTUFF
{
   HGRN hrgn;
   char name[15];
   char buffer[USUALLY_ENOUGH];
};
```

To fix this, you need to align the `buffer` the same way as a `RGNDATA`. One way to do this is with a union.

```
struct REGIONSTUFF
{
   HGRN hrgn;
   char name[15];
   union {
     char buffer[USUALLY_ENOUGH];
     RGNDATA data;
   } u;
};

REGIONSTUFF stuff;
DWORD bytesRequired = GetRegionData(hrgn, 0, NULL);
RGNDATA* data = (RGNDATA*)(bytesRequired > sizeof(stuff.u.buffer) ?
                           malloc(bytesRequired) : stuff.u.buffer);
data->rdh.dwSize = sizeof(data->rdh);
DWORD bytesUsed = GetRegionData(hrgn, bytesRequired, data);
```

Another way is to use an alignment annotation. The appropriate annotation varies depending on which compiler you are using.

```
// Microsoft Visual C++
__declspec(align(__alignof(RGNDATA)))
char buffer[USUALLY_ENOUGH];

 // gcc
char buffer[USUALLY_ENOUGH]
__attribute__((aligned(__alignof__(RGNDATA))));

// C++11
alignas(RGNDATA)
char buffer[USUALLY_ENOUGH];
```

Raymond Chen

Follow