If you need anything other than natural alignment, you have to ask for it

devblogs.microsoft.com/oldnewthing/20071227-00

December 27, 2007

Raymond Chen

If you need variables to be aligned a particular way, you need to ask for it.

> Let's say I have the following code:
>
> ```
> void fn()
> {
>  int a;
>  char b;
>  long c;
>  char d[10];
> }
> ```
>
> What would the alignment of the starting adresses of a,b,c and d be?
>
> What would the alignment be if the memory were allocated on heap?
>
> If this alignment varies for different data types within the same translation unit, is there a way to force uniform alignment for all types?

If you need a particular alignment, you have to ask for it. By default, all you can count on is that variables are aligned according to their natural requirements.

First, of course, there is no guarantee that local variables even reside on the stack. The optimizer may very well decide that particular local variables can reside in registers, in which case it has no alignment at all!

There are a few ways to force a particular alignment. The one that fits the C language standard is to use a union:

```
union char_with_int_alignment {
 char ch;
 int Alignment;
} u;
```

Given this union, you can say `u.ch` to obtain a character whose alignment is suitable for an integer.

The Visual C++ compiler supports a declaration specifier to override the default alignment of a variable.

```
typedef struct __declspec(align(16)) _M128 {
    unsigned __int64 Low;
    __int64 High;
} M128, *PM128;
```

This structure consists of two eight-byte members. Without the `__declspec(align(#))` directive, the alignment of this structure would be 8-byte, since that is the alignment of the members with the most restrictive alignment. (Both `unsigned __int64` and `__int64` are naturally 8-byte-aligned.) But with the directive, the aligment is expanded to 16 bytes, which is more restrictive than what the structure normally would be. This particular structure is declared with more restrictive alignment because it is intended to be use to hold 128-bit values that will be used by the 128-bit XMM registers.

A third way to force alignment with the Visual C++ compiler is to use the `#pragma pack(#)` directive. (There is also a "push" variation of this pragma which remembers the previous ambient alignment, which can be restored by a "pop" directive. And the `/Zp#` directive allows you to specify this pragma from the compiler command line.) This directive specifies that members can be placed at alignments suitable for `#` -byte objects rather than their natural alignment requirements, if the natural alignment is more restrictive. For example, if you set the pack alignment to 2, then all objects that are bigger than two bytes will be aligned as if they were two-byte objects. This can cause 32-bit values and 64-bit values to become mis-aligned; it is assumed that you know what you're doing any can compensate accordingly.

For example, consider this structure whose natural alignment has been altered:

```
#pragma pack(1)
struct misaligned_members {
 WORD w;
 DWORD dw;
 BYTE b;
};
```

Given this structure, you cannot pass the address of the `dw` member to a function that expects a pointer to a `DWORD`, since the ground rules for programming specify that all pointers must be aligned unless unaligned pointers are explicitly permitted.

```
void ExpectsAlignedPointer(DWORD *pdw);
void UnalignedPointerOkay(UNALIGNED DWORD *pdw);
misaligned_members s;
ExpectsAlignedPointer(&s.dw); // wrong
UnalignedPointerOkay(&s.dw);  // okay
```

What about the member `w`? Is it aligned or not? Well, it depends.

If you allocate a single structure on the heap, then the `w` member is aligned, since heap allocations are always aligned in a manner suitable for any fundamental data type. (I vaguely recall some possible weirdness with 10-byte floating point values, but that's not relevant to the topic at hand.)

```
misaligned_members *p = (misaligned_members)
    HeapAllocate(hheap, 0, sizeof(misaligned_members));
```

Given this code fragment, the member `p->w` is aligned since the entire structure is suitably aligned, and therefore so too is `w`. If you allocated an array, however, things are different.

```
misaligned_members *p = (misaligned_members)
    HeapAllocate(hheap, 0, 2*sizeof(misaligned_members));
```

In this code fragment, `p[1].w` is not aligned because the entire `misaligned_members` structure is 2+4+1=7 bytes in size since the packing is set to 1. Therefore, the second structure begins at an unaligned offset relative to the start of the array.

One final issue is the expectations for alignment when using header files provided by an outside component. If you are writing a header file that will be consumed by others, and you require special alignment, you need to say so explicitly in your header file, because you don't control the code that will be including your header file. Furthermore, if your header file changes any compiler settings, you need to restore them before your header file is complete. If you don't follow this rule, then you create the situation where a program stops working if a program changes the order in which it includes seemingly-unrelated header files.

```
// this code works
#include <foo.h>
#include <bar.h>
// this code doesn't
#include <bar.h>
#include <foo.h>
```

The problem was that `bar.h` changed the default structure alignment and failed to return it to the original value before it was over. As a result, in the second case, the structure alignment for the `foo.h` header file got "infected" and no longer matched the structure alignment used by the `foo` library.

You can imagine an analogous scenario where deleting a header file can cause a program to stop working.

Therefore, if you're writing a header file that will be used by others, and you require nonstandard alignment for your structures, you should use this pattern to change the default alignment:

```
#include <pshpack1.h> // change alignment to 1
... stuff that assumes byte packing ...
#include <poppack.h>  // return to original alignment
```

In this way, you "leave things the way you found them" and avoid the mysterious infection scenarios described above.

Raymond Chen

Follow