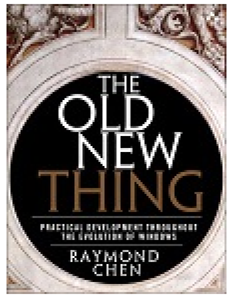
How to compress out interior padding in a std::pair and why you don’t want to

devblogs.microsoft.com/oldnewthing/20240731-00

July 31, 2024



Raymond Chen

My survey of many popular STL types began with `std::pair`, and in the comments, Jan Ringoš noted that the layout of a `std::pair` could result in padding between the two elements that could be recovered from padding within one of the elements.

```
struct bulky
{
    uint16_t a;
    void* b;
};
```

If you assume 64-bit pointers and natural alignment, then the `bulky` structure contains 2 bytes for `a`, then 6 bytes of padding to reach the next aligned pointer boundary, and then 8 bytes for `b`.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
a								b							

This padding is unavoidable for `bulky` on its own, since you need to get `b` aligned on a pointer boundary, and the entire structure needs to be 8-byte aligned so that you can have an array of `bulky` objects.

But what if we put it inside a `std::pair<lithe, bulky>`, where `lithe` is something like this:

```
struct lithe
{
    uint16_t v;
};

std::pair<lithe, bulky> p;
```

This produces the following `std::pair`:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	1
v								a								b				
first								second												

In order to ensure that the `bulky` starts on a pointer-aligned boundary, there are 6 bytes of padding after the `first`'s `uint16_t`, bringing the total size of the pair to 24 bytes. On the other hand, if we represented the same data in a single structure:

```
struct pair_lithe_with_bulky
{
    uint16_t v;
    uint16_t a;
    void* b;
};
```

then the result would be much smaller since we could coalesce the padding.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
v		a						b							
first		quasi-second													

Could there be a way to tell C++, “Hey, I know you need to put padding in this `bulky` structure, but if this structure is a subobject of another structure, just lay out the members as if they were all part of one big structure”?

I mean, you could propose anything.

The problem with this idea is that if you try to access `p.second`, you don't get a normal `bulky`, but rather a wacked-out version of `bulky` that has a *misaligned* pointer, and whose size is not a multiple of its alignment.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D
a						b							
quasi-second													

This bizarro-world version of `bulky` cannot be put in an array since it would misalign the subsequent element. And since its layout and size aren't the same as that of a normal `bulky`, its type wouldn't be the same as that of a normal `bulky`. It would have to be a "misaligned by 2 `bulky`" (which would need to have a name like `[[header_offset(2)]] bulky`). Now you have to decide what `std::pair<lithe, bulky>::second_type` is. Is it `bulky`? Or is it `[[header_offset(2)]] bulky`? Either choice you make is going to create confusion, because they will cause one or the other of the following to be false:

```
// given p of type std::pair<T, U>
std::is_same_v<U, std::pair<T, U>::second_type>>
std::is_same_v<U, decltype(p.second)>
```

But wait, there's still more to fret over.

Consider this slightly fatter version of `lithe`:

```
struct medium
{
    uint32_t v;
    uint16_t w;
};
```

The normal layout of the `std::pair` would be this:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	1
v				w				a								b				
first								second												

If you try to squeeze out the padding, you end up with this quite compact structure:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
v				w		a		b							
first						second									

This is even more confusing because the embedded `medium` structure is also not a normal `medium` structure: It lacks the trail padding and has a size that is not a multiple of its alignment. The compiler cannot optimize

```
extern medium special;
p.first = special; // make the first part special
```

to a `memcpy(&p.first, &special, sizeof(medium))` because that would accidentally overwrite the `a` hiding inside the trail padding. It would have to be `memcpy(&p.first, &special, sizeof(medium) - trail_padding_size)` to avoid overwriting data hiding in the trail padding.

You might think to solve the need for a `header_offset` attribute for alternate layouts by putting the `lithe` *inside* the padding of the `bulky`, assuming it fits.

		first													
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
a		v						b							
sec…								…ond							

Great! Now, all the structures are normal-sized and normally-aligned.

Now you have a problem with this:

```
p.second = bulky{}; // clear out the second part
```

In order for this to work, the compiler cannot optimize the assignment to a `memcpy` because that would accidentally overwrite the `first` embedded in the internal padding.

If you are really keen on squeezing out the padding, you can do it by setting custom packing for the `bulky` structure.

```
#include <pshpack4.h>
struct bulky
{
    uint16_t a;
    void* b;
};
#include <poppack.h>
```

Overriding normal packing to 4-byte alignment means that you get this layout for bulky, which matches the “quasi-`second`” we discovered earlier.

00	01	02	03	04	05	06	07	08	09	0A	0B
a				b							

Hooray! This pairs nicely with `lithe`:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
v				a				b							
first				quasi-`second`											

The downside of this is that systems that are alignment sensitive will have to load the `b` as an unaligned value, which tends to be rather expensive. It’s not quite as bad as byte alignment, since we can often load it in just two steps instead of eight, but it’s still worse than a straight load.

In general, this sort of tight memory optimization does save you memory, but it costs you in code flexibility (a vector of `bulky` objects is not going to be fun), and it can cost you in runtime costs (on alignment-sensitive platforms).

Bonus chatter: The introduction of `[[no_unique_address]]` in C++20 makes things more complicated. Base classes and members with the `[[no_unique_address]]` attribute are permitted to overlap. A common use for this is to extend the so-called empty base optimization to empty members, thereby avoiding the need for the complex dance employed by compressed pairs.

But another use for `[[no_unique_address]]` is to allow overlap between non-empty objects, specifically, to allow one type to hide inside the padding of another. In practice, compilers that take advantage of it[2] limit themselves to reusing tail padding, so that they can still use `memcpy` to assign two objects (just with a smaller object size).

In other words, it is legal for a compiler to do this:

```
struct part1
{
    void* ptr;
    int16_t a;
};

struct part2
{
    int32_t b;
};

struct combined
{
    [[no_unique_address]] part1 p1;
    [[no_unique_address]] part2 p2;
};
```

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
ptr								a				b			
												p2			
p1															

To avoid damaging any data hiding in the tail padding, copying a `part1` copies only 10 bytes instead of 16. This is not too heavy a burden on the compiler, since avoiding tail padding is easier than avoiding internal padding.

[2] The Microsoft Visual C++ compiler does not take advantage of `[[no_unique_address]]` for ABI compatibility reasons. You have to say `[[msvc::no_unique_address]]` with the understanding that you are accepting the ABI break and promise not to mix code compiled in C++17 mode with code compiled in C++20 mode.